Driver Design & Internals

Introduction

This document serves to describe the high-level design of the Swift 2.0 compiler driver (which includes what the driver is intended to do, and the approach it takes to do that), as well as the internals of the driver (which is meant to provide a brief overview of and rationale for how the high-level design is implemented).

The Swift driver is not intended to be GCC/Clang compatible, as it does not need to serve as a drop-in replacement for either driver. However, the design of the driver is inspired by Clang's design.

Driver Stages

The compiler driver for Swift roughly follows the same design as Clang's compiler driver:

1. Parse: Command-line arguments are parsed into `Arg`s. A ToolChain is selected based on the current platform.
2. Pipeline: Based on the arguments and inputs, a tree of `Action`s is generated. These are the high-level processing steps that need to occur, such as "compile this file" or "link the output of all compilation actions".
3. Bind: The ToolChain converts the `Action`s into a set of `Job`s. These are individual commands that need to be run, such as "ld main.o -o main". Jobs have dependencies, but are not organized into a tree structure.
4. Execute: The `Job`s are run in a `Compilation`, which spawns off sub-processes for each job that needs execution. The `Compilation` is responsible for deciding which `Job`s actually need to run, based on dependency information provided by the output of each sub-process. The low-level management of sub-processes is handled by a `TaskQueue`.

Parse: Option parsing

The command line arguments are parsed as options and inputs into Arg instances. Some miscellaneous validation and normalization is performed. Most of the implementation is provided by LLVM.

An important part of this step is selecting a ToolChain. This is the Swift driver's view of the current platform's set of compiler tools, and determines how it will attempt to accomplish tasks. More on this below.

One of the optional steps here is building an *output file map*. This allows a build system (such as Xcode) to control the location of intermediate output files. The output file map uses a simple JSON format mapping inputs to a map of output paths, keyed by file type. Entries under an input of "" refer to the top-level driver process.

> Certain capabilities, like incremental builds or compilation without linking, currently require an output file map. This should not be necessary.

Pipeline: Converting Args into Actions

At this stage, the driver will take the input Args and input files and establish a graph of Actions. This details the high-level tasks that need to be performed. The graph (a DAG) tracks dependencies between actions, but also manages ownership.

> Actions currently map one-to-one to sub-process invocations. This means that there are actions for things that should be implementation details, like generating dSYM output.

Build: Translating Actions into Jobs using a ToolChain

Once we have a graph of high-level Actions, we need to translate that into actual tasks to execute. This starts by determining the output that each Action needs to produce based on its inputs. Then we ask the ToolChain how to perform that Action on the current platform. The ToolChain produces a Job, which wraps up both the output information and the actual invocation. It also remembers which Action it came from and any Jobs it depends on. Unlike the Action graph, Jobs are owned by a single Compilation object and stored in a flat list.

When a Job represents a compile of a single file, it may also be used for dependency analysis, to determine whether it is safe to not recompile that file in the current build. This is covered by checking if the input has been modified since the last build; if it hasn't, we only need to recompile if something it depends on has changed.

Schedule: Ordering and skipping jobs by dependency analysis

A Compilation's goal is to make sure every Job in its list of Jobs is handled. If a Job needs to be run, the Compilation attempts to *schedule* it. If the Job's dependencies have all been completed (or determined to be skippable), it is scheduled for execution; otherwise it is marked as *blocked*.

To support Jobs compiling individual Swift files, which may or may not need to be run, the Compilation keeps track of a DependencyGraph. (If file A depends on file B and file B has changed, file A needs to be recompiled.) When a Job completes successfully, the Compilation will both re-attempt to schedule Jobs that were directly blocked on it, and check to see if any other Jobs now need to run based on the DependencyGraph. See the section on DependencyAnalysis for more information.

Batch: Optionally combine similar jobs

The Driver has an experimental "batch mode" that examines the set of scheduled jobs just prior to execution, looking for jobs that are identical to one another aside from the primary input file they are compiling in a module. If it finds such a set, it may replace the set with a single BatchJob, before handing it off to the TaskQueue; this helps minimize the overall number of frontend processes that run (and thus do potentially redundant work).

Once any batching has taken place, the set of scheduled jobs (batched or otherwise) is transferred to the TaskQueue for execution.

Execute: Running the Jobs in a Compilation using a TaskQueue

The Compilation's TaskQueue controls the low-level aspects of managing subprocesses. Multiple Jobs may execute simultaneously, but communication with the parent process (the driver) is handled on a single thread. The level of parallelism may be controlled by a compiler flag.

If a Job does not finish successfully, the Compilation needs to record which jobs have failed, so that they get rebuilt next time the user tries to build the project.