Author: lberki@google.com
Last significant change: 2020 June 24

THIS DOCUMENT IS WORLD-READABLE.

THIS COPY IS DEPRECATED

read https://bazel.build/contribute/codebase instead

Introduction

The code base of Bazel is large (~350KLOC production code and ~260 KLOC test code) and no one is familiar with the whole landscape: everyone knows their particular valley very well, but few know what lies over the hills in every direction.

In order for people midway upon the journey not to find themselves within a forest dark with the straightforward pathway being lost, this document tries to give an overview of the code base so that it's easier to get started with working on it.

The public version of the source code of Bazel lives on GitHub at http://github.com/bazelbuild/bazel . This is not the “source of truth”; it’s derived from a Google-internal source tree that contains additional functionality that is not useful outside Google. The long term goal is to make GitHub the source of truth.

Contributions are accepted through the regular GitHub pull request mechanism, and manually imported by a Googler into the internal source tree, then re-exported back out to GitHub.

Client/server architecture

The bulk of Bazel resides in a server process that stays in RAM between builds. This allows Bazel to maintain state between builds.

This is why the Bazel command line has two kinds of options: startup and command. In a command line like this:

```
bazel --host_jvm_args=-Xmx8G build -c opt //foo:bar
```

Some options (`--host_jvm_args=`) are before the name of the command to be run and some are after (`-c opt`); the former kind is called a "startup option" and affects the server process as a whole, whereas the latter kind, the "command option", only affects a single command.

Each server instance has a single associated source tree ("workspace") and each workspace usually has a single active server instance. This can be circumvented by specifying a custom output base (see the "Directory layout" section for more information).

Bazel is distributed as a single ELF executable that is also a valid .zip file. When you type `bazel`, the above ELF executable implemented in C++ (the "client") gets control. It sets up an appropriate server process using the following steps:

1. Checks whether it has already extracted itself. If not, it does that. This is where the implementation of the server comes from.
2. Checks whether there is an active server instance that works: it is running, it has the right startup options and uses the right workspace directory. It finds the running server by looking at the directory `$OUTPUT_BASE/server` where there is a lock file with the port the server is listening on.
3. If needed, kills the old server process
4. If needed, starts up a new server process

After a suitable server process is ready, the command that needs to be run is communicated to it over a gRPC interface, then the output of Bazel is piped back to the terminal. Only one command can be running at the same time. This is implemented using an elaborate locking mechanism with parts in C++ and parts in Java. There is some infrastructure for running multiple commands in parallel, since the inability to run e.g. `bazel version` in parallel with another command is somewhat embarrassing. The main blocker is the life cycle of `BlazeModule`s and some state in `BlazeRuntime`.

At the end of a command, the Bazel server transmits the exit code the client should return. An interesting wrinkle is the implementation of `bazel run`: the job of this command is to run something Bazel just built, but it can't do that from the server process because it doesn't have a terminal. So instead it tells the client what binary it should exec() and with what arguments.

When one presses Ctrl-C, the client translates it to a Cancel call on the gRPC connection, which tries to terminate the command as soon as possible. After the third Ctrl-C, the client sends a SIGKILL to the server instead.

The source code of the client is under `src/main/cpp` and the protocol used to communicate with the server is in `src/main/protobuf/command_server.proto` .

The main entry point of the server is `BlazeRuntime.main()` and the gRPC calls from the client are handled by `GrpcServerImpl.run()`.

Directory layout

Bazel creates a somewhat complicated set of directories during a build. A full description is available here.

The "workspace" is the source tree Bazel is run in. It usually corresponds to something you checked out from source control.

Bazel puts all of its data under the "output user root". This is usually `$HOME/.cache/bazel/_bazel_${USER}`, but can be overridden using the `--output_user_root` startup option.

The "install base" is where Bazel is extracted to. This is done automatically and each Bazel version gets a subdirectory based on its checksum under the install base. It's at `$OUTPUT_USER_ROOT/install` by default and can be changed using the `--install_base` command line option.

The "output base" is the place where the Bazel instance attached to a specific workspace writes to. Each output base has at most one Bazel server instance running at any time. It's usually at `$OUTPUT_USER_ROOT/<checksum of the path to the workspace>`. It can be changed using the `--output_base` startup option, which is, among other things, useful for getting around the limitation that only one Bazel instance can be running in any workspace at any given time.

The output directory contains, among other things:

- The fetched external repositories at `$OUTPUT_BASE/external` .
- The exec root, i.e. a directory that contains symlinks to all the source code for the current build. It's located at `$OUTPUT_BASE/execroot`. During the build, the working

directory is `$EXECROOT/<name of main repository>`. We are planning to change this to `$EXECROOT`, although it's a long term plan because it's a very incompatible change.
- Files built during the build.

The process of executing a command

Once the Bazel server gets control and is informed about a command it needs to execute, the following sequence of events happens:

1. `BlazeCommandDispatcher` is informed about the new request. It decides whether the command needs a workspace to run in (almost every command except for ones that don't have anything to do with source code, e.g. version or help) and whether another command is running.

2. The right command is found. Each command must implement the interface `BlazeCommand` and must have the `@Command` annotation (this is a bit of an antipattern, it would be nice if all the metadata a command needs was described by methods on `BlazeCommand`)

3. The command line options are parsed. Each command has different command line options, which are described in the `@Command` annotation.

4. An event bus is created. The event bus is a stream for events that happen during the build. Some of these are exported to outside of Bazel under the aegis of the Build Event Protocol in order to tell the world how the build goes.

5. The command gets control. The most interesting commands are those that run a build: build, test, run, coverage and so on: this functionality is implemented by `BuildTool`.

6. The set of target patterns on the command line is parsed and wildcards like `//pkg:all` and `//pkg/...` are resolved. This is implemented in `AnalysisPhaseRunner.evaluateTargetPatterns()` and reified in Skyframe as `TargetPatternPhaseValue`.

7. The loading/analysis phase is run to produce the action graph (a directed acyclic graph of commands that need to be executed for the build).

8. The execution phase is run. This means running every action required to build the top-level targets that are requested are run.

Command line options

The command line options for a Bazel invocation are described in an `OptionsParsingResult` object, which in turn contains a map from "option classes" to the values of the options. An "option class" is a subclass of `OptionsBase` and groups command line options together that are related to each other. For example:

1. Options related to a programming language (`CppOptions` or `JavaOptions`). These should be a subclass of `FragmentOptions` and are eventually wrapped into a `BuildOptions` object.
2. Options related to the way Bazel executes actions (`ExecutionOptions`)

These options are designed to be consumed in the analysis phase and (either through `RuleContext.getFragment()` in Java or `ctx.fragments` in Starlark). Some of them (for example, whether to do C++ include scanning or not) are read in the execution phase, but that always requires explicit plumbing since `BuildConfiguration` is not available then. For more information, see the section “Configurations”.

WARNING: We like to pretend that `OptionsBase` instances are immutable and use them that way (e.g. as part of `SkyKeys`). This is not the case and modifying them is a really good way to break Bazel in subtle ways that are hard to debug. Unfortunately, making them actually immutable is a large endeavor. (Modifying a `FragmentOptions` immediately after construction before anyone else gets a chance to keep a reference to it and before `equals()` or `hashCode()` is called on it is okay.)

Bazel learns about option classes in the following ways:

1. Some are hard-wired into Bazel (`CommonCommandOptions`)
2. From the @Command annotation on each Bazel command
3. From `ConfiguredRuleClassProvider` (these are command line options related to individual programming languages)
4. Starlark rules can also define their own options (see here)

Each option (excluding Starlark-defined options) is a member variable of a `FragmentOptions` subclass that has the `@Option` annotation, which specifies the name and the type of the command line option along with some help text.

The Java type of the value of a command line option is usually something simple (a string, an integer, a Boolean, a label, etc.). However, we also support options of more complicated types; in this case, the job of converting from the command line string to the data type falls to an implementation of `com.google.devtools.common.options.Converter`.

The source tree, as seen by Bazel

Bazel is in the business of building software, which happens by reading and interpreting the source code. The totality of the source code Bazel operates on is called "the workspace" and it is structured into repositories, packages and rules. A description of these concepts for the users of Bazel is available here.

Repositories

A "repository" is a source tree on which a developer works; it usually represents a single project. Bazel's ancestor, Blaze, operated on a monorepo, i.e. a single source tree that contains all source code used to run the build. Bazel, in contrast, supports projects whose source code spans multiple repositories. The repository from which Bazel is invoked is called the “main repository”, the others are called “external repositories”.

A repository is marked by a file called `WORKSPACE` (or `WORKSPACE.bazel`) in its root directory. This file contains information that is "global" to the whole build, for example, the set of available external repositories. It works like a regular Starlark file which means that one can `load()` other Starlark files. This is commonly used to pull in repositories that are needed by a repository that's explicitly referenced (we call this the "`deps.bzl` pattern")

Code of external repositories is symlinked or downloaded under `$OUTPUT_BASE/external`.

When running the build, the whole source tree needs to be pieced together; this is done by SymlinkForest, which symlinks every package in the main repository to `$EXECROOT` and every external repository to either `$EXECROOT/external` or `$EXECROOT/..` (the former of course makes it impossible to have a package called `external` in the main repository; that's why we are migrating away from it)

Packages

Every repository is composed of packages, i.e. a collection of related files and a specification of the dependencies. These are specified by a file called `BUILD` or `BUILD.bazel`. If both exist, Bazel prefers `BUILD.bazel`; the reason why BUILD files are still accepted is that Bazel’s ancestor, Blaze, used this file name. However, it turned out to be a commonly used path segment, especially on Windows, where file names are case-insensitive.

Packages are independent of each other: changes to the BUILD file of a package cannot cause other packages to change. The addition or removal of BUILD files *can* change other packages, since recursive globs stop at package boundaries and thus the presence of a BUILD file stops the recursion.

The evaluation of a BUILD file is called "package loading". It's implemented in the class `PackageFactory`, works by calling the Starlark interpreter and requires knowledge of the set of available rule classes. The result of package loading is a `Package` object. It's mostly a map from a string (the name of a target) to the target itself.

A large chunk of complexity during package loading is globbing: Bazel does not require every source file to be explicitly listed and instead can run globs (e.g. `glob(["**/*.java"])`). Unlike the shell, it supports recursive globs that descend into subdirectories (but not into subpackages). This requires access to the file system and since that can be slow, we implement all sorts of tricks to make it run in parallel and as efficiently as possible.

Globbing is implemented in the following classes:

- `LegacyGlobber`, a fast and blissfully Skyframe-unaware globber
- `SkyframeHybridGlobber`, a version that uses Skyframe and reverts back to the legacy globber in order to avoid “Skyframe restarts” (described below)

The `Package` class itself contains some members that are exclusively used to parse the WORKSPACE file and which do not make sense for real packages. This is a design flaw because objects describing regular packages should not contain fields that describe something else. These include:

- The repository mappings
- The registered toolchains
- The registered execution platforms

Ideally, there would be more separation between parsing the WORKSPACE file from parsing regular packages so that `Package` does not need to cater for the needs of both. This is unfortunately difficult to do because the two are intertwined quite deeply.

Labels, Targets and Rules

Packages are composed of targets, which have the following types:

1. **Files:** things that are either the input or the output of the build. In Bazel parlance, we call them *artifacts* (discussed elsewhere). Not all files created during the build are targets; it’s common for an output of Bazel not to have an associated label.
2. **Rules:** these describe steps to derive its outputs from its inputs. They are generally associated with a programming language (e.g. `cc_library`, `java_library` or `py_library`), but there are some language-agnostic ones (e.g. `genrule` or `filegroup`)
3. **Package groups:** discussed in the Visibility section.

The name of a target is called a *Label*. The syntax of labels is `@repo//pac/kage:name`, where `repo` is the name of the repository the Label is in, `pac/kage` is the directory its BUILD file is in and `name`

is the path of the file (if the label refers to a source file) relative to the directory of the package. When referring to a target on the command line, some parts of the label can be omitted:

1. If the repository is omitted, the label is taken to be in the main repository.
2. If the package part is omitted (e.g. `name` or `:name`), the label is taken to be in the package of the current working directory (relative paths containing uplevel references (..) are not allowed)

A kind of a rule (e.g. "C++ library") is called a "rule class". Rule classes may be implemented either in Starlark (the `rule()` function) or in Java (so called “native rules”, type `RuleClass`). In the long term, every language-specific rule will be implemented in Starlark, but some legacy rule families (e.g. Java or C++) are still in Java for the time being.

Starlark rule classes need to be imported at the beginning of BUILD files using the `load()` statement, whereas Java rule classes are "innately" known by Bazel, by virtue of being registered with the `ConfiguredRuleClassProvider`.

Rule classes contain information such as:

1. Its attributes (e.g., `srcs`, `deps`): their types, default values, constraints, etc.
2. The configuration transitions and aspects attached to each attribute, if any
3. The implementation of the rule
4. The transitive info providers the rule "usually" creates

Terminology note: In the code base, we often use “Rule” to mean the target created by a rule class. But in Starlark and in user-facing documentation, “Rule” should be used exclusively to refer to the rule class itself; the target is just a “target”. Also note that despite `RuleClass` having “class” in its name, there is no Java inheritance relationship between a rule class and targets of that type.

Skyframe

The evaluation framework underlying Bazel is called Skyframe. Its model is that everything that needs to be built during a build is organized into a directed acyclic graph with edges pointing from any pieces of data to its dependencies, that is, other pieces of data that need to be known to construct it.

The nodes in the graph are called `SkyValue`s and their names are called `SkyKey`s. Both are deeply immutable, i.e. only immutable objects should be reachable from them. This invariant almost always holds, and in case it doesn't (e.g. for the individual options classes `BuildOptions`, which is a member of `BuildConfigurationValue` and its `SkyKey`) we try really hard not to change them or to change them in only ways that are not observable from the outside. From this it follows that everything that is computed within Skyframe (e.g. configured targets) must also be immutable.

The most convenient way to observe the Skyframe graph is to run `bazel dump --skyframe=detailed`, which dumps the graph, one `SkyValue` per line. It's best to do it for tiny builds, since it can get pretty large.

Skyframe lives in the `com.google.devtools.build.skyframe` package. The similarly-named package `com.google.devtools.build.lib.skyframe` contains the implementation of Bazel on top of Skyframe. More information about Skyframe is available here.

Generating a new `SkyValue` involves the following steps:

1. Running the associated `SkyFunction`
2. Declaring the dependencies (i.e. `SkyValue`s) that the `SkyFunction` needs to do its job. This is done by calling the various overloads of `SkyFunction.Environment.getValue()`.
3. If a dependency is not available, Skyframe signals that by returning null from `getValue()`. In this case, the `SkyFunction` is expected to yield control to Skyframe by returning null, then Skyframe evaluates the dependencies that haven't been evaluated yet and calls the `SkyFunction` again, thus going back to (1).
4. Constructing the resulting `SkyValue`

A consequence of this is that if not all dependencies are available in (3), the function needs to be completely restarted and thus computation needs to be re-done. This is obviously inefficient. We work around this in a number of ways:

1. Declaring dependencies of `SkyFunction`s in groups so that if a function has, say, 10 dependencies, it only needs to restart once instead of ten times.
2. Splitting `SkyFunction`s so that one function does not need to be restarted many times. This has the side effect of interning data into Skyframe that may be internal to the `SkyFunction`, thus increasing memory use.
3. Using caches "behind the back of Skyframe" to keep state (e.g. the state of actions being executed in `ActionExecutionFunction.stateMap` . In the extreme, this ends up resulting in writing code in continuation-passing style (e.g. action execution), which does not help readability.

Of course, these are all just workarounds for the limitations of Skyframe, which is mostly a consequence of the fact that Java doesn't support lightweight threads and that we routinely have hundreds of thousands of in-flight Skyframe nodes.

Starlark

Starlark is the domain-specific language people use to configure and extend Bazel. It's conceived as a restricted subset of Python that has far fewer types, more restrictions on control flow, and most importantly, strong immutability guarantees to enable concurrent reads.

It is not Turing-complete, which discourages some (but not all) users from trying to accomplish general programming tasks within the language.

Starlark is implemented in the `com.google.devtools.build.lib.syntax` package. It also has an independent Go implementation here. The Java implementation used in Bazel is currently an interpreter.

Starlark is used in four contexts:

1. **The BUILD language.** This is where new rules are defined. Starlark code running in this context only has access to the contents of the BUILD file itself and Starlark files loaded by it.
2. **Rule definitions.** This is how new rules (e.g. support for a new language) are defined. Starlark code running in this context has access to the configuration and data provided by its direct dependencies (more on this later).
3. **The WORKSPACE file.** This is where external repositories (code that's not in the main source tree) are defined.
4. **Repository rule definitions.** This is where new external repository types are defined. Starlark code running in this context can run arbitrary code on the machine where Bazel is running, and reach outside the workspace.

The dialects available for BUILD and .bzl files are slightly different because they express different things. A list of differences is available here.

More information about Starlark is available here.

The loading/analysis phase

The loading/analysis phase is where Bazel determines what actions are needed to build a particular rule. Its basic unit is a "configured target", which is, quite sensibly, a (target, configuration) pair.

It's called the "loading/analysis phase" because it can be split into two distinct parts, which used to be serialized, but they can now overlap in time:

1. Loading packages, that is, turning BUILD files into the `Package` objects that represent them
2. Analyzing configured targets, that is, running the implementation of the rules to produce the action graph

Each configured target in the transitive closure of the configured targets requested on the command line must be analyzed bottom-up, i.e. leaf nodes first, then up to the ones on the command line. The inputs to the analysis of a single configured target are:

1. **The configuration.** ("how" to build that rule; for example, the target platform but also things like command line options the user wants to be passed to the C++ compiler)
2. **The direct dependencies.** Their transitive info providers are available to the rule being analyzed. They are called like that because they provide a "roll-up" of the information in the transitive closure of the configured target, e.g. all the .jar files on the classpath or all the .o files that need to be linked into a C++ binary)
3. **The target itself**. This is the result of loading the package the target is in. For rules, this includes its attributes, which is usually what matters.
4. **The implementation of the configured target.** For rules, this can either be in Starlark or in Java. All non-rule configured targets are implemented in Java.

The output of analyzing a configured target is:

1. The transitive info providers that configured targets that depend on it can access
2. The artifacts it can create and the actions that produce them.

The API offered to Java rules is `RuleContext`, which is the equivalent of the `ctx` argument of Starlark rules. Its API is more powerful, but at the same time, it's easier to do Bad Things™, for example to write code whose time or space complexity is quadratic (or worse), to make the Bazel server crash with a Java exception or to violate invariants (e.g. by inadvertently modifying an `Options` instance or by making a configured target mutable)

The algorithm that determines the direct dependencies of a configured target lives in `DependencyResolver.dependentNodeMap()`.

Configurations

Configurations are the "how" of building a target: for what platform, with what command line options, etc.

The same target can be built for multiple configurations in the same build. This is useful, for example, when the same code is used for a tool that's run during the build and for the target code and we are cross-compiling or when we are building a fat Android app (one that contains native code for multiple CPU architectures)

Conceptually, the configuration is a `BuildOptions` instance. However, in practice, `BuildOptions` is wrapped by `BuildConfiguration` that provides additional sundry pieces of functionality. It propagates from the top of the dependency graph to the bottom. If it changes, the build needs to be re-analyzed.

This results in anomalies like having to re-analyze the whole build if e.g. the number of requested test runs changes, even though that only affects test targets (we have plans to "trim" configurations so that this is not the case, but it's not ready yet)

When a rule implementation needs part of the configuration, it needs to declare it in its definition using `RuleClass.Builder.requiresConfigurationFragments()`. This is both to avoid mistakes (e.g. Python rules using the Java fragment) and to facilitate configuration trimming so that e.g. if Python options change, C++ targets don't need to be re-analyzed.

The configuration of a rule is not necessarily the same as that of its "parent" rule. The process of changing the configuration in a dependency edge is called a "configuration transition". It can happen in two places:

1. On a dependency edge. These transitions are specified in `Attribute.Builder.cfg()` and are functions from a `Rule` (where the transition happens) and a `BuildOptions` (the original configuration) to one or more `BuildOptions` (the output configuration).
2. On any incoming edge to a configured target. These are specified in `RuleClass.Builder.cfg()`.

The relevant classes are `TransitionFactory` and `ConfigurationTransition`.

Configuration transitions are used, for example:

1. To declare that a particular dependency is used during the build and it should thus be built in the execution architecture
2. To declare that a particular dependency must be built for multiple architectures (e.g. for native code in fat Android APKs)

If a configuration transition results in multiple configurations, it's called a *split transition*.

Configuration transitions can also be implemented in Starlark (documentation here)

Transitive info providers

Transitive info providers are a way (and the *only* way) for configured targets to tell things about other configured targets that depend on it. The reason why "transitive" is in their name is that this is usually some sort of roll-up of the transitive closure of a configured target.

There is generally a 1:1 correspondence between Java transitive info providers and Starlark ones (the exception is `DefaultInfo` which is an amalgamation of `FileProvider`, `FilesToRunProvider` and `RunfilesProvider` because that API was deemed to be more Starlark-ish than a direct transliteration of the Java one). Their key is one of the following things:

1. A Java Class object. This is only available for providers that are not accessible from Starlark. These providers are a subclass of `TransitiveInfoProvider`.
2. A string. This is legacy and heavily discouraged since it's susceptible to name clashes. Such transitive info providers are direct subclasses of `build.lib.packages.Info`.

3. A provider symbol. This can be created from Starlark using the `provider()` function and is the recommended way to create new providers. The symbol is represented by a `Provider.Key` instance in Java.

New providers implemented in Java should be implemented using `BuiltinProvider`. `NativeProvider` is deprecated (we haven't had time to remove it yet) and `TransitiveInfoProvider` subclasses cannot be accessed from Starlark.

Configured targets

Configured targets are implemented as `RuleConfiguredTargetFactory`. There is a subclass for each rule class implemented in Java. Starlark configured targets are created through `SkylarkRuleConfiguredTargetUtil.buildRule()`.

Configured target factories should use `RuleConfiguredTargetBuilder` to construct their return value. It consists of the following things:

1. Their `filesToBuild`, i.e. the hazy concept of "the set of files this rule represents". These are the files that get built when the configured target is on the command line or in the srcs of a genrule.
2. Their runfiles, regular and data.
3. Their output groups. These are various "other sets of files" the rule can build. They can be accessed using the output_group attribute of the filegroup rule in BUILD and using the `OutputGroupInfo` provider in Java.

Runfiles

Some binaries need data files to run. A prominent example is tests that need input files. This is represented in Bazel by the concept of "runfiles". A "runfiles tree" is a directory tree of the data files for a particular binary. It is created in the file system as a symlink tree with individual symlinks pointing to the files in the source of output trees.

A set of runfiles is represented as a `Runfiles` instance. It is conceptually a map from the path of a file in the runfiles tree to the `Artifact` instance that represents it. It's a little more complicated than a single `Map` for two reasons:

- Most of the time, the runfiles path of a file is the same as its execpath. We use this to save some RAM.
- There are various legacy kinds of entries in runfiles trees, which also need to be represented.

Runfiles are collected using `RunfilesProvider`: an instance of this class represents the runfiles a configured target (e.g. a library) and its transitive closure needs and they are gathered like a nested set (in fact, they are implemented using nested sets under the cover): each target unions the runfiles of its dependencies, adds some of its own, then sends the resulting set

upwards in the dependency graph. A `RunfilesProvider` instance contains two `Runfiles` instances, one for when the rule is depended on through the "data" attribute and one for every other kind of incoming dependency. This is because a target sometimes presents different runfiles when depended on through a data attribute than otherwise. This is undesired legacy behavior that we haven't gotten around removing yet.

Runfiles of binaries are represented as an instance of `RunfilesSupport`. This is different from `Runfiles` because `RunfilesSupport` has the capability of actually being built (unlike `Runfiles`, which is just a mapping). This necessitates the following additional components:

- **The input runfiles manifest.** This is a serialized description of the runfiles tree. It is used as a proxy for the contents of the runfiles tree and Bazel assumes that the runfiles tree changes if and only if the contents of the manifest change.
- **The output runfiles manifest.** This is used by runtime libraries that handle runfiles trees, notably on Windows, which sometimes doesn't support symbolic links.
- **The runfiles middleman.** In order for a runfiles tree to exist, one needs to build the symlink tree and the artifact the symlinks point to. In order to decrease the number of dependency edges, the runfiles middleman can be used to represent all these.
- **Command line arguments** for running the binary whose runfiles the `RunfilesSupport` object represents.

Aspects

Aspects are a way to "propagate computation down the dependency graph". They are described for users of Bazel here. A good motivating example is protocol buffers: a `proto_library` rule should not know about any particular language, but building the implementation of a protocol buffer message (the “basic unit” of protocol buffers) in any programming language should be coupled to the `proto_library` rule so that if two targets in the same language depend on the same protocol buffer, it gets built only once.

Just like configured targets, they are represented in Skyframe as a `SkyValue` and the way they are constructed is very similar to how configured targets are built: they have a factory class called `ConfiguredAspectFactory` that has access to a `RuleContext`, but unlike configured target factories, it also knows about the configured target it is attached to and its providers.

The set of aspects propagated down the dependency graph is specified for each attribute using the `Attribute.Builder.aspects()` function. There are a few confusingly-named classes that participate in the process:

1. `AspectClass` is the implementation of the aspect. It can be either in Java (in which case it's a subclass) or in Starlark (in which case it's an instance of `SkylarkAspectClass`). It's analogous to `RuleConfiguredTargetFactory`.
2. `AspectDefinition` is the definition of the aspect; it includes the providers it requires, the providers it provides and contains a reference to its implementation, i.e. the appropriate `AspectClass` instance. It's analogous to `RuleClass`.

3. `AspectParameters` is a way to parametrize an aspect that is propagated down the dependency graph. It's currently a string to string map. A good example of why it's useful is protocol buffers: if a language has multiple APIs, the information as to which API the protocol buffers should be built for should be propagated down the dependency graph.
4. `Aspect` represents all the data that's needed to compute an aspect that propagates down the dependency graph. It consists of the aspect class, its definition and its parameters.
5. `RuleAspect` is the function that determines which aspects a particular rule should propagate. It's a `Rule` -> `Aspect` function.

A somewhat unexpected complication is that aspects can attach to other aspects; for example, an aspect collecting the classpath for a Java IDE will probably want to know about all the .jar files on the classpath, but some of them are protocol buffers. In that case, the IDE aspect will want to attach to the (`proto_library` rule + Java proto aspect) pair.

The complexity of aspects on aspects is captured in the class `AspectCollection`.

Platforms and toolchains

Bazel supports multi-platform builds, that is, builds where there may be multiple architectures where build actions run and multiple architectures for which code is built. These architectures are referred to as *platforms* in Bazel parlance (full documentation here)

A platform is described by a key-value mapping from *constraint settings* (e.g. the concept of "CPU architecture") to *constraint values* (e.g. a particular CPU like x86_64). We have a "dictionary" of the most commonly used constraint settings and values in the `@platforms` repository.

The concept of *toolchain* comes from the fact that depending on what platforms the build is running on and what platforms are targeted, one may need to use different compilers; for example, a particular C++ toolchain may run on a specific OS and be able to target some other OSes. Bazel must determine the C++ compiler that is used based on the set execution and target platform (documentation for toolchains here).

In order to do this, toolchains are annotated with the set of execution and target platform constraints they support. In order to do this, the definition of a toolchain are split into two parts:

1. A `toolchain()` rule that describes the set of execution and target constraints a toolchain supports and tells what kind (e.g. C++ or Java) of toolchain it is (the latter is represented by the `toolchain_type()` rule)
2. A language-specific rule that describes the actual toolchain (e.g. `cc_toolchain()`)

This is done in this way because we need to know the constraints for every toolchain in order to do toolchain resolution and language-specific `*_toolchain()` rules contain much more information than that, so they take more time to load.

Execution platforms are specified in one of the following ways:

1. In the WORKSPACE file using the `register_execution_platforms()` function
2. On the command line using the --extra_execution_platforms command line option

The set of available execution platforms is computed in `RegisteredExecutionPlatformsFunction`.

The target platform for a configured target is determined by `PlatformOptions.computeTargetPlatform()`. It's a list of platforms because we eventually want to support multiple target platforms, but it's not implemented yet.

The set of toolchains to be used for a configured target is determined by `ToolchainResolutionFunction`. It is a function of:

- The set of registered toolchains (in the WORKSPACE file and the configuration)
- The desired execution and target platforms (in the configuration)
- The set of toolchain types that are required by the configured target (in `UnloadedToolchainContextKey)`
- The set of execution platform constraints of the configured target (the `exec_compatible_with` attribute) and the configuration (`--experimental_add_exec_constraints_to_targets`), in `UnloadedToolchainContextKey`

Its result is an `UnloadedToolchainContext`, which is essentially a map from toolchain type (represented as a `ToolchainTypeInfo` instance) to the label of the selected toolchain. It's called "unloaded" because it does not contain the toolchains themselves, only their labels.

Then the toolchains are actually loaded using `ResolvedToolchainContext.load()` and used by the implementation of the configured target that requested them.

We also have a legacy system that relies on there being one single "host" configuration and target configurations being represented by various configuration flags, e.g. `--cpu`. We are gradually transitioning to the above system. In order to handle cases where people rely on the legacy configuration values, we have implemented "platform mappings" to translate between the legacy flags and the new-style platform constraints. Their code is in `PlatformMappingFunction` and uses a non-Starlark "little language".

Constraints

Sometimes one wants to designate a target as being compatible with only a few platforms. Bazel has (unfortunately) multiple mechanisms to achieve this end:

- Rule-specific constraints
- `environment_group()` / `environment()`
- Platform constraints

Rule-specific constraints are mostly used within Google for Java rules; they are on their way out and they are not available in Bazel, but the source code may contain references to it. The attribute that governs this is called `constraints=` .

`environment_group()` and `environment()`

These rules are a legacy mechanism and are not widely used.

All build rules can declare which "environments" they can be built for, where a "environment" is an instance of the `environment()` rule.

There are various ways supported environments can be specified for a rule:

1. Through the `restricted_to=` attribute. This is the most direct form of specification; it declares the exact set of environments the rule supports for this group.
2. Through the `compatible_with=` attribute. This declares environments a rule supports in addition to "standard" environments that are supported by default.
3. Through the package-level attributes `default_restricted_to=` and `default_compatible_with=`.
4. Through default specifications in `environment_group()` rules. Every environment belongs to a group of thematically related peers (e.g. "CPU architectures", "JDK versions" or "mobile operating systems"). The definition of an environment group includes which of these environments should be supported by "default" if not otherwise specified by the `restricted_to=` / `environment()` attributes. A rule with no such attributes inherits all defaults.
5. Through a rule class default. This overrides global defaults for all instances of the given rule class. This can be used, for example, to make all `*_test` rules testable without each instance having to explicitly declare this capability.

`environment()` is implemented as a regular rule whereas `environment_group()` is both a subclass of `Target` but not `Rule` (`EnvironmentGroup`) and a function that is available by default from Starlark (`StarlarkLibrary.environmentGroup()`) which eventually creates an eponymous target. This is to avoid a cyclic dependency that would arise because each environment needs to declare the environment group it belongs to and each environment group needs to declare its default environments.

A build can be restricted to a certain environment with the `--target_environment` command line option.

The implementation of the constraint check is in `RuleContextConstraintSemantics` and `TopLevelConstraintSemantics`.

Platform constraints

The current "official" way to describe what platforms a target is compatible with is by using the same constraints used to describe toolchains and platforms. It's under review in pull request #10945.

Visibility

If you work on a large codebase with a lot of developers (like at Google), you don't necessarily want everyone else to be able to depend on your code so that you retain the liberty to change things that you deem to be implementation details (otherwise, as per Hyrum's law, people *will* come to depend on all parts of your code).

Bazel supports this by the mechanism called *visibility:* you can declare that a particular rule can only be depended on using the visibility attribute (documentation here). This attribute is a little special because unlike every other attribute, the set of dependencies it generates is not simply the set of labels listed (yes, this is a design flaw).

This is implemented in the following places:

- The `RuleVisibility` interface represents a visibility declaration. It can be either a constant (fully public or fully private) or a list of labels.
- Labels can refer to either package groups (predefined list of packages), to packages directly (`//pkg:__pkg__`) or subtrees of packages (`//pkg:__subpackages__`). This is different from the command line syntax, which uses `//pkg:*` or `//pkg/...`.
- Package groups are implemented as their own target and configured target types (`PackageGroup` and `PackageGroupConfiguredTarget`). We could probably replace these with simple rules if we wanted to.
- The conversion from visibility label lists to dependencies is done in `DependencyResolver.visitTargetVisibility` and a few other miscellaneous places.
- The actual check is done in `CommonPrerequisiteValidator.validateDirectPrerequisiteVisibility()`

Nested sets

Oftentimes, a configured target aggregates a set of files from its dependencies, adds its own, and wraps the aggregate set into a transitive info provider so that configured targets that depend on it can do the same. Examples:

- The C++ header files used for a build
- The object files that represent the transitive closure of a `cc_library`
- The set of .jar files that need to be on the classpath for a Java rule to compile or run

- The set of Python files in the transitive closure of a Python rule

If we did this the naive way by using e.g. `List` or `Set`, we'd end up with quadratic memory usage: if there is a chain of N rules and each rule adds a file, we'd have 1+2+...+N collection members.

In order to get around this problem, we came up with the concept of a `NestedSet`. It's a data structure that is composed of other `NestedSet` instances and some members of its own, thereby forming a directed acyclic graph of sets. They are immutable and their members can be iterated over. We define multiple iteration order (`NestedSet.Order`): preorder, postorder, topological (a node always comes after its ancestors) and "don't care, but it should be the same each time".

The same data structure is called `depset` in Starlark.

Artifacts and Actions

The actual build consists of a set of commands that need to be run to produce the output the user wants. The commands are represented as instances of the class `Action` and the files are represented as instances of the class `Artifact`. They are arranged in a bipartite, directed, acyclic graph called the "action graph".

Artifacts come in two kinds: source artifacts (i.e. ones that are available before Bazel starts executing) and derived artifacts (ones that need to be built). Derived artifacts can themselves be multiple kinds:

1. **Regular artifacts.** These are checked for up-to-dateness by computing their checksum, with mtime as a shortcut; we don't checksum the file if its ctime hasn't changed.
2. **Unresolved symlink artifacts.** These are checked for up-to-dateness by calling readlink(). Unlike regular artifacts, these can be dangling symlinks. Usually used in cases where one then packs up some files into an archive of some sort.
3. **Tree artifacts.** These are not single files, but directory trees. They are checked for up-to-dateness by checking the set of files in it and their contents. They are represented as a `TreeArtifact`.
4. **Constant metadata artifacts.** Changes to these artifacts don't trigger a rebuild. This is used exclusively for build stamp information: we don't want to do a rebuild just because the current time changed.

There is no fundamental reason why source artifacts cannot be tree artifacts or unresolved symlink artifacts, it's just that we haven't implemented it yet (we should, though -- referencing a source directory in a BUILD file is one of the few known long-standing incorrectness issues with Bazel; we have an implementation that kind of works which is enabled by the `BAZEL_TRACK_SOURCE_DIRECTORIES=1` JVM property)

A notable kind of `Artifact` are middlemen. They are indicated by `Artifact` instances that are the outputs of `MiddlemanAction`. They are used to special-case some things:

- Aggregating middlemen are used to group artifacts together. This is so that if a lot of actions use the same large set of inputs, we don't have N*M dependency edges, only N+M (they are being replaced with nested sets)
- Scheduling dependency middlemen ensure that an action runs before another. They are mostly used for linting but also for C++ compilation (see `CcCompilationContext.createMiddleman()` for an explanation)
- Runfiles middlemen are used to ensure the presence of a runfiles tree so that one does not separately need to depend on the output manifest and every single artifact referenced by the runfiles tree.

Actions are best understood as a command that needs to be run, the environment it needs and the set of outputs it produces. The following things are the main components of the description of an action:

- The command line that needs to be run
- The input artifacts it needs
- The environment variables that need to be set
- Annotations that describe the environment (e.g. platform) it needs to run in

There are also a few other special cases, like writing a file whose content is known to Bazel. They are a subclass of `AbstractAction`. Most of the actions are a `SpawnAction` or a `StarlarkAction` (the same, they should arguably not be separate classes), although Java and C++ have their own action types (`JavaCompileAction`, `CppCompileAction` and `CppLinkAction`).

We eventually want to move everything to `SpawnAction`; `JavaCompileAction` is pretty close, but C++ is a bit of a special-case due to .d file parsing and include scanning.

The action graph is mostly "embedded" into the Skyframe graph: conceptually, the execution of an action is represented as an invocation of `ActionExecutionFunction`. The mapping from an action graph dependency edge to a Skyframe dependency edge is described in `ActionExecutionFunction.getInputDeps()` and `Artifact.key()` and has a few optimizations in order to keep the number of Skyframe edges low:

- Derived artifacts do not have their own `SkyValue`s. Instead, `Artifact.getGeneratingActionKey()` is used to find out the key for the action that generates it
- Nested sets have their own Skyframe key.

Shared actions

Some actions are generated by multiple configured targets; Starlark rules are more limited since they are only allowed to put their derived actions into a directory determined by their

configuration and their package (but even so, rules in the same package can conflict), but rules implemented in Java can put derived artifacts anywhere.

This is considered to be a misfeature, but getting rid of it is really hard because it produces significant savings in execution time when e.g. a source file needs to be processed somehow and that file is referenced by multiple rules (handwave-handwave). This comes at the cost of some RAM: each instance of a shared action needs to be stored in memory separately.

If two actions generate the same output file, they must be exactly the same: have the same inputs, the same outputs and run the same command line. This equivalence relation is implemented in `Actions.canBeShared()` and it is verified between the analysis and execution phases by looking at every Action. This is implemented in `SkyframeActionExecutor.findAndStoreArtifactConflicts()` and is one of the few places in Bazel that requires a "global" view of the build.

The execution phase

This is when Bazel actually starts running build actions, i.e. commands that produce outputs.

The first thing Bazel does after the analysis phase is to determine what Artifacts need to be built. The logic for this is encoded in `TopLevelArtifactHelper`; roughly speaking, it's the `filesToBuild` of the configured targets on the command line and the contents of a special output group for the explicit purpose of expressing "if this target is on the command line, build these artifacts".

The next step is creating the execution root. Since Bazel has the option to read source packages from different locations in the file system (`--package_path`), it needs to provide locally executed actions with a full source tree. This is handled by the class `SymlinkForest` and works by taking note of every target used in the analysis phase and building up a single directory tree that symlinks every package with a used target from its actual location. An alternative would be to pass the correct paths to commands (taking `--package_path` into account). This is undesirable because:

- It changes action command lines when a package is moved from a package path entry to another (used to be a common occurrence)
- It results in different command lines if an action is run remotely than if it's run locally
- It requires a command line transformation specific to the tool in use (consider the difference between e.g. Java classpaths and C++ include paths)
- Changing the command line of an action invalidates its action cache entry
- `--package_path` is slowly and steadily being deprecated

Then, Bazel starts traversing the action graph (the bipartite, directed graph composed of actions and their input and output artifacts) and running actions. The execution of each action is represented by an instance of the `SkyValue` class `ActionExecutionValue`.

Since running an action is expensive, we have a few layers of caching that can be hit behind Skyframe:

- `ActionExecutionFunction.stateMap` contains data to make Skyframe restarts of `ActionExecutionFunction` cheap
- The local action cache contains data about the state of the file system
- Remote execution systems usually also contain their own cache

The local action cache

This cache is another layer that sits behind Skyframe; even if an action is re-executed in Skyframe, it can still be a hit in the local action cache. It represents the state of the local file system and it's serialized to disk which means that when one starts up a new Bazel server, one can get local action cache hits even though the Skyframe graph is empty.

This cache is checked for hits using the method `ActionCacheChecker.getTokenIfNeedToExecute()`.

Contrary to its name, it's a map from the path of a derived artifact to the action that emitted it. The action is described as:

1. The set of its input and output files and their checksum
2. Its "action key", which is usually the command line that was executed, but in general, represents everything that's not captured by the checksum of the input files (e.g. for `FileWriteAction`, it's the checksum of the data that's written)

There is also a highly experimental “top-down action cache” that is still under development, which uses transitive hashes to avoid going to the cache as many times.

Input discovery and input pruning

Some actions are more complicated than just having a set of inputs. Changes to the set of inputs of an action come in two forms:

- An action may discover new inputs before its execution or decide that some of its inputs are not actually necessary. The canonical example is C++, where it's better to make an educated guess about what header files a C++ file uses from its transitive closure so that we don't heed to send every file to remote executors; therefore, we have an option not to register every header file as an "input", but scan the source file for transitively included headers and only mark those header files as inputs that are mentioned in `#include` statements (we overestimate so that we don't need to implement a full C preprocessor)

- An action may realize that some files were not used during its execution. In C++, this is called ".d files": the compiler tells which header files were used after the fact, and in order to avoid the embarrassment of having worse incrementality than Make, Bazel makes use of this fact. This offers a better estimate than the include scanner because it relies on the compiler.

These are implemented using methods on Action:

1. `Action.discoverInputs()` is called. It should return a nested set of Artifacts that are determined to be required. These must be source artifacts so that there are no dependency edges in the action graph that don't have an equivalent in the configured target graph.
2. The action is executed by calling `Action.execute()`.
3. At the end of `Action.execute()`, the action can call `Action.updateInputs()` to tell Bazel that not all of its inputs were needed. This can result in incorrect incremental builds if a used input is reported as unused.

When an action cache returns a hit on a fresh Action instance (e.g. created after a server restart), Bazel calls `updateInputs()` itself so that the set of inputs reflects the result of input discovery and pruning done before.

Starlark actions can make use of the facility to declare some inputs as unused using the `unused_inputs_list=` argument of `ctx.actions.run()`.

Various ways to run actions: Strategies/ActionContexts

Some actions can be run in different ways. For example, a command line can be executed locally, locally but in various kinds of sandboxes, or remotely. The concept that embodies this is called an `ActionContext` (or `Strategy`, since we successfully went only halfway with a rename...)

The life cycle of an action context is as follows:

1. When the execution phase is started, `BlazeModule` instances are asked what action contexts they have. This happens in the constructor of `ExecutionTool`. Action context types are identified by a Java `Class` instance that refers to a sub-interface of `ActionContext` and which interface the action context must implement.
2. The appropriate action context is selected from the available ones and is forwarded to `ActionExecutionContext` and `BlazeExecutor` .
3. Actions request contexts using `ActionExecutionContext.getContext()` and `BlazeExecutor.getStrategy()` (there should really be only one way to do it...)

Strategies are free to call other strategies to do their jobs; this is used, for example, in the dynamic strategy that starts actions both locally and remotely, then uses whichever finishes first.

One notable strategy is the one that implements persistent worker processes (`WorkerSpawnStrategy`). The idea is that some tools have a long startup time and should therefore be reused between actions instead of starting one anew for every action (This does represent a potential correctness issue, since Bazel relies on the promise of the worker process that it doesn't carry observable state between individual requests)

If the tool changes, the worker process needs to be restarted. Whether a worker can be reused is determined by computing a checksum for the tool used using `WorkerFilesHash`. It relies on knowing which inputs of the action represent part of the tool and which represent inputs; this is determined by the creator of the Action: `Spawn.getToolFiles()` and the runfiles of the `Spawn` are counted as parts of the tool.

More information about strategies (or action contexts!):

- Information about various strategies for running actions is available here.
- Information about the dynamic strategy, one where we run an action both locally and remotely to see whichever finishes first is available here.
- Information about the intricacies of executing actions locally is available here.

The local resource manager

Bazel *can* run many actions in parallel. The number of local actions that *should* be run in parallel differs from action to action: the more resources an action requires, the less instances should be running at the same time to avoid overloading the local machine.

This is implemented in the class `ResourceManager`: each action has to be annotated with an estimate of the local resources it requires in the form of a `ResourceSet` instance (CPU and RAM). Then when action contexts do something that requires local resources, they call `ResourceManager.acquireResources()` and are blocked until the required resources are available.

A more detailed description of local resource management is available here.

The structure of the output directory

Each action requires a separate place in the output directory where it places its outputs. The location of derived artifacts is usually as follows:

```
$EXECROOT/bazel-out/<configuration>/bin/<package>/<artifact name>
```

How is the name of the directory that is associated with a particular configuration determined? There are two conflicting desirable properties:

1. If two configurations can occur in the same build, they should have different directories so that both can have their own version of the same action; otherwise, if the two

configurations disagree about e.g. the command line of an action producing the same output file, Bazel doesn't know which action to choose (an "action conflict")
2. If two configurations represent "roughly" the same thing, they should have the same name so that actions executed in one can be reused for the other if the command lines match: for example, changes to the command line options to the Java compiler should not result in C++ compile actions being re-run.

So far, we have not come up with a principled way of solving this problem, which has similarities to the problem of configuration trimming. A longer discussion of options is available here. The main problematic areas are Starlark rules (whose authors usually aren't intimately familiar with Bazel) and aspects, which add another dimension to the space of things that can produce the "same" output file.

The current approach is that the path segment for the configuration is `<CPU>-<compilation mode>` with various suffixes added so that configuration transitions implemented in Java don't result in action conflicts. In addition, a checksum of the set of Starlark configuration transitions is added so that users can't cause action conflicts. It is far from perfect. This is implemented in `OutputDirectories.buildMnemonic()` and relies on each configuration fragment adding its own part to the name of the output directory.

Tests

Bazel has rich support for running tests. It supports:

- Running tests remotely (if a remote execution backend is available)
- Running tests multiple times in parallel (for deflaking or gathering timing data)
- Sharding tests (splitting test cases in same test over multiple processes for speed)
- Re-running flaky tests
- Grouping tests into test suites

Tests are regular configured targets that have a TestProvider, which describes how the test should be run:

- The artifacts whose building result in the test being run. This is a "cache status" file that contains a serialized `TestResultData` message
- The number of times the test should be run
- The number of shards the test should be split into
- Some parameters about how the test should be run (e.g. the test timeout)

Determining which tests to run

Determining which tests are run is an elaborate process.

First, during target pattern parsing, test suites are recursively expanded. The expansion is implemented in `TestsForTargetPatternFunction`. A somewhat surprising wrinkle is that if a test suite declares no tests, it refers to *every* test in its package. This is implemented in `Package.beforeBuild()` by adding an implicit attribute called `$implicit_tests` to test suite rules.

Then, tests are filtered for size, tags, timeout and language according to the command line options. This is implemented in `TestFilter` and is called from `TargetPatternPhaseFunction.determineTests()` during target parsing and the result is put into `TargetPatternPhaseValue.getTestsToRunLabels()`. The reason why rule attributes which can be filtered for are not configurable is that this happens before the analysis phase, therefore, the configuration is not available.

This is then processed further in `BuildView.createResult()`: targets whose analysis failed are filtered out and tests are split into exclusive and non-exclusive tests. It's then put into `AnalysisResult`, which is how `ExecutionTool` knows which tests to run.

In order to lend some transparency to this elaborate process, the `tests()` query operator (implemented in `TestsFunction`) is available to tell which tests are run when a particular target is specified on the command line. It's unfortunately a reimplementation, so it probably deviates from the above in multiple subtle ways.

Running tests

The way the tests are run is by requesting cache status artifacts. This then results in the execution of a `TestRunnerAction`, which eventually calls the `TestActionContext` chosen by the `--test_strategy` command line option that runs the test in the requested way.

Tests are run according to an elaborate protocol that uses environment variables to tell tests what's expected from them. A detailed description of what Bazel expects from tests and what tests can expect from Bazel is available here. At the simplest, an exit code of 0 means success, anything else means failure.

In addition to the cache status file, each test process emits a number of other files. They are put in the "test log directory" which is the subdirectory called `testlogs` of the output directory of the target configuration:

- `test.xml`, a JUnit-style XML file detailing the individual test cases in the test shard
- `test.log`, the console output of the test. stdout and stderr are not separated.
- `test.outputs`, the "undeclared outputs directory"; this is used by tests that want to output files in addition to what they print to the terminal.

There are two things that can happen during test execution that cannot during building regular targets: exclusive test execution and output streaming.

Some tests need to be executed in exclusive mode, i.e. not in parallel with other tests. This can be elicited either by adding `tags=["exclusive"]` to the test rule or running the test with `--test_strategy=exclusive` . Each exclusive test is run by a separate Skyframe invocation requesting the execution of the test after the "main" build. This is implemented in `SkyframeExecutor.runExclusiveTest()`.

Unlike regular actions, whose terminal output is dumped when the action finishes, the user can request the output of tests to be streamed so that they get informed about the progress of a long-running test. This is specified by the `--test_output=streamed` command line option and implies exclusive test execution so that outputs of different tests are not interspersed.

This is implemented in the aptly-named `StreamedTestOutput` class and works by polling changes to the `test.log` file of the test in question and dumping new bytes to the terminal where Bazel rules.

Results of the executed tests are available on the event bus by observing various events (e.g. `TestAttempt`, `TestResult` or `TestingCompleteEvent`). They are dumped to the Build Event Protocol and they are emitted to the console by `AggregatingTestListener`.

Coverage collection

Coverage is reported by the tests in LCOV format in the files `bazel-testlogs/$PACKAGE/$TARGET/coverage.dat` .

To collect coverage, each test execution is wrapped in a script called `collect_coverage.sh` .

This script sets up the environment of the test to enable coverage collection and determine where the coverage files are written by the coverage runtime(s). It then runs the test. A test may itself run multiple subprocesses and consist of parts written in multiple different programming languages (with separate coverage collection runtimes). The wrapper script is responsible for converting the resulting files to LCOV format if necessary, and merges them into a single file.

The interposition of `collect_coverage.sh` is done by the test strategies and requires `collect_coverage.sh` to be on the inputs of the test. This is accomplished by the implicit attribute `:coverage_support` which is resolved to the value of the configuration flag `--coverage_support` (see `TestConfiguration.TestOptions.coverageSupport`)

Some languages do offline instrumentation, meaning that the coverage instrumentation is added at compile time (e.g. C++) and others do online instrumentation, meaning that coverage instrumentation is added at execution time.

Another core concept is *baseline coverage*. This is the coverage of a library, binary, or test if no code in it was run. The problem it solves is that if you want to compute the test coverage for a binary, it is not enough to merge the coverage of all of the tests because there may be code in the binary that is not linked into any test. Therefore, what we do is to emit a coverage file for every binary which contains only the files we collect coverage for with no covered lines. The baseline coverage file for a target is at `bazel-testlogs/$PACKAGE/$TARGET/baseline_coverage.dat`. It is also generated for binaries and libraries in addition to tests if you pass the `--nobuild_tests_only` flag to Bazel.

Baseline coverage is currently broken.

We track two groups of files for coverage collection for each rule: the set of instrumented files and the set of instrumentation metadata files.

The set of instrumented files is just that, a set of files to instrument. For online coverage runtimes, this can be used at runtime to decide which files to instrument. It is also used to implement baseline coverage.

The set of instrumentation metadata files is the set of extra files a test needs to generate the LCOV files Bazel requires from it. In practice, this consists of runtime-specific files; for example, gcc emits .gcno files during compilation. These are added to the set of inputs of test actions if coverage mode is enabled.

Whether or not coverage is being collected is stored in the `BuildConfiguration`. This is handy because it is an easy way to change the test action and the action graph depending on this bit, but it also means that if this bit is flipped, all targets need to be re-analyzed (some languages, e.g. C++ require different compiler options to emit code that can collect coverage, which mitigates this issue somewhat, since then a re-analysis is needed anyway).

The coverage support files are depended on through labels in an implicit dependency so that they can be overridden by the invocation policy, which allows them to differ between the different versions of Bazel. Ideally, these differences would be removed, and we standardized on one of them.

We also generate a "coverage report" which merges the coverage collected for every test in a Bazel invocation. This is handled by `CoverageReportActionFactory` and is called from `BuildView.createResult()`. It gets access to the tools it needs by looking at the `:coverage_report_generator` attribute of the first test that is executed.

The query engine

Bazel has a little language used to ask it various things about various graphs. The following query kinds are provided:

- `bazel query` is used to investigate the target graph
- `bazel cquery` is used to investigate the configured target graph
- `bazel aquery` is used to investigate the action graph

Each of these is implemented by subclassing `AbstractBlazeQueryEnvironment`. Additional additional query functions can be done by subclassing `QueryFunction` . In order to allow streaming query results, instead of collecting them to some data structure, a `query2.engine.Callback` is passed to `QueryFunction`, which calls it for results it wants to return.

The result of a query can be emitted in various ways: labels, labels and rule classes, XML, protobuf and so on. These are implemented as subclasses of `OutputFormatter`.

A subtle requirement of some query output formats (proto, definitely) is that Bazel needs to emit *all* the information that package loading provides so that one can diff the output and determine whether a particular target has changed. As a consequence, attribute values need to be serializable, which is why there are only so few attribute types without any attributes having complex Starlark values. The usual workaround is to use a label, and attach the complex information to the rule with that label. It's not a very satisfying workaround and it would be very nice to lift this requirement.

The module system

Bazel can be extended by adding modules to it. Each module must subclass `BlazeModule` (the name is a relic of the history of Bazel when it used to be called Blaze) and gets information about various events during the execution of a command.

They are mostly used to implement various pieces of "non-core" functionality that only some versions of Bazel (e.g. the one we use at Google) need:

- Interfaces to remote execution systems
- New commands

The set of extension points `BlazeModule` offers is somewhat haphazard. Don't use it as an example of good design principles.

The event bus

The main way BlazeModules communicate with the rest of Bazel is by an event bus (`EventBus`): a new instance is created for every build, various parts of Bazel can post events to it and modules can register listeners for the events they are interested in. For example, the following things are represented as events:

- The list of build targets to be built has been determined (`TargetParsingCompleteEvent`)
- The top-level configurations have been determined (`BuildConfigurationEvent`)
- A target was built, successfully or not (`TargetCompleteEvent`)
- A test was run (`TestAttempt`, `TestSummary`)

Some of these events are represented outside of Bazel in the Build Event Protocol (they are `BuildEvent`s). This allows not only `BlazeModule`s, but also things outside the Bazel process to observe the build. They are accessible either as a file that contains protocol messages or Bazel can connect to a server (called the Build Event Service) to stream events.

This is implemented in the `build.lib.buildeventservice` and `build.lib.buildeventstream` Java packages.

External repositories

Whereas Bazel was originally designed to be used in a monorepo (a single source tree containing everything one needs to build), Bazel lives in a world where this is not necessarily true. "External repositories" are an abstraction used to bridge these two worlds: they represent code that is necessary for the build but is not in the main source tree.

The WORKSPACE file

The set of external repositories is determined by parsing the WORKSPACE file. For example, a declaration like this:

```
local_repository(name="foo", path="/foo/bar")
```

Results in the repository called `@foo` being available. Where this gets complicated is that one can define new repository rules in Starlark files, which can then be used to load new Starlark code, which can be used to define new repository rules and so on...

To handle this case, the parsing of the WORKSPACE file (in `WorkspaceFileFunction`) is split up into chunks delineated by `load()` statements. The chunk index is indicated by

`WorkspaceFileKey.getIndex()` and computing `WorkspaceFileFunction` until index X means evaluating it until the Xth `load()` statement.

Fetching repositories

Before the code of the repository is available to Bazel, it needs to be *fetched*. This results in Bazel creating a directory under `$OUTPUT_BASE/external/<repository name>`.

Fetching the repository happens in the following steps:

1. `PackageLookupFunction` realizes that it needs a repository and creates a `RepositoryName` as a `SkyKey`, which invokes `RepositoryLoaderFunction`
2. `RepositoryLoaderFunction` forwards the request to `RepositoryDelegatorFunction` for unclear reasons (the code says it's to avoid re-downloading things in case of Skyframe restarts, but it's not a very solid reasoning)
3. `RepositoryDelegatorFunction` finds out the repository rule it's asked to fetch by iterating over the chunks of the WORKSPACE file until the requested repository is found
4. The appropriate `RepositoryFunction` is found that implements the repository fetching; it's either the Starlark implementation of the repository or a hard-coded map for repositories that are implemented in Java.

There are various layers of caching since fetching a repository can be very expensive:

1. There is a cache for downloaded files that is keyed by their checksum (`RepositoryCache`). This requires the checksum to be available in the WORKSPACE file, but that's good for hermeticity anyway. This is shared by every Bazel server instance on the same workstation, regardless of which workspace or output base they are running in.
2. A "marker file" is written for each repository under `$OUTPUT_BASE/external` that contains a checksum of the rule that was used to fetch it. If the Bazel server restarts but the checksum does not change, it's not re-fetched. This is implemented in `RepositoryDelegatorFunction.DigestWriter` .
3. The `--distdir` command line option designates another cache that is used to look up artifacts to be downloaded. This is useful in enterprise settings where Bazel should not fetch random things from the Internet. This is implemented by `DownloadManager` .

Once a repository is downloaded, the artifacts in it are treated as source artifacts. This poses a problem because Bazel usually checks for up-to-dateness of source artifacts by calling stat() on them, and these artifacts are also invalidated when the definition of the repository they are in changes. Thus, `FileStateValue`s for an artifact in an external repository need to depend on their external repository. This is handled by `ExternalFilesHelper`.

Managed directories

Sometimes, external repositories need to modify files under the workspace root (e.g. a package manager that houses the downloaded packages in a subdirectory of the source tree). This is at odds with the assumption Bazel makes that source files are only modified by the user and not by itself and allows packages to refer to every directory under the workspace root. In order to make this kind of external repository work, Bazel does two things:

1. Allows the user to specify subdirectories of the workspace Bazel is not allowed to reach into. They are listed in a file called `.bazelignore` and the functionality is implemented in `BlacklistedPackagePrefixesFunction`.
2. We encode the mapping from the subdirectory of the workspace to the external repository it is handled by into `ManagedDirectoriesKnowledge` and handle `FileStateValue`s referring to them in the same way as those for regular external repositories.

Repository mappings

It can happen that multiple repositories want to depend on the same repository, but in different versions (this is an instance of the "diamond dependency problem"). For example, if two binaries in separate repositories in the build want to depend on Guava, they will presumably both refer to Guava with labels starting `@guava//` and expect that to mean different versions of it.

Therefore, Bazel allows one to re-map external repository labels so that the string `@guava//` can refer to one Guava repository (e.g. `@guava1//`) in the repository of one binary and another Guava repository (e.g. `@guava2//`) the the repository of the other.

Alternatively, this can also be used to **join** diamonds. If a repository depends on `@guava1//`, and another depends on `@guava2//`, repository mapping allows one to re-map both repositories to use a canonical `@guava//` repository.

The mapping is specified in the WORKSPACE file as the `repo_mapping` attribute of individual repository definitions. It then appears in Skyframe as a member of `WorkspaceFileValue`, where it is plumbed to:

- `Package.Builder.repositoryMapping` which is used to transform label-valued attributes of rules in the package by `RuleClass.populateRuleAttributeValues()`
- `Package.repositoryMapping` which is used in the analysis phase (for resolving things like `$(location)` which are not parsed in the loading phase)
- `SkylarkImportLookupFunction` for resolving labels in load() statements

JNI bits

The server of Bazel is *mostly* written in Java. The exception is the parts that Java cannot do by itself or couldn't do by itself when we implemented it. This is mostly limited to interaction with the file system, process control and various other low-level things.

The C++ code lives under src/main/native and the Java classes with native methods are:

- `NativePosixFiles` and `NativePosixFileSystem`
- `ProcessUtils`
- `WindowsFileOperations` and `WindowsFileProcesses`
- `com.google.devtools.build.lib.platform`

Console output

Emitting console output seems like a simple thing, but the confluence of running multiple processes (sometimes remotely), fine-grained caching, the desire to have a nice and colorful terminal output and having a long-running server makes it non-trivial.

Right after the RPC call comes in from the client, two `RpcOutputStream` instances are created (for stdout and stderr) that forward the data printed into them to the client. These are then wrapped in an `OutErr` (an (stdout, stderr) pair). Anything that needs to be printed on the console goes through these streams. Then these streams are handed over to `BlazeCommandDispatcher.execExclusively()`.

Output is by default printed with ANSI escape sequences. When these are not desired (`--color=no`), they are stripped by an `AnsiStrippingOutputStream`. In addition, `System.out` and `System.err` are redirected to these output streams. This is so that debugging information can be printed using `System.err.println()` and still end up in the terminal output of the client (which is different from that of the server). Care is taken that if a process produces binary output (e.g. `bazel query --output=proto`), no munging of stdout takes place.

Short messages (errors, warnings and the like) are expressed through the `EventHandler` interface. Notably, these are different from what one posts to the `EventBus` (this is confusing). Each `Event` has an `EventKind` (error, warning, info, and a few others) and they may have a `Location` (the place in the source code that caused the event to happen).

Some `EventHandler` implementations store the events they received. This is used to replay information to the UI caused by various kinds of cached processing, for example, the warnings emitted by a cached configured target.

Some `EventHandler`s also allow posting events that eventually find their way to the event bus (regular `Event`s do *not* appear there). These are implementations of `ExtendedEventHandler` and their main use is to replay cached `EventBus` events. These `EventBus` events all implement `Postable`, but not everything that is posted to `EventBus` necessarily implements this interface; only those that are cached by an `ExtendedEventHandler` (it would be nice and most of the things do; it's not enforced, though)

Terminal output is *mostly* emitted through `UiEventHandler`, which is responsible for all the fancy output formatting and progress reporting Bazel does. It has two inputs:

- The event bus
- The event stream piped into it through Reporter

The only direct connection the command execution machinery (i.e. the rest of Bazel) has to the RPC stream to the client is through `Reporter.getOutErr()`, which allows direct access to these streams. It's only used when a command needs to dump large amounts of possible binary data (e.g. `bazel query`).

Profiling Bazel

Bazel is fast. Bazel is also slow, because builds tend to grow until just the edge of what's bearable. For this reason, Bazel includes a profiler which can be used to profile builds and Bazel itself. It's implemented in a class that's aptly named `Profiler`. It's turned on by default, although it records only abridged data so that its overhead is tolerable; The command line `--record_full_profiler_data` makes it record everything it can.

It emits a profile in the Chrome profiler format; it's best viewed in Chrome. It's data model is that of task stacks: one can start tasks and end tasks and they are supposed to be neatly nested within each other. Each Java thread gets its own task stack. **TODO:** How does this work with actions and continuation-passing style?

The profiler is started and stopped in `BlazeRuntime.initProfiler()` and `BlazeRuntime.afterCommand()` respectively and attempts to be live for as long as possible so that we can profile everything. To add something to the profile, call `Profiler.instance().profile()`. It returns a `Closeable`, whose closure represents the end of the task. It's best used with try-with-resources statements.

We also do rudimentary memory profiling in `MemoryProfiler`. It's also always on and it mostly records maximum heap sizes and GC behavior.

Testing Bazel

Bazel has two main kinds of tests: ones that observe Bazel as a "black box" and ones that only run the analysis phase. We call the former "integration tests" and the latter "unit tests", although they are more like integration tests that are, well, less integrated. We also have some actual unit tests, where they are necessary.

Of integration tests, we have two kinds:

1. Ones implemented using a very elaborate bash test framework under `src/test/shell`
2. Ones implemented in Java. These are implemented as subclasses of `AbstractBlackBoxTest`.

`AbstractBlackBoxTest` has the virtue that it works on Windows, too, but most of our integration tests are written in bash.

Analysis tests are implemented as subclasses of `BuildViewTestCase`. There is a scratch file system you can use to write BUILD files, then various helper methods can request configured targets, change the configuration and assert various things about the result of the analysis.