Java 语言的 MVC 架构客户管理系统

MVC 是 Model-View-Controller 的简称，即模型-视图-控制器。MVC 是 Xerox PARC 在 20 世纪 80 年代为编程语言 Smalltalk-80 发明的一种软件设计模式，至今已被广泛使用。

本应用首先介绍 MVC 设计模式的概念，然后创建一个基于 MVC 的 Java 应用。

1 MVC 设计模式简介

MVC 把应用程序分成 3 个核心模块：模型（Model）、视图（View）和控制器（Controller），它们分别担当不同的任务。如图 1 所示显示了这几个模块各自的功能及它们的相互关系。

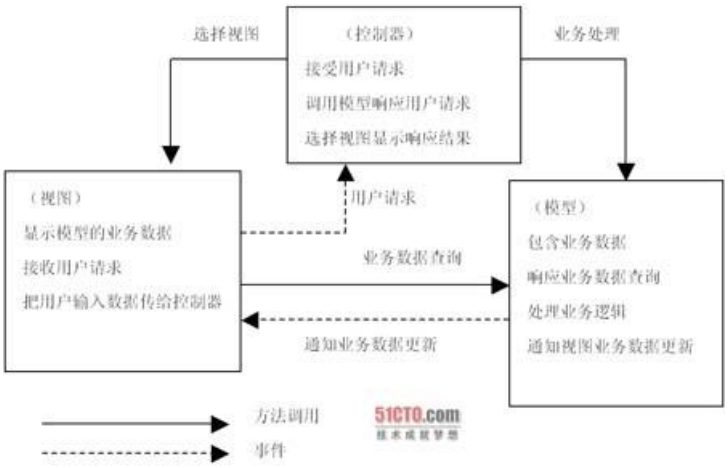

图 1　MVC 设计模式

1．视图

视图是用户看到并与之交互的界面。视图向用户展示用户感兴趣的业务数据，并能接收用户的输入数据，但是视图并不进行任何实际的业务处理。视图可以向模型查询业务数据，但不能直接改变模型中的业务数据。视图还能接收模型发出的业务数据更新事件，从而对用户界面进行同步更新。

2．模型

模型是应用程序的主体部分。模型表示业务数据和业务逻辑。一个模型能为多个视图提供业务数据。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 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。

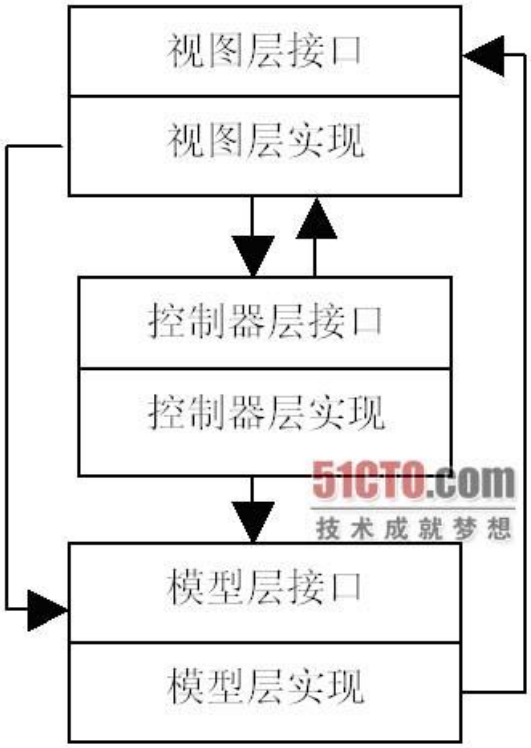

图 2　MVC 的 3 个模块也可以看做软件的 3 个层次

4．MVC 处理过程

如图 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。

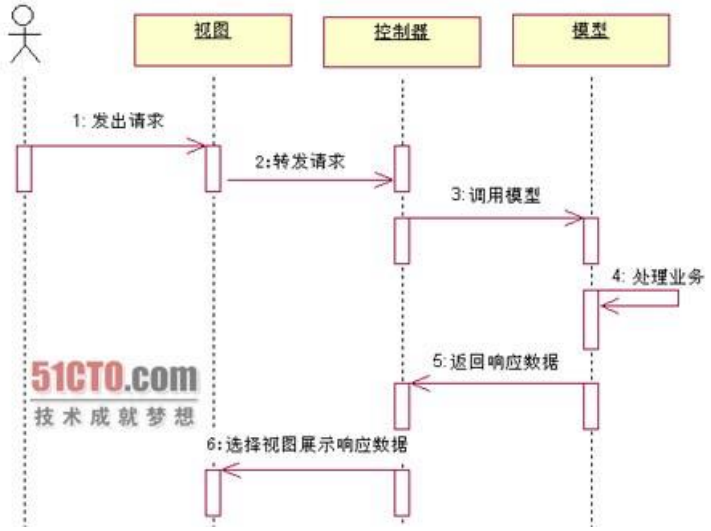

图 3　MVC 的处理过程

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 数据源改变成文件数据源，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文件，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偶合的组件。

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。

2 store 应用简介

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

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

store 应用使用文本文件存储数据，它的永久业务数据都存放在一个指定的文本文件中（data.txt）。

。StoreException 类是异常类，如例程 13-1 所示是它的源程序：

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{
   try{
       if(!idExists(cust.getId())){
           throw new StoreException("Customer +cust.getId()+"
not found");
       }

       }catch(Exception e){
           e.printStackTrace();
           throw new
```

```
StoreException("StoreDbImpl.deleteCustomer\n"+e);
        }
}
```

Customer 类与文件内容对应，它表示 store 应用的业务数据。模型层负责把 Customer 对象保存到文件中，从文件中加载特定的 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 {
   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;
   }
```

```
}
```

◆store 应用包括 3 个核心接口。
◆StoreView 接口：视图层的接口，负责生成与用户交互的图形界面。
◆StoreController 接口：控制器层的接口，负责调用模型和视图。
◆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。

◆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。
◆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。
◆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;

public interface StoreView {
   /** 注册处理用户动作的监听器，即 StoreController 控制器 */
   public void addUserGestureListener(StoreController ctrl)
throws StoreException;

   /** 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;

   /** 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;
}
```

以上 StoreView 接口的 handleCustomerChange()方法由模型调用。

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.util.*;
public interface StoreModel {
   /** 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;
   /** 向数据库中添加一个新的客户 */
   public void addCustomer(Customer cust) throws StoreException;
   /** 从数据库中删除一个客户 */
   public void deleteCustomer(Customer cust) throws
StoreException;
   /** 更新数据库中的客户 */
   public void updateCustomer(Customer cust) throws
StoreException;
   /** 根据参数 id 检索客户 */
   public Customer getCustomer(long id) throws StoreException;
   /** 返回数据库中所有的客户清单 */
```

```
    public Set getAllCustomers() throws StoreException;
}
```

如图 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。

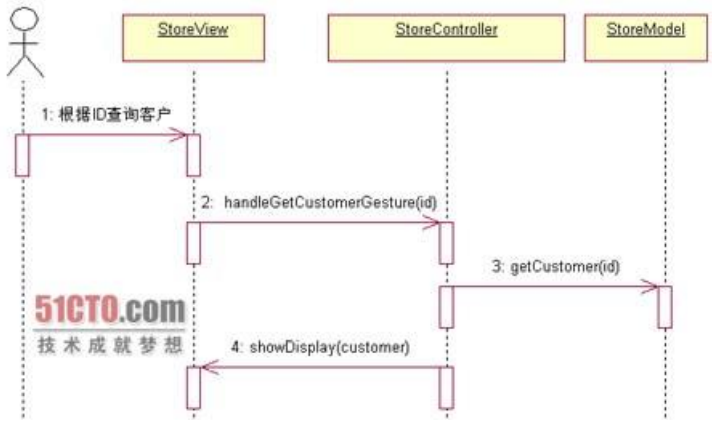

图 5　根据用户指定的 ID 查询客户详细信息的时序图

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。

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

图 6　显示单个客户详细信息的图形界面

图 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。

◆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。
◆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】。
◆所有客户面板 allCustPan：用 javax.swing.JTable 组件来显示所有客户的清单。
◆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 6 和图 7 所示的图形界面。如例程 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 custs){
    showCard("allcustomers");
    String newData[][];
    newData=new String[custs.size()][4];
    Iterator 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

◆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。
◆refreshAllCustPan(Set 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。
◆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

◆getCustIdOnCustPan()：读取单个客户面板 custPan 上用户输入的 ID。
◆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

◆addSelectionPanelListeners(ActionListener a[])：为选择面板 selPan 中的两个按钮注册监听器。
◆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。

#p#

例程 13-7　StoreViewImpl.java

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

   private ArrayList storeControllers=
new ArrayList(10);

   public StoreViewImpl(StoreModel model) {
       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{
       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{
       if(!(display instanceof Exception))this.display=display;

       if(display instanceof Customer){
             gui.refreshCustPane((Customer)display);
       }
       if(display instanceof Set){
             gui.refreshAllCustPan((Set)display);
       }
```

```
        if(display instanceof Exception){
              gui.updateLog(((Exception)display).getMessage());
        }
    }

    /** 刷新界面上的客户信息*/
    public void handleCustomerChange(Customer cust)throws
StoreException{
          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(){…};</storecontrollers.size();i++){

    /** 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);
                   }
               }
         }
   };</storecontrollers.size();i++){

   /** 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);
           }
     }
};</storecontrollers.size();i++){
```

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。从第 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。

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

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);  //调用视图去显示异常
}catch(Exception e){
System.out.println("StoreControllerImpl
reportException"+e);
}
}
```

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。

5 创建模型

StoreModelImpl 类实现了 StoreModel 接口。StoreModelImpl 类需要通过文本文件保存对象内容。

。【实现利用文本文件存储对象内容】

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。

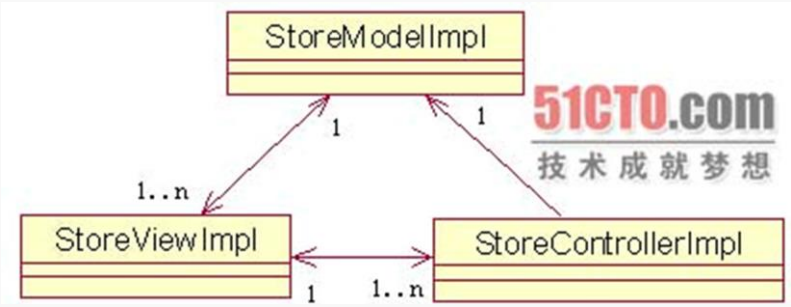

图 9　模型、视图和控制器对象之间的关联关系

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);
}
}
```

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

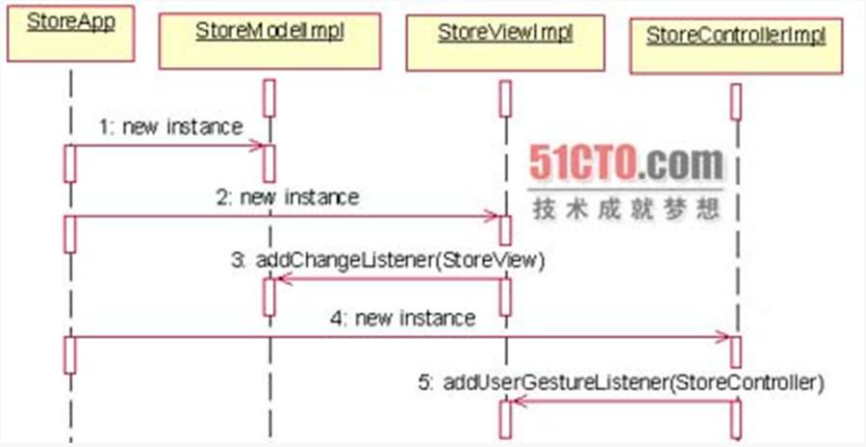

图 11　执行 StoreApp 类的 main()方法的时序图

7 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，模型一般会完成数据持久化（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。

本应用的 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。