

---

# Knowledge

Qitas

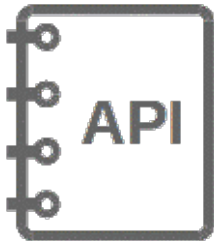

2023 年 02 月 16 日



1	软件开发	3
2	工具依赖	41
3	硬件基础	59
4	系统依赖	63
5	项目管理	75
6	文件管理	77
7	质量管理	79
8	人工智能	81



## Experiences Sharing

		
软件开发	实用工具	系统依赖



# CHAPTER 1

## 软件开发

- 编程语言
- 算法实现
- 开发策略

## 1.1 编程语言

### 1.1.1 编译语言

- *C*
- *C++*
- *Summary*

## C

C 语言是一种通用的、面向过程式的计算机程序设计语言。1972 年，为了移植与开发 UNIX 操作系统，丹尼斯·里奇在贝尔电话实验室设计开发了 C 语言。

C 语言最初是用于系统开发工作，特别是组成操作系统的程序。由于 C 语言所产生的代码运行速度与汇编语言编写的代码运行速度几乎一样，所以采用 C 语言作为系统开发语言。

### 相关概念

#### 数据结构

- 指针
- 数组
- 链表
- 其他

### 指针

#### 指针

指针也就是内存地址，指针变量是用来存放内存地址的变量，在同一 CPU 构架下，不同类型的指针变量所占用的存储单元长度是相同的，而存放数据的变量因数据的类型不同，所占用的存储空间长度也不同。有了指针以后，不仅可以对数据本身，也可以对存储数据的变量地址进行操作。

指针描述了数据在内存中的位置，标示了一个占据存储空间的实体，在这段空间起始位置的相对距离值。在 C/C++ 语言中，指针一般被认为是指针变量，指针变量的内容存储的是其指向的对象的首地址，指向的对象可以是变量（指针变量也是变量），数组，函数等占据存储空间的实体。

- 有 10 个指针的数组，该指针指向一个整数：`int* a[10]`
- 指向有 10 个整型数组的指针：`int (*a)[10]`
- 函数指针 是一个指向函数的指针，该函数有一个整形参数并返回一个整形数：`int (*func)(int)`

- 指针与数组
- 指针与结构体
- 指针与字符



## 指针与数组

```
int main()
{
    int a[5] = { 1, 2, 3, 4, 5 };
    int *ptr = (int *)(&a + 1);    //&a取出整个数组的地址; &a+1跳过一个数组
    //&a的类型为: 数组指针 int(*)[5] 所以要强转
    //a为数组名, 首元素地址, 即为1的地址, +1, 跳过一个元素, 即为2的地址
    printf( "%d,%d", *(a + 1), *(ptr - 1)); // 2 5
    return 0;
}
```

```
int main()
{
    int a[4] = { 1, 2, 3, 4 };
    int *ptr1 = (int *)(&a + 1);
    int *ptr2 = (int *)((int)a + 1);
    printf( "%x,%x", ptr1[-1], *ptr2);    // 4 2000000
    return 0;
}
```

`int ptr1 = (int *)(&a + 1)`: 取出数组的地址+1, 跳过一个数组, 因为 `&a` 的类型为数组指针: `int(*)[4]` 类型不匹配, 所以强转为 `int` 类型;

- `ptr1[-1] ==> *(ptr1 + (-1)) ==> *(ptr1 - 1)`
- `int *ptr2 = (int *)((int)a + 1)`

此时的 `a` 代表的首元素地址, 地址值是一个常量, 整数+1: 相当于跳过一个字节, 注意要考虑小端存放, 读取时倒着读取的问题, 所以 `ptr2` 指向的是 `00 00 00 02` 这四个字节, 所以打印结果为: `02000000`

```
int main()
{
    int a[5][5];
    int (*p)[4];    //p是数组指针, 指向的数组有4个元素
    p = a;
    printf( "%p,%d\n", &p[4][2]-&a[4][2], &p[4][2]-&a[4][2]);    // 指针-
    ↪ 指针得到的是二者之间的元素个数
    return 0;
}
```

- `p[4] = *(p+4)`
- `p[4][2] ==> *((p+4)+2)`
- `&p[4][2]` 为小地址, `&a[4][2]` 为大地址, 小地址减大地址, 所以最后结果为-4

**警告：** a 是二维数组，对应数组指针的类型为：int(\*)[5]，指向的是有 5 个元素的一维数组，而 p 是数组指针，指向的数组只有 4 个元素，所以会有警告，可以写成 int(p)[4] = (int(\*)[4])a 消除警告

```
int main()
{
    int a[3][2] = { (0, 1), (2, 3), (4, 5) }; //逗号表达式-
    ↪ 结果为最后一个表达式的结果，所以只是初始化了{ 1, 3, 5 }
    int *p;
    p = a[0]; //a[0] : ↪
    ↪ 二维数组第一行的数组名，在这里是首元素地址，即第一行第一个元素的地址
    printf("%d ", p[0]); //1
    return 0;
}
```

```
int main()
{
    int aa[2][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int *ptr1 = (int *)(&aa + 1);
    int *ptr2 = (int *)(*(&aa + 1));
    printf(" %d, %d", *(ptr1 - 1), *(ptr2 - 1)); // 5 10
    return 0;
}
```

- &aa: 取出二维数组的地址; &aa+1: 跳过二维数组, & 二维数组应该使用数组指针接收, 现在保存到整形指针, 所以要强转。
- aa: 没有单独放在 sizeof 内部, 没有 & 数组名, 所以代表的是二维数组首元素地址, 即二维数组第一行的地址; aa+1: 跳过一行
- \*(aa+1): 相当于拿到了第二行的数组名, 等价于 aa[1]

## 指针与结构体

**提示：** 指针 +1 的步长取决于指针指向的数据的类型, 整数 +1 -> 跳过一个字节, 执行普通的加减运算, 而整形指针 +1 -> 跳过四个字节

```
struct Test
{
    int Num;
    char *pcName;
```

(续下页)

(接上页)

```

short sDate;
char cha[2];
short sBa[4];
}*p; //这里告知结构体的大小是20个字节，假设p的值为0x100000。
int main()
{
    p = 0x00100000;
    //0x1-->对应的值就是1 相当于0x00000001
    printf("%p\n", p + 0x1);//
    ↪ p为结构体指针，指向一个结构体，+1，跳过一个结构体，即跳过20个字节，
    // 0x00100000+20 -> 0x00100020 错误， _
    ↪ 要将20转化为16进制再加，或者将16进制0x00100000转化为10进制之后加上20，然后再转化为16进制
    // 20 -> 0x00000014
    //所以最终结果为：0x00100014
    printf("%p\n", (unsigned long)p + 0x1);//将p转化为长整形，+1，即为整形+1， _
    ↪ 例如：500+1= 501，
    //所以结果为： 0x00100001
    printf("%p\n", (unsigned int*)p + 0x1);
    //将p强转为无符号整形，+1跳过一个整形->跳过4个字节
    //所以结果为：0x00100004
    return 0;
}

```

## 指针与字符

```

#include <stdio.h>
int main()
{
    char *a[] = {"work", "at", "alibaba"}; //a是数组，元素类型为：char*_
    ↪，存放指向字符串首字符地址，根据后面初始化内容确定数组的大小
    char**pa = a; //char**pa：一颗*说明pa是指针，另一颗*说明pa指向的类型是char*
    pa++; //pa+1:跳过char*
    printf("%s\n", *pa); //打印结果为：at
    return 0;
}

```

```

int main()
{
    char *c[] = {"ENTER", "NEW", "POINT", "FIRST"};
    char**cp[] = {c+3, c+2, c+1, c};
    char***cpp = cp;
    printf("%s\n", **++cpp); //cpp先自增，此时cpp存放了指向存放c+2地址的空间（地址），

```

(续下页)

(接上页)

```

→ 打印结果为：POINT
    printf("%s\n", *--*++cpp+3); // *++cpp 拿到存放 c+1 地址的空间，再--
→ 自减 c+1 的地址，把存放 c+1 的地址变成存放 c 的地址，"ENTER" 首字符后 +3 打印：ER
    printf("%s\n", *cpp[-2]+3); // 打印结果为：ST
    printf("%s\n", cpp[-1][-1]+1); // 打印结果为：EW
    return 0;
}

```

- `cpp-2`: 从指向存放 `c` 地址空间又变为了指向存放 `c+3` 地址的空间
- `*(cpp-2)`: 得到 `cpp` 现在指向的内容，即 `c+3` 的地址
- `** (cpp-2)`: 得到 `c+3` 空间的内容（首字符 `F` 的地址）
- `** (cpp-2)+3`: 从首字符 `F` 的地址向后 +3，即为 `S` 的地址

`cpp[-1] ==> *(cpp-1); cpp[-1][-1] ==> ** (cpp-1)-1; cpp[-1][-1] +1 ==> ** (cpp-1)-1 +1`

此时的 `cpp` 指向为第二条表达式之后的状态，`cpp` 存放 `c` 的地址，`cpp-1` 指向存放 `c+2` 地址的空间，`*(cpp-1)-1` 自减变成了 `c+1` 的地址，即得到了 `c+1` 的地址，再 +1 字符输出。

## 数组

### 零长数组

今天在看代码中遇到一个结构中包含 `char data[0]`，第一次见到时感觉很奇怪，数组的长度怎么可以为零呢？于是上网搜索一下这样的用法的目的，发现在 `linux` 内核中，结构体中经常用到 `data[0]`。这样设计的目的是让数组长度是可变的，根据需要进行分配。方便操作，节省空间。

```

struct buffer
{
    int data_len;    // 长度
    char data[0];    // 起始地址
};

```

在这个结构中，`data` 是一个数组名；但该数组没有元素；该数组的真实地址紧随结构体 `buffer` 之后，而这个地址就是结构体后面数据的地址（如果给这个结构体分配的内容大于这个结构体实际大小，后面多余的部分就是这个 `data` 的内容）；这种声明方法可以巧妙的实现 C 语言里的数组扩展。

## 对比指针

从结果可以看出 `data[0]` 和 `data[]` 不占用空间，且地址紧跟在结构后面，而 `char *data` 作为指针，占用 4 个字节，地址不在结构之后。

在实际程序中，数据的长度很多是未知的，这样通过变长的数组可以方便的节省空间。对指针操作，方便数据类型的转换。

采用 `char *data`，需要进行二次分配，操作比较麻烦，很容易造成内存泄漏。而直接采用变长的数组，只需要分配一次，然后进行取值即可以。

## 链表

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。

相比于线性表顺序结构，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到  $O(1)$  的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要  $O(n)$  的时间，而线性表和顺序表相应的时间复杂度分别是  $O(\log n)$  和  $O(1)$ 。

使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

链表最明显的好处就是，常规数组排列关联项目的方式可能不同于这些数据项目在记忆体或磁盘上顺序，数据的存取往往要在不同的排列顺序中转换。

链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。

- 双向链表其实是单链表的改进。在双向链表中，结点除含有数据域外，还有两个链域，一个存储直接后继结点地址，一般称之为右链域；一个存储直接前驱结点地址，一般称之为左链域。
- 循环链表是与单链表一样，是一种链式的存储结构，所不同的是，循环链表的最后一个结点的指针是指向该循环链表的第一个结点或者表头结点，从而构成一个环形的链。

### 链表

- 单向链表
- 双向链表

### 单向链表

```
typedef int SLTDataType;
typedef struct SListNode
{
    SLTDataType data;
    struct SListNode* next;
}SListNode;
```

无头 + 单向 + 非循环链表增删查改实现

```
SListNode* BuySListNode(SLTDataType x)
{
    SListNode* tmp = (SListNode*)malloc(sizeof(SListNode));
    if (tmp == NULL){
        printf("无法给节点开辟空间\n");
        return NULL;
    }
    else{
        tmp->data = x;
        tmp->next = NULL;
        return tmp;
    }
}
```

单链表尾插入和删除

```
void SListPushBack(SListNode** pplist, SLTDataType x)
{
    SListNode* newnode = BuySListNode(x);
    if (*pplist == NULL){
        *pplist = newnode;
    }
    else{
        SListNode* tail = *pplist;
```

(续下页)

(接上页)

```

        while (tail->next != NULL){
            tail = tail->next;
        }
        tail->next = newnode;
    }
}

void SListPopBack(SListNode** pplist)
{
    assert(*pplist);
    SListNode* cur = *pplist;
    SListNode* prev = NULL;
    if (cur->next == NULL){
        free(cur);
        *pplist = NULL;
    }
    else{
        while (cur->next != NULL){
            prev = cur;
            cur = cur->next;
        }
        free(cur);
        prev->next = NULL;
    }
}

```

### 单链表打印输出和查找

```

void SListPrint(SListNode* plist)
{
    SListNode* head = plist;
    while (head != NULL){
        printf("%d ", head->data);
        head = head->next;
    }
}

SListNode* SListFind(SListNode* plist, SLTDateType x)
{
    assert(plist);
    while (plist != NULL)
    {
        if (plist->data == x)
        {
            return plist;
        }
    }
}

```

(续下页)

(接上页)

```
    plist = plist->next;
}
return NULL;
}
```

### 双向链表

```
typedef int LTDataType;
typedef struct ListNode
{
    ListDateType val;
    struct ListNode* prev;
    struct ListNode* next;
}ListNode;
```

创建返回链表的头结点

```
ListNode* BuyList(ListDateType x)
{
    ListNode* newnode = (ListNode*)malloc(sizeof(ListNode));
    if (newnode == NULL){
        printf("BuyList fail\n");
        exit(-1);
    }
    newnode->val = x;
    newnode->next = NULL;
    newnode->prev = NULL;
    return newnode;
}
```

在双向链表尾插入和删除

```
void ListPushBack(ListNode* phead, ListDateType x)
{
    assert(phead);
    ListNode* newnode = BuyList(x);
    ListNode* tail = phead->prev;
    tail->next = newnode;
    phead->prev = newnode;
    newnode->next = phead;
    newnode->prev = tail;
}
```

(续下页)



(接上页)

```

void ListPopBack(ListNode* phead)
{
    assert(phead->next != phead);
    ListNode* tail = phead->prev;
    ListNode* prev = tail->prev;
    phead->prev = prev;
    prev->next = phead;
    free(tail);
    tail = NULL;
}

```

在双向链表头插入和删除

```

void ListPushFront(ListNode* phead, ListDateType x)
{
    assert(phead);
    ListNode* newnode = BuyList(x);
    ListNode* head = phead->next;
    phead->next = newnode;
    head->prev = newnode;
    newnode->next = head;
    newnode->prev = phead;
}

void ListPopFront(ListNode* phead)
{
    assert(phead);
    assert(phead->next != phead);
    ListNode* head = phead->next;
    ListNode* next = head->next;
    phead->next = next;
    next->prev = phead;
    free(head);
    head = NULL;
}

```

在 pos 之前插入和删除

```

void ListInsert(ListNode* pos, ListDateType x)
{
    assert(pos);
    ListNode* newnode = BuyList(x);
    ListNode* prev = pos->prev;
    prev->next = newnode;
    pos->prev = newnode;
}

```

(续下页)

(接上页)

```
newnode->prev = prev;
newnode->next = pos;
}
void ListErase(ListNode* pos)
{
    assert(pos);
    ListNode* prev = pos->prev;
    ListNode* next = pos->next;
    prev->next = next;
    next->prev = prev;
    free(pos);
    pos = NULL;
}
```

## 其他

- 堆栈
- 队列
- 树
- 图
- 字典树（这是一种高效的树形结构，但值得单独说明）
- 散列表（哈希表）

## 堆栈

heap stack

堆栈都是一种数据项按序排列的[数据结构](#)，只能在一端（称为栈顶 (top)）对数据项进行插入和删除。在单片机应用中，堆栈是个特殊的存储区，主要功能是暂时存放数据和地址，通常用来保护断点和现场。

- 堆 heap：顺序随意，一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收，分配方式倒是类似于链表。
- 栈 stack：后进先出 (Last-In/First-Out)，由操作系统自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

stack 的空间由操作系统自动分配和释放，heap 的空间是手动申请和释放的，heap 常用 new 关键字来分配。通常，stack 空间有限，heap 的空间是很大的自由区。

- 堆栈对比
- 堆栈区别
- 内存空间
- 初始化

## 堆栈对比

栈区由编译器自动分配，内纯的分配是连续的，堆区由程序员自行分配，需由程序员释放变量内存。

从申请方式，申请大小，申请效率简单比较：Stack 的空间由操作系统自动分配/释放，Heap 上的空间手动分配/释放。Stack 空间有限，Heap 是很大的自由存储区。Stack 申请效率高，Heap 申请效率低。

堆是大家共有的空间，分全局堆和局部堆。全局堆就是所有没有分配的空间，局部堆就是用户分配的空间。堆在操作系统对进程初始化的时候分配，运行过程中也可以向系统要额外的堆，但是记得用完了要还给操作系统，要不然就是内存泄漏。栈是线程独有的，保存其运行状态和局部自动变量的。栈在线程开始的时候初始化，每个线程的栈互相独立。每个函数都有自己的栈，栈被用来在函数之间传递参数。操作系统在切换线程的时候会自动的切换栈，就是切换 SS/ESP 寄存器。

栈空间不需要在高级语言里面显式的分配和释放。

C 语言程序编译的内存分配，堆与栈的区别：

- 栈是由编译器自动分配释放，存放函数的参数值、局部变量的值等。操作方式类似于数据结构中的栈。
- 堆一般由程序员分配释放，若不释放，程序结束时可能由 OS 回收。注意这里说是可能，并非一定。

栈区 (stack)：

//windows 下，栈内存分配 2M（确定的常数），超出了限制，提示 stack overflow 错误 //编译器自动分配释放，主要存放函数的参数值，局部变量值等；堆区 (heap)：程序员手动分配释放，操作系统 80% 内存全局区或静态区：存放全局变量和静态变量；程序结束时由系统释放，分为全局初始化区和全局未初始化区；字符常量区：常量字符串放与此，程序结束时由系统释放；程序代码区：存放函数体的二进制代码。

1. 就算没有 free()，main() 结束后也是会自动释放 malloc() 的内存的，这里监控者是操作系统，设计严谨的操作系统会登记每一块给每一个应用程序分配的内存，这使得它能够在应用程序本身失控的情况下仍然做到有效地回收内存。你可以试一下在 TaskManager 里强行结束你的程序，这样显然是没有执行程序自身的 free() 操作的，但内存并没有发生泄漏。
2. free() 的用处在于实时回收内存。如果你的程序很简单，那么你不写 free() 也没关系，在你的程序结束之前你不会用掉很多内存，不会降低系统性能；而你的程序结束之后，操作系统会替你完成这个工作。但你开始开发大型程序之后就会发现，不写 free() 的后果是很严重的。很可能你在程序中要重复 10k 次分配 10M 的内存，如果每次使用完内存后都用 free() 释放，你的程序只需要占用 10M 内存就能运行；但如果你不用 free()，那么你的程序结束之前就会吃掉 100G 的内存。这其中当然包括绝大部分的虚拟内存，而由于虚拟内存的操作是要读写磁盘，因此极大地影响系统的性能。你的系统很可能因此而崩溃。

3. 任何时候都为每一个 `malloc()` 写一个对应的 `free()` 是一个良好的编程习惯。这不但体现在处理大程序时的必要性上,更体现在程序的优良的风格和健壮性上。毕竟只有你自己的程序知道你为哪些操作分配了哪些内存以及什么时候不再需要这些内存。因此,这些内存当然最好由你自己的程序来回收。

### 堆栈区别

- (1) 申请方式
- (2) 操作系统的相应
- (3) 申请的大小限制
- (4) 申请速度
- (5) 堆和栈的存储内容

堆区的头部用一个字节存放堆区的大小,其他的内容由程序员自己安排;栈区在函数调用子函数的时候,首先进栈的是函数调用语句的下一条可执行语句的地址,然后是函数的各个参数进栈,在大多数 C 编译器中,函数的参数是从右向左一次进栈,接下来是局部变量进栈。当本次函数执行结束时候,首先出栈的是局部变量,其次是函数参数,最后是栈顶指向的可执行语句的地址。

Q: 局部变量能否和全局变量重名?

A: 能,局部会屏蔽全局。要用全局变量,需要使用“:” 局部变量可以与全局变量同名,在函数内引用这个变量时,会用到同名的局部变量,而不会用到全局变量。对于有些编译器而言,在同一个函数内可以定义多个同名的局部变量,比如在两个循环体内都定义一个同名的局部变量,而那个局部变量的作用域就在那个循环体内

### 内存空间

对于一个 C 语言程序而言,内存空间主要由五个部分组成:代码段 (.text)、数据段 (.data)、静态区 (.BSS)、堆和栈组成。

- BSS 段: BSS 段 (bss segment) 通常是指用来存放程序中未初始化的全局变量和静态变量 (这里注意一个问题:一般的书上都会说全局变量和静态变量是会自动初始化的,那么哪来的未初始化的变量呢?变量的初始化可以分为显示初始化和隐式初始化,全局变量和静态变量如果程序员自己不初始化的话,的确也会被初始化,那就是不管什么类型都初始化为 0,这种没有显示初始化的就是我们这里所说的未初始化。既然都是 0 那么就没必要把每个 0 都存储起来,从而节省磁盘空间,这是 BSS 的主要作用) 的一块内存区域。BSS 是英文 Block Started by Symbol 的简称。BSS 段属于静态内存分配。BSS 节不包含任何数据,只是简单的维护开始和结束的地址,即总大小,以便内存区能在运行时分配并被有效地清零。BSS 节在应用程序的二进制映像文件中并不存在,即不占用磁盘空间而只在运行的时候占用内存空间,所以如果全局变量和静态变量未初始化那么其可执行文件要小很多。
- 数据段: 数据段 (data segment) 通常是指用来存放程序中已初始化的全局变量和静态变量的一块内存区域。数据段属于静态内存分配,可以分为只读数据段和读写数据段。字符串常量等,但一般都是放在只读数据段中。

- 代码段：代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等，但一般都是放在只读数据段中。
- 栈区：由系统自动分配，栈区的分配运算内置于处理器的指令集，当函数执行结束时由系统自动释放。存放局部变量。栈的缺点是：容量有限，当相应的区间被释放时，局部变量不可再使用。查询栈容量的命令：`ulimits -s`。栈是一块连续的区域，向高地址扩展，栈顶和容量是事先约定好的。
- 堆区：在程序的执行过程中才能分配，由程序员决定，编译器在编译时无法为他们分配空间，只有在程序运行时分配，所以被称为动态分配。堆是不连续的区域，向高地址扩展。由于系统用链表来描述空闲的地址空间，链表的遍历是由低地址向高地址的，故堆区是不连续的动态的存储空间。

## 初始化

全局变量在 `main` 函数第一次使用之前已经初始化，初始化可细分为：编译时初始化和加载时初始化，即 `static initialization` 和 `dynamic initialization`。

静态初始化先于动态初始化。因为静态初始化发生在编译时期，直接写进 `.bss` 段和 `.data` 段，在程序执行时直接加载；而动态初始化则是在运行时期，由运行时库调用相应构造函数进行初始化，同样要写进 `.bss` 段或 `.data` 段。

## 编译时初始化

编译时初始化是针对于那些简单的、`c++` 内部定义的数据结构（也称内置结构），如 `int/double/bool` 及数组的初始化，又可分为两种方式：

- `.bss` 段：未初始化的变量，也就是我们没指定初值，编译器分配 0 值，编译时编译器将其分配在 `.bss` 段，不占用 `rom` 空间
- `.data` 段：已初始化好的全局变量和静态变量，也就是我们指定了初值，编译器将其分配在 `.data` 段，占用 `rom` 空间

---

**提示：** `bss` 段不占用 `rom` 空间，但是在内核加载到内存时，会保留相应的空间；在有些编译器中，初始化为 0 的静态变量和全局变量也放在 `.bss` 段

---

### 加载时初始化

全局类对象在 `main` 函数执行前，由加载程序完成其初始化，其无法在编译期初始化，由于那时候还无法调用类的构造函数。

同时，在加载期，是线程安全的。例如，饿汉方式的单例类：借助 `main` 执行前的加载期完成初始化，由于还在加载所以确保线程安全。

另外针对静态变量：`ref:lan_c_static`，若其是普通的具有本文可见性的普通静态变量，其可能在编译期（内置类型）初始化或者在加载期（类的静态成员）初始化。但针对函数内部的局部 `static` 变量，其在第一次被调用时初始化，并且只初始化一次。

### 函数

函数声明：

- 1. 告诉编译器有一个函数叫什么，参数是什么，返回类型是什么，但是具体是不是存在，并不重要。
- 2. 函数的声明一般出现在函数的使用之前，要满足先声明后使用。
- 3. 函数的声明一般要放在头文件之中。（后缀为.h）

函数定义：函数的定义是指函数的具体实现，交代函数的功能实现。

- 函数与指针
- 函数的递归
- 标准库引用

### 函数与指针

#### 函数指针

函数指针，其本质是一个指针变量，该指针指向这个函数。

- 声明形式：`type (*func)(参数列表)`

如果在程序中定义了一个函数，那么在编译时系统就会为这个函数代码分配一段存储空间，这段存储空间的首地址称为这个函数的地址。而且函数名表示的就是这个地址。既然是地址我们就可以定义一个指针变量来存放，这个指针变量就叫作函数指针变量，简称函数指针。

```
int (*p)(int, int);
```

这个语句就定义了一个指向函数的指针变量 `p`。首先它是一个指针变量，所以要有一个 “\*”，即 `(p)`；其次前面的 `int` 表示这个指针变量可以指向返回值类型为 `int` 型的函数；后面括号中的两个 `int` 表示这个指针变量

可以指向有两个参数且都是 *int* 型的函数。所以合起来这个语句的意思就是：定义了一个指针变量 *p*，该指针变量可以指向返回值类型为 *int* 型，且有两个整型参数的函数。*p* 的类型为 *int()*(*int*, *int*)。

指向函数指针数组的指针声明形式：type (\*(\*func)[])(参数列表)

其中 (\* pfunarr2)[3] 表示数组指针，而 void(\*)() 表示函数指针，两者结合起来就是指向函数指针数组的指针。

```
x = (*fun)();
x = fun();
```

两种调用函数的方式，建议使用第一种，因为可以清楚的指明这是通过指针的方式来调用函数。

## 指针函数

指针函数就是返回指针值的函数，本质是一个函数。

- 声明形式：type \*func (参数列表)

```
int *fun(int x,int y);
```

指针函数等价于“返回值为指针的函数”。从上面的定义形式可以看出，函数指针和指针函数的直观上区别在于指针符号 \* 与函数名/指针名有没有用括号 () 包裹起来，从这一点来看是很容易区分两者的。

**备注：**函数名带括号的就函数指针，否则就是指针函数。

## 函数的递归

程序调用自身的编程技巧称为递归 (recursion)，递归作为一种算法在程序设计语言中广泛应用。一个过程或函数在其定义或说明中有直接或间接调用自身的一种方法，它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解，递归策略只需要少量的程序就可描述出解题过程所需要的多次重复运算，大大减少了程序的代码量

递归的主要思考方式在于：化繁为简。递归的两个必要条件 \* 1. 存在限制条件，当满足这个限制条件的时候，递归便不再继续 \* 2. 每次递归调用之后越来越接近这个限制条件。

## 标准库引用

### stdio

standard input & output

标准输入输出头文件 `stdio.h` 头定义了三个变量的类型，几个宏及各种功能进行输入和输出。



```
#include <stdio.h>
int main() {
    char ch = -1;
    printf(" %x,%02x", ch, (unsigned char)ch);
    return (0);
}
```

- 输出答案: ffffffff,ff

%x 默认输出 unsigned int; 所以 char 会被自动扩展至 unsigned int; 因此会扩展符号位; 而 unsigned char 扩展至 unsigned int; 会直接用 0 填充;

```
double x = 218.82631;
printf("%-6.2e\n", x);
```

%: 表示格式说明的起始符号, 也是转义符号, 有一题 printf (“%%”) 输出几个? 答案输出%% 两个 -: 有-表示左对齐输出, 如省略表示右对齐输出 0: 有 0 表示指定空位填 0, 如省略表示指定空位不填

```
m.n
m指域宽, 即对应的输出项在输出设备上所占的字符数。
n指精度, 用于说明输出的实型数的小数位, 没有指定n时, 隐含的精度为n=6位。
e格式表示以指数形式输出实数
```

**\*\* 以左对齐、指数形式、总长度 m=6、小数 n=2 两位输出 \*\***

## 编译原理

- 1 编译器是一种翻译程序, 它用于将源语言 (即用某种程序设计语言写成的) 程序翻译为目标语言 (即用二进制数表示的伪机器代码写成的) 程序。后者在 windows 操作系统平台下, 其文件的扩展名通常为.obj。该文件通常还要经过进一步的连接, 生成可执行文件 (机器代码写成的程序, 文件扩展名为.exe)。通常有两种方式进行这种翻译, 一种是编译, 另一种是解释。后者并不生成可执行文件, 只是翻译一条语句、执行一条语句。这两种方式相编译比解释运行的速度要快得多。
- 2 编译过程的 5 个阶段: 词法分析; 语法分析; 语义分析与中间代码产生; 优化; 目标代码生成。
- 3 在这五个阶段中, 词法分析的任务是识别源程序中的单词是否有误, 编译程序中实现这种功能的部分一般称为词法分析器。在编译器中, 词法分析器通常仅作为语法分析程序的一个子程序以便在它需要单词符号时调用。在这一编译阶段中发现的源程序错误, 称为词法错误。
- 4 语法分析阶段的目的是识别出源程序的语法结构 (即语句或句子) 是否错误, 所以有时又常为句子分析。编译程序中负责这一功能的程序称为语法分析器或语法分析程序。在这一阶段中发现的错误称为语法错误。
- 5 C 语言的 (源) 程序必须经过编译才能生成目标代码, 再经过链接才能运行。PASCAL 语言、FORTRAN 语言的源程序也要经过这样的过程。通常将 C、PASCAL、FORTRAN 这样的语言统称为高级语言。而将最终的可执行程序称为机器语言程序。



- 6 在编译 C 语言程序的过程中，发现源程序中的一个标识符过长，超过了编译程序允许的范围，这个错误应在词法分析阶段发现，这种错误通常被称作词法错误。

## 编译过程

### 预处理，编译，组装，链接

- 预处理：gcc -E project.c -o project.i //宏展开，宏替换
- 编译：gcc -S project.i -o project.s //将目标文件编译成汇编文件
- 汇编：gcc -c project.s -o project.o //汇编成二进制文件
- 链接：gcc project.o -o project //加载库文件，生成可执行文件

---

**备注：** 组装才是平台相关的，之前的操作都与平台无关

---

编译程序的工作过程一般也可以划分为五个阶段：词法分析、语法分析、语义分析与中间代码产生、优化、目标代码生成。

## ASSERT

ASSERT() 是一个调试程序时经常使用的宏，在程序运行时它计算括号内的表达式，如果表达式为 FALSE (0)，程序将报告错误，并终止执行。如果表达式不为 0，则继续执行后面的语句。

ASSERT 只有在 Debug 版本中才有效，如果编译为 Release 版本则被忽略。

## 编程基础

### inline

- 定义
- 比较

## 定义

`inline` 关键字用来定义一个类的内联函数，引入它的主要原因是用它替代 C 中表达式形式的宏定义。

- 1. C 中使用 `define` 这种形式宏定义的原因是因为 C 语言是一个效率很高的语言，这种宏定义在形式及使用上像一个函数，但它使用预处理器实现，没有了参数压栈，代码生成等一系列的操作。因此，效率很高，这是它在 C 中被使用的一个主要原因。
- 2. 这种宏定义在形式上类似于一个函数，但在使用它时，仅仅只是做预处理器符号表中的简单替换，因此它不能进行参数有效性的检测，也就不能享受 C++ 编译器严格类型检查的好处，另外它的返回值也不能被强制转换为可转换的合适的类型。这样，它的使用就存在着一系列的隐患和局限性。
- 3. 在 C++ 中引入了类及类的访问控制，这样，如果一个操作或者说一个表达式涉及到类的保护成员或私有成员，你就不可能使用这种宏定义来实现（因为无法将 `this` 指针放在合适的位置）。
- 4. `inline` 推出的目的，也正是为了取代这种表达式形式的宏定义，它消除了宏定义的缺点，同时又很好地继承了宏定义的优点。

在何时使用 `inline` 函数：

首先，你可以使用 `inline` 函数完全取代表达式形式的宏定义。另外要注意，内联函数一般只会用在函数内容非常简单的时候。这是因为，内联函数的代码会在任何调用它的地方展开，如果函数太复杂，代码膨胀带来的恶果很可能会大于效率的提高带来的益处。内联函数最重要的使用地方是用于类的存取函数。

## 比较

从 `inline` 的作用来看，其放置于函数声明中应当也是毫无作用的：`inline` 只会影响函数在 `translation unit`（可以简单理解为 C 源码文件）内的编译行为，只要超出了这个范围 `inline` 属性就没有任何作用了。所以 `inline` 关键字不应该出现在函数声明中，没有任何作用不说，有时还可能造成编译错误（在包含了 `sys/compiler.h` 的情况下，声明中出现 `inline` 关键字的部分通常无法编译通过）；`inline` 关键字仅仅是建议编译器做内联展开处理，而不是强制。在 `gcc` 编译器中，如果编译优化设置为 `-O0`，即使是 `inline` 函数也不会被内联展开，除非设置了强制内联（`__attribute__((always_inline))`）属性。

相对于 C99 的 `inline` 来说，GCC 的 `inline` 更容易理解：可以认为它是一个普通全局函数加上了 `inline` 的属性。即在其定义所在文件内，它的表现和 `static inline` 一致：在能展开的时候会被内联展开编译。但是为了能够在文件外调用它，`gcc` 一定会为它生成一份独立的汇编码，以便在外部进行调用。即从文件外部看来，它和一个普通的 `extern` 的函数无异。

GCC 的 `static inline` 定义很容易理解：你可以把它认为是一个 `static` 的函数，加上了 `inline` 的属性。这个函数大部分表现和普通的 `static` 函数一样，只不过在调用这种函数的时候，`gcc` 会在其调用处将其汇编码展开编译而不为这个函数生成独立的汇编码。除了以下几种情况外：\*（1）函数的地址被使用的时候。如通过函数指针针对函数进行了间接调用。这种情况下就不得不为 `static inline` 函数生成独立的汇编码，否则它没有自己的地址。\*（2）其他一些无法展开的情况，比如函数本身有递归调用自身的行为等。

`static inline` 函数和 `static` 函数一样，其定义的范围是 `local` 的，即可以在程序内有多个同名的定义（只要不位于同一个文件内即可）。注意：`gcc` 的 `static inline` 的表现行为和 C99 标准的 `static inline` 是一致的。所以这种定义可以放心使用而没有兼容性问题。

要点: gcc 的 `static inline` 相对于 `static` 函数来说只是在调用时建议编译器进行内联展开; gcc 不会特意为 `static inline` 函数生成独立的汇编码, 除非出现了必须生成不可的情况 (如通过函数指针调用和递归调用); gcc 的 `static inline` 函数仅能作用于文件范围内。

我全都要, 但鉴于你们有些人不想要, 我就把要的东西的共同部分抽象出来加到新版本里, 需要的人你们自己去做就好了。

C 是没什么是指针和宏不能解决的, 加那么多东西没意义。C++ 几乎是 C 的超集, 只有少量功能 C++ 不支持。

C++ 看起来很强大 (也的确很强大), 但你真用起来发现其实什么 (库) 也没有……

C 看起来什么都能做到 (汇编小声嘀咕: 我也可以), 但真写起来其实就是把 C++ 再发明了一遍 (有些还真做不到) ……

说面向对象/面向过程区别必然是错的, 因为 C 的程序写大了不可避免地还是要模拟一下面向对象的, 而 C++ 本身根本不局限于面向对象……

C++ 几乎是 C 的超集, 只有少量功能 C++ 不支持。

## static

`static` 的三条重要作用, 首先 `static` 的最主要功能是隐藏, 其次因为 `static` 变量存放在静态存储区, 所以它具备持久性和默认值 0。

- 静态变量
- 静态函数
- 静态成员

## 静态变量

全局变量本身就是静态存储方式, 静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同。

这两者的区别在于非静态全局变量的作用域是整个源程序, 当一个源程序由多个源文件组成时, 非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域, 即只在定义该变量的源文件内有效, 在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内, 只能为该源文件内的函数公用, 因此可以避免在其它源文件中引起错误。

把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域, 限制了它的使用范围。

`static` 局部变量只被初始化一次, `static` 变量会在静态存储区完成唯一的一次初始化。

### 静态函数

在函数的返回类型前加上关键字 `static`，函数就被定义成为静态函数。普通函数的定义和声明默认情况下是 `extern` 的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。因此定义静态函数有以下好处：

- 其他文件中可以定义相同名字的函数，不会发生冲突。
- 静态函数不能被其他文件所用。

`static` 函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝。

---

**备注：**静态数据成员属于类，非静态成员函数也可访问。

---

### 静态成员

静态成员是与类本身直接相关，而不是与类的各个对象保持关联，故类的静态成员不属于对象。但可以用类的对象、引用或指针来访问静态成员。

和其他函数一样，静态成员函数可以再类内部和外部定义。初始化一般在外部，在内部可以为静态成员提供一个 `const` 整数类型的类内初始化值。

静态成员不与任何对象绑定，不包含 `this` 指针，故其也不能声明为 `const` 的。

### const

C 语言关键字 `const` 就是用来限定一个变量不允许被改变的修饰符（Qualifier）。

ANSI C 规定数组定义时长度必须是“常量”（C99 标准，数组下标可以用变量来表示），“只读变量”也是可以的，“常量”不等于“不可变的变量”。但是在 C++ 中，局部数组是可以使用变量作为其长度的。

- 相关定义

### 相关定义

`const` 修饰的数据类型是指常类型，常类型的变量或对象的值是不能被更新的。

“只读变量”是在内存中开辟一个地方来存放它的值，只不过这个值由编译器限定不允许被修改。

`const` 使用的基本形式：

- 1) `const` 在前面

```
const int nValue;           //int是const
const char *pContent;       //char是const, pContent可变
const char * const pContent; //pContent和*pContent都是const
```

## 2) const 在后面

```
int const nValue;           //nValue是const
char const * pContent;       //*pContent是const, pContent可变
char* const pContent;       //pContent是const, *pContent可变
char const* const pContent; //pContent和*pContent都是const
```

一个简单的判断方法：指针运算符\*，是从右到左，那么如：char const \* pContent，可以理解为 char const (\*pContent)，即 \*pContent 为 const，而 pContent 则是可变的。

```
int const *p1,p2;
```

p2 是 const; (p1) 是一整体，因此 (\*p1) 是 const，但 p1 是可变的。int \*p1,p2 只代表 p1 是指向整型的指针，要表示 p1、p2 都是指针是需写成 int \*p1, p2。所以无论是 \*const p1,p2 还是 const \*p1,p2，里面的 \* 都是属于 p1 的。

```
int const * const p1,p2;
```

p2 是 const，是前一个 const 修饰的，\*p1 也被前一个 const 修饰，而 p1 被后一个 const 修饰。

```
int * const p1,p2;
```

p1 是 const，(\*const p1) 是整体，所以 const 不修饰 p2。

指针指向及其指向变量的值的变化，const 在 \* 的左边，则指针指向的变量的值不可直接通过指针改变（可以通过其他途径改变）；在 \* 的右边，则指针的指向不可变。简记为“左定值，右定向”。

const 默认修饰左边的内容，如果左边没有东西则修饰其右边的内容

- const int \*a //const 左边没有东西，因此 const 修饰 int，则指针指向的内容不可通过指针修改
- int const \*a //const 左边有东西，因此 const 修饰 int
- int\* const a //const 修饰 \*，即指针不能改变指向
- const int \* const a //第一个 const 修饰 int，第二个 const 修饰 \*，即指针指向内容不可修改，也不能改变指针指向

**提示：**若 \* 在 const 左边，不能改变指针指向，\* 在 const 右边，不能修改所指的值

const 成员只能在构造函数的初始化列表中初始化，如果非要在类中声明处初始化，就要加上 static 才行，而且初始化的对象必须是整型

C++11 `const` 成员可以初始化，结构体内部也可以初始化。但是 `static`（非 `const`）成员一定只能在类外初始化。用 `const` 常量代替宏定义可以让编译器进行安全性检查类的 `const` 成员函数不能修改类的成员变量，而且一个 `const` 类对象只能调用其 `const` 成员函数，不能调用非 `const` 成员函数 `const` 成员函数与同名、同返回值、同参数列表的非 `const` 成员函数属于重载现象

### endian

big-endian little-endian

在计算机里，对于地址的描述，很少用“大”和“小”来形容；对应地，用的更多的是“高”和“低”；

- big-endian：大端——高尾端
- little-endian：小端——低尾端

如果把一个数看成一个字符串，比如 11223344 看成“11223344”，‘11’到‘44’各占用一个存储单元，它的尾端是‘44’，如果是小端模式则‘44’存储在低地址位。

### 预编译

在 C 语言中，凡是以“#”开头的都叫预处理指令。

预编译又称预处理，是整个编译过程最先做的工作，即程序执行前的一些预处理工作。主要处理 # 开头的指令。如拷贝 `#include` 包含的文件代码、替换 `#define` 定义的宏、条件编译 `#if` 等。

# 是把宏参数转化为字符串的运算符，## 是把两个宏参数连接的运算符

```
#define STR(arg) #arg 则宏STR(hello)展开时为" hello"
#define NAME(y) name_y 则宏NAME(1)展开时仍为name_y
#define NAME(y) name_##y 则宏NAME(1)展开为name_1
```

- `int n = Conn(123,456);` 结果就是 `n=123456;`
- `char* str = Conn("asdf", "adf")` 结果就是 `str = "asdfadf";`
- `char a = ToChar(1);` 结果就是 `a='1';`

做个越界试验 `char a = ToChar(123);` 结果是 `a='3';` 但是如果你的参数超过四个字符，编译器就给你报错了！  
error C2015: too many characters in constant : P

- `char* str = ToString(123132);` 就成了 `str="123132";`

## 后缀表达式

**\*\* 后缀表达式：**先写运算对象再写符号，一般格式：`{运算对象}{运算对象}{操作符}` **\*\***

后缀表达式就是将运算符号移到两个运算对象后面，但不影响原有计算顺序

后缀表达式又称逆波兰表达式，明显的特点是：逆波兰表达式中没有括号，计算时将操作符之前的第一个数作为右操作数，第二个数作为左操作数，进行计算，得到的值继续放入逆波兰表达式中。但日常生活中我们总是习惯于写中缀表达式，所以需要先将中缀表达式转为后缀表达式。

---

**备注：**规则：从左到右遍历表达式的每个数字和符号，遇到是数字就进栈，遇到是符号，就将处于栈顶两个数字出栈，进行运算，运算结果进栈，一直到最终获得结果。

---

## 规范工具

### 编码风格

clang-format，它是基于 clang 的一个命令行工具，能够自动化格式 C/C++/Obj-C 代码，支持多种代码风格：Google, Chromium, LLVM, Mozilla, WebKit，也支持自定义风格（通过编写 clang-format 文件）很方便的同意代码格式。

### 静态检查

- 功能度：Cppcheck > TscanCode > Flawfinder
- 友好度：TscanCode > Cppcheck > Flawfinder
- 易用性：TscanCode > Cppcheck > Flawfinder

## C++

C++ 几乎是 C 的超集，只有少量功能 C++ 不支持。

- 相关区别
- *GP vs OOP*

### 相关区别

C++ 中的类具有成员保护功能，并且具有继承，多态这类 oo 特点，而 c 里的 struct 没有。C 里面的 struct 没有成员函数，不能继承，派生等等

C++ 中 struct 和 class 的主要区别在于默认的存取权限不同，struct 默认为 public，而 class 默认为 private

### GP vs OOP

- GP(generic programming): 类属编程，也叫泛型编程
- OOP(Object Oriented Programming): 面向对象编程

类属编程是构成库的另一种方式，这与传统的 oop 是不同的。这类程序库一般由类属组件和类属算法组成，组件和算法通过迭代器组装起来，组件则对迭代器提供一定的封装。这种程序库的优点在于能够提供比传统程序库更灵活的组装方式，而不损失效率。

广义的，将泛型程序设计描述为“利用模板设计的程序”(programming with template)，将面向对象程序设计描述为“利用继承的程序设计”(programming with inheritance)。

说面向对象/面向过程区别必然是错的，因为 C 的程序写大了不可避免地还是要模拟一下面向对象的，而 C++ 本身根本不局限于面向对象……

面向对象 4 大特性

- 封装 (encapsulation)
- 继承 (Inheritance)
- 多态 (Polymorphism)
- 抽象 (abstract)

### Summary

### 语法逻辑

- 判断条件的 && || 的逻辑关系常出现问题，大于两个条件的组合，需要理清楚是否有特殊情况
- 边界条件 >=, <= 等情况是否理清，在边界上的值是否归属到正常的类别中，有没有两边都属于，或者是两边都不属于



## 潜在问题

- 除数为 0
- 使用空指针
- 访问已经释放的动态内存
- 字符串尾不是空
- 超出数组范围赋值
- 不同类型的变量进行运算
- 不同类型的指针访问内存
- 终止条件不明确导致无法检查递归函数
- 无法检查地址及其运算
- 使用指针前未初始化

### 1.1.2 脚本语言

- *Python*
- *Lua*
- *Ruby*

## Python

Python 是一种解释型、面向对象、动态数据类型的高级程序设计语言。由 Guido van Rossum 于 1989 年底发明，第一个公开发行人版发行于 1991 年。像 Perl 语言一样，Python 源代码同样遵循 GPL(GNU General Public License) 协议。

## python

针对数组的 `index()` 方法可以返回指定值首次出现的位置

```
fruits = ['apple', 'banana', 'cherry']  
x = fruits.index("cherry")
```

### Lua

Lua 是一种轻量小巧的脚本语言，用标准 C 语言编写并以源代码形式开放，其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Lua 是巴西里约热内卢天主教大学（Pontifical Catholic University of Rio de Janeiro）里的一个研究小组于 1993 年开发的

- 轻量级: 它用标准 C 语言编写并以源代码形式开放，编译后仅仅一百余 K，可以很方便的嵌入别的程序里。
- 可扩展: Lua 提供了非常易于使用的扩展接口和机制：由宿主语言（通常是 C 或 C++）提供这些功能，Lua 可以使用它们，就像是本来就内置的功能一样。
- 支持面向过程 (procedure-oriented) 编程和函数式编程 (functional programming)；
- 自动内存管理；只提供了一种通用类型的表 (table)，用它可以实现数组，哈希表，集合，对象；
- 语言内置模式匹配；闭包 (closure)；函数也可以看做一个值；提供多线程（协同进程，并非操作系统所支持的线程）支持；
- 通过闭包和 table 可以很方便地支持面向对象编程所需要的一些关键机制，比如数据抽象，虚函数，继承和重载等。

### Ruby

Ruby 是一种简单快捷的面向对象（面向对象程序设计）脚本语言，在 20 世纪 90 年代由日本人松本行弘 (Yukihiro Matsumoto) 开发，遵守 GPL 协议和 Ruby License。

Ruby 是动态语言，你可以在程序中修改先前定义过的类。也可以在某个类的实例中定义该实例特有的方法，这叫做单例方法。

### 1.1.3 汇编语言

汇编语言的特点:

- 所占空间、执行速度与机器语言相仿
- 直接、简捷，能充分控制计算机的硬件功能

三种语言的层次关系

- 机器语言
- 汇编语言
- 高级语言

## 案例解析

### WS2812 驱动

- WS2812 简介
- STM32 实现
- STC 实现

### WS2812 简介

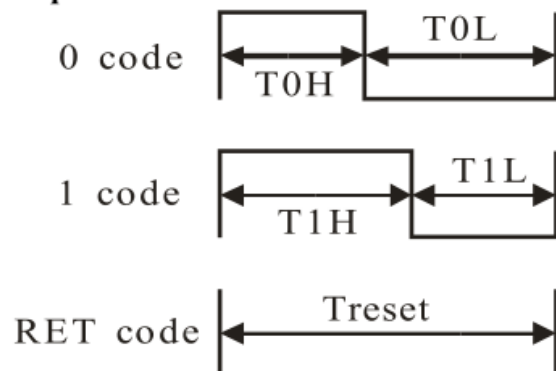
这款 RGB LED 采用单总线通信方式，可以多级串联。但对时序的要求比较高，低速单片机开发难度大。

该芯片的驱动时序要求如下：

**Data transfer time( TH+TL=1.25μs±600ns)**

T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.7us	±150ns
T0L	0 code , low voltage time	0.8us	±150ns
T1L	1 code ,low voltage time	0.6us	±150ns
RES	low voltage time	Above 50μs	

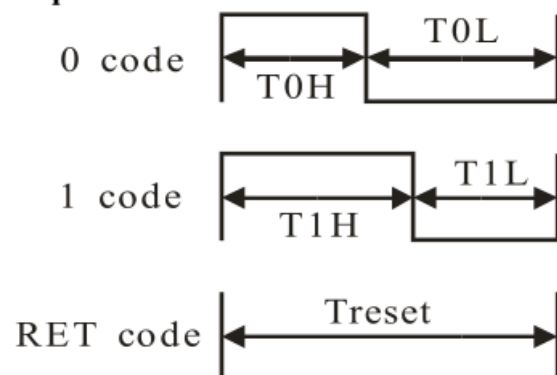
**Sequence chart:**



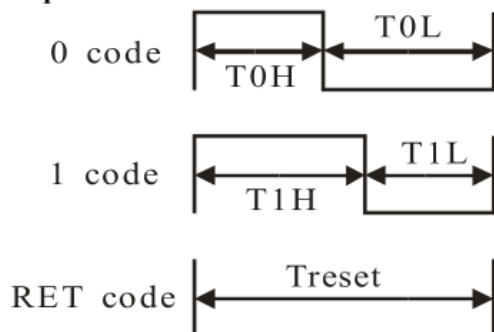
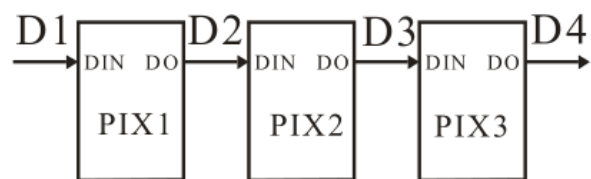
<https://blog.csdn.net/u013062709>

**Data transfer time( TH+TL=1.25 $\mu$ s $\pm$ 150ns)**

T0H	0 code ,high voltage time	0.35us	$\pm$ 150ns
T1H	1 code ,high voltage time	0.9us	$\pm$ 150ns
T0L	0 code , low voltage time	0.9us	$\pm$ 150ns
T1L	1 code ,low voltage time	0.35us	$\pm$ 150ns
RES	low voltage time	Above 50 $\mu$ s	

**Sequence chart:**
<https://blog.csdn.net/u013062709>
**Data transfer time( TH+TL=1.25 $\mu$ s $\pm$ 150ns)**

T0H	0 code ,high voltage time	0.3 us	$\pm$ 150ns
T1H	1 code ,high voltage time	0.6 us	$\pm$ 150ns
T0L	0 code , low voltage time	0.9 us	$\pm$ 150ns
T1L	1 code ,low voltage time	0.6us	$\pm$ 150ns
RES	low voltage time	80 $\mu$ s	

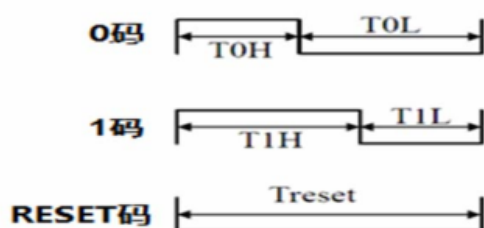
**Sequence chart:****Cascade method:**
<https://blog.csdn.net/u013062709>

### 10. The data transmission time ( $T_H+T_L=1.25\mu s\pm 600ns$ ):

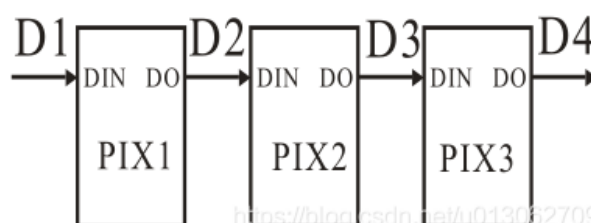
T0H	0 code, high level time	0.3 $\mu s$	$\pm 0.15\mu s$
T0L	0 code, low level time	0.9 $\mu s$	$\pm 0.15\mu s$
T1H	1 code, high level time	0.6 $\mu s$	$\pm 0.15\mu s$
T1L	1 code, low level time	0.6 $\mu s$	$\pm 0.15\mu s$
Trst	Reset code, low level time	80 $\mu s$	

### 11. Timing waveform:

Input code:



Connection mode:

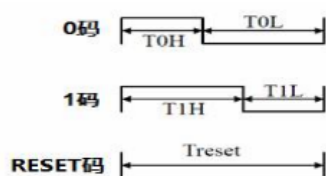


### 数据传输时间 ( $T_H+T_L=1.25\mu s\pm 600ns$ )

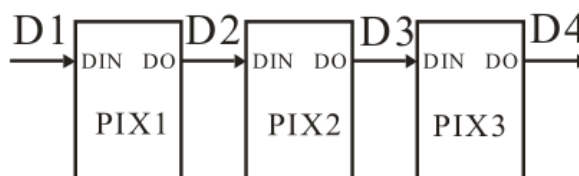
T0H	0码, 高电平时间	0.35 $\mu s$	$\pm 150ns$
T1H	1码, 高电平时间	0.7 $\mu s$	$\pm 150ns$
T0L	0码, 低电平时间	0.8 $\mu s$	$\pm 150ns$
T1L	1码, 低电平时间	0.6 $\mu s$	$\pm 150ns$
RES	帧单位, 低电平时间	50 $\mu s$ 以上	

### 时序波形图

输入码型:



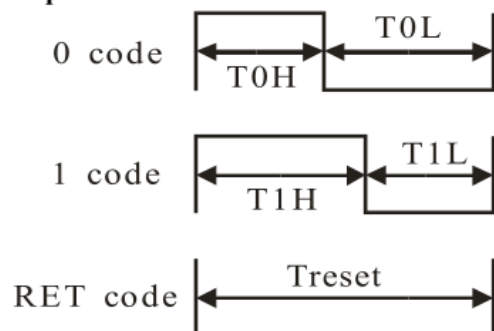
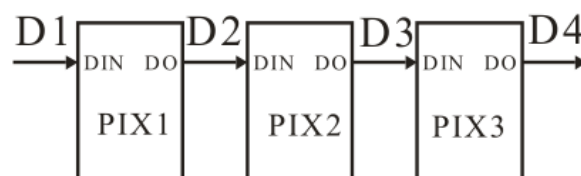
连接方法:



### 数据传输方法:

**Data transfer time( TH+TL=1.25μs±150ns)**

T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.85us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.4us	±150ns
RES	low voltage time	Above 50μs	

**Sequence chart:****Cascade method:**

<https://blog.csdn.net/u013062709>

**STM32 实现**

对于 STM32 等主频较高的 MCU 而言，并不存在需要通过汇编来实现，其主要的实现方式有：

- 直接控制 IO 口，并精确调整延时；
- 将 SPI 的时钟调整为 8MHz，发送一字节正好是 1.25us，给 ws2812 发送 0 即通过 SPI 总线发送 11000000b，发送 1 即通过 SPI 总线发送 11111100b；
- 第三种方式使用 PWM，周期设置为 3MHz，发送 0 就把占空比设置为 33%，发送 1 就把占空比设置为 66%；

## STC 实现

假设 STC 主频为 12MHz,  $1.25\mu\text{s}$  即  $0.00000125/(1/12\text{M})=15$  个周期, 也就是说我们要在 15 个周期内执行一定数量的指令来完成 IO 置高、IO 置低、数据移位、跳转等所有必要的操作。

指令集	STC-Y1	STC-Y3	STC-Y5	STC-Y6
代表型号	STC89C52	STC12C5A60S2	STC15W4K32S4	STC8A8K64S4A12
MOV A, Rn (寄存器送入累加器)	12	1	1	1
MOV Rn, A (累加器送入寄存器)	12	2	1	1
MOV Rn, #data (立即数送入寄存器)	12	2	2	1
MOV bit, C (进位位送入直接地址位)	24	4	3	1
RLA (累加器循环左移)	12	1	1	1
RLC A (累加器带进位循环左移)	12	1	1	1
CLR A (清零累加器)	12	1	1	1
CLR C (清零进位位)	12	1	1	1
CLR bit (清零直接地址位)	12	4	3	1
SETB bit (置一直接地址位)	12	4	3	1
INC Rn (寄存器加一)	12	3	2	1
DEC Rn (寄存器减一)	12	3	2	1
ADD A, direct (直接地址单元中的数据加到累加器)	12	3	2	1
SUBB A, #data (累加器带借位减立即数)	12	2	2	1
JZ rel (累加器为零转移)	24	3	4	条件不成立不转移1时钟; 条件成立则转移3时钟
JNC rel (进位为零转移)	24	3	3	条件不成立不转移1时钟; 条件成立则转移3时钟

### 1. C 实现

```
void ws2812_write_byte(u8 dat)
```

(续下页)

(接上页)

```
{
    u8 i = 8;
    dat <=<= 1;
    while(i)
    {
        WS2812_IO = 1;
        WS2812_IO = CY;
        WS2812_IO = 0;
        dat <=<= 1;
        i--;
    }
}
```

**备注：**CY 进位标志位（在进行算术运算时，可以被硬件置位或清零，以表示运算结果中高位是否有进位或借位的状态。

## 2. 汇编实现

```
void ws2812asm(unsigned char dat)
{
#pragma asm
    MOV A, R7
    MOV R6, #0x08
WS2812LOOP:
    SETB P3.7
    RLC A
    MOV P3.7, C
    NOP
    CLR P3.7
    DJNZ R6, WS2812LOOP
#pragma endasm
}
```

R7 即函数调用时传入的 dat，将 dat 放入累加器 A 中，然后将 8 放入 R6 中作为循环计数，这里我用的是 P3.7 引脚，用 SETB 语句将它置为高电平，然后用 RLC 指令将 dat 左移一位，最高位进入进位位 C，使用 MOV 语句将进位位的值赋给 P3.7 引脚，NOP 延时，之后将 P3.7 引脚置为低电平，DJNZ 将计数值减一不为零则跳转到 WS2812LOOP 继续执行。整个代码的循环体耗时为  $3+1+3+1+3+4=15$  个 CLK

循环体外有两个 MOV 语句，函数调用前会有一个 MOV 语句将参数传入寄存器 R7，之后 LCALL 语句调用函数，调用完之后还有 RET 语句返回，这样一来整个函数调用的总时间为  $3+4+1+2+14+4=28$  个 CLK

小于 45us 的高电平为判定为逻辑 0，大于 45us 的高电平被判定为逻辑 1，低电平的时长只要不要超过复位信号的时长都可以完成数据的传输！



## ESP ULP

- 案例简介

### 案例简介

用于编程控制 ESP32 ULP 协处理器

## 1.2 算法实现

Algorithm

算法复杂度分为时间复杂度和空间复杂度。

- 时间复杂度是指执行算法所需要的计算工作量；
- 空间复杂度是指执行这个算法所需要的内存空间；

- 时间复杂度
- 空间复杂度

### 1.2.1 时间复杂度

常见的时间复杂度量级有：

- 常数阶  $O(1)$
- 对数阶  $O(\log N)$
- 线性阶  $O(n)$
- 线性对数阶  $O(n \log N)$
- 平方阶  $O(n^2)$
- 立方阶  $O(n^3)$
- K 次方阶  $O(n^k)$
- 指数阶  $(2^n)$

**备注：**上面从上至下依次的时间复杂度越来越大，执行的效率越来越低。

```
for(i=1; i<=n; ++i)
{
    j = i;
    j++;
}
```

该示例的时间复杂度为： $O(n)$ ；整个耗时  $T(n) = (1+2n) \times \text{时间颗粒}$ ，算法的耗时是随着  $n$  的变化而变化，可以简化这个算法的时间复杂度表示为： $T(n) = O(n)$ ，如果  $n$  无限大， $T(n) = \text{time}(1+2n)$  中的常量 1 就没有意义了，倍数 2 也意义不大。因此直接简化为  $T(n) = O(n)$

时间复杂度公式  $T(n) = O(f(n))$  中  $f(n)$  表示每行代码执行次数之和，而  $O$  表示正比例关系，公式全称：**算法的渐进时间复杂度**

### 哈希表解法

复杂度  $O(n)$

时间复杂度和空间复杂度均为  $O(n)$

### 双指针算法

如果最开始数组有序，可以利用数组最大值 + 最小值和 **target** 的关系进行筛选，这样空间复杂度直接降到 1，但是如果没序的话，就要对原值进行排序，所浪费的空间和时间比起哈希表来讲，得不偿失。

## 1.2.2 空间复杂度

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的一个量度，同样反映的是一个趋势，我们用  $S(n)$  来定义。

```
int[] m = new int[n]
for(i=1; i<=n; ++i)
{
    j = i;
    j++;
}
```

示例代码第一行 **new** 了一个数组出来，这个数据占用的大小为  $n$ ，这段代码的 2-6 行，虽然有循环，但没有再分配新的空间，因此，这段代码的空间复杂度主要看第一行即可，即  $S(n) = O(n)$

编程练习 [leetcode](#)

## 1.3 开发策略

### 1.3.1 RAF 策略

RAF(Run and Fix) 是大量开发者和团队采用的开发策略，通过试错的方式降低精力投入，但在实际编写的代码最后可能会遗留下大麻烦，浪费不少于编码的时间进行修复。

- 增加在程序构思阶段的时间投入
- 采用流程图和伪代码，梳理程序逻辑
- 在纸上投入的时间多于电脑
- 事先预测可能的问题并寻找解决方法
- 深思后再编程



## 2.1 GCC

- 简介
- 对比
- 库链接
- 存储访问
- *SDCC*
- *Lint*

### 2.1.1 简介

GCC(GNU Compiler Collection) 是由 GNU 开发的编程语言译器。GNU 编译器套件包括 C、C++、Objective-C、Fortran、Java、Ada 和 Go 语言前端，也包括了这些语言的库（如 libstdc++，libgcj 等。）

gcc 与 g++ 分别是 gnu 的 c & c++ 编译器 gcc/g++ 在执行编译工作的时候，总共需要 4 步：

1、预处理, 生成.i 的文件 [预处理器 cpp] 2、将预处理后的文件转换成汇编语言, 生成文件.s [编译器 egcs] 3、有汇编变为目标代码 (机器代码) 生成.o 的文件 [汇编器 as] 4、连接目标代码, 生成可执行程序 [链接器 ld]

arm-elf-gcc 跟 arm-linux-gcc 一样，也是基于 ARM 目标机的交叉编译软件。但是它们不是同一个交叉编译软件，两者是有区别的，两者区别主要在于使用不同的 C 库文件。arm-linux-gcc 使用 GNU 的 Glibc，而

arm-elf-gcc 一般使用 uClibc/uC-libc 或者使用 RedHat 专门为嵌入式系统的开发的 C 库 newlib。只是所应用的领域不同而已，Glibc 是针对 PC 开发的，uClibc/uC-libc 是与 Glibc API 兼容的小型化 C 语言库，实现了 Glibc 部分功能。

uC-libc 是最早为 uClinux 开发的库，是 Jeff Dionne 和 Kenneth Albanowski 为在 EKLs 项目中支持 m68000 在 Linux-8086 C 库源码上移植的。uC-libc 是一个完全的 libc 实现，但其中有一些 api 是非标准的，有些 libc 的标准也没有实现。uC-libc 稳定地支持 m68000，ColdFire 和没有 MMU 的 ARM。其主要设计目标是“小”、“轻”，并尽量与标准一致，虽然它的 API 和很多 libc 兼容，但是似乎并不像它期望的那样和所有标准一致。

uClibc 就是为了解决这个问题从 uC-libc 中发展出来的。它的所有 API 都是标准的 (正确的返回类型，参数等等)，它弥补了 uC-libc 中没有实现的 libc 标准，现在已经被移植到多种架构中。一般来讲，它尽量兼容 glibc 以便使应用程序用 uClibc 改写变的容易。uClibc 能够在标准的 VM linux 和 uClinux 上面使用。为了应用程序的简洁，它甚至可以在许多支持 MMU 的平台上被编译成共享库。

arm-linux-\*和 arm-elf-\*的使用没有一个绝对的标准，排除不同库实现的差异，gcc 可以编译任何系统。arm-linux-\*和 arm-elf-\*都可以用来编译裸机程序和操作系统，只是在遵循下面的描述时系统程序显得更加协调：

arm-linux-\*针对运行 linux 的 ARM 机器，其依赖于指定的 C 语言库 Glibc，因为同样使用 Glibc 的 linux 而使得 arm-linux-\*在运行 linux 的 ARM 机器上编译显得更加和谐。

arm-elf-\*则是一个独立的编译体系，不依赖于指定的 C 语言库 Glibc，可以使用 newlib 等其他 C 语言库，不要求操作系统支持，当其使用为嵌入式系统而设计的一些轻巧的 C 语言库时编译裸机程序 (没有 linux 等大型操作系统的程序)，如监控程序，bootloader 等能使得系统程序更加小巧快捷。

## 2.1.2 对比

ARM 的嵌入式系统开发中，常常用到交叉编译的 GCC 工具链有两种：arm-linux-和 arm-elf-，两者区别主要在于使用不同的 C 库文件。arm-linux-\*使用 GNU 的 Glibc，而 arm-elf-\*一般使用 uClibc/uC-libc 或者使用 REDHAT 专门为嵌入式系统的开发的 C 库 newlib.Glibc。uClibc/uC-libc 以及 newlib 都是 C 语言库文件，只是所应用的领域不同而已，Glibc 是针对 PC 开发的，uClibc/uC-libc 是与 Glibc API 兼容的小型化 C 语言库，实现了 Glibc 部分功能。

1、EABIarm-2008q3-39-arm-none-eabi Sourcery G++ Lite 2008q3-39 All versions... Sourcery G++ for ARM EABI is for use in bare-metal and/or RTOS environments. (适用于编译裸机或 RTOS 环境上的应用，比如 u-boot 等)；Run-Time Libraries：ARMv4 - Little-Endian, Soft-Float；ARMv4 Thumb -Little- Endian, Soft-Float；ARMv6-M Thumb - Little-Endian, Soft-Float；ARMv7 Thumb-2 - Little-Endian, Soft-Float。2、uClinux arm-2008q3-42-arm-uclinuxeabi Sourcery G++ Lite 2008q3-42 All versions... Sourcery G++ for ARM uClinux is for systems running the Linux kernel without using a memory-management unit (MMU). You can use Sourcery G++ to build both the uClinux kernel and uClinux applications. ) 适用于编译 linux 内核和应用程序，不带 MMU 的 CPU)；Run-Time Libraries：ARMv4T - Little-Endian, Soft-Float；ARMv6-M Thumb - Little-Endian, Soft-Float；ARMv7 Thumb-2 - Little-Endian, Soft-Float。3、GNU/Linux arm-2008q3-41-arm-none-linux-gnueabi Sourcery G++ Lite 2008q3-41 All versions... Sourcery G++ for ARM GNU/Linux is for use in developing for systems which run the Linux kernel. You can use Sourcery G++ to build both the Linux kernel and Linux applications. (适用于编译 linux 内核和应用程序，带 MMU 的 CPU)；Run-Time Libraries：ARMv4T - Little-Endian, Soft-Float, GLIBC；ARMv5T - Little-Endian, Soft-Float, GLIBC；ARMv7-A Thumb-2 - Little-Endian, Soft-Float, GLIBC。4、SymbianOS arm-2008q3-40-arm-none-symbianelf Sourcery G++ Lite

2008q3-40 All versions... 适用于编译 Symbian 应用程序；Run-Time Libraries：ARMv5 - Little-Endian, Soft-Float；ARMv5 - Little-Endian, VFP。

arm-linux-gcc 是针对 arm + Linux 的开发环境的，kernel 使用的是 linux，不是 uclinux，arm 是有硬件 MMU 的。

而 arm-elf-gcc 是针对 no MMU arm + uclinux 的开发环境，kernel 使用的是 uclinux，硬件是廉价的无 MMU 的 arm 芯片。

arm-linux-gcc 倒是有点类似 X86 PC 环境下的 linux 开发。

其实这两个交叉编译器只不过是 gcc 的选项-mfloat-abi 的默认值不同。gcc 的选项-mfloat-abi 有三种值 soft,softfp,hard(其中后两者都要求 arm 里有 fpu 浮点运算单元,soft 与后两者是兼容的，但 softfp 和 hard 两种模式互不兼容)：

soft：不用 fpu 进行浮点计算，即使有 fpu 浮点运算单元也不用，而是使用软件模式。softfp：armel 架构 (对应的编译器为 gcc-arm-linux-gnueabi) 采用的默认值，用 fpu 计算，但是传参数用普通寄存器传，这样中断的时候，只需要保存普通寄存器，中断负荷小，但是参数需要转换成浮点的再计算。hard：armhf 架构 (对应的编译器 gcc-arm-linux-gnueabi) 采用的默认值，用 fpu 计算，传参数也用 fpu 中的浮点寄存器传，省去了转换，性能最好，但是中断负荷高。

### arm-none-linux-gnueabi-gcc

(ARM architecture, no vendor, creates binaries that run on the Linux operating system, and uses the GNU EABI)

主要用于基于 ARM 架构的 Linux 系统，可用于编译 ARM 架构的 u-boot、Linux 内核、linux 应用等。arm-none-linux-gnueabi 基于 GCC，使用 Glibc 库，经过 Codesourcery 公司优化过推出的编译器。arm-none-linux-gnueabi-xxx 交叉编译工具的浮点运算非常优秀。一般 ARM9、ARM11、Cortex-A 内核，带有 Linux 操作系统的会用到。

### arm-eabi-gcc

Android ARM 编译器。

### armcc

ARM 公司推出的编译工具，功能和 arm-none-eabi 类似，可以编译裸机程序 (u-boot、kernel)，但是不能编译 Linux 应用程序。

armcc 一般和 ARM 开发工具一起，Keil MDK、ADS、RVDS 和 DS-5 中的编译器都是 armcc，所以 armcc 编译器都是收费的。

### **arm-none-uclinuxeabi**

用于 uCLinux，使用 Glibc。

### **arm-none-symbianelf**

用于 symbian

## **ABI vs EABI**

ABI：二进制应用程序接口 (Application Binary Interface (ABI) for the ARM Architecture)。在计算机中，应用二进制接口描述了应用程序（或者其他类型）和操作系统之间或其他应用程序的低级接口。

EABI：嵌入式 ABI。嵌入式应用二进制接口指定了文件格式、数据类型、寄存器使用、堆积组织优化和在一个嵌入式软件中的参数的标准约定。开发者使用自己的汇编语言也可以使用 EABI 作为与兼容的编译器生成的汇编语言的接口。

两者主要区别是，ABI 是计算机上的，EABI 是嵌入式平台上（如 ARM，MIPS 等）。

### **2.1.3 库链接**

我们用 gcc 编译程序时，可能会用到 “-I”（大写 i），“-L”（大写 l），“-l”（小写 l）等参数，下面做个记录：

例：gcc -o hello hello.c -I /home/hello/include -L /home/hello/lib -lworld

上面这句表示在编译 hello.c 时：

-I /home/hello/include

表示将/home/hello/include 目录作为第一个寻找头文件的目录，寻找的顺序是：

/home/hello/include-->/usr/include-->/usr/local/include

### **2.1.4 存储访问**

\_no\_init 用于禁止系统启动时的变量初始化

用 \_\_ramfunc 定义的函数企图访问 ROM 将导致编译器产生警告

arm-none-eabi-gcc（ARM architecture，no vendor，not target an operating system，complies with the ARM EABI）用于编译 ARM 架构的裸机系统（包括 ARM Linux 的 boot、kernel，不适用编译 Linux 应用 Application），一般适合 ARM7、Cortex-M 和 Cortex-R 内核的芯片使用，所以不支持那些跟操作系统关系密切的函数，比如 fork(2)，他使用的是 newlib 这个专用于嵌入式系统的 C 库。



## 2.1.5 SDCC

### SDCC

- 简介

### 简介

## 2.1.6 Lint

### LINT

- *PC-Lint*
  - 安装配置
  - 代码分析
    - \* *641: (Warning -- Converting enum 'XXX' to int)*

### PC-Lint

随着项目的推进与迭代，一个 Project 的代码量往往会不知不觉增长，当项目代码达到数万行，迭代经历较长时间后，仅靠开发人员自身的代码质量已不能满足对整体质量的把控。

难以避免出现一些潜在的逻辑错误与非逻辑错误。这种情况下，定期 code review 是不错的选择，但是在开发人员数量较少，开发模式趋于敏捷开发的今天，快速迭代，开发人员技术参差不齐，即便选择 code review 也不能发现大多数潜在风险。如果自己进行全盘代码的 review，其难度不亚于将项目重构，此时引入工具就变得迫在眉睫。

PC-Lint 是 C/C++ 软件代码静态分析工具，你可以把它看作是一种更加严格的编译器。它不仅可以检查出一般的语法错误，还可以检查出那些虽然符合语法要求但不易发现的潜在错误。

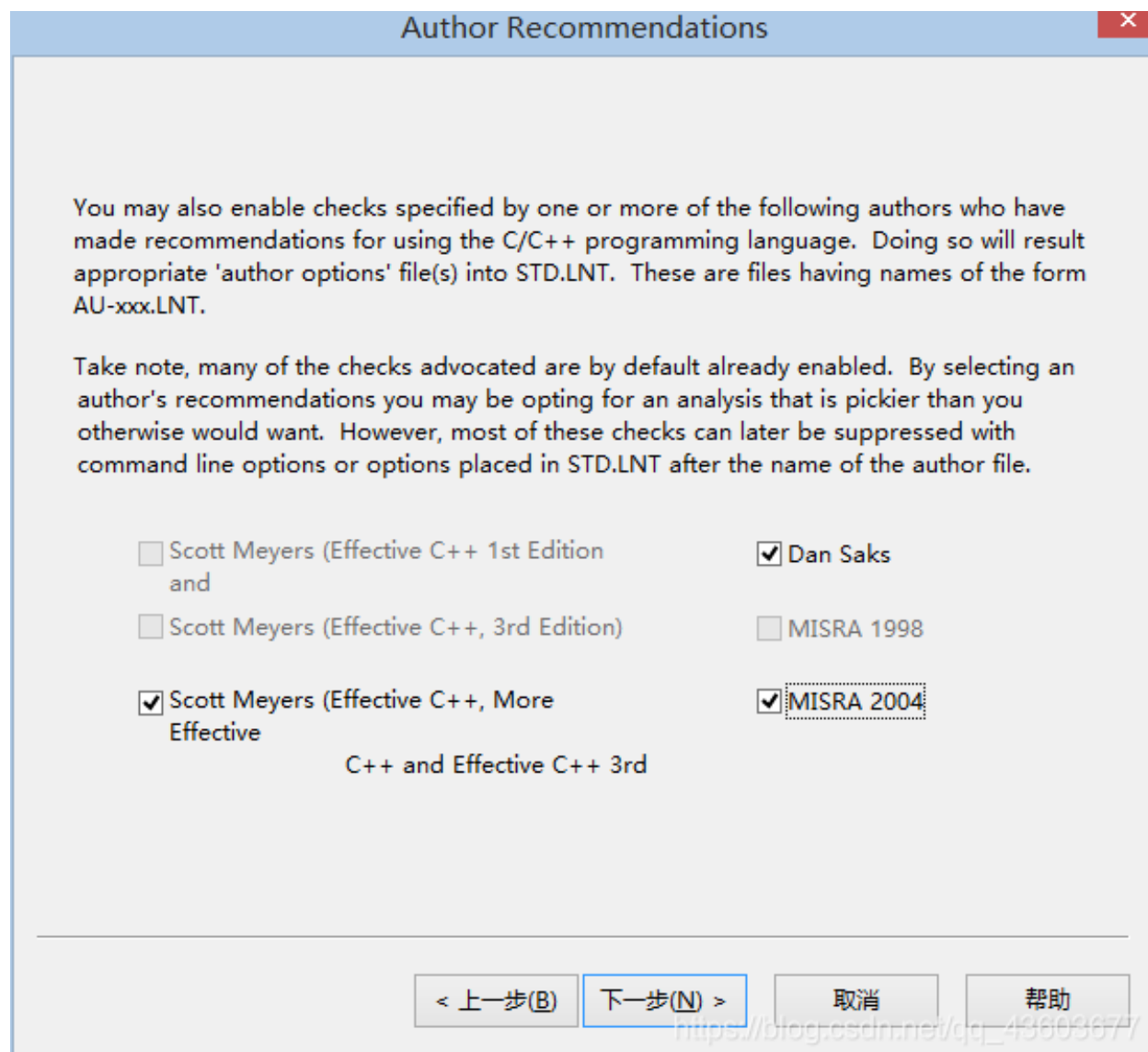
PCLint 识别并报告 C 语言中的编程陷阱和格式缺陷的发生。它进行程序的全局分析，能识别没有被适当检验的数组下标，报告未被初始化的变量，警告使用空指针，冗余的代码，等等。软件除错是软件项目开发成本和延误的主要因素。PCLint 能够帮你在程序动态测试之前发现编码错误。这样消除错误的成本更低。使用 PC-Lint 在代码走读和单元测试之前进行检查，可以提前发现程序隐藏错误，提高代码质量，节省测试时间。并提供编码规则检查，规范软件人员的编码行为。

PC-Lint 版本：PC-lint for C/C++ (NT) Vers. 9.00L ([https://files.cnblogs.com/files/godan/Gimpel\\_PC\\_Lint\\_9.rar](https://files.cnblogs.com/files/godan/Gimpel_PC_Lint_9.rar)) 免费可用版本

下载好 PC-Lint 后，需要再去官网下载最新的 patch 包。<https://gimpel.com/>

## 安装配置

PC-Lint 是 C/C++ 软件代码静态分析工具，你可以把它看作是一种更加严格的编译器。它不仅可以检查出一般的语法错误，还可以检查出那些虽然符合语法要求但不易发现的潜在错误。C 语言的灵活性带来了代码效率的提升，但相应带来了代码编写的随意性，另外 C 编译器不进行强制类型检查，也带来了代码编写的隐患。PCLint 识别并报告 C 语言中的编程陷阱和格式缺陷的发生。它进行程序的全局分析，能识别没有被适当检验的数组下标，报告未被初始化的变量，警告使用空指针，冗余的代码，等等。软件除错是软件项目开发成本和延误的主要因素。PCLint 能够帮你在程序动态测试之前发现编码错误。这样消除错误的成本更低。使用 PC-Lint 在代码走读和单元测试之前进行检查，可以提前发现程序隐藏错误，提高代码质量，节省测试时间。并提供编码规则检查，规范软件人员的编码行为。由于 PC-LINT 对于一般程序员来说可能比较陌生，有好多人都安装了也不知道怎样配置和使用。



## 代码分析

### 641: (Warning -- Converting enum 'XXX' to int)

不能直接把 enum 转成 int;

## 2.2 IDE

- *Keil*
- *IAR*
- *STM32CubeIDE*
- *Embedded Studio*
- *Summary*

### 2.2.1 Keil

Keil MDK-ARM（旧称 RealView MDK）开发工具源自德国 Keil 公司，被全球上百万的嵌入式开发工程师验证和使用，是 ARM 公司目前最新推出的针对各种嵌入式处理器的软件开发工具。

KEIL MDK 集成了业内最领先的技术，包括 uVision3、uVision4、uVision5 集成开发环境与 ARM 编译器。支持 ARM7、ARM9、Cortex-M0、Cortex-M0+、Cortex-M3、Cortex-M4、Cortex-R4 内核处理器。

Keil MDK 可以自动配置启动代码，集成 Flash 烧写模块，强大的 Simulation 设备模拟，性能分析等功能，与 ARM 之前的工具包 ADS 等相比，ARM 编译器的最新版本可将性能改善超过 20% 以上。

#### Keil

- *MDK*
  - 生成文件
    - \* 概念区别
  - 优化等级
  - 编程算法 *FLM*
  - 格式输出
    - \* *fromelf*

- 使用技巧
- 常见问题
- C51
  - 常见问题

## MDK

### 生成文件

Program Size: Code=100800 RO-data=14396 RW-data=1200 ZI-data=35312

- Code：即代码域，它指的是编译器生成的机器指令，这些内容被存储到 ROM 区。
- RO-data：Read Only data，即只读数据域，它指程序中用到的只读数据，这些数据被存储在 ROM 区，因而程序不能修改其内容。例如 C 语言中 const 关键字定义的变量就是典型的 RO-data。
- RW-data：Read Write data，即可读写数据域，它指初始化为“非 0 值”的可读写数据，程序刚运行时，这些数据具有非 0 的初始值，且运行的时候它们会常驻在 RAM 区，因而应用程序可以修改其内容。例如 C 语言中使用定义的全局变量，且定义时赋予“非 0 值”给该变量进行初始化。
- ZI-data：Zero Initialie data，即 0 初始化数据，它指初始化为“0 值”的可读写数据域，它与 RW-data 的区别是程序刚运行时这些数据初始值全都为 0，而后续运行过程与 RW-data 的性质一样，它们也常驻在 RAM 区，因而应用程序可以更改其内容。例如 C 语言中使用定义的全局变量，且定义时赋予“0 值”给该变量进行初始化（若定义该变量时没有赋予初始值，编译器会把它当 ZI-data 来对待，初始化为 0）；
- ZI-data 的栈空间 (Stack) 及堆空间 (Heap)：在 C 语言中，函数内部定义的局部变量属于栈空间，进入函数的时候从向栈空间申请内存给局部变量，退出时释放局部变量，归还内存空间。而使用 malloc 动态分配的变量属于堆空间。在程序中的栈空间和堆空间都是属于 ZI-data 区域的，这些空间都会被初始化为 0 值。编译器给出的 ZI-data 占用的空间值中包含了堆栈的大小（经实际测试，若程序中完全没有使用 malloc 动态申请堆空间，编译器会优化，不把堆空间计算在内）。

### 概念区别

在 keil 里编译完后被分成 5 个内存段（堆、栈、bss 段、data 段、text 段）

在 stm32 中 flash 就是 ROM，掉电数据不会丢失；通常保存着 text 段、Code、Ro-data、Rw-data RAM 就是运行内存，掉电数据就丢失；通常保存着堆、栈、bss 段、data 段、ZI-data、RW-data

## 优化等级

- -O0 最少的优化，可以最大程度上配合产生代码调试信息，可以在任何代码行打断点，特别是死代码处。
- -O1 有限的优化，去除无用的 inline 和无用的 static 函数、死代码消除等，在影响到调试信息的地方均不进行优化。在适当的代码体积和充分的调试之间平衡，代码编写阶段最常用的优化等级。
- -O2 高度优化，调试信息不友好，有可能会修改代码和函数调用执行流程，自动对函数进行内联等。
- -O3 最大程度优化，产生极少量的调试信息。会进行更多代码优化，例如循环展开，更激进的函数内联等。

另外，可以通过单独设置 `--loop_optimization_level=option` 来控制循环展开的优化等级。

**警告：**勾选了 “use cross-module optimization//跨模块优化，KEIL 每次都要编译全部文件并且每个文件编译三次

## 编程算法 FLM

MDK 在下载程序之前需要都在 Debug 设置的 Flash Download 子选项卡选择编程算法。大多数时候，我们只要安装了芯片包之后，就可以直接得到对应的编程算法，并不需要我们去修改它。

但是，当你是一个芯片包的开发者，或者你有独特的下载需求（比如在你的程序里加入一些校验信息），这个时候你就需要去了解它了！

编程算法主要功能就是擦除相应的内存块，并将我们的程序写入到相应的内存区域上去。在你点击下载按钮的时候，这段程序会被先下载到 RAM 上（RAM for Algorithm 上的设置），然后才会通过它，将你的程序写入到指定的内存区域内。

## 格式输出

使用 fromelf 工具, 通过上面的示例, 想必都能很轻松的生成 bin 文件, 今天补写一下 fromelf 工具的基本命令:

`--bin`: 输出二进制文件 `--i32`: Intel 32 位 Hex `--m32`: Motorola 32 位 Hex `--output <file>:file` 为输出文件名 `-o<file>`: 这个是 armcc 编译器命令, 也可用于这里, 指定输出文件的名字

```
fromelf --bin "$L@L.axf" --output "$L@L.bin"
fromelf.exe --bin -o "$L@L.bin" "#L"
```

### fromelf

fromelf.exe 绝对路径 + 空格 + --bin(注意是两个短横的)+ 空格 + --output(两短横)+ 空格 + ../输出目录相对路径 + 空格 + 名字.bin+ 空格 + ../输出目录相对路径 + 空格 + 名字.axf

- fromelf.exe --bin --output .output@test2.bin .output@test2.axf
- fromelf.exe --bin --output ..@test2.bin ..@test2.axf

fromelf 中 \$L、@L、L 用来指定对应的路径或名称

- L 是指 axf 文件路径加文件名，例如 D:\qitasouttest.axf
- \$L 是指 axf 的文件路径不含文件名 (包含最后 “.”)，例如 D:\qitasout
- @L 是指 axf 的不含 axf 的后缀文件名，例如 test
- #L 的 # 表示引用的是本身，#L 即工程的输出文件

---

**备注：**\$K 表示的是 KEIL MDK 工具链的安装路径

---

### 使用技巧

- HardFault\_Handler

<https://blog.csdn.net/electrocrazy/article/details/78173558>

### 常见问题

MDK 偶尔会出现错误提示 “Error: Encountered an improper argument”。大概意思是说 “错误：遇到不正确的参数”。出现这种情况时，对话框关掉之后会再次出现，只能使用任务管理器强制停止才行。在官网上查一下这个错误信息，原来是 Keil 软件的 BUG。

在某些情况下，当您退出调试会话时，可能会显示一个错误对话框，提示 “遇到不正确的参数”。如果发生这种情况，μVision 需要使用 Windows 任务管理器终止。在大多数情况下，亚洲使用 Windows 操作系统的客户在项目路径中使用亚洲字符时会受到此问题的影响。很有可能你的工程路径中有中文（不过之前 Keil 是支持的），将路径变成中文就可以的了。

- WARNING L2: REFERENCE MADE TO UNRESOLVED EXTERNAL

如果你在用 C51 编译器出现上面的警告，这个只是初学者和粗心者才会犯的错误：没把 C 文件添加到项目中！另外，还有可能是因为没有被调用的已经定义的函数，或者相关的已经定义的变量没有使用。

- WARNING L15: MULTIPLE CALL TO SEGMENT

该警告表示连接器发现有一个函数可能会被主函数和一个中断服务程序（或者调用中断服务程序的函数）同时调用，或者同时被多个中断服务程序调用。

出现这种警告的原因一般有两种：

第一, 这个函数是不可重入函数, 当该函数运行时可能被打断, 打断后该函数又被再次运行, 从而造成函数内部数据丢失;

第二, 该函数的内部变量数据所占有的内存在 link 时被连接器认为是可覆盖的, 因此在连接时进行了数据覆盖优化, 但是连接器同时发现该函数在运行时被打断后, 其他函数 (如中断服务子程序) 的运行造成了该函数的数据被覆盖。

- WARNING L16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS

定义的函数没有调用而已

## C51

- 常见问题

## 常见问题

### 2.2.2 IAR

IAR Embedded Workbench 是一套用于编译和调试嵌入式系统应用程序的开发工具, 支持汇编、C 和 C++ 语言。它提供完整的集成开发环境, 包括工程管理器、编辑器、编译链接工具和 C-SPY 调试器。

IAR Systems 以其高度优化的编译器而闻名。每个 C/C++ 编译器不仅包含一般全局性的优化, 也包含针对特定芯片的低级优化, 以充分利用您所选芯片的所有特性, 确保较小的代码尺寸。IAR Embedded Workbench 能够支持由不同的芯片制造商生产, 且种类繁多的 8 位、16 位或 32 位芯片。

## IAR

- *IAR ARM*
- *IAR 8051*
  - 常见问题
- *IAR STM8*
  - 常见问题

### IAR ARM

#### IAR 8051

##### 常见问题

Warning[Pa082]: undefined behavior: the order of volatile accesses is undefined in this statement

运算符两边都是 volatile 变量的警告

报警的这条语句中有两个或两个以上被 volatile 定义过的变量。编译器会认为有问题。用 volatile 修饰的变量一般不直接参与运算，volatile 就以为着这个变量在运算过程中有可能已经改变了

Error[e16]: Segment ISTACK (size: 0xc0 align: 0) is too long for segment definition. At least 0xe more bytes needed. The problem occurred while processing the segment placement command

解决：依次打开 Project -> Options -> General Option -> Target，在 Target 标签中找到 “Number of virtual”，原来默认为 16，修改为 8。

#### IAR STM8

```
extern volatile BYTE sppRxStatus;  
extern volatile BYTE sppTxStatus;  
__no_init SPP_RX_STRUCT rxData @ "PM0_XDATA";  
__no_init SPP_TX_STRUCT txData @ "PM0_XDATA";
```

\_\_no\_init 在编程环境中是蓝色的字。

@ 是指定地址，\_\_no\_init 是一个 SEGMENT，是给 LINKER 用的，定义到不初始化的块中去。@ 就是指定地址，这个应该没什么好说的了，大部分编译器都这么用。你应该理解这个吧。你定义全局变量的时候比如 int char; 即使你没有赋值给他，编译器还是会给他一个初始化值 0，编译的时候编译器把他分配到初始化为零的那个 SEGMENT 中去了。编译器默认的有几个块，初始化为零的块，初始化不为零的块，和不初始化的块，你可以定义自己的块，如你的 PM0\_XDATA，这个就是你自己定义的一个块，那你的这个块是个什么属性呢，就是，\_\_no\_init 属性，有了这个属性，编译器只给你分配空间，不给你初始化。

##### 常见问题

用 IAR 打开 STM8 时，出现 “Unable to create configuration 'Debug' using tool chain ‘STM8’”，

出现这个问题的原因是按装的 IAR 不正确，要装 ST for STM8 版本的，而不能用 ST for ARM 版本的

IAR 安装多个不同的版本，会存在点击 eww 文件自动通过上次打开的版本打开工程文件，所以会有这个问题，解决办法是通过点击打开 STM8 对应的 IAR 版本，通过文件导入打开工程文件，之后就可以通过点击打开 STM8 的开发文件了



## STVD

- 简介
- 相关问题

### 简介

### 相关问题

用 STVD+COSMIC 编译工程时出现以下错误（加载的别人的工程）：  
#error clnk Debugdemo.lkf:47 can't openfile crtsi0.sm8  
#error clnk Debugdemo.lkf:60 can't openfile libis0.sm8  
#error clnk Debugdemo.lkf:61 can't openfile libm0.sm8

解决方法：

打开 STVD 软件，选择 Tools-> Options -> Directories -> Show Directories for 选择:Libraryfiles 将 D:program file-sCOSMICCXSTM8\_32KLib 添加进去，如安装在其它目录，添加相应的目录即可。

关于启动 STVD 编译环境，启动编译连接出现错误 Error creating process for executable cxstm8 系统找不到指定的文件解决方法

第一步：先点击截图里面的第一个文件来安装，安装过程中，会有很多的提示，直接 NEXT，可以。

第二步：点击第二个文件，找到刚才 cxstm8\_32k.exe。安装路径。点击启动应用按钮，即完成安装。

再来启动 STVD 软件，点击 project 项目中的 settings. 如下截图所示：在 project specific toolset p: 左侧打上小勾，然后单击找到刚才按照 CXSTM8\_32K 的安装路径，即可。

### 2.2.3 STM32CubeIDE

STM32CubeIDE 是 ST 公司基于 Eclipse/CDT 框架和 GUN GCC 工具链制作的免费 IDE，并集成了 STM32CubeMX。可以实现 STM32 系列芯片的外围设备配置、代码生成、代码编辑、代码编译、在线调试，并且支持数百个 Eclipse 现有插件。

## STM32CubeIDE



STM32 IDEs



- 简介
  - 参数说明

## 简介

STM32CubeIDE 是 ST 官方提供的免费软件开发工具，也是 STM32Cube 生态系统的一员大将。它基于 Eclipse®/CDT 框架，GCC 编译工具链和 GDB 调试工具，支持添加第三方功能插件。同时，STM32CubeIDE 还集成了部分 STM32CubeMX 和 STM32CubeProgrammer 的功能，是一个“多合一”的 STM32 开发工具。

## 参数说明

## 2.2.4 Embedded Studio

SEGGER RISC-V

Embedded Studio 是一款使用 C 和 C++ 语言的精简的、专业的嵌入式开发工具。它配备了强大的项目构建和管理系统，具有代码自动完成和折叠功能的源代码编辑器，以及分包管理系统用于下载和安装开发板和器件的软件支持包。它还包括 SEGGER 高度优化的运行时库 emRun、浮点库 emFloat 以及 SEGGER 的智能链接器，所有这些都为资源有限的嵌入式系统量身定做。其内置调试器包括了所有必要的功能，与 J-Link 配合使用，提供了卓越的性能和稳定性。

## 2.2.5 Summary

- *Comparison*

### Comparison

#### C51 vs EW8051

#### MDK vs EWARM

一般来说，如果主要是采用 C，并且也不会有太多的 library 需要连接，MDK 和 IAR 都能胜任。不过这种情形就比较推荐 IAR，因为其非常简洁，上手也快，代码层次也能清晰明了。

如果主要是采用 C++，并且用到很多特性，或是需要有多多个工程进行协作，那么注定只能选择 MDK，只不过这样就一定要每个文件最后加上新的空行了。

- 1、MDK 不支持层叠文件夹，在文件夹的下一级中必须为文件；IAR 支持层叠，可以比较方便管理代码，理清层次。
- 2、MDK 连接 library，直接添加到文件夹即可；IAR 则需要从工程中选项中设置。这应该不算什么问题，毕竟大多数 IDE 都是这么做的，但最让人很郁闷的是，IAR 不能采用相对路径。比如../MUF/MUF.LIB 在编译时，就会连接到别的目录，只能采用 d:/MUF/MUF.lib 绝对路径的形式。
- 3、MDK 支持 `dynamic_cast<>` 运算符，而 IAR 文档中明确表示不支持。如果在 IAR 中强行使用该运算符，则编译会报错：Error[Pe020]: identifier "dynamic\_cast" is undefined
- 4、MDK 默认只创建工程，工作区是不会直接创建。如果想多个工程聚合，则首先需要创建一个 multi 的工作区，然后再添加相应的工程。IAR，默认是创建工程和工作区，如果想多个工程并存，直接添加即可。相比之下，MDK 创建工程的文件比较少，而 IARM 创建工程生成的文件比较多。
- 5、MDK 编译时，只有 level 的选择；IAR 有 debug 和 Release 的快速选择
- 6、默认状态，MDK 的工具栏功能比较多，有点繁杂；IAM 的比较简洁，但相对，也比较单薄。
- 7、MDK 的 C++ 有 `std::` 这个命名空间；IAR 下面的所有容器和算法，都不采用 `std` 命名空间
- 8、MDK 的程序文件，最后必须要有一个新的空行，否则会有编译警告：warning: #1-D: last line of file ends without a newline

## 2.3 Editor

### 2.3.1 VSCode

- 简介
- 功能

#### 简介

VisualStudioCode 是由微软研发的一款 开源 的跨平台代码编辑器，目前是前端 (网页) 开发使用最多的一款软件开发工具。

#### 功能

#### 快捷键

- 折叠所有区域代码的快捷：ctrl + k ctrl + 0 ;( 注意这个是零，不是欧)
- 展开所有折叠区域代码的快捷：ctrl +k ctrl + J ;

调试常见的快捷键

- F9 打开和停止调试断点
- F11 单步调试
- F5 启动调试

### 2.3.2 Source Insight

- 简介

#### 简介

Source Insight 是一个面向项目开发的程序编辑器和代码浏览器，它拥有内置的对 C/C++, C# 和 Java 等程序的分析。能分析源代码并在工作的同时动态维护它自己的符号数据库，并自动显示有用的上下文信息。

### 2.3.3 Markdown

Markdown 是一种轻量级标记语言，创始人为约翰·格鲁伯（John Gruber）。

#### UTF-8

UTF-8（8-bit Unicode Transformation Format）是一种针对 Unicode 的可变长度字符编码，又称万国码，由 Ken Thompson 于 1992 年创建。现在已经标准化为 RFC 3629。UTF-8 用 1 到 6 个字节编码 Unicode 字符。用在网页上可以统一页面显示中文简体繁体及其它语言（如英文，日文，韩文）。

事实证明，对可以用 ASCII 表示的字符使用 UNICODE 并不高效，因为 UNICODE 比 ASCII 占用大一倍的空间，而对 ASCII 来说高字节的 0 对他毫无用处。为了解决这个问题，就出现了一些中间格式的字符集，他们被称为通用转换格式，即 UTF（Unicode Transformation Format）。常见的 UTF 格式有：UTF-7, UTF-7.5, UTF-8, UTF-16, 以及 UTF-32。

如果 UNICODE 字符由 2 个字节表示，则编码成 UTF-8 很可能需要 3 个字节。而如果 UNICODE 字符由 4 个字节表示，则编码成 UTF-8 可能需要 6 个字节。用 4 个或 6 个字节去编码一个 UNICODE 字符可能太多了，但很少会遇到那样的 UNICODE 字符。UTF-8 编码规则：如果只有一个字节则其最高二进制位为 0；如果是多字节，其第一个字节从最高位开始，连续的二进制位值为 1 的个数决定了其编码的字节数，其余各字节均以 10 开头。

- 优点：UTF-8 编码可以通过屏蔽位和移位操作快速读写。字符串比较时 `strcmp()` 和 `wcscmp()` 的返回结果相同，因此使排序变得更加容易。字节 FF 和 FE 在 UTF-8 编码中永远不会出现，因此他们可以用来表明 UTF-16 或 UTF-32 文本（见 BOM）UTF-8 是字节顺序无关的。它的字节顺序在所有系统中都是一样的，因此它实际上并不需要 BOM。
- 缺点：你无法从 UNICODE 字符数判断出 UTF-8 文本的字节数，因为 UTF-8 是一种变长编码它需要用 2 个字节编码那些用扩展 ASCII 字符集只需 1 个字节的字符 ISO Latin-1 是 UNICODE 的子集，但不是 UTF-8 的子集 8 位字符的 UTF-8 编码会被 email 网关过滤，因为 internet 信息最初设计为 7 位 ASCII 码。因此产生了 UTF-7 编码。UTF-8 在它的表示中使用值 100xxxxx 的几率超过 50%，而现存的实现如 ISO 2022, 4873, 6429, 和 8859 系统，会把它错认为是 C1 控制码。因此产生了 UTF-7.5 编码。



### 3.1 ARM

- 函数调用
- 中断处理
- 代码密度

### 3.2 函数调用

函数调用是通过栈来实现的，在栈中存放着该函数的局部变量。函数在调用的时候都是在栈空间上开辟一段空间以供函数使用。

栈帧结构就是在程序运行中发生函数调用时，会产生一个调用栈，调用栈里面存储四类数据，分别为：函数参数、程序返回地址、基址寄存器 EBP 的值以及局部变量。

- ESP：栈指针寄存器 (extended stack pointer)，放着一个指针，该指针永远指向系统栈最上面一个栈帧（栈帧不理解没关系）的栈顶。
- EBP：基址指针寄存器 (extended base pointer)，放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。
- EIP：指令寄存器 (extended instruction pointer)，存储 CPU 下一个要执行的指令的地址。

### 3.3 中断处理

ARM 处理器中断处理过程与 8051 单片机中断处理过程基本一样，区别在于 arm 处理器可能好几个中断共用一个中断向量地址，所以需要在中断程序中判断是哪个中断源，同时软件清除中断标志位。

### 3.4 代码密度

对于内存受限的嵌入式芯片（包括 MCU 和成本要求的 AP 类芯片）来说代码密度非常重要。同样功能的程序，如果代码密度过大，就可能导致因 ROM 空间装载不下而无法使用。在嵌入式领域中，代码密度是最重要的指标之一。代码密度主要由指令集、ABI、编译器、Runtime 库、程序代码五个部分决定。处在金字塔的越底端，说明该因素的越底层，更新的频率越小，但辐射和影响范围却越广。



- 指令集
- ABI
- 编译器



- *Runtime*
- 程序代码

### 3.4.1 指令集

指令集是代码密度最根本的决定性因素，它决定了一个操作在最优的情况下需要编译成多少位宽的编码。很多体系结构比如 ARM、RISC-V、C-SKY 都是 16 位指令、32 位指令混编的，同样的一条指令，如果能够被编译成 16 位指令，那么它显然比编译成 32 位指令占用更小的空间；再比如，一个乘累加的操作，如果指令集中存在乘累加指令，那么它只需要一条指令来实现乘累加操作，如果没有则需要至少两条指令来完成相同的操作，假设指令都是 32 位的，显然一条指令将占用更少的空间。由于指令集的编码空间是有限的，所以指令集设计的核心是将哪些指令（包括指令操作数的范围）放到编码空间当中，就像一个商场的店面是有限的，当我们把需求最广的商家引进来时，商场的销量就会达到最高。

### 3.4.2 ABI

ABI（Application Binary Interface）是二进制级别的协议，它指导着编译器如何生成代码和二进制程序，同样也指导着用户如何写汇编代码。它主要包含函数调用约定（calling convention）、数据的对齐方式等内容。其中对代码密度影响最大的就是函数调用约定，它规定了堆栈寄存器、链接寄存器、哪些寄存器寄存器需要在函数头尾保存和恢复、哪些寄存器可以作为参数寄存器等，还有一些特殊用途的寄存器。大部分特殊寄存器都是会被高频使用的，配合指令集设计可以降低代码密度；需要保存和恢复的寄存器个数同样也会影响代码密度。

### 3.4.3 编译器

编译器是开发者最直接接触的工具，也是给开发者体感最强的代码密度影响因素。它对代码密度的影响主要体现在两方面：

- 编译器本身的优化能力，优化能力的强弱是影响编译器产品竞争力的最主要的因素。
- 编译器的使用方法，比如 GCC，除了添加 -Os 之外，还可以添加 -ffunction-sections -fdata-sections -Wl,--gc-sections 来删除没有用到的函数。

### 3.4.4 Runtime

Runtime 库是指程序运行所需的一些基本的函数库，它们一般都是预先编译好，和编译器一起打包发布，是工具链的一部分。

由于这些函数的使用频率较高，一般程序都会用到一部分 Runtime 库的函数，对这些函数做针对性地优化会有比较好的收益。

### 3.4.5 程序代码

开发者写的代码质量也会影响程序的代码密度，虽然编译器能够优化一部分冗余代码，但是并不能保证百分之百的优化，所以开发者也要注意代码的质量。

系统依赖包括实时系统 RTOS，云端环境和通信协议等。

## 4.1 RTOS

### 4.1.1 freeRTOS

FreeRTOS 是一款适用于微控制器的开源实时操作系统，让您可以轻松编写、部署、保护、连接和管理低功耗的小型边缘设备。

FreeRTOS 在 [MIT 开源许可证](#) 下免费分发，包括一个内核和一组不断丰富的软件库，适用于各种行业部门和应用程序。许多半导体厂商产品的 SDK(Software Development Kit—软件开发工具包) 包就使用 FreeRTOS 作为其操作系统，尤其是 WiFi、BLE 这些带协议栈的芯片或模块。

关于 FreeRTOS 开发框架应用场景可访问 [qio framework](#)

- 系统特点
- 任务调度
- 任务间通讯
- 源文件解读
- 系统移植

- 问题总结
- RTOS 对比

### 系统特点

MIT

FreeRTOS 是一个可裁剪小型 RTOS 系统，其特点包括：

- 内核支持抢占式，合作式和时间片调度。
- 提供了一个用于低功耗的 Tickless 模式。
- 系统的组件在创建时可以选择动态或者静态的 RAM，比如任务、消息队列、信号量、软件定时器等。
- FreeRTOS-MPU 支持 Corex-M 系列中的 MPU 单元，如 STM32F429。
- FreeRTOS 系统简单、小巧、易用，通常情况下内核占用 4k-9k 字节的空间。
- 高可移植性，代码主要 C 语言编写。
- 高效的软件定时器。
- 强大的跟踪执行功能。
- 堆栈溢出检测功能。
- 任务数量不限。
- 任务优先级不限。

FreeRTOS 中的任务一共有四种状态分别是运行状态（Running State），就绪状态（Ready State），阻塞状态（Blocked State），挂起状态（Suspended State）

TCB\_t 的全称为 Task Control Block，也就是任务控制块，这个结构体包含了一个任务所有的信息

### 任务调度

FreeRTOS 三种调度方式：抢占式，时间片、合作式，实际应用主要是抢占式调度和时间片调度，合作式调度用到的很少。

抢占式调度（不同优先级）：每个任务都有不同的优先级，任务会一直运行直到被高优先级任务抢占或者遇到阻塞式的 API 函数，比如 `vTaskDelay`。时间片调度（相同优先级）：每个任务都有相同的优先级，任务会运行固定的时间片个数或者遇到阻塞式的 API 函数，比如 `vTaskDelay`，才会执行同优先级任务之间的任务切换。

FreeRTOS 对任务的调度采用时间片（time slicing）的调度方式。时间片，顾名思义，把一段时间等分成了很多个时间段，在每一个时间段保证优先级最高的任务能执行，同时如果几个任务拥有相等的优先级，则它们会轮流使用每个时间段占用 CPU 资源。调度器会在每个时间片结束的时候通过周期中断（tick interrupt）执行一次，选择哪个任务在下一个时间片会运行。

时间片的大小由 configTICK\_RATE\_HZ 这个参数设置。如果 configTICK\_RATE\_HZ 设置为 10HZ，则时间片的大小为 100ms。configTICK\_RATE\_HZ 的值由应用需求决定，通常设为 100HZ（时间片大小相应为 10ms）。

vTaskList( char \* pcWriteBuffer ) 这个函数可以打印出栈名、栈状态、优先级、栈的剩余空间，使用该功能 FreeRTOSConfig.h 要配置：

```
configUSE_TRACE_FACILITY 1
configUSE_STATS_FORMATTING_FUNCTIONS 1
```

任务调度机制是嵌入式实时操作系统的一个重要概念，也是其核心技术。对于可剥夺型内核，优先级高的任务一旦就绪就能剥夺优先级较低任务的 CPU 使用权，提高了系统的实时响应能力。不同于  $\mu$ C/OS-II，FreeRTOS 对系统任务的数量没有限制，既支持优先级调度算法也支持轮转调度算法，因此 FreeRTOS 采用双向链表而不是采用查任务就绪表的方法来进行任务调度。

具有固定优先级调度程序的 RTOS 的核心思想是，应该在具有较低优先级的任务之前安排高优先级任务，但是当两个或多个任务需要协调其工作与全局数据区等共享资源或外围设备时，可能会导致系统出错。

其中一个可能出错的事情就是优先级反转 (priorityinversion)，低优先级任务无意中阻止了具有更高优先级的任务。如果你意识到这个陷阱，这也很容易地避免。但是，如果发现系统的响应性偶尔会出现延迟，则可能是因为优先级反转。使用 Tracealyzer，可以通过绘制任务的响应时间来发现此类延迟。要查看此图中任何极端值的原因，只需双击以显示相应的任务执行跟踪。

## 抢占式调度器

在实际的应用中，不同的任务需要不同的响应时间。例如，在一个应用中需要使用电机，键盘和 LCD 显示。电机比键盘和 LCD 需要更快速的响应，如果我们使用时间片调度，那么电机将无法得到及时的响应，这时抢占式调度是必须的。如果使用了抢占式调度，最高优先级的任务一旦就绪，总能得到 CPU 的控制权。比如，当一个运行着的任务被其它高优先级的任务抢占，当前任务的 CPU 使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了 CPU 的控制权并运行。又比如，如果中断服务程序使一个高优先级的任务进入就绪态，中断完成时，被中断的低优先级任务被挂起，优先级高的那个任务开始运行。每个任务都被分配了不同的优先级，抢占式调度器会获得就绪列表中优先级最高的任务，并运行这个任务。

## 时间片调度器

时间片调度适合用于不要求任务实时响应的情况。

需要给同优先级的任务分配一个专门的列表，用于记录当前就绪的任务，并为每个任务分配一个时间片，也就是需要运行的时间长度，时间片用完了就进行任务切换。在 FreeRTOS 操作系统中只有同优先级任务才会使用时间片调度，另外还需要用户在 FreeRTOSConfig.h 文件中使能宏定义：#define configUSE\_TIME\_SLICING 1

### 任务间通讯

任务通讯的几种方式：queue，semaphores mutexes 和 event groups。

其中 semaphores mutexes 都是基于队列的方式实现，notify 机制和 event groups 最为类似，但是实现方式有较大差异。Notify 机制是在每个任务中添加一个 32 位无符号字符标记，其他任务对该任务的通知。

- 信号量
  - 二值型信号量
  - 互斥型信号量
  - 递归互斥信号量

### 信号量

信号量通过一个计数器控制对共享资源的访问，信号量的值是一个非负整数，所有通过它的线程都会将该整数减一。如果计数器大于 0，则访问被允许，计数器减 1；如果为 0，则访问被禁止，所有试图通过它的线程都将处于等待状态。

- 整型信号量 (integer semaphore)：信号量取值是整数，它可以被多个线程同时获得，直到信号量的值变为 0。
- 记录型信号量 (record semaphore)：每个信号量 s 除一个整数值 value (计数) 外，还有一个等待队列 List，其中是阻塞在该信号量的各个线程的标识。当信号量被释放一个，值被加一后，系统自动从等待队列中唤醒一个等待中的线程，让其获得信号量，同时信号量再减一。
- 二进制信号量 (binary semaphore)：只允许信号量取 0 或 1 值，其同时只能被一个线程获取。

### 二值型信号量

二值信号量相当于长度为 1 的队列，那么计数型信号量就是长度大于 1 的队列，同二值信号量一样，用户不需要关心队列中存储了什么数据，只需要关心队列是否为空即可。

二值型信号量是任务间、任务与中断间同步的重要手段。

- 1、没有优先级继承
- 2、可以在中断中使用
- 3、可以在其他任务释放

## 互斥型信号量

互斥型信号量是任务间资源保护的重要手段。

申明互斥型信号量，在 FreeRTOS 中二值型信号量和互斥型信号量类型完全相同。从功能上二值型信号量用于同步，而互斥型信号量用于资源保护。

不同于二值信号量的是互斥信号量具有优先级继承的特性，可以有效解决优先级反转现象。当一个互斥信号量正在被一个低优先级的任务使用，而此时有个高优先级的任务也尝试获取这个互斥信号量的话就会被阻塞。不过这个高优先级的任务会被低优先级任务的优先级提升到与自己相同的优先级，这个过程就是优先级传承。

- 1、优先级继承
- 2、互斥量不能在中断中使用
- 3、互斥量获取和释放需要再同一个 task 中

## 递归互斥信号量

递归互斥信号量可以看做一个特殊的互斥信号量，已经获取了互斥信号量的任务就不能再次获取这个互斥信号量，但是递归互斥信号量不同，已经获取了递归互斥信号量的任务可以再次获取这个递归互斥信号量，而且次数不限制。并且获取多少次信号量，就需要释放多少次信号量。

## 源文件解读

- *FreeRTOS*
- *FreeRTOS-Plus*

## FreeRTOS

Demo 文件夹里面就是 FreeRTOS 针对不同的 MCU 提供的相关例程，其中就有 ST 的 F1、F4 和 F7 的相关例程。

License 文件夹里面就是相关的许可信息，要用 FreeRTOS 做产品的得仔细看看，尤其是要出口的产品。

Source 文件夹里面就是 FreeRTOS 的源码文件，include 文件夹是一些头文件，移植的时候是需要的，下面的这些.C 文件就是 FreeRTOS 的源码文件。

portable 文件夹里面就是 FreeRTOS 系统和具体的硬件之间的连接桥梁！MemMang 这个文件夹是跟内存管理相关的，我们移植的时候是必须的。

RVDS 文件夹针对不同的架构的 MCU 做了详细的分类，STM32F429 就参考 ARM\_CM4F，打开 ARM\_CM4F 文件夹，里面有两个文件，这两个文件就是我们移植的时候所需要的！

### FreeRTOS-Plus

里面也有 Demo 和 Source, Demo 文件夹里存放的肯定是一些例程, 而 Source 文件夹中存放的并不是 FreeRTOS 系统的源码, 是在这个 FreeRTOS 系统上另外增加的一些功能代码, 比如 CLI、FAT、Trace 等等。

### 系统移植

#### 下载 FreeRTOS

打开文件夹之后有两个文件夹: 一个是 FreeRTOS, 另一个是 FreeRTOS-Plus。打开 FreeRTOS 文件夹: 里面有三个文件夹 Demo, License, Source。

我们进行系统移植主要使用的就是 FreeRTOS 里面的内容, FreeRTOS-Plus 中的内容是一个扩展功能, 和系统内核是没有关系的, 我们在系统的移植的过程中不用管。

FreeRTOS/Source 文件夹下包含的 FreeRTOS 的通用的头文件和 C 文件, 这两部分的文件适用于各种编译器和处理器, 是通用的。需要移植的头文件和 C 文件放在 portbllle 这个文件夹下。

打开 portable 文件夹, 可以看到其中有很多文件夹, 我们需要使用到是如下所示几个。针对 Keil 开发环境我们只需要保留 Keil、RVDS、MemMang 三个文件夹即可, 其他的都可以删除掉。

MemMang 文件夹下存放的是跟内存管理相关的源文件。

### 问题总结

- 优先级翻转
  - 优先级继承
- 死锁
- 内存泄漏

### 优先级翻转

taskA 的任务优先级高于 taskB, 但是由于 taskA 等待请求获取 shareData 资源, taskC 持有 shareData 资源但被优先级高于它的 taskB 抢占阻塞, 于是高优先级的 taskA 被挂起。



## 优先级继承

在高优先级的 taskA 获取资源锁时，将 taskC 的优先级临时提高为 taskA 的优先级，那么上述案例中，taskB 就无法打断 taskC 的执行，因此 taskC 执行完成释放资源锁后，taskA 能及时的进入 ready 状态

优先级恢复流程相对比较简单，在 taskC 使用完，调用释放接口的时候，会执行优先级恢复，此时 taskC 继续恢复其低优先级。

信号量一般是用于同步的，同步的场景上，需要保证优先级高的任务优先执行，做到真正的实时性，优先级继承会打破这个需求。

## 死锁

死锁是两个或多个任务之间的循环依赖。

例如，如果任务 1 已经获得 A，并且被阻止等待 B，而任务 2 先前已获得 B，并且被阻止等待 A，则这两个任务都不会被唤醒。尽管没有更高优先级的任务正在运行，但是当多个任务突然停止执行时，可能是出现死锁问题的明确迹象。同样，死锁的检测是 Tracealyzer 可以展示的内容。

如果希望避免死锁，首先要注意的是，只有当任务试图同时持有两个资源时才会发生死锁。因此：构建代码时，使任何任务在同一时间都不会持有多个共享资源，这样不会产生死锁。

## 内存泄漏

通常不建议在嵌入式软件中进行动态内存分配，但有时会出于各种原因（对或错）进行动态内存分配。问题在于，如果使用它，则必须确保一旦内存块不再使用时，就释放每个已分配的内存块。如果在某些情况下遗漏了这一点，就会出现内存泄漏，并最终耗尽内存。请记住：即使在项目中禁止动态内存分配，也可能有第三方软件库或外部开发团队在不知情的情况下使用动态内存分配。

如果内存泄漏只是偶尔发生，那么它就特别危险，因为在功能测试期间很容易错过“缓慢”的内存泄漏，但在部署单元一段时间后，可能会导致严重错误。考虑到许多嵌入式系统的长期运行特性，以及一些安全关键系统可能存在的致命或严重故障，内存泄漏是绝对不希望在软件中出现的一个错误。

ARM 对嵌入式操作系统进行了顶层设计，不同的操作系统要对他进行适配，这样更换操作系统就比较方便了，使用 ARM 提供的 API 编写的应用层程序，更换操作系统后是不需要修改的。

## RTOS 对比

- 对比 *uCOS-III*

– 相关问题

## 对比 uCOS-III

从文件数量上来看 FreeRTOS 要比 uC/OSII 和 uC/OSIII 小的多。

uCOS-III 中所有的内核对象（如任务控制块、消息队列、信号量等）都是静态创建的，需要用户提供。FreeRTOS 中的内核对象支持动态和静态两种创建方法。

为了实现中断和任务的同步，需要在中断中进行 post 操作，uC/OS-III 为了减少中断执行的时间，提高系统中断响应的实时性，设计了 OS\_tickTask 和 OS\_IntQTask，这样原本在中断里需要进行的一些较为耗时的操作就被放到了任务级代码中执行了。而 FreeRTOS 并没有这样的设计。

在 FreeRTOS 的 PendSV 中断中，它会计算就绪的最高优先级的任务，再去进行上下文切换。而 uC/OS-III 在触发 PendSV 中断前，会计算好已就绪的最高优先级的任务，放在 OSTCBHighRdyPtr 中，这样在 PendSV 中断中就不用计算就绪的最高优先级的任务是谁了。所以 uC/OS-III 中 PendSV 中断的执行时间更短，这有利于提高系统的实时性。

uCOS-III 的任务操作句柄就是任务控制块 TCB 的指针。FreeRTOS 中单独设置了任务操作句柄这种数据类型，它实质上也是 TCB 的指针。表面上看，多此一举，但其实这种设计对用户是友好的，用户不需要了解 TCB 这种内核数据结构的存在，就可以操作任务了。

uCOS-III 内核中的链表大多是不循环的双向链表（有头有尾），在插入和删除操作时，要考虑特殊情况（比如插入表头、插入表尾等特殊情况）。

而 FreeRTOS 内核中的链表为双向循环链表，并引入了 xListEnd 保证了链表永远非空，所以每个元素的插入和删除都是作为表中的一般元素（非表头和表尾）进行的，操作效率要比 uC/OS-III 高一些。

FreeRTOS 功能更丰富、更易用；uC/OS-III 的实时性更好、效率更高、健壮性更好。

其实 RTOS 最主要的功能就是任务调度，其它功能都可以自己开发，难度不大。单独从任务调度器的角色出发去对比这两个 RTOS，我觉得 uC/OS-III 更漂亮、更优秀。

uC/OS-III 通过的安全认证比 FreeRTOS 要多，FreeRTOS 的代码书写是不符合一些标准的。在 FreeRTOS 的基础上建立了另外两个 RTOS：SafeRTOS、OpenRTOS，它们具有更好的安全性，通过了更多的检验和标准，但是与 FreeRTOS 不一样，需要收费。

## 相关问题

μC/OS 2.86 任务卡死在低优先级任务出不来，高优先级任务不执行，后来从 Micrium 下载 μC/OS 2.91 从里面 what's new.pdf 里面查到对 Cortex-M3 有问题（中断优先级大小顺序问题），已修正，然后用高版本的果然没问题。

然后在下一个项目里面使用了 FreeRTOS，感觉跟 μC/OS 差不多，只是任务栈消耗的稍大。我用的 IAR，里面有 μC/OS、FreeRTOS 插件，可以在运行的时候看到任务栈历史最大使用和当前使用，以及 CPU 负载率等等很重要的信息。

NVIC\_PriorityGroup\_4 抢占优先级的要比“MAX”更大，而比“LOWEST”更小

```
configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY=5  
configLIBRARY_LOWEST_INTERRUPT_PRIORITY=15
```

FreeRTOS 中数值越大优先级越高，这种优先级可以成为逻辑优先级。Cortex M3/M4 中断中，数值越大优先级越低，这种优先级成为中断优先级。两者相反，所以才会出现比“MAX”更大而比“LOWEST”更小的情况。

## 4.2 云平台

- 阿里云
- 华为云
- 私有云

### 4.2.1 阿里云

### 4.2.2 华为云

### 4.2.3 私有云

运维监控: 当项目的业务越来越庞大，人为去运维服务器已经是不可能了，需要把运维尽量自动化，部署 open-falcon、Zabbix 等运维监控系统，监控服务器网络、系统、服务等状态，出现问题及时报警，通知运维管理员，管理员接到通知尽快解决问题，保证服务器业务的正常运营。

文件监控: 使用 inotify、FileMonitor 等开源文件监控系统，监听系统中重要的敏感文件，常见的用户、密码、权限、主机系统、网络配置、服务程序配置等文件：passwd、shadow、group、hosts 等。同样，可以使用 shell 脚本来完成。

进程监控: 使用 monit、supervisord 等开源进程监控系统，监听 Web 等服务进程是正常运行，也可以使用脚本周期性查看进程是否正在运行。

日志监控: 使用 log\_monitor、Log Watcher 等开源日志监控系统，监听 Web 服务等日志信息，对高频 IP 进行报警。如果，日志监控系统不存在自定义的功能需求，可以使用脚本来完成，获取我们关心的日志信息。

漏洞扫描: 使用 openvas、nessus 等漏洞扫描系统，定期更新漏洞库与扫描插件，也定期扫描服务器所在的网络，及时发现网络上的脆弱点以及服务器等主机存在的漏洞，修复暴露出来的问题，避免漏洞被入侵者利用，入侵网络系统。

防火墙: 在服务器上或者在网关上配置防护墙，控制网络流量，过滤掉不必要的网络数据报文，之允许服务器上服务相关的流量通过。这里，把 ping 流量禁止掉，可以隐藏一些操作系统类型信息。当然，把防火墙部署在单独的网关上面，可以减轻服务器的负担，服务器可以使用全部资源支持当前运营的服务。

入侵防御: 把 snort 系统配置成入侵防御的模式，系统直连在服务器的网络路线上，充当服务器的安全网关，抵御网络攻击。当然，这样会对服务器的网络性能产生影响。

入侵检测: 安装 snort 轻量级入侵检测系统, 部署在网络的旁路, 既可以监听流过的网络流量是否存在入侵攻击, 也不会影响服务器的网络性能。

### 常用系统命令

Vmstat、sar、iostat、netstat、free、ps、top

利用 sar 命令监控系统 CPU

sar 对系统每方面进行单独统计, 但会增加系统开销, 不过开销可以评估, 对系统的统计结果不会有很大影响。下面是 sar 命令对某个系统的 CPU 统计输出:

```
[root@webserver ~]# sar -u 3 5
```

输出解释如下:

- %user 列显示了用户进程消耗的 CPU 时间百分比。
- %nice 列显示了运行正常进程所消耗的 CPU 时间百分比。
- %system 列显示了系统进程消耗的 CPU 时间百分比。
- %iowait 列显示了 IO 等待所占用的 CPU 时间百分比
- %steal 列显示了在内存相对紧张的环境下 pagein 强制对不同的页面进行的 steal 操作。
- %idle 列显示了 CPU 处在空闲状态的时间百分比。

systemctl 是 CentOS7 的服务管理工具中主要的工具, 它融合之前 service 和 chkconfig 的功能于一体。

### chkconfig

chkconfig [--add][--del][--list][系统服务] 或 chkconfig [--level <levels 等级代号>][系统服务][on/off/reset]

使用范例:

chkconfig --list # 列出所有的系统服务  
chkconfig --add httpd # 增加 httpd 服务  
chkconfig --del httpd # 删除 httpd 服务  
chkconfig --level 2345 httpd on # 设置 httpd 在运行级别为 2、3、4、5 的情况下都是 on (开启) 的状态  
chkconfig --list # 列出系统所有的服务启动情况  
chkconfig --list mysqld # 列出 mysqld 服务设置情况  
chkconfig --level 35 mysqld on # 设定 mysqld 在等级 3 和 5 为开机运行服务, --level 35 表示操作只在等级 3 和 5 执行, on 表示启动, off 表示关闭  
chkconfig mysqld on # 设定 mysqld 在各等级为 on, “各等级” 包括 2、3、4、5 等级

## 4.3 Protocol

- 网络通信

### 4.3.1 网络通信

#### LwIP

LwIP(Light weight IP) 用少量的资源消耗 (RAM) 实现一个较为完整的 TCP/IP 协议栈，其中“完整”主要指的是 TCP 协议的完整性，实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用，LwIP 也可以在没有操作系统的情况下独立运行。

LwIP 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用，它只需十几 KB 的 RAM 和 40K 左右的 ROM 就可以运行，这使 LwIP 协议栈适合在低端的嵌入式系统中使用。

LwIP 具有主要特性：

- 1) 支持 ARP 协议（以太网地址解析协议）。
- 2) 支持 ICMP 协议（控制报文协议），用于网络的调试与维护。
- 3) 支持 IGMP 协议（互联网组管理协议），可以实现多播数据的接收。
- 4) 支持 UDP 协议（用户数据报协议）。
- 5) 支持 TCP 协议（传输控制协议），包括阻塞控制、RTT 估算、快速恢复和快速转发。
- 6) 支持 PPP 协议（点对点通信协议），支持 PPPoE。
- 7) 支持 DNS（域名解析）。
- 8) 支持 DHCP 协议，动态分配 IP 地址。
- 9) 支持 IP 协议，包括 IPv4、IPv6 协议，支持 IP 分片与重装功能，多网络接口下的数据包转发。
- 10) 支持 SNMP 协议（简单网络管理协议）。
- 11) 支持 AUTOIP，自动 IP 地址配置。
- 12) 提供专门的内部回调接口 (Raw API)，用于提高应用程序性能。
- 13) 提供可选择的 Socket API、NETCONN API (在多线程情况下使用)。

### MQTT

MQTT (Message Queuing Telemetry Transport, 消息队列遥测传输协议), 是一种基于发布/订阅 (publish/subscribe) 模式的”轻量级”通讯协议, 该协议构建于 TCP/IP 协议上, 由 IBM 在 1999 年发布。MQTT 最大优点在于, 可以以极少的代码和有限的带宽, 为连接远程设备提供实时可靠的消息服务。作为一种低开销、低带宽占用的即时通讯协议, 使其在物联网、小型设备、移动应用等方面有较广泛的应用。

MQTT 是一个基于客户端-服务器的消息发布/订阅传输协议。MQTT 协议是轻量、简单、开放和易于实现的, 这些特点使它适用范围非常广泛。在很多情况下, 包括受限的环境中, 如: 机器与机器 (M2M) 通信和物联网 (IoT)。其在, 通过卫星链路通信传感器、偶尔拨号的医疗设备、智能家居、及一些小型化设备中已广泛使用。

## CHAPTER 5

---

### 项目管理

---





## 6.1 Git

### 6.1.1 分支管理

创建空白分支，查看提交信息

```
git checkout --orphan emptybranch
git log --graph --pretty=oneline --abbrev-commit
git log --oneline --graph --decorate --all
```

### 合并分支

当一个正在开发的 **feature**、**bugfix** 分支完成后，需要提交 **Merge Request** 请求合并入 **master** 或其他分支。解决冲突的一种方法是将 **master** 分支合入到当前分支。

**rebase** 在 **git** 中是一个非常有魅力的命令，使用得当会极大提高自己的工作效率；相反，如果乱用，会给团队中其他人带来麻烦。

它的作用简要概括为：可以对某一段线性提交历史进行编辑、删除、复制、粘贴；因此，合理使用 **rebase** 命令可以使我们的提交历史干净、简洁！

### 6.1.2 补丁管理

生成补丁文件

```
git format-patch 8d3ce02436066e3 --stdout > test.patch
git format-patch -2 //生成距离HEAD最近的2个patch
git diff > test.patch
git apply test.patch
```

## 7.1 DevOps

DevOps 是一个完整的面向 IT 运维的工作流，以 IT 自动化以及持续集成（CI）、持续部署（CD）为基础，来优化程式开发、测试、系统运维等所有环节。

DevOps（英文 Development 和 Operations 的组合）是一组过程、方法与系统的统称，用于促进开发（应用程序/软件工程）、技术运营和质量保障（QA）部门之间的沟通、协作与整合。它的出现是由于软件行业日益清晰地认识到：为了按时交付软件产品和服务，开发和运营工作必须紧密合作

DevOps 的引入能对产品交付、测试、功能开发和维护（包括 —— 曾经罕见但如今已屡见不鲜的 —— “热补丁”）起到意义深远的影响。在缺乏 DevOps 能力的组织中，开发与运营之间存在着信息“鸿沟”—— 例如运营人员要求更好的可靠性和安全性，开发人员则希望基础设施响应更快，而业务用户的需求则是更快地将更多的特性发布给最终用户使用。这种信息鸿沟就是最常出问题的地方。

一般而言，当企业希望将原本笨重的开发与运营之间的工作移交过程变得流畅无碍，他们通常会遇到以下三类问题：

- 发布管理问题：很多企业有发布管理问题。他们需要更好的发布计划方法，而不止是一份共享的电子数据表。他们需要清晰了解发布的风险、依赖、各阶段的入口条件，并确保各个角色遵守既定流程行事。
- 发布/部署协调问题：有发布/部署协调问题的团队需要关注发布/部署过程中的执行。他们需要更好地跟踪发布状态、更快地将问题上升、严格执行流程控制和细粒度的报表。
- 发布/部署自动化问题：这些企业通常有一些自动化工具，但他们还需要以更灵活的方式来管理和驱动自动化工作 —— 不必要将所有手工操作都在命令行中加以自动化。理想情况下，自动化工具应该能够在非生产环境下由非运营人员使用。

全面自动化：部署、升级、扩展、维护、数据卫生、测试、监测、安全和策略管理。

- [工具链](#)

### 7.1.1 工具链

现将工具类型及对应的不完全列举整理如下：

- 代码管理 (SCM)：GitHub、GitLab、BitBucket、SubVersion
- 构建工具：Ant、Gradle、maven
- 自动部署：Capistrano、CodeDeploy
- 持续集成 (CI)：Bamboo、Hudson、Jenkins
- 配置管理：Ansible、Chef、Puppet、SaltStack、ScriptRock GuardRail
- 容器：Docker、LXC、第三方厂商如 AWS
- 编排：Kubernetes、Core、Apache Mesos、DC/OS
- 服务注册与发现：Zookeeper、etcd、Consul
- 脚本语言：python、ruby、shell
- 日志管理：ELK、Logentries
- 系统监控：Datadog、Graphite、Icinga、Nagios
- 性能监控：AppDynamics、New Relic、Splunk
- 压力测试：JMeter、Blaze Meter、loader.io
- 预警：PagerDuty、pingdom、厂商自带如 AWS SNS
- HTTP 加速器：Varnish
- 消息总线：ActiveMQ、SQS
- 应用服务器：Tomcat、JBoss
- Web 服务器：Apache、Nginx、IIS
- 数据库：MySQL、Oracle、PostgreSQL 等关系型数据库；cassandra、mongoDB、redis 等 NoSQL 数据库
- 项目管理 (PM)：Jira、Asana、Taiga、Trello、Basecamp、Pivotal Tracker

在工具的选择上，需要结合公司业务需求和技术团队情况而定。Ansible 是用于在可重复的方式将应用程序部署到远程节点和配置服务器的开源工具。它为您提供了使用推送模型设置推送多层应用程序和应用程序工件的通用框架，但如果愿意，您可以将其设置为主客户端。Ansible 是建立在 playbooks，你可以应用于各种各样的系统部署你的应用程序。

## 8.1 regression

回归分析 (regression analysis) 是统计学的一个概念，回归是根据样本研究其两个（或多个）变量之间的依存关系，对其趋势的一个分析预测。

回归方法是一种对数值型连续随机变量进行预测和建模的监督学习算法。回归任务的特点是标注的数据集具有数值型的目标变量。

经典的线性回归模型主要用来预测一些存在着线性关系的数据集。回归模型可以理解为：存在一个点集，用一条曲线去拟合它分布的过程。如果拟合曲线是一条直线，则称为线性回归。如果是一条二次曲线，则被称为二次回归。线性回归是回归模型中最简单的一种。

在线性回归中：

- （1）假设函数是指，用数学的方法描述自变量和因变量之间的关系，它们之间可以是一个线性函数或非线性函数。假设函数为  $Y' = wX + b$ ，其中， $Y'$  表示模型的预测结果（预测房价），用来和真实的  $Y$  区分，模型要学习的参数即： $w, b$ 。
- （2）损失函数是指，用数学的方法衡量假设函数预测结果与真实值之间的误差。这个差距越小预测越准确，而算法的任务就是使这个差距越来越小。建立模型后，我们需要给模型一个优化目标，使得学到的参数能够让预测值  $Y'$  尽可能地接近真实值  $Y$ 。这个实值通常用来反映模型误差的大小。不同问题场景下采用不同的损失函数。对于线性模型来讲，最常用的损失函数就是均方误差（Mean Squared Error, MSE）。
- （3）优化算法：神经网络的训练就是调整权重（参数）使得损失函数值尽可能得小，在训练过程中，将损失函数值逐渐收敛，得到一组使得神经网络拟合真实模型的权重（参数）。所以，优化算法的最终目

标是找到损失函数的最小值。而这个寻找过程就是不断地微调变量  $w$  和  $b$  的值，一步一步地试出这个最小值。常见的优化算法有随机梯度下降法（SGD）、Adam 算法等等

## 8.2 activation

如果不用 Activation Function，每一层输出都是上层输入的线性函数，无论多少层输出都是输入的线性组合，与没有隐藏层效果相当，这是最原始的感知机（Perceptron）；激活函数给神经元引入了非线性因素，使得神经网络可以任意逼近任何非线性函数，这样神经网络就可以应用到众多的非线性模型中。

### 8.2.1 Sigmoid

$$f(x) = \text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

Logistic Sigmoid (Sigmoid) 给神经网络引进了概率的概念，值域  $[0,1]$ ，可以将实数映射到  $[0,1]$  区间用做二分类。特别的，如果是非常大的负数，那么输出就是 0；如果是非常大的正数，输出就是 1。

在特征相差比较复杂或是相差不是特别大时效果比较好。它的导数是非零的，并且很容易计算（是其初始输出的函数）。

在深度神经网络中梯度反向传递时导致梯度爆炸和梯度消失，其中梯度爆炸发生的概率非常小，而梯度消失发生的概率比较大。Sigmoid 的 output 不是 0 均值（即 zero-centered）。这是不可取的，因为这会导致后一层的神经元将得到上一层输出的非 0 均值的信号作为输入。其解析式中含有幂运算，计算机求解时相对来讲比较耗时。

### 8.2.2 Tanh

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

双曲正切函数（Tanh），值域  $[-1,1]$ ，逐渐取代 Sigmoid 函数作为标准的激活函数，为奇函数（关于原点对称），解决了 Sigmoid 函数的不是 zero-centered 输出问题。为了解决学习缓慢和/或梯度消失问题，可以使用这个函数的更加平缓的变体（log-log、softsign、symmetrical sigmoid 等等）

它是完全可微分的，反对称，对称中心在原点。tanh 在特征相差明显时的效果会很好，在循环过程中会不断扩大特征效果。

实际上，tanh 函数只是规模变化的 sigmoid 函数，将 sigmoid 函数值放大 2 倍之后再向下平移 1 个单位： $\tanh(x) = 2\text{sigmoid}(2x) - 1$ 。

### 8.2.3 ReLU

修正线性单元 (Rectified linear unit, ReLU) 是神经网络中最常用的激活函数用于隐层神经元输出, 输入信号  $\leq 0$  时, 输出都是 0,  $> 0$  的情况下输出等于输入, 输出不是 zero-centered, 并不是全区间可导。

它保留了 step 函数的生物学启发 (只有输入超出阈值时神经元才激活), 不过当输入为正的时候, 导数不为零, 从而允许基于梯度的学习 (尽管在  $x=0$  的时候, 导数是未定义的)。使用这个函数能使计算变得很快, 因为无论是函数还是其导数都不包含复杂的数学运算。

解决了 gradient vanishing 问题 (在正区间) 计算速度非常快, 只需要判断输入是否大于 0 收敛速度远快于 sigmoid 和 tanh 然而, 当输入为负值的时候, ReLU 的学习速度可能会变得很慢, 甚至使神经元直接无效, 因为此时输入小于零而梯度为零, 从而其权重无法得到更新, 在剩下的训练过程中会一直保持静默。

一个非常大的梯度流过一个 ReLU 神经元, 更新过参数之后, 这个神经元再也不会对任何数据有激活现象, 需要小心设置 learning rate

## 8.3 optimizer

### 8.3.1 梯度下降法

梯度下降法 (Gradient Descent) 最基本的一类优化器, 目前主要分为三种: 标准梯度下降法 (GD, Gradient Descent)、随机梯度下降法 (SGD, Stochastic Gradient Descent) 及批量梯度下降法 (BGD, Batch Gradient Descent)。

标准梯度下降法主要有两个缺点:

训练速度慢: 在应用于大型数据集中, 每输入一个样本都要更新一次参数, 且每次迭代都要遍历所有的样本。会使得训练过程及其缓慢, 需要花费很长时间才能得到收敛解。局部最优解: 所谓的局部最优解就是鞍点, 落入鞍点, 梯度为 0, 使得模型参数不在继续更新。BGD 每次权值调整发生在批量样本输入之后, 而不是每输入一个样本就更新一次模型参数, 这样就会大大加快训练速度

批量梯度下降法比标准梯度下降法训练时间短, 且每次下降的方向都很正确。

### 8.3.2 SGD

SGD 根据的是一整个数据集的随机一部分

虽然 SGD 需要走很多步的样子, 但是对梯度的要求很低 (计算梯度快)。而对于引入噪声, 大量的理论和实践工作证明, 只要噪声不是特别大, SGD 都能很好地收敛。

应用大型数据集时, 训练速度很快。比如每次从百万数据样本中, 取几百个数据点, 算一个 SGD 梯度, 更新一下模型参数。相比于标准梯度下降法的遍历全部样本, 每输入一个样本更新一次参数, 要快得多。

SGD 在随机选择梯度的同时会引入噪声, 使得权值更新的方向不一定正确。此外, SGD 也没能单独克服局部最优解的问题。

### 8.3.3 动量优化法

动量优化方法是在梯度下降法的基础上进行的改变，具有加速梯度下降的作用。一般有标准动量优化方法 Momentum、NAG (Nesterov accelerated gradient) 动量优化方法。

使用动量 (Momentum) 的随机梯度下降法 (SGD)，主要思想是引入一个积攒历史梯度信息动量来加速 SGD。

牛顿加速梯度 (NAG, Nesterov accelerated gradient) 算法，是 Momentum 动量算法的变种。

### 8.3.4 自适应学习率优化算法

自适应学习率优化算法针对于机器学习模型的学习率，传统的优化算法要么将学习率设置为常数要么根据训练次数调节学习率。极大忽视了学习率其他变化的可能性。然而，学习率对模型的性能有着显著的影响，因此需要采取一些策略来想办法更新学习率，从而提高训练速度。

目前的自适应学习率优化算法主要有：AdaGrad 算法，RMSProp 算法，Adam 算法以及 AdaDelta 算法。

### 8.3.5 Adam 算法

首先，Adam 中动量直接并入了梯度一阶矩（指数加权）的估计。其次，相比于缺少修正因子导致二阶矩估计可能在训练初期具有很高偏置的 RMSProp，Adam 包括偏置修正，修正从原点初始化的一阶矩（动量项）和（非中心的）二阶矩估计。

目前，最流行并且使用很高的优化器（算法）包括 SGD、具有动量的 SGD、RMSprop、具有动量的 RMSProp、AdaDelta 和 Adam。在实际应用中，选择哪种优化器应结合具体问题；同时，也优化器的选择也取决于使用者对优化器的熟悉程度（比如参数的调节等等）。

## 8.4 LSTM

循环神经网络 RNN (Recurrent Neural Network) 是一种用于处理序列数据的神经网络。RNN 都具有一种重复神经网络模块的链式形式。在标准 RNN 中，重复的神经网络结构往往也非常简单，例如单个 tanh 层，它对于短期的输入非常敏感。

LSTM (long short term memory)，是一种特殊的 RNN，主要是为了解决长序列训练过程中的梯度消失和梯度爆炸问题。LSTM 不同之处在于它的重复模块结构不同，是 4 个以特殊方式进行交互的神经网络。相比普通的 RNN，能用 RNN 实现的东西，LSTM 都能做，LSTM 还能够在更长的序列中有更好的表现。

LSTM 的关键是怎样控制长期状态  $c$ ，思路是使用三个控制开关。第一个开关，负责控制继续保存长期状态  $c$ ；第二个开关，负责控制把即时状态输入到长期状态  $c$ ；第三个开关，负责控制是否把长期状态  $c$  作为当前的 LSTM 的输出。

LSTM 的输入向量有三个：当前时刻网络的输入值、上一时刻 LSTM 的输出值、以及上一时刻的单元状态；LSTM 的输出向量有两个：当前时刻 LSTM 输出值、和当前时刻的单元状态。





芯片选型验证 Xindex，开源资源集锦 OS-Q，平台服务工具 STOPstop