A Brief Introduction to Build Systems

Chuan Zhang · Follow
Published in The Startup · 6 min read · Aug 13, 2020

Jan. 21. 2016

Roughly speadking, *build* in software development is the process of "translating" source code files into executable binary code files[1]; and a *build system* is a collection of software tools that is used to facilitate the build process[2]. Despite the fact that different build systems have been "invented" and used for over three decades, the core algorithms used in most of them are not changed much since the first introduction of the *directed acyclic graph (DAG)* by *Make*[3]. Popular build systems nowadays include the classical *GNU Make*, *CMake*, *QMake*, *Ninja*, *Ant*, *Scons*, and many others. In this note, I am going to demonstrate how to use some of these popular build systems to build C++ projects (to emphasize how the build systems work, the projects are extremely simplified).

Toy Project

The toy project I am going to use will just produce an executable binary code file which simply print out the string "Hello World!" on the screen. Below is the architecture of the project (a name without extension represent a directory).

```
// Project Directory Tree
/home/[username]/project-hello
        bin
        include
                print.cpp
                print.h
        lib
        obj
```

```
        src
                main.cpp
```

Below are the code in the three source code files.

```
// print.h
#include <iostream>
#include <string>

class Print
{
public:
    void operator()(const std::string&);
};

// print.cpp
#include “print.h”

void Print::operator()(const std::string &str)
{
    std::cout << str;
}

// main.cpp
#include “../include/print.h”

int main(int argc, char* argv[])
{
    Print print;
    print(“Hello World!\n”);
    return 0;
}
```

GNU Make

GNU Make build projects following the information provided in a file called *makefile*[4]. To build a project, one needs to write a makefile to tell GNU Make how. In the example below, I demonstrate how to write *makefile* and build an extremely simplified project. Put the following *Makefile* into the root directory of the project, i.e. the `project-hello` directory.

```
# Makefile
CXX = g++
AR = ar

SOURCEPATH = $(PWD)/src
INCLUDEPATH = $(PWD)/include
OBJECTPATH = $(PWD)/obj
LIBRARYPATH = $(PWD)/lib
BINARYPATH = $(PWD)/bin

all: hello-static hello-shared

hello-static: print-static $(SOURCEPATH)/main.cpp
    $(CXX) $(SOURCEPATH)/main.cpp -L$(LIBRARYPATH) -lprint_static -o
```

```
$(BINARYPATH)/$@

hello-shared: print-shared $(SOURCEPATH)/main.cpp
    $(CXX) $(SOURCEPATH)/main.cpp -L$(LIBRARYPATH) -lprint_shared -o
$(BINARYPATH)/$@
    export LD_LIBRARY_PATH=”$(LIBRARYPATH)”

print-static: obj-static
    $(AR) -cru $(LIBRARYPATH)/libprint_static.a
$(OBJECTPATH)/print_static.o

print-shared: obj-shared
    $(CXX) -shared $(OBJECTPATH)/print_shared.o -o
$(LIBRARYPATH)/libprint_shared.so

obj-static: $(INCLUDEPATH)/print.h
    $(CXX) -c -Wall $(INCLUDEPATH)/print.cpp -o
$(OBJECTPATH)/print_static.o

obj-shared: $(INCLUDEPATH)/print.h
    $(CXX) -fPIC -c $(INCLUDEPATH)/print.cpp -o
$(OBJECTPATH)/print_shared.o

clean:
    find . -name “*.so” -type f -print0 | xargs -0 rm -f
    find . -name “*.a”  -type f -print0 | xargs -0 rm -f
    find . -name “*.o”  -type f -print0 | xargs -0 rm -f
    rm -f $(BINARYPATH)/*
```

With the *Makefile* provided about, an executable program, static library, and shared library can be produced by respectively running `make all` (or simply `make`), `make print-static`, and `make print-shared` commands in terminal. To remove all generated binary code files, one just needs to run the command `make clean`. For more details, please check [4] or other similar references.

QMake

Like CMake, QMake is an open-source, cross-platform build system. QMake was created by Trolltech (now owned by Digia), and shipped with the Qt toolkit [5]. Essentially QMake automates the generation of `Makefile`s. Although Qt has its own *moc* (meta-object compiler) and *uic* (user-interface compiler) for its own project building process, qmake can be used for building any C++ projects with or without using these features. Next, I demonstrate how to use qmake to build our *toy project*.

To tell qmake how to generate Makefiles appropriately, project (`.pro`, `.pri`) files are needed. Although in some trivial special cases, the `.pro` file automatically generated by running `qmake` in `-project` mode may be enough, in most cases, project files need to be manually edited to meet the production build requirements. Below I give another directory tree to

include the project files I added for building the toy project (again names without extension are directories).

```
// Project Directory Tree
        project-hello
                project-hello.pro
                bin
                include
                        print.cpp
                        print.h
                lib
                        lib.pro
                src
                        main.cpp
                        src.pro
```

Below are the three project files.

```
# project-hello.pro
TEMPLATE = subdirs
SUBDIRS += lib
SUBDIRS += src
CONFIG += ordered

# src.pro
TEMPLATE = app
TARGET = hello
DESTDIR = ../bin
DEPENDPATH += .

LIBS += -L../lib -lprint-static # link against static library
#LIBS += -L../lib -lprint-shared # link against shared library

SOURCES += main.cpp

# lib.pro, for building static library
TEMPLATE = lib
TARGET = print-static
VERSION = 1.0.0
CONFIG += staticlib

# Directories
DESTDIR = ../lib
INCLUDEPATH += ../include
DEPENDPATH += ../include
HEADERS += ../include/print.h
SOURCES += ../include/print.cpp

# lib.pro, for building shared library
TEMPLATE = lib
TARGET = print-shared
VERSION = 1.0.0
CONFIG += dll

# Directories
DESTDIR = ../lib
INCLUDEPATH += ../include
DEPENDPATH += ../include
```

```
HEADERS += ../include/print.h
SOURCES += ../include/print.cpp
```

After the project files are created and edited appropriately, we are ready to build our toy project. By running `qmake -makefile` command (as `-makefile` is the default mode, we may simply run `qmake` command) to generate `Makefile`s, and `make` command, the project will be build successfully.

Possibly you have noticed that in the above project files, I gave two versions of the project file, *lib.pro*. One is for building a *static library*, the other is for *shared library*. If the executable binary code file is linked against the static library, it can run without loading anything else. But if it is linked against the shared library, the executable binary code file needs to load the shared library to get the code implementing the *functor* (aka *function object*), `print`. We have three different ways to tell the system how to find the shared library. One is to copy the shared library file to the `./bin` directory; the second is to copy the shared library file to the system directory `/usr/local/lib`; the third is to put the line

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/[username]/project-
hello/lib
```

into the hidden file `/home/[username]/.bash_profile`.

GYP — A Meta-Build System

`GYP`, abbreviation for *Generate Your Projects*, is a *Meta-Build system*, a build system that generates other build systems[8]. Basically, it is a *python dictionary*[8], which contains all information on how to *generate your project*. In the entire gyp file as a python dictionary, you can have four different types of data structures, *string*, *integer*, *list*, and again, *dictionary*. Not only can you set up your project, but also execute commands. Next, I demonstrate how to use it to generate our toy project, and build it. For a detailed tutorial on gyp, the official documentation [8] is thorough and decent.

```
# project-hello.gyp
{
    'targets': [{
```

```
        'target_name': 'hello-static',
        'type': 'static_library',
        'include_dirs': ['include',],
        'sources': [
            'include/print.cpp',
            'include/print.h',
        ],
    }, {
        'target_name': 'hello-shared',
        'type': 'shared_library',
        'include_dirs': ['include',],
        'sources': [
            'include/print.cpp',
            'include/print.h',
        ],
        'cflags': ['-fPIC',],
    }, {
        'target_name': 'hello-gyp',
        'type': 'executable',
        'include_dirs': [
            '<(DEPTH)',
            '<(DEPTH)/include',
        ],
        'dependencies': [
            'hello-static',
            #'hello-shared',
        ],
        'sources': [
            'src/main.cpp',
            'include/print.h',
        ],
    },],
}
```

Using `gyp`, projects for most build systems can be generated. To choose a specific build system for out projects, one just needs to specify it when invoking the generator command `gyp` by setting `-f build_system_name`. Below, I demonstrate this for generating projects for both *GNU Make* and *Ninja* build systems. For instructions for generating projects for other build systems, please refer to the official documentation[8].

```
gyp project-hello.gyp -f make — depth=. — generator-
output=build/makefiles

cd build/makefiles

make

gyp project-hello.gyp -f ninja — depth=. — generator-
output=build/ninja

cd build/ninja

ninja -C out/Default/
```

Although projects for most build systems can be generated with gyp generator, Qt project can not be generated with gyp. This causes some inconvenience sometimes when people needs to use something like the widgets from Qt framework, but for some reason, switching the build system from gyp to qmake is not possible, at least practically. In this case, one may try to "embed" some qmake build system features, say *moc*, the *meta-object compiler* into gyp system. In my *BuildSystems* repository, I included one example project, which uses both *widgets* and *signal-slot* from qt framework. I used gyp with *moc* "embedded" to build the project. If you are interested, you can check the code and scripts here.

References

[1]. Wikipedia: Software Build

[2]. Dan Williams (2009) Overview of Build Systems

[3]. Mike Shal (2009) Build System Rules and Algorithms

[4]. Robert Mecklenburg (2004) Managing Projects with GNU Make, 3rd Edition, O'Reilly Media, Inc., Sebastopol, CA; GNU Make

[5]. Ken Martin & Bill Hoffman (2015) Mastering CMake — A Cross- Platform Build System, CMake 3.1, Kitware, Inc.

[6]. Johan Thelin (2007) Foundations of Qt Development, Apress, Berkeley, CA; qmake manual; Chuan Zhang (2016) Develop Qt GUI Widgets without Using QtCreator

[7]. The Ninja Build System

[8]. GYP: Generate Your Projects

Cpp Gnu Make Qt Gyp Build System

Written by Chuan Zhang

61 Followers · Writer for The Startup

Follow

Software Engineer

More from Chuan Zhang and The Startup

Chuan Zhang in Python in Plain English

Introduction to Flask AppBuilder — Building a Simple Web Service

A step-by-step guide on how to use FAB to build a simple web app.

Dec 13, 2021 77 3

Jano le Roux in The Startup

How to Get a Secret Verified Email Badge (That Almost No One Has)

Stop landing in that damn SPAM folder.

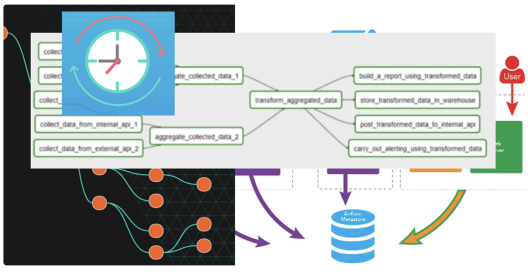
Aug 21 1.2K 29

Greyson Ferguson in The Startup

Countries You Can Move To Indefinitely Without a Visa

Why deal with all of that visa paperwork when you could just move without filling out a sing...

Jul 4 7.1K 86



Chuan Zhang in Python in Plain English

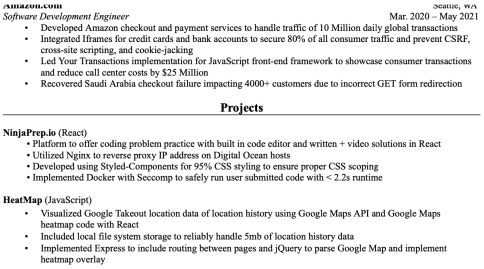
Asynchronous Task Queuing with Celery

A detailed guide on how Celery asynchronous task queue works.

Jan 5, 2022 61

See all from Chuan Zhang | See all from The Startup

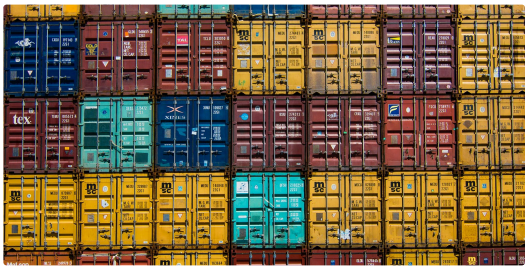
Recommended from Medium


Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021
- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay




Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

Jun 1 · 19.97K · 358


Mahdi Mallaki in ITNEXT

Dockerfile-less and Daemon-less Build

Building a Docker image without requiring a Dockerfile or Docker Daemon

2d ago · 100 · 2

Lists

Staff Picks

723 stories · 1268 saves

Stories to Help You Level-Up at Work

19 stories · 776 saves

Self-Improvement 101

20 stories · 2665 saves

Productivity 101

20 stories · 2287 saves