Pithon

南京大学　电子科学与工程学院

撰稿人：方元

2020 年 2 月

目录

插图

表格

1

智能系统概述

1.1 智能系统的应用

智能这个术语在很多应用领域都能看到, 包括智能家居、智能交通、人工智能等等。对于智能一词, 并没有单一的确定的定义。经济合作与发展组织 (**O**rganisation for **E**conomic **C**o-operation and **D**evelopment, OECD) 给出过一个定义 “一个具有学习能力的应用或服务, 能将这些情况综合告知其他设备和用户。然后这些设备及用户可以改变自己的行为来更好地适应这个情况。这就意味着关于情况的数据需要被产生、发送、处理、纠错、解释、适配及以有意义的方式显示, 然后据此作出动作。”

智能系统一般被理解为一种可以对环境进行感知, 并可以做出最优化的响应的系统。下面是一些典型的智能系统的例子。

- 智能电网。它利用传感器、通信网络和自动化来增强电力配电网络, 实行更有效的管理。它可以提高能源效率, 有助于改善环境、降低成本, 提高网络的可靠性和对局部故障的响应能力, 以及可以自动收集测量数据。
- 智慧农业。它通过使用监视和自动化系统来增强对农作物、土地和牲畜的管理：控制温度和湿度, 针对局部气候条件实施农田灌溉及排水系统的控制, 以及监视牲畜的健康状况并及时报告和提醒。
- 智慧城市。它的目标是集成并管理城市的各种系统, 以实现可持续的发展和经济的增长, 并保障市民的生活质量。这可能包括交通、建筑、设施等等, 也包括在公共区域提供的互联网服务、基础设施的监测和维护、政府服务、保安和疾病管理。
- 智能交通。它有效地集成先进的信息技术、通讯技术、传感技术、控制技术及计算机技术, 运用于整个交通运输管理体系, 从而建立起一种在大范围内及全方位发挥作用的、实时、准确及高效率的综合运输和管理系统。智能交通系统包括公共和私人交通、货运和客运。智能交通的目的可以包括交通流量的管理 (使拥堵最小化)、降低排放, 及更快更可靠连接会带来的可观的经济效益。它是智慧城市的重要组成部分。
- 智能建筑和智能家居。它广泛利用传感器采集的各种数据, 根据监测到的情况来调整建筑的运作, 比如灯光、温度和通风系统。它的目标是建造一个舒适的人居环境, 降低成本和能源消

耗。

- 智能网络。所有智能系统都会需要通信基础设施的支持。由于各种智能系统背景的多样性, 智能网络的形式也会有各种形式。例如家居和建筑的网络是局部化的、而且大多数是室内的, 农业、交通则是大规模的、室外的。智能网络并不是一个明确的技术, 可能是有线连接, 也可能是无线网络, 它是可适配、可扩展、并具备分布式计算能力的有智慧的网络。

信息技术、网络技术和通信技术是构成各种智能系统的基础。

1.2 人工智能的发展

人工智能 (**A**rtificial **I**ntelligence, AI) 是比智能系统更高层次的概念, 它指由人制造出来的机器所表现出来的智能。通常人工智能是指通过计算机程序实现的人类智能技术。

人工智能始终伴随着计算机的发展而发展。电子计算机的发明, 延伸了人脑的功能。初期的研究人员尝试探索人类智能是否能简化成符号处理。这一时期的人工智能直接模仿人类进行逐步的推理, 在一些有限规模的决策系统中取得了很大的成功。1996—1997 年的深蓝计算机对卡斯帕罗夫的人机大战可算是一个典型的案例。

然而人类解决问题的模式通常是用最快捷、直观的判断, 而不是靠有意识的、一步一步的推导。人工神经网络(**A**rtificial **N**eural **N**etwork, ANN) 研究试图模拟人的大脑结构重现这种技能。ANN 是上世纪 70 年代提出的算法, 它采用数学模型模拟神经信号的传递方式。每个神经元的结构非常简单, 但组成网络后却能破解传统串行算法所面临的困局, 在模式识别、目标分类问题上取得了相当的成功。

虽然深蓝在二十多年前就已经战胜了人类的象棋高手, 但当问题的复杂性超过一定规模时, 计算机会需要天文量级的存储器或是运算时间, 因此在面对围棋这样的大规模变局却没有更好的办法。随着计算机性能的提高, 特别是 DeepMind 提出了深度学习算法之后, 极大地推动了人工智能的发展。2016 年, 载有深度神经网络算法的 AlphaGo 战胜了韩国围棋棋王李世乭, 次年 AlphaGo Master 又战胜了世界排名第一的柯洁。随后问世的 AlphaGoZero 和 AlphaZero 将人工智能推上了新的高度。和传统的对弈软件不同, AlphaZero 不存储任何人类棋局作为数据库, 完全是从零开始自我学习。经过数十小时的训练, 已经把人类棋手远远甩在了身后。

当前人工智能的研究已经形成了多个研究领域, 机器学习、计算机感知、模式识别、自然语言处理、机器人是其中一些热门的研究方向。

2

Linux 操作系统

2.1 操作系统概述

操作系统是管理计算机硬件与软件资源的核心软件。它位于计算机硬件与用户之间, 是两者沟通的桥梁。用户可以通过操作系统的用户界面输入命令, 操作系统则对命令进行解释, 驱动硬件设备, 实现用户要求。现代操作系统通常应提供以下的功能：

- 进程管理
- 存储管理
- 文件系统
- 设备驱动

Unix、Linux、Windows 是通用计算机使用的典型的操作系统。由于没有特定的应用需求, 通用操作系统为了适应更广泛的应用, 需要支持更多的硬件与软件, 需要针对所有的用户体验, 对系统进行更新。通用计算机的操作系统一般是分时操作系统, 它更关心系统资源分配的合理性而不是实效性。

2.2 Linux 简介

Linux 是一种遵循 GPL 版权协议的操作系统。1991 年 4 月, 当时还在芬兰赫尔辛基大学读研究生的 Linus Torvalds, 在学习计算机操作系统课程时, 出于兴趣, 编写了这个操作系统, 并向互联网公开。系统一经发布, 迅速受到了全世界程序员的热烈响应, 自此一发不可收拾, 直至发展到今天, 成为有超级影响力的操作系统。成千上万的的程序员为 Linux 贡献了代码。仅 Linux 内核的维护人员名单就长达四千多行, 更不要说大量的应用软件开发人员了。

归纳起来, Linux 系统有以下这些特点和优势：

1. 源代码开放

Linux 内核遵循开源版权协议, 各种发行版使用的绝大多数软件都是开源的。开源软件不仅为开发提供了方便, 更重要的是安全性的保障。这种情况下, 很难有恶意代码的容身之地。世界各地的软件工程师和 Linux 用户都在热心地为开源社区提出建议。一旦出现 bug, 可以很快得到修正。稳定、可靠的 Linux 操作系统成为绝大多数服务器的首选。

2. 可靠性、稳定性

   Linux 运行于保护模式, 内核态和用户态地址空间分离, 对不同用户读、写权限控制, 带保护的子系统及核心授权, 多种安全技术措施共同保障系统安全。再加上良好的用户使用习惯, 使得一个系统能够长期稳定地运行。

3. 良好的可移植性

   Linux 软件开发遵循 POSIX 标准, 在不同平台之间不经修改或只需要很少的修改就可以直接使用。Linux 内核支持包括 Intel、Arm、PowerPC 等在内的数十种处理器架构和上百种硬件平台, 可以移植到几乎所有的主流处理器平台。

4. 设备独立性

   Linux 系统中, 有 “一切皆是文件” 之说, 所有设备也统一被当作文件看待。操作系统核心为每个设备提供了统一的接口调用。应用程序可以像使用文件一样, 操作、使用这些设备, 而不必了解设备的具体细节。设备独立性的关键在于内核的适应能力。它带来的好处是, 用户程序和物理设备无关, 系统变更外设时程序不必修改, 提高了外围设备分配的灵活性, 能更有效地利用外设资源。

5. 多种人机交互界面

   Linux 从初期单调的字符界面, 发展到如今丰富的图形化人机接口和功能强大的命名行方式并存, 可以满足不同系统资源的需求。传统的命令行界面利用 shell 强大的编程能力, 即使在不具备图形能力的计算机系统上也能胜任服务器的工作。Linux 的图形桌面有多种选择, 可以充分体现用户的个性化设置。

   这里需要说明, 命令行方式只是一种选择, 而不是 Linux 的本性。对于熟练的计算机用户, 它提供了一种更高效的使用手段。作为消费型操作系统, Linux 桌面系统完全满足普通用户的需求。

6. 多用户、多任务支持

   Unix 设计之初就是为了满足多人使用计算机的需求。Linux 继承了 Unix 操作系统的这一特性, 它天生是一个多用户操作系统。除了像其他运行在个人计算机的操作系统一样, 可以为每个用户分配独立的系统资源以外, 还可以让多人同时使用一台计算机。这一点, 与专为个人计算机设计的操作系统不一样。

   多任务是现代操作系统的一个重要特性, 它是指计算机同时执行多个程序, 并且各个程序的运行互相独立。Linux 系统调度为每一个任务独立分配处理器资源和存储空间, 相互之间不会受到干扰。某个任务的失败一般不会影响到其他任务。这为系统的可靠性提供了保障。

7. 完善的网络功能

   Linux 操作系统从 Unix 演变而来, 技术上具备 Unix 操作系统的全部优势。Unix 最早实

现了 TCP/IP 网络协议, Linux 继承了这一特点, 支持完整的 TCP/IP 客户与服务器功能, 具有强大的网络通信能力。

8. 多种文件系统支持

   Linux 通过虚拟文件系统层实现对不同文件系统的支持, 几乎可以识别目前所有已知的磁盘分区格式, 软件不经修改就可以对不同分区的文件进行读写操作。

9. 便捷的开发和维护手段

   Linux 各发行版提供了多种编程语言开发工具, 此外还大量地使用了 脚本语言, 除了方便编程以外, 还为系统的可维护性建立了基础。Linux 的各种服务都是通过脚本程序维护的, 服务器的功能也是通过脚本文件配置的, 包括系统的启动过程, 也都是通过脚本程序完成。管理员可以使用任何自己熟悉的文本编辑工具管理计算机。

时直今日, Linux 操作系统已遍及各种档次的计算机系统, 小到可穿戴设备、大到超级计算机和集群计算, 无不展现出它的优势。除个人计算机桌面系统以外, 在它所能涉足的领域几乎都占据主导地位。

Linux 社区非常活跃, 差不多每 6 个月就会发布一个新的内核版本。2019 年 10 月发布到 Linux 5.4 版。Linux 是自由软件 (free software, 取其 “自由” 之意)。由于英语中的 free 有 “自由” 和 “免费” 两个主要意思, 故而也有人解释为免费软件, 但自由软件的开发人员特别强调了自由和免费的区别, 单纯的免费软件是 freeware。由于版权协议的规定, Linux 操作系统中几乎所有的软件都可以免费地获得源代码, 任何人在遵守版权协议的基础上都可以对源代码进行修改和再发布, 从而导致用户可以免费地获得和使用 Linux 软件这样的结果, 但并不意味着开发者不能从中收取费用。

Linux 自问世以来, 以其稳定性和可靠性迅速占领服务器市场。几乎所有的超级计算机都使用 Linux 操作系统[1]。在手机和平板电脑这类移动设备上, Android 操作系统占 60% 以上的市场份额 [2], 超过 iPhone 的 iOS 操作系统。

Linux 内核支持数十种处理器架构, 它也是树莓派等卡片式计算机操作系统的首选。

2.3 Linux 使用

目前 Linux 的桌面发行版包含了完善的图形用户界面 (**G**raphical **U**ser **I**nterface, GUI) 环境, 它的组成形式与 Windows 系统基本相同, 有的只是操作习惯的差别。作为普通的计算机用户, 鼠标可以完成绝大部分的日常工作。但作为软件开发人员, 仅仅会使用图形界面工具显然是远远不够的, 至少效率不够高, 而且在开发目标系统上很可能就没有图形界面。

Linux 系统提供了功能强大的字符操作界面, 习惯上把这种界面称为命令行方式 (**C**ommand **L**ine **I**nterface, CLI)。命令行方式需要掌握一些基本操作方法, 比直接使用图形界面操作复杂, 由此给人产生了 “Linux 系统难用” 这样的印象。这里需要分清普通计算机用户和

[1]在 www.top500.org 网站公布的全球最强大的超级计算机前 500 强中, 2017 年榜单上有 498 台使用 Linux, 而在 2018 年 6 月的榜单上, 这个数字变成了 500。

[2]Android 系统使用 Linux 内核。由于版权协议和一些技术细节方面的原因, Android 操作系统未被归入 Linux 的发行版

计算机开发人员。普通计算机用户需要的是消费型操作系统, 即使没有命令行工具, Linux 也完全满足这部分用户的要求, 而命令行方式则为开发人员提供了更灵活的手段和更完善的开发功能, 使得很多在图形界面下不方便甚至不可能的任务在这里得以顺利完成。

2.3.1 命令行的格式

命令行界面通常由一个终端提供。终端程序又叫 shell。在终端提示符 "$" 或 "#" 下键入的一行字符即构成了一条命令。"$" 是普通用户权限的提示符, "#" 是超级用户的提示符。一些改变系统设置的命令, 普通用户的执行时会受到一定的限制。

一个完整的命令行由命令、选项、参数三部分构成:

```
$ command [options] [arguments]
```

命令`command` 表示你要进行何种操作, 选项`options` 表示如何操作, 参数`arguments` 表示这条命令的操作对象是什么。有些命令可以没有选项和参数, 有的命令可能有不止一个选项或参数, 有的选项要求后面紧跟它的参数。命令、选项、参数之间用空格分隔。选项前面通常有一个或两个 "-" (取决于不同选项风格), 它是选项的不可分割部分, 中间不能加空格。

Linux 命令行中的所有字母都严格区分大小写。

例如, 要强制删除一个名为 hello.py 的文件:

```
$ rm -f hello.py
```

删除文件的命令是 `rm` (**rm**ove), 强制删除的选项是 "-f" (**f**orce 的首字母)。

2.3.2 用户和权限

我们通过一个列表文件清单的命令`ls` (**ls**t) 解释用户和权限, 该命令一个常用的选项是 "-l", 表示列出文件的详细信息; 另一个不常用的选项 "-i" 用于列出 inode。在文件系统上, inode 表示文件存储的物理信息。如果选项对应的参数不会混淆, 多个选项可以合并在一个 "-" 之后。

```
$ ls -li
4850428 drwxr-xr-x 2 pi   pi    4096 2月   8 11:35 graphics
4850423 -rw-r--r-- 2 pi   pi 1524296 2月   8 12:03 Pithon.pdf
4850317 -rwxr-xr-x 1 pi   pi    5315 2月   8 14:21 hello.py
4850423 -rw-r--r-- 2 pi   pi 1524296 2月   8 12:03 lecture.pdf
4850510 drwxr-xr-x 2 pi   pi    4096 2月   4 16:00 program
4850553 drwxr-xr-x 2 pi   pi    4096 2月   8 13:16 text
```

清单中的每一行列出了一个文件的完整属性, 它包括 8 个字段: inode、文件类型和操作权限、链接数、所属用户, 所属组别、文件大小、最后修改日期 (日期和时间)、文件名。

文件类型和操作权限包含 10 个字符, 第一个字符表示文件类型。表2.1 是字符和文件类型的对应关系。

表 2.1 文件类型的表示

符号	意义	常见来源
-	普通文件	由编辑工具创建或应用软件生成
b	块设备文件	由命令 mknod 创建
c	字符设备文件	由命令 mknod 创建
d	目录	由命令 mkdir 创建
l	符号链接	由命令 ln 选项 -s 创建
p	管道文件	由命令 mkfifo 或 mknod 创建
s	套接字文件	由系统调用 bind() 创建

文件类型和操作权限的后 9 个字符分成三段, 每段 3 个字符, 分别对应用户本人、同组用户和其他人对该文件的访问权限。访问权限由读允许、写允许和运行允许组成, 分别用字母 r (**r**ead)、w (**w**rite)、x (e**x**ecute) 表示。当不具备某个权限时, 相应位置上的字母用 "-" 代替。

Linux 操作系统的分组策略除了方便项目合作以外, 它也是保证系统安全的第一道屏障。所有用户的所有操作都与其权限直接相关。超越用户权限的操作是无效的。Linux 系统的超级用户 root 拥有系统的最高权限。普通人在使用计算机时应尽量避免用 root 账户登录。

2.3.3 目录操作

以下是常用的目录操作命令：

- 创建新目录 mkdir (**m**a**k**e **dir**ectory)

```
$ mkdir dir1 dir2 dir3
$ mkdir -p newdir/dir2
```

当创建的目录的上层目录尚不存在时, 选项 "-p" (**p**arents) 表示连同父目录一起创建。

- 删除空目录 rmdir (**r**e**m**ove **dir**ectory)

```
$ rmdir dir1 newdir/dir2
```

此命令只能删除空目录, 非空目录不允许轻易删除, 以避免无意中丢失数据。

- 转移工作目录 cd (**c**hange **d**irctory)

```
$ cd /home/student/Desktop
```

此命令不带参数运行时, 将回到用户的主目录。在目录操作中, 符号 "~" 也表示用户主目录。

- 打印当前目录 pwd (**p**rint current **w**orking **d**irctory)。此命令不带参数。

Linux 系统中, 目录层次之间使用 "/" 分隔。最顶层目录被称为根目录。除根目录以外, 所有目录都含有两个特殊的目录："." 和 "..", 前者表示当前目录, 后者表示父目录。

2.3.4 文件操作

下面是一些常用的文件操作命令。

- 复制文件 cp (**cp**y)

  单个文件的复制命令形式是

```
$ cp [选项] 源文件 目标文件
```

  如果目标文件已存在, 复制时会将目标文件覆盖, 用复制的源文件替换。为避免误操作, 复制时可以加一个选项 “-i” (**i**nteractive), 表示交互操作方式: 当目标文件存在时, 会提示用户确认覆盖或者放弃。

  多个文件复制时, 最后一个参数必须是目录, 复制的结果是将前面所列的文件复制到目标目录下。如果包含了目录复制, 应给出选项 “-r” (**r**ecursive), 表示递归操作, 否则目录和目录里的内容都将被跳过。

  对文件名具有某种特征的一组文件进行操作时, 为避免输入烦琐和遗漏, Linux 系统使用一种被叫做通配符的符号系统来代替。表2.2是在文件操作中常用的通配符。

表 2.2 常用的文件操作通配符

符号	含义	举例
?	匹配任意一个字符	“???” (三个字符的文件名)
*	匹配任意个字符 (. 起头的除外)	“*.c” (以.c 为后缀的文件名)
[ ]	匹配列表中的字符	“*[Aa]” (以字母 A 或 a 结尾的文件名)
		“[A-Z]” (任意大写字母开头的文件)
^、!	不包含	“[^0-9]* (不以数字开头的文件名)
{ }	匹配括号中的列表	“*.c,[a-z]*” (所有.c 文件和小写字母开头的文件)

  例如, 要将主目录下的所有文件和目录复制到 /mnt 目录下, 执行的操作应该是:

```
$ cp -ri ~/*  /mnt
```

- 移动文件 mv (**mv**e)

  mv 命令有两种功能: 它可以将一个文件移动到另一个文件名上, 即文件改名；或将若干文件移动到一个目录里。与cp 命令类似, 为避免覆盖已有的文件, 可加上选项 “-i” 用于交互提示操作。

- 删除文件 rm (**rm**ove)

  删除文件的命令rm 和桌面环境中将文件移动到回收站的操作结果不同。回收站只是一个临时存放文件的目录, 移动到那里的文件并不释放硬盘存储空间, 仅当在回收站清除文件后, 该文件才永久消失；而rm 则是立即将文件删除。使用选项 “-i”, 会在每个文件删除之前给出一个提示。对于限制权限的文件, 删除时也会给出提示。

```
$ ls -l file?
-rw-rw-r-- 1 pi   pi  0 2月  20 08:44 file1
-r--r--r-- 1 pi   pi  0 2月  20 08:44 file2
$ rm file?
rm:是否删除有写保护的普通空文件 'file2'? y
```

如果想在操作过程中避免烦琐的确认, 不假思索地贯彻此命令, 可使用选项 “-f” 或 “--force”。

选项 “-d” 可以用于删除包括空目录在内的文件, 可代替`rmdir`；“-r” 选项执行目录的递归操作, 即逐层从下到上删除目录和文件。

命令行中, 使用`rm` 删除的文件不能用正常手段恢复。

- 浏览文件

  使用文本编辑器可以查看文件的内容, 但有两个问题：一是有可能无意中修改文件, 二是编辑器比较慢。如果没有编辑要求, 而仅仅是浏览的话, 可以有下面几个命令：

  1. `cat`：将文件内容一口气打印在终端上。如果文件很长, 前面的内容需要在终端上翻页才能看到。终端翻页的方法是组合键 SHIFT+PgUp/PgDn。但终端的缓存也是有限的 (一般被设为 1000 行或 2000 行), 太多的内容会被丢弃。
  2. `more`或 `less`：分屏显示。当一屏打印满后会暂停, 等待用户键盘干预继续显示。
  3. `head/tail` 及其组合。默认的方式下, `head`打印文件头 10 行, `tail`打印文件最后 10 行, 选项 “-n” 用于改变默认的行数。利用 Linux 的管道技术, 将二者结合使用, 可以打印一个文件中间的任意行内容：

     ```
     $ head -n 20 car.py | tail -n 5
     ```

     “|” 是管道操作符, 它表示将前面一条命令的标准输出 送给后面的命令处理。上面的命令执行效果就是打印文件 car.py 的第 16—20 行。
  4. `tac`: 它是命令`cat` 的逆序, 功能也是逆序：文件以行为单位逆序打印。它通常用于打印日志文件, 因为日志文件是按时间先后顺序生成的, 但我们更关心它最近记录的事件。

- 文件压缩/解压、打包/拆包

  我们常常需要将一组文件打包合并成一个文件, 便于携带和传输。为了减小体积, 还要进行压缩。`tar` (**t**ape **ar**chive) 是一个常用的工具。它可以通过不同的选项选择压缩方式, 打包和压缩一次完成。表2.3列出了一些常用的选项。

  传统方式,`tar` 选项前面没有 “-”。选项必须包含 A、c、d、r、t、u、x 中的一个。例如, 将目录 “桌面” 下的所有文件以 bzip2 格式压缩并打包成文件 desktop.tar.bz2:

  ```
  $ tar cjf  desktop.tar.bz2 桌面/
  ```

  拆包时, `tar` 能自动识别压缩文件的格式, 因此不需要用参数指定。错误的参数反而会导致命令执行失败。下面是解压命令的例子:

表 2.3 tar 的常用选项

选项	功能
-A	追加
-c	创建一个新的打包文件
-d	找出打包文件和文件系统的不同
-u	更新文件包中的指定文件
--delete	删除打包文件中的成员
-r	在打包文件后追加文件
-t	列打包文件的成员清单
-x	从打包文件中提取文件
-X	排除指定文件中列出的文件
-f	指定打包文件名
-j	使用 bzip2 压缩
-J	使用 xz 压缩
-z	使用 gz 压缩
-C	进入指定目录后再进行操作
-v	显示处理文件的过程

```
$ tar xvf desktop.tar.bz2 -C 下载
```

上面的命令表示在 “下载” 目录下解包文件 desktop.tar.bz2, 解包过程中会显示压缩包里的文件名。

单纯压缩、解压的命令可以用 `gzip`/`gunzip` 或者 `zip`/`unzip`。不同压缩/解压命令使用不同的算法, 它影响压缩率、压缩时间和解压时间。

2.3.5 进程控制

进程是指计算机正在运行的的程序, 是操作系统对任务管理的一个单位。对小程序来说, 一个程序运行就是一个进程。但一些复杂的程序运行时, 可能会有多个进程。

显示进程状态的命令是 `ps` (**p**rocess **s**tatus)。选项 “ax” 列出当前系统中所有进程的状态:

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     1:45 /sbin/init splash
    2 ?        S      0:00 [kthreadd]
    4 ?        S<     0:00 [kworker/0:0H]
  ...
20755 ?        Ssl    0:00 /usr/lib/evince/evinced
```

```
21326 ?        Ss     0:00 /lib/systemd/systemd --user
21327 ?        S      0:00 (sd-pam)
23376 pts/0    Sl     0:47 evince LinuxBasis.pdf
23885 pts/2    S      0:01 bash
25034 pts/0    S+     3:20 vim text/commandline.tex
  ...
```

列表的第一栏 PID (**P**rocess **ID**) 表示进程号。Linux 系统在创建进程时会为每个进程开一个进程槽, 并用进程号标记, 作为管理的入口。进程号是一个 16 位的正整数, 从小到大顺序安排。1 号进程是系统的初始化进程。进程号达到最大值 (32767) 后会绕回来再从小的空闲数字中挑选。第二栏表示进程启动的终端 (TTY 源自早期的电传打字机 **t**ele**ty**pewriter), 初始化程序和图形桌面启动的程序没有 TTY。STAT 表示进程当前的状态。一个进程可能处于四种状态之一: 运行、睡眠 (等待)、停止、僵尸 (死亡), 有些状态还可以进一步细分。

2.3.6 改变进程状态

用户可以使用 `kill`、`pkill` 或 `killall` 命令终止一个进程。`kill` 使用进程号作为操作对象, `pkill` 或 `killall` 使用进程名 (程序名) 作为操作对象:

```
$ kill 23376                # 以进程号为命令参数
$ pkill evince              # 以程序名为命令参数
```

`kill` 命令并不像它的名字那样残酷。它的本意是向进程发送信号, , 而信号是进程间通信的一种机制。但缺省的方式是发送 SIGTERM, 它一般导致进程终止。表2.4 是常用的信号值。信号值或者信号名都可以作为`kill` 的选项, 表示向指定进程发送指定信号。

有的进程会忽略 SIGTERM 信号, 这时如果想结束那个进程就必须用`kill -9 PID`。信号并非都是导致进程停止的, 例如下面的命令就可以让一个进程暂停、继续:

```
$ kill -STOP 23376
$ kill -CONT 23376
```

2.3.7 前台与后台

在终端启动图形界面程序时, 为了避免对当前终端的占用, 习惯上在命令行的尾部加一个“&” 符号, 该程序即以后台方式运行, 终端仍可以继续其他的命令操作:

```
$ idle3 &
```

shell 的两个内部命令 fg、bg用于命令的前、后台切换。上面这个后台命令可以用`fg` 将其拖回前台。

当一个终端运行了多个后台命令时, 另一个 shell 内部命令`jobs` 可以列出当前的后台命令和序号。序号可作为`fg` 的参数, 用于对指定命令的操作。

一个终端启动的进程接收到停止信号 (SIGSTOP) 时, 或在终端用快捷键 CTRL+Z 操作时, 该进程进入停止状态, 也会将终端让出。这时候在这个终端上用 `fg` 或`bg` 可以继续刚才的进程。

表 2.4 kill 命令常用信号描述

信号名	信号值	说明
SIGHUP	1	挂起
SIGINT	2	中断 (终端 CTRL+C)
SIGQUIT	3	进程退出
SIGABRT	6	异常中止
SIGKILL	9	杀死进程 (不可阻塞)
SIGUSR1	10	用户定义的信号 1
SIGSEGV	11	段错误 (存储器访问非法)
SIGUSR2	12	用户定义的信号 2
SIGPIPE	13	管道错误
SIGALRM	14	闹钟
SIGTERM	15	进程中止
SIGCHLD	17	子进程状态改变
SIGCONT	18	进程继续
SIGSTOP	19	进程停止 (不可阻塞)

2.3.8 I/O 重定向与管道

标准 I/O 设备的重定向是 Linux 文件操作的一种特性。在应用层面, 所有操作都将落实到文件读写上。文件是信息的一种载体, 重定向意味着改变信息的来源和去向。

表2.5是重定向操作符。

表 2.5 重定向操作符

操作符	含义
`> 文件`	输出重定向
`>> 文件`	追加式输出重定向
`< 文件`	输入重定向
`<< 分界符`	用分界符控制输入重定向的结束
`<<< string`	重定向输入字符串

下面的命令将文件 car.py 的第 16—20 行写到一个新的文件 car_mid.py 中：

```
$ head -n 20 car.py | tail -n 5 > car_mid.py
```

向一个文件中追加一段文字 “hello, world”, 可以像下面这样：

```
$ echo "hello, world" >> car_mid.py
```

用 “>” 进行输出重定向, 目标文件如果不存在, 则创建它, 如果存在, 则会将来的内容覆盖。用追加式输出重定向 “>>” 时, 定向的目标文件如果存在, 会把新的内容追加到该文件之后, 不会覆盖原来的内容。

当一个程序需要由键盘输入信息时, 可以事先将这些信息写到一个文件中, 再使用输入重定向方法, 可以避免程序运行时的人工干预。例如：

```
$ head < car.py
```

2.4 嵌入式系统

嵌入式系统是嵌入在产品中的计算机系统。它们没有刻意制造成通用计算机的形态, 而是以产品本身的形态展现在世人面前, 但它们同时又具有计算机的基本组成结构。通俗地说, 它们是面向应用、面向产品的、具有特定用途的计算机。嵌入式系统是伴随着微型计算机的发展而发展的。

2.4.1 硬件和软件

硬件和软件是计算机的两大基本组成。嵌入式系统的硬件包括处理器、存储器、总线和外设等等。与通用处理器不同的是, 嵌入式处理器内部除了 CPU, 还包括应用系统所必须的 I/O 接口以及存储器, 以便在构成系统时能最大限度地简化系统结构、减小体积、降低成本。嵌入式系统的软件同样也包括系统软件和应用软件, 系统软件的核心是操作系统内核。

嵌入式系统由于其资源配置及应用场景的多样化, 操作系统的形态也多种多样, 如 VxWorks、eCos、RTEMS、Palm OS, 也包括经裁减的 Linux 或者其他桌面操作系统。在一些简单的嵌入式应用中, 所谓的操作系统就是指其上唯一运行的应用程序。

多数嵌入式应用对软件的实效性有严格的要求, 因此通常会采用实时操作系统。不同语境下, “实时” 的意义有所不同, 通俗的解释就是 “快”。在操作系统理论中, “实时性” 是指对特定任务的处理时间的上限是可预知的。超过这个上限, 该任务即宣告失败。实时性在工业控制、医疗设备、以及军事领域都是不可或缺的特性。

2.4.2 嵌入式处理器

由于产品的多样性, 应用于不同产品中的嵌入式处理器从形态和性能上都有很大差异。概括来说, 嵌入式处理器有以下几种类型：

- 单片机 (又叫微控制器, micro controller) 类。这类处理器处于嵌入式系统的低端应用, 它们体积小、功耗低、价格低, 不擅长算法实现, 主要用于工业及民用控制系统。典型的如 Intel 的 MCS51 系列、TI 的 MSP430 系列等等, 普通个人计算机键盘里的 MCS8048 也属此类。主频一般在几 MHz 到几十 MHz。
- 数字信号处理器 (DSP)。数字信号处理理论在语音、图像、视频等方面都有广泛的应用。根据信号处理的需求特点,DSP 对系统结构和指令进行了特殊设计, 使其在卷积、数字滤波、频

谱分析等典型数字信号处理算法上具有更高的效率。这类处理器以算法见长, 典型的 DSP 包括 TI C5000 系列、C6000 系列、ADI 的 SHARC 系列和 Blackfin 系列等, 性能从数百 MIPS 到上万 MIPS/FLOPS 不等。

- 微处理器类,
- 片上系统 (SOC, **S**ystem **o**n **C**hip)。

2.5 网络连接

2.5.1 远程终端

作为嵌入式应用, 树莓派一般不配置键盘、显示器这样的人–机接口。如果需要, 树梅派连接人–机接口很简单：Linux 操作系统直接支持 USB 键盘/鼠标, HDMI 输出可以直连带有 HDMI 接口的显示器, 或通过 HDMI-VGA 转接器连接 VGA 显示器, 这样就构成了一个完整的通用计算机系统。但通常没必要这样做。开发人员如果需要在树莓派上开发软件, 或者使用树莓派嵌入式应用, 最简单的方法是通过网络连接。

多数 Linux 发行版都会启动一个用于远程连接的 SSH (**S**ecure **SH**ell) 的服务, 它用于提供通过网络远程登录使用计算机的功能。如果树莓派启动了 sshd (字母 “d” 守护进程 **d**aemon process 的首字母) 并连接到局域网, 我们就可以在个人计算机终端上使用远程登录功能：

```
$ ssh pi@192.168.2.103
The authenticity of host '192.168.2.103 (192.168.2.103)' can't be established.
ECDSA key fingerprint is SHA256:rrFsuRrEhzNaGea15HNvGRCcYDJmaCbAwMYmja+fyD0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.2.103' (ECDSA) to the list of known hosts.
pi@192.168.2.103's password:
```

首次登录时, 系统会给出安全提示。如果用户确认无误, 应回答 “yes”。一旦建立安全连接后, 以后只需要在每次登录或连接时输入用户密码就可以使用了。登录成功后, 会看到树莓派给出的问候界面：

```
##############################################
#           Raspberry Pi3 (aarch64)          #
#                  Desktop                   #
#                  WELCOME!                  #
##############################################
Raspberrypi:~$
```

这里假设树莓派的 IP 地址是 192.168.2.103, 用户名是 pi。默认的方式下, 出于安全的考虑, 多数 SSH 服务不允许 root 用户远程登录。登录树莓派后, 在这个终端上所有的操作都是在树莓派上进行的, 除了没有图形界面以外, 它与本地的操作完全相同。如果目标系统启动了 X 服务, ssh 也可以通过选项 “-X” 将远程服务器的窗口延伸到本地。

SSH 服务除了提供远程登录以外, 同时还提供远程文件传输功能。用于文件传输的本地命令是 scp, 用法是：

```
$ scp file1 user@192.168.200.170:file2
```

它表示把本地文件 file1 复制到 192.168.200.170 计算机用户 user 的主目录, 作为 file2 保存。复制过程中会要求输入远程用户 user 的密码。如果复制的目标文件不想重新命名, 可以省去 file2 部分, 但 IP 地址后面的冒号需要保留。反方向的复制同理：

```
$ scp user@192.168.200.170:file1 .
```

“.” 表示当前目录。复制结果是将远程文件 file1 复制到本地当前目录, 不重新命名。

使用密码登录、复制文件较为繁琐。SSH 还支持密钥访问方式。下面是建立密钥的过程：

1. 在主机上使用 ssh-keygen 创建密钥对:

   ```
   $ ssh-keygen -t rsa -b 4096
   ```

   选项 “-t” 用于指定加密算法, “-b” 用于指定密钥长度。创建密钥过程中会提示输入密钥文件和保护密码。缺省的密钥文件是 id_rsa (私钥) 和 id_rsa.pub (公钥)。生成的密钥文件在 ~/.ssh 目录下。当使用保护密码时, 首次使用密钥会要求输入保护密码加以认证。公钥是公开的, 但私钥必须妥善保管。
2. 将公钥复制到远程计算机：

   ```
   $ scp ~/.ssh/id_rsa.pub pi@192.168.2.103:
   ```

   此处假设远程计算机的地址是 192.168.2.103。
3. 将公钥添加到远程计算机的 SSH 认证文件里。下面是在远程机上的操作：

   ```
   pi@remote:~$ cat id_rsa.pub >> ~/.ssh/authorized_keys
   ```

   上面使用了追加式输出重定向方法。

完成以上工作后, 远程登录或在本机和远程机之间复制文件时不再需要输入密码。

2.5.2 远程桌面

Linux 系统的网络功能非常强大, 除了上面介绍的 ssh 终端连接方式以外 (ssh 也可以实现图形化方式), 一个较为简单的方法是 VNC (**V**irtual **N**etwork **C**omputing)。服务器端 (树莓派) 启动 X-Server 和 VNC 服务后, 其他机器就可以通过 VNC 客户端连接服务器的图形界面了。Linux 系统常用的客户端命令是 vncviewer：

```
$ vncviewer 192.168.2.103
```

如果一切正常, 服务器的桌面就展现在主机的一个窗口中。

图2.1 是通过 vnc 登录到树莓派上展示的图形界面, 树莓派运行一个轻量级 桌面环境XFCE4。

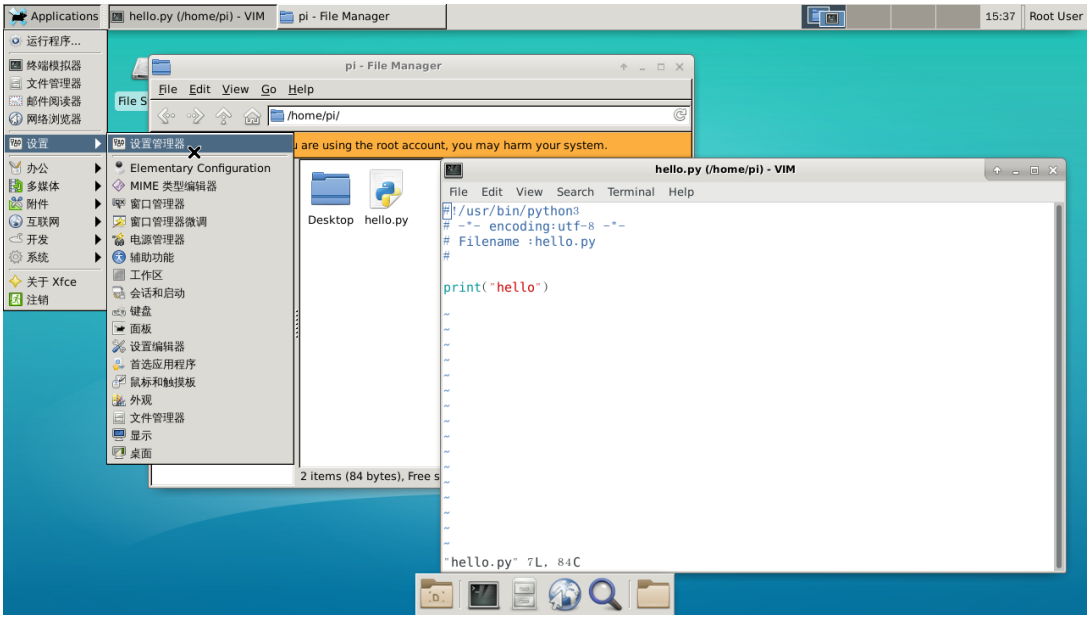


图 2.1 树莓派上的 XFCE4 桌面

2.5.3 远程网络存储

嵌入式系统通常不可能有大容量的存储空间。此时网络文件系统 NFS (**N**etwork **F**ile**S**ystem) 就可以发挥作用了。使用网络文件系统时, 要求 PC 端作为 NFS 的服务器, 启动 NFS 服务。PC 机上的文件 /etc/exports 限定了提供网络文件服务的目录和权限:

```
/srv/nfs4     192.168.*.*(rw,sync,subtree_check,no_root_squash)
```

假设 PC 的 IP 地址是 192.168.200.170。在树莓派上执行下面的命令:

```
# mount 192.168.200.170:/srv/nfs4 /mnt -o nolock,proto=tcp
```

便可将 PC 的 /srv/nfs4 目录挂载到 /mnt 目录下。在 PC 上访问 /srv/nfs4 和在树莓派上访问 /mnt, 二者访问的是同一个资源, 真正的存储设备在 PC 上。`mount` 命令的选项 "-o" 指定的参数表示强制使用 TCP 协议 (NFS 默认优先使用 UDP 协议,UDP 协议方式在访问大文件时有问题)。

网络连接在嵌入式开发中非常有用, 它除了可以省下一套输入输出设备 (键盘、鼠标、显示器) 以外, 还可以借助主机的强大的处理能力、灵活的软件配置和大量的存储空间, 为嵌入式系统开发提供强有力的支持。

2.6 本章小结

本章介绍了操作系统的一些基本概念, Linux 操作系统的特点和基本使用方法。下面是一些 Linux 命令行环境中最常用的命令, 建议熟练掌握。

命令	功能	常用选项
ls	列文件、目录清单	-l
more	分屏打印文件内容	
cp	复制文件	-r,-a
cd	改变工作目录	
mv	文件移动/改名	-i
mkdir	创建新目录	-p
rmdir	删除空目录	
rm	删除文件	-i, -f
ps	显示系统进程表	ax
tar	打包解压/压缩文件	-z,-x,-f,-j,-J

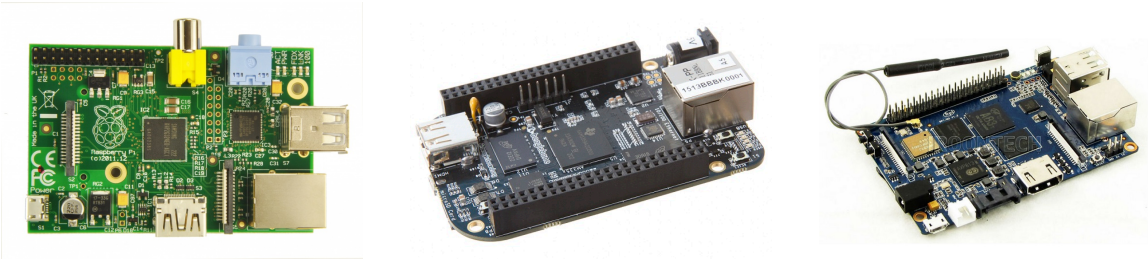
3

树莓派

3.1 嵌入式系统

树莓派是英国一个非盈利机构树莓派基金会开发的一种卡片式计算机, 最初的目的是用于对少年儿童进行计算机普及教育。第一代树莓派产品发布于 2012 年 2 月。目前最新的版本树莓派 3-B+ 于 2018 年 3 月 14 日发布, 这一天又被叫做 π 日。树莓派采用博通公司的处理器芯片 BCM283X 作为核心处理器, 内核是 Cortex-A53 架构, 是一个 4 核的 64 位 ARM处理器, 主频可以达到 1.4GHz, 此外还带有 VideoCore-IV 的图像处理单元 GPU (**G**raphical **P**rocessing **U**nit)。板上支持无线网络和蓝牙, 并可以通过 HDMI (**H**igh **D**efinition **M**ultimedia **I**nterface) 接口输出高清视频。

由于树莓派的成功, 其他一些计算机开发商也仿照树莓派开发了类似的卡片式计算机产品。



(a) RPi-B(2012.2) **(b)** BB-Black(2013.4) **(c)** BananaPi-M3(2015.12)

图 3.1 各种派

树莓派这类计算机结构简单、体积小、耗电低, 却拥有与普通计算机几乎相同的功能和性能, 可以很方便地植入各种应用系统中。这种具有计算机的基本结构但又不具备普通计算机形态的计算机, 业界称为 “嵌入式系统”。

关于嵌入式系统, 目前尚没有严格的定义。一个被普遍接受的概念是：以应用为中心、以计算机技术为基础、软件硬件可裁剪的、适应系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。

所谓的 “嵌入式” 是指它是嵌在产品中的, 是面向产品的专用计算机。人们看到的只是产品本身, 看不到计算机。目前 98% 的计算机产品都属于嵌入式计算机。嵌入式系统广泛应用于工业控制、机电、航天、通信、环境监测、汽车电子、家用电器等各种使用微处理器系统的环境。

3.1.1 核心处理器

计算机系统的硬件核心是 CPU (**C**entral **P**rocessing **U**nit, 中央处理器)。对于通用计算机来说, CPU 是一个独立的芯片, 大家所熟知的个人计算机中的 Intel 或 AMD 的处理器皆如此。CPU 还需要配合其他外设 (如中断控制器、I/O 接口、总线、存储器等等) 才能组成一个计算机系统。但对于嵌入式处理器来说就是另一回事了。嵌入式处理器中, 除了 CPU 以外, 本身就已经包含了系统所需要的各种片内接口和存储器, 其中最具技术含量的就是 CPU。嵌入式处理器的 CPU 又被称作内核 (core)。目前比较著名的嵌入式内核包括 ARM、MIPS、PowerPC 等等。

处理器的形态来自产品的要求。嵌入式产品生产商向 IP core 供应方购买内核, 加上产品需要的接口, 形成嵌入式处理器芯片。通用计算机的 CPU 需要配合其他设备才能组织起一个计算机, 而嵌入式处理器独自就可以构成一个计算机系统。由于目前的技术水平所限, 一些人–机接口设备暂时不能集成。为了体现个性化的功能通常也不集成。

3.1.2 嵌入式处理器的类型

目前, 嵌入式处理器主要有以下几种类型:

Micro Controller 中文译为 “单片机”, 顾名思义, 它是由单个芯片构成的计算机系统。这类处理器的特点是功能单一、低成本、低功耗, 广泛应用于工业和家电的低端控制系统。一个典型的应用是计算机键盘控制器中使用的 MCS-48。

DSP 数字信号处理器 (**D**igital **S**ignal **P**rocessor)。计算机能处理的信息都是数字化的, 数字信号处理理论为这些信息处理建立了理论基础。但如果要让这些信息处理具有实用价值, 计算机必须足够快。上世纪 80 年代, 一些处理器公司专门针对数字信号处理算法的特点设计了这类处理器。这类处理器针对算法密集型应用, 构成系统时相比其他类型的处理器具有较大的成本优势。它们对语音、图像和视频应用系统产生了巨大的影响。

随着处理器性能的提高、并行算法的兴起, 近年来数字信号处理器的发展有停滞的趋势, 相关设计企业的研发重点也在逐渐转移, 但仍有一些产品在使用专用的数字信号处理器。

MPU **M**icro **P**rocessing **U**nit。集成了嵌入式内核和必要的片内接口的、面向产品的专用处理器。这类处理器为数众多, 覆盖从低端到高端的各种嵌入式系统应用。

SoC 在通用可编程逻辑器件实现产品应用的片上系统 (**S**ystem **o**n **C**hip)。对于产品来说, 这是最具竞争力的嵌入式应用。但由于受到 IP core 的核心设计能力的限制, 此类嵌入式产品尚未成为主流。

3.2 树莓派简介

3.2.1 树莓派的性能

计算机系统的性能主要取决于 CPU。评估处理器性能的一项重要指标是 MIPS (**M**illions of **I**nstructions **p**er **S**econd, 每秒执行指令的百万条数)。以树莓派 3B 为例, 处理器 BCM2837 是一个 4 核的 Cortex-A53 架构处理器, 它支持 ARM 32 位和 64 位指令集[1], 主频 1.2GHz。相同架构的处理器, 处理器性能与核心数、主频直接相关。根据上面的数据, 可以粗略估计该处理器性能达到 4800MIPS。不同架构之间, 同主频的处理器性能会有一定差距。

一些处理器还具有硬件浮点处理能力。浮点数 (floating-point) 是计算机表示小数的一种格式。由于其格式的特点, 定点 (fixed point) 处理器在处理这类数值运算问题时效率极低。举例来说, "1+1" 这个计算在几乎所有处理器中都只需要一条指令, 而 "1.0+1.0" 在定点处理器中可能需要数十条甚至上百条指令。同档次的浮点处理器价格一般会高一些, 功耗也会大一些。即使在浮点处理器中, 浮点处理的效率也不会高过定点处理能力, 因此, 只要能解决计算精度问题, 多数工程应用中都会尽可能采用定点算法实现。

评估浮点处理能力的指标是 MFLOPS (**M**illions of **Fl**oating-point **O**perations **p**er **S**econd), 它的数值通常不会超过同款处理器的 MIPS 值。[2]

决定计算机系统性能的另一个重要因素是存储器的性能和大小。BCM2837 片内带有 16KB 一级高速缓存和 512KB 二级高速缓存树莓派 3B 板载 1G DDR 同步动态随机存储器 DRAM (**D**ynamic **R**andom **A**ccess **M**emory), 时钟频率 500MHz/600MHz。由于 DRAM 比 CPU 工作节奏慢很多, 利用 cache 结构, BCM2837 片内高速缓存可以使整个系统性能接近处理器的最高性能。

3.2.2 图像处理单元

现代计算机系统应用中常常要处理图像和视频信号, 图像处理的特征是它具有巨大的并行性。一幅图像由二维的像素点的矩阵组成, 而且数据量庞大。全高清视频每幅图像 1920×1080 像素, 每秒 30 幅以上, 每个像素又分解成 RGB (**R**ed-**G**reen-**B**lue, 红、绿、蓝) 三种颜色分量, 每种颜色 8bits。但 RGB 很难进行数值运算 (不好说红色比绿色的强度大、或者绿色比黄色更鲜艳), 因此在图像处理中, 会把 RGB 转换到另一个具有数值意义的色彩空间, 比如用亮度、色度、饱和度等参数表示。一个典型做法是转换到 YUV 空间 (Y 是亮度分量, U、V 是颜色分量)。二者的转换公式是:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

[1] Cortex 是处理器硬件设计架构名称, ARM 是指令集名称。ARM 公司设计的处理器统称 ARM。每一种处理器还有一个架构名称; 不同处理器支持的指令集版本也有不同。

[2] 在超级计算机中, MIPS 和 MFLOPS 单位过小, 通常会将 M 换成 G 甚至 P, 它表示 10^9 或 10^{15}。

由于人的视觉感知对色度敏感度低, 在 YUV 空间中还有利于数据压缩。上面的计算在串行计算结构中需要 9 次乘法和 9 次加法, 共需要 18 条指令。注意到, Y、U、V 之间的计算是不相关的, GPU可以将这类的算法进行并行处理, 从而成倍地提高处理速度。

现代计算机系统越来越看重 GPU 的作用, 它在图像处理、视频编解码、计算机视觉等方面有着及其重要的地位。BCM2837 内部集成了一个 VideoCore-IV 图像处理单元, 主频 400MHz, 运算性能 28.8GFLOPS。这个数值已经远远超过 ARM 核中的硬浮点处理单元的运算指标, 它可以实现 1080p@60 帧的高清视频解码。

3.2.3 外设及接口

树莓派 3B/3B+ 带有一个以太网接口、4 个 USB host、1 个无线网接口和蓝牙接口。片内大量 I/O 接口通过一组 2×20 引脚引出, 作为扩展设备控制接口。引脚功能见表3.1。

3.3 操作系统安装

根据树莓派官网 https://www.raspberrypi.org提供的信息, 树莓派支持 FreeBSD、OpenBSD (BSD Unix 的变种)、Plan 9 (源于 Bell 实验室的 Unix 分布式操作系统)、Windows10 Iot Core、RISC OS, 以及多种 Linux 发行版。大多数树莓派应用都基于 Linux 操作系统。

树莓派没有板载的非易失性存储设备, 需要使用 microSD (**S**ecure **D**igital, 又称 TF 卡, **T**rans**F**lash) 存储卡作为其系统运行和存储设备。它通常被划分成两个或两个以上的分区, 其中第一个分区必须格式化成 FAT (**F**ile **A**llocation **T**able, 文件定位表) 文件系统, 作为 Boot 分区。该分区安装了系统引导程序 Bootloader (又称固件, firmware。树莓派的这款 firmware 不是开源软件) 和内核镜像文件, 此外还有可能安装一个作为 Linux 系统初始化的 RAM Disk 镜像。Linux 系统正常启动后, 会先加载这个镜像文件, 作为它的根文件系统, 进行一些必要的初始化工作, 然后切换到另一个由 Bootloader 参数指定的分区作为真正的根文件系统, 这个分区必须是符合 Linux 根文件系统要求的格式 (Ext2/3/4FS 文件系统、ReiserFS 文件系统、YAFFS 文件系统等等)。

树莓派的 Bootloader 可在 https://github.com/raspberrypi/firmware 下载。将下载的 boot 目录下的文件直接复制到 TF 卡的第一个分区即可。针对树莓派移植的内核源码在 https://github.com/raspberrypi/linux, 可根据需要自行剪裁和编译, 将编译后的内核镜像文件 zImage 替换 TF 卡 Boot 分区里的 kernel7.img 或 kernel8.img, 就成为更新后的系统。kernel7.img 是 32 位 ARM 指令集 armv7-a 内核镜像, 支持树莓派 2 代和 3 代, kernel8.img 是 64 位 ARM 指令集 armv8-a 内核镜像, 仅支持树莓派 3 代以上。

树莓派官网提供了若干 Linux 发行版。一个完整的 Linux 发行版, 使用方法与个人计算机上的桌面系统完全一样。

表 3.1 I/O 引脚功能

P1: The Main GPIO connector							
WiringPi	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi
		3.3v	1	2	5v		
8	GPIO2	SDA1	3	4	5v		
9	GPIO3	SCL1	5	6	GND		
7	GPIO4	GCLK	7	8	TxD	GPIO14	15
		GND	9	10	RxD	GPIO15	16
0	GPIO17	GEN0	11	12	GEN1	GPIO18	1
2	GPIO27	GEN2	13	14	GND		
3	GPIO22	GEN3	15	16	GEN4	GPIO23	4
		3.3v	17	18	GEN5	GPIO24	5
12	GPIO10	MOSI	19	20	GND		
13	GPIO9	MISO	21	22	GEN6	GPIO25	6
14	GPIO11	SCLK	23	24	CE0	GPIO8	10
		GND	25	26	CE1	GPIO7	11
	GPIO0	ID-SD	27	28	ID-SC	GPIO1	
	GPIO5		29	30	GND		
	GPIO6		31	32		GPIO12	
	GPIO13		33	34	GND		
	GPIO19		35	36		GPIO16	
	GPIO26		37	38		GPIO20	
		GND	39	40		GPIO21	
WiringPi	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi

注: GPIO18、GPIO19 可用于硬件 PWM.

3.4 树莓派外设

各种设备的驱动程序属于 Linux 内核的一部分。设备驱动被加载后, 会在 /dev 目录下创建一个设备文件, 用户程序通过读写这个设备文件实现对设备的操作。在树莓派内核中, 提供了一些较常用设备的用户空间操作方法。

3.4.1 输入和输出

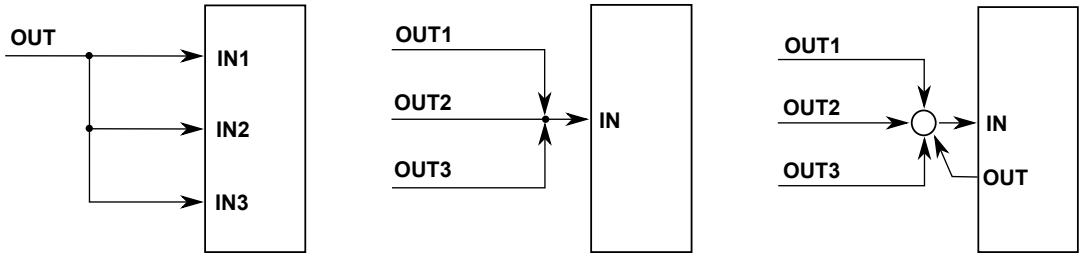
计算机系统的工作, 最终结果都将落实到输入和输出这两个操作上。在计算机系统内部, 它们被称作读和写：从存储器或 I/O 接口读数据或信息、向存储器或 I/O 接口写数据或信息。当这些信息反映到外部引脚时, 就是输入信号和输出信号。

在一个数字测控系统中, 输入引脚接受外部设备产生的电信号, 它们通常是量化的, 要么是高电平, 要么是低电平。高低电平的界限取决于系统的硬件设计。例如在 TTL (**T**ransistor-**T**ransistor **L**ogic, 晶体管逻辑) 电路中, 超过 2.0V 算高电平, 低于 0.8V 算低电平; CMOS (**C**omplementary **M**etal-**O**xide-**S**emiconductor, 互补型金属氧化物半导体) 电路中, 1/3Vdd 以下算低电平, 2/3Vdd 以上算高电平。信号的临界状态会导致不确定的结果。

输出引脚在计算机系统的控制下产生对外的输出信号, 同样, 输出信号也有高、低两种电平。在输出引脚无负载的情况下, 高电平接近电路的供电电压, 低电平接近 0, 除非电路故障, 否则不可能出现临界电平。

以上 "输入"、"输出" 的概念是站在计算机系统的角度讨论的。一个系统的输出信号会接到另一个系统的输入端, 一个系统的输入接收的也是另一个系统的输出。只有明确了定位, 讨论输入或输出才有意义。

在电路中, 一个输出信号可以连向多个输入端, 同时控制多个输入设备 (想像一下一根电线同时供应多个用电设备), 但多个输出信号通常不允许同时接到一个输入端上 (想像一下用多个电源插头接到一盏灯的效果), 除非电路有特殊的设计。图3.2a是正常使用情况; 图3.2c中通过一个电路将三个输出信号重新组合, 对于这个电路来说相当于一对一的关系, 不违背原则, 比如用一个可控的多路开关, 实现多选一的控制; 而图3.2b是错误的。



(a) 单输出多输入 (正确) **(b)** 多输出单输入 (错误) **(c)** 多出单入 (可能正确)

图 3.2 输入/输出接口的几种情况

3.4.2 LED 控制

GPIO

嵌入式处理器集成了各种设备接口, GPIO (**G**eneral **P**urpose **I**nput/**O**utput, 通用 I/O 接口) 是较常用的一类, 它通过一只引脚实现简单的输入/输出功能。用户空间可以通过下面简单的方法使用 GPIO。

首先, 在 /sys/class/gpio 目录中, 可以看到有这样几个文件和目录:

```
/sys/class/gpio       (目录)
  |-- export          (用于创建GPIO结点)
  |-- unexport        (用于删除GPIO结点)
  |-- gpiochip0       (符号链接)
  |-- gpiochip100     (符号链接)
  `-- gpiochip128     (符号链接)
```

用户空间操作 GPIO 时, 重点关心 export 和 unexport 这两个文件。

文件 export 用于创建一个 GPIO 的目录 (结点), 该目录提供了对应引脚输入/输出功能的进一步操作方法。为达到这个目的, 首先确定我们想使用的 GPIO 序号。这里的序号指的是芯片内部的 GPIO 编号, 不是电路板插座上的编号。例如我们想使用 GPIO18, 它对应树莓派 3B 的 2×20 脚排针的第 12 脚位置。创建 GPIO18 的功能可以通过向文件 export 写入一个数字 "18" 实现：

```
# echo 18 > export
```

命令执行后, 会在该目录下看到多出一个链接文件 gpio18, 它实际上是一个目录的链接。进入这个目录, 可以看到有 value 和 direction 等文件。direction 用于设置 GPIO 的输入或输出方式, 它接受两个值：'in 和 out; value 用于获取或控制对应引脚的电平, 是一个可读文件, 如果里面的值是 0, 表示当前引脚电平为低, 如果是 1, 表示电平为高。当 direction 内容为 out 时 value 是可写的, 写入值将影响到引脚的电平输出。

对于可读文件, 可以使用 cat或 more命令查看里面的内容：

```
# cat direction
in
# cat value
1
```

LED

发光二极管 (**L**ight **E**mitting **D**iode) 是一种特殊的二极管, 通过掺入不同的材料, 在电子与空穴的复合中被激发而发光, 此谓电致发光效应。LED 所发射出的光的波长 (颜色) 由半导体物料的禁带能量决定。普通的硅和锗在能级跃迁时没有释出光子, 而是把能量转化为热能。LED 的材料被激发时, 能量会以光子形式释放, 这些禁带能量对应着近红外线、可见光、或近紫外线波段的光能量。

可见光 LED 广泛应用于信息显示、大屏幕显示器及照明设备, 红外 LED 应用在红外遥控和通信中。

LED 工作时需要加上正向偏置电压。二极管的电流与电压成近似指数关系, 通常在使用时需要串联一个电阻以限制电流, 以免造成永久损坏。LED 的反向击穿电压值一般比较低。除了特殊功能的二极管以外, 反向击穿也会造成二极管的损坏。

将两个不同颜色的 LED 反向并联, 封装在一个透明容器里, 就构成了双色二极管, 不同方向的偏置电压产生不同的颜色。一些发光二极管内集成了红绿蓝三色二极管, 改变每个颜色的亮度可以产生彩色的效果。

因发明蓝色发光二极管, 天野浩、赤崎勇、中村修二三人共同获得 2014 年度的诺贝尔物理学奖。评委会对此的评价是 "发明了可带来明亮节能的白色光源的高效率蓝色发光二极管"。

Python 文件操作

文件操作的一般顺序是：1.`open()` 打开文件, 2. `read()`、`write()` 读写文件, 3. `close()` 关闭文件。Python 内建了文件对象, 用于文件读写。以下是一些常用的方法函数。

- `open`(file, [mode]), 以某种方式打开文件, 返回文件对象。方式 mode 可以是 “r”(**r**eading)、“w”(**w**riting) 或 “a” (**a**ppending), 缺省方式是读文件。使用 “w” 或 “a” 打开文件时, 如果文件不存在则会新建, 如果文件存在, “w” 会将其截断。`mode` 中添加 “b” 表示以二进制方式操作, 添加 “+” 表示读写可同时进行。
- `read`([size]), 从文件中读取 `size` 个字节, 返回读取的字符串。当不指定 `size` 时, 读到文件结束符为止。在非阻塞方式中, 读到的数量可能会比请求的数量少。
- `write`(str), 将字符串对象 `str` 写入文件, 返回成功写入的字节数。由于缓冲的存在, 可能需要使用 `flush`() 或 `close`() 才能真正让写操作生效。
- `readline`([size]), 读入一行, `size` 用于指定读取的最大字节数, 当它小于一行的字节数时, 只读取一行的一部分。
- `readlines`([size]), 读入多行, 返回一个列表。
- `writelines`(sequence_of_strings), 将字符串序列写入文件。
- `seek`(offset, [whence]), 定位文件指针。`offset` 指定偏移量, `whence` 指定参考位置, 0 (缺省方式) 表示文件头, 1 表示当前位置, 2 表示文件尾。
- `tell`(), 返回当前位置, 以文件头为参考点。
- `flush`(), 清空缓冲区。
- `close`(), 关闭文件。

控制程序

有了上面的准备, 我们就可以实现一些简单的控制功能了。下面是一个实现交替输出高低电平的 Python 程序：

清单 3.1 led.py

```
1  #!/usr/bin/python3
2  # -*- encoding: utf-8 -*-
3
4  import time
5
6  f = open('/sys/class/gpio/export')
7  f.write('18')
8  f.close()
9  f = open('/sys/class/gpio/gpio18/direction')
10 f.write('out')
```

```
11 f.close()
12
13 f = open('/sys/class/gpio/gpio18/value')
14 while True:
15     f.write('0\n')
16     time.sleep(0.5)
17     f.write('1\n')
18     time.sleep(0.5)
```

在 GPIO18 对应的排针第 12 引脚接上 LED 发光二极管, 运行上面的程序, 可以看到发光管以每秒一次的节奏闪烁。电路连接见图3.3。

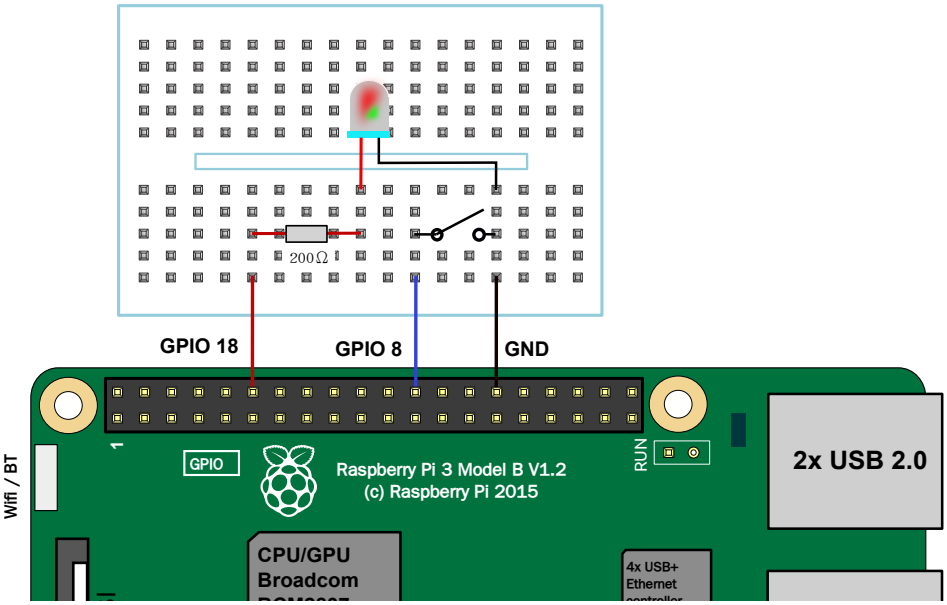


图 3.3 树莓派控制一个 LED 和一个按键开关

GPIO 不再使用时, 应向文件 unexport 写入对应的序号将结点删除。已经存在的结点不能重复创建, 否则会产生文件操作错误。

嵌入式处理器通常都有大量的 GPIO 引脚, 有些引脚是功能复用的, 使用前需要先明确该引脚未用于其他功能, 否则会造成系统故障。

PWM

PWM (Pulse Width Modulation, 脉冲宽度调制), 或称脉冲时间调制 (Pulse Duration Modulation, PDM), 是一种使用脉冲信号进行信息编码的技术[3]。它可以产生可变占空比的矩形波, 通过数字方法实现模拟控制信号。

在上面的 LED 控制程序中, 改变两个 `sleep()` 函数的时间参数, 并将一个周期的总时间缩短到 0.1 秒以下, 此时肉眼无法分辨 LED 的闪烁, 高、低电平所占时间比例不同, 看到的就是

[3]引自 wikipedia。

LED 不同的亮度。这实际上就是软件实现的 PWM 形式。

软件实现的 PWM, 频率不能太高, 一方面是因为处理器负担加重, 另一方面软件实现的精度也会有较大的误差。

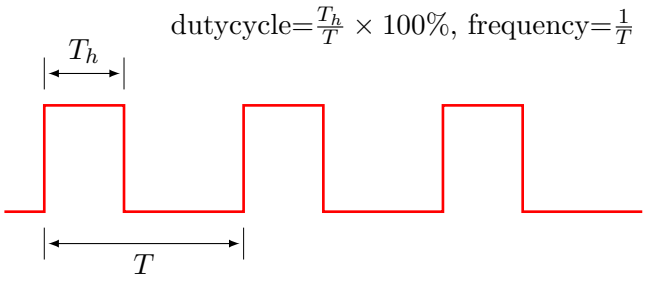


图 3.4 PWM 波形参数

要启用硬件 PWM, 需要将设备树 pwm-2chan 加入系统启动设置文件 config.txt 中, 该文件位于引导分区：

```
dtoverlay=pwm-2chan,pin=18,func=2,pin2=19,func2=2
```

并在内核中使能 pwm-bcm2835 (可以是模块加载, 也可以直接由内核支持)。上面的参数不是随便设的。在 2×20 的引出脚中, 只有 GPIO12、GPIO13、GPIO18 和 GPIO19 具有 PWM 功能, 并且每次最多只能同时有 2 个硬件 PWM 功能引脚。它们所对应的 func 见表3.2。

表 3.2 PWM 功能引脚设置参数

	PWM0	PWM1
GPIO12	4	
GPIO13		4
GPIO18	2	
GPIO19		2

设置了 PWM 功能后, 同时也会将 PWM0 和 PWM1 连向耳机插座。因此如果 PWM 输出的是音频信号时, 插上耳机就可以听到声音了。

硬件 PWM 使用硬件定时器产生高频率、准确周期的脉冲信号, 精确控制引脚输出高低电平的时间。树莓派有两个硬件 PWM 模块, 可以通过配置引导参数激活其功能。当硬件 PWM 功能可用时, 可以在 /sys/class/pwm 目录下看到 pwmchip0 的链接 (目录)。这个目录里有下面这些文件：

```
/sys/class/pwm/pwmchip0      (目录)
  |-- export                 (用于创建 PWM 结点)
  |-- unexport               (用于删除 PWM 结点)
  |-- npwm                   (PWM 数目)
  `-- uevent
```

npwm 显示的是 2, 因此可以通过 “echo 0 > export” 和 “echo 1 > export” 创建两个 PWM 结点, 它们分别对应 GPIO18 和 GPIO19 (也可能对应 GPIO12 和 GPIO13, 取决于系统启动时向内核传递的参数设置)。此时引脚的 GPIO 功能便切换到 PWM 功能。

在 pwm0 或 pwm1 目录下有这几个重要的文件：

```
/sys/class/pwm/pwm0       (目录)
  |-- enable              (使能/禁止 PWM 输出)
  |-- capture             (PWM 捕获)
  |-- period              (周期设置, 单位 ns)
  |-- duty_cycle          (占空周期设置, 单位 ns)
  `-- polarity            (极性设置, normal/inversed)
```

占空周期和周期的参数以 ns 为单位, duty_cycle/period 即为 占空比。设置完这两个参数后, 向 enable 写入 1 即启动 PWM 输出。polarity 用于控制输出的极性, 接受两个控制值：normal 和 inversed。正常情况下 (normal) duty_cycle 是高电平的持续周期, 当写入 inversed 后输出波形颠倒。

下面的操作将产生 500Hz、50% 占空比的矩形波：

```
# echo "2000000" > period
# echo "1000000" > duty_cycle
# echo 1 > enable
```

PWM 模块创建后, 下面的程序以硬件 PWM 方式改变 LED 的亮度：

清单 3.2 dim.py

```
1 #!/usr/bin/python3
2 # -*- encoding: utf-8 -*-
3
4 import time
5
6 period = open('/sys/class/pwm/pwm0/period')
7 dc = open('/sys/class/pwm/pwm0/duty_cycle')
```

PWM 模块不再使用时, 应向 unexport 写入 PWM 通道号 (0 或 1), 以删除 pwmX 结点。

3.5 本章小结

树莓派是一款典型的卡片式计算机, 配合丰富的引出接口, 可以灵活应用在许多嵌入式场合。本章介绍了树莓派的硬件性能、引出引脚的功能, 和在文件系统层上直接操作 GPIO 的简单方法。

4

Python 基础知识

4.1 Python 简介

Python 最早出现于 1980 年代晚期[1], 它的作者是荷兰程序员 Guido van Rossum, 当时就职于荷兰国家数学与计算机科学研究院 (该机构的荷兰语缩写是 CWI)。谈及创作动机时, van Rossum 说是为了填补 1989 年圣诞节放假期间的空虚, 手边正好有一台计算机, 于是决定为一个新的脚本语言 (ABC 语言的继任者。ABC 语言是一种用于教学的程序设计语言) 写一个解释器。当时有一个英国的喜剧小组, 名叫 Monty Python。由于 van Rossum 本人非常喜欢 Monty Python 的节目, 于是他决定用 “Python” 命名他的这项工作。

Python 在英语里有 “蟒蛇” 的意思, Python 的图标就是两条纠缠在一起的蟒蛇。

4.1.1 Python 发展史

Python 属于自由软件。它的版权协议几经变化, 目前采用 Python 软件基金会版权协议 (Python Software Foundation License) 发布, 它是开源软件协议的一种, 与通用公共版权协议 GPL (GNU **G**eneral **P**ublic **L**icense) 兼容。与 GPL 协议不同的是, 它不强制发布修改后的软件版本必须提供源代码。

目前在计算机系统中普遍使用的是 Python 2 和 Python 3 两个大版本。Python 2.0 于 2000 年 10 月 16 日发布, Python 3.0 于 2008 年 12 月 3 日发布。Python 3 并不是 Python 2 的升级版, 它与先前的版本不兼容, 但其中的主要特性都已经做了移植, 以保持和 Python 2.6.x 和 Python 2.7.x 系列的兼容性。Python 在发展过程中, 语法规则没有发生大的变化, 不兼容主要体现在细节上, 比如 Python 2 中, `print` 是一个关键字, 而在 Python 3 中则是一个普通的函数; Python 2 的除法运算符 “/” 在进行除法运算时, 结果只保留除法的整数部分, 而 Python 3 则保留小数部分。2017 年发布的 Python 2.7.14 被确定是 2.7 系列的最后一个版本, Python 开发者希望逐渐淘汰 Python 2。最初 Python 2 的终止日期被定在 2015 年, 但因为目前尚有大量软件

[1] 官方公布的发布时间是 1991 年 2 月 20 日

在使用 Python 2, 淘汰的计划不得不向后推迟。建议读者在新开发的程序中尽可能使用 Python 3, 而本书也根据 Python 3 的规则讲解。

4.1.2 Python 的应用

Python 是多模态的编程语言, 支持完整的面向对象编程和结构化编程的特性。大量扩展模块的支持, 使其具有强大的功能, 从数学运算、数据库处理到网络和视频处理, 无不体现出其灵活和便捷。

4.1.3 Python 编辑工具

任何编程语言都需要有一个开发环境, Python 也不例外。其实, 作为像 Python 这样的脚本语言, 开发环境倒真的可有可无, 只要有文本编辑工具就够了。但是程序写出来后总要运行、调试吧? 即使以源码形式发布的 Python 软件, 安装到用户计算机上, 也要求该计算机具备 Python 运行环境。

Python 官方网站 https://www.python.org/downloads 提供了各种操作系统的安装包, 使用者可根据各自的需要安装相应的版本。如果愿意, 也可以下载源代码在自己的机器上编译、安装。这可是一项费时费力的活儿, 但有时候却不得不做, 比如想在某个嵌入式平台上跑 Python。Python 是跨平台的编程语言, 但运行环境与硬件平台和操作系统都有关。

Python 对 Linux 操作系统有天然的亲和力。大多数 Linux 的发行版都已默认安装了 Python 解释器, 因为 Linux 的很多基础软件要依赖 Python。安装了 Python 系统后便同时具备了 Python 的运行环境和开发环境。Python 运行环境是 Python 源程序的解释器 python3 (Windows 系统则是 python3.exe) 和相关的动态链接库。如果是 Python 2, 可执行程序名则是 python。Python 的集成开发环境 idle3.6 (Python 2 对应的名字是 `idle`) 本身就是由 Python 语言写成的。如果你有兴趣用文本编辑器打开, 它仅有下面的寥寥数行:

清单 4.1 idle3.6

```
#!/usr/bin/python3

from idlelib.PyShell import main
if __name__ == '__main__':
    main()
```

它提供了一个运行 Python 指令的交互界面 (Python shell) 和符合 Python 编程风格的编辑工具。不得不承认, Linux 系统中的 Python 环境`idle` 确实过于简陋。不过 Linux 系统的开发人员本来也不喜欢集成开发环境, 而更多的是利用功能强大的编辑器编辑源代码, 特别像 Python 这样简洁的编程语言, 集成开发环境对程序员来说更是累赘。良好的编辑工具有助于提高效率, 减少差错。由于 Python 对源程序版面格式有一定的要求, 针对 Python 语言设计的编辑工具也方便了程序的输入。有些其他的编辑工具 (如 vim、emacs 等) 通过适当的配置也可以很方便地用于编辑 Python 源程序。

即使不启动idle, 在终端上输入不带任何参数的 python 命令本身, 也会进入 python 交互环境。在这个环境下, 可以逐条输入 Python 代码, 随时获得这行或这段代码的结果。

4.2 Python 语言基础

4.2.1 运行 Python 程序

按照惯例, 学习程序语言的第一个例子总是从一句问候语开始:

清单 4.2 hello.py

```
print("Hello, Python!")
```

这一行很容易看懂, 但似乎有点太简单了。计算机怎么知道按照 Python 的规则运行呢?

将含有上面一行语句的内容保存成一个文件, 比如说 hello.py, 它就是 Python 的源程序。在安装了 Python 的 Windows 系统中, 该文件会以 Python 的图标提示用户。只需要用鼠标点击它, 系统便自动调用 Python 解释器运行这个程序, 打开一个窗口, 打印一串字符。只是由于过程太短, 随着程序的结束, 打开的窗口瞬间就关闭了。

在 Linux 系统上有多种运行方式: [2]

- Linux 操作系统习惯使用终端操作。因此可以直接在终端中运行下面的命令:

  ```
  $ python3 hello.py
  ```

  程序在终端上打印一句问候语。

- 将下面的内容写在源程序文件的第一行:

  ```
  #!/usr/bin/python3
  ```

  并通过命令 "`chmod +x hello.py`" 为该文件设置可执行属性, 该文件形式上就成为一个独立的可执行文件, 可以通过下面的命令直接运行:

  ```
  $ ./hello.py
  ```

  文件开头的两个字节 "`#!`" 是脚本文件的标识, 系统会根据后面指定的程序作为解释器。

- 在.py 文件中指定了解释器且设置了可执行属性后, 桌面环境中使用鼠标点击该文件的图标, 也可以运行程序。这种形式与其它桌面软件的运行方式相同。

Python 一行可以有多个语句, 每个语句之间用分号分隔, 最后一个语句后面的分号可有可无。表达式之间可以有空格, 但每一行语句最前面的空格 (缩进) 是有意义的, Python 用缩进表示程序的逻辑层次, 不要随意增减。上面这个小程序, 每行代码都应该顶格书写。

[2] 类 Unix 系统, 包括 Unix、iOS 及 Linux 的各种发行版, 运行方式基本相同。

4.2.2 交互方式

终端上键入 python3, 或者 idle3&, 可以启动交互方式, 在提示符 “>>>” 下可以直接输入 Python 语句:

```
Python 3.5.2 (default, Sep 14 2017, 22:51:06)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello, Python!")
Hello, Python!
>>>
```

这个环境下的操作不能保存, 语句写错了也不便于修改, 因此不是一个真正的编程环境, 但对于一些简单的操作和不太复杂的程序来说, 具有反应及时, 操作便捷的优点。作为基本操作, 我们利用这个环境学习 Python 的一些基础知识。需要查询一些内建函数和对象的文档, 可以简单地调用 help() 方法；对于非内建模块, 导入后也可以使用 help() 查阅。

idle 则是一个图形化工作环境, 通过 File 菜单的 “打开” 或 “新建” 子菜单, 会创建一个新的编辑窗口, 真正的编程工作可以在这个环境下完成。

4.2.3 数值表示

常数

Python 中, 表示常数的数值类型有整数、浮点数和 复数三种。书写时, 没有小数点的数字被当作整数处理, 数字中带有小数点的被当作浮点数处理, 如果数字后面紧跟着字母 “j” 或 “J”, 则这个数就是复数, 字母 “j” 或 “J” 是虚数单位。以上需要注意的是:

- 在 Python 2 中, 整数之间的四则运算 (主要是除法运算) 结果总是整数, 因此 3/4=0。如果要提高计算精度, 必须至少将其中的一个数字用浮点数表示。但 Python 3 改变了这一规则, 运算结果总是保持足够的精度, 因此 3/4=0.75。如果仍想让结果保持整数, 则应使用整除运算符 “//”。整除运算的结果并不一定是整数类型, 只是小数部分为零。
- 虚数单位 “j” 或 “J” 必须紧跟在数字后面, 中间不能写乘号 “*”, 也不能写在数字前面, 否则会被作为变量对待。

试看下面的操作:

```
>>> 12345678+87654321
99999999
>>> 1000/330
3.0303030303030303
>>> (1 + 2j)*(3 - 2j)
(7+4j)
>>> 2.8 // 0.03
```

```
93.0
>>> (1 + j)*(1 - j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'j' is not defined
```

缺省的数字格式被作为十进制数字处理。Python 语言中可以将数值用 二进制、八进制和十六进制表示, 表示方法是在数字前面分别加上 0b、0o、0x (字母不分大小写。三个字母分别是 **b**inary、**o**ctal 和 **h**exadecimal 三个单词的首字母), 例如:

```
>>> 0xff - 0b1111
240
```

变量

变量是用于存储信息的单元。Python 中没有单独声明变量的语句, 为变量赋值无需事先定义, 赋值的同时也就声明了这个变量。未经赋值的变量则被认为是不存在的对象, 不能参与运算。看下面的例子:

```
>>> a1 = 10
>>> a2 = a2 + a1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a2' is not defined
>>> a2 = 20
>>> a2 = a2 + a1
>>> a1, a2
(10, 30)
```

在多数计算机语言中, 单个 “=” 符号不是等号, 而是赋值号, 它表示将该符号右边的表达式结果传递给左边的变量中。等号有专门的表示方法, 我们在稍后会看到。

Python 允许同时为多个变量赋值。基于此, 若干个变量交换也只需要一个语句:

```
>>> a, b = 200, 'hello'
>>> b, a = a, b
>>> print ('a=%s,' % a, 'b=%d' % b)
a=hello, b=200
```

上面使用了函数 `print()` 的格式化打印功能, 引号中的 “%” 表示打印变量的属性, 而引号外面的 “%” 用于关联格式变量。表4.1列出了常用的打印格式。

格式字母前面可以有数字, 表示格式化字符的长度, 长度不足时, 前面用空格或 “0” 填充:

```
>>> "%4x" % 123
```

表 4.1 变量的格式化表示

格式	含义
%s	字符串
%d, %i, %u	十进制整数
%o	八进制整数
%x, %X	十六进制整数
%e, %E	科学记数格式
%f, %F	浮点数

```
'  7b'
>>> "%04X" % 456
'01C8'
```

合法的变量名由若干个字母、数字和下划线构成, 字母区分大小写, 首个字符不能是数字。由于 Python 3 支持 Unicode, 因此, 如果使用 Unicode 中文编码, 像下面的名称也是合法的:

```
>>> 张三的年龄 = 32
>>> 李四 = 张三的年龄 + 3
>>> print("李四的年龄是 ", 李四)
李四的年龄是 35
```

但不建议使用这种命名方式。

4.2.4 字符串

字符串的表示

变量除了可以保存数值信息, 也可以保存非数值信息。字符或字符串 就是典型的非数值信息:

```
>>> greeting = "Hello, Python!"
>>> print (greeting)
Hello, Python!
>>> number = 35
>>> number + number
70
>>> string = "35"
>>> string * 3
'353535'
```

一对引号之间的内容被当作字符串对待, 即使是数字, 也不具备数值含义。

表示字符串的引号可以用单引号也可以用双引号, 但前后不能混用。如果字符串本身含有引号, 可以使用下面的方法之一处理:

- 交替使用引号。例如字符串中有双引号时, 则可用单引号作为字符串的引号。
- 在字符串中的引号前面加反斜线 “\”, 进行转义处理。

反斜线被称作转义符, 它表示后面紧跟的一个字母或符号将改变原有的含义。转义符通常还用于处理一些不能直接用字母表示的场合, 例如换行 (“\n”)、制表符 (“\t”)、续行 (“\”) 等等。

多行文本可以使用三引号 ''' 或 """。

字符串除了实现程序功能以外, 也是程序注释和帮助文档的重要组成部分。

Python 中的字符串有三种类型: 裸字符串 (raw string)、Unicode编码字符串和二进制字符串 (binary string), 分别用对应单词的首字母前缀声明。像下面的例子:

```
>>> rStr = r"Python is easy."
>>> uStr = u'Python isn\'t hard to learn.'
>>> bStr = b'Hello.!'
```

对于英文 ASCII (**A**merican **S**tandard **C**oding for **I**nformation **I**nterchange) 字符和符号, 裸字符串和 Unicode 字符串是等价的 (裸字符串不接受转义符), 但对其他语言文字则不同。我们在使用中文时特别需要注意。在中国大陆, 中文存在两种不同的编码方案 (中国国家标准 GB18030 和国际标准 Unicode), 他们在计算机中的信息存储方式互不兼容, 一些中文显示乱码的根源就在于此。Python 建议将字符串用 Unicode 表示。因此, 一个标准的 Python 源程序开头两行应该是这样的:

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

“#” 在 Python 中可以作为单行注释符, 但这两行的 # 比较特殊, 不完全是注释。UTF-8 是 Unicode 实现方案的一种, 它使用若干个 8 位的二进制按一定的规则拼接表示一个文字符号的 Unicode。英文字符的 ASCII 码和 UTF-8 码恰好相同。使用二进制字符串定义的字符, 则可以直接得到字符的 ASCII 码:

```
>>> print (bStr[0],bStr[1],bStr[2])
72 101 108
```

字符串的基本运算

字符串属于常量的一种, 一旦设定, 其中的元素便不可更改, 例如:

```
>>> str = 'hello'
>>> print (str[0])
h
>>> str[0] = 'H'
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  TypeError: 'str' object does not support item assignment
```

Python 定义了字符串的加法和字符串与整数的乘法。字符串相加就是将字符串按书写顺序前后串接；字符串与整数相乘, 等效于若干个字符串相加。

字符串对象

字符串 (str) 在信息处理中是最基本也是最常用的对象, Python 为其构建了一整套处理函数。表4.2 – 4.4 列出了字符串对象的主要方法。

表 4.2 字符串对象的常用函数–字符转换

函数	功能
capitalize()	将首字母大写, 其它字母小写
lower()	将全部字母变成小写
upper()	将全部字母变成大写
swapcase()	将全部字母大小写互换
center(width[, fill])	使字符串居中, 两边用 “fill” 填充
encode(encoding)	按 “encoding” 设定的方式编码
expandtabs(tabsize)	将字符串中的制表符转换成 “tabsize” 个空格
replace(old, new[, count])	字符串替换
title()	标题化字符串 (每个单词首字母大写)
ljust(width[,fill])	字符串左对齐, 右边填充 “fill”
rjust(width[,fill])	字符串右对齐, 左边填充 “fill”
lstrip(chars)	去掉字符串左边的 “chars” (缺省时去掉空格)
rstrip(chars)	去掉字符串右边的 “chars” (缺省时去掉空格)
strip(chars)	去掉字符串两边的 “chars” (缺省时去掉空格)
translate(table)	返回字符串在映射表 “table” 中的拷贝
zfill(width)	在数字字符串前面用 “0” 填充到长度 “width”

由于字符串变量的不可改变属性, 字符转换的结果返回一个新的字符串, 而原字符串变量不变。例如:

```
>>> a = 'hello, python'
>>> b = a.title()
>>> print (a + '\n' + b)
hello, python
```

```
Hello, Python
```

表 4.3 字符串对象的常用函数–字符特征

函数	功能
isalnum()	是否都是是字母或数字
isalpha()	是否都是字母
isdigit()	是否都是数字 (包括编码数字, 罗马数字)
islower()	是否都是小写
isspace()	是否都是空格
istitle()	是否符合标题格式
isupper()	是否都是大写
isidentifier()	是否为识别符 (语法关键字)
isdecimal()	是否是数字 (不包括包括罗马数字)
isnumeric()	是否是数字 (不包括字节数字)
isprintable()	是否都是可打印字符
endswith(suffix)	是否以 “suffix” 结尾
startswith(prefix)	是否以 “prefix” 开头

表 4.4 字符串对象的常用函数–查找与处理

函数	功能
count(sub[,start, end])	返回子串 “sub” 的数目
join(iterable)	将可迭代数据 “iterable” 用字符串串接
partition(sep)	以 “sep” 为中心将字符串分割成首尾三段
rpartition(sep)	类似 partition, 但是从字符串尾部开始
split(sep, max)	以 “sep” (或空字符) 为分隔符将字符串分割成列表
rsplit(sep, max)	与 split() 类似, 但方向是从后往前
splitlines()	以断行符为特征将字符串分割成列表
find(sub[,start, end])	查找子串 “sub” 的位置, 不存在时返回 -1
rfind(sub[, start, end])	与 find() 功能类似, 但是从字符串尾部开始
index(sub[,start, end])	与 find() 功能类似, 子串 “sub” 不存在时报错
rindex(sub[, start, end])	与 index() 类似, 从字符串尾部开始
format(...)	将变量进行格式化输出

format() 常用于格式化显示输出。下面是一个例子:

```
>>> "{0} + {1} = {2}".format(3, 4, 3 + 4)
'3 + 4 = 7'
```

“{ }” 中的数字对应format() 参数序号, 缺省时从 0 开始自然递增。

4.2.5 逻辑值

逻辑值又称布尔值 (boolean 的音译), 它只有 “真” 和 “假” 两个取值, 分别用 True 和 False 表示。在各类对象中, 以下几种情况的逻辑值为 False:

1. False 本身;
2. 任何类型的数值 0 (整数、浮点数、复数);
3. 长度为 0 的字符串;
4. 任何一种仅有空元素的数据结构 (空词典、0 个元素的集合、列表或元组, 以及自定义的、不包含任何成员的数据结构[3]);
5. 空对象 None。

除此以外的其他对象, 如果作为逻辑值使用时, 都是 True。

None 是一个较特殊的对象, 它表示什么对象也不是。当一个函数不返回任何值时, 调用者得到的返回值就是 None。

逻辑值常常来自条件判断, 因此 “False == 0” 条件成立, 它的结果为 True。这里连写在一起的两个等号 “==” 是表示相等的条件判断。

作为变量参与数值运算时, 逻辑变量的值是 1 (True) 或 0 (False)。

4.2.6 函数的使用

将一组实现特定功能的代码组织起来, 形成一个可供其他代码直接调用的接口, 这段代码就是函数。一个函数通常有若干个输入参数和若干个输出参数 (输出参数通常又被称为返回值)。前面用到的 print() 就是一个函数的典型例子。

再如, 内建函数 pow(x, y) 用于计算 x^y, 它甚至可以实现复数运算功能: [4]

```
>>> pow(-1, 0.5)
(6.123233995736766e-17+1j)
```

内建的函数 pow() 与 Python 的运算符 “**” 是等价的。

Python 的函数形式非常灵活, 函数可以没有输入参数; 可以有多个返回值, 也可以没有返回值。即使同一个函数, 输入参数和返回值的数量及格式都是可变的。

[3] 自定义对象的逻辑值可以根据对象本身的需要决定, 但遵循这个原则总是有好处的。

[4] 严格地说, -1 开平方的结果是 $\pm 1j$, 同其他语言的函数功能一样, Python 的数学函数只给出主值结果。这里计算出的结果实部不等于 0, 原因是由计算精度导致的。

4.2.7 导入模块

Python 已经内建了大量的函数, 但还是远远不够, 比如我们要计算以自然数 e 为底的指数, 虽然可以用 pow(2.7182818, y) 这样的形式, 但毕竟不方便, 而且谁记得 e 是多少啊！一些我们认为的常用函数 (如三角函数) 也不在内建函数中。

Python 通过导入其他模块 (俗称 “库”) 实现更为灵活的功能。上面提到的指数函数和三角函数就在数学函数模块中, 导入模块的关键字是 import。

```
>>> import math
>>> math.exp(1.0)
2.718281828459045
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/3)
0.8660254037844386
```

正是由于有丰富的模块支持, 才使得 Python 实现各种功能变得相当简单。

4.2.8 运算符和优先级

Python 一行语句可以写很长, 实际编程时, 并不主张使用过于复杂的运算, 可以用分行、加括号的方式明确运算的先后顺序。

编程语言中, 掌握运算符的用法, 准确了解各种运算的优先级, 对程序的正确性至关重要。以下分类列出 Python 使用的运算符及其作用

算术运算

表 4.5 列出的是算术运算符的功能。

表 4.5 算术运算符 (优先级由低到高)

符号	含义	说明
+, −	加减运算	
*, /	乘除运算	
//	整除	得到商的整数部分
%	取模	得到除法的余数
**	指数 (幂) 运算	

表中, 除法运算符在 Python 2 中, 用于整数之间的运算结果, 返回商的整数部分。取模运算 “%” 在字符串运算中另有含义, 见4.2.3节格式化打印的例子。“*” 和 “**” 在函数的形式参数中也不再是运算符, 在函数章节会详细讨论。

关系运算

关系运算用于比较变量的大小关系, 返回逻辑值。表4.6 列出了关系运算符的功能。

表 4.6 关系运算符

符号	含义
==	比较两个对象是否相等
!=	比较两个对象是否不等
>	前者是否大于后者
<	前者是否小于后者
>=	前者是否大于或等于后者
$<=	前者是否小于或等于后者

关系运算的总体优先级低于算术运算。关系运算中的不等于 "!=" 曾经使用过 "<>", 现已逐渐被淘汰, 建议不再使用。

位操作运算

位操作运算符以二进制位为操作单位, 它只能用于整数或之间的运算。表4.7 列出了位运算的符号和功能。位运算虽然是以二进制位为单位, 但并不要求在编程序时用二进制书写。Python

表 4.7 位操作运算符

符号	含义	说明
&	两个数的二进制按位与操作	
|	两个数的二进制按位或操作	
^	两数的二进制按位异或操作	
~	将二进制位按位取反	
<<	左移位操作	a << n, 将 a 左移 n 位, 低位补 0
>>	右移位操作	a >> n, 将 a 右移 n 位, 高位保持符号位不变

在内部会自动进行二进制处理。移位操作中, 移位次数必须是非负的整数。

赋值

除了直接赋值 "=" 以外, Python 还支持一些与运算混合的赋值方式, 以便让代码更加紧凑。表4.8 列出了所有这类赋值符号。这类赋值语句要求赋值对象必须参与运算, 并且可以成为运算式中的第一个成员。像 "a = b - a" 就无法用这类的赋值方式表示。此外, 表中运算符是一个整体, 中间不能塞空格。

表 4.8 赋值运算

符号	说明
+=	a = a + b 可以写成 a += b
-=	a = a - b 可以写成 a -= b
*=	a = a * b 可以写成 a *= b
/=	a = a / b 可以写成 a /= b
%=	a = a % b 可以写成 a %= b
//=	a = a // b 可以写成 a //= b
**=	a = a ** b 可以写成 a **= b
&=	a = a & b 可以写成 a &= b
|=	a = a | b 可以写成 a |= b
^=	a = a ^ b 可以写成 a ^= b
<<=	a = a << b 可以写成 a <<= b
>>=	a = a >> b 可以写成 a >>= b

逻辑运算

基本逻辑运算有三个, 用关键字 “and”(与)、“or” (或)、“not” (非) 表示。前两个是二元运算 (运算符前后各有一个对象), 返回两对象的逻辑运算结果。“not” 是单运算符, 它返回对象相反的逻辑结果。逻辑运算的返回值是逻辑值, 参与逻辑运算的对象可以是其他数据类型。

除了以上三个基本逻辑运算以外, 还有两个比较特殊的逻辑运算:

- is, 用于判断前后两个实例是不是同一个对象。它与条件判断 “==” 意义不同。这里, 同一个对象指的是使用同一个存储单元。同一存储单元允许有不同的符号引用, 但他们的值肯定是相等的；反之, 相等的值却不一定存储在同一个位置上。试看下面的例子:

```
>>> a = 2
>>> b = a
>>> a is b
False
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
>>> a[0] = 9
>>> print (b)
[9,2,3]
```

is 的反义运算是 is not。

- in, 用于确认一个对象是否是另一个对象的所属成员。例如:

```
>>> s = 'Hello'
>>> 'e' in s
True
```

in 后面的操作数必须是可迭代的对象, 如列表、元组等。in 的反义运算是 not in。

运算优先级

表4.9按从高到低顺序列出了 Python 运算优先级。同优先级运算中, 按语言书写顺序先左后右的顺序执行, 赋值则是从右到左的顺序。

表 4.9 运算优先级

运算符	分类
**	指数 (最高)
~	位反
*, /, %, //	乘除类
+, −	加减类
>>, <<	移位运算
&	位与运算
^ , |	位或、位异或运算
<, >, <=, >=	关系运算
==, !=	关系运算
=, +=, −=, ...	所有赋值运算
is, is not	
in, not in	
not, or, and	逻辑运算符

4.3 结构变量

4.3.1 列表

列表的形式

列表(list) 是一组有序数据的集合, 通过顺序下标 (index) 访问其中的元素。下标的序号从 0 开始, 第一个元素的下标是 0, 第二个元素的下标是 1, ...。定义一个列表时, 用 “[ ]” 将列表的元素括起来, 列表元素之间用逗号分隔。通过下标访问列表中的元素时, 下标也使用 “[ ]” 括起来。列表中的元素可以是不同的对象类型: [5]

[5]Python 中所有结构数据中的元素都不要求是同一类对象。

```
>>> lst = [-2, 'alpha', 3.14, [1,2,3]]
>>> print (lst[0] + lst[2])
1.14
>>> print (lst[3][2])
3
```

Python 中的下标可以是负数, 它表示从最后一个元素的位置倒数的序号, 即: −1 表示倒数第一个元素的位置, −2 表示倒数第二个元素的位置[6], ...。

访问列表元素时, 每次可以访问多个单元, 此时下标是一个序列。例如针对上面的 “lst” 列表, “print(lst[1:-1])” 将打印第二个元素 (下标为 1) 到倒数第二个元素 (下标为 −1 的前一个)。这里需要注意的是, 不能用 −0 表示访问到最后一个元素, 最后一个元素下标位置空缺即表示访问到最后一个元素, 如 “print (lst[1:])”。下一小节 “序列” 将对此进一步说明。

序列

上面这种下标形式, 在 Python 中被称为序列 (sequence)。一个完整的序列表示由三个数字组成: 起点、终点、步长, 每个数字之间用冒号分隔。序列中只有一个数字时, 即表示该数字本身；如果有两个数字, 这两个数字分别表示起点和终点。有冒号存在即表示前后有数字, 不论数字是否明确地写出。未写明的数字, 默认起点为 0, 终点为序列的边界 (第一个或者最后一个, 取决于步长的方向), 步长为 1。起点值被包含在序列之内, 而终点值不被包含在序列内 (未显式写出的终点值除外)。下面的例子有助于理解这些概念:

```
>>> a = range(10)      # 构造一个长度为10的序列 0,1,...9
>>> b = a[:]           # 赋值给一个新的列表
>>> c = a[-1::-1]      # 逆序后赋值给另一个变量
>>> print (c[::2])     # 打印偶数下标的元素
[9, 7, 5, 3, 1]
```

以上使用了 “b = a[:]” 的赋值形式。如果用 “b = a” 直接 赋值, 则列表 a 和 b 是同一个对象, 改变其中任何一个的元素, 另一个也会随之发生变化。

列表的运算

列表的运算主要包括相加、倍乘、片段、元素增减等等。使用基本符号运算时, 列表只定义了加法运算 “+” 和倍乘运算 “*”。加法运算中, 列表只能与列表相加, 非列表的对象必须转换成列表之后才能与列表进行加法运算；列表乘以一个正整数 (倍乘) 等效于多个列表累加。将一个元素添加到列表的首部或尾部, 可以先将该元素转换成列表再进行加法运算。如果添加在其他位置, 则必须通过函数实现。

表4.10是 Python 内建的列表函数。

[6] 此处按正常的语言习惯理解, 倒数第一个指的是最后一个。

表 4.10 与列表操作相关的函数

函数	功能	说明
list(object)	将一个可迭代的对象转换成列表	L=list() 创建一个空列表
append(object)	在列表最后添加一个元素	
clear()	清空列表	
copy()	硬拷贝列表	如果直接使用 “=” 赋值, 不会创建新对象
count(value)	返回列表中 value 的数目	
extend(object)	将一个迭代数的对象追加到列表中	
index(value, [start,[stop]])	返回第一个 value 在列表中的位置	
insert(index, object)	在 index 位置前插入一个元素	
pop(index)	返回 index 位置的元素, 并将该元素从列表中删除	index 缺省时, 表示最后一个元素
remove(value)	删除元素 value	
reverse()	列表反序	
sort(key, reverse)	根据 key 的规则排序, reverse 为 True 则反序	

可迭代对象是由多个元素组成的数据结构, 包括字符串、序列、列表、元组、集合、词典等。字符串转换成列表时, 其中的每一个字符成为列表中的一个元素。

表4.10中除了 `list()` 以外, 其余都是列表对象的成员函数, 即 “L = list([1,2,3])” 创建一个列表 L, 而 “L.clear()” 清除列表 L 中的元素。

下面的程序实现一组列表的操作:

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

season = ['立春', '立夏', '立秋', '立冬']     # 创建一个列表
season.insert(1, '雨水')                      # 插入一个元素
season.extend(('小雪', '大雪'))               # 将元组元素追加到列表
index = season.index('立秋')                  # 得到``立秋''的下标
cell = season.pop(index)                      # 删除 index 处的元素
season[-1] = '大寒'                           # 修改最后一个元素
```

```
season.sort()
print ('清明' in season)
```

函数 copy() 可以创建一个新的列表对象, 但如果列表中的元素是可迭代的数据结构时, 这个结构在复制的列表和原列表中仍然是同一对象。看下面的例子:

```
lst1 = [['中秋', '重阳'], 8.15, 9.9]
lst2 = lst1.copy()
lst2[0][1] = '清明'
lst2[1] = 4.5
print (lst1)
```

完整地复制一个对象可以使用 copy 模块中的 deepcopy() 函数。

4.3.2 元组

元组 (tuple) 在很多性质上与列表完全一样, 它也是一组有序数据的集合。唯一不同的是, 它是一种不可变的数据结构, 这意味着一旦赋值, 该变量的成员不能单独修改, 除非对这个变量重新赋值。

定义元组时, 元组的元素之间用逗号分隔, 如果只有一个元素, 该元素后面也需要保留逗号, 以区分表示元素本身的对象。有时候为了避免歧义, 也可以在整组元素的外面加上括号 “( )”。访问元组中的元素时, 下标的使用方法与列表完全一致。

由于元组的不可变更属性, 它的函数比列表的函数要少得多, `index()` 是一个较常用的元组函数, 它用于获得元素所在的位置。

多数情况下, 元组的功能都可以用列表来实现。少数情况 (如词典的关键字) 是列表不能替代的。此外由于元组是不可变数据, 不用担心误操作更改了其中的元素。

4.3.3 集合

集合 (set) 包含一组无序的对象。与列表和元组不同, 由于是无序数据, 因此没有下标。与数学上的集合概念一样, 集合中没有重复的元素。

集合使用 “{ }” 符号包装元素[7], 元素之间用逗号分隔。集合的运算主要包括并集、交集、差异、查询等操作, 表4.11列出了内建的集合函数。

表 4.11 与集合操作相关的函数

函数	功能
set(object)	将一个可迭代的对象转换成集合
add(object)	向集合中添加一个元素
clear()	清空集合中的所有元素

[7]注意, 默认使用 “{ }” 创建的不是空集合, 而是空词典。

表 4.11 与集合操作相关的函数 (续)

函数	功能
copy()	硬拷贝集合。直接使用 “=” 赋值不会创建新对象
difference()	返回两个或多个集合的差集
difference_update()	按差集更新集合集
discard(value)	删除集合中的一个元素
intersection()	返回集合的交集
intersection_update()	按交集更新集合
isdisjoint()	判断两个集合是否有交集 (无交集返回 True)
issubset()	判断集合是否为另一个集合的子集
issuperset()	判断集合是否为另一个集合的超集
pop()	返回集合中的任意一个元素并将其从集合中删除
remove(value)	从集合中删除元素 value
symmetric_difference()	返回集合的对称差集
symmetric_difference_update()	以对称差集更新集合集
union()	返回若干个集合的并集
update()	更新到若干个集合的并集

表中除了函数 set() 以外, 其它均为集合对象的成员函数。S=set() 创建一个空集合。

4.3.4 词典

词典 (dictionary) 是通过关键字与对应值建立的一组数据的集合。关键字与值的对应关系用 “:” 表示, 其中关键字必须是不可变数据类型, 如数值、字符串、元组。每个元素之间用逗号分隔, 整体则放在 “{ }” 中。

词典也是无序数据, 访问词典中的元素不通过有序下标, 而是通过关键字。词典中的关键字必须是唯一的。下面是一些词典的典型操作:

```
zhang3 = {}                      # 创建一个空词典, 等效于 dict()
zhang3['name'] = 'Zhang3'        # 增加一个元素, 关键字 ``name'',
                                 # 值为 ``zhang3''
li4 = zhang3.copy()              # 复制到一个新的词典变量
li4['age'] = 23                  # 增加一个新的元素
print (zhang3, li4)
```

同列表和集合类似, 直接将词典使用 “=” 赋值不会创建新的对象。表 4.12 是词典对象的函数表。

Python 2 曾使用 `has_key(key)` 函数来判断词典中是否存在关键字 “`key`” 项, 现已逐渐用运算关系 “`in`” 取代, 而 Python 3 则直接取消了这个函数。

表 4.12 词典操作相关的函数

函数	功能	说明
dict(object)	将一个映射对象转换成词典	L=dict() 等效于 L={ }
clear()	清空词典	
fromkeys(keylist)	根据给出的关键字创建新词典	缺省方式下, 值为 None
get(key[,d])	获得词典中关键字的值	关键字 key 不存在时返回 d
items()	返回词典 (key,value) 对应关系的列表	
keys()	返回词典中关键字的列表	
pop(key[, d])	删除关键字 key 对应项, 返回对应值	删除关键字项还可使用内建关键字 “del”, 但无返回值
popitem(key, value)	删除 (key, value) 对, 将该项作为元组返回	
setdefault(key[,d])	添加/修改关键字 key 的值	
update(dict)	根据 dict 更新词典	
values()	返回词典中值的列表	

```
zhang3 = dict(name='张三', age=23)
# 等效于 zhang3={'name': '张三', 'age': 23}
zhang3['phone'] = '010-87654321'
print (zhang3['gender'])
# ``gender'' 不存在, 将触发异常错误
print (zhang3.get('gender', 'M'))
# 函数 get() 不会触发错误, 缺省方式下返回 None
extra = {'gender': 'M'}
zhang3 = zhang3.update(extra)
# 将另一个词典并入
```

4.3.5 对象

Python 内建了很多结构化数据, 面对不同的场合使用提供了很大方便。Python 是面向对象的程序, 所有数据结构都是对象, 包括在数值计算中使用的整数、浮点数也是对象。表 4.13 归纳了 Python 常用内建对象及其创建函数。

表 4.13 常用内建对象

对象名	列表	元组	集合	词典	字符串	整数	浮点数
创建函数	list()	tuple()	set()	dict()	str()	int()	float()

函数 `isinstance()` 用于判断某个对象是否属于某个类的实例, 类名称与创建函数的字母所对应的名称一致:

```
>>> a = '123'
>>> isinstance(a, int)
False
>>> isinstance(float(a), float)
True
```

4.4 Python 程序结构

4.4.1 顺序结构

我们先从一个简单的程序开始, 理解 Python 程序的书写方法:

清单 4.3 prime.py

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  a = 127
5  for i in range(2, number):
6      if number % i == 0:
7          break
8
9  if prime is False:
10     print(a, 'is a prime.')
11 else:
12     print(a, 'is not a prime.')
```

range(start, stop, step) 是一个常用的函数, 它构造一个从 start 开始、到 stop 为止、间隔 step 的数列。

Python 根据源程序的书写顺序执行, 没有特殊的起始语句或起始函数。由于它是解释性语言, 没有预处理过程, 因此每条语句执行时, 其中的所有符号必须是在之前已经声明或定义过的。这里所谓的 “前”、“后”, 不一定是体现在书写顺序上, 而是由程序运行的顺序决定。

Python 一行超过一条语句时, 语句之间用分号分隔。为增强源代码的可读性, 一般不建议一行写太长的语句。Python 使用行首空格表现代码块的层次结构, 因此对行首的空格有严格要求, 对表达式之间的空格则没有限制。每一个层次块由上一行语句末的冒号引导, 整块代码向内缩进。结束缩进时意味着块的结束。同一结构层次中要求相同的缩进, 不同结构中的缩进空格数可以不同。对于缩进, 一个总体原则是: 冒号之后的语句如果换行, 则必须缩进。

使用缩进时, 制表符和空格不能混用。一个制表符不等效于任何数量的空格, 空格 + 制表符与制表符 + 空格也不等效。多数 Python 开发人员不建议使用制表符作为缩进。

有时候, 为了源代码的紧凑, 一些简短的块可以不另起行, 直接写在块引导行的后面, 例如第 24–25 行合并为一行, 26–27 两行也可以合并为一行。

4.4.2 条件分支

与条件判断相关的关键字有 if、else、elif。条件语句通常有下面三种形式:

- 单独条件

```
a = '1234'
if isinstance(a, int):
    print ('a is an integer.')
```

- 互斥条件

```
a = 2018
if a > 0:
    print ('a is a positive.')
else:
    print ('a is not a positive.')
```

- 多重条件

```
if a > 0:
    print ('a is a positive.')
elif a < 0:
    print ('a is a negative.')
else:
    print ('a is zero.')
```

需要注意的是, `elif` 和 `else if` 不完全等价, `else` 必须构成一个新的块, 也就是说 `else` 后面必须要有一个冒号, 不能紧跟`if`。

条件语句的另一种用法是直接夹在其他语句中, 不作为块:

```
str = 'a is a positive' if a > 0 else 'a is not a positive'
```

4.4.3 循环结构

Python 中有两种循环结构: for...in和 while, 前者将某个变量在一定范围内依次取值, 属于计数式循环, 后者在满足一定条件时执行一段代码, 属于条件式循环。此外, 与循环相关的关键字还有 continue、break和 pass。continue 用于跳过本轮循环后面的语句, 进入下一轮; break 用于提前结束循环; pass 什么事也不做, 它起到语法结构作用。

- for 循环

```
days = [('Jan', 31), ('Feb', 28), ('Mar', 31), ('Apr', 30)]
for x, y in days:
    print('There are {} days in {}.'.format(y, x))
```

- while 循环

```
# 10 秒倒计时
import time
c = 10
while True:
    print (c)
    c -= 1
    if c <= 0: break
    time.sleep(1)
print ('Time up!')
```

在某些情况下, 循环可以和 else搭配使用:

```
lists = [3, 9, 4, 2, 0, 5]
for x in lists:
    if x <= 0:
        print ('Find a non-positive number')
        break
else:
    print ('All numbers in list are positive')
```

注意, 这里的 else 与 for 属同一层, 与循环中的 if 不在同一层。

与条件语句类似, 循环也可以嵌在其他语句中, 起到简化结构的目的:

```
# 构造10以内的平方表
lists = [x*x for x in range(10)]
```

4.5 本章小结

Python 是应用极广的编程语言, 大量的软件由 Python 写成, 特别是在 Linux 操作系统环境中, 很多基础软件都依赖 Python。本章介绍了 Python 的基本数据结构和语法规则。

5

GPIO 模块

模块是一组具有重复使用价值的功能性代码。合理地使用模块, 可以简化程序设计过程, 提高程序的可维护性, 控制程序规模, 同时还有助于减少错误。

5.1 Python 模块的导入

导入模块的方法有以下几种：

1. 使用 `import`直接导入模块名, 如：

   ```
   import math
   ```

   这种方法书写最简单, 但使用最麻烦。模块中的所有符号 (函数名、变量名、常量名等等) 前面都需要加上模块名。

2. 导入指定的符号：

   ```
   from math import exp, pi, sin
   ```

   以后在使用这些符号时不再用模块名为前缀, 直接使用符号名本身, 如 “print (sin(pi/6))”。未导入的符号不能使用。

   如果想导入模块中的所有符号, 一个比较粗暴的方式是：

   ```
   from math import *
   ```

   它导入除下划线 “_” 开头的所有符号。星号 “*” 在这里起到的是通配符的作用。不论该模块里有什么符号, 都可以以原符号形式使用, 不需要加前缀。虽然方便, 但却是不太推荐的一种方式。它容易引起符号使用的混淆。

3. 使用`import...as`导入符号, 并将其重新命名, 如：

   ```
   from math import exp as EXP
   ```

   之后使用模块中的符号时只能用重命名的形式, 如 `print (EXP(2.0)/10)`。重命名符号可能由于个人习惯的需要, 也可能是出于代码兼容性方面的考虑。

在 Python 环境中使用命令 `help('modules')` 可以列出当前环境下支持的所有模块。如果需要了解某个模块的具体用法, 可以在导入后使用 `help(mod_name)` 查看, 也可在命令行中使用 `pydoc mod_name` 查看。

5.2 树莓派 GPIO 模块

5.2.1 安装模块

很多程序员为树莓派的 GPIO 接口开发了各种函数库。推荐使用 Ben Croston 编写的 RP.GPIO Python 模块, 这也是官方发布的树莓派 Linux 系统缺省安装的模块。下面的命令将该模块升级到最新版:

```
# sudo apt-get update
# sudo apt-get install python-rpi.gpio python3-rpi.gpio
```

源代码托管在https://sourceforge.net, 可使用下面的命令下载:

```
$ wget https://sourceforge.net/projects/raspberry-gpio-python/files/\
latest/download -O RPi.GPIO-0.6.5.tar.gz
```

此版本在一些 Linux 系统上对硬件识别有误。https://github.com/yfang1644/RPi.GPIO 针对树莓派 64 位系统专门做了修改, 删除了对其他平台的硬件识别部分, 并增加了硬件 PWM 支持。

注意: 此修改可能导致非树莓派平台硬件识别错误!

从源码开始编译安装树莓派 GPIO 模块的过程如下:

1. 下载源码:

   ```
   # git clone https://github.com/yfang1644/RPi.GPIO
   ```

2. 进入下载目录, 编译和安装:

   ```
   # cd RPi.gpio
   # python3 setup.py build
   # python3 setup.py install
   ```

3. 安装完成后, 建议删除源码, 以免在错误的目录下导入模块错误。

5.2.2 模块基本使用规则

导入模块

习惯上, 使用下面的方式导入模块。它既缩短了名称的长度, 也兼顾了模块的可识别特征。

```
import RPi.GPIO as GPIO
```

如果不是以超级用户权限执行程序, 下面的做法保险一些:

```
try:
    import RPi.GPIO as GPIO
except RuntimeError:
    print('''权限不足。你可能需要超级用户权限使用GPIO模块''')
```

try...except是 Python 的错误处理机制。由于 Python 是解释性程序语言, 没有编译过程, 不能事先知道语法或功能性错误, 常常需要这种方法动态地排除故障。

引脚编号系统

RPi.GPIO 模块有两种引脚编号系统：BOARD 和 BCM, 前者按板上 2×20 针脚位置编号, 后者按 BCM283X 芯片引出的 GPIO 下标编号。两种编号系统在模块内部转换, 用户不必关心。用户要做的, 是在开始使用 GPIO 时用下面的函数明确其中的一种编号方式, 并在整个程序中保持这种方式:

```
GPIO.setmode(GPIO.BOARD)
  # or
GPIO.setmode(GPIO.BCM)
```

如果引脚编号方式已经确定, 函数 getmode() 用于获得当前编号方式, 返回值可以是 BOARD、BCM, 未指定编号方式将返回 None。

当配置引脚时, RPi.GPIO 模块检测到与已配置的功能发生冲突, 会打印警告信息。多数情况下, 警告只是警告, 不会产生错误。如果不想看到警告, 可以用下面的函数：

```
GPIO.setwarnings(False)
```

GPIO 输出功能

引脚使用前必须明确设定输入或输出功能。函数 setup(channel, direction) 做的就是这件事。channel 是引脚编号, direction 可以是 IN 或者 OUT, 取决于功能要求。例如：

```
GPIO.setup(18, GPIO.OUT)
```

上面的函数将 18 脚设定为输出功能 (对于BOARD 方式, 它是排针的第 18 脚, 对应 GPIO24; 对于 BCM 方式, 它是排针第 12 脚, GPIO18)。

初始化输出功能时, 还可以用 initial 参数设定初始输出电平：

```
GPIO.setup(18, GPIO.OUT, initial=GPIO.HIGH)
```

输出引脚通过函数 output() 改变输出电平：

```
GPIO.output(18, GPIO.LOW)
GPIO.output(18, GPIO.HIGH)
```

函数 setup() 和 output() 可以一次性操作多个引脚:

```
chan_list = [11, 12]    # add as many channels as you want!
                        # you can tuples instead i.e.:
                        #    chan_list = (11, 12)
GPIO.setup(chan_list, GPIO.OUT)
GPIO.output(chan_list, [GPIO.LOW, GPIO.HIGH])
```

GPIO 基本输入功能

GPIO 引脚作为输功能时, 内部带有 10kΩ 的上拉/下拉电阻, 分别接到 3.3V 和地。上拉/下拉的作用是保证引脚在未接输入信号时有一个明确的输入值, 避免一些干扰信号导致输入判断错误。上拉/下拉控制通过 `setup()` 函数的 `pull_up_down` 参数实现：

```
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
  # or
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

不设置上拉/下拉, 默认的是浮空状态。

当引脚设置为输入功能时, 函数 `input`(channel) 返回引脚的当前状态：LOW 或 HIGH。

在 RPi.GPIO 内部, `IN`/`LOW` 被定义为 0, `OUT`/`HIGH` 被定义为 1, 但不同版本不能确保一致, 建议使用符号表示, 而不是直接用数字 0/1 或逻辑值 False/True。

清除已设定的功能

程序结束之前, 将使用过的资源恢复原状, 是一个好的习惯。函数 `cleanup()` 清除引脚使用标记, 将他们恢复成输入状态。设置成输入状态可以避免因为不小心造成引脚短路而损坏器件。该函数同时也清除引脚编号系统。

```
# 清除所有引脚设置
GPIO.cleanup()

# 清除部分引脚功能
GPIO.cleanup(channel)
GPIO.cleanup((channel1, channel2))
GPIO.cleanup([channel1, channel2])
```

5.2.3 使用 GPIO 模块控制 I/O 设备

仍按27页图3.3连接设备, GPIO18 控制 LED, GPIO8 控制一个开关。由于开关接通时输入低电平, 因此悬空时应使其内部电阻上拉, 确保未接通时读到的是高电平。下面是一个简单的程序, 开关每接通一次, LED 状态改变一下。

清单 5.1 switch.py

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  import RPi.GPIO as GPIO
5  import time
6
7  LED = 18
8  SWITCH = 8
9  status = GPIO.LOW
10 GPIO.setmode(GPIO.BCM)
11 GPIO.setup(LED, GPIO.OUT, initial=status)
12 GPIO.setup(SWITCH, GPIO.IN, pull_up_down=GPIO.PUD_UP)
13
14 while True:
15     if GPIO.input(SWITCH) == GPIO.LOW:
16         if status == GPIO.LOW:
17             status = GPIO.HIGH
18         else:
19             status = GPIO.LOW
20         GPIO.output(LED, status)
21     time.sleep(0.5)
```

5.3 函数与模块

模块化是软件设计中常用的方法, 它有助于隐藏无关的细节, 化繁为简, 使一些看上去很困难的问题变得相对简单。对比清单3.1和5.1 可以看到, 后者不仅功能实现简单, 程序结构也清楚了许多。

根据不同层次要求, Python 的模块化可通过函数或模块实现。

编写下面的函数:

```
def toggle(ledon):
    if ledon == GPIO.LOW:
        ledon = GPIO.HIGH
    else:
        ledon = GPIO.LOW
    GPIO.output(LED, ledon)
    return ledon
```

清单5.1第 16—20 行可以用一个函数调用 status = toggle(status) 简化。

5.3.1 函数的结构

函数又叫子程序。关键字 def定义函数的名称, 函数的形式参数列表写在括号里。

主程序调用函数时, 有下面几种参数传递的形式：

- 参数顺序一一对应。这种形式比较容易理解和接受, 也是其他很多编程语言普遍采用的形式：

```
def add(x, y):
    return x + y

print (add (3, 5))                # This will print "8"
print (add ('Hello', 'world'))    # This will print "Helloworld"
```

- 给形式参数一个名称, 并设定一个缺省值。此时参数首先根据关键字对应, 在不提供参数关键字时再根据顺序对应：

```
def add(string='hello', number=1):
    return string*number

print (add ())                    # This will print "hello"
print (add (number=2))            # This will print "hellohello"
```

- 形式参数任意, 实际参数由调用的主程序决定。这种形式可以和前两种结合使用, 所有任意形式参数表放在形式参数的最后, 作为元组传递给子程序：

```
def add(title, *args):
    print (title, end='=')
    return sum(args)

print (add('total',1,2,3,4))      # This will print "total=10"
```

- 形式参数任意, 实际参数以关键字传递给子程序, 所有关键字和对应值组成一个词典传递给子程序：

```
def add(**args):
    return sum(args.values())

print (add(a1=2, x=3, b=4, z3=-5))
# This will print "4"
```

关键字 return结束子程序, 并向主程序返回结果。结果可以是子程序运行结果, 也可以是子程序运行的状态, 甚至可以没有返回值。没有返回值时, 主程序调用后得到的是 None。

子程序向主程序的返回值数目原则上应与主程序接收的变量数相对应。当使用单个变量接收多个返回值时, 返回值以元组的形式向变量赋值。如果多个变量接收返回值, 且与返回值数目不等, 则会导致 ValueError 错误。

```
def algorithm(a, b):
    return a+b, a-b

x, y = algorithm(20, 5)
```

5.3.2 模块化方法

一段 Python 代码独立保存成文件时即成为一个模块, 文件名 (不包括 ".py") 就是模块名, 这使得 Python 构造模块十分简单。`import`命令会执行模块中的所有可执行部分。为使模块中的符号可重复使用, 功能性代码应通过函数封装起来。利用一个特殊的变量 `__name__` 将主程序与可导入符号分离, 对于模块来说, `__name__` 是模块的名称, 对于主程序来说则是字符串"`__main__`"。这个变量可以让一个独立运行的程序也可以作为模块使用, 提高了代码的重复利用率。例如:

清单 5.2 fibo.py

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

# Fibonacci numbers module

def fib(n):             # print Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print (b, end=', ')
        a, b = b, a+b

def fib2(n):            # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__ == '__main__':
    fib(30)
```

上面的程序可以独立运行:

```
$ python3 fibo.py
```

```
1, 1, 2, 3, 5, 8, 13, 21,
```

同时, 另一个模块可以使用 fibo.py 中的函数：

清单 5.3 func.py

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

from fibo import fib2

if __name__ == '__main__':
    print(fib2(200))
```

运行结果如下：

```
$ python3 func.py
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

同样, func.py 也可以继续作为模块, 供其他模块使用。尽管在这个例子中, func.py 没有实质性内容, 但它导入的模块 `fib2` 可以传递到上层。

当使用 `import`导入模块时, Python 首先寻找是否存在该名称的内建模块, 如果没有, 则按 sys.path 列出的目录表顺序查找。通常, 当前目录是这个表中的第一项, 因此, 置于当前目录的模块总是以第一顺序被导入。

5.3.3 以目录名作为模块

目录本身就可以成为 Python 的一个模块, 其下的 py 文件和目录则成为该模块的子模块。其中的文件 __init__.py 承担该模块的初始化功能。导入模块时将首先运行 __init__.py。`__init__()` 是所有对象的初始化函数, 一些初始化工作可以写在这个函数里。这里创建的 __init__.py 承担的是同样的功能。Python 这种构建模块的方法使得开发复杂功能的模块便得简单, 模块也更容易维护。例如, 我们可以这样构建一个模块目录：

```
fibo                (目录)
  |-- __init__.py   (空文件)
  |-- fib.py        (函数fib())
  `-- fib2.py       (函数fib2())
```

当目录 fibo 属于 `sys.path` 的一部分时, 可以通过下面的方式使用这个模块：

```
from fibo import fib, fib2
a = fib.fib(30)
b = fib2.fib2(100)
```

5.4 GPIO 高级输入功能

程序检测 GPIO 输入状态的原始方法是在循环中反复读取端口状态, 在计算机术语中, 这种方式被叫做 “查询方式”。这种方式会带来很多问题。首先, 它非常消耗 CPU 资源; 第二, 如果该端口长期没有响应, 程序会被这个端口套死, 不能及时处理其他事务; 第三, 响应事件不够及时(程序5.1的响应时间是 0.5 秒, 缩短响应时间必然加重 CPU 的负担。), 如果事件在这 0.5 秒之间发生过并且又复原, 就会漏掉一次处理。

除了查询方式以外, RPi.GPIO 提供阻塞和中断方式用于输入功能。GPIO 具有对输入信号沿跳变的捕获能力。从低电平到高电平的变化瞬间称为上升沿 (rising egde), 从高电平到低电平的变化瞬间称为下降沿 (falling edge)。

5.4.1 阻塞方式

下面的例子等待 GPIO 的输入, 在给定时间内如果没有输入则不再等待。可检测的变化状态包括 `RISING`、`FALLING` 或 `BOTH`。如果不设置 `timeout` 参数则会无休止地等待, 直到状态发生变化。

```
...
channel = 8
# wait for up to 5 seconds for a falling edge
# (timeout is in milliseconds)
channel = GPIO.wait_for_edge(channel,
                             GPIO.FALLING
                             timeout=5000)
if channel is None:
    print('Timeout occurred')
else:
    print('Edge detected on channel', channel)
```

等待时间消耗的 CPU 资源很低, 只是程序在此期间不能处理其他事务。

另一种方案是安排事件检测, 过一段时间再来检查看看是否事件已经发生, 做法是：

```
# add falling edge detection on a channel
GPIO.add_event_detect(channel, GPIO.FALLING)

do_something()

if GPIO.event_detected(channel):
    print('Button pressed')
```

5.4.2 中断方式

中断方式通过设置线程回调函数实现并行处理, 它可以即时响应预先安排的事务。

```
def my_callback(channel):
    print('This is a edge event callback function!')
    print('Edge detected on channel ', channel)

# add falling edge detection on a channel
GPIO.add_event_detect(channel,
                      GPIO.FALLING,
                      callback=my_callback)
time.sleep(300)
```

sleep() 函数可以换成其他任何事务处理代码, 只要程序不结束, 回调函数就有效。回调函数是一种特殊的函数, 它不会被显式地调用, 而是在满足预先设定的条件时处理器转去执行, 因此, 它的形式参数是确定的, 通常也没有返回值。

一个事件可以设置多个回调函数, 但不允许多个事件使用同一个回调函数。多个回调函数通过下面的方法设置:

```
...
def my_callback_one(channel):
    print('Callback one')

def my_callback_two(channel):
    print('Callback two')

GPIO.add_event_detect(channel, GPIO.RISING)
GPIO.add_event_callback(channel, my_callback_one)
GPIO.add_event_callback(channel, my_callback_two)
...
```

这种情况, 回调函数是顺序执行的, 而不是并行执行的, 因为在 RPi.GPIO 内部为每个事件只安排了一个线程。

5.4.3 消除抖动

当输入设备是按键时, 如果机械开关设计不好, 手工按键时总会发生一些抖动, 一次按键会被识别成多次输入。消除抖动有硬件方法也有软件方法。硬件方法是用一个电容滤除高频信号; 软件方法是设置一个等待时间: 当检测到有按键动作时, 等一段时间再读按键的状态, 这时候的状态就是稳定的了。根据人的生理特点, 这段时间大概是数十毫秒量级。

去抖动参数可以在 add_event_detect() 和 add_event_callback() 时通过 bouncetime 设置, 单位是 ms, 例如:

```
# add falling edge detection on a channel, ignoring further
# edges for 200ms for switch bounce handling
GPIO.add_event_detect(channel,
                      GPIO.FALLING,
                      callback=my_callback,
                      bouncetime=200)
# or
GPIO.add_event_callback(channel,
                        my_callback,
                        bouncetime=200)
```

事件检测功能使用完毕, 应通过remove_event_detect() 将其释放:

```
GPIO.remove_event_detect(channel)
```

5.5 PWM 模块

RPi.GPIO 模块中, PWM需要设置两个参数：频率 (单位: Hz) 和占空比 (单位: 百分比)。所有 GPIO 引脚都可以设置为 PWM 功能, 但只有 GPIO18 和 GPIO19 是硬件 PWM, 其他是通过软件实现的 PWM。软件 PWM 消耗更多的 CPU 资源, 且频率和占空比的稳定性和准确性都不能保证。

创建 PWM 模块首先要将相应引脚设置为 GPIO 的输出功能：

```
GPIO.setup(channel, GPIO.OUT)
p = GPIO.PWM(channel, frequency)
```

通过下面的函数启动 PWM 的工作：

```
# dc is the duty cycle (0.0 <= dc <= 100.0)
p.start(dc)
```

启动后通过下面的函数改变频率和占空比：

```
p.ChangeFrequency(freq)      # freq is the new frequency in Hz.
p.ChangeDutyCycle(dc)        # where 0.0 <= dc <= 100.0
```

停止 PWM:

```
p.stop()
```

软件 PWM 通过线程实现, 不要在同一只引脚上反复创建 PWM 对象。

当 GPIO18 (BOARD 模式 12 脚) 接 LED 时, 下面的例子按 2 秒的节奏闪烁：

```
...
GPIO.setmode(GPIO.BOARD)
GPIO.setup(12, GPIO.OUT)

p = GPIO.PWM(12, 0.5)             # 0.2Hz
p.start(1)                        # 占空比 1%
input('Press return to stop:')    # raw_input() in Python 2
p.stop()
GPIO.cleanup()
```

下面的程序使 LED 渐亮/渐暗：

```
...
GPIO.setmode(GPIO.BOARD)
GPIO.setup(12, GPIO.OUT)

p = GPIO.PWM(12, 50)              # channel=12 frequency=50Hz
p.start(0)
try:
    while True:
        for dc in range(0, 101, 5):
            p.ChangeDutyCycle(dc)
            time.sleep(0.1)
        for dc in range(100, -1, -5):
            p.ChangeDutyCycle(dc)
            time.sleep(0.1)
except KeyboardInterrupt:
    pass
p.stop()
GPIO.cleanup()
```

下面的程序在 GPIO18 输出变化的频率, 插上耳机可以听到一个音阶：

清单 5.4 notes.py

```
...
freq = [220*2**(x/12.0) for x in (0, 2, 4, 5, 7, 9, 11, 12)]

p = GPIO.PWM(18, freq[0])
p.start(50)
for x in freq:
    p.ChangeFrequency(x)
```

```
    time.sleep(0.5)
...
```

软件 PWM 产生的声音可以明显地听出抖动。

5.6 本章小结

模块化是软件设计中常用的方法。本章介绍了如何导入模块、引用模块的符号、以及创建模块的方法。RPi.GPIO 是树莓派平台上常用的一个控制 GPIO 的 Python 模块。本章也介绍了 RPi.GPIO 基本输入/输出控制的函数。

基本 GPIO 用法小结：

```
# 导入模块
import RPi.GPIO as GPIO

# 设置引脚编号模式 (BCM 或 BOARD)
GPIO.setmode(GPIO.BCM)

# 设置引脚输入或输出功能 (IN 或者 OUT)
# 对于输入功能，可以同时设置上拉或下拉电阻 (PUD_UP 或 PUD_DOWN)
GPIO.setup(channels, GPIO.IN, pull_up_down=GPIO.PUD_UP)

# 读取输入状态 (返回 LOW 或者 HIGH)
state = GPIO.input(channel)

# 对于输出功能，可以同时设置输出初始值 (LOW 或者 HIGH)
GPIO.setup(channels, GPIO.OUT, initial=GPIO.LOW)

# 改变输出电平 (LOW 或者 HIGH)
GPIO.output(channels, GPIO.HIGH)

# 创建 PWM 对象，设置初始频率 (Hz)
p = GPIO.PWM(channel, frequency)

# 启动 PWM 输出 (占空比0--100)
p.start(dutycycle)

# 改变频率和占空比
p.ChangeFrequency(frequency)
p.ChangeDutyCycle(dutycycle)
```

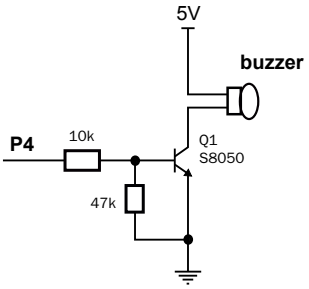
```
# 停止 PWM 输出功能
p.stop()
```

6

传感器和控制器

6.1 蜂鸣器

蜂鸣器是一种一体化结构的电子发声设备, 主要有压电式和电磁式两种类型。多谐振荡器通电源后输出 1.5~2.5kHz 的音频信号, 推动压电蜂鸣片发声。



6.2 操纵杆

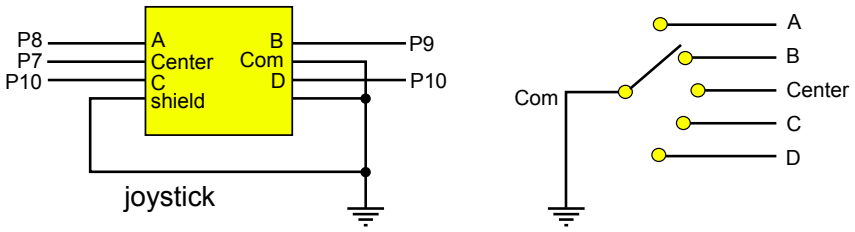


图 6.1 操纵杆

操纵杆由 5 个开关组成, 与其说是电子设备, 勿宁说更是一个机械设备。操纵杆的电极与公共端相连, 其他 4 极 A、B、C、D 以及中心极 Center 分别连至输入端接口。计算机根据读到的端口状态判断操纵杆当前处于什么位置。

清单6.1是操纵杆示例。

清单 6.1 joystick.py

```
1 #!/usr/bin/python3
2 # -*- coding:utf-8 -*-
3
4 import RPi.GPIO as GPIO
```

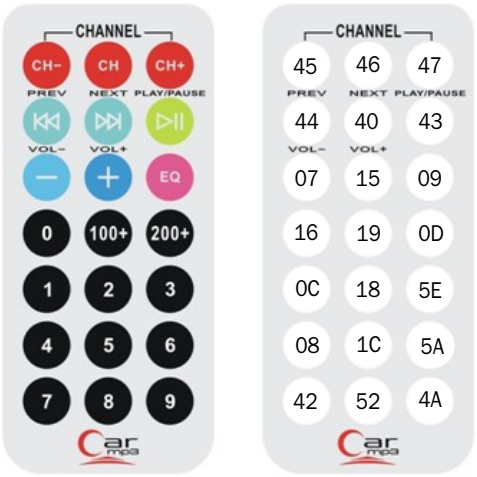
表 6.1 AlphaBot2 资源分配表

Item	GPIO	Item	GPIO
Buzzer	GPIO4	Joystick	GPIO7–11
IR sensors (Bottom)	GPIO5(CS)	Wheel A	GPIO12–13
(TLC1543)	GPIO25(CLK)		GPIO6 (CTRL)
	GPIO24(ADDR)	Wheel B	GPIO20–21
	GPIO23(DOUT)		GPIO26(CTRL)
IR remoter recv.	GPIO17	Servo (I2C)	GPIO2(SDA)
Color LEDs (Bottom)	GPIO18	(PCA9685)	GPIO3(SCL)
UltraSonic	GPIO22(Trig)	IR (Front-R)	GPIO19
	GPIO27(Echo)	IR (Front-L)	GPIO16

```
5  import time
6
7  CENTER = 7
8  FRONT = 8
9  RIGHT = 9
10 LEFT = 10
11 BACK = 11
12 pingroup = (LEFT, RIGHT, CENTER, FRONT, BACK)
13
14 GPIO.setmode(GPIO.BCM)
15 GPIO.setup( pingroup, GPIO.IN, GPIO.PUD_UP)
16
17 message = {CENTER: 'center', LEFT: 'left', RIGHT: 'right',
18            FRONT: 'front', BACK: 'back'}
19
20 def pin_catched(channel):
21     print (message[channel])
22
23 for pin in pingroup:
24     GPIO.add_event_detect (pin, GPIO.FALLING, callback=pin_catched)
25
26 while True:
27     time.sleep(5)
```

6.3 红外遥控器

红外遥控器发射的信号使用 38kHz 左右的载波对基带进行调制。接收端对信号监测、放大、滤波、解调等等一系列处理, 然后输出基带信号。收发方采用约定的协议进行通信。图6.2是 NEC 红外通信协议发送信号的波形。



发送端首先发送一个 9ms 低电平 +4.5ms 高电平的引导码, 接收方检测到引导码后开始识别后面的数据：0.56ms 低 +0.56ms 高表示 “0”, 0.56ms 低 +1.69ms 高表示 “1”。一组 01 序列构成一个按键特征字。接收方根据收到的特征字产生一定的动作。

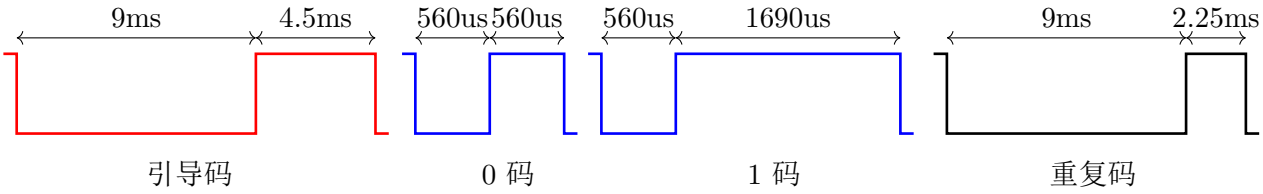


图 6.2 NEC 红外遥控器协议

清单6.2是红外接收示例程序。

清单 6.2 irremote.py

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

import RPi.GPIO as GPIO
import time

IR = 17

GPIO.setmode(GPIO.BCM)
GPIO.setup(IR, GPIO.IN, GPIO.PUD_UP)

keymap = {
    0x45: 'CH-', 0x46: 'CH ', 0x47: 'CH+',
    0x44: '<<<', 0x40: '>>>', 0x43: '>>|',
    0x07: ' - ', 0x15: ' + ', 0x09: 'EQ ',
    0x16: ' 0 ', 0x19: '100+', 0x0D: '200+',
    0x0C: ' 1 ', 0x18: ' 2 ', 0x5E: ' 3 ',
```

```
    0x08: ' 4 ', 0x1C: ' 5 ', 0x5A: ' 6 ',
    0x42: ' 7 ', 0x52: ' 8 ', 0x4A: ' 9 '}

def getkey():
    if GPIO.input(IR) == GPIO.HIGH:
        return
    channel = GPIO.wait_for_edge(IR, GPIO.RISING, timeout=8)
    GPIO.remove_event_detect(IR)
    if channel is not None:          # less than 9ms, maybe noise
        return

    time.sleep(0.0045)               # wait 4.5ms

    data = 0
    for shift in range(0, 32):
        # 0.56ms low level
        while GPIO.input(IR) == GPIO.LOW:
            time.sleep(0.0001)

        count = 0
        while GPIO.input(IR) == GPIO.HIGH and count < 10:
            count += 1
            time.sleep(0.0005)

        # "0": 0.56ms, "1": 1.69ms
        if (count > 1):
            data |= 1<<shift

    a1 = (data >> 24) & 0xff
    a2 = (data >> 16) & 0xff
    a3 = (data >> 8) & 0xff
    a4 = (data) & 0xff
    if ((a1+a2) == 0xff) and ((a3+a4) == 0xff):
        return a2
    else: print("repeat key")

print('IRremote Test Start ...')

```

```
56 while True:
57     key = getkey()
58     if(key != None):
59         print('key = ', keymap[key])
```

红外遥控是常见的控制设备, Linux 系统本身有完善的驱动。当加载了对应的设备驱动程序以后, 读取红外设备有更简单的方法：

1. 加载设备树文件。在系统引导区的 config.txt 文件中加入如下一行：

   ```
   dtoverlay=gpio-ir,gpio_pin=17,gpio_pull=up
   ```

2. Linux 系统启动后加载 GPIO 的红外遥控接收端驱动:

   ```
   # modprobe gpio-ir-recv
   ```

   然后设定 NEC 协议:

   ```
   # echo "nec" >/sys/class/rc/rc0/protocols
   ```

   驱动加载后应可在/dev/input 目录下看到一个设备文件 event0。如果之前已有 input event 设备, 文件名的序号有可能增加, 可通过查阅 /proc/bus/input/devices 文件获知红外遥控对应的设备文件名。

   以上加载设备驱动和设置协议命令可写入系统启动脚本。

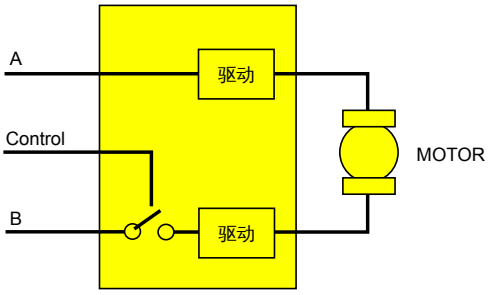
3. 红外遥控二极管接受到红外信号后, 设备驱动会自动进行解码。每次读取 /dev/input/event0 可得到 48 个字节的数据, 其中 20–23 这 4 个字节对应的就是遥控器上的按键编码。测试程序如下:

   ```
   #!/usr/bin/python3

   f = open('/dev/input/event0', 'rb')

   while True:
       d = f.read(48)
       key = d.hex()[40:48]
       print (key)
   ```

6.4 直流电机

直流电机控制电路由两个功率输出端和一个控制开关组成。当控制开关接通时, 输出电流方向由两个输入端的电压差决定, 从而实现电机的双向转动控制。调整开关通、断时间比例即可调整电机转速。

清单6.3是电机控制示例。GPIO12、GPIO13 接输入端 A、B, GPIO6 接控制端。控制端采用 PWM 方式调整转速。



清单 6.3 motor.py

```
1  #!/usr/bin/python3
2  # -*- coding:utf-8 -*-
3
4  import RPi.GPIO as GPIO
5  import time
6
7  A = 13
8  B = 12
9  CONTROL = 6
10
11 GPIO.setmode(GPIO.BCM)
12 GPIO.setup((A, B, CONTROL), GPIO.OUT)
13
14 speed = GPIO.PWM(CONTROL, 1000)
15 # forward, 90% speed
16 GPIO.output((A, B), (1, 0))
17 speed.start(90)
18 time.sleep(1)
19 speed.stop()
20
21 # backward, 20% speed
22 GPIO.output((A, B), (0, 1))
23 speed.ChangeDutyCycle(20)
24 time.sleep(1)
25 speed.stop()
26
27 GPIO.cleanup()
```

6.5 超声波测距

超声波测距模块原理见图6.3。工作时, 要求在 Trig 端口产生不短于 $10\mu s$ 的正脉冲, 模块会自动发出 8 个周期的超声脉冲信号 (40kHz), 并在 Echo 端输出高电平。当检测到回波时将 Echo 回置到低电平。通过测量 Echo 高电平维持周期, 再根据声波速度, 就可以算出发射器到障碍物之间的距离。

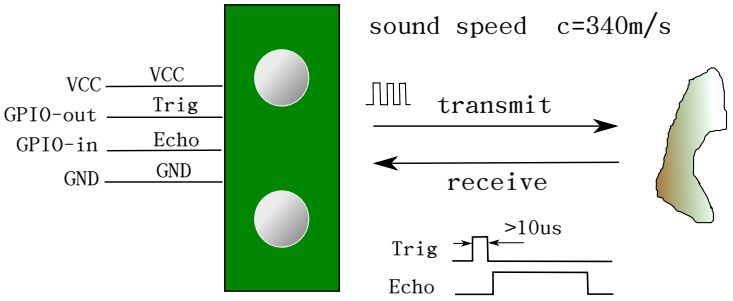


图 6.3 超声测距模块

清单6.4是超声测距模块子程序。

清单 6.4 ranging.py

```
def ranging():
    GPIO.output(TRIG,GPIO.HIGH)
    time.sleep(0.000015)
    GPIO.output(TRIG,GPIO.LOW)

    while GPIO.input(ECHO) == GPIO.LOW:
        pass
    t1 = time.time()
    while GPIO.input(ECHO) == GPIO.HIGH:
        pass
    t2 = time.time()
    return (t2 - t1)*34000/2        # distance in cm
```

6.6 红外传感器

红外传感器由一个红外发射管和一个红外接收管组成。作为循迹应用时, 发射管和接收管之间光隔离。发射管发出的红外光被反射后由接收管接收, 再经过 模数转换器得到量化的红外强度值。接收强度与传播距离和反射面对红外光的吸收性质有关。

TLC1543 是一个 10 通道、10bits 的 SPI 接口模数转换器, 清单6.5用软件仿真 SPI (**S**erial **P**ort **I**nterface) 协议：在 IOCLK 产生时钟脉冲, ADDR 逐位输入通道编码, 从 DOUT 读出转换值。

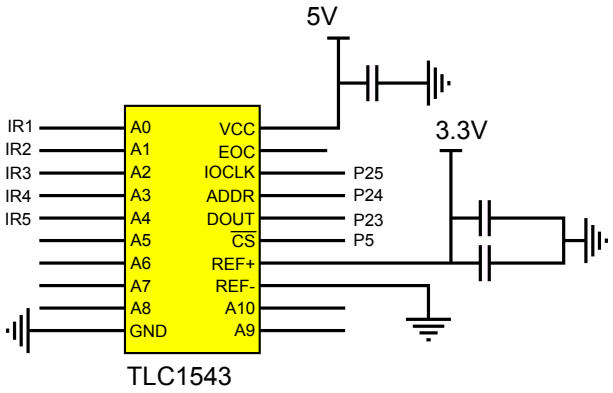


图 6.4 模数转换器 TLC1543

清单 6.5 irsensors.py

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

import RPi.GPIO as GPIO
import time

CS = 5
CLOCK = 25
ADDRESS = 24
DATAOUT = 23

GPIO.setmode(GPIO.BCM)
GPIO.setup((CS, CLOCK, ADDRESS), GPIO.OUT)
GPIO.setup(DATAOUT, GPIO.IN, GPIO.PUD_UP)

def AnalogRead():
    value = [0]*(6)
    #Read Channel0~channel5 AD value
    for j in range(0, 6):
        GPIO.output(CS, GPIO.LOW)
        for i in range(0, 10):
            #sent 4-bit Address
            if (i < 4):
                bit = (((j) >> (3 - i)) & 0x01)
                GPIO.output(ADDRESS, bit)

```

```
27              #read 10-bit data
28              value[j] <<= 1
29              value[j] |= GPIO.input(DATAOUT)
30              GPIO.output(CLOCK, GPIO.HIGH)
31              GPIO.output(CLOCK, GPIO.LOW)
32
33          GPIO.output(CS, GPIO.HIGH)
34          time.sleep(0.0001)
35      return value[1:]           # invalid address for channel 0
36
37  while True:
38      print (AnalogRead())
39      time.sleep(1)
```

避障传感器与循迹原理类似, 只是接收管输出的是数字量，可通过 RPi.GPIO 基本输入模块直接读取 0/1 状态。

6.7 彩灯

每一个彩灯灯珠是一个三色 LED，通过红、绿、蓝三个发光二极管产生的不同亮度形成不同的颜色。每个发光二极管由一个 8 位的数字信号控制，实现 256 级亮度。数字信号以串行方式输入，多个灯珠串接，形成一个灯带。控制灯带的数字信号对时钟要求比较严格，必须通过硬件实现。`rpi_ws281x` 模块提供了 python 接口。

安装 `rpi_ws281x` 模块:

```
# git clone https://github.com/yfang1644/rpi_ws281x
# cd rpi_ws281x/python
# python3 setup.py build
# python3 setup.py install
```

该模块同时提供 C 语言库和 Python 库。模块可以使用 PWM、PCM 和 SPI，但只有 PWM 可以同时独立控制两条灯带。他们使用的引脚见表6.2。GPIO 引脚编号超过 27 的在树莓派板上没有被引出。python/examples 目录中有一些例子。

模块使用方法：

```
from neopixel import *

# LED strip configuration:
LED_COUNT      = 16      # Number of LED pixels
LED_PIN        = 18      # GPIO pin connected to the pixels (18 uses PWM)
LED_FREQ_HZ    = 800000  # LED signal frequency in Hertz (usually 800kHz)
```

表 6.2 rpi_ws281x 模块支持的 GPIO 引脚

Function	GPIO
PWM 0	12、18、40、52
PWM 1	13、19、41、53、45
PCM	21、31
SPI	10、38

```
LED_DMA        = 10      # DMA channel to use for generating signal
LED_BRIGHTNESS = 255     # Set to 0 for darkest and 255 for brightest
LED_INVERT     = False   # NPN transistor needs invert
LED_CHANNEL    = 0       # set to '1' for PWM 1
LED_STRIP      = ws.WS2812_STRIP

# create NeoPixel object
strip = Adafruit_NeoPixel(
                    LED_COUNT,
                    LED_PIN,
                    LED_FREQ_HZ,
                    LED_DMA,
                    LED_INVERT,
                    LED_BRIGHTNESS,
                    LED_CHANNEL
                    )
# Intialize the library (must be called once before other functions).
strip.begin()

# red, green and blue each 8 bits
red, green, blue = 200, 100, 150
color = (red << 16)|(green << 8)|blue
for x in range(4):
    strip.setPixelColor(x, color)

# Flash data
strip.show()

time.sleep(2)
strip.setBrightness(128)
```

```
strip.show()
```

6.8 舵机

AlphaBot2 通过 PCA9685 芯片产生 PWM脉冲, 控制舵机的旋转角度。PCA9685 是一个 16 通道的 PWM 发生器, 采用 I2C 接口进行控制。PCA9685 模块由 Adafruit_Python_PureIO、Adafruit_Python_GPIO 和 Adafruit_Python_PCA9685 三个模块堆叠而成。它们的原始代码在 https://github.com/adafruit。为适应 Python 3 的编译，对其中的 setup.py 略作修改, fork 版本在 https://github.com/yfang1644。安装方法与 RPi.GPIO 类似。

```
# git clone https://github.com/yfang1644/Adafruit_Python_PureIO
# cd Adafruit_Python_PureIO
# python3 setup.py install
```

按同样的方法安装 Adafruit_GPIO 和 Adafruit_PCA9685。

此外还需要手工加载 I2C 驱动模块：

```
# modprobe i2c-bcm2835
# modprobe i2c-dev
```

这两条命令可以写入启动脚本，上电后自动执行。

以上工作完成后，就可以着手准备舵机的程序了。原理很简单：舵机的旋转角度由 PWM 的脉冲宽度决定。在 AlphaBot2 板上，PCA9685 的 0 通道和 1 通道可以用来控制可旋转摄像头的两个自由度，用于摄取特定角度的图像。

下面是一个简单的示例代码：

清单 6.6 servotest.py

```
1  import Adafruit_PCA9685
2
3  # Initialize the PCA9685 with default address (0x40).
4  pwm = Adafruit_PCA9685.PCA9685()
5
6  # Alternatively specify a different address and/or bus:
7  #pwm = Adafruit_PCA9685.PCA9685(address=0x41, busnum=2)
8
9  '''
10 Calculate PWM width for a specified angle.
11
12 angle in [0, 180] convert pulse width [0.5ms, 2.5ms]
13
14 PCA9685 has 12bits data.
```

```
When PWM frequency is set to 50Hz(20ms),
    value = (2**12)*(angle*11 + 500)/20000

PCA9685 has 16 output channels.
'''
def set_servo_angle(channel, angle):
    value = 4095*(angle*11 + 500)/20000
    pwm.set_pwm(channel, 0, int(value))

# Set frequency to 50hz, good for servos.
pwm.set_pwm_freq(50)

for angle in range(45, 135):
    # Move servo on channel O
    set_servo_angle(0, angle)
    time.sleep(0.5)
```

6.9 本章小结

本章介绍了 AlphaBot2 板上各个控制部件的基本原理和使用方法。

7

Python 应用模块

7.1 对象

Python 是面向对象的编程语言。对象集合了一组用于访问特定数据的方法。相比于面向过程的方法, 对象有下面几个明显的优势:

1. 对象具有封装性, 可以隐蔽一些不重要的细节, 为使用者提供方便统一的接口;
2. 由于对象是建立在函数级之上的封装, 便于构造并行算法结构;
3. 对象可以继承, 对于可扩展性功能, 省去了重复构造代码的繁琐;
4. 对象有助于构造多态性, 例如, 不同的对象可以使用相同的运算符号, 这使得程序设计更加简单, 也更便于维护。

Python 的所有变量都是对象。对象是通过定义类的结构实现的。

7.1.1 类的结构

Python 使用关键字 `class`定义一个类。与函数不同的是, 函数内部通常只涉及变量操作, 虽然在函数内部也可以定义函数接口, 但只能是局部的, 而且结构会比较复杂, 一般不建议采用; 而对于类来说, 变量和函数都是它的基本成员, 通常我们将类中的变量和函数称为 “属性” 和 “方法”。

类是抽象化了的对象, 对一个类的引用便成为一个实例(instance)。例如, 我们可以依据动物的特性构造一个`Animal` 类:

清单 7.1 animal.py

```
1 class Animal(object):
2     'A simple class'
3
4     legs = 2
5     def move(self):
6         print('I can move.')
```

```
7
8       def has_legs(self, legs):
9           self.legs = legs
10
11  Animal().has_legs(2)
12  cat = Animal()
13  cat.move()
```

对象支持两种类型操作: 属性引用和实例化。形如 obj.name 属于属性引用, 它不经创建类实例而直接访问类成员；而上例最后两行称为实例化, 它先创建一个类的实例, 然后通过实例访问类的方法和属性。

针对上述代码, 有几点说明:

1. Animal 由基类object派生。object 是 Python 内建的基类, 它基本上什么事都不做。派生于 object 的类, 基类名可以省略, 甚至括号也可以省略。上面例子中第一行可以写成:

   ```
   class Animal():
   ```

   或者

   ```
   class Animal:
   ```

2. 对象的初始化方法函数是__init__(), 如果存在, 则类实例化时将首先调用该函数, 每次属性引用时也都会调用该函数。__init__() 函数名称是确定的, 形式参数是可变的, 取决于类的构造需要。
3. 类方法中的第一个形式参数 self 是类的引用, 它只用于类的内部访问, 不作为形式参数传递。上面例子中, 在类的内部, 访问属性legs 时使用 self.legs, 访问函数 has_legs() 使用self.has_legs(legs)。
4. 方法函数的第一个参数self 仅供内部访问, 外部引用时不占参数位置。self 也不是 Python 的关键字, 只是一个习惯上被大多数 Python 程序员使用的名词。

以上类的方法被称为 “实例方法”。类方法还有两个修饰词 @staticmethod 和 @classmethod, 分别称为 “静态方法” 和 “类方法”。

7.1.2 类的继承

在 Animal 类的基础上派生一个新的类, 将继承Animal 已有的特性, 同时扩展出新的特性:

```
class Bird(Animal):

    wings = 0
    def __init__(self, wings):
        self.wings = wings
```

```
    def fly(self):
        print('I have %d wings, I can fly.' % self.wings)

    def move(self):
        print('I can move with my legs.')

    def getName(self):
        return self.name

parrot = Bird(2)
parrot.has_legs(2)
parrot.move()
parrot.fly()
```

一个类可以从多个类继承 (多重继承), 这种情况下, 子类继承每个父类的特性。

通常对象的方法是供外部访问的。默认地, 对象的属性也可以通过外部访问, 即使在属性不存在的情况下, 也可以从外部创建。虽然不像对方法调用那么常见, 甚至也不安全, 但确实是 Python 类的一个特性:

```
parrot.name = 'talky bird'
print(parrot.getName())
```

这种访问方式破坏了类的封装性, 多数情况下不建议采用。对对象属性的访问通常应通过对象的方法来实现。例如在上面的`Bird` 类中, 使用一个诸如 `setname()` 的方法设置 `name` 属性。

7.1.3 类的继承关系

下面是一些判断类之间关系的有用的函数:

- 类 `Bird` 继承自 `Animal`, 我们称 `Animal` 为 基类(superclass), 称 `Bird` 为子类 (subclass), 函数 `issubclass()` 可以得到二者的关系:

  ```
  >>> issubclass(Bird, Animal)
  True
  >>> issubclass(Animal, Bird)
  False
  ```

- 所有类都有一些缺省的属性, 其中一个属性 `__class__` 可以提供给对象判断它来自哪个类。例如:

  ```
  >>> print (cat.__class__)
  <class '__main__.Animal'>
  >>> print (parrot.__class__)
  <class '__main__.Bird'>
  ```

- 属性__bases__ 提供基类的判断 (复数表示一个对象的基类可以不止一个)。以上例子中, Animal 的基类是 object, Bird 的基类是 Animal;
- 要想知道一个对象是否具备某种属性, 通常可使用内建函数 hasattr():

```
>>> hasattr(parrot, 'move')
True
```

函数 hasattr() 不区分方法和属性, 也就是说, wings 和 move 都是对象 parrot 的属性。但有时候我们想知道哪个是可调用的函数, 可以通过下面的方法:

```
>>> callable(getattr(parrot, 'move'))
True
>>> callable(getattr(parrot, 'wings'))
False
```

7.2 图形界面

7.2.1 Tkinter

Python 的出发点不是图形化, 因此没有内建的图形用户接口函数。但图形化又是如此重要, 以至于很多开发人员为 Python 提供了不同的 GUI (**G**raphical **U**ser **I**nterface, 图形用户接口) 工具包, 其中比较著名的有 PyGTK、PyQT、Tkinter 等等。

表 7.1 几种 Python 图形包

模块	说明	特点
Tkinter	基于 Tcl/Tk 库	Python 的半标准化图形库
wxPython	基于 wxWindows 系统	跨平台
PythonWin	使用 Windows GUI	仅用于 Windows 操作系统
PyGTK	基于 GTK 库	主要见于 Linux 系统
PyQt	基于 Qt 库	跨平台

其中 tkinter 在编译 Python 的时候就可以获得, 很多 Python 发布版默认就已经包含了这个模块, 无需额外安装其他软件包, 依赖关系简单, 使用较方便。该模块在 Python 3 中名称是 tkinter, Python 2 中的名称是 Tkinter。Python 自带的图形化集成开发环境idle 使用的就是 tkinter。

7.2.2 Tkinter 的使用

一个使用Tkinter 的 (最小) 图形化界面如下:

清单 7.2 button.py

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  import tkinter as tkinter
5
6  def func():
7      print('Button Pressed')
8
9  win = tkinter.Tk()
10 win.geometry('200x100+300+400')
11 win.title('tkinter')
12 hello = tkinter.Button(win,
13                         text='Hello',
14                         command=func)
15 hello.pack(expand=True, padx=20, pady=10)
16 win.mainloop()
```

下面是逐行解释:

- 第 4 行, 导入Tkinter 模块;
- 第 6–7 行, 定义一个函数, 供下面调用;
- 第 9 行, 创建顶层窗口;
- 第 10 行, 设置窗口大小和位置。该项设置在多数情况下不是必须的, 窗口会根据内部的组件自行调整大小;
- 第 11 行, 设置窗口标题;
- 第 12–14 行, 创建一个按钮组件, 其容器是win 对象, 按钮上的文字使用参数text 设置, 按钮动作 (响应函数) 通过 command 设置;
- 第 15 行, 将组件 hello 加入容器, 并设置与容器的衔接关系。
- 第 16 行, 所有顶层窗口显示, 进入消息循环, 此后各组件并行工作, 接收消息, 做出反应。由于在这个例子中只创建了一个顶层窗口 win, 因此只显示这一个窗口。

上面的例子运行时, 将在屏幕坐标 (300, 400) 处显示 200×100 大小的窗口, 窗口内有一个按钮, 鼠标点击按钮时会在终端打印一串文本。

由此可以归纳 Tkinter 图形化编程步骤 (使用其他图形化模块的步骤大同小异):

1. 导入Tkinter 模块;
2. 创建顶层窗口;
3. 创建 GUI组件, 设计其样式、响应函数;

4. 将组件组装进窗口；
5. 调用`mainloop()`, 进入主循环。

同其他 GUI 程序一样, GUI 库只负责窗口内的组件, 窗口样式则由窗口管理器负责。换句话说, 在不同的系统中, 同一个程序的标题栏、边框、窗口控制按钮等表现形式可以是不同的。

7.2.3 常用组件

表7.2列出了比较常用的 Tkinter 组件。

表 7.2 Tkinter 常用组件

组件名称	说明
Label	标签文字或图像, 一般不响应消息
Message	消息框, 与 Label 功能类似, 用于多行文本显示
Button	按钮, 与标签类似, 但通常需要响应鼠标及键盘消息事件
Checkbutton	复选按钮, 使用两个不同状态反映一项功能有/无
Radiobutton	单选按钮, 由一组互斥的按钮组成, 其中有且只有一个被选中
Entry	单行文本输入框, 提供一个可编辑的文本输入环境
Text	多行文本输入框
Scale	标尺, 通过滑块设置/获取数值, 有水平方向和垂直方向两种
Scrollbar	滚动条
Canvas	画布, 提供画图环境
Frame	框架, 本身不可见, 通常作为其他组件的容器
Menu	菜单

下面是一个组合了画布、标尺、按钮、文本框的示例程序。程序界面如图7.1。

清单 7.3 canvas.py

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 from tkinter import *
5
6 # change height of the rectangle
```

```
7  def resizev(v):
8      height = int(v)
9      width = scaleh.get()
10     canvas.coords(box, 150 - width/2,
11                        150 - height/2,
12                        150 + width/2,
13                        150 + height/2)
14     if filled.get():
15         canvas.itemconfig(box, fill=rgbv.get())
16     else:
17         canvas.itemconfig(box, fill='white')
18
19 # change width of the rectangle
20 def resizeh(v):
21     width = int(v)
22     height = scalev.get()
23     canvas.coords(box, 150 - width/2,
24                        150 - height/2,
25                        150 + width/2,
26                        150 + height/2)
27     if filled.get():
28         canvas.itemconfig(box, fill=rgbv.get())
29     else:
30         canvas.itemconfig(box, fill='white')
31
32 # set color by RGB
33 def setcolor():
34     if color.get() == 0:
35         rgbv.set('#ff0000')
36     elif color.get() == 1:
37         rgbv.set('#00ff00')
38     elif color.get() == 2:
39         rgbv.set('#0000ff')
40
41 win = Tk()
42 win.title('ColorBox')
43
44 # create 2 frames as layout
```

```
45 frame1 = Frame(win).pack(side=RIGHT)
46 frame2 = Frame(win).pack()
47
48 # create a rectangle
49 canvas = Canvas(frame1, width=300, height=300,bg='white')
50 box = canvas.create_rectangle(150, 150, 150, 150,
51                               width=2,
52                               outline='red',
53                               fill='white')
54
55 # horizontal scale
56 scaleh = Scale(frame1, length=300,
57                        from_=0,
58                        to=300,
59                        orient=HORIZONTAL,
60                        command=resizeh)
61 # vertical scale
62 scalev = Scale(frame1, length=300,
63                        from_=300,
64                        to=0,
65                        orient=VERTICAL,
66                        command=resizev)
67
68 filled = IntVar()
69 # set if the box is filled with color or not
70 fill = Checkbutton(frame2, text='fill',
71                            onvalue=1,
72                            offvalue=0,
73                            variable=filled)
74
75 # set fill color in RGB
76 color = IntVar()
77 color.set(0)
78 red   = Radiobutton(frame2, text='Red',
79                             variable=color,
80                             value=0,
81                             command=setcolor)
82 green = Radiobutton(frame2, text='Green',
```

```
                              variable=color,
                              value=1,
                              command=setcolor)
blue  = Radiobutton(frame2, text='Blue',
                              variable=color,
                              value=2,
                              command=setcolor)

rgbv = StringVar()
rgbv.set('#ff0000')
rgb = Entry(frame2, textvariable=rgbv)

scalev.pack(side=LEFT)
canvas.pack()
scaleh.pack()
fill.pack(side=TOP)
rgb.pack(side=TOP)

red.pack(side=LEFT)
green.pack(side=LEFT)
blue.pack(side=LEFT)

win.mainloop()
```

7.3 多任务方式

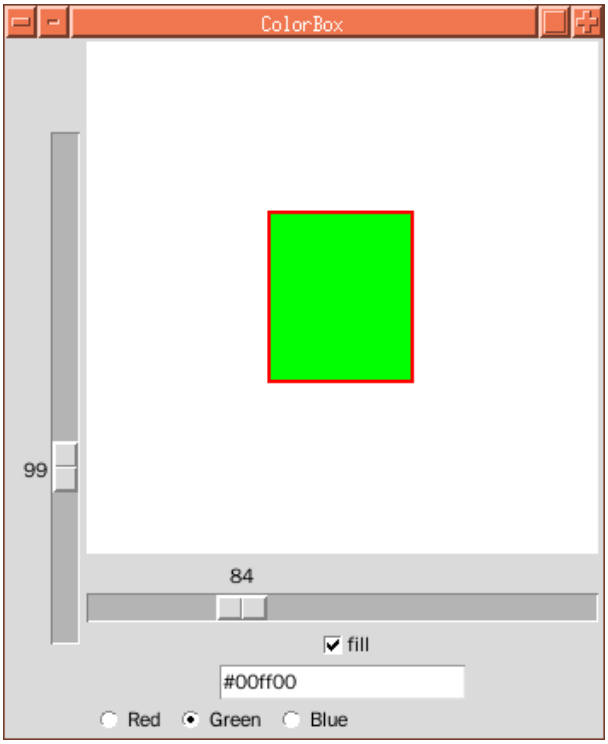
操作系统实现多任务的方式有线程和进程两种形式, 这里仅讨论线程形式。多个线程与主程序共享相同的数据空间, 它比进程交换数据容易得多, 而且比进程占用的资源更少。线程常常又被叫做轻量级进程。

Python 有两个常用的线程模块: 原始的`_thread` (Python 2 是 thread) 和高级的`threading`。

`_thread` 模块的 `start_new_thread(func, args[, kwargs])` 创建一个新的线程, 线程入口函数是 `func`, 向线程传递元组形式的参数 `args`, 或辞典关键字 `kwargs`, 返回线程`ID`。线程函数返回时即告结束, 不需要返回值。如果主程序先于线程函数结束, 则线程函数也自动结束。

```
import _thread as thread
import time

def newthread(msg, delay):
```

图 7.1 tkinter 组件的组合使用




```
    while True:
        print(msg)
        time.sleep(delay)

# create 2 threads, arguments in tuple
thread.start_new_thread(newthread, ("Faster", 1,))
thread.start_new_thread(newthread, ("Slower", 4,))

# main function wait 20 seconds
time.sleep(20)
```

threading 模块具有对线程管理更灵活的手段。threading 的 Thread 类提供了下面一些方法:

- run() — 线程入口
- start() — 启动线程运行
- join ([time]) — 等待线程结束
- isAlive() — 检查线程是否在运行
- getName() — 返回线程名
- setName() — 设置线程名

使用 threading 模块, 一般采用如下步骤:

1. 定义子类, 继承 Thread 类;
2. 重写__init__() 方法, 添加自己的参数;
3. 重写run(), 实现自己的线程入口;
4. 用start() 启动线程运行。

```
#inherit threading and rewrite method run()
class MyThread(threading.Thread):
    def __init__(self, arg):
        super(MyThread, self).__init__()
        self.msg = arg[0]
        self.delay = arg[1]
    def run(self):
        while True:
            print (self.msg)
            time.sleep(self.delay)

t1 = MyThread(('Faster', 1))
t2 = MyThread(('Slower', 4))
t1.start()
t2.start()
time.sleep(20)
```

7.4 网络套接字

在 TCP/IP 网络通信中, 套接字代表了一个网络接口端点, 它由 IP 地址和端口号组成。套接字的性质与文件描述符相同, 从而对网络的操作就变成了对文件的操作。IP (**I**nternet **P**rotocol) 地址是一个 32 位的整数[1], 通常将每个字节写成一个十进制, 中间用 "." 分隔, 这种表示方法称为 "点分十进制表示法", 如 202.119.32.7。另外还有一种字母形式的表示方法, 如 "www.nju.edu.cn", 这种方法便于记忆。字母形式和点分十进制形式有明确的对应关系, 网络中专门有一个服务器存放这张关系表, 这个服务器被叫做域名系统 (DNS, **D**omain **N**ame **S**ystem)。

每台计算机可能会提供多种网络服务, 不同服务之间使用端口号加以区分。例如前面用到的 SSH 是 22 端口、VNC 是 5900 端口、HTTP 是 80 端口, 等等。服务器的端口号是确定的, 并且要让客户端知道, 这样, 客户端才能有一个明确的访问目标。客户端向服务器发起连接请求时, 操作系统核心会自动分配一个端口, 客户端的端口号是不确定的。

在客户/服务器模型中 (C/S, **C**lient/**S**erver) 又有两种协议形式: 建立连接的 TCP (**T**ransfer **C**ontrol **P**rotocol) 协议和 UDP (**U**ser **D**atagram **P**rotocol) 协议, 前者是建立连接的、

[1] 此处仅讨论第 4 版 IP 协议。

可靠的协议, 发送方发送的数据要确保接收方正确接收, 后者是无连接的、不可靠的协议。他们的套接字类型分别是SOCK_STREAM 和SOCK_DGRAM。

7.4.1 TCP 协议通信

提供 TCP/IP 网络通信的模块是 socket。无论是客户端还是服务器, 都需要通过函数 socket() 创建套接字。创建 TCP 套接字的方法是:

```
import socket

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

服务器需要通过 bind() 函数绑定 IP 地址和端口。多数情况下, IP 地址无需绑定, 这样, 程序可以适用不同的 IP 地址, 如果一台主机有多个网络设备也不在意具体使用哪一个。

TCP 协议还要使用 listen() 设置backlog。backlog 是当系统来不及响应所有连接请求时, 等待连接的最大数目。

接下来, 服务器阻塞于accept() 等待客户端连接, 客户端使用 connect() 向服务器发起连接请求。成功建立连接后, accept() 返回连接描述符和客户端地址, 服务器使用连接描述符与客户端对话。

清单7.4和7.5是 TCP 协议的服务器、客户端示例,

清单 7.4 TCP 服务器端 server.py

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  import socket
5
6  # create IPv4, TCP socket
7  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8
9  # server needs binding socket to a port (assume 9999)
10 s.bind(('', 9999,))           # server doesn't need to specify host
11 s.listen(5)                   # set back log
12 while True:
13     c, addr = s.accept()      # wait a client to connect
14     print ('connection request from ', addr)
15     c.send('hello, client!')
16     c.close()                 # close this connection
```

清单 7.5 TCP 客户端 client.py

```
1  #!/usr/bin/python3
```

```
2  # -*- coding: utf-8 -*-
3
4  import socket
5
6  # create IPv4, TCP socket
7  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8
9  s.connect(('192.168.88.1', 9999))
10 msg = s.recv(1024)        # read 1024 byte from socket
11 print(msg)
12 s.close()
```

7.4.2 UDP 协议通信

清单7.6和7.7是 UDP 协议的服务器、客户端示例。UDP 不建立连接, 服务器可以接收来自任何主机的信息。

清单 7.6 UDP 服务器端 server.py

```
1  import socket
2  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3  s.bind(('', 9999,))
4
5  while True:
6      print(s.recv(1024))
```

清单 7.7 UDP 客户端 client.py

```
1  import socket
2  s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
3  while True:
4      msg = input('Input message:')
5      if msg == 'exit':
6          break
7      s.sendto(str(msg).encode(), ('192.168.88.1', 9999))
8
9  s.close()
```

7.5 本章小结

Python 是面向对象的语言。对象具有封装性和多态性。通过对象的继承, 可以很方便地扩展功能。

许多程序员为 Python 开发了大量的实用模块, 丰富了 Python 的功能。本章介绍了一些较常用的模块: 用于构建图形界面的 Tkinter, 用于实现多任务工作的线程模块 threading, 网络通信中的套接字模块 socket。

索引