The MB (Meta-Build wrapper) user guide

[TOC]

Introduction

`mb` is a simple python wrapper around the GYP and GN meta-build tools to be used as part of the GYP->GN migration.

It is intended to be used by bots to make it easier to manage the configuration each bot builds (i.e., the configurations can be changed from chromium commits), and to consolidate the list of all of the various configurations that Chromium is built in.

Ideally this tool will no longer be needed after the migration is complete.

For more discussion of MB, see also the design spec.

MB subcommands

`mb analyze`

`mb analyze` is responsible for determining what targets are affected by a list of files (e.g., the list of files in a patch on a trybot):

```
mb analyze -c chromium_linux_rel //out/Release input.json output.json
```

Either the `-c/--config` flag or the `-m/--master` and `-b/--builder` flags must be specified so that `mb` can figure out which config to use.

The first positional argument must be a GN-style "source-absolute" path to the build directory.

The second positional argument is a (normal) path to a JSON file containing a single object with the following fields:

- `files`: an array of the modified filenames to check (as paths relative to the checkout root).
- `test_targets`: an array of (ninja) build targets that needed to run the tests we wish to run. An empty array will be treated as if there are no tests that will be run.
- `additional_compile_targets`: an array of (ninja) build targets that reflect the stuff we might want to build *in addition to* the list passed in `test_targets`. Targets in this list will be treated specially, in the following way: if a given target is a "meta" (GN: group, GYP: none) target like 'blink_tests' or or even the ninja-specific 'all' target, then only the *dependencies* of the target that are affected by the modified files will be rebuilt (not the target itself, which might also cause unaffected dependencies to be rebuilt). An empty list will be treated as if there are no additional targets to build. Empty lists for both `test_targets` and

`additional_compile_targets` would cause no work to be done, so will result in an error.
- `targets`: a legacy field that resembled a union of `compile_targets` and `test_targets`. Support for this field will be removed once the bots have been updated to use compile_targets and test_targets instead.

The third positional argument is a (normal) path to where mb will write the result, also as a JSON object. This object may contain the following fields:

- `error`: this should only be present if something failed.
- `compile_targets`: the list of ninja targets that should be passed directly to the corresponding ninja / compile.py invocation. This list may contain entries that are *not* listed in the input (see the description of `additional_compile_targets` above and design_spec.md for how this works).
- `invalid_targets`: a list of any targets that were passed in either of the input lists that weren't actually found in the graph.
- `test_targets`: the subset of the input `test_targets` that are potentially out of date, indicating that the matching test steps should be re-run.
- `targets`: a legacy field that indicates the subset of the input `targets` that depend on the input `files`.
- `build_targets`: a legacy field that indicates the minimal subset of targets needed to build all of `targets` that were affected.
- `status`: a field containing one of three strings:
  - `"Found dependency"` (build the `compile_targets`)
  - `"No dependency"` (i.e., no build needed)
  - `"Found dependency (all)"` (`test_targets` is returned as-is; `compile_targets` should contain the union of `test_targets` and `additional_compile_targets`. In this case the targets do not need to be pruned).

See design_spec.md for more details and examples; the differences can be subtle. We won't even go into how the `targets` and `build_targets` differ from each other or from `compile_targets` and `test_targets`.

The `-b/--builder`, `-c/--config`, `-f/--config-file`, `-m/--master`, `-q/--quiet`, and `-v/--verbose` flags work as documented for `mb gen`.

`mb audit`

`mb audit` is used to track the progress of the GYP->GN migration. You can use it to check a single master, or all the masters we care about. See `mb help audit` for more details (most people are not expected to care about this).

`mb gen`

`mb gen` is responsible for generating the Ninja files by invoking either GYP or GN as appropriate. It takes arguments to specify a build config and a directory, then runs GYP or GN as appropriate:

```
% mb gen -m tryserver.chromium.linux -b linux_rel //out/Release
% mb gen -c linux_rel_trybot //out/Release
```

Either the `-c/--config` flag or the `-m/--master` and `-b/--builder` flags must be specified so that `mb` can figure out which config to use. The `--phase` flag must also be used with builders that have multiple build/compile steps (and only with those builders).

By default, MB will look for a bot config file under `//ios/build/bots` (see design_spec.md for details of how the bot config files work). If no matching one is found, will then look in `//tools/mb/mb_config.pyl` to look up the config information, but you can specify a custom config file using the `-f/--config-file` flag.

The path must be a GN-style "source-absolute" path (as above).

You can pass the `-n/--dryrun` flag to mb gen to see what will happen without actually writing anything.

You can pass the `-q/--quiet` flag to get mb to be silent unless there is an error, and pass the `-v/--verbose` flag to get mb to log all of the files that are read and written, and all the commands that are run.

If the build config will use the Goma distributed-build system, you can pass the path to your Goma client in the `-g/--goma-dir` flag, and it will be incorporated into the appropriate flags for GYP or GN as needed.

If gen ends up using GYP, the path must have a valid GYP configuration as the last component of the path (i.e., specify `//out/Release_x64`, not `//out`). The gyp script defaults to `//build/gyp_chromium`, but can be overridden with the `--gyp-script` flag, e.g. `--gyp-script=gypfiles/gyp_v8`.

`mb help`

Produces help output on the other subcommands

`mb lookup`

Prints what command will be run by `mb gen` (like `mb gen -n` but does not require you to specify a path).

The `-b/--builder`, `-c/--config`, `-f/--config-file`, `-m/--master`, `--phase`, `-q/--quiet`, and `-v/--verbose` flags work as documented for `mb gen`.

`mb validate`

Does internal checking to make sure the config file is syntactically valid and that all of the entries are used properly. It does not validate that the flags make sense, or that the builder names are legal or comprehensive, but it does complain about configs and mixins that aren't used.

The `-f/--config-file` and `-q/--quiet` flags work as documented for `mb gen`.

This is mostly useful as a presubmit check and for verifying changes to the config file.

`mb gerrit-buildbucket-config`

Generates a gerrit buildbucket configuration file and prints it to stdout. This file contains the list of trybots shown in gerrit's UI.

The master copy of the buildbucket.config file lives in a separate branch of the chromium repository. Run `mb gerrit-buildbucket-config > buildbucket.config.new && git fetch origin refs/meta/config:refs/remotes/origin/meta/config && git checkout -t -b meta_config origin/meta/config && mv buildbucket.config.new buildbucket.config` to update the file.

Note that after committing, `git cl upload` will not work. Instead, use `git push origin HEAD:refs/for/refs/meta/config` to upload the CL for review.

Isolates and Swarming

`mb gen` is also responsible for generating the `.isolate` and `.isolated.gen.json` files needed to run test executables through swarming in a GN build (in a GYP build, this is done as part of the compile step).

If you wish to generate the isolate files, pass `mb gen` the `--swarming-targets-file` command line argument; that arg should be a path to a file containing a list of ninja build targets to compute the runtime dependencies for (on Windows, use the ninja target name, not the file, so `base_unittests`, not `base_unittests.exe`).

MB will take this file, translate each build target to the matching GN label (e.g., `base_unittests -> //base:base_unittests`, write that list to a file called `runtime_deps` in the build directory, and pass that to `gn gen $BUILD ... --runtime-deps-list-file=$BUILD/runtime_deps`.

Once GN has computed the lists of runtime dependencies, MB will then look up the command line for each target (currently this is hard-coded in mb.py), and write out the matching `.isolate` and `.isolated.gen.json` files.

The `mb_config.pyl` config file

The `mb_config.pyl` config file is intended to enumerate all of the supported build configurations for Chromium. Generally speaking, you should never need to (or want to) build a configuration that isn't listed here, and so by using the configs in this file you can avoid having to juggle long lists of GYP_DEFINES and gn args by hand.

`mb_config.pyl` is structured as a file containing a single PYthon Literal expression: a dictionary with three main keys, `masters`, `configs` and `mixins`.

The `masters` key contains a nested series of dicts containing mappings of master -> builder -> config . This allows us to isolate the buildbot recipes from the actual details of the configs. The config should either be a single string value representing a key in the `configs` dictionary, or a list of strings, each of which is a key in the `configs` dictionary; the latter case is for builders that do multiple compiles with different arguments in a single build, and must *only* be used for such builders (where a –phase argument must be supplied in each lookup or gen call).

The `configs` key points to a dictionary of named build configurations.

There should be an key in this dict for every supported configuration of Chromium, meaning every configuration we have a bot for, and every configuration commonly used by developers but that we may not have a bot for.

The value of each key is a list of "mixins" that will define what that build_config does. Each item in the list must be an entry in the dictionary value of the `mixins` key.

Each mixin value is itself a dictionary that contains one or more of the following keys:

- `gyp_crosscompile`: a boolean; if true, GYP_CROSSCOMPILE=1 is set in the environment and passed to GYP.
- `gyp_defines`: a string containing a list of GYP_DEFINES.
- `gn_args`: a string containing a list of values passed to gn –args.
- `mixins`: a list of other mixins that should be included.
- `type`: a string with either the value `gyp` or `gn`; setting this indicates which meta-build tool to use.

When `mb gen` or `mb analyze` executes, it takes a config name, looks it up in the 'configs' dict, and then does a left-to-right expansion of the mixins; gyp_defines and gn_args values are concatenated, and the type values override each other.

For example, if you had:

```
{
  'configs`: {
    'linux_release_trybot': ['gyp_release', 'trybot'],
    'gn_shared_debug': None,
```

```
  }
  'mixins': {
    'bot': {
      'gyp_defines': 'use_goma=1 dcheck_always_on=0',
      'gn_args': 'use_goma=true dcheck_always_on=false',
    },
    'debug': {
      'gn_args': 'is_debug=true',
    },
    'gn': {'type': 'gn'},
    'gyp_release': {
      'mixins': ['release'],
      'type': 'gyp',
    },
    'release': {
      'gn_args': 'is_debug=false',
    }
    'shared': {
      'gn_args': 'is_component_build=true',
      'gyp_defines': 'component=shared_library',
    },
    'trybot': {
      'gyp_defines': 'dcheck_always_on=1',
      'gn_args': 'dcheck_always_on=true',
    }
  }
}
```

and you ran `mb gen -c linux_release_trybot //out/Release`, it would translate into a call to `gyp_chromium -G Release` with `GYP_DEFINES` set to `"use_goma=true dcheck_always_on=false dcheck_always_on=true"`.

(From that you can see that mb is intentionally dumb and does not attempt to de-dup the flags, it lets gyp do that).

Debugging MB

By design, MB should be simple enough that very little can go wrong.

The most obvious issue is that you might see different commands being run than you expect; running `'mb -v'` will print what it's doing and run the commands; `'mb -n'` will print what it will do but *not* run the commands.

If you hit weirder things than that, add some print statements to the python script, send a question to gn-dev@chromium.org, or file a bug with the label 'mb' and cc: dpranke@chromium.org.