Python 正则表达式

正则表达式，Regular Expression，缩写为regex、regexp、RE等。

正则表达式是文本处理极为重要的技术，用它可以对字符串按照某种规则进行检索、替换。

1970年代，Unix之父Ken Thompson将正则表达式引入到Unix中文本编辑器ed和grep命令中，由此正则表达式普及开来。

1980年后，perl语言对Henry Spencer编写的库，扩展了很多新的特性。1997年开始，Philip Hazel开发出了PCRE (Perl Compatible Regular Expressions)，它被PHP和HTTPD等工具采用。

正则表达式应用极其广泛，shell中处理文本的命令、各种高级编程语言都支持正则表达式。

参考 https://www.w3cschool.cn/regex_rmjc/

正则表达式是一个特殊的字符序列，用于判断一个字符串是否与我们所设定的字符序列是否匹配，也就是说检查一个字符串是否与某种模式匹配。

正则表达式可以包含普通或者特殊字符。绝大部分普通字符，比如 `'A'`, `'a'`, 或者 `'0'`，都是最简单的正则表达式。它们就匹配自身。你可以拼接普通字符，所以 `geektime` 匹配字符串 `'geektime'`.

一些字符，如`'|'` or `'('`，是特殊的。特殊字符要么代表普通字符的类别，要么影响它们周围的正则表达式的解释方式。正则表达式模式字符串可能不包含空字节，但可以使用`\number`符号指定空字节，例如`'\x00'`.

重复修饰符 (`*`, `+`, `?`, `{m,n}`, 等) 不能直接嵌套。这样避免了非贪婪后缀 `?` 修饰符，和其他实现中的修饰符产生的多义性。要应用一个内层重复嵌套，可以使用括号。 比如，表达式 `(?:a{6})*` 匹配6个 `'a'` 字符重复任意次数。

基本语法

元字符

metacharacter

代码	说明	举例
.	匹配除换行符外任意一个字符	.
[abc]	字符集合，只能表示一个字符位置。 匹配所包含的任意一个字符	[abc]匹配plain中的'a'
[^abc]	字符集合，只能表示一个字符位置。 匹配除去集合内字符的任意一个字符	[^abc]可以匹配plain中的'p'、'l'、'i'或者'n'
[a-z]	字符范围，也是个集合，表示一个字符位置 匹配所包含的任意一个字符	常用[A-Z] [0-9]
[^a-z]	字符范围，也是个集合，表示一个字符位置 匹配除去集合内字符的任意一个字符	
\b	匹配单词的边界	\bb 在文本中找到单词中b开头的b字符
\B	不匹配单词的边界	t\B 包含t的单词但是不以t结尾的t字符，例如write \Bb不以b开头的含有b的单词，例如able
\d	[0-9]匹配1位数字	\d
\D	[^0-9]匹配1位非数字	
\s	匹配1位空白字符，包括换行符、制表符、空格 [\f\r\n\t\v]	
\S	匹配1位非空白字符	
\w	匹配[a-zA-Z0-9_]，包括中文的字	\w
\W	匹配\w之外的字符	

转义

凡是在正则表达式中有特殊意义的符号，如果想使用它的本意，请使用\转义。
反斜杠自身，得使用\\
\r、\n还是转义后代表回车、换行

重复

代码	说明	举例
*	表示前面的正则表达式会重复0次或多次	e\w* 单词中e后面可以有非空白字符
+	表示前面的正则表达式重复至少1次	e\w+ 单词中e后面至少有一个非空白字符
?	表示前面的正则表达式会重复0次或1次	e\w? 单词中e后面至多有一个非空白字符
{n}	重复固定的n次	e\w{1} 单词中e后面只能有一个非空白字符
{n,}	重复至少n次	e\w{1,} 等价 e\w+ e\w{0,} 等价 e\w* e\w{0,1} 等价 e\w?
{n,m}	重复n到m次	e\w{1,10} 单词中e后面至少1个，至多10个非空白字符

比如在一段字符串中寻找是否含有某个字符或某些字符，通常我们使用内置函数来实现，如下：

```
# 设定一个常量
a = '极客时间|twowater|liangdianshui|网络安全|ReadingWithU'

# 判断是否有 “极客时间” 这个字符串，使用 PY 自带函数

print('是否含有“极客时间”这个字符串：{0}'.format(a.index('极客时间') > -1))
print('是否含有“极客时间”这个字符串：{0}'.format('极客时间' in a))
```

输出的结果如下：

```
是否含有“极客时间”这个字符串：True
是否含有“极客时间”这个字符串：True
```

那么，如果使用正则表达式呢?

刚刚提到过，Python 给我们提供了 re 模块来实现正则表达式的所有功能，那么我们先使用其中的一个函数：

```
re.findall(pattern, string[, flags])
```

该函数实现了在字符串中找到正则表达式所匹配的所有子串，并组成一个列表返回,具体操作如下：

```
import re

# 设定一个常量
a = '极客时间|twowater|liangdianshui|网络安全|ReadingWithU'

# 正则表达式

findall = re.findall('极客时间', a)
```

```
print(findall)

if len(findall) > 0:
    print('a 含有“极客时间”这个字符串')
else:
    print('a 不含有“极客时间”这个字符串')
```

输出的结果：

```
['极客时间']
a 含有“极客时间”这个字符串
```

从输出结果可以看到，可以实现和内置函数一样的功能，可是在这里也要强调一点，上面这个例子只是方便我们理解正则表达式，这个正则表达式的写法是毫无意义的。为什么这样说呢？

因为用 Python 自带函数就能解决的问题，我们就没必要使用正则表达式了，这样做多此一举。而且上面例子中的正则表达式设置成为了一个常量，并不是一个正则表达式的规则，正则表达式的灵魂在于规则，所以这样做意义不大。

那么正则表达式的规则怎么写呢？先不急，我们一步一步来，先来一个简单的，找出字符串中的所有小写字母。首先我们在 `findall` 函数中第一个参数写正则表达式的规则，其中 `[a-z]` 就是匹配任何小写字母，第二个参数只要填写要匹配的字符串就行了。具体如下：

```
import re

# 设定一个常量
a = '极客时间|twowater|liangdianshui|网络安全|ReadingWithU'

# 选择 a 里面的所有小写英文字母

re_findall = re.findall('[a-z]', a)

print(re_findall)
```

输出的结果：

```
['t', 'w', 'o', 'w', 'a', 't', 'e', 'r', 'l', 'i', 'a', 'n', 'g', 'd', 'i', 'a', 'n',
's', 'h', 'u', 'i', 'e', 'a', 'd', 'i', 'n', 'g', 'i', 't', 'h']
```

这样我们就拿到了字符串中的所有小写字母了。

或

代码	说明	举例
x|y	匹配x或者y	wood took foot food 使用 w|food 或者 (w|f)ood

捕获

代码	说明	举例
(pattern)	使用小括号指定一个子表达式，也叫分组 捕获后会自动分配组号**从1开始** 可以改变优先级	
\数字	匹配对应的分组	(very) \1 匹配very very，但捕获的组group是very
(?:pattern)	如果仅仅为了改变优先级，就不需要捕获分组	(?:w|f)ood 'industr(?:y|ies)等价 'industry|industries'
(?<name>exp) (?'name'exp)	命名分组捕获，但是可以通过name访问分组 Python语法必须是(?P<name>exp)	

字符集

字符集是由一对方括号 “[]” 括起来的字符集合。使用字符集，可以匹配多个字符中的一个。

举个例子，比如你使用 `C[ET]O` 匹配到的是 CEO 或 CTO ，也就是说 `[ET]` 代表的是一个 E 或者一个 T 。像上面提到的 `[a-z]` ,就是所有小写字母中的其中一个，这里使用了连字符 “-” 定义一个连续字符的字符范围。当然，像这种写法，里面可以包含多个字符范围的，比如： `[0-9a-fA-F]` ,匹配单个的十六进制数字，且不分大小写。注意了，字符和范围定义的先后顺序对匹配的结果是没有任何影响的。

其实说了那么多，只是想证明，字符集一对方括号 “[]” 里面的字符关系是"或（OR）"关系，下面看一个例子：

```
import re
a = 'uav,ubv,ucv,uwv,uzv,ucv,uov'

# 字符集

# 取 u 和 v 中间是 a 或 b 或 c 的字符
findall = re.findall('u[abc]v', a)
print(findall)
# 如果是连续的字母，数字可以使用 - 来代替
l = re.findall('u[a-c]v', a)
print(l)

# 取 u 和 v 中间不是 a 或 b 或 c 的字符
re_findall = re.findall('u[^abc]v', a)
print(re_findall)
```

输出的结果：

```
['uav', 'ubv', 'ucv', 'ucv']
['uav', 'ubv', 'ucv', 'ucv']
['uwv', 'uzv', 'uov']
```

在例子中，使用了取反字符集，也就是在左方括号 “[” 后面紧跟一个尖括号 “^”，就会对字符集取反。需要记住的一点是，取反字符集必须要匹配一个字符。比如：`q[^u]` 并不意味着：匹配一个 q，后面没有 u 跟着。它意味着：匹配一个 q，后面跟着一个不是 u 的字符。具体可以对比上面例子中输出的结果来理解。

我们都知道，正则表达式本身就定义了一些规则，比如 `\d`,匹配所有数字字符,其实它是等价于 [0-9]，下面也写了个例子，通过字符集的形式解释了这些特殊字符。

```
import re

a = 'uav_ubv_ucv_uwv_uzv_ucv_uov&123-456-789'

# 概括字符集

# \d 相当于 [0-9] ,匹配所有数字字符
# \D 相当于 [^0-9] ，  匹配所有非数字字符
findall1 = re.findall('\d', a)
findall2 = re.findall('[0-9]', a)
findall3 = re.findall('\D', a)
findall4 = re.findall('[^0-9]', a)
print(findall1)
print(findall2)
print(findall3)
print(findall4)

# \w 匹配包括下划线的任何单词字符，等价于 [A-Za-z0-9_]
findall5 = re.findall('\w', a)
findall6 = re.findall('[A-Za-z0-9_]', a)
print(findall5)
print(findall6)
```

输出结果：

```
['1', '2', '3', '4', '5', '6', '7', '8', '9']
['1', '2', '3', '4', '5', '6', '7', '8', '9']
['u', 'a', 'v', '_', 'u', 'b', 'v', '_', 'u', 'c', 'v', '_', 'u', 'w', 'v', '_', 'u',
'z', 'v', '_', 'u', 'c', 'v', '_', 'u', 'o', 'v', '&', '-', '-']
['u', 'a', 'v', '_', 'u', 'b', 'v', '_', 'u', 'c', 'v', '_', 'u', 'w', 'v', '_', 'u',
'z', 'v', '_', 'u', 'c', 'v', '_', 'u', 'o', 'v', '&', '-', '-']
['u', 'a', 'v', '_', 'u', 'b', 'v', '_', 'u', 'c', 'v', '_', 'u', 'w', 'v', '_', 'u',
'z', 'v', '_', 'u', 'c', 'v', '_', 'u', 'o', 'v', '1', '2', '3', '4', '5', '6', '7',
'8', '9']
['u', 'a', 'v', '_', 'u', 'b', 'v', '_', 'u', 'c', 'v', '_', 'u', 'w', 'v', '_', 'u',
'z', 'v', '_', 'u', 'c', 'v', '_', 'u', 'o', 'v', '1', '2', '3', '4', '5', '6', '7',
'8', '9']
```

数量词

数量词的词法是：{min,max} 。min 和 max 都是非负整数。如果逗号有而 max 被忽略了，则 max 没有限制。如果逗号和 max 都被忽略了，则重复 min 次。比如，`\b[1-9][0-9]{3}\b`,匹配的是 1000 ~ 9999 之间的数字(“\b” 表示单词边界），而 `\b[1-9][0-9]{2,4}\b`，匹配的是一个在 100 ~ 99999 之间的数字。

下面看一个实例，匹配出字符串中 4 到 7 个字母的英文

```
import re

a = 'java*&39android##@@python'

# 数量词

findall = re.findall('[a-z]{4,7}', a)
print(findall)
```

输出结果：

```
['java', 'android', 'python']
```

注意，这里有贪婪和非贪婪之分。那么我们先看下相关的概念：

贪婪与非贪婪

贪婪与非贪婪模式影响的是被量词修饰的子表达式的匹配行为，贪婪模式在整个表达式匹配成功的前提下，尽可能多的匹配，而非贪婪模式在整个表达式匹配成功的前提下，尽可能少的匹配。

上面例子中的就是贪婪的，如果要使用非贪婪，也就是懒惰模式，怎么用呢?

如果要使用非贪婪，则加一个 `?` ，上面的例子修改如下：

```
import re

a = 'java*&39android##@@python'

# 贪婪与非贪婪

re_findall = re.findall('[a-z]{4,7}?', a)
print(re_findall)
```

输出结果如下：

```
['java', 'andr', 'pyth']
```

从输出的结果可以看出，android 只打印除了 andr ， Python 只打印除了 pyth ， 因为这里使用的是懒惰模式。

当然，还有一些特殊字符也是可以表示数量的，比如：

> `?`：告诉引擎匹配前导字符 0 次或 1 次
>
> `+`：告诉引擎匹配前导字符 1 次或多次
>
> `*`：告诉引擎匹配前导字符 0 次或多次

把这部分的知识点总结一下,就是下面这个表了：

贪 婪	懒 惰	描 述
?	? ?	零次或一次出现，等价于{0,1}
+	+?	一次或多次出现 ， 等价于{1,}
*	*?	零次或多次出现 ， 等价于{0,}
{n}	{n}?	恰好 n 次出现
{n,m}	{n,m}?	至少 n 次枝多 m 次出现
{n,}	{n,}?	至少 n 次出现

默认是贪婪模式，也就是说尽量多匹配更长的字符串。

非贪婪很简单，在重复的符号后面加上一个?问号，就尽量的少匹配了。

代码	说明	举例
*?	匹配任意次，但尽可能少重复	
+?	匹配至少1次，，但尽可能少重复	
??	匹配0次或1次，，但尽可能少重复	
{n,}?	匹配至少n次，但尽可能少重复	
{n,m}?	匹配至少n次，至多m次，但尽可能少重复	

```
very very happy 使用v.*y和v.*?y
```

断言

零宽断言

测试字符串为wood took foot food

代码	说明	举例
(?=exp)	零宽度正预测先行断言 断言exp一定在匹配的右边出现，也就是说断言后面一定跟个exp	f(?=oo) f后面一定有oo出现
(?<=exp)	零宽度正回顾后发断言 断言exp一定出现在匹配的左边出现，也就是说前面一定有个exp前缀	(?<=f)ood、(?<=t)ook分别匹配ood、ook，ook前一定有t出现

负向零宽断言

代码	说明	举例
(?!exp)	零宽度负预测先行断言 断言exp一定不会出现在右侧，也就是说断言后面一定不是exp	\d{3}(?!\d)匹配3位数字，断言3位数字后面一定不能是数字 foo(?!d) foo后面一定不是d
(?<!exp)	零宽度负回顾后发断言 断言exp一定不能出现在左侧，也就是说断言前面一定不能是exp	(?<!f)ood ood的左边一定不是f

代码	说明	举例
(?#comment)	注释	f(?=oo)(?#这个后断言不捕获)

注意：
断言会不会捕获呢？也就是断言占不占分组号呢?
断言不占分组号。断言如同条件，只是要求匹配必须满足断言的条件。

分组和捕获是同一个意思。
使用正则表达式时，能用简单表达式，就不要复杂的表达式。

边界匹配符和组

现在介绍一些边界匹配符和组的概念。一般的边界匹配符有以下几个：

语法	描述
^	匹配字符串开头（在有多行的情况中匹配每行的开头）
$	匹配字符串的末尾(在有多行的情况中匹配每行的末尾)
\A	仅匹配字符串开头
\Z	仅匹配字符串末尾
\b	匹配 \w 和 \W 之间（单词的边界）
\B	[^\b]

分组，被括号括起来的表达式就是分组。分组表达式 `(...)` 其实就是把这部分字符作为一个整体，当然，可以有多分组的情况，每遇到一个分组，编号就会加 1 ，而且分组后面也是可以加数量词的。

re.sub

实战过程中，我们很多时候需要替换字符串中的字符，这时候就可以用到 `def sub(pattern, repl, string, count=0, flags=0)` 函数了，re.sub 共有五个参数。其中三个必选参数：pattern, repl, string ; 两个可选参数：count, flags .

具体参数意义如下：

参数	描述
pattern	表示正则中的模式字符串
repl	repl，就是replacement，被替换的字符串的意思
string	即表示要被处理，要被替换的那个 string 字符串
count	对于pattern中匹配到的结果，count可以控制对前几个group进行替换
flags	正则表达式修饰符

具体使用可以看下下面的这个实例，注释都写的很清楚的了，主要是注意一下，第二个参数是可以传递一个函数的，这也是这个方法的强大之处，例如例子里面的函数 `convert` ,对传递进来要替换的字符进行判断，替换成不同的字符。

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

import re

a = 'Python*Android*Java-888'

# 把字符串中的 * 字符替换成 & 字符
sub1 = re.sub('\*', '&', a)
print(sub1)

# 把字符串中的第一个 * 字符替换成 & 字符
sub2 = re.sub('\*', '&', a, 1)
print(sub2)


# 把字符串中的 * 字符替换成 & 字符,把字符 - 换成 |

# 1、先定义一个函数
def convert(value):
    group = value.group()
    if (group == '*'):
        return '&'
    elif (group == '-'):
        return '|'


# 第二个参数，要替换的字符可以为一个函数
sub3 = re.sub('[\*-]', convert, a)
print(sub3)
```

输出的结果：

```
Python&Android&Java-888
Python&Android*Java-888
Python&Android&Java|888
```

re.match 和 re.search

re.match 函数

语法：

```
re.match(pattern, string, flags=0)
```

re.match 尝试从字符串的**起始**位置匹配一个模式，如果不是起始位置匹配成功的话，match() 就返回 none。

```
# 使用 re.match
match = re.match('[a-zA-Z]{4,7}', a)
# group(0) 是一个完整的分组
print(match.group(0))

# 限制匹配长度为 7
match = re.match('[a-zA-Z]{7}', a)
print(match.group(0))
```

re.search 函数

语法：

```
re.search(pattern, string, flags=0)
```

re.search 扫描整个字符串并返回第一个成功的匹配。

re.match 和 re.search 的参数，基本一致的，具体描述如下：

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写

那么它们之间有什么区别呢?

re.match 只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回 None；而 re.search 匹配整个字符串，直到找到一个匹配。这就是它们之间的区别了。

可能在个人的使用中，可能更多的还是使用 `re.findall`

看下下面的实例，可以对比下 re.search 和 re.findall 的区别，还有多分组的使用。具体看下注释，对比一下输出的结果：

示例：

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

# 提取图片的地址

import re

a = '<img src="https://www.geektime.com/a8c49ef606e0e1f3ee28932usk89105e.jpg">'
```

```
# 使用 re.search
search = re.search('<img src="(.*)">', a)
# group(0) 是一个完整的分组
print(search.group(0))
print(search.group(1))

# 使用 re.findall
findall = re.findall('<img src="(.*)">', a)
print(findall)

# 多个分组的使用（比如我们需要提取 img 字段和图片地址字段）
re_search = re.search('<(.*) src="(.*)">', a)
# 打印 img
print(re_search.group(1))
# 打印图片地址
print(re_search.group(2))
# 打印 img 和图片地址，以元祖的形式
print(re_search.group(1, 2))
# 或者使用 groups
print(re_search.groups())
```

输出的结果：

```
<img src="https://www.geektime.com/a8c49ef606e0e1f3ee28932usk89105e.jpg">
https://www.geektime.com/a8c49ef606e0e1f3ee28932usk89105e.jpg
['https://www.geektime.com/a8c49ef606e0e1f3ee28932usk89105e.jpg']
img
https://www.geektime.com/a8c49ef606e0e1f3ee28932usk89105e.jpg
('img', 'https://www.geektime.com/a8c49ef606e0e1f3ee28932usk89105e.jpg')
('img', 'https://www.geektime.com/a8c49ef606e0e1f3ee28932usk89105e.jpg')
```

正则习题

IP地址

匹配合法的IP地址

```
192.168.1.150
0.0.0.0
255.255.255.255
17.16.52.100
172.16.0.100
400.400.999.888
001.022.003.000
257.257.255.256
```

提取文件名

选出含有ftp的链接，且文件类型是gz或者xz的文件名

```
ftp://ftp.astron.com/pub/file/file-5.14.tar.gz
ftp://ftp.gmplib.org/pub/gmp-5.1.2/gmp-5.1.2.tar.xz
ftp://ftp.vim.org/pub/vim/unix/vim-7.3.tar.bz2
http://anduin.linuxfromscratch.org/sources/LFS/lfs-packages/conglomeration//iana-
etc/iana-etc-2.30.tar.bz2
http://anduin.linuxfromscratch.org/sources/other/udev-lfs-205-1.tar.bz2
http://download.savannah.gnu.org/releases/libpipeline/libpipeline-1.2.4.tar.gz
http://download.savannah.gnu.org/releases/man-db/man-db-2.6.5.tar.xz
http://download.savannah.gnu.org/releases/sysvinit/sysvinit-2.88dsf.tar.bz2
http://ftp.altlinux.org/pub/people/legion/kbd/kbd-1.15.5.tar.gz
http://mirror.hust.edu.cn/gnu/autoconf/autoconf-2.69.tar.xz
http://mirror.hust.edu.cn/gnu/automake/automake-1.14.tar.xz
```

匹配邮箱地址

```
test@hot-mail.com
v-ip@geektime.com
web.manager@geektime.com.cn
super.user@google.com
a@w-a-com
```

匹配html标记

提取href中的链接url，提取文字“极客时间”

```
<a href='http://www.geektime.com/index.html' target='_blank'>极客时间</a>
```

匹配URL

```
http://www.geektime.com/index.html
https://login.geektime.com
file:///ect/sysconfig/network
```

匹配二代中国身份证ID

```
321105700101003
321105197001010030
11210020170101054X

17位数字+1位校验码组成
前6位地址码，8位出生年月，3位数字，1位校验位（0-9或X）
```

Python 网络编程

使用python发起网络请求，处理网络请求。

requests

Requests 允许你发送原始的HTTP请求，无需手工劳动。你不需要手动为 URL 添加查询字串，也不需要对 POST 数据进行表单编码。Keep-alive 和 HTTP 连接池的功能是 100% 自动化的。

安装requests

要安装 Requests，只要在你的终端中运行这个简单命令即可：

`$ pip install requests`

发送请求

使用 Requests 发送网络请求非常简单。

一开始要导入 Requests 模块：

```
>>> import requests
```

然后，尝试获取某个网页。本例子中，我们来获取 极客时间 的网站：

```
>>> r = requests.get('https://www.geektime.com')
```

现在，我们有一个名为 `r` 的 `Response` 对象。我们可以从这个对象中获取所有我们想要的信息。

Requests 简便的 API 意味着所有 HTTP 请求类型都是显而易见的。例如，你可以这样发送一个 HTTP POST 请求：

```
>>> r = requests.post('http://www.geektime.com', data = {'key':'value'})
```

那么其他 HTTP 请求类型：PUT，DELETE，HEAD 以及 OPTIONS 又是如何的呢？都是一样的简单：

```
>>> r = requests.put('http://www.geektime.com/put', data = {'key':'value'})
>>> r = requests.delete('http://www.geektime.com/delete')
>>> r = requests.head('http://www.geektime.com/get')
>>> r = requests.options('http://www.geektime.com/get')
```

使用requests 发起各种http请求方法都非常简单。

传递 URL 参数

当使用GET请求时，我们经常要传递参数，Requests 允许你使用 `params` 关键字参数，以一个字符串字典来提供这些参数。

举例来说，如果你想传递 `key1=value1` 和 `key2=value2` 到 `www.geektime.com` ，那么你可以使用如下代码：

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.get("http://www.geektime.com", params=payload)
```

通过打印输出该 URL，你能看到 URL 已被正确编码：

```
>>> print(r.url)
http://www.geektime.com?key2=value2&key1=value1
```

注意字典里值为 `None` 的键都不会被添加到 URL 的查询字符串里。

你还可以将一个列表作为值传入：

```
>>> payload = {'key1': 'value1', 'key2': ['value2', 'value3']}

>>> r = requests.get('http://www.geektime.com', params=payload)
>>> print(r.url)
http://www.geektime.com?key1=value1&key2=value2&key2=value3
```

响应内容

我们能读取服务器响应的内容。再次以 GitHub 时间线为例：

```
>>> import requests
>>> r = requests.get('https://api.github.com/events')
>>> r.text
u'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

Requests 会自动解码来自服务器的内容。大多数 unicode 字符集都能被无缝地解码。

请求发出后，Requests 会基于 HTTP 头部对响应的编码作出有根据的推测。当你访问 `r.text` 之时，Requests 会使用其推测的文本编码。你可以找出 Requests 使用了什么编码，并且能够使用 `r.encoding` 属性来改变它：

```
>>> r.encoding
'utf-8'
>>> r.encoding = 'ISO-8859-1'
```

如果你改变了编码，每当你访问 `r.text` ，Request 都将会使用 `r.encoding` 的新值。

在你需要的情况下，Requests 也可以使用定制的编码。如果你创建了自己的编码，并使用 `codecs` 模块进行注册，你就可以轻松地使用这个解码器名称作为 `r.encoding` 的值， 然后由 Requests 来为你处理编码。

JSON 响应内容

Requests 中也有一个内置的 JSON 解码器，助你处理 JSON 数据：

```
>>> import requests

>>> r = requests.get('https://api.github.com/events')
>>> r.json()
[{u'repository': {u'open_issues': 0, u'url': 'https://github.com/...
```

如果 JSON 解码失败， `r.json()` 就会抛出一个异常。例如，响应内容是 401 (Unauthorized)，尝试访问 `r.json()` 将会抛出 `ValueError: No JSON object could be decoded` 异常。

需要注意的是，成功调用 `r.json()` 并**不**意味着响应的成功。有的服务器会在失败的响应中包含一个 JSON 对象（比如 HTTP 500 的错误细节）。这种 JSON 会被解码返回。要检查请求是否成功，请使用 `r.raise_for_status()` 或者检查 `r.status_code` 是否和你的期望相同。

原始响应内容

在特殊的情况下，你可能想获取来自服务器的原始套接字响应，那么你可以访问 `r.raw`。 如果你确实想这么干，那请你确保在初始请求中设置了 `stream=True`。具体你可以这么做：

```
>>> r = requests.get('https://api.github.com/events', stream=True)
>>> r.raw
<requests.packages.urllib3.response.HTTPResponse object at 0x101194810>
>>> r.raw.read(10)
'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'
```

但一般情况下，你应该以下面的模式将文本流保存到文件：

```
with open(filename, 'wb') as fd:
    for chunk in r.iter_content(chunk_size):
        fd.write(chunk)
```

使用 `Response.iter_content` 将会处理大量你直接使用 `Response.raw` 不得不处理的。 当流下载时，上面是优先推荐的获取内容方式。 Note that `chunk_size` can be freely adjusted to a number that may better fit your use cases.

定制请求头

如果你想为请求添加 HTTP 头部，只要简单地传递一个 `dict` 给 `headers` 参数就可以了。

例如，在前一个示例中我们没有指定 content-type:

```
>>> url = 'https://api.github.com/some/endpoint'
>>> headers = {'user-agent': 'my-app/0.0.1'}

>>> r = requests.get(url, headers=headers)
```

POST 请求

通常，你想要发送一些编码为表单形式的数据——非常像一个 HTML 表单。要实现这个，只需简单地传递一个字典给 data 参数。你的数据字典在发出请求时会自动编码为表单形式：

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}

>>> r = requests.post("http://httpbin.org/post", data=payload)
>>> print(r.text)
{
  ...
  "form": {
    "key2": "value2",
    "key1": "value1"
  },
  ...
}
```

你还可以为 `data` 参数传入一个元组列表。在表单中多个元素使用同一 key 的时候，这种方式尤其有效：

```
>>> payload = (('key1', 'value1'), ('key1', 'value2'))
>>> r = requests.post('http://httpbin.org/post', data=payload)
>>> print(r.text)
{
  ...
  "form": {
    "key1": [
      "value1",
      "value2"
    ]
  },
  ...
}
```

很多时候你想要发送的数据并非编码为表单形式的。如果你传递一个 `string` 而不是一个 `dict`，那么数据会被直接发布出去。

例如，Github API v3 接受编码为 JSON 的 POST/PATCH 数据：

```
>>> import json

>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}

>>> r = requests.post(url, data=json.dumps(payload))
```

此处除了可以自行对 `dict` 进行编码，你还可以使用 `json` 参数直接传递，然后它就会被自动编码。这是 2.4.2 版的新加功能：

```
>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}

>>> r = requests.post(url, json=payload)
```

响应状态码

我们可以检测响应状态码：

```
>>> r = requests.get('http://httpbin.org/get')
>>> r.status_code
200
```

为方便引用，Requests还附带了一个内置的状态码查询对象：

```
>>> r.status_code == requests.codes.ok
True
```

响应头

我们可以查看以一个 Python 字典形式展示的服务器响应头：

```
>>> r.headers
{
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'nginx/1.0.4',
    'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json'
}
```

HTTP 头部是大小写不敏感的。

因此，我们可以使用任意大写形式来访问这些响应头字段：

```
>>> r.headers['Content-Type']
'application/json'

>>> r.headers.get('content-type')
'application/json'
```

Cookie

如果某个响应中包含一些 cookie，你可以快速访问它们：

```
>>> url = 'http://example.com/some/cookie/setting/url'
>>> r = requests.get(url)

>>> r.cookies['example_cookie_name']
'example_cookie_value'
```

要想发送你的cookies到服务器，可以使用 `cookies` 参数：

```
>>> url = 'http://httpbin.org/cookies'
>>> cookies = dict(cookies_are='working')

>>> r = requests.get(url, cookies=cookies)
>>> r.text
'{"cookies": {"cookies_are": "working"}}'
```

Cookie 的返回对象为 `RequestsCookieJar`，它的行为和字典类似，但接口更为完整，适合跨域名跨路径使用。你还可以把 Cookie Jar 传到 Requests 中：

```
>>> jar = requests.cookies.RequestsCookieJar()
>>> jar.set('tasty_cookie', 'yum', domain='httpbin.org', path='/cookies')
>>> jar.set('gross_cookie', 'blech', domain='httpbin.org', path='/elsewhere')
>>> url = 'http://httpbin.org/cookies'
>>> r = requests.get(url, cookies=jar)
>>> r.text
'{"cookies": {"tasty_cookie": "yum"}}'
```

重定向与请求历史

默认情况下，除了 HEAD, Requests 会自动处理所有重定向。

可以使用响应对象的 `history` 方法来追踪重定向。

`Response.history` 是一个 `Response` 对象的列表，为了完成请求而创建了这些对象。这个对象列表按照从最老到最近的请求进行排序。

例如，Github 将所有的 HTTP 请求重定向到 HTTPS：

```
>>> r = requests.get('http://github.com')

>>> r.url
'https://github.com/'

>>> r.status_code
200

>>> r.history
[<Response [301]>]
```

如果你使用的是GET、OPTIONS、POST、PUT、PATCH 或者 DELETE，那么你可以通过 `allow_redirects` 参数禁用重定向处理：

```
>>> r = requests.get('http://github.com', allow_redirects=False)
>>> r.status_code
301
>>> r.history
[]
```

如果你使用了 HEAD，你也可以启用重定向：

```
>>> r = requests.head('http://github.com', allow_redirects=True)
>>> r.url
'https://github.com/'
>>> r.history
[<Response [301]>]
```

超时

你可以告诉 requests 在经过以 `timeout` 参数设定的秒数时间之后停止等待响应。基本上所有的生产代码都应该使用这一参数。如果不使用，你的程序可能会永远失去响应：

```
>>> requests.get('http://github.com', timeout=0.001)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
requests.exceptions.Timeout: HTTPConnectionPool(host='github.com', port=80): Request
timed out. (timeout=0.001)
```

注意

`timeout` 仅对连接过程有效，与响应体的下载无关。 `timeout` 并不是整个下载响应的时间限制，而是如果服务器在 `timeout` 秒内没有应答，将会引发一个异常.

Python数据库编程

Python 的 MySQL 连接器模块用于将 MySQL 数据库与 Python 程序连接起来。它使用 Python 标准库并且没有依赖项。

安装模块

```
pip install mysql-connector
```

使用 MySQL-Connector Python 连接 MySQL 数据库

Python Mysql 连接器模块方法

1. connect()： 该函数用于与MySQL服务器建立连接。 以下是用于启动连接的参数：

1. **user：** 与用于验证连接的 MySQL 服务器关联的用户名
2. **密码：** 与用户名关联的密码，用于身份验证
3. **数据库：** MySQL中用于创建表的

2. cursor() : 游标是执行SQL命令时在系统内存中创建的工作空间。 该内存是临时的，并且游标连接在整个会话/生命周期内都是有界的，并且命令被执行

3. execute() ： execute 函数将 SQL 查询作为参数并执行。 查询是用于创建、插入、检索、更新、删除等的 SQL 命令。

在以下示例中，我们将使用 connect() 连接到 MySQL 数据库

```
# Python program to connect
# to mysql database


import mysql.connector


# Connecting from the server
conn = mysql.connector.connect(user = 'username',
                               host = 'localhost',
                              database = 'database_name')

print(conn)

# Disconnecting from the server
conn.close()
```

同样，我们可以使用 connection.MySQLConnection() 类而不是 connect()：

```
# Python program to connect
# to mysql database


from mysql.connector import connection

# Connecting to the server
conn = connection.MySQLConnection(user = 'username',
                              host = 'localhost',
                              database = 'database_name')

print(conn)
```

```
# Disconnecting from the server
conn.close()
```

另一种方法是使用 '**' 运算符在 connect() 函数中传递字典：

```
# Python program to connect
# to mysql database


from mysql.connector import connection


dict = {
'user': 'root',
'host': 'localhost',
'database': 'dvwa'
}

# Connecting to the server
conn = connection.MySQLConnection(**dict)

print(conn)

# Disconnecting from the server
conn.close()
```

Python MySQL – Select Query

使用查询语句

```
# importing required library
import mysql.connector

# connecting to the database
dataBase = mysql.connector.connect(
          host = "localhost",
          user = "root",
          passwd = "",
          database = "dvwa" )

# preparing a cursor object
cursorObject = dataBase.cursor()

# selecting query
query = "SELECT NAME, ROLL FROM USERS"
cursorObject.execute(query)
```

```
myresult = cursorObject.fetchall()

for x in myresult:
  print(x)

# disconnecting from server
dataBase.close()
```

Python MySQL – Update Query

使用update 语句

```
# Python program to demonstrate
# update clause


import mysql.connector

# Connecting to the Database
mydb = mysql.connector.connect(
host ='localhost',
database ='dvwa',
user ='root',
)

cs = mydb.cursor()

statement ="UPDATE USERS SET AGE = 23 WHERE Name ='Rishi Kumar'"

cs.execute(statement)
mydb.commit()

# Disconnecting from the database
mydb.close()
```

Python使用ORM框架

安装sqlalchemy

在使用sqlalchemy之前要先给python安装mysql驱动，由于我使用的是python3原来的mysqldb不可用，所以这里推荐使用pymysql。
我们通过pip进行安装,在windows下使用pip安装包的时候要记得使用管理员身份运行cmd不然有些操作是无法进行的。

```
pip install pymysql
```

安装完以后安装再安装sqlalchemy

```
pip install sqlalchemy
```

创建一个连接引擎

```
from sqlalchemy import create_engine

engine = create_engine("mysql+pymysql://root:@localhost:33060/dvwa", echo=True)
```

create_engine("数据库类型+数据库驱动://数据库用户名:数据库密码@IP地址:端口/数据库"，其他参数)
这 `echo=True` 参数表示连接发出的 SQL 将 被记录到标准输出。

根据dvwa users 表的内容，写orm model

```
mysql> desc users;
+--------------+-------------+------+-----+-------------------+-----------------------
-----+
| Field        | Type        | Null | Key | Default           | Extra
     |
+--------------+-------------+------+-----+-------------------+-----------------------
-----+
| user_id      | int(6)      | NO   | PRI | 0                 |
     |
| first_name   | varchar(15) | YES  |     | NULL              |
     |
| last_name    | varchar(15) | YES  |     | NULL              |
     |
| user         | varchar(15) | YES  |     | NULL              |
     |
| password     | varchar(32) | YES  |     | NULL              |
     |
| avatar       | varchar(70) | YES  |     | NULL              |
     |
| last_login   | timestamp   | NO   |     | CURRENT_TIMESTAMP | on update
CURRENT_TIMESTAMP |
| failed_login | int(3)      | YES  |     | NULL              |
     |
+--------------+-------------+------+-----+-------------------+-----------------------
-----+
8 rows in set (0.01 sec)
```

```
from sqlalchemy import Column
```

```
from sqlalchemy import Integer
from sqlalchemy import String
from sqlalchemy import TIMESTAMP
from sqlalchemy.orm import declarative_base

Base = declarative_base()


class Users(Base):
    __tablename__ = "users"

    user_id = Column(Integer, primary_key=True, nullable=False)
    first_name = Column(String(15), nullable=True)
    last_name = Column(String(15), nullable=True)
    user = Column(String(15), nullable=True)
    password = Column(String(32), nullable=True)
    avatar = Column(String(70), nullable=True)
    last_login = Column(TIMESTAMP(True), nullable=False)
    failed_login = Column(Integer, nullable=True)
```

使用ORM框架写查询逻辑

```
from conn.connect import engine
from sqlalchemy.orm import Session
from sqlalchemy import select
from dvwa_model import Users

session = Session(engine)

stmt = select(Users)

# session.scalars 返回可遍历对象
for user in session.scalars(stmt):
    print("user id:%s name:%s" % (user.user_id, user.first_name))

session.close()
```

使用ORM框架插入数据

```
from conn.connect import engine
from sqlalchemy.orm import Session
from sqlalchemy import select
from dvwa_model import Users

session = Session(engine)

session.add(Users(user_id=6, first_name="geektime", last_name=".com", user="mage"))

session.commit()

session.close()
```

```
mysql> select * from users;
+---------+------------+-----------+---------+----------------------------------+----------------------------------------+---------------------+--------------+
| user_id | first_name | last_name | user    | password                         | avatar                                 | last_login          | failed_login |
+---------+------------+-----------+---------+----------------------------------+----------------------------------------+---------------------+--------------+
|       1 | admin      | admin     | admin   | 14c879f3f5d8ed93a09f6090d77c2cc3 | http://127.0.0.1/hackable/users/admin.jpg    | 2022-02-19 03:44:43 |            0 |
|       2 | Gordon     | Brown     | gordonb | e99a18c428cb38d5f260853678922e03 | http://127.0.0.1/hackable/users/gordonb.jpg | 2022-01-21 14:03:18 |            0 |
|       3 | Hack       | Me        | 1337    | 8d3533d75ae2c3966d7e0d4fcc69216b | http://127.0.0.1/hackable/users/1337.jpg     | 2022-01-21 14:03:18 |            0 |
|       4 | Pablo      | Picasso   | pablo   | 0d107d09f5bbe40cade3de5c71e9e9b7 | http://127.0.0.1/hackable/users/pablo.jpg    | 2022-01-21 14:03:18 |            0 |
|       5 | Bob        | Smith     | smithy  | 5f4dcc3b5aa765d61d8327deb882cf99 | http://127.0.0.1/hackable/users/smithy.jpg   | 2022-01-21 14:03:18 |            0 |
|       6 | magedu     | .com      | mage    | NULL                             | NULL                                   | 2022-05-02 15:23:57 |         NULL |
+---------+------------+-----------+---------+----------------------------------+----------------------------------------+---------------------+--------------+
6 rows in set (0.00 sec)
```

使用ORM框架更新数据

```
from conn.connect import engine
from sqlalchemy.orm import Session
from sqlalchemy import select
from dvwa_model import Users

session = Session(engine)

stmt = select(Users).where(Users.user_id == 6)
data = session.scalars(stmt).one()
data.user = 'geektime'

session.commit()

session.close()
```

使用ORM框架删除数据

```
from conn.connect import engine
from sqlalchemy.orm import Session
from sqlalchemy import select
from dvwa_model import Users

session = Session(engine)

stmt = select(Users).where(Users.user_id == 6)
data = session.scalars(stmt).one()
```

```
session.delete(data)
session.commit()

session.close()
```

Python多线程与多进程

进程和线程

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

进程和程序的关系：程序是源代码编译后的文件，而这些文件存放在磁盘上。当程序被操作系统加载到内存中，就是进程，进程中存放着指令和数据（资源）。一个程序的执行实例就是一个进程。它也是线程的容器。

Linux进程有父进程、子进程，Windows的进程是平等关系。

在实现了线程的操作系统中，线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。

线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。
一个标准的线程由线程ID，当前指令指针(PC）、寄存器集合和堆、栈组成。

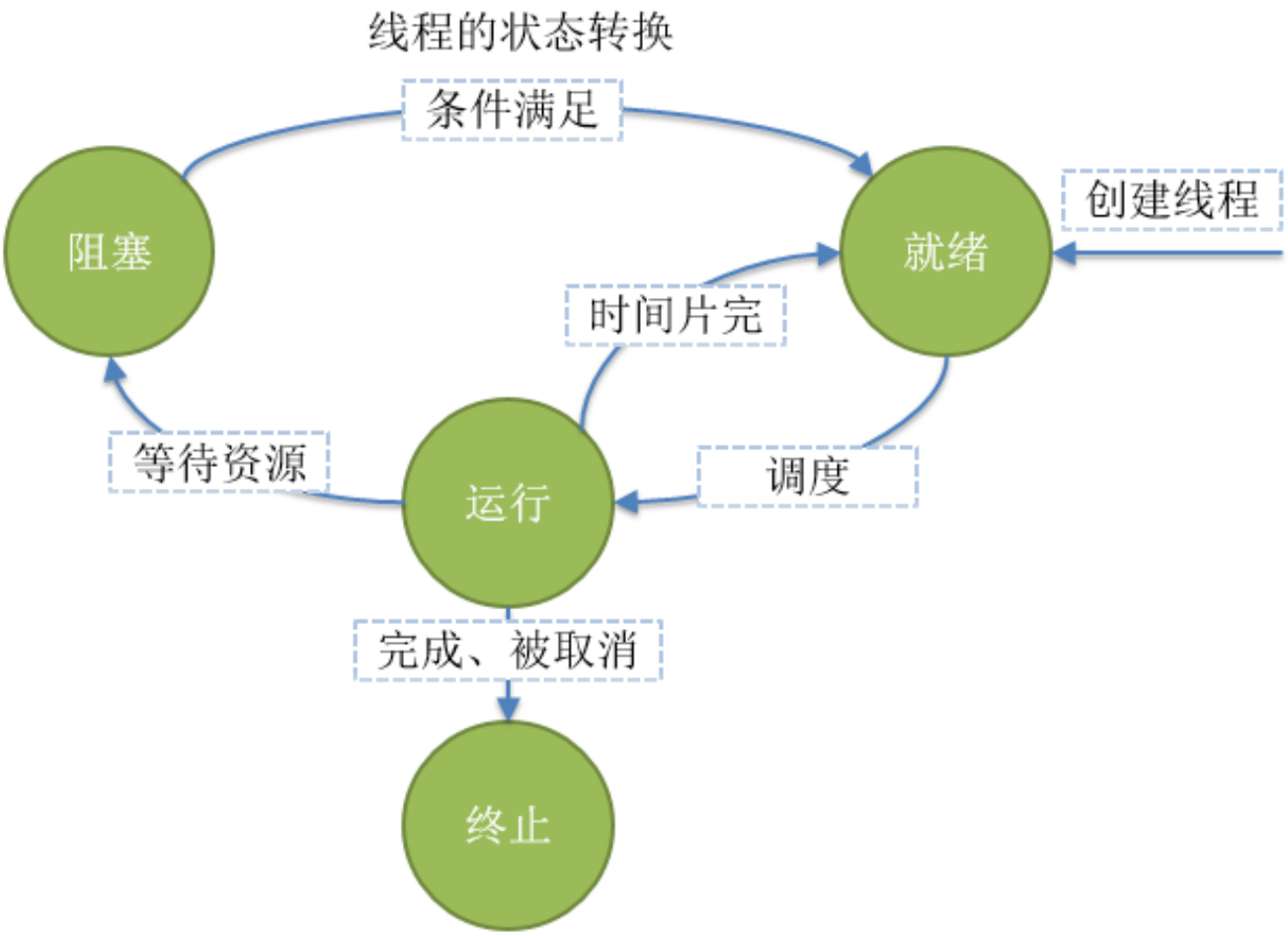
在许多系统中，创建一个线程比创建一个进程快10-100倍。

进程、线程的理解

现代操作系统提出进程的概念，每一个进程都认为自己独占所有的计算机硬件资源。
进程就是独立的王国，进程间不可以随便的共享数据。
线程就是省份，同一个进程内的线程可以共享进程的资源，每一个线程拥有自己独立的堆栈。

线程的状态

状态	含义
就绪(Ready)	线程能够运行，但在等待被调度。可能线程刚刚创建启动，或刚刚从阻塞中恢复，或者被其他线程抢占
运行(Running)	线程正在运行
阻塞(Blocked)	线程等待外部事件发生而无法运行，如I/O操作
终止(Terminated)	线程完成，或退出，或被取消



Python中的进程和线程

运行程序会启动一个解释器进程，线程共享一个解释器进程。

Python的线程开发

Python的线程开发使用标准库threading。

进程靠线程执行代码，至少有一个**主线程**，其它线程是工作线程。主线程是第一个启动的线程，由解释器创建。

父线程：如果线程A中启动了一个线程B，A就是B的父线程。
子线程：B就是A的子线程。

Thread类

```
# 签名
def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, *, daemon=None)
```

参数名	含义
target	线程调用的对象，就是目标函数
name	为线程起个名字
args	为目标函数传递实参，元组
kwargs	为目标函数关键字传参，字典

group必须为None，保留，留给未来实现线程组。截止3.8还未实现。

线程启动

```
import threading

# 最简单的线程程序
def worker():
    print("I'm working")
    print('Fineshed')

t = threading.Thread(target=worker, name='worker') # 线程对象
t.start() #  启动
```

通过threading.Thread创建一个线程对象，target是目标函数，可以使用name为线程指定名称。
但是线程没有启动，需要调用start方法。

线程之所以执行函数，是因为线程中就是要执行代码的，而最简单的代码封装就是函数，所以还是函数调用。
函数执行完，线程也就退出了。
那么，如果不让线程退出，或者让线程一直工作怎么办呢?

```
import threading
import time

def worker():
    while True: # for i in range(10):
        time.sleep(0.5)
        print("I'm working")
    print('Fineshed')

t = threading.Thread(target=worker, name='worker') # 线程对象
t.start() #  启动

print('=' * 30) # 注意看这行等号什么时候打印的?
```

线程退出

Python没有提供线程退出的方法，线程在下面情况时退出
1、线程函数内语句执行完毕
2、线程函数中抛出未处理的异常

```
import threading
import time

def worker():
    for i in range(10):
        time.sleep(0.5)
        if i > 5:
            #break # 终止循环
            #return # 函数返回
            raise RuntimeError # 抛异常
        print('I am working')
    print('finished')

t = threading.Thread(target=worker, name='worker')
t.start()

print('=' * 30)
```

Python的线程没有优先级、没有线程组的概念，也不能被销毁、停止、挂起，那也就没有恢复、中断了。

线程的传参

```
import threading
import time

def add(x, y):
    print('{} + {} = {}'.format(x, y, x + y, threading.current_thread().ident))

t1 = threading.Thread(target=add, name='add', args=(4, 5))
t1.start()
time.sleep(2)

t2 = threading.Thread(target=add, name='add', args=(6,), kwargs={'y':7})
t2.start()
time.sleep(2)

t3 = threading.Thread(target=add, name='add', kwargs={'x':8, 'y':9})
t3.start()
```

线程传参和函数传参没什么区别，本质上就是函数传参。

threading的属性和方法

名称	含义
current_thread()	返回当前线程对象
main_thread()	返回主线程对象
active_count()	当前处于alive状态的线程个数
enumerate()	返回所有活着的线程的列表，不包括已经终止的线程和未开始的线程
get_ident()	返回当前线程的ID，非0整数

active_count、enumerate方法返回的值还包括主线程。

```
import threading
import time

def showtreadinfo():
    print('current thread = {}\nmain thread = {}\nactive count = {}'.format(
        threading.current_thread(), threading.main_thread(), threading.active_count()
    ))

def worker():
    showtreadinfo()
    for i in range(5):
        time.sleep(1)
        print('i am working')
    print('finished')

t = threading.Thread(target=worker, name='worker') # 线程对象
showtreadinfo()
time.sleep(1)
t.start() # 启动

print('===end===')
```

Thread实例的属性和方法

名称	含义
name	只是一个名字，只是个标识，名称可以重名。getName()、setName()获取、设置这个名词
ident	线程ID，它是非0整数。线程启动后才会有ID，否则为None。线程退出，此ID依旧可以访问。此ID可以重复使用
is_alive()	返回线程是否活着

注意：线程的name这是一个名称，可以重复；ID必须唯一，但可以在线程退出后再利用。

```
import threading
import time

def worker():
    for i in range(5):
        time.sleep(1)
        print('i am working')
    print('finished')

t = threading.Thread(target=worker, name='worker') # 线程对象
print(t.name, t.ident)
time.sleep(1)
t.start() # 启动

print('===end===')

while True:
    time.sleep(1)
    print('{} {} {}'.format(t.name, t.ident,
            'alive' if t.is_alive() else 'dead'))

    if not t.is_alive():
        print('{} restart'.format(t.name))
        t.start() # 线程重启？？
```

start和run方法

```
import threading
import time

def worker():
    for i in range(5):
        time.sleep(1)
        print('I am working')
    print('finished')

class MyThread(threading.Thread):
    def start(self):
        print('start~~~~')
        super().start()

    def run(self):
        print('run~~~~~~')
        super().run()

t = MyThread(target=worker, name='worker') # 线程对象
t.start() # 启动
t.start()
```

```
# t.run() # 或调用run方法
# t.run()
```

尝试start两次，或run两次都失败了，但是它们抛出的异常不一样。

但是单独运行start或者run都可以，是否可以不需要start方法了吗？在worker中打印线程名称、id。

```
import threading
import time

def worker():
    t = threading.current_thread()
    for i in range(5):
        time.sleep(1)
        print('I am working', t.name, t.ident)
    print('finished')

class MyThread(threading.Thread):
    def start(self):
        print('start~~~~')
        super().start()

    def run(self):
        print('run~~~~~~')
        super().run()

t = MyThread(target=worker, name='worker') # 线程对象
t.start() # 启动
```

start方法才能启动操作系统线程，并运行run方法。run方法内部调用了目标函数。

多线程

顾名思义，多个线程，一个进程中如果有多个线程运行，就是多线程，实现一种并发。

```
import threading
import time
import sys

def worker(f=sys.stdout):
    t = threading.current_thread()
    for i in range(5):
        time.sleep(1)
        print('i am working', t.name, t.ident, file=f)
    print('finished', file=f)

t1 = threading.Thread(target=worker, name='worker1')
t2 = threading.Thread(target=worker, name='worker2', args=(sys.stderr,))
t1.start()
```

```
t2.start()
```

可以看到worker1和work2交替执行。

当使用start方法启动线程后，进程内有多个活动的线程并行的工作，就是多线程。

一个进程中至少有一个线程，并作为程序的入口，这个线程就是**主线程**。
一个进程至少有一个主线程。
其他线程称为**工作线程**。

线程安全

多线程执行一段代码，不会产生不确定的结果，那这段代码就是线程安全的。

多线程在运行过程中，由于共享同一进程中的数据，多线程并发使用同一个数据，那么数据就有可能被相互修改，从而导致某些时刻无法确定这个数据的值，最终随着多线程运行，运行结果不可预期，这就是线程不安全。

daemon线程

注：有人翻译成后台线程，也有人翻译成守护线程。

Python中，构造线程的时候，可以设置daemon属性，这个属性必须在start方法前设置好。

```
# 源码Thread的__init__方法中
if daemon is not None:
    self._daemonic = daemon # 用户设定bool值
else:
    self._daemonic = current_thread().daemon
```

线程daemon属性，取用户设置的值，否则就取当前线程（父线程）的daemon值。
主线程是non-daemon线程，即daemon = False。

```
class _MainThread(Thread):
    def __init__(self):
        Thread.__init__(self, name="MainThread", daemon=False)
```

这说明，daemon如果不设置取父线程的daemon值。

```
import time
import threading

def foo():
    time.sleep(5)
    for i in range(20):
        print(i)

# 主线程是non-daemon线程
t = threading.Thread(target=foo, daemon=False)
t.start()

print('Main Thread Exits')
```

发现线程t依然执行，主线程已经执行完，但是一直等着线程t。
修改为`t = threading.Thread(target=foo, daemon=True)`试一试，结果程序立即结束了，进程根本没有等daemon线程t。

名称	含义
daemon属性	表示线程是否是daemon线程，这个值必须在start()之前设置，否则引发RuntimeError异常
isDaemon()	是否是daemon线程
setDaemon	设置为daemon线程，必须在start方法之前设置

看一个例子，，看看主线程何时结束daemon线程

```
import time
import threading

def worker(name, timeout):
    time.sleep(timeout)
    print('{} working'.format(name))

# 主线程 是non-daemon线程
t1 = threading.Thread(target=worker, args=('t1', 5), daemon=True) # 调换5和10看看效果
t1.start()

t2 = threading.Thread(target=worker, args=('t2', 10), daemon=False)
t2.start()

print('Main Thread Exits')
```

上例说明，如果还有non-daemon线程在运行，进程不结束，进程也不会杀掉其它所有daemon线程。直到所有non-daemon线程全部运行结束（包括主线程），不管有没有daemon线程，程序退出。

总结

- 主线程是non-daemon线程，即daemon = False
- 线程具有一个daemon属性，可以手动设置为True或False，也可以不设置，则取默认值None
- 如果daemon=None，就取当前线程（父线程）的daemon来设置它
- 从主线程创建的所有线程的不设置daemon属性，则默认都是daemon = False，也就是non-daemon线程
- Python程序在没有活着的non-daemon线程运行时，程序退出，包括non-daemon的主线程。也就是除主线程之外剩下的只能都是daemon线程，主线程才能退出，否则即使是主线程已经执行完，进程也不能结束，只能等待

join方法

先看一个简单的例子，看看效果

```
import time
import threading

def worker(name, timeout):
    time.sleep(timeout)
    print('{} working'.format(name))

t1 = threading.Thread(target=worker, args=('t1', 3), daemon=True)
t1.start()
t1.join()# 设置join，取消join对比一下

print('Main Thread Exits')
```

使用了join方法后，当前线程阻塞了，daemon线程执行完了，主线程才退出了。

```
import time
import threading

def worker(name, timeout):
    time.sleep(timeout)
    print('{} working'.format(name))

t1 = threading.Thread(target=worker, args=('t1', 10), daemon=True)
t1.start()
t1.join(2)
print('~~~~~~~~~~~')
t1.join(2)
print('~~~~~~~~~~~')

print('Main Thread Exits')
```

`join(timeout=None)`

- join方法是线程的标准方法之一
- 一个线程中调用另一个线程的join方法，调用者将被阻塞，直到被调用线程终止，或阻塞超时
- 一个线程可以被join多次
- timeout参数指定调用者等待多久，没有设置超时，就一直等到被调用线程结束
- 调用谁的join方法，就是join谁，就要等谁

线程同步

概念

线程同步，线程间协同，通过某种技术，让一个线程访问某些数据时，其他线程不能访问这些数据，直到该线程完成对数据的操作。

Event ***

Event事件，是线程间通信机制中最简单的实现，使用一个内部的标记flag，通过flag的True或False的变化来进行操作。

名称	含义
set()	标记设置为True
clear()	标记设置为False
is_set()	标记是否为True
wait(timeout=None)	设置等待标记为True的时长，None为无限等待。等到返回True，未等到超时了返回False

练习

老板雇佣了一个工人，让他生产杯子，老板一直等着这个工人，直到生产了10个杯子

```
# 下面的代码是否能够完成功能?
from threading import Event, Thread
import logging
import time

FORMAT = '%(asctime)s %(threadName)s %(thread)s %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)

cups = []
flag = False

def boss():
    logging.info("I'm boss, waiting for U")
    while True:
        time.sleep(1)
        if flag:
```

```
            break
    logging.info('Good Job.')

def worker(count=10):
    logging.info('I am working for U')

    while True:
        logging.info('make 1 cup')
        time.sleep(0.5)
        cups.append(1)

        if len(cups) >= count:
            flag = True
            break
    logging.info('I finished my job. cups={}'.format(cups))

b = Thread(target=boss, name='boss')
w = Thread(target=worker, name='worker')
b.start()
w.start()
```

上面代码基本能够完成，但上面代码问题有：

- bug，应该将worker中的flag定义为global就可解决
- 老板一直要不停的查询worker的状态变化

```
from threading import Event, Thread
import logging
import time

FORMAT = '%(asctime)s %(threadName)s %(thread)s %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)


def boss(event:Event):
    logging.info("I'm boss, waiting for U")
    event.wait() # 阻塞等待
    logging.info('Good Job.')

def worker(event:Event, count=10):
    logging.info('I am working for U')
    cups = []
    while True:
        logging.info('make 1 cup')
        time.sleep(0.5)
        cups.append(1)

        if len(cups) >= count:
```

```
            event.set()
            break
    logging.info('I finished my job. cups={}'.format(cups))


event = Event()
b = Thread(target=boss, name='boss', args=(event,))
w = Thread(target=worker, name='worker', args=(event,))
b.start()
w.start()
```

总结

需要使用同一个Event对象的标记flag。
谁wait就是等到flag变为True，或等到超时返回False。
不限制等待者的个数，通知所有等待者。

wait的使用

```
# 修改上例worker中的 while 条件
def worker(event:Event, count=10):
    logging.info('I am working for U')
    cups = []
    while not event.wait(0.5): # 使用wait阻塞等待
        logging.info('make 1 cup')
        cups.append(1)

        if len(cups) >= count:
            event.set()
            #break # 为什么可以注释break呢?
    logging.info('I finished my job. cups={}'.format(cups))
```

Lock ***

- Lock类是mutex互斥锁
- 一旦一个线程获得锁，其它**试图获取锁的线程**将被阻塞，只到拥有锁的线程释放锁
- 凡是存在共享资源争抢的地方都可以使用锁，从而保证只有一个使用者可以完全使用这个资源。

名称	含义
acquire(blocking=True, timeout=-1)	默认阻塞，阻塞可以设置超时时间。非阻塞时，timeout禁止设置。 成功获取锁，返回True，否则返回False
release()	释放锁。可以从任何线程调用释放。 已上锁的锁，会被重置为unlocked 未上锁的锁上调用，抛RuntimeError异常。

锁的基本使用

```
import threading
import time

lock = threading.Lock() # 互斥mutex

lock.acquire()
print('-' * 30)


def worker(l):
    print('worker start', threading.current_thread())
    l.acquire()
    print('worker done', threading.current_thread())

for i in range(10):
    threading.Thread(target=worker, name="w{}".format(i),
                     args=(lock,), daemon=True).start()

print('-' * 30)

while True:
    cmd = input(">>>")
    if cmd == 'r': # 按r后枚举所有线程看看
        lock.release()
        print('released one locker')
    elif cmd == 'quit':
        lock.release()
        break
    else:
        print(threading.enumerate())
        print(lock.locked())
```

上例可以看出不管在哪一个线程中，只要对一个已经上锁的锁发起阻塞地请求，该线程就会阻塞。

练习

订单要求生产1000个杯子，组织10个工人生产。请忽略老板，关注工人生成杯子

```
import threading
from threading import Thread, Lock
import time
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
```

```
logging.basicConfig(format=FORMAT, level=logging.INFO)

cups = []

def worker(count=1000):
    logging.info("I'm working")
    while True:
        if len(cups) >= count:
            break
        time.sleep(0.0001) # 为了看出线程切换效果，模拟杯子制作时间
        cups.append(1)
    logging.info('I finished my job. cups = {}'.format(len(cups)))

for i in range(1, 11):
    t = Thread(target=worker, name="w{}".format(i), args=(1000,))
    t.start()
```

从上例的运行结果看出，多线程调度，导致了判断失效，多生产了杯子。

如何修改解决这个问题？加锁

上例使用锁实现如下：

```
import threading
from threading import Thread, Lock
import time
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

cups = []
lock = Lock() # 锁

def worker(count=1000):
    logging.info("I'm working")
    while True:
        lock.acquire() # 获取锁

        if len(cups) >= count:
            #lock.release() # 1
            break
        #lock.release()     # 2
        time.sleep(0.0001) # 为了看出线程切换效果，模拟杯子制作时间
        cups.append(1)
        lock.release()     # 3
    logging.info('I finished my job. cups = {}'.format(len(cups)))

for i in range(1, 11):
    t = Thread(target=worker, name="w{}".format(i), args=(1000,))
```

```
    t.start()
```

锁分析

位置2分析

- 假设某一个瞬间，有一个工作线程A获取了锁，len(cups)正好有999个，然后就释放了锁，可以继续执行下面的语句，生产一个杯子，这地方不阻塞，但是正好杯子也没有生产完。锁释放后，其他线程就可以获得锁，线程B获得了锁，发现len(cups)也是999个，然后释放锁，然后也可以去生产一个杯子。锁释放后，其他的线程也可能获得锁。就说A和B线程都认为是999个，都会生产一个杯子，那么实际上最后一定会超出1000个。
- 假设某个瞬间一个线程获得锁，然后发现杯子到了1000个，没有释放锁就直接break了，由于其他线程还在阻塞等待锁释放，这就成了**死锁**了。

位置3分析

- 获得锁的线程发现是999，有资格生产杯子，生产一个，释放锁，看似很完美
- 问题在于，获取锁的线程发现杯子有1000个，直接break，没释放锁离开了，死锁了

位置1分析

- 如果线程获得锁，发现是1000，break前释放锁，没问题
- 问题在于，A线程获得锁后，发现小于1000，继续执行，其他线程获得锁全部阻塞。A线程再次执行循环后，自己也阻塞了。死锁了。

问题：究竟怎样加锁才正确呢?

要在位置1和位置3同时加release。

上下文支持

锁是典型必须释放的，Python提供了上下文支持。查看Lock类的上下文方法，__enter__方法返回bool表示是否获得锁，__exit__方法中释放锁。

由此上例可以修改为

```
import threading
from threading import Thread, Lock
import time
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

cups = []
lock = Lock() # 锁

def worker(count=1000):
    logging.info("I'm working")
    while True:
```

```
        with lock: # 获取锁，离开with释放锁
            if len(cups) >= count:
                logging.info('leaving')
                break
            time.sleep(0.0001) # 为了看出线程切换效果，模拟杯子制作时间
            cups.append(1)
            logging.info(lock.locked())

    logging.info('I finished my job. cups = {}'.format(len(cups)))

for i in range(1, 11):
    t = Thread(target=worker, name="w{}".format(i), args=(1000,))
    t.start()
```

感觉一下正确得到结果了吗？感觉到了执行速度了吗？慢了还是快了，为什么？

锁的应用场景

锁适用于访问和修改同一个共享资源的时候，即读写同一个资源的时候。

如果全部都是读取同一个共享资源需要锁吗?
不需要。因为这时可以认为共享资源是不可变的，每一次读取它都是一样的值，所以不用加锁

使用锁的注意事项：

- 少用锁，必要时用锁。使用了锁，多线程访问被锁的资源时，就成了串行，要么排队执行，要么争抢执行
    - 举例，高速公路上车并行跑，可是到了省界只开放了一个收费口，过了这个口，车辆依然可以在多车道上一起跑。过收费口的时候，如果排队一辆辆过，加不加锁一样效率相当，但是一旦出现争抢，就必须加锁一辆辆过。注意，不管加不加锁，只要是一辆辆过，效率就下降了。
- 加锁时间越短越好，不需要就立即释放锁
- 一定要避免死锁

不使用锁，有了效率，但是结果是错的。
使用了锁，效率低下，但是结果是对的。
所以，我们是为了效率要错误结果呢？还是为了对的结果，让计算机去计算吧

Queue的线程安全

标准库queue模块，提供FIFO的Queue、LIFO的队列、优先队列。
Queue类是线程安全的，适用于同一进程内多线程间安全的交换数据。内部使用了Lock和Condition。

特别注意下面的代码在多线程中使用

```
import queue

q = queue.Queue(8)

if q.qsize() == 7:
    q.put() # 上下两句可能被打断

if q.qsize() == 1:
    q.get() # 未必会成功
```

如果不加锁，是不可能获得准确的大小的，因为你刚读取到了一个大小，还没有取走数据，就有可能被其他线程改了。
Queue类的size虽然加了锁，但是，依然不能保证立即get、put就能成功，因为读取大小和get、put方法是分开的。

多进程

多线程未必是CPU密集型程序的好的选择。
多进程可以完全独立的进程环境中运行程序，可以较充分地利用多处理器。
但是进程本身的隔离带来的数据不共享也是一个问题。而且线程比进程轻量级。

multiprocessing

Process类

Process类遵循了Thread类的API，减少了学习难度。

先看一个例子，前面介绍的单线程、多线程比较的例子的多进程版本

```
import multiprocessing
import datetime

def calc(i):
    sum = 0
    for _ in range(1000000000): # 10亿
        sum += 1
    return i, sum

if __name__ == '__main__':
    start = datetime.datetime.now() # 注意一定要有这一句

    ps = []
    for i in range(4):
        p = multiprocessing.Process(target=calc, args=(i,), name='calc-{}'.format(i))
        ps.append(p)
        p.start()

    for p in ps:
```

```
        p.join()
        print(p.name, p.exitcode)

    delta = (datetime.datetime.now() - start).total_seconds()
    print(delta)
    for p in ps:
        print(p.name, p.exitcode)
    print('===end====')
```

对于上面这个程序，在同一主机（授课主机）上运行时长的对比

- 使用单线程、多线程跑了4分钟多
- 多进程用了1分半

看到了多个进程都在使用CPU，这是真并行，而且进程库几乎没有什么学习难度

注意：多进程代码一定要放在`__name__ == "__main__"`下面执行。

名称	说明
pid	进程id
exitcode	进程的退出状态码
terminate()	终止指定的进程

进程间同步

Python在进程间同步提供了和线程同步一样的类，使用的方法一样，使用的效果也类似。
不过，进程间代价要高于线程间，而且系统底层实现是不同的，只不过Python屏蔽了这些不同之处，让用户简单使用多进程。

multiprocessing还提供共享内存、服务器进程来共享数据，还提供了用于进程间通讯的Queue队列、Pipe管道。

多进程、多线程的选择

1、CPU密集型
CPython中使用到了GIL，多线程的时候锁相互竞争，且多核优势不能发挥，选用Python多进程效率更高。
2、IO密集型
在Python中适合是用多线程，可以减少多进程间IO的序列化开销。且在IO等待的时候，切换到其他线程继续执行，效率不错。

应用

请求/应答模型：WEB应用中常见的处理模型

master启动多个worker工作进程，一般和CPU数目相同。发挥多核优势。
worker工作进程中，往往需要操作网络IO和磁盘IO，启动多线程，提高并发处理能力。worker处理用户的请求，往往需要等待数据，处理完请求还要通过网络IO返回响应。
这就是nginx工作模式。

concurrent.futures包

3.2版本引入的模块。

异步并行任务编程模块，提供一个高级的异步可执行的便利接口。

提供了2个池执行器：

- **ThreadPoolExecutor** 异步调用的线程池的Executor
- **ProcessPoolExecutor** 异步调用的进程池的Executor

ThreadPoolExecutor对象

首先需要定义一个池的执行器对象，Executor类的子类实例。

方法	含义
ThreadPoolExecutor(max_workers=1)	池中至多创建max_workers个线程的池来同时异步执行，返回Executor实例 支持上下文，进入时返回自己，退出时调用shutdown(wait=True)
submit(fn, *args, **kwargs)	提交执行的函数及其参数，如有空闲开启daemon线程，返回Future类的实例
shutdown(wait=True)	清理池，wait表示是否等待到任务线程完成

Future类

方法	含义
done()	如果调用被成功的取消或者执行完成，返回True
cancelled()	如果调用被成功的取消，返回True
running()	如果正在运行且不能被取消，返回True
cancel()	尝试取消调用。如果已经执行且不能取消返回False，否则返回True
result(timeout=None)	取返回的结果，timeout为None，一直等待返回；timeout设置到期，抛出concurrent.futures.TimeoutError 异常
exception(timeout=None)	取返回的异常，timeout为None，一直等待返回；timeout设置到期，抛出concurrent.futures.TimeoutError 异常

```
from concurrent.futures import ThreadPoolExecutor, wait
import datetime
import logging
```

```
FORMAT = "%(asctime)s [%(processName)s %(threadName)s] %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

def calc(base):
    sum = base
    for i in range(100000000):
        sum += 1
    logging.info(sum)
    return sum

start = datetime.datetime.now()
executor = ThreadPoolExecutor(3)
with executor: # 默认shutdown阻塞
    fs = []
    for i in range(3):
        future = executor.submit(calc, i*100)
        fs.append(future)

    #wait(fs) # 阻塞
    print('-' * 30)
for f in fs:
    print(f, f.done(), f.result()) # done不阻塞，result阻塞
print('=' * 30)

delta = (datetime.datetime.now() - start).total_seconds()
print(delta)
```

ProcessPoolExecutor对象

方法一样。就是使用多进程完成。

```
from concurrent.futures import ProcessPoolExecutor, wait
import datetime
import logging

FORMAT = "%(asctime)s [%(processName)s %(threadName)s] %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

def calc(base):
    sum = base
    for i in range(100000000):
        sum += 1
    logging.info(sum)
    return sum

if __name__ == '__main__':
    start = datetime.datetime.now()
    executor = ProcessPoolExecutor(3)
    with executor: # 默认shutdown阻塞
```

```
        fs = []
        for i in range(3):
            future = executor.submit(calc, i*100)
            fs.append(future)

        #wait(fs) # 阻塞
        print('-' * 30)
    for f in fs:
        print(f, f.done(), f.result()) # done不阻塞，result阻塞
    print('=' * 30)

    delta = (datetime.datetime.now() - start).total_seconds()
    print(delta)
```

总结

该库统一了线程池、进程池调用，简化了编程。
是Python简单的思想哲学的体现。

唯一的缺点：无法设置线程名称。但这都不值一提。