# What’s Up With BUILD.gn

With Special Guest Nico

## Intro

Building Chrome involves following a bunch of instructions copied over from a doc, but what’s the difference between these commands, and what do they actually do? Our special guest today is Nico. He’s responsible for making Ninja the Chrome default build system. He’s worked on Clang and all parts of the Chrome build.

## A quick guide on how to build Chromium

See Checking out and building Chromium on Linux. In short, download Depot tools, add to path, fetch Chromium, cd source, gclient sync, gn gen out/default, ninja -C out/default chrome, done! Let’s go over those in more detail.

**Depot tools**: A collection of utilities, including build tools and tools to get code from git and ninja. Used elsewhere and flexible in theory, but in practice, mostly used by Chromium projects
**GN**: stands for “generate ninja”. Describes the build and generates ninja files
**Ninja:** A build system, similar to Make. Gets a list of source files and build outputs, when you run it, figures out which build steps to run and runs them. Simpler and faster than make. Doesn’t have conditionals.
**.gclient:** reads the deps file at the root of the directory, downloads a couple hundred repos Chrome depends on, runs ‘hooks’ (python scripts, which download more stuff). Hooks are OS dependent.
**DEPS**: specifies dependencies that Chrome pulls in. Similar to git submodules.
**args.gn:** What’s needed for the specific build, e.g. whether or not DCHECKs (check out episode 2!) are enabled. Both gclient and argos.gn can be used to specify arguments about the build, such as operating system.

### autoninja -C out/default content

This is a typical command used to build Chrome. Let’s break it down by each part of the command.
**autoninja**: A wrapper around ninja, one of the things that’s in depot tools. Autoninja is optional, can just use ninja. autoninja will try to look at if goma is enabled if so it will build Chrome with multiple processes. Can run ‘which autoninja’ to get details about autoninja configurations
**-C**: sets the out directory to the argument passed in next (out/Default)
**out/Default**: the directory that built artifacts get put in, such as executables and anything else generated during the build. Can be any directory name, but out/* is already in the global .gitignore, which makes it a common choice. Want something that’s 2 directories deep, so the

path from the build directory to the source is ../.., which important for build determinism (see below for more). Many people have multiple out directories, for separate release, debug, ASAN builds, etc
**content**: name of the build target (check out episode 3 for more on content!). There are many ninja files that say which files are needed to build which target.
**Goma**: distributed compiler, access to server is not public

## Build Determinism

Build determinism means that you should get the exact same binary if you build Chrome at the same revision, independent of the host machine.
It seems like builds should just be deterministic to an average engineer, but that's because the build team has done such a great job of overcoming this.
Check out this blog post that gets more into all of this.

Some things that contribute to nondeterminism include:

- C++  preprocessor macro which embeds current date into the build
- Get absolute paths to everything in debug info, which differs for each build setup/engineer
- Tools get run that produce output, so if that binary iterates over some hash map, which might not have deterministic iteration orders

## Clang, compiling and build

Compiling means turning source files into object files. And there are a whole bunch of C++ compilers. Because c++ is evolving fast, we have to update many different compilers for different OSes, So over time, we moved from using basically the system compiler to using a hermetically built Clang that we download as a gclient DEPS hook. So when you run gclient sync, that downloads a prebuilt Clang binary. And we use that Clang binary to build Chrome on all operating systems. And that also enables cross builds, so you can build Chrome for Windows on Linux if you want to because your compiler is the same.
Also, Clang has a pretty good tooling interface. It's used to index Chrome for code search.

## gclient sync

When you run **gclient sync**, it basically syncs all these other Git repositories that are mentioned in the DEPS file. And after that, it runs hooks, which download a new Clang compiler and download a bunch of other binaries called the CIPD, for example, GN.
It makes sure that all the dependencies are up to date.

## //third_party

They are code that we didn't write ourselves. For example Blink is one of the things in //third_party, and it's there as a historical artifact. Also we use libPNG for reading PNG files. Some dependencies are listed in the DEPS file and will be pulled from third party libs and put into your source tree. For other third-party code, we actually check in the code into the Chrome main repository instead of DEPSing it in.
Almost no third-party dependency has a GN file upstream so usually you have to write your own BUILD.gn file for the third-party dependency you want to add.

## BUILD.gn

BUILD.gn contains build targets, a list of CC files that belong together and that either make up a static library or a shared library on executable. Check out Quick GN for more information.

How to involve other languages into Chrome build besides C++?

- First, you have to teach GN how to generate Ninja files for that language.
- Then you need to think how to make it work with Goma so that the compilation can work remotely instead of locally.

To make chrome build fast one of the techniques we use is Link Time Optimization, which means the C++ or the Rust code is compiled into something called "bitcode." And then all the bitcode files at link time are analyzed together so you can do cross-file linking. So if you want to do cross-language LTO, you have to update your C++ compiler and your Rust compiler at the same time. Check out chromium blog for more information.

## Code search (CS)

A CS bot, every six hours or so, pulls the latest code, bundles it up, and sends it to an indexer service. Clang is used to analyze the code for C++ and Java. Rust is not supported yet.
It generates metadata about classes, identifiers, functions, declarations and so on.

**git cl**: in depot_tools, when executed it redirects to git_cl.py.
**Presubmits**: run **git cl presubmit**, it'll look at a file called presubmit.py. One of the things it checks is code format.
**Tricium**: adds comments to your change list when you upload it, can do spelling correction, and also runs Clang Tidy which checks for unused variables.

Link Time Optimization blog post:
https://blog.chromium.org/2021/12/faster-chrome-let-the-compiler-do-the-work.html

Build mailing list: https://groups.google.com/a/chromium.org/g/build