AN13242

在 i.MX RT1050/1060 上使用 KEIL 构建 Micropython 系统并体验 Python 开发

Rev. 0 — 27 April, 2021 Application Note

目录

1 简介

这篇应用笔记以 i.MX RT1060evk 开发板为例，介绍了对 Micropython 的移植、对外设功能的封装，以及对电路板的适配。它们用 C 语言编写，但是以 Python 模块和类型的方式呈现给用户。读者既可以在这个开发板上评估和使用 Micropython，也可以在此基础之上，对自己新设计的电路板进行移植和适配。Micropython 原生的项目管理和构建环境是在 Linux 下基于 GCC 和 Make 的，本文为了方便大多 MCU 嵌入式工程师的开发习惯，把开发环境移植到了 KEIL MDK5 上。

本文假定读者有使用 KEIL MDK 开发的基本经验，比如了解 CMSIS-Pack，了解 KEIL MDK 工程中的 Target 概念以及切换方法等。

i.MX RT1050/60 是一个基于 arm Cortex-M7 核心的处理器，运行速度可达 600 mhz。强大的处理能力、实时特性以及丰富的外围设备，使 i.MX RT1050/60 成为众多高性能应用的理想选择，如工业计算、电机控制、电能转换、智能消费产品、高端音频系统、家庭和楼宇自动化。

2 硬件平台

2.1 i.MX RT1050/60 跨界处理器

NXP 公司提供的 i.MX RT1050/60 是基于 Arm® Cortex®-M7 内核的处理器，可以在高达 600 MHz 的速度下运行。它具有 512 KB 片上 RAM，可灵活配置为核心紧耦合内存(TCM)或通用 RAM; 此外，还有专门的 512KB OCRAM。

i.MX RT1050/60 提供各种接口用于连接各种外部存储器，以及各种串行通信接口，如 USB、以太网、SDIO、CAN、UART、I2C 和 SPI。它还具有丰富的音频和视频功能，包括 LCD 显示、基本的 2D 图形、摄像头接口、SPDIF 和 I2S 音频接口。其他值得注意的功能包括各种模块的安全，电机控制，模拟信号处理和电源管理。

2.2 i.MX RT1050/60 EVK 开发板

i.MX RT1050 EVKB/1060 EVKB 开发板设计用于展示 i.MX RT1050/60 处理器的常用功能，包含了以下的特征：

- 存储器: 256 Mbit SDRAM, 64 Mbit Quad SPI Flash, 512 Mbit Hyper Flash, TF 卡槽
- 通信接口: USB 2.0 OTG 连接器, USB 2.0 主机连接器, 10/100 Mbit/s 以太网连接器, CAN 总线连接器
- 多媒体接口：CMOS 传感器连接器，LCD 连接器
- 音频接口: 3.5 mm 立体声耳机插孔, 板载麦克风, SPDIF 连接器（默认未安装）
- 调试接口: 板载 DAP-Link 兼容调试器以及标准 JTAG 20 针连接器
- Arduino 扩展口
- 用户按钮与 LED

图 1 是 i.MX RT1050 EVKB 的照片.

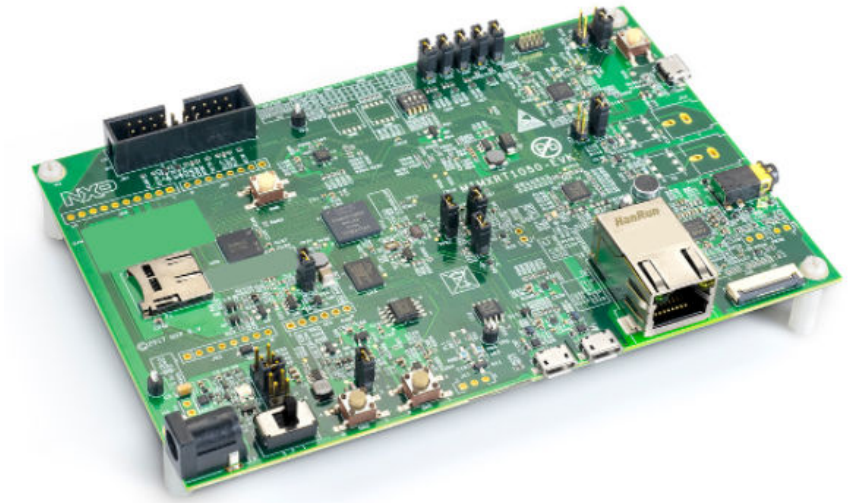

图 1. i.MX RT1050 EVKB

3 Micropython

3.1 Python 语言简介

Python 语言的学习对初学者非常友好，学习曲线几乎是从平坦开始，即使是没有计算机基础的青少年都可以快速入门。除此之外，Python 还有强大的表达能力，一行代码常常能顶得上多行 C 代码。 Python 还提供了常见的容器式数据结构以及相关的操作函数，这些数据结构有线性表，字典（也就是散列表），集合，数组等。Python 中的变量都是对象，对变量的使用基本都是通过地址引用。所以它们里面容纳的内容也都可以形象地认为是 void*，使得异构和嵌套得到了天然的支持，善于表达变化多端的动态数据结构。当然了，Python 运行环境知道它们的实际类型。在此基础上，Python 函数的参数与返回值都非常灵活，完全颠覆了 C 语言上面向机器的哲学，使得 Python API 看上去极为简洁、稀少，但非常易用而强大。

Python 的字符串、大整数运算支持也非常完整，尤其是字符串，和 C 语言相比简直就是石器时代到信息时代的飞跃。程序员之间常有“人生苦短，我用 Python”的俗语，就是强调 Python 语言让人能方便高效地编程，直奔主题，免于在处理计算机的底层问题上疲于奔命。Python 目前的主流版本是 3.x。

当然了，Python 的这些好处也不是免费的，它其实是把程序员的岁月静好建立在 CPU 的负重前行上。这使得在电脑上使用纯 Python 代码的运行效率不高，也几乎没有什么硬实时性可言。因此，常常是 Python 在提供 API 的同时，底层的工作使用效率更高的 C/C++等库来完成。

3.2 Micropython 简介

不要被 Micropython 中的“micro”所干扰。对于 MCU 来说，两个 micro 互相抵消后，它仍然是一个大型的软件项目，有数万行源代码。

Micropython 是 Python 3.6 的另一种实现，支持全部常用的 Python 语法。MicroPython 也是 Python 的一个精简版本，它是为了运行在单片机这样的性能有限的微控制器上，最小体积仅 256 K，运行时仅需 16 K 内存，但这样寒酸的配置基本上也只能运行最简单的脚本了。若要使用功能比较完整的配置，建议配置 512 KB Flash 和 64 KB RAM，或者更大。对于 i.MX RT 系列来说，资源已经远远超过了这个建议的配置。

注释

在 Python 这类动态语言中，一般把程序代码称呼为“脚本”，也是要以编程的方式来编写的。

Micropython 为了适应嵌入式微控制器，裁剪了大部分标准库，仅保留部分模块如 math、sys 的部分函数和类。此外，很多标准模块如 json、re 等在 MicroPython 中变成了以 u 开头的 ujson、ure，表示针对 MicroPython 开发的标准库。目前，MicroPython 除

了可以运行在最初开发的 pyboard 微控制器上外，还可以运行在大量基于 ARM 的嵌入式系统。本文就介绍了把 Micropython 移植到 NXP i.MX RT1050/1060 上，并且把开发环境由官方的 GCC+Make 改成了 KEIL MDK 5。

Micropython 中的“micro”容易让人以为它只是在 Python 上减少功能。其实不然，它也为了适合在 MCU 上使用而增加了新的功能和特点。比如，在 Micropython 上，对于性能和实时性有要求的部分可以仍然使用 C 来编程，并且导出绑定到 Python 的接口，一般可以做到易用和严肃开发的兼顾。Micropython 本身也通过一些特殊的扩展和模块来支持在 MCU 更高效和底层的操作，包括 native 修饰，以不使用动态内存换性能的 viper 修饰，可以直接访问地址空间的 mem8/16/32 模块，甚至直接内联汇编等。

Micropython 的官方网站是 http://www.micropython.org/。它在 github 上以友好的 MIT 协议开源了代码，仓库位于 https://github.com/micropython/micropython。

4 在自己的 i.MX RT1050/1060evk 上构建并运行 Micropython 固件

4.1 下载源代码

4.1.1 下载为本文特制的版本

在 micropython-rocky 上，下载 source code (zip)，并解压缩。请牢记 **an_mpy1050_rev1** 标签，这是一个与本文配套的版本，下文的介绍基于这个版本。

4.1.2 以 git clone 方式下载

如果您熟悉 git 的使用，可以打开一个命令窗口，导航到自己打算放入的目录中，然后执行以下命令：

```
git clone https://github.com/RockySong/micropython-rocky.git
```

这会默认克隆“omv_initial_integrate”分支。这个分支在 Micropython 的移植上还添加了 OpenMV 的功能。OpenMV 的话题暂不在本文讨论。**为了保证和本文所叙述的内容完全相同，建议 check out 到“an_mpy_rt1050_60”分支。**

注释

如果使用最新的代码遇到问题，最好切换到带 **an_mpy1050_rev1** 标记(tag)的版本。

4.2 打开 KEIL 工程并生成固件

基于 **an_mpy1050_rev1** 版本，打开\ports\prj_keil_rt1060\ mpyrt1060.uvprojx 目录，使用 KEIL MDK 5.0 及以上版本打开它。

注释

这个工程依赖 NXP.MIMXRT1062_DFP.12.1.0 或更高版本，这是 CMSIS pack 包。

这个工程里有多个目标配置（KEIL 称它们为“Targets”），如 图 2 所列。

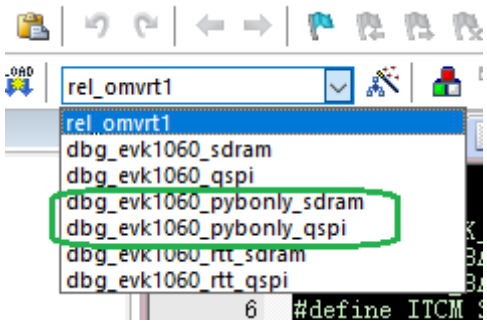

图 2. 目标配置

选择有“pybonly”关键字的，强调“只包含 micropython，不包含 openmv 主体功能”。笔者用的版本分别有在 SDRAM 中调试和在 QSPI Flash 中调试的两种。因为在 SDRAM 中调试最方便，建议一般先选择在 SDRAM 中调试。无论选哪一种，都可以点

击  或者按下“F7”来生成整个工程。

如果出现以下的报错，则表示没有指定的 AC5 编译器的版本。

```
Build Output
*** Target 'dbg_evk1060_pybonly_sdram' uses ARM-Compiler 'V5.06 update 6 (build 750)' which is not available.
*** Please review the installed ARM Compiler Versions:
   'Manage Project Items - Folders/Extensions' to manage ARM Compiler Versions.
   'Options for Target - Target' to select an ARM Compiler Version for the target.
*** Build aborted.
```

图 3. 报错信息

点击 

或按“Alt+F7”呼出工程配置对话框，按 图 4 切换到“Target”选择卡，再在 Code Generation 框的 ARM Compiler 下拉菜单里选择“Use default compiler 5”，确定后再重新编译。

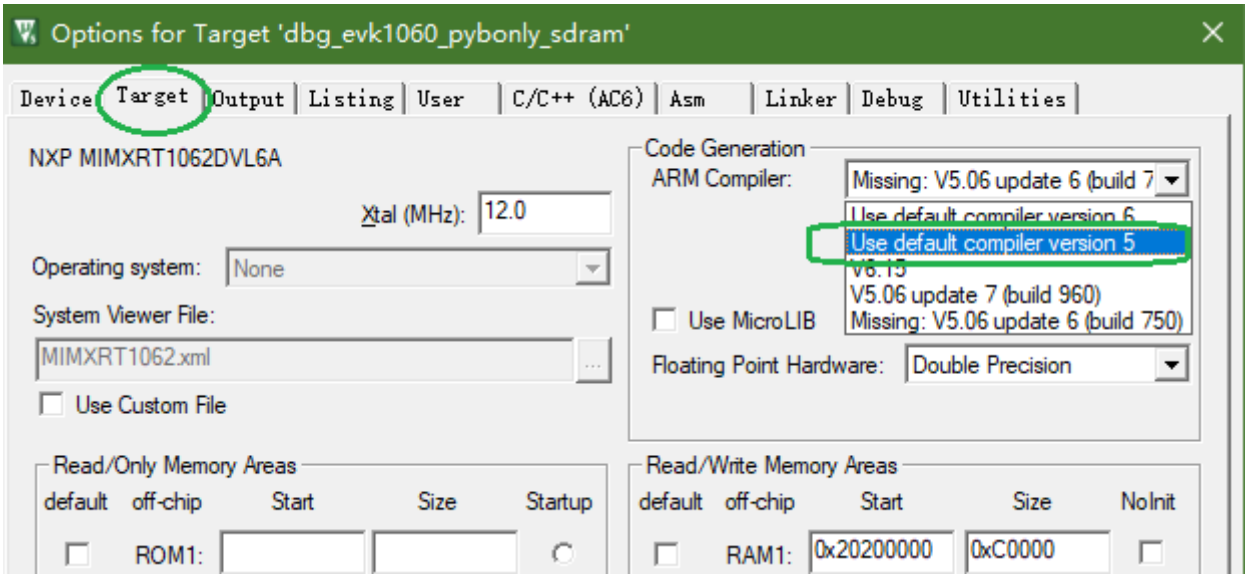

图 4. Options for Target

4.3 准备文件系统

Micropython 下，文件系统的概念极为重要，所有的 Python 脚本都是存储在文件系统中的。在我们移植的系统中，支持 TF/microSD 卡上的文件系统和程序 Flash 中的文件系统。在启动期间，系统先检测是否插入了 TF 卡，如果有就挂载 TF 卡到根文件系统下；如果没有，则挂载从 QSPI Flash 中分配的一块约 2 MB 的空间作为文件系统。如果是首次使用，则系统会自动格式化 QSPI Flash 中的这块区域。

Flash 文件系统使得在缺失 TF 卡的时候仍然可以运行 Micropython。然而，它的写入性能极差(约 10 KB/s 左右)，并且没有包含磨损平衡，一般建议只放一些不常改动的文件，比如配置文件、已调试好的脚本等。除此之外，在往 Flash 文件系统写入数据时，会关闭中断，影响实时性。因此，**强烈建议在板上插入一个使用 FAT/FAT32 格式化过的 TF 卡。**

访问文件系统的方法见下文介绍。

4.4 下载和运行

首先确定自己使用的调试器。1060 EVK 上自带了 CMSIS-DAP，但是由于本工程生成的固件比较大，下载慢一些。也可以使用 J-Link。在 1060EVK 上，若使用 J-Link，需要断开 J47 和 J48；若使用板载 CMSIS-DAP 兼容调试器则短接它们。在 KEIL 下，J-Link 的程序下载远比板载的 CMSIS-DAP 要快。

注释

如果使用 J-Link，那么在下载到 SDRAM 中后，在执行程序之前，最好再点击一下复位按钮，否则可能会意外进入 hard fault。

按 图 5 所示连接 1060 EVK 开发板。

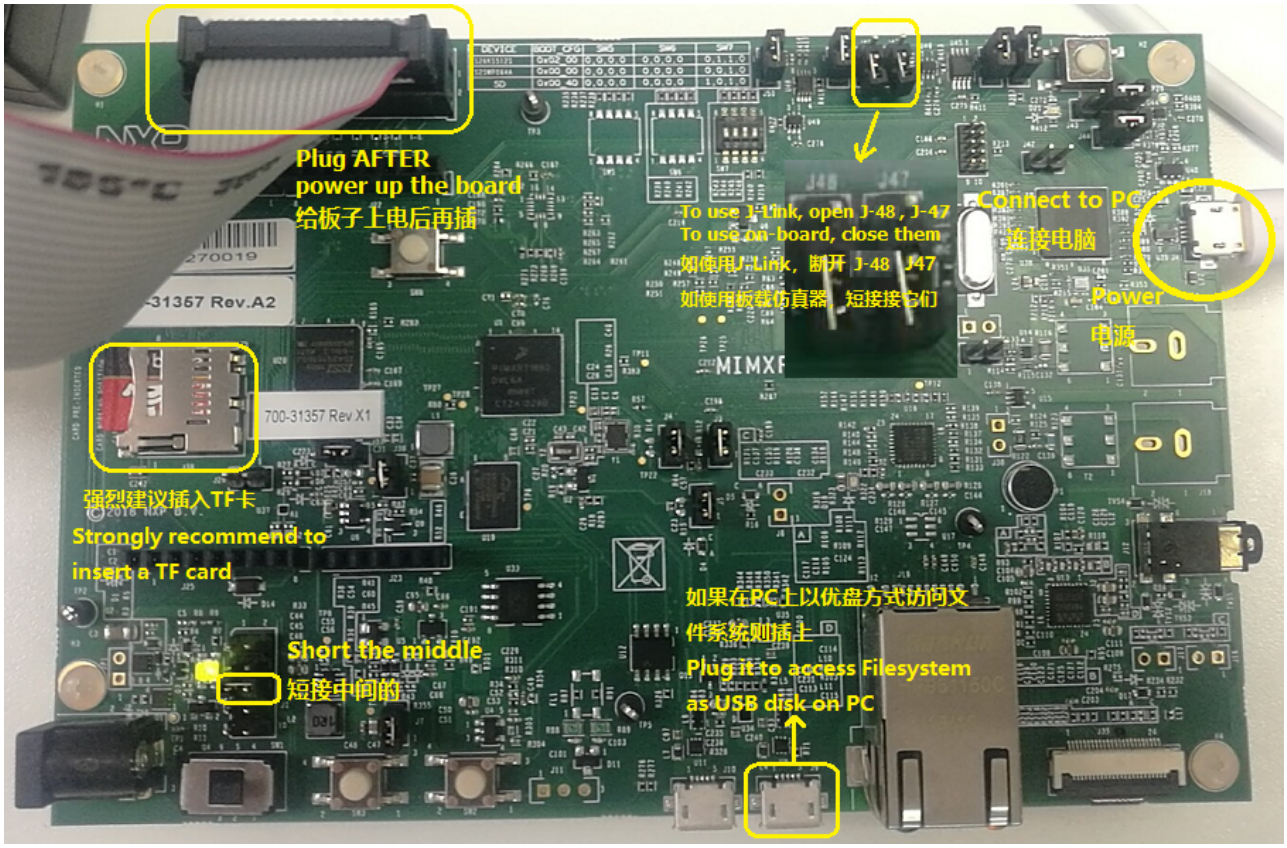

图 5. 连接 i.MX RT1060 EVK

打开一个串口终端，连接开发板的板载调试器所呈现的虚拟串口，波特率设置为 115200。

如果在 SDRAM 中调试，在生成完毕后点击 

或按“Ctrl-F5”，等程序下载完毕进入调试界面时，按“F5”全速运行。

切换到串口终端上，稍等片刻，可见少量启动消息，并且进入“>>>”提示符。可以试着输入“print(‘Hello micropython!”)，效果如下：

图 6. 启动期间输出的信息

至此，已经可以在这个交互式开发环境中使用 Micropython 了，这个交互式环境叫 REPL。在 PC 上的 Python 也可以按照相似的方式在命令行下使用，这也是使用 Python 编程的最简单的形式。

5 在 PC 上访问 Micropython 文件系统

通过 PC 访问 Micropython 文件系统是极为重要的，这是因为在 Micropython 上缺少好用 Python 编辑器。我们常常是在 PC 上使用自己喜欢的编辑器书写 Python 代码。写好的代码需要下载到 Micropython 的文件系统上。为了方便使用，我们在移植时实现了 USB 大容量存储设备类，以一块优盘的形式呈现文件系统。

如果希望在 PC 上访问文件系统的内容，请在靠近以太网口的 USB 口上连接到电脑，这会在电脑上呈现出一个可移动磁盘。当没有插入 TF 卡时，可移动磁盘中显示的是 Flash 文件系统中的内容；当插入了 TF 卡时，可移动磁盘里显示的是 TF 卡中的内容。

当访问的是 Flash 文件系统时，写入速度只有 10 KB/s 左右。当访问的是 TF 卡时，一般读写速度均在 10MB/s 左右。

注释

不要在 TF 卡中包含名为“flash”（区分大小写）的目录，这会导致在 Python 脚本访问“/flash”时仍然访问 Flash 文件系统，而在 PC 上看到的却是 TF 卡中的内容。

6 体验 Python 的开发

从上一节的打出“Hello micropython”开始，我们已经初步在 REPL 上体验了 Micropython 的编程。与在 PC 上的体验相似，这种命令式的方式非常利于快速入门，或者试验一个新功能，甚至是一小段程序。其实，在 Micropython 中，可以有多种开发、部署、执行 Python 脚本的方式。图 7 总结了三种方式：

- 从文件加载脚本
- 从交互式终端上实时执行
- 交叉编译，固化执行

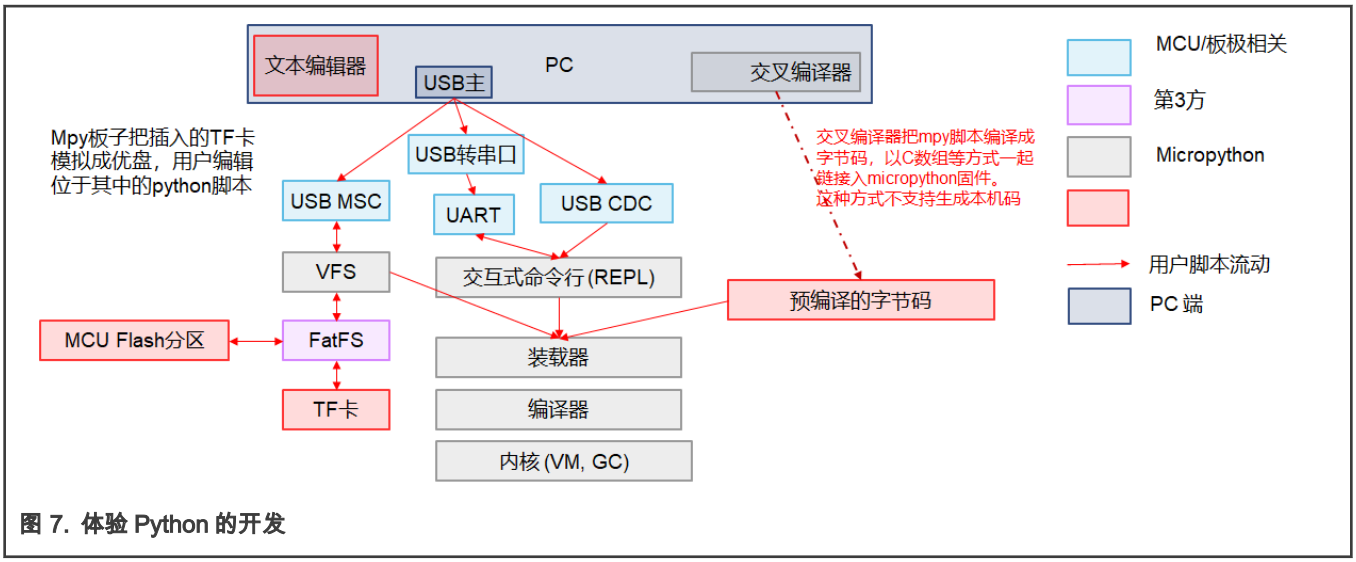

图 7. 体验 Python 的开发

其中，左边的一条路是从文件系统中加载脚本，并由 Micropython 的装载器和编译器生成字节码，提交给核心的虚拟机。文件系统可位于 TF 卡或程序 Flash 中。

再看中间的两条路，尽管可以分别使用 MCU 的 USB 虚拟串口或 MCU 的 UART 通过板载调试器的虚拟串口，都是要通过 REPL 组件来获取 Python 程序行或片段，剩下的步骤就与从文件系统中执行的相似了。

从文件系统和从 REPL 执行是本文接下来要重点介绍的部分。

这里也简单介绍一下右边的那条路，这和 C 语言项目开发习惯比较接近。需要我们在宿主机上使用 Micropython 提供的交叉编译工具，一个名叫 mpy-cross 的程序，来把 Python 代码编译成 Micropython 的虚拟机可以解的字节码。字节码被封装成一个特殊的可执行对象，并序列化为 C 源代码，放置到 Micropython 的工程中，随其它源码一起编译和链接。

使用这种方式可以从固件中剪裁掉 Micropython 编译器，节省少量 Flash 空间。但是由于这种方式牺牲了快速运行不同代码的便利，不是我们移植的初衷，这里就不多加讨论了。有些基于 Micropython 的项目，如 Micro:bit，会使用这种方式。

6.1 直接在交互式串口终端（REPL）上使用

在刚开始体验 Micropython 时，最直接的方式就是通过 REPL 了，这非常像是在一个命令行的界面输入一行一行的命令，比如前面“print(‘hello micropython!)”。不过，REPL 并不是一个简单的命令解器，它更像是让我们一行一行地输入一段脚本，并且每输入一行就立即执行。前面执行过的脚本的结果在后面可以继续使用。

了解了逐行输入并执行脚本后，接下来我们介绍一下 REPL 更多的重要功能。

6.1.1 中断脚本的执行

REPL 亦支持“Ctrl-C”来强制打断正在执行中的脚本，如 图 8 所示：

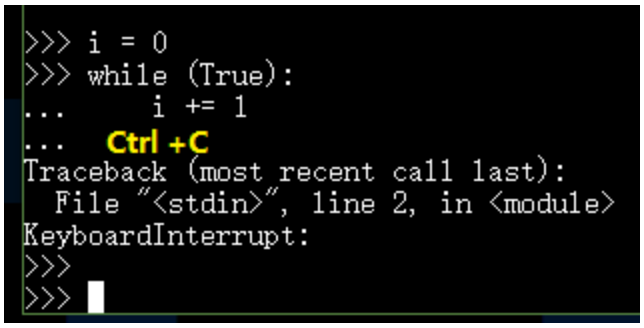

图 8. 在 REPL 中强制打断正在执行的脚本

上面显示的“KeyboardInterrupt”，解释了脚本被打断的原因：出现了“KeyboardInterrupt”异常。在实现时，是在串口中断服务程序中触发这个异常，并由 Micropython VM 检测出来。

6.1.2 在 REPL 下输入多行脚本

在 REPL 下，当一行以“:”结束并换行时，REPL 不会直接执行这一行，而是会自动缩进，并且允许手工输入后面的脚本。此后，需要根据程序语法和语义手工调整缩进级别。当缩进级别为 0 时，REPL 会认为一段脚本已输入完毕，并且一起执行。

然而，用这种方法在 REPL 下直接手工输入多行脚本不方便。此外，如果是缩进级别同为 0 的多行脚本，就无法一次输入多行了。为此，REPL 提供了粘贴模式。只要在 REPL 的一个空行下按 Ctrl-E，即进入粘贴模式：

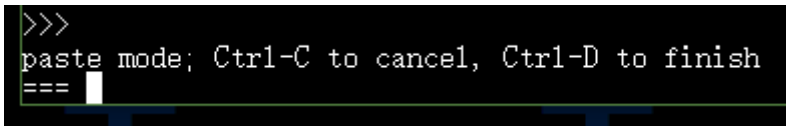

图 9. 进入 Paste mode

此时，可以在某个编辑器中复制一段已编辑好的 Python 脚本，例如：

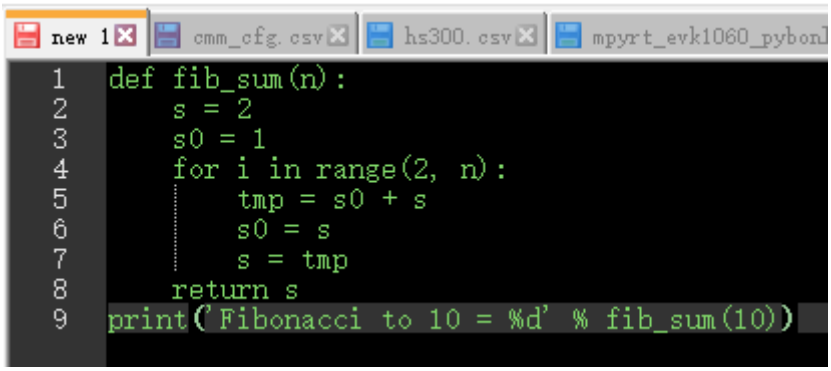

图 10. 在 PC 上编辑多段脚本

```
def fib_sum(n):
    s = 2
    s0 = 1
    for i in range(2, n):
        tmp = s0 + s
        s0 = s
        s = tmp
    return s
print('Fibonacci to 10 = %d' % fib_sum(10))
```

再在串口终端下粘贴（一般是鼠标右键）即可，例如：

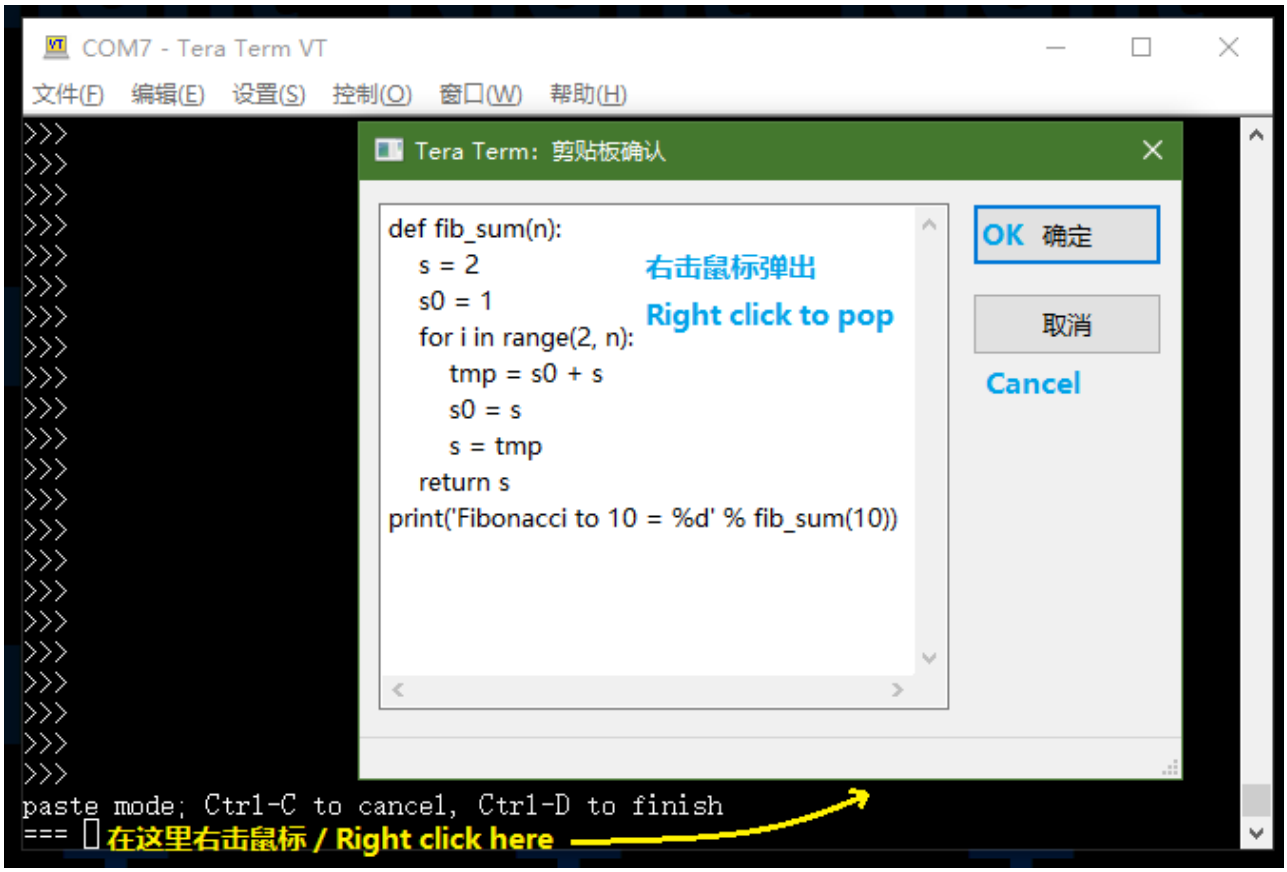

图 11. 使用 REPL 的粘贴模式

单击“确定”，则自动粘贴，然后在 REPL 里按“Ctrl-D”，则完成一次多行脚本的执行，例如：

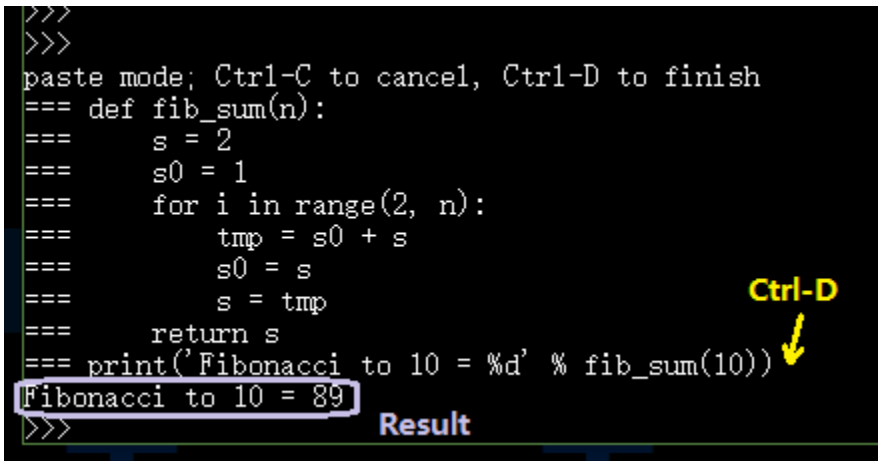

图 12. 在 REPL 下执行以粘贴模式输入的脚本

6.2 事先编写脚本并存储到 TF 卡中

直接在交互式终端上使用 Micropython 虽然很方便、直观，但是不便于开发有很多行代码组成的脚本，并且也不方便存储在文件中，反复使用。其实，真正的基于 Micropython 的项目开发基本上是把程序按文件组织的，这和传统的使用 C 语言开发嵌入式系统是相同的。

C 语言编程的程序从 main 启动，并且整个固件必须安置在指定的位置，符合特定的格式，才能成功启动。那在 Micropython 下又如何呢？

在 Micropython 下，启动流程其实是可以由移植者自己设计的。但约定俗成的是，**在启动期间，会先在根目录里搜索 boot.py，再搜索 main.py**。只要找到了它们就执行。我们在移植时，也遵守了这个启动流程。

其中，boot.py，顾名思义，是在启动期间执行的，扮演启动脚本的角色。比如，boot.py 脚本可以读取一些按键、跳线、拨码开关的状态来做一些配置，比如 USB 以什么设备打开等。在执行 boot.py 时，大部分外设和软件模块还没有初始化，所以 boot.py 能做的事也非常少。于是，如果没有定制化的启动流程，一般就不需要 boot.py。

我们真正要牢记的入口点是 **main.py**。

main.py，顾名思义，如同是 C 语言中的“main”函数，它负责运行用户程序的主体业务逻辑。在 Micropython 系统中的全部必需初始化已经完成后，会自动执行 main.py。

除此以外，还可以像在 PC 上那样，把整个程序的功能分散在多个 Python 源文件中。对于“xxx.py”，它们在 main.py 中可以通过“import xxx”来使用。这些文件也可以是使用 Python 开发的驱动或中间件。当然了，它们也必须是能被 Micropython 执行的才成。

6.3 通过 Python 代码访问 Micropython 文件系统

访问 Micropython 文件系统的方法和在 PC 上的差不多，都可以用 os 模块或者 open 函数。在本文对应的 Micropython 移植中，如果有 TF 卡，则 TF 卡的内容被加载到根目录”/”下，否则根目录中就是虚拟文件系统，并且有一个名为“/flash”的子目录，表示 QSPI Flash 文件系统的挂载点。

无论是否插入了 TF 卡，QSPI Flash 文件系统都被挂载为“/flash”，可以在 REPL 上通过以下命令来浏览内容：

```
import os
os.listdir('/flash')
```

如果要访问文件的内容，在 Python 脚本中，可以使用”open”内建函数返回文件对象，再使用 read，write，seek 等方法，这和在 PC 上使用 Python 一致。

再次提醒，不要在 TF 卡的根目录中放置名为“flash”（区分大小写）的文件夹，否则会被内部 QSPI Flash 文件系统所遮蔽。

7 Micropython 下的基础软件资源

在对 Micropython 有了最初体验后，接下来可以开始进一步了解一下 Micropython 这套软件系统。

7.1 Micropython 的目录结构

了解新事物，最好由表及里。所以，我们先对 Micropython 的顶层目录划分做一个简单的介绍。观察 Micropython 源代码的目录结构，可以直观地了解它的功能和组织方式。以下分别介绍各个重要目录的功能。

- py：这是 Micropython 的核心，支持 Python 程序的运行。包括编译器、字节码生成器、字节码加载器、虚拟机、支持垃圾回收的动态内存管理器、内建的模块和类等。
- ports：mp 在多款硬件/软件平台上的移植。Micropython 可以运行在包括 arm、RISC-V、xtensa、x86 等众多架构上，支持非常多的 MCU 器件。
- drivers：片外外设的驱动，它们一般是焊接在电路板上的独立芯片。
  - 既有 C 写的也有 mp 写的
- extmod: 非内建的 mp 模块，非常多。有些模块，例如如 vfs，仍然是常常使用的。
- lib: 支撑多个 mp 模块的底层 C 语言部分。
- mpy-cross : 交叉编译 mp 源代码，生成字节码。字节码可以固化到 Flash 里，由 mp 的虚拟机直接执行，不再需要在设备上编译 Python 源代码。
- tools：周边和扩展功能的脚本（如 mpy 模块)。

7.2 Micropython 基本内置功能

学习一门语言，在了解了它的基本语法之后，就可以熟悉它的重点内置功能了，这些一般也是程序员最常用到的。

- 内建函数与异常：与标准 Python 接近，常用的如：
  - print, len, range, enumerate, open, dir, str, int; type, isinstance, sorted, zip, sum, chr, ord, hex, oct, bin, min, max, sum, map; iter, next; input
- 支持垃圾回收的堆管理器(gc)
  - 扫描栈中的指针和静态规划的“root 指针”，把整个依赖链的内存块打上“免死标”
  - enable, disable, collect,mem_alloc, mem_free, threshold
- 实数与复数数学库。
- 其它微型库： uarray, ubinascii, ucollections, uhashlib, uos, …

8 Micropython 中的库

学习一门语言，有相当大的比重是熟悉它的库。语言越高级，对于库的学习就越重要。C 语言的库比较单薄，C++库就丰富得多了。而对于 Python，语言本身非常容易入门，对各种库的了解和运用反而更加重要，Micropython 也同样如此。在这里简要介绍一下。

8.1 模块与类型

Micropython 下，库一般有两种呈现形式：

1. 模块

   通过“import <module_name>”使用。在导入了模块后，就可以使用它里面的方法和类型了。这里的方法其实就像是函数调用，它们没有与之关联的对象实例，有些语言也称之为静态方法。

2. 类型（types）

   类型由模块提供，它们好比是面向对象语言中的类，封装了数据和操作。在 C 语言中，其实就是在某个结构体，以及所有能操作这个结构体的实例的函数。在使用它们时，先创建出某个类型的对象，然后就可以调用对象上的方法了。

8.2 精简的 Python 标准库

作为 Python 3 的一个精简实现，Micropython 也携带了常用的 Python 库，大多亦被精简成微型库。微型库以“u”开头，只实现了 CPython 模块功能的一个子集。这里列出一些常用的：

表 1. 常用 Micropython 精简标准库

名称	用途
内置函数与异常	不需要 import 直接使用
math, cmath	数学库，复数数学库
gc	垃圾回收器
uarray	数组
uos	微型 os 模块
uio	输入和输出流
ustruct	打包和解包二进制数据块

Table continues on the next page...

表 1. 常用 Micropython 精简标准库 (continued)

名称	用途
sys/usys	系统特定的函数，也包含了 stdio
ujson	JSON 编码解码
ure	正则表达式引擎
uzlib	Zip 解压库

8.3 与 MCU 和电路板的硬件相关的库

在 Micropython 中，通过 Python 模块和 Python 模块创建出来的对象来封装 MCU 和电路板上的资源。封装的方式有两种：

- 面向外设式：这也是最直接的封装方式，就仿佛是通过 Python 使用的基础 SDK，以驱动常用的外设。如果要使用以这种方式封装的 Python 模块和对象，需要对相应的外设类别有基础的了解。例如，ADC，I2C，UART，SPI 等，它们名字一般与外设同名。这类功能大多封装在 pyb 模块和 machine 模块中。值得一提的是，相比于 NXP 提供的 C 语言基础库(SDK)，这些同样面向外设的模块都只提供一个最常用的功能子集，不像 SDK 那样功能完备。既不包含少见的特色功能，也基本不包含直接面向寄存器的 API。不过，有趣的是，有些功能子集却是外设不一定包含的。比如，I2C 类的“scan”方法，用于把 I2C 外设当作主机使用，并且扫描 I2C 总线上的全部已连接的 I2C 器件。一般情况下，外设的寄存器不会直接提供这类高级功能。而且，在 C 语言下，对于这种结果不确定的 API，不便于用简单的类型做返回值，很可能需要手动申请动态内存。然而，在 Micropython 之下，只是简单地返回一个 list 或者 tuple，根本无需关心它要多长、内存在哪、如何释放等问题。这也给在 Micropython 下的 API 设计添加了非常大的灵活性。
- 面向应用场景式：相比面向外设式的 API，它们更加针对某一个具体的使用方式。例如，LED 类型，它直接提供基础 LED 控制的 API，但是底层在实现是自动调用 GPIO 和 PWM（带调光功能时）。它们提供的方法常常更方便特定应用场景的使用。

有一些简单的功能，例如 Pin 类型，它封装了 GPIO 的功能，兼有两者的特点。

在本篇文档介绍的 Micropython 移植中，有两个模块封装了芯片与开发板上硬件资源的功能：

1. machine 模块：系统级的基础操作和信息，如获取主频、休眠控制，甚至是像复位和直接读写 4 GB 地址空间这样的“root”级别操作，用不好足以危及系统的正常运行。
2. pyb 模块：对大部分 MCU 外设的功能都封装在这个模块里。有一部分比较基础和常用的功能也包含在 machine 模块中，例如 UART 类型，freq()函数。

Machine 模块和 pyb 模块最早出现在 Micropython 原作者打造的 PyBoard 开发板中，本移植也模仿它们的用法来封装 i.MX RT1050/1060 系列处理器中的 MCU 常用功能。

9 查看在自己的 Micropython 系统中编入的库并使用

Micropython 系统是高度可定制的，而且在移植过程中还可以添加自己定制的专用库。要想查看自己正在使用的 Micropython 系统中已经包含的模块，可以在 REPL 上输入以下命令：

```
help('modules')
```

例如，笔者在 i.MX RT1060evk 上配置的一个 Micropython 系统，有以下的模块：

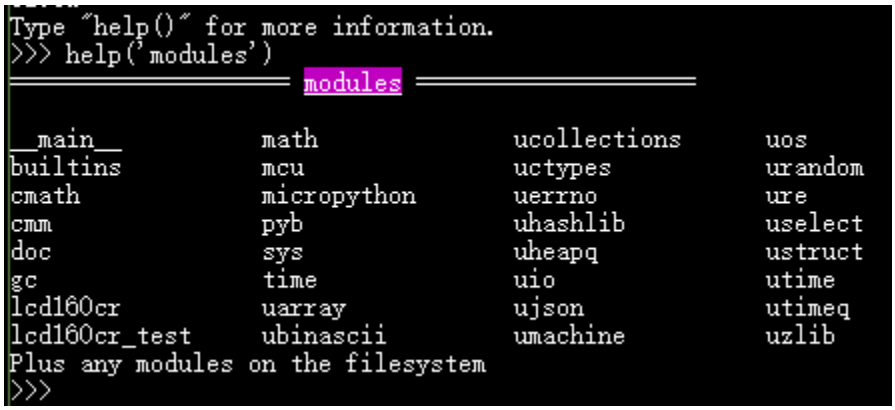

图 13. 查看 Micropython 固件中编入的模块

可以看到有大量以“u”开头的 Micropython 微型库。

上面列出来的只是众模块名，如果想要查看模块里的方法和类型，可以导入这个模块，并且使用“dir”内置函数。在 Micropython 下，还可以在输入“模块名.”后面按一下 tab 键，REPL 会自动列出可使用的方法、类型、以及常数。例如，我们看一下 pyb 模块的内部：

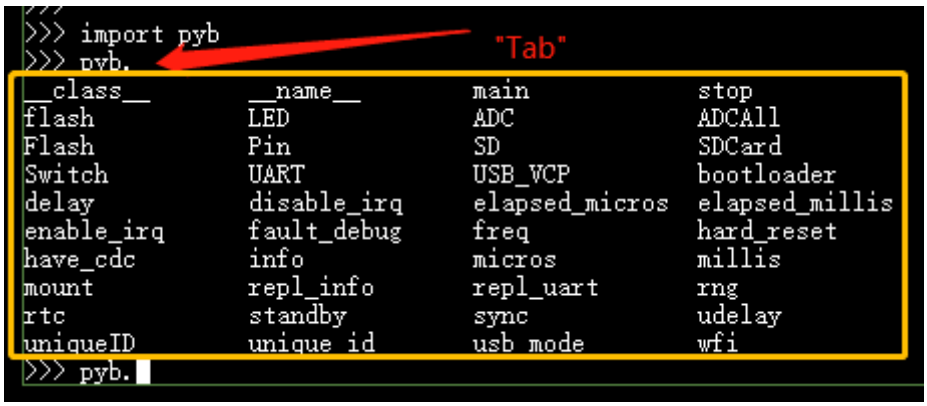

图 14. 使用 Tab 键浏览模块

pyb 模块封装了 MCU 的常见功能，它模仿了 Micropython 在其官方的 pyboard 上对 MCU 功能的封装方式。从上图中可见 pyb 模块里面包含了很多成员。一般来说，小写的是模块中的方法，大写的是模块中的类型或者是常数。通过 Python 的”type()”内置函数，可以查看每个名称背后是什么。例如，对于”freq”这个名字,查出它是一个函数 (class “function”)。

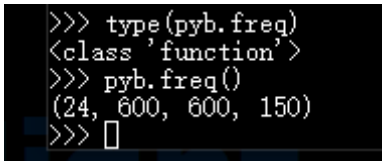

图 15. 获取模块成员的类型

在调用它后，显示出当前系统的各项主频（振荡器频率、CPU 频率、AHB 频率、外设根时钟频率）。

通过同样的方法，知道 UART 是一个类型：

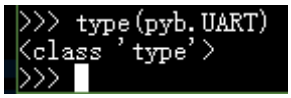

图 16. UART

“UART”是在 pyb 模块下对 UART 基本功能的封装类型。可以创建此类型的实例，再调用它的方法，例如：

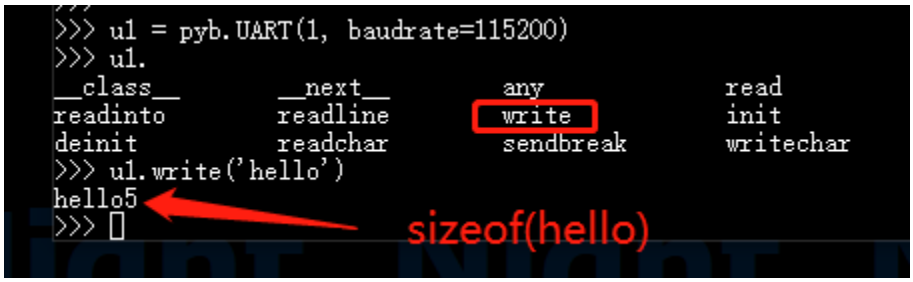

图 17. 创建实例

上例中，通过“pyb.UART(1, baudrate=115200)”，使用 1 号 LPUART 外设创建了一个 UART 类型的对象实例，设置波特率为 115200，并且由名称“u1“来引用。输入“u1.<Tab>”后，REPL 自动列出了 UART 类型支持的方法。比如，write 方法可以写出一行字符串。

还可以发现，输入命令“u1.write(‘hello’)后，输出了“hello5”。这是因为 micropython 中的字符串在后面还记录了字符串的长度，并不是像 C 中那样的零结尾。UART1 在 1060evk 上刚好就是 REPL 所使用的 UART，所以它的输出才能被 REPL 所捕获并显示。

10 用好带垃圾回收（GC）功能的动态内存管理器

对 Micropython 的使用有了初步了解后，还是需要再多了解一点垃圾回收这块，这毕竟与使用 C 语言编程的内存管理哲学完全相反——相信程序员 VS. 不相信程序员！

在 C 语言中，我们必须自己管理内存，清楚地知晓静态内存、栈内存和堆内存的区别，以及各变量，尤其是指针变量，是如何安置的，作用域如何。特别地，对于通过 malloc 得到的动态内存，必须小心翼翼地管理它们的生存期，以及指针的数目，稍不留神就可能埋下隐蔽的安全隐患。这也是 C 语言“一切相信程序员”的哲学带来的结果：我们在享受至高无上的内存使用权力时，也要承受事无巨细的内存管理的义务。最常见的麻烦事之一，就是为每个 malloc 配对一个 free。

在 Python 中，这种权力和义务被收回了。我们在使用任何 Python 对象的时候，都无需手动分配和释放它们的内存，Python 语言的运行环境接管了这一切，在 Micropython 中也是这样。如果你使用 Python 只是为了让计算机系统以可编程的方式给你干活，那这种方式无疑是莫大的解放：要成为 Python 程序员不再首先得成为一个系统软件程序员了，我们可以集中精力对付自己的应用要解决的问题，而不用亲自处理内存管理的繁文缛节。

事实上，Python 中的对象也是从一个堆内存分配出来的，这个堆的管理器也同样有 malloc 和 free 的方法。但不同的是，这个堆管理器还记录了其它的一些线索，可以知道某一时刻哪些它正在管理的内存块还在使用中，并且自动为不再使用的内存块调用 free 操作。

这个功能在 Micropython 中叫作 Garbage Collector (GC)，它基本上是一个基于定长块的分配器，只是它一次可以分配多个连续的块。块的长度是 2 的方幂，一般在 16 字节以上。Gc 使用双位数组来跟踪各个块的分配情况。当 gc 发现没有可用内存可供分配时，就启动垃圾回收操作。gc 会扫描脚本的栈和当前寄存器，以及一个特殊的“root pointer”清单，查找它们之中是否有可以当作指针来解读的 32 位变量，并且指向了自己分配出去的内存。只要找到了，就会标记这段内存的所有块为“它们不是垃圾”。这还不算完，它还会扫描所有“不是垃圾”的内存块的内容，以对齐的 4 字节为单位，全部当作潜在的指针来处理。只要发现任何一个潜在的指针指向了自己分配的某段内存，就把这段内存也当作“不是垃圾”。这个操作被递归地执行。这非常像筛查新冠肺炎的感染者和他们的密切接触者，以及密切接触者的密切接触者……。最后，那些与“不是垃圾”的内存没有关系的内存块，就会被当作是垃圾，并且被 free 掉。

分析上面的过程，可知这个递归的过程是非常不确定的。如果给 gc 管理的内存容量大，块的粒度小，Python 程序中使用的变量多，变量之间的引用又盘根错节的话，这种递归式的筛查可以消耗大量的时间，甚至长达十万个 CPU 周期以上！在此期间，脚本是不能执行的。所以，对于绝大多数硬实时的控制系统，都不可能接受这种最坏情况的响应延迟。在实践中，为了使基于 Micropython 的系统能满足更多的实时应用，常常把实时任务由底层的 C 语言部分实现，并且手动管理它们的内存。

即使是在 Python 语言层面，也有方法尽量降低捡垃圾所带来的不确定性。Gc 把一些关键的功能绑定到了一个名为“gc”的 Python 模块上。它有一个重要的方法叫“collect()”。在写 Python 脚本时，如果意识到自己的脚本会快速生产和消费大量的内存块——也就是快速制造内存垃圾，可以适当插入一些“gc.collect()”调用，以更加频繁地“大扫除”，避免垃圾“堆积如山”后才花大量时间做一次大扫除。当然，这个代价是降低垃圾的回收效率，也就是以牺牲整体性能为代价降低不确定性。

11 获取更多帮助

除了前文提到的内置“help()”命令，Micropython 官方网站有非常完备的文档系统，位于 MicroPython documentation。包括对 Micropython 语言本身、库、以及一些开发板的功能的介绍，可以参考。

12 参考

以下文档/网站提供了更多的信息：

- MicroPython
- MicroPython libraries
- micropython
- micropython-rocky

13 版本历史

版本号	日期	重大更新
0	2021 年 4 月 27 日	初次发布

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 27 April, 2021

Document identifier: AN13242

arm