Debugging modules

- Detailed debugging steps
- Simple debugging

Detailed debugging steps

Ansible modules are put together as a zip file consisting of the module file and the various Python module boilerplate inside of a wrapper script. To see what is actually happening in the module, you need to extract the file from the wrapper. The wrapper script provides helper methods that let you do that.

The following steps use `localhost` as the target host, but you can use the same steps to debug against remote hosts as well. For a simpler approach to debugging without using the temporary files, see `simple debugging <simple_debugging>`.

1. Set `ANSIBLE_KEEP_REMOTE_FILES` to `1` on the control host so Ansible will keep the remote module files instead of deleting them after the module finishes executing. Use the `-vvv` option to make Ansible more verbose. This will display the file name of the temporary module file.

```
$ ANSIBLE_KEEP_REMOTE_FILES=1 ansible localhost -m ping -a
'data=debugging_session' -vvv
<127.0.0.1> ESTABLISH LOCAL CONNECTION FOR USER: badger
<127.0.0.1> EXEC /bin/sh -c '( umask 77 && mkdir -p "` echo
$HOME/.ansible/tmp/ansible-tmp-1461434734.35-235318071810595 `" &&
echo "` echo $HOME/.ansible/tmp/ansible-tmp-1461434734.35-
235318071810595 `" )'
<127.0.0.1> PUT /var/tmp/tmpjdbJ1w TO
/home/badger/.ansible/tmp/ansible-tmp-1461434734.35-
235318071810595/AnsiballZ_ping.py
```

```
<127.0.0.1> EXEC /bin/sh -c 'LANG=en_US.UTF-8 LC_ALL=en_US.UTF-8
LC_MESSAGES=en_US.UTF-8 /usr/bin/python
/home/badger/.ansible/tmp/ansible-tmp-1461434734.35-
235318071810595/AnsiballZ_ping.py && sleep 0'
localhost | SUCCESS => {
    "changed": false,
    "invocation": {
        "module_args": {
            "data": "debugging_session"
        },
        "module_name": "ping"
    },
    "ping": "debugging_session"
}
```

2. Navigate to the temporary directory from the previous step. If the previous command was run against a remote host, connect to that host first before trying to navigate to the temporary directory.

```
$ ssh remotehost  # only if not debugging against localhost
$ cd /home/badger/.ansible/tmp/ansible-tmp-1461434734.35-235318071810595
```

3. Run the wrapper's `explode` command to turn the string into some Python files that you can work with.

> ```
> $ python AnsiballZ_ping.py explode
> Module expanded into:
> /home/badger/.ansible/tmp/ansible-tmp-1461434734.35-
> 235318071810595/debug_dir
> ```
>
> If you want to examine the wrapper file you can. It will show a small Python script with a large base64 encoded string. The string contains the module to execute.

4. When you look into the temporary directory you'll see a structure like this:

> ```
> ├── AnsiballZ_ping.py
> └── debug_dir
> ```

```
    ├── ansible
    │   ├── __init__.py
    │   ├── module_utils
    │   │   ├── __init__.py
    │   │   ├── _text.py
    │   │   ├── basic.py
    │   │   ├── common
    │   │   ├── compat
    │   │   ├── distro
    │   │   ├── parsing
    │   │   ├── pycompat24.py
    │   │   └── six
    │   └── modules
    │       ├── __init__.py
    │       └── ping.py
    └── args
```

- `AnsiballZ_ping.py` is the Python script with the module code stored in a base64 encoded string. It contains various helper functions for executing the module.
- `ping.py` is the code for the module itself. You can modify this code to see what effect it would have on your module, or for debugging purposes.
- The `args` file contains a JSON string. The string is a dictionary containing the module arguments and other variables that Ansible passes into the module to change its behavior. Modify this file to change the parameters passed to the module.
- The `ansible` directory contains the module code in `modules` as well as code from `ansible.module_utils` that is used by the module. Ansible includes files for any `ansible.module_utils` imports in the module but not any files from any other module. If your module uses `ansible.module_utils.url` Ansible will include it for you. But if your module includes requests, then you'll have to make sure that the Python requests library is installed on the system before running the module.

You can modify files in this directory if you suspect that the module is having a problem in some of this boilerplate code rather than in the module code you have written.

5. Once you edit the code or arguments in the exploded tree, use the `execute` subcommand to run it:

> ```
> $ python AnsiballZ_ping.py execute
> {"invocation": {"module_args": {"data": "debugging_session"}},
> "changed": false, "ping": "debugging_session"}
> ```
>
> This subcommand inserts the absolute path to `debug_dir` as the first item in `sys.path` and invokes the script using the arguments in the `args` file. You can continue to run the module like this until you understand the problem. Then you can copy the changes back into your real module file and test that the real module works by using the `ansible` or `ansible-playbook`.

Simple debugging

The easiest way to run a debugger in a module, either local or remote, is to use madbg. Add `import madbg; madbg.set_trace()` in the module code on the control node at the desired break point. To connect to the debugger, run `madbg connect`. See the madbg documentation for how to specify the `host` and `port`. If connecting to a remote node, make sure to use a port that is allowed by any firewall between the control node and the remote node.

This technique should work with any remote debugger, but we do not guarantee any particular remote debugging tool will work.

The q library is another very useful debugging tool.

Since `print()` statements do not work inside modules, raising an exception is a good approach if you just want to see some specific data. Put `raise Exception(some_value)` somewhere in the module and run it normally. Ansible will handle this exception, pass the message back to the control node, and display it.

□□□ #2
□ □□□ □□□ 2026-10-31 14:33:31 UTC
□ □□□ □□□ 2026-10-31 14:33:31 UTC