小小调度器 V2.0 简易版分析

本文作者：Gthgth

注意：小小调度器 V2.0 作者：　兔子、smset　　QQ 群：371719283

在作者和“兔子”大虾的努力下，小小调度器迎来一个激动人心的新版本。（2.0 正式版，为了大家方便学习才有 V2.0 简易版）

在作者和兔子的帮助下，开始学习 V2.0 简易版。

V1.1 版本和 V2.0 简易版本不冲突，是两个相对独立的版本，各有各的优点，V1.1 突出强调小，省资源。并不是 v2.0 的存在就取代了 V1.1；和 V1.1 版本相比，v2.0 简易版，在 V1.1 的基础上，支持任务重入；当然了 V2.0 也是一如既往的小。

在百度中查了一下：可重入代码指可被多个函数或程序调用的一段代码(通常是一个函数),而且它保证在被任何一个函数调用时都以同样的方式运行。

在小小调度器 V2.0 中：

子任务可以被多个主任务调用，主任务可以给子任务传递参数。子任务也可以访问主任务的数据。 每个任务之间可以相互访问数据。

具体反映在：

1.把每个任务函数的私有变量和行号、延时时间等都独立出去，保存在自己的结构体变量里面了；

2.在运行任务函数时，有关数据不能直接传给结构体，而是地址进去，进去后转换回结构体。

一.主函数分析

```
void main() {
  while(1){
    delay_ms(1);//延时 1 毫秒
    runtasks();
  }
}
```

分析：很简单，延时 1ms 执行 runtasks()函数；这样就相当于每隔 1ms 扫描一次 runtasks()函数。没有用到定时器中断，这个 1ms 时基可以根据要求修改;如果用定时器，时基写的很小就会频繁的打断 CPU。用延时感觉时基选择小一点这样更节省 CPU 资源,如果延时太长，就会占用太长 CPU。（问：作者为什么用延时作为时基没用定时器？答：那种都行，看情况；在示例中用延时作为时基是考虑到调度器中统一没有涉及到中断。）

（smset 补充：一是由于以 arduino 为例，arduino 默认代码没有提供中断，因此没有采用中断时基。
另一个原因是 V2.0 简易版默认使用 short 类型的任务 Timer 变量，如果使用中断进行 UpdateTimer 更新，是存在隐患的，所以
如果在中断里进行 UpdateTimer 更新，则必须使用 unsigned char 类型的任务 Timer 变量）。

1，展开 runtasks();函数

```
void runtasks(){
  //指定 led1 任务驱动的 IO 管脚
  led1.pin=13;

  //更新顶级任务的时间
  UpdateTimer(led1);
  UpdateTimer(breath1);
  UpdateTimer(serial1);

  //执行顶级任务
```

```
    RunTask(LedTask, led1);
    RunTask(BreathTask, breath1);
    RunTask(SerialTask, serial1);
}
```

LED 的 I/O 管脚初始化其实可以写在专门的初始化函数里面，这里是为了更好的说明，写在了 runtasks()函数里。编程很灵活。

一般来说有几个任务，就有几个对应更新顶级任务的时间函数和对应的执行顶级任务。

2 把 UpdateTimer(led1); 函数展开。

宏 #define UpdateTimer(TaskVar) do { if((TaskVar.task.timer!=0)&&(TaskVar.task.timer!=END)) TaskVar.task.timer--; } while(0)

带入展开： { if((led1.task.timer!=0)&&(led1.task.timer!=END)) led1.task.timer--; }

延时时间 unsigned short timer; 变量值不等于 0，也不等于 END（65535），它的值就减一。等于 0 就去执行对应的函数；等于 65535 就挂起。这个和 1.1 版本是一样的。

led1 是个结构体变量。在 led1 结构体里包含了一个 task 个结构体变量，要引用里面的元素，用.分隔开写到里面的最小元素。

task 结构体里面有两个变量:unsigned short timer; 和 unsigned char lc; (有关结构体变量展开看后面的第 5 部分 ：有关结构体及其宏的展开。)

3 把 RunTask(LedTask, led1);函数展开

宏 #define RunTask(TaskName,TaskVar) do { if(TaskVar.task.timer==0) TaskVar.task.timer=TaskName(&(TaskVar)); } while(0)

展开带入： { if(led1.task.timer==0) led1.task.timer=LedTask(&(led1)); }

假如延时时间到了 timer==0，就执行后面的函数，执行 LedTask(&(led1))函数，执行完把结果赋值给 led1.task.timer 这个变量。这个和 1.0 版本是一样的。就是延时时间到，就去执行任务函数，然后把新的延时时间赋值给自己的 timer，开始下一轮的循环。

因为小小调度器是协作式的，假如某个任务的时间延时到了，并不意味着要马上执行这个任务函数，要等上一个任务释放掉 CPU 后才执行本任务函数；这样就意味着，我们在编制任务函数的时候对任务函数的执行，时间要求不是那么的严格，在一定范围内执行就可以了；同时也意味着 CPU 只有把某个任务函数执行完，把本任务该做的事做完后，然后再做其他的事情；因为 cpu 运行速度是比较快的，一般情况下占用 CPU 资源比较多的是等待条件满足和延时（来个数学运算或者什么的占用 cpu 时间较长怎么破？查表？？，这和 cpu 有关，和调度器没关？）；对于等待条件满足的可以用宏#define WaitUntil(A) ，对于延时的可以用宏#define WaitX(ticks)，这样可以在本任务等待或者延时的时候，做其他事情，提高效率。

我们看一下这个函数：执行 LedTask(&(led1))的结果是一个值，这个函数的原型是 LedTask(C_LedTask *cp) 。

也就是取 led1 结构体变量的首地址(&(led1))传递到函数的原型中定义的结构体变量指针(C_LedTask *cp)。把它们对应起来，就是定义一个结构体指针，并

指向 led1 的首地址，这样两者对应起来（结构体变量 led1 和结构体指针 C_LedTask *cp 类型都是一样的；有关结构体变量展开看后面的第 5 部分 ：有关结构体及其宏的展开。）。这里用结构体指针主要的目的就是把彼此剥离，为实现重入做好准备。实现任务函数多次调用，彼此没有影响。
为了书写方便，作者做了一个宏#define TaskFun(TaskName)　　TimeDef TaskName(C_##TaskName *cp) {switch(me.task.lc){default:
因为这个函数有返回值，所以函数前面加了类型限制符 unsigned short（宏为：#define TimeDef　　unsigned short）。
为了书写或者阅读方便作者就做了一个语法糖（就是一个宏 #define me　(*cp)，为了防止出错，指针一定要加括号，涉及到优先级的问题）。
其实所用的宏定义都可以认为是语法糖，用糖把语法包裹着，就是为了方便书写、理解等等。
4，把 TaskFun(LedTask);函数展开
宏#define TaskFun(TaskName)　　TimeDef TaskName(C_##TaskName *cp) {switch(me.task.lc){default:
展开，替换后：

```
//TaskFun(LedTask){
    unsigned short    LedTask(C_LedTask *cp) {switch(me.task.lc){default:{     //编译器初始化的时候给 lc 赋值为 0。？
    me.timelen=20;//LED PWM 总周期为 20 毫秒。//一般在没有进入循环前，可以对一些变量赋值。V1.1 版本用到私有变量一般是在这里定义的，为局部
    pinMode(me.pin, OUTPUT);//设置管脚输出   //   静态变量；2.0 版本用到的变量在前面统一定义，变量的应用是通过指针进行的。当然了平
                                                //   时怎么用就怎么写。
    while(1)
    {
        digitalWrite(me.pin,HIGH);//点亮 LED
        //WaitX(me.timeon);
       // #define WaitX(ticks)          do { me.task.lc=LINE; return (ticks); case LINE:;} while(0)
        { me.task.lc=LINE; return (me.timeon); case LINE:;}

        digitalWrite(me.pin,LOW);//关闭 LED
        WaitX(me.timelen-me.timeon);
    }
}EndFun
```

展开后可以看到，里面用到的变量都是 通过结构体指针，指向我们当初在“任务类及任务变量”那里定义的结构体变量。这样有关任务函数的操作其实所有的数据都是存在任务自己所定义的变量里面；这样函数重入就不出现问题，多次调用任务函数彼此不影响；当然了运行任务函数前要对先对任务用到的变量定义，全部都是全局变量。这个也是 V2.0 简易版和 v1.1 板的一个区别。

返回 me.timeon 是个 unsigned char 类型的。返回去的时候类型被转换为 unsigned short 型。其余和 1.0 版本一样，在记录行号的时候没有用到静态局部变量，用的是每个任务变量里 task 结构体里面 unsigned char lc;。lc 默认为 uchar 也就是说 TaskFun(TaskName) 任务函数里面 WaitX(ticks)的个数（不包扩它调用的子任务）不能超过 256（0-255）个，这个和 1.1 版本是一样的。 每个任务函数里面所写语句的行数是没有限制的，每个任务函数里面只能有 256 个 WaitX(ticks)，如果发现编译错误，也是在前面增加空行。在 V1.1 版本的时候，有位网友在编制任务函数的时候，有一个里面用的 WaitX(ticks)个数比较多，编写代码的行数也比较多，他发现编译错误，就在 WaitX(ticks)前面加空行，可是又和其他的 WaitX(ticks)冲突，到后面每个 WaitX(ticks)前面都有数量不等的空行；为了避免这种事情出现，一个是修改 lc 变量的类型，这个在 2.0 版本是非常方便的，只要修改宏就行（#define LineDef unsigned char）。在 V1.1 也可以修改，只不过要修改两三处地方。另外一个就是把函数优化或者拆分等等，让任务函数里面不要出现这么多的 WaitX(ticks)。

执行这个函数，返回一个延时的数值，下次执行的时候，通过 SWITCH 语句跳转到上次执行的位置，继续执行相关语句，并返回一个延时数值。这个也是 PT 的精华所在。如果不明白请参考 1.0 版本的分解。

5.有关结构体及其宏的展开。

(1).原型：

```
#define Class(type)              typedef struct C_##type C_##type; struct C_##type
Class(task)
{
   TimeDef timer;
   LineDef lc;
};
```

(2).把宏替换掉

```
typedef struct C_task C_task; struct C_task{
   TimeDef timer;
   LineDef lc;
};
```

把宏替换掉后对于 typedef struct C_task C_task; struct C_task{　　这句的理解分两部分
a.红色部分，用 C_task 代替 struct C_task。用 C_task 可以定义结构体变量。
b.蓝色部分 因为 struct C_task 没有定义，它的定义在下面，告诉上面不是没定义嘛，在这定义了 。
c.一般来说类型定义 typedef struct C_task C_task;，应该放在它所重定义的类型的后面，就是应该在结构体定义后面，像 u8,u16 那样。
d.先做 typedef 类型定义，也就是说，这属于事先声明，之后才有具体定义，跟函数声明一样 。
(3).等价于

```
typedef struct C_task C_task;
 struct C_task{
   TimeDef timer;
   LineDef lc;
};
```

在这里要注意，宏展开后可以看到，可以用 C_task 定义结构体,这个结构体变量里面只包含 unsigned short timer; 和 unsigned char lc;。
(4).任务类及任务变量展开：

```
//Class(LedTask)
typedef struct C_LedTask C_LedTask; struct C_LedTask
{
   C_task task;//每个任务类都必须有 task 变量，里面只包含 timer 和 lc 变量
   unsigned char pin;//LED 对应的管脚
   unsigned char timeon;//LED 点亮的时长
   unsigned char timelen;   //LED 循环点亮的周期
}led1;
```

定义了一个名为 led1 的结构体变量。
在这里需要注意一点：用 C_LedTask 可以定义结构体变量。用如果是 C_LedTask led2; 这是定义了一个名为 led2 的结构体，里面的元素和 led1 里面的一样；
要区分用 C_LedTask 和用 C_task 定义结构体的区别。

在这个任务类里面定义了每个任务函数所用到的私有变量，及每个任务函数用到的记录执行地址的 lc 变量和记录需要延时的变量 timer；是个完整的独立

的个体。 用到子任务时，在父任务结构体中用 C_*** task 定义一个子任务结构体变量 ，从某种意义上讲任务重入也是需要代价的。

有时候感觉绕来绕去，其实就是这个任务类的问题，
1.因为任务类及任务变量定义中用 Class 定义任务所用到的私有变量和一个独立的 C_task task，里面放着这个任务函数的行号和延时时间变量。用宏 C_***Task 可以定义和本任务相关所有变量的结构体。
2.如果任务类里面包含了其他的子任务。一般包含一个或者几个用 C_*** task 定义的 结构体变量；（其实就是相当于把子任务中用到的所有变量在这个父任务中又重新定义了一下）。父任务调用这个子任务，会把数据放到这个子任务的结构体里；同理其他父任务调用同一个子任务也会把数据放到自己的子任务结构体里。
3.父任务函数调用子任务函数时，用到的数据是通过指针传递的，把这些变量传递给子任务函数。当然了父任务函数变量的传递也是通过指针的。这样结合每个任务定义的结构体变量，就能解决任务重入的问题了。**子任务可以被多个主任务调用，主任务可以给子任务传递参数。主任务彼此独立互不影响。**

V1.1 版本，记录行号的变量是局部静态变量，涉及到跨任务的变量都是静态局部变量或者静态全局变量；延时变量是个全局变量。
V2.0 版本，每个任务函数用到的变量都是自己的私有全局变量，在调用的时候通过指针传递。

二.呼吸灯

在上面 LED 控制的基础上设计一个呼吸灯，指示灯从暗到亮变化，分 20 个阶段；再从亮到暗变化，也分 20 个阶段，每个阶段保持 100ms
分析：
作为一个独立的顶级任务，设置的时候，就需要有自己的任务变量和任务函数。
把呼吸灯用到的变量统一放在一个结构体中，起名：breath1；呼吸灯对应的任务函数定义为 BreathTask。编写任务函数的时候，注意把他们定义的结构体名和函数名对应起来，这样就不容易弄混了。
1.呼吸灯任务类及任务变量

```
Class(BreathTask)//LED 呼吸灯控制任务
{
  C_task task;//每个任务类都必须有 task 变量，里面存放着延时变量和行号。
  unsigned char i; //呼吸灯变量，
```

```
}breath1;   //呼吸灯的结构体变量名
2.呼吸灯任务函数（呼吸灯的具体动作）
TaskFun(BreathTask){//实现呼吸灯效果
  while(1)
  {
    //从暗到亮变化
   for (me.i=0;me.i<20;me.i++){   //这个变量 i 就是我们在结构体 breath1 中定义的 unsigned char i; //呼吸灯变量。
       WaitX(100);                 //和 1.0 版本原理一样，释放 CPU，过 100ms 再往下执行。
       led1.timeon=me.i;          //把呼吸灯的变量值，赋值给了 LED 任务函数里的变量了，因为定义的结构体都是全局变量的，可以相互调用，赋值。
   }
   //再从亮到暗变化
   for (me.i=20;me.i>0;me.i--){
       WaitX(100);
       led1.timeon=me.i;          //用到的本任务函数的变量是通过指针;在这里引用其他任务的变量，是直接引用的。执行到 LED 任务函数的时候值变了。
   }
  }
}EndFun
```

3. 任务函数里的变量，都是通过指针传递的，和各自定义的结构体对应起来。由于 2.0 版本需要支持重入，任务函数值不能直接传给结构体，而是地址进去，进去后转换回结构体。

4.在这个呼吸灯的任务函数中 用到了其他任务函数中的变量：led1.timeon=me.i;，通过赋值，下次执行 LED 函数的时候就会发生变化。也就是说这个调度器支持任务之间的数据互访。

5.如果这个呼吸灯任务函数里面不用数据互访赋值，而用子任务调用，怎么写？假如没有这个呼吸灯的任务函数，执行 led 任务函数，led 灯的状态是什么？

三．子任务分析

涉及到的宏#define CallSub(SubTaskName,SubTaskVar) do { WaitX(0);SubTaskVar.task.timer=SubTaskName(&(SubTaskVar)); \
if(SubTaskVar.task.timer!=END) return SubTaskVar.task.timer;} while(0)

看串口任务类及任务变量，

```
Class(SerialTask)
{
  C_task task;   //每个任务类都必须有 task 变量
  C_WaitsecTask waitsec1;//串口任务拥有一个秒延时子任务
  String comdata; //串口任务自己用的变量
}serial1;
```

定义了一个结构体变量 serial1，里面除了自己用的变量外，增加了一个 C_WaitsecTask waitsec1;（定义了一个结构体，里面包含了 WaitsecTask 任务函数所用到的全部变量） 接下来我们看一下串口的任务函数。

```
TaskFun(SerialTask){ //串口任务，定时输出 hello
    Serial.begin(9600);
    Serial.println("start");
    while(1){
        me.waitsec1.seconds=1;//总共延迟 1+2 =3 秒
        CallSub(WaitsecTask,me.waitsec1);
        Serial.println("hello");
    }
}EndFun
```

分解开来看一看

```
TaskFun(SerialTask){ //串口任务，定时输出 hello
```

根据上面分析的经验，执行完任务函数的有关指令，返回一个延时函数给 timer。

我们看一下有关语句：

Serial.begin(9600); Serial.println("start");不用关心，串口的波特率和起始位什么的，（猜的）。
程序执行到 me.waitsec1.seconds=1;很简单，给自己里面子任务中的变量赋了一个值，看清楚是要求子任务延时 1 个单位。
接着继续执行到 CallSub(WaitsecTask,me.waitsec1);我们看一下它的宏

```
#define CallSub(SubTaskName,SubTaskVar)        do { WaitX(0);SubTaskVar.task.timer=SubTaskName(&(SubTaskVar));          \
                                                  if(SubTaskVar.task.timer!=END) return SubTaskVar.task.timer;} while(0)
```

把有关参数带进去。

```
  { WaitX(0);me.waitsec1.task.timer=WaitsecTask(&(me.waitsec1));   if(me.waitsec1.task.timer!=END) return me.waitsec1.task.timer;}
```

展开分析：
执行 WaitX(0);在这里设置一个“断点”，让任务下次从这里执行。记录当前 LC 位置，这样如果子任务有 WAIT(X),出来以后下次能顺利进去。（分析一下如果没有 WaitX(0);会发生什么问题？）
执行自己的子任务 WaitsecTask(&(me.waitsec1)把结果赋值给自己定义的子任务变量里的 timer 变量，
在程序中找找到 WaitsecTask()，函数的原型：

```
#define TaskFun(TaskName)         TimeDef TaskName(C_##TaskName *cp) {switch(me.task.lc){default:

TaskFun(WaitsecTask){//实现指定的秒数延迟（me.waitsec1.seconds=1;在本例中赋值为 1S），之后再加上 2 秒延迟
      for (me.i=0;me.i<me.seconds;me.i++){
          WaitX(1000);
       }
      CallSub(Wait2Task,me.wait2);//这里通过调用 2 秒固定延迟子任务，实现额外的 2 秒延迟。
}EndFun
```

执行完自己指定的延时后，继续执行自己子任务里面子任务调用的它的子任务，CallSub(Wait2Task,me.wait2)，再实现 2S 的延时，展开略。

通过上面的分析，我们很清楚的看到用 Class(task)定义结构体用起来是很方便的，除了考虑自己父任务函数里必须的变量外，对于子函数的调用只要定义一个宏，（其实是把每一层的变量都放在了自己定义的宏里面了），用 CallSub(SubTaskName,SubTaskVar)函数调用就可以了。只要你的内存大你可以

无限的调用，无论子程序怎么调用，彼此互不影响。

定义了任务类（Class(task)），在函数变量应用和子程序变量定义的时候很灵活，减少我们的书写量，每个任务函数用到的数据，都保持在自己独立定义的变量中；函数调用用指针；这样，函数就可以实现重入。任务函数可以相互调用；只要你的内存足够大，就可以无限调用。

以上展开后都在强调为任务重入做准备，其实如果不用到任务重入功能，把 time 变量改为 uchar 感觉 V2.0 简易版和 V1.1 所用的资源相差不多，V2.0 用到的变量全部是全局变量，V1.1 用到的变量涉及到任务之间的切换都是局部静态变量。其实 v2.0 简易版这种写法感觉比 V1.1 的更加清晰。

四．总结

通过上面的分解，我们再回头看一下作者 smset 对 V2.0 的评价

主要改进：

1）彻底解决了任务重入问题

2）很好的解决了任务之间的通信问题

3）引入面向任务对象的概念

4）任务具有自己的变量，提高了程序封装程度