1.流程上一共3大步骤：配置 生成 编译；在cmake的命令行中前两步骤被整合到一起去了

3.1. CMake stages

We start with a theory. Let’s introduce some terminology about CMake commands we have executed before.

3.1.1. Configure step

第一步

一般你的CMakeLists.txt是和你的source code放置到一起的

On this step CMake will parse top level CMakeLists.txt of source tree and create CMakeCache.txt file with cache variables. Different types of variables will be described further in details. For CMake-GUI this step triggered by clicking on `Configure` button. For CMake command-line this step is combined with generate step so terms configure and generate will be used interchangeably. The end of this step expressed by `Configuring done` message from CMake.

对于cmake-gui，configuration是和generation分开的，但是CLI中这两个是放置在一起的

3.1.1.1. GUI + Xcode example

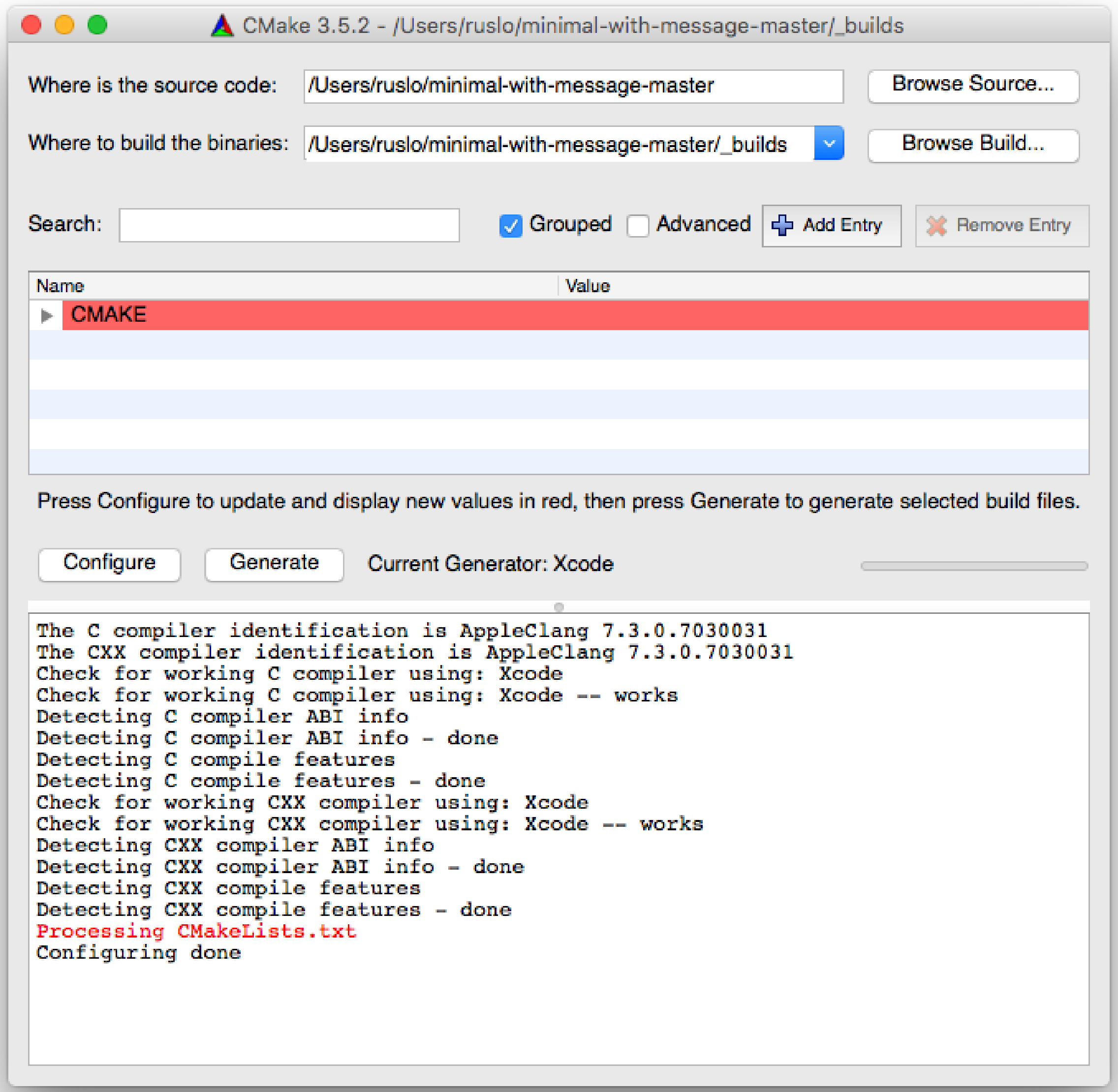
Let’s add message command to the example:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo foo.cpp)

message("Processing CMakeLists.txt")
```

Examples on GitHub

- Repository
- Latest ZIP

Line `Processing CMakeLists.txt` will be printed by CMake while parsing CMakeLists.txt file, i.e. on configure step. Open CMake-GUI, setup directories and hit `Configure`:



Where is the source code: /Users/ruslo/minimal-with-message-master

Where to build the binaries: /Users/ruslo/minimal-with-message-master/_builds

Press Configure to update and display new values in red, then press Generate to generate selected build files.

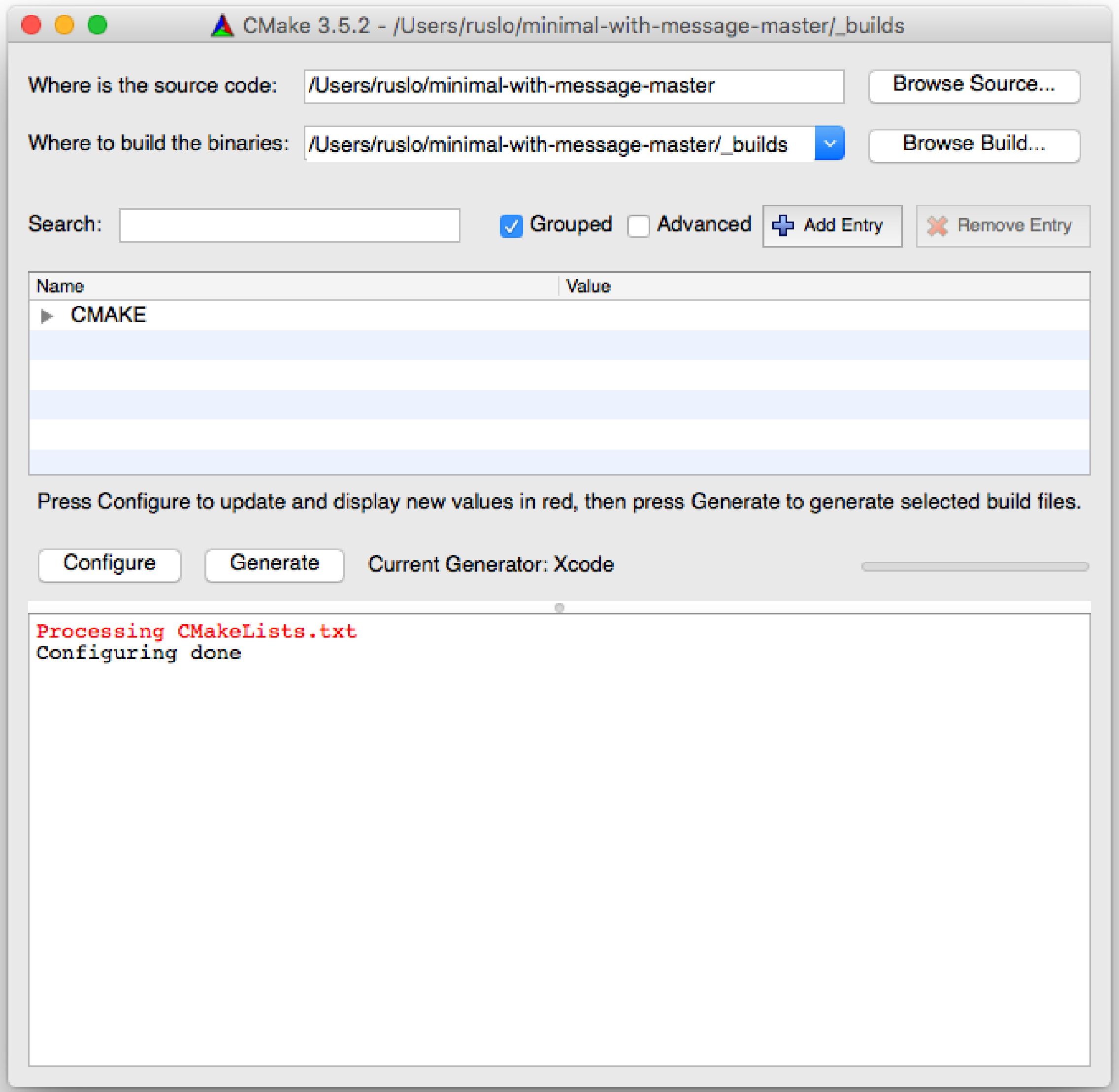
Current Generator: Xcode

```
The C compiler identification is AppleClang 7.3.0.7030031
The CXX compiler identification is AppleClang 7.3.0.7030031
Check for working C compiler using: Xcode
Check for working C compiler using: Xcode -- works
Detecting C compiler ABI info
Detecting C compiler ABI info - done
Detecting C compile features
Detecting C compile features - done
Check for working CXX compiler using: Xcode
Check for working CXX compiler using: Xcode -- works
Detecting CXX compiler ABI info
Detecting CXX compiler ABI info - done
Detecting CXX compile features
Detecting CXX compile features - done
Processing CMakeLists.txt
Configuring done
```

You can verify that there is no Xcode project generated by now but only CMakeCache.txt with cache variables:

```
[minimal-with-message-master]> ls _builds
CMakeCache.txt CMakeFiles/
```

Let’s run configure one more time:


CMake 3.5.2 - /Users/ruslo/minimal-with-message-master/_builds
Where is the source code: /Users/ruslo/minimal-with-message-master
Browse Source...
Where to build the binaries: /Users/ruslo/minimal-with-message-master/_builds
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Name
Value
CMAKE
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate
Current Generator: Xcode

```
Processing CMakeLists.txt
Configuring done
```

We still see `Process CMakeLists.txt` message which means that CMakeLists.txt was parsed again but there is no check/detect messages. This is because information about compiler and different tools detection results saved in CMake internal directories and reused. You may notice that second run happens much faster than first.

No surprises, there is still no Xcode project:

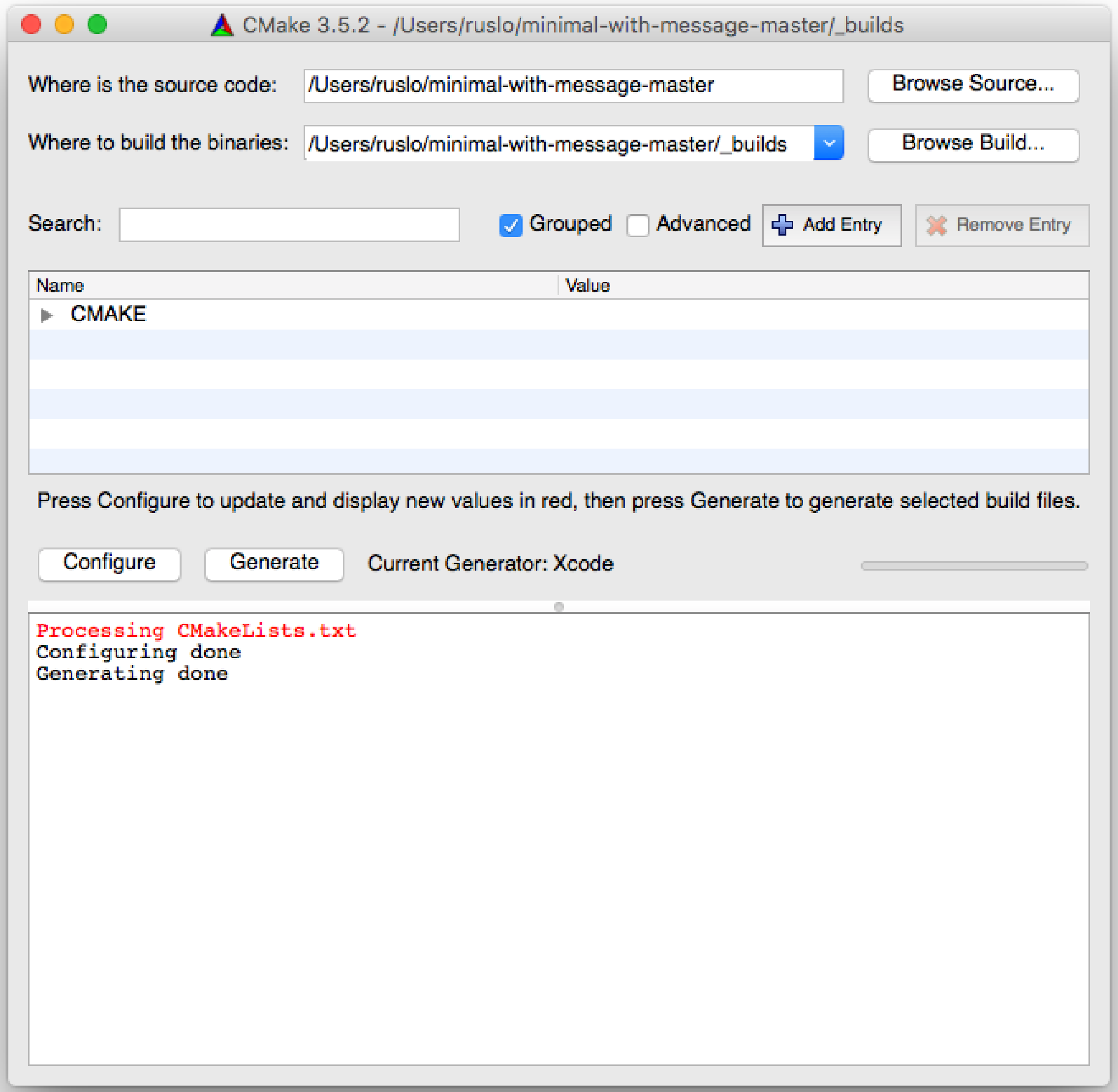
```
[minimal-with-message-master]> ls _builds
CMakeCache.txt CMakeFiles/
```

3.1.2. Generate step

On this step CMake will generate native build tool files using information from CMakeLists.txt and variables from CMakeCache.txt. For CMake-GUI this step triggered by clicking on `Generate` button. For CMake command-line this step is combined with configure step. The end of this step expressed by `Generating done` message from CMake.

3.1.2.1. GUI + Xcode example

Hit `Generate` now:



Where is the source code: /Users/ruslo/minimal-with-message-master

Where to build the binaries: /Users/ruslo/minimal-with-message-master/_builds

Search:

Grouped [✓] Advanced [ ]

CMAKE

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Xcode

```
Processing CMakeLists.txt
Configuring done
Generating done
```

Now Xcode project created:

```
[minimal-with-message-master]> ls -d _builds/foo.xcodeproj
_builds/foo.xcodeproj/
```

3.1.2.2. Makefile example

Example of generating Makefile on Linux:

```
[minimal-with-message-master]> rm -rf _builds
[minimal-with-message-master]> cmake -H. -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Processing CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../minimal-with-message-master/_builds
```

We see `Processing CMakeLists.txt`, `Configuring done` and `Generating done` messages meaning that CMakeLists.txt was parsed and both configure/generate steps combined into one action.

Verify Makefile generated:

```
[minimal-with-message-master]> ls _builds/Makefile
_builds/Makefile
```

If you run configure again CMakeLists.txt will be parsed one more time and `Processing CMakeLists.txt` will be issued:

```
[minimal-with-message-master]> cmake -H. -B_builds
Processing CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../minimal-with-message-master/_builds
```

3.1.3. Build step

This step is orchestrated by native build tool. On this step targets of your project will be build.

3.1.3.1. Xcode example

Run native tool build:

```
[minimal-with-message-master]> cmake --build _builds

=== BUILD AGGREGATE TARGET ZERO_CHECK OF PROJECT foo WITH CONFIGURATION Debug ===

Check dependencies

...

=== BUILD TARGET foo OF PROJECT foo WITH CONFIGURATION Debug ===

...

    /.../Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/clang -x c++
...
        -c /.../minimal-with-message-master/foo.cpp
        -o /.../minimal-with-message-master/_builds/foo.build/Debug/foo.build/Objects-
normal/x86_64/foo.o

...

    /.../Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/clang++ ...
        -o /Users/ruslo/minimal-with-message-master/_builds/Debug/foo

=== BUILD AGGREGATE TARGET ALL_BUILD OF PROJECT foo WITH CONFIGURATION Debug ===

...

Build all projects

** BUILD SUCCEEDED **
```

You can see that `foo.cpp` compiled into `foo.o` and then executable `foo` created. There is no `Processing CMakeLists.txt` message in output because on this stage CMake doesn't parse CMakeLists.txt however re-configure may be triggered on build step automatically, this will be shown in workflow section.

3.1.3.2. Makefile example

Run native tool build:

```
[minimal-with-message-master]> cmake --build _builds
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

You can see that `foo.cpp` compiled into `foo.cpp.o` and then executable `foo` created. There is no `Processing CMakeLists.txt` message in output because on this stage CMake doesn't parse CMakeLists.txt however re-configure may be triggered on build step automatically, this will be shown in workflow section.