Namespaces when using multiple bazel repositories

Klaus Aehlig <aehlig@google.com>, Dmitry Lomov <dslomov@google.com>; Dec 2017

This memo describes some of the different name spaces involved when working in bazel using multiple repositories, as of bazel `0.8.0`. It also proposes some ways to handle those different name spaces.

Status of this memo

This memo serves to collect name-space issues to keep in mind when designing an overhaul of bazel's handling of external repositories. It is not an agreed-upon design document. Distribution of this memo is unlimited.

Background

While originally designed to handle a single, but large, repository, bazel does allow builds involving multiple repositories. To do so, the `WORKSPACE` file of the primary repository can declare additional repositories using declarations like `local_repository`, `git_repository`, and `http_repository`.

While there are plans to support evaluating `WORKSPACE` files recursively, we expect that the top-level `WORKSPACE` file will always have the power to override any repository requests by transitive `WORKSPACE` files. So for the purpose of this discussion, we can assume that all repositories are declared (maybe by means of delegation) by the top-level `WORKSPACE`.

Technically, there is a distinction between the "main" repository and the included repositories. But we nevertheless have to handle the case of included repository interacting with another included repository. This is already the general case, we can always change bazel to treat the main repository as a `local_repository` that happens to be located at the current working directory. So for the purpose of this discussion, we can assume that the main repository only brings a `WORKSPACE` declaration and nothing else.

Non-goals of this memo

This memo does not address the problem of depending on pre-installed libraries or tools. We expect a separate design discussion for handling situation where the other wants to satisfy the dependency of repository on another by providing replacements for the targets in the second repository that the first repository depends on (e.g., by pointing to tools or libraries already installed on the system by some form of package manager).

It does not address either the question of applying patches and algorithmic transformations to an external repository upon import, nor the question of the cache key for external repositories.

Repository names

When a repository refers to a target (tool, data file, etc) in a different repository, it does so using the `@other_repo` notation. I.e., repositories refer to other repositories by a symbolic name. We currently pretend that those names are canonical for the repository and that equality of repositories coincides with equality of the symbolic names, no matter which repository uses them. Bazel gives the general advice of using as symbolic name the name the referred-to repository gives itself in its `WORKSPACE` and additionally advises that a `WORKSPACE` call itself the reverse of the "canonical" URL of that repository, like `com_example_foo_utils`. While such advice helps to mostly align the repository names, this does not suffice in a number of legitimate use cases, including, but not limited to, the following.

- different versions: splitting diamonds.
  A typical dependency that might require using different version is the protobuf compiler. While the binary exchange format is stable, the generated code changes frequently, also in incompatible ways. So it might well be that a repository depends on two tools in different repositories that can cooperate by exchanging data in serialized proto format, but in their internal code require a specific, but different, version of `protoc`. As both those repositories would refer to their needed version of the protobuf repository as `com_google_protobuf` there is a name conflict that requires renaming of at least one repository.

- drop-in replacements: joining diamonds.
  Sometimes for a task (e.g., an ssl-library) there are several implementations with compatible programming interfaces. In fact, sometimes one library is even designed to be a drop-in replacement of another one. Such libraries necessarily expose common symbols and hence linking both in the same binary likely causes problems.
  Nevertheless, two needed repositories may well specify different of those equivalent

libraries---and use the canonical name of each. In such a situation it is desirable to identify (in the top-level workspace) two different repositories (that are called differently).

Additional problems arise due to non-qualified naming (simply calling a repository `tools` or `utils`, as initially it was not anticipated that it might have a wider use), or repositories that change their name within their lifetime (this causes problems if two repositories refer to the same third repository to versions before and after the name change). Such a name change might happen for various reasons, e.g., change of the product name for branding reasons or change of primary repository location from a hosting service to an own domain.

Fortunately, the symbolic names for repositories are only used in `BUILD` or `*.bzl` files, so they can be interpreted the context of a specific repository.

Paths and file names

Executables executed via `bazel run` are executed in a directory, reflecting the package structure, i.e., an action may assume that each declared dependency from its own repository is found (relative to its working directory) at a relative path equal to that of the label (relative to the repository root). E.g., an executable declaring `//foo/bar:baz.txt` as data dependency may assume the artifact is located at `foo/bar/baz.txt`, relative to the directory it is executed in. While the specification language for actions allows to request that the relative path be provided explicitly on the command line, it is common practise to rely on this assumption and refer to a declared dependency directly knowing the relative path. However, this guarantee is not true if the same executable target is called from an action; this can lead to situations where, even for a single repository, you can have an executable that works fine when called via `bazel run`, but breaks if called from an action. Nevertheless, this guarantee is true (and used) for purely source files (leading to the other undesirable effect of actions breaking once a source file is replaced by a generated source file).

Moreover, as soon as we have more than one repository, that mean we effectively have one name space per repository: different repositories may have different artifacts under the same (repo-local) label.

Unfortunately, those name spaces are not separate, as soon as an action of one repository calls a tool of a different repository. Then the caller might pass a reference to a data file in its repository, whereas the callee might have a data-dependency on a file in its own repository. Those repository-root-relative paths then refer to different repository roots.

The current approach is to ignore the namespace clash and, in fact crash on calling an executable from a different repository (https://github.com/bazelbuild/bazel/issues/4170).

Proposal

This section proposes a way to live in the situation described in the sections before. tl;dr: don't assume, ask. Bazel will not make guarantees about the relative location of the various roots (for each repository, there is the root of the sources and the root of the generated files), instead it will provide portable cache-friendly paths to reach those.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Standardizing on execroot layout

While the runfiles layout is more convenient from a programmer's point of view, bazel won't provide this layout for actions. Therefore executable that are intended to be used as tools during the build MUST be prepared to run in an execroot. In particular, all files they need during execution that might be generated MUST be addressed via appropriate embedded path (from generated files) obtained by `$location` expansion of labels.

The reason for that design choice is that generating the runfiles layout is expensive, especially on systems where creating symbolic links to files is not easily possible and the respective files would have to be copied.

To still unify the directories seen by actions and `bazel run`, we plan to standardize on the execroot layout, i.e., the way actions see files. As this part of the proposal is more controversial, it can be implemented later; moreover, there will be a transition period of sufficient length where at `bazel run` generated files are visible in both ways, directly via the respective package-indicated path, as well as under `bazel-out`.

Repository name mapping

The top-level `WORKSPACE` MAY make arbitrary declarations of the form "What `@repoA` calls `@repoB` should refer to `@repoC` in my namespace" in a syntax still to be specified, provided that for each pair `(@repoA, @repoB)` there is at most once such declaration. It is RECOMMENDED to use such declarations only when necessary and use the default names otherwise. Entities sharing a common cache SHOULD agree on a top-level `WORKSPACE` declaration to increase the usefulness of the shared cache.

As Labels are always evaluated in the context of some repository, such mappings can be handled in the way Skylark interprets those labels. Repositories accessing resources from a different repository MUST do so via appropriate qualified labels and the `$location` expansion of those. In particular, repositories MUST NOT assume that the referred-to resource is located below a directory containing the repository name as in the label used, or make other assumptions about the layout on the file system (unless explicitly asserted in this document); such assumption would prevent the repository-name mapping just described.

Execroots for each repository

We keep the concept of referring to internal resources via a local relative paths. That is, actions MAY continue to assume that dependencies that are truely source files from their own repository are located at the package-indicated path. Dependencies that may be generated MUST be referenced via the corresponding `$location` expansion; packages SHOULD NOT assume that labels outside their own package are source files.

In particular, we do _not_ force all repositories into a common namespace. So single repository builds will not be changed at all. Note that forcing everyone into a global namespace would imply quite some effort for everyone as repositories cannot hard-code their own name, as the top-level work space including them may assign them a different name to avoid conflicts.

Repository-independent paths

We call a path _repository independent_, if it can be interpreted in the execroot of any of the involved repositories and still refers to the same file. Obviously, absolute paths are repository independent, but there are also repository-independent paths that only depend on the top-level `WORKSPACE`; if bazel lays out the execroots in such a way that they all have the same distance from their closest (hence any) common ancestor directory, a relative path starting with enough `../` will also be repository independent.

`$location` for qualified labels expands repository-independent

Whenever a label that is qualified with a repository is expanded to a location, a repository-independent path is returned. We allow referring to a label in the same repository in a qualified way (using the name the repository refers to itself, as specified in its `WORKSPACE` specification).

Bazel will ensure that those repository-independent paths will depend only on the top-level work

space definition, not the location of the checkout, the user name, or similar. In particular, actions MAY embed those paths in the generated artifacts and still consider themselves hermetic.

Callee responsible for being callable repository-independent

Environment in which actions are executed

When bazel calls an executable via `ctx.action.run` or an equivalent mechanism, it does so with the working directory set to the execroot of the repository the action belongs to. In particular, all repository-independent paths are valid ways to address the respective file.

Bazel also sets `argv[0]` to a value that, interpreted relative to the working directory, names the executable.

Responsibilities of the caller

When an action calls an executable from a different repository it MUST do so from the execroot of one of the involved repositories. Whenever they pass paths as arguments these paths MUST be correct when interpreted from the directory where the executable is called. Note that actions are executed in the execroot of the repository they belong to, so not changing the working directory before calling the executable of a different repository is a correct implementation.

Actions SHOULD set `argv[0]` to a value pointing to the executable they are calling, unless the executable expects to be called differently (e.g., because it is dispatching on `argv[0]` in a `busybox`-like fashion). Again, calling the foreign executable via `ctx.action.run` sets `argv[0]` correctly.

Responsibilities of externally visible executables

Executables visible from outside the repository they belong to MUST be prepared to be called from a different execroot. They MAY do so by embedding a repository-independent non-absolute path to their own execroot. Note that this obligation only applies to visible executables, so if a visible executable itself calls a tool internal to its repository, it MUST do so

from the correct execroot; this obligation to change the directory also implies the obligation to appropriately rewrite all paths passed to this internal tool. If a tool is known to use paths only to read or write the referenced file but does not embed those paths anywhere, then computing the absolute paths for those arguments before changing the directory is fine.

While executables in most cases are called with a meaningful `argv[0]`, they MUST NOT do computations based on `argv[0]` if they are visible from outside their repository.

The same obligation to be prepared to be callable from a different execroot applies to rules visible from outside their repository, in particular, if the rule-repository comes with some form tool (e.g., a preprocessor, compiler, or similar language-specific tool). One way to deal with this obligation is to only use repository-independent relative paths to the tools in the construction of the action command line.

Example

C

As C references other files by names, but relative to a given _set_ of paths that serve as roots for includes, the only use of repository-independent paths is on the command-line for the `-I` options. The file layout, if we do minimal changes with respect to the current layout, is as follows.

https://drive.google.com/file/d/1quTm1YTo-dqxB-aqunoRZZwEGBjMxnwS/view?usp=sharing

However, as rules must get the names of the relevant workspace-independent paths to the roots from Skylark, they must not make any assumptions about the relative locations of those roots. So the layout on disk may as well look as follows.

https://drive.google.com/file/d/1I2rSfshgPHBEaKJMt2Auws6kr4pS_ite/view?usp=sharing

In fact, we plan to move to that layout (or a similar one, in any case one with the include roots being incomparable on disk) in the future; again, due to google's internal legacy code base the time frame is completely unclear and a long transition period may happen.