Ansible Callback Plugin - Line-by-Line Code Flow Explanation

Phase 1: Plugin File Structure and Imports

```python
#!/usr/bin/python
```

Flow: Shebang line - tells the system this is a Python script.

```python
from ansible.plugins.callback import CallbackBase
```

Flow: Import the base class that all callback plugins must inherit from. This provides the framework for plugin functionality.

```python
from datetime import datetime
```

Flow: Import datetime module to generate timestamps for log entries.

Phase 2: Class Definition and Metadata

```python
class CallbackModule(CallbackBase):
```

Flow:

1. Define a class named `CallbackModule` (this exact name is required by Ansible)
2. Inherit from `CallbackBase` to get all the callback framework functionality
3. Ansible looks for this specific class name when loading plugins

```python
    """
    Simple callback plugin that logs task results to a file
    """
```

Flow: Documentation string describing the plugin's purpose.

```python
CALLBACK_VERSION = 2.0
```

Flow:

1. Class attribute defining the callback API version
2. Ansible checks this to ensure compatibility
3. Version 2.0 is the current standard

```python
CALLBACK_TYPE = 'notification'
```

Flow:

1. Defines the plugin type
2. Types: 'notification' (runs alongside default output), 'stdout' (replaces default output)
3. 'notification' type allows multiple plugins to run simultaneously

```python
CALLBACK_NAME = 'simple_logger'
```

Flow:

1. Unique identifier for this plugin
2. Used in ansible.cfg to enable the plugin: `callback_whitelist = simple_logger`
3. Must match the plugin filename (simple_logger.py)

Phase 3: Initialization (init method)

```python
def __init__(self):
```

Flow:

1. Constructor method called when Ansible loads the plugin
2. Runs once before playbook execution
3. Sets up initial state for the plugin

```python
        super(CallbackModule, self).__init__()
```

Flow:

1. Call the parent class (CallbackBase) constructor
2. Initializes the base callback framework
3. Sets up internal Ansible callback mechanisms
4. Must be called before any plugin-specific initialization

```python
        self.log_file = '/tmp/ansible_playbook.log'
```

Flow:

1. Instance variable storing the log file path
2. This variable persists throughout the entire playbook execution
3. All log methods will write to this file

```python
        self.task_count = 0
```

Flow:

1. Counter initialized to zero
2. Will be incremented each time a task starts
3. Used to track total number of tasks executed

```python
        self.task_ok = 0
```

Flow:

1. Counter for successful task executions
2. Incremented in v2_runner_on_ok method
3. Used in final statistics

```python
```

```
        self.task_failed = 0
```

Flow:

1. Counter for failed task executions
2. Incremented in v2_runner_on_failed method
3. Used in final statistics

Phase 4: Helper Method (log function)

```
python

    def log(self, message):
```

Flow:

1. Define a custom helper method (not an Ansible hook)
2. Takes a message string as parameter
3. Will be called by other methods to write to log file

```
python

        """Write message to log file"""
```

Flow: Documentation for the helper method.

```
python

        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
```

Flow:

1. Call datetime.now() to get current date and time
2. Use strftime() to format as: 2024-11-14 10:30:15
3. Store formatted string in timestamp variable

```
python

        with open(self.log_file, 'a') as f:
```

Flow:

1. Open the log file in append mode ('a' means add to end, don't overwrite)
2. Use context manager (with) to automatically close file after writing
3. Assign file object to variable 'f'

```python
        f.write(f"[{timestamp}] {message}\n")
```

Flow:

1. Use f-string to format: [2024-11-14 10:30:15] message text
2. Add newline character (\n) at the end
3. Write the formatted string to the file
4. File is automatically closed when with block ends

Phase 5: Playbook Start Hook

```python
    def v2_playbook_on_start(self, playbook):
```

Flow:

1. **ANSIBLE CALLS THIS METHOD** when playbook execution begins
2. Ansible automatically passes the playbook object as parameter
3. This is the first hook that fires in the execution sequence

```python
        """Called when playbook starts"""
```

Flow: Documentation explaining when Ansible calls this method.

```python
        self.log("=" * 50)
```

Flow:

1. Create a string of 50 equal signs
2. Call our log helper method with this string

3. Writes: [timestamp] ==========

```
python

        self.log("PLAYBOOK STARTED")
```

Flow:

1. Call log helper method with "PLAYBOOK STARTED"
2. Writes: [timestamp] PLAYBOOK STARTED

```
python

        self.log("=" * 50)
```

Flow:

1. Another separator line
2. Creates visual boundary in log file

Phase 6: Task Start Hook

```
python

    def v2_playbook_on_task_start(self, task, is_conditional):
```

Flow:

1. **ANSIBLE CALLS THIS METHOD** each time a task is about to execute
2. Receives two parameters:
   - task: object containing task information
   - is_conditional: boolean indicating if task has 'when' conditions
3. Called for EVERY task in the playbook

```
python

        """Called when each task starts"""
```

Flow: Documentation.

```
python
```

```
        self.task_count += 1
```

Flow:

1. Increment the task counter (0 → 1, then 1 → 2, etc.)
2. Tracks total number of tasks executed
3. Value persists because it's an instance variable

```
python

        self.log(f"TASK {self.task_count}: {task.get_name()}")
```

Flow:

1. Call task.get_name() method to retrieve task name from task object
2. Format string: "TASK 1: Create test directory"
3. Call log helper method with this formatted string
4. Writes: [timestamp] TASK 1: Create test directory

Phase 7: Task Success Hook

```
python

    def v2_runner_on_ok(self, result):
```

Flow:

1. **ANSIBLE CALLS THIS METHOD** when a task completes successfully
2. Receives result object containing:
   - Host information
   - Task details
   - Return values from the task
3. Called once per host per successful task

```
python

        """Called when task succeeds"""
```

Flow: Documentation.

```python
        self.task_ok += 1
```

Flow:

1. Increment success counter
2. Tracks how many tasks succeeded
3. Used in final statistics

```python
        host = result._host.get_name()
```

Flow:

1. Access the _host attribute of result object
2. Call get_name() method to get hostname
3. Returns string like "localhost" or "webserver1"
4. Store in local variable 'host'

```python
        task = result._task.get_name()
```

Flow:

1. Access the _task attribute of result object
2. Call get_name() method to get task name
3. Returns string like "Create test directory"
4. Store in local variable 'task'

```python
        self.log(f"  ✓ SUCCESS on {host}: {task}")
```

Flow:

1. Format string with host and task variables
2. Two spaces at start for indentation
3. Checkmark symbol (✓) indicates success

4. Example: " ✓ SUCCESS on localhost: Create test directory"
5. Call log helper to write to file

Phase 8: Task Failure Hook

```
python

    def v2_runner_on_failed(self, result, ignore_errors=False):
```

Flow:

1. **ANSIBLE CALLS THIS METHOD** when a task fails
2. Parameters:
   - result: contains failure details
   - ignore_errors: boolean, True if task has ignore_errors=True
3. Called once per host per failed task

```
python

        """Called when task fails"""
```

Flow: Documentation.

```
python

        self.task_failed += 1
```

Flow:

1. Increment failure counter
2. Tracks total failed tasks

```
python

        host = result._host.get_name()
        task = result._task.get_name()
```

Flow:

1. Same as v2_runner_on_ok
2. Extract host and task names from result object

```python
        self.log(f"  ✗ FAILED on {host}: {task}")
```

Flow:

1. Format string with X symbol (✗) for failure
2. Example: " ✗ FAILED on localhost: Invalid command"
3. Write to log file

Phase 9: Playbook Statistics Hook

```python
    def v2_playbook_on_stats(self, stats):
```

Flow:

1. **ANSIBLE CALLS THIS METHOD** when playbook completes
2. This is the LAST hook that fires
3. Receives stats object containing execution statistics
4. Called once per playbook run

```python
        """Called when playbook ends"""
```

Flow: Documentation.

```python
        self.log("-" * 50)
```

Flow:

1. Create separator line with dashes
2. Visually separates summary from task logs

```python
        self.log(f"SUMMARY: Total={self.task_count}, OK={self.task_ok}, Failed={self.task_failed}")
```

Flow:

1. Access instance variables: task_count, task_ok, task_failed
2. These were incremented throughout playbook execution
3. Format summary string: "SUMMARY: Total=5, OK=5, Failed=0"
4. Write to log file

```
python
    self.log("PLAYBOOK FINISHED")
```

Flow:

1. Write completion message to log

```
python
    self.log("=" * 50)
```

Flow:

1. Final separator line
2. Matches the opening separator from v2_playbook_on_start

```
python
    print(f"\n📝 Detailed log saved to: {self.log_file}")
```

Flow:

1. Use print() instead of log() to write to console
2. \n adds blank line before message
3. Shows user where to find the detailed log
4. Appears in Ansible console output

Complete Execution Flow Timeline

When you run: `ansible-playbook test_callback.yml`

Step 1: Ansible startup

- Reads ansible.cfg
- Finds `callback_whitelist = simple_logger`

- Looks for callback_plugins/simple_logger.py

Step 2: Plugin loading

- Imports the file
- Finds CallbackModule class
- Checks CALLBACK_VERSION, CALLBACK_TYPE, CALLBACK_NAME

Step 3: Plugin initialization

- Calls `__init__()` method
- Sets log_file = '/tmp/ansible_playbook.log'
- Initializes counters: task_count=0, task_ok=0, task_failed=0

Step 4: Playbook starts

- Ansible calls `v2_playbook_on_start(playbook)`
- Writes "PLAYBOOK STARTED" to log

Step 5: First task begins

- Ansible calls `v2_playbook_on_task_start(task, is_conditional)`
- task_count becomes 1
- Writes "TASK 1: Create test directory" to log

Step 6: Task executes on host

- Ansible runs the actual task (creates directory)

Step 7: Task completes

- If success: Ansible calls `v2_runner_on_ok(result)`
  - task_ok becomes 1
  - Writes "✓ SUCCESS on localhost: Create test directory"
- If failure: Ansible calls `v2_runner_on_failed(result)`
  - task_failed becomes 1
  - Writes "✗ FAILED on localhost: Create test directory"

Step 8: Repeat steps 5-7 for each task

- Task 2: task_count=2, then v2_runner_on_ok, task_ok=2
- Task 3: task_count=3, then v2_runner_on_ok, task_ok=3

- Task 4: task_count=4, then v2_runner_on_ok, task_ok=4
- Task 5: task_count=5, then v2_runner_on_ok, task_ok=5

Step 9: Playbook ends

- Ansible calls `v2_playbook_on_stats(stats)`
- Writes summary: "SUMMARY: Total=5, OK=5, Failed=0"
- Writes "PLAYBOOK FINISHED"
- Prints log file location to console

Step 10: Plugin cleanup

- Python garbage collection destroys the CallbackModule instance
- File handles are closed
- Plugin execution complete

Key Concepts

1. Hook Methods

- Methods starting with `v2_` are HOOKS
- Ansible automatically calls these at specific events
- You don't call them - Ansible does

2. Instance Variables

- Variables like `self.task_count` persist throughout playbook
- Accessible in all methods of the class
- Maintain state across different hook calls

3. Result Object Structure

```
result object contains:
  ├── _host (object)
  │   └── get_name() → returns hostname
  ├── _task (object)
  │   └── get_name() → returns task name
  └── _result (dict)
      └── contains task output, return codes, etc.
```

4. Execution Order

```
v2_playbook_on_start      (once)
  ↓
v2_playbook_on_task_start (per task)
  ↓
v2_runner_on_ok/failed    (per task per host)
  ↓
v2_playbook_on_task_start (next task)
  ↓
...repeat...
  ↓
v2_playbook_on_stats      (once at end)
```

5. Why Use Callbacks?

- Non-invasive: Don't modify playbooks
- Reusable: Same plugin works with any playbook
- Parallel: Multiple callbacks can run simultaneously
- Event-driven: Respond to specific execution events