GN Language and Operation

Contents

Introduction

This page describes many of the language details and behaviors.

Use the built-in help!

GN has an extensive built-in help system which provides a reference for every function and built-in variable. This page is more high-level.

```
gn help
```

You can also see the slides from a March, 2016 introduction to GN. The speaker notes contain the full content.

Design philosophy

- Writing build files should not be a creative endeavour. Ideally two people should produce the same buildfile given the same requirements. There should be no flexibility unless it's absolutely needed. As many things should be fatal errors as possible.
- The definition should read more like code than rules. I don't want to write or debug Prolog. But everybody on our team can write and debug C++ and Python.
- The build language should be opinionated as to how the build should work. It should not necessarily be easy or even possible to express arbitrary things. We should be changing source and tooling to make the build simpler rather than making everything more complicated to conform to external requirements (within reason).
- Be like Blaze when it makes sense (see "Differences and similarities to Blaze" below).

Language

GN uses an extremely simple, dynamically typed language. The types are:

- Boolean (`true` , `false`).
- 64-bit signed integers.
- Strings.
- Lists (of any other types).
- Scopes (sort of like a dictionary, only for built-in stuff).

There are some built-in variables whose values depend on the current environment. See `gn help` for more.

There are purposefully many omissions in the language. There are no user-defined function calls, for example (templates are the closest thing). As per the above design philosophy, if you need this kind of thing you're probably doing it wrong.

The variable `sources` has a special rule: when assigning to it, a list of exclusion patterns is applied to it. This is designed to automatically filter out some types of files. See `gn help set_sources_assignment_filter` and `gn help label_pattern` for more.

The full grammar for language nerds is available in `gn help grammar` .

Strings

Strings are enclosed in double-quotes and use backslash as the escape character. The only escape sequences supported are:

- `\"` (for literal quote)
- `\$` (for literal dollars sign)
- `\\` (for literal backslash)

Any other use of a backslash is treated as a literal backslash. So, for example, `\b` used in patterns does not need to be escaped, nor do most Windows paths like `"C:\foo\bar.h"`.

Simple variable substitution is supported via `$`, where the word following the dollars sign is replaced with the value of the variable. You can optionally surround the name with `{}` if there is not a non-variable-name character to terminate the variable name. More complex expressions are not supported, only variable name substitution.

```
a = "mypath"
b = "$a/foo.cc"  # b -> "mypath/foo.cc"
c = "foo${a}bar.cc"  # c -> "foomypathbar.cc"
```

You can encode 8-bit haracters using “$0xFF” syntax, so a string with newlines (hex 0A) would `“look$0x0Alike$0x0Athis”

Lists

There is no way to get the length of a list. If you find yourself wanting to do this kind of thing, you’re trying to do too much work in the build.

Lists support appending:

```
a = [ "first" ]
a += [ "second" ]  # [ "first", "second" ]
a += [ "third", "fourth" ]  # [ "first", "second", "third", "fourth" ]
b = a + [ "fifth" ]  # [ "first", "second", "third", "fourth", "fifth" ]
```

Appending a list to another list appends the items in the second list rather than appending the list as a nested member.

You can remove items from a list:

```
a = [ "first", "second", "third", "first" ]
b = a - [ "first" ]  # [ "second", "third" ]
a -= [ "second" ]  # [ "first", "third", "fourth" ]
```

The - operator on a list searches for matches and removes all matching items. Subtracting a list from another list will remove each item in the second list.

If no matching items are found, an error will be thrown, so you need to know in advance that the

item is there before removing it. Given that there is no way to test for inclusion, the main use-case is to set up a master list of files or flags, and to remove ones that don't apply to the current build based on various conditions.

Stylistically, prefer to only add to lists and have each source file or dependency appear once. This is the opposite of the advice Chrome-team used to give for GYP (GYP would prefer to list all files, and then remove the ones you didn't want in conditionals).

Lists support zero-based subscripting to extract values:

```
a = [ "first", "second", "third" ]
b = a[1]  # -> "second"
```

The [] operator is read-only and can not be used to mutate the list. The primary use-case of this is when an external script returns several known values and you want to extract them.

There are some cases where it's easy to overwrite a list when you mean to append to it instead. To help catch this case, it is an error to assign a nonempty list to a variable containing an existing nonempty list. If you want to get around this restriction, first assign the destination variable to the empty list.

```
a = [ "one" ]
a = [ "two" ]  # Error: overwriting nonempty list with a nonempty list.
a = []         # OK
a = [ "two" ]  # OK
```

Note that execution of the build script is done without intrinsic knowledge of the meaning of the underlying data. This means that it doesn't know that `sources` is a list of file names, for example. So if you remove an item, it must match the literal string rather than specifying a different name that will resolve to the same file name.

Conditionals

Conditionals look like C:

```
if (is_linux || (is_win && target_cpu == "x86")) {
  sources -= [ "something.cc" ]
} else if (...) {
  ...
} else {
  ...
}
```

You can use them in most places, even around entire targets if the target should only be declared in certain circumstances.

Looping

You can iterate over a list with `foreach`. This is discouraged. Most things the build should do can normally be expressed without doing this, and if you find it necessary it may be an indication you're doing too much work in the metabuild.

```
foreach(i, mylist) {
  print(i)  # Note: i is a copy of each element, not a reference to it.
}
```

Function calls

Simple function calls look like most other languages:

```
print("hello, world")
assert(is_win, "This should only be executed on Windows")
```

Such functions are built-in and the user can not define new ones.

Some functions take a block of code enclosed by `{ }` following them:

```
static_library("mylibrary") {
  sources = [ "a.cc" ]
}
```

Most of these define targets. The user can define new functions like this with the template mechanism discussed below.

Precisely, this expression means that the block becomes an argument to the function for the function to execute. Most of the block-style functions execute the block and treat the resulting scope as a dictionary of variables to read.

Scoping and execution

Files and function calls followed by `{ }` blocks introduce new scopes. Scopes are nested. When you read a variable, the containing scopes will be searched in reverse order until a matching name is found. Variable writes always go to the innermost scope.

There is no way to modify any enclosing scope other than the innermost one. This means that when you define a target, for example, nothing you do inside of the block will "leak out" into the rest of the file.

`if` / `else` / `foreach` statements, even though they use `{ }`, do not introduce a new scope so changes will persist outside of the statement.

Naming things

File and directory names

File and directory names are strings and are interpreted as relative to the current build file's directory. There are three possible forms:

Relative names:

```
"foo.cc"
"src/foo.cc"
"../src/foo.cc"
```

Source-tree absolute names:

```
"//net/foo.cc"
"//base/test/foo.cc"
```

System absolute names (rare, normally used for include directories):

```
"/usr/local/include/"
"/C:/Program Files/Windows Kits/Include"
```

Labels

Everything that can participate in the dependency graph (targets, configs, and toolchains) are identified by labels which are strings of a defined format. A common label looks like this:

```
"//base/test:test_support"
```

which consists of a source-root-absolute path, a colon, and a name. This means to look for the thing named "test_support" in `src/base/test/BUILD.gn`.

When loading a build file, if it doesn't exist in the given location relative to the source root, GN will look in the secondary tree in `build/secondary`. The structure of this tree mirrors the main repository and is a way to add build files for directories that may be pulled from other repositories where we can't easily check in BUILD files. The secondary tree is a fallback rather than an override, so a file in the normal location always takes precedence.

A canonical label also includes the label of the toolchain being used. Normally, the toolchain label is implicitly inherited, but you can include it to specify cross-toolchain dependencies (see "Toolchains" below).

```
"//base/test:test_support(//build/toolchain/win:msvc)"
```

In this case it will look for the toolchain definition called "msvc" in the file `//build/toolchain/win` to know how to compile this target.

If you want to refer to something in the same buildfile, you can omit the path name and just start with a colon.

```
":base"
```

Labels can be specified as being relative to the current directory. Stylistically, we prefer to use absolute paths for all non-file-local references unless a build file needs to be run in different contexts (like a project needs to be both standalone and pulled into other projects in difference places in the directory hierarchy).

```
"source/plugin:myplugin"  # Prefer not to do these.
"../net:url_request"
```

If a name is unspecified, it will inherit the directory name. Stylistically, we prefer to omit the colon and name in these cases.

```
"//net" = "//net:net"
"//tools/gn" = "//tools/gn:gn"
```

Build configuration

Overall build flow

1. Look for `.gn` file in the current directory and walk up the directory tree until one is found. Set this directory to be the “source root” and interpret this file to find the name of the build config file.
2. Execute the build config file (this is the default toolchain). In Chrome this is `//build/config/BUILDCONFIG.gn`.
3. Load the `BUILD.gn` file in the root directory.
4. Recursively load `BUILD.gn` in other directories to resolve all current dependencies. If a BUILD file isn’t found in the specified location, GN will look in the corresponding location inside `build/secondary`.
5. When a target’s dependencies are resolved, write out the `.ninja` file to disk.
6. When all targets are resolved, write out the root `build.ninja` file.

The build config file

The first file executed is the build config file. The name of this file is specified in the `.gn` file that marks the root of the repository. In Chrome it is `//build/config/BUILDCONFIG.gn`. There is only one build config file.

This file sets up the scope in which all other build files will execute. Any arguments, variables, defaults, etc. set up in this file will be visible to all files in the build.

It is executed once for each toolchain (see “Toolchains”).

Build arguments

Arguments can be passed in from the command line (and from other toolchains, see "Toolchains" below). You declare which arguments you accept and specify default values via `declare_args`.

See `gn help buildargs` for an overview of how this works. See `gn help declare_args` for specifics on declaring them.

It is an error to declare a given argument more than once in a given scope. Typically arguments would be declared in an imported file (to share them among some subset of the build) or in the main build config file (to make them global).

Target defaults

You can set up some default values for a given target type. This is normally done in the build config file to set a list of default configs that defines the build flags and other setup information for each target type.

See `gn help set_defaults`.

For example, when you declare a `static_library`, the target defaults for a static library are applied. These values can be overwritten, modified, or preserved by a target.

```
# This call is typically in the build config file (see above).
set_defaults("static_library") {
  configs = [ "//build:rtti_setup", "//build:extra_warnings" ]
}

# This would be in your directory's BUILD.gn file.
static_library("mylib") {
  # At this point configs is set to [ "//build:rtti_setup", "//build:extra_wa
  # by default but may be modified.
  configs -= "//build:extra_warnings"  # Don't want these warnings.
  configs += ":mylib_config"  # Add some more configs.
}
```

The other use-case for setting target defaults is when you define your own target type via `template` and want to specify certain default values.

Targets

A target is a node in the build graph. It usually represents some kind of executable or library file that will be generated. Targets depend on other targets. The built-in target types (see `gn help <targettype>` for more help) are:

- `action`: Run a script to generate a file.
- `action_foreach`: Run a script once for each source file.
- `bundle_data`: Declare data to go into a Mac/iOS bundle.
- `create_bundle`: Creates a Mac/iOS bundle.

- `executable` : Generates an executable file.
- `group` : A virtual dependency node that refers to one or more other targets.
- `shared_library` : A .dll or .so.
- `loadable_module` : A .dll or .so loadable only at runtime.
- `source_set` : A lightweight virtual static library (usually preferrable over a real static library since it will build faster).
- `static_library` : A .lib or .a file (normally you'll want a `source_set` instead).

You can extend this to make custom target types using templates (see below). In Chrome some of the more commonly-used templates are:

- `component` : Either a source set or shared library, depending on the build type.
- `test` : A test executable. On mobile this will create the appropritate native app type for tests.
- `app` : Executable or Mac/iOS application.
- `android_apk` : Make an APK. There are a *lot* of other Android ones, see `//build/config/android/rules.gni` .

Configs

Configs are named objects that specify sets of flags, include directories, and defines. They can be applied to a target and pushed to dependent targets.

To define a config:

```
config("myconfig") {
  includes = [ "src/include" ]
  defines = [ "ENABLE_DOOM_MELON" ]
}
```

To apply a config to a target:

```
executable("doom_melon") {
  configs = [ ":myconfig" ]
}
```

It is common for the build config file to specify target defaults that set a default list of configs. Targets can add or remove to this list as needed. So in practice you would usually use `configs += ":myconfig"` to append to the list of defaults.

See `gn help config` for more information about how configs are declared and applied.

Public configs

A target can apply settings to other targets that depend on it. The most common example is a third party target that requires some defines or include directories for its headers to compile properly. You want these settings to apply both to the compile of the third party library itself, as

well as all targets that use the library.

To do this, you write a config with the settings you want to apply:

```
config("my_external_library_config") {
  includes = "."
  defines = [ "DISABLE_JANK" ]
}
```

Then this config is added to the target as a “public” config. It will apply both to the target as well as targets that directly depend on it.

```
shared_library("my_external_library") {
  ...
  # Targets that depend on this get this config applied.
  public_configs = [ ":my_external_library_config" ]
}
```

Dependent targets can in turn forward this up the dependency tree another level by adding your target as a “public” dependency.

```
static_library("intermediate_library") {
  ...
  # Targets that depend on this one also get the configs from "my external lib
  public_deps = [ ":my_external_library" ]
}
```

A target can forward a config to all dependents until a link boundary is reached by setting it as an `all_dependent_config`. This is strongly discouraged as it can spray flags and defines over more of the build than necessary. Instead, use public_deps to control which flags apply where.

In Chrome, prefer the build flag header system (`build/buildflag_header.gni`) for defines which prevents most screw-ups with compiler defines.

Toolchains

A toolchain is a set of build commands to run for different types of input files and link tasks.

You can have multiple toolchains in the build. It’s easiest to think about each one as completely separate builds that can additionally have dependencies between them. This means, for example, that the 32-bit Windows build might depend on a 64-bit helper target. Each of them can depend on `"//base:base"` which will be the 32-bit base in the context of the 32-bit toolchain, and the 64-bit base in the context of the 64-bit toolchain

When a target specifies a dependency on another target, the current toolchain is inherited unless it is explicitly overridden (see “Labels” above).

Toolchains and the build configuration

When you have a simple build with only one toolchain, the build config file is loaded only once at the beginning of the build. It must call `set_default_toolchain` to tell GN the label of the toolchain definition to use. This toolchain definition has the commands to use for the compiler and linker. The `toolchain_args` section of the toolchain definition is ignored.

When a target has a dependency on a target using different toolchain, GN will start a build using that secondary toolchain to resolve the target. GN will load the build config file with the arguments specified in the toolchain definition. Since the toolchain is already known, calls to `set_default_toolchain` are ignored.

So the toolchain configuration is two-way. In the default toolchain (i.e. the main build target) the configuration flows from the build config file to the toolchain: the build config file looks at the state of the build (OS type, CPU architecture, etc.) and decides which toolchain to use (via `set_default_toolchain`). In secondary toolchains, the configuration flows from the toolchain to the build config file: the `toolchain_args` in the toolchain definition specifies the arguments to re-invoke the build.

Toolchain example

Say the default build is a 64-bit build. Either this is the default CPU architecture based on the current system, or the user has passed `target_cpu="x64"` on the command line. The build config file might look like this to set up the default toolchain:

```
# Set default toolchain only has an effect when run in the context of
# the default toolchain. Pick the right one according to the current CPU
# architecture.
if (target_cpu == "x64") {
  set_default_toolchain("//toolchains:64")
} else if (target_cpu == "x86") {
  set_default_toolchain("//toolchains:32")
}
```

If a 64-bit target wants to depend on a 32-bit binary, it would specify a dependency using `data_deps` (data deps are like deps that are only needed at runtime and aren't linked, since you can't link a 32-bit and a 64-bit library).

```
executable("my_program") {
  ...
  if (target_cpu == "x64") {
    # The 64-bit build needs this 32-bit helper.
    data_deps = [ ":helper(//toolchains:32)" ]
  }
}

if (target_cpu == "x86") {
  # Our helper library is only compiled in 32-bits.
```

```
  shared_library("helper") {
    ...
  }
}
```

The toolchain file referenced above (`toolchains/BUILD.gn`) would define two toolchains:

```
toolchain("32") {
  tool("cc") {
    ...
  }
  ... more tools ...

  # Arguments to the build when re-invoking as a secondary toolchain.
  toolchain_args() {
    toolchain_cpu = "x86"
  }
}

toolchain("64") {
  tool("cc") {
    ...
  }
  ... more tools ...

  # Arguments to the build when re-invoking as a secondary toolchain.
  toolchain_args() {
    toolchain_cpu = "x64"
  }
}
```

The toolchain args specifies the CPU architecture explicitly, so if a target depends on something using that toolchain, that cpu architecture will be set when re-invoking the build. These args are ignored for the default toolchain since by the time they're known the build config has already been run. In general, the toolchain args and the conditions used to set the default toolchain should agree.

The nice thing about the multiple-build setup is that you can write conditionals in your targets referencing the current toolchain state. The build files will be re-run with different state for each toolchain. For the `my_program` example above, you can see it queries the CPU architecture, adding a dependency only for the 64-bit build of the program. The 32-bit build would not get this dependency.

Declaring a toolchain

Toolchains are declared with the `toolchain` command, which sets the commands to use for each compile and link operation. The toolchain also specifies a set of arguments to pass to the build config file when executing. This allows you to pass configuration information to the

alternate toolchain.

Templates

Templates are GN's primary way to re-use code. Typically, a template would expand to one or more other target types.

```
# Declares a script that compiles IDL files to source, and then compiles those
# source files.
template("idl") {
  # Always base helper targets on target_name so they're unique. Target name
  # will be the string passed as the name when the template is invoked.
  idl_target_name = "${target_name}_generate"
  action_foreach(idl_target_name) {
    ...
  }

  # Your template should always define a target with the name target_name.
  # When other targets depend on your template invocation, this will be the
  # destination of that dependency.
  source_set(target_name) {
    ...
    deps = [ ":$idl_target_name" ]  # Require the sources to be compiled.
  }
}
```

Typically your template definition would go in a `.gni` file and users would import that file to see the template definition:

```
import("//tools/idl_compiler.gni")

idl("my_interfaces") {
  sources = [ "a.idl", "b.idl" ]
}
```

Declaring a template creates a closure around the variables in scope at that time. When the template is invoked, the magic variable `invoker` is used to read variables out of the invoking scope. The template would generally copy the values its interested in into its own scope:

```
template("idl") {
  source_set(target_name) {
    sources = invoker.sources
  }
}
```

The current directory when a template executes will be that of the invoking build file rather than

the template source file. This is so files passed in from the template invoker will be correct (this generally accounts for most file handling in a template). However, if the template has files itself (perhaps it generates an action that runs a script), you will want to use absolute paths ("//foo/...") to refer to these files to account for the fact that the current directory will be unpredictable during invocation. See `gn help template` for more information and more complete examples.

Other features

Imports

You can import `.gni` files into the current scope with the `import` function. This is *not* an include in the C++ sense. The imported file is executed independently and the resulting scope is copied into the current file (C++ executes the included file in the current context of when the include directive appeared). This allows the results of the import to be cached, and also prevents some of the more "creative" uses of includes like multiply-included files.

Typically, a `.gni` would define build arguments and templates. See `gn help import` for more.

Your `.gni` file can define temporary variables that are not exported files that include it by using a preceding underscore in the name like `_this`.

Path processing

Often you will want to make a file name or a list of file names relative to a different directory. This is especially common when running scripts, which are executed with the build output directory as the current directory, while build files usually refer to files relative to their containing directory.

You can use `rebase_path` to convert directories. See `gn help rebase_path` for more help and examples. Typical usage to convert a file name relative to the current directory to be relative to the root build directory would be: `new_paths = rebase_path("myfile.c", root_build_dir)`

Patterns

Patterns are used to generate the output file names for a given set of inputs for custom target types, and to automatically remove files from the `sources` variable (see `gn help set_sources_assignment_filter`).

They are like simple regular expressions. See `gn help label_pattern` for more.

Executing scripts

There are two ways to execute scripts. All external scripts in GN are in Python. The first way is as a build step. Such a script would take some input and generate some output as part of the build. Targets that invoke scripts are declared with the "action" target type (see `gn help action`).

The second way to execute scripts is synchronously during build file execution. This is necessary in some cases to determine the set of files to compile, or to get certain system configurations

that the build file might depend on. The build file can read the stdout of the script and act on it in different ways.

Synchronous script execution is done by the `exec_script` function (see `gn help exec_script` for details and examples). Because synchronously executing a script requires that the current buildfile execution be suspended until a Python process completes execution, relying on external scripts is slow and should be minimized.

To prevent abuse, files permitted to call `exec_script` can be whitelisted in the toplevel `.gn` file. Chrome does this to require additional code review for such additions. See `gn help dotfile`.

You can synchronously read and write files which is discouraged but occasionally necessary when synchronously running scripts. The typical use-case would be to pass a list of file names longer than the command-line limits of the current platform. See `gn help read_file` and `gn help write_file` for how to read and write files. These functions should be avoided if at all possible.

Actions that exceed command-line length limits can use response files to get around this limitation without synchronously writing files. See `gn help response_file_contents`.

Differences and similarities to Blaze

Blaze is Google's internal build system, now publicly released as Bazel. It has inspired a number of other systems such as Pants and Buck.

In Google's homogeneous environment, the need for conditionals is very low and they can get by with a few hacks (`abi_deps`). Chrome uses conditionals all over the place and the need to add these is the main reason for the files looking different.

GN also adds the concept of "configs" to manage some of the trickier dependency and configuration problems which likewise don't arise on the server. Blaze has a concept of a "configuration" which is like a GN toolchain, but built into the tool itself. The way that toolchains work in GN is a result of trying to separate this concept out into the build files in a clean way.

GN keeps some GYP concept like "all dependent" settings which work a bit differently in Blaze. This is partially to make conversion from the existing GYP code easier, and the GYP constructs generally offer more fine-grained control (which is either good or bad, depending on the situation).

GN also uses GYP names like "sources" instead of "srcs" since abbreviating this seems needlessly obscure, although it uses Blaze's "deps" since "dependencies" is so hard to type. Chromium also compiles multiple languages in one target so specifying the language type on the target name prefix was dropped (e.g. from `cc_library`).

Powered by Gitiles source log blame