```python
# -*- coding: utf-8 -*-
from .utils import escape_filter

OPERATORS = {u'=', u'!=', u'<', u'<=', u'>=', u'>'}


def gen_filter(name, op, value, is_or=False):
    """Generates a single filter expression for ``filter[]``."""
    if op not in OPERATORS:
        raise ValueError('Unknown operator {}'.format(op))
    result = u'{}{}{}'.format(name, op, escape_filter(value))
    if is_or:
        result = u'or' + result
    return result


class Q(object):
    """A Django-like query composition class.

  Note:
    Only supports simple chaining using ``|`` and ``&`` (as ``filter[]`` supports)

  Args:
    name: Name of the value to be compared
    op: An operator
    value: Value to be matched.

  .. code-block:: python

    api.collections.vms.filter(Q('name', '=', 'foo') | Q('name', '=', 'bar'))

  Et cetera ... You can use ``|`` and ``&`` operators and they only work in chain (no parentheses)
  because that is what filter supports.

  """
    @classmethod
    def from_dict(cls, d):
        """Creates a query (AND and =) from a dictionary."""
        if not d:
            raise ValueError('Empty dictionary!')
        items = list(d.items())
        key, value = items.pop(0)
        q = cls(key, u'=', value)
        for key, value in items:
            q = q & cls(key, u'=', value)
        return q

    def __init__(self, name, op, value):
        self.name = name
        if op not in OPERATORS:
            raise ValueError('Invalid operator {}'.format(op))
        self.op = op
        self.value = value
        self.preceeding = None
        self.is_or = False

    def _set_preceeding(self, preceeding):
        self.preceeding = preceeding

    @property
    def as_filters(self):
        if self.preceeding is not None:
            filters = self.preceeding.as_filters
        else:
            filters = []

        filters.append(gen_filter(self.name, self.op, self.value, self.is_or))

        return filters

    def __or__(self, other_q):
        other_q._set_preceeding(self)
        other_q.is_or = True
        return other_q

    def __and__(self, other_q):
        other_q._set_preceeding(self)
        return other_q
```