The Ninja build system

v1.10.2, Nov 2020

Introduction

Ninja is yet another build system. It takes as input the interdependencies of files (typically source code and output executables) and orchestrates building them, *quickly*.

Ninja joins a sea of other build systems. Its distinguishing goal is to be fast. It is born from my work on the Chromium browser project, which has over 30,000 source files and whose other build systems (including one built from custom non-recursive Makefiles) would take ten seconds to start building after changing one file. Ninja is under a second.

Philosophical overview

Where other build systems are high-level languages, Ninja aims to be an assembler.

Build systems get slow when they need to make decisions. When you are in a edit-compile cycle you want it to be as fast as possible — you want the build system to do the minimum work necessary to figure out what needs to be built immediately.

Ninja contains the barest functionality necessary to describe arbitrary dependency graphs. Its lack of syntax makes it impossible to express complex decisions.

Instead, Ninja is intended to be used with a separate program generating its input files. The generator program (like the `./configure` found in autotools projects) can analyze system dependencies and make as many decisions as possible up front so that incremental builds stay fast. Going beyond autotools, even build-time decisions like "which compiler flags should I use?" or "should I build a debug or release-mode binary?" belong in the `.ninja` file generator.

Design goals

Here are the design goals of Ninja:

- very fast (i.e., instant) incremental builds, even for very large projects.
- very little policy about how code is built. Different projects and higher-level build systems have different opinions about how code should be built; for example, should built objects live alongside the sources or should all build output go into a separate directory? Is there a "package" rule that builds a distributable package of the project? Sidestep these decisions by trying to allow either to be implemented, rather than choosing, even if that results in more verbosity.
- get dependencies correct, and in particular situations that are difficult to get right with Makefiles (e.g. outputs need an implicit dependency on the command line used to generate them; to build C source code you need to use gcc's `-M` flags for header dependencies).
- when convenience and speed are in conflict, prefer speed.

Some explicit *non-goals*:

- convenient syntax for writing build files by hand. *You should generate your ninja files using another program*. This is how we can sidestep many policy decisions.
- built-in rules. *Out of the box, Ninja has no rules for e.g. compiling C code.*
- build-time customization of the build. *Options belong in the program that generates the ninja files*.
- build-time decision-making ability such as conditionals or search paths. *Making decisions is slow.*

To restate, Ninja is faster than other build systems because it is painfully simple. You must tell Ninja exactly what to do when you create your project's `.ninja` files.

Comparison to Make

Ninja is closest in spirit and functionality to Make, relying on simple dependencies between file timestamps.

But fundamentally, make has a lot of *features*: suffix rules, functions, built-in rules that e.g. search for RCS files when building source. Make's language was designed to be written by humans. Many projects find make alone adequate for their build problems.

In contrast, Ninja has almost no features; just those necessary to get builds correct while punting most complexity to generation of the ninja input files. Ninja by itself is unlikely to be useful for most projects.

Here are some of the features Ninja adds to Make. (These sorts of features can often be implemented using more complicated Makefiles, but they are not part of make itself.)

- Ninja has special support for discovering extra dependencies at build time, making it easy to get header dependencies correct for C/C++ code.
- A build edge may have multiple outputs.
- Outputs implicitly depend on the command line that was used to generate them, which means that changing e.g. compilation flags will cause the outputs to rebuild.
- Output directories are always implicitly created before running the command that relies on them.
- Rules can provide shorter descriptions of the command being run, so you can print e.g. `CC foo.o` instead of a long command line while building.
- Builds are always run in parallel, based by default on the number of CPUs your system has. Underspecified build dependencies will result in incorrect builds.
- Command output is always buffered. This means commands running in parallel don't interleave their output, and when a command fails we can print its failure output next to the full command line that produced the failure.

Using Ninja for your project

Ninja currently works on Unix-like systems and Windows. It's seen the most testing on Linux (and has the best performance there) but it runs fine on Mac OS X and FreeBSD.

If your project is small, Ninja's speed impact is likely unnoticeable. (However, even for small projects it sometimes turns out that Ninja's limited syntax forces simpler build rules that result in faster builds.) Another way to say this is that if you're happy with the edit-compile cycle time of your project already then Ninja won't help.

There are many other build systems that are more user-friendly or featureful than Ninja itself. For some recommendations: the Ninja author found the tup build system influential in Ninja's design, and thinks redo's design is quite clever.

Ninja's benefit comes from using it in conjunction with a smarter meta-build system.

gn

The meta-build system used to generate build files for Google Chrome and related projects (v8, node.js), as well as Google Fuchsia. gn can generate Ninja files for all platforms supported by Chrome.

CMake
: A widely used meta-build system that can generate Ninja files on Linux as of CMake version 2.8.8. Newer versions of CMake support generating Ninja files on Windows and Mac OS X too.

others
: Ninja ought to fit perfectly into other meta-build software like premake. If you do this work, please let us know!

Running Ninja

Run `ninja`. By default, it looks for a file named `build.ninja` in the current directory and builds all out-of-date targets. You can specify which targets (files) to build as command line arguments.

There is also a special syntax `target^` for specifying a target as the first output of some rule containing the source you put in the command line, if one exists. For example, if you specify target as `foo.c^` then `foo.o` will get built (assuming you have those targets in your build files).

`ninja -h` prints help output. Many of Ninja's flags intentionally match those of Make; e.g `ninja -C build -j 20` changes into the `build` directory and runs 20 build commands in parallel. (Note that Ninja defaults to running commands in parallel anyway, so typically you don't need to pass `-j`.)

Environment variables

Ninja supports one environment variable to control its behavior: `NINJA_STATUS`, the progress status printed before the rule being run.

Several placeholders are available:

`%s`
: The number of started edges.

`%t`
: The total number of edges that must be run to complete the build.

`%p`
: The percentage of started edges.

`%r`
: The number of currently running edges.

`%u`
: The number of remaining edges to start.

`%f`
: The number of finished edges.

`%o`
: Overall rate of finished edges per second

`%c`
: Current rate of finished edges per second (average over builds specified by `-j` or its default)

`%e`
: Elapsed time in seconds. *(Available since Ninja 1.2.)*

`%%`
: A plain `%` character.

The default progress status is `"[%f/%t] "` (note the trailing space to separate from the build rule). Another example of possible progress status could be `"[%u/%r/%f] "`.

Extra tools

The `-t` flag on the Ninja command line runs some tools that we have found useful during Ninja's development. The current tools are:

`query`
: dump the inputs and outputs of a given target.

`browse`
: browse the dependency graph in a web browser. Clicking a file focuses the view on that file, showing inputs and outputs. This feature requires a Python installation. By default port 8000 is used and a web browser will be opened. This can be changed as follows:

```
ninja -t browse --port=8000 --no-browser mytarget
```

`graph`
: output a file in the syntax used by `graphviz`, a automatic graph layout tool. Use it like:

```
ninja -t graph mytarget | dot -Tpng -ograph.png
```

In the Ninja source tree, `ninja graph.png` generates an image for Ninja itself. If no target is given generate a graph for all root targets.

`targets`
: output a list of targets either by rule or by depth. If used like `ninja -t targets rule` *`name`* it prints the list of targets using the given

rule to be built. If no rule is given, it prints the source files (the leaves of the graph). If used like `ninja -t targets depth *digit*` it prints the list of targets in a depth-first manner starting by the root targets (the ones with no outputs). Indentation is used to mark dependencies. If the depth is zero it prints all targets. If no arguments are provided `ninja -t targets depth 1` is assumed. In this mode targets may be listed several times. If used like this `ninja -t targets all` it prints all the targets available without indentation and it is faster than the *depth* mode.

`commands` given a list of targets, print a list of commands which, if executed in order, may be used to rebuild those targets, assuming that all output files are out of date.

`clean` remove built files. By default it removes all built files except for those created by the generator. Adding the `-g` flag also removes built files created by the generator (see the rule reference for the `generator` attribute). Additional arguments are targets, which removes the given targets and recursively all files built for them.

If used like `ninja -t clean -r *rules*` it removes all files built using the given rules.

Files created but not referenced in the graph are not removed. This tool takes in account the `-v` and the `-n` options (note that `-n` implies `-v`).

`cleandead` remove files produced by previous builds that are no longer in the build file. *Available since Ninja 1.10.*

`compdb` given a list of rules, each of which is expected to be a C family language compiler rule whose first input is the name of the source file, prints on standard output a compilation database in the JSON format expected by the Clang tooling interface. *Available since Ninja 1.2.*

`deps` show all dependencies stored in the `.ninja_deps` file. When given a target, show just the target's dependencies. *Available since Ninja 1.4.*

`recompact` recompact the `.ninja_deps` file. *Available since Ninja 1.4.*

`restat` updates all recorded file modification timestamps in the `.ninja_log`

file. *Available since Ninja 1.10.*

`rules` output the list of all rules (eventually with their description if they have one). It can be used to know which rule name to pass to `ninja -t targets rule name` or `ninja -t compdb`.

Writing your own Ninja files

The remainder of this manual is only useful if you are constructing Ninja files yourself: for example, if you're writing a meta-build system or supporting a new language.

Conceptual overview

Ninja evaluates a graph of dependencies between files, and runs whichever commands are necessary to make your build target up to date as determined by file modification times. If you are familiar with Make, Ninja is very similar.

A build file (default name: `build.ninja`) provides a list of *rules* — short names for longer commands, like how to run the compiler — along with a list of *build* statements saying how to build files using the rules — which rule to apply to which inputs to produce which outputs.

Conceptually, `build` statements describe the dependency graph of your project, while `rule` statements describe how to generate the files along a given edge of the graph.

Syntax example

Here's a basic `.ninja` file that demonstrates most of the syntax. It will be used as an example for the following sections.

```
cflags = -Wall

rule cc
  command = gcc $cflags -c $in -o $out

build foo.o: cc foo.c
```

Variables

Despite the non-goal of being convenient to write by hand, to keep build files readable (debuggable), Ninja supports declaring shorter reusable names for strings. A declaration like the following

```
cflags = -g
```

can be used on the right side of an equals sign, dereferencing it with a dollar sign, like this:

```
rule cc
  command = gcc $cflags -c $in -o $out
```

Variables can also be referenced using curly braces like `${in}`.

Variables might better be called "bindings", in that a given variable cannot be changed, only shadowed. There is more on how shadowing works later in this document.

Rules

Rules declare a short name for a command line. They begin with a line consisting of the `rule` keyword and a name for the rule. Then follows an indented set of `variable = value` lines.

The basic example above declares a new rule named `cc`, along with the command to run. In the context of a rule, the `command` variable defines the command to run, `$in` expands to the list of input files (`foo.c`), and `$out` to the output files (`foo.o`) for the command. A full list of special variables is provided in the reference.

Build statements

Build statements declare a relationship between input and output files. They begin with the `build` keyword, and have the format `build` *`outputs`*`:` *`rulename`* *`inputs`*. Such a declaration says that all of the output files are derived from the input files. When the output files are missing or when the inputs change, Ninja will run the rule to regenerate the outputs.

The basic example above describes how to build `foo.o`, using the `cc` rule.

In the scope of a `build` block (including in the evaluation of its associated `rule`), the variable `$in` is the list of inputs and the variable `$out` is the list of outputs.

A build statement may be followed by an indented set of `key = value` pairs, much like a rule. These variables will shadow any variables when evaluating the variables in the command. For example:

```
cflags = -Wall -Werror
rule cc
  command = gcc $cflags -c $in -o $out

# If left unspecified, builds get the outer $cflags.
build foo.o: cc foo.c

# But you can shadow variables like cflags for a particular build.
build special.o: cc special.c
  cflags = -Wall

# The variable was only shadowed for the scope of special.o;
# Subsequent build lines get the outer (original) cflags.
build bar.o: cc bar.c
```

For more discussion of how scoping works, consult the reference.

If you need more complicated information passed from the build statement to the rule (for example, if the rule needs "the file extension of the first input"), pass that through as an extra variable, like how `cflags` is passed above.

If the top-level Ninja file is specified as an output of any build statement and it is out of date, Ninja will rebuild and reload it before building the targets requested by the user.

Generating Ninja files from code

`misc/ninja_syntax.py` in the Ninja distribution is a tiny Python module to facilitate generating Ninja files. It allows you to make Python calls like `ninja.rule(name='foo', command='bar', depfile='$out.d')` and it will generate the appropriate syntax. Feel free to just inline it into your project's build system if it's useful.

More details

The `phony` rule

The special rule name `phony` can be used to create aliases for other targets. For example:

```
build foo: phony some/file/in/a/faraway/subdir/foo
```

This makes `ninja foo` build the longer path. Semantically, the `phony` rule is equivalent to a plain rule where the `command` does nothing, but phony rules are handled specially in that they aren't printed when run, logged (see below), nor do they contribute to the command count printed as part of the build process.

`phony` can also be used to create dummy targets for files which may not exist at build time. If a phony build statement is written without any dependencies, the target will be considered out of date if it does not exist. Without a phony build statement, Ninja will report an error if the file does not exist and is required by the build.

To create a rule that never rebuilds, use a build rule without any input:

```
rule touch
  command = touch $out
build file_that_always_exists.dummy: touch
build dummy_target_to_follow_a_pattern: phony file_that_always_exists.dummy
```

Default target statements

By default, if no targets are specified on the command line, Ninja will build every output that is not named as an input elsewhere. You can override this behavior using a default target statement. A default target statement causes Ninja to build only a given subset of output files if none are specified on the command line.

Default target statements begin with the `default` keyword, and have the format `default` *`targets`*. A default target statement must appear after the build statement that declares the target as an output file. They are cumulative, so multiple statements may be used to extend the list of default targets. For example:

```
default foo bar
default baz
```

This causes Ninja to build the `foo`, `bar` and `baz` targets by default.

The Ninja log

For each built file, Ninja keeps a log of the command used to build it. Using this log Ninja can know when an existing output was built with a different command line than the build files specify (i.e., the command line changed) and knows to rebuild the file.

The log file is kept in the build root in a file called `.ninja_log`. If you provide a variable named `builddir` in the outermost scope, `.ninja_log` will be kept in that directory instead.

Version compatibility

Available since Ninja 1.2.

Ninja version labels follow the standard major.minor.patch format, where the major version is increased on backwards-incompatible syntax/behavioral changes and the minor version is increased on new behaviors. Your `build.ninja` may declare a variable named `ninja_required_version` that asserts the minimum Ninja version required to use the generated file. For example,

```
ninja_required_version = 1.1
```

declares that the build file relies on some feature that was introduced in Ninja 1.1 (perhaps the `pool` syntax), and that Ninja 1.1 or greater must be used to build. Unlike other Ninja variables, this version requirement is checked immediately when the variable is encountered in parsing, so it's best to put it at the top of the build file.

Ninja always warns if the major versions of Ninja and the `ninja_required_version` don't match; a major version change hasn't come up yet so it's difficult to predict what behavior might be required.

C/C++ header dependencies

To get C/C++ header dependencies (or any other build dependency that works in a similar way) correct Ninja has some extra functionality.

The problem with headers is that the full list of files that a given source file depends on can only be discovered by the compiler: different preprocessor defines and

include paths cause different files to be used. Some compilers can emit this information while building, and Ninja can use that to get its dependencies perfect.

Consider: if the file has never been compiled, it must be built anyway, generating the header dependencies as a side effect. If any file is later modified (even in a way that changes which headers it depends on) the modification will cause a rebuild as well, keeping the dependencies up to date.

When loading these special dependencies, Ninja implicitly adds extra build edges such that it is not an error if the listed dependency is missing. This allows you to delete a header file and rebuild without the build aborting due to a missing input.

depfile

`gcc` (and other compilers like `clang`) support emitting dependency information in the syntax of a Makefile. (Any command that can write dependencies in this form can be used, not just `gcc`.)

To bring this information into Ninja requires cooperation. On the Ninja side, the `depfile` attribute on the `build` must point to a path where this data is written. (Ninja only supports the limited subset of the Makefile syntax emitted by compilers.) Then the command must know to write dependencies into the `depfile` path. Use it like in the following example:

```
rule cc
  depfile = $out.d
  command = gcc -MD -MF $out.d [other gcc flags here]
```

The `-MD` flag to `gcc` tells it to output header dependencies, and the `-MF` flag tells it where to write them.

deps

(Available since Ninja 1.3.)

It turns out that for large projects (and particularly on Windows, where the file system is slow) loading these dependency files on startup is slow.

Ninja 1.3 can instead process dependencies just after they're generated and save a compacted form of the same information in a Ninja-internal database.

Ninja supports this processing in two forms.

1. `deps = gcc` specifies that the tool outputs `gcc`-style dependencies in the form of Makefiles. Adding this to the above example will cause Ninja to

process the `depfile` immediately after the compilation finishes, then delete the `.d` file (which is only used as a temporary).

2. `deps = msvc` specifies that the tool outputs header dependencies in the form produced by Visual Studio's compiler's `/showIncludes` flag. Briefly, this means the tool outputs specially-formatted lines to its stdout. Ninja then filters these lines from the displayed output. No `depfile` attribute is necessary, but the localized string in front of the the header file path. For instance `msvc_deps_prefix = Note: including file:` for a English Visual Studio (the default). Should be globally defined.

```
msvc_deps_prefix = Note: including file:
rule cc
  deps = msvc
  command = cl /showIncludes -c $in /Fo$out
```

If the include directory directives are using absolute paths, your depfile may result in a mixture of relative and absolute paths. Paths used by other build rules need to match exactly. Therefore, it is recommended to use relative paths in these cases.

Pools

Available since Ninja 1.1.

Pools allow you to allocate one or more rules or edges a finite number of concurrent jobs which is more tightly restricted than the default parallelism.

This can be useful, for example, to restrict a particular expensive rule (like link steps for huge executables), or to restrict particular build statements which you know perform poorly when run concurrently.

Each pool has a `depth` variable which is specified in the build file. The pool is then referred to with the `pool` variable on either a rule or a build statement.

No matter what pools you specify, ninja will never run more concurrent jobs than the default parallelism, or the number of jobs specified on the command line (with `-j`).

```
# No more than 4 links at a time.
pool link_pool
  depth = 4

# No more than 1 heavy object at a time.
pool heavy_object_pool
  depth = 1
```

```
rule link
  ...
  pool = link_pool

rule cc
  ...

# The link_pool is used here. Only 4 links will run concurrently.
build foo.exe: link input.obj

# A build statement can be exempted from its rule's pool by setting an
# empty pool. This effectively puts the build statement back into the default
# pool, which has infinite depth.
build other.exe: link input.obj
  pool =

# A build statement can specify a pool directly.
# Only one of these builds will run at a time.
build heavy_object1.obj: cc heavy_obj1.cc
  pool = heavy_object_pool
build heavy_object2.obj: cc heavy_obj2.cc
  pool = heavy_object_pool
```

The `console` pool

Available since Ninja 1.5.

There exists a pre-defined pool named `console` with a depth of 1. It has the special property that any task in the pool has direct access to the standard input, output and error streams provided to Ninja, which are normally connected to the user's console (hence the name) but could be redirected. This can be useful for interactive tasks or long-running tasks which produce status updates on the console (such as test suites).

While a task in the `console` pool is running, Ninja's regular output (such as progress status and output from concurrent tasks) is buffered until it completes.

Ninja file reference

A file is a series of declarations. A declaration can be one of:

1. A rule declaration, which begins with `rule rulename`, and then has a series of indented lines defining variables.
2. A build edge, which looks like `build output1 output2: rulename input1 input2`. Implicit dependencies may be tacked on the end with `| dependency1 dependency2`. Order-only dependencies may be tacked on the end with `|| dependency1 dependency2`. (See the reference on dependency types.)

   Implicit outputs *(available since Ninja 1.7)* may be added before the `:` with `| output1 output2` and do not appear in `$out`. (See the reference on output types.)

3. Variable declarations, which look like `variable = value`.
4. Default target statements, which look like `default target1 target2`.
5. References to more files, which look like `subninja path` or `include path`. The difference between these is explained below in the discussion about scoping.
6. A pool declaration, which looks like `pool poolname`. Pools are explained in the section on pools.

Lexical syntax

Ninja is mostly encoding agnostic, as long as the bytes Ninja cares about (like slashes in paths) are ASCII. This means e.g. UTF-8 or ISO-8859-1 input files ought to work.

Comments begin with `#` and extend to the end of the line.

Newlines are significant. Statements like `build foo bar` are a set of space-separated tokens that end at the newline. Newlines and spaces within a token must be escaped.

There is only one escape character, `$`, and it has the following behaviors:

`$` followed by a newline
: escape the newline (continue the current line across a line break).

`$` followed by text
: a variable reference.

`${varname}`
: alternate syntax for `$varname`.

`$` followed by space
: a space. (This is only necessary in lists of paths, where a space would otherwise separate filenames. See below.)

`$:`
: a colon. (This is only necessary in `build` lines, where a colon would otherwise terminate the list of outputs.)

`$$`
: a literal `$`.

A `build` or `default` statement is first parsed as a space-separated list of filenames and then each name is expanded. This means that spaces within a variable will result in spaces in the expanded filename.

```
spaced = foo bar
build $spaced/baz other$ file: ...
# The above build line has two outputs: "foo bar/baz" and "other file".
```

In a `name = value` statement, whitespace at the beginning of a value is always stripped. Whitespace at the beginning of a line after a line continuation is also stripped.

```
two_words_with_one_space = foo $
    bar
one_word_with_no_space = foo$
    bar
```

Other whitespace is only significant if it's at the beginning of a line. If a line is indented more than the previous one, it's considered part of its parent's scope; if it is indented less than the previous one, it closes the previous scope.

Top-level variables

Two variables are significant when declared in the outermost file scope.

`builddir`
: a directory for some Ninja output files. See the discussion of the build log. (You can also store other build output in this directory.)

`ninja_required_version`
: the minimum version of Ninja required to process the build correctly. See the discussion of versioning.

Rule variables

A `rule` block contains a list of `key = value` declarations that affect the processing of the rule. Here is a full list of special keys.

`command` (*required*)
: the command line to run. Each `rule` may have only one `command` declaration. See the next section for more details on quoting and executing multiple commands.

`depfile`
: path to an optional `Makefile` that contains extra *implicit dependencies* (see the reference on dependency types). This is explicitly to support C/C++ header dependencies; see the full discussion.

`deps`
: *(Available since Ninja 1.3.)* if present, must be one of `gcc` or `msvc` to specify special dependency processing. See the full discussion. The generated database is stored as `.ninja_deps` in the `builddir`, see the discussion of `builddir`.

`msvc_deps_prefix`
: *(Available since Ninja 1.5.)* defines the string which should be stripped from msvc's /showIncludes output. Only needed when `deps = msvc` and no English Visual Studio version is used.

`description`
: a short description of the command, used to pretty-print the command as it's running. The `-v` flag controls whether to print the full command or its description; if a command fails, the full command line will always be printed before the command's output.

`dyndep`
: *(Available since Ninja 1.10.)* Used only on build statements. If present, must name one of the build statement inputs. Dynamically discovered dependency information will be loaded from the file. See the dynamic dependencies section for details.

`generator`
: if present, specifies that this rule is used to re-invoke the generator program. Files built using `generator` rules are treated specially in two ways: firstly, they will not be rebuilt if the command line changes; and secondly, they are not cleaned by default.

`in`
: the space-separated list of files provided as inputs to the build line referencing this `rule`, shell-quoted if it appears in commands. (`$in` is provided solely for convenience; if you need some subset or variant of this list of files, just construct a new variable with that list and use that instead.)

`in_newline`
: the same as `$in` except that multiple inputs are separated by newlines rather than spaces. (For use with `$rspfile_content`; this works around a bug in the MSVC linker where it uses a fixed-size buffer for processing input.)

`out`
: the space-separated list of files provided as outputs to the build line referencing this `rule`, shell-quoted if it appears in commands.

`restat`
: if present, causes Ninja to re-stat the command's outputs after execution of the command. Each output whose modification time the command did not change will be treated as though it had never needed to be built. This may cause the output's reverse dependencies to be removed from the list of pending build actions.

`rspfile`, `rspfile_content`
: if present (both), Ninja will use a response file for the given command, i.e. write the selected string (`rspfile_content`) to the given file (`rspfile`) before calling the command and delete the file after successful execution of the command.

  This is particularly useful on Windows OS, where the maximal length of a command line is limited and response files must be used instead.

  Use it like in the following example:

```
rule link
  command = link.exe /OUT$out [usual link flags here] @$out.rsp
  rspfile = $out.rsp
  rspfile_content = $in

build myapp.exe: link a.obj b.obj [possibly many other .obj files]
```

Interpretation of the `command` variable

Fundamentally, command lines behave differently on Unixes and Windows.

On Unixes, commands are arrays of arguments. The Ninja `command` variable is passed directly to `sh -c`, which is then responsible for interpreting that string into an argv array. Therefore the quoting rules are those of the shell, and you can use all the normal shell operators, like `&&` to chain multiple commands, or `VAR=value cmd` to set environment variables.

On Windows, commands are strings, so Ninja passes the `command` string directly to `CreateProcess`. (In the common case of simply executing a compiler this means there is less overhead.) Consequently the quoting rules are determined by the called program, which on Windows are usually provided by the C library. If you need shell interpretation of the command (such as the use of `&&` to chain multiple commands), make the command execute the Windows shell by prefixing the

command with `cmd /c`. Ninja may error with "invalid parameter" which usually indicates that the command line length has been exceeded.

Build outputs

There are two types of build outputs which are subtly different.

1. *Explicit outputs*, as listed in a build line. These are available as the `$out` variable in the rule.

   This is the standard form of output to be used for e.g. the object file of a compile command.

2. *Implicit outputs*, as listed in a build line with the syntax `| out1 out2` + before the `:` of a build line *(available since Ninja 1.7)*. The semantics are identical to explicit outputs, the only difference is that implicit outputs don't show up in the `$out` variable.

   This is for expressing outputs that don't show up on the command line of the command.

Build dependencies

There are three types of build dependencies which are subtly different.

1. *Explicit dependencies*, as listed in a build line. These are available as the `$in` variable in the rule. Changes in these files cause the output to be rebuilt; if these files are missing and Ninja doesn't know how to build them, the build is aborted.

   This is the standard form of dependency to be used e.g. for the source file of a compile command.

2. *Implicit dependencies*, either as picked up from a `depfile` attribute on a rule or from the syntax `| dep1 dep2` on the end of a build line. The semantics are identical to explicit dependencies, the only difference is that implicit dependencies don't show up in the `$in` variable.

   This is for expressing dependencies that don't show up on the command line of the command; for example, for a rule that runs a script, the script itself should be an implicit dependency, as changes to the script should cause the output to rebuild.

Note that dependencies as loaded through depfiles have slightly different semantics, as described in the rule reference.

3. *Order-only dependencies*, expressed with the syntax `|| dep1 dep2` on the end of a build line. When these are out of date, the output is not rebuilt until they are built, but changes in order-only dependencies alone do not cause the output to be rebuilt.

   Order-only dependencies can be useful for bootstrapping dependencies that are only discovered during build time: for example, to generate a header file before starting a subsequent compilation step. (Once the header is used in compilation, a generated dependency file will then express the implicit dependency.)

File paths are compared as is, which means that an absolute path and a relative path, pointing to the same file, are considered different by Ninja.

Variable expansion

Variables are expanded in paths (in a `build` or `default` statement) and on the right side of a `name = value` statement.

When a `name = value` statement is evaluated, its right-hand side is expanded immediately (according to the below scoping rules), and from then on `$name` expands to the static string as the result of the expansion. It is never the case that you'll need to "double-escape" a value to prevent it from getting expanded twice.

All variables are expanded immediately as they're encountered in parsing, with one important exception: variables in `rule` blocks are expanded when the rule is *used*, not when it is declared. In the following example, the `demo` rule prints "this is a demo of bar".

```
rule demo
  command = echo "this is a demo of $foo"

build out: demo
  foo = bar
```

Evaluation and scoping

Top-level variable declarations are scoped to the file they occur in.

Rule declarations are also scoped to the file they occur in. *(Available since Ninja 1.6)*

The `subninja` keyword, used to include another `.ninja` file, introduces a new scope. The included `subninja` file may use the variables and rules from the parent file, and shadow their values for the file's scope, but it won't affect values of the variables in the parent.

To include another `.ninja` file in the current scope, much like a C `#include` statement, use `include` instead of `subninja`.

Variable declarations indented in a `build` block are scoped to the `build` block. The full lookup order for a variable expanded in a `build` block (or the `rule` is uses) is:

1. Special built-in variables (`$in`, `$out`).
2. Build-level variables from the `build` block.
3. Rule-level variables from the `rule` block (i.e. `$command`). (Note from the above discussion on expansion that these are expanded "late", and may make use of in-scope bindings like `$in`.)
4. File-level variables from the file that the `build` line was in.
5. Variables from the file that included that file using the `subninja` keyword.

Dynamic Dependencies

Available since Ninja 1.10.

Some use cases require implicit dependency information to be dynamically discovered from source file content *during the build* in order to build correctly on the first run (e.g. Fortran module dependencies). This is unlike header dependencies which are only needed on the second run and later to rebuild correctly. A build statement may have a `dyndep` binding naming one of its inputs to specify that dynamic dependency information must be loaded from the file. For example:

```
build out: ... || foo
  dyndep = foo
build foo: ...
```

This specifies that file `foo` is a dyndep file. Since it is an input, the build statement for `out` can never be executed before `foo` is built. As soon as `foo` is finished Ninja

will read it to load dynamically discovered dependency information for `out`. This may include additional implicit inputs and/or outputs. Ninja will update the build graph accordingly and the build will proceed as if the information was known originally.

Dyndep file reference

Files specified by `dyndep` bindings use the same lexical syntax as ninja build files and have the following layout.

1. A version number in the form `<major>[.<minor>][<suffix>]`:

   ```
   ninja_dyndep_version = 1
   ```

   Currently the version number must always be `1` or `1.0` but may have an arbitrary suffix.

2. One or more build statements of the form:

   ```
   build out | imp-outs... : dyndep | imp-ins...
   ```

   Every statement must specify exactly one explicit output and must use the rule name `dyndep`. The `| imp-outs...` and `| imp-ins...` portions are optional.

3. An optional `restat` variable binding on each build statement.

The build statements in a dyndep file must have a one-to-one correspondence to build statements in the ninja build file that name the dyndep file in a `dyndep` binding. No dyndep build statement may be omitted and no extra build statements may be specified.

Dyndep Examples

Fortran Modules

Consider a Fortran source file `foo.f90` that provides a module `foo.mod` (an implicit output of compilation) and another source file `bar.f90` that uses the module (an implicit input of compilation). This implicit dependency must be discovered before we compile either source in order to ensure that `bar.f90` never compiles before `foo.f90`, and that `bar.f90` recompiles when `foo.mod` changes. We can achieve this as follows:

```
rule f95
  command = f95 -o $out -c $in
rule fscan
  command = fscan -o $out $in

build foobar.dd: fscan foo.f90 bar.f90

build foo.o: f95 foo.f90 || foobar.dd
  dyndep = foobar.dd
build bar.o: f95 bar.f90 || foobar.dd
  dyndep = foobar.dd
```

In this example the order-only dependencies ensure that `foobar.dd` is generated before either source compiles. The hypothetical `fscan` tool scans the source files, assumes each will be compiled to a `.o` of the same name, and writes `foobar.dd` with content such as:

```
ninja_dyndep_version = 1
build foo.o | foo.mod: dyndep
build bar.o: dyndep |  foo.mod
```

Ninja will load this file to add `foo.mod` as an implicit output of `foo.o` and implicit input of `bar.o`. This ensures that the Fortran sources are always compiled in the proper order and recompiled when needed.

Tarball Extraction

Consider a tarball `foo.tar` that we want to extract. The extraction time can be recorded with a `foo.tar.stamp` file so that extraction repeats if the tarball changes, but we also would like to re-extract if any of the outputs is missing. However, the list of outputs depends on the content of the tarball and cannot be spelled out explicitly in the ninja build file. We can achieve this as follows:

```
rule untar
  command = tar xf $in && touch $out
rule scantar
  command = scantar --stamp=$stamp --dd=$out $in
build foo.tar.dd: scantar foo.tar
  stamp = foo.tar.stamp
build foo.tar.stamp: untar foo.tar || foo.tar.dd
  dyndep = foo.tar.dd
```

In this example the order-only dependency ensures that `foo.tar.dd` is built before the tarball extracts. The hypothetical `scantar` tool will read the tarball (e.g.

via `tar tf`) and write `foo.tar.dd` with content such as:

```
ninja_dyndep_version = 1
build foo.tar.stamp | file1.txt file2.txt : dyndep
  restat = 1
```

Ninja will load this file to add `file1.txt` and `file2.txt` as implicit outputs of `foo.tar.stamp`, and to mark the build statement for `restat`. On future builds, if any implicit output is missing the tarball will be extracted again. The `restat` binding tells Ninja to tolerate the fact that the implicit outputs may not have modification times newer than the tarball itself (avoiding re-extraction on every build).