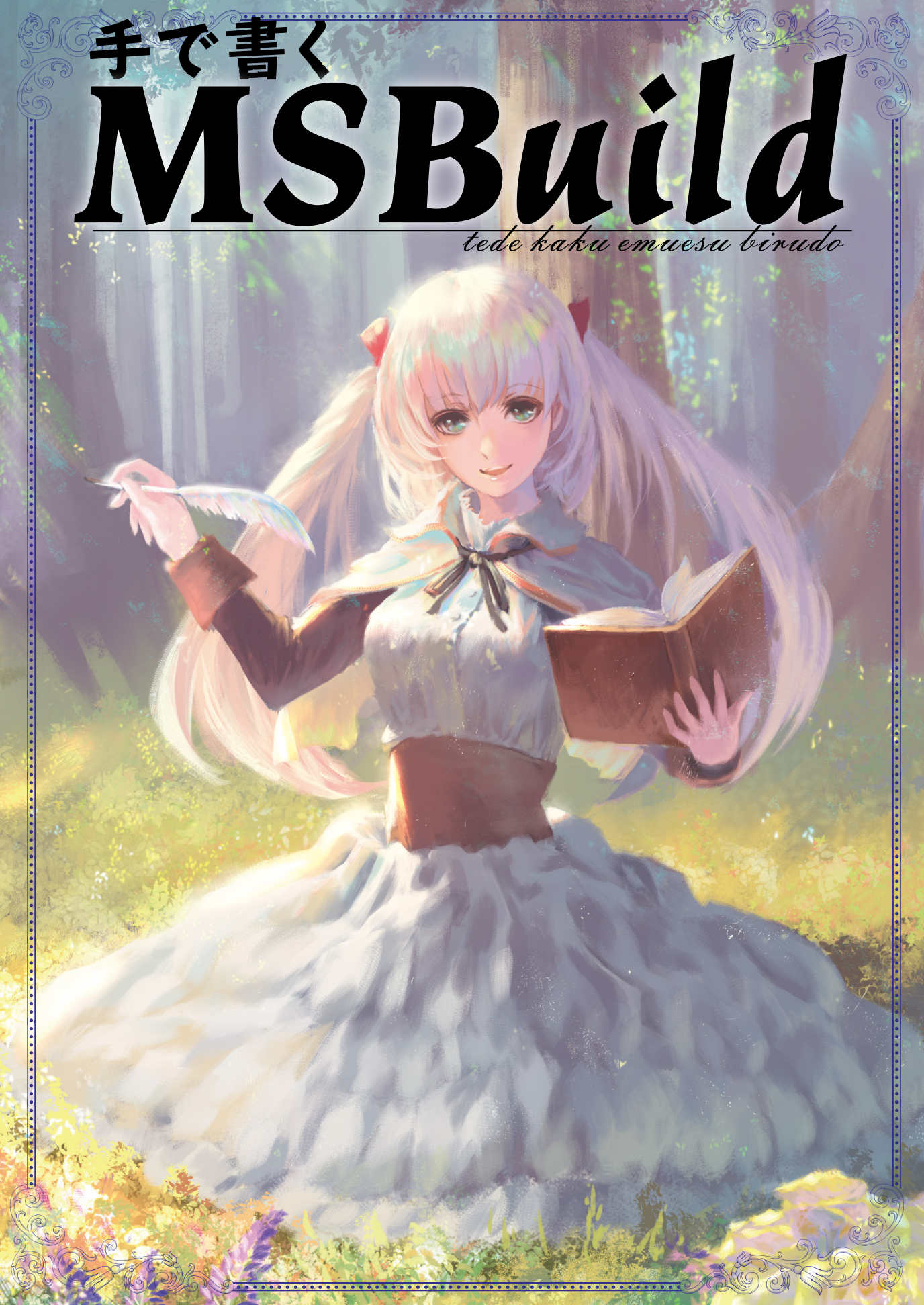
手で書く MSBuild

tede kaku emuesu birudo



手で書く

MSBuild

tede kaku emuesu birudo

2018 あれくま

目次

1. はじめに 6
    1.1. ビルドツールとは 7
    1.2. MSBuild の特徴 8
2. 最初の一歩 10
    2.1. インストール 10
        2.1.1. Windows 10
        2.1.2. macOS 12
        2.1.3. Linux 等 13
        2.1.4. ソースからビルドする 13
    2.2. MSBuild コマンドの使い方 13
    2.3. プロジェクトを書いてみる 15
    2.4. ソリューションファイルについて 17
3. プロジェクトの中身を見てみよう - プロジェクトファイルの説明 19
    3.1. 小さいプロジェクトファイル 19
    3.2. ファイルの構成を見てみよう 21
        3.2.1. プロジェクトの定義 21
        3.2.2. プロパティとアイテムの定義 22
        3.2.3. ターゲットの定義とタスク 24
        3.2.4. プロジェクトファイル全体を再確認 25
    3.3. プロパティについてくわしく 25
    3.4. アイテムについてくわしく 27
        3.4.1. アイテムのワイルドカードについて 29
    3.5. ターゲットについてくわしく 32
        3.5.1. タスクが失敗したらどうなるの？ 33
        3.5.2. 更新されたファイルだけビルドしたい - ターゲットの入出力 33
        3.5.3. 先に他のターゲットを実行したい - ターゲットの依存関係 35
    3.6. タスクについてくわしく 36
        3.6.1. Message タスク 37
        3.6.2. Warning / Error タスク 37
        3.6.3. Touch タスク 38
        3.6.4. Copy タスク 38
        3.6.5. Move タスク 39
        3.6.6. Delete タスク 40
        3.6.7. MakeDir タスク 40
        3.6.8. RemoveDir タスク 41
        3.6.9. Exec タスク 41

3.6.10. MSBuild タスク 42
4. 本格的に使いたい - もっと詳しい機能について 45
4.1. ファイル名をいろんな形で渡したい - アイテムの参照 45
4.1.1. ツリー構造を維持したままファイルコピーをしたい 47
4.2. 一つの指定でタスクを複数回実行したい - バッチ処理 48
4.3. プロパティをいろんな形で渡したい - プロパティの参照 49
4.4. 環境変数を参照したい 50
4.5. 既定のプロパティ 52
4.6. 特定の値の時だけ違うことしたい - 簡単な条件実行 53
4.7. 一部のファイルだけ違うことしたい - アイテムのメタデータ 57
4.7.1. 既定のメタデータを作る 58
4.8. 他のプロジェクトを流用したい - プロジェクトのインポート 59
5. プログラムをビルドしよう - .NET のプロジェクト 60
5.1. .NET Framework のプロジェクトファイル 60
5.2. .NET Core のプロジェクトファイル 61
5.3. 外部ライブラリを参照したい - NuGet パッケージの参照 62
5.4. 既存のプロジェクトを活用しよう 63
5.4.1. 前処理や後処理を追加したい - ターゲットのフック 63
5.4.2. 実行結果を受け取りたい - ターゲットからの値の出力 66
5.4.3. 既存のプロジェクトファイルを活用したい - 既存のターゲット 68
6. MSBuild に更なるパワー - タスクの追加 72
6.1. 新しいタスクを追加したい - アセンブリのロード 72
6.2. C# のコードを実行したい - インラインタスクの作成 73
6.3. 自分でタスクを作りたい - カスタムアセンブリを作る 76
6.3.1. 以前の MSBuild で使えるタスクを作りたい 78
7. 最後に 80
7.1. 参考文献 80
7.1.1. MSBuild のドキュメント 80
7.1.2. MSBuild の GitHub プロジェクトページ 80
7.2. ツール類 81
7.2.1. Albacore 81
7.2.2. Fake 81

1. はじめに

この本は MSBuild について解説するニッチな本です。なんでそんなの書こうと思っちゃったんですかね。

ちょっと Visual C++ のプロジェクトをバッチビルドしたくて、せっかくなので MSBuild を使おうとプロジェクトファイルを書いてみたのですが、変な書き方で意外と面白かったんですよ。ただちょっと書き方が変で、理解できるまで少し時間がかかったので本にまとめてみました。

そもそも MSBuild ってなんじゃろなって人がこの本を読むかは疑問なところがありますが、一応説明しておきますと MSBuild は Microsoft 製のビルドツールです。

ビルドツールっていうのは Make とか Ant とか Maven とか Rake とか SCons とかまあそういうツールです。プログラムのビルドなんかをする時に、ビルド手順を書いとくと、それに沿って実行してくれるやつです。バッチファイルみたいなものではあるけど、ビルド用にもうちょい便利な機能が付いてるやつです。

そういうビルドツールの中で MSBuild は C# 用 (というか .NET 用) という理解をするとまあだいたい合ってると思います。Make なんかは C や C++ 用、Ant とか Maven なんかは Java 用、Rake なんかは Ruby 用、SCons なんかは Python 用とかなると思いますが[1]、MSBuild は C# や VB.NET なんかの .NET 用です。あと Visual C++ でも使われます。

この本は、そんな MSBuild を使ってバッチ処理をちょこっと書けるようになると良いなあ、というのを目標にしたものです。ツールでプロジェクトファイルを生成するなんて邪道！全て手で書くべき！……とかいうハードコアなところは目指していません。あんま意味ないので。それよりは、既にある、またはツールで作ったプロジェクトファイルをいじったり呼び出したりして活用することを目的としています。

Visual Studio や .NET Core SDK を使ってる人で、普通にビルドをする以上の追加のバッチ処理をしたいなぁという人を対象としています。追加のバッチ処理をしたいとか思ったこともない人も居るかもしれませんけども、きっとリリース用のファイルを zip にまとめるとかはよくやると思うんですよね。手動でやっても大したことではないんですけど、そういうのを自動的にやっちゃおうという話です。

Visual Studio も .NET も使ってないわって人はあんま面白くないと思います。

前提知識としては、ちょこっとながらコマンドラインをいじれることが必要です。Visual Studio は全然登場しません。コマンドラインで操作します。といっても基本的に MSBuild コマンドを起動させるだけなんで、GUI じゃねえと全く触れんとか言わずにやってみてください。

また、XML の基本的な書き方について知識が必要です。MSBuild のファイルは XML で書かれているので XML を書きまくることになるんですが、さすがに XML 自体の説明はこの本ではしません。基本的ってのは普通のタグの書き方くらい分かれば十分です。名前空間みたいな高度な知識は求め

1　もちろんそれら以外に使われることは多々ありますが、基本的には、ね？

られないので XAML 書くよりむしろ簡単です。

1.1. ビルドツールとは

MSBuild はビルドツールってことでしたが、ビルドツールについて簡単に説明しておきましょう。そんなんわかるだろ……っていう人はここは読み飛ばしてかまいません。

名前の通りプログラムのビルドに使うツールです。プログラムが小さいうちはいちいちコマンドを打ってコンパイルしてもいいのですが、大きくなってくるとそうもいきません。大きくなくても何度も書き直したくはないですしね。

バッチファイルなりシェルスクリプトでバッチビルドするという手もあるのですが、沢山あるファイルを毎度コンパイルしなおしていると時間がかかります。ちょっと修正して動かしてデバッグしてというのを繰り返すのにビルドの度に 5 分もかかったら全く集中できませんね。ビルドしてる間にどうせどっかのサイトでも見て暇潰してると、終わったあとにそういやビルドしたんだっけこれからするんだっけ？と分かんなくなってまたビルドし始めるんですよ。それからまた 5 分もかかったら無限ループだね！

ファイルの変更があった場合にだけ、必要な部分をコンパイルするようにすれば毎度そんなに時間はかからなくなります。ソースとコンパイル結果それぞれのファイルの変更日時を見て、ソースの方が新しければ変更されているのでコンパイルするようにしましょう。これで完璧！……と言いたいところですが、ソースをコンパイルしてライブラリを作り、そのライブラリを使う実行ファイルをビルドしたいとしたらどうでしょう。ソースが実行ファイルより新しければライブラリから作り直せばいい？もっと多段になってたり、複数のライブラリを作って使う実行ファイルだったらどうする？うわめんどくせぇ……。

というわけでその辺りを解決してくれるのがビルドツールです。

基本的には、入力ファイルと出力ファイルとそのビルド方法 (作り方) を書いておくと、出力ファイルより入力ファイルが新しければビルドを実行してくれる、というだけのツールです。また、入力ファイルが他のファイルから作られる出力ファイルである場合には、さらに元のファイルから先に更新されているか確認してビルドするという依存関係の解決というのもやってくれます。

書き方や細かい機能なんかに差異はありますが、どのツールもやってくれることは基本的に同じです。

主にプログラムのビルドに使われるツールではありますが、やることはファイルが更新されていたらコマンドを実行する、というだけなので、ビルド自体だけでなくいろいろと使うことができます。

たとえばドキュメント。Markdown 等のテキスト形式で書いといて、リリース時には HTML なんかに変換するということもあるでしょう。Markdown ファイルが更新されていたら HTML ファイルに変換！というのはまさにビルドツールで実行させると良い例ですね。てか普通にコンパイルしてるのと同じですしね。

たとえばリリース用のパッケージ。インストーラのビルドなんかは明らかにビルドって言っちゃってるのでたぶんなんらかのビルドツールを使うことになると思いますが、それ以外にも zip ファイルにつっこむだけでもビルドツールが使えます。ビルド結果の実行ファイルなりライブラリファイルと、ドキュメントファイルを入力にして、zip ファイルを出力として zip コマンドを実行します。この時、zip ファイルにつっこもうとした実行ファイルより、ソースファイルが新しければ、先にビルドして実行ファイルを更新してから zip ファイルに入れて欲しいですよね。そういう処理をビルドツールは (ちゃんと書いてやれば) 自動的に解決して実行してくれます。あら便利。

プログラムのビルド以外にもたびたび使える場面が出てくるビルドツール、是非 MSBuild を、とは言いませんが何か一つくらいは使えるように覚えておくと便利でしょう。

1.2. MSBuild の特徴

さてあらためまして。MSBuild はマイクロソフト製のビルドツールです。

元々は C# や VB.NET 用に作られたものですが、特にそれら専用というわけでもありません。Visual Studio 2010 以降は Visual C++ でも使われるようになりました。

従来はプロプライエタリな Windows 専用のツールでありましたが、近年オープンソース化されて Windows 以外でも使えるように移植され、.NET Framework だけでなく mono や .NET Core でも動くようになりました。

どこで使われているかというと、Visual Studio でよく使われています。実は Visual Studio のプロジェクトファイルは MSBuild の形式で書かれており、Visual Studio でビルドボタンをポチると MSBuild が動いています。

また、.NET Framework のリメイク版とも言える .NET Core でもマルチプラットフォームに拡張されて利用されています。.NET Core で使われるために多少ながら手で書きやすいように拡張もされました。嬉しいですね。

MSBuild の特徴としては以下のようなものがあります。

- Visual Studio に標準添付
- Visual Studio のプロジェクトファイルに使われている
- 拡張用の DLL を読み込んで拡張可能
- XML でちょっと書きづらい

  なんかあんまり良いこと書いてない気がしますが気のせいですよ、たぶん。

なんといっても Visual Studio に標準添付というのは強いですね。いやべつに Visual Studio が必要というわけでなくコマンドラインだけでインストールもできるんですが、とにかく .NET のビルド環境を入れると必然的に入ってるというのは環境構築が楽で使いやすい点です。従来は .NET Framework に標準添付だったので、Windows のかなり広い環境で使えたようですが、今は開発環境にしか付いてこなくなったので残念ながらどのマシンで使えるものではありません。開発環境以外でそんなに使う必要もないですけどね。

Visual Studio のプロジェクトファイルに使われてるからなんなんだって話ですが、自分で手で書いた MSBuild のプロジェクトファイルから Visual Studio のプロジェクトファイルを呼び出すのが簡単にできます。Visual Studio のプロジェクトファイルには他から呼び出される前提の機能なども入っているので、単純にバッチファイルなどからビルドするより細かい制御ができます。そしてなんと、MSBuild のプロジェクトファイルのいじり方を覚えると、Visual Studio のプロジェクトファイルを手でいじることができるようになり、GUI からでは設定しづらいこと、設定できないことが直接書けるようになります！全くおすすめできませんけどね！！

機能の拡張に関しては .NET 用の言語 (C# や VB.NET 以外にも F# など .NET のアセンブリが吐ければなんでも) でちょこっとプログラムを書くだけで拡張することができます。拡張といっても実行するコマンドを自分で作るくらいのことなんですけど、複雑な処理は C# や VB.NET などの使い慣れた言語で書いといて DLL を呼び出すだけで済ませられるのは便利です。また、ちゃんと作ればこれらの拡張用 DLL もマルチプラットフォームでそのまま動くのは嬉しいですね。

XML でちょっと書きづらいというのはまあそのままです。XML は手で書けないものではないですが、ちょっとめんどいですね。あまりにも面倒だという人は良い XML エディタでも探すと良いんじゃないでしょうか。

一方で一般的な XML ではあるので、まず基本的な文法が分からなくて困るということはあまり無いでしょう。Makefile とかまず書き方がわからないもんね！もちろん属性やテキストに MSBuild 独自の書き方は入ってくるので分からん時は分からんのですが。

上記のような特徴がありますので、MSBuild は Visual Studio や .NET Core で作ったプロジェクトのバッチビルドや CI 環境で使われることが多く、やはりそういうところで使うのが良いでしょう。もちろん汎用的なツールですので、プログラム自体のビルドだけでなく、ドキュメントのビルド等にも使えますし、配布用パッケージを作ったりもできます。

とはいえ、特別使いやすいってわけでもないので、なんでもかんでも MSBuild でこなそうとするのは間違いです。MSBuild から他のコマンドを呼び出したり、逆に他のツールから MSBuild を呼び出すというのは当然できますので、適材適所で使っていきましょう。

2. 最初の一歩

ここから MSBuild を実際使っていくところに入ります。

最初にインストールから、動作を確認できる小さいファイルを作るところまでやっていきましょう。

2.1. インストール

まずは MSBuild 自体が無いといけないのでインストールしましょう。

2.1.1. Windows

Windows ではいろいろ方法がありますが、一番普通なのは Visual Studio を入れることでしょう。

Visual Studio の公式サイト[2]からダウンロードして入れることができます。個人で使うなら Community 版が使えますが、仕事で使う場合には Community 版は使えないことが多いでしょう。Professional 版を買うなりしてください。Express でもいいんですが、2015 までしかないので最新の MSBuild は使えません。

Visual Studio Code には MSBuild は付いてこないことに気をつけてください。

CI サーバ等でバッチビルドするだけなのに Visual Studio 入れたくなーい、という場合には Build Tools が使えます。これはコマンドラインのビルドツールだけインストールできるもので、Visual Studio 本体は入れずに MSBuild や各種ビルドツールが使えます。上記の Visual Studio の公式サイトのダウンロードから探すと Build Tools for Visual Studio 2017 というのがあるのでこれをインストールしましょう。ライセンスについては詳しく書いていないのですが、どうも Visual Studio のおまけ扱いらしく、Visual Studio と同等のライセンスが必要になるようです[3]。仕事で Community 版使えないよって人はやはり Professional 版以上のライセンスを買う必要があるのでお気をつけください。

Visual Studio のライセンスが使えないよって場合や、普通の .NET Framework じゃなくて .NET Core で開発するよって場合には .NET Core SDK が使えます。.NET Core のサイト[4]から SDK をダウンロードして入れましょう。基本的には一番新しいのを入れれば大丈夫です。

.NET Core SDK はコマンドラインのツールしか入っていませんが、最新の Visual Studio であれば .NET Core の対応もされています。また、Visual Studio Code や Rider といった IDE でも対応しているので好きなものを使いましょう。もちろんコマンドラインのツールで開発してもなんら問題

2 https://www.visualstudio.com/ja/

3 https://social.msdn.microsoft.com/Forums/vstudio/en-US/08d62115-0b51-484f-afda-229989be9263/license-for-visual-c-2017-build-tools?forum=visualstudiogeneral

4 http://www.microsoft.com/net/core

ありません。

他には何を思ったか Windows に mono を入れた人は MSBuild が使えます。Windows 用の mono は公式サイト[5]からダウンロードできます。mono の 5.0 より古いバージョンでは MSBuild でなく互換の xbuild というツールが入っていますが、動きが違うと思いますのでなるべく新しいのを入れて MSBuild を使いましょう。

起動方法ですが、Visual Studio や Build Tools の場合はスタートメニューに「開発者コマンドプロンプト for Visual Studio 2017」といったようなものが追加されていますので、それを起動してください。mono だと「Open Mono Command Prompt」といったものです。いずれも `MSBuild` と打つと起動できます (小文字でも可)。バージョンが出るので思った通りの物が起動しているか確認しておきましょう。Visual Studio 2017 世代だとバージョンは 15.x ですので、それ以上であれば大丈夫でしょう。

```
C:\Users\kumaryu\Source>msbuild
.NET Framework 向け Microsoft (R) Build Engine バージョン 15.3.409.57025
Copyright (C) Microsoft Corporation.All rights reserved.

MSBUILD : error MSB1003: プロジェクト ファイルまたはソリューション ファイルを指定してください。現在の作業ディレクトリは プロジェクト ファイルまたはソリューション ファイルを含んでいません。
```

```
C:\Program Files\Mono>msbuild
Microsoft (R) Build Engine version 15.2.0.0 (xplat-2017-02/c2edfeb Thu May 18 13:58:03 EDT 2017)
Copyright (C) Microsoft Corporation. All rights reserved.

MSBUILD : error MSB1003: Specify a project or solution file. The current working directory does not contain a project or solution file.
```

.NET Core SDK はちょっと起動方法が違っており、MSBuild.exe は入ってきません。たぶん PATH に勝手に `dotnet` コマンドが追加されていますので、適当にコマンドプロンプトを開いて `dotnet msbuild` と打つと起動します。.NET Core SDK の 2.0 時点ではバージョンは 15.3 だったので、それ以上のバージョンだったら大丈夫でしょう。

```
C:\Users\kumaryu\Documents\hoge>dotnet msbuild
.NET Core 向け Microsoft (R) Build Engine バージョン 15.3.409.57025
Copyright (C) Microsoft Corporation.All rights reserved.

MSBUILD : error MSB1003: プロジェクト ファイルまたはソリューション ファイルを指定してください。現在の作業ディレクトリは プロジェクト ファイルまたはソリューション ファイルを含んでいません。
```

5 http://www.mono-project.com/

2.1.2. macOS

macOS では mono か .NET Core SDK を入れると MSBuild が使えるようになります。.NET Framework 向けの開発をする場合は mono、.NET Core 向けの開発をする場合は .NET Core SDK を入れると良いでしょう。

Visual Studio for Mac を入れると mono が勝手に入り、.NET Core SDK もオプションですがいっしょに入れることができます。どちらも使う場合には Visual Studio for Mac を入れるのも手でしょう。Visual Studio の公式サイト[6]からダウンロードして入れることができます。ただし Windows 版と同様に、個人で使うなら Community 版が使えますが、仕事で使う場合には Community 版は使えないかもしれませんので利用条件は確認しましょう。

mono を単体で入れる場合には mono は公式サイト[7]からダウンロードできます。mono の 5.0 より古いバージョンを使っている人は MSBuild でなく互換の xbuild というツールが入っていますが、動きが違うと思いますのでなるべく mono の 5.0 以降を入れて MSBuild を使いましょう。

.NET Core SDK を単体で入れるには、.NET Core のサイト[8]から SDK をダウンロードしましょう。基本的には一番新しいのを入れれば大丈夫です。

どれをインストールした場合も勝手に PATH に追加されていると思います。ターミナルを開いてコマンドを打ちこんでください。

mono を入れた場合には `msbuild` で起動できます。全て小文字で打ち込んでください。

```
MacMini:~ kumaryu$ msbuild
Microsoft (R) Build Engine version 15.2.0.0 (xplat-2017-02/c2edfeb Thu May 18
13:58:03 EDT 2017)
Copyright (C) Microsoft Corporation. All rights reserved.

MSBUILD : error MSB1003: Specify a project or solution file. The current working
directory does not contain a project or solution file.
```

.NET Core SDK では `dotnet` コマンドが追加されていますので、ターミナルで `dotnet msbuild` と打つと起動します。

```
MacMini:~ kumaryu$ dotnet msbuild
.NET Core 向け Microsoft (R) Build Engine バージョン 15.3.409.57025
Copyright (C) Microsoft Corporation.All rights reserved.

MSBUILD : error MSB1003: プロジェクト ファイルまたはソリューション ファイルを指定してください。現在の作業ディレクトリはプロジェクト ファイルまたはソリューション ファイルを含んでいません。
```

6 https://www.visualstudio.com/ja/

7 http://www.mono-project.com/

8 http://www.microsoft.com/net/core

2.1.3. Linux 等

mono か .NET Core SDK を入れると使えます。

mono はパッケージマネージャを探すときっとあると思います。ディストリビューションによっては開発用のパッケージが別だったりしますので、mono の開発環境が揃うやつを入れましょう。ただし、標準のパッケージリポジトリにある mono はなかなか新しいのが入らない可能性があります。MSBuild が入るのは mono の 5.0 以降ですので、それより古いバージョンが入るようであれば、別途インストールしてください。

主要なディストリビューションでの新しいバージョンのインストール方法は mono の公式サイト[9]に記載があります。最悪は自分でビルドするはめになるかもしれません。幸い mono のビルドはそんなに難しくないです。

.NET Core SDK も標準のリポジトリにあれば良いですが、無い場合は .NET Core のサイト[10]にインストール方法がありますのでこれで入れましょう。標準リポジトリにもここへの記載もないディストリビューションでは諦めて mono を使った方が良いでしょう。.NET Core SDK のビルドは結構大変そうです。.NET Core SDK は 1.0 以降であれば MSBuild が付いてくるので最新でなくても MSBuild を使うだけなら大丈夫です。

起動方法は macOS と同じです。環境構築が大変だったので実行例は省略します……。

2.1.4. ソースからビルドする

GitHub にある MSBuild のページ[11]からソースがダウンロードできます。

ビルド方法もここに書いてありますので頑張ってください……。

2.2. MSBuild コマンドの使い方

MSBuild コマンドは基本的にコマンドラインから起動します。Visual Studio 等の IDE から知らないうちに使われてることもありますが、手で起動する時はコマンドラインから起動します。残念ながら GUI から MSBuild だけを簡単に起動するツールはありません。

コマンドラインといっても、MSBuild はそんなに指定するオプションも無いので簡単に使い方を覚えましょう。

まず単純な形式は以下の通りです。

```
msbuild [オプション] [プロジェクトファイル | ディレクトリ]
dotnet msbuild [オプション] [プロジェクトファイル | ディレクトリ]
```

9 http://www.mono-project.com/

10 http://www.microsoft.com/net/core

11 https://github.com/Microsoft/msbuild

コマンド名は既にインストールのところで紹介した通り msbuild か dotnet msbuild です。MSBuild.exe でもいいですが、Windows だと大文字小文字は区別されませんし、Windows 以外だと小文字で統一されているので msbuild と打つとどこでも動いて良いでしょう。.NET Core SDK で使う場合には dotnet msbuild として起動してください。

プロジェクトファイルを指定した場合は、そのプロジェクトファイルをビルドしようとします。ディレクトリを指定した場合は、そのディレクトリ内で *.proj や *. ～ proj なファイルを探し、見つかればそのファイルをビルドしようとします。一つも見つからなかったり、複数見つかった場合はエラーになるので、その時はファイル名を一つだけ明示的に指定してください。どちらも指定しなかった場合は、カレントディレクトリを指定したものとしてプロジェクトファイルを探してビルドしようとします。

オプションの頭は / でも - でも指定できます。Unix 形式のコマンドに慣れてる方でも安心。ただし長いオプションでも -- で始まるのはだめなので気をつけてください。

指定できるオプションですが、まずこれは覚えておきたいのが /help オプションです。/? やら /h やら -help やら -? やら -h でも大丈夫ですが、とりあえずこれだけ指定してみれば使い方が出てくるので、なんかオプションを忘れたらまず打ってみましょう。

以下にはよく使うオプションを簡単に説明します。他にもありますが、あとはヘルプなどドキュメントを見てみてください。

オプション	説明
/target: ターゲット名 /t: ターゲット -target: ターゲット名 -t: ターゲット	ビルドするターゲットを指定します。 複数のターゲットをビルドする場合にはカンマかセミコロンで区切って指定できます。もしくはオプションを複数回指定してください。 例 :/target:Resources;Compile
/property: プロパティ名 = 値 /p: プロパティ名 = 値 -property: プロパティ名 = 値 -p: プロパティ名 = 値	プロジェクト内で使われるプロパティの値を上書きします。 複数のプロパティと値を指定するにはセミコロンで区切ってください。もしくはオプションを複数回指定してください。 例 :/property:WarningLevel=2;OutDir=bin\Debug\
/verbosity: レベル /v: レベル -verbosity: レベル -v: レベル	表示されるログの量です。 指定できるレベルは少ない順に q[uiet]・m[inimal]・n[ormal]・d[etailed]・diag[nostic] です。 省略時は msbuild では normal ですが、dotnet msbuild だと minimal です。 例 :/verbosity:quiet・/v:n
/nologo	実行時に先頭に出てくる著作権情報を表示しません。

表 1. msbuild コマンドのオプション

ログ関係などは特に沢山オプションがあるんですが、普段使うのはこれだけでしょう。

特によく使うのは /target と /property です。指定する詳しい意味はまたあとでも解説するので大丈夫ですが、とりあえずこの 2 つだけは指定方法を覚えておくと良いでしょう。あとはヘルプでもみて思い出せばいいので忘れてもかまいません。

2.3. プロジェクトを書いてみる

インストールと起動は無事できたでしょうか？さすがに出来たよな？うん、できたのでプロジェクトファイルを書いてみましょう。

いきなりプロジェクトファイルと言いましたが、MSBuild で実行するビルドの設定を書くファイルはプロジェクトファイルと呼びます。プロジェクトファイルは XML で書きますが、ここで XML を解説してると終わらないので XML の書き方は別で調べてきてください。とはいえ XML としては最低限の機能しか使わないのでタグの書き方だけ分かれば十分です。

次のコードを適当なファイル名で保存してください。ファイル名はなんでもいいのですが、MSBuild のなんでもない (C# とか特定の言語に関連しない) プロジェクトファイルの拡張子は .proj にすることが多いのでここでもそうしましょう。中身は XML なので、エディタによっては .xml の方が編集しやすいこともあります。その場合は .xml でもかまいません。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="Build">
    <Message Text="Hello!"/>
  </Target>
</Project>
```

ここでは hello.proj という名前で保存したとしましょう。改行コードはなんでもいいんですが、文字コードは 1 行目にあるように UTF-8 にしてください。実は UTF-8 であれば 1 行目の XML 宣言は省略してもかまいません。

できたら次のコマンドで実行します。

```
msbuild hello.proj
```

.NET Core SDK を使ってる場合は次のコマンドを使ってください。

```
dotnet msbuild /v:n hello.proj
```

.NET Core SDK の msbuild は標準でメッセージが少ないモードになっているので /v:n オプションで普通のメッセージを出すようにしています。/v:n は /verbosity:normal の略です。.NET Framework 用の MSBuild でも同じオプションは使えます。

macOS や Linux なんかで /v:n とかスラッシュでオプションを指定するのが気持ち悪いなぁという場合は -v:n でも大丈夫です。他のオプション指定時も頭の / は全部 - で指定しても通ってくれます。安心。

プロジェクトファイル名として hello.proj を渡していますが、MSBuild は . 〜 proj という拡張子のファイルがカレントディレクトリに 1 つしかなければそれを自動的に使うので、hello.proj の指定は省略することもできます。拡張子が違ったりプロジェクトファイルが複数ある場合は明示的に指定しましょう。

```
> msbuild hello.proj
.NET Framework 向け Microsoft (R) Build Engine バージョン 15.3.409.57025
Copyright (C) Microsoft Corporation.All rights reserved.

2017/09/30 23:18:55 にビルドを開始しました。
ノード 1 上のプロジェクト "hello.proj" (既定のターゲット)。
Build:
  Hello
プロジェクト "hello.proj" (既定のターゲット) のビルドが完了しました。


ビルドに成功しました。
    0 個の警告
    0 エラー

経過時間 00:00:00.07
```

実行できるとこんな感じの表示が出ると思います。ビルドに成功しました。おめでとうございます！

……いやべつにおめでたくはないか？まあちょっとずつ説明していきましょう。

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
</Project>
```

まず一番外側の Project 要素ですが、ここにはプロジェクトファイルの設定なんかを書きます。今回は特に設定は書いてませんが、xmlns という XML の名前空間宣言だけ書いてあります。これは MSBuild のプロジェクトファイルであることを表しており、いつも同じ値を指定するので、なんか知らんけどコピペして使うものと覚えておいて大丈夫です。実は MSBuild のバージョン 15 以降だとこれが無くても動きます。Visual Studio 2015 などを使っている場合には無いとエラーになります。古い MSBuild で動かす予定が無い場合には省略しちゃってかまいません。

```
<Target Name="Build">
</Target>
```

ここでは Build という名前のターゲットを宣言しています。ターゲットってのは一連の処理 (コマンド) をまとめたものです。

プロジェクトには複数のターゲットを含めることができますが、特にどこにも指定がないと一番上のターゲットが実行されるようです。今回は一つしかターゲットがないので Build が実行されます。

```
<Message Text="Hello!"/>
```

ターゲットの中のこれが実行されるコマンドで、タスクと呼びます。ここでは Message タスクてのを呼び出しています。これは Text 属性で指定した文字列をログに出力するタスクです。Message タスク一つしか書いてませんが、もちろんターゲットの中には複数のタスクを書くことができます。その場合は上から順番に実行されます。

とりあえず最初の一歩ということで最小限のプロジェクトを作って実行してみました。どうだったでしょうか。

は？これだけじゃどうもうこうもねえよ。というのが大方の感想だと思いますので、次からもうちょいまともな形のプロジェクトを作っていきましょう。

2.4. ソリューションファイルについて

本題に入る前にちょっと寄り道。

MSBuild コマンドには、実はソリューションファイルを渡してビルドすることも出来ます。

ソリューションファイルというのは Visual Studio で使われる、複数のプロジェクトファイルをまとめるファイルですね。.sln てやつです。複数のプロジェクトファイルを一気に開けるだけでなく、各プロジェクト間の依存関係も記述されています。

msbuild hoge.sln のように、プロジェクトファイルの代わりにソリューションファイルを渡すと、ソリューションファイルに書かれているプロジェクトを一度にビルドすることができます。

じゃあそれも書こうかと言い出したいところですが、ソリューションファイルのフォーマットは非公開です。まあテキストファイルですし、そんな難しいことは書いてないので開いてみればなんとなくはわかるのですが、フォーマットに関するドキュメントはほとんどないので手で書くのは全くおすすめできません。おとなしく Visual Studio で作るか .NET SDK の dotnet コマンドで操作しましょう。

dotnet コマンドでソリューションファイルを作るのは dotnet new sln でできます。ファイル名はカレントディレクトリ名 .sln になります。名前を指定したい場合は dotnet new sln -n 名前としてください。名前 .sln が出来ます。名前に .sln まで付けると名前 .sln.sln というファイルが出来てしまうので注意しましょう。

作った (もしくは既存の) ソリューションファイルの操作は dotnet sln のサブコマンドで行います。

dotnet sln ソリューションファイル名 list で、ソリューションファイルに入っているプロジェクトを一覧表示します。

dotnet sln ソリューションファイル名 add プロジェクトファイル名でソリューションファイルに指定したプロジェクトを追加、dotnet sln ソリューションファイル名 remove プロジェクトファイル名でソリューションファイルから指定したプロジェクトを削除します。いずれもプロジェクトファイル名は複数指定することもできます。

`dotnet sln` ではソリューションファイル名を省略するとカレントディレクトリにあるソリューションファイルを勝手に使ってくれるので、ソリューションファイルが一つしか無い場合は省略するのが便利でしょう。

ソリューションファイルは複数のプロジェクトファイルを扱うのに便利ですが、残念ながら追加できるプロジェクトファイルに制限があります。プロジェクトの種類 (GUID) を決定できるファイルでないと追加できません。C# のプロジェクトなど Sdk が指定されているプロジェクトファイルだと追加できるのですが、何かビルドするわけでもない素のプロジェクトファイルだと追加できません。おそらく Visual Studio が判別するのに使ってるのかと思いますが、素のプロジェクトファイルは Visual Studio で扱えませんしね……。

まあ MSBuild から使う場合には必ずしもソリューションファイルは便利でもありません。単に複数のプロジェクトファイルをビルドするプロジェクトファイルを作るのが良いでしょう。この本に一通り目を通せばそういったプロジェクトファイルが書けるようになっているはずです！

3. プロジェクトの中身を見てみよう - プロジェクトファイルの説明

ここからはちゃんと実用になる形のプロジェクトファイルから中身を見ていきます。

3.1. 小さいプロジェクトファイル

まずは一通りの物が入った小さいプロジェクトファイルを見ていきましょう。前の章で作ったのはあまりにも最小限すぎてほぼ意味ありませんでしたしね。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="12.0" DefaultTargets="Build">
  <!-- プロパティ(変数)の定義 -->
  <PropertyGroup>
    <Pandoc>pandoc</Pandoc>
    <OutputFormat>docx</OutputFormat>
  </PropertyGroup>
  <!-- アイテム(ファイル一覧)の定義 -->
  <ItemGroup>
    <Inputs Include="hoge.md" />
    <Inputs Include="fuga.md" />
  </ItemGroup>
  <!-- ターゲットの定義 -->
  <Target Name="Build">
    <Message Text="Converting @(Inputs) to $(OutputFormat)" />
    <Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Inputs.
Filename).$(OutputFormat) @(Inputs)" />
  </Target>
  <Target Name="Clean">
    <Message Text="Deleting @(Inputs->'%(Filename).$(OutputFormat)')" />
    <Delete Files="@(Inputs->'%(Filename).$(OutputFormat)')" />
  </Target>
</Project>
```

いきなり複雑になりましたが、少しずつ説明していきますので大丈夫です。

とりあえずどんな動作をしているかだけ説明しますと、pandoc というコマンド[12]で、文書形式である Markdown の hoge.md と fuga.md を Word の docx 形式に変換しています。この例はプログラムのビルドではないですが、ドキュメントのビルドを行っているものですね。コマンドこそ違い

12　文書ファイルを他のフォーマットに変換するオープンソースなソフトです。Pandoc のサイト (http://pandoc.org/) からダウンロードできます。

ますがソースから何か他の形式に変換するという意味において、プログラムをコンパイルしてるのとやってることは同じです。

また、生成する Build ターゲットだけでなく、生成したものを削除する Clean ターゲットも追加しています。

これを実行するには pandoc コマンドをパスが通った場所に起き、適当な内容の hoge.md と fuga.md が必要になります。外部のツールが不要な例ができれば良かったんですが、まともな例になると外部コマンド無しではちょっと難しいですね。

Build を実行すると以下のようになります。

```
> msbuild
.NET Framework 向け Microsoft (R) Build Engine バージョン 15.3.409.57025
Copyright (C) Microsoft Corporation.All rights reserved.

2017/10/01 16:55:17 にビルドを開始しました。
ノード 1 上のプロジェクト "minimal.proj" (既定のターゲット)。
Build:
  Converting hoge.md;fuga.md to docx
  pandoc -t docx -o hoge.docx hoge.md
  pandoc -t docx -o fuga.docx fuga.md
プロジェクト "minimal.proj" (既定のターゲット) のビルドが完了しました。


ビルドに成功しました。
    0 個の警告
    0 エラー

経過時間 00:00:00.35
```

真ん中あたりに実行結果が出ています。Build: のところで Build ターゲットを実行したことを示しており、その下に実行したコマンド (タスク) が表示されています。最終的には警告もエラーもなく終了しています。

Clean を実行すると以下のようになります。

```
> msbuild /t:Clean
.NET Framework 向け Microsoft (R) Build Engine バージョン 15.3.409.57025
Copyright (C) Microsoft Corporation.All rights reserved.

2017/10/01 16:57:02 にビルドを開始しました。
ノード 1 上のプロジェクト "minimal.proj" (Clean ターゲット)。
Clean:
  Deleting hoge.docx;fuga.docx
  ファイル "hoge.docx" を削除しています。
  ファイル "fuga.docx" を削除しています。
プロジェクト "minimal.proj" (Clean ターゲット) のビルドが完了しました。


ビルドに成功しました。
    0 個の警告
    0 エラー

経過時間 00:00:00.09
```

まず一番上のコマンドラインが違うことに気付いたでしょうか。Clean はデフォルトじゃないターゲットなので、実行する場合には msbuild /t:Clean のように /t: オプションで指定します。/t: は省略形で、/target: と指定してもかまいません。

こちらも実行結果は真ん中あたりに出ています。Clean: のところで Clean ターゲットを実行したことがわかり、その下に実行したコマンド (タスク) が表示されています。当然のことではありますが Build の時とは違ったものを実行しています。

3.2. ファイルの構成を見てみよう

何をするプロジェクトファイルかわかったところで、この小さいプロジェクトファイルを構成しているものについて一つずつ見ていきましょう。

3.2.1. プロジェクトの定義

まずはプロジェクトファイルに必要なのはプロジェクト要素です。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="12.0" DefaultTargets="Build">
</Project>
```

こんな感じです。文字エンコーディングが UTF-8 なら最初の XML 宣言は無くても認識してくれます。

Project 要素にある xmlns=" 〜 " の部分は XML の名前空間の宣言です。これは固定値なので、常に http://schemas.microsoft.com/developer/msbuild/2003 を指定します。MSBuild

の 15.0 以降なら無くても動きますが、Visual Studio 2015 まで付属の MSBuild 14 までで動かそうとすると必要になります。MSBuild 15 以降で .NET Core 用のプロジェクトを作ると既定では省略されるようになっています。使いたい MSBuild のバージョンによって付けるかどうか決めれば良いでしょう。

ToolsVersion 属性は動作する最小の MSBuild バージョンを指定しています。特に指定しなければどのバージョンでも動きますが[13]、指定したより古いバージョンの MSBuild で動かそうとするとエラーが出ます。ここでは特に指定する理由もないのですが、例として 12.0(Visual Studio 2013 付属のバージョン) を指定しています。

省略できるものを省略すると次のような感じになります。

```
<Project DefaultTargets="Build">
</Project>
```

あらすっきり。MSBuild 15 以降で使うだけならこれでも大丈夫でしょう。なんなら ToolsVersion に 15.0 を指定しても良いかもしれません。

DefaultTargets 属性は MSBuild をターゲット指定無しで起動した時に実行される、既定のターゲットを指定します。指定がなければ一番上に定義したターゲットが勝手に使われるようですが、うっかり入れ替えたり上に他のターゲットを追加して思ったのと違うのが実行されるとかあるとなんなので、できればいつも指定しといた方がいいでしょう。

実行例でも紹介しましたが、ここの DefaultTargets で指定したのと違うターゲットを実行したい場合には、msbuild /t:Clean のように /t: オプションで指定してください。

DefaultTargets 属性は複数形であること (DefaultTarget ではない！) に気をつけてください。ターゲットは一度に複数実行することもできるので、DefaultTargets="Clean;Build" のようにセミコロンで区切って複数書くと、既定で複数のターゲットを実行させるようにもできます。既定のターゲットだけでなく、オプションで実行するターゲットを指定する時も、msbuild /t:Clean;Build のようにセミコロンで区切って複数書くことができます。

3.2.2. プロパティとアイテムの定義

次はプロパティとアイテムの定義です。

```
<!-- プロパティ(変数)の定義 -->
<PropertyGroup>
  <Pandoc>pandoc</Pandoc>
  <OutputFormat>docx</OutputFormat>
</PropertyGroup>
```

まずはプロパティ。

13　もちろんそのバージョンに無い機能を使ってない限り、です。新しいバージョンじゃないと動かないのが分かってる場合に指定すると良いでしょう。

念のため書いておくと <!-- --> は XML のコメントですよ。MSBuild では単に無視されますので、あとで見て分からなくなるようなところには適宜入れときましょう。

PropertyGroup 要素の中にプロパティを定義します。いやそもそもプロパティってなんだよって話ですね。初めて出てきたやつですね。

コメントにも書いてある通り、プロパティは変数です。後の方で $(Pandoc) とか $(OutputFormat) とか出てるのに気付いた人も居るかもしれませんが、その部分がここで定義した値で置き換えられます。

PropertyGroup 要素の中のタグ名はプロパティ名になりますが、プロパティ名は好きにつけられます。つまり PropertyGroup 要素の中のタグはこの名前で書かないといけないというのは無く、あとで参照したい名前で好きに付けてかまいません。ただし、半角英数とアンダーバー以外の文字は入らないので、残念ながら日本語は通りません。

プロパティのタグの中身はプロパティの値になります。型とかはないので全て文字列になります。こちらはただの値なので半角英数とアンダーバーだけでなく、スペースとか日本語を入れてもいいですが、使う時は単に文字列としてそのまま扱われます。XML で意味のある文字を値として入れる場合は文字参照とか (> みたいなの) でエスケープしてください。

```
<!-- アイテム(ファイル一覧)の定義 -->
<ItemGroup>
  <Inputs Include="hoge.md" />
  <Inputs Include="fuga.md" />
</ItemGroup>
```

こっちはアイテムの定義です。プロパティとだいたい同じに見えますが微妙に違います。

ItemGroup 要素の中でアイテムを定義します。こちらもコメントに書いてある通りですが、ファイル一覧を指定するものです。ここでは入力ファイルの一覧を定義してます。

ItemGroup 要素の中のタグ名はプロパティと同じようにアイテム名になるので、好きに決めてかまいません。例で Inputs とかそれらしい要素名になっていますが、好きな名前を付けられるので Markdown なんかにしちゃっても問題ありません。しかしやっぱり半角英数とアンダーバーくらいしか入らないので、残念ながら日本語にするとエラーになってしまいます。

プロパティと違うところは、まず見た目を挙げると、タグ内のテキストでなく Include 属性でファイル名を指定しているところ、同じ名前の要素が並んでるところ、でしょうか。アイテムは複数のファイル名を一つのアイテム名でまとめて参照できる点がプロパティと大きく違います。

例では Inputs という名前のアイテムに対して 2 回定義があってそれぞれ別なファイル名を指定していますが、2 回以上の定義は上書きでなく追加になるため、これで Inputs アイテムを参照すると hoge.md と fuga.md の両方が取得できます。

Include 属性はファイル名を指定していますが、ここには一つのファイル名だけでなく、複数のファイルを一度に指定することができます。セミコロンで区切って複数のファイル名を書いたり、ワイルドカードで指定もできます。詳しくはまた後で解説します。

アイテムの参照については後の方で @(Inputs) とか %(Inputs.Filename) とかで参照されてます。

3.2.3. ターゲットの定義とタスク

ターゲットの定義を見ていきましょう。ターゲットってのは前でさらっと説明しましたが、一連の処理 (コマンド) をまとめたものです。

```
<!-- ターゲットの定義 -->
<Target Name="Build">
  <Message Text="Converting @(Inputs) to $(OutputFormat)" />
  <Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Inputs.
Filename).$(OutputFormat) @(Inputs)" />
</Target>
```

Target 要素でターゲットを定義します。

Name 属性は必須です。ここにターゲットの名前を書いてください。ターゲットの名前はプロパティやアイテムの名前と違って日本語も使えるのですが、コマンドラインから指定することもちょくちょくあるので日本語にすると入力がめんどくさそうです。

中身には処理を XML の要素で書きます。処理は上から順番に実行されます。それだけ。

Message や Exec といった要素が実際に実行される処理です。こいつらはタスクと呼びます。プロパティやアイテム、ターゲットだのの定義を見てきたんでこいつも定義かと思われますが、ここではタスクを定義してるわけじゃなくて使ってるだけですね。ターゲットの中では定義でなく使うタスクを書きます。ここで使っている Message や Exec タスクは標準で使えるタスクなので定義の必要がなく使えるものです。

タスクに渡すパラメータは属性で指定します。上の例では Text や Command 属性がそうですね。パラメーターはタスク毎に異なります。

使ってるのだけ説明しておくと、Message タスクは Text パラメータで指定した文字列をログに出力します。Exec タスクは Command パラメータで指定した文字列をコマンドとして実行します。

上から順番に実行されるので、ここでは Message タスクでこれからコンバートすることをログに出力し、それから実際に Exec タスクで外部のコマンドを実行してファイルの変換を行っています。

Text パラメータや Command パラメータの中に書いてある文字列が @(～) やら $(～) と複雑なことになっていますが、それぞれプロパティの中身やアイテムの中身がここに展開された文字列になるだけです。細かい説明はまた後ほど。

形は同じですが、一応 Clean ターゲットの方も見ておきましょう。

```
<Target Name="Clean">
  <Message Text="Deleting @(Inputs->'%(Filename).$(OutputFormat)')" />
  <Delete Files="@(Inputs->'%(Filename).$(OutputFormat)')" />
</Target>
```

Clean という名前のターゲットを定義して、中では Message タスクと Delete タスクを順番に実行しているだけですね。Delete タスクは名前から予想が付く通り、Files パラメータで指定したファイルを削除するものです。なにやらパラメータの中に書いてあるのがとても複雑なことになっていますが、これもまた後で説明しましょう。

3.2.4. プロジェクトファイル全体を再確認

一通り中身を見てみたので、もう一度全体を確認しましょう。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="12.0" DefaultTargets="Build">
  <!-- プロパティ(変数)の定義 -->
  <PropertyGroup>
    <Pandoc>pandoc</Pandoc>
    <OutputFormat>docx</OutputFormat>
  </PropertyGroup>
  <!-- アイテム(ファイル一覧)の定義 -->
  <ItemGroup>
    <Inputs Include="hoge.md" />
    <Inputs Include="fuga.md" />
  </ItemGroup>
  <!-- ターゲットの定義 -->
  <Target Name="Build">
    <Message Text="Converting @(Inputs) to $(OutputFormat)" />
    <Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Inputs.
Filename).$(OutputFormat) @(Inputs)" />
  </Target>
  <Target Name="Clean">
    <Message Text="Deleting @(Inputs->'%(Filename).$(OutputFormat)')" />
    <Delete Files="@(Inputs->'%(Filename).$(OutputFormat)')" />
  </Target>
</Project>
```

プロジェクトの中でプロパティ (変数) とアイテム (ファイル一覧) とターゲットを定義し、ターゲットの中では上から順番に実行するタスクを並べるだけです。

タスクのパラメータがなにやら怪しいことになっている以外はそう難しくないと思います。あ、プロパティとアイテムの違いも難しいか。

その辺りの詳しいところをこれから説明していきましょう。

3.3. プロパティについてくわしく

ここから説明が足りなかった部分を詳しく見ていきます。まずはプロパティから。

```
<!-- プロパティ(変数)の定義 -->
<PropertyGroup>
  <Pandoc>pandoc</Pandoc>
  <OutputFormat>docx</OutputFormat>
</PropertyGroup>
```

プロパティは軽く説明した通り変数です。事前に定義しておいた文字列を後で参照することができます。何度も使うけどたまに書き換える必要がある値とかをまとめておくと便利です。

また、コマンドラインから `msbuild /p:OutputFormat=html5` といった形 (省略無しだと `msbuild /property:OutputFormat=html5`) でオプションを渡すとプロパティを上書きできます。ユーザーが指定するオプションを作りたい時もプロパティを作りましょう。

プロパティは全て文字列になります。スペースやカンマやセミコロンで区切って複数の文字列を書いてみるといったことも出来ますが、結局のところただつながった文字列として参照されます。

同じプロパティを複数定義すると後から出てきた値で上書きになります。

```
<PropertyGroup>
  <OutputFormat>docx</OutputFormat>
  <OutputFormat>html5</OutputFormat>
</PropertyGroup>
```

この場合は `OutputFormat` の値は `html5` になります。こんな直後に同じプロパティを書くことはないと思いますが、あとで紹介する他のプロジェクトファイル読み込んだり、条件によって値を上書きしたい場合などに使えます。

プロパティの定義の中でプロパティの参照も可能です。単純に上書きでなく、前の値に追加したいような場合は自分自身を参照しましょう。

```
<PropertyGroup>
  <OutputFormat>docx</OutputFormat>
  <OutputFormat>$(OutputFormat);html5</OutputFormat>
</PropertyGroup>
```

これで `OutputFormat` プロパティを参照すると `docx;html5` という値が取得できます。もちろん自分自身だけでなく他のプロパティも参照可能です。

気をつけたいところは、プロパティの値は定義した時点で値が決定されます。プロパティの定義内でプロパティ参照を使っていると、定義時点の値が展開されます。プロパティの定義はファイルの上から行われるので、定義より下で参照しているプロパティが書き換えられても影響はありません。また、定義より下で定義されているプロパティを参照すると空になってしまいます (コマンドラインなどで外で定義されていない限り)。プロパティの中で他のプロパティを参照する時には定義場所には気をつけてください。

`PropertyGroup` 要素の位置は例だと `Project` 要素内の一番最初に書いてありますが、べつに最初の必要はなく `Project` 要素の直下であればどこに何回出てきてもかまいません。何回でも書けますので、沢山プロパティがある場合は適当に `PropertyGroup` 要素で分けてグループ化してお

くと便利でしょう。あと PropertyGroup は Project 要素の中だけでなく Target 要素の中とかにも入れられます。Target 要素の中に入れた場合はちょびっとだけ動作が違うのですが、それについては後で紹介します。

3.4. アイテムについてくわしく

次はアイテムの詳細について説明しましょう。アイテムはプロパティよりちょっと複雑で、そして重要です。

```
<!-- アイテム(入力ファイル)の定義 -->
<ItemGroup>
  <Inputs Include="hoge.md" />
  <Inputs Include="fuga.md" />
</ItemGroup>
```

アイテムはプロパティと違ってファイル一覧を指定するともの、としましたが、もう少しだけ正確に書くと、アイテムは配列です。実はプロパティはスカラー (ただの文字列) である一方で、アイテムはリスト (文字列の配列) です。いや本当に正確に書くとアイテムはただの文字列の配列ではないんですが、当面は文字列の配列と思ってもらって大丈夫です。

文字列の配列なので一応ファイル名以外もなんでも入るんですが、だいたいファイル名かせいぜいディレクトリ名を入れるくらいが普通の使い方ですし、それ以外の使い方をしようとすると余計に難しいことをしようとしている可能性があるのであまりおすすめはできません。ただ、指定するファイル名は (基本的には文字列なので) 存在するファイル名でなくてもかまわないことは覚えておきましょう。出力ファイル名とかを指定する場合には必然的に存在しないファイルを指定することになりますしね。

ただの文字列と文字列の配列との違いについて分かりやすい点としては、プロパティは後から書いた方で上書きになりますが、アイテムの場合は追加されるだけという点が挙げられます。例では Inputs アイテムを 2 回書いていますが、プロパティだったら上書きされて hoge.md になってしまうところ、アイテムでは hoge.md と fuga.md の両方が入った配列として取得できます。

また、Include 属性にはファイル名を指定していますが、ここには一つのファイル名だけでなく、複数のファイルを一度に指定することができます。

複数のファイルを指定するには方法がいくつかありますが、一つはセミコロンで区切って複数書く方法です。

```
<ItemGroup>
  <Inputs Include="hoge.md; fuga.md" />
</ItemGroup>
```

これで前の例と同じ意味になります。セミコロン前後のスペースはあってもなくてもかまいません。複数のターゲット指定方法について書いた時もセミコロンで区切っていましたが、MSBuild では複数の何かを一度に指定する時はセミコロンで区切ります。

もう一つの方法として Include にはワイルドカードの指定もできます。次のようになっていてもかまいません。

```
<ItemGroup>
  <Inputs Include="*.md" />
</ItemGroup>
```

この場合はこのディレクトリ (.proj ファイルがあるディレクトリ) にある、拡張子が .md のファイルを全て列挙して Inputs アイテムとして参照できるようにします。もちろんワイルドカードを使った場合でも、複数の定義で追加できますし、一度に複数のワイルドカードをセミコロンで区切って書いても大丈夫です。

```
<ItemGroup>
  <Inputs Include="*.md;*.txt" />
</ItemGroup>
```

これですと、拡張子が .md のファイルに加えて .txt のファイルも Inputs アイテムとして参照できます。

ワイルドカードに含めてほしくないファイルがある場合は Exclude 属性を使うと除外できます。

```
<ItemGroup>
  <Inputs Include="*.md;*.txt" Exclude="readme.md;readme.txt" />
</ItemGroup>
```

この例では、ディレクトリ内にある .md と .txt ファイルのうち、readme.md と readme.txt だけ除外したリストを Inputs アイテムに追加しています。

Exclude 属性は今 Include 属性で列挙してリストに追加しようとしているものから除外するだけですので、前に列挙しといたやつから取り除くのはできません。

```
<ItemGroup>
  <Inputs Include="*.md;*.txt" />
  <Inputs Exclude="readme.md;readme.txt" />
</ItemGroup>
```

これは無理です。二番目の方で Include 属性が無いよというエラーが出るでしょう。

あまり無いとは思いますが、あとから取り除きたい場合には Exclude でなく Remove 属性を使ってください。

```
<ItemGroup>
  <Inputs Include="*.md;*.txt" />
  <Inputs Remove="readme.md;readme.txt" />
</ItemGroup>
```

こちらだと思った通りの動きをしてくれます。Remove 属性にもワイルドカードを指定したり、さらに Exclude 属性を付ける (取り除くリストから除外する、つまり取り除かない！) こともできますが、ややこしくなるのでほどほどに。

アイテムのファイル名列挙にはプロパティ参照や、他のアイテム参照も使えます。

```
<PropertyGroup>
  <InputFormat>md</InputFormat>
</ItemGroup>
<ItemGroup>
  <Inputs Include="*.$(InputFormat)" />
</ItemGroup>
```

$(プロパティ名) でプロパティの値を参照できるので、この例では *.md というワイルドカードに適合するファイルが列挙されます。

```
<ItemGroup>
  <InputsMarkdown Include="*.md" />
  <InputsText Include="*.txt" />
  <Inputs Include="@(InputsMarkdown);@(InputsText)" />
</ItemGroup>
```

@(アイテム名) で他のアイテムの値を参照できます。アイテムの値はワイルドカードでなく、ファイルを列挙した後のファイル名の配列として参照されます。単純に文字列に変換される時にはセミコロン区切りでファイル名を並べたものになります。この例だと @(InputsMarkdown) は hoge.md と fuga.md などに、@(InputsText) は foo.txt と bar.txt と baz.txt といった形に展開され、最終的に Inputs アイテムは Include 属性に hoge.md;fuga.md;foo.txt;bar.txt;baz.txt を指定したのと同じ形になります。

アイテムの参照はこの単純な参照以外にもいろんな機能があるのですが、かなり複雑なのであとで紹介しましょう。

ところで ItemGroup 要素を書く位置ですが、PropertyGroup 要素と同様に Project 要素の直下であればどこに何回出てきてもかまいません。沢山アイテムがある場合は適当に ItemGroup 要素を分けておくと良いでしょう。ただ、アイテムの定義内で他のアイテムやプロパティを参照している場合には、そのアイテムやプロパティの定義の方が上にないと空になってしまいますので、位置に気をつけてください。

ItemGroup を Target 要素の中に入れることもできます。Target 要素の中でアイテムの定義を書くことがワイルドカードと組み合わせて重要なテクニックになるのですが、その点について次で説明しちゃいましょう。

3.4.1. アイテムのワイルドカードについて

アイテムのワイルドカードは * だけさらっと使ってましたが、他にもいくらか使えるパターンがあるので紹介しておきます。

パターン	説明
*	0 個以上のパス区切り以外の文字とマッチします。

パターン	説明
?	1 個のパス区切り以外の文字とマッチします。
**	0 個以上のサブディレクトリとマッチします。

表 2. アイテムで使えるワイルドカード

パス区切り文字とさらっと書いてますが、MSBuild でパス区切りに使える文字は / と \ です。どちらも同じに扱ってくれるので混ぜて書いても大丈夫です。

パターンはどれも難しいものではないのですが、** がちょっと分かりづらいかもしれないので解説しておきます。

** はサブディレクトリを再帰的に辿って検索してくれるものです。

```
a.txt
foo/
 |- bar/
     |- b.txt
     |- baz/
         |- c.txt
```

といったサブディレクトリ構造があった場合に、**/*.txt で検索すると a.txt;foo/bar/b.txt;foo/bar/baz/c.txt が列挙できます。便利ですね。0 個以上のサブディレクトリへのマッチなのでa.txtにもマッチしています。a.txtにはマッチさせたくない場合はfoo/**/*.txtといった形で書きましょう。

ただ、** はディレクトリにしかマッチしないので、**.txt とか書くと結果として **.txt そのものになります。ちょっと不思議な動作してますね。まあ、横着せずに普通に **/*.txt と書けば良いでしょう。

ワイルドカードの展開タイミングについても説明しておきましょう。ワイルドカードの展開タイミング、つまりファイルが検索されるのはアイテムの定義時点です。Project 要素の直下の ItemGroup 要素では全てのターゲットの実行前 (MSBuild 起動時点) に列挙されるため、なんらかのタスクの実行時に作られるファイルを列挙しようとしても上手くいきません。

たとえば……

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="12.0" DefaultTargets="Build">
  <ItemGroup>
    <Inputs Include="*.md" />      <!-- 起動時に*.mdファイルを列挙 -->
    <Outputs Include="*.docx" />  <!-- 起動時に*.docxファイルを列挙 -->
  </ItemGroup>
  <Target Name="Build">
    <!-- 起動時に列挙した*.mdファイルを表示 -->
    <Message Text="入力ファイル: @(Inputs)" />

    <!-- Buildタスク実行時に*.docxファイルを作る -->
    <Exec Command="pandoc -t docx -o %(Inputs.Filename).docx @(Inputs)" />

    <!-- 起動時に列挙した*.docxファイルを表示……？ -->
    <Message Text="できたファイル: @(Outputs)" />
  </Target>
</Project>
```

こんな感じで出力ファイルを表示したいとしましょう。Outputs アイテムは起動時に *.docx を検索しています。メッセージ表示はコマンド実行後に行っているので、できたファイル： hoge.docx; fuga.docx と表示して欲しいのですが、検索は MSBuild 起動時に行われてしまっているので、docx ファイルは一つも見つけられません。

言われれば当たりまえじゃんと分かるんですが、これ実は 2 回目動かすとできたファイル：hoge.docx; fuga.docx が表示されてしまいます。というのも、1 回目の実行でファイルが出来ているので 2 回目には MSBuild 起動時に docx ファイルが存在してしまっているのですね。で、動いてるわーと出来たわーと思いつつ Clean したあとに動かすとなんか上手く動かなくて首を捻ることになります。なんとなく上手くいってるように見えても、展開されるタイミングを間違えると上手く動かないことがあるので気をつけましょう。

上のように出力ファイルを列挙したい場合には、Target 要素の中に ItemGroup 要素を入れてアイテムを定義してしまいます。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="12.0" DefaultTargets="Build">
  <ItemGroup>
    <Inputs Include="*.md" />
  </ItemGroup>
  <Target Name="Build">
    <Message Text="入力ファイル: @(Inputs)" />
    <Exec Command="pandoc -t docx -o %(Inputs.Filename).docx @(Inputs)" />
    <ItemGroup>
      <Outputs Include="*.docx" />
    </ItemGroup>
    <Message Text="できたファイル: @(Outputs)" />
  </Target>
</Project>
```

Target 要素の中は上から順番に実行されるので、これで見ての通りのタイミングでファイルの検索をしてくれます。

3.5. ターゲットについてくわしく

ターゲットについてももう少し詳しく見ていきます。

```
<!-- ターゲットの定義 -->
<Target Name="Build">
  <Message Text="Converting @(Inputs) to $(OutputFormat)" />
  <Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Inputs.
Filename).$(OutputFormat) @(Inputs)" />
</Target>
```

Name 属性に名前を書いて、中身にはタスクを実行したい順番で上から並べるだけでしたね。

中に書ける処理としては、タスク以外にアイテムやプロパティの定義もできます。

```
<Target Name="Build">
  <PropertyGroup>
    <OutputFormat>docx</OutputFormat>
  </ItemGroup>
  <Message Text="Converting @(Inputs) to $(OutputFormat)" />
  <Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Inputs.
Filename).$(OutputFormat) @(Inputs)" />
  <ItemGroup>
    <Outputs Include="*.docx" />
  </ItemGroup>
  <Message Text="Converted: @(Outputs)" />
</Target>
```

こんなこともできます。ターゲット内に書いたプロパティやアイテムの定義もタスクといっしょに上から順番に実行されます。

アイテムのワイルドカードの節で説明もしましたが、アイテムのファイル列挙は定義タイミングで行われるので、Target の中で出力ファイルを生成するタスク実行後にアイテムの定義を行うことで、出力されたファイルを列挙することができます。

この例だと OutputFormat プロパティをターゲットの中で定義する意味はありませんが、プロパティの値もやはり定義タイミングで行われます。タスクによっては出力をプロパティやアイテムに受け取れるものがあります。出力された値をさらに加工したい場合などに使えるでしょう。

3.5.1. タスクが失敗したらどうなるの？

ターゲットの実行途中になんらかのエラーで失敗した時って、どうなってしまうんでしょう。失敗したところで終了する？なんとなく先まで進んじゃう？

答えは、失敗したところで終了する、です。

何かタスクが失敗したら、それ以降のタスクは実行されずに終了します。依存しているターゲット内で失敗したら、呼び出し元のターゲットも失敗でそのまま終了します。

ただし、失敗してもかまわないタスクがあった場合には、失敗しても続けることを指定できます。

```
<Exec Command="hoge.exe" ContinueOnError="true" />
```

このように、タスクの ContinueOnError パラメータで true を指定すると、タスク失敗時にも警告表示をするだけで続けて実行することができます。false を指定すると失敗時にはエラー終了になりますが、既定の動作なのでわざわざ false を指定することはないと思います。

この ContinueOnError パラメータは、どのタスクでも共通で使えるパラメータなので、上のような Exec タスク以外でも指定することができます。

3.5.2. 更新されたファイルだけビルドしたい - ターゲットの入出力

今までの例を実際に何度か実行してみた人が居れば、MSBuild を実行する度に pandoc コマンドが実行されて docx ファイルが作り直されてたのに気付いたかもしれません。実行までやってなかったり、動かしても気付かなかったかもしれませんが、まあそれはそれでいいです。実は実行の度に作り直されてたんだよ！！

例みたいな Markdown から docx に変換するといったちょっとした処理だと、余程長いテキストでもない限りすぐ終わるからいいんですけど、PDF を書き出すだとか、C++ プログラムのコンパイルだとかだとそれなりに時間がかかります。ターゲットは特に何も指定しない限り毎度実行されてしまいますが、時間がかかる処理は変更があった時だけビルドしなおしして欲しいですよね。

MSBuild ではターゲットに入力・出力ファイルを指定すると、それらの更新日時を比較して、入力の方が新しい (もしくは出力側が存在しない) 場合だけターゲットを実行してくれます。

```
<ItemGroup>
  <Markdown Include="hoge.md;fuga.md" />
  <Docx Include="hoge.docx;fuga.docx" />
</ItemGroup>
<Target Name="Build" Inputs="@(Markdown)" Outputs="@(Docx)">
  <Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Markdown.
Filename).$(OutputFormat) @(Markdown)" />
</Target>
```

Target 要素の Inputs 属性に入力ファイルのリスト、Outputs 属性に出力ファイルのリストを指定すると、実行前にそれらの更新日時を比較して、出力の方が新しければターゲットの実行を省略します。

```
> msbuild
...中略...
Build:
すべての出力ファイルが入力ファイルに対して最新なので、ターゲット "Build" を省略します。
プロジェクト "target.proj" (Build ターゲット) のビルドが 完了しました。
...後略...
```

こんな感じで省略されました。出力側のファイルが (一つでも) 無い、または入力より古い場合はターゲットのコマンドが実行されます。

もし、入力ファイルのうち少しだけが変更されてた時にはどうなるでしょう。普通に考えれば全部ビルドしなおしちゃうんだろうなと思いたくなりますが、なんと一部だけビルドしなおしてくれます。

```
> msbuild /t:Build
...中略...
Build:
いくつかの出力ファイルが、それらの入力ファイルに対して古い形式であるため、ターゲット
"Build" を部分的にビルドしています 。
  pandoc -t docx -o hoge.docx hoge.md
プロジェクト "target.proj" (Build ターゲット) のビルドが 完了しました。
...後略...
```

タスクのパラメータで %(アイテム名 . なんとか) の形式でアイテムを参照した場合には、アイテム内の項目 (ファイル) それぞれに対してタスクが実行されますが、更新されてないファイルに関しては、その項目だけスキップされるようになっています。ただし、どの入力ファイルがどの出力ファイルに対応してるかまでは判別してないので、出力ファイルのいずれかより新しい入力ファイルは全て省略されずにビルドされます。

この省略機能は便利ではありますが、Outputs の指定が難しいのが難点です。Outputs に指定する出力ファイルは当然実行前に列挙できてないとファイルがあるかどうか判別できませんし、空にもできません (空の場合は常に省略されてしまいます)。つまり出力側に指定するファイルはワイルドカードで列挙できないのです。これはちょっと困りましたね。

そこでアイテムを一部変更しつつ参照するというテクニックが使えます。

```
<ItemGroup>
  <Markdown Include="hoge.md;fuga.md" />
  <Docx Include="@(Markdown->'%(Filename).docx')" />
</ItemGroup>
<Target Name="Build" Inputs="@(Markdown)" Outputs="@(Docx)">
  <Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Markdown.
Filename).$(OutputFormat) @(Markdown)" />
</Target>
```

Docx アイテムの Include 属性をこんな感じで指定して出力側のアイテムを作ってやります。@(Markdown->'%(Filename).docx') の意味がわかりませんね。詳しくは後で説明しますが、@(Hoge->'Fuga') という形で Hoge アイテムの中身をそれぞれ Fuga で置き換えたものにする、という意味になります。さらに %(Filename) はファイル名のうち拡張子を含まない部分を参照します。

これを組み合わせると、@(Markdown->'%(Filename).docx') は、Markdown アイテムのそれぞれの拡張子を含まないファイル名に .docx をくっつけた物、となり、Docx アイテムには hoge.docx;fuga.docx が入ることになります。いやー難しいですねー。

この辺の難しい変換については、あとでタスクの解説とアイテムの参照のところで詳しく説明します。

3.5.3. 先に他のターゲットを実行したい - ターゲットの依存関係

ターゲットには依存関係をつけることができます。このターゲットを実行するには先にあっちのターゲットを実行しとかないといけないといった設定です。

依存関係は Target 要素の DependsOnTargets 属性で指定します。

```
<Target Name="Build">
  ～
</Target>
<Target Name="Clean">
  ～
</Target>
<Target Name="Rebuild" DependsOnTargets="Clean;Build">
  <Message Text="Rebuild completed"/>
</Target>
```

ちょっと雑な例ですがこんな感じです。Rebuild ターゲットを実行しようとすると、先に Clean と Build を実行しないといけないので実行します。ここで Rebuild はそれ以上何もすることないので空でも問題ないんですが、寂しいのでとりあえずメッセージでも表示するだけしています。普通は空で問題ないです。

DependsOnTargets にはセミコロンで区切って複数のターゲットを指定できますが、複数指定した場合は左に指定したものから順番に実行されます。

ターゲットの依存関係は深く連鎖することも当然可能です。依存先が他のターゲットに依存していたら、さらに依存先が先に実行されます。辿っていった結果ループしてしまった場合はもちろんエラーになります。辿っていった結果、一度の MSBuild で 2 回以上同じターゲットが依存先に出てくる場合が出てくるかもしれませんが、その場合 2 回目以降は省略されるため、同じターゲットは 1 度しか実行されません。

3.6. タスクについてくわしく

ターゲットの中ではタスクは使うだけでした。

```
<Message Text="Converting @(Inputs) to $(OutputFormat)" />
<Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Inputs.
Filename).$(OutputFormat) @(Inputs)" />
```

じゃあタスクの定義はどこで行なわれるかというと、別なアセンブリ (DLL) などで定義したやつを読み込んで使います。読み込む方法はまた後で紹介するのですが、ここで使った Message や Exec のように別に読み込まなくても標準で使えるタスクが十分揃ってますので、まずはそれらについて紹介していきましょう。

タスクは標準のだけでもけっこう沢山ありますが、よく使うのだけ紹介していきます。物によってはパラメーターも沢山ありますが、誌面の都合上全部は紹介しません。

たまに出力パラメータというのもありますが、あとで紹介するので気にしなくて大丈夫です。

まずどのタスクでも使える共通パラメータを紹介しておきます。

名前	説明	必須
Condition	実行されるかどうかの条件式を指定します。 未指定の場合は常に実行されます。	
ContinueOnError	失敗した時に実行を続けるかどうかを指定します。 true だと失敗時も警告を出すだけで続けます。 false では失敗時にそれ以降の実行をせずに終了します。 省略すると false になります。	

表 3. タスクの共通パラメータ

Condition パラメータを使うと条件が成立した時のみ実行されるタスクというのを作ることができます。条件式などについては複雑になるのでまた後での解説をします。

ContinueOnError パラメータはタスクが失敗したらどうなるの？のところでも説明しましたが、ここに true を指定すると、失敗してもそのまま実行を続けるようにできます。

これらの共通パラメータはどのタスクにでも指定することができます。

3.6.1. Message タスク

メッセージを表示するだけのタスクです。現実的に良く使うかというと微妙ですが、printf デバッグ的に使えるという意味では良く使うでしょう。

```
<Message Text="Converting @(Inputs) to $(OutputFormat)" Importance="normal" />
```

パラメーターは Text と Importance があります。

指定できるパラメータは以下の通りです。

名前	説明	必須
Text	表示するテキストです。好きな文字列を指定できます。	✓
Importance	どのログレベルまで出すかを指定します。 high・normal・low のいずれかを指定してください。 省略すると normal になります。	

表 4. Message タスクのパラメータ

MSBuild 実行時にあまり余計なログを出したくない場合は msbuild /verbosity:minimal といった形で出すメッセージの量をコントロールできますが、Importance はどのレベルで出すかを指定できます。

Importance / verbosity	quiet	minimal	normal	detailed	diagnostic
high		✓	✓	✓	✓
normal			✓	✓	✓
low				✓	✓

表 5. Importance パラメータと表示される verbosity レベル

まあ特に指定する必要はないと思います。ただ、dotnet msbuild コマンドの場合は、標準で verbosity が minimal になってるので省略したり normal 以下を指定すると普通には出てきません。出したい時には dotnet msbuild /v:n オプションで verbosity を normal まで出るようにしましょう。

3.6.2. Warning / Error タスク

警告やエラーを表示するタスクです。

Message タスクと似たような物ですが、Warning は warning の文字列と共に目立って表示されるので分かりやすいです。また、verbosity に quiet を指定した時も表示されます。

Error は error の文字列と共にやはり目立って表示され、実行時点でエラー終了し、これ以降のタスクは実行されません。

```
<Warning Text="何か警告" />
<Error Text="何かエラー" /> <!-- これ以降は実行されない -->
```

指定できるパラメータは以下の通りです。

名前	説明	必須
Text	表示するテキストです。 省略するとメッセージが指定されていない旨が表示されます。	
File	警告やエラーがあった場所として表示されるものです。 ただのテキストなので、ファイル名だけでなく行番号なども含められます。 省略するとプロジェクトファイル名と 行番号が表示されます。	

表 6. Warning/Error タスクのパラメータ

Warning や Error タスクですが、普通に使うと当然ながら常に警告やエラーが出てしまいます。廃止予定のターゲットに仕込むとかなら常に出てもいいのですが、多くの場合はあとで説明する条件実行と組み合わせて、何か警告やエラーがあるようなことがあった場合に出力することになるでしょう。

3.6.3. Touch タスク

ファイルの更新日時を設定します。また、ファイルが無ければ空のファイルを作ることもできます。

```
<Touch Files="@(なんかファイル一覧)" AlwaysCreate="true"/>
```

指定できるパラメータは以下の通りです。

名前	説明	必須
Files	日時を更新する対象のファイルのリストです。 複数指定できます。	✓
ForceTouch	読み取り専用でも強制的に変更するかどうかを true か false で指定します。省略すると false です。	
AlwaysCreate	対象のファイルが無い場合に作成するかどうかを true か false で指定します。省略すると false です。	
Time	設定する日時を文字列で指定します。 省略すると現在時刻が設定されます。	
TouchedFiles	出力パラメータです。 正常に日付更新されたアイテムが入ります。	

表 7. Touch タスクのパラメータ

AlwaysCreate を true にすると、ファイルが存在しない場合に空のファイルを作ってくれますが、パスが存在しない場合にはエラーになるのでディレクトリは先に作っておく必要があります。

3.6.4. Copy タスク

ファイルのコピーをします。

```
<Copy SourceFiles="@(Files)" DestinationFolder="foo/bar"/>
```

指定できるパラメータは以下の通りです。

名前	説明	必須
SourceFiles	コピー元のファイルのリストを指定します。 複数指定できます。	✓
DestinationFiles	コピー先のファイル名のリストを指定します。 複数指定できますが、SourceFiles に指定したのと同じ数を指定する 必要があります。	※
DestinationFolder	コピー先のフォルダ (ディレクトリ) です。 一つだけ指定できます。 指定したディレクトリが存在しなければ 作ります。	※
OverwriteReadOnlyFiles	読み取り専用でも強制的に上書きするか どうかを true か false で指定します。 省略すると false です。	
SkipUnchangedFiles	変更が無い (サイズと日時が同じ) ファイルは コピーしないかどうかを true か false で指定します。 省略すると false です。	
CopiedFiles	出力パラメータです。 実際にコピーされたアイテムが入ります。	

表 8. Copy タスクのパラメータ

※ : DestinationFiles と DestinationFolder はどちらか一方だけ必須です。

ファイル名を変更しつつコピーする場合は DestinationFiles パラメータが使えますが、普通にコピーする場合は DestinationFolder を使います。両方は指定できません。

コピー先のディレクトリが無い場合は勝手に作ってくれます。便利ですね。

コピーができるのはファイルだけなので、ディレクトリを指定するとエラーになります。ただ、ワイルドカードを駆使すれば同じことはできるでしょう (空のディレクトリ以外は)。

3.6.5. Move タスク

ファイルの移動をします。

```
<Move SourceFiles="@(Files)" DestinationFolder="foo/bar"/>
```

指定できるパラメータは以下の通りです。

名前	説明	必須
SourceFiles	移動元のファイルのリストを指定します。 複数指定できます。	✓
DestinationFiles	移動先のファイル名のリストを指定します。 複数指定できますが、SourceFiles に指定したのと同じ数を指定する 必要があります。	※

名前	説明	必須
DestinationFolder	移動先のフォルダ (ディレクトリ) です。 一つだけ指定できます。 指定したディレクトリが存在しなければ 作ります。	※
OverwriteReadOnlyFiles	読み取り専用でも強制的に上書きするかを true か false で指定します。 省略すると false です。	
MovedFiles	出力パラメータです。 実際に移動されたアイテムが入ります。	

表 9. Move タスクのパラメータ

ファイル名を変更しつつ移動する場合は DestinationFiles パラメータが使えますが、普通に移動する場合は DestinationFolder を使います。両方は指定できません。

移動先のディレクトリが無い場合も勝手に作ってくれます。

移動ができるのはファイルだけなので、ディレクトリを指定するとエラーになります。ディレクトリを移動させたい場合は、中のファイルをワイルドカードを駆使して移動させつつ元のディレクトリを RemoveDir タスクで消す、ということをやればできますが、普通に外部コマンドを呼び出した方が早いかもしれません。

3.6.6. Delete タスク

ファイルを削除します。

```
<Delete Files="@(IranaiFiles)"/>
```

指定できるパラメータは以下の通りです。

名前	説明	必須
Files	削除するファイルのリストを指定します。 複数指定できます。	✓
TreatErrorsAsWarnings	エラー発生時も警告扱いにして進めるかを true か false で指定します。 省略すると false です。	
DeletedFiles	出力パラメータです。 実際に削除されたアイテムが入ります。	

表 10. Delete タスクのパラメータ

削除しようとしたファイルが既に存在しない場合は何も起きずに正常終了します。書き込み禁止などで削除できなかった場合のみエラーになります。

削除できるのはファイルだけです。ディレクトリを削除したい時には RemoveDir タスクを使います。

3.6.7. MakeDir タスク

ディレクトリを作成します。

```
<MakeDir Directories="@(Dirs)"/>
```

指定できるパラメータは以下の通りです。

名前	説明	必須
Directories	作成するディレクトリのリストを指定します。 複数指定できます。	✓
DirectoriesCreated	出力パラメータです。 実際に作成されたディレクトリが入ります。	

表 11. MoveDir タスクのパラメータ

作ろうとしたディレクトリの途中のディレクトリも無い場合、途中のディレクトリも合わせて一気に作ってくれます。また、作ろうとしたディレクトリが既にある場合は、何も言わずに成功します。

3.6.8. RemoveDir タスク

ディレクトリとその中身を削除します。

```
<RemoveDir Directories="@(Dirs)"/>
```

指定できるパラメータは以下の通りです。

名前	説明	必須
Directories	削除するディレクトリのリストを指定します。 複数指定できます。	✓
RemovedDirectories	出力パラメータです。 実際に削除されたディレクトリが入ります。	

表 12. RemoveDir タスクのパラメータ

削除しようとしたディレクトリが空でなくとも、中身も合わせて全部消します。既に無いディレクトリを指定した場合は、何も言わずに成功します。

3.6.9. Exec タスク

外部コマンドを実行します。手でプロジェクトファイルを書く場合には良く使うでしょう。

```
<Exec Command="$(Pandoc) -t $(OutputFormat) -o %(Inputs.
Filename).$(OutputFormat) @(Inputs)" />
```

指定できるパラメータは以下の通りです。

名前	説明	必須
Command	実行するコマンドを文字列で指定します。 コマンドはシェル経由で実行されます。	✓
WorkingDirectory	コマンド実行時のカレントディレクトリを 指定します。 省略するとプロジェクトファイルがある場所に なります。	
EnvironmentVariables	コマンド実行時に追加・上書きする環境変数を 名前と値を = でつないだ文字列で指定します。 複数指定することも可能です。	
Timeout	コマンドを強制するまでの時間をミリ秒単位で 指定します。省略した場合はタイムアウトしません。	

名前	説明	必須
IgnoreExitCode	終了コードを無視するかどうかを true か false で指定します。無視しない場合は終了コードが 0 でない時に エラーとします。省略した場合は false(無視しない) になります。	
ExitCode	出力パラメータです。 コマンドの終了コードが入ります。	
ConsoleToMSBuild	コマンドの標準出力や標準エラー出力のキャプチャ 有無を true か false で指定します。 キャプチャした文字列は ConsoleOutput に入ります。 MSBuild 15 以降でないと使えません。	
ConsoleOutput	出力パラメータです。 コマンドで出力された標準出力と標準エラー出力が 入ります。 ConsoleToMsBuild が true の時だけ入ってきます。 MSBuild 15 以降でないと使えません。	

表 13. Exec タスクのパラメータ

実行するコマンドはシェル経由 (Windows だと cmd.exe) で起動されるので、環境変数やリダイレクトも使えますし、シェル内蔵のコマンドも使えます。ただし、シェルはプラットフォームによって変わるので、凝ったことをする場合は気をつけてください。

EnvironmentVariables を指定するとコマンド実行時の環境変数を指定できます。MSBuild 実行時の環境変数はそのまま引き継がれますが、それに対してさらに追加や上書きを指定します。複数指定する場合はセミコロンで区切ってください。hoge=foo;fuga=bar のように指定します。

IgnoreExitCode を指定しない場合 (もしくは false を指定した場合) は、コマンドの終了コードが 0 でない時にエラー扱いとし、そこで失敗します。コマンドが失敗しても問題ないので続けて欲しい場合は IgnoreExitCode を true にしましょう。

MSBuild 15 以降 (Visual Studio 2017 や .NET Core SDK の MSBuild) では、実行したコマンドの標準出力や標準エラーを受け取ることができます。出力パラメータから取得する必要があるのですが、出力パラメータに関してはだいぶ後で解説するのでそちらを参照してください。

リダイレクトでファイルに書き出すのは MSBuild のバージョン関係なくできますので、標準出力やエラー出力が欲しい場合にはリダイレクトを駆使するのも良いでしょう。

3.6.10. MSBuild タスク

MSBuild を呼び出して、他のプロジェクトやターゲットを実行します。

```
<MSBuild Projects="@(BuildProjects)" Targets="Build"
Properties="Condfiguration=Release;Platform=Win32" />
```

指定できるパラメータは以下の通りです。

名前	説明	必須
Projects	ビルドするプロジェクトファイルを指定します。 複数指定できます。	✓

名前	説明	必須
Targets	実行するターゲットのリストを指定します。 指定したプロジェクト全てが 指定したターゲットを持っている必要があります。 省略するとデフォルトターゲットを実行します。	
Properties	プロジェクトに渡すプロパティのリストを指定します。 省略した場合は特に何も渡しません。	
StopOnFirstFailure	true を指定すると プロジェクトのビルドに一つでも失敗した時に、それ以降のプロジェクトをビルドせずに失敗とします。 false を指定するか省略すると、一つ失敗しても 他の全プロジェクトをビルドします。 BuildInParallel とは同時に使えません。	
BuildInParallel	true を指定すると、各プロジェクトを並列ビルドします。false を指定するか省略すると、一つずつ順番にビルドします。StopOnFirstFailure とは同時に使えません。	
TargetOutputs	出力パラメータです。 実行したターゲットの Returns で指定された出力が 入ります。	

表 14. MSBuild タスクのパラメータ

なにげに良く使うタスクです。

というのも Visual Studio のプロジェクトは MSBuild のプロジェクトですので、Visual Studio で作ったプロジェクトをバッチビルドしたーい、と思ったら MSBuild タスクを呼び出すと、各プロジェクトをビルドするのが簡単にできます。コマンドラインから /p:hoge=fuga オプションで指定するように Properties パラメータでプロパティの上書き設定もできるので、いろんな設定でビルドしたりも簡単です。

Visual Studio のプロジェクトで指定できるプロパティは後で紹介しますが、良く使う 2 つをここに挙げておきます。

- Configuration
  Debug や Release などビルド設定を指定します。
- Platform
  Win32・Any CPU・x86・x64 などターゲットとするプラットフォームを指定します。
  複数指定する場合は Configuration=Release;Platform=x64 のようにセミコロンで区切って指定してください。

ちなみに、このタスクを実行する時点で定義されているプロパティやアイテムは、呼び出し先に引き継がれたりはしません。引き継ぎたいプロパティは Properties パラメータで明示的に指定しましょう。ただ、起動時に指定されたコマンドラインのパラメータだけは引き継がれるようで、そこで /p などがあればそれはそのまま渡されます。ちょっと難しいですね。

自分自身のファイルを実行する場合には、Projects パラメータに $(MSBuildProjectFullPath) を指定します。これは後で説明する既定のプロパティですが、起動されたプロジェクトファイル自身が入っているものです。

MSBuild のプロジェクトファイルを手で書く目的に、バッチビルドはよくあるものだと思いますので `MSBuild` タスクは覚えておくと良いでしょう。他にもあとで説明するターゲットからの出力と組み合わせるとさらに強力なことができたりと、いろいろと使いでがあるタスクです。

4. 本格的に使いたい - もっと詳しい機能について

ここまでで小さいプロジェクトファイルについて説明しました。

これだけでも最低限使えそうなところまでは行ったのですが、まだちゃんと解説されてないところがあって、たぶん自分で何か書き始めると詰まるんじゃないでしょうか。ここから本格的に複雑な機能の説明をしていきましょう！

4.1. ファイル名をいろんな形で渡したい - アイテムの参照

タスクを使う時にパラメータにアイテムを渡すことが多いと思います。

アイテムは @(アイテム名) といった形で参照すると、セミコロン区切りの文字列になります。たとえば hoge.md;foo.md とかですね。Copy タスクとかならその形で受け取ってくれるので全く問題ないのですが、Exec タスクで外部コマンドに渡す場合はだいたい困ります。

```
<Exec Command="pandoc -t docx -o hon.docx @(Inputs)" />
```

pandoc コマンドで複数の Markdown ファイルをくっつけつつ docx に変換したい、といった場合ですが、これを実行しようとすると

```
pandoc -t docx -o hon.docx hoge.md;fuga.md
```

に展開されますが、pandoc は hoge.md;fuga.md とかいうファイルを開こうとしてエラーになってしまいます。ここはスペース区切りにしてほしいですね。

区切り文字の変更は簡単で、@(アイテム名 ,' 区切り文字 ') という形で参照するとセミコロン以外の区切り文字が使えます。スペースで区切りたい場合には次のようにすれば良いでしょう。

```
<Exec Command="pandoc -t docx -o hon.docx @(Inputs,' ')" />
```

また、以前にも説明した通り、各アイテムの名前を変換したリストを参照することができます。

@(アイテム名 ->' 文字列 ') という形で参照すると、各アイテムをそれぞれ文字列で置き換えたリストの参照になります。全てのアイテムを固定の文字列で置き換えても意味ないので、それぞれのアイテム名やアイテムの一部を参照するのに %(メタデータ) というのが使えます。メタデータというのはアイテム毎に定義できるおまけデータのことです。自分で定義することもできますが、標準でファイル名の操作に便利なメタデータが使えるのでそれを利用しましょう。

メタデータ名	説明	値の例
Identity	ファイルの相対パス (Include 属性で指定した値) を参照します。 ワイルドカードは展開した後の値になります。	bar\hoge.txt
RelativeDir	ファイルの相対パスのうち ディレクトリ部分を参照します。最後のパス区切りも含みます。	bar\
RecursiveDir	ワイルドカードの ** にマッチした部分だけを 参照します。	foo\bar\
Filename	ファイル名のうち拡張子を含まない部分を 参照します。ディレクトリ名も含みません。	hoge
Extension	拡張子だけを参照します。 拡張子の . も含まれます。	.txt
FullPath	ファイルの絶対パスを参照します。 元が相対パスで指定されていても 絶対パスに展開してくれます。	C:\foo\bar\hoge.txt
Directory	ファイルの絶対パスのディレクトリ部分から ルートディレクトリを除いた部分を参照します。 なぜかルートディレクトリは別になっているので RootDir で参照してください。	foo\bar\
RootDir	ファイルの絶対パスの ルートディレクトリだけを参照します。	C:\

表 15. アイテムの既定のメタデータ

RootDir と Directory が分かれているとか拡張子込みのファイル名だけが一発で取れない (Filename と Extension を組み合わせる) とかちょっと気になる部分はありますが、よく使いたいものは用意されているので有効活用しましょう。

あと変換後の文字列にはメタデータだけでなくプロパティの参照も含められます。他と同じように $(プロパティ名) で参照してください。

よく使いそうな例をいくつか挙げときます。

- **ファイル名を絶対パスに展開したい**
  `@(Inputs -> '%(FullPath)')`
- **ファイル名だけを取得したい**
  `@(Inputs -> '%(Filename)%(Extension)')`
- **拡張子を変えた (たとえば .html に) をリストを取得したい**
  `@(Inputs -> '%(RelativeDir)%(Filename).html')`
- **拡張子を変えて別なディレクトリ (たとえば out/) に置くファイル名を取得したい**
  `@(Inputs -> 'out/%(Filename).$(OutputFormat)')`
- **ツリー構造を維持したまま別なディレクトリ (たとえば out/) に置くファイル名を取得したい**
  `@(Inputs -> 'out/%(RecursiveDir)%(Filename)$(Extension)')` この場合、Inputs は C:\foo*** のように ** を含むワイルドカード指定をされている必要があります。
- **ファイルのあるディレクトリのリストを取得したい**
  `@(Inputs -> '%(RelativeDir)')`
- **ファイルのあるディレクトリのリストを絶対パスで取得したい**
  `@(Inputs -> '%(RootDir)%(Directory)')`
  ついでに区切り文字の置き換えもいっしょに使えます。変換の後ろにカンマ , と変えたい区切り

文字をくっつけてください。

- **ファイル名を絶対パスに展開して拡張子を変えたものをスペースでつなげたい**
  `@(Inputs -> '%(RootDir)%(Directory)%(Filename).html', ' ')`

このアイテムの参照や変換は、タスクのパラメータだけではなく、アイテムの参照ができるところならどこでも使えます。アイテムの参照をしてアイテムを定義したりするのにも便利でしょう。前に必要な時だけ実行されるターゲットを定義する時に、出力ファイルリストの定義に使いましたのでもう一度見てみましょう。

```
<ItemGroup>
  <Markdown Include="hoge.md;fuga.md" />
  <Docx Include="@(Markdown->'%(Filename).docx')" />
</ItemGroup>
```

Docx アイテムを Markdown アイテムの拡張子を .docx に変更したものとして定義しています。ただ、これだとちょっと不具合があります。まあこの例だとたまたま動くんですが、汎用的にはもうちょっと気をつけて書いた方が良いでしょう。どこだかわかりますか？

```
<Docx Include="@(Markdown->'%(RelativeDir)%(Filename).docx')" />
```

これが正しいですね。例ではたまたま Markdown アイテムがカレントディレクトリの物しか参照していないのでファイル名だけでも動いてしまいますが、Markdown に別ディレクトリのアイテムが含まれる可能性も考えるとこれが正しいでしょう。もちろん出力ファイルは全てカレントディレクトリに出力したいというのであれば、%(RelativeDir) は付けなくてもいいですが。

4.1.1. ツリー構造を維持したままファイルコピーをしたい

ファイルのコピーをする際に、ツリー構造を維持したままファイルコピーをしたいことがよくあります。

Copy タスクではコピー先のディレクトリを指定はできますが、コピー元のファイルを全部指定したディレクトリに入れるだけで、元のディレクトリ構造を維持はしてくれません。よくやりたくなる作業のわりに一工夫必要になってしまうのですね。

というわけで一工夫ですが、ワイルドカードの ** と上で紹介した RecursiveDir メタデータを使うと上手くいきます。

```
<ItemGroup>
  <CopySources Include="bin/Release/**/*.exe;resources/**/*" />
</ItemGroup>
<Target Name="Release">
  <Copy SourceFiles="@(CopySources)"
DestinationFiles="package/%(RecursiveDir)%(Filename)%(Extension)"/>
</Target>
```

こんな感じで指定します。%(RecursiveDir) には ** でマッチした部分だけが入ってきます。

たとえばこの例で bin/Release/hello.exe と、resources/locale/en_US/hello.po、resources/locale/ja_JP/hello.po というファイルがあったとしましょう。すると、RecursiveDir メタデータはそれぞれ、(空文字列)、locale/en_US/、locale/ja_JP となります。ディレクトリツリー構造が取得できましたね。

あとはコピー先のディレクトリ名を先頭に、ファイル名を末尾に付けてあげると、ディレクトリツリー構造を維持したままのコピー先ファイル名を作ることができます。

```
ディレクトリ "package" を作成しています。
"bin\Release\hello.exe" から "package\hello.exe" へファイルをコピーしています。
ディレクトリ "package\locale\en_US" を作成しています。
"resources\locale\en_US\hello.po" から "package\locale\en_US\hello.po" へファイルをコピーしています。
ディレクトリ "package\locale\ja_JP" を作成しています。
"resources\locale\ja_JP\hello.po" から "package\locale\ja_JP\hello.po" へファイルをコピーしています。
```

実行するとこんな感じになります。思った通りに行きましたね。

よくやりたくなるわりにはちょっと複雑ですが、ファイル名や拡張子を変えつつ参照するなどの柔軟性が高いという利点もあります。** と RecursiveDir メタデータは少なくともこういうのが存在するということだけでも覚えておきましょう。名前は忘れたら調べればいいので。

4.2. 一つの指定でタスクを複数回実行したい - バッチ処理

アイテムの参照方法によって区切り文字の変更などができるようになったので、コマンドラインへのアイテムが渡せるようになりました。でも、これだけで十分でしょうか？

```
<Exec Command="pandoc -t docx -o hon.docx @(Inputs,' ')" />
```

このコマンドだと Inputs アイテム全てを一度に pandoc コマンドに渡して hon.docx を作っています。pandoc コマンドは渡された入力ファイルを全てくっつけて一つのファイルに出力してしまします。入力全部を一つのファイルにする場合は良いのですが、入力ファイルを一つずつ docx に変換したい場合にはこれでは出来ません。pandoc 側に入力のリストを個別に変換するような機能があればなんとかなりますが、残念ながらそんな機能もありませんし、他のアプリケーションでも出来ないものが多いでしょう。

やりたいことを実現するには Exec タスクを入力ファイルの数だけ起動させないといけなさそうです。実は上で説明したメタデータの参照を使うと、それが実現できます。

```
<Exec Command="pandoc -t docx -o %(Inputs.Filename).docx %(Inputs.Identity)" />
```

こんな感じでメタデータの参照をします。アイテムの変換で使ったメタデータの参照とはちょっ

と形が違うので説明が必要ですね。

まず違いは @(アイテム名 ->' 文字列 ') の変換の中に %(メタデータ名) が無いことです。変換の外でメタデータの参照をすると、各アイテムに対して同じタスクを実行するような意味になります。

また、%(メタデータ名) ではなく %(アイテム名 . メタデータ名) になっています。これはアイテム名の指定が無いとどれだかわからんからですね。一つのタスク内で他に @(アイテム名) を使ってアイテムを参照している場合は、%(アイテム名 . メタデータ名) のアイテム名を省略して %(メタデータ名) で参照できます。メタデータ名の指定は必須なので %(アイテム名) という参照はできません。そういうことをしたい場合は %(アイテム名 .Identity) で期待したものになるでしょう。

これで、アイテム内の全てのアイテム回数分、メタデータの置き換えをしてタスクを実行してくれます。

@(アイテム名) と %(アイテム名 . メタデータ名) の違いが難しいですね。Message タスクあたりでいくらかやってみると分かりやすいかと思います。

```
<ItemGroup>
  <Inputs Include="**/*.txt" />
</ItemGroup>
<Message Text="@(Inputs)" />
<Message Text="%(Inputs.Identity)" />
```

こんなのを実行すると、上の @(Inputs) は

```
foo\bar\baz\fuga.txt;foo\bar\hoge.txt;hoge.txt
```

を表示します。Message タスクは一回しか実行されないので、一行にまとめて表示されていますね。

一方で下の %(Inputs.Identity) は

```
foo\bar\baz\fuga.txt
foo\bar\hoge.txt
hoge.txt
```

を表示します。Message タスクが 3 回実行されているので、3 行表示されます。

参照に使う記号がちょっと違うだけでだいぶ動作が変わるので面喰らうかもしれませんが、使いこなすには必須のテクニックですので是非覚えましょう。簡単なプロジェクトファイルを作って動かしながら理解するのが良いと思います。

4.3. プロパティをいろんな形で渡したい - プロパティの参照

アイテムはいろんな形で参照することができましたが、実はプロパティも変換しつつ参照するようなことができます。というか、プロパティの参照では .NET のメソッド呼び出しができてしまいます。

```
<PropertyGroup>
  <SrcDir>C:\Users\kumaryu\Documents\tede-msbuild\src</SrcDir>
</PropertyGroup>
<Target Name="Hoge">
  <Message Text="$(SrcDir)"/>
  <Message Text="$(SrcDir.Replace(\,/))"/>
</Target>
```

$(プロパティ名 . メソッド名 (引数)) という形でメソッドを呼び出せます。プロパティは文字列型 (String 型) なので String クラスのメソッドを呼び出すことができます。引数の部分は型変換が適当に実行されます。引数に文字列を指定する時もそのまま書いちゃってください。また、引数に $(プロパティ名) などを指定すれば展開されますし、メソッド呼び出しを入れ子に指定することも可能です。

実行すると以下のようになります。

```
> msbuild property.proj
...中略...
Hoge:
  C:\Users\kumaryu\Documents\tede-msbuild\src
  C:/Users/kumaryu/Documents/tede-msbuild/src
...後略...
```

上の表示は普通に参照したもので、下は \ を / で置き換えて表示しました。

$() の中にはプロパティだけでなく、.NET の静的メソッド呼び出しや静的プロパティの参照も書くことができます。

```
<Message Text="$([System.IO.Path]::GetTempPath())"/>
<Message Text="$([System.Environment]::OSVersion)"/>
```

$([クラス名]:: メソッド名 (引数)) とすることで静的メソッドの呼び出し、$([クラス名]:: プロパティ名) とすることで静的プロパティの参照ができます。System.IO.Path や System.Environment なんかは使いどころがあるんじゃないかと思います。

上の例ではタスクのパラメータに渡すものだけでしたが、プロパティの参照ができるところならどこでも実行できます。ただ、.NET の呼び出しはいろいろ出来る一方で書き方が複雑になってしまうので、ほどほどにしといた方が良いでしょう。

4.4. 環境変数を参照したい

たまにプロジェクトファイル内で環境変数を参照したいことなんかがあると思います。たとえば別な CI ツールから起動した場合など、ビルド時のバージョン番号が環境変数に入ってくることがよくあります。

そういう時は単にプロパティの参照をすることで環境変数を参照することができます。

```
<Target Name="Build">
  <Message Text="Building Version: $(VERSION)"/>
  <!-- ここになんかビルド処理 -->
</Target>
```

こんな感じでただのプロパティを参照します。

普通に実行すると

```
> msbuild envvar.proj
...中略...
Build:
  Building Version:
...後略...
```

$(VERSION) は定義してないプロパティなので空でした。

しかし、環境変数を設定しておくと……

```
> set VERSION=1.2.3
> msbuild envvar.proj
...中略...
Build:
  Building Version: 1.2.3
...後略...
```

ちゃんと参照できました！全く定義されていないプロパティを参照した場合には環境変数から値を持ってきてくれます。

ただし、プロジェクトの中でプロパティを定義したり、コマンドラインから /p オプションで明示的にプロパティを指定するとそちらが優先されるので注意してください。

このため、プロパティの既定値を指定したい時には……

```
<PropertyGroup>
  <!-- バージョン番号の既定値を指定したい -->
  <VERSION Condition="$(VERSION)==''">0.0.0</VERSION>
</PropertyGroup>
```

こう書くと、値が参照できない (空である) 場合にのみ既定値の定義をするようにします。Condition 属性は条件実行なのですが、詳しくは次で紹介します。

```
<PropertyGroup>
  <!-- バージョン番号の既定値を指定したい -->
  <VERSION Condition="$(VERSION)==''">0.0.0</VERSION>
</PropertyGroup>
<Target Name="Build">
  <Message Text="Building Version: $(VERSION)"/>
  <!-- ここになんかビルド処理 -->
</Target>
```

こうしておくことで、環境変数やコマンドラインからの指定がない時だけ既定値を使うようになります。

```
> msbuild envvar.proj
...
Build:
  Building Version: 0.0.0
...

> set VERSION=1.2.3
> msbuild envvar.proj
...
Build:
  Building Version: 1.2.3
...
```

もし複雑なプロジェクトファイルで、既に環境変数と同じ名前のプロパティが定義されているが、どうしても環境変数の方を参照したいといった時には、`$([System.Environment]::GetEnvironmentVariable(` 環境変数名 `))` の力技で参照すると良いでしょう。

4.5. 既定のプロパティ

プロパティには名前が `MSBuild` で始まる既定の物があり、これらは定義しなくても最初から参照できるようになっています。

よく使えそうなものをいくつか挙げておきます。

プロパティ名	内容
MSBuildStartupDirectory	msbuild を起動した時点でのカレントディレクトリの絶対パスが入ります。 例 : C:\Users\kumaryu\Documents\tede-msbuild
MSBuildProjectFullPath	起動したプロジェクトファイルの絶対パスが入ります。 例 : C:\Users\kumaryu\Documents\tede-msbuild\hoge.proj
MSBuildProjectDirectory	起動したプロジェクトファイルの絶対パスのうち、ディレクトリ部分だけが入ります。 例 : C:\Users\kumaryu\Documents\tede-msbuild
MSBuildProjectFile	起動したプロジェクトファイルのファイル名のみが入ります。 例 : hoge.proj
MSBuildThisFileFullPath	現在実行しているターゲットが書かれているプロジェクトファイルの絶対パスが入ります。 例 : C:\Users\kumaryu\Documents\tede-msbuild\sub\fuga.targets

プロパティ名	内容
MSBuildThisFileDirectory	現在実行しているターゲットが書かれているプロジェクトファイルの絶対パスのうち、ディレクトリ部分だけが入ります。 例 : C:\Users\kumaryu\Documents\tede-msbuild\sub
MSBuildThisFile	現在実行しているターゲットが書かれているプロジェクトファイルのファイル名のみが入ります。 例 : fuga.targets
MSBuildToolsPath	MSBuild のインストールされているパスが入ります。 例: C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\MSBuild\15.0\Bin
MSBuildExtensionsPath	MSBuild のインストールされているパスの上の方が入ります。 例: C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\MSBuild
MSBuildToolsVersion	MSBuild のバージョンが入ります。 例 : 15.0

表 16. 既定のプロパティ

ディレクトリが入る既定のプロパティの値には、最後にパス区切り文字が付きません。

MSBuildProjectFullPath と MSBuildThisFileFullPath(および関連する既定のプロパティ) の違いですが、あとで説明する別なプロジェクトファイルの読み込みを使った時に影響してきます。

4.6. 特定の値の時だけ違うことしたい - 簡単な条件実行

プロジェクトファイルをいじっていると、特定の値の時だけ違う処理をしたいんだけどなぁと思うことがあります。たとえばコマンドラインからプロパティを指定することで動作を変えたいといった場合です。そんな場合のための条件実行の機能が用意されています。

MSBuild ではいろんな XML の要素に Condition 属性を付けることで、条件が成立した時のみ指定できるようになります。

```
<PropertyGroup>
  <CopyBin>false</CopyBin>
</PropertyGroup>
<ItemGroup>
  <Sources Include="src/*.c" />
  <Binaries Include="bin/*.exe" />
</ItemGroup>
<Target Name="Release">
  <Copy SourceFiles="@(Sources)" DestinationFolder="release/src"/>
  <Copy SourceFiles="@(Binaries)" DestinationFolder="release/bin"
Condition="$(CopyBin)"/>
</Target>
```

上の例では最後の Copy タスクに Condition 属性を指定しています。Condition 属性は中に書いた条件が true の時のみ XML 要素が実行されます。ここでは CopyBin プロパティが true の時のみ bin/*.exe ファイルを release/bin にコピーしようとしています。

普通に実行すると次のようになります。

```
> msbuild conditional.proj
... 中略 ...
Release:
  ディレクトリ "release\src" を作成しています。
  "src\hello.c" から "release/src\hello.c" へファイルをコピーしています。
プロジェクト "conditional.proj" (既定のターゲット) のビルドが完了しました。
... 後略 ...
```

src/hello.c から release/src/hello.c だけがコピーされていますね。bin/hello.exe も存在はしているのですが、$(CopyBin) が false のためコピーが実行されていません。

コマンドラインから CopyBin プロパティに true を設定してバイナリファイルもコピーされるようにしてみましょう。

```
> msbuild conditional.proj /p:CopyBin=true
... 中略 ...
Release:
  "src\hello.c" から "release/src\hello.c" へファイルをコピーしています。
  ディレクトリ "release\bin" を作成しています。
  "bin\hello.exe" から "release/bin\hello.exe" へファイルをコピーしています。
プロジェクト "conditional.proj" (既定のターゲット) のビルドが完了しました。
... 後略 ...
```

このように Condition に true が入っている時だけ実行されるタスクができました。

Condition 属性はタスクだけでなく他の場所にもつけられます。

```
<PropertyGroup>
  <CopyBin>false</CopyBin>
</PropertyGroup>
<ItemGroup>
  <Inputs Include="src/*.c" />
  <Inputs Include="bin/*.exe" Condition="$(CopyBin)" />
</ItemGroup>
<Target Name="Release">
  <Copy SourceFiles="@(Inputs)" DestinationFiles="release/%(Inputs.Identity)"/>
</Target>
```

この例では Inputs アイテムにつけています。やってることは前とだいたい同じです。さっきの例では *.exe ファイルをコピーするタスクを条件によって実行するしないを切り替えていたんですが、こちらではコピー元として列挙するかどうかを切り替えています。CopyBin が false の場合は 2 つめの Inputs 要素が無かったことにされます。

Condition 属性はだいたいどの要素にも付けることができ、条件が true になればその要素が存在することに、false であればその要素がまるごと無かった扱いにされます。たとえばアイテムやタスクなど一つずつの要素でなく、ItemGroup なんかに付けることもできます。

```
<PropertyGroup>
  <CopyBin>false</CopyBin>
</PropertyGroup>
<ItemGroup>
  <Inputs Include="src/*.c" />
</ItemGroup>
<ItemGroup Condition="$(CopyBin)">
  <Inputs Include="bin/*.exe" />
  <Inputs Include="bin/*.pdb" />
</ItemGroup>
<Target Name="Release">
  <Copy SourceFiles="@(Inputs)" DestinationFiles="release/%(Inputs.Identity)"/>
</Target>
```

この例では ItemGroup につけています。CopyBin が true の時のみ bin/*.exe と bin/*.pdb が Inputs アイテムに追加されます。他に Condition 属性は PropertyGroup でも Target でもだいたいどんな要素にもくっつけることができます。

Condition 属性の条件に書く式は単純な true か false に展開できる値の他に、簡単な条件式も書くことができます。

条件	説明
' 文字列 '	文字列を直接書きたい時はシングルクォーテーション で括ってください。'' だと空文字列を表します。
' 文字列 A' == ' 文字列 B'	文字列 A と文字列 B が等しい時に true になります。
' 文字列 A' != ' 文字列 B'	文字列 A と文字列 B が等しくない時に true になります。

条件	説明
数値 A < 数値 B	数値 A が数値 B より小さい時に true になります。 数値は 0x を頭に付けると 16 進数で指定できます。
数値 A > 数値 B	数値 A が数値 B より大きい時に true になります。 数値は 0x を頭に付けると 16 進数で指定できます。
数値 A <= 数値 B	数値 A が数値 B 以下の時に true になります。 数値は 0x を頭に付けると 16 進数で指定できます。
数値 A >= 数値 B	数値 A が数値 B 以上の時に true になります。 数値は 0x を頭に付けると 16 進数で指定できます。
Exists(' 文字列 ')	文字列で指定したファイルが存在する場合に true になります。
HasTrailingSlash(' 文字列 ')	文字列の末尾が / か \ で終わっている場合に true になります。
! 式	式が false の時に true に、式が true の時は false になります。
式 A AND 式 B	式 A と式 B が両方 true の時に true になります。
式 A OR 式 B	式 A と式 B のいずれかが true の時に true になります。
(式)	括弧です。難しく言うと式のグルーピング化です。

表 17. Condition で使える条件

あんまり凝ったことをしようとすると大変になるので、だいたい == と !=、あと AND と OR だけあれば足りると思います。

他に条件式の例としてよくありそうなのはこんなんです。

```
<PropertyGroup>
  <DestDir>Win32Debug</DestDir>
  <DestDir Condition="$(Configuration)=='Debug'   AND
$(Platform)=='Win32'">Win32Debug</DestDir>
  <DestDir Condition="$(Configuration)=='Release' AND
$(Platform)=='Win32'">Win32Release</DestDir>
  <DestDir Condition="$(Configuration)=='Debug'   AND
$(Platform)=='x64'">Win64Debug</DestDir>
  <DestDir Condition="$(Configuration)=='Release' AND
$(Platform)=='x64'">Win64Release</DestDir>
</PropertyGroup>
```

ビルドの設定によって出力先が変わるようなのですね。一番上に条件が書いてないのがあるのは、どの条件も成立しなかった時用のデフォルト値です。他の条件が成立したら上書きされて欲しいので一番上に書いてありあす。

条件にはプロパティだけでなく、アイテムやアイテムのメタデータを入れることもできます。

```
<Target Name="Copy">
  <Message Text="%(Inputs.Identity)は既にあるのでコピーしません"
Condition="Exists('release/%(Inputs.Filename)%(Inputs.Extension)')" />
  <Copy SourceFiles="%(Inputs.Identity)" DestinationFolder="release"
Condition="!Exists('release/%(Inputs.Filename)%(Inputs.Extension)')" />
</PropertyGroup>
```

あまり意味のある例ではなくてなんですが、コピー先にファイルが既にあったら上書きせずにメッセージを出す動作をしています。%(Inputs.Filename) で Inputs アイテムの Filename メタデータを参照するのは前もやってるのでわかると思いますが、Condition 属性の中でも使えます。条件の中に文字列を書くにはシングルクォーテーションで括る必要があるのには注意してください。単一の変数の展開だと括るのを省略しても動くのですが、複数の展開したのをくっつけるといった場合には全部をシングルクォーテーションで括ります。

上の例で気付いたかもしれませんが、Condition 属性では条件が成立した時に存在するという指定しかできません。普通のプログラミング言語でいう else に相当する機能はありませんので、そういうのが必要なら条件が反対に評価される要素を並べて書くといった方法で対応してください。

Choose・When・Otherwise といった要素を使う複雑な条件分岐を使えばできるのですが、だいたいは Condition 属性で足りると思うのでここでは省略します。そんなに難しいもんではないので気になったら調べてみてください。

4.7. 一部のファイルだけ違うことしたい - アイテムのメタデータ

アイテムで指定したファイルの中で特定のものだけ違う処理をしたいことがあります。たとえば特定のファイルだけオプションを変えてコンパイルしたいといった場合です。条件分岐で頑張ることもできるのですが、ファイル毎に条件の違うタスクを書くのは大変ですし、分かりづらくなります。そういう時にはアイテムのメタデータを使うことができます。

メタデータは %(hoge.Identity) や %(hoge.Filename) といった形で参照できる、アイテムのファイル名の変形みたいなものでしたが、これらは既定のものです。既定のものの他に自分でアイテム毎に付属するデータを作ることができます。

```
<ItemGroup>
  <Inputs Include="chapter*.md">
    <StyleSheet>default.css</StyleSheet>
  </Inputs>
  <Inputs Include="index.md">
    <StyleSheet>index.css</StyleSheet>
  </Inputs>
</ItemGroup>
<Target Name="Build">
  <Exec Command="pandoc -t html5 -css css/%(Inputs.StyleSheet) -o html/%(Inputs.
Filename).html @(Inputs)" />
</Target>
```

この例では pandoc コマンドで Markdown を HTML に変換していますが、各章のページとインデックスのページで参照する css ファイルを変えたいので、アイテムのメタデータとして参照する css ファイル名を追加しています。Inputs アイテム定義のタグの子として、StyleSheet タグが書かれていますが、こいつが StyleSheet メタデータの定義です。ここでは css ファイル名を書きた

いので StyleSheet としていますが、名前は好きにつけてかまいません。

参照する時には、既定の物と同じように %(アイテム名 . メタデータ名) として参照できます。例では %(Inputs.StyleSheet) としていますが、これで各 Inputs の StyleSheet メタデータの中身が参照できます。メタデータを参照した内容は、定義したタグ内の文字列そのままになります。

実行すると次のようになります。

```
> msbuild metadata.proj
... 中略 ...
Build:
  pandoc -t html5 -css css/default.css -o html/chapter1.html chapter1.md
  pandoc -t html5 -css css/default.css -o html/chapter2.html chapter2.md
  pandoc -t html5 -css css/default.css -o html/chapter3.html chapter3.md
  pandoc -t html5 -css css/default.css -o html/chapter4.html chapter4.md
  pandoc -t html5 -css css/index.css -o html/index.html index.md
プロジェクト "metadata.proj" (既定のターゲット) のビルドが完了しました。
... 後略 ...
```

うまいこと chapter1 ～ 4.html には default.css が、index.html には index.css が適用されるように指定できました。条件実行などは使ってないので、比較的すっきり書けるのが良いですね。

4.7.1. 既定のメタデータを作る

メタデータを定義するにはアイテムのタグ内に書く必要がありました。ファイルやメタデータが一つや二つなら大したことないんですが、それらが沢山あると書くのが面倒になります。一部のファイルだけちょっと違う処理をするのに楽をしたい機能のはずなのに、たくさん書くのは本末転倒ですね。

そこで、メタデータの既定値を定義しておける機能を使います。Project 要素の下に ItemDefinitionGroup 要素を入れましょう。

```
<ItemDefinitionGroup>
  <Inputs>
    <StyleSheet>default.css</StyleSheet>
    <AdditionalOptions>-toc</AdditionalOptions>
  </Inputs>
</ItemDefinitionGroup>
<ItemGroup>
  <Inputs Include="chapter*.md"/>
  <Inputs Include="index.md">
    <StyleSheet>index.css</StyleSheet>
  </Inputs>
</ItemGroup>
<Target Name="Build">
  <Exec Command="pandoc -t html5 -css css/%(Inputs.StyleSheet) -o html/%(Inputs.
Filename).html %(Inputs.AdditionalOptions) @(Inputs)" />
</Target>
```

ItemGroup でなく ItemDefinitionGroup タグの中にアイテムとメタデータの定義を書きます。その際、アイテムのタグには Include 属性をつけません。ItemDefinitionGroup では既定値の定義だけですのでファイル名の指定はしません。ここでは StyleSheet メタデータに default.css を、AdditionalOptions メタデータに -toc を既定値で設定しています。

アイテム自体の定義の方はいつも通り書きます。chapter*.md には特にメタデータを指定していませんが、ItemDefinitionGroup で設定した既定値が適用されます。index.md の方は StyleSheet メタデータに index.css を指定しています。アイテムの定義でメタデータを指定すると既定値を上書きしますが、指定してないメタデータはそのまま既定値が入るので、index.md アイテムにも AdditionalOptions メタデータが設定された扱いになります。

4.8. 他のプロジェクトを流用したい - プロジェクトのインポート

プロジェクトファイルを作り込んでいくと、一度作ったプロジェクトファイルを流用して、追加部分だけ書きたいなぁという要求が出てくると思います。コピペして変更部分だけいじることもできますが、それは格好悪いと思うのはプログラマーなら当然のことです。

もちろん MSBuild にも流用の仕組みは用意されています。Import 要素で別なファイルを読み込むことができます。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="Other.proj"/>
  ... 中略 ...
</Project>
```

こんな感じで Import 要素の Project 属性に読み込みたいファイルを指定するだけです。これで指定したファイルの中身をこの位置にコピペしたのと同じような動作をします。簡単ですね。

インポートするプロジェクトファイルはなんでもいいのですが、プロパティ定義をまとめたファイルとターゲット定義をまとめたファイルに分けられることが多いようです。その際、プロパティ定義をまとめたファイルは拡張子が .props、ターゲット定義をまとめたファイルは拡張子が .targets とされます。どちらにしても中身はただのプロジェクトファイルで、拡張子によって違いはありません。

5. プログラムをビルドしよう - .NET のプロジェクト

ここまでは主に外部コマンドを使ったプロジェクトの書き方を解説してきましたが、MSBuild はやっぱりプログラムのビルドに使われるのがメインなものですので、プログラムのビルドをするプロジェクトファイルについても説明します。

とはいえ、プログラムをビルドするプロジェクトファイルは Visual Studio や .NET Core SDK だと dotnet なんかのツールで作るのが普通でしょうから、いちから手で書けるほどの解説はしません。中身をちょっといじったり流用する方法を覚えていきましょう。

5.1. .NET Framework のプロジェクトファイル

まずは Visual Studio で作ってみた .NET Framework のプロジェクトを見てみましょう。

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/
msbuild/2003">
  <Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.
Common.props" />
...中略...
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="System.Core" />
...中略...
  </ItemGroup>
  <ItemGroup>
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
  </ItemGroup>
...中略...
  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
</Project>
```

Visual Studio 2017 で作った C# プロジェクトの .csproj ファイルです。拡張子こそ .csproj ですが、今まで見てきた MSBuild プロジェクトと大きく変わるところはありませんね。ところどころ省略してあるところはプロパティやアイテムの定義で、長いので省略しました。

Import 要素でいくつかファイルを読み込んでいる以外はプロパティとアイテムの定義くらいしか書いてなく、ターゲットやタスクの記述がありません。実際のビルドはどこでやっているのかというと、インポートした $(MSBuildToolsPath)\Microsoft.CSharp.targets に書かれています。ちなみに上で読み込んでいる $(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\

Microsoft.Common.props は既定のプロパティを定義しているファイルです。

Microsoft.CSharp.targets をインポートするだけで、Build・Rebuild・Clean などのターゲットが定義され、Compile アイテムに C# のソースを指定してしておくだけでビルドができるようになっています。

いくつかアイテムが定義してありますが、既定のターゲットでは決まった名前のアイテムを参照してビルドしてくれます。Reference アイテムは参照するアセンブリ (ライブラリ) の指定、Compile アイテムはコンパイルするソースファイルの指定ですね。省略してありますが、他には app.config など特に処理しないアイテムとして None というのも入っています。

ここでは C# のプロジェクトを見てみましたが、他の言語でも基本的な構造は同じで、.props ファイルから既定のプロパティ定義を、.targets からターゲットの定義を読み込み、プロジェクトファイル自体にはプロパティとアイテムの定義だけが書かれる形になっています。

5.2. .NET Core のプロジェクトファイル

.NET Core SDK で作ったプロジェクトファイルはまた構造が違うので見てみましょう。.NET Core SDK の dotnet コマンドからプロジェクトを作った場合や、Visual Studio 2017 で .NET Core のプロジェクトを作った場合にこの形式になります。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

省略無しでこれだけでした。これだけでプログラムが正しくビルドされます。

今まで見たことないものとして、Project 要素に Sdk 属性が付いてますね。これは Import 要素と同じようなもので、Sdk 属性に指定したフォルダ内の .props と .targets ファイルをインポートしてくれます。複数の SDK を使う必要がある場合はセミコロンで区切ると沢山指定できます。

ここではどこかにある Microsoft.NET.Sdk フォルダ内の .props と .targets ファイルをインポートしています。

ちょっと不思議なのはターゲットの定義どころか、アイテムの定義すらないことです。Compile アイテムすらないので、どのファイルをコンパイルしたらいいか分からないんじゃないでしょうか？

実は Microsoft.NET.Sdk でインポートしたプロジェクトの中に次のような定義がされています。

```
<ItemGroup>
  <Compile Include="**/*.cs"/>
</ItemGroup>
```

実際の記述はもっとややこしいんですが、分かりやすく書くとこんな定義です。

標準でプロジェクトファイル以下の .cs ファイルを全部コンパイルするようにしています。確かにだいたいこれで足りますね。

既定の記述を Sdk でインポートできる .props や .targets にてんこもりにすることで、プロジェクトファイル側に書くことをなるべく減らすのが .NET Core SDK のプロジェクトファイルのスタイルです。これのおかげでプロジェクトファイルを手でいじるのも可能なくらいシンプルになっています。

ところで Sdk でインポートできるファイルがどこにあるかなんですが、標準の SDK は MSBuild がインストールされている場所あたりの Sdks ディレクトリに入っています。普通の msbuild と dotnet msbuild で違うんですが、普通の MSBuild だと C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\MSBuild\Sdks あたりに、.NET Core SDK の MSBuild だと C:\Program Files\dotnet\sdk\2.1.4\Sdks や /usr/local/share/dotnet/sdk/2.1.4/Sdks に入ってます。.NET Core SDK では dotnet --info コマンドでインストールされているパスが表示されます。

標準で入った SDK の他に、NuGet で入れたパッケージの中に Sdk(または sdk) ディレクトリがあれば、そこからも SDK を読み込むことができます。この場合は Sdk 属性に NuGet パッケージ名を書いておいてください。また、次で説明する NuGet パッケージの参照をプロジェクトに書いておけば、プロジェクトの中で参照しているパッケージを Sdk に指定してビルドするということもできます。

5.3. 外部ライブラリを参照したい - NuGet パッケージの参照

今時の .NET のプログラムだとなんか NuGet のパッケージ拾ってこないとやってられないですよね。

既定のターゲットでは NuGet パッケージの参照ができるようになってるので、簡単に参照を追加することができます。

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="11.0.1" />
</ItemGroup>
```

こういうのをどこかに入れとくだけで OK です。実際のパッケージのダウンロードとインストールは msbuild /restore(または msbuild /r) で実行できます。.NET Core SDK だと dotnet msbuild /restore でなく dotnet restore でもかまいません。msbuild /t:Restore でも似たようなことができますが、NuGet パッケージ内に .props や .targets ファイルがある場合に上手く動かない場合があります。できるだけ /restore オプションか、dotnet restore を使ってください。

msbuild /restore(または msbuild /r) では /t:Build などを指定することでそのままビルドもできます。msbuild /restore だけだとパッケージのリストアをしただけで終わりますが、msbuild /r /t:Build とすると、パッケージのリストアをしてその後ビルドもしてくれるので便利です。後述しますが、msbuild /t:Restore;Build は上手く動かないことがありますのでやらないでください。

Visual Studio で作った、packages.config が使われているプロジェクトでは PackageReference アイテムは生成されません。この場合は msbuild /restore を実行しても何も起きないので気をつけましょう。packages.config が使われている場合は nuget restore コマンドでリストアしてください。

Visual Studio 2017 以降ですと、packages.config が使われているプロジェクトを PackageReference 形式に変換したり、プロジェクトのデフォルトのパッケージ参照形式を PackageReference 形式に変更したりできます。msbuild /restore を使いたい場合は Visual Studio 上から PackageReference 形式に変換すると良いでしょう。

5.4. 既存のプロジェクトを活用しよう

プログラムをビルドしようとか言って .NET Framework や .NET Core 用のプロジェクトファイルの解説を軽くしておいてなんですけど、実際にビルドする部分ってあんま手で書かないと思うんですよ。というのも既に解説した通り、既定の .props や .targets ファイルにがっつり書かれているので、その部分を自分で書き起こすのは無駄な努力でしょう。自分で Sdk 作りたいなら必要だと思いますが、それはこの本で解説する範囲を越えています。

ここではむしろプログラムのビルドそのものより、前処理や後処理を追加していくことを覚えていきましょう。

5.4.1. 前処理や後処理を追加したい - ターゲットのフック

ビルドする前に処理を追加したいとしましょう。たとえばバージョン番号をソースに埋め込みたいとかよくあると思います。

ビルドする前なので Build ターゲットの前に実行したいですね。

```
<PropertyGroup>
  <VersionNumber>1.2.3.4</VersionNumber>
</PropertyGroup>
<Target Name="Build">
  <!-- 外部プログラムでバージョン番号の設定をする -->
  <Exec Command="ruby replaceversion.rb $(VersionNumber)" />
  <!-- 本来のビルド処理する -->
  <MSBuild Projects="MyProgram.csproj" Targets="Build" />
</Target>
```

簡単に思いつくのはこんな感じです。hoge.proj とか適当なプロジェクトファイルにこれを書いといて、ビルドする時に本来の MyProgram.csproj でなく hoge.proj をビルドするようにします。MyProgram.csproj の中に直接 Build ターゲットを定義してしまうとデフォルトの Build ターゲットを置き換えてしまうので、別なファイルにして置き換えを回避しています。

これで十分目的は達成できますけど、別なファイルになるのはちょっと面倒ですね。まあ .NET Framework のプロジェクトファイルは直接いじったら大変なことになるので別なファイルにしておくべきなんですけど、.NET Core SDK のプロジェクトファイルだとシンプルなので同じファイルに入れておきたいところです。

そこでターゲットのフックを使います。指定したターゲットの前や後に必ず実行されるターゲットを作ることができます。

```
<PropertyGroup>
  <VersionNumber>1.2.3.4</VersionNumber>
</PropertyGroup>
<Target Name="SetVersionNumber" BeforeTargets="Build">
  <Exec Command="ruby replaceversion.rb $(VersionNumber)" />
</Target>
```

新しいターゲットとして SetVersionNumber というのを作りましたが、初めて見る BeforeTargets 属性というのに Build が指定されています。中身は本来のビルド処理の呼び出しが無くなって前処理だけになりました。

Target 要素に BeforeTargets 属性を設定すると、指定したターゲットが実行されそうになった時にはいつも割り込みでこいつが先に実行されるようになります。

逆に指定したターゲットの後に実行したいターゲットは AfterTargets 属性を付けます。

```
<Target Name="CopyOutputs" AfterTargets="Build">
  <Copy SourceFiles="@(OutputFiles)" DestinationFolder="foo/bar"/>
</Target>
```

CopyOutputs ターゲットは Build の後に実行されます。

簡単ながら完全な例を見てみましょう。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
DefaultTargets="Build">
  <Target Name="Build">
    <Message Text="ビルドしてるつもり..." />
  </Target>
  <PropertyGroup>
    <VersionNumber>1.2.3.4</VersionNumber>
  </PropertyGroup>
  <Target Name="SetVersionNumber" BeforeTargets="Build">
    <Message Text="バージョン番号を設定したい: $(VersionNumber)" />
  </Target>
  <Target Name="CopyOutputs" AfterTargets="Build">
    <Message Text="ビルド結果をコピーするつもり" />
  </Target>
</Project>
```

シンプルにメッセージ表示してるだけになっちゃってますが、ターゲットの中身は上から順番に実行されるだけなので簡単な例でも大丈夫です。ビルド前にバージョン番号を設定し、ビルドして、ビルド結果をどこかにコピーしています。全部つもりですけど。

```
> msbuild hook.proj /t:Build
... 中略 ...
ノード 1 上のプロジェクト "hook.proj" (Build ターゲット)。
SetVersionNumber:
  バージョン番号を設定したい: 1.2.3.4
Build:
  ビルドしてるつもり...
CopyOutputs:
  ビルド結果をコピーするつもり
プロジェクト "hook.proj" (Build ターゲット) のビルドが 完了しました。
... 後略 ...
```

実行するとこんな感じになります。Build ターゲットを指定しただけですが、SetVersionNumber が先に実行されて、ビルド後には CopyOutputs が実行されました。この例では簡単のために Build ターゲットが同じファイルに入っちゃってるので、そんな書き方しなくても直接書けよって気分になりますが、実際のプロジェクトでは Sdk や Import の指定で Build ターゲットとかいろいろをどこかのファイルからインポートしてくるんですよ！

フックする側のターゲットの BeforeTargets や AfterTargets 属性にフックされる側のターゲット名を書くのに注意してください。たまにフックする側に書くんだっけ？される側に書くんだっけとこんがらがりそうになりますが (ならない？俺はなる)、フックされる側は手を出せないファイルに書いてある場合に使うもの、と覚えておけば間違わないでしょう。

フックされる側が手を出せるファイルにある場合は、素直にフックされる側に DependsOnTargets で依存関係を追加した方が良いと思います。

5.4.2. 実行結果を受け取りたい - ターゲットからの値の出力

後処理とかでビルド結果の実行ファイルをコピーしたい時とかあると思います。あるよね。

```
<Target Name="Package">
  <!-- ビルドする -->
  <MSBuild Projects="$(MSBuildProjectFullPath)" Targets="Build"
Properties="Configuration=Release"/>
  <!-- コピーしたい -->
  <Copy SourceFiles="何をコピーすればいいんだ？" DestinationFolder="package" />
</Target>
```

リリース用に zip ファイルか何か作ろうとして package ディレクトリにビルド結果をコピーしようとしています。しかし何をコピーしたらいいんでしょうかね。**/*.exe とか指定してもいいんですが、bin/Debug/MyProgram.exe と bin/Release/MyProgram.exe が出来てた時にどっちが入るかわからなくてやばいですね。bin/Release/MyProgram.exe を直接指定してもいいんですが、ちょっとかっこ悪いのでもうすこし汎用的になんとかならないでしょうか。

なんとかならないでしょうか、なんて言うってことは当然なるわけですね。

ターゲットは関数のように戻り値を設定することができ、Build ターゲットはビルド結果のファイルを戻り値に指定してくれます。こいつを受け取ることでビルド結果のファイルを列挙することができます。

```
<Target Name="Package">
  <MSBuild Projects="$(MSBuildProjectFullPath)" Targets="Build"
Properties="Configuration=Release">
    <Output TaskParameter="TargetOutputs" ItemName="BuildOutputs"/>
  </MSBuild>
  <Copy SourceFiles="@(BuildOutputs)" DestinationFolder="package" />
</Target>
```

こんな感じなりました。今まで空だった MSBuild タスク要素の中に Output 要素という新しいのが出てきました。Output 要素はタスクの戻り値をアイテムやプロパティに代入するのを指定する要素です。TaskParameter 属性にタスクの出力パラメータ名、ItemName 属性に代入先のアイテム名を指定します。

上の例では、TargetOutputs パラメータの値を BuildOutputs アイテムに代入しています。MSBuild タスクでは実行したターゲットの戻り値を TargetOutputs 出力パラメータに入れてくれます。代入先のアイテム名は自分で決める名前なのでなんでもいいですが、ここでは BuildOutputs という名前にしています。指定したアイテムが既に定義されていればそこに値を追加しますし、定義されていなければ新しくアイテムを作って追加します。

その後で @(BuildOutputs) として出力されたアイテムを参照してファイルをコピーしています。新しく作られたアイテムですが、普通のアイテムと同じように参照できます。

出力された値をアイテムでなくプロパティに代入した場合には Output 要素の ItemName 属性のかわりに PropertyName 属性を使います。既存のプロパティに出力値を代入した場合は、アイテムと違って追加でなく上書きされます。

ここでは MSBuild タスクの TargetOutputs 出力パラメータから値を取ってきましたが、もちろん他のタスクの出力パラメータも同様に取得することができます。使いそうな出力パラメータを持つタスクってあんまり無いんですが、Exec タスクの ExitCode 出力パラメータや、ConsoleOutput 出力パラメータなんかは使えることもあるんじゃないかと思います。

逆にターゲットから値を返す方法も見ておきましょう。

```
<ItemGroup>
  <InputFiles Include="*.md" />
</ItemGroup>
<Target Name="Build" Returns="@(OutputFiles)">
  <Exec Command="pandoc -t docx -o %(InputFiles.Filename).docx @(InputFiles)" />
  <ItemGroup>
    <OutputFiles Include="*.docx" />
  </ItemGroup>
</Target>
<Target Name="Package">
  <MSBuild Projects="$(MSBuildProjectFullPath)" Targets="Build">
    <Output TaskParameter="TargetOutputs" ItemName="BuildOutputs"/>
  </MSBuild>
  <Copy SourceFiles="@(BuildOutputs)" DestinationFolder="package" />
</Target>
```

上の Target 要素に Returns 属性が付きました。ここに指定された文字列 (アイテム) がターゲットの戻り値として返されます。Returns 属性で @(OutputFiles) を参照してるのが OutputFiles の定義より上なのが気になりますが、ちゃんとターゲットの中が全部実行されたあとで Returns 属性の中身が評価されるようなので、これで大丈夫です。

```
> msbuild outputs.proj /t:Package
... 中略 ...
Build:
  pandoc -t docx -o fuga.docx fuga.md
  pandoc -t docx -o hoge.docx hoge.md
プロジェクト "outputs.proj" (Build ターゲット) のビルドが完了しました。

Package:
  "fuga.docx" から "package\fuga.docx" へファイルをコピーしています。
  "hoge.docx" から "package\hoge.docx" へファイルをコピーしています。
プロジェクト "outputs.proj" (Package ターゲット) のビルドが完了しました。
... 後略 ...
```

Target 要素に Outputs 属性を指定している場合には、既定で Outputs 属性に指定した値が戻り値に設定されます。ただ Outputs 属性はターゲットの実行前に評価されてしまうので、この例のように実行時にファイル名を列挙する形では使えませんね。

必要な時だけ実行されるターゲットのところで説明しましたが、Outputs 属性を指定する場合は事前に出力ファイル名を列挙しておく形で指定しましょう。

```
<ItemGroup>
  <InputFiles Include="*.md" />
</ItemGroup>
<Target Name="Build" Inputs="@(InputFiles)" Outputs="@(InputFiles ->
'%(Filename).docx')">
  <Exec Command="pandoc -t docx -o %(InputFiles.Filename).docx @(InputFiles)" />
</Target>
<Target Name="Package">
  <MSBuild Projects="$(MSBuildProjectFullPath)" Targets="Build">
    <Output TaskParameter="TargetOutputs" ItemName="BuildOutputs"/>
  </MSBuild>
  <Copy SourceFiles="@(BuildOutputs)" DestinationFolder="package" />
</Target>
```

この形だと、Build ターゲットは入力ファイルが更新されている時だけ実行されますが、実行が省略された場合でも戻り値は正しく設定されます。

```
> msbuild outputs.proj /t:Package
... 中略 ...
Build:
すべての出力ファイルが入力ファイルに対して最新なので、ターゲット "Build" を省略します。
プロジェクト "outputs.proj" (Build ターゲット) のビルドが完了しました。

Package:
  "fuga.docx" から "package\fuga.docx" へファイルをコピーしています。
  "hoge.docx" から "package\hoge.docx" へファイルをコピーしています。
プロジェクト "outputs.proj" (Package ターゲット) のビルドが完了しました。
... 後略 ...
```

実行が省略されましたが、戻り値はちゃんと受け取れているのがわかります。

5.4.3. 既存のプロジェクトファイルを活用したい - 既存のターゲット

ターゲットの実行前後になにやら追加の処理をしたり、実行結果を受け取る方法は分かったのですが、普通にプログラムをビルドするプロジェクトファイルで使える既存のターゲットには何があってどんな値を戻してくるのでしょうか？

これを調べる方法ですが、なんと……ありません！

どんなターゲットがあるのかくらい表示できれば良かったんですけど、無いんですね。どういうことでしょうね。仕方がないのでインポートしているファイルなどを全部辿って調べていく必要があります。

インポートされている .targets ファイルを追っていくのはめちゃくちゃめんどくさいですが、

/preprocess オプションを使うとインポートを全部処理してくれるので助けになります。msbuild /preprocess:tmp.xml のようにすると tmp.xml にインポートを全部解決した単一のプロジェクトファイルを出力してくれます。ここから読み解くのが楽でしょう。

ある程度は頑張って調べておきましたので紹介します。

ターゲット名	説明
Build	普通にビルドを実行します。 ビルド結果のファイルを返します。
Clean	ビルドで生成される中間ファイルやビルド結果を削除します。
Rebuild	生成物を消してからビルドを実行します。 Clean してから Build と同じです。 ビルド結果のファイルを返します。
Run	ビルド結果の実行ファイルを実行します。 ビルド結果が実行ファイルでない場合は失敗します。 試した限りでは .NET Core 用のアプリケーション DLL でも失敗しますね……。
Restore	NuGet パッケージの復元を行います。 PackageReference アイテムで参照したものだけ復元されます。 packages.config を使っている場合は何も起きません。 msbuild /restore や dotnet restore から呼び出されます。
Publish	ビルドしてから発行を実行します。 .NET Core SDK の場合は dotnet publish でも実行できるようです。
GetTargetPath	ビルドを実行せずにビルドの結果生成されるファイル (.exe や .dll) を返します (中間ファイルは除く)。 ビルドが一度も実行されていない状態からでも生成されるファイルを返してくれます。
GetCopyToOutputDirectoryItems	出力ディレクトリにコピーされるファイルを返します。
GetCopyToPublishDirectoryItems	発行用のディレクトリにコピーされるファイルを返します。
GetResovledLinkLibs	生成される .lib ファイルを返します。 Visula C++ のプロジェクトでのみ使えます。 静的ライブラリを生成するプロジェクトの場合はその .lib を返すので GetTargetPath と同じですが、DLL を生成するプロジェクトの場合はインポートライブラリの .lib ファイルを返すのでちょっと便利です。
GetResolvedWinMD	生成される .winmd ファイルを返します。 Visual C++ のプロジェクトでのみ使えます。 .NET で WinRT コンポーネントを作った場合は GetTargetPath で取得できます。

表 18. 使える既定のターゲット

Get ～といった名前のターゲットがいくつかありますが、これらは単純に実行しても何も起きません。MSBuild タスクで呼び出して Output 要素で戻り値を取得するのを前提としたターゲットです。ファイルを作るなどの副作用がなく単純に生成されるファイル (やソースファイル) の列挙ができます。Build などのターゲットもビルド結果のファイルを返してくれますが、実際にビルド

することなく取得したかったり、ビルド結果以外のファイルも欲しい場合に便利です。

Restore ターゲットの注意点

Restore ターゲットは NuGet パッケージの復元を実行します。

ビルド前にパッケージの復元をしたいというのはよくある要求で、それではということでビルドターゲットを指定する時に Restore;Build としたくなりますが、これは動かない場合があるのでやらないでください。

NuGetパッケージの中には .propsや .targetファイルが入っていることがあり、MSBuildはパッケージのリストア時にはこれらを読み込むようなプロジェクトファイルを再生成します。ビルド時には自動的に再生成されたプロジェクトファイルを読み込んで使ってくれます。

そこで Restore;Build と連続して実行してしまうと、Restore の実行後、再生成されたプロジェクトファイルを読み直すタイミングがないまま Build が実行されてしまい、NuGet パッケージ内にあった .props や .target ファイルが読み込まれずにビルドされてしまいます。これのおそろしいところは、たまたま NuGet パッケージ内 .props や .target が入ってない場合には問題なく動いてしまうこと、同じことを 2 回やると既に再生成されたプロジェクトがあって読み込まれるので 2 回目以降は正しく動いてしまうのに最初の 1 回は必ず失敗してしまうことです。

ちなみに msbuild /r /t:Build とした場合には、まず Restore 実行をして、再生成されたプロジェクトファイルを読み直してから Build を実行するので安心です。

コマンドラインから実行する場合には msbuild /r /t:Build とするなり、msbuild /restore と msbuild /t:Build を個別に実行するなりすればいいので簡単なのですが、MSBuild タスクを使う場合が少し面倒になります。

たとえば、自分で書いたプロジェクトファイルから C# のプロジェクトをビルドしようとしたときに、次のような書き方をしたくなります。

```
<Target Name="Build">
  <MSBuild Projects="hoge.csproj" Targets="Restore;Build">
</Target>
```

これは上で書いたように思ったように動かない場合があるのでやってはいけません。

MSBuild タスクに /restore オプションに相当する機能を指定したいところですが、残念ながらありません。じゃあ 2 つに分けるかとやりたくなりますが、これも上手く動きません。

```
<Target Name="Build">
  <MSBuild Projects="hoge.csproj" Targets="Restore">
  <MSBuild Projects="hoge.csproj" Targets="Build">
</Target>
```

これじゃだめです。何故かというと、最初の Restore を実行した時点で読み込んだ hoge.csproj を、Build 実行時にも読み直しせずにそのまま使い回します。そのため、msbuild コマン

ドの実行全部が一旦終わるまで、再生成されたプロジェクトファイルが使われないのです。

この問題の正しい解決方法は以下のようになります。

```
<Target Name="Restore">
  <MSBuild Projects="hoge.csproj" Targets="Restore">
</Target>
<Target Name="Build">
  <MSBuild Projects="hoge.csproj" Targets="Build">
</Target>
```

`Restore` と `Build` を分けたうえで、`msbuild /r /t:Build` を実行してください。シンプルに分けるのが正攻法です。

6. MSBuild に更なるパワー - タスクの追加

MSBuild は既定のタスクや外部コマンド呼び出しで十分いろいろできるのですが、いろんな環境で動くようにと考えると外部コマンドの呼び出しはけっこう面倒です。

どうせ .NET Framework なり .NET Core で動いてるので、プログラムを直接実行できたらいいですよね。それを実現するのにカスタムのタスクを作ることができます。

6.1. 新しいタスクを追加したい - アセンブリのロード

タスクを作りたいと思うことはありますが、いきなり作り始める前にちょっと待って！世の中には他の人が作った便利なタスクライブラリがあったりしますので、できればあるものを使いましょう。

MSBuild Extension Pack[14] というライブラリがあります。いくらか便利なタスクがあるのでちょっと使ってみましょう。残念なことに .NET Core では動かないと思うので、mono か .NET Framework の msbuild で使います。たぶん Windows でしか動かないタスクもいくらもありますが、いくつかは Windows 以外でも動くでしょう。

zip をダウンロードするといくらか DLL が入っていますのでプロジェクトファイルと同じ場所にコピーして使います。インストーラも付いているので、インストールして付属の `.targets` ファイルを読み込んでもいいのですが、それでは例にならんのでコピーして使ってください。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask AssemblyFile="MSBuild.ExtensionPack.dll" TaskName="MSBuild.
ExtensionPack.Compression.Zip"/>
  <ItemGroup>
    <InputFiles Include="*.md"/>
  </ItemGroup>
  <Target Name="Archive">
    <MSBuild.ExtensionPack.Compression.Zip
      TaskAction="Create"
      RemoveRoot="$(MSBuildProjectDirectory)"
      CompressFiles="@(InputFiles)"
      ZipFileName="sources.zip"/>
  </Target>
</Project>
```

こんな感じで使います。ここでは `MSBuild.ExtensionPack.dll` 内にある `MSBuild.`

14 http://www.msbuildextensionpack.com/

`ExtensionPack.Compression.Zip` というタスクを使っています。名前の通り zip ファイルを作ってくれるタスクです (展開もできるみたい)。ここでは Markdown のファイル (`*.md`) をまとめて `sources.zip` に入れてるだけです。

まあ実際やってることはともかくとして、アセンブリ内のタスクを使う方法を見てみましょう。といっても簡単で `UsingTask` 要素を作って `AssemblyFile` 属性に DLL のパス、`TaskName` 属性に DLL 内で定義されているタスク (のうちで使いたいもの) を指定します。

どの DLL でどのタスクが使えるや、使えるパラメータなんかは特に調べる方法はないので、そのアセンブリのドキュメントを見てください。

`AssemblyFile` に指定するアセンブリのパスはどこでもいいので、今回の例のようにプロジェクトファイルと同じ場所に置いといて読み込むのも可能です。外部のライブラリでもインストールなど不要で DLL さえコピっておけば実行できるのは便利ですね。もちろん .NET Core や .NET Framework のバージョンなど実行環境に合った DLL を用意する必要はありますが、ネイティブの実行ファイルやプラグイン的なものよりはポータブルに動かしやすいでしょう。また、どこからか拾ってきたアセンブリ込みのプロジェクトファイルをいきなり実行してしまうのはセキュリティ的に考えものですが、そこは実行ファイルだろうがなんだろうが同じなので気をつけてください。

6.2. C# のコードを実行したい - インラインタスクの作成

.NET のアセンブリを読み込んでタスクを使えるのが分かったら作りたくなるのが人情です。え、そんなの作らなくても良くない？というものでも、何かと理由をつけてそりゃもう作りたくなります。

ただいきなり DLL を作り始めるのは面倒ですし、そこまででかいタスクを作るつもりはないんだけどなぁという人のために C#(や VB.NET) のコードをいきなりプロジェクトファイルの中に書いてカスタムのタスクを作るのができるようになっています。いやあ楽しくなってきましたねえ。

```
<UsingTask TaskName="Hello" AssemblyName="Microsoft.Build.Tasks.v4.0"
TaskFactory="CodeTaskFactory">
  <ParameterGroup>
    <Name ParameterType="System.String" Required="false" Output="false"/>
  </ParameterGroup>
  <Task>
    <Code Type="Fragment" Language="cs">
    <![CDATA[
    Log.LogMessage("Hello {0}!", Name ?? "World");
    ]]>
    </Code>
  </Task>
</UsingTask>
<Target Name="Hello">
  <Hello Name="$(MyName)"/>
</Target>
```

こんな感じに使います。既存のタスクを使うのと同様に UsingTask 要素で使うタスクを定義しますが、新しく TaskFactory 属性を指定しています。また、AssemblyFile 属性でなく AssemblyName 属性を指定しています。

AssemblyName 属性は AssemblyFile 属性とだいたい同じですが、アセンブリの場所が言わなくても分かる場合にファイル名を指定せずに名前で指定できるものです。Microsoft.Build.Tasks.v4.0 というアセンブリはどこにあるか分かりませんが、ラッキーなことに名前指定で読める場所にあるようです。

TaskFactory 属性はアセンブリからタスクそのものを読み込むのでなく、タスクを作るクラスを読み込むのに指定します。ここでは Microsoft.Build.Tasks.v4.0 というアセンブリから CodeTaskFactory というタスクを作るクラスを読み込んでいます。新しく作られるタスク名は TaskName 属性で指定しています。ここでは Hello タスクを作っています。

TaskFactory を使う場合はタスクを作るための情報を指定しないといけないので、UsingTask 要素の中身で指定します。

ParameterGroup 要素の中にタスクのパラメータを定義します。要素名は作りたいパラメータの名前にし、ParameterType 属性に型名、Required 属性に必須かどうか、Output 属性に出力用の属性かどうかを指定します。ここでは文字列型の Name パラメータを入力用に作っています。

Task 要素の中はタスクファクトリ毎に異なりますが、ここでは CodeTaskFactory 以外紹介するつもりはないのでそのつもりで説明していきます。

Task 要素の中で Code 要素を作って早速中に C# のコードを書いていますね。Type 属性にはどこからどこまでを自分で書くか指定するのですが、Fragment を指定しておけばだいたい OK です。Class や Method を指定すると。Language 属性には C# を使いたいので cs を指定していますが、VB.NET を使いたいなら vb も指定できるようです。あと JavaScript(JScript ？) の js や C++(C++/CLI ？) の c++ とかも指定できるようですけど何が起きるかわかりませんね！

Code 要素の中に実際のコードを書きます。Type 属性が Fragment の場合はいきなり実行するところだけ書けば OK です。スクリプト言語ぽくて楽ですね。実際には Microsoft.Build.Utilities.Task クラスを継承したクラスの Execute メソッドの中身を書くようになっているようです。パラメータはパラメータの名前そのままのプロパティになっているので、その名前で読み書きできます。

あと Log というのが出てきますが Microsoft.Build.Utilities.Task クラスの Log プロパティで、なにやらログを書き出せるオブジェクトが入っています。詳しくはドキュメントを調べてください。Message タスクで出力したようなログが書き出せます。System.Console.WriteLine() なんかを使えないこともないですが、実行元が Visual Studio だったりすると System.Console が使えるか怪しいので Log を使うのが良いでしょう。

例のコードは単純なもので Name パラメータに名前が入っていれば Hello 名前 !、入ってなければ Hello World! をログに出すようにしているだけです。

呼び出す側はいつも通りな感じで呼び出せます。例では作った Hello タスクの Name パラメータに MyName というプロパティを渡しています。MyName プロパティはどこでも定義してませんので、環境変数やコマンドラインから指定してあげないと空になり、Hello World! が表示されることでしょう。

Hello World! だけではちょっとなんなのでもうちょっとだけ例を挙げておきます。

```
<UsingTask TaskName="LogItems" TaskFactory="CodeTaskFactory"
AssemblyName="Microsoft.Build.Tasks.v4.0" >
  <ParameterGroup>
    <Items ParameterType="Microsoft.Build.Framework.ITaskItem[]" Required="true"
Output="false"/>
  </ParameterGroup>
  <Task>
    <Reference Include="System.Linq" />
    <Using Namespace="System.Linq" />
    <Code Type="Fragment" Language="cs">
    <![CDATA[
    foreach (var item in Items.Select(i => i.ItemSpec)) {
      Log.LogMessage(item);
    }
    ]]>
    </Code>
  </Task>
</UsingTask>
<ItemGroup>
  <Files Include="*.md"/>
</ItemGroup>
<Target Name="LogFiles">
  <LogItems Items="@(Files)"/>
</Target>
```

LogItems というタスクを定義して使っています。まあ結局ログに出力するだけなんで大して変わってないし実用的でもないんですけど、実用的な例がすぐに思い付かなかったので……。

パラメータの型に Microsoft.Build.Framework.ITaskItem の配列を受け取っています。これはアイテムの実際の型で、アイテムを受け取ってメタデータを取得したりもできます。メタデータを受け取る必要が無ければ System.String[] でもいいんですけどね。いずれにせよ配列にしておくと自分であらためてセミコロンで分割とかやらなくて済みます。

あと Task 要素の中に Reference 要素と Using 要素が追加されています。Reference 要素ではコンパイル時に参照するアセンブリを指定できます。Linq を使いたかったので System.Linq を参照しています。Using 要素ではコードの前で using されてて欲しい名前空間を指定します。やっぱり System.Linq を指定しています。おかげでコードの中で Linq が使えています (Item.Select(〜) のところ)。

あんまり短くて実用的な例は示せませんでしたが、ファイルの読み書きなんかも当然できるのであとはやる気があればなんでもできます。とはいえそれなりに手間はかかるので、もしかしたら外部コマンドでスクリプト実行した方が早いかもしれません。あまり無理せず早くて必要十分な方法を選択しましょう。

さらに、ここで好き勝手出来て楽しいなあとお思いの皆様に残念なお報せです。この CodeTaskFactory というのは .NET Core では使えません。.NET Framework で使ってた動的なコンパイル方法 (CodeDOM てやつ) が .NET Core では使えなくなったせいのようです。Roslyn とか使っていつか復活することに期待しましょう。まあなんなら自分でそういうタスクファクトリを作るのはできるようなので頑張っても良いでしょう。

6.3. 自分でタスクを作りたい - カスタムアセンブリを作る

最後になりますが、自分でタスクを含むアセンブリを作る方法も紹介しておきます。やってみると意外にもめっちゃ簡単です。

まずは Visual Studio や dotnet コマンドで普通にクラスライブラリのプロジェクトを作りましょう。言語は .NET でクラスが作れればなんでもいいんですが、ここでは C# にしときます。Visual Studio 2017 では .NET Standard のクラスライブラリが作れるので、それにしておくと .NET Core の MSBuild でも使えて便利です。

次に必要な参照を追加しますが、NuGet で Microsoft.Build.Utilities.Core を探して追加してください。基本的にはこれだけで大丈夫です。他に使いたいパッケージがあれば入れときましょう。

試しに作ってみたプロジェクトファイル (MyTask.csproj) は下のような内容になりました。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Build.Utilities.Core"
Version="15.5.180" />
  </ItemGroup>
</Project>
```

.NET Standard は 2.0 になっていますが、1.3 以降ならいくつでも大丈夫そうです。

あとはコードを書きます。

```
using Microsoft.Build.Framework;

namespace MyTask
{
  public class Hello : Microsoft.Build.Utilities.Task
  {
    public string Name { get; set; }
    public override bool Execute()
    {
      Log.LogMessage("Hello {0}!", Name ?? "World");
      return true;
    }
  }
}
```

これだけです。どこかで見たような処理をしていますが、これで Hello タスクを定義しています。パラメータとして Name を文字列で受け取って、実行するとログに Hello 名前！と表示します。

目立った点としては Microsoft.Build.Utilities.Task を継承しているくらいですね。実際は Microsoft.Build.Framework.ITask インターフェースを実装したクラスを作れば良いんですが、Microsoft.Build.Utilities.Task を継承するといろいろなメンバの実装を省略して Execute メソッドだけ実装すればいいだけになるので便利です。是非これを使いましょう。

プロパティはそのままタスクのパラメータになります。ここでは string(System.String) 型で受け取っていますが、ITaskItem[](Microsoft.Build.Framework.ITaskItem[]) のプロパティを作ればアイテムを配列で受け取ることもできます。他にも文字列から変換できれば int(System.Int32)、float(System.Single)、bool(System.Boolean) でもかまいません。勝手に変換してくれます。

普通のプロパティは省略可能なパラメータになりますが、必須のパラメータを作るにはプロパティに [Required](Microsoft.Build.Framework.RequiredAttribute) 属性をくっつけます。プロパティに [Output](Microsoft.Build.Framework.OutputAttribute) 属性をつけると出力パラメータが作れます。

Execute メソッドには実行するタスクの中身を書きます。渡されたパラメータは既にプロパティに入っていますので、Execute 内ではプロパティを参照するだけです。出力パラメータがある場合は、Execute 内でプロパティに値を設定しておいてください。

例で使ってる Log は Microsoft.Build.Utilities.Task クラスのプロパティで、ログ出力用のオブジェクトが入っています。Message タスクで出力されるようなメッセージを出力できます。

Execute メソッドの戻り値は成功したら true、失敗したら false を返します。

実に単純ですね。これだけです。あとは Execute の中身を埋めてまともな動作をするように頑張りましょう。

こいつのビルドはたぶん素直に通ると思うので、できた DLL をコピってきて読み込ませて使ってみましょう。

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask TaskName="Hello" AssemblyFile="MyTask.dll"/>
  <Target Name="Hello">
    <Hello Name="$(MyName)"/>
  </Target>
</Project>
```

使い方は前に紹介したように UsingTask 要素の AssemblyFile 属性で DLL のパスを指定するだけです。読み込むタスク名はクラス名だけを指定すれば大丈夫です。一つのアセンブリ内で同じ名前のクラスが複数無ければ、名前空間まで指定しなくても探して読み込んでくれるようです。何を思ったか同じ名前のタスクを複数作ってしまった場合は MyTask.Hello のように完全限定名で使ってください。

例はインラインタスクの例と同じですが、新しく作った Hello タスクの Name パラメータに MyName というプロパティ[15]を渡しています。MyName プロパティはどこでも定義してませんので、環境変数やコマンドラインから指定してあげないと空になり、Hello World! が表示されることでしょう。msbuild /p:MyName=Hoge などとして起動してやると Hello Hoge! が表示れます。

6.3.1. 以前の MSBuild で使えるタスクを作りたい

Visual Studio 2017 や .NET Core SDK で使える MSBuild 15 用のタスクを作りましたが、これは以前のバージョンの MSBuild では読み込めません。NuGet で追加できる MSBuild 用のパッケージが MSBuild 15 用の物だからですね。

なんらかの理由で古いバージョンの Visual Studio や MSBuild を使う場合には、普通に .NET

15 Microsoft はプロパティという語が好き過ぎるようで、C# のクラスのプロパティと MSBuild プロジェクトのプロパティで名前が被ってるのでややこしいですね。ここでは MSBuild プロジェクトのプロパティを指しています。

Framework のクラスライブラリを作り、NuGet からではなくシステムにある MSBuild のアセンブリを参照に追加してください。

`Microsoft.Build.Framework` と `Microsoft.Build.Utilities.v4.0` の参照を追加すれば大丈夫です。タスクのコード自体は上で書いたものと全く同じで動きます。

これで作った場合も .NET Framework の MSBuild 15 では読み込めるようですが、残念ながら .NET Core の MSBuild では読み込めなくなってしまいます。

どうしても古いバージョンの MSBuild を使う必要が無ければ、新しく作る場合には .NET Standard で作っておいた方が汎用性があって良いでしょう。

7. 最後に

ひととおり MSBuild の機能を説明したつもりなので、この本はこれでおしまいです。

後半はちょっと駆け足になってしまったので難しかったかもしれませんが、もう気力とページが尽きたよ……。動かしながら読んでいくと理解しやすいかと思いますので、是非試してみてください。

増刷にあたっていい機会だったので、第二版としてちょびっとだけ加筆・修正してあります。ソリューションファイルのあたり、`Exec` タスクのパラメータのあたり、`RecursiveDir` メタデータとそれを使ったツリー構造を維持したファイルコピー、NuGet パッケージのリストアのあたりです。特に `Exec` タスクの `EnvironmentVariables` パラメータやツリー構造を維持したファイルコピーなんかは、普段使いに必要な機能なわりには初版では抜けていました。

これを書いている 2017 年から 2018 年あたりは、ちょうど .NET Core が本格始動してきたあたりで、MSBuild のプロジェクトの書き方も変わったり変わらなかったり、かといって従来のフォーマットや従来のバージョンもまだまだ使われているだろうというところで、こまごまと説明するところが増えてしまいました。

これが MSBuild 15 の .NET Core SDK の形式だけ説明となれば少し量は減らせたんですが、古いツールを使わざるを得ない状況もまだ沢山あるでしょうしね。

最後に参考文献と関連するツール (の中で自分が興味あるものだけ) を挙げときます。

7.1. 参考文献

7.1.1. MSBuild のドキュメント

MS のサイトにドキュメントがあります。この本より当然ながら詳しいので (特にリファレンスは)、何か困ったらめんどくさがらずドキュメントを読みましょう。

- **日本語**
  https://docs.microsoft.com/ja-jp/visualstudio/msbuild/msbuild
- **英語**
  https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild
  日本語もありますがたまにおかしいので困ったら英語の方を見ましょう。

7.1.2. MSBuild の GitHub プロジェクトページ

いまどきは MSBuild もオープンソースになってて助かりますね。

- **GitHub のプロジェクトページ**
  https://github.com/Microsoft/msbuild
  ドキュメントに書いてないこともよくあるので、いざとなったらソースを読みます。嬉しいことに思ったより複雑じゃないので読みやすいです。

何か問題があったら Issue を検索する、Issue を報告してみる、最終的には改造してパッチを送りつけるとかなると良いですね！

7.2. ツール類

7.2.1. Albacore

MSBuild 自体にはあまり関係ないんですが、Ruby でよく使われるビルドシステムに Rake というのがあります。そこから MSBuild を呼び出しやすくする Albacore というライブラリがあります。

- **Albacore**
  https://github.com/Albacore/albacore
  こんな本書いといてなんですが、MSBuild でなんでもかんでもやるのはぶっちゃけ面倒なんですよね。他に Ruby とか使えるなら使った方が楽できることも多いでしょう。

MSBuild の中から Ruby を呼び出すなら ruby コマンドの実行をしますが、逆に Rake で全体の処理を行って、Rake の中から MSBuild のプロジェクトをビルドしたい時に便利なライブラリです。

7.2.2. Fake

F# 版 Rake みたいなので Fake というのもあります。F# のコードでプロジェクトを書けるようなものですが、実際のビルド自体は MSBuild の呼び出しで行うようです。

- **Fake**
  https://fake.build/
  もーお前 F# も .NET なんだからおとなしく MSBuild 使っておけよーと言いたくなりますが、コードでターゲットとか定義したくなるのはプログラマ根性とでも言うんでしょうか。

F# なら MSBuild が動く環境だと動かない理由があんまないので、Ruby 等の他にインストールが必要なスクリプト言語に比べて MSBuild といっしょに使いやすいとも言えるでしょう。

著者紹介

本文 :kumaryu

PeerCastStation とかいう P2P なライブ配信ソフトを作ってる人。
バッチビルドなんか Ruby スクリプト書けばいいと思うの。

表紙 : シドルフェス

https://pixiv.me/graif
裏表紙のうさぎの横にいるのは「すたちゅー」です。

ライセンス

ダウンロード版について

この本の電子版が以下の URL(または右下の QR コード) からダウンロードできます。
https://www.kumaryu.net/books/tede-msbuild.html
ID: TEDEBUILD　　パスワード : 8150008973683
いつまでもダウンロードできるとは限らないのでなるべくお早めに御ダウンロードください。

奥付

タイトル :　手で書く MSBuild
発行日 :　2019 年 4 月 14 日 第二版発行
　　2018 年 4 月 22 日 初版発行
サークル名 :　あれくま
発行者 :　kumaryu
連絡先
　Web:　https://www.kumaryu.net/
　メール :　kumaryu@kumaryu.net

2018
あれくま