

视图提供业务数据。同一个模型可以被多个视图重用。

3. 控制器

控制器接收用户的输入并调用模型和视图去完成用户的请求。当用户在视图上选择按钮或菜单时，控制器接收请求并调用相应的模型组件去处理请求，然后调用相应的视图来显示模型返回的数据。

如图 13-2 所示，MVC 的 3 个模块也可以看做软件的 3 个层次，最上层为视图层，中间为控制器层，下层为模型层。总地说来，层与层之间为自上而下的依赖关系，下层组件为上层组件提供服务。视图层与控制器层依赖模型层来处理业务逻辑和提供业务数据。此外，层与层之间还存在两处自下而上的调用，一处是控制器层调用视图层来显示业务数据，另一处是模型层通知客户层同步刷新界面。为了提高每个层的独立性，应该使每个层对外公开接口，封装实现细节。

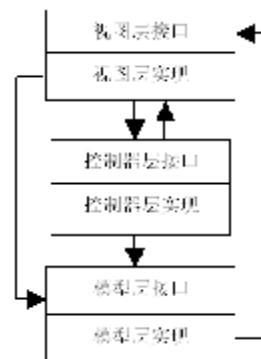


图 13-2 MVC 的 3 个模块也可以看做软件的 3 个层次

4. MVC 处理过程

如图 13-3 所示，首先用户在视图提供的界面上发出请求，视图把请求转发给控制器，控制器调用相应的模型来处理用户请求，模型进行相应的业务逻辑处理，并返回数据。最后控制器调用相应的视图来显示模型返回的数据。

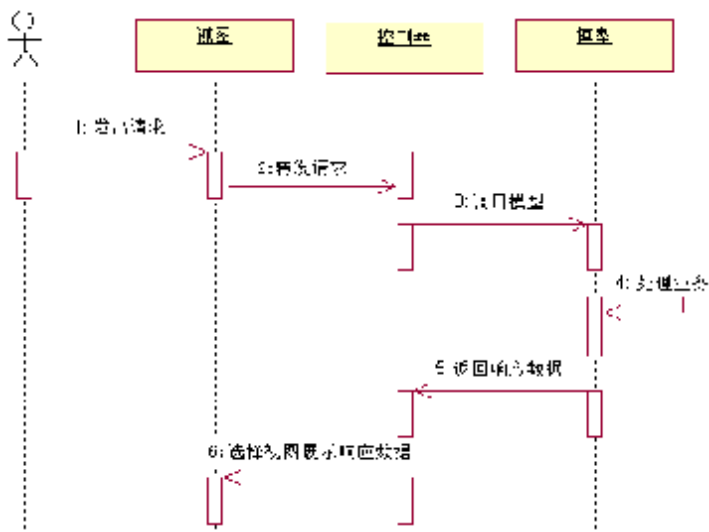


图 13-3 MVC 的处理过程

5. MVC 的优点

首先，多个视图能共享一个模型。在 MVC 设计模式中，模型响应用户请求并返回响应数据，视图负责格式化数据并把它们呈现给用户，业务逻辑和数据表示分离，同一个模型可以被不同的视图重用，所以大大提高了模型层的程序代码的可重用性。

其次，模型是自包含的，与控制器和视图保持相对独立，因此可以方便地改变应用程序的业务数据和业务规则。如果把数据库从 MySQL 移植到 Oracle，或者把 RDBMS 数据源改变成 LDAP 数据源，只需改变模型即可。一旦正确地实现了模型，不管业务数据来自数据库还是 LDAP 服务器，视图都会正确地显示它们。由于 MVC 的 3 个模块相互独立，改变其中一个不会影响其他两个，所以依据这种设计思想能构造良好的松耦合的组件。

此外，控制器提高了应用程序的灵活性和可配置性。控制器可以用来连接不同的模型和视图去完成用户的需求，控制器为构造应用程序提供了强有力的重组手段。给定一些可重用的模型和视图，控制器可以根据用户的需求选择适当的模型进行业务逻辑处理，然后选择适当的视图将处理结果显示给用户。

6. MVC 的适用范围

使用 MVC 需要精心的设计，由于它的内部原理比较复杂，所以需要花费一些时间去理解它。将 MVC 运用到应用程序中，会带来额外的工作量，增加应用的复杂性，所以 MVC 不适合小型应用程序。

但对于开发存在大量用户界面，并且业务逻辑复杂的大型应用程序，MVC 将会使软件在健壮性、代码重用和结构方面上一个新的台阶。尽管在最初构建 MVC 框架时会花费一定的工作量，但从长远角度看，它会大大提高后期软件开发的效率。

13.2 store 应用简介

本章介绍的 Java 应用实现了一个商店的客户管理系统，本书把此应用简称为 store 应用。store 应用包含以下用例（Use Case）：

- I 创建新客户
- I 删除客户
- I 更新客户的信息
- I 根据客户 ID 查询特定客户的详细信息
- I 列出所有客户的清单

store 应用使用 MySQL 数据库服务器，它的永久业务数据都存放在 STOREDB 数据库，其中 CUSTOMERS 表用来存放客户信息，它的定义如下：

```
create table CUSTOMERS (  
    ID bigint not null auto_increment primary key,  
    NAME varchar(16) not null,  
    AGE INT,  
    ADDRESS varchar(255)  
);
```

STOREDB 数据库的创建过程可参见本书第 12 章的 12.2 节（安装和配置 MySQL 数据库）。如图 13-4 所示是 store 应用的类框图。其中 ConnectionPool 接口、ConnectionPoolImpl2 类、ConnectionProvider 类和 PropertyReader 类都来自于本书第 12 章。ConnectionPool 接口表示连接池，负责为模型提供数据库连接。StoreException 类

是异常类，如例程 13-1 所示是它的源程序：

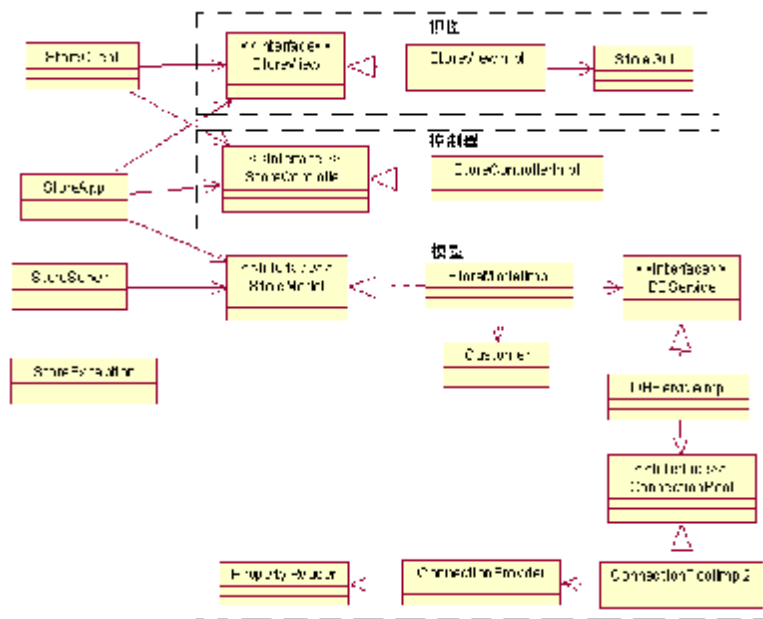


图 13-4 store 应用的类框图

例程 13-1 StoreException.java

```
package store;
public class StoreException extends Exception{
    public StoreException() {
        this("StoreException");
    }
    public StoreException(String msg) {
        super(msg);
    }
}
```

当模型层处理业务逻辑时出现错误，就会抛出 StoreException，例如：

```
public void deleteCustomer(Customer cust) throws StoreException, RemoteException{
    try{
        if(!idExists(cust.getId())){
            throw new StoreException("Customer "+cust.getId()+" not found");
        }
        String sql="delete from CUSTOMERS where ID="+cust.getId();
        dbService.modifyTable(sql);
        fireModelChangeEvent(cust);
    }catch(Exception e){
        e.printStackTrace();
        throw new StoreException("StoreDbImpl.deleteCustomer\n"+e);
    }
}
```

Customer 类与数据库中的 CUSTOMERS 表对应，它表示 store 应用的业务数据。模型层负责把 Customer 对象保存到数据库中，以及从数据库中加载特定的 Customer

对象。视图层则负责在图形界面上展示 Customer 对象的信息，以及接收用户输入的 Customer 对象的信息。如例程 13-2 所示是 Customer 类的源程序。

例程 13-2 Customer.java

```
package store;
import java.io.*;
public class Customer implements Serializable {
    private long id;
    private String name="";
    private String addr="";
    private int age;
    public Customer(long id,String name,String addr,int age) {
        this.id=id;
        this.name=name;
        this.addr=addr;
        this.age=age;
    }

    public Customer(long id){
        this.id=id;
    }
    public Long getId(){
        return id;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name=name;
    }
    ...
    public String toString(){
        return "Customer: "+id+" "+name+" "+addr+" "+age;
    }
}
```

在分布式运行环境中，Customer 对象会从服务器端传送到客户端，也会从客户端传送到服务器端。Customer 类实现了 java.io.Serializable 接口，从而保证 Customer 对象可以在网络上传输。

- l store 应用包括 3 个核心接口。
- l StoreView 接口：视图层的接口，负责生成与用户交互的图形界面。
- l StoreController 接口：控制器层的接口，负责调用模型和视图。
- l StoreModel 接口：模型层的接口，负责处理业务逻辑，访问数据库。

如例程 13-3 所示是 StoreView 接口的源程序。它包括以下 3 个方法。

- l addUserGestureListener(StoreController ctrl)方法：在视图中注册处理各种用户动作（比如用户按下【查询客户】按钮）的控制器，参数 ctrl 指定控制器。
- l showDisplay(Object display)方法：在图形界面上显示数据，参数 display 指定

待显示的数据。

- I **handleCustomerChange()**方法：当模型层修改了数据库中某个客户的信息时，同步刷新视图的图形界面。

例程 13-3 StoreView.java

```
package store;
import java.rmi.*;
public interface StoreView extends Remote{
    /** 注册处理用户动作的监听器，即 StoreController 控制器 */
    public void addUserGestureListener(StoreController ctrl)
        throws StoreException,RemoteException;

    /** 在图形界面上显示数据，参数 display 表示待显示的数据 */
    public void showDisplay(Object display)throws StoreException,RemoteException;

    /** 当模型层修改了数据库中某个客户的信息时，同步刷新视图层的图形界面 */
    public void handleCustomerChange(Customer cust)throws StoreException,RemoteException;
}
```

以上 StoreView 接口的 handleCustomerChange()方法由模型调用，在分布式运行环境中，模型位于服务器层，视图位于客户层。为了使模型能回调 StoreView 对象的 handleCustomerChange()方法，特地把 StoreView 接口设计为远程接口，handleCustomerChange()方法是远程方法，声明抛出 RemoteException。本章 13.7 节的图 13-14 展示了模型对视图的回调过程。

如例程 13-4 所示是 StoreController 接口的源程序。用户在视图提供的图形界面上会执行各种操作，比如按下【查询客户】、【添加客户】、【删除客户】和【更新客户】按钮，StoreController 接口中声明了一系列 handleXXX()方法，它们分别响应用户在图形界面做出的某种动作。

例程 13-4 StoreController.java

```
package store;
public interface StoreController {
    /** 处理根据 ID 查询客户的动作 */
    public void handleGetCustomerGesture(long id);
    /** 处理添加客户的动作 */
    public void handleAddCustomerGesture(Customer c);
    /** 处理删除客户的动作 */
    public void handleDeleteCustomerGesture(Customer c);
    /** 处理更新客户的动作 */
    public void handleUpdateCustomerGesture(Customer c);
    /** 处理列出所有客户清单的动作 */
    public void handleGetAllCustomersGesture();
}
```

如例程 13-5 所示是 StoreModel 接口的源程序。StoreModel 接口中声明了操纵数据库的一系列方法，这些方法用于添加、更新、删除和查询数据库中的客户信息。此外，StoreModel 接口的 addChangeListener(StoreView sv)方法用于在模型中注册视图，当模型修改了数据库中的客户信息时，就可以回调所有注册过的视图的 handleCustomer-

Change(Customer cust)方法，以便同步刷新所有的视图。

例程 13-5 StoreModel.java

```
package store;
import java.rmi.*;
import java.util.*;
public interface StoreModel extends Remote {
    /** 注册视图，以便当模型修改了数据库中的客户信息时，可以回调视图的刷新界面的方法 */
    public void addChangeListener(StoreView sv) throws StoreException, RemoteException;
    /** 向数据库中添加一个新的客户 */
    public void addCustomer(Customer cust) throws StoreException, RemoteException;
    /** 从数据库中删除一个客户 */
    public void deleteCustomer(Customer cust) throws StoreException, RemoteException;
    /** 更新数据库中的客户 */
    public void updateCustomer(Customer cust) throws StoreException, RemoteException;
    /** 根据参数 id 检索客户 */
    public Customer getCustomer(long id) throws StoreException, RemoteException;
    /** 返回数据库中所有的客户清单 */
    public Set<Customer> getAllCustomers() throws StoreException, RemoteException;
}
```

如图 13-5 所示显示了 store 应用根据用户指定的 ID 查询客户详细信息的时序图。

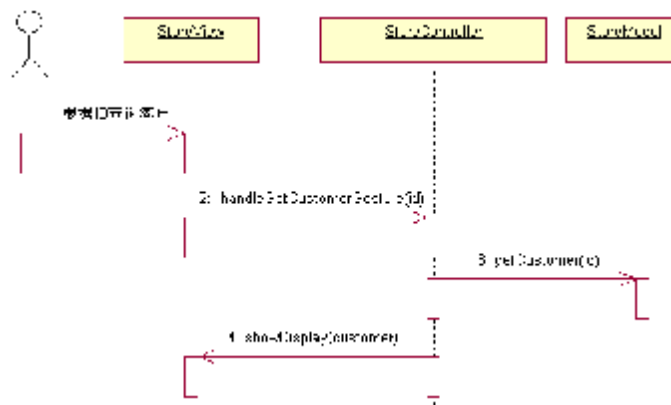


图 13-5 根据用户指定的 ID 查询客户详细信息的时序图

用户在视图的图形界面上输入 ID，然后按下【查询客户】按钮，StoreView 调用 StoreController 的 handleGetCustomerGesture(id)方法处理用户的请求，StoreController 调用 StoreModel 的 getCustomer(id)方法从数据库中获得相应的客户信息。StoreController 接着调用 StoreView 的 showDisplay(customer)方法在图形界面上显示客户信息。

13.3 创建视图

视图包括 StoreView 接口、StoreViewImpl 类和 StoreGui 类。StoreGui 类利用 Swing 组件生成图形用户界面。StoreViewImpl 类实现了 StoreView 接口，StoreViewImpl 类依

赖 StoreGui 类生成图形界面，并且委托 StoreController 来处理 StoreGui 界面上产生的事件。

如图 13-6 和图 13-7 所示是 store 应用的图形用户界面，图 13-6 显示单个客户的详细信息，图 13-7 显示所有客户的清单。



图 13-6 显示单个客户详细信息的图形界面



图 13-7 显示所有客户清单的图形界面

store 应用的图形界面主要包括以下面板。

- l 选择面板 selPan: 位于界面的最顶端，包括两个按钮，【客户详细信息】按钮和【所有客户清单】按钮。【客户详细信息】按钮使界面的中央区域显示 custPan 面板，【所有客户清单】按钮使界面的中央区域显示 allCustPan 面板。
- l 单个客户面板 custPan: 输出或者输入单个客户的详细信息，并且包括 4 个按钮，【查询客户】、【更新客户】、【添加客户】和【删除客户】。
- l 所有客户面板 allCustPan: 用 javax.swing.JTable 组件来显示所有客户的清单。
- l 日志面板 logPan: 显示操作失败时的错误信息。

StoreGui 类负责生成如图 13-6 和图 13-7 所示的图形界面。如例程 13-6 所示是 StoreGui 类的源程序。

例程 13-6 StoreGui.java

```
package store;
//此处省略 import 语句
```



```
...
public class StoreGui {

    //界面的主要窗体组件
    protected JFrame frame;
    protected Container contentPane;
    protected CardLayout card=new CardLayout();
    protected JPanel cardPan=new JPanel();

    //包含各种按钮的选择面板上的组件
    protected JPanel selPan=new JPanel();
    protected JButton custBt=new JButton("客户详细信息");
    protected JButton allCustBt=new JButton("所有客户清单");

    //显示单个客户的面板上的组件
    protected JPanel custPan=new JPanel();
    protected JLabel nameLb=new JLabel("客户姓名");
    protected JLabel idLb=new JLabel("ID");
    protected JLabel addrLb=new JLabel("地址");
    protected JLabel ageLb=new JLabel("年龄");

    protected JTextField nameTf=new JTextField(25);
    protected JTextField idTf=new JTextField(25);
    protected JTextField addrTf=new JTextField(25);
    protected JTextField ageTf=new JTextField(25);
    protected JButton getBt=new JButton("查询客户");
    protected JButton updBt=new JButton("更新客户");
    protected JButton addBt=new JButton("添加客户");
    protected JButton delBt=new JButton("删除客户");

    //列举所有客户的面板上的组件
    protected JPanel allCustPan=new JPanel();
    protected JLabel allCustLb=new JLabel("所有客户清单",SwingConstants.CENTER);
    protected JTextArea allCustTa=new JTextArea();
    protected JScrollPane allCustSp=new JScrollPane(allCustTa);

    String[] tableHeaders={"ID","姓名","地址","年龄"};
    JTable table;
    JScrollPane tablePane;
    DefaultTableModel tableModel;

    //日志面板上的组件
    protected JPanel logPan=new JPanel();
    protected JLabel logLb=new JLabel("操作日志",SwingConstants.CENTER);

    protected JTextArea logTa=new JTextArea(9,50);
    protected JScrollPane logSp=new JScrollPane(logTa);

    /** 显示单个客户面板 custPan */
    public void refreshCustPane(Customer cust){
        showCard("customer");

        if(cust==null || cust.getId()==-1){
            idTf.setText(null);
            nameTf.setText(null);
            addrTf.setText(null);
            ageTf.setText(null);
            return;
        }
        idTf.setText(new Long(cust.getId()).toString());
    }
}
```

```
nameTf.setText(cust.getName().trim());
addrTf.setText(cust.getAddr().trim());
ageTf.setText(new Integer(cust.getAge()).toString());
}

/** 显示所有客户面板 allCustPan */
public void refreshAllCustPan(Set<Customer> custs){
    showCard("allcustomers");
    String newData[][];
    newData=new String[custs.size()][4];
    Iterator<Customer> it=custs.iterator();
    int i=0;
    while(it.hasNext()){
        Customer cust=it.next();
        newData[i][0]=new Long(cust.getId()).toString();
        newData[i][1]=cust.getName();
        newData[i][2]=cust.getAddr();
        newData[i][3]=new Integer(cust.getAge()).toString();
        i++;
    }
    tableModel.setDataVector(newData,tableHeaders);
}

/** 在日志面板 logPan 中添加日志信息 */
public void updateLog(String msg){
    logTa.append(msg+"\n");
}

/** 获得客户面板 custPan 上用户输入的 ID */
public long getCustIdOnCustPan(){
    try{
        return Long.parseLong(idTf.getText().trim());
    }catch(Exception e){
        updateLog(e.getMessage());
        return -1;
    }
}

/** 获得单个客户面板 custPan 上用户输入的客户信息 */
public Customer getCustomerOnCustPan(){
    try{
        return new Customer(Long.parseLong(idTf.getText().trim()),
            nameTf.getText().trim(),addrTf.getText().trim(),
            Integer.parseInt(ageTf.getText().trim()));
    }catch(Exception e){
        updateLog(e.getMessage());
        return null;
    }
}

/** 显示单个客户面板 custPan 或者所有客户面板 allCustPan */
private void showCard(String cardStr){
    card.show(cardPan,cardStr);
}

/** 构造方法 */
public StoreGui(){
    buildDisplay();
}
```

```

/** 创建图形界面 */
private void buildDisplay(){
    frame=new JFrame("商店的客户管理系统");
    buildSelectionPanel();
    buildCustPanel();
    buildAllCustPanel();
    buildLogPanel();

    /** carPan 采用 CardLayout 布局管理器，包括 custPan 和 allCustPan 两张卡片 */
    cardPan.setLayout(card);
    cardPan.add(custPan,"customer");
    cardPan.add(allCustPan,"allcustomers");

    //向主窗体中加入各种面板
    contentPane=frame.getContentPane();
    contentPane.setLayout(new BorderLayout());
    contentPane.add(cardPan,BorderLayout.CENTER);
    contentPane.add(selPan,BorderLayout.NORTH);
    contentPane.add(logPan,BorderLayout.SOUTH);

    frame.pack();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}

/** 创建选择面板 selPan */
private void buildSelectionPanel(){...}

/** 为选择面板 selPan 中的两个按钮注册监听器 */
public void addSelectionPanelListeners(ActionListener a[]){
    int len=a.length;
    if(len!=2){ return;}

    custBt.addActionListener(a[0]);
    allCustBt.addActionListener(a[1]);
}

/** 创建单个客户 custPan 面板 */
private void buildCustPanel(){ ...}

/** 为单个客户面板 custPan 中的 4 个按钮注册监听器 */
public void addCustPanelListeners(ActionListener a[]){
    int len=a.length;
    if(len!=4){ return;}

    getBt.addActionListener(a[0]);
    addBt.addActionListener(a[1]);
    delBt.addActionListener(a[2]);
    updBt.addActionListener(a[3]);
}

/** 创建所有客户 allCustPan 面板 */
private void buildAllCustPanel(){
    allCustPan.setLayout(new BorderLayout());
    allCustPan.add(allCustLb,BorderLayout.NORTH);
    allCustTa.setText("all customer display");

    tableModel=new DefaultTableModel(tableHeaders,10);
    table=new JTable(tableModel);
    tablePane=new JScrollPane(table);
}

```

```
allCustPan.add(tablePane, BorderLayout.CENTER);

Dimension dim=new Dimension(500,150);
table.setPreferredScrollableViewportSize(dim);
}

/** 创建日志面板*/
private void buildLogPanel(){...}
}
```

StoreGui 类中的 public 类型的方法可分为 3 类。

(1) 让图形界面展示数据的方法

- l refreshCustPane(Customer cust): 在单个客户面板 custPan 上显示参数 cust 指定的特定客户的信息。
- l refreshAllCustPan(Set<Customer> custs): 在所有客户面板 allCustPan 上显示参数 custs 指定的所有客户的信息。
- l public void updateLog(String msg): 在日志面板上显示参数 msg 指定的日志信息。

(2) 从图形界面上读取数据的方法

- l getCustIdOnCustPan(): 读取单个客户面板 custPan 上用户输入的 ID。
- l getCustomerOnCustPan(): 读取单个客户面板 custPan 上用户输入的客户信息。

(3) 为图形界面上的按钮注册监听器的方法

- l addSelectionPanelListeners(ActionListener a[]): 为选择面板 selPan 中的两个按钮注册监听器。
- l addCustPanelListeners(ActionListener a[]): 为单个客户面板 custPan 中的 4 个按钮注册监听器。

StoreViewImpl 类实现了 StoreView 接口。一个 StoreViewImpl 对象与一个 StoreModel 对象、一个 StoreGui 对象，以及若干 StoreController 对象关联。如例程 13-7 所示是 StoreViewImpl 类的源程序。

例程 13-7 StoreViewImpl.java

```
package store;
//此处省略 import 语句
...
public class StoreViewImpl extends UnicastRemoteObject
    implements StoreView,Serializable{
    private transient StoreGui gui;
    private StoreModel storemodel;
    private Object display;

    private ArrayList<StoreController> storeControllers=
        new ArrayList<StoreController>(10);

    public StoreViewImpl(StoreModel model)throws RemoteException {
        try{
            storemodel=model;
            model.addChangeListener(this); //向 model 注册自身
        }catch(Exception e){
```

```
        System.out.println("StoreViewImpl constructor "+e);
    }

    gui=new StoreGui();
    //向图形界面注册监听器
    gui.addSelectionPanelListeners(selectionPanelListeners);
    gui.addCustPanelListeners(custPanelListeners);
}

/** 注册控制器*/
public void addUserGestureListener(StoreController b)
    throws StoreException,RemoteException{
    storeControllers.add(b);
}

/** 在图形界面上展示参数 display 指定的数据 */
public void showDisplay(Object display) throws StoreException,RemoteException{
    if(!(display instanceof Exception))this.display=display;

    if(display instanceof Customer){
        gui.refreshCustPane((Customer)display);
    }
    if(display instanceof Set){
        gui.refreshAllCustPan((Set<Customer>)display);
    }
    if(display instanceof Exception){
        gui.updateLog(((Exception)display).getMessage());
    }
}

/** 刷新新界面上的客户信息*/
public void handleCustomerChange(Customer cust)throws StoreException,RemoteException{
    long cIdOnPan=-1;

    try{
        if(display instanceof Set){
            gui.refreshAllCustPan(storemodel.getAllCustomers());
            return;
        }
        if(display instanceof Customer){
            cIdOnPan=gui.getCustIdOnCustPan();
            if(cIdOnPan!=cust.getId())return;

            gui.refreshCustPane(cust);
        }
    }catch(Exception e){
        System.out.println("StoreViewImpl processCustomer "+e);
    }
}

/** 监听图形界面上【查询客户】按钮的 ActionEvent 的监听器 */
transient ActionListener custGetHandler=new ActionListener(){
    public void actionPerformed(ActionEvent e){
        StoreController sc;
        long custId;
        custId=gui.getCustIdOnCustPan();

        for(int i=0;i<storeControllers.size();i++){
            sc=storeControllers.get(i);
            sc.handleGetCustomerGesture(custId);
        }
    }
}
```

```
    }  
};  
  
/** 监听图形界面上【添加客户】按钮的 ActionEvent 的监听器 */  
transient ActionListener custAddHandler=new ActionListener(){...};  
  
/** 监听图形界面上【删除客户】按钮的 ActionEvent 的监听器 */  
transient ActionListener custDeleteHandler=new ActionListener(){...};  
  
/** 监听图形界面上【更新客户】按钮的 ActionEvent 的监听器 */  
transient ActionListener custUpdateHandler=new ActionListener(){...};  
  
/** 监听图形界面上【客户详细信息】按钮的 ActionEvent 的监听器 */  
transient ActionListener custDetailsPageHandler=new ActionListener(){  
    public void actionPerformed(ActionEvent e){  
        StoreController sc;  
        long custId;  
        custId=gui.getCustIdOnCustPan();  
        if(custId!=-1){  
            try{  
                showDisplay(new Customer(-1));  
            }catch(Exception ex){ex.printStackTrace();}  
        }else{  
            for(int i=0;i<storeControllers.size();i++){  
                sc=storeControllers.get(i);  
                sc.handleGetCustomerGesture(custId);  
            }  
        }  
    }  
};  
  
/** 监听图形界面上【所有客户清单】按钮的 ActionEvent 的监听器 */  
transient ActionListener allCustsPageHandler=new ActionListener(){...};  
  
/** 负责监听单个客户面板 custPan 上的所有按钮的 ActionEvent 事件的监听器 */  
transient ActionListener custPanelListeners[]={custGetHandler,custAddHandler,  
    custDeleteHandler,custUpdateHandler};  
  
/** 负责监听选择面板 selPan 上的所有按钮的 ActionEvent 事件的监听器 */  
transient ActionListener selectionPanelListeners[]={  
    custDetailsPageHandler,allCustsPageHandler};  
}
```

在 StoreViewImpl 类中定义了 6 个 ActionListener 监听器，它们分别监听图形界面上的 6 个按钮发出的 ActionEvent 事件。例如，以下 custGetHandler 是【查询客户】按钮发出的 ActionEvent 事件的监听器：

```
transient ActionListener custGetHandler=new ActionListener(){  
    public void actionPerformed(ActionEvent e){  
        StoreController sc;  
        long custId;  
        custId=gui.getCustIdOnCustPan();  
  
        for(int i=0;i<storeControllers.size();i++){  
            sc=storeControllers.get(i);  
            sc.handleGetCustomerGesture(custId);  
        }  
    }  
}
```

```
};
```

在以上 `actionPerformed()` 方法中，先从界面中读取用户输入的 ID，然后调用 `StoreController` 的 `handleGetCustomerGesture()` 方法进行处理。由此可见，视图本身并不处理具体业务逻辑，仅负责输入和输出数据，用户的请求则由控制器来处理。从本章 13.4 节（创建控制器）的控制器实现中可以看出，控制器实际上也不处理业务逻辑，而是调用模型来处理。

13.4 创建控制器

`StoreControllerImpl` 类实现了 `StoreController` 接口。每个 `StoreControllerImpl` 对象与一个 `StoreModel` 对象和一个 `StoreView` 对象关联。如例程 13-8 所示是 `StoreControllerImpl` 类的源程序。

例程 13-8 `StoreControllerImpl.java`

```
package store;
import java.util.*;
public class StoreControllerImpl implements StoreController{
    private StoreModel storeModel;
    private StoreView storeView;
    public StoreControllerImpl(StoreModel model, StoreView view ) {
        try{
            storeModel=model;
            storeView=view;
            view.addUserGestureListener(this);           //向视图注册控制器自身
        }catch(Exception e){
            reportException(e);
        }
    }

    /** 报告异常信息 */
    private void reportException(Object o){
        try{
            storeView.showDisplay(o);
        }catch(Exception e){
            System.out.println("StoreControllerImpl reportException"+e);
        }
    }

    /** 处理根据 ID 查询客户的动作 */
    public void handleGetCustomerGesture(long id){
        Customer cust=null;
        try{
            cust=storeModel.getCustomer(id);
            storeView.showDisplay(cust);
        }catch(Exception e){
            reportException(e);
            cust=new Customer(id);
            try{
                storeView.showDisplay(cust);
            }catch(Exception ex){
                reportException(ex);
            }
        }
    }
}
```



```
    }  
}  
  
/** 处理添加客户的动作 */  
public void handleAddCustomerGesture(Customer c){  
    try{  
        storeModel.addCustomer(c);  
    }catch(Exception e){  
        reportException(e);  
    }  
}  
  
/** 处理删除客户的动作 */  
public void handleDeleteCustomerGesture(Customer c){...}  
  
/** 处理更新客户的动作 */  
public void handleUpdateCustomerGesture(Customer c){...}  
  
/** 处理列出所有客户清单的动作 */  
public void handleGetAllCustomersGesture(){...}  
}
```

StoreControllerImpl 类的 handleGetCustomerGesture(long id)方法处理用户在界面上按下【查询客户】按钮的事件，该方法先调用 StoreModel 对象的 getCustomer(id)方法获得相应的客户信息，然后调用 StoreView 对象的 showDisplay(cust)方法显示客户信息：

```
try{  
    cust=storeModel.getCustomer(id);  
    storeView.showDisplay(cust);  
}catch(Exception e){  
    reportException(e);  
    ...  
}  
//调用模型去处理业务逻辑  
//调用视图去显示数据
```

由此可见，控制器是视图与模型之间的调度者，控制器调用模型去处理业务逻辑，并且调用视图去显示数据。

StoreControllerImpl 类会捕获模型抛出的各种异常，然后由 reportException()方法在图形界面上向用户报告异常：

```
private void reportException(Object o){  
    try{  
        storeView.showDisplay(o);  
    }catch(Exception e){  
        System.out.println("StoreControllerImpl reportException"+e);  
    }  
}  
//调用视图去显示异常
```

StoreViewImpl 类的 showDisplay()方法不仅能显示客户信息，还能显示异常信息。异常信息在 StoreGui 的日志面板 logPan 中显示。

13.5 创建模型

StoreModelImpl 类实现了 StoreModel 接口。StoreModelImpl 类需要通过 JDBC API 访问数据库。本范例创建了一个 DBService 接口，它对 JDBC API 做了轻量级的封装，

主要是封装了 Connection 接口，如图 13-8 所示。

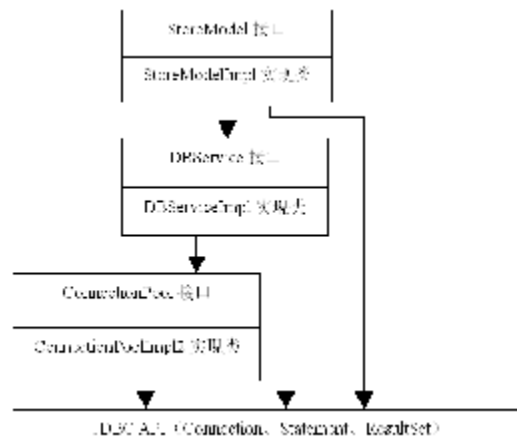


图 13-8 DBService 接口对 JDBC API 做了轻量级封装

如例程 13-9 所示是 DBService 接口的源程序。

例程 13-9 DBService.java

```

package store;
import java.sql.*;
public interface DBService {
    /** 获得 Statement 对象 */
    public Statement getStatement() throws Exception;
    /** 关闭 Statement 对象，以及与之关联的 Connection 对象 */
    public void closeStatement(Statement stmt);
    /** 执行 SQL update、delete 和 insert 语句 */
    public void modifyTable(String sql) throws Exception;
}
    
```

StoreModelImpl 类通过 DBService 接口来访问数据库。如果要执行 SQL update、delete 和 insert 语句，只需调用 DBService 接口的 modifyTable(String sql)方法。如果要执行 SQL select 语句，需要调用 DBService 接口的 getStatement()方法得到一个 Statement 对象，然后通过这个 Statement 对象执行 select 语句。当 StoreModelImpl 类使用完包含查询结果的 ResultSet 对象后，应该调用 DBService 接口的 closeStatement()方法关闭 Statement 对象，以及与之关联的 Connection 对象。由此可见，StoreModelImpl 类只会访问 JDBC API 中的 Statement 和 ResultSet 接口，而不会访问 Connection 接口。所以说，DBService 接口对 JDBC API 做了轻量级的封装。

DBServiceImpl 类实现了 DBService 接口，DBServiceImpl 类使用了本书第 12 章的 12.10.1 节（创建数据库连接池）创建的数据库连接池 ConnectionPool，从该连接池中获得连接。如例程 13-10 所示是 DBServiceImpl 类的源程序。

例程 13-10 DBServiceImpl.java

```

package store;
import java.sql.*;
import java.util.*;
    
```

```
import java.io.*;
public class DBServiceImpl implements DBService{
    private ConnectionPool pool; //连接池

    public DBServiceImpl() throws Exception{
        //ConnectionPoolImpl2 连接池实现提供 Connection 对象的动态代理
        pool=new ConnectionPoolImpl2();
    }
    /** 创建并返回一个 Statement 对象 */
    public Statement getStatement() throws Exception{
        return pool.getConnection().createStatement();
    }

    /** 关闭 Statement 对象，以及与之关联的 Connection 对象*/
    public void closeStatement(Statement stmt){
        try{
        }finally{
            try{
                if(stmt!=null){
                    Connection con=stmt.getConnection();
                    stmt.close();
                    //con 引用 Connection 对象的动态代理对象，它的 close()方法把自身放回连接池
                    con.close();
                }
            }catch(Exception e){e.printStackTrace();}
        }
    }
    /** 执行 SQL update、delete 和 insert 语句 */
    public void modifyTable(String sql) throws Exception{
        Statement stmt=getStatement();
        try {
            stmt.executeUpdate(sql);
        }finally{closeStatement(stmt);}
    }
}
```

一个 StoreModelImpl 对象与一个 DBService 对象和若干 StoreView 对象关联。如例程 13-11 所示是 StoreViewImpl 类的源程序。

例程 13-11 StoreModelImpl.java

```
package store;
import java.util.*;
import java.sql.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class StoreModelImpl extends UnicastRemoteObject implements StoreModel{
    private ArrayList<StoreView> changeListeners=new ArrayList<StoreView>(10);
    private DBService dbService;

    public StoreModelImpl()throws StoreException,RemoteException{
        try{
            dbService=new DBServiceImpl();
        }catch(Exception e){
            throw new StoreException("数据库异常");
        }
    }

    /** 判断数据库中是否存在参数指定的客户 ID */
    protected boolean idExists(long id){
```

```
Statement stmt=null;
try{
    stmt=dbService.getStatement();
    ResultSet result=stmt.executeQuery("select ID from CUSTOMERS where ID="+id);
    return result.next();
}catch(Exception e){
    return false;
}finally{
    dbService.closeStatement(stmt);
}
}

/** 注册视图，以便当模型修改了数据库中的客户信息时，可以回调视图的刷新界面的方法 */
public void addChangeListener(StoreView sv) throws StoreException,RemoteException{
    changeListeners.add(sv);
}

/** 当数据库中客户信息发生变化时，同步刷新所有的视图 */
private void fireModelChangeEvent(Customer cust){
    StoreView v;
    for(int i=0;i<changeListeners.size();i++){
        try{
            v=changeListeners.get(i);
            v.handleCustomerChange(cust);
        }catch(Exception e){
            System.out.println(e.toString());
        }
    }
}

/** 向数据库中添加一个新的客户 */
public void addCustomer(Customer cust) throws StoreException,RemoteException{...}

/** 从数据库中删除一个客户 */
public void deleteCustomer(Customer cust) throws StoreException,RemoteException{...}

/** 更新数据库中的客户 */
public void updateCustomer(Customer cust)throws StoreException,RemoteException{
    try{
        if(!idExists(cust.getId())){
            throw new StoreException("Customer "+cust.getId()+" not found");
        }
        String sql="update CUSTOMERS set "+
            "NAME="+cust.getName()+" "+
            "AGE="+cust.getAge()+" "+
            "ADDRESS="+cust.getAddr()+" "+
            "where ID="+cust.getId()+" ";

        dbService.modifyTable(sql);
        fireModelChangeEvent(cust); //同步刷新所有视图
    }catch(Exception e){
        throw new StoreException("StoreDbImpl.updateCustomer\n"+e);
    }
}

/** 根据参数 id 检索客户 */
public Customer getCustomer(long id)throws StoreException,RemoteException{
    Statement stmt=null;
    try{
        if(!idExists(id)){
            throw new StoreException("Customer "+id+" not found");
        }
    }
}
```

```

    }
    stmt=dbService.getStatement();
    ResultSet rs=stmt.executeQuery("select ID,NAME,ADDRESS,AGE from CUSTOMERS"
        +"where ID="+id);
    rs.next();
    return new Customer(rs.getLong(1),rs.getString(2),rs.getString(3),rs.getInt(4));
} catch (Exception e){
    throw new StoreException("StoreDbImpl.getCustomer\n"+e);
} finally{
    dbService.closeStatement(stmt);
}
}

/** 返回数据库中所有的客户清单 */
public Set<Customer> getAllCustomers() throws StoreException,RemoteException{...}
}

```

在分布式运行环境中，一个服务器端的 `StoreModelImpl` 对象会被多个客户端的视图共享。`StoreModelImpl` 对象的 `updateCustomer()`、`deleteCustomer()`和 `addCustomer()` 方法在更新了数据库中的客户信息后，都会调用 `fireModelChangeEvent()`方法，该方法会同步刷新与 `StoreModelImpl` 对象关联的所有视图，如果这些视图正在展示的数据刚好和被更新的客户信息有关，那么这些视图就会重新显示最新的客户信息。

13.6 创建独立应用

`StoreApp` 类表示一个独立的应用程序，它的 `main()` 方法依次创建了 `StoreModelImpl`、`StoreViewImpl` 和 `StoreControllerImpl` 对象，这些对象都位于同一个 Java 虚拟机中。如图 13-9 所示显示了这 3 个对象之间的关联关系。

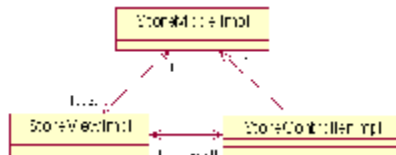


图 13-9 模型、视图和控制器对象之间的关联关系

如例程 13-12 所示是 `StoreApp` 类的源程序。

例程 13-12 `StoreApp.java`

```

package store;
public class StoreApp {
    public static void main(String args[])throws Exception{
        StoreModel model=new StoreModelImpl();
        StoreView view=new StoreViewImpl(model);
        StoreController ctrl=new StoreControllerImpl(model,view);
    }
}

```

store 应用的目录结构如图 13-10 所示。

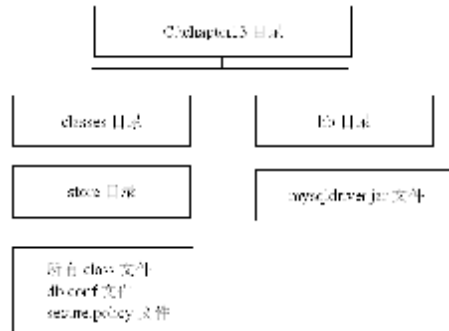


图 13-10 store 应用的目录结构

下面按如下步骤运行 store 应用。

(1) 在 MySQL 中创建 dbuser 用户，口令为 1234，创建 STOREDB 数据库和 CUSTOMERS 表，参见本书第 12 章的 12.2 节（安装和配置 MySQL 数据库）。

(2) 在 DOS 控制台设置 classpath。命令如下：

```
set classpath=C:\chapter13\lib\mysql.driver.jar;C:\chapter13\classes
```

(3) 运行命令“java store.StoreApp”，就会出现本章 13.3 节的图 13-6 所示的图形界面。

如图 13-11 所示显示了执行 StoreApp 类的 main()方法的时序图。

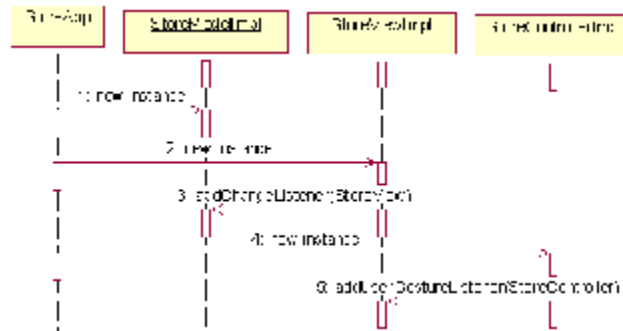


图 13-11 执行 StoreApp 类的 main()方法的时序图

13.7 创建分布式应用

在分布式运行环境中，可以把模型分布在服务器端，视图和控制器分布在客户端。模型可以为多个客户端的视图提供服务。

如例程 13-13 所示 StoreServer 类是 store 应用的服务器程序，它向 RMI Registry 注册表注册了一个 StoreModel 远程对象。

例程 13-13 StoreServer.java

```
package store;
import java.rmi.*;
import javax.naming.*;
```

```
public class StoreServer {
    public static void main( String args[] ){
        try{
            System.setProperty("java.security.policy",
                StoreClient.class.getResource("secure.policy").toString());
            System.setSecurityManager(new RMISecurityManager());

            StoreModel storeModel=new StoreModelImpl();
            Context namingContext=new InitialContext();
            namingContext.rebind( "rmi:storeModel", storeModel );
            System.out.println( "服务器注册了 StoreModel 对象" );
        }catch( Exception e ){
            e.printStackTrace();
        }
    }
}
```

如例程 13-14 所示 StoreClient 类是 store 应用的客户程序，它与本章 13.6 节的例程 13-12 的 StoreApp 类很相似，区别在于 StoreClient 类从服务器上获得 StoreModel 对象的远程引用，而不是像 StoreApp 类那样在本地创建 StoreModel 对象。

例程 13-14 StoreClient.java

```
package store;
import javax.naming.*;
import java.rmi.*;

public class StoreClient {
    public static void main( String args[] ){
        System.setProperty("java.security.policy",
            StoreClient.class.getResource("secure.policy").toString());
        System.setSecurityManager(new RMISecurityManager());

        String url="rmi://localhost/";
        try{
            StoreModel model;
            StoreView view;
            StoreController ctrl;

            Context namingContext=new InitialContext();
            //获得 StoreModel 远程对象的远程引用
            model=(StoreModel)namingContext.lookup(url+"storeModel");
            view=new StoreViewImpl(model);
            ctrl=new StoreControllerImpl(model,view);
        }catch( Exception e ){
            e.printStackTrace();
        }
    }
}
```

store 应用的目录结构参见本章 13.6 节的图 13-10。下面按如下步骤运行分布式的 store 应用。

(1) 在 MySQL 中创建 dbuser 用户，口令为 1234。创建 STOREDB 数据库和

CUSTOMERS 表，参见本书第 12 章的 12.2 节（安装和配置 MySQL 数据库）。

(2) 在 DOS 控制台设置 classpath。命令如下：

```
set classpath=C:\chapter13\lib\mysql.driver.jar;C:\chapter13\classes
```

(3) 启动 RMI Registry，运行命令 “start rmiregistry”。

(4) 启动 StoreServer 服务器，命令如下：

```
start java -Djava.rmi.server.codebase=file:///C:\chapter13\classes\ store.StoreServer
```

(5) 运行 StoreClient 客户程序，运行命令 “start java store.StoreClient”。

(6) 再运行一个 StoreClient 客户程序，运行命令 “start java store.StoreClient”。

以上操作启动了一个 StoreServer 服务器进程和两个 StoreClient 进程，如图 13-12 所示。两个 StoreClient 进程都会访问同一个 StoreServer 服务器进程中的模型。

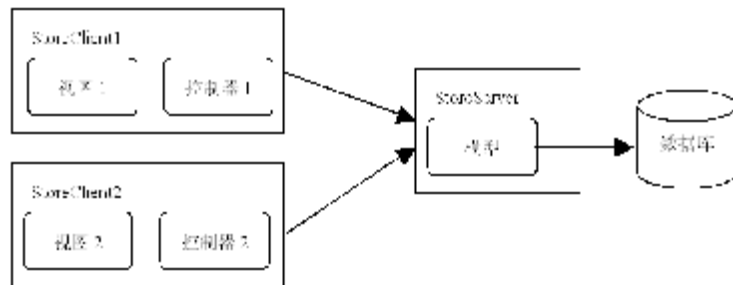


图 13-12 两个 StoreClient 进程访问同一个 StoreServer 服务器进程中的模型

如图 13-13 所示，在两个 StoreClient 的界面上都查询 ID 为 1 的客户的信息。接下来在 StoreClient1 界面上把客户的地址由原来的“北京”改为“上海”，然后按下【更新客户】按钮，你会发现 StoreClient2 界面上的客户地址也会被自动刷新，显示“上海”。



图 13-13 模型同步刷新所有客户端的视图

当 StoreClient1 调用模型的 updateCustomer() 方法修改客户信息后，模型会同步刷新所有客户端的视图，使它们显示最新的客户信息。如图 13-14 所示显示了用户在 StoreClient1 的界面上修改客户信息的时序图。图中 StoreView1 和 StoreController1 是

StoreClient1 进程中的视图和控制器，StoreView2 是 StoreClient2 进程中的视图。

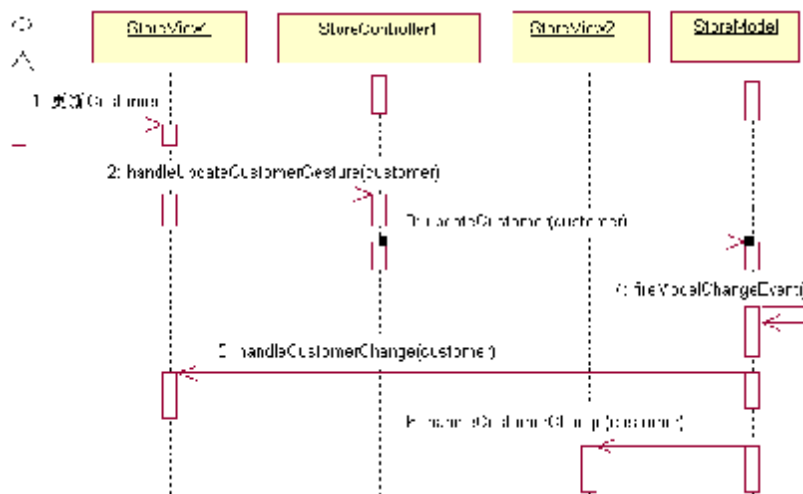


图 13-14 用户在 StoreClient1 的界面上修改客户信息的时序图

13.8 小结

应用软件一般都包含界面、业务逻辑和业务数据。MVC 设计模式把软件应用分为视图、控制器和模型 3 个模块，或者说 3 个层次。视图负责创建界面，并且在界面上展示数据，此外还能接收用户输入的数据。模型负责处理业务逻辑，模型一般会访问数据库，向数据库中查询、添加、更新或删除业务数据。控制器是视图与模型之间的调度枢纽，它根据用户的请求，调用模型去执行业务逻辑，并且调用视图去展示模型返回的响应结果。

本章的 store 应用为视图层、控制器层和模型层分别抽象出了 StoreView、StoreController 和 StoreModel 接口，层与层之间通过接口来交互，提高了各个层的独立性，并且削弱了层与层之间的耦合。由于视图、控制器和模型是各自独立的，因此可以方便地把它们分布到网络中的不同机器上。通常，模型是重用性最高的模块，它作为远程对象分布在服务器上，为多个客户端的视图提供服务。

13.9 练习题

- 以下哪些属于视图的任务？（多选）
 - 展示数据
 - 选择视图显示响应结果
 - 处理业务逻辑
 - 通知视图业务数据更新
 - 接收用户的输入数据
 - 触发事件
 - 调用模型响应用户请求
- 以下哪些属于控制器的任务？（多选）

- A. 展示数据
 - B. 选择视图显示响应结果
 - C. 处理业务逻辑
 - D. 通知视图业务数据更新
 - E. 接收用户的输入数据
 - F. 触发事件
 - G. 调用模型响应用户请求
3. 以下哪些属于模型的任务？（多选）
- A. 展示数据
 - B. 选择视图显示响应结果
 - C. 处理业务逻辑
 - D. 通知视图业务数据更新
 - E. 接收用户的输入数据
 - F. 触发事件
 - G. 调用模型响应用户请求
4. MVC 设计模式有哪些优点？（多选）
- A. 提高程序代码的可重用性
 - B. 提高应用程序的灵活性和可配置性
 - C. 软件规模越小，MVC 设计模式越能缩短软件的开发周期
 - D. 提高程序代码的可维护性
5. 对于本章介绍的 store 应用，在运用 RMI 框架时，控制器层位于客户端还是服务器端？
- A. 客户端
 - B. 服务器端
6. 在 MVC 设计模式中，哪个模块的可重用性最高？（单选）
- A. 视图
 - B. 控制器
 - C. 模型
7. 参照本章 13.2 节的图 13-5（根据用户指定的 ID 查询客户详细信息的时序图），绘制出用户修改一个客户信息时的时序图。
8. 参考本章的 store 应用，创建一个聊天系统。如图 13-15 所示是聊天系统的界面。



图 13-15 聊天系统的界面

一个视图表示一个聊天用户的界面。模型被多个视图共享，所有视图都向模型注册自身，即模型持有所有视图的远程引用。模型负责消息的转发，此外，当一个新的用户登录到聊天系统中或从该系统中退出时，模型会通知所有视图刷新用户名单。模型位于服务器端，视图与控制器位于客户端。

如图 13-16 所示为用户 Client1 给用户 Client2 发送一条消息的时序图。

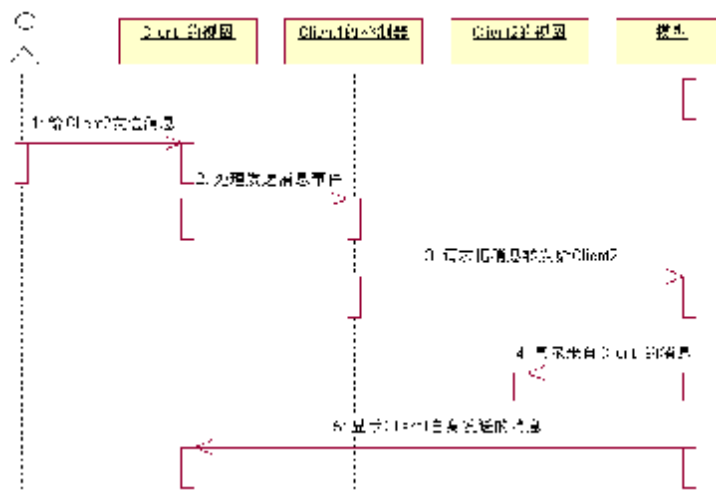


图 13-16 用户 Client1 给用户 Client2 发送一条消息的时序图

模型层的接口 ChatModel 的定义如下：

```

import java.rmi.*;
public interface ChatModel extends Remote{
    /** 登录一个聊天用户，为了简化程序，可以由该方法为用户分配一个临时的唯一的用户名，
     * 如第一个登录的用户为“client0”，第二个登录的用户为“client1”，依次类推。
     */
    public void registerClient(ChatView client)throws RemoteException;
    /** 退出一个聊天用户*/
    public void unregisterClient(ChatView client)throws RemoteException;
    /** 转发消息，参数 sendFrom 表示发送者的用户名，参数 sendTo 表示接收者的用户名 */
    public void transferMsg(String sendFrom,String sendTo,String msg) throws RemoteException;
}
    
```

以下是 ChatModel 接口的 registerClient()方法的实现：

```

public void registerClient(ChatView client)throws RemoteException{
    clients.add(client); //clients 为模型中存放所有视图的缓存
    client.setName("client"+num++); //为视图分配一个用户名，变量 num 为模型的一个实例变量
    refreshChaters(); //通知所有视图刷新用户名单
}
    
```

答案：1. AEF 2. BG 3. CD 4. ABD 5. A
6. C