Excel VBA という諸刃の剣を真っすぐに扱うために

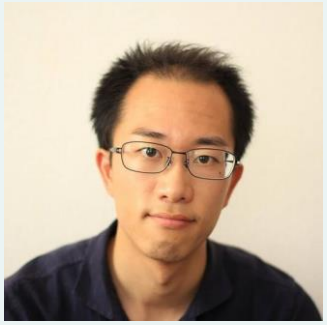
2018年 8月

那須野 拓実

このスライドでは、

エンジニアではないけれど
現場から業務改善をしたくて
できるところから小さくPDCAを回そうとして
Excel VBA（マクロ）をかじってみた人

を、主な読者と想定して、
飛躍のきっかけになるかもしれない発想をまとめました。
少しでも参考になれば幸いです。



那須野 拓実 （なすの たくみ）

http://takuminasuno.com/ja/

たなぐら応援大使（福島県棚倉町）。
トリプレッソを勝手に応援する人。
ネイチャーフォト中心の多言語ブログを書いてます。

本業はナレッジマネジメントとかデータ分析とかの何でも屋ですが、
今は半年間の育児休業中。

コンタクトはこちらからどうぞ！ ⇒ takumi_nasuno

※上記の自己紹介は、2018年8月執筆当時のものです。

なお、本スライドに出てくる挿絵的写真は
応援大使をしている福島県棚倉町で撮影したものです

第1章　Excel VBAをどう捉えるか

第2章　キーボードタッチを武器にする

第3章　Excel VBAコーディングガイドライン

第4章　ループ処理を考える

第5章　ファイル処理を考える


第1章
Excel VBAをどう捉えるか
© Takumi Nasuno

Excel VBAは 諸刃の剣

○　ほとんどのビジネス環境で
一般社員権限にて業務改善に即座に挑戦できる
プロトタイピングツールである。

○　頑張ればExcel処理だけでなく、
PowerPoint や Word、テキスト、メール、
DBなど、あらゆるものと連携できる。

×　個人で簡単に始められるがゆえに
保守体制が伴わないまま継続することが多く、
長期的に運用リスクとなりやすい。

Excel VBAは
誰でも手軽に
業務効率を 2桁 上げられる

でも

うまくツールができあがって
組織利用が始まるのなら
早々に対策をとるべき

組織利用が始まったら検討すべき対策例

(1) 業務効率化のインパクトを評価できるようにする

何事も、組織を動かす場合には組織へのインパクトを評価できることが望ましいです。

定量的には、ツールを使い始める前にどんな作業がどれくらいの頻度で、どれくらいの時間かかっていたか、何人がしていたか等を計測し、ツールを使い始めたことでそれがどう変わったかを計測して、差分を評価する方法が一般的です。それ以外にも定性的には、ツールを使う人に事前事後のヒアリングを行うことで、ツールの影響を感情面も含めて明文化し、組織内での評価材料に使う方法もあります。

(2) 複数人による保守体制を作る

実務上、自分以外が書き換えができない状態は往々にしてあり、最悪の場合は誰もExcel VBAを知りすらしない可能性もあります。
そういった場合、開発者（自分）が休職したり退職したときに、運用が破綻するリスクがあります。破綻の仕方は主に、
①運用変更のときにツールを改修できずに使えなくなる、②原因不明のバグが発生して誰も直せずに使えなくなる、
③WindowsやExcelの更新で一部コードが機能不全に陥って使えなくなる、の3パターンがあり、意図せず起こりうる点に留意が必要です。

Excel VBAといえど初見のツールを改修するのは相当な熟練度が要されるので、正式に複数人による保守体制を作るか、アウトソースするか、必要に応じて、組織内でのExcel VBA普及活動をしたり、中のコードを複数人に共有したりしておくことが望ましいです。

(3) 現行ツールの期限を設定し、他言語へのシステム移行を計画する

Excel VBAで作ったツールは、本来は他のプログラミング言語（システム）で作るべきものである可能性が高いです。

とりあえずExcel VBAで作るのは立ち上げやすさ、ひとまずの完成までの速さ、PDCAサイクルの速さ的には優秀ですが、
長期的な目で見ると、社内インフラとして、しかるべきプロセスを踏んで、しかるべき投資をすべきだと思います。
あくまでExcel VBAはプロトタイピングツールと考えて、半年ないし長くても3年程度を目途に、システム移行の計画を吟味します。

リスクに向き合って
うまく使っていきましょう


第2章
キーボードタッチを武器にする
© Takumi Nasuno

Excel VBAは
下手に勉強するよりも
キーボード入力を鍛えた方が
業務効率はアップします

効率化のためにやるのに
書くのに時間がかかったら
元も子もないです

“マウスは なるべく使わない”
まずはこれ

いくつか 便利な技を
見ていきます

前提として開発環境はWindowsを想定しており、これから紹介するショートカットキーはすべてWindows仕様となっている。Excel VBAを書くのにiOS環境を使う人はほとんどいないと思うが、念のためのお断りを申し上げておくことにする。

ウィンドウ処理

Excelからエディターを開く	Alt + F11
エディターを閉じてExcelに戻る	Alt + F4
左のウィンドウに移る	Alt + Tab
右のウィンドウに移る	Alt + Shift + Tab
ウィンドウを最大化する	Windows + ↑
ウィンドウを左に移動する	Windows + ←
ウィンドウを右に移動する	Windows + →
デスクトップを表示する	Windows + M

エディターのカーソル処理

次のプロシージャに進む	Ctrl + ↓
前のプロシージャに戻る	Ctrl + ↑
モジュールの一番下に進む	Fn + Ctrl + → もしくは Fn + End
モジュールの一番上に戻る	Fn + Ctrl + ← もしくは Fn + Home
行のテキスト右端に進む	Fn + → もしくは End
行のテキスト左端に戻る	Fn + ← もしくは Home
行の左端に戻る	Fn + ←2回
行の次の単語に進む	Ctrl + →
行の前の単語に戻る	Ctrl + ←
行のテキストを全選択する	(Fn + →) の後に (Shift + Fn + ←)

エディターの機能のショートカットキー

元に戻す	Ctrl + Z
元に戻す、の取り消し	Alt を押して、E を押した後に R
予測検索する（自動メンバー表示）	Ctrl + Space もしくは Ctrl + J
プロジェクトエクスプローラー(左窓)にフォーカスを移す	Ctrl + R
プロパティウィンドウにフォーカスを移す	F4
コード (普通のエディター部分)にフォーカスを移す	F7
カーソルのある関数を実行する	F5
実行中の関数を強制的に一時停止する	Esc
一時停止中の関数を再開する	F5
1行だけ実行する（ステップイン）	F8

とにかく 練習あるのみ


第3章
Excel VBA
コーディングガイドライン
©Takumi Nasuno

効率的に書きつつ

人にも共有しやすくするには

ガイドラインを作るとよいです

ということで
マイルールを紹介します

マイルールとはいっても一般的なものが多く、ネットで検索してヒットするようなExcel VBAガイドラインとの共通点が多数あると思われる。Excel VBAはノンエンジニアが通常のExcel業務の延長で扱うことが多く、結果的に一般的なコーディングのお作法によらずに書かれることが多いが、他人が見ることや、遠い将来に自分が見返す可能性を考えたら、整った書き方を心掛けたほうが良いのは言うまでもない。また綺麗に書く方が、結果的に書くスピードが速くなる・・・はず。

1. 命名規則を揃える

キャメルケース

基本は小文字で、単語の冒頭だけを大文字で書く。ただし、最初の単語だけは冒頭を小文字にする。ローワーキャメルケースとも呼ぶ。

通常の変数名に使用する。つまりこれが基本。

lastRow

パスカルケース

基本は小文字で、単語の冒頭だけ大文字で書く。キャメルケースと違って最初の単語も冒頭を大文字。アッパーキャメルケースとも呼ぶ。

クラス名やメソッド名（関数名）に使用する。

LastRow

アッパーケース

すべて大文字とし、単語の間を半角アンダースコア（_）で繋げて書く。

あまり推奨されないグローバル変数をどうしても使いたいときに、注意喚起の意味を込めてアッパーケースを使用する。

LAST_ROW

なお、これ以外の書き方として、スネークケースとハンガリアン記法がよく挙げられる。スネークケースはアッパーケースがすべて小文字になったもの（last_row）、ハンガリアン記法はパスカルケースの頭に型の略称を付けたもの（lngLastRow）である。VBAは定義済み変数の入浴時に大文字小文字補正がかかるため、キャメル系（キャメルケースとパスカルケース）同士は共存させやすく、スネーク系列（スネークケースとアッパーケース）同士は共存させやすいが、キャメルケースとスネークケースのように系列を超えると意図的に文字列を変えないといけないため、記述が面倒になる。そのため、VBAコーディングではキャメル系列を基本としつつ、スコープが全方位に及ぶ要注意のグローバル変数のみ例外的にアッパーケースを使うようにしている。あと余談だが、モジュールレベル変数はスコープの種類が増えて管理が面倒なので、個人的には使わないことにしている。

2. 変数は他人が見ても分かる名前を付ける

(1) 変数名に日本語は使わない

可読性が落ちます。あとクールじゃないです。以上。

(2) 変数名に省略文字は使わない

省略文字を使うと、他人が見たときに理解しづらくなり、保守性を損います。というか半年後の自分が理解できません。
wb とか sh とか cnt とかはまだ分からなくもないですが、tgtb とか crrsh とかだともはや謎になります。
sheetCount や targetBook、currentSheet など、音読して読めるかどうかを基準に命名します。多少長くても問題ないです。

(3) 配列かどうかは複数形の s で区別する

変数が配列かどうかが分かるように、配列には複数形の s を付け、逆に配列でないものは s を付けないようにします。
例えば、複数のワークシートオブジェクトが格納される変数を sheets とし、For Each で格納する変数を sheet とする等。
そうすると、パッと見で要素番号を指定すべきものかどうかを区別でき、可読性が上がります。

(4) 慣例的な変数名があるものは素直に使う

よくあるところだと、一時的な Variant を放り込む temp、一時的なStringを放り込む buf、Objectを放り込む obj、
FileSystemObjectの fso、RegExpの re などはよく使われるので、素直に使います。あと、ループのカウント用の変数は i が原則ですが、
複数のループがあって区別したい時は iSheet や iRow などのようにループ対象を明記した変数を別途作成します。

3. 変数はしっかり定義する

(1) 変数宣言は強制する

未定義の変数があるとアラートを出してくれるので、想定外の変数がタイプミスで生まれることがなくなり、誤作動のリスクが減ります。

(2) 変数の型は明確に定義する

型を省略するとVariant型になり、どんなデータでも入るため、想定外の値が入って誤作動を起こす可能性が出てきます。
そういうリスクを少しでも減らすために、また可読性を上げるために、型は必ず定義します。
よく使う型は、Workbook, Worksheet, Range, Object, String, Long, Currency, Variant あたりです。

また、複数の変数を同時に定義するときに型を忘れがちなのは要注意。
例えば「Dim iSheet, sheetCount As Long」はダメなので、「Dim iSheet as Long, sheetCount As Long」のように書きます。

(3) 変数は初期値を明示的に代入する

カウント用のループ変数をインクリメント（+1）するときとか、文字列をカンマ区切りで繋げるときとかの場合に、
初期値が empty でもループ内の代入時に 0 ないし “” 扱いされるのをいいことに、初期値を設定せずに処理をしがちです。
このあたりで手を抜くと可読性が下がるので、明確に定義をします。

(4) 配列の添え字は最小値を明示的に定義する

添え字を省略しても 0 から始まりますが、可読性が落ちます。0 でも明示的に「Dim targetNames(0 to 3) As String」のように定義します。

4. 綺麗に書く

(1) タブと空白行で構造を可視化する

いかに変数の命名規則を揃えて誰でも分かる名前を付けて、しっかり定義したとしても、ビジュアル的にごちゃごちゃだと読めません。
適切にタブを使って If / End If や For / Next の対応が分かるように書いて階層を表現し、
一連の処理の塊を数行単位でまとめて空白行で離してコードを意味の単位で理解しやすいようにします。

(2) なるべく階層が浅くなるように書く

上述のようにタブをしっかりと書いて階層を表現する場合、
何度も If を重ねたり、多重ループを書こうとすると、やけに階層が深くなることがあります。とても読みにくいです。
Boolean型の変数で条件管理をしたり、Exit を設定したりすれば浅く書けることが多いので、深いときは速やかに見直します。

(3) 変数名は小文字で書いて、大文字小文字補正を活用する

定義済み変数を書くと、大文字小文字が定義されたとおりに補正される特性を利用して、
すべて小文字で書くようにすると、書き終わってフォーカスをずらしたときに、大文字であるべきところが大文字にパッと変わるので、
定義済み変数であること、つまり誤字がないことが直観的に分かります。わざわざ Shift ボタンで大文字にしなくてよいので、
ダブルの意味でサクサクとコードを書くことができます。

(4) マクロの記録は使わない

言わずもがなですが、マクロの記録は不要なコードが多く、変なコーディングの癖がつくので、使いません。

5. そのほか雑多なマイルール

(1) 画面更新とアラートは止める

処理の高速化ならびにシート削除などで煩わしいアラートが出るのを避けるため、「Application.ScreenUpdating = False」 と
「Application.DisplayAlerts = False」を実行関数の直後に入れます。また、実行関数終了時は忘れず True に戻します。

(2) コピーメソッドは使わない

セルのコピーは色々な手間を省きやすいしマクロの記録で登場するために使われがちですが、
OSのクリップボードが上書き/初期化されるため、ほかに平行作業をしていたときに意図せぬ挙動が起きることがあります。
なので、どうしても使わざるを得ないことがない限りは、他の書き方で代用します。

(3) ウィンドウ枠の固定は、全画面表示にしてから

いい感じの表を作りたいと思ったらウィンドウ枠の固定（ActiveWindow.FreezePane = True）を考えることがありますが、ここに落とし穴。
ウィンドウ枠が狭くて選択範囲が画面に表示されていない場合、変なところでウィンドウ枠が固定されてしまいます。
作業の流れによってはウィンドウが小さい状態で実行される可能性もあるので、事前にウィンドウサイズを調整しておきます。

(4) Do構文のループは、最初にインクリメント（+1）処理を書く

ループ処理のシンプルさは For Each > For > Do なわけですが、
Do はループを出るためのインクリメント（+1）処理を忘れると無限ループに陥り、最悪の場合にExcelが落ちます。
どうしても Do を使いたい場合は、内部処理を書く前にインクリメント処理を書き、ループを出られるようにしてからにします。

普段から
意識していきましょう

第4章
ループ処理を考える

効果的なコードを書くには
作業の構造化が必須

やりたい作業の中に

どんな **繰り返し** があるか

構造化して考える

よく使う 繰り返し(ループ) といえば

ループをどう分類するかはいくつかの見方がある。このスライドでは直観的な分かりやすさを重視して、どんな対象物をループするかで区分けしている。コーディング上の処理による区分けを考えると、①ForEach構文によるループ、②For構文によるLboundからUboundまでのループ、③For構文による0ないし1個目から個数分までのループ、④ループ変数で管理するDo構文によるループ、の4つに大別されると思われる。①と②はもれなくループしていることがコードを見てすぐに理解できるが、③と④は上限を間違えたり、ループを出る条件がおかしかったりする可能性が否定できないため、注意が必要になってくる。極力①ないし②で書くことが望まれる。

行のループ

```
'行のループ（For構文）
Public Sub LoopRowsByFor()

    'ループ用の変数を定義
    Dim iRow As Long
    Dim firstRow As Long
    Dim lastRow As Long

    '開始行と終了行を設定
    firstRow = 2
    lastRow = 100

    'For構文による行のループ
    For iRow = firstRow To lastRow

        'A列に行番号を記述する
        Cells(iRow, 1) = "行" & iRow

    Next

End Sub
```

	A
1	
2	行2
3	行3
4	行4
5	行5
6	行6
7	行7
8	行8
9	行9
10	行10
11	行11
12	行12

- 行のループは、ワークシートの縦の方向に、ルールに従って処理を繰り返す書き方。
- ループ自体はFor構文がシンプルなので推奨。ループ用の変数（ループ変数）は、「i」の後に「Row」を付けて「iRow」とし、行のループであることを分かりやすくする。
- ループを始める行（開始行）と終える行（終了行）はFor構文に直接書くとマジックナンバーと化してしまうので、ループの外で明示的に変数として定義してから使うことにする。
- もし一定条件のもとでループを出たい場合は、「Exit For」を書けば出ることができる。

どれでも同じように、行のループができる

```
'For構文による行のループ
For iRow = firstRow To lastRow

    'A列に行番号を記述する
    Cells(iRow, 1) = "行" & iRow

Next
```

○ シンプルで使いやすい。ループの基本。

○ Stepを使ってとびとびに処理したり、下から上に処理したりできる。

```
'Do構文による行のループ
iRow = firstRow
Do While iRow <= lastRow

    'A列に行番号を記述する
    Cells(iRow, 1) = "行" & iRow

    '一つ下の行に進む
    iRow = iRow + 1
Loop
```

○ ループ変数の増やし方を変えたり、終了行を途中で更新したりなど、トリッキーな処理が可能。

× 明示的にループ変数（iRow）のカウントを増やす必要がある。（忘れると、もれなく無限ループが発生する。）

```
'For Each構文による行のループ
Dim cell As Range, targetCells As Range
Set targetCells = Range(Cells(firstRow, 1), Cells(lastRow, 1))
For Each cell In targetCells

    'A列に行番号を記述する
    targetCell = "行" & cell.Row

Next
```

○ 任意のレンジを変数に格納することで、縦横無尽のループや、飛び地的なセルのループも可能。

× ループの規則性が分かりにくい。
（Unionを使うともっと分かりにくい）

列のループ

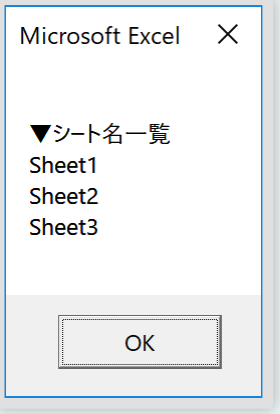
```
'列のループ（For構文）
Public Sub LoopColumnsByFor()

    'ループ用の変数を定義
    Dim iColumn As Long
    Dim firstColumn As Long
    Dim lastColumn As Long

    '開始列と終了列を設定
    firstColumn = 1
    lastColumn = 10

    'For構文による列のループ
    For iColumn = firstColumn To lastColumn

        '1行目に列番号を記述する
        Cells(1, iColumn) = "列" & iColumn

    Next

End Sub
```

	A	B	C	D	E	F	G
1	列1	列2	列3	列4	列5	列6	列7
2							

- 列のループは、ワークシートの横の方向に、ルールに従って処理を繰り返す書き方。考え方は、行のループと同様。
- 前頁で出てきた行のループのように、もちろんDo構文、For Each構文で書くことも可能。

シートのループ

```
'シートのループ（For構文）
Public Sub LoopSheetsByFor()

    'アウトプット用の変数を定義
    Dim buf As String
    buf = "▼シート名一覧"

    'For構文によるシートのループ
    Dim iSheet As Long, sheetCount As Long
    sheetCount = Worksheets.Count
    For iSheet = 1 To sheetCount

        '文字列変数に改行付きでシート名を格納する
        buf = buf & vbNewLine & Worksheets(iSheet).Name

    Next

    'シート名をメッセージ表示
    MsgBox buf

End Sub
```



- シートのループは、ワークブック内の各シートを順々に処理していく書き方。
- シート番号によるループが基本で、シート数を「sheetCount」に格納してからループ変数「iSheet」にてループさせることにしている。

実はFor Each構文でも、シートのループはできる

For構文

```
'For構文によるシートのループ
Dim iSheet As Long, sheetCount As Long
sheetCount = Worksheets.Count
For iSheet = 1 To sheetCount

    '文字列変数に改行付きでシート名を格納する
    buf = buf & vbNewLine & Worksheets(iSheet).Name

Next
```

○ ループ変数であるシート番号によって、処理内容を分岐させることができる。

○ シート数が変数に格納してあり、ReDimを含めたループ以外の処理も書きやすい。

× 途中でシート数が増減したときにループの上限を変更し忘れたら、ループ漏れが発生する。

For Each構文

```
'For Each構文によるシートのループ
Dim sheet As Worksheet
For Each sheet In ActiveWorkbook.Worksheets

    '文字列変数に改行付きでシート名を格納する
    buf = buf & vbNewLine & sheet.Name

Next
```

○ コーディングが非常にシンプル。

○ 全シートをループすることが明示的されているため、ループ漏れがないという安心感がある。

実務上は、シートの順序が手違いで入れ替わることで処理に不具合が出ることは少なくない。上記の処理はシートの順序が影響ない数少ない例と思われるが、こういうタイプでない場合は、シート名やヘッダー名によるチェックを加えて本当に処理対象とすべきファイルかの確認をする。なお、完全にコントロールされた環境下でオブジェクトに格納すれば、後工程でチェックが要らなくなるので便利である。

ファイルのループ

```
'ファイルのループ（静的参照）
Public Sub LoopStaticFiles()

    'ループ用の変数を定義
    Dim i As Long, temp As Variant, parentFolderPath As String
    Dim book As Workbook, wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    parentFolderPath = wsh.SpecialFolders("MyDocuments") & "¥vba test"
    temp = Array( _
        parentFolderPath & "¥excel_01.xlsx", _
        parentFolderPath & "¥excel_02.xlsx", _
        parentFolderPath & "¥excel_03.xlsx" _
    )

    'For構文によるファイルパス配列のループ
    For i = LBound(temp) To UBound(temp)

        'ファイルを開く（更新する場合はReadOnly:=Falseにする）
        Set book = Workbooks.Open(Filename:=temp(i), ReadOnly:=True)

        'ファイルに対して何か処理をする（この例は適当）
        MsgBox book.Worksheets.Count

        'ファイルを閉じる
        book.Close

    Next

End Sub
```

- ファイルのループは、特定ルールに従ってファイルパスを連続して参照し、ファイルを開いて同様の処理をした後で適宜保存して閉じる書き方。
- ファイルパスの連続参照は、配列ないしセルに並べたファイルパスをループする静的参照と、指定したフォルダ（ディレクトリ）内にある全てのファイルを自動取得する動的参照とに分かれる。
- なお、ファイルパスが動的参照の場合、元となるフォルダパスがVBA内部に書かれている固定型と、ダイアログボックスなどでユーザーが指定するユーザー指定型とに分かれる。また、連続参照の処理はDir関数を使う方法とFileSystemObjectを使う方法とに分かれる。

実用的な FileSystemObject による動的参照

```
'ファイルのループ（動的参照）※Microsoft Scripting Runtimeを参照設定
Public Sub LoopDynamicFiles()

    '元になるフォルダパスに格納された全ファイルを取得する
    Dim folderPath As String, fso As FileSystemObject, file As Object, files As Object
    folderPath = Application.FileDialog(msoFileDialogFolderPicker).SelectedItems(1)
    Set fso = New FileSystemObject
    Set files = fso.GetFolder(folderPath).files

    '取得したファイルをループし、Excelファイルであれば処理を続行する
    Dim buf As String, book As Workbook, sheet As Worksheet
    For Each file In files
        buf = LCase(Mid$(file.Name, InStrRev(file.Name, ".") + 1))
        If buf = "xls" Or buf = "xlsx" Or buf = "xlsm" Then

            'Excelファイルを開いて、各シートの倍率を100%にして左上表示にする
            Set book = Workbooks.Open(Filename:=file.Path, ReadOnly:=False)
            For Each sheet In book.Worksheets
                sheet.Select
                sheet.Cells(1, 1).Select
                ActiveWindow.Zoom = 100
                ActiveWindow.ScrollRow = 1
                ActiveWindow.ScrollColumn = 1
            Next

            '一番左のシートを選択して上書き保存して閉じる
            book.Worksheets(1).Select
            book.Save
            book.Close

        End If
    Next

End Sub
```

本スライドでは省略したが、FileDialogはファイルを選択しなくても先に進めてしまうため、エラーチェックと分岐処理を入れるべき点には留意が必要である。

配列のループ

```
'配列のループ（For構文）
Public Sub LoopArrayByFor()

    '存在をチェックするシート名を配列で定義
    Dim i As Long, sheetNames As Variant, sheet As Worksheet
    sheetNames = Array("ユーザー", "商品", "購入レコード")

    'For構文による配列のループ
    On Error GoTo ErrorHandling
    For i = LBound(sheetNames) To UBound(sheetNames)
        Set sheet = ActiveWorkbook.Worksheets(sheetNames(i))
    Next
    MsgBox "全てのシートを確認できました。"
    Exit Sub

'シートが存在しなかった場合にアラートを上げる
ErrorHandling:
    MsgBox "「" & sheetNames(i) & "」シートが存在しません。"

End Sub
```

- 配列のループは、自作した配列の要素をループして、要素内の内容に従って処理をする書き方。
- Excel VBAで配列を駆使する場合は、“いわゆる多次元配列”に手を出すわけだが、多次元配列のループには色々な流儀があることを認識しておいた方がよい。
  ①入れ子配列にするか二次元配列にするか、
  ②値を一括代入するか分割代入するか、
  ③For構文にするかFor Each構文にするか、
  については後のスライドに書く。

VBAで配列を定義すると添え字の最小値は「0」になるため、要素の数はUbound(sheetNames)の値とは一致せず、Ubound(sheetNames)+1になる（厳密にはUbound(sheetNames)-Lbound(sheetNames)+1になる）点は留意しておきたい。なお、Option Baseステートメントで添え字の最小値を上書きすることはできるが、開発者以外は気付きづらく、保守性を損なうため、安易に宣言することは避けたい。

①使うのは入れ子配列。二次元配列は面倒なので使わない

入れ子配列

```
'一括代入型による入れ子配列のループ（For Each構文）
Public Sub LoopNestArrayByForEach_BulkInput()

    'ループ用の変数を定義
    Dim setting As Variant, settings As Variant
    settings = Array( _
        Array(1, "赤色", RGB(255, 0, 0)), _
        Array(3, "緑色", RGB(0, 255, 0)), _
        Array(5, "青色", RGB(0, 0, 255)) _
    )

    'For Each構文による配列のループ
    For Each setting In settings

        '指定行に、指定の文字を指定色で記述する
        With Cells(setting(0), 1)
            .Value = setting(1)
            .Font.Color = setting(2)
        End With

    Next

End Sub
```

二次元配列

```
'二次元配列のループ（For Each構文）
Public Sub LoopTwoDimensionalArrayByFor()

    'ループ用の変数を定義
    Dim i As Long, settings() As Variant
    ReDim settings(0 To 2, 0 To 2)
    settings(0, 0) = 1
    settings(0, 1) = "赤色"
    settings(0, 2) = RGB(255, 0, 0)
    settings(1, 0) = 3
    settings(1, 1) = "緑色"
    settings(1, 2) = RGB(0, 255, 0)
    settings(2, 0) = 5
    settings(2, 1) = "青色"
    settings(2, 2) = RGB(0, 0, 255)

    'For構文による配列のループ
    For i = LBound(settings, 1) To UBound(settings, 1)

        '指定行に、指定の文字を指定色で記述する
        With Cells(settings(i, 0), 1)
            .Value = settings(i, 1)
            .Font.Color = settings(i, 2)
        End With

    Next

End Sub
```

見ての通り、二次元配列の場合は代入を1つずつ行わなければならないため、非常にコード数が多くなる。また、入れ子配列と違ってFor Each構文が使えず、縦のループを行う際もLboundとUboundに第何要素かを明示する必要がある。とにかく面倒なので二次元配列は使わないのが原則。ただし、後述する値配列としての活用だけは有益である。

②入れ子配列の代入には、一括代入型に分がある

一括代入型

```
'ループ用の変数を定義
Dim setting As Variant, settings As Variant
settings = Array( _
    Array(1, "赤色", RGB(255, 0, 0)), _
    Array(3, "緑色", RGB(0, 255, 0)), _
    Array(5, "青色", RGB(0, 0, 255)) _
)
```

○ Arrayで入れ子を作ってからVariant型の変数に代入するため、ReDimがいらず、代入が1回で済むのでシンプル。

○ 要素番号は昇順に設定されるため、要素追加が楽。

× 分かりやすくするために半角アンダースコアで改行することが多いが、改行は24回が上限のため、大きすぎるものは綺麗に改行できず、推奨できない。

分割代入型

```
'ループ用の変数を定義
Dim i As Long, settings() As Variant
ReDim settings(0 To 2)
settings(0) = Array(1, "赤色", RGB(255, 0, 0))
settings(1) = Array(3, "緑色", RGB(0, 255, 0))
settings(2) = Array(5, "青色", RGB(0, 0, 255))
```

○ 要素数をReDimで定義するので、大きさを理解しやすい。

○ 要素番号を指定して代入するので、必ずしも順番通りに処理を書かなくともよい。

× 代入時に要素番号を数字（マジックナンバー）で振ってしまうと、要素の追加時に要素番号の書き換えが面倒。そのため、要素の書き換えがありうる場合は、Long型の変数を代入ごとにインクリメント（+1）して要素番号を指定しておくことが推奨。

③入れ子配列のループは、For Each構文の方が有利

For Each構文

```
'For Each構文による配列のループ
For Each setting In settings

    '指定行に、指定の文字を指定色で記述する
    With Cells(setting(0), 1)
        .Value = setting(1)
        .Font.Color = setting(2)
    End With

Next
```

○ コーディングが非常にシンプル。

× For Each構文に放り込むために、ループ変数はVariant型かObject型で定義する必要がある。

× センス良くループ変数を命名しないと、何をやっているか分からなくなる。

For構文

```
'For構文による配列のループ
For i = LBound(settings) To UBound(settings)

    '指定行に、指定の文字を指定色で記述する
    With Cells(settings(i)(0), 1)
        .Value = settings(i)(1)
        .Font.Color = settings(i)(2)
    End With

Next
```

○ たぶんこちらの方が一般的。

○ 要素番号が変数に格納してあり、要素番号による処理の分岐が書きやすい。

× 入れ子配列の場合、要素番号の参照のために括弧が複数並ぶため、分かりづらくなる。

なお、二次元配列は値配列としての使い方のみ有益

```
'値配列の処理
Public Sub LoopValueArrayByFor()

    'ループ用の変数を定義
    Dim valueArray As Variant, targetRange As Range, iRow As Long, iColumn As Long
    Set targetRange = Range(Cells(1, 1), Cells(5, 2))
    valueArray = targetRange.Value

    'For構文による行×列をループして、とりあえず値配列に更新後テキストを格納する
    For iRow = LBound(valueArray, 1) To UBound(valueArray, 1)
        For iColumn = LBound(valueArray, 2) To UBound(valueArray, 2)
            valueArray(iRow, iColumn) = "Cells(" & iRow & "," & iColumn & ")=""" & valueArray(iRow, iColumn) & """"
        Next
    Next

    'テキストを更新した値配列をまとめてセルに戻す
    targetRange.Value = valueArray

End Sub
```

	A	B
1	月曜日	184
2	火曜日	321
3	水曜日	32
4	木曜日	89
5	金曜日	51

	A	B
1	Cells(1,1)="月曜日"	Cells(1,2)="184"
2	Cells(2,1)="火曜日"	Cells(2,2)="321"
3	Cells(3,1)="水曜日"	Cells(3,2)="32"
4	Cells(4,1)="木曜日"	Cells(4,2)="89"
5	Cells(5,1)="金曜日"	Cells(5,2)="51"

とりあえず変数に格納して後でセルに代入するのは二度手間に見えるが、事実、二度手間である。このような処理をするのは次頁で紹介する圧倒的な高速化のためではあるのだが、結果としてコードが直観的には分かりづらいものになるため、スピードが気にされない場合では控えるべき書き方である。

値配列でまとめて代入すると高速になる不思議

なんと 24 倍

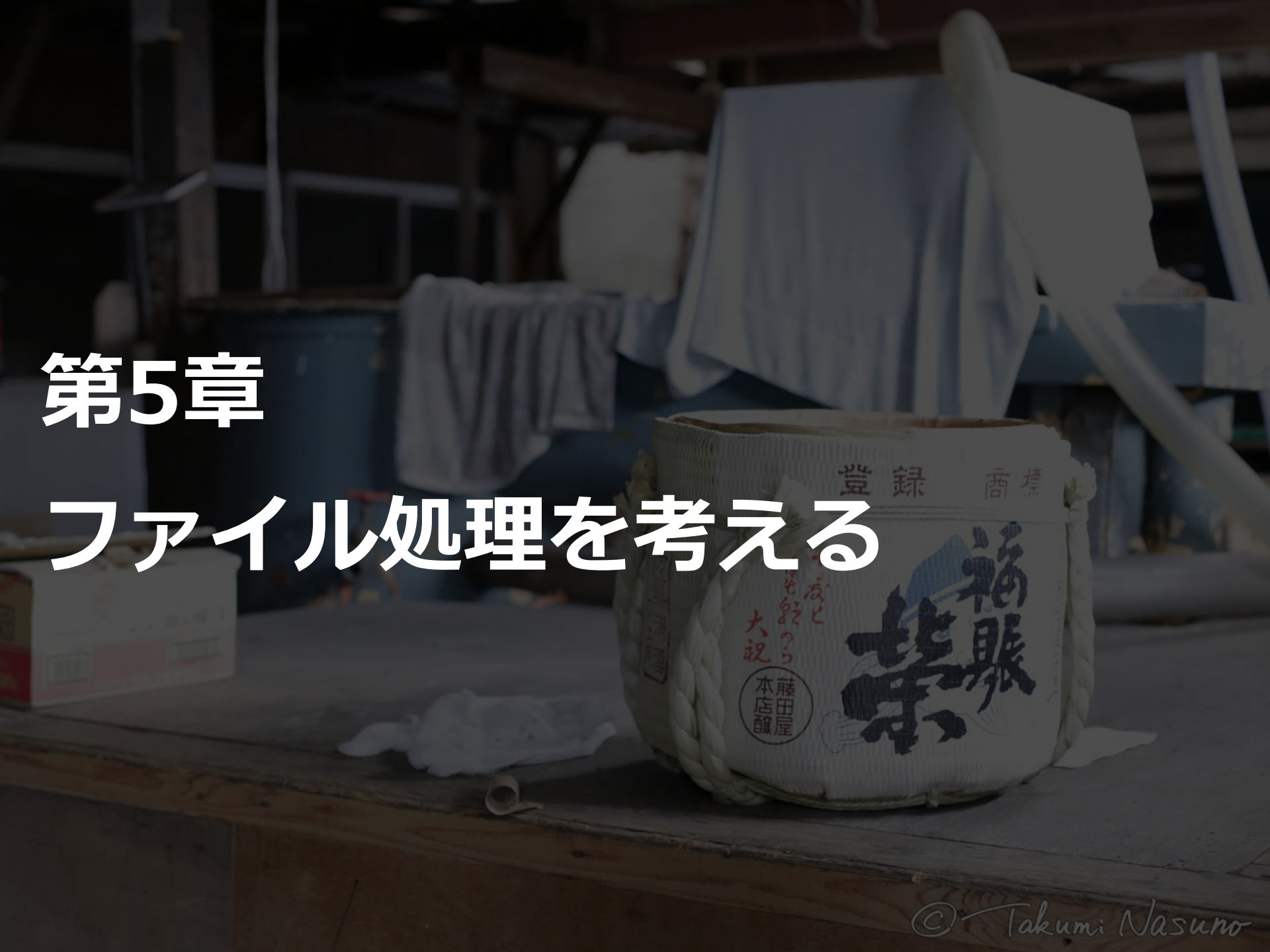
220 秒 → 9 秒

```
'▼セルにそのまま代入する場合
Public Sub WriteCells()

    Dim iRow As Long, iColumn As Long
    Dim firstRow As Long, lastRow As Long
    Dim firstColumn As Long, lastColumn As Long
    firstRow = 1
    lastRow = 1000000
    firstColumn = 1
    lastColumn = 10

    For iRow = firstRow To lastRow
        For iColumn = firstColumn To lastColumn
            Cells(iRow, iColumn) = iRow * iColumn
        Next
    Next

End Sub
```

```
'▼値配列にまとめて、そのままセルに代入する場合
Public Sub WriteCellsValueByValueArray()

    Dim iRow As Long, iColumn As Long
    Dim firstRow As Long, lastRow As Long
    Dim firstColumn As Long, lastColumn As Long
    firstRow = 1
    lastRow = 1000000
    firstColumn = 1
    lastColumn = 10

    Dim temp() As Variant
    ReDim temp(firstRow To lastRow, _
        firstColumn To lastColumn)
    For iRow = firstRow To lastRow
        For iColumn = firstColumn To lastColumn
            temp(iRow, iColumn) = iRow * iColumn
        Next
    Next
    Range(Cells(firstRow, firstColumn), _
        Cells(lastRow, lastColumn)).Value = temp

End Sub
```

恐らくセルへの代入が変数への代入に比べて圧倒的に遅く、かつ1回あたりの代入にかかる時間がデータ量に依存しないようなので、いったん変数にすべて格納した後でまとめてセルに代入することで、圧倒的に遅いセルへの代入を1回に済ませられることが理由だと考えている。これは値の代入（write）に限らず値の参照（read）に関しても同様であるため、総じてセルへのアクセスは圧倒的に遅いと考えたほうが良い。なお、セルへの代入にかかる時間は0.000001秒程度と思われるため、それなりの数の処理があって初めて高速化の恩恵を体感できる点は留意されたい。また、高速化の代償として値配列定義時には要素分だけPCメモリを消費し、その規模およそ15MB/100万セルに相当する。100万行×100列クラスの配列を定義しようとするとメモリ不足を誘発するケースが出てくるので、大規模なデータを扱うときは処理の分割を検討すること。

うまく構造化して
シンプルかつ効果的なコードを
書きましょう



第5章
ファイル処理を考える

色々なファイルを処理できると仕事の幅が広がります

ということで、5つのファイル形式について見ていきます

Excel VBA では、他にも Word や PowerPoint などのファイルを開いたり、MySQL や PostgreSQL などのデータベースに接続したり、メールを送ったり、Google Apps内のスプレッドシートと連携したりなど、けっこう色々なことができる。でもここではローカル環境でよくある5つのファイル形式について、簡単に紹介するにとどめておき、これ以外の紹介はGoogle先生に譲ることにする。

5つのファイル形式のメリット、デメリット

Excel	CSV	Text	XML	JSON
もっとも一般的。 文字列以外にも計算式やチャート、図形など、いろいろな情報を保存できる。	非常に大規模な構造化データの保持に向いている。 Excelのように行×列の形式で保持できる。	何も縛りがない。	規則性のある非構造化データとしてread/writeともに自由にできる。	Web界隈で最もメジャーな規則性のある非構造化データなので、他システムとの連携がスムーズ。 文字列以外のデータも扱える。
読み取る際は、ファイルをワークブックオブジェクトとして開く手間がかかる。 バイナリ形式のため、テキストエディタで開くことはできない。 自由度が高すぎるため、機械的な処理にはやや不向き。	ワークブックオブジェクトとして開く際に自動計算されて元の値が崩れる場合があるので、必要に応じて文字列に変換して開く必要がある。	規則性のない非構造化データなので、複雑なデータには向いていない。	規則性のある非構造化データとしては、JSONに比べてマイナーになってきている。	VBAの標準機能はJSONに対応していないため、別途モジュール(VBA-JSON等)を導入する必要あり。 ただし、VBA-JSONでは、readはできてもwriteができない。

各形式の使い方は
ググるべし

でもそれだと不便なので
最低限の構文だけ書きます

Excel/CSVを開く方法

```
'Excelを開いて各シートの行をループ
Public Sub OpenExcel()

    'ファイルパスを定義
    Dim filePath As String, wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    filePath = wsh.SpecialFolders("MyDocuments") & "¥vba test¥excel_01.xlsx"

    'ファイルを開く
    Dim targetBook As Workbook
    Set targetBook = Workbooks.Open(Filename:=filePath, ReadOnly:=False)

    'シートをループ
    Dim sheet As Worksheet, iRow As Long, firstRow As Long
    firstRow = 2
    For Each sheet In targetBook.Worksheets

        'A列に値がある間まで行をループ
        iRow = firstRow
        Do While sheet.Cells(iRow, 1) <> ""

            'ここにループ中の処理を書く
            MsgBox sheet.Cells(iRow, 1) & ":" & sheet.Cells(iRow, 2)

            iRow = iRow + 1
        Loop

    Next

    'ファイルを閉じる
    targetBook.Close

End Sub
```

ReadOnlyパラメータをTrue/Falseどちらにするかは実務上非常に重要で、①上書きする可能性あり⇒True、②上書きしないor複数人が開く可能性あり⇒False、という区別が必要である。なお、複数人が開く可能性があるのに①にすると、途中でアラートが出て終了する可能性があるので、エラー分岐が必要になる点は留意が必要。

CSVを文字列に変換して開く方法

```
'CSVを文字列に変換してExcelとして開く
Public Sub OpenCsvAsText()

    'ファイルパスを定義
    Dim filePath As String, wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    filePath = wsh.SpecialFolders("MyDocuments") & "¥vba test¥test.csv"

    '出力先のシートを指定
    Dim targetSheet As Worksheet
    Set targetSheet = ActiveSheet

    '出力先のシートに貼り付け
    With targetSheet
        With .QueryTables.Add(Connection:="TEXT;" & filePath, Destination:=.Cells(1, 1))
            .AdjustColumnWidth = False
            .TextFilePlatform = 932
            .TextFileTextQualifier = xlTextQualifierDoubleQuote
            .TextFileTabDelimiter = True
            .TextFileColumnDataTypes = Array(2, 2, 2) '各列を文字列(2)として指定。列分だけ要素が必要。
            .RefreshStyle = xlOverwriteCells
            .Refresh
            .Delete
        End With
    End With

End Sub
```

本例は、タブ区切り×ダブルクオテーション囲いのCSVファイルを開く構文となっている。文字列に変換して開く際の一番の問題は、TextFileColumnDataTypesプロパティである。というのも、もちろんデータ型は文字列である「2」を指定するわけだが、各列ごとに設定しないといけないため、列数分だけ「2」を要素に持つ配列を代入しないといけない、つまり代入する値が動的に変化してしまうからである。幸いなことに、配列の要素数は列数より多くてもかまわないため、厳密に要素数を考えるのが面倒であれば要素数を100個とか1000個とかにするなどの対応でよいかもしれない。

Textを開く方法：Shift-JISの場合

```
'テキスト（Shift_JIS）を開く
Public Sub OpenTextByShiftJis()

    'ファイルパスを定義
    Dim filePath As String, wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    filePath = wsh.SpecialFolders("MyDocuments") & "¥vba test¥txt_shiftjis_01.txt"

    'ファイルを開く
    Dim buf As String
    Open filePath For Input As #1

        '行のループ
        Do Until EOF(1)

            'ここにループ中の処理を書く
            Line Input #1, buf
            MsgBox buf

        Loop

    'ファイルを閉じる
    Close #1

End Sub
```

テキストファイルを開く場合に最も有名なのが、このOpenメソッドである。対になる書き込みの場合は、新規作成が Output メソッド、既存ファイルへの追記が Append メソッドとなっている。このあたりはググればすぐに出てくるので、説明は他文献に譲ることにする。

Textを開く方法：UTF-8の場合

```
'テキスト（UTF-8）を開く
Public Sub OpenTextByUTF8()

    '※最新のMicrosoft ActiveX Data Objects x.x Libraryを参照設定する

    'ファイルパスを定義
    Dim filePath As String, wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    filePath = wsh.SpecialFolders("MyDocuments") & "¥vba test¥txt_utf8_01.txt"

    'ファイルを開く
    Dim adoSt As Object
    Set adoSt = CreateObject("ADODB.Stream")
    adoSt.Charset = "UTF-8"
    adoSt.Open
    adoSt.LoadFromFile filePath

    '行のループ
    Do Until adoSt.EOS

        'ここにループ中の処理を書く
        MsgBox adoSt.Readtext(adReadLine)

    Loop

    'ファイルを閉じる
    adoSt.Close

End Sub
```

前頁で紹介したOpenメソッドによるテキストファイルの読み取りは、Shif-JISによる文字コードに限定される。UTF-8をはじめとしたShift-JIS以外の文字コードの場合はOpenメソッドが使えないため、しかるべきライブラリを参照した後に ADODB.Stream オブジェクトとして文字コードを指定して開く必要がある。そのため、「文字コードって何？」というレベルの方は、文字コードの不整合によるバグで慌てふためく可能性が高いため、安易にテキストファイルを扱わない方が無難である。

なお、読み取りは上記のようにReadtextメソッドを使う一方で、書き込みはWritetextメソッド、保存はSaveToFileメソッドが用意されている。書き込み、保存ともに第2引数を正しく設定しないと不具合が生じる可能性があるので、分からない方は適宜ググって確認すること。

XMLを開く方法

```
'XMLを開く
Public Sub OpenXML()

    '※Microsoft XML,v6.0を参照設定する

    'ファイルパスを定義
    Dim filePath As String, wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    filePath = wsh.SpecialFolders("MyDocuments") & "¥vba test¥xml_01.xml"

    'ファイルを開く
    Dim xmlDoc As MSXML2.DOMDocument60
    Set xmlDoc = New MSXML2.DOMDocument60
    xmlDoc.async = False
    xmlDoc.Load filePath

    '要素のループ
    Dim rootNode As IXMLDOMNode, node As IXMLDOMNode
    Set rootNode = xmlDoc.SelectSingleNode("//root")
    For Each node In rootNode.ChildNodes

        'ここにループ中の処理を書く
        MsgBox node.BaseName & ":" & node.Text

    Next

End Sub
```

XMLを開くにはXMLをオブジェクトとして定義する必要がある。上記の例では「Microsoft XML, v6.0」を参照したうえで事前バインディングにてxmlDoc変数を定義してセットしている。一方、古い実行環境が混ざるなどの理由で事後バインディングにしたい場合は、代わりに「Dim xmlDoc As IXMLDOMDocument」で定義し、「Set xmlDoc = CreateObject(“Microsoft.XMLDom”)」でセットすればよい。

JSONを開く方法

```
'JSONを開く
Public Sub OpenJSON()

    '※VBA-JSONをimportする (https://github.com/VBA-tools/VBA-JSON)

    'ファイルパスを定義
    Dim filePath As String, wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    filePath = wsh.SpecialFolders("MyDocuments") & "¥vba test¥json_01.json"

    'ファイルを開く
    Dim fso As FileSystemObject, jsonText As String, json As Object
    Set fso = New FileSystemObject
    jsonText = fso.OpenTextFile(filePath, ForReading).ReadAll
    Set json = JsonConverter.ParseJson(jsonText)

    '要素のループ
    Dim key As Variant
    For Each key In json

        'ここにループ中の処理を書く
        MsgBox key & ":" & json(key)

    Next

End Sub
```

ありとあらゆるWebサービスで利用されているJSON形式、非構造化データを自由に設定でき、文字列以外も扱えるという圧倒的なスペックを持つデータ形式だが、VBAとの相性が悪すぎる点は嘆かわしい。VBA-JSONモジュールを導入しても、できるのは read に限られ、新規にJSONファイルを生成したり、既存のJSONファイルを更新したりなどの処理ができないため、使い勝手が非常に悪くなってしまっている。自力でライブラリを作るのは手間なので、いいライブラリがあったら教えてください・・・

おまけ：XMLの処理例

```
'XMLファイルの処理　※Microsoft XML, v6.0を参照設定する
Public Sub ProcessXmlFile()

    'ファイルを新規作成する（rootノードまで作成する）
    Dim xmlDoc As MSXML2.DOMDocument60, rootNode As IXMLDOMNode
    Set xmlDoc = New MSXML2.DOMDocument60
    xmlDoc.async = False
    xmlDoc.appendChild xmlDoc.createProcessingInstruction("xml", "version=""1.0"" encoding=""UTF-8""")
    Set rootNode = xmlDoc.appendChild(xmlDoc.createElement("root"))

    '新しいノードの追加とテキスト代入
    Dim townsNode As IXMLDOMNode, townNode As IXMLDOMNode, townNames As Variant, townName As Variant
    townNames = Array("鶴見区", "棚倉町", "西会津町")
    Set townsNode = rootNode.appendChild(xmlDoc.createElement("towns"))
    For Each townName In townNames
        Set townNode = townsNode.appendChild(xmlDoc.createElement("town"))
        townNode.Text = townName
    Next

    'ファイルをいったん保存する
    Dim folderPath As String, wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    folderPath = wsh.SpecialFolders("MyDocuments") & "¥vba test"
    xmlDoc.Save folderPath & "¥processXmlFile_01.xml"

    'ファイルを再度開く
    Set xmlDoc = New MSXML2.DOMDocument60
    xmlDoc.async = False
    xmlDoc.Load folderPath & "¥processXmlFile_01.xml"

    'ノードをループして条件に合致するものを削除し、別名保存する（鶴見区だけ削除）
    For Each townNode In xmlDoc.SelectNodes("//root/towns/town")
        If townNode.Text = "鶴見区" Then townNode.parentNode.RemoveChild townNode
    Next
    xmlDoc.Save folderPath & "¥processXmlFile_02.xml"

End Sub
```

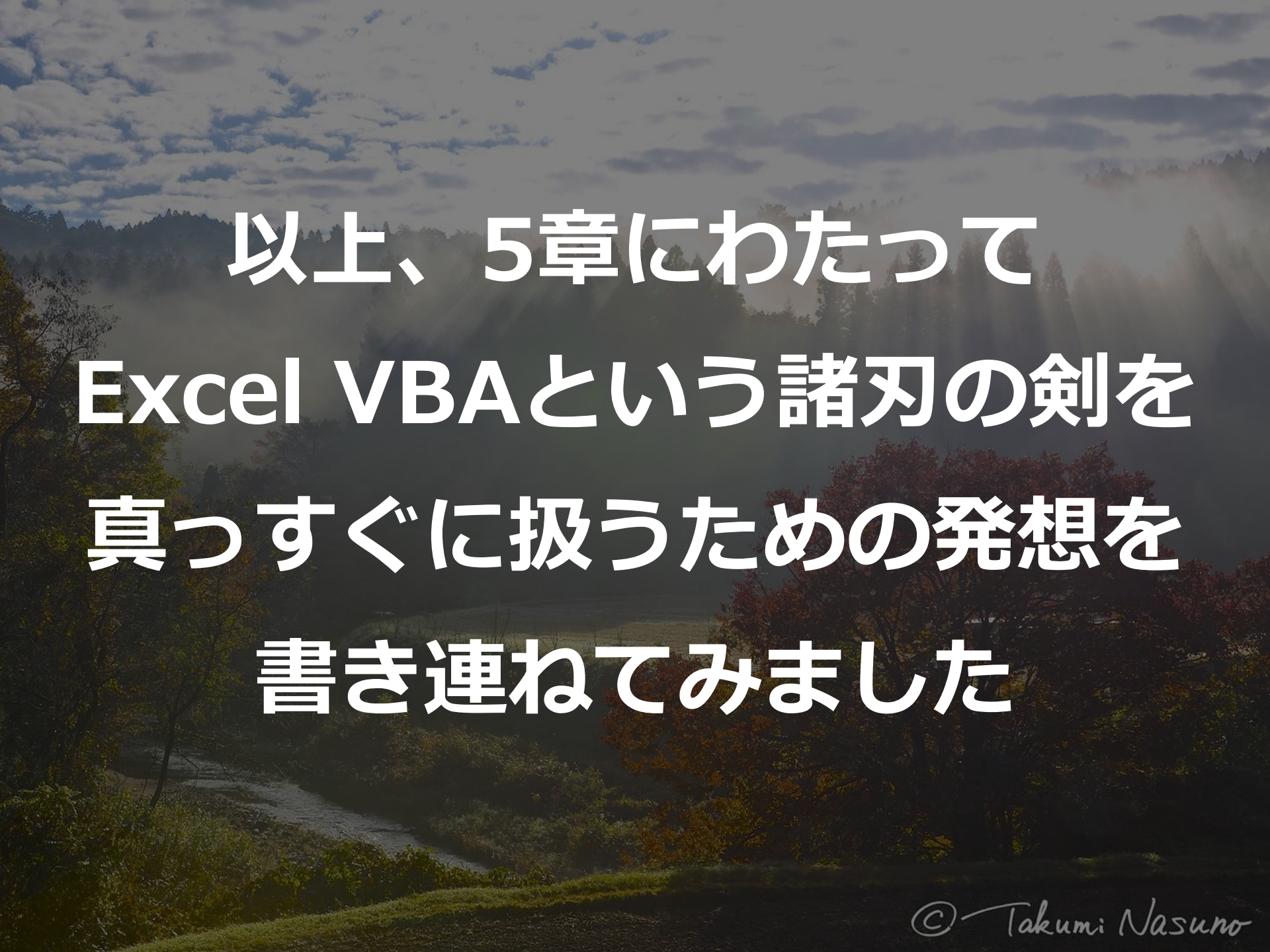
「processXmlFile_01.xml」の中身

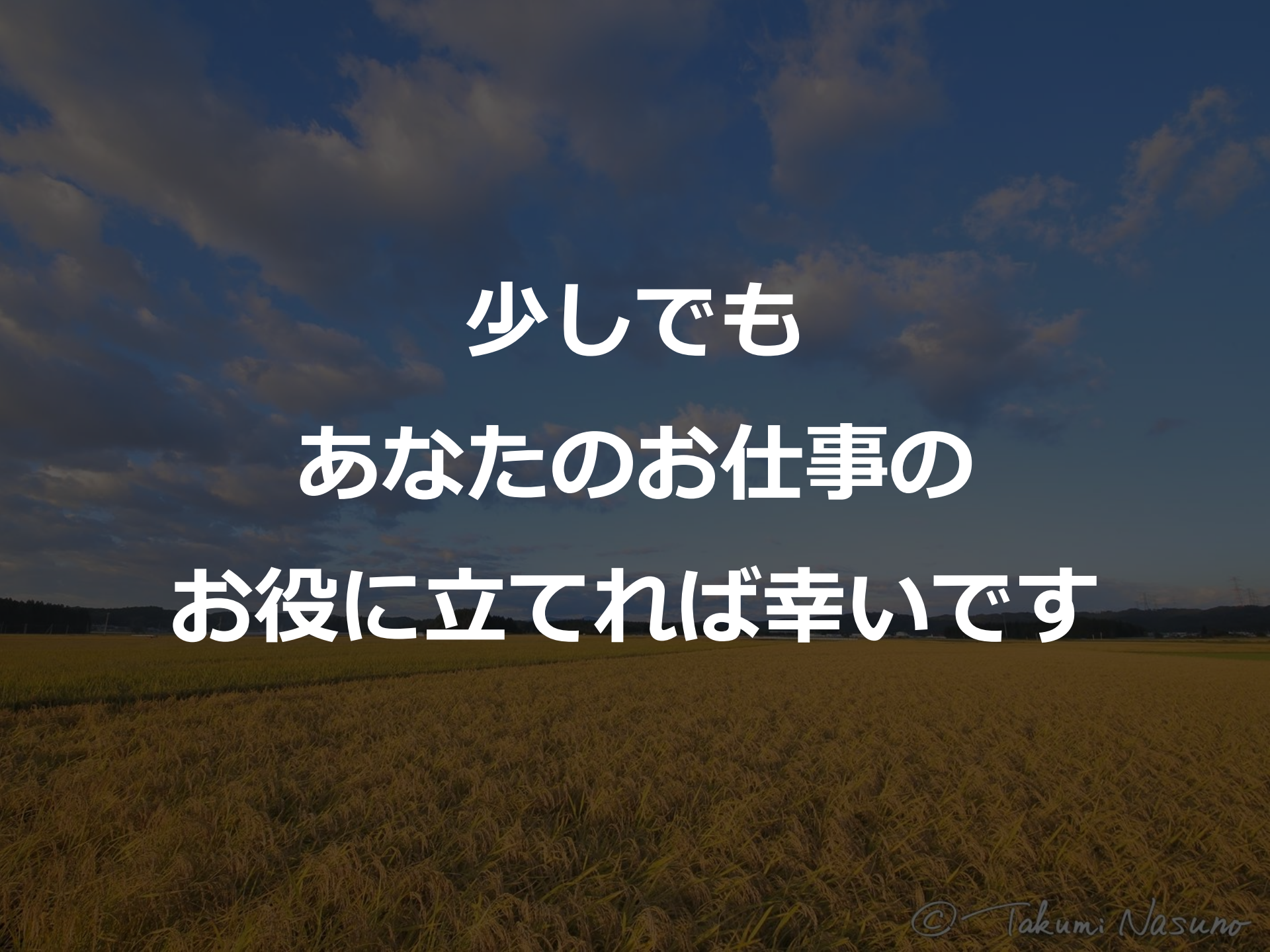
```
<?xml version="1.0" encoding="UTF-8"?>
<root><towns><town>鶴見区</town><town>
棚倉町</town><town>西会津町
</town></towns></root>
```

「processXmlFile_02.xml」の中身

```
<?xml version="1.0" encoding="UTF-8"?>
<root><towns><town>棚倉町</town><town>
西会津町</town></towns></root>
```

必要最小限の範疇で
最適なファイル形式を
扱いましょう


以上、5章にわたって
Excel VBAという諸刃の剣を
真っすぐに扱うための発想を
書き連ねてみました
© Takumi Nasuno


少しでも
あなたのお仕事の
お役に立てれば幸いです
© Takumi Nasuno

最後まで お読みいただき
ありがとうございました