Developing Modules

Topics

Ansible modules are reusable units of magic that can be used by the Ansible API, or by the *ansible* or *ansible-playbook* programs.

See About Modules for a list of various ones developed in core.

Modules can be written in any language and are found in the path specified by *ANSIBLE_LIBRARY* or the `--module-path` command line option.

By default, everything that ships with ansible is pulled from its source tree, but additional paths can be added.

The directory "./library", alongside your top level playbooks, is also automatically added as a search directory.

Should you develop an interesting Ansible module, consider sending a pull request to the modules-extras project. There's also a core repo for more established and widely used modules. "Extras" modules may be promoted to core periodically, but there's no fundamental

difference in the end - both ship with ansible, all in one package, regardless of how you acquire ansible.

Tutorial

Let's build a very-basic module to get and set the system time. For starters, let's build a module that just outputs the current time.

We are going to use Python here but any language is possible. Only File I/O and outputting to standard out are required. So, bash, C++, clojure, Python, Ruby, whatever you want is fine.

Now Python Ansible modules contain some extremely powerful shortcuts (that all the core modules use) but first we are going to build a module the very hard way. The reason we do this is because modules written in any language OTHER than Python are going to have to do exactly this. We'll show the easy way later.

So, here's an example. You would never really need to build a module to set the system time, the 'command' module could already be used to do this. Though we're going to make one.

Reading the modules that come with ansible (linked above) is a great way to learn how to write modules. Keep in mind, though, that some modules in ansible's source tree are internalisms, so look at *service* or *yum*, and don't stare too close into things like *async_wrapper* or you'll turn to stone. Nobody ever executes async_wrapper directly.

Ok, let's get going with an example. We'll use Python. For starters, save this as a file named *timetest.py*:

```
#!/usr/bin/python

import datetime
import json

date = str(datetime.datetime.now())
print json.dumps({
    "time" : date
})
```

Testing Modules

There's a useful test script in the source checkout for ansible:

```
git clone git://github.com/ansible/ansible.git --recursive
source ansible/hacking/env-setup
chmod +x ansible/hacking/test-module
```

Let's run the script you just wrote with that:

```
ansible/hacking/test-module -m ./timetest.py
```

You should see output that looks something like this:

```
{u'time': u'2012-03-14 22:13:48.539183'}
```

If you did not, you might have a typo in your module, so recheck it and try again.

Reading Input

Let's modify the module to allow setting the current time. We'll do this by seeing if a key value pair in the form *time=<string>* is passed in to the module.

Ansible internally saves arguments to an arguments file. So we must read the file and parse it. The arguments file is just a string, so any form of arguments are legal. Here we'll do some basic parsing to treat the input as key=value.

The example usage we are trying to achieve to set the time is:

```
time time="March 14 22:10"
```

If no time parameter is set, we'll just leave the time as is and return the current time.

> **注解**
>
> This is obviously an unrealistic idea for a module. You'd most likely just use the shell module. However, it probably makes a decent tutorial.

Let's look at the code. Read the comments as we'll explain as we go. Note that this is highly verbose because it's intended as an educational example. You can write modules a lot shorter than this:

```
#!/usr/bin/python

# import some python modules that we'll use.  These are all
# available in Python's core

import datetime
import sys
import json
import os
import shlex

# read the argument string from the arguments file
args_file = sys.argv[1]
args_data = file(args_file).read()

# for this module, we're going to do key=value style arguments
# this is up to each module to decide what it wants, but all
# core modules besides 'command' and 'shell' take key=value
# so this is highly recommended

arguments = shlex.split(args_data)
for arg in arguments:

    # ignore any arguments without an equals in it
    if "=" in arg:

        (key, value) = arg.split("=")

        # if setting the time, the key 'time'
        # will contain the value we want to set the time to

        if key == "time":

            # now we'll affect the change.  Many modules
            # will strive to be 'idempotent', meaning they
            # will only make changes when the desired state
            # expressed to the module does not match
            # the current state.  Look at 'service'
            # or 'yum' in the main git tree for an example
            # of how that might look.

            rc = os.system("date -s \"%s\"" % value)

            # always handle all possible errors
            #
            # when returning a failure, include 'failed'
            # in the return data, and explain the failure
            # in 'msg'.  Both of these conventions are
            # required however additional keys and values
            # can be added.

            if rc != 0:
                print json.dumps({
                    "failed" : True,
                    "msg"    : "failed setting the time"
                })
                sys.exit(1)

            # when things do not fail, we do not
            # have any restrictions on what kinds of
            # data are returned, but it's always a
            # good idea to include whether or not
            # a change was made, as that will allow
            # notifiers to be used in playbooks.

            date = str(datetime.datetime.now())
            print json.dumps({
                "time" : date,
                "changed" : True
            })
            sys.exit(0)

# if no parameters are sent, the module may or
# may not error out, this one will just
# return the time
```

```
date = str(datetime.datetime.now())
print json.dumps({
    "time" : date
})
```

Let's test that module:

```
ansible/hacking/test-module -m ./time -a "time=\"March 14 12:23\""
```

This should return something like:

```
{"changed": true, "time": "2012-03-14 12:23:00.000307"}
```

Module Provided 'Facts'

The 'setup' module that ships with Ansible provides many variables about a system that can be used in playbooks and templates. However, it's possible to also add your own facts without modifying the system module. To do this, just have the module return a *ansible_facts* key, like so, along with other return data:

```
{
    "changed" : True,
    "rc" : 5,
    "ansible_facts" : {
        "leptons" : 5000,
        "colors" : {
            "red"   : "FF0000",
            "white" : "FFFFFF"
        }
    }
}
```

These 'facts' will be available to all statements called after that module (but not before) in the playbook. A good idea might be to make a module called 'site_facts' and always call it at the top of each playbook, though we're always open to improving the selection of core facts in Ansible as well.

Common Module Boilerplate

As mentioned, if you are writing a module in Python, there are some very powerful shortcuts you can use. Modules are still transferred as one file, but an arguments file is no longer needed, so these are not only shorter in terms of code, they are actually FASTER in terms of execution time.

Rather than mention these here, the best way to learn is to read some of the source of the modules that come with Ansible.

The 'group' and 'user' modules are reasonably non-trivial and showcase what this looks like.

Key parts include always ending the module file with:

```
from ansible.module_utils.basic import *
if __name__ == '__main__':
    main()
```

And instantiating the module class like:

```
module = AnsibleModule(
    argument_spec = dict(
        state     = dict(default='present', choices=['present', 'absent']),
        name      = dict(required=True),
        enabled   = dict(required=True, choices=BOOLEANS),
        something = dict(aliases=['whatever'])
    )
)
```

The AnsibleModule provides lots of common code for handling returns, parses your arguments for you, and allows you to check inputs.

Successful returns are made like this:

```
module.exit_json(changed=True, something_else=12345)
```

And failures are just as simple (where 'msg' is a required parameter to explain the error):

```
module.fail_json(msg="Something fatal happened")
```

There are also other useful functions in the module class, such as module.sha1(path). See lib/ansible/module_common.py in the source checkout for implementation details.

Again, modules developed this way are best tested with the hacking/test-module script in the git source checkout. Because of the magic involved, this is really the only way the scripts can function outside of Ansible.

If submitting a module to ansible's core code, which we encourage, use of the AnsibleModule class is required.

Check Mode

1.1 新版功能.

Modules may optionally support check mode. If the user runs Ansible in check mode, the module should try to predict whether changes will occur.

For your module to support check mode, you must pass `supports_check_mode=True` when instantiating the AnsibleModule object. The AnsibleModule.check_mode attribute will evaluate to True when check mode is enabled. For example:

```
module = AnsibleModule(
    argument_spec = dict(...),
    supports_check_mode=True
)

if module.check_mode:
    # Check if any changes would be made but don't actually make those changes
    module.exit_json(changed=check_if_system_state_would_be_changed())
```

Remember that, as module developer, you are responsible for ensuring that no system state is altered when the user enables check mode.

If your module does not support check mode, when the user runs Ansible in check mode, your module will simply be skipped.

Common Pitfalls

You should also never do this in a module:

```
print "some status message"
```

Because the output is supposed to be valid JSON.

Modules must not output anything on standard error, because the system will merge standard out with standard error and prevent the JSON from parsing. Capturing standard error and returning it as a variable in the JSON on standard out is fine, and is, in fact, how the command module is implemented.

If a module returns stderr or otherwise fails to produce valid JSON, the actual output will still be shown in Ansible, but the command will not succeed.

Always use the hacking/test-module script when developing modules and it will warn you about these kind of things.

Conventions/Recommendations

As a reminder from the example code above, here are some basic conventions and guidelines:

- If the module is addressing an object, the parameter for that object should be called 'name' whenever possible, or accept 'name' as an alias.
- If you have a company module that returns facts specific to your installations, a good name for this module is *site_facts*.
- Modules accepting boolean status should generally accept 'yes', 'no', 'true', 'false', or anything else a user may likely throw at them. The AnsibleModule common code supports this with "choices=BOOLEANS" and a module.boolean(value) casting function.
- Include a minimum of dependencies if possible. If there are dependencies, document them at the top of the module file, and have the module raise JSON error messages when the import fails.
- Modules must be self-contained in one file to be auto-transferred by ansible.
- If packaging modules in an RPM, they only need to be installed on the control machine and should be dropped into /usr/share/ansible. This is entirely optional and up to you.
- Modules must output valid JSON only. The toplevel return type must be a hash (dictionary) although they can be nested. Lists or simple scalar values are not supported, though they can be trivially contained inside a dictionary.
- In the event of failure, a key of 'failed' should be included, along with a string explanation in 'msg'. Modules that raise tracebacks (stacktraces) are generally considered 'poor' modules, though Ansible can deal with these returns and will automatically convert anything unparseable into a failed result. If you are using the AnsibleModule common Python code, the 'failed' element will be included for you automatically when you call 'fail_json'.
- Return codes from modules are not actually not significant, but continue on with 0=success and non-zero=failure for reasons of future proofing.
- As results from many hosts will be aggregated at once, modules should return only relevant output. Returning the entire contents of a log file is generally bad form.

Documenting Your Module

All modules included in the CORE distribution must have a `DOCUMENTATION` string. This string MUST be a valid YAML document which conforms to the schema defined below. You may find it easier to start writing your `DOCUMENTATION` string in an editor with YAML syntax highlighting before you include it in your Python file.

Example

See an example documentation string in the checkout under examples/DOCUMENTATION.yml.

Include it in your module file like this:

```
#!/usr/bin/python
# Copyright header....

DOCUMENTATION = '''
---
module: modulename
short_description: This is a sentence describing the module
# ... snip ...
'''
```

The `description`, and `notes` fields support formatting with some special macros.

These formatting functions are `U()`, `M()`, `I()`, and `C()` for URL, module, italic, and constant-width respectively. It is suggested to use `C()` for file and option names, and `I()` when referencing parameters; module names should be specified as `M(module)`.

Examples (which typically contain colons, quotes, etc.) are difficult to format with YAML, so these must be written in plain text in an `EXAMPLES` string within the module like this:

```
EXAMPLES = '''
- action: modulename opt1=arg1 opt2=arg2
'''
```

The EXAMPLES section, just like the documentation section, is required in all module pull requests for new modules.

The RETURN section documents what the module returns. For each value returned, provide a `description`, in what circumstances the value is `returned`, the `type` of the value and a `sample`. For example, from the `copy` module:

```
RETURN = '''
dest:
    description: destination file/path
    returned: success
    type: string
    sample: "/path/to/file.txt"
src:
    description: source file used for the copy on the target machine
    returned: changed
    type: string
    sample: "/home/httpd/.ansible/tmp/ansible-tmp-1423796390.97-147729857856000/source"
md5sum:
    description: md5 checksum of the file after running copy
    returned: when supported
    type: string
    sample: "2a5aeecc61dc98c4d780b14b330e3282"
...
'''
```

Building & Testing

Put your completed module file into the 'library' directory and then run the command: `make webdocs`. The new 'modules.html' file will be built and appear in the 'docsite/' directory.

小技巧

If you're having a problem with the syntax of your YAML you can validate it on the YAML Lint website.

小技巧

You can set the environment variable ANSIBLE_KEEP_REMOTE_FILES=1 on the controlling host to prevent ansible from deleting the remote files so you can debug your module.

Module Paths

If you are having trouble getting your module "found" by ansible, be sure it is in the `ANSIBLE_LIBRARY` environment variable.

If you have a fork of one of the ansible module projects, do something like this:

```
ANSIBLE_LIBRARY=~/ansible-modules-core:~/ansible-modules-extras
```

And this will make the items in your fork be loaded ahead of what ships with Ansible. Just be sure to make sure you're not reporting bugs on versions from your fork!

To be safe, if you're working on a variant on something in Ansible's normal distribution, it's not a bad idea to give it a new name while you are working on it, to be sure you know you're pulling your version.

Getting Your Module Into Ansible

High-quality modules with minimal dependencies can be included in Ansible, but modules (just due to the programming preferences of the developers) will need to be implemented in Python and use the AnsibleModule common code, and should generally use consistent arguments with the rest of the program. Stop by the mailing list to inquire about requirements if you like, and submit a github pull request to the extras project. Included modules will ship with ansible, and also have a chance to be promoted to 'core' status, which gives them slightly higher development priority (though they'll work in exactly the same way).

Module checklist

- The shebang should always be #!/usr/bin/python, this allows ansible_python_interpreter to work
- **Documentation: Make sure it exists**
    - *required* should always be present, be it true or false
    - If *required* is false you need to document *default*, even if the default is 'None' (which is the default if no parameter is supplied). Make sure default parameter in docs matches default parameter in code.
    - *default* is not needed for *required: true*
    - Remove unnecessary doc like *aliases: []* or *choices: []*
    - The version is not a float number and value the current development version
    - Verify that arguments in doc and module spec dict are identical
    - For password / secret arguments no_log=True should be set
    - Requirements should be documented, using the *requirements=[]* field
    - Author should be set, name and github id at least
    - Made use of U() for urls, C() for files and options, I() for params, M() for modules?
    - GPL 3 License header
    - Does module use check_mode? Could it be modified to use it? Document it
    - Examples: make sure they are reproducible
    - Return: document the return structure of the module

- **Exceptions: The module must handle them. (exceptions are bugs)**
    - Give out useful messages on what you were doing and you can add the exception message to that.
    - Avoid catchall exceptions, they are not very useful unless the underlying API gives very good error messages pertaining the attempted action.

- The module must not use sys.exit() –> use fail_json() from the module object
- Import custom packages in try/except and handled with fail_json() in main() e.g.:

```
try:
    import foo
    HAS_LIB=True
except:
    HAS_LIB=False
```

- The return structure should be consistent, even if NA/None are used for keys normally returned under other options.
- Are module actions idempotent? If not document in the descriptions or the notes
- Import module snippets *from ansible.module_utils.basic import ** at the bottom, conserves line numbers for debugging.
- Call your `main()` from a conditional so that it would be possible to test them in the future example:

```
if __name__ == '__main__':
    main()
```

- Try to normalize parameters with other modules, you can have aliases for when user is more familiar with underlying API name for the option
- Being pep8 compliant is nice, but not a requirement. Specifically, the 80 column limit now hinders readability more that it improves it
- Avoid '*action*/*command*', they are imperative and not declarative, there are other ways to express the same thing
- Do not add *list* or *info* state options to an existing module - create a new *_facts* module.
- If you are asking 'how can I have a module execute other modules' ... you want to write a role
- Return values must be able to be serialized as json via the python stdlib json library. basic python types (strings, int, dicts, lists, etc) are serializable. A common pitfall is to try returning an object via exit_json(). Instead, convert the fields you need from the object into the fields of a dictionary and return the dictionary.
- When fetching URLs, please use either fetch_url or open_url from ansible.module_utils.urls rather than urllib2; urllib2 does not natively verify TLS certificates and so is insecure for https.

Windows modules checklist

- Favour native powershell and .net ways of doing things over calls to COM libraries or calls to native executables which may or may not be present in all versions of windows
- modules are in powershell (.ps1 files) but the docs reside in same name python file (.py)
- look at ansible/lib/ansible/module_utils/powershell.ps1 for common code, avoid duplication
- Ansible uses strictmode version 2.0 so be sure to test with that enabled
- start with:

```
#!powershell
```

then::
    <GPL header>

then::
    # WANT_JSON # POWERSHELL_COMMON

then, to parse all arguments into a variable modules generally use::
    $params = Parse-Args $args

- **Arguments:**
    - Try and use state present and state absent like other modules
    - You need to check that all your mandatory args are present. You can do this using the builtin Get-AnsibleParam function.
    - **Required arguments::**
        $package = Get-AnsibleParam -obj $params -name name -failifempty $true

- **Required arguments with name validation::**
  $state = Get-AnsibleParam -obj $params -name “State” -ValidateSet “Present”,”Absent” -resultobj $resultobj -failifempty $true

- **Optional arguments with name validation::**
  $state = Get-AnsibleParam -obj $params -name “State” -default “Present” -ValidateSet “Present”,”Absent”

- the If “FailIfEmpty” is true, the resultobj parameter is used to specify the object returned to fail-json. You can also override the default message using $emptyattributefailmessage (for missing required attributes) and $ValidateSetErrorMessage (for attribute validation errors)
- Look at existing modules for more examples of argument checking.

- **Results**
  - The result object should allways contain an attribute called changed set to either $true or $false
  - Create your result object like this:

```
$result = New-Object psobject @{
changed = $false
other_result_attribute = $some_value
};

If all is well, exit with a
Exit-Json $result
```

  - Ensure anything you return, including errors can be converted to json.
  - Be aware that because exception messages could contain almost anything.
  - ConvertTo-Json will fail if it encounters a trailing in a string.
  - If all is not well use Fail-Json to exit.

- Have you tested for powershell 3.0 and 4.0 compliance?

Deprecating and making module aliases

Starting in 1.8 you can deprecate modules by renaming them with a preceding _, i.e. old_cloud.py to _old_cloud.py, This will keep the module available but hide it from the primary docs and listing.

You can also rename modules and keep an alias to the old name by using a symlink that starts with _. This example allows the stat module to be called with fileinfo, making the following examples equivalent

EXAMPLES = ‘’’ ln -s stat.py _fileinfo.py ansible -m stat -a “path=/tmp” localhost ansible -m fileinfo -a “path=/tmp” localhost ‘’‘

参见

About Modules
Learn about available modules

Developing Plugins
Learn about developing plugins

Python API
Learn about the Python API for playbook and task execution

GitHub Core modules directory
Browse source of core modules

Github Extras modules directory
Browse source of extras modules.

Mailing List
Development mailing list

irc.freenode.net
#ansible IRC chat channel