The MB (Meta-Build wrapper) design spec

[TOC]

Intro

MB is intended to address two major aspects of the GYP -> GN transition for Chromium:

1. "bot toggling" - make it so that we can easily flip a given bot back and forth between GN and GYP.
2. "bot configuration" - provide a single source of truth for all of the different configurations (os/arch/`gyp_define` combinations) of Chromium that are supported.

MB must handle at least the `gen` and `analyze` steps on the bots, i.e., we need to wrap both the `gyp_chromium` invocation to generate the Ninja files, and the `analyze` step that takes a list of modified files and a list of targets to build and returns which targets are affected by the files.

For more information on how to actually use MB, see the user guide.

Design

MB is intended to be as simple as possible, and to defer as much work as possible to GN or GYP. It should live as a very simple Python wrapper that offers little in the way of surprises.

Command line

It is structured as a single binary that supports a list of subcommands:

- `mb gen -c linux_rel_bot //out/Release`
- `mb analyze -m tryserver.chromium.linux -b linux_rel /tmp/input.json /tmp/output.json`

Configurations

`mb` will first look for a bot config file in a set of different locations (initially just in //ios/build/bots). Bot config files are JSON files that contain keys for 'GYP_DEFINES' (a list of strings that will be joined together with spaces and passed to GYP, or a dict that will be similarly converted), 'gn_args' (a list of strings that will be joined together), and an 'mb_type' field that says whether to use GN or GYP. Bot config files require the full list of settings to be given explicitly.

If no matching bot config file is found, `mb` looks in the `//tools/mb/mb_config.pyl` config file to determine whether to use GYP or GN for a particular build directory, and what set of flags (`GYP_DEFINES` or `gn args`) to use.

A config can either be specified directly (useful for testing) or by specifying the master name and builder name (useful on the bots so that they do not need to specify a config directly and can be hidden from the details).

See the user guide for details.

Handling the analyze step

The interface to `mb analyze` is described in the user_guide.

The way analyze works can be subtle and complicated (see below).

Since the interface basically mirrors the way the "analyze" step on the bots invokes `gyp_chromium` today, when the config is found to be a gyp config, the arguments are passed straight through.

It implements the equivalent functionality in GN by calling `gn refs [list of files] --type=executable --all --as=output` and filtering the output to match the list of targets.

Analyze

The goal of the `analyze` step is to speed up the cycle time of the try servers by only building and running the tests affected by the files in a patch, rather than everything that might be out of date. Doing this ends up being tricky.

We start with the following requirements and observations:

- In an ideal (un-resource-constrained) world, we would build and test everything that a patch affected on every patch. This does not necessarily mean that we would build 'all' on every patch (see below).
- In the real world, however, we do not have an infinite number of machines, and try jobs are not infinitely fast, so we need to balance the desire to get maximum test coverage against the desire to have reasonable cycle times, given the number of machines we have.
- Also, since we run most try jobs against tip-of-tree Chromium, by the time one job completes on the bot, new patches have probably landed, rendering the build out of date.
- This means that the next try job may have to do a build that is out of date due to a combination of files affected by a given patch, and files affected for unrelated reasons. We want to rebuild and test only the targets affected by the patch, so that we don't blame or punish the patch author for unrelated changes.

So:

1. We need a way to indicate which changed files we care about and which we don't (the affected files of a patch).

2. We need to know which tests we might potentially want to run, and how those are mapped onto build targets. For some kinds of tests (like GTest-based tests), the mapping is 1:1 - if you want to run base_unittests, you need to build base_unittests. For others (like the telemetry and layout tests), you might need to build several executables in order to run the tests, and that mapping might best be captured by a *meta* target (a GN group or a GYP 'none' target like `webkit_tests`) that depends on the right list of files. Because the GN and GYP files know nothing about test steps, we have to have some way of mapping back and forth between test steps and build targets. That mapping is *not* currently available to MB (or GN or GYP), and so we have to enough information to make it possible for the caller to do the mapping.

3. We might also want to know when test targets are affected by data files that aren't compiled (python scripts, or the layout tests themselves). There's no good way to do this in GYP, but GN supports this.

4. We also want to ensure that particular targets still compile even if they are not actually tested; consider testing the installers themselves, or targets that don't yet have good test coverage. We might want to use meta targets for this purpose as well.

5. However, for some meta targets, we don't necessarily want to rebuild the meta target itself, perhaps just the dependencies of the meta target that are affected by the patch. For example, if you have a meta target like `blink_tests` that might depend on ten different test binaries. If a patch only affects one of them (say `wtf_unittests`), you don't want to build `blink_tests`, because that might actually also build the other nine targets. In other words, some meta targets are *prunable*.

6. As noted above, in the ideal case we actually have enough resources and things are fast enough that we can afford to build everything affected by a patch, but listing every possible target explicitly would be painful. The GYP and GN Ninja generators provide an 'all' target that captures (nearly, see crbug.com/503241) everything, but unfortunately neither GN nor GYP actually represents 'all' as a meta target in the build graph, so we will need to write code to handle that specially.

7. In some cases, we will not be able to correctly analyze the build graph to determine the impact of a patch, and need to bail out (e.g,. if you change a build file itself, it may not be easy to tell how that affects the graph). In that case we should simply build and run everything.

The interaction between 2) and 5) means that we need to treat meta targets two different ways, and so we need to know which targets should be pruned in the sense of 5) and which targets should be returned unchanged so that we can map them back to the appropriate tests.

So, we need three things as input:

- `files`: the list of files in the patch
- `test_targets`: the list of ninja targets which, if affected by a patch, should be reported back so that we can map them back to the appropriate tests to run. Any meta targets in this list should *not* be pruned.
- `additional_compile_targets`: the list of ninja targets we wish to compile *in addition to* the list in `test_targets`. Any meta targets present in this list should be pruned (we don't need to return the meta targets because they aren't mapped back to tests, and we don't want to build them because we might build too much).

We can then return two lists as output:

- `compile_targets`, which is a list of pruned targets to be passed to Ninja to build. It is acceptable to replace a list of pruned targets by a meta target if it turns out that all of the dependencies of the target are affected by the patch (i.e., all ten binaries that blink_tests depends on), but doing so is not required.
- `test_targets`, which is a list of unpruned targets to be mapped back to determine which tests to run.

There may be substantial overlap between the two lists, but there is no guarantee that one is a subset of the other and the two cannot be used interchangeably or merged together without losing information and causing the wrong thing to happen.

The implementation is responsible for recognizing 'all' as a magic string and mapping it onto the list of all root nodes in the build graph.

There may be files listed in the input that don't actually exist in the build graph: this could be either the result of an error (the file should be in the build graph, but isn't), or perfectly fine (the file doesn't affect the build graph at all). We can't tell these two apart, so we should ignore missing files.

There may be targets listed in the input that don't exist in the build graph; unlike missing files, this can only indicate a configuration error, and so we should return which targets are missing so the caller can treat this as an error, if so desired.

Any of the three inputs may be an empty list:

- It normally doesn't make sense to call analyze at all if no files were modified, but in rare cases we can hit a race where we try to test a patch after it has already been committed, in which case the list of modified files is empty. We should return 'no dependency' in that case.
- Passing an empty list for one or the other of test_targets and additional_compile_targets is perfectly sensible: in the former case, it can indicate that you don't want to run any tests, and in the latter, it can indicate that you don't want to do build anything else in addition to the test targets.

- It doesn't make sense to call analyze if you don't want to compile anything at all, so passing [] for both test_targets and additional_compile_targets should probably return an error.

In the output case, an empty list indicates that there was nothing to build, or that there were no affected test targets as appropriate.

Note that passing no arguments to Ninja is equivalent to passing `all` to Ninja (at least given how GN and GYP work); however, we don't want to take advantage of this in most cases because we don't actually want to build every out of date target, only the targets potentially affected by the files. One could try to indicate to analyze that we wanted to use no arguments instead of an empty list, but using the existing fields for this seems fragile and/or confusing, and adding a new field for this seems unwarranted at this time.

There is an "error" field in case something goes wrong (like the empty file list case, above, or an internal error in MB/GYP/GN). The analyze code should also return an error code to the shell if appropriate to indicate that the command failed.

In the case where build files themselves are modified and analyze may not be able to determine a correct answer (point 7 above, where we return "Found dependency (all)"), we should also return the `test_targets` unmodified and return the union of `test_targets` and `additional_compile_targets` for `compile_targets`, to avoid confusion.

Examples

Continuing the example given above, suppose we have the following build graph:

- `blink_tests` is a meta target that depends on `webkit_unit_tests`, `wtf_unittests`, and `webkit_tests` and represents all of the targets needed to fully test Blink. Each of those is a separate test step.
- `webkit_tests` is also a meta target; it depends on `content_shell` and `image_diff`.
- `base_unittests` is a separate test binary.
- `wtf_unittests` depends on `Assertions.cpp` and `AssertionsTest.cpp`.
- `webkit_unit_tests` depends on `WebNode.cpp` and `WebNodeTest.cpp`.
- `content_shell` depends on `WebNode.cpp` and `Assertions.cpp`.
- `base_unittests` depends on `logging.cc` and `logging_unittest.cc`.

Example 1 We wish to run 'wtf_unittests' and 'webkit_tests' on a bot, but not compile any additional targets.

If a patch touches WebNode.cpp, then analyze gets as input:

```
{
  "files": ["WebNode.cpp"],
  "test_targets": ["wtf_unittests", "webkit_tests"],
```

```
  "additional_compile_targets": []
}
```

and should return as output:

```
{
  "status": "Found dependency",
  "compile_targets": ["webkit_unit_tests"],
  "test_targets": ["webkit_tests"]
}
```

Note how `webkit_tests` was pruned in compile_targets but not in test_targets.

Example 2 Using the same patch as Example 1, assume we wish to run only `wtf_unittests`, but additionally build everything needed to test Blink (`blink_tests`):

We pass as input:

```
{
  "files": ["WebNode.cpp"],
  "test_targets": ["wtf_unittests"],
  "additional_compile_targets": ["blink_tests"]
}
```

And should get as output:

```
{
  "status": "Found dependency",
  "compile_targets": ["webkit_unit_tests"],
  "test_targets": []
}
```

Here `blink_tests` was pruned in the output compile_targets, and test_targets was empty, since blink_tests was not listed in the input test_targets.

Example 3 Build everything, but do not run any tests.

Input:

```
{
  "files": ["WebNode.cpp"],
  "test_targets": [],
  "additional_compile_targets": ["all"]
}
```

Output:

```
{
  "status": "Found dependency",
  "compile_targets": ["webkit_unit_tests", "content_shell"],
```

```
  "test_targets": []
}
```

Example 4 Same as Example 2, but a build file was modified instead of a source file.

Input:

```
{
  "files": ["BUILD.gn"],
  "test_targets": ["wtf_unittests"],
  "additional_compile_targets": ["blink_tests"]
}
```

Output:

```
{
  "status": "Found dependency (all)",
  "compile_targets": ["webkit_unit_tests", "wtf_unittests"],
  "test_targets": ["wtf_unittests"]
}
```

test_targets was returned unchanged, compile_targets was pruned.

Random Requirements and Rationale

This section is collection of semi-organized notes on why MB is the way it is ...

in-tree or out-of-tree

The first issue is whether or not this should exist as a script in Chromium at all; an alternative would be to simply change the bot configurations to know whether to use GYP or GN, and which flags to pass.

That would certainly work, but experience over the past two years suggests a few things:

- we should push as much logic as we can into the source repositories so that they can be versioned and changed atomically with changes to the product code; having to coordinate changes between src/ and build/ is at best annoying and can lead to weird errors.
- the infra team would really like to move to providing product-independent services (i.e., not have to do one thing for Chromium, another for NaCl, a third for V8, etc.).
- we found that during the SVN->GIT migration the ability to flip bot configurations between the two via changes to a file in chromium was very useful.

All of this suggests that the interface between bots and Chromium should be a simple one, hiding as much of the chromium logic as possible.

Why not have MB be smarter about de-duping flags?

This just adds complexity to the MB implementation, and duplicates logic that GYP and GN already have to support anyway; in particular, it might require MB to know how to parse GYP and GN values. The belief is that if MB does *not* do this, it will lead to fewer surprises.

It will not be hard to change this if need be.

Integration w/ gclient runhooks

On the bots, we will disable `gyp_chromium` as part of runhooks (using `GYP_CHROMIUM_NO_ACTION=1`), so that mb shows up as a separate step.

At the moment, we expect most developers to either continue to use `gyp_chromium` in runhooks or to disable at as above if they have no use for GYP at all. We may revisit how this works once we encourage more people to use GN full-time (i.e., we might take `gyp_chromium` out of runhooks altogether).

Config per flag set or config per (os/arch/flag set)?

Currently, mb_config.pyl does not specify the host_os, target_os, host_cpu, or target_cpu values for every config that Chromium runs on, it only specifies them for when the values need to be explicitly set on the command line.

Instead, we have one config per unique combination of flags only.

In other words, rather than having `linux_rel_bot`, `win_rel_bot`, and `mac_rel_bot`, we just have `rel_bot`.

This design allows us to determine easily all of the different sets of flags that we need to support, but *not* which flags are used on which host/target combinations.

It may be that we should really track the latter. Doing so is just a config file change, however.

Non-goals

- MB is not intended to replace direct invocation of GN or GYP for complicated build scenarios (a.k.a. Chrome OS), where multiple flags need to be set to user-defined paths for specific toolchains (e.g., where Chrome OS needs to specify specific board types and compilers).
- MB is not intended at this time to be something developers use frequently, or to add a lot of features to. We hope to be able to get rid of it once the GYP->GN migration is done, and so we should not add things for developers that can't easily be added to GN itself.
- MB is not intended to replace the CR tool. Not only is it only intended to replace the gyp_chromium part of `'gclient runhooks'`, it is not really meant as a developer-facing tool.