Springhead - How to use CMake

Contents

List of Figures

Chapter 1

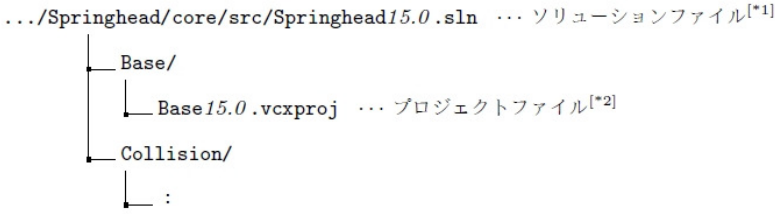
2ukqrQFbNykczxHg

1.1 QIXNuxHNIBFl3Hkv

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。 Springheadの開発に携わらない方はこの章の内容は不要です。 “2.1 Springheadをライブラリとして利用する方へ”をお読みください。 また、Cmakeについて理解をされている方は “1.1.3 CMakeを使用した場合の問題点”、 “1.1.4 対処法”および “3.1 Springheadを開発する方へ”をお読みください。

1.1.1 M6fyRuhEFB7guPkI

GitHubからSpringheadをダウンロードすると、 Springheadライブラリ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す。



上記のソリューションファイル[*1]を実行すればSpringheadライブラリを生成することができ、 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[*2]を“既存のプロジェクト”として追加すれば、 アプリケーションとSpringheadライブラリの開発を同時に行なうことができました。

後者の場合は、プロジェクトファイル[*2]は直接共有されることになります。 このため、複数のソリューションファイルを同時に開き、あるアプリケーションで プロジェクトファイル[*2]に変更が及ぶような修正(ソースファイルの追加・削除)を実施しても、 その変更は他のアプリケーションに直ちに反映されました。 (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す)

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のバージョンが変わる度に、 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す必要がある。
- Windows以外のプラットフォームに対しては、 Makefileなどを別途作成する必要がある。

1.1.2 CzE2SHJQPHzlgjQB

CMakeは、Visual Studioやmakeのように ソースコードのビルドやデバッグの制御をすることを目的としたツールではなく、 ビルドの自動化を図るためのツールです。 CMakeは、`"CMakeLists.txt"`というパラメータファイルから solution/project file (Windows Visual Studio)やMakefile (unix make)を自動

的に生成します (unixのconfigureに近いものと考えていただいてよいでしょう)。

Cmakeはout of source (out of place)によるビルドに対応しています。 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、次のような特徴があります。

- 互いに干渉しない複数のビルドツリーを作成することができる。
- 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。

我々はCMakeをout of sourceの方法で使用します。

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 (図3.1.3および図1.2)。

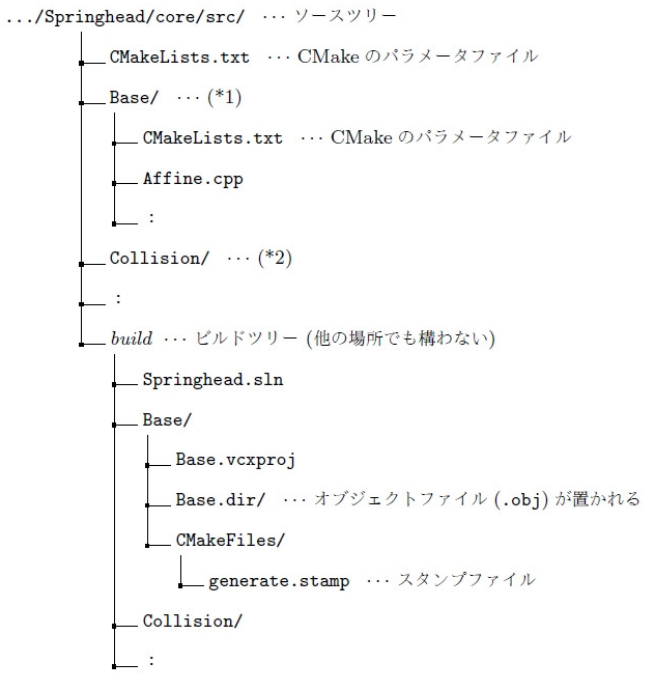


Fig. 1.1: Springhead Library構成

"CMakeLists.txt"は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。

1.1.3 sXn8UpnPwdz1mDIw

前節の図1.2に示したような 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。 ここでは*App1* と*App2* が共通して参照するSpringheadライブラリのプロジェクトについてのみ考えます。

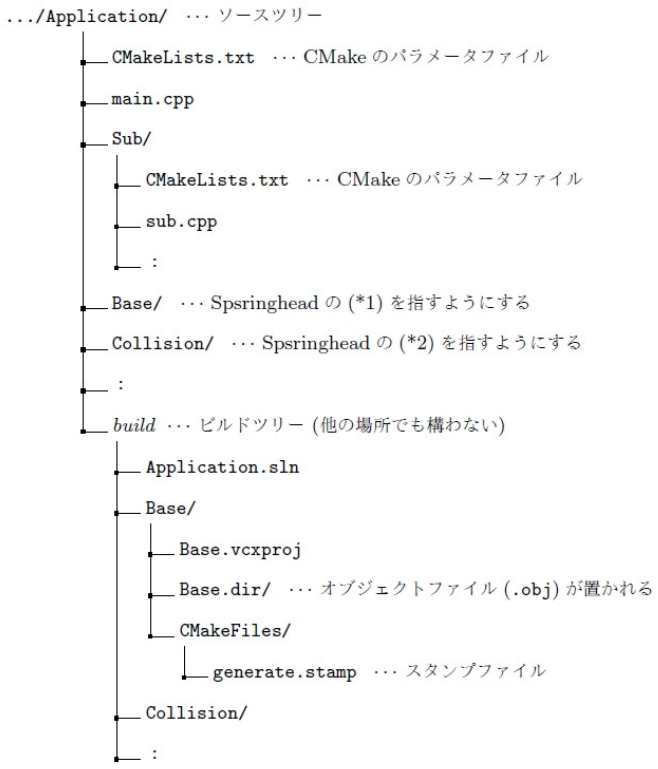


Fig. 1.2: Application構成

ソースファイルの整合性

ソースファイルの整合性には問題がありません。 *App1* と*App2* のどちらもSpringheadライブラリのソースツリー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)

ソースファイルの追加や削除を実施したとしても ソースファイルの整合性自体が崩れるわけではありません。 ただし、この場合には“**プロジェクトファイルの整合性**”の問題が発生します。

ビルドの最適性(無駄なコンパイル)

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(オブジェクトファイル)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。

App2 でソースの変更を反映させようとすれば *App2* で再ビルドを行なう必要があります (この時点で変更されている ソースファイルが再コンパイルされ、オブジェクトファイルの整合性がとれます)。 従来はオブ

ジェクトファイルも共有されていましたから、 このような無駄なコンパイルは発生しませんでした。

プロジェクトファイルの整合性

ソースファイルの内容を修正しただけならば 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、 もしくは*App1* での変更後に再cmakeを行なった場合)。

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* で再ビルドしたとしても*App1* で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。

これを解消するには*App2* で再度cmakeを実行する必要がありますが、問題は“**いつ再cmakeが必要なのかが分からない**”ということです。

1.1.4 2oir3Im2bBWI9URv

前節で示した問題点への対処法について述べます。

ソースファイルの整合性

Springheadライブラリのソースツリーにあるプロジェクトディレクトリ (`Base`, `Collision`, ...)を 直接‘`add_subdirectory`’すれば問題ありません。

ビルドの最適性(無駄なコンパイル)

Springheadライブラリ, *App1, App2* などのすべてにおいて、 ライブラリのオブジェクトが生成される場所を共通化してしまうことで この問題を回避することとします。 具体的には、Springheadライブラリソースツリーの中(ビルドツリーの外)に オブジェクトの共通格納場所を作り、Springheadライブラリおよびすべてのアプリケーションのオブジェクト格納領域が そこを指すようにlinkを張ることとします。
オブジェクトの共通格納領域を設定する作業はSpringheadライブラリの`cmake` (configure)時に、 linkを張る作業はアプリケーションの`cmake` (configure)時に行なうものとします。

プロジェクトファイルの整合性

ビルドの最適性の場合と同様、 プロジェクトファイルも共通化することでこの問題に対処します。 具体的には、すべてのアプリケーションのプロジェクトファイルは、 Springheadライブラリビルドツリーにあるプロジェクトファイルを参照するlinkとします。

ただしこれでは不完全で、 *App1* で実施したプロジェクトファイルへの変更が *App2* に伝わりません。 このため*App1* でプロジェクトファイルを変更した場合には、 その変更をSpringheadライブラリビルドツリ

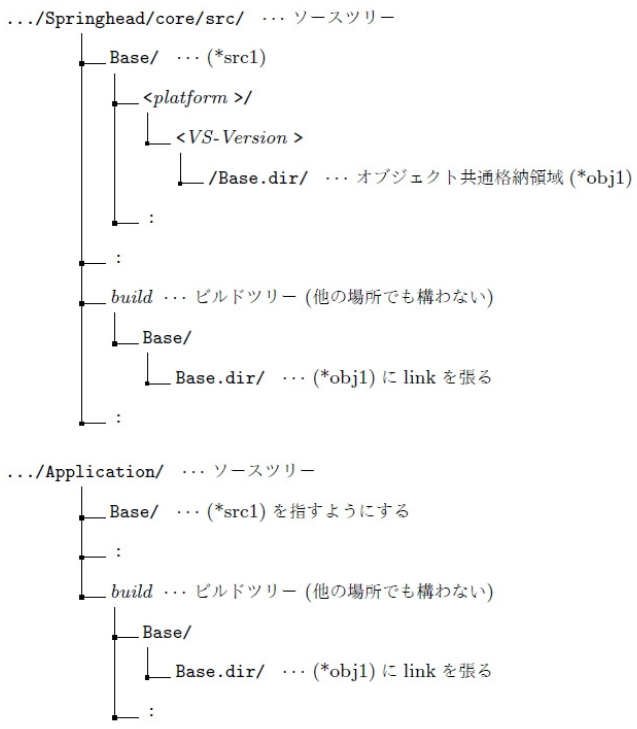


Fig. 1.3: ビルドの最適性への対処法

ーにあるプロジェクトファイルに反映させるものとします。 つまり、Springheadライブラリのビルドツリーにあるプロジェクトファイルを 常に最新の状態に保つということです。

この作業はアプリケーションのビルド時に行なうものとします。 そのために、各アプリケーションのソリューションファイルに 特別なターゲット‘sync’を作成し、 このターゲットが毎回のビルドに先立って実行されるようにします。

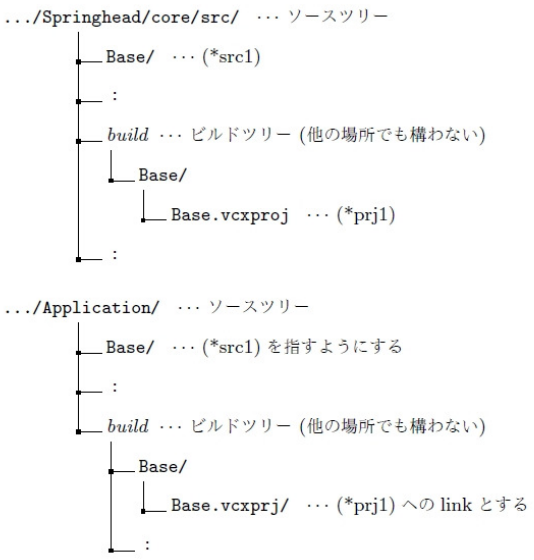


Fig. 1.4: プロジェクトファイルの最適性への対処法

Chapter 2

COtOfv3iniSnphzw

2.1 K2H8cRjPq0Vxb98X

この章は、Springheadをライブラリとして利用する方のために、 ライブラリファイルの作成方法について説明します。

2.1.1 GkrZwDollfzZLadk

まずSpringhead LibraryをGitHubからダウンロード(clone)してください。 以下、 ダウンロードするディレクトリを"C:/Springhead"として説明を進めます。

```
> chdir C:/Springhead
> git clone --recurse-submodules (次の行に続く)
                    https://github.com/sprphys/Springhead
```

サブモジュールを選択する場合は、 GUIの場合は、

```
> chdir C:/Springhead
> git clone https://github.com/sprphys/Springhead
> git submodule update --init --checkout buildtool
> git submodule update --init --checkout dependency
```

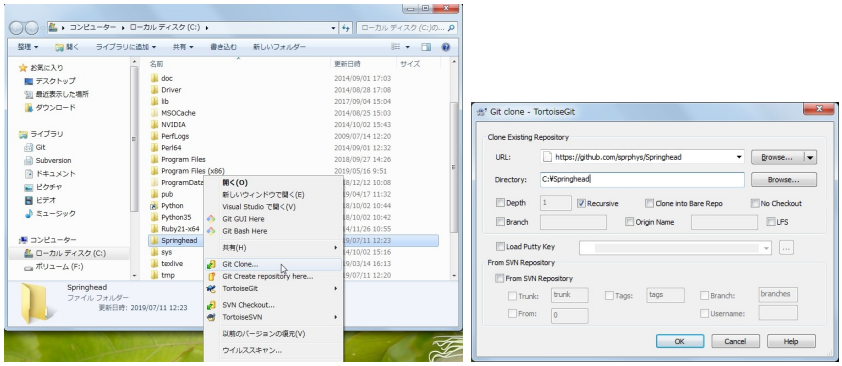


Fig. 2.1: Springheadダウンロード

2.1.2 K11cOJIYzUE65BBo

ダウンロードが済んだら"C:/Springhead/core/src"に移動してください。

まず、配布されたファイル"CMakeLists.txt.Lib.dist"を "CMakeLists.txt"という名前でコピーします (誤コミットを防止するためにもリネームではなくコピーしてください)。

```
> chdir C:/Springhead/core/src
> copy CMakeLists.txt.Lib.dist CMakeLists.txt
```

自前でインストールしているパッケージ (boost, glew, freeglut, glui)を使用する場合 およびライブラリファイルとヘッダファイルのインストール先を指定する場合には、 さらに、 配布されたファイル"CMakeConf.txt.dist"を"CMakeConf.txt"という名前でコピーして 必要な編集をします。 編集の方法については"CMakeConf.txt"に記述されています (“=ESCAPEx23=set(variable *value*)”から“=ESCAPEx23=”を削除し、 “*value*”を適切に設定し直す)。

```
> copy CMakeConf.txt.dist CMakeConf.txt
> edit CMakeConf.txt
    ⋮
#set(CMAKE_PREFIX_PATH
# "F:/Project/ExternalPackage/boost_1_70_0/include/boost-1_70"
# "F:/Project/ExternalPackage/glew-2.1.0"
# "F:/Project/ExternalPackage/glew-2.1.0/lib/Release/x64"
# "F:/Project/ExternalPackage/freeglut"
# "F:/Project/ExternalPackage/glui-2.37/include"
#)
    ↓
set(CMAKE_PREFIX_PATH
    "C:/somewhere/appropreate"
    :          (絶対パスで指定する)
    :          (パスを複数指定するときは空白、改行またはセミコロンで区切る)
)

#set(SPRINGHEAD_INCLUDE_PREFIX "C:/usr/local")
#set(SPRINGHEAD_LIBRARY_DIR_DEBUG "C:/usr/local/lib")
#set(SPRINGHEAD_LIBRARY_DIR_RELEASE "C:/usr/local/lib")
    ↓
set(SPRINGHEAD_INCLUDE_PREFIX "C:/somewhere/appropreate")
set(SPRINGHEAD_LIBRARY_DIR_DEBUG "C:/somewhere/appropreate")
set(SPRINGHEAD_LIBRARY_DIR_RELEASE "C:/somewhere/appropreate")
```

以上で準備作業は終了です。

2.1.3 dk1fTMPIllv1rQIl

以下では、cmakeの生成物(ビルドの生成物ではありません)を格納する 作業場所(ディレクトリ)を“*build*”として話を進めます(作業場所は任意で構いません)。

cmakeにはConfigureとGenerateの2段階があります。

コマンドプロンプトの場合は、1回のコマンドで両方を実行できます。

```
> chdir C:/Springhead
> mkdir build
> cmake -B build generator
```

GUIの場合は、

まずConfigureボタン(図2.2 左図の下)を押します。

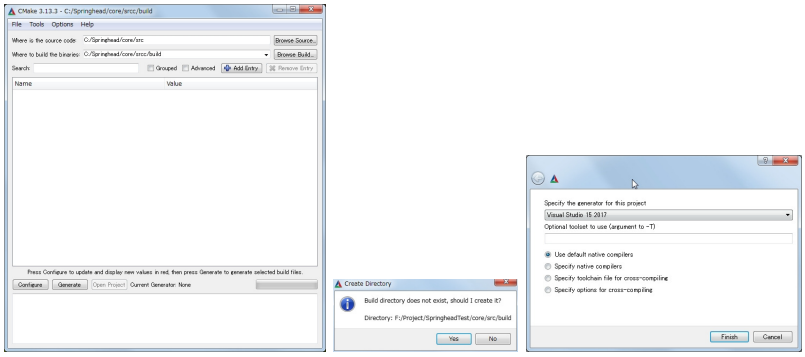


Fig. 2.2: cmake configure

build ディレクトリがなければ作成するかどうか聞かれ(同中央図)、 次にgenerator指定画面となります(同右図)。 cmake version 3.15.0では、generatorとして図2.3のものが指定できます。

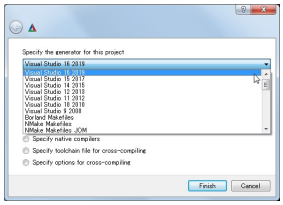


Fig. 2.3: cmake generator

最後に図2.2 左図の下のGenerateボタンを押します。

以上で、*build* 以下にsolution/project fileなどが生成されたはずです。

2.1.4 FPMSbspgcr5R4oj6

Springhead Libraryのビルド方法は、従来と変わりがありません。
build へ移動し、"Springhead.sln"を指定してVisual Studioを実行してください。

“2.1.2 準備”でライブラリファイルのインストール先を 指定していなければ、ライブラリファイルは次のいずれかに生成されます。

"C:/Springhead/generated/lib/win64"

または

"C:/Springhead/generated/lib/win32"

Chapter 3

2rgZt0T7lfFaxUHi

3.1 Icqvaj3n56jFFpbk

この章では、Springheadの開発を行なう方のために、 cmakeを使用したアプリケーションプログラムの作成方法について説明します。

アプリケーションプログラムについての作業をする前に、 あらかじめSpringehadライブラリをダウンロードしてcmakeを実行しておいてください (“2.1 Springheadをライブラリとして利用する方”参照)。

3.1.1 sByVELhUuWQYiYYM

アプリケーションと並行してSpringheadライブラリを開発する場合には、“1.1.3 CMakeを使用した場合の問題点”で示した問題に対処する必要があるため、 ここで述べる方法に従って作業を進めてください (“1.1.4 対処法”で示した方策が組み込まれます)。

以下、Springheadライブラリをダウンロードしたディレクトリを"C:/Springhead"、 アプリケーションプログラムを作成するディレクトリを"C:/Develop/Application"として説明を進めます。

"C:/Develop/Application"に移動してください。

配布されたファイル"CMakeTopdir.txt.dist"を"CMakeTopdir.txt"という名前で、 "CMakeLists.txt.Dev.dist"を"CMakeLists.txt"という名前でコピーします (誤ってコミットするのを防ぐためにも、リネームではなくコピーしてください)。

```
> chdir C:/Develop/Application
> copy C:/Springhead/core/src/CMakeTopdir.txt.dist CMakeTopdir.txt
> copy C:/Springhead/core/src/CMakeLists.txt.Dev.dist CMakeLists.txt
```

"CMakeTopdir.txt"の編集

Springheadライブラリをダウンロードしたディレクトリを"CMakeTopdir.txt"にはSpringheadライブラリに設定します。 これは、CMakeにSpringheadのソースツリーの場所を教えるために必要な設定です。

```
#set(TOPDIR "C:/Springhead")
  ↓
set(TOPDIR "C:/Springhead")
```

"CMakeLists.txt"の編集

1. プロジェクト名を設定します(11行目)。

```
#set(TOPDIR "C:/Springhead")
  ↓
set(TOPDIR "C:/Springhead")
```

2. Customization section (19行目以降)を必要に応じて変更します。
   各変数の意味は次のとおりです。

OOS_BLD_DIR	CMakeの作業領域(ディレクトリ)の名前(本ドキュメントで*build* としているもの)。
CMAKE_CONFIGURATION_TYPES	ビルド構成。
SRCS	ビルドの対象とするファイル。 設定はset(SRCS …)またはfile(GLOB SRCS …)とします。 後者ではワイルドカードが使えます。 SRCSの直後に“RELATIVE <*base-dir*>”を付加すると相対パス指定となります。デフォルトは file(GLOB RELATIVE ${CMAKE_SOURCE_DIR} *.cpp *.h) です。
EXCLUDE_SRCS	ビルドの対象から外すファイル。 上のSRCSでRELATIVEとしていないときは絶対パスで指定します。 SRCSでワイルドカードを使用した場合に有用です。
SPR_PROJS	アプリケーションに組み込むSpringheadライブラリのプロジェクト名。 不要なプロジェクト名を削除します。 この中にRunSwigを含めてはいけません。
ADDITIONAL_INCDIR	追加のインクルードパス指定。 現在のディレクトリは ${CMAKE_SOURCE_DIR}で参照できます。
ADDITIONAL_LIBDIR	追加のライブラリパス指定。
ADDITIONAL_LIBS	追加のライブラリファイル名。
EXCLUDE_LIBS	linkの対象から外すライブラリファイル名。 デフォルトで組み込まれてしまうライブラリファイルを排除するために指定します。
DEBUGGER_WORKING_DIRECTORY	Visual Studio Debuggerの作業ディレクトリ名。 デバッガはこのディレクトリで起動されたように振る舞います。
DEBUGGER_COMMAND_ARGUMENTS	Visual Studio Debuggerに渡すコマンド引数

自前でインストールしているパッケージ (boost, glew, freeglut, glui)を使用する場合には、 さらに、 配布されたファイル"CMakeConf.txt.dist"を"CMakeConf.txt"と

いう名前でコピーして 必要な編集をします。 “2.1.2 準備”を参照してください。

ビルドの条件(compile/linkのパラメータ)を変更したいときは、 配布されたファイル"CMakeOpts.txt.dist"を"CMakeOpts.txt"という名前でコピーして 適宜変更してください。

以上で準備作業は終了です。

3.1.2 7RJG1g82P0Zb5WIz

cmakeの実行手順については、 “2.1.3 cmakeの実行”と同じですから、 そちらを参照してください。 ここでは、 cmakeを実行することで “1.1.4 対処法”で示した対策がどのように実施されるかについて述べます。

ビルドの最適性

組み込まれているSpringheadライブラリの各プロジェクト (ここではBaseを例にします)に対して
"C:/Springhead/core/src/Base/*<x64>*/*<15.0>*/Base.dir"(*1)
というディレクトリを作成し、 "C:/Develop/Application/*build*/Base/Base.dir"から上記ディレクトリ(*1)へ linkを張る作業をcmake configure時に(自動的に)行ないます。

これに関して皆さんが行なうべきことはありませんが、 次のことに注意してください。

1. cmakeをした後でSpringheadソースツリー上ので上記ディレクトリ(*1)を削除すると、 以降のビルドで
   “エラー MSB3191 ディレクトリ"Base.dir/Debug/"を作成できません。”
   というエラーが発生します。
2. Springheadライブラリ側でcmake (configure)を実行していないと オブジェクトの共通格納領域が作成されていないため、 前項と同じエラーが発生します。
3. アプリケーション側で"C:/Develop/Application/*build*/Base/Base.dir"を削除すると、 ビルド時にVisual Studioが*build*下に"Base.dir"を 自動的に作成するためにビルドの最適性が崩れてしまいます (無駄なビルドが発生するだけで、ビルド自体は正常に行なえます)。

   Windowsでは実行権限の都合上linkをjunctionで実現していますが、explorerでもcommand prompt上でも"Base.dir"がjunctionか 通常のディレクトリかの区別がつきません。 そのためこのような事態の発見が困難になる可能性があります。

これらの状態を解消するためには、アプリケーション側 (もしくは**Springhead**ライブラリ側)で再度cmakeを実行する必要があります。

プロジェクトファイルの整合性

“1.1.4 対処法”でも述べたように、 プロジェクトファイル(ソリューションファイルの含む。以下同様)は Springheadライブラリのビルドツリーにあるものを最新の状態に保つことを前提として、 アプリケーション側のプロジェクトファイルは これらSpringhead側にあるものへのlinkとなるようにします。 このためにアプリケーションプログラムのソリューションに `sync`というターゲットを追加して、次の処理を実行させます。

1. もしもアプリケーション側のプロジェクトファイルの内容と Springhead側のプロジェクトファイルの内容とが異なっていたならば、 アプリケーション側のプロジェクトファイルを Springhead側にコピーする。

   これは、アプリケーション側でソースファイル構成を変更 (ファイルの追加・削除)を行ない、 Visual Studio上でプロジェクトファイルを保存したとき、 またはアプリケーション側で再度`cmake`を実行したときです。

2. アプリケーション側のプロジェクトファイルをSpringhead側の プロジェクトファイルへのlinkとする。

ターゲット`sync`はアプリケーションのビルドにおいて 必ず最初に実行されるように依存関係が設定されます。

次のことに注意してください。

1. Springhead側または他のアプリケーションが実施した変更は、 アプリケーションをビルドするだけで自動的に反映されます。

2. 自アプリケーションで実施したソース構成の変更は、 ビルドを実施した時点でSpringhead側に反映されます。 つまり、プロジェクトファイルの整合性を保つためには、 ソース構成の変更後に少なくとも1回はビルドを実行する必要がある ということです(`sync`の実行だけでもよい)。

   ソース構成を変更したらビルドするでしょうから、 このことが問題になることはほとんどないと思われます。
   もしSpringhead側でプロジェクトファイルが削除されたならば、 ビルドエラー(`sync`でlink先のファイルが見つからない) となります。
   この場合には`Springhead`側で再度`cmake`を実行する必要があります (アプリケーション側では駄目)。

3. `sync`ターゲットが実行されるとプロジェクトファイルが更新されることがあるため、“プロジェクトが環境外で変更された”旨の メッセージが出ることがあります。 「すべて再読み込み」としてください。

3.1.3 R8w47HeMnww1Wqw5

アプリケーションのビルドは従来と変わりがありませんが、 ソリューションに新しいターゲット ALL_BUILD, syncが追加されています。 ここでは、これらについて説明します。

ALL_BUILD

これはcmakeが自動的に作成するターゲットでmake allに相当するものとされています。ただしVisual Studio上ではALL_BUILDの依存関係の設定が不正確で、 このターゲットをビルドしても正しい結果は得られないようです。 **このターゲットは無視してください**

sync

これは“3.1.2 cmakeの実行”で述べたとおり、 プロジェクトファイルの整合性を保つために作られたものです。 Springheadライブラリのプロジェクトを一つでもビルドすれば このターゲットは必ず最初に実行されますから、 このターゲットに対して何らかのアクションを起こす必要はないでしょう。

補足

2019/9/30 (commit 1d8e5ce)以前に配布した"CMakeLists.txt.*.dist"を元に "CMakeLists.txt"を作成して使用している場合

RunSwigでclean/rebuildの対応ができていなかったため、 cleanと同等の機能を実現するためのターゲットRunSwig_Cleanが作成されて いるはずです。
上記日付以降のSpringheadライブラリをダウンロードしcmakeを実行していただければ、 RunSwigはclean/rebuild対応となります。 ただし、RunSwig_Cleanをビルドすると

```
python:  can't open file '.../Clean.py':  ...  No such file
...
```

というエラーが起きます。実害はありませんがRunSwigのcleanは行なわれません。

RunSwig_Cleanターゲットを生成されないようにするには、 新しい配布ファイルから"CMakeLists.txt"を再作成するか、 または既存の"CMakeLists.txt"から以下の部分を削除して再cmakeしてください。
EmbPython_RunSwig_Cleanについても同様です。

```
# ------------------------------------------------------------------------------
#  Clean (only for RunSwig).
#
if ("${CMAKE_GENERATOR_PLATFORM}" STREQUAL "x64")
    set(LIBDIR ${SPR_TOP_DIR}/generated/lib/win64)
else ()
    set(LIBDIR ${SPR_TOP_DIR}/generated/lib/win32)
endif()
set(LIBPATH_DEBUG   ${LIBDIR}/Springhead${LIBNAME_DEBUG}.lib)
set(LIBPATH_RELEASE ${LIBDIR}/Springhead${LIBNAME_RELEASE}.lib)
set(LIBPATH_TRACE   ${LIBDIR}/Springhead${LIBNAME_TRACE}.lib)
set(CLEAN ${Python} ${CMAKE_SOURCE_DIR}/RunSwig/Clean.py ${CMAKE_SOURCE_DIR})

if(${Windows})
    add_custom_target(RunSwig_Clean
        COMMAND if "'$(Configuration)'" equ "'Debug'"   ${CLEAN} ${LIBPATH_DEBUG}
        COMMAND if "'$(Configuration)'" equ "'Release'" ${CLEAN} ${LIBPATH_RELEASE}
        COMMAND if "'$(Configuration)'" equ "'Trace'"   ${CLEAN} ${LIBPATH_TRACE}
        COMMENT [[  clearing RunSwig generated files and Springhead Library... ]]
    )
elseif(${Linux})
endif()
```