



于天恩 编著

Ruby

几门权威经典



Ruby 入门权威经典

于天恩 编著

内 容 简 介

本书基于 Ruby 的 1.86-25 版本进行讲解。全书共包括 8 章,包含与 Ruby 编程相关的一切核心基础知识以及使用 Ruby 开发常规 Web 应用程序的方法,可以分成三个部分。第一部分(第 1 章):Ruby 简介和环境配置。介绍了 Ruby 语言及 Ruby 框架的特点以及配置开发平台的方法。第二部分(第 2~4 章):Ruby 的基础知识。介绍了 Ruby 语言的基本结构、流程控制、数据类型、模块和线程等基础知识。第三部分(第 5~8 章):Ruby 的高级知识。介绍了 Ruby 的文件和目录操作以及数据库操作方法。最后提供了一些案例,用以实践 Ruby 的 Web 开发。

本书适用于对 Ruby 开发感兴趣的院校学生以及专业工程师。

图书在版编目(CIP)数据

Ruby 入门权威经典/于天恩编著. —北京:北京航空航天大学出版社,2009.4

ISBN 978-7-81124-576-9

I. R… II. 于… III. 程序语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2008)第 199245 号

Ruby 入门权威经典

于天恩 编著

责任编辑 罗晓莉

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(100191) 发行部电话:010-82317024 传真:010-82328026

<http://www.buaapress.com.cn> E-mail: bhpressell@263.net

印刷有限公司印装 各地书店经销

*

开本:787×1 092 1/16 印张:16.75 字数:429 千字

2009 年 4 月第 1 版 2009 年 4 月第 1 次印刷 印数:3 000 册

ISBN 978-7-81124-576-9 定价:28.80 元

前 言

说说 Ruby

Ruby 是一种语言,其单词含义为“红宝石”,许多女孩都用 Ruby 作为自己的名字。

既然是语言,当然是用来编程的。使用 Ruby 进行编程的效率很高,严格地讲是特别高。它是专门用来进行高效编程的,学习起来也很轻松。

Rails 是一种技术框架,其单词含义为“铁轨”,于是“Ruby On Rails”的含义就是铺满红宝石的铁轨。Rails 是 B/S 结构的编程框架,基于 Ruby 语言。这类似于 ASP 基于 VBScript 和 JSP(Struts)基于 Java。

单独学习 Ruby 是可以的,你可以使用 Ruby 去编写各种程序。不过,通常人们不会这么做,对我而言,最好的万能语言是 Java,用 JBuilder 开发 Java 程序是非常舒服的。

如果你也不打算使用 Ruby 作为万能语言的话,那么,最好使用它来做 Web 开发,也就是使用 Ruby On Rails 框架,这是 Ruby 最好的应用。

要学习 Ruby On Rails,首先要学习 Ruby。个人认为使用 Ruby On Rails 做 Web 开发的效率是非常高的,比 JSP,ASP,PHP 都要高,和 ASP.NET 2.0 也不相上下。尤其,Ruby On Rails 是开源的,免费的,因此,做 Web 开发,它是优于 ASP.NET 2.0 的选择。但,它是解释执行的,而不是编译执行的,这导致它和 JSP,ASP.NET 相比,在执行效率上有些差距。

虽然我没法证明 Ruby On Rails 是最佳的 Web 开发技术,但它在许多时候是我做开发的首选。推己及人,建议你使用,不是忽悠你。

写作动机

写书以前,我问了一个问题:外国权威书写得不好吗?国人翻译的不好吗?为什么还要我写?

Ruby 和 Rails 的书全世界有两本好的,这就够了,那些烂书我们可以不去看它。这是我原来的想法。

但论坛上许多人提出了重复的问题,他们在学习 Ruby 时捉襟见肘,不得要领。我向一个不懂 Ruby 的计算机博士推荐 Ruby 语言,他居然花了好长的时间都没有学会。这使我感到惊讶,在详细地了解了他的难处之后,我才明白国人遇到的问题。我开始觉得有必要写一本适合中国人学习的 Ruby 教材。

国人学习 Ruby,许多是认为这技术新鲜,有前途,能赚钱。Ruby 是一门总结性的优化型的语言,如果学习者具有 Java,C# 等语言的开发经验的话,要学习这门语言是非常轻松的。但多数人没有什么编程经验就直接学习 Ruby,这样一来,那些国外的权威著作就不太适合了。国人需要的是直接从零起点来讲解 Ruby 的书籍。

正因为此,我放下宝贵的休息时间,开始了此书的创作。

写作过程

作为一本基础教程,一本入门教程,写作的关注点是知识的次序,而不是知识的全面性和系统性。所以,我在这本书中充分考虑到初学者的实际情况,尽可能合理地编排次序,在这方面花去了不少时间。初稿完成后印刷给别人试用,收到了良好效果。之后联系出版社出版。然后,这本书就到了你——读者的手中。

本书特点

写书,应以教育为目的。此书内容以实践为主,并不深挖理论。教育是目的,写书是方法,方法服从于目的。书,不见得要写得多深多透。书的好坏要由书所面向的读者群来评价。我写书的原则是:实用。

本书包含了必要的理论,但以实践为主。所讲的理论都不是纸上谈兵,都是可以立即付诸实践进行工程应用的。代码可以直接拿出来用,只是不要忘了输入信息验证等基本的安全检查。

书中含有大量的案例,讲解由浅入深。浅,并不从“什么是程序设计”开始,所以读者需要具备一些编程的基础知识才能看懂。深,并没有深到“只可意会,无法言传”的地步,所以读者不需要担心无法看懂。

当然,我在书中的确阐述了一些思想,这是因为:我以为教育不单是要传知识,更重要地是要传思想。李阳的《疯狂英语》、胡敏的《新魅力英语》都是在给人以信念、生活的动力。一本书,如果能够启迪读者的思维,使读者找到自己人生的方向,那便是功德无量了。

本书面向初学者,专门为初学者编写。初学者并不仅仅意味着“不懂”和“不会”,还意味着“没有受到错误思想的污染”。

写一本大全,或者心得,之后或许能得到“厚重”,“精深”的美誉。但就当前的现实情况而言,写基础教程比写大全更有意义。基础教程面向的读者多,看到大量的读者通过我的书学到知识,改善自己乃至父母的生活,我会很开心。中国需要 Ruby On Rails 的基础教材,很多人要学习这门技术却不知道从何处入手,他们就是我写书的读者群。

这本书就是要让你知道:原来 Ruby 如此简单!

本书保持了我一贯的风格,以循序渐进为写作原则,先介绍概念然后再讲应用,应用程序由简入繁。这看起来像废话,多简单啊!每个人都这样啊!说实话,这的确也是废话,但就是这么一件简单到掉渣的事情,却没几个人能做好。有人写 Rails 基础教材,先写一堆 Rails 心得,然后讲 Ruby。也有人写 Java Web 开发教材,先讲 JSP 后讲 Servlet。我猜这书是写给他们自己看的。所以,世界上的书不少,但有资格作为基础教材的还真是凤毛麟角。

这本书的循序渐进,读者可以体会得到。写一本 Ruby 的基础教材不难,但要把循序渐进贯彻其中,就要用心编排了,这是难点。介绍运算符之前绝对不使用运算符,介绍变量之前,不使用变量。所有出现的知识都在前面有所铺垫。这是我希望做到的。

我这本书不是面向天才的,天才学 Ruby 哪里需要看书啊,只要浏览一下技术文档也就足

够了,最多几个小时就精通了。

我这本书面向的是你,需要技术,需要工作,需要赚钱,需要养家的你。衷心祝愿你成功!努力!奋斗!

本书内容

这本书基于 Ruby 的 1.8.6-25 版本来讲解。全书共包括 8 章,可以分成三个部分。你将从这里学习到和 Ruby 编程相关的一切核心基础知识以及使用 Ruby 开发常规 Web 应用程序的方法。

第一部分(第 1 章):Ruby 简介和环境配置。介绍了 Ruby 语言及 Ruby On Rails 框架的特点以及配置开发平台的方法。

第二部分(第 2~4 章):Ruby 的基础知识。介绍了 Ruby 语言的基本结构、流程控制、数据类型、模块和线程等基础知识。

第三部分(第 5~8 章):Ruby 的高级知识。介绍了 Ruby 的文件和目录操作以及数据库操作方法。最后提供了一些案例,用以实践 Ruby 的 Web 开发。

Ruby 包含的内容很多,操作系统编程、网络编程等应有尽有。手册里面写得很详细。但我写的是基础教程,所以不能面面俱到,写的是基础知识中的基础,目的是使读者入门,然后再向前走一步以进行常规应用开发。同时,恰恰由于这本书是基础中的基础,因此,在 Ruby 迅速发展并且不时发生巨大变化的现实情况下,本书将始终保持着生命力。

学完这本书,读者就可以掌握所有 Ruby 核心基础知识并应付常规开发,可以使用 eruby 解决常规的 Web 开发问题。读者还将具备看懂高级 Ruby 和 Rails 资料的能力。这便是本书最浅显的意义。

之后,如果读者学有余力,想要继续学习 Ruby 和 Rails,那么我推荐《Programming Ruby》和《Agile Web Development with Rails》。对这两本书,我曾认真地看了,很不错,都是很好的书,是同类中的翘楚。

谅解和支持

这本书是我用心写成的。

我喜欢读书,读过很多书,读好书对我来说是享受,它能减少我的思考时间,哪怕只有几分钟,也是相当值得的。做人做事得将心比心,我写的每一本书都是用心的,用心,尽力做好。这样就可以节约你们的时间。

我尽心尽力地写了这本书,从章节的安排到案例的编写,都是经过仔细揣摩的。我希望做到最好。然而,没有最好,只有更好。

我不能确定这里面没有错误,但我很希望经过多次的检查可以使里面的错误减到最少。即使有错,也应该是文字或者其他一些错误,而不会有涉及程序设计或算法的原则性错误。

我希望你说这是一本好书,但我更希望你告诉我这本书的缺点和不足。因为我知道这里一定有不足。我不能因为自己用了心,就认为自己做到了完美。问心有没有愧是一回事,做得

好不好是另一回事。前者,我说了算,后者,你们说了算。

如果你在本书中发现任何问题,我都很希望你能告诉我,以帮助我改良它。而且,你有可能收到意想不到的神秘礼物,或许是我珍藏十几年的树叶书签,或许是我亲手叠的纸飞机,或许是我写的诗词……总之,有钱买不到!

我的邮箱:yutianen@163.com 将真诚地用来为读者答疑解惑,同时接受所有读者的反馈意见。需要说明的是,我工作之外的时间非常少,如果不能及时查收或回复你的邮件,希望谅解。我若看到了你的邮件,就一定会认真回复的。

希望这本书在你做 Ruby 应用开发时有帮助,那便是我的欣慰。

如果这是你学习 Ruby 的第一本教材,我赞赏你的眼光,你选中了一本用心写的书。

愿这本书带给你知识,带给你幸运,带给你真实的人生。

一些勉励

艺多不压身。

家财万贯不如艺在身。

学技术很有用,可以让人心里踏实。只要技术活着,至少我们可以靠技术活着。技术死了,我们再去学新技术,从而维持自己活着。这是一种与时俱进。

只要肯钻研,没有学不通的技术。

人类若不灭亡,技术就无穷无尽。只要有兴趣,有动力,就可以不停地学,锲而不舍,最终把所有的技术都学会。

我不建议把所有的心思都用在技术上。我更加不认为技术的进步就是人类的进步,人类的社会,始不可知,终无可料。不要执着于任何事情,别想不开。

回到技术,诚然,想要掌握主流的所有软件开发理论和技术并不是什么难事,确切地说这非常简单。但,如果你为了学技术而学技术,而且并非天资聪颖轻松学会,而是花了大力气去苦学技术,那就太不值得了。看看你失去了多少和亲人朋友在一起的时间,看看你失去了多少登山野游的时间,看看你在感情和健康方面的损失,想明白这个事情,不要舍本逐末。

生而知之者,上也;学而知之者,次也;困而学之者,又其次也。

“人事有可陷者,亦有不可陷者”,陷于技术,不值得。

悟性,因人而异,不可强求,欲望,因人而异,不应执着。

学技术是简单的事情,做工程师是简单的事情,做科学家也是简单的事情,这些事都比做思想工作轻松得多。

但简单的事情未必好办,因为简单是相对的。不过有一点是肯定的,“世上无难事,只怕有心人”。即使我们无法确定自己可以做成每件事情,但至少我们可以对想做的事情用心,尽力,做到问心无愧。

诚然,人的天资很重要,非常重要,相当重要。但有几句话要提醒读者:“勤能补拙是良训,一分辛劳一分才”,“只要有恒心,铁杵磨成针”。天资是天给的,不是人定的。但在天资之外,你还有很多事情可以做,通过勤奋、坚持、刻苦,纵然超不过天才,你也定然能成为超越常人的

人才,这不是也很好吗?

幸福在哪里? 在精心的耕耘中,在艰苦的劳动里。

努力吧! 用晶莹的汗水去酿造成功,用辛勤的双手去编织绚丽的人生!

衷心感谢

石志国,他的《ASP 精解案例教程》是我编程的开端,提起编程,总会想起。想起 2004 的初冬在寒风凛冽的雨夜读书的事情,想起那时陪伴我的歌,不禁潸然泪下。

对哈工大天萌联合的一切成员表示感谢! 那些曾跟我在一起的朋友,我会记得你们为我泡的每一杯咖啡和茶。那些始终保持独立的朋友,我也祝愿你们会有更加辉煌的未来。我创立的天萌联合,我们的天萌联合,永远是哈工大最强、最自由的社团,你们这些天萌的元老的名字,将永远铭刻在我的心里。

本书原代码可以在北京航空航天大学出版社网站(<http://www.buaapress.com>)下载中心下载。

于天恩 中国·天人居
2008 年 11 月

目 录

第一部分 Ruby 简介和环境配置

第 1 章 配置 Ruby 运行环境	1
1.1 面向对象和 MVC 模式	1
1.1.1 面向对象	1
1.1.2 MVC 模式	1
1.2 Ruby 概述	2
1.2.1 Ruby 的由来	2
1.2.2 Ruby 的优点	2
1.3 Ruby 的安装	4
1.3.1 下载 Ruby	4
1.3.2 安装 Ruby	7
小 结	11
思考和练习	11

第二部分 Ruby 的基础知识

第 2 章 Ruby 语言基础	12
2.1 最基本的知识	12
2.1.1 基本输出	12
2.1.2 引号的用法	13
2.1.3 转义字符输出	14
2.1.4 连句和换行	15
2.1.5 连行符	17
2.1.6 注释符	18
2.1.7 局部变量	18
2.1.8 基本输入	19
2.1.9 数字和字符串连接	19
2.2 类	20
2.2.1 最简单的类	21
2.2.2 类的基本使用	21
2.2.3 继 承	22

2.2.4	单态方法	24
2.2.5	方法访问控制	25
2.2.6	属性读写控制	27
2.2.7	垃圾收集	31
2.2.8	异常处理	31
2.3	常量和变量	34
2.3.1	常量	35
2.3.2	全局变量	35
2.3.3	实例变量	37
2.3.4	局部变量	37
2.4	运算符	37
2.4.1	算术运算符	37
2.4.2	关系运算符	38
2.4.3	逻辑运算符	39
2.4.4	其他运算符	40
2.5	流程控制	40
2.5.1	顺序结构	40
2.5.2	选择结构	41
2.5.3	循环结构	45
2.6	块	48
2.6.1	块的概念	48
2.6.2	块的基本使用	49
2.6.3	带参数的块	50
2.7	迭代器	51
2.7.1	迭代的概念	51
2.7.2	编写迭代器	52
2.8	过程对象	54
2.8.1	创建过程对象	55
2.8.2	把过程对象当作参数	55
小 结		56
思考和练习		56

第3章 Ruby 的数据类型 57

3.1	数 字	57
3.1.1	数字的基本使用	57
3.1.2	数字的常用方法	59
3.1.3	数学计算方法	60

3.2 字符串	62
3.2.1 字符串的基本用法	62
3.2.2 字符串的常用方法	63
3.2.3 字符串方法总结	91
3.3 正则表达式	92
3.3.1 Ruby 正则表达式的基本用法	92
3.3.2 正则表达式在字符串函数中的使用	94
3.4 日期和时间	94
3.4.1 Time 对象	94
3.4.2 Date 和 DateTime 对象	98
3.5 散列表	99
3.5.1 散列表的构造	99
3.5.2 散列表的常用方法	100
3.6 区间	103
3.6.1 区间的概念	103
3.6.2 区间的使用	104
3.7 数组	106
3.7.1 构造数组	106
3.7.2 数组的主要方法	107
3.8 结构体	114
3.8.1 建立结构体	114
3.8.2 结构体的主要方法	115
3.9 数据类型转换	116
3.9.1 通用的转换方法	116
3.9.2 自定义转换方法	118
小 结	118
思考和练习	118

第 4 章 模块和线程 119

4.1 模 块	119
4.1.1 模块的概念	119
4.1.2 Mixin	122
4.1.3 Ruby 的命名约定	122
4.2 线 程	123
4.2.1 线程的概念	123
4.2.2 线程的同步	125
小 结	128
思考和练习	128

第三部分 Ruby 的高级知识

第 5 章 文件和目录	129
5.1 文件操作	129
5.1.1 文件操作的概念	129
5.1.2 文件的基本操作方法	129
5.1.3 文件操作标准方法	138
5.2 目录操作	146
5.2.1 目录操作的概念	146
5.2.2 目录操作的方法	146
小 结	153
思考和练习	153

第 6 章 Ruby 的数据库操作	154
6.1 Ruby 数据库访问的概念	154
6.1.1 数据库访问的方式	154
6.1.2 数据库访问的目的	154
6.2 访问 Access 数据库	154
6.2.1 配置环境	154
6.2.2 执行数据操纵语句	158
6.2.3 执行数据查询语句	163
6.3 访问 SQL Server 数据库	170
6.3.1 建立 ODBC 数据源	170
6.3.2 访问数据库	174
6.4 访问 MySQL 数据库	179
6.4.1 下载和安装 MySQL/Ruby 模块	179
6.4.2 使用 MySQL 模块进行数据库访问	181
6.4.3 安装 DBI	192
6.4.4 使用 DBI 访问 MySQL 数据库	197
小 结	201
思考和练习	201

第 7 章 桌面应用和 Web 开发	202
7.1 Ruby 的桌面开发	202
7.2 Ruby 的 Web 开发方法	202
7.2.1 CGI 类	202

7.2.2	eRuby 概述	204
7.2.3	eruby 的基本使用	207
7.2.4	文件包含	208
7.2.5	中文显示	209
7.2.6	参数的传递和接收	211
7.3	详解表单处理	214
7.3.1	表单的提交	214
7.3.2	表单的接收	214
7.4	文件操作	218
7.4.1	文件读取	218
7.4.2	文件写入	220
7.5	数据库操作	222
7.5.1	数据读取	222
7.5.2	分页显示数据	224
7.5.3	数据更新	229
7.6	session 的基本用法	233
7.7	Web 开发案例	234
7.7.1	留言本(基于文本文件)	234
7.7.2	聊天室(基于文本文件)	237
7.7.3	留言本(基于数据库)	242
7.7.4	聊天室(基于数据库)	245
小 结	251
思考和练习	251

第一部分 Ruby 简介和环境配置

第 1 章 配置 Ruby 运行环境

本章要点

本章介绍 Ruby 的历史和发展状况,主要介绍其编程特点,从而使读者从整体上了解这种编程技术。

1.1 面向对象和 MVC 模式

在接触 Ruby 之前,先来看以下这两个概念。这是两个基本概念,是和 Ruby 关系最紧密的概念。

1.1.1 面向对象

面向对象是一种程序设计方法,其基本思想是使用对象、类、继承、封装以及消息等基本概念来进行程序设计。它是从现实世界中客观存在的事物(即对象)出发来构造软件系统,并在系统构造中尽可能运用人类的自然思维方式,强调直接以事物为中心来思考问题、认识问题,并根据这些事物的本质特点,把它们抽象地表示为系统中的对象,作为系统的基本构成单位。这可以使系统直接地映射问题域,保持问题域中事物及其相互关系的本来面貌。

从程序设计的角度来看,面向对象的程序设计语言必须有描述对象及其相互之间关系的语言成分。这些程序设计语言可以归纳为以下几类:系统中一切皆为对象;对象是属性及其操作的封装体;对象可按其性质划分为类,对象成为类的实例;实例关系和继承关系是对象之间的静态关系;消息传递是对象之间动态联系的唯一形式,也是计算的唯一形式;方法是消息的序列。

本书不详细介绍面向对象理论的各个方面,市面上介绍这方面的书非常多。后面的讲解中将假设读者已经了解面向对象理论,所以不会对“类、对象、方法以及继承”这些概念做论述。Ruby 是纯粹的面向对象语言。

1.1.2 MVC 模式

MVC 模式即是 Model - View - Controller 模式,中文翻译为“模型-视图-控制器”。MVC 应用程序总是由这三个部分组成。事件导致控制器改变模型或视图,或者同时改变两者。只要控制器改变了模型的数据,所有依赖的视图都会自动更新。类似地,只要控制器改变了视图,视图会从潜在的模型中获取数据来刷新自己。MVC 模式最早是 smalltalk 语言研究团提出的,应用在用户交互应用程序中。

MVC 模式具有清晰的逻辑划分。Controller 用来处理请求的事务,负责响应客户对业务逻辑的请求并根据用户的请求行为,决定将哪个 View 页面发送给客户。Model 则负责数据的处理。其结构如图 1-1 所示。

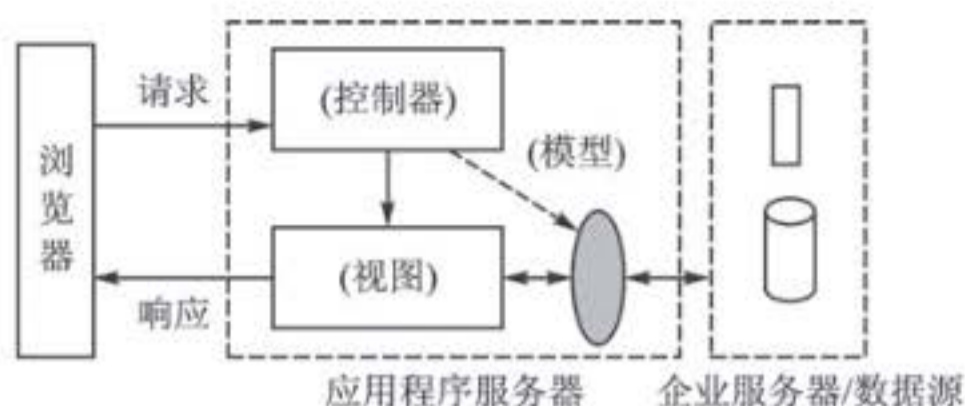


图 1-1 MVC 结构

MVC 可以使每项技术都发挥各自的长处,初始的请求由控制器来处理。控制器调用商业逻辑和数据处理代码,并创建模型来表示相应的结果。然后控制器确定哪个页面适合于表达这些特定的结果,并将请求转发到相应的页面(视图)。

MVC 模式能够有效地区分不同的开发者,避免彼此间的互相影响,充分发挥每个人的特长,在开发大型项目时表现出的优势尤其突出。

Rails 是完善的 MVC 模式的 Web 框架。

1.2 Ruby 概述

1.2.1 Ruby 的由来

Ruby 的作者是日本人,松本行弘。他很喜欢编程,也精通很多编程语言,对 Perl 和 Python 都有相当的了解。为了编写一个最好的脚本语言,他吸取了 Perl,Python 等语言的精华,写成了 Ruby。

Ruby 是 1995 年公开发布的。最开始的名字叫 Red Stone(红石头),后改为 Ruby(红宝石)。

1.2.2 Ruby 的优点

Ruby 可以做桌面应用开发,也可以做 Web 开发。在笔者精通的技术中,与 Ruby 类似的语言还有 Perl,Python,PHP 和 Java。写语言不难,写好却不容易。Ruby 现在还存在问题,有许多不完善的地方,许多地方都在改进,所以,它是一门处于发展中的语言。

是什么使得 Ruby 如此流行呢? 大约有以下几个方面。

(1) 解释执行

Ruby 是解释型语言,因此不需编译即可运行,这样就提高了调试的速度。当然,也要注意解释型语言的执行效率低的问题。

(2) 变量无类型

Ruby 的变量没有类型。相应地,错误检查功能也变弱了。Ruby 的标识名区分大小写。

(3) 变量不需要声明

所有变量均无需声明即可使用。

(4) 语法简单

Ruby 语法比较简单,类似 Algol 系语法。

(5) 不需要内存管理

具有垃圾回收(garbage collect,GC)功能,能自动回收不再使用的对象。

(6) 纯粹的面向对象

Ruby 是纯粹的面向对象语言,因此包括整数等基本数据类型都是对象,它们都有发送信息的统一接口。在 Java 里面,基本类型和类是不一样的(如:int 是基本类型,Integer 则是类)。

(7) 迭代器

迭代器功能可以将循环抽象化,从而使得代码简洁。

(8) 功能强大的字符串操作

Ruby 以 Perl 为样板创造出了功能强大的字符串操作和正则表达式检索功能。

(9) 超长整数

添加超长整数功能后,可以计算非常大的整数。例如计算 400 的阶乘也轻而易举。

(10) 可以直接访问操作系统

Ruby 可以使用(UNIX 的)绝大部分的系统调用。单独使用 Ruby 也可以进行系统编程。

(11) 特殊方法

可向某对象添加方法。

(12) 用模块进行混合插入

Ruby 故意舍弃了多重继承,但拥有混合插入功能。使用模块来超越类的界限以共享数据和方法等。

(13) 闭 包

可以将某过程片段对象化。对象化后的该过程片段就称作闭包。

(14) 功能强大的字符串操作/正则表达式

Ruby 以 Perl 为样板创造出了功能强大的字符串操作和正则表达式检索功能。

(15) 具有错误处理功能

错误处理功能使得人们可以编写代码处理出错情况。

(16) 动态加载

若操作系统支持,可以在运行时读入对象文件。

Ruby 将许多常用的功能以方法的形式提供了出来,这使得在很多时候都不需要去编写函数来实现需要的功能。许多人使用过 Ruby 之后,都发现编程的效率大大提高了。使用 Ruby 编程很快,开发效率高,程序员就可以提高工作效率。于是这语言越来越流行。

1.3 Ruby 的安装

1.3.1 下载 Ruby

从 RubyForge 网站中,可以下载到与 Ruby 开发相关的软件。进入 <http://rubyforge.org/> 网站,如图 1-2 所示。



图 1-2 RubyForge 网站

进入 <http://rubyforge.org/projects/ruby/>,可以下载 Ruby,如图 1-3 所示。

单击“下载”按钮,页面转入 Ruby 不同版本的下载列表,如图 1-4 所示。

为了安装 Ruby,需要下载的是 Ruby Installer,这需要进入 <http://rubyforge.org/projects/rubyinstaller/>,如图 1-5 所示。

单击“下载”按钮,页面转入 Ruby Installer 不同版本的下载列表页面,如图 1-6 所示。

当前(2007-6-22)最新版本的 Ruby Installer 是 186-25,将其下载,结果如图 1-7 所示。



图 1-3 下载 Ruby 界面



图 1-4 Ruby 下载列表



图 1-5 下载 Ruby Installer 界面



图 1-6 Ruby Installer 下载列表



图 1-7 Ruby Installer 软件

1.3.2 安装 Ruby

安装 Ruby 的过程很简单,其步骤如下

(1) 开始安装

双击 ruby186-25.exe,即可开始安装,如图 1-8 所示。



图 1-8 安装 Ruby

单击“Next”按钮,进入下一个窗口,如图 1-9 所示。

单击“I Agree”按钮,接受协议。进入下一个窗口,如图 1-10 所示。选择安装组件。

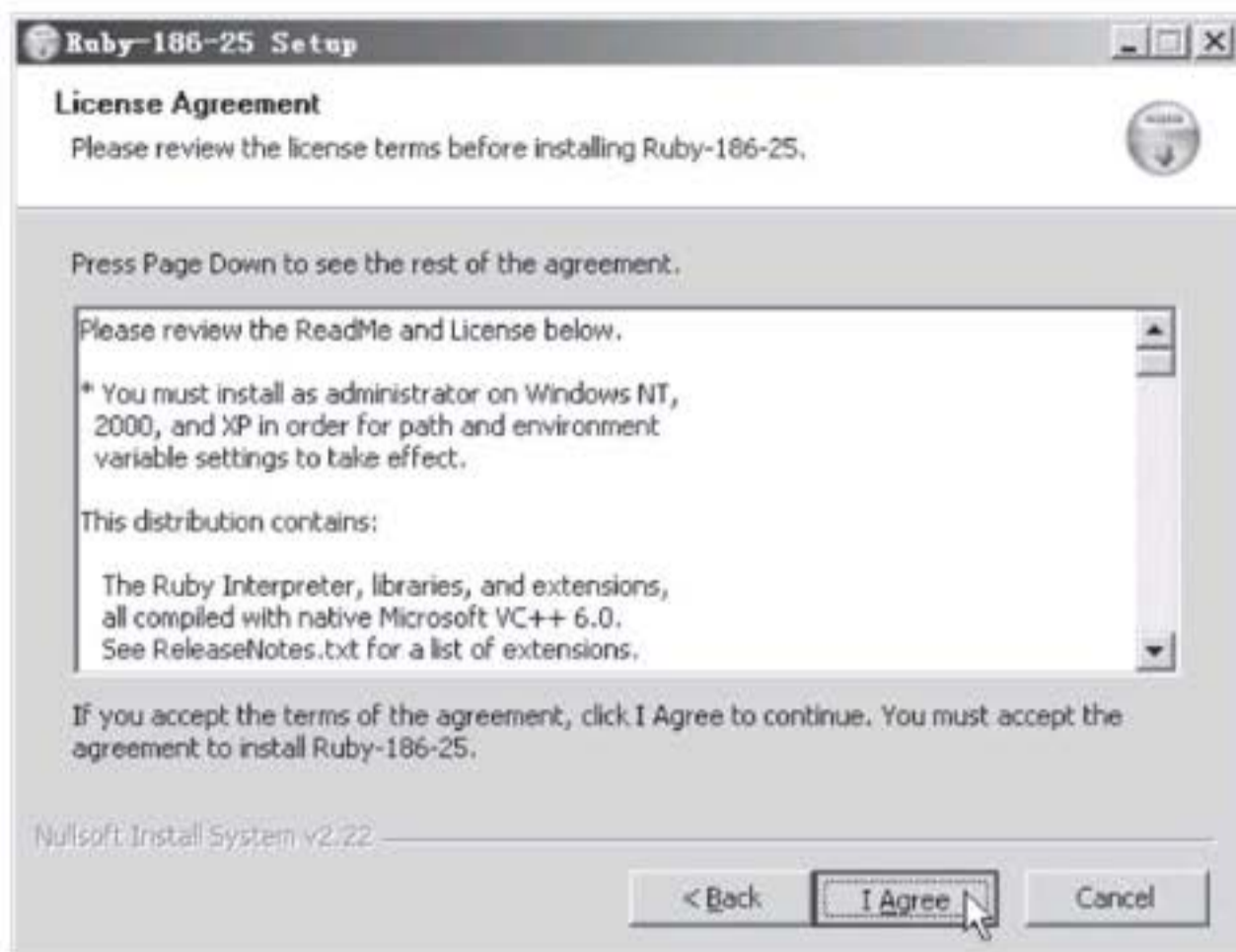


图 1-9 接受协议

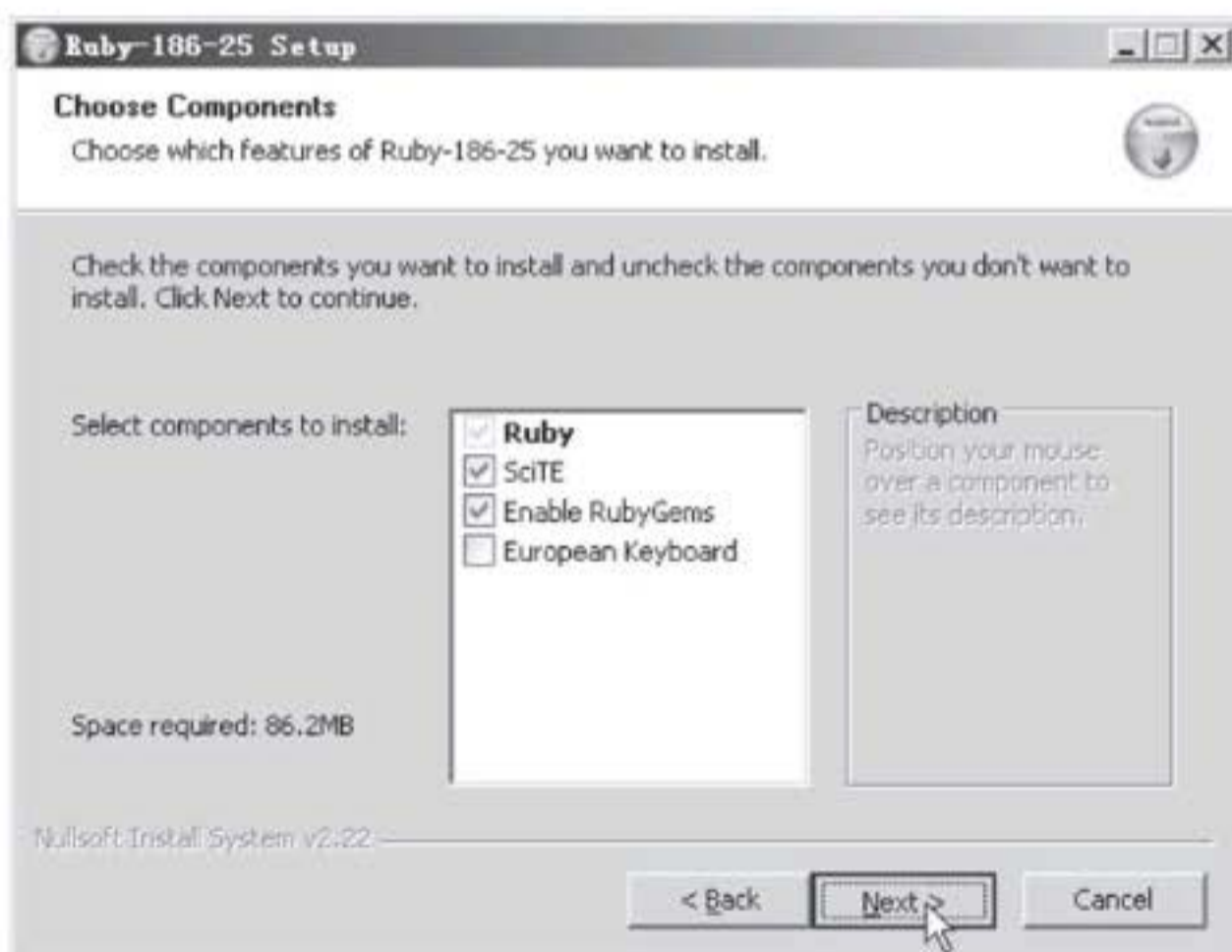


图 1-10 选择安装组件

(2) 修改安装路径

默认情况下, Ruby 被安装在 c:\ruby 目录下, 可以对其进行修改, 本书将 Ruby 安装到 d:\ruby 目录下, 如图 1-11 所示。

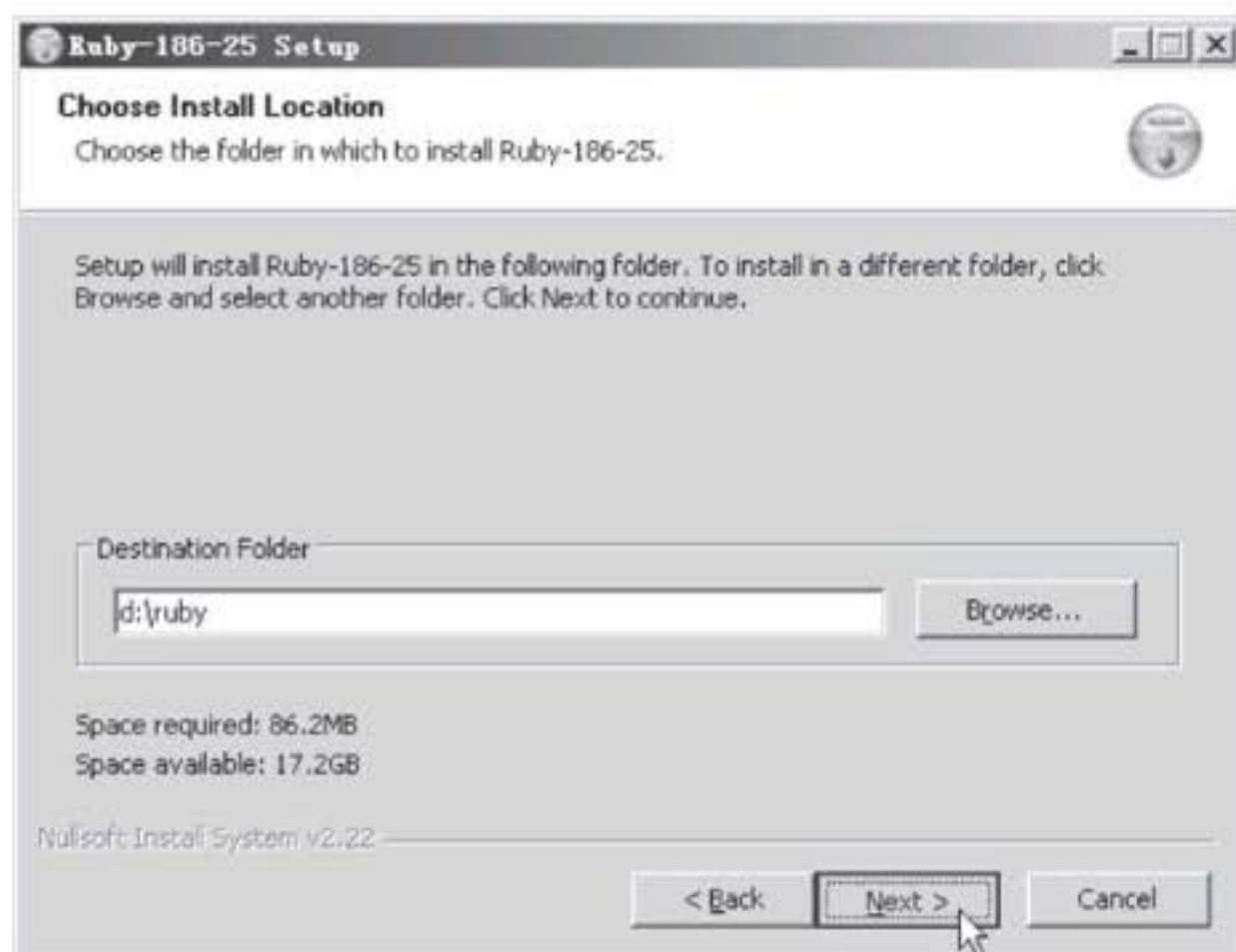


图 1 - 11 修改安装路径

(3) 继续安装

单击“Next”按钮，后面的安装过程保持默认即可，如图 1 - 12 和图 1 - 13 所示。

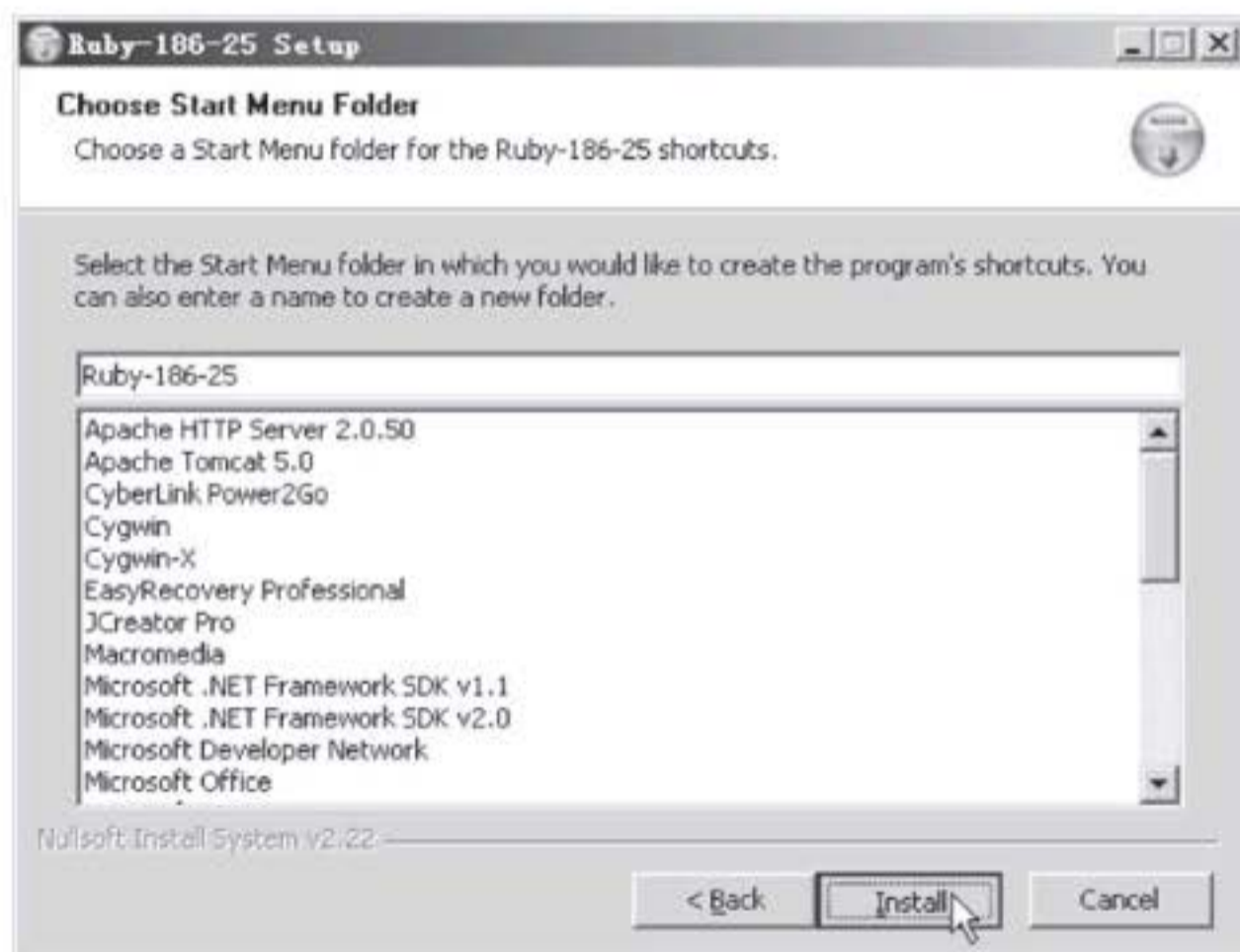


图 1 - 12 设置开始菜单

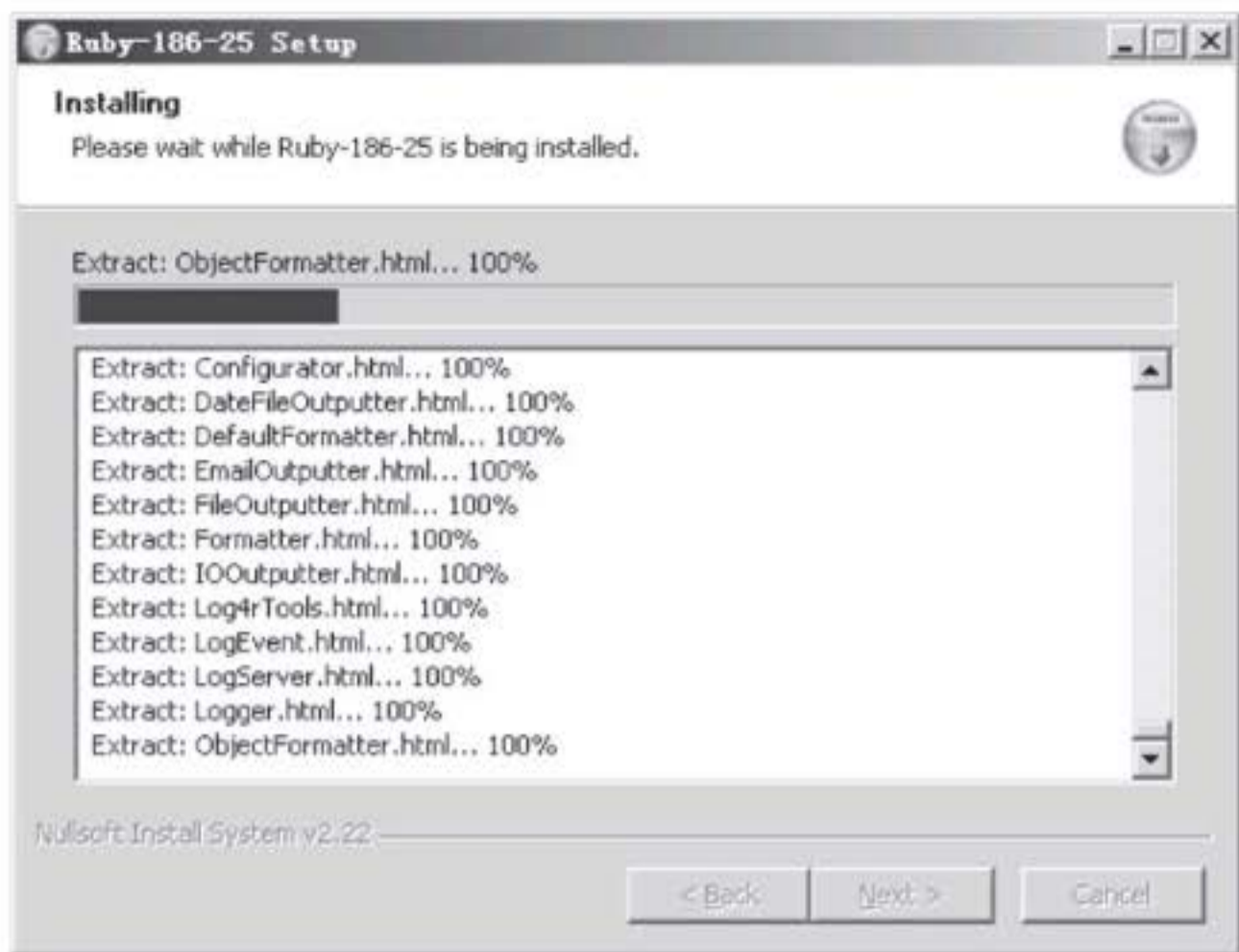


图 1-13 安装过程

(4) 安装结束

安装结束后,窗体如图 1-14 所示。



图 1-14 安装结束

查看 D:\ruby 目录,如图 1-15 所示。

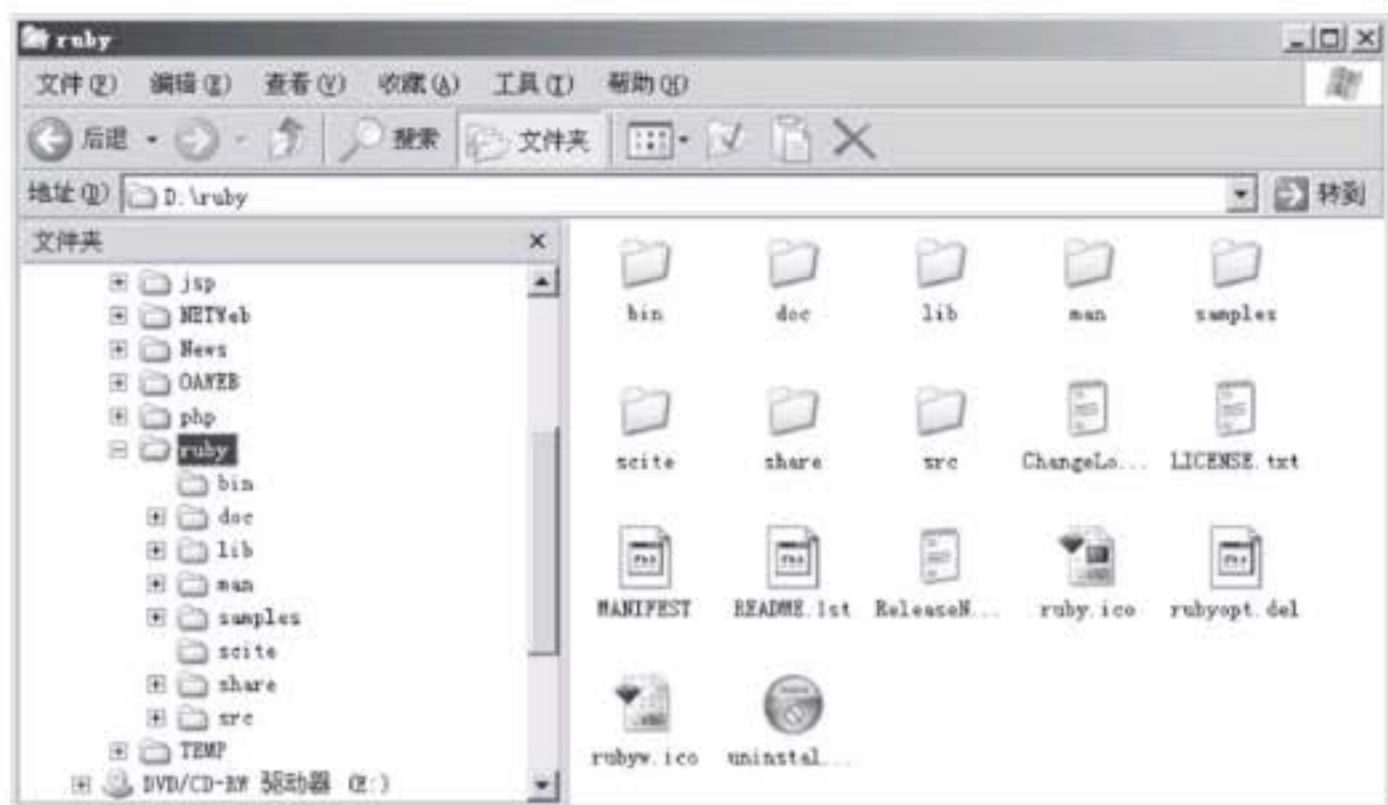


图 1-15 Ruby 安装目录

(5) 测试安装

进入 DOS 窗口,在命令行中输入“ruby -v”命令,将会显示出 Ruby 的版本号,如图 1-16 所示。这说明 Ruby 的安装成功了。

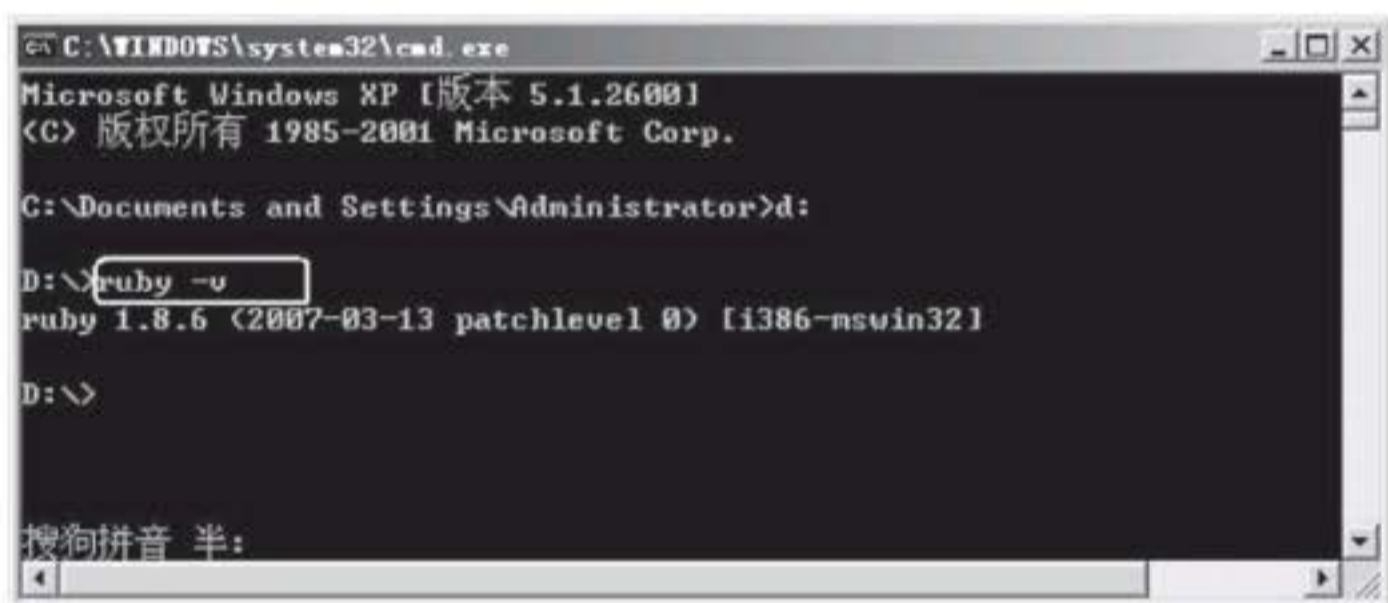


图 1-16 测试安装

小 结

本章介绍了 Ruby 的发展史、特性,使读者对其有所了解。

思考和练习

Ruby 具有哪些特点?

第二部分 Ruby 的基础知识

第 2 章 Ruby 语言基础

本章要点

本章主要介绍 Ruby 的语言基础。包括 Ruby 的常量、变量、运算符、流程控制、面向对象特性、块、迭代器以及过程对象等。

2.1 最基本的知识

对于一门语言,最基本的知识就是输入输出。Ruby 有一套输入输出库,本节介绍其最基本内容,并借此机会将最底层概念一并讲述,学完此节之后,读者将可以自由做尝试。

2.1.1 基本输出

最基本的输出方法是:puts 和 print。另外一个常用的输出方法是 printf,它很适合做格式化输出。类似于 C 语言的 printf 函数,在 PHP 中也有类似的函数。

可以用 puts"tianen"或 prints"tianen"来输出字符串“tianen”。现在,在 Ruby 环境中运行这些,其方法是:在命令行中输入“ruby -e 'puts"tianen"'”,需要执行的 Ruby 语句按照这个格式输入就可以。如图 2-1 所示。

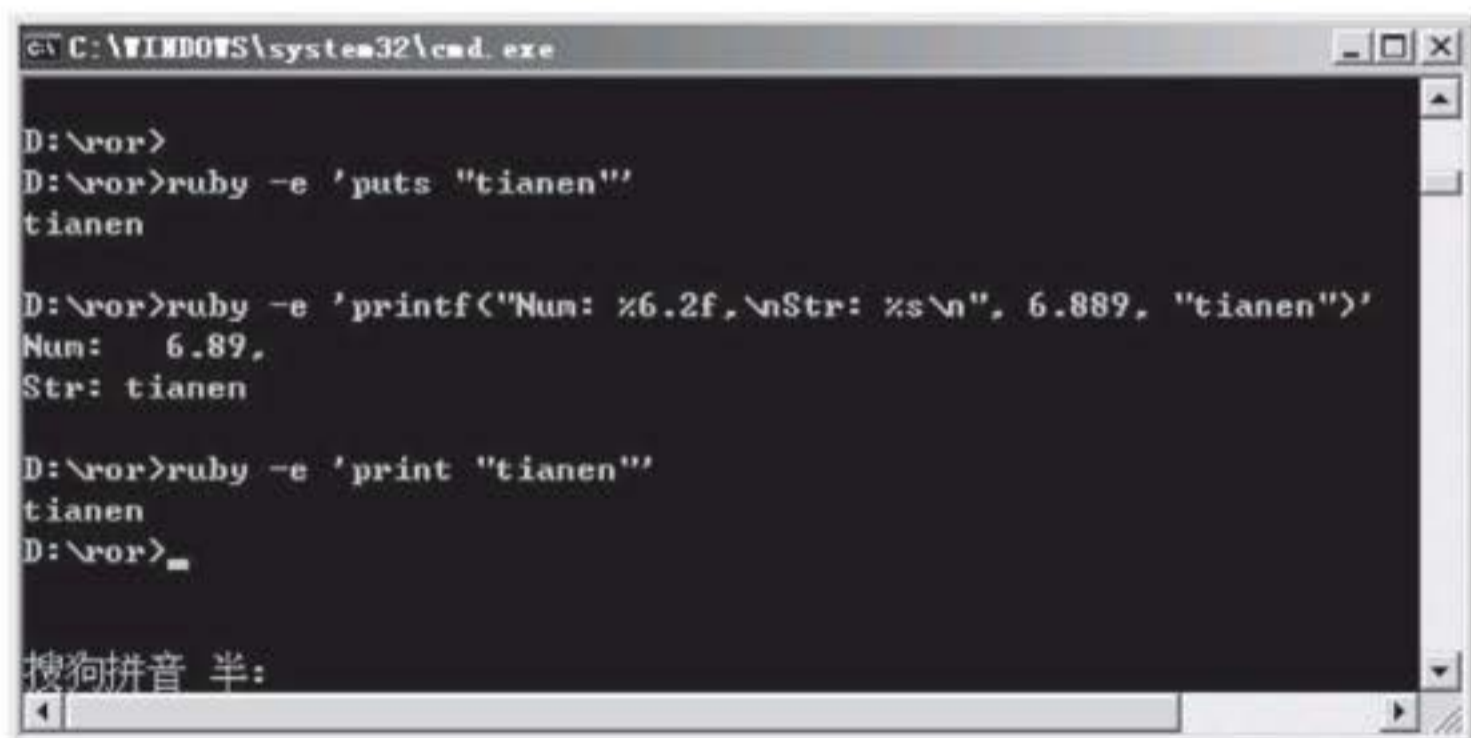


图 2-1 基本输出方法

这种通过命令行来执行 Ruby 的方法很不方便,可以将 Ruby 语句写入文件,然后执行这个文件,会达到相同的效果。

现在,把 `puts"tianen"` 和 `prints"tianen"` 这两条语句存入 `a.rb` 文件中。这是一个文本文件,是 Ruby 的程序文件。如图 2-2 和图 2-3 所示。



图 2-2 a.rb 文件的内容



图 2-3 rb 文件的图标

在命令行中,可以用“`ruby a.rb`”来执行 Ruby 程序,如图 2-4 所示。这里把 Ruby 程序文件放到 `D:/ror` 目录下,以便于管理。



图 2-4 执行 ruby 程序

2.1.2 引号的用法

Ruby 中引号的用法与 PHP 类似,有单引号和双引号两种,单引号中可以包含双引号,双引号中也可以包含单引号。在大多数情况下,这两种符号是通用的。所以,输入“`puts "tianen"`”和输入“`puts 'tianen'`”的运行效果是一致的。如图 2-5 所示。



图 2-5 引号的用法

2.1.3 转义字符输出

基于前面的知识,要想在一条语句中输出半角单引号或双引号都很容易。如下面案例所示。

案例名称:输出引号

程序名称:b.rb

```
puts "tianen'"
puts 'tianen'"
```

运行结果如图 2-6 所示。

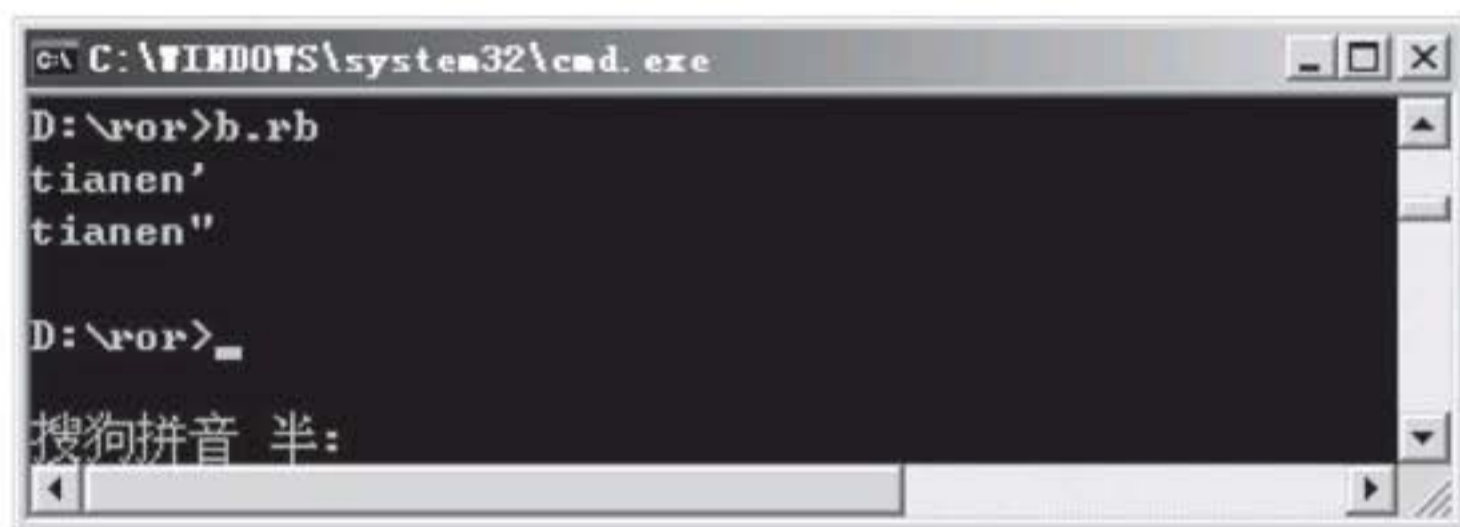


图 2-6 输出引号

现在提出一个问题:要输出字符串“tianen's "universe"”该怎么办? 这个字符串中同时含有半角的单引号和双引号。这时要使用转义字符。这个概念对于稍有编程经验的人来说都不陌生。C 语系的语言(如:Java,Ruby,C)等使用“\”作为转义字符,VB 中使用“'”和“””作为转义字符。

案例名称:转义字符输出

程序名称:c.rb

```
puts "tianen\' \"universe\\""
puts 'tianen\' \"universe\"'
```

运行结果如图 2-7 所示。

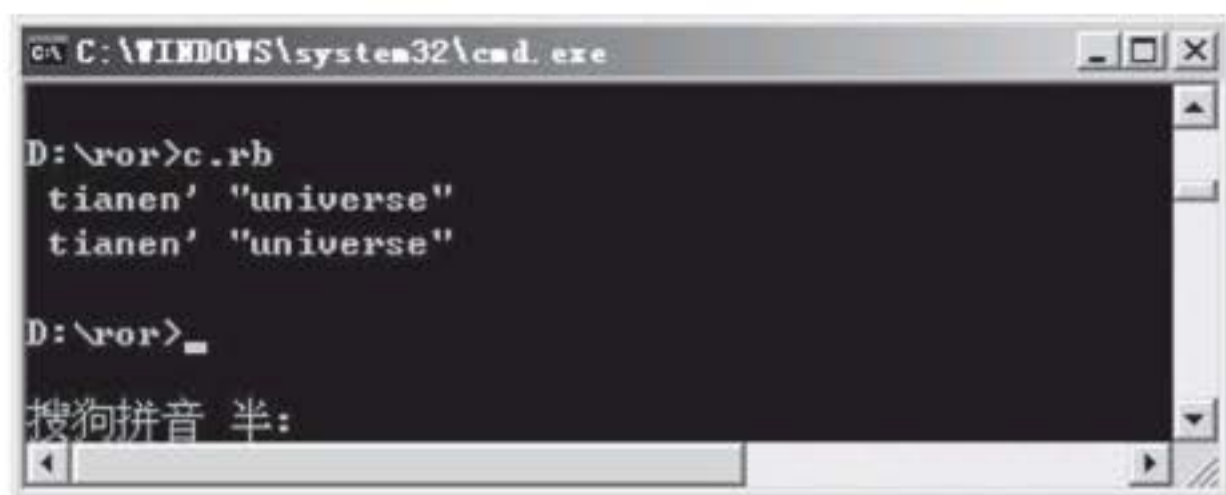


图 2-7 转义字符输出

常用的转义控制字符有“\n”,“\t”等,这是最基本的程序设计常识,读者若不详,可查 ASCII 码表。

程序名称:d.rb

```
puts "tianen\tuniverse\nxiaoyue"
```

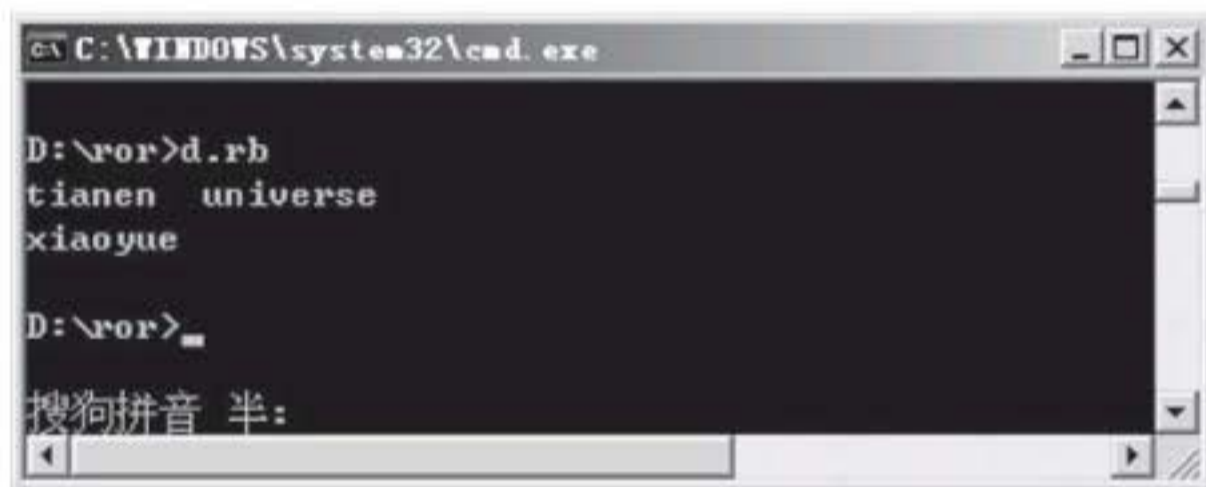


图 2-8 转义控制字符

2.1.4 连句和换行

从前面可以看出,多条 Ruby 语句写在一起时,每条语句占有一行。如果想要将语句连起来写,就需要用分号“;”间隔,分号就是语句连接符,即连句符。

案例名称:连句符

程序名称:e.rb

```
puts "tianen"  
puts "xiaoyue"  
puts "- - - - -"  
puts "tianen";puts "xiaoyue"
```

运行结果如图 2-9 所示。

在所写的各个程序中,可以发现,每条语句输出之后,都自动地输出了一个换行符,从而使得下一条语句在下一行输出。这是因为使用了 puts 方法,如果被输出字符串不是以换行符结尾,这个方法就会在输出的语句后增加一个换行符。如果使用 print 方法,就不会有这个问题。

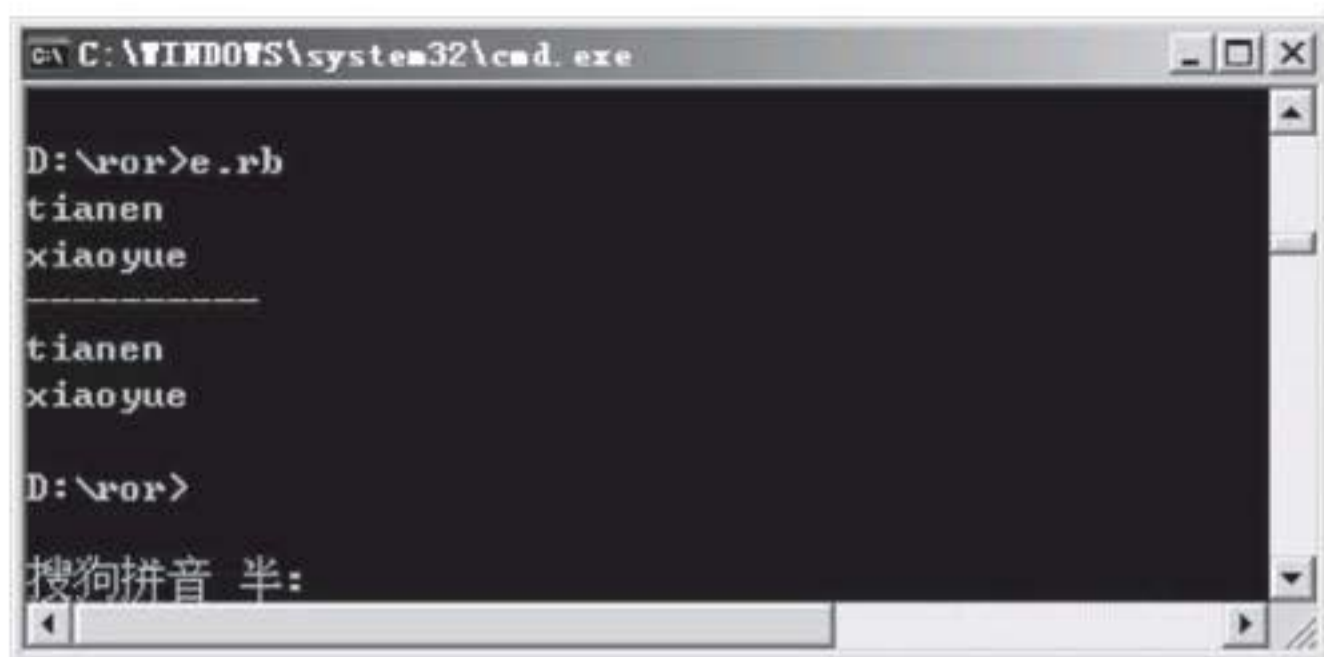


图 2-9 连句符

案例名称:换行符

程序名称:f.rb

```
puts "tianen"
puts "xiaoyue"
print "tianen"
print "xiaoyue"
print "-----"
print "tianen";print "xiaoyue"
```

运行结果如图 2-10 所示。



图 2-10 换行符

同时,还应注意,“如果被输出字符串不是以换行符结尾,puts 方法就会在输出的语句后增加一个换行符”。这意味着,如果被输出字符串以换行符结尾,那么 puts 方法就不会在语句结尾增加换行符。这就是说 puts "tianen\n" 语句只输出一个换行符,而不是两个换行符。要想输出两个换行符,需要用 puts "tianen\n\n"。

案例名称:换行符

程序名称:g.rb

```
puts "xiaoyue\n"
puts "tianen\n\n"
puts "xiaoyue"
```

运行结果如图 2-11 所示。

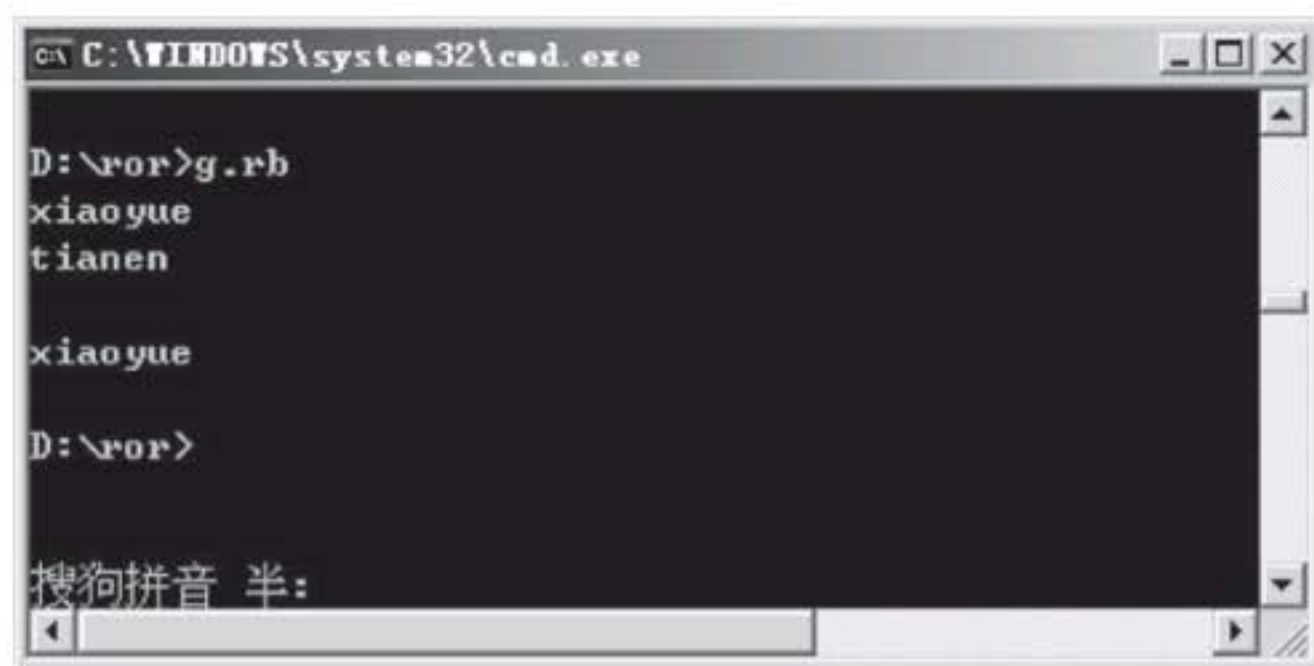


图 2-11 换行符

2.1.5 连行符

连行符是什么概念？

当语句过长的时候，会影响阅读。希望分成多行去写，却不希望语句逻辑被中断。在这个时候，一种方式是使用字符串连接符号来连接语句，但字符串连接符属于运算符，因此字符串连接是消耗资源的。而连行符属于格式控制字符，提倡使用。

Ruby 使用“\”作为连行符。

案例名称：连行符

程序名称：h.rb

```

puts "tianen
xiao yue"
puts "tianen\
xiao yue"

```

运行结果如图 2-12 所示。



图 2-12 连行符

可见，没有使用连行符的语句被当成了两条。而使用了连行符的语句被当成一条执行。

2.1.6 注释符

Ruby 中的注释符包括两种,单行注释和多行注释。单行注释使用“#”符号,多行注释使用“=begin”和“=end”块,注释符案例如下所示。

案例名称:注释符

程序名称:i.rb

```
puts "tianenxiaoyue a"  
# puts "tianenxiaoyue"  
puts "tianenxiaoyue b"  
= begin  
puts "tianenxiaoyue"  
puts "tianenxiaoyue"  
puts "tianenxiaoyue"  
puts "tianenxiaoyue"  
= end  
puts "tianenxiaoyue c"
```

运行结果如图 2-13 所示。



图 2-13 注释符

2.1.7 局部变量

Ruby 中的变量是不能脱离类而存在的,但在谈类的概念之前介绍局部变量却是可以的,不违反循序渐进的原则。局部变量就是“普通变量”,在 Ruby 中由小写字母或下划线开头。局部变量在使用之前如果不初始化,将会出错,局部变量案例如下所示。

案例名称:局部变量

程序名称:j.rb

```
a= "tianen"  
puts a  
b  
puts b  
b= "xiaoyue"  
puts b
```

运行结果如图 2-14 所示。



图 2-14 局部变量

2.1.8 基本输入

Ruby 中使用 `gets` 方法来解决最基本的输入问题。例如：可以用 `a=gets` 来将用户输入的字符保存在局部变量 `a` 中。

注意：Ruby 使用“=”作为赋值符号。基本输入案例如下所示。

案例名称：基本输入

程序名称：`k.rb`

```
a= gets
puts a
```

程序运行，输入一些文字，然后回车。结果如图 2-15 所示。



图 2-15 基本输入

2.1.9 数字和字符串连接

Ruby 支持整型（如：1,0 等）和浮点型（如：1.5,0.618）。数字可以有前缀：0（零）表示八进制（如：013），0x 表示十六进制（如：0x5BC），0b 表示二进制（如：0b1101），案例如下所示。

案例名称：数字

程序名称：`l.rb`

```
puts 0
puts 1
puts 1.5
puts 0.618
```

```
puts 013  
puts 0x5BC  
puts 0b1101
```

程序运行,输出了十进制数值。结果如图 2-16 所示。



图 2-16 数字

Ruby 的字符串使用单引号或双引号括起来。Ruby 使用“+”作为字符串连接符号,案例如下所示。

案例名称:字符串连接

程序名称:m.rb

```
s= "tianen"  
a= "xiaoyue"  
s = s + "&" + a  
puts s
```

程序运行结果如图 2-17 所示。



图 2-17 字符串连接

2.2 类

Ruby 的面向对象比 Java 纯粹,一切都是对象。所以,Ruby 中没有函数,只有方法。Ruby 中所有的变量、常量等都不能脱离类而存在。故而,在介绍了最基本的知识之后,紧接着就要讲类。

本书假设读者已经对面向对象的理论有掌握,所以在这里不再赘述。直入主题,介绍 Ruby 的面向对象之实现。

2.2.1 最简单的类

下面,用 Ruby 写一个最简单的类。这包含如下两方面内容:

①属性

②方法

在 Ruby 中,类名以大写字母开头,实例属性以“@”开头,变量和方法名应该用一个小写字母开头或者用一个下划线开头。

假设要定义的类名为“Man”,包含“name”和“age”两个实例属性、“sayname”方法和“sayage”方法。

定义类和方法使用“def ... end”语句块,如果需要方法返回值,可用“return”语句。类的初始化使用“initialize”方法,相当于 Java 和 C++ 的构造方法,案例如下所示。

案例名称:最简单的类

程序名称:n.rb

```
class Man
  def initialize( name, age )
    @ name = name
    @ age = age
  end
  def sayname
    puts @ name
  end
  def sayage
    puts @ age
  end
end
```

在 Ruby 中,Object 类是所有其他类的父类。

2.2.2 类的基本使用

Ruby 使用 new 方法来构建类的实例,如:m=Man.new。构造了实例之后,就可以使用类的方法。

案例名称:类的基本使用

程序名称:o.rb

```
class Man
  def initialize( name,age )
    @ name = name
    @ age = age
  end
  def sayname
    puts @ name
```

```

end
def sayage
  puts @ age
end
end
m= Man.new("tianen",22)
m.sayname
m.sayage
b= Man.new("bird",99)
b.sayname
b.sayage

```

程序运行结果如图 2-18 所示。



图 2-18 类的基本使用

2.2.3 继 承

Ruby 使用“<”符号来实现继承,这与 Java 中的“extends”及 C++的“:”不同。这里,我们继承前面构建的 Man 类,实现一个 Tianen 类,增加一个 love 方法,其案例如下所示。

案例名称:类的继承增加新方法

程序名称:p.rb

```

class Man
  def initialize( name,age )
    @ name = name
    @ age = age
  end
  def sayname
    puts @ name
  end
  def sayage
    puts @ age
  end
end
end

```

```

class Tianen< Man
  def love
    puts "我爱我妈!"
  end
end
t= Tianen.new("天恩",22)
t.sayname
t.sayage
t.love

```

程序运行结果如图 2-19 所示。



图 2-19 类的继承

在子类中增强父类方法要使用“super”作为关键字,案例如下所示。

案例名程:增强父类中的方法

程序名称:q.rb

```

class Man
  def initialize( name,age )
    @ name = name
    @ age = age
  end
  def sayname
    puts @ name
  end
  def sayage
    puts @ age
  end
end
class T< Man
  def sayname
    super
    puts "我爱我妈!"
  end
end

```

```
t= T.new("天恩",22)
t.sayname
```

程序运行结果如图 2-20 所示。



图 2-20 类的继承

注意:如果要加强的父类方法有参数,那么就用 super(参数)的方式加强。

2.2.4 单态方法

Ruby 是弱类型语言,只有重写,没有重载。所以没办法显示出其多态性。但是,Ruby 具有单态方法。这意味着,从某个共同的类衍生的多个对象可以拥有各自不同的方法,其案例如下所示。

案例名称:单态方法

程序名称:r.rb

```
class Man
  def initialize( name,age )
    @ name = name
    @ age = age
  end
  def sayname
    puts @ name
  end
  def sayage
    puts @ age
  end
end
a = Man.new("a",6)
b = Man.new("b",7)
def b.sayname
  puts "bbbbbb"
end
a.sayname
a.sayage
b.sayname
b.sayage
```

这里,对象 b 重写了 sayname 方法。程序运行结果如图 2-21 所示。



图 2-21 单态方法

2.2.5 方法访问控制

在 Ruby 中,只有方法,没有函数,每一个方法都是存在于类中的。那么,如果不在任何类中定义方法,那么这个方法属于哪个类呢? 答案是:Object 类。理论上讲,这个方法可以被所有的对象使用,但如果真的,这样就乱了。所以,Ruby 将其实现为 Object 类的私有方法,于是,它不能被对象使用。

案例名称:方法访问控制

程序名称:s.rb

```
def my
  puts "my love !"
end
my
s= "good"
s.my
```

程序运行结果如图 2-22 所示。



图 2-22 方法访问控制

Ruby 提供了如下三个关键字来限制对方法的存取,这类似于 Java,C++ 和 C#,其中最类似于 C++。

① public:可以为任何对象所存取的方法(public 是所有方法的默认设置)。

② `protected`:可以在定义它的实例或者子类的实例中调用。

③ `private`:只可以在这个方法所处的对象中被使用,不能直接调用另一个对象的 `private` 方法,包括调用自身也不允许。

这些关键字被插在两个方法之间的代码中。所有从 `private` 关键字开始定义的方法都是私有的,直到代码中出现另一个存取控制关键字为止。

一个类的 `initialize` 方法自动为私有的。

Ruby 和其他面向对象语言的另一个重要的不同点在于,Ruby 动态确定访问控制,在程序运行而不是静止时,且只有运行到那一行,才会去判断是否出错。

程序名称:t.rb

```
class Man
  def initialize( name,age )
    @ name =  name
    @ age =  age
  end
  private
  def sayname
    puts @ name
  end
  def sayage
    puts @ age
  end
  public
  def free
    puts "i am public";
  end
end
t= Man.new("天恩",22)
t.free
t.sayage
t.sayname
```

程序运行结果如图 2-23 所示。

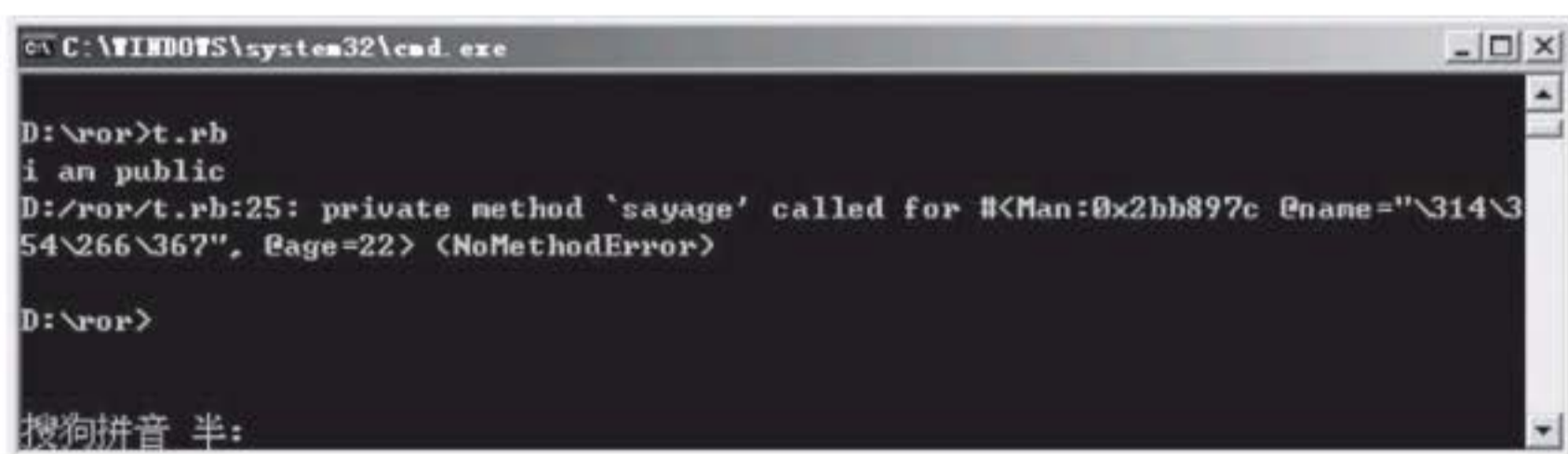


图 2-23 方法访问控制

这里,sayname 和 sayage 方法都是 `private` 类型的,free 方法是 `public` 类型的。除了在方

法之前设置其访问权限之外,还可以在方法定义之后设置,其案例程序如下所示。

程序名称:u.rb

```
class Man
  def initialize( name,age )
    @ name =  name
    @ age =  age
  end
  def sayname
    puts @ name
  end
  def sayage
    puts @ age
  end
  def free
    puts "i am public";
  end
public :free
private :sayname,:sayage
end
t= Man.new("天恩",22)
t.free
t.sayage
t.sayname
```

程序运行结果如图 2-24 所示。

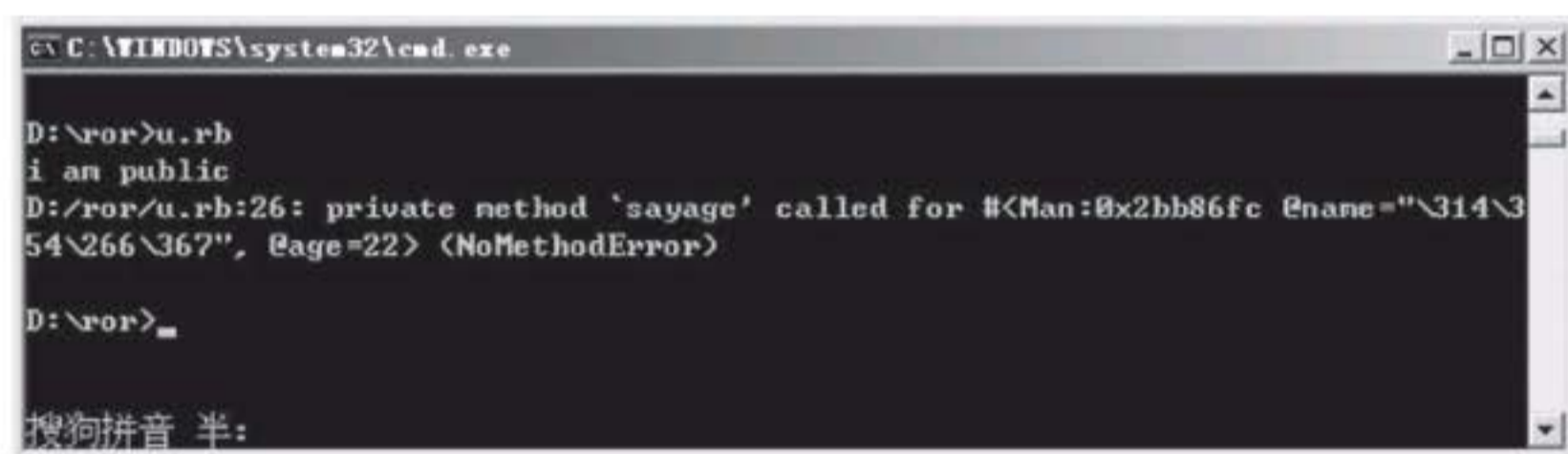


图 2-24 方法访问控制

2.2.6 属性读写控制

类中的属性,包括只读属性,只写属性和读写属性,当然,也有不可读写的属性。在 C# 中对属性的设置很好,是高于 Java 的。Ruby 的设置方法与之相类。

典型的属性读写控制如下面的案例所示,为不同的属性定义 public 方法即可。

案例名称:属性读写控制

程序名称:v.rb

```

class Man
  def initialize( name,sex)
    @ name =  name
    @ age =  16
    @ sex =  sex
  end
  # 只读
  def get_name
    return @ name
  end
  # 只写
  def set_age(age)
    @ age= age
  end
  # 读写
  def get_sex
    return @ sex
  end
  def set_sex(sex)
    @ sex= sex
  end
end
t= Man.new("天恩","male")
puts t.get_name
puts t.get_sex
t.set_sex("man")
puts t.get_sex
t.set_age(22)

```

程序运行结果如图 2-25 所示。



图 2-25 属性读写控制

还有一种简略的方法,用来实现属性的访问控制,典型程序如下所示。

程序名称:w.rb

```

class Man
  def initialize( name,sex)
    @ name =  name

```

```

    @ age = 16
    @ sex = sex
end
# 只读
def name
    @ name
end
# 只写
def age= (age)
    @ age= age
end
# 读写
def sex
    @ sex
end
def sex= (sex)
    @ sex= sex
end
end
t= Man.new("天恩","male")
puts t.name
puts t.sex
t.sex= "man"
puts t.sex
t.age= 22
puts t.age

```

程序运行结果如图 2-26 所示。



图 2-26 属性读写控制

除此之外,还有 attr_系列方法可用来实现属性的读写控制。如下面案例所示。

程序名称:x.rb

```

class Man
  def initialize( name,sex)
    @ name = name
    @ age = 16
    @ sex = sex
  end

```

```

attr_reader:name
attr_writer:age
attr_accessor:sex
= begin
  # 只读
  def name
    @ name
  end
  # 只写
  def age= (age)
    @ age= age
  end
  # 读写
  def sex
    @ sex
  end
  def sex= (sex)
    @ sex= sex
  end
= end
end
t= Man.new("天恩","male")
puts t.name
puts t.sex
t.sex= "man"
puts t.sex
t.age= 22
puts t.age

```

程序运行结果如图 2-27 所示。



图 2-27 属性读写控制

在这里,使用了 attr_reader,attr_writer,attr_accessor 三个符号来设置属性的读写权限。attr_reader 设置只读权限,attr_writer 设置只写权限,attr_accessor 设置读写权限。

此外,还可以用 attr:sex 来设置只读属性,并用 attr:sex,true 来达到和 attr_accessor::sex 相同的效果。这种用法不够简洁,不建议使用。

2.2.7 垃圾收集

Ruby 实现了自动的内存管理,实现了垃圾收集。它没有类似 Java 的 finalize 方法,但它有类似 .NET 的 GC 类用来实现垃圾清理和内存回收。此外,还有 ObjectSpace 类能提供垃圾收集功能,但 ObjectSpace 类提供的垃圾收集功能只是对 GC.start 方法的一个调用。

GC 类具有 disable、enable、start、garbage_collect 4 个方法,其中前两个是关闭和启动垃圾收集器,start 方法和 garbage_collect 意义相同,都是调用 start 方法来实现垃圾收集功能。

关于垃圾收集器的使用,有如下三点需要说明:

① 垃圾收集器回收已经无用的对象的内存空间,从而避免内存泄漏和程序因内存不断膨胀而崩溃;

② 判断一个对象的内存空间是否能被清空的标准是,该对象不再被程序中任何一个地方所引用(一般是指引用计数为 0);

③ 垃圾收集器线程定时轮询,但在系统的可用内存过低时会突然启动释放内存。

案例名称:垃圾收集

程序名称:y.rb

```
s= "cat"  
s= nil  
GC.start
```

程序运行结果如图 2-28 所示。



图 2-28 垃圾收集

通常不需要手动调用 GC 类的方法,但有些时候确实需要迅速释放内存就该调用它。这和在 .NET 中的实现是类似的。相比之下,VC++ .NET 在托管和非托管之间给了人更多的选择。

2.2.8 异常处理

(1) 基本结构

Ruby 使用 begin ... rescue ... end 结构来处理异常,这类似于 Java 中的 try...catch...finally。应用方法是一致的,都是在某个块中监视,找到异常就处理。

下面的例子使用 1 除以 0,当然会出现错误。将其捕捉,并显示出错信息。

注意:这里用的“/”,是除号。

案例名称:异常处理

程序名称:z.rb

```
s= 1
begin
  s= 1/0
  puts s
rescue
  puts "error!"
end
```

程序运行结果如图 2-29 所示。



图 2-29 异常处理

很明显,begin 块中的代码运行被监视,类似 Java 中的 try;rescue 用来捕捉异常,类似 Java 的 catch。

(2) 重试语句

在捕捉到错误之后,可以通过 retry 语句重新执行 begin 块中的内容。如下面案例所示。程序名称:aa.rb

```
s= 1
begin
  s= 1/0
  puts s
rescue
  puts "error!"
  retry
end
```

要注意,这里不停地产生异常,问题没有解决,不停地 retry 下去,导致程序不停地提示错误信息。如图 2-30 所示。



图 2-30 异常处理

(3) 抛出异常

Ruby 中使用 raise 语句抛出异常,类似于 Java 的 throw 语句。异常的信息可以被全局变量 \$! 获得,如下面案例所示。

程序名称:ab.rb

```
begin
  raise "test"
rescue
  puts "error! " + $!
end
```

运行结果如图 2-31 所示。



图 2-31 异常处理

(4) 结束清理

Ruby 中使用 ensure 语句来实现语句块的最后清理。不论语句块中是否出现异常,ensure 块都将执行,这类似于 Java 中的 finally 语句,案例如下所示。

程序名称:ac.rb

```
begin
  s= 1/0
rescue
  puts "error! "
ensure
  puts "finished!"
end
puts "- - - - -"
begin
  s= 1
ensure
  puts "finished!"
end
puts "- - - - -"
begin
  s= 1/0
ensure
  puts "finished!"
end
```

```
end
puts "- - - - -"
```

这个程序含有三段,第一段使用 rescue 捕捉异常,捕捉到异常之后,执行了 ensure 语句。第二段没有发生异常,但使用了 ensure 语句,被执行。第三段,发生异常,没有捕捉,先执行 ensure 语句,然后显示了异常信息。而且,最后的横线没有输出,这是异常所致。程序运行结果如图 2-32 所示。



图 2-32 异常处理

注意:在 begin ... rescue ... end 块中可以只使用 ensure 或 rescue(这类似与 try ... catch 和 try ... finally),但如果同时使用 ensure 和 rescue,那么 ensure 必须在 rescue 后面。(这类似于 try ... catch ... finally)。其实,与 Ruby 的异常处理结构最相似的是 Delphi 和 VB.NET。

2.3 常量和变量

Ruby 有三类变量、一种常量和两种严格意义上的伪变量。变量和常量都没有类型,类似 PHP。

在 Ruby 里我们不需要声明变量,由首字母标识符将常量和变量分类。如表 2-1 所列。

表 2-1 常量和变量分类

首字符	变量类别	首字符	变量类别
\$	全局变量	@	实例变量
[a-z] 或 _	局部变量	[A-Z]	常量
@@	类变量		

注意:Ruby 的伪变量 self 永远指向当前正执行着的对象或未初始化变量的空值 nil。虽然这两者的命名都像是局部变量,但 self 却是个由解释器把持的全局变量,而 nil 实际上是个常量。

变量的命名有以下的限制:

①必须以英文字母(大小写均可)、汉字、下划线开头。

②第二个字符开始可以使用数字、英文字母、汉字、下划线。

③不能使用保留字作变量的名字。

保留字是保留给系统用的,不能使用保留字作为变量名。

Ruby 的保留字如下:

```
alias def false nil return unless and do for not self until begin else if or super when break  
elsif in redo then while case end module rescue true yield class ensure next retry undef
```

2.3.1 常 量

常量由大写字母开头,最多被赋值一次。在 Ruby 的 1.8.6 - 2.5 版本中,常量的再赋值只会产生警告而不产生错误。

案例名称:常量

程序名称:ad.rb

```
Ad= 1  
Ad= 2  
puts Ad
```

程序运行结果如图 2-33 所示。



图 2-33 常量

不论常量是否定义在类中,它都可以在类外被访问。而且,常量还可以定义在模块(后面介绍)中。

2.3.2 全局变量

全局变量由"\$"开头,可以在程序的任何位置访问到。在初始化前,全局变量有一个特殊的值 nil,案例如下所示。

案例名称:全局变量

程序名称:ae.rb

```
$ t  
puts $ t  
$ t= "love"  
puts $ t
```

运行结果如图 2-34 所示。

注意:由于全局变量可以在任何地方被访问,所以使用的时候一定要慎重。



图 2-34 全局变量

现在,列出了一些以"\$"开头的特殊变量。比如,"\$\$"包含了 Ruby 解释器的进程 id,它是只读的。如表 2-2 所列。

表 2-2 特殊变量

变 量	含 义	变 量	含 义
\$!	最近一次的错误信息	\$@	错误产生的位置
\$ _	gets 最近读的字符串	\$.	解释器最近读的行数
\$&	最近一次与正则表达式匹配的字符串	\$~	作为子表达式组的最近一次匹配
\$n	最近匹配的第 n 个子表达式(和\$~[n]一样)	\$=	是否区别大小写的标志
\$/	输入记录分隔符	\$\	输出记录分隔符
\$0	Ruby 脚本的文件名	\$*	命令行参数
\$\$	解释器进程 ID	\$?	最近一次执行的子进程退出状态

举一个例子来说明特殊全局变量的用法。

程序名称:af.rb

```
puts $$
puts $ _
puts $.
```

运行结果如图 2-35 所示。



图 2-35 特殊全局变量

2.3.3 实例变量

在 Ruby 中实变量由"@"开头,它的范围限制在 self 对象内。两个不同的对象即使属于同一个类也可以拥有不同值的实例变量。从对象外部来看,实例变量不能改变甚至观察(比如:Ruby 的实例变量从来不是公用的)除非方法由程序员明确声明。

像全局变量一样,实例变量在初始化前的值是 nil。Ruby 的实例变量也不用声明。实际上,每个实例变量都是在第一次出现时动态加入对象的,案例如下所示。

案例名称:实例变量

程序名称:ag.rb

```
class Test
  def set_t(t)
    @t = t
  end
  def set_n(n)
    @n = n
  end
end
```

2.3.4 局部变量

局部变量由小写字母或下划线"_"开头,局部变量不像全局和实例变量一样在初始化前含 nil 值。前面我们介绍过局部变量,这里就不再介绍了。

2.4 运算符

Ruby 有多种运算符,主要包括:算术运算符,关系运算符,逻辑运算符。以及其他运算符,如:字符串连接符,数组下标运算符等。

2.4.1 算术运算符

Ruby 的算术运算符用来实现“加减乘除”等算术运算,如表 2-3 所列。

表 2-3 算术运算符

符 号	含 义	符 号	含 义
+	加	-	减
*	乘	/	除
%	取余数	++	自加
--	自减	>>	右移
<<	左移	**	乘方
& (只用于整数)	按位与	^ (只用于整数)	按位异或
(只用于整数)	按位或	~ (只用于整数)	按位非

下面,举一个例子来演示算术运算符的使用方法。

案例名称:算术运算符

程序名称:ah. rb

```
a= 1
b= 2
c= a+ b
c= c* * c
puts c
a= 6
b= 4
c= 6% 4
puts c
```

运行结果如图 2-36 所示。



图 2-36 算术运算符

2.4.2 关系运算符

Ruby 的关系运算符用来表达数据之间的“大、小、是否相等”的关系。常用的如表 2-4 所列。

表 2-4 关系运算符

符 号	含 义	符 号	含 义
>	大于	>=	大于等于
<	小于	<=	小于等于
!=	不等于	==	等于
eql?	比较两个对象的值、类型是否相等	equal?	比较两个对象在内存中地址是否相同
<=>	比较两个对象的大小,大于、等于、小于分别返回 1,0,-1	===	判断右边的对象是否在左边区间之内
=~(匹配)	用来比较是否符合一个正则表达式	!~(不匹配)	用来比较是否符合一个正则表达式

下面,举一个例子来演示关系运算符的使用方法。

案例名称:关系运算符

程序名称:ai. rb

```
a= 1
b= 2
puts (a== b)
```

运行结果如图 2-37 所示。



图 2-37 关系运算符

2.4.3 逻辑运算符

Ruby 的逻辑运算符用来实现数据之间的“与、或、非”等运算。如表 2-5 所列。

表 2-5 逻辑运算符

符 号	含 义	符 号	含 义
&&	短路与	and	与
	短路或	or	或
xor	异或	!	非
not	非		

下面,举一个例子来演示逻辑运算符的使用方法。

案例名称:逻辑运算符

程序名称:aj.rb

```
a= true
b= false
puts (a and b)
```

运行结果如图 2-38 所示。



图 2-38 逻辑运算符

2.4.4 其他运算符

其他运算符主要包括赋值运算符和字符串连接符等。常用的如表 2-6 所列。

表 2-6 其他运算符

符 号	含 义	符 号	含 义
+	字符串连接符	defined	检查类型
? :	三元运算符	=	赋值
%=	赋值	~=	赋值
/=	赋值	-=	赋值
+=	赋值	=	赋值
&=	赋值	>>=	赋值
<<=	赋值	*=	赋值
&&=	赋值	=	赋值
* *=	赋值	..	区间起始符
...	区间起始符	[]	数组下标
[]=	数组元素赋值	()	括号,用于运算逻辑划分
.	将对象与它的方法隔开	::	域作用符,将模块(类)与它的常量隔开

举一个三元运算符的例子。

案例名称:三元运算符

程序名称:ak.rb

```
a= ( 1== 2) ? "equal" : "not equal" )
puts a
```

运行结果如图 2-39 所示。



图 2-39 三元运算符

2.5 流程控制

2.5.1 顺序结构

在不受干扰的情况下,程序按照代码书写的先后顺序执行,即:顺序结构,其案例如下所示。

案例名称:顺序结构

程序名称:al.rb

```
a= 1
b= 2
c= 3
d= a+ b* c
puts d
```

运行结果如图 2-40 所示。



图 2-40 顺序结构

2.5.2 选择结构

选择结构,就是先就某条件进行判断,然后根据判断的结果选择所要运行的代码。它包括下面几种形式:

(1) 单重判断

```
if 条件
# 执行代码
end
或
if 条件 then
# 执行代码
end
```

案例名称:单重判断

程序名称:am.php

```
a= 1
b= 2
if b> a then
    puts b
end
if b> a then
    puts a
end
```

运行结果如图 2-41 所示。



图 2-41 单重判断

(2) 双分支

```
if 条件
# 执行代码
else
# 执行代码
end
或
if 条件 then
# 执行代码
else
# 执行代码
end
```

案例名称:双分支

程序名称:an.php

```
a= 1
b= 2
if b> a then
    puts b
else
    puts a
end
if b> a
    puts b
else
    puts a
end
```

运行结果如图 2-42 所示。



图 2-42 双分支

(3) 多分支(if - else 多重选择)

```
if 条件 1
    # 执行代码
elseif 条件 2
    # 执行代码
elseif 条件 3
    # 执行代码
else
    # 执行代码
end
或
if 条件 1 then
    # 执行代码
elseif 条件 2 then
    # 执行代码
elseif 条件 3 then
    # 执行代码
else
    # 执行代码
end
```

案例名称:多分支(if - else 多重选择)

程序名称:ao. php

```
a= 1
b= 2
if b> a then
    puts b
elseif b< a then
    puts a
else
    puts "test"
end
if b> a
    puts b
elseif b< a
    puts a
else
    puts "test"
end
```

运行结果如图 2-43 所示。



图 2-43 多分支 (if - else 多重选择)

(4) 多分支 (case 分支)

```
case 对象
when 条件 1
    # 执行代码
when 条件 2
    # 执行代码
when 条件 3
    # 执行代码
else
    # 执行代码
end
```

案例名称:多分支(case 分支)

程序名称:ap.php

```
a= 1
case a
when 0
    puts 0
when 1
    puts "a> 0"
else
    puts "a< 0"
end
```

运行结果如图 2-44 所示。



图 2-44 多分支 (case 分支)

(5) unless 语句

Ruby 提供了 unless 语句,用于选择条件。它是“if”选择的否定版。其用法如下面的例子所示。

案例名称:unless 语句

程序名称:aq. php

```
a= 1
unless a== 1 then
  puts "good"
else
  puts "next"
end
if a== 1 then
  puts "next"
else
  puts "good"
end
puts "my" unless a== 2
```

运行结果如图 2-45 所示。



图 2-45 unless 语句

2.5.3 循环结构

循环结构就是当满足某条件时,不停地执行某一块代码。它包括下面几种形式:

(1) while 循环

```
while 条件
  # 代码
end
```

案例名称:while 循环

程序名称:ar. php

```
i= 1
while i<= 3
```

```
puts i
i+ = 1
end
```

运行结果如图 2-46 所示。



图 2-46 while 循环

(2) 单行 while 循环

单条语句 while 条件

案例名称:单行 while 循环

程序名称:as. php

```
i= 1
(puts i;i+ = 1) while i< = 3
```

运行结果如图 2-47 所示。



图 2-47 单行 while 循环

(3) until 循环

```
until 条件
# 代码
end
```

案例名称:until 循环

程序名称:at. php

```
i= 1
until i== 4
    puts i
    i+= 1
end
```

运行结果如图 2-48 所示。



图 2-48 until 循环

(4) for...in 循环

```
for 变量 in 集合对象
    # 代码
end
```

这个集合对象可以是数组或者区间(后面介绍)等,读者可以从下面的例子中看到其基本用法,更详细内容将在下一章介绍区间和数组时讲解。

案例名称:for...in 循环

程序名称:au. php

```
i= 1
for i in 1..3
    puts i
end
```

运行结果如图 2-49 所示。



图 2-49 for...in 循环

(5) 循环控制

Ruby 具有类似 C 的 break 语句以及其所专有的特殊循环控制语句。

在循环体内,如果遇到:

break,跳出当前循环;

next,忽略本次循环的剩余部分,开始下一次的循环,相当于 C 中的 continue;

redo,不检查循环条件,重新开始当前循环;

retry,从头开始重复这个循环体。

案例名称:循环控制

程序名称:av.php

```
for i in 1..6
  puts i
end
puts "- - - - -"
for i in 1..6
  if i == 3 then
    break
  end
  puts i
end
```

运行结果如图 2-50 所示。



图 2-50 循环控制

2.6 块

块(block)是 Ruby 的一个特性,它是一种可以和方法调用相关联的代码块,如同参数。有了块,我们可以减少许多方法,使编程变简单。

2.6.1 块的概念

块是在大括号或者 do ... end 之间的一组代码。

如：

```
{ puts "tianen" }  
或  
do  
  # 代码  
end
```

创建的块可以与方法的调用相关联。把块的开始放在含有方法调用的源码行的结尾处，就可以实现关联。比如，在后面的代码中，含有 puts "Tianen" 的块与 my 方法的调用相关联。

```
my { puts "Tianen" }
```

如果方法有参数，它们应出现在块之前。

```
my("love") { puts "Tianen" }
```

2.6.2 块的基本使用

使用 Ruby 的 yield 语句方法可以一次或多次地调用相关联的块。如果某个方法含有 yield 语句并且关联了块，那么执行到 yield 语句的时候，就会将其所关联的块进行调用。如下面例子所示。

案例名称：块的基本使用

程序名称：aw.rb

```
def test_block  
  puts "Start"  
  yield  
  yield  
  puts "End"  
end  
test_block { puts "test block" }
```

运行结果如图 2-51 所示。



图 2-51 块的基本使用

程序名称：ax.rb

```
def test_block  
  puts "Start"  
  yield
```

```

yield
puts "End"
end
test_block { puts "test block" }
test_block do
  puts "test block 1"
  puts "test block 2"
end
= begin
# error
test_block
do
  puts "test block 1"
  puts "test block 2"
end
= end

```

运行结果如图 2-52 所示。



图 2-52 块的基本使用

2.6.3 带参数的块

使用 Ruby 的 `yield` 语句方法可以一次或多次地调用相关联的块,还可以向块传递参数。带参数的块如下所示。

```
{ |a,b| puts a + b }
```

参数被两条竖线包围,多个参数之间使用逗号分隔。

案例名称:带参数的块

程序名称:ay.rb

```

def test_block
  puts "Start"
  yield(1,2)
  yield(3,4)
  puts "End"
end

```

```
end
test_block { |a,b| puts a + b }
```

运行结果如图 2-53 所示。



图 2-53 带参数的块

2.7 迭代器

迭代器是从某种集合对象(如:数组)中连续返回元素的方法。在 Ruby 库中大量使用了块来实现迭代器。

这一节要介绍的概念,会提到数组和区间数据类型。这概念是很好理解的,这里简要提一下,后面的章节会详细讲述。

Ruby 中的数组如: `a = ['m', 'eee', 'ca']`, 引用方式如: `a[0]`。

区间就是范围,如: `a=1..3`, 表示 1 和 3 之间这个范围,而且包括 1 和 3 两个端点。数组和区间都属于集合对象,因为它们可以表示一组数据。

2.7.1 迭代的概念

迭代就是重复地执行某一块代码,类似循环,但比循环简洁。下面举一个例子,从中可以看出迭代的含义。

案例名称:迭代的概念

程序名称:az.rb

```
# 单行循环块
# 重复三次
3.times{ puts "I am Genius!" }
puts "- - - - -"
# 从 1 到 9 的循环
1.upto(9){ |i| print i if i < 6 }
puts "\n- - - - -"
1.upto(9) do |x|
  print x, " "
end
puts "\n- - - - -"
# 从 9 到 1 的循环
```

```

9.downto(1){|i| print i if i< 6 }
puts "\n- - - - - "
# 一个从 0 到 12,步长为 3 的循环
0.step(12,3) {|i| print i }
puts "\n- - - - - "
# 取出区间中的元素
(1..9).each {|i| print i if i< 6}
puts "\n- - - - - "
[ 1, 1, 2, 3, 5 ].each {|val| print val, " " }

```

运行结果如图 2-54 所示。



图 2-54 迭代的概念

这个例子中展示了常用的几个迭代器,读者应该仔细体会。

2.7.2 编写迭代器

我们可以自编迭代器来实现简化编程。yield 常常会在一个迭代器的定义中出现。yield 将流程控制移至传递给迭代器的代码块。下面的例子定义了一个最简单的 repeat 迭代器,会依参数的设置执行多次代码块,案例如下所示。

案例名称:编写迭代器

程序名称:ba.rb

```

def repeat(num)
  while num > 0
    yield
    num -= 1
  end
end
repeat(3) { puts "test" }

```

运行结果如图 2-55 所示。

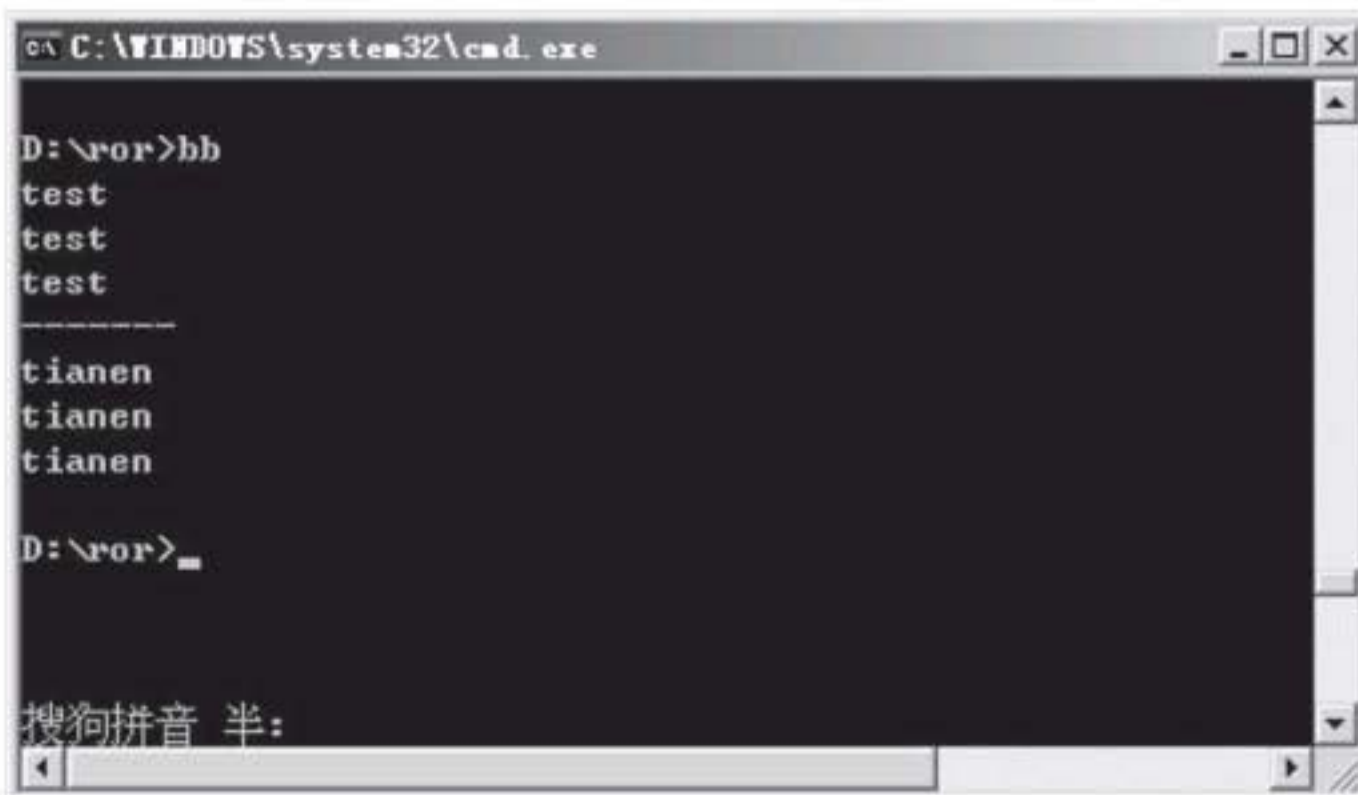


图 2-55 编写迭代器

在这个例子的基础上,我们可以编写出稍微复杂一点的迭代器,给块传递参数。
程序名称:bb, rb

```
def repeat(num)
  while num > 0
    yield
    num -= 1
  end
end
repeat(3) { puts "test" }
puts "- - - - -"
def rep(num,s)
  while num > 0
    yield(s)
    num -= 1
  end
end
rep(3,"tianen") { |s| puts s }
```

运行结果如图 2-56 所示。



当我们定义一个新的数据类型时,为它定义一个合适的迭代器是很好的事情。这意味着我们将迭代器定义在某一个特定的自创类中(前面是 Object),案例如下所示。

程序名称:bc.rb

```
class My
  def initialize(name,age)
    @name = name
    @age = age
  end
  def sayname
    puts @name
  end
  def sayage
    puts @age
  end
  def repeat(num)
    while num > 0
      yield
      num -= 1
    end
  end
end

m= My.new("tianen",22)
m.repeat(3){ puts "test" }
```

运行结果如图 2-57 所示。



图 2-57 编写迭代器

2.8 过程对象

把一块代码作为参数向某个方法传递,这件事情只有 Ruby 做得到。虽然不难写出自己的语言来实现这项功能,但,Java 没有这样做,Ruby 这样做了。这样做的目的是:快,开发快。可以把一些代码作为一个过程对象,从而使其成为一个整体。

2.8.1 创建过程对象

创建过程对象,需要使用 `proc` 关键字,其方法如下面案例所示,要注意大括号的位置。创建好的过程对象可以使用,通过 `call` 方法就能使用这个对象。

案例名称:创建过程对象

程序名称:bd.rb

```
# wrong
# t = proc
# {
#   puts "tianen"
# }
t = proc{
  puts "tianen"
}
t.call
```

运行结果如图 2-58 所示。



图 2-58 创建过程对象

2.8.2 把过程对象当作参数

过程对象可以当作参数传递给某个函数,案例如下所示。

案例名称:把过程对象当作参数

程序名称:be.rb

```
def test(p)
  puts "start"
  p.call
  puts "end"
end

t = proc{
  puts "tianen"
}

test(t)

test(proc{puts "xiaoyue"})
```

运行结果如图 2 - 59 所示。



图 2 - 59 把过程对象当作参数

小 结

本章介绍了 Ruby 的语言基础。包括 Ruby 的常量、变量、运算符、流程控制、面向对象特性、块、迭代器、过程对象等内容,这些内容是 Ruby 语言的最基本内容,需要牢固掌握。

思考和练习

自行编写程序,熟练掌握本章的知识。

第 3 章 Ruby 的数据类型

本章要点

Ruby 有多种数据类型,如:数字、数组、字符串、区间、时间日期、散列表、块及正则表达式等。前一章中只简要介绍了字符串和数字类型,本章将详述各种数据类型的用法。

3.1 数 字

Ruby 支持整型和浮点型两种数字类型。

3.1.1 数字的基本使用

整型可以是任意长度(最大值由机器的内存大小决定)。在一定范围内(通常是一230~230-1 或-262~262-1)在内部由二进制方式表示,内部类为 Fixnum。大小超过这个范围的整数由 Bignum 表示,如果 Fixnum 计算之后结果超出范围,自动转换为 Bignum。Ruby 在两者之间自动转换,对用户来说是透明的(就是用户不知道 Ruby 做了些什么)。

因为在 Ruby 中一切都是对象,所以整数和浮点数都是按类来定义的。Numeric 是所有数字类型的基类,Float 和 Integer 类是 Numeric 的子类。Fixnum 和 Bignum 都是 Integer 的子类型,它们分别定义了“常规大小”的整数和大型整数。

下面的例子说明了数字和其类的关系。

案例名称:数字的基本使用

程序名称:a.rb

```
puts 16777216.class  
puts 281474976710656.class
```

程序运行结果如图 3-1 所示。



图 3-1 数字的基本使用

可以在使用整型的时候在前面使用进制标示符,比如 0 表示八进制,0x 表示十六进制,0b 表示二进制等。而且,如果一个整型数字中有一个下划线,这个下划线将被忽略。

程序名称:b.rb

```
puts 123456
puts 123_45
puts - 543
puts 123_456_789_123_345_789
puts 0xaab
puts 036
puts - 0b101_01
```

程序运行结果如图 3-2 所示。



图 3-2 数字的基本使用

整型可以是任意长度(最大值由机器的内存大小决定)。在一定范围内(通常是 $-2^{30} \sim 2^{30}-1$ 或 $-2^{62} \sim 2^{62}-1$)通过在一个 ASCII 字符或者一个转义字符前面加一个问号得到它的数字值。Control 和 Meta 键的组合也可以用 $?\backslash C-x$, $?\backslash M-x$ 和 $?\backslash M-\backslash C-x$ 表达。

程序名称:c.rb

```
puts ? \a # 字符的数字值
puts ? \n # 换行符的值
puts ? \t
puts ? \C- x # control x
puts ? \M- x
puts ? \M- \C- x
puts ? \C- ? # 删除字符
```

程序运行结果如图 3-3 所示。



图 3-3 数字的其他用法

3.1.2 数字的常用方法

Numeric 类具有如下常用方法,这些方法为一切数字所共有。

- ① abs: 返回数字的绝对值。
- ② ceil: 返回一个等于或大于数字的最小的整数。
- ③ div(other): 返回数字除以 other 的整数商。
- ④ modulo(other): 返回数字除以 other 的余数。
- ⑤ floor: 返回一个不超过该数字的最大的整数。
- ⑥ integer?: 若数字为整数则返回真。
- ⑦ nonzero?: 若数字为 0 则返回 nil, 非 0 则返回自身。
- ⑧ remainder(other): 返回数字除以 other 的余数, 但是余数的符号与 self 相同(或为 0)。
- ⑨ round: 返回最接近该数字的整数。
- ⑩ truncate: 舍弃小数点后面的部分。
- ⑪ zero?: 若为 0 则返回真。
- ⑫ to_s: 将数字转换成字符串。

Integer 类具有如下常用方法,这些方法为一切整数所共有。

- ① 类方法 Integer.induced_from(num): 将 num 变为 Integer 类型并返回变换结果。
- ② chr: 返回与字符代码相对应的 1 个字节的字符串。例如 65.chr 返回 "A"。相反地,若想得到某字符串的字符代码时,可以使用 "A"[0], 整数必需在 0~255 之间,若超出该范围将引发 RangeError 异常。

③ to_f: 将数值变为浮点数(Float)。

④ to_s, to_s(base): 将整数变为 10 进制字符串形式。

若使用了参数,则把整数变为以参数为基数的字符串形式。基数只能是 2~36 之间的数,若超出范围则引发 ArgumentError 异常。

Float 类具有如下常用方法,这些方法为一切浮点数所共有。

- ① 类方法 Float.induced_from(num): 将 num 变换为 Float 并返回其结果。
- ② finite?: 若某数值既非 ∞ 又非 NaN 则返回真。

③ `infinite?`:若某数值为 $+\infty$ 则返回 1,若为 $-\infty$ 则返回 -1,除此以外返回 nil. 浮点数除以 0 得 ∞ 。

④ `nan?`:当某数值为 NaN(Not a number)时返回真,浮点数的 0 除以 0 得 NaN。

⑤ `to_i`:删除某数的小数部分后将其变为整数。

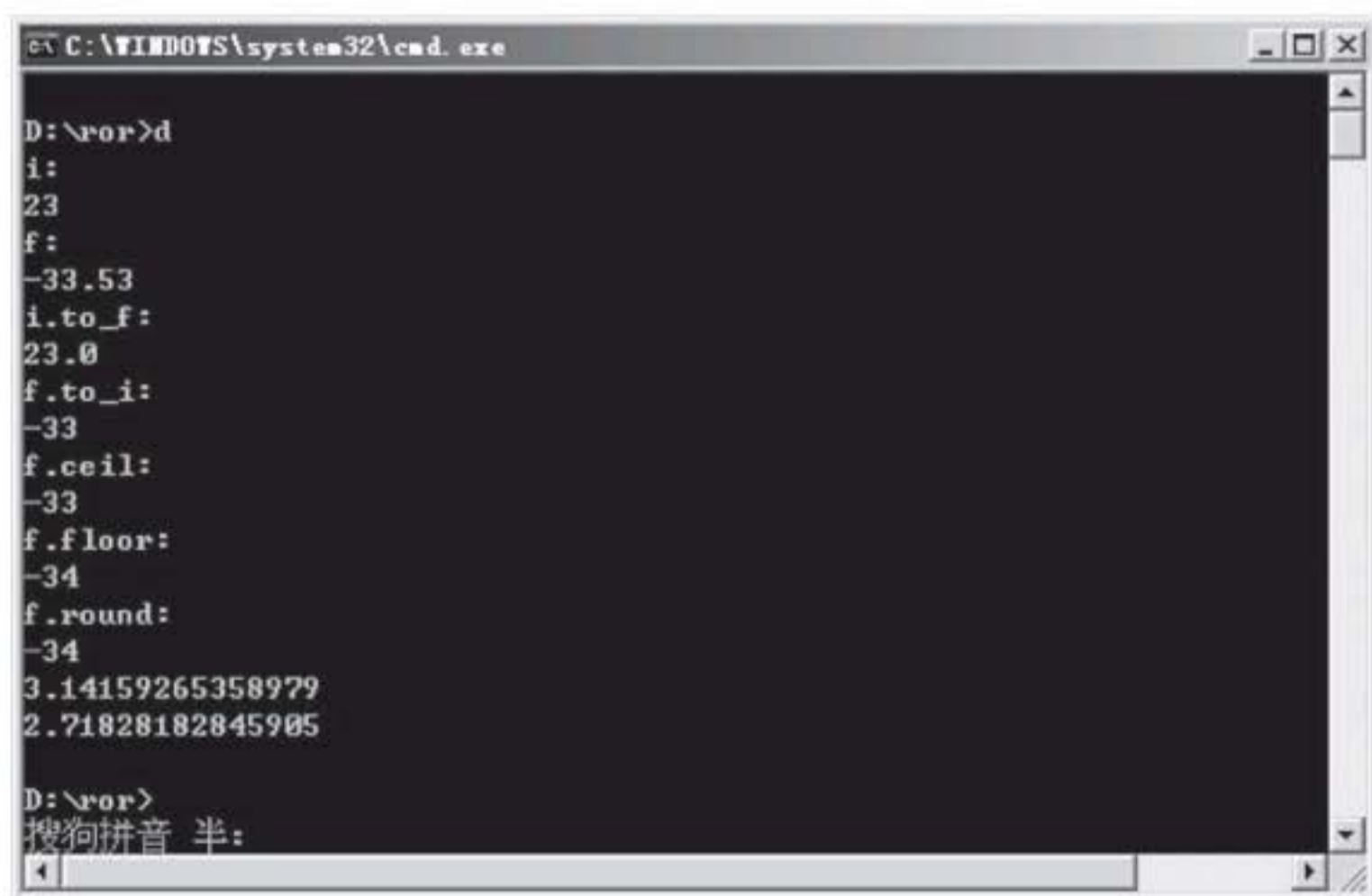
下面举一个例子,演示数字的使用方法。

案例名称:数字的常用方法

程序名称:d.rb

```
i= 23
f= - 33.53
puts "i:",i
puts "f:",f
puts "i.to_f:",i.to_f
puts "f.to_i:",f.to_i
puts "f.ceil:",f.ceil
puts "f.floor:",f.floor
puts "f.round:",f.round
p Math::PI
p Math::E
```

程序运行结果如图 3-4 所示。



```
C:\WINDOWS\system32\cmd.exe
D:\ror>d
i:
23
f:
-33.53
i.to_f:
23.0
f.to_i:
-33
f.ceil:
-33
f.floor:
-34
f.round:
-34
3.14159265358979
2.71828182845905
D:\ror>
搜狗拼音 半:
```

图 3-4 数字的常用方法

3.1.3 数学计算方法

数字常用来做数学计算,在 Ruby 中,进行数学计算的函数集中在 Math 模块(模块的概念将在下一章介绍,这里只需认为模块和类的使用方法相似即可)中。

在 Math 模块中,定义了两套内容相同的方法和特殊方法,因此,既可以使用模块的特殊

方法,又可以将模块包含到类中之后,使用它的普通方法。

Math 模块的主要方法如下。

① 三角函数: `Math.acos(x)`; `Math.asin(x)`; `Math.atan(x)`。

以弧度返回 x 的反三角函数值,返回值的范围分别是 $[0, +\pi]$ 、 $[-\pi/2, +\pi/2]$ 、 $(-\pi/2, +\pi/2)$ 。在 `acos(x)`, `asin(x)` 中, x 的取值范围必为 $-1.0 \leq x \leq 1$ 。(通常返回 NaN)。`acos(π)`, `asin(π)` 遇到超出范围的参数时,会引发 `Errno::EDOM` 异常。

`Math.atan2(y, x)`

返回 $[-\pi, \pi]$ 之间的 y/x 的反正切值。

`Math.acosh(x)`, `Math.asinh(x)`, `Math.atanh(x)` 返回 x 的反双曲线函数值。

$\text{asinh}(x) = \log(x + \sqrt{x * x + 1})$

$\text{acosh}(x) = \log(x + \sqrt{x * x - 1})$ [$x \geq 1$]

$\text{atanh}(x) = \log((1+x)/(1-x)) / 2$ [$-1 < x < 1$]

在 `acosh(x)` 中, x 的取值范围是 $x \geq 1$ 。

在 `atanh(x)` 中, x 的取值范围是 $-1.0 < x < 1$ 。

`Math.cos(x)`

`Math.sin(x)`

`Math.tan(x)`

返回 $[-1, 1]$ 之间的以弧度表示的 x 的三角函数值。

`Math.cosh(x)`, `Math.sinh(x)`, `Math.tanh(x)` 返回 x 的双曲线函数的值。

```
cosh(x) = (exp(x) + exp(-x)) / 2
```

```
sinh(x) = (exp(x) - exp(-x)) / 2
```

```
tanh(x) = sinh(x) / cosh(x)
```

② 误差函数: `Math.erf(x)`, `Math.erfc(x)` 返回 x 的误差函数(erf)、补余误差函数(erfc)的值。

③ 指数: `Math.exp(x)` 返回 x 的指数函数的值。

`Math.frexp(x)` 返回实数 x 的指数部分和假数部分。

④ 平方和: `Math.hypot(x, y)` 返回 $\sqrt{x * x + y * y}$ 。

⑤ 指数: `Math.ldexp(x, exp)` 先以 2 为底数进行 `exp` 的指数运算,然后乘以实数 x ,返回计算结果。

⑥ 对数: `Math.log(x)` 返回 x 的自然对数, x 必为正数(通常情况下,若为负值则返回 NaN,若为 0 则返回 `-Infinity`),遇到超出范围的参数时,若为负数则引发 `Errno::EDOM` 异常,若为 0 则引发 `Errno::ERANGE` 异常。

`Math.log10(x)` 返回 x 的常用对数, x 必为正数(通常情况下,若为负值则返回 NaN,若为 0 则返回 `-Infinity`),遇到超出范围的参数时,若为负数则引发 `Errno::EDOM` 异常,若为 0 则引发 `Errno::ERANGE` 异常。

⑦ 平方根: `Math.sqrt(x)` 返回 x 的平方根. 若 x 为负值,则引发 `ArgumentError` 异常,通常情况下,若 x 为负值,则引发 `Errno::EDOM` 异常。

⑧ 常数: `E` 自然对数的底数; `PI` 圆周率。

3.2 字符串

Ruby 的字符串是简单的 8 位字节序列。它们通常保存可打印字符序列,但也可以保存二进制数据。字符串是 String 类的对象。

字符串的应用非常广泛,因此这一节将会进行详细的介绍。

3.2.1 字符串的基本用法

在前一章已经介绍过字符串的概念,包括转义字符的使用。Ruby 的字符串处理功能是强大的。可以用 `# {表达式}` 来把任何的 Ruby 表达式的值插入到字符串中。如果那个表达式是全局变量,类变量或者实例变量,就可以省略大括号。这与 MFC 的 CString 类相似。

案例名称:字符串的基本用法

程序名称:e.rb

```
a= "my age : # {11+ 11}"  
b= "line NO. # $ ."  
puts a  
puts b
```

程序运行结果如图 3-5 所示。



图 3-5 字符串的基本用法

此外,还可以用 `%q`, `%Q` 来构造字符串。这方法在构建长字符串的时候比较有用。`%q` 和 `%Q` 用来界定单引号和双引号的范围,即字符串的起止点。跟在 `'q'` 或者 `'Q'` 后面的字符是分隔符,如果那个字符是括号,大括号,圆括号或者小于等于符号,那么程序会一直向下读直到遇见最近的停止符号,或者到匹配到相应的符号才停止,然后把读入的字符作为一个字符串整体。

程序名称:f.rb

```
a= % q/i am yu tian en/  
b= % Q! i am yu tian en!  
c= % Q(i am yu tian en)  
puts a  
puts b  
puts c
```

程序运行结果如图 3-6 所示。



图 3-6 字符串的基本用法

3.2.2 字符串的常用方法

String 类含有大量标准方法,这里仅介绍最常用的方法。在该类的众多方法中,那些方法名尾部是!的方法将会直接修改字符串的内容。所以,使用不带!的方法是比较安全的。

(1) * 符号

将字符串的内容重复指定次数之后,返回新字符串,案例如下所示。

案例名称:字符串的常用方法

程序名称:g.rb

```
a = "tianen"
a = a * 3
puts a
```

程序运行结果如图 3-7 所示。



图 3-7 * 符号

(2) []符号

对于字符串 a,a[nth]以整数形式(字符代码)返回第 nth 字节的内容,若 nth 为负值,则从字符串尾部算起,若 nth 超出范围则返回 nil。

a[nth, len]返回从第 nth 字节算起的长度为 len 字节的子字符串。若 nth 为负数则从字符串尾部算起。若 nth 超出范围则返回 nil。

程序名称:h.rb

```
a = "tianen"
puts a[0]
```

```
puts a[- 1]
puts a[- 6]
puts a[5]
puts a[6]
puts "- - - - -"
puts a[0].chr
puts a[- 1].chr
puts a[- 6].chr
puts a[5].chr
puts "- - - - -"
a = "tianen"
puts a[0,2]
puts a[- 1,3]
```

程序运行结果如图 3-8 所示。

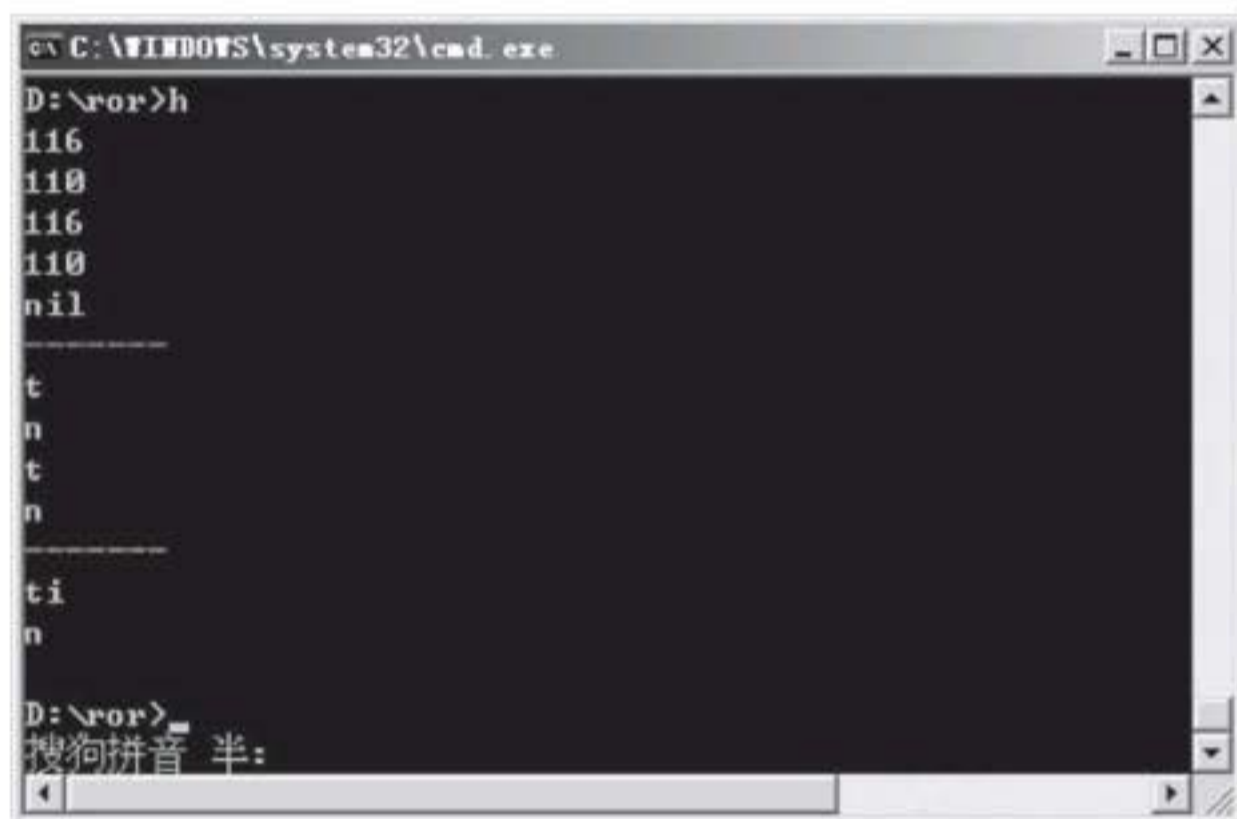


图 3-8 []符号

(3) next, next!, succ, 和 succ!

next, next!, succ 和 succ! 这四个方法用来返回下一个字符串。“下一个”是指：按照 26 个字母顺序或 10 进制数的顺序继续向下数时得到的结果。这里不考虑负号的问题。若字符串中包含字母或数字的话，则其他字符将保持不变，相反地，若不包含字母或数字的话，就返回下一个 ASCII 字符。””。succ 会返回 ””。如果遇到多字节字符串的话，则只把它当做普通的字节串来处理。另外，没有与 succ 动作相反的方法。

注意：succ! 和 next! 会强行修改字符串的内容。

程序名称：i.rb

```
p "aa".succ
p "99".succ
p "a9".succ
p "Az".succ
p "zz".succ
p "- 9".succ
```

```

p "9".succ
p "09".succ
p "1.9.9".succ
p "/" .succ
p "\0".succ
p "\377".succ

```

程序运行结果如图 3-9 所示。

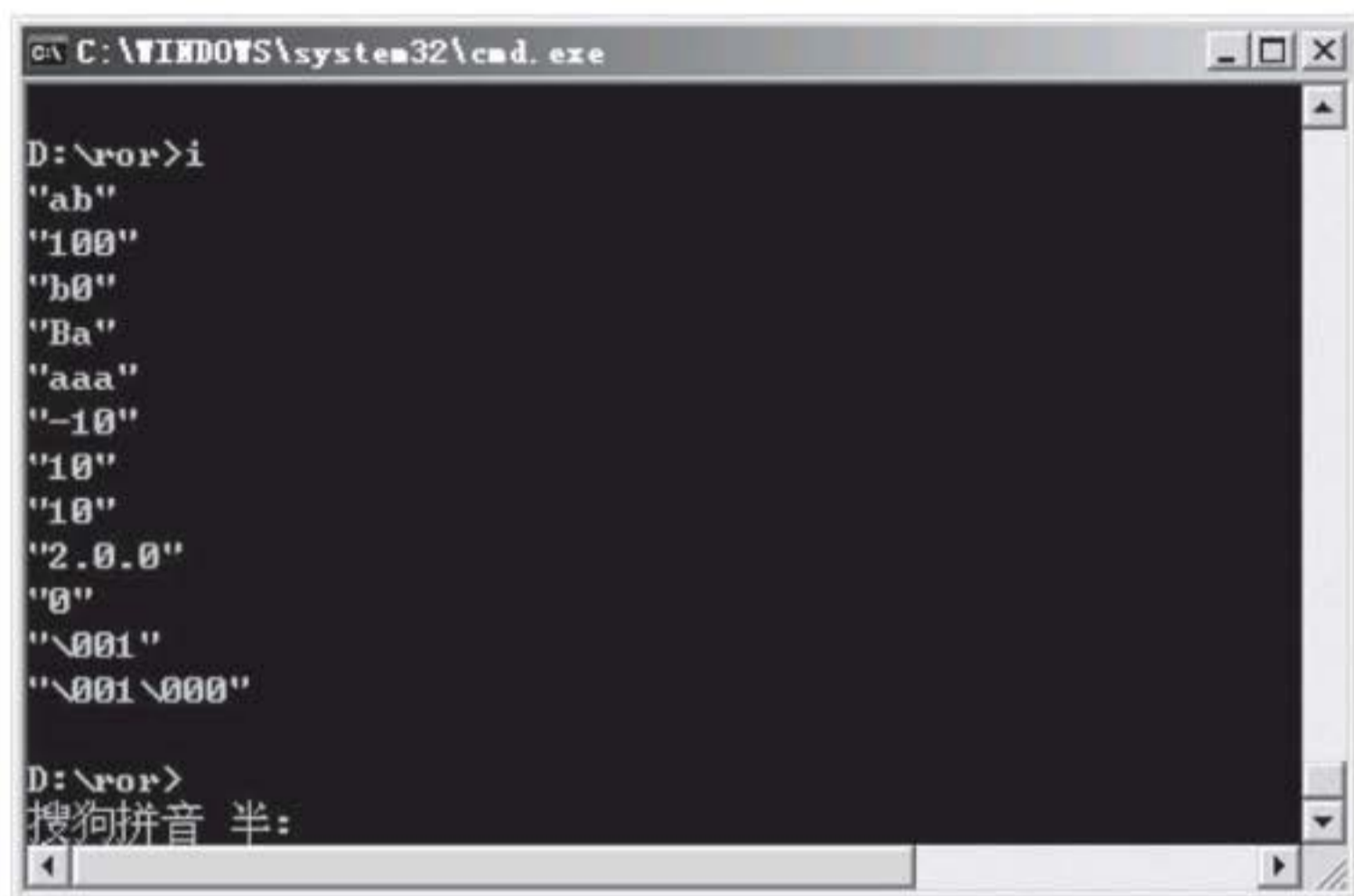


图 3-9 后续方法

(4) [substr]符号

若字符串当中包含 substr, 则生成并返回一致的字符串。若不包含 substr 的话, 就返回 nil, 案例如下所示。

程序名称:j. rb

```

a = "tianen"
puts a["an"]
puts a["na"]

```

程序运行结果如图 3-10 所示。



图 3-10 [substr]符号

(5) [regexp]和[regexp, nth]符号

[regexp]和[regexp, nth]符号用来完成正则表达式匹配,与正则表达式相关的内容在下一节介绍。

返回最初与 regexp 相匹配的子字符串。与匹配操作相关的信息被存入内部变量 \$ ~ 中。若与 regexp 的匹配失败则返回 nil。

若使用了 nth 参数,则返回最初那个与 regexp 中第 nth 个括号相匹配的子字符串。若 nth 为 0,则返回整个匹配的字符串。若匹配失败或没有与 nth 相对应的括号时,返回 nil,案例如下所示

程序名称:k.rb

```
a = "tianen"  
puts a[/an/]  
puts $ ~ .begin(0)
```

程序运行结果如图 3-11 所示。



图 3-11 [regexp]符号

(6) length 和 size

length 和 size 用来返回字符串的字节数。

程序名称:l.rb

```
a = "tianen"  
puts a.length  
puts a.size  
puts "天恩".length  
puts "天恩".size
```

程序运行结果如图 3-12 所示。



图 3-12 length 和 size

(7) [first..last]符号

[first..last]用来生成并返回一个包含从索引 first 到索引 last 之间所有内容的字符串。当 last 超过字符串长度时,就认为它的值等于(字符串长度-1)。当 first 小于 0 或大于字符串长度,或者 first > last + 1 时返回 nil。但是,若 first 和 last 中的一个或者两个都为负数时,将会补足字符串的长度然后再次执行。

程序名称:m.rb

```
a = "tianen"  
puts a[2..4]  
puts a[2..5]  
puts a[2..6]  
puts a[-2..5]
```

程序运行结果如图 3-13 所示。



图 3-13 [first..last]符号

(8) [first...last]符号

将字符串的头部看作序号为 0 的缝隙,将字符串的末尾看作序号为字符串.length 个缝隙,然后生成并返回一个包含从第 first 个缝隙到第 last 个缝隙之间的所有内容的字符串。当 last 大于字符串长度时,就认为它的值等于字符串的长度。

当 first 小于 0 或大于字符串长度,或者 first > last 时返回 nil。但是,若 first 和 last 中的一个或者两个都为负数时,将会补足字符串的长度然后再次执行。

程序名称:n.rb

```
a = "tianen"  
puts a[2...4]  
puts a[2..4]
```

程序运行结果如图 3-14 所示。



图 3-14 [first...last]符号

(9) [nth]=val

以 val 来替换第 nth 字节的内容。若 val 是 0~255 之间的整数时,就把它看作是字符代码,并以该代码所对应的字符来进行替换操作,返回 val,案例如下所示。

程序名称:o.rb

```
a = "tianen"
puts a
puts a[2]
puts a[2]= "ttttt"
puts a
```

程序运行结果如图 3-15 所示。



图 3-15 [nth]=val

(10) [nth, len]=val

以 val 来替换从第 nth 字节起长度为 len 字节的子字符串。若 nth 为负数则从尾部算起,返回 val。

程序名称:p.rb

```
a = "tianen"
puts a
puts a[2]
puts a[2,4]= "ttttt"
puts a
```

程序运行结果如图 3-16 所示。



图 3-16 [nth, len]=val

(11) [substr]=val

以 val 来替换字符串中第一个与 substr 相对应的子字符串。若 self 中不包含 substr 时，将引发 IndexError 异常。返回 val，案例如下所示。

程序名称:q.rb

```
a = "tianen"
puts a
puts a["an"]
puts a["an"] = "ttttt"
puts a
```

程序运行结果如图 3-17 所示。



图 3-17 [substr]=val

(12) [regexp]=val 和 [regexp, nth]=val

以 val 来替换第一个与正则表达式 regexp 相匹配的子字符串。若使用了参数 nth，则以 val 替换第一个与正则表达式 regexp 中的第 nth 个括号相匹配的子字符串。若 nth 为 0 时，则以 val 来替换整个匹配部分。若正则表达式的匹配失败则引发 IndexError 异常。返回 val，案例如下所示。

程序名称:r.rb

```

a = "tianen tianen"
puts a
puts a[/an/]
puts a[/an/,0]= "ttttt"
puts a

```

程序运行结果如图 3-18 所示。



图 3-18 [regexp]=val 和 [regexp, nth]=val

(13) [first..last]=val 和 [first...last]=val

以 val 来替换从 first 到 last 之间的内容。返回 val。

程序名称:s.rb

```

a = "tianen tianen"
puts a[2..5]
puts a[2..5]= "AAAAA"
puts a

```

程序运行结果如图 3-19 所示。



图 3-19 [first..last]=val 和 [first...last]=val

(14) capitalize 和 capitalize!

将首字符(若为字母的话)改为大写字母,其余的改为小写字母。

capitalize 生成并返回修改后的字符串。而 capitalize! 会修改字符串本身并返回结果,若

未作修改时返回 nil。

若没有正确设置 \$KCODE, 部分汉字代码也会被修改(在 ShiftJIS 编码中就会发生这种情况)相反, 即使设置了 \$KCODE, 也不会修改多字节字符的字母。案例如下所示。

程序名称: t.rb

```
a = "tianen"
puts a
puts a.capitalize
puts a
puts a.capitalize!
puts a
a = "t 天恩"
puts a
puts a.capitalize
```

程序运行结果如图 3-20 所示。



图 3-20 capitalize 和 capitalize!

(15) 若干格式化方法

```
center(width)
ljust(width)
rjust(width)
center(width[, padding])
ljust(width[, padding])
rjust(width[, padding])
```

分别返回居中、靠左、靠右的字符串, 当字符串长度超过 width 时, 将返回原字符串的拷贝。若使用了第二参数 padding 的话, 将使用 padding 来填充空白。

程序名称: u.rb

```
a = "tianen"
p a.center(16)
p a.ljust(16)
p a.rjust(16)
p a.center(16, "* & ")
```

```
p a.ljust(16,"* % ")
p a.rjust(16,"* % ")
```

程序运行结果如图 3-21 所示。

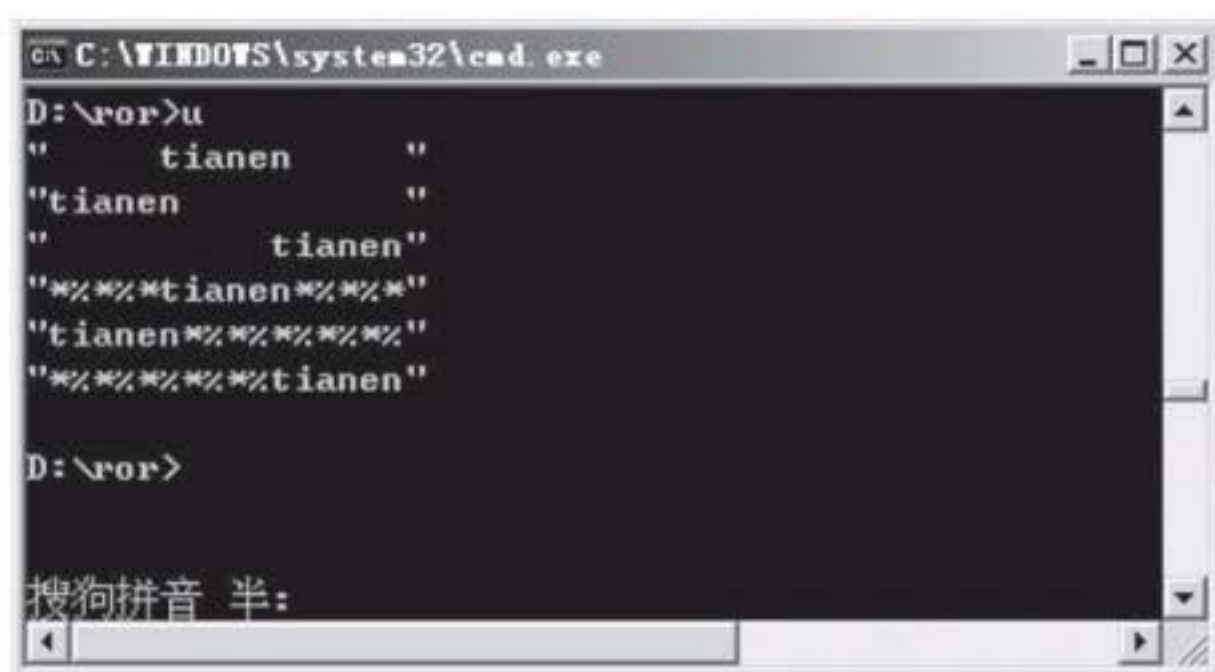


图 3-21 若干格式化方法

(16) `chomp([rs])` 和 `chomp!([rs])`

删除字符串尾部的换行符,该换行符由 `rs` 指定。`rs` 的默认值取自变量 `$/` 的值。若 `rs` 的取值是 `nil` 的话,将不作任何动作。若 `rs` 是空字符串(段落模式)的话,将删除字符串尾部的所有的连续换行符。`chomp` 生成并返回修改后的字符串。而 `chomp!` 会修改字符串本身并返回结果,若没作修改时返回 `nil`。当 `rs` 的值为 `"\n"` (默认值)时,将会把 `"\r"`、`"\r\n"` 和 `"\n"` 全部看作换行符并加以删除,具体案例如下所示。

程序名称: `v.rb`

```
p "tianen \n".chomp
p "tianen \n\r".chomp
p "tianen \n\r".chomp[" "]
p "tianen \r\n".chomp
p "tianen \r".chomp
```

程序运行结果如图 3-22 所示。

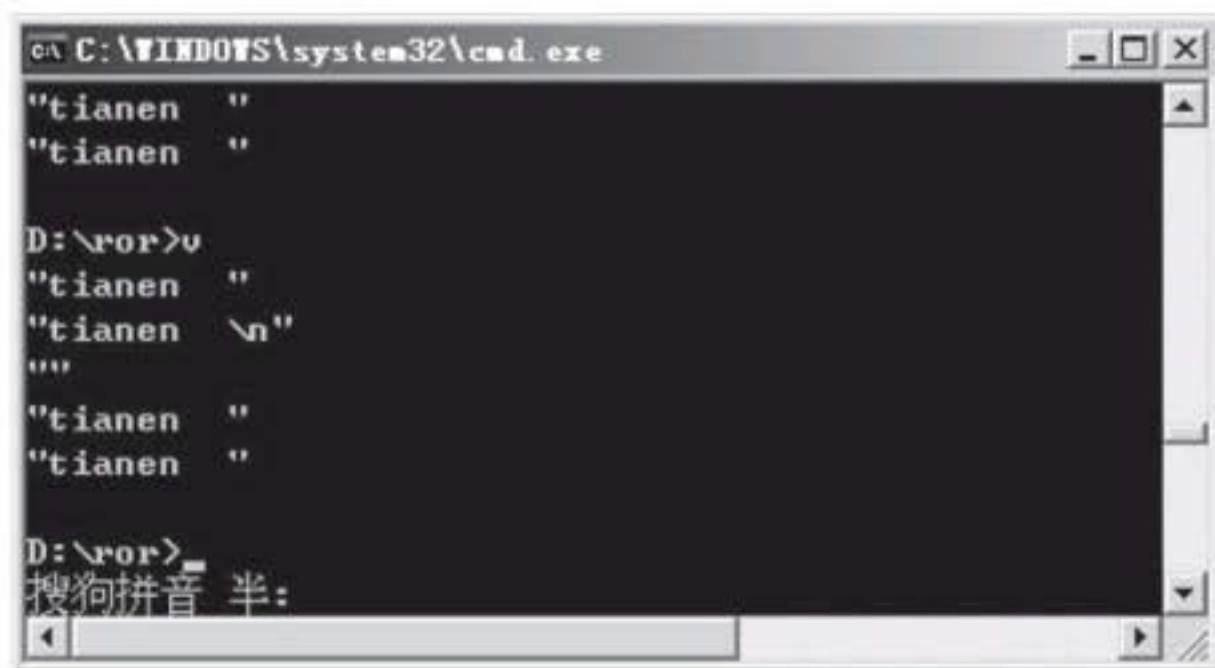


图 3-22 `chomp([rs])` 和 `chomp!([rs])`

(17) chop 和 chop!

删除字符串末尾的字符(若字符串末尾是"\r\n"的话,就删除 2 个字符)。chop 会生成并返回修改后的字符串,而 chop! 会修改字符串本身然后返回结果,若没作修改则返回 nil。

程序名称:w.rb

```
p "tianen \n".chop
p "tianen \n\r".chop
p "tianen \r\n".chop
p "tianen \r".chomp
```

程序运行结果如图 3-23 所示。



图 3-23 chop 和 chop!

(18) clone 和 dup

返回一个与原字符串内容相同的新字符串。对被冻结的字符串使用 clone 会得到一个同样被冻结的字符串,而使用 dup 就会得到一个内容相同但未被冻结的字符串。

程序名称:x.rb

```
p "tianen \n".clone
p "tianen \n\r".dup
```

程序运行结果如图 3-24 所示。



图 3-24 clone 和 dup

(19) count(str[, str2[, ...]])

返回在该字符串中 str 所含字符出现的次数。

str 的形式为:'a-c'表示从 a 到 c,而像"^0-9"这样,当'^'出现在头部时表示"取反"。只有当'-'出现在字符串内部,而非两端时才表示指定一个范围。同样地,只有当'^'出现在字符串头部时才表示"取反"。另外,可以使用反斜线('\')来对'-'、'^'、'\'进行转义操作。

若给出多个参数,则意味着会使用所有参数的交集。

程序名称:y.rb

```
p 'abcdefgc'.count('c')
p '123456789'.count('2378')
p '123456789'.count('2- 8')
p '123456789'.count('^4- 6')
p '123456789'.count('2- 8', '^4- 6')
```

程序运行结果如图 3-25 所示。



图 3-25 count(str[, str2[, ...]])

(20) delete 和 insert

delete(str[, str2[, ...]]) 和 delete!(str[, str2[, ...]]) 从该字符串中删除 str 所包含的字符。str 的形式为:'a-c'表示从 a 到 c,而像"^0-9"这样,当'^'出现在头部时表示"取反"。只有当'-'出现在字符串内部,而非两端时才表示指定一个范围。同样地,只有当'^'出现在字符串头部时才表示"取反"。另外,可以使用反斜线('\')来对'-'、'^'、'\'进行转义操作。若给出多个参数,则意味着会使用所有参数的交集。

delete 会生成并返回修改后的字符串,而 delete! 会修改字符串本身并返回结果,若没作修改则返回 nil。

insert(nth, other) 在第 nth 个字符的前面插入 other 字符串。(除了返回值以外)它等同于 self[nth, 0] = other。

程序名称:aa.rb

```
p "123456789".delete("2- 8", "^4- 6")
p "123456789".delete("2378")
str = "foobaz"
p str.insert(3, "bar")
```

程序运行结果如图 3-26 所示。



图 3-26 delete 和 insert

(21) downcase 和 downcase!

将字符串中的大写字母都改为小写字母。downcase 生成并返回修改后的字符串。而 downcase! 则会修改字符串本身并返回结果,若没有作修改,则返回 nil。

若没有正确设置 \$KCODE,部分汉字代码也会被修改(在 ShiftJIS 编码中就会发生这种情况)。相反,即使设置了 \$KCODE,也不会修改多字节字符的字母。

程序名称:ab.rb

```
p "ABCDsef 天".downcase
```

程序运行结果如图 3-27 所示。



图 3-27 downcase 和 downcase!

(22) dump

使用反斜线表示法替换字符串中的非显示字符,并返回修改后的字符串。

程序名称:ac.rb

```
puts "abc\r\n\f\x00\b10\\\"".dump
```

程序运行结果如图 3-28 所示。



图 3-28 downcase 和 downcase!

(23) 迭代器

字符串有三种重要的迭代器：

```
each([rs]) {|line| ... }
each_line([rs]) {|line| ... }
each_byte {|byte| ... }
upto(max) {|s| ... }
```

`each([rs]) {|line| ... }` 和 `each_line([rs]) {|line| ... }` 对字符串中的各行进行迭代操作。此时, `rs` 中的字符串将成为行切分符, 行切分符的默认值取自变量 `$/` 的值。各 `line` 中包含用作切分符的字符串。若将 `rs` 设为 `nil`, 则意味着不作行的切分, 若设为空字符串 `""` 则将连续的换行当做行切分符(段落模式)。

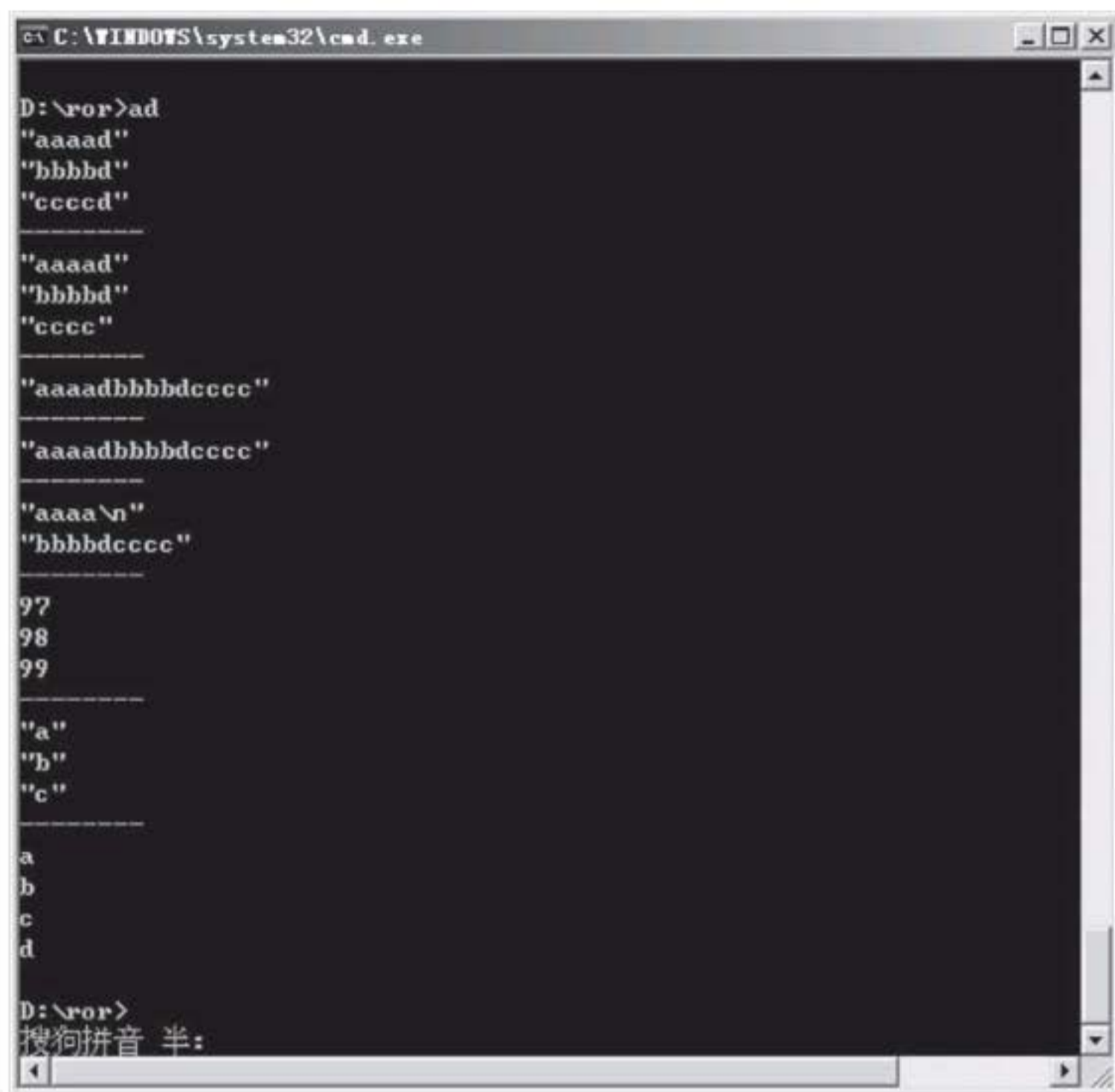
`each_byte {|byte| ... }` 对字符串中的各个字节进行迭代操作。

`upto(max) {|s| ... }` 在从字符串到 `max` 的范围内, 依次取出“下一个字符串”后将其传给块, 进行迭代操作。

程序名称: `ad.rb`

```
"aaaadbbbbbdcdddd".each("d") {|line| p line}
puts "- - - - -"
"aaaadbbbbbdcddd".each("d") {|line| p line}
puts "- - - - -"
"aaaadbbbbbdcddd".each("") {|line| p line}
puts "- - - - -"
"aaaadbbbbbdcddd".each() {|line| p line}
puts "- - - - -"
"aaaa\nbbbbbdcddd".each() {|line| p line}
puts "- - - - -"
"abc".each_byte {|byte| p byte }
puts "- - - - -"
"abc".each_byte {|byte| p byte.chr }
puts "- - - - -"
("a" .. "d").each do |str|
  puts str
end
```

程序运行结果如图 3-29 所示。



```
C:\WINDOWS\system32\cmd.exe
D:\ror>ad
"aaaad"
"bbbbd"
"cccd"
-----
"aaaad"
"bbbbd"
"cccc"
-----
"aaaadbbbbdcccc"
-----
"aaaadbbbbdcccc"
-----
"aaaa\n"
"bbbbdcccc"
-----
97
98
99
-----
"a"
"b"
"c"
-----
a
b
c
d
D:\ror>
搜狗拼音 半:
```

图 3-29 迭代器

(24) empty?

若字符串为空(也就是说其长度为 0),则返回真。

程序名称:ae.rb

```
puts "a".empty?
puts "".empty?
```

程序运行结果如图 3-30 所示。



```
C:\WINDOWS\system32\cmd.exe
D:\ror>ae
false
true
D:\ror>
搜狗拼音 半:
```

图 3-30 empty?

(25) 正则替换和迭代

Ruby 中依照正则表达式进行匹配与替换的方法主要有如下几个：

```
gsub(pattern, replace)
gsub! (pattern, replace)
gsub(pattern) {|matched| .... }
gsub! (pattern) {|matched| .... }
```

以 `replace` 来替换字符串中所有与 `pattern` 相匹配的部分。`replace` 中的 `\&` 和 `\0` 被替换为匹配的子字符串,而 `\1 ... \9` 被替换为第 `n` 个括号的内容。在替换字符串 `replace` 中,还可以使用 `\'`或`\+`,它们分别对应于 `$'`、`$+`。

省略 `replace` 参数时,该方法就相当于一个迭代器,它以块的计算结果进行替换。匹配的子字符串被当做参数传递给块。与没带块时不同的是,可以在块中调用内部变量 `$`。

`gsub` 生成并返回替换之后的字符串。而 `gsub!` 会修改 `self` 本身并返回结果,若没有进行置换则返回 `nil`。

注意:不能在 `replace` 中使用 `$`,这是因为在对该字符串进行计算时,尚未进行匹配操作所致。另外,在 `replace` 中必需对`\`进行二重转义。

程序名称:af.rb

```
p 'abcabc'.gsub(/b/, '(\&')
p 'abcabc'.gsub(/b/) {|s| s.upcase }
p 'abcabc'.gsub(/b/) { $ &.upcase }
p 'abcdefg'.gsub(/cd/, 'CD')
str = 'abcdefg'
str.gsub! (/cd/, 'CD')
p str
p 'abbbxabx'.gsub(/a(b+ )/, '\1')
# p 'abbbcd'.gsub(/a(b+ )/, "# {$ 1}")          # 错误
# p 'abbbcd'.gsub(/a(b+ )/, "\1")                # 错误
p 'abbbcd'.gsub(/a(b+ )/, "\\1")                  # 正确
p 'abbbcd'.gsub(/a(b+ )/, '\1')                  # 正确
p 'abbbcd'.gsub(/a(b+ )/, '\\1')                 # 正确(更安全)
p 'abbbcd'.gsub(/a(b+ )/) { $ 1 }                 # 正确(最安全)
```

程序运行结果如图 3-31 所示。

类似的方法有：

```
sub(pattern, replace)
sub! (pattern, replace)
sub(pattern) {|matched| ... }
sub! (pattern) {|matched| ... }
```

用 `replace` 来替换首次匹配 `pattern` 的部分。若带块调用的话,就以块的计算值来替换首次匹配的部分。`sub` 生成并返回替换后的字符串。而 `sub!` 会修改字符串本身并返回结果。若没有进行替换则返回 `nil`。



图 3-31 正则替换和迭代

(26) hex 和 oct

hex:把字符串看作 16 进制数形式,并将其变为整数。"0x"和"0X"前缀将被忽略。若遇到[_0-9a-fA-F]之外的字符时,就只转换它前面的部分。若变换对象是空字符串,则返回 0。

oct:将字符串看做是 8 进制字符串,并将其变为整数。oct 可根据字符串的前缀("0", "b", "0B", "0x", "0X")来进行 8 进制以外的相关处理。若遇到不能看作整数的字符,就只变换此前的内容。若变换对象为空字符串,则返回 0。

程序名称:ag.rb

```
p "10".hex
p "ff".hex
p "0x10".hex
p "- 0x10".hex
p "xyz".hex
p "10z".hex
p "1_0".hex
p "10".oct
p "010".oct
p "8".oct
p "0b10".oct
p "10".oct
p "010".oct
p "0x10".oct
p "1_0_1x".oct # => 65
```

程序运行结果如图 3-32 所示。



图 3-32 hex 和 oct

(27) include? (substr)

若字符串中包含 substr 子字符串,就返回真。若 substr 是从 0~255 之间的 Fixnum 时,将把它看作字符代码,若包含该代码所对应的字符,就返回真。

程序名称:ah. rb

```
p "i love you".include? ("love")
p "i love you".include? ("me")
```

程序运行结果如图 3-33 所示。



图 3-33 include

(28) index(pattern[, pos])和 rindex(pattern[, pos])

index 方法按照从左到右的顺序搜索子字符串,并返回搜索到的子字符串的左侧位置。若没有搜索到则返回 nil。在参数 pattern 中,可以使用字符串、0~255 之间的字符代码或正则表达式来指定想要搜索的子字符串。

若给出了 pos,则从相应位置开始搜索。省略 pos 时其默认值为 0。若 pos 为负,则从字符串尾部找到相应位置后开始搜索。

rindex 方法按照从右到左的顺序来搜索子字符串,并返回找到的子字符串左侧的位置。若搜索失败则返回 nil。在参数 pattern 中,可以使用字符串,从 0~255 之间的字符代码或正则表达式来指定想要搜索的子字符串。若给出了 pos,就从相应位置开始搜索。省略 pos 时,其值为"字符串.size(右端)"。若 pos 为负,则从尾部找到相对应的位置后开始搜索。该方法

的运作情况并非与 index 完全相反。开始搜索时,起始位置是从右向左移动,但是对比子字符串时还是从左向右进行的。

程序名称:ai. rb

```
p "astrochemistry".index("str")
p "character".index(? c)
p "regexpindex".index(/e.* x/, 2)
p "foobarfoobar".index("bar", 6)
p "foobarfoobar".index("bar", - 6)
p "astrochemistry".rindex("str")
p "character".rindex(? c)
p "regexprindex".rindex(/e.* x/, 2)
p "foobarfoobar".rindex("bar", 6)
p "foobarfoobar".rindex("bar", - 6)
p "foobar".index("bar", 2)
p "foobar".rindex("bar", - 2)
```

程序运行结果如图 3-34 所示。

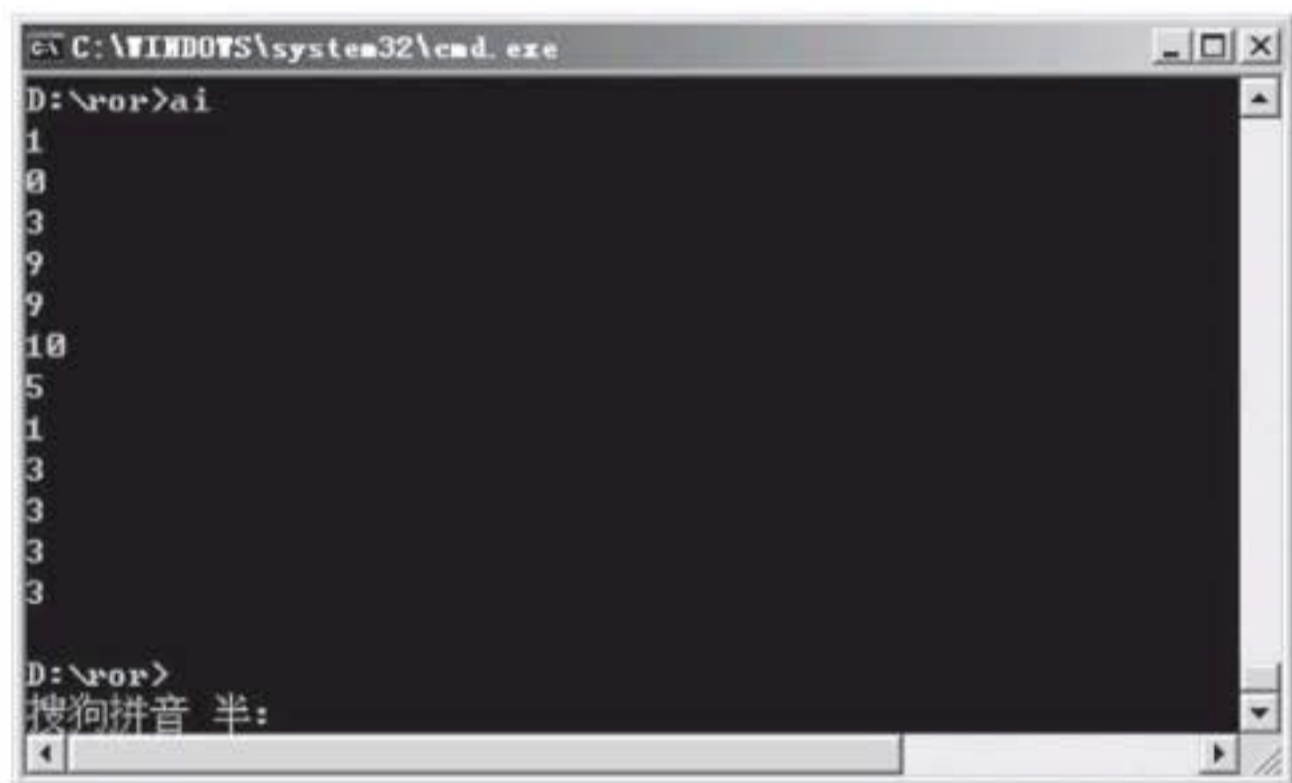


图 3-34 index(pattern[, pos]) 和 rindex(pattern[, pos])

(29) scan

scan(re) 和 scan(re) {|s| ...} 是使用正则表达式 re 反复对字符串进行匹配操作,并以数组的形式返回匹配成功的子字符串。若正则表达式中包含括号,则返回与括号内的 pattern 匹配成功的子字符串的数组。若带块调用,匹配成功的子字符串(若包含括号,则是与括号内的 pattern 匹配成功的字符串的数组)将成为块的形参。

程序名称:aj. rb

```
p "foobar".scan(/./)
p "foobarbazfoobarbaz".scan(/ba./)
p "foobar".scan(/(.)/)
p "foobarbazfoobarbaz".scan(/(ba)(.)/)
"foobarbazfoobarbaz".scan(/ba./) {|s| p s}
"foobarbazfoobarbaz".scan(/(ba)(.)/) {|s| p s}
```

程序运行结果如图 3-35 所示。

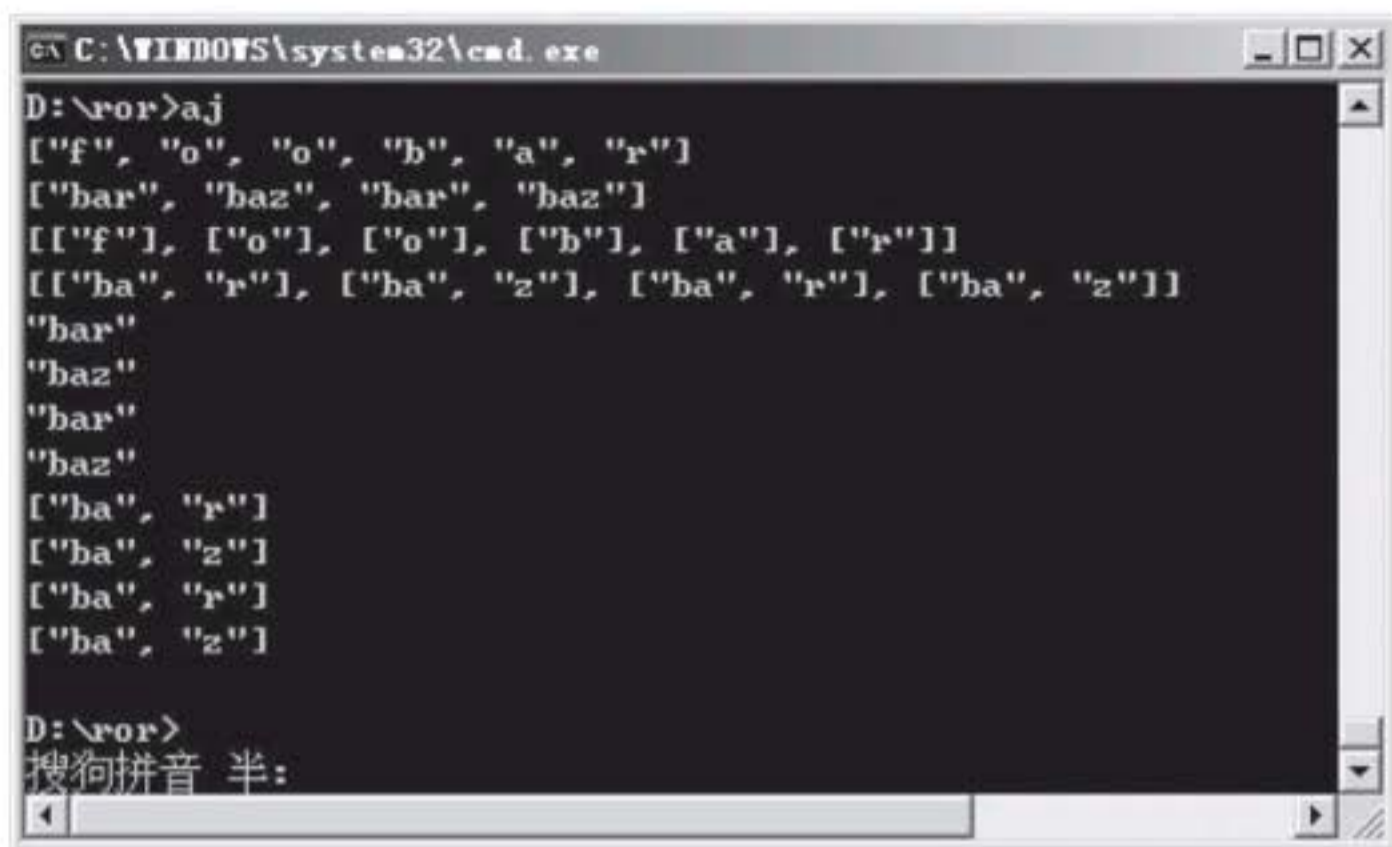


图 3-35 scan

(30) slice

slice 方法的作用与[]符号类似。包含如下几种形式：

```
slice(nth[, len])
slice(substr)
slice(first..last)
slice(first...last)
slice(regexp[, nth])
slice! (nth[, len])
slice! (substr)
slice! (first..last)
slice! (first...last)
slice! (regexp[, nth])
```

注意：若参数超出范围则返回 nil。

程序名称：ak.rb

```
p "foobar".slice(0,2)
p "foobar"[0..2]
```

程序运行结果如图 3-36 所示。



图 3-36 slice

(31) split([sep[, limit]])

使用 sep 指定的 pattern 来分割字符串,并将分割结果存入数组。

sep 可以是下列之一:

正则表达式:把正则表达式的匹配结果当作切分符来切分字符串。如果使用了括号群组,与群组相匹配的字符串也会出现在最后的数组中。

1 字节的字符串:把该字符当作切分符来进行切分(ruby 1.6)。

2 字节以上的字符串:把与 Regexp.new(sep)相匹配的字符串当作切分符来进行切分。

省略或 nil:把 \$; 的值当作切分符来进行切分。

1 字节的空白 ' ' 或使用了 \$; 且其值为 nil 时:除了头部的空白之外,使用空白进行切分。

空字符串 '' 或与空字符串相匹配的正则表达式:以单个字符为单位进行切分。可识别多字节字符。

若 sep 中的 pattern 与空字符串相匹配的话,将以单个字符为单位对字符串进行切分。

程序名称:al.rb

```
p " a \t b \n c".split(/\s+ /)
p " a \t b \n c".split
p " a \t b \n c".split("")
p "aaaabaaabaaa".split('b')
p "abc".split(//)
```

程序运行结果如图 3-37 所示。



图 3-37 split

limit 参数可以是下列之一:

省略或 0:删除数组尾部的空字符串。

limit > 0:为数组分配至多 limit 个元素。

limit < 0:相当于指定 limit 的值为正无穷。

程序名称:am.rb

```
p "a,b,c,...".split(//,/)
p "a,b,c,...".split(//,/, 0)
```

```
p "a,b,c,,,".split(/,/ , 3)
p "a,b,c,,,".split(/,/ , 6)
p "a,b,c,,,".split(/,/ , - 1)
p "a,b,c,,,".split(/,/ , 100)
```

程序运行结果如图 3-38 所示。



图 3-38 split

(32) squeeze

```
squeeze([str[,str2[, ... ]]])
squeeze! ([str[,str2[, ... ]]])
```

压缩由 str 所含字符构成的重复字符串。str 的形式为: 'a-c' 表示从 a 到 c, 而像 '^0-9' 这样, 当 '^' 出现在头部时表示“取反”。只有当 '-' 出现在字符串内部, 而非两端时才表示指定一个范围。同样地, 只有当 '^' 出现在字符串头部时才表示“取反”。另外, 可以使用反斜线 ('\') 来对 '-', '^', '\ 进行转义操作。若给出多个参数, 则意味着会使用所有参数的交集进行压缩。

squeeze 会生成并返回修改后的字符串。而 squeeze! 会修改字符串本身并返回结果。若没作修改则返回 nil。

这个方法很有用, 当一个字符串中含有多个空格时, 可以用这个方法将多个空格变成一个。

程序名称: an.rb

```
p "112233445566778899".squeeze
p "112233445566778899".squeeze("2- 8")
p "112233445566778899".squeeze("2- 8", "^ 4- 6")
p "112233445566778899".squeeze("2378")
p "i love you".squeeze(" ")
```

程序运行结果如图 3-39 所示。

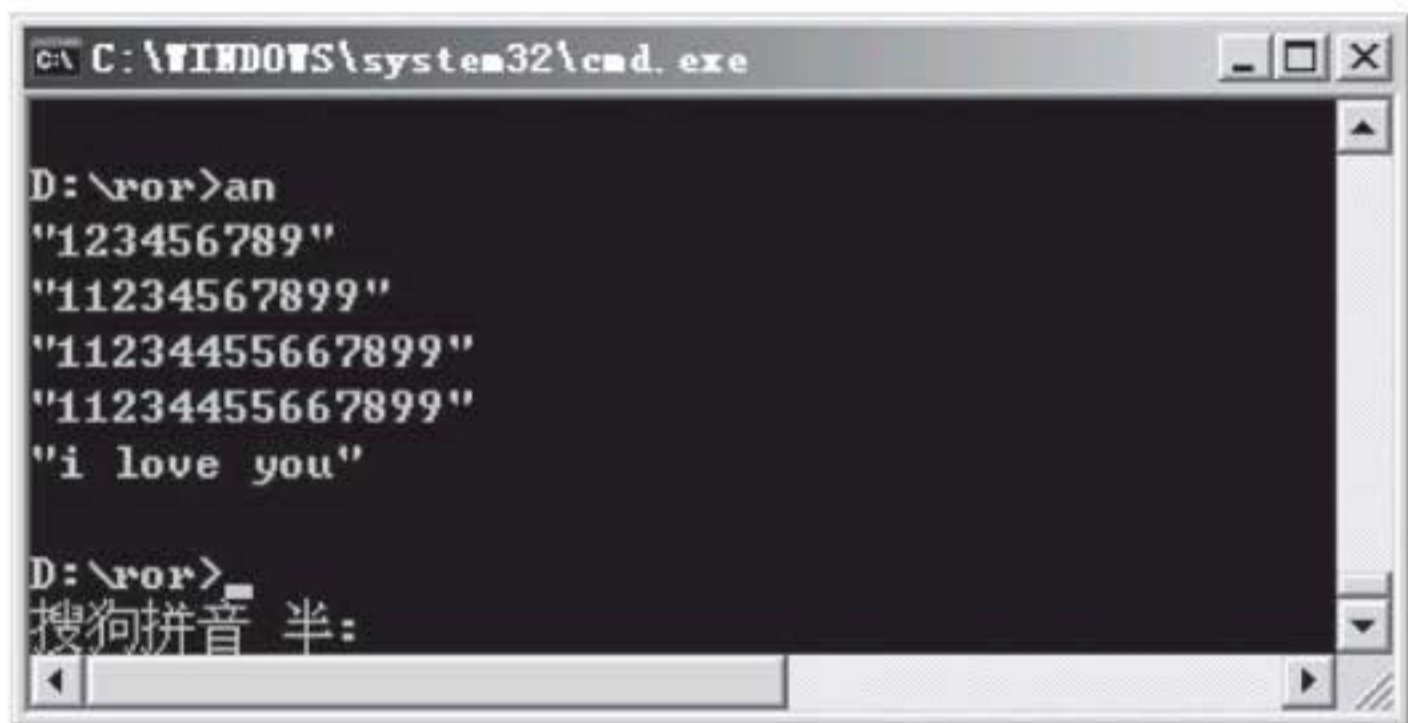


图 3-39 squeeze

(33) strip 和 strip!

strip 和 strip! 方法用来删除头部和尾部的所有空白字符。空白字符是指" \t\r\n\f\v"。strip 生成并返回修改后的字符串。strip! 会修改字符串本身并返回结果。若没有进行删除动作,则返回 nil。

注意:它会删除右侧的空白字符和"\0",但不会对左侧的"\0"进行特殊处理。

程序名称:ao.rb

```
p " abc \r\n".strip
p "abc\n".strip
p " abc".strip
p "abc".strip
str = "\tabc\n"
p str.strip
p str
str = " abc\r\n"
p str.strip!
p str
str = "abc"
p str.strip!
p str
str = " \0 abc \0"
p str.strip
```

程序运行结果如图 3-40 所示。

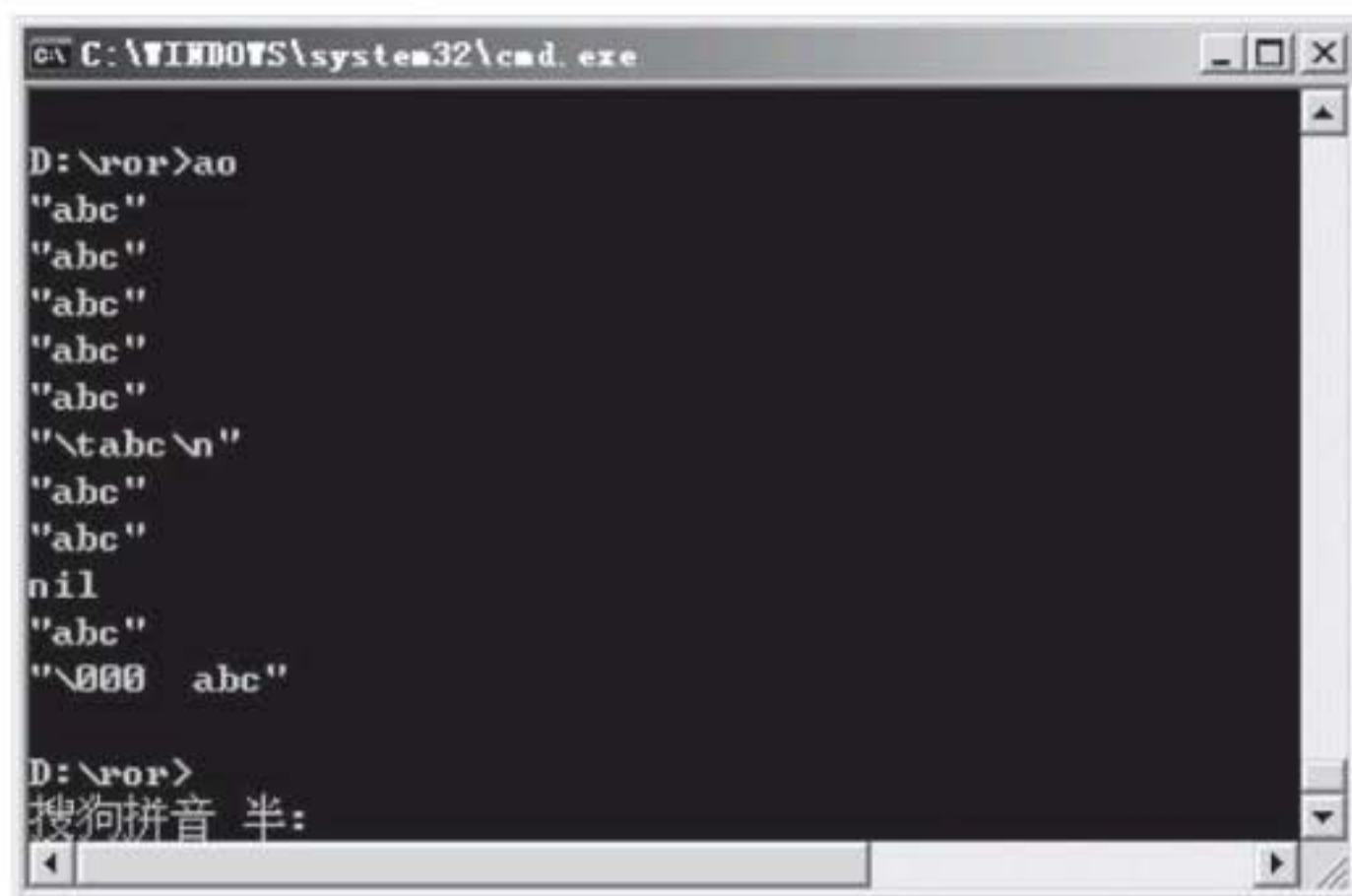


图 3-40 strip 和 strip!

(34) lstrip 和 lstrip!

删除字符串头部的所有空白字符。空白字符是指" \t\r\n\f\v"。lstrip 会生成并返回加工后的字符串。lstrip! 会修改字符串本身并返回结果。如果没有删除空白字符,则返回 nil。

注意:lstrip 和 lstrip! 不会删除"\0"。

程序名称:ap.rb

```
p " abc\n".lstrip
p "\t abc\n".lstrip
p "abc\n".lstrip
str = "\nabc"
p str.lstrip
p str
str = " abc"
p str.lstrip!
p str
str = "abc"
p str.lstrip!
p str
p "\0abc".lstrip
p "\0abc".lstrip!
```

程序运行结果如图 3-41 所示。

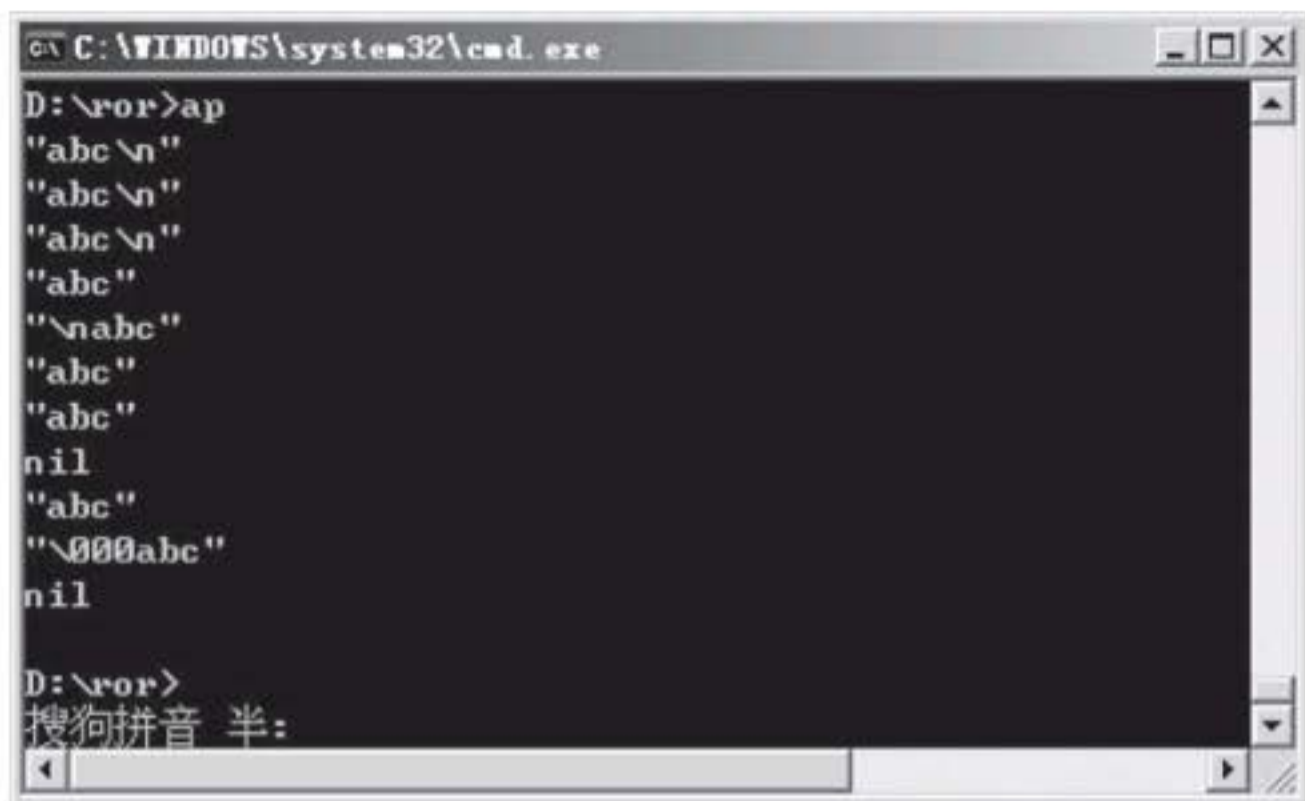


图 3-41 lstrip 和 lstrip!

(35) rstrip 和 rstrip!

删除字符串尾部的所有空白字符。空白字符是指" \t\r\n\f\v"。rstrip 会生成并返回加工后的字符串。rstrip! 会修改字符串本身并返回结果。如果没有删除空白字符,则返回 nil。

注意:rstrip 可以删除 "\0"。

程序名称:aq.rb

```
p "abc \n".rstrip
p "abc \t\r\n".rstrip
p "abc".rstrip
str = "abc\n"
p str.rstrip
p str
str = "abc "
p str.rstrip!
p str
str = "abc"
p str.rstrip!
p str
p "abc\0\0\0".rstrip
p "abc\0\0".rstrip!
```

程序运行结果如图 3-42 所示。



图 3-42 rstrip 和 rstrip!

(36) sum([bits=16])

计算字符串的 bits 位的校验和。它等同于：

```
sum = 0
str.each_byte {|c| sum += c}
sum = sum & ((1 << bits) - 1) if bits != 0
```

例如,可以使用如下代码来得到与 System V 的 sum(1)命令相同的值。

```
sum = 0
while gets
  sum += $_.sum
end
sum % = 65536
```

(37) swapcase 和 swapcase!

将所有的大写字母改为小写字母,小写字母改为大写字母。swapcase 生成并返回修改后的字符串。而 swapcase! 则会修改字符串本身并返回结果,若没有作修改,则返回 nil。

程序名称:ar,rb

```
p "AbCd 天 deF".swapcase
p "AbCd 天 deF".swapcase!
p "AbCddeF".swapcase
```

程序运行结果如图 3-43 所示。

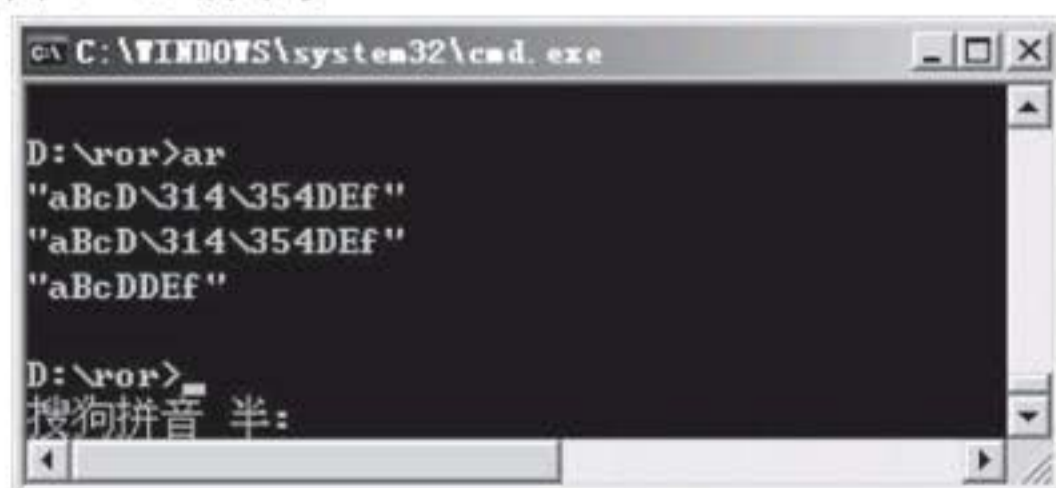


图 3-43 swapcase 和 swapcase!

(38) tr(search, replace) 和 tr! (search, replace)

若字符串中包含 search 字符串中的字符时,就将其替换为 replace 字符串中相应的字符。search 的形式为:'a-c'表示从 a 到 c,而像"^0-9"这样,当'^'出现在头部时表示“取反”。replace 中也可以使用'-'来指定范围。

只有当'-'出现在字符串内部,而非两端时才表示指定一个范围。同样地,只有当'^'出现在字符串头部时才表示“取反”。另外,可以使用反斜线('\')来对'-'、'^'、'\'进行转义操作。

若 replace 所指定的范围比 search 的范围小的话,则认为 replace 的最后一个字符会一直重复出现下去。tr 生成并返回替换后的字符串。而 tr! 会修改字符串本身并返回结果,若没有进行替换操作则返回 nil。

程序名称:as.rb

```
p "foo".tr('a- z', 'A- Z')  
p "FOO".tr('A- Z', 'a- z')
```

程序运行结果如图 3-44 所示。

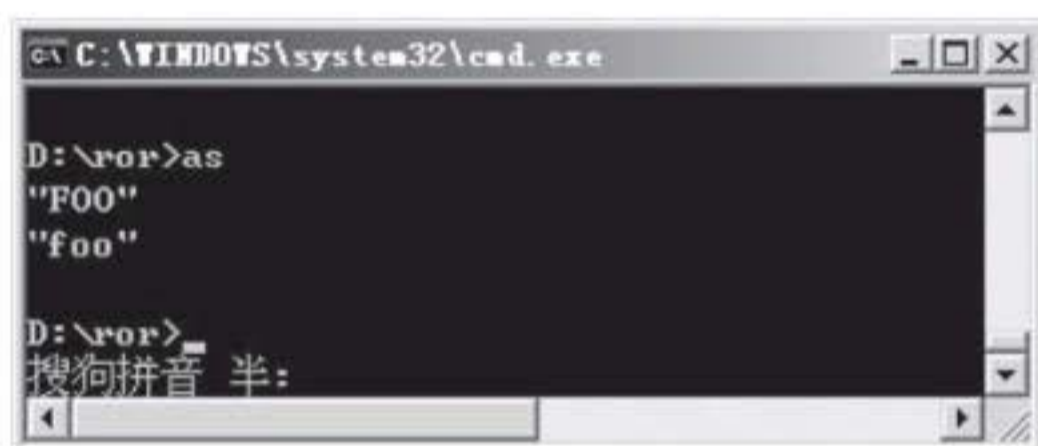


图 3-44 tr

(39) tr_s(search, replace) 和 tr_s! (search, replace)

若字符串中包含 search 字符串中的字符,就将其替换为 replace 字符串中相应的字符。同时,若替换部分中出现重复字符串时,就将其压缩为 1 个字符。

search 的形式为:'a-c'表示从 a 到 c,而像"^0-9"这样,当'^'出现在头部时表示“取反”。replace 中也可以使用'-'来指定范围。

只有当'-'出现在字符串内部,而非两端时才表示指定一个范围。同样地,只有当'^'出现在字符串头部时才表示“取反”。另外,可以使用反斜线('\')来对'-'、'^'、'\'进行转义操作。

若 replace 所指定的范围比 search 的范围小的话,则认为 replace 的最后一个字符会一直重复出现下去。

tr_s 生成并返回替换后的字符串。而 tr_s! 会修改字符串本身并返回结果,若没有进行替换操作则返回 nil。

注意:该方法的运作方式与 tr(search, replace).squeeze(replace)并不相同。tr 与 squeeze 连用时,会把整个替换后的字符串当作 squeeze 的对象来处理,而 tr_s 只把被替换的部分当作 squeeze 的对象来处理。

程序名称:at.rb

```

p "foo".tr_s('a- z', 'a- Z')
p "FOO".tr_s('a- Z', 'a- z')
p "foo".tr_s! ('a- z', 'a- Z')
p "FOO".tr_s! ('a- Z', 'a- z')
p "foo".tr_s("o", "f")
p "foo".tr("o", "f").squeeze("f")

```

程序运行结果如图 3-45 所示。



图 3-45 tr_s

(40) upcase 和 upcase!

在 ASCII 字符串的范围内,将所有字母都变为大写形式。upcase 生成并返回修改后的字符串。而 upcase! 会修改字符串本身并返回结果,若没有进行转换操作则返回 nil。

程序名称:au, rb

```

p "foo".upcase
p "FoO".upcase!

```

程序运行结果如图 3-46 所示。

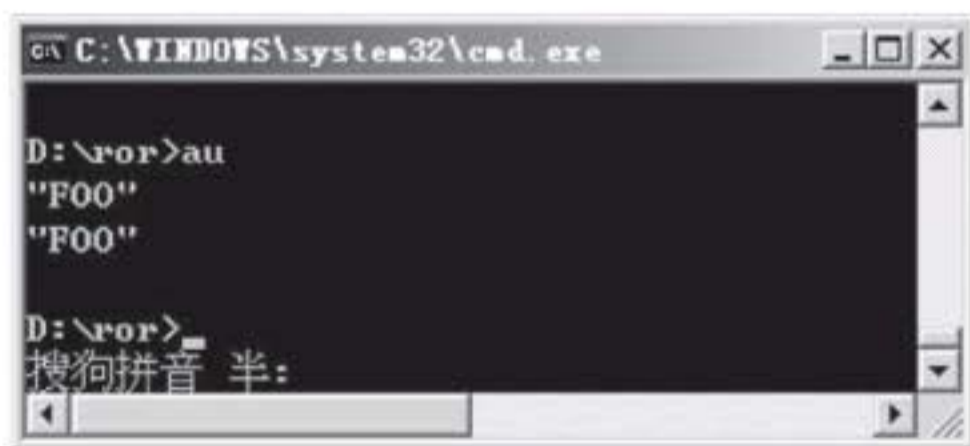


图 3-46 upcase 和 upcase!

(41) match

match(regex)和 match(regex[, pos])方法与 regexp.match(self[, pos])相同,后面介绍。

程序名称:au, rb

```

p "foo".upcase
p "FoO".upcase!

```

程序运行结果如图 3-47 所示。

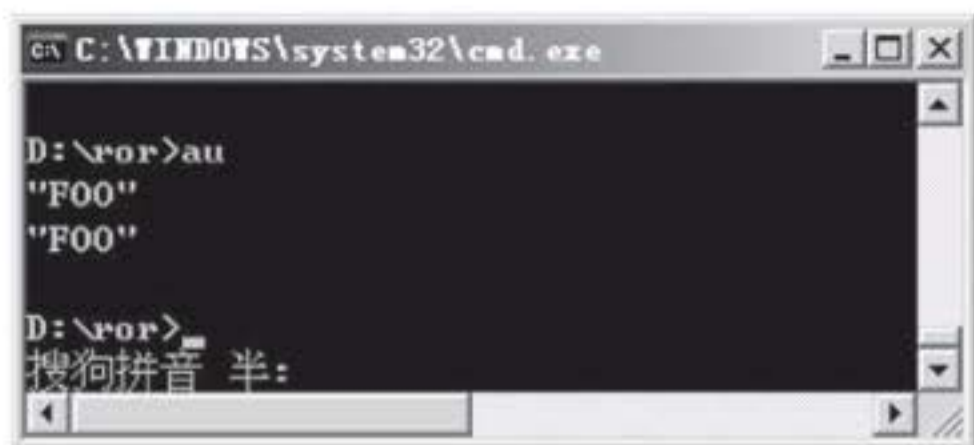


图 3-47 upcase 和 upcase!

3.2.3 字符串方法总结

前面列举了字符串的许多方法,为便于记忆,这里小做总结。

①字符串有两个重要的符号:“*”和“[]”,灵活地使用它们,可以完成大量的工作。包括字符串查询、匹配、替换等,而且可以使用正则表达式。

②基本的格式化方法。包括:center,ljust,rjust,hex,oct,capitalize 和 capitalize!,downcase 和 downcase!,clone 和 dup,dump,next 和 next!,succ 和 succ!。

③细致的字符串处理方法。包括:length 和 size,include?,empty?,count,chomp 和 chomp!,chop 和 chop!,delete 和 delete!,insert,gsub! 和 sub!,index,rindex,scan,slice,split,squeeze 和 squeeze!,strip 和 strip!,lstrip 和 lstrip!,rstrip 和 rstrip!,tr 和 tr!,tr_s 和 tr_s!,match。这些方法可以用来实现字符串的查找、替换等高级操作。

④字符串有四个重要的迭代器:each,each_line,each_byte,upto。

在字符的处理方面,存在语言问题。单字节语言和双字节语言的处理有时候不能统一。这一点,目前 Java 和 C# 处理得很好,Ruby 和 PHP 这类语言的处理还欠妥当。

对于“天恩”这个词,使用 length 方法得到的结果是 4,不是 2。因为汉字是双字节的,length 得到的结果是字节数目而不是字符数目。

程序名称:av.rb

```
require 'jcode'
$ KCODE= 'u'
p "genius 天恩".length
p "genius 天恩".split('').length
p "genius 天恩".split('')
```

程序运行结果如图 3-48 所示。



图 3-48 字符串处理

3.3 正则表达式

Ruby 中常会用到正则表达式。正则表达式是用来表达字符串的模式(pattern)的。也可以这么理解:先使用正则表达式来表示某种搜索规则,然后使用该正则表达式来完成字符串的搜索工作。

3.3.1 Ruby 正则表达式的基本用法

Ruby 用//将正则表达式括起来。[^]表示开头,\$表示结尾,.*表示0个以上的任意字符。正则表达式中有很多具有特别意义的字符。如下:

[] 是范围描述符。[a-z]表示从a到z之间的任意一个。

\w 表示英文字母和数字。即[0-9 A-Z a-z]。

\W 表示非英文字母和数字。即[^0-9 A-Z a-z]。

\s 表示空字符,即[\t\n\r\f]。

\S 表示非空字符,即[^ \t\r\n\f]。

\d 表示数字,即[0-9]。

\D 表示非数字,即[^0-9]。

\b 表示词边界字符(在范围描述符外部时)。

\B 表示非词边界符

\b 是退格符(0x08)(在范围描述符内部时)。

* 表示前面元素出现0次以上。

• 匹配任意字符。

+ 表示前面元素出现1次以上。

{m,n} 表示前面元素最少出现m次,最多出现n次。

? 表示前面元素出现0次或1次。

| 表示选择。

() 表示群组。

其他字符表示该字符本身。

例如,“[^][a-z]+”表示“第一个字符是f,后面是若干个从a到z之间的字符”,可以是“fobar”或“fool”等。这种表达法就是正则表达式(regular expression)。在搜索字符串时正则表达式非常有用。

前面已经学到了许多和正则表达式相关的字符串函数,这一节就可以用了。

正则表达式是类型 Regexp 的对象。它们可以用显式的构造函数建立或者直接用/pattern/和 %r/pattern/这种格式的字符常量构造。

如:

```
a = Regexp.new('^ \s* [a- z]')
b = /^ \s* [a- z]/
c = %r{^ \s* [a- z]}
```

一旦有了一个正则表达式对象,就可以用它和一个字符串比较,通过使用正则表达式对象的 match(aString) 方法或者用匹配操作符 =~(确定匹配)和 ! =~匹配操作符返回模式匹配

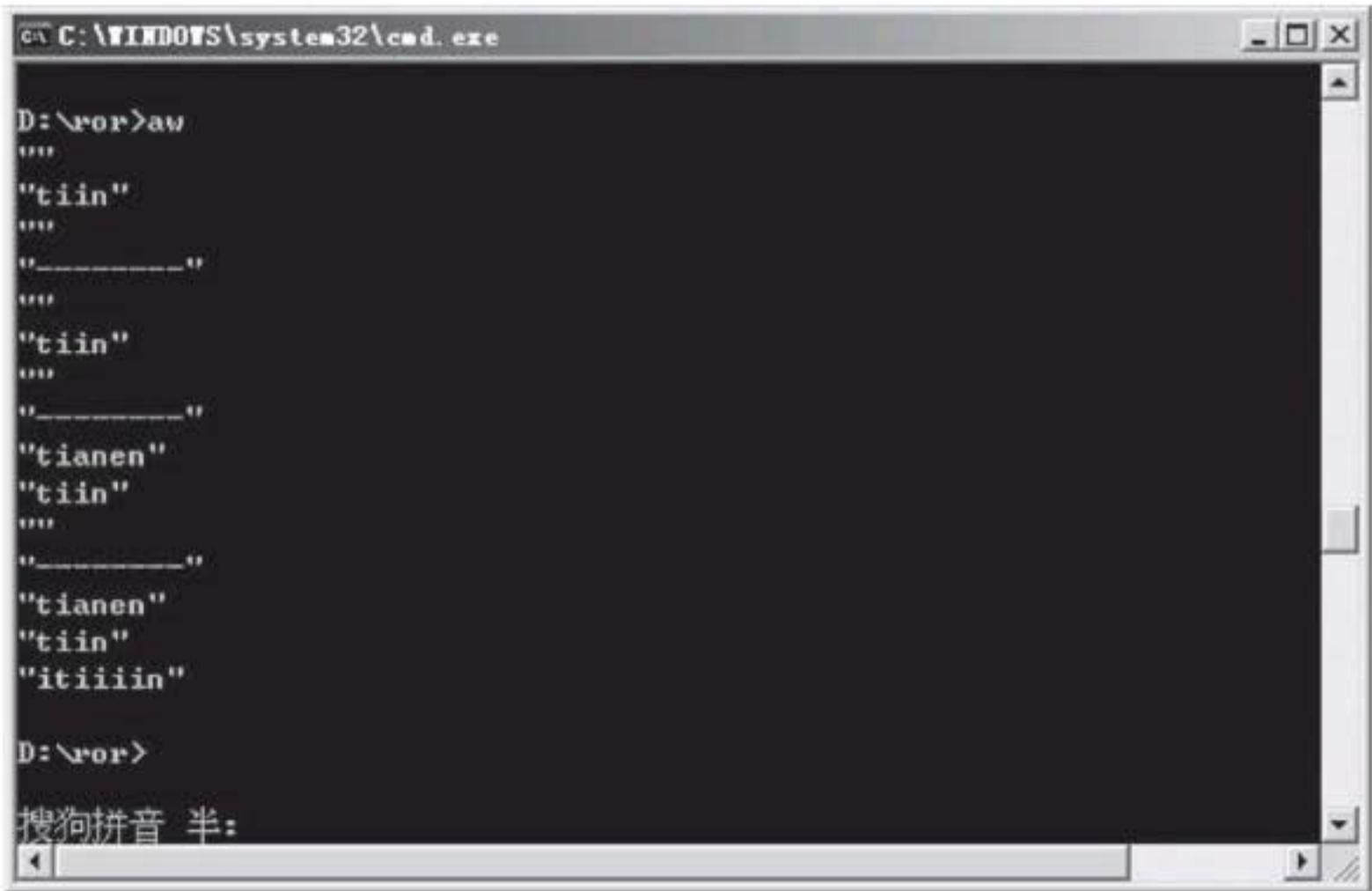
成功的字符位置。它们还有一个设置所有 Ruby 变量的额外作用。\$& 接受模式匹配成功的那部分字符，\$(键盘 1 左边那个键)接受模式匹配成功前面那一部分字符，\$'接受模式匹配成功后面那部分字符。

每个正则表达式都有一个模式,用来和字符串做匹配。在一个模式中,除了.,|,(,),[, {, +, \, ^, \$, * 和? 之外的字符都是和它本身匹配。如果想在字面上匹配一个上面的特殊字符,要在它前面加一个'\'。

程序名称:aw.rb

```
r= /^ti+ n$ /
p r.match("tianen").to_s
p r.match("tiin").to_s
p r.match("itiin").to_s
p " - - - - - "
r= /^ti* n$ /
p r.match("tianen").to_s
p r.match("tiin").to_s
p r.match("itiiin").to_s
p " - - - - - "
r= /^ti* .* n$ /
p r.match("tianen").to_s
p r.match("tiin").to_s
p r.match("itiiin").to_s
p " - - - - - "
r= /^.* ti* .* n$ /
p r.match("tianen").to_s
p r.match("tiin").to_s
p r.match("itiiin").to_s
```

程序运行结果如图 3-49 所示。



3.3.2 正则表达式在字符串函数中的使用

在前面介绍的字符串函数中,许多都支持正则表达式,这里简介一下,案例如下所示。

程序名称:ax.rb

```
p "i love you".index(/ov/)
p "i love you".index(/ou/)
p "i love you".rindex(/o/)
p "i love you".split(/o/)
```

程序运行结果如图 3-50 所示。



图 3-50 正则表达式

3.4 日期和时间

3.4.1 Time 对象

Time 是 Ruby 中的时间对象。Time.now 返回当前时间。Time 对象中保存的是自起算时间以来所经过的秒数。起算时间定在协调世界时(UTC,或旧称 GMT)的 1970 年 1 月 1 日上午 0 点。现在 Unix 系统的最大时间是协调世界时 2038 年 1 月 19 日上午 3 点 14 分 7 秒。

Time 对象的类方法如下。

① Time.at(time[, usec]) 返回 time 所指时间的 Time 对象。time 可以是 Time 对象,也可以是表示自起算时间以来的秒数的整数或浮点数。

若浮点精度不够,可以使用 usec。它将返回 time + (usec/1000000) 所表示的时间。此时,time 和 usec 都必须是整数。生成的 Time 对象将使用地方时的时区。

② 返回当前时间的 Time 对象用 Time.new 和 Time.now。new 与 now 的区别在于,它会调用 initialize。

通过这个方法,可以很方便地将 Time 对象变成整数保存起来,需要的时候再将整数还原成时间。

程序名称:ay.rb

```
t= Time.new
p Time.now
```

```

now = Time.new
i= now.to_i
p i
now = Time.at(i)
p now

```

程序运行结果如图 3-51 所示。

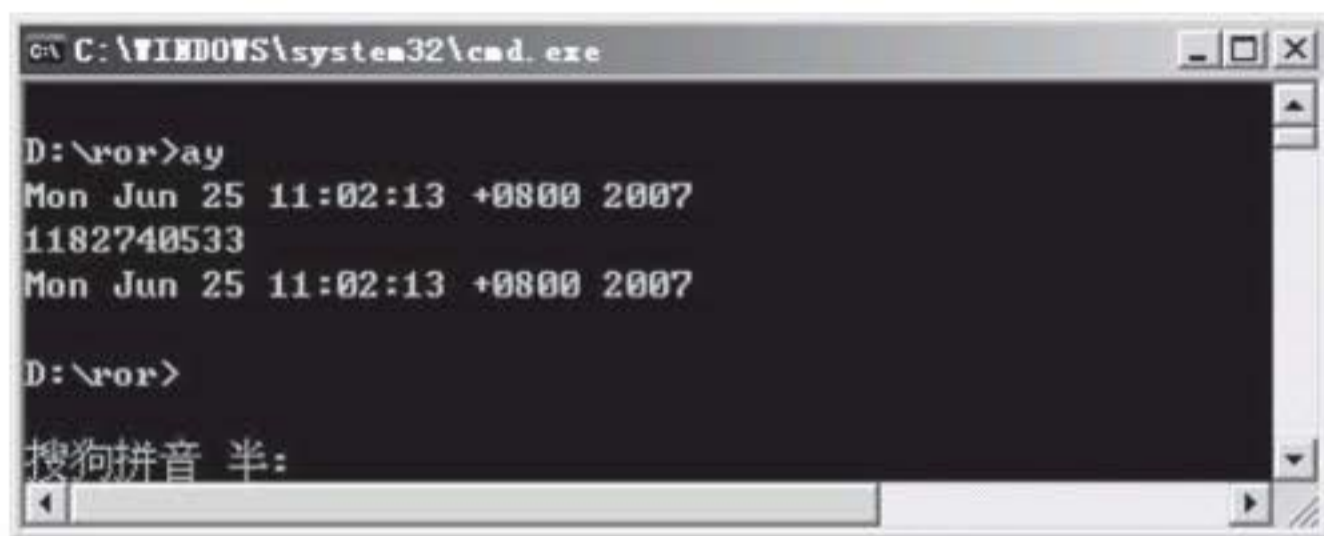


图 3-51 时间对象

③ 将时间变为字符串 `strftime(format)`, 按照 `format` 字符串的模式, 将时间变为字符串, 并返回结果。`format` 字符串可以使用下列形式:

- %A: 星期几的名称(Sunday, Monday ...)
- %a: 星期几的简称(Sun, Mon ...)
- %B: 月份的名称(January, February ...)
- %b: 月份的简称(Jan, Feb ...)
- %c: 日期和时间
- %d: 日期(01~31)
- %H: 24 时制的小时(00~23)
- %I: 12 时制的小时(01~12)
- %j: 一年中的第几天(001~366)
- %M: 分钟(00~59)
- %m: 表示月份的数字(01~12)
- %p: 上午/下午(AM, PM)
- %S: 秒钟(00~60) (60 是闰秒)
- %U: 表示周的数字, 以第一个星期天作为第一周的开始(00~53)
- %W: 表示周的数字, 以第一个星期一作为第一周的开始(00~53)
- %w: 表示星期几的数字, 星期天是 0(0~6)
- %X: 时刻
- %x: 日期
- %Y: 表示公历年份的数字
- %y: 公历年份的后两位数字(00~99)
- %Z: 时区
- %%: % 本身

程序名称:az.rb

```
t= Time.now
a= t.strftime("% a")
c= t.strftime("% c")
d= t.strftime("% a % c")
p a
p c
p d
```

程序运行结果如图 3-52 所示。

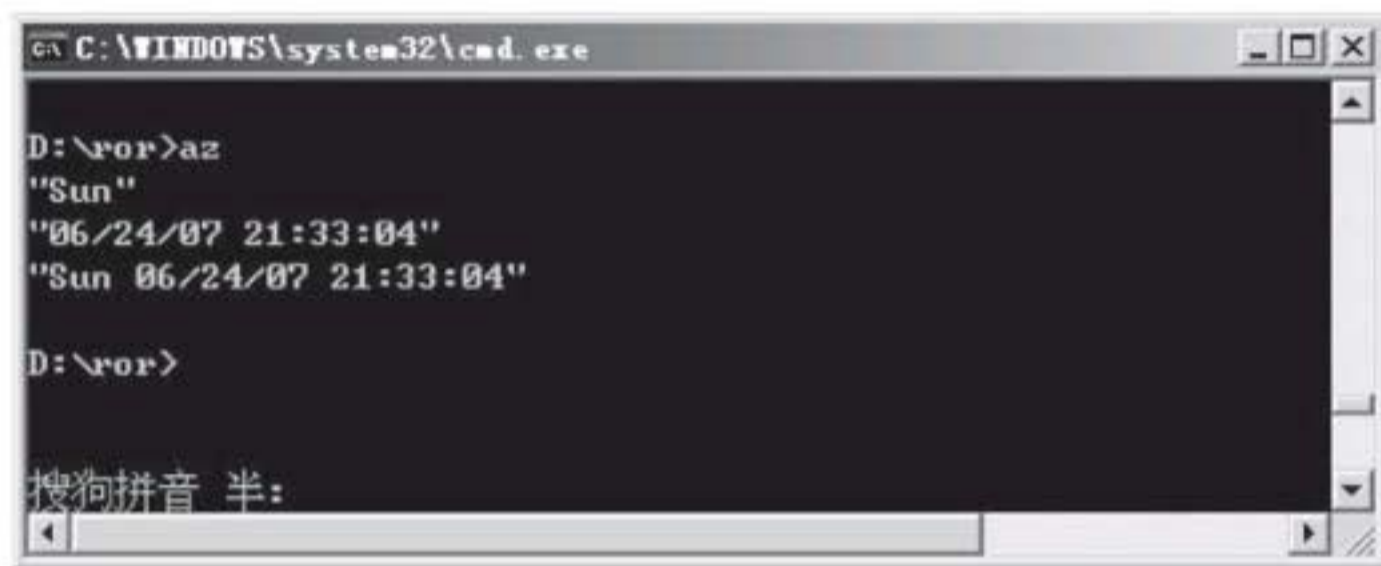


图 3-52 时间对象的格式化

④ 提取时间要素,如下方法可以提取出一个事件对象中所含的要素,如:星期几,年份等。

sec

min

hour

mday

day (mday 的别名)

mon

month (mon 的别名)

year

wday

yday

isdst

dst? (isdst 的别名)

zone

其中:

hour, min, sec:以整数形式返回时、分、秒。

year, month(mon), day(mday):以整数形式返回年、月、日。

wday: 以 0(星期日)到 6(星期六)之间的整数返回星期几。

yday: 以整数(1 到 366)的形式返回某天是一年中的第几天,以 1 月 1 日作为第 1 天。

isdst(dst?): 若为夏令时就返回 true,否则返回 false。

zone: 返回表示时区的字符串。

对于提取的结果,如果是 1 月份,month 就返回 1;如果是 1998 年,year 就返回 1998。另外,yday 从 1 算起。

程序名称:ba.rb

```
t= Time.now  
p t.sec  
p t.mon  
p t.day  
p t.mday
```

程序运行结果如图 3-53 所示。



图 3-53 提取时间要素

⑤ to_a,以一个包括 10 个元素的数组的形式返回某时间。数组元素的排列如下。

sec: 秒 (整数 0~60)

min: 分 (整数 0~60)

hour: 时 (整数 1~24)

mday: 日 (整数)

mon: 月 (整数 1~12)

year: 年 (整数)

wday: 星期几 (整数 0~6)

yday: 一年的第几天 (整数 1~366)

isdst: 有无夏令时 (true/false)

zone: 时区 (字符串)

程序名称:bb.rb

```
p Time.now.to_a
```

程序运行结果如图 3-54 所示。

⑥ 返回已经过去的时间,to_f 以浮点数形式返回自起算时间以来经过的秒数,它可以表示那些非整秒的情况。

to_i 和 tv_sec 以整数形式返回自起算时间以来经过的秒数。

to_s 将某时刻变换为形如 date(1)形式的字符串。它等同于:



图 3-54 to_a

```
self.strftime("%a %b %d %H:%M:%S %Z %Y")
```

usec 和 tv_usec 返回某时刻的微秒部分

程序名称:bc.rb

```
p Time.now.to_f
p Time.now.to_i
p Time.now.to_s
p Time.now.usec
p Time.now.tv_usec
```

程序运行结果如图 3-55 所示。

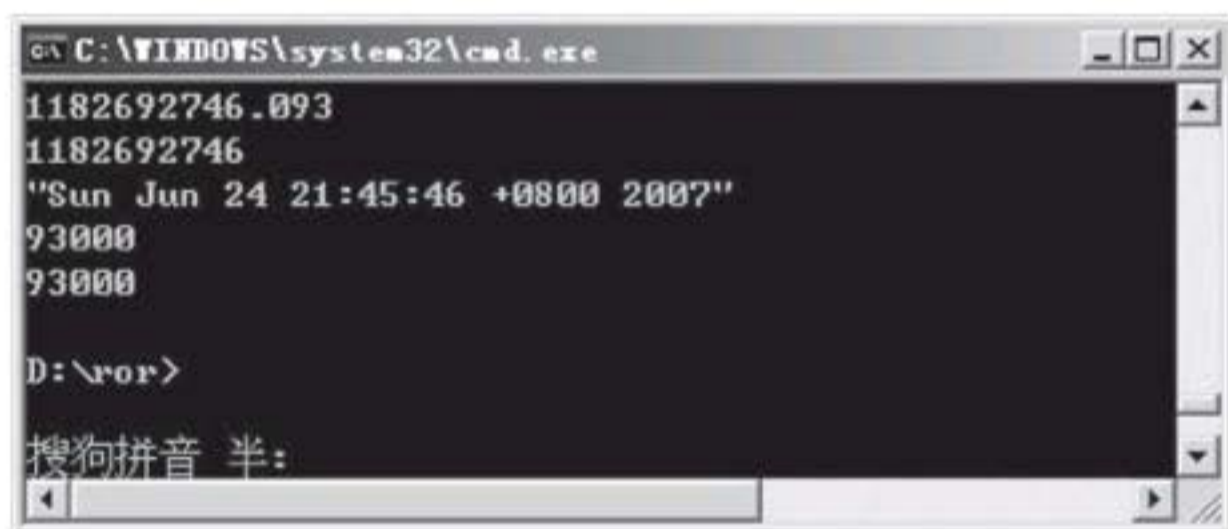


图 3-55 返回已经过去的时间

3.4.2 Date 和 DateTime 对象

Date 是 Ruby 中的日期对象。DateTime 是日期时间对象。它们的功能比 Time 强大。其最常用的方法 parse 可以分析某个表示时间和日期的字符串。

程序名称:bd.rb

```
p Date.parse('6/24/2007').to_s
p DateTime.parse('24- 06- 2007 11:30:00 AM').to_s
p Date.parse('Wednesday, January 10, 2008').to_s
```

程序运行结果如图 3-56 所示。

日期是可以实现迭代的。如下。

程序名称:be.rb



图 3-56 parse 方法

```
(Date.new(2007, 5, 1)..Date.new(2007, 5, 5)).each { |x| puts x }
the_first = Date.new(2007, 6, 1)
the_fifth = Date.new(2007, 6, 5)
the_first.upto(the_fifth) { |x| puts x }
```

程序运行结果如图 3-57 所示。



图 3-57 日期的迭代

3.5 散列表

散列表,也有人翻译为哈希表,一个意思,就是“Hash”,一种数据结构。是由键和值组成的一一映射。

3.5.1 散列表的构造

如下两种方法可用来构造散列表:

Hash[key,value,...]

Hash[hash]

使用第一种形式时,参数的个数必须是偶数(奇数位参数是索引,偶数位参数是元素值)。

使用第二种形式(将一个散列表对象指定给参数)时,将生成并返回一个与指定散列表相同的全新哈希表。(生成的散列表的默认值为 nil)

通常,可以使用 Hash.new 方法构造一个新的空散列表,然后再向散列表中添加元素。

3.5.2 散列表的常用方法

(1) 引用元素

引用散列表的元素,只需使用[key]符号或 fetch 方法。

fetch 方法包括 fetch(key[, default]) 和 fetch(key) {|key| ... } 两种。返回索引 key 对应的元素的值。若该索引未被注册则分为两种情况:若给出了参数 default 的话,就返回它的值;若给出了块,将返回块的计算值。除此之外将引发 IndexError 异常。

(2) 清空散列表

使用 clear 方法,可以清空散列表。

(3) 删除元素

delete(key) 和 delete(key) {|key| ... } 方法,删除索引 key 与对应元素之间的关系,返回被删除的值。若没有值与 key 相对应,则返回 nil。若给出块,将在 key 匹配失败时对块进行计算,然后返回其结果。

delete_if {|key, value| ... } 和 reject! {|key, value| ... } 方法把 key 和 value 当做参数来计算块,若计算值为真则删除相应的元素。通常 delete_if 和 reject! 有所不同,未删除元素则返回 nil,除此之外则返回散列表本身。

(4) 判断散列表是否为空

empty? 方法,若表为空则返回真。

(5) 判断键

has_key? (key)

include? (key)

key? (key)

member? (key)

若 key 是表的键则返回真。

(6) 判断值

has_value? (value)

value? (value)

若 value 是表的元素值则返回真。对值进行判断时使用 == 操作符。

(7) 返回与元素对应的索引

index(val) 返回与 val 对应的索引。若无法对应则返回 nil。若有若干个对应的索引时,则会随机地返回其中之一。

(8) 将元素值和键互换

invert 将元素值和键互换,返回变换后的散列表。

(9) keys

返回一个包含所有索引的数组。

(10) length 和 size

返回散列表中的元素的个数。

(11) rehash

重新计算键对应的值。当与键对应的值发生变化时,若不使用该方法来重新计算的话,将无法取出与键对应的值。

(12) values

返回一个包括散列表中的所有元素值的数组。

(13) to_a

生成并返回一个数组,数组中的元素也是数组,且由[key,value]构成。

(14) values_at(key_1, ..., key_n)

返回一个数组,该数组包括与参数所指键相对应的元素值。

(15) 迭代器

each { |key, value| ... } 和 each_pair { |key, value| ... } 以 key 和 value 为参数对块进行计算。

each_key { |key| ... } 以 key 为参数来计算块。

each_value { |value| ... } 以 value 为参数来计算块。

下面,举一个例子,来说明这些方法的使用。

案例名称:散列表的用法

程序名称:bf.rb

```
# create hash
h= Hash.new
# add
h["a"]= "ask"
h["b"]= "boy"
h.store("c","catch up!")
# keys
p h.keys
# values
p h.values
# array
p h.to_a
p h.values_at("a","c")
# get value
p h["a"]
p h["d"]= "door"
# index
```

```

p h.index("boy")
# size
p h.size
p h.length
# has key
p h.has_key? ("a")
p h.include? ("b")
p h.key? ("c")
p h.member? ("d")
# has value
p h.value? ("ask")
p h.has_value? ("boy")
# delete
p h.delete("a")
p h.size
# clear
h.clear
p h.size
# recreate
h.rehash
h["a"]= "ask"
h["b"]= "boy"
h.store("c","catch up!")
h.each_pair {|k, v| p [k, v]}

```

程序运行结果如图 3-58 所示。

```

C:\WINDOWS\system32\cmd.exe
D:\ror>bf
["a", "b", "c"]
["ask", "boy", "catch up!"]
[["a", "ask"], ["b", "boy"], ["c", "catch up!"]]
["ask", "catch up!"]
"ask"
"door"
"b"
4
4
true
true
true
true
true
true
"ask"
3
{}
["a", "ask"]
["b", "boy"]
["c", "catch up!"]
D:\ror>

```

3.6 区 间

3.6.1 区间的概念

区间就是范围(range)。Ruby 中的区间有三种用途:序列、条件和间隔。

构造区间对象,可以使用标准的 `Range.new(first,last[, exclude_end])` 方法,这个方法生成并返回一个从 `first` 到 `last` 的范围对象。若 `exclude_end` 为真,则该对象不包含范围终点。若省略 `exclude_end`,则包含范围终点。

下面举一个例子,介绍区间的基本构造方法。

案例名称:区间的构造方法

程序名称:bg.rb

```
r= Range.new(1,3,true)
r.each {|i| p i if i< 6}
p "- - - -"
rr= Range.new(1,3)
rr.each {|i| p i if i< 6}
p "- - - -"
ra= 1..3
ra.each {|i| p i if i< 6}
p "- - - -"
rb= 1...3
rb.each {|i| p i if i< 6}
```

程序运行结果如图 3-59 所示。

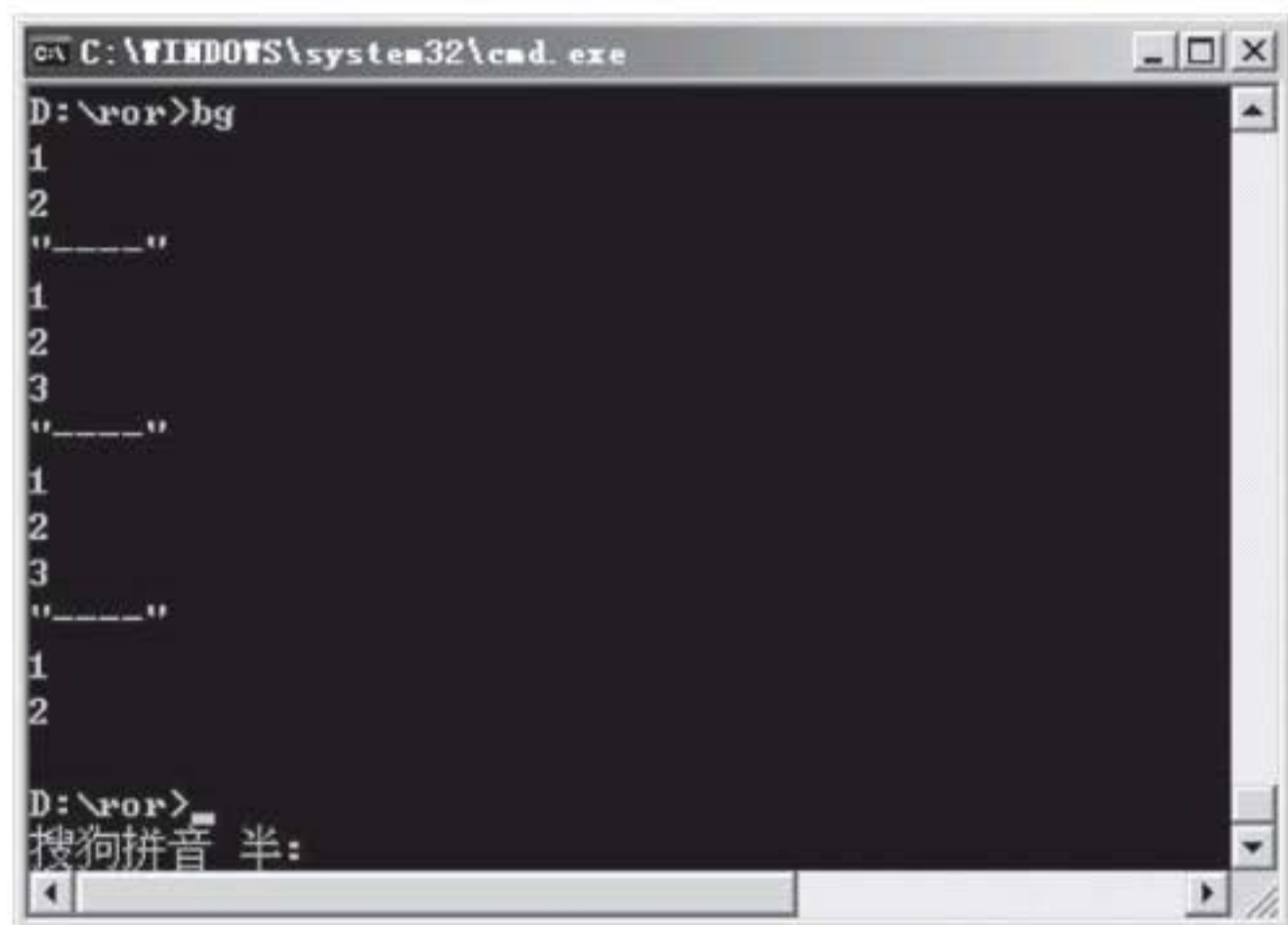


图 3-59 区间的构造方法

3.6.2 区间的使用

区间具有如下的常用方法。

(1) 判断某数值是否在区间内

include? (other)和 member? (other)方法。

案例名称:判断某数值是否在区间内

程序名称:bh.rb

```
p (0.1 .. 0.2).include? (0.15)
p (0.1 .. 0.2).member? (0.15)
p ("a" .. "c").include? ("ba")
p ("a" .. "c").include? ("abd")
p ("a" .. "c").member? ("ba")
p ("a" .. "c").member? ("abd")
```

程序运行结果如图 3-60 所示。



图 3-60 判断某数值是否在区间内

(2) 返回端点元素

begin 和 first 方法返回起始元素,end 和 last 返回终点元素。

案例名称:返回端点元素

程序名称:bi.rb

```
p (0.1 .. 0.2).begin
p (0.1 .. 0.2).first
p ("a" .. "c").end
p ("a" .. "c").last
```

程序运行结果如图 3-61 所示。

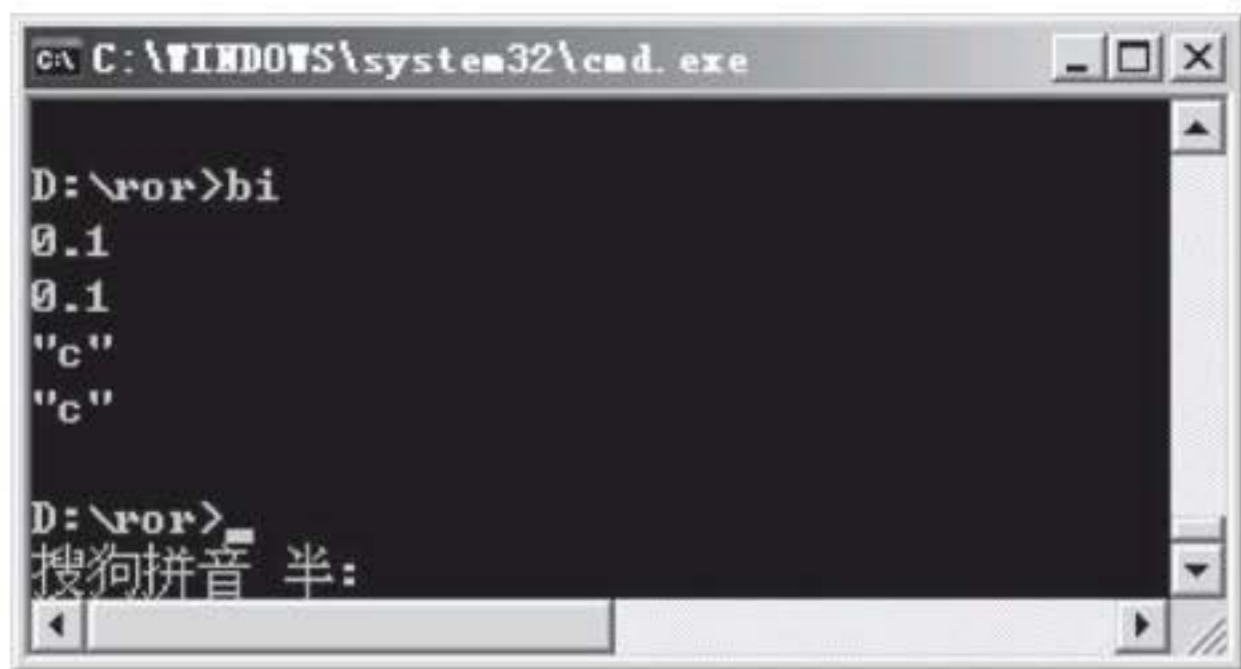


图 3-61 返回端点元素

(3) exclude_end? 方法

区间对象不包含终点时返回真。

案例名称:判断终点

程序名称:bj, rb

```
p ("a" ... "c").exclude_end?  
p ("a" .. "c").exclude_end?
```

程序运行结果如图 3-62 所示。



图 3-62 判断终点

(4) 迭代器

区间对象的迭代器 `step([s]) { |item| ... }`, 以 `s` 的步长对范围内的元素进行迭代操作。`s` 是正整数。默认值为 1。

要返回范围对象的长度, 需要先将其转换成数组, 然后求得数组的长度。

案例名称:迭代器

程序名称:bk, rb

```
p ("a" .. "c").to_a.length  
p ("a" ... "c").to_a.length  
("a" .. "f").step(2) { |v| p v }
```

程序运行结果如图 3-63 所示。



图 3-63 迭代器

3.7 数 组

数组是一种很重要的数据结构,Ruby 对数组的支持是完善的。本节介绍数组的常用方法,尽管只是介绍常用的,但内容还是比较多,原因是:数组确实特别重要。

3.7.1 构造数组

构造数组的方法很多,这一节列出常用的几个基本方法以及数组的常用操作方法,如下面例子所示。

案例名称:构造数组

程序名称:bl.rb

```
p Array.new([1,2,3])
p Array.new(5) {|i| i }
ary = Array.new(3, "foo")
ary.each {|obj| p obj.object_id }
a= ["i","love","you","my","mum"]
p a.size
p a.length
p a[0]
p a.at(0)
p a[1..2]
a[0]= "you"
p a[0]
a[1..2]= ["LOVE","U"]
p a[1..2]
p a* 3
p a* "- "
# 追加
a<< "!"
p a
a.push("!")
p a
```

```
# 删除重复元素
p([1, 2, 1, 3, 1, 4, 1, 5] - [2, 3, 4, 5])
aa= [{"a",15}, {"love",22}, {3,35}]
p aa.assoc("love")
```

程序运行结果如图 3-64 所示。

```
C:\WINDOWS\system32\cmd.exe
D:\ror>hl
[1, 2, 3]
[0, 1, 2, 3, 4]
22918620
22918620
22918620
5
5
"i"
["love", "you"]
"you"
["LOUE", "U"]
["you", "LOUE", "U", "ny", "nun", "you", "LOUE", "U", "ny", "nun", "you", "LOUE",
, "U", "ny", "nun"]
"you-LOUE-U-ny-nun"
["you", "LOUE", "U", "ny", "nun", "\243\241"]
["you", "LOUE", "U", "ny", "nun", "\243\241", "\243\241"]
[1, 1, 1, 1]
["love", 22]
D:\ror>
搜狗拼音 半:
```

图 3-64 构造数组

3.7.2 数组的主要方法

数组很重要,本节列举其主要方法。

(1) 迭代器

`collect! {|item| ...}` 和 `map! {|item| ...}` 依次将数组的各个元素传给块进行计算,然后以计算结果来替换该数组元素。

`each_index {|index| ...}` 依次使用每个元素的索引来对块进行计算。

`each {|item| ...}` 依次使用每个元素来计算块。

`reverse_each {|item| ...}` 对各个元素以逆序对块进行计算。

案例名称:迭代器

程序名称:bm.rb

```
a= [{"a",15}, {"love",22}, {3,35}]
a.each_index{|index| p "tianen: " + index.to_s}
p "- - - - -"
a.each{|item| p item}
p "- - - - -"
a.reverse_each{|item| p "tianen: " + item.to_s}
p "- - - - -"
a= [1,2,3]
```

```

p a
a.map! {|item| item* 3}
p a

```

程序运行结果如图 3-65 所示。

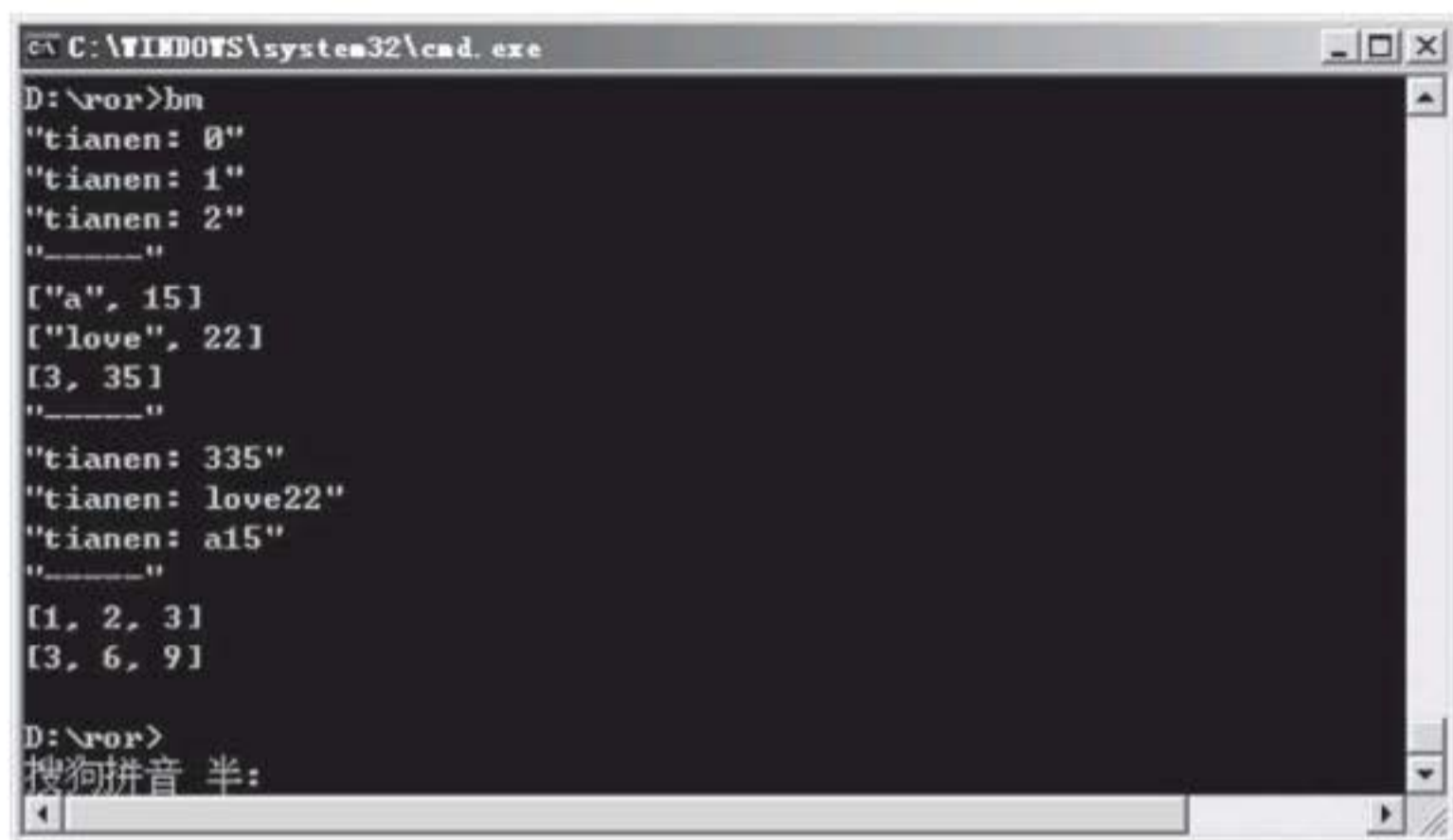


图 3-65 迭代器

(2) compact 和 compact!

compact 从数组中删除值为 nil 的元素后生成新数组并返回它。compact! 是具有破坏性的，若对原数组进行了改动就返回数组，若没有改动则返回 nil。

案例名称: compact 和 compact!

程序名称: bn.rb

```

ary = [1, nil, 2, nil, 3, nil]
p ary.compact
p ary
ary.compact!
p ary
p ary.compact!

```

程序运行结果如图 3-66 所示。

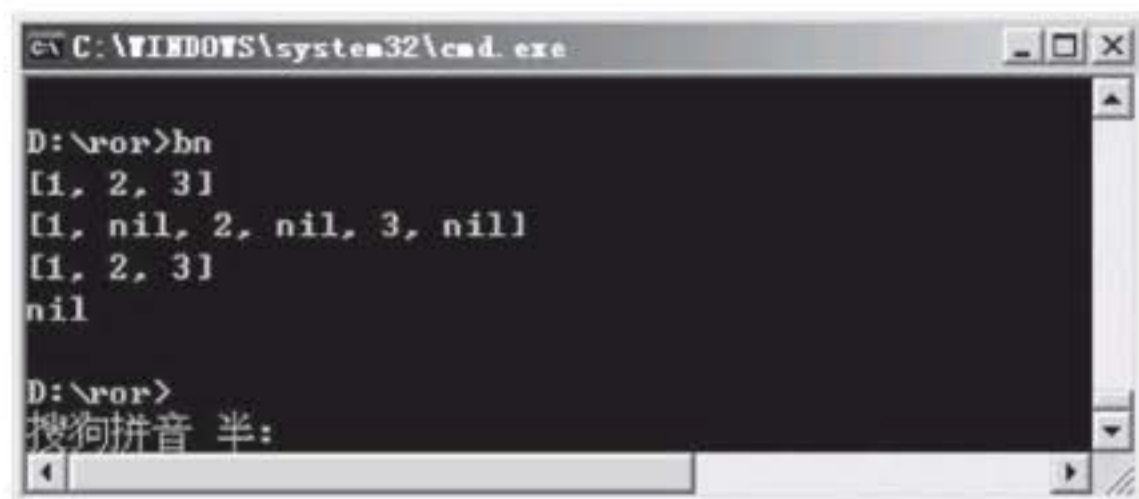


图 3-66 compact 和 compact!

(3) concat(other)

将 other 连接到数组末尾(该动作具有破坏性)。

案例名称:concat(other)

程序名称:bo.rb

```
array = [1, 2]
a      = [3, 4]
array.concat a
p array
p a
```

程序运行结果如图 3-67 所示。



图 3-67 concat(other)

(4) delete

delete(val) 和 delete(val) { ... } 使用 == 来分别比较 val 与每个数组元素,若相等则删除该元素。若发现了与 val 相等的元素就返回 val。若没有发现与 val 相等的元素则返回 nil,若指定了块就对块进行计算并返回结果。

delete_at(pos) 删除 pos 所指位置的元素并返回它。若 pos 超出数组范围则返回 nil。可以使用负的索引从尾部起指定元素。

delete_if { |x| ... } 和 reject! { |x| ... } 依次将元素传给块进行计算,若结果为真就删除相应元素。delete_if 通常返回数组本身。若没有删除任何元素,reject! 会返回 nil。

案例名称:delete

程序名称:bp.rb

```
array = [1, 2, 3, 2, 1]
p array.delete(2)
p array
ary = [nil,nil,nil]
p ary.delete(nil)
p ary
p ary.delete(nil)
array = [0, 1, 2, 3, 4]
array.delete_at 2
p array
```

程序运行结果如图 3-68 所示。

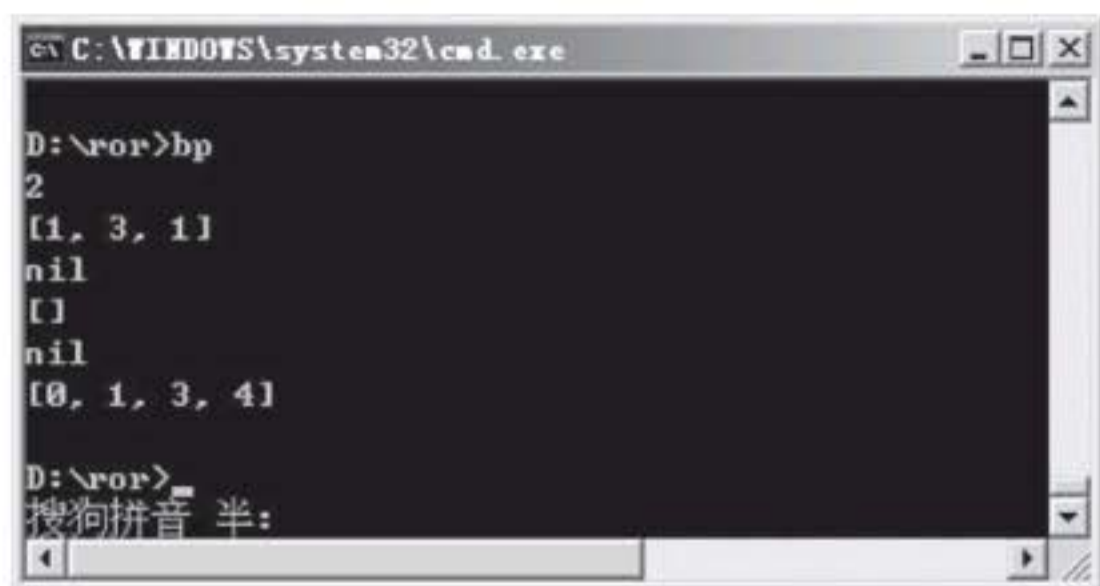


图 3-68 delete

(5) 数组比较

empty? 方法:若数组元素数目为 0 则返回真。

eql? (other) 方法:使用 eql? 来依次计算自身和 other 的各个元素,若所有元素均相等则返回真。

程序名称:bq.rb

```
a = [1, 2, 3]
b = [1, 2, 4]
p a.empty?
p a.eql? (b)
```

程序运行结果如图 3-69 所示。

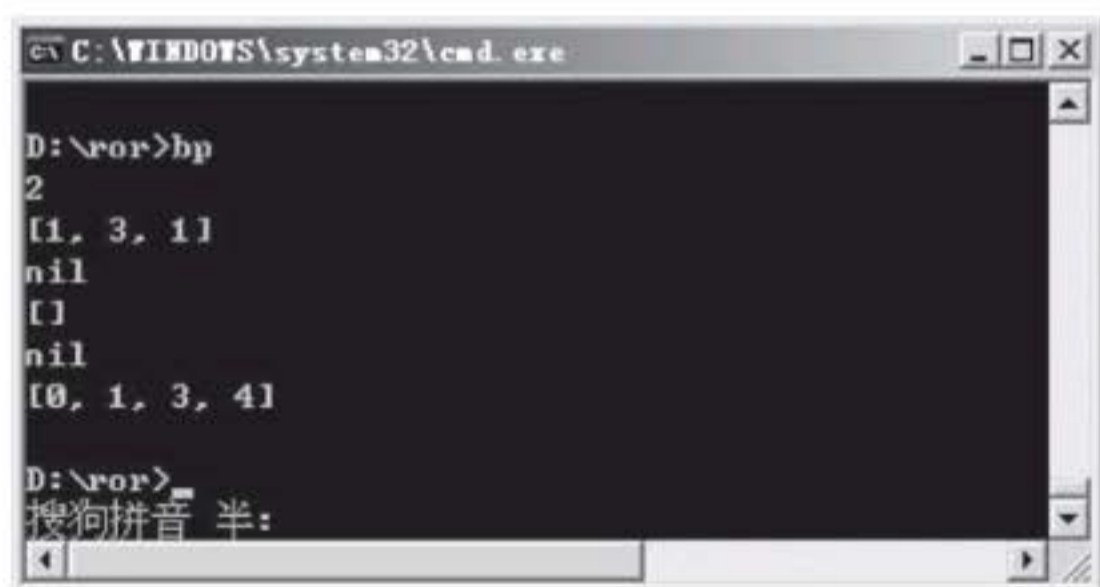


图 3-69 数组比较

(6) 数组元素的提取和修改

first 和 first(n)方法返回数组的首元素。若没有首元素则返回 nil。若指定了可选参数 n ,则以数组形式返回前 n 个元素。 n 必须大于 0。

flatten 和 flatten! 方法将带嵌套的数组重整为不带嵌套的单纯数组,并返回它。flatten! 的重整具有破坏性,若原数组不带嵌套则返回 nil。

`include?(val)` :若数组中包含等于 `val` 的元素就返回真。

`index(val)` 返回数组中第一个等于 `val` 的元素的位置。若没有与其相等的元素则返回 `nil`。

`values_at(index_1, ..., index_n)`方法以数组形式返回其索引值与各参数值相等的元素。若指定了超出范围的索引值,将指派 `nil` 与其对应。

`insert(nth, val[, val2 ...])` 和 `insert(nth, [val[, val2 ...]])`在索引为 `nth` 的元素前面插入第 2 个参数以后的值。

`join([sep])` 将 `sep` 字符串夹在各元素中间使数组转换为字符串,并返回该字符串。若数组元素并非字符串,就调用 `to_s` 然后再进行连接。若元素依然是数组,将再次调用(同样适用 `sep`)`join` 来连接字符串。若 `sep` 为 `nil` 则使用空字符串。

`last` 和 `last(n)`方法返回数组末尾的元素。若数组为空,返回 `nil`。指定了可选参数 `n` 时,将返回末尾的 `n` 个元素。`n` 必须大于 0。

`length` 和 `size` 返回数组长度。若数组为空则返回 0。

`nitems` 返回非 `nil` 元素的个数。

`pop` 删除末尾元素并返回它。若数组为空则返回 `nil`。

程序名称:br.rb

```
p [0, 1, 2].first
p [].first
ary = [0, 1, 2]
p ary.first(0)
p ary.first(1)
p ary.first(2)
p ary.first(3)
p ary.first(4)
p [1, [2, 3, [4], 5]].flatten
array = [[[1, [2, 3]]]]
array.flatten!
p array
ary = %w(a b c d e)
p ary.values_at(0, 2, 4)
p ary.values_at(3, 4, 5, 6, 35)
p ary.values_at(0, -1, -2)
p ary.values_at(-4, -5, -6, -35)
ary = %w(foo bar baz)
ary.insert 2, 'a', 'b'
p ary
ary = [1,2,3]
ary.push ary
p ary
p ary.join
```

```

p [0, 1, 2].last
p [].last
ary = [0, 1, 2]
p ary.last(0)
p ary.last(1)
p ary.last(2)
p ary.last(3)
p ary.last(4)
array = [1, [2, 3], 4]
p array.pop
p array.pop
p array
p array.pop
p array.pop
p array

```

程序运行结果如图 3-70 所示。

```

C:\WINDOWS\system32\cmd.exe
D:\ror>br
[]
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2]
[1, 2, 3, 4, 5]
[1, 2, 3]
["a", "c", "e"]
["d", "e", nil, nil, nil]
["a", "e", "d"]
["b", "a", nil, nil]
["foo", "bar", "a", "b", "baz"]
[1, 2, 3, [...]]
"123123[...]"
2
nil
[]
[2]
[1, 2]
[0, 1, 2]
[0, 1, 2]
4
[2, 3]
[1]
1
nil
[]
D:\ror>
搜狗拼音 半:

```

图 3-70 数组的方法

(7) 数组的其他方法

`rassoc(obj)` 方法:某个数组的元素中也会有数组。(如 `a=[[1,2],3,[3,4],[4,4,4]]`)对于其中的数组元素 `x`,若 `x[1]=obj`,那么 `a,rassoc(obj)` 就会将靠前的 `x` 数组取出来。如 `a,rassoc(4)` 为 `[3,4]`,`a,rassoc(2)` 为 `[1,2]`。

`replace(another)` 以数组 `another` 来替换该数组的内容。

`reverse` 和 `reverse!`:`reverse` 将所有元素以逆序重新排列生成新数组并返回它。`reverse!` 的逆序排列过程具有破坏性。`reverse` 通常返回新数组。若数组只有 1 个元素,`reverse!` 会返回 `nil`。

`slice(pos[, len])` 和 `slice(start..last)` 相当于 `[]` 的用法。类似的还有 `slice!(pos[, len])` 和 `slice!(start..last)`。

`sort`,`sort!`,`sort {|a, b| ... }`,`sort! {|a, b| ... }` 对数组内容进行排序。若带块调用,将把 2 个参数传给块,然后使用块的计算结果进行比较。若没有块时,则使用 `<=>` 操作符进行比较。`sort` 将生成一个经过排序的新数组并返回它。`sort!` 对 `self` 的排序过程具有破坏性。

`transpose` 将数组看作由行和列构成的矩阵,然后进行行列互调(将行和列互换),生成一个新数组并返回它。若原数组为空,则生成空数组并返回它。若原数组为一维数组则会引发 `TypeError` 异常。若各个元素中包含的子元素个数不一时,会引发 `IndexError` 异常。

`uniq` 和 `uniq!`:`uniq` 会删除数组中的重复元素然后生成新数组并返回它。剩下的元素会向前移动。`uniq!` 具有破坏性,若进行了删除则返回新数组,若没有删除则返回 `nil`。

`unshift(obj1[, obj2 ...])` 和 `unshift([obj1[, obj2 ...]])` 依次将 `obj1`, `obj2 ...` 插到数组的头部。

程序名称:bs.rb

```
array = [1, 2, 3]
array.push 4
array.push [5, 6]
array.push 7, 8
p array
a = [[15,1], [25,2], [35,3]]
p a.rassoc(2)
a = [1, 2, 3]
a.replace [4, 5, 6]
p a
p [[1,2],
   [3,4],
   [5,6]].transpose
p [1, 1, 1].uniq
p [1, 4, 1].uniq
p [1, 3, 2, 2, 3].uniq
```

```

arr = [1,2,3]
arr.unshift 0
p arr
arr.unshift [0]
p arr
arr.unshift 1, 2
p arr

```

程序运行结果如图 3-71 所示。

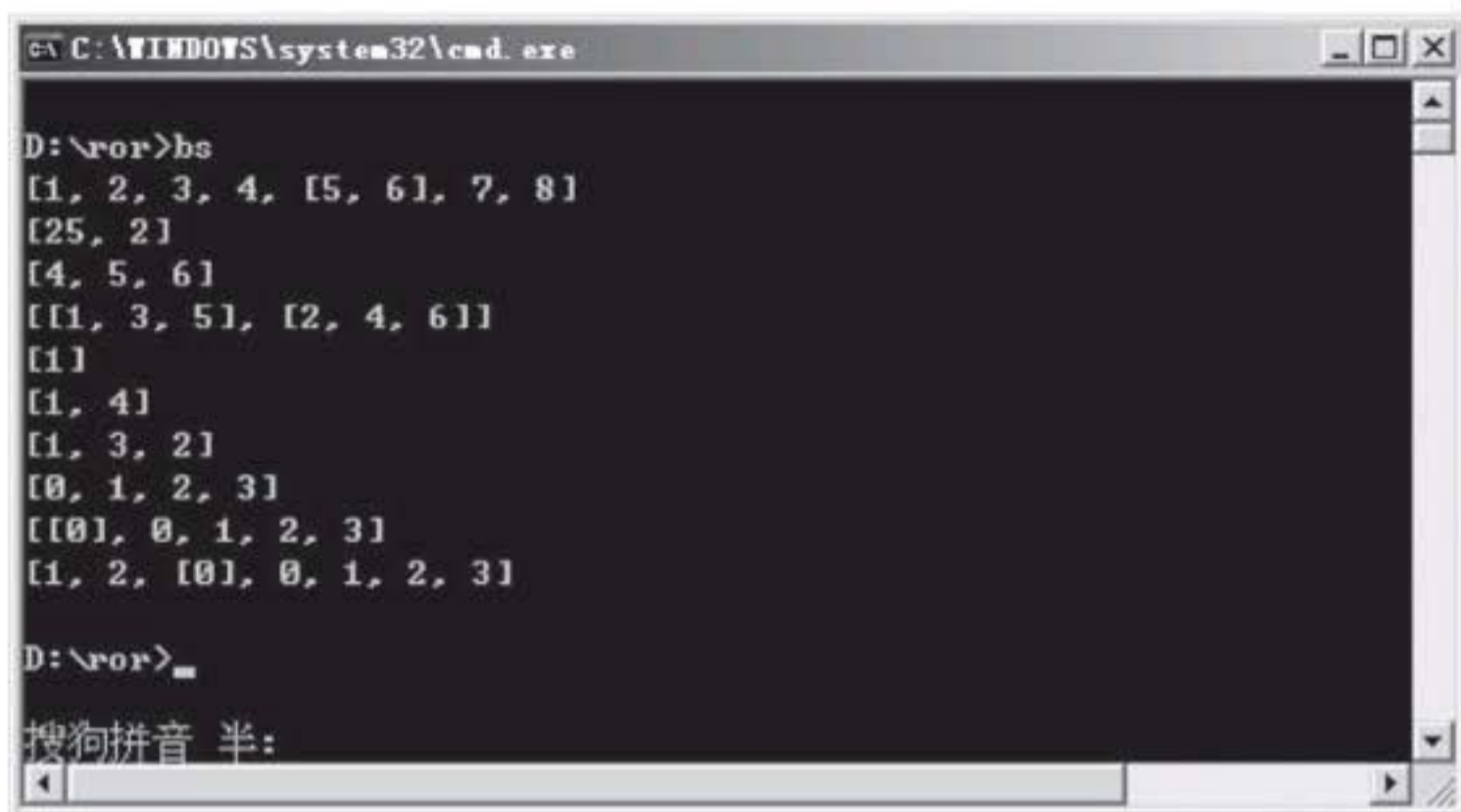


图 3-71 数组的方法

3.8 结构体

结构体就是 Struct 类, Ruby 支持 Struct, 但使用得并不多。

3.8.1 建立结构体

建立结构体需使用 Struct 的类方法 new。如下:

```
Struct.new([name,] member ...)
```

生成并返回一个名为 name 的 Struct 类的子类。子类中定义了访问结构体成员的方法。

例如:

```

g = Struct.new("Girl", :name, :age)
lucy = g.new("lucy", 22)
p lucy.age

```

结构体名 name 将成为 Struct 的类常数名, 所以必须以大写字母开始。member 可以是符号(以冒号开头的变量)或字符串。建议使用符号, 否则容易出错。

若省略 name(第一参数为符号), 就会生成无名的结构体类。当首次要求无名类提供其类名时, 它会搜索被赋值的常数名, 若找到就以该常数名为类名。

如

```
p Struct.new("Girl", :name, :age)
=> Struct::Girl
p Girl = Struct.new(:name, :age)
=> Girl
```

3.8.2 结构体的主要方法

结构体具有如下主要方法。

(1) []符号

返回结构体的第 `nth` 个成员的值。若指定了并不存在的成员,会引发 `IndexError` 或 `NameError` 异常。可以使用负的索引值,查询方向相反。

(2) [nth]=value

将结构体的第 `nth` 个成员的值设定为 `value`,并返回 `value` 值。若指定了并不存在的成员,会引发 `IndexError` 或 `NameError` 异常。

(3) length 和 size

返回结构体的成员数量。

(4) members

以数组形式返回结构体的成员名(字符串)。

(5) values 和 to_a

将结构体的成员的值存入数组,并返回它。

(6) values_at(member_1, ... member_n)

以数组的形式返回参数所指成员的值。若指定了并不存在的成员,会引发 `IndexError` 或 `NameError` 异常。

下面举一个例子,来说明这些方法的使用情况。

案例名称:结构体的主要方法

程序名称:bt.rb

```
g = Struct.new("Girl", :name, :age)
lucy = g.new("lucy", 22)
p lucy.length
p lucy.size
p lucy.age
p lucy[0]
p lucy[1]
p lucy[-1]
lucy.age= 23
p lucy.age
```

```
lucy[0]= "big lucy"
p lucy[0]
m= lucy.members
p m
p lucy.values
p lucy.to_a
p lucy.values_at(0)
p lucy.values_at(1)
```

程序运行结果如图 3-72 所示。

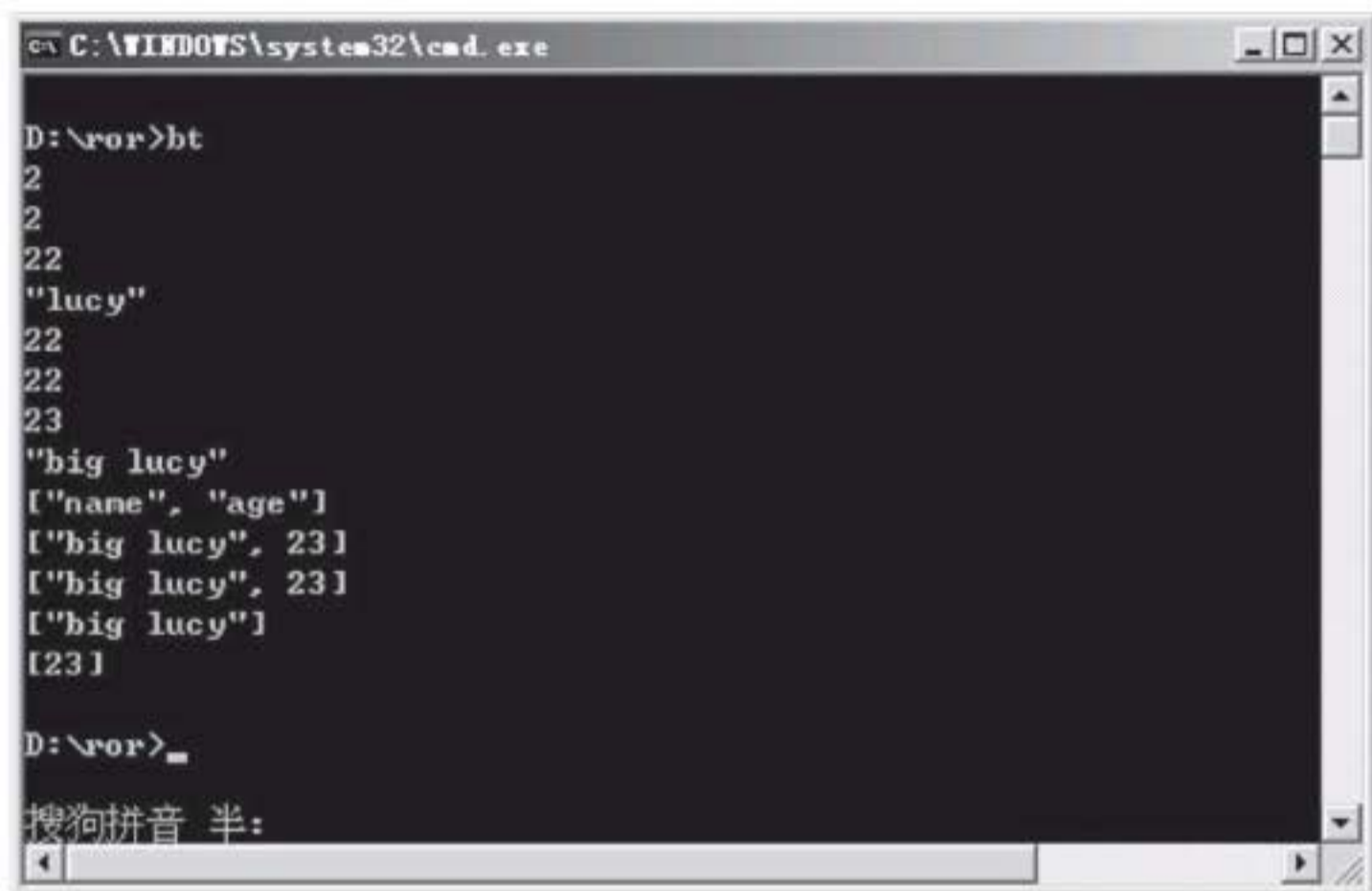


图 3-72 数组的方法

3.9 数据类型转换

至此,笔者已经介绍了大部分常用的数据类型。在本章的最后,来介绍一下这些数据类型之间的转换。

3.9.1 通用的转换方法

在 Object 类中定义了如下方法,用来实现数据类型转换。这些方法是一切对象共有的。

(1) to_ary

将对象转换为数组并返回该数组。将对象隐式地转换为数组时,在内部自动调用。若某对象中定义了该方法,就可以把它单独置入多重赋值表达式的右边进行调用。

(2) to_hash

将对象隐式地转换为散列表时,在内部自动调用。

(3) to_int

将对象隐式地转换为整数时在内部自动调用。

(4) to_s

将对象转换为字符串并返回该字符串。如果把非字符串对象当作参数传给 print 或 sprintf, 将调用该方法把非字符串对象转换为字符串。

(5) to_str

将对象隐式地转换为字符串时自动调用它。

(6) to_f

将字符串看作 10 进制数形式, 并将其变为浮点数 Float。

(7) to_i 和 to_i(base)

将字符串看作 10 进制数形式, 并将其变为整数。

程序名称: bu.rb

```
p "0xa.a".to_f
p " \n10".to_f
p "1_0_0".to_f
p "".to_f
p " 10".to_i
p "010".to_i
p "- 010".to_i
p "0x11".to_i
p "0b10".to_i(0)
p "0o10".to_i(0)
p "010".to_i(0)
p "0d10".to_i(0)
p "0x10".to_i(0)
p "10".to_f
p "10e2".to_f
p "1e- 2".to_f
p ".1".to_f
p "nan".to_f
p "INF".to_f
p "- Inf".to_f
p (("10" * 1000).to_f)
```

程序运行结果如图 3-73 所示。

```
C:\WINDOWS\system32\cmd.exe

D:\ror>bu
0.0
10.0
100.0
0.0
10
10
-10
0
2
8
8
10
16
10.0
1000.0
0.01
0.1
0.0
0.0
0.0
D:/ror/bu.rb:26: warning: Float 10101010101010101010... out of range
Infinity

D:\ror>
```

图 3-73 类型转换

3.9.2 自定义转换方法

系统提供的通用类型转换方法有时是不够的,但大多数时候都够用。当另外需要把时间字符串转换成时间对象或者诸如面对此类的其他转换需求的时候,就要自定义类型转换方法。其原则是:将复杂类型分解成简单类型,在简单类型中调用系统的原始类型转换方法。

小 结

Ruby 有多种数据类型,如:数字,数组,字符串,区间,时间日期,散列表,块,正则表达式等。前一章中只简要介绍了字符串和数字类型,本章详述了各种数据类型的用法。

思考和练习

编写一段程序,实现把阿拉伯数字转化成大写的汉字功能。如:“123.6”转化为“壹佰贰拾叁点陆”。

第 4 章 模块和线程

本章要点

本章介绍 Ruby 的两个基本概念：模块和线程。

4.1 模 块

4.1.1 模块的概念

模块类似于类和结构体，在 Ruby 中它是方法和属性的集合体。不同的模块之间可以相互引用，互相调用方法。但，模块不是类，不能实例化。模块实现了类似 Java 中的包和，NET 中的命名空间之类的结构，便于组织代码。

(1) 模块的定义

定义模块使用 `module` 关键字，模块中方法和属性的定义与类类似。引用模块中的方法使用点（“.”），引用模块中属性使用两个冒号“::”。

案例名称：模块的基本使用

程序名称：a.rb

```
module My
  NA= "China"
  def My.set_name(name)
    @ name= name
  end
  def My.get_name
    return @ name
  end
  def My.set_age(age)
    @ age= age
  end
end

My.set_name("tianen")
p My.get_name
p My::NA
```

程序运行结果如图 4-1 所示。



图 4-1 模块的基本使用

(2) 模块的引入

不同的属性和方法可以定义在不同的模块中,然后,可以在一个模块中引用另一个模块,从而在一个模块中调用另一个模块的方法和属性。若不同的模块在不同的文件中,要先使用 `load?"filename.rb"` 或 `require?"filename"` 方法将模块所在的文件载入。`load` 方法每次执行都会包含一个 Ruby 源文件,而 `require` 只会包含一个文件一次。

案例名称:模块的引入

程序名称:b.rb

```
module My
  NA= "China"
  def My.set_name(name)
    @ name= name
  end
  def My.get_name
    return @ name
  end
  def My.set_age(age)
    @ age= age
  end
end

My.set_name("tianen")
p My.get_name
p My::NA
module Your
  NA= "AC"
  attr:name
  def Your.set_name(name)
    My.set_name(name)
  end
  def Your.get_name
    return My.get_name
  end
end
```

```

def Your.set_age(age)
  @ age= age
end
def Your.get_age
  return @ age
end
end
Your.set_name("lucy")
p Your.get_name
Your.set_age("22")
p Your.get_age
p Your::NA

```

程序运行结果如图 4-2 所示。



图 4-2 基本模块

程序名称:c.rb

```

require "b"
p My::NA

```

程序运行结果如图 4-3 所示。



图 4-3 模块的引入

4.1.2 Mixin

可以在一个类的定义里包含一个模块,这时候,这个模块中所有的实例方法都变成了在这个类所拥有的方法了,这就叫作 Mixin。

模块不可以实例化,定义实例变量被类包含之后就成为了类的实例变量。使用类包含模块要用 include 语句。

案例名称:Mixin

程序名称:d.rb

```
module My
  NA= "China"
  attr:name
  attr:age
  def set_name(name)
    @ name= name
  end
  def get_name
    return @ name
  end
  def set_age(age)
    @ age= age
  end
end
class Test
  include My
end
t= Test.new
t.set_name("Tianen")
p t.get_name
p t.name
```

程序运行结果如图 4-4 所示。



图 4-4 Mixin

4.1.3 Ruby 的命名约定

学过了模块之后,现在可以完整地总结出一个 Ruby 的命名约定。如下所列。

局部变量和方法参数以小写字母开头。
方法名字以小写字母开头。
全局变量以 \$ 开头。
实例变量以 @ 开头。
类变量以两个 @ 开头。
常量以大写字母开头(它们经常被指定全部大写)。
类和模块名以大写字母开头。

4.2 线 程

4.2.1 线程的概念

Ruby 使用的线程是用户级线程,由 Ruby 解释器进行切换管理。其效率要低于由 OS 管理线程的效率,且不能使用多个 CPU。但,便于移植,这一点优于 Java。

可以使用 Thread.start 方法来生成新线程。如:Thread.start { ... }

下面举一个例子,来看一下线程的使用。

案例名称:线程的概念

程序名称:e.rb

```
Thread.start {  
  while true  
    print "thread 1\n"  
  end  
}  
while true  
  print "thread 2\n"  
end
```

程序运行时,两个无限循环在同时运作,交替执行。按 Ctrl-C 终止程序。结果如图 4-5 所示。

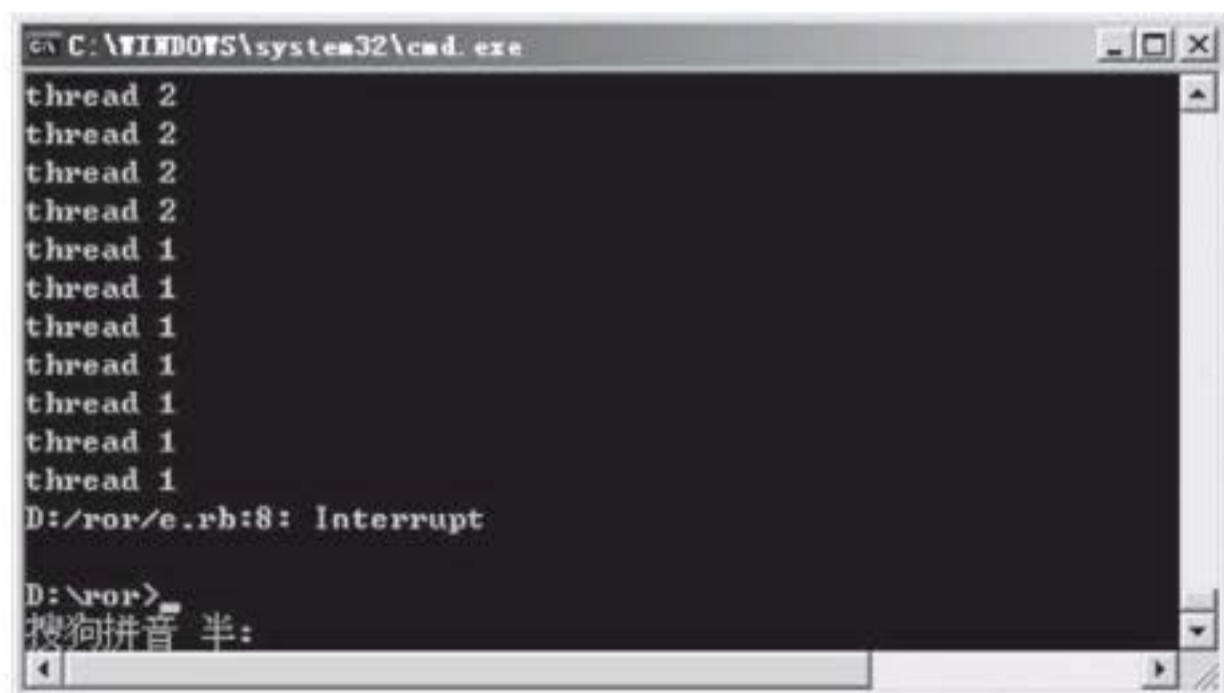


图 4-5 线程的概念

线程的主要方法如下。

- ① Thread.start {…} 和 Thread.new {…}: 生成新线程, 并对迭代程序块进行判断。返回新生成的线程对象。new 是 start 的别名。
- ② Thread.current: 返回当前运行的线程对象。
- ③ Thread.exit: 终止当前运行的线程对象。
- ④ Thread.join thread: 挂起现在的线程, 直到指定线程运行结束为止。
- ⑤ Thread.kill thread: 终止指定线程的运行。
- ⑥ Thread.pass: 将控制权显式地交给其他可运行的线程。
- ⑦ Thread.stop: 挂起现在的线程, 直到其他线程运行 thread#run 为止。
- ⑧ Thread#exit: 终止 receiver 线程。
- ⑨ Thread#run: 重新开启 receiver 线程。
- ⑩ Thread#stop: 挂起 receiver 线程。
- ⑪ Thread#status: 若 receiver 线程存在则返回真。若线程因错误而终止, 则引发那个错误。
- ⑫ Thread#value: 返回判断 receiver 迭代程序块的结果。若判断迭代程序块的过程尚未完成, 则等到该线程终止为止。

案例名称: 线程的方法

程序名称: f.rb

```
i= 1
Thread.start{
  while true
    print "thread 1\n"
    i+ = 1
    if i== 5 then
      Thread.kill Thread.current
    end
  end
}
j= 1
while true
  print "thread 2\n"
  j+ = 1
  if j== 4 then
    Thread.exit
  end
end
end
```

程序运行结果如图 4-6 所示。



图 4-6 线程的方法

4.2.2 线程的同步

线程共享内存空间,同步很重要。若同步失败可能引发死锁。

Ruby 的线程库提供了两种同步方法:Mutex 和 Queue。前者只负责同步,后者还负责数据交接。

(1) Mutex

Mutex 是互斥锁。若对 Mutex 加锁时发现已经处于锁定状态时,线程会挂起直到解锁为止。

Mutex 具有如下方法。

Mutex.new 生成新的互斥锁。

Mutex# lock 加锁。若已经处于加锁状态则会一直等待下去,直到解锁为止。

Mutex# unlock 解锁。若有其它等锁的线程则会让它们通过。

Mutex# synchronize 执行从获得锁到解锁全过程的迭代器。

Mutex# try_lock 获得锁。若已处于加锁状态,则返回 false 且不会挂起。

在并行访问中保护共享数据时,可以使用下列代码。

```
m:Mutex.new
begin
  m.lock
  # 访问受 m 保护的共享数据
ensure
  m.unlock
end
```

Mutex 有个 synchronize 方法可以简化这一过程。

```
m.synchronize {
  # 访问受 m 保护的共享数据
}
```

下面举一个例子,从中可以看到互斥锁的作用。

案例名称:互斥锁

程序名称:g.rb

```
m = Mutex.new
v = 0
Thread.start{
  m.synchronize{
    while v< 300
      v = v + 100
      p v
    end
  }
}
m.synchronize{
  while v> 0
    v = v - 33
    p v
  end
}
p "- - - - -"
m = Mutex.new
v = 0
Thread.start{
  while v< 300
    v = v + 100
    p v
  end
}
m.synchronize{
  v = v - 33
  p v
}
```

程序运行结果如图 4-7 所示。

可以看到,前者,使用了互斥锁,变量 v 先后被两块代码引用。后者没有使用互斥锁,变量 v 被两个线程争抢。

(2) Queue

Queue 就像一条读写数据的管道。提供数据的线程在一边写入数据,而读取数据的线程则在另一边读出数据。若 Queue 中没有可供读取的数据,则读取数据的线程会挂起等待数据的到来。

Queue 有下列方法。

Queue.new 生成新的 Queue。

Queue.empty? 若 Queue 为空则返回真。

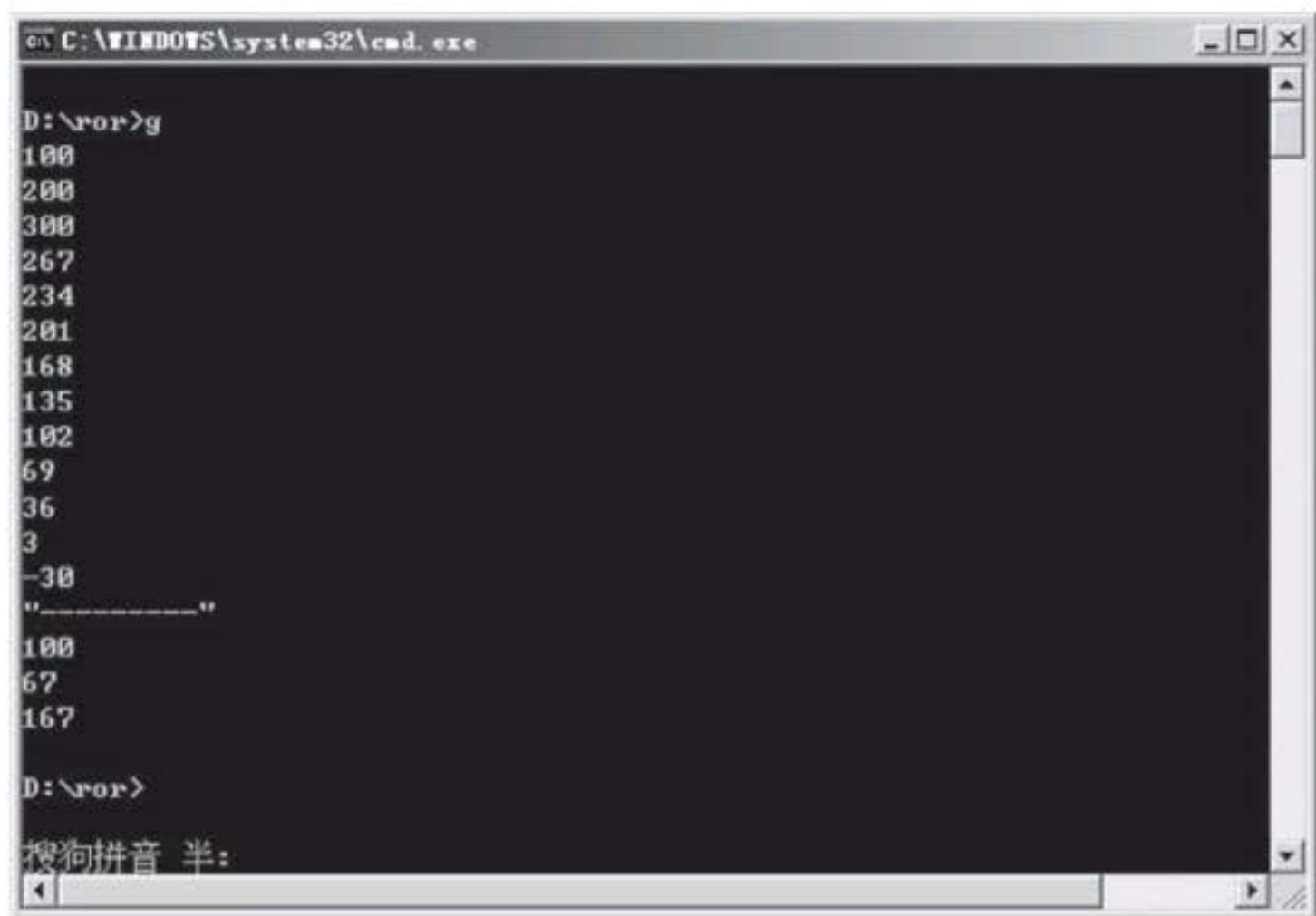


图 4-7 互斥锁

Queue.push value 向 Queue 添加 value。

Queue.pop [non_block] 从 Queue 中取出数据。若参数 non_block 被指定为非假值而且 Queue 为空,则引发错误。其他情况下,若 Queue 为空,读取数据的线程会被挂起直到有新数据加入。

下面举一个例子,可以看到 Queue 的基本使用方法。

案例名称:Queue

程序名称:h.rb

```
q = Queue.new
t = Thread.start {
  while line = q.pop
    print line
  end
}
while gets
  q.push $ _
end
q.push nil
t.join
```

程序运行结果如图 4-8 所示。

程序运行时,等待用户输入,用户输入内容后回车,然后输入的内容就被反显出来。



图 4-8 Queue

小 结

本章介绍了 Ruby 的两个基本概念:模块和线程。这两个概念是常用的,需要掌握。

思考和练习

练习使用模块的 Mixin。

第三部分 Ruby 的高级知识

第 5 章 文件和目录

本章要点

文件和目录操作在任何编程语言中都是非常重要的。本章介绍 Ruby 中实现文件和目录操作的相关类和方法。

5.1 文件操作

5.1.1 文件操作的概念

Ruby 中与文件操作相关的类和模块为：File 类、IO 类、File::Stat 类、File::Constants 模块以及 FileTest 模块。

笔者不面面俱到地介绍这些类和模块，本节介绍的文件操作的主要内容就是：

- ① 打开文件；
- ② 读取文件内容；
- ③ 修改文件内容；
- ④ 关闭文件；
- ⑤ 新建、移动、删除文件。

除了这些内容之外，还要学一些与文件属性相关的操作方法。本节就围绕这些内容来介绍文件的操作方法，如果读者需要更多更详细的 API，可以查看手册。

5.1.2 文件的基本操作方法

(1) IO 类

IO 类是 File 类的父类，尽管使用 IO 类直接操作文件并不方便，笔者在此也不推荐，但作为一种操作文件的基本方式，读者需要做到了解。

IO 类具有如下方法。

- ① 生成 IO 对象：生成 IO 对象，要使用这四个类方法

IO.new(fd[, mode])

IO.for_fd(fd[, mode])

IO.open(fd[, mode])

IO.open(fd[, mode]) { |io| ... }

② `IO.foreach(path[, rs]) {|line| ... }`:对 `path` 所指文件的各行内容进行迭代操作(与 `open` 一样,若 `path` 以 `"|"` 开头则读取命令输出)。若成功打开 `path` 就返回 `nil`,若失败则引发 `Errno::EXXX` 异常。

使用 `rs` 指定的字符串进行行的切分,`rs` 的默认值取自内部变量 `$/`。

若将 `rs` 指定为 `nil`,则意味着忽略行的切分。若指定为空字符串 `""`,将把连续的换行当作行的切分符(段落模式)。

③ `IO.popen(command [, mode])` 和 `IO.popen(command [, mode]) {|io| ... }`:将 `command` 作为子进程来运行,然后建立一条连接到该进程的输入输出的管线(`pipeline`)。`mode` 指定打开 IO 端口的模式。

④ `IO.read(path,[length],[offset])`:在 `path` 指文件中,从 `offset` 位置开始读入 `length` 字节的内容并将其返回。若 `length` 为 `nil` 或被省略,将一直读到 EOF。

当 IO 到达 EOF 时返回 `nil`。

若无法打开 `path`,或者设定 `offset` 所指位置失败,又或者无法读入文件内容,都将引发 `Errno::EXXX` 异常。当 `length` 为负数,则会引发 `ArgumentError` 异常。

⑤ `IO.readlines(path[, rs])`:读入 `path` 指定的文件的全部内容后,以行为单位将其转化为数组并返回该数组。若 IO 已经到达 EOF 返回空数组 `[]`。使用 `rs` 指定的字符串进行行的切分,`rs` 的默认值取自内部变量 `$/`。

若将 `rs` 指定为 `nil`,则意味着忽略行的切分。若指定为空字符串 `""` 的话,将把连续的换行当作行的切分符(段落模式)。

若无法打开 `path` 或者读取文件失败,将引发 `Errno::EXXX` 异常。

⑥ `IO.select(reads[, writes[, excepts[, timeout]])`:传递给 `reads/writes/excepts` 的是等待输入的 IO(或其子类)的实例的数组。

`timeout` 可以是整数、float 或 `nil`(省略时的默认值)。指定为 `nil` 时,将会一直等到 IO 变成就绪状态。

`timeout` 时将返回 `nil`,除此以外将返回一个包含 3 个元素的数组,这 3 个元素分别是等待输入/输出/异常的对象数组(指定数组的子集)。

⑦ `IO.sysopen(path[, mode [, perm]])`:打开 `path` 指定的文件并返回文件描述符。若无法打开文件时将引发 `Errno::EXXX` 异常。

参数 `mode`, `perm` 的具体情况与内部函数 `open` 相同。

如果不使用 `IO.for_fd` 等方法来转化为 IO 对象,就无法关闭使用该方法打开的文件。

⑧ `<< object`:输出 `object`。若 `object` 并非字符串,将使用 `to_s` 方法将其变为字符串。类似 C++。

⑨ `binmode`:将流变为二进制模式。该方法只在那些支持二进制模式的 OS,如 MSDOS 中有效(除此以外将毫无反应)。若想从二进制模式恢复到普通模式,只能重新打开。

⑩ `clone` 和 `dup`:返回一个全新的 IO 对象,该对象与 `receiver` 使用同一个 IO。

⑪ `close`:关闭输入输出端口。此后,若对该端口进行输入输出则会引发 `IOError` 异常。进行垃圾回收时,未关闭的 IO 端口将被关闭。

⑫ `close_read`:关闭读取用的 IO。主要用在管道或读写两用的 IO 对象上。

⑬ `close_write`:关闭写入用的 IO。

⑭ closed?: 端口已被关闭则返回真。

⑮ each([rs]) { |line| ... } 和 each_line([rs]) { |line| ... }: 每次从 IO 端口读入 1 行来进行相关处理的迭代器。IO 端口必须是以读模式打开的。使用 rs 指定的字符串进行行的切分, rs 的默认值取自内部变量 \$/。

若将 rs 指定为 nil, 则意味着忽略行的切分。若指定为空字符串 "", 将把连续的换行当作行的切分符(段落模式)。

⑯ each_byte { |ch| ... }: 每次从 IO 端口读入 1 字节的内容。IO 端口必须是以读模式打开的。

⑰ eof 和 eof?: 流(stream)到达文件末端则返回真。

⑱ fileno 和 to_i: 返回文件描述符的号码。

⑲ flush: 刷新 IO 端口的内部缓冲区。

⑳ getc: 从 IO 端口读入 1 个字符, 并返回该字符对应的 Fixnum。到达 EOF 时返回 nil。

㉑ gets([rs]): 读入一行, 若成功就返回读入的字符串。到达文件末尾时返回 nil。使用 rs 所指字符串进行行的切分, rs 的默认值取自内部变量 \$/。

若将 rs 指定为 nil, 则意味着忽略行的切分。若指定为空字符串 "", 将把连续的换行当作行的切分符(段落模式)。

IO#gets 与 gets 一样, 将读入的字符串赋值给变量 \$_。

㉒ lineno: 返回当前的行号。

㉓ lineno=number: 设定行号。

㉔ pid: 若是由 IO.popen 生成的 IO 端口, 则返回子进程的进程 ID。除此以外返回 nil。

㉕ pos 和 tell: 返回文件指针的当前位置。

㉖ pos = n: 将文件指针移动到指定位置。与 io.seek(pos, IO::SEEK_SET) 相同。

㉗ print([arg[, ...]]): 依次将参数输出到 IO 端口。省略参数时将输出 \$_。该方法处理参数的方式与 print 相同。

㉘ printf(format[, arg[, ...]]): 与 C 语言的 printf 一样, 按照 format 将参数变为字符串, 然后输出。

㉙ putc(ch): 输出字符 ch 到对象。该方法处理参数的方式与 putc 相同。

㉚ puts([obj[, ...]]): 输出各个 obj 然后换行。该方法处理参数的方式与 puts 相同。

㉛ read([length]) 和 read([length[, buf]]): 若给出了 length, 就读取 length 字节的内容, 然后返回该字符串。若 IO 已到达 EOF 则返回 nil。若将某字符串传给第二参数, 将使用读入的数据来覆盖该字符串对象, 并返回读入的数据。当该字符串的长度刚好等于 length 时, 将不会进行内存分配。当该字符串的长度不等于 length 时, 将暂时把字符串的长度拉长(或缩短)到 length 的长度(随后又将变成实际读入数据的长度)。

若在调用 read 方法时使用了第二参数且数据为空(read 返回 nil 时), buf 将变为空字符串。

㉜ readchar: 与 IO#getc 一样, 读入 1 个字符后返回与该字符相对应的 Fixnum。若到达 EOF 则引发 EOFError 异常。

㉝ readline([rs]): 与 IO#gets 一样, 读入 1 行后返回读入的字符串。若到达 EOF 则引发 EOFError 异常。

使用 `rs` 指定的字符串进行行的切分, `rs` 的默认值取自内部变量 `$/`。

若将 `rs` 指定为 `nil`, 则意味着忽略行的切分。若指定为空字符串 `""`, 将把连续的换行当作行的切分符(段落模式)。

`readline` 与 `gets` 一样, 将读入的字符串赋值给变量 `$_`。

③④ `readlines([rs])`: 读入所有数据后, 以行为单位将其转化为数组, 并返回该数组。若 IO 已到达 EOF 则返回空数组 `[]`。

使用 `rs` 指定的字符串进行行的切分, `rs` 的默认值取自内部变量 `$/`。

若将 `rs` 指定为 `nil`, 则意味着忽略行的切分。若指定为空字符串 `""`, 将把连续的换行当作行的切分符(段落模式)。

③⑤ `reopen(io)` 和 `reopen(name[, mode])`: 将自身转接到 `io` 上。类也将等同于 `io` (请注意)。

若第一个参数为字符串时, 将把流转接到 `name` 所指的文件上。

第二个参数的默认值为 `"r"`。

③⑥ `rewind`: 将文件指针移动到头部。 `IO#lineno` 变为 0。

③⑦ `seek(offset[, whence])`: 将文件指针由 `whence` 移动到 `offset`。 `whence` 的值是下列之一。

`IO::SEEK_SET`: 从文件头部开始(默认)。

`IO::SEEK_CUR`: 从当前文件指针开始。

`IO::SEEK_END`: 从文件尾部开始。

省略 `whence` 时, 其默认值为 `IO::SEEK_SET`。

若向着 `offset` 移动成功就返回 0。若失败则引发 `Errno::EXXX` 异常。

若读取数据失败则引发 `Errno::EXXX` 异常。若 `length` 为负数则引发 `ArgumentError` 异常。

③⑧ `stat`: 生成并返回一个包含文件状态的 `File::Stat` 对象。

若未能成功读取状态则引发 `Errno::EXXX` 异常。

③⑨ `sync`: 以 `true` 和 `false` 来表示当前的输出同步模式。当输出同步模式为真时, 每次调用输出函数时都会刷新缓冲区。

④⑩ `sync=newstate`: 设定输出同步模式。 `newstate` 为 `true` 则表示同步模式, `false` 则表示非同步模式。返回 `newstate`。

④⑪ `sysread(length)` 和 `sysread(length[, buf])`: 进行输入, 并返回包含输入数据的字符串。到达文件尾部时会引发 `EOFError` 异常。因其不使用 `stdio`, 所以如果和 `gets`、`getc` 或 `eof?` 等混合使用会引起意想不到的混乱。

若不能成功读取数据则引发 `Errno::EXXX` 异常。

若将某字符串传给第二个参数, 将使用读入的数据来覆盖该字符串对象, 并返回读入的数据。若该字符串的长度刚好等于 `length` 时, 将不会进行内存分配。若该字符串的长度不等于 `length` 时, 将暂时把字符串的长度拉长(或缩短)到 `length` 的长度, 随后又将变成实际读入数据的长度。

若在调用 `sysread` 方法时使用了第二个参数且数据为空(`sysread` 引发 `EOFError` 异常时), `buf` 将变为空字符串。

④ `sysseek(offset[, whence])`:若将 `seek` 与 `sysread`, `syswrite` 混用会引起混乱,此时应该使用本方法。

若对已缓冲的读取 IO 执行该方法,会引起 `IOError` 异常。若对已缓冲的写入 IO 执行该方法,会发出警告。

④ `syswrite(string)`:输出 `string`。若 `string` 非字符串,将使用 `to_s` 尝试将其变为字符串。本方法不使用 `stdio`,因此如果与其他输出方法混用会引起意想不到的问题。

返回实际输出的字节数。若输出失败则引发 `Errno::EXXX` 异常。

④ `ungetc(char)`:送回(unreading) `char`。若送回 2 字节以上的内容,则无法保证其准确性。

④ `write(str)`:对 IO 端口输出 `str`。若 `str` 并非字符串,将使用 `to_s` 尝试将其变为字符串。

除 `IO#syswrite` 之外的所有输出方法最后都会调用一个名为“`write`”的方法,因此替换本方法就可以改变输出函数的行为。

返回实际输出的字节数。若输出失败则引发 `Errno::EXXX` 异常。

(2) `File::Stat` 类

`File::Stat` 类用于保存文件信息的对象的类。通过这个类可以返回文件的创建时间、读写权限等相关属性。

这个类具有如下属性和方法:

① `File::Stat.new(path)`:生成并返回一个关于 `path` 的 `File::Stat` 对象,与 `File.stat` 相同。

② `ftype`:返回一个表示文件类型的字符串,该字符串应是下列之一。

"file"

"directory"

"characterSpecial"

"blockSpecial"

"fifo"

"link"

"socket"

"unknown"

在下列属性方法中,若系统不支持某方法就会返回 `nil`。

③ `dev`:设备号(文件系统)。

④ `ino`:i-node 号。

⑤ `mode`:文件模式。

⑥ `nlink`:hard link 数。

⑦ `uid`:owner 的 `userID`。

⑧ `gid`:owner 的 `groupID`。

⑨ `rdev`:设备类型(只有专用文件)。

⑩ `rdev_major`:`rdev` 的 `major` 号码部分。

⑪ `rdev_minor`:`rdev` 的 `minor` 号码部分。

- ⑫ size:文件大小(以字节为单位)。
- ⑬ blksize:理想的 I/O 的块大小。
- ⑭ blocks:被分配的块数。
- ⑮ atime:最终访问时间。
- ⑯ mtime:最终更新时间。
- ⑰ ctime:最终 i-node 变更时间。
- ⑱ directory?:若为目录则为真。
- ⑲ readable?:若可读则为真。
- ⑳ readable_real?:若实用户/实组可以读取就为真。
- ㉑ writable?:若可写则返回真。
- ㉒ writable_real?:若实用户/实组可以写入就为真。
- ㉓ executable?:若有效用户/组 ID 能执行就为真。
- ㉔ executable_real?:若实用户/组 ID 能执行就为真。
- ㉕ file?:若为普通文件则返回真。
- ㉖ zero?:若大小为 0 就为真。
- ㉗ size?:文件大小(若为 0 则为伪)。
- ㉘ owned?:若属于自己则为真。
- ㉙ grpowned?:若组 ID 与执行组 ID 相等则为真。
- ㉚ pipe?:若是带名称管道(FIFO)则为真。
- ㉛ symlink?:若是符号连接则为真。
- ㉜ socket?:若是 socket 则为真。
- ㉝ blockdev?:若是块专用文件则为真。
- ㉞ chardev?:若是字符专用文件则为真。
- ㉟ setuid?:若被 setuid 则为真。
- ㊱ setgid?:若被 setgid 则为真。
- ㊲ sticky?:若设定了粘着位(sticky bit)则为真。

下面来做一些练习,使用 IO 类读写文件。

案例名称:使用 IO 类读写文件

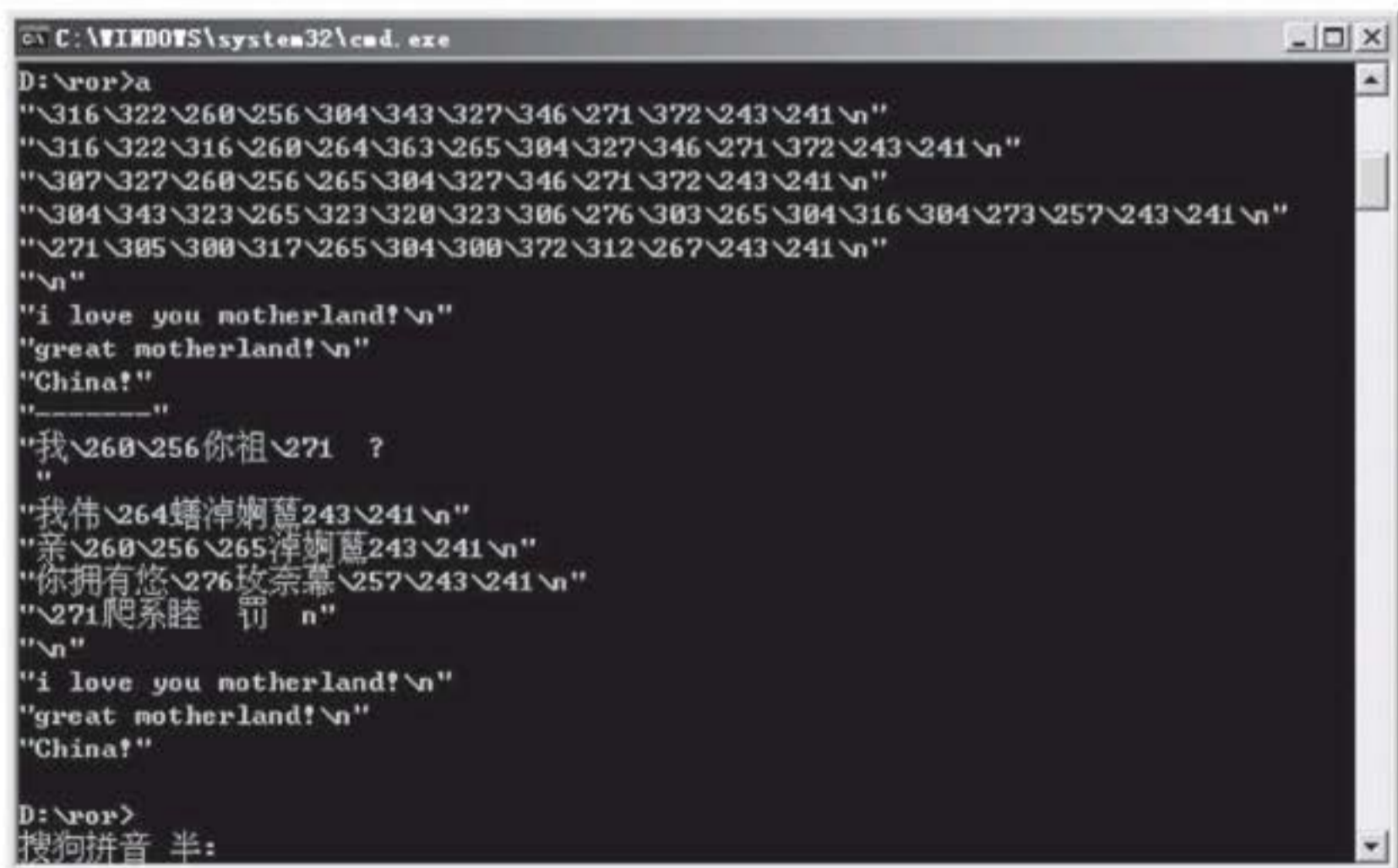
程序名称:test.txt

```
我爱你祖国!  
我伟大的祖国!  
亲爱的祖国!  
你拥有悠久的历史!  
古老的历史!  
i love you motherland!  
great motherland!  
China!
```

程序名称:a.rb

```
path= "test.txt"
port = open(path)
begin
  port.each_line {|line|
    p line.to_s
  }
ensure
  port.close
end
p "- - - - -"
$ KCODE= 'u'
port = open(path)
begin
  port.each_line {|line|
    p line.to_s
  }
ensure
  port.close
end
```

程序运行结果如图 5-1 所示。



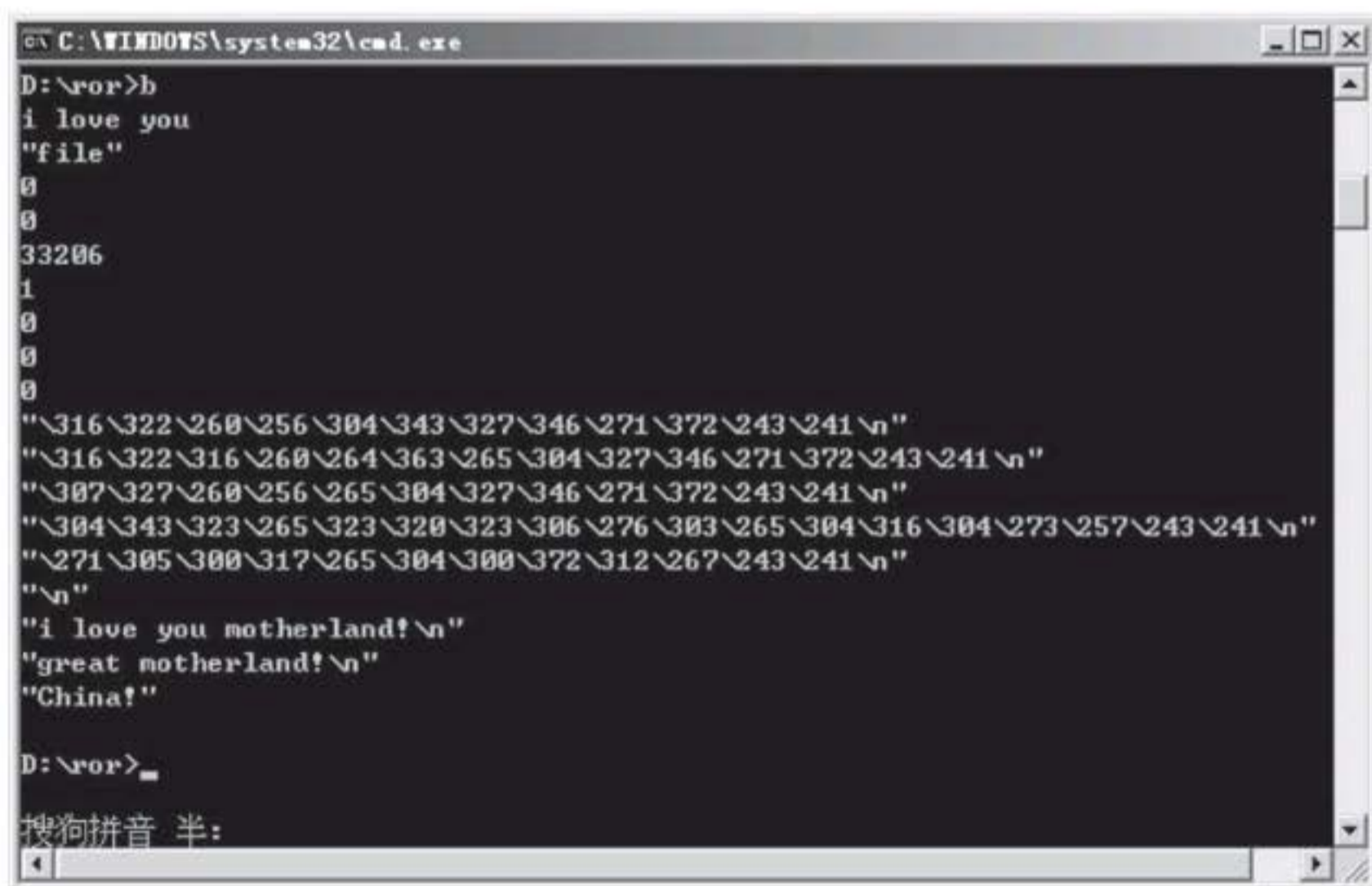
```
C:\WINDOWS\system32\cmd.exe
D:\ror>a
"\316\322\260\256\304\343\327\346\271\372\243\241\n"
"\316\322\316\260\264\363\265\304\327\346\271\372\243\241\n"
"\307\327\260\256\265\304\327\346\271\372\243\241\n"
"\304\343\323\265\323\320\323\306\276\303\265\304\316\304\273\257\243\241\n"
"\271\305\300\317\265\304\300\372\312\267\243\241\n"
"\n"
"i love you motherland!\n"
"great motherland!\n"
"China!"
"-----"
"我\260\256你祖\271 ?
"
"我伟\264蟾淖婀蕙243\241\n"
"亲\260\256\265淖婀蕙243\241\n"
"你拥有悠\276玫奈慕\257\243\241\n"
"\271爬系睦 罚 n"
"\n"
"i love you motherland!\n"
"great motherland!\n"
"China!"
D:\ror>
搜狗拼音 半:
```

图 5-1 使用 IO 类读写文件

程序名称:b.rb

```
STDOUT << "i love you \n"
port = open("test.txt")
s= port.stat
p s.ftype
p s.dev
p s.ino
p s.mode
p s.nlink
p s.uid
p s.gid
p s.rdev
begin
  a= port.readlines
  a.each{|e| p e}
ensure
  port.close
end
```

程序运行结果如图 5-2 所示。



```
C:\WINDOWS\system32\cmd.exe
D:\ror>b
i love you
"file"
0
0
33206
1
0
0
0
"\316\322\260\256\304\343\327\346\271\372\243\241\n"
"\316\322\316\260\264\363\265\304\327\346\271\372\243\241\n"
"\307\327\260\256\265\304\327\346\271\372\243\241\n"
"\304\343\323\265\323\320\323\306\276\303\265\304\316\304\273\257\243\241\n"
"\271\305\300\317\265\304\300\372\312\267\243\241\n"
"\n"
"i love you motherland!\n"
"great motherland!\n"
"China!"
D:\ror>_
搜狗拼音 半:
```

图 5-2 使用 IO 类读写文件

程序名称:c.rb

```
path= "test.txt"
s= File::Stat.new(path)
print "s.ftype:", s.ftype, "\n"
print "s.dev:", s.dev, "\n"
print "s.ino:", s.ino, "\n"
print "s.mode:", s.mode, "\n"
print "s.nlink:", s.nlink, "\n"
print "s.uid:", s.uid, "\n"
print "s.gid:", s.gid, "\n"
print "s.rdev:", s.rdev, "\n"
print "s.rdev_major", s.rdev_major, "\n"
print "s.rdev_minor:", s.rdev_minor, "\n"
print "s.size:", s.size, "\n"
print "s.blksize:", s.blksize, "\n"
print "s.blocks:", s.blocks, "\n"
print "s.atime:", s.atime, "\n"
print "s.mtime:", s.mtime, "\n"
print "s.ctime:", s.ctime, "\n"
print "s.directory?:", s.directory?, "\n"
print "s.readable?:", s.readable?, "\n"
print "s.readable_real?:", s.readable_real?, "\n"
print "s.writable?:", s.writable?, "\n"
print "s.writable_real?:", s.writable_real?, "\n"
print "s.executable?:", s.executable?, "\n"
print "s.executable_real?:", s.executable_real?, "\n"
print "s.file?:", s.file?, "\n"
print "s.zero?:", s.zero?, "\n"
print "s.size?:", s.size?, "\n"
print "s.owned?:", s.owned?, "\n"
print "s.grpowned?:", s.grpowned?, "\n"
print "s.pipe?:", s.pipe?, "\n"
print "s.symlink?:", s.symlink?, "\n"
print "s.socket?:", s.socket?, "\n"
print "s.blockdev?:", s.blockdev?, "\n"
print "s.chardev?:", s.chardev?, "\n"
print "s.setuid?:", s.setuid?, "\n"
print "s.setgid?:", s.setgid?, "\n"
print "s.sticky?:", s.sticky?, "\n"
```

程序运行结果如图 5-3 所示。

```
C:\WINDOWS\system32\cmd.exe

D:\ror>c
s.ftype:file
s.dev:3
s.ino:0
s.mode:33188
s.nlink:1
s.uid:0
s.gid:0
s.rdev:3
s.rdev_major:nil
s.rdev_minor:nil
s.size:129
s.blksize:nil
s.blocks:nil
s.atime:Mon Jun 25 00:00:00 +0800 2007
s.mtime:Mon Jun 25 14:04:48 +0800 2007
s.ctime:Mon Jun 25 13:40:37 +0800 2007
s.directory?:false
s.readable?:true
s.readable_real?:true
s.writable?:true
s.writable_real?:true
s.executable?:false
s.executable_real?:false
s.file?:true
s.zero?:false
s.size?:129
s.owned?:true
s.grpowned?:false
s.pipe?:false
s.symlink?:false
s.socket?:false
s.blockdev?:false
s.chardev?:false
s.setuid?:false
s.setgid?:false
s.sticky?:false

D:\ror>_
搜狗拼音 半:
```

图 5-3 读取文件属性

5.1.3 文件操作标准方法

文件操作的标准方法是使用 File 类。它有两个辅助模块,File::Constants 和 FileTest。

File::Constants 模块囊括了与 File 类有关的常数,File 类包含该模块,因此可以像使用 File 类的常数那样来使用这些常数。

下列常数用于 File#flock:

LOCK_SH 共享锁。可同时被多个进程共享的锁。

LOCK_EX 互斥锁。只能单独被某进程使用的锁。

LOCK_UN 解锁。

LOCK_NB 锁定时不会进入阻塞状态。使用 or 后,可与其他常数并用。

下列常数用于 File.open:

RDONLY

WRONLY
RDWR
APPEND
CREAT
EXCL
NONBLOCK
TRUNC
NOCTTY
BINARY
SYNC

下列常数用于 File, fnmatch 和 Dir, glob 中:

FNM_NOESCAPE
FNM_PATHNAME
FNM_PERIOD
FNM_CASEFOLD
FNM_DOTMATCH

FileTest 模块中包含了文件测试函数。

模块函数如下:

- ① FileTest, blockdev? (filename): 若 filename 是块专用文件, 就返回 true。
- ② FileTest, chardev? (filename): 若 filename 是字符专用文件, 就返回 true。
- ③ FileTest, executable? (filename): 若能用有效用户/组 ID 来执行 filename, 就返回 true。
- ④ FileTest, executable_real? (filename): 若能用实用户/组 ID 来执行 filename 的话, 就返回真。
- ⑤ FileTest, exist? (filename): 若 filename 确实存在, 就返回真。
- ⑥ FileTest, grpowned? (filename): 若 filename 的组 ID 与执行组 ID 相等, 就返回 true。
- ⑦ FileTest, directory? (filename): 若 filename 是目录名, 就返回 true。
- ⑧ FileTest, file? (filename): 若 filename 是普通文件, 就返回 true。
- ⑨ FileTest, pipe? (filename): 若 filename 是带名的管道(FIFO), 就返回 true。
- ⑩ FileTest, socket? (filename): 若 filename 是 socket, 就返回 true。
- ⑪ FileTest, owned? (filename): 若 filename 属于自己, 则返回 true。
- ⑫ FileTest, readable? (filename): 若 filename 可读, 则返回 true。
- ⑬ FileTest, readable_real? (filename): 若 filename 可被实用户/实组所读取时, 返回 true。
- ⑭ FileTest, setuid? (filename): 若 filename 被 setuid(2)的话, 就返回 true。
- ⑮ FileTest, setgid? (filename): 若 filename 被 setgid(2)的话, 就返回 true。
- ⑯ FileTest, size(filename): 返回 filename 的大小, 若 filename 不存在, 则引发 Errno::EXXX(可能是 Errno::ENOENT)异常。
- ⑰ FileTest, size? (filename): 返回 filename 的大小, 若 filename 不存在或 filename 的大

小为 0 时,返回 nil。

⑱ FileTest.sticky? (filename):若 filename 的 sticky 位为真,则返回 true。

⑲ FileTest.symlink? (filename):若 filename 是符号连接,则返回 true。

⑳ FileTest.writable? (filename):若 filename 可写,则返回 true。

㉑ FileTest.writable_real? (filename):若 filename 可被实用户/实组所写入时,返回 true。

㉒ FileTest.zero? (filename):若 filename 确实存在,且其大小为 0 时,就返回真。若 filename 不存在,则返回 false。

File 类具有如下方法。

① 构造方法:使用 open 或 File.open 来生成 File 对象。

② File.atime(filename), File.ctime(filename) 和 File.mtime(filename):它们分别返回:文件的最终访问时间/文件状态的最终变更时间/最终更新时间(Time 对象)。若未能成功取得文件时间则引发 Errno::EXXX 异常。

③ File.basename(filename[, suffix]):返回 filename 中最后一条斜线后面的部分。若给出了参数 suffix 且它和 filename 的尾部一致时,该方法会将其删除并返回结果。

④ File.chmod(mode[, filename[, ...]]) 和 File.lchmod(mode[, filename[, ...]]):将文件的模式改为 mode。它返回修改文件的个数。若修改模式失败则引发 Errno::EXXX 异常。

⑤ File.chown(owner, group[, filename[, ...]]) 和 File.lchown(owner, group[, filename[, ...]]):修改文件的 owner 和 group。只有超级用户才能修改文件的 owner 和 group。它返回修改文件的个数。若修改失败则引发 Errno::EXXX 异常。

⑥ File.delete(filename ...) 和 File.unlink(filename ...):删除文件。返回删除文件的个数。若删除失败则引发 Errno::EXXX 异常。该方法是用来删除普通文件的。而删除目录时要使用 Dir.rmdir。

⑦ File.dirname(filename):以字符串的形式返回 filename 中最后一条斜线之前的部分。若文件名中不含斜线,则返回"."(当前目录)。

⑧ File.expand_path(path[, default_dir]):将 path 展开为绝对路径后返回该路径字符串。若 path 是相对路径则以 default_dir 为基准进行扩展。若没有 default_dir 或其值为 nil 时将使用当前目录。

开头的 ~ 会被扩展为主目录(使用环境变量 HOME),而 ~ USER 会被扩展为相应用户的主目录。

⑨ File.extname(filename):返回文件名 filename 中的扩展名部分(跟在最后的"."之后的部分)。目录名中的"."和文件名头上的"."不会被看作扩展名的一部分。若 filename 中不含扩展名则返回空字符串。

⑩ File.fnmatch(pattern, path[, flags]) 和 File.fnmatch?(pattern, path[, flags]):对文件名进行模式匹配。若 path 与 pattern 相匹配则返回真。pattern 中可使用的通配符有'*','?'和'['(与 Dir.glob 不同的是,这里不能使用'{'或'* */')。

flags 中可以任意使用下列常数(定义在 File::Constants 模块中)中的若干个来改变模式匹配的运作情况。flags 的默认值为 0(不指定标识)。

FNM_NOESCAPE

将转义字符\'看作普通字符。

FNM_PATHNAME

通配符'*', '?', '['将不再匹配'/'。主要作用在 shell 的模式匹配中。

FNM_CASEFOLD

进行匹配时,不区分大小写字母。

FNM_DOTMATCH

通配符'*', '?', '['将匹配于开头的'.'。

⑪ `File.ftype(filename)`:返回表示文件类型的字符串。该字符串应为下列之一。与 `File.lstat(filename).ftype` 相同(若为符号连接则返回"link")。

```
"file"  
"directory"  
"characterSpecial"  
"blockSpecial"  
"fifo"  
"link"  
"socket"  
"unknown"
```

⑫ `File.join(item, item, ...)`:将 `File::SEPARATOR` 置入其中连成字符串。它与 `[item, item, ...].join(File::SEPARATOR)` 相同。

⑬ `File.link(old, new)`:生成一个指向 `old` 且名为 `new` 的硬连接(hard link)。old 必须是现存的。若成功生成硬连接就返回 0。若失败则引发 `Errno::EXXX` 异常。

⑭ `File.new(path[, mode[, perm]])`, `File.open(path[, mode[, perm]])` 和 `File.open(path[, mode[, perm]]) {|file| ... }`:打开 `path` 所指文件并返回文件对象。若打开文件失败时引发 `Errno::EXXX` 异常。参数 `mode`, `perm` 的用法与内部函数 `open` 相同。

`open()`可以带块。若带块调用时,将把文件对象传给块,然后开始块的运行。当块终止运行时,文件将被自动关闭。

带块调用时的返回值是块的执行结果。

⑮ `File.readlink(path)`:以字符串形式返回字符连接的连接对象。若读取连接失败则引发 `Errno::EXXX` 异常。

⑯ `File.rename(from, to)`:修改文件名称。若目录不同时,将进行移动。若目标目录中有同名文件则会进行覆盖。若成功移动文件就返回 0。若失败则引发 `Errno::EXXX` 异常。

⑰ `File.split(pathname)`:将 `pathname` 分为 `dirname` 和 `basename`,并返回一个包含这两部分的数组。它与 `[File.dirname(pathname), File.basename(pathname)]` 相同。

⑱ `File.stat(filename)` 和 `File.lstat(filename)`:生成并返回一个包含 `filename` 信息的 `File::Stat` 对象。若获取信息失败则引发 `Errno::EXXX` 异常。`lstat` 返回符号连接中的连接本身的信息。

⑲ `File.symlink(old, new)`:生成一个指向 `old` 且名为 `new` 的符号连接。若成功生成符号连接则返回 0。若失败则引发 `Errno::EXXX` 异常。

⑳ `File.truncate(path, length)`:将 `path` 指定的文件裁减为至多 `length` 字节的文件。若成功更改大小时返回 0。若失败则引发 `Errno::EXXX` 异常。

㉑ `File.umask([umask])`:修改 `umask`。返回修改前的 `umask` 值。若省略 `umask` 则不修改,返回当前的 `umask` 值。

㉒ `File.utime(ctime, mtime[, filename[, ...]])`:修改文件的最终访问时间和更新时间。返回修改的文件数。若修改失败则引发 `Errno::EXXX` 异常。开始的两个参数必须是,表示时间的数值或 `Time` 类的实例。

㉓ `File.blockdev?(path)`:与 `FileTest.blockdev?` 相同。

㉔ `File.chardev?(path)`:与 `FileTest.chardev?` 相同。

㉕ `File.directory?(path)`:与 `FileTest.directory?` 相同。

㉖ `File.executable?(path)`:与 `FileTest.executable?` 相同。

㉗ `File.executable_real?(path)`:与 `FileTest.executable_real?` 相同。

㉘ `File.exist?(path)`:与 `FileTest.exist?` 相同。

㉙ `File.file?(path)`:与 `FileTest.file?` 相同。

㉚ `File.grpowned?(path)`:与 `FileTest.grpowned?` 相同。

㉛ `File.owned?(path)`:与 `FileTest.owned?` 相同。

㉜ `File.pipe?(path)`:与 `FileTest.pipe?` 相同。

㉝ `File.readable?(path)`:与 `FileTest.readable?` 相同。

㉞ `File.readable_real?(path)`:与 `FileTest.readable_real?` 相同。

㉟ `File.setgid?(path)`:与 `FileTest.setgid?` 相同。

㊱ `File.setuid?(path)`:与 `FileTest.setuid?` 相同。

㊲ `File.size(path)`:与 `FileTest.size` 相同。

㊳ `File.size?(path)`:与 `FileTest.size?` 相同。

㊴ `File.socket?(path)`:与 `FileTest.socket` 相同。

㊵ `File.sticky?(path)`:与 `FileTest.sticky?` 相同。

㊶ `File.symlink?(path)`:与 `FileTest.symlink?` 相同。

㊷ `File.writable?(path)`:与 `FileTest.writable?` 相同。

㊸ `File.writable_real?(path)`:与 `FileTest.writable_real?` 相同。

㊹ `File.zero?(path)`:与 `FileTest.zero?` 相同。

`File` 类的实例方法如下。

① `ctime`, `mtime` 和 `atime`:它们分别返回:文件的最终访问时间/文件状态的最终变更时间/最终更新时间(`Time` 对象)。若未能成功取得文件时间则引发 `Errno::EXXX` 异常。

② `chmod(mode)`:将文件模式修改为 `mode`。若修改成功则返回 0。若失败则引发 `Errno::EXXX` 异常。

③ `chown(owner, group)`:修改文件的 `owner` 和 `group`。只有超级用户才能修改文件的 `owner` 和 `group`。修改成功则返回 0,失败则引发 `Errno::EXXX` 异常。

④ `flock(operation)`:锁定文件。锁定成功时返回 0。失败则引发 `Errno::EXXX` 异常。若在 `LOCK_NB` 状态下发生阻塞(block)则返回 `false`。下列是可用的 `operation` 列表:

`LOCK_SH`

共享锁。若干个进程可同时共享锁定。

根据系统的不同,有时要求锁定对象必须是以读取模式("r","r+"等)打开的才行。在这些系统中,若对不可读文件进行锁定时,可能会引发 `Errno::EBADF` 异常。

LOCK_EX

排它锁。在某时刻,只有一个进程能控制锁。

根据系统的不同,有时要求锁定对象必须是以写入模式("w","r+"等)打开的才行。在这些系统中,若对不可写文件进行锁定时,可能会引发 `Errno::EBADF` 异常。

LOCK_UN

解锁。

除了这种显式的解锁以外,当 Ruby 解释器结束(进程结束)时,也会自动解锁。

LOCK_NB

非阻塞模式。同 `LOCK_SH | LOCK_NB` 一样,它可以借用 `or` 与其他 operation 一起使用。若没有指定 `LOCK_NB` 的话,若在阻塞条件下调用 `flock` 时,该阻塞将会一直持续到解锁时。

若指定了 `LOCK_NB` 的话,若在阻塞时调用 `flock` 则返回 `false`。

所谓"阻塞时"是指发生下述情况:在其他进程已设置排它锁的情况下进行锁定,在其他进程已经锁定的情况下设置排它锁的时候。

⑤ `path`:以字符串形式返回打开文件时使用的路径。

⑥ `lstat`:生成并返回一个包含文件状态信息的 `File::Stat` 对象。若未能成功取得文件信息则引发 `Errno::EXXX` 异常。

⑦ `truncate(length)`:将文件裁剪成至多 `length` 字节大小的文件。若文件并不是以写入模式打开的话,将引发 `IOError` 异常。若修改成功则返回 `0`,若失败则引发 `Errno::EXXX` 异常。

File 类所含常数如下:

① `ALT_SEPARATOR`:当系统的文件路径分隔符与 `SEPARATOR` 不同时,即可使用该常数。在 MS-DOS 等系统中是 "\",而在 UNIX 或 Cygwin 等系统中则是 `nil`。

② `PATH_SEPARATOR`:`PATH` 环境变量的元素分隔符。在 UNIX 中是 ":",而在 MS-DOS 等系统中是 ";"。

③ `SEPARATOR` 和 `Separator`:文件路径的分隔符。例如,在将路径名传给处理文件的方法时就会用到它。该分隔符使得脚本内的路径名的格式统一且不受环境的影响。其值为 "/"。

此外,File 类含有内部类:Constants 和 Stat,这在前面已经介绍过了。

下面来做一些练习,使用 File 类读写文件。

案例名称:使用 File 类读写文件

程序名称:d.rb

```
p File.basename("ruby/a.c")
p File.basename("ruby/a.c", ".c")
p File.basename("ruby/a.c", ".* ")
p File.basename("ruby/a.exe", ".* ")
n= "test.txt"
```

```

p File.atime(n)
p File.ctime(n)
p File.mtime(n)
p File.dirname("dir/file.ext")
p File.dirname("file.ext")
p File.expand_path("../")
p File.expand_path("../", "/tmp")
p File.extname("foo/foo.txt")
p File.extname("foo/foo.tar.gz")
p File.extname("foo/bar")
p File.extname("foo/.bar")
p File.extname("foo.txt/bar")
p File.extname(".foo")
% w(foo foobar bar).each {|f|
  p File.fnmatch("foo* ", f)
}
p File.fnmatch('\a', 'a')
p File.fnmatch('\a', '\a', File::FNM_NOESCAPE)
p File.fnmatch('* ', 'a')
p File.fnmatch('* ', '\a', File::FNM_NOESCAPE)
p File.fnmatch('\ ', '\ ')
p File.fnmatch('\ ', '\ ', File::FNM_NOESCAPE)
p File.fnmatch('* ', '/', File::FNM_PATHNAME)
p File.fnmatch('? ', '/', File::FNM_PATHNAME)
p File.fnmatch('[/]', '/', File::FNM_PATHNAME)
p File.fnmatch('A', 'a', File::FNM_CASEFOLD)
p File.fnmatch('* ', '.', File::FNM_DOTMATCH)
p File.fnmatch('? ', '.', File::FNM_DOTMATCH)
p File.fnmatch('[.]', '.', File::FNM_DOTMATCH)
p File.fnmatch('foo/* ', 'foo/.', File::FNM_DOTMATCH)

```

程序运行结果如图 5-4 所示。

程序名称:e.rb

```

n= "test.txt"
f= open(n)
a= f.readlines
s= a.join("")
p s
f.close
$ KCODE= 'u'
f= open(n)
a= f.readlines
s= a.join("")
p s
f.close

```

```
C:\WINDOWS\system32\cmd.exe

D:\ror>d
"a.c"
"a"
"a"
"a"
Mon Jun 25 00:00:00 +0800 2007
Mon Jun 25 13:40:37 +0800 2007
Mon Jun 25 14:04:48 +0800 2007
"dir"
"."
"D:/"
"D:/"
".txt"
".gz"
""
""
""
""
true
true
false
true
true
false
true
false
true
false
false
false
true
true
true
true
true
true

D:\ror>
搜狗拼音 半:
```

图 5-4 使用 File 类读写文件

程序运行结果如图 5-5 所示。

```
C:\WINDOWS\system32\cmd.exe

D:\ror>e
"\316\322\260\256\304\343\327\346\271\372\243\241\n\316\322\316\260\264\363\265\
304\327\346\271\372\243\241\n\307\327\260\256\265\304\327\346\271\372\243\241\n\
304\343\323\265\323\320\323\306\276\303\265\304\316\304\273\257\243\241\n\271\30
5\300\317\265\304\300\372\312\267\243\241\n\ni love you motherland!\ngreat nothe
rland!\nChina!"
"我\260\256你祖\271 ?
我\260\264蟾\243\241\n亲\260\256\265\243\241\n你拥有悠\276\257\243\241\n\271爬系睦 罚 n\ni love you motherland!\ngreat motherland!\nChina!"

D:\ror>

搜狗拼音 半:
```

图 5-5 使用 File 类读写文件

5.2 目录操作

5.2.1 目录操作的概念

Ruby 中与目录操作相关的类为:Dir 类。它是专门为实现目录流操作而设置的类,它可以依次返回目录中的元素。

学习目录操作的主要内容就是:

- ① 打开和关闭目录;
- ② 遍历目录下的文件;
- ③ 修改文件名称;
- ④ 读取目录相关信息;
- ⑤ 新建、移动、删除目录。

5.2.2 目录操作的方法

Dir 类含有如下类方法。

(1) 创建目录对象

使用 Dir.new(path),Dir.open(path),Dir.open(path){|dir|...} 打开并返回一个针对 path 的目录流。若打开失败则引发 Errno::EXXX 异常。

open()方法可以带块。若带块调用,将执行该块。当块的运行结束时,目录将被自动关闭。带块调用该方法的返回值为块的运行结果。

(2) 改变当前目录

Dir.chdir([path]) 和 Dir.chdir([path]){|path|...} 将当前目录改为 path。若省略 path,则会调用环境变量 HOME 或 LOGDIR。若它们中包含设定值,就将当前目录改为它们的值。

若当前目录变更成功就返回 0,若失败则引发 Errno::EXXX 异常。

若指定了块,则只会在块的执行过程中变更当前目录。这等价于

```
savedir = Dir.pwd
Dir.chdir(newdir)
begin
  :
ensure
  Dir.chdir(savedir)
end
```

此时,带块方法的返回值就是块的执行结果。

(3) 删除目录

Dir.delete(path)

Dir.rmdir(path)

`Dir.unlink(path)`

删除目录,要求目录必须为空。若删除成功则返回 0,若失败则引发 `Errno::EXXX` 异常。

(4) 改变根目录

`Dir.chroot(path)`

将根目录改为 `path`。只有超级用户才能改变根目录。改变之后将无法还原到原来的根目录。

根目录变更成功时返回 0,失败时将引发 `Errno::EXXX` 异常。

(5) 返回当前目录的完整路径

使用 `Dir.getwd` 和 `Dir.pwd` 方法。若无法取得当前目录则会引发 `Errno::EXXX` 异常(一般不会失败)。

(6) 生成新目录

`Dir.mkdir(path[, mode])` 生成新目录 `path`。权限值取决于 `mode`(默认值 0777)和 `umask` 的按位与计算结果(`mode & ~umask`)。

若成功生成新目录就返回 0,若失败则引发 `Errno::EXXX` 异常。

(7) 迭代器

`Dir.foreach(path) {|file| ...}`

针对 `path` 目录中的元素的迭代器。该方法的运作方式如下:

```
dir = Dir.open(path)
begin
  dir.each {|file|
    :
  }
ensure
  dir.close
end
```

(8) `Dir.entries(path)`

以数组形式返回 `path` 目录中所含的文件名。等同于

```
def dir_s_entries( path )
  Dir.open(path) {|d|
    return d.to_a
  }
end
```

(9) 目录名匹配

```
Dir[pattern]
Dir.glob(pattern)
Dir.glob(pattern) {|file| ...}
```

```
Dir.glob(pattern[, flags])
Dir.glob(pattern[, flags]) {|file| ...}
```

以字符串数组的形式返回通配符匹配结果。若指定了块,将以匹配成功的文件为参数,依次计算该块,然后返回 nil。

可以一次指定多个 pattern,此时需要使用空白(space,tab,换行)或"\0"将它们分割开来。可用的通配符如下:

* 匹配任何字符串,包括空字符串。

? 匹配任何一个字符。

[] 只要匹配方括号中任何一个字符即可。若使用-则表示字符范围。若方括号中的首字符是^时,表示匹配^后的字符以外的任意字符。(在 ksh 或 POSIX shell 中可以使用!来代替^。)

{ } 表示依次与括号内的字符进行组合。例如,foo{a,b,c}将变为 fooa, foob, fooc,然后再依次进行匹配。括号可以嵌套。例如,{foo,bar{foo,bar}}表示将依次与 foo, barfoo, bar-bar 进行匹配。

** / 表示通配符 */重复出现 0 次以上,这将对目录进行递归式的搜索。例如,foo/* */bar 表示将依次对 foo/bar, foo/* /bar, foo/* / * /bar ... (此后无限延伸)进行匹配。

可使用反斜线对通配符进行转义处理。在双引号内的字符串中必须进行 2 层转义。另外,空白类字符将失去特殊意义(但"\0"依然有效)。可以为第 2 个可选参数指定一个类似于 File.fnmatch 所使用的标识。指定标识后就可改变匹配的运作情况。

Dir 类含有如下实例方法:

① close:关闭目录流。此后若再对目录进行操作则会引发 IOError 异常。若成功关闭则返回 nil。若失败则引发 IOError 异常。

② each {|item| ... }:为目录中的各个元素计算块的内容。

③ path:以字符串形式返回当前打开目录的路径名。

④ pos 和 tell:以整数形式返回目录流的当前位置。

⑤ pos=(pos) 和 seek(pos):将目录流的读入位置移动到 pos。pos 必须是使用 Dir#tell 所得到值。

⑥ read:从目录流中读出并返回下一个元素。若读到最后一个元素则返回 nil。若读取目录失败则引发 Errno::EXXX 异常。

⑦ rewind:将目录流的读取位置移到开头。

现在,做一些练习,将文件和目录联系到一起。这些练习包括:

① 打开文件,读取其内容,然后修改其内容,保存文件。

② 遍历某个目录下的所有文件和目录,将其显示出来。

③ 创建文件、修改文件名、删除文件、移动文件。

④ 创建目录、修改目录名、删除目录、移动目录。

案例名称:读写文件

程序名称:test.txt

```
i love you China!  
i love you mom!
```

程序名称:f.rb

```
n= "test.txt"  
# open  
f= File.open(n,File::RDWR)  
# read  
a= f.readlines  
s= a.join("")  
p s  
f.close  
# append  
f= File.open(n,File::RDWR)  
f.seek(0, IO::SEEK_END)  
str= "append string"  
f.write(str)  
f.close  
f= File.open(n,File::RDWR)  
a= f.readlines  
s= a.join("")  
p s  
f.close  
# modify  
f= File.open(n,File::RDWR)  
f.truncate(0)  
str= "new string"  
f.write(str)  
f.close  
f= File.open(n,File::RDWR)  
a= f.readlines  
s= a.join("")  
p s  
f.close
```

程序运行结果如图 5-6 所示。

案例名称:创建文件

程序名称:g.rb

```

n= "ta.txt"
# create
f= File.open(n,File::CREAT)
f.close
f= File.open(n,"w+ ")
str= "new string"
f.write(str)
f.close
f= File.open(n,File::RDWR)
a= f.readlines
s= a.join("")
p s
f.close

```

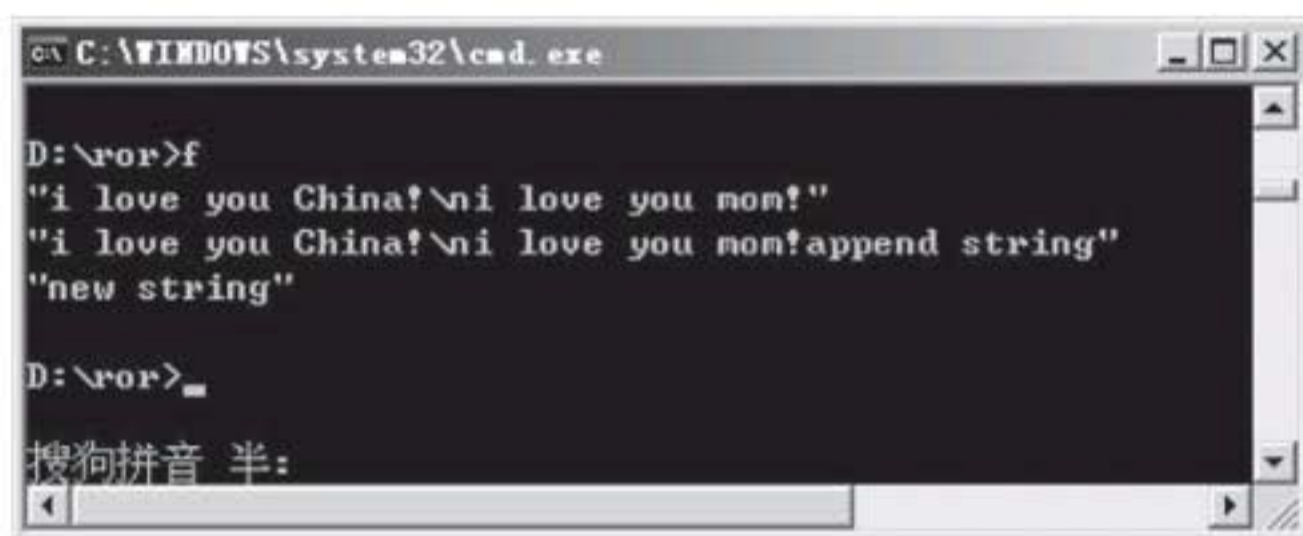


图 5-6 读写文件

程序运行结果如图 5-7 所示。



图 5-7 创建文件

案例名称:重命名和删除文件

程序名称:h. rb

```

n= "ta.txt"
# rename
p "rename tb.txt- - > tc.txt :"
i= File.rename("tb.txt","tc.txt")
p i
p "delete td.txt: "
i= File.delete("td.txt")
p i

```

程序运行结果如图 5-8 所示。



图 5-8 重命名和删除文件

案例名称:遍历某目录下的所有文件和目录

程序名称:i.rb

```
path= "d:/ror/a"
dir = Dir.open(path)
begin
  dir.each {|file|
    if File.file? (file) then
      p "file:" + file.to_s
    end
    if File.directory? (file) then
      p "directory:" + file.to_s
    end
  }
ensure
  dir.close
end
p "- - - - -"
Dir.foreach(path) {|file|
  if File.file? (file) then
    p "file:" + path + "/" + file.to_s
  else
    p "directory:" + file.to_s
  end
}
p "- - - - -"
p "考虑多层目录"
def all(path)
  Dir.foreach(path) {|file|
    if File.file? (file) then
      p "file:" + path + "/" + file.to_s
    else
```

```

        if file.to_s! = "." && file.to_s! = ".." then
            all(path.to_s + "/" + file.to_s)
        end
    end
end
all(path)
a= Dir.entries(path)
p a

```

程序运行结果如图 5-9 所示。

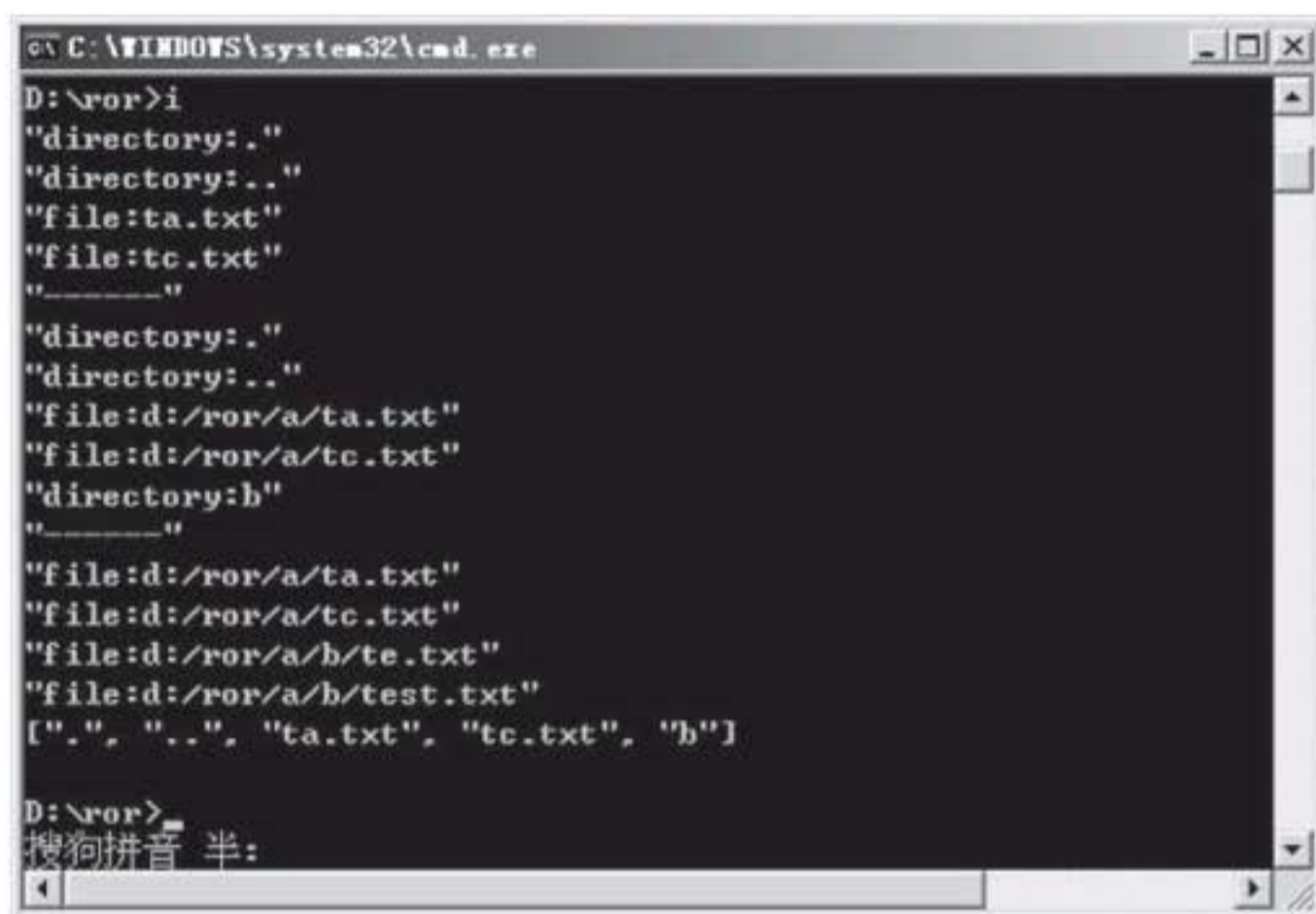


图 5-9 遍历某目录下的所有文件和目录

案例名称:创建和删除目录

程序名称:j.rb

```

path= "d:/ror/a"
Dir.chdir(path)
Dir.mkdir("root")
Dir.chdir(path + "/root")
Dir.mkdir("sub1")
Dir.mkdir("sub2")
Dir.mkdir("sub3")
Dir.delete("sub3")

```

程序运行结果如图 5 - 10 所示。

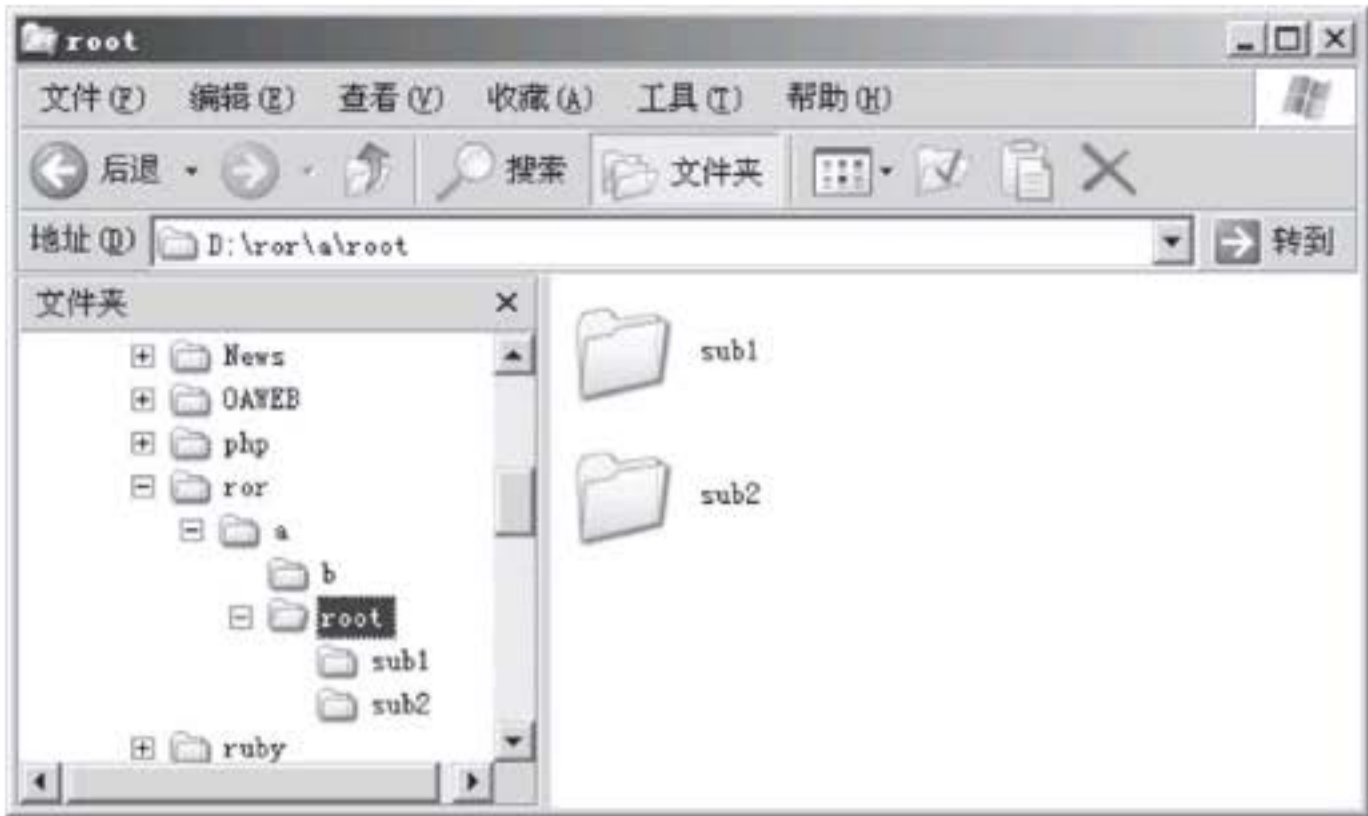


图 5 - 10 创建和删除目录

小 结

文件和目录操作在任何编程语言中都是非常重要的。本章介绍 Ruby 中实现文件和目录操作的相关类和方法。

思考和练习

熟练掌握文件的读写操作。

第 6 章 Ruby 的数据库操作

本章要点

Ruby 最重要的功能就是数据库访问,这也是任何一种编程语言的重要功能。本章详细介绍使用 Ruby 进行数据库访问的方法,主要以 MySQL 数据库为例。

6.1 Ruby 数据库访问的概念

6.1.1 数据库访问的方式

使用 Ruby 进行数据库访问,大体有 3 种方式:

- ① 使用与数据库相关的驱动程序来访问数据库。
- ② 使用抽象的数据库无关的访问层(DBI)来访问数据库。
- ③ 使用 Ruby 的 ORM 组件 ActiveRecord 来访问数据库(Rails 中使用)。

6.1.2 数据库访问的目的

进行数据库访问,主要是对连接的数据库执行数据提取、修改、删除等操作。在保持数据连接的情况下执行不同的 SQL 语句即可。

常用的数据库有:Oracle,DB2,Sybase,SQL Server,MySQL,Access,Postgre 等。对国人而言,SQL Server,Access 和 MySQL 是最常用的,所以,笔者在此只介绍这 3 种数据库的访问方式。其他数据库的访问,读者可效仿。

6.2 访问 Access 数据库

6.2.1 配置环境

访问 Access 数据库,最好用 DBI 方式。DBI 访问方式是官方提供的一种标准数据库访问方法。

Ruby 的 DBI 模块架构分为两层:

数据库接口层(database interface, DBI)与数据库类型无关的,它提供一些与具体使用数据库无关的通用的访问方法。

数据库驱动层(database driver, DBD)是与数据库相关的,不同的驱动用来访问不同的数据库。一个驱动用来访问 mysql,另一个用来访问 postgre,每一个具体的数据库都有不同的驱动。每个驱动解释 DBI 层传送的请求,并转换成对应于具体数据库的请求,发送到数据库。

(1) 下载 Ruby/DBI 软件包

进入 <http://rubyforge.org/projects/ruby-dbi/> 网站, 可以下载 Ruby/DBI 软件包。如图 6-1 所示。



图 6-1 下载 Ruby/DBI 软件包

点击“下载”链接, 进入具体软件包下载页面, 如图 6-2 所示。

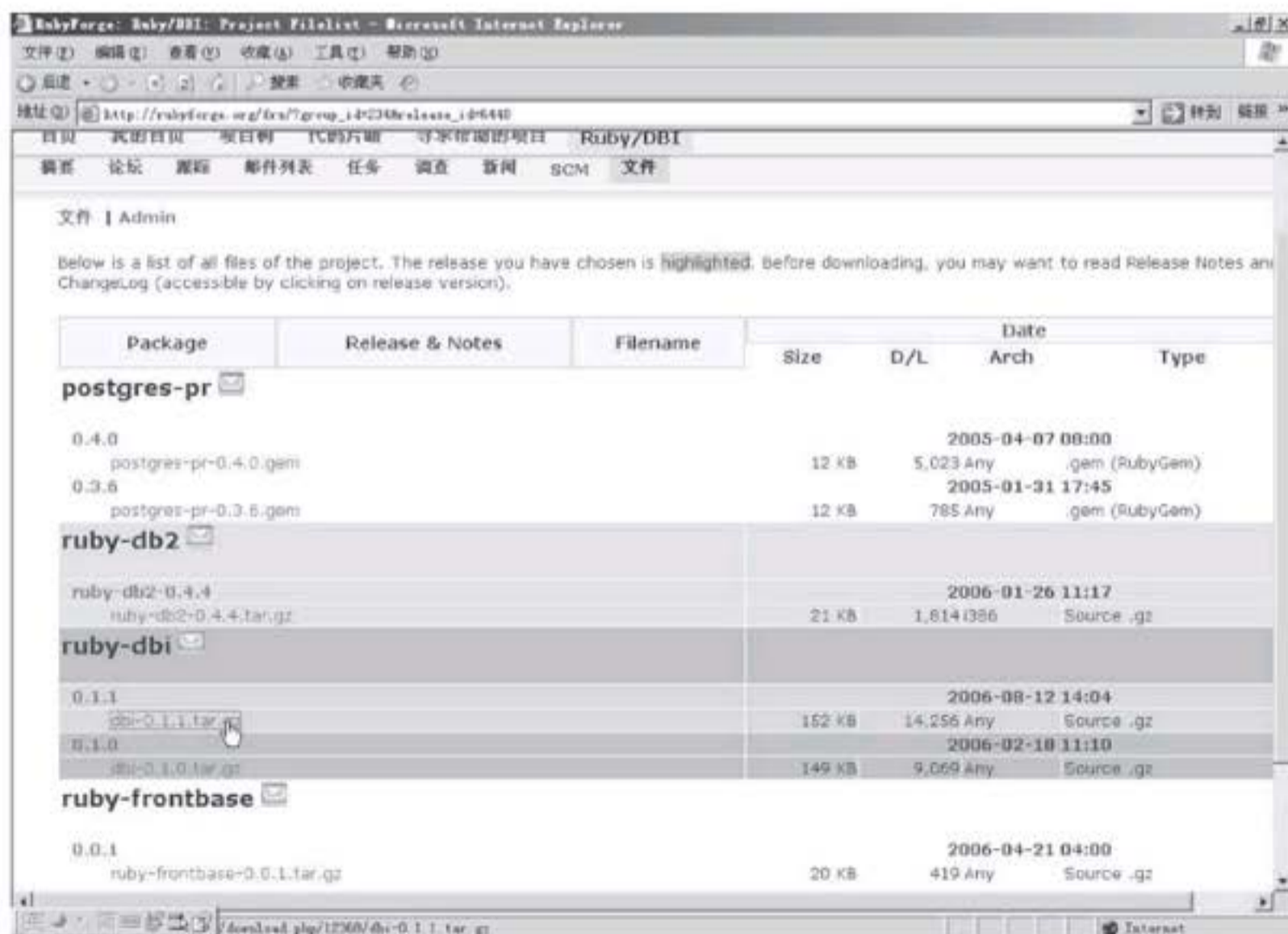


图 6-2 下载 Ruby/DBI 软件包

当前最新版本的 ruby-dbi 是 0.1.1, 将其下载, 结果如图 6-3 所示。



图 6-3 下载 Ruby/DBI 软件包

(2) 安装驱动程序

要使用 DBI 来访问 Access 数据库, 需要安装 ADO 驱动程序。解压缩 ruby-dbi 软件包, 从 lib/dbd 文件夹下取出 ADO.rb 文件, 复制到 D:\ruby\lib\ruby\site_ruby\1.8\DBD\ADO 目录下。如图 6-4 所示。这样, 就完成了安装。

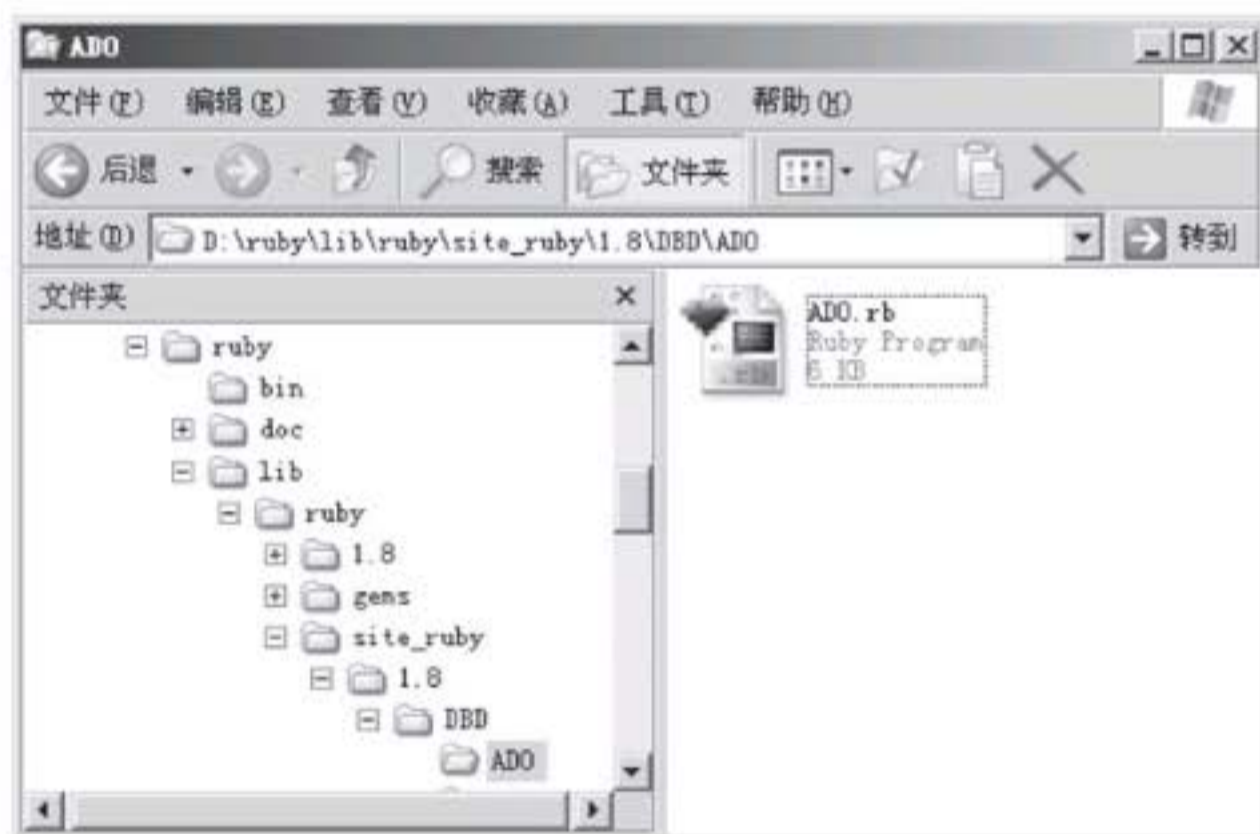


图 6-4 安装驱动程序

(3) 测试安装

建立一个 Access 数据库, 名为“db.mdb”, 如图 6-5 所示。



图 6-5 Access 数据库

在数据库中建立一个表格,名为 test,如图 6-6 所示。

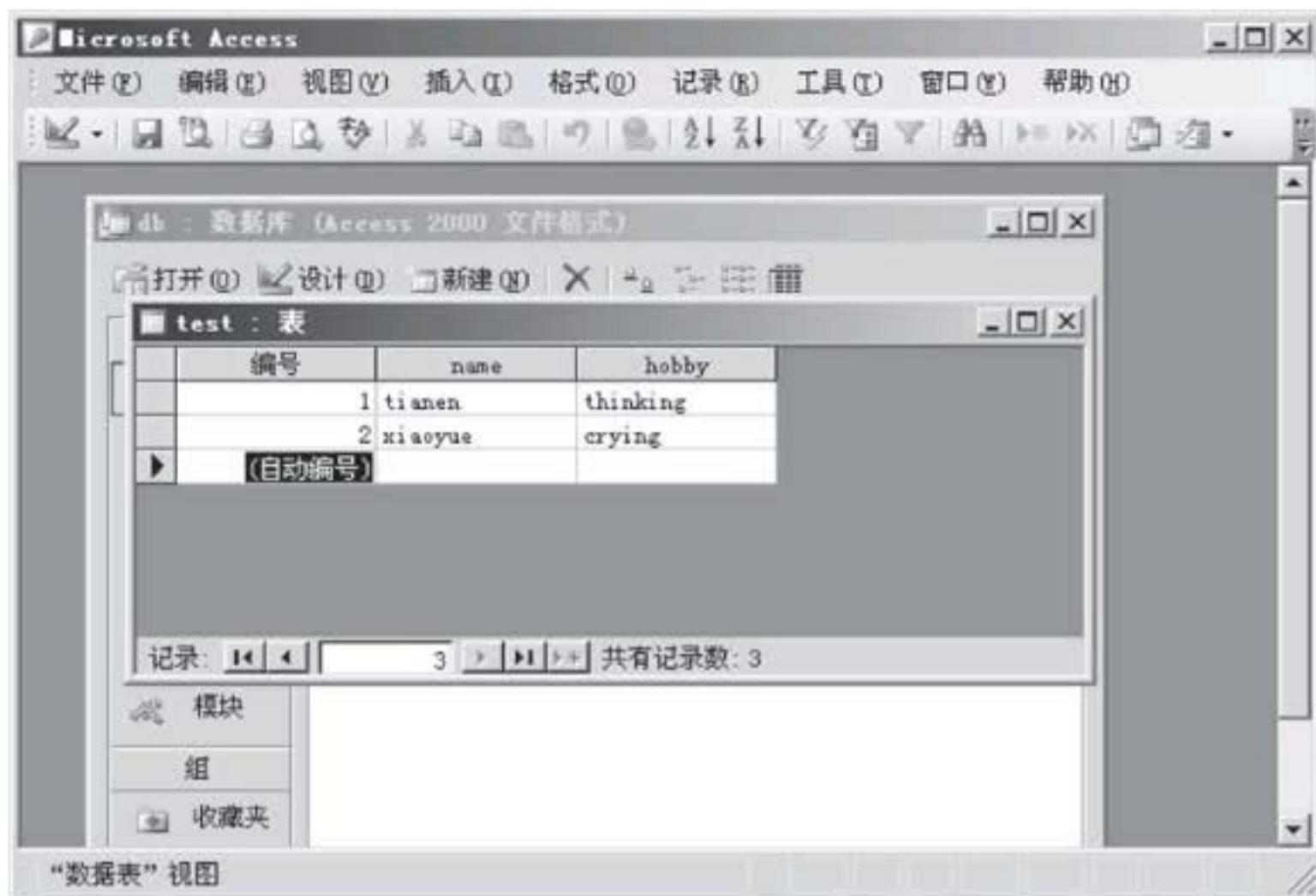


图 6-6 Access 数据库

建立一个 Ruby 程序,如下:

案例名称:访问 Access

程序名称:access.rb

```
require 'dbi'
dbh= DBI.connect (" DBI:ADO:Provider = Microsoft.Jet.OLEDB.4.0; Data Source =
db.mdb;")
sth= dbh.prepare('select * from test')
sth.execute
while row= sth.fetch do
  p row
end
sth.finish
dbh.disconnect
```

程序运行结果如图 6-7 所示。



图 6-7 访问 Access

如此,则证明环境配置成功,下面就可以通过 Access 来详细介绍 DBI 的使用方法了。

6.2.2 执行数据操纵语句

基本方式

使用 DBI 执行数据操纵语句的基本方式如下:

(1) 引入 DBI 模块

要使用 DBI 进行数据库访问,首先要在程序中引入 DBI 模块。如下:

```
require 'dbi'
```

(2) 连接数据库

使用 DBI 连接 Access 数据库,需要使用 DBI 模块的 connect 方法,方法的参数为数据库连接字符串,返回一个数据库连接对象。如下:

```
dbh=DBI.connect("DBI:ADO:Provider=Microsoft.Jet.OLEDB.4.0;Data Source=db.mdb;")
```

(3) 传递数据操纵语句

传递的 SQL 语句不需要返回记录(如:update,delete),就需要用“do”语句。

如:dbh.do("delete from test")

这条语句可以有返回值,返回受到影响的记录数目。

Ruby DBI 提供了占位符机制,使得不用在查询语句中把数据值的字面值写里面,而是用一些特殊的符号标记数据的位置,当要执行的时候,用真实的数据值填充占位符的位置。DBI 会用数据值替换占位符,完成对字符串等加引号、特殊字符的转义(如果需要的话)等,而且占位符机制能很好地处理 NULL 值,只需要提供一个 nil 值,它会自动被换成 NULL 放到查询中。

比如,执行 dbh.do("insert into test values ('lily','coffee')") 等同于执行:dbh.do("insert into test values (?,?)", 'lily', 'coffee')

而且,用得更广泛的方式是

```
sth = dbh.prepare("insert into test values (?,?)")
sth.execute('lily','coffee')
```

(4) 关闭数据库连接

使用 dbh.disconnect。

代码格式

对于代码格式,要注意,所有的代码都应包括在一个 begin/rescue/ensure 结构中:

begin 部分处理所有的数据库请求。

rescue 部分用来处理出错信息,它将获取出错信息,并显示出来。

ensure 部分确保程序不管出错与否,最后都将关闭数据库连接。

事务处理

DBI 提供了事务支持,但是怎样支持取决于底层数据库和 DBD 层数据库驱动的实现情况。实现事务处理的基本方式如下:

```
dbh['AutoCommit'] = false # 取消自动提交事务
begin
  # 执行代码
dbh.commit # 提交事务
rescue
dbh.rollback # 回滚事务
end
```

此外,还可以用 transaction 块,根据这个块执行结果是成功还是失败自动执行 commit 或者 rollback。如下:

```
dbh['AutoCommit'] = false
dbh.transaction do |dbh|
  # 执行代码
end
```

这两种做法在 .NET 中都有类似的实现。

基本案例

有了这些知识,就可以在数据库中实现数据操纵。下面举例子来说明。

案例名称:执行数据操纵语句

程序名称:aa.rb

```
require 'dbi'
dbh= DBI.connect ( "DBI:ADO:Provider = Microsoft.Jet.OLEDB.4.0;Data Source =
db.mdb;" )
begin
  r= dbh.do("update test set hobby= 'laughing' where name= 'xiaoyue'")
  printf "% d rows updated\n", r
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end
```

对这段程序执行“update”语句,运行结果如图 6-8 所示。



图 6-8 执行“update”语句

查看数据表,如图 6-9 所示。

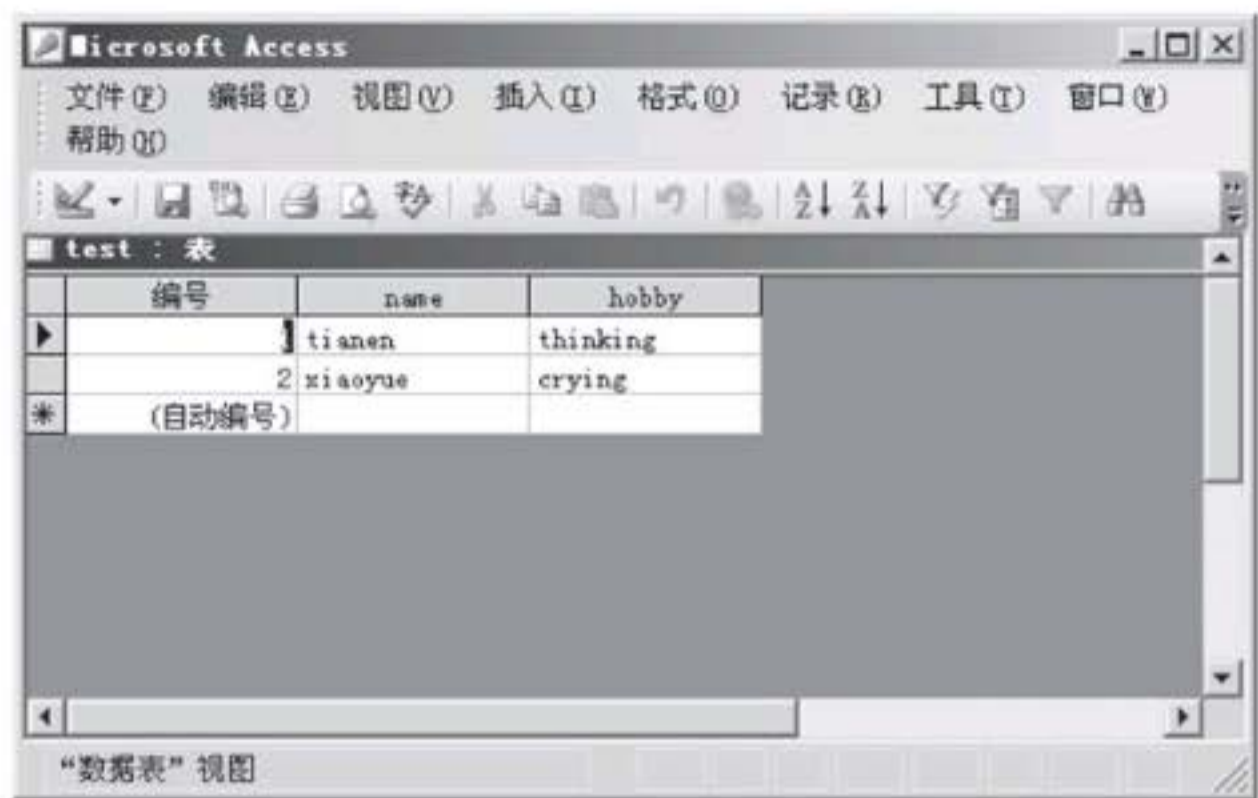


图 6-9 执行“update”语句

这不是想要的结果,数据并没有被更新,原因是 DBI 访问 Access 时,默认情况下禁止了事务的自动提交。所以,在执行“update”,“insert”,“delete”等语句时,要将操作提交给数据库,就得使事务自动提交或者显式地提交。

程序名称:ab.rb

```
require 'dbi'
dbh= DBI.connect (" DBI:ADO: Provider = Microsoft.Jet.OLEDB.4.0; Data Source =
db.mdb;")
dbh['AutoCommit'] = true
begin
  r= dbh.do("update test set hobby= 'laughing' where name= 'xiaoyue'")
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end
```

这个程序设置了事务的自动提交。运行程序,查看数据表,如图 6-10 所示。

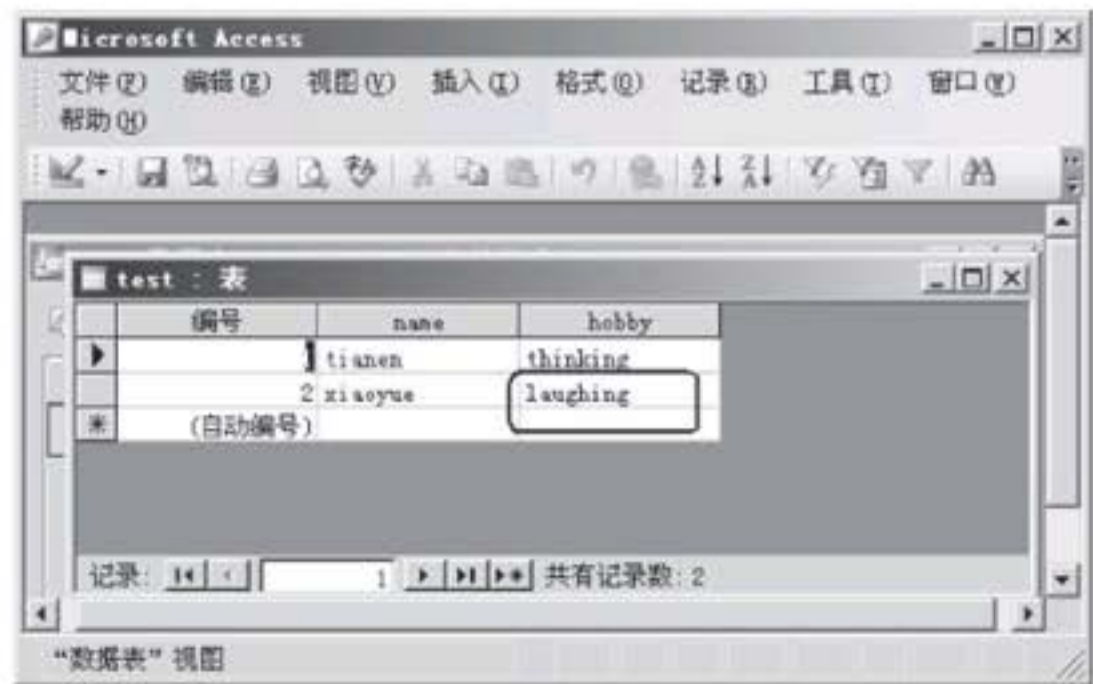


图 6-10 执行“update”语句

现在,保持事务的自动提交被禁止,然后手动提交事务。

程序名称:ac.rb

```
require 'dbi'
dbh= DBI.connect ( "DBI:ADO:Provider= Microsoft.Jet.OLEDB.4.0;Data Source =
db.mdb;" )
begin
  r= dbh.do("insert into test(name,hobby) values ('lucy','dancing')")
  dbh.commit
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end
```

运行程序,查看数据表,如图 6-11 所示。



图 6-11 执行“update”语句

批量执行 SQL 语句

前面的内容很基础,下面再做几个练习,来巩固学习成果。

① 使用散列表和 DBI 的占位符功能向数据表中添加多条记录

程序名称:ad.rb

```
require 'dbi'
h= Hash.new
h["lily"]= "coffee"
h["tom"]= "tea"
h["amy"]= "grape"
h["jim"]= "banana"
dbh= DBI.connect("DBI:ADO:Provider= Microsoft.Jet.OLEDB.4.0;Data Source= db.mdb;")
begin
  sth = dbh.prepare("insert into test(name,hobby) values (?,?)")
  h.each_pair {|k, v| sth.execute(k,v)}
```

```

dbh.commit
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

运行程序,查看数据表,如图 6-12 所示。



图 6-12 批量执行 SQL 语句

② 使用数组和 DBI 的占位符功能向数据表中添加多条记录

程序名称:ae.rb

```

require 'dbi'
a= Array.new
a.push ["john","playing"]
a.push ["green","tree"]
a.push ["jack","pen"]
dbh= DBI.connect (" DBI: ADO: Provider = Microsoft.Jet.OLEDB.4.0; Data Source =
db.mdb;")
begin
  sth = dbh.prepare("insert into test (name,hobby) values (?,?)")
  a.each{|e| sth.execute(e[0],e[1])}
  dbh.commit
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

运行程序,查看数据表,如图 6-13 所示。

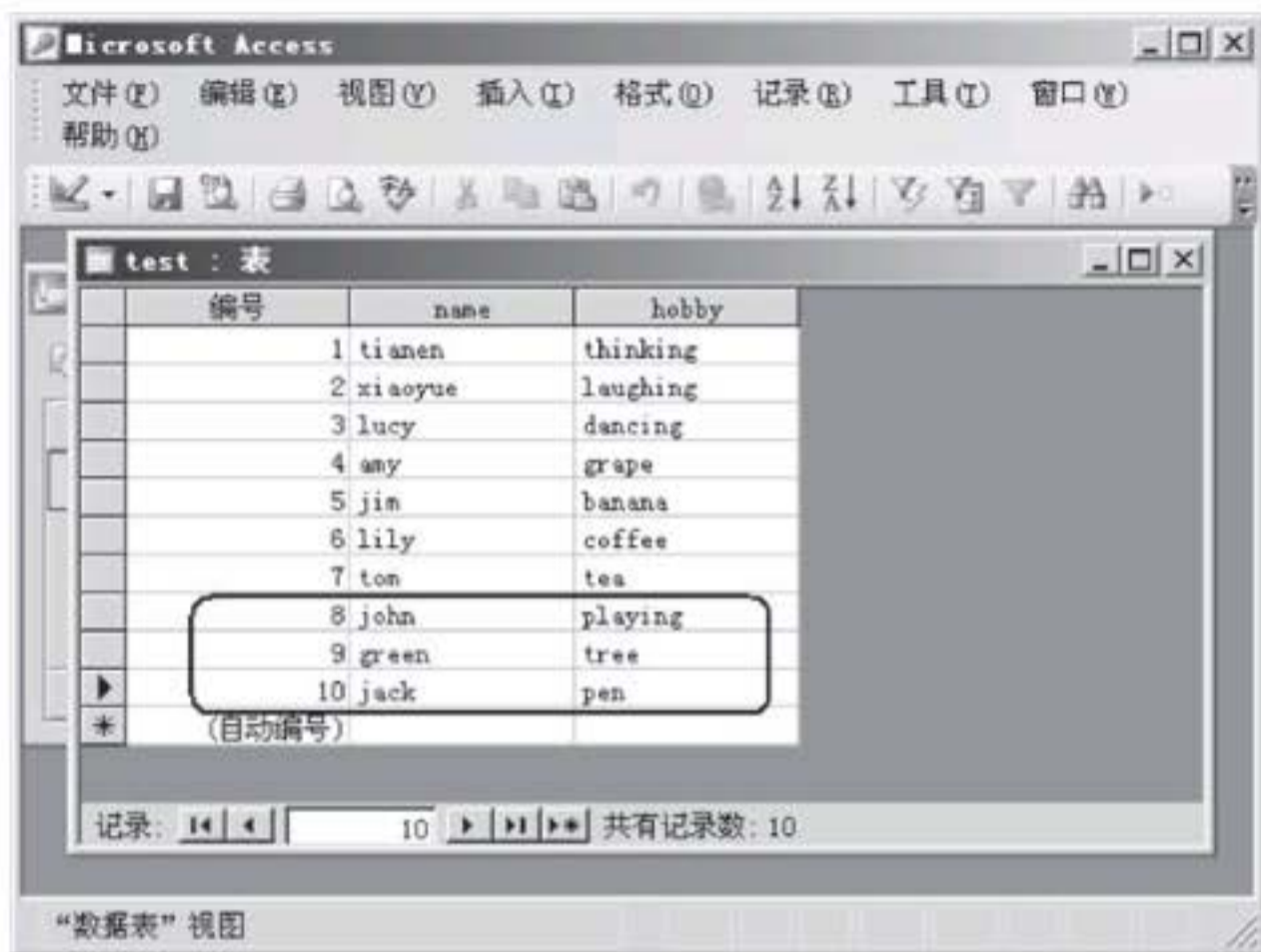


图 6-13 批量执行 SQL 语句

6.2.3 执行数据查询语句

要执行数据查询,基本步骤与执行数据操纵是一样的。但是,在处理返回结果方面有很多不同。

(1) 引入 DBI 模块

要使用 DBI 进行数据库访问,首先要在程序中引入 DBI 模块。如下:

```
require 'dbi'
```

(2) 连接数据库

使用 DBI 连接 Access 数据库,需要使用 DBI 模块的 connect 方法,方法的参数为数据库连接字符串,返回一个数据库连接对象。如下:

```
dbh=DBI. connect("DBI:ADO;Provider=Microsoft.Jet.OLEDB.4.0;Data Source=db.mdb;")
```

(3) 传递数据查询语句并处理返回结果

像 select 和 show 这样的语句是要返回行记录的,处理这样的语句,要先向服务器提交查询,处理查询产生的每条记录,然后把结果集销毁。

提交查询的方法有两种。如下:

```
dbh.execute("select * from test")
```

和

```
sth = dbh.prepare("select * from test")
```

```
sth.execute
```

处理查询结果的方法也很多。

可以在一个循环里调用 `fetch` 方法直到返回 `nil` 为止,如下:

```
sth = dbh.execute("SELECT * FROM test")
while row = sth.fetch do
  printf "ID: % d, Name: % s, Hobby: % s\n", row[0], row[1], row[2]
end
```

也可以使用迭代器,如下:

```
sth = dbh.execute("SELECT * FROM test")
sth.fetch do |row|
  printf "ID: % d, Name: % s, Hobby: % s\n", row[0], row[1], row[2]
end
```

或

```
sth = dbh.execute("SELECT * FROM test")
sth.each do |row|
  printf "ID: % d, Name: % s, Hobby: % s\n", row[0], row[1], row[2]
end
```

`fetch` 和 `each` 都产生了 `DBI::Row` 对象,这个对象提供了访问其内容的方法。

可以用 `by_index` 或 `by_field` 来通过顺序或者名字访问字段值,如

```
val = row.by_index(2)    val = row.by_field("hobby")
```

字段值也可以将 `row` 对象当成数组来取得

```
val = row[2]    val = row["hobby"]
```

迭代方法 `each_with_name` 生成每个字段名和它们的值,如

```
sth = dbh.execute("SELECT * FROM test")
sth.each do |row|
  row.each_with_name do |val, name|
    printf "% s: % s, ", name, val.to_s
  end
  print "\n"
end
```

`DBI::Row` 对象提供了一个方法 `column_names` 来得到一个包含每个字段名的数组。`field_names` 是 `column_names` 的别名。

其他的返回行数据的方法包括 `fetch_array` 和 `fetch_hash`,他们不返回 `DBI::Row` 对象,而是将下一行数据作为数组或者散列表返回,如果已经到结果集的最后,就会返回 `nil`。`fetch_hash` 返回散列表结构,由列名作为 `key`,而列的值作为这个 `key` 对应的值。这两个方法可以独立使用,也可以在迭代中使用。下面的例子用了 `fetch_hash` 方法:

```
sth = dbh.execute("SELECT * FROM test")
while row = sth.fetch_hash do
  printf "ID: % d, Name: % s, Hobby: % s\n", row["id"], row["name"], row["hobby"]
end
sth = dbh.execute("SELECT * FROM test")
```

```
sth.fetch_hash do |row|
  printf "ID: % d, Name: % s, Hobby: % s\n", row["id"], row["name"], row["hobby"]
end
```

也可以不用依照“查询—取结果—完成”这种顺序来执行语句,可以一次取回所有的结果,如

```
row = dbh.select_one(statement)
rows = dbh.select_all(statement)
```

`select_one` 执行一个查询,然后将结果的第一行作为一个数组返回,如果没有匹配记录就返回 `nil`。

`select_all` 返回一个 `DBI::Row` 的数组,如果没有匹配结果,则返回空数组而不是 `nil`。

除了查询数据库记录之外,还可以查询元数据,可以得到返回结果集行和列的个数,以及各列的信息。

要想得到返回的列的个数,可以从 `sth.column_names.size` 得到。方法 `column_info` 返回各列的详细信息。

下面的例子说明了如何用一个查询得到元数据:

```
query = "select * from test"
sth = dbh.execute(query)
if sth.column_names.size == 0 then
  puts "no result set"
  printf "Number of rows affected: % d\n", sth.rows
else
  puts "Query has a result set"
  rows = sth.fetch_all
  printf "Number of rows: % d\n", rows.size
  printf "Number of columns: % d\n", sth.column_names.size
  sth.column_info.each_with_index do |info, i|
    printf "- - - Column % d (% s) - - - \n", i, info.name
  end
end
end
sth.finish
```

(4) 释放结果集

释放结果集使用 `sth.finish`。

(5) 关闭数据库连接

关闭数据库连接使用 `dbh.disconnect`。

基本案例

有了这些知识,就可以在数据库中实现数据操纵。下面举例来说明。

案例名称:获得元数据

程序名称:af.rb

```

require 'dbi'
dbh= DBI.connect("DBI:ADO:Provider= Microsoft.Jet.OLEDB.4.0;Data Source= db.mdb;")
begin
  query = "select * from test"
  sth = dbh.execute(query)
  if sth.column_names.size == 0 then
    puts "no result set"
    printf "Number of rows affected: % d\n", sth.rows
  else
    puts "Query has a result set"
    rows = sth.fetch_all
    printf "Number of rows: % d\n", rows.size
    printf "Number of columns: % d\n", sth.column_names.size
    sth.column_info.each_with_index do |info, i|
      printf "- - - Column % d (% s) - - - \n", i, info.name
    end
  end
  sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

运行程序,结果如图 6-14 所示。



图 6-14 获得元数据

案例名称:执行查询

程序名称:ag.rb

```

require 'dbi'
dbh= DBI.connect("DBI:ADO:Provider= Microsoft.Jet.OLEDB.4.0;Data Source= db.mdb;")
begin
  query = "select * from test"
  sth = dbh.execute(query)

```

```

while row = sth.fetch do
  printf "ID: % d, Name: % s, Hobby: % s\n", row[0], row[1], row[2]
end
sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

运行程序,结果如图 6-15 所示。

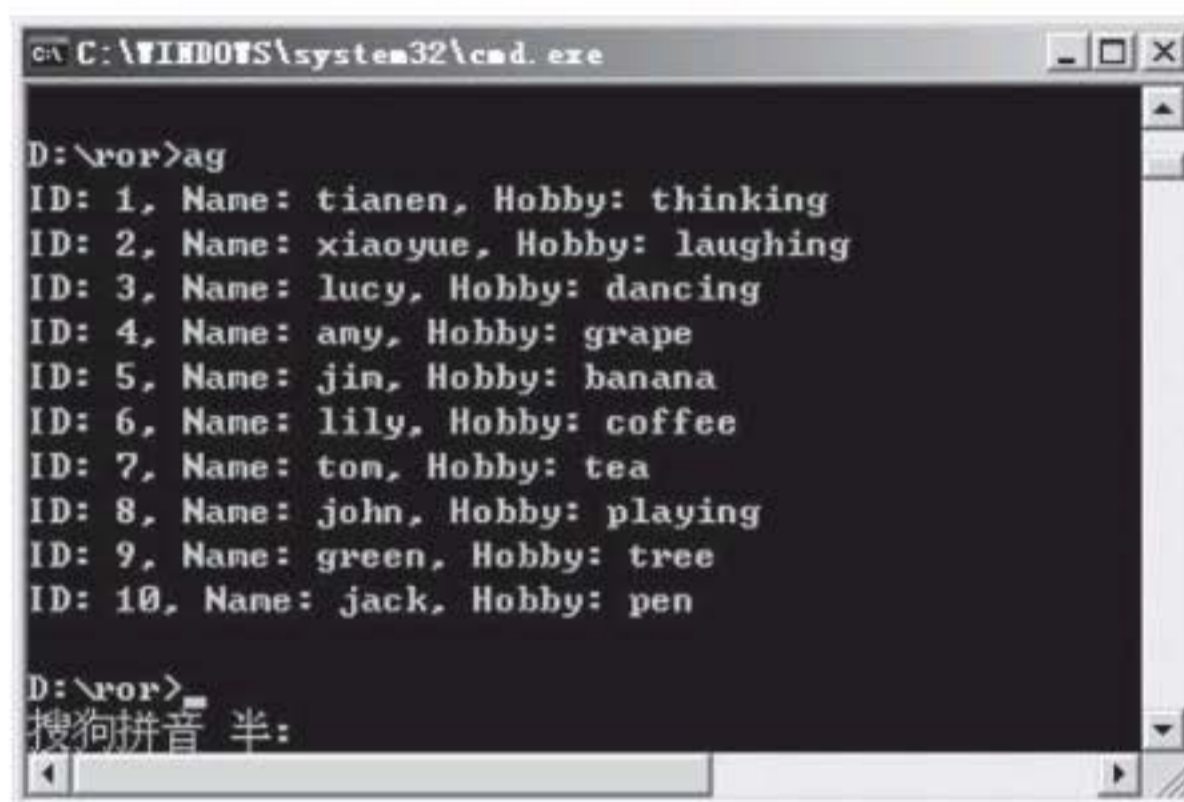


图 6-15 执行查询

案例名称:执行查询

程序名称:ah.rb

```

require 'dbi'
dbh= DBI.connect ( " DBI: ADO: Provider = Microsoft.Jet.OLEDB.4.0; Data Source =
db.mdb;" )
begin
  query = "select * from test"
  sth = dbh.execute(query)
  sth.fetch do |row|
    printf "ID: % d, Name: % s, Hobby: % s\n", row[0], row[1], row[2]
  end
  sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

运行程序,结果如图 6-16 所示。

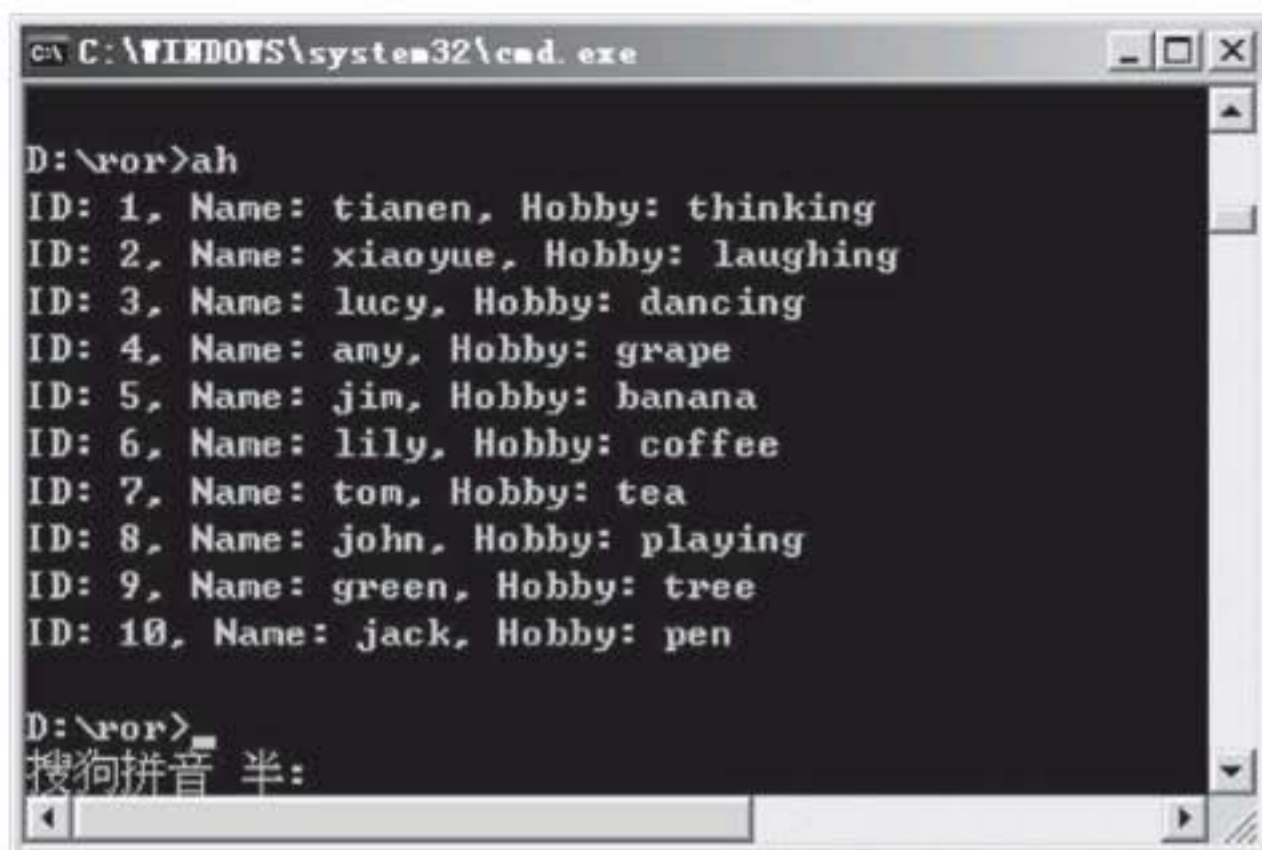


图 6-16 执行查询

案例名称:执行查询

程序名称:ai.rb

```

require 'dbi'
dbh= DBI.connect("DBI:ADO:Provider= Microsoft.Jet.OLEDB.4.0;Data Source= db.mdb;")
begin
  query = "select * from test"
  sth = dbh.execute(query)
  sth.each do |row|
    row.each_with_name do |val, name|
      printf "% s: % s, ", name, val.to_s
    end
    print "\n"
  end
  sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

运行程序,结果如图 6-17 所示。

案例名称:执行查询

程序名称:aj.rb

```

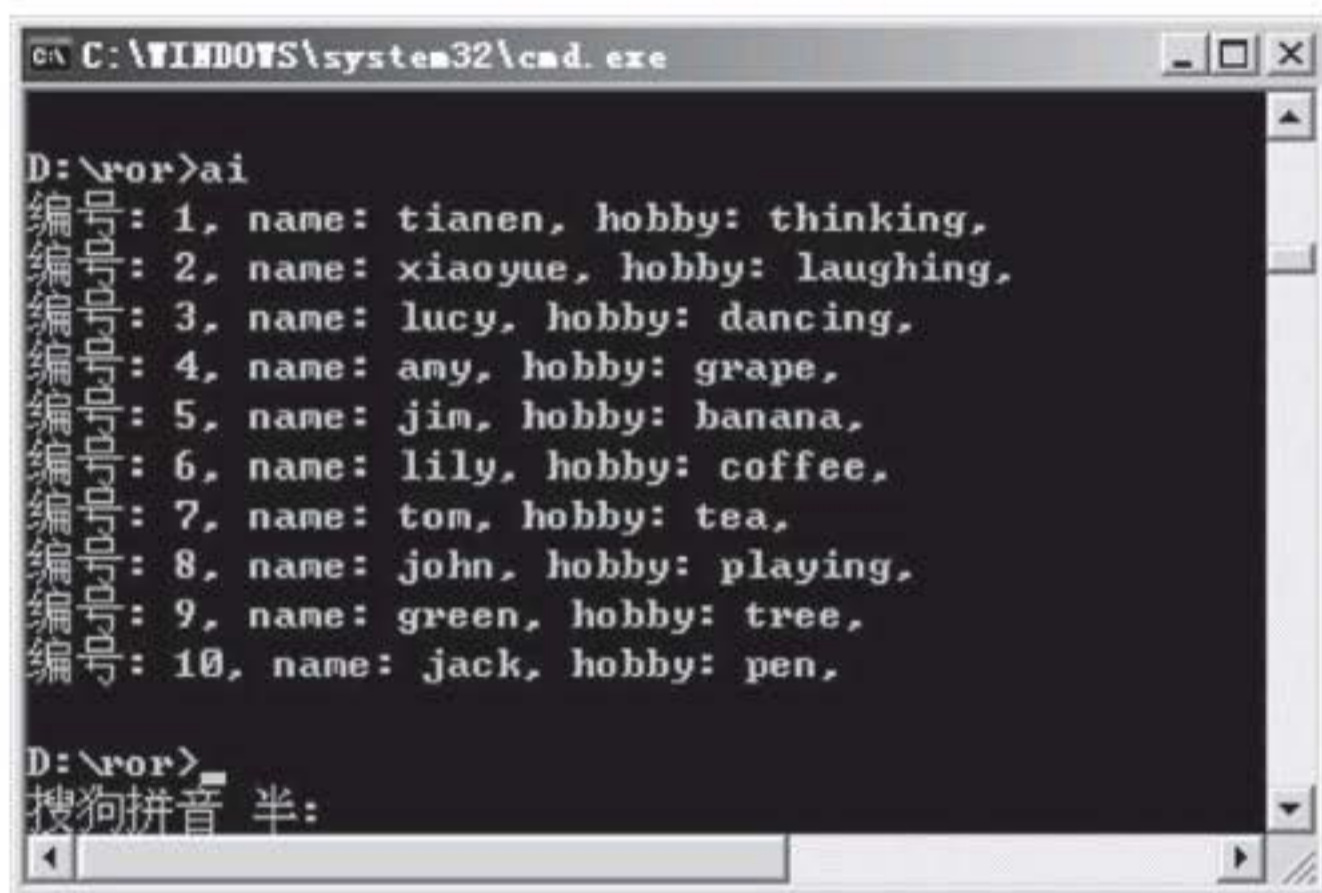
require 'dbi'
dbh= DBI.connect("DBI:ADO:Provider= Microsoft.Jet.OLEDB.4.0;Data Source= db.mdb;")
begin
  query = "select * from test"
  rows = dbh.select_all(query)

```

```

rows.each{|e|
    printf "% s, % s, % s\n", e[0], e[1],e[2]
}
rescue
    p "error!"
ensure
    dbh.disconnect if dbh
end

```



```

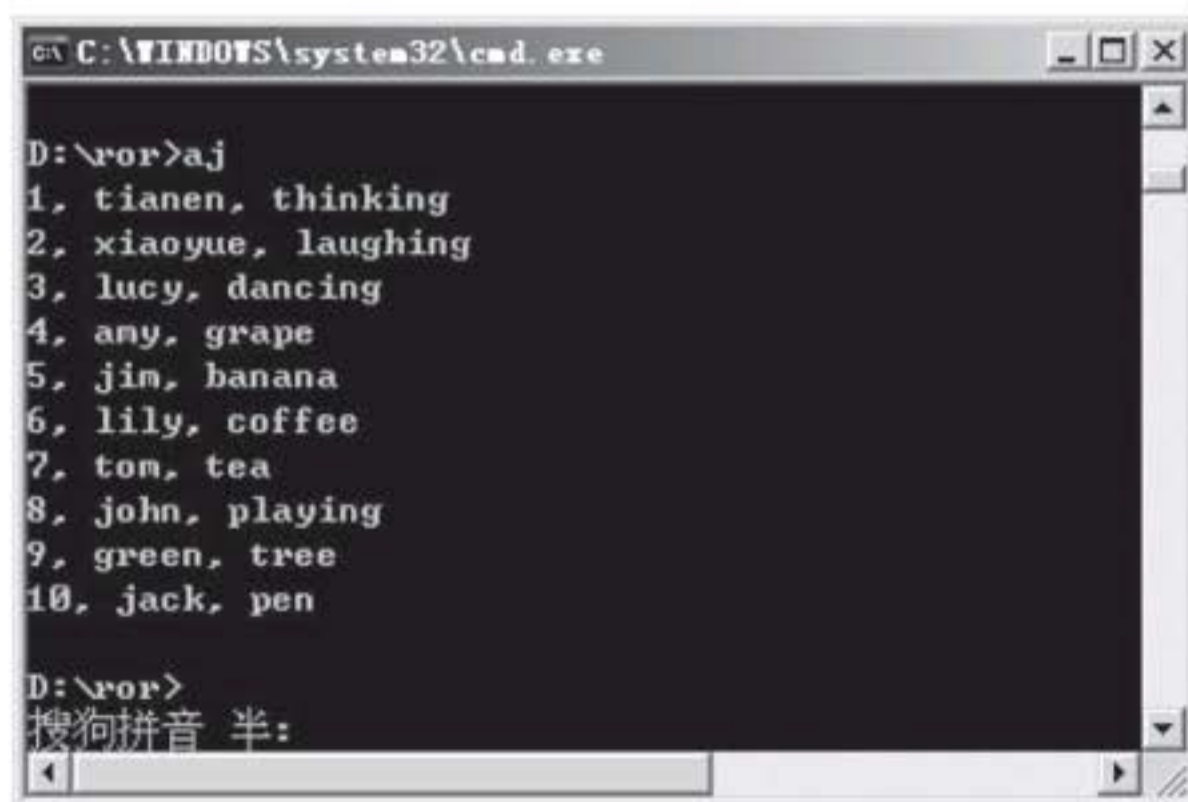
C:\WINDOWS\system32\cmd.exe
D:\ror>ai
编号: 1, name: tianen, hobby: thinking,
编号: 2, name: xiaoyue, hobby: laughing,
编号: 3, name: lucy, hobby: dancing,
编号: 4, name: any, hobby: grape,
编号: 5, name: jim, hobby: banana,
编号: 6, name: lily, hobby: coffee,
编号: 7, name: tom, hobby: tea,
编号: 8, name: john, hobby: playing,
编号: 9, name: green, hobby: tree,
编号: 10, name: jack, hobby: pen,

D:\ror>
搜狗拼音 半:

```

图 6-17 执行查询

运行程序,结果如图 6-18 所示。



```

C:\WINDOWS\system32\cmd.exe
D:\ror>aj
1, tianen, thinking
2, xiaoyue, laughing
3, lucy, dancing
4, any, grape
5, jim, banana
6, lily, coffee
7, tom, tea
8, john, playing
9, green, tree
10, jack, pen

D:\ror>
搜狗拼音 半:

```

图 6-18 执行查询

6.3 访问 SQL Server 数据库

6.3.1 建立 ODBC 数据源

可以通过 DBI 来访问 SQL Server 数据库,首先要为 SQL Server 数据库建立 ODBC 数据源。其方法如下

(1) 启动数据源管理器

从“控制面板”中可以启动数据源管理器。如图 6-19 所示。



图 6-19 启动数据源管理器

(2) 选择 SQL Server 驱动程序

单击“添加”按钮,弹出窗口,从中选择驱动程序“SQL Server”,如图 6-20 所示。



图 6-20 选择 SQL Server 驱动程序

(3) 设定数据源名称

单击“完成”按钮,在弹出的窗口中设定数据源的名称和所属数据库服务器。如图 6-21 所示。此处设定数据源名称为“my”。

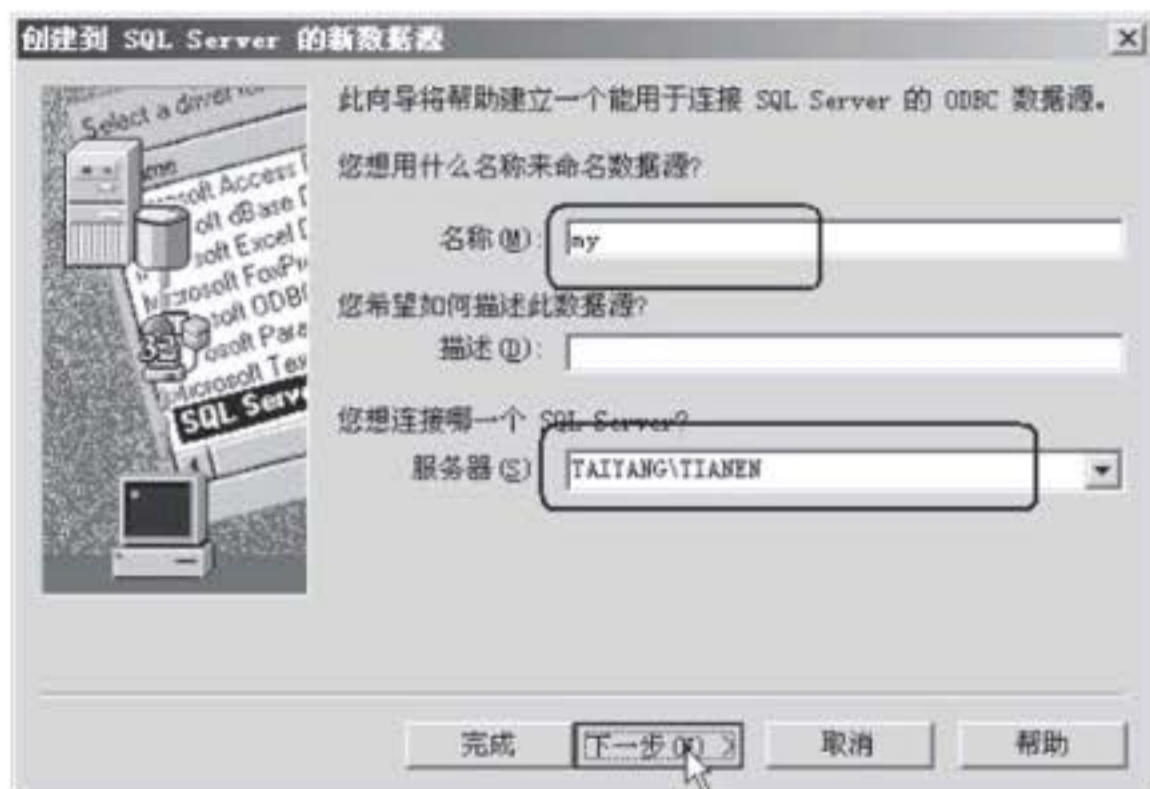


图 6-21 设定数据源名称

(4) 设置登录方式

单击“下一步”按钮,在弹出的窗口中设置登录数据库的方式,这一步保持默认即可。如图 6-22 所示。

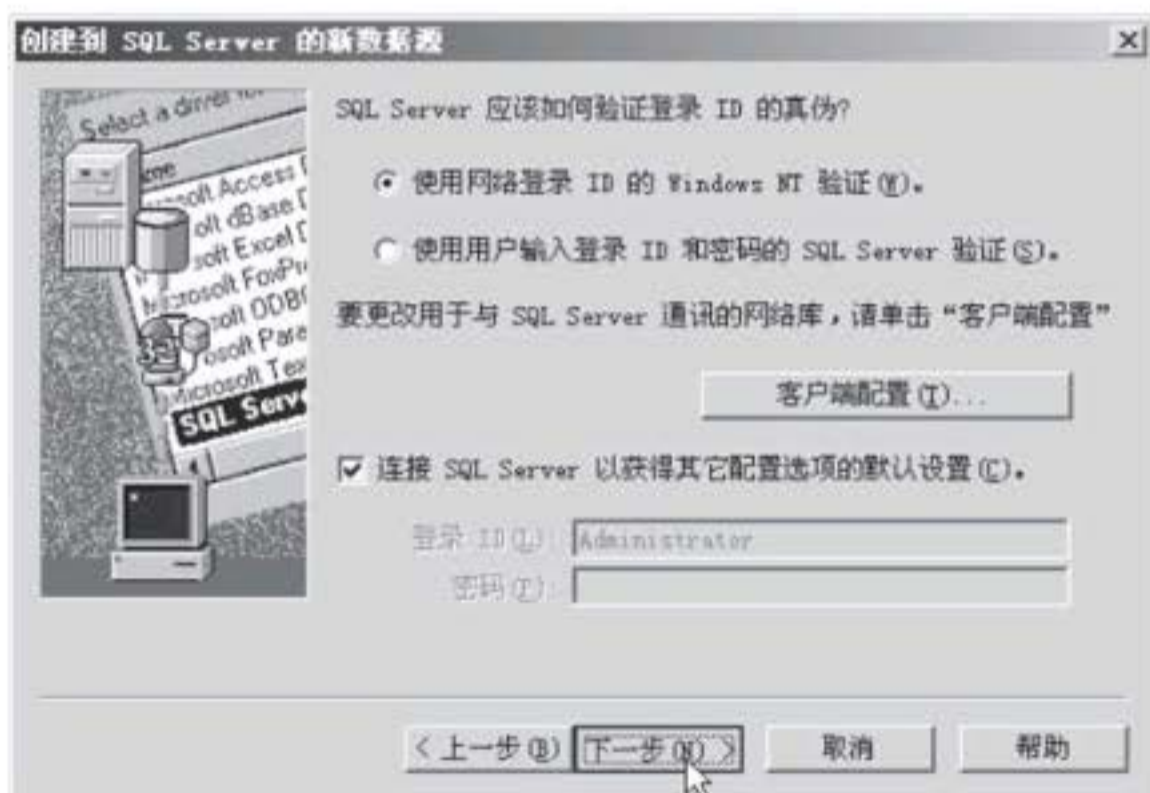


图 6-22 设置登录方式

(5) 选择数据库

单击“下一步”按钮,在弹出的窗口中选择数据库,默认是“master”,此处选择“pubs”数据库。如图 6-23 所示。

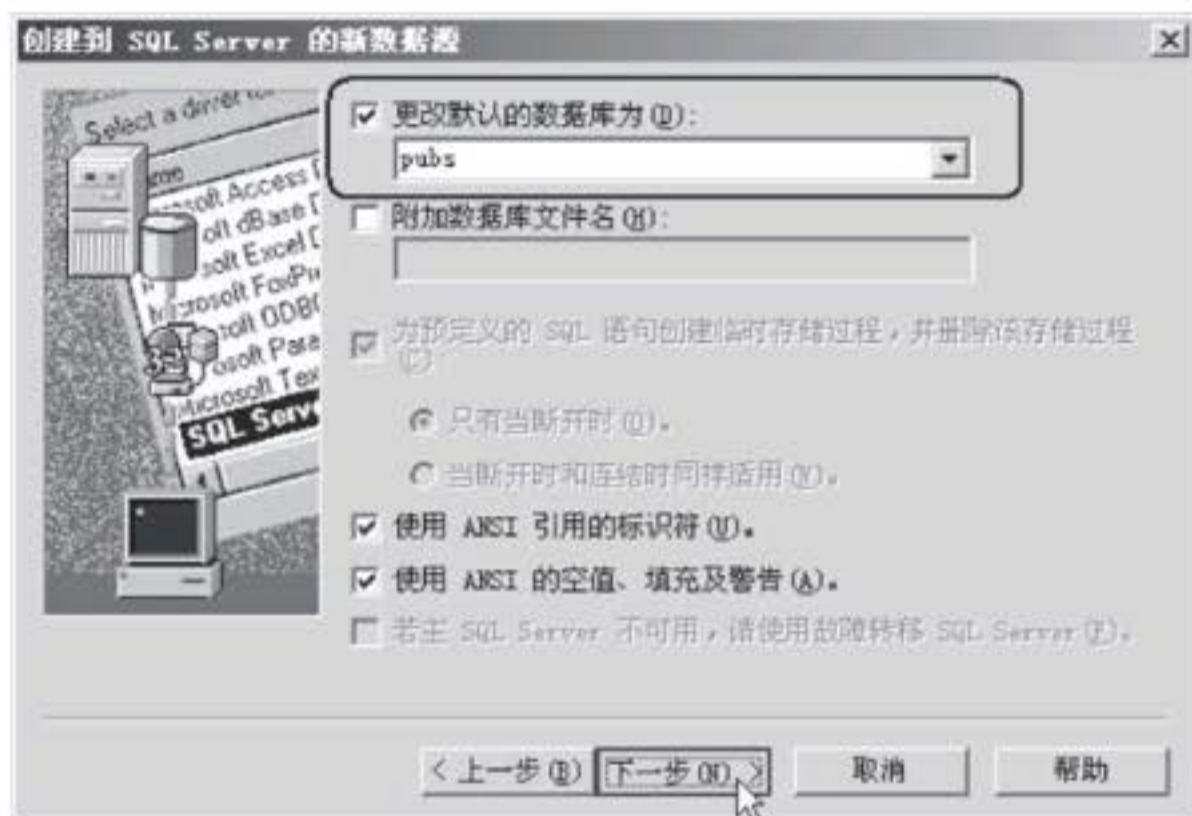


图 6-23 选择数据库

(6) 设置其他信息

单击“下一步”按钮，在弹出的窗口中可以设置数据库的字符集、日志文件等。通常可以保持默认。如图 6-24 所示。

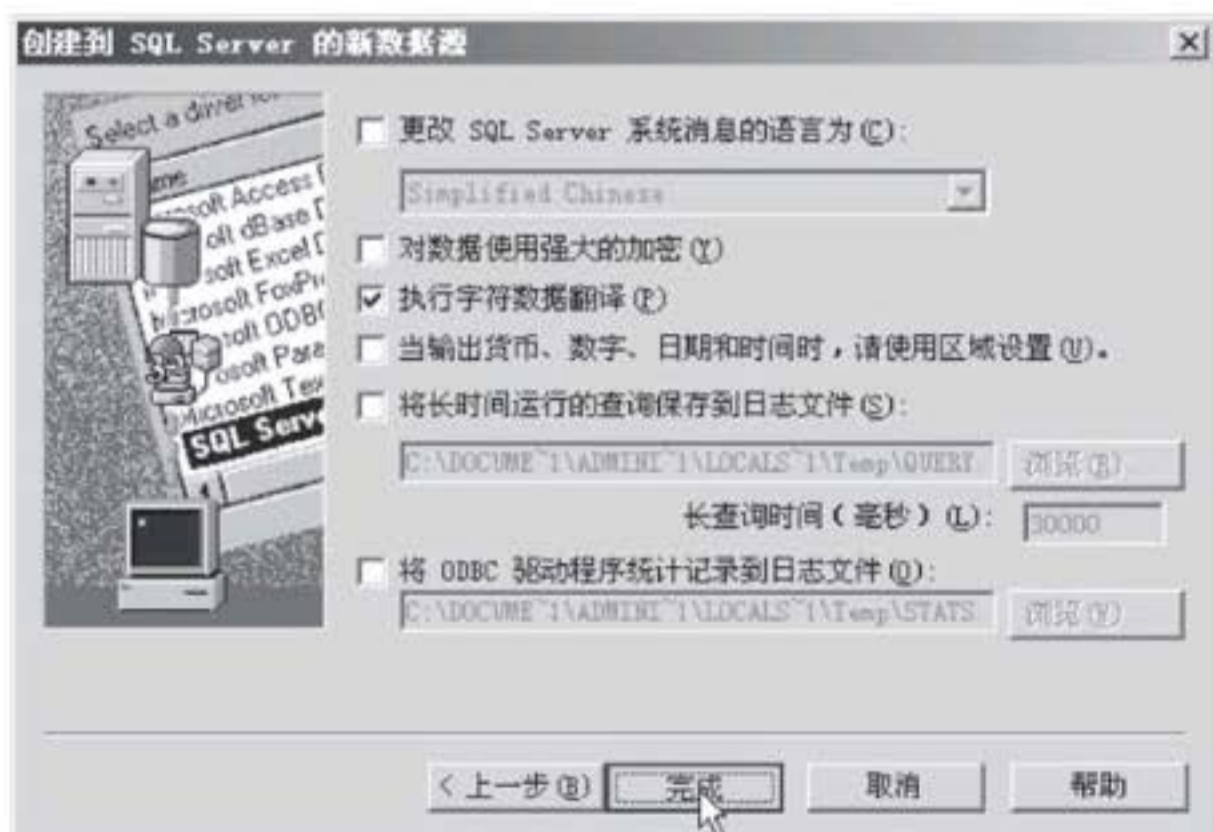


图 6-24 设置其他信息

(7) 测试数据源

单击“完成”按钮，将弹出测试数据源窗口，如图 6-25 所示。

单击“测试数据源”按钮，弹出测试窗口，如果测试成功，将弹出如图 6-26 所示窗口。

单击“确定”按钮，回到数据源管理器界面，可以看到成果：数据源“my”已经建立成功了。如图 6-27 所示。



图 6 - 25 测试数据源

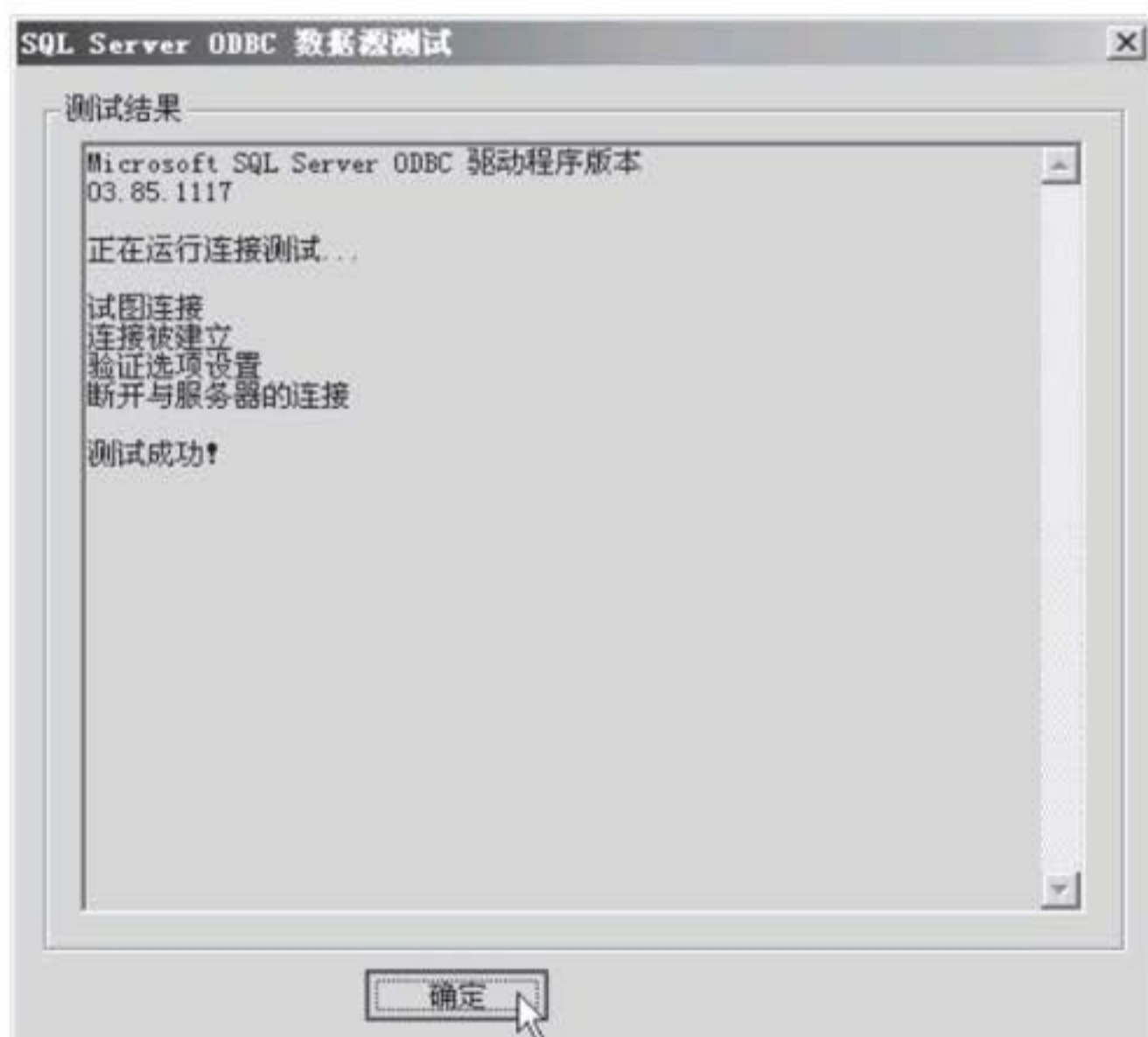


图 6 - 26 测试数据源



图 6-27 建立数据源成功

6.3.2 访问数据库

访问 SQL Server 的方法与访问 Access 是一致的,这里不再重复介绍方法。笔者给一些例子以便于读者学习。

案例名称:获得元数据

程序名称:ak.rb

```
require 'dbi'
dbh= DBI.connect("DBI:ADO:my","sa","")
begin
  query = "select * from authors"
  sth = dbh.execute(query)
  if sth.column_names.size == 0 then
    puts "no result set"
    printf "Number of rows affected: % d\n", sth.rows
  else
    puts "Query has a result set"
    rows = sth.fetch_all
    printf "Number of rows: % d\n", rows.size
    printf "Number of columns: % d\n", sth.column_names.size
    sth.column_info.each_with_index do |info, i|
      printf "- - - Column % d (% s) - - - \n", i, info.name
    end
  end
end
sth.finish
rescue
  p "error!"
ensure
```

```
dbh.disconnect if dbh
end
```

运行结果如图 6-28 所示。

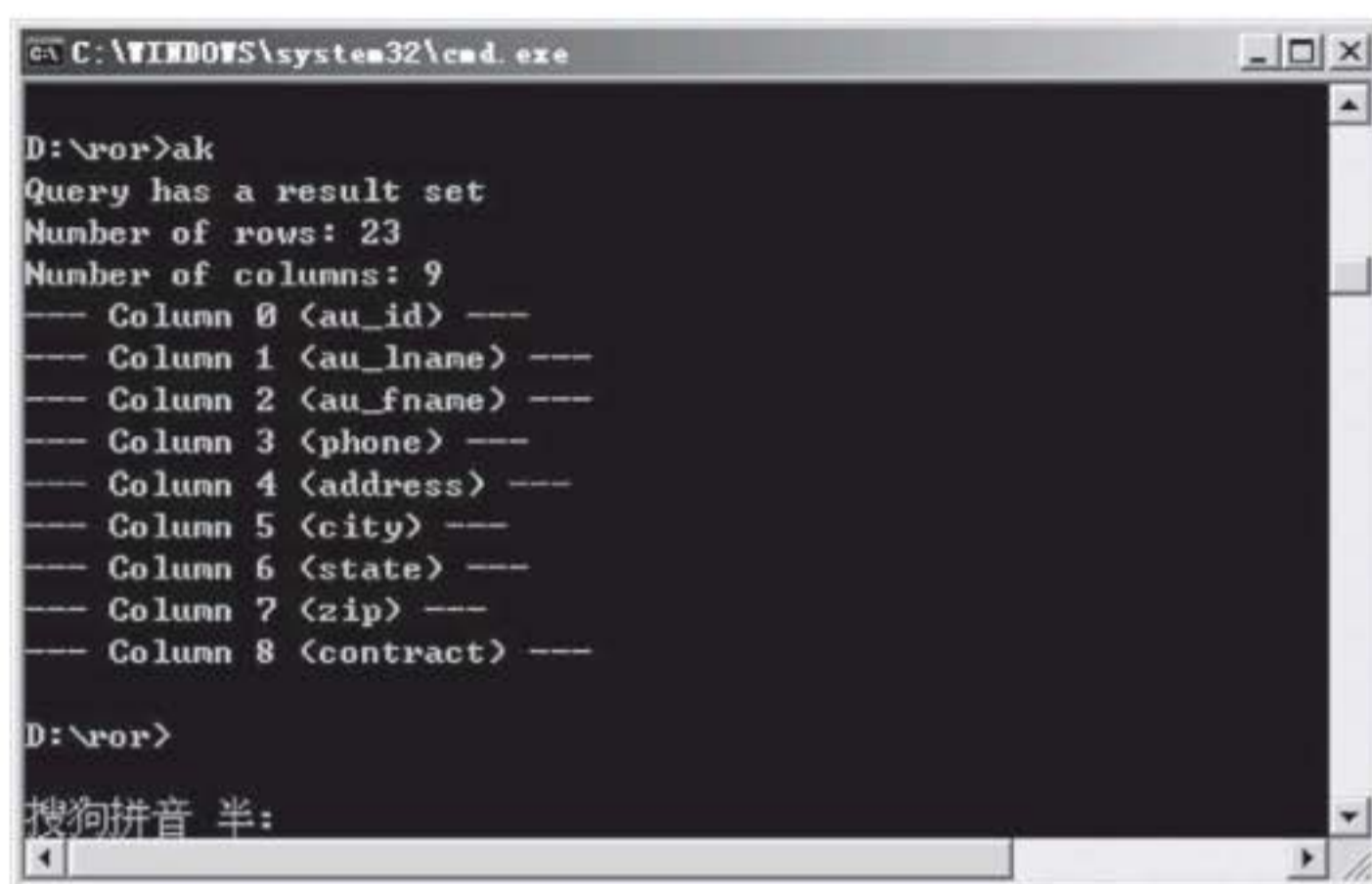


图 6-28 获得元数据

案例名称:执行查询

程序名称:al.rb

```
require 'dbi'
dbh= DBI.connect("DBI:ADO:my","sa","")
begin
  query = "select * from authors"
  sth = dbh.execute(query)
  while row = sth.fetch do
    printf "au_id % s, au_lname: % s, au_fname: % s\n", row[0], row[1], row[2]
  end
  sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end
```

运行结果如图 6-29 所示。

案例名称:执行查询

程序名称:am.rb

```
require 'dbi'
dbh= DBI.connect("DBI:ADO:my","sa","")
begin
  query = "select * from authors"
```

```

sth = dbh.execute(query)
sth.fetch do |row|
  printf "au_id: % s, au_lname: % s, au_fname: % s\n", row[0], row[1], row[2]
end
sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

```

C:\WINDOWS\system32\cmd.exe
D:\ror>al
au_id 172-32-1176, au_lname: White, au_fname: Johnson
au_id 213-46-8915, au_lname: Green, au_fname: Marjorie
au_id 238-95-7766, au_lname: Carson, au_fname: Cheryl
au_id 267-41-2394, au_lname: O'Leary, au_fname: Michael
au_id 274-80-9391, au_lname: Straight, au_fname: Dean
au_id 341-22-1782, au_lname: Smith, au_fname: Meander
au_id 409-56-7008, au_lname: Bennet, au_fname: Abraham
au_id 427-17-2319, au_lname: Dull, au_fname: Ann
au_id 472-27-2349, au_lname: Gringlesby, au_fname: Burt
au_id 486-29-1786, au_lname: Locksley, au_fname: Charlene
au_id 527-72-3246, au_lname: Greene, au_fname: Morningstar
au_id 648-92-1872, au_lname: Blotchet-Halls, au_fname: Reginald
au_id 672-71-3249, au_lname: Yokomoto, au_fname: Akiko
au_id 712-45-1867, au_lname: del Castillo, au_fname: Innes
au_id 722-51-5454, au_lname: DeFrance, au_fname: Michel
au_id 724-08-9931, au_lname: Stringer, au_fname: Dirk
au_id 724-80-9391, au_lname: MacFeather, au_fname: Stearns
au_id 756-30-7391, au_lname: Karsen, au_fname: Livia
au_id 807-91-6654, au_lname: Panteley, au_fname: Sylvia
au_id 846-92-7186, au_lname: Hunter, au_fname: Sheryl
au_id 893-72-1158, au_lname: McBadden, au_fname: Heather
au_id 899-46-2035, au_lname: Ringer, au_fname: Anne
au_id 998-72-3567, au_lname: Ringer, au_fname: Albert

D:\ror>
搜狗拼音 半:

```

图 6-29 执行查询

运行结果如图 6-30 所示。

案例名称:执行查询

程序名称:an.rb

```

require 'dbi'
dbh= DBI.connect("DBI:ADO:my","sa","")
i= 0
begin
  query = "select * from authors"
  sth = dbh.execute(query)
  sth.each do |row|
    i+ = 1
    break if i = 3
    p "[ # {i} ]"
  end
end

```

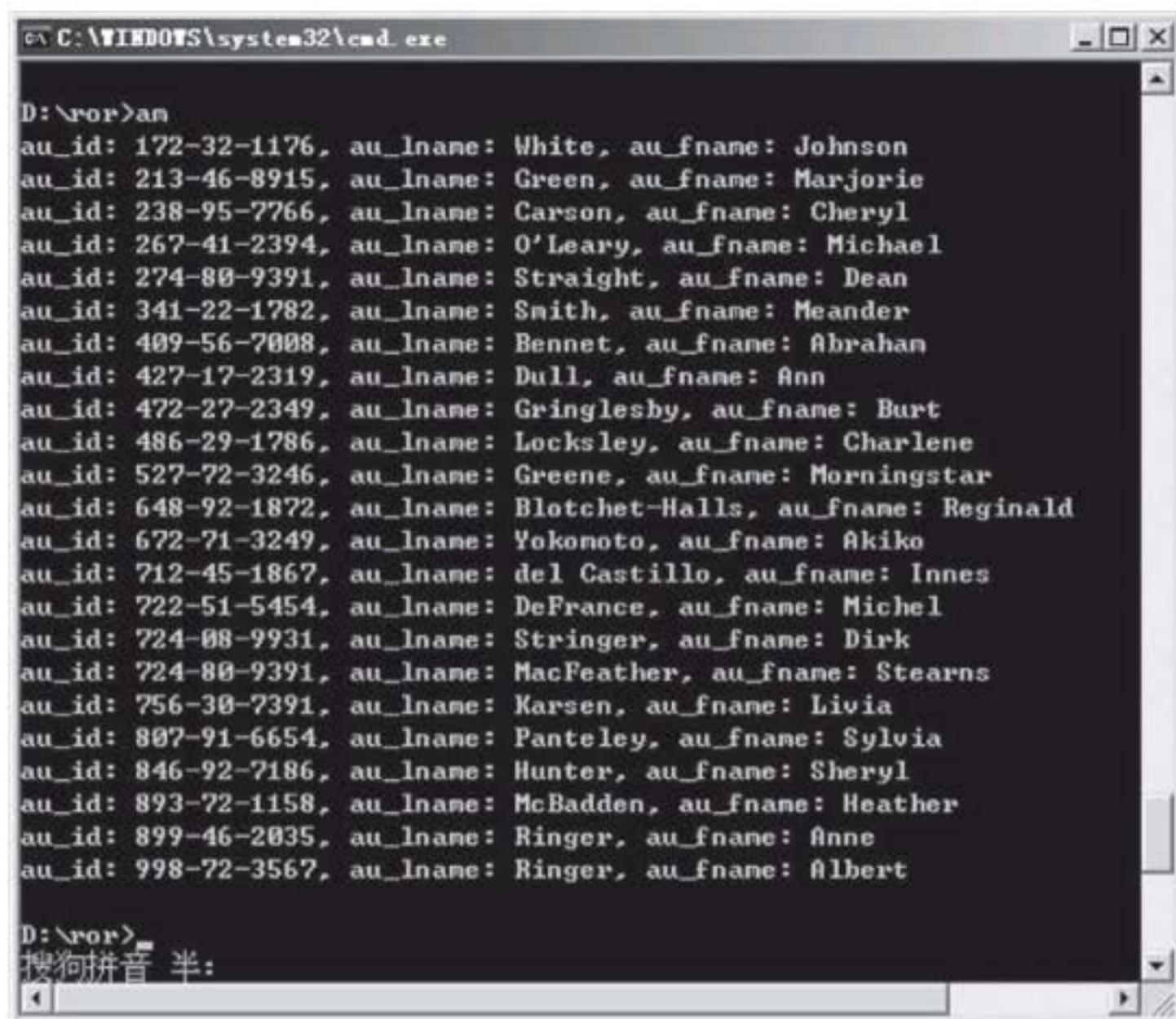


图 6-30 执行查询

```

    row.each_with_name do |val, name|
      printf "% s: % s\n", name, val.to_s
    end
    print "\n"
  end
  sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

运行结果如图 6-31 所示。

案例名称:执行查询

程序名称:ao.rb

```

require 'dbi'
dbh= DBI.connect("DBI:ADO:my","sa","")
begin
  query = "select * from authors"
  rows = dbh.select_all(query)
  rows.each{|e|
    printf "% s, % s, % s\n", e[0], e[1], e[2]
  }
end

```

```

}
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

```

C:\WINDOWS\system32\cmd.exe
D:\ror>an
"[ 1 ]"
au_id: 172-32-1176
au_lname: White
au_fname: Johnson
phone: 408 496-7223
address: 10932 Bigge Rd.
city: Menlo Park
state: CA
zip: 94025
contract: true

"[ 2 ]"
au_id: 213-46-8915
au_lname: Green
au_fname: Marjorie
phone: 415 986-7020
address: 309 63rd St. #411
city: Oakland
state: CA
zip: 94618
contract: true

D:\ror>
搜狗拼音 半:

```

图 6-31 执行查询

运行结果如图 6-32 所示。

```

C:\WINDOWS\system32\cmd.exe
D:\ror>an
172-32-1176, White, Johnson
213-46-8915, Green, Marjorie
238-95-7766, Carson, Cheryl
267-41-2394, O'Leary, Michael
274-80-9391, Straight, Dean
341-22-1782, Smith, Meander
409-56-7008, Bennet, Abraham
427-17-2319, Dull, Ann
472-27-2349, Gringlesby, Burt
486-29-1786, Locksley, Charlene
527-72-3246, Greene, Morningstar
648-92-1872, Blotchet-Halls, Reginald
672-71-3249, Yokamoto, Akiko
712-45-1867, del Castillo, Innes
722-51-5454, DeFrance, Michel
724-08-9931, Stringer, Dirk
724-80-9391, MacFeather, Stearns
756-30-7391, Karsen, Livia
807-91-6654, Panteley, Sylvia
846-92-7186, Hunter, Sheryl
893-72-1158, McBadden, Heather
899-46-2035, Ringer, Anne
998-72-3567, Ringer, Albert

D:\ror>
搜狗拼音 半:

```

图 6-32 执行查询

6.4 访问 MySQL 数据库

MySQL/Ruby 模块可以支持 Ruby 访问 MySQL 数据库。它有 Unix 和 Windows 的版本,安装方法不同。在 Windows 上可以安装 Unix 版本的模块,笔者在此不作介绍。这里,笔者介绍其在 Windows 下的安装和使用方法。这个方法,是使用 Ruby 来访问 MySQL 数据库的主要方法。

6.4.1 下载和安装 MySQL/Ruby 模块

(1) 下载 MySQL/Ruby 模块

进入 <http://www.vandenburg.net/> 网站,从这里可以下载到 Windows 版本的 MySQL/Ruby 模块,如图 6-33 所示。

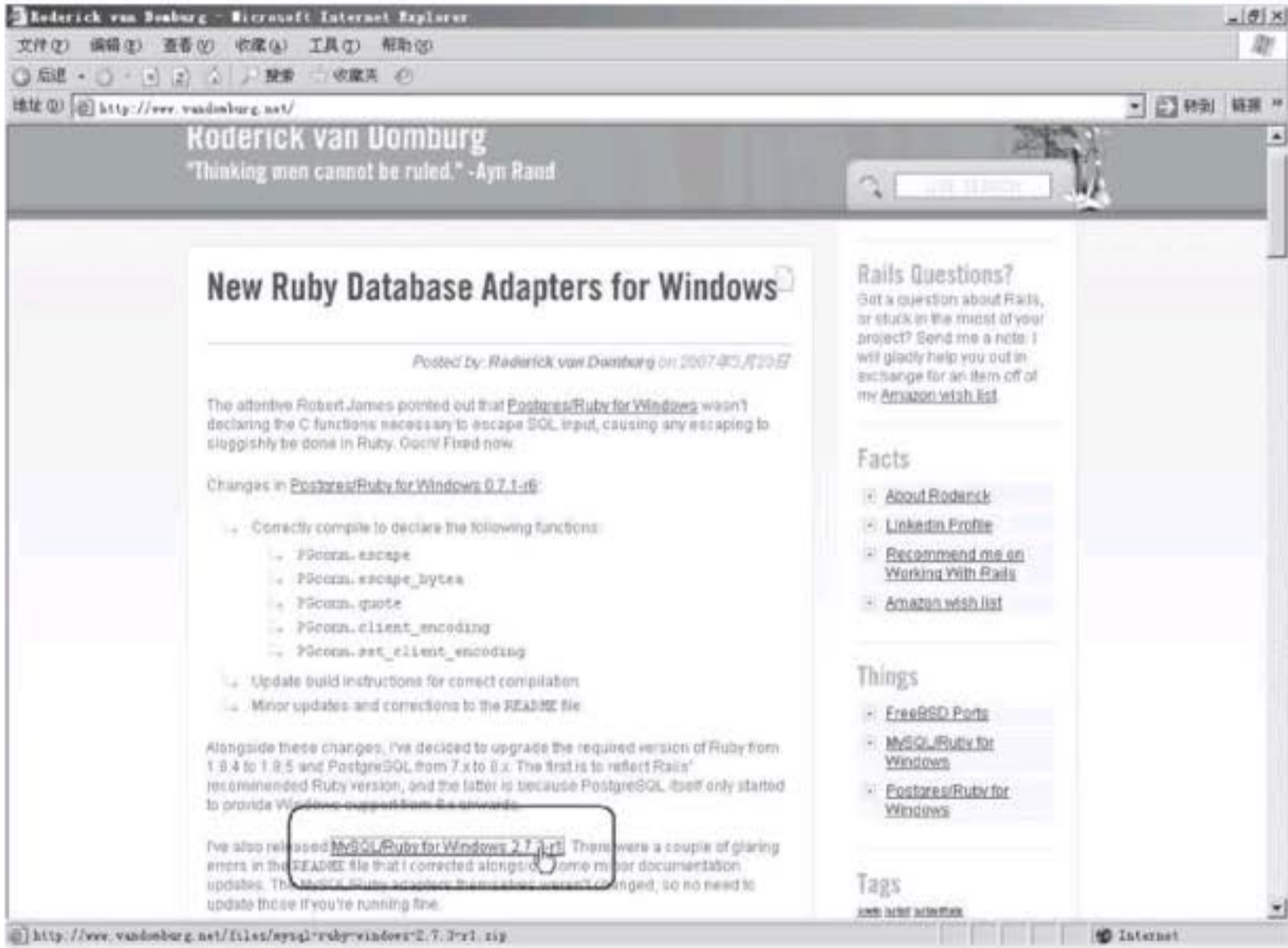


图 6-33 下载 MySQL/Ruby 模块

当前(2007-6-24)最新版本是 2.7.3-r1。下载结果如图 6-34 所示。



图 6-34 MySQL/Ruby 模块

(2) 安装 MySQL/Ruby 模块

将软件包解压缩,如图 6-35 所示。

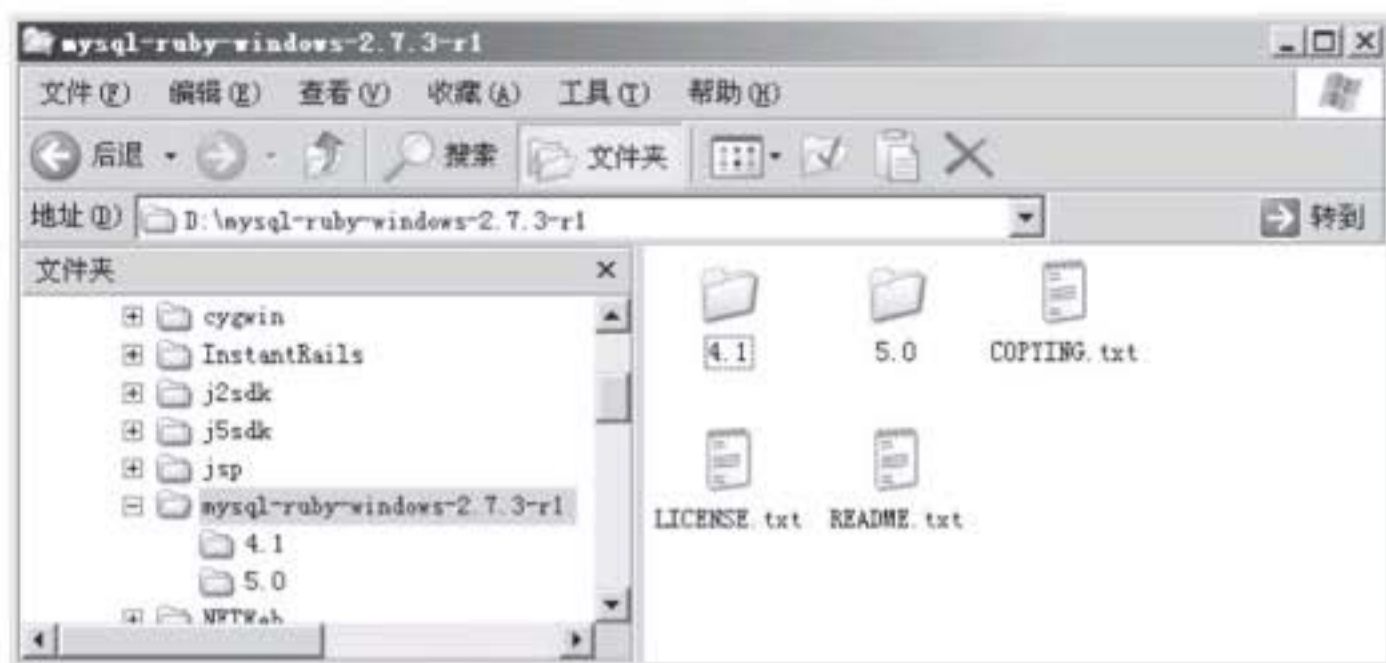


图 6-35 MySQL/Ruby 模块

在 5.0 目录下,有两个文件,任取其一,改名为“mysql.so”,然后复制到 D:\ruby\lib\ruby\site_ruby\1.8\i386-msvcrt 目录下。如图 6-36 和 7-37 所示。

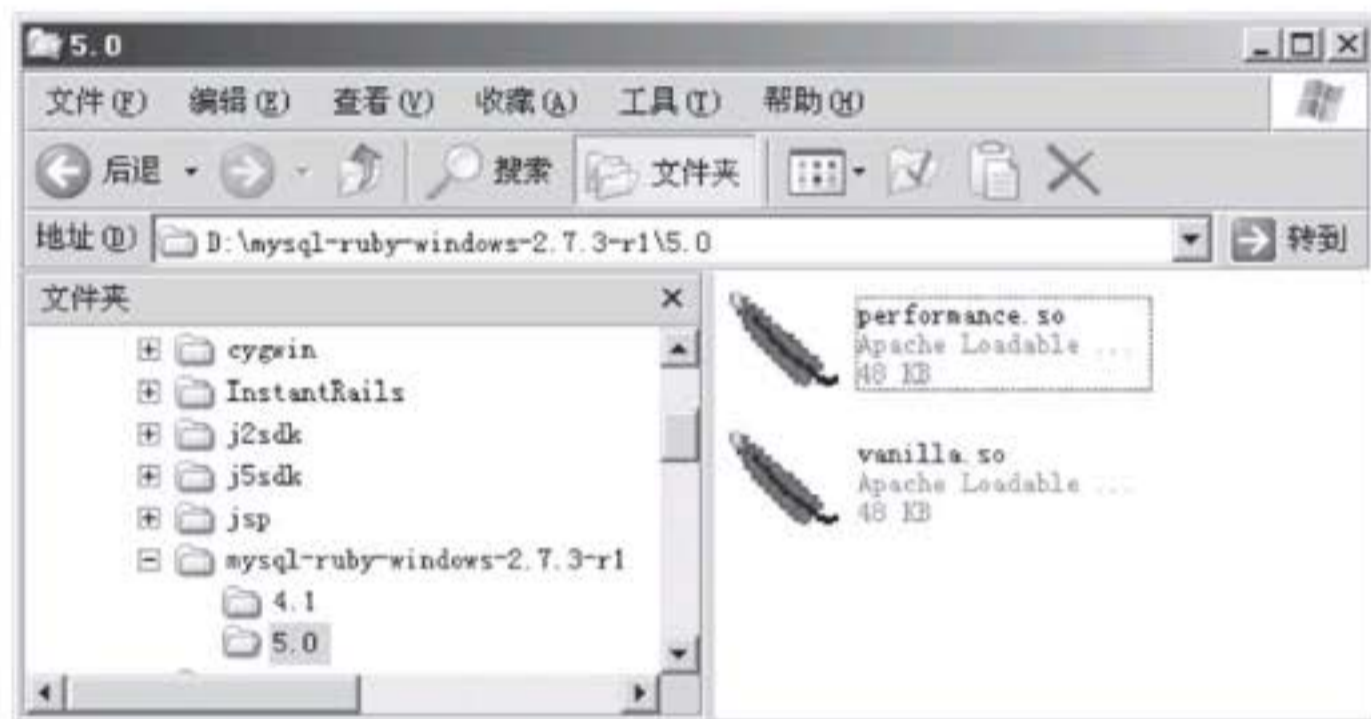


图 6-36 安装 MySQL/Ruby 模块



图 6-37 安装 MySQL/Ruby 模块

最后,将 C:\mysql\bin 目录下的 libmySQL.dll 复制到 D:\ruby\bin 目录下。如图 6-38 所示。这样就完成了安装。

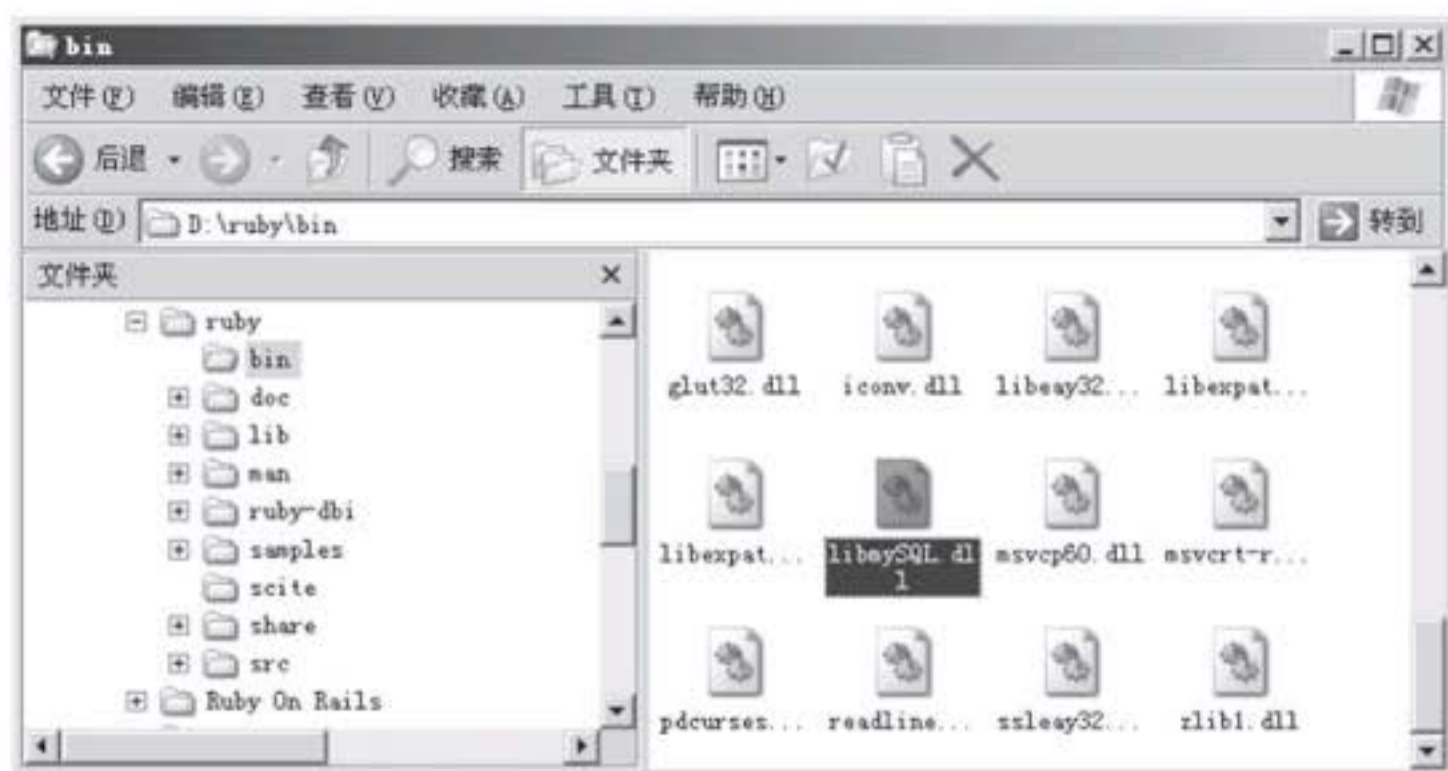


图 6-38 安装 MySQL/Ruby 模块

(3) 测试安装

建立访问 MySQL 的文件。

案例名称:访问 MySQL

程序名称:mysql.rb

```
require 'mysql'
puts Mysql::VERSION
```

运行结果若如图 6-39 所示,则证明 MySQL/Ruby 安装成功。



图 6-39 访问 MySQL

6.4.2 使用 MySQL 模块进行数据库访问

本节介绍 MySQL 模块的相关方法。

(1) 连接数据库

连接数据库要使用 Mysql 类及其实例。

Mysql 类具有如下类方法:

init()方法用于返回 Mysql 对象。

如下方法用于连接数据库,同时返回 Mysql 对象。

```
real_connect(host= nil, user= nil, passwd= nil, db= nil, port= nil, sock= nil, flag= nil)
connect(host= nil, user= nil, passwd= nil, db= nil, port= nil, sock= nil, flag= nil)
new(host= nil, user= nil, passwd= nil, db= nil, port= nil, sock= nil, flag= nil)
```

其中主要参数的意义为: host 是主机, user 是用户名, passwd 是密码, db 是数据库名, port 是端口号。

比如, 用 `con = Mysql.new('localhost', 'root', 'tianen', 'my')` 连接本机的 MySQL 数据库服务器的 my 数据库。

此外, 还可以通过 `get_client_info()` 和 `client_info()` 方法以字符串获得数据库服务器客户端的版本信息, 而 `get_client_version()` 和 `client_version()` 则以数字形式获得数据库服务器客户端的版本信息。

还可以通过 Mysql 类的如下方法连接数据库:

```
real_connect(host= nil, user= nil, passwd= nil, db= nil, port= nil, sock= nil, flag= nil)
connect(host= nil, user= nil, passwd= nil, db= nil, port= nil, sock= nil, flag= nil)
```

为了后面的学习, 这里新建一个数据库“my”, 并新建数据表“test”。代码如下:

程序名称: my.sql

```
create database my;
use my;
create table test
(
    name varchar(20),
    hobby varchar(20)
);
insert into test values('tianen','thinking');
insert into test values('xiaoyue','laughing');
```

执行 SQL 语句, 如图 6-40 所示。



图 6-40 建立测试表格

进入 MySQL, 可以查看建立表格的结果, 如图 6-41 所示。

案例名称: 连接数据库

程序名称: ap.rb

```
require 'mysql'
con = Mysql.new('localhost', 'root', 'tianen', 'my')
p con.get_client_info()
p con.get_client_version()
con.close
```

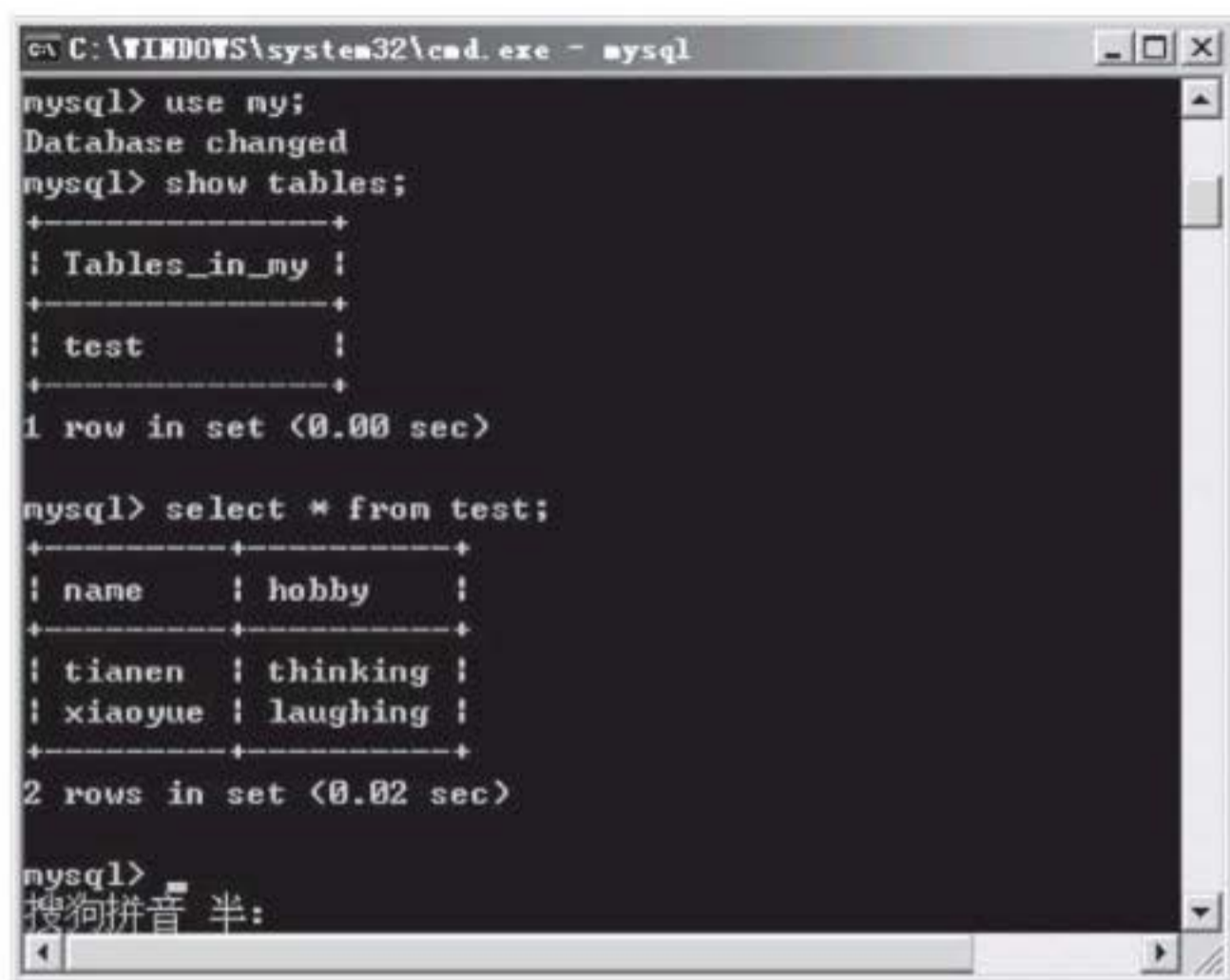


图 6-41 建立测试表格

运行结果如图 6-42 所示。



图 6-42 连接数据库

(2) 执行数据操纵语句

执行数据操纵语句要使用 Mysql 类的实例方法 query 和 real_query 如

```

con = Mysql.new('localhost','root','tianen','my')
con.query("insert into test values('a','b')")
con.real_query("insert into test values('a','b')")

```

使用 affected_rows() 方法返回查询影响的行数。使用 info() 方法获得上次查询的信息。insert_id() 方法获得上一个自动增长的字段值。

案例名称: 执行数据操纵语句

程序名称: aq.rb

```

require 'mysql'
con = Mysql.new('localhost','root','tianen','my')

```

```
con.prepare("insert into test values('lucy','tea')")
p con.affected_rows()
con.close
```

运行后,向数据库中添加一条记录。结果如图 6-43 和 7-44 所示。



图 6-43 执行数据操纵语句



图 6-44 执行数据操纵语句

(3) 执行数据查询语句

执行数据查询语句要使用 Mysql 类的实例方法 query 和 real_query, 这些方法返回 Mysql::Result 类的实例, 保存返回的结果。

案例名称: 执行数据查询语句

程序名称: ar.rb

```
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
p con.affected_rows()
rs.free()
con.close
```

程序运行结果如图 6-45 所示。



图 6-45 执行数据查询语句

(4) 处理查询结果

查询结果被保存在 Mysql::Result 类的实例中,结果的每个字段保存在 Mysql::Field 对象中。

Mysql 类具有如下的实例方法,用于控制结果集:

more_results? () 如果当前执行的查询还有结果集,就返回 true。

next_result() 获得下一个结果集,然后可以执行 store_result() 来获得结果集。

field_count() 取得字段数目。

use_result() 获得结果集实例。

list_fields(table, field=nil) 返回所有字段。

list_tables(table=nil) 返回所有表格(数组)。

Mysql::Result 类具有如下实例方法:

data_seek(offset) 用于定位记录行。

fetch_field() 用于取得下一个 Mysql::Field 对象。

fetch_fields() 以数组形式返回 Mysql::Field 对象。

fetch_field_direct(fieldnr) 以记录号码为参数,获得 Mysql::Field 对象。

fetch_lengths() 获得字段长度,返回一个数组。

field_seek(offset) 查找某个字段。

field_tell() 获得字段的位置。

num_fields() 获得字段数目。

fetch_row() 获取记录中的一行,返回一个数组。

fetch_hash(with_table=false) 获取记录中的一行,返回一个散列表。如果 with_table 参数为“true”,那么散列表的键格式为“tablename.fieldname”。

num_rows() 获得记录数目。

row_tell() 获得记录的位置。

row_seek(offset) 查找某个记录。

free() 释放记录集。

此外,Mysql::Result 类具有如下迭代器。

each() { |x| ... } x 是列值数组。

each_hash(with_table=false) { |x| ... } x 是列值散列表,散列表的键是列名。

Mysql::Field 类具有如下实例方法:

hash() 以散列表形式获取字段。

is_not_null? () 如果字段被定义为非空,则返回 true。

is_num? () 如果字段类型是数字,则返回 true。

is_pri_key? () 如果字段是主键,则返回 true。

inspect() 返回“#<Mysql::Field:fieldname>”

Mysql::Field 类还具有如下只读属性:

name 字段名

table 表格名

def 默认值
type 字段类型
length 字段长度
max_length 字段最大长度
flags 字段标志
decimals 字段精度

联合使用 Mysql::Result 类和 Mysql::Field 类可以很好地处理查询结果。下面举一些例子。

案例名称:处理查询结果

程序名称:as.rb

```
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
rs.each_hash(with_table= false) {|x|
p x['name'] + ":" + x['hobby']
}
rs.free()
con.close
```

程序运行结果如图 6-46 所示。



图 6-46 处理查询结果

程序名称:at.rb

```
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
rs.data_seek(0)
p '- - - - -'
p rs.fetch_field()
f= rs.fetch_field()
p f.name
p f.table
p f.def
p f.type
```

```

p f.length
p f.max_length
p f.flags
p f.decimals
p '-----'
p rs.fetch_field()
rs.free()
con.close

```

程序运行结果如图 6-47 所示。

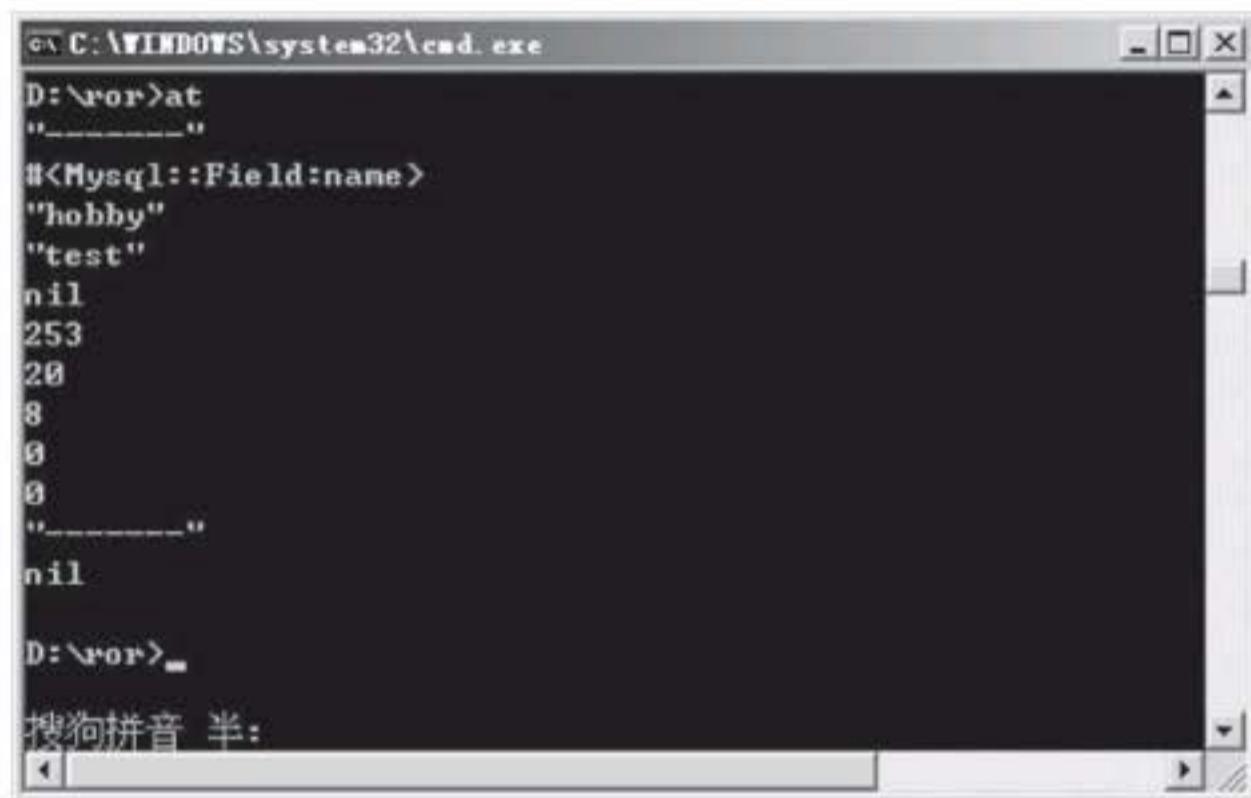


图 6-47 处理查询结果

程序名称:au.rb

```

require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
p rs.num_fields()
p '-----'
num= rs.num_rows()
for i in 0..num- 1
  rs.data_seek(i)
  a= rs.fetch_row()
  p a[0] + ":" + a[1]
end
p '-----'
for i in 0..num- 1
  rs.data_seek(i)
  a= rs.fetch_hash()
  p a['name'] + ":" + a['hobby']
end
rs.free()
con.close

```

程序运行结果如图 6-48 所示。



图 6-48 处理查询结果

除了使用 `Mysql::Result` 之外,还可以用 `Mysql::Stmt` 类处理查询结果。使用 `Mysql` 对象的 `prepare` 方法可以获得 `Mysql::Stmt` 对象。

`Mysql::Stmt` 对象具有如下方法:

`affected_rows()` 获得被影响的记录数目。

`num_rows()` 获得记录数目。

`row_tell()` 获得记录位置。

`field_count()` 获得字段数目。

`insert_id()` 获得插入的自动增长 id 值。

`row_seek(offset)` 定位记录。

`data_seek(offset)` 定位记录。

`prepare(q)` 准备语句。

`execute(arg, ...)` 执行语句。

`fetch()` 获得下一条记录。

`result_metadata()` 获得结果集元数据。

`free_result()` 释放结果集。

`close()` 关闭。

`each()` `{|x| ...}` 迭代器

举例如下:

程序名称:av.rb

```
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
st = con.prepare("insert into test values (?,?)")
st.execute("jim","eat")
st.prepare("select * from test")
st.execute
while f= st.fetch() do
```

```

p f
end
st.close
con.close

```

程序运行结果如图 6-49 所示。



图 6-49 处理查询结果

查看数据库,如图 6-50 所示。

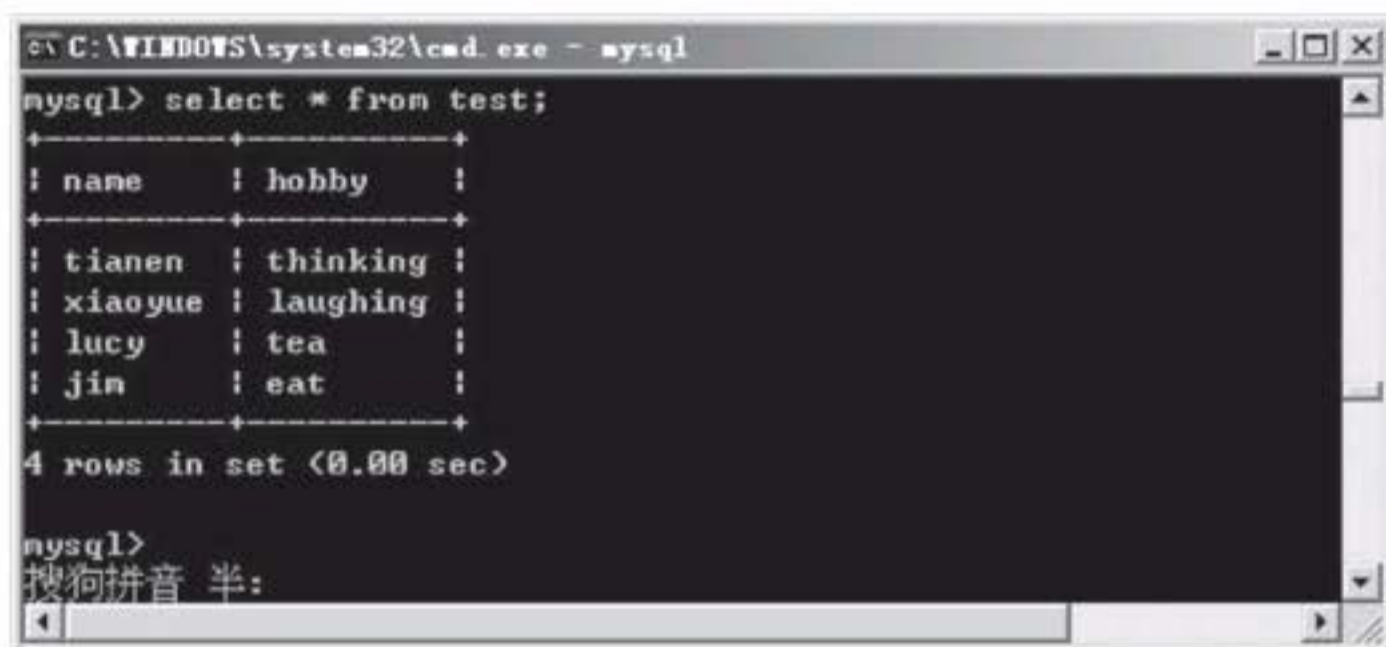


图 6-50 处理查询结果

程序名称:aw.rb

```

require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
st = con.prepare("select * from test")
st.execute
p st.result_metadata()
num= st.num_rows()
for i in 0..num- 1
  st.data_seek(i)
  f= st.fetch()
  p f[0] + ":" + f[1]
end
st.close
con.close

```

查看数据库,如图 6-51 所示。



图 6-51 处理查询结果

(5) 关闭数据库连接

使用:Mysql 实例的 close()方法。

(6) 其他方法

除了可以进行标准的数据操纵、查询之外,Mysql 类还有一些辅助的实例方法用来完善数据库操作。

change_user(user=nil, passwd=nil, db=nil) 用来切换数据库用户。

character_set_name()获得字符集名称。

select_db(db) 选择数据库

create_db(db) 建立数据库。

drop_db(db) 删除数据库。

errno()获得错误号码。

error()获得错误信息。

escape_string(str) 和 quote(str) 为输入的 insert/update 语句做转义。

get_client_info()和 client_info()获得数据库客户端版本信息。

get_client_version()和 client_version()获得数据库客户端版本(数字)。

get_host_info()和 host_info()获取连接信息。

get_proto_info()和 proto_info()获取连接协议。

get_server_info()和 server_info()获取数据库服务器版本信息。

get_server_version()server_version()获取数据库服务器版本(数字)。

thread_id()获取线程 id。

kill(id) 杀死线程。

list_dbs(db=nil) 获取数据库列表。

ping()检查服务器是否连通。

shutdown()关闭服务器。

stat()返回服务器状态。

Mysql::Error 类具有两个只读的实例属性:error 和 errno,分别用来返回错误信息和错误的号码。

Mysql::Time 类用来操作时间。

具有类方法 new(year=0,month=0,day=0,hour=0,minute=0,second=0,neg=false,second_part=0) 和如下的实例属性:year,month,day,hour,minute,second,neg,second_part,

error number。

表 6-1 显示了 MySQL 的数据类型与 Ruby 的类的对应关系。

表 6-1 MySQL 的数据类型与 Ruby 的类的对应关系

MySQL 的数据类型	Ruby 的类
TINYINT, SMALLINT, MEDIUMINT, YEAR	Fixnum
INT, BIGINT	Fixnum or Bignum
FLOAT, DOUBLE	Float
DECIMAL	String
DATE, DATETIME, TIMESTAMP, TIME	Mysql::Time
CHAR, VARCHAR, BINARY, VARBINARY, TINYBLOB, TINYTEXT, TINYBLOB, TINYTEXT, MEDIUMBLOB, MEDIUMTEXT, LONG-BLOB, LONGTEXT, ENUM, SET, BIT	String
NULL	NilClass

(7) 事务处理

Mysql 类具有如下实例方法用来实现事务处理：

autocommit(mode) 设定是否启用自动事务处理。

commit()提交事务。

rollback()回滚事务。

通常，在执行多条数据操纵语句时，要使用事务处理。

案例名称：事务处理

程序名称：ax.rb

```
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
con.autocommit(false)
begin
  con.query("insert into test values('a','b')")
  con.query("insert into test values('c','d')")
  con.commit
rescue
  p "error!"
  con.rollback
ensure
  con.close
end
con = Mysql.new('localhost','root','tianen','my')
st = con.prepare("select * from test")
st.execute
num= st.num_rows()
for i in 0..num- 1
  st.data_seek(i)
  f= st.fetch()
  p f[0] + ":" + f[1]
```

```
end
st.close
con.close
```

查看数据库,如图 6-52 所示。



图 6-52 处理查询结果

6.4.3 安装 DBI

要使用 DBI 访问 MySQL 数据库,需要安装 MySQL 模块,DBI 软件包和 MySQL Gem。

(1) 下载 MySQL Gem 软件包

要使得 Ruby 可以访问 MySQL,要安装 MySQL Gem 软件包。进入 <http://rubyforge.net/projects/mysql-win/>,如图 6-53 所示。



图 6-53 下载 MySQL Gem 软件包

单击“下载”链接,进入下载页面。选择版本 2.7.3。如图 6-54 所示。



图 6-54 下载 MySQL Gem 软件包

下载结果(mysql-2.7.3-mswin32.gem)如图 6-55 所示。

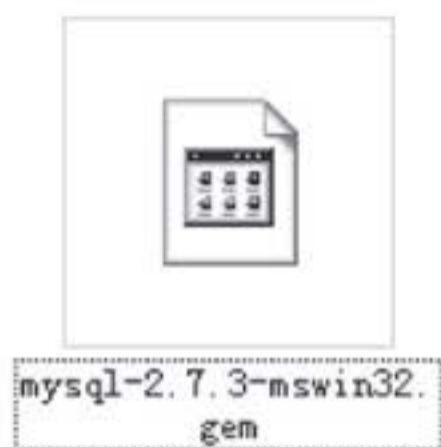


图 6-55 下载 MySQL Gem 软件包

(2) 安装 Ruby/DBI 软件包

将 DBI 软件包(dbi-0.1.1.tar.gz)解压缩到 ruby-dbi 文件夹下,将 MySQL Gem(mysql-2.7.3-mswin32.gem)也复制到 ruby-dbi 文件夹中。将文件夹复制到 D:/ruby 目录下,如图 6-56 所示。

在 DOS 界面下,从命令行进入 D:/ruby/ruby-dbi 目录,执行命令“ruby setup.rb config --with=dbi,dbd_mysql”,如图 6-57 所示。

在命令行中输入:“ruby setup.rb setup”,如图 6-58 所示。

在命令行中输入:“ruby setup.rb install”,如图 6-59 所示。



图 6-56 安装 DBI 软件包

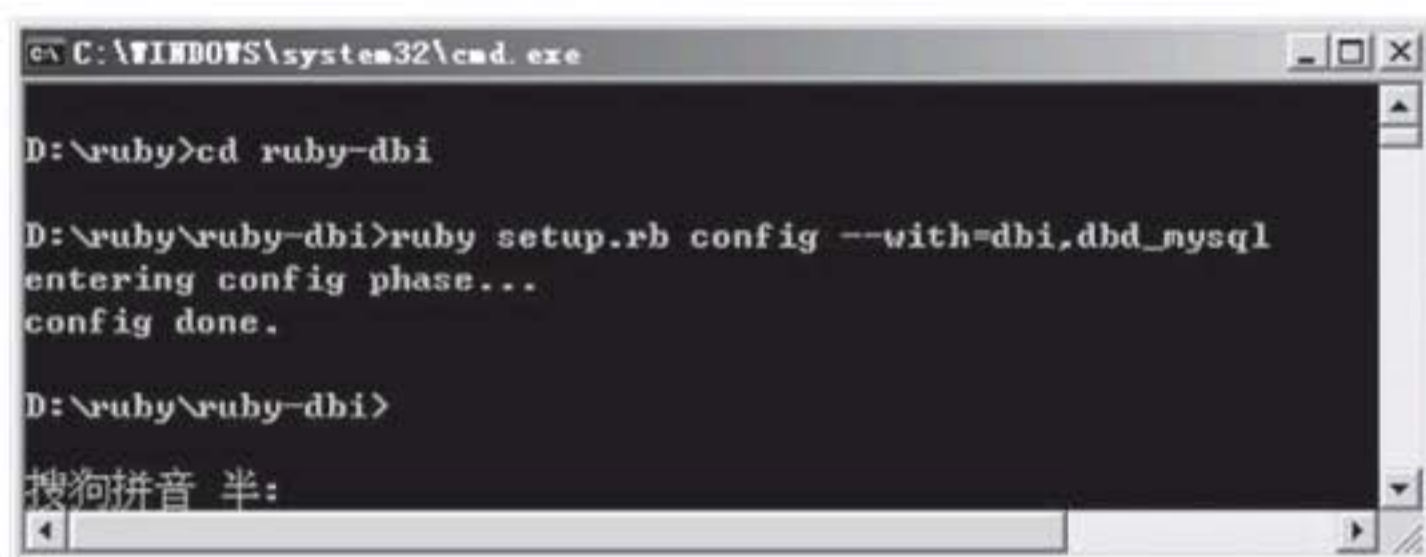


图 6-57 安装 DBI 软件包

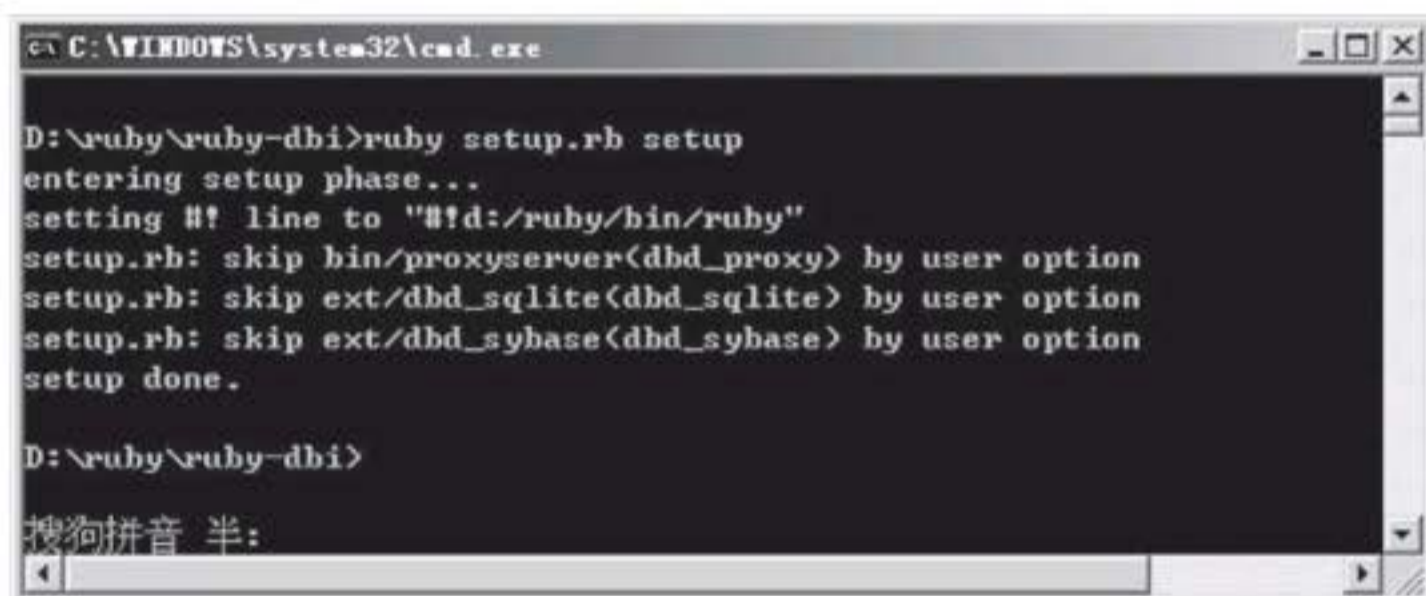


图 6-58 安装 DBI 软件包

注意:DBI 软件包包含有多个数据库的驱动程序,使用“`ruby setup.rb config --with=dbi,dbd_mysql`”是安装了 MySQL 的驱动程序,如果要安装 Oracle 的驱动程序需要使用“`ruby setup.rb config --with=dbi,dbd_oracle`”。

```
C:\WINDOWS\system32\cmd.exe

D:\ruby\ruby-dbi>ruby setup.rb setup
entering setup phase...
setting #! line to "#!d:/ruby/bin/ruby"
setup.rb: skip bin/proxyserver<dbd_proxy> by user option
setup.rb: skip ext/dbd_sqlite<dbd_sqlite> by user option
setup.rb: skip ext/dbd_sybase<dbd_sybase> by user option
setup done.

D:\ruby\ruby-dbi>ruby setup.rb install
entering install phase...
mkdir -p d:/ruby/bin
install sqlsh.rb d:/ruby/bin
setup.rb: skip bin/proxyserver<dbd_proxy> by user option
mkdir -p d:/ruby/lib/ruby/site_ruby/1.8/DBD/Mysql
install Mysql.rb d:/ruby/lib/ruby/site_ruby/1.8/DBD/Mysql/Mysql.rb
mkdir -p d:/ruby/lib/ruby/site_ruby/1.8/dbi
install columninfo.rb d:/ruby/lib/ruby/site_ruby/1.8/dbi
install row.rb d:/ruby/lib/ruby/site_ruby/1.8/dbi
install sql.rb d:/ruby/lib/ruby/site_ruby/1.8/dbi
install trace.rb d:/ruby/lib/ruby/site_ruby/1.8/dbi
install utils.rb d:/ruby/lib/ruby/site_ruby/1.8/dbi
install version.rb d:/ruby/lib/ruby/site_ruby/1.8/dbi
install dbi.rb d:/ruby/lib/ruby/site_ruby/1.8
setup.rb: skip ext/dbd_sqlite<dbd_sqlite> by user option
setup.rb: skip ext/dbd_sybase<dbd_sybase> by user option
install done.

D:\ruby\ruby-dbi>
```

图 6-59 安装 DBI 软件包

最后,执行“gem install mysql --local”命令,安装 MySQL Gem,如图 6-60 所示。

```
C:\WINDOWS\system32\cmd.exe

D:\ruby\ruby-dbi>gem install mysql --local
Successfully installed mysql, version 2.7.3
Installing ri documentation for mysql-2.7.3-mswin32...
Installing RDoc documentation for mysql-2.7.3-mswin32...
While generating documentation for mysql-2.7.3-mswin32
... MESSAGE: Unhandled special: Special: type=17, text="<!-- $Id: README.html,
v 1.20 2006-12-20 05:31:52 tonny Exp $ -->"
... RDOC args: --op d:/ruby/lib/ruby/gems/1.8/doc/mysql-2.7.3-mswin32/rdoc --excl
ude ext --main README --quiet ext README docs/README.html
<continuing with the rest of the installation>

D:\ruby\ruby-dbi>
```

图 6-60 安装 MySQL Gem

(3) 安装 MySQL/Ruby 模块

前面,已经安装好了 MySQL/Ruby 模块,所以这一步就可以省略了。不妨提供读者一个信息,在 <http://www.tmtm.org/en/mysql/ruby> 网站可以下载到最新的 Unix 版本 MySQL 模块,而且该网站提供了关于此模块的详细用法说明。如图 6-61 所示。

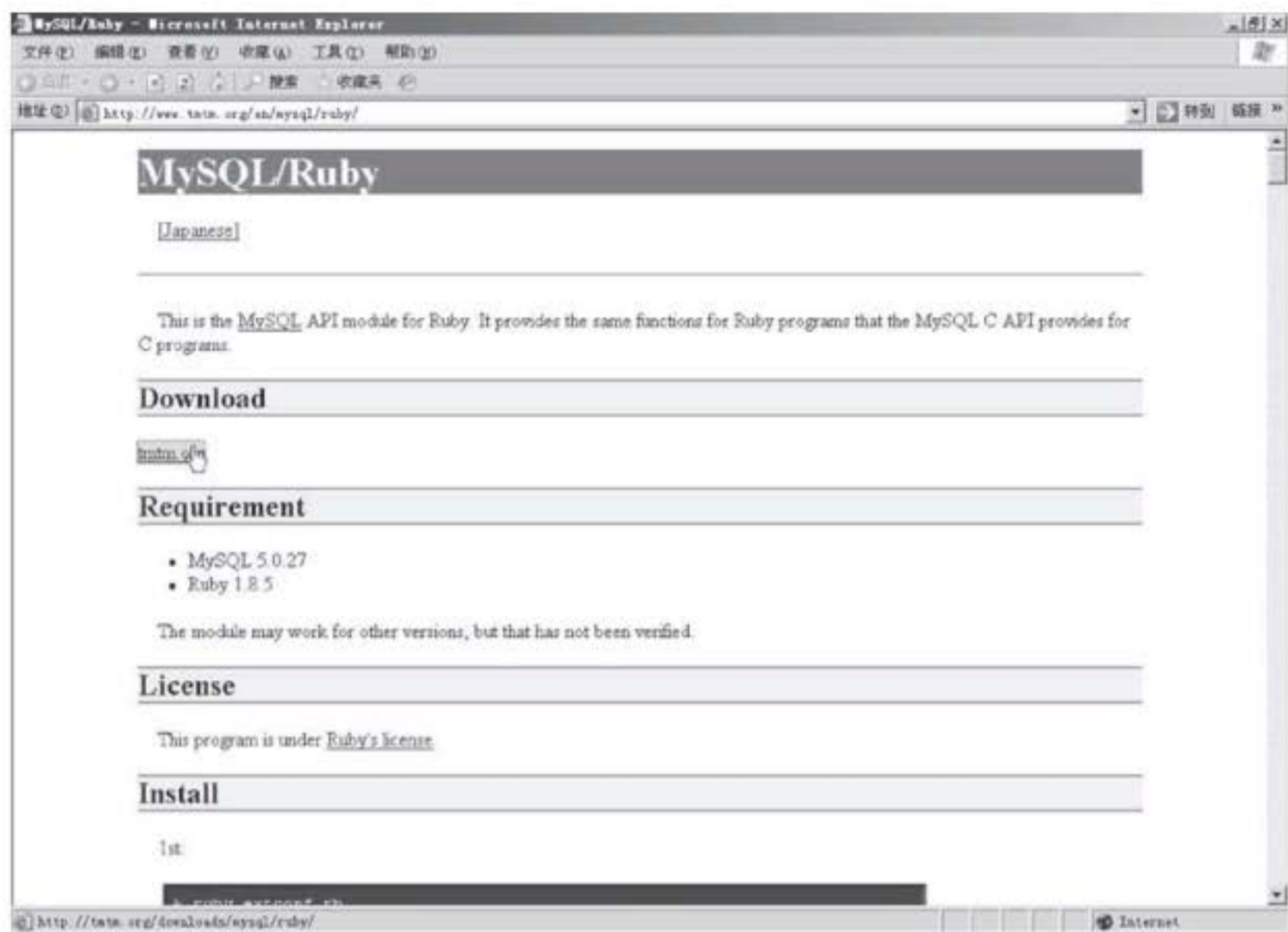


图 6-61 MySQL/Ruby 模块

(4) 测试安装

运行下面的程序,如果运行结果如图 6-62 所示,就证明安装成功了。

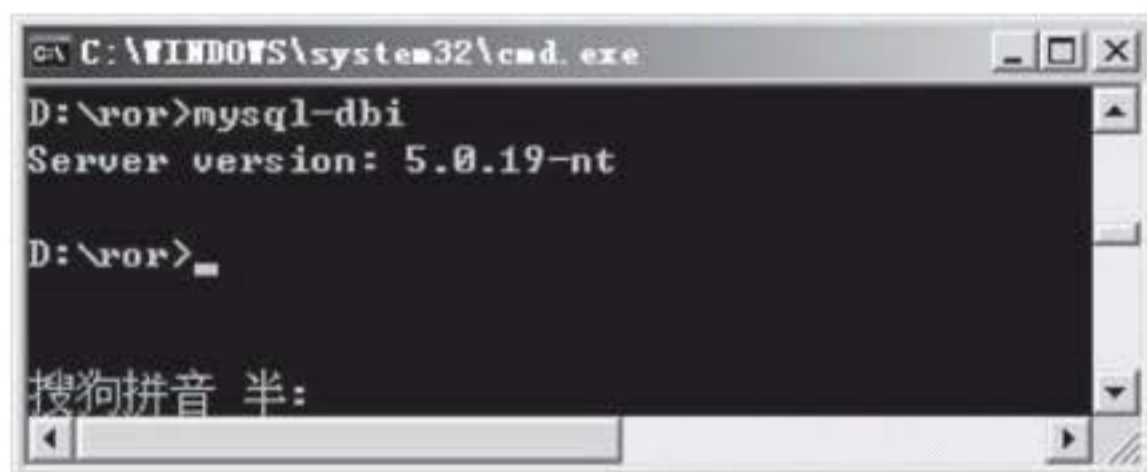


图 6-62 测试安装

案例名称:测试安装

程序名称:mysql-dbi.rb

```
require 'dbi'
begin
  # connect to the MySQL server
  dbh = DBI.connect("dbi:Mysql:my:localhost","root","tianen")
  # get server version string and display it
  row = dbh.select_one("SELECT VERSION()")
  puts "Server version: " + row[0]
```

```

rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code: # {e.err}"
  puts "Error message: # {e.errstr}"
ensure
  # disconnect from server
  dbh.disconnect if dbh
end

```

6.4.4 使用 DBI 访问 MySQL 数据库

使用 DBI 访问 MySQL 的方法和访问 Access 的方法相同。这里举一些例子,以便读者学习。

案例名称:执行查询

程序名称:ay.rb

```

require 'dbi'
dbh = DBI.connect("dbi:Mysql:my:localhost","root","tianen")
begin
  query = "select * from test"
  sth = dbh.execute(query)
  if sth.column_names.size == 0 then
    puts "no result set"
    printf "Number of rows affected: % d\n", sth.rows
  else
    puts "Query has a result set"
    rows = sth.fetch_all
    printf "Number of rows: % d\n", rows.size
    printf "Number of columns: % d\n", sth.column_names.size
    sth.column_info.each_with_index do |info, i|
      printf "- - - Column % d (% s) - - - \n", i, info.name
    end
  end
end
sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

程序运行结果如图 6-63 所示。

案例名称:批量操纵数据

程序名称:az.rb

```

require 'dbi'
h= Hash.new

```

```

h["lily"] = "coffee"
h["tom"] = "tea"
h["amy"] = "grape"
h["jim"] = "banana"
dbh = DBI.connect("dbi:Mysql:my:localhost", "root", "tianen")
begin
  sth = dbh.prepare("insert into test(name,hobby) values (?,?)")
  h.each_pair {|k,v| sth.execute(k,v)}
  dbh.commit
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

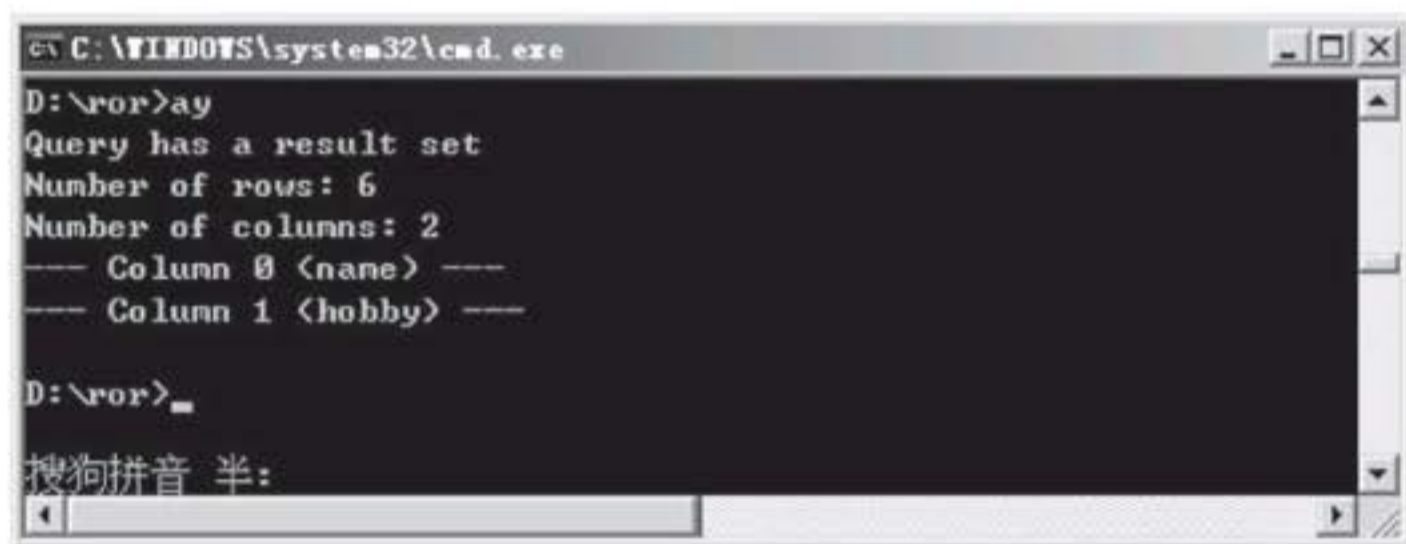


图 6-63 执行查询

程序运行后,向数据库表中增加了 4 条数据。结果如图 6-64 所示。

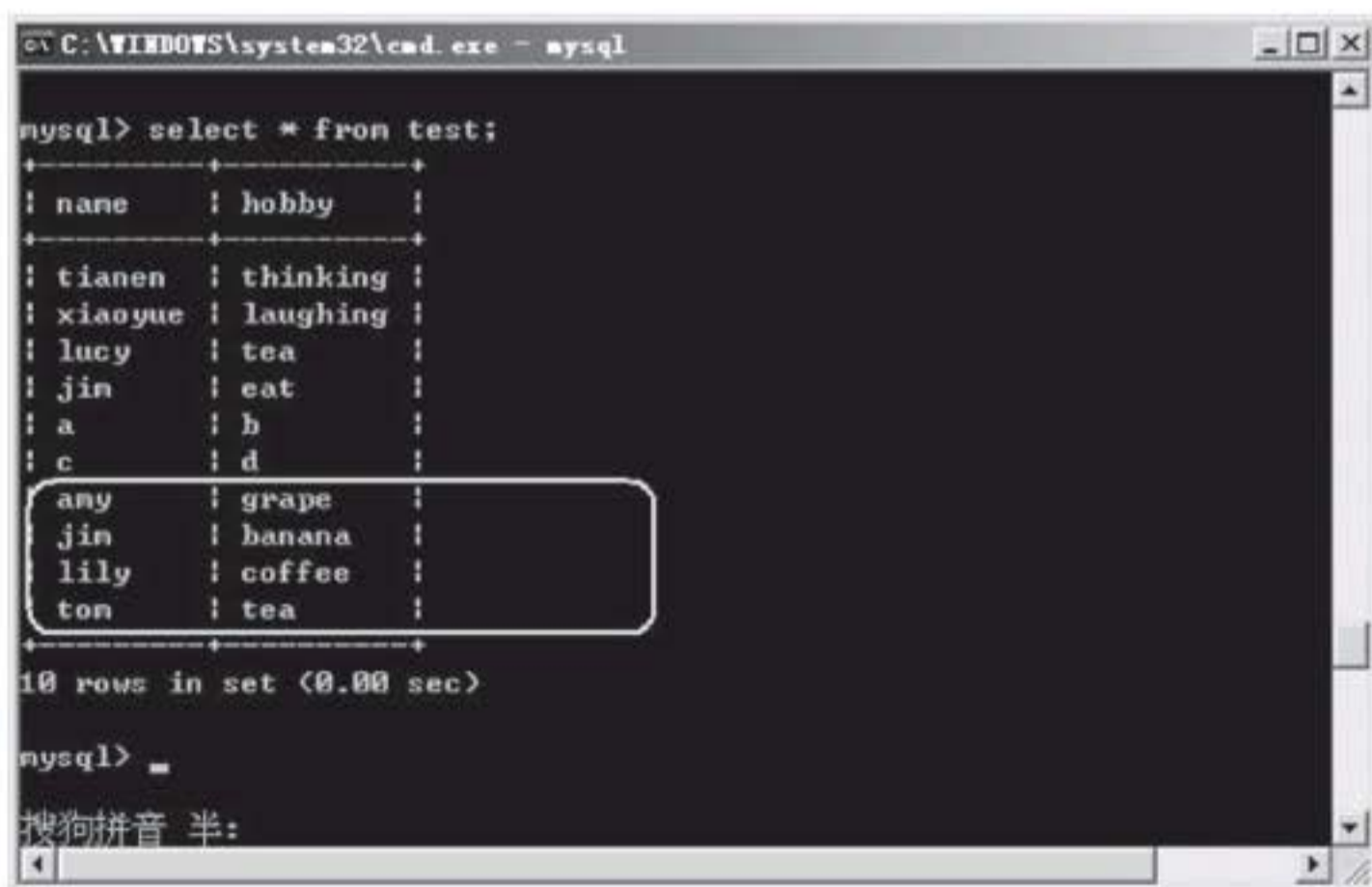


图 6-64 批量操纵数据

案例名称:执行查询

程序名称:ba.rb

```
require 'dbi'
dbh = DBI.connect("dbi:Mysql:my:localhost", "root", "tianen")
begin
  query = "select * from test"
  sth = dbh.execute(query)
  while row = sth.fetch do
    printf "Name: % s, Hobby: % s\n", row[0], row[1]
  end
  sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end
```

程序运行结果如图 6-65 所示。

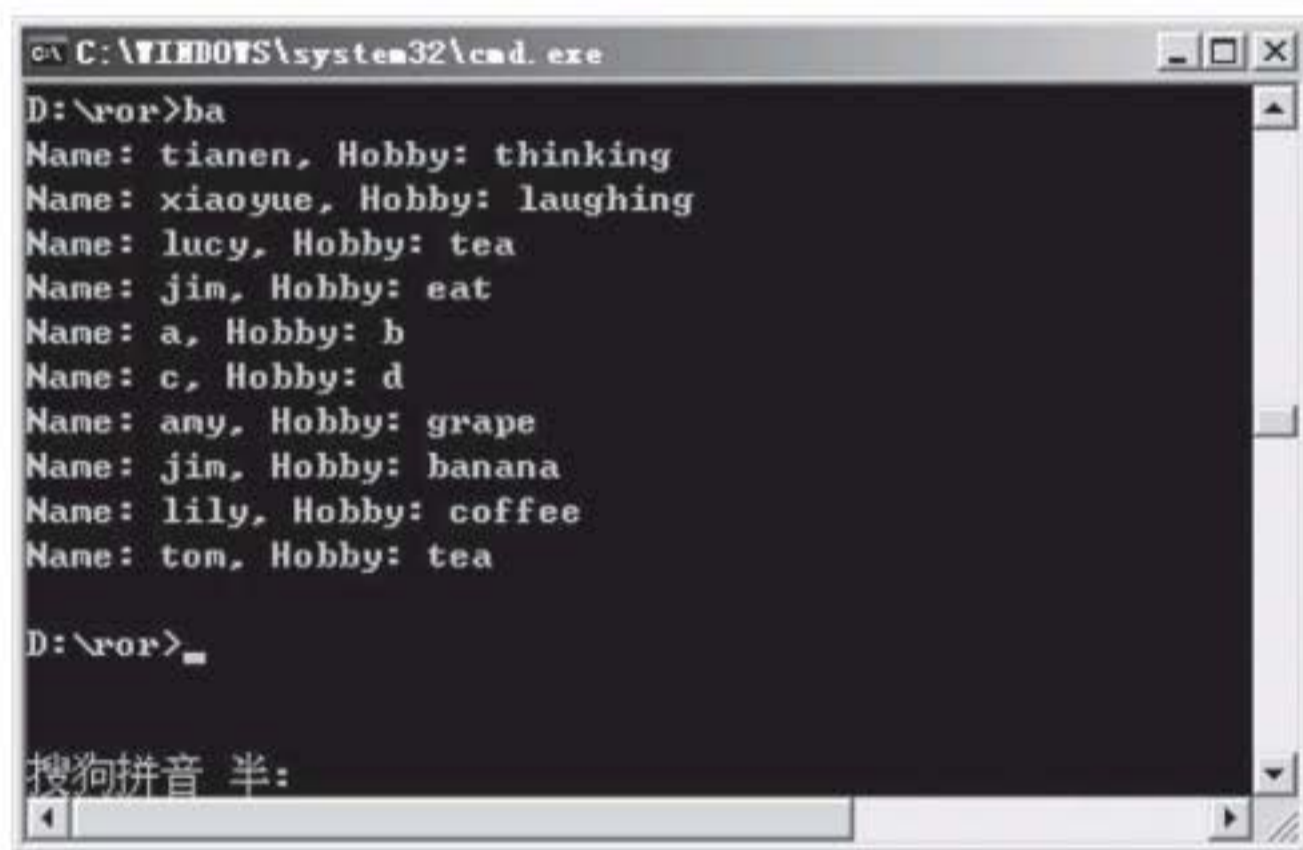


图 6-65 执行查询

案例名称:执行查询

程序名称:bb.rb

```
require 'dbi'
dbh = DBI.connect("dbi:Mysql:my:localhost", "root", "tianen")
begin
  query = "select * from test"
  sth = dbh.execute(query)
  sth.fetch do |row|
```

```

    printf "Name: % s, Hobby: % s\n", row[0], row[1]
  end
  sth.finish
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

程序运行结果如图 6-66 所示。

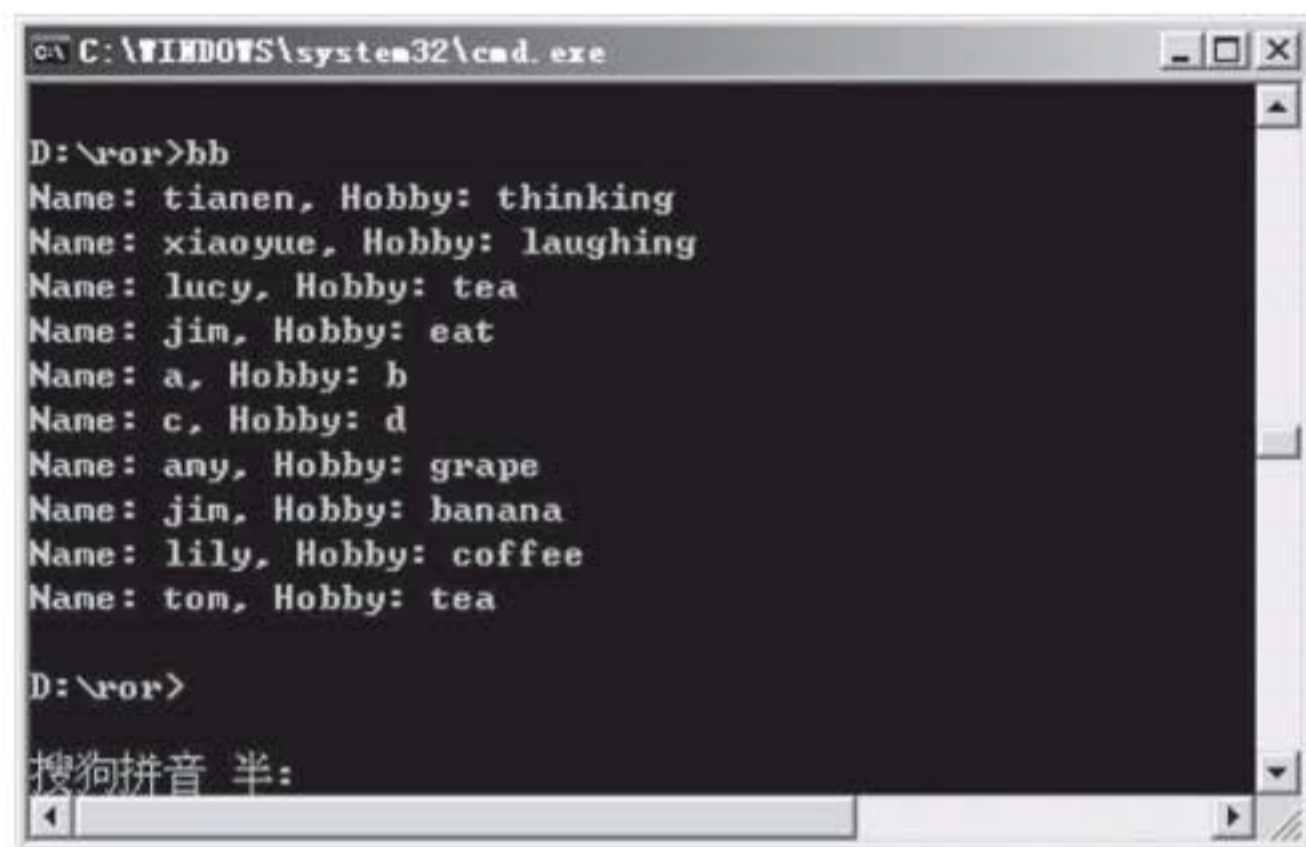


图 6-66 执行查询

案例名称:执行查询

程序名称:bc.rb

```

require 'dbi'
dbh = DBI.connect("dbi:Mysql:my:localhost", "root", "tianen")
begin
  query = "select * from test"
  rows = dbh.select_all(query)
  rows.each{|e|
    printf "% s, % s\n", e[0], e[1]
  }
rescue
  p "error!"
ensure
  dbh.disconnect if dbh
end

```

程序运行结果如图 6-67 所示。



图 6-67 执行查询

小 结

本章详细介绍了使用 Ruby 进行数据库访问的方法,读者需要掌握这些数据访问组件的安装方法和常用的 API,这样就可以自由地操作数据库。

思考和练习

1. 练习安装 MySQL 模块。
2. 基于 Access 数据库建立一个 DOS 界面的通讯录。

第 7 章 桌面应用和 Web 开发

本章要点

本章介绍使用 Ruby 开发桌面应用和 Web 应用的相关知识。需要重点掌握的是 eRuby 的开发方法。

7.1 Ruby 的桌面开发

使用 Ruby 是可以开发桌面应用的,这与 PHP,Perl 是类似的。Ruby 通过 Tcl/Tk, GTK,OpenGL 等扩展可以很方便地进行桌面应用程序开发。Ruby 的发行版是不含 TK 库的,读者需要自行下载相应的扩展库。

7.2 Ruby 的 Web 开发方法

Ruby 可以进行 Web 的开发,但最著名的是 Rails。然而本书不介绍 Rails,只讲 Ruby。那么,不用 Rails 框架,单纯使用 Ruby 可以进行 Web 开发吗?

当然是可以的。

Ruby 除了可以采用 Rails 框架做 Web 开发之外,还可以使用类似 ASP 和 PHP 的代码嵌入式方法来做 Web 开发,而且这种开发方式对初学者而言更容易接受。

本章笔者将较为详细地介绍 Ruby 的 Web 开发方法,作为 Ruby 基础知识的终结,也作为 Rails 学习的开始。

7.2.1 CGI 类

要使用 Ruby 输出 HTML 内容,可以用下面的方式。

```
print "HTTP/1.0 200 OK\r\n"
print "Content-type: text/html\r\n\r\n"
print "< html> < body> Hello Tianen's World! < /body> < /html> \r\n"
```

这是最原始的方式。

如果使用 CGI 类,就可以操纵表单(form)、cookies 和环境变量及 session 等。这个方式比最基本的方式有所进步。最简单的用法如例子“使用 CGI 类”所示。

案例名称:使用 CGI 类

程序名称:a.rb

```
require 'cgi'
puts "tianen"
```

程序运行结果如图 7-1 所示。



图 7-1 使用 CGI 类

CGI 的执行速度比较慢, mod_ruby 在 apache 中嵌入了 Ruby 解释器, 能使 Ruby CGI 更快地执行。软件可以从 http://www.modruby.net/en/index.rbx/mod_ruby/download.html 下载。如图 7-2 所示。

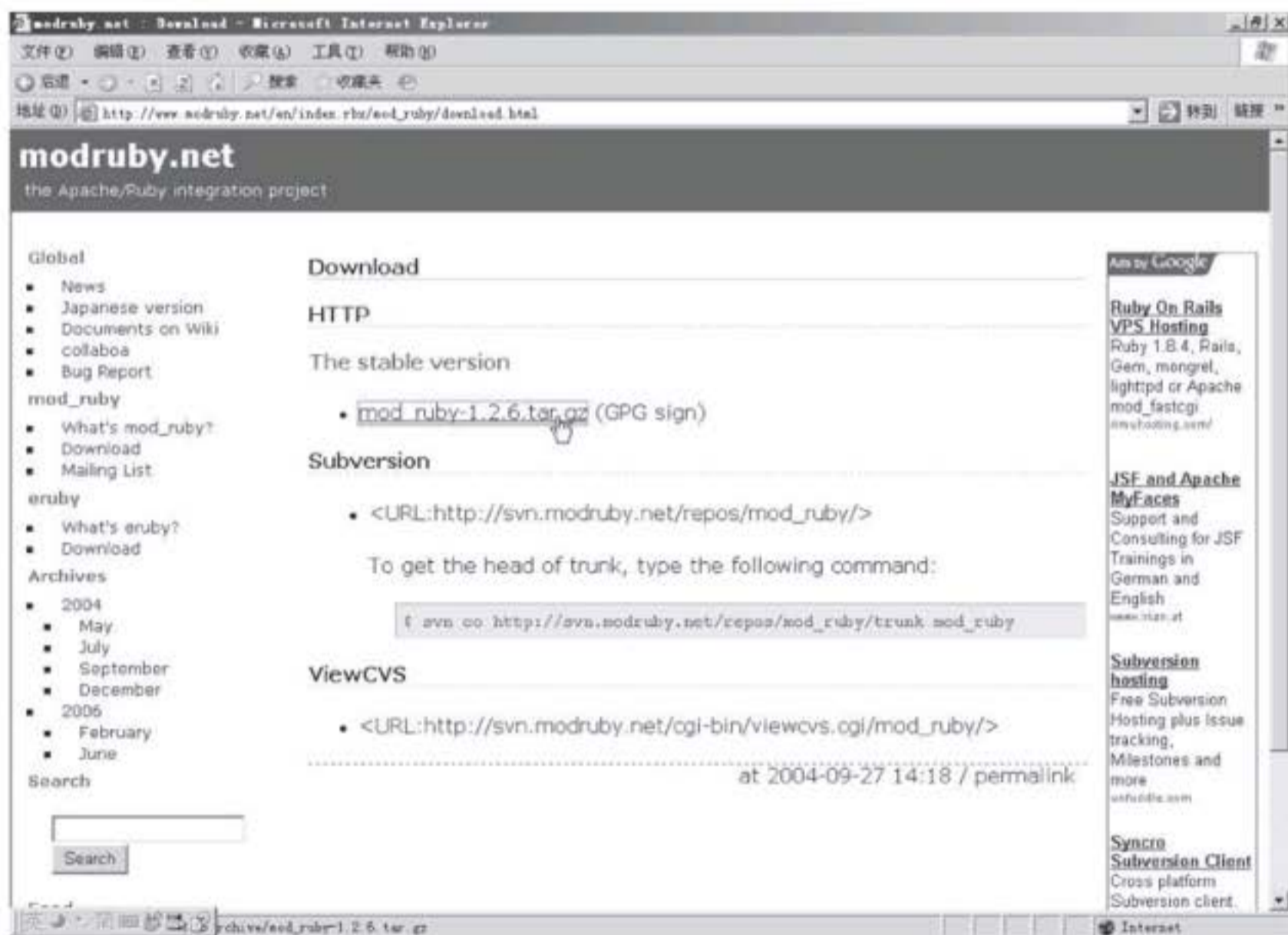


图 7-2 下载 mod_ruby

下载结果如图 7-3 所示。



图 7-3 mod_ruby

7.2.2 eRuby 概述

eRuby 是一种技术。就是把 Ruby 代码嵌入到页面中,从而构建 Web 应用。如果读者对 Java 有些了解的话,那么可以这样类比:

Ruby 类似 Java,Ruby CGI 类似 Servlet,eRuby 类似 JSP,Rails 类似 Struts。

实现嵌入式 Ruby 脚本执行的方法有多种,笔者介绍应用最广的一种,名为“eruby”。要注意:eRuby 是一种技术,而 eruby 是一种实现。

(1) 下载 eruby

要安装 eruby,首先要下载相关的软件包。读者可以在 <http://www.modruby.net/en/index.rbx/eruby/download.html> 下载。如图 7-4 所示。

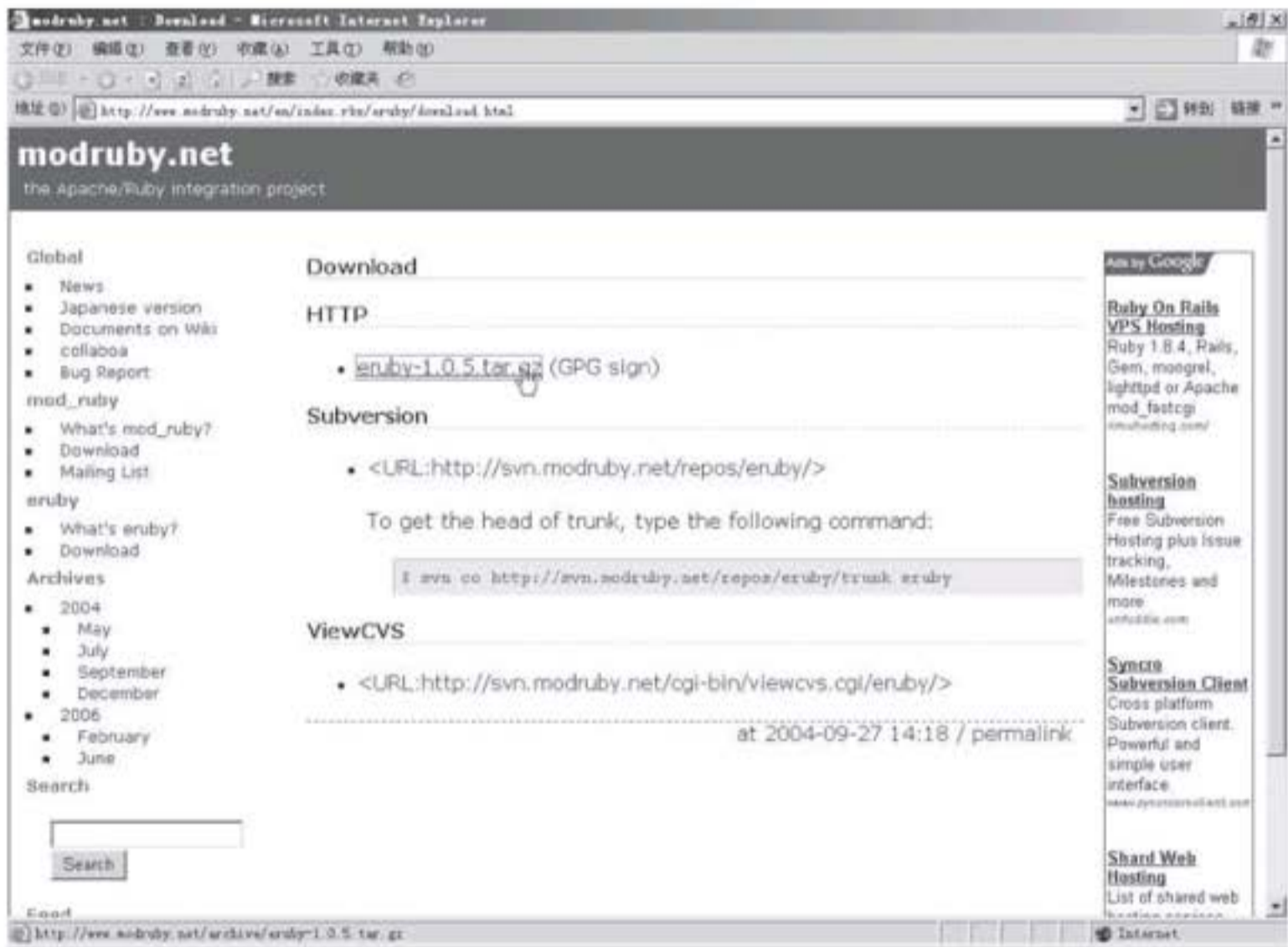


图 7-4 下载 eruby

下载结果如图 7-5 所示。

要使用这个组件,需要重新编译,不方便。

读者可以从 <http://www.unbe.cn/blog/wp-content/uploads/2007/06/eruby-105-i386-mingw32-18tar.gz> 下载已经编译好的组件。如图 7-6 所示。



图 7-5 eruby



图 7-6 eruby

将其解压缩,在 local/bin 目录下可以找到 eruby.exe 文件,这就是所需要的文件。

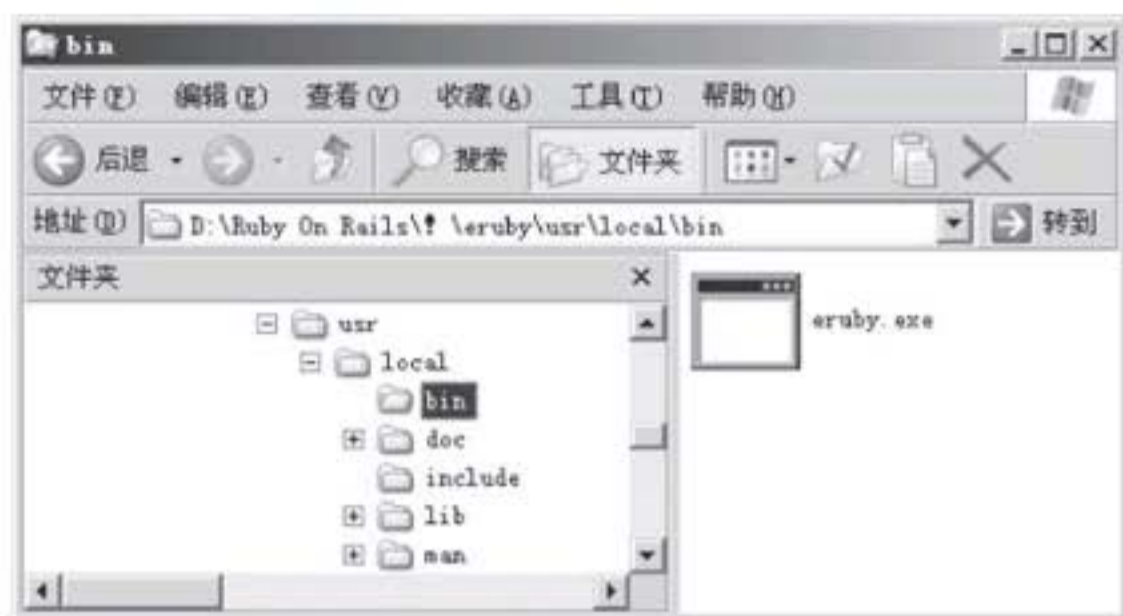


图 7-7 eruby

(2) 使用 IIS 配合 eruby

将 eruby.exe 文件复制到某个目录下,比如放到 D:\ruby\bin 目录下。然后配置 IIS,使扩展名为“.erb”和“.rhtml”的文件被 eruby.exe 处理。打开 IIS 属性窗口,如图 7-8 所示。



图 7-8 IIS 属性窗口

单击“配置”按钮,将弹出应用程序配置窗口,如图 7-9 所示。

单击“添加”按钮,弹出扩展名映射窗口,如图 7-10 所示。在可执行文件文本框中添加 eruby.exe 的路径,在扩展名中添加“.rhtml”,然后单击“确定”按钮,这样就完成了 rhtml 扩展名的映射。

同理,将扩展名为“.erb”的文件映射给 eruby.exe 处理,如图 7-11 所示。

当然,也可以指定 Ruby 程序为其他扩展名。只要和 eruby.exe 映射好就不会出错。



图 7-9 应用程序配置窗口



图 7-10 扩展名映射



图 7-11 扩展名映射

(3) 建立虚拟目录

在 IIS 中建立一个名为“eruby”的虚拟目录。然后在目录中添加如下文件。

案例名称:测试 eruby

程序名称:a.erb

```
< %  
(1..10).each do |i|  
  print i.to_s + "< br> "  
end  
% >
```

运行结果如图 7-12 所示。

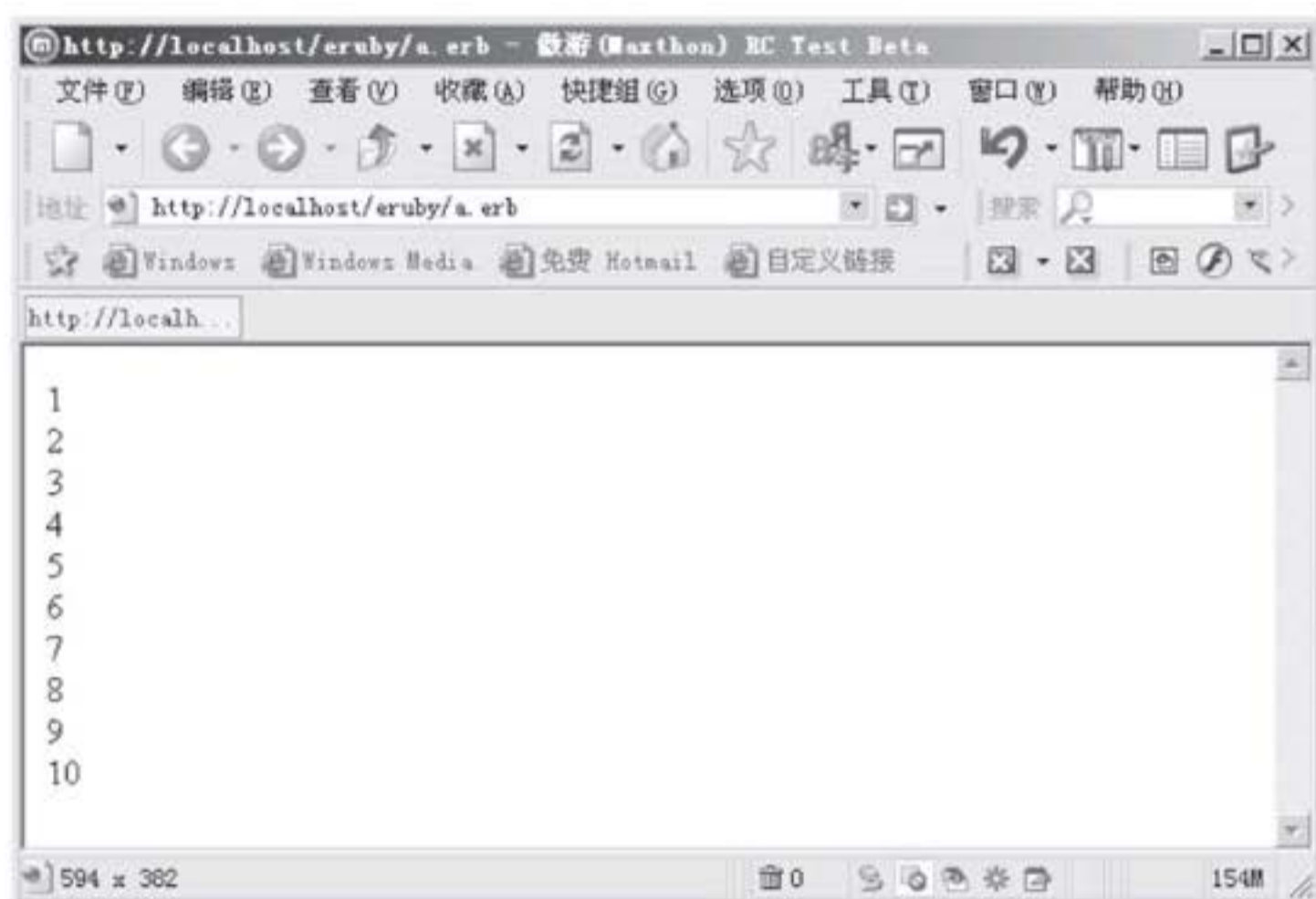


图 7-12 测试 eruby

可以看到,eruby 已经成功运行了。

类似地,也可以将 eruby 和其他的 web 服务器相关联,如:Apache。在此不赘述。

7.2.3 eruby 的基本使用

配置好了 eruby 之后,就可以用它来编写程序了。eruby 中,所有的 Ruby 代码都嵌入 html 之中,被`<%`和`%>`包围。

代码主要有如下三种形式:

(1) 执行 Ruby 代码

`<% Ruby 代码 %>` 将定界符内的代码转换为它的结果输出。

(2) 表达式

`<%=表达式 %>` 输出这个表达式的值到 HTML 中。

(3) 注 释

`<%# ruby code %>` 代码注释,这些内容将被忽略。

案例名称:eruby 的基本使用

程序名称:b. erb

```
< % # 测试% >
i love you< p>
< %
    print 'i love you< p> '
% >
China! < p>
< % = 'China! '% >
< %
    a= 1
    b= 2
% >
< %
    p a+ b
% >
```

运行结果如图 7-13 所示。

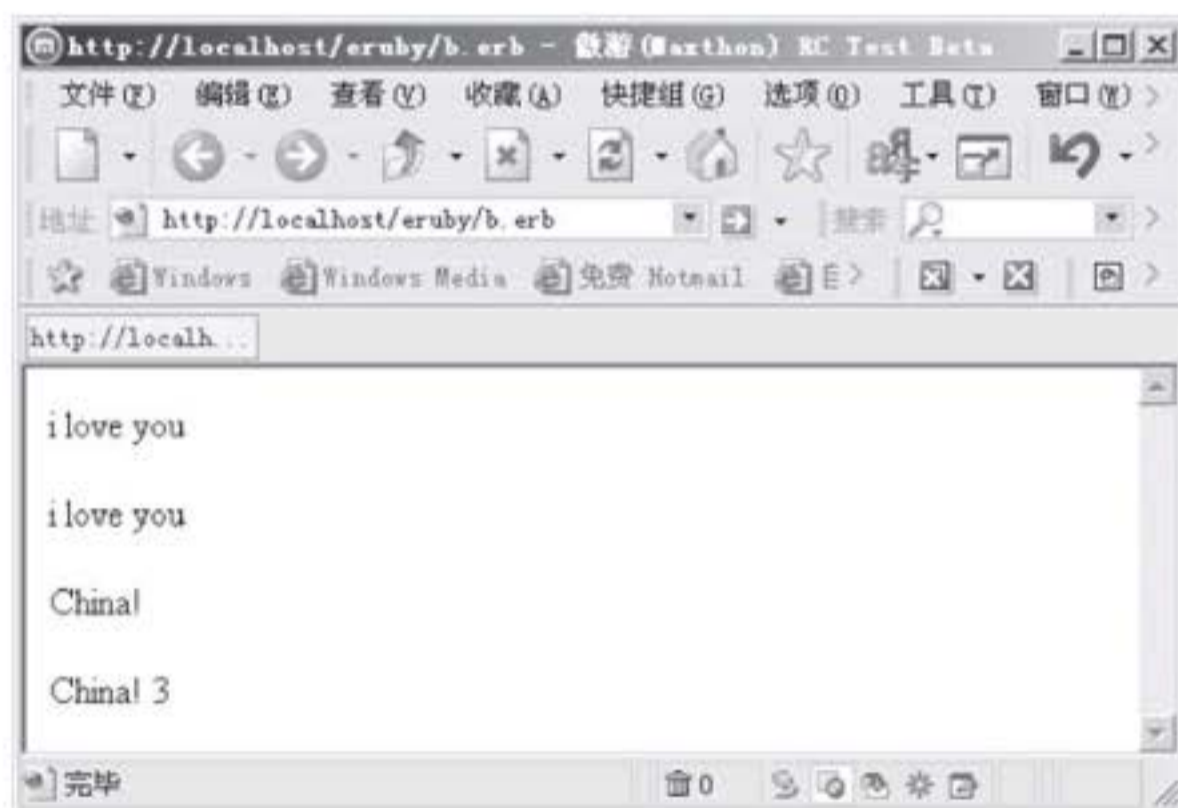


图 7-13 eruby 的基本使用

7.2.4 文件包含

使用 `ERuby.import(" ")` 语句可以实现页面的包含。将某一页面包含到本程序中运行。

案例名称:文件包含

程序名称:c. erb

```
< %
    ERuby.import("a.erb")
...

```

```

print "< hr/> "
ERuby.import("b.erb")
% >
< p>
Now is : < % = Time.now% >

```

运行结果如图 7-14 所示。

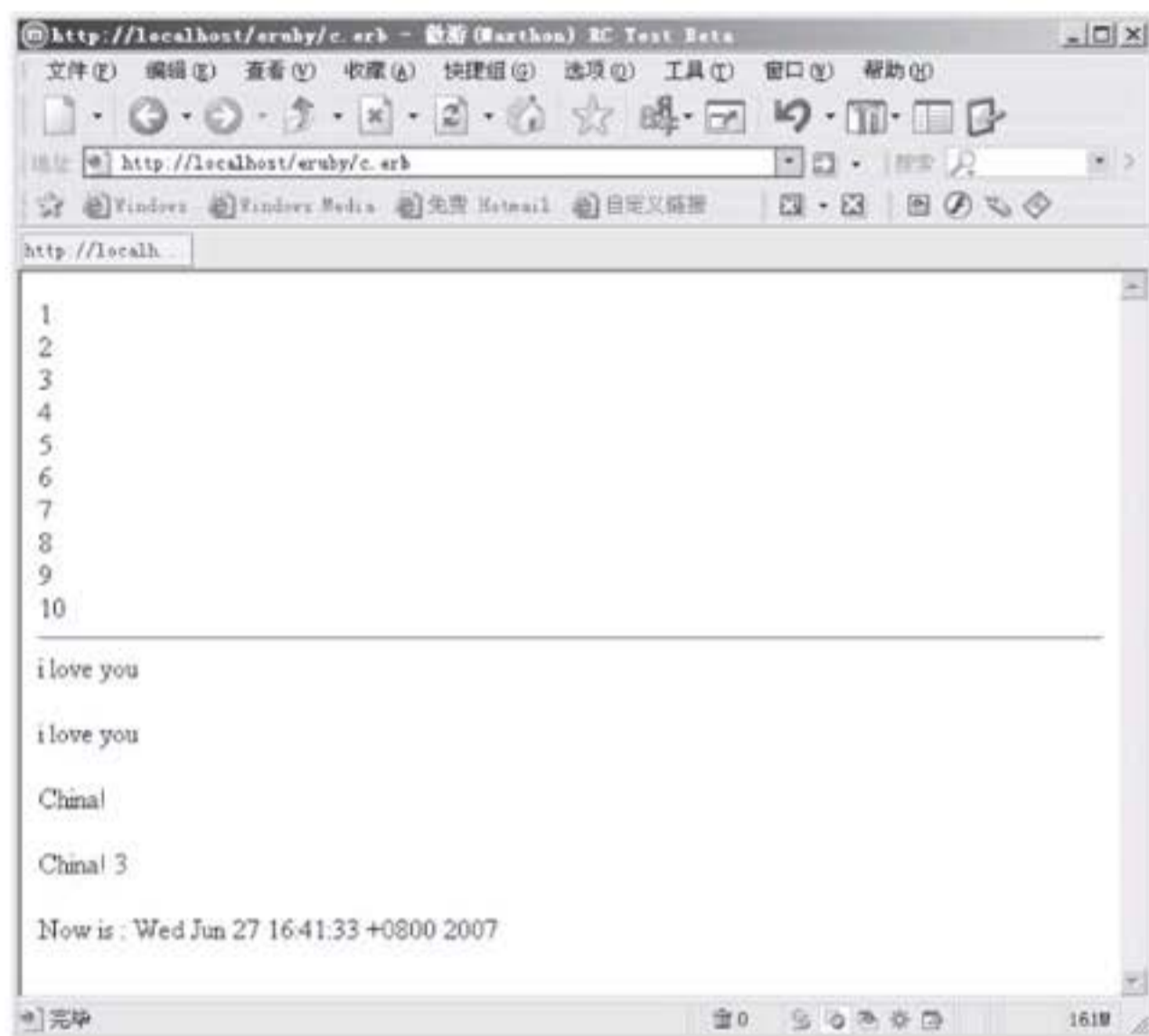


图 7-14 文件包含

7.2.5 中文显示

使用 eruby 输出英文是没有问题的,但是输出中文就会存在乱码问题。如下:

案例名称:中文显示问题

程序名称:d.erb

```

< %
print "hello! < p> "
print "好!"
% >

```

运行结果如图 7-15 所示。

如果要正常显示,需要在浏览器中设置网页编码,如图 7-16 所示。

设置网页编码之后,页面显示结果如图 7-17 所示。

当然,制作者不能期待浏览者去设置网页编码,而应该使得浏览者看到网页的时候就没有乱码。方法如下:

案例名称:中文显示问题

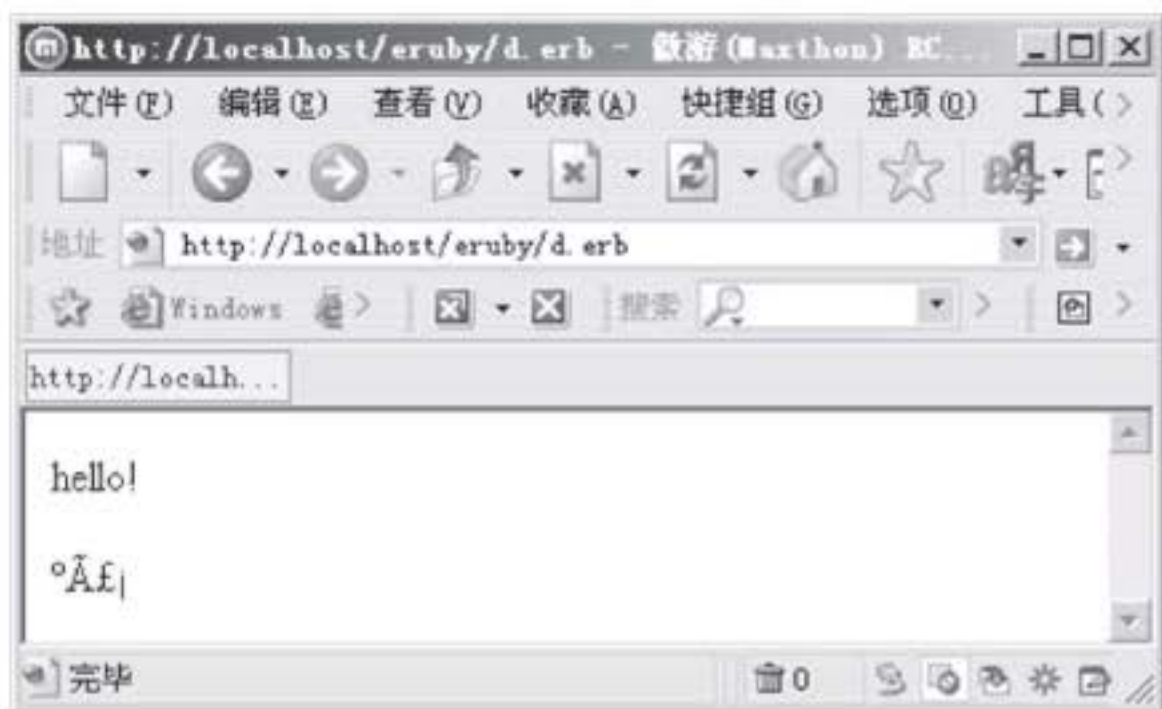


图 7-15 中文乱码

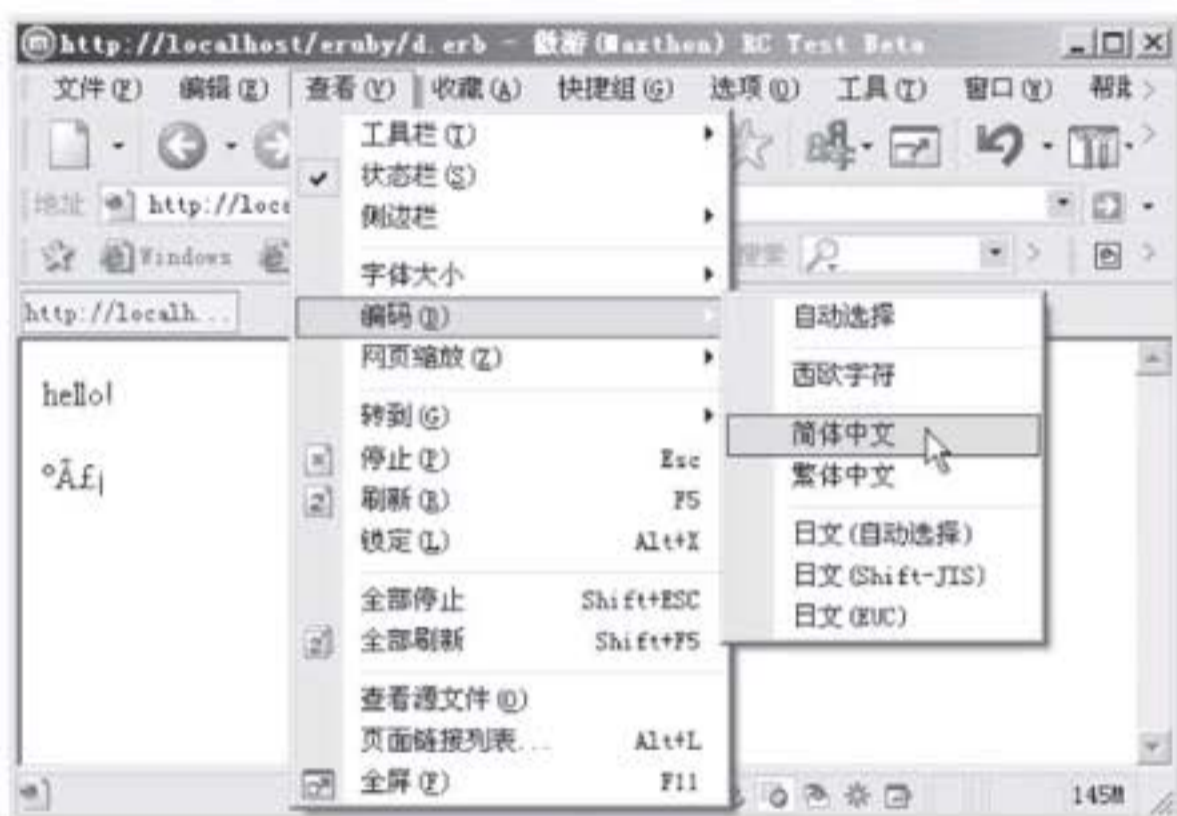


图 7-16 设置网页编码



图 7-17 设置网页编码后正常显示

程序名称:e.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
```

```
< %
print "hello! < p> "
print "好!"
% >
```

运行结果如图 7-18 所示。

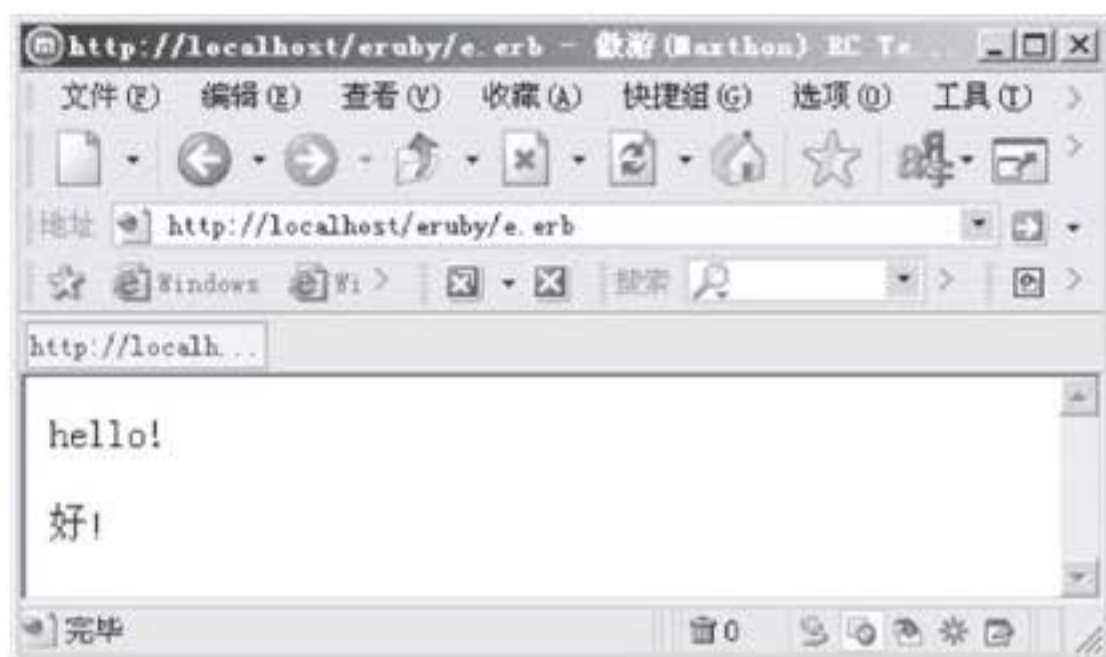


图 7-18 中文显示问题解决

7.2.6 参数的传递和接收

传递参数可以通过浏览器地址栏或者表单,常用的有 GET 和 POST 两种。要接收参数,需使用 CGI 对象,其创建方法如下:

```
require 'cgi'
cgi = CGI.new
```

下面的例子演示了使用地址栏和表单进行数据传递的方法。

案例名称:参数的传递和接收

程序名称:f.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'cgi'
cgi = CGI.new
s = cgi['test'].to_s
print s
% >
```

运行结果如图 7-19 和图 7-20 所示。

程序名称:g.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'cgi'
```

```

cgi = CGI.new
a = cgi['a'].to_s
b = cgi['b'].to_s
print "a= " + a + "<br>" + "b= " + b
% >

```

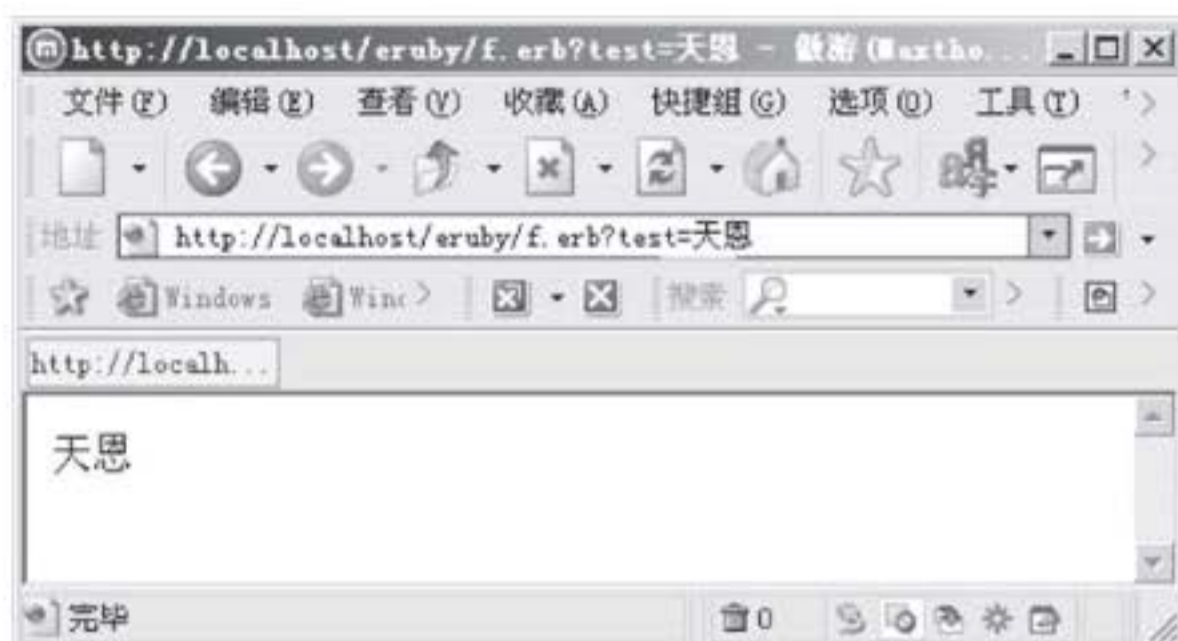


图 7-19 参数的传递和接收

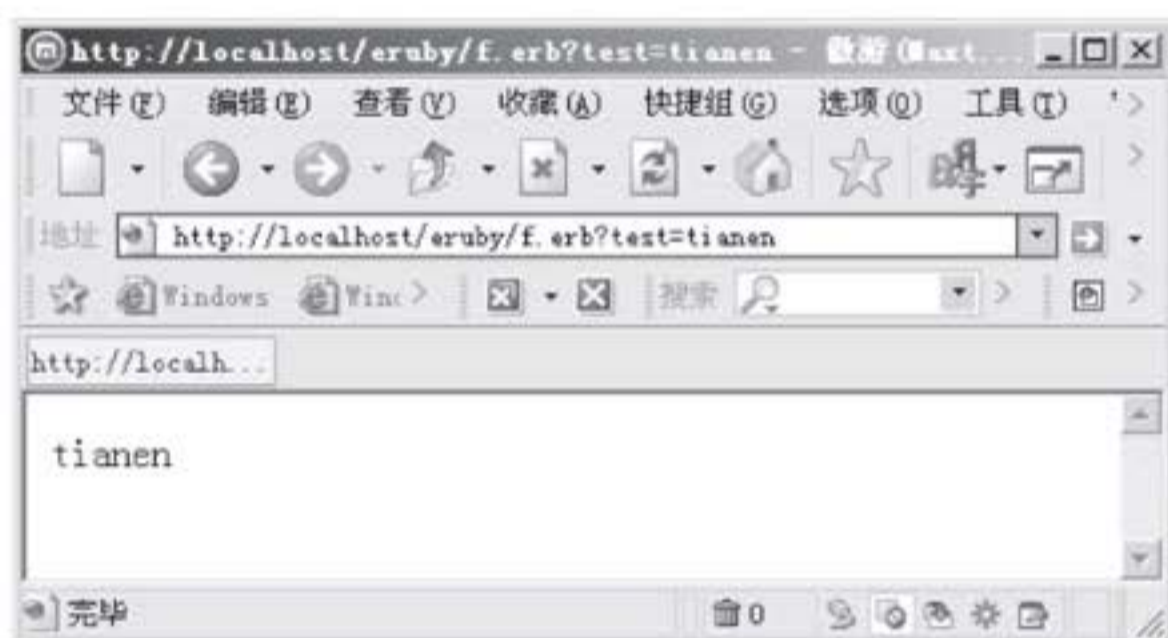


图 7-20 参数的传递和接收

运行结果如图 7-21 所示。

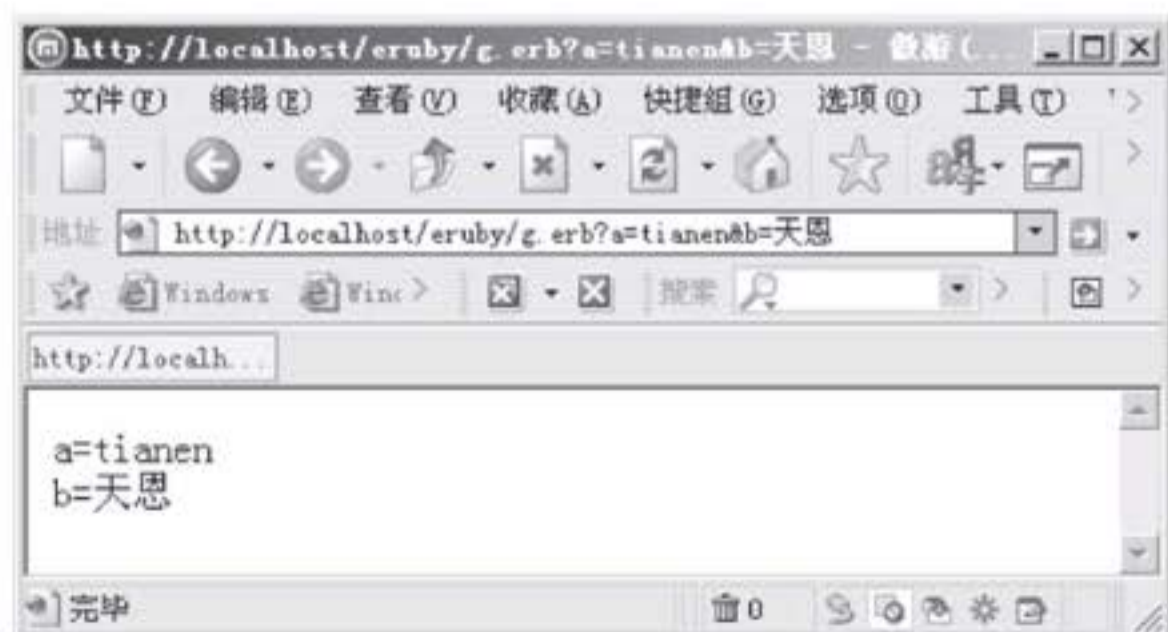


图 7-21 参数的传递和接收

程序名称:h. htm

```
< form action= h.erb method= post>
< input type= text name= "a">
< textarea name= "b">
< /textarea>
< input type= submit value= "do">
< /form>
```

程序名称:h. erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content- type: text/html
< %
require 'cgi'
cgi = CGI.new
a = cgi['a'].to_s
b = cgi['b'].to_s
print "a= " + a + "< br> " + "b= " + b
% >
```

运行结果如图 7-22 和图 7-23 所示。

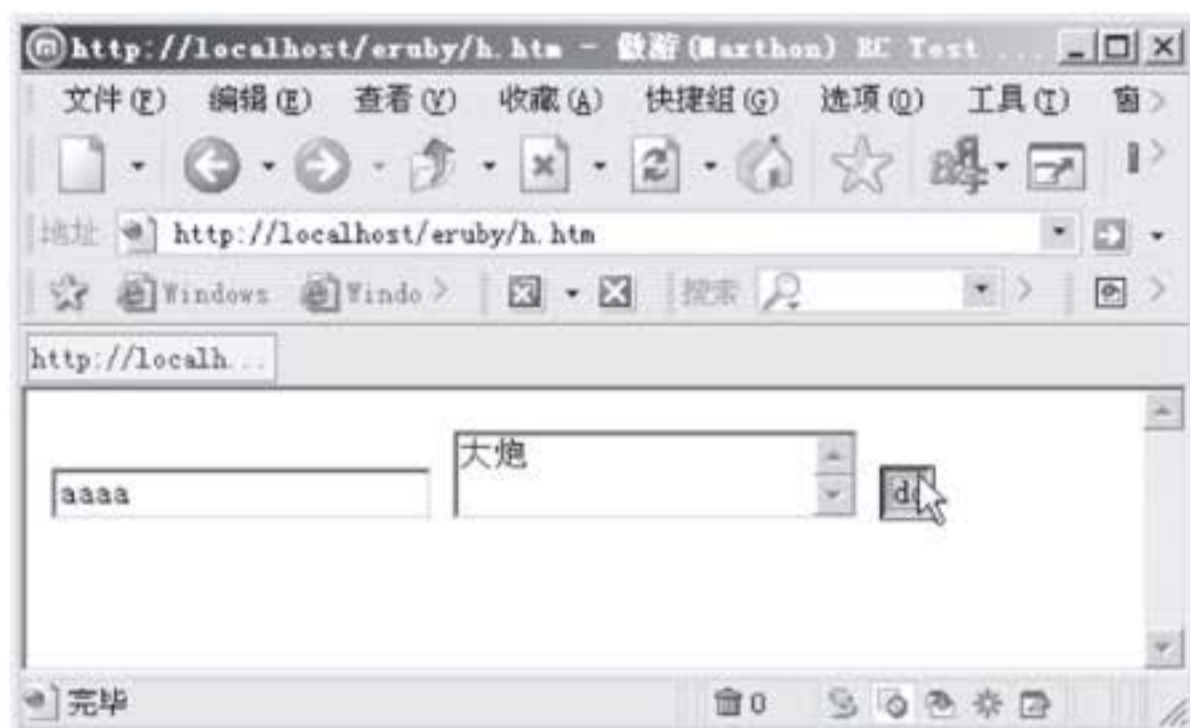


图 7-22 参数的传递和接收

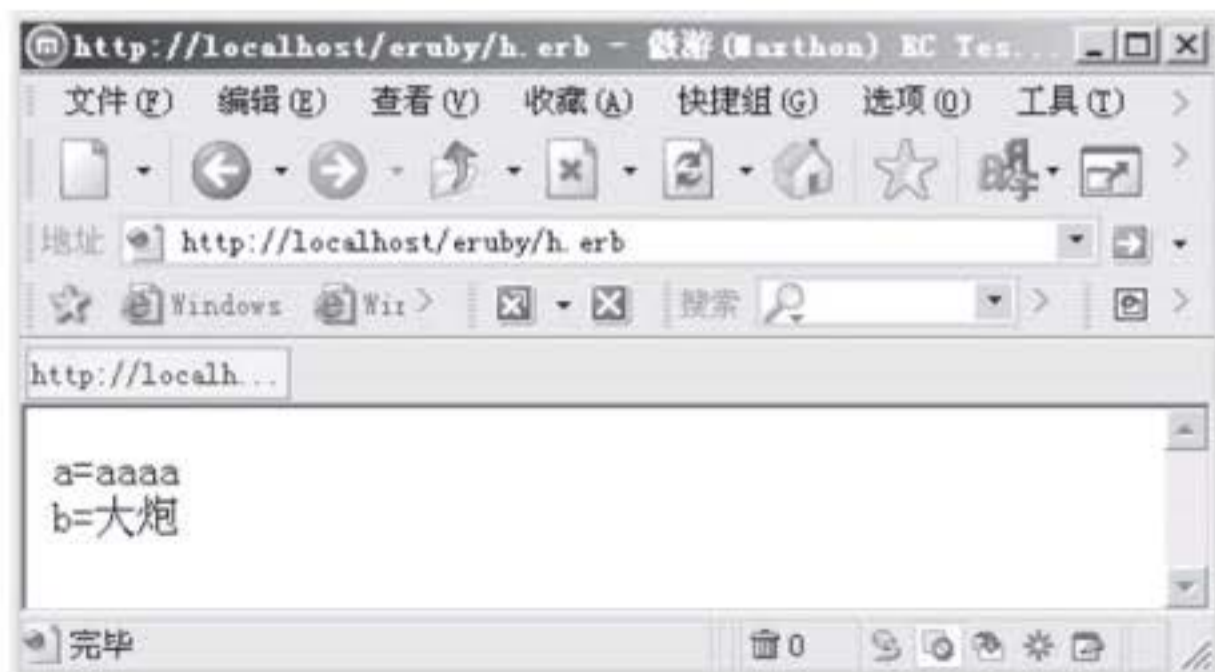


图 7-23 参数的传递和接收

7.3 详解表单处理

7.3.1 表单的提交

表单的提交有 GET 和 POST 两种方式。

如：`<form action="tianen.php" method="post 或 get" enctype='text/plain(默认文本数据)'>`

GET 方式是表单的默认提交方式,也就是如果不在上面的 method 里声明,那么表单将会采用 GET 方式提交到服务器,这时提交的数据信息将会显示在浏览器栏里面。GET 方式提交数据有弊端。首先,这种方法依赖于 Web 服务器的操作系统和软件,发送数据有限制,许多系统都限制在 256 个字符。其次,每个 get 请求都会保存在 Web 服务器日志里,如果使用的是共享服务器,那么其他人就可以解密用户使用 GET 方式提交的数据了。

POST 方式没有上面讲的弊端,提交的数据信息不会显示在浏览器栏里面。如果提交的是文本内容(text/plain),那么数据量没有什么限制。

使用 GET 方式提交的信息将会被存储在 `$_GET[]` 数组中。

使用 POST 方式提交的信息将会被存储在 `$_POST[]` 数组中。

使用 GET 或 POST 方式提交的信息都将会被存储在 `$_REQUEST[]` 数组中。

注意:读者千万不要以为使用 POST 就比 GET 安全。许多软件可以对网站实施模拟 POST 的攻击,看起来某些数据是用 POST 方式提交来的,实际上未必如此。

7.3.2 表单的接收

通常一个页面里只有一个 form,而且表单域的名字各不相同。这时的表单提交是很常规的,非常容易处理。

如果一个页面含有多个 form,而且这些 form 名字相同,甚至含有多个同名的表单域。提交按钮也有很多个,这时表单该如何处理呢?

笔者自创了一种解决方案,在本书后面的实例中用到了这种表单技巧。这种技巧的灵活使用可以减少工程项目的页面总数,而且代码模块化程度非常高,便于维护。

(1) 一个页面含有多个相同表单的情况

一个页面含有多个相同表单的时候,无法同时提交多个,一次只能提交一个。服务器端程序只接收被提交的表单。

下面的案例演示了 form_a.htm 向 form_a.erb 提交表单的情况。

案例名称:一个页面含有多个相同表单的情况

程序名称:form_a.htm

```
< h3> 多个表单向同一个 eruby 程序提交时的接收情况:< /h3>
```

这些表单完全相同。为了方便测试,我在 value 里面预设了值。

```
< FORM METHOD= POST ACTION= "form_a.erb">
```

```

< INPUT TYPE= "text" NAME= "book" value= "book- a">
< INPUT TYPE= "submit">
< INPUT TYPE= "reset">
< /FORM>
< FORM METHOD= POST ACTION= "forma.erb">
< INPUT TYPE= "text" NAME= "book" value= "book- b">
< INPUT TYPE= "submit">
< INPUT TYPE= "reset">
< /FORM>
< FORM METHOD= POST ACTION= "forma.erb">
< INPUT TYPE= "text" NAME= "book" value= "book- c">
< INPUT TYPE= "submit">
< INPUT TYPE= "reset">
< /FORM>

```

显示的结果如图 7-24 所示。

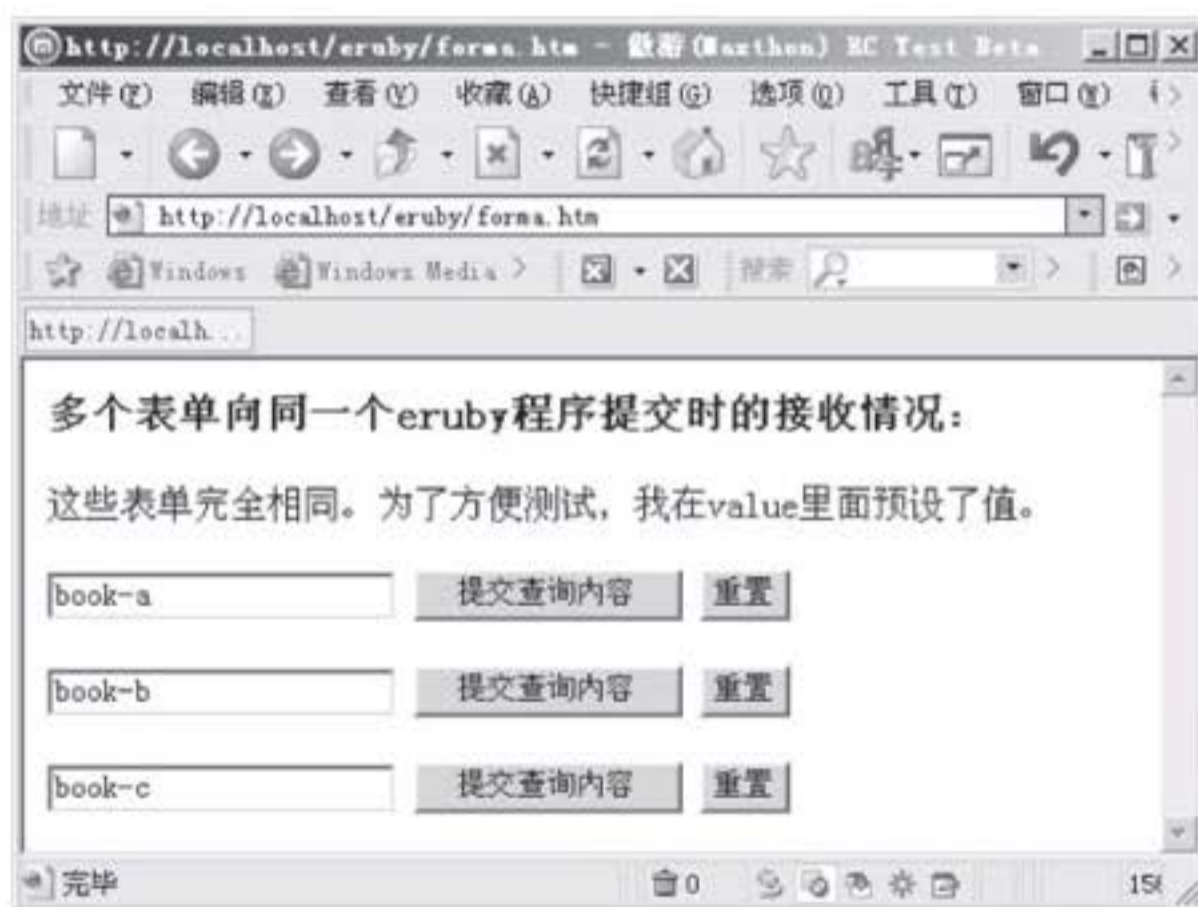


图 7-24 一个页面含有多个相同表单的情况

程序名称:forma.erb

```

# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'cgi'
cgi = CGI.new
book = cgi['book'].to_s.strip
print "book:" + book + "< p> "
% >

```

当在 form-a.php 中分别提交 3 个表单的时候,显示的结果如图 7-25,图 7-26,图 7-27 所示。



图 7-25 提交第一个表单的结果



图 7-26 提交第二个表单的结果



图 7-27 提交第三个表单的结果

(2) 一个表单含有多个提交按钮的情况

一个表单含有多个提交按钮的时候,单击任何一个按钮都将提交表单。

案例名称:一个表单含有多个提交按钮的情况

程序名称:formb.htm

< h3> 一个表单含有多个提交按钮向一个 erb 程序提交时的接收情况:< /h3>

为了方便测试,我在 value 里面预设了值。

< FORM METHOD= POST ACTION= "formb.erb">

```
< INPUT TYPE= "text" NAME= "book" value= "a lot of books">
< INPUT TYPE= "submit" name= "sa" value= "sa">
< INPUT TYPE= "submit" name= "sb" value= "sb">
< INPUT TYPE= "reset">
< /FORM>
```

显示的结果如图 7-28 所示。

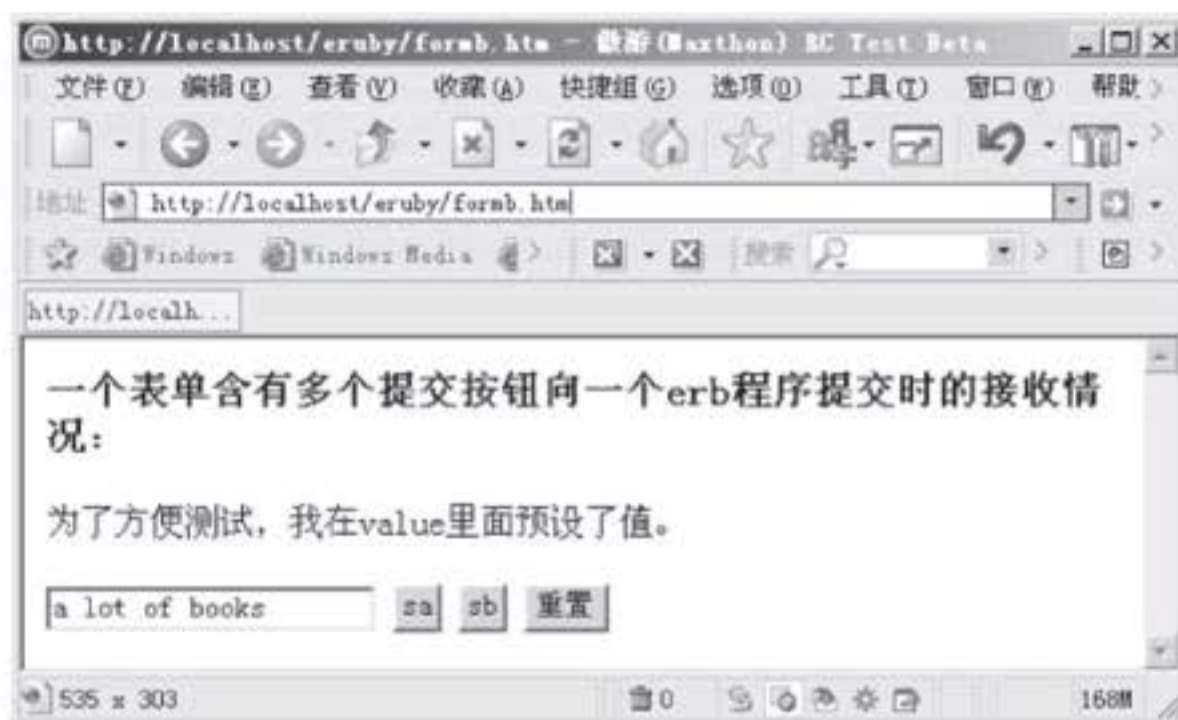


图 7-28 一个表单含有多个提交按钮的情况

程序名称:formb. erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'cgi'
cgi = CGI.new
if ! cgi['sa'].empty? then
  print "数据由 sa 提交过来:< br> "
  print cgi['book']
end
if ! cgi['sb'].empty? then
  print "数据由 sb 提交过来:< br> "
  print cgi['book']
end
% >
```

在 formb. htm 中单击不同按钮提交表单的时候,在 formb. erb 中对按钮进行判断,从而处理表单数据。单击不同提交按钮的效果如图 7-29,图 7-30 所示。

这个例子在工程中是很有现实意义的,根据提交过来的按钮的属性来决定如何处理数据。比如:一个按钮的名字为显示,另一个名字为添加。那么,就可以判断是哪一个按钮提交了表单,如果是“显示”,就把表单内容显示出来,如果是“添加”,就把表单内容写入数据库。

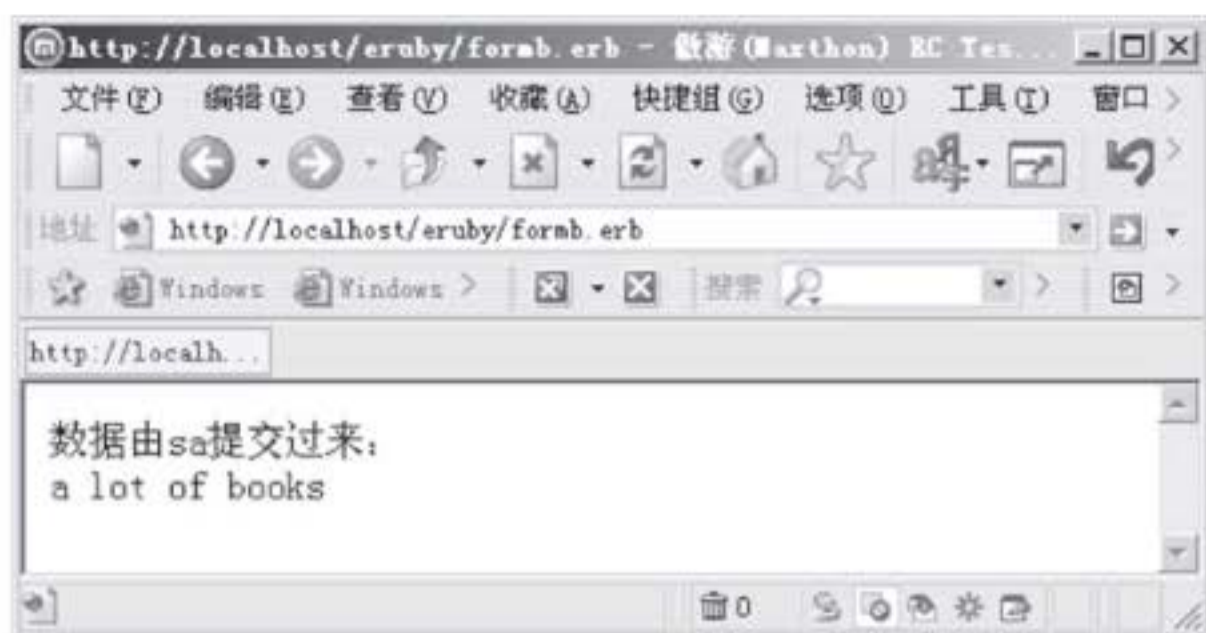


图 7-29 单击 sa 按钮

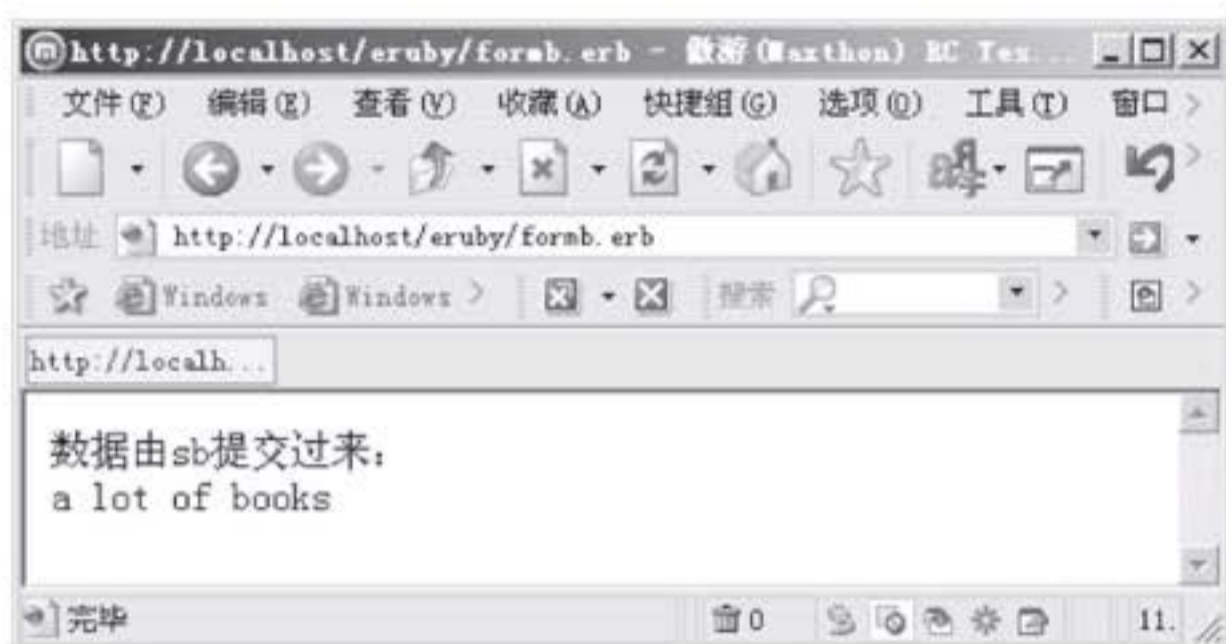


图 7-30 单击 sb 按钮

7.4 文件操作

Ruby 可以做的,eRuby 就可以做。所以,可以使用 eRuby 进行文件操作。

7.4.1 文件读取

使用 Ruby 读取文件的基本方式如下。

案例名称:文件读取

程序名称:test.txt

```

我爱我老妈!
我爱我老爸!
i love you motherland!
great motherland!
China!

```

显示的结果如图 7-31 所示。

程序名称:file.erb

```

# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
path= "test.txt"
port = open(path)
begin
  port.each_line {|line|
    p line.to_s
  }
ensure
  port.close
end
print "< hr> "
port = open(path)
begin
  port.each_line {|line|
    print line
  }
ensure
  port.close
end
% >

```



图 7-31 test.txt

运行结果如图 7-32 所示。



图 7-32 文件读取

7.4.2 文件写入

使用 Ruby 写入文件的两种方式是:内容追加和内容重写。其实现方法如下:

案例名称:文件内容追加

程序名称:file2. erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content- type: text/html
< %
n= "test.txt"
# read
port = open(n)
begin
  port.each_line {|line|
    print line
  }
ensure
  port.close
end
print "< hr> "
# append
f= File.open(n,File::RDWR)
f.seek(0, IO::SEEK_END)
str= "\n伟大啊! 高尚啊!"
f.write(str)
f.close
# read
port = open(n)
begin
  port.each_line {|line|
    print line
  }
ensure
  port.close
end
% >
```

运行结果如图 7-33 所示。

查看 test.txt,其内容如图 7-34 所示。

案例名称:文件内容重写

程序名称:file3. erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content- type: text/html
< %
n= "test.txt"
```



图 7-33 文件内容追加



图 7-34 test.txt

```
# read
port = open(n)
begin
  port.each_line {|line|
    print line
  }
ensure
  port.close
end
print "< hr> "
# modify
f= File.open(n,File::RDWR)
f.truncate(0)
str= "\n 伟大啊! 高尚啊! 我的老爸老妈!"
f.write(str)
f.close
# read
port = open(n)
begin
  port.each_line {|line|
    print line
  }
end
```

```
ensure
  port.close
end
% >
```

运行结果如图 7-35 所示。

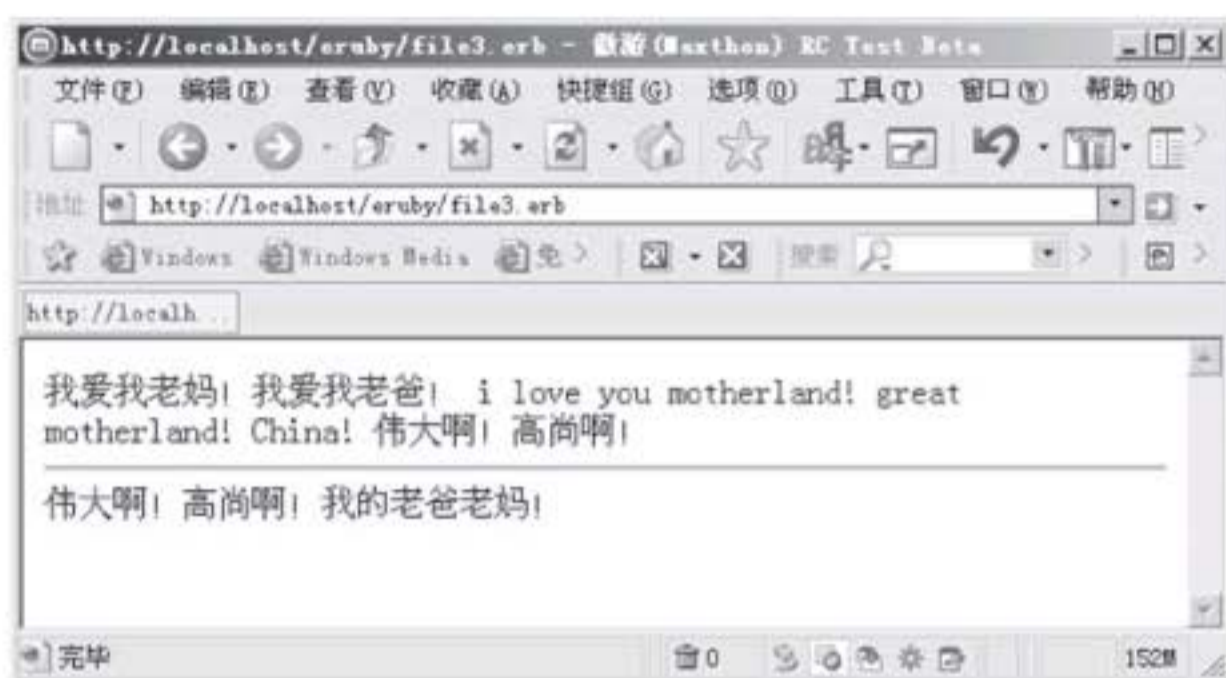


图 7-35 文件内容重写

查看 test.txt, 其内容如图 7-36 所示。



图 7-36 test.txt

7.5 数据库操作

Ruby 可以做的, eRuby 就可以做。所以, 我们可以使用 eRuby 进行数据库操作。

7.5.1 数据读取

现在, 用 eRuby 读取数据库的内容, 将其显示在网页上。

案例名称: 数据库操作

程序名称: db.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'mysql'
```

...

```

con = Mysql.new('localhost','root','tianen','my')
print "< table border= 1> "
rs= con.query("select * from test")
rs.each_hash(with_table= false) {|x|
print "< tr> < td> " + x['name'] + " < /td> < td> " + x['hobby'] + " < /td> < /tr> "
}
print "< /table> "
rs.free()
con.close
% >

```

运行结果如图 7-37 所示。

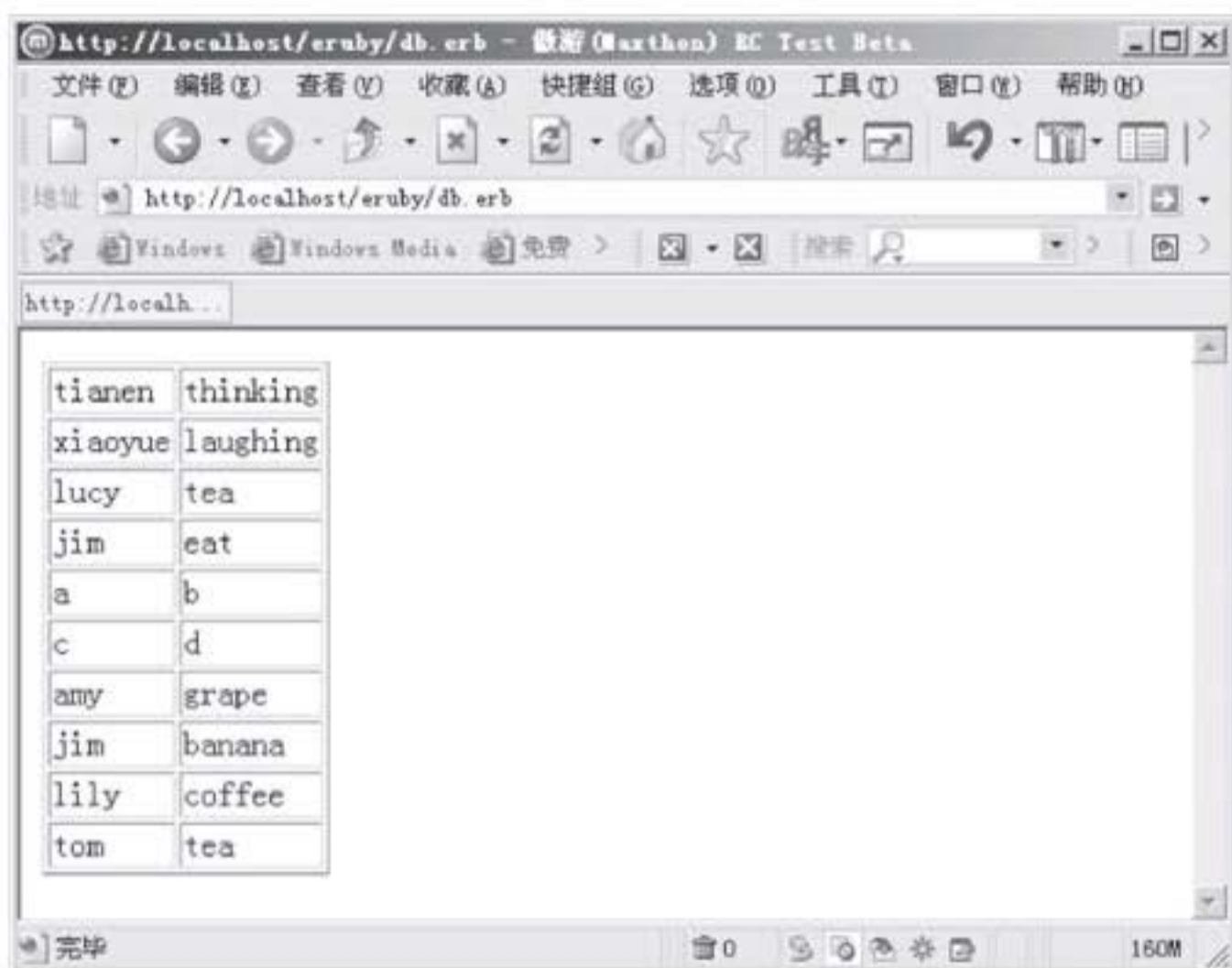


图 7-37 数据库读取

程序名称:dba.erb

```

# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
num= rs.num_rows()
print "< table border= 1> "
for i in 0..num- 1
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + " < /td> < td> " + a['hobby'] + " < /td> < /tr> "
end

```

```

print "< /table> "
rs.free()
con.close
% >

```

运行结果如图 7-38 所示。

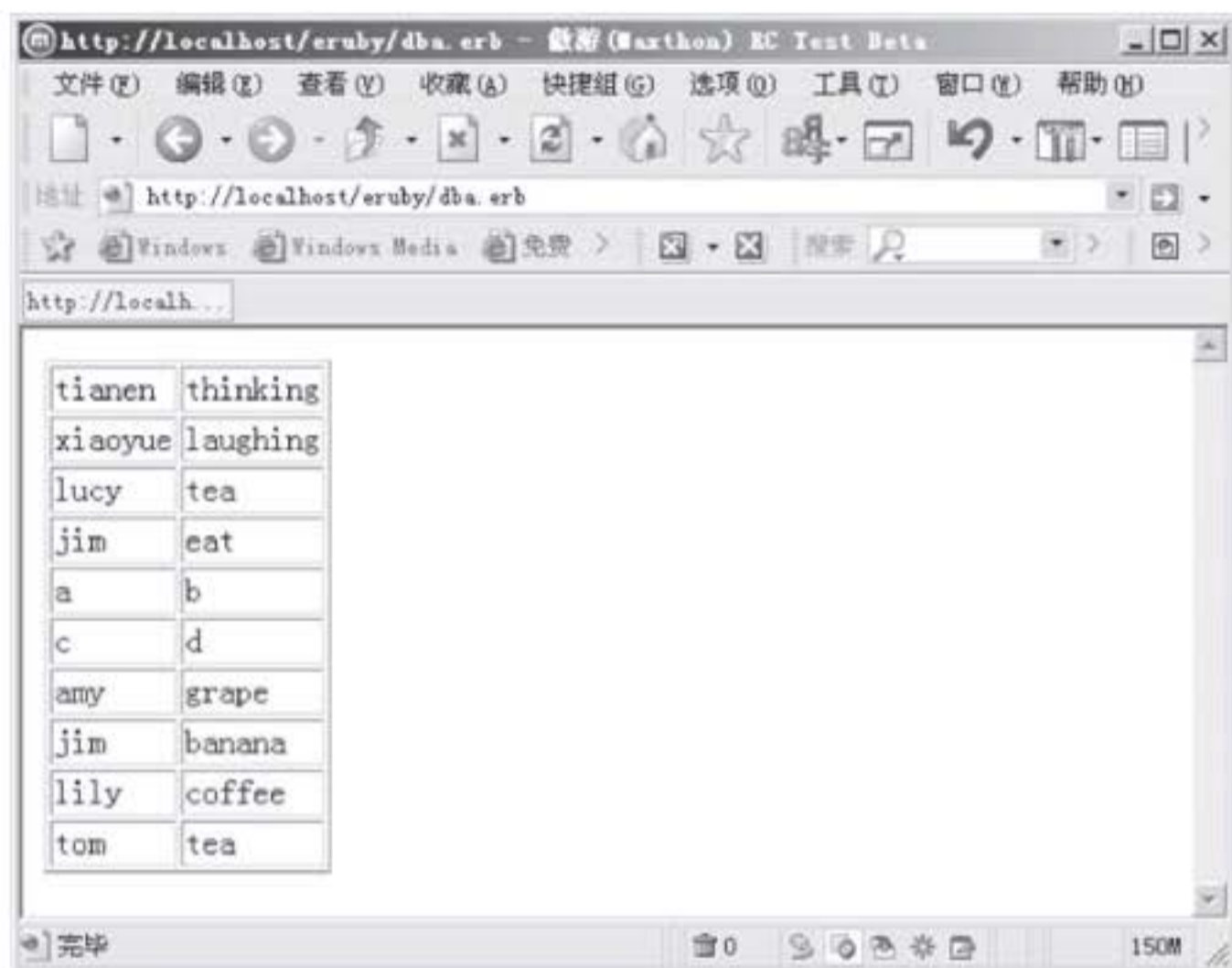


图 7-38 数据库读取

7.5.2 分页显示数据

用户可以实现数据库信息的分页显示。其原理是：使用 data_seek 方法定位数据记录。现在逐步来实现数据库信息的分页显示。

第一步：静态表格分割

程序名称：dbb.erb

```

# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
num= rs.num_rows()
print "< table border= 1> "
for i in 0..2
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "

```

```

print "< table border= 1> "
for i in 3..5
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
print "< table border= 1> "
for i in 6..8
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
print "< table border= 1> "
for i in 9..num- 1
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
rs.free()
con.close
% >

```

运行结果如图 7-39 所示。

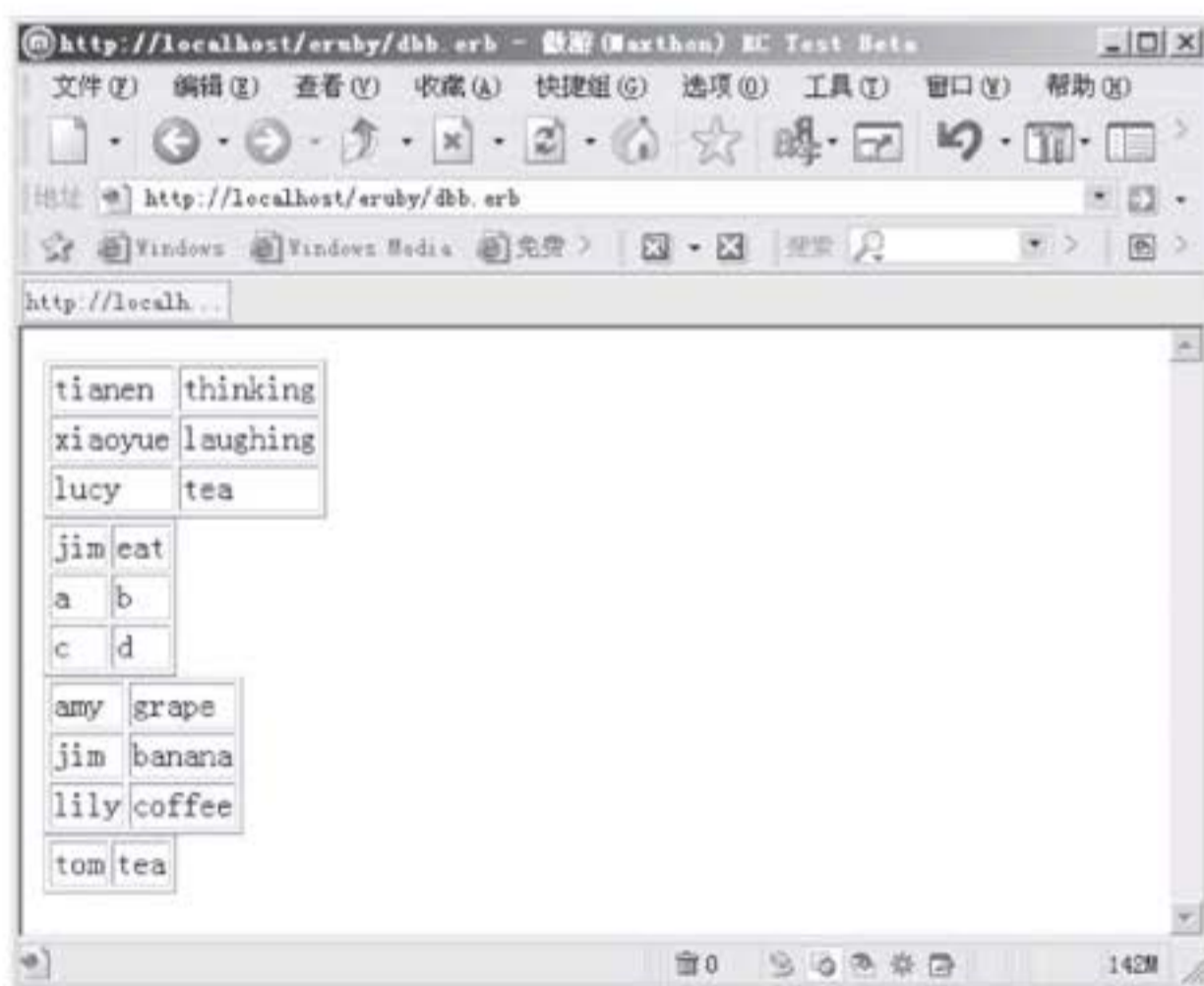


图 7-39 第一步:静态表格分割

从这一步中,读者可以看到,每 3 条记录分成一组,10 条记录分成了 4 组,显示在了 4 个表格中。

下面来计算应该产生多少个表格。

第二步:计算表格数目

程序名称:dbc.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
num= rs.num_rows()
# 每个表格中的记录数目
size= 3
# 表格数
pagenum= 1
# num< size
if num/size == 0 then
  pagenum= 1
else
  # 整除
  if num% size== 0 then
    pagenum= num/size
  else
    pagenum= num/size+ 1
  end
end
end
print "记录总数:",num,"< br> "
print "一个表格含有的记录数量:",size,"< br> "
print "表格数目:",pagenum,"< br> "
rs.free()
con.close
% >
```

运行结果如图 7-40 所示。

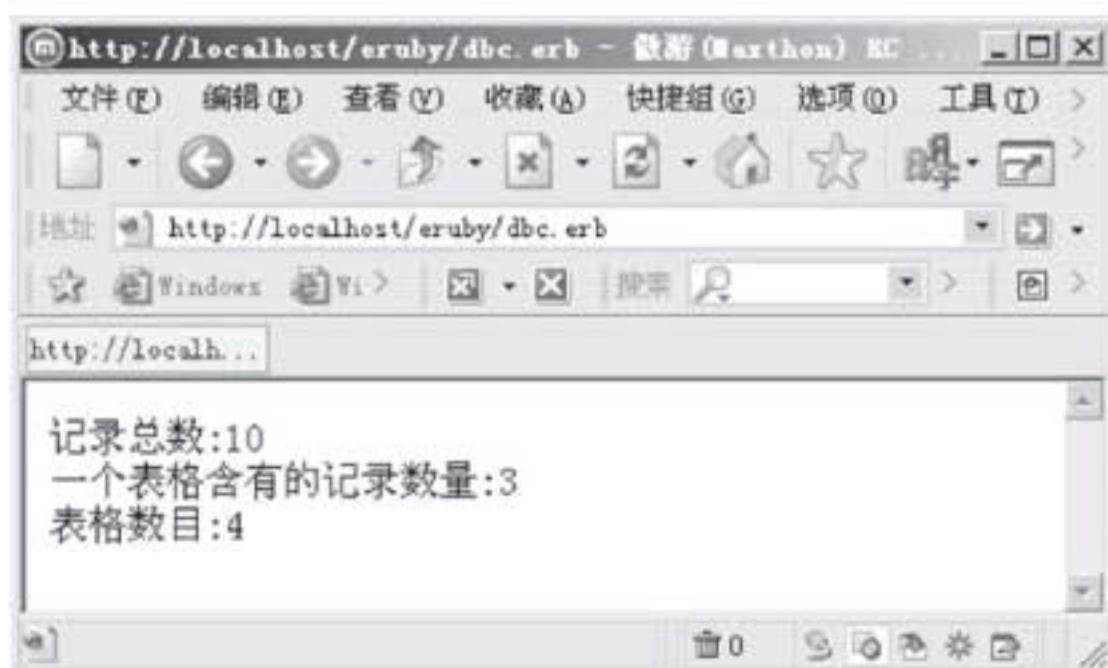


图 7-40 第二步:计算表格数目

这里,每个表格含有的记录数量就是每页含有的记录数量。计算出来的表格的数目就是分页的页数。

下面正式开始分页操作。

第三步:分页

程序名称:dbd.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
#####
require 'cgi'
cgi = CGI.new
pa = cgi['page'].to_i
if ! pa then
  pa= 0
end
#####
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
num= rs.num_rows()
# 每个表格中的记录数目
size= 3
# 结束点
pe= pa* size+ 3- 1
if pe> num- 1 then
  pe= num- 1
end
print "< table> "
for i in pa* size..pe
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
#####
# 表格数
pagenum= 1
# num< size
if num/size == 0 then
  pagenum= 1
else
  # 整除
  if num% size== 0 then
    pagenum= num/size
  else
```

```

    pagenum= num/size+ 1
  end
end
print "记录总数:< b> # {num}< /b> "
print "每页最多含有的记录数量:< b> # {size}< /b> "
print "页数:< b> # {pagenum}< /b> "
print "当前页:< b> # {pa+ 1}< /b> "
print "< p> "
print "分页:"
for i in 0..pagenum- 1
  print "< a href= dbd.erb? page= # {i}> # {i+ 1}< /a> "
end
#####
rs.free()
con.close
% >

```

运行结果如图 7-41,图 7-42 和图 7-43 所示。

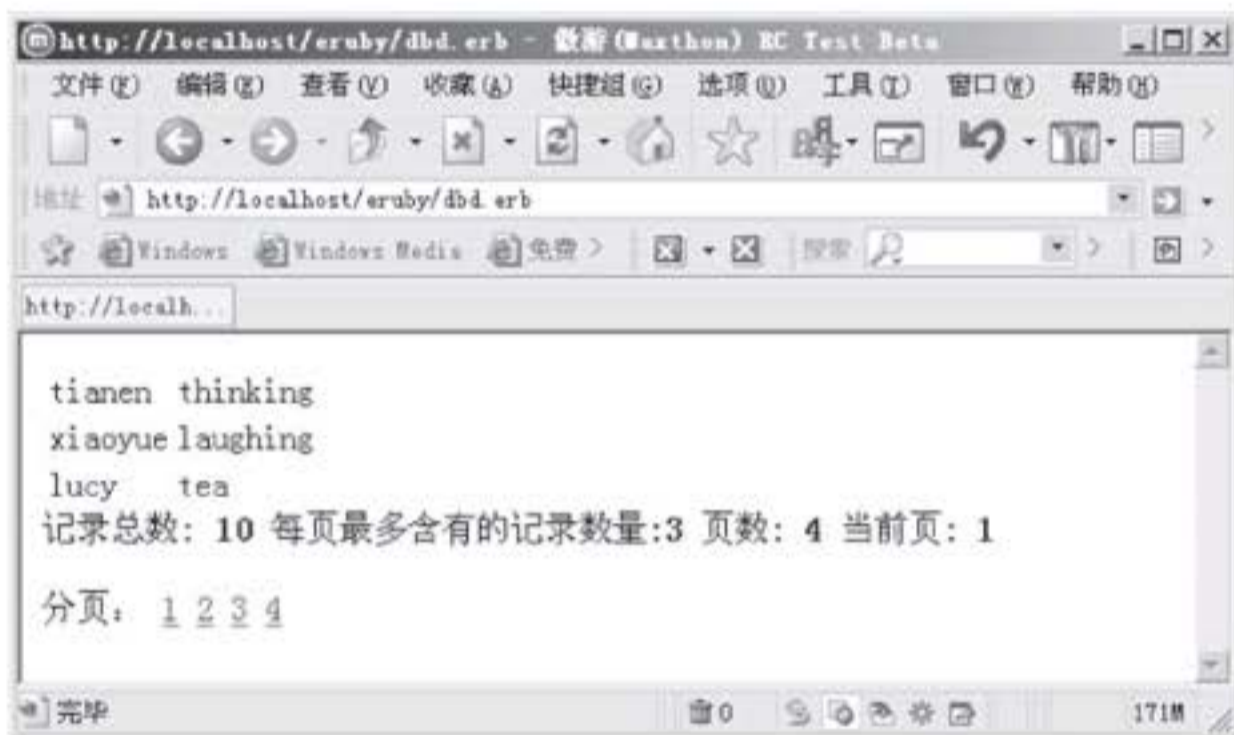


图 7-41 第三步:分页

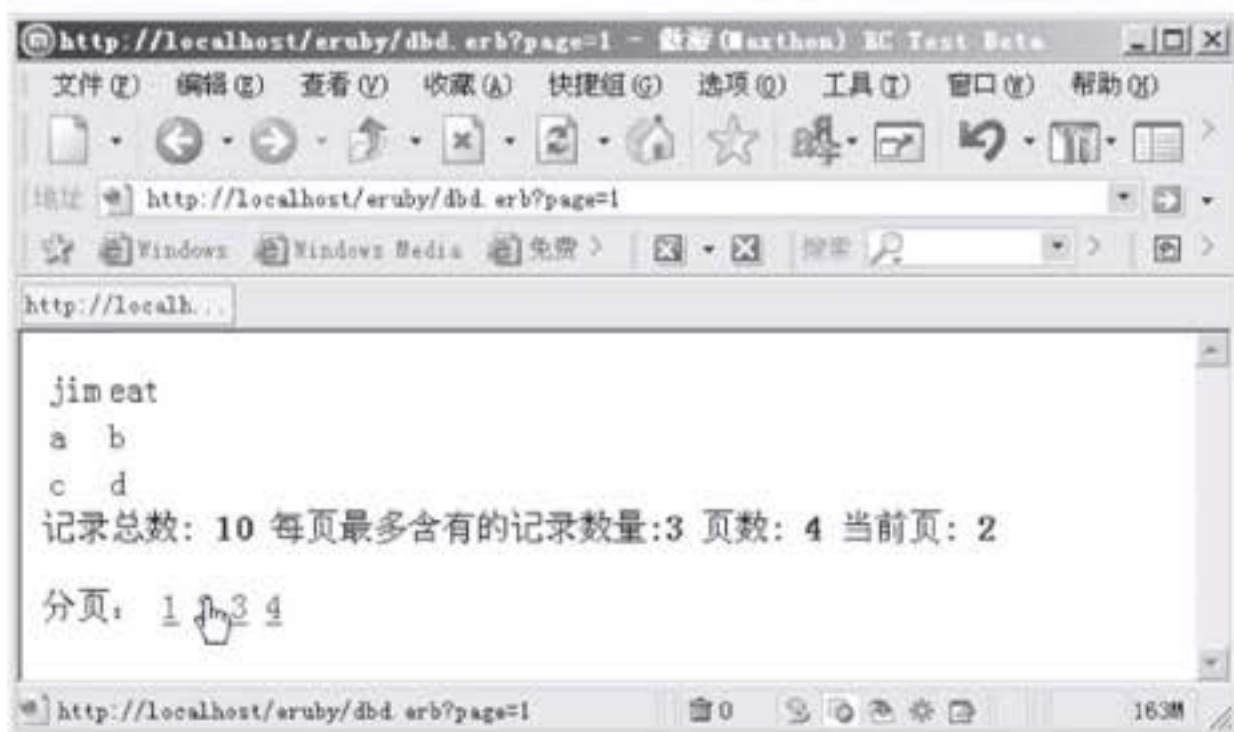


图 7-42 第三步:分页



图 7-43 第三步:分页

7.5.3 数据更新

数据更新,就是执行“update,insert,delete”等语句。本节举几个例子,以便读者学习。

案例名称:删除数据

程序名称:dbe.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'mysql'
con = Mysql.new('localhost', 'root', 'tianen', 'my')
rs= con.query("select * from test")
num= rs.num_rows()
print "< table border= 1> "
for i in 0..num- 1
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
rs.free()
print "< hr/> "
con.query("delete from test where name= 'a'")
rs= con.query("select * from test")
num= rs.num_rows()
print "< table border= 1> "
for i in 0..num- 1
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
```

```

end
print "< /table> "
con.close
% >

```

运行结果如图 7-44 所示。

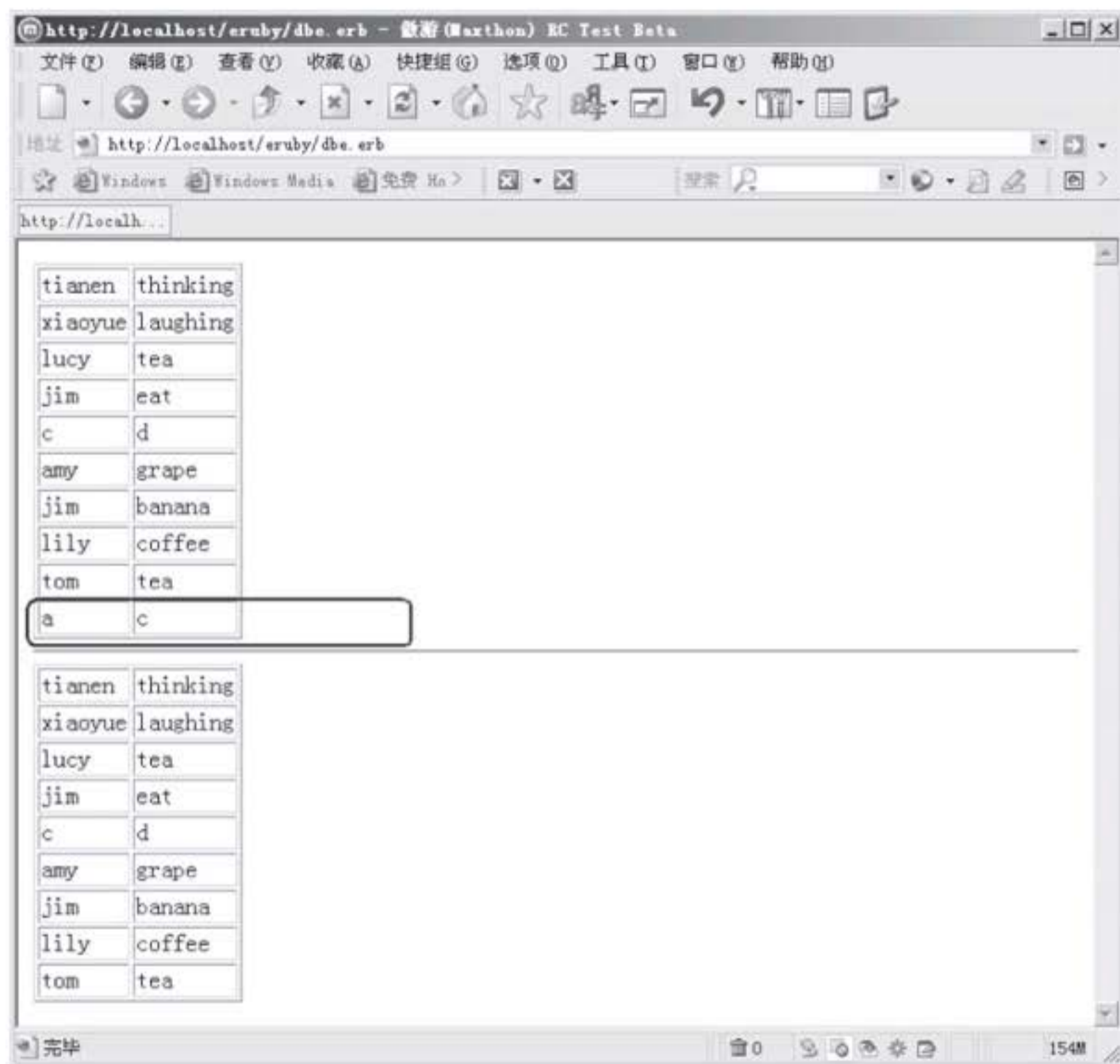


图 7-44 删除数据

案例名称:添加数据

程序名称:dbf.erb

```

# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< %
require 'mysql'
con = Mysql.new('localhost', 'root', 'tianen', 'my')
rs= con.query("select * from test")
num= rs.num_rows()
print "< table border= 1> "
for i in 0..num- 1
  rs.data_seek(i)

```

```

a= rs.fetch_hash()
print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
rs.free()
print "< hr/> "
con.query("insert into test values('天恩','伟大啊!')")
rs= con.query("select * from test")
num= rs.num_rows()
print "< table border= 1> "
for i in 0..num- 1
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
con.close
% >

```

运行结果如图 7-45 所示。

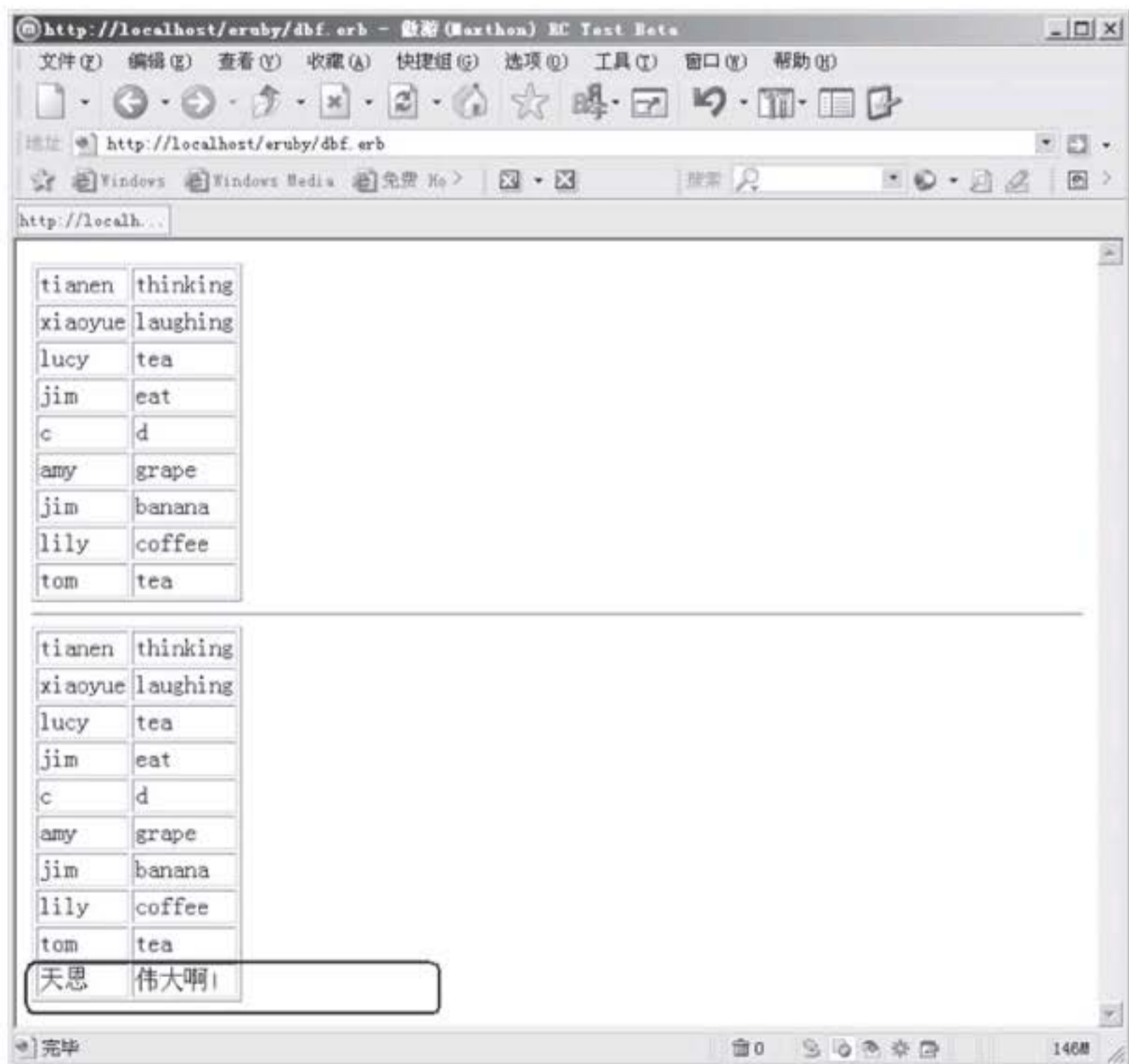


图 7-45 添加数据

案例名称:修改数据

程序名称:dbg.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content- type: text/html
< %
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
rs= con.query("select * from test")
num= rs.num_rows()
print "< table border= 1> "
for i in 0..num- 1
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
rs.free()
print "< hr/> "
con.query("update test set name= '老爸',hobby= '伟大啊!' where name= 'c'")
con.query("update test set name= '老妈',hobby= '伟大啊!' where name= 'tom'")
con.query("update test set name= '小月',hobby= '伟大啊!' where name= 'amy'")
rs= con.query("select * from test")
num= rs.num_rows()
print "< table border= 1> "
for i in 0..num- 1
  rs.data_seek(i)
  a= rs.fetch_hash()
  print "< tr> < td> " + a['name'] + "< /td> < td> " + a['hobby'] + "< /td> < /tr> "
end
print "< /table> "
con.close
% >
```

运行结果如图 7-46 所示。

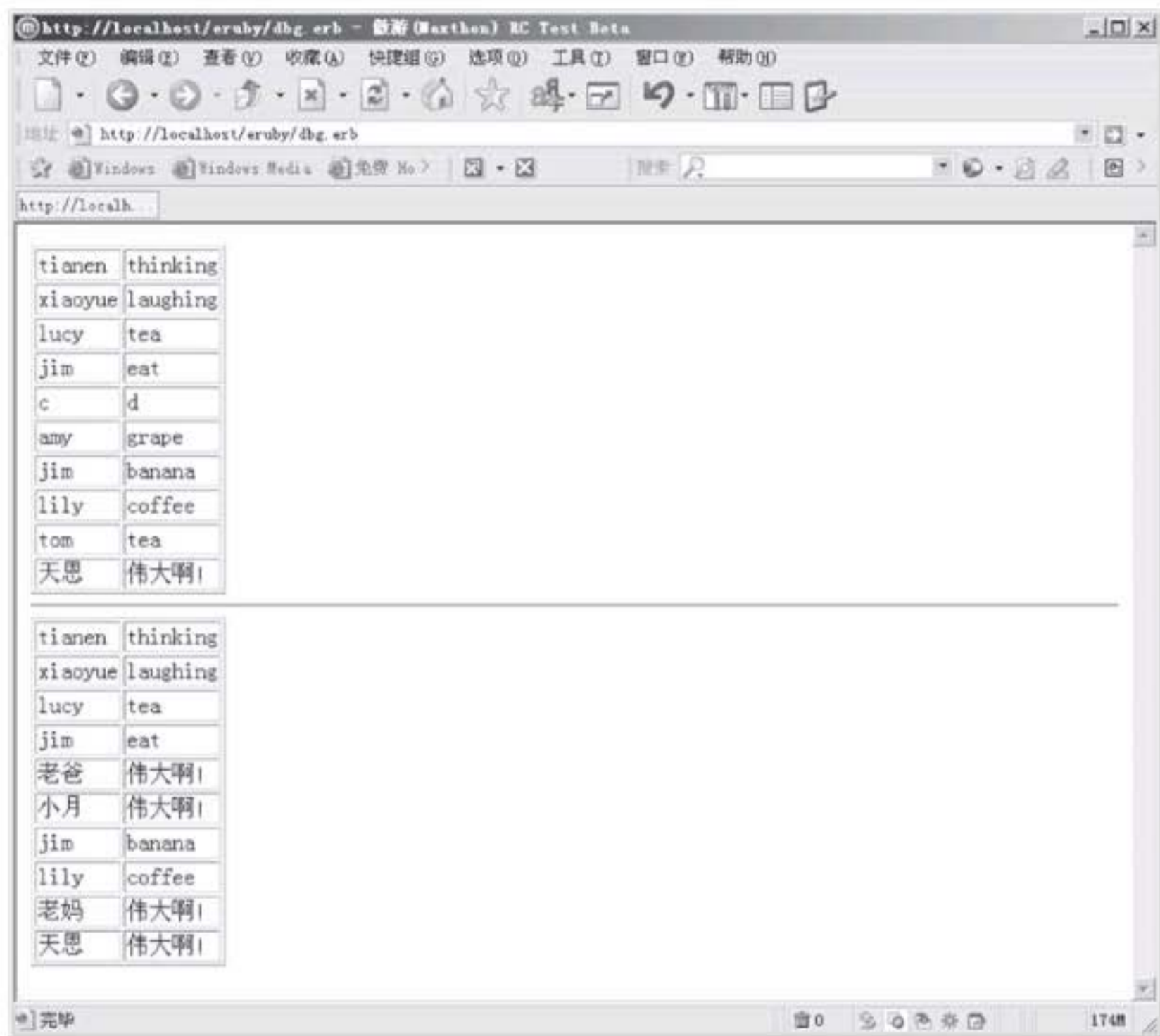


图 7-46 修改数据

7.6 session 的基本用法

eRuby 可以用 session 和 cookie 来存储用户信息。在此对 session 的用法进行介绍。

有了 session, 就可以很轻松地实施页面保护和制作权限系统。

Ruby 的 session 用法和 PHP 类似, 十分简洁。下面的代码就建立了一个 session 对象:

```
require "cgi/session"
cgi = CGI.new
session = CGI::Session::new(cgi)
```

下面介绍一个最基本的 session 使用案例。

案例名称: session 的基本用法

程序名称: sa.erb

```
< %
require "cgi/session"
cgi = CGI.new
```

```

session = CGI::Session::new(cgi)
session["test"] = "aaa"
print session["test"]
% >

```

运行结果如图 7-47 所示。

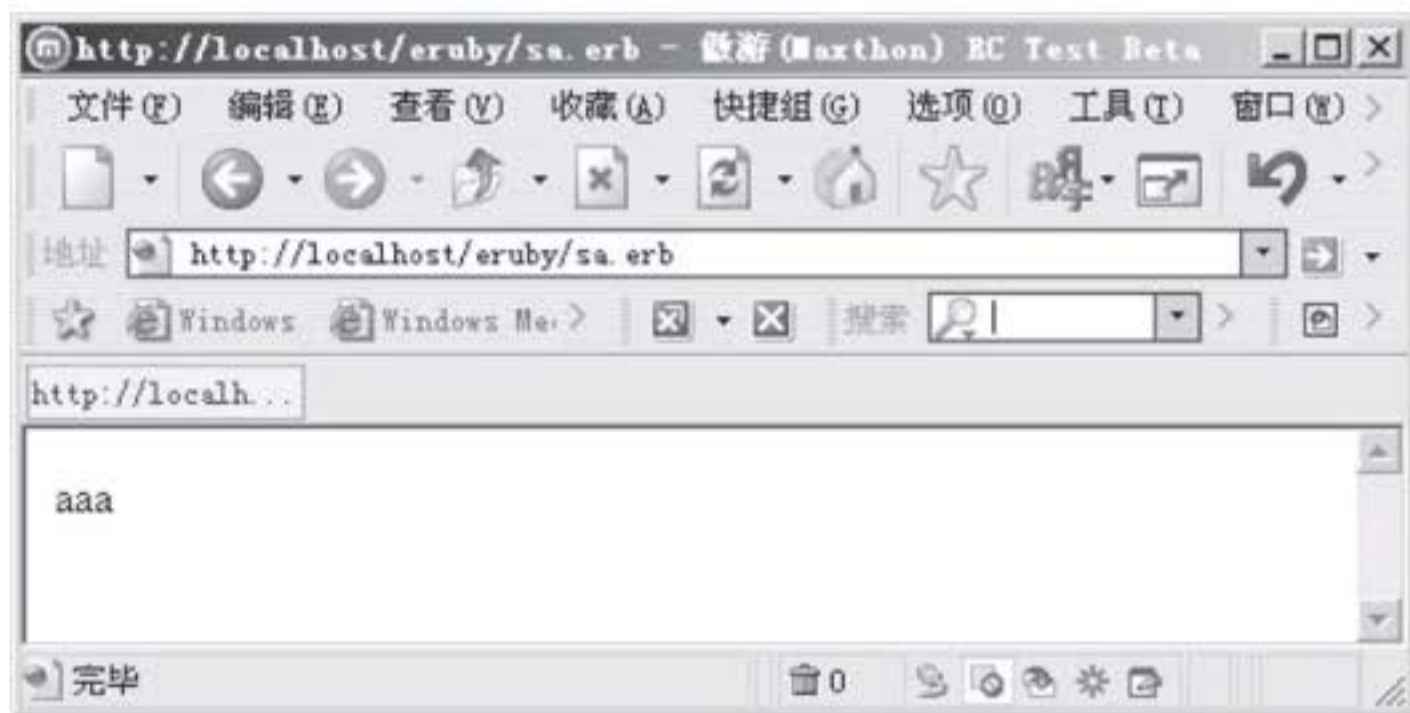


图 7-47 session 的基本用法

7.7 Web 开发案例

在本书的最后,总结一切知识,写成如下 4 个案例。读者应认真揣摩,将全书内容贯通。

7.7.1 留言本(基于文本文件)

现在做一个基于文本文件的留言本。使用文本文件作为存储数据的介质,在程序运行时读取文本文件的内容显示出来,将用户新留言的内容写入文本文件。

这个案例由两个程序组成。一个是文本文件“lyb.txt”,这个文件最初是空的。另一个文件是“lyb.erb”,负责显示留言内容并处理新的留言。

案例名称:留言本(基于文本文件)

程序名称:lyb.erb

```

# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< h4> 留言本< /h4>
< ! - - 显示原来的内容 - - >
< % n= "lyb.txt"% >
< % ERuby.import(n) % >
< hr/>
< form action= lyb.erb method= post>
姓名:< input name= "xm"> < br>
内容:

```

```

< textarea name= "nr"> < /textarea>
< br>
< input type= submit value= "do"/>
< /form>
< %
require 'cgi'
cgi = CGI.new
if cgi['xm'] && cgi['nr'] then
  if ! cgi['xm'].empty? && ! cgi['nr'].empty? then
    xm= cgi['xm']
    nr= cgi['nr']

    f= File.open(n,File::RDWR)
    f.seek(0, IO::SEEK_END)
    str= "姓名:" + xm + "\n< br> "
    str+= "内容:" + "\n< br> "
    str+= nr
    str+= "< hr/> "
    f.write(str)
    f.close
    print "< script> location.href= 'lyb.erb'< /script> "
  end
end
end
% >

```

程序最初运行结果如图 7-48 所示。

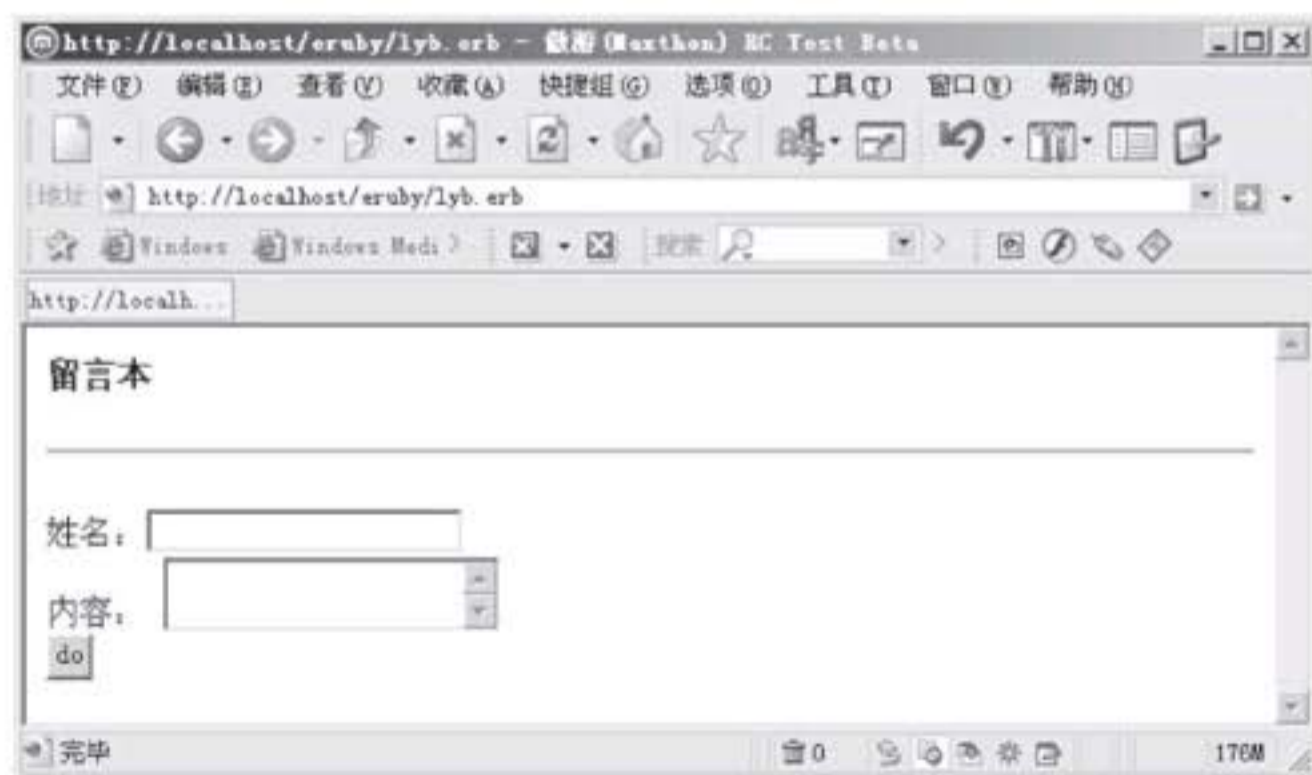


图 7-48 留言本程序

在文本框中输入内容,然后单击按钮,新留言即被写入。如图 7-49,图 7-50 和图 7-51 所示。

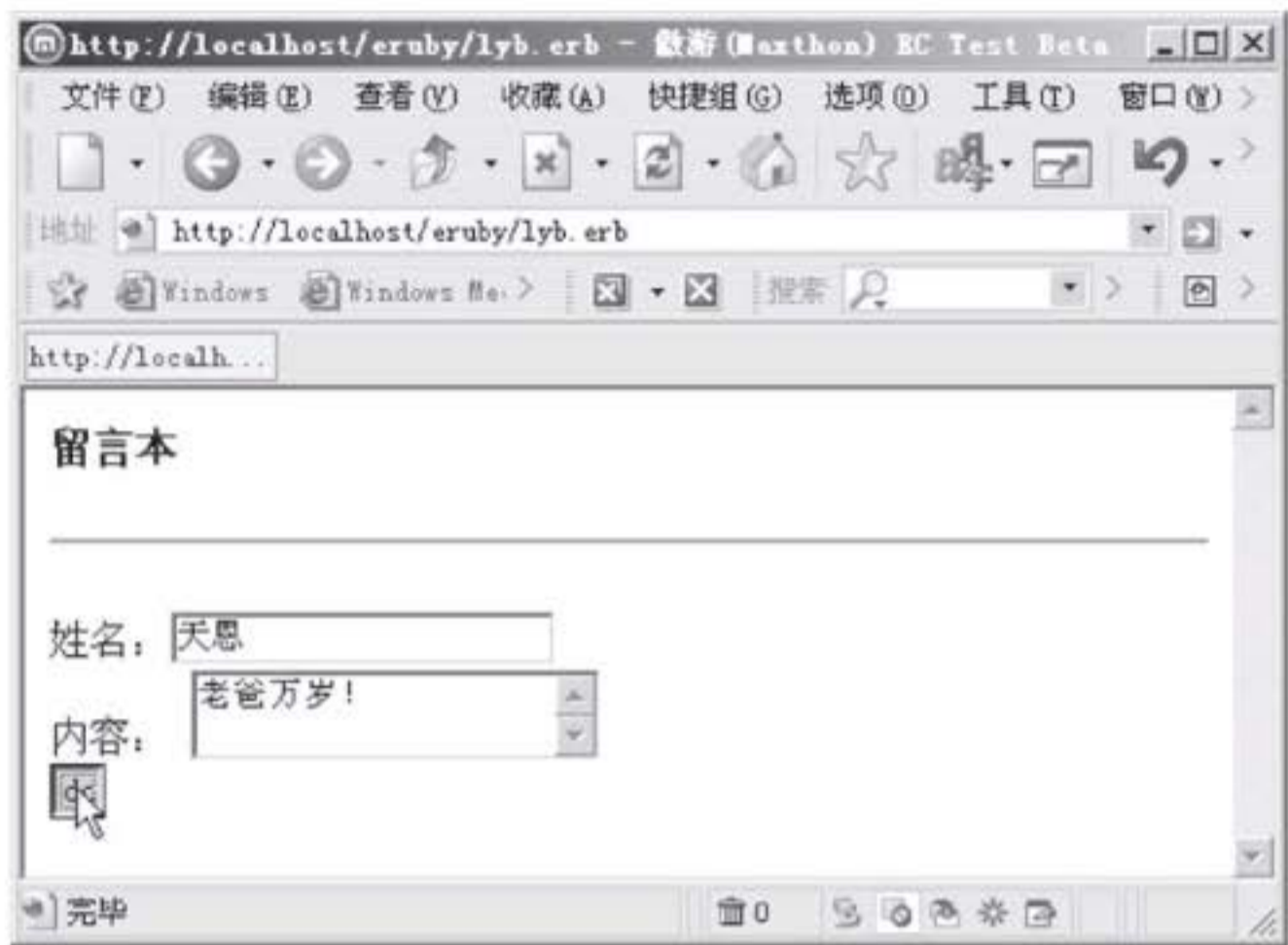


图 7-49 留言本程序

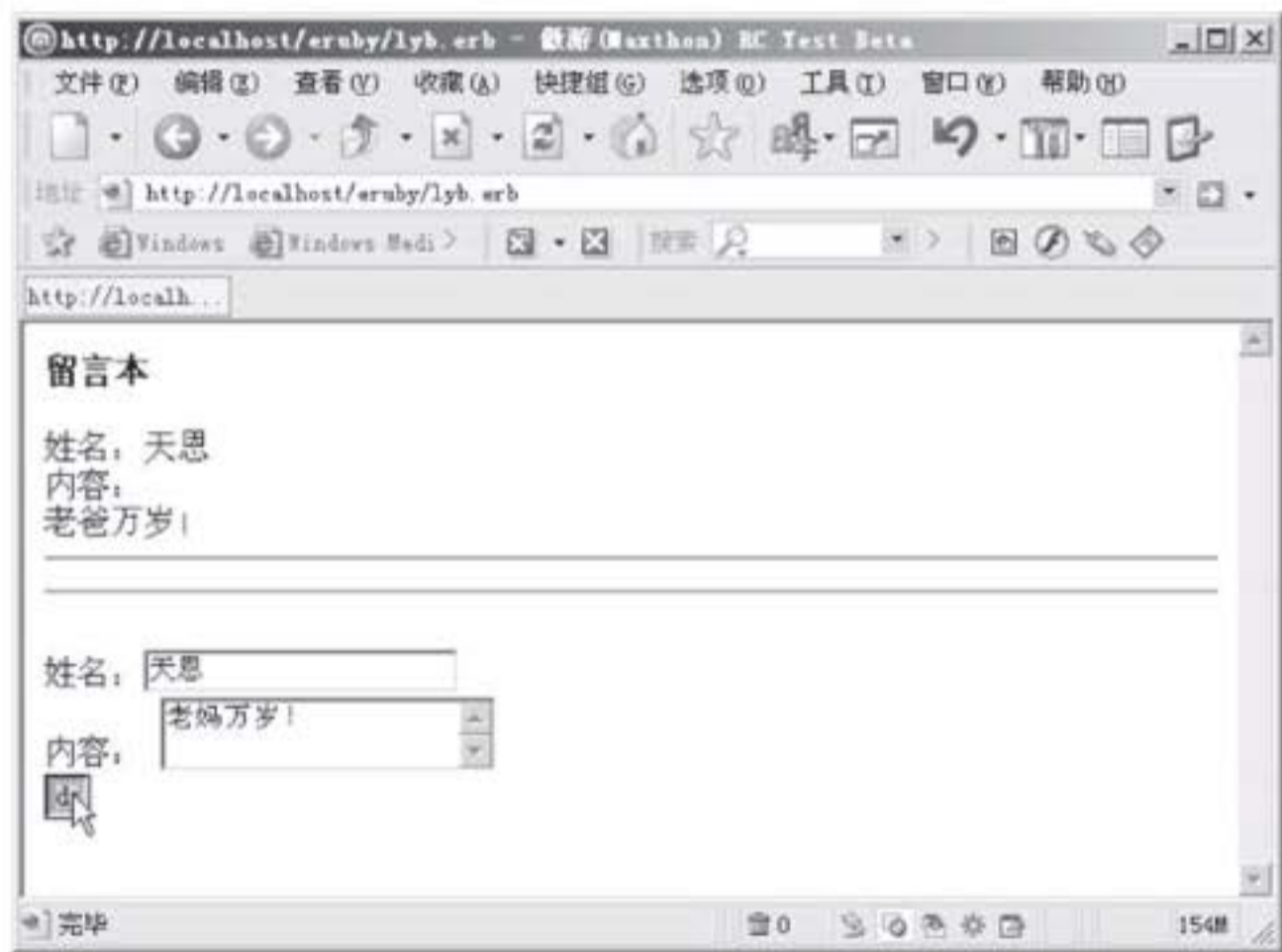


图 7-50 留言本程序

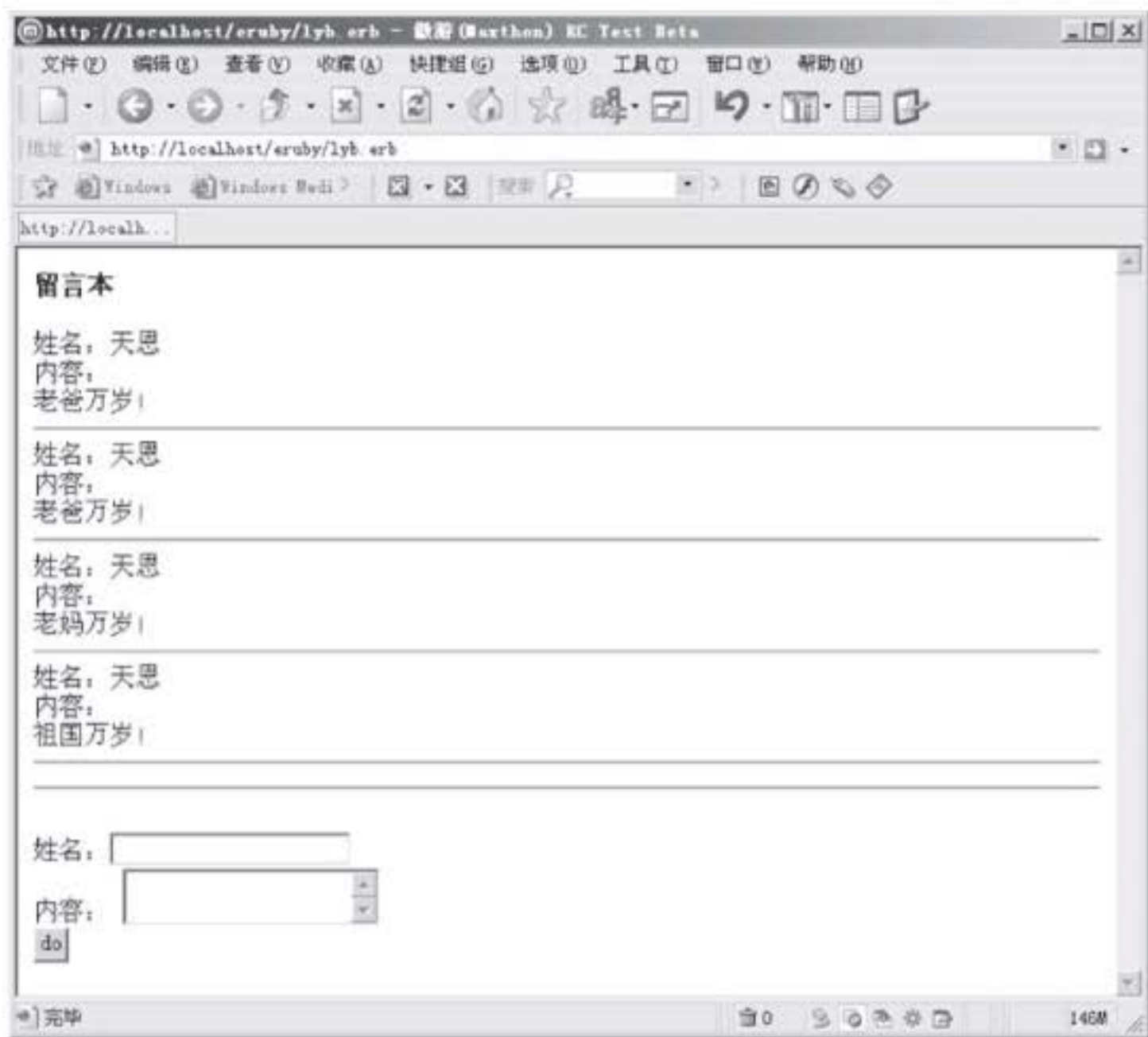


图 7-51 留言本程序

7.7.2 聊天室(基于文本文件)

聊天室程序和留言本差不多,不同之处在于要求实时显示留言信息。这就需要用定时刷新和网页框架。这里给出最基本的例子,这是一切聊天室程序的原理。

这个案例包含 4 个文件:

lt.txt 是文本文件,用于存储聊天信息的内容。

ltup.erb 每 3 秒自动刷新一次,用于读取 lt.txt 文件的内容,从而及时显示出聊天信息。

ltdown.erb 是发言文件,用于输入发言内容并提交发言。

ltmain.erb 是主框架文件,上框架是 ltup.erb 显示聊天内容,下框架 ltdown.erb 用于发言。

相关代码如下:

案例名称:聊天室(基于文本文件)

程序名称:ltup.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< h4> 聊天室< /h4>
< meta http-equiv= "refresh" content= "3;url= ltup.erb"/>
< ! - - 显示原来的内容 - - >
< % n= "lt.txt"% >
< % ERuby.import(n) % >
< hr/>
```

程序名称:lttdown.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< form action= ltdown.erb method= post>
姓名:< input name= "xm">
内容:< input name= "nr" size= 50>
< input type= submit value= "发言"/>
< /form>
< %
  n= "lt.txt"
  require 'cgi'
  cgi = CGI.new
  if cgi['xm'] && cgi['nr'] then
    if ! cgi['xm'].empty? && ! cgi['nr'].empty? then
      xm= cgi['xm']
      nr= cgi['nr']

      f= File.open(n,File::RDWR)
      f.seek(0, IO::SEEK_END)
      str= "姓名:" + xm + "\n< br> "
      str+ = "内容:" + "\n< br> "
      str+ = nr
      str+ = "< hr/> "
      f.write(str)
      f.close
      print "< script> location.href= 'lttdown.erb'< /script> "
    else
      print "< script> alert('请填写发言内容')< /script> "
    end
  else
    print "< script> alert('请填写发言内容')< /script> "
  end
% >
```

程序名称:ltmain.erb

```
< frameset rows= "400,* ">
< frame src= ltup.erb>
< frame src= ltdown.erb>
< /frameset>
```

程序最初运行结果如图 7-52 所示。

如果不填写任何内容就点击发言按钮的话,程序将会发出警告。如图 7-53 所示。现在,如图 7-54 和图 7-55 所示。可以发现,发言内容迅速显示出来了。

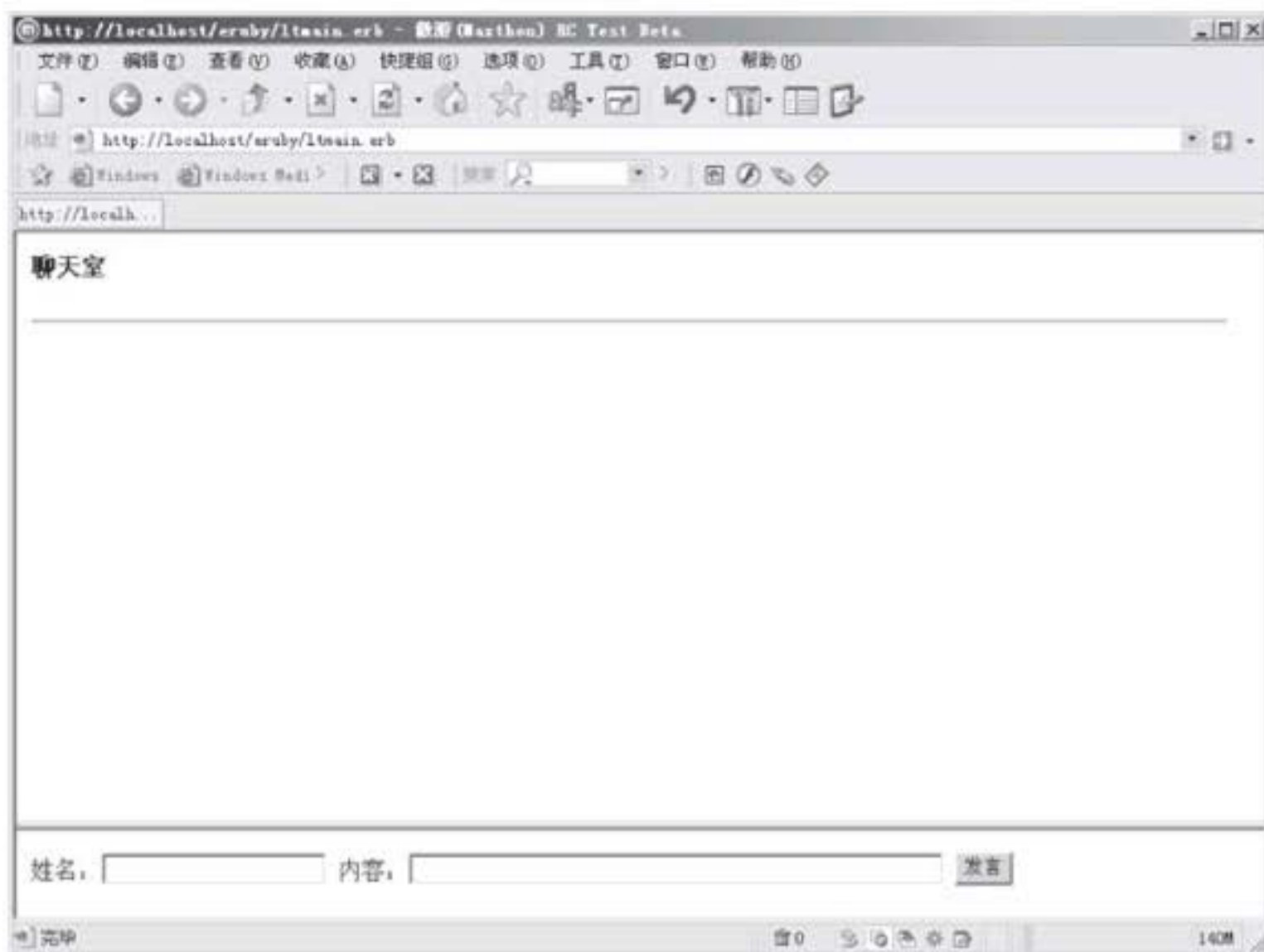


图 7-52 聊天室程序

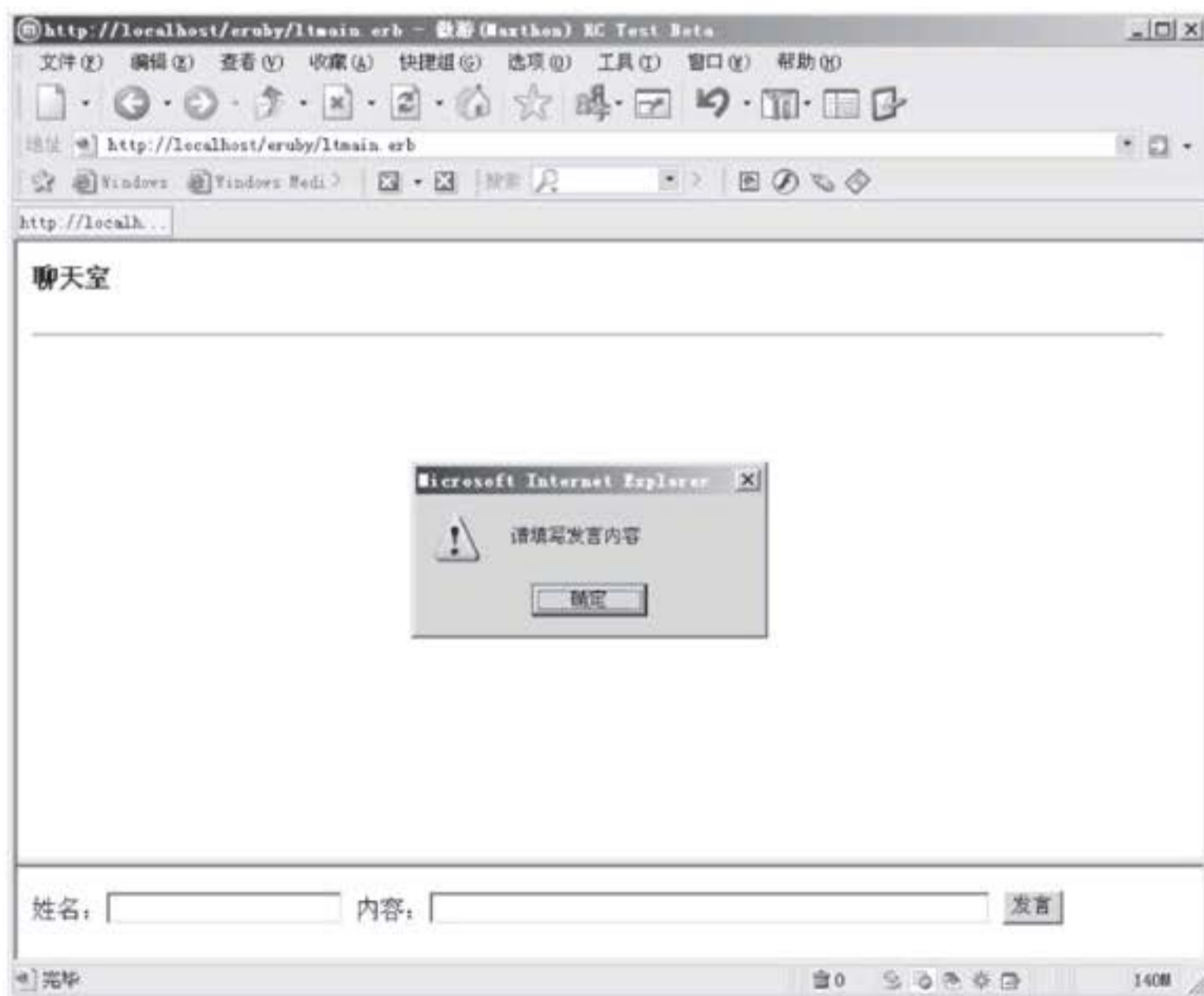


图 7-53 聊天室程序

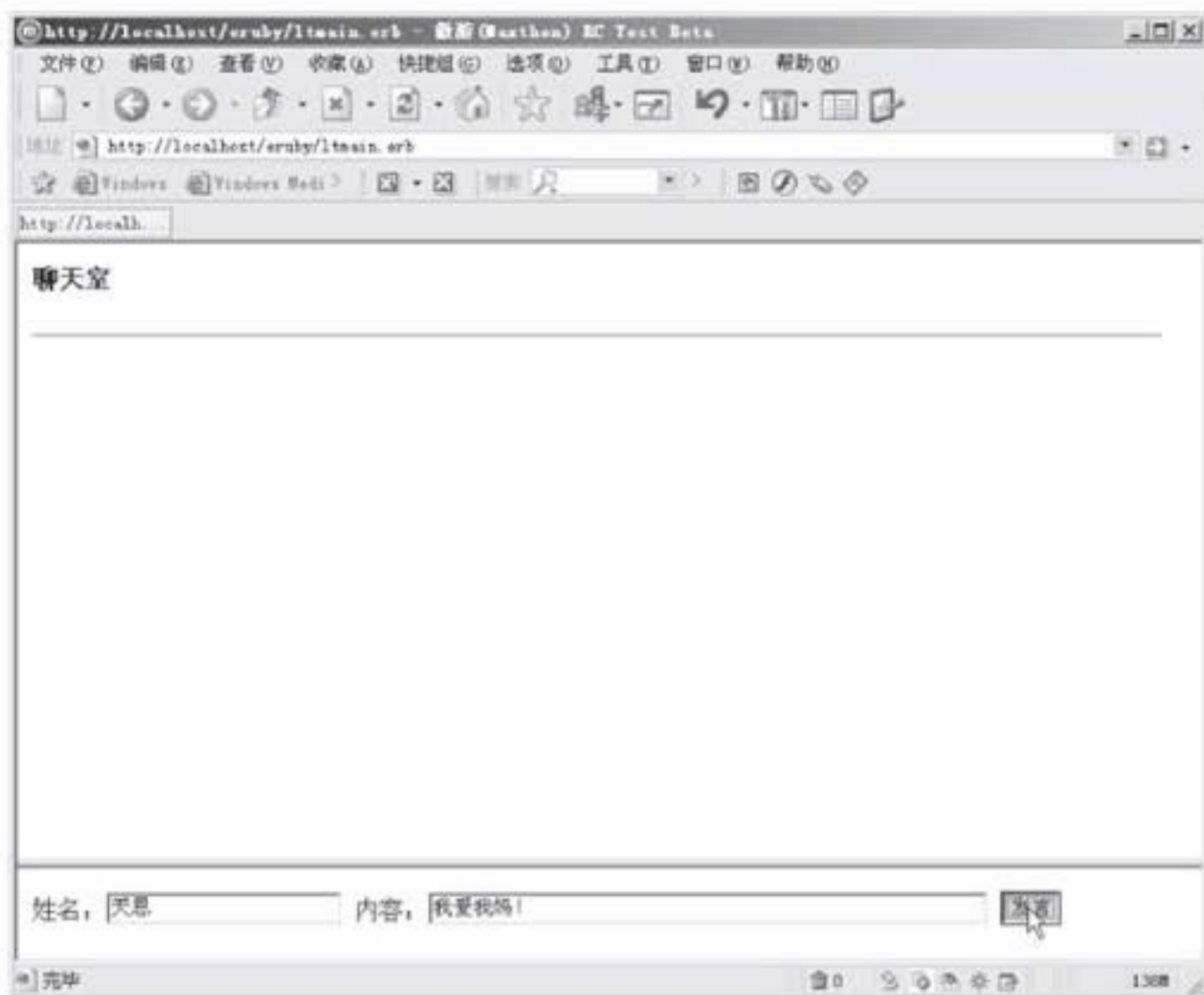


图 7-54 发 言

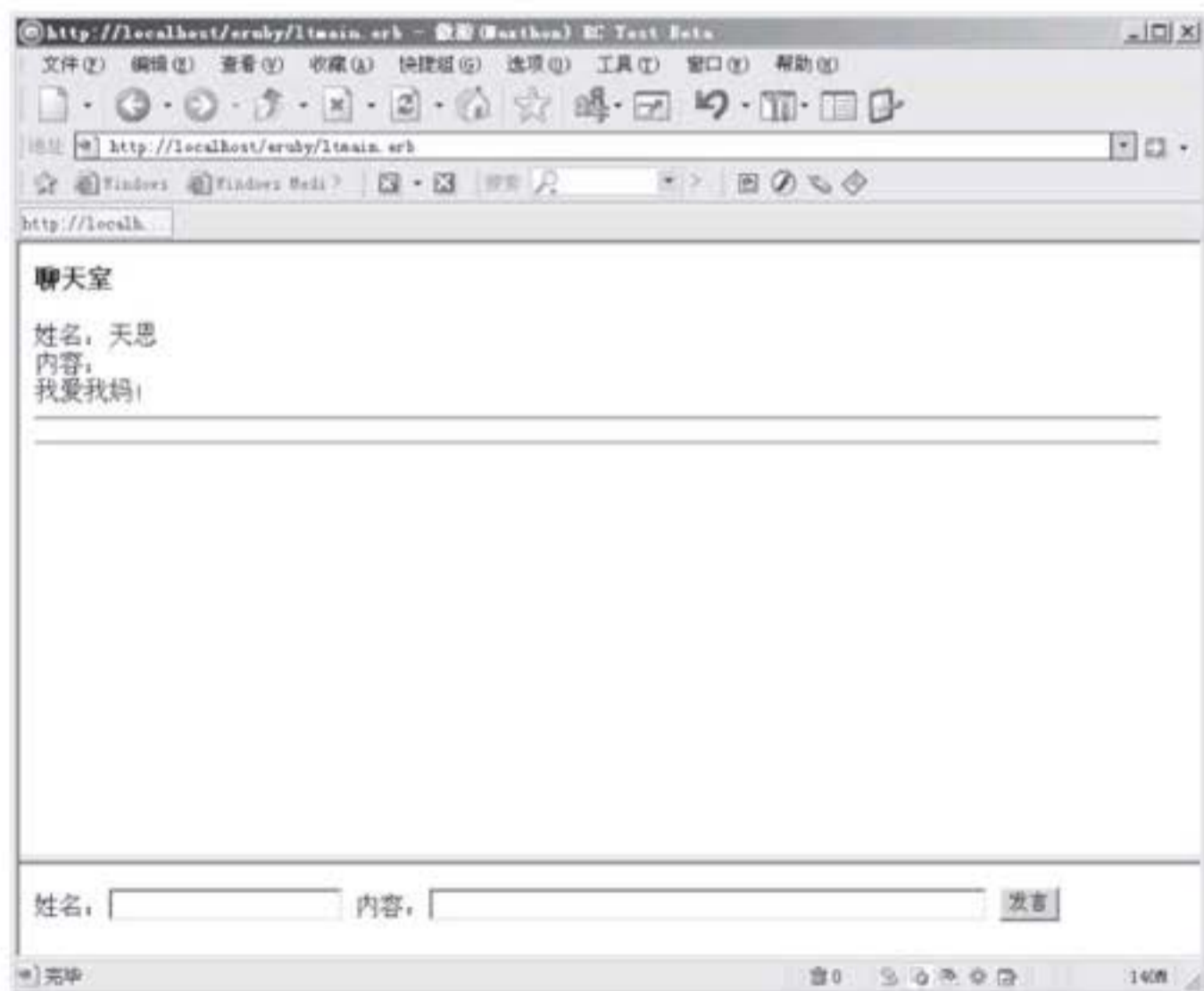


图 7-55 发 言

继续发言,如图 7-56 和图 7-57 所示。

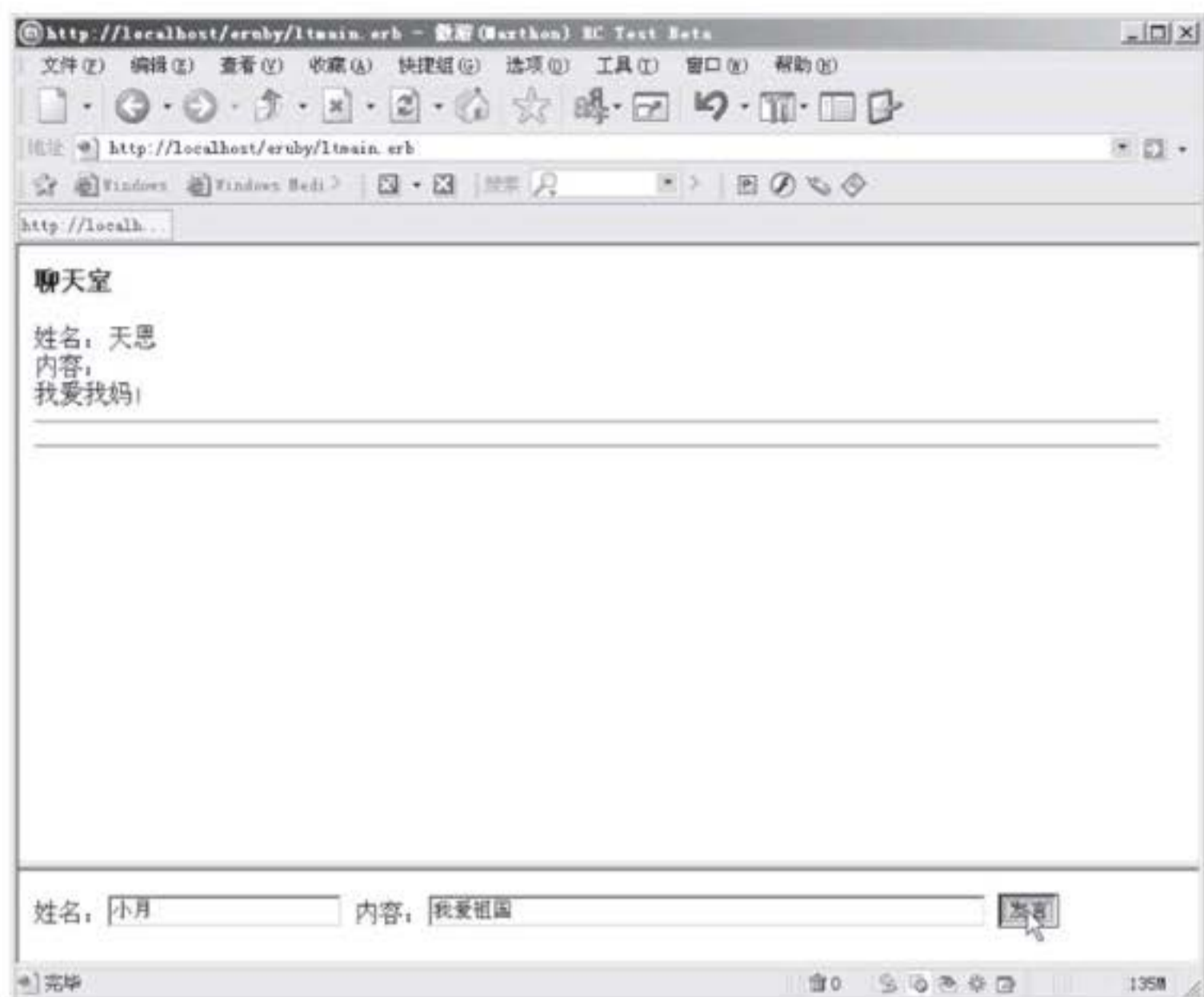


图 7-56 聊天室程序

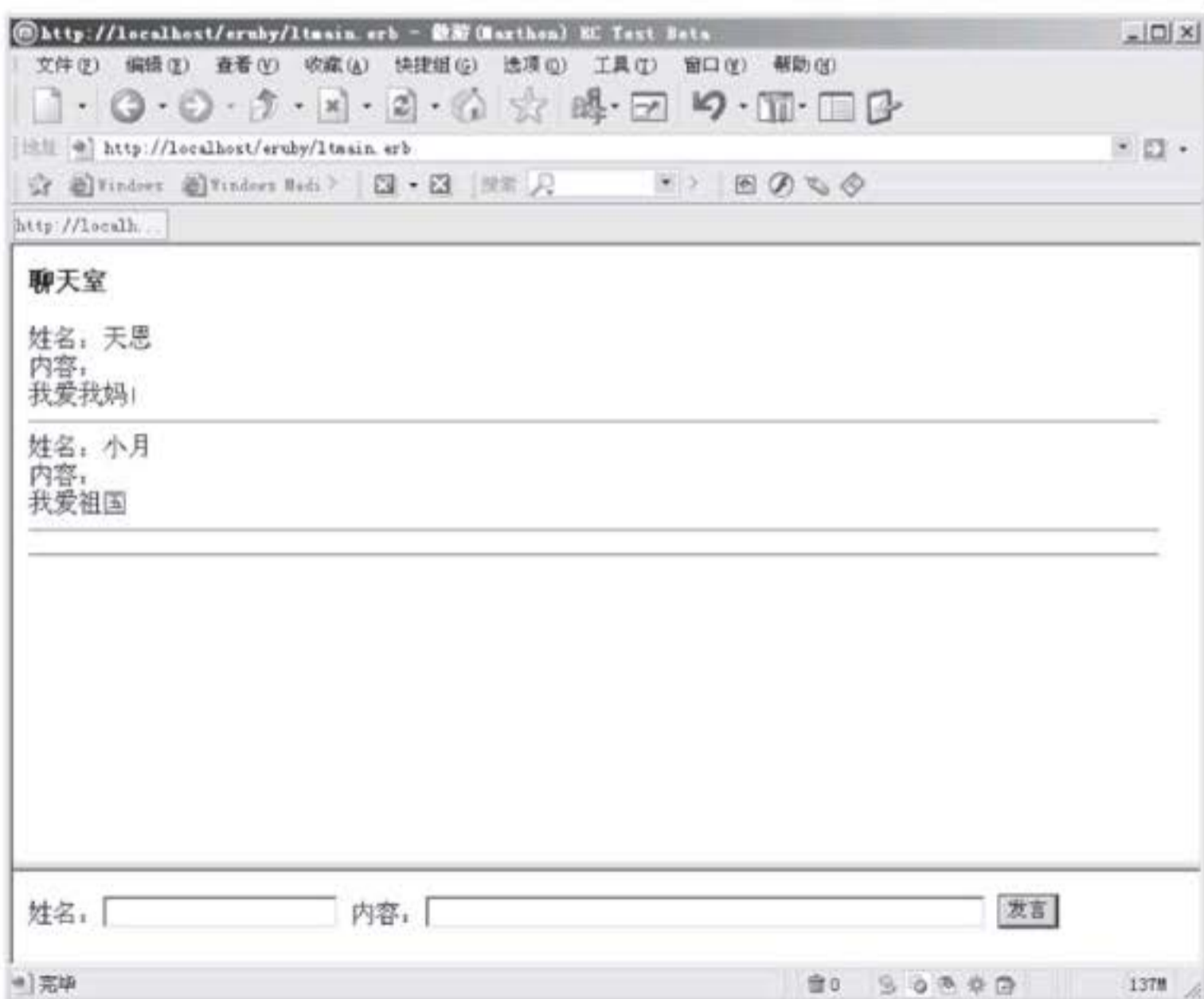


图 7-57 聊天室程序

7.7.3 留言本(基于数据库)

基于数据库的留言本,与前面做的留言本只有一处不同,就是存储介质。相应地,程序代码也有些差异。

使用 MySQL 数据库,在数据库“my”中建立表格,名为“lyb”。代码如下:

程序名称:lyb.sql

```
use my;
create table lyb
(
  xm char(20),
  nr char(50)
);
```

运行结果如图 7-58 所示。

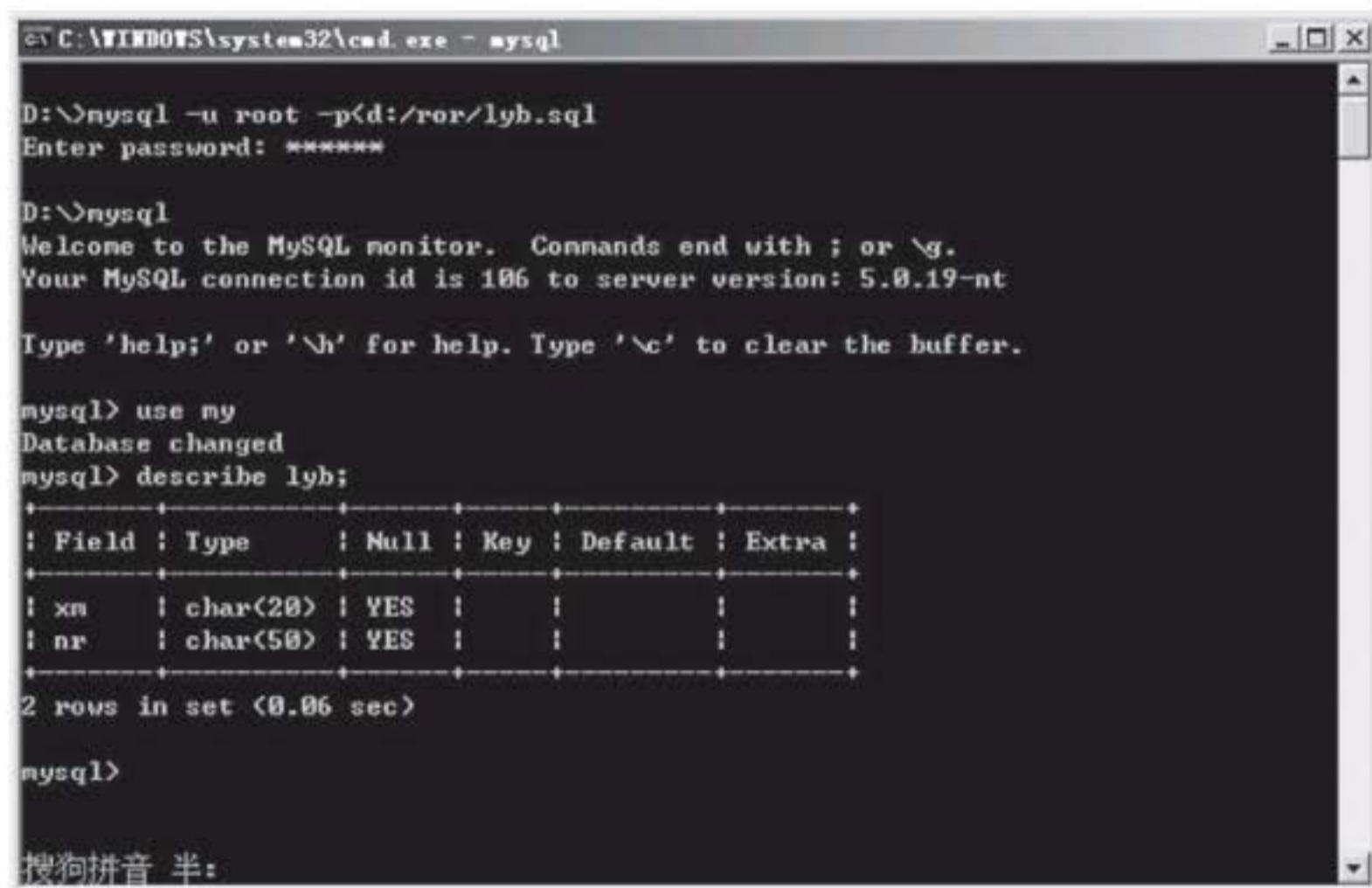


图 7-58 留言本数据库

程序名称:dblyb.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< h4> 留言本< /h4>
< %
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
begin
  rs= con.query("select * from lyb")
  num= rs.num_rows()
  for i in 0..num- 1
```

```

rs.data_seek(i)
a= rs.fetch_hash()
print "姓名:" + a['xm'] + "<br>" + "留言内容:<br>" + a['nr'] + "<hr/>"
end
rs.free()
#####
require 'cgi'
cgi = CGI.new
if cgi['xm'] && cgi['nr'] then
  if ! cgi['xm'].empty? && ! cgi['nr'].empty? then
    xm= cgi['xm']
    nr= cgi['nr']
    con.query("insert into lyb values('" + xm + "','" + nr + "')")
    print "<script> location.href= 'dblyb.erb'</script>"
  end
end
end
rescue
  print "error"
ensure
  con.close
end
% >
< form action= dblyb.erb method= post>
姓名:< input name= "xm"> <br>
内容:
< textarea name= "nr"> </textarea>
<br>
< input type= submit value= "do"/>
</form>

```

最初运行结果如图 7-59 所示。

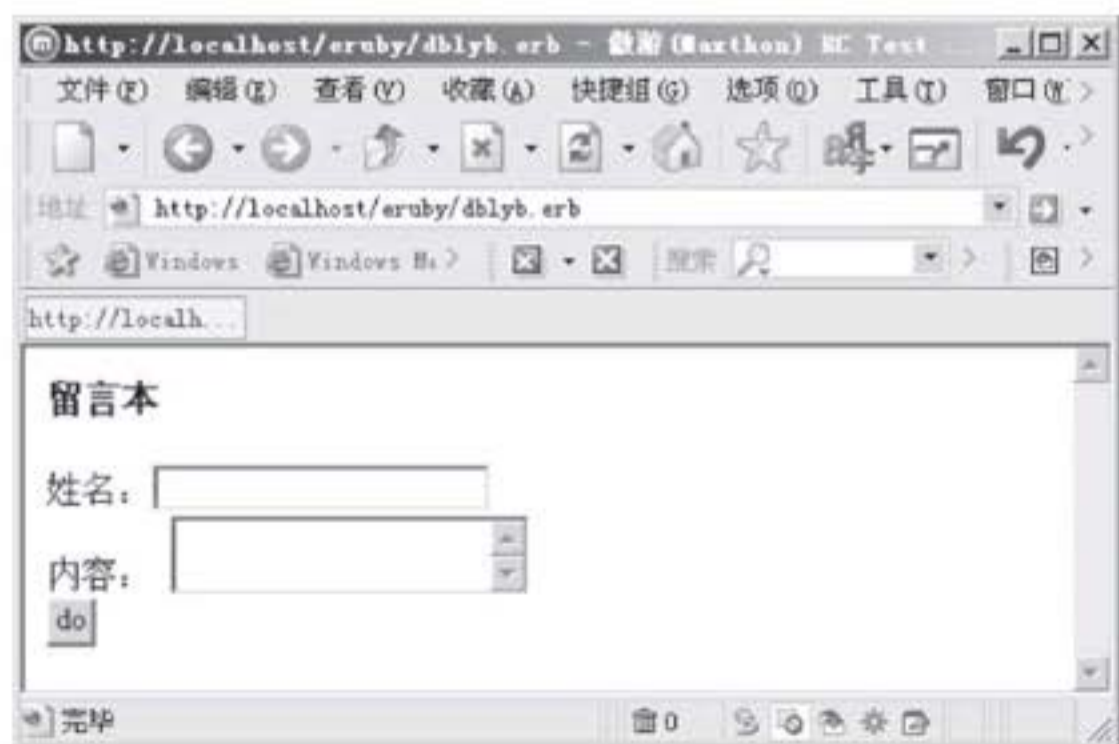


图 7-59 留言本

留言,结果如图 7-60 和图 7-61 所示。

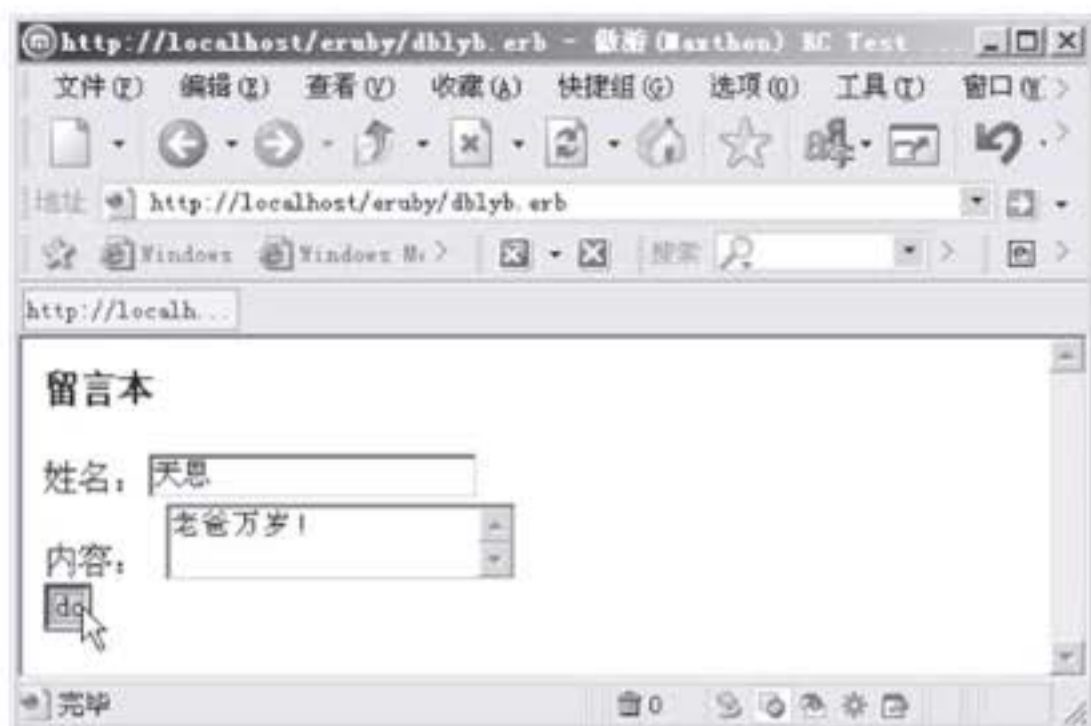


图 7-60 留 言

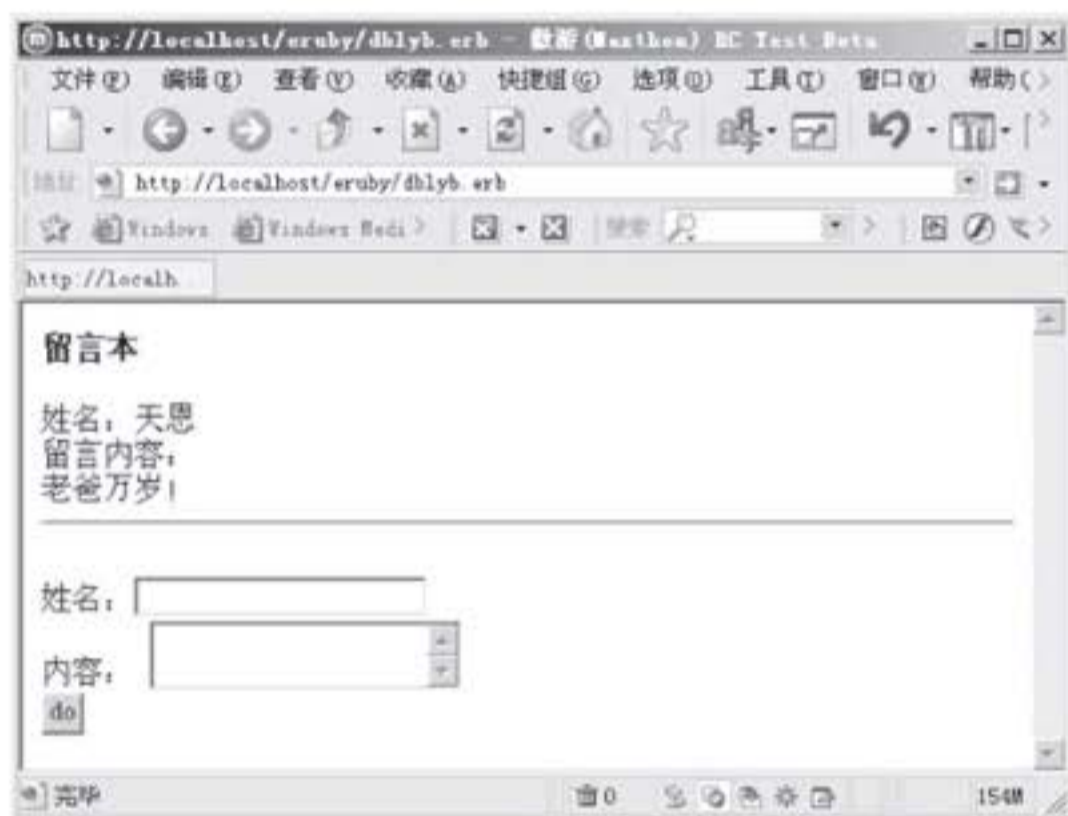


图 7-61 留 言

继续留言,结果如图 7-62 和图 7-63 所示。

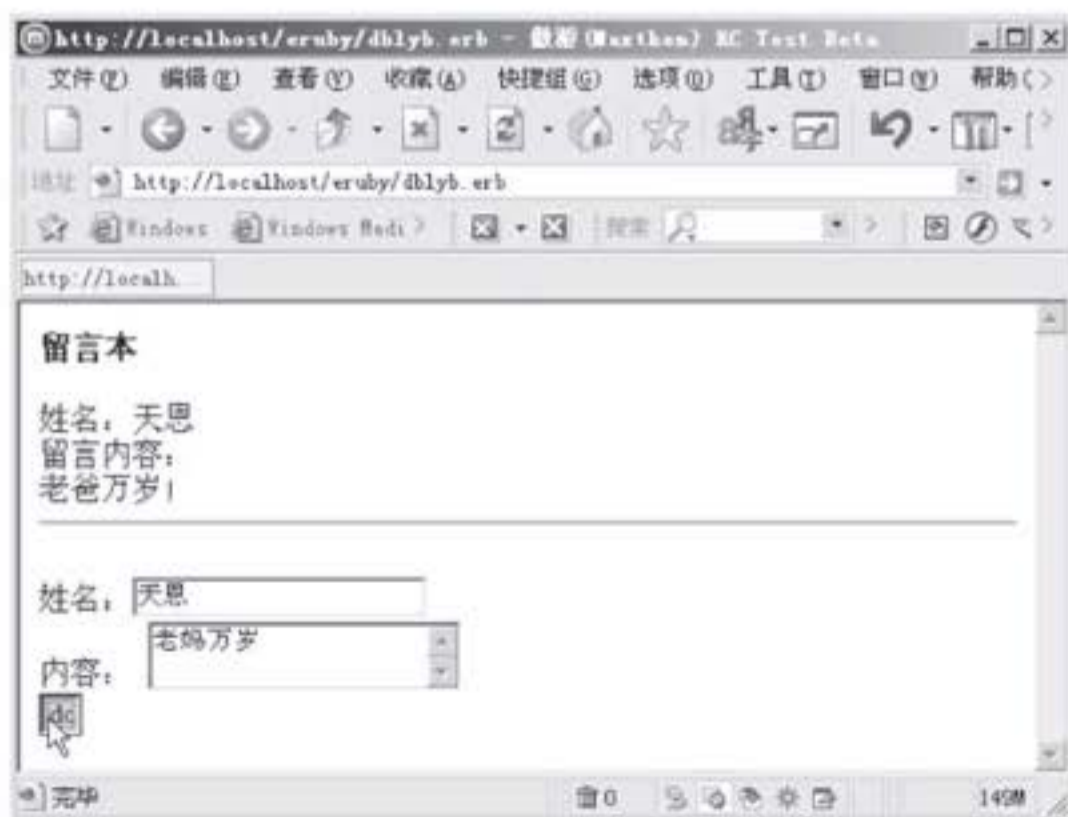


图 7-62 留 言

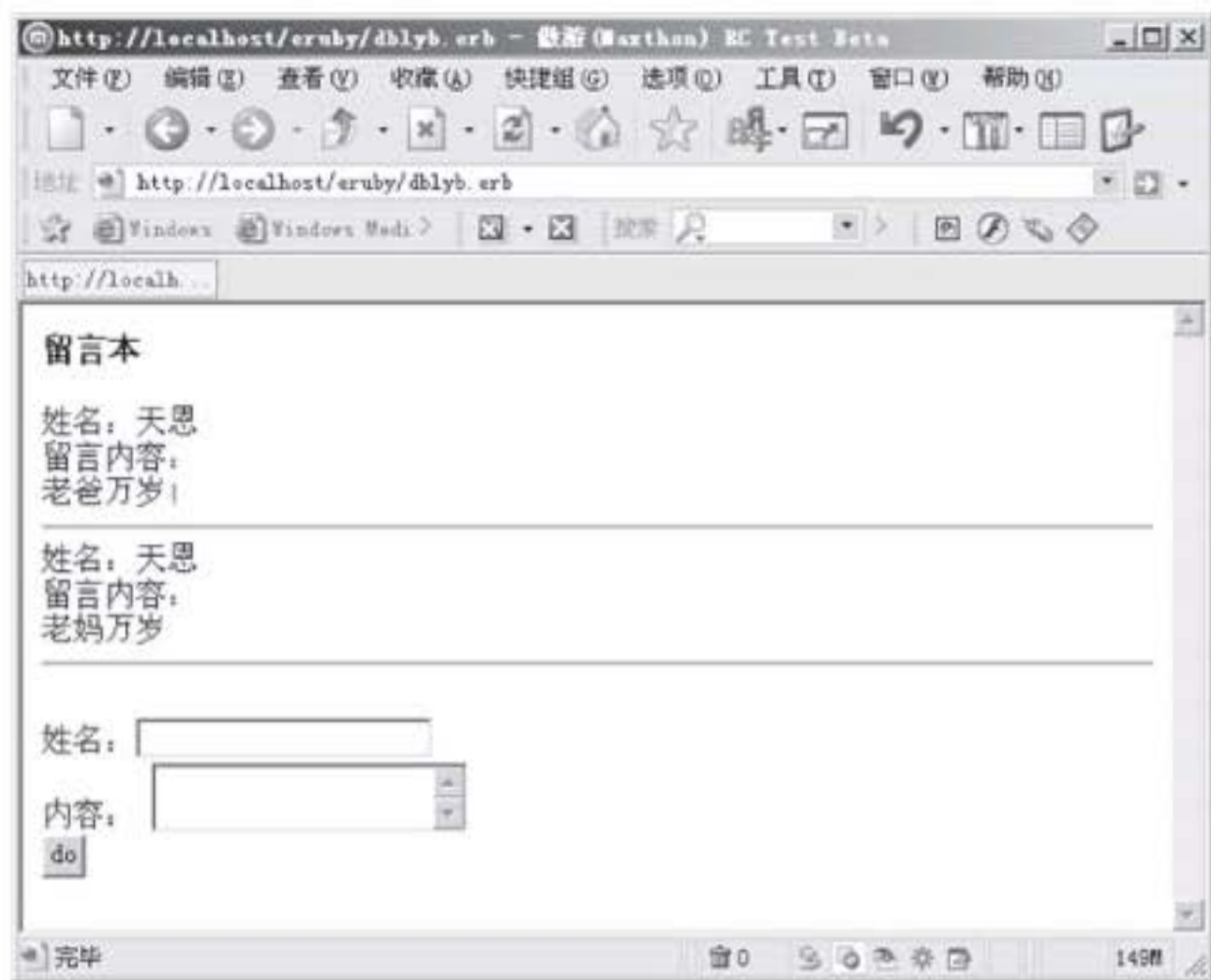


图 7-63 留 言

7.7.4 聊天室(基于数据库)

基于数据库的聊天室,与前面做的聊天室只有一处不同,就是存储介质。相应地,程序代码也有些差异。

使用 MySQL 数据库,在数据库“my”中建立表格,名为“lt”。代码如下:

程序名称:lt.sql

```
use my;
create table lt
(
  xm char(20),
  nr char(50)
);
```

运行结果如图 7-64 所示。

这个案例包含 3 个文件:

dbltup.erb 每 3 秒自动刷新一次,用于读取 lt 数据表的内容,从而及时显示出聊天信息。

dbltdown.erb 是发言文件,用于输入发言内容并提交发言。

dbltmain.erb 是主框架文件,上框架是 dbltup.erb 显示聊天内容,下框架 dbltdown.erb 用于发言。

相关代码如下:

案例名称:聊天室(基于文本文件)

程序名称:dbltup.erb

```
# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
```

```

< h4> 聊天室< /h4>
< meta http-equiv= "refresh" content= "3;url= dbltup.erb"/>
< ! - - 显示原来的内容 - - >
< %
require 'mysql'
con = Mysql.new('localhost','root','tianen','my')
begin
  rs= con.query("select * from lt")
  num= rs.num_rows()
  for i in 0..num- 1
    rs.data_seek(i)
    a= rs.fetch_hash()
    print "姓名:" + a['xm'] + "< br> " + "留言内容:< br> " + a['nr'] + "< hr/> "
  end
  rs.free()
rescue
  print "error"
ensure
  con.close
end
% >
< hr/>

```

```

C:\WINDOWS\system32\cmd.exe - mysql

D:\>mysql -u root -p<d:/ror/lt.sql
Enter password: *****

D:\>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 119 to server version: 5.0.19-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use my;
Database changed
mysql> describe lt;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| xm    | char(20) | YES  |     |         |       |
| nr    | char(50) | YES  |     |         |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.03 sec)

mysql>
搜狗拼音 半:

```

图 7 - 64 聊天室数据库

程序名称:dbltupdown.erb

```

# ! d:/Ruby/bin/eruby.exe - C gbk - Mc - n
Content-type: text/html
< form action= dbltupdown.erb method= post>

```

```

姓名:< input name= "xm">
内容:< input name= "nr" size= 50>
< input type= submit value= "发言"/>
< /form>
< %
  require 'cgi'
  cgi = CGI.new
  if cgi['xm'] && cgi['nr'] then
    if ! cgi['xm'].empty? && ! cgi['nr'].empty? then
      xm= cgi['xm']
      nr= cgi['nr']
      require 'mysql'
      con = Mysql.new('localhost','root','tianen','my')
      begin
        rs= con.query("select * from lt")
        num= rs.num_rows()
        con.query("insert into lt values('" + xm + "','" + nr + "')")
        print "< script> location.href='dbltdown.erb'< /script> "
      rescue
        print "error"
      ensure
        con.close
      end
    else
      print "< script> alert('请填写发言内容')< /script> "
    end
  else
    print "< script> alert('请填写发言内容')< /script> "
  end
% >

```

程序名称:dbltmain.erb

```

< frameset rows= "400,* ">
< frame src= dbltup.erb>
< frame src= dbltdown.erb>
< /frameset>

```

程序最初运行结果如图 7-65 所示。

如果不填写任何内容就点击发言按钮的话,程序将会发出警告。如图 7-66 所示。

现在,我们发言,如图 7-67 和图 7-68 所示。可以发现,发言内容迅速显示出来了。

继续发言,如图 7-69 和图 7-70 所示。

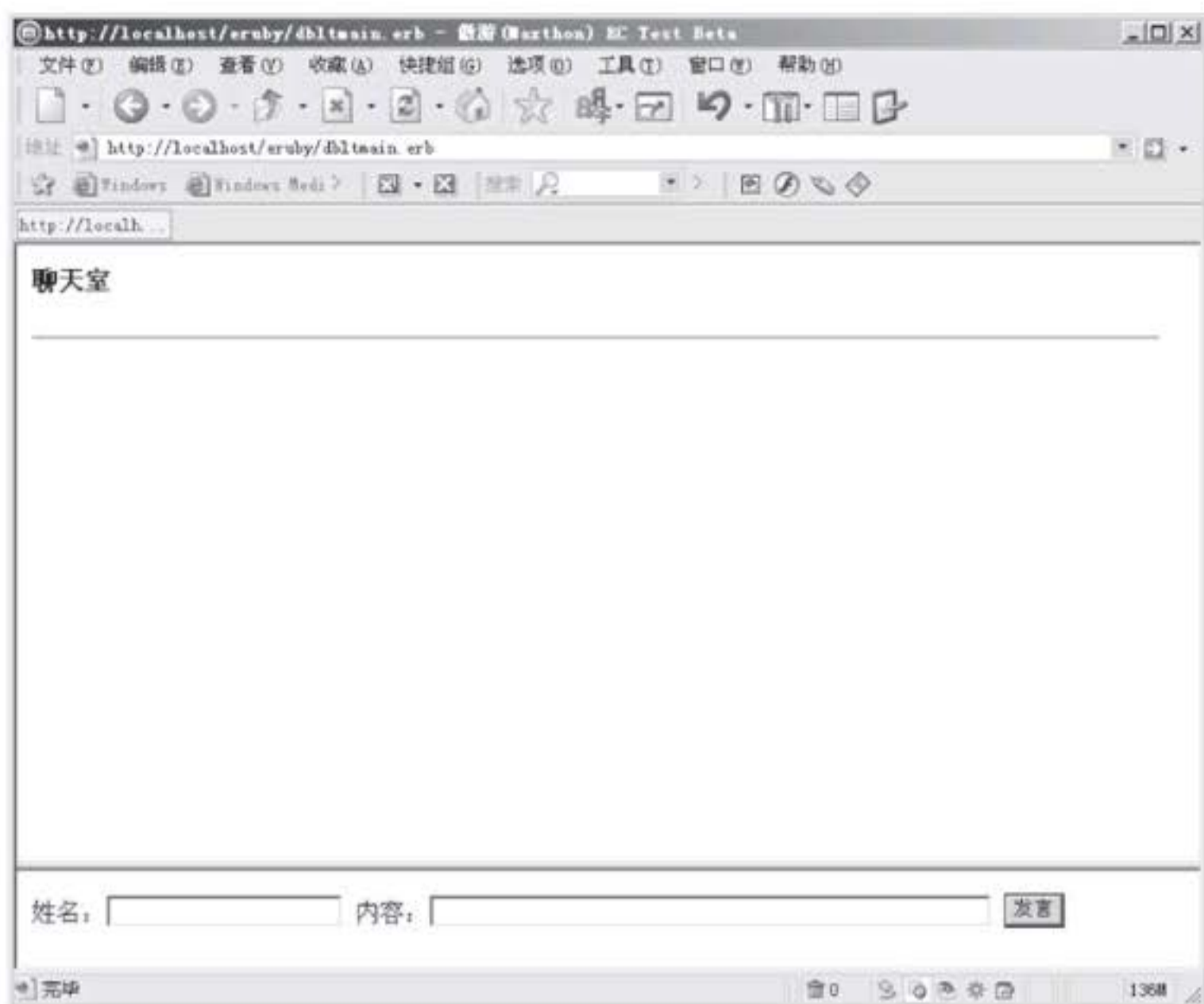


图 7-65 聊天室

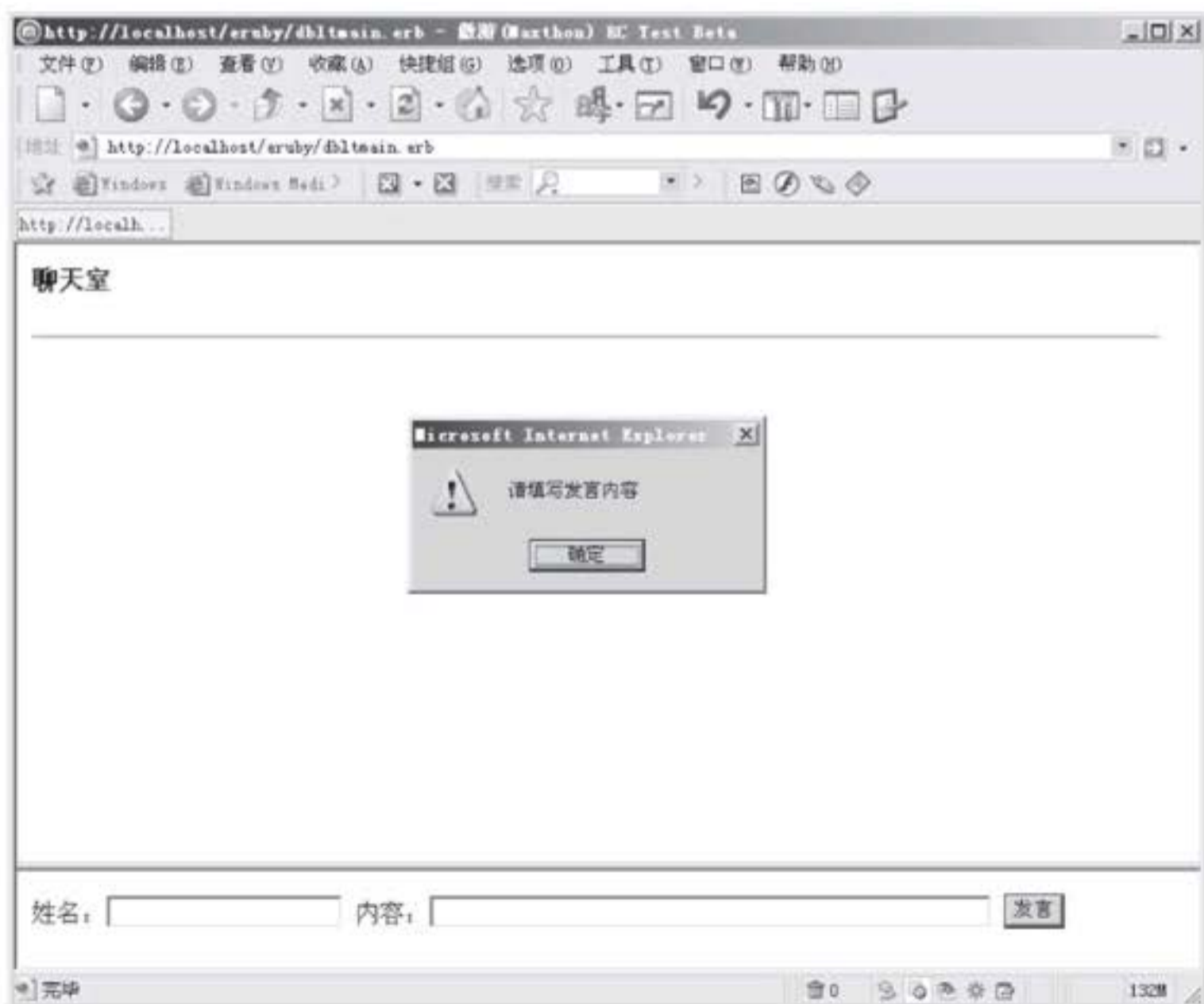


图 7-66 聊天室程序

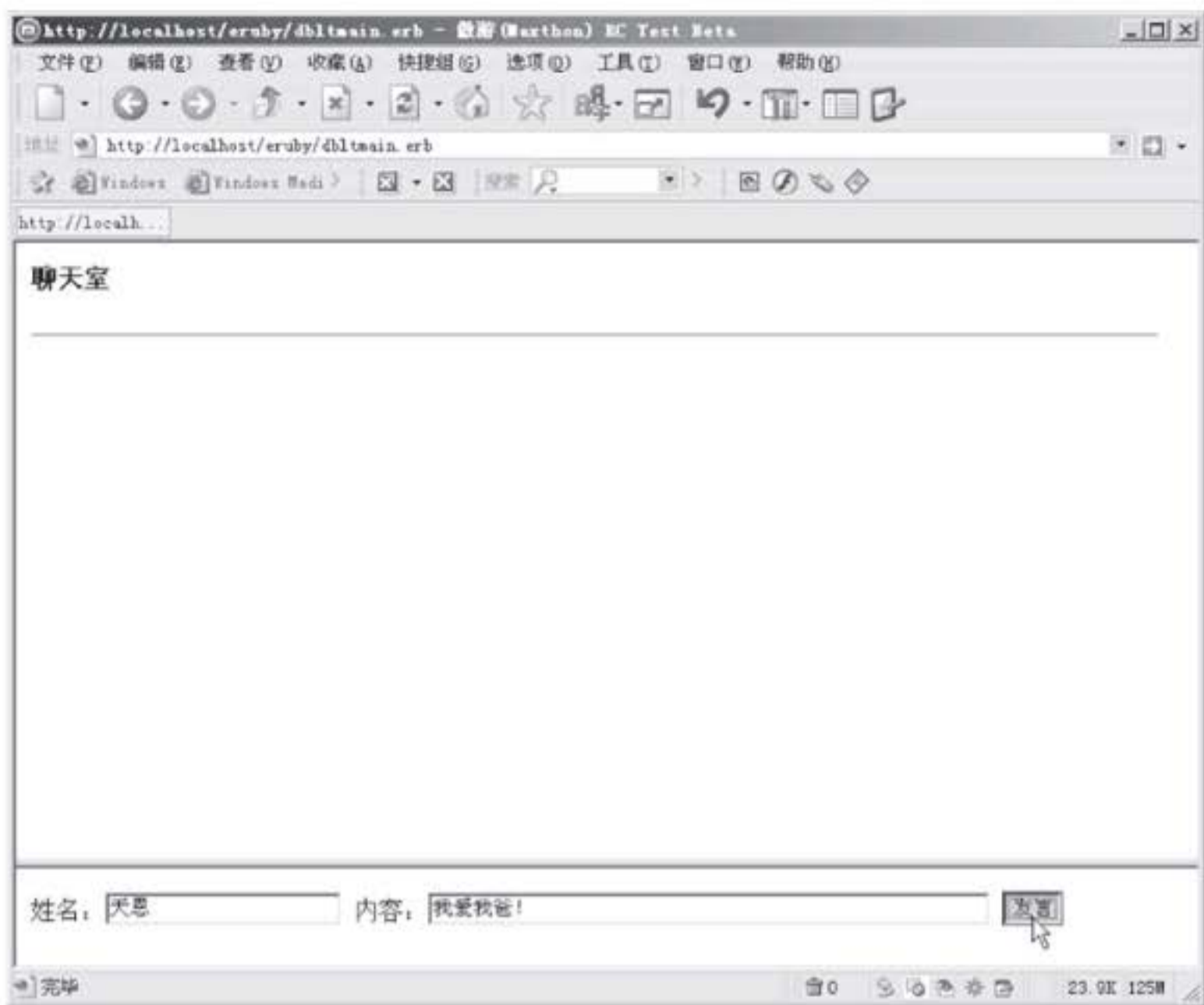


图 7-67 发 言

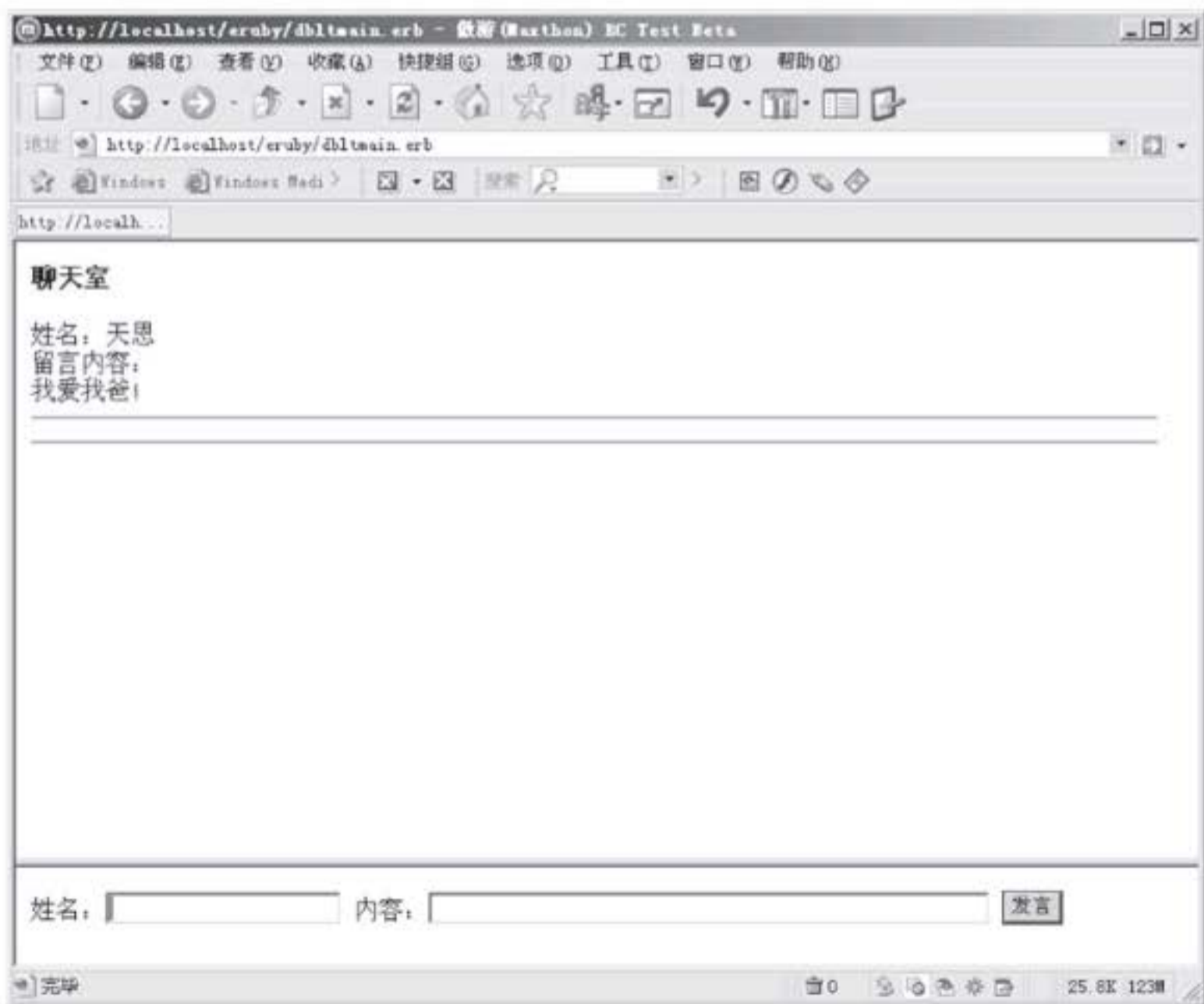


图 7-68 发 言

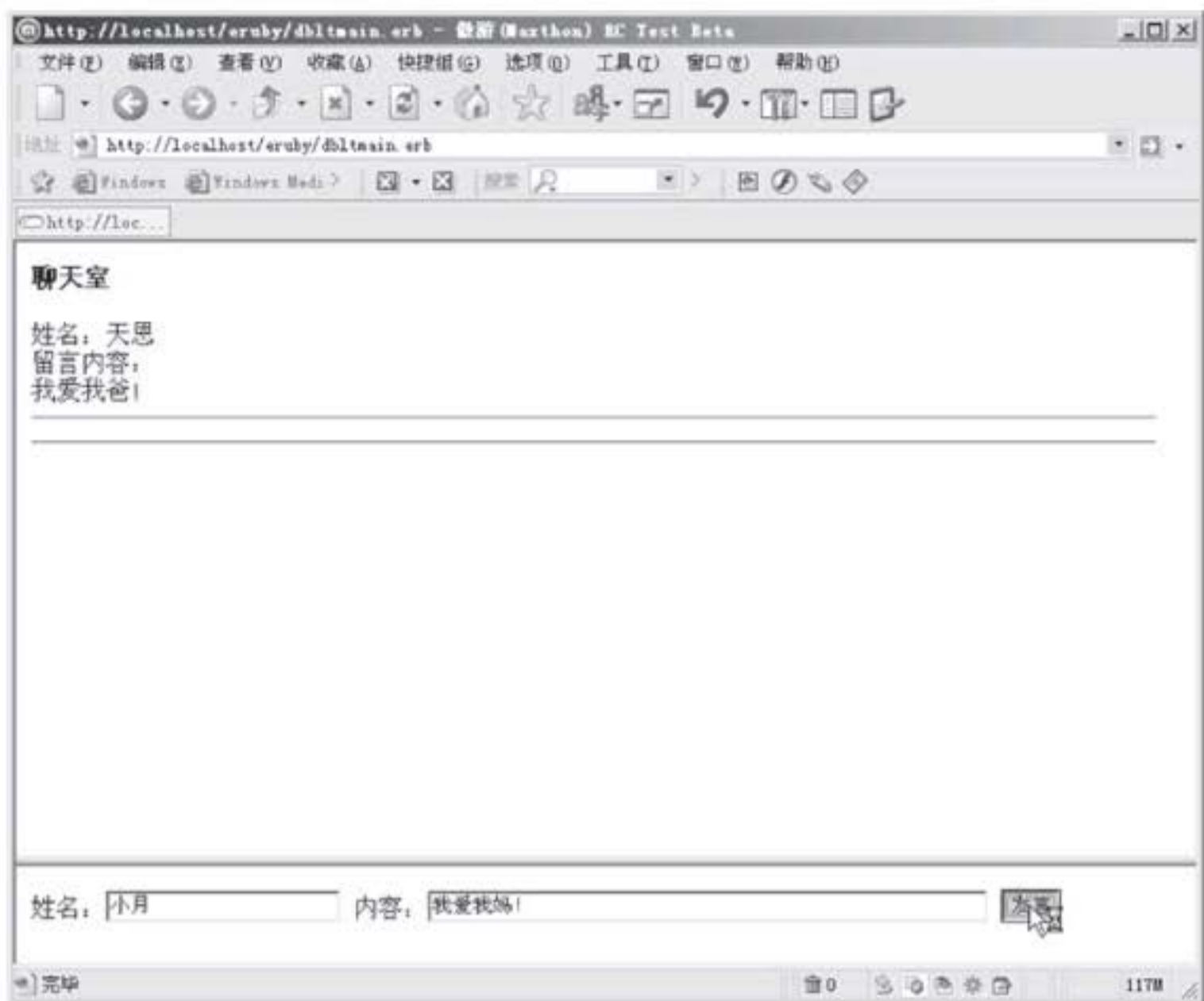


图 7-69 聊天室程序

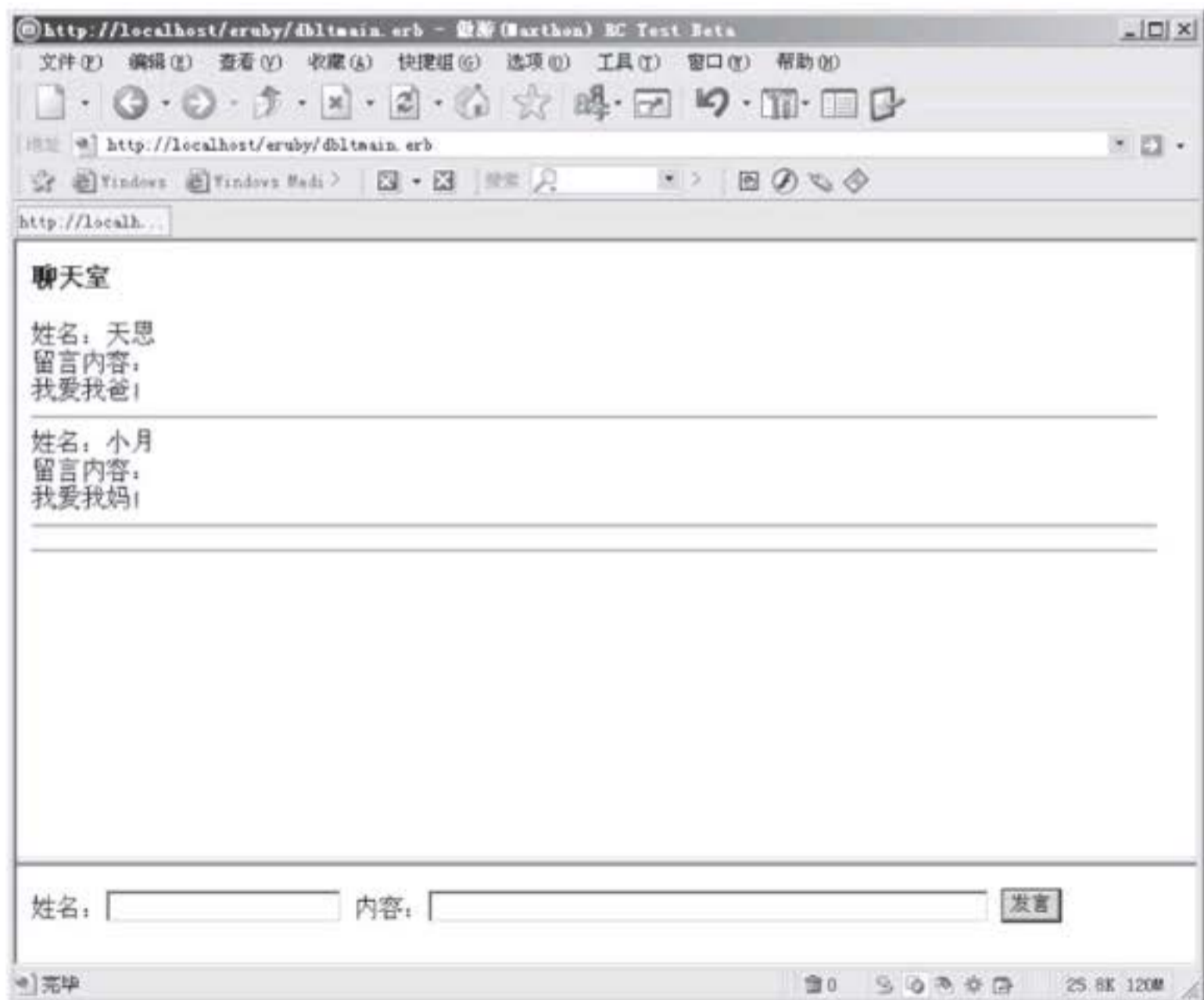


图 7-70 聊天室程序

小 结

本章介绍使用 Ruby 开发桌面应用和 Web 应用的相关知识。需要重点掌握的是 eRuby 的开发方法。时间太少,所以笔者没有细致去讲 eRuby,但也还是介绍了不少东西。有了这些内容,读者就可以进行常规的 Web 开发。若将来能有时间,机会成熟,很希望可以写一些 Ruby 桌面应用和 eRuby Web 开发方面的书。读者若有心,不妨期待。

思考和练习

1. 独立完成一个基于数据库的留言本。
2. 独立完成一个基于数据库的聊天室。