Proposed GN “metadata” features

juliehockett@google.com, mcgrathr@google.com

Proposal note

This proposal was forked from @brettw’s Proposed GN "tags" feature. This iteration attempts to clarify and extend the feature proposed there based on feedback from the mailing list discussion and to provide concrete examples of the motivating use cases.

Background

A principle of the GN expression language is to isolate the information accessible to any given buildfile. Code in a file can’t interrogate the build graph directly because there’s no explicit ordering to the loading of the build files, and therefore no guarantee about the state of dependency graph during the file load. A key realization to note here is that the `gen` command is a two-phase operation: the first phase involves evaluating the GN expression language to produce the build graph, while the second phase is when the Ninja files are actually generated. The second phase by nature sees the whole build graph, and so there is not a barrier to doing more with that knowledge (with the caveat that the ‘more’ doesn’t affect the build graph itself in any way).

We currently have a few ways that these use cases get around this restriction. GN’s iOS support involves a custom GN feature with two custom built-in target types: `bundle_data` and `create_bundle`, which collect and pass an explicit dependency list. The Chrome Android build created an elaborate system that includes having scripts write dependencies to files that can be interrogated at build-time to reconstruct some dependency information. In Fuchsia, there is a packaging system that has similar requirements and a convoluted implementation.

Motivation

The motivating use case for this proposal is the Fuchsia archive packaging system. In building the system image, the build system must generate an archive based on the set of built targets. GN is currently quite good at building a collection of individual targets, but cannot collect those targets back and emit them to a single archive. This step could be performed outside of the build system, but the introspective view GN has on the dependency graph and the build itself would be useful for knowing when it is necessary to rebuild and repackage the archive.

The example given by @abarth on the mailing list is as follows:

You might want to include the `ls` program in your disk image. Somewhere in gn there's an executable() target that builds `ls`. For shared libraries, we usually only want to include a shared library in the disk image if there's actually a program that depends on that shared library. For example, suppose we had a `shared_library("skia")` target in the build. If we happened to assemble a disk image that had an executable that depended on libskia.so, then we'd want to include `libskia.so` in the disk image. If not, then we wouldn't want to include it.

The proposed metadata feature has gone through several iterations, and this is an attempt to consolidate and discuss them as we now have a more concrete knowledge of how the Fuchsia build in particular would use the feature.

Proposal

This proposal is for a way to attach metadata (like file names) to GN targets and to propagate it across the dependency tree. This data can be collected and passed to action targets that can consume the data and execute the corresponding packaging step.

The goal of this proposal is to add a minimal set of features required to meet the packaging requirements described above. It is specified in a very flexible way so that we can extend the functionality over time and eventually begin to converge the different packaging requirements for different systems.

The main disadvantage is that the generality of the feature makes it more difficult to understand than most GN features. However, it is easier to understand than the custom bundle systems that have been created for iOS, Android, and Fuchsia. The feature is intended to be driven primarily by templates that provide a natural API for the metadata and for consumption.

Alternatives considered

- A similar proposal is to follow the iOS bundle example and create separate target types for declaring and consuming metadata. It is theoretically equivalent to the metadata proposal but comes down to an aesthetic question of whether it's preferable to have more targets and target types or more target variables.

`metadata` target variable

A variable of type scope containing opaque key/value pairs that can be passed up the dependency tree. The values must always be a list, as this provides an interface for collecting all values of a single key in a simple operation with clear and intuitive results. This constraint also simplifies the extraction function below, as no additional details on

flattening or combination logic need be specified.

Metadata values are not interpreted by GN, but can be read by the `get_metadata()` function (see below).

Defining metadata

Metadata is defined for a particular target in the `metadata` target variable. This variable should be a scope that maps metadata keys to list values:

```
source_set("foo") {
  metadata = {
    archive_inputs = [
        "file1.cc",
        "file2.cc",
      ]
  }
}
```

This example sets a metadata field on the target: `archive_inputs`, containing a list of filenames. Metadata values can only be lists of any value type, but generally will be lists of strings or paths.

Example

This example defines a shared library that adds a `bundle_files` metadata value:

```
lib_with_data("mylib") {
  sources = [ "foo.cc" ]
  deps = [ ":bar" ]
  metadata = {
    bundle_files = [ "mydata1.dat" ]
  }
}
```

`metaresult` value type

A type of Value containing an opaque collection of metadata.

This Value type introduces a way of accessing and using metadata in the GN expression language that separates it from the existing notion of lists. This type of value is only returned from the `get_metadata()` function, and can only be used in the places described below (see Valid Uses). The value will be opaque at parse time, storing only the parameters

of the `get_metadata()` function. It will not expose the ability to manipulate or edit those contained values. All `metaresult` values are by definition unequal, as the parser and expression language will not be able to interpret them literally at parse time.

The value will be expanded after the build graph is constructed, walking the tree as specified in the function (see Walking the dependency tree below). The expansion site can then determine what to do with the data during the second phase. If the expansion site expects a single file or a list of files , the value will attempt to `rebase_path()` the value onto the relative source path of the target as the function walks the graph. If any value in the `metaresult` is expected to be a string and isn't, the value will raise an error to the user.

The critical element of the metadata implementation is that a user will be able to statically express a need for dynamic data. Thus, this type is an opaque, list-like collection. Metadata from different targets can be gathered by the `get_metadata()` function into a `metaresult` after the build graph is complete, with the expectation that the user cannot edit or otherwise manipulate the data or define an instance of it.

Valid uses

As this is a fairly powerful feature, there must be some limitations on where it is acceptable to access it. It can be used anywhere it does not modify the dependency tree; that is, any place you would put a value that only ninja will read (e.g. toolchain definitions, argument lists, response file contents). It could also be used in file lists that only affect ninja dependencies, including the sources list of a compiling target or `action_foreach`, as those affect how many ninja targets are generated but do not introduce any ordering or dependency changes into GN.

The semantics of the `metaresult` depend on the context in which it is used; it could be used in a singleton context or a list context.

Use in a singleton context (e.g. an action's `script`) would require one and only one element to be present in the `metaresult`, and would give an error during the generation if zero or more than one element were present on expansion. Use in the list context would indicate that though it is one element at definition, it would expand to zero or more elements after the build graph is traversed.

It could also be used as the the parameter containing the data to be written in the `write_file()` function. This would require delayed `write_file()` semantics, which is a separate proposal[1].

[1] GN write_file() semantics design doc here.

`get_metadata()` function

A function to trigger the collection of metadata from the specified target(s) into a `metaresult`.

GN build scripts cannot query the dependency tree programmatically because the load order of targets is not defined. This function provides a way to pass data from direct and indirect dependencies to a script at build-time. A call returns a single `metaresult` value containing the parameters of the collection at parse time, and is expanded to a value consisting of the concatenation of all metadata values for the given keys in the set of defined walkable targets.

The order of the data in the result will be that of a depth-first-search of the build graph. The walk visits each node exactly once, so a graph with multiple paths to the same target will only ever collect that target's metadata once. The traversal will mimic the existing traversal order used elsewhere in GN (e.g. collecting link inputs) to begin, but we may want to iterate on the exact specification of the ordering.

Parameters

`get_metadata(label_list, data_key_list, walk_key_list=[“”])` is defined to take three parameters:

- `label_list` is the list of targets to gather metadata on. It is a list of target labels, which will be used by the function as starting points for the collection.
- `data_key_list` is the list of metadata keys from which to collect data.
- `walk_key_list` is the list of metadata keys whose values are lists of dependencies to recurse into. See below for details. By default, the function gathers data from all `deps` and `data_deps` (see Walking the dependency tree below for ways to restrict this).
- walk_key_list represents.

Example

```
source_set("foo") {
  metadata = {
    inputs = [
        "foo1.cc",
        "foo2.cc",
      ]
  }
}
```

```
source_set("bar") {
  metadata = {
    inputs = [
        "bar1.cc",
        "bar2.cc",
      ]
  }
}

shared_library("foo_lib") {
  metadata = {
    inputs = [ "foo_lib.cc" ]
  }

  deps = [ "//foo", "//bar" ]
}
```

The above snippet defines three targets, each containing metadata. The `get_metadata()` call in the last line will walk the dependency tree to gather the metadata values in the `inputs` field, beginning with `//foo_lib`. It will then recurse into `//foo` and `//bar`, gathering their values. The results of this example will be:

```
get_metadata(["//foo_lib"], ["inputs"]) → metaresult[
  "foo_lib.cc",
  "foo1.cc",
  "foo2.cc",
  "bar1.cc",
  "bar2.cc"
]
```

Walking the dependency tree

The `walk_key_list` identifies an explicit list of dependencies recurse into. There will be many cases where we don't want the metadata to include everything from all dependencies. Thus, the `walk_key_list` provides a way to limit the recursion and define categories of dependencies.

The expected behavior of this list is to provide a barrier to the recursive walk. The function will check the metadata scope for any specified walk keys containing a list of zero or more labels and modify its graph traversal based on those labels. If the function finds one of these keys, it will recurse into the targets listed.

Placing an empty string element in the `walk_key_list` indicates that the walk should go to all `deps` and `data_deps` in its traversal, in addition to any targets listed in the specified walk

keys. The function provides `[“”]` as the default value, so that the default behavior is to walk all `deps` and `data_deps` of the specified targets, recursively.

Example

```
source_set("foo") {
  metadata = {
    inputs = [
        "foo1.cc",
        "foo2.cc",
      ]
    include = [ ]
  }
  deps = [ "//bar" ]
}

source_set("bar") {
  metadata = {
    inputs = [
        "bar1.cc",
        "bar2.cc",
      ]
  }
}

shared_library("foo_lib") {
  metadata = {
    inputs = [ "foo_lib.cc" ]
  }
  deps = [ "//foo" ]
}
```

The above snippet defines the same targets as the previous example, with the addition of the `include` key on the `//foo` target. The `get_metadata()` call will now walk the same dependency tree as before, but will check each target’s metadata for the presence of the `include` key. As this key is set on `//foo`, it knows to not recurse into any target, and will ignore the `//bar` dependency. The results of this example will be:

```
get_metadata(["//foo_lib"], ["inputs"], ["include"]) → metaresult[
  "foo_lib.cc",
  "foo1.cc",
  "foo2.cc",
]
```

This barrier solution addresses some of the concerns of previous iterations of this proposal, in that it allows for more complex barrier logic than a simple feature. For

example, in the situation where an action is defined that includes a tool and targets that produce inputs to that tool:

```
executable("tool") {
  metadata = {
    args = [ "tool.cc" ]
  }
  deps = [ ...tool_deps... ]
}

source_set("input") {
  metadata = {
    args = [ "input.cc" ]
  }
  deps = [ ...input_deps... ]
}

action("run_tool") {
  metadata = {
    stop = [ "//input" ]
  }
  deps = [ "//tool", "//input" ]
}
```

The metadata walk may only want to list the inputs' deps, and not the tool's. In this case, we would use the `stop` key as our walk key, which yields the result:

```
get_metadata(["//run_tool", ["args"], ["stop"]) → metaresult[
  "input.cc",
]
```