Development Tips

Compile times

Build using Ninja

To *be* a productivity ninja, one must *use* `ninja`. `ninja` can be invoked inside the swift build directory, e.g. `<path>/build/Ninja-ReleaseAssert/swift-macosx-x86_64/`. Running `ninja` (which is equivalent to `ninja all`) will build the local swift, stdlib and overlays. It doesn't necessarily build all the testing infrastructure, benchmarks, etc.

`ninja -t targets` gives a list of all possible targets to pass to ninja. This is useful for grepping.

For this example, we will figure out how to quickly iterate on a change to the standard library to fix 32-bit build errors while building on a 64-bit host, suppressing warnings along the way.

`ninja -t targets | grep stdlib | grep i386` will output many targets, but at the bottom we see `swift-stdlib-iphonesimulator-i386`, which looks like a good first step. This target will just build i386 parts and not waste our time also building the 64-bit stdlib, overlays, etc.

Going further, ninja can spawn a web browser for you to navigate dependencies and rules. `ninja -t browse swift-stdlib-iphonesimulator-i386` will open a webpage with hyperlinks for all related targets. "target is built using" lists all this target's dependencies, while "dependent edges build" list all the targets that depend directly on this.

Clicking around a little bit, we can find `lib/swift/iphonesimulator/i386/libswiftCore.dylib` as a commonly-depended-upon target. This will perform just what is needed to compile the standard library for i386 and nothing else.

Going further, for various reasons the standard library has lots of warnings. This is actively being addressed, but fixing all of them may require language features, etc. In the mean time, let's suppress warnings in our build so that we just see the errors. `ninja -nv lib/swift/iphonesimulator/i386/libswiftCore.dylib` will show us the actual commands ninja will issue to build the i386 stdlib. (You'll notice that an incremental build here is merely 3 commands as opposed to ~150 for `swift-stdlib-iphonesimulator-i386`).

Copy the invocation that has `-o <build-path>/swift-macosx-x86_64/stdlib/public/core/iphonesimulator/i386/Swift.o`, so that we can perform the actual call to swiftc ourselves. Tack on `-suppress-warnings` at the end, and now we have the command to just build `Swift.o` for i386 while only displaying the actual errors.

Use sccache to cache build artifacts

Compilation times for the compiler and the standard library can be agonizing, especially for cold builds. This is particularly painful if

- You're bisecting an issue over a large period of time.
- You're switching between Xcode versions to debug issues.
- You have multiple distinct work directories, say for working on multiple things at once.

`sccache` provides a mostly automatic caching experience for C and C++ build artifacts. Here is how you can set it up and use it on macOS:

```
$ brew install sccache
$ ./swift/utils/build-script MY_ARGS --sccache
```

If you want to always use sccache, you can `export SWIFT_USE_SCCACHE=1` and omit the `--sccache` flag from the `build-script` invocation.

Given the size of artifacts generated, you might also want to bump the cache size from the default 10GB to something larger, say by putting `export SCCACHE_CACHE_SIZE="50G"` in your dotfile(s). You'll need to restart the `sccache` server after changing that environment variable (`sccache --stop-server && sccache --start-server`).

You can run some compiles to see if it is actually doing something by running `sccache --show-stats` . Depending on the exact compilation task you're running, you might see very different cache hit rates. For example, `sccache` is particularly effective if you're rebuilding LLVM, which doesn't change so frequently from the Swift compiler's perspective. On the other hand, if you're changing the compiler's AST, the cache hit rate is likely to be much lower.

One known issue with `sccache` is that you might occassionally get an "error: Connection to server timed out", even though you didn't stop the server. Simply re-running the build command usually works.

Memory usage

When building Swift, peak memory usage happens during the linking phases that produce llvm and swift executables. In case your build fails because a process ran out of RAM, you can use one or more of the following techniques to reduce the peak memory usage.

Reduce the amount of debug symbols

We can use debug symbols for only the part of the project we intend to work on. For example, when working on the compiler itself, we can build with debug symbols enabled only for the compiler:

```
build-script --release --debug-swift
```

Reduce the number of parallel link jobs

By default, `build-script` will spawn as many parallel compile / link jobs as there are CPUs in the machine. We can reduce the number of parallel link jobs by setting the `LLVM_PARALLEL_LINK_JOBS` and `SWIFT_PARALLEL_LINK_JOBS` CMake properties. We can set them through the `--llvm-cmake-options` and `--swift-cmake-options` arguments to `build-script` .

For example, to have `build-script` spawn only one link job at a time, we can invoke it as:

```
build-script --llvm-cmake-options==-DLLVM_PARALLEL_LINK_JOBS=1 --swift-cmake-options=-
DSWIFT_PARALLEL_LINK_JOBS=1
```