Bazel

A new home for the host platform

- **Authors:** wyv@bazel.build (Xudong Yang)
- **Status:** Approved
- **Reviewers:** fabian@meumertzhe.im (Fabian Meumertzheim), jcater@google.com (John Cater), pcloudy@google.com (Yun Peng)
- **Created:** 2024-03-18
- **Updated:** 2024-03-18

Please read Bazel Code of Conduct before commenting.

Background

- We're in the process of Starlarkifying all native repo rules (#18285), and `local_config_platform` is one of them.
  - This repo rule is trivially Starlarkifiable.
- The rule is currently used by the identically-named built-in module (see code).
  - Being a built-in module, it's implicitly added as a dependency to every other module in the external dependency graph, which means that the apparent repo name `@local_config_platform` is visible to *all* repos.
  - This is important because the current default value for the `--host_platform` flag is `@local_config_platform//:host` (see code).
  - We'd like to remove this special case. Ideally, `@bazel_tools` should be the only built-in module.
- There's a related feature request to inject custom constraints into the detected host platform (#8766).

Proposed solution

Create a new home for the host platform

- We can easily Starlarkify the repo rule `local_config_platform` as it only depends on information we already have access to via `rctx.os`.
- We can place the code in the module `platforms`.

Bazel

    - `platforms` is a crucial dependency of effectively all Bazel projects (since it's a dependency of `bazel_tools`), and is an appropriate home for the host platform (but see alternatives below).
    - We can put the code for both the repo rule and the module extension into the package `@platforms//host`. The repo rule only needs to generate the list of host constraints, whereas the host platform itself can be located at `@platforms//host:host`.
- In summary, the following changes are made to `@platforms`: (see proof of concept PR platforms#86)
    - New file `host/extension.bzl` which contains the code for an extension `host_platform` and a repo rule `host_platform_repo`. The extension does nothing but call the repo rule `host_platform_repo` generating a repo called `host_platform`.
    - New file `host/constraints.bzl` which simply re-exports the symbol `HOST_CONSTRAINTS` from `@host_platform//:constraints.bzl` (this is the generated host constraints list).
    - New file `host/BUILD` which defines a `platform()` target called `host`, whose constraints are `HOST_CONSTRAINTS` from `constraints.bzl`.
    - The MODULE.bazel file of `@platforms` uses the `host_platform` extension, and imports its generated repo `host_platform` with `use_repo`.
- We can then change the default value of the `--host_platform` flag to point to the new target in `platforms`.
    - Do note that label-typed flag default values *must* point into a built-in module, since they are resolved from the view of the main repo and we can't make any assumptions about the main repo other than the fact that it has an implicit dependency on all built-in modules.
    - Since `platforms` is not a built-in module, we can't directly set the default value of `--host_platforms` to `@platforms//host`.
    - This means that our only choice is to use the remaining built-in module `bazel_tools`. We can make `@bazel_tools//tools:host_platform` an alias of `@platforms//host`.

Deprecate `local_config_platform`

- We can then work to deprecate `local_config_platform`, both the native repo rule and the built-in module.
- As a first step, we can change the native repo rule (implementation: `LocalConfigPlatformFunction.java`) to instead produce a simple wrapper repo around the new stuff in `platforms`:
    - The file `@local_config_platform//:constraints.bzl` simply re-exports the symbol `HOST_CONSTRAINTS` from `@platforms//host:constraints.bzl`.
    - The target `@local_config_platform//:host` becomes an alias of `@platforms//host`.
- We then add a new incompatible flag to disable the built-in module and disallow usage of the native repo rule.

    - This flag can be flipped in Bazel 8.0, and removed in Bazel 9.0.
- User code that directly refers to `@local_config_platform` will need to be updated to use things in `@platforms//host` instead.
    - The migration can start right away; all changes described above can be back-ported to 7.x (sans the flag flip and removal).

Implement custom constraint injection

- The fact that the new `host_platform` repo is generated by a module extension opens up new possibilities; most notably, a way for modules to inject custom constraints into the host platform.
    - For example, a ruleset offering GPU acceleration might wish to augment the host platform with constraints on the GPU type.
- The core idea is that such an "injector module" could define its own repo rule to run some arbitrary detection logic, and output its findings into a file. It can then inform the `host_platform` module extension about this file; the extension can then work the custom constraints in this file into the `HOST_CONSTRAINTS` list. (see proof of concept in fmeum/host_platform)
- Concretely, the `host_platform` module extension defined in `platforms` can offer a tag class, `add_constraints`, which accepts a singular label attribute that points to a JSON file (why not a .bzl file? see alternatives). This JSON file should contain a single list of strings, each of which is a `constraint_value` label.
- An example:

```python
############################
### rules_gpu's MODULE.bazel
bazel_dep(name = "platforms", version = "0.0.9")

# `my_extension` calls a repo rule that runs arbitrary detection logic, and
generates
# a repo `my_constraints_repo` containing the list of custom constraints.
my_extension = use_extension("//:my_extension.bzl", "my_extension")
use_repo(my_extension, "my_constraints_repo")

host_platform = use_extension("@platforms//host:extension.bzl",
"host_platform")
host_platform.add_constraints(file =
"@my_constraints_repo//:constraints.json")

############################
### @rules_gpu//:my_extension.bzl
def _my_constraints_repo_impl(rctx):
  gpu_name = _somehow_retrieve_current_gpu_name(rctx)
  gpu_label = Label("//gpu:" + gpu_name)
  rctx.file("BUILD.bazel")
  rctx.file("constraints.json", '["' + str(gpu_label) + '"]')
my_constraints_repo = repository_rule(_my_constraints_repo_impl)
```

```
my_extension = module_extension(
    lambda _mctx: my_constraints_repo(name = "my_constraints_repo"),
)

############################
### @my_constraints_repo//:constraints.json (a JSON list of canonical
labels)
["@@rules_gpu~//gpu:rtx_4090"]
```

Alternatives considered

Place the host platform into a module other than `platforms`

- Instead of placing the host platform detection code (including the module extension, repo rule, actual `platform()` target) into `platforms`, we could create another module dedicated to the host platform.
- Pros
    - `platforms` is a very simple module today; it contains nothing but `constraint_setting` and `constraint_value` definitions for CPU and OS types. Host platform detection code is rather unlike these current occupants.
    - There is precedent for this: `platform_data` was added experimentally into `platforms` (platforms#78), but was then moved into its own module at `rules_platform`.
- Cons
    - Above all, a module named `platforms` seems like a natural place to place the host platform.
    - Because `bazel_tools` will need to depend on whichever module we put the host platform in, this new module would necessarily be present in every Bazel project and pollute the dependency graph further.
    - Creating another module has the usual administrative costs: separate source repo, release process, etc.
    - The considerations that resulted in the creation of `rules_platform` don't necessarily apply in this case. See this doc comment for more details.

Retain the name `local_config_platform`

- Instead of calling the extension (and generated repo) `host_platform`, we could retain the old name `local_config_platform`.
- Pros

    - It's slightly more familiar.
- Cons
    - It doesn't match the flag name (`--host_platform`), nor the name of the concept used in human language ("the host platform").

Custom constraint injection: Ask for a .bzl instead

- In the tag class `add_constraints`, instead of asking for a label to a JSON file, we could ask for a label pointing into a Starlark file (.bzl) that declares a label list constant with some conventional name (for example, `HOST_CONSTRAINTS` or `CUSTOM_HOST_CONSTRAINTS`).
- Pros
    - Starlark is used throughout Bazel.
- Cons
    - Starlark is too powerful for this use case. All we need is a string list.
    - Using a .bzl file passed in via a label attribute is very awkward because we can't dynamically `load()` a .bzl file. We'd have to use another repo rule to generate a repo containing a .bzl file that `load()`s from the given file, and then load from this other repo to export the constant. JSON parsing, however, is readily available in Starlark.

Custom constraint injection: Ask for a plain text file instead

- Same as above, but even simpler: just ask for a plain text file, with one constraint label per line.
- Pros
    - Can't get much simpler than a plain text file. Even JSON is arguably too powerful for a "string list" use case.
- Cons
    - Amazingly, a newline-separate text file is somewhat of a "custom file format" in that it doesn't have a short name, so it could be argued that it's worsening the "too many file formats in Bazel" problem. (Consider .bazelignore, which is at a similar spot.)

Document History

Date	**Description**
2024-03-18	First proposal

Bazel