GN: A Modern Build System For BSD ?

https://gn.googlesource.com/gn

Bucharest, Septembre 23rd 2018

Benjamin Jacobs

(Original) Motivation

- As an **easy** way to import LLVM into DragonFlyBSD base
- Under a (naive) assumption, given that:
  - Upsream LLVM uses CMake which generates Ninja files
  - GN generates Ninja files
  - ~-> With **little** effort it could be possible to merge both (?)

What is GN ?

- Developped at Google to replace Gypp
- Used to build the chromium browser and the chrome OS since 2016
- Generates Ninja files
- Can also generate VS and XCode projects
- Used to live inside the chromium repository
- C++ code base
- Uses 200+ C++ file from base and platform support libraries of chromium + libevent
- 3.8MB all statically linked
- 3-Clause BSD License

(And Ninja?)

- «It is designed to have its input files generated by a higher-level build system, and it is designed to run builds as fast as possible.» (https://ninja-build.org/)
- Computes the dependency graph and executes the commands used to update inexisting or outdated target
- Also updates a target when the command has been changed, e.g. modifying CFLAGS.
- Some well known generators: CMake, meson.
- Apache License 2.0

What Could Be a Modern Build System?

- Performant*
- Correct
- Easy to use and still powerful enough
- But more importantly, hard to misuse
- Extensible ?

*: See http://gittup.org/tup/ Mike Shal's paper (2009)

For BSD?

- Open source license
- Community support
- Portable C/C++ codebase
- Better cross-compilation support
- Your answer at the end?

GN's Language

- `gn help grammar`
- Simple C-like syntax
- Booleans, integers, strings and (nestable) lists variables
- Primitive expressions: value substitution, list concatenation, logical, …
- Primitive control structures: `foreach (iterator, list) {...}` and `if (expr) {...}`
- Limited path substitution functions, e.g. to construct path relative to the build directory
- Script can be run at the evaluation/generation stage whose output can be used by GN itself.
- Note: The language is evaluated, however the separation imposed between the generation phase and the compilation/building phase makes it "mostly declarative".

GN's Language (continued)

- Higher-level elements:
    - targets
    - configurations
    - args: build knobs (versus /etc/default/make.conf)
    - toolchains
    - template: macro-like construction, e.g.
- Hygenic and strict variable propagation and scoping rules
- Import (versus make's #include)

Only a Handful of Target Declaration Types

- Builtin types: It is not possible to define arbitrary rules like Make's wildcard rules (`sys.mk`)!
- Strictly tailored towards the C language:
  - `executable`
  - `loadable_module`
  - `shared_library`
  - `static_library`
  - `source_set`: like a static library without the intermediary linking step, to be used as dependency
- A `toolchain` provides the actual command to be run
- The `action` / `action_foreach` targets can call out to an external command. This is mostly used to replace inline sh/sed/awk rules generating files.

More Targets

- `action` / `action_foreach`: run an external script
- macOS specifics
- file copy
- group: meta target

Toolchain Definition

- `gn help toolchain` / `gn help tool`
- Compiling tools: "cc", "cxx", "objc", "objcxx", "rc", "asm"
- Linking tools: "alink", "solink", "link"
- There must be a single "default" toolchain defined but a target can be build using another one
- When it is the case, all the dependency graph will be duplicated using the other toolchain
- GN determines the compiler to be used by looking at the file extension (hardcoded)
- Causes an issue with .c files which really need to be compiled in C++ mode, e.g. binutils's gold.

Target configuration elements

- Hold include directories, defines, compiler *flags, an inputs dependencies.
- Can be specified on the target element or can be named and referenced by targets using `configs`, `public_configs` or `all_dependent_configs`.

```
config ("xxx_config") {
  includes = [ ".", "//contrib/xxx/include" ]
  defines = [ "HAVE_FOO" ]
}
target ("executable" ) { ... configs = [":xxx_config"] }
```

- Config are merged together: include directories, cflags
- `public_configs` or `all_dependent_configs` also apply to direct dependent, or transitively to all dependent's dependents.
- Help avoiding the proliferation of those unnecessary or redundant -I../../path/ and -Ddefines
- GN Goodie: header check mode: search for C `#include "file"`, where the include search directory wouldn't have been provided explicitely or implicitely by transitive dependencies.

Args, and Special Variables or Functions

- `target_gen_dir`, `target_out_dir`, `get_target_outputs()`. `rebase_path()`
- Whose values vary with the current toolchain

GN file locations, labels and references

- Label are use to reference target, dependencies, or config.
- 3 kind of reference: `/absolute/target:name`, `relative/target:name` and `//root-of-sources/relative/target:name` references.
- Looks for a `name` target defined in the `target/BUILD.gn` file.
- Alternate hierachy is possible, to avoid cluttering the tree with BUILD.gn files.

Optimistic demonstration

Final Words and Feedback

- It is all about lowering the burden of maintaining build scripts.
- What is your experience ?