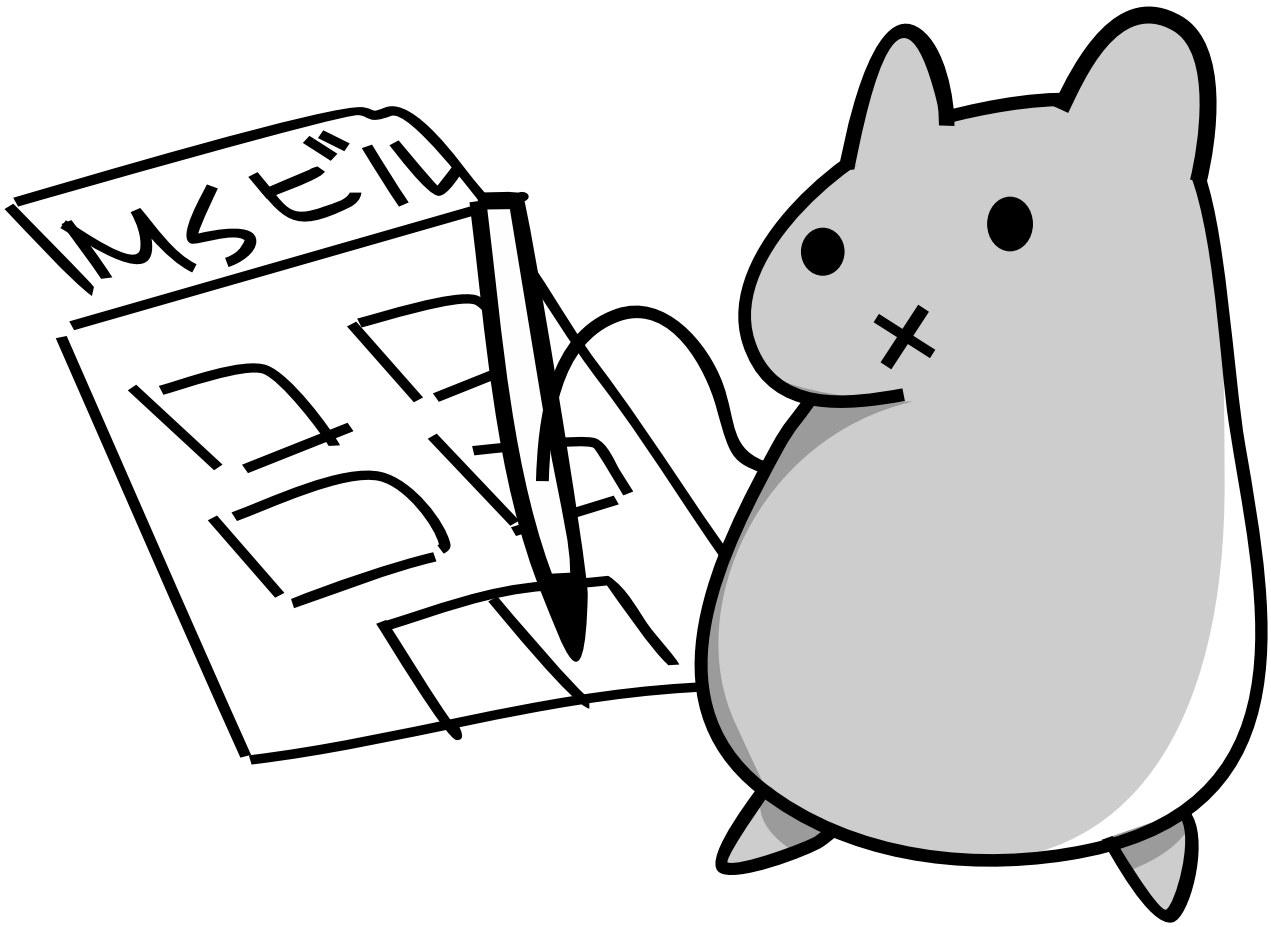
手で書く MSBuild

プレビュー版



あれくま

手で書く MSBuild

プレビュー版

あれくま

目次

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。既に存在するソリューションファイルは`dotnet sln`コマンドで操作できるようですが、新しく作るのは出来ません。

ソリューションファイルが必要な場合はおとなしくVisual Studioなどで作るか、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

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。

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。	
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。	

名前	説明	必須
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。

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。	✓
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。	

名前	説明	必須
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。

プレビュー版はここまで

さて、ここまでで最低限使えそうなところまで解説しましたが、ここでプレビュー版はおしまいです。

紹介したのは最低限の機能にだけなので、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。この先は完成版でご確認ください！

完成版では残り次のような項目が盛り込まれています。

- ファイル名をいろんな形で参照する
- 一つの指定でタスクを複数回実行する
- プロパティをいろんな形で参照する
- 特定の値の時だけ違うことをする条件実行
- 一部のファイルだけ違うことするアイテムのメタデータ
- 他のプロジェクトを流するプロジェクトのインポート
- .NET Framework や .NET Core のプロジェクトファイルの解説
- 既存のプロジェクトファイルを呼び出したり拡張したりする
- タスクの追加と自前のタスクの作成

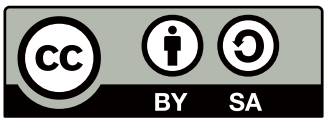
完成版を一通り読めば、MSBuild を使うのに困るところはない程度の必要十分な知識を得られることでしょう。たぶんね。

著者紹介

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。

ライセンス



奥付

タイトル : 手で書く MSBuild プレビュー版 第 2 版
発行日 : 2018 年 4 月 10 日第 2 版発行
2017 年 10 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

あれくま
2017