

Создание собственных modules



Алексей
Метляков



Алексей Метляков

DevOps Engineer

OpenWay



Алексей Метляков

План занятия

1. [Что такое Plugin?](#)
2. [Что такое Module?](#)
3. [Как ansible исполняет Module?](#)
4. [Простейший Module](#)
5. [Когда стоит писать свой Module?](#)
6. [Совместимость Python 2 и Python 3](#)
7. [От идеи к реализации](#)
8. [Хранение и использование Module](#)
9. [Итоги](#)
10. [Домашнее задание](#)



Что такое Plugin?

Что такое Plugin?

Plugin – спроектированный на **python** код, который выполняет одно логическое действие. Все **plugin** делятся на несколько типов:

- connection
- become
- inventory
- vars
- shell
- lookup
- strategy
- modules



Что такое Module?

Что такое Module?

Module – программный код, написанный на языке программирования, основной целью которого является реализация манипуляций над входными данными, обработка этих данных на стороне исполнителя и отправка выходных данных в **json** формате.

Иными словами, **module** должен:

- принять данные на вход;
- обработать их;
- провести действия на системе;
- получить результат этих действий;
- подготовить выходные данные из сырого формата в json;
- отправить ответ в сторону ansible.

Что такое Module?

Вызвать **module** для использования можно двумя путями:

- через командную строку;
ansible localhost -m service -a "name=memcached state=restarted"
- описать в плейбуке.

```
---  
- name: Restart memcached  
  service:  
    name: memcached  
    state: restarted  
...
```

Справку по любому **modules** можно получить через команду `ansible-doc`.



Что Ansible исполняет Module?

Как Ansible исполняет Module?

- Пользователь указывает в **task** имя **module** и его параметры;
- **TaskExecutor** получает на вход имя **module**, определяет его суть (**module** или **action plugin**), передаёт в **Normal Action Plugin** полученную информацию;
- **Normal Action Plugin** выполняет основную работу на удаленном хосте, он основной её координатор:
 - Вызывает **connection plugin** для установки соединения,
 - Добавляет внутренние свойства **Ansible** к **module**,
 - Работает с остальными **plugin** для создания временных файлов,
 - Отправляет **module** и параметры на удаленный хост,
 - Обрабатывает любые исключительные случаи работы модулей.

Как Ansible исполняет Module?

- Запускается **Executor/module_common.py**. Он собирает **module** для отправки на удаленный узел, предварительно определив его тип:
 - **PowerShell** и **JSON-args modules** передаются через **Module Replacer**,
 - **New-style Python modules** собирает **Ansiballz framework**,
 - **Non-native-want-JSON**, **Binary** и **Old-style modules** отправляются на удаленный узел без сборки в том виде, в котором они есть.
- После подготовки проверяется директива **shebang** и сравнивается с тем, что возможно указано в параметрах **ansible**. Если в **ansible** отличается – подставляет значение **module**.

Полное описание типов module можно прочитать [здесь](#)

Как Ansible исполняет Module?

- В нашем случае module собирается при помощи **Ansiballz framework**:
 - Узнаёт, какие зависимости есть у **module**;
 - Создаёт **zip** с **module** и его зависимостями и шаблоном передачи параметров **module**;
 - **Zip** кодируется **base64**, упаковывается в небольшой **python script** и отправляет на удалённый хост во временное хранилище;
 - Извлекает только код **module** в это же хранилище;
 - Устанавливает **PYTHONPATH** на **zip** для поиска зависимостей;
 - Импортирует **module** как **__main__**, чтобы была возможность исполнять **python wrapper** и код **module** в одном процессе.

Как Ansible исполняет Module?

- В конце исполнения, **module** формирует **JSON** строку с результатом и отправляет её на стандартный вывод;
- The **normal action plugin** десериализует этот **JSON** в **dict** и возвращает в executor;
- **Jinja** шаблоны будут восприняты **Ansible** как обычные строки, если они помечены, как небезопасные;
 - Строки, возвращаемые через **ActionPlugin._execute_module()**, автоматически помечаются как небезопасные через **normal action plugin**;
 - В остальных случаях строки необходимо пометить самостоятельно.



Простейший Module

Простейший Module

Взглянем на структуру самого простого **module**:

- Указание пути до исполняемого бинарника через **shebang**;
- Отдельная переменная с документацией;
- Переменная с примерами использования;
- Переменная с возможными ответами **module**;
- Исполняемый код записан в функцию **main()**;
- Функция **main()** вызывается при старте исполнения **module**.



Когда стоит писать свой Module?

Когда стоит писать свой Module?


Перед тем как начать писать свой собственный **module**, документация предлагает ответить нам на четыре вопроса:

- [Существует](#) ли похожий **module**?
- Стоит ли использовать **module** вместо **action plugin**?
- Можно ли написать **role** вместо **module**?
- Стоит ли создать **collection** вместо **module**?

Когда стоит писать свой Module?

Module должен соответствовать данным требованиям:

- Каждый **module** должен иметь атомарную функциональность, должен преследовать **UNIX-философию**: делать всё хорошо;
- **Module не должен** требовать глубоких познаний в той области в которой он работает. Например, необходимы значения обязательных параметров не могут быть задокументированы - **module** лучше отклонить;
- **Module должен** обладать достаточно ёмкой логикой. Например, если он работает с API инструмента и просто отдаёт информацию пользователю - при его использовании будет слишком много логики внутри **playbook**.



Совместимость Python 2 и Python 3

Совместимость Python 2 и Python 3

Разработчики **Ansible** сознательно сохранили поддержку **Python 2**, чтобы иметь возможность управления наибольшего количества хостов.

- На стороне **control node ansible** поддерживает **python** версий **>= 3.5** и **>= 2.7**
- На стороне **managed node ansible** поддерживает **python** версий **>= 3.5** и **>= 2.6**
- **Python >= 3.5** был выбран, так как используется в **LTS** дистрибутивах
- **Python 2.6** может работать не со всеми **modules**, например **docker-py**

Совместимость Python 2 и Python 3

Разработчики **Ansible** предлагают воспользоваться [guide](#) о сохранении совместимости между разными версиями **python**. Сами разработчики используют [единую базу кода](#).

- В Python 3 строка может быть массивом байтов (как в C) или массивом слов.
- Чтобы объединять эти типы строк - их нужно предварительно преобразовывать друг к другу
- Python 2 использует эти виды как взаимозаменяемые (str для байтов и unicode для строк). Строки автоматически преобразуются, пока состоят из ASCII, иначе вызывается исключение

Совместимость Python 2 и Python 3

- **Unicode Sandwich** – стратегия, при которой строки вне работы модуля используются как массив байтов, а внутри обрабатывается, как массив слов.
- При чтении из файла, **Python 2** преобразует данные в байты, а **Python 3** может преобразовать в слова.
- Работа с именами файлов на **UNIX** тоже возвращает нас к преобразованиям, так как они хранятся в байтовых строках.
- В **Python 2** нативная строка состоит из байт, а в **Python 3** из слов.

Для решения этих задач существует два импорта:

```
from ansible.module_utils._text import to_bytes
from ansible.module_utils._text import to_text
```

Совместимость Python 2 и Python 3

- Использовать шаблон прямой совместимости

```
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type
```

- **__metaclass__ = type** – превращает все классы в классы нового стиля
- **__future__** делает следующее:
 - **absolute_import** - заставляет импорт искать зависимости в **sys.path**, игнорируя директорию с **module**
 - **division** - заставляет при делении всегда возвращать число с плавающей запятой, если нужно получить частное - `x // y`
 - **print_function** - оборачивает стандартный вывод в отдельную функцию



От идеи к реализации

От идеи к реализации

Основные советы к реализации:

- Один **module** должен содержаться в одном файле;
- Название **module** должно разделяться подчёркиванием, использование дефиса или пробела недопустимо;
- Всегда использовать **hacking/test_module.py**;
- Старайтесь минимизировать зависимости;
- Не производите запись в файлы напрямую. Лучше сделать записи во временный файл, а потом перенести их при помощи **atomic_move** функции из **ansible.module_utils.basic**;
- Избегайте создания кешей.

От идеи к реализации

Основные **советы к реализации**:

- Каждая функция должна именоваться с использованием подчёркивания в качестве разделителя;
- «Не повторяйтесь» – это лучшая философия;
- Функции должны быть лаконичными и описывать большой объём работы;
- Имя функции в полном объёме обозначает её назначение;
- Каждая функция должна иметь строку с описанием;
- У кода должна быть минимальная вложенность. Если вложенность большая, то лучше переиспользовать его в функциях.

От идеи к реализации

Основные советы к реализации:

- При обработке **URL** лучше использовать **fetch_url** или **open_url** из **ansible.module_utils.urls** вместо **urllib2**;
- Используйте **main()** для оборачивания основного кода **module**;
- Вызывайте **main()** через условие, чтобы можно было импортировать её в тесты.

От идеи к реализации

- Для начала необходимо установить зависимости, например, для ansible: **build-essential, libssl-dev, libffi-dev, python-dev**;
- Клонировать репозиторий [ansible](#) и войти в его директорию;
- Создать виртуальное окружение: **python3 -m venv venv**;
- Активировать виртуальное окружение **.venv/bin/activate**;
- Установить зависимости через **pip**;
- Запускать **env-setup** скрипт каждый раз при создании нового **shell** процесса: **. hacking/env-setup**.

Теперь окружение готово к работе. Каждый раз при желании создать начать работу необходимо выполнять: **.venv/bin/activate && . hacking/env-setup**

От идеи к реализации

- Перейти в **lib/ansible/modules**;
- Создать новый **.py** файл для модуля;
- Заполнить его [содержимым](#);
- Исправить содержимое так, чтобы **module** выполнял нужную вам функцию, в соответствии с рекомендациями **ansible**;
- Если ваш module работает только с локальным хостом:
 - Создайте JSON файл с наполнением аргументов,
 - Запустите виртуальное окружение,
 - Запустите свой module: `python -m ansible.modules.<module_name> <path_to_json>`.

От идеи к реализации

- Дополнительно рекомендуется использовать **hacking/test_module** для тестирования **modules**;
- Можно использовать дополнительные **framework** для тестирования, например **pytest**;
- Для полного цикла тестирования нужно создать простейший **playbook** из одной **task**, которая вызывает данный **module** и проверяет его работоспособность.

Если все указанные пункты пройдены: вы гениальны, а ваш **module** готов отправляться в свободное плавание, наши поздравления!



Хранение и использование Module

Хранение и использование module

Места хранения module определены самим ansible:

- Может находиться в системной директории **ansible**;
- Рядом с **role** в директории **plugins**;
- В **collection** в директории **plugins**.

Ansible Collections

Collection – способ доставки **ansible** сущностей. Структурно она может содержать внутри себя: **roles**, **playbooks**, **modules**, **tests**.

- Именованное collection подчиняется следующему правилу:
<namespace>.<collection>
- Следовательно, модуль вызывается по имени collection:
<namespace>.<collection>.<module>
- По требованиям **ansible**, любая **collection** обязана содержать в себе хотя бы один **module**
- Файл **__init__.py** - пустой файл для инициализации **namespace**
- Чтобы создать пустую коллекцию необходимо воспользоваться конструктором: **ansible-galaxy collection init**
<namespace>.<collection>

Ansible Collections

Collection можно сделать общедоступными, но так как мы рассматриваем с вами частный случай своего использования:

- **Collection** можно отправить на хранение в свой частный репозиторий;
- Для того, чтобы скачать collection на другой хост, нужно воспользоваться: **ansible-galaxy collection install -r requirements.yml** со следующим содержимым:

```
---
collections:
  - name: <url_to_repo>
    type: git
    version: <version_tag>
```

- **Collection** можно локально собрать в **.tar.gz** и передавать в таком виде: **ansible-galaxy collection build**.



Итоги

Итоги

Сегодня мы узнали, что:

- **Module** способен расширить и без того широкий арсенал работы с окружением;
- Мы можем делать как **module** для частного использования, так и общедоступные;
- **Module** легко распространяется через **collection**;
- **Ansible** предоставляет полное описание процесса разработки, тестирования и сопровождения **module**;
- **Позволяет** contribute в их набор **module**.

Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

Алексей Метляков