Luz

Release 1.0.0

Jaidan

Aug 25, 2023

CONTENTS:

Luz is a build system for Apple Darwin-based systems. It's name is derived from the Spanish word for "light." It's meant to be a lightweight, drop in replacement for other build systems such as Dragon and Theos.

CHAPTER

ONE

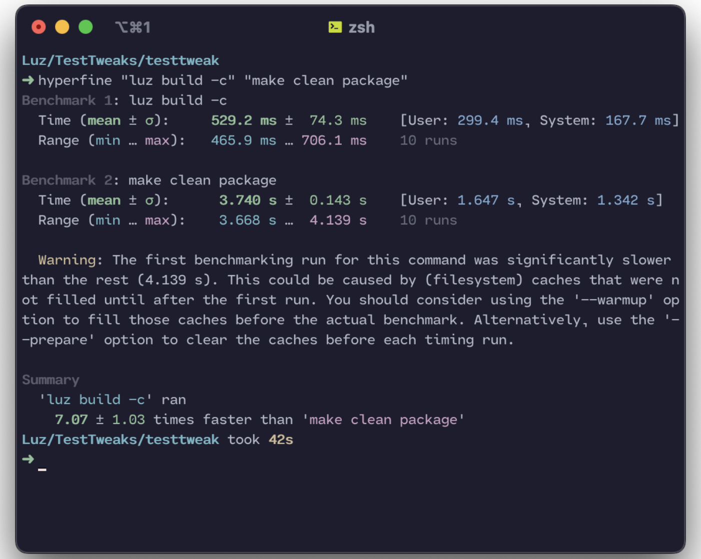
BENIFITS OVER “THEOS” AND “DRAGON”

Note: Luz is a work-in-progress project. Features will change, and bugs will be fixed. If you find a bug, please report it on the GitHub repository.

1.1 Speed

Luz is written in pure Python, and only uses libraries that I’ve created myself in its code. This means that it’s very fast. Luz especially shines when building projects with submodules, as it can build all of the submodules in parallel. This means that building a project only takes as long as the longest build time of any of the submodules.

Below you can find a benchmark of Luz vs. Theos, using the time comparison tool *hyperfine*. The same tweak was built (clean) with both build systems.



```
Luz/TestTweaks/testtweak
➜ hyperfine "luz build -c" "make clean package"
Benchmark 1: luz build -c
  Time (mean ± σ):      529.2 ms ±  74.3 ms    [User: 299.4 ms, System: 167.7 ms]
  Range (min … max):    465.9 ms … 706.1 ms    10 runs

Benchmark 2: make clean package
  Time (mean ± σ):       3.740 s ±  0.143 s    [User: 1.647 s, System: 1.342 s]
  Range (min … max):     3.668 s …  4.139 s    10 runs

  Warning: The first benchmarking run for this command was significantly slower
than the rest (4.139 s). This could be caused by (filesystem) caches that were n
ot filled until after the first run. You should consider using the '--warmup' op
tion to fill those caches before the actual benchmark. Alternatively, use the '-
-prepare' option to clear the caches before each timing run.

Summary
  'luz build -c' ran
    7.07 ± 1.03 times faster than 'make clean package'
Luz/TestTweaks/testtweak took 42s
➜ _
```

Note: This benchmark was ran on a 2020 MacBook Pro with an M1 processor, 8 GB of RAM, and a 256 GB SSD.

As you can see, Luz is much faster than Theos, and is able to build the same project in less than half the time.

1.2 Source Code Structure

Each of Luz's modules have a different source file, which are all subclassed from a main class called *Module*. This allows for easy extensibility, and allows for the creation of new modules without having to modify the core of the build system.

1.2.1 Setup

Installation

To install Luz, run the following command in your terminal:

```
$ python -c "$(curl -fsSL https://raw.githubusercontent.com/LuzProject/luz/main/install.
↪py)"
```

This will install Luz and all of its dependencies.

Options

You can call the install script with the following options:

Option	Type	Description
`-ns`, `--no-sdks`	Flag	Whether or not to install the SDKs. If this is set, you will need to install the SDKs manually.
`-u`, `--update`	Flag	Update Luz. (You can use `--ref` to specify a different ref to update to.)
`-r`, `--ref`	String	Ref of `luz` to install. This can be a branch, tag, or commit hash. Defaults to `main`.

Notes

- If you are on Windows, you will need to install the Windows Subsystem for Linux (WSL). You can find instructions on how to do this here.
- If you are on macOS, you will need to install Xcode and the Xcode Command Line Tools.

1.2.2 Commands

Luz is a command line tool. It is used to create, build, run, and test Luz projects.

`build`

Builds a project using the LuzBuild in the working directory.

Option	Type	Description
`-c` / `--clean`	Flag	Whether or not to clean the build directory before building.
`-p` / `--path`	Flag	Path to the directory to build. (i.e. `luz build -p /path/to/project`, defaults to the current working directory)
`-m` / `--meta`	Flag	Add meta information to the build. (i.e. `luz build -m release=true`)

`verify`

Verifies the structure of `luz.py`.

Option	Type	Description
`-p` / `--path`	Flag	Path to the directory to verify. (i.e. `luz verify -p /path/to/project`, defaults to the current working directory)

`gen`

Generate a project.

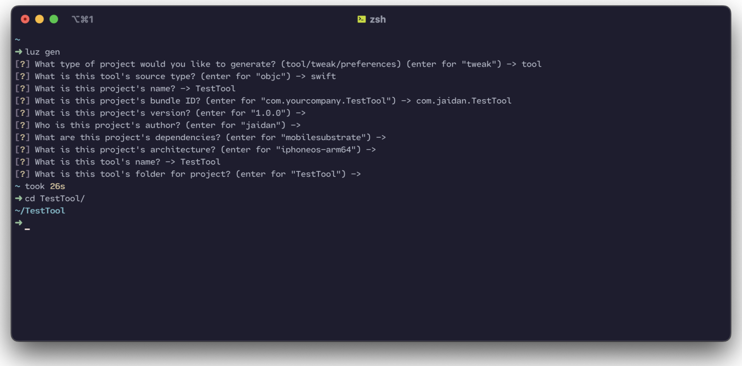
Option	Type	Description
`-t` / `--type`	String	The type of project to generate. (`tweak` if not specified)

1.2.3 Generation

Luz comes with a built-in project generator called `LuzGen`. It can be used to create a new project with the following command:

```
$ luz gen
```

This command will walk you through the steps to create a new project. First, it will ask you what kind of project you want to generate. Then, you can choose from different languages, such as Objective-C, Swift or Assembly. Finally, you enter project metadata, such as the name, author, version, etc. Below, you can find an example of how to use `LuzGen`.



```
~
➜ luz gen
[?] What type of project would you like to generate? (tool/tweak/preferences) (enter for "tweak") -> tool
[?] What is this tool's source type? (enter for "objc") -> swift
[?] What is this project's name? -> TestTool
[?] What is this project's bundle ID? (enter for "com.yourcompany.TestTool") -> com.jaidan.TestTool
[?] What is this project's version? (enter for "1.0.0") ->
[?] Who is this project's author? (enter for "jaidan") ->
[?] What are this project's dependencies? (enter for "mobilesubstrate") ->
[?] What is this project's architecture? (enter for "iphoneos-arm64") ->
[?] What is this tool's name? -> TestTool
[?] What is this tool's folder for project? (enter for "TestTool") ->
~ took 26s
➜ cd TestTool/
~/TestTool
➜ _
```

1.2.4 luzconf.py Formatting

Luz uses a Python file to define the settings for the build. Python is used so that compile-time variables can be specified, much like a Makefile. The file is called `luzconf.py` and is located in the root of your project.

`LuzGen` will automatically generate a `luzconf.py` file for any project that you create with it. It's not recommended to create your own `luzconf.py`, and you should only do so if you know what you're doing.

Meta

This is where you define the settings for the build, such as the SDK, the architectures to build for, and the `clang` path.

Meta variables are defined in a class called `Meta` that can be imported from `luz`.

Variable	Type	Description
`debug`	Boolean	Whether or not to build a debug version of the package. (`true` if not specified)
`release`	Boolean	Whether or not to build a release version of the package. (`false` if not specified)
`sdk`	String	SDK path to use for building. (uses `xcrun` to find an SDK if not specified)
`prefix`	String	Prefix to use for compilation commands. (`/` if not specified)
`cc`	String	Path to `clang` to use for compilation. (Finds `clang` in PATH if not specified)
`swift`	String	Path to `swift` to use for compilation. (Finds `swift` in PATH if not specified)
`rootless`	String	Whether or not to make a rootless DEB archive. (`true` if not specified)
`compression`	String	Command to use to compress the DEB archive. (`xz` if not specified)
`pack`	String	Whether or not to pack the DEB archive. (`true` if not specified)
`archs`	List	List of architectures to build for. (`['arm64', 'arm64e']` if not specified)
`platform`	String	Platform to build for. Can be `macosx`, `iphoneos` or `watchos`. (`iphoneos` if not specified)
`min_vers`	String	Minimum version to build for. (`15.0` if not specified)

Control

This is where you define the settings for the control file.

Control variables are defined in a class called `Control` that can be imported from `luz`.

Variable	Type	Description
`id`	String	ID of the package.
`name`	String	Name of the package.
`author`	String	Author of the package.
`maintainer`	String	Maintainer of the package.
`version`	String	Version of the package.
`section`	String	Section of the package.
`depends`	List	Dependencies of the package.
`architecture`	String	Architecture of the package.
`description`	String	Description of the package.

Additional control options can be found here.

Scripts

This is where maintainer scripts are defined.

Scripts are defined in a class called `Script` that can be imported from `luz`.

Variable	Type	Description
`type`	String	Type of script to run. Can be `preinst`, `postinst`, `prerm`, `postrm`.
`path`	String (Optional)	Path to the script to copy. (`None` if not specified)
`content`	String (Optional)	Content of the script to copy. (`None` if not specified)

Please note that either `path` or `content` must be specified. If both are specified, `path` will be used.

Modules

Modules are where you define the files to compile and the settings for the build.

Modules are defined in a class called `Modules` that can be imported from `luz`.

Variable	Type	Description
`type`	String	Type of module to build. (`tweak` if not specified)
`c_flags`	List	Flags to pass to `clang` when compiling C files.
`swift_flags`	List	Flags to pass to `swift` when compiling Swift files.
`linker_flags`	List	Flags to pass to the linker.
`optimization`	String	Optimization level to use for `clang`. (`0` if not specified)
`warnings`	List	Warnings flags to pass to `clang`. (`["-Wall"]` if not specified)
`ent_flags`	List	Entitlement flags to pass to `ldid`. (`["-S"]` if not specified)
`use_arc`	Boolean	Whether or not to use ARC for `clang`. (`true` if not specified)
`only_compile_changed`	Boolean	Whether or not to only compile changed files. (`true` if not specified)
`bridging_headers`	List	List of bridging headers to use for `swift`.
`frameworks`	List	List of frameworks to link against.
`private_frameworks`	List	List of private frameworks to link against.
`libraries`	List	List of libraries to link against.
`before_stage`	Callable	Function to run before staging.
`after_stage`	Callable	Function to run after staging.

Additional module options can be found here.

Submodules

Submodules are where you define paths to directories with `luz.py` files to include in your project.

Submodules are defined in a class called `Submodule` that can be imported from `luz`.

Variable	Type	Description
`path`	String	Path to the submodule.
`inherit`	String	Whether or not to inherit non-specified `meta` options from the parent project. (`true` if not specified)

Example `luzconf.py`

```
from luz import Control, Meta, Modules, Script, Submodule

# define meta options
meta = Meta(
    release=True,
    archs=['arm64', 'arm64e'],
    cc='/usr/bin/gcc',
    swift='/usr/bin/swift',
    compression='zstd',
    platform='iphoneos',
    sdk='~/.luz/sdks/iPhoneOS14.5.sdk',
    rootless=True,
    min_vers='15.0'
)

# define control options
control = Control(
    id='com.jaidan.demo',
    name='LuzBuildDemo',
    author='Jaidan',
    maintainer='Jaidan',
    description='LuzBuild demo',
    section='Tweaks',
    version='1.0.0',
    depends=['firmware (>= 15.0)', 'mobilesubstrate'],
    architecture='iphoneos-arm64'
)

# define scripts
scripts = [
    Script(type='postinst', path='./scripts/postinst'),
    Script(type='prerm', path='./scripts/prerm')
]

# define modules
modules = [
    Module(
        name='TestTweak',
        filter={
```

(continues on next page)

(continued from previous page)

```
            'bundles': ['com.apple.SpringBoard']
        },
        files=['Tweak.xm']
    )
]

# define submodules
submodules = [
    Submodule(path="./Preferences")
]
```