Building with waf

BV

[2014-03-01 Sat 09:56]

The venerable workhorse of building software has long been `make` driven by its quirky and sometimes cryptic `Makefile` configurations. Anytime I must write even a moderately complex build system based on `make` I dream of having a simple, consistent and real programming language to describe the configuration instead of the language of a `Makefile`. Lacking it, when I use `make` I end up relying on a zoo of external tools which leads to a hodgepodge.

At some point I came across `scons` and thought my wishes were answered. It uses the lovely Python programming language so must be perfect. Unfortunately as I tried to use it I found a few problems and personal dislikes. It was difficult to extend and I found the built-in functionality awkward. These are admittedly very personal judgments.

More recently I came across `waf` and found it just about exactly what I was looking for. It uses Python, it can be extended easily (it forms the basis of my meta-build system worch). It has good, if not great documentation. This latter issue is what this topic is meant to address. It is written to collect my understanding as a user of waf and hopefully fill a needed gap. It tries to provide concrete examples of how to do various things and refer to the official documentation wherever possible.

1 Gestalt of waf

Waf is like `make` but with a `Makefile` called `wscript` and written in Python. Waf is unlike in several ways:

- waf can be extended by providing Python modules loaded through `wscript` files
- waf can be bundled along with extensions to provide a single executable that performs specific tasks

- waf is cross-platform, no compilation needed and can be included as a single file along with the project it builds
- waf is parallel by default, it will run tasks as parallel as possible constrained by available CPU, dependencies or any limits imposed by the user

2 The waf configuration file

Waf expects to find a file called `wscript` in the current directory.

2.1 Commands

Functions in this file become exposed through waf as command line commands. For example:

```
def chirp(ctx):
    print (ctx)
```

can be exercised as:

```
cd examples/commands/
waf chirp
```

2.2 Context object

A waf command function is given a context object. This context object may be specialized depending on the function called. The figure from the context reference docs shows the inheritance:

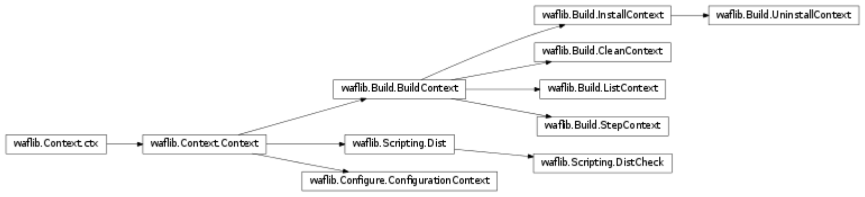
waflib.Context.ctx
waflib.Context.Context
waflib.Build.BuildContext
waflib.Scripting.Dist
waflib.Configure.ConfigurationContext
waflib.Build.InstallContext
waflib.Build.UninstallContext
waflib.Build.CleanContext
waflib.Build.ListContext
waflib.Build.StepContext
waflib.Scripting.DistCheck

Figure 1: Inheritance of context classes. Note, the arrows are reversed from the sense they would be drawn in a UML inheritance diagram.

3 Predefined commands

Certain commands are reserved and treated special by waf.

3.1 Options

The `options(ctx)` function will be passed an options context object. This function can be used to define command line options that waf will recognize on behalf of the project.

```
def options(opt):
    opt.add_option('-x','--extra',action='store',
   help='Add something extra from the command line')

def chirp(ctx):
    print ('The little bit of extra is: "%s"' % ctx.options.extra)

cd examples/options/
waf --extra='Just a bit of extra stuff' chirp
```

3.2 Configure

The `configure(cfg)` function is passed a configuration context object. This function can be used to persist any information between other command calls. On possible use is to make command line options persisted.

```
def options(opt):
    opt.add_option('-x','--extra',action='store',
   help='Add something extra from the command line')

def configure(cfg):
    cfg.env.EXTRA_MSG = 'The little bit of extra is: "%s"' % cfg.options.extra

def build(bld):
    print (bld.env.EXTRA_MSG)

cd examples/configure/
waf -x 'Persist This' configure
echo "Configure done"
waf
```

3.3 Build

In the `build(bld)` function is where one describes to waf how to build everything. It is passed a build context object. There are several ways to do this but a simple and powerful way is to declare a task generator by calling `bld` as a callable object.

```
def configure(cfg):
    return

def build(bld):
    bld(rule="date > ${TGT}", target = "one.txt")
    bld(rule="cp ${SRC} ${TGT} && date >> ${TGT}", source="one.txt", target="two.txt")
    bld.install_files("${PREFIX}/examples", "one.txt two.txt")
```

```
cd examples/build
rm -rf build install
waf --prefix=install configure build install
ls -l build
ls -l install/examples
```

In this example two files are created, `one.txt` out of thin air and `two.txt` based on `one.txt`. Both of these files are then installed into a location based on the value of `PREFIX` which is set by the standard waf `--prefix` command line option. A task is generated for each invocation of `bld()` as set by the `rule`. Here the rule is a scriptlet which is essentially a shell script command with some string interpolation. As can be seen, file redirection and other shell operators can be used.