llbuild Documentation

Release 0.1

Apple, Inc.

2015-12-04

Contents

CHAPTER 1

Development

This document contains information on developing llbuild.

The project is set up in the following fashion, generally following the LLVM and Swift conventions.

- For C++ code:
  - The code is written against the C++14 standard.
  - The style should follow the LLVM conventions, but variable names use camelCase.
  - Both exceptions and RTTI are **disallowed**.
- The project is divided into conceptually distinct layers, which are organized into distinct "libraries" under *lib/*. The current set of libraries, and their dependencies, is:

  llvm

  Shared LLVM support facilities, for llbuild use. These are intended to be relatively unmodified versions of data structures which are available in LLVM, but are just not factored in a way that we can use them. The goal is to eventually factor out a common LLVM-support infrastructure that can be shared.

  Basic

  Support facilities available to all libraries.

  Core

  The core build engine implementation. Depends on **Basic**.

  BuildSystem

  The "llbuild"-native build system library. Depends on **Basic**, **Core**.

  Ninja

  Programmatic APIs for dealing with Ninja build manifests. Depends on **Basic**.

  Commands

  Implementations of command line tool frontends. Depends on **BuildSystem**, **Core**, **Ninja**.

  Code in libraries in the lower layers is **forbidden** from using code in the higher layers.
- Public facing products (tools and libraries) are organized under *products/*. Currently the products are:

  llbuild

  The implementation of the command line *llbuild* tool, which is used for command line testing.

  libllbuild

A C API for llbuild.

swift-build-tool

The command line build tool used by the Swift package manager.

- Examples of using *llbuild* are available under *examples/*.
- There are two kinds of correctness tests include in the project:

  LLVM-Style Functional Tests

  These tests are located under *tests/* and then layed out according to library and the area of functionality. The tests themselves are written in the LLVM "ShTest" style and run using the *Lit* testing tool, for more information see LLVM's [Testing Guide](http://llvm.org/docs/TestingGuide.html#writing-new-regression-tests).

  C++ Unit Tests

  These tests are located under *unittests/* and then layed out according to library. The tests are written using the [Googletest](https://code.google.com/p/googletest/) framework.

  All of these tests are run by default (by *lit*) during the build.

- There are also additional performance tests:

  Xcode Performance Tests

  These tests are located under *perftests/Xcode*. They use the Xcode XCTest based testing infrastructure to run performance tests. These tests are currently only supported when using Xcode.

- Header includes are placed in the directory structure according to their purpose:

  include/llbuild/<LIBRARY_NAME>/

  Contains the *internal* (in Swift terminology) APIs available for use by any other code in the *llbuild* project (subject to layering constraints).

  All references to these includes should follow the form:

```
#include "llbuild/<LIBRARY_NAME>/<HEADER_NAME>.h"
```

  lib/llbuild/<LIBRARY_NAME>

  Contains the *internal* (in Swift terminology) APIs only available for use by code in the same library.

  All references to these includes should follow the form:

```
#include "<HEADER_NAME>.h"
```

  The Xcode project disables the use of headermaps, to aid in following these conventions.

CHAPTER 2

Build Engine

This document currently only provides additional information on some aspects of the build engine design. See the source code documentation comments for information on the actual `BuildEngine` APIs.

2.1 Order-only Dependencies

The build engine supports "order-only" dependencies (in the style of Ninja) through a "must-follow" input request or barrier. Internally, this is treated much the same as a normal input request, in that the task is responsible for calling `taskMustFollow()` on the engine during its preparation phase.

However, these input requests do not need to be scanned or persisted. This is because the dependency is only relevant when the dependee is executing, and in those cases the task will be responsible for dynamically supplying the barrier. Thus, the implementation can piggy-back on the existing mechanisms for ensuring input requests are available prior to executing the task, and simply discard the order-only input request once satisfied.

2.2 Parallelism

There are two obvious approaches to introducing parallelism into the core build engine.

1. The first approach (which is taken currently) changes the task definition so that it gets a callback `inputsAvailable()`, and adds a new task API on the `BuildEngine` called `taskIsComplete()` which is allowed to be called concurrently.

   When the task gets all of its inputs, then it is responsible for dispatching its work in a parallel fashion and informing the engine when that is complete. The engine tracks the task as pending completion until it received that callback.

   This keeps the engine simple, as it doesn't really deal with concurrency at all except for in specific cases that should be easy to reason about. It also makes the engine separable from the specific concurrency model, which may make it easier to plug in schedulers and the like (for example, it is somewhat clear how a posix_spawn + ppoll/pselect concurrency model works in this system).

2. The alternate approach is to change the engine so that it manages all of the concurrency, changing the Task contract such that it stays synchronous but can be called on any thread.

   This means that all clients of the engine get concurrency automatically, but also more intimately ties the engine implementation to the concurrency model.

2.3 Dependency Scanning

The engine is designed to scan dependencies concurrently with other build work. This introduces a fair amount of complexity into the engine, but is important because we want to ensure that the engine is able to dispatch any identified tasks as soon they are found.

2.3.1 Recursive Dependency Scanning

The current design of the engine handles dependency scanning in the same manner as other tasks, it associates a scan request with a rule and then enqueues the actual scanning work so that it can proceed in parallel with other work.

Currently, the dependency scanning processes each input for an individual rule in sequence, and it suspends itself when it is waiting on an individual input to complete. This is made more complex by the need to sometimes execute a task before being able to determine that its downstream clients need to rerun (when the result has not changed). To manage that, the engine tracks for each task which other tasks are waiting for it to complete as well as which tasks are waiting for it to be scanned.

The current implementation is not yet capable of scanning multiple inputs for a particular task concurrently, because when doing it is necessary to track additional state about whether or not the input was *actually* requested by a task or just a dependency present on a previous iteration. For example, consider the case where a task depended upon **A** and **B** in the last build. If a concurrent scan indicates the task should rerun because **B** has changed, it is not necessarily true that **B** itself should rerun (because the task may no longer request **B** as an input). The current implementation side-steps this problem by scanning dependencies one at a time.

CHAPTER 3

Build System

This document describes the *llbuild*-native `BuildSystem` component. While *llbuild* contains a Core engine which can be used to build a variety of build systems (and other incremental, persistent, and parallel computational systems), it also contains a native build system implementation on which higher level facilities are built.

3.1 Design Goals

As with the core engine, the `BuildSystem` is designed to serve as a building block for constructing higher level build systems.

The goal of the `BuildSystem` component is that it should impose enough policy and requirements so that the process of executing a typical software build is *primarily* managed by code within the system itself, and that the system has enough information to be able to perform sophisticated tasks like automatic distributed compilation or making use of a in-process compiler design.

3.2 Build Graph

Conceptually, the `BuildSystem` is organized around a bipartite graph of commands and nodes. A `Command` represent a units of computation that need to be done as part of the build, and a `Node` represent a concrete value used as an input or output of those commands. Each `Command` declares which `Tool` is to be used to create it. The tools themselves are integrated into the `BuildSystem` component (either as builtin tools, or via the client), and this allows them to provided customizable features for use in the build file, and to be deeply integrated with *llbuild*.

3.2.1 Nodes

Each node represents an individual value that can be produced as part of the build. In a file based context, each node would conceptually represent a file on the file system, but there is no built-in requirement that a node be synonymous with a file (that said, it is a common case and the `BuildSystem` will have special support to ensure that using nodes which are proxies for files on disk is convenient and featureful).

Currently, the build system automatically treats nodes as files unless they have a name matching `'<.*>'`, see the documentation of the `is-virtual` node attribute for more information.

3.2.2 Commands

Each `Command` is used to represent the things that do actual work in the build graph, and they take input nodes and transform them into output ones. A command could be implemented as something which invokes an external command (like *cc*) to do work, or it could be something which is implemented internally to the `BuildSystem` or the client (for example, a command to compute the SHA-1 hash of a file).

A `Command` as modeled in the `BuildSystem` component is related to, but different, from the tasks that are present at the lower `BuildEngine` layer. A `Command` in the `BuildSystem` is roughly equivalent to a `BuildEngine::Rule` and the `BuildEngine::Task` which computes that rule, but in practice a `Command` in the `BuildSystem` might be implemented using any number of coordinating rules and tasks from the lower level. For example, a compiler `Command` in the `BuildSystem` might end up with a task to invoke the compiler command, a task to parse the diagnostics output, and another task to parse the header dependencies output.

Every `Command` has a unique identifier which can be used to refer to the command, and which is expected to be stable even as the build graph is updated by the generating tool.

3.2.3 Tools

A `Tool` defines how a specific command is executed, and are the general mechanism through which the `BuildSystem` can support various styles of work (as opposed to just running commands) and extension by clients.

Every `Command` has an associated tool which defines how it will be run, and can provide additional tool-specific properties to control its execution. For example, a `Command` which invokes a generic tool that runs external commands would typically provide the list of command line arguments to use. On the other hand, a `Command` which uses a higher-level tool to invoke the compiler may set additional properties requesting that automatic header dependencies be used.

3.3 Build File

The build file is the base input to the native build system, similar to a Makefile or a Ninja manifest. It contains a description of the things that can be built, the commands that need to be executed to build them, and the connections between those commands. Similar to Ninja, the basic build file language is not intended to be written directly, but is expected to be an artifact produced by the higher level build system.

The build file syntax is currently YAML, to faciliate ease of implementation and evolution. At some future point, we may wish to change to a custom file format to optimize for the native build system's specific requirements (in particular, to reduce the file size).

A small example build file is below:

```
# Declare the client information.
client:
  name: example-client
  version: 1

# Define the tools.
tools:
  cc:
    enable-dependencies: True
    cwd: /tmp/example
  link:
    cwd: /tmp/example

# Define the targets.
```

```
targets:
  hello: ["hello"]

# Define properties on nodes.
nodes:
  hello.o:
    hash-content: True

# Define the commands.
commands:
  link-hello:
    tool: link
    inputs: ["hello.o"]
    outputs: ["hello"]
  cc-hello.o:
    tool: cc
    input: ["hello.c"]
    outputs: ["hello.o"]
    args: -O0
```

The build file is logically organized into five different sections (grouped by keys in a YAML mapping). These sections *MUST* appear in the following order if present.

- Client Definition (*client* key)

  Since the BuildFile format is intended to be reused by all clients of the `BuildSystem` component, the client section is used to provide information to identify exactly which client should be used to build this build file. The section gives the name of the client, and an additional version that can be used by the client to version semantic changes in the client hooks.

  The name field is required, and must be non-empty.

  The version field is optional, and defaults to 0.

  Additional string keys and values may be specified here, and are passed to the client to handle.

- `Tool` Definitions (*tools* key)

  This section is used to configure common properties on any of the tools used by the build file. Exactly what properties are available depends on the tool being used.

  Each property is expected to be a string key and a string value.

- Target Definitions (*targets* key)

  This section defines top-level targets which can be used to group commands which should be build together for a particular purpose. This typically would include definitions for all of the things a user might want to build directly.

  The default target to build can be specified by including an entry for the empty string (“”).

- `Node` Definitions (*nodes* key)

  This section can be used to configure additional properties on the node objects. `Node` objects are automatically created whenever they appear as an input or output, and the properties of the object will be inferred from the context (i.e., by the command that produces or consumes them). However, this section allows customizing those properties or adding additional ones.

  Each key must be a scalar string naming identifying the node, and the value should be a map containing properties for the node.

  Each property is expected to be a string key and a string value.

Note: FIXME: We may want to add a mechanism for defining default properties.

Note: FIXME: We may want to add the notion of types to nodes (for example, file versus string).

- `Command` Definitions (*commands* key)

  This section defines all of the commands as a YAML mapping, where each key is the name of the command and the value is the command definition. The only required field is the *tool* key to specify which tool produces the command.

  The *tool* key must always be the leading key in the mapping.

  The *description* key is available to all tools, and should be a string describing the command.

  The *inputs* and *outputs* keys are shared by all tools (although not all tools may use them) and are lists naming the input and output nodes of the `Command`. It is legal to use undeclared nodes in a command definition – they will be automatically created.

  All other keys are `Tool` specific. Most tool specific properties can also be declared in the tool definitions section to set a default for all commands in the file, although this is at the discretion of the individual tool.

  Note: FIXME: We may want some provision for providing inline node attributes with the command definitions. Otherwise we cannot really stream the file to the build system in cases where node attributes are required.

3.3.1 Format Details

The embedding of the build file format in YAML makes use of the built in YAML types for most structures, and should be self explanatory for the most part. There are two important details that are worth calling out:

1. In order to support easy specification of command lines, some tools may allow specifying command line arguments as a single string instead of a YAML list of arguments. In such cases, the string will be quoted following basic shell syntax.

Note: FIXME: Define the exact supporting shell quoting rules.

2. The build file specification is designed to be able to make use of a streaming YAML parser, to be able to begin building before the entire file has been read. To this end, it is recommended that the commands be laid out starting with the commands that define root nodes (nodes appearing in targets) and then proceeding in depth first order along their dependencies.

3.3.2 Dynamic Content

Note: FIXME: Add design for how dynamically generated work is embedded in the build file.

3.4 Node Attributes

As with commands, nodes can also have attributes which configured their behavior.

The following attributes are currently supported:

Name	Description
is-virtual	A boolean value, indicating whether or not the node is "virtual". By default, the build system assumes that nodes matching the pattern `'<.*>'` (e.g., `<link>`) are virtual, and all other nodes correspond to files in the file system matching the name. This attribute can be used to override that default.

Note: FIXME: At some point, we probably want to support custom node types.

3.5 Builtin Tools

The build system provides several built-in tool definitions which are available regardless of the client.

The following tools are currently built in:

Phony Tool Identifier: *phony*

A dummy tool, used for imposing ordering and grouping between input and output nodes.

No attributes are supported other than the common keys.

Shell Tool Identifier: *shell*

A tool used to invoke shell commands.

This tool supports an "args" key used to provide the shell command to be run (using `/bin/sh -c`).

Clang Tool Identifier: *clang*

A tool used to invoke the Clang compiler.

This tool supports an "args" key used to provide the shell command to be run (using `/bin/sh -c`), as well as a "deps" key which specified the path to a Makefile fragment (presumed to be output by the compiler) specifying additional discovered dependencies for the output.

CHAPTER 4

TODO

4.1 Generic

- It would be nice to have some basic infrastructure for handling command line parsing better.
- It would be nice to have some infrastructure for statistics (a la LLVM).

4.2 Ninja Manifests

- Diagnose multiple build decls with same output.

4.3 Build Engine

- Generalize key type.
- Generalize value type (cleanly, current solution is gross).
- Support multiple outputs.
- Support active scheduling (?).
- Consider moving the task model to one where the build engine just invokes callbacks on the delegate (probably using task IDs, and maybe kind IDs for the use of clients which want to follow class-based dispatch models). This has two advantages, (1) it maps more neatly to an obvious C API, and (2) it should make it easier to cleanly generalize the value type because only the engine and delegate need to be templated, and neither support subclassing.
- Figure out when taskNeedsInput() should be allowed, and if taskDiscoveredDependency() should be eliminated in favor of loosened rules about when it can be invoked.
- Figure out if Rule should be subclassed, with virtual methods for action generation and input checking.
- Implement proper error handling.
- Think about how to implement dynamic command failure – how should this be communicated downstream? This is another area where flexibility might be nice.
- Investigate how Ninja handles syncing the build graph with the on disk state when it has no DB (nevermind, seems to rebuild). What happens when an output of a compile command is touched?
- Introspection features for watching build status.

- Performance
  - Think about # of allocations in engine loop.
  - Think about whether there is any way to get of per-task variable length list.
  - Think about making-progress policy, what order do we prefer starting tasks vs. providing inputs vs. finishing tasks.
  - Should we finalize immediately instead of moving things around on queues.
  - We need to avoid the duplicate stating we currently do of input files.
  - Implement an efficient shared queue for FinishedTaskInfos, if we stay with the current design.
  - Figure out if we should change the Rule interface so that the engine can manage a free-list of allocated Tasks.

4.4 Build Database

- Performance
  - Should we support DB stashing extra information in Rule (its own ID number). Should the engine officially maintain the ID (maybe useful once we generalize anyway).
  - Should we support the DB doing a bulk load of the rule data? Probably more efficient for most databases, but on the other hand this would go against moving to a model where the data is *only* stored in the DB layer, and the BuildEngine always has to go to the DB for it. That model might be slower, but more scalable.
  - Investigate using DB entirely on demand, per above comment. Very scalable.
  - Normalize key and dependency node types to a unique entry to reduce duplication of large key values.
  - Many clients end up having additional information about a key that then gets lost when they serialize it and get it back somewhere else (e.g., one task requests the key, another starts the key). For example, the client most likely has some internal state associated with that key that would be really nice to be able to pass around.

    We can solve this by making the key type richer, and allowing it to have serialization as just one of its methods. We could use a virtual interface or just a simple mechanism to attach a payload.

4.5 Ninja Specific

4.5.1 Tasks for Usable Tool

- Implement path normalization (for Nodes as well as things like imported dependencies from compiler output).
- Implement Ninja failure semantics for order-only dependencies, which block the downstream command.
- Handle removed implicit dependencies properly, currently this generates an error and then builds ok on the next iteration. The latter problem may indicate a latent issue in handling of discovered dependencies.
- Handle rerunning a command on introduction of new dependencies. For example, before we reran on command changes we wouldn't rebuild a library just because it depended on a new file. We should probably make sure this happens even if the command doesn't change, although it might be good to check vs Ninja.
- Support update-if-newer for commands with discovered dependencies.
- We should probably store a rule type along with the result, and always invalidate if the rule type has changed.

4.5.2 Random Tasks

- Ninja reruns a command if the restat= flag changes? What triggers this? This behavior probably also happens for depfiles and things, we should match.
- There are some subtle differences in how we handle restat = 0, and generally we don't do the same thing Ninja would (we can rebuild less). This is because Ninja will just run things downstream if an incoming edge was dirty, but we will do so and also allow the update-if-newer behavior to trigger on interior commands. As an example, look at how the multiple-outputs test case behaves in Ninja with restat = 0.
- Tasks should have a way to examine their discovered dependencies from a previous run. For example, this is necessary to implement update-if-newer for commands with discovered dependencies.
- Add support for cleaning up output files (deleting them on failed tasks)?
- Investigate using pselect mechanisms vs blocked threads.
- Support traditional style handling of updated outputs (in which consumer commands are rerun but not the producer itself).
- Performance
  - Need to optimize the evalString() stuff. A good performance test for this is to compare `ninja -n` on the chromium fake build.ninja file vs `llb ninja build --quiet --no-execute` (62% of which is currently the loading, and 43% of which is evalString()).
  - It would be super cool to use the build engine itself for managing the loading of the .ninja file, if we cached the result in a compact binary representation (the build engine would then be used to recompute that as necessary). This would be a nice validation of the generic engine approach too.
  - There is some bad-smelling redundancy between how we check the file info in the **IsValid()* functions, and how we then recompute that info later as part of the task (and the engine internally will compare it to see if it has changed to know if it needs to propagate the change). We need to think about this and figure out what is ideal. There might be a cleaner modeling where we discretely represent each stat-of-file as an input that is then consumed by each item that requires it. This would make it easy to guarantee we compute such things once.
  - I have heard a claim that one can actually improve performance by strategically purging the OS buffer cache – the claim was that it is faster to build Swift after building LLVM & Clang if there is a purge in between. If true, this may be better things we can do to communicate to the kernel the purpose and lifetime of things like object files.
  - We should consider allowing the write of the target result to go directly into the stored Result field. That would avoid the need for spurious allocations when updating results.
  - We need to switch the Rule Dependencies to be stored using the ID of the rule (which means we need to assign rule IDs, but the DB would like that anyway). This dramatically reduces the storage required by the database (although a lot of that is because of our subpar phony command implementation, and it would drop significantly if we switch to a specialized implementation for phony commands, because we don't need the clunky giant composite-key).
  - We should use a custom task for Phony commands, they have a lot of special cases (like the one above about the composite key size).
  - We should move to a compact encoding for the build value. Not worth doing until we address the rule_dependencies table size.

4.6 Build System

4.6.1 Build File

- We will probably want some way to define properties shared by groups of tasks (for example, common flags), for efficiencies sake. There are a couple ways to do this:
  - We could make the build file an "immediate-mode" sort of interface, and allow interleaving of tool and task maps. Then the client could just generate the file with updated information interleaved. This would be similar to how Ninja files get generated in practice by nice generators (*gyp*, not *CMake*).
  - We could allow the definition of tool aliases, that can define additional properties. This lets the format be better definite and not have immediate mode stateful problems.
- We want some way to allow the task name and one of the tasks outputs to be the same, without having a redundant specification.
- We might need a mechanism for defining default properties for nodes.
- We may want to add a notion of types for nodes. We could try and be context dependent too, but having a type here would make it easier for the client to bind the node to the right type during loading.
- We may want some provision for providing inline node attributes with the task definitions. Otherwise we cannot really stream the file to the build system in cases where node attributes are required.

CHAPTER 5

Indices and tables

- genindex
- search