Springhead の開発における CMake の使い方

第 1.0 版 – 2019/07/05

本ドキュメントでは、Springhead Library の開発およびそれを用いたアプリケーションプログラムの開発において、CMake をどのように使用するかについて説明をします。

改 訂 履 歴

第 1.0 版　初版

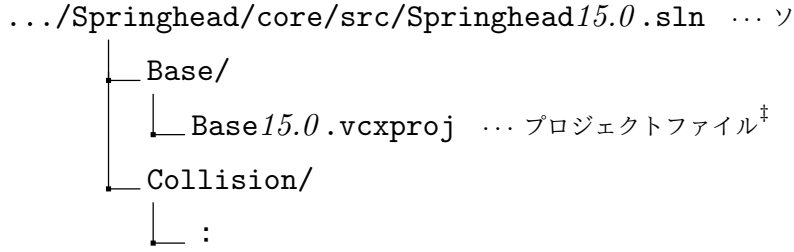
目次

1　CMake とは何をするものか

まず最初に、cmake とは何をするものかについて簡単に説明をします。

1.1　従来の方法

GitHub から Springhead をダウンロードすると、Springhead Libaray をビルドするためのソリューションファイルおよびプロジェクトファイルがその中に含まれていました。



上記のソリューションファイル†を実行すれば Springhead Library を生成することが、アプリケーションプログラム用のソリューションファイルに上記のプロジェクトファイル‡を“既存のプロジェクト” として追加すれば、アプリケーションの開発と同時に Springhead Libaray の開発が行なえました。

後者では、プロジェクトファイル‡が直接共有されることにご注意ください。このため、複数のアプリケーションのソリューションファイルを同時に実行してプロジェクトファイル‡に変更が及ぶような修正を実施しても、その変更が他のアプリケーションにも反映されました (プロジェクトが環境外で変更された旨のダイアログが出ました)。

この方法はうまく機能していますが、次の点が難点として挙げられます。

- Visual Studio のバージョンが上がる度に、ソリューションファイルとプロジェクトファイルを作り直して提供しなければならない。
- Windows 以外のプラットフォームに対しては、Makefile などを別途作成して提供しなければならない。

1.2 CMake を使用した場合

cmake はビルドの自動化を図るためのツールであり、ソースコードをコンパイルしたりデバッグの制御をしたりすることを目的としたツールではありません。Visual Studio や make ではなく configure に近いツールと考えていただいてよいでしょう。

cmake の特徴の一つに out of source (out of place) でのビルドに対応しているということがあります。これはソースツリーの外側にビルドツリーを生成する機能で、

- 複数のビルドツリーを作成することが可能である。
- ビルドツリーが削除されてもソースツリーに影響が及ばない。

という特徴があります (Wikipedia より)。ここでは cmake を out of source で使用するものとして説明します。

ソースツリーおよびビルドツリーは次のようになるでしょう。

```
.../Springhead/core/src/  ··· ソースツリー
    ├─ CMakeLists.txt
    ├─ Base/  ··· (*1)
    │   ├─ CMakeLists.txt
    │   ├─ Affine.cpp
    │   └─ :
    ├─ Collision/  ··· (*2)
    │   └─ :
    ├─ :
    └─ build  ··· ビルドツリー (他の場所でも構わない)
        ├─ Springhead.sln
        ├─ Base/
        │   ├─ Base.vcxproj
        │   ├─ Base.dir/  ··· オブジェクトファイル (.obj) が置かれる
        │   └─ CMakeFiles/
        │       └─ generate.stamp  ··· スタンプファイル
        ├─ Collision/
        └─ :
```

図 1 Springhead Library 構成

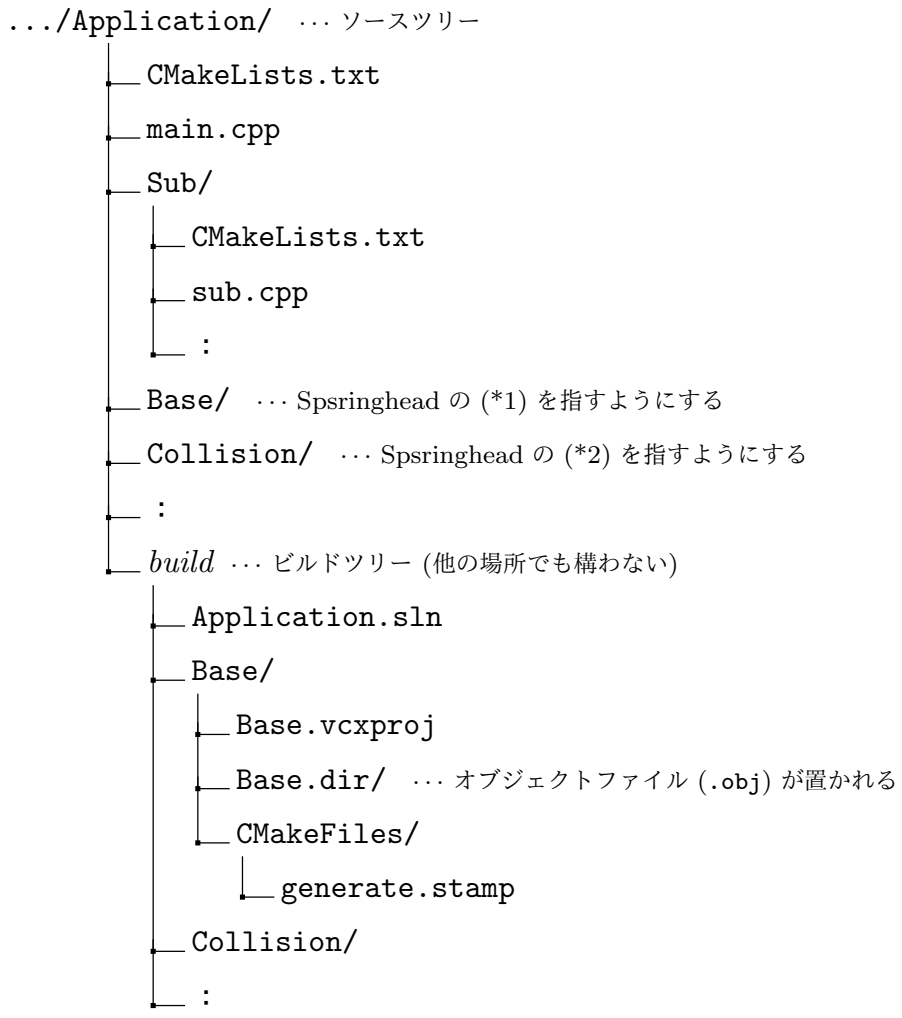


図 2 Application 構成

ここで "CMakeLists.txt" は cmake を制御するためのパラメータファイルで、メインディレクトリおよびサブディレクトリのそれぞれに作成します。

1.3 CMake を使用した場合の問題点

前節の図 2 に示したアプリケーション (App1) の他にアプリケーション (App2) があったとして、その両方を同時に開いて作業を行なう場合を考えます (App2 も App1 同様 Springhead Library のプロジェクトを利用しているものとします)。

ソースファイルについては、どちらも Springhead Library のソースツリーの (*1), (*2) などを指すようにしてありますから、どちらで実施したソースファイルの変更も直ちに他方に反映されることになります (同じファイルを参照しているのですから当然です)。

オブジェクトファイルはどうでしょうか。残念なことに App1 と App2 のビルドツリーは異なるものですから、App1 でソースを変更してビルドしたとしてもそのビルド結果がそのまま App2 に反映されることはありません (これらは異なるファイルです)。しかし App2 で再ビルドを行なえば、その時点で App1 の変更が App2 に反映されることになります。

では、プロジェクトファイルはどうでしょうか。ソースファイルの変更を行なうだけならば、プロジェクトファイルが変更されることはありません。しかし、ソースファイルの追加や削除を行なったならばプロジェクトファイルにも変更が及びます (Visual Studio 上でソースの追加/削除を行なってセーブを行なった場合、再度 `cmake` を実行した場合など)。

プロジェクトファイルもオブジェクトファイルと同様にビルドツリー内に生成されますから、App1 で実施した変更が App2 に反映されることはありません。しかも App2 で再 `cmake` したとしても App1 の変更が反映されることはありません。App2 のプロジェクトファイルは App1 のものとは異なるファイルですから App2 は依然として古い状態を残したままなのです。

オブジェクトファイルの件は、無駄なコンパイルが発生するという点に目をつぶれば許容できなくはない範疇かも知れません (Visual Studio では実行しようとすればビルドが走ります。unix の場合は実行前にかならず `make` をしないといけないですが)。これは “**ビルドの最適化**” の問題です。

しかし、プロジェクトファイルの件は許容できる範囲を超えています。例えば App1 で今まであったファイルを削除する変更を行なったとします (ソースツリーが変更される)。すると App2 でビルドをすると “ファイルがない” というエラーが発生してビルドに失敗することになります。このような場合には、App2 で再度 `cmake` を実行してプロジェクトファイルを作り直す (configure し直す) 必要が生じるのです。これは “**ビルド構成の整合性**” の問題です。