工作笔记

根据应用内容，简化

tangyapeng

# \*\*\*\*\*\*\*\*\*\*star-gather项目学习笔记\*\*\*\*\*\*\*\*\*

## 远程桌面登录信息

• 使用windows远程桌面，一般使用自己账户登录远程桌面：116.113.133.5:19014

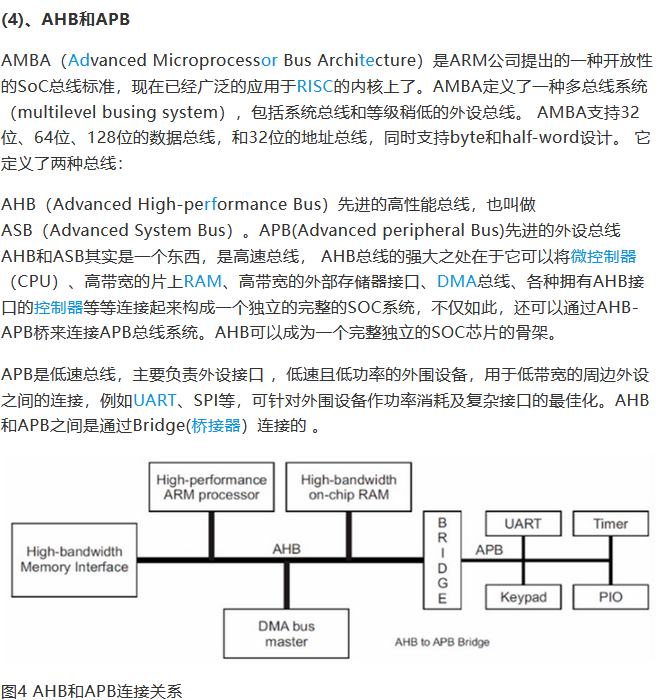
• 用户名：star-gather\<姓名的全拼>

• **密码：Typ@24061729 (其他的密码也是用这个)**

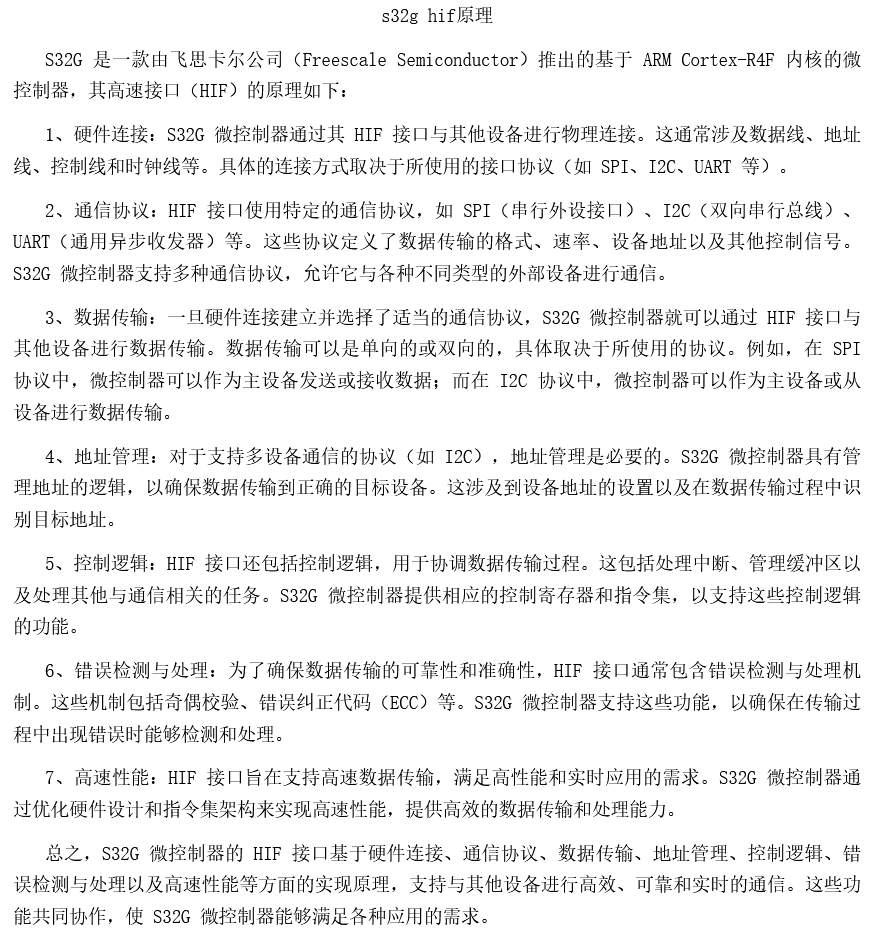
## 代码阅读笔记

C1\_Software\_Architecture\_Spec.docx 为顶层设计文档。

## 关于Cortex系列的总线



## HIF高速接口interface



## IPC进程间通信

综合介绍几个方式。



## MPECAN四个核心

Core0：发送及配置信息的处理，当然底层配置信息是写入到配置寄存器中的；

Core1：传输协议处理。

Core2：接收数据流的处理；

Core3：CAN帧格式转换器；

**章节8.4.1.1.**

其他：再host总线上具有多个FIFO、Share memory用于数据交互。

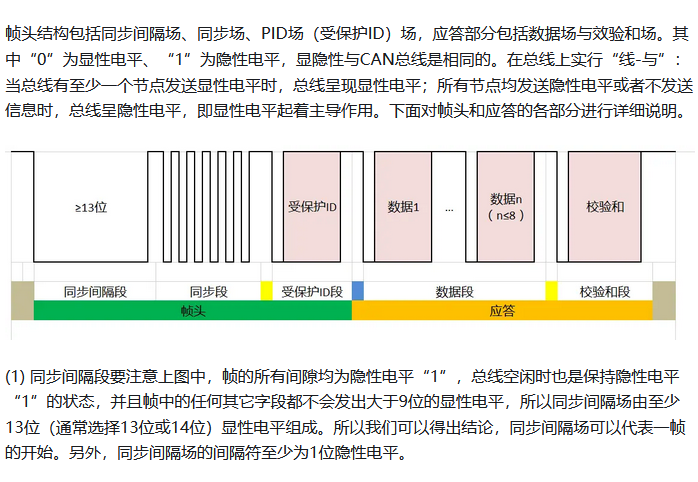
## 几个疑问

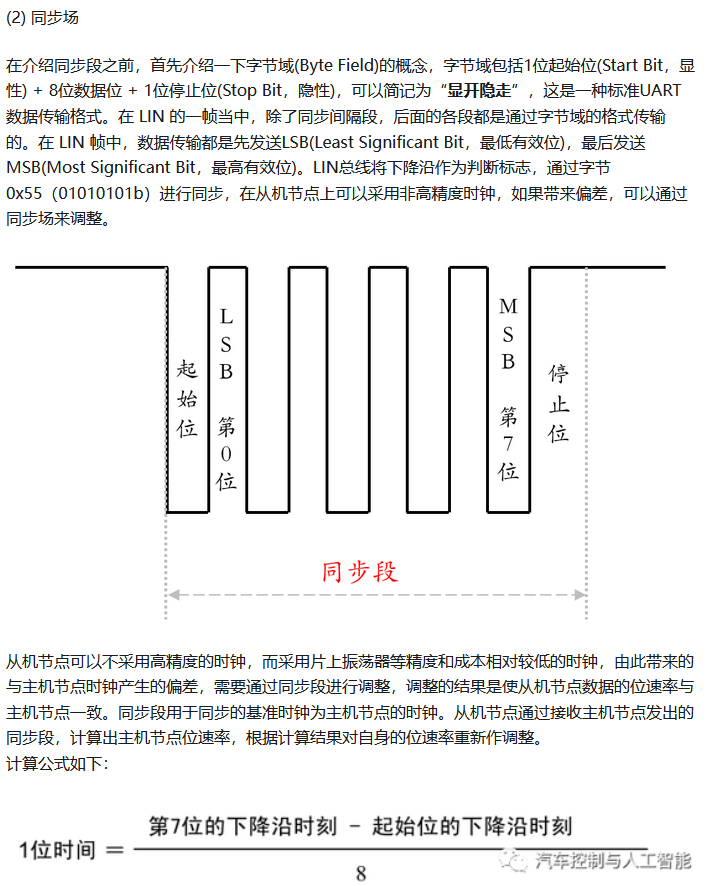
1. HOST和MPECAN之间交互接口有哪些？协议是什么？有哪些指令和内容？
2. MPECAN FW要实现的功能有哪些？

比较繁杂

## Lin总线

一主多从的总线结构，点名通信的方式。单总线节点个数限制16（1主15从），速率20Kbps。







一般而言，车内会选择统一字节数，最常用的是每帧传递8个字节。没有数据长度段。

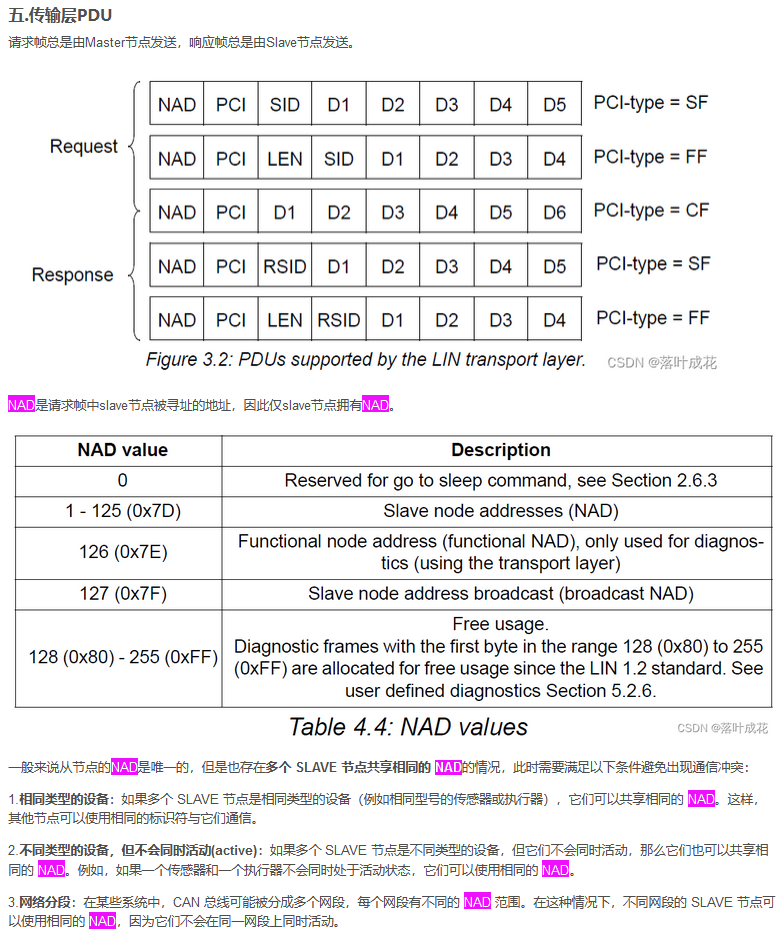
LIN帧类型

1. 无条件帧

这个帧类型类似于点名某个节点，把当前的帧空间留给该节点来进行报文的发送。此时，其他的节点可以接收来自该节点的报文信息（包括主节点）。

如何点名？

1. 事件触发帧
2. 偶发帧
3. 诊断帧 用于配置、识别和诊断。
4. 保留帧 后续扩展使用。



## LIN传输层知识

核心是了解 传输层完成地址绑定。

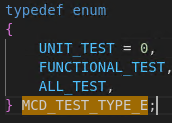
## MCU测试接口代码阅读

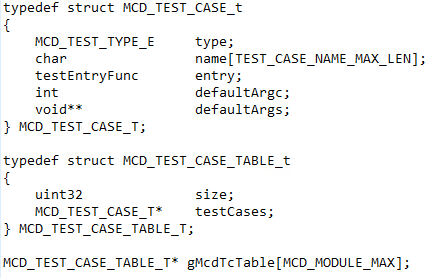
实现架构为：PC端运行脚本或者直接发送串口命令到目标机，目标机接收到指令后进行指令的解析处理，然后按照指令要求调用事先注册的函数，这些函数都是测试程序集。

程序实现逻辑：

1. 平台、外设等初始化；
2. 注册测试程序集；
3. 根据宏定义，按照命令执行对应的测试项，或者全部执行；执行测试项时候调用简单的log函数，可以根据配置为存储到ram或者通过uart存储。

//记录一些必要的数据结构

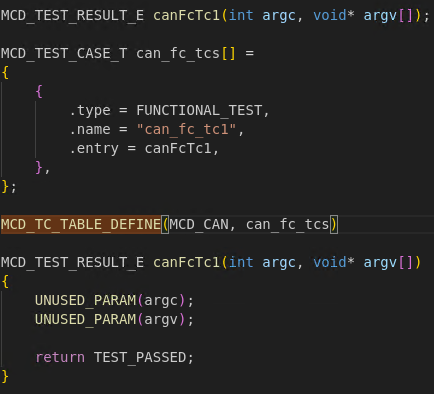






测试项的实现方式：

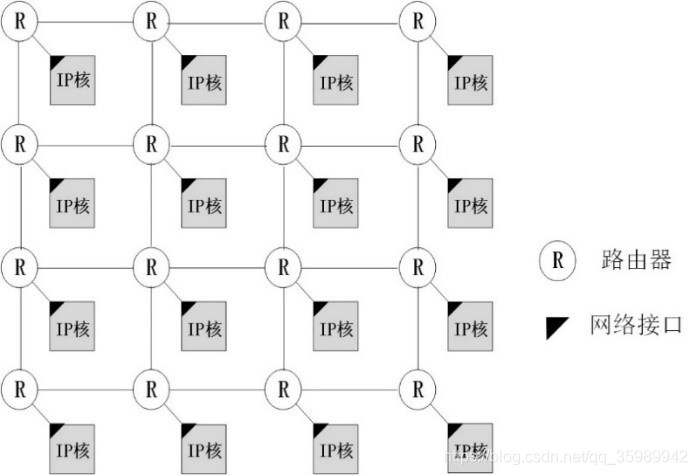
通过该函数进行调用，创建对应的全局结构体，作为某个外设的测试项列表结构体变量，



## NoC （Network-on-Chip）

片上网络，其结构组成如下所示：区别于

传统的总线型或交叉开关（crossbar）等互联结构有可扩展性差、带宽较低、延迟较大和功耗较高等缺点



# \*\*\*\*\*\*\*\*\*\*MPECAN 代码相关 \*\*\*\*\*\*\*\*\*

## Test

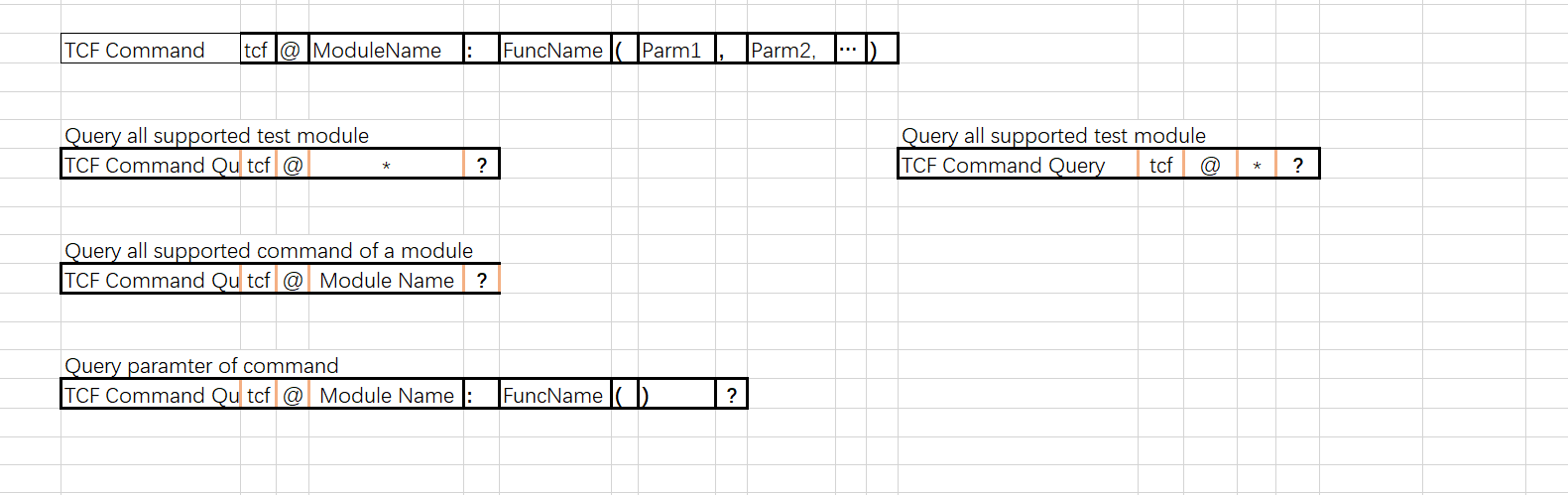
测试框架：

PC端运行python测试脚本，控制串口进行测试指令的发送；

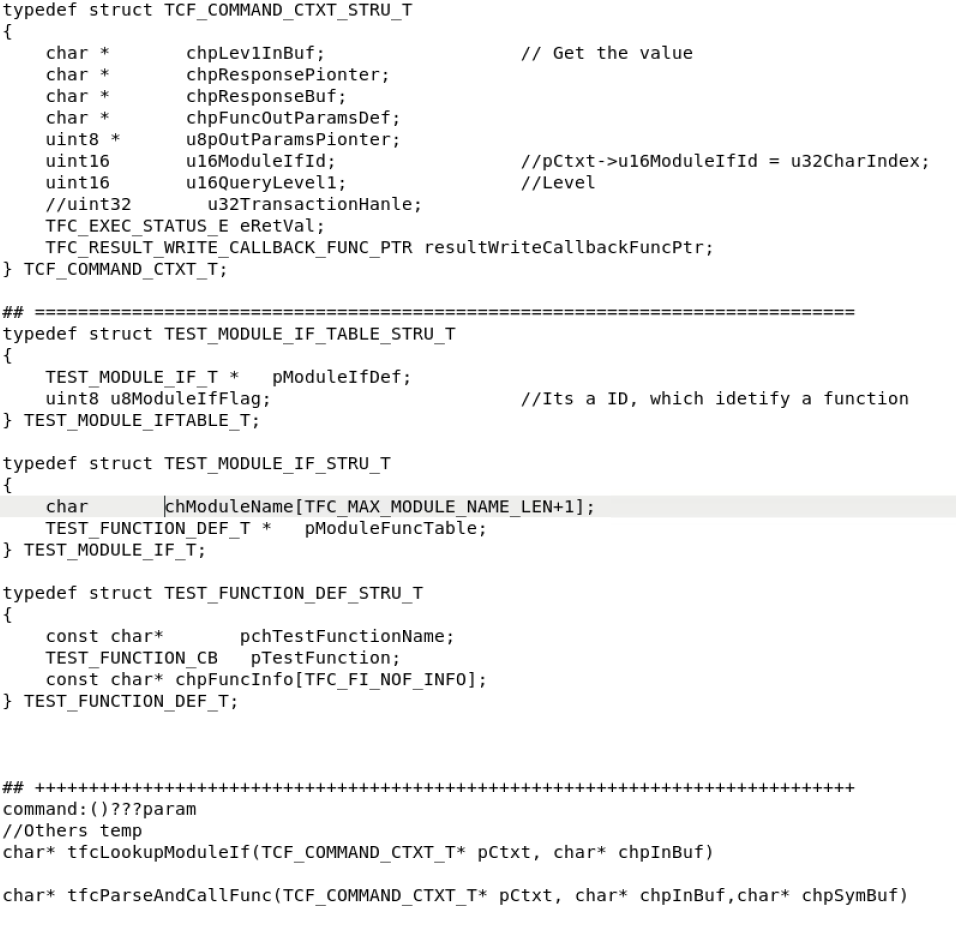
下位机（C1）进行测试指令的接收、解析、处理过程；

下位机中，MCU完成：LIN的初始化配置，core\_dst完成串口接收，指令解析工作。

测试命令格式：



几个用到的核心数据结构体：（完善中）。。。



该部分的核心是：

通过串口接收字符串，在接收中断中，check buffer，解析出测试指令、测试层级等信息，然后在全局测试程序集中调用对应的函数执行调用，中间涉及到name匹配等操作。

## Host and MPECAN之间的交互都是通过中断函数实现执行序列的

1. 通过指令FIFO中断执行命令中断服务函数；
2. 通过IN、OUT FIFO中断，执行对应的数据收发的中断服务函数；

## Host and MPECAN interface data structure

CMD FIFO用于传递CAN控制器通道ID。

Host和MPECAN交互使用FIFO和share memorys，其中，memory结构体名称为：



MPECAN\_CAN\_SHARE\_MEMORY\_TYPE\_T;

目前需要记得几个核心结构有：

全局数组gChangelMaxTxMBCount：保存不同CAN控制器对应的最大传输帧数，大概是每次发送的帧数，后续再详细确认。

**Host对MPECAN通信而定义的命令**：有命令号、命令参数（定义成不同的数据结构体，采用union方式存储），返回值（用于Host读取返回值）；

**接收消息缓冲区描述符RxMB：**filterID/搜索器搜索结果等

**发送消息缓冲区描述符TxMB：**

**CAN报文缓冲区**：主要是Index字段用于检索。

*关于具体配置、操作的寄存器等相关的细节，暂时不深究，后期遇到问题再钻研。*

## Host and MPECAN LIN模块之间接口

MPECAN CAN模块和LIN模块采用不同的指令和数据通讯方式，可能是方便进行通道控制等；

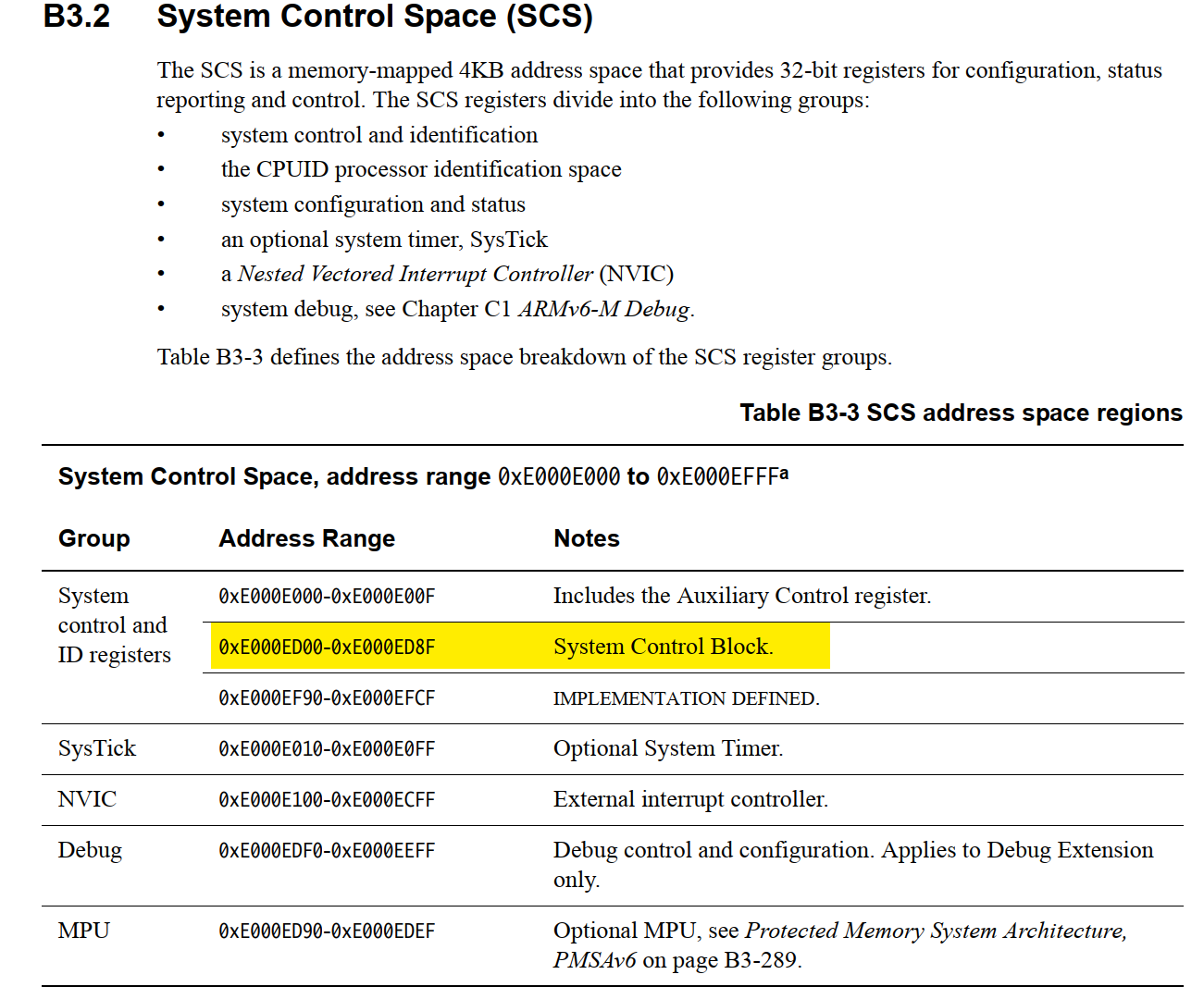
Lin总线与host之间采用core2core事件同步，数据和指令的传输采用share memory通讯。

其中core2core模块每个core之间有16个事假可以选择，目前的用法是标志为4个不同的lin通道。

## FIFO 101/102用作HOST and MPECAN commucation port

并且该fifo写入数据触发中断，中断由中断集中器转到core\_DTX.

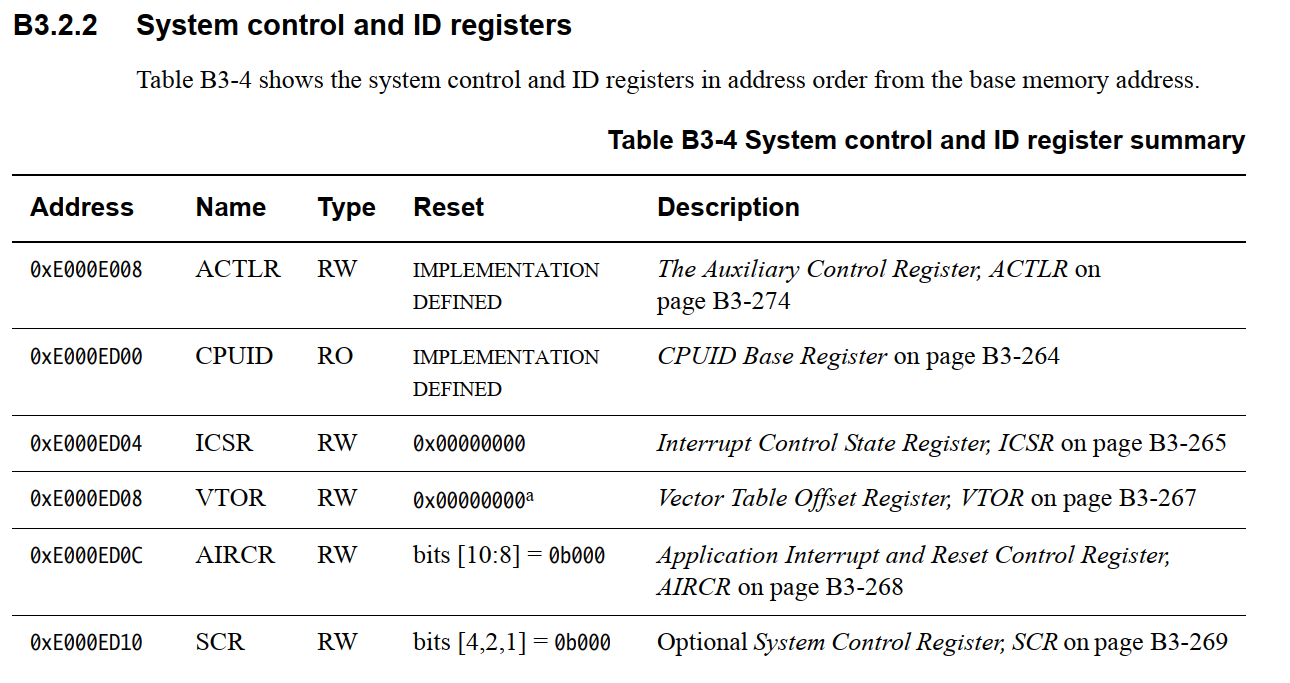
## cortexM0+(armV6m)System control space

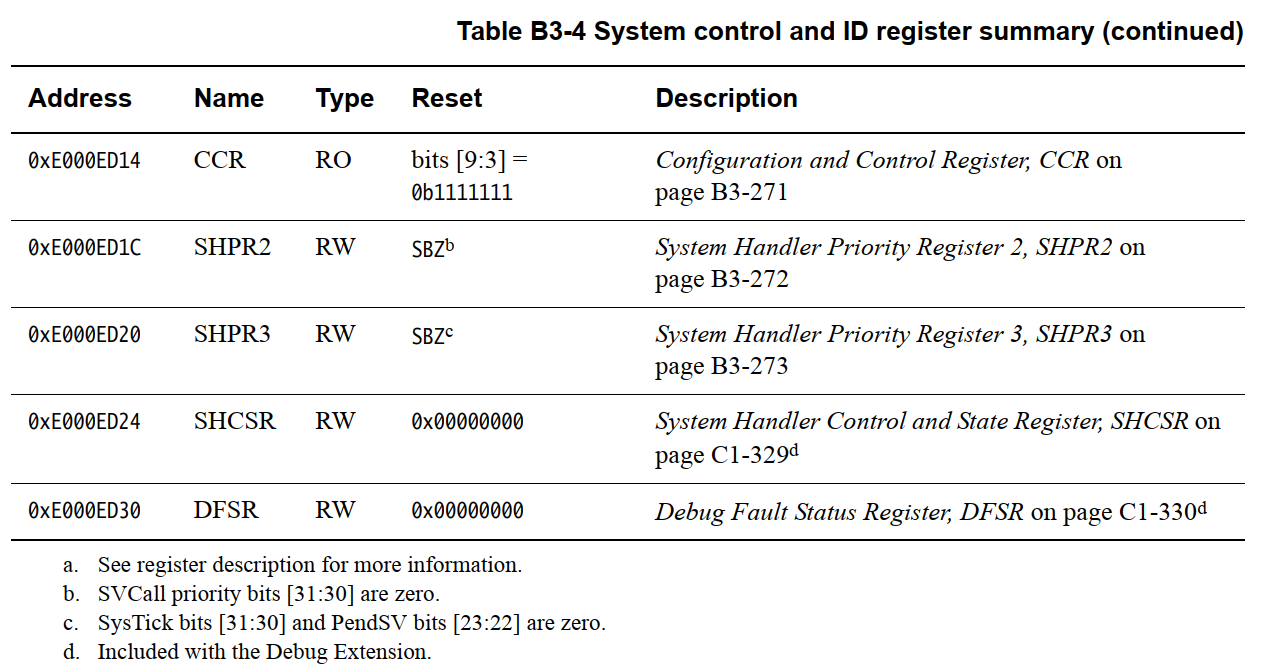


This excludes external interrupt handling. The NVIC handles all external interrupts.

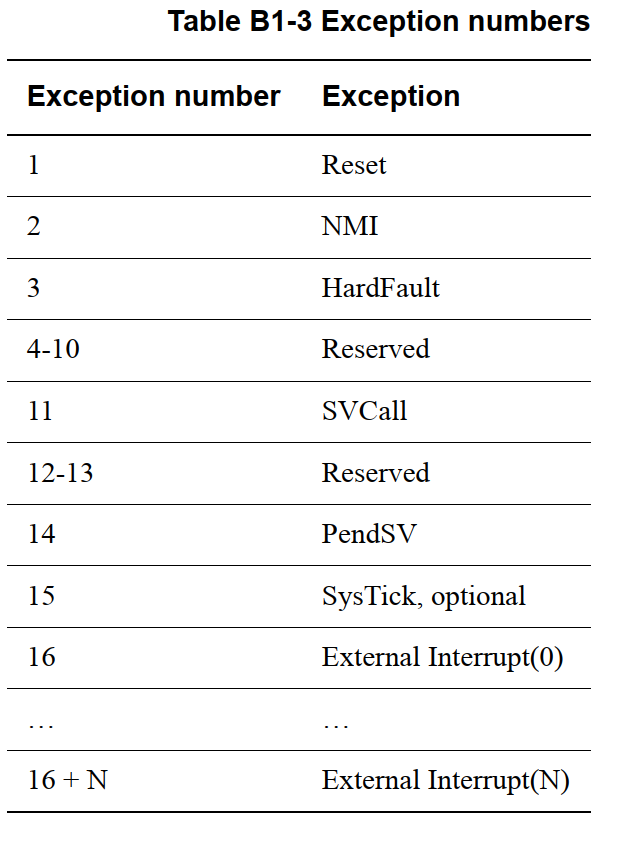
• The exception number of the currently executing code and of the highest priority pending exception

• Miscellaneous control and status features





## cortexM0异常号/中断号



Stm32F4的中断配置表格，可以参考英文手册的P374.

目前对于MPECAN的中断号定义，没有找到相关出处。

## 代码中关于中断的处理逻辑

* 1. 在启动代码定义的中断向量表中，各个中断的调用函数为统一的接口函数，即：cm0InterruptHandler()；
  2. 如本章节的3小节的说明，在ICSR寄存器中，可以读取当前异常号，异常 号和中断向量表是对应的；
  3. 代码中，代用全局数组的方式把异常号和中断服务函数关联在一起，在任意中断发生时候，进入到中断服务函数cm0InterruptHandler(),该函数的执行序列是这样的：首先读取ICSR寄存器，获取异常号，作为参数，获取对应的中断服务函数；
  4. 中断函数的定义和注册，使用接口INSTALL\_ISR()

## 检索中断号

搜索：

INSTALL\_ISR关键字。

# \*\*\*\*\*\*\*\*\*\* 嵌入式知识 \*\*\*\*\*\*\*\*\*

## LD链接脚本基础知识

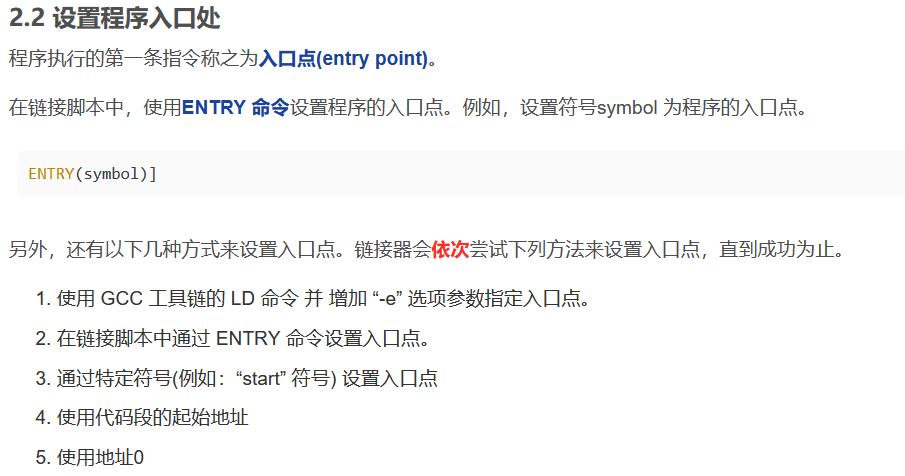
因为最初直接用的机器加载，所以习惯叫loader。

arm-linux-ld -Ttext 0 start.o sdram.o -o sdram.elf

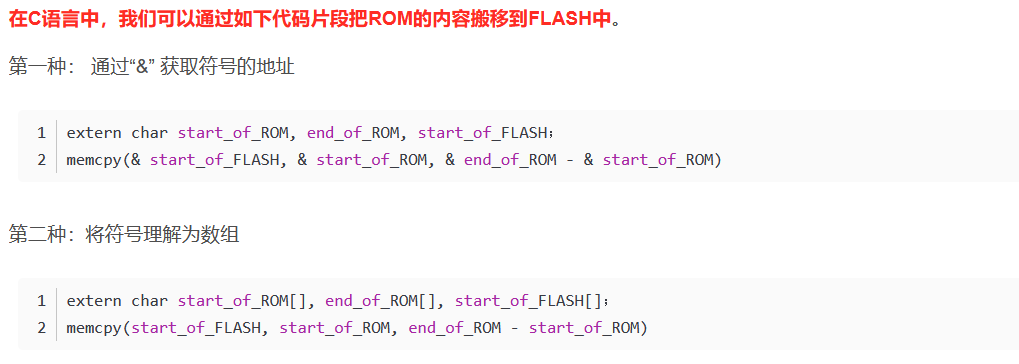


联想 程序员的自我修养 中关于elf中的段的相关知识，最终程序需要知道代码段、数据段、bss段等段所在的地址及地址符号表以及其他的符号表，基于这些目的，进行考虑。

**设置程序的入口地址，根据优先级，可有如下顺序：**



**符号引用，采用 “ = ”来赋值符号，并且该符号作为地址符号可以进行运算；并且符号作为符号表存在于elf文件中，可以在代码中引用，应用方法有如下两种：**



**输入段，一般为自定义或者生成的\*.o文件的各个段组成，输出段，一般为可执行文件或者库文件的总和段。**

**两个例子：**





Linker script中的常用命令：



新增：

INCLUDE命令，一如其他脚本的应用。

NOLOAD命令， 表示在程序加载时候当前段不被加载到内存中。

简单应用总结：

链接器用在链接阶段，用于把编译、汇编后生成的目标文件整合到一个统一的text段、data段、bss段，自定义段、调试相关的段等，也包括用户对内存的分段定义，地址定义，用于生成符号表，以在源码中应用之。

一般是先用MEMARY命令标识当前设备的内存地址等信息，然后执行SECTIONS命令，以分配各个段。并且，当加载地址和存储地址不同时候，可以用AT命令引用ram地址的方式来实现嵌入式的程序加载。

优点：

链接器可以在内存部分耗尽时自动检查，并以错误结束

我们不需要将点指针保持在正确的值

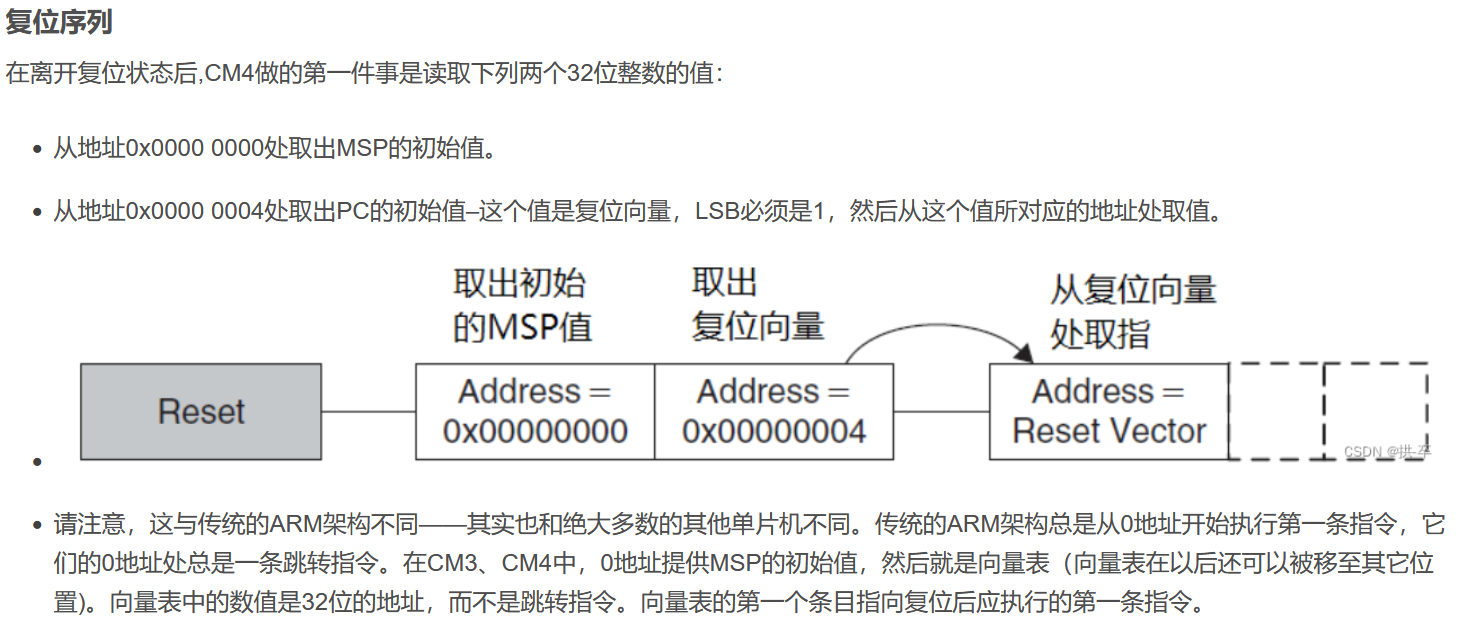
链接器脚本会自动解释代码的位置

新版本的 ld 可以以用户友好的方式显示所有内存区域的内存消耗

## ARM启动代码理解

1. **单片机上电、程序启动过程**

目前已知cortex-M3/M4/M0单片机的起始地址是以Stack Top开始的，存储的是栈顶地址，紧接着第二个地址存储的是reset\_handler的地址。该系列单片机的启动方式为：



在reset代码中，可以执行可执行文件的拷贝命令等操作，具体后续遇到再完善；另外，reset还执行未初始化变量的clear操作等。

通过reset函数，最终可以通过调用C库的\_start符号最终调用main函数，或者可以直接修改启动改吗中reset\_handler直接调用自定义的函数接口。（问题：*C库中的调用过程是什么？*）

（PS：在不应用C语言时候，也就是不使用栈顶指针，这时候可以不用初始化MSP。因为C语言的函数调用的本质就是基于堆栈空间的各种入栈、出栈操作；当然，这也要求汇编中不能使用堆栈相关的命令。）

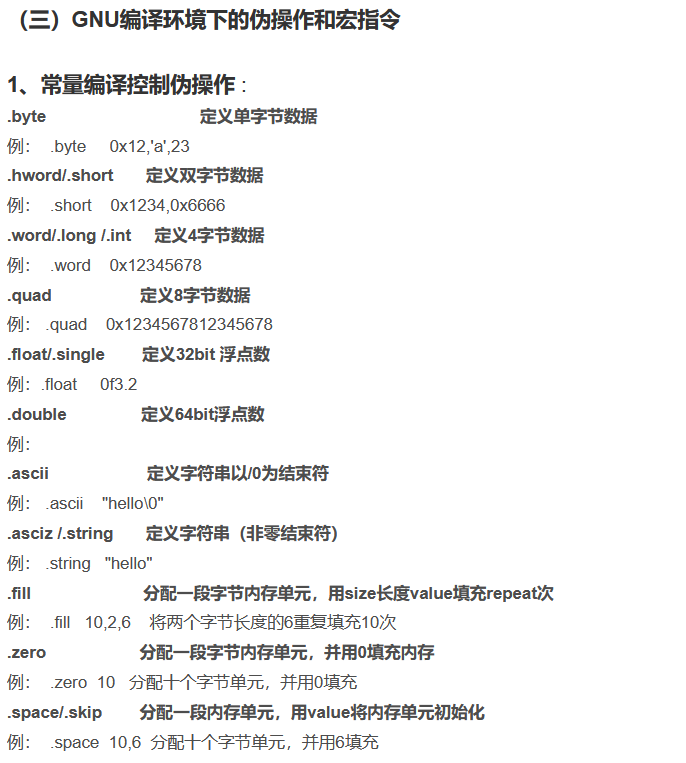
也就是在起始阶段写入了栈顶指针，该符号的值一般在启动代码中定义或者在链接脚本中定义。

那么堆空间呢？

堆空间是程序设计者通过函数动态引用的内存空间，所以只在运行时会用到，单片机硬件是不用感知堆空间的存在的。

1. **启动代码的作用及与链接脚本之间的关联**
2. 启动代码可以认为和其他生成的目标文件类似，具有各种段，最终通过链接脚本进行最终的链接，生成text、data、bss等段。根据cortexm3等系列单片机的启动过程，需要在链接脚本中更改vector段放在最开始的位置，以执行上电启动过程。
3. 代码中自定义的一些内存段符号表可以在链接脚本中定义，方便在源代码中引用，并在链接时候链接到对应的地址空间。
4. 其他：待查询完善；

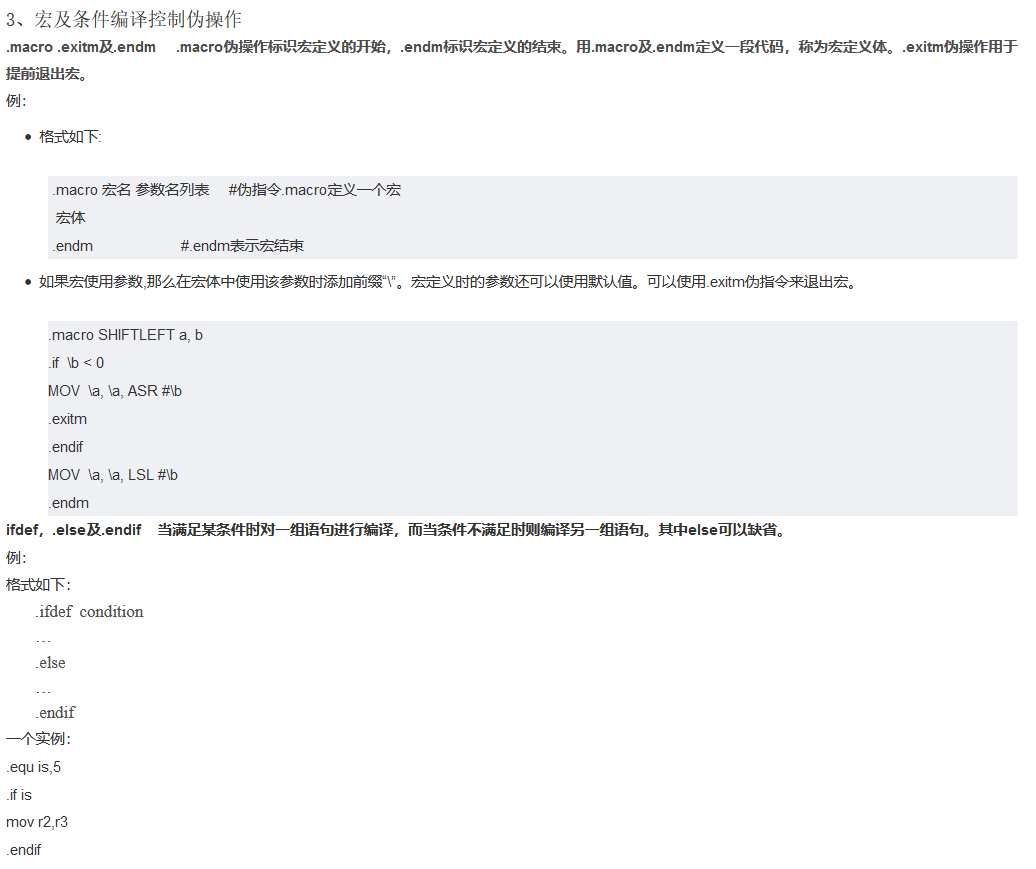
**Arm汇编一些用到的指令记录：**





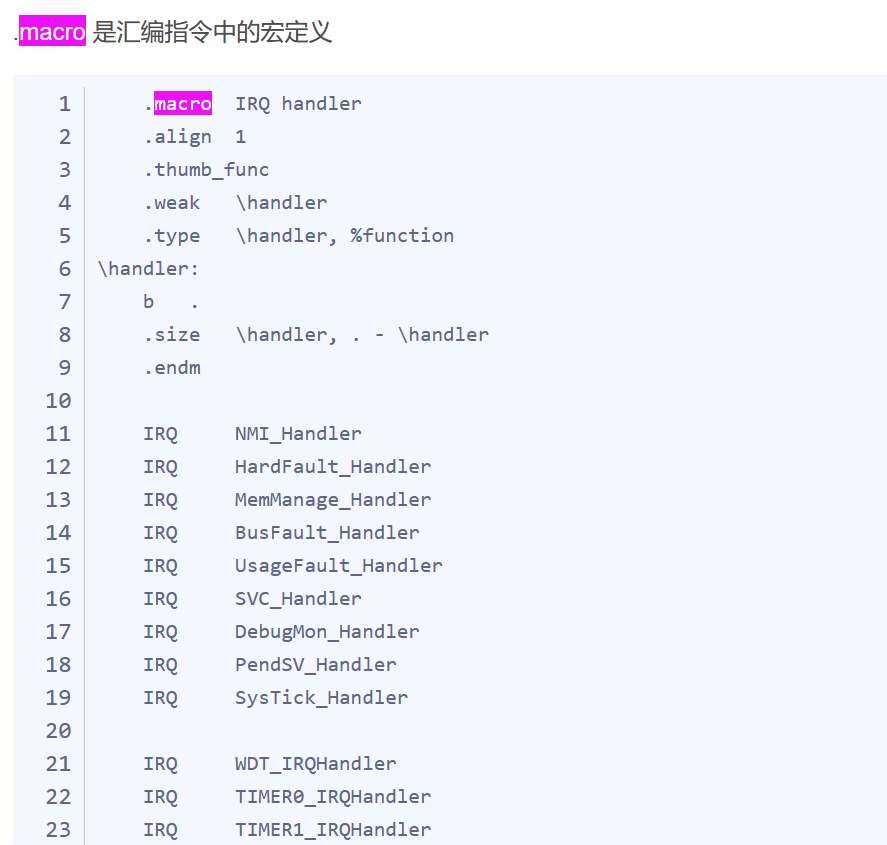
---------------------------------------------------------------------------------------------------------------------

\_start意义及在LD文件中的entry。



**基于该知识点的一个cortexM3中的使用实例：**

**其中，handler是IRQ宏名字的参数，宏的功能是对该参数进行一系列的操作。**





## ARM架构介绍

ARMv6-M与Cortex-M的关系：

ARMv6-M是ARMv6架构的一个子集或变种，专为微控制器设计。

Cortex-M0和Cortex-M0+是基于ARMv6-M架构的具体处理器实现。

Cortex-M0处理器采用了三级流水线结构（取值、译码、执行），并且支持Thumb指令集，包括16位和32位的Thumb-2指令集。

其中，cortexM0中NVIC部分作为系统控制器的一部分，其寄存器结构及地址是在armV6m中规定好的，在cortexM0手册中也是引用armV6m的信息。

## 打算

# \*\*\*\*\*\*\*\*\*\* Python \*\*\*\*\*\*\*\*\*

## 安装问题

在windows平台安装

pip install --upgrade --target=d:\python\python37\lib\site-packages urllib3

*其中 –upgrade是来自命令行的提醒*

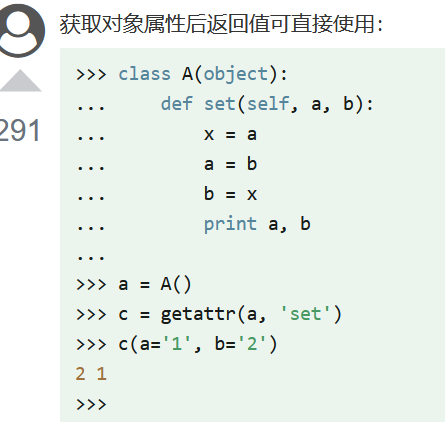
*git@github.com:leeguandong/learn\_python.git*

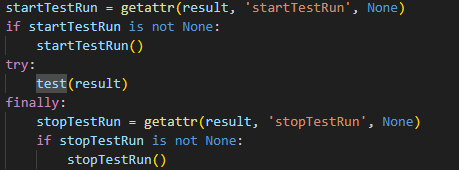
## Linux下python库的默认路径



## 内建函数

**getattr()** 函数//用字符串的方式获取 《类的实例》的成员，并可直接调用。





## Python模块导入的搜索路径及其顺序



所以，当module存在，但是import找不到时候，可以采用如下几个方法：

1.放在mod.py脚本所在目录，如果是交互模式放入当前目录

2.在启动解释器之前，修改环境变量包含所在的PYTHONPATH目录。

3.放入mod.py与安装相关的目录之一，您可能有也可能没有写入权限，具体取决于操作系统。

4.附加选项：



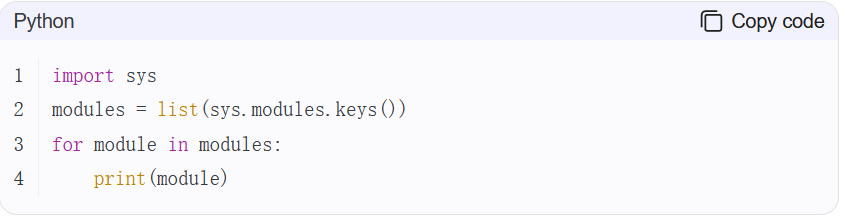
## 打印所有安装的模块列表

import sys

modules = list(sys.modules.keys())

for module in modules:

print(module)

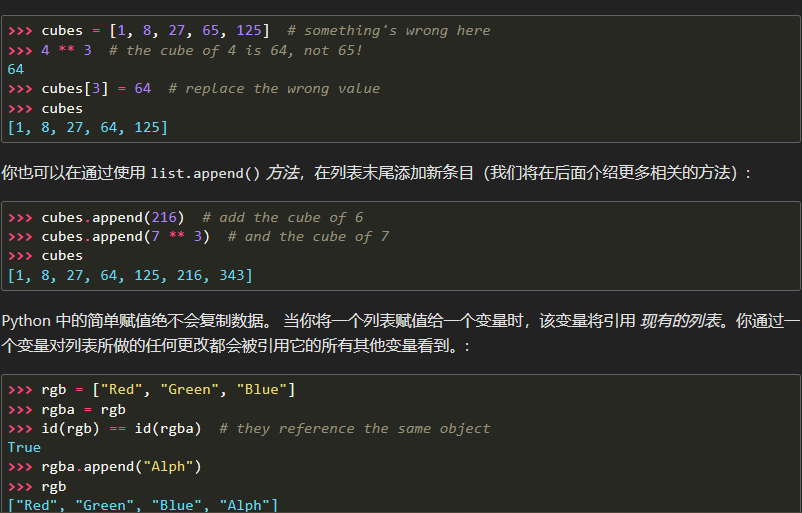


## 终端输入which python 确认当前python版本

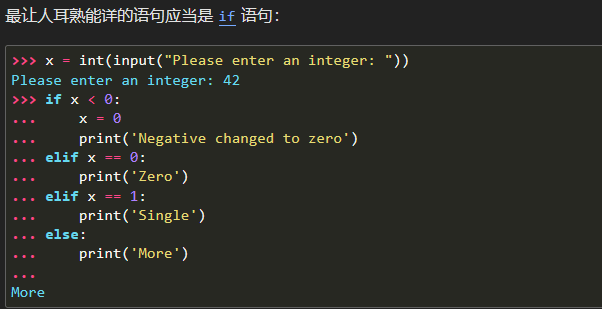
## Conda(环境管理工具) 待研究

## List

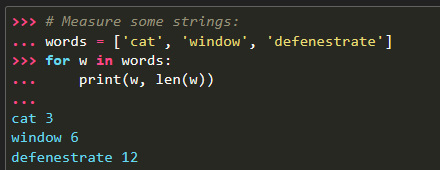
赋值操作为引用，没有拷贝list。



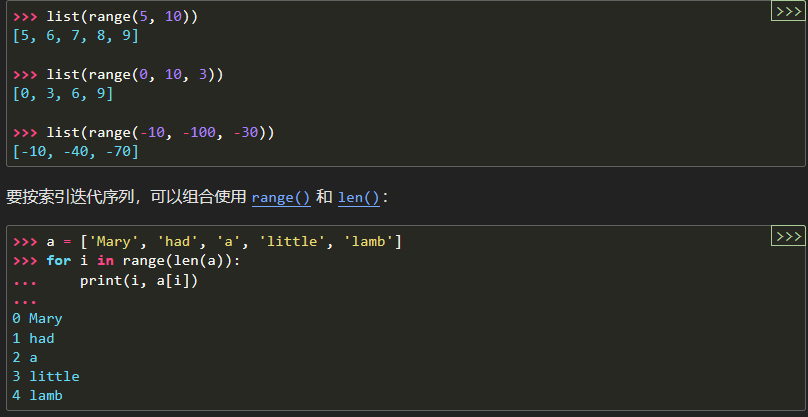
## If （注意缩进tab）



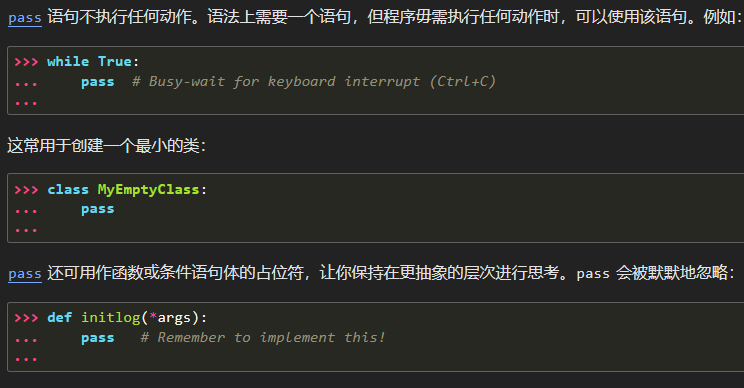
## For



## Range()函数



## Pass 不执行任何操作，语法需要

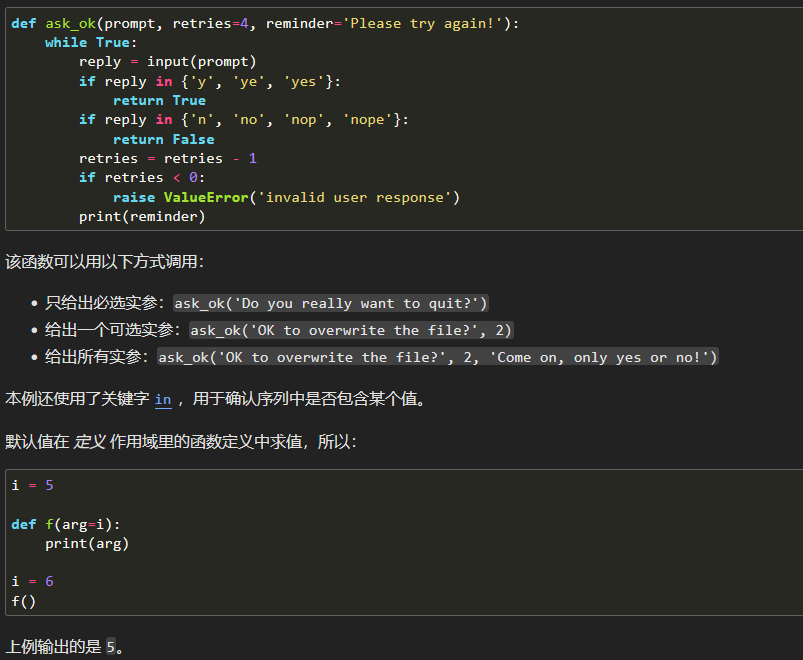


## Match语句（组合形式待查）

类似switch语句；注意case \_:匹配其他所有，类似default。



## 函数——默认值 包含位置和关键字两个方式传值





## 列表函数





## List[] 元组() 集合{} dictionary （字典，类似于map结构）

List[]

元组，

集合{} 没有重复，重复的会被除去，用于逻辑运算。



## 字典 类似map数据结构



循环



还有其他的set等，后续。

## Set 相关

Set空实例的创建必须用set()函数，{}默认是创建字典。

## 模块

\_\_name\_\_ 预定义变量，可以获取模块名（字符串）

Import module\_name



**修改默认的模块搜索路径；**

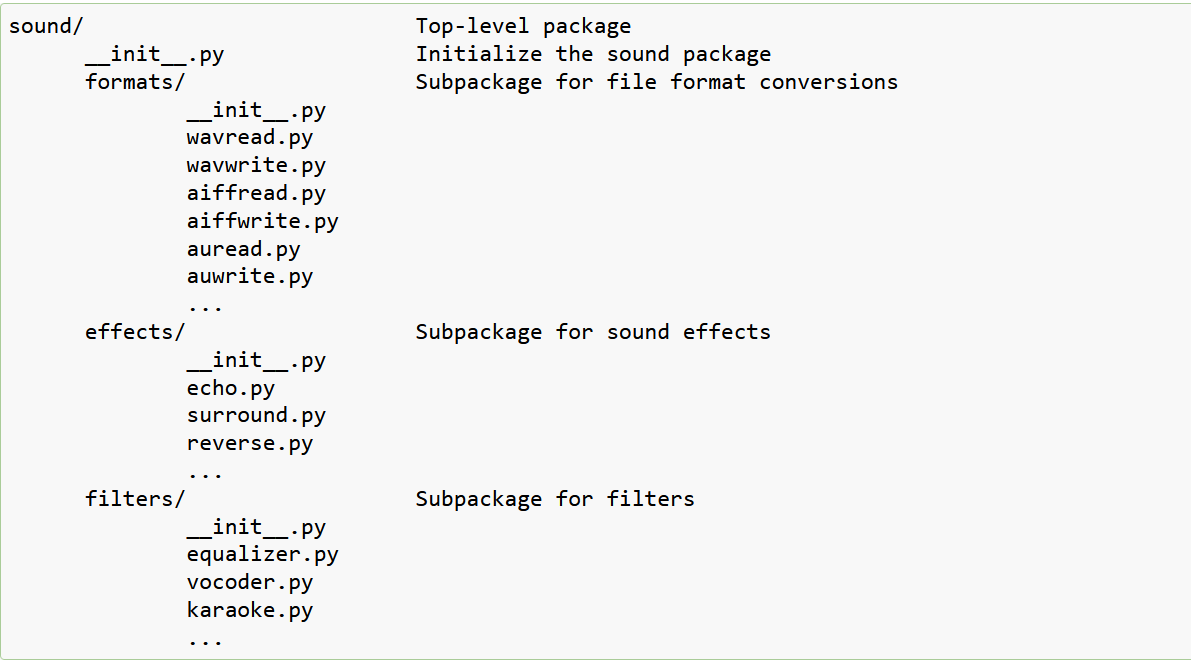


## Dir函数 dir(module\_name) or dir()

## 包

包含init\_py的两级路径，用于区分一个类型的合集

如下：



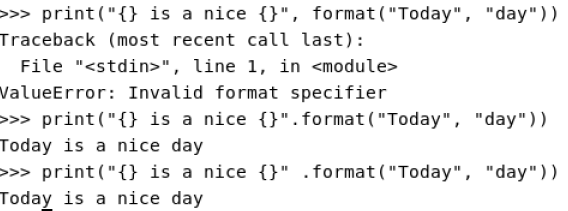
引用方式：区别就是从不同的层级引入：



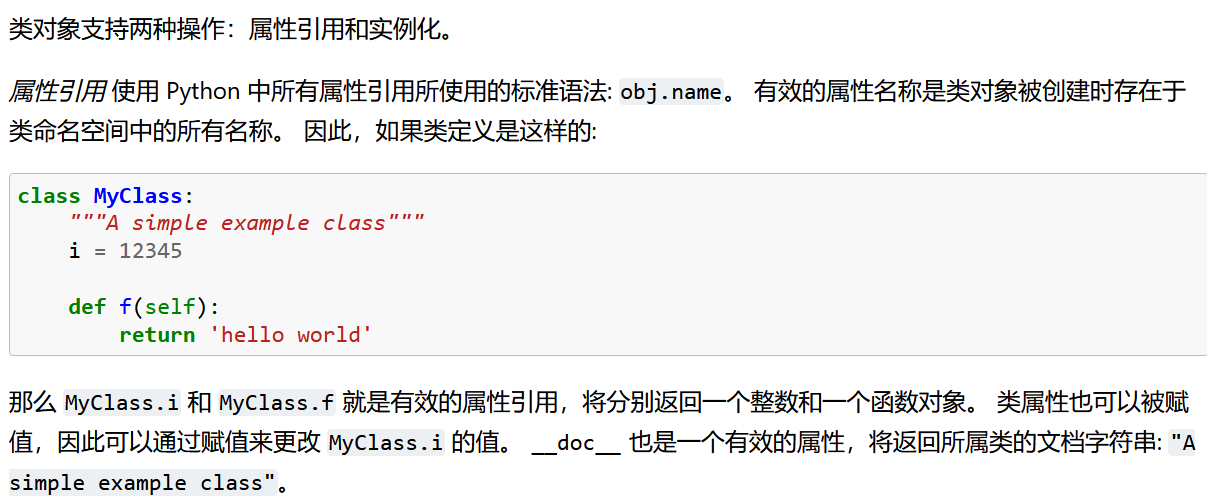
推荐以上方法显式引入，不推荐from \* import \* 形式，关于\_\_all\_\_变量等定义相关。

## 格式化输出

{0}{1},Format（val1,val2）



## Class定义及对象



注意\_\_DOC\_\_

对象：可自定义init函数，类似于构造函数。

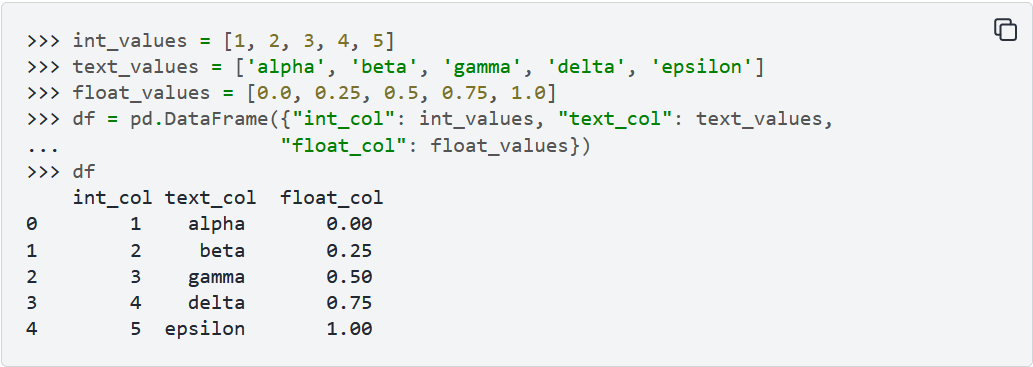
## Class \_\_new\_\_ \_\_init\_\_ singleton 模式



## Pandas\_data frame

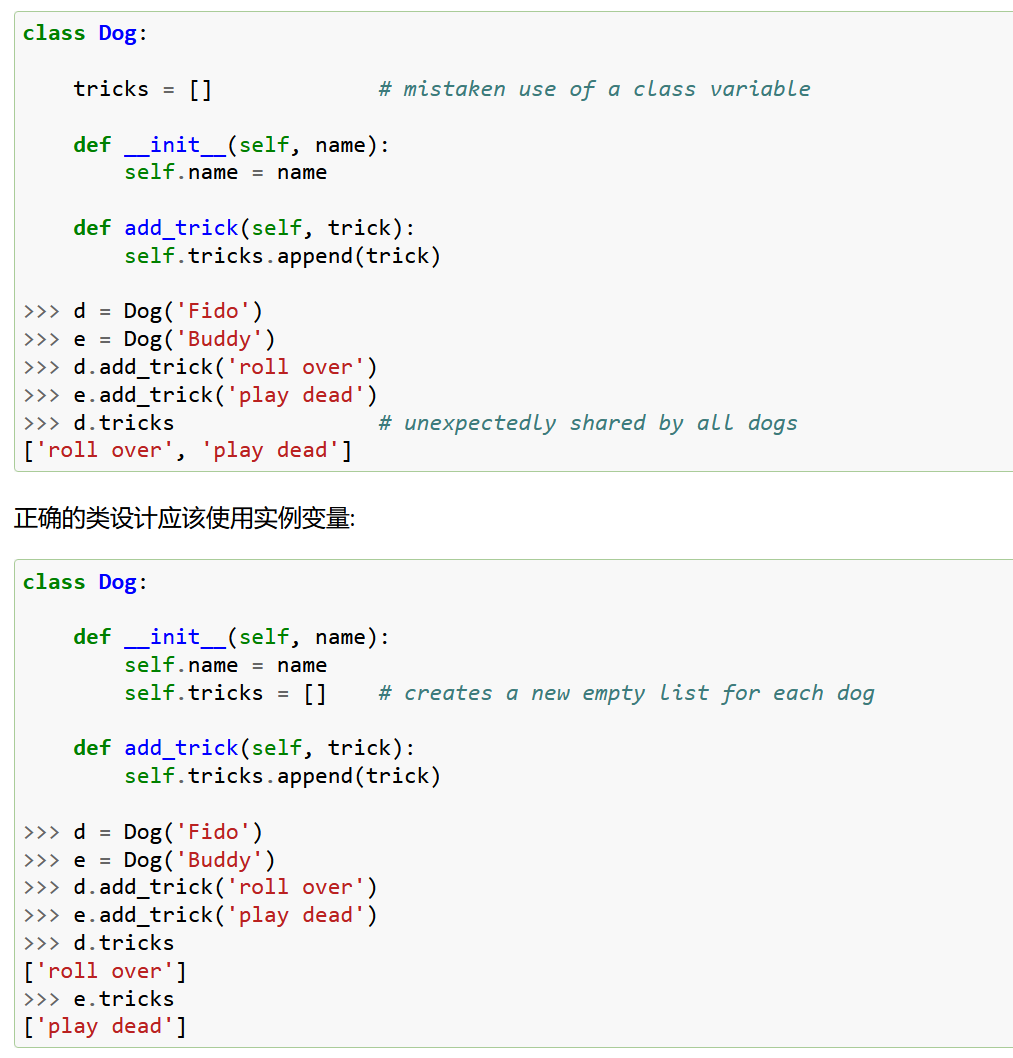
https://pandas.pydata.org

Pandas数据帧，其实就是多列数据；

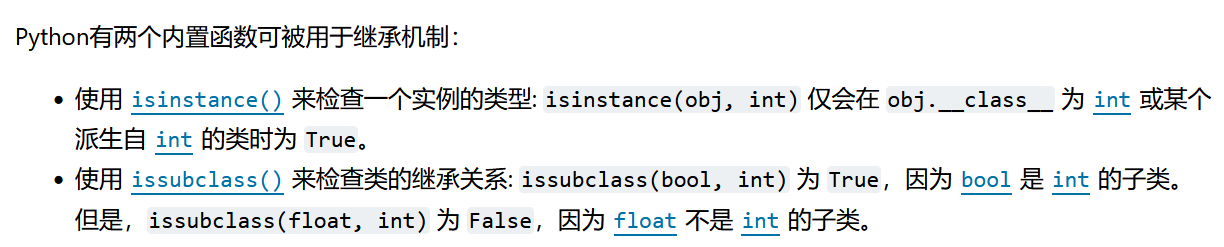




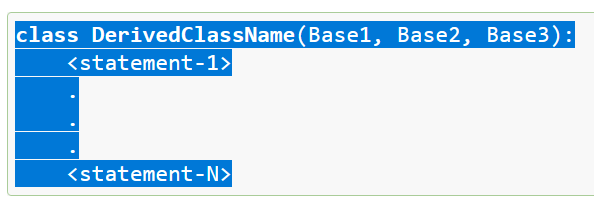
私有变量：this代表的是对象的指针（可以这么认为）



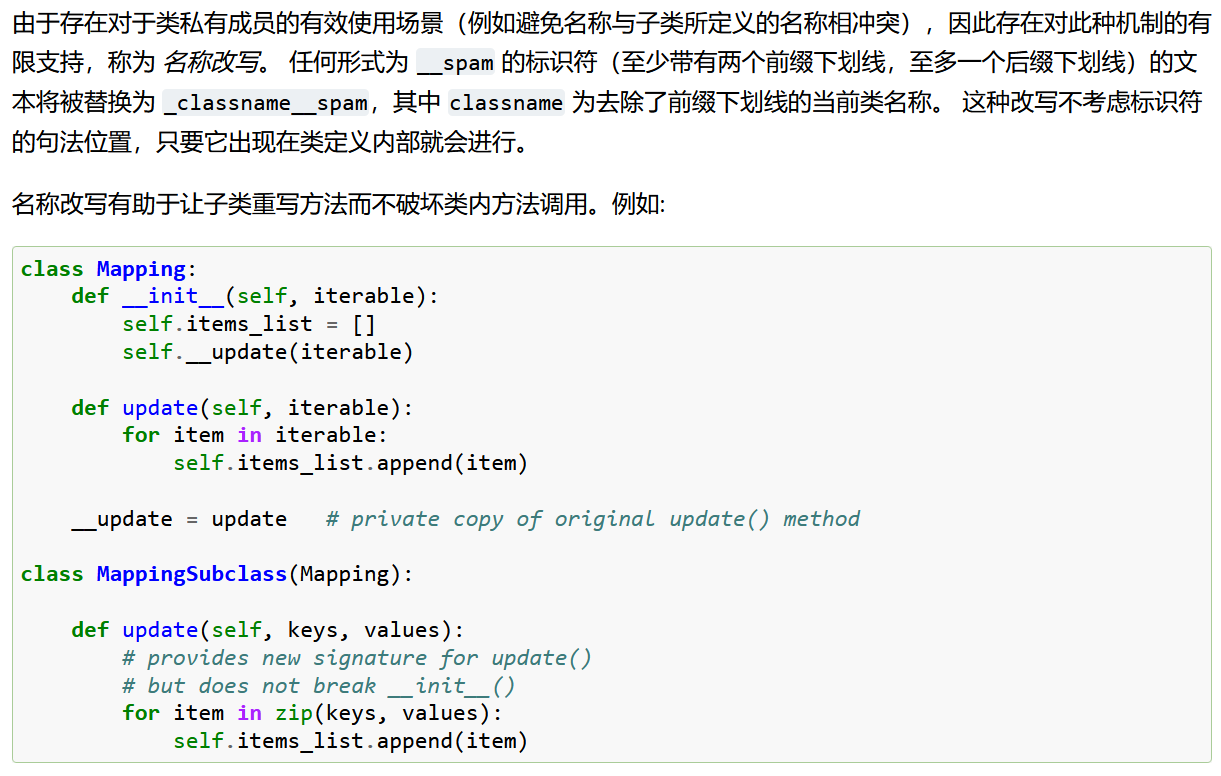
继承：



多重继承：



私有变量，采用改名方法来实现的，所以：



## Python中global 和 nonlocal

Global用在局部声明一个变量为全局变量；

Nonlocal用在嵌套函数内部，用于声明该变量为函数外部定义的变量，该变量为外部函数所定义。

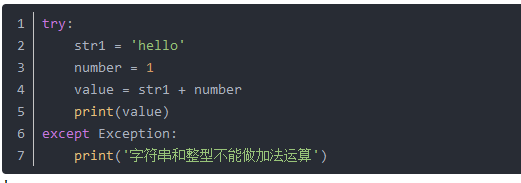
## 关于python中引用上级包的方法

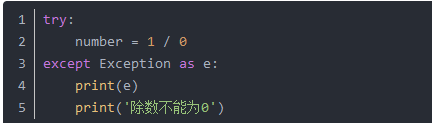


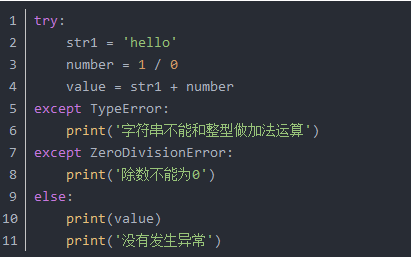
## Python \_\_init\_\_.py作用

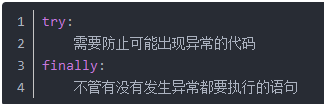
实际上，如果目录中包含了 \_\_init\_\_.py 时，当用 import 导入该目录时，会执行 \_\_init\_\_.py 里面的代码。

## Python 异常捕获（try except finally）









## Python unittest

四、unittest框架主要工作

1、测试发现

从多个文件中收集并加载测试执行用例

2、测试执行

将测试用例按照一定的顺序和条件去执行

3、测试判断

用断言去判断测试用例的执行结果（实际结果）与预期结果是否一致

4、测试报告

统计测试测试进度，通过个数，失败个数等，形成测试报告

unittest 是 Python 内置的测试框架，它基于 Java 的 JUnit 和 Smalltalk 的 SUnit 测试框架的思想设计。unittest 提供了一组类和方法，用于定义测试用例、测试套件、测试运行器等，使得编写和管理测试代码变得更加结构化和模块化。

测试用例

在 unittest 中，测试用例是测试的基本单位。每个测试用例都是一个独立的方法，用于测试特定的功能或行为。测试用例通常继承自 unittest.TestCase 类，并使用断言方法（如 assertEqual、assertTrue 等）来验证预期结果。

测试套件

测试套件是一组相关的测试用例的集合。unittest 提供了 TestSuite 类，用于将多个测试用例或测试套件组合在一起，形成一个更大的测试单元。测试套件可以嵌套，允许创建层次化的测试结构。

测试运行器

测试运行器是负责执行测试用例和测试套件的组件。unittest 提供了 TextTestRunner 类作为默认的测试运行器，它会以文本形式输出测试结果。测试运行器还支持自定义输出格式和行为，可以通过扩展或替换测试运行器来满足特定的需求。

测试固件

unittest 提供了测试固件（test fixture）的概念，用于在测试之前进行必要的准备工作，以及在测试之后进行清理工作。测试固件通过重写 setUp 和 tearDown 方法来实现，这些方法会在每个测试用例执行前后被自动调用。

断言方法

unittest 提供了丰富的断言方法，用于验证测试结果是否符合预期。常用的断言方法包括 assertEqual、assertNotEqual、assertTrue、assertFalse、assertRaises 等。断言方法可以比较值、检查条件、验证异常等，并在断言失败时抛出 AssertionError 异常。

测试发现

unittest 支持自动发现测试用例。通过使用 unittest.defaultTestLoader.discover 方法，可以递归查找指定目录下的所有测试模块，并自动加载其中的测试用例。这样可以避免手动维护测试套件，提高测试的可维护性。

测试报告

unittest 可以生成测试报告，用于展示测试执行的结果和统计信息。默认情况下，测试报告以文本格式输出到控制台。但是，可以通过扩展或集成其他工具（如 HTMLTestRunner）来生成更丰富的测试报告，如 HTML 格式的报告。

集成与扩展

unittest 可以与其他工具和框架集成，以实现更高级的测试功能。例如，可以与 mock 库集成进行模拟和打桩，与 coverage 库集成进行代码覆盖率统计，与持续集成工具（如 Jenkins、Travis CI）集成进行自动化测试等。

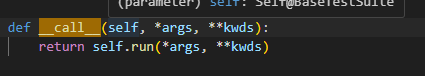
## Python \_\_init\_\_调用

另：当suclass没有定义\_\_init\_\_函数时候，默认会调用基类的\_\_init\_\_;

## Python的 魔术方法 \_\_call\_\_

只要一个类中有\_\_call\_\_方法，那么这个对象就可以用instance()形式来调用。





## Python enumerate()函数

功能：讲一个可迭代对象作为输入，输出枚举对象；

## Python –type --instance

## Iter()内建函数 执行迭代器操作

my\_list = [1, 2, 3, 4, 5]

my\_iter = iter(my\_list)

print(next(my\_iter)) # 输出: 1

print(next(my\_iter)) # 输出: 2

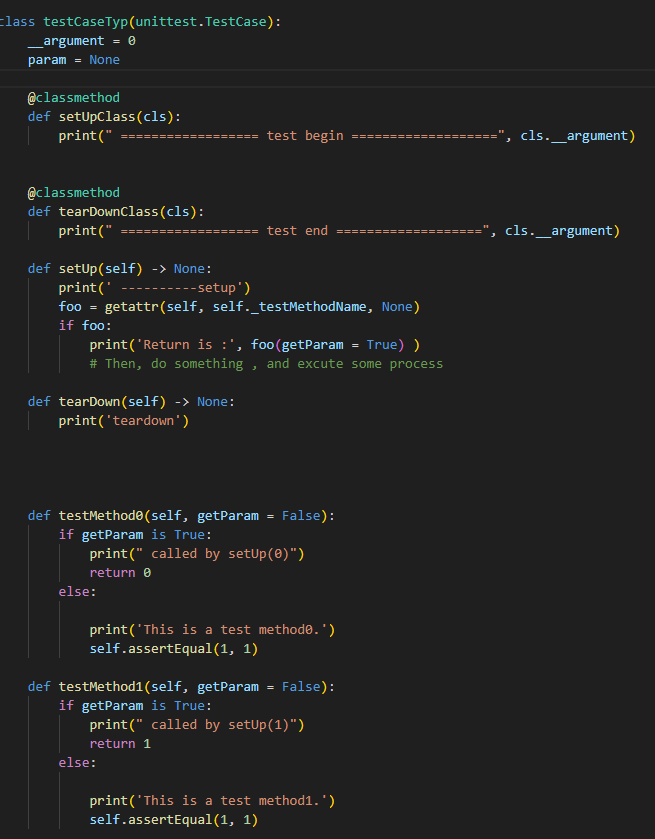
print(next(my\_iter)) # 输出: 3

## Getattr（object（not a class），“name”， default\_return）

当获取的对象不存在时候，返回 default\_return ，这个是自定义的。

## Raise主动增加一个异常

## Unittest通过为method添加choice方式实现参数化测试



## Python os.chmod(file\_name, mode)八进制

Mode:



## 两个属性相关修饰符

@property 表示把方法当做属性使用, 表示当获取属性时会执行下面修饰的方法

@方法名.setter 表示把方法当做属性使用,表示当设置属性时会执行下面修饰的方法

装饰器方式的property属性修饰的方法名一定要和属性名一样。



# \*\*\*\*\*\*\*\*\*\* Tools \*\*\*\*\*\*\*\*\*

## Gerrit相关

git commit --review username1,username2 //添加reviewer

###### Gerrit代码审查工具提交代码

git push origin HEAD:refs/for/master

## Git 重设URL



## Git reset 的恢复操作

Git reset –hard ORG\_HEAD

## Linux socat 虚拟串口命令（注意，前后没有空格）



