遗产代码迁移中的用户界面自动转换技术

石学林　张兆庆
（中国科学院计算技术研究所　中国科学院研究生院　北京 100080）

摘　要　随着 Internet 计算技术的迅速发展，将遗产系统迁移到 Web 平台逐渐成为一种必需。传统的遗产代码通常使用字符终端界面，它与 Web 图形界面在界面显示和用户交互方面都有着显著的不同。本文实现了一种基于停止等待协议的方法可以将这种字符界面自动化地迁移到 Web 图形界面。实验表明，此方法可以显著地增加整个转换过程的自动化程度。

关键词　遗产代码，界面迁移，字符界面，Web 界面，Cobol2Java

Using Stop and Wait Protocol to Migrate Character-based UI to Web

SHI Xue-Lin　ZHANG Zhao-Qing
(Institute of Computing Technology, CAS, Graduate School of CAS, Beijing100080)

Abstract　As the rapid growth of Internet computing, it's critical to migrate Cobol-like legacy system to the distributed web platform. However, legacy systems usually use a character-based user-interface. It's different from web GUI not only in display style but also in interactive model. This paper implements amethod based on a stop and wait protocol to automatically migrate character user-interface to web. Our experience shows that it can increase the automation of this shifting process dramatically.

Keywords　Legacy code, User-interface migration, Character ui, Web ui, Cobol2Java

1　引言

在遗产系统向现代语言或平台的迁移过程中，用户界面的现代化是其中的一个重要方面。许多遗产程序，尤其是精于数据处理的 Cobol 程序通常都采用基于字符表单式的界面，运行在专有的大型机平台上。随着 Internet 技术的广泛普及，使用 Web 技术替换或者集成遗产代码系统向用户提供网络服务，不仅成为企业有力的竞争手段，并且几乎成为一种必需。今天的计算机用户已经不仅仅是那些训练有素的大型机终端用户，对于许多人来说，Web 浏览器已经成为他们使用计算机获取服务的标准界面[1]。然而，原有的遗产代码系统通常工作在大型机-终端方式，多采用 25 行 80 列的传统字符终端界面，由程序逻辑直接在屏幕上精细地在各个坐标点输出字符来构成屏幕布局，并且采用 I/O 库的方式在程序逻辑和用户输入之间轮换执行。如何将这种系统迁移到以事件驱动方式运行的 Web 平台，并且向普通用户提供简单、一致的 Web 图形用户界面，成为必需解决的一个基本问题。

通常这种迁移过程是手工操作的[2]，包括手工重新编写那些界面交互代码，或大量地对代码结构进行重构。研究表明[3]，一个交互式系统中，通常有超过一半的代码都在处理界面交互，因此完全采用手工操作进行界面迁移将是不切实际的。为了减少这种手工操作，系统维护人员需要半自动化或完全自动化的迁移工具来辅助界面迁移过程[4]。

本文对界面转换过程中的界面布局、界面交互、参数传递和单用户与多用户等几个基本问题进行了详细分析，并且实现了以停止等待协议为核心的自动化的界面转换方法。该方法在一个 Cobol2Java 转换器——Cota[10]中得到实现。实例研究表明，采用此方法可以无需人工干预地将遗产系统迁移到 Web 平台，提供给用户与原系统基本一致的 Web 用户界面。

2　相关工作

将遗产系统中的将字符界面迁移到 Web 平台的一个直观方法就是使用程序分析技术从遗产代码系统中分离出用户界面表示逻辑和其他数据处理逻辑，然后用 HTML 语言重新实现界面逻辑，并对其他数据处理逻辑采用包装方法以 CGI 方式加以调用，从而实现用户与程序之间的交互，比如文[5,8,9]。这种方法通常要求被转换的程序本身具有良好的结构，程序设计之初就对程序用户界面逻辑和数据处理与访问逻辑作了明确的划分，也就是说程序本身是基于客户/服务器模式的。不幸的是在遗产系统中，用户界面交互经常和数据访问高度交织在一起[7]，使用文[2]中的程序切片技术对这种交织函数进行分解，可能导致函数分解过于细致，从而导致目标系统的性能严重降低[6]。在这种情况下，手工对源代码进行改写将是不可避免的。

另一种方法就是使用目标平台所提供的 GUI 编程功能模拟字符界面，比如 Java Applet 等。采用这种方法仍然需要找到一种对原遗产系统进行自动分解和部署的方法，决定哪些部分在客户端以 Applet 运行，哪些部分在服务器端运行。如果需要在客户端和服务器端做一个平衡的任务分解，减少客户与服务器之间的交互，那么仍然会面对上文所描述的函数分解困境。如果不对遗产程序进行重构，尽可能最大限度地将原程序逻辑封装在 Applet 中在客户端运行，那么至少需要客户的浏览器系统安装一个巨大的 Java 运行时环境

石学林　博士生，主要研究方向为程序理解技术，编译优化技术。张兆庆　研究员，博士生导师，研究领域为编译优化技术和并行可视化技术。

(JRE),同时从服务器端下载遗产系统和运行时支持系统,比如为了支持 Cobol 语义的 Cobol 运行时系统。这一方面意味着对客户端的软硬件环境以及网络连接方面有较高的要求,另一方面也需要服务端对客户端有充分的信任,允许客户端完全下载遗产系统运行,甚至直接访问服务端的数据。也就是说需要在原遗产系统中增加安全策略模块,可以有效地和遗产系统整合在一起,并且自动地部署。这不可避免地会给自动转换工作带来额外的负担,进而需要手工对旧系统的代码进行适当调整。

3 界面迁移需要考虑的问题

3.1 怎样保持界面布局

界面布局指的是组成界面的各种交互式元素和静态元素之间的相对位置,比如图形用户界面程序中的各窗口、文本框、按钮、提示文本和图标等各种元素在屏幕上的相对位置。遗产系统多采用传统的 25 行 80 列的字符终端界面,比如 Cobol 系统。在这种用户界面系统中,可显示的字符是构成界面的主要元素,系统使用光标在全屏范围内提示用户的输入位置。任何一个字符的位置可以由(行号,列号)构成的一个二维坐标表示。系统提供在任何一个坐标点进行输入和输出的功能,比如下面的 Cobol 全屏操作:

```
display "input a number:" at 1201
accept num at 1217.
```

第一个 display 语句要求用户在 12 行 1 列输入一个数据,第二个 accept 语句则在 12 行 17 列处获得用户输入的数据并赋值给变量 num。在用户输入提示中,常量字符串"input a number:"在屏幕上占据 16 个位置,因此输入语句从 12 行 17 列开始接收数据,也就是说正好排放在输入提示的后面。另外变量 num 是有固定长度的,因此相当于从屏幕上 12 行 17 列的位置开始的地方,接受与 num 等长的字符串并将其转换为数字数据存储于变量 num 中。这是一种灵活的布局方式,其布局内容和位置不但可以在编译前确定,也可以在程序运行期间动态变化。如果不能保持这种布局方式,将导致转换后的界面的凌乱。

3.2 如何转换界面交互方式

大部分具有字符用户界面的单进程程序都是由系统发起的交互[2]。例如在一个 Cobol 程序中,程序从某个入口开始执行,直到碰到 accept 语句。此时程序停止,等待用户输入。当用户输入数据后,程序继续执行。但是在 Web 程序中,通常都是服务端程序先启动,然后由客户端请求传送一个页面,服务端程序根据客户端的请求,或者返回一个预先设定好的静态页面,或者使用动态页面生成技术,比如 php、asp 和 jsp 等脚本动态生成一个页面返回。每一个输入语句意味着需要用户和程序间的一次交互,也就是说需要返回一个页面给客户端输入。如本文第 2 节所述,文献中的两种方法通常都需要将两个输入语句间的代码片断封装在一个独立的动态脚本中,比如一个 jsp 文件,作为第一个输入的应答。但是这两个输入语句可能位于不同的函数中,或者有多个输入语句都在一个函数内部,因此意味着需要对函数进行分解和重组。从而也使这类方法的适用性受到限制,不能自动化地处理用户输入和输出过于复杂的耦合情况。

3.3 界面参数传递

在基于字符界面的遗产程序中,用于用户交互的界面处理逻辑和数据处理逻辑通常位于单一进程之中,编程语言在语句级或库调用一级上直接对用户输入和输出进行支持。比如上文中的"accept num at 1217",当程序运行到此时,程序停止,并且在屏幕 12 行 17 列位置显示一个提示光标等待用户输入。当用户输入一个数据后,该数据自动转换为数字存入变量 num,然后程序继续运行。然而在 Web 界面下,用于和用户交互的界面逻辑显示在远程的浏览器中,数据处理逻辑运行在服务器端。两部分程序运行在不同的进程之中,并且大多数情况是异地的两个机器之间。因此必须将用户输入反馈到服务器端的数据处理逻辑之中。

3.4 单用户与多用户

大多数基于字符界面的遗产程序都是单用户的,用户在本地启动该程序,然后在用户界面的提示下与系统完成交互。我们把用户从启动程序开始,直到程序退出的过程叫做一次会话。一次会话通常由一次或多次用户交互构成,这取决于用户程序处理逻辑和每次交互中输入的数据。在单用户环境下,由多次交互构成的会话完全是在这个用户的控制之下的(通过界面交互)。程序中的数据处理逻辑完全不用担心从用户界面传入的数据是从哪个用户传过来的,因为它只有一个客户,就是启动本次会话的用户。在 Web 环境下,位于服务器端的程序可能同时有多个用户访问,因此不同的会话之间必须保持不会受到干扰。必须找到一种机制使得单用户服务的程序自动转换到多用户程序环境。

4 自动界面转换方法

由前文的分析可知,面向字符界面的遗产程序交互方式和现代 Web 程序的交互方式具有显著的不同。如果程序按照"界面更新——业务逻辑计算——界面更新"这样严格的轮换方式编写,在业务逻辑计算过程中不包含任何界面更新逻辑,那么将两个界面更新逻辑作为动态页面返回将是直截了当的。但是,传统字符界面程序的用户界面逻辑和业务逻辑往往高度耦合,使得这样的分解非常困难,尤其是自动化的分解几乎不可能。本文提出的自动转换方法包括输入输出重定向和停止-等待交互协议两个主要部分,通过输入输出重定向可以保持界面布局,通过停止-等待协议可以转换交互方式。图 1 显示了经过翻译后的新应用程序的结构。

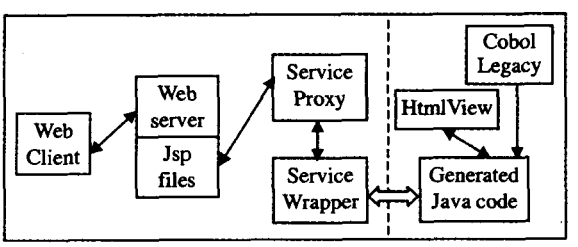

图 1 新应用程序结构框图

整个应用程序包括 5 个组成部分,原始的 Cobol 源码经过 Cota 转换器被翻译成 Java 源代码和 HtmlView 两个部分,其中 HtmlView 是屏幕映像,Cobol 源代码中输入输出语句对界面的修改被重定向到 HtmlView 中,这样在任何时刻都可以调用 HtmlView 的 getHtmlPage 接口动态生成当前屏幕界面,而 Java 源代码部分则对应着原 Cobol 中的业务逻辑,它被封装成服务独立地在后台运行。另外三个部分 jsp 文件、服务代理和服务包装等则封装了 Web 服务器用来与传统系统进行交互的控制逻辑结构。它们主要和 Web 服务器以及后台服务进行交互,同时根据协议决定后台服务以及 jsp 文

件的交替执行。在后台服务的基础上抽象出一个服务代理层，主要是为了增加这种控制结构的适应性。

4.1 封装 I/O 操作到屏幕映像 HtmlView

屏幕映像 HtmlView 将屏幕看成是一个由 25 行 80 列字符构成的字符矩阵，其中每一行由若干字段构成，包括两种字段类型，即字符串字段类型和文本输入框字段类型。每一个字段包括在屏幕上开始的列位置和字段长度，所有字段的长度不超过 80 列。当遇到上文中的输出语句时，可以在第 12 行的字段集合中插入需要显示的字符串，同样当需要处理输入语句时，可以在对应行的字段集合中插入文本框字段，比如上文的"accept num at 1217"可以显示为：

<input name=num size=num.getLength() maxLength=num.getLength()>。

同时，HtmlView 提供一个 getHtmlPage()调用实时地创建整个 HTML 页面，其算法为：

```
for each line in HtmlViewer{
 for each segment in the line{
   if segment is a display string then
      just copy this string to the page
   else if segment is an input field
      output "<input name="+segment name+" size="+segment
size+…+">"
 }
}
```

整个页面被嵌入<pre>和</pre>标记中，这样可以保持原有的字符界面的布局可以不被改变地显示在客户端的浏览器中，从而有效地解决了界面布局问题。

4.2 采用停止等待协议转换交互方式

从图 1 可以看出，遗产系统被封装成一个服务与 Web 交互部分独立运行，因此必须找到某种同步机制使界面交互和服务程序能够交替运行。为了实现这种交替式的运行，我们在服务程序和服务代理程序之间设计了停止等待协议用于这种同步。这个协议的工作原理如下所述：

(1)首先，Web 客户端请求第一个页面，或输入数据后进行提交，请求回答页面。

(2)Web 服务器收到这个请求，将其重定向到 jsp 文件。jsp 文件请求服务代理，即图 1 中的 service proxy 返回一个页面。

(3)服务代理检测被代理的服务是否已开始运行，如果没有，它以一个独立的线程启动后台服务，同时将自己挂起。如果被代理的服务已经启动，那么将用户的输入传送到被代理的服务中并重新启动先前停止等待的后台服务，同时也将自己挂起。

(4)经过上一步后，客户端的请求已经被挂起，后台服务端也就是遗产系统(已被翻译成 Java)开始运行。当后台服务遇到输出语句时，这些语句并不直接在屏幕上进行输出，而是重定向到 HtmlView 中；当遇到输入语句时，后台服务将自己挂起，同时通知第三步处于等待的 jsp 文件继续运行；其他语句并不直接影响界面显示和交互。

(5)jsp 文件从被代理的服务中取得已经准备好的页面(这个页面被记录在 HtmlView 中)返回给 Web 客户端。到此为止，完成了从客户端请求一直到获得返回页面的一次完全交互过程。

在第二步中，jsp 文件并不需要了解后台服务的细节，它只需要向服务代理请求返回当前的屏幕页面即可。但是屏幕页面可能并未准备好，比如后台服务还没有运行，因此在第三步，服务代理必须阻塞，同时启动后台服务开始运行，并且等待后台服务发送一个唤醒信号后继续运行。在第四步，后台服务可能请求用户输入，因此也需要停止等待，同时向服务代理程序发送唤醒信号。在第五步，服务代理得到唤醒信号后，从后台服务中取得当前的用户界面返回给客户端等待用户输入。

分析这个协议我们可以看出，其核心问题是需要在两个线程之间进行同步，服务代理和遗产服务两者之间都需要某种机制可以将对方挂起或重新启动。在这里我们用锁对象实现了对线程的挂起和唤醒。服务程序和服务代理程序都具有一个锁对象，用于挂起和启动线程，同时服务代理程序持有服务程序的锁对象，这样当用户提交输入后，可以唤醒服务程序继续运行。锁对象包含的主要算法用 Java 语法描述如下：

```
public class Suspendable{
  private boolean suspended = false;
  public synchronized boolean get(){ return suspended; }
  public synchronized void set(boolean status){
      suspended = status;
  }
  public void block(){
    (1)set( true );
    (2)while( get() ){ 休眠若干时间;//比如 10 毫秒 }
  }

  public void deblock(){
    (3)while( ! get() ){ 休眠若干时间;//比如 10 毫秒 }
    (4)set( false );
  }
}
```

算法中的 synchronized 表示相关的若干个方法之间是排他的，比如本例中的 get 和 set 方法必须是排他执行的，当程序控制流进入 get 方法体，但没有返回之前，set 方法体并不能被执行，只能等待。这一点是由 Java 虚拟机保证的语义。算法的核心是需要保证当程序调用 block 方法进入循环等待时，有 deblock 方法可以将其从循环等待中释放出来，不会出现 deblock 方法已经返回，而 block 方法还在等待 deblock 方法将其从循环中释放出来。另外，算法中的 while 循环必须使用类似休眠的办法，将本线程的时间片让出来，使得线程调度器可以调度其他线程运行。下面对锁对象算法做一个非形式化的正确性说明。

证明：上述的锁对象算法不会造成相互等待。

首先需要明确的是，用于同步的方法 block 和 deblock 通常在两个独立的程序控制流中调用，比如 Java 中的线程，否则就不会存在同步问题。为叙述的方便，这里假定有两个线程 A 和 B 需要同步。考虑上面代码中语句(1)(2)(3)(4)的执行顺序，根据程序的顺序执行语义，(1)必须先于(2)，(3)必须先于(4)，那么剩下可能的执行顺序有六种：即(1)(2)(3)(4)、(1)(3)(2)(4)、(1)(3)(4)(2)、(3)(1)(4)(2)、(3)(1)(2)(4)、(3)(4)(1)(2)。

(a)考虑第一种情况，线程 A 首先调用 block 方法，语句(1)将标志改为 true，然后在语句(2)进入循环等待。然后线程 B 调用 deblock 方法，语句(3)的循环条件并不满足，因此将标志改为 false 后退出。线程 A 将在下一次循环中，发现标志位为 false，从而退出，因而并不造成死锁。并且线程 A 中的方法的确在线程 B 中 deblock 作用下返回。

(b)考虑第二种情况，线程 A 进入 block 方法执行语句(1)，标志位变为 true。然后线程 B 进入 deblock 方法执行语句(3)，此时仍然不进入循环。然后线程 A 执行语句(2)进入循环等待，然后线程 B 执行语句(4)。与上一种情况类似，同样不会死锁，而且 block 方法在 deblock 将标志改为 false 后返回。

(c)考虑第五种情况。线程 B 首先执行语句(3),因为 suspended 初始化为 false,因此进入 while 循环等待。然后线程 A 执行(1),将标志改为 true,继续执行(2)进入 while 循环等待。当线程 B 再次开始执行时,因为标志已在语句(1)改为 true,因此退出循环。然后执行语句(4),将标志改为 false。当线程 A 再次执行时,因为标志为 false,因此也退出循环。其它几种情况也类似,这里不再赘述。

在上述六种情况下,算法都能够保证线程 B 调用 deblock 方法取消线程 A 调用 block 方法时的可能等待,两者之间不会造成死锁。

到此为止,证毕。

4.3 从界面传送输入到服务端

在 HtmlView 中保存有当前页面所有可输入字段的一个表,从这个表中我们可以根据参数名字查到对应的变量,从而将从客户端页面传送过来的变量值传送到对应的变量完成变量赋值操作。值得说明的是,由于在 Cota 翻译系统生成的目标代码即 Java 代码中,变量的名字已经丢失。在我们的实现中,使用了屏幕字段开始的坐标作为变量的名字,比如"accept price at 1201"中变量的名字映射为"var1201"。因为在一次交互中,不存在同时在一个位置接受两个变量的情况,所以当用户输入数据从客户端返回后,我们可以根据表单中返回的字段名在 HtmlView 保存的表格中查到对应的变量,从而可以将用户输入的数据准确无误地赋值到对应的变量。另外,从客户端输入的数据都是字符串形式的,赋给对应的变量之前需要进行适当的数据类型转换。在我们的实现中,每一个变量都被包装为一个对应数据类型的实例,并且每一种数据类型都提供了 moveFrom(String args)的接口,所以对于从客户端传送过来的字符串形的参数值 value,只需要调用对应变量的 moveFrom 方法即可,比如 price. moveFrom(value)。

4.4 多线程化

大多数的遗产程序都是单用户的,也就是说用户对整个程序的运行具有完全的控制权。但是在 Web 环境下,服务程序的运行不得不面对多个用户的同时访问。传统的解决方法是服务器端运行一个无限循环程序用于在特定的端口监听客户的请求,当某个用户的连接请求被检测到时,服务器端需要采用类似于 fork 或 createThread 的系统调用产生一个独立的进程或线程与用户进行交互,这样多个用户可以独立的与系统进行交互而不会相互影响。对遗产代码进行大范围的重构,将单机程序改造成多用户的客户/服务器模式的应用,这一过程的自动化还远远不能令人满意,还需要较多的人工干预。为了尽量自动化地将遗产程序迁移到 Web 环境,同时也因为上文采用基于锁机制的停止等待协议与客户端进行同步,这使得我们将单一执行流的遗产程序封装成一个独立的线程运行成为必需。用 Java 代码表示如下:

```
public class Service extends Thread{
   private Suspendable thisLock;
   …
   public void run(){
     legacy. main(params);
   }
   …
}
```

代码片断中的 Suspendable 对象 thisLock 是用来与客户进行同步的同步对象,而 run 方法就是整个线程的执行逻辑入口,也就是所要提供服务的遗产程序的执行逻辑入口,它直接调用遗产程序的 main()方法。当用户请求第一个页面时,由动态脚本程序,比如 jsp 控制启动一个新线程的运行,同时在余下的交互中,用户都是与此线程的应用逻辑进行交互。由于不同的用户与不同的遗产程序实例线程进行交互,因此可以解决多个客户并发请求的问题。

另一个值得注意的问题是,由于本方法采用严格的锁机制与客户进行同步,因此需要在客户与遗产程序服务进行的交互过程中,禁止客户端回退已经执行过的客户输入,否则可能导致服务端的锁同步机制出现混乱。这一点可以使用诸如"网页过期"等技术实现。另外当客户端异常终止时,比如断电或者用户主动关闭浏览器等,也就是说没有完成服务端所要求的全部流程时,服务端需要某种机制了解到交互的中断,否则将导致服务端线程的无限等待和资源的无限占据。解决这个问题可以在服务端中设定一个计时器,当某一次交互在预定时间内没有得到用户的反馈时,可以认为客户端异常终止,服务端就可以停止当前线程的运行并释放占据的资源。

5 实例研究

为了验证以上描述的界面自动转换方法的有效性,我们选取了一个相对规模较小的实际 Cobol 股票管理系统 mudemo 进行了实验。该系统总共有 5 个文件组成,共计 1798 行代码。该程序主要使用 Cobol 记录描述来定义屏幕布局,比如图 2 中的记录定义片断。

```
01  stock-00
   03    stock-00-0101  pic  x
         (0080)  value
         "----------------
         ----------------
         --------".
   03    stock-00-0201  pic  x
(0001) value"|".
   03  filler pic x(0078).
   03    stock-00-0280  pic  x
(0001) value "|".
```

```
01  stock-01 redefines stock-00
    03  filer pic x(0658).
    03  stock-01-code pic
    9(0006).
    03  filler pic x(0158).
    03    stock-01-description-1
    pic x(0053).
```

图 2 Cobol 屏幕定义记录

左边记录 stock-00 中的第一项 stock-00-0101 描绘了 80 个字符"-"组成的横线,stock-00-0201 和 stock-00-0280 则分别占据了第二行的第一个字符和最后一个字符,其值都是"|",程序主要用这些字符在屏幕上描绘一个方框。右边记录 stock-01 重新定义了左边记录,因为 stock-01 的第一个子项是一个填充项并且占据 658 个字符,也就是第 9 行开始的第 18 个字符处,接着第二个子项是一个 6 个十进制位的数字数据。通过这种叠加方式,程序可以在运行中构造任意复杂的用户界面。

在其实现主控界面的源代码文件 mudemo. cbl 中有如下所示的代码片段:

```
re-enter-choice.
  accept choice at 2445.
  evaluate choice
     when 1   call "STOCKIN"
     when 2   call "STOCKIOA"
     …
     when 5   go to endit
     when other go to re-enter-choice
  end-evaluate.
```

程序在屏幕的 24 行 45 列处接受用户输入的一个数字,然后根据用户的输入的不同数字调用不同的模块处理。由于对用户输入的反应在另外的函数甚至在另外的源文件中,如果采用文[5,8,9]中的方法,势必需要采用过程间分析技术,同时需要根据目标函数的情况进行可能的分解。这种情况下,手工对源程序代码进行调整将不可避免。另外,这里的调用只

是采用了静态调用方式，即直接给出被调用函数的名字。如果采用动态调用方式，比如调用一个字符串变量，程序在运行过程中根据变量的当前值确定调用的目标函数，那么在这种情况下，过程间分析将变得更加困难，函数分解方法因此受到限制，从而不得不再次使用手工辅助调整源代码。

在我们的试验中，mudemo 的全部 5 个文件经过 Cota 转换器[10]自动翻译为等价的 Java 源文件。程序中的输入和输出语句被翻译为对 HtmlView 屏幕映像的操作。整个系统被自动封装为一个后台服务以独立线程运行。用户的输入按照图 1 中虚线左边所示的控制框架经过服务代理重新定向到后台服务中。每一个输入语句的末尾都会插入启动客户端线程和停止自己所在线程的调用，用来实现停止-等待协议。

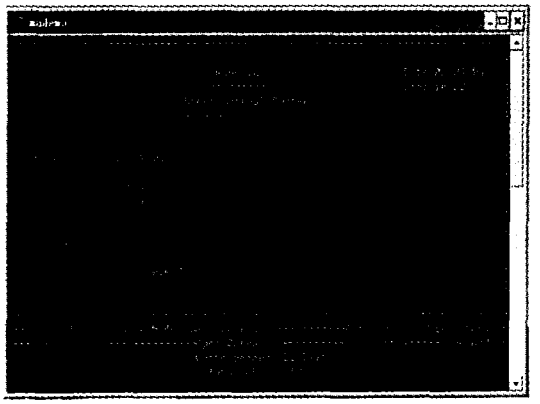

图 3　旧系统界面截图

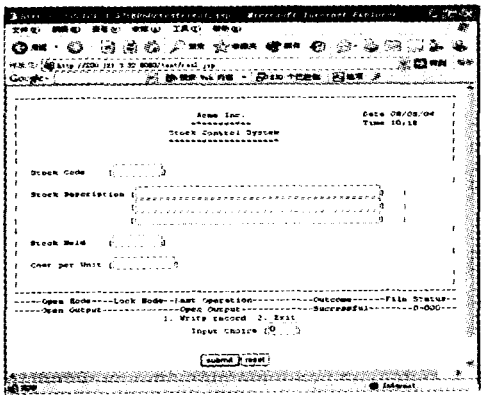

图 4　转换后 Web 界面截图

图 3 显示了旧系统中基于字符终端的用户界面，图 4 为转换后的新系统中的 Web 界面。从图中我们可以看出，新系统中的界面较好地保持了和旧系统一致的布局。在新界面的底部增加了两个按钮，分别是提交和取消按钮，用于对界面中输入域完成输入后将数据提交到服务器端或取消输入重新进行。两个系统的使用方法几乎完全相同。只是在新系统中，用户填写完数据后，需要用鼠标点击位于屏幕底端的 submit 提交按钮。和传统的用户界面相比，用户并不需要在命令行上输入复杂的命令行参数来执行 mudemo 程序，只需要输入网址即可。同时因为 4.4 节多线程的引入，原来的遗产系统在不需要任何手工调整的情况下，就可以通过 Cota 转换器自动将其迁移到 Web 平台，同时可以支持多个用户的访问。

结论　随着 Internet 技术的广泛普及，将基于字符界面的遗产系统部分或全部迁移到 Internet 平台，并且提供给用户简单一致的 Web 界面，使得普通用户可以通过熟悉的浏览器访问旧有系统所提过的各种服务，成为商业应用减少维护成本，保存旧有投资的有效手段。本文通过对传统遗产代码中用户界面的构成和交互特性的详细分析，实现了以停止-等待协议为核心的界面转换方法，同时给出了详细的锁对象算法及其正确性的非形式化证明。实验证明，采用这种界面转换方法可以有效减少采用函数分解技术进行界面转换时所带来的不可避免的手工辅助工作，可以正确地将字符交互方式转换到 Web 交互方式，从而自动化地将遗产系统中的字符界面迁移到 Web 界面。

参 考 文 献

1　Aversano L, Canfora G, Cimitile A, et al. Migrating Legacy Systems to the Web: an Experience Report. In: Proc. of the 5th European Conference on Software Maintenance and Reengineering (CSMR'01), Lisbon, Portugal, Mar. 2001. 14～16

2　Moore M M, Moshkina L. Migrating Legacy User Interfaces to the Internet: Shifting Dialogue Initiative. In: Proc. of the 7th Working Conf. on Reverse Engineering (WCRE'00), Brisbane, Australia, Nov. 2000. 23～25

3　Brad M, Mary Beth R. Survey on User Interface Programming, In: Proc. of SIGCHI 1992, Human Factors in Computing Systems, Monterey, CA, May 1992

4　Merlo E, de Montreal E P , Gagne P, et al. Reengineering User Interfaces. IEEE Software. Jan. 1995, 12: 64～73

5　Cimitile A, Visaggio G. Software Salvaging and the Call Dominance Tree. The Journal of Systems and Software, 1995, 28(2): 117～127

6　Canfora G, Cimitile A, De Lucia A, et al. Decomposing Legacy Programs: A First Step Towards Migrating to Client-Server Platforms, In: Proc, of 6th IEEE Intl. Workshop on Program Comprehension, Ischia, Italy, 1998. 136～144

7　Sneed H M. Accessing Legacy Mainframe Applications via the Internet. http://objectz. com/columnists/harry/08282000. html

8　Horowitz E. IEEE Software Special Issue on Migrating Software to the World Wide Web, 1988, 15(3)

9　Bodhuin T, Guardabascio E , Tortorella M. Migrating COBOL Systems to the WEB by using MVC Design Pattern, In: Proc. of the Ninth Working Conference on Reverse Engineering (WCRE'02)

10　武成岗，张兆庆，乔如良，等. 代码翻译中 PERFORM 和 GOTO 语句复合结构的变换. 软件学报，2004，15(4)