A Tour of Waf

BV

April 9, 2023

Contents

1 Introduction

The Waf book opens by aptly describing Waf as an extensible build system. That book is well written and along with the API documentation and the FAQ waf documentation is comprehensive. The goal of this Waf tour is to collect additional understanding of the Waf system from the point of view of a long time, if still amateur user, in the hopes it will augment existing Waf documentation.

1.1 Concepts

A Waf **build** is the execution of a set of interdependent *tasks*. A task consumes a number of **sources** and produces a number of **targets**. Sources and targets may be files, which Waf abstracts into objects called **nodes**. Sources may also be values of variables in a Waf **environment**. Tasks may be defined directly in a `wscript` file that provides Waf directives expressed in the Python language. Tasks may also be created during the build with **task generators**. Code to produce tasks may be bundled into reusable Waf **tools**. The detailed nature of the build may be controlled through a Waf **configuration** and that configuration is communicated and persisted with one or more Waf **environments**. The graph of tasks, sources and targets may be factored into one or more subgraphs or Waf **groups**. Each group will be executed in sequence. Within a group, tasks will execute in parallel to the extent possible, limited by dependencies and a maximum allowed number of jobs. Finally, configuration and build and other commands are managed through Waf **contexts**.

1.2 Examples

This tour will have a number of examples. They are run as part of preparing this document into HTML or PDF. Generally they include three major waf commands in a step like:

```
waf distclean configure build
```

The `distclean` is not typically required nor desired. It is included here to assure each example runs from a known initial state.

2 Environment

A Waf environment is similar to but distinct from both the shell environment that runs the `waf` command and the shell environments that may run commands in the service of a task. Where not otherwise qualified, an *environment* or *env* will refer to a Waf environment.

2.1 Calling environment

In general, the value of a variable in the calling shell environment has no impact on values in a Waf environment. However, some Waf code will allow the shell environment to influence values in the Waf environment.

```
from waflib.Logs import info
def configure(cfg):
    cfg.env.FOO = "bar"
    cfg.find_program("cc")
    info(str(cfg.env))
def build(bld):
    info(str(bld.env))
```

In `configure()` we set the Waf environment variable `FOO`. We also instruct Waf to find the program `cc`. By default, the resulting location will be placed in an environment variable of the same name but capitalized, namely here `CC`.

```
waf distclean configure
```

```
'distclean' finished successfully (0.002s)
Setting top to                           : /home/bv/org/topics/waf/tour/calling-enviro
Setting out to                           : /home/bv/org/topics/waf/tour/calling-enviro
Checking for program 'cc'                : /usr/bin/cc
'BINDIR' '/usr/local/bin'
'CC' ['/usr/bin/cc']
'FOO' 'bar'
'LIBDIR' '/usr/local/lib'
'PREFIX' '/usr/local'
'configure' finished successfully (0.003s)
```

In addition to the variables we set directly or through `find_program()`, Waf has set `PREFIX`, `BINDIR` and `LIBDIR`. We may return `configure` with some shell environment variables set.

```
PREFIX=/tmp CC=gcc FOO=baz waf configure
```

```
Setting top to                           : /home/bv/org/topics/waf/tour/calling-enviro
Setting out to                           : /home/bv/org/topics/waf/tour/calling-enviro
Checking for program 'cc'                : gcc
from configure:
FOO = bar
CC = ['gcc']
'configure' finished successfully (0.003s)
```

Note `FOO` remains unchanged while `CC` takes the value we gave in the shell environment. Not only is `PREFIX` changed but so are the two directories. Waf will persist an environments persist between calls to `configure` and `build`.

```
PREFIX=/nope waf build
```

```
Waf: Entering directory `/home/bv/org/topics/waf/tour/calling-environment/build'
from build:
FOO = bar
CC = ['gcc']
Waf: Leaving directory `/home/bv/org/topics/waf/tour/calling-environment/build'
'build' finished successfully (0.011s)
```

Rerunning `configure` will reset all variables.

```
waf configure
```

```
Setting top to                           : /home/bv/org/topics/waf/tour/calling-enviro
Setting out to                           : /home/bv/org/topics/waf/tour/calling-enviro
Checking for program 'cc'                : /usr/bin/cc
'BINDIR' '/usr/local/bin'
'CC' ['/usr/bin/cc']
'FOO' 'bar'
'LIBDIR' '/usr/local/lib'
'PREFIX' '/usr/local'
'configure' finished successfully (0.004s)
```

3 Nodes

Waf abstracts the file system into instances of a class `Node`. Nodes may be collected into a tree with a node representing a directory continuing children nodes or representing a file. Nodes may be created to represent an as yet non-existent file or directory.

3.1 Node features

```
from waflib.Logs import info
def configure(cfg):
    info(f'cfg.path = {cfg.path}')
    cfg.find_program('cat')
def build(bld):
    info(f'bld.path = {bld.path}')
    info(f'bld.bldnode = {bld.bldnode}')
    wscript = bld.path.find_resource("wscript")
    info(f'wscript = {wscript}')
    info(f'parent = {wscript.parent}')
    this = bld.path.make_node("this.txt")
    this.write("# end of file.\n")
    that = bld.path.find_or_declare("that.txt")
    bld(rule="${CAT} ${SRC} > ${TGT}", shell=True,
source=[wscript, this], target=that)
```

Here we print a few path values, make use of `.parent`. We also call node methods `find_resource()` (a node that must exist in the source) and `find_or_declare()` (a node that may exist already in the source or the build area or which may be created during the build) and `make_node()` to create a node from thin air. We also use a node as an file-like object to write. Finally, we tie it all into a rule that combines two source files, one that we created, to a target.

We will discus rules next.

```
waf configure build

Setting top to                           : /home/bv/org/topics/waf/tour/node-features
Setting out to                           : /home/bv/org/topics/waf/tour/node-features/
cfg.path = /home/bv/org/topics/waf/tour/node-features
Checking for program 'cat'               : /usr/bin/cat
'configure' finished successfully (0.004s)
Waf: Entering directory `/home/bv/org/topics/waf/tour/node-features/build'
bld.path = /home/bv/org/topics/waf/tour/node-features
bld.bldnode = /home/bv/org/topics/waf/tour/node-features/build
wscript = /home/bv/org/topics/waf/tour/node-features/wscript
parent = /home/bv/org/topics/waf/tour/node-features
Waf: Leaving directory `/home/bv/org/topics/waf/tour/node-features/build'
'build' finished successfully (0.011s)
```

The `.path` points to the current source directory. In a simple project, as in the example, this is the top-level directory. When a project consists of a number of sub-projects each in a sub directory and each with a `wscript` file, the `.path` will changes as the build descends. By setting the top-level variable `top` in a `wscript` file an alternative source directory may be specified. In addition the default location of `build/` for receiving targets may be changed by setting the variable `out` in the `wscript` file.

```
tail -1 build/that.txt

# end of file.
```

4 Rules

The kernel of Waf operation is a **rule**. It specifies a file transformation parameterized by the input or **source** files and/or the output or **target** files.

4.1 This to that

Produce one file from another file with a command and everything is hard-wired.

```
def configure(cfg):
    pass
def build(bld):
    bld(rule="cp ../this.txt that.txt")
```

Normally, one will never define fully literal rule commands nor hide input and output files from Waf as we do in this example.

Waf runs tasks from the `build/` directory and normally all targets produced by a task should place files into this same directory. Waf will assure proper file placement but here we are cutting Waf out of that duty. Next section will repeat this example in a more proper manner.

```
date > this.txt
waf distclean configure build

'distclean' finished successfully (0.002s)
Setting top to                           : /home/bv/org/topics/waf/tour/this-to-that
Setting out to                           : /home/bv/org/topics/waf/tour/this-to-that/bu
```

```
'configure' finished successfully (0.002s)
Waf: Entering directory `/home/bv/org/topics/waf/tour/this-to-that/build'
[1/1] Running cp ../this.txt that.txt
Waf: Leaving directory `/home/bv/org/topics/waf/tour/this-to-that/build'
'build' finished successfully (0.019s)
Sat Apr  8 05:19:18 PM EDT 2023
Sat Apr  8 05:19:18 PM EDT 2023

cat this.txt build/that.txt
```

Even though we have denied Waf any knowledge of what files the rule produced, it knows that it successfully ran the task once and will not rerun it:

```
waf build

Waf: Entering directory `/home/bv/org/topics/waf/tour/this-to-that/build'
Waf: Leaving directory `/home/bv/org/topics/waf/tour/this-to-that/build'
'build' finished successfully (0.018s)
```

4.2 Source to target

Here we do not deny Waf the knowledge of the input and output to a rule to repeat the above example in a more proper way.

```
def configure(cfg):
    pass
def build(bld):
    bld(rule="cp ${SRC} ${TGT}", source="this.txt", target="that.txt")
```

The `SRC` and `TGT` variables referenced in the rule string map to an list of input and output nodes, respectively. A node is a Waf file or directory object. By default, `${SRC}` expands to a space separated list of the paths of the nodes in the input array. Individual nodes in the array may be addressed through indexing, as in `${SRC[0]}`. Node methods may be called for example, `${SRC[0].abspath()}`.

Although we know that targets go into `build/` we need not mention that directory.

And, when specifying `source` or `target`, Waf is generous in that it can accept a file name as a string or array of file name strings, a node or array of node.

```
date > this.txt
waf distclean configure build
```

We suppress the output this time as it is same as above.

4.3 Rule function

So far, a rule has been given as a parameterized or templated command line string. A rule can also be a Python function.

```
from waflib.Utils import subst_vars
def copy(task):
    src = task.inputs[0]
    tgt = task.outputs[0]
    tgt.write(src.read())
def configure(cfg):
    pass
def build(bld):
    bld(rule=copy, source="this.txt", target="that.txt")
```

The `copy()` rule implements a primitive version of the `cp` command. We will again omit the output of running this example.

5 Tasks

6 Task generators

7 Contexts

8 Options

9 Configuration

The `configuration()` function has already been used in a number of examples. Here we further explore what this function may do.

10 Tools

10.1 Using tools

10.2 Writing tools

10.3 Built in and extra tools

10.4 Distributing tools

11 Advanced Waf

Much can be done simply with Waf as described so far. A dividing line between simple and advance may be crossed when it becomes necessary to extend Waf itself. Extension is done by "bolting on" and sometimes altering methods and values to Waf internal classes and objects. This is a perfectly acceptable thing to do and Waf provides some facilities and documentation on how to do it. However, it is one of the more complex ways to work with Waf, at least for this tourist.