相见恨晚！C 语言的驱动法编程详细解析（超多代码案例）

Linux爱好者 昨天

数据压倒一切。如果选择了正确的数据结构并把一切组织的井井有条，正确的算法就不言自明。编程的核心是数据结构，而不是算法。——Rob Pike

说明

本文基于这样的认识：数据是易变的，逻辑是稳定的。

本文例举的编程实现多为代码片段，但不影响描述的完整性。

本文例举的编程虽然基于C语言，但其编程思想也适用于其他语言。

此外，本文不涉及语言相关的运行效率讨论。

1 概念提出

所谓表驱动法(Table-Driven Approach)简而言之就是用查表的方法获取数据。此处的“表”通常为数组，但可视为数据库的一种体现。

根据字典中的部首检字表查找读音未知的汉字就是典型的表驱动法，即以每个字的字形为依据，计算出一个索引值，并映射到对应的页数。相比一页一页地顺序翻字典查字，部首检字法效率极高。

具体到编程方面，在数据不多时可用逻辑判断语句(if…else或switch…case)来获取值；但随着数据的增多，逻辑语句会越来越长，此时表驱动法的优势就开始显现。

例如，用36进制(A表示10，B表示11，…)表示更大的数字，逻辑判断语句如下：

```
if(ucNum < 10)
{
    ucNumChar = ConvertToChar(ucNum);
}
else if(ucNum == 10)
{
    ucNumChar = 'A';
}
else if(ucNum == 11)
{
    ucNumChar = 'B';
}
else if(ucNum == 12)
{
    ucNumChar = 'C';
}
//... ...
else if(ucNum == 35)
{
    ucNumChar = 'Z';
}
```

当然也可以用switch…case结构，但实现都很冗长。而用表驱动法(将numChar存入数组)则非常直观和简洁。如：

```
CHAR aNumChars[] = {'0', '1', '2', /*3~9*/'A', 'B', 'C', /*D~Y*/'Z'};
CHAR ucNumChar = aNumChars[ucNum % sizeof(aNumChars)];
```

像这样直接将变量当作下数组下标来读取数值的方法就是直接查表法。

注意，如果熟悉字符串操作，则上述写法可以更简洁：

```
CHAR ucNumChar = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"[ucNum];
```

使用表驱动法时需要关注两个问题：一是如何查表，从表中读取正确的数据；二是表里存放什么，如数值或函数指针。前者参见1.1节“查表方式”内容，后者参见1.2节“实战示例”内容。

1.1 查表方式

常用的查表方式有直接查找、索引查找和分段查找等。

1.1.1 直接查找

即直接通过数组下标获取到数据。如果熟悉哈希表的话，可以很容易看出这种查表方式就是哈希表的直接访问法。

如获取星期名称，逻辑判断语句如下：

```
if(0 == ucDay)
{
    pszDayName = "Sunday";
}
else if(1 == ucDay)
{
    pszDayName = "Monday";
}
//... ...
else if(6 == ucDay)
{
    pszDayName = "Saturday";
}
```

而实现同样的功能，可将这些数据存储到一个表里：

```
CHAR *paNumChars[] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",  "Saturday"
CHAR *pszDayName = paNumChars[ucDay];
```

类似哈希表特性，表驱动法适用于无需有序遍历数据，且数据量大小可提前预测的情况。

对于过于复杂和庞大的判断，可将数据存为文件，需要时加载文件初始化数组，从而在不修改程序的情况下调整里面的数值。

有时，访问之前需要先进行一次键值转换。如表驱动法表示端口忙闲时，需将槽位端口号映射为全局编号。所生成的端口数目大小的数组，其下标对应全局端口编号，元素值表示相应端口的忙闲状态。

1.1.2 索引查找

有时通过一次键值转换，依然无法把数据(如英文单词等)转为键值。此时可将转换的对应关系写到一个索引表里，即索引访问。

如现有100件商品，4位编号，范围从0000到9999。此时只需要申请一个长度为100的数组，且对应2位键值。但将4位的编号转换为2位的键值，可能过于复杂或没有规律，最合适的方法是建立一个保存该转换关系的索引表。采用索引访问既节省内存，又方便维护。比如索引A表示通过名称访问，索引B表示通过编号访问。

1.1.3 分段查找

通过确定数据所处的范围确定分类(下标)。有的数据可分成若干区间，即具有阶梯性，如分数等级。此时可将每个区间的上限(或下限)存到一个表中，将对应的值存到另一表中，通过第一个表确定所处的区段，再由区段下标在第二个表里读取相应数值。注意要留意端点，可用二分法查找，另外可考虑通过索引方法来代替。

如根据分数查绩效等级：

```
#define MAX_GRADE_LEVEL   (INT8U)5
DOUBLE aRangeLimit[MAX_GRADE_LEVEL] = {50.0, 60.0, 70.0, 80.0, 100.0};
CHAR *paGrades[MAX_GRADE_LEVEL] = {"Fail", "Pass", "Credit", "Distinction", "High Distinction"};

static CHAR* EvaluateGrade(DOUBLE dScore)
{
    INT8U ucLevel = 0;
    for(; ucLevel < MAX_GRADE_LEVEL; ucLevel++)
    {
        if(dScore < aRangeLimit[ucLevel])
            return paGrades[ucLevel];
    }
    return paGrades[0];
}
```

上述两张表(数组)也可合并为一张表(结构体数组)，如下所示：

```
typedef struct{
    DOUBLE  aRangeLimit;
    CHAR    *pszGrade;
}T_GRADE_MAP;

T_GRADE_MAP gGradeMap[MAX_GRADE_LEVEL] = {
    {50.0,              "Fail"},
```

```
    {60.0,              "Pass"},
    {70.0,              "Credit"},
    {80.0,              "Distinction"},
    {100.0,             "High Distinction"}
};

static CHAR* EvaluateGrade(DOUBLE dScore)
{
    INT8U ucLevel = 0;
    for(; ucLevel < MAX_GRADE_LEVEL; ucLevel++)
    {
        if(dScore < gGradeMap[ucLevel].aRangeLimit)
            return gGradeMap[ucLevel].pszGrade;
    }
    return gGradeMap[0].pszGrade;
}
```

该表结构已具备的数据库的雏形，并可扩展支持更为复杂的数据。其查表方式通常为索引查找，偶尔也为分段查找；当索引具有规律性(如连续整数)时，退化为直接查找。

使用分段查找法时应注意边界，将每一分段范围的上界值都考虑在内。找出所有不在最高一级范围内的值，然后把剩下的值全部归入最高一级中。有时需要人为地为最高一级范围添加一个上界。

同时应小心不要错误地用“<”来代替“<=”。要保证循环在找出属于最高一级范围内的值后恰当地结束，同时也要保证恰当处理范围边界。

1.2 实战示例

本节多数示例取自实际项目。表形式为一维数组、二维数组和结构体数组；表内容有数据、字符串和函数指针。基于表驱动的思想，表形式和表内容可衍生出丰富的组合。

1.2.1 字符统计

问题：统计用户输入的一串数字中每个数字出现的次数。

普通解法主体代码如下：

```
INT32U aDigitCharNum[10] = {0}; /* 输入字符串中各数字字符出现的次数 */
INT32U dwStrLen = strlen(szDigits);

INT32U dwStrIdx = 0;
```

```
for(; dwStrIdx < dwStrLen; dwStrIdx++)
{
    switch(szDigits[dwStrIdx])
    {
        case '1':
            aDigitCharNum[0]++;
            break;
        case '2':
            aDigitCharNum[1]++;
            break;
        //... ...
        case '9':
            aDigitCharNum[8]++;
            break;
    }
}
```

这种解法的缺点显而易见，既不美观也不灵活。其问题关键在于未将数字字符与数组aDigitCharNum下标直接关联起来。

以下示出更简洁的实现方式：

```
for(; dwStrIdx < dwStrLen; dwStrIdx++)
{
    aDigitCharNum[szDigits[dwStrIdx] - '0']++;
}
```

上述实现考虑到0也为数字字符。该解法也可扩展至统计所有ASCII可见字符。

1.2.2 月天校验

问题：对给定年份和月份的天数进行校验(需区分平年和闰年)。

普通解法主体代码如下：

```
switch(OnuTime.Month)
{
    case 1:
    case 3:
    case 5:
    case 7:
```

```
    case 8:
    case 10:
    case 12:
        if(OnuTime.Day>31 || OnuTime.Day<1)
        {
            CtcOamLog(FUNCTION_Pon,"Don't support this Day: %d(1~31)!!!\n", OnuTime.Day);
            retcode = S_ERROR;
        }
        break;
    case 2:
        if(((OnuTime.Year%4 == 0) && (OnuTime.Year%100 != 0)) || (OnuTime.Year%400 == 0))
        {
            if(OnuTime.Day>29 || OnuTime.Day<1)
            {
                CtcOamLog(FUNCTION_Pon,"Don't support this Day: %d(1~29)!!!\n", OnuTime.Day);
                retcode = S_ERROR;
            }
        }
        else
        {
            if(OnuTime.Day>28 || OnuTime.Day<1)
            {
                CtcOamLog(FUNCTION_Pon,"Don't support this Day: %d(1~28)!!!\n", OnuTime.Day);
                retcode = S_ERROR;
            }
        }
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        if(OnuTime.Day>30 || OnuTime.Day<1)
        {
            CtcOamLog(FUNCTION_Pon,"Don't support this Day: %d(1~30)!!!\n", OnuTime.Day);
            retcode = S_ERROR;
        }
        break;
    default:
        CtcOamLog(FUNCTION_Pon,"Don't support this Month: %d(1~12)!!!\n", OnuTime.Month);
        retcode = S_ERROR;
        break;
}
```

以下示出更简洁的实现方式：

```
#define MONTH_OF_YEAR 12    /* 一年中的月份数 */
/* 闰年：能被4整除且不能被100整除，或能被400整除 */
#define IS_LEAP_YEAR(year) ((((year) % 4 == 0) && ((year) % 100 != 0)) || ((year) % 400 == 0))

/* 平年中的各月天数，下标对应月份 */
INT8U aDayOfCommonMonth[MONTH_OF_YEAR] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

INT8U ucMaxDay = 0;
if((OnuTime.Month == 2) && (IS_LEAP_YEAR(OnuTime.Year)))
    ucMaxDay = aDayOfCommonMonth[1] + 1;
else
    ucMaxDay = aDayOfCommonMonth[OnuTime.Month-1];

if((OnuTime.Day < 1) || (OnuTime.Day > ucMaxDay)
{
    CtcOamLog(FUNCTION_Pon,"Month %d doesn't have this Day: %d(1~%d)!!!\n",
              OnuTime.Month, OnuTime.Day, ucMaxDay);
    retcode = S_ERROR;
}
```

1.2.3 名称构造

问题：根据WAN接口承载的业务类型(Bitmap)构造业务类型名称字符串。

普通解法主体代码如下：

```
void Sub_SetServerType(INT8U *ServerType, INT16U wan_servertype)
{
    if ((wan_servertype & 0x0001) == 0x0001)
    {
        strcat(ServerType, "_INTERNET");
    }
    if ((wan_servertype & 0x0002) == 0x0002)
    {
        strcat(ServerType, "_TR069");
    }
    if ((wan_servertype & 0x0004) == 0x0004)
    {
        strcat(ServerType, "_VOIP");
    }
    if ((wan_servertype & 0x0008) == 0x0008)
    {
        strcat(ServerType, "_OTHER");
```

```
    }
}
```

以下示出C语言中更简洁的实现方式：

```
/* 获取var变量第bit位，编号从右至左 */
#define  GET_BIT(var, bit)   (((var) >> (bit)) & 0x1)
const CHAR* paSvrNames[] = {"_INTERNET", "_TR069", "_VOIP", "_OTHER"};
const INT8U ucSvrNameNum = sizeof(paSvrNames) / sizeof(paSvrNames[0]);

VOID SetServerType(CHAR *pszSvrType, INT16U wSvrType)
{
    INT8U ucIdx = 0;
    for(; ucIdx < ucSvrNameNum; ucIdx++)
    {
        if(1 == GET_BIT(wSvrType, ucIdx))
            strcat(pszSvrType, paSvrNames[ucIdx]);
    }
}
```

新的实现将数据和逻辑分离，维护起来非常方便。只要逻辑(规则)不变，则唯一可能的改动就是数据(paSvrNames)。

1.2.4 值名解析

问题：根据枚举变量取值输出其对应的字符串，如PORT_FE(1)输出“Fe”。

```
//值名映射表结构体定义，用于数值解析器typedef struct{
    INT32U dwElem;    //待解析数值，通常为枚举变量
    CHAR*  pszName;   //指向数值所对应解析名字符串的指针
}T_NAME_PARSER;

/*****************************************************************************
* 函数名称:  NameParser
* 功能说明:  数值解析器，将给定数值转换为对应的具名字符串
* 输入参数:  VOID *pvMap        :值名映射表数组，含T_NAME_PARSER结构体类型元素
                                 VOID指针允许用户在保持成员数目和类型不变的前提下，
                                 定制更有意义的结构体名和/或成员名。
             INT32U dwEntryNum :值名映射表数组条目数
             INT32U dwElem     :待解析数值，通常为枚举变量
             INT8U* pszDefName :缺省具名字符串指针，可为空
* 输出参数:  NA
* 返回值   :  INT8U *: 数值所对应的具名字符串
              当无法解析给定数值时，若pszDefName为空，则返回数值对应的16进制格式
```

```
            字符串；否则返回pszDefName。
*****************************************************************************/
INT8U *NameParser(VOID *pvMap, INT32U dwEntryNum, INT32U dwElem, INT8U* pszDefName)
{
    CHECK_SINGLE_POINTER(pvMap, "NullPoniter");

    INT32U dwEntryIdx = 0;
    for(dwEntryIdx = 0; dwEntryIdx < dwEntryNum; dwEntryIdx++)
    {
        T_NAME_PARSER *ptNameParser = (T_NAME_PARSER *)pvMap;
        if(dwElem == ptNameParser->dwElem)
        {
            return ptNameParser->pszName;
        }
        //ANSI标准禁止对void指针进行算法操作；GNU标准则指定void*算法操作与char*一致。
        //若考虑移植性，可将pvMap类型改为INT8U*，或定义INT8U*局部变量指向pvMap。
        pvMap += sizeof(T_NAME_PARSER);
    }

    if(NULL != pszDefName)
    {
        return pszDefName;
    }
    else
    {
        static INT8U szName[12] = {0}; //Max:"0xFFFFFFFF"
        sprintf(szName, "0x%X", dwElem);
        return szName;
    }
}
```

以下给出NameParser的简单应用示例：

```
//UNI端口类型值名映射表结构体定义
typedef struct{
    INT32U dwPortType;
    INT8U* pszPortName;
}T_PORT_NAME;
//UNI端口类型解析器
T_PORT_NAME gUniNameMap[] = {
    {1,      "Fe"},
    {3,      "Pots"},
    {99,     "Vuni"}
};
const INT32U UNI_NAM_MAP_NUM = (INT32U)(sizeof(gUniNameMap)/sizeof(T_PORT_NAME));
VOID NameParserTest(VOID)
{
    INT8U ucTestIndex = 1;

    printf("[%s]<Test Case %u> Result: %s!\n", __FUNCTION__, ucTestIndex++,
```

```
            strcmp("Unknown", NameParser(gUniNameMap, UNI_NAM_MAP_NUM, 0, "Unknown")) ? "ERROR" : "
    printf("[%s]<Test Case %u> Result: %s!\n", __FUNCTION__, ucTestIndex++,
            strcmp("DefName", NameParser(gUniNameMap, UNI_NAM_MAP_NUM, 0, "DefName")) ? "ERROR" : "
    printf("[%s]<Test Case %u> Result: %s!\n", __FUNCTION__, ucTestIndex++,
            strcmp("Fe", NameParser(gUniNameMap, UNI_NAM_MAP_NUM, 1, "Unknown")) ? "ERROR" : "OK");
    printf("[%s]<Test Case %u> Result: %s!\n", __FUNCTION__, ucTestIndex++,
            strcmp("Pots", NameParser(gUniNameMap, UNI_NAM_MAP_NUM, 3, "Unknown")) ? "ERROR" : "OK"
    printf("[%s]<Test Case %u> Result: %s!\n", __FUNCTION__, ucTestIndex++,
            strcmp("Vuni", NameParser(gUniNameMap, UNI_NAM_MAP_NUM, 99, NULL)) ? "ERROR" : "OK");
    printf("[%s]<Test Case %u> Result: %s!\n", __FUNCTION__, ucTestIndex++,
            strcmp("Unknown", NameParser(gUniNameMap, UNI_NAM_MAP_NUM, 255, "Unknown")) ? "ERROR" :
    printf("[%s]<Test Case %u> Result: %s!\n", __FUNCTION__, ucTestIndex++,
            strcmp("0xABCD", NameParser(gUniNameMap, UNI_NAM_MAP_NUM, 0xABCD, NULL)) ? "ERROR" : "O
    printf("[%s]<Test Case %u> Result: %s!\n", __FUNCTION__, ucTestIndex++,
            strcmp("NullPoniter", NameParser(NULL, UNI_NAM_MAP_NUM, 0xABCD, NULL)) ? "ERROR" : "OK"
}
```

gUniNameMap在实际项目中有十余个条目，若采用逻辑链实现将非常冗长。

1.2.5 取值映射

问题：不同模块间同一参数枚举值取值可能有所差异，需要适配。

此处不再给出普通的switch…case或if…else if…else结构，而直接示出以下表驱动实现：

```
typedef struct{
    PORTSTATE loopMEState;
    PORTSTATE loopMIBState;
}LOOPMAPSTRUCT;

static LOOPMAPSTRUCT s_CesLoop[] = {
    {NO_LOOP,                   e_ds1_looptype_noloop},
    {PAYLOAD_LOOP,              e_ds1_looptype_PayloadLoop},
    {LINE_LOOP,                 e_ds1_looptype_LineLoop},
    {PON_LOOP,                  e_ds1_looptype_OtherLoop},
    {CES_LOOP,                  e_ds1_looptype_InwardLoop}};

PORTSTATE ConvertLoopMEStateToMIBState(PORTSTATE vPortState)
{
    INT32U num = 0, ii;

    num = ARRAY_NUM(s_CesLoop);
    for(ii = 0; ii < num; ii++)
```

```
    {
        if(vPortState == s_CesLoop[ii].loopMEState)
            return s_CesLoop[ii].loopMIBState;
    }
    return e_ds1_looptype_noloop;
}
```

相应地，从loopMIBState映射到loopMEState需要定义一个ConvertLoopMIBStateToMEState函数。更进一步，所有类似的一对一映射关系都必须如上的映射(转换)函数，相当繁琐。

事实上，从抽象层面看，该映射关系非常简单。提取共性后定义带参数宏，如下所示：

```
/*************************************************************
* 功能描述：进行二维数组映射表的一对一映射，用于参数适配
* 参数说明：map         -- 二维数组映射表
            elemSrc     -- 映射源，即待映射的元素值
            elemDest    -- 映射源对应的映射结果
            direction   -- 映射方向字节，表示从数组哪列映射至哪列。
                           高4位对应映射源列，低4位对应映射结果列。
            defaultVal -- 映射失败时置映射结果为缺省值
* 示例：ARRAY_MAPPER(gCesLoopMap, 3, ucLoop, 0x10, NO_LOOP);
            则ucLoop = 2(LINE_LOOP)
*************************************************************/
#define ARRAY_MAPPER(map, elemSrc, elemDest, direction, defaultVal) do{\
    INT8U ucMapIdx = 0, ucMapNum = 0; \
    ucMapNum = sizeof(map)/sizeof(map[0]); \
    for(ucMapIdx = 0; ucMapIdx < ucMapNum; ucMapIdx++) \
    { \
        if((elemSrc) == map[ucMapIdx][((direction)&0xF0)>>4]) \
        { \
            elemDest = map[ucMapIdx][(direction)&0x0F]; \
            break; \
        } \
    } \
    if(ucMapIdx == ucMapNum) \
    { \
        elemDest = (defaultVal); \
    } \
}while(0)
```

参数取值转换时直接调用统一的映射器宏，如下：

```
static INT8U gCesLoopMap[][2] = {
```

```
    {NO_LOOP,                   e_ds1_looptype_noloop},
    {PAYLOAD_LOOP,              e_ds1_looptype_PayloadLoop},
    {LINE_LOOP,                 e_ds1_looptype_LineLoop},
    {PON_LOOP,                  e_ds1_looptype_OtherLoop},
    {CES_LOOP,                  e_ds1_looptype_InwardLoop}};

ARRAY_MAPPER(gCesLoopMap, tPara.dwParaVal[0], dwLoopConf, 0x01, e_ds1_looptype_noloop);
```

另举一例：

```
#define  CES_DEFAULT_JITTERBUF        (INT32U)2000   /* 默认jitterbuf为2000us，而1帧=125us */
#define  CES_JITTERBUF_STEP           (INT32U)125    /* jitterbuf步长为125us，即1帧 */
#define  CES_DEFAULT_QUEUESIZE        (INT32U)5
#define  CES_DEFAULT_MAX_QUEUESIZE    (INT32U)7

#define  ARRAY_NUM(array)             (sizeof(array) / sizeof((array)[0]))  /* 数组元素个数 */
typedef struct{
    INT32U  dwJitterBuffer;
    INT32U  dwFramePerPkt;
    INT32U  dwQueueSize;
}QUEUE_SIZE_MAP;
/* gCesQueueSizeMap也可以(JitterBuffer / FramePerPkt)值为索引，更加紧凑 */
static QUEUE_SIZE_MAP gCesQueueSizeMap[]= {
       {1,1,1},  {1,2,1},  {2,1,2},  {2,2,1},
       {3,1,3},  {3,2,1},  {4,1,3},  {4,2,1},
       {5,1,4},  {5,2,3},  {6,1,4},  {6,2,3},
       {7,1,4},  {7,2,3},  {8,1,4},  {8,2,3},
       {9,1,5},  {9,2,4},  {10,1,5}, {10,2,4},
       {11,1,5}, {11,2,4}, {12,1,5}, {12,2,4},
       {13,1,5}, {13,2,4}, {14,1,5}, {14,2,4},
       {15,1,5}, {15,2,4}, {16,1,5}, {16,2,4},
       {17,1,6}, {17,2,5}, {18,1,6}, {18,2,5},
       {19,1,6}, {19,2,5}, {20,1,6}, {20,2,5},
       {21,1,6}, {21,2,5}, {22,1,6}, {22,2,5},
       {23,1,6}, {23,2,5}, {24,1,6}, {24,2,5},
       {25,1,6}, {25,2,5}, {26,1,6}, {26,2,5},
       {27,1,6}, {27,2,5}, {28,1,6}, {28,2,5},
       {29,1,6}, {29,2,5}, {30,1,6}, {30,2,5},
       {31,1,6}, {31,2,5}, {32,1,6}, {32,2,5}};
/*********************************************************
* 函数名称：CalcQueueSize
* 功能描述：根据JitterBuffer和FramePerPkt计算QueueSize
* 注意事项：配置的最大缓存深度
*           = 2 * JitterBuffer / FramePerPkt
*           = 2 * N Packet = 2 ^ QueueSize
*           JitterBuffer为125us帧速率的倍数，
*           FramePerPkt为每个分组的帧数，
*           QueueSize向上取整，最大为7。
*********************************************************/
```

```
INT32U CalcQueueSize(INT32U dwJitterBuffer, INT32U dwFramePerPkt)
{
    INT8U ucIdx = 0, ucNum = 0;

    //本函数暂时仅考虑E1
    ucNum = ARRAY_NUM(gCesQueueSizeMap);
    for(ucIdx = 0; ucIdx < ucNum; ucIdx++)
    {
       if((dwJitterBuffer == gCesQueueSizeMap[ucIdx].dwJitterBuffer) &&
          (dwFramePerPkt == gCesQueueSizeMap[ucIdx].dwFramePerPkt))
       {
            return gCesQueueSizeMap[ucIdx].dwQueueSize;
       }
    }

    return CES_DEFAULT_MAX_QUEUESIZE;
}
```

1.2.6 版本控制

问题：控制OLT与ONU之间的版本协商。ONU本地设置三比特控制字，其中bit2(MSB)~bit0(LSB)分别对应0x21、0x30和0xAA版本号；且bitX为0表示上报对应版本号，bitX为1表示不上报对应版本号。其他版本号如0x20、0x13和0x1必须上报，即不受控制。

最初的实现采用if…else if…else结构，代码非常冗长，如下：

```
pstSendTlv->ucLength = 0x1f;
if (gOamCtrlCode == 0)
{
    vosMemCpy(pstSendTlv->aucVersionList, ctc_oui, 3);
    pstSendTlv->aucVersionList[3] = 0x30;
    vosMemCpy(&(pstSendTlv->aucVersionList[4]), ctc_oui, 3);
    pstSendTlv->aucVersionList[7] = 0x21;
    vosMemCpy(&(pstSendTlv->aucVersionList[8]), ctc_oui, 3);
    pstSendTlv->aucVersionList[11] = 0x20;
    vosMemCpy(&(pstSendTlv->aucVersionList[12]), ctc_oui, 3);
    pstSendTlv->aucVersionList[15] = 0x13;
    vosMemCpy(&(pstSendTlv->aucVersionList[16]), ctc_oui, 3);
    pstSendTlv->aucVersionList[19] = 0x01;
    vosMemCpy(&(pstSendTlv->aucVersionList[20]), ctc_oui, 3);
    pstSendTlv->aucVersionList[23] = 0xaa;
}
else if (gOamCtrlCode == 1)
{
    vosMemCpy(pstSendTlv->aucVersionList, ctc_oui, 3);
    pstSendTlv->aucVersionList[3] = 0x30;
    vosMemCpy(&(pstSendTlv->aucVersionList[4]), ctc_oui, 3);
    pstSendTlv->aucVersionList[7] = 0x21;
```

```
    vosMemCpy(&(pstSendTlv->aucVersionList[8]), ctc_oui, 3);
    pstSendTlv->aucVersionList[11] = 0x20;
    vosMemCpy(&(pstSendTlv->aucVersionList[12]), ctc_oui, 3);
    pstSendTlv->aucVersionList[15] = 0x13;
    vosMemCpy(&(pstSendTlv->aucVersionList[16]), ctc_oui, 3);
    pstSendTlv->aucVersionList[19] = 0x01;
}
//此处省略gOamCtrlCode == 2~6的处理代码
else if (gOamCtrlCode == 7)
{
    vosMemCpy(&(pstSendTlv->aucVersionList), ctc_oui, 3);
    pstSendTlv->aucVersionList[3] = 0x20;
    vosMemCpy(&(pstSendTlv->aucVersionList[4]), ctc_oui, 3);
    pstSendTlv->aucVersionList[7] = 0x13;
    vosMemCpy(&(pstSendTlv->aucVersionList[8]), ctc_oui, 3);
    pstSendTlv->aucVersionList[11] = 0x01;
}
```

以下示出C语言中更简洁的实现方式(基于二维数组)：

```
/*****************************************************************
* 版本控制字数组定义
* gOamCtrlCode:   Bitmap控制字。Bit-X为0时上报对应版本，Bit-X为1时屏蔽对应版本。
* CTRL_VERS_NUM:  可控版本个数。
* CTRL_CODE_NUM:  控制字个数。与CTRL_VERS_NUM有关。
* gOamVerCtrlMap: 版本控制字数组。行对应控制字，列对应可控版本。
                  元素值为0时不上报对应版本，元素值非0时上报该元素值。
* Note: 该数组旨在实现“数据与控制隔离”。后续若要新增可控版本，只需修改
                  -- CTRL_VERS_NUM
                  -- gOamVerCtrlMap新增行(控制字)
                  -- gOamVerCtrlMap新增列(可控版本)
*****************************************************************/
#define CTRL_VERS_NUM    3
#define CTRL_CODE_NUM    (1<<CTRL_VERS_NUM)
u8_t gOamVerCtrlMap[CTRL_CODE_NUM][CTRL_VERS_NUM] = {
 /* Ver21         Ver30        VerAA */
    {0x21,         0x30,        0xaa},    /*gOamCtrlCode = 0*/
    {0x21,         0x30,          0 },    /*gOamCtrlCode = 1*/
    {0x21,           0,         0xaa},    /*gOamCtrlCode = 2*/
    {0x21,           0,           0 },    /*gOamCtrlCode = 3*/
    {  0,          0x30,        0xaa},    /*gOamCtrlCode = 4*/
    {  0,          0x30,          0 },    /*gOamCtrlCode = 5*/
    {  0,            0,         0xaa},    /*gOamCtrlCode = 6*/
    {  0,            0,           0 }     /*gOamCtrlCode = 7*/
};
#define INFO_TYPE_VERS_LEN    7  /* InfoType + Length + OUI + ExtSupport + Version */

u8_t verIdx = 0;
u8_t index = 0;
for(verIdx = 0; verIdx < CTRL_VERS_NUM; verIdx++)
```

```
for(verIdx = 0; verIdx < CTRL_VERS_NUM; verIdx++)
{
    if(gOamVerCtrlMap[gOamCtrlCode][verIdx] != 0)
    {
        vosMemCpy(&pstSendTlv->aucVersionList[index], ctc_oui, 3);
        index += 3;
        pstSendTlv->aucVersionList[index++] = gOamVerCtrlMap[gOamCtrlCode][verIdx];
    }
}
vosMemCpy(&pstSendTlv->aucVersionList[index], ctc_oui, 3);
index += 3;
pstSendTlv->aucVersionList[index++] = 0x20;
vosMemCpy(&pstSendTlv->aucVersionList[index], ctc_oui, 3);
index += 3;
pstSendTlv->aucVersionList[index++] = 0x13;
vosMemCpy(&pstSendTlv->aucVersionList[index], ctc_oui, 3);
index += 3;
pstSendTlv->aucVersionList[index++] = 0x01;

pstSendTlv->ucLength = INFO_TYPE_VERS_LEN + index;
```

1.2.7 消息处理

问题：终端输入不同的打印命令，调用相应的打印函数，以控制不同级别的打印。

这是一段消息(事件)驱动程序。本模块接收其他模块(如串口驱动)发送的消息，根据消息中的打印级别字符串和开关模式，调用不同函数进行处理。常见的实现方法如下：

```
void logall(void)
{
    g_log_control[0] = 0xFFFFFFFF;
}

void noanylog(void)
{
    g_log_control[0] = 0;
}

void logOam(void)
{
    g_log_control[0] |= (0x01 << FUNCTION_Oam);
}
void nologOam(void)
{
    g_log_control[0] &= ~(0x01 << FUNCTION_Oam);
}
//... ...
void logExec(char *name, INT8U enable)
{
```

```
    CtcOamLog(FUNCTION_Oam,"log %s %d\n",name,enable);
    if (enable == 1) /*log*/
    {
        if (strcasecmp(name,"all") == 0) { /*字符串比较，不区分大小写*/
            logall();
        } else if (strcasecmp(name,"oam") == 0) {
            logOam();
        } else if (strcasecmp(name,"pon") == 0) {
            logPon();
        //... ...
        } else if (strcasecmp(name,"version") == 0) {
            logVersion();
    }
    else if (enable == 0) /*nolog*/
    {
        if (strcasecmp(name,"all") == 0) {
            noanylog();
        } else if (strcasecmp(name,"oam") == 0) {
            nologOam();
        } else if (strcasecmp(name,"pon") == 0) {
            nologPon();
        //... ...
        } else if (strcasecmp(name,"version") == 0) {
            nologVersion();
    }
    else
    {
        printf("bad log para\n");
    }
}
```

以下示出C语言中更简洁的实现方式：

```
typedef struct{
    OAM_LOG_OFF = (INT8U)0,
    OAM_LOG_ON  = (INT8U)1
}E_OAM_LOG_MODE;
typedef FUNC_STATUS (*OamLogHandler)(VOID);
typedef struct{
    CHAR           *pszLogCls;    /* 打印级别 */
    E_OAM_LOG_MODE eLogMode;      /* 打印模式 */
    OamLogHandler  fnLogHandler;  /* 打印函数 */
}T_OAM_LOG_MAP;

T_OAM_LOG_MAP gOamLogMap[] = {
    {"all",         OAM_LOG_OFF,       noanylog},
```

```
    {"oam",         OAM_LOG_OFF,       nologOam},
    //... ...
    {"version",     OAM_LOG_OFF,       nologVersion},

    {"all",         OAM_LOG_ON,        logall},
    {"oam",         OAM_LOG_ON,        logOam},
    //... ...
    {"version",     OAM_LOG_ON,        logVersion}
};
INT32U gOamLogMapNum = sizeof(gOamLogMap) / sizeof(T_OAM_LOG_MAP);

VOID logExec(CHAR *pszName, INT8U ucSwitch)
{
    INT8U ucIdx = 0;
    for(; ucIdx < gOamLogMapNum; ucIdx++)
    {
        if((ucSwitch == gOamLogMap[ucIdx].eLogMode) &&
           (!strcasecmp(pszName, gOamLogMap[ucIdx].pszLogCls));
        {
            gOamLogMap[ucIdx].fnLogHandler();
            return;
        }
    }
    if(ucIdx == gOamLogMapNum)
    {
        printf("Unknown LogClass(%s) or LogMode(%d)!\n", pszName, ucSwitch);
        return;
    }
}
```

这种表驱动消息处理实现的优点如下：

1.增强可读性，消息如何处理从表中一目了然。

2.增强可扩展性。更容易修改，要增加新的消息，只要修改数据即可，不需要修改流程。

3.降低复杂度。通过把程序逻辑的复杂度转移到人类更容易处理的数据中来，从而达到控制复杂度的目标。

4.主干清晰，代码重用。

若各索引为顺序枚举值，则建立多维数组(每维对应一个索引)，根据下标直接定位到处理函数，效率会更高。

注意，考虑到本节实例中logOam/logPon或nologOam/nologPon等函数本质上是基于打印级别的比特操作，因此可进一步简化。以下例举其相似实现：

```
/* 日志控制类型定义 */
typedef enum
{
    LOG_NORM = 0,        /* 未分类日志，可用于通用日志 */
    LOG_FRM,             /* Frame，OMCI帧日志 */
    LOG_PON,             /* Pon，光链路相关日志 */
    LOG_ETH,             /* Ethernet，Layer2以太网日志 */
    LOG_NET,             /* Internet，Layer3网络日志 */
    LOG_MULT,            /* Multicast，组播日志 */
    LOG_QOS,             /* QOS，流量日志 */
    LOG_CES,             /* Ces，TDM电路仿真日志 */
    LOG_VOIP,            /* Voip，语音日志 */
    LOG_ALM,             /* Alarm，告警日志 */
    LOG_PERF,            /* Performance，性能统计日志 */
    LOG_VER,             /* Version，软件升级日志 */
    LOG_XDSL,            /* xDsl日志 */
    LOG_DB,              /* 数据库操作日志 */
    //新日志类型在此处扩展，共支持32种日志类型
    LOG_ALL = UINT_MAX   /* 所有日志类型 */
}E_LOG_TYPE;

/*****************************************************************************
 * 变量名称：gOmciLogCtrl
 * 作用描述：OMCI日志控制字，BitMap格式(比特编号从LSB至MSB依次为Bit0->BitN)。
 *           Bit0~N分别对应E_LOG_TYPE各枚举值(除LOG_ALL外)。
 *           BitX为0时关闭日志类型对应的日志功能，BitX为1时则予以打开。
 * 变量范围：该变量为四字节整型静态全局变量，即支持32种日志类型。
 * 访问说明：通过GetOmciLogCtrl/SetOmciLogCtrl/OmciLogCtrl函数访问/设置控制字。
 *****************************************************************************/
static INT32U gOmciLogCtrl = 0;

//日志类型字符串数组，下标为各字符串所对应的日志类型枚举值。
static const INT8U* paLogTypeName[] = {
    "Norm",        "Frame",   "Pon",  "Ethernet",  "Internet",
    "Multicast",   "Qos",     "Ces",  "Voip",      "Alarm",
    "Performance", "Version", "Xdsl",  "Db"
};
static const INT8U  ucLogTypeNameNum = sizeof(paLogTypeName) / sizeof(paLogTypeName[0]);

static VOID SetGlobalLogCtrl(E_LOG_TYPE eLogType, INT8U ucLogSwitch)
{
    if(LOG_ON == ucLogSwitch)
        gOmciLogCtrl = LOG_ALL;
    else
        gOmciLogCtrl = 0;
}
static VOID SetSpecificLogCtrl(E_LOG_TYPE eLogType, INT8U ucLogSwitch)
{
    if(LOG_ON == ucLogSwitch)
        SET_BIT(gOmciLogCtrl, eLogType);
    else
```

```
        CLR_BIT(gOmciLogCtrl, eLogType);
}

VOID OmciLogCtrl(CHAR *pszLogType, INT8U ucLogSwitch)
{
    if(0 == strncasecmp(pszLogType, "All", LOG_TYPE_CMP_LEN))
    {
        SetGlobalLogCtrl(LOG_ALL, ucLogSwitch);
        return;
    }

    INT8U ucIdx = 0;
    for(ucIdx = 0; ucIdx < ucLogTypeNameNum; ucIdx++)
    {
        if(0 == strncasecmp(pszLogType, paLogTypeName[ucIdx], LOG_TYPE_CMP_LEN))
        {
            SetSpecificLogCtrl(ucIdx, ucLogSwitch);
            printf("LogType: %s, LogSwitch: %s\n", paLogTypeName[ucIdx],
                   (1==ucLogSwitch)?"On":"Off");
            return;
        }
    }

    OmciLogHelp();
}
```

2 编程思想

表驱动法属于数据驱动编程的一种，其核心思想在《Unix编程艺术》和《代码大全2》中均有阐述。两者均认为人类阅读复杂数据结构远比复杂的控制流程容易，即相对于程序逻辑，人类更擅长于处理数据。

本节将由Unix设计原则中的“分离原则”和“表示原则”展开。

分离原则：策略同机制分离，接口同引擎分离

机制即提供的功能；策略即如何使用功能。

策略的变化要远远快于机制的变化。将两者分离，可以使机制相对保持稳定，而同时支持策略的变化。

代码大全中提到“隔离变化”的概念，以及设计模式中提到的将易变化的部分和不易变化的部分分离也是这个思路。

表示原则：把知识叠入数据以求逻辑质朴而健壮

即使最简单的程序逻辑让人类来验证也很困难，但就算是很复杂的数据，对人类来说，还是相对容易推导和建模的。数据比编程逻辑更容易驾驭。在复杂数据和复杂代码中选择，宁可选择前者。更进一步，在设计中，应该主动将代码的复杂度转移到数据中去(参考“版本控制”)。

在“消息处理”示例中，每个消息处理的逻辑不变，但消息可能是变化的。将容易变化的消息和不容易变化的查找逻辑分离，即“隔离变化”。此外，该例也体现消息内部的处理逻辑(机制)和不同的消息处理(策略)分离。

数据驱动编程可以应用于：

1.函数级设计，如本文示例。2.程序级设计，如用表驱动法实现状态机。3.系统级设计，如DSL。

注意，数据驱动编程不是全新的编程模型，只是一种设计思路，在Unix/Linux开源社区应用很多。数据驱动编程中，数据不但表示某个对象的状态，实际上还定义程序的流程，这点不同于面向对象设计中的数据“封装”。

3 附录

3.1 网友观点

(以下观点摘自博客园网友“七心葵”的回帖，非常具有启发性。)

Booch的《面向对象分析与设计》一书中，提到所有的程序设计语言大概有3个源流：结构化编程；面向对象编程；数据驱动编程。

我认为数据驱动编程的本质是“参数化抽象”的思想，不同于OO的“规范化抽象”的思想。

数据驱动编程在网络游戏开发过程中很常用，但是少有人专门提到这个词。

数据驱动编程有很多名字：元编程，解释器/虚拟机，LOP/微语言/DSL等。包括声明式编程、标记语言、甚至所见即所得的拖放控件，都算是数据驱动编程的一种吧。

数据驱动编程可以帮助处理复杂性，和结构化编程、OO 均可相容。(正交的角度)

将变和不变的部分分离，策略和机制分离，由此联想到的还有：(数据和代码的分离，微语言和解释器的分离，被生成代码和代码生成器的分离)；更近一步：(微内核插件式体系结构)

元编程应该说是更加泛化的数据驱动编程，元编程不是新加入一个间接层，而是退居一步，使得当前的层变成一个间接层。元编程分为静态元编程(编译时)和动态元编程(运行时)，静态元编

程本质上是一种 代码生成技术或者编译器技术；动态元编程一般通过解释器(或虚拟机)加以实现。

数据驱动编程当然也不应该说是“反抽象的”，但的确与“OO抽象”的思维方式是迥然不同，泾渭分明的，如TAOUP一书中所述：“在Unix的模块化传统和围绕OO语言发展起来的使用模式之间，存在着紧张的对立关系”应该说数据驱动编程的思路与结构化编程和OO是正交的，更类似一种“跳出三界外，不在五行中”的做法。

编程和人的关系

人类心智的限制，一切的背后都有人的因素作为依据：

a 人同时关注的信息数量：7+-2 (所以要分模块)

b 人接收一组新信息的平均时间5s (所以要简单，系统总的模块数不要太多)

c 人思维的直观性(人的视觉能力和模糊思维能力)，这意味这两点：

A “直”——更善于思考自己能直接接触把玩的东西；(所以要“浅平透”、使用具象的设计，要尽量代码中只有顺直的流程)，

B “观”——更善于观图而不是推算逻辑；(所以要表驱动法，数据驱动编程，要UML，要可视化编程——当然MDA是太理想化了)

d 人不能持续集中注意力(人在一定的代码行数中产生的bug数量的比例是一定的，所以语言有具有表现力，要体现表达的经济性)

所以要机制与策略分离，要数据和代码分离(数据驱动编程)，要微语言，要DSL，要LOP……

e 人是有创造欲，有现实利益心的(只要偶可能总是不够遵从规范，或想创造规范谋利——只要成本能承受，在硬件领域就不行)

另外，开一个有意思的玩笑，Unix编程艺术艺术的英文缩写为TAOUP，我觉得可以理解为UP之TAO——向上抛出之道——将复杂的易变的逻辑作为数据或更高层代码抛给上层！

3.2 函数指针

“消息处理”一节示例中的函数指针有点插件结构的味道。可对这些插件进行方便替换，新增，删除，从而改变程序的行为。而这种改变，对事件处理函数的查找又是隔离的(隔离变化)。

函数指针非常有用，但使用时需注意其缺陷：无法检查参数(parameter)和返回值(return value)的类型。因为函数已经退化成指针，而指针不携带这些类型信息。缺少类型检查，当参数或返回值不一致时，可能会造成严重的错误。

例如，定义三个函数，分别具有两个参数：

```
int max(int x, int y)  {  return x>y?x:y;  }
int min(int x, int y)  {  return x<y?x:y;  }
int add(int x, int y)  {  return x+y;  }
```

而处理函数却定义为：

```
int process(int x, int y, int (*f)())  {  return (*f)(x, y);  }
```

其中，第三个参数是一个没有参数且返回int型变量的函数指针。但后面却用process(a,b,max)的方式进行调用，max带有

因此在C语言中使用函数指针时，一定要小心"类型陷阱"。

> 本文源自网络，侵删

- EOF -

推荐阅读 — 点击标题可跳转

1、20 年嵌入式经验：如何从零开始开发一款嵌入式产品

2、5.3 万 Star！世界上最快的静态网站构建框架！

3、VSCode 摸鱼神器，确定不试一下？

看完本文有收获？请分享给更多人

推荐关注「Linux 爱好者」，提升Linux技能

Linux爱好者

点击获取《每天一个Linux命令》系列和精选Linux技术资源。「Linux爱好者」日常分...

73篇原创内容

公众号

点赞和在看就是最大的支持♡