附录C 消息传递框架与完整的ATM示例

ATM：自动取款机。

1回到第4章，我举了一个使用消息传递框架在线程间发送信息的例子。这里就会使用这个实现来完成ATM功能。下面完整代码就是功能的实现，包括消息传递框架。

清单C.1实现了一个消息队列。其可以将消息以指针(指向基类)的方式存储在列表中；指定消息类型会由基类派生模板进行处理。推送包装类的构造实例，以及存储指向这个实例的指针；弹出实例的时候，将会返回指向其的指针。因为message_base类没有任何成员函数，在访问存储消息之前，弹出线程就需要将指针转为wrapped_message指针。

清单C.1 简单的消息队列

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <memory>

namespace messaging
{
  struct message_base  // 队列项的基础类
  {
    virtual ~message_base()
    {}
  };

  template<typename Msg>
  struct wrapped_message:  // 每个消息类型都需要特化
    message_base
  {
    Msg contents;

    explicit wrapped_message(Msg const& contents_):
      contents(contents_)
    {}
  };

  class queue  // 我们的队列
  {
    std::mutex m;
    std::condition_variable c;
    std::queue<std::shared_ptr<message_base> > q;  // 实际存储指向message_base类
  public:
    template<typename T>
    void push(T const& msg)
    {
      std::lock_guard<std::mutex> lk(m);
      q.push(std::make_shared<wrapped_message<T> >(msg));  // 包装已传递的信息，
      c.notify_all();
    }

    std::shared_ptr<message_base> wait_and_pop()
    {
      std::unique_lock<std::mutex> lk(m);
      c.wait(lk,[&]{return !q.empty();});  // 当队列为空时阻塞
      auto res=q.front();
      q.pop();
      return res;
    }
  };
}
```

发送通过sender类(见清单C.2)实例处理过的消息。只能对已推送到队列中的消息进行包装。对sender实例的拷贝，只是拷贝了指向队列的指针，而非队列本身。

清单C.2 sender类

```
namespace messaging
{
  class sender
  {
    queue*q;  // sender是一个队列指针的包装类
  public:
    sender():  // sender无队列(默认构造函数)
      q(nullptr)
    {}

    explicit sender(queue*q_):  // 从指向队列的指针进行构造
      q(q_)
    {}
    template<typename Message>
    void send(Message const& msg)
    {
      if(q)
      {
        q->push(msg);  // 将发送信息推送给队列
      }
    }
  };
}
```

接收信息部分有些麻烦。不仅要等待队列中的消息，还要检查消息类型是否与所等待的消息类型匹配，并调用处理函数进行处理。那么就从receiver类的实现开始吧。

清单C.3 receiver类

```
namespace messaging
{
  class receiver
  {
    queue q;  // 接受者拥有对应队列
  public:
    operator sender()  // 允许将类中队列隐式转化为一个sender队列
    {
      return sender(&q);
    }
    dispatcher wait()  // 等待对队列进行调度
    {
      return dispatcher(&q);
    }
  };
}
```

sender只是引用一个消息队列，而receiver是拥有一个队列。可以使用隐式转换的方式获取sender引用的类。难点在于wait()中的调度。这里创建了一个dispatcher对象引用receiver中的队列。dispatcher类实现会在下一个清单中看到；如你所见，任务是在析构函数中完成的。在这个例子中，所要做的工作是对消息进行等待，以及对其进行调度。

清单C.4 dispatcher类

```
namespace messaging
```

```
{
  class close_queue  // 用于关闭队列的消息
  {};

  class dispatcher
  {
    queue* q;
    bool chained;

    dispatcher(dispatcher const&)=delete;  // dispatcher实例不能被拷贝
    dispatcher& operator=(dispatcher const&)=delete;

    template<
      typename Dispatcher,
      typename Msg,
      typename Func>  // 允许TemplateDispatcher实例访问内部成员
    friend class TemplateDispatcher;

    void wait_and_dispatch()
    {
      for(;;)  // 1 循环，等待调度消息
      {
        auto msg=q->wait_and_pop();
        dispatch(msg);
      }
    }

    bool dispatch(  // 2 dispatch()会检查close_queue消息，然后抛出
      std::shared_ptr<message_base> const& msg)
    {
      if(dynamic_cast<wrapped_message<close_queue>*>(msg.get()))
      {
        throw close_queue();
      }
      return false;
    }
  public:
    dispatcher(dispatcher&& other):  // dispatcher实例可以移动
      q(other.q),chained(other.chained)
    {
      other.chained=true;  // 源不能等待消息
    }

    explicit dispatcher(queue* q_):
      q(q_),chained(false)
    {}

    template<typename Message,typename Func>
    TemplateDispatcher<dispatcher,Message,Func>
    handle(Func&& f)  // 3 使用TemplateDispatcher处理指定类型的消息
    {
      return TemplateDispatcher<dispatcher,Message,Func>(
        q,this,std::forward<Func>(f));
    }

    ~dispatcher() noexcept(false)  // 4 析构函数可能会抛出异常
    {
      if(!chained)
      {
        wait_and_dispatch();
      }
    }
  };
}
```

从wait()返回的dispatcher实例将马上被销毁，因为是临时变量，也向前文提到的，析构函数在这里做真正的工作。析构函数调用wait_and_dispatch()函数，这个函数中有一个循环①，等待消息的传入(这样才能进行弹出操作)，然后将消息传递给dispatch()函数。dispatch()函数本身②很简单；会检查小时是否是一个close_queue消息，当是close_queue消息时，抛出一个异常；如果不是，函数将会返回false来表明消息没有被处理。因为会抛出close_queue异常，所以析构函数会标示为 `noexcept(false)` ；在没有任何标识的情况下，一般情况下析构函数都 `noexcept(true)` ④型，这表示没有任何异常抛出，并且close_queue异常将会使程序终止。

虽然，不会经常的去调用wait()函数，不过，在大多数时间里，你都希望对一条消息进行处理。这时就需要handle()成员函数③的加入。这个函数是一个模板，并且消息类型不可推断，所以你需要指定需要处理的消息类型，并且传入函数(或可调用对象)进行处理，并将队列传入当前dispatcher对象的handle()函数。这将在清单C.5中展示。这就是为什么，在测试析构函数中的chained值前，要等待消息耳朵原因；不仅是避免“移动”类型的对象对消息进行等待，而且允许将等待状态转移到新的TemplateDispatcher实例中。

清单C.5 TemplateDispatcher类模板

```
namespace messaging
{
  template<typename PreviousDispatcher,typename Msg,typename Func>
  class TemplateDispatcher
  {
    queue* q;
    PreviousDispatcher* prev;
    Func f;
    bool chained;

    TemplateDispatcher(TemplateDispatcher const&)=delete;
    TemplateDispatcher& operator=(TemplateDispatcher const&)=delete;

    template<typename Dispatcher,typename OtherMsg,typename OtherFunc>
    friend class TemplateDispatcher;  // 所有特化的TemplateDispatcher类型实例都是

    void wait_and_dispatch()
    {
      for(;;)
      {
        auto msg=q->wait_and_pop();
        if(dispatch(msg))  // 1 如果消息处理过后，会跳出循环
          break;
      }
    }

    bool dispatch(std::shared_ptr<message_base> const& msg)
    {
      if(wrapped_message<Msg>* wrapper=
         dynamic_cast<wrapped_message<Msg>*>(msg.get()))  // 2 检查消息类型，并且
      {
        f(wrapper->contents);
        return true;
      }
      else
      {
        return prev->dispatch(msg);  // 3 链接到之前的调度器上
      }
    }
  public:
    TemplateDispatcher(TemplateDispatcher&& other):
        q(other.q),prev(other.prev),f(std::move(other.f)),
        chained(other.chained)
    {
      other.chained=true;
    }
    TemplateDispatcher(queue* q_,PreviousDispatcher* prev_,Func&& f_):
        q(q_),prev(prev_),f(std::forward<Func>(f_)),chained(false)
    {
      prev_->chained=true;
    }

    template<typename OtherMsg,typename OtherFunc>
    TemplateDispatcher<TemplateDispatcher,OtherMsg,OtherFunc>
    handle(OtherFunc&& of)  // 4 可以链接其他处理器
    {
      return TemplateDispatcher<
          TemplateDispatcher,OtherMsg,OtherFunc>(
          q,this,std::forward<OtherFunc>(of));
    }

    ~TemplateDispatcher() noexcept(false)  // 5 这个析构函数也是noexcept(false)的
    {
      if(!chained)
      {
        wait_and_dispatch();
      }
    }
  };
}
```

TemplateDispatcher<>类模板仿照了dispatcher类，二者几乎相同。特别是在析构函数上，都是调用wait_and_dispatch()等待处理消息。

在处理消息的过程中，如果不抛出异常，就需要检查一下在循环中①，消息是否已经得到了处理。当成功的处理了一条消息，处理过程就可以停止，这样就可以等待下一组消息的传入了。当获取了一个和指定类型匹配的消息，使用函数调用的方式②，就要好于抛出异常(虽然，处理函数也可能会抛出异常)。如果消息类型不匹配，那么就可以链接前一个调度器③。在第一个实例中，dispatcher实例确实作为一个调度器，当在handle()④函数中进行链接后，就允许处理多种类型的消息。在链接了之前的TemplateDispatcher<>实例后，当消息类型和当前的调度器类型不匹配的时候，调度链会依次的向前寻找类型匹配的调度器。因为任何调度器都可能抛出异常(包括dispatcher中对close_queue消息进行处理的默认处理器)，析构函数在这里会再次被声明为 `noexcept(false)` ⑤。

这种简单的架构允许你想队列推送任何类型的消息，并且调度器有选择的与接收端的消息进行匹配。同样，也允许为了推送消息，将消息队列的引用进行传递的同时，保持接收端的私有性。

为了完成第4章的例子，消息的组成将在清单C.6中给出，各种状态机将在清单C.7,C.8和C.9中给出。最后，驱动代码将在C.10给出。

清单C.6 ATM消息

```
struct withdraw
{
  std::string account;
  unsigned amount;
  mutable messaging::sender atm_queue;

  withdraw(std::string const& account_,
           unsigned amount_,
           messaging::sender atm_queue_):
    account(account_),amount(amount_),
    atm_queue(atm_queue_)
  {}
};

struct withdraw_ok
{};

struct withdraw_denied
{};

struct cancel_withdrawal
{
  std::string account;
  unsigned amount;
  cancel_withdrawal(std::string const& account_,
                    unsigned amount_):
    account(account_),amount(amount_)
  {}
};

struct withdrawal_processed
{
  std::string account;
  unsigned amount;
  withdrawal_processed(std::string const& account_,
                       unsigned amount_):
    account(account_),amount(amount_)
  {}
};

struct card_inserted
{
  std::string account;
  explicit card_inserted(std::string const& account_):
    account(account_)
  {}
};

struct digit_pressed
{
  char digit;
  explicit digit_pressed(char digit_):
    digit(digit_)
  {}
};

struct clear_last_pressed
{};

struct eject_card
{};

struct withdraw_pressed
{
  unsigned amount;
  explicit withdraw_pressed(unsigned amount_):
    amount(amount_)
  {}
};

struct cancel_pressed
{};

struct issue_money
{
  unsigned amount;
  issue_money(unsigned amount_):
    amount(amount_)
  {}
};

struct verify_pin
{
  std::string account;
  std::string pin;
  mutable messaging::sender atm_queue;

  verify_pin(std::string const& account_,std::string const& pin_,
             messaging::sender atm_queue_):
    account(account_),pin(pin_),atm_queue(atm_queue_)
  {}
};

struct pin_verified
{};

struct pin_incorrect
{};

struct display_enter_pin
{};

struct display_enter_card
{};

struct display_insufficient_funds
{};

struct display_withdrawal_cancelled
{};

struct display_pin_incorrect_message
{};

struct display_withdrawal_options
{};

struct get_balance
{
  std::string account;
  mutable messaging::sender atm_queue;

  get_balance(std::string const& account_,messaging::sender atm_queue_):
    account(account_),atm_queue(atm_queue_)
  {}
};

struct balance
{
  unsigned amount;
  explicit balance(unsigned amount_):
    amount(amount_)
  {}
};

```

```
struct display_balance
{
  unsigned amount;
  explicit display_balance(unsigned amount_):
    amount(amount_)
  {}
};

struct balance_pressed
{};
```

清单C.7 ATM状态机

```
class atm
{
  messaging::receiver incoming;
  messaging::sender bank;
  messaging::sender interface_hardware;

  void (atm::*state)();

  std::string account;
  unsigned withdrawal_amount;
  std::string pin;

  void process_withdrawal()
  {
    incoming.wait()
      .handle<withdraw_ok>(
       [&](withdraw_ok const& msg)
       {
         interface_hardware.send(
           issue_money(withdrawal_amount));

         bank.send(
           withdrawal_processed(account,withdrawal_amount));

         state=&atm::done_processing;
       })
      .handle<withdraw_denied>(
       [&](withdraw_denied const& msg)
       {
         interface_hardware.send(display_insufficient_funds());

         state=&atm::done_processing;
       })
      .handle<cancel_pressed>(
       [&](cancel_pressed const& msg)
       {
         bank.send(
           cancel_withdrawal(account,withdrawal_amount));

         interface_hardware.send(
           display_withdrawal_cancelled());

         state=&atm::done_processing;
       });
   }

  void process_balance()
  {
    incoming.wait()
      .handle<balance>(
       [&](balance const& msg)
       {
         interface_hardware.send(display_balance(msg.amount));

         state=&atm::wait_for_action;
       })
      .handle<cancel_pressed>(
       [&](cancel_pressed const& msg)
       {
         state=&atm::done_processing;
       });
  }

  void wait_for_action()
  {
    interface_hardware.send(display_withdrawal_options());

    incoming.wait()
      .handle<withdraw_pressed>(
       [&](withdraw_pressed const& msg)
       {
         withdrawal_amount=msg.amount;
         bank.send(withdraw(account,msg.amount,incoming));
         state=&atm::process_withdrawal;
       })
      .handle<balance_pressed>(
       [&](balance_pressed const& msg)
       {
         bank.send(get_balance(account,incoming));
         state=&atm::process_balance;
       })
      .handle<cancel_pressed>(
       [&](cancel_pressed const& msg)
       {
         state=&atm::done_processing;
       });
  }

  void verifying_pin()
  {
    incoming.wait()
      .handle<pin_verified>(
       [&](pin_verified const& msg)
       {
         state=&atm::wait_for_action;
       })
      .handle<pin_incorrect>(
       [&](pin_incorrect const& msg)
       {
         interface_hardware.send(
         display_pin_incorrect_message());
         state=&atm::done_processing;
       })
      .handle<cancel_pressed>(
       [&](cancel_pressed const& msg)
       {
         state=&atm::done_processing;
       });
  }

  void getting_pin()
  {
    incoming.wait()
      .handle<digit_pressed>(
       [&](digit_pressed const& msg)
       {
         unsigned const pin_length=4;
         pin+=msg.digit;

         if(pin.length()==pin_length)
         {
           bank.send(verify_pin(account,pin,incoming));
           state=&atm::verifying_pin;
         }
       })
      .handle<clear_last_pressed>(
       [&](clear_last_pressed const& msg)
       {
         if(!pin.empty())
         {
           pin.pop_back();
         }
       })
      .handle<cancel_pressed>(
       [&](cancel_pressed const& msg)
       {
         state=&atm::done_processing;
       });
  }

  void waiting_for_card()
  {
    interface_hardware.send(display_enter_card());

    incoming.wait()
      .handle<card_inserted>(
       [&](card_inserted const& msg)
       {
         account=msg.account;
         pin="";
         interface_hardware.send(display_enter_pin());
         state=&atm::getting_pin;
       });
  }

  void done_processing()
  {
    interface_hardware.send(eject_card());
    state=&atm::waiting_for_card;
  }

  atm(atm const&)=delete;
  atm& operator=(atm const&)=delete;
public:
  atm(messaging::sender bank_,
  messaging::sender interface_hardware_):
```

```
  bank(bank_),interface_hardware(interface_hardware_)
  {}

  void done()
  {
    get_sender().send(messaging::close_queue());
  }

  void run()
  {
    state=&atm::waiting_for_card;
    try
    {
      for(;;)
      {
        (this->*state)();
      }
    }
    catch(messaging::close_queue const&)
    {
    }
  }

  messaging::sender get_sender()
  {
    return incoming;
  }
};
```

清单C.8 银行状态机

```
class bank_machine
{
  messaging::receiver incoming;
  unsigned balance;
public:
  bank_machine():

  balance(199)
  {}

  void done()
  {
    get_sender().send(messaging::close_queue());
  }

  void run()
  {
    try
    {
      for(;;)
      {
        incoming.wait()
          .handle<verify_pin>(
           [&](verify_pin const& msg)
           {
             if(msg.pin=="1937")
             {
               msg.atm_queue.send(pin_verified());
             }
             else
             {
               msg.atm_queue.send(pin_incorrect());
             }
           })
          .handle<withdraw>(
           [&](withdraw const& msg)
           {
             if(balance>=msg.amount)
             {
               msg.atm_queue.send(withdraw_ok());
               balance-=msg.amount;
             }
             else
             {
               msg.atm_queue.send(withdraw_denied());
             }
           })
          .handle<get_balance>(
           [&](get_balance const& msg)
           {
             msg.atm_queue.send(::balance(balance));
           })
          .handle<withdrawal_processed>(
           [&](withdrawal_processed const& msg)
           {
           })
          .handle<cancel_withdrawal>(
           [&](cancel_withdrawal const& msg)
           {
           });
      }
    }
    catch(messaging::close_queue const&)
    {
    }
  }

  messaging::sender get_sender()
  {
  return incoming;
  }
};
```

清单C.9 用户状态机

```
class interface_machine
{
  messaging::receiver incoming;
public:
  void done()
  {
    get_sender().send(messaging::close_queue());
  }

  void run()
  {
    try
    {
      for(;;)
      {
        incoming.wait()
          .handle<issue_money>(
           [&](issue_money const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Issuing "
                 <<msg.amount<<std::endl;
             }
           })
          .handle<display_insufficient_funds>(
           [&](display_insufficient_funds const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Insufficient funds"<<std::endl;
             }
           })
          .handle<display_enter_pin>(
           [&](display_enter_pin const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Please enter your PIN (0-9)"<<std::endl;
             }
           })
          .handle<display_enter_card>(
           [&](display_enter_card const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Please enter your card (I)"
                 <<std::endl;
             }
           })
          .handle<display_balance>(
           [&](display_balance const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout
                 <<"The balance of your account is "
                 <<msg.amount<<std::endl;
             }
           })
          .handle<display_withdrawal_options>(
           [&](display_withdrawal_options const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Withdraw 50? (w)"<<std::endl;
               std::cout<<"Display Balance? (b)"
                 <<std::endl;
               std::cout<<"Cancel? (c)"<<std::endl;
             }
           })
          .handle<display_withdrawal_cancelled>(
           [&](display_withdrawal_cancelled const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Withdrawal cancelled"
                 <<std::endl;
             }
           })
          .handle<display_pin_incorrect_message>(
           [&](display_pin_incorrect_message const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"PIN incorrect"<<std::endl;
             }
           })
          .handle<eject_card>(
           [&](eject_card const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Ejecting card"<<std::endl;
             }
           });
      }
    }
    catch(messaging::close_queue&)
    {
    }
  }

  messaging::sender get_sender()
  {
    return incoming;
  }
};
```

清单C.10 驱动代码

```
int main()
{
  bank_machine bank;
  interface_machine interface_hardware;

  atm machine(bank.get_sender(),interface_hardware.get_sender());

  std::thread bank_thread(&bank_machine::run,&bank);
  std::thread if_thread(&interface_machine::run,&interface_hardware);
  std::thread atm_thread(&atm::run,&machine);

  messaging::sender atmqueue(machine.get_sender());

  bool quit_pressed=false;

  while(!quit_pressed)
  {
    char c=getchar();
    switch(c)
    {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
      atmqueue.send(digit_pressed(c));
      break;
    case 'b':
      atmqueue.send(balance_pressed());
      break;
    case 'w':
      atmqueue.send(withdraw_pressed(50));
      break;
    case 'c':
      atmqueue.send(cancel_pressed());
      break;
    case 'q':
      quit_pressed=true;
      break;
    case 'i':
      atmqueue.send(card_inserted("acc1234"));
      break;
    }
  }

  bank.done();
  machine.done();
  interface_hardware.done();

  atm_thread.join();
  bank_thread.join();
  if_thread.join();
}
```