argparse --- 命令行选项、参数和子命令解析器 — Python 3.9.6 文档

https://docs.python.org/zh-cn/3/library/argparse.html

None Fri Jul, 30 23:35

3.2 新版功能.

源代码： Lib/argparse.py

---

argparse 模块可以让人轻松编写用户友好的命令行接口。程序定义它需要的参数，然后 argparse 将弄清如何从 sys.argv 解析出那些参数。 argparse 模块还会自动生成帮助和使用手册，并在用户给程序传入无效参数时报出错误信息。

示例¶

以下代码是一个 Python 程序，它获取一个整数列表并计算总和或者最大值：

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

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

假设上面的 Python 代码保存在名为 `prog.py` 的文件中，它可以在命令行运行并提供有用的帮助消息：

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

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

当使用适当的参数运行时，它会输出命令行传入整数的总和或者最大值：

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

如果传入无效参数，则会报出错误：

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

以下部分将引导你完成这个示例。

创建一个解析器¶

使用 argparse 的第一步是创建一个 ArgumentParser 对象：

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

ArgumentParser 对象包含将命令行解析成 Python 数据类型所需的全部信息。

添加参数¶

给一个 ArgumentParser 添加程序参数信息是通过调用 add_argument() 方法完成的。通常，这些调用指定 ArgumentParser 如何获取命令行字符串并将其转换为对象。这些信息在 parse_args() 调用时被存储和使用。例如：

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

稍后，调用 parse_args() 将返回一个具有 `integers` 和 `accumulate` 两个属性的对象。`integers` 属性将是一个包含一个或多个整数的列表，而 `accumulate` 属性当命令行中指定了 `--sum` 参数时将是 sum() 函数，否则则是 max() 函数。

解析参数¶

ArgumentParser 通过 parse_args() 方法解析参数。它将检查命令行，把每个参数转换为适当的类型然后调用相应的操作。在大多数情况下，这意味着一个简单的 Namespace 对象将从命令行参数中解析出的属性构建：

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

在脚本中，通常 parse_args() 会被不带参数调用，而 ArgumentParser 将自动从 sys.argv 中确定命令行参数。

ArgumentParser 对象¶

class **argparse.ArgumentParser**(*prog=None, usage=None, description=None, epilog=None, parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, conflict_handler='error', add_help=True, allow_abbrev=True, exit_on_error=True*)¶

创建一个新的 ArgumentParser 对象。所有的参数都应当作为关键字参数传入。每个参数在下面都有它更详细的描述，但简而言之，它们是：

- prog - 程序的名称（默认：`sys.argv[0]`）
- usage - 描述程序用途的字符串（默认值：从添加到解析器的参数生成）
- description - 在参数帮助文档之前显示的文本（默认值：无）
- epilog - 在参数帮助文档之后显示的文本（默认值：无）
- parents - 一个 ArgumentParser 对象的列表，它们的参数也应包含在内
- formatter_class - 用于自定义帮助文档输出格式的类
- prefix_chars - 可选参数的前缀字符集合（默认值：'-'）
- fromfile_prefix_chars - 当需要从文件中读取其他参数时，用于标识文件名的前缀字符集合（默认值：`None`）
- argument_default - 参数的全局默认值（默认值：`None`）
- conflict_handler - 解决冲突选项的策略（通常是不必要的）
- add_help - 为解析器添加一个 `-h/--help` 选项（默认值：`True`）
- allow_abbrev - 如果缩写是无歧义的，则允许缩写长选项 （默认值：`True`）
- exit_on_error - 决定当错误发生时是否让 ArgumentParser 附带错误信息退出。(默认值：`True`)

在 3.5 版更改: 添加 *allow_abbrev* 参数。

在 3.8 版更改: 在之前的版本中，*allow_abbrev* 还会禁用短旗标分组，例如 `-vv` 表示为 `-v -v`。

在 3.9 版更改: 添加了 *exit_on_error* 形参。

以下部分描述这些参数如何使用。

prog¶

默认情况下，ArgumentParser 对象使用 `sys.argv[0]` 来确定如何在帮助消息中显示程序名称。这一默认值几乎总是可取的，因为它将使帮助消息与从命令行调用此程序的方式相匹配。例如，对于有如下代码的名为 `myprogram.py` 的文件：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

该程序的帮助信息将显示 `myprogram.py` 作为程序名称（无论程序从何处被调用）：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

要更改这样的默认行为，可以使用 `prog=` 参数为 ArgumentParser 提供另一个值：

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

需要注意的是，无论是从 `sys.argv[0]` 或是从 `prog=` 参数确定的程序名称，都可以在帮助消息里通过 `%(prog)s` 格式串来引用。

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage¶

默认情况下，ArgumentParser 根据它包含的参数来构建用法消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

可以通过 `usage=` 关键字参数覆盖这一默认消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

在用法消息中可以使用 `%(prog)s` 格式说明符来填入程序名称。

description¶

大多数对 ArgumentParser 构造方法的调用都会使用 `description=` 关键字参数。 这个参数简要描述这个程序做什么以及怎么做。 在帮助消息中，这个描述会显示在命令行用法字符串和各种参数的帮助消息之间:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit
```

在默认情况下，description 将被换行以便适应给定的空间。如果想改变这种行为，见 formatter_class 参数。

epilog¶

一些程序喜欢在 description 参数后显示额外的对程序的描述。这种文字能够通过给 ArgumentParser:: 提供 `epilog=` 参数而被指定。

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog='And that's how you'd foo a bar')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

和 description 参数一样， `epilog=` text 在默认情况下会换行，但是这种行为能够被调整通过提供 formatter_class 参数给 `ArgumentParse`.

parents¶

有些时候，少数解析器会使用同一系列参数。 单个解析器能够通过提供 `parents=` 参数给 ArgumentParser 而使用相同的参数而不是重复这些参数的定义。 `parents=` 参数使用 ArgumentParser 对象的列表，从它们那里收集所有的位置和可选的行为，然后将这写行为加到正在构建的 ArgumentParser 对象。

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

请注意大多数父解析器会指定 `add_help=False` . 否则， `ArgumentParse` 将会看到两个 `-h/--help` 选项（一个在父参数中一个在子参数中）并且产生一个错误。

注解

你在通过``parents=`` 传递解析器之前必须完全初始化它们。 如果你在子解析器之后改变父解析器，这些改变将不会反映在子解析器上。

formatter_class¶

ArgumentParser 对象允许通过指定备用格式化类来自定义帮助格式。目前，有四种这样的类。

class **argparse.RawDescriptionHelpFormatter** ¶

class **argparse.RawTextHelpFormatter** ¶

class **argparse.ArgumentDefaultsHelpFormatter** ¶

class **argparse.MetavarTypeHelpFormatter** ¶

RawDescriptionHelpFormatter 和 RawTextHelpFormatter 在正文的描述和展示上给与了更多的控制。ArgumentParser 对象会将 description 和 epilog 的文字在命令行中自动换行。

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

optional arguments:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

传 `RawDescriptionHelpFormatter` 给 `formatter_class=` 表示 description 和 epilog 已经被正确的格式化了，不能在命令行中被自动换行:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

`RawTextHelpFormatter` 保留所有种类文字的空格，包括参数的描述。然而，多重的新行会被替换成一行。如果你想保留多重的空白行，可以在新行之间加空格。

ArgumentDefaultsHelpFormatter 自动添加默认的值的信息到每一个帮助信息的参数中:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter 为它的值在每一个参数中使用 type 的参数名当作它的显示名（而不是使用通常的格式 dest):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars¶

许多命令行会使用 `-` 当作前缀，比如 `-f/--foo` 。如果解析器需要支持不同的或者额外的字符，比如像 `+f` 或者 `/foo` 的选项，可以在参数解析构建器中使用 `prefix_chars=` 参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

`prefix_chars=` 参数默认使用 `'-'`。提供一组不包括 `-` 的字符将导致 `-f/--foo` 选项不被允许。

fromfile_prefix_chars¶

有些时候，先举个例子，当处理一个特别长的参数列表的时候，把它存入一个文件中而不是在命令行打出来会很有意义。如果 `fromfile_prefix_chars=` 参数提供给 ArgumentParser 构造函数，之后所有类型的字符的参数都会被当成文件处理，并且会被文件包含的参数替代。举个栗子:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

从文件读取的参数在默认情况下必须一个一行（但是可参见 convert_arg_line_to_args()）并且它们被视为与命令行上的原始文件引用参数位于同一位置。所以在以上例子中，`['-f', 'foo', '@args.txt']` 的表示和 `['-f', 'foo', '-f', 'bar']` 的表示相同。

`fromfile_prefix_chars=` 参数默认为 `None`，意味着参数不会被当作文件对待。

argument_default¶

一般情况下，参数默认会通过设置一个默认到 add_argument() 或者调用带一组指定键值对的 ArgumentParser.set_defaults() 方法。但是有些时候，为参数指定一个普遍适用的解析器会更有用。这能够通过传输 `argument_default=` 关键词参数给 ArgumentParser 来完成。举个栗子，要全局禁止在 parse_args() 中创建属性，我们提供 `argument_default=SUPPRESS`：

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev¶

正常情况下，当你向 ArgumentParser 的 parse_args() 方法传入一个参数列表时，它会 recognizes abbreviations。

这个特性可以设置 `allow_abbrev` 为 `False` 来关闭:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

3.5 新版功能.

conflict_handler¶

ArgumentParser 对象不允许在相同选项字符串下有两种行为。默认情况下， ArgumentParser 对象会产生一个异常如果去创建一个正在使用的选项字符串参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

有些时候（例如：使用 parents），重写旧的有相同选项字符串的参数会更有用。为了产生这种行为， `'resolve'` 值可以提供给 ArgumentParser 的 `conflict_handler=` 参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

注意 ArgumentParser 对象只能移除一个行为如果它所有的选项字符串都被重写。所以，在上面的例子中，旧的 `-f/--foo` 行为 回合 `-f` 行为保持一样, 因为只有 `--foo` 选项字符串被重写。

add_help¶

默认情况下，ArgumentParser 对象添加一个简单的显示解析器帮助信息的选项。举个栗子，考虑一个名为 `myprogram.py` 的文件包含如下代码:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

如果 `-h` or `--help` 在命令行中被提供，参数解析器帮助信息会打印：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

有时候可能会需要关闭额外的帮助信息。这可以通过在 ArgumentParser 中设置 `add_help=` 参数为 `False` 来实现。

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
 --foo FOO  foo help
```

帮助选项一般为 `-h/--help` 。如果 `prefix_chars=` 被指定并且没有包含 `-` 字符，在这种情况下， `-h` `--help` 不是有效的选项。此时， `prefix_chars` 的第一个字符将用作帮助选项的前缀。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

exit_on_error¶

正常情况下，当你向 ArgumentParser 的 parse_args() 方法传入一个无效的参数列表时，它将会退出并发出错误信息。

如果用户想要手动捕获错误，可通过将 `exit_on_error` 设为 `False` 来启用该特性：

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None, const=None,
default=None, type=<class 'int'>, choices=None, help=None, metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

3.9 新版功能.

add_argument() 方法¶

`ArgumentParser.add_argument`(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])¶

定义单个的命令行参数应当如何解析。每个形参都在下面有它自己更多的描述，长话短说有：

- name or flags - 一个命名或者一个选项字符串的列表，例如 `foo` 或 `-f, --foo`。
- action - 当参数在命令行中出现时使用的动作基本类型。
- nargs - 命令行参数应当消耗的数目。
- const - 被一些 action 和 nargs 选择所需求的常数。
- default - 当参数未在命令行中出现并且也不存在于命名空间对象时所产生的值。
- type - 命令行参数应当被转换成的类型。
- choices - 可用的参数的容器。
- required - 此命令行选项是否可省略 （仅选项可用）。
- help - 一个此选项作用的简单描述。
- metavar - 在使用方法消息中使用的参数值示例。
- dest - 被添加到 `parse_args()` 所返回对象上的属性名。

以下部分描述这些参数如何使用。

name or flags¶

add_argument() 方法必须知道它是否是一个选项，例如 -f 或 --foo，或是一个位置参数，例如一组文件名。第一个传递给 add_argument() 的参数必须是一系列 flags 或者是一个简单的参数名。例如，可以选项可以被这样创建:

```
>>> parser.add_argument('-f', '--foo')
```

而位置参数可以这么创建:

```
>>> parser.add_argument('bar')
```

当 parse_args() 被调用，选项会以 - 前缀识别，剩下的参数则会被假定为位置参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

action¶

ArgumentParser 对象将命令行参数与动作相关联。这些动作可以做与它们相关联的命令行参数的任何事，尽管大多数动作只是简单的向 parse_args() 返回的对象上添加属性。 action 命名参数指定了这个命令行参数应当如何处理。供应的动作有:

- 'store' - 存储参数的值。这是默认的动作。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo')
  >>> parser.parse_args('--foo 1'.split())
  Namespace(foo='1')
  ```

- 'store_const' - 存储被 const 命名参数指定的值。 'store_const' 动作通常用在选项中来指定一些标志。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const', const=42)
  >>> parser.parse_args(['--foo'])
  Namespace(foo=42)
  ```

- `'store_true'` and `'store_false'` - 这些是 `'store_const'` 分别用作存储 `True` 和 `False` 值的特殊用例。另外，它们的默认值分别为 `False` 和 `True`。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- `'append'` - 存储一个列表，并且将每个参数值追加到列表中。在允许多次使用选项时很有用。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- `'append_const'` - 这存储一个列表，并将 const 命名参数指定的值追加到列表中。（注意 const 命名参数默认为 `None`。）```'append_const''' 动作一般在多个参数需要在同一列表中存储常数时会有用。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- `'count'` - 计算一个关键字参数出现的数目或次数。例如，对于一个增长的详情等级来说有用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

请注意，*default* 将为 `None`，除非显式地设为 *0*。

- `'help'` - 打印所有当前解析器中的选项和参数的完整帮助信息，然后退出。默认情况下，一个 help 动作会被自动加入解析器。关于输出是如何创建的，参与 `ArgumentParser`。

- `'version'` - 期望有一个 `version=` 命名参数在 `add_argument()` 调用中，并打印版本信息并在调用后退出:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

- `'extend'` - 这会存储一个列表，并将每个参数值加入到列表中。 示例用法:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='extend', nargs='+', type=str)
>>> parser.parse_args(['--foo', 'f1', '--foo', 'f2', 'f3', 'f4'])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

3.8 新版功能.

你还可以通过传递一个 Action 子类或实现相同接口的其他对象来指定任意操作。 `BooleanOptionalAction` 在 `argparse` 中可用并会添加对布尔型操作例如 `--foo` 和 `--no-foo` 的支持:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

3.9 新版功能.

创建自定义动作的推荐方式是扩展 `Action`，重载 `__call__` 方法以及可选的 `__init__` 和 `format_usage` 方法。

一个自定义动作的例子:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError('nargs not allowed')
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

更多描述，见 Action。

nargs¶

ArgumentParser 对象通常关联一个单独的命令行参数到一个单独的被执行的动作。 `nargs` 命名参数关联不同数目的命令行参数到单一动作。支持的值有：

- `N` （一个整数）。命令行中的 `N` 个参数会被聚集到一个列表中。 例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  注意 `nargs=1` 会产生一个单元素列表。这和默认的元素本身是不同的。

- `'?'`。如果可能的话，会从命令行中消耗一个参数，并产生一个单一项。如果当前没有命令行参数，则会产生 default 值。注意，对于选项，有另外的用例 - 选项字符串出现但没有跟随命令行参数，则会产生 const 值。一些说用用例:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c', default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

`nargs='?'` 的一个更普遍用法是允许可选的输入或输出文件:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
...                     default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- `'*'`。所有当前命令行参数被聚集到一个列表中。注意通过 `nargs='*'` 来实现多个位置参数通常没有意义，但是多个选项是可能的。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`。和 `'*'` 类似，所有当前命令行参数被聚集到一个列表中。另外，当前没有至少一个命令行参数时会产生一个错误信息。例如:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

如果不提供 `nargs` 命名参数，则消耗参数的数目将被 action 决定。通常这意味着单一项目（非列表）消耗单一命令行参数。

const¶

add_argument() 的``const`` 参数用于保存不从命令行中读取但被各种 ArgumentParser 动作需求的常数值。最常用的两例为：

- 当 add_argument() 通过 action='store_const' 或 action='append_const 调用时。这些动作将 const 值添加到 parse_args() 返回的对象的属性中。在 action 的描述中查看案例。
- 当 add_argument() 通过选项（例如 -f 或 --foo）调用并且 nargs='?' 时。这会创建一个可以跟随零个或一个命令行参数的选项。当解析命令行时，如果选项后没有参数，则将用 const 代替。在 nargs 描述中查看案例。

对 'store_const' 和 'append_const' 动作， const 命名参数必须给出。对其他动作，默认为 None。

default¶

所有选项和一些位置参数可能在命令行中被忽略。add_argument() 的命名参数 default，默认值为 None，指定了在命令行参数未出现时应当使用的值。对于选项， default 值在选项未在命令行中出现时使用：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

如果目标命名空间已经有一个属性集，则 *default* 动作不会覆盖它：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

如果 default 值是一个字符串，解析器解析此值就像一个命令行参数。特别是，在将属性设置在 Namespace 的返回值之前，解析器应用任何提供的 type 转换参数。否则解析器使用原值：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

对于 nargs 等于 `?` 或 `*` 的位置参数， `default` 值在没有命令行参数出现时使用。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

提供 `default=argparse.SUPPRESS` 导致命令行参数未出现时没有属性被添加:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type¶

默认情况下，解析器会将命令行参数当作简单字符串读入。 然而，命令行字符串经常应当被解读为其他类型，例如 float 或 int。 `add_argument()` 的 `type` 关键字允许执行任何必要的类型检查和类型转换。

如果 type 关键字使用了 default 关键字，则类型转换器仅会在默认值为字符串时被应用。

传给 `type` 的参数可以是任何接受单个字符串的可调用对象。 如果函数引发了 `ArgumentTypeError`，TypeError 或 ValueError，异常会被捕获并显示经过良好格式化的错误消息。 其他异常类型则不会被处理。

普通内置类型和函数可被用作类型转换器:

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('source_file', type=open)
parser.add_argument('dest_file', type=argparse.FileType('w', encoding='latin-1'))
parser.add_argument('datapath', type=pathlib.Path)
```

用户自定义的函数也可以被使用:

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['''The Tale of Two Cities'''])
Namespace(short_title='''the-tale-of-two-citi')
```

不建议将 `bool()` 函数用作类型转换器。 它所做的只是将空字符串转为 `False` 而将非空字符串转为 `True`。 这通常不是用户所想要的。

通常，`type` 关键字是仅应被用于只会引发上述三种被支持的异常的简单转换的便捷选项。 任何具有更复杂错误处理或资源管理的转换都应当在参数被解析后由下游代码来完成。

例如，JSON 或 YAML 转换具有复杂的错误情况，要求给出比 `type` 关键字所能给出的更好的报告。 `JSONDecodeError` 将不会被良好地格式化而 `FileNotFound` 异常则完全不会被处理。

即使 `FileType` 在用于 `type` 关键字时也存在限制。 如果一个参数使用了 *FileType* 并且有一个后续参数出错，则将报告一个错误但文件并不会被自动关闭。 在此情况下，更好的做法是等待直到解析器运行完毕再使用 `with` 语句来管理文件。

对于简单地检查一组固定值的类型检查器，请考虑改用 choices 关键字。

choices¶

某些命令行参数应当从一组受限值中选择。 这可通过将一个容器对象作为 *choices* 关键字参数传给 `add_argument()` 来处理。 当执行命令行解析时，参数值将被检查，如果参数不是可接受的值之一就将显示错误消息:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

请注意 *choices* 容器包含的内容会在执行任意 type 转换之后被检查，因此 *choices* 容器中对象的类型应当与指定的 type 相匹配:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

任何容器都可作为 *choices* 值传入，因此 list 对象，set 对象以及自定义容器都是受支持的。

不建议使用 enum.Enum，因为要控制其在用法、帮助和错误消息中的外观是很困难的。

已格式化的选项会覆盖默认的 *metavar*，该值一般是派生自 *dest*。 这通常就是你所需要的，因为用户永远不会看到 *dest* 形参。 如果不想要这样的显示（或许因为有很多选择），只需指定一个显式的 metavar。

required¶

通常，argparse 模块会认为 `-f` 和 `--bar` 等旗标是指明 *可选的* 参数，它们总是可以在命令行中被忽略。 要让一个选项成为 *必需的*，则可以将 `True` 作为 `required=` 关键字参数传给 add_argument():

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

如这个例子所示，如果一个选项被标记为 `required`，则当该选项未在命令行中出现时，parse_args() 将会报告一个错误。

注解

必需的选项通常被认为是不适宜的，因为用户会预期 *options* 都是 *可选的*，因此在可能的情况下应当避免使用它们。

help¶

`help` 值是一个包含参数简短描述的字符串。 当用户请求帮助时（一般是通过在命令行中使用 `-h` 或 `--help` 的方式），这些 `help` 描述将随每个参数一同显示:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

`help` 字符串可包括各种格式描述符以避免重复使用程序名称或参数 default 等文本。有效的描述符包括程序名称 `%(prog)s` 和传给 `add_argument()` 的大部分关键字参数，例如 `%(default)s`，`%(type)s` 等等:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

optional arguments:
 -h, --help  show this help message and exit
```

由于帮助字符串支持 %-formatting，如果你希望在帮助字符串中显示 `%` 字面值，你必须将其转义为 `%%`。

argparse 支持静默特定选项的帮助，具体做法是将 `help` 的值设为 `argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

dest¶

大多数 ArgumentParser 动作会添加一些值作为 parse_args() 所返回对象的一个属性。 该属性的名称由 add_argument() 的 dest 关键字参数确定。 对于位置参数动作， dest 通常会作为 add_argument() 的第一个参数提供:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

对于可选参数动作， dest 的值通常取自选项字符串。 ArgumentParser 会通过接受第一个长选项字符串并去掉开头的 -- 字符串来生成 dest 的值。 如果没有提供长选项字符串，则 dest 将通过接受第一个短选项字符串并去掉开头的 - 字符来获得。 任何内部的 - 字符都将被转换为 _ 字符以确保字符串是有效的属性名称。 下面的例子显示了这种行为:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

dest 允许提供自定义属性名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Action 类¶

Action 类实现了 Action API，它是一个返回可调用对象的可调用对象，返回的可调用对象可处理来自命令行的参数。 任何遵循此 API 的对象均可作为 action 形参传给 add_argument() 。

***class* argparse. Action (*option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None*)¶**

Action 对象会被 ArgumentParser 用来表示解析从命令行中的一个或多个字符串中解析出单个参数所必须的信息。 Action 类必须接受两个位置参数以及传给 ArgumentParser.add_argument() 的任何关键字参数，除了 action 本身。

Action 的实例（或作为or return value of any callable to the `action` 形参的任何可调用对象的返回值）应当定义 'dest', 'option_strings', 'default', 'type', 'required', 'help' 等属性。 确保这些属性被定义的最容易方式是调用 `Action.__init__`。

Action 的实例应当为可调用对象，因此所有子类都必须重载 `__call__` 方法，该方法应当接受四个形参:

- `parser` - 包含此动作的 ArgumentParser 对象。
- `namespace` - 将由 `parse_args()` 返回的 `Namespace` 对象。 大多数动作会使用 `setattr()` 为此对象添加属性。
- `values` - 已关联的命令行参数，并提供相应的类型转换。 类型转换由 `add_argument()` 的 type 关键字参数来指定。
- `option_string` - 被用来发起调用此动作的选项字符串。 `option_string` 参数是可选的，且此参数在动作关联到位置参数时将被略去。

`__call__` 方法可以执行任意动作，但通常将基于 `dest` 和 `values` 来设置 `namespace` 的属性。

动作子类可定义 `format_usage` 方法，该方法不带参数，所返回的字符串将被用于打印程序的用法说明。 如果未提供此方法，则将使用适当的默认值。

parse_args() 方法¶

`ArgumentParser.` `parse_args` (*args=None, namespace=None*)¶

将参数字符串转换为对象并将其设为命名空间的属性。 返回带有成员的命名空间。

之前对 `add_argument()` 的调用决定了哪些对象被创建以及它们如何被赋值。 请参阅 `add_argument()` 的文档了解详情。

- args - 要解析的字符串列表。 默认值是从 `sys.argv` 获取。
- namespace - 用于获取属性的对象。 默认值是一个新的空 `Namespace` 对象。

选项值语法¶

`parse_args()` 方法支持多种指定选项值的方式（如果它接受选项的话）。 在最简单的情况下，选项和它的值是作为两个单独参数传入的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

对于长选项（名称长度超过一个字符的选项），选项和值也可以作为单个命令行参数传入，使用 `=` 分隔它们即可:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

对于短选项（长度只有一个字符的选项），选项和它的值可以拼接在一起:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

有些短选项可以使用单个 `-` 前缀来进行合并，如果仅有最后一个选项（或没有任何选项）需要值的话:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

无效的参数¶

在解析命令行时，parse_args() 会检测多种错误，包括有歧义的选项、无效的类型、无效的选项、错误的位置参数个数等等。 当遇到这种错误时，它将退出并打印出错误文本同时附带用法消息:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

包含 `-` 的参数¶

parse_args() 方法会在用户明显出错时尝试给出错误信息，但某些情况本身就存在歧义。 例如，命令行参数 `-1` 可能是尝试指定一个选项也可能是尝试提供一个位置参数。 parse_args() 方法在此会谨慎行事：位置参数只有在它们看起来像负数并且解析器中没有任何选项看起来像负数时才能以 `-` 打头。：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

如果你有必须以 `-` 打头的位置参数并且看起来不像负数，你可以插入伪参数 `'--'` 以告诉 parse_args() 在那之后的内容是一个位置参数:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

参数缩写（前缀匹配）¶

parse_args() 方法 在默认情况下 允许将长选项缩写为前缀，如果缩写无歧义（即前缀与一个特定选项相匹配）的话:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

可产生一个以上选项的参数会引发错误。 此特定可通过将 allow_abbrev 设为 `False` 来禁用。

在 `sys.argv` 以外¶

有时在 `sys.argv` 以外用 ArgumentParser 解析参数也是有用的。 这可以通过将一个字符串列表传给 `parse_args()` 来实现。 它适用于在交互提示符下进行检测:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

命名空间对象¶

class **`argparse.Namespace`** ¶

由 `parse_args()` 默认使用的简单类，可创建一个存放属性的对象并将其返回。

这个类被有意做得很简单，只是一个具有可读字符串表示形式的 `object`。 如果你更喜欢类似字典的属性视图，你可以使用标准 Python 中惯常的 `vars()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

另一个用处是让 `ArgumentParser` 为一个已存在对象而不是为一个新的 `Namespace` 对象的属性赋值。 这可以通过指定 `namespace=` 关键字参数来实现:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

其它实用工具¶

子命令¶

`ArgumentParser.` `add_subparsers`(*[title][, description][, prog][, parser_class][, action][, option_string][, dest][, required][, help][, metavar]*)¶

许多程序都会将其功能拆分为一系列子命令，例如，`svn` 程序包含的子命令有 `svn checkout`，`svn update` 和 `svn commit`。 当一个程序能执行需要多组不同种类命令行参数时这种拆分功能的方式是一个非常好的主意。 `ArgumentParser` 通过 `add_subparsers()` 方法支持创建这样的子命令。 `add_subparsers()` 方法通常不带参数地调用并返回一个特殊的动作对象。 这种对象只有一个方法 `add_parser()`，它接受一个命令名称和任意多个 `ArgumentParser` 构造器参数，并返回一个可以通常方式进行修改的 `ArgumentParser` 对象。

形参的描述

- title - 输出帮助的子解析器分组的标题；如果提供了描述则默认为 'subcommands'，否则使用位置参数的标题
- description - 输出帮助中对子解析器的描述，默认为 `None`
- prog - 将与子命令帮助一同显示的用法信息，默认为程序名称和子解析器参数之前的任何位置参数。
- parser_class - 将被用于创建子解析器实例的类，默认为当前解析器类（例如 ArgumentParser）
- action - 当此参数在命令行中出现时要执行动作的基本类型
- dest - 将被用于保存子命令名称的属性名；默认为 `None` 即不保存任何值
- required - 是否必须要提供子命令，默认为 `False`（在 3.7 中新增）
- help - 在输出帮助中的子解析器分组帮助信息，默认为 `None`

- metavar - 帮助信息中表示可用子命令的字符串；默认为 `None` 并以 {cmd1, cmd2, ..} 的形式表示子命令

一些使用示例:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the 'a' command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the 'b' command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

请注意 `parse_args()` 返回的对象将只包含主解析器和由命令行所选择的子解析器的属性（而没有任何其他子解析器）。 因此在上面的例子中，当指定了 `a` 命令时，将只存在 `foo` 和 `bar` 属性，而当指定了 `b` 命令时，则只存在 `foo` 和 `baz` 属性。

类似地，当从一个子解析器请求帮助消息时，只有该特定解析器的帮助消息会被打印出来。 帮助消息将不包括父解析器或同级解析器的消息。 （每个子解析器命令一条帮助消息，但是，也可以像上面那样通过提供 `help=` 参数给 `add_parser()` 来给出。）

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

add_subparsers() 方法也支持 `title` 和 `description` 关键字参数。 当两者都存在时，子解析器的命令将出现在输出帮助消息中它们自己的分组内。 例如:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

此外，`add_parser` 还支持附加的 `aliases` 参数，它允许多个字符串指向同一子解析器。 这个例子类似于 `svn`，将别名 `co` 设为 `checkout` 的缩写形式:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

一个特别有效的处理子命令的方式是将 [add_subparsers()](#) 方法与对 [set_defaults()](#) 的调用结合起来使用，这样每个子解析器就能知道应当执行哪个 Python 函数。 例如:

```
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the 'foo' command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the 'bar' command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

通过这种方式，你可以在参数解析结束后让 parse_args() 执行调用适当函数的任务。像这样将函数关联到动作通常是你处理每个子解析器的不同动作的最简便方式。但是，如果有必要检查被发起调用的子解析器的名称，则 add_subparsers() 调用的 dest 关键字参数将可实现:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

在 3.7 版更改: 新增 *required* 关键字参数。

FileType 对象¶

class **argparse.FileType**(*mode='r', bufsize=-1, encoding=None, errors=None*)¶

FileType 工厂类用于创建可作为 ArgumentParser.add_argument() 的 type 参数传入的对象。以 FileType 对象作为其类型的参数将使用命令行参数以所请求模式、缓冲区大小、编码格式和错误处理方式打开文件（请参阅 open() 函数了解详情）:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=<_io.FileIO
name='raw.dat' mode='wb'>)
```

FileType 对象能理解伪参数 '-' 并会自动将其转换为 sys.stdin 用于可读的 FileType 对象，或是 sys.stdout 用于可写的 FileType 对象:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

3.4 新版功能: *encodings* 和 *errors* 关键字参数。

参数组¶

ArgumentParser.add_argument_group(*title=None, description=None*)¶

在默认情况下，ArgumentParser 会在显示帮助消息时将命令行参数分为“位置参数”和“可选参数”两组。 当存在比默认更好的参数分组概念时，可以使用 add_argument_group() 方法来创建适当的分组:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

add_argument_group() 方法返回一个具有 add_argument() 方法的参数分组对象，这与常规的 ArgumentParser 一样。 当一个参数被加入分组时，解析器会将它视为一个正常的参数，但是会在不同的帮助消息分组中显示该参数。 add_argument_group() 方法接受 *title* 和 *description* 参数，它们可被用来定制显示内容:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

请注意任意不在你的自定义分组中的参数最终都将回到通常的“位置参数”和“可选参数”分组中。

互斥¶

ArgumentParser. **add_mutually_exclusive_group** (*required=False*)¶

创建一个互斥组。 argparse 将会确保互斥组中只有一个参数在命令行中可用:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

add_mutually_exclusive_group() 方法也接受一个 *required* 参数，表示在互斥组中至少有一个参数是需要的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

注意，目前互斥参数组不支持 add_argument_group() 的 *title* 和 *description* 参数。

解析器默认值¶

ArgumentParser.set_defaults(kwargs)¶**

在大多数时候，parse_args() 所返回对象的属性将完全通过检查命令行参数和参数动作来确定。 set_defaults() 则允许加入一些无须任何命令行检查的额外属性:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

请注意解析器层级的默认值总是会覆盖参数层级的默认值:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

解析器层级默认值在需要多解析器时会特别有用。 请参阅 add_subparsers() 方法了解此类型的一个示例。

ArgumentParser. **get_default** (*dest*)¶

获取一个命名空间属性的默认值，该值是由 add_argument() 或 set_defaults() 设置的:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

打印帮助¶

在大多数典型应用中，parse_args() 将负责任何用法和错误消息的格式化和打印。 但是，也可使用某些其他格式化方法:

ArgumentParser. **print_usage** (*file=None*)¶

打印一段简短描述，说明应当如何在命令行中发起调用 ArgumentParser。 如果 *file* 为 `None`，则默认使用 sys.stdout。

ArgumentParser. **print_help** (*file=None*)¶

打印一条帮助消息，包括程序用法和通过 ArgumentParser 注册的相关参数信息。 如果 *file* 为 `None`，则默认使用 sys.stdout。

还存在这些方法的几个变化形式，它们只返回字符串而不打印消息:

ArgumentParser. **format_usage** ()¶

返回一个包含简短描述的字符串，说明应当如何在命令行中发起调用 ArgumentParser。

ArgumentParser. **format_help** ()¶

反回一个包含帮助消息的字符串，包括程序用法和通过 ArgumentParser 注册的相关参数信息。

部分解析¶

ArgumentParser. **parse_known_args** (*args=None, namespace=None*)¶

有时一个脚本可能只解析部分命令行参数，而将其余的参数继续传递给另一个脚本或程序。 在这种情况下，parse_known_args() 方法会很有用处。 它的作用方式很类似 parse_args() 但区别在于当存在额外参数时它不会产生错误。 而是会返回一个由两个条目构成的元组，其中包含带成员的命名空间和剩余参数字符串的列表。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

警告

前缀匹配 规则应用于 `parse_known_args()` 。 一个选项即使只是已知选项的前缀部分解析器也能识别该选项，不会将其放入剩余参数列表。

自定义文件解析¶

ArgumentParser. **convert_arg_line_to_args** (*arg_line*)¶

从文件读取的参数（见 ArgumentParser 的 *fromfile_prefix_chars* 关键字参数）将是一行读取一个参数。 convert_arg_line_to_args() 可被重载以使用更复杂的读取方式。

此方法接受从参数文件读取的字符串形式的单个参数 *arg_line*。 它返回从该字符串解析出的参数列表。 此方法将在每次按顺序从参数文件读取一行时被调用一次。

此方法的一个有用的重载是将每个以空格分隔的单词视为一个参数。 下面的例子演示了如何实现此重载:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

退出方法¶

ArgumentParser. **exit** (*status=0, message=None*)¶

此方法将终结程序，退出时附带指定的 *status*，并且如果给出了 *message* 则会在退出前将其打印输出。 用户可重载此方法以不同方式来处理这些步骤:

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

ArgumentParser. **error** (*message*)¶

此方法将向标准错误打印包括 *message* 的用法消息并附带状态码 2 终结程序。

混合解析¶

ArgumentParser. **parse_intermixed_args** (*args=None, namespace=None*)¶

ArgumentParser. **parse_known_intermixed_args** (*args=None, namespace=None*)¶

许多 Unix 命令允许用户混用可选参数与位置参数。 parse_intermixed_args() 和 parse_known_intermixed_args() 方法均支持这种解析风格。

这些解析器并不支持所有的 argparse 特性，并且当未支持的特性被使用时将会引发异常。 特别地，子解析器， argparse.REMAINDER 以及同时包括可选与位置参数的互斥分组是不受支持的。

下面的例子显示了 parse_known_args() 与 parse_intermixed_args() 之间的差异：前者会将 ['2', '3'] 返回为未解析的参数，而后者会将所有位置参数收集至 rest 中。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

parse_known_intermixed_args() 返回由两个条目组成的元组，其中包含带成员的命名空间以及剩余参数字符串列表。 当存在任何剩余的未解析参数字符串时 parse_intermixed_args() 将引发一个错误。

3.7 新版功能.

升级 optparse 代码¶

起初，argparse 曾经尝试通过 optparse 来维持兼容性。 但是，optparse 很难透明地扩展，特别是那些为支持新的 nargs= 描述方式和更好的用法消息所需的修改。 当When most everything in optparse 中几乎所有内容都已被复制粘贴或打上补丁时，维持向下兼容看来已是不切实际的。

argparse 模块在许多方面对标准库的 optparse 模块进行了增强，包括:

- 处理位置参数。
- 支持子命令。

- 允许替代选项前缀例如 `+` 和 `/`。
- 处理零个或多个以及一个或多个风格的参数。
- 生成更具信息量的用法消息。
- 提供用于定制 `type` 和 `action` 的更为简单的接口。

从 optparse 到 argparse 的部分升级路径:

- 将所有 optparse.OptionParser.add_option() 调用替换为 ArgumentParser.add_argument() 调用。
- 将 `(options, args) = parser.parse_args()` 替换为 `args = parser.parse_args()` 并为位置参数添加额外的 ArgumentParser.add_argument() 调用。 请注意之前所谓的 `options` 在 argparse 上下文中被称为 `args`。
- 通过使用 parse_intermixed_args() 而非 parse_args() 来替换 optparse.OptionParser.disable_interspersed_args()。
- 将回调动作和 `callback_*` 关键字参数替换为 `type` 或 `action` 参数。
- 将 `type` 关键字参数字符串名称替换为相应的类型对象（例如 int, float, complex 等）。
- 将 `optparse.Values` 替换为 Namespace 并将 `optparse.OptionError` 和 `optparse.OptionValueError` 替换为 `ArgumentError`。
- 将隐式参数字符串例如使用标准 Python 字典语法的 `%default` 或 `%prog` 替换为格式字符串，即 `%(default)s` 和 `%(prog)s`。
- 将 OptionParser 构造器 `version` 参数替换为对 `parser.add_argument('--version', action='version', version='<the version>')` 的调用。