有限状态机-函数指针实现

> 本文原创作者 fukai

一、状态机的介绍

1. 定义

有限状态机，（英语：Finite-state machine, FSM），又称有限状态自动机，简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。

2. 状态机的四个要素

状态机包含了以下4个要素：现态、条件、动作、次态

- 现态：程序所处的当前状态；
- 条件：符合某个条件时，发生状态跳转；
- 动作：程序在每个状态所应当执行的相应的具体算法；
- 次态：符合特定条件后发生跳转，跳转的目标状态即为次态，跳转后次态变为新的现态；

举个具体点的例子，智能车的程序设计。小车在实际运行中会根据实时采集回来的图像，分为许多不同的状态，比如出界，直道，弯道，十字等；小车在特定元素的赛道上，现态即为当前元素应有的状态，动作即为当前状态所应当执行的特定算法（如直道加速、弯道减速等）；当小车从直道经过十字元素时，十字判断条件成立，现态将转变为次态十字状态；转换后，十字状态成为新的现态，动作变为十字状态所应当执行的特定算法。

3. 状态机的实现

- if-else实现
- switch-case实现（目前普遍的做法）
- 函数指针实现（效率最高）

二、状态机的实现

目前普遍采用switch-case的方式来实现有限状态机，因此本文首先使用switch来实现状态机，并由此引申出使用函数指针来实现状态机。

1. switch-case实现

- C语言编程中通常会以枚举来表示各个状态：

```
typedef enum {
    STATE1 = 0,
    STATE2,
    STATE3,
    ...
    STATEn,
} State;
```

- 当然，会事先编写好相对应的条件跳转函数以及状态动作函数：

```
// 条件跳转函数
// 函数返回值为State类型的变量
State State1Jump(void);
State State2Jump(void);
State State3Jump(void);
...
State StateNJump(void);
// 状态动作函数
void State1Work(void);
void State2Work(void);
void State3Work(void);
...
void StateNWork(void);
```

- 用switch来实现不同状态间的转换以及相应动作的实现

```
// 条件跳转函数
State StateJump(State state) {
    switch (state) {
        case STATE1:
            state = State1Jump();
            break;
        case STATE2:
            state = State2Jump();
            break;
        case STATE3:
            state = State3Jump();
            break;
        ...
        case STATEn:
            state = StateNJump();
            break;
    }
    return state;
}
// 状态动作函数
void StateWork(State state) {
    switch (state) {
```

```
        case STATE1:
            State1Work();
            break;
        case STATE2:
            State2Work();
            break;
        case STATE3:
            State3Work();
            break;
        ...
        case STATEn:
            StateNWork();
            break;
    }
}
// 状态机具体实现
void FSMWork(void) {
    State state = STATE1;       // 定义并初始化State类型的变量
    state = StateJump(state);   // 状态跳转
    StateWork(state);           // 状态动作执行
}
```

- switch-case的缺点：

  switch-case在无优化编译的模式下（Keil、IAR等IDE在编译单片机程序时一般都把优化关闭，即为无优化模式），case的判断是从第一个case开始，逐渐往下，直到case条件符合为止，若遍历结束后case条件仍不符合则执行default或者退出switch；所以其时间复杂度为O(n)；当状态数量较多时，该程序的执行效率相对于时间复杂度为O(1)的程序要低很多。所以，接下去要讲解如何使用函数指针来实现状态机。

2. 函数指针实现

所谓函数指针，顾名思义为指向函数的指针，本质上即为一种指针；若将函数指针存放在一个数组当中，便可以通过访问数组下标来执行相应的函数指针所指向的函数，**此过程的时间复杂度为O(1)**。

在状态机中，将条件跳转函数与状态动作函数分别存放在两个数组当中（此时的数组为函数指针数组），以枚举变量作为数组的下标，即可实现一个有限状态机；具体代码如下：

```
// 声明状态枚举变量
typedef enum {
    STATE1 = 0,
    STATE2,
    STATE3,
    ...
    STATEn,
} State;
// 声明条件跳转函数
State State1Jump(void);
State State2Jump(void);
State State3Jump(void);
```

```
...
State StateNJump(void);
// 声明状态动作函数
void State1Work(void);
void State2Work(void);
void State3Work(void);
...
void StateNWork(void);
// 定义存放指向条件跳转函数的函数指针的函数指针数组
// 数组内的函数指针按照枚举变量中的相应状态顺序存放
State (*Jump[])(void) = {
    State1Jump,
    State2Jump,
    State3Jump,
    ...
    StateNJump,
};
// 定义存放指向状态动作函数的函数指针的函数指针数组
// 数组内的函数指针按照枚举变量中的相应状态顺序存放
void (*Work[])(void)  = {
    State1Work,
    State2Work,
    State3Work,
    ...
    StateNWork,
};
// 状态机具体实现
void FSMWork(void) {
    State state = STATE1;    // 定义并初始化State类型的变量
    state = Jump[state]();   // 状态跳转
    Work[state]();           // 状态动作执行
}
```

三、总结

在程序设计中，当状态数量较多时，尽量使用**函数指针**来实现相应的状态机，从而减去在状态判断上所花费的时间开销。