分类：

本文讲述层次状态机实现形式中的行为继承。从行为继承与类继承之间的OO类相似来看，一个成功的层次状态机应该能够模拟下列属于C++对象模型。

        使用和维护简单

        应允许转态机拓扑容易改变，不应要求转换连接的人工代码，所需要的修改限制在代码的一个地方。

        提供好的运行-时间效率和小的存储。

        遵守C++中的“零额外开销”原则。

为了满足上面的要求，层次状态机的实现着重于下面的主要元素：

        完全支持行为继承的层次关系

        用状态进入和退出动作实现有保证得初始化和清除

        通过类继承支持规定的状态模型

1.      基本要素

(1)     状态：在层次状态的情形下，状态处理器必须返回朝状态，这导致层次状态处理特征标记的递归定义。构造这种特征标记在C++是不可能的，于是定义下面宏来近似：

typedef void (\* QPseudoState)(QEVENT const \*pEvent);

typedef QPseudoState (\* QState)(QEVENT const \*pEvent);

(2)     进入/退出动作和初始状态：这些元素是状态专有的特征，在转态中他们被定义，而特别的是与到达状态所进过的路径无关。保留信号的转态定义为：

typedef enum tagQSIG

{

            Q\_EMPTY\_SIG = 1,

            Q\_INIT\_SIG,

            Q\_ENTRY\_SIG,

            Q\_EXIT\_SIG,

            Q\_USER\_SIG,

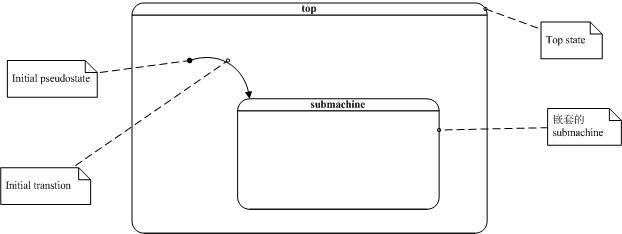
} QSIG;

状态处理机可以用一般的switch语句规定适当的用例处理这些信号，可以自由的执行相应信号的操作。

(3)     状态转换：状态处理机用Q\_TRAN实现状态转换，并且在原状态的上下文中执行动作，即：改变状态之前调用Q\_TRAN（和UML规定不一致）。

#define Q\_TRAN(target\_) Q\_TranDyc((QState)(target\_))

(4)     Top状态和初始伪状态：每个层次状态机都有一个Top状态，包括整个状态的所有其它元素。Top状态没有超状态，用户也不能覆盖；Top状态的唯一目的是提供状态层次的最高的根，使最高处理器能返回Top; Top状态唯一能订制的是初始化。初始化伪状态处理机仅规定初始化转换，必须指明被窃套的Top状态的状态机的缺省状态。



2.         实现代码：

(1)         头文件代码：

#ifndef STATE\_INHERIT\_H

#define STATE\_INHERIT\_H

typedef unsigned short QSIG;

// Define the signal of state machine

enum

{

    Q\_EMPTY\_SIG = 0,

    Q\_INIT\_SIG = 1,

    Q\_ENTRY\_SIG,

    Q\_EXIT\_SIG,

    Q\_USER\_SIG

};

// Define the signal of state machine

typedef struct tagQEVENT

{

  QSIG sig;

  unsigned char \*pEvent1;

  unsigned char \*pEvent2;

  // TODO: add fields to the event

} QEVENT;

// define state data type

typedef void (\* QPseudoState)(QEVENT const \*pEvent);

typedef QPseudoState (\* QState)(QEVENT const \*pEvent);

typedef QPseudoState QSTATE;

#define Q\_TRIGGER(state, sig) \

   (QState)(\*(state))((QEVENT\*)&pkgStdEvt[sig])

// define a transation that don't change the state,

// just treat the pEvent with the target state.

// this is used by concurrent state

#define Q\_INIT(target\_) Init\_((QState)(target\_));

#define Q\_TRAN(target\_) Q\_TranDyc((QState)(target\_));

void Init\_(QState target);

void Q\_Init(QSTATE target);

void Q\_Initial(QEVENT const\* pQevt);

void Q\_Dispatch(QEVENT const\* pQevt);

void Q\_TranDyc(QState target);

#endif //STATE\_INHERIT\_H

(2)         实体代码：

#include <stdio.h>

#include <assert.h>

#include "state\_inherit.h"

static QState srcState;         // source state

static QState actState;         // active state

static QEVENT const pkgStdEvt[] =

{

   {Q\_EMPTY\_SIG, 0, 0},

   {Q\_INIT\_SIG, 0, 0},

   {Q\_ENTRY\_SIG, 0, 0},

   {Q\_EXIT\_SIG, 0, 0}

};

void Q\_Initial(QEVENT const\* pQevt)

{

    printf("Top\_Init;");

}

void Q\_Dispatch(QEVENT const\* pQevt)

{

    for (srcState = actState; srcState;

         srcState = (QState)(\*srcState)(pQevt))

    {}

}

void Init\_(QState target)

{

    actState = target;

}

void Q\_Init(QSTATE target)

{

    register QState s;

    actState = (QState)target;

    srcState = (QState)Q\_Initial;

    s = actState;                               // save actState in a temporary

    (\*(QPseudoState)srcState)((QEVENT\*)0);      // top-most initial tran.

                                    // initial transition must go one level deep

    s = actState;                                     // update the temporary

    Q\_TRIGGER(s, Q\_ENTRY\_SIG);           // enter the state

    while (0 == Q\_TRIGGER(s, Q\_INIT\_SIG))

    {

        // init handled

        // initial transition must go one level deep

        s = actState;

        Q\_TRIGGER(s, Q\_ENTRY\_SIG);       // enter the substate

    }

}

void Q\_TranDyc(QState target)

{

    QState entry[8], p, q, s, \*e, \*lca;

    for (s = actState; s != srcState; )

    {

        QState t;

        t = Q\_TRIGGER(s, Q\_EXIT\_SIG);

        if (t)

        {

            // exit action unhandled, t points to superstate

            s = t;

        }

        else

        {

           // exit action handled, elicit superstate

            s = Q\_TRIGGER(s, Q\_EMPTY\_SIG);

        }

    }

    \*(e = &entry[0]) = 0;

    \*(++e) = target;                              // assume entry to target

    // (a) check source == target (transition to self)

    if (srcState == target)

    {

        Q\_TRIGGER(srcState, Q\_EXIT\_SIG);            // exit source

        goto inLCA;

    }

    // (b) check source == target->super

    p = Q\_TRIGGER(target, Q\_EMPTY\_SIG);

    if (srcState == p) goto inLCA;

    //(c) check source->super == target->super (most common)

    q = Q\_TRIGGER(srcState, Q\_EMPTY\_SIG);

    if (q == p)

    {

        Q\_TRIGGER(srcState, Q\_EXIT\_SIG);            // exit source

        goto inLCA;

    }

    // (d) check source->super == target

    if (q == target)

    {

        Q\_TRIGGER(srcState, Q\_EXIT\_SIG);           // exit source

        --e;                                     // not enter the LCA

        goto inLCA;

    }

    // (e) check rest of source == target->super->super... hierarchy

    \*(++e) = p;

    for (s = Q\_TRIGGER(p, Q\_EMPTY\_SIG); s; s = Q\_TRIGGER(s, Q\_EMPTY\_SIG))

    {

        if (srcState == s)

        {

            goto inLCA;

        }

        \*(++e) = s;

    }

    Q\_TRIGGER(srcState, Q\_EXIT\_SIG);                // exit source

    // (f) check rest of source->super == target->super->super...

    for (lca = e; \*lca; --lca)

    {

        if (q == \*lca)

        {

            e = lca - 1;                      // do not enter the LCA

            goto inLCA;

        }

    }

    // (g) check each srcState->super->super..for each target...

    for (s = q; s; s = Q\_TRIGGER(s, Q\_EMPTY\_SIG))

    {

        for (lca = e; \*lca; --lca)

        {

            if (s == \*lca)

            {

                e = lca - 1;                           // do not enter the LCA

                goto inLCA;

            }

        }

        Q\_TRIGGER(s, Q\_EXIT\_SIG);                        // exit s

    }

    assert(0);                                         // malformed HSM

    inLCA:         // now we are in the LCA of srcState and target

    assert(e < &entry[sizeof(entry) / sizeof(\*entry)]); // entry fits

    while (s = \*e--)

    {

        // retrace the entry path in reverse order

        Q\_TRIGGER(s, Q\_ENTRY\_SIG);                       // enter s

    }

    actState = target;                   // update current state

    while (0 == Q\_TRIGGER(target, Q\_INIT\_SIG))

    {

        // initial transition must go one level deep

        assert(target == Q\_TRIGGER(actState, Q\_EMPTY\_SIG));

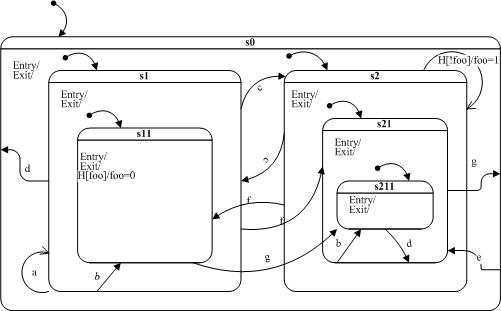
        target = actState;

        Q\_TRIGGER(target, Q\_ENTRY\_SIG);                // enter target

    }

3.         范例：

(1)         范例状态图



(2)         范例代码

#include <stdio.h>

#include "state\_inherit.h"

QSTATE s0(QEVENT const \*e);

QSTATE s1(QEVENT const \*e);

QSTATE s2(QEVENT const \*e);

QSTATE s11(QEVENT const \*e);

QSTATE s21(QEVENT const \*e);

QSTATE s211(QEVENT const \*e);

QSTATE Q\_Top(QEVENT const \*e);

static void Initial(QEVENT const \*e);

static bool bFoo;

enum QSignals {

   A\_SIG = Q\_USER\_SIG,

   B\_SIG, C\_SIG, D\_SIG, E\_SIG, F\_SIG, G\_SIG, H\_SIG

};

static const QEVENT testQEvt[] =

{

   {A\_SIG, 0, 0}, {B\_SIG, 0, 0}, {C\_SIG, 0, 0}, {D\_SIG, 0, 0},

   {E\_SIG, 0, 0}, {F\_SIG, 0, 0}, {G\_SIG, 0, 0}, {H\_SIG, 0, 0}

};

int main()

{

   printf("Hiberarchy state machine testing\n");

   Initial(0);       // trigger initial transition

   for (;;)

   {

      char c;

      printf("\nSignal<-");

      c = getc(stdin);

      getc(stdin);              // discard '\n'

      if (c < 'a' || 'h' < c) {

         return 0;

      }

      Q\_Dispatch(&testQEvt[c - 'a']); // dispatch

   }

   return 0;

}

static QSTATE Q\_Top(QEVENT const \*e)

{

    return 0;

}

void Initial(QEVENT const \*e)

{

   bFoo = false;

   Q\_Init((QSTATE)s0);

}

QSTATE s0(QEVENT const \*e) {

    if (e != NULL)

    {

       switch (e->sig)

       {

       case Q\_ENTRY\_SIG: printf("s0-ENTRY;"); return 0;

       case Q\_EXIT\_SIG: printf("s0-EXIT;");  return 0;

       case Q\_INIT\_SIG: printf("s0-INIT;"); Q\_INIT(s1); return 0;

       case E\_SIG:      printf("s0-E;"); Q\_TRAN(s211);  return 0;

       }

    }

   return (QSTATE)Q\_Top;

}

QSTATE s1(QEVENT const \*e) {

   switch (e->sig) {

   case Q\_ENTRY\_SIG: printf("s1-ENTRY;"); return 0;

   case Q\_EXIT\_SIG: printf("s1-EXIT;");   return 0;

   case Q\_INIT\_SIG: printf("s1-INIT;");Q\_INIT(s11); return 0;

   case A\_SIG:      printf("s1-A;");   Q\_TRAN(s1);  return 0;

   case B\_SIG:      printf("s1-B;");   Q\_TRAN(s11); return 0;

   case C\_SIG:      printf("s1-C;");   Q\_TRAN(s2);  return 0;

   case D\_SIG:      printf("s1-D;");   Q\_TRAN(s0);  return 0;

   case F\_SIG:      printf("s1-F;");   Q\_TRAN(s211);return 0;

   }

   return (QSTATE)s0;

}

QSTATE s11(QEVENT const \*e) {

   switch (e->sig) {

   case Q\_ENTRY\_SIG: printf("s11-ENTRY;"); return 0;

   case Q\_EXIT\_SIG:  printf("s11-EXIT;");  return 0;

   case G\_SIG:  printf("s11-G;"); Q\_TRAN(s211); return 0;

   case H\_SIG:                 // internal transition with a guard

      if (bFoo)

      {                      // test the guard condition

         printf("s11-H;");

         bFoo = false;

         return 0;

      }

      break;

   }

   return (QSTATE)s1;

}

QSTATE s2( QEVENT const \*e) {

   switch (e->sig) {

   case Q\_ENTRY\_SIG: printf("s2-ENTRY;"); return 0;

   case Q\_EXIT\_SIG: printf("s2-EXIT;");   return 0;

   case Q\_INIT\_SIG: printf("s2-INIT;");Q\_INIT(s21); return 0;

   case C\_SIG:      printf("s2-C;");   Q\_TRAN(s1);  return 0;

   case F\_SIG:      printf("s2-F;");   Q\_TRAN(s11); return 0;

   }

   return (QSTATE)s0;

}

QSTATE s21(QEVENT const \*e) {

   switch (e->sig) {

   case Q\_ENTRY\_SIG: printf("s21-ENTRY;"); return 0;

   case Q\_EXIT\_SIG: printf("s21-EXIT;");   return 0;

   case Q\_INIT\_SIG:printf("s21-INIT;");Q\_INIT(s211);return 0;

   case B\_SIG:     printf("s21-C;");   Q\_TRAN(s211);return 0;

   case H\_SIG:                     // self transition with a guard

      if (!bFoo)

      {            // test the guard condition

         printf("s21-H;");

         bFoo = true;

         Q\_TRAN(s21);                   // self transition

         return 0;

      }

      break;                     //break to return the superstate

   }

   return (QSTATE)s2;

}

QSTATE s211(QEVENT const \*e) {

   switch (e->sig) {

   case Q\_ENTRY\_SIG: printf("s211-ENTRY;"); return 0;

   case Q\_EXIT\_SIG:  printf("s211-EXIT;");  return 0;

   case D\_SIG: printf("s211-D;"); Q\_TRAN(s21); return 0;

   case G\_SIG: printf("s211-G;"); Q\_TRAN(s0);  return 0;

   }

   return (QSTATE)s21;

}

(3)         输出结果：

Hiberarchy state machine testing

Top\_Init;s0-ENTRY;s0-INIT;s1-ENTRY;s1-INIT;s11-ENTRY;

Signal<-a

s1-A;s11-EXIT;s1-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;

Signal<-e

s0-E;s11-EXIT;s1-EXIT;s2-ENTRY;s21-ENTRY;s211-ENTRY;

Signal<-e

s0-E;s211-EXIT;s21-EXIT;s2-EXIT;s2-ENTRY;s21-ENTRY;s211-ENTRY;

Signal<-a

Signal<-h

s21-H;s211-EXIT;s21-EXIT;s21-ENTRY;s21-INIT;s211-ENTRY;

Signal<-h

Signal<-x

说明：上面功能都是通过C语言实现的，大家可以将其用C++实现，共享一下。

