argparse 教程

作者: Tshepang Mbambo

这篇教程旨在作为 `argparse` 的入门介绍，此模块是 Python 标准库中推荐的命令行解析模块。

备注: 标准库还包括另两个与命令行形参处理直接相关的库：低层级的 `optparse` 模块 (对于特定应用程序它可能需要更多的代码来配置，但也允许应用程序请求 `argparse` 所不支持的行为)，以及更低层级的 `getopt` (它被作为供 C 程序员使用的 `getopt()` 函数族的等价物)。 这些模块并未在本指南中直接介绍，`argparse` 中的许多核心概念最初都是来自 `optparse`，因此本教程的某些部分对 `optparse` 用户来说也是有用的。

概念

让我们利用 **ls** 命令来展示我们将要在这篇入门教程中探索的功能：

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

我们可以从这四个命令中学到几个概念：

- **ls** 是一个即使在运行的时候没有提供任何选项，也非常有用的命令。在默认情况下他会输出当前文件夹包含的文件和文件夹。
- 如果我们想要使用比它默认提供的更多功能，我们需要告诉该命令更多信息。在这个例子里，我们想要查看一个不同的目录，`pypy`。我们所做的是指定所谓的位置参数。之所以这样命名，是因为程序应该如何处理该参数值，完全取决于它在命令行出现的位置。更能体现这个概念的命令如 **cp**，它最基本的用法是 `cp SRC DEST`。第一个位置参数指的是*你想要复制的*，第二个位置参数指的是*你想要复制到的位置*。
- 现在假设我们想要改变这个程序的行为。在我们的例子中，我们不仅仅只是输出每个文件的文件名，还输出了更多信息。在这个例子中，`-l` 被称为可选参数。

- 这是一段帮助文档的文字。它是非常有用的，因为当你遇到一个你从未使用过的程序时，你可以通过阅读它的帮助文档来弄清楚它是如何运行的。

基础

让我们从一个简单到（几乎）什么也做不了的例子开始：

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

以下是该代码的运行结果：

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

options:
  -h, --help  show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

程序运行情况如下：

- 在没有任何选项的情况下运行脚本不会在标准输出显示任何内容。这没有什么用处。
- 第二行代码开始展现出 `argparse` 模块的作用。我们几乎什么也没有做，但已经得到一条很好的帮助信息。
- `--help` 选项，也可缩写为 `-h`，是唯一一个可以直接使用的选项（即不需要指定该选项的内容）。指定任何内容都会导致错误。即便如此，我们也能直接得到一条有用的用法信息。

位置参数介绍

举个例子：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

运行此程序：

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo
```

```
positional arguments:
  echo

options:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

程序运行情况如下：

- 我们增加了 `add_argument()` 方法，该方法用于指定程序将能接受哪些命令行选项。 在这个例子中，我将它命名为 `echo` 以与其对应的函数保持一致。
- 现在调用我们的程序必须要指定一个选项。
- `parse_args()` 方法实际将返回来自指定选项的某些数据，在这个例子中是 `echo`。
- 这一变量是 `argparse` 免费施放的某种“魔法”（即是说，不需要指定哪个变量是存储哪个值的）。你也可以注意到，这一名称与传递给方法的字符串参数一致，都是 `echo`。

然而请注意，尽管显示的帮助看起来清楚完整，但它可以比现在更有帮助。比如我们可以知道 `echo` 是一个位置参数，但我们除了靠猜或者看源代码，没法知道它是用来干什么的。所以，我们可以把它改造得更有用：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

然后我们得到：

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

options:
  -h, --help  show this help message and exit
```

现在，来做一些更有用的事情：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print(args.square**2)
```

以下是该代码的运行结果：

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

进展不太顺利。那是因为 argparse 会把我们传递给它的选项视作为字符串，除非我们告诉它别这样。所以，让我们来告诉 argparse 来把这一输入视为整数：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

以下是该代码的运行结果：

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

做得不错。当这个程序在收到错误的无效的输入时，它甚至能在执行计算之前先退出，还能显示很有帮助的错误信息。

可选参数介绍

到目前为止，我们一直在研究位置参数。让我们看看如何添加可选的：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

和输出：

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

options:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

程序运行情况如下：

- 这一程序被设计为当指定 `--verbosity` 选项时显示某些东西，否则不显示。
- 为表明此选项确实是可选的，当不附带该选项运行程序时将不会提示任何错误。 请注意在默认情况下，如果一个可选参数未被使用，则关联的变量，在这个例子中是 `args.verbosity`，将被赋

值为 `None`，这也就是它在 `if` 语句中无法通过真值检测的原因。
- 帮助信息有点不同。
- 使用 `--verbosity` 选项时，必须指定一个值，但可以是任何值。

上述例子接受任何整数值作为 `--verbosity` 的参数，但对于我们的简单程序而言，只有两个值有实际意义：`True` 或者 `False`。让我们据此修改代码：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

和输出：

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

options:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

程序运行情况如下：

- 现在此选项更像是一个旗标而不需要接受特定的值。 我们甚至改变了此选项的名字来匹配这一点。 请注意我们现在指定了一个新的关键词 `action`，并将其赋值为 `"store_true"`。 这意味着，如果指定了该选项，则将值 `True` 赋给 `args.verbose`。 如未指定则表示其值为 `False`。
- 当你为其指定一个值时，它会报错，符合作为标志的真正的精神。
- 留意不同的帮助文字。

短选项

如果你熟悉命令行的用法，你会发现我还没讲到这一选项的短版本。这也很简单：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

效果就像这样：

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]

options:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

可以注意到，这一新的能力也反映在帮助文本里。

结合位置参数和可选参数

我们的程序变得越来越复杂了：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print(f"the square of {args.square} equals {answer}")
else:
    print(answer)
```

接着是输出：

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- 我们带回了一个位置参数，结果发生了报错。
- 注意顺序无关紧要。

给我们的程序加上接受多个冗长度的值，然后实际来用用：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
```

```
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

和输出：

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

除了最后一个，看上去都不错。最后一个暴露了我们的程序中有一个 bug。我们可以通过限制 `--verbosity` 选项可以接受的值来修复它：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

和输出：

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

options:
  -h, --help            show this help message and exit
  -v, --verbosity {0,1,2}
                        increase output verbosity
```

注意这一改变同时反应在错误信息和帮助信息里。

现在，让我们使用另一种的方式来改变冗长度。这种方式更常见，也和 CPython 的可执行文件处理它自己的冗长度参数的方式一致（参考 `python --help` 的输出）：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

我们引入了另一种动作 "count"，来统计特定选项出现的次数。

```
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

options:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- 是的，它现在比前一版本更像是一个标志（和 `action="store_true"` 相似）。这能解释它为什么报错。
- 它也表现得与 “store_true” 的行为相似。
- 这给出了一个关于 `count` 动作的效果的演示。你之前很可能应该已经看过这种用法。
- 如果你不添加 `-v` 标志，这一标志的值会是 `None`。
- 如期望的那样，添加该标志的长形态能够获得相同的输出。
- 可惜的是，对于我们的脚本获得的新能力，我们的帮助输出并没有提供很多信息，但我们总是可以通过改善文档来修复这一问题（比如通过 `help` 关键字参数）。
- 最后一个输出暴露了我们程序中的一个 bug。

让我们修复一下：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

这是它给我们的输出：

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: '>=' not supported between instances of 'NoneType' and 'int'
```

- 第一组输出很好，修复了之前的 bug。也就是说，我们希望任何 >= 2 的值尽可能详尽。
- 第三组输出并不理想。

让我们修复那个 bug：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

我们刚刚引入了又一个新的关键字 `default`。我们把它设置为 `0` 来让它可以与其他整数值相互比较。记住，默认情况下如果一个可选参数没有被指定，它的值会是 `None`，并且它不能和整数值相比较（所以产生了 `TypeError` 异常）。

然后：

```
$ python prog.py 4
16
```

凭借我们目前已学的东西你就可以做到许多事情，而我们还仅仅学了一些皮毛而已。 argparse 模块是非常强大的，在结束本篇教程之前我们将再探索更多一些内容。

进行一些小小的改进

如果我们想扩展我们的简短程序来执行其他幂次的运算，而不仅是乘方:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"{args.x} to the power {args.y} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.x}^{args.y} == {answer}")
else:
    print(answer)
```

输出:

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                the base
  y                the exponent

options:
  -h, --help       show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

请注意到目前为止我们一直在使用详细级别来 *更改* 所显示的文本。 以下示例则使用详细级别来显示 *更多的* 文本:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"Running '{__file__}'")
if args.verbosity >= 1:
```

```
    print(f"{args.x}^{args.y} == ", end="")
print(answer)
```

输出：

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

指定有歧义的参数

当在确定一个参数是位置参数还是从属于另一个参数存在歧义时，可以使用 `--` 来告诉 parse_args() 在它之后的参数是位置参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-n', nargs='+')
>>> parser.add_argument('args', nargs='*')

>>> # ambiguous, so parse_args assumes it's an option
>>> parser.parse_args(['-f'])
usage: PROG [-h] [-n N [N ...]] [args ...]
PROG: error: unrecognized arguments: -f

>>> parser.parse_args(['--', '-f'])
Namespace(args=['-f'], n=None)

>>> # ambiguous, so the -n option greedily accepts arguments
>>> parser.parse_args(['-n', '1', '2', '3'])
Namespace(args=[], n=['1', '2', '3'])

>>> parser.parse_args(['-n', '1', '--', '2', '3'])
Namespace(args=['2', '3'], n=['1'])
```

矛盾的选项

到目前为止，我们一直在使用 argparse.ArgumentParser 实例的两个方法。 让我们再介绍第三个方法 add_mutually_exclusive_group()。 它允许我们指定彼此相冲突的选项。 让我们再修改程序的其余部分以便使新功能更有意义：我们将引入 `--quiet` 选项，它将与 `--verbose` 的作用相反:

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y
```

```
if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

我们的程序现在变得更简洁了，我们出于演示需要略去了一些功能。 无论如何，输出是这样的:

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

这应该很容易理解。 我添加了末尾的输出这样你就可以看到其所达到的灵活性，即混合使用长和短两种形式的选项。

在我们收尾之前，你也许希望告诉你的用户这个程序的主要目标，以免他们还不清楚:

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

请注意用法文本中有细微的差异。 注意 `[-v | -q]`，它的意思是说我们可以使用 `-v` 或 `-q`，但不能同时使用两者:

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent
```

```
options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

如何翻译 argparse 的输出

argparse 模块的输出例如它的帮助文本和错误消息都可以通过 gettext 模块实现翻译。 这允许应用程序轻松本地化 argparse 所产生的消息。 另请参见 国际化 (I18N) 你的程序和模块。

例如，在这个 argparse 输出中:

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

字符串 `usage:`, `positional arguments:`, `options:` 和 `show this help message and exit` 都是可翻译的。

要翻译这些字符串，必须先将它们提取到一个 `.po` 文件中。 例如，使用 Babel，运行这条命令:

```
$ pybabel extract -o messages.po /usr/lib/python3.12/argparse.py
```

此命令将从 argparse 模块提取所有可翻译的字符串，并将其输出到名为 `messages.po` 的文件中。 此命令假定你的 Python 安装位置为 `/usr/lib`。

你可以使用以下脚本查找 argparse 模块在系统中的位置:

```
import argparse
print(argparse.__file__)
```

一旦 `.po` 文件中的文本信息翻译完毕并使用 gettext 安装了译文， argparse 将能显示翻译后的信息。

要翻译在 argparse 输出中的字符串，请使用 gettext。

自定义类型转换器

argparse 模块允许您为命令行参数指定自定义类型转换器。 这使您能够在用户输入存储在 argparse.Namespace 中之前对其进行修改。 当您需要在程序中使用输入之前对其进行预处理时，

这会很有用。

使用自定义类型转换器时，您可以使用任何可调用对象，该对象接受单个字符串参数（参数值）并返回转换后的值。但是，如果需要处理更复杂的情况，可以使用带有 **action** 形参的自定义动作类。

例如，假设您希望处理带有不同前缀的参数并相应地进行处理:

```
import argparse

parser = argparse.ArgumentParser(prefix_chars='-+')

parser.add_argument('-a', metavar='<value>', action='append',
                    type=lambda x: ('-', x))
parser.add_argument('+a', metavar='<value>', action='append',
                    type=lambda x: ('+', x))

args = parser.parse_args()
print(args)
```

输出:

```
$ python prog.py -a value1 +a value2
Namespace(a=[('-', 'value1'), ('+', 'value2')])
```

在这个例子中，我们:

- 使用 `prefix_chars` 形参创建了带有自定义前缀字符的解析器 。
- 定义了两个参数 `-a` 和 `+a`, 它们使用 `type` 形参创建自定义类型转换器，以便将值存储在带有前缀的元组中。

如果没有自定义类型转换器，参数会将 `-a` 和 `+a` 视为同一个参数，这是不可取的。 通过使用自定义类型转换器，我们能够区分这两个参数。

后记

除了这里显示的内容，`argparse` 模块还提供了更多功能。 它的文档相当详细和完整，包含大量示例。 完成这个教程之后，你应该能毫不困难地阅读该文档。