Валидация конфигурации или "Факты, только факты и ничего кроме фактов"

Введение

Для начала рекомендуется прочитать статью .....

О фактах

Валидацию можно описать как процесс сравнения некоторого актуального состояния реальности с эталоном. При этом, само актуальное состояние можно представить в виде **фактов**. Н-р, идёт дождь, скорость машины 90 км/ч. В случае конфигурации оборудования, будет так: маршут по умолчанию 127.0.0.1, NTP сервер 1.1.1.1 и т.д.... Для удобства валидации, хорошо бы извлекать факты из конфигурации. Этим занимается парсер фактов.

В задачу парсера входит проход конфигурации и заполнение фактов на её основе. Т.е. на вход парсер получает конфигурациюю оборудования, собранную дискавери, а на выходе некоторый набор фактов, который, после записи в систему будет доступен для использования при валидации конфигурации.

> Конечно, отсутствие извлечённых фактов не мешает нам осуществлять в конфигурации какие-либо проверки конфигурации другими инструментами (н-р регулярные выражения, совпадения строк ...), но их наличие сильно упрощает жизнь при написании сложных проверок.

Список доступных для заполнения фактов можно найти в папке **cm/facts/*.py**. Каждый файл описывает отдельную сущность. В него можно зайти и посмотреть доступные для заполнения поля:

- interface
- subinterface
- service
- ntpserver
- staticroute
- sysloghost
- system
- user
- vlan
- vrf

Сборник фактов из interafces.py

```
def __init__(self, name, description=None, admin_status=False,
             speed="auto", duplex="auto", protocols=None,
             profile=None, type=None, mac=None, default_name=None,
             aggregated_interface=None,
             **kwargs):
    super(Interface, self).__init__()
    self.name = name
    self.description = description
    self.admin_status = admin_status
    self.has_description = False
    self.speed = speed
    self.duplex = duplex
    self.protocols = protocols
    self.profile = profile
    self.type = type
    self.mac = mac
    self.default_name = default_name
    self.aggregated_interface = aggregated_interface
```

Наш парсер

И так, как же нам написать свой парсер конфигурации и получить многие знания, имея доступ только к конфигурации оборудования? Сами по себе парсеры лежат в папке **cm/parsers**. Структура подпапок напоминает структуру профилей оборудования в sa/profiles. Парсеры в НОКе можно, условно, поделить на 2 типа: использующие библиотеку pyparsing и самописные (не использующие эту библиотеку). Попробуем показать оба подхода, на примере создания парсера для Huawei.

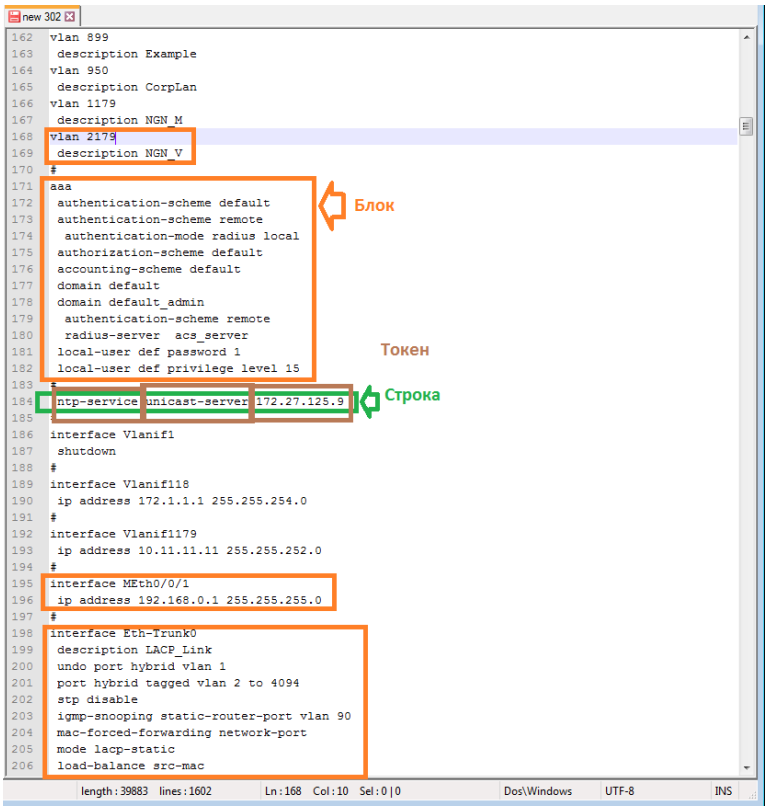
PyParsing

Представляет из себя анализатор текстов. Позволяет разбивать их на различные лексически единицы (читай слова) и дальше что-то с ними делать. Фактически, парсер, написанный с применением pyparsing и выглядит как описание структуру текста с тем, что с ними делать. Н-р на нём написан парсер для Cisco.IOS **cm/parsers/Cisco/IOS/base.py**. Для лучшего ознакомления с этим инструментом необходимо прочитать статьи:

- Парсим на Python: Pyparsing для новичков (рус.)
- Вглубь Pyparsing: парсим единицы измерения на Python (рус.)
- pyparsing quick reference: A Python text processing tool

> ⚠ Несколько важных особенностей pyparsing
>
> - По умолчанию, пробелы служат разделителями токенов, и, поэтому нет необходимости выделять их специальным образом
>
> Поезности по поводу работы pyparsing

Работа с pyparsing чем-то напоминает работу с регулярными выражениями, только здесь проще синтаксис и понятнее что мы делаем. Возьмём кусок конфига Huawei и на его примере познакомимся с терминологией:



Здесь выделены:

- Токены (tokens) - коричневый цвет. Формально, токены - это некоторые структурные единицы, из которых состоит выражение н-р, строку *ntp-service unicast-server 172.27.125.9* можно разибить на токены в следующих вариантах (разный цвет - разный токен):
  - *ntp-service unicast-server 172.27.125.9*
  - *ntp-service unicast-server 172.27.125.9*
  - и т.д
- Строки (line) - выделены зелёным. Начинаются и заканчиваются разделителем строки **\n** или начинаются с начала строки (символ ^ в регулряных выражениях) а заканчиваются концом строки (символ # в регулярных варежениях). Для работы с ними используется операторы StartLine() и EndLine()

- Выражения. Условное определение для набора токенов. Сами по себе токены могут состоять из других токенов. Для простоты будем называть это выражением (или фразой).
- Блоки (block) - выделены оранжевым. Состоят из некоторого количества выражений. Н-р. блок, описывающий интерфейсы, или ааа.

Можно сделать вывод, чты pyparser подходит для уже оформленного и структурированного текста. Парсить им неотформатированный текст может быть не простой задачей. Для такого случая может подойти один из вариантов, описанный в главе Handmade

В нашем случае текст структурирован за счёт выделения блоков отступами и разделения их знаком решётки **#**. Чем мы и воспользуемся.

Наиболее употребительные токены уже объедины в выражения и доступны для использования (импорта) из **cm/parsers/tokens.py**. Например:

- SPACE = Suppress(Word(" ").leaveWhitespace()) - Описывает пробел/s (слово Word(" ").leaveWhitespace()), который/е необходимо пропустить (слово Suppress)
- INDENT = Suppress(LineEnd() + SPACE) - Описывает конструкцию вида "\n\s+" т.е. перевод строки (LineEnd()) после которого идут пробелы (SPACE)
- REST = SPACE + restOfLine - ключевое слово restOfLine обозначает до конца строки

Для ознакомления с инструкциями и работой с текстом можно воспользоваться shell'ом. Ниже приведён пример использования конструкций pyparsing для работы с текстом.

```
from pyparsing import *
from noc.cm.parsers.tokens import INDENT, REST

#    .
# from noc.sa.models.managedobject import *
# m = ManagedObject.objects.get(name='cisco-hostname')
# config_cisco = m.config.read()



#
config_cisco = """service timestamps debug datetime
service timestamps log datetime
service password-encryption
!
hostname C2960_2
!
no logging console
"""
#       .
HOSTNAME = LineStart() + Literal("hostname") + REST.copy()
#  .searchString
HOSTNAME.searchString(config_cisco)
Out[60]: ([(['hostname', 'C2960_2'], {})], {})

#       .     ,     ,       StartLine() ..
config_huawei = """!Software Version V100R005C01SPC100
 sysname Huawe23_Stack
#
 vlan batch 2 to 91 95 to 590 592 to 596 600 to 4089
#
 stp instance 0 priority 16384
 stp enable
#
"""
#   INDENT, .. \n\s+ -
HOSTNAME = INDENT + Literal("sysname") + REST.copy()
HOSTNAME.searchString(config_huawei)
Out[13]:
([(['sysname', 'Huawe23_Stack'], {})], {})
```

После того как токены извлечены, можно начинать писать их обработку. Для этого применяются небольшие функции-методы, которым на вход поступает токен, а они устанавливают факт. Выглядят они так:

```
#   tokens      , - ['sysname', 'Huawe23_Stack'],
def on_vlan_range(self, tokens):
    #      .
    for v in ranges_to_list(tokens[0].strip()):
        self.get_vlan_fact(v)

def on_vlan_name(self, tokens):
    self.get_current_vlan().name = tokens[0]

def on_http_server(self, tokens):
    self.get_service_fact("http").enabled = tokens[0] != "undo"
```

Для вызова нужного обработчика мы используем конструкцию *.setParseAction(<обработчик>)*. При совпадениина она передаёт обработчику список токенов. Внутри обработчика мы можем проделать с токенами необходимы нам действия.

Список доступных методов для установки фактов можно посмотреть в других парсерах или в файлике **cm/parsers/base.py**.

```
VLAN_RANGE = LineStart() + Literal("vlan") + Combine(DIGITS + Word("-,") + restOfLine).setParseAction(self.
on_vlan_range)
VLAN = LineStart() + Literal("vlan") + DIGITS.copy().setParseAction(self.on_vlan)
VLAN_NAME = Literal("name") + REST.copy().setParseAction(self.on_vlan_name)


HOSTNAME = INDENT + Literal("sysname") + REST.copy().setParseAction(self.on_hostname)
```

После того, как мы научились извлекать необходимые нам факты можем смело скопировать один из готовых парсеров (н-р Cisco.IOS) и доработать его под свои нужды. Не обязательно стремиться извлечь всё что можно, достаточно только нужное для работы.

Берём наши блоки, подменяем в исходном тексте парсера и всё должно и всю конструкцию возвращаем через return. Если всё сделано без ошибок, то можно переходить в раздел Что-то пошло не так.... В нём описан запуск тестирования парсера.

Handmade

На случай, если наш текст имеет плохо формализованную структуру, в нём сложно выделить блоки и описать его для парсера, ну или просто лень со всем этим возиться... можно воспользоваться более привычными инструментами:

- Регулярные выражения
- splitlines(), if - end
- какой-нибудь ещё парсер...

Все способы связаны с тем, что к нам, фактически, прилетает просто кусок текста и способов, что с ним можно сделать, придумано вагон и маленькая тележека с обозом. Примеры парсеров, организованных таким способом, это DLink.DxS, Juniper.JUNOS.

Подход не отличается, мы ищем нужные нам сведения и вызываем обработчики, передавая им параметры.

Что-то пошло не так....

Теперь, когда парсер написан, необходимо подключить его к профилю и проверить. Подключение делается путём прописывания конструкции default_parser = "**noc.cm.parsers.Huawei.VRP.base.BaseVRPParser**" в файл **__init__.py** соотв. профиля. Для тестирования парсера удобно использоваться shell:

> Необходимо помнить, что при дискавери парсинг фактов происходит когда есть изменения в конфигурации.

```
import logging
from noc.lib.debug import error_report
from noc.cm.engine import Engine
from noc.sa.models.managedobject import ManagedObject

#   MO,
mo = ManagedObject.objects.get(name=MO_NAME)
#
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

engine = Engine(mo)
#
try:
        engine.check()
except:
    print error_report()
```

На выходе получаем огромную простыню вывода. Если в парсере есть ошибки - будет трейс. Смотрим и разбираемся что не так

Производим изменения в парсере и перезапускаем discovery чтобы изменения применились.

```
./noc ctl restart discovery-default:*
```

Примеры

В папке **cm/parsers** есть некоторое количество уже написанных парсеров. Рекомендется использовать их, для изучения и как шаблон для написания своего.