运行的文件名为 code.py 注意 positional arguments 和 optional arguments 中的内容

两种输出方式

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

1 Class：argparse.ArgumentParser

argparse.ArgumentParser 参数

```
        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)
```

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。

1.1 无指定参数

无参数输入

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

#  注意 parser.parse_args 所在位置，输入格式python 文件名 -h

#Output:
#        usage: code.py [-h]

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

1.2 prog 参数

prog - 程序的名称（默认值：sys.argv[0]）

```
import argparse
parser = argparse.ArgumentParser(prog='myprogram')
args = parser.parse_args()

#Output:
#       usage: myprogram [-h],          显示了prog里的字符串

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

1.3 usage 参数

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）

```
import argparse
parser = argparse.ArgumentParser(prog='PROG', usage='\%(prog)s [option]'
    )
args = parser.parse_args()

#Output:
#        usage: PROG [options]

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

在用法消息中可以使用 (prog)s 格式说明符来填入程序名称。通过 usage= [关键字参数] 覆盖这一默认消息即-h

1.4 description 参数

description - 在参数帮助文档之前显示的文本（默认值：无）

```

import argparse
parser = argparse.ArgumentParser(description='A foo that bars')
args = parser.parse_args()

#Output:
#       usage: code.py [-h]

#       A foo that bars

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

在帮助消息中，这个描述会显示在 usage: 和 optional arguments: 之间, 新加上的。

1.5 epilog 参数

epilog - 在参数帮助文档之后显示的文本（默认值：无）

```
import argparse
parser = argparse.ArgumentParser(epilog="And that's how you'd foo a bar"
    )
parser.print_help()

#Output:
#       usage: code.py [-h]

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

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

1.6 parents 参数

parents - 一个 ArgumentParser 对象的列表，它们的参数也应包含在内

```
import argparse
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')
arg = foo_parser.parse_args(['--parent', '2', 'XXX'])
print(arg)

#Output:
#       Namespace(foo='XXX', parent=2)

import argparse
parent_parser = argparse.ArgumentParser(add_help=False)
parent_parser.add_argument('--parent')

bar_parser = argparse.ArgumentParser(parents=[parent_parser])
bar_parser.add_argument('--bar')
arg = bar_parser.parse_args(['--bar', 'YYY'])
print(arg)

#Output:
```

```
#       Namespace(bar='YYY', parent=None)

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

import argparse
#父参数
p_parser = argparse.ArgumentParser(add_help = False)
p_parser.add_argument('-f')
#子参数并使用父参数
son = argparse.ArgumentParser(parents = [p_parser])
son.add_argument('-p')
sonargs = son.parse_args()
print(sonargs)

# Input:
#       python code.py -p aa -f bb
# Output:
#       Namespace(f='bb', p='aa')
```

1.7 formatter_class 参数

1.7.1 formatter_class = argparse.RawDescriptionHelpFormatter

formatter_class=argparse.RawDescriptionHelpFormatter 表示 description 和 epilog 已经被正确的格式化了，不会在命令行中被改变输出格式:

argparse.RawDescriptionHelpFormatter

```
import argparse
import textwrap
parser = argparse.ArgumentParser\
        (
        formatter_class=argparse.RawDescriptionHelpFormatter,
        description=textwrap.dedent
            ('''
                Please do not mess up this text!
                --------------------------------
                I have indented it
                exactly the way
                I want it
            '''),
        epilog = '''
            likewise for this epilog whose whitespace will
            be cleaned up and whose words will be wrapped
            across a couple lines'''
```

```
        )
parser.print_help()

#Output:
#       usage: code.py [-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
#       likewise for this epilog whose whitespace will
#       be cleaned up and whose words will be wrapped
#       across a couple lines
```

1.7.2 formatter_class = argparse.RawTextHelpFormatter

formatter_class=argparse.RawTextHelpFormatter 保留所有种类文字空格。但是多个新行被替换为一行。需保留多个空行，请在换行符之间添加空格

argarse.RawTextHeilFormatter

```
import argparse
parser = argparse.ArgumentParser(formatter_class=argparse.
    RawTextHelpFormatter,
description='hello\n \n \n world!! \n\n\n\n I love Python')
parser.print_help()

#Output:
#       usage: code.py [-h]

#       hello


#       world!!

#       I love Python

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

1.7.3 formatter_class = argparse.ArgumentDefaultsHelpFormatter

formatter_class=ArgumentDefaultsHelpFormatter 自动将有关默认值的信息添加到每个参数帮助消息中

argarse.ArgumentDefaultsHelpFormatter

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

#Output:
#       usage: code.py [-h] [--foo FOO] [bar [bar ...]]

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

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

1.7.4 formatter_class = argparse.MetavarTypeHelpFormatter

formatter_class=MetavarTypeHelpFormatter 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)

argparse.MetavarTypeHelpFormatter

```
import argparse
parser = argparse.ArgumentParser(formatter_class=argparse.
    MetavarTypeHelpFormatter)
parser.add_argument('foo', type=int)
parser.add_argument('--bar', type=float)
parser.print_help()

#Output:
#       usage: code.py [-h] [--bar float] int

#       positional arguments:
#       int

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

1.8 prefix_chars 参数

prefix_chars= 参数默认使用'-'. 支持一系列字符，但是不包括 - ，这样会产生不被允许的 -f/–foo 选项

prefix_chars- 可选参数的前缀字符集合（默认值：'-'）

```
import argparse
parser = argparse.ArgumentParser(prefix_chars='+')
parser.add_argument('++foo', type=int)
parser.print_help()

# Output:
#        usage: code.py [+h] [++foo FOO]

#        optional arguments:
#          +h, ++help  show this help message and exit
#          ++foo FOO

import argparse
parser = argparse.ArgumentParser(prefix_chars='+')
parser.add_argument('--foo', type=int)
parser.print_help()

#Output:
#       usage: code.py [+h] --foo

#       positional arguments:
#         --foo

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

1.9 fromfile_prefix_chars 参数

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）

fromfile_prefix_chars-用于标识文件名的前缀字符集合 (默认值：None)

```
import argparse

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

```
print(arg)

#Output:
#       Namespace(f='bar')
```

当参数过多时,可以将参数放到文件中读取,例子中 parser.parse_args(['-f', 'foo', '@args.txt']) 解析时会从文件 args.txt 读取，相当于 ['-f', 'foo', '-f', 'bar']。

1.10 argument_default 参数

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。

提供 argument_default 为 argperse.SUPPRESS

```
import argparse

parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
parser.add_argument('--foo')
parser.add_argument('bar')
arg =parser.parse_args(['--foo', '1', 'BAR'])
print(arg)

#Output:
#       Namespace(bar='BAR', foo='1')
```

全局禁止在 parse_args 中创建属性

1.11 conflict_handler 参数

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，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，conflict_handler='resolve' :

conflict_handler 为'resolve' 创建正在使用的参数

```
import argparse

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()

#Output:
#       usage: PROG [-h] [-f FOO] [--foo FOO]
```

```
10
11 #       optional arguments:
12 #       -h, --help  show this help message and exit
13 #       -f FOO      old foo help
14 #       --foo FOO   new foo help
```

1.12 add_help 参数

隐藏了 -h, –help show this help message and exit 的信息

add_help 参数默认为 Ture

```
1
2 import argparse
3
4 parser = argparse.ArgumentParser(add_help=False)
5 parser.print_help()
6
7 #Output:
8 #       usage: code.py
```

1.13 allow_abbrev 参数

allow_abbrev 为 False 来关闭向 ArgumentParser 的 parse_args() 方法传入一个参数列表。无法使用 parser.parse_args。

allow_abbrev 为 False 关闭参数传递

```
1 import argparse
2
3 parser = argparse.ArgumentParser(allow_abbrev=False)
4 parser.add_argument('--foobar')
5 parser.add_argument('--foonley')
6 parser.parse_args(['--foon'])
7
8 #Output:
9 #       usage: code.py [-h] [--foobar FOOBAR] [--foonley FOONLEY]
10 #       code.py: error: unrecognized arguments: --foon
```

2 Class：argparse.ArgumentParser.add_argument

argparse.ArgumentParser.add_argument 参数

```
1         ArgumentParser.add_argument
2         (
```

```
        name or flags...
        [action]
        [nargs]
        [const]
        [default]
        [type]
        [choices]
        [required]
        [help]
        [metavar]
        [dest]
        )
```

2.1 name or flags 参数

add_argument() 方法必须知道它是否是一个选项或是一个位置参数，例如一组文件名

```

parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-f', '--foo')
parser.add_argument('bar')
parser.parse_args(['BAR'])

#Output:
#       Namespace(bar='BAR', foo=None)

import argparse
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-f', '--foo')
parser.add_argument('bar')
parser.parse_args(['BAR', '--foo', 'FOO'])

#Output:
#       Namespace(bar='BAR', foo='FOO')

import argparse
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-f', '--foo')
parser.add_argument('bar')
parser.parse_args(['--foo', 'FOO'])

#Output:
#       usage: PROG [-h] [-f FOO] bar
#       PROG: error: the following arguments are required: bar
```

2.2 action 参数

大多数动作只是简单的向 parse_args() 返回的对象上添加属性

2.2.1 action 存储默认参数

不指定 action 的值

```
1
2  import argparse
3
4  parser = argparse.ArgumentParser()
5  parser.add_argument('--foo')
6  arg = parser.parse_args('--foo 1'.split())
7  print(arg)
8
9  #Output:
10 #       Namespace(foo='1')
```

2.2.2 'store_const'

'store_const' 动作通常用来存储被 const 命名参数指定的元素。

'store_const' 为数字

```
1  import argparse
2
3  parser = argparse.ArgumentParser()
4  parser.add_argument('--foo', action='store_const', const=42)
5  arg = parser.parse_args(['--foo'])
6  print(arg)
7
8  #Output:
9  #       Namespace(foo=42)
```

2.2.3 'store_true' 和'store_false'

'store_true' 和'store_false' 分别用作存储 True 和 False 值的特殊用例

store_true 默认值 True;store_false 默认值 False

```
1  import argparse
2
3  parser = argparse.ArgumentParser()
4  parser.add_argument('--foo', action='store_true')
5  parser.add_argument('--bar', action='store_false')
6  parser.add_argument('--baz', action='store_false')
```

```
# arg = parser.parse_args('--foo'.split())
# print(arg)

#Output:
#       Namespace(bar=True, baz=True, foo=True)

arg = parser.parse_args('--foo --bar'.split())
print(arg)

#Output:
#       Namespace(bar=False, baz=True, foo=True)
```

2.2.4 'append'

存储一个列表，并且将每个参数值追加到列表中。

存储列表

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', action='append')
arg = parser.parse_args('--foo 1 --foo auifa'.split())
print(arg)

#Output:
#       parser.add_argument('--foo', action='append')
```

2.2.5 'append_const'

存储一个列表，并将 conse 命名参数指定的值追加到列表中，默认为 None。该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

参见下面的 dest 用法

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--str', dest='types',action='append_const', const=
    str )
parser.add_argument('--int1', action='append_const', const=int)
parser.add_argument('--int', dest='jkhd',action='append_const', const=
    int)
arg = parser.parse_args('--str --int --int1'.split())
print(arg)

```

```
#Output:
#       Namespace(int1=[<class 'int'>], jkhd=[<class 'int'>], types=[<
    class 'str'>])
```

2.2.6 'count'

计算一个关键字参数出现的数目或次数. 注意次数能够自己指定，也能是默认，默认为0.

```

parser = argparse.ArgumentParser()
parser.add_argument('--count', '-v',action='count', default=1)
arg = parser.parse_args(['-vvv'])
print(arg)

#Output:
#       Namespace(count=4)
```

2.2.7 '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。

加入帮助信息

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-count', '--hjsdv',action='help')
arg = parser.parse_args()
print(arg)

#Output:
#       usage: Argparse.py [-h] [--count]
#
#       optional arguments:
#           -h, --help       show this help message and exit
#           -count, --hjsdv
```

2.2.8 'version'

能够打印版本信息并在调用后退出

打印版本信息

```
import argparse

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

#Output:
#       PROG 2.0
```

2.2.9 自定义函数

还可以通过传递 Action 子类或实现相同接口的其他对象来指定任意操作。

自定义

```
import argparse

class FooAction(argparse.Action):
def __init__(self, option_strings, dest, nargs=None, **kwargs):
if nargs is not None:
raise ValueError("nargs not allowed")
super(FooAction,self).__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())
#OUTPUT:
#        Namespace(bar=None, foo=None) '1,' None
#        Namespace(bar='1,', foo=None) '2' '--foo'
print(args)
#OUTPUT:
#        Namespace(bar='1,', foo='2')
```

2.3 nargs 参数

nargs 命名参数关联不同数目的命令行参数到单一动作。

2.3.1 N(一个整数)

命令行中的 N 个参数会被聚集到一个列表中。

nargs

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', nargs=2)
parser.add_argument('bar', nargs=2)
arg = parser.parse_args('c d --foo a b'.split())
print(arg)

#OUTPUT:
#       Namespace(bar=['c', 'd'], foo=['a', 'b'])
```

2.3.2 ?

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。

允许可选的输入或输出文件

```
import argparse
import sys

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)
arg = parser.parse_args(['input.txt', 'output.txt'])
print(arg)

#OUTPUT:
#       Namespace(infile=<_io.TextIOWrapper name='input.txt' mode='r'
    encoding='cp936'>, outfile=
# <_io.TextIOWrapper name='output.txt' mode='w' encoding='cp936'>)

arg = parser.parse_args([])
print(arg)

#OUTPUT:
#       Namespace(infile=<_io.TextIOWrapper name='<stdin>' mode='r'
    encoding='utf-8'>, outfile=
# <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)
```

2.3.3 *

所有当前命令行参数被聚集到一个人列表中。

多个选项进行分配

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', nargs='*')
parser.add_argument('bar', nargs='*')
parser.add_argument('--baz', nargs='*')
args = parser.parse_args('a b --foo x y --baz 1 2'.split())
print(args)

#OUTPUT:
#       Namespace(bar=['a', 'b'], baz=['1', '2'], foo=['x', 'y'])
```

2.3.4 +

所有当前命令行参数被聚集到一个列表中。

没有命令行参数时报错

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('foo', nargs='+')
args = parser.parse_args(['a', 'b'])
print(args)

#OUTPUT:
#       Namespace(foo=['a', 'b'])

args = parser.parse_args([])
print(args)

#OUTPUT:
#       usage: Argparse.py [-h] foo [foo ...]
#       Argparse.py: error: the following arguments are required: foo
```

2.3.5 argarse.REMAINDER

剩余的命令行参数被聚集到一个列表中。

命令行功能的传递

```
import argparse

parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('--foo')
```

```
parser.add_argument('command')
parser.add_argument('args', nargs=argparse.REMAINDER)
print(parser.parse_args('--foo B cmd --arg1 XX ZZ'.split()))

#OUTPUT:
#       Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

2.4 const 参数

用于保存不从命令行中读取但被需求的常数值。

1. 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

2. 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。

const 提供默认值

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', nargs='?', const='c', default='d')
a = parser.parse_args(['--foo', 'YY'])
b = parser.parse_args(['--foo'])
c = parser.parse_args([])
print(a, b, c)

#OUTPUT:
#       Namespace(foo='YY') Namespace(foo='c') Namespace(foo='d')
```

2.5 default 参数

所有选项和一些位置参数可能在命令行中被忽略，用于指定在命令行参数未出现时应当使用的值。

常见应用 1

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', default=42)
a = parser.parse_args(['--foo', '2'])
b = parser.parse_args([])
print(a, b)

```

```
#OUTPUT:
#       Namespace(foo='2') Namespace(foo=42)
```

常见应用 2

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--length', default='10', type=int)
parser.add_argument('--width', default=10.535, type=int)
b = parser.parse_args([])
print(b)

#OUTPUT:
#       Namespace(length=10, width=10.535)
```

nargs 可以为? 或 *，注意分别。

常见运用 3

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('length', nargs='*', default=42)     #Namespace(
    length=['a']) Namespace(length=42)
parser.add_argument('width', nargs='?', default=20)     #Namespace(width
    ='a') Namespace(width=20)
a = parser.parse_args(['a'])
b = parser.parse_args([])
print(a, b)

#OUTPUT:
#       Namespace(length=['a'], width=20) Namespace(length=42, width=20)
```

常见运用 4

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', default=argparse.SUPPRESS)
a = parser.parse_args([])
b = parser.parse_args(['--foo', '1'])
print(a,b)

#OUTPUT:
#       Namespace() Namespace(foo='1')
```

2.6 type 参数

type 关键词参数允许任何的类型检查和类型转换。一般的内建类型和函数可以直接被 type 参数使用

规定类型

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('foo', type=int)
parser.add_argument('bar', type=open)
a = parser.parse_args('2 temp.txt'.split())
print(a)

#OUTPUT:
#       Namespace(bar=<_io.TextIOWrapper name='temp.txt' mode='r'
    encoding='cp936'>, foo=2)
```

调用定义对象

```
import argparse
import math

def perfect_square(string):
value = int(string)
sqrt = math.sqrt(value)
if sqrt != int(sqrt):
msg = "%r is not a perfect square" % string
raise argparse.ArgumentTypeError(msg)
return value

parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('foo', type=perfect_square)
a = parser.parse_args(['9'])
b = parser.parse_args(['7'])
print(a,b)

#OUTPUT:
#       Namespace(foo=9)
#       usage: PROG [-h] foo
#       PROG: error: argument foo: '7' is not a perfect square
```

2.7 choices 参数

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。

只能从中选择

```
import argparse

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'])
#OUTPUT:
#       usage: doors.py [-h] {1,2,3}
#       doors.py: error: argument door: invalid choice: 4 (choose from
    1, 2, 3)
```

2.8 required 参数

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，将 required=True 作为关键字参数传给 add_argument()

一般不使用，用户预期是可选的参数

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', required=True)
a = parser.parse_args(['--foo', 'BAR'])
print(a)
#Namespace(foo='BAR')
b = parser.parse_args([])
print(b)
#OUTPUT:
#       usage: Argparse.py [-h] --foo FOO
#       Argparse.py: error: the following arguments are required: --foo
```

2.9 help 参数

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:

另外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%%

显示简短描述

```
import argparse

parser = argparse.ArgumentParser(prog='frobble')
```

```
parser.add_argument('--foo', action='store_true',
help='foo the bars before frobbling')
parser.add_argument('bar', nargs='?', type=int, default=42,
help='the bar tp %(prog)s (default: %(default)s)')
parser.parse_args(['-h'])

#OUTPUT:
#       usage: frobble [-h] [--foo] [bar]
#
#       positional arguments:
#           bar         the bar tp frobble (default: 42)
#
#       optional arguments:
#           -h, --help  show this help message and exit
#           --foo       foo the bars before frobbling
```

静默特定选项的帮助

```
import argparse

parser = argparse.ArgumentParser(prog='frobble')
parser.add_argument('--foo', help=argparse.SUPPRESS)
parser.add_argument('--bar', help='one of the bars to be frobbled')
parser.print_help()

#OUTPUT:
#       usage: frobble [-h] [--bar BAR]
#
#       optional arguments:
#           -h, --help  show this help message and exit
#           --bar BAR   one of the bars to be frobbled
```

2.10 metavar 参数

可以使用 metavar 来指定一个替代名称

名称指定

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', metavar='YYY')
parser.add_argument('bar', metavar='XXX')
parser.parse_args('X --foo Y'.split())

parser.print_help()

```

```
#OUTPUT:
#       usage: Argparse.py [-h] [--foo YYY] XXX
#
#       positional arguments:
#           XXX
#
#       optional arguments:
#           -h, --help  show this help message and exit
#           --foo YYY
```

2.11 dest 参数

对于可选参数动作，dest 的值通常取自选项字符串。

允许提供自定义属性名称

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--foo', dest='bar')
a = parser.parse_args('--foo XXX'.split())
print(a)

#OUTPUT:
#       Namespace(bar='XXX')
```

3 Class：argparse.Action