Bazel Blog

Scalable Android Builds with Incremental Dexing (/2018/02/28/incremental-dexing.html)

28 February 2018

Bazel supports building Android apps with Java and C++ code out of the box through the `android_binary` (https://docs.bazel.build/versions/master/be/android.html#android_binary) rule and related rules. Android binary builds need a lot of machinery--more than we can cover in a blog post. However, one aspect that's fairly important to Bazel's Android support is scalability. That's because we build most of Google's own Android apps with Bazel and those apps are not only comparably large but also come with hundreds of engineers that want to build and test their changes quickly.

For over a year now, Bazel has used a feature we call **incremental dexing** to speed up Android builds. As the name implies, incremental dexing is designed to minimize the work needed to rebuild an app after code changes, but it also parallelizes builds and lets them scale better to the needs of Google's own apps. But how does it work and what is "dexing" anyway?

Dexing is what we call the build step that converts Java bytecode to Android's .dex file format. Traditionally, that's been done for an entire app at once by a tool fittingly called "dx". Even if only a single class changed, dx would reprocess the entire app, which could take a while. But dx really has two jobs: compile bytecode to corresponding .dex code, and merge all the classes that are going into the app into as few .dex files as possible. The latter is needed because while Java bytecode uses a separate file per class, a single .dex file can contain thousands of classes. But because of the differences in instruction encoding, the compilation step is the more time-consuming one, while merging can be done separately and relatively quickly even for large apps.

Incremental dexing, as you might have guessed, separates bytecode compilation and dex merging. Specifically, it runs the compilation step separately in parallel for each .jar file that's part of the app's runtime classpath. To arrive at a final app, Bazel then merges the compilation results from each .jar.

How does that help? In a number of ways:

1. We can take advantage of the parallelism inherent in the build and farm out compilation to as many processes as there are .jars in the app.
2. When rebuilding after making code changes, we only have to recompile the .jars that changed, avoiding potentially a whole lot of work that doesn't change the output.
3. We can re-use compilation results in cases where the same .jar is part of multiple apps (for example, common libraries or just to build test inputs).
4. We can also parallelize other build steps that are needed as inputs for the merge step.
5. With the `--experimental_spawn_scheduler` flag, it simplifies caching past dexing results (for example, when one class in a large .jar changes).
6. Most importantly, this strategy scales much better for large apps.

What we mean by scalability is that the total number of classes in the app matters much less for how long it takes to build the app. This is especially important when rebuilding the app after small changes: with incremental dexing, the time spent on dexing is proportional to the size of the change. Previously, dexing time was always proportional to the number of classes in the app, no matter the change. This scalability has been critical in keeping up with our ever-growing apps.

One prerequisite for taking full advantage of incremental dexing is to split up the app into multiple, ideally small, .jars. Bazel naturally encourages and enables this with the `java_library` (https://docs.bazel.build/versions/master/be/java.html#java_library) and `android_library` (https://docs.bazel.build/versions/master/be/android.html#android_library) rules, which build .jar files from a set of Java sources, conventionally for a single Java package at a time. Third-party libraries are also often distributed as .jar or .aar files that Bazel ingests with the `java_import` (https://docs.bazel.build/versions/master/be/java.html#java_import) and `aar_import` (https://docs.bazel.build/versions/master/be/android.html#aar_import) rules.

Tooling-wise it was reasonably straightforward to separate the merging step because Android provides a tool called dexmerger for just this purpose and compiling separately more or less just means running dx on one class at a time. One wrinkle for now is that dexmerger creates final .dex files that are larger than necessary. That means you want to turn off incremental dexing when you're building a binary that you want to give to users, but it can speed up your development and test builds every day with no known adverse effect. Plus, we expect this to get better since Android Studio has started to use a similar scheme to build Android apps in Gradle.

By Kevin Bierhoff (https://github.com/kevin1e100)

Bazel 0.11 (/2018/02/26/bazel-0.11.html)

26 February 2018

The Bazel team is happy to announce the release of version 0.11.0 (https://github.com/bazelbuild/bazel/releases/tag/0.11.0).

Notable Changes

- android_binary (https://docs.bazel.build/versions/master/be/android.html#android_binary) targets built with ProGuard can now enjoy the benefits of incremental dexing (https://blog.bazel.build/2018/02/28/incremental-dexing.html) by specifying the attribute `incremental_dexing = 1`.
- aar_import (https://docs.bazel.build/versions/master/be/android.html#aar_import) rules now support importing assets (https://github.com/bazelbuild/bazel/issues/4439) from the `assets/` subdirectory of the AAR being imported.
- Remote caching now supports HTTP Basic Authentication (https://github.com/bazelbuild/bazel/commit/cf3f81aef7c32019d70cbce218a64a03276268f0).
- You can now depend on `@bazel_tools//tools/runfiles:java-runfiles` to get a platform-independent runfiles library for Java. See JavaDoc of Runfiles.java (https://github.com/bazelbuild/bazel/blob/master/src/tools/runfiles/java/com/google/devtools/build/runfiles/Runfiles.java) for usage information.

Community Updates

Here are some updates on what happened in the Bazel community over the past month.

Languages & Rules

- You can use Bazel to build Rust with rules_rust (https://github.com/bazelbuild/rules_rust) for the Rust compiler toolchain and cargo_raze (https://github.com/google/cargo-raze) to resolve Cargo.toml dependencies into WORKSPACE rules. See cargo_raze_crater (https://github.com/acmcarther/cargo-raze-crater) for an example.
- Marcel Hlopko published the C++ rules roadmap (https://docs.google.com/document/d/1K3Dq1JDDWjGtvTyFcIwy_-Crm7ZuD9FzSxjDawDhJRA/edit#heading=h.feujy6f2j7mi), which is open for comments.
- Jay Conrod published a roadmap (https://github.com/bazelbuild/rules_go/blob/master/roadmap.rst) for the Go rules and Gazelle. If you are interested in contributing or have a question, head over to the bazel-go-discuss (https://groups.google.com/forum/#!forum/bazel-go-discuss) mailing list.
- VSCO released the latest version of their C++ CROSSTOOL (https://github.com/vsco/bazel-toolchains) using Chromium's LLVM toolchain.
- The Android build process involves many moving parts and can be complex. Alex Steinberg wrote here a detailed walkthrough on How Android Builds Work in Bazel (https://blog.bazel.build/2018/02/14/how-android-builds-work-in-bazel.html).

Tools

- Justine Tunney made an experiment about turning a Bazel BUILD file into a Makefile (https://gist.github.com/jart/082b1078a065b79949508bbe1b7d8ef0) using the results of bazel query (https://docs.bazel.build/versions/master/query-how-to.html).

Community

- Nick Santos wrote a blog post on whether you should adopt Bazel for Go projects: Bazel is the Worst Build System, Except for All the Others (https://medium.com/windmill-engineering/bazel-is-the-worst-build-system-

except-for-all-the-others-b369396a9e26)
- Filip Nikolovski wrote an article about how InPlayer is Managing a Go monorepo with Bazel (https://filipnikolovski.com/managing-go-monorepo-with-bazel/).
- Alex Eagle of the Angular core team did a 2-hour long deep dive of Angular's Buildtools Convergence (https://www.youtube.com/watch?v=z9Q_2N9oaG8) (ABC).
- Kyle Cordes also gave a demo (https://www.youtube.com/watch?v=KmaE6z_ECRg) on building Angular applications with Bazel at Angular Lunch.

Did we miss anything? Fill the form (https://docs.google.com/forms/d/e/1FAIpQLSde7NGMKA1xK2RZnOLk8XKm3A-Y09guJAFrkX35RCJxn3RB4w/viewform?usp=sf_link) to suggest content for a next blog post.

Discuss on Hacker News (https://news.ycombinator.com/item?id=16468244) or Reddit (https://www.reddit.com/r/bazel/comments/80focy/bazel_011/).

By Jingwen Chen (https://github.com/jin)

How Android Builds Work in Bazel (/2018/02/14/how-android-builds-work-in-bazel.html)

14 February 2018

Background: How Bazel Works

In Bazel, `BUILD` files in directories specify **targets** that can be built from the contents of those directories.

Bazel goes through three steps (https://docs.bazel.build/versions/master/user-manual.html#phases) when building targets:

1. In the **loading** phase, Bazel parses the `BUILD` file of the target being built and all `BUILD` files that file transitively depends on.
2. In the **analysis** phase, Bazel builds a graph of actions needed to build the specified targets.
3. In the **execution** phase, Bazel runs those actions.

Each Bazel target is defined by a **rule**, which specifies inputs, outputs, and how to get from one to the other. Rules can specify things like creating an executable binary or defining a library. In Bazel's code, individual rules are represented by instances of implementations of `RuleConfiguredTargetFactory` (https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/analysis/RuleConfiguredTargetFactory.java). Users can also extend Bazel and create new rules with Skylark (https://docs.bazel.build/versions/master/skylark/concepts.html).

Rules create, in turn, any number of **action**s. Each action takes any number of **artifact**s as inputs and produces one or more artifacts as outputs. These artifacts represent files that may not yet be available. They can either be *source artifacts*, such as source code checked in to the repository, or *generated artifacts*, such as output of other actions. For example, an action to compile a piece of code might take in source artifacts representing the code to be compiled and generated artifacts representing compiled dependencies, even though those dependencies have not yet been compiled, and output a generated artifact representing the compiled result. Additionally, rules may expose any number of **provider** objects. These providers are the API rules provide to other rules. They provide read-only information about internal state.

During analysis, Bazel runs the rules for each target being built and their transitive dependencies. Each rule generates and records all the actions it depends on. Bazel won't necessarily run all of those actions; if an action doesn't end up being required, Bazel will just ignore it. **Skyframe** is used to evaluate and cache the results of rules.

Information from the rules, including artifacts representing the future output of actions, are made available to other rules through the rules' providers. Each rule has access to its direct dependencies' providers. Because most information passed between rules is actually transitive, providers make use of the `NestedSet` (https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/collect/nestedset/NestedSet.java) class, a DAG-like data structure (it's not actually a set!) made up of items and pointers to other (nested) `NestedSet` objects. `NestedSet` s are specially optimized to work efficiently for analysis. For part of a provider that represents some transitive state, for example, a trivial implementation might be to build a new list that contains the items for the current rule and each transitive dependency (for a chain of n transitive dependencies, that means we'd add n + (n

- 1) + … 1 = O(n^2) items to some list), but building a nested set containing the new item and a pointer to the previous nested set is much more efficient (we'd add 2 + 2 + … 2 = O(n) items to some nested set). This introduces similar efficiency in memory usage as well.

Artifacts can each be added to any number of **output groups**. Each output group represents a different group of outputs that a user might choose to build. For example, the `source_jars` (https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/rules/java/JavaSemantics.java#L132) output group specifies that Bazel should also produce the JARs of the source for a Java target and its transitive dependencies. The special default (https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/analysis/OutputGroupInfo.java#L113) output group holds output that is specifically built for a target - for example, building a Java binary might produce a compiled `.jar` file in the default output group.

During execution, Bazel first looks at the artifacts in the requested output groups (plus, unless the user explicitly requested otherwise, the default output group). For each of those artifacts, it finds the actions that generate the artifact, then each of the artifacts each of those actions need, and so on until it finds all the actions and artifacts needed. If the action is not cached or the cache entry should be invalidated, Bazel follows this same process for the action's dependencies, then runs the action. Once all of the actions in the requested output groups has been run or returned from cache, the build is complete.

Android Builds

There are a few important kinds of rules when building code for Android (https://docs.bazel.build/versions/master/be/android.html):

- `android_binary` (https://docs.bazel.build/versions/master/be/android.html#android_binary) rules build Android packages (`.apk` files)
- `android_library` (https://docs.bazel.build/versions/master/be/android.html#android_library) rules build individual libraries that binaries and other libraries can consume.
- `android_local_test` (https://docs.bazel.build/versions/master/be/android.html#android_local_test) rules run test on Android code in a JVM.
- `aar_import` (https://docs.bazel.build/versions/master/be/android.html#aar_import) rules import `.aar` libraries built outside of Bazel into a Bazel target.

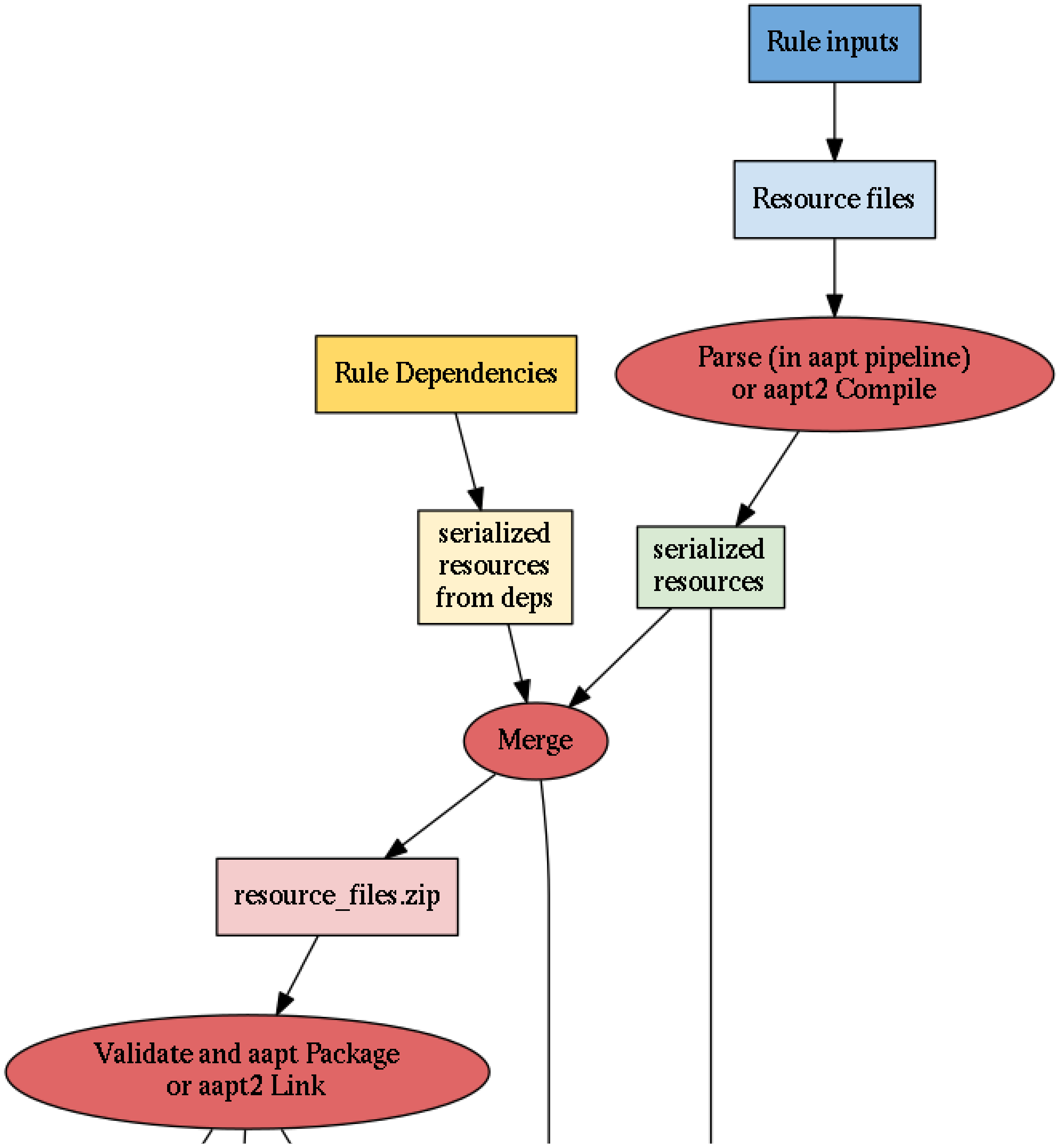
Android Resources

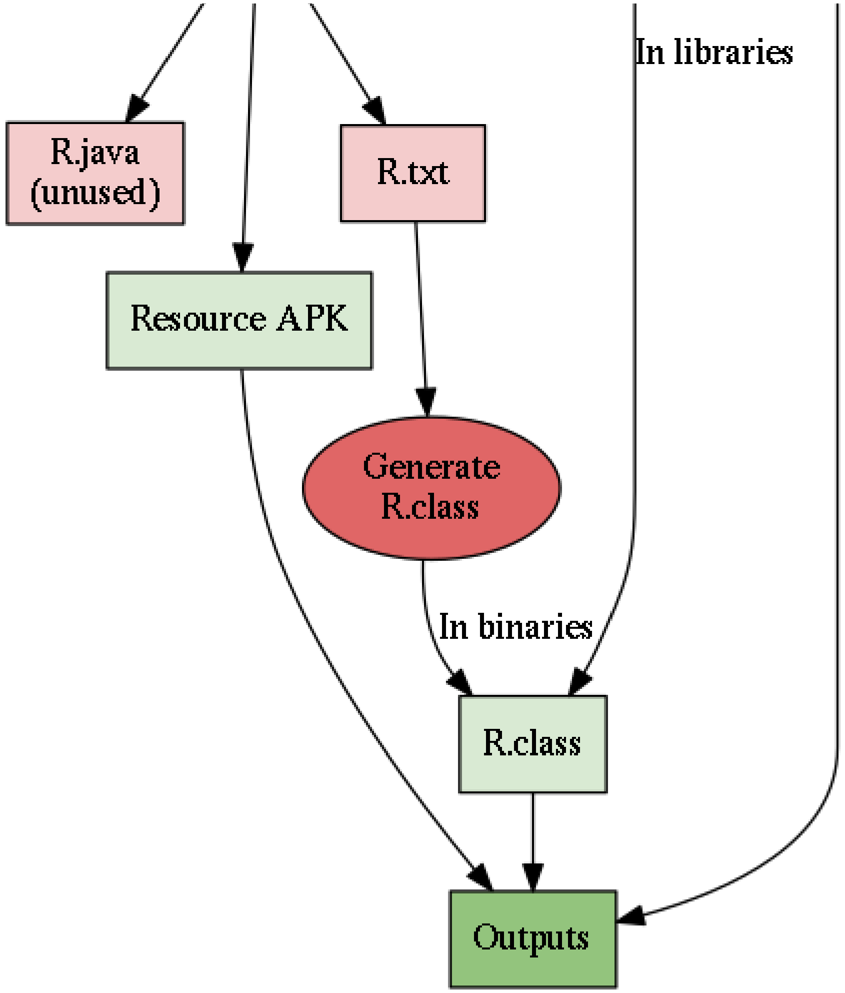
One way in which the Android library build process differs from the normal Java build process is Android **resources** (https://developer.android.com/guide/topics/resources/index.html). Resources are anything that's not code - strings, images, layouts, and so on (https://developer.android.com/guide/topics/resources/providing-resources.html#ResourceTypes).

Bazel generates **`R.java`** files (as well as related `R.class` and `R.txt` files) to contain references to available resources. These R files contain integer **resource ID**s that developers can use to refer to their resources. Within an app, each resource ID refers to one unique resource.

Developers can provide different versions (https://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources) of the same resource (to support, for example, different languages, regions, or screen sizes). Android makes references to the base resource available in the R files, and Android devices select the best available version of that resource at runtime.

Resource processing with `aapt` and `aapt2`


Rule inputs
Resource files
Parse (in aapt pipeline) or aapt2 Compile
Rule Dependencies
serialized resources from deps
serialized resources
Merge
resource_files.zip
Validate and aapt Package or aapt2 Link



Bazel supports processing resources using the original Android resource processor, `aapt`, or the new version, `aapt2`. Both methods are fundamentally similar but have a few important differences.

Bazel goes through three steps to build resources.

First, Bazel serializes the files that define the resources. In the `aapt` pipeline, the **parse** action serializes information about resources into `symbols.bin` files. In the `aapt2` pipeline, an action calls into the **`aapt2` compile** command which serializes the information into a format used by `aapt2`.

Next, the serialized resources are **merged** with similarly serialized resources inherited from dependencies. Conflicts between identically named resources are identified and, if possible, resolved during this merging. The contents of `values` resource files are generally explicitly merged. For other files, if resources from the target or its dependencies have the same name and qualifiers, the contents of the files are compared and, if they are different, a warning is produced and the resource that was provided last is chosen to be used.

Finally, Bazel checks that the resources for the target are reasonable and packages them up. In the `aapt` pipeline, the **validate** action calls into the `aapt` **package** command, and in the `aapt2` pipeline, the **`aapt2` link** command is called. In both cases, any malformed resources or references to unavailable resources cause a failure, and, if no failures are encountered, `R.java` and `R.txt` files are produced with information about the validated resources, and a Resource APK containing those resources is produced.

Using `aapt2` rather than `aapt` provides better and more efficient support for a variety of cases. Additionally, more of the resource processing steps are handled by `aapt2` as opposed to Bazel's custom resource processing tools. Finally, since the serialized format can be understood as-is by future calls to `aapt2`, Bazel no longer has to deserialize information about resources to a form `aapt2` can understand.

The resource ID values generated for `android_library` targets are only temporary, since higher-level targets might depend on multiple targets where different resources were assigned the same ID. To ensure that resource IDs aren't persisted anywhere permanent, the R files record the IDs as nonfinal, ensuring that compilation doesn't inline them into other Java code. Additionally, an `android_library`'s R files should be discarded after building is complete.

(Even though `android_library` files are eventually discarded, we still need to run resource processing to generate a temporary `R.class` to allow compilation, to merge resources so they can be inherited by consumers, and to validate that the resources can be compiled correctly - otherwise, if a developer introduces a bug in their resource definitions, it won't be caught until they're used in an `android_binary`, resulting in a lot of wasted work done by Bazel.)

Code in android libraries and binaries make references to code in the R files, so the `R.class` file must be generated before regular compilation can start. For `android_library` targets, since all resource IDs are temporary anyway, we can speed things up by generating a `R.class` file at the end of resource merging. For `android_binary` targets, we need to wait for the output of validation to get correct resource IDs. Validation does produce an `R.java` file, but **generating an `R.class`** file directly from the contents of the `R.txt` file is much faster than compiling the `R.java` file into an `R.class` file.

Android Resource Optimizations

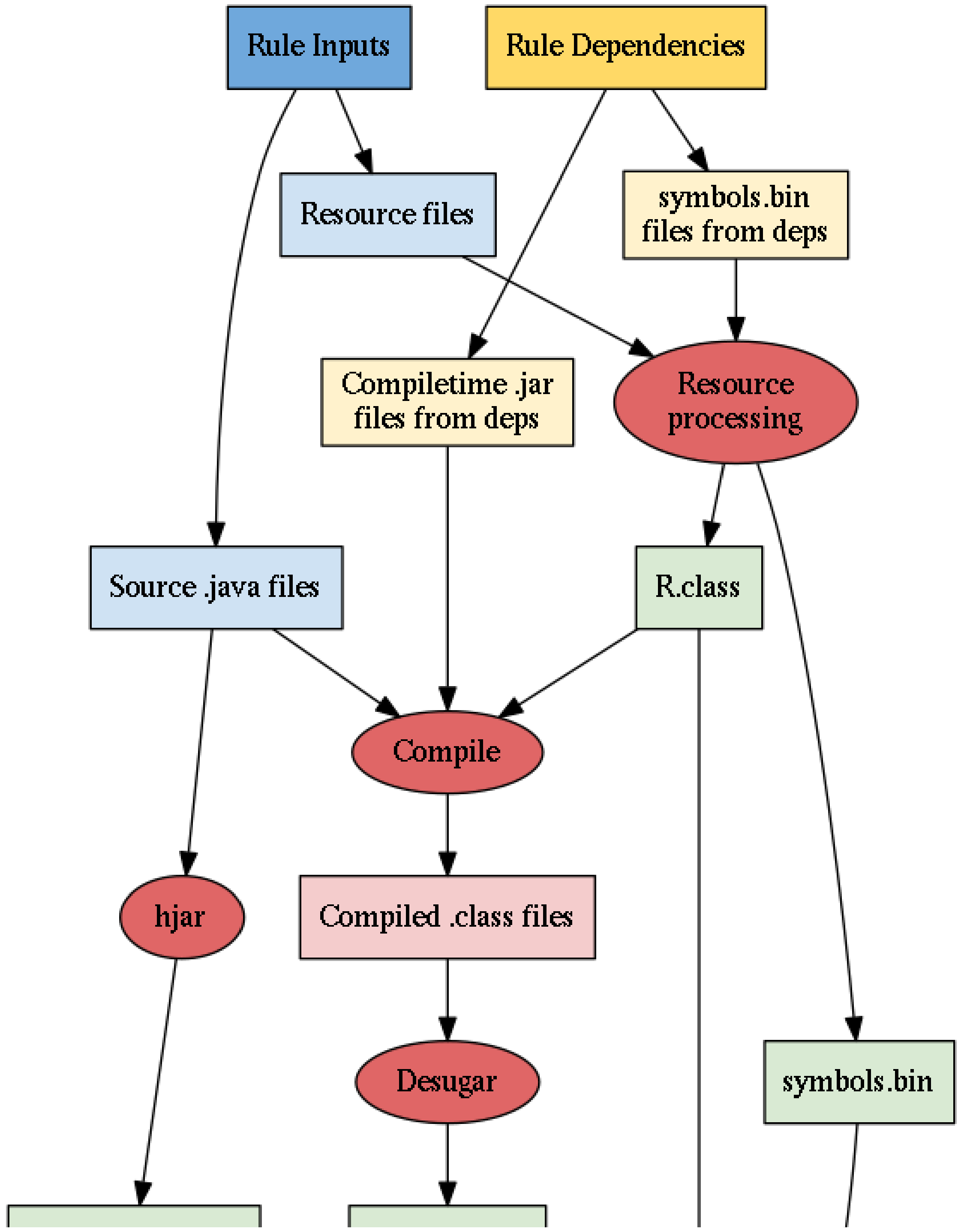
Resource Filtering

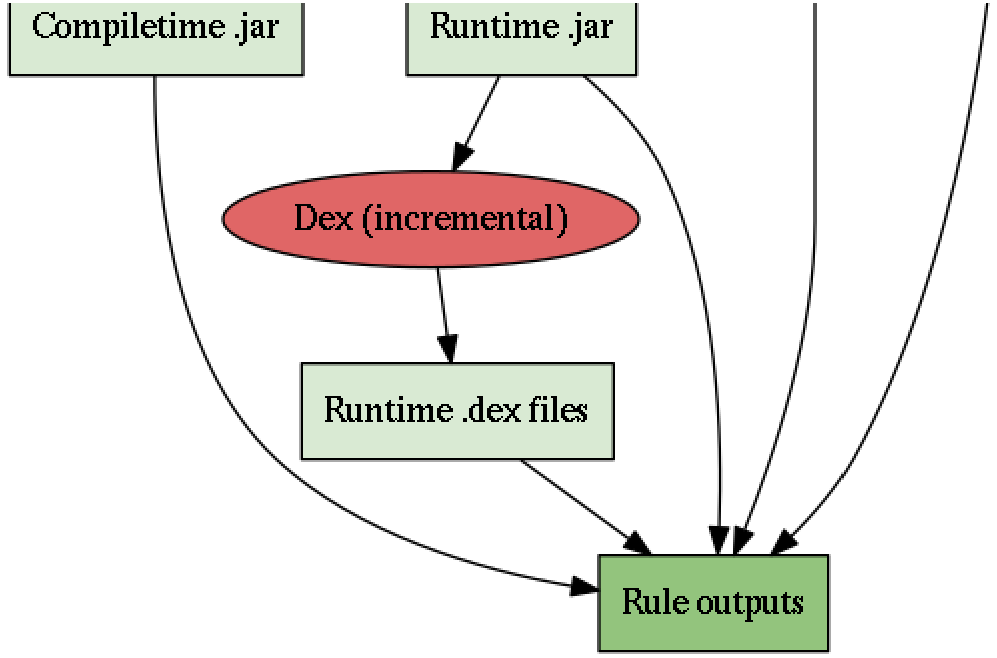
The `android_binary` rule includes optional `resource_configuration_filters` (https://docs.bazel.build/versions/master/be/android.html#android_binary.resource_configuration_filters) and `densities` (https://docs.bazel.build/versions/master/be/android.html#android_binary.densities) fields. These fields limit the types of devices that will be built for. For example, if you only wanted to build for English-language devices with HDPI displays, you could specify:

```
android_binary(
  # ...
  densities = ["hdpi"],
  resource_configuration_filters = ["en"],
)
```

Bazel will now be able to skip unneeded resources. As a result, the build will be faster and the resulting APK will be smaller. It won't support all kinds of devices and user preferences, but this speed improvement means developers can build and iterate faster.

Android Libraries


Rule Inputs
Rule Dependencies
Resource files
symbols.bin files from deps
Compiletime .jar files from deps
Resource processing
Source .java files
R.class
Compile
hjar
Compiled .class files
Desugar
symbols.bin



An `android_library` (https://docs.bazel.build/versions/master/be/android.html#android_library) rule is a pretty simple rule that builds and organizes an android library for use in another Android target. In the analysis phase, there are basically three groups of actions generated:

First, Bazel processes the library's resources, as described above.

Next comes the actual compilation of the library. This mostly just uses the regular Bazel Java compilation path. The biggest difference is that the `R.class` file produced in resource processing is also included in the compilation path (but is not inherited by consumers, since the R files need to be regenerated for each target).

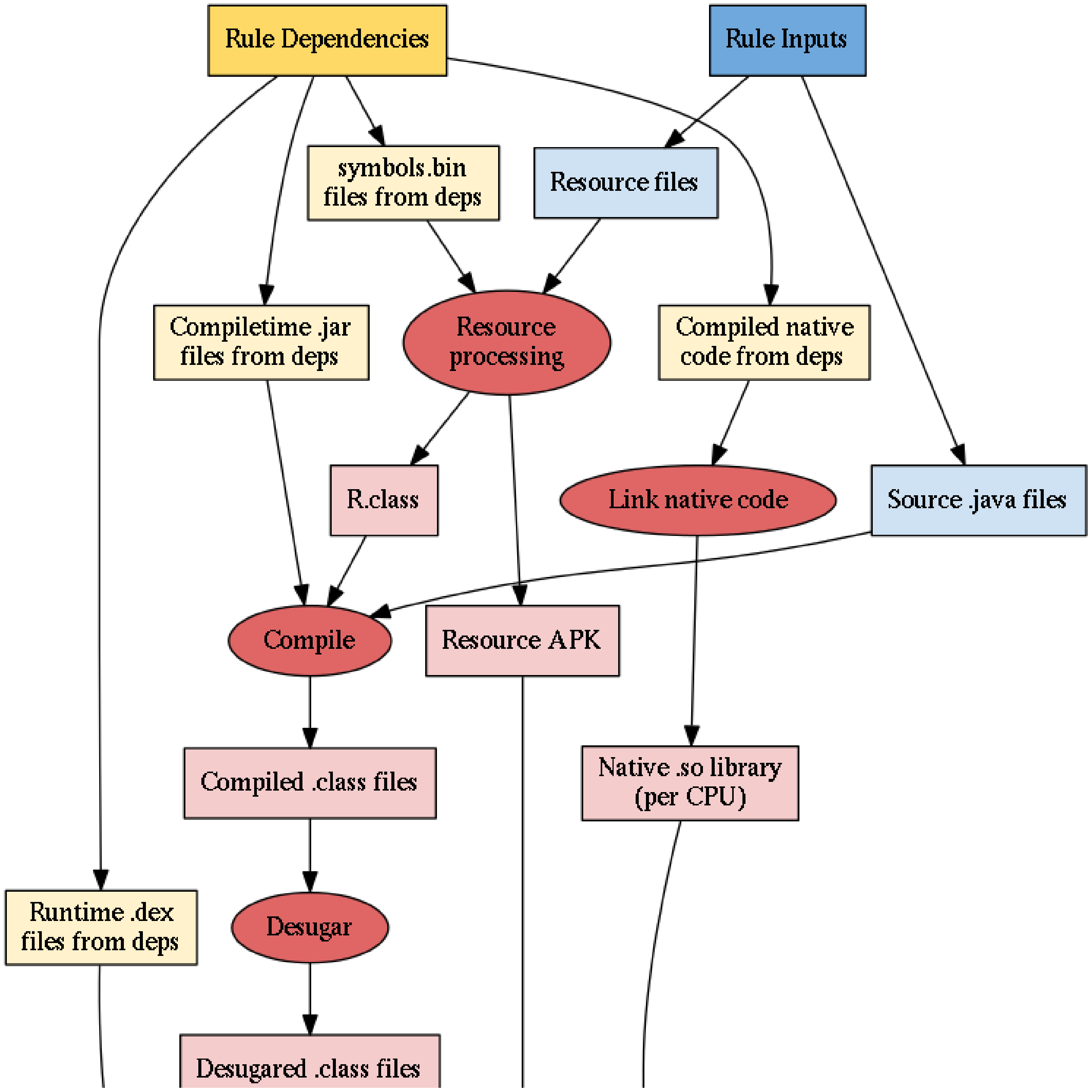
Finally, Bazel does some additional work on the compiled code:

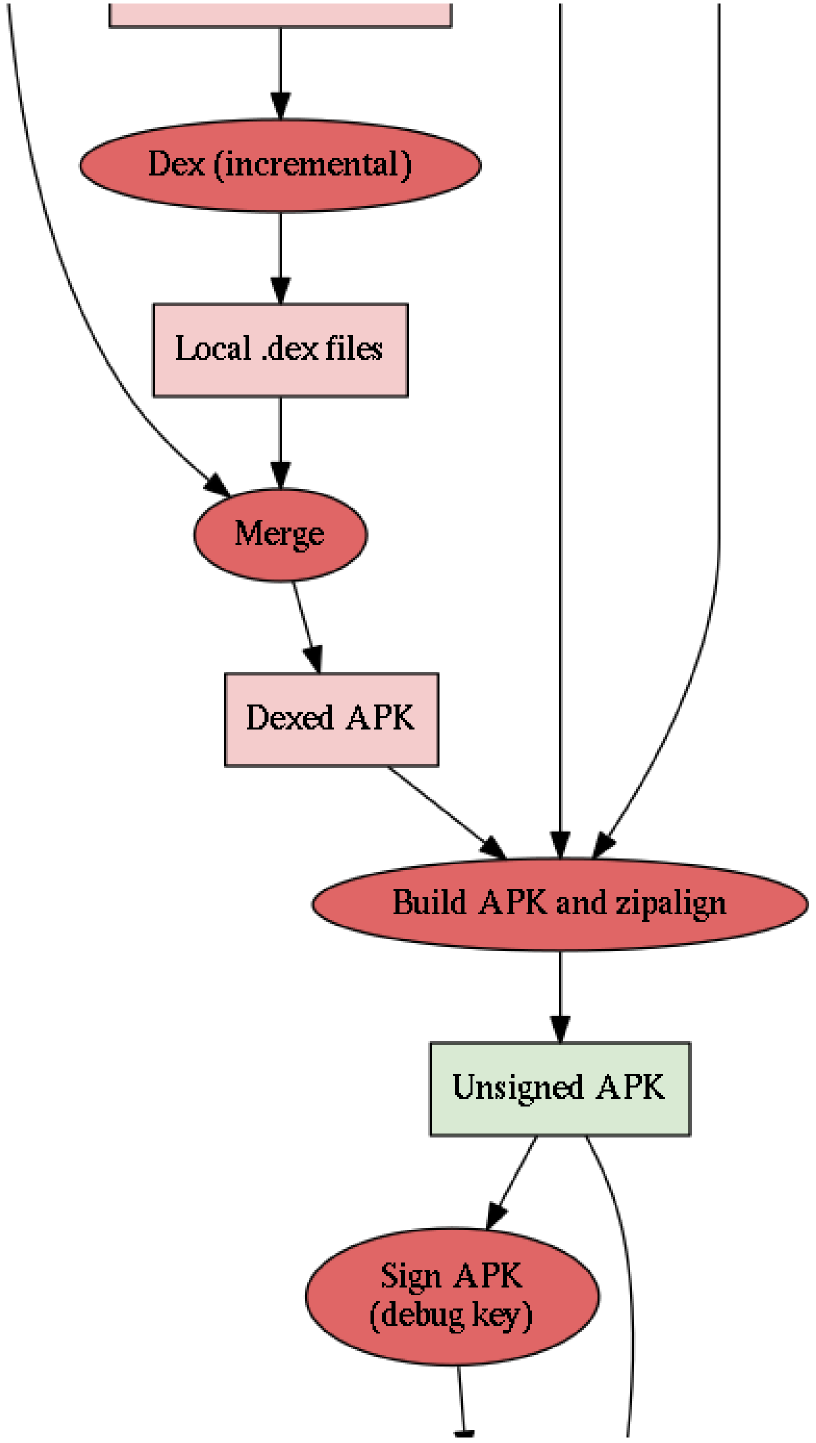
1. The compiled `.class` files are desugared (https://github.com/bazelbuild/bazel/blob/master/src/tools/android/java/com/google/devtools/build/android/desugar/Desugar.java) to replace bytecode only supported on Java 8 with Java 7 equivalents. Bazel does this so that Java 8 language features can be used for developing the app, even though the next tool, `dx`, does not support Java 8 bytecode.
2. The desugared `.class` files are converted to `.dex` files, executables for Android devices, by `dx`. These `.dex` files are then packed into the `.jar` file used at runtime. These *incremental* *`.dex`* files, produced for

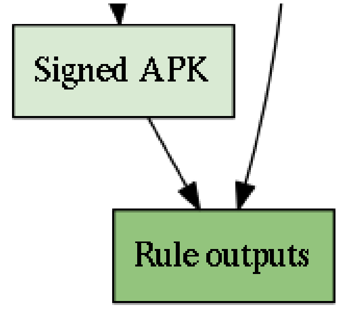
each library, mean that, when some libraries from an app are changed, only those libraries, and not the entire app, need to be re-dexed.

3. The source `.java` files for this library are used by `hjar` to generate a `jar` of `.class` files. Method bodies and private fields are removed from this compile-time `.jar`, and targets that depend on this library are compiled against this smaller `.jar`. Since these jars contain just the interface of the library, when private fields or method implementations change, dependent libraries do not need to be recompiled (they need to be recompiled only when the interface of the library changes), which results in faster builds.

Android Binaries


Rule Dependencies
Rule Inputs
symbols.bin files from deps
Resource files
Compiletime .jar files from deps
Resource processing
Compiled native code from deps
R.class
Link native code
Source .java files
Compile
Resource APK
Compiled .class files
Native .so library (per CPU)
Runtime .dex files from deps
Desugar
Desugared .class files


Dex (incremental)
Local .dex files
Merge
Dexed APK
Build APK and zipalign
Unsigned APK
Sign APK (debug key)



An `android_binary` (https://docs.bazel.build/versions/master/be/android.html#android_binary) rule packages the entire target and its dependencies into an APK. On a high level, binaries are built similarly to libraries. However, there are a few key differences.

For binaries, the three main resource processing actions (parse, merge, and validate), are all combined into a single large action. In libraries, Java compilation can get started while validation is still ongoing, but in binaries, since we need the final resource IDs from validation, we can't take advantage of similar parallelization. Since creating more actions always introduces a small cost, and there's no parallelization available to make up for it, having a single resource processing action is actually more efficient.

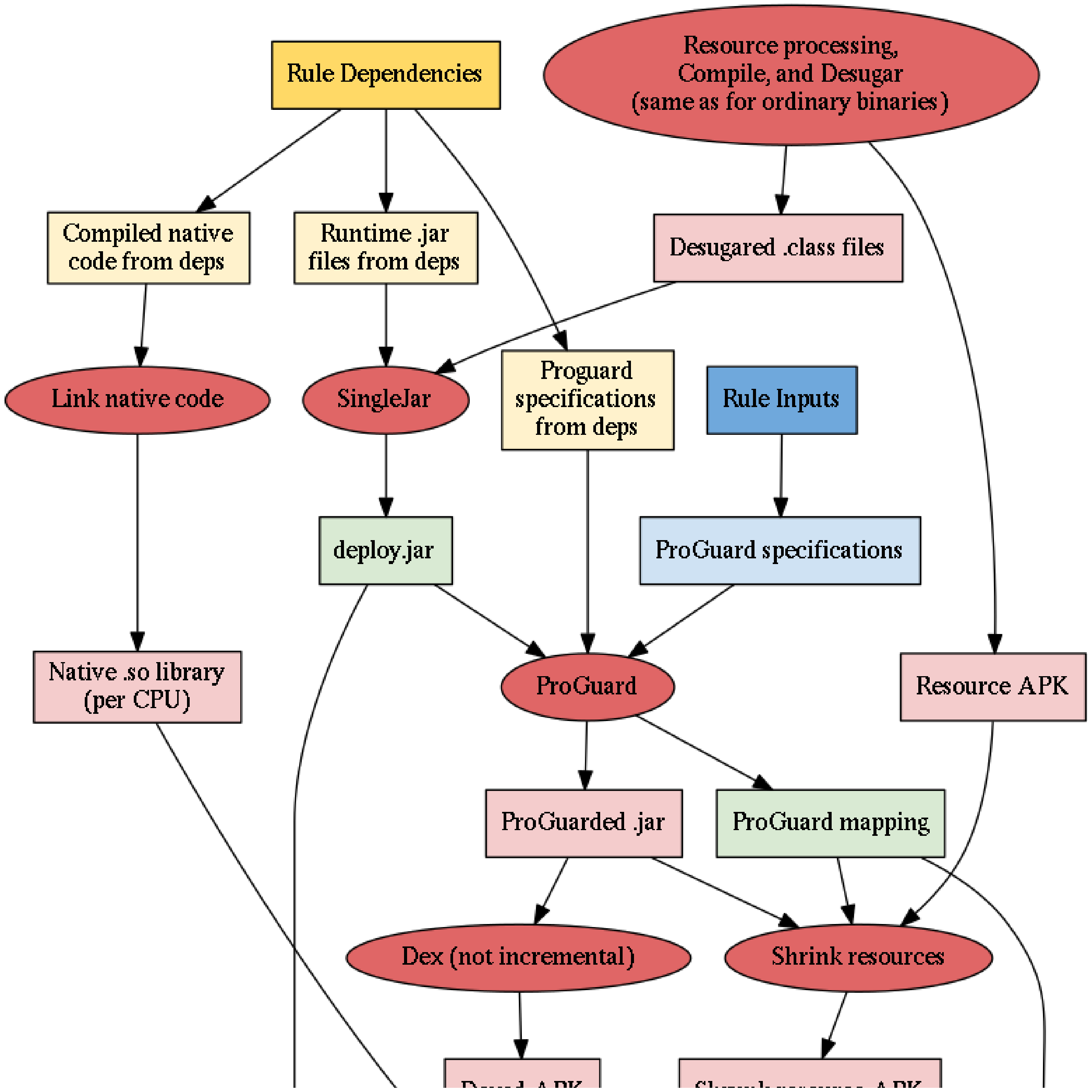
In binaries, the Java code is compiled, desugared, and dexed, just like in libraries. However, afterwards, the `.dex` files from the binary are merged together with the `.dex` files from dependencies.

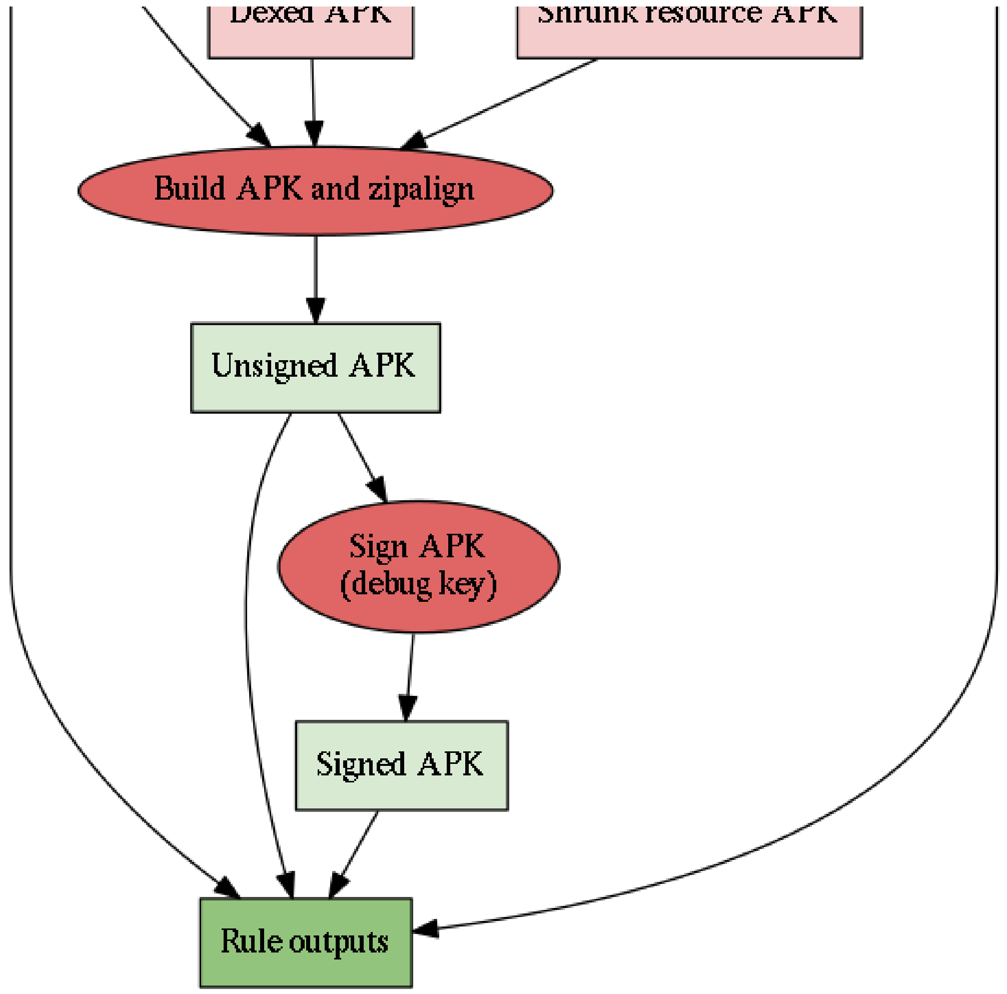
Bazel also links together compiled `C` and `C++` native code from dependencies into a single `.so` file for each CPU architecture specified by the `--fat_apk_cpu` (https://docs.bazel.build/versions/master/user-manual.html#flag--fat_apk_cpu) flag.

The merged `.dex` files, the `.so` files, and the resource APK are all combined to build an initial binary APK, which is then zipaligned (https://developer.android.com/studio/command-line/zipalign.html) to produce an unsigned APK. Finally, the unsigned APK is signed with the binary's debug key to produce a signed APK.

The merged `.dex` files are combined with the resource APK to build an initial binary APK, which is then zipaligned (https://developer.android.com/studio/command-line/zipalign.html) to produce an unsigned APK. Finally, the unsigned APK is signed with the binary's debug key to produce a signed APK.

ProGuarded Android Binaries


Rule Dependencies
Resource processing, Compile, and Desugar (same as for ordinary binaries)
Compiled native code from deps
Runtime .jar files from deps
Desugared .class files
Link native code
SingleJar
Proguard specifications from deps
Rule Inputs
deploy.jar
ProGuard specifications
Native .so library (per CPU)
ProGuard
Resource APK
ProGuarded .jar
ProGuard mapping
Dex (not incremental)
Shrink resources



Bazel supports running ProGuard (https://www.guardsquare.com/en/proguard) against `android_binary` targets to optimize them and reduce their size (https://www.guardsquare.com/en/proguard/manual/introduction). Using ProGuard substantially changes elements of the build process. In particular, the build process does not use incremental `.dex` files at all, as ProGuard can only run on `.class` files, not `.dex` files.

ProGuarding uses a `deploy.jar` file, a single `.jar` file with all of the binary's Java bytecode, created from the binary's desugared (but not dexed) `.class` files as well as the binary's transitive runtime `.jar` files. (This `deploy.jar` file is an output of all `android_binary` targets, but it doesn't play a substantial role in builds without ProGuarding.)

Based on information from a series of Proguard specifications (from both the binary and its transitive dependencies), ProGuard makes serveral passes through the `deploy.jar` in order to optimize the code, remove unused methods and fields, and shorten and obfuscate the names of the methods and fields that remain. In addition to the resulting proguarded `.jar` file, ProGuard also outputs a mapping from old to new names of methods and fields.

ProGuard's output is not dexed, so when building with ProGuard, the entire `.jar` must be re-dexed (even code from dependencies that were dexed incrementally). The dexed code is then built into the APK as usual.

ProGuard will also remove references to unused resources from the class files. If resource shrinking (https://docs.bazel.build/versions/master/be/android.html#android_binary.shrink_resources) is enabled, the resource shrinker uses the proguard output to figure out what resources are no longer used, and then uses `aapt` or `aapt2` to create a new, smaller resource APK with those resources removed. The shrunk resource APK and the dexed APK are then fed into the APK building process, which operates the same as it would without ProGuard.

Mobile-install

Mobile-install (https://docs.bazel.build/versions/master/mobile-install.html) is a way of rapidly building and deploying Android applications iteratively. It's based off of `android_binary`, but has some additional functionality to make builds and deployments more incremental.

By Alex Steinberg (https://github.com/asteinb)

Bazel 0.10 (/2018/02/06/bazel-0.10.html)

06 February 2018

We're proud to announce the release of Bazel 0.10 (https://github.com/bazelbuild/bazel/releases/tag/0.10.0). The 400+ commits since last release include performance optimizations, bug fixes, and various improvements.

There is a new android test rule. `android_local_test` tests `android_library` code using Robolectric, a unit test framework designed for test-driven development without the need for an emulator or device. See the documentation (https://docs.bazel.build/versions/master/be/android.html#android_local_test) for setup instructions and examples.

The depset type has evolved. To merge multiple depsets or add new elements, do not use the operators `+`, `+=`, `|`, or the `.union` method. They are deprecated and will be removed in the future. Instead use the new `depset` constructor, which has a better performance. For example, instead of `d1 + d2 + d3`, use `depset(transitive = [d1, d2, d3])`. See the documentation (https://docs.bazel.build/versions/master/skylark/depsets.html) for more information and examples.

In addition to this new release, the Bazel community has been very active. See below what happened recently.

Languages & Rules

- New Python Interest Group! Evan Jones kickstarted a thread (https://groups.google.com/forum/#!msg/bazel-discuss/NMO6KPyPKh4/VnxtEVP6EQAJ) about Python development on bazel-discuss, which resulted in a new Python special interest group. Read the meeting notes (https://goo.gl/dyd49i) and join the mailing list (https://groups.google.com/forum/#!forum/bazel-sig-python).
- Alex Eagle has been working on a prototype for deploying an Angular app to Kubernetes using Bazel (https://medium.com/@Jakeherringbone/deploying-an-angular-app-to-kubernetes-using-bazel-preview-91432b8690b5).
- Hassan Syed has been working on new Kotlin rules (https://github.com/bazelbuild/rules_kotlin), using persistent workers to significantly improve the performance.
- Paul Johnston created some initial rules for building static websites with hugo (https://github.com/stackb/rules_hugo).
- Harvey Tuch, a contributor on Envoy (https://www.envoyproxy.io/), wrote a blog post about External C++ dependency management in Bazel (https://blog.envoyproxy.io/external-c-dependency-management-in-bazel-dd37477422f5).

Tools

- Spotify released a collection of tools (https://github.com/spotify/bazel-tools) for working with Bazel workspaces, mostly tailored towards writing JVM backend services.

- Ever wanted to save a file and have your tests automatically run? How about restart your webserver when one of the source files change? Take a look at the Bazel watcher (https://github.com/bazelbuild/bazel-watcher).

Performance

- Jason Lunz wrote a nice blog post: We Switched from Maven to Bazel and Builds Got 10x Faster (https://redfin.engineering/we-switched-from-maven-to-bazel-and-builds-got-10x-faster-b265a7845854)
- Mike Morearty wrote an article about How to Create a Persistent Worker for Bazel (https://medium.com/@mmorearty/how-to-create-a-persistent-worker-for-bazel-7738bba2cabb). This technique gave a 3x speedup of their Typescript compilation.
- Nicolò Valigi explained how to get faster Bazel builds with remote caching (https://nicolovaligi.com/faster-bazel-remote-caching-benchmark.html).
- We added a section on Remote Caching (https://docs.bazel.build/versions/master/remote-caching.html) to our website. It's a great way to significantly speed up your CI builds.

Did you know?

- The heart of Bazel is a parallel evaluation and incrementality model called Skyframe (https://bazel.build/designs/skyframe.html).
- Bazel is more than 10 years old, even though it was just open-sourced 3 years ago. John Field goes into the prehistory of Bazel in the opening remarks of Bazel Conference 2017 here (https://youtu.be/3eFllvz8_0k?list=PLxNYxgaZ8RseY0KmkXQSt0StE71E7yizG&t=424).

Did we miss anything? Fill the form (https://docs.google.com/forms/d/e/1FAIpQLSde7NGMKA1xK2RZnOLk8XKm3A-Y09guJAFrkX35RCJxn3RB4w/viewform?usp=sf_link) to suggest content for a next blog post.

Discuss on Hacker News (https://news.ycombinator.com/item?id=16317161) or Reddit (https://www.reddit.com/r/bazel/comments/7vcm1l/bazel_010_released/).

By Laurent Le Brun (https://github.com/laurentlb)

Migration Help: --config parsing order (/2018/01/19/config-parsing-order.html)

19 January 2018

`--config` (https://docs.bazel.build/versions/master/user-manual.html#config) expansion order is changing, in order to make it better align with user expectations, and to make layering of configs work as intended. To prepare for the change, please test your build with startup option `--expand_configs_in_place`.

Please test this change with Bazel 0.10, triggered by the startup option `--expand_configs_in_place`. The change is mostly live with Bazel 0.9, but the newest release adds an additional warning if explicit flags are overriden, which should be helpful when debugging differences. The new expansion order will become the default behavior soon, probably in Bazel 0.11 or 0.12, and will no longer be configurable after another release.

Background: bazelrc & --config

The Bazel User Manual (https://docs.bazel.build/versions/master/user-manual.html#bazelrc) contains the official documentation for bazelrcs and will not be repeated here.
A Bazel build command line generally looks something like this:

```
bazel <startup options> build <command options> //some/targets
```

For the rest of the doc, command options are the focus. Startup options can affect which bazelrc's are loaded, and the new behavior is gated by a startup option, but the config mechanisms are only relevant to command options.

The bazelrcs allow users to set command options by default. These options can either be provided unconditionally or through a config expansion:

- Unconditionally, they are defined for a command, and all commands that inherit from it,
  `build --foo # applies "--foo" to build, test, etc.`
- As a config expansion `# applies "--foo" to build, test, etc. when --config=foobar is set.`
  `build:foobar --foo`

What is changing

The current order: fixed-point --config expansion

The current semantics of --config expansions breaks last-flag-wins expectations. In broad strokes, the current option order is

1. Unconditional rc options (options set by a command without a config, " `build --opt` ")
2. All `--config` expansions are expanded in a "fixed-point" expansions.
   *This does not check where the `--config` option initially was (rc, command line, or another `--config`), and will parse a single `--config` value at most once. Use `--announce*rc` to see the order used!_
3. Command-line specified options

Bazel claims to have a last-flag-wins command line, and this is usually true, but the fixed-point expansion of configs makes it difficult to rely on ordering where `--config` options are concerned.

See the Boolean option example below.

The new order: Last-Flag-Wins

Everywhere else, the last mention of a single-valued option has "priority" and overrides a previous value. The same will now be true of `--config` expansion. Like other expansion options, `--config` will now expand to its rc-defined expansion "in-place," so that the options it expands to have the same precedence.

Since this is no longer a fixed-point expansion, there are a few other changes:

- `--config=foo --config=foo` will be expanded twice. If this is undesirable, more care will need to be taken to avoid redundancy. Double occurrences will cause a warning.
- cycles are no longer implicitly ignored, but will error out.

Other rc ordering semantics remain. "common" options are expanded first, followed by the command hierarchy. This means that for an option added on the line " `build:foo --buildopt` ", it will get added to `--config=foo` 's expansion for bazel build, test, coverage, etc. " `test:foo --testopt` " will add `--testopt` after the (less specific and therefore lower priority) build expansion of `--config=foo`. If this is confusing, avoid alternating command types in the rc file, and group them in order, general options at the top. This way, the order of the file is close to the interpretation order.

How to start using new behavior

1. Check your usual `--config` values' expansions by running your usual bazel command line with `--announce_rc`. The order that the configs are listed, with the options they expand to, is the order in which they are interpreted.
2. Spend some time understanding the applicable configs, and check if any configs expand to the same option. If they do, you may need to move rc lines around to make sure the same value has priority with the new ordering. See "Suggestions for config definers."
3. Flip on the startup option `--expand_configs_in_place` and debug any differences using `--announce_rc`

If you have a shared bazelrc for your project, note that changing it will propagate to other users who might be importing this bazelrc into a personal rc. Proceed with caution as needed

1. Add the startup option to your bazelrc to continue using this new expansion order.

Suggestions for config definers

You might be in a situation where you own some `--config` definitions that are shared between different people, even different teams, so it might be that the users of your config are using both `--expand_configs_in_place` behavior and the old, default behavior.

In order to minimize differences between old and new behavior, here are some tips.

1. Avoid internal conflicts within a config expansion (redefinitions of the same option)
2. Define recursive `--config` at the END of the config expansion
    - Only critical if #1 can't be followed.

Suggestion #1 is especially important if the config expands to another config. The behavior will be more predictable with `--expand_configs_in_place`, but without it, the expansion of a single `--config` depends on previous `--configs`.

Suggestion #2 helps mitigate differences if #1 is violated, since the fixed-point expansion will expand all explicit options, and then expand any newly-found config values that were mentioned in the original config expansions. This is equivalent to expanding it at the end of the list, so use this order if you wish to preserve old behavior.

Motivating examples

The following example violates both #1 and #2, to help motivate why #2 makes things slightly better when #1 is impossible. `bazelrc contents: build:foo --cpu=x86 build:misalteredfoo --config=foo # Violation of #2! build:misalteredfoo --cpu=arm64 # Violation of #1!`

- `bazel build --config=misalteredfoo`

effectively x86 in fixed-point expansion, and arm64 with in-place expansion

The following example still violates #1, but follows suggestion #2: `bazelrc contents: build:foo --cpu=x86 build:misalteredfoo --cpu=arm64 # Violation of #1! build:misalteredfoo --config=foo`

- `bazel build --config=misalteredfoo`

effectively x86 in both expansions, so this does not diverge and appears fine at first glance. (thanks, suggestion #2!)

- `bazel build --config=foo --config=misalteredfoo`

effectively arm64 in fixed-point expansion, x86 with in-place, since misalteredfoo's expansion is independent of the previous config mention.

Suggestions for users of `--config`

Lay users of `--config` might also see some surprising changes depending on usage patterns. The following suggestions are to avoid those differences. Both of the following will cause warnings if missed.

A. Avoid including to the same `--config` twice

B. Put `--config` options FIRST, so that explicit options continue to have precedence over the expansions of the configs.

Multiple mentions of a single `--config`, when combined with violations of #1, may cause surprising results, as shown in #1's motivating examples. In the new expansion, multiple expansions of the same config will warn. Multi-valued options will receive duplicates values, which may be surprising.

Motivating example for B

```
bazelrc contents:
    build:foo --cpu=x86
```

- `bazel build --config=foo --cpu=arm64 # Fine`

effectively arm64 in both expansion cases

- `bazel build --cpu=arm64 --config=foo # Violates B`

The explicit value arm64 has precedence with fixed-point expansion, but the config value x86 wins in in-place expansion. With in-place expansion, this will print a warning.

Additional Boolean Option Example

There are 2 boolean options, `--foo` and `--bar`. Each only accept one value (as opposed to accumulating multiple values).

In the following examples, the two options `--foo` and `--bar` have the same apparent order (and will have the same behavior with the new expansion logic). What changes from one example to the next is where the options are specified.

bazelrc	Command Line	Current final value	New final value

bazelrc	Command Line	Current final value	New final value

	`--nofoo`
	`--foo`
	`--bar`
	`--nobar`
		`--foo`
		`--nobar`
			`--foo`
			`--nobar`

`# Config definitions`
`build:all --foo`
`build:all --bar`
	`--nofoo`
	`--config=all`
	`--nobar`
		`--nofoo`
		`--nobar`
			`--foo`
			`--nobar`

`# Set for every build`
`build --nofoo`
`build --config=all`
`build --nobar`
 	
`# Config definitions`
`build:all --foo`
`build:all --bar`
		`--foo`
		`--bar`
			`--foo`
			`--nobar`

Now to make this more complicated, what if a config includes another config?

bazelrc	Command Line	Current final value	New final value

`# Config definitions`
`build:combo --nofoo`
`build:combo --config=all`
`build:combo --nobar`
`build:all --foo`
`build:all --bar`
	`--config=combo`	`--foo`
		`--bar`
			`--foo`
			`--nobar`

	`--config=all`
	`--config=combo`
		`--nofoo`
		`--nobar`
			`--foo`
			`--nobar`

Here, counterintuitively, including `--config=all` explicitly makes its values effectively disappear. This is why it is basically impossible to create an automatic migration script to run on your rc - there's no real way to know what the *intended* behavior is.

Unfortunately, it gets worse, especially if you have the same config for different commands, such as build and test, or if you defined these in different files. It frankly isn't worth going into the further detail of the ordering semantics as it's existed up until now, this should suffice to demonstrate why it needs to change.

To understand the order of your configs specifically, run Bazel as you normally would (remove targets for speed) with the option `--announce_rc`. The order in which the config expansions are output to the terminal is the order in which they are currently interpreted (again, between rc and command line).

By Chloe Calvarin (https://github.com/cvcal)

Introducing Bazel Code Search (/2017/12/14/introducing-bazel-code-search.html)

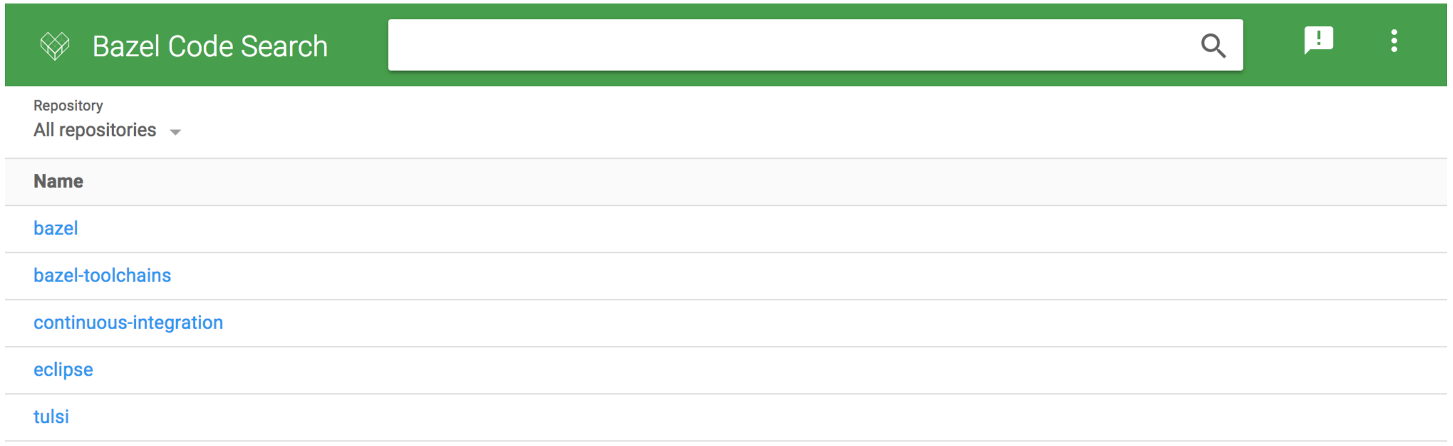
14 December 2017

We are always looking for new ways to improve the experience of contributing to Bazel and helping users understanding how Bazel works. Today, we're excited to share a preview of Bazel Code Search (https://source.bazel.build), a major upgrade to Bazel's code search site. This new site features a refreshed user interface for code browsing and cross-repository semantic search with regular expression support, and a navigable semantic index of all definitions and references for the Bazel codebase. We've also updated the "Contribute" page on the Bazel website with documentation for this tool (https://bazel.build/browse-and-search-user-guide.html).

Getting started with Bazel Code Search

You can try Bazel Code Search right now by visiting https://source.bazel.build (https://source.bazel.build).

Select the repository you want to browse from the list on the main screen, or search across all Bazel repositories on the site using the search box at the top of the page.

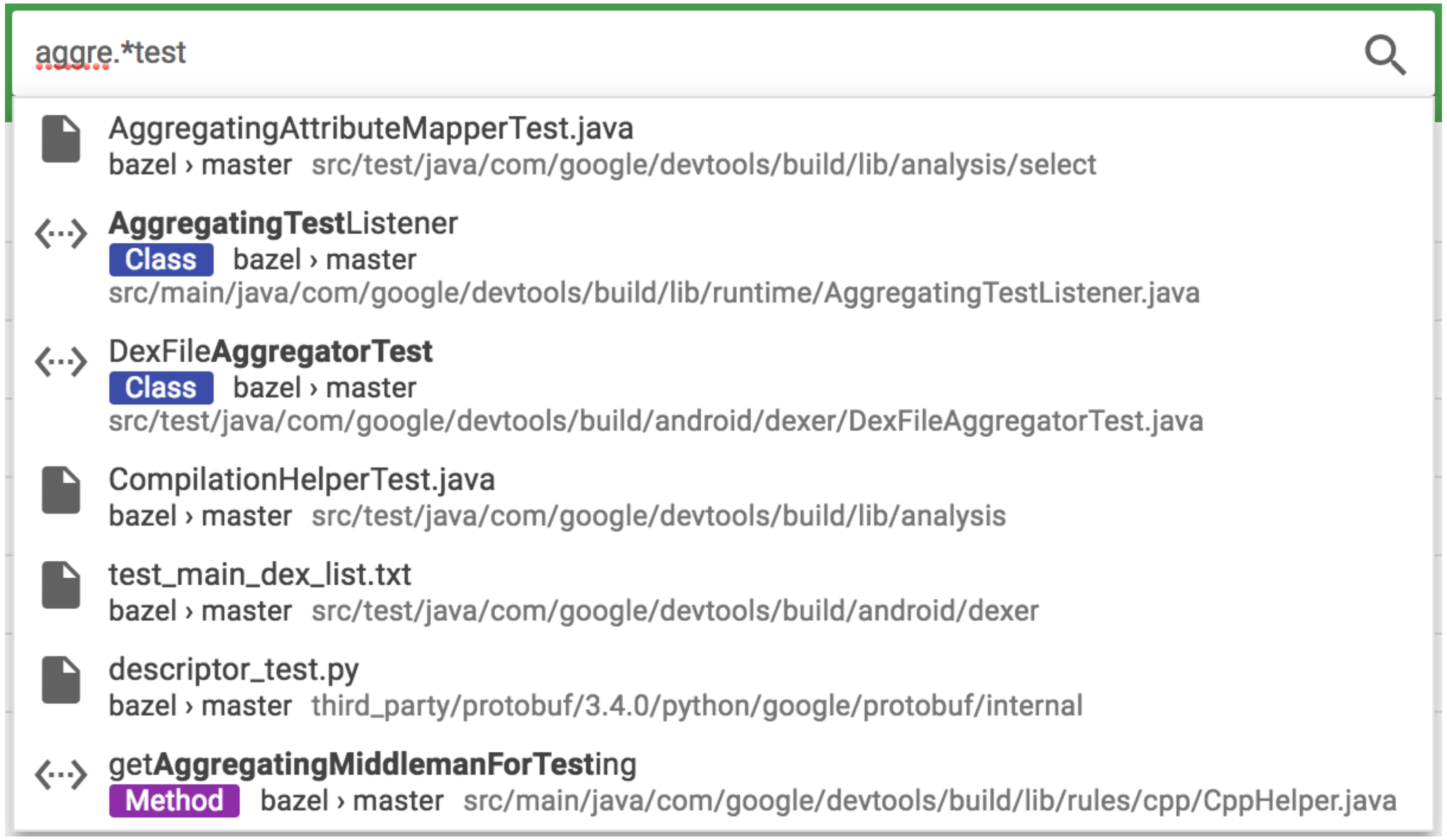


Name
bazel
bazel-toolchains
continuous-integration
eclipse
tulsi

Searching the Bazel codebase

Bazel Code Search has a semantic understanding of the Bazel codebase and allows you to search for either files or code within files (https://bazel.build/browse-and-search-user-guide.html#search). This semantic understanding of the code means that the search index identifies which parts of your code are entities such as classes, functions, and fields. Since the search index has classified these entities, your queries can include filters to scope the search to classes or functions (https://bazel.build/browse-and-search-user-guide.html#search) and allows for improved search relevance by ranking important parts of code like classes, functions, and fields higher. By default, all searches use RE2 regular expressions (https://github.com/google/re2/wiki/Syntax) though you can escape individual special characters with a backslash, or an entire string by enclosing it in quotes.

To search, start typing in the search box at the top of the screen and you'll see suggestions for matching results. For Java, JavaScript, and Proto, result suggestions indicate if the match is an entity such as a Class, Method, Enum or Field. Semantic understanding for more languages is on the way.



aggre.*test

AggregatingAttributeMapperTest.java
bazel › master src/test/java/com/google/devtools/build/lib/analysis/select

AggregatingTestListener
Class bazel › master
src/main/java/com/google/devtools/build/lib/runtime/AggregatingTestListener.java

DexFile**AggregatorTest**
Class bazel › master
src/test/java/com/google/devtools/build/android/dexer/DexFileAggregatorTest.java

CompilationHelperTest.java
bazel › master src/test/java/com/google/devtools/build/lib/analysis

test_main_dex_list.txt
bazel › master src/test/java/com/google/devtools/build/android/dexer

descriptor_test.py
bazel › master third_party/protobuf/3.4.0/python/google/protobuf/internal

get**AggregatingMiddlemanForTest**ing
Method bazel › master src/main/java/com/google/devtools/build/lib/rules/cpp/CppHelper.java

If you don’t see the result you want in the suggestions, you can submit your search and find all matches on the search result page. From the results page, you can select a matching line or file to view.

Here’s a sampling of different search examples to try out on your own:

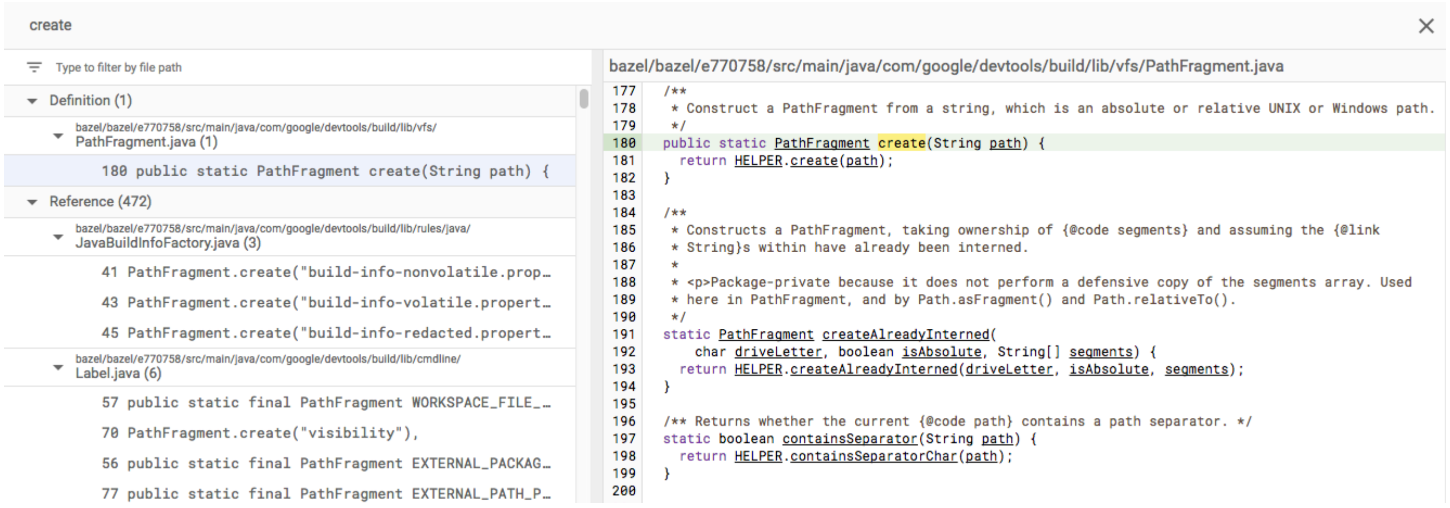
- ccToolchain (https://source.bazel.build/search?q=ccToolchain)
    - search for the substring “ccToolchain”
- class ccToolchain (https://source.bazel.build/search?q=class%20ccToolchain)
    - search for files containing both “class” and “ccToolchain” substrings
- “class ccToolchain” (https://source.bazel.build/search?q=%22class%20ccToolchain%22)
    - search for files containing the phrase “class ccToolchain”
- class:ccToolchain (https://source.bazel.build/search?q=class:ccToolchain)
    - search for classes where the name of a class contains the substring “ccToolchain”
- file:cpp ccToolchain (https://source.bazel.build/search?q=file:cpp%20ccToolchain)

    - search for files containing the substring “ccToolchain” where “cpp” is in the file path
- file:cpp lang:java ccToolchain (https://source.bazel.build/search?q=file:cpp%20lang:java%20ccToolchain)
    - search for Java files containing the substring “ccToolchain” where “cpp” is in the file path
- aggre.*test (https://source.bazel.build/search?q=aggre.*test)
    - search for the regular expression “aggre.*test”
- ccToolchain -test (https://source.bazel.build/search?q=ccToolchain%20-test)
    - search for the substring “ccToolchain” excluding any files containing the substring “test”
- cTool case:yes (https://source.bazel.build/search?q=cTool%20case:yes)
    - search for the substring “cTool” (case sensitive)

Note that all searches are case insensitive unless you specify “case:yes” in the query.

Understanding the Bazel codebase using cross references

Another way to understand the Bazel repository is through the use of cross references (https://bazel.build/browse-and-search-user-guide.html#browsing-cross-references). If you’ve ever wondered how to properly use a method, cross references can help by displaying all references to that method so you can see how it is used in other parts of the codebase. Alternatively, if you see a method being used but don’t understand what that method actually does, cross references enables you to click the method to view the definition or see how it’s used elsewhere.

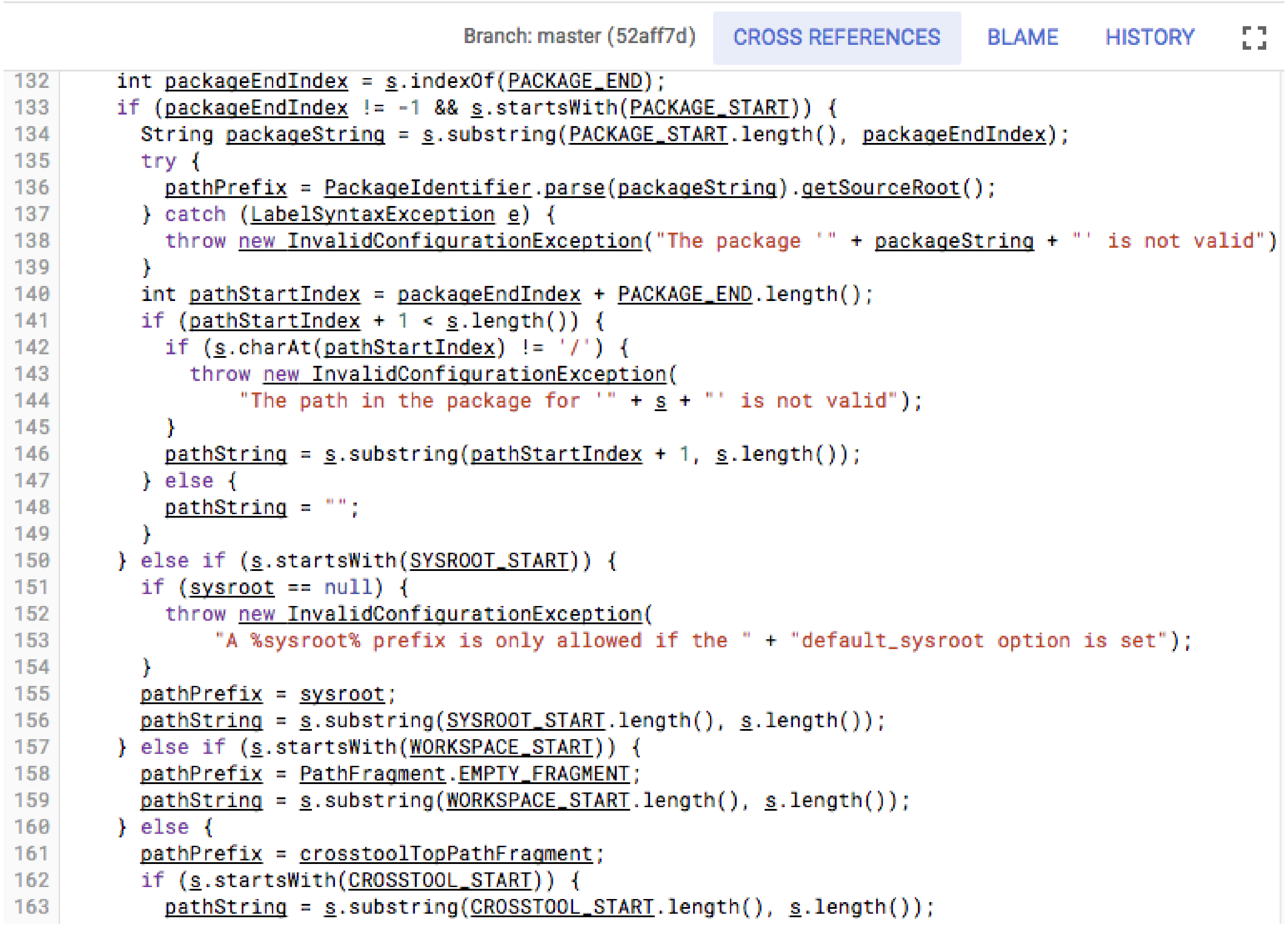


```
Definition (1)
  bazel/bazel/e770758/src/main/java/com/google/devtools/build/lib/vfs/
  PathFragment.java (1)
      180 public static PathFragment create(String path) {
Reference (472)
  bazel/bazel/e770758/src/main/java/com/google/devtools/build/lib/rules/java/
  JavaBuildInfoFactory.java (3)
      41 PathFragment.create("build-info-nonvolatile.prop…
      43 PathFragment.create("build-info-volatile.propert…
      45 PathFragment.create("build-info-redacted.propert…
  bazel/bazel/e770758/src/main/java/com/google/devtools/build/lib/cmdline/
  Label.java (6)
      57 public static final PathFragment WORKSPACE_FILE_…
      70 PathFragment.create("visibility"),
      56 public static final PathFragment EXTERNAL_PACKAG…
      77 public static final PathFragment EXTERNAL_PATH_P…
```

bazel/bazel/e770758/src/main/java/com/google/devtools/build/lib/vfs/PathFragment.java

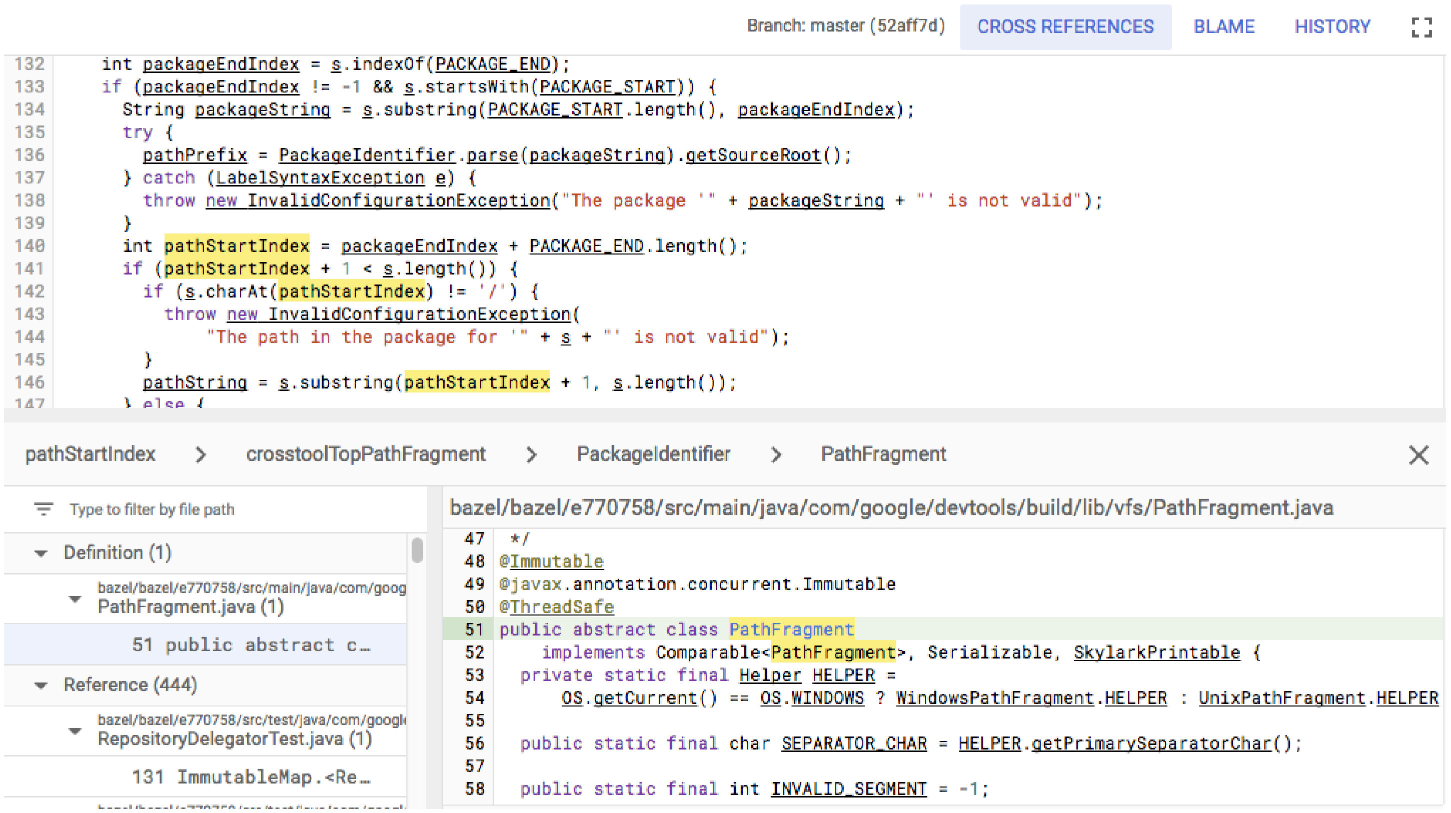
```java
  /**
   * Construct a PathFragment from a string, which is an absolute or relative UNIX or Windows path.
   */
  public static PathFragment create(String path) {
    return HELPER.create(path);
  }

  /**
   * Constructs a PathFragment, taking ownership of {@code segments} and assuming the {@link
   * String}s within have already been interned.
   *
   * <p>Package-private because it does not perform a defensive copy of the segments array. Used
   * here in PathFragment, and by Path.asFragment() and Path.relativeTo().
   */
  static PathFragment createAlreadyInterned(
      char driveLetter, boolean isAbsolute, String[] segments) {
    return HELPER.createAlreadyInterned(driveLetter, isAbsolute, segments);
  }

  /** Returns whether the current {@code path} contains a path separator. */
  static boolean containsSeparator(String path) {
    return HELPER.containsSeparatorChar(path);
  }

```

Cross references aren't only available for methods, they're also generated for classes, fields, imports, and enums. Bazel Code Search uses the Kythe (https://kythe.io/docs/kythe-overview.html) open source project to generate a semantic index of cross references for the Bazel codebase. These cross references appear automatically as hyperlinks within source files. To make cross references easier to identify, click the **Cross References** button to underline all cross references in a file.



```
    int packageEndIndex = s.indexOf(PACKAGE_END);
    if (packageEndIndex != -1 && s.startsWith(PACKAGE_START)) {
      String packageString = s.substring(PACKAGE_START.length(), packageEndIndex);
      try {
        pathPrefix = PackageIdentifier.parse(packageString).getSourceRoot();
      } catch (LabelSyntaxException e) {
        throw new InvalidConfigurationException("The package '" + packageString + "' is not valid")
      }
      int pathStartIndex = packageEndIndex + PACKAGE_END.length();
      if (pathStartIndex + 1 < s.length()) {
        if (s.charAt(pathStartIndex) != '/') {
          throw new InvalidConfigurationException(
              "The path in the package for '" + s + "' is not valid");
        }
        pathString = s.substring(pathStartIndex + 1, s.length());
      } else {
        pathString = "";
      }
    } else if (s.startsWith(SYSROOT_START)) {
      if (sysroot == null) {
        throw new InvalidConfigurationException(
            "A %sysroot% prefix is only allowed if the " + "default_sysroot option is set");
      }
      pathPrefix = sysroot;
      pathString = s.substring(SYSROOT_START.length(), s.length());
    } else if (s.startsWith(WORKSPACE_START)) {
      pathPrefix = PathFragment.EMPTY_FRAGMENT;
      pathString = s.substring(WORKSPACE_START.length(), s.length());
    } else {
      pathPrefix = crosstoolTopPathFragment;
      if (s.startsWith(CROSSTOOL_START)) {
        pathString = s.substring(CROSSTOOL_START.length(), s.length());
```

Once you've clicked on a cross reference, the cross references pane will be displayed where you can view all the definitions and references organized by file. Within the cross references pane, you can navigate into multiple levels of depth of cross references while continuing to view the original file you were viewing in the File pane allowing you to maintain context of the original task.


Branch: master (52aff7d)
CROSS REFERENCES
BLAME
HISTORY
pathStartIndex > crosstoolTopPathFragment > PackageIdentifier > PathFragment
Type to filter by file path
Definition (1)
Reference (444)

```
    int packageEndIndex = s.indexOf(PACKAGE_END);
    if (packageEndIndex != -1 && s.startsWith(PACKAGE_START)) {
      String packageString = s.substring(PACKAGE_START.length(), packageEndIndex);
      try {
        pathPrefix = PackageIdentifier.parse(packageString).getSourceRoot();
      } catch (LabelSyntaxException e) {
        throw new InvalidConfigurationException("The package '" + packageString + "' is not valid");
      }
      int pathStartIndex = packageEndIndex + PACKAGE_END.length();
      if (pathStartIndex + 1 < s.length()) {
        if (s.charAt(pathStartIndex) != '/') {
          throw new InvalidConfigurationException(
              "The path in the package for '" + s + "' is not valid");
        }
        pathString = s.substring(pathStartIndex + 1, s.length());
      } else {
```

bazel/bazel/e770758/src/main/java/com/goog
PathFragment.java (1)
51 public abstract c…

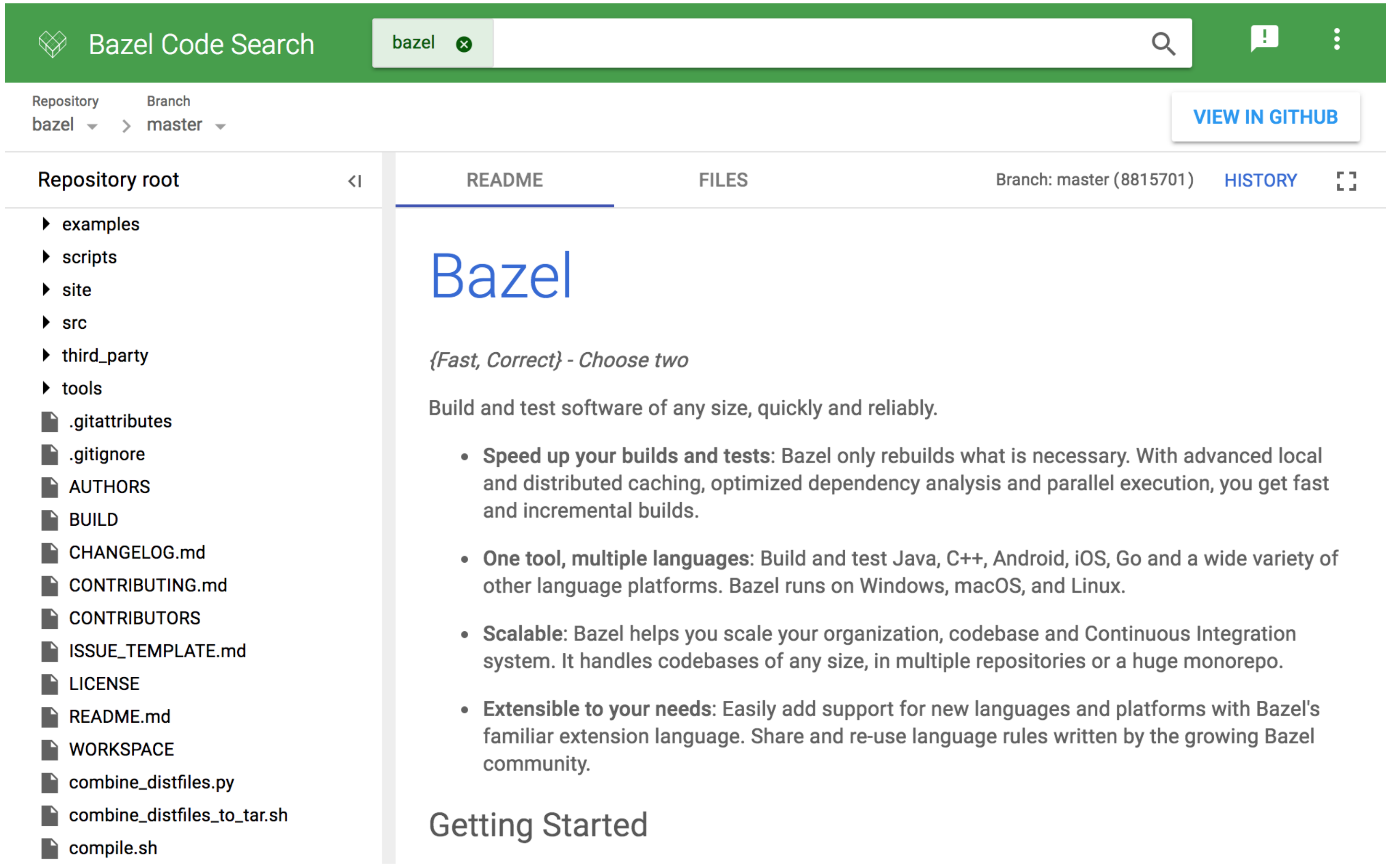
bazel/bazel/e770758/src/test/java/com/googl
RepositoryDelegatorTest.java (1)
131 ImmutableMap.<Re…

bazel/bazel/e770758/src/main/java/com/google/devtools/build/lib/vfs/PathFragment.java

```
 */
@Immutable
@javax.annotation.concurrent.Immutable
@ThreadSafe
public abstract class PathFragment
    implements Comparable<PathFragment>, Serializable, SkylarkPrintable {
  private static final Helper HELPER =
      OS.getCurrent() == OS.WINDOWS ? WindowsPathFragment.HELPER : UnixPathFragment.HELPER

  public static final char SEPARATOR_CHAR = HELPER.getPrimarySeparatorChar();

  public static final int INVALID_SEGMENT = -1;
```

Browsing through Bazel repositories

Selecting a repository from the main screen will take you to a view of the chosen repository with search scoped to its contents. The breadcrumb toolbar at the top allows you to quickly navigate to other repositories (https://bazel.build/browse-and-search-user-guide.html#switch-repositories), refs (https://bazel.build/browse-and-search-user-guide.html#open-a-branch-commit-or-tag), or folders.



examples
scripts
site
src
third_party
tools
.gitattributes
.gitignore
AUTHORS
BUILD
CHANGELOG.md
CONTRIBUTING.md
CONTRIBUTORS
ISSUE_TEMPLATE.md
LICENSE
README.md
WORKSPACE
combine_distfiles.py
combine_distfiles_to_tar.sh
compile.sh

Bazel

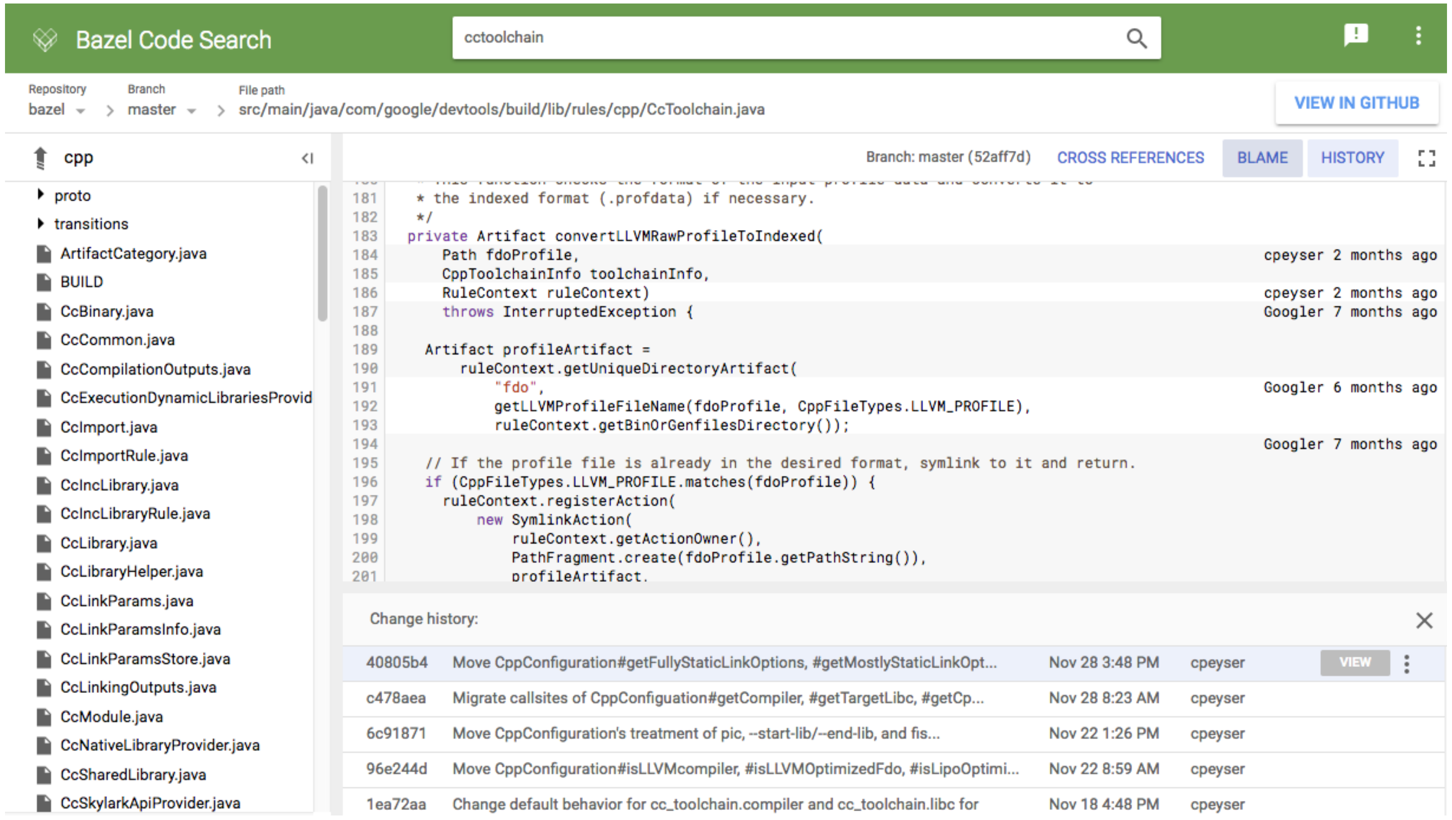
{Fast, Correct} - Choose two

Build and test software of any size, quickly and reliably.

- **Speed up your builds and tests**: Bazel only rebuilds what is necessary. With advanced local and distributed caching, optimized dependency analysis and parallel execution, you get fast and incremental builds.
- **One tool, multiple languages**: Build and test Java, C++, Android, iOS, Go and a wide variety of other language platforms. Bazel runs on Windows, macOS, and Linux.
- **Scalable**: Bazel helps you scale your organization, codebase and Continuous Integration system. It handles codebases of any size, in multiple repositories or a huge monorepo.
- **Extensible to your needs**: Easily add support for new languages and platforms with Bazel's familiar extension language. Share and re-use language rules written by the growing Bazel community.

Getting Started

From the view of the repository, you can browse through folders and files in the repository while taking advantage of blame (https://bazel.build/browse-and-search-user-guide.html#view-file-changes), change history (https://bazel.build/browse-and-search-user-guide.html#view-change-history), a diff view (https://bazel.build/browse-and-search-user-guide.html#compare-a-file-to-a-different-commit) and many other features.



```
   * the indexed format (.profdata) if necessary.
   */
  private Artifact convertLLVMRawProfileToIndexed(
      Path fdoProfile,                                                        cpeyser 2 months ago
      CppToolchainInfo toolchainInfo,
      RuleContext ruleContext)                                                cpeyser 2 months ago
      throws InterruptedException {                                           Googler 7 months ago

    Artifact profileArtifact =
        ruleContext.getUniqueDirectoryArtifact(
            "fdo",                                                            Googler 6 months ago
            getLLVMProfileFileName(fdoProfile, CppFileTypes.LLVM_PROFILE),
            ruleContext.getBinOrGenfilesDirectory());
                                                                              Googler 7 months ago
    // If the profile file is already in the desired format, symlink to it and return.
    if (CppFileTypes.LLVM_PROFILE.matches(fdoProfile)) {
      ruleContext.registerAction(
          new SymlinkAction(
              ruleContext.getActionOwner(),
              PathFragment.create(fdoProfile.getPathString()),
              profileArtifact,
```

40805b4	Move CppConfiguration#getFullyStaticLinkOptions, #getMostlyStaticLinkOpt...	Nov 28 3:48 PM	cpeyser
c478aea	Migrate callsites of CppConfiguation#getCompiler, #getTargetLibc, #getCp...	Nov 28 8:23 AM	cpeyser
6c91871	Move CppConfiguration's treatment of pic, --start-lib/--end-lib, and fis...	Nov 22 1:26 PM	cpeyser
96e244d	Move CppConfiguration#isLLVMcompiler, #isLLVMOptimizedFdo, #isLipoOptimi...	Nov 22 8:59 AM	cpeyser
1ea72aa	Change default behavior for cc_toolchain.compiler and cc_toolchain.libc for	Nov 18 4:48 PM	cpeyser

Give Feedback

We hope you'll try Bazel Code Search (https://source.bazel.build) and provide feedback through the "**!**" button in the top right of any page on the Bazel Code Search site. We would love to hear whether this tool helps you work with Bazel and what else you'd like to see Bazel Code Search offer.

Keep in mind that this project is still experimental and is subject to change.

By Russell Wolf (https://github.com/russwolf)

Thank you for BazelCon 2017 (/2017/12/11/thanks-bazelcon2017.html)

11 December 2017

We are truly thankful to our community for making our first annual Bazel Conference (https://sites.google.com/corp/bazel.build/conference2017) a success! Check out all the videos of all the talks from **BazelCon 2017 on YouTube** (https://www.youtube.com/playlist?list=PLxNYxgaZ8RseY0KmkXQSt0StE71E7yizG).

Your feedback reflected high level of satisfaction, and there was something of interest for everyone:

- Bazel usage and migration stories from TensorFlow, Kubernetes, SpaceX, Pinterest, Wix, Stripe, DataBricks, and Dropbox.
- Talks about upcoming features and tools such as remote execution & caching (https://github.com/bazelbuild/bazel-buildfarm), buildozer (https://github.com/bazelbuild/buildtools/tree/master/buildozer), robolectric tests, PodToBUILD (https://github.com/pinterest/PodToBUILD), bazeltfc (https://github.com/Asana/bazeltsc), rules_typescript (https://github.com/bazelbuild/rules_typescript) presented by Uber, Two Sigma, Google, Pinterest, and Asana.
- Office hours where you got your Bazel questions answered, met engineers, and debugged on the fly. One attendee used their session to configure remote execution with buildfarm (https://github.com/bazelbuild/bazel-buildfarm)!

BazelCon2017 by the Numbers:

- 200+ attendees
- 60 organizations
- 30 speakers
- 12 informative talks
- 14 hours of hands-on debugging and Q&A during Office Hours

What we heard from you:

- Bazel Query is great!
- Python is used more widely than we realized, and needs better Bazel support.

- Reproducible builds are important especially when you're flying rockets, building autonomous vehicles, delivering media, and calculating financial transactions, leading to wider Bazel adoption by these communities.
- Documentation is often hard to find, and is either too basic or too advanced.
- Bazel's parallelism is impressive: much faster than Maven.
- IDE integration is important, particularly with Visual Studio, CLion, XCode.
- Build Event Protocol enables many options for internal visualization of events.
- Aspects are a powerful tool.

What we can do next:

- Work on better documentation and training, particularly for intermediate-level topics.
- Prioritize IDE integration.
- Engage the community in building better Python support in Bazel.
- Implement improved support for third-party dependencies.
- Continue work on cross-platform improvements.
- Engage the community in wider adoption and contribution to Bazel.

What you can do next:

- Contribute to Bazel, particularly to the new community driven buildfarm (https://github.com/bazelbuild/bazel-buildfarm) effort.
- Kick off local meet-ups (we will reach out to volunteers who responded to this in our survey).
- Get more info on Bazel at bazel.build (https://bazel.build/).
- Join the discussion on bazel-discuss@googlegroups.com.

We look forward to working with you and growing our Bazel user community in 2018!

By Helen Altshuler (https://github.com/helenalt) and David Stanke (https://github.com/davidstanke)

Bazel on Windows -- a year in retrospect (/2017/10/16/windows-retrospect.html)

16 October 2017

Bazel on Windows is no longer experimental. We think Bazel on Windows is now stable and usable enough to qualify for the Beta status we have given Bazel on other platforms.

Over the last year, we put a lot of work into improving Bazel's Windows support:

- **Bazel no longer depends on the MSYS runtime.** This means Bazel is now a native Windows binary and we no longer link it to the MSYS DLLs. Bazel still needs Bash (MSYS or Git Bash) and the GNU binutils (binaries under `/usr/bin`) if your dependency graph includes `genrule` or `sh_*` rules (similarly to requiring `python.exe` to build `py_*` rules), but you can use any MSYS version and flavour you like, including Git Bash.
- **Bazel can now build Android applications.** If you use native `android_*` rules, Bazel on Windows can now build and deploy Android applications.
- **Bazel is easier to set up.** You now (typically) no longer need to set the following environment variables:
  - `BAZEL_SH` and `BAZEL_PYTHON` -- Bazel attempts to autodetect your Bash and Python installation paths.
  - `JAVA_HOME` -- we release Bazel with an embedded JDK. (We also release binaries without an embedded JDK if you want to use a different one.)
- **Visual C++ is the default C++ compiler.** We no longer use GCC by default, though Bazel still supports it.
- **Bazel integrates better with the Visual C++ compiler.** Bazel no longer dispatches to a helper script to run the compiler; instead Bazel now has a `CROSSTOOL` definition for Visual C++ and drives the compiler directly. This means Bazel creates fewer processes to run the compiler. By removing the script, we have eliminated one more point of failure.
- **Bazel creates native launchers.** Bazel builds native Windows executables from `java_binary`, `sh_binary`, and `py_binary` rules. Unlike the `.cmd` files that Bazel used to build for these rules, the new `.exe` files no longer dispatch to a shell script to launch the `xx_binaries`, resulting in faster launch times. (If you see errors, you can use the `--[no]windows_exe_launcher` flag to fall back to the old behavior; if you do, please file a bug (https://github.com/bazelbuild/bazel/issues/new). We'd like to remove this flag and only support the new behavior.)

Coming soon

We are also working on bringing the following to Bazel on Windows:

- **Android Studio integration.** We'll ensure Bazel works with Android Studio on Windows the same way it does on Linux and macOS. See issue #3888 (https://github.com/bazelbuild/bazel/issues/3888).
- **Dynamic C++ Linking.** Bazel will support building and linking to DLLs on Windows. See issue #3311 (https://github.com/bazelbuild/bazel/issues/3311).
- **Skylark rule migration guide.** We'll publish tutorials on writing Skylark rules that work not just on Linux and macOS, but also on Windows. See issue #3889 (https://github.com/bazelbuild/bazel/issues/3889).

Looking ahead, we aim to maintain feature parity between Windows and other platforms. We aim to maximize portability between host systems, so you get the same fast, correct builds on your developer OS of choice. If you run into any problems, please don't hesitate to file a bug (https://github.com/bazelbuild/bazel/issues/new).

By László Csomor (https://github.com/laszlocsomor)

Bazel Conference 2017 (/2017/10/03/conference-2017.html)

03 October 2017

Bazel team is pleased to announce our first annual **Bazel Conference** (https://sites.google.com/corp/bazel.build/conference2017), focused on the needs of our community. The conference will feature user stories and feedback, migration talks, roadmap, hands-on and break-out tech sessions with Bazel engineers, contributors and users.

Dates: November 6 and 7, 2017 **Location:** Sunnyvale, California

We are inspired by our community engagement in the upcoming conference. We've received a large number of requests for talks and topics for discussion, and have scheduled tech talks by engineers at large and small companies on working with Bazel in wide variety of languages, including Scala, JavaScript and Go. We are looking forward to having 200 engineers from 35 companies across the world share their experience with Bazel.

We are humbled by the commitment to make Bazel even better, and are seeing engineers develop advanced features like Remote Execution, and sharing migration tips, tricks and tools. You will hear user stories and tips about iOS migration, and TensorFlow and Kubernetes experience with Bazel. We will also discuss as a community different tools out there that could be open sourced.

We are excited to see all of you at this first annual Bazel Conference (https://sites.google.com/corp/bazel.build/conference2017) in Sunnyvale, California on November 6 and 7, 2017!

Register by October 15, as seating is limited and we won't allow walk-ins. Detailed schedule will be published mid October, and location details will be sent out to registered attendees.

By Helen Altshuler (https://github.com/helenalt)

The New World of Bazel Releases (/2017/09/27/release-cadence.html)

27 September 2017

Bazel has been open-sourced exactly 2.5 years ago (https://blog.bazel.build/2015/03/27/Hello-World.html). It continues to be quite a journey, and we are very happy we have acquired some fellow travellers: many projects, organizations and companies that we all know and love (https://sites.google.com/corp/bazel.build/conference2017/agenda) rely on Bazel every day.

As our community grows, we owe to it a transparent and predictable release process. So we are taking some steps to bring more clarity and order to the world of Bazel releases:

- We will cut (from master (https://github.com/bazelbuild/bazel/tree/master)) a candidate for *minor releases* (0.7, 0.8 and so on) **every month** on approximately first business day of the month. The planned target dates for the cuts are published as GitHub issues with label 'release' (https://github.com/bazelbuild/bazel/issues?q=label%3Arelease%20)
- After the release candidate is cut, we will let it bake for *two weeks* before promoting it to the release. We will cherry-pick fixes for critical bugs, issuing new release candidates (0.7.0rc1, 0.7.0rc2 and so on), and release

0.*minor*.0 version at the end of baking period.

- If critical bugs are discovered after the release, we will fix them and issue *patch releases* as needed (0.7.1, 0.7.2 and so on). Patch releases are always patches on top of existing minor releases - they are never cut from master.
- No new features or backward incompatible changes ever appear in patch releases, and no new features or backward incompatible changes are added to release candidates after they have been cut.

Our website has more details on release policy (https://bazel.build/support.html#releases).

As a result of this change, we now issue one minor release per month.

Our roadmap (https://bazel.build/roadmap.html) reflects our vision for Bazel 1.0 and beyond. We will annotate features on the roadmap with the release versions as those features get shipped.

By Dmitry Lomov (https://github.com/dslomov)

← Newer

Older → (/page2)

About

Who's using Bazel (https://github.com/bazelbuild/bazel/wiki/Bazel-Users)
Roadmap (https://bazel.build/roadmap.html)
Contribute (https://bazel.build/contributing.html)
Governance Plan (https://bazel.build/governance.html)

Support

Stack Overflow (https://stackoverflow.com/questions/tagged/bazel)
Issue Tracker (https://github.com/bazelbuild/bazel/issues)
Documentation (https://docs.bazel.build)
FAQ (https://bazel.build/faq.html)
Support Policy (https://bazel.build/support.html)

Stay Connected

Twitter (https://twitter.com/bazelbuild)
Blog (https://blog.bazel.build)
GitHub (https://github.com/bazelbuild/bazel)
Discussion group (https://groups.google.com/forum/#!forum/bazel-discuss)