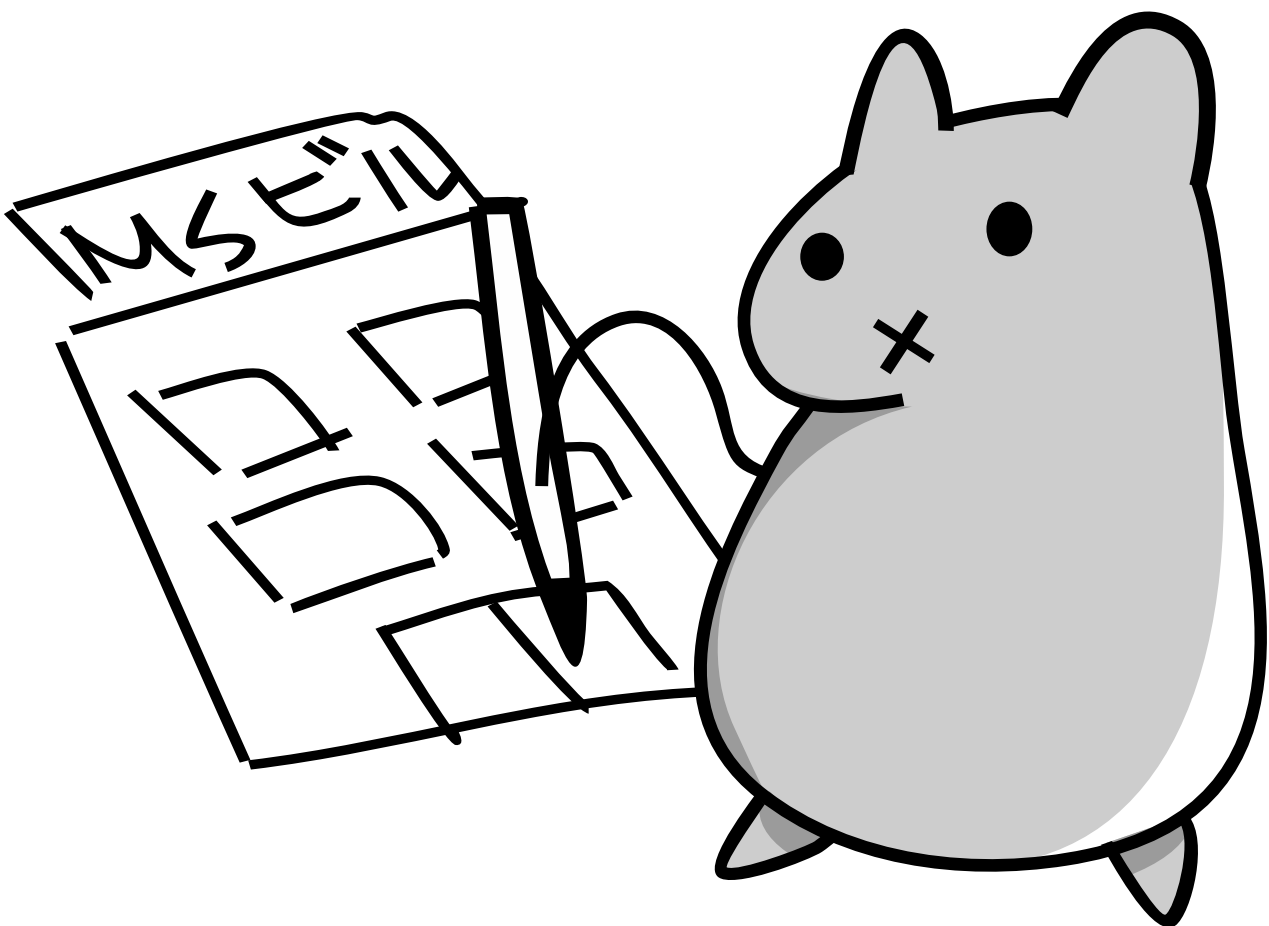



手で書く プレビュー版

MSBuild



あれくま

手で書く  プレビュー版

MSBuild

あれくま

目次

1. MSBuild とは	5
1.1. ビルドツールとは	5
1.2. MSBuild の特徴	5
2. 最初の一步	8
2.1. インストール	8
2.1.1. Windows	8
2.1.2. macOS	10
2.1.3. Linux 等	11
2.1.4. ソースからビルドする	11
2.2. プロジェクトを書いてみる	11
3. プロジェクトファイルの構成	14
3.1. 小さいプロジェクト	14
3.2. プロジェクト	16
3.3. プロパティ	16
3.4. アイテム	17
3.4.1. アイテムのワイルドカード	20
3.5. ターゲット	22
3.5.1. 必要な時だけ実行されるターゲット	23
3.5.2. ターゲットの依存関係	25
3.6. タスク	25
3.6.1. Message タスク	26
3.6.2. Warning / Error タスク	26
3.6.3. Touch タスク	27
3.6.4. Copy タスク	28
3.6.5. Move タスク	28
3.6.6. Delete タスク	29
3.6.7. MakeDir タスク	30
3.6.8. RemoveDir タスク	30
3.6.9. Exec タスク	30
3.6.10. MSBuild タスク	31
3.6.11. アイテムの参照	32
3.6.12. タスクの実行回数	35
4. プレビュー版はここまで	37

1. 1. MSBuild とは

MSBuild はマイクロソフト製のビルドツールです。ビルドツールがなにかって？あれだよ、make とか Ant とか Rake とか SCons とかそういうやつだよ。わかるだろ？

1.1. ビルドツールとは

ビルドツールとはなんでしょう。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。

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する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。

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速くなりますね。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！……とりたいところですが、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？うわめんどくせえ……。

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は差異はありますが、どのツールもやってくれることは基本的に同じです。

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++ でも使いやすくなりました。

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点、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
- C# や VB.NET で比較的簡単に拡張可能
- 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は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。

機能の拡張に関してはちょっとしたものであればプロジェクトファイル内に直接 C# や VB.NET で書くことができます。ちょっとしたもので済まなければ .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) で拡張することができます。ちゃんと作ればこれら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よりはあまり無いでしょう。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に、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. 2. 最初の一步

ここから MSBuild を実際使っていくところに入ります。

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ってきます。

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¹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 までしかないなので最新の 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¹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 のおまけ扱らしく、Visual Studio と同等のライセンスが必要になるようです。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。³.NET Core のサイトから 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

1 [https://www.visualstudio.com/ja/]

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

3 [http://www.microsoft.com/net/core]

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の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。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。
```

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 の公式サイトからダウンロードして入れることができます。⁵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 を使しましょう。

.NET Core SDK を単体で入れるには、 .NET Core のサイトから SDK をダウンロードしましょう。⁷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。
```

5 [\[https://www.visualstudio.com/ja/\]](https://www.visualstudio.com/ja/)

6 [\[http://www.mono-project.com/\]](http://www.mono-project.com/)

7 [\[http://www.microsoft.com/net/core\]](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⁸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。

.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 のサイトにインストール方法がありますのでこれに入れましょう。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難しいので諦めて mono を使った方が良いでしょう。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¹⁰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. プロジェクトを書いてみる

インストールはできたと思う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 を解説してい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) 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` でかまいません。

8 [\[http://www.mono-project.com/\]](http://www.mono-project.com/)

9 [\[http://www.microsoft.com/net/core\]](http://www.microsoft.com/net/core)

10 [\[https://github.com/Microsoft/msbuild\]](https://github.com/Microsoft/msbuild)

```
<?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。

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。

```
C:\Users\kumaryu\Documents\hello>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 "C:\Users\kumaryu\Documents\hello\hello.proj" (既定のターゲット)。

Build:

Hello

プロジェクト "C:\Users\kumaryu\Documents\hello\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！

……いやいや、つまんねーって？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** タグですが、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いますが、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。

```
<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步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す。

3. 3. プロジェクトファイルの構成

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

まずは一通りの物が入った小さいの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(入力ファイル)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、¹¹pandoc というコマンドで、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

¹¹ 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。

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。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 を実行すると以下ようになります。

```
C:\Users\kumaryu\Documents\minimal>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 "C:\Users\kumaryu\Documents\minimal\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 "C:\Users\kumaryu\Documents\minimal\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
```

Clean を実行すると以下ようになります。

```
C:\Users\kumaryu\Documents\minimal>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 "C:\Users\kumaryu\Documents\minimal\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 "C:\Users\kumaryu\Documents\minimal\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

デフォルトじゃないターゲットを実行する場合には `msbuild /t:Clean` のように `/t:` オプ
ションで指定します。

3.2. プロジェクト

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。

また、Project 要素にある `xmlns="～"` の部分は 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。

`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、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。

`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、複数ターゲットがある場合は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。

3.3. プロパティ

次はプロパティの定義です。

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

念のため書いておくと `<!-- -->` は XML のコメントですよ。MSBuild では無視されますので分かりやすいように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。

また、MSBuild に `/p:OutputFormat=html5` といった形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。

`PropertyGroup` 要素の中のタグ名はプロパティ名になるので、特にこの要素で書かないといけないというのは無いです。好きに付けてかまいません。もちろん XML の要素名にならないような文字は入らないですが、日本語とかで入れても大丈夫です。

```
<PropertyGroup>
  <ほげ>hoge</ほげ>
  <ふが>fuga</ふが>
</PropertyGroup>
```

こんなんでもちゃんと通ります。まあほげとかふがとかいうプロパティあっても意味わからんので付けないと思いますが、ちゃんと伝わるプロパティ名を日本語で付けても大丈夫です。日本語にすると入力が面倒になりますけどね。

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。スペースとか入れてもいいですし、他の XML のタグとか入れてもかまいませんが、使う時は単に文字列としてそのまま扱われます。

PropertyGroup 要素の位置は上の例だと **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のが便利でしょう。ただし、**Target** 要素の中とかには入れられません。**Project** 要素の直下のみです。

3.4. アイテム

次はアイテムの定義です。

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

ItemGroup 要素の定義でアイテムを定義します。アイテムもまだ説明されてないですけど、まあコメントに書いてある通り入力ファイルを列挙するものです。後の方で **@(Inputs)** とか **%(Inputs.FileName)** とかで参照されています。

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ります。例で **Inputs** とかそれらしい要素名になっていますが、入力なんかにしちゃっても問題ありません。プロパティと違うところは見た目を挙げると、タグのテキストでなく **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。

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 (ファイル名の配列) です。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** 要素を参照すると **hoge.md** と **fuga.md** の両方が取得できます。

ちなみに、プロパティも 2 回以上定義を書けますが、プロパティは後から書いた方で上書きになります。

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。

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>
```

これで例と同じ意味になります。セミicolon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icolon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icolonで区切ります。

他に **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icolon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。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リストで参照され、文字列として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。この例で @(InputsMarkdown) は hoge.md; fuga.md などに、@(InputsText) は foo.txt; bar.txt; baz.txt といった形に展開され、最終的に Inputs アイテムは hoge.md; fuga.md; foo.txt; bar.txt; baz.txt 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。

3.4.1. アイテムのワイルドカード

アイテムのワイルドカードは * だけさ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。
?	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 1. 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" />
    <Outputs Include="*.docx" />
  </ItemGroup>
  <Target Name="Build">
    <Message Text="入力ファイル: @(Inputs)" />
    <Exec Command="pandoc -t docx -o %(Inputs.FileName).docx @(Inputs)" />
    <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 *.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ため、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

いるので、docx ファイルは一つも見つけれられません。

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 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るので気をつけましょう。

上のよう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、これで見ても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

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>
```

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。ここも別に日本語でもいいですが、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めんどくさそうです。

中身には処理を書きます。処理は上から順番に実行されます。それだけ。

中に行ける処理はあとで説明するタスク以外に、アイテムやプロパティの定義もできます。

```
<Target Name="Build">
  <PropertyGroup>
    <OutputFormat>md</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。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、出力されたファイルを列挙したい場合には **Target** の中でプロパティを定義するのが使えます。

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、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. 必要な時だけ実行されるターゲット

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。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一瞬で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て欲しいですね。

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>
```

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。

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

2017/10/04 0:11:21 にビルドを開始しました。

ノード 1 上のプロジェクト "C:\Users\kumaryu\Documents\target\target.proj" (Build ターゲット)。

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 "C:\Users\kumaryu\Documents\target\target.proj" (Build 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 00:00:00.06

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。

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省略できないので意味ありませんし、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>

```

こんな感じで出力側のアイテムを作ります。@(\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。

3.5.2. 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つちのターゲットを実行しとかな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、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のから順番に実行されます。

ターゲットの依存関係は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

ターゲットの中で実行されるタスクを見ていきましょう。

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

この `Message` や `Exec` 要素がタスクです。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。じゃあタスクの定義はどこで行なわれるかというと、別なアセンブリなどで定義したやつを読み込んで使います。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別途読み込まなくても使えます。

タスクに渡すパラメーター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るので、詳しくはそれぞれのタスクのドキュメントを見てください。

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。

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ーは以下の通りです。

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

表 2. Message タスクのパラメーター

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 3. 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 を指定した時也表示されます。

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まれます。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 4. 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ができます。

```
<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。	

表 5. 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。	※

名前	説明	必須
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。	

表 6. 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も勝手に作ってくれるのは便利ですね。

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。

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。	※
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。	

表 7. 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 <code>true</code> か <code>false</code> で指定します。省略すると <code>false</code> です。	

表 8. Delete タスクのパラメータ

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。	✓

表 9. MakeDir 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。	✓

表 10. 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。たぶん一番良く使うでしょう。

```
<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。	

表 11. 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。

Exec タスクでは、実行したコマンドの標準出力を受け取るようなことはできません。リダイレクトでファイルに書き出すのはできますので、それでなんとかするのが良いでしょう。

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="Configuration=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。	
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。	

表 12. 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 `/p: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。

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、いろいろと使い出があるタスクです。

3.6.11. 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 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

```
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。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、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まれます。	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まれません。	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 13. 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まれます。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)')
```

» 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括弧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用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 アイテムの括弧子を .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したが、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。

3.6.12. タスクの実行回数

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。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. 4. 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。これ以上は時間ももう無いですし、なによりページ数がやべえです。

MSBuild の機能はまだまだありますし、あまり使わない機能はともかく、条件実行など良く使う機能もまだなので紹介したいところですが、それらは今後出す予定の完成版にまわすようにしましょう。

完成版では次のような項目を紹介したいと思っています。ただし予定なので書いてる間に増減したりするでしょう。

- 条件実行
- アイテムのメタデータ
- タスクからの出力
- .NET Framework のメソッド呼び出し
- ターゲットの割り込み
- Visual Studio 用のターゲットファイルを使う
- .NET Core 用の SDK を使う
- C# のコードを実行する
- カスタムアセンブリを追加する
- カスタムアセンブリを作る
- Visual C# のプロジェクトファイルを呼び出す
- Visual C++ のプロジェクトファイルを呼び出す

完成版まで待てないよー、という人やもっと詳しく知りたい人は MSBuild のドキュメント¹²を読みましょう。

プレビュー版で解説したことと、ドキュメントさえ読めばだいたい不自由なく使えると思います。

完成版は早くて 2018 年春頃の頒布予定です。あまり期待せず待っていてください。

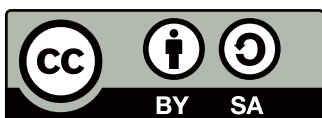
12 [https://docs.microsoft.com/ja-jp/visualstudio/msbuild/msbuild]

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。
バッチビルドとか Ruby スクリプト書けばいいじゃん。

ライセンス



この作品の本文 (表紙を除く) は、クリエイティブ・コモンズの表示 - 継承 4.0 国際 ライセンスで提供されています。ライセンスの写しをご覧になるには、<http://creativecommons.org/licenses/by-sa/4.0/> をご覧頂くか、Creative Commons, PO Box 1866, Mountain View, CA 94042, USA までお手紙をお送りください。

手で書く MSBuild プレビュー版

発行日: 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: <http://www.kumaryu.net/>

メール: kumaryu@kumaryu.net

あれくま
2017