Checking out and Building Chromium for Windows

There are instructions for other platforms linked from the get the code page.

Instructions for Google Employees

Are you a Google employee? See go/building-chrome-win instead.

Contents

System requirements

- A 64-bit Intel machine with at least 8GB of RAM. More than 16GB is highly recommended.
- At least 100GB of free disk space on an NTFS-formatted hard drive. FAT32 will not work, as some of the Git packfiles are larger than 4GB.
- An appropriate version of Visual Studio, as described below.
- Windows 10 or newer.

Setting up Windows

Visual Studio

Chromium requires Visual Studio 2017 (>=15.7.2) to build, but VS2019 (>=16.0.0) is preferred. Visual Studio can also be used to debug Chromium, and VS2019 is preferred for this as it handles Chromium's large debug information much better. The clang-cl compiler is used but Visual Studio's header files, libraries, and some tools are required. Visual Studio Community Edition should work if its license is appropriate for you. You must install the "Desktop development with C++" component and the "MFC/ATL support" sub-components. This can be done from the command line by passing these arguments to the Visual Studio installer (see below for ARM64 instructions):

```
$ PATH_TO_INSTALLER.EXE ^
--add Microsoft.VisualStudio.Workload.NativeDesktop ^
--add Microsoft.VisualStudio.Component.VC.ATLMFC ^
--includeRecommended
```

If you want to build for ARM64 Win32 then some extra arguments are needed. The full set for that case is:

```
$ PATH_TO_INSTALLER.EXE ^
--add Microsoft.VisualStudio.Workload.NativeDesktop ^
--add Microsoft.VisualStudio.Component.VC.ATLMFC ^
--add Microsoft.VisualStudio.Component.VC.Tools.ARM64 ^
--add Microsoft.VisualStudio.Component.VC.MFC.ARM64 ^
--includeRecommended
```

You must have the version 10.0.19041 or higher Windows 10 SDK installed. This can be installed separately or by checking the appropriate box in the Visual Studio Installer.

The SDK Debugging Tools must also be installed. If the Windows 10 SDK was installed via the Visual Studio installer, then they can be installed by going to: Control Panel → Programs → Programs and Features → Select the "Windows Software Development Kit" → Change → Change → Check "Debugging Tools For Windows" → Change. Or, you can download the standalone SDK installer and use it to install the Debugging Tools.

Install `depot_tools`

Download the depot_tools bundle and extract it somewhere.

> **Warning: DO NOT** use drag-n-drop or copy-n-paste extract from Explorer, this will not extract the hidden ".git" folder which is necessary for depot_tools to autoupdate itself. You can use "Extract all..." from the context menu though.

Add depot_tools to the start of your PATH (must be ahead of any installs of Python). Assuming you unzipped the bundle to C:\src\depot_tools, open:

Control Panel → System and Security → System → Advanced system settings

If you have Administrator access, Modify the PATH system variable and put `C:\src\depot_tools` at the front (or at least in front of any directory that might already have a copy of Python or Git).

If you don't have Administrator access, you can add a user-level PATH environment variable and put `C:\src\depot_tools` at the front, but if your system PATH has a Python in it, you will be out of luck.

Also, add a DEPOT_TOOLS_WIN_TOOLCHAIN system variable in the same way, and set it to 0. This tells depot_tools to use your locally installed version of Visual Studio (by default, depot_tools will try to use a google-internal version).

You may also have to set variable `vs2017_install` or `vs2019_install` to your installation path of Visual Studio 2017 or 19, like `set vs2019_install=C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional` for Visual Studio 2019.

From a cmd.exe shell, run the command gclient (without arguments). On first run, gclient will install all the Windows-specific bits needed to work with the code, including msysgit and python.

- If you run gclient from a non-cmd shell (e.g., cygwin, PowerShell), it may appear to run properly, but msysgit

- If you run gclient from a non-cmd shell (e.g., cygwin, PowerShell), it may appear to run properly, but msysgit, python, and other tools may not get installed correctly.
- If you see strange errors with the file system on the first run of gclient, you may want to disable Windows Indexing.

After running gclient open a command prompt and type `where python` and confirm that the depot_tools `python.bat` comes ahead of any copies of python.exe. Failing to ensure this can lead to overbuilding when using gn - see crbug.com/611087.

Get the code

First, configure Git:

```
$ git config --global user.name "My Name"
$ git config --global user.email "my-name@chromium.org"
$ git config --global core.autocrlf false
$ git config --global core.filemode false
$ git config --global branch.autosetuprebase always
```

Create a `chromium` directory for the checkout and change to it (you can call this whatever you like and put it wherever you like, as long as the full path has no spaces):

```
$ mkdir chromium && cd chromium
```

Run the `fetch` tool from `depot_tools` to check out the code and its dependencies.

```
$ fetch chromium
```

If you don't want the full repo history, you can save a lot of time by adding the `--no-history` flag to `fetch`.

Expect the command to take 30 minutes on even a fast connection, and many hours on slower ones.

When `fetch` completes, it will have created a hidden `.gclient` file and a directory called `src` in the working directory. The remaining instructions assume you have switched to the `src` directory:

```
$ cd src
```

Optional: You can also install API keys if you want your build to talk to some Google services, but this is not necessary for most development and testing purposes.

Setting up the build

Chromium uses Ninja as its main build tool along with a tool called GN to generate `.ninja` files. You can create any number of *build directories* with different configurations. To create a build directory:

```
$ gn gen out/Default
```

- You only have to run this once for each new build directory, Ninja will update the build files as needed.
- You can replace `Default` with another name, but it should be a subdirectory of `out`.
- For other build arguments, including release settings or using an alternate version of Visual Studio, see GN build

configuration. The default will be a debug component build matching the current host operating system and CPU.

- For more info on GN, run `gn help` on the command line or read the quick start guide.

Faster builds

- Reduce file system overhead by excluding build directories from antivirus and indexing software.
- Store the build tree on a fast disk (preferably SSD).
- The more cores the better (20+ is not excessive) and lots of RAM is needed (64 GB is not excessive).

There are some gn flags that can improve build speeds. You can specify these in the editor that appears when you create your output directory (`gn args out/Default`) or on the gn gen command line (`gn gen out/Default --args="is_component_build = true is_debug = true"`). Some helpful settings to consider using include:

- `is_component_build = true` - this uses more, smaller DLLs, and incremental linking.
- `enable_nacl = false` - this disables Native Client which is usually not needed for local builds.
- `target_cpu = "x86"` - x86 builds are slightly faster than x64 builds and support incremental linking for more targets. Note that if you set this but don't' set enable_nacl = false then build times may get worse.
- `blink_symbol_level = 0` - turn off source-level debugging for blink to reduce build times, appropriate if you don't plan to debug blink.

In order to speed up linking you can set `symbol_level = 1` or `symbol_level = 0` - these options reduce the work the compiler and linker have to do. With `symbol_level = 1` the compiler emits file name and line number information so you can still do source-level debugging but there will be no local variable or type information. With `symbol_level = 0` there is no source-level debugging but call stacks still have function names. Changing `symbol_level` requires recompiling everything.

In addition, Google employees should use goma, a distributed compilation system. Detailed information is available internally but the relevant gn arg is:

- `use_goma = true`

To get any benefit from goma it is important to pass a large -j value to ninja. A good default is 10*numCores to 20*numCores. If you run autoninja then it will automatically pass an appropriate -j value to ninja for goma or not.

```
$ autoninja -C out\Default chrome
```

When invoking ninja specify 'chrome' as the target to avoid building all test binaries as well.

Still, builds will take many hours on many machines.

Why is my build slow?

Many things can make builds slow, with Windows Defender slowing process startups being a frequent culprit. Have you ensured that the entire Chromium src directory is excluded from antivirus scanning (on Google machines this means putting it in a `src` directory in the root of a drive)? Have you tried the different settings listed above, including different link settings and -j values? Have you asked on the chromium-dev mailing list to see if your build is slower than expected for your machine's specifications?

The next step is to gather some data. If you set the `NINJA_SUMMARIZE_BUILD` environment variable to 1 then `autoninja` will do three things. First, it will set the NINJA_STATUS environment variable so that ninja will print additional information while building Chrome. It will show how many build processes are running at any given time, how many build steps have completed, how many build steps have completed per second, and how long the build has been running, as shown here:

```
$ set NINJA_SUMMARIZE_BUILD=1
$ autoninja -C out\Default base
ninja: Entering directory `out\Default'
[1 processes, 86/86 @ 2.7/s : 31.785s ] LINK(DLL) base.dll base.dll.lib base.dll.pdb
```

This makes slow process creation immediately obvious and lets you tell quickly if a build is running more slowly than normal.

In addition, setting `NINJA_SUMMARIZE_BUILD=1` tells `autoninja` to print a build performance summary when the build completes, showing the slowest build steps and slowest build-step types, as shown here:

```
$ set NINJA_SUMMARIZE_BUILD=1
$ autoninja -C out\Default base
Longest build steps:
       0.1 weighted s to build obj/base/base/trace_log.obj (6.7 s elapsed time)
       0.2 weighted s to build nasm.exe, nasm.exe.pdb (0.2 s elapsed time)
       0.3 weighted s to build obj/base/base/win_util.obj (12.4 s elapsed time)
       1.2 weighted s to build base.dll, base.dll.lib (1.2 s elapsed time)
Time by build-step type:
       0.0 s weighted time to generate 6 .lib files (0.3 s elapsed time sum)
       0.1 s weighted time to generate 25 .stamp files (1.2 s elapsed time sum)
       0.2 s weighted time to generate 20 .o files (2.8 s elapsed time sum)
       1.7 s weighted time to generate 4 PEFile (linking) files (2.0 s elapsed
time sum)
      23.9 s weighted time to generate 770 .obj files (974.8 s elapsed time sum)
26.1 s weighted time (982.9 s elapsed time sum, 37.7x parallelism)
839 build steps completed, average of 32.17/s
```

The “weighted” time is the elapsed time of each build step divided by the number of tasks that were running in parallel. This makes it an excellent approximation of how “important” a slow step was. A link that is entirely or mostly serialized will have a weighted time that is the same or similar to its elapsed time. A compile that runs in parallel with 999 other compiles will have a weighted time that is tiny.

You can also generate these reports by manually running the script after a build:

```
$ python depot_tools\post_build_ninja_summary.py -C out\Default
```

Finally, setting `NINJA_SUMMARIZE_BUILD=1` tells autoninja to tell Ninja to report on its own overhead by passing “-d stats”. This can be helpful if, for instance, process creation (which shows up in the StartEdge metric) is making builds slow, perhaps due to antivirus interference due to clang-cl not being in an excluded directory:

```
$ set NINJA_SUMMARIZE_BUILD=1
$ autoninja -C out\Default base
"c:\src\depot_tools\ninja.exe" -C out\Default base -j 10 -d stats
metric                  count   avg (us)        total (ms)
.ninja parse            3555    1539.4          5472.6
canonicalize str        1383032 0.0             12.7
canonicalize path       1402349 0.0             11.2
lookup node             1398245 0.0             8.1
.ninja log load         2       118.0           0.2
```

```
.ninja_deps load        2       67.5            0.1

node stat               2516    29.6            74.4
depfile load            2       1132.0          2.3
StartEdge               88      3508.1          308.7
FinishCommand           87      1670.9          145.4
CLParser::Parse         45      1889.1          85.0
```

You can also get a visual report of the build performance with ninjatracing. This converts the .ninja_log file into a .json file which can be loaded into chrome://tracing:

```
$ python ninjatracing out\Default\.ninja_log >build.json
```

Build Chromium

Build Chromium (the "chrome" target) with Ninja using the command:

```
$ autoninja -C out\Default chrome
```

`autoninja` is a wrapper that automatically provides optimal values for the arguments passed to `ninja`.

You can get a list of all of the other build targets from GN by running `gn ls out/Default` from the command line. To compile one, pass to Ninja the GN label with no preceding "//" (so for `//chrome/test:unit_tests` use ninja -C out/Default chrome/test:unit_tests`).

Run Chromium

Once it is built, you can simply run the browser:

```
$ out\Default\chrome.exe
```

(The ".exe" suffix in the command is actually optional).

Running test targets

You can run the tests in the same way. You can also limit which tests are run using the `--gtest_filter` arg, e.g.:

```
$ out\Default\unit_tests.exe --gtest_filter="PushClientTest.*"
```

You can find out more about GoogleTest at its GitHub page.

Update your checkout

To update an existing checkout, you can run

```
$ git rebase-update
$ gclient sync -D
```

The first command updates the primary Chromium source repository and rebases any of your local branches on top of tip-of-tree (aka the Git branch `origin/master`). If you don't want to use this script, you can also just use `git pull` or other common Git commands to update the repo.

The second command syncs the subrepositories to the appropriate versions, deleting those that are no longer needed, and re-runs the hooks as needed.

Editing and Debugging With the Visual Studio IDE

You can use the Visual Studio IDE to edit and debug Chrome, with or without Intellisense support.

Using Visual Studio Intellisense

If you want to use Visual Studio Intellisense when developing Chromium, use the `--ide` command line argument to `gn gen` when you generate your output directory (as described on the get the code page):

```
$ gn gen --ide=vs out\Default
$ devenv out\Default\all.sln
```

GN will produce a file `all.sln` in your build directory. It will internally use Ninja to compile while still allowing most IDE functions to work (there is no native Visual Studio compilation mode). If you manually run "gen" again you will need to resupply this argument, but normally GN will keep the build and IDE files up to date automatically when you build.

The generated solution will contain several thousand projects and will be very slow to load. Use the `--filters` argument to restrict generating project files for only the code you're interested in. Although this will also limit what files appear in the project explorer, debugging will still work and you can set breakpoints in files that you open manually. A minimal solution that will let you compile and run Chrome in the IDE but will not show any source files is:

```
$ gn gen --ide=vs --filters=//chrome --no-deps out\Default
```

You can selectively add other directories you care about to the filter like so: `--filters=//chrome;//third_party/WebKit/*;//gpu/*`.

There are other options for controlling how the solution is generated, run `gn help gen` for the current documentation.

Using Visual Studio without Intellisense

It is also possible to debug and develop Chrome in Visual Studio without the overhead of a multi-project solution file. Simply "open" your chrome.exe binary with `File->Open->Project/Solution`, or from a Visual Studio command prompt like so: `devenv /debugexe out\Debug\chrome.exe <your arguments>`. Many of Visual Studio's code exploration features will not work in this configuration, but by installing the VsChromium Visual Studio Extension you can get the source code to appear in the solution explorer window along with other useful features such as code search. You can add multiple executables of interest (base_unittests.exe, browser_tests.exe) to your solution with `File->Add->Existing Project...` and change which one will be debugged by right-clicking on them in `Solution Explorer` and selecting `Set as Startup Project`. You can also change their properties, including command line arguments, by right-clicking on them in `Solution Explorer` and selecting `Properties`.

By default when you start debugging in Visual Studio the debugger will only attach to the main browser process. To debug all of Chrome, install Microsoft's Child Process Debugging Power Tool. You will also need to run Visual Studio as administrator, or it will silently fail to attach to some of Chrome's child processes.