内容简介

本书是面向编程零基础读者的 Python 入门教程，内容涵盖了 Python 的初步应用。以比较轻快的风格，向零基础的学习者介绍了一门时下比较流行、并且用途比较广泛的编程语言。同时，其语法简洁而清晰，类库丰富而强大，非常适合于进行快速原型开发。另外，Python 可以运行在多种系统平台下，从而使得只需要编写一次代码，就可以多个系统平台下保持有同等的功能。

为了能够使广大读者既能够掌握 Python 语言，又能够将 Python 语言用于某个特定的领域，本书将全面介绍和 Python 相关的这些内容。在学习完本书之后，相信读者能够很好地掌握 Python 语言，同时可以使用 Python 语言进行实际项目的开发。

本书以理论与实际相结合为原则，为每个知识点都设计了对应的示例，让 Python 的初学者能够既快速又深刻的理解这些知识点。同时在每章的最后设计了针对各章内容的作业题，能够让读者趁热打铁，以达到巩固所学知识的目的。

本书可配合尚学堂科技推出的《Python400 集大型视频课程》，作为参考教材。

前言

本书特色

学员需求：学习不枯燥、能够快速入门、实战操作、熟悉底层原理；企业要求：程序员既有实战技能可以快速上手，也内功扎实熟悉底层原理后劲十足。因此，针对以上需求我们北京尚学堂科技有限公司设计了本书。

本书的主要有以下几个特点：

1. 循序渐进，由浅入深

为了方便读者学习，本书首先让读者了解了 Python 的历史和特点，通过具体的例子逐渐把读者带入 Python 的世界，掌握 Python 语言的基本要点以及基础类库、常用库和工具的使用。

2. 技术全面，内容充实

本书在保证内容使用的前提下，详细介绍了 Python 语言的各个知识点。同时，本书所涉及的内容非常全面，无论从事什么行业的读者，都可以从本书中找到可应用 Python 与本行业的地方。

3. 代码完整，详解详尽

对于书中的每个知识点都有一段示例代码，并对代码的关键点进行了注释说明。每段代码的后面都有详细的分析，同时给出了代码运行后的结果。读者可以参考运行结果阅读源程序，可以加深对程序的理解。

本书如何学习

本书共分七章，为了方便大家的学习，我们对各章节做简要说明。

第一章：讲解了正则表达式的概念以及 re 模块处理正则表达式。

第二章：讲解了 Python 中进程和线程的概念。主要的内容包括进程和线程的创建及管理。重点讲解了多线程环境下数据同步。

第三章：讲解了 Python 中和网络编程相关的内容，包括使用 TCP/UDP 协议实现服务器端和客户端的通信。

第四章：讲解了 Python 自带的 GUI 开发库 Tkinter 的基本组件及其使用方法，并给出每种组件的详细示例代码。

第五章：讲解了坦克大战游戏详细实现。用一个游戏项目将前面的基础知识做了串联，让大家了解项目开发的全流程。

第六章：讲解了 Python 数据库开发方面的知识，重点讲解了 SQLite 数据库和 MySQL 数据库。数据库技术是实现动态软件技术的必要手段，因此掌握数据库开发是非常必要的。

第七章：协程和异步 IO。本章讲解了线程、进程和协程的优缺点、协程的概念、协程的创建、协程阻塞、协程嵌套和并行与并发的概念。

第八章：函数式编程和高阶函数。本章主要讲解了什么是函数式编程、常用高阶函数（map、reduce、filter、sorted）、匿名函数、闭包装饰器及偏函数的应用。

鸣谢

本书由北京尚学堂科技的教研部编制。北京尚学堂科技是一家从事人工智能、大数据、区块链、Java 技术、数据库、前端全栈、UI 设计等互联网技术培训的公司，其编写的教材和发布的教程影响了一代程序员。公司多次获得新浪、腾讯、网易、央视等年度最佳机构的荣誉。

本书在出版过程中，得到了清华大学出版社杨如林老师的大力支持，在此表示衷心感谢。另外，本书所有的编审、发行人员为本书的出版发行付出了辛勤的劳动，在此一并致以诚挚的谢意。

本书主编为高淇，主要参编人员有：高淇、张巧英等。我们以科学、严谨的态度，力求精益求精，但错误之处，在所难免，敬请广大读者批评指正，我们将不胜感激。

教研部出版组邮箱：book@sxt.cn　　高淇老师邮箱：gaoqi@sxt.cn

目录

第二篇 【python 进阶和高级编程】

案例目录

第一章 正则表达式

正则表达式就是通过一个文本模式来匹配一组符合条件的字符串。这个文本模式是由一些字符和特殊符号组成的字符串，它们描述了模式的重复或表述多个字符，所以正则表达式能按着某种模式匹配一系列有相似特征的字符串。

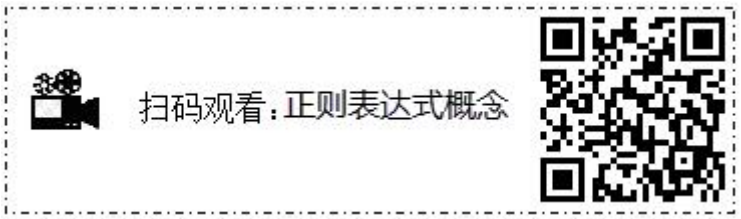
正则表述式用于搜索、替换和解析字符串。正则表达式遵循一定的语法规则，使用非常灵活，功能强大。使用正则表达式编写一些逻辑验证非常方便，例如电子邮件地址格式的验证。

通过阅读本章，你可以：

- 了解什么是正则表达式
- 掌握用 match 方法匹配字符串
- 掌握用 search 方法搜索满足条件的字符串
- 掌握用 findall 方法和 finditer 方法查找字符串
- 掌握用 split 方法分隔字符串
- 掌握常用的正则表达式表示法

1.1 正则表达式简介

1.1.1 概念



正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑（可以用来做检索，截取或者替换操作）。

正则表述式用于搜索、替换和解析字符串。正则表达式遵循一定的语法规则，使用非常灵活，功能强大。使用正则表达式编写一些逻辑验证非常方便，例如电子邮件地址格式的验证。

正则表达式是对字符串（包括普通字符（例如，a 到 z 之间的字母）和特殊字符）操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑，正则表达式是一种文本模式，模式描述在搜索文本时要匹配一个或多个字符串。

1.1.2 作用

（1）给定的字符串是否符合正则表达式的过滤逻辑（称作“匹配”）。

（2）可以通过正则表达式，从字符串中获取我们想要的特定部分。

（3）还可以对目标字符串进行替换操作。

1.2 正则表达式的使用

Python 语言通过标准库中的 re 模块支持正则表达式。re 模块提供了一些根据正则表达式进行查找、替换、分隔字符串的函数，这些函数使用一个正则表达式作为第一个参数。re 模块常用的函数如表 1-1 所示：

表 1-1 re 模块常用的函数

函数	描述
match(pattern,string,flags=0)	根据 pattern 从 string 的头部开始匹配字符串，只返回第 1 次匹配成功的对象；否则，返回 None
findall(pattern,string,flags=0)	根据 pattern 在 string 中匹配字符串。如果匹配成功，返回包含匹配结果的列表；否则，返回空列表。当 pattern 中有分组时，返回包含多个元组的列表，每个元组对应 1 个分组。flags 表示规则选项，规则选项用于辅助匹配。
sub(pattern,repl,string,count=0)	根据指定的正则表达式，替换源字符串中的子串。pattern 是一个正则表达式，repl 是用于替换的字符串，string 是源字符串。如果 count 等于 0，则返回 string 中匹配的所有结果；如果 count 大于 0，则返回前 count 个匹配结果
subn(pattern,repl,string,count=0)	作用和 sub()相同，返回一个二元的元组。第 1 个元素是替换结果，第 2 个元素是替换的次数
search(pattern,string,flags=0)	根据 pattern 在 string 中匹配字符串，只返回第 1 次匹配成功的对象。如果匹配失败，返回 None
compile(pattern,flags=0)	编译正则表达式 pattern，返回 1 个 pattern 的对象
split(pattern,string,maxsplit=0)	根据 pattern 分隔 string，maxsplit 表示最大的分隔数
escape(pattern)	匹配字符串中的特殊字符，如*、+、?等

1.2.1 match 方法

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。语法格式如下：

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

函数参数说明如表 1-2 所示：

表 1-2 match 函数参数说明

参数	描述
pattern	匹配的正则表达式

参数	描述
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，多行匹配等等。如下表列出正则表达式修饰符 - 可选标志

表 1-3 正则表达式修饰符 - 可选标志

修饰符	描述
re.I	使匹配对大小写不敏感
re.L	做本地化识别（locale-aware）匹配
re.M	多行匹配，影响 ^ 和 $
re.S	使 . 匹配包括换行在内的所有字符
re.U	根据 Unicode 字符集解析字符。这个标志影响 \w, \W, \b, \B.
re.X	该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。

匹配字符串是正则表达式中最常用的一类应用。也就是设定一个文本模式，然后判断另外一个字符串是否符合这个文本模式。

如果文本模式只是一个普通的字符串，那么待匹配的字符串和文本模式字符串在完全相等的情况下，match 方法会认为匹配成功。如果匹配成功，则 match 方法返回匹配的对象，然后可以调用对象中的 group 方法获取匹配成功的字符串，如果文本模式就是一个普通的字符串，那么 group 方法返回的就是文本模式字符串本身。

【示例 1-1】match 方法的使用

```
import re
s='hello python'
pattern='hello'
v=re.match(pattern,s)
print(v)
print(v.group())
print(v.span())
```

执行结果如图 1-1 所示：

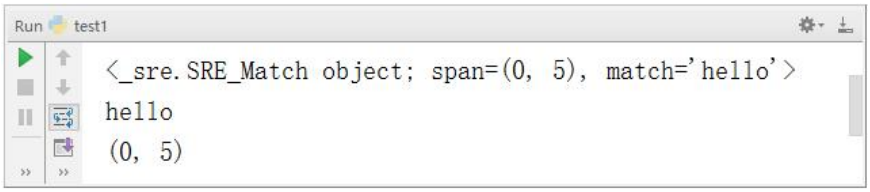


图 1-1 示例 1-1 执行结果

从上面的代码可以看出，进行文本模式匹配时，只要待匹配的字符串开始部分可以匹配

文本模式，就算匹配成功。

【示例 1-2】match 方法中 flag 可选标志的使用

```
import re
s = 'hello Python!'
m=re.match('hello python',s,re.I)    #忽略大小写
if m is not None:
    print('匹配成功结果是：',m.group())
else:
    print('匹配失败')
```

执行结果如图 1-2 所示：

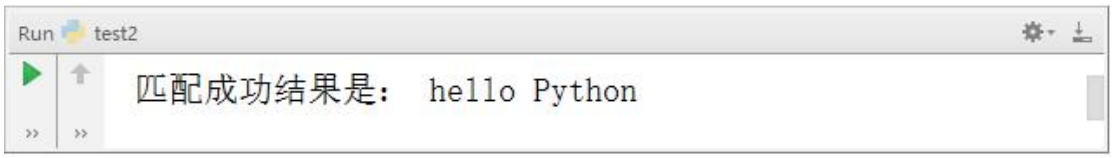
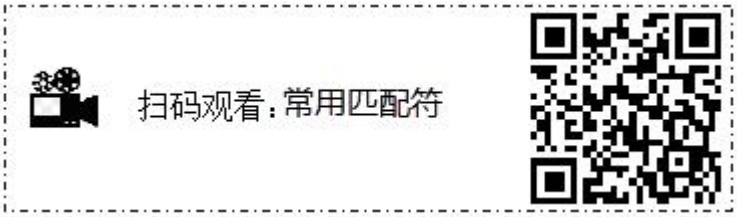


图 1-2 示例 1-2 执行结果

1.2.2 常用匹配符



一个正则表达式是由字母、数字和特殊字符（括号、星号、问号等）组成。正则表达式中有许多特殊的字符，这些特殊字符是构成正则表达式的要素。表 1-4 说明了正则表达式中常用字符的含义。

表 1-4 正则表达式中常用的字符

符号	描述
.	匹配任意一个字符（除了\n）
[]	匹配列表中的字符
\w	匹配字母、数字、下划线，即 a-z, A-Z, 0-9, _
\W	匹配不是字母、数字、下划线
\s	匹配空白字符, 即空格（\n, \t）
\S	匹配不是空白的字符
\d	匹配数字, 即 0-9
\D	匹配非数字的字符

【示例 1-3】常用匹配符.的使用

```
import re
pattern='.' #匹配任意一个字符（除了\n）
s='a'
print('匹配字符 a：',re.match(pattern,s))
s='C'
print('匹配字符 C：',re.match(pattern,s))
s='_'
print('匹配字符_：',re.match(pattern,s))
s='\n'
print('匹配字符\\n：',re.match(pattern,s))
```

执行结果如图 1-3 所示：

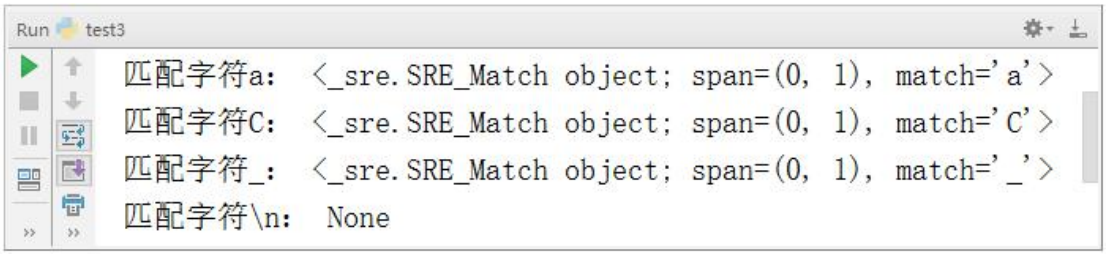


图 1-3 示例 1-3 执行结果

【示例 1-4】常用匹配符\d 的使用

```
import re
pattern='\d'   #匹配数字, 即 0-9
s='9'
print('匹配数字 9：',re.match(pattern,s))
s='4'
print('匹配数字 4：',re.match(pattern,s))
s='a'
print('匹配字符 a：',re.match(pattern,s))
s='_'
print('匹配字符_：',re.match(pattern,s))
```

执行结果如图 1-4 所示：

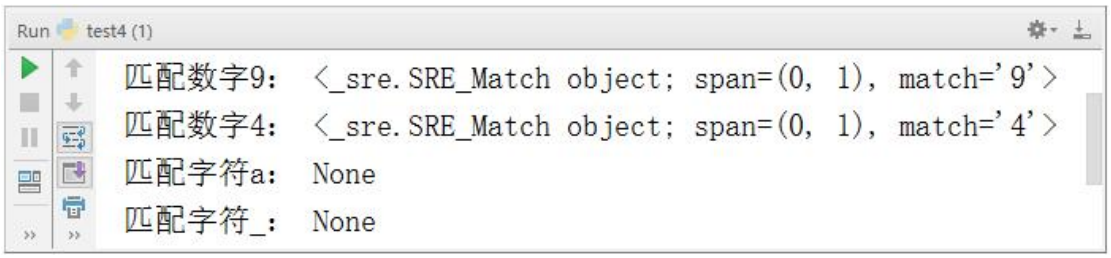


图 1-4 示例 1-4 执行结果

【示例 1-5】常用匹配符\D 的使用

```
import re
pattern='\D' #匹配非数字的字符
s='9'
print('匹配数字 9：',re.match(pattern,s))
s='4'
print('匹配数字 4：',re.match(pattern,s))
s='a'
print('匹配字符 a：',re.match(pattern,s))
s='_'
print('匹配数字_：',re.match(pattern,s))
```

执行结果如图 1-5 所示：

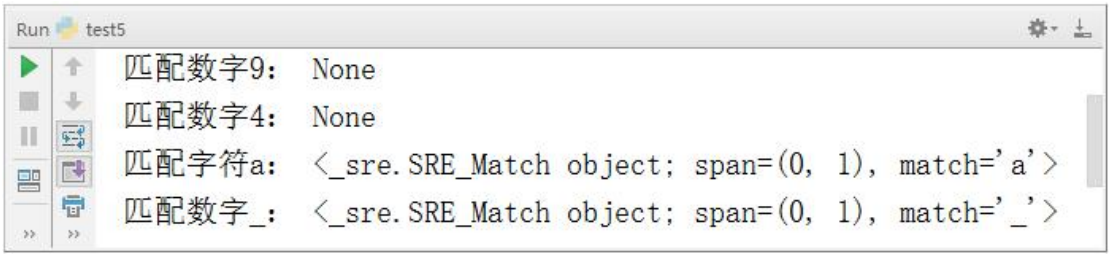


图 1-5 示例 1-5 执行结果

【示例 1-6】常用匹配符\s 的使用

```
import re
pattern='\s' #匹配空白字符,即空格（\n,\t）
s=' '
print('匹配字符' '：',re.match(pattern,s))
s='\t'
print('匹配字符\\t：',re.match(pattern,s))
```

```
s='\n'
print('匹配字符\\n：',re.match(pattern,s))
s='_'
print('匹配字符_：',re.match(pattern,s))
```

执行结果如图 1-6 所示：

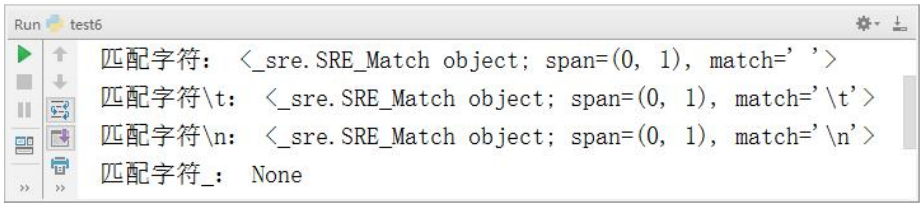


图 1-6 示例 1-6 执行结果

【示例 1-7】常用匹配符\S 的使用

```
import re
pattern='\S' #匹配不是空白的字符
s=' '
print('匹配字符' '：',re.match(pattern,s))
s='\t'
print('匹配字符\\t：',re.match(pattern,s))
s='\n'
print('匹配字符\\n：',re.match(pattern,s))
s='_'
print('匹配字符_：',re.match(pattern,s))
```

执行结果如图 1-7 所示：

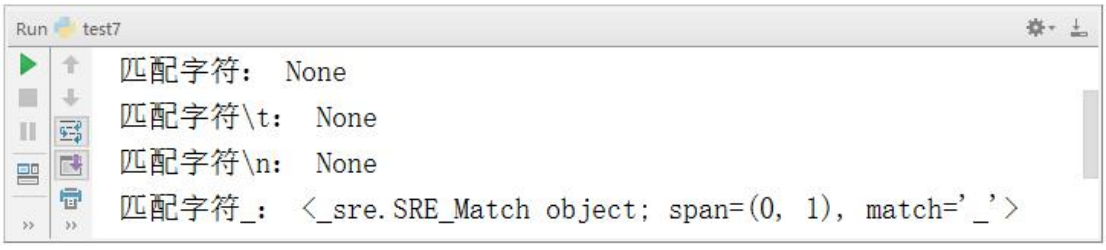


图 1-7 示例 1-7 执行结果

【示例 1-8】常用匹配符\w 和\W 的使用

```
import re
print('-------\w 匹配字母、数字、下划线--------')
pattern='\w' #匹配字母、数字、下划线
s='a'
```

```
print('匹配字符 a：',re.match(pattern,s))
s='_'
print('匹配字符_：',re.match(pattern,s))
s='5'
print('匹配数字 5：',re.match(pattern,s))
s='A'
print('匹配字符 A：',re.match(pattern,s))
s='#'
print('匹配字符#：',re.match(pattern,s))
print('-------\W 匹配不是字母、数字、下划线--------')
pattern='\W' #匹配不是字母、数字、下划线
s='a'
print('匹配字符 a：',re.match(pattern,s))
s='_'
print('匹配字符_：',re.match(pattern,s))
s='5'
print('匹配数字 5：',re.match(pattern,s))
s='A'
print('匹配字符 A：',re.match(pattern,s))
s='#'
print('匹配字符#：',re.match(pattern,s))
```

执行结果如图 1-8 所示：

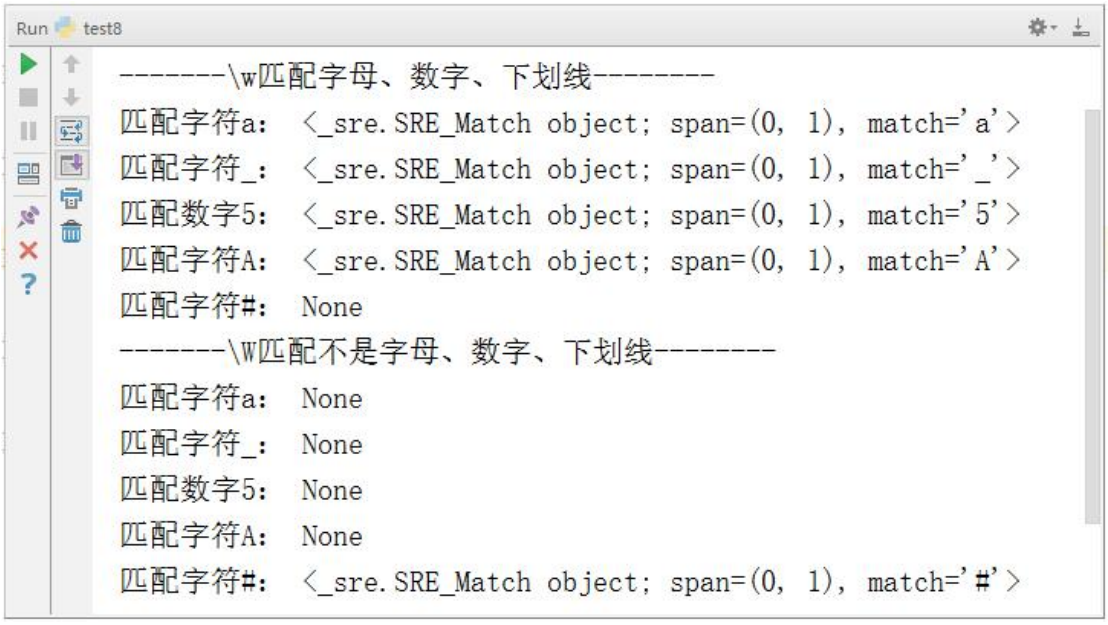


图 1-8 示例 1-8 执行结果

【示例 1-9】[]匹配列表中的字符

```
import re
pattern='[2468]' #匹配列表中的字符
s='1'
print('匹配数字 1：',re.match(pattern,s))
s='2'
print('匹配数字 2：',re.match(pattern,s))
s='3'
print('匹配数字 3：',re.match(pattern,s))
s='4'
print('匹配数字 4：',re.match(pattern,s))
s='#'
print('匹配字符#：',re.match(pattern,s))

print('---------手机号码----------')
s='13456788789'
pattern='\d\d\d\d\d\d\d\d\d\d\d'#匹配手机号
print('匹配手机号码：',re.match(pattern,s))
pattern='1[35789]\d\d\d\d\d\d\d\d\d'#匹配手机号
print('匹配手机号码：',re.match(pattern,s))
```

执行结果如图 1-9 所示：

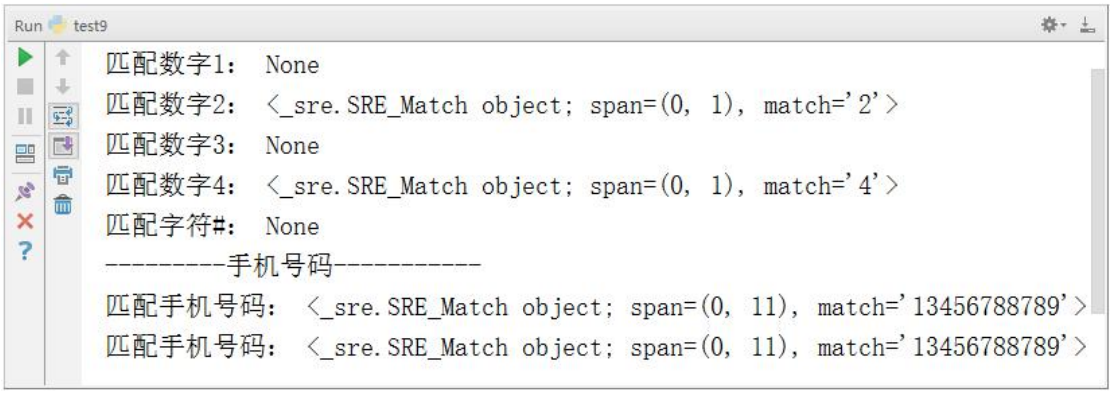


图 1-9 示例 1-9 执行结果

其中，匹配符“[]”可以指定一个范围，例如：“[ok]”将匹配包含“o”或“k”的字符。同时“[]”可以与\w、\s、\d 等标记等价。例如，[0-9a-zA-Z]等价于\w,[^0-9] 等价于\D。

1.2.3 限定符

从上面示例中可以看到如果要匹配手机号码，需要形如“\d\d\d\d\d\d\d\d\d\d\d”这样的正则表达式。其中表现了 11 次“\d”，表达方式烦琐。正则表达式作为一门小型的语言，还提供了对表达式的一部分进行重复处理的功能。例如，“*”可以对正则表达式的某个部分重复匹配多次。这种匹配符号称为限定符。表 1-5 列出了正则表达式中常用的限定符。利用{}可以控制符号重复的次数。

表 1-5 正则表达式中常用的限定符

符号	描述	符号	描述
*	匹配零次或多次	{m}	重复 m 次
+	匹配一次或多次	{m,n}	重复 m 到 n 次，其中 n 可以省略，表示 m 到任意次
?	匹配一次或零次	{m,}	至少 m 次

【示例 1-10】限定符*+?的使用

```
import re
print('------*匹配零次或多次--------')
pattern='\d*'   #0 次或多次
s='123abc'
print('匹配 123abc：',re.match(pattern,s))
s='abc'   #这时候不是 None 而是''
print('匹配 abc：',re.match(pattern,s))
print('-----+匹配一次或多次---------')
pattern='\d+'   #1 次或多次
s='123abc'
print('匹配 123abc：',re.match(pattern,s))
s='abc'   #这时候是 None
print('匹配 abc：',re.match(pattern,s))

print('-----?匹配一次或零次---------')
pattern='\d?'   #0 次或 1 次
s='123abc'
print('匹配 123abc：',re.match(pattern,s))
```

```
s='abc'  #这时候是空
print('匹配 abc：',re.match(pattern,s))
```

执行结果如图 1-10 所示：

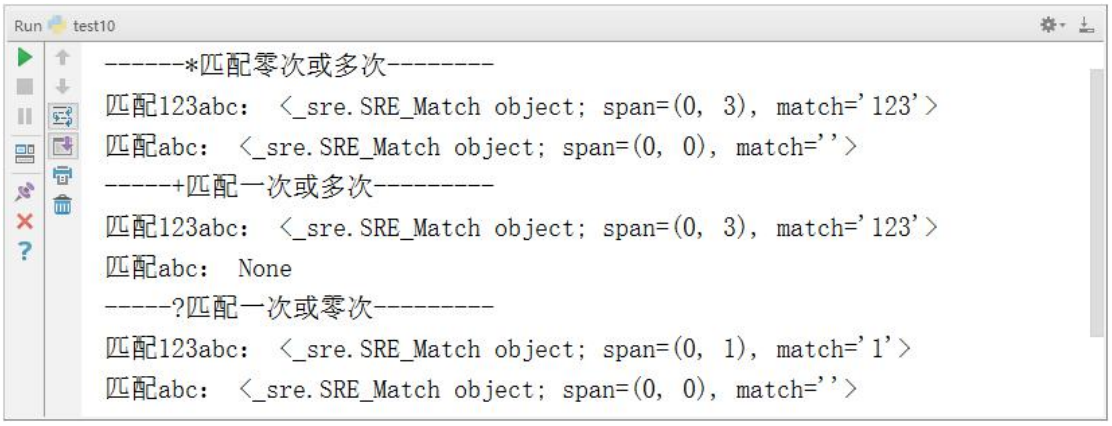


图 1-10 示例 1-10 执行结果

【示例 1-11】限定符{}的使用

```
import re
print('-----{m}重复 m 次---------')
pattern='\d{3}'  #出现 m 次
s='123abc'
print('pattern 为\\d{3}匹配 123abc 结果：',re.match(pattern,s))
pattern='\d{4}'  #出现 m 次
print('pattern 为\\d{4}匹配 123abc 结果：',re.match(pattern,s))

print('-----{m,}至少 m 次---------')
s='1234567abc'
pattern='\d{3,}'  #出现大于 m 次  尽可能满足的都返回
print('pattern 为\\d{3,}匹配 1234567abc 结果：\n',re.match(pattern,s))
print('-----{m,n}重复 m 到 n 次---------')
pattern='\d{2,4}'  #出现 m 到 n 次
print('pattern 为\\d{2,4}匹配 1234567abc 结果：\n',re.match(pattern,s))
```

执行结果如图 1-11 所示：

图 1-11 示例 1-11 执行结果

【示例 1-12】匹配出一个字符串首字母为大写字符，后边都是小写字符，这些小写字母可有可无

```
pattern='[A-Z][a-z]*'
s='Hello world'
s='HEllo world'
v=re.match(pattern,s)
print(v)
```

执行结果如图 1-12 所示：

图 1-12 示例 1-12 执行结果

【示例 1-13】匹配出有效的变量名

```
import re
pattern='[A-Za-z_][0-9A-Za-z_]*'
print('pattern 为[A-Za-z_][0-9A-Za-z_]*')
s='a'
print('匹配变量名 a 的结果：',re.match(pattern,s))
s='ab'
print('匹配变量名 ab 的结果：',re.match(pattern,s))
s='_ab'
```

```
print('匹配变量名_ab 的结果：',re.match(pattern,s))
s='2ab'
print('匹配变量名 2ab 的结果：',re.match(pattern,s))
print('pattern 为[A-Za-z_]\w*')
pattern='[A-Za-z_]\w*'
s='a'
print('匹配变量名 a 的结果：',re.match(pattern,s))
s='ab'
print('匹配变量名 ab 的结果：',re.match(pattern,s))
s='_ab'
print('匹配变量名_ab 的结果：',re.match(pattern,s))
s='2ab'
print('匹配变量名 2ab 的结果：',re.match(pattern,s))
```

执行结果如图 1-13 所示：

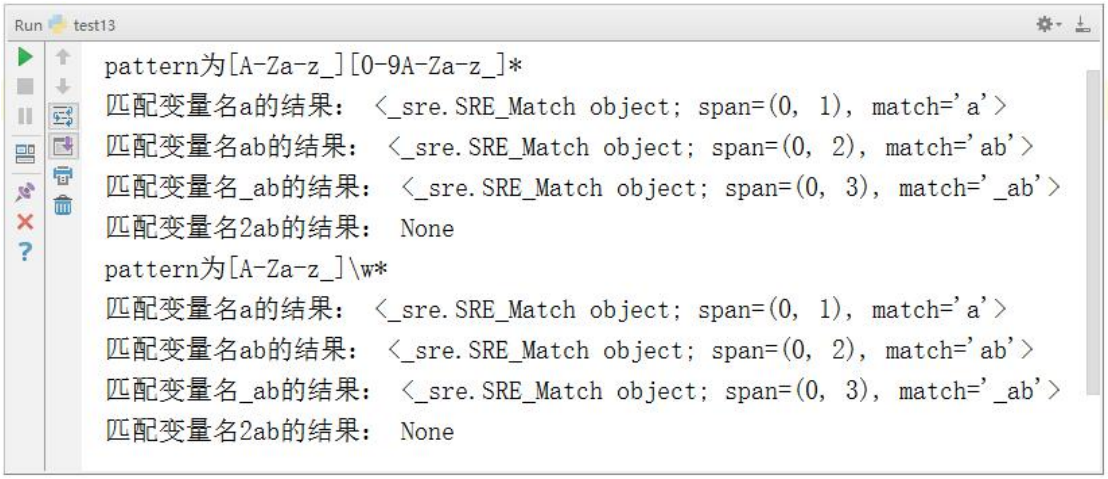


图 1-13 示例 1-13 执行结果

【示例 1-14】匹配出 1-99 直接的数字

```
import re
pattern='[1-9]\d?'
s='1'
print('匹配数字 1：',re.match(pattern,s))
s='55'
print('匹配数字 55：',re.match(pattern,s))
s='99'
```

```
print('匹配数字 99：',re.match(pattern,s))
s='199'
print('匹配数字 199：',re.match(pattern,s))
```

执行结果如图 1-14 所示：

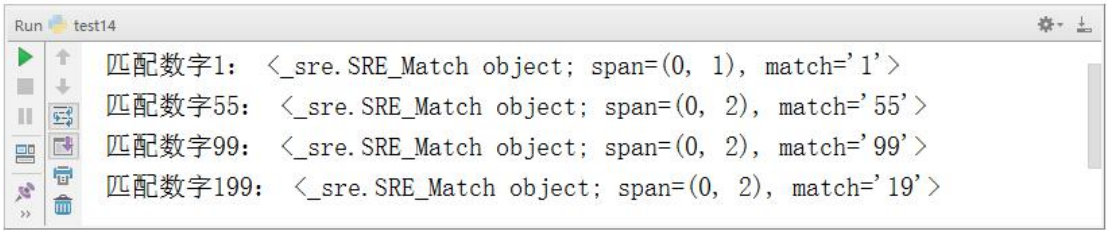


图 1-14 示例 1-14 执行结果

【示例 1-15】匹配出一个随机密码 8-20 位以内 (大写字母 小写字母 下划线 数字)

```
import re
pattern='\w{8,20}'
s='12345678'
print('匹配 12345678 的结果：\n',re.match(pattern,s))
s='123__456'
print('匹配 123__456 的结果：\n',re.match(pattern,s))
```

执行结果如图 1-15 所示：

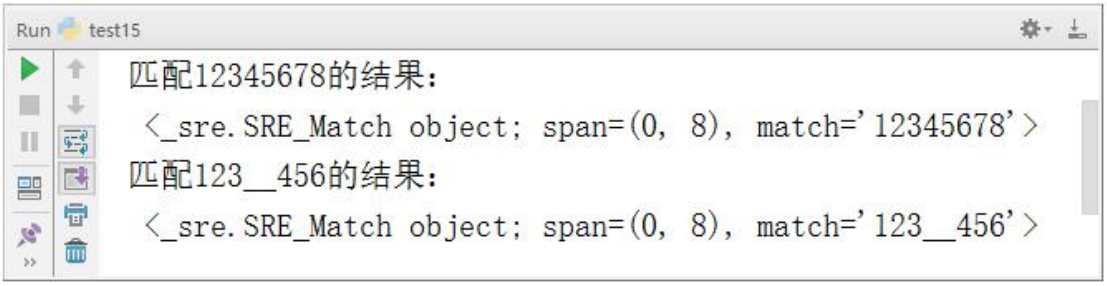


图 1-15 示例 1-15 执行结果

1.2.4 原生字符串

和大多数编程语言相同，正则表达式里使用“\”作为转义字符，这就可以能造成反斜杠困扰。示例如下：

【示例 1-16】“\”作为转义字符

```
s='c:\\a\\b\\c'
print(s)
```

执行结果如图 1-16 所示：

图 1-16 示例 1-16 执行结果

假如你需要匹配文本中的字符“\”，那么使用编程语言表示的正则表达式里将需要 4 个反斜杠“\\\\”：前面两个和后两个分别用于在编程语言里转义成反斜杠，转换成两个反斜杠后再在正则表达式里转义成一个反斜杠。Python 里的原生字符串很好地解决了这个问题，使用 Python 的 r 前缀。例如匹配一个数字的“\\d”可以写成 r“\d”。有了原生字符串，再也不用担心是不是漏写了反斜杠，写出来的表达式也更直观。

【示例 1-17】Python 中的 r 前缀的使用

```
import re
s='c:\\a\\b\\c'
m=re.match('c:\\a\\b',s)
if m is not None:
    print('匹配的结果 1：',m.group())
m=re.match('c:\\\\a\\\\b',s)
if m is not None:
    print('匹配的结果 2：',m.group())
#使用 Python 中的 r 前缀
m=re.match(r'c:\\a\\b',s)
if m is not None:
    print('使用 Python 中的 r 前缀后匹配的结果：',m.group())
```

执行结果如图 1-17 所示：

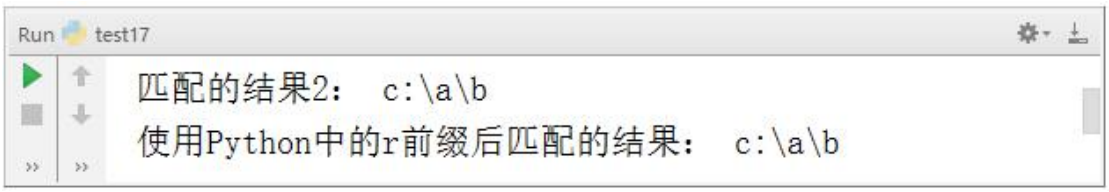


图 1-17 示例 1-17 执行结果

1.2.5 边界字符

正则表达式中有匹配开始或者结束位置的边界字符如表 1-6 所示：

表 1-6 正则表达式中的边界字符

字符	功能
^	匹配字符串开头
$	匹配字符串结尾。
\b	匹配一个单词的边界
\B	匹配非单词的边界

注意：

- ^与[^m]中的“^”的含义并不相同，后者“^”表示“除了….”的意思

【示例 1-18】匹配符$的使用

```
import re
#匹配 qq 邮箱，    5-10 位
print('未限制结尾'.center(30,'-'))
pattern = '[\d]{5,10}@qq.com'
print('正确的邮箱匹配结果：\n',re.match(pattern,'12345@qq.com'))
print('不正确的邮箱匹配结果：\n',re.match(pattern,'12345@qq.comabc'))
print('限制结尾'.center(30,'-'))
pattern = '[1-9]\d{4,9}@qq.com$'
print('正确的邮箱匹配结果：\n',re.match(pattern,'12345@qq.com'))
print('不正确的邮箱匹配结果：\n',re.match(pattern,'12345@qq.comabc'))
```

执行结果如图 1-18 所示：

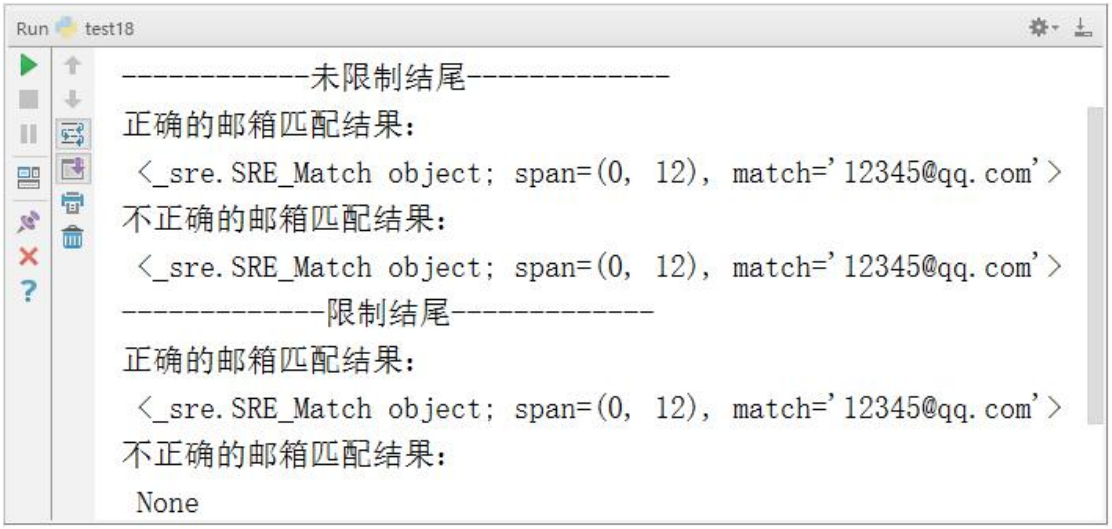


图 1-18 示例 1-18 执行结果

从上面的示例可以看到中如果没有使用结尾$限定符，不正确的邮箱也会匹配成功，更加体现了限定符的重要性。

【示例 1-19】匹配符^的使用

```
import re
s='hello python'
pattern=r'^hello.*'
print('匹配字符串 hello python 的结果：\n',re.match(pattern,s))
s='hepython'
pattern=r'^hello.*'
print('匹配字符串 hepython 的结果：\n',re.match(pattern,s))
```

执行结果如图 1-19 所示：

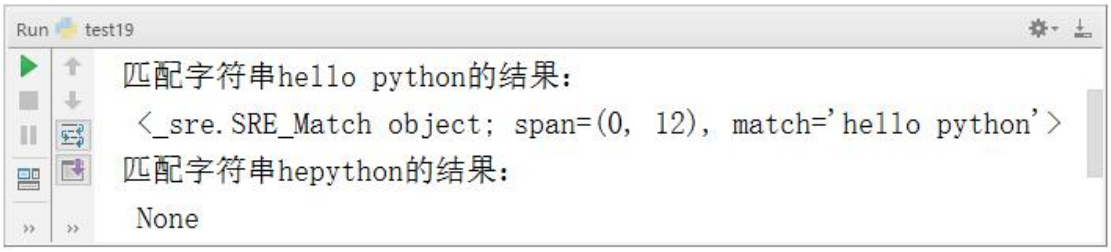


图 1-19 示例 1-19 执行结果

【示例 1-20】\b 匹配单词边界

```
pattern = r'.*\bab'
#ab 左边界的情况
v = re.match(pattern,'123 abr')
print(v)

pattern = r'.*ab\b'
#ab 为右边界的情况
v = re.match(pattern,'wab')
print(v)
```

执行结果如图 1-20 所示：

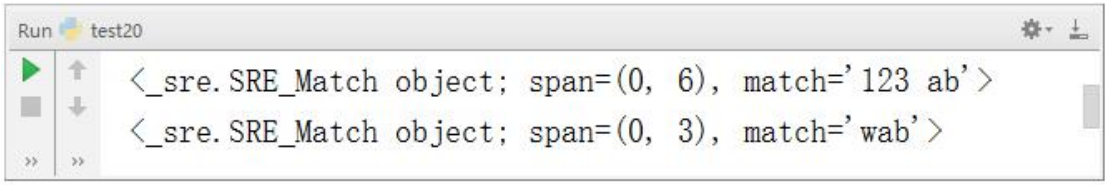


图 1-20 示例 1-20 执行结果

【示例 1-21】\B 匹配非单词边界

```
#ab 不为左边界
pattern = r'.*\Bab'
v = re.match(pattern,'123 abr')
print(v)
#ab 不为右边界
pattern = r'.*ab\B'
v = re.match(pattern,'wab')
print(v)
```

执行结果如图 1-21 所示：

图 1-21 示例 1-21 执行结果

1.2.6 search 方法

search 在一个字符串中搜索满足文本模式的字符串。语法格式如下：


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

函数参数与 match 方法类似，如表 1-7 所示：

表 1-7 search 函数参数说明

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，多行匹配等等。 如下表列出正则表达式修饰符 - 可选标志

【示例 1-22】search 方法的使用

```
import re
m=re.search('abc','abcdefg')
print(m)
print(m.group())
```

执行结果如图 1-22 所示：

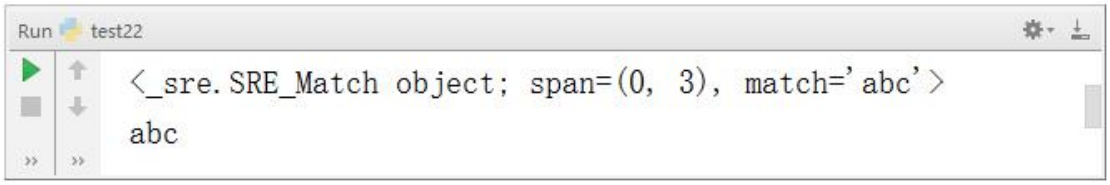


图 1-22 示例 1-22 执行结果

1.2.7 match 与 search 的区别

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。

【示例 1-23】match 方法与 search 方法的使用对比

```
import re
#进行文本模式匹配，匹配失败，match 方法返回 None
m=re.match('love','I love you')
if m is not None:
    print(m.group())
print('match 运行结果：',m)
#进行文本模式搜索，
m=re.search('love','I love you')
if m is not None:
    print(m.group())
print('search 的运行结果：',m)
```

执行结果如图 1-23 所示：

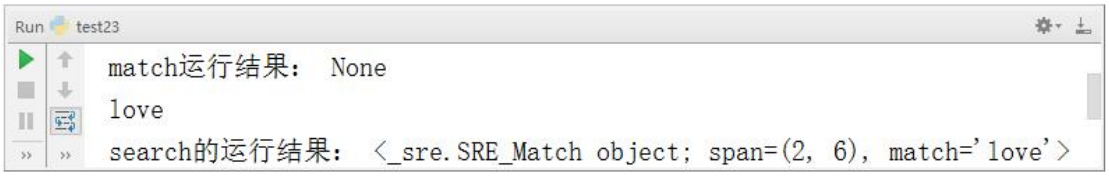


图 1-23 示例 1-23 执行结果

1.2.8 匹配多个字符串

search 方法搜索一个字符串，要想搜索多个字符串，如搜索 aa、bb 和 cc，最简单的方法是在文本模式字符串中使用择一匹配符号（|）。择一

匹配符号和逻辑或类似，只要满足任何一个，就算匹配成功。

【示例 1-24】择一匹配符号（|）的使用

```
import re
s='aa|bb|cc'
#match 进行匹配
m=re.match(s,'aa')     #aa 满足要求，匹配成功
print(m.group())
m=re.match(s,'bb')     #bb 满足要求，匹配成功
print(m.group())
#search 查找
m=re.search(s,'Where is cc')
print(m.group())
```

执行结果如图 1-24 所示：

图 1-24 示例 1-24 执行结果

从上面的代码可以看出，待匹配的字符串只要是 aa、bb 和 cc 中的任何一个就会匹配成功。

【示例 1-25】匹配 0-100 之间所有的数字

```
import re
pattern = '[1-9]?\d$|100$'
print(re.match(pattern,'0'))
print(re.match(pattern,'10'))
print(re.match(pattern,'100'))
print(re.match(pattern,'99'))
print(re.match(pattern,'200'))
```

执行结果如图 1-25 所示：

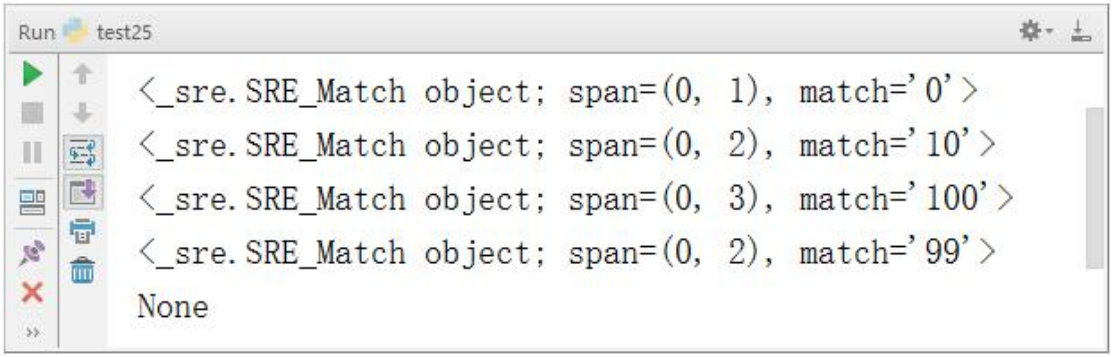


图 1-25 示例 1-25 执行结果

如果待匹配的字符串中，某些字符可以有多个选择，就需要使用字符集（[]），也就是一对中括号括起来的字符串。例如，[xyz]表示 x、y、z 三个字符可以取其中任何一个，相当于“x|y|z”，所以对单个字符使用或关系时，字符集和择一匹配符的效果是一样的。示例如下：

【示例 1-26】字符集（[]）和择一匹配符(|)完成相同的效果

```
import  re
m=re.match('[xyz]','x')  #匹配成功
print(m.group())
m=re.match('x|y|z','x')  #匹配成功
print(m.group())
```

执行结果如图 1-26 所示：

图 1-26 示例 1-26 执行结果

【示例 1-27】字符集（[]）和择一匹配符(|)的用法，及它们的差异

```
import re
#匹配以第 1 个字母是 a 或者 b，第 2 个字母是 c 或者 d，如 ac、bc、ad、bd
m=re.match('[ab][cd]','aceg')
print(m)
#匹配以 ab 开头，第 3 个字母是 c 或者 d，如 abc、abd
m=re.match('ab[cd]','abcd')
print(m)
```

```
#匹配 ab 或者 cd
m=re.match('ab|cd','cd')
print(m)
```

执行结果如图 1-27 所示：

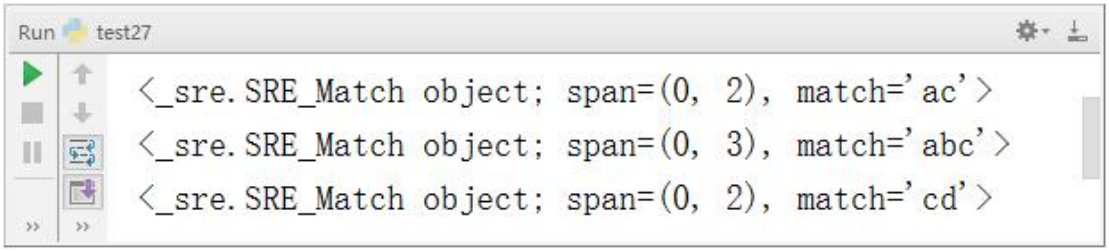


图 1-27 示例 1-27 执行结果

1.2.9 分组

如果一个模式字符串中有用一对圆括号括起来的部分，那么这部分就会作为一组，可以通过 group 方法的参数获取指定的组匹配的字符串。当然，如果模式字符串中没有任何用圆括号括起来的部分，那么就不会对待匹配的字符串进行分组。

表 1-8 正则表达式中的分组字符

字符	功能
(ab)	将括号中的字符作为一个分组
\num	引用分组 num 匹配到的字符串
(?p<name>)	分别起组名
(?p=name)	引用别名为 name 分组匹配到的字符串

【示例 1-28】匹配座机号码

```
pattern = r'(\d+)-(\d{5,8}$)'
v = re.match(pattern,'010-66668888')
print(v)
print(v.group())
print(v.group(1))
print(v.group(2))
print(v.groups())
print(v.groups()[0])
```

```
print(v.groups()[1])
```

执行结果如图 1-28 所示：

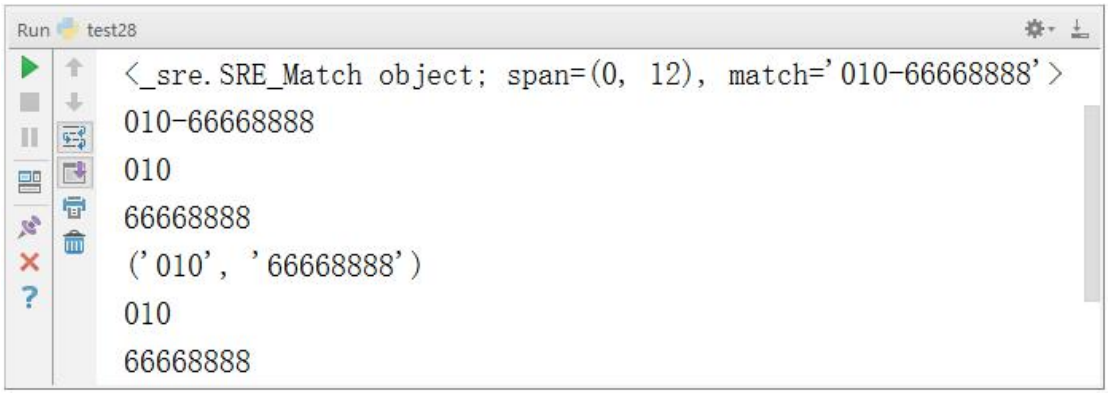


图 1-28 示例 1-28 执行结果

【示例 1-29】\num 的使用

```
import re
#匹配合法的网页标签
s = '<html><title>我是标题</title></html>'
#匹配不合法的网页标签
ss = '<html><title>我是标题</html></title>'
#优化前
pattern = r'<.+><.+>.+</.+></.+>'
print(re.match(pattern,s))
print(re.match(pattern,ss))

#优化后 可以使用分组 \2 表示引用第 2 个分组 \1 表示引用第 1 个分组
pattern = r'<(.+)><(.+)>.+</\2></\1>'
print(re.match(pattern,s))
print(re.match(pattern,ss))
```

执行结果如图 1-29 所示：

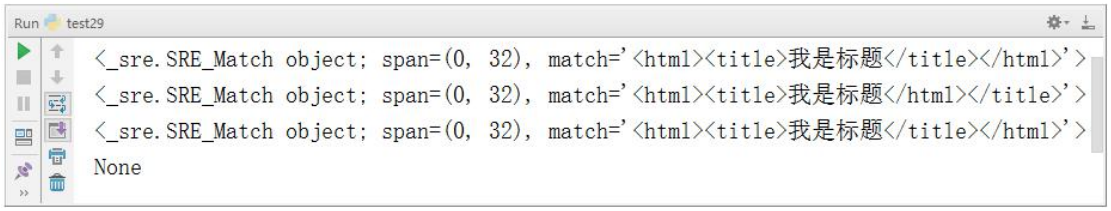


图 1-29 示例 1-29 执行结果

从上面示例可以看出如果不使用分组，不合法的网页标签都可以匹配成功。

【示例 1-30】?P<要起的别名> (?P=起好的别名)

```
s = '<html><h1>我是一号字体</h1></html>'
# pattern = r'<(.+)><(.+)>.+</\2></\1>'
#如果分组比较多的话，数起来比较麻烦，可以使用起别名的方法?P<要起的名字>  以及使用别名(?P=之前起的别名)
pattern = r'<(?P<key1>.+)><(?P<key2>.+)>.+</(?P=key2)></(?P=key1)>'
v = re.match(pattern,s)
print(v)
```

执行结果如图 1-30 所示：

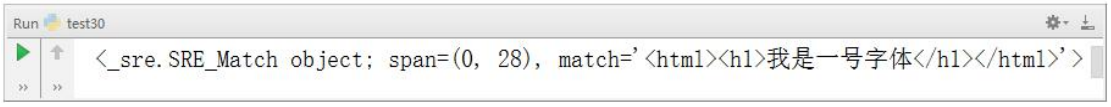


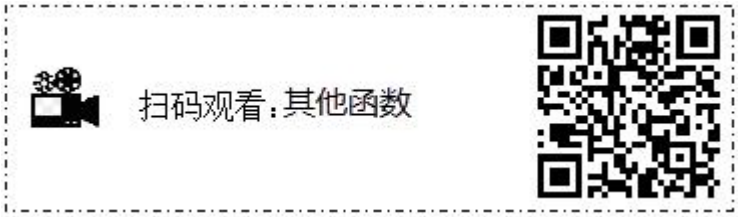
图 1-30 示例 1-30 执行结果

使用分组要了解如下几点：

- 只有圆括号括起来的部分才算一组，如果模式字符串中既有圆括号括起来的部分，也有没有被圆括号括起来的部分，那么只会将被圆括号括起来的部分算作一组，其它的部分忽略。
- 用 group 方法获取指定组的值时，组从 1 开始，也就是说，group(1)获取第 1 组的值，group(2)获取第 2 组的值，以此类推。
- groups 方法用于获取所有组的值，以元组形式返回。所以除了使用 group(1)获取第 1 组的值外，还可以使用 groups()[0]获取第 1 组的值。获取第 2 组以及其它组的值的方式类似。

1.3 re 模块中其他常用的函数

1.3.1sub 和 subn 搜索与替换



sub 函数和 subn 函数用于实现搜索和替换功能。这两个函数的功能几乎完全相同，都是将某个字符串中所有匹配正则表达式的部分替换成其他字符串。用来替换的部分可能是一个字符串，也可以是一个函数，该函数返回一个用来替换的字符串。sub 函数返回替换后的结果，subn 函数返回一个元组，元组的第 1 个元素是替换后的结果，第 2 个元素是替换的总数。语法格式如下：

```
re.sub(pattern, repl, string, count=0, flags=0)
```

参数说明：

表 1-9 sub 函数参数说明

参数	描述
pattern	匹配的正则表达式
repl	替换的字符串，也可为一个函数
string	要被查找替换的原始字符串。
count	模式匹配后替换的最大次数，默认 0 表示替换所有的匹配

【示例 1-31】sub 和 subn 方法的使用

```
import re
phone = "2004-959-559 #  这是一个国外电话号码"
#  删除字符串中的  Python 注释
num = re.sub(r'#.*$', "", phone)
print("电话号码是: ", num)
#  删除非数字(-)的字符串
num = re.sub(r'\D', "", phone)
print("电话号码是  : ", num)
#subn 函数的使用
result=re.subn(r'\D', "", phone)
print(result)
print('替换的结果：',result[0])
print('替换的次数：',result[1])
```

在上执行结果如图 1-31 所示：

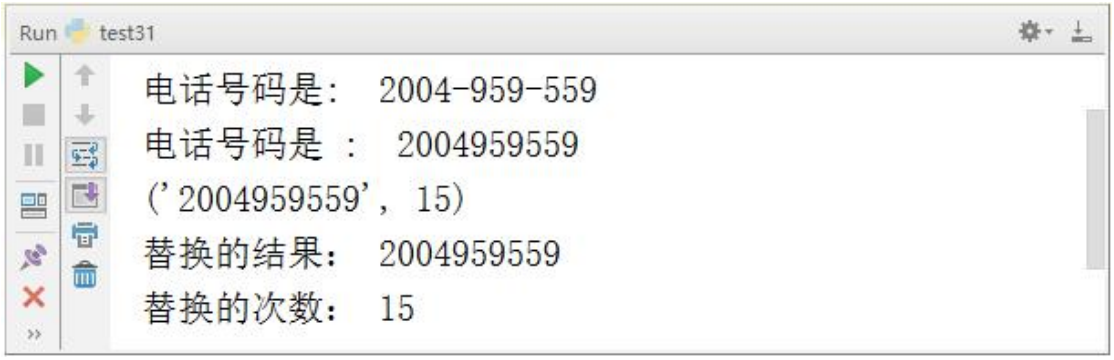


图 1-31 示例 1-31 执行结果

1.3.2 compile 函数

compile 函数用于编译正则表达式，生成一个正则表达式（ Pattern ）对象，供 match() 和 search() 这两个函数使用。语法格式为：

```
re.compile(pattern[, flags])
```

参数说明：

表 1-10 compile 函数参数说明

参数	描述
pattern	一个字符串形式的正则表达式
flags	可选，表示匹配模式，比如忽略大小写，多行模式等，

【示例 1-32】compile 函数的使用

```
import re
s='first123 line'
regex=re.compile(r'\w+') #匹配至少一个字母或数字
m=regex.match(s)
print(m.group())
# s 的开头是 "f", 但正则中限制了开始为 i 所以匹配失败
regex = re.compile("^i\w+")
print(regex.match(s))
```

在上执行结果如图 1-32 所示：

图 1-32 示例 1-32 执行结果

1.3.3 findall 函数

在字符串中找到正则表达式所匹配的所有子串，并返回一个列表，如果没有找到匹配的，则返回空列表。语法格式如下：

```
findall(pattern, string, flags=0)
```

表 1-11 findall 函数参数说明

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，多行匹配等等。 如下表列出正则表达式修饰符 - 可选标志

【示例 1-33】findall 函数的使用

```
import re
pattern=r'\w+'
s='first 1 second 2 third 3'
o=re.findall(pattern,s)
print(o)
```

在上执行结果如图 1-33 所示：

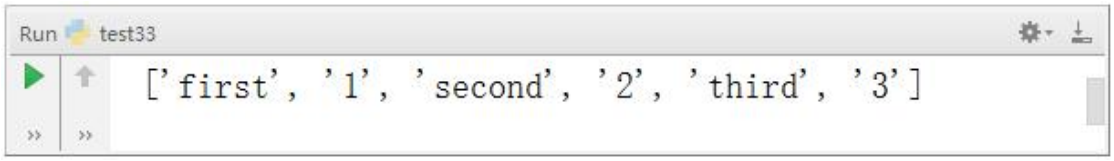


图 1-33 示例 1-33 执行结果

注意：

- match 和 search 是匹配一次 findall 匹配所有

1.3.4 finditer 函数

和 findall 类似，在字符串中找到正则表达式所匹配的所有子串，并把它们作为一个迭代器返回。

【示例 1-34】finditer 函数的使用

```
pattern=r'\w+'
s='first 1 second 2 third 3'
o=re.finditer(pattern,s)
print(o)
for i in o:
    print(i.group())
```

在上执行结果如图 1-34 所示：

图 1-34 示例 1-34 执行结果

1.3.5 split 函数

split 函数用于根据正则表达式分隔字符串，也就是说，将字符串与模式匹配的子字符串都作为分隔符来分隔这个字符串。split 函数返回一个列表形式的分隔结果，每一个列表元素都是分隔的子字符串。语法格式如下：

```
re.split(pattern, string[, maxsplit=0, flags=0])
```

参数说明：

表 1-12 split 函数参数说明

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
maxsplit	分隔次数，maxsplit=1 分隔一次，默认为 0，不限制次数。
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，多行匹配等等。

【示例 1-35】split 函数的使用

```
import re
s='first 11 second 22 third 33'
#按数字切分
print(re.split(r'\d+',s))
# maxsplit 参数限定分隔的次数，这里限定为 1，也就是只分隔一次
print(re.split(r'\d+',s,1))
```

在上执行结果如图 1-35 所示：

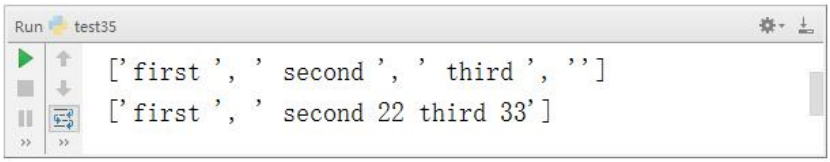
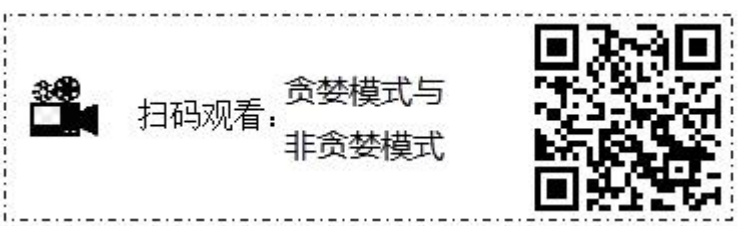


图 1-35 示例 1-35 执行结果

1.4 贪婪模式和非贪婪

贪婪模式指 Python 里数量词默认是贪婪的，总是尝试匹配尽可能多的字符。非贪婪模式与贪婪相反，总是尝试匹配尽可能少的字



符，可以使用"*"，"?"，"+"，"{m,n}"后面加上？，使贪婪变成非贪婪。

【示例 1-36】贪婪模式，.+中的'.'会尽量多的匹配

```
v = re.match(r'(.+)(\d+-\d+-\d+)','This is my tel:133-1234-1234')
print('----------贪婪模式---------')
print(v.group(1))
print(v.group(2))
print('----------非贪婪模式---------')
v = re.match(r'(.+?)(\d+-\d+-\d+)','This is my tel:133-1234-1234')
print(v.group(1))
print(v.group(2))
```

执行结果如图 1-36 所示：

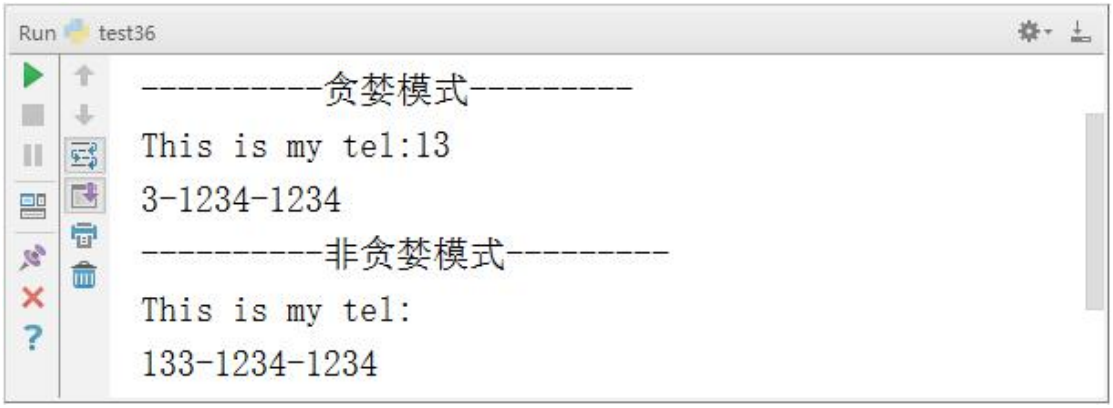


图 1-36 示例 1-36 执行结果

【示例 1-37】贪婪模式非贪婪模式测试

```
print('贪婪模式')
v= re.match(r'abc(\d+)','abc123')
print(v.group(1))
#非贪婪模式
print('非贪婪模式')
v= re.match(r'abc(\d+?)','abc123')
print(v.group(1))
```

执行结果如图 1-37 所示：

图 1-37 示例 1-37 执行结果

习题

一、选择题

1. Python 正则表达式中，____可以表示 0 个或者 1 个。

A. "+"

B. "^"

C. "？"

D. "*"

2. 什么是正则表达式？

A. 任何在 Python 中可能的正确表达式

B. 程序员经常使用的编程语言表达式的集合

C. 用来匹配文本字符串（如特定字符、单词或字符模式）的一种工具

D. 从 Python 字典中快速获取数据的一种算法

3. 给定 re.match('.',s)表达式，不满足此匹配条件的 s 值（）：

A. '\n'

B. 'a'

C. 'C'

D. '_'

4. 给定 re.match('\D',s)表达式，不满足此匹配条件的 s 值（）：

A. 'a'

B. 6

C. 'C'

D. '_'

5. 执行如下代码输出的结果是：（）

```
pattern = r'(\d+)-(\d{5,8}$)'
v = re.match(pattern,'010-66668888')
print(v.groups()[1])
```

A. 010

B. 66668888

C. 010-66668888

D. 出现异常

二、解答题

1. 正则表达式的应用范围。
2. \d，\w,\s,[a-zA-Z0-9],\b,.,*,+,?,x{3},^,$分别是什么？
3. re 模块中常用函数有哪些？
4. re 模块中 match 和 search 的区别。

5. 什么是贪婪模式和非贪婪模式？

三、编码题

1. 写一个正则表达式，使其能同时识别下面所有的字符串：'bat','bit', 'but', 'hat', 'hit', 'hut'。

2. 匹配所有的有效的 Python 标识符集合。

3. 将每行中的电子邮件地址替换为你自己的电子邮件地址。

第二章 多线程和并发编程

如果一个程序在同一时间只能做一件事情，它就是单线程程序，这类程序的功能会显得过于简单，肯定无法满足现实的需求。在本章将从多进程、多线程方面讲解并发运行的使用，多线程程序的功能更加强大，能够满足现实生活中需求多变的情况。

通过阅读本章，你可以：

- 了解多任务、进程和线程的概念，以及进程和线程的区别
- 掌握进程间通信
- 掌握 Python 中的多线程的使用
- 掌握 threading 模块中的 Thread 类的使用方法
- 掌握如何利用线程锁让代码同步
- 掌握生产者、消费者模型以及实现方法

2.1 多任务

在现实生活中，有很多场景中的事情是同时进行的。例如：开车的时候，手和脚需要共同操作来完成驾驶。再比如演唱会中唱歌和跳舞也是同时进行的。如果把唱歌和跳舞这两件事情分开依次来完成，估计效果不是很好。示例代码如下：

【示例 2-1】模拟唱歌跳舞

```
from time import sleep
def sing():
    for i in range(3):
        print('正在唱歌...%d'%i)
        sleep(1)

def dance():
    for i in range(3):
        print('正在跳舞...%d'%i)
        sleep(1)

if __name__=='__main__':
    sing()
    dance()
```

执行结果如图 2-1 所示：

图 2-1 示例 2-1 执行结果

从上面的示例可以看出，程序并没有完成唱歌和跳舞同时进行的要求，如果想要实现“唱歌和跳舞”同时进行，那么就需要一个新的方法，叫做：多任务。

什么叫做多任务呢？简单的说，就是操作系统可以同时进行多个任务。例如：你一边在使用浏览器上网，一边在听 MP3，一边在用 word 赶作业，这就是多任务。

图 2-2 多任务

现在，多核 CPU 已经非常普及了，但是，即使过去的单核 CPU，也可以执行多任务。由于 CPU 执行代码都是顺序执行的，那么，单核 CPU 是怎么执行多任务的呢？

操作系统轮流让各个任务交替执行，任务 1 执行 0.01 秒，切换到任务 2，任务 2 执行 0.01 秒，再切换到任务 3，执行 0.01 秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于 CPU 的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核 CPU 上实现，但是，由于任务数量远远多于 CPU 的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

2.2 线程与进程的概念

2.2.1 进程

计算机程序有静态和动态的区别，静态的计算机程序就是存储在磁盘上的可执行二进制(或其他型)文件，而动态的计算机程序就是将这些可执行文件加载到内存中并被操作系统调用，这些动态的计算程序称为一个进程。

现代的操作系统都可以同时启动多个进程。比如：我们在用酷狗听音乐，也可以使用Pycharm 写代码，也可以同时用浏览器查看网页。进程具有如下特点：

- 进程是程序的一次动态执行过程， 占用特定的地址空间。
- 每个进程由 3 部分组成：cpu、data、code。每个进程都是独立的，保有自己的 cpu 时间，代码和数据，即便用同一份程序产生好几个进程，它们之间还是拥有自己的这 3 样东西，这样的缺点是：浪费内存，cpu 的负担较重。
- 多任务(Multitasking)操作系统将 CPU 时间动态地划分给每个进程，操作系统同时执行多个进程，每个进程独立运行。以进程的观点来看，它会以为自己独占 CPU 的使用权。
- 进程的查看
  - Windows 系统: Ctrl+Alt+Del，启动任务管理器即可查看所有进程。
  - Unix 系统: ps or top。

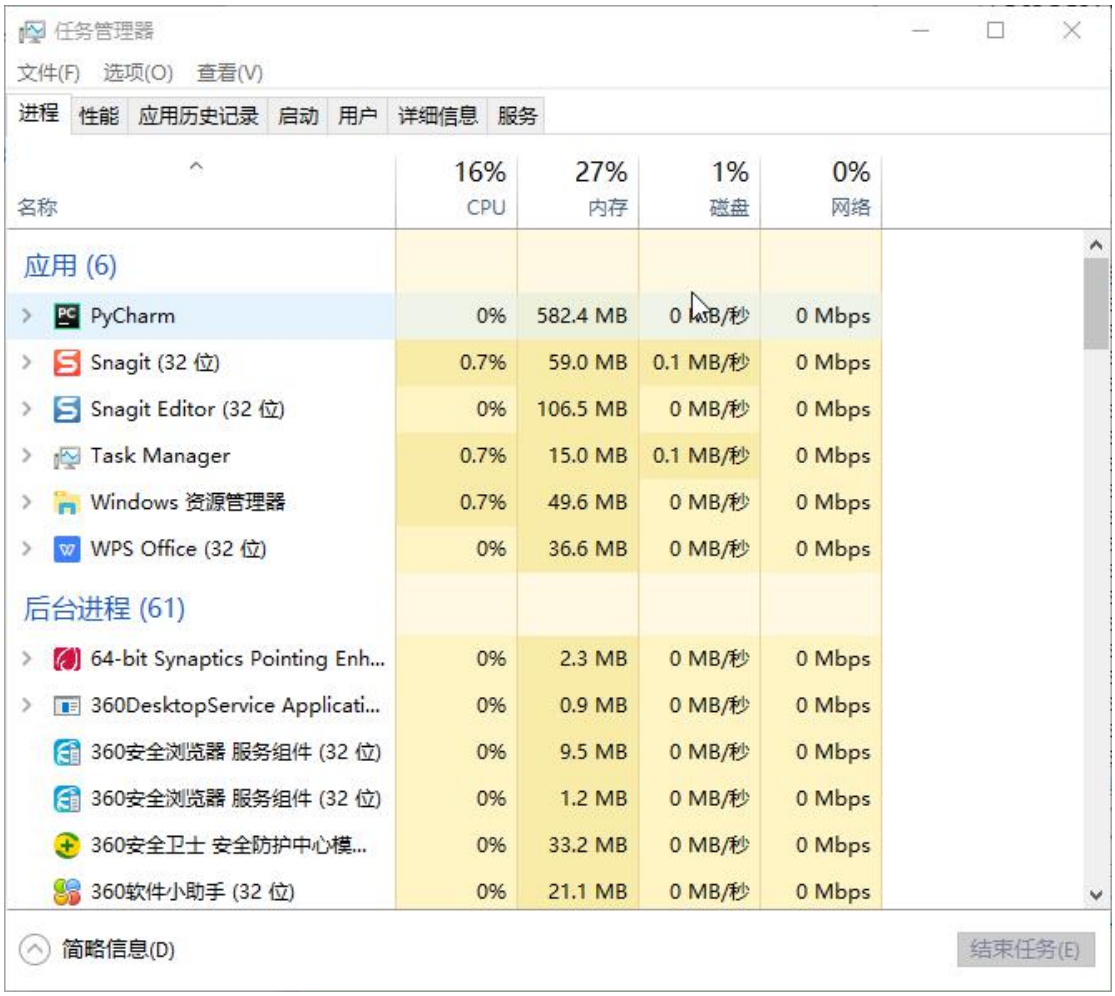


图 2-3 Windows 下查看进程

2.2.2 线程

一个进程可以产生多个线程。同多个进程可以共享操作系统的某些资源一样，同一进程的多个线程也可以共享此进程的某些资源（比如：代码、数据），所以线程又被称为轻量级进程(lightweight process)。

- ❑ 一个进程内部的一个执行单元，它是程序中的一个单一的顺序控制流程。
- ❑ 一个进程可拥有多个并行的(concurrent)线程。
- ❑ 一个进程中的多个线程共享相同的内存单元/内存地址空间，可以访问相同的变量和对象，而且它们从同一堆中分配对象并进行通信、数据交换和同步操作。
- ❑ 由于线程间的通信是在同一地址空间上进行的，所以不需要额外的通信机制，这就使得通信更简便而且信息传递的速度也更快。
- ❑ 线程的启动、中断、消亡，消耗的资源非常少。

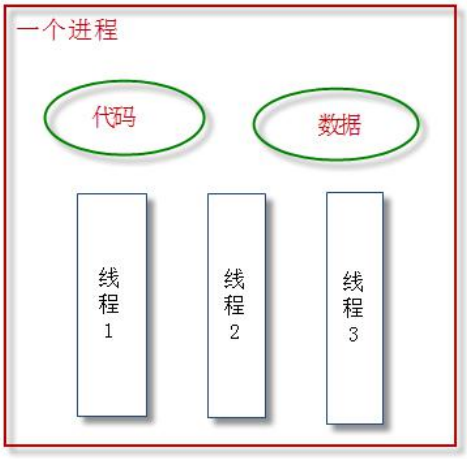
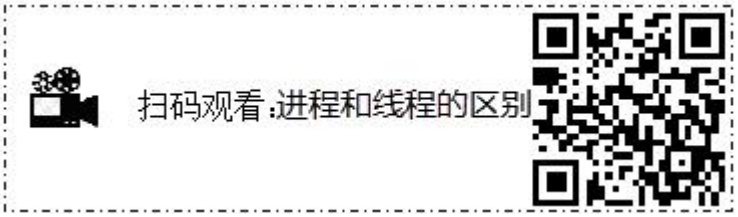


图 2-4 线程共享资源示意图

2.2.3 线程和进程的区别

- 每个进程都有独立的代码和数据空间(进程上下文)，进程间的切换会有较大的开销。

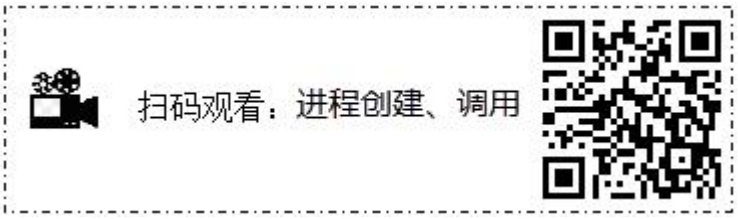


- 线程可以看成是轻量级的进程，属于同一进程的线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换的开销小。
- 线程和进程最根本的区别在于:进程是资源分配的单位,线程是调度和执行的单位。
- 多进程: 在操作系统中能同时运行多个任务(程序)。
- 多线程: 在同一应用程序中有多个顺序流同时执行。
- 线程是进程的一部分，所以线程有的时候被称为轻量级进程。
- 一个没有线程的进程是可以被看作单线程的，如果一个进程内拥有多个线程，进程的执行过程不是一条线（线程）的，而是多条线（线程）共同完成的。
- 系统在运行的时候会为每个进程分配不同的内存区域，但是不会为线程分配内存（线程所使用的资源是它所属的进程的资源），线程组只能共享资源。那就是说，除了 CPU 之外（线程在运行的时候要占用 CPU 资源），计算机内部的软硬件资源的分配与线程无关，线程只能共享它所属进程的资源。

2.3 进程

程序编写完没有运行称之为程序。正在运行的代码就是进程。在 Python3 语言中，对多进程支持的是 multiprocessing 模块和 subprocess 模块。multiprocessing 模块为在子进程中运行任务、通讯和共享数据，以及执行各种形式的同步提供支持。

2.3.1 进程创建



Python 提供了非常好用的多进程包 multiprocessing，只需要定义一个函数，Python 会完成其他所有事情。借助这个包，可以轻松完成从单进程到并发执行的转换。multiprocessing 支持子进程、通信和共享数据。语法格式如下：

```
Process（[group [, target [, name [, args [, kwargs]]]]]）
```

其中 target 表示调用对象，args 表示调用对象的位置参数元组。kwargs 表示调用对象的字典。name 为别名。group 参数未使用，值始终为 None。

构造函数简单地构造了一个 Process 进程，Process 的实例方法、Process 的实例属性如下表 2-1 和表 2-2 所示：

表 2-1 Process 实例方法表

方法	描述
is_alive()	如果 p 仍然运行，返回 True
join（[timeout]）	等待进程 p 终止。Timeout 是可选的超时期限，进程可以被链接无数次，但如果连接自身则会出错
run()	进程启动时运行的方法。默认情况下，会调用传递给 Process 构造函数的 target。定义进程的另一种方法是继承 Process 类并重新实现 run()函数
start()	启动进程，这将运行代表进程的子进程，并调用该子进程中的 run()函数
terminate()	强制终止进程。如果调用此函数，进程 p 将被立即终止，同时不会进行任何清理动作。如果进程 p 创建了它自己的子进程，这些进程将变为僵尸进程。使用此方法时需要特别小心。如果 p 保存了一个锁或参与了进程间通信，那么终止它可能会导致死锁或 I/O 损坏

【示例 2-2】创建函数并将其作为单个进程

```
from multiprocessing import Process
#定义子进程代码
def run_proc():
    print('子进程运行中')


if __name__=='__main__':
    print('父进程运行')
```

```
    p=Process(target=run_proc)
    print('子进程将要执行')
    p.start()
```

执行结果如图 2-5 所示：

图 2-5 示例 2-2 执行结果

【示例 2-3】创建子进程，传递参数

```
from multiprocessing import Process
import os
from time import sleep
#创建子进程代码
def run_proc(name,age,**kwargs):
    for i in range(5):
        print('子进程运行中，参数 name：%s,age:%d'%(name,age))
        print('字典参数 kwargs：',kwargs)
        sleep(0.5)

if __name__=='__main__':
    print('主进程开始运行')
    p=Process(target=run_proc,args=('test',18),kwargs={'m':23})
    print('子进程将要执行')
    p.start()
```

执行结果如图 2-6 所示：

图 2-6 示例 2-3 执行结果

【示例 2-4】进程中 join()方法的使用

```
from multiprocessing import Process
from time import sleep
def worker(interval):
    print("work start");
    sleep(interval)
    print("work end");
if __name__ == "__main__":
    p = Process(target = worker, args = (3,))
    p.start()
    #等待进程 p 终止
    p.join()
    print("主进程结束!")
```

执行结果如图 2-7 所示：

图 2-7 示例 2-4 执行结果

【示例 2-5】进程中 join()方法加超时的使用

```
from multiprocessing import Process
from time import sleep
def worker(interval):
    print("work start");
    sleep(interval)
    print("work end");

if __name__ == "__main__":
    p = Process(target = worker, args = (5,))
    p.start()
    #等待进程 p 终止
    p.join(3)
    print("主进程结束!")
```

执行结果如图 2-8 所示：

图 2-8 示例 2-5 执行结果

表 2-2 Process 实例属性表

方法	描述
authkey	进程的身份证验证键。除非显示设定，这是由 os.urandom()函数生成的 32 自负的字符串。这个键的用途是涉及网络连接的底层进程间通信提供安全性。这类连接只有在两端具有相同的身份验证键时才能成功
daemon	一个布尔标志，指示进程是否是后台进程。当创建它的 Python 进程终止时，后台(Daemonic)进程将自动终止。另外，禁止后台进程创建自己的新进程。p.daemon 的值必须在使用 p.start()函数启动进程之前进行设置
exitcode	进程的整数退出代码。如果进程仍然在运行，它的值为 None。如果值为负数，-N 表示进程由信号 N 所终止
name	进程的名称
pid	进程的整数进程 ID

【示例 2-6】进程属性的使用

```
#导入模块
import multiprocessing
import time
#定义进程执行函数
def clock(interval):
    for i in range(5):
        print('当前时间为{0}：'.format(time.ctime()))
        time.sleep(interval)
if __name__=='__main__':
    #创建进程
    p=multiprocessing.Process(target=clock,args=(1,))
    #启动进程
    p.start()
    p.join()
    #获取进程的 ID
    print('p.id:',p.pid)
    #获取进程的名称
    print('p.name:',p.name)
    #判断进程是否运行
    print('p.is_alive:',p.is_alive())
```

执行结果如图 2-9 所示：

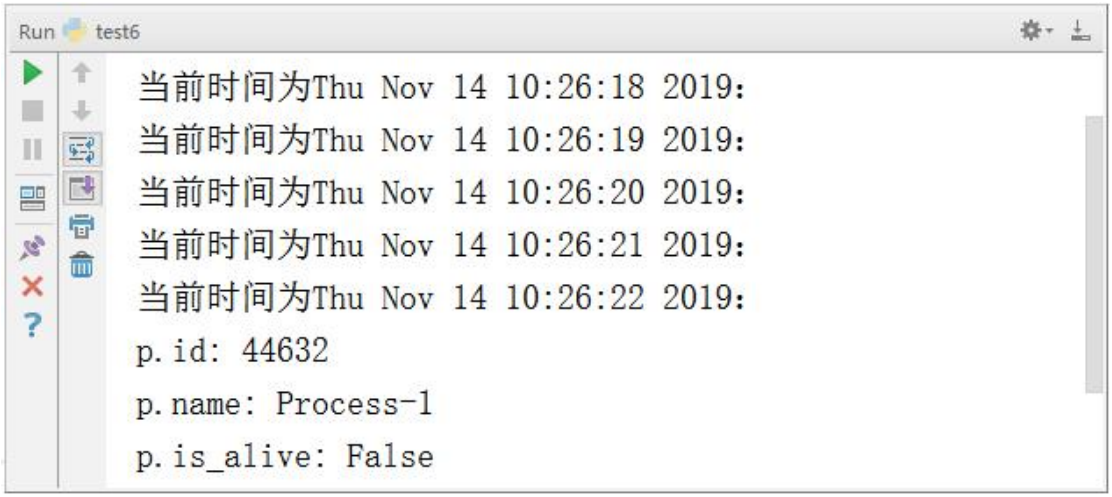


图 2-9 示例 2-6 执行结果

【示例 2-7】创建函数并将其作为多个进程

```
#导入模块
import multiprocessing
import time
#创建进程调用函数
def work1(interval):
    print('work1')
    time.sleep(interval)
    print('end work1')
def work2(interval):
    print('work2')
    time.sleep(interval)
    print('end work2')
def work3(interval):
    print('work3')
    time.sleep(interval)
    print('end work3')
if __name__=='__main__':
    #创建进程对象
    p1=multiprocessing.Process(target=work1,args=(4,))
    p2=multiprocessing.Process(target=work2,args=(3,))
    p3=multiprocessing.Process(target=work3,args=(2,))
    #启动进程
    p1.start()
    p2.start()
    p3.start()
    p1.join()
    p2.join()
    p3.join()
    print('主进程结束')
```

执行结果如图 2-10 所示：

图 2-10 示例 2-7 执行结果

2.3.2 进程的创建-Process 子类

创建进程的方式还可以使用类的方式，可以自定义一个类，继承 Process 类，每次实例化这个类的时候，就等同于实例化一个进程对象。

【示例 2-8】继承 Process 的类，重写 run()方法创建进程

```
#导入模块
from multiprocessing import Process
import time
#定义线程类
class ClockProcess(Process):
    def __init__(self,interval):
        Process.__init__(self)
        self.interval=interval
    def run(self):
        print('子进程开始执行的时间:{}'.format(time.ctime()))
        time.sleep(self.interval)
        print('子进程结束的时间:{}'.format(time.ctime()))

if __name__=='__main__':
    #创建进程
    p=ClockProcess(2)
    #启动进程
```

```
    p.start()
    p.join()
    print('主进程结束')
```

执行结果如图 2-11 所示：

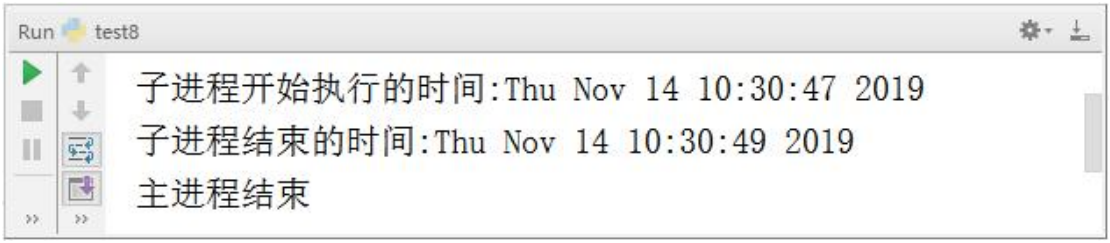


图 2-11 示例 2-8 执行结果

2.3.3 进程池

在利用 Python 进行系统管理的时候，特别是同时操作多个文件目录，或者远程控制多台主机，并行操作可以节约大量的时间。当被操作对象数目不大时，可以直接利用 multiprocessing 中的 Process 动态成生多个进程，十几个还好，但如果是上百个，上千个目标，手动的去限制进程数量却又太过繁琐，此时可以发挥进程池的功效。

Pool 可以提供指定数量的进程，供用户调用，当有新的请求提交到 pool 中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求。但如果池中的进程数已经达到规定最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程。Pool 的语法格式如下：

```
Pool（[numprocess [, initializer [, initargs]]]）
```

其中 numprocess 是要创建的进程数。如果省略此参数，将使用 cpu_count()的值。Initializer 是每个工作进程启动时要执行的可调用对象。Initargs 是要传递给 initializer 的参数元祖。Initializer 默认为 None。Pool 类的实例方法如表 2-3 所示：

表 2-3 Pool 实例方法表

方法	描述
apply（func [,args [,kwargs]]）	在一个池工作进程中执行函数（*args，**kwargs），然后返回结果。
apply_async（func [, args [,kwargs [,callback]]]）	在一个池工作进程中异步地执行函数（*args，**kwargs），然后返回结果。此方法的结果是 AsyncResult 类的实例，稍后可用于获得最终结果。Callback 是可调用对象，接受输入参数。当 func 的结果变为可用时，将立即传递给 callback。Callback 禁止执行任何阻塞操作，否则将阻塞接收其他异步操作中的结果

方法	描述
close()	关闭进程池，防止进行进一步操作。如果还有挂起的操作，它们将在工作进程终止之前完成
join()	等待所有工作进程退出。此方法只能在 close（）或者 terminate（）方法之后调用
imap(func , iterable [,chunksize])	map（）函数的版本之一，返回迭代器而非结果列表
imap_unordered(func，iterable [, chunksize])	同 imap（）函数一样，只是结果的顺序根据从工作进程接收到的时间任意确定
map(func , iterable [, chunksize])	将可调用对象 func 应用给 iterable 中的所有项，然后以列表的形式返回结果。通过将 iterable 划分为多块并将工作分派给工作进程，可以并行地执行这项操作。chunksize 指定每块中的项数。如果数量较大，可以增大 chunksize 的值来提升性能
map_async(func , iterable [, chunksize [, callback]])	同 map（）函数，但结果的返回是异步的。返回值是 AsyncResult 类的实例，稍后可用与获取结果。Callback 是指接受一个参数的可调对象。如果提供 callable，当结果变为可用时，将使用结果调用 callable
terminate()	立即终止所有工作进程,同时不执行任何清理或结束任何挂起工作。如果 p 被垃圾回收，将自动调用此函数
get([timeout])	返回结果，如果有必要则等待结果到达。Timeout 是可选的超时。如果结果在指定时间内没有到达，将引发 multiprocessing.TimeoutError 异常。如果远程操作中引发了异常，它将在调用此方法时再次被引发
ready()	如果调用完成，则返回 True
sucessful()	如果调用完成且没有引发异常，返回 True。如果在结果就绪之前调用此方法，将引发 AssertionError 异常
wait([timeout])	等待结果变为可用。Timeout 是可选的超时

注意：

apply_async(func[, args[, kwds[, callback]]]) 它是非阻塞，apply(func[, args[, kwds]])是阻塞的

【示例 2-9】进程池的使用（非阻塞）

```
import multiprocessing
import time
```

```
def func(msg):
    print("start:", msg)
    time.sleep(3)
    print("end：",msg)

if __name__ == "__main__":
    pool = multiprocessing.Pool(processes = 3)
    for i in range(5):
        msg = "hello %d" %(i)
        #维持执行的进程总数为 processes，当一个进程执行完毕后会添加新的进程进去
        pool.apply_async(func, (msg, ))

    pool.close()#进程池关闭之后不再接收新的请求
    #调用 join 之前，先调用 close 函数，否则会出错。
    #  执行完 close 后不会有新的进程加入到 pool,join 函数等待所有子进程结束
    pool.join()
```

执行结果如图 2-12 所示：

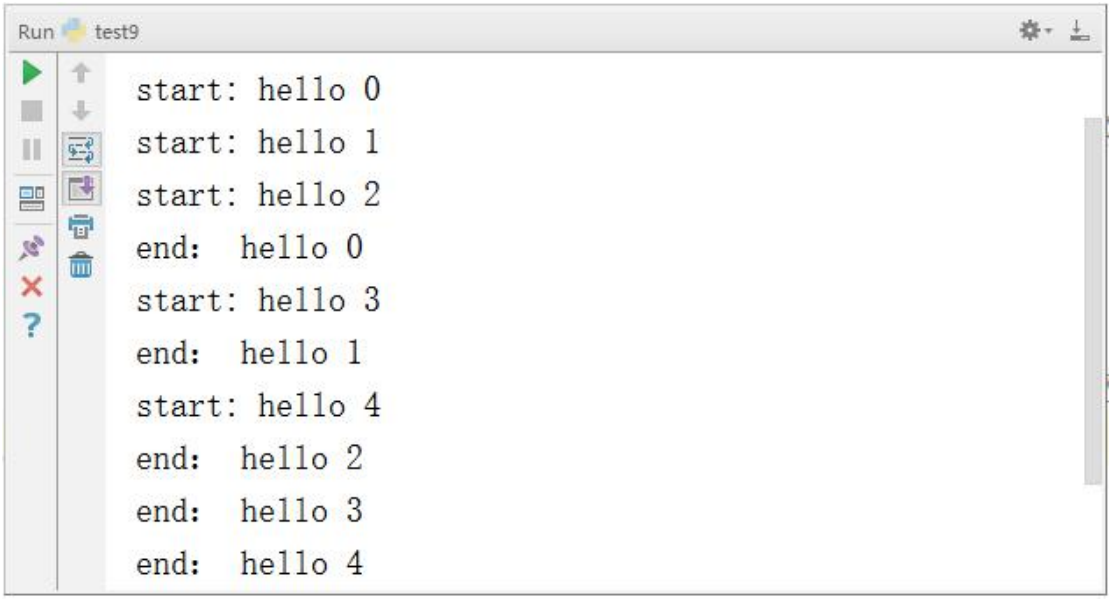


图 2-12 示例 2-9 执行结果

【示例 2-10】进程池的使用（阻塞）

```
import multiprocessing
import time
```

```
def func(msg):
    print("start:", msg)
    time.sleep(3)
    print("end",msg)

if __name__ == "__main__":
    pool = multiprocessing.Pool(processes = 3)
    for i in range(5):
        msg = "hello %d" %(i)
        #维持执行的进程总数为 processes，当一个进程执行完毕后会添加新的进程进去
        pool.apply(func, (msg, ))

    pool.close()
    #调用 join 之前，先调用 close 函数，否则会出错。
    #  执行完 close 后不会有新的进程加入到 pool,join 函数等待所有子进程结束
    pool.join()
```

执行结果如图 2-13 所示：

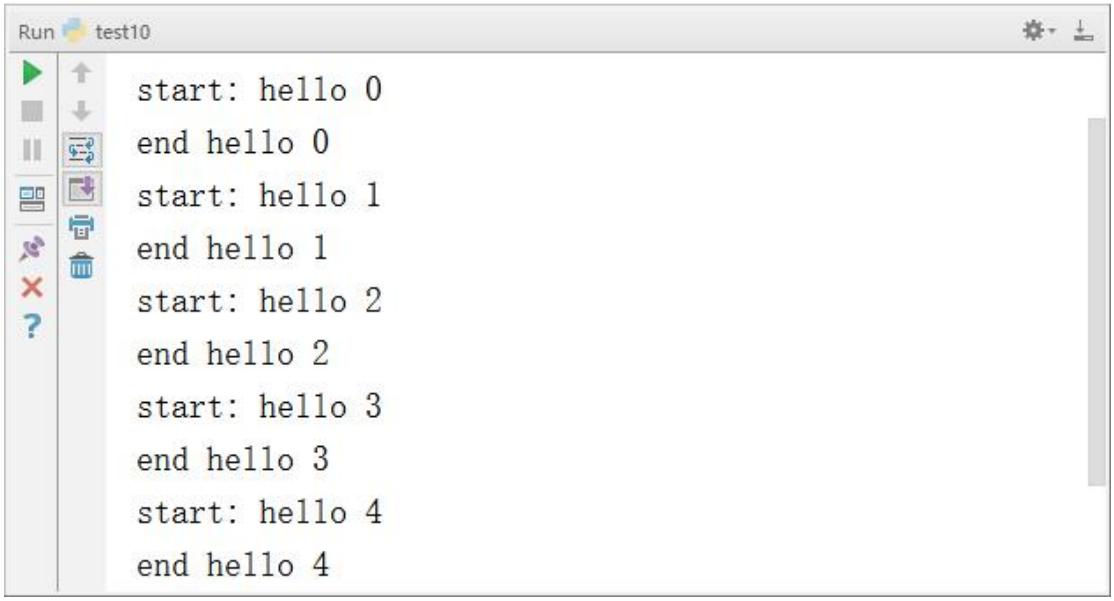


图 2-13 示例 2-10 执行结果

2.3.4 进程间通信

全局变量在多个进程中不共享，进程之间的数据是独立的，默认情况下互不影响。

【示例 2-11】多个进程中数据不共享

```
from multiprocessing import Process
num=1
def work1():
    global num
    num+=5
    print('子进程 1 运行，num:',num)

def work2():
    global num
    num += 10
    print('子进程 2 运行，num：',num)

if __name__=='__main__':
    print('父进程开始运行')
    p1=Process(target=work1)
    p2=Process(target=work2)
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

执行结果如图 2-14 所示：

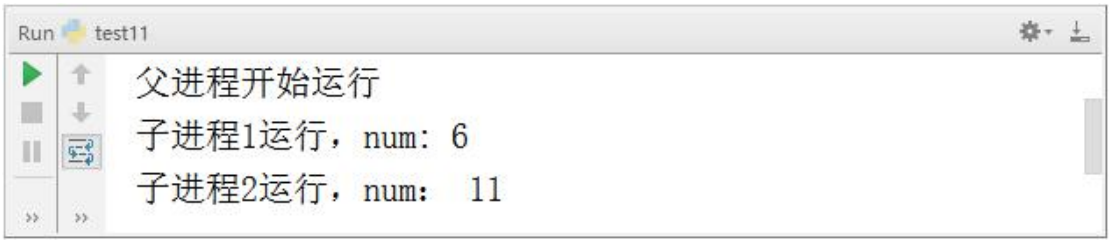


图 2-14 示例 2-11 执行结果

Queue 是多进程安全的队列，可以使用 Queue 实现多进程之间的数据传递。put 方法用以插入数据到队列中，put 方法还有两个可选参数：blocked 和 timeout。如果 blocked 为 True（默认值），并且 timeout 为正值，该方法会阻塞 timeout 指定的时间，直到该队列有剩余的空间。如果超时，会抛出 Queue.full 异常。如果 blocked 为 False，但该 Queue 已满，会立即抛出 Queue.full 异常。

get 方法可以从队列读取并且删除一个元素。同样，get 方法有两个可选参数：blocked

和 timeout。如果 blocked 为 True（默认值），并且 timeout 为正值，那么在等待时间内没有取到任何元素，会抛出 Queue.Empty 异常。如果 blocked 为 False，有两种情况存在，如果 Queue 有一个值可用，则立即返回该值，否则，如果队列为空，则会抛出 Queue.Empty 异常。

表 2-4 Queue 实例方法表

方法	描述
cancle_join_thread()	不会在进程退出时自动连接后台线程。这可以防止 join_thread()方法阻塞
close()	关闭队列，防止队列中加入更多数据。调用此方法时，后台线程将继续写入那些已入队列尚未写入数据，但将在此方法完成时马上关闭
empty()	如果调用此方法时 q 为空，返回 True
full()	如果 q 已满，返回 True
get([block [,timeout])	返回 q 中的一个项。如果 q 为空，此方法将阻塞，直到队列中有项可用为止。Block 用于控制阻塞行为，默认为 True。如果设置为 False，将引发 Queue.Empty 异常(定义在 Queue 模块中)。Timeout 是可选超时时间，用在阻塞模式中。如果在指定的时间间隔内没有项变为可用，将引发 Queue.Empty 异常
join_thread()	连接队列的后台线程。此方法用于在调用 q.close()方法之后，等待所有队列项被消耗。默认情况下此方法由不是 q 的原始创建者的所有进程调用。调用 q.cancle_join_thread()方法可以禁止这种行为
put(item [, block [, timeout]])	将 item 放入队列。如果队列已满，此方法将阻塞至有空间可用为止。Block 控制阻塞行为，默认为 True。如果设置为 False，将引发 Queue.Empty 异常(定义在 Queue 模块中)。Timeout 指定在阻塞模式中等待可用时空间的时间长短。超时后将引发 Queue.Full 异常。
qsize()	返回目前队列中项的正确数量。
joinableQueue([maxsize])	创建可连接的共享进程队列。这就像是一个 Queue 对象，但队列允许项的消费者通知生产者项已经被成功处理。通知进程是使用共享的信号和条件变量来实现的
task_done()	消费者使用此方法发出信号，表示 q.get()返回的项已经被处理。如果调用此方法的次数大于从队列中删除的项的数量，将引发 ValueError 异常
join()	生产者使用此方法进行阻塞，知道队列中的所有项均被处理。阻塞将持续到位队列中的每个项均调用 q.task_done()方法为止

【示例 2-12】Queue 队列的基本使用

```
from multiprocessing import Queue
```

```
q=Queue(3)
q.put('消息 1')
q.put('消息 2')
print('消息队列是否已满：',q.full())
q.put('消息 3')
print('消息队列是否已满：',q.full())

# q.put('消息 4')因为消息队列已满，需要直接写入需要等待，如果超时会抛出异常，
#  所以写入时候需判断，消息队列是否已满
if not q.full():
    q.put('消息 4')

#同理读取消息时，先判断消息队列是否为空，再读取
if not q.empty():
    for i in range(q.qsize()):
        print(q.get())
```

执行结果如图 2-15 所示：

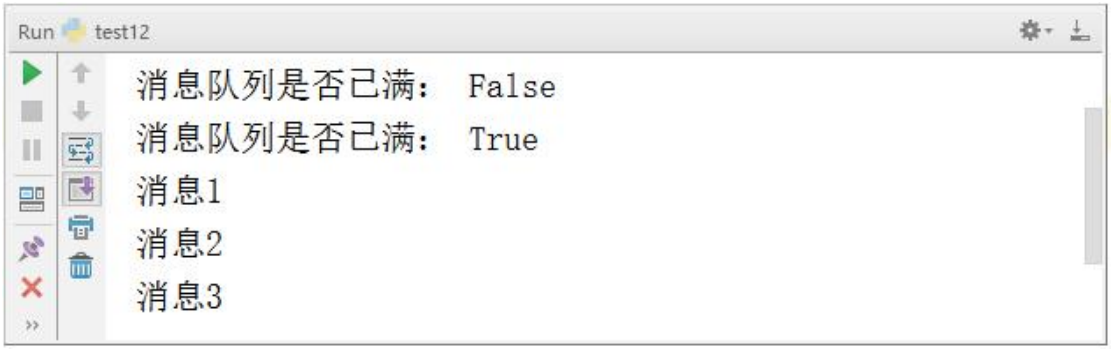


图 2-15 示例 2-12 执行结果

【示例 2-13】Queue 队列实现进程间通信

```
from multiprocessing import *
import time
def write(q):
    #将列表中的元素写入队列中
    for i in ["a","b","c"]:
        print('开始写入值%s' % i)
        q.put(i)
        time.sleep(1)
```

```
#读取
def read(q):
    print('开始读取')
    while True:
        if not q.empty():
            print('读取到:',q.get())
            time.sleep(1)
        else:
            break
if __name__=='__main__':
    #创建队列
    q=Queue()
    #创建写入进程
    pw=Process(target=write,args=(q,))
    pr=Process(target=read,args=(q,))
    #启动进程
    pw.start()
    pw.join()
    pr.start()
    pr.join()
```

执行结果如图 2-16 所示：

图 2-16 示例 2-13 执行结果

如果使用 Pool 创建进程，就需要使用 multiprocessing.Manager()中的 Queue()来完成进程间的通信，而不是 multiprocessing.Queue()，否则会抛出如下异常。

```
RuntimeError: Queue objects should only be shared between processes through inheritance
```

【示例 2-14】进程池创建进程完成进程之间的通信

```
from multiprocessing import Manager,Pool
import time
def write(q):
    #将列表中的元素写入队列中
    for i in ["a","b","c"]:
        print('开始写入值%s' % i)
        q.put(i)
        time.sleep(1)

#读取
def read(q):
    print('开始读取')
    while True:
        if not q.empty():
            print('读取到:',q.get())
            time.sleep(1)
        else:
            break
if __name__=='__main__':
    #创建队列
    q=Manager().Queue()
    #创建进程池
    p=Pool(3)
    #使用阻塞模式创建进程
    p.apply(write,(q,))
    p.apply(read,(q,))
    p.close()
    p.join()
```

执行结果如图 2-17 所示：

图 2-17 示例 2-14 执行结果

2.4 线程

线程也是实现多任务的一种方式，一个进程中，也经常需要同时做多件事，就需要同时运行多个‘子任务’，这些子任务就是线程。一个进程可以拥有多个并行的线程，其中每一个线程，共享当前进程的资源。

进程和线程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护，而进程正相反。

在 Python 程序中，可以通过“_thread”和 threading（推荐使用）这两个模块来处理线程。在 Python3 中，thread 模块已经废弃。可以使用 threading 模块代替。所以，在 Python3 中不能再使用 thread 模块，但是为了兼容 Python3 以前的程序，在 Python3 中将 thread 模块重命名为“_thread”。

2.4.1 _thread 模块

在 Python 程序中，可以通过两种方式来使用线程：使用函数或者使用类来包装线程对象。当使用 thread 模块来处理线程时，可以调用里面的函数 start_new_thread()来生成一个新的线程，语法格式如下：

```
_thread.start_new_thread ( function, args[, kwargs] )
```

其中 function 是线程函数；args 表示传递给线程函数的参数，他必须是个 tuple 类型；kwargs 是可选参数。

【示例 2-15】使用_thread 模块创建线程

```
import _thread
import time
```

```
def fun1():
    print('开始运行 fun1')
    time.sleep(4)
    print('运行 fun1 结束')
def fun2():
    print('开始运行 fun2')
    time.sleep(2)
    print('运行 fun2 结束')
if __name__=='__main__':
    print('开始运行')
    #启动一个线程运行函数 fun1
    _thread.start_new_thread(fun1,())
     #启动一个线程运行函数 fun2
    _thread.start_new_thread(fun2,())
    time.sleep(6)
```

执行结果如图 2-18 所示：

图 2-18 示例 2-15 执行结果

从程序运行结果可以看出，在 fun2 函数中调用了 sleep 函数休眠，当休眠期间，会释放 CPU 的计算资源，这时 fun1 抢占了 CPU 资源开始执行。

【示例 2-16】为线程传递参数

```
import _thread
import time
def fun1(thread_name,delay):
    print('线程{0}开始运行 fun1'.format(thread_name))
    time.sleep(delay)
    print('线程{0}运行 fun1 结束'.format(thread_name))
def fun2(thread_name,delay):
```

```
    print('线程{0}开始运行 fun2'.format(thread_name))
    time.sleep(2)
    print('线程{0}运行 fun2 结束'.format(thread_name))
if __name__=='__main__':
    print('开始运行')
    #启动一个线程运行函数 fun1
    _thread.start_new_thread(fun1,('thread-1',4))
     #启动一个线程运行函数 fun2
    _thread.start_new_thread(fun2,('thread-2',2))
    time.sleep(6)
```

执行结果如图 2-19 所示：

图 2-19 示例 2-16 执行结果

从输出结果可以看出，由于每个线程函数的休眠时间是通过函数参数传入的，所以随机输出了这个结果，传入的参数不同，输出的结果是不一样的。

2.4.2 threading 模块

Python3 通过两个标准库 _thread 和 threading 提供对线程的支持。_thread 提供了低级别的、原始的线程以及一个简单的锁，它相比于 threading 模块的功能还是比较有限的。

threading 模块除了包含 _thread 模块中的所有方法外，还提供的其他方法：

- threading.currentThread(): 返回当前的线程变量。
- threading.enumerate(): 返回一个包含正在运行的线程的 list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- threading.activeCount(): 返回正在运行的线程数量，与 len(threading.enumerate())有相同的结果。

在 Python3 程序中，对多线程支持最好的是 threading 模块，使用这个模块，可以灵活地创建多线程程序，并且可以在多线程之间进行同步和通信。在 Python3 程序中，可以通过

如下两种方式来创建线程：

- 通过 threading.Thread 直接在线程中运行函数
- 通过继承类 threading.Thread 来创建线程

在 Python 中使用 threading.Thread 的基本语法格式如下所示：

```
Thread(group=None, target=None, name=None, args=(), kwargs={})
```

其中 target: 要执行的方法；name: 线程名；args/kwargs: 要传入方法的参数。

Thread 类的方法如表 2-5 所示：

表 2-5 Thread 类的方法

方法名	描述
run()	用以表示线程活动的方法
start()	启动线程活动
join([time])	等待至线程中止。这阻塞调用线程直至线程的 join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生
isAlive()	返回线程是否活动的
getName()	返回线程名
setName()	设置线程名

【示例 2-17】threading.Thread 直接创建线程

```
import threading
import time
def fun1(thread_name,delay):
    print('线程{0}开始运行 fun1'.format(thread_name))
    time.sleep(delay)
    print('线程{0}运行 fun1 结束'.format(thread_name))
def fun2(thread_name,delay):
    print('线程{0}开始运行 fun2'.format(thread_name))
    time.sleep(delay)
    print('线程{0}运行 fun2 结束'.format(thread_name))
if __name__=='__main__':
    print('开始运行')
    #创建线程
    t1=threading.Thread(target=fun1,args=('thread-1',2))
    t2=threading.Thread(target=fun2,args=('thread-2',4))
    t1.start()
    t2.start()
```

执行结果如图 2-20 所示：

图 2-20 示例 2-17 执行结果

在 Python 中，通过继承类 threading.Thread 的方式来创建一个线程。这种方法只要重写类 threading.Thread 中的方法 run()，然后再调用方法 start()就能创建线程，并运行方法 run()中的代码。

【示例 2-18】继承 threading.Thread 类创建线程

```
import threading
import time
def fun1(delay):
    print('线程{0}开始运行 fun1'.format(threading.current_thread().getName()))
    time.sleep(delay)
    print('线程{0}运行 fun1 结束'.format(threading.current_thread().getName()))
def fun2(delay):
    print('线程{0}开始运行 fun2'.format(threading.current_thread().getName()))
    time.sleep(2)
    print('线程{0}运行 fun2 结束'.format(threading.current_thread().getName()))
#创建线程类继承 threading.Thread
class MyThread(threading.Thread):
    #重写父类的构造方法，其中 func 是线程函数，args 是传入线程的参数,name 是线程名
    def __init__(self,func,name,args):
        super().__init__(target=func,name=name,args=args)
    #重写父类的 run()方法
    def run(self):
        self._target(*self._args)

if __name__=='__main__':
    print('开始运行')
```

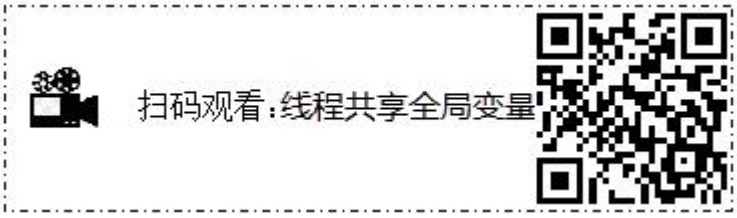
```
    #创建线程
    t1=MyThread(fun1,'thread-1',(2,))
    t2=MyThread(fun2,'thread-2',(4,))
    t1.start()
    t2.start()
```

执行结果如图 2-21 所示：

图 2-21 示例 2-18 执行结果

在调用 Thread 类的构造方法时，需要将线程函数、参数等值传入构造方法，其中 name 表示线程的名字，如果不指定这个参数，默认的线程名字格式为 Thread-1、Thread-2。每一个传入构造方法的参数值，在 Thread 类中都有对应的成员变量保存这些值，这些成员变量都以下划线(_)开头，如果_target、_args 等。在 run 方法中需要使用这些变量调用传入的线程函数，并为线程函数传递参数。

2.4.3 线程共享全局变量



在一个进程内所有线程共享全局变量，多线程之间的数据共享比多进程要好。但是可能造成多个进程同时修改一个变量（即线程非安全），可能造成混乱。

【示例 2-19】线程共享全局变量

```
import time
from threading import *
#定义全局变量 num
num=10
def test1():
    global num
    for i in range(3):
```

```
            num+=1
        print('test1 输出 num:',num)


def test2():
        global num
        print('test2 输出 num:',num)
if __name__=='__main__':
        t1=Thread(target=test1)
        t2=Thread(target=test2)
        t1.start()
        t1.join()
        t2.start()
        t2.join()
```

执行结果如图 2-22 所示：

图 2-22 示例 2-19 执行结果

【示例 2-20】线程共享全局变量存在问题

```
import time
from threading import *
#定义全局变量 num
num=0
def test1():
        global num
        for i in range(100000):
            num+=1
        print('test1 输出 num:',num)


def test2():
        global num
        for i in range(100000):
            num+=1
```

```
    print('test2 输出 num:',num)
if __name__=='__main__':
    t1=Thread(target=test1)
    t2=Thread(target=test2)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

执行结果如图 2-23 所示：

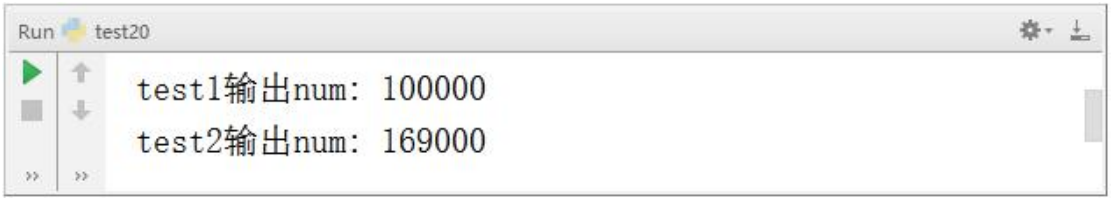


图 2-23 示例 2-20 执行结果

2.4.4 互斥锁

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。最简单的同步机制就是引入互斥锁。

锁有两种状态——锁定和未锁定。某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改；直到该线程释放资源，将资源的状态变成“非锁定”状态，其他的线程才能再次锁定该资源。就如图 2-22 所示，排队上厕所。互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。

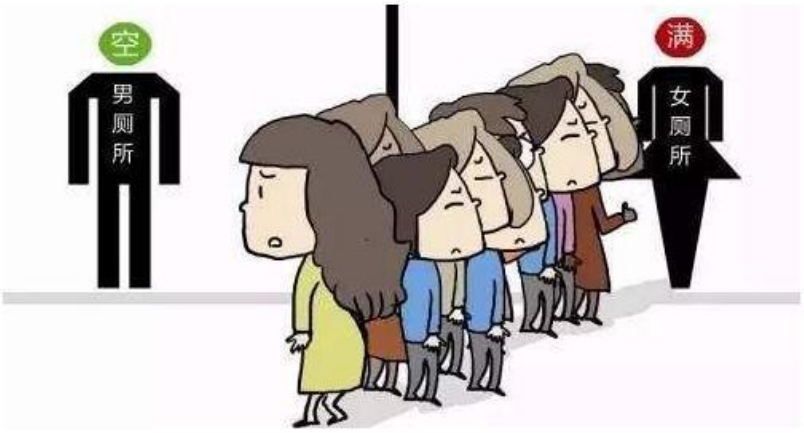


图 2-24 排队上厕所

使用 Thread 对象的 Lock 可以实现简单的线程同步，有上锁 acquire 方法和 释放 release 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire 和 release 方法之间。

【示例 2-21】互斥锁

```
import time
from threading import Thread,Lock
#定义全局变量 num
num=0
#创建一把互斥锁
mutex=Lock()
def test1():
    global num
    '''
    在两个线程中都调用上锁的方法，则这两个线程就会抢着上锁，
    如果有 1 方成功上锁，那么导致另外一方会堵塞（一直等待）直到这个锁被解开
    '''
    mutex.acquire()#上锁
    for i in range(100000):
        num+=1
    mutex.release()
    print('test1 输出 num:',num)
```

```
def test2():
    global num
    mutex.acquire()  # 上锁
    for i in range(100000):
        num+=1
    mutex.release()
    print('test2 输出 num:',num)

if __name__=='__main__':
    t1=Thread(target=test1)
    t2=Thread(target=test2)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

执行结果如图 2-25 所示：

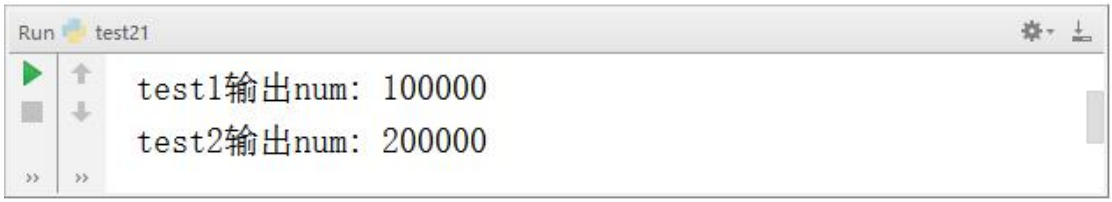


图 2-25 示例 2-21 执行结果

【示例 2-22】互斥锁改进

```
import time
from threading import Thread,Lock
#定义全局变量 num
num=0
#创建一把互斥锁
mutex=Lock()
def test1():
    global num
    '''
    在两个线程中都调用上锁的方法，则这两个线程就会抢着上锁，
```

```
    如果有 1 方成功上锁，那么导致另外一方会堵塞（一直等待）直到这个锁被解开
    '''
    for i in range(100000):
        mutex.acquire()  # 上锁
        num+=1
        mutex.release()
    print('test1 输出 num:',num)


def test2():
    global num
    for i in range(100000):
        mutex.acquire()  # 上锁
        num+=1
        mutex.release()
    print('test2 输出 num:',num)

if __name__=='__main__':
    t1=Thread(target=test1)
    t2=Thread(target=test2)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

执行结果如图 2-26 所示：

图 2-26 示例 2-22 执行结果

2.4.5 死锁

在线程共享多个资源的时候，如果两个线程分别占有一部分资源并且同时等待对方的资源，就会造成死锁。如图 2-27 所示：

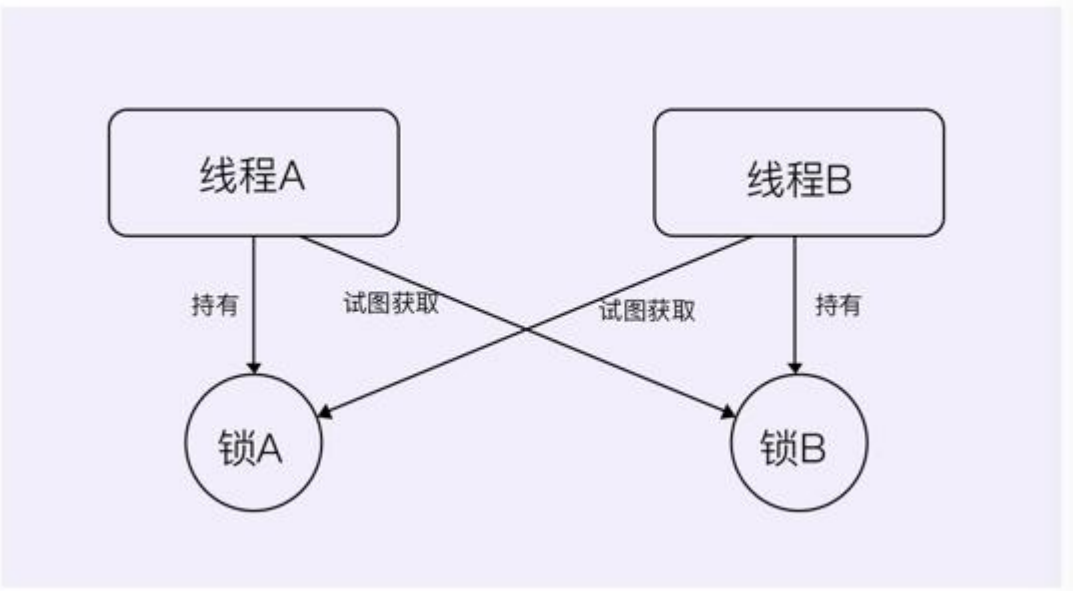


图 2-27 死锁

【示例 2-23】死锁

```
import time
from threading import Thread,Lock
import threading
mutexA=threading.Lock()
mutexB=threading.Lock()
class MyThread1(Thread):
    def run(self):
        if mutexA.acquire():
            print(self.name,'执行')
            time.sleep(1)
            if mutexB.acquire():
                print(self.name,'执行')
                mutexB.release()
            mutexA.release()
class MyThread2(Thread):
    def run(self):
        if mutexB.acquire():
            print(self.name,'执行')
            time.sleep(1)
            if mutexA.acquire():
```

```
                print(self.name,'执行')
                mutexA.release()
            mutexB.release()
if __name__ == '__main__':
    t1=MyThread1()
    t2=MyThread2()
    t1.start()
    t2.start()
```

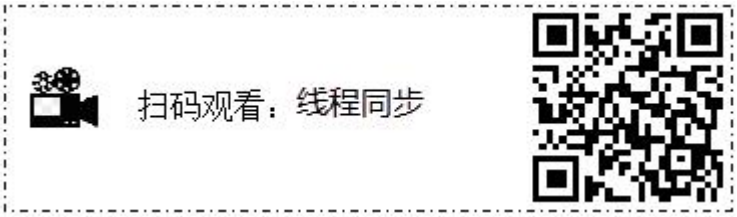
执行结果如图 2-28 所示：

图 2-28 示例 2-23 执行结果

从程序运行结果可以看出，线程 Thread-1 和线程 Thread-2 都在等待对方释放资源，就造成死锁。

2.4.6 线程同步的应用

同步就是协同步调，按预定的先后次序进行运行。例如：开会。“同”字指协同、协助、互相配合。



如进程、线程同步，可以理解为进程或线程 A 和 B 一块配合，A 执行到一定程度时要依靠 B 的某个结果，于是停下来，示意 B 运行，B 运行后将结果给 A，A 继续运行。

【示例 2-24】线程同步应用

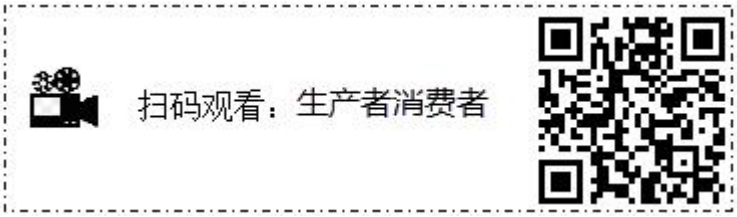
```
import time
from threading import Thread,Lock
import threading
lock1=Lock()
lock2=Lock()
lock3=Lock()
lock2.acquire()
lock3.acquire()
class Task1(Thread):
```

```
    def run(self):
        while True:
            if lock1.acquire():
                print('...task1...')
                time.sleep(1)
                lock2.release()

class Task2(Thread):
    def run(self):
        while True:
            if lock2.acquire():
                print('...task2...')
                time.sleep(1)
                lock3.release()

class Task3(Thread):
    def run(self):
        while True:
            if lock3.acquire():
                print('...task3...')
                time.sleep(1)
                lock1.release()

if __name__ == '__main__':
    t1=Task1()
    t2=Task2()
    t3=Task3()
    t1.start()
    t2.start()
    t3.start()
```

执行结果如图 2-29 所示：

图 2-29 示例 2-24 执行结果

2.4.7 生产者消费者模式



生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入生产者和消费者模式

生产者消费者模式通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者之间不直接通信。生产者生产商品，然后将其放到类似队列的数据结构中，消费者不找生产者要数据，而是直接从队列中取。这里使用 queue 模块来提供线程间通信的机制，也就是说，生产者和消费者共享一个队列。生产者生产商品后，会将商品添加到队列中。消费者消费商品，会从队列中取出商品。

【示例 2-25】生产者-消费者模型

```
import time
import threading
from queue import Queue
class Producer(threading.Thread):
    def run(self):
        global queue
        count=0
        while True:
            if queue.qsize()<1000:
                for i in range(100):
```

```
                count += 1
                msg = '生成产品' + str(count)
                queue.put(msg)
                print(msg)
        time.sleep(0.5)
class Consumer(threading.Thread):
    def run(self):
        global queue
        while True:
            if queue.qsize()>100:
                for i in range(3):
                    msg=self.name+'消费了'+queue.get()
                    print(msg)
            time.sleep(1)
if __name__ == '__main__':
    queue = Queue()
    p=Producer()
    p.start()
    time.sleep(1)
    c=Consumer()
    c.start()
```

执行结果如图 2-30 所示：

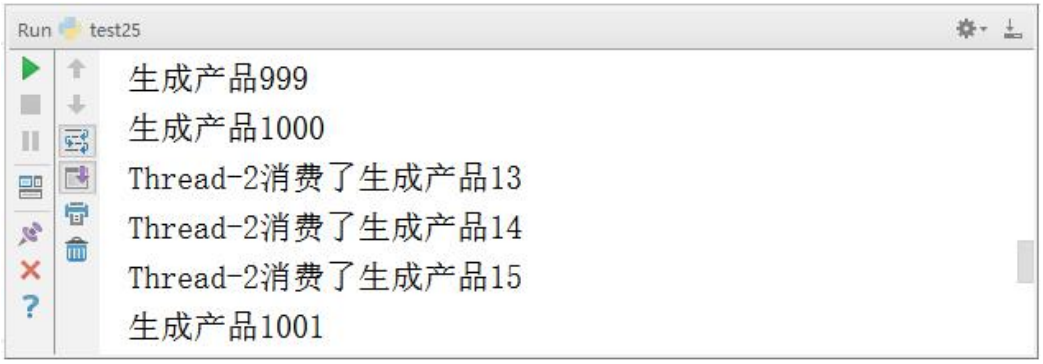


图 2-30 示例 2-25 执行结果

2.4.8 ThreadLocal

多线程环境下，每一个线程均可以使用所属进程的全局变量。如果一个线程对全局变量进行了修改，将会

影响到其他所有的线程对全局变量的计算操作，从而出现数据混乱，即为脏数据。为了避免多个线程同时对变量进行修改，引入了线程同步机制，通过互斥锁来控制对全局变量的访问。所以有时候线程使用局部变量比全局变量好，因为局部变量只有线程自身可以访问，同一个进程下的其他线程不可访问。

但是局部变量也是有问题，就是在函数调用的时候，传递起来很麻烦。

【示例 2-26】局部变量作为参数传递

```
def process_student(name):
    std=Student(name)
    do_task1(std)
    do_task2(std)

def do_task1(std):
    do_sub_task1(std)
    do_sub_task2(std)

def do_task2(std):
    do_sub_task1(std)
    do_sub_task2(std)
```

从上面的实例可以看到每个函数一层一层调用都需要传递 std 参数，非常麻烦，如果使用全局变量也不行，因为每个线程处理不同的 Student 对象，不能共享。因此 Python 还提供了 ThreadLocal 变量，它本身是一个全局变量，但是每个线程却可以利用它来保存属于自己的私有数据，这些私有数据对其他线程也是不可见的。

【示例 2-27】ThreadLocal 的使用

```
import threading
# 创建全局 ThreadLocal 对象:
local = threading.local()
def process_student():
    # 获取当前线程关联的 name:
    student_name = local.name
    print('线程名：%s 学生姓名:%s' % (threading.current_thread().name,student_name))
def process_thread(name):
    # 绑定 ThreadLocal 的 name:
    local.name = name
```

```
    process_student()
t1 = threading.Thread(target=process_thread, args=('张三',), name='Thread-A')
t2 = threading.Thread(target=process_thread, args=('李四',), name='Thread-B')
t1.start()
t2.start()
t1.join()
t2.join()
```

执行结果如图 2-31 所示：

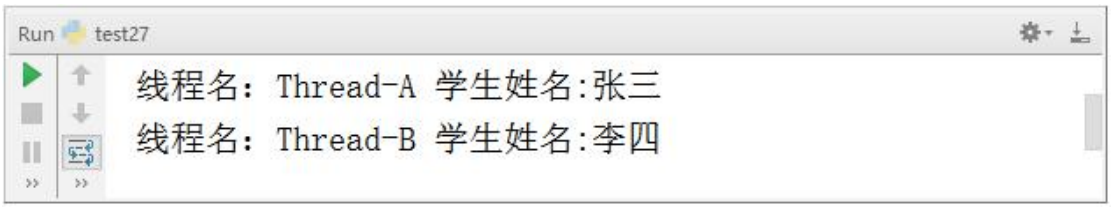


图 2-31 示例 2-27 执行结果

习题

一、选择题

1. 同一主机间的信息交互，可以通过（）（多选）

A. 无名管道

B. 有名管道

C. 消息队列

D. 共享内存

2. 当线程调用 start( )后，其所处状态为（）。

A 阻塞状态

B. 运行状态

C. 就绪状态

D. 新建状态

3. 以下选项中关于进程控制方法的说法不正确的是（）。

A. join ( ) 的作用是等待进程 p 终止

B. is_alive()可以用来判断进程是否仍然运行

C. start() 启动进程，这将运行代表进程的子进程，并调用该子进程中的start()函数

D. terminate()强制终止进程

4. 关于进行和线程说法正确的是（）。（多选）

A. 进程是资源分配的基本单位，线程是 CPU 执行和调度的基本单位

B. CPU 上真正执行的是线程，线程比进程轻量，其切换和调度代价比进程要小

C. 线程间对于共享的进程数据需要考虑线程安全问题

D. 如果 CPU 和系统支持多线程与多进程，多个进程并行执行的同时，每个进程中的线程也可以并行执行

5. **Python 中线程安全问题是通过关键字（）解决的。**

A. finally

B. wait( )

C. lock

D. TreadLocal( )

二、解答题

1. threading 模块提供了哪些类。
2. 简述线程和进程的区别和联系。
3. 简述进程的基本状态及状态之间的关系。

4. 如何在 Python 中实现多线程？
5. Python 中如何使用 TreadLocal？

三、**编码题**

1. 编写一个程序， 使用 Process 创建 1 个子进程，让子进程每 1 秒钟打印 1 个数字，数字从 1 开始一直到 10，即 1.2.3.....10。
2. 编写一个程序，使用 multiprocessing 模块中的 Queue，完成子进程中将 hello 传递到父进程中，父进程打印出来。
3. 用线程实现生产者消费者模式。

第三章 TCP 与 UDP 编程

自从互联网诞生以来，现在基本上所有的程序都是网络程序，很少有单机版的程序了。计算机网络就是把各个计算机连接到一起，让网络中的计算机可以互相通信。网络编程就是如何在程序中实现两台计算机的通信。

网络编程对所有开发语言都是一样的，Python 也不例外。网络是一个互联网应用的重要组成部分，在 Python 语言中提供了大量的内置模块和第三方模块用于支持各种网络访问，而且 Python 语言在网络通信方面的优点特别突出，远远领先其他语言。

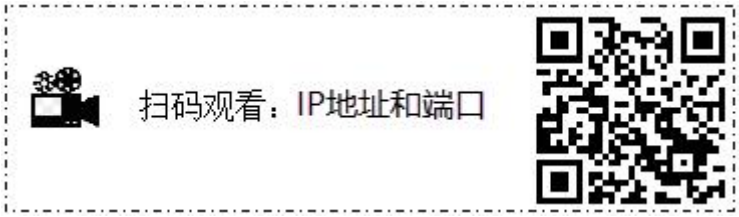
通过阅读本章，你可以：

- 了解 TCP 和 UDP
- 掌握编写 UDP Socket 客户端应用
- 掌握编写 UDP Socket 服务器端应用
- 掌握编写 TCP Socket 客户端应用
- 掌握编写 TCP Socket 服务器端应用

3.1 基本概念

3.1.1 IP 地址与端口

- **IP 地址**



用来标识网络中的一个通信实体的地址。通信实体可以是计算机、路由器等。比如互联网的每个服务器都要有自己的 IP 地址，而每个局域网的计算机要通信也要配置 IP 地址。路由器是连接两个或多个网络的网络设备。

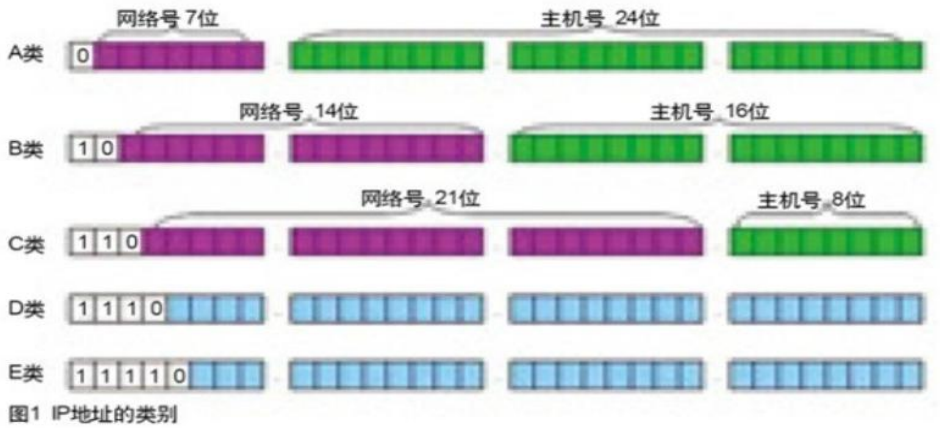


图 3-1 IP 地址分类

目前主流使用的 IP 地址是 IPV4，但是随着网络规模的不断扩大，IPV4 面临着枯竭的

危险，所以推出了 IPV6。

IP 地址实际上是一个 32 位整数（称为 IPv4），以字符串表示的 IP 地址如 192.168.0.1 实际上是把 32 位整数按 8 位分组后的数字表示，目的是便于阅读。

IPv6 地址实际上是一个 128 位整数，它是目前使用的 IPv4 的升级版，以字符串表示类似于 2001:0db8:85a3:0042:1000:8a2e:0370:7334。

注意事项

- 127.0.0.1 本机地址
- 192.168.0.0--192.168.255.255 为私有地址，属于非注册地址，专门为组织机构内部使用。

- **端口**

IP 地址用来标识一台计算机，但是一台计算机上可能提供多种网络应用程序，如何来区分这些不同的程序呢？这就要用到端口。

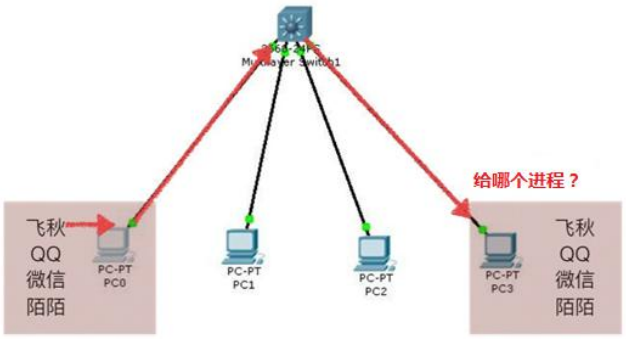


图 3-2 端口

端口是虚拟的概念，并不是说在主机上真的有若干个端口。通过端口，可以在一个主机上运行多个网络应用程序。 端口的表示是一个 16 位的二进制整数，对应十进制的 0-65535。

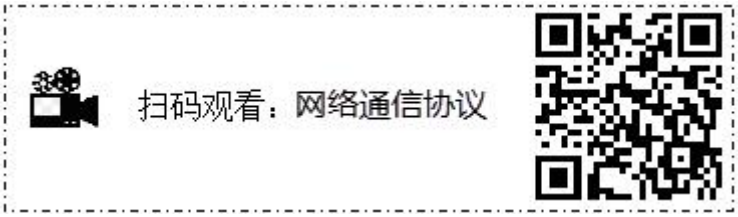
Oracle、MySQL、Tomcat、QQ、msn、迅雷、电驴、360 等网络程序都有自己的端口。

总结

- IP 地址好比每个人的地址（门牌号），端口好比是房间号。必须同时指定 IP 地址和端口号才能够正确的发送数据。
- IP 地址好比为电话号码，而端口号就好比为分机号。

3.1.2 网络通信协议

- **网络通信协议**



通过计算机网络可以实现不同计算机之间的连接与通信，但是计算机网络中实现通信必须有一些约定即通信协议，对速率、传输代码、代码结构、传输控制步骤、出错控制等制定标准。就像两个人想要顺利沟通就必须使用同一种语言一样，如果一个人只懂英语而另外一个人只懂中文，这

样就会造成没有共同语言而无法沟通。

国际标准化组织(ISO，即 International Organization for Standardization)定义了网络通信协议的基本框架，被称为 OSI（Open System Interconnect，即开放系统互联）模型。要制定通讯规则，内容会很多，比如要考虑 A 电脑如何找到 B 电脑，A 电脑在发送信息给 B 电脑时是否需要 B 电脑进行反馈，A 电脑传送给 B 电脑的数据格式又是怎样的？内容太多太杂，所以 OSI 模型将这些通讯标准进行层次划分，每一层次解决一个类别的问题，这样就使得标准的制定没那么复杂。OSI 模型制定的七层标准模型，分别是：应用层，表示层，会话层，传输层，网络层，数据链路层，物理层。

OSI 七层协议模型如图 3-3 所示：

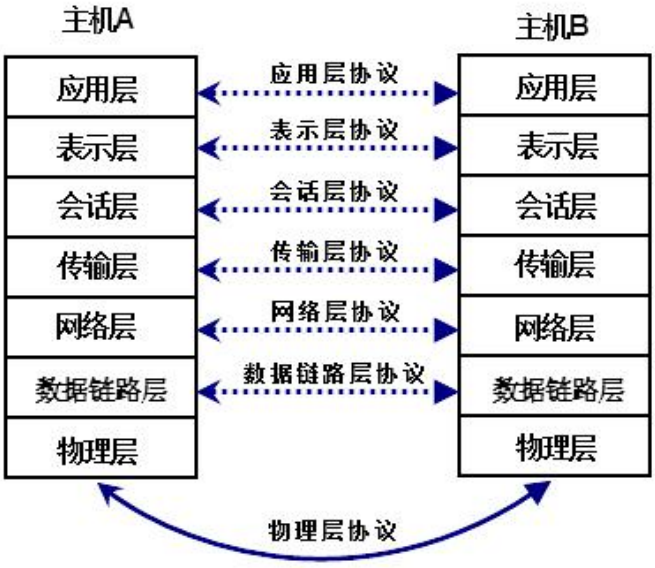


图 3-3 OSI 七层协议模型

虽然国际标准化组织制定了这样一个网络通信协议的模型，但是实际上互联网通讯使用最多的网络通信协议是 TCP/IP 网络通信协议。

TCP/IP 是一个协议族，也是按照层次划分，共四层：应用层，传输层，互连网络层，网络接口层（物理+数据链路层）。

那么 TCP/IP 协议和 OSI 模型有什么区别呢？OSI 网络通信协议模型，是一个参考模型，而 TCP/IP 协议是事实上的标准。TCP/IP 协议参考了 OSI 模型，但是并没有严格按照 OSI 规定的七层标准去划分，而只划分了四层，这样会更简单点，当划分太多层次时，你很难区分某个协议是属于哪个层次的。TCP/IP 协议和 OSI 模型也并不冲突，TCP/IP 协议中的应用层协议，就对应于 OSI 中的应用层，表示层，会话层。就像以前有工业部和信息产业部，现在实行大部制后只有工业和信息化部一个部门，但是这个部门还是要做以前两个部门一样多的事情，本质上没有多大的差别。TCP/IP 中有两个重要的协议，传输层的 TCP 协议和互连网络层的 IP 协议，因此就拿这两个协议做代表，来命名整个协议族了，再说 TCP/IP 协议时，是指整个协议族。

- **网络协议的分层**

由于网络结点之间联系很复杂，在制定协议时，把复杂成份分解成一些简单的成份，再将它们复合起来。最常用的复合方式是层次方式，即同层间可以通信、上一层可以调用下一层，而与再下一层不发生关系。

把用户应用程序作为最高层，把物理通信线路作为最低层，将其间的协议处理分为若干层，规定每层处理的任务，也规定每层的接口标准。

ISO 模型与 TCP/IP 模型的对应关系如图 3-4 所示：

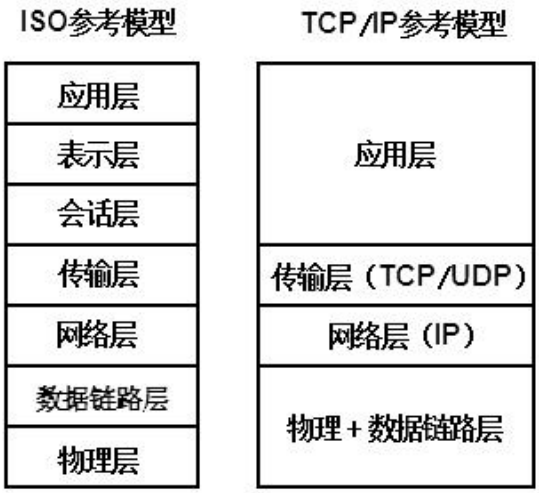


图 3-4 ISO 模型与 TCP/IP 模型的对应关系

3.1.3 TCP/UDP

- **联系和区别**

TCP 协议和 UDP 协议是传输层的两种协议。Socket 是传输层供给应用层的编程接口，所以 Socket 编程就分为 TCP 编程和 UDP 编程两类。

在网络通讯中，TCP 方式就类似于拨打电话，使用该种方式进行网络通讯时，需要建立专门的虚拟连接，然后进行可靠的数据传输，如果数据发送失败，则客户端会自动重发该数据。而 UDP 方式就类似于发送短信，使用这种方式进行网络通讯时，不需要建立专门的虚拟连接，传输也不是很可靠，如果发送失败则客户端无法获得。

这两种传输方式都在实际的网络编程中使用，重要的数据一般使用 TCP 方式进行数据传输，而大量的非核心数据则可以通过 UDP 方式进行传递，在一些程序中甚至结合使用这两种方式进行数据传递。

由于 TCP 需要建立专用的虚拟连接以及确认传输是否正确，所以使用 TCP 方式的速度稍微慢一些，而且传输时产生的数据量要比 UDP 稍微大一些。

总结

- TCP 是面向连接的，传输数据安全，稳定，效率相对较低。
- UDP 是面向无连接的，传输数据不安全，效率较高。

3.2 套接字编程

应用程序通常通过“套接字”（socket）向网络发出请求或者应答网络请求，使用主机之间或者一台计算机上的进程间可以通信。Python 语言提供了两种访问网络服务的功能。其中低级别的网络服务通过套接字实现，而高级别的网络服务通过模块 SocketServer 实现，它提供了服务中心类，可以简化网络服务器的开发。

3.2.1 socket()函数

在 Python 语言标准库中，通过使用 socket 模块提供的 socket 对象，可以在计算机网络中建立可以互相通信的服务器与客户端。在服务器端需要建立一个 socket 对象，并等待客户端的连接。客户端使用 socket 对象与服务器端进行连接，一旦连接成功，客户端和服务器端就可以进行通信了。

在 Python 中，通常用一个 Socket 表示“打开了一个网络连接”，语法格式如下：

```
socket.socket([family[, type[, proto]]])
```

其中参数 family: 套接字家族可以使 AF_UNIX 或者 AF_INET；type: 套接字类型可以根据是面向连接的还是非连接分为 SOCK_STREAM 或 SOCK_DGRAM；protocol: 一般不填默认为 0。

socket 主要分为面向连接的 socket 和无连接的 socket。面向连接的 socket 使用的主要协议是传输控制协议，也就是常说的 TCP，TCP 的 socket 名称是 SOCK_STREAM。创建套接字 TCP/IP 套接字，可以调用 socket.socket()。语法格式如下：

```
tcpSocket=socket.socket(AF_INET,SOCK_STREAM)
```

无连接 socket 的主要协议是用户数据报协议，也就是常说的 UDP，UDP 的 socket 的名字是 SOCK_DGRAM。创建套接字 UDP/IP 套接字，可以调用 socket.socket()。语法格式如下：

```
udpSocket=socket.socket (AF_INET,SOCK_DGRAM)
```

在 Python 语言中 socket 对象中，提供如表 3-1 所示的内置函数。

表 3-1 内置函数

函数	功能
服务器端套接字函数	
s.bind()	绑定地址(host,port)到套接字， 在 AF_INET 下,以元组(host,port)的形式表示地址。
s.listen()	开始 TCP 监听。backlog 指定在拒绝连接之前，操作系统可以挂起的最大连接数量。该值至少为 1，大部分应用程序设为 5 就可以了。

s.accept()	被动接受 TCP 客户端连接,(阻塞式)等待连接的到来
客户端套接字	
s.connect()	主动初始化 TCP 服务器连接，。一般 address 的格式为元组（hostname,port），如果连接出错，返回 socket.error 错误。
s.connect_ex()	connect()函数的扩展版本,出错时返回出错码,而不是抛出异常
公共用途的套接字函数	
s.recv()	接收 TCP 数据，数据以字符串形式返回，bufsize 指定要接收的最大数据量。flag 提供有关消息的其他信息，通常可以忽略。
s.send()	发送 TCP 数据，将 string 中的数据发送到连接的套接字。返回值是要发送的字节数量，该数量可能小于 string 的字节大小。
s.sendall()	完整发送 TCP 数据，完整发送 TCP 数据。将 string 中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回 None，失败则抛出异常。
s.recvfrom()	接收 UDP 数据，与 recv()类似，但返回值是（data,address）。其中 data 是包含接收数据的字符串，address 是发送数据的套接字地址。
s.sendto()	发送 UDP 数据,将数据发送到套接字,address 是形式为(ipaddr,port)的元组，指定远程地址。返回值是发送的字节数。
s.close()	关闭套接字
s.getpeername()	返回连接套接字的远程地址。返回值通常是元组（ipaddr,port）。
s.getsockname()	返回套接字自己的地址。通常是一个元组(ipaddr,port)
s.setsockopt(level,optname,value)	设置给定套接字选项的值。
s.getsockopt(level,optname[.buflen])	返回套接字选项的值。
s.settimeout(timeout)	设置套接字操作的超时期，timeout 是一个浮点数，单位是秒。值为 None 表示没有超时期。一般，超时期应该在刚创建套接字时设置，因为它们可能用于连接的操作（如 connect()）
s.gettimeout()	返回当前超时期的值，单位是秒，如果没有设置超时期，则返回 None。
s.fileno()	返回套接字的文件描述符。
s.setblocking(flag)	如果 flag 为 0，则将套接字设为非阻塞模式，否则将套接字设为阻塞模式（默认值）。非阻塞模式下，如果调用 recv()没有发现任何数据，或 send()调用无法立即发送数据，那么将引起 socket.error 异常。
s.makefile()	创建一个与该套接字相关连的文件

3.2.2 UDP 编程

TCP 是建立可靠连接，并且通信双方都可以以流的形式发送数据。相对 TCP，UDP 则是面向无连接的协议。使用 UDP 协议时，不需要建立连接，只需要知道对方的 IP 地址和端口号，就可以直接发数据包。但是，能不能到达就不知道了。虽然用 UDP 传输数据不可靠，但它的优点是和 TCP 比，速度快，对于不要求可靠到达的数据，就可以使用 UDP 协议。

创建 Socket 时，SOCK_DGRAM 指定了这个 Socket 的类型是 UDP。绑定端口和 TCP 一样，但是不需要调用 listen()方法，而是直接接收来自任何客户端的数据。recvfrom()方法返回数据和客户端的地址与端口，这样，服务器收到数据后，直接调用 sendto()就可以把数据用 UDP 发给客户端。

发送数据：为看到效果借助“网络调试助手”。双击 NetAssist.exe 就完成安装。使用详细如图 3-5 所示：

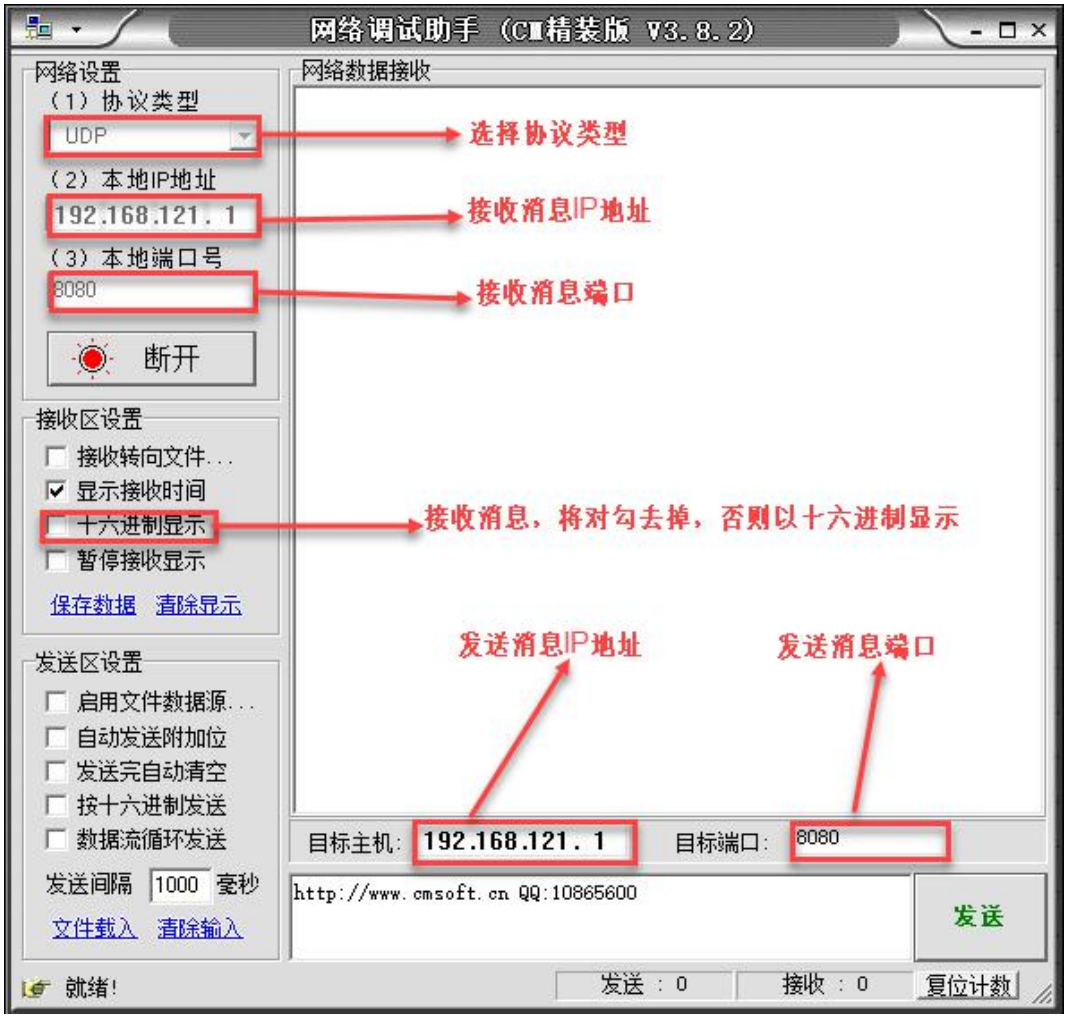


图 3-5 网络调试助手

【示例 3-1】使用 UDP 发送数据

```
from socket import *
s = socket(AF_INET, SOCK_DGRAM) #创建套接字
addr = ('192.168.121.1', 8080) #准备接收方地址
data = input("请输入要发送的内容：")
#发送数据时，python3 需要将字符串转成 byte
s.sendto(data.encode('gb2312'),addr) #默认的网络助手使用的编码是 gb2312
s.close()
```

首先运行上述代码输入要发送的内容，执行结果如图 3-6 所示：

图 3-6 示例 3-1 执行结果

打开网络调试助手协议类型选择 UDP 协议，输入本地 IP 地址、端口后点击连接。会收到发送的信息内容，如图 3-7 所示：

图 3-7 测试示例 3-1 网络调试助手执行结果

【示例 3-2】使用 UDP 先发送数据，再接收数据

```
from socket import *
s = socket(AF_INET, SOCK_DGRAM) #创建套接字

s.bind(('', 8788))    #绑定一个端口，ip 地址和端口号，ip一般不用写

addr = ('192.168.121.1', 8080)    #准备接收方地址
data = input("请输入：")
s.sendto(data.encode('gb2312'),addr)

redata = s.recvfrom(1024) #1024 表示本次接收的最大字节数

print(redata)
print('接收到%s 的消息：%s'%(redata[1],redata[0].decode('gb2312')))
s.close()
```

从上述代码中可以看到，创建了套接字并将其绑定到本地地址，运行程序，首先要输入发送的内容（按 Enter 键）。然后打开网络调试助手协议类型选择 UDP 协议，输入本地 IP

地址/端口后点击连接，会收到发送的信息内容。接着填写网络调试助手的目标主机和目标端口（程序中绑定的主机和端口），输入要发送的内容，点击发送。程序执行结果如图 3-8，网络助手执行结果如图 3-9 所示：

图 3-8 示例 3-2 执行结果

图 3-9 测试示例 3-2 网络调试助手执行结果

【示例 3-3】UDP 实现简单聊天

```
from socket import *
udp_socket=socket(AF_INET,SOCK_DGRAM)
#绑定端口
udp_socket.bind(('',8989))
```

```
addr = ('192.168.121.1', 8080)     #准备接收方地址
#接收数据
while True:
    msg = input('请输入：')
    udp_socket.sendto(msg.encode('gb2312'),addr)
    recv_data = udp_socket.recvfrom(1024)
    print('接收到%s 的消息是：%s' % (recv_data[1], recv_data[0].decode('gb2312')))
udp_socket.close()
```

从上述代码中可以看到，首先建立 UDP 连接之后使用 while 语句多次与网络助手进行数据交换。程序执行结果如图 3-10，网络调试助手执行结果如图 3-11 所示：

图 3-10 示例 3-3 执行结果

图 3-11 测试示例 3-3 网络调试助手执行结果

在上述的示例中，可以看到程序必须首先输入再接收，网络助手首先接收再输入。因此上述示例并没有实现真正的聊天。实际的聊天输入和接收可以是多次而且互不干扰。要想实现可以使用多线程并发编程。

【示例 3-4】UDP 使用多线程实现聊天

```
from socket import *
from threading import Thread
udp_socket=socket(AF_INET,SOCK_DGRAM)
#绑定端口
udp_socket.bind(('',8989))


#不停接收
def recv_data():
```

```
        while True:
            recv_msg=udp_socket.recvfrom(1024)
            print('>>%s:%s'%(recv_msg[1],recv_msg[0].decode('gb2312')))
#不停发送
def send_data():
        while True:
            data=input('<<：')
            addr=('192.168.121.1',8080)
            udp_socket.sendto(data.encode('gb2312'),addr)


if __name__=='__main__':
        # 创建两个线程
        t1=Thread(target=send_data)
        t2=Thread(target=recv_data)
        t2.start()
        t1.start()
        t1.join()
        t2.join()
```

上述示例代码中，定义两个线程分别接收消息和发送消息。这样接收和发送消息就可以并发执行，执行结果如图 3-12 所示：

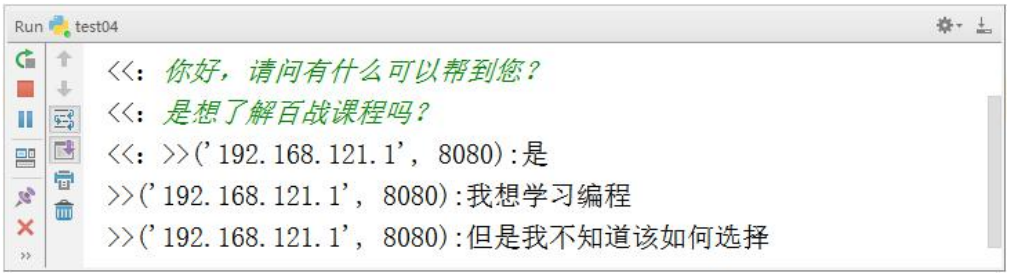


图 3-12 示例 3-4 执行结果

3.2.3 TFTP 文件下载器

TFTP（Trivial File Transfer Protocol）是简单文件传输协议，使用这个协议就可以实现简单文件的下载，tftp 主要特点有：

- 简单
- 占用资源少

- 适合传递小文件
- 适合在局域网进行传递
- 端口号为 69
- 基于 UDP 实现

实现 TFTP 下载器简单说就是从服务器上将一个文件复制到本机上，首先在本地创建一个空文件（要与下载的文件同名）向里面写数据（接收到一点就向空文件里写一点），当接收完所有数据关闭文件。

（1）TFTP 文件下载过程

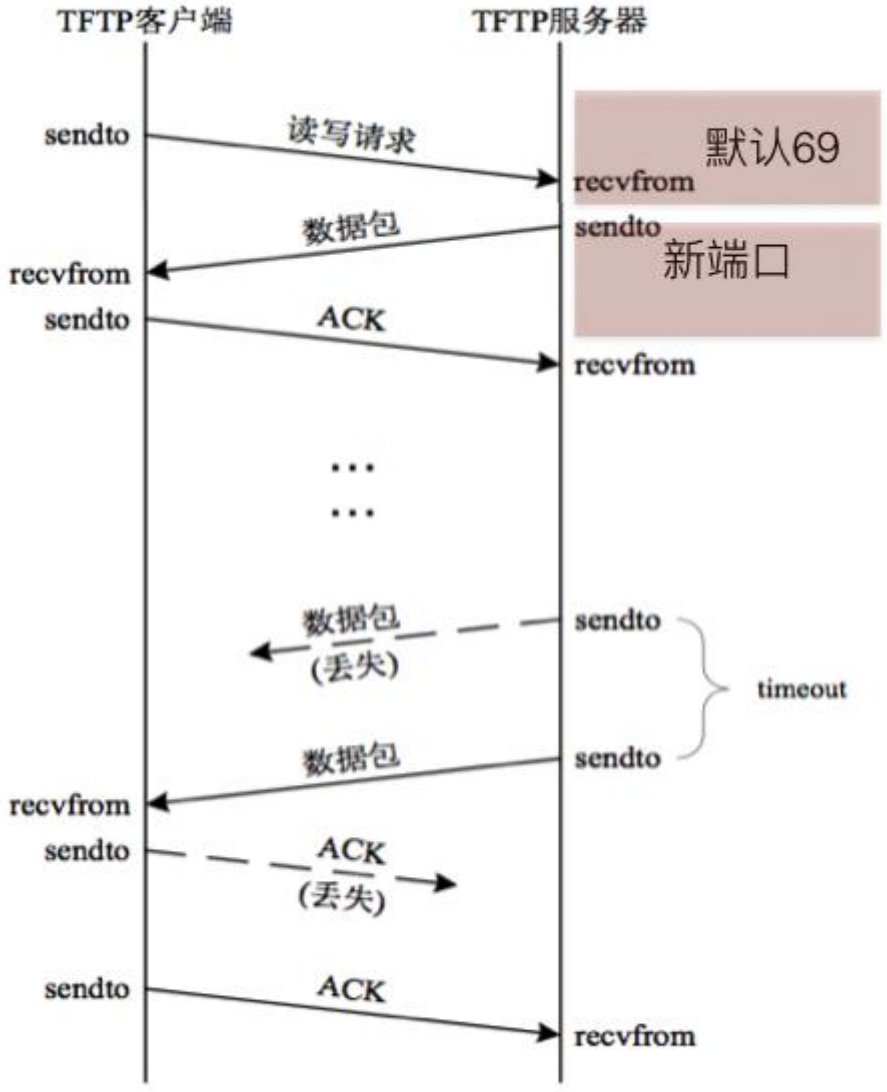


图 3-13 TFTP 文件下载过程

1、搜索

当服务器找到需要现在的文件后，会立刻打开文件，把文件中的数据通过 TFTP 协议发

送给客户端。

2、分段

如果文件的总大小较大（比如 3M），那么服务器分多次发送，每次会从文件中读取 512 个字节的数据发送过来。

3、添加序号

因为发送的次数有可能会很多，所以为了让客户端对接收到的数据进行排序，所以在服务器发送那 512 个字节数据的时候，会多发 2 个字节的数据，用来存放序号，并且放在 512 个字节数据的前面，序号是从 1 开始的。

4、添加操作码

因为需要从服务器上下载文件时，文件可能不存在，那么此时服务器就会发送一个错误的信息过来，为了区分服务发送的是文件内容还是错误的提示信息，所以又用了 2 个字节来表示这个数据包的功能（称为操作码），并且在序号的前面。

表 3-2 操作码

操作码	功能
1	读请求，即下载
2	写请求，即上传
3	表示数据包，即 DATA
4	确认码，即 ACK
5	错误

5、发送确认码（ACK）

因为 udp 的数据包不安全，即发送方发送是否成功不能确定，所以 TFTP 协议中规定，为了让服务器知道客户端已经接收到了刚刚发送的那个数据包，所以当客户端接收到一个数据包的时候需要向服务器进行发送确认信息，即发送收到了，这样的包成为 ACK(应答包)。

6.发送完毕

为了标记数据已经发送完毕，所以规定，当客户端接收到的数据小于 516（2 字节操作码+2 个字节的序号+512 字节数据）时，就意味着服务器发送完毕了（如果恰好最后一次数据长度为 516，会再发一个长度为 0 的数据包）。

（2）TFTP 数据包格式

图 3-14 TFTP 数据包格式

struct 模块可以按照指定格式将 Python 数据转换为字符串，该字符串为字节流。struct 模块中最重要的三个函数是 pack()、unpack()、calcsize()。

表 3-3 内置函数

函数名	描述
pack(fmt, v1, v2, ...)	按照给定的格式(fmt)，把数据封装成字符串(实际上是类似于 c 结构体的字节流)
unpack(fmt, string)	按照给定的格式(fmt)解析字节流 string，返回解析出来的元组
calcsize(fmt)	计算给定的格式(fmt)占用多少字节的内存

【示例 3-5】构造下载请求数据：“1test.jpg0octet0”

```
import struct
cmb_buf = struct.pack("!H8sb5sb",1,b"test.jpg",0,b"octet",0)
# 如何保证操作码（1/2/3/4/5）占两个字节？如何保证 0 占一个字节？
#!H8sb5sb:！表示按照网络传输数据要求的形式来组织数据（占位的格式）
# H 表示将后面的 1 替换成占两个字节
# 8s 相当于 8 个 s（ssssssss）占 8 个字节
# b 占一个字节
```

【示例 3-6】TFTP 文件下载客户端

```
import struct
```

```
from socket import *
filename = 'test.jpg'
server_ip = '127.0.0.1'
send_data = struct.pack('!H%dsb5sb' % len(filename), 1, filename.encode(), 0, 'octet'.encode(), 0)
s = socket(AF_INET, SOCK_DGRAM)
s.sendto(send_data, (server_ip, 69))  # 第一次发送，连接服务器 69 端口
f = open(filename, 'ab')  #a:以追加模式打开（必要时可以创建）append;b:表示二进制
while True:
    recv_data = s.recvfrom(1024)  # 接收数据
    caozuoma, ack_num = struct.unpack('!HH', recv_data[0][:4])  # 获取数据块编号
    rand_port = recv_data[1][1]  # 获取服务器的随机端口

    if int(caozuoma) == 5:
        print('文件不存在...')
        break
    print("操作码：%d,ACK：%d,服务器随机端口：%d,数据长度：%d"%(caozuoma, ack_num, rand_port, len(recv_data[0])))
    f.write(recv_data[0][4:])#将数据写入
    if len(recv_data[0]) < 516:
        break
    ack_data = struct.pack("!HH", 4, ack_num)
    s.sendto(ack_data, (server_ip, rand_port))  # 回复 ACK 确认包
```

上诉写的是 TFTP 客户端的代码实现，要想实现客户端的下载功能，首先要有一个 TFTP 服务器，从网上可以下载并安装 Tftp32 服务器软件。打开 Tftp32 服务器软件，将要下载的文件上传到服务器如图 3-15 所示：

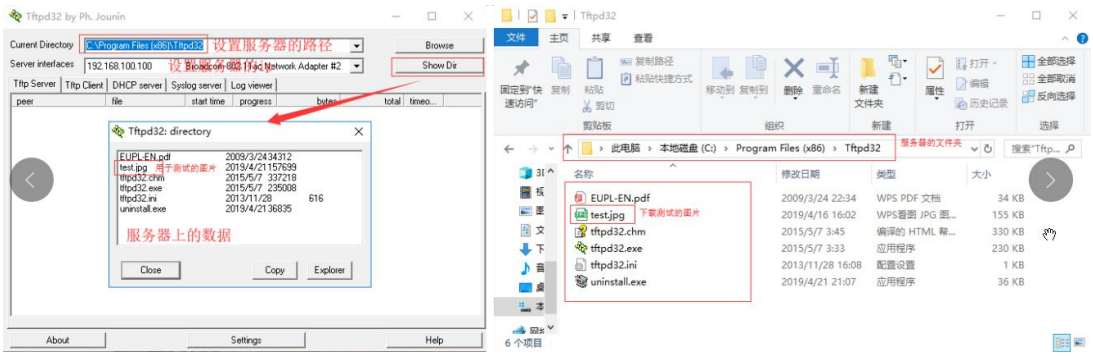


图 3-15 TFTP 服务器

3.2.4 TCP 编程

面向连接的 Socket 使用的主要协议是传输控制协议，也就是常说的 TCP，TCP 的 Socket 名称是 SOCK_STREAM。创建套接字 TCP/IP 套接字，可以调用 socket.socket()。语法格式如下：

```
tcpSocket=socket.socket(AF_INET,SOCK_STREAM)
```

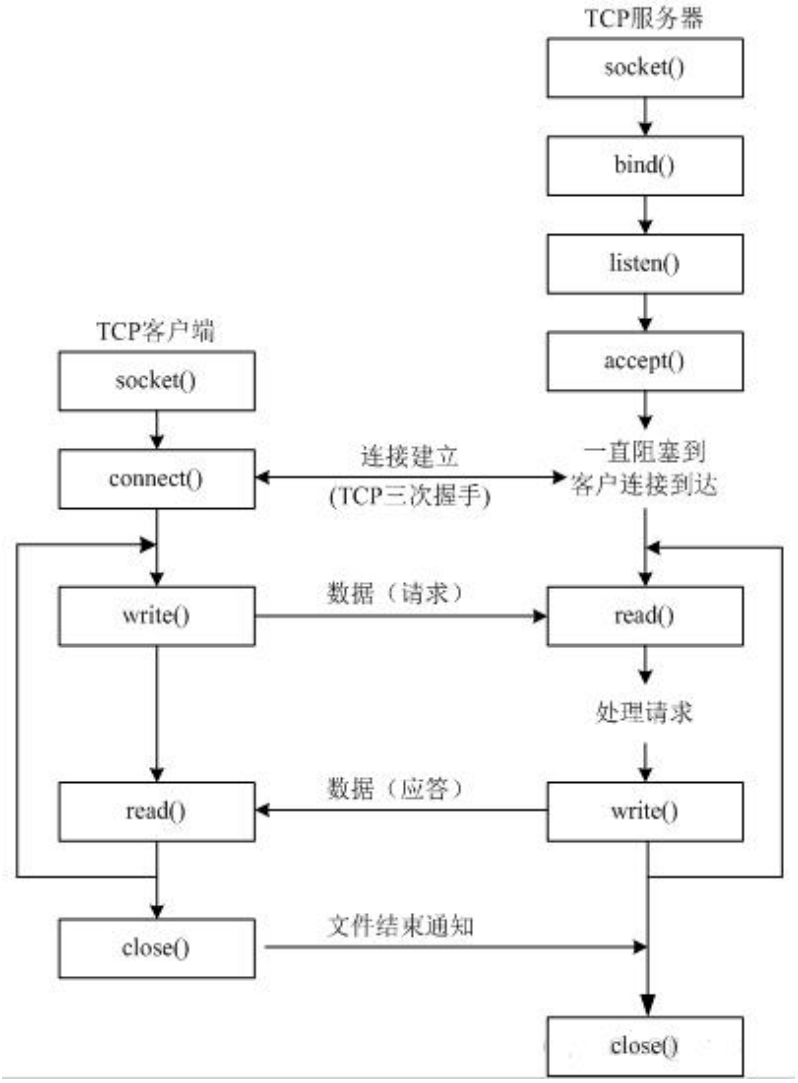


图 3-16 TCP 执行流程

3.2.5 三次握手

- 第一步，客户端发送一个包含 SYN 即同步（Synchronize）标志的 TCP 报文，SYN 同步报文会指明客户端使用的端口以及 TCP 连接的初始序号。

- 第二步，服务器在收到客户端的 SYN 报文后，将返回一个 SYN+ACK 的报文，表示客户端的请求被接受，同时 TCP 序号被加一，ACK 即确认（Acknowledgement）。
- 第三步，客户端也返回一个确认报文 ACK 给服务器端，同样 TCP 序列号被加一，到此一个 TCP 连接完成。然后才开始通信的第二步：数据处理。
- 这就是所说的 TCP 的三次握手（Three-way Handshake）。

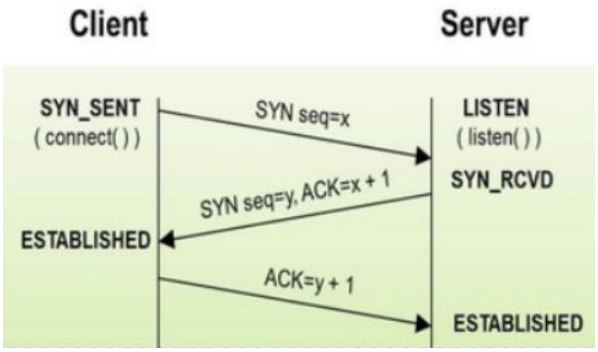


图 3-17 三次握手

在 Python 语言中创建 Socket 服务端程序，需要使用 socket 模块中的 socket 类。创建 Socket 服务器程序的步骤如下：

（1） 创建 Socket 对象。

（2） 绑定端口号。

（3） 监听端口号。

（4） 等待客户端 Socket 的连接。

（5） 读取客户端发送过来的数据。

（6） 向客户端发送数据。

（7） 关闭客户端 Socket 连接。

（8） 关闭服务端 Socket 连接。

【示例 3-7】TCP 服务器端接收数据

```
from socket import *
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(("", 8899))
serverSocket.listen(5)
clientSocket,clientInfo = serverSocket.accept()
#clientSocket  表示这个新的客户端
#clientInfo  表示这个新的客户端的 ip 以及 port
recvData = clientSocket.recv(1024)
print("%s:%s"%(str(clientInfo), recvData.decode('gb2312')))
clientSocket.close()
```

```
serverSocket.close()
```

在上述示例代码中，建立 TCP 连接之后，服务器端会等待客户端连接，打开网络调试助手，网络协议类型选择 TCP Client，填写服务器的 IP/端口。输入消息点击发送如图 3-18 所示：

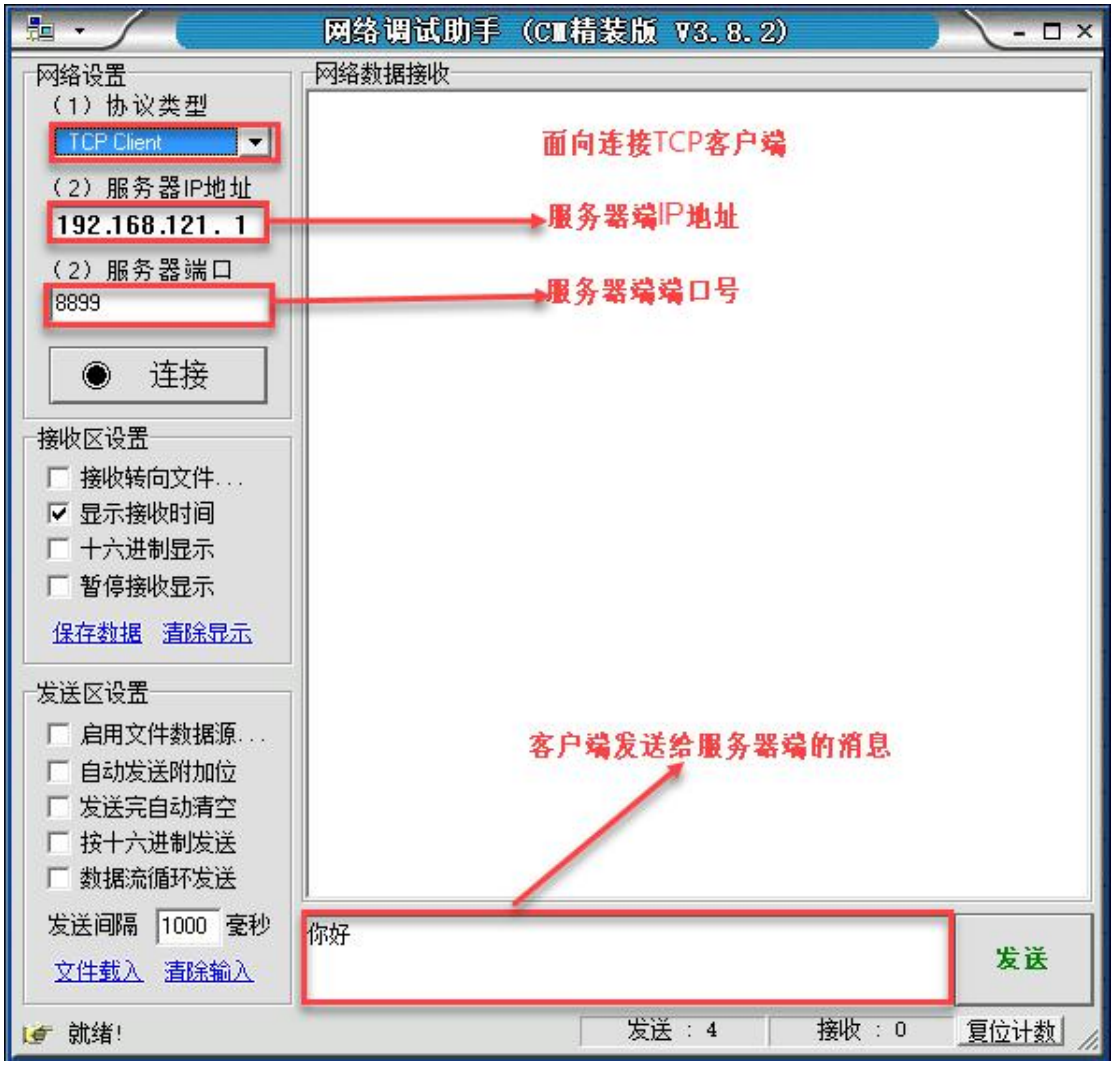


图 3-18 测试示例 3-7 网络调试助手执行结果

程序控制台就会接收到消息，如图 3-8 所示：

图 3-19 示例 3-7 执行结果

【示例 3-8】TCP 客户端

```
from socket import *
clientSocket = socket(AF_INET, SOCK_STREAM)
```

```
clientSocket.connect(("192.168.121.1", 8899))
#注意：
# 1. tcp 客户端已经链接好了服务器，所以在以后的数据发送中，不需要填写对方的 ip 和 port----->如打电话
# 2. udp 在发送数据的时候，因为没有之前的链接，所依需要　在每次的发送中　都要填写接收方的 ip 和 port－－－－－＞如写信
clientSocket.send("haha".encode("gb2312"))
recvData = clientSocket.recv(1024)
print("recvData:%s"%recvData.decode('gb2312'))
clientSocket.close()
```

先配置网络助手，选择网络协议为 TCP Server，填写本地 IP/端口，点击连接，执行编写的 TCP 客户端代码，网络调试助手会收到消息如图 3-20 所示：

图 3-20 测试示例 3-8 网络调试助手执行结果

在网络调试助手中输入要发送的信息，点击发送，程序控制台输出接收信息如图 3-21 所示：

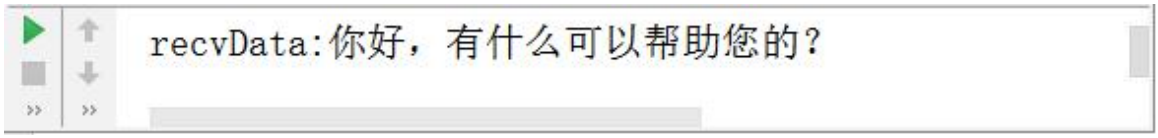


图 3-21 示例 3-8 执行结果

【示例 3-9】TCP：双向通信 Socket 之服务器端

```
'''
双向通信 Socket 之服务器端
        读取客户端发送的数据，将内容输出到控制台
        将控制台输入的信息发送给客户器端
'''
#导入 socket 模块
from socket import *
#创建 Socket 对象
tcpServerSocket=socket(AF_INET,SOCK_STREAM)
#绑定端口
tcpServerSocket.bind(('',8888))
#监听客户端的连接
tcpServerSocket.listen()
#接收客户端连接
tcpClientSocket,host=tcpServerSocket.accept()
while True:
    #读取客户端的消息
    re_data=tcpClientSocket.recv(1024).decode('utf-8')
    #将消息输出到控制台
    print('客户端说：',re_data)
    if re_data=='end':
        break
    #获取控制台信息
    msg=input('>')
    tcpClientSocket.send(msg.encode('utf-8'))
tcpClientSocket.close()
tcpServerSocket.close()
```

【示例 3-10】TCP：双向通信 Socket 之客户端

```
'''
双向通信 Socket 之客户端
        将控制台输入的信息发送给服务器端
        读取服务器端的数据，将内容输出到控制台
'''
#导入 socket 模块
from socket import *
#创建客户端 Socket 对象
tcpClientSocket=socket(AF_INET,SOCK_STREAM)
#连接服务器端
tcpClientSocket.connect(('127.0.0.1',8888))
while True:
    msg=input('>')
    #向服务器发送数据
    tcpClientSocket.send(msg.encode('utf-8'))
    if msg=='end':
        break
    #接收服务器端数据
    re_data=tcpClientSocket.recv(1024)
    print('服务器端说：',re_data.decode('utf-8'))
tcpClientSocket.close()
```

上述示例代码客户端与服务器以 TCP 方式建立连接，首先客户端发送数据给服务器端，服务器收到信息后回信给客户端，客户端和服务器端使用 while 语句多次进行数据交换，直到收到的数据为 end 时才会终止服务器的运行。执行结果如图 3-22 所示：

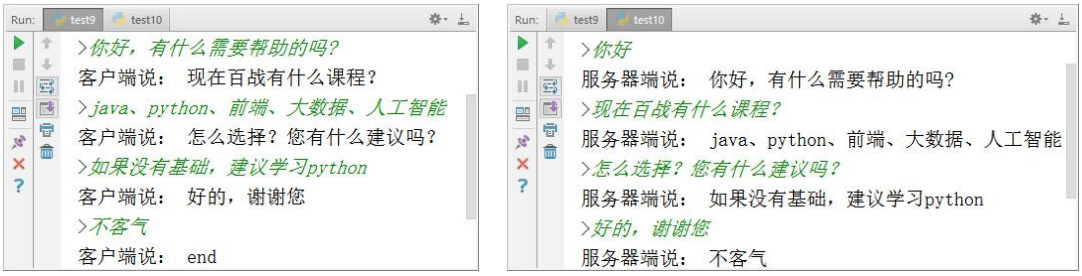


图 3-22 示例 3-9、示例 3-10 执行结果

注意事项

- 运行时，要先启动服务器端，再启动客户端，才能得到正常的运行效果。

上述示例代码，必须按照安排好的顺序，服务器和客户端一问一答！不够灵活！！可以

使用多线程实现更加灵活的双向通讯！！

服务器端：一个线程专门发送消息，一个线程专门接收消息。

客户端：一个线程专门发送消息，一个线程专门接收消息。

习题

一、选择题

1. 以下协议都属于 TCP/IP 协议栈，其中位于传输层的协议是（ ）。（多选）

A TCP

B. HTTP

C. SMTP

D. UDP

2. 以下说法中关于 UDP 协议的说法正确的是（ ）。（多选）

A. 发送不管对方是否准备好，接收方收到也不确认

B. 面向连接

C. 占用系统资源多、效率低

D. 非常简单的协议，可以广播发送

3. 使用客户端套接字 Socket 连接服务器时，需要指定（ ）。

A 服务器主机名称和端口

B. 服务器端口和文件

C. 服务器名称和文件

D. 服务器地址和文件

4. 在开发 Python 的网络程序中，协议打包成二进制，经常会用到模块____（ ）

A、pickle

B、pack

C、network

D、struct

5. 下面的网络协议中，面向连接的的协议是什么？（ ）

A 传输控制协议

B 用户数据报协议

C 网际协议

D 网际控制报文协议

二、解答题

1. 简述 OSI 七层协议。
2. TCP 和 UDP 的区别?
3. 为何基于 tcp 协议的通信比基于 udp 协议的通信更可靠？
4. 简述基于 TCP 的 Socket 编程的主要步骤。
5. 通过类比打电话，详细描述三次握手机制。

三、编码题

1．使用基于 UDP 的 Socket 编程，完成在线咨询功能：

1) 客户向咨询人员咨询。

2) 咨询人员给出回答。

3) 客户和咨询人员可以一直沟通，直到客户发送 bye 给咨询人员。

2．使用基于 TCP 编程，完成如下功能：

1) 要求从客户端录入几个字符，发送到服务器端。

2) 由服务器端将接收到的字符进行输出。

3) 服务器端向客户端发出“您的信息已收到”作为响应。

4) 客户端接收服务器端的响应信息。

第四章 GUI 图形用户界面编程

前面实现的都是基于控制台的程序，程序和用户的交互通过控制台来完成；本章将学习 GUI（Graphics User Interface GUI），即图形用户界面编程，可以通过 python 提供的丰富的组件，快速的实现使用图形界面和用户交互。

通过阅读本章，你可以：

- 了解什么是 tkinter
- 掌握使用 tkinter 的三种布局
- 掌握 tkinter 的常用控件
- 掌握 tkinter 中常用对话框的使用

4.1 常用的 GUI 库

作为一门功能强大的的面向对象编程语言，Python 能够通过 GUI 库开发出专业的 GUI 程序。在开发 Python 程序过程中，比较常用的 GUI 库如下。

（1） tkinter

tkinter（Tk interface）是 Python 的标准 GUI 库，支持跨平台的 GUI 程序开发。tkinter 适合小型的 GUI 程序编写，也特别适合初学者学习 GUI 编程。本书以 tkinter 为核心进行讲解。

（2） wxPython

wxPython 是比较流行的 GUI 库，适合大型应用程序开发，功能强于 tkinter，整体设计框架类似于 MFC(Microsoft Foundation Classes 微软基础类库)。

（3）PyQT

Qt 是一种开源的 GUI 库，适合大型 GUI 程序开发，PyQT 是 Qt 工具包标准的 Python 实现。我们也可以使用 Qt Desginer 界面设计器快速开发 GUI 应用程序。

4.2 编写第一个 tkinter 程序

使用 tkinter 模块开发 GUI 应用的基本步骤如下所诉：

（1） 导入 tkinter 模块。


```
from tkinter import *
```

（2） 创建 Tk 类的实例，Tk 对象表示一个窗口。

```
root = Tk()
```

（3） 对窗口进行设置，如通过 title 方法设置窗口的标题，通过 geometry 方法设置窗

口的尺寸和位置。

```
root.title("第一个 tkinter 程序")
#宽度 500，高度 400；距屏幕左边 100，距屏幕上边 200
root.geometry("500x400+100+200")
```

（4） 创建控件类的实例，并将控件添加到窗口上。比如：按钮（Button）、文本等。

```
btn01 = Button(root)
btn01["text"] = "点我有惊喜"
```

（5） 通过几何布局管理器，管理组件的大小和位置。

```
btn01.pack()
```

（6） 通过绑定事件处理程序，响应用户操作所触发的事件（比如：单击、双击等）。

```
def   songhua(e):
      messagebox.showinfo("Message","送你一朵玫瑰花 ")
      print('送给你一朵玫瑰花')
btn01.bind("<Button-1>",songhua)
```

【示例 4-1】第一个 tkinter 程序

```
#导入模块
from tkinter import *
from tkinter import messagebox
#创建窗口对象
root=Tk()
#设置窗口的标题
root.title("tkinter 的第一个程序")
#设置窗口的大小
#宽度 500，高度 400；距屏幕左边 100，距屏幕上边 200
root.geometry("500x400+100+200")
#创建组件
button=Button(root)
button['text']='点我有惊喜'
button.pack()
#定义事件函数
def songhua(e):
```

```
    messagebox.showerror('Message','送给你一朵玫瑰花')
    print('送给你一朵玫瑰花')

#给控件绑定事件
button.bind('<Button-1>',songhua)
#调用组件的 mainloop 方法，进入事件循环
root.mainloop()
```

执行结果如图 4-1 所示：

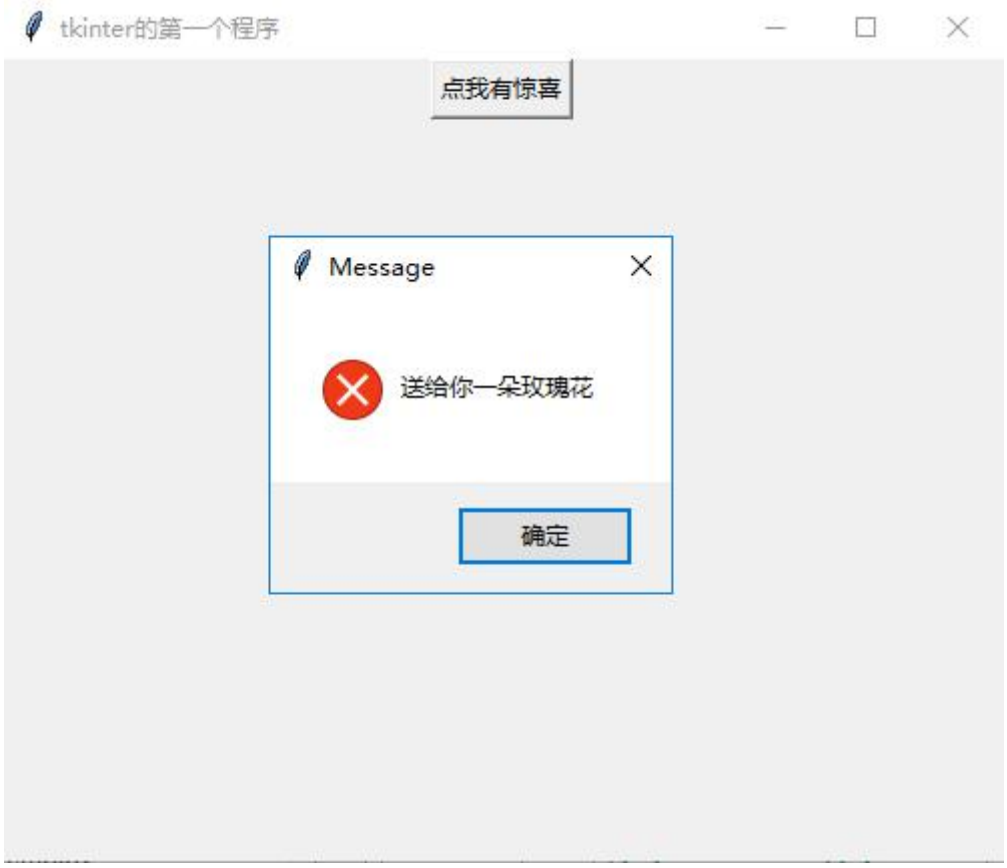
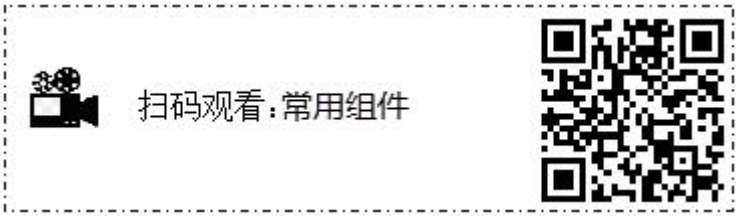


图 4-1 示例 4-1 运行效果图

4.3 常用组件



Tkinter 的提供各种控件，如按钮，标签和文本框，一个 GUI 应用程序中使用。这些控件通常被称为控件或者部件。这些组件及其简短的介绍如表 4-1 所示：

表 4-1 常用组件

控件	描述
Button	按钮控件；在程序中显示按钮。
Canvas	画布控件；显示图形元素如线条或文本

控件	描述
Checkbutton	多选框控件；用于在程序中提供多项选择框
Entry	输入控件；用于显示简单的文本内容
Frame	框架控件；在屏幕上显示一个矩形区域，多用来作为容器
Label	标签控件；可以显示文本和位图
Listbox	列表框控件；在 Listbox 窗口小部件是用来显示一个字符串列表给用户
Menubutton	菜单按钮控件，由于显示菜单项。
Menu	菜单控件；显示菜单栏,下拉菜单和弹出菜单
Message	消息控件；用来显示多行文本，与 label 比较类似
Radiobutton	单选按钮控件；显示一个单选的按钮状态
Scale	范围控件；显示一个数值刻度，为输出限定范围的数字区间
Scrollbar	滚动条控件，当内容超过可视化区域时使用，如列表框。.
Text	文本控件；用于显示多行文本
Toplevel	容器控件；用来提供一个单独的对话框，和 Frame 比较类似
Spinbox	输入控件；与 Entry 类似，但是可以指定输入范围值
PanedWindow	PanedWindow 是一个窗口布局管理的插件，可以包含一个或者多个子控件。
LabelFrame	labelframe 是一个简单的容器控件。常用与复杂的窗口布局。
tkMessageBox	用于显示你应用程序的消息框。

4.3.1 Label 标签

Label（标签）主要用于显示文本信息，也可以显示图像。Label（标签）的常见属性如表 4-2 所示：

表 4-2 Label（标签）的常见属性

属性名	功能
width,height	用于指定区域大小，如果显示是文本，则以单个字符大小为单位；如果显示是图像，则以像素为单位。默认值是根据具体显示的内容动态调整
font	指定字体和字体大小，如：font = (font_name,size)
image	显示在 Label 上的图像
fg 和 bg	fg（foreground）:前景色、bg（background）:背景色
justify	针对多行文字的对齐，可设置 justify 属性，可选值"left", "center" or "right"
borderwidth（bd）	指标签的边框宽度
text	标签中的文本，可以使用“\n”表示换行

【示例 4-2】Label（标签）的用法

```
#Label 标签的用法
from tkinter import *
root = Tk()
root.title("测试 Label 的用法")
root.geometry("300x300+200+200")

#文本的显示
widget1 = Label(root,text="百战程序员",width=10,height=1,bg="black",fg="white")
print(widget1.keys())
widget1.pack()

#控制字体颜色，大小
widget2 = Label(root,text="高淇老师",width=20,bg="#cc00cc",font=("黑体",30))
widget2.pack()

#显示图像
photo = PhotoImage(file="images/logo.gif")        #tkinter 里面只支持 gif 格式
widget3 = Label(root,image=photo)
widget3.pack()
#显示多行文本
widget4 = Label(root,text="北京尚学堂\n 百战程序员\nPython400 集打造完美教程",\
                        borderwidth=1,relief="solid",justify="right")
widget4.pack()

root.mainloop()
```

执行结果如图 4-2 所示：

图 4-2 示例 4-2 运行效果图

4.3.2 Button

Button（按钮）用来执行用户的单击操作。Button 可以包含文本，也可以包含图像。按钮被单击后会自动调用对应事件绑定的方法。常用的属性参数如表 4-3 所示：

表 4-3 Button（按钮）的常见属性

属性名	功能
bg	按钮的背景色
command	按钮关联的函数，当按钮被点击时，执行该函数
font	文本字体
height	按钮的高度
image	按钮上要显示的图片
justify	显示多行文本的时候,设置不同行之间的对齐方式,可选项包括 LEFT, RIGHT, CENTER
padx	按钮在 x 轴方向上的内边距(padding)，是指按钮的内容与按钮边缘的距离
pady	按钮在 y 轴方向上的内边距(padding)
state	设置按钮组件状态,可选的有 NORMAL、ACTIVE、 DISABLED。默认 NORMAL。
width	按钮的宽度，如未设置此项，其大小以适应按钮的内容（文本或图片的大小）
text	按钮的文本内容
anchor	锚选项，控制文本的位置，默认为中心

【示例 4-3】Button 按钮用法(文字、图片、事件)

```
#Button 按钮用法(文字、图片、事件)
from tkinter import *
from tkinter import messagebox
root = Tk();
root.title("Button 用法示例")
root.geometry("300x150")

def login():
    messagebox.showinfo('Message','我要登录了！ ')
    print("我要登录了！ ")

btn01 = Button(root,text="登录",command=login)
btn01.pack()

photo = PhotoImage(file="images/start.gif")
btn02 = Button(root,image=photo)
btn02.pack()
# btn02.config(state="disabled")    #设置按钮为禁用
root.mainloop()
```

执行结果如图 4-3 所示：

图 4-3 示例 4-3 运行效果图

4.3.3 Entry 单行文本框

Entry 用来接收一行字符串的控件。如果用户输入的文字长度长于 Entry 控件的

宽度时，文字会自动向后滚动。如果想输入多行文本，需要使用 Text 控件。常用的属性控制参数如表 4-4 所示，常用的方法如表 4-5 所示：

表 4-4 Entry（单行文本框）的常见属性

属性名	功能
bg	文本框的背景色
bd	边框的大小，默认为 2 个像素
cursor	光标的形状设定，如 arrow, circle, cross, plus 等
font	文本字体
exportselection	默认情况下，你如果在输入框中选中文本，默认会复制到粘贴板，如果要忽略这个功能刻工艺设置 exportselection=0。
fg	文字颜色。值为颜色或为颜色代码，如：'red','#ff0000'
highlightcolor	文本框高亮边框颜色，当文本框获取焦点时显示
justify	显示多行文本的时候,设置不同行之间的对齐方式，可选项包括 LEFT, RIGHT, CENTER
relief	边框样式，设置控件 3D 效果，可选的有：FLAT、SUNKEN、RAISED、GROOVE、RIDGE。默认为 FLAT。
selectbackground	选中文字的背景颜色
selectborderwidth	选中文字的背景边框宽度
selectforeground	选中文字的颜色
show	指定文本框内容显示为字符,值随意,满足字符即可。如密码可以将值设为 show="*"
state	默认为 state=NORMAL, 文框状态，分为只读和可写，值为：normal/disabled
textvariable	文本框的值，是一个 StringVar()对象
width	文本框宽度
xscrollcommand	设置水平方向滚动条，一般在用户输入的文本框内容宽度大于文本框显示的宽度时使用。

表 4-5 Entry（单行文本框）的常见方法

方法名	功能
delete(first,last=None)	删除文本框里直接位置值 text.delete(10)　　# 删除索引值为 10 的值 text.delete(10, 20)　# 删除索引值从 10 到 20 之前的值 text.delete(0, END)　# 删除所有值
get()	获取文件框的值
icursor (index)	将光标移动到指定索引位置，只有当文框获取焦点后成立
index (index)	返回指定的索引值

insert (index, s)	向文本框中插入值，index：插入位置，s：插入值
select_adjust (index)	选中指定索引和光标所在位置之前的值
select_clear()	清空文本框
select_from (index)	设置光标的位置，通过索引值 index 来设置
select_present()	如果有选中，返回 true，否则返回 false。
select_range (start, end)	选中指定索引位置的值,start(包含) 为开始位置,end(不包含) 为结束位置 start 必须比 end 小
select_to (index)	选中指定索引与光标之间的值
xview (index)	该方法在文本框链接到水平滚动条上很有用。
xview_scroll (number, what)	用于水平滚动文本框。what 参数可以是 UNITS, 按字符宽度滚动，或者可以是 PAGES, 按文本框组件块滚动。 number 参数，正数为由左到右滚动，负数为由右到左滚动。

【示例 4-4】Entry 单行文本框实现简单登录界面

```
#测试 Entry 单行文本框
from tkinter import   *
root = Tk()
#设置标题
root.title('登录界面')
#设置窗口大小
root.geometry("230x160+100+200")
label1 = Label(root,text="用户名")
label1.pack()
#StringVar 变量绑定到指定的组件上。改变 StringVar 的值时，组件的内容也发生改变。
v1 = StringVar()
entry1 = Entry(root,textvariable=v1)
entry1.pack()
v1.set("admin")
#设置密码
Label(root,text="密码").pack()
v2 = StringVar()
entry2 = Entry(root,textvariable=v2,show="*")
entry2.pack()
def confirm():
```

```
    print("用户名：",v1.get())
    print("密码：",v2.get())


Button(root,text="确定",command=confirm).pack()
root.mainloop()
```

执行结果如图 4-4 所示：

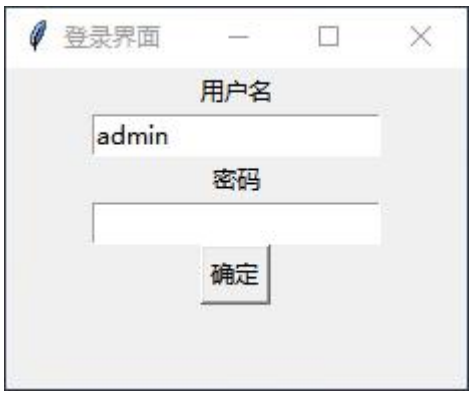


图 4-4 示例 4-4 运行效果图

4.3.4 Text 多行文本框

Text(多行文本框)的主要用于显示多行文本，还可以显示网页链接、图片、HTML 页面，甚至 CSS 样式表，添加组件等。因此，也常被当做简单的文本处理器、文本编辑器或者网页浏览器来使用。比如 IDLE 就是 Text 组件构成的。

【示例 4-5】Text 多行文本框基本用法(文本输入、组件和图像显示)

```
from tkinter import *
root = Tk();
#设置窗口的大小
root.geometry("300x300+400+400")
#宽度 20 个字母(10 个汉字)，高度一个行高
w1 = Text(root,width=40,height=20)
w1.pack()
# 核心：行号以 1 开始 列号以 0 开始
w1.insert(1.0,"0123456789\nabcdefg")
w1.insert(2.0,"锄禾日当午，汗滴禾下土。谁知盘中餐，粒粒皆辛苦")
```

```
#INSERT 索引表示在光标处插入
w1.insert(INSERT,'I Love')
#END 索引号表示在最后插入
w1.insert(END,' you')
photo = PhotoImage(file="images/logo.gif")
def show():
    print('我被点了一下')
    #  添加图片用 image_create
    w1.image_create(END, image=photo)
    # Indexes(索引)是用来指向 Text 组件中文本的位置，Text 的组件索引也是对应实际字符之间的位置。
    #  核心：行号以 1 开始  列号以 0 开始
    print(w1.get(1.2, 1.6))
    w1.insert(1.8, "gaoqi")
    print("所有文本内容：",w1.get(1.0,END))
# text 还可以插入按钮   图片等
b1 = Button(w1, text='爱尚学堂', command=show)
#  在 text 创建组件的命令
w1.window_create(INSERT, window=b1)
root.mainloop()
```

执行结果如图 4-5 所示：

图 4-5 示例 4-5 运行效果图

Tags 通常用于改变 Text 组件中内容的样式和功能。你可以修改文本的字体、尺寸和颜

色。另外，Tags 还允许你将文本、嵌入的组件和图片与鼠标和键盘等事件相关联。

【示例 4-6】Tags 实现更加强大的文本显示和控制

```
from tkinter import *
import webbrowser
root = Tk();
root.geometry("300x300+400+400")
#宽度 20 个字母(10 个汉字)，高度一个行高
w1 = Text(root,width=40,height=20)
w1.pack()
w1.insert(INSERT,"good good study,day day up!\n 北京尚学堂\n 百战程序员\n 百度，搜一下就知道")
w1.tag_add("good",1.0,1.9)
w1.tag_config("good",background="yellow",foreground="red")

w1.tag_add("baidu",4.0,END)
w1.tag_config("baidu",underline=True)

def webshow(event):
    webbrowser.open("http://www.baidu.com")

w1.tag_bind("baidu","<Button-1>",webshow)

root.mainloop()
```

执行结果如图 4-6 所示：

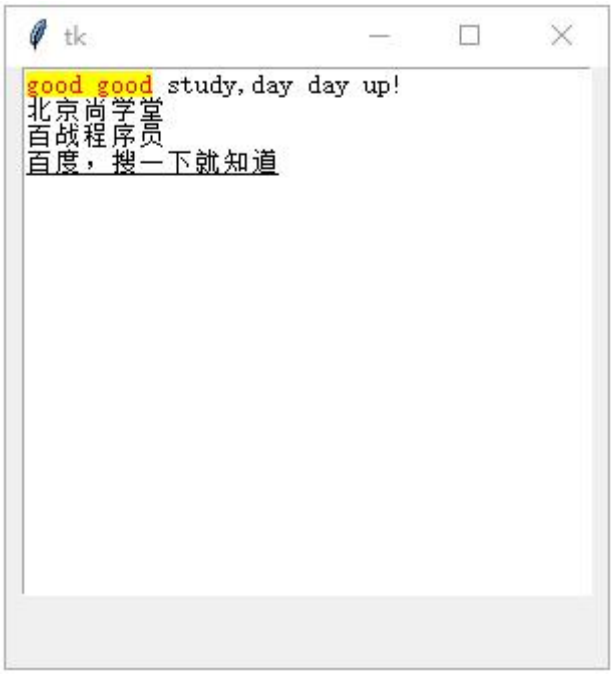
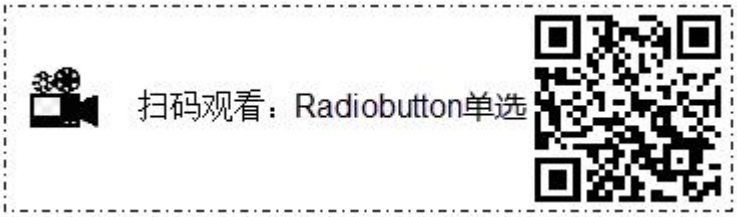


图 4-6 示例 4-6 运行效果图

点击“百度，搜一下就知道”后，系统默认浏览器打开百度页面。

4.3.5 Radiobutton 单选按钮

Radiobutton 控件用于选择同一组单选按钮中的一个。Radiobutton 可以显示文本，也可以显示图像。



【示例 4-7】Radiobutton 基础用法

```
from tkinter import *
root = Tk();
root.title('单选按钮的基本用法')
root.geometry("300x100+400+400")
#初始化变量的值
v = StringVar();
v.set("M")
#创建单选按钮
r1 = Radiobutton(root,text="男性",value="M",variable=v)
r2 = Radiobutton(root,text="女性",value="F",variable=v)
#将按钮添加到窗口
r1.pack();
```

```
r2.pack()
#定义点击事件处理函数
def confirm():
    print("用户选择的性别：",v.get())
#创建按钮
Button(root,text="确定",command=confirm).pack()
root.mainloop()
```

执行结果如图 4-7 所示：

图 4-7 示例 4-7 运行效果图

4.3.6 Checkbutton 复选按钮

Checkbutton 控件用于选择多个按钮的情况。Checkbutton 可以显示文本，也可以显示图像。

【示例 4-8】Checkbutton 复选按钮基础用法

```
from tkinter import *
root = Tk();
root.title('复选框的使用')
root.geometry("300x100+400+400")
#初始化变量
codeHobby = IntVar();
videoHobby=IntVar();
musicHobby=IntVar()
# print(codeHobby.get())     #默认值是 0
#创建复选框组件
c1 = Checkbutton(root,text="敲代码",variable=codeHobby,onvalue=1,offvalue=0)
c2 = Checkbutton(root,text="看视频",variable=videoHobby,onvalue=1,offvalue=0)
c3 = Checkbutton(root,text="听音乐",variable=musicHobby,onvalue=1,offvalue=0)
#添加复选框组件
c1.pack(side="left");
```

```
c2.pack(side="left");
c3.pack(side="left")
def confirm():
    if videoHobby.get()==1:
        print("看视频，都是正常人有的爱好！你喜欢看什么类型？")
    if musicHobby.get()==1:
        print("听音乐，不错！你喜欢听摇滚吗?")
    if codeHobby.get()==1:
        print("抓获野生程序猿一只，赶紧送给他尚学堂的视频充饥")
Button(root,text="确定",command=confirm).pack(side="left")
root.mainloop()
```

执行结果如图 4-8 所示：

图 4-8 示例 4-8 运行效果图

控制台输出结果如图 4-9 所示：

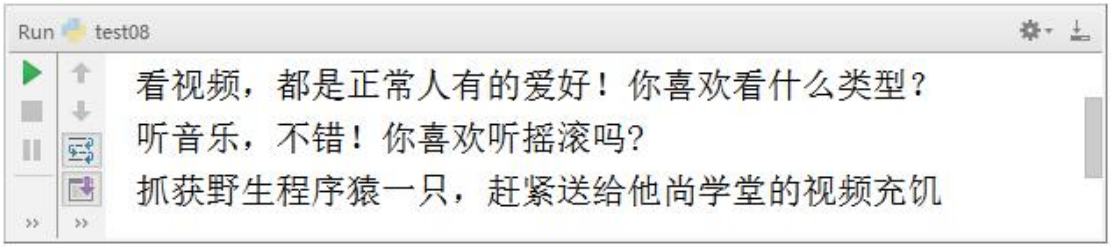


图 4-9 示例 4-8 运行效果图

4.3.7 Canvas 画布

Python Tkinter 画布（Canvas）组件和 html5 中的画布一样，都是用来绘图的。您可以将图形，文本，小部件或框架放置在画布上。在绘图控件 Canvas 中，常用的属性控制参数如表 4-6 所示：

表 4-6 Canvas 画布常用属性

属性名	功能
bd	边框宽度，单位像素，默认为 2 像素。
bg	背景色
confine	如果为 true（默认），画布不能滚动到可滑动的区域外。
cursor	光标的形状设定，如 arrow，circle，cross，plus 等
height	高度
highlightcolor	要高亮的颜色
relief	边框样式,可选值为 FLAT、SUNKEN、RAISED、GROOVE、RIDGE。 默认为 FLAT。
scrollregion	一个元组 tuple（w，n，e，s），定义了画布可滚动的最大区域，w 为左边，n 为头部，e 为右边，s 为底部。
width	画布在 X 坐标轴上的大小。
xscrollincrement	用于滚动请求水平滚动的数量值。
xscrollcommand	水平滚动条，如果画布是可滚动的，则该属性是水平滚动条的 .set() 方法。
yscrollincrement	类似 xscrollincrement，但是垂直方向。
yscrollcommand	垂直滚动条，如果画布是可滚动的，则该属性是垂直滚动条的 .set() 方法。

在绘图控件 Canvas 中，常用的绘图方法如表 4-7 所示：

表 4-7 Canvas 画布常用的方法

方法名	功能
create_arc	绘制圆弧
create_image	绘制图片
create_line	绘制直线
create_oval	绘制椭圆
create_polygon	绘制多边形
create_rectangle	绘制矩形
create_text	绘制文字
create_window	绘制窗口
delete	删除绘制的窗口

【示例 4-9】Canvas 画布的基础用法

```
from tkinter import *
import random
root = Tk();
root.title('画布的基本使用')
root.geometry("200x100+400+400")
```

```
#创建画布，设置其背景色为绿色
canvas = Canvas(root,width=200,height=100,bg="white")
#添加画布
canvas.pack()
#前两个参数描述的是直线的起始位置是水平方向的第 0 个像素和竖直方向的第 50 像素。
fill 设置填充颜色
canvas.create_line(0, 50, 200, 50, fill='yellow')
canvas.create_line(100, 0, 100, 100, fill='red', dash=(4, 4))  #dash 是设置虚线
canvas.create_rectangle(50, 25, 150, 75, fill='blue')  #左上角右下角的坐标
#画一个椭圆.坐标为椭圆的边界矩形左上角和底部右下角
canvas.create_oval(50, 25, 150, 75, fill='green')
root.mainloop()
```

执行结果如图 4-10 所示：

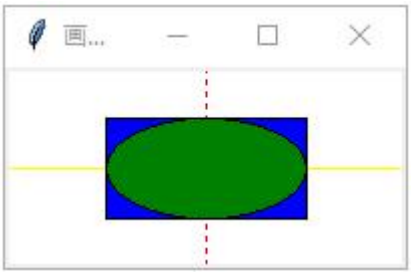


图 4-10 示例 4-9 运行效果图

4.4 布局

Tkinter 有三种布局管理方式：pack、grid 和 place。这三种布局在窗口中不可以混用，也就是说同时只能使用一种布局。Tkinter 中布局的主要工作是将控件放置在窗口上，并根据具体的布局调整控件的位置和控件的尺寸。

4.4.1grid 布局

grid 表格布局，采用表格结构组织组件。子组件的位置由行和列的单元格来确定，并且可以跨行和跨列，从而实现复杂的布局。grid 方法需要制定 row 和 column 两个关键字参数，其中 row 表示当前的行（从 0 开始），column 表示当前的列（从 0 开始），常用属性如表 4-8 所示：

表 4-8 grid 布局的常见属性

选项	说明	取值范围

column	单元格的列号	从 0 开始的正整数
columnspan	跨列，跨越的列数	正整数
row	单元格的行号	从 0 开始的正整数
rowspan	跨行，跨越的行数	正整数
ipadx, ipady	设置子组件之间的间隔，x 方向或者 y 方向，默认单位为像素	非负浮点数，默认 0.0
padx, pady	与之并列的组件之间的间隔，x 方向或者 y 方向，默认单位是像素	非负浮点数，默认 0.0
sticky	组件紧贴所在单元格的某一角，对应于东南西北中以及 4 个角	“n”，“s”，“w”，“e”，“nw”，“sw”，“se”，“ne”，“center”（默认）

【示例 4-10】grid 布局用法实现登录界面设计

```
from tkinter import *
root = Tk();
root.geometry("200x100+400+400")
root.title("grid 布局实现登录界面")
#创建组件
Label(root,text="用户名").grid(row=0,column=0)
Entry(root).grid(row=0,column=1,columnspan=2)

Label(root,text="密码").grid(row=1,column=0)
Entry(root,show="*").grid(row=1,column=1,columnspan=2)

Button(root,text="登录").grid(row=2,column=1,sticky=E)
Button(root,text="取消").grid(row=2,column=2,sticky=W)

root.mainloop()
```

执行结果如图 4-11 所示：

图 4-11 示例 4-10 运行效果图

【示例 4-11】根据实际简易计算器的按键分布，设计一个如图 4-12 所示相仿的计算器界面

图 4-12 计算器界面

```
from tkinter import *
#创建窗口对象
root = Tk();
#设置窗口的大小
root.geometry("150x220")
#设置标题
root.title("计算器")
#创建所有组件名称
btnText = (("MC","M+","M-","MR"),
                ("C","±","÷","×"),
                (7,8,9,"-"),
                (4,5,6,"+"),
                (1,2,3,"="),
                (0,"."))
#创建文本框
Entry(root).grid(row=0,column=0,columnspan=4,pady="10")
#遍历组件名称，创建组件并添加
for rindex,r in enumerate(btnText):
    for cindex,c in enumerate(r):
        if c == "=":
            Button(root,text=c,width=2).grid(row=rindex+1,\
              column=cindex,rowspan=2,sticky=NSEW)
        elif c == 0:
            Button(root,text=c,width=2).grid(row=rindex+1,\
```

```
                        column=cindex,columnspan=2,sticky=EW)
            elif c == ".":
                Button(root, text=c, width=2).grid(row=rindex + 1,\
                                                   column=cindex+1, sticky=EW)
            else:
                Button(root,text=c,width=2).grid(row=rindex+1,\
                                                 column=cindex,sticky=EW)
root.mainloop()
```

执行结果如图 4-13 所示：

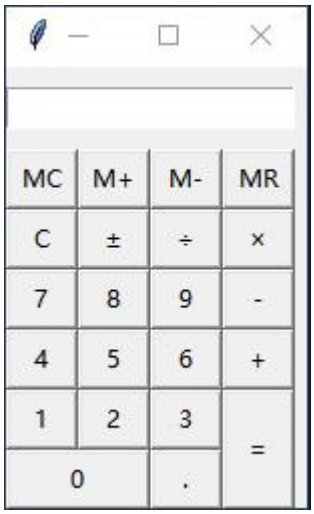
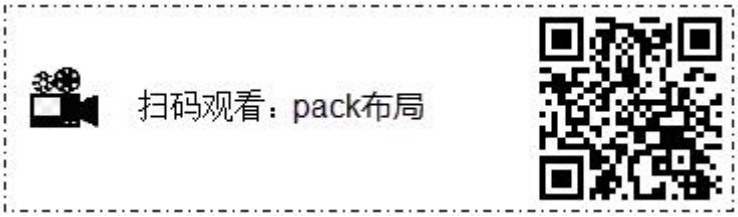


图 4-13 示例 4-11 运行效果图

4.4.2 pack 布局

pack 按照组件的创建顺序将子组件添加到父组件中，按照垂直或者水平的方向自然排布。如果不指定任何选项，默认在父组件中自顶向下垂直添加组件。



pack 是代码量最少，最简单的一种，可以用于快速生成界面。

表 4-9 pack 布局的常见属性

名称	描述	取值范围
expand	当值为“yes”时，side 选项无效。组件显示在父配件中心位置；若 fill 选项为”both”,则填充父组件的剩余空间	“yes”，自然数,”no”，0（默认值”no”或 0）
fill	填充 x(y)方向上的空间，当属性 side=”top”	“x”，“y”，“both”,“none”

	或”bottom”时，填充 x 方向；当属性 side=”left”或”right”时，填充”y”方向；当 expand 选项为”yes”时，填充父组件的剩余空间	（默认值为 none）
ipadx, ipady	设置子组件之间的间隔，x 方向或者 y 方向，默认单位为像素	非负浮点数，默认 0.0
padx, pady	与之并列的组件之间的间隔，x 方向或者 y 方向，默认单位是像素	非负浮点数，默认 0.0
side	定义停靠在父组件的哪一边上	“top”，“bottom”，“left”，“right”（默认为”top”）
before	将本组件于所选组建对象之前 pack，类似于先创建本组件再创建选定组件	已经 pack 后的组件对象
after	将本组件于所选组建对象之后 pack，类似于先创建选定组件再本组件	已经 pack 后的组件对象
in_	将本组件作为所选组建对象的子组件，类似于指定本组件的 master 为选定组件	已经 pack 后的组件对象
anchor	对齐方式，左对齐”w”，右对齐”e”，顶对齐”n”，底对齐”s”	“n”，“s”，“w”，“e”，“nw”，“sw”，“se”， “ne”，“center”（默认）

【示例 4-12】pack 布局用法，制作钢琴按键布局

```
from tkinter import    *
root = Tk();
root.title('pack 布局')
root.geometry("760x220")
#Frame 是一个矩形区域，就是用来防止其他子组件
f1 = Frame(root)
f1.pack()
f2 = Frame(root);
f2.pack()

btnText = ("流行风","中国风","日本风","重金属","轻音乐")
#循环创建按钮并添加组件
for txt in btnText:
    Button(f1,text=txt).pack(side="left",padx="10")
for i in range(1,20):
```

```
    Button(f2,width=5,height=10,bg="black" if i%2==0 else "white").pack(side="left")
root.mainloop()
```

执行结果如图 4-14 所示：

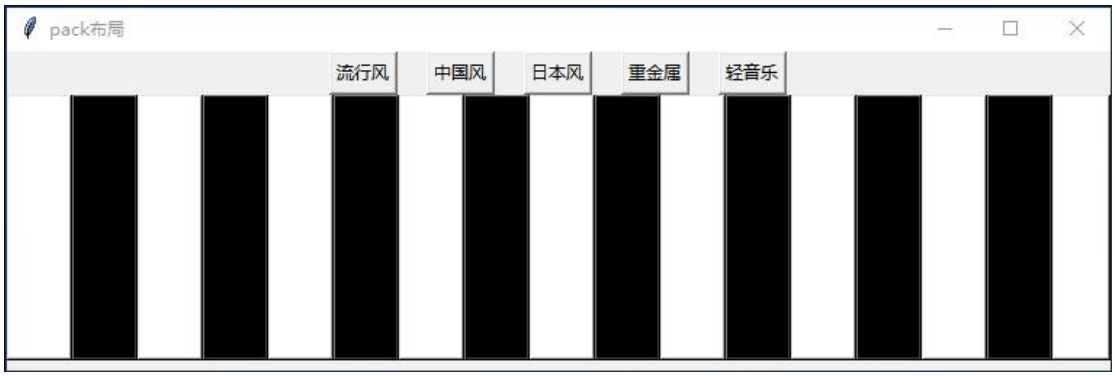


图 4-14 示例 4-12 运行效果图

4.4.3place 布局

place 布局管理器可以通过坐标精确控制组件的位置，适用于一些布局更加灵活的场景。

表 4-10 place 布局的常见属性

选项	说明	取值范围
x,y	组件左上角的绝对坐标（相对于窗口）	非负整数 x 和 y 选项用于设置偏移(像素)，如果同时设置 relx(rely)和 x(y)，那么 place 将优先计算 relx 和 rely，然后再实现 x 和 y 指定的偏移值
relx rely	组件左上角的坐标 （相对于父容器）	relx 是相对父组件的位置。0 是最左边，0.5 是正中间，1 是最右边； rely 是相对父组件的位置。0 是最上边，0.5 是正中间，1 是最下边；
width, height	组件的宽度和高度	非负整数
选项	**说明**	**取值范围**
relwidth, relheight	组件的宽度和高度 （相对于父容器）	与 relx、rely 取值，但是 相对于父组件的尺寸
anchor	对齐方式，左对齐”w”，右对齐”e”，顶对齐”n”，底对齐”s”	“n”，“s”，“w”，“e”，“nw”，“sw”，“se”，“ne”，“center”(默认)

【示例 4-13】place 布局的基本使用

```
'''
通过 place 方法的参数指定控件的位置(x,y)和尺寸(width 和 height)
'''
from tkinter import *
import random
root=Tk()
root.title('place 布局')
root['background']='white'
root.geometry('200x200+30+30')
languages=['python','javascript','c++','java','c']
for i in range(len(languages)):
    #随机产生背景的三原色
    ct=[random.randrange(256) for x in range(3)]
    #得到背景色的十六进制形式
    ct_hex='%02x%02x%02x'%tuple(ct)
    bg_color='#'+"".join(ct_hex)
    label=Label(root,text=languages[i],fg='white',bg=bg_color)
    label.place(x=25,y=30+i*30,width=120,height=25)
mainloop()
```

执行结果如图 4-15 所示：

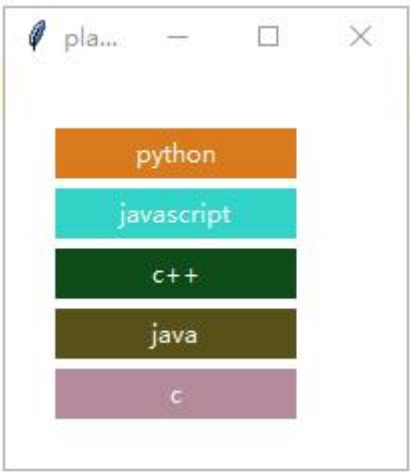


图 4-15 示例 4-13 运行效果图

4.5 事件处理

在使用库 Tkinter 实现 GUI 开发的过程中，属性和

方法是 Tkinter 控件的两个重要元素。但是除此之外，还需要借助于事件来实现 Tkinter 控件的动态功能效果。Tkinter 提供了用以处理相关事件的机制。处理函数可被绑定给各个控件的各种事件。

4.5.1 事件基础

在计算机系统中有很多事件，例如鼠标事件，键盘事件和窗口事件等。鼠标事件主要指鼠标按键的按下、释放、鼠标滚轮的滚动，鼠标指针移入、移出组件等所触发的事件。键盘事件主要指键的按下、释放等所触发的事件。窗口事件是指改变窗口的大小、控件状态等变化所触发的事件外，在创建右键弹出菜单时还需处理右击事件。

在 Python 程序的 Tkinter 库中，鼠标事件、键盘事件和窗口事件可以采用事件绑定的方法来处理消息。为了实现控件绑定功能，可以使用控件中的 bind()实现，或者使用方法 bind_class()实现类绑定，分别调用函数或者类来响应事件。方法 bind_all()也可以绑定事件，方法 bind_all()能够将所有组件事件绑定到事件响应函数上。语法格式如下所示：

```
bind(sequence,func,add)
bind_class(className,sequence,func,add)
bind_all(sequence,func,add)
```

各个参数的具体说明如表 4-11 所示：

表 4-11 绑定事件函数参数说明

参数名	说明
func	所绑定的事件处理函数
参数名	**说明**
add	可选参数，为空字符或者“+”
className	所绑定的类
sequence	表示所绑定的事件，必须是以尖括号“<>”包围的字符串

在库 Tkinter 中，常用的鼠标事件如表 4-12 所示：

表 4-12 鼠标事件

代码	说明
<Button-1> <ButtonPress-1> <1>	鼠标左键按下。 2 表示中键，3 表示右键；
<ButtonRelease-1>	鼠标左键释放
<B1-Motion>	按住鼠标左键移动
<Double-Button-1>	双击左键
<Enter>	鼠标指针进入某一组件区域
<Leave>	鼠标指针离开某一组件区域

<MouseWheel>	滚动滚轮；

在库 Tkinter 中，常用的键盘事件如表 4-13 所示：

表 4-13 键盘事件

代码	说明
<KeyPress-a>	按下 a 键，a 可用其他键替代
<KeyRelease-a>	释放 a 键。
<KeyPress-A>	按下 A 键（大写的 A）
<Alt-KeyPress-a>	同时按下 alt 和 a；alt 可用 ctrl 和 shift 替代
<Double-KeyPress-a>	快速按两下 a
<Control-V>	CTRL 和 V 键被同时按下，V 可以换成其它键位

在库 Tkinter 中，常用的窗口事件如表 4-14 所示：

表 4-14 窗口事件

代码	说明
Activate	当组件由不可用转为可用时触发
Configure	当组件大小改变时触发
Deactivate	当组件由可用转为不可用时触发
Destroy	当组件被销毁时触发
Expose	当组件从被遮挡状态中暴露出来时触发
代码	**说明**
FocusIn	当组件获得焦点时触发
FocusOut	当组件失去焦点时触发
Map	当组件由隐藏状态变为显示状态时触发
Property	当窗体的属性被删除或改变时触发
Unmap	当组件由显示状态变为隐藏状态时触发
Visbility	当组件变为可视状态时触发

当把窗口中的事件绑定到函数后，如果触发该事件，将会调用所绑定的函数进行处理。触发事件后，系统将向该函数传递一个 event 对象的参数。绑定的响应事件函数格式如下：

```
def  function(event):
     <语句>
```

event 对象具有的属性信息如表 4-15 所示：

表 4-15 event 对象的属性信息

名称	说明

char	按键字符，仅对键盘事件有效
keycode	按键编码，仅对键盘事件有效
num	鼠标按键，仅对鼠标事件有效
type	所触发的事件类型
keysym	按键名称，仅对键盘事件有效 比如按下空格键： 键的 char： 键的 keycode：32 键的 keysym：space 比如按下 a 键： 键的 char：a||键的 keycode：65||键的 keysym：a
widget	引起事件的组件
width,height	组件改变后的大小，仅 Configure 有效
x，y	鼠标当前位置，相对于父容器
x_root,y_root	鼠标当前位置，相对于整个屏幕

【示例 4-14】鼠标事件的基本使用

```
#测试鼠标事件
from tkinter import   *
root = Tk();
root.geometry("530x300")
#创建画布
c1 = Canvas(root,width=200,height=200,bg="green")
c1.pack()
#定义鼠标事件触发函数
def mouseTest(event):
    print("鼠标左键单击位置(相对于父容器)：{0},{1}".format(event.x,event.y))
    print("鼠标左键单击位置(相对于屏幕)：{0},{1}".format(event.x_root,event.y_root))
    print("事件绑定的组件：{0}".format(event.widget))
#给画布绑定鼠标事件函数
c1.bind("<Button-1>",mouseTest)
root.mainloop()
```

执行结果如图 4-16 所示：

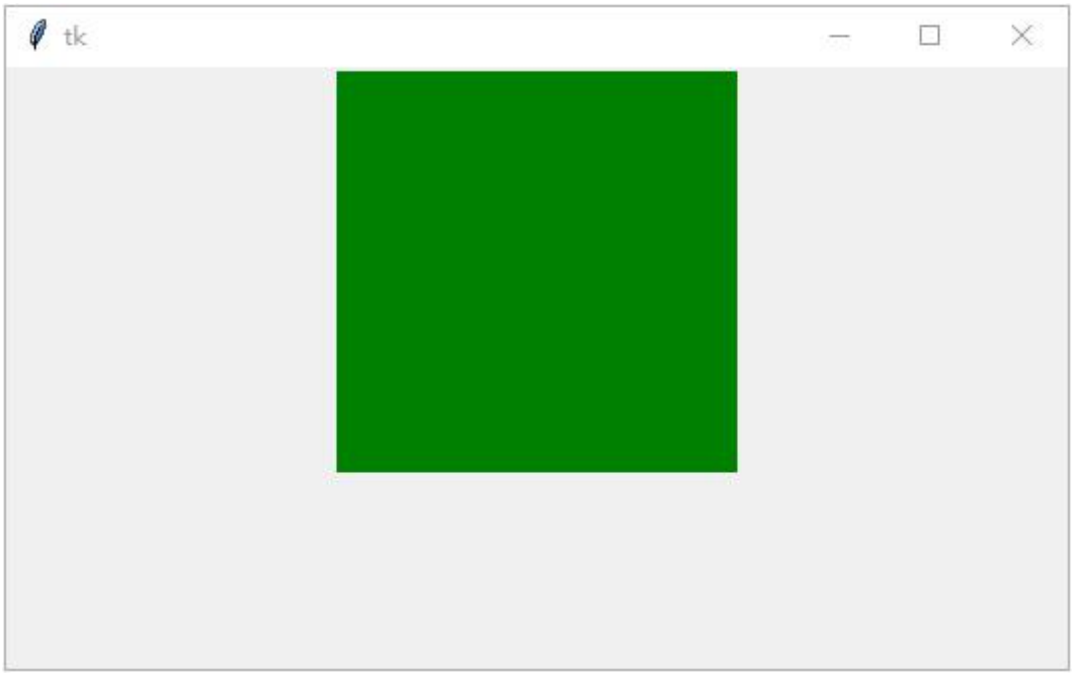


图 4-16 示例 4-14 运行效果图

点击鼠标控制台输出结果如图 4-17 所示：

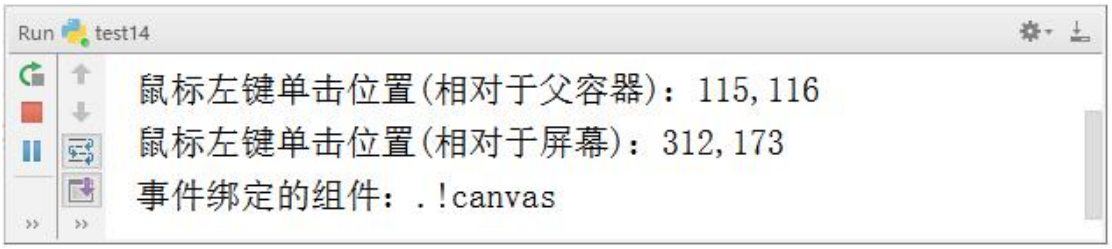


图 4-17 示例 4-14 运行效果图

【示例 4-15】键盘事件的基本使用

```
#测试键盘事件
from tkinter import    *
root = Tk();
root.geometry("530x300")
c1 = Canvas(root,width=200,height=200,bg="green")
c1.pack()
#定义键盘事件触发函数
#按下键盘的键
def keyboardTest(event):
      print("键的 keycode:{0},键的 char:{1},键的 keysym:{2}"
            .format(event.keycode,event.char,event.keysym))
#按下键盘的 a 触发
```

```
def press_a_test(event):
    print("press a")
#释放键盘的 a 触发
def release_a_test(event):
    print("release a")
#给窗口绑定键盘事件函数
root.bind("<KeyPress>",keyboardTest)
root.bind("<KeyPress-a>",press_a_test)
root.bind("<KeyRelease-a>",release_a_test)

root.mainloop()
```

执行结果如图 4-18 所示：

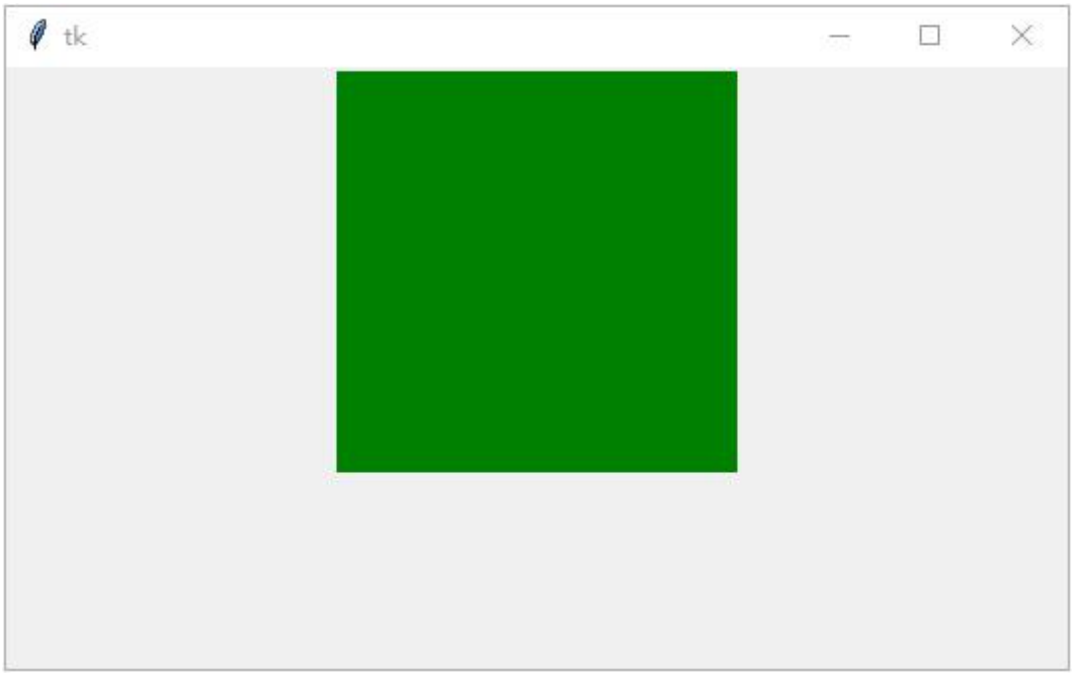


图 4-18 示例 4-15 运行效果图

点击鼠标控制台输出结果如图 4-19 所示：

图 4-19 示例 4-15 运行效果图

4.5.2 lambda 表达式

lambda 表达式定义的是一个匿名函数，只适合简单输入参数，简单计算返回结果，不适合功能复杂情况。lambda 定义的匿名函数也有输入、也有输出，只是没有名字。语法格式如下：

```
lambda 参数值列表：表达式
```

lambda 表达式的参数值列表如表 4-16 所示：

表 4-16 lambda 表达式的参数值列表

lambda 格式	说明
lambda x, y: x*y	函数输入是 x 和 y，输出是它们的积 x*y
lambda:None	函数没有输入参数，输出是 None
lambda:aaa(3,4)	函数没有输入参数，输出是 aaa(3,4)的结果
lambda *args: sum(args)	输入是任意个数的参数，输出是它们的和
lambda **kwargs: 1	输入是任意键值对参数，输出是 1

【示例 4-16】使用 lambda 帮助 command 属性绑定传参

```
from tkinter import *
root=Tk()
root.title('多种方式绑定事件')
root.geometry('300x300')
#创建画布，设置其背景色为白色
canvas = Canvas(root,width=200,height=200,bg="white")
#添加画布
canvas.pack()
#创建事件函数
def drawLine():
    canvas.create_line(180,100,180,0,fill='green')
def drawRec():
    print('矩形')
    canvas.create_rectangle(0,50,100,100,fill='red')
```

```
def drawoval(x1,y1,x2,y2):
    canvas.create_oval(x1,y1,x2,y2,fill='blue')
#创建 Button 组件
b1=Button(root,text='直线',command=drawLine)
b2=Button(root,text='矩形',command=lambda :drawRec())
b3=Button(root,text='圆',command=lambda :drawoval(100, 100, 200, 200))
b1.pack(side='left')
b2.pack(side='left')
b3.pack(side='left')
mainloop()
```

左击三个按钮执行结果如图 4-20 所示：

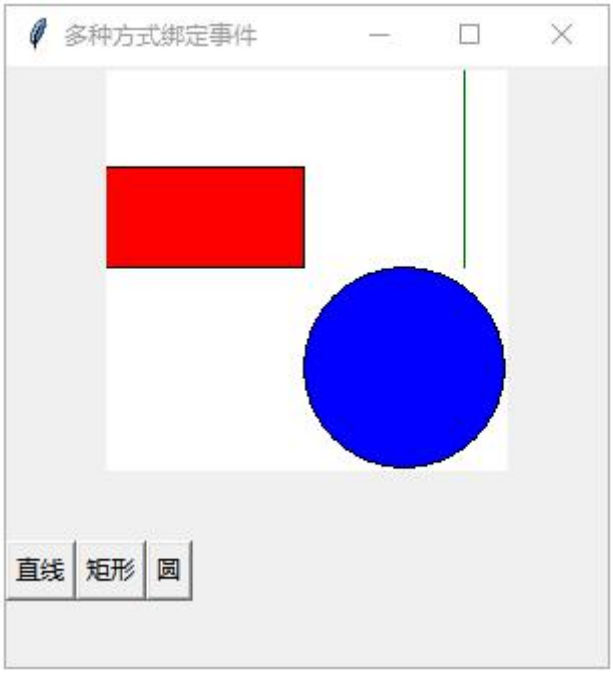


图 4-20 示例 4-16 运行效果图

13.5.3 bind_class 函数

调用对象的 bind_class 函数，将该组件类所有的组件绑定事件。

【示例 4-17】bind_class 函数绑定事件

```
from tkinter import *
root=Tk()
```

```
root.title('多种方式绑定事件')
root.geometry('300x300')
#创建事件函数
def buttonTest3(event):
    print("右键单击事件")

#创建 Button 组件
b1=Button(root,text='but1')
b2=Button(root,text='but2')
b3=Button(root,text='but3')
#给三个按钮绑定右击事件
b1.bind_class('Button','<Button-3>',buttonTest3)
b1.pack(side='left')
b2.pack(side='left')
b3.pack(side='left')
mainloop()
```

右击三个按钮执行结果如图 4-21 所示：

图 4-21 示例 4-17 运行效果图

4.6 其他组件

4.6.1 OptionMenu 选择项

OptionMenu(选择项)用来做多选一，选中的项会在顶部显示。

【示例 4-18】OptionMenu(选择项)的基本用法

```
#optionmenu 的使用测试
from tkinter import    *
root = Tk();
```

```
root.geometry("200x100")
v = StringVar(root);
v.set("百战程序员")
om = OptionMenu(root,v,"尚学堂","百战程序员","卓越班[保底 18 万]")
om["width"]=10
om.pack(pady=20)
def test1():
    v.set("尚学堂")          #直接修改了 optionmenu 中选中的值
    print("最喜爱的机构:",v.get())
Button(root,text="确定",command=test1).pack()
root.mainloop()
```

执行结果如图 4-22 所示：

图 4-22 示例 4-18 运行效果图

4.6.2 Scale 移动滑块

Scale(移动滑块)用于在指定的数值区间，通过滑块的移动来选择值。

【示例 4-19】Scale(移动滑块)的基本用法

```
#Scale(移动滑块)
from tkinter import    *
root = Tk()
root.geometry("400x150")
def test1(value):
    print("滑块的值:",value)
    newFont = ("宋体",value)
    a.config(font=newFont)
s1 = Scale(root,from_=10,to=50,length=200,orient=HORIZONTAL,command=test1)
s1.pack()
a = Label(root,text="百战程序员",width=10,height=1,bg="black",fg="white")
a.pack()
```

```
root.mainloop()
```

执行结果如图 4-23 所示：

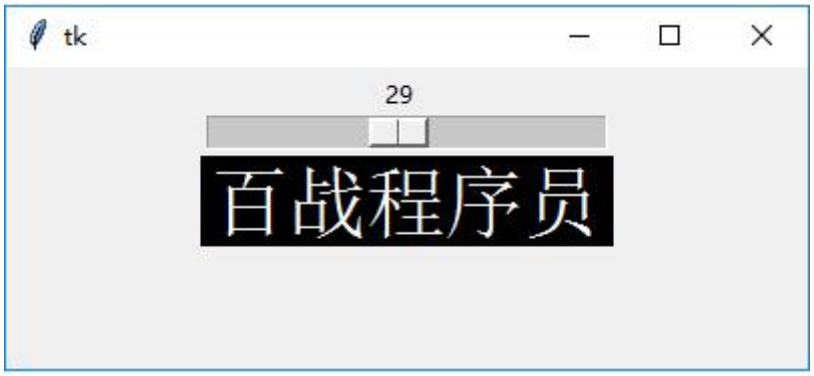


图 4-23 示例 4-19 运行效果图

4.6.3 颜色选择框

颜色选择框设置背景色、前景色、画笔颜色、字体颜色等等。

【示例 4-20】Scale(移动滑块)的基本用法

```
#askcolor 颜色选择框的测试，改变背景色
from tkinter import    *
from tkinter.colorchooser import *
root = Tk()
root.geometry("400x150")
#创建颜色选择框
s1 = askcolor(color="green", title="选择背景色")
root.config(bg=s1[1])
root.mainloop()
```

执行结果如图 4-24 所示：

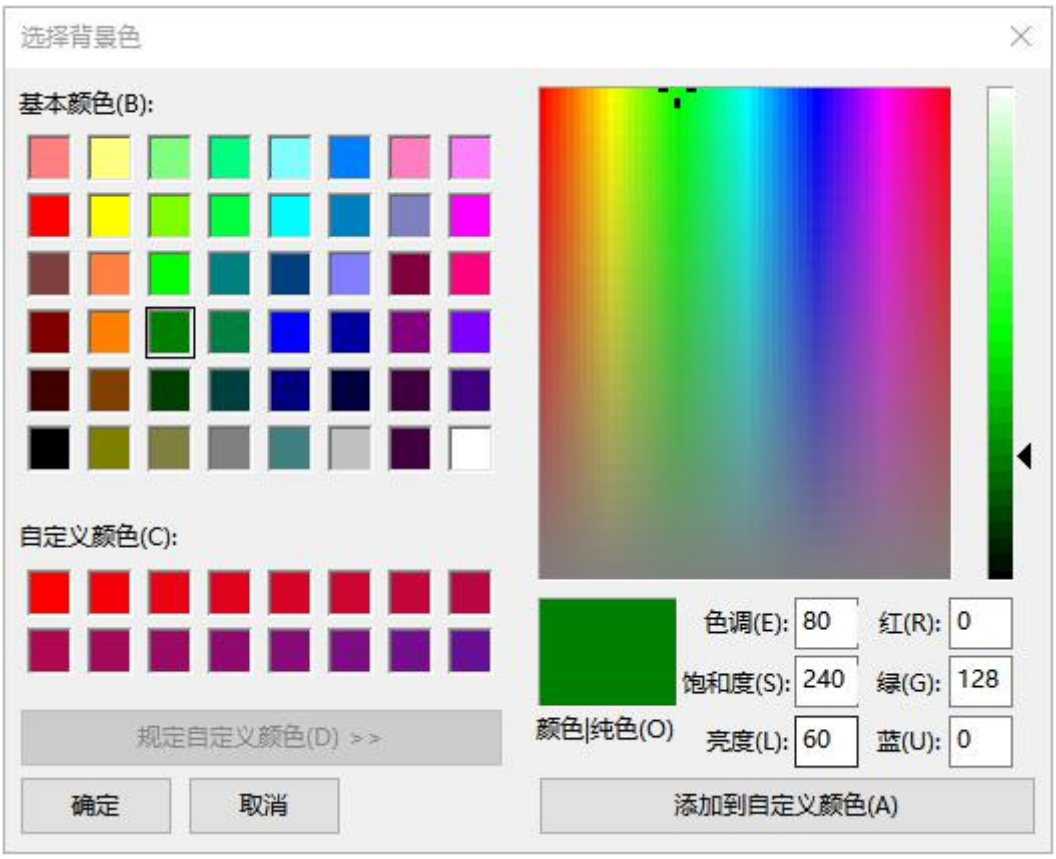


图 4-24 示例 4-20 运行效果图

4.6.4 文件对话框

文件对话框来实现可视化的操作目录、操作文件。最后，将文件、目录的信息传入到程序中。文件对话框包含如下一些常用函数如表 4-17 所示：

表 4-17 文件对话框常用函数

函数名	对话框	说明
askopenfilename(**options)	文件对话框	返回打开的文件名
askopenfilenames(**options)		返回打开的多个文件名列表
askopenfile(**options)		返回打开的文件对象
askopenfiles(**options)		返回打开的文件对象的列表
askdirectory(**options)	目录对话框	返回目录名
asksaveasfile(**options)	保存对话框	返回保存的文件对象
asksaveasfilename(**options)		返回保存的文件名

命名参数 options 的常见值如表 4-18 所示：

表 4-18 命名参数 options 的常见值

参数名	说明
defaultextension	默认后缀：.xxx 用户没有输入则自动添加
filetypes=[(label1,pattern1),(labe2,pattern2)]	文件显示过滤器

initaldir	初始目录
initalfile	初始文件
parent	父窗口，默认根窗口
title	窗口标题

【示例 4-21】文件对话框基本用法

```
from tkinter import    *
from tkinter.filedialog import *
root = Tk()
root.geometry("400x100")
def test1():
    f = askopenfilename(title="上传文件",
                        initialdir="e:/file",filetypes=[("图片文件",".jpg")])
    show["text"]=f
Button(root,text="选择编辑的图片文件",command=test1).pack()
show = Label(root,width=40,height=3,bg="green")
show.pack()
root.mainloop()
```

执行结果如图 4-25 所示：

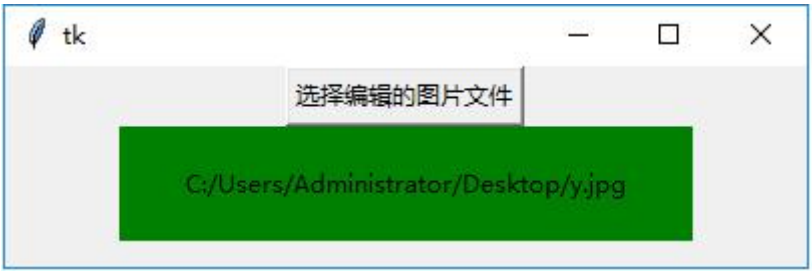
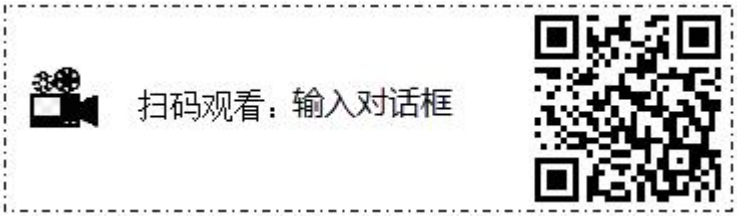


图 4-25 示例 4-21 运行效果图

4.6.5 输入对话框



simpledialog(简单对话框)常用函数如表 4-19 所示：

表 4-19 simpledialog(简单对话框)

函数名	说明
askfloat(title, prompt, **kw)	输入并返回浮点数
askinteger(title, prompt, **kw)	输入并返回整数

askstring(title,prompt,**kw)	输入并返回字符串

其中 title 表示窗口标题；prompt 是提示信息；命名参数 kw 为各种选项：initialvalue（初始值）、minvalue（最小值）、maxvalue（最大值）。

【示例 4-22】输入对话框基本用法

```
#简单对话框
from tkinter import    *
from tkinter.simpledialog import *
root = Tk()
root.geometry("400x100")
show = Label(root,width=40,height=3,bg="green")
show.pack()
a = askinteger(title="输入年龄",prompt="请输入年龄",initialvalue=18,minvalue=1,maxvalue=150)
show["text"]="年龄："+str(a)


root.mainloop()
```

执行结果如图 4-26 所示：

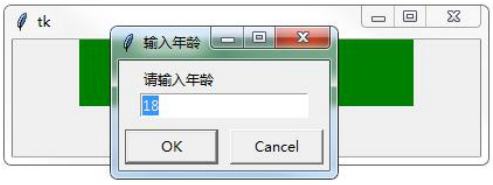


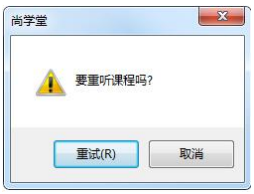
图 4-26 示例 4-22 运行效果图

4.6.6 消息框

messagebox（通用消息框）用于和用户简单的交互，用户点击确定、取消。messagebox 的常见函数如表 4-20 所示：

表 4-20 messagebox 的常见函数

函数名	说明	例子
askokcancel(title,message,**options)	OK/Cancel 对话框	

askquestion(title,message,**options)	Yes/No 问题对话框	
askretrycancel(title,message,**options)	Retry/Cancel 问题对话框	
showerror(title,message,**options)	错误消息对话框	
showinfo(title,message,**options)	消息框	
showwarning(title,message,**options)	警告消息框	

【示例 4-23】输入对话框基本用法

```
#简单对话框
from tkinter import    *
from tkinter.simpledialog import *
root = Tk()
root.geometry("400x100")
show = Label(root,width=40,height=3,bg="green")
show.pack()
a = askinteger(title="输入年龄",prompt="请输入年龄",initialvalue=18,minvalue=1,maxvalue=150)
show["text"]="年龄："+str(a)


root.mainloop()
```

执行结果如图 4-27 所示：

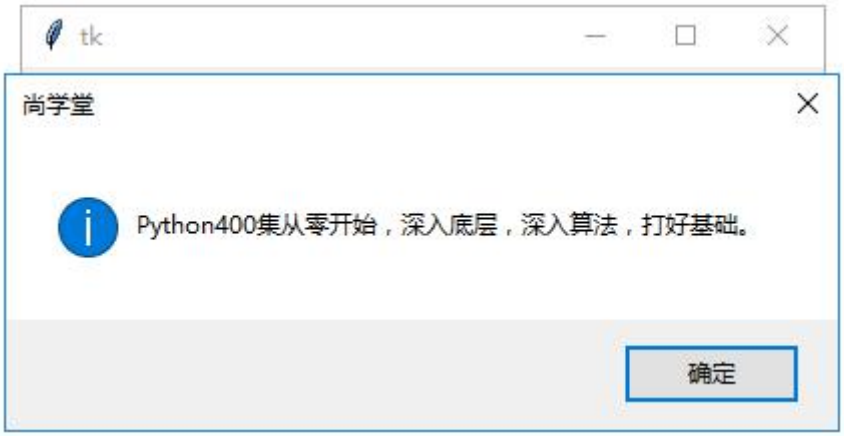


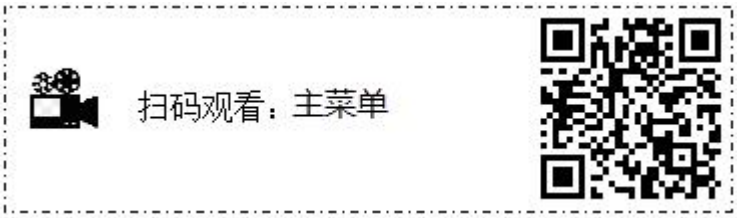
图 4-27 示例 4-23 运行效果图

4.7 菜单和工具栏

在库 Tkinter 的控件中，使用菜单控件的方式与使用其他控件的方式有所不同。在创建菜单控件时，需要使用创建主窗口的方法 config()将菜单添加到窗口中。

4.7.1 主菜单

主菜单一般包含：文件、编辑、帮助等，位于 GUI 窗口的上面。创建主菜单一般有如下 4 步：



（1） 创建主菜单栏对象

```
menubar = tk.Menu(root)
```

（2） 创建菜单，并添加到主菜单栏对象

```
file_menu = tk.Menu(menubar)
menubar.add_cascade(label=”文件”,menu=file_menu)
```

（3） 添加菜单项到菜单

```
file_menu.add_command(label=”打开”)
file_menu.add_command(label=”保存”,accelerator=”^p” command=mySaveFile)
file_menu.add_separator()
file_menu.add_command(label=”退出”)
```

（4） 将主菜单栏添加到根窗口

```
root.config(menu=menubar)
```

【示例 4-24】记事本菜单设计

```
#记事本软件,练习主菜单的设计
```

```
from tkinter import    *
from tkinter.messagebox import *
from tkinter.filedialog import *
root = Tk()
root.geometry("400x400")
#创建主菜单栏
menubar = Menu(root)
#创建子菜单
menuFile = Menu(menubar)
menuEdit = Menu(menubar)
menuHelp = Menu(menubar)

#将子菜单加入到主菜单栏
menubar.add_cascade(label="文件(F)",menu=menuFile)
menubar.add_cascade(label="编辑(E)",menu=menuEdit)
menubar.add_cascade(label="帮助(H)",menu=menuHelp)

menuEdit.add_command(label='剪切')
menuEdit.add_command(label='复制')
menuEdit.add_command(label='粘贴')
filename = ""
def openFile():
    global filename
    with askopenfile(title="打开文件") as f:
        content = f.read()
        w1.insert(INSERT,content)
        filename = f.name
        print(f.name)

def saveFile():
    with open(filename,"w") as f:
        content = w1.get(1.0,END)
        f.write(content)

def exit():
```

```
    root.quit()
#添加菜单项
menuFile.add_command(label="打开",accelerator="^O",command=openFile)
menuFile.add_command(label="保存",command=saveFile)
menuFile.add_separator()  #添加分割线
menuFile.add_command(label="退出",command=exit)
#将主菜单栏加到根窗口
root.config(menu=menubar)
w1 = Text(root,width=50,height=30)
w1.pack()
root.mainloop()
```

执行结果如图 4-28 所示：

图 4-28 示例 4-24 运行效果图

4.7.2 上下文菜单

快捷菜单（上下文菜单）是通过鼠标右键单击组件而弹出的菜单，一般是和这个组件相关的操作，比如：剪切、复制、粘贴、属性等。创建快捷菜单步骤如下所诉：

（1） 创建菜单

menubar = tk.Menu(root)

menubar.add_command(label=”字体”)

（2） 绑定鼠标右键单击事件

def test(event):

menubar.post(event.x_root,event.y_root) #在鼠标右键单击坐标处显示菜单

root.bind(“<Button-3>”,test)

【示例 4-25】为记事本增加快捷菜单

```
from tkinter import    *
from tkinter.colorchooser import *
from tkinter.filedialog import *
root = Tk()
root.geometry("400x400")
def openAskColor():
    s1 = askcolor(color="red", title="选择背景色")
    root.config(bg=s1[1])
#创建快捷菜单
menubar = Menu(root)
menubar.add_command(label="颜色",command=openAskColor)

menuedit = Menu(menubar,tearoff=0)
menuedit.add_command(label="剪切")
menuedit.add_command(label="复制")
menuedit.add_command(label="粘贴")

menubar.add_cascade(label="编辑",menu=menuedit)

def test(event):
    #菜单在鼠标右键单击的坐标处显示
    menubar.post(event.x_root,event.y_root)
#编辑区
w1 = Text(root,width=50,height=30)
w1.pack()
w1.bind("<Button-3>",test)
```

```
root.mainloop()
```

执行结果如图 4-29 所示：

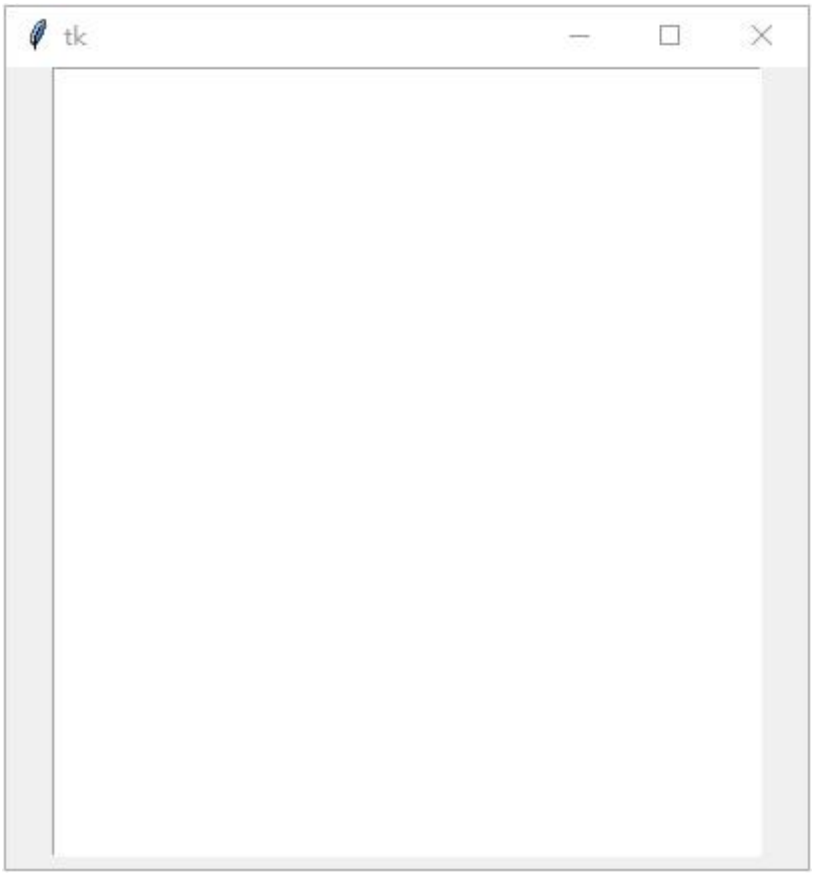


图 4-29 示例 4-25 运行效果图

现在右键点击颜色，运行效果如图 4-30 所示：

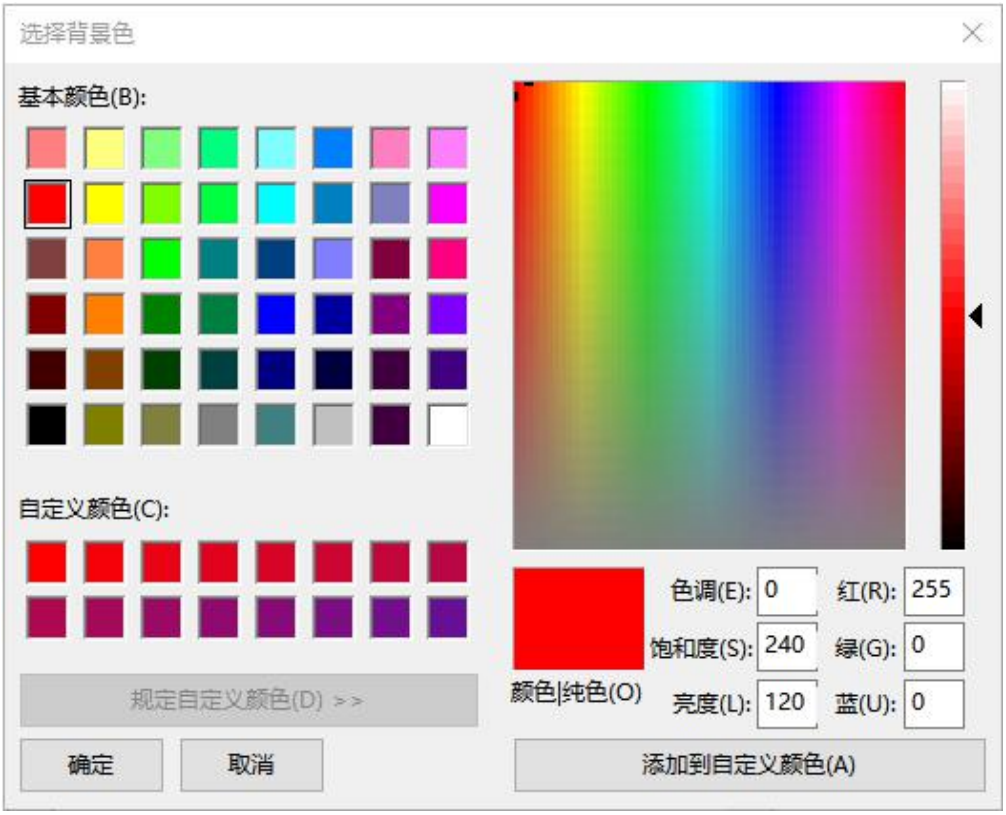


图 4-30 示例 4-25 运行效果图

鼠标移到编辑会显示剪切、复制、粘贴三个按钮如图 4-31 所示：

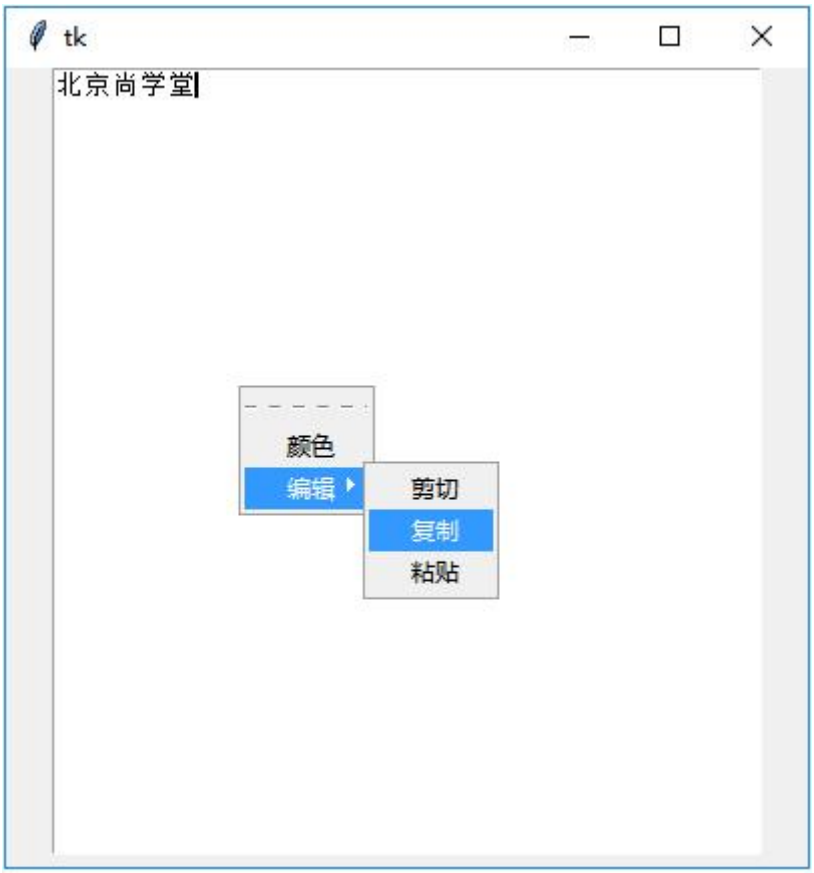


图 4-31 示例 4-25 运行效果图

习题

一、选择题

1. Python 作为功能强大的语言，同样支持 GUI 的开发，下面的库中，____不是 Python 的 GUI 库。

A. PyGTK

B. Cbk

C. wxPython

D. Tkinter

2. 生成主窗口 root=tkinter.Tk()说法正确的是（）（多选）

A. root.title('标题名') 通过 title 方法设置窗口的标题

B. root.resizable(0,0) 框体大小可调性，分别表示 x,y 方向的可变性

C. root.update() 刷新页面；

D. root.exit() 退出；

3. 下面能够实现选择项组件的是（ ）。

A. OptionMenu

B. Scale

C. simpledialog

D. messagebox

4. 下面能实现文本框（单行文本和多行文本）的是（ ）。（多选）

A. Entry

B. Text

C. Listbox

D. Radiobutton

5. 组件的放置和排版的方式（ ）（多选）

A. pack 组件设置位置属性参数

B. grid 组件使用行列的方法放置组件的位置

C. place 组件可以直接使用坐标来放置组件

D. flow 组件顺序摆放组件

二、解答题

1. 什么是 GUI 编程？
2. 常用的 GUI 库有哪些？
3. 什么是 TKinter？
4. 常用的 Tkinter 组件有哪些？
5. 控件如何绑定事件？

三、编码题

1. 用 tkinter 编写一个程序，实现 QQ 登录页面。
2. 在习题 1 的基础上，给按钮绑定事件。点击“登录”按钮，会验证输入的用户名和密码，单击“取消”按钮会关闭登录对话框。
3. 用 tkinter 编写一个程序，使用 grid 布局，实现如图 4-32 普通计算器的效果。

图 4-32 普通计算器

4. 用 tkinter 编写一个程序，使用菜单，实现记事本。

第五章 坦克大战

Pygame 是跨平台 Python 模块，专为电子游戏设计。Pygame 包含的图像和声音建立在 SDL 的基础上，它允许实时电子游戏研发而无须被低级语言（如机器语言和汇编语言）束缚。基于这样一个设想，所有需要的游戏功能和理念（主要是图像方面）都完全简化为游戏逻辑本身，所有的资源结构都可以由高级语言提供。本章将详细讲解在 Python 语言中说明 Pygame 开发游戏的知识。

通过阅读本章，你可以：

- 了解游戏开发流程
- 掌握 Pygame 框架中的常用模块
- 掌握 Pygame 中事件处理
- 掌握 Pygame 中字体样式显示
- 掌握使用 Pygame 开发坦克大战游戏

5.1 Pygame 开发基础

5.1.1 Pygame 框架中的模块

在 Pygame 框架中有很多模块，其中最常用模块的具体说明如表 5-1 所示：

表 5-1 Pygame 框架中的常用模块

模块名	功能说明
pygame.cdrom	访问光驱
pygame.cursors	加载光标
pygame.display	访问显示设备
pygame.draw	绘制形状、线和点
pygame.event	管理事件
pygame.font	使用字体
pygame.image	加载和存储图片
pygame.joystick	使用游戏手柄或者类似的东西
pygame.key	读取键盘按键
pygame.mixer	声音
pygame.mouse	鼠标
pygame.movie	播放视频
pygame.music	播放音频
pygame.overlay	访问高级视频叠加
pygame.rect	管理矩形区域
pygame.sndarray	操作声音数据

模块名	功能说明
pygame.sprite	操作移动图像
pygame.surface	管理图像和屏幕
pygame.surfarray	管理点阵图像数据
pygame.time	管理时间和帧信息
pygame.transform	缩放和移动图像

【示例 5-1】开发第一个 Pygame 程序

```
import pygame
#初始函数，使用 pygame 的第一步；
pygame.init()
#生成主屏幕 screen
screen=pygame.display.set_mode((600,500),0,32)
#设置标题
pygame.display.set_caption('Hello Pygame')
while True:
    #刷新屏幕
    pygame.display.update()
```

执行结果如图 5-1 所示：

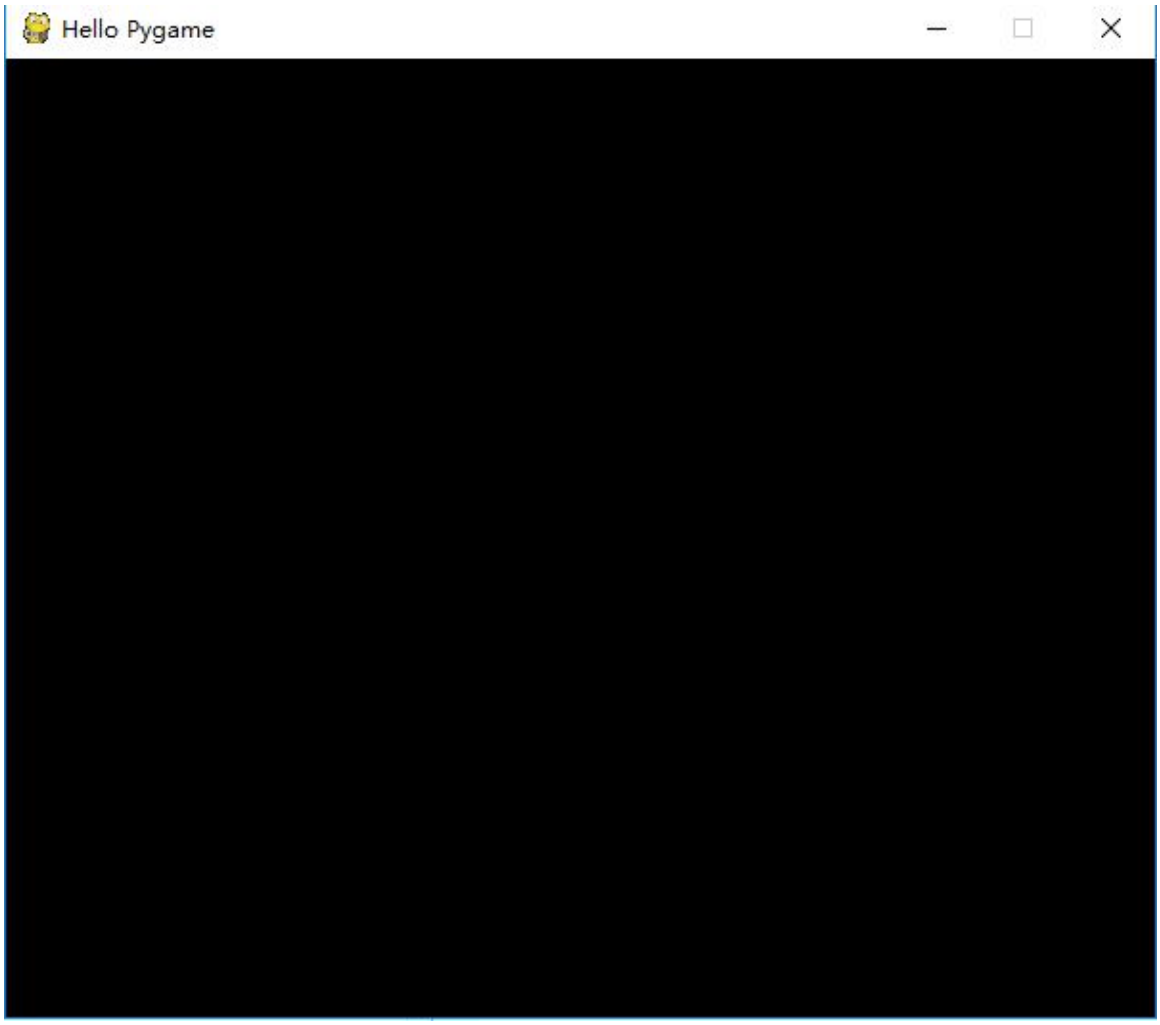


图 5-1 示例 5-1 运行效果图

对上述示例代码的具体说明如下所示：

（1）set_mode 函数：会返回一个 Surface 对象，代表了在桌面上出现的那个窗口。在 3 个参数中，第 1 个参数为元组，表示屏幕的大小；第 2 个标志位，具体含义如表 5-2 所示，如果不用什么特性，就指定 0；第 3 个为色深。

表 5-2 各个标志位的具体含义

标志位	含义
FULLSCREEN	创建一个全屏窗口
DOUBLEBUF	创建一个“双缓冲”窗口，建议和 HWSURFACE 和 OPENGL 同时使用
NOFRAME	创建一个没有边框的窗口
RESIZEBLE	创建一个可以改变大小的窗口
OPENGL	创建一个 OPENGL 渲染的窗口
HWSURFACE	创建一个硬件加速的窗口，必须和 FULLSCREEN 同时使用

（2）游戏的主循环是一个无限循环，直到用户退出。在这个主循环里面做的事情就是

不停的刷新新画面。

注意：

- 如果未安装 pygame 模块的，打开控制台执行 pip install pygame 命令进行安装。

5.1.2 事件操作

事件是一个操作，通常来说，Pygame 会接收用户的各种操作（比如按键盘，移动鼠标等）。这些操作会产生对应的事件，例如按键盘事件，移动鼠标事件。事件在软件开发中非常重要，Pygame 把一系列的事件存放在一个队列里，并逐个进行处理。

1. 事件检索

使用函数 pygame.event.get()获取所有的事件，表 5-3 列出了 Pygame 中常用的事件。

表 5-3 Pygame 中常用的事件

事件	参数	产生途径
QUIT	none	用户按下关闭按钮
ACTIVEEVENT	gain, state	激活或者隐藏 Pygame
KEYDOWN	unicode, key, mod	按下键
KEYUP	key, mod	放开键
MOUSEMOTION	pos, rel, buttons	鼠标移动
MOUSEBUTTONUP	pos, button	放开鼠标键
MOUSEBUTTONDOWN	pos, button	按下鼠标键
JOYBUTTONUP	joy, button	游戏手柄放开
JOYBUTTONDOWN	joy, button	游戏手柄按下
VIDEORESIZE	size, w, h	Pygame 窗口缩放
VIDEOEXPOSE	none	Pygame 窗口部分公开（expose）
USEREVENT	code	触发一个用户事件

2. 处理鼠标事件

在 Pygame 框架中，MOUSEMOTION 事件会在鼠标动作的时候发生，它有如下所示 3 个参数。

- buttons：一个含有 3 个数字的元组，3 个值分别代表左键、中键和右键，1 就表示按下。
- pos：位置
- rel：代表现在距离上次产生鼠标事件时的距离

和 MOUSEMOTION 类似，常用的鼠标事件还有 MOUSEBUTTONUP 和 MOUSEBUTTONDOWN 两个。在很多时候，开发者只需要知道鼠标按下就可以不用上面那

个比较强大的事件了。这两个事件的参数如下：

- button：这个值代表操作哪个按键
- pos：位置

3. 处理键盘事件

在 Pygame 框架中，键盘和游戏手柄的事件比较类似，处理键盘的事件为 KEYDOWN 和 KEYUP。KEYDOWN 和 KEYUP 事件的参数描述如下：

- key：按下或者放开的键值，是一个数字，因为很少有人可以记住，所以在 Pygame 中可以使用 K_xxx 来表示，比如字母 a 就是 K_a，还有 K_SPACE 和 K_RETURN 等。
- mod：包含了组合键信息，如果 mod&KMOD_CTRL 是真，表示用户同时按下了 Ctrl 键，类似的还有 KMODE_SHIFT 和 KMODE_ALT。
- unicode：代表了按下键对应的 Unicode 值。

【示例 5-2】键盘事件

```
import pygame
pygame.init()
screen=pygame.display.set_mode((600,500),0,32)
#加载图片
image=pygame.image.load('img/logo.gif')
#设置 x，y 初始值作为初始位置
x,y=0,0
#设置横向和纵向两个方向的移动距离
while True:
    for event in pygame.event.get():
        #如果是退出
        if event.type==pygame.QUIT:
            exit()
        #如果是按下
        if event.type==pygame.KEYDOWN:
            #如果按下的是左键
            if event.key==pygame.K_LEFT:
                x-=10
            elif event.key==pygame.K_RIGHT:
                x+=10
            elif event.key==pygame.K_UP:
```

```
                y-=10
            elif event.key==pygame.K_DOWN:
                y+=10
        screen.fill((0,0,0))
        screen.blit(image,(x,y))
        pygame.display.update()
```

执行结果如图 5-2 所示：

图 5-2 示例 5-2 运行效果图

在上面示例中，首先在主窗口加载图片，然后添加了鼠标和键盘事件，按键盘的上下左右键会移动图片的位置。

5.1.3 字体处理

在 Pygame 模块中可以直接调用系统字体，或者可以直接使用 TTF 字体。为了使用字体，需要先创建一个 Font 对象。对系统自带的字体来说，可以使用如下代码创建一个 Font 对象。

```
font = pygame.font.SysFont('arial',18)
```

在上述代码中，第一个参数是字体名，第二个参数表示大小。一般来说，“Arial”字体在很多系统都是存在的，如果找不到，就会使用一个默认字体，这个默认字体和每个操作系

统相关。也可以使用 pygame.font.get_fonts()函数来获取当前系统中所有可用的字体。

一旦创建了一个 font 对象，就可以通过如下代码使用 render 方法来写字，并且可以显示到屏幕中。

```
COLOR_RED = pygame.Color(255, 0, 0)
textSurface = font.render('Hello Pygame',True,COLOR_RED)
```

在上述代码中，第一个参数是写的文字，第二个参数是布尔值，它控制是否开启抗锯齿功能，如果设置为 True 字体会比较平滑，不过相应的速度会有一点点影响；第三个参数是字体的颜色；第四个是背景色，如果你想没有背景色，那么可以不加第四个参数。

【示例 5-3】显示指定样式的文字

```
import pygame
pygame.init()
#创建窗口
screen=pygame.display.set_mode((600,500),0,32)
#设置标题
pygame.display.set_caption('字体处理')
#创建 font 对象
font=pygame.font.SysFont('kaiti',20)
#设置文本内容和颜色
COLOR_RED=pygame.Color(255,255,255)
textSurface=font.render('你好，Pygame',True,COLOR_RED)
#设置文字的坐标
x,y=5,5
#游戏主循环
while True:
    #将文件添加到窗口
    screen.blit(textSurface,(x,y))
    pygame.display.update()
```

执行结果如图 5-3 所示：

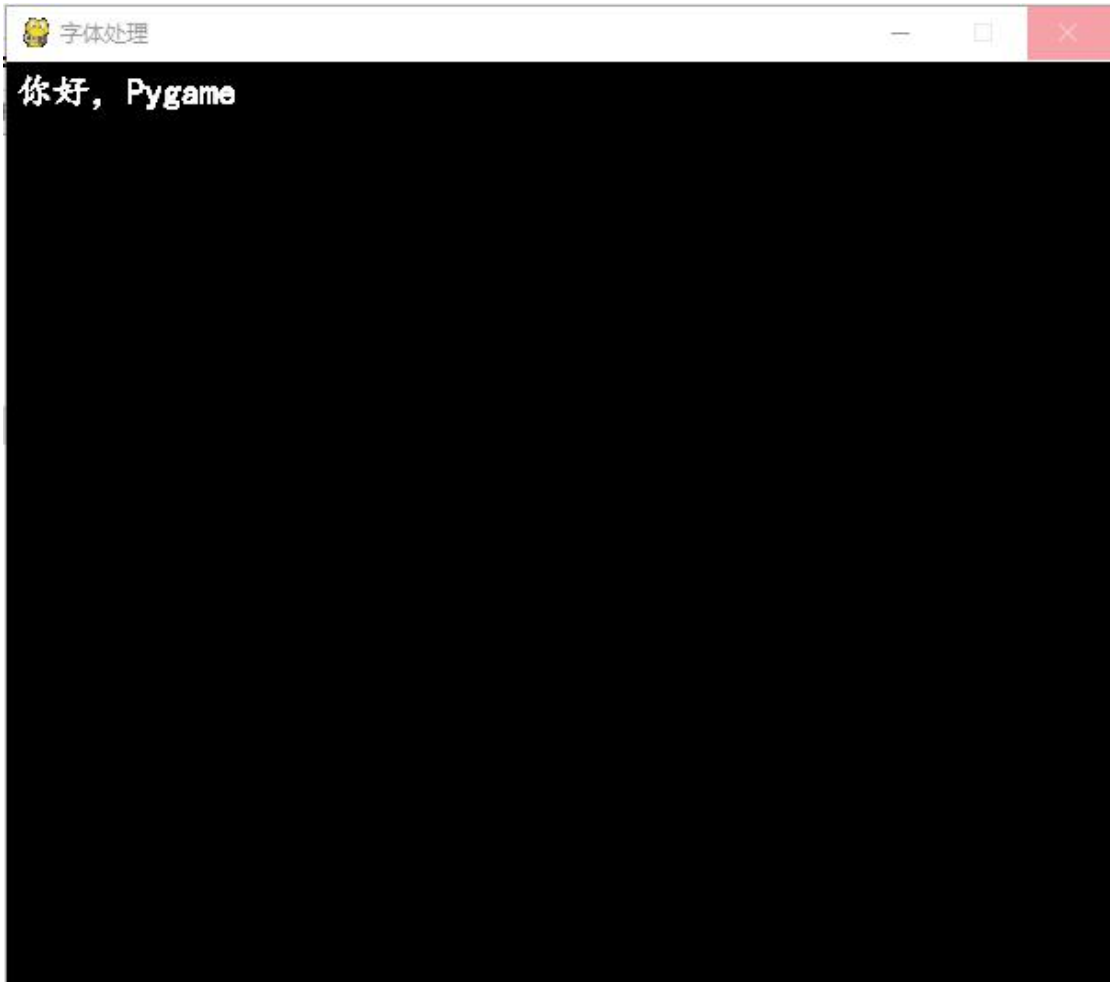


图 5-3 示例 5-3 运行效果图

5.2 坦克大战游戏开发

《坦克大战》是由 Namco 游戏公司开发的一款平面射击游戏，于 1985 年发售。游戏以坦克战斗及保卫基地为主题，属于策略型联机类。同时也是 FC 平台上少有的内建关卡编辑器的几个游戏之一，玩家可自己创建独特的关卡，并通过获取一些道具使坦克和基地得到强化。它看似简单但变化无穷，令人上瘾。本节将介绍使用“Python+Pygame”开发一个简单坦克大战游戏的方法，并详细介绍其具体的实现流程。

5.2.1 项目搭建

本游戏主要分为两个对象，分别是我方坦克和敌方坦克。用户可以通过控制我方的坦克来摧毁敌方的坦克保护自己的“家”，把所有的敌方坦克消灭完达到胜利。敌方的坦克在初始的时候是默认 5 个的（这可以自己设置），当然，如果我方坦克被敌方坦克的子弹打中，游戏结束。从面向对象分析该项目有以下类组成：

- 主类：主要包括开始游戏、结束游戏的功能。

```
class MainGame():
    #开始游戏方法
    def startGame(self):
        pass
    def endGame(self):
        pass
```

- 坦克类：主要包括坦克的创建、显示、移动及射击的功能。

```
class Tank():
    def __init__(self):
        pass
    #坦克的移动方法
    def move(self):
        pass
    #碰撞墙壁的方法
    def hitWalls(self):
        pass
    #射击方法
    def shot(self):
        pass
    #展示坦克
    def displayTank(self):
        pass
```

- 我方坦克类继承坦克类，主要包括创建、与敌方坦克的碰撞方法。

```
class MyTank(Tank):
    def __init__(self):
        pass
    #碰撞敌方坦克的方法
    def hitEnemyTank(self):
        pass
```

- 敌方坦克类继承坦克类，主要包括创建、与我方坦克碰撞方法。

```
class EnemyTank(Tank):
```

```
    def __init__(self):
        pass
    def hitMyTank(self):
        pass
```

□ 子弹类：主要包括子弹的创建、显示及移动的功能。

```
class Bullet():
    def __init__(self):
        pass
    #子弹的移动方法
    def bulletMove(self):
        pass
    #展示子弹的方法
    def displayBullet(self):
        pass
    #我方子弹碰撞敌方坦克的方法
    def hitEnemyTank(self):
        pass
    #敌方子弹与我方坦克的碰撞方法
    def hitMyTank(self):
        pass
    #子弹与墙壁的碰撞
    def hitWalls(self):
        pass
```

□ 墙壁类：主要包括墙壁的创建、显示的功能。

```
class Wall():
    def __init__(self):
        pass
    #展示墙壁的方法
    def displayWall(self):
        pass
```

□ 爆炸效果类：主要展示爆炸效果。

```
class Explode():
```

```
    def __init__(self):
        pass
    #展示爆炸效果
    def displayExplode(self):
        pass
```

□ 音效类：主要播放音乐。

```
class Music():
    def __init__(self):
        pass
    #开始播放音乐
    def play(self):
        pass
```

5.2.2 显示游戏窗口

在游戏设计的前期，要先创建游戏的界面，也就是要为所设计的游戏创建一个窗口，可以通过如下代码实现：

【示例 5-4】显示游戏窗口

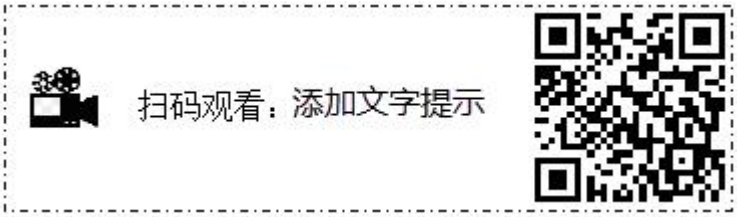
```
import pygame
_display = pygame.display
COLOR_BLACK = pygame.Color(0, 0, 0)
class MainGame():
    #游戏主窗口
    window = None
    SCREEN_HEIGHT = 500
    SCREEN_WIDTH = 800
    def __init__(self):
        pass
    #开始游戏方法
    def startGame(self):
        _display.init()
        #创建窗口加载窗口
```

```
        MainGame.window =
_display.set_mode([MainGame.SCREEN_WIDTH,MainGame.SCREEN_HEIGHT])
        #设置一下游戏标题
        _display.set_caption("坦克大战 v1.03")
        #让窗口持续刷新操作
        while True:
            #给窗口完成一个填充颜色
            MainGame.window.fill(COLOR_BLACK)
            #窗口的刷新
            _display.update()
MainGame().startGame()
```

执行结果如图 5-4 所示：

图 5-4 示例 5-4 运行效果图

5.2.3 添加提示文字



在运行代码时会发现，创建的窗口没有任何提示。然而在实际中希望窗口提示敌方坦克的数量，因此，需要在现有窗口进行必须的改进，添加敌方坦克数量提示。

【示例 5-5】添加左上角提示文字

```
COLOR_RED = pygame.Color(255, 255, 255)
#左上角文字绘制的功能
def getTextSurface(self,text):
    # 初始化字体模块
    pygame.font.init()
    # 选中一个合适的字体
    font = pygame.font.SysFont('kaiti',18)
    # 使用对应的字符完成相关内容的绘制
    textSurface = font.render(text,True,COLOR_RED)
    return textSurface
```

执行结果如图 5-5 所示：

图 5-5 加载我方坦克运行效果图

5.2.4 加载我方坦克

通过加载图片来表示游戏中的坦克，此坦克代表我方坦克，完善我方坦克类。

【示例 5-6】完善我方坦克类

```
class Tank():
    def __init__(self,left,top):
        self.images = {
            'U':pygame.image.load('img/p1tankU.gif'),
            'D':pygame.image.load('img/p1tankD.gif'),
            'L':pygame.image.load('img/p1tankL.gif'),
            'R':pygame.image.load('img/p1tankR.gif')
        }
        self.direction = 'U'
        self.image = self.images[self.direction]
        #坦克所在的区域  Rect->
        self.rect = self.image.get_rect()
        #指定坦克初始化位置 分别距 x，y 轴的位置
        self.rect.left = left
        self.rect.top = top
    #展示坦克(将坦克这个 surface 绘制到窗口中  blit())
    def displayTank(self):
        #1.重新设置坦克的图片
        self.image = self.images[self.direction]
        #2.将坦克加入到窗口中
        MainGame.window.blit(self.image,self.rect)
```

【示例 5-7】开始游戏方法，创建坦克，将坦克添加到窗口

```
import pygame
_display = pygame.display
COLOR_BLACK = pygame.Color(0, 0, 0)
COLOR_RED = pygame.Color(255, 255, 255)
class MainGame():
    #游戏主窗口
    window = None
    SCREEN_HEIGHT = 500
    SCREEN_WIDTH = 800
    #创建我方坦克
```

```
    TANK_P1 = None
    def __init__(self):
        pass
    #开始游戏方法
    def startGame(self):
        _display.init()
        #创建窗口加载窗口(借鉴官方文档)
        MainGame.window = _display.set_mode([MainGame.SCREEN_WIDTH,MainGame.SCREEN_HEIGHT])
        #创建我方坦克
        MainGame.TANK_P1 = Tank(400,300)
        #设置一下游戏标题
        _display.set_caption("坦克大战 v1.03")
        #让窗口持续刷新操作
        while True:
            #给窗口完成一个填充颜色
            MainGame.window.fill(COLOR_BLACK)
            #将绘制文字得到的小画布，粘贴到窗口中
            MainGame.window.blit(self.getTextSurface("剩余敌方坦克%d 辆"%5),(5,5))
            #将我方坦克加入到窗口中
            MainGame.TANK_P1.displayTank()
            #窗口的刷新
            _display.update()
#调用开始游戏的方法
MainGame().startGame()
```

执行结果如图 5-6 所示：

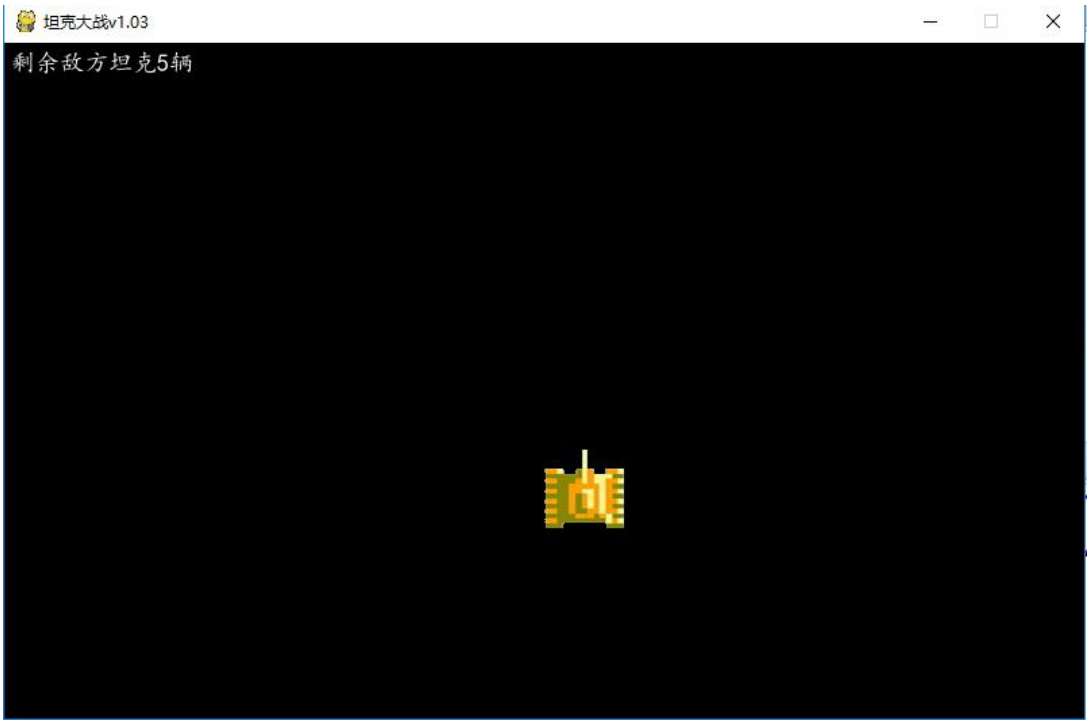
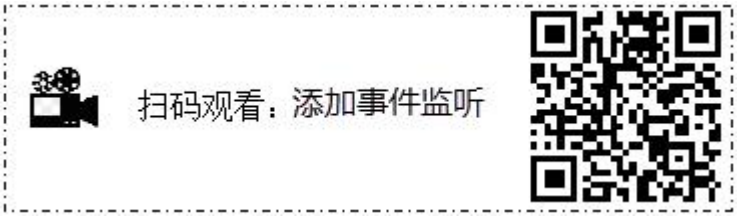


图 5-6 加载我方坦克运行效果图

5.2.5 添加事件监听



上面创建的坦克还不能动，显然不是创建游戏的目的，因此，要给创建的坦克赋予“生命”。添加事件监听，控制上、下、左、右四个方向键，实现针对不同的键改变坦克的方向及移动功能，点击关闭退出游戏。

【示例 5-8】退出方法实现

```
def endGame(self):
    print("谢谢使用")
    #结束 python 解释器
    exit()
```

【示例 5-9】坦克类中添加速度属性，实现坦克移动

```
#坦克的移动方法
def move(self):
    if self.direction == 'L':
```

```
            if self.rect.left > 0:
                self.rect.left -= self.speed
        elif self.direction == 'R':
            if self.rect.left + self.rect.height < MainGame.SCREEN_WIDTH:
                self.rect.left += self.speed
        elif self.direction == 'U':
            if self.rect.top > 0:
                self.rect.top -= self.speed
        elif self.direction == 'D':
            if self.rect.top + self.rect.height < MainGame.SCREEN_HEIGHT:
                self.rect.top += self.speed
```

【示例 5-10】坦克类中添加移动开关属性，按下上、下、左、右四个方向键修改坦克的方向及开关状态，按下关闭键，调用关闭方法退出游戏

```
#获取程序期间所有事件(鼠标事件，键盘事件)
    def getEvent(self):
        #1.获取所有事件
        eventList = pygame.event.get()
        #2.对事件进行判断处理(1、点击关闭按钮  2、按下键盘上的某个按键)
        for event in eventList:
            #判断 event.type  是否 QUIT，如果是退出的话，直接调用程序结束方法
            if event.type == pygame.QUIT:
                self.endGame()
            #判断事件类型是否为按键按下，如果是，继续判断按键是哪一个按键，来进行对应的处理
            if event.type == pygame.KEYDOWN:
                #具体是哪一个按键的处理
                if event.key == pygame.K_LEFT:
                    print("坦克向左调头，移动")
                    #修改坦克方向
                    MainGame.TANK_P1.direction = 'L'
                    MainGame.TANK_P1.stop = False
                elif event.key == pygame.K_RIGHT:
                    print("坦克向右调头，移动")
```

```
                    # 修改坦克方向
                    MainGame.TANK_P1.direction = 'R'
                    MainGame.TANK_P1.stop = False
                elif event.key == pygame.K_UP:
                    print("坦克向上调头，移动")
                    # 修改坦克方向
                    MainGame.TANK_P1.direction = 'U'
                    MainGame.TANK_P1.stop = False
                elif event.key == pygame.K_DOWN:
                    print("坦克向下掉头，移动")
                    # 修改坦克方向
                    MainGame.TANK_P1.direction = 'D'
                    MainGame.TANK_P1.stop = False
                elif event.key == pygame.K_SPACE:
                    print("发射子弹")
            #结束游戏方法
            if event.type == pygame.KEYUP:
                #松开的如果是方向键，才更改移动开关状态
                if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT or
event.key == pygame.K_UP or event.key == pygame.K_DOWN:
                    # 修改坦克的移动状态
                    MainGame.TANK_P1.stop = True
```

执行结果如图 5-7 所示：

图 5-7 我方坦克运行效果图

5.2.6 随机生成敌方坦克

敌方坦克的创建与我方坦克相同，都是通过加装图片来实现。

【示例 5-11】初始化敌方坦克

```
class EnemyTank(Tank):
    def __init__(self,left,top,speed):
        self.images = {
            'U': pygame.image.load('img/enemy1U.gif'),
            'D': pygame.image.load('img/enemy1D.gif'),
            'L': pygame.image.load('img/enemy1L.gif'),
            'R': pygame.image.load('img/enemy1R.gif')
        }
```

```
        self.direction = self.randDirection()
        self.image = self.images[self.direction]
        # 坦克所在的区域   Rect->
        self.rect = self.image.get_rect()
        # 指定坦克初始化位置 分别距 x，y 轴的位置
        self.rect.left = left
        self.rect.top = top
        # 新增速度属性
        self.speed = speed
        self.stop = True
```

【示例 5-12】生成随机的四个方向

```
def randDirection(self):
    num = random.randint(1,4)
    if num == 1:
        return 'U'
    elif num == 2:
        return 'D'
    elif num == 3:
        return 'L'
    elif num == 4:
        return 'R'
```

【示例 5-13】创建敌方坦克

```
#创建敌方坦克
def creatEnemyTank(self):
    top = 100
    speed = random.randint(3,6)
    for i in range(MainGame.EnemTank_count):
        #每次都随机生成一个 left 值
        left = random.randint(1, 7)
        eTank = EnemyTank(left*100,top,speed)
        MainGame.EnemyTank_list.append(eTank)
```

【示例 5-14】实现敌方坦克的随机移动

```
#随机移动
def randMove(self):
    if self.step <= 0:
        self.direction = self.randDirection()
        self.step = 50
    else:
        self.move()
        self.step -= 1
```

【示例 5-15】将敌方坦克加到窗口中

```
#将敌方坦克加入到窗口中
def blitEnemyTank(self):
    for eTank in MainGame.EnemyTank_list:
        eTank.displayTank()
        #坦克移动的方法
        eTank.randMove()
```

执行结果如图 5-8 所示：

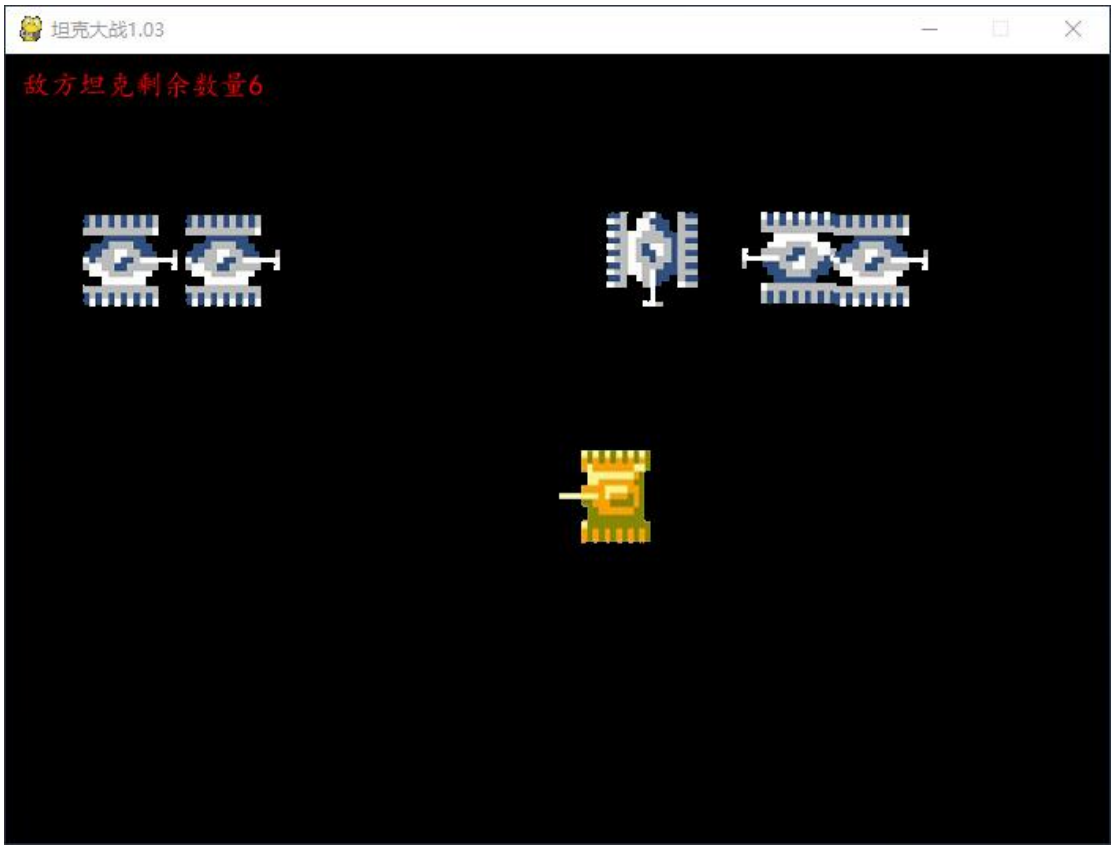
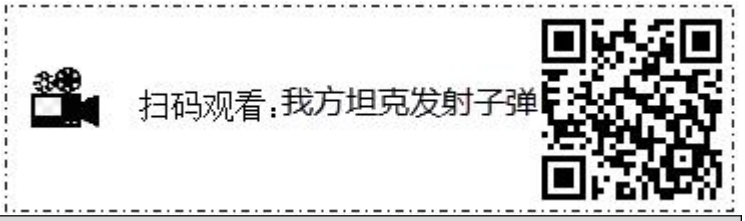


图 5-8 敌方坦克运行效果图

5.2.7 我方坦克发射子弹

【示例 5-16】初始化子弹




```
def __init__(self,tank):
    #图片
    self.image = pygame.image.load('img/enemymissile.gif')
    #方向（坦克方向）
    self.direction = tank.direction
    #位置
    self.rect = self.image.get_rect()
    if self.direction == 'U':
        self.rect.left = tank.rect.left + tank.rect.width/2 - self.rect.width/2
        self.rect.top = tank.rect.top - self.rect.height
    elif self.direction == 'D':
        self.rect.left = tank.rect.left + tank.rect.width / 2 - self.rect.width / 2
```

```
            self.rect.top = tank.rect.top + tank.rect.height
        elif self.direction == 'L':
            self.rect.left = tank.rect.left - self.rect.width / 2 - self.rect.width / 2
            self.rect.top = tank.rect.top + tank.rect.width / 2 - self.rect.width / 2
        elif self.direction == 'R':
            self.rect.left = tank.rect.left + tank.rect.width
            self.rect.top = tank.rect.top + tank.rect.width / 2 - self.rect.width / 2
        #速度
        self.speed = 7
        #用来记录子弹是否活着
        self.live = True
```

【示例 5-17】实现子弹移动

```
    #子弹的移动方法
    def bulletMove(self):
        if self.direction == 'U':
            if self.rect.top > 0:
                self.rect.top -= self.speed
            else:
                #修改状态值
                self.live = False
        elif self.direction == 'D':
            if self.rect.top < MainGame.SCREEN_HEIGHT - self.rect.height:
                self.rect.top += self.speed
            else:
                #  修改状态值
                self.live = False
        elif self.direction == 'L':
            if self.rect.left > 0:
                self.rect.left -= self.speed
            else:
                #  修改状态值
                self.live = False
        elif self.direction == 'R':
```

```
            if self.rect.left < MainGame.SCREEN_WIDTH - self.rect.width:
                self.rect.left += self.speed
            else:
                # 修改状态值
                self.live = False
```

【示例 5-18】展示子弹

```
#展示子弹的方法
def displayBullet(self):
    MainGame.window.blit(self.image,self.rect)
```

【示例 5-19】按空格键产生子弹，并将子弹添加到子弹列表中

```
                elif event.key == pygame.K_SPACE:
                    print("发射子弹")
                    #产生一颗子弹
                    m = Bullet(MainGame.TANK_P1)
                    #将子弹加入到子弹列表
                    MainGame.Bullet_list.append(m)
```

【示例 5-20】将子弹添加到窗口

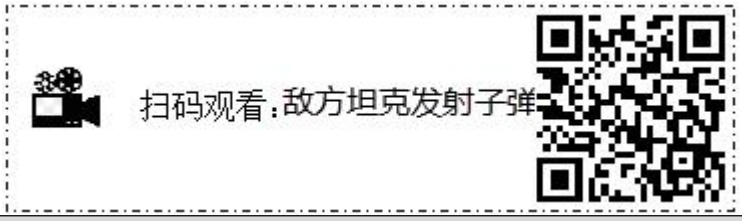
```
#将我方子弹加入到窗口中
def blitBullet(self):
    for bullet in MainGame.Bullet_list:
        #如果子弹还活着，绘制出来，否则，直接从列表中移除该子弹
        if bullet.live:
            bullet.displayBullet()
            # 让子弹移动
            bullet.bulletMove()
        else:
            MainGame.Bullet_list.remove(bullet)
```

执行结果如图 5-9 所示：

图 5-9 敌方坦克运行效果图

5.2.8 敌方坦克随机发射子弹

【示例 5-21】敌方坦克发射子弹




```
def shot(self):
    num = random.randint(1,1000)
    if num    <= 20:
        return Bullet(self)
```

【示例 5-22】敌方坦克加入窗口后，发射子弹，并将子弹添加到敌方子弹列表中

```
#将敌方坦克加入到窗口中
def blitEnemyTank(self):
    for eTank in MainGame.EnemyTank_list:
        eTank.displayTank()
        #坦克移动的方法
```

```
            eTank.randMove()
            #调用敌方坦克的射击
            eBullet = eTank.shot()
            #如果子弹为 None。不加入到列表
            if eBullet:
                # 将子弹存储敌方子弹列表
                MainGame.Enemy_bullet_list.append(eBullet)
```

【示例 5-23】将敌方发射的子弹添加到窗口

```
    #将敌方子弹加入到窗口中
    def blitEnemyBullet(self):
        for eBullet in MainGame.Enemy_bullet_list:
            # 如果子弹还活着，绘制出来，否则，直接从列表中移除该子弹
            if eBullet.live:
                eBullet.displayBullet()
                # 让子弹移动
                eBullet.bulletMove()
            else:
                MainGame.Enemy_bullet_list.remove(eBullet)
```

执行结果如图 5-10 所示：

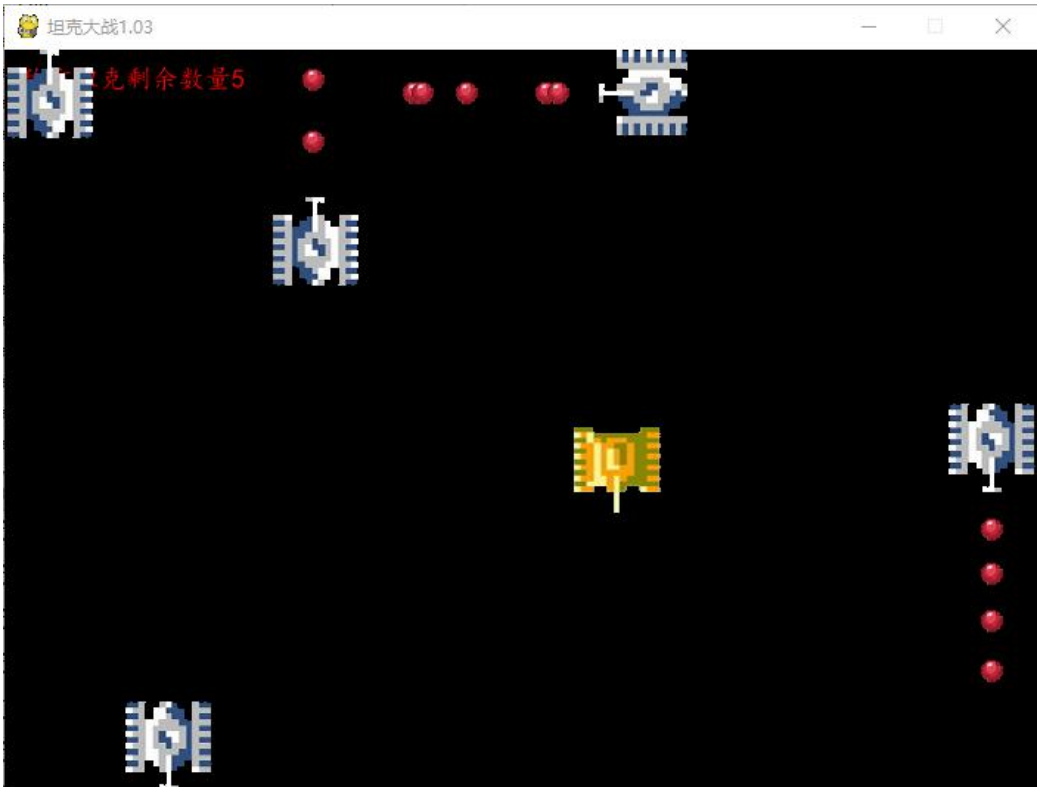
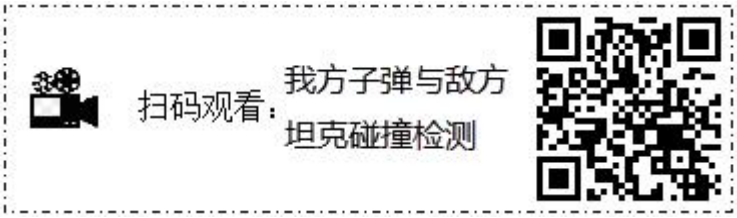


图 5-10 敌方坦克运行效果图

5.2.9 我方法子弹与敌方坦克的碰撞检测

在游戏开发中，通常把显示图像的对象叫做精灵 Spire，精灵需要有两个属性 image 要显示的图像，rect 图像要显示在屏幕的位置。



在 Pygame 框架中，使用 pygame. sprite 模块中的内置函数可以实现碰撞检测。代码如下：

```
pygame.sprite.collide_rect(first, second) #返回布尔值
```

pygame.sprite.Sprite 是 pygame 精灵的基类，一般来说，总是需要写一个自己的精灵类继承 pygame.sprite.Sprite。让坦克类、子弹类都继承自己的精灵类。

【示例 5-24】精灵类的实现

```
class BaseItem(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
class Tank(BaseItem):
class Bullet(BaseItem):
```

【示例 5-25】在子弹类中增加我方子弹碰撞敌方坦克的方法，如果发生碰撞，修改我方子弹及敌方坦克 live 属性的状态值

```
#新增我方子弹碰撞敌方坦克的方法
def hitEnemyTank(self):
    for eTank in MainGame.EnemyTank_list:
        if pygame.sprite.collide_rect(eTank,self):
            self.live = False
            eTank.live = False
```

【示例 5-26】我方子弹移动后判断子弹是否与敌方坦克碰撞

```
#将我方子弹加入到窗口中
def blitBullet(self):
    for bullet in MainGame.Bullet_list:
        #如果子弹还活着，绘制出来，否则，直接从列表中移除该子弹
        if bullet.live:
            bullet.displayBullet()
            # 让子弹移动
            bullet.bulletMove()
            # 调用我方子弹与敌方坦克的碰撞方法
            bullet.hitEnemyTank()
        else:
            MainGame.Bullet_list.remove(bullet)
```

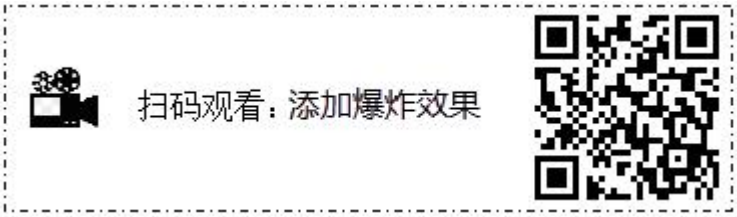
执行结果如图 5-11 所示：

图 5-11 我方子弹与敌方坦克碰撞运行效果图

从上图可以看到如果我方子弹与敌方坦克发生碰撞后，敌方坦克就会消失，同时敌方坦克的数量也减一。

5.2.10 添加爆炸效果

在上述示例代码中只实现了我方子弹与敌方坦克发生碰撞后，敌方坦克就会消失，现在在敌方坦克消失时候再添加爆炸效果。



【示例 5-27】初始化爆炸类

```
class Explode():
    def __init__(self,tank):
        self.rect = tank.rect
        self.step = 0
        self.images = [
            pygame.image.load('img/blast0.gif'),
            pygame.image.load('img/blast1.gif'),
            pygame.image.load('img/blast2.gif'),
```

```
            pygame.image.load('img/blast3.gif'),
            pygame.image.load('img/blast4.gif')
        ]
        self.image = self.images[self.step]
        self.live = True
```

【示例 5-28】展示爆炸效果

```
    #展示爆炸效果
    def displayExplode(self):
        if self.step < len(self.images):
            MainGame.window.blit(self.image, self.rect)
            self.image = self.images[self.step]
            self.step += 1
        else:
            self.live = False
            self.step = 0
```

【示例 5-29】在我方子弹碰撞敌方坦克的方法中，如果检测到碰撞，产生爆炸类，并将爆炸效果添加到爆炸列表。

```
    #新增我方子弹碰撞敌方坦克的方法
    def hitEnemyTank(self):
        for eTank in MainGame.EnemyTank_list:
            if pygame.sprite.collide_rect(eTank,self):
                #产生一个爆炸效果
                explode = Explode(eTank)
                #将爆炸效果加入到爆炸效果列表
                MainGame.Explode_list.append(explode)
                self.live = False
                eTank.live = False
```

【示例 5-30】将爆炸效果添加到窗口。

```
    #新增方法：  展示爆炸效果列表
    def displayExplodes(self):
        for explode in MainGame.Explode_list:
```

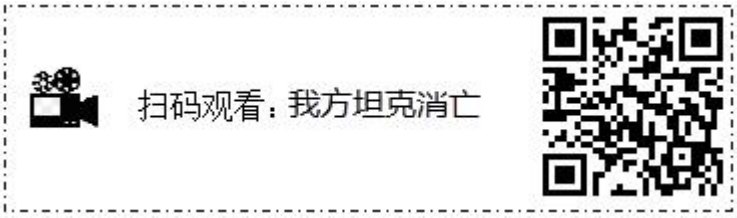
```
            if explode.live:
                explode.displayExplode()
            else:
                MainGame.Explode_list.remove(explode)
```

执行结果如图 5-12 所示：

图 5-12 敌方子弹与我方坦克碰撞运行效果图

5.2.11 我方坦克的消亡

在子弹类中，新增敌方子弹与我方坦克的碰撞方法。如果发生碰撞，修改敌方子弹和我方坦克的状态，同时产生爆炸效果。



【示例 5-31】在子弹类中，新增敌方子弹与我方坦克的碰撞方法

```
    #新增敌方子弹与我方坦克的碰撞方法
    def hitMyTank(self):
        if pygame.sprite.collide_rect(self,MainGame.TANK_P1):
            # 产生爆炸效果，并加入到爆炸效果列表中
            explode = Explode(MainGame.TANK_P1)
```

```
                MainGame.Explode_list.append(explode)
                #修改子弹状态
                self.live = False
                #修改我方坦克状态
                MainGame.TANK_P1.live = False
```

【示例 5-32】添加敌方子弹到窗口，如果子弹还活着，显示子弹、调用子弹移动并判断敌方子弹是否与我方坦克发生碰撞。

```
    #将敌方子弹加入到窗口中
    def blitEnemyBullet(self):
        for eBullet in MainGame.Enemy_bullet_list:
            # 如果子弹还活着，绘制出来，否则，直接从列表中移除该子弹
            if eBullet.live:
                eBullet.displayBullet()
                # 让子弹移动
                eBullet.bulletMove()
                if MainGame.TANK_P1 and MainGame.TANK_P1.live:
                    eBullet.hitMyTank()
            else:
                MainGame.Enemy_bullet_list.remove(eBullet)
```

执行结果如图 5-13 所示：

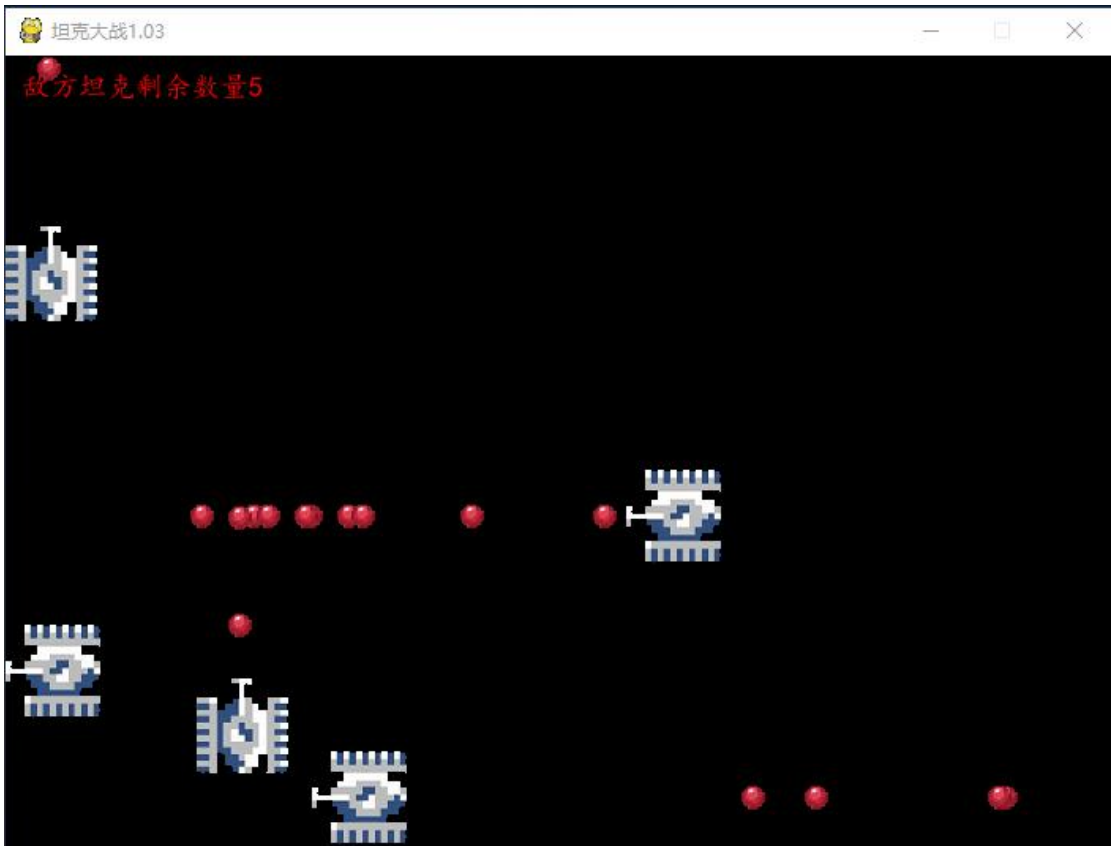
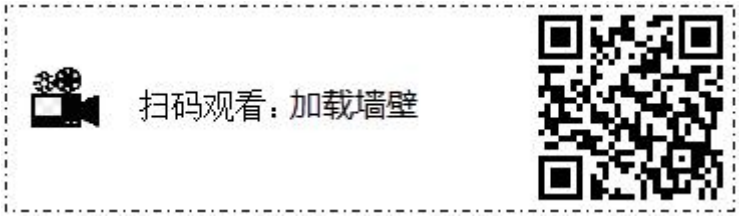


图 5-13 敌方子弹与我方坦克碰撞运行效果图

5.2.12 加载墙壁



【示例 5-33】初始化墙壁类

```
class Wall():
    def __init__(self,left,top):
        self.image = pygame.image.load('img/steels.gif')
        self.rect = self.image.get_rect()
        self.rect.left = left
        self.rect.top = top
        #用来判断墙壁是否应该在窗口中展示
        self.live = True
        #用来记录墙壁的生命值
        self.hp = 3
    #展示墙壁的方法
    def displayWall(self):
```

```
        MainGame.window.blit(self.image,self.rect)
```

【示例 5-34】完善创建墙壁的方法。

```
    #创建墙壁的方法
    def creatWalls(self):
        for i in range(6):
            wall = Wall(130*i,240)
            MainGame.Wall_list.append(wall)
```

【示例 5-35】将墙壁加入到窗口。

```
    def blitWalls(self):
        for wall in MainGame.Wall_list:
            if wall.live:
                wall.displayWall()
            else:
                MainGame.Wall_list.remove(wall)
```

执行结果如图 5-14 所示：

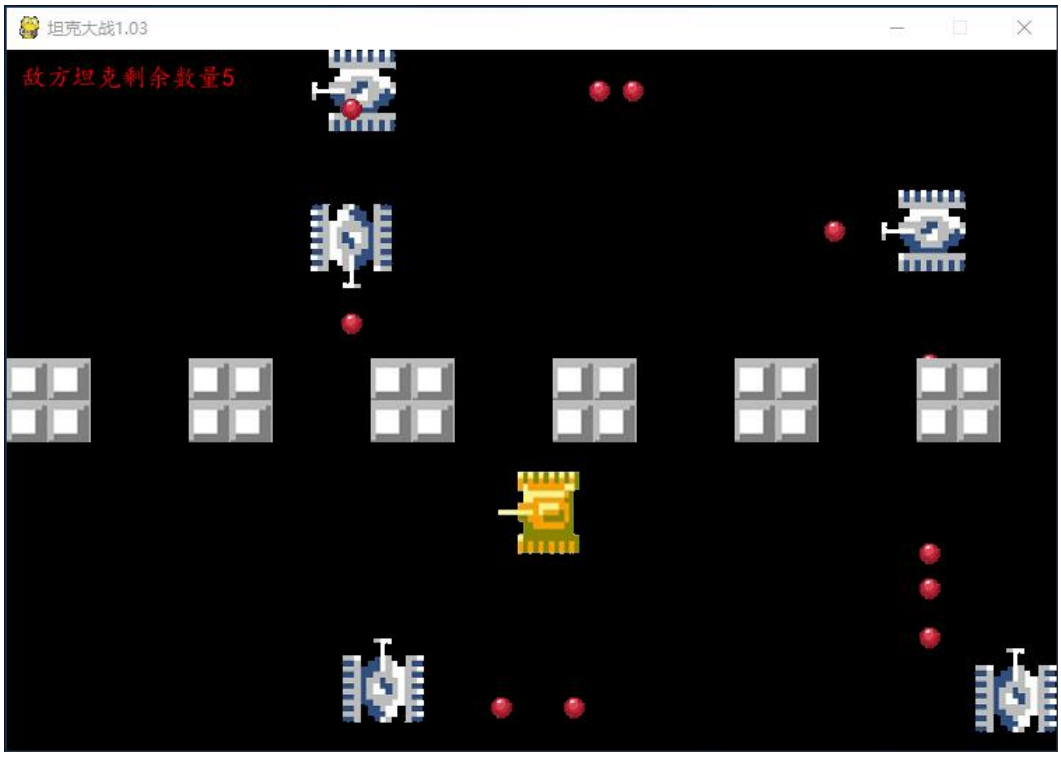
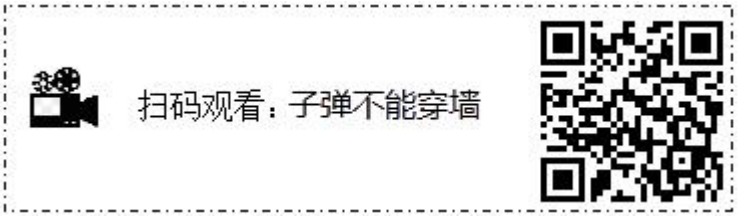


图 5-14 添加墙壁运行效果图

5.2.13 子弹不能穿墙



从上面的示例中可以看到子弹可以穿过墙壁，因此需要在子弹类中新增方法，子弹与墙壁的碰撞，如果子弹与墙壁碰撞，修改子弹的状态，墙壁的生命值减少。如果墙壁的生命值小于等于零时候修改墙壁的状态。

【示例 5-36】子弹与墙壁的碰撞。

```
#新增子弹与墙壁的碰撞
def hitWalls(self):
    for wall in MainGame.Wall_list:
        if pygame.sprite.collide_rect(wall,self):
            #修改子弹的 live 属性
            self.live = False
            wall.hp -= 1
            if wall.hp <= 0:
                wall.live = False
```

执行结果如图 5-15 所示：

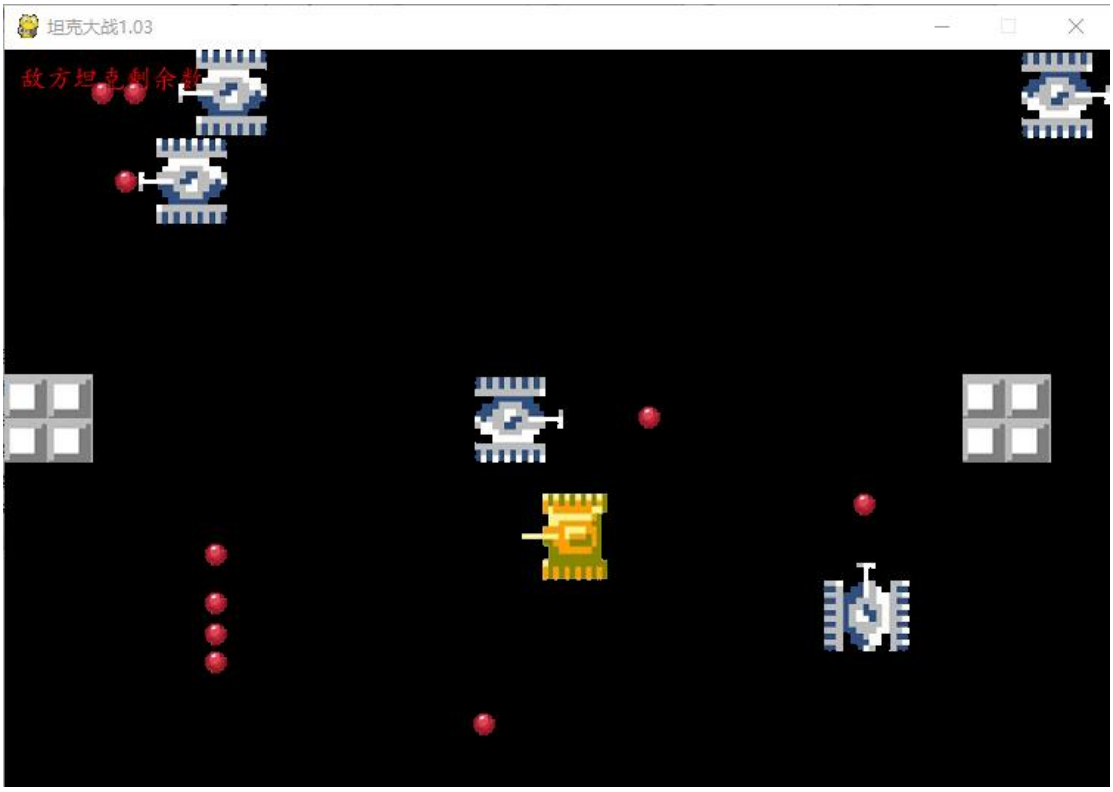
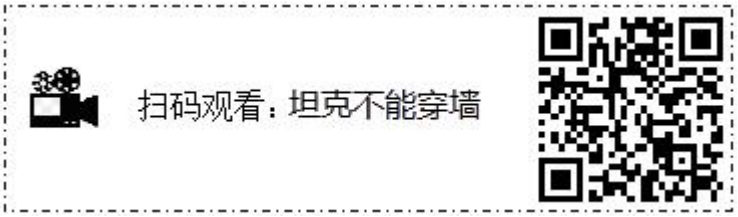


图 5-15 子弹与墙壁碰撞运行效果图

5.2.14 坦克不能穿墙



如果坦克与墙壁碰撞，则坦克不能继续移动，需要修改坦克的坐标为移动之前的。因此在坦克类中新增属性 oldLeft、oldTop 记录移动之前的坐标，新增 stay()、hitWalls()方法。

【示例 5-37】坦克不能穿墙

```
    def stay(self):
        self.rect.left = self.oldLeft
        self.rect.top = self.oldTop
    #新增碰撞墙壁的方法
    def hitWalls(self):
        for wall in MainGame.Wall_list:
            if pygame.sprite.collide_rect(wall,self):
                self.stay()
```

执行结果如图 5-16 所示：

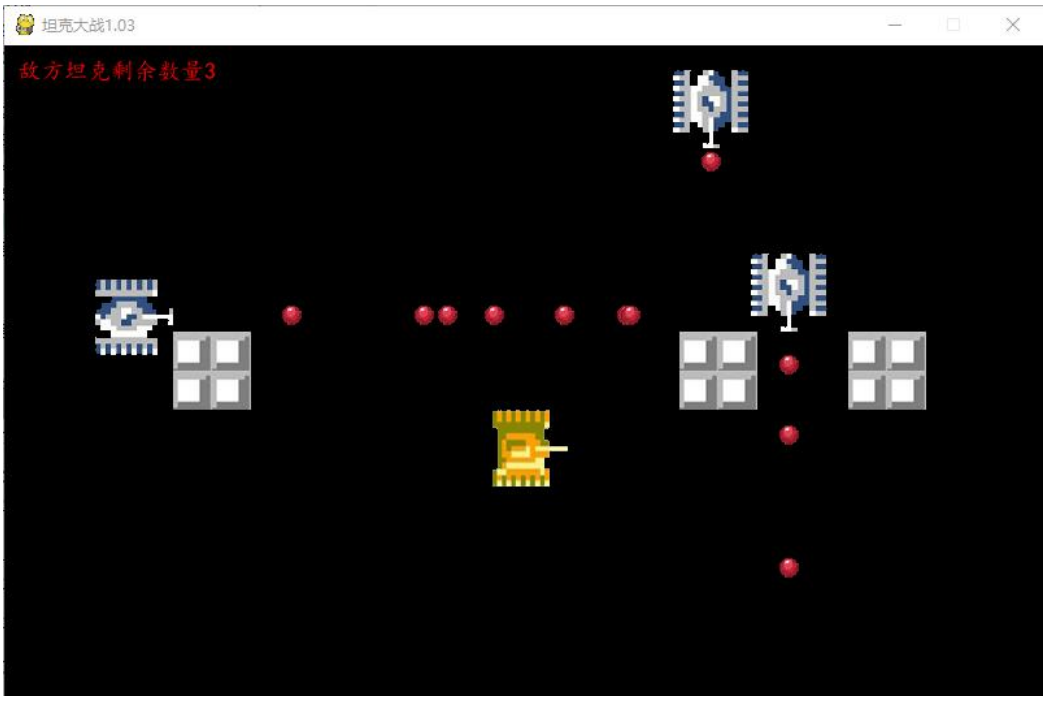
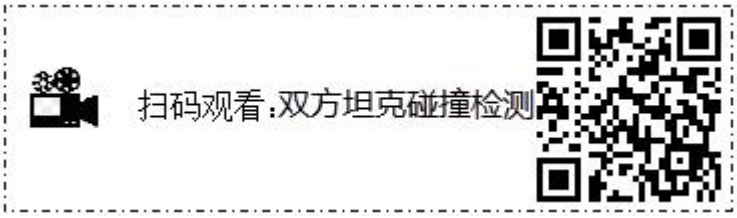


图 5-16 子弹与墙壁碰撞运行效果图

5.2.15 双方坦克之间的碰撞检测

如果我方坦克碰撞到敌方坦克，则我方坦克再不能继续移动。同理如果敌方坦克碰撞到我方坦克也不能继续移动。



【示例 5-38】在我方坦克类中新增我方坦克与敌方坦克碰撞的方法。

```
class MyTank(Tank):
    def __init__(self,left,top):
        super(MyTank, self).__init__(left,top)
    #新增主动碰撞到敌方坦克的方法
    def hitEnemyTank(self):
        for eTank in MainGame.EnemyTank_list:
            if pygame.sprite.collide_rect(eTank,self):
                self.stay()
```

【示例 5-39】我方坦克移动后，调用是否与敌方坦克发生碰撞的方法。

```
            #根据坦克的开关状态调用坦克的移动方法
            if MainGame.TANK_P1 and not MainGame.TANK_P1.stop:
                MainGame.TANK_P1.move()
                #调用碰撞墙壁的方法
                MainGame.TANK_P1.hitWalls()
                MainGame.TANK_P1.hitEnemyTank()
```

【示例 5-40】在敌方坦克类中，新增敌方坦克碰撞我方坦克的方法。

```
def hitMyTank(self):
    if MainGame.TANK_P1 and MainGame.TANK_P1.live:
        if pygame.sprite.collide_rect(self, MainGame.TANK_P1):
            # 让敌方坦克停下来    stay()
            self.stay()
```

【示例 5-41】敌方坦克添加到窗口时候，调用是否与我方坦克碰撞的方法。

```
#将敌方坦克加入到窗口中
def blitEnemyTank(self):
    for eTank in MainGame.EnemyTank_list:
        if eTank.live:
            eTank.displayTank()
            # 坦克移动的方法
            eTank.randMove()
            #调用敌方坦克与墙壁的碰撞方法
            eTank.hitWalls()
            #敌方坦克是否撞到我方坦克
            eTank.hitMyTank()
            # 调用敌方坦克的射击
            eBullet = eTank.shot()
            # 如果子弹为 None。不加入到列表
            if eBullet:
                # 将子弹存储敌方子弹列表
```

```
                MainGame.Enemy_bullet_list.append(eBullet)
            else:
                MainGame.EnemyTank_list.remove(eTank)
```

执行结果如图 5-17 所示：

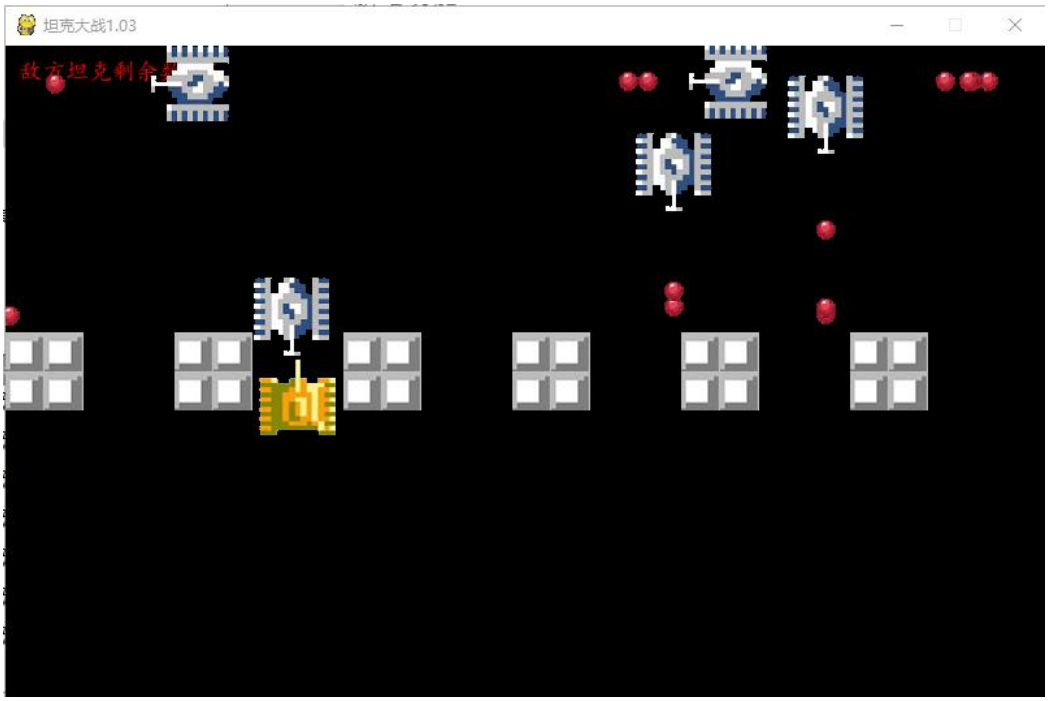


图 5-17 双方坦克发生碰撞运行效果图

5.2.16 坦克大战之音效处理

music 是 pygame 中控制流音频的 pygame 模块，音乐模块与 pygame.mixer 紧密相连，pygame.mixer 是一个用来处理声音的模块，其含义为“混音器”。游戏中对声音的处理一般包括制造声音和播放声音两部分。使用 pygame.mixer.music.load()加载一个播放音乐的文件，pygame.mixer.music.play() 开始播放音乐流。

【示例 5-42】初始化音效类。

```
class Music():
    def __init__(self,fileName):
        self.fileName = fileName
        #先初始化混合器
        pygame.mixer.init()
```

```
        pygame.mixer.music.load(self.fileName)
    #开始播放音乐
    def play(self):
        pygame.mixer.music.play()
```

【示例 5-43】创建坦克时，添加音效。

```
    #创建我方坦克的方法
    def creatMyTank(self):
        # 创建我方坦克
        MainGame.TANK_P1 = MyTank(400, 300)
        #创建音乐对象
        music = Music('img/start.wav')
        #调用播放音乐方法
        music.play()
```

【示例 5-44】我方坦克发射子弹时，添加音效。

```
                    elif event.key == pygame.K_SPACE:
                        print("发射子弹")
                        if len(MainGame.Bullet_list) < 3:
                            # 产生一颗子弹
                            m = Bullet(MainGame.TANK_P1)
                            # 将子弹加入到子弹列表
                            MainGame.Bullet_list.append(m)
                            music = Music('img/fire.wav')
                            music.play()
```

到此为止，整个实例介绍完毕，执行后的效果如图 5-18 所示：

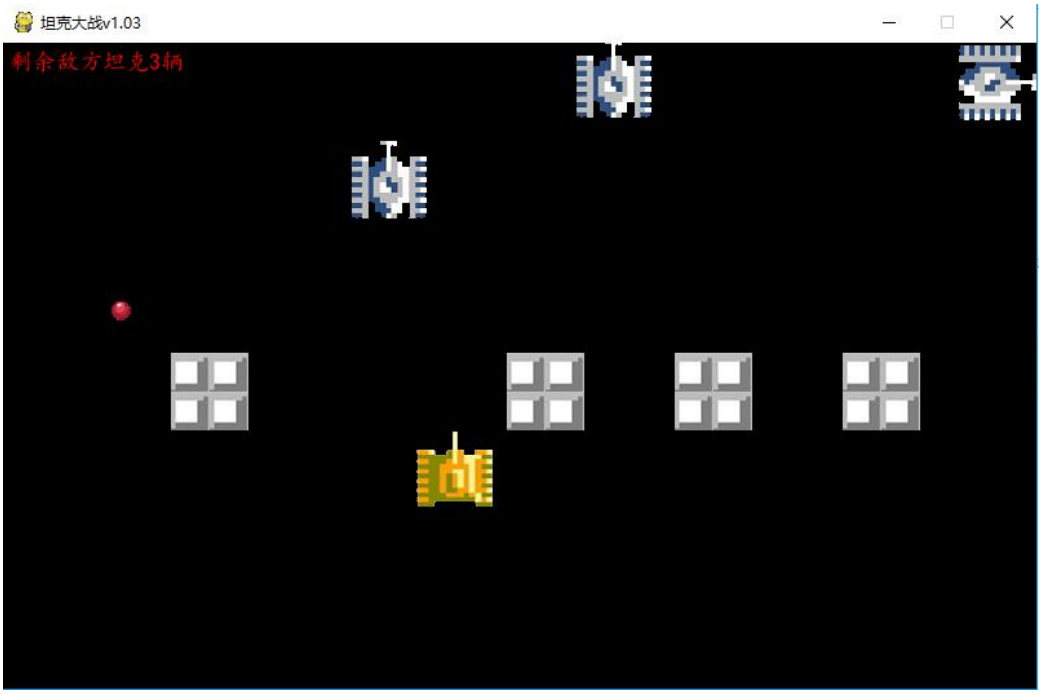


图 5-18 运行效果图

习题

一、编码题

1. 使用 Pygame 模块设置窗口和加载图片，以及加载音乐
2. 使用 Pygame 模块，完成坦克大战游戏。
3. 使用 Pygame 模块，编写一个太空飞机大战游戏。
4. 使用 Pygame 模块，编写一个俄罗斯方块游戏。

第六章 数据库编程

在前面介绍了使用 open 函数打开文件，然后以文件或二进制形式读写文件中内容的方式，尽管可以将数据保存到文件中，以及从文件中读取数据。但使用这种方式读写的数据都是原始的格式，而且非常简单。如果要保存非常多的数据，而且要求快速被程序识别，甚至在海量数据中搜索到想要的数据，就要使用结构化的存储方式。Python 通过模块支持了大量的数据存储和查找解决方案，如桌面数据库 SQLite，关系型数据库 MySQL，还有 NoSQL、Excel 等。本章将带领读者学习 Python 数据存储方式桌面数据库 SQLite，关系型数据库 MySQL，让读者深入了解如何使用这些存储技术对数据进行处理。

通过阅读本章，你可以：

- 掌握如何在 Python 语言中操作 SQLite 数据库
- 了解什么是关系型数据库
- 掌握 MySQL 数据库的下载、安装
- 掌握如何在 Python 语言中操作 MySQL 数据库

6.1 SQLite 数据库

SQLite 是一个开源、小巧、零配置的关系型数据库，支持多种平台。现在运行 Android、IOS 等系统的设备基本都使用 SQLite 数据库作为本地存储方案。尽管 Python 语言在很多场景用于开发服务器端应用，使用的是网络关系型数据库或 NoSQL 数据库，但有一些数据需要保存到本地，虽然可以使用 XML、JSON 等格式保存这些数据，但对数据检索很不方便，因此将数据保存到 SQLite 数据库中，将成为本地存储的最佳方案。

6.1.1 管理 SQLite 数据库

SQLite 数据库的管理工具很多，SQLite 官方提供了一个命令行工具用于管理 SQLite 数据库，不过这个命令行工具需要输入大量的命令才能操作 SQLite 数据库，太麻烦，不建议使用。本节介绍一款平台的 SQLite 数据库管理工具 DB Browser for SQLite，这是一款免费开源的 SQLite 数据库管理工具。官网地址：https://sqlitebrowser.org/。

进入 DB Browser for SQLite 官网后，选择菜单 Download，如图 6-1 所示：

图 6-1 DB Browser for SQLite 官网

选择对应的版本进行下载，例如我的操作系统是 Windows 64 位。如图 6-2 所示：

图 6-2 Windows 安装版下载

安装好 DB Browser for SQLite 后，直接启动即可看到如图 6-3 所示的主界面。

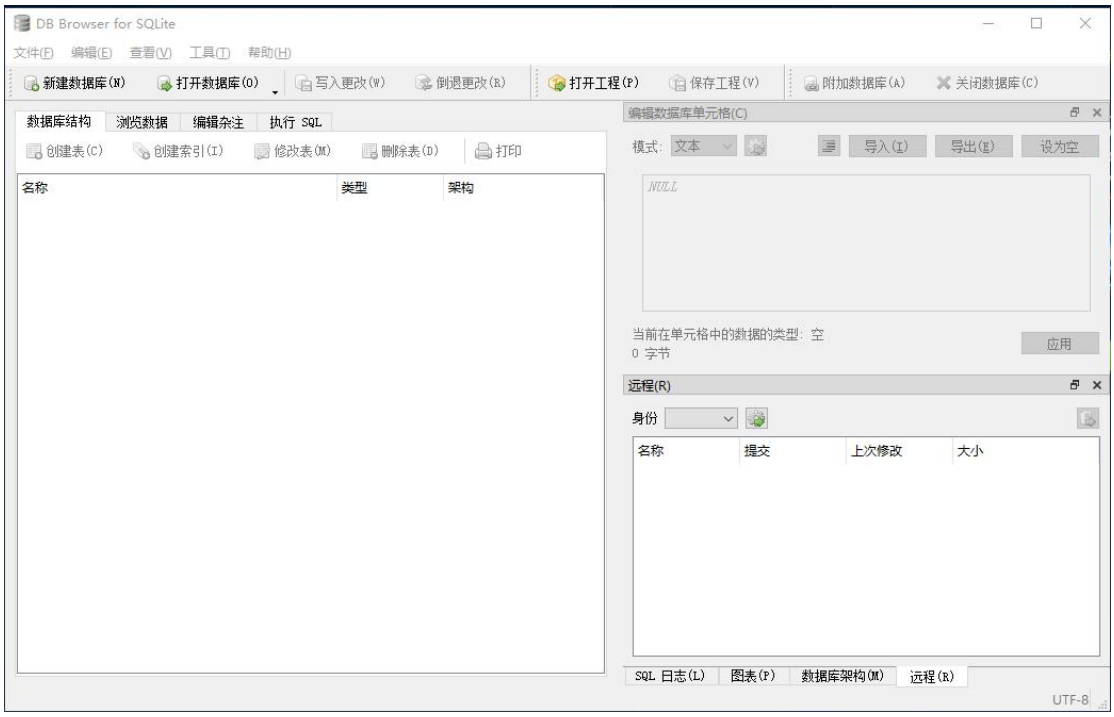


图 6-3 DB Browser for SQLite 主界面

单击左上角的“新建数据库”和“打开数据库”按钮，可以新建和打开 SQLite 数据库。图 6-4 所示是打开数据库后的效果，在主界面会列出数据库中的表、视图等内容。

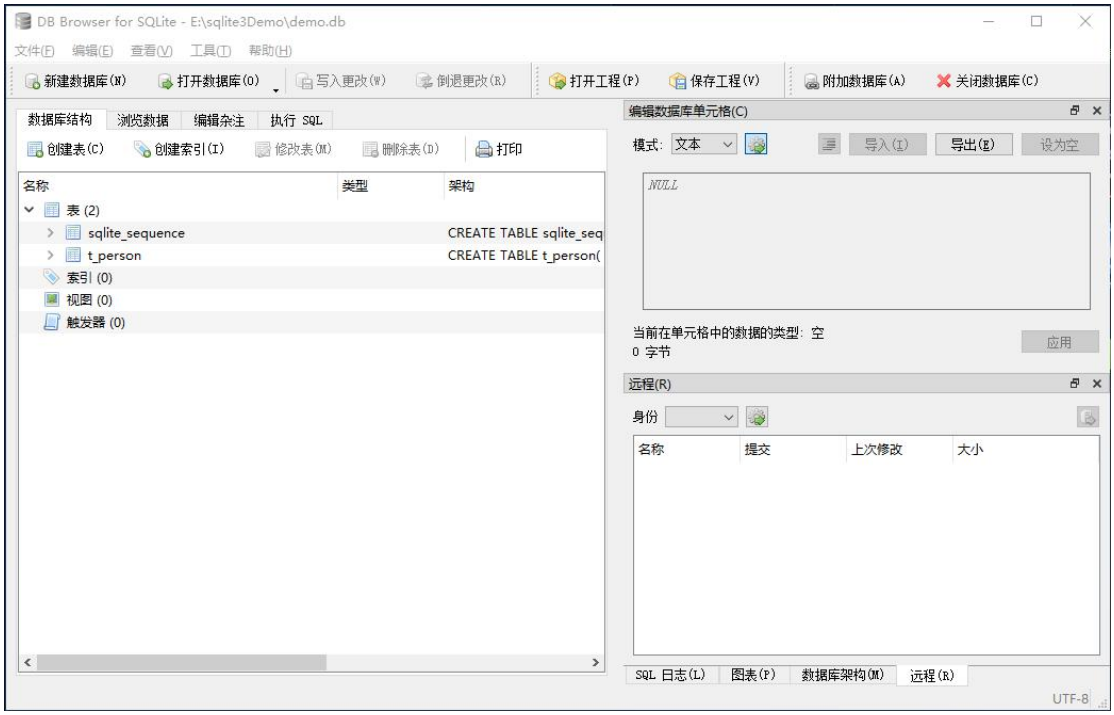


图 6-4 打开 SQLite 数据库

如果想查看表或视图中的记录，可以切换到主界面上方的“浏览数据”选项卡，再从下方的列表中选择要查看的表或视图，如图 6-5 所示：

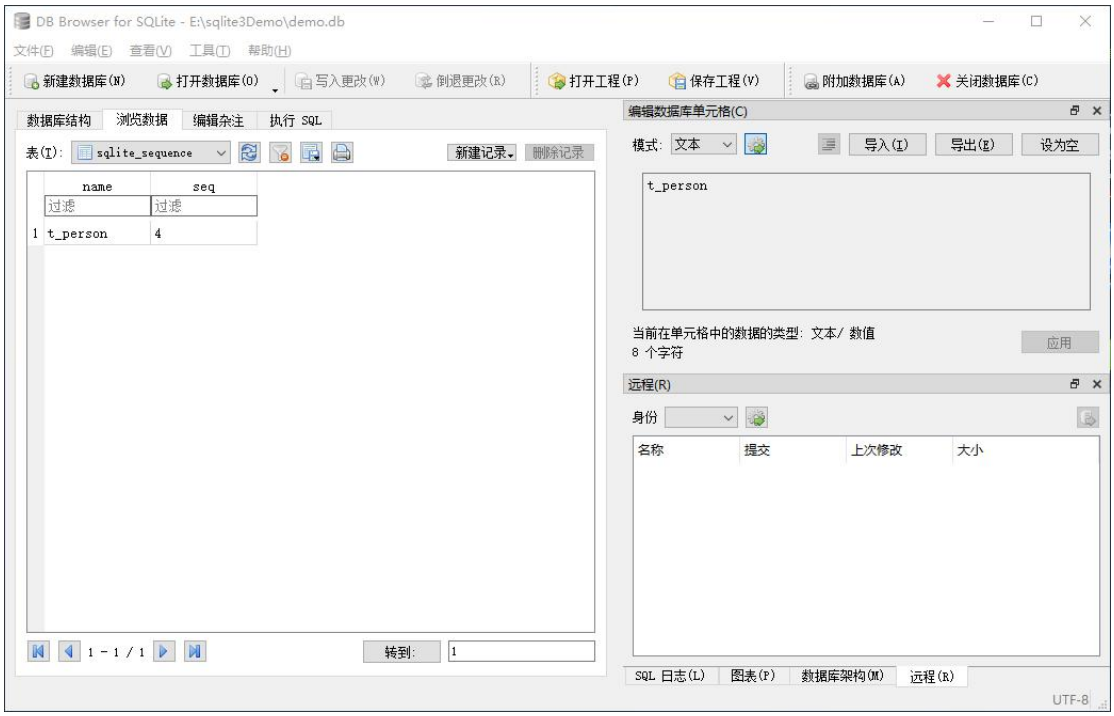


图 6-5 浏览表的记录

从上面的描述可以看出，DB Browser for SQLite 在操作上非常简便，读者只要稍加摸索就可以掌握任何其他的功能。

6.2 操作 SQLite 数据库

从 Python3.x 版本开始，在标准库中已经内置了 SQLlite3 模块，它可以支持 SQLite3 数

据库的访问和相关的数据库操作。在需要操作 SQLite3 数据库数据时，只须在程序中导入 SQLite3 模块即可。Python 语言操作 SQLite3 数据库的基本流程如下所示：

(1) 导入相关库或模块（SQLite3）。

(2) 使用 connect()连接数据库并获取数据库连接对象。它提供了以下方法：

.cursor() 方法来创建一个游标对象

.commit() 方法来处理事务提交

.rollback() 方法来处理事务回滚

.close() 方法来关闭一个数据库连接

(3) 使用 con.cursor()获取游标对象。

(4) 使用游标对象的方法(execute()、executemany()、fetchall()等)来操作数据库，实现插入、修改和删除操作，并查询获取显示相关的记录。在 Python 程序中，连接函数 sqlite3.connect()有如下两个常用参数。

database：表示要访问的数据库名。

timeout：表示访问数据的超时设定。

(5) 使用 close()关闭游标对象和数据库连接。数据库操作完成之后，必须及时调用其 close()方法关闭数据库连接，这样做的目的是减轻数据库服务器的压力。

6.2.1 使用 SQLite 创建表

使用 sqlite3 模块的 connect 方法来创建/打开数据库，需要指定数据库路径，不存在则创建一个新的数据库。


```
con=sqlite3.connect('e:/sqllitedb/first.db')
```

下面实例代码演示使用 SQLite3 创建数据库的过程。

【示例 6-1】使用 SQLite 创建表

```
#导入 sqllite3 模块
import sqlite3
# 1.硬盘上创建连接
con = sqlite3.connect('e:/sqlite3Demo/sqlite_test.db')
#  获取 cursor 对象
cur = con.cursor()
#  执行 sql 创建表
sql = 'create table t_person(pno INTEGER PRIMARY KEY  AUTOINCREMENT ,pname varchar(30) NOT NULL ,age INTEGER)'
```

```
try:
    cur.execute(sql)
except Exception as e:
    print(e)
    print('创建表失败')
finally:
    # 关闭游标
    cur.close()
    # 关闭连接
    con.close()
```

执行结果如图 6-6 所示：

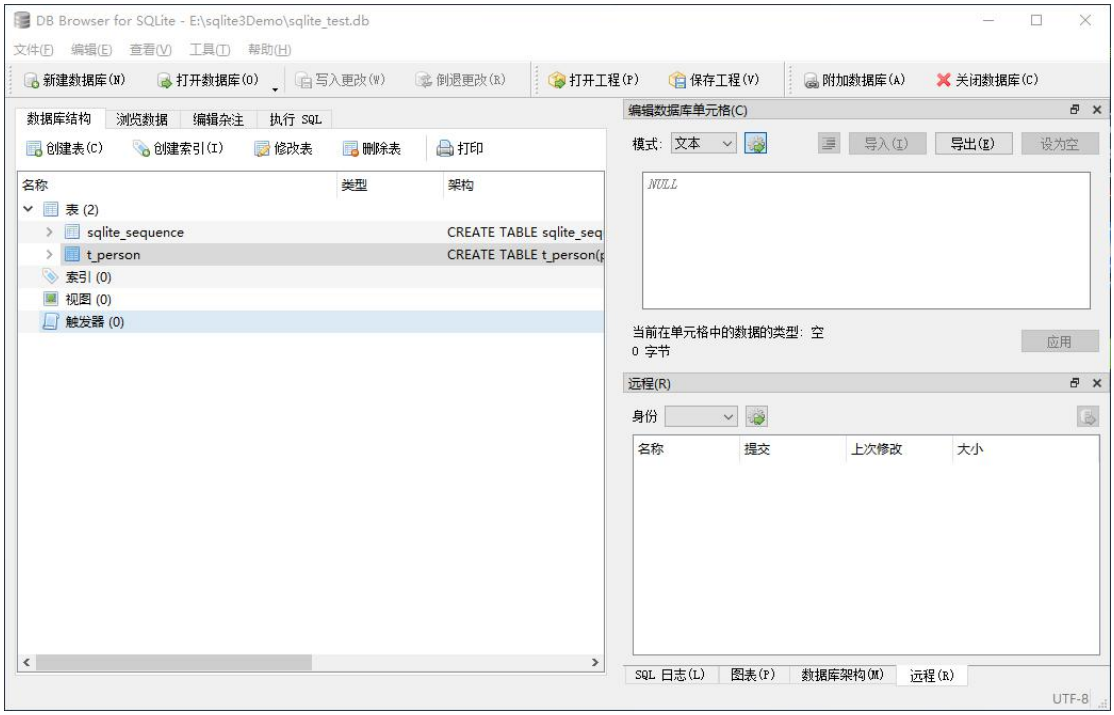


图 6-6 示例 6-1 执行结果

6.2.2 使用 SQLite 插入数据

调用游标对象的 execute 执行插入的 sql，使用 executemany()执行多条 sql 语句，使用 executmany()比循环使用 excute()执行多条 sql 语句效率高。

【示例 6-2】使用 SQLite 插入一条数据

```
#导入 sqllite3 模块
import sqlite3
# 1.硬盘上创建连接
con = sqlite3.connect('e:/sqlitedb/first.db')
#  获取 cursor 对象
cur = con.cursor()
#  执行 sql 创建表
sql = 'insert into t_person(pname,age) values(?,?)'
try:
    cur.execute(sql,('张三',23))
    #提交事务
    con.commit()
    print('插入成功')
except Exception as e:
    print(e)
    print('插入失败')
    con.rollback()
finally:
    #  关闭游标
    cur.close()
    #  关闭连接
    con.close()
```

执行结果如图 6-7 所示：

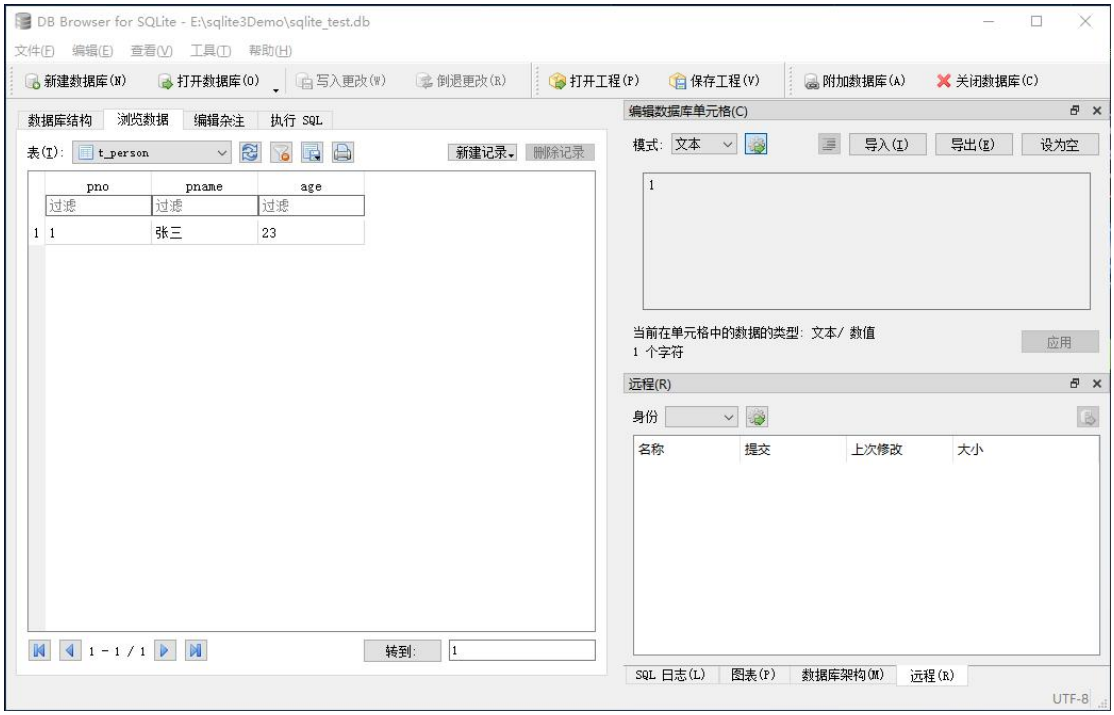


图 6-7 示例 6-2 执行结果

【示例 6-3】使用 SQLite 插入多条数据

```
#导入 sqllite3 模块
import sqlite3
# 1.硬盘上创建连接
con = sqlite3.connect('e:/sqlite3Demo/sqlite_test.db')
#  获取 cursor 对象
cur = con.cursor()
try:
    #执行 sql 创建表
    sql = 'insert into t_person(pname,age) values(?,?)'
    cur.executemany(sql, [('小张', 23), ('李四', 25), ('小红', 24), ('小李', 12)])
    #提交事务
    con.commit()
    print('插入成功')
except Exception as e:
    print('插入失败')
    con.rollback()
finally:
    # 关闭游标
```

```
    cur.close()
    # 关闭连接
    con.close()
```

执行结果如图 6-8 所示：

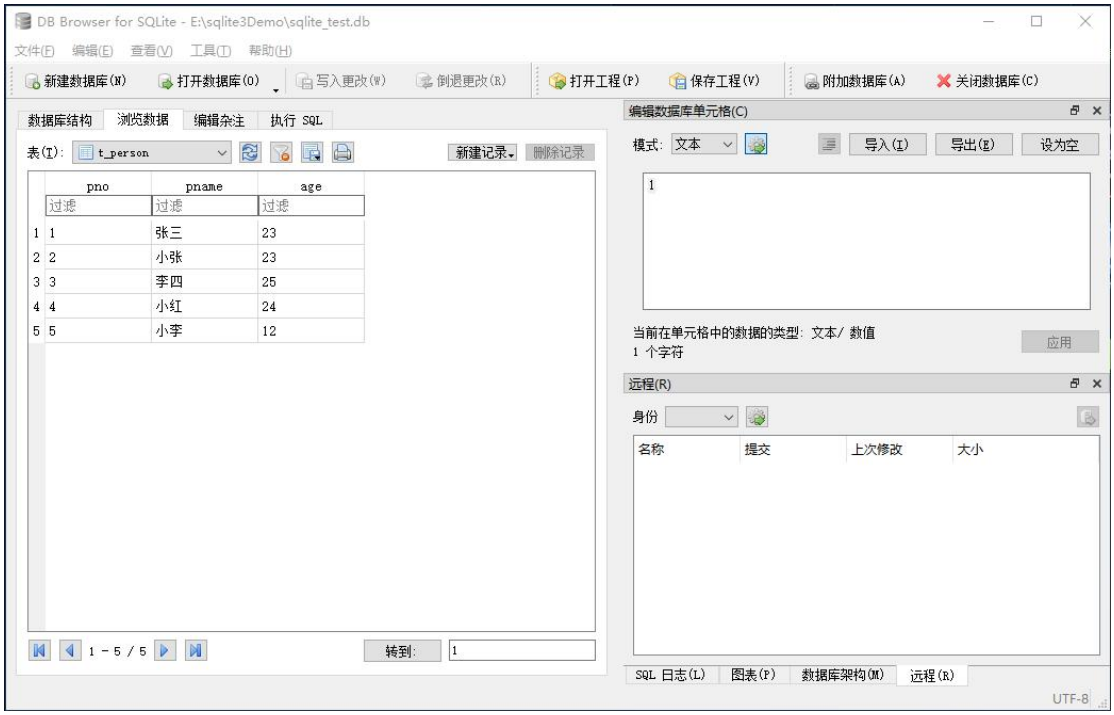


图 6-8 示例 6-3 执行结果

6.2.3 使用 SQLite 查询数据

查询数据，游标对象提供了 fetchall() 和 fetchone() 方法 。fetchall()方法获取所有数据，返回一个列表。fetchone()方法获取其中一个结果，返回一个元组。

【示例 6-4】SQLite 使用 fetchall()查询所有数据

```
#导入 sqllite3 模块
import sqlite3
# 1.硬盘上创建连接
con = sqlite3.connect('e:/sqlitedb/first.db')
#  获取 cursor 对象
cur = con.cursor()
#  执行 sql 创建表
```

```
sql = 'select * from t_person'
try:
    cur.execute(sql)
    # 获取所有数据
    person_all = cur.fetchall()
    # print(person_all)
    # 遍历
    for p in person_all:
        print(p)
except Exception as e:
    print(e)
    print('查询失败')
finally:
    # 关闭游标
    cur.close()
    # 关闭连接
    con.close()
```

执行结果如图 6-9 所示：

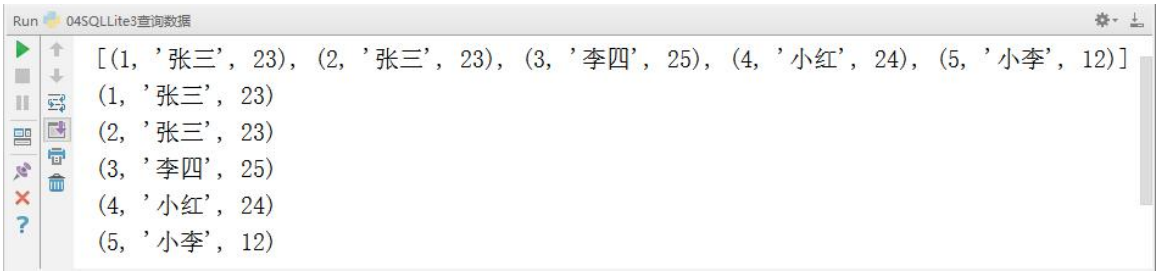


图 6-9 示例 6-4 执行结果

【示例 6-5】SQLite 使用 fetchone()查询一条数据

```
#导入 sqllite3 模块
import sqlite3
# 1.硬盘上创建连接
con = sqlite3.connect('e:/sqlitedb/first.db')
# 获取 cursor 对象
cur = con.cursor()
# 执行 sql 创建表
sql = 'select * from t_person'
```

```
try:
    cur.execute(sql)
    # 获取一条数据
    person = cur.fetchone()
    print(person)
except Exception as e:
    print(e)
    print('查询失败')
finally:
    # 关闭游标
    cur.close()
    # 关闭连接
    con.close()
```

执行结果如图 6-10 所示：

图 6-10 示例 6-5 执行结果

【示例 6-6】SQLite 修改数据

```
#导入 sqllite3 模块
import sqlite3
#1.硬盘上创建连接
con=sqlite3.connect('e:/sqlitedb/first.db')
#获取 cursor 对象
cur=con.cursor()
try:
    #执行 sql 创建表
    update_sql = 'update t_person set pname=? where pno=?'
    cur.execute(update_sql, ('小明', 1))
    #提交事务
    con.commit()
    print('修改成功')
except Exception as e:
    print(e)
```

```
        print('修改失败')
        con.rollback()
finally:
        # 关闭游标
        cur.close()
        # 关闭连接
        con.close()
```

执行结果如图 6-11：

图 6-11 示例 6-6 执行结果

【示例 6-7】SQLite 删除数据

```
#导入 sqllite3 模块
import sqlite3
#1.硬盘上创建连接
con=sqlite3.connect('e:/sqlitedb/first.db')
#获取 cursor 对象
cur=con.cursor()
#执行 sql 创建表
delete_sql = 'delete from t_person where pno=?'
try:
        cur.execute(delete_sql, (2,))
        #提交事务
        con.commit()
        print('删除成功')
except Exception as e:
        print(e)
        print('删除失败')
        con.rollback()
finally:
        # 关闭游标
        cur.close()
```

```
    # 关闭连接
    con.close()
```

执行结果如图 6-12：

图 6-12 示例 6-7 执行结果

在上述实例代码中，首先定义查询所有数据、插入数据、修改数据、删除数据的方法。然后，定义主方法中依次建立连接，获取连接的 cursor，通过 cursor 的 execute()等方法来执行 SQL 语句，调用插入记录、更新记录、删除记录的方法。

6.3 MySQL 数据库

MySQL 是非常常用的关系型数据库，现在很多互联网应用都使用了 MySQL 数据库。在使用之前先需要下载 MySQL 数据库。

6.3.1 下载 MySQL

如果大家安装 MySQL 只是为了个人的学习和软件开发，那么安装免费的社区版即可。首先进入 MySQL 的官网：https://www.mysql.com/，如图 6-13 所示：

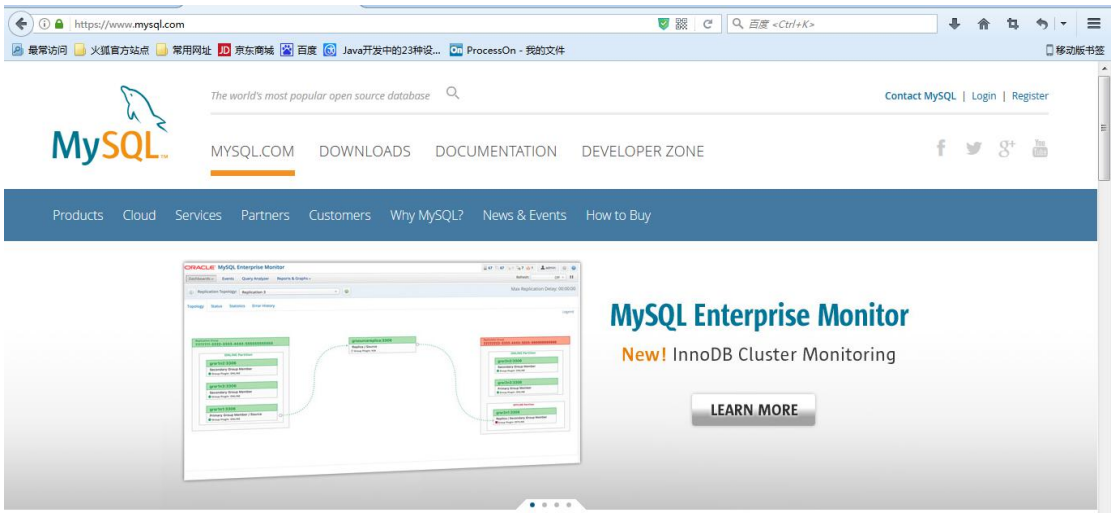


图 6-13 MySQL 的官网

然后点击 DOWNLOADS 导航栏，就会默认进入到 MySQL 的 Enterprise（企业版）产

品下载页面，所以还需要我们点击 Community（社区版），切换到社区版的下载页面，最后点击 MySQL Community Server 下边的 DOWNLOAD 按钮即可进入 MySQL 数据库的下载页面。操作如图 6-14 所示：

图 6-14 MySQL 社区版产品下载页面

进入 MySQL 的数据库下载界面后，首先在“Select Operating System”下拉菜单中选择“Microsoft Windows”平台，然后进入 MySQL Installer MSI 下载页面，如图 6-15 所示：

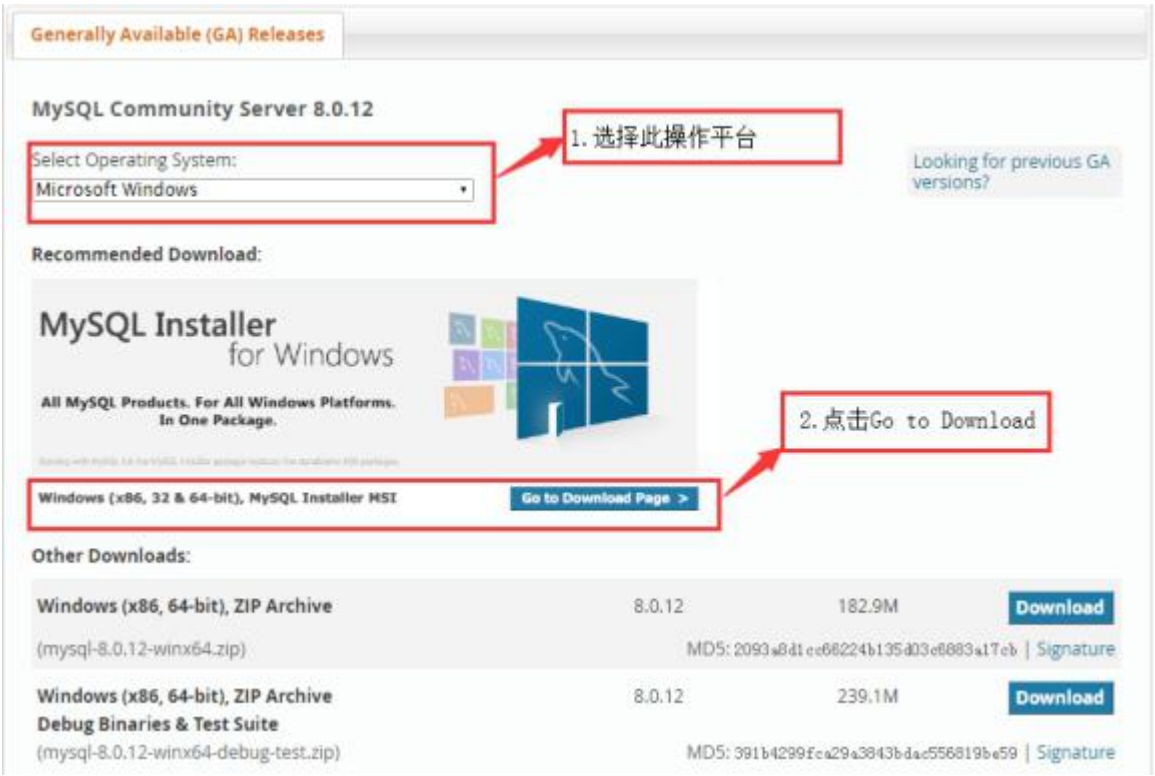


图 6-15 Windows 平台下的 MySQL 数据库产品页面

在 MSI 下载页面，按照图 6-16 所示，选择正确的文件下载，此时 MySQL 官网会建议你注册或者登陆账号然后下载，当然我们也可以选择“No thanks, just start my download.”直接下载。

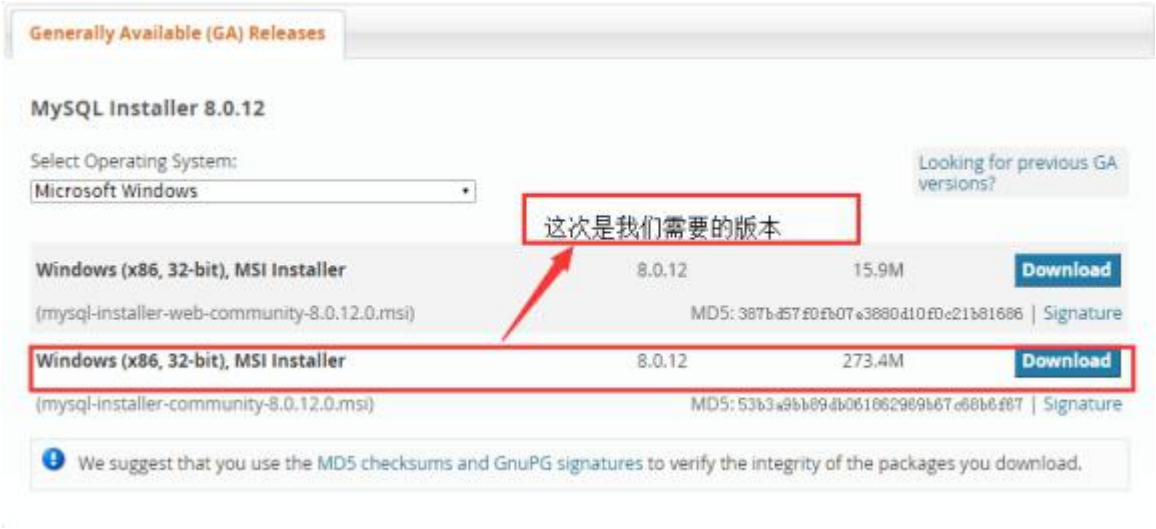


图 6-16 Windows 平台下的 MSI 版 MySQL 下载页面

6.3.2 安装 MySQL

根据下载路径找到下载好的 MySQL 安装程序

（mysql-installer-community-8.0.12.0.msi），具体步骤如下所示：

1) 双击安装程序 mysql-installer-community-8.0.12.0.msi，此时会弹出 MySQL 许可协议界面，如图 6-17 所示。单击选中复选框“I accept the license terms”后，点击“Next”按钮，进入安装类型选择界面。

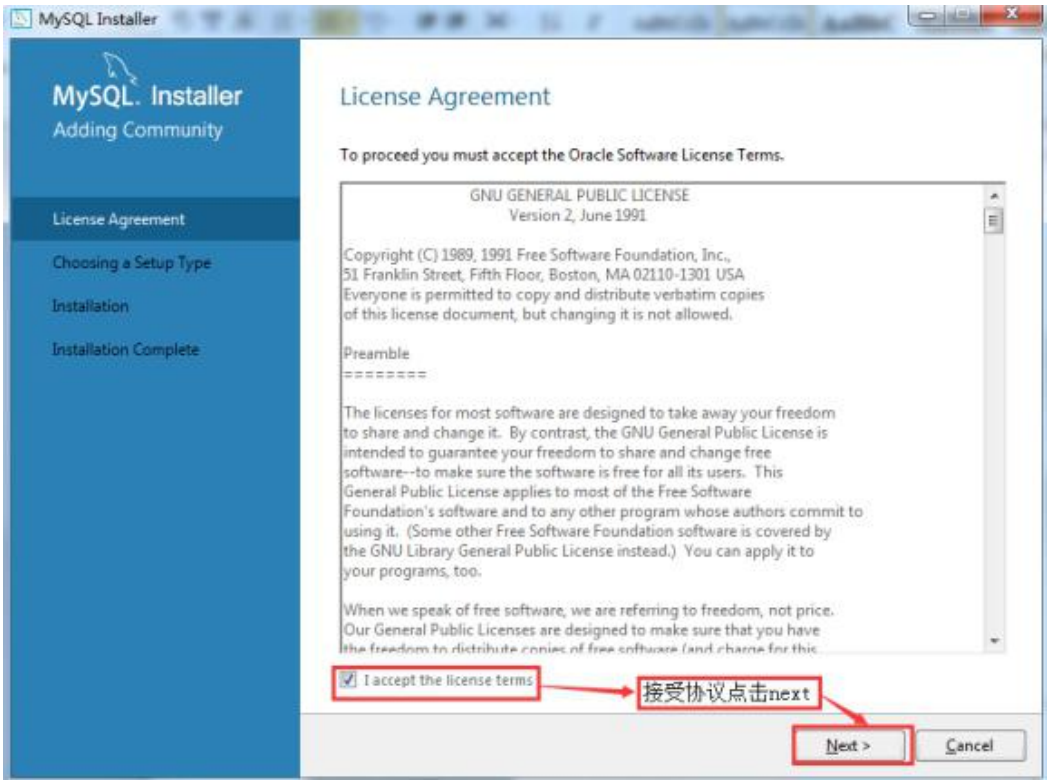


图 6-17 许可协议界面

2) 选择自定义安装类型“Custom”（此类型可以根据用户自己的需求选择安装需要的产品），然后单击“Next”按钮，如图 6-18 所示：

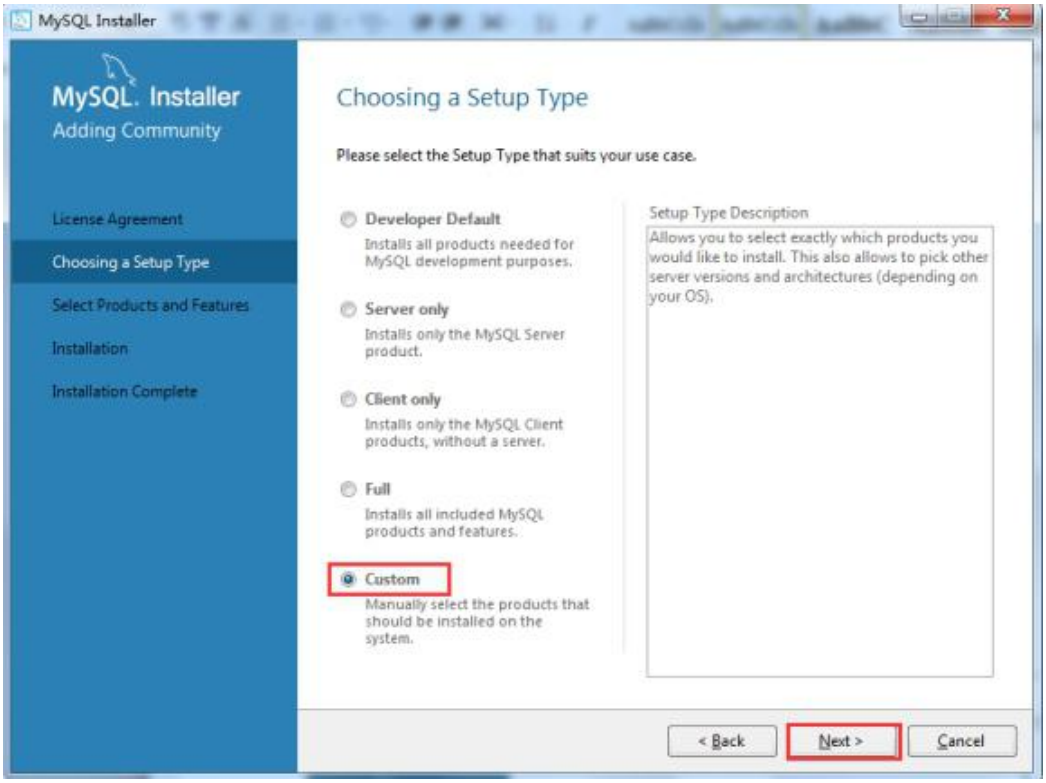


图 6-18 安装类型选择界面

3) 在选择安装版本界面，展开第一个节点“MySQL Servers”，找到并点击“MySQL Server

8.0.12-X64”，之后向右的箭头会变成绿色，，如图 6-19 所示。点击该绿色的箭头，将选中的产品添加到右边的待安装列表框中，然后在展开安装列表中的 MySQL Server 8.0.12-X64 节点，取消“Development Components”选项前边的“√”，然后点击“Next”按钮进入安装列表界面，如图 6-20 所示：

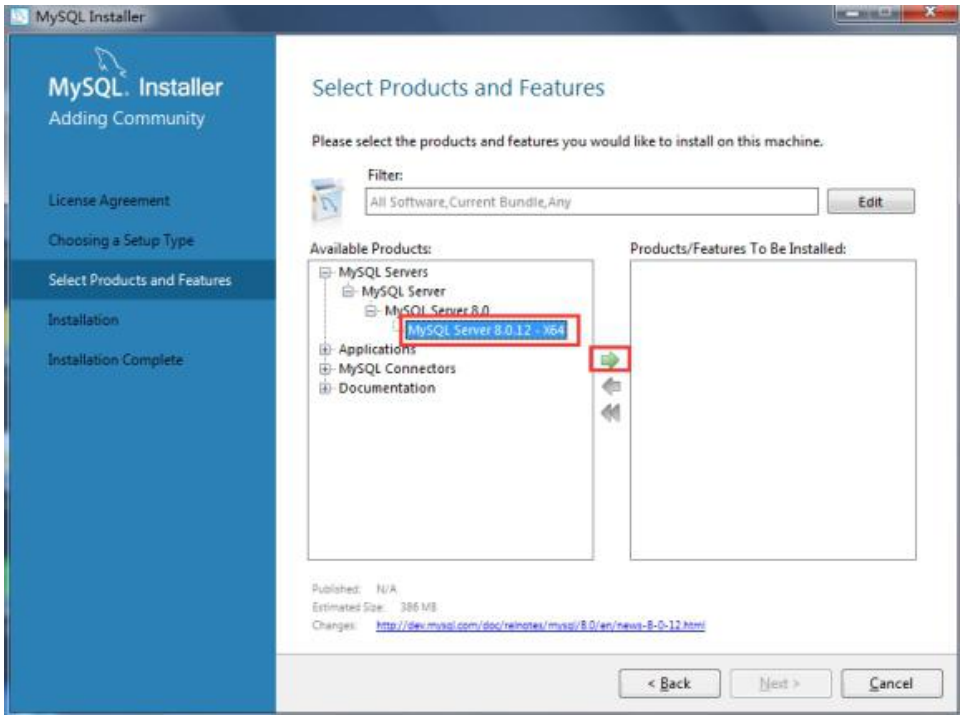


图 6-19 选择安装版本界面

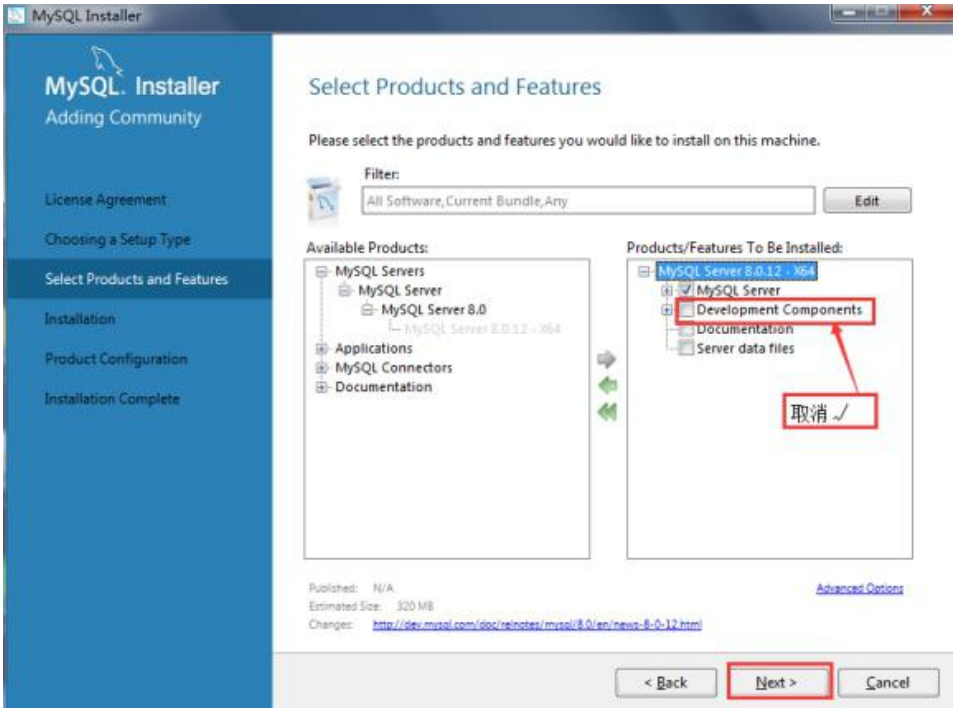


图 6-20 添加要安装的产品界面

4) 点击安装列表界面的“Execute”按钮后，要安装的产品右边会显示一个进度百分比，安装完成之后会前边会出现绿色个的“√”，如图 6-21 所示。之后继续点击“Next”按钮即可。

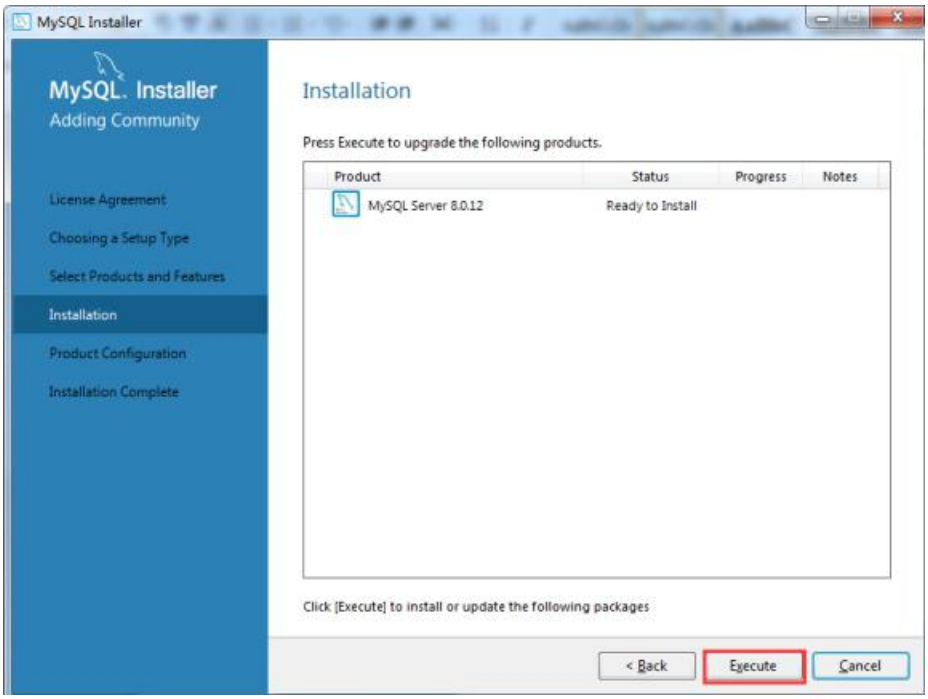


图 6-21 安装列表界面

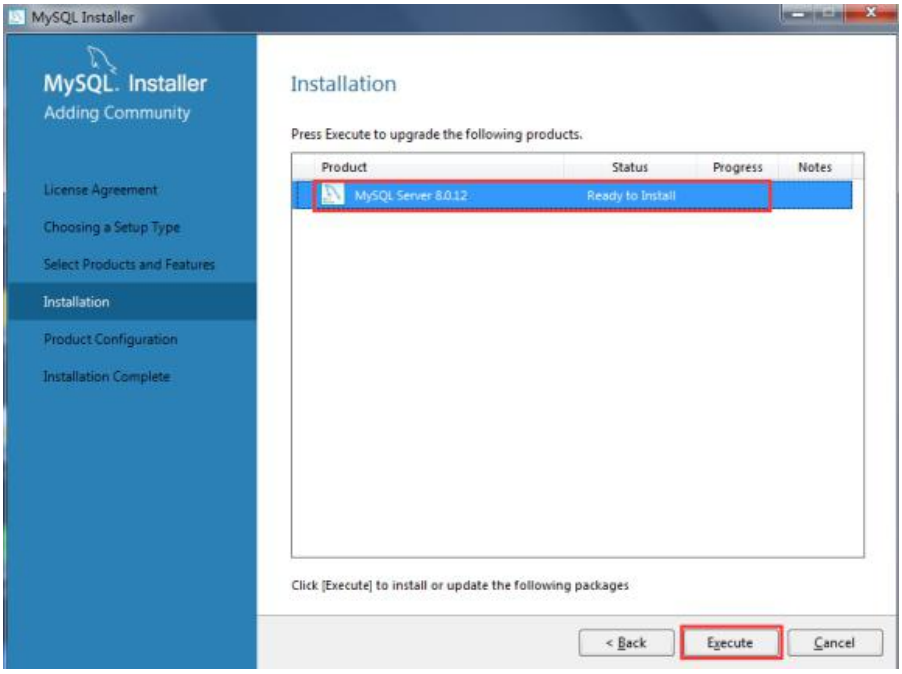


图 6-22 安装成功界面

完成上述 4 个步骤后，MySQL 终于安装成功了，剩下的就是对其进行配置，我们将在下一小节中讲述。

6.3.3 配置 MySQL

安装完成后，还需要设置 MySQL 的各项参数才能正常使用。仍然使用图形化界面对其进行配置，具体步骤如下所示：

1) 直接点击如图 6-23 中的“Next”按钮，直接进入参数配置页面中的“Type and NetWorking”界面。

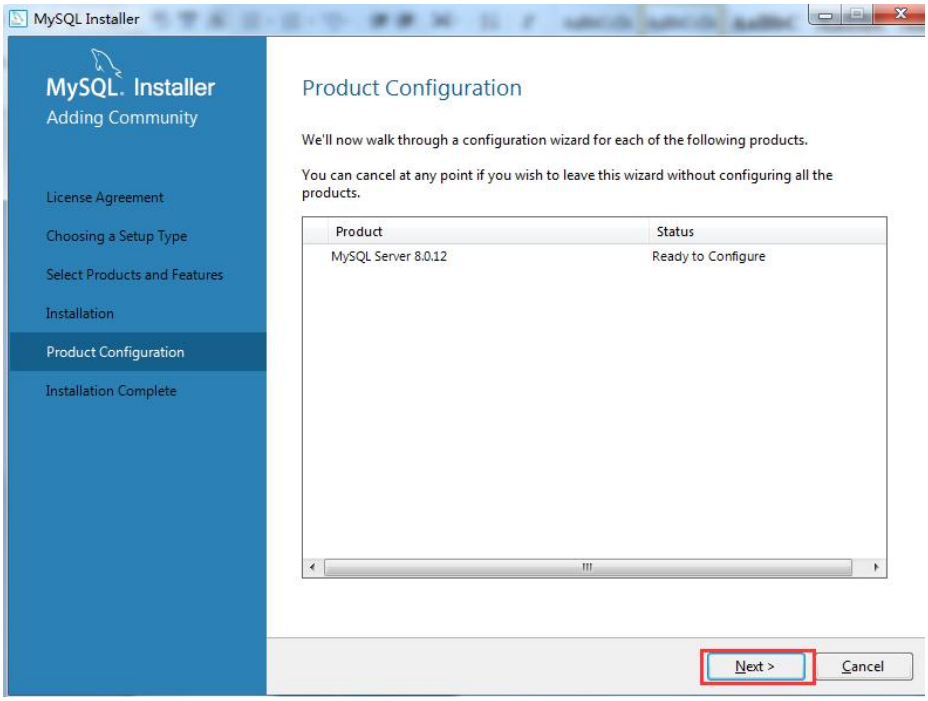


图 6-23 安装成功界面

2) 进入“Type and Networking”界面后，会看到两个选项“Standalone MySQL Server / Classic MySQL Replication”和“InnoDB Cluster Sandbox Test Setup（for testing only）”。

如果要运行独立的 MySQL 服务器可以选择前者，以便稍后配置经典的 MySQL 复制，使用该选项，用户可以手动配置复制设置，并在需要时提供自己的高可用性解决方案。

而后者是 InnoDB 集群沙箱测试设置，仅用于测试。

我们要选择的是“Standalone MySQL Server / Classic MySQL Replication”选项，然后点击“Next”按钮即可，如图 6-24 所示：

3) 如图 6-24 所示：服务器配置类型“Config Type”选择“Development machine”，不同的选择将决定系统为 MySQL 服务器实例分配资源的大小，“Development machine”占用的内存是最少的；连接方式保持默认的 TCP/IP，端口号也保持默认的 3306 即可；点击“Next”按钮。

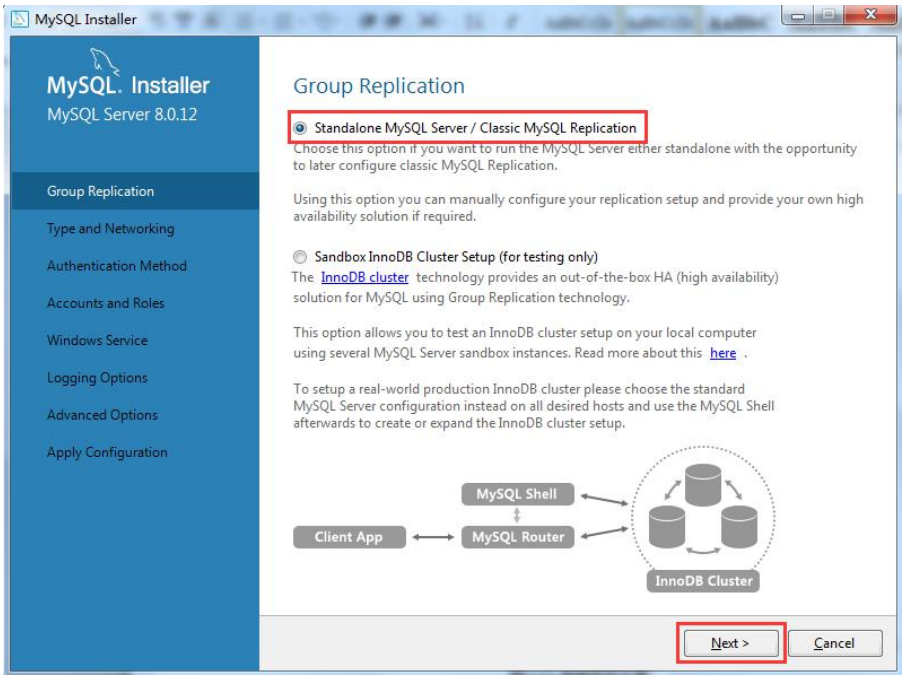


图 6-24 类型选择界面

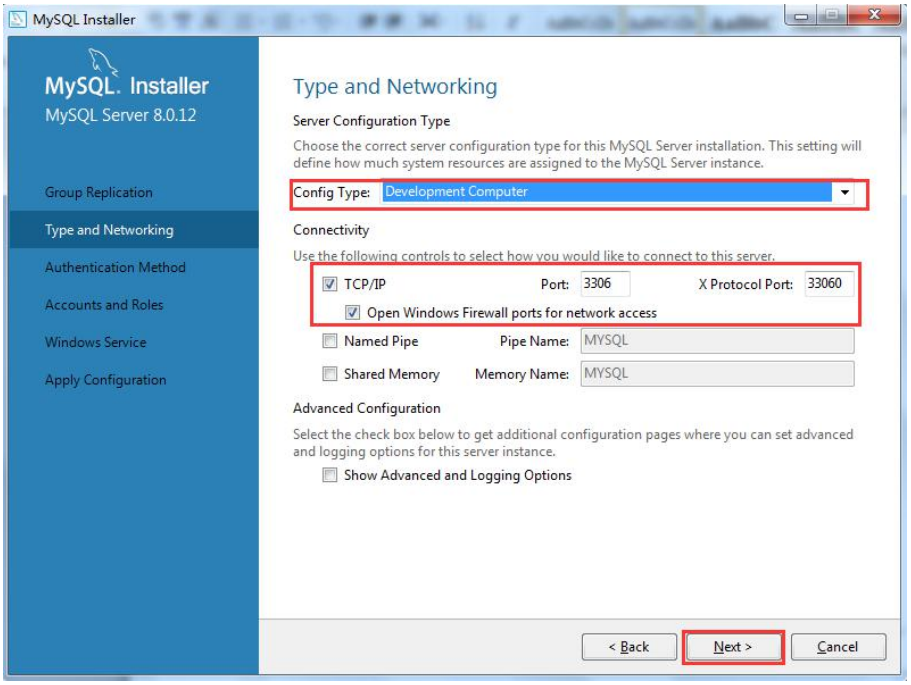


图 6-25 类型及网络参数配置界面

4) 接下来就是设置 MySQL 数据库 Root 账户密码，需要输入两遍。这个密码必须记住，后边会用到。此处我们将密码设置成“bjsxt”，之后点击“Next”按钮，如图 6-26 所示：

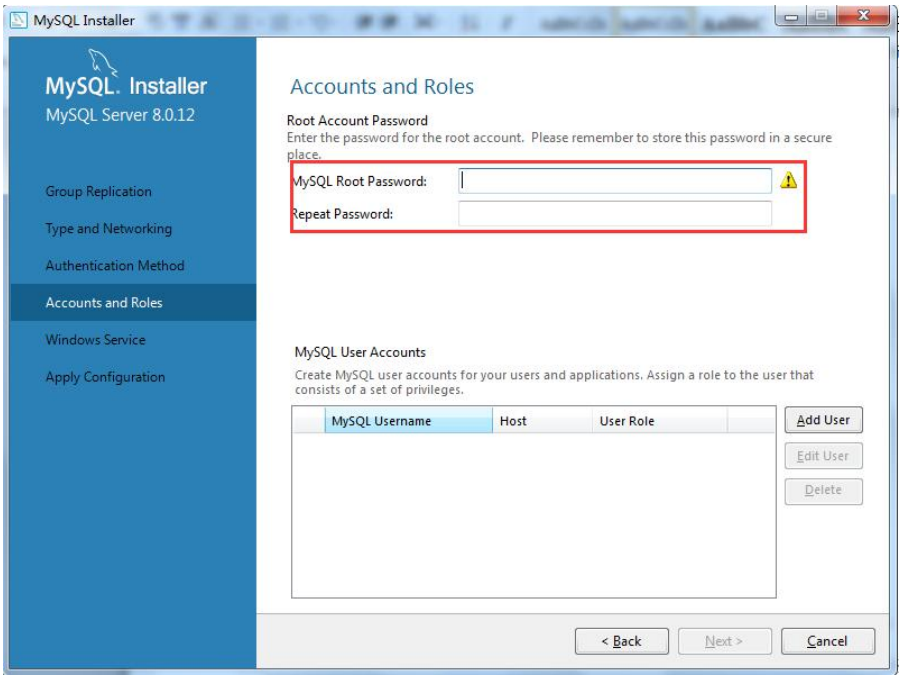


图 6-26 设置 Root 账户的密码界面

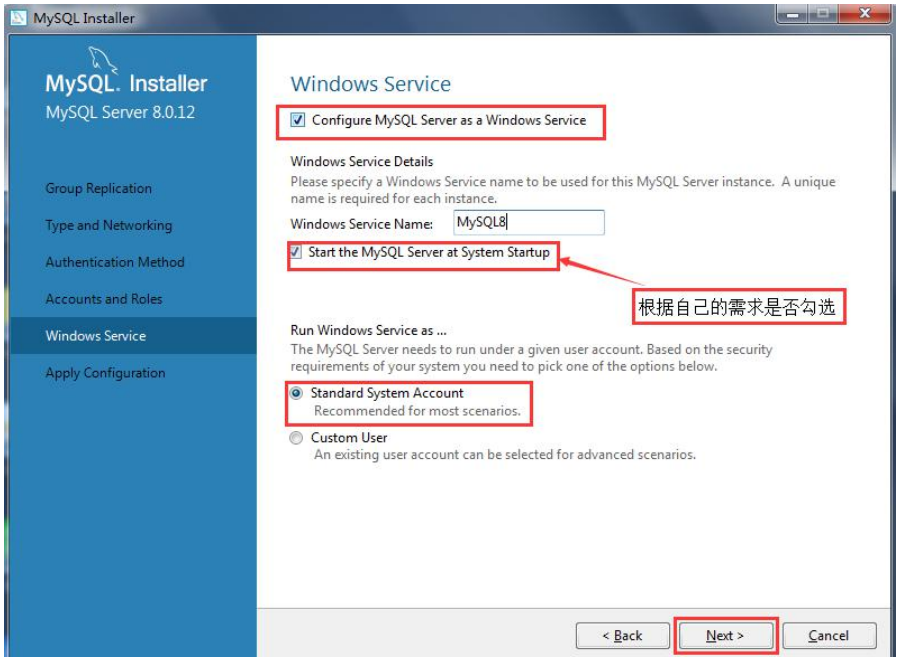


图 6-27 设置 Windows 服务界面

5) 在配置 Windows 服务时，需要以下几部操作：勾选“Configure MySQL Server as a Windows Service”选项，将 MySQL 服务器配置为 Windows 服务；取消“Start the MySQL Server at System Startup”选项前边的“√”（该选项是设置是否开机自启动 MySQL 服务，在此我们选择开机不启动，大家也可以根据自己的需要来选择）；勾选“Standard System Account”选项，该选项是标准系统账户，推荐使用该账户；点击“Next”按钮。如图 6-27 所示：

6) 下面就是准备执行上述一系列配置的时候了，直接点击“Execute”按钮。等到所有

的配置完成之后，会出现如图 6-28 所示的界面，点击“Finish”按钮，就会跳到配置成功界面，之后点击界面的“Next”按钮，再弹出的界面中点击“Finish”按钮即可完成配置。

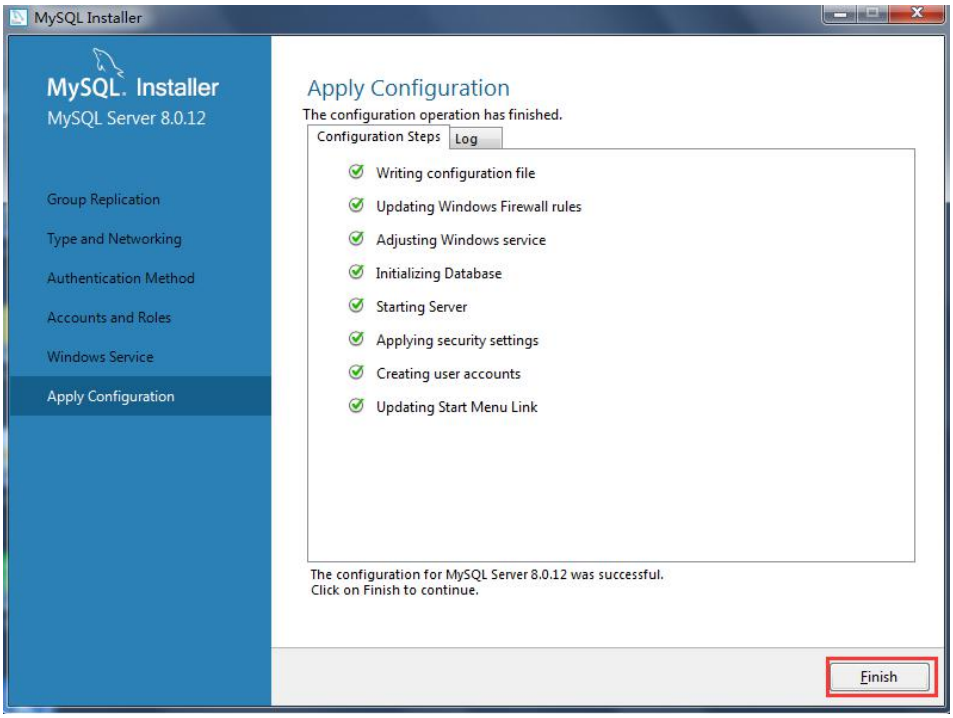


图 6-28 配置成功界面

6.4 操作 MySQL 数据库

PyMySQL 是在 Python3.x 版本中用于连接 MySQL 服务器的一个库，Python2 中则使用 mysqldb。

6.4.1 搭建 PyMySQL 环境

在使用 PyMySQL 之前，我们需要确保 PyMySQL 已安装。如果还未安装，使用以下命令安装最新版的 PyMySQL。


```
pip  install PyMySQL
```

如果使用命令无法安装，需要下载 PyMySQL-0.9.3-py2.py3-none-any.whl 文件，进行安装。

（1）进入 python 官网 https://www.python.org 点击菜单 PyPI ，如图 6-29 所示：

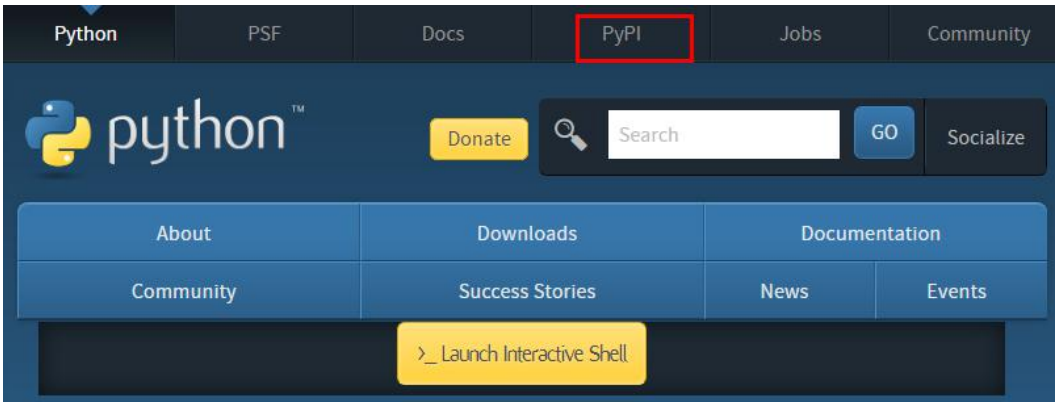


图 6-29 python 官网

（2）输入 pymsql，进行搜索。如图 6-30 所示：

图 6-30 搜索 pymysql

（3）点击 PyMySQL0.9.3，直接点击左侧 Download files 进行下载，如图 6-31 所示：

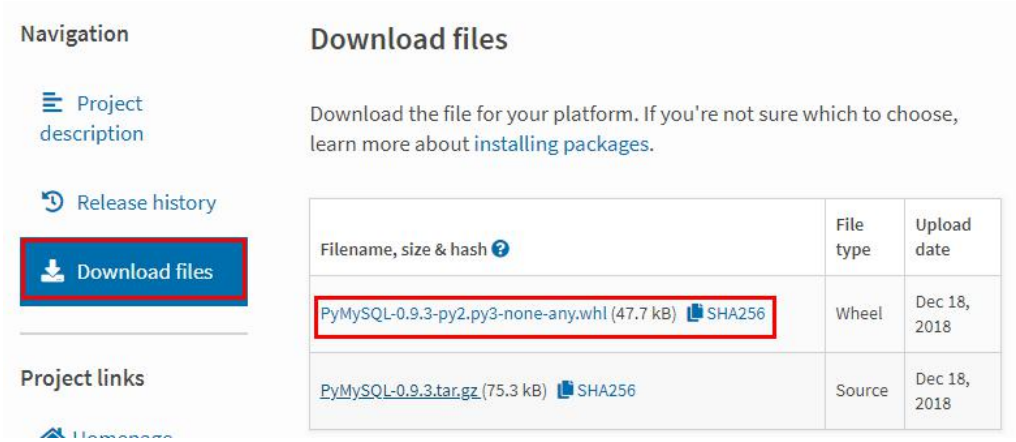


图 6-31 下载 pymysql

（4）windows+R 打开 doc 窗口，进入 PyMySQL-0.9.3-py2.py3-none-any.whl 文件所在目录，执行如命令进行安装。

```
pip install PyMySQL-0.9.3-py2.py3-none-any.whl
```

6.4.2 创建数据库表

pymysql 模块中提供的 API 与 sqlite3 模块中提供的 API 类似，不必详细研究 Python DB 规范，只需要记住几个函数和方法，绝大多数的数据库的操作就可以搞定。

- connect 函数：连接数据库，根据连接的数据库类型不同，该函数的参数也不同。connect 函数返回 Connection 对象。
- cursor 方法：获取操作数据库的 Cursor 对象。cursor 方法属于 Connection 对象。
- execute 方法：用于执行 SQL 语句，该方法属于 Cursor 对象。
- commit 方法：在修改数据库后，需要调用该方法提交对数据库的修改，commit 方法属于 Cursor 对象。
- rollback 方法：如果修改数据库失败，一般需要调用该方法进行数据库回滚，也就是将数据库恢复成修改之前的样子。

在 Python 程序中，可以使用 execute()在数据库中创建一个新表。下面的实例代码演示了在 PyMySQL 数据库中创建新表 student 的过程。

【示例 6-8】使用 MySQL 创建表 student

```
import pymysql
try:
    #创建与数据库的连接
    db=pymysql.connect('localhost','root','root','testdb')
    #创建游标对象 cursor
    cursor=db.cursor()
    #使用 execute()方法执行 sql，如果表存在则删除
    cursor.execute('drop table if EXISTS student')
    #创建表的 sql
    sql='''
        create table student(
        sno int(8) primary key auto_increment,
        sname varchar(30) not null,
        sex varchar(5) ,
        age int(2),
        score float(3,1)
        )
```

```
    '''
    cursor.execute(sql)
except:
    print('创建表失败')
finally:
    #关闭数据库连接
    db.close()
```

执行结果如图 6-32 所示：

图 6-32 示例 6-8 执行结果

6.4.3 数据库插入操作

在 Python 程序中，可以使用 SQL 语句向数据库中插入新的数据信息。

【示例 6-9】使用 MySQL 插入数据

```
import pymysql
#创建与数据库的连接
db=pymysql.connect('localhost','root','root','testdb')
#创建游标对象 cursor
cursor=db.cursor()
#插入 sql 语句
sql='''
    insert into student(sname,sex,age,score) values(%s,%s,%s,%s)
'''
try:
    #执行 sql 语句
```

```
        cursor.execute(sql,('李四','woman',25,99.6))
        #提交事务
        db.commit()
        print('插入成功')
except Exception as e:
        print(e)
        #如果出现异常，回滚
        db.rollback()
        print('插入失败')
finally:
        #关闭数据库连接
        db.close()
```

执行结果如图 6-33 所示：

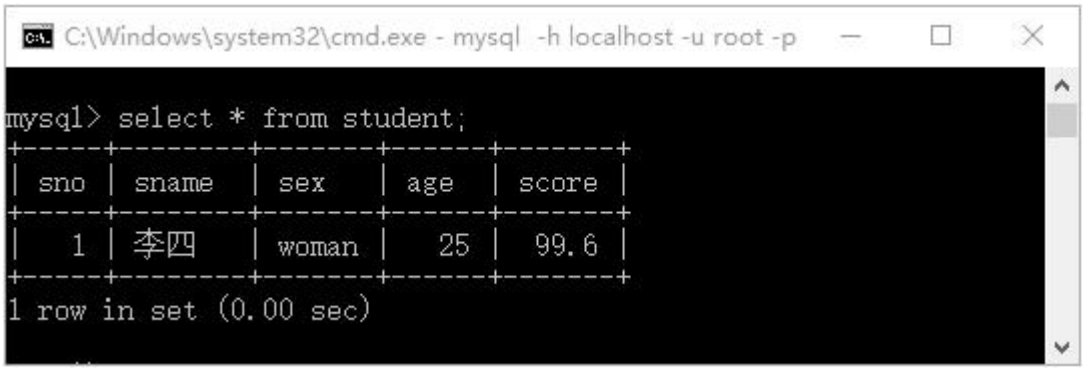


图 6-33 示例 6-9 执行结果

【示例 6-10】使用 MySQL 插入多条数据

```
import pymysql
#创建与数据库的连接
db=pymysql.connect('localhost','root','root','testdb')
#创建游标对象 cursor
cursor=db.cursor()
#插入 sql 语句
sql='''
        insert into student(sname,sex,age,score) values(%s,%s,%s,%s)
'''
args=[('王五','woman',22,98.6),('赵六','man',21,99.1)]
try:
        #执行 sql 语句
```

```
        cursor.executemany(sql,args)
        #提交事务
        db.commit()
        print('插入成功')
except Exception as e:
        print(e)
        #如果出现异常，回滚
        db.rollback()
        print('插入失败')
finally:
        #关闭数据库连接
        db.close()
```

执行结果如图 6-34 所示：

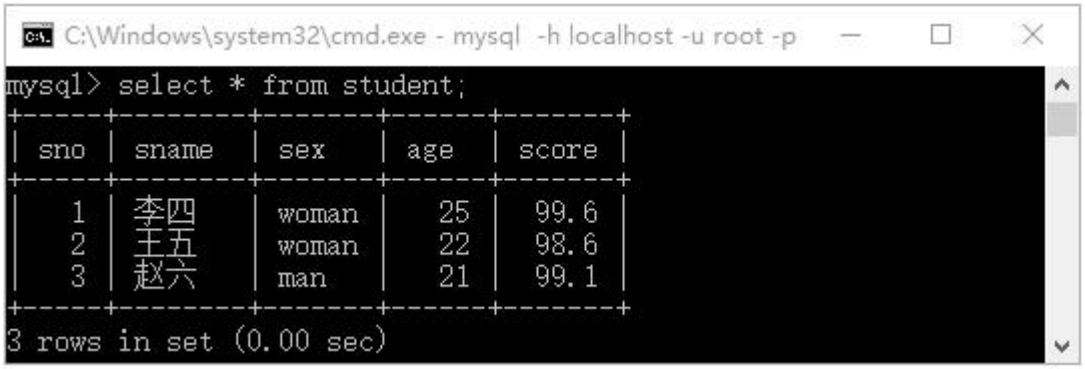


图 6-34 示例 6-10 执行结果

6.4.4 数据库查询操作

Python 查询 Mysql 使用 fetchone() 方法获取单条数据使用 fetchall() 方法获取多条数据。

fetchone(): 该方法获取下一个查询结果集。结果集是一个对象

fetchall(): 接收全部的返回结果行.

rowcount: 这是一个只读属性，并返回执行 execute()方法后影响的行数。

【示例 6-11】使用 MySQL 查询学生 年龄大于等于 23 的所有学生信息

```
import pymysql
#创建与数据库的连接
```

```
db=pymysql.connect('localhost','root','root','testdb')
#创建游标对象 cursor
cursor=db.cursor()
#查询年龄大于等于 23 的所有学生信息
sql='select * from student where age>=23'
try:
    #执行 sql
    cursor.execute(sql)
    #获取查询结果
    results=cursor.fetchall()
    for row in results:
        sno=row[0]
        sname=row[1]
        sex=row[2]
        age=row[3]
        score=row[4]
        #输出
        print('sno:',sno,'sname:',sname,'sex:',sex,'age:',age,'score:',score)
except Exception as e:
    print(e)
    print('查询失败')
finally:
    db.close()
```

执行结果如图 6-35 所示：

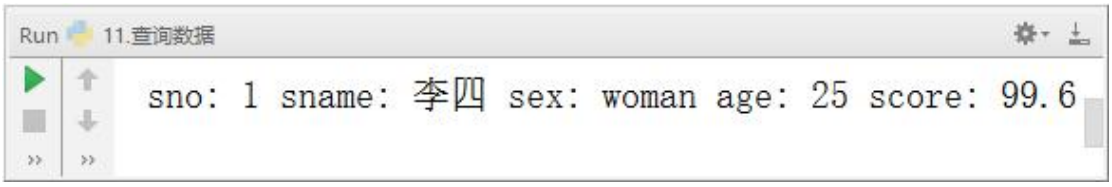


图 6-35 示例 6-11 执行结果

6.4.5 数据库更新操作

在 Python 程序中，可以使用 update 语句更新数据库中数据信息。

【示例 6-12】使用 MySQL 更新数据库中的数据

```
import pymysql
#创建与数据库的连接
db=pymysql.connect('localhost','root','root','testdb')
#创建游标对象 cursor
cursor=db.cursor()
#将 sno=2 的学生成绩修改为 99.5
sql='update student set score=%s where sno=%s'
try:
    #执行 sql
    cursor.execute(sql,(99.5,2))
    #提交数据
    db.commit()
    print('修改成功')
except:
    print('修改失败')
    db.rollback()
finally:
    db.close()
```

执行结果如图 6-36 所示：

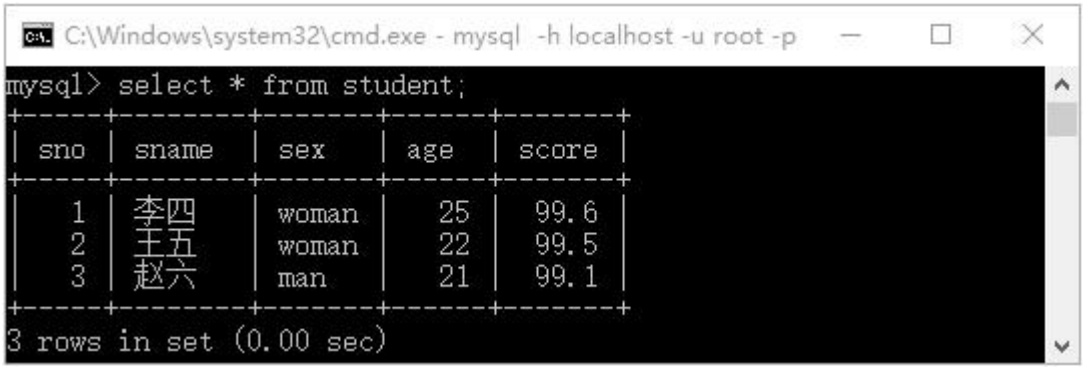


图 6-36 示例 6-12 执行结果

6.4.6 数据库删除操作

在 Python 程序中，可以使用 delete 语句删除数据库中的数据信息

【示例 6-13】使用 MySQL 删除年龄小于 22 的学生

```
import pymysql
#创建与数据库的连接
```

```
db=pymysql.connect('localhost','root','root','testdb')
#创建游标对象 cursor
cursor=db.cursor()
#删除 sql
sql='delete from student where age < 22'
try:
    #执行 sql 语句
    cursor.execute(sql)
    #提交事务
    db.commit()
    print('删除数据成功')
except:
    db.rollback()
    print('删除数据失败')
finally:
    #关闭连接
    db.close()
```

执行结果如图 6-37 所示：

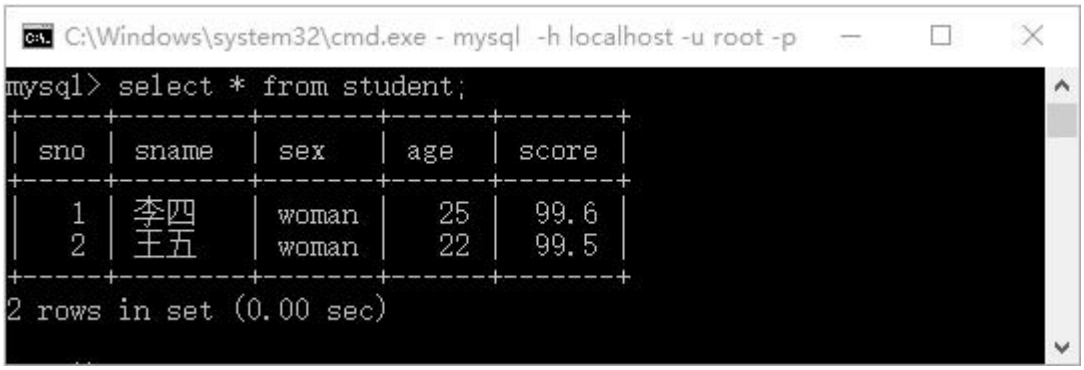


图 6-37 示例 6-13 执行结果

习题

一、选择题

1. 能否对 SQLite 中的表建索引（）

A. 不能，但系统会根据查询情况自动创建

B. 不能

C. 能

2. Mysql 安装模式有（）（多选）

A. 典型安装

B. 完全安装

C. 自定义安装

3. MySQL 中定义 DECIMAL 类型的数，如果不声明精度和标度，则系统按一下哪个选项进行显示？（）

A. DECIMAL(13,0)

B. DECIMAL(10,0)

C. DECIMAL(11,0)

D. DECIMAL(12,0)

4. 通过 SQL 语句，如何往表“Persons”插入一条新纪录？（）

A. INSERT ('Jimmy', 'Jackson') INTO Persons

B. INSERT INTO Persons VALUES ('Jimmy', 'Jackson')

C. INSERT VALUES ('Jimmy', 'Jackson') INTO Persons

D. INSERT ('Jimmy', 'Jackson') INTO Persons Values

5. 在 MySQL 中输入正确的命令后可以显示某张表的属性及其类型，如果一个数是整型 INT，常显示为 INT(11)，请问 11 代表（）

A. 在数据库中的实际存储大小

B. 在数据库中的存储宽度

C. 显示宽度

二、解答题

1. SQlite 中查看表数据的命令是什么？
2. SQlite 中 SQL 语句 distinct 如何使用，举例说明？
3. SQlite 中 SQL 语句 IN 如何使用，举例说明？
4. MySQL 数据库中聚合函数有哪些？
5. MySQL 数据库有哪些特点。

三、编码题

1．设有职工基本表：EMP（ENO，ENAME，AGE，SEX，SALARY），其属性分别表示职工号、姓名、年龄、性别、工资。使用 SQLite3 完成如下 SQL 语句：

1) 创建职工基本表
2) 为每个工资低于 1000 元的女职工加薪 200 元

2．编写一个程序，使用 MySQL 完成如下 SQL 语句：

1) 1)创建学生表、课程表和成绩表。设有学生－课程数据库，其数据库模式为：学生 Student （学号 Sno ，姓名 Sname ，学生性别 Ssex ，学生出生年月 Sbirthday，所在班级 Class ）、课程 Course（课程号 Cno ，课程名 Cname ）、成绩表 Score （学号 Sno ，课程号 Cno ，成绩 Degree ）；
2) 向三张表中分别插入数据 ；
3) 查询学生表所有信息；
4) 查询 Score 表中成绩在 60 到 80 之间的所有记录；
5) 查询 Score 表中成绩为 85，86 或 88 的记录。。

第七章 协程和异步 IO

在前面从多进程、多线程方面介绍了并发编程。其实多进程和多线程已经解决了异步 IO 的问题。但是它们都存在一个限制，就是线程或进程的数量在一个操作系统中是有限的，同时它们切换也要消耗一定的资源。要知道很多时候我们要并发的代码块其实很小，比如说只是 get 一个网页，因此，人们会把这个代码块进一步的从线程中脱离出来，这就是协程，它更小巧，它是程序级别的由程序根据需要自己调度。无需线程上下文切换的开销，协程避免了无意义的调度，由此可以提高性能。

通过阅读本章，你可以：

- 了解协程的概念
- 了解协程的优缺点
- 掌握如何定义一个协程
- 掌握协程阻塞、await 的使用
- 了解并发和并行的概念

7.1 协程的概念

协程，又称微线程，纤程。英文名 Coroutine，是一种用户态的轻量级线程。

子程序，或者称为函数，在所有语言中都是层级调用，比如 A 调用 B，B 在执行过程中又调用了 C，C 执行完毕返回，B 执行完毕返回，最后是 A 执行完毕。所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

线程是系统级别的它们由操作系统调度，而协程则是程序级别的由程序根据需要自己调度。在一个线程中会有很多函数，我们把这些函数称为子程序，在子程序执行过程中可以中断去执行别的子程序，而别的子程序也可以中断回来继续执行之前的子程序，这个过程就称为协程。也就是说在同一线程内一段代码在执行过程中会中断然后跳转执行别的代码，接着在之前中断的地方继续开始执行。

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

【示例 7-1】代码描述协程

```
def A():
```

```
    print('1')
    print('2')
    print('3')

def B():
    print('x')
    print('y')
    print('z')
```

由协程执行，在执行 A 的过程中，可以随时中断，去执行 B，B 也可能在执行过程中中断再去执行 A，可能的结果是：

```
1
2
x
y
3
z
```

但是在 A 中是没有调用 B 的，所以协程的调用比函数调用理解起来要难一些，看起来 A、B 的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

协程的优点：

（1）无需线程上下文切换的开销，协程避免了无意义的调度，由此可以提高性能（但也因此，程序员必须自己承担调度的责任，同时，协程也失去了标准线程使用多 CPU 的能力）。

（2）无需原子操作锁定及同步的开销。

（3）方便切换控制流，简化编程模型。

（4）高并发+高扩展性+低成本：一个 CPU 支持上万的协程都不是问题。所以很适合用于高并发处理。

协程的缺点：

（1）无法利用多核资源：协程的本质是个单线程，它不能同时将单个 CPU 的多个核用上，协程需要和进程配合才能运行在多 CPU 上。当然日常所编写的绝大部分应用都没有这个必要，除非是 cpu 密集型应用。

（2）进行阻塞（Blocking）操作（如 IO 时）会阻塞掉整个程序。

7.1.1 yield 的使用

Python 对协程的支持是通过 generator 实现的。在

generator 中，不但可以通过 for 循环来迭代，还可以不断调用 next()函数获取由 yield 语句返回的下一个值。

先把 yield 看做“return”，这个是直观的，它首先是个 return，普通的 return 是什么意思，就是在程序中返回某个值，返回之后程序就不再往下运行了。看做 return 之后再把它看做一个是生成器（generator）的一部分（带 yield 的函数才是真正的迭代器）。

【示例 7-2】yield 的使用

```
def foo():
    print("starting...")
    while True:
        res = yield 4
        print("res:",res)
g = foo()
print(next(g))
print("*"*20)
print(next(g))
```

执行结果如图 7-1 所示：

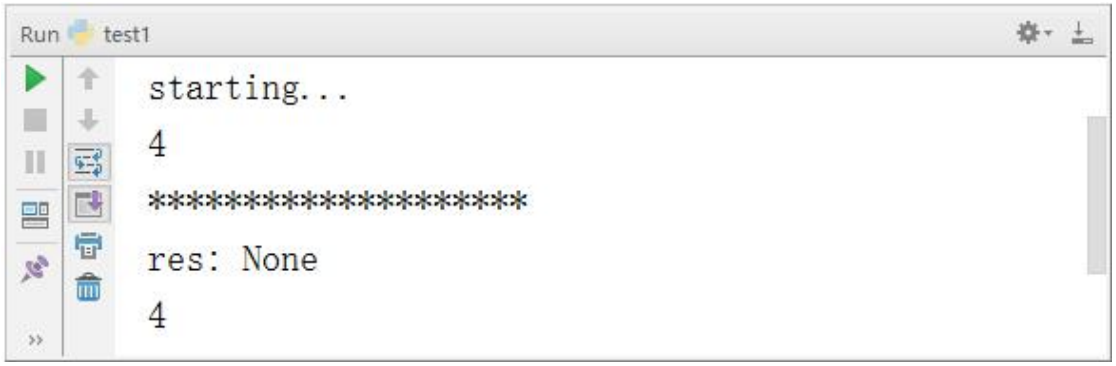


图 7-1 示例 7-2 运行效果图

上述代码执行过程：

（1）程序开始执行以后，因为 foo 函数中有 yield 关键字，所以 foo 函数并不会真的执行，而是先得到一个生成器 g(相当于一个对象)。

（2）直到调用 next 方法，foo 函数正式开始执行，先执行 foo 函数中的 print 方法，然后进入 while 循环。

（3）程序遇到 yield 关键字，然后把 yield 想成 return，return 了一个 4 之后，程序停止，并没有执行赋值给 res 操作，此时 next(g)语句执行完成，所以输出的前两行（第一个是 while 上面的 print 的结果，第二个是 return 出的结果）是执行 print(next(g))的结果。

（4）程序执行 print("*"*20)，输出 20 个*。

（5）又开始执行下面的 print(next(g))，这个时候和上面那个差不多，不过不同的是，

这个时候是从刚才那个 next 程序停止的地方开始执行的，也就是要执行 res 的赋值操作，这时候要注意，这个时候赋值操作的右边是没有值的（因为刚才那个是 return 出去了，并没有给赋值操作的左边传参数），所以这个时候 res 赋值是 None，所以接着下面的输出就是 res:None。

（6）程序会继续在 while 里执行，又一次碰到 yield，这个时候同样 return 出 4，然后程序停止，print 函数输出的 4 就是这次 return 出的 4。

带 yield 的函数是一个生成器，而不是一个函数了，这个生成器有一个函数就是 next 函数，next 就相当于“下一步”生成哪个数，这一次的 next 开始的地方是接着上一次的 next 停止的地方执行的，所以调用 next 的时候，生成器并不会从 foo 函数的开始执行，只是接着上一步停止的地方开始，然后遇到 yield 后，return 出要生成的数，此步就结束。

【示例 7-3】yield 简单实现协程

```
import time
def A():
    while True:
        print('----A----')
        yield
        time.sleep(0.5)

def B(c):
    while True:
        print('----B----')
        c.__next__()
        time.sleep(0.5)

if __name__ == '__main__':
    a=A()
    B(a)
```

执行结果如图 7-2 所示：

图 7-2 示例 7-3 运行效果图

7.1.2 send 发送数据

send 是发送一个参数给 res 的，因为上面讲到，return 的时候，并没有把 4 赋值给 res，下次执行的时候只好继续执行赋值操作，只好赋值为 None 了，而如果用 send 的话，开始执行的时候，先接着上一次（return 4 之后）执行，先把 10 赋值给了 res，然后执行 next 的作用，遇见下一回的 yield，return 出结果后结束。

【示例 7-4】yield 中 send 函数的使用

```
def foo():
    print("starting...")
    while True:
        res = yield 4
        print("res:",res)
g = foo()
print(next(g))
print("*"*20)
print(g.send(10))
```

执行结果如图 7-3 所示：

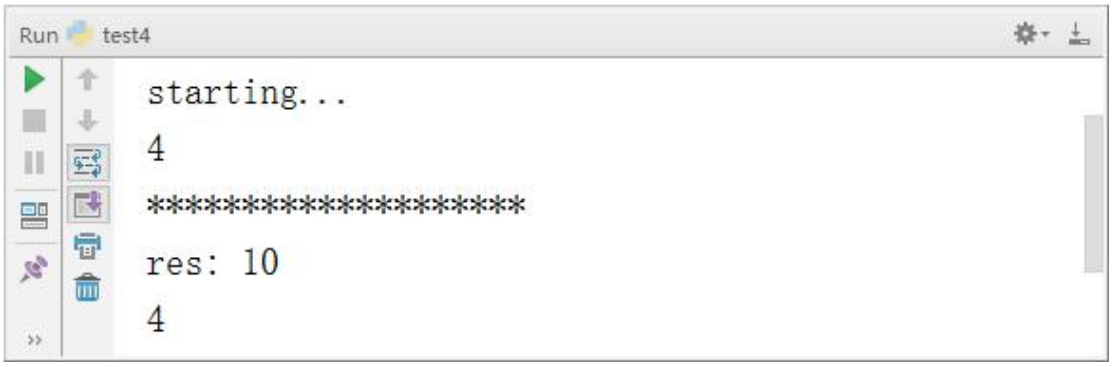


图 7-3 示例 7-4 运行效果图

【示例 7-5】协程实现生产者消费者

```
import time
#生产者
def produce(c):
    c.send(None)
    for i in range(1,6):
        print('生产者生产%d 产品'%i)
        c.send(str(i))
        time.sleep(1)
#消费者
def customer():
    res=''
    while True:
        data = yield res
        if not data:
            return
        print('消费者消费%s 产品'%data)

if __name__ == '__main__':
    c=customer()
    produce(c)
```

执行结果如图 7-4 所示：

图 7-4 示例 7-5 运行效果图

7.2 异步 IO（ asyncio） 协程

使用异步 IO，无非是提高我们写的软件系统的并发。这个软件系统，可以是网络爬虫，也可以是 Web 服务等等。

并发的方式有多种，多线程，多进程，异步 IO 等。多线程和多进程更多应用于 CPU 密集型的场景，比如科学计算的时间都耗费在 CPU 上，利用多核 CPU 来分担计算任务。多线程和多进程之间的场景切换和通讯代价很高，不适合 IO 密集型的场景。而异步 IO 就是非常适合 IO 密集型的场景，比如网络爬虫和 Web 服务。

IO 就是读写磁盘、读写网络的操作，这种读写速度比读写内存、CPU 缓存慢得多，前者的耗时是后者的成千上万倍甚至更多。这就导致，IO 密集型的场景 99%以上的时间都花费在 IO 等待的时间上。异步 IO 就是把 CPU 从漫长的等待中解放出来的方法。

7.3 asyncio

asyncio 是 Python 3.4 版本引入的标准库，直接内置了对异步 IO 的支持。asyncio 的编程模型就是一个消息循环。从 asyncio 模块中直接获取一个 EventLoop 的引用，然后把需要执行的协程扔到 EventLoop 中执行，就实现了异步 IO。

（1）event_loop 事件循环：程序开启一个无限的循环，程序员会把一些函数注册到事件循环上。当满足事件发生的时候，调用相应的协程函数。

（2）coroutine 协程：协程对象，指一个使用 async 关键字定义的函数，它的调用不会

立即执行函数，而是会返回一个协程对象。协程对象需要注册到事件循环，由事件循环调用。

（3）task 任务：一个协程对象就是一个原生可以挂起的函数，任务则是对协程进一步封装，其中包含任务的各种状态。

（4）future： 代表将来执行或没有执行的任务的结果。它和 task 上没有本质的区别。

（5）async/await 关键字：python3.5 用于定义协程的关键字，async 定义一个协程，await 用于挂起阻塞的异步调用接口。

7.3.1 定义一个协程

定义一个协程很简单，使用 async 关键字，就像定义普通函数一样：

【示例 7-6】定义一个协程

```
import time,asyncio
now = lambda : time.time()
#通过 aysnc 定义一个协程，该协程不能直接运行，需要将协程加入到事件循环中
async def do_work(x):
    print('waiting:%d'%x)

start = now()
#得到一个协程对象
coroutine=do_work(2)
#创建一个事件循环
loop=asyncio.get_event_loop()
#将协程对象加入到事件循环中
loop.run_until_complete(coroutine)
print('TIME: ', now() - start)
```

执行结果如图 7-5 所示：

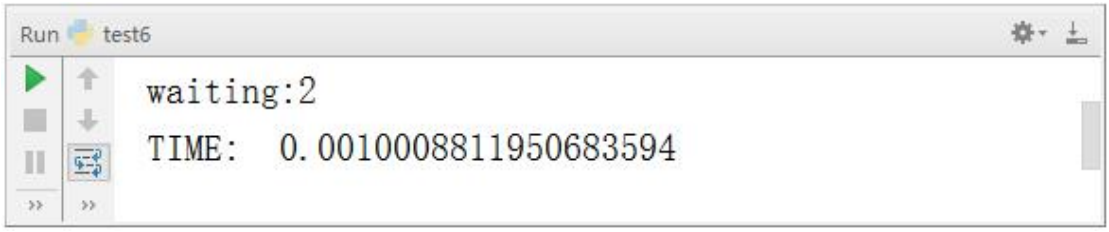
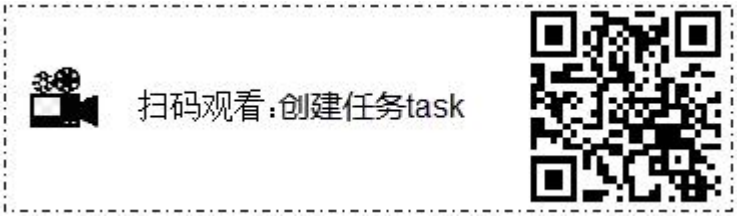


图 7-5 示例 7-6 运行效果图

通过 async 关键字定义一个协程（coroutine），协程也是一种对象。协程不能直接运行，需要把协程加入到事件循环（loop），由后者在适当的时候调用协程。asyncio.get_event_loop 方法可以创建一个事件循环，然后使用 run_until_complete 将协程注册到事件循环，并启动事件循环。

7.3.2 创建一个 task



协程对象不能直接运行，在注册事件循环的时候，其实是 run_until_complete 方法将协程包装成为了一个任务（task）对象。所谓 task 对象是 future 类的子类。保存了协程运行后的状态，用于未来获取协程的结果。

asyncio.ensure_future(coroutine) 和 loop.create_task(coroutine)都可以创建一个 task，run_until_complete 的参数是一个 futrue 对象。当传入一个协程，其内部会自动封装成 task，task 是 future 的子类。isinstance(task, asyncio.Future)将会输出 True。

【示例 7-7】创建一个 task

```
import asyncio
import time
now = lambda: time.time()
async def do_work(x):
    print('Waiting: ', x)
start = now()
coroutine = do_work(2)
loop = asyncio.get_event_loop()
#创建一个 task
task = loop.create_task(coroutine)
#task=asyncio.ensure_future(coroutine)
print(task)
loop.run_until_complete(task)
print(task)
print('TIME: ', now() - start)
```

执行结果如图 7-6 所示：

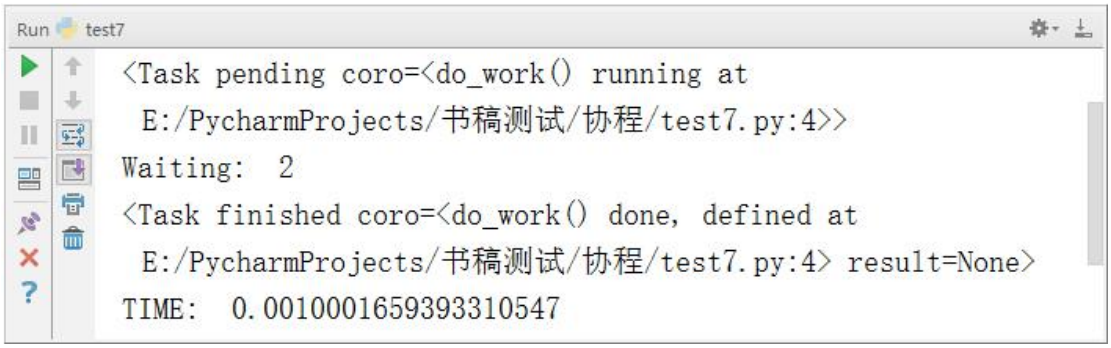


图 7-6 示例 7-7 运行效果图

创建 task 后，task 在加入事件循环之前是 pending 状态，因为 do_work 中没有耗时的阻塞操作，task 很快就执行完毕了。

7.3.3 绑定回调

绑定回调，在 task 执行完毕的时候可以获取执行的结果，回调的最后一个参数是 future 对象，通过该对象可以获取协程返回值。如果回调需要多个参数，可以通过偏函数导入。

【示例 7-8】绑定回调

```
import time
import asyncio
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    return 'Done after {}s'.format(x)
def callback(future):
    print('Callback: ', future.result())
start = now()
coroutine = do_some_work(2)
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(coroutine)
task.add_done_callback(callback)
loop.run_until_complete(task)
print('TIME: ', now() - start)
```

执行结果如图 7-7 所示：

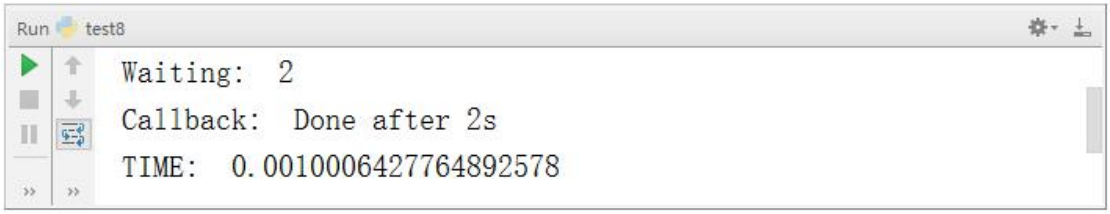


图 7-7 示例 7-8 运行效果图

从上面实例可以看到，coroutine 执行结束时候会调用回调函数。并通过参数 future 获取

协程执行的结果。创建的 task 和回调里的 future 对象，实际上是同一个对象。

7.3.4 future 与 result

回调一直是很多异步编程的恶梦，程序员更喜欢使用同步的编写方式写异步代码，以避免回调的恶梦。回调中使用了 future 对象的 result 方法。前面不绑定回调的例子中，可以看到 task 有 fiinished 状态。在那个时候，可以直接读取 task 的 result 方法。

【示例 7-9】直接读取 task 的 result 方法

```
import time
import asyncio
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting {}'.format(x))
    return 'Done after {}s'.format(x)
start = now()
coroutine = do_some_work(2)
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(coroutine)
loop.run_until_complete(task)
print('Task ret: {}'.format(task.result()))
print('TIME: {}'.format(now() - start))
```

执行结果如图 7-8 所示：

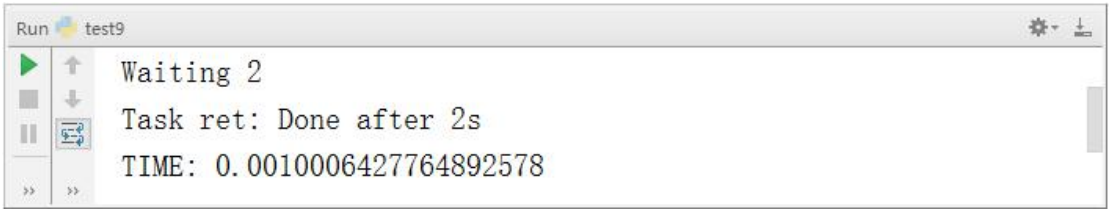


图 7-8 示例 7-9 运行效果图

7.3.5 阻塞和 await

使用 async 可以定义协程对象，使用 await 可以针对耗时的操作进行挂起，就像生成器里的 yield 一样，函数让出控制权。协程遇到 await，事件循环将会挂起该协程，执行别的协程，直到其他的协程也挂起或者执行完毕，再进行下一个协程的执行。

耗时的操作一般是一些 IO 操作，例如网络请求，文件读取等。使用 asyncio.sleep 函数

来模拟 IO 操作。协程的目的也是让这些 IO 操作异步化。

【示例 7-10】asyncio.sleep 函数来模拟 IO 操作

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)
start = now()
coroutine = do_some_work(2)
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(coroutine)
loop.run_until_complete(task)
print('Task ret: ', task.result())
print('TIME: ', now() - start)
```

执行结果如图 7-9 所示：

图 7-9 示例 7-10 运行效果图

7.3.6 并发和并行

并发和并行一直是容易混淆的概念。并发通常指有多个任务需要同时进行，并行则是同一时刻有多个任务执行。用上课来举例就是，并发情况下是一个老师在同一时间段辅助不同的人功课。并行则是好几个老师分别同时辅助多个学生功课。简而言之就是一个人同时吃三个馒头还是三个人同时分别吃一个的情况，吃一个馒头算一个任务。

asyncio 实现并发，就需要多个协程来完成任务，每当有任务阻塞的时候就 await，然后其他协程继续工作。创建多个协程的列表，然后将这些协程注册到事件循环中。

【示例 7-11】asyncio 实现并发

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)
start = now()
coroutine1 = do_some_work(1)
coroutine2 = do_some_work(2)
coroutine3 = do_some_work(4)

tasks = [
    asyncio.ensure_future(coroutine1),
    asyncio.ensure_future(coroutine2),
    asyncio.ensure_future(coroutine3)
]
loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(tasks))

for task in tasks:
    print('Task ret: ', task.result())

print('TIME: ', now() - start)
```

执行结果如图 7-10 所示：

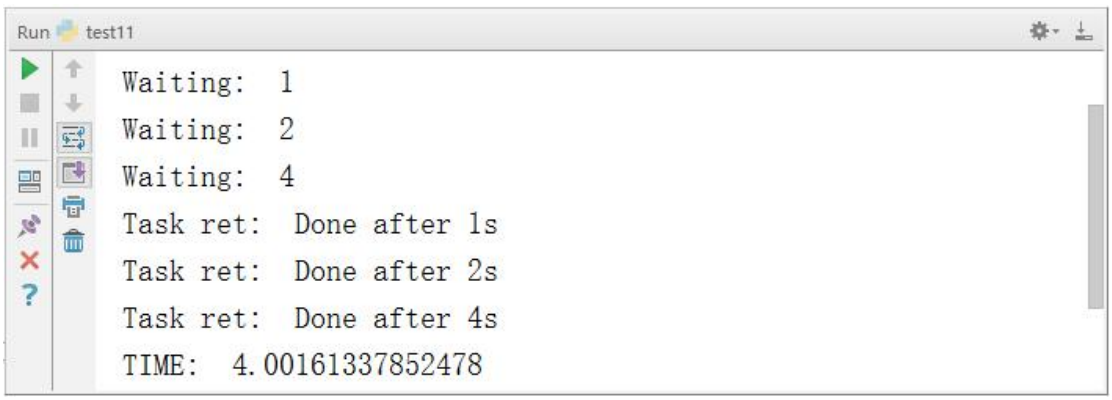
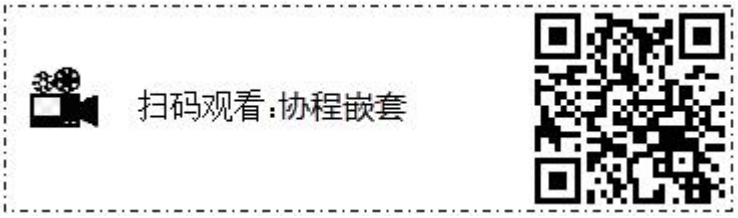


图 7-10 示例 7-11 运行效果图

总时间为 4s 左右。4s 的阻塞时间，足够前面两个协程执行完毕。如果是同步顺序的任务，那么至少需要 7s。此时我们使用了 aysncio 实现了并发。asyncio.wait(tasks) 也可以使用 asyncio.gather(*tasks) ，前者接受一个 task 列表，后者接收一堆 task。

7.3.7 协程嵌套



使用 async 可以定义协程，协程用于耗时的 io 操作，也可以封装更多的 io 操作过程，这样就实现了嵌套的协程，即一个协程中 await 了另外一个协程，如此连接起来。

【示例 7-12】协程嵌套

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)
    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    dones, pendings = await asyncio.wait(tasks)
    for task in dones:
        print('Task ret: ', task.result())
start = now()
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
print('TIME: ', now() - start)
```

执行结果如图 7-11 所示：

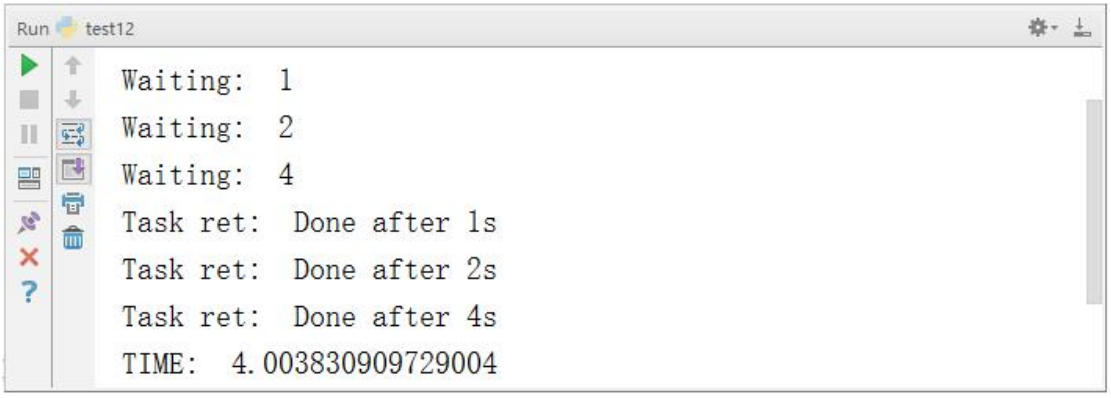


图 7-11 示例 7-12 运行效果图

如果使用的是 asyncio.gather 创建协程对象，那么 await 的返回值就是协程运行的结果。

【示例 7-13】asyncio.gather 创建协程对象

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)
    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    results = await asyncio.gather(*tasks)
    for result in results:
        print('Task ret: ', result)
start = now()
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
print('TIME: ', now() - start)
```

执行结果如图 7-12 所示：

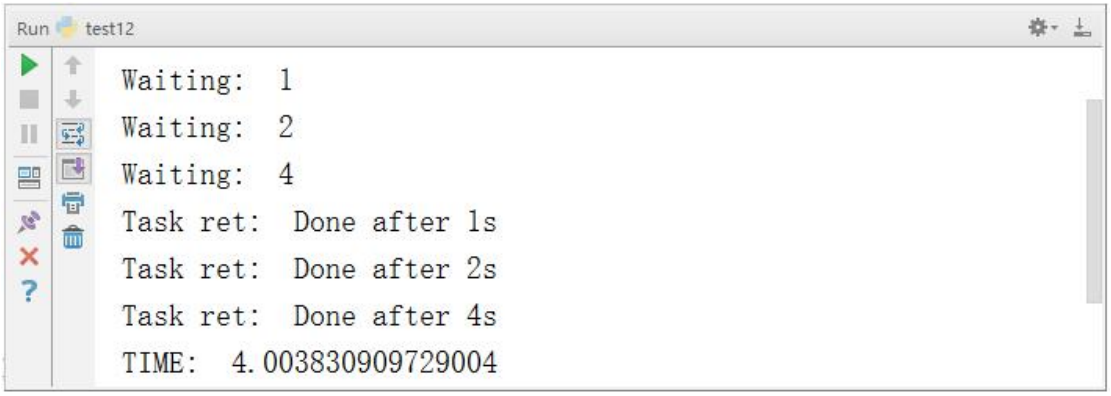


图 7-12 示例 7-13 运行效果图

不在 main 协程函数里处理结果，直接返回 await 的内容，那么最外层的 run_until_complete 将会返回 main 协程的结果。

【示例 7-14】不在 main 协程函数里处理结果

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(2)
    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    return await asyncio.gather(*tasks)
start = now()
loop = asyncio.get_event_loop()
results = loop.run_until_complete(main())
for result in results:
```

```
    print('Task ret: ', result)
```

执行结果如图 7-13 所示：

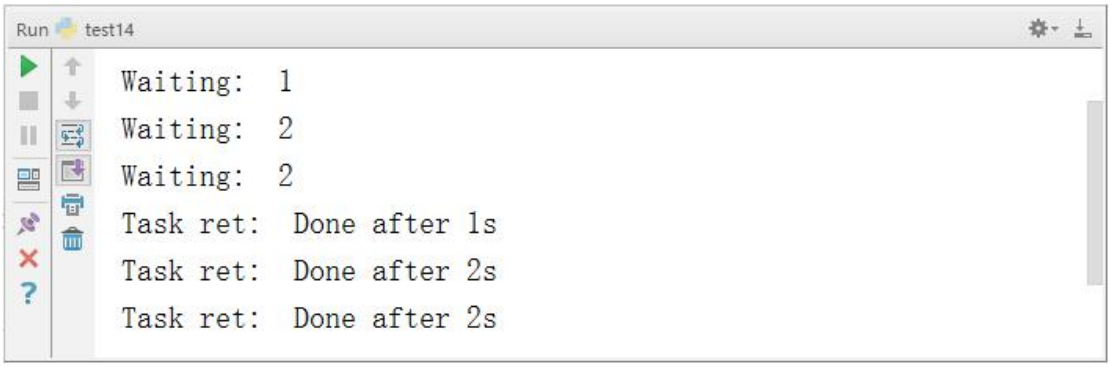


图 7-13 示例 7-14 运行效果图

或者返回使用 asyncio.wait 方式挂起协程。

【示例 7-15】使用 asyncio.wait 方式挂起协程

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)
    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    return await asyncio.wait(tasks)
start = now()
loop = asyncio.get_event_loop()
done, pending = loop.run_until_complete(main())
for task in done:
    print('Task ret: ', task.result())
```

执行结果如图 7-14 所示：

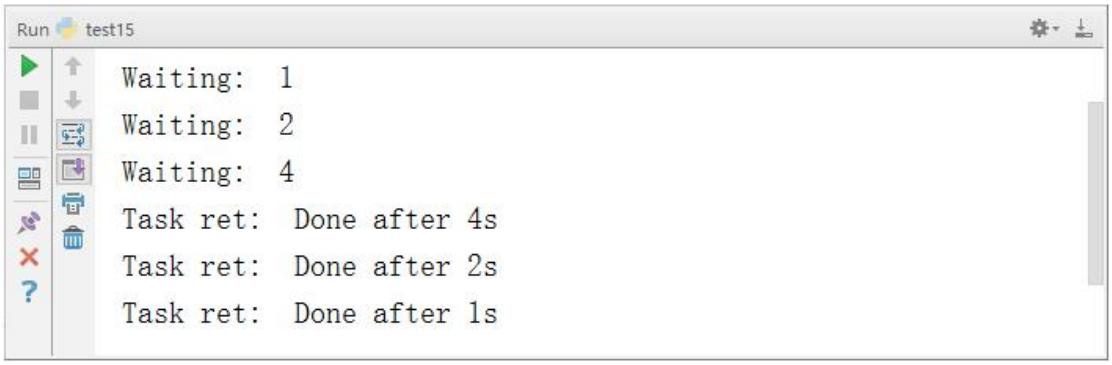


图 7-14 示例 7-15 运行效果图

【示例 7-16】使用 asyncio 的 as_completed 方法

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)

async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    for task in asyncio.as_completed(tasks):
        result = await task
        print('Task ret: {}'.format(result))
start = now()

loop = asyncio.get_event_loop()
```

```
done = loop.run_until_complete(main())
print('TIME: ', now() - start)
```

执行结果如图 7-15 所示：

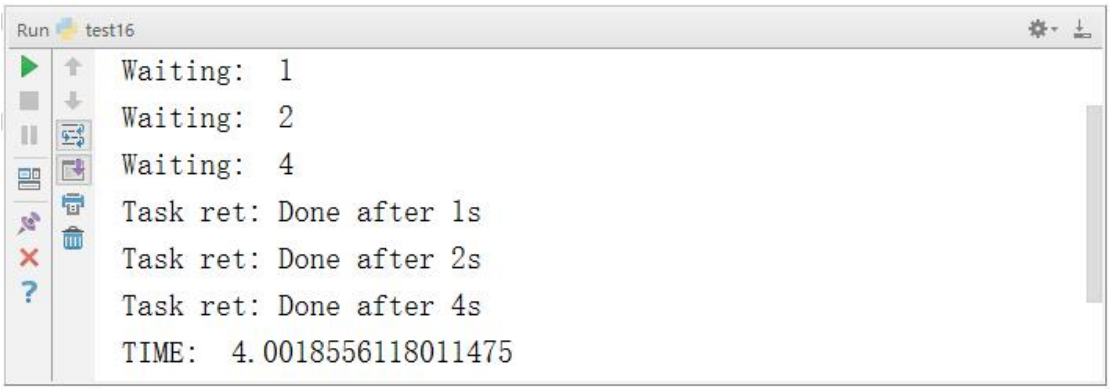
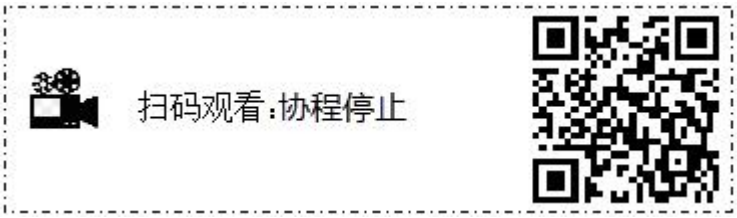


图 7-15 示例 7-16 运行效果图

7.3.8 协程停止



上面见识了协程的几种常用的用法，都是协程围绕着事件循环进行的操作。future 对象有几个状态：

（1）Pending

（2）Running

（3）Done

（4）Cancelled

创建 future 的时候，task 为 pending，事件循环调用执行的时候当然就是 running，调用完毕自然就是 done，如果需要停止事件循环，就需要先把 task 取消。可以使用 asyncio.Task 获取事件循环的 task。

循环 task，逐个 cancel 是一种方案，可是正如上面我们把 task 的列表封装在 main 函数中，main 函数外进行事件循环的调用。这个时候，main 相当于最外出的一个 task，那么处理包装的 main 函数即可。

【示例 7-17】把 task 的列表封装在 main 函数中，协程停止

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
```

```
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(2)
    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    done, pending = await asyncio.wait(tasks)
    for task in done:
        print('Task ret: ', task.result())
start = now()
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(main())
try:
    loop.run_until_complete(task)
except KeyboardInterrupt as e:
    print(asyncio.Task.all_tasks())
    print(asyncio.gather(*asyncio.Task.all_tasks()).cancel())
    loop.stop()
    loop.run_forever()
finally:
    loop.close()
```

执行结果如图 7-16 所示：

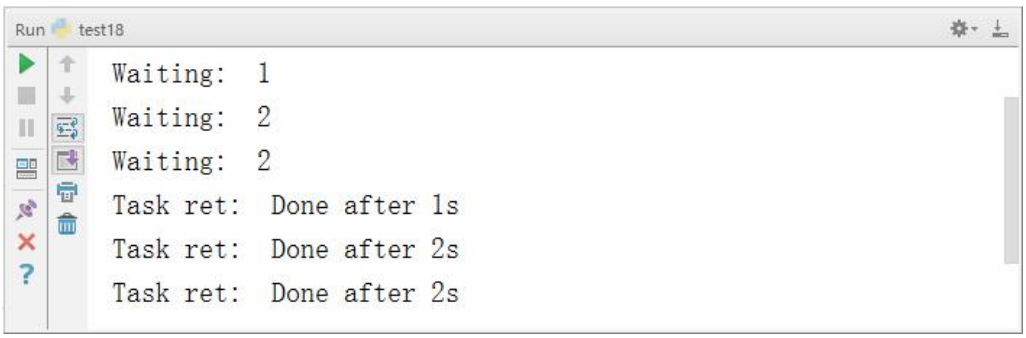


图 7-16 示例 7-17 运行效果图

习题

一、简答题

1. 什么是协程
2. 协程与线程的区别
3. 协程的优缺点
4. 协程的应用
5. 执行下面代码输出结果是：

```
def test():
    print("1"*30)
    yield "A"
    print("A"*30)
    yield "B"
    print("B"*30)


t = test()
print(next(t))
print(next(t))
print(next(t))
```

二、编码题

1. 编写一个程序，使用 yield 简单实现协程。
2. 请写一个简单的协程示例或利用协程实现一个生产者消费者模式。

第八章 函数式编程和高阶函数

函数是 Python 内建支持的一种封装，通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数。

通过阅读本章，你可以：

- 了解函数式编程
- 掌握高阶函数 map、reduce、filter、sorted 的使用
- 掌握匿名函数 lambda 的使用
- 掌握闭包及装饰器的使用
- 掌握偏函数的使用

8.1 高阶函数

变量可以指向函数

【示例 8-1】求绝对值的函数 abs()

```
print(abs(-10))
```

执行结果如图 8-1 所示：

图 8-1 示例 8-1 运行效果图

【示例 8-2】打印绝对值的函数名 abs

```
print(abs)
```

执行结果如图 8-2 所示：

图 8-2 示例 8-2 运行效果图

可见，abs(-10)是函数调用，而 abs 是函数本身。要获得函数调用结果，可以把结果赋值给变量：

【示例 8-3】把函数本身赋值给变量

```
f = abs
print(f)
```

执行结果如图 8-3 所示：

图 8-3 示例 8-3 运行效果图

函数本身也可以赋值给变量，即：变量可以指向函数。如果一个变量指向了一个函数，那么，可以通过该变量来调用这个函数。

【示例 8-4】通过变量来调用函数

```
f = abs
print(f(-10))
```

执行结果如图 8-4 所示：

图 8-4 示例 8-4 运行效果图

函数名也是变量

函数名其实就是指向函数的变量！对于 abs()这个函数，完全可以把函数名 abs 看成变量，它指向一个可以计算绝对值的函数。

【示例 8-5】把 abs 指向其他对象

```
abs = 10
print(abs(-10))
```

执行结果如图 8-5 所示：

图 8-5 示例 8-5 运行效果图

把 abs 指向 10 后，就无法通过 abs(-10)调用该函数了！因为 abs 这个变量已经不指向求绝对值函数了！

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

【示例 8-6】高阶函数的示例

```
def add(x, y, f):
    return f(x) + f(y)
print(add(-5, 6, abs))
```

执行结果如图 8-6 所示：

图 8-6 示例 8-6 运行效果图

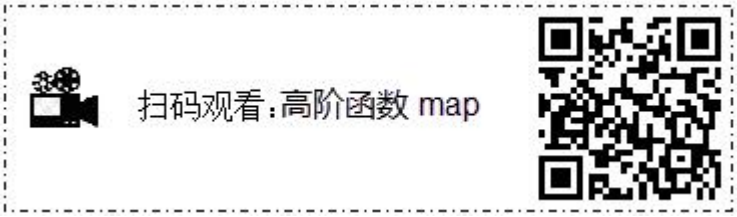
当调用 add(-5, 6, abs)时，参数 x，y 和 f 分别接收-5，6 和 abs，根据函数定义，可以推导计算过程为：

```
x ==> -5
y ==> 6
f ==> abs
f(x) + f(y) ==> abs(-5) + abs(6) ==> 11
```

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。Python 内建的高阶函数有 map、reduce、filter、sorted。

8.1.1 map

map()函数接收两个参数，一个是函数，一个是序列，map 将传入的函数依次



作用到序列的每个元素，并把结果作为新的 list 返回。

比如有一个函数 f(x)=x2，要把这个函数作用在一个 list [1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用 map()实现如图 8-7：

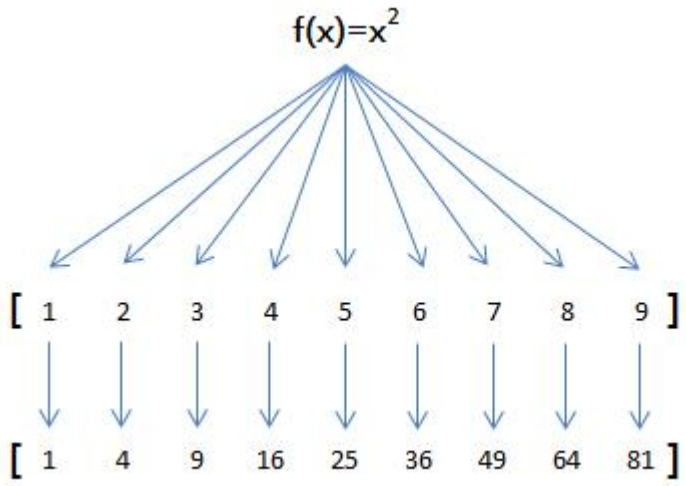


图 8-7 map 函数的实现

可能会想，不需要 map()函数，也可以计算出结果，写一个循环，实现代码如下：

【示例 8-7】循环代码实现 f(x)=x2

```
def f(x):
    return x * x
L = []
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    L.append(f(n))
print(L)
```

执行结果如图 8-8 所示：

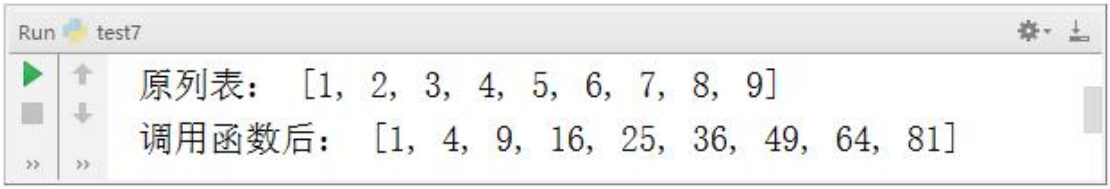


图 8-8 示例 8-7 运行效果图

【示例 8-8】高阶函数 map()的实现 f(x)=x2

```
def f(x):
    return x * x
L=map(f,[1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
print(list(L))
```

执行结果如图 8-9 所示：

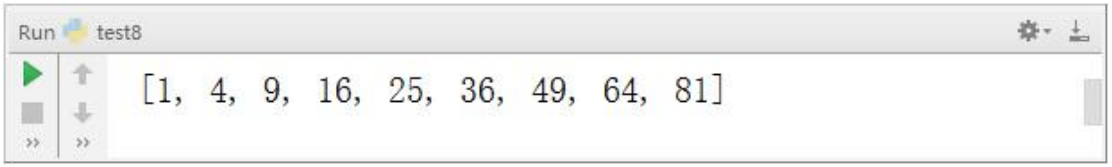


图 8-9 示例 8-8 运行效果图

从上面的实例可以看到，map()作为高阶函数，事实上它把运算规则抽象了，因此，不但可以计算简单的 f(x)=x2，还可以计算任意复杂的函数，比如，把这个 list 所有数字转为字符串，实例代码如下：

【示例 8-9】高阶函数 map()的实现将列表元素转为字符串

```
L=map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print(list(L))
```

执行结果如图 8-10 所示：

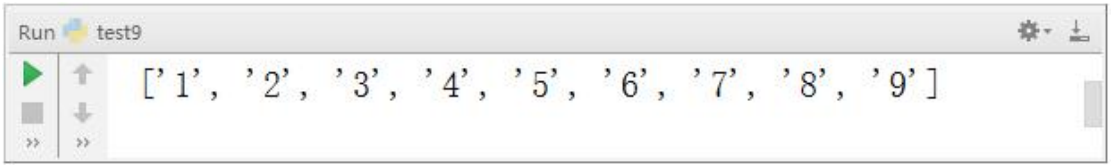


图 8-10 示例 8-9 运行效果图

【示例 8-10】高阶函数 map()的传入两个列表的使用

```
def f2(x,y):
    return x+y
L=map(f2,[1,2,3,4],[10,20,30])
print(list(L))
```

执行结果如图 8-11 所示：

图 8-11 示例 8-10 运行效果图

从上面的运行结果可以看到，map()函数将传入的两个列表对应位置元素作为函数 f2 的参数。

8.1.2 reduce

reduce 把一个函数作用在一个序列[x1, x2, x3...]上，

这个函数必须接收两个参数，reduce 把结果继续和序列的下一个元素做累积计算，其效果就是：

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

比如说对一个序列求和，就可以用 reduce 实现，实例如下：

【示例 8-11】高阶函数 reduce()实现对一个序列求和

```
from functools import reduce
def add(x, y):
    return x + y
sum=reduce(add, [1, 3, 5, 7, 9])
print(sum)
```

执行结果如图 8-12 所示：

图 8-12 示例 8-11 运行效果图

如果想把序列[1, 3, 5, 7, 9]变换成整数 13579。将列表中的每个元素乘以 10 加上后一个元素。

【示例 8-12】高阶函数 reduce()实现把序列[1, 3, 5, 7, 9]变换成整数 13579

```
from functools import reduce
def fn(x, y):
    return x * 10 + y
a = reduce(fn, [1, 3, 5, 7, 9])
print("reduce 执行结果:",a)
```

执行结果如图 8-13 所示：

图 8-13 示例 8-12 运行效果图

8.1.3 filter

Python 内建的 filter()函数用于过滤序列。和 map()类似，filter()也接收一个函数

和一个序列。和 map()不同的时，filter()把传入的函数依次作用于每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素。

【示例 8-13】高阶函数 filter()过滤列表，删掉偶数，只保留奇数

```
# 在一个 list 中，删掉偶数，只保留奇数
def is_odd(n):
    return n % 2 == 1
L=filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])
print(list(L))
```

执行结果如图 8-14 所示：

图 8-14 示例 8-13 运行效果图

【示例 8-14】高阶函数 filter()删除序列中的空字符串

```
def not_empty(s):
    return s and s.strip()


L=filter(not_empty, ['A', '', 'B', None, 'C', '  '])
print(list(L))
```

执行结果如图 8-15 所示：

图 8-15 示例 8-14 运行效果图

8.1.4 sorted

排序算法也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，可以直接比较，但如果是字符串或者两个 dict 呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。通常规定，对于两个元素 x 和 y，如果认为 x < y，则返回-1，如果认为 x == y，则返回 0，如果认为 x > y，则返回 1，

这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。

Python 内置的 sorted()函数就可以对 list 进行排序：

【示例 8-15】高阶函数 sorted()对 list 进行排序

```
sorter1 = sorted([1,3,6,-20,34])
print("升序排列:",sorter1)
```

执行结果如图 8-16 所示：

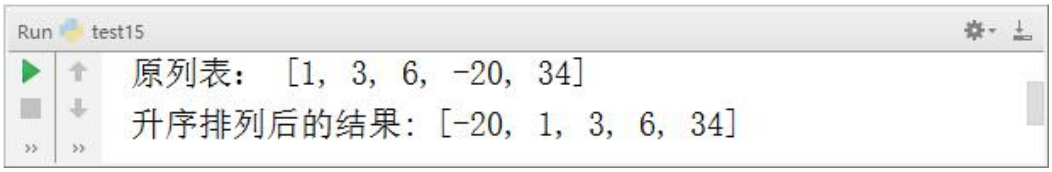


图 8-16 示例 8-15 运行效果图

sorted()函数也是一个高阶函数，它还可以接收一个 key 函数来实现自定义的排序。

【示例 8-16】高阶函数 sorted()参数 key、reverse 的使用

```
sorter1 = sorted([1,3,6,-20,34])
print("升序排列:",sorter1)

# sorted()函数也是一个高阶函数，它还可以接收一个 key 函数来实现自定义的排序
sorter2 = sorted([1,3,6,-20,-70],key=abs)
print("自定义排序:",sorter2)

sorter2 = sorted([1,3,6,-20,-70],key=abs,reverse=True)
print("自定义反向排序:",sorter2)
# 4.2 字符串排序依照 ASCII
sorter3 = sorted(["ABC","abc","D","d"])
print("字符串排序:",sorter3)
# 4.3 忽略大小写排序
sorter4 = sorted(["ABC","abc","D","d"],key=str.lower)
print("忽略字符串大小写排序:",sorter4)
# 4.4 要进行反向排序，不必改动 key 函数，可以传入第三个参数 reverse=True：
sorter5 = sorted(["ABC","abc","D","d"],key=str.lower,reverse=True)
print("忽略字符串大小写反向排序:",sorter5)
```

执行结果如图 8-17 所示：

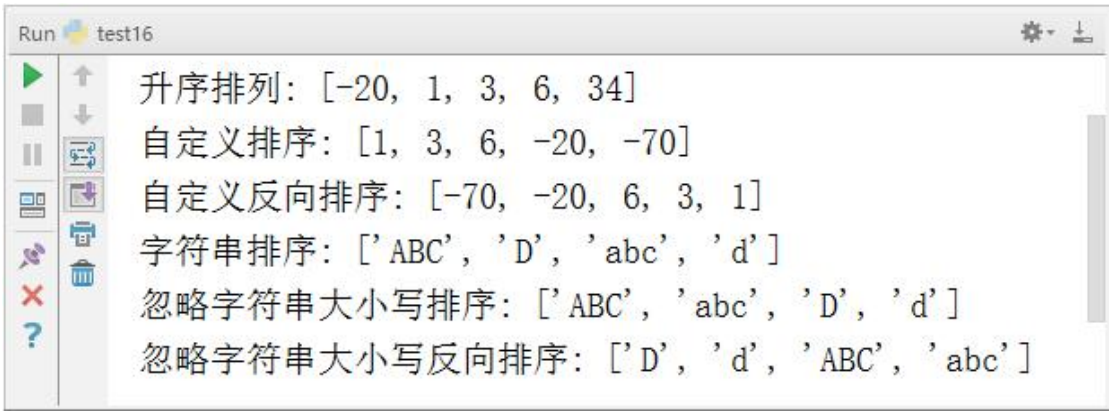


图 8-17 示例 8-16 运行效果图

8.2 匿名函数

在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在 Python 中，对匿名函数提供了支持。计算 f(x)=x2 时，除了定义一个 f(x)的函数外，还可以直接传入匿名函数。使用 lambda 可以声明一个匿名函数。

lambda 表达式就是一个简单的函数。使用 lambda 声明的函数可以返回一个值，在调用函数时，直接使用 lambda 表达式的返回值。使用 lambda 声明函数的语法格式如下。

```
lambda  arg1,arg2,arg3...  :  <表达式>
```

其中 arg1/arg2/arg3 为函数的参数。<表达式>相当于函数体。运算结果是：表达式的运算结果。

匿名函数有个限制，就是只能有一个表达式，不用写 return，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数。

【示例 8-17】lambda 表达式使用

```
f = lambda a,b,c:a+b+c
print('2+3+4 的结果:',f(2,3,4))
```

执行结果如图 8-18 所示：

图 8-18 示例 8-17 运行效果图

【示例 8-18】匿名函数实现 f(x)=x*x

```
L=map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print(list(L))
```

执行结果如图 8-19 所示：

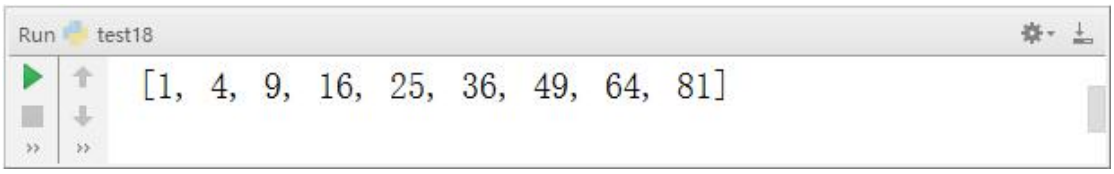


图 8-19 示例 8-18 运行效果图

【示例 8-19】高阶函数 sorted()对自定义对象进行排序

```
class Student:
    def __init__(self,age,name):
        self.name=name
        self.age=age
stu1=Student(11,'aaa')
stu2=Student(21,'ccc')
stu3=Student(31,'bbb')
student_list=sorted([stu1,stu2,stu3],key=lambda x:x.age)
# student_list=sorted([stu1,stu2,stu3],key=lambda x:x.name)
for stu in student_list:
    print('name:',stu.name,'age:',stu.age)
```

执行结果如图 8-20 所示：

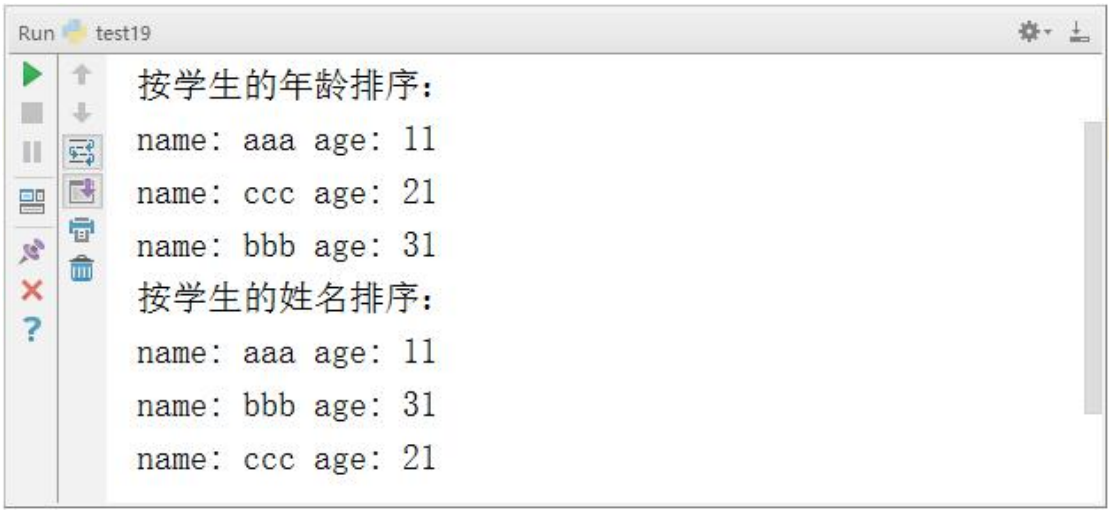


图 8-20 示例 8-19 运行效果图

8.3 闭包

根据字面意思，可以形象地把闭包理解为一个封闭的包裹，这个包裹就是一个函数。在 Python 语言中，闭包意味着要有嵌套定义，内部函数使用外部函数中定义的变量，外部函数返回内部函数名。如果调用一个函数 A，这个函数 A 返回一个函数 B，这个返回的函数 B 就叫作闭包。

【示例 8-20】闭包的定义

```
def func_out(num1):
    def func_in(num2):
        return num1+num2
    return func_in
f=func_out(10)
result=f(20)
print('结果：',result)
```

执行结果如图 8-21 所示：

图 8-21 示例 8-20 运行效果图

【示例 8-21】求两点之间的距离(传统方式实现)

```
import math
def getDis(x1,y1,x2,y2):
    return math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))
#求(1,1)和（2,2）分别到原点(0,0)的距离
print('点(1,1)到(0,0)的距离：',getDis(1,1,0,0))
print('点(2,2)到(0,0)的距离：',getDis(2,2,0,0))
```

执行结果如图 8-22 所示：

图 8-22 示例 8-21 运行效果图

【示例 8-22】求两点之间的距离(闭包方式实现)

```
#使用闭包求两点之间的距离
def getDisOut(x1,y1):
    def getDisIn(x2,y2):
        return math.sqrt((x1-x2)**2+(y1-y2)**2)
    return getDisIn

#求点(1,1)距离原点(0,0)的距离
#调用外部函数
getDisIn=getDisOut(0,0)
result=getDisIn(1,1)
print('点(1,1)距离原点(0,0)的距离',result)
#求点(2,2)距离原点(0,0)的距离
result=getDisIn(2,2)
print('点(2,2)距离原点(0,0)的距离',result)
```

执行结果如图 8-23 所示：

图 8-23 示例 8-22 运行效果图

现在，有两个函数 fun1 和 fun2，假设要增强 fun1()函数和 fun2()函数的功能，比如，在函数调用前自动打印日志，可以在函数 fun1()、fun2()直接调用写入日志的函数 writeLog()。

【示例 8-23】添加日志功能（传统方式实现）

```
import time
def writeLog(func):
    try:
        f=open('log.txt','a')
```

```
            f.write(func.__name__)
            f.write('\t')
            f.write(time.asctime())
            f.write('\n')
        except Exception as e:
            print(e.args)
        finally:
            f.close()
def fun1():
    writeLog(fun1)
    print('功能 1')

def fun2():
    writeLog(fun2)
    print('功能 2')

fun1()
fun2()
```

执行上述代码会生成日志 log.txt 文件内容如下：

```
    fun1 Tue Nov 26 14:37:23 2019
    fun2 Tue Nov 26 14:37:23 2019
```

上述代码已经实现了调用函数 fun1()和 fun2()前添加日志的功能，但修改了函数 fun1()和 fun2()的代码。实际应用是在不修改源代码的前提下，添加新的功能，这就需要使用闭包。

【示例 8-24】添加日志功能（闭包方式实现）

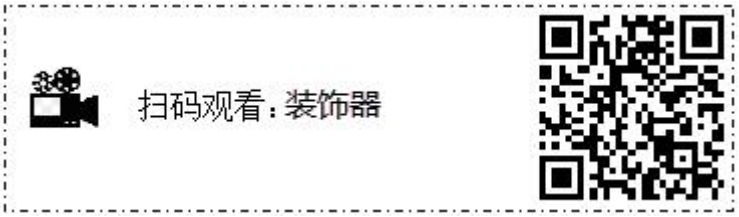
```
import time
def writeLog(func):
    try:
        f=open('log.txt','a')
        f.write(func.__name__)
        f.write('\t')
        f.write(time.asctime())
        f.write('\n')
    except Exception as e:
        print(e.args)
```

```
        finally:
            f.close()
def fun1():
        print('功能 1')

def fun2():
        print('功能 2')

#不修改源代码的基础上，添加日志功能
def func_out(func):
        def func_in():
            #调用添加日志功能的方法
            writeLog(func)
            func()
        return func_in
fun_in1=func_out(fun1)
fun_in1()
fun_in2=func_out(fun2)
fun_in2()
```

执行上述代码日志 log.txt 文件内容如下：

```
        fun1 Tue Nov 26 14:37:23 2019
        fun2 Tue Nov 26 14:37:23 2019
        fun1 Tue Nov 26 14:51:18 2019
        fun2 Tue Nov 26 14:51:18 2019
```

8.4 装饰器



在上述代码中闭包的调用实在繁琐。在 python 程序中，装饰器就是一种闭包，它可以使闭包的访问方式更简单。本质上，装饰器（decorator）就是一个返回函数的高阶函数。所以，要定义一个能打印日志的装饰器（decorator）。

在 Python 中使用装饰器，需要使用一个特殊的符号“@”来实现。在定义装饰器函数或类时，使用“@装饰器名称”的形式将符号“@”放在函数或类的定义行之前。

【示例 8-25】装饰器实现添加日志功能。

```
import time
def writeLog(func):
    try:
        f=open('log.txt','a')
        f.write(func.__name__)
        f.write('\t')
        f.write(time.asctime())
        f.write('\n')
    except Exception as e:
        print(e.args)
    finally:
        f.close()
#不修改源代码的基础上，添加日志功能
def func_out(func):
    def func_in():
        #调用添加日志功能的方法
        writeLog(func)
        func()
    return func_in
@func_out
def fun1():
    print('功能 1')
@func_out
def fun2():
    print('功能 2')
fun1()
fun2()
```

执行上述代码日志 log.txt 文件内容如下：

```
    fun1 Tue Nov 26 14:37:23 2019
    fun2 Tue Nov 26 14:37:23 2019
    fun1 Tue Nov 26 14:51:18 2019
    fun2 Tue Nov 26 14:51:18 2019
    fun1 Tue Nov 26 15:32:28 2019
    fun2 Tue Nov 26 15:32:28 2019
```

在上述示例中可以看到，在使用了装饰器后，调用函数 fun1 和 fun2 和普通函数调用没有区别，而装饰器定义的功能会自动插入函数 fun1 和 fun2 中。

【示例 8-26】多个装饰器的使用

```
def funOut(func):
    print('我是装饰器 1')
    def funIn():
        print('I am foo')
        func()
    return funIn

def funOut2(func):
    print('我是装饰器 2')
    def funIn():
        print('hello')
        func()
    return funIn
@funOut2
@funOut
def foo():
    print('foo  函数正在运行')

foo()
```

执行结果如图 8-24 所示：

图 8-24 示例 8-26 运行效果图

由上面的示例可以看到，如果给功能函数添加多个装饰器时候，离功能函数最近的先进行装饰。

【示例 8-27】两个参数的装饰器

```
def fun_out(func):
    def fun_in(x,y):
        func(x,y)
    return fun_in
@fun_out
def test(a,b):
    print('参数 a b  的值：',a,b)
test(10,20)
```

执行结果如图 8-25 所示：

图 8-25 示例 8-27 运行效果图

【示例 8-28】三个参数的装饰器

```
def fun_out(func):
    def fun_in(x,y):
        func(x,y)
    return fun_in
@fun_out
def test(a,b):
    print('参数 a b  的值：',a,b)
test(10,20)
#三个参数
def fun_out1(func):
    def fun_in(x,y,z):
        func(x,y,z)
    return fun_in
def test1(a,b,c):
    print('参数 a b c 的值：',a,b,c)


test1(10,20,30)
```

执行结果如图 8-26 所示：

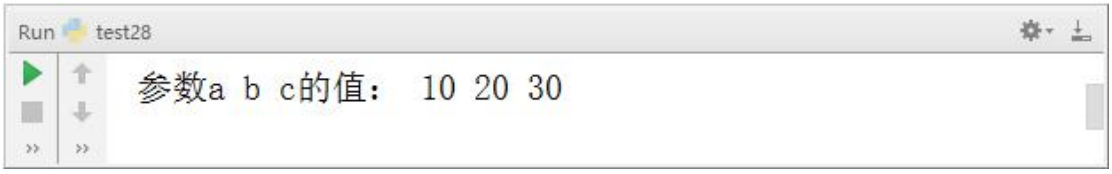


图 8-26 示例 8-28 运行效果图

从上述示例可以看到如果指定两个参数，需要定义一个装饰器。如果指定三个参数也需要定义一个装饰器。因此可以定义一个通用的装饰器。

【示例 8-29】通用的装饰器

```
def fun_out(func):
    def fun_in(*args,**kwargs):
        return func(*args,**kwargs)
    return fun_in

@fun_out
def test1(a):
    print('一个参数 a：',a)

@fun_out
def test2(a,b):
    print('两个参数 a b：',a,b)\

@fun_out
def test3(a,b,c):
    print('两个参数 a b c：',a,b,c)
test1(10)
test2(10,20)
test3(10,20,30)
```

执行结果如图 8-27 所示：

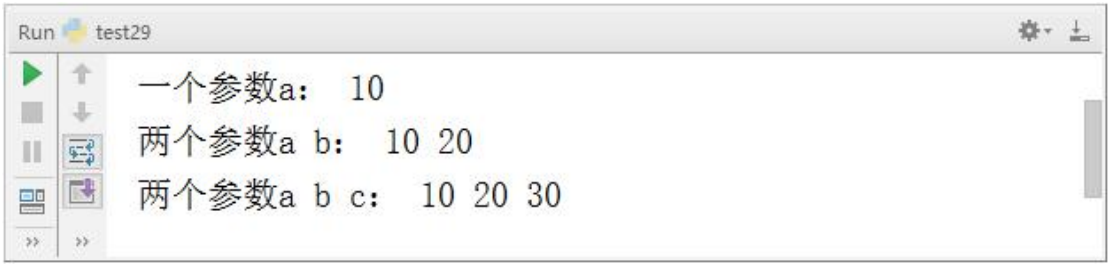


图 8-27 示例 8-29 运行效果图

8.5 偏函数

Python 的 functools 模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

偏函数是用于对函数固定属性的函数，作用就是把一个函数某些参数固定住（也就是设置默认值），返回一个新的函数，调用这个新的函数会更简单。

在介绍函数参数的时候，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。示例如下：

【示例 8-30】int()函数的使用

```
print(int('12345'))
```

执行结果如图 8-28 所示：

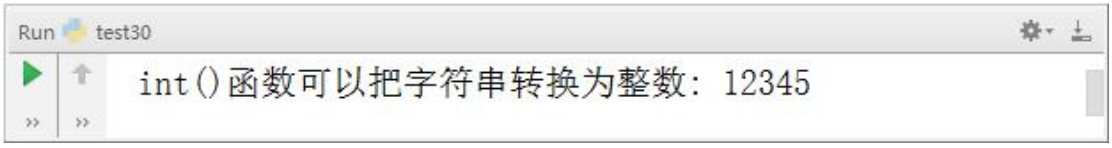


图 8-28 示例 8-30 运行效果图

但 int()函数还提供额外的 base 参数，默认值为 10。如果传入 base 参数，就可以做 N 进制的转换。

【示例 8-31】int()函数指定 base 参数

```
#base 参数
print('转换为八进制',int('12345', base=8))
print('转换为十六进制',int('12345', 16))
```

执行结果如图 8-29 所示：

图 8-29 示例 8-31 运行效果图

假设要转换大量的二进制字符串，每次都传入 int(x, base=2)非常麻烦，于是，可以定义一个 int2()的函数，默认把 base=2 传进去，现在定义一个 int2 函数，代码如下：

【示例 8-32】int2()函数的定义

```
def int2(x, base=2):
    return int(x, base)
print(int2('1000000')) #64
print(int2('1010101')) #85
```

执行结果如图 8-30 所示：

图 8-30 示例 8-32 运行效果图

从上述代码可以看到，定义了函数 int2 后转换二进制就非常方便了。

functools.partial 就是创建一个偏函数的，不需要自己定义 int2()，可以直接使用 functools.partial 创建一个新的函数 int2。

【示例 8-33】functools.partial 创建 int2()函数

```
import functools
int2 = functools.partial(int, base=2)
print(int2('1000000')) #64
print(int2('1010101')) #85
```

执行结果如图 8-31 所示：

图 8-31 示例 8-33 运行效果图

简单说 functools.partial 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

习题

一、选择题

1. 关于装饰器，下列说法错误的是（）。

A. 装饰器是一个包裹函数

B. 装饰器只能有一个参数

C. 通过在函数定义的面前加上@符号阿和装饰器名，使得装饰器函数生效

D. 如果装饰器带有参数，则必须在装饰函数的外层再嵌套一层函数

2. 下列函数中，用于使用函数对制定序列进行过滤的是（）。

A. map 函数 B. select 函数 C.filter 函数 D.reduce 函数

3. 下列选项中，不能作为 filter 函数参数的是（）。

A. 列表 B. 元祖 C.字符串 D.整数

4. 下列关于闭包和装饰器叙述正确的是（）（多选）

A. 闭包是内部函数对外部作用域的变量进行引用。

B. 当外部函数执行结束，其内部闭包引用的变量一定会立即释放。

C. 装饰器函数至少要接收一个函数。

D. 装饰器既能装饰带参数的函数，也能自己带参数。

5. 下列有关 map 函数叙述正确的是（）

A. 如果 map 函数传入的两个序列个数不同，那么个数多的序列会把多余的元素删除。

B. map 函数只能传递一个序列。

C. map 传入函数的参数个数必须跟序列的个数一样。

D. map 函数处理的结果是迭代器。

二、解答题

1. 什么是函数式编程？
2. 高阶函数的应用场景。
3. 请简述 map、filter、reduce 函数的作用。
4. 闭包和装饰器的区别。
5. 执行下面代码，输出结果是：

```
def f(x):
    return x*x
r=map(f,[1,2,3,4,5,6,7,8,9])
print(r)
```

三、编码题

1. 利用 map()函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的

规范名字。 例如输入：[‘adam’, ‘LISA’, ‘barT’]，输出：[‘Adam’, ‘Lisa’, ‘Bart’]。

2. 用 filter()删除 1~100 的素数。

3. 为函数写一个装饰器，把函数的返回值 +100 然后再返回。

4. 使用 lambda 定义匿名函数，实现两数之和。

附录 1：Python3.8 新特性

目前 Python 3.8 是 Python 语言的最新版本，它适合用于编写脚本、自动化以及机器学习和 Web 开发等各种任务。现在 Python 3.8 已经进入官方的 beta 阶段，这个版本带来了许多语法改变、内存共享、更有效的序列化和反序列化、改进的字典和更多新功能。

Python 3.8 还引入了许多性能改进。总的来说，我们即将拥有一个更快、更精确、更一致和更现代的 Python。下面是 Python 3.8 的新功能和最重要的改变。

1）赋值表达式

Python 3.8 最明显的变化就是赋值表达式，即:=操作符。赋值表达式可以将一个值赋给一个变量，即使变量不存在也可以。它可以用在表达式中，无需作为单独的语句出现。

【示例 1-1】赋值表达式

```
#原来实现，需要两个语句
flag = False
print(flag)

#Python3.8 中，可以使用 flag 运算符将上面两个语句合并为一句
print(flag := False)
```

执行结果如图 1-1 所示:

图 1-1 示例 1-1 执行效果图

从上述示例可以看到，赋值表达式可以把 True 分配给 flag，并直接 print 这个值。一定要有（:= ），不然表达式也是无法正常执行的，有了新的赋值表达式符号，不仅在构造上更简便，有时也可以更清楚的传达代码意图。

【示例 1-2】在 while 循环中，使用赋值表达式（:= ）

```
#原来实现方式
inputs = list()
current = input("请输入: ")
while current != "quit":
    inputs.append(current)
    current = input("请输入: ")
print('原来实现方式执行结果：',inputs)
```

```
#使用赋值表达式，进一步简化这段循环：
inputs = list()
while (current := input('请输入：')) != 'quit':
    inputs.append(current)
print('使用赋值表达式执行结果：',inputs)
```

执行结果如图 1-2 所示:

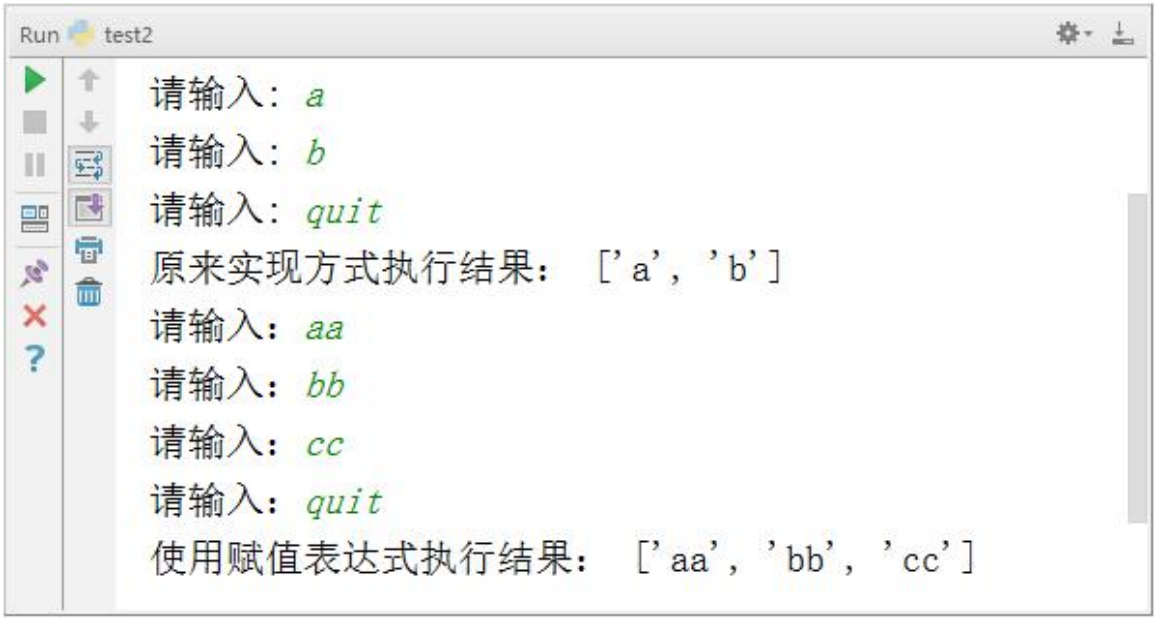


图 1-2 示例 1-2 执行效果图

从上述示例可以看到，使用赋值表达式，可以再进一步简化代码，但是代码可读性确变差了，所以，大家要使用赋值表达式的方法还需要结合自身进行判断。

注意：

□ PEP572 中描述了复制表达式的所有细节，大家可以深入阅读。

2） 仅通过位置指定的参数

仅通过位置指定的参数是函数定义中的一个新语法，可以让程序员强迫某个参数只能通过位置来指定。这样可以解决 Python 函数定义中哪个参数是位置参数、哪个参数是关键字参数的模糊性。

仅通过位置指定的参数可以用于如下情况：某个函数接受任意关键字参数，但也能接受一个或多个未知参数。Python 的内置函数通常都是这种情况，所以允许程序员这样做，能增强 Python 语言的一致性。

【示例 1-3】仅通过位置指定的参数

```
def pow(x, y, z=None, /):
    r = x**y
    if z is not None:
        r %= z
    return r
print('调用 pow(2,3)执行结果：',pow(2,3))
print('调用 pow(2,3,4)执行结果：',pow(2,3,4))
print('调用 pow(2,3,z=4)执行结果：',pow(2,3,z=4))
```

执行结果如图 1-3 所示:

图 1-3 示例 1-3 执行效果图

符号 / 分隔了位置参数和关键字参数。在上述示例中，所有参数都是未知参数。在以前版本的 Python 中，z 会被认为是关键字参数。但采用上述函数调用 pow(2, 3)和 pow(2, 3, 4)都是正确的调用方式，而 pow(2, 2, z=4)是不正确的。

3） 支持 f 字符串调试

在 Python 3.6 版本中增加了 f-strings，可以使用 f 前缀更方便地格式化字符串，同时还能进行计算，示例如下：

【示例 1-4】使用 f 前缀格式化字符串，并进行计算

```
a = 20
print(f'a 的值:{a}')
print(f'a 的值:{a+1}')
```

执行结果如图 1-4 所示:

图 1-4 示例 1-4 执行效果图

在 3.8 中只需要增加一个 = 符号，即可在同一个表达式内进行输出文本和值或变量的计算，而且效率更高。

【示例 1-5】在 3.8 中前缀格式化字符串 f 的使用

```
print(f'{a=}')
print(f'{a+1=}')
```

执行结果如图 1-5 所示:

图 1-5 示例 1-5 执行效果图

4） 多进程共享内存

在旧版本的 Python 中，进程间共享数据只能通过写入文件、通过网络套接字发送，或采用 Python 的 pickle 模块进行序列化等方式。

在 Python 3.8 中，multiprocessing 模块提供了 SharedMemory 类，可以在不同的 Python 进城之间创建共享的内存区域。

共享内存提供了进程间传递数据的更快的方式，从而使得 Python 的多处理器和多内核编程更有效率。

共享内存片段可以作为单纯的字节区域来分配，也可以作为不可修改的类似于列表的对象来分配，其中能保存数字类型、字符串、字节对象、None 对象等一小部分 Python 对象。

【示例 1-6】多进程共享内存

```
from multiprocessing import Process
from multiprocessing import shared_memory
share_nums = shared_memory.ShareableList(range(5))
def do_work1(nums):
    for i in range(5):
        nums[i] += 10
    print('do_work1 nums = %s'% nums)

def do_work2(nums):
    print('do_work2 nums = %s'% nums)

if __name__ == '__main__':
    p1 = Process(target=do_work1, args=(share_nums, ))
```

```
    p1.start()
    p1.join()
    p2 = Process(target=do_work2, args=(share_nums, ))
    p2.start()
```

执行结果如图 1-6 所示:

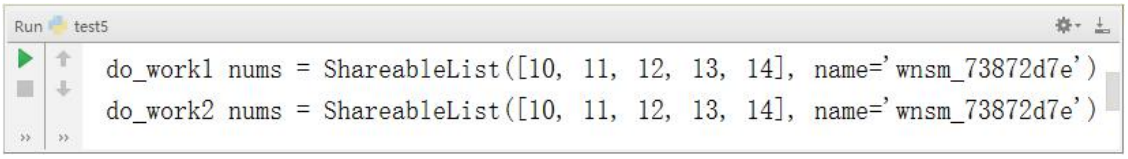


图 1-6 示例 1-6 执行效果图

从上述示例可以看到，do_work1 与 do_work2 虽然运行在两个进程中，但都可以访问和修改同一个 ShareableList 对象。

5） Asyncio 异步交互模式

asyncio.run() 已经从暂定状态晋级为稳定 API。 此函数可被用于执行一个 coroutine（协程对象）并返回结果，同时自动管理事件循环。

【示例 1-7】asyncio.run()执行一个协程对象，并返回结果，同时自动管理事件循环

```
import asyncio
async def main():
    await asyncio.sleep(0)
    return 42
asyncio.run(main())
```

【示例 1-8】Asyncio 异步交互，原来实现

```
import asyncio
async def main():
    await asyncio.sleep(0)
    return 42
loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
try:
    loop.run_until_complete(main())
finally:
    asyncio.set_event_loop(None)
    loop.close()
```

6） 新增和改进的数学和统计功能

Python 3.8 对现有的标准库软件包和模块进行了许多改进。标准库中的数学有了一些新功能。math.prod（）与内置 sum（）类似，但对于乘法乘积。

【示例 1-9】math.prod()的使用

```
import math
result = math.prod((2,3,4,5))
print('math.prod((2,3,4,5))执行结果：',result)
```

执行结果如图 1-7 所示:

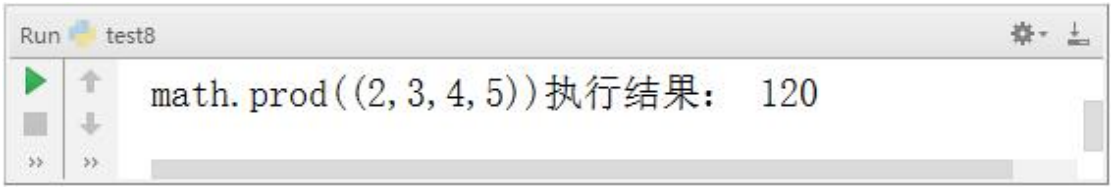


图 1-7 示例 1-9 执行效果图

使用 isqrt（）来找到平方根的整数部分。

【示例 1-10】math.isqrt()的使用

```
import math
result = math.isqrt(25)
print('math.isqrt(25)执行结果：',result)
result = math.isqrt(15)
print('math.isqrt(15)执行结果：',result)
```

执行结果如图 1-8 所示:

图 1-8 示例 1-10 执行效果图

25 的平方根是 5。你可以看到 isqrt（）返回整数结果，而 math.sqrt（）始终返回浮点数。15 的平方根约等于 3.9。isqrt（）将答案截断为一个整数。

表 1-1 统计模块新增功能

函数名称	说明
statistics.fmean（）	计算浮点数的平均值
statistics.geometric_mean（）	计算浮点数的几何平均值
statistics.multimode（）	查找序列中最频繁出现的值
statistics.quantiles（）	计算用于将数据等概率分为 n 个连续区间的切点

【示例 1-11】统计模块新增功能的使用

```
import statistics
data = [11, 2, 33, 14, 2, 26, 33, 38,9]
print('浮点数的平均值：',statistics.fmean(data))
print('浮点数的几何平均值:',statistics.geometric_mean(data))
print('查找序列中最频繁出现的值:',statistics.multimode(data))
print('数据等概率分为 n 个连续区间的切点:',statistics.quantiles(data, n=4))
```

执行结果如图 1-9 所示:

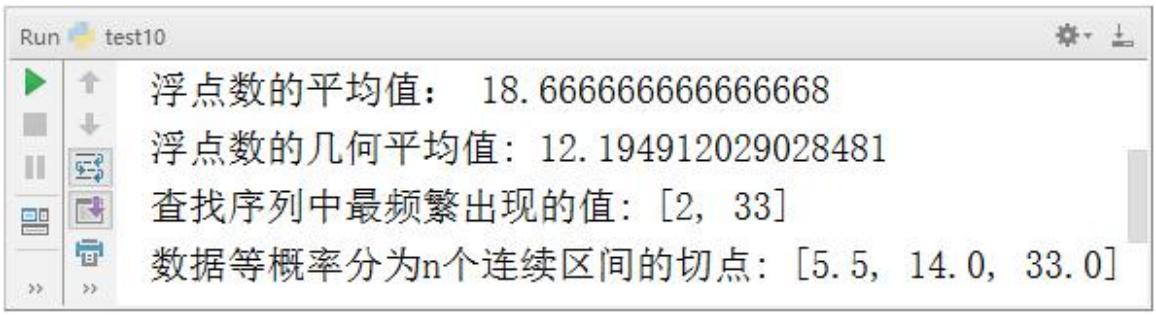


图 1-9 示例 1-11 执行效果图

7） 新增危险语法警告功能

Python 有一个 SyntaxWarning 功能，可以警告不是 SyntaxError 的可疑语法。Python 3.8 添加了一些新功能，可以在编码和调试过程中为你提供帮助。

is 和==之间的区别可能会造成混淆。后者用于检查是否有相等的值，而只有在对象相同时才为 true。Python 3.8 将在应该使用==而不是 is 时发出警告。

【示例 1-12】危险语法警告功能

```
version = "3.8"
print(version == "3.8")
print(version is "3.8")
```

执行结果如图 1-10 所示:

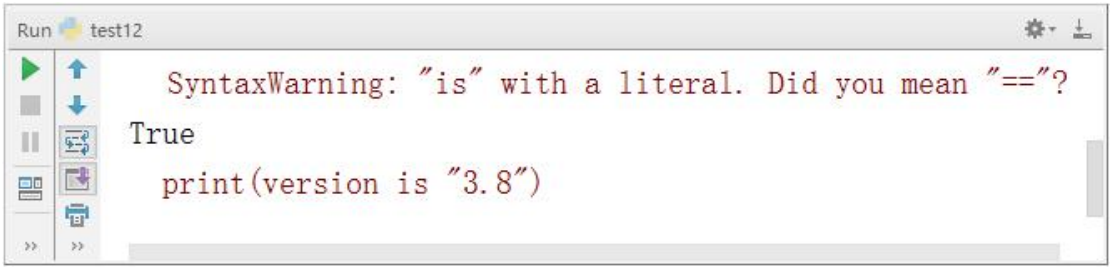


图 1-10 示例 1-12 执行效果图

写长列表时，尤其是垂直格式化时，很容易漏掉逗号。当忘记元组列表中的逗号时会发出让你不解的不可调用元组错误消息。Python 3.8 不仅会发出警告，还会指出实际问题。

【示例 1-13】危险语法警告功能

```
a = [
  (1, 3)
  (2, 4)
]
print(a)
```

执行结果如图 1-11 所示:

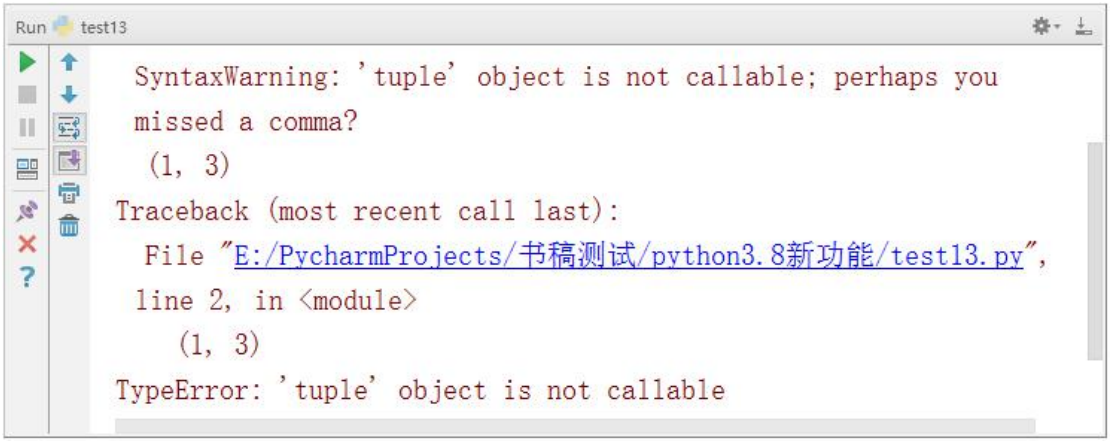


图 1-11 示例 1-13 执行效果图

该警告正确地将丢失的逗号标识为真正的罪魁祸首。

8） 优化

Python 3.8 进行了一些优化，有的让代码运行得更快，有的优化减少了内存占用。示例如下：

【示例 1-14】Python3.7 环境

```
import collections
from timeit import timeit
import sys
Person = collections.namedtuple("Person", "name twitter")
raymond = Person("Raymond", "@raymondh")
# Python 3.7
print('Python3.7 环境下执行时间：')
print(timeit("raymond.twitter", globals=globals()))
print('Python3.7 环境下所占内存：')
print(sys.getsizeof(list(range(20191014))))
```

执行结果如图 1-12 所示:

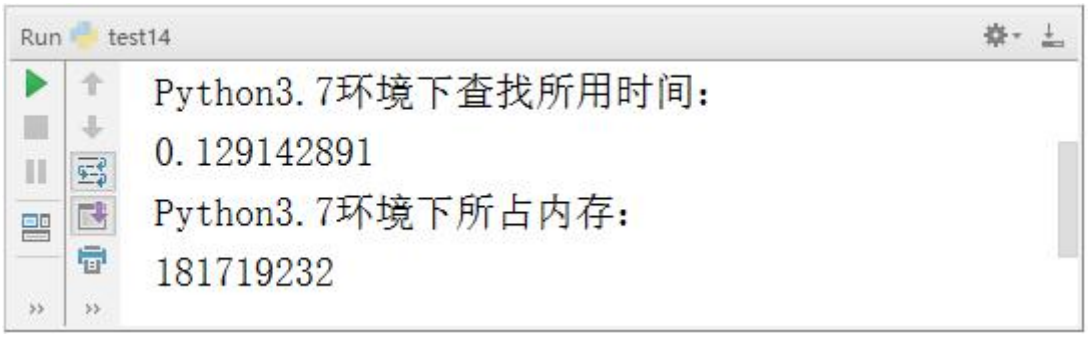


图 1-12 示例 1-14 执行效果图

【示例 1-15】Python3.8 环境

```
import collections
from timeit import timeit
import sys
Person = collections.namedtuple("Person", "name twitter")
raymond = Person("Raymond", "@raymondh")
# Python 3.8
print('Python3.8 环境下执行时间：')
print(timeit("raymond.twitter", globals=globals()) )
print('Python3.8 环境下所占内存：')
print(sys.getsizeof(list(range(20191014))))
```

执行结果如图 1-13 所示:

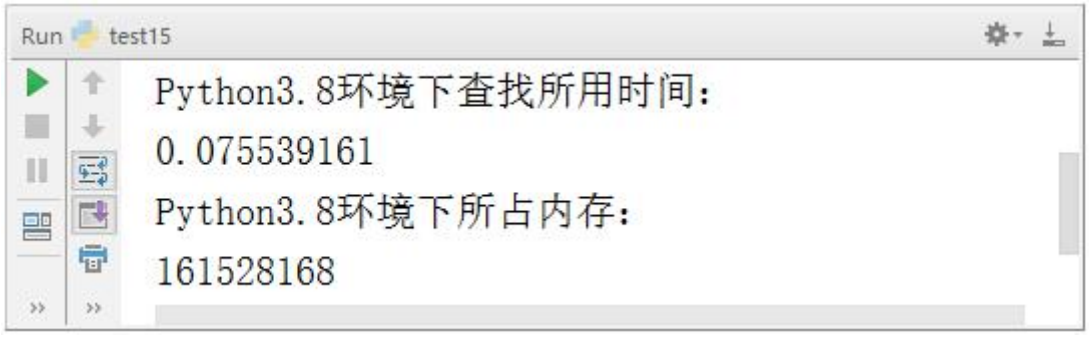


图 1-13 示例 1-15 执行效果图

从示例 14 和示例 15 可以看到，与 Python 3.7 相比，在 Python 3.8 中查找命名元组中的字段要快得多，而且对已知长度的可迭代对象初始化列表时，可以节省一些空间。

附录 2：Python400 集教学视频目录

Python400 集教学视频介绍

《Python400 集》由高淇、张巧英老师历经两年多时间录制而成。整个教学视频以较轻快的风格，从零基础开始讲解。不仅讲解 Python 的基础知识和初步应用，同时注重对底层原理的讲解（内存分析、数据结构、丰富强大的类库）。让大家学习完本视频之后，很好地掌握 Python 语言，同时可以使用 Python 语言进行实际项目的开发。

整套视频以理论与实际相结合，为每个知识点都设计了对应的示例，让 Python 的初学者能够既快速又深刻的理解这些知识点。并且已经被北京大学教授推荐为学习 Python 必看视频。

全套视频分为四季：

- 第一季 【基础篇】Python 基础 （1～115 集）
- 第二季 【提高篇】Python 深入和扩展（116～248 集）
- 第三季 【扩展篇】网络编程、多线程、扩展库（249～278 集）
- 第四季 【高手篇】算法、函数式编程、基于 TensorFlow 人工智能实战（279～402 集）

Python400 集教学视频内容如下：

第一季 1.1 Python 入门

顺序	名称	时长
课时 1	Python 介绍_特性_版本问题_应用范围	17:00
课时 2	Python 下载_安装_配置_第一个 Python 程序	8:04
课时 3	环境介绍_交互模式的使用_IDLE 介绍和使用	8:59
课时 4	IDLE 开发环境的使用_建立 Python 源文件	7:18
课时 5	Python 程序格式_缩进_行注释_段注释	8:18
课时 6	简单错误如何处理_守破离学习法_程序员修炼手册	11:53
课时 7	海归绘图_坐标系问题_画笔各种方法	8:26
课时 8	海龟绘图_画出奥运五环	12:58

第一季 1.2 内置数据类型

顺序	名称	时长
课时 9	程序的构成	7：25
课时 10	对象的基本组成和内存示意图	13:57
课时 11	引用的本质_栈内存和堆内存_内存示意图	4:01
课时 12	标识符_帮助系统的简单实用_命名规则	14:03
课时 13	变量声明_初始化_删除变量_垃圾回收机制	4:25

顺序	名称	时长
课时 14	链式赋值_系列解包赋值_常量	5:13
课时 15	内置数据类型_基本算术运算符	5:55
课时 16	整数_不同进制_其他类型转换成整数	9:59
课时 17	浮点数_自动转换_强制转换_增强赋值运算符	8:39
课时 18	时间表示_unix 时间点_毫秒_微妙_time 模块	8:39
课时 19	多点坐标_绘出折线图_计算两点距离	8:03
课时 20	布尔值_比较运算符_逻辑运算符_短路问题	7:07
课时 21	同一运算符_整数缓存问题	9:56
课时 22	字符串_unicode 字符集_三种创建字符串方式 len()	11:38
课时 23	字符串_转义字符_字符串拼接_字符串复制_input()获得键盘输入	10:17
课时 24	字符串_str()提取字符_replace()替换_内存分析	10:41
课时 25	字符串_切片 slice 操作_逆序	9:49
课时 26	字符串_split()分割_join()合并_join 效率测试	16:08
课时 27	字符串_驻留机制_内存分析_字符串同一判断_值相等判断	7:00
课时 28	字符串_常用查找方法_去除首位信息_大小写转换_排版	12:21
课时 29	字符串_format 格式化_数字格式化操作	13:51
课时 30	可变字符串_io.StringIO	3:34
课时 31	运算符总结_位操作符_优先级问题	12:14

第一季 1.3 序列

顺序	名称	时长
课时 32	列表_特点_内存分析	9:34
课时 33	创建列表的 4 种方式_推导式创建列表	12:03
课时 34	列表_元素的 5 种添加方式_效率问题	10:40
课时 35	列表_元素删除的三种方式_删除本质是数组元素拷贝	10:27
课时 36	列表_元素的访问_元素出现次数统计_成员资格判断	5:43
课时 37	列表_切片 slice 操作	9:45
课时 38	列表_排序_revered 逆序_max_min_sum	7:33
课时 39	列表_二维列表_表格数据的存储和读取	11:53
课时 40	元组_特点_创建的两种方式_tuple()要点	8:25
课时 41	元组_元素访问_计数方法_切片操作_成员资格判断_zip()	5:34
课时 42	元组_生成器推导式创建元组_总结	7:59
课时 43	字典_特点_4 种创建方式_普通_dict_zip_formkeys	10:46

顺序	名称	时长
课时 44	字典_元素的访问_键的访问_值的访问_键值对的访问	5:10
课时 45	字典_元素的添加_修改_删除	6:04
课时 46	字典_序列解包用于列表元组的字典	3:33
课时 47	字典_复杂表格数据存储_列表和字典综合嵌套	10:24
课时 48	字典_核心底层原理_内存分析_存储键值对过程	11:23
课时 49	字典_核心底层原理_内存分析_查找值对象过程	6:22
课时 50	集合_特点_创建和删除_交集并集差集运算	5:05

第一季 1.4 流程控制语句

顺序	名称	时长
课时 51	Pycharm 开发环境的下载安装配置_项目管理	14:02
课时 52	单分支选择结构_条件表达式详解	15:39
课时 53	双分子选择结构_三元运算符的使用详解	5:17
课时 54	多分支选择结构	9:17
课时 55	选择结构的嵌套	14:08
课时 56	while 循环结构_死循环处理	10:38
课时 57	for 循环结构_遍历各种迭代对象_range 对象	13:00
课时 58	嵌套循环	6:12
课时 59	嵌套循环练习_九九乘法表_打印表格数据	8:38
课时 60	break 语句	6:05
课时 61	continue 语句	5:09
课时 62	else 语句	3:56
课时 63	循环代码优化技巧	10:44
课时 64	zip()并行迭代	5:00
课时 65	推导式创建序列_列表推导式_字典推导式_集合推导式_生成器推导式	5:00
课时 66	综合练习_绘制不同颜色的多个同心圆_绘制棋盘	15:07

第一季 1.5 函数和内存分析

顺序	名称	时长
课时 67	函数的基本概念_内存分析_函数的分类_定义和调用	14:00
课时 68	形参和实参_文档字符串_函数注释	11:12
课时 69	返回值详解	8:57
课时 70	函数也是对象_内存分析	7:35

顺序	名称	时长
课时 71	变量的作用域_全局变量_局部变量_栈帧内存分析讲解	14:05
课时 72	局部变量和全局变量_效率测试	5:15
课时 73	参数的传递_传递可变对象_内存分析	8:38
课时 74	参数的传递_传递不可变对象_内存分析	5:05
课时 75	浅拷贝和深拷贝_内存分析	15:23
课时 76	参数的传递_不可变对象包含可变对象子对象_内存分析	10:47
课时 77	参数的类型_位置参数_默认值参数_命名参数	8:36
课时 78	参数的类型_可变参数_强制命名参数	4:17
课时 79	lambda 表达式和匿名函数	10:30
课时 80	eval()函数用法	6:05
课时 81	递归函数_函数调用内存分析_栈帧的创建	21:38
课时 82	递归函数_阶乘计算案例	8:23
课时 83	嵌套函数_内部函数_数据隐藏	11:22
课时 84	nonlocal_global	5:41
课时 85	LEGB 规则	6:07

第一季 1.6 面向对象和内存分析

顺序	名称	时长
课时 86	面向对象和面向过程的区别_执行者思维_设计者思维	14:50
课时 87	对象的进化故事	8:26
课时 88	类的定义_类和对象的关系	15:49
课时 89	构造函数__init__	7:39
课时 90	实例属性_内存分析	9:21
课时 91	实例方法_内存分析方法调用过程_dir()_isinstance()	13:21
课时 92	类和对象	6:38
课时 93	类属性_内存分析创建类和对象的底层	9:26
课时 94	类方法_静态方法_内存分析图示	10:13
课时 95	__del__()析构方法和垃圾回收机制	7:34
课时 96	__call__()方法和可调用对象	7:56
课时 97	方法没有重载_方法的动态性	10:24
课时 98	私有属性	7:00
课时 99	私有方法	5:12
课时 100	@property 装饰器_get 和 set 方法	15:23

顺序	名称	时长
课时 101	面向对象的三大特征说明（封装、继承、多态）	7:22
课时 102	继承	17:13
课时 103	方法的重写	5:03
课时 104	object 根类_dir()	4:54
课时 105	重写__str__()方法	3:25
课时 106	多重继承	4:03
课时 107	mro()	2:46
课时 108	super()获得父类的定义	5:05
课时 109	多态	7:33
课时 110	特殊方法和运算符重载	9:53
课时 111	特殊属性	6:14
课时 112	对象的浅拷贝和深拷贝_内存分析	12:34
课时 113	组合	8:38
课时 114	设计模式_工厂模式实现	9:23
课时 115	设计模式_单例模式实现	13:00

第二季 2.1 文件处理

顺序	名称	时长
课时 116	file 文件操作_操作系统底层关系_写入文件	18:21
课时 117	编码知识_中文乱码问题解决	18:53
课时 118	关闭流要点 1_try 异常管理	11:51
课时 119	关闭流要点 2_with 上下文管理_现场还原	4:47
课时 120	文本文件的读取	8:10
课时 121	enumerate()函数和推导式生成列表_操作每行增加行号	12:48
课时 122	二进制文件的读写_图片文件拷贝	4:02
课时 123	文件对象常用方法和属性总结_seek()任意位置操作	9:25
课时 124	使用 pickple 实现序列化和反序列化_神经元记忆移植	14:18
课时 125	CSV 文件的读取_写入	12:49
课时 126	os 模块_调用操作系统可执行文件_控制台乱码问题	8:25
课时 127	os 模块_获得文件信息_创建文件夹_递归创建	22:05
课时 128	os.path 模块_常用方法	18:24
课时 129	os 模块_使用 walk 遍历	9:05
课时 130	shutil 模块_文件和目录拷贝	8:03

顺序	名称	时长
课时 131	shutil 和 zipfile 模块_压缩和解压缩	5:56
课时 132	递归算法原理_阶乘计算	14:04
课时 133	递归算法原理_目录树结构展示	9:07

第二季 2.2 异常

顺序	名称	时长
课时 134	异常本质_调试核心理念	26:25
课时 135	try_except 基本结构	16:20
课时 136	try 多个 except 结构	8:32
课时 137	else 结构	3:54
课时 138	finally 结构	9:03
课时 139	常见异常汇总和说明	8:38
课时 140	with 上下文管理	4:39
课时 141	trackback 模块的使用_异常写入日志文件	6:11
课时 142	自定义异常类_raise 抛出异常	10:15
课时 143	Pycharm 的调试模块	22:16

第二季 2.3 模块

顺序	名称	时长
课时 144	模块化编程理念_什么是模块_哲学思想	11:59
课时 145	模块化编程的流程_设计和实现分离	19:26
课时 146	模块导入_import 和 from_import 语句详解和区别	11:46
课时 147	import 加载底层原理_importlib 模块	7:38
课时 148	包的概念和创建包和导入包	12:10
课时 149	包的本质和 init 文件_批量导入_包内引用	8:29
课时 150	sys.path 和模块搜索路径详解	12:07
课时 151	模块的本地发布_模块的安装	9:11
课时 152	PyPI 官网_远程上传和管理模块_PIP 方式安装模块	10:24

第二季 2.4 GUI 编程

顺序	名称	时长
课时 153	GUI 编程和 tinkter 介绍_第一个 GUI 程序	19:18
课时 154	PEP8 编码规范_窗口大小和位置	7：06

顺序	名称	时长
课时 155	GUI 编程整体描述_常用组件汇总	7:39
课时 156	GUI 程序的经典面向对象写法	22:41
课时 157	Label 组件_tkinter 中图像正确显示全局变量写法	16:49
课时 158	options 选项详解_底层源码分析和阅读_可变参数和运算符重载复习	15:31
课时 159	Button_anchor 位置控制	10:33
课时 160	Entry_StringVar_登录界面设计和功能实现	18:51
课时 161	Text 多行文本框详解_复杂 tag 标记	12:40
课时 162	Radiobutton_Checkbutton 详解	8:17
课时 163	Canvas 画布组件	8:09
课时 164	Grid 布局管理器详解	7:43
课时 165	计算器软件界面的设计	17:27
课时 166	Pack 布局管理器_钢琴软件界面设计	6:38
课时 167	Place 管理器_绝对位置和相对位置	6:28
课时 168	扑克游戏界面设计_增加事件操作	14:54
课时 169	事件机制和消息循环原理_鼠标事件_键盘事件_event 对象	18:32
课时 170	lambda 表达式_事件传参应用	7:41
课时 171	三种事件绑定方式总结	4:23
课时 172	optionmenu 选项菜单_scale 滑块	7:34
课时 173	颜色框_文件选择框_读取文件内容	9:01
课时 174	简单对话框_通用消息框_ttk 子模块问题	5:59
课时 175	主菜单_上下文菜单	7:44
课时 176	【记事本项目 01】_打开和保存修改文件的实现	11:27
课时 177	【记事本项目 02】_新建文件_背景色改变_快捷键功能	13:02
课时 178	【记事本项目 03】python 项目打包成 exe 可执行文件	4:04
课时 179	【画图项目 01】_界面实现	13:35
课时 180	【画图项目 02】_绘制直线_拖动删除上一个图形	12:55
课时 181	【画图项目 03】_箭头直线_矩形绘制	5:36
课时 182	【画图项目 04】_画笔和橡皮檫实现	7:48
课时 183	【画图项目 05】_清屏_颜色框_快捷键处理	8:33

第二季 2.5 坦克大战

顺序	名称	时长
课时 184	Pygame 模块的安装	7:25

顺序	名称	时长
课时 185	面向对象分析项目需求	6:42
课时 186	坦克大战项目框架搭建	6:31
课时 187	加载主窗口	12:05
课时 188	坦克大战之事件处理	10:24
课时 189	左上角文件的绘制	16:58
课时 190	加载我方坦克	15:16
课时 191	我方坦克切换方向_移动	7:00
课时 192	我方坦克移动优化	8:30
课时 193	我方坦克优化 2	13:11
课时 194	加载敌方坦克	15:31
课时 195	敌方坦克随机移动	11:15
课时 196	完善子弹类	12:00
课时 197	我方坦克发射子弹	7:24
课时 198	子弹移动	10:35
课时 199	子弹消亡及数量控制	8:51
课时 200	敌方坦克发射子弹	12:07
课时 201	我方子弹与敌方坦克的碰撞	12:47
课时 202	实现爆炸效果	13:13
课时 203	我方坦克的消亡	10:42
课时 204	我方坦克无限重生	7:18
课时 205	加载墙壁	12:40
课时 206	子弹不能穿墙	8:09
课时 207	坦克不能穿墙	10:14
课时 208	敌我双方坦克发生碰撞	10:12
课时 209	音效处理	9:43

第二季 2.6 数据库编程输出

顺序	名称	时长
课时 210	操作 SQLite3 创建表	10:43
课时 211	操作 SQLite3 数据库插入数据	11:22
课时 212	操作 SQLite3 数据库查询数据	8:05
课时 213	操作 SQLite3 数据库修改_删除数据	8:40
课时 214	MySQL 数据库的下载	4:25

顺序	名称	时长
课时 215	MySQL 数据库的安装	5:54
课时 216	PyMySQL 模块的安装	5:18
课时 217	操作 MySQL 数据库创建表	8:36
课时 218	操作 MySQL 数据库插入数据	10:57
课时 219	操作 MySQL 数据库查询数据	9:31
课时 220	操作 MySQL 数据库修改_删除数据	8:00

第二季 2.7 numpy

顺序	名称	时长
课时 221	numpy 是什么及 numpy 模块的安装	8:34
课时 222	array 进行创建数组	8:39
课时 223	arange 创建数组	7:34
课时 224	随机数创建数组 1	11:44
课时 225	随机数创建数组 2	10:30
课时 226	ndarray 对象的属性	14:01
课时 227	其他方式创建数组	15:37
课时 228	一维数组的切片索引	8:35
课时 229	二维数组的切片和索引	18:59
课时 230	数组的复制	8:28
课时 231	修改数组的维度	12:14
课时 232	数组的拼接	15:32
课时 233	数组的分割	18:55
课时 234	数组的转置	10:54
课时 235	函数 1	19:19
课时 236	函数 2	11:11

第二季 2.8 matplotlib

顺序	名称	时长
课时 237	基本绘制图形	12:26
课时 238	设置样式	5:44
课时 239	绘制曲线	9:05
课时 240	subplot 的使用	8:03
课时 241	绘制散点图	19:28

顺序	名称	时长
课时 242	绘制不同样式不同颜色的线条	10:44
课时 243	绘制柱状图	9:17
课时 244	bar 及 barh 函数的使用	14:50
课时 245	柱状图使用实例	16:23
课时 246	绘制饼状图	10:22
课时 247	绘制直方图	8:56
课时 248	绘制等高线图和三维图	10:57

第三季 3.1 并发编程

顺序	名称	时长
课时 249	多任务的概念	8:04
课时 250	创建子进程并调用	14:24
课时 251	join 方法的使用	9:34
课时 252	属性的使用_多任务的创建	14:16
课时 253	使用继承方式创建进程	8:23
课时 254	进程池的使用	14:06
课时 255	多个进程之间数据不共享	6:40
课时 256	队列常用方法使用	10:57
课时 257	进程之间通信	9:41
课时 258	进程和线程的区别	6:01
课时 259	thread 创建线程	11:18
课时 260	threading 模块创建线程	18:13
课时 261	线程之间共享全局变量	17:46
课时 262	互斥锁	9:06
课时 263	线程同步的使用	10:08
课时 264	生产者消费者模块	12:51
课时 265	ThreadLocal 的使用	9:03

第三季 3.2 网络编程

顺序	名称	时长
课时 266	IP 地址_端口	10:43
课时 267	网络通信协议	6:20
课时 268	TCP 协议_UDP 协议	7:09

顺序	名称	时长
课时 269	UDP 发送数据	8:40
课时 270	UDP 接收数据	13:27
课时 271	UDP 使用多线程实现聊天	11:26
课时 272	TFTP 文件下载器过程及格式介绍	23:02
课时 273	TFTP 下载器客户端实现	18:55
课时 274	TCP 通信	9:15
课时 275	TCP 服务器端接收 数据	7:27
课时 276	TCP 模拟 QQ	12:21
课时 277	TCP 多线程完成聊天	23:51
课时 278	TCP 多线程聊天优化	8:12

第四季 4.1 算法

顺序	名称	时长
课时 279	算法的概念	26:02
课时 280	第二次获取值	8:02
课时 281	时间复杂度	13:27
课时 282	最坏时间复杂度_常见时间复杂度与大小关系	10:12
课时 283	空间复杂度	7:06
课时 284	排序算法的稳定性	3:57
课时 285	冒泡排序	28:00
课时 286	选择排序	12:55
课时 287	选择排序_时间复杂度_稳定性	5:32
课时 288	插入排序	13:35
课时 289	选择排序_时间复杂度	6:06
课时 290	快速排序的思想	8:18
课时 291	快速排序实现	11:35
课时 292	快速排序_时间复杂度	9:49
课时 293	归并排序思想	10:28
课时 294	归并排序实现	14:30
课时 295	归并排序_时间复杂度	6:34
课时 296	顺序查找法	5:43
课时 297	二分法查找	14:29

第四季 4.2 数据结构

顺序	名称	时长
课时 298	数据结构的引入	12:15
课时 299	顺序表	11:26
课时 300	测试 list 列表中 insert 和 append 的执行速度	9:05
课时 301	链表的引入	8:20
课时 302	单链表及节点的定义	14:14
课时 303	单链表_是否为空_计算长度方法的实现	16:44
课时 304	单链表_查找_遍历方法的实现	6:58
课时 305	单链表_头部_尾部添加节点	20:14
课时 306	单链表指定位置插入元素	15:24
课时 307	单链表删除节点	14:39
课时 308	链表与顺序表的对比	6:18
课时 309	双向链表节点定义	6:57
课时 310	双向链表添加节点	12:40
课时 311	双向链表指定位置插入节点	7:29
课时 312	双向链表其它操作	13:49
课时 313	栈的实现	10:38
课时 314	队列的实现	7:17
课时 315	树的概念	21:05
课时 316	二叉树的概念	6:59
课时 317	二叉树节点定义_添加节点	14:26
课时 318	广度优先遍历	9:09
课时 319	深度优先遍历	18:58

第四季 4.3 函数式编程和高阶函数

顺序	名称	时长
课时 320	高阶函数概念	13:42
课时 321	高阶函数 map 的使用	16:05
课时 322	高阶函数 reduce 的使用	12:38
课时 323	高阶函数 filter 的使用	11:55
课时 324	高阶函数 sorted 的使用	12:43
课时 325	匿名函数	14:55
课时 326	闭包定义及使用	10:43

顺序	名称	时长
课时 327	使用闭包求两点之间的距离	9:43
课时 328	闭包的特殊用途	16:54
课时 329	装饰器的基本使用	13:29
课时 330	多个装饰器的使用	9:32
课时 331	带参数的装饰器	11:51
课时 332	通用装饰器	7:04
课时 333	偏函数	8:09

第四季 4.4 正则表达式

顺序	名称	时长
课时 334	正则表达式的概念	8:18
课时 335	match 方法的使用	10:03
课时 336	常用字符的使用	12:20
课时 337	重复数量限定符	15:23
课时 338	重复数量限定符的使用	16:21
课时 339	原生字符串	5:48
课时 340	边界字符的使用	9:17
课时 341	search 方法的使用	12:29
课时 342	择一匹配符和列表的使用差异	9:44
课时 343	正则表达式分组的使用	17:01
课时 344	其他函数的使用	18：12
课时 345	贪婪模式和非贪婪模式	6:44

第四季 4.5 pillow 图像处理

顺序	名称	时长
课时 346	Image 打开显示图片	12:10
课时 347	Image 完成图像混合	8:19
课时 348	图像的缩放_复制_剪切_粘贴	12:23
课时 349	图像的旋转_分离合并	10:48
课时 350	图像滤镜	10:22
课时 351	图片合成	11:26
课时 352	调整图像色彩_亮度	17:26
课时 353	ImageDraw 绘图二维图像	15:36

顺序	名称	时长
课时 354	ImageFont 的使用	12:09
课时 355	绘制十字	6:51
课时 356	绘制验证码	15:29
课时 357	绘制九宫格	11:35
课时 358	将图片中黄色修改为红色	9:56
课时 359	读取图片实例	21:28

第四季 4.6 人脸识别

顺序	名称	时长
课时 360	加载图片	10:51
课时 361	将图片灰度转换	6:40
课时 362	修改图片尺寸	8:18
课时 363	绘制矩形_圆	9:38
课时 364	人脸检测	13:00
课时 365	检测多张人脸	14:37
课时 366	检测视频中人脸	8:49
课时 367	训练数据	25:06
课时 368	人脸识别	10:13

第四季 4.7 语音识别

顺序	名称	时长
课时 369	原生字符串	5:54
课时 370	边界字符的使用	9:13
课时 371	search 方法的使用	8:10

第四季 4.8 协程_异步 IO

顺序	名称	时长
课时 372	协程的概念	12:03
课时 373	yeild 的使用	14:45
课时 374	yield 实现生产者消费者	9:21
课时 375	同步和异步的概念	10:18
课时 376	定义协议	9:03
课时 377	创建任务 task	8:14

顺序	名称	时长
课时 378	绑定回调	7:57
课时 379	阻塞和 await	7:46
课时 380	asyncio 实现并发	9:51
课时 381	协程嵌套	15:25
课时 382	协程停止	16:04

第四季 4.4 正则表达式

顺序	名称	时长
课时 383	神经网络原理	12:52
课时 384	激活函数	7:45
课时 385	TensorFlow 简介	10:46
课时 386	查看默认图	8:36
课时 387	自定义图	6:48
课时 388	使用 TensorBoard 将图可视化	8:36
课时 389	会话 Session	11:37
课时 390	feed_dict 的使用	10:59
课时 391	张量的创建	14:59
课时 392	修改张量形状	16:45
课时 393	矩阵运算	13:38
课时 394	变量的定义	9:11
课时 395	线性回归案例实现	24:12
课时 396	增加变量显示	10:37
课时 397	增加命名空间	6:58
课时 398	保存读取模型	9:23
课时 399	加载 mnist 数据集	22:10
课时 400	手写数字识别	18:13
课时 401	手写数字识别_增加变量显示_命名空间	11:36
课时 402	手写数字识别_保存读取模型	8:45