Skylark Remote Repositories

http://goo.gl/OZV3o0
Status: implemented
Author: dmarting@
Reviewers: lberki@, laurentlb@, kchodorow@
Publicly visible

Context

Skylark is the extension language for Bazel and lets Bazel users describe the build for new languages easily. External users do not create native rules and we want to avoid them doing so.
Remote repositories are a convenient way to specify your third party dependencies and to fetch them along with the build if you don't want to check them in your repository.
This document discuss "Skylark Remote Repositories", that is creating new remote repository rules using Skylark.

Why?

- Enable users to specify new kind of repositories, we already have requests for PyPI for example. We don't want to be supporting every kind of repository that exists outside.
- Enable Skylark rules to write macros to have one-liners for including all their tools in your WORKSPACE file.
- Enable configuration of languages tooling: `bazel init` first approach as a separate tools is not really user-friendly and the same kind of flexibility can be achieved by creating repositories rule in Skylark. An example for the JDK is here.

User interface (see the JDK example)

The load statement will now be available from the WORKSPACE, working the same way it does for build file but with WORKSPACE specific functions instead.

In the same way that we have macros and rules for the BUILD file, we are going to have macros and rule for the WORKSPACE file. The former will be a convenient way to combine remote repositories and the latter enable creation of new repositories kind.

Macros

Skylark macros would be activated in the WORKSPACE file and would behave as expected. Macros would enable to combine remote repositories creation and bind into a single rules E.g. `setup_java()` would set-up all bindings and the local repository needed to build java target:

```
def setup_java():
  native.new_local_repository(name = “jdk-local”, path = “/usr/share/java/jdk8”, build_file
= “jdk.BUILD”)
  for target in ["jni_header", "jni_md_header", "langtools", "bootclasspath", "extdir",
"toolchain", "jdk", "java", "javac", "jar"]:
    native.bind(name=target, actual="@%s//:%s" % (name, target))
  native.bind(name="jni_md_header-linux", actual="@%s//:jni_md_header" % name)
  native.bind(name="jni_md_header-darwin", actual="@%s//:jni_md_header" % name)
```

Remote repository rule

A remote repository rule would be set-up the same way we set-up a build rule but with the `repository_rule` statement:

```
jdk_repository = repository_rule(
  implementation = my_impl,
  attrs = {
    “java_home”: attr.string(mandatory=False),
    “java_version”: attr.string(default=“1.8”),
  }
```

This statement takes only 2 arguments: an implementation function and a list of attributes. The syntax is similar to the rule statement but attributes can only takes primitive type (String, Label, Integer, Boolean, …) and not artifacts.

The implementation function takes exactly one argument: the repository context. This context will provides many convenience methods for doing non hermetic operations, e.g., :

- For discovering the environment:
  - access system environment (`ctxt.os`),
  - execute a program and get the standard output[1] (`ctxt.execute`),
  - …
- For creating the remote repository:
  - fetch an artifact from URL (`ctxt.download`),
  - uncompress an artifact (`ctxt.path(...).uncompress(outputPath)`),
  - “copy” a directory from the system (ctx.`fetch_path(...)`),
  - create a build file (`ctxt.build_file(...)`)
  - ...

The precise list of methods the repository context will support will be augmented on-demand depending on what makes sense for our users.

How?

A preliminary quick and dirty prototype can be found here and here.
Here what the prototype does:

1. First commit activate Skylark macros and repositories

[1] This execution is designed only for discovering the environment, not for creating the remote repository. This might be reconsidered in the future depending on the usage.

    a. Allow Skylark load statements in the WORKSPACE file by adding the various hook and a WorkspaceContext.
    b. A new repository_rule in Skylark that can be called only from the WORKSPACE file.
    c. A new repository context that is passed to repository rule and that should contain all the non-hermetic stuff so the rest of skylark stays hermetic.
    d. A bit of hack for tweaking the SkylarkNativeModule when in WORKSPACE file to comply with the structure of the WORKSPACE rules.
    e. A dirty hack to load the SkylarkRepositoryFunction as a Skylark module without breaking the package boundaries. This is due of technical debts on loading Skylark module nicely (there is a TODO to do it correctly).
2. Second commit showcase the usage of Skylark remote repositories as a configuration step.
    a. Add an example for fetching JDK dependencies. It does both the detection and the fetching.
    b. Add the necessary methods in the SkylarkRepositoryContext for making the example work.
    c. Added the search method to the Skylark string object (to do a regex search).

Roadmap

The obvious choice for the roadmap is to remake all those works, correctly commented and tested, and then add methods to the SkylarkRepositoryContext for full support.
More precisely the correct order of the work should be:

1. Activate Skylark Macros taking part of 1.a and doing correctly 1.d [DONE]
2. Fix Skylark module load (1.e) [DONE]
3. Add the SkylarkRepositoryFunction and empty context (1.b and 1.c) [DONE]
4. Extends SkylarkRepositoryContext for handling C++ configuration [DONE].
5. Extends SkylarkRepositoryContext [DONE] for handling PyPI
6. Document [DONE]
7. Extends SkylarkRepositoryContext for handling Docker pull