Bazel

A true repository cache for Bazel

Authors: wyv@bazel.build (Xudong Yang)
Status: In review
Reviewers: fabian@meumertzhe.im (Fabian Meumertzheim), lberki@google.com (Lukács Berki), pcloudy@google.com (Yun Peng)
Tracking issue: #12227
Created: 2023-09-07
Updated: 2023-09-07

Please read Bazel Code of Conduct before commenting.

Background

- Bazel's `--repository_cache` flag today is technically a misnomer: it's actually a download cache, containing blobs (often unextracted archives) keyed by their sha256 checksums.
    - This cache can be (and often is) very safely shared by multiple Bazel workspaces on the same machine.
- There is a need for a true "repository cache": that is, a cache containing actual repository contents, for example extracted archive contents.
    - The major use case is for CI -- runners could be pre-populated with a repo cache to save archive download and extraction time during CI runs.
    - This cache could also be shared among different Bazel workspaces in many cases, speeding up the workflow for users with such setups.

Proposed design

Concepts and paths

- Amend the "repository cache" concept to include the "true" repo cache.
- Taking advantage of the directory layout of the existing `--repository_cache` flag, add a new subdirectory for the "true" repo cache:
    - `$REPO_CACHE/content_addressable/sha256` is the *download cache*. It already exists today and contains blobs downloaded from the internet, keyed by their sha256 hash.

        - The download cache contains one subdirectory for each entry. The name of the subdirectory is the sha256 hash. The subdirectory contains one file, named `file`, which contains the content of the blob.
        - We could also follow the directory name and call this the *content-addressable cache*... which is not wrong, but also doesn't prevent confusion by pointing out that it only serves blob downloads.
    - `$REPO_CACHE/contents` is the *true repo cache*, or the *repo contents cache*.
        - Pairs of metadata files and directories exist under this directory, each pair representing a cache entry.
        - For each `$REPO_CACHE/contents/$FOO` directory containing the cached contents, the file `$REPO_CACHE/contents/$FOO.metadata` contains cache entry metadata. See the sections below for more information.

Inputs

- Define the "input" of a repo fetch as any entity that could affect the output of the repo fetch (i.e. the contents).
    - Note that we need to distinguish the "identity" of the input (e.g. "the time of day") from the "value" of the input (e.g. "12:34").
- As repo rules can run arbitrary code, the comprehensive list of "inputs" is effectively everything. Nevertheless, we could split them into three categories.
- **Predeclared**: any input we can know before fetching the repo. This includes:
    - The `repository_rule` definition, including the implementation function, attribute definitions, the list of environment variables its behavior depends on (`environ`), etc.
    - The repo's attribute values. For example, for an `http_archive` repo, its `name`, `urls`, `patches`, etc.
    - The `StarlarkSemantics` object, which represents flags that affect Starlark behavior.
- **Recorded**: any input we can record during the fetch and thus know about after the fetch. This includes:
    - Any files addressed by a label that the repo rule accessed during the fetch.
    - Any environment variables accessed through the `rctx.getenv` method.
- **Undiscoverable**: any input that is used during the fetch but is not recorded/recordable. This includes:
    - Any downloads done using `rctx.download{,_and_extract}`.
        - Thankfully these are usually reproducible thanks to sha256/integrity checks.
    - Any environment variables that are accessed through `rctx.os.environ` but not declared in the `repository_rule`'s `environ` parameter.
    - Any filesystem reads done by `rctx.read` that do not originate from labels.
    - Any filesystem traversal done by `path.readdir`.
    - The workspace root reported by `rctx.workspace_root`.
    - Potentially *any* input used by executables run by `rctx.execute`.

Opt-in to caching

- The presence of undiscoverable inputs poses a fundamental challenge to caching repo contents.
  - An earlier version of this document proposed to always enable caching for any non-local repos, but this is dangerous and may result in cache poisoning that is hard to recover from.
  - Allowing certain repo rules to explicitly opt in to caching avoids this risk, while we can already reap most of the benefits of repo caching by having the built-in `http_archive` opt in.
- Hence we don't cache repo contents by default. A repo rule can return a new `rctx.RepoMetadata` object from its implementation function to opt in:

```
def _http_archive_impl(rctx):
    # ...
    return rctx.RepoMetadata(
        reproducible = rctx.attr.integrity is not None,
    )
http_archive = repository_rule(implementation = _http_archive_impl, ...)
```

  - The `rctx.RepoMetadata` method takes two parameters:
    - `reproducible`: Boolean, defaults to False. If True, the repo contents can be cached.
    - `attrs_for_reproducibility`: Dict or None, defaults to None. Can only be non-None if `reproducible` is False. If non-None, represents the attributes to change so that the repo is reproducible. This replaces the old semantics of returning a struct from the impl function.

Cache lookup process: broad strokes

- Before we fetch a repo *R*, we need to determine whether a suitable entry already exists in the cache. Such an entry should have been fetched using the same inputs at the same values as *R*.
- Before fetching *R*, we don't know the identities of its *recorded* inputs. But we do know the identities (and hence values) of all its *predeclared* inputs.
- Given that, we could adopt a two-step approach:
  - The first step is a simple lookup by the hash of *R*'s predeclared inputs.
    - If no cache entry matches, it's a definite cache miss, so we can look to write a new entry into the cache.
  - The second step is to verify that the entry is up-to-date. Each entry advertises its *recorded* inputs (both the identities and the hashes of values at time of fetch), and we simply check that the hashes match the up-to-date values.
    - If the hashes are up-to-date, it's a cache hit and we can simply reuse the cache contents.
    - Otherwise, we need to refetch a fresh *R* and put it into the cache instead, while being mindful of race conditions.

Cache lookup process: details

- Definitions
  - For a repo *R*, let *Pre(R)* be the set of *R*'s predeclared inputs, and *Rec(R)* be the set of *R*'s recorded inputs. These are representations of the *identities* of those inputs (for example, *Rec(R)* could contain *which* files *R* accessed during its fetch).
  - Let *Val(I)* be the *value* of an input *I* (or the set of values of a set of inputs *I*). For example, for an *F* in *Rec(R)* that represents a file accessed during *R*'s fetch, *Val(F)* represents the contents of that file.
  - Let *Hash()* be a hash function that produces a digest. Which function is used doesn't particularly matter; we can assume SHA-256 for example.

1. Before fetching a repo *R*, compute the hash of *R*'s predeclared input values, *HPre(R) = Hash(Val(Pre(R)))*. If the file `$REPO_CACHE/contents/${HPre(R)}.metadata` doesn't exist, this is a cache miss; go to step 3.
2. The directory `$REPO_CACHE/contents/${HPre(R)}` represents a previously fetched repo R_c which is a candidate to be reused as *R*.
   - The file `$REPO_CACHE/contents/${HPre(R)}.metadata` should contain a list of pairs, representing all elements of $Rec(R_c)$ and the hash of each individual element at the time of R_c's fetch.
   - If, for all <*I*, *HV*> in the list of pairs in `$REPO_CACHE/contents/${HPre(R)}.metadata`, *Hash(Val(I)) = HV*, then we've located a cache hit. Go to step 3e.
   - Otherwise, we need to refetch *R*. Go to step 3.
3. **[Cache miss]** We need to fetch *R* and write an entry into the repo cache.
   a. See if the file `$REPO_CACHE/contents/${HPre(R)}.lock` exists. If so, wait until it's gone, and go to step 3e.
   b. Create the file `$REPO_CACHE/contents/${HPre(R)}.lock`.
   c. Clear the directory `$REPO_CACHE/contents/${HPre(R)}` and fetch *R* into it. Populate `$REPO_CACHE/contents/${HPre(R)}.metadata` with the list of pairs being <*I*, *Hash(Val(I))*> for all *I* in *Rec(R)*.
   d. Delete `$REPO_CACHE/contents/${HPre(R)}.lock`.
   e. **[Cache hit]** Symlink/hardlink *R*'s originally designated location in the output base to `$REPO_CACHE/contents/${HPre(R)}`.

Labels in inputs

- Labels in inputs require special attention as the repository cache is shared across all workspaces on the machine.
  - The same canonical label `@@foo//:bar` could mean different things in different workspaces.
  - The principle we need to uphold is that a cache hit should yield the same contents as a fresh fetch (disregarding any differences resulting from undiscoverable inputs).
- Most notably, labels are the identities of the only kind of recorded inputs.

    - In step 2, when we try to calculate *Hash(Val(I))*, *I* here is a label that was used during the fetch of R_c, and was thus resolved in the context of the workspace that R_c originated from.
    - But to calculate *Hash(Val(I))*, we need to resolve *I* in the context of the workspace that *R* originates from, which may end up being a completely different file! But that's okay, since if the hash of the contents remains the same, we can assume that the result of the fetch will remain the same.
    - Note that technically the repo rule could resolve the label to an absolute path and perform different logic based on the resolved path.
        - However, this is likely uncommon at best and pathological at worst, so we can count this kind of usage as an undiscoverable input.
        - In the event that this case is deemed important, we can split this input into two: 1) identity=label, value=resolved absolute path; 2) identity=resolved absolute path, value=file contents. This may lead to thrashing.
- Labels can also appear in repo attribute values, as part of the values of predeclared inputs.
    - These can be stored verbatim as canonical labels without problems.
    - Consider two workspaces W_1 and W_2, each containing a repo R_1 and R_2 respectively. R_1 and R_2 have identical predeclared inputs, including a label `@@foo//:bar`. Even if this label points to completely different files when resolved in the two workspaces, as long as those files have the same contents (checked by the recorded input hash step), R_1 and R_2 should have the exact same contents.

Automatic eviction

- Since any change to *HPre(R)* would result in a new cache entry, and the repo cache is shared among multiple workspaces, we'd likely need to consider having some sort of automatic cache entry eviction policy.
- A simple TTL-based (time-to-live) policy should suffice.
    - We could add a flag `--repository_cache_ttl` defaulting to `30d` to specify the TTL.
    - In step 3e, when we create the symlink/hardlink, we would additionally update the "last used time" of the cache entry in its metadata file.
    - The Bazel server could periodically go through all entries in the repo cache and clear out any entries that are past TTL. This could be done in a background thread, potentially rate-limited to once per day.

Manual eviction / opt-out

- In the current design, there is no way to forcibly refetch a repo. This is potentially problematic due to the various undiscoverable inputs affecting the result of fetches, but not being reflected in cache metadata whatsoever.
    - Without a repo cache, the user could rely on the sledgehammer `bazel clean --expunge`. But this command would essentially become

synonymous with `bazel clean` if a true repo cache is used, unless the user explicitly clears the repo cache too.

- To combat this, we could introduce some sort of manual eviction command. This could be tagged onto `bazel fetch`: the flag `--force` could be used to forcibly refetch the repo even if it's already in the cache, overwriting the cache contents.
- A repo rule could also simply opt out of ever being cached by specifying `local=True`.
    - This is consistent with its semantics today: a local repo rule never checks its marker file.

Race conditions

- Since the repo cache is shared by multiple workspaces (and hence potentially multiple running Bazel server processes), we need to be aware of potential race conditions.
- The existing download cache doesn't seem to guard against concurrent usage by multiple Bazel servers. But entries in the download cache are single files, and a cache write is a single `mv` operation (see code), so there was hardly any reason for concern.
- With the true repo cache, things get a bit more complicated.
    - First, multiple files are involved; having two processes populate the same directory, or having one process populate a directory and another read from it, are both very dangerous.
    - Second, we can't very easily `mv` a fetched repo directory, since there may be symlinks pointing inside.
- As described in steps 3a and 3b above, we could employ a "lock" file that serves no other purpose than to signal that another Bazel server is currently writing into a cache entry.
    - This "lock" file could simply contain the PID of the active server.
    - Not sure if Skyframe has facilities to wait for the disappearance of a file, but such an operation should be technically feasible.

Interaction with Directory watching API for repo rules

- With Directory watching API for repo rules, all file reads and filesystem traversals done with `rctx` (except inside `rctx.execute`) become recorded inputs instead of undiscoverable inputs.
- A major difference is that, in that design, we could always store the absolute file path as the "key" in the marker data; but with the true repo cache, we need to be careful to always store the *original form* of the input. That is, if it's a label, store it as a label; if it's an absolute path, store it as an absolute path, etc.
    - Consider the following scenario: in a workspace *W*, we fetch the repo *R*, during which we perform a `rctx.read(Label("@@foo//:bar"))`. `@@foo//:bar` corresponds to the absolute path

`${OUTPUT_BASE(W)}/external/foo/bar`. In *R*'s metadata file, we record this absolute path, and its hash at time of fetch.
    - Now in workspace *W'*, we fetch the repo *R'*, which has exactly the same definition as *R* in *W*. So it necessarily has the same prerecorded input hash (i.e. *HPre(R) = HPre(R')*). In step 2 when we try to verify the metadata file, we see that `${OUTPUT_BASE(W)}/external/foo/bar`'s hash hasn't changed, so we reuse the cache entry. But that's wrong! Because in *W*, `@@foo//:bar` corresponds to the absolute path `${OUTPUT_BASE(W)}/external/foo/bar`, which could have completely different contents.
    - If we store the label itself (`@@foo//:bar`) in the metadata instead, then when fetching *R'*, we'd be checking the hash of `${OUTPUT_BASE(W')}/external/foo/bar` (note the `W'`), thus recognizing that the file should have been updated.
- To deduce the "original form" of an input (while fetching *R* in the workspace *W*):
    - If the input is a file inside `${OUTPUT_BASE(W)}/external/$R`, then this file is not really an "input" -- it must have been produced by an earlier step in the repo rule. So we don't store this one.
    - If the input is a file inside another subdirectory of `${OUTPUT_BASE(W)}/external`, for example `${OUTPUT_BASE(W)}/external/foo/bar/baz`, then we store it as the label `@@foo//bar:baz`.
    - If the input is a file inside *W* itself, for example `$W/foo/bar`, then we store it as the label `@@//foo:bar`.
    - Otherwise, we store it as an absolute path.
    - The decision process is similar for readdir()'d directories.
- Note that there is still the obscure pathological case of labels with apparent repo names directly used in the repo rule implementation (e.g. `rctx.read(Label("@foo//:bar"))`).
    - This is dangerous because technically, this label could point to `@@foo//:bar` in workspace *W*, but `@@foo'//:bar` in workspace *W'*. In the metadata file, the best we could do is to store it as `@@foo//:bar`, and when we try to fetch *R'*, this would be an erroneous cache hit if `@@foo'//:bar` in *W'* has different contents.
    - This is only possible in a very obscure setting, so we can probably afford to ignore it.
    - Note that if the label with apparent repo names is passed in as an attribute to the repo rule, then there is no issue (since the repo rule only sees a canonical repo name).

Unresolved questions

Alternative: second layer of lookups instead of mutation

- This document originally proposed an "append-only" cache with no mutation.
    - The first-step lookup using *HPre(R)* would yield a set of candidates, instead of just one. Each candidate would have its own metadata file advertising its *Rec(R_c)* and corresponding hashes.
    - Then we would do a second layer of lookups by iterating through each candidate, and looking for one whose advertised metadata is up-to-date.
    - If we don't find one (cache miss), instead of mutating an entry, we would write a new entry under *HPre(R)* by fetching *R* there and recording its metadata. The directory structure would be accordingly adjusted.
- This setup is quite a bit more complex, but avoids thrashing in the rare case where *Val(Rec(R_c))* for a candidate entry R_c could be different depending on the workspace, even if *Rec(R_c)* is the same.
    - In English, that means: the file addressed by the label `@@foo//:bar` in workspace *W* could have hash *X*, but the file addressed by the same label in workspace *W'* could have a different hash *X'*.
    - Both entries are valid and "up-to-date". With the mutation-instead-of-second-layer approach, we'd potentially be constantly refetching if workspaces *W* and *W'* are both active.
- Additionally, the second layer of lookups could help in other cases where inputs evaluate differently based on the workspace.
    - For example, the workspace root (`rctx.workspace_root`) could be a recorded input to make sure running the same repo rule in different workspaces doesn't result in cache poisoning.
    - We don't want the workspace root to be a predeclared input, as that would preclude any repos from different workspaces being shared. So the best we could do is to make the workspace root a recorded input, in the event that `rctx.workspace_root` is ever called.
        - This is actually fairly hard to achieve today, as `rctx.workspace_root` is a field and not a method, which means that it has no access to the `StarlarkThread` (code comment).
- TODO: Find out whether the two scenarios above are likely enough that we should switch to the second layer of lookups.

Worse race conditions?

- Does Skyframe notice if a source artifact suddenly changes from under us during a build? This has a higher chance of happening with a true repo cache.
- TODO: Find out if we have means of defense against this scenario.

Document History

Date	Description
2023-09-07	First proposal