



# TCL、Python 和软件 测试自动化

*TCL、Python 和软件测试自动化*

作者：雷雨后

# TCL、Python 和软件测试自动化

## Credits

题目虽然涉及到了 TCL 和 Python 两种语言，但是实际上目前完成的只有 TCL，并且写的非常的深入。TCL 是一门很奇特的语言，有些地方让你爱不释手，但是一些地方却让你产生失望情绪；他不是一门所谓完美的语言，但是它是一个简单易学，能够大幅度提高你工作效率的语言。

其实这是一本没有写完的学习笔记。开始动手是在 2004 年左右，一直写到 2006 年初。前后持续了一年多的时间。那个时候虽然工作很紧张，但是每天下班后我都会在电脑前写一个小时。后来随着儿子出生，连着一个小时也没有了。虽然一直想全部写完后，要么出版，要么免费共享到网络上，几次努力，发现都难以持续。于是就有了摆在各位面前的这本半成品的电子书。

感谢工作，让我可以学习和了解到 TCL；

感谢儿子，他的出生让我决定将这本电子文档给贡献出来。

## 版权和免责声明

作者雷雨后拥有本书的版权以及后续处理该书的所有权利。

任何其它人和组织可以阅读本书，可以自由传阅本书，但是不可以将本书用于任何商业行为，包括但是不限于：

- 1：自行出版和销售本书谋取利益；这里的出版包括传统纸质出版和电子出版。
- 2：直接抄袭本书本书章节获取利益的行为。

同时作者不保证本书的所有内容的完全正确性。所以本人不对任何因为使用本书的知识或者源代码而导致的任何事故或者损失而承担任何责任。

## About Author

1998 年毕业于上海交通大学计算机及应用专业；

在 Windows 下干过一些年的 C/C++ 软件开发，同时也干过一些年的电信设备自动化测试技术研究，开发过一些自动化测试工具，这期间主要使用 C/C++ 以及 TCL 语言的混合编程。

现在主要从事软件工程以及软件开发工具方面的工作。

本文档没有经过严格的校对，而且由于水平有限，所以错误等在所难免，期待各位读者能够指出。

可以通过 [leiyuhou010@gmail.com](mailto:leiyuhou010@gmail.com) 来联系我。

也可以关注新浪微博：<http://t.sina.com.cn/leiyuhou>



## 前言

当前，作为软件质量保证的一个有效过程，软件测试技术引起了越来越多人和公司的注意，并且获得了长足的进展。在这个领域里面各种思想、方法和工具层出不穷。并且与软件测试相关的咨询、测试工具也形成了一个较大的产业。在业界就有不少的公司提供了许许多多的软件测试工具，这些工具都提供了脚本，用户可以编写脚本来使测试工作自动化，它们的出现一定程度上提高了软件测试的效率。但是很多时候，这些工具并不能够完全满足我们的要求，更多的时候，我们需要开发自己的测试工具，编写自己的自动化测试脚本，来提高测试效率，特别是在针对嵌入式开发的领域。

脚本语言，是不需要编译就能够解释执行的语言。具有语言层次高，开发迅速，易于扩充，方便移植等特点，正是因为这些特点，在软件测试领域里面获得了广泛的应用。基本上现在商用工具都支持某种脚本语言：

表格 0-1 商业工具支持的脚本

工具名称	厂家	支持脚本
QARun		
WinRunner		
Robot	IBM	Basic
SmartBits	思博伦	TCL

自己实现测试工具，编写自动化脚本是一项非常具有挑战性的，但是非常有意义的工作。选择哪些脚本作为我们测试工具的脚本语言，是需要仔细考虑的事情。而 TCL 脚本作为软件测试领域里面最经典的语言，在业界已经获得了广泛的应用。并且 TCL 和其他语言相比，有如下几个重要特点：

- 1) 简单易学。TCL 脚本基于命令和替换的语法结构，对于有 C/C++ 编程经验或者 C Shell 编程经验的而言，非常容易掌握。
- 2) 功能强大，极易扩充。使用 TCL 脚本可以完成很多的事情，包括网络 Socket 编程、GUI 图形界面程序开发、数据库应用程序、实现简单的 WebServer 和 FtpServer、实现 COM 自动化服务器等。我们只需要极少量的 TCL 代码就能够完成需要数十倍 C 代码才能完成的功能。并且该语言的扩展性非常好，你甚至可以扩充出你自己的语言控制结构！比如实现你自己的 switch 结构。
- 3) 面向对象的完美支持。TCL 本身不支持面向对象，但是可以通过程序包的形式来扩展，并且这些扩展包还非常多。不过最好用的就是 ITcl。这个扩展包在 TCL 脚本上实现了对面向对象编程的完美支持。
- 4) 可以非常方便的嵌入其他应用程序。我们可以非常方便的将 TCL 解释器嵌入到我们自己的应用程序中，使我们的程序具有通过脚本来被用户控制的功能。
- 5) 可移植性良好。TCL 发源于 Unix，但是现在几乎在所有的操作系统上都实现了这个脚本解释器，包括 Windows 和 MAC。所以你写的 TCL 脚本基本上不用修改就能够在这些环境上运行。

- 6) 网上资源非常丰富。到这个地方去看看：<http://www.tcl.tk/>。这里是 TCL 的大本营，里面的相关资源丰富的让人窒息。几乎你所想要的扩展包和程序库，都能够在这里找到。更重要的是，这些都是免费的。

在测试领域应用广泛。大名鼎鼎的风河公司（WindRiver）的实时操作系统 VxWorks 的开发环境 Tornado 中就大量使用了 TCL 脚本；TeleLogic 公司的测试工具 Logiscope 中也使用了 TCL 脚本；数据通信测试仪器 SmartBits 的控制端，提供了专门的脚本库，方便用户来控制仪器，进行自动化测试工作。并且业界的很多通信设备制造商都使用 TCL 作为测试语言。

TCL 是一门比较古老的脚本语言。与之相比，Python 则是上个世纪 90 年代出现并且发展起来的脚本语言。Python 的语言风格和 TCL 差异极大，并且和 TCL 没有什么太大的关系。与 TCL 相比，其功能更加强大：

- 1) 对多线程的支持。虽然 TCL 也能够支持多线程，但是实现起来比较复杂；而 Python 则内建了对多线程的支持。
- 2) 运行速度更快。现在已经有了使用 Python 来编写游戏程序的扩展包！大名鼎鼎的 BT 下载软件就是采用 Python 写成。
- 3) 支持 JVM。在 Python 基础之上发展起来的 Jython，能够编译成 JAVA 虚拟机上的中间代码，给 Java 的爱好者另外一个不错的选择！

在测试领域里面，Python 属于后来者，但是也是非常适合做自动化测试的脚本语言。所以本书也会介绍 Python。

## 关于本书

本书重点介绍 TCL、Python 的语法，强调编程注意事项。至于语言的扩展包、函数库，本书不作详细介绍，只是介绍哪些扩展包能够完成哪些功能，能够在什么地方找到这些扩展包。因为在我看来，掌握语言的核心是最重要的，至于扩展包、模块等功能，在网络上都能够找到详细的帮助文档。

同时，本书还会详细的论述软件测试自动化思路，方法，以及如何采用 TCL 或者 Python 来编写自动化测试脚本。

本书首先介绍你所需要的工具，然后逐步深入介绍语言，以及编写高效简洁易于维护的脚本的技巧。希望本书能够成为一本 TCL 和 Python 脚本的入门学习宝典和进一步提高的帮手。但是不希望它成为你编写程序时的参考手册，事实上，TCL、Python 的联机帮助手册更加适合担当这一角色。

本书的重点在于 TCL。对于 Python，本书只介绍语言基础以及如何使用 Python 来进行图形界面的程序设计，其他的模块我们制作简单的介绍。

有人把 MSDN 中的 C++ 或者 MFC 章节翻译成中文，然后成书出版。这样的书我从来不买，因为在使用 Visual C++ 的时候，我会及时直接从 MSDN 中找到我所需要的任何信息。但是像《Inside the C++ Object Model》以及《Effective C++》这样的书籍，我却非常喜欢。我给本书的定位就是《Effective TCL》以及脚本语言在软件测试自动化中的实践！

## 你需要做的

如果你还不能够上网，那么赶快想想办法！作为两种开放源代码的脚本语言，如果没有互联网，是不可能发展到今天如此规模的。同样，作为用户如果不能上网，你会觉得这两门语言学起来是如此的艰难。

如果你有 C/C++ 程序编写经验，那么 TCL/Python 入门对你而言轻而易举，而且你会发现脚本语

言的奇妙无穷。但是如果没有也没有关系，直接学习这两门语言吧，你可以免去很多 C/C++ 程序员的痛苦经历，并且你会发现，C/C++ 能够做的事情基本上他们都能做，并且还做的挺好！

另外一个必须工作就是下载并且安装相关的工具。你需要安装一个 TCL 脚本解释器，安装一个 Python 解释器。同时还需要安装一个功能强劲的程序编辑器。具体有哪些工具可以选择，请参考第一章“安装相关工具”。

## 如何学习一门新的语言

这个问题很重要，如果没有好的学习方法，掌握一门语言就需要更多的时间；但是这个问题却似乎没有唯一答案，下面是我的一点建议：

1. 不要花太多的时间在书上，多动手，当然还是要以先看为前提。
2. 给自己一个明确的目标：“我需要使用这门语言实现一个什么小需求”，然后在动手来编程实现。在解决问题的过程中，你的技术和水平会得到飞速的提高。所以本书中会经常出现这样的例子。

## 本书组织结构

本书在章节组织上的原则是“相关的内容会放到一起”，这样当你阅读的时候，不会有太大的思维上的变化。希望这样的组织和本书的语言风格能够让你在阅读的时候犹如阅读一本不错的小说，能够体会到阅读的乐趣。对于文章而言，没有什么东西比给读者带来阅读快感更重要的了。

### I. 安装相关工具

当你要开始学习 TCL 和 Python 的时候，相关的工具是必不可少的。这一章节介绍了你所需要安装的工具，包括脚本解释器，文本编辑器。以及从网上下载这些工具的地址。这里推荐的脚本解释器是 ActiveTCL 和 ActivePython，编辑器是 Source Insight 和历史悠久的 VIM。

### II. TCL 语言

该部分的第一章节是一个 TCL 快速入门教程，让你在大概一个小时的时间之内了解这门语言最核心的内容。当你在浏览本章节的时候，在电脑上及时地试验一下书中的例子，会对你掌握这门语言有很大的帮助。

本部分的后面的章节是 TCL 语言各方面的详细论述，能够让 TCL 新手迅速了解 TCL 的高级用法和细节。也是老程序员的一个重要参考。

TCL 具有强大的字符串处理能力，这一部分会对该功能进行的讨论，包括赫赫有名的正则表达式。字符串的编码方式，是一个很容易让人迷惑的问题。TCL 采用非常巧妙的方式来处理这一问题。本章节会对 Unicode 等编码方式进行详细的分析，讲解其实质。如果你一直为 Unicode 而困扰，这一章非常适合你！

### III. TCL 中的文件、数据库和操作系统相关操作

同时还会讲述如何使用 TCL 来进行文件系统相关的操作，如何使用 TCL 来操作数据库，进行数据库相关的编程。

### IV. TCL 中网络和 Internet

使用 TCL 脚本可以非常方便的进行 Socket 编程，实现服务器应用程序以及 TCP 客户端应用程序。同时还会讨论如何使用已有的扩展包，来实现 FTP 客户端和服务端，以及如何实现一个简单的 HTTP 客户端。

### V. TCL 中的事件驱动、GUI 和多媒体编程

采用 TCL 来实现事件驱动，编写图形界面的应用程序是一件非常惬意和容易的事情，本部分会



详细讲述。同时如何使用 Tk，或者其他的图形组件库比如 Tix，也是本部分的重点。

## VI. TCL 的高级用法

这里讲述如何使用 C/C++ 来扩展 TCL，也会讲述如何将 TCL 解释器嵌入到其他应用程序中。同时还会讲述 TCL 中进程间通信，多线程程序开发。最后还包括如何使用 TCL 来操作自动化服务器，以及如何使用 TCL 来实现 COM 组件。

## VII. TCL 在线参考资料

TCL 的内核是非常精简的，但是其可扩充性是在是太好了，这直接导致了网络上的 TCL 的程序包应有尽有，本章节会告诉你到什么什么可以找到你所需要的扩展包。

## VIII. Python 语言

这里介绍 Python 语言的基础，该部分的第一节是 Python 语言的快速入门，后面的章节分别对第一章中的各种概念进行详细的讲解。同时会和前面的 TCL 脚本进行对比。有了前面的 TCL 脚本方面的知识，掌握 Python 不会是难事。

## IX. Python 的高级用法

Python 里面怎样编写多线程的程序？怎样来优化 Python 脚本的性能？如何将 Python 解释器嵌入到你的应用程序中？如何使用 C/C++ 来编写 Python 的模块扩展 Python 的功能？如何在 Python 里面操作 COM 组件？这一部分会进行详细的讨论。

## X. Python 中的网络和 Internet

Python 对网络编程支持度比 TCL 要强，程序模块也丰富的多。这里你将会看到如何使用 Python 的模块来实现底层的 Socket 程序设计，以及访问 Ftp-Server 和 Web-Server 等服务器。

## XI. Python 中的 GUI 和多媒体编程

支持 Python 的 GUI 模块非常多，有 TkInter（Tk 在 Python 里面的名字），wxWindow 等。这里重点介绍 wxWindow。这是一个非常优秀的 GUI 程序库。Borland C++BuilderX 就是采用 wxWindow 控件。

## XII. Python 在线参考资料

因为 Internet 的存在，Python 这门语言才能够得到如此迅猛的发展，在各个领域里面得到了广泛的应用。如果你决定采用 Python 来进行软件测试或者其他开发，Internet 是必不可少的。这一部分将详细的介绍在网络上有哪些 Python 资源。

## XIII. 自动化测试脚本

掌握脚本语言语法，不一定就能够写好自动化测试脚本。以测试自动化作为目的的脚本，必须满足一定的条件，遵照一定的设计方法，才能够真正的做到重用和移植，才能够真正的提高测试效率，否则将会适得其反。这一部分将会就这一问题进行重点阐述。

## XIV. 集成测试与自动化脚本

集成测试介于单元测试和系统测试之间，属于“灰盒测试”，能够有效的发现系统中多个模块之间的问题，并且与系统测试相比，可以更加方便问题的定位。通过自动化测试脚本，能够有效的提高集成测试的效率。本部分将详细的介绍集成测试的概念原理，以及如何使用脚本来构造一个继承测试框架。

## XV. 系统测试与自动化脚本

系统测试过程中，也可以通过自动化脚本来使测试工作效率得到明显的提高。对于 GUI 应用程序，可以采用商用测试工具（比如 WinRunner, QARun, Robot），但是这些工具价格昂贵，如果从成本考虑，你完全可以自己采用 TCL 或者 Python 来进行 GUI 应用程序的开发。对于服务器系统，嵌入式系统，采用脚本语言进行自动化测试则是最好的选择。本部分将会对上面两个例子进行详细的论述。

## XVI. 软件可测试性分析与自动化脚本

软件可测试性是对软件系统进行测试、问题定位和远程维护的难易程度。可测试性好的软件，



在测试阶段会使测试人员工作得更加轻松，定位问题更加容易。通过一定的方法来进行软件可测试性分析和设计，可以提高软件的可测试性！这一章节将会介绍软件可测试性分析、设计和实现的方法。显然，可测试性好的软件，进行测试自动化工作会更加荣誉。

## XVII. 构建你自己的自动化测试框架

自动化测试的目的，是提高测试的效率，完成手工测试不能够完成的任务。效率的提高不仅仅是编写脚本这么简单，从你决定要自动化测试的时候开始，所有的工作包括测试用例设计，测试脚本编写调试维护，测试用例执行都要围绕提高效率的目的来进行。一个好的测试框架是自动化能够真正收到效果的有效保证。本部分将会详细的讨论如何构建一个好的自动化测试框架（你也可以称之为测试工具）。

## 本书的习惯记法

本书中所有的源代码都采用灰色底纹，其中代码注释采用斜体字。如下所示：



```
#注释。下面的代码输出 Hello,World
package require Itcl
::itcl::class ExampleClass {
    public proc Main {} {
        puts "Hello,World"
    }
}

ExampleClass::Main
```

代码的运行结果采用黑体字和灰色底纹，如下：

```
Hello,World
```

## 图标含义

	表示特性是语言的最新版本中出现的，如果是旧版本，可能就不支持！
	使用提示和技巧。如果采用这里介绍的方法，可能会使你的效率有较大的提高。

## 和我联系

Email: [leiyuhou010@hotmail.com](mailto:leiyuhou010@hotmail.com)

Sina 微博: <http://t.sina.com.cn/leiyuhou>

## 安装相关工具

TCL 和 Python 都是开放源代码的语言，在网络上有非常多与之相关的开发工具。本部分给大家介绍几款我常用的开发工具，至少我对他们比较有好感。当然如果你采用其他的开发工具，本书的内容同样适合你。

使用浏览器打开网址 <http://www.activestate.com>，你就来到了 ActiveState 公司的主页。该公司主要致力于开放源代码脚本语言的二次开发封装，同时也开发自己的开发工具。我们使用的 TCL 解释器和 Python 解释器就从这里下载。本书中所有的工具都是 Windows 版本，当然大家也可以使用 Linux，FreeBSD 等操作系统，ActiveState 公司为这些工具都提供了其他操作系统上的版本。

## 安装 TCL 解释器

ActiveState 公司提供多种 TCL 开发工具套件，ActiveTCL 就是其中之一。该工具由 ActiveState 公司免费提供下载，免费使用。ActiveState 同时还提供了另外一款 TCL 开发工具 Tcl Dev Kit，这一款功能更加强大，不过是需要收费的。下面是两款工具的对比：

	ActiveTCL	TCL Dev Kit
下载地址	<a href="http://www.activestate.com/Products/ActiveTcl">http://www.activestate.com/Products/ActiveTcl</a>	<a href="http://www.activestate.com/Products/Tcl_Dev_Kit">http://www.activestate.com/Products/Tcl_Dev_Kit</a>
版权	免费	免费下载，需要购买使用许可
脚本调试器	不支持	支持
打包器	不支持	支持
代码覆盖率分析	不支持	支持
代码加密	不支持	支持
代码性能分析	不支持	支持
静态代码分析	不支持	支持
将 TCL 脚本转换为 Windows 服务	不支持	支持
浏览器插件	不支持	支持
支持程序包	两者大概一致	两者大概一致

安装完毕之后，安装目录下的 bin 子目录中都带有可执行程序 tclsh.exe，它是一个基于命令行的程序，可以工作在交互式状态下。如果在命令提示符下输入：tclsh.exe filename。那么 Windows 就会启动脚本解释器程序，脚本解释器就会马上执行后面参数 filename 中指定的脚本文件名。执行结束之后解释器就马上退出。

如果在命令行提示符下面只输入 tclsh.exe，那么 Windows 就会启动脚本解释器，然后脚本解释器就会显示一个百分号提示符，等待用户输入 tcl 命令。用户输入一条完整的命令，回车结束，那么解释器就会执行用户输入的语言命令，显示结果。有些时候你输入的命令需要换行，没关系，回车之后解释器会等待你把命令输入完整。

下图非常清楚地说明了上面的两种情况：

```

C:\WINDOWS\System32\cmd.exe - tclsh

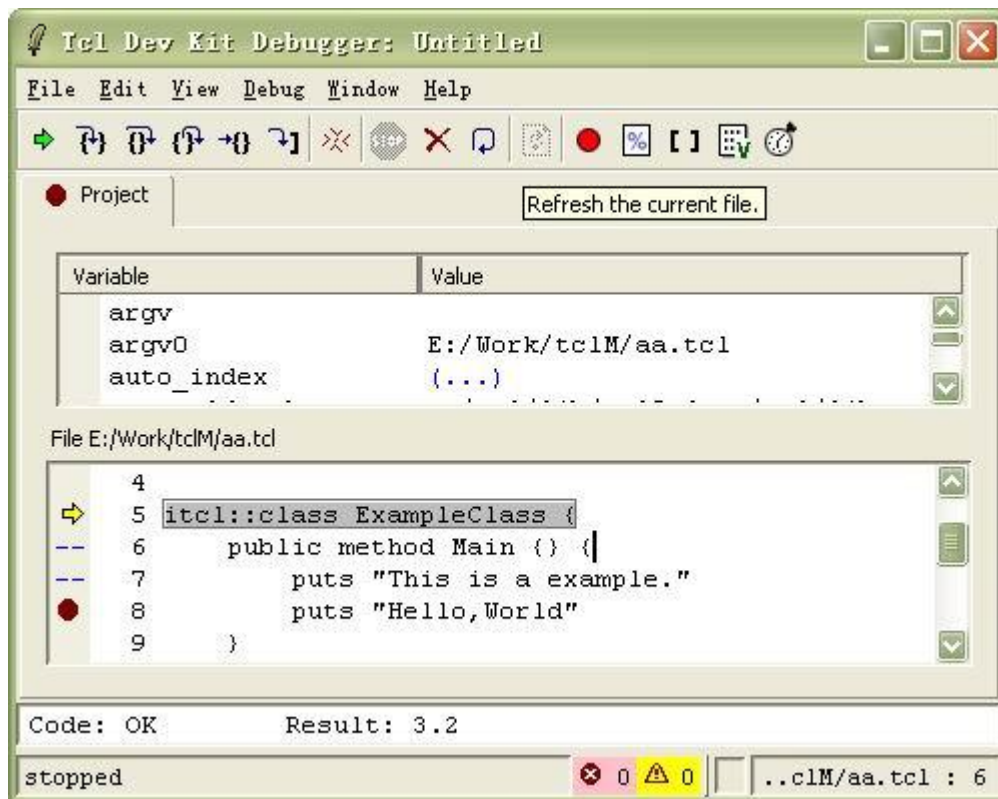
E:\Work\tclM>tclsh.exe ./aa.tcl
This is a example.
Hello,World

E:\Work\tclM>tclsh
% set a 100
100
% if <$a>=100> {
puts "hello"
}
hello
% puts $a
100
%

```

除了 `tclsh.exe` 之外，安装程序还提供了 `wish.exe`。这也是一个交互式应用程序，和 `tclsh` 差别不大，只不过是图形界面，并且帮我们加载了 Tk 程序包。如果我们的脚本使用了 Tk，一般情况下就使用 `wish` 来作为脚本解释器。

两个工具的语言内核完全一样，只不过 TCL Dev Kit 增加了一些额外的工具。这其中最有用的大概就是脚本语言调试器了，只在 Tcl Dev Kit 版本中提供。其界面如下：



调试器的用法和一般的调试器类似，具体如何使用，本书不做详细介绍。

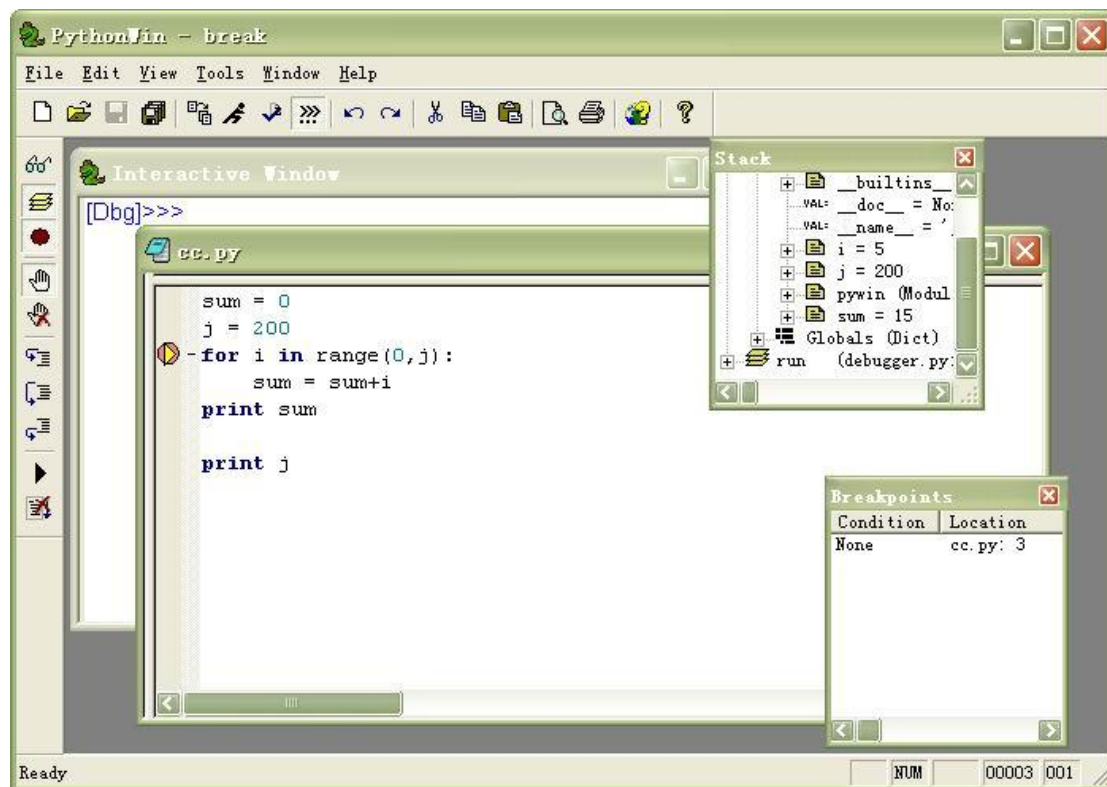
两个工具中都带有一个小程序 `tkcon`，这是用 TCL 脚本开发的一个小工具，实现了一个图形界面的，交互式的 TCL 命令输入和解释执行环境，使用起来它比基于字符界面的解释器 `tclsh.exe` 要方便的多。安装完毕之后能够在启动菜单发现他的存在。该工具为我们学习 TCL 提供了很大的便利。

## 安装 Python 解释器

网络上 Python 脚本的解释器非常多, <http://www.python.org> 网站是 Python 的官方网站, 你可以到这里下载源代码或者下载已经编译好的版本。但是这里推荐的是 ActiveState 的发行版本: ActivePython。

## 安装 ActivePython

登录到 <http://www.activestate.com/Products/Activepython/>, 就可以下载 ActiveState 公司提供的免费 Python 开发工具 ActivePython, 作者现在所用的版本是 2.3.2。这是一个完全免费的软件。安装完毕之后可以在程序菜单中看到专门为 Windows 操作系统开发的 PythonWin, 它是一个不错的 IDE (集成开发环境), 集成了代码编写, 调试, 运行等功能。同时还集成了 Python 模块浏览的功能。该工具界面如下:



和 TCL 解释器类似, 安装完毕之后, 在安装目录下会存在两个可执行程序: python.exe 和 Pythonw.exe。前者的功能和用法与 tclsh.exe 类似, 既可以工作在交互式模式下, 也可以将脚本文件名作为命令行参数传递给 python.exe。而后者 pythonw.exe 则是非控制台的 Win32 程序, 它只能够从命令行参数中得到脚本文件名, 然后执行这个脚本, 执行结束之后就马上退出。

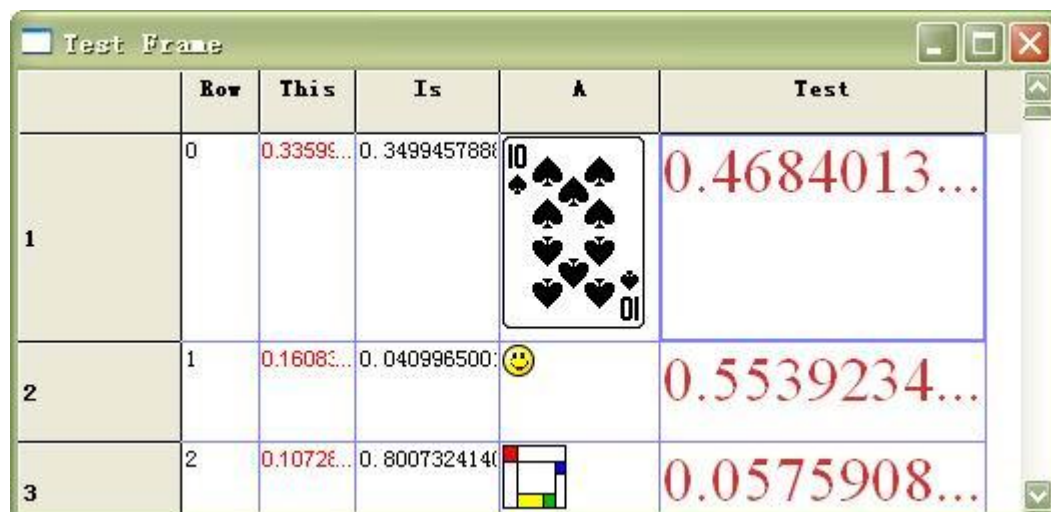
读者可能会问了: 它和 python.exe filename 这种方式有何差别? 答案是“PythonW.exe 没有标准输入输出, python.exe 总会显示一个控制台窗口”, 一般情况下:

1. 运行那些使用了 GUI 图形元素、同时也没有使用标准输入输出的脚本, 就采用 PythonW。
2. 运行使用了标准输入输出的脚本 (不管有没有 GUI), 就采用 python;
3. 所有的脚本都能够使用 python.exe 来运行, 但是并不是所有的脚本都能够使用 pythonW.exe 来运行并且得到正确结果。

ActivePython 软件包中包含了 Tkinter 模块, 这是 Tk 的 Python 版本。如果你对采用 TCL/Tk 编写 GUI 程序比较熟悉, 那么使用 Python 和 Tkinter 来进行 GUI 编程也是一件很容易的事情。除了 Tkinter 之外, 现在有一个功能更加强劲的 GUI 开发包, 这就是 wxPython。

## 安装 wxPython

先来看看 wxPython 提供的一个表格控件!



如果你曾经是 MFC 程序员, 那么你可能还记得寻找一个强大并且易用的 Grid 控件是多么的困难。可是 wxPython 里面, 表格控件的强大功能一定会给你一个惊喜, 其易用性会让你爱不释手。

wxPython 是 GUI 程序库 wxWindow 移植到 python 上的版本。现在最新的 C++BuilderX 也是采用 wxWindow 作为 GUI 程序库。能够得到 Borland 公司的青睐, 足以说明这个程序库的实力。

在安装 wxPython 之前, 首先必须安装好了 Python 解释器, 在这里是 ActivePython。然后到 <http://sourceforge.net/projects/wxpython/> 去下载 wxPython。要注意的是: 你所下载的 wxPython 的版本一定要和 Python 解释器的内核版本的版本相匹配。否则你的 wxPython 可能就没有办法正常工作。

怎样知道你的 Python 解释器的内核版本? 启动 Python 解释器就能看到了, 例如:

```
C:\>python
ActivePython 2.3.2 Build 232 (ActiveState Corp.) based on
Python 2.3.2 (#49, Nov 13 2003, 10:34:54) [MSC v.1200 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

可以看到作者的 Python 是 2.3.2 版本的, 在前面指定地址列出的一系列下载文件名中, wxPythonWIN32-2.5.1.5-Py23.exe 文件就是和作者解释器相匹配的版本。其中的 2.5.1.5 是 wxPython 的版本号, 后面的 Py23 表示为 Python 2.3.\* 所 Build 的。如果你使用的是 Windows XP、Windows 2000 或者 NT, 那么建议下载的版本是 wxPythonWIN32-2.5.1.5u-Py23.exe, 其中的 2.5.1.5u 表示这是一个 Unicode 版本。这是因为 NT 操作系统内部都是采用 Unicode 构造的, wxPython 的 Unicode 版本在 NT 上运行起来会快一些。

wxPython 的安装路径和你的 Python 解释器安装路径相关: 它是作为模块的形式安装到你的 Python 解释器当中的, 例如我的 Python 安装在 c:\python23 目录, 那么 wxPython 就安装在 C:\Python23\Lib\site-packages\wxPython 目录中。

wxPython 安装之后, 你可以看到 wxPython 程序组里面的“Run the wxPython Demo”, 运行这个

程序，你就可以看到所有 wxPython 的使用例子和对应的源代码。同时安装程序也安装了 wxPython 的联机帮助手册。基本上我就是通过这两个东西学会 wxPython 的。相信对大家来说也不困难。而且 wxPython 程序库的结构非常好，学起来也是非常容易的。

## 被包含的工具



## 编辑器

前面介绍的都是脚本解释器，是用来执行我们编写好的脚本的，不附带脚本编辑功能。一个强大的脚本编辑器，会大大提高我们编写脚本的效率。用过 Visual C++，Delphi 的可能会对集成开发环境的强大功能和使用方便性记忆犹新。事实上，网络上也有很多专门为 TCL，Python 等脚本语言开发的集成开发环境，但是为了使大家对脚本编写有更加深入的认识，对脚本的组织，开发过程有更加深入的体会，我推荐大家直接使用程序文本编辑器来编写脚本文件，然后通过解释器来运行。下面就推荐两个我经常使用的脚本编辑器。

## Source Insight

Source Insight 是一款功能强大的、专门为程序员设计的程序编辑器，当前最新版本是 3.5。可以从如下的网址下载：<http://www.sourceinsight.com/>。它是一款商业软件，需要购买使用许可。但是其诸多比较杰出的功能是值得付出的：

1. 通过 Project 的方式来组织源程序；方便代码文件的组织。
2. 支持多种编程语言。能够对代码进行语法分析，并且可以非常方便的查找到标识符的定义位置，被引用情况。这是 Source Insight 最引以为傲的功能！
3. 支持的语言类型可以被定制，被扩展。Source Insight 本身并不提供对 python 和 tcl 的语法支持，但是你可以从作者网站上下载到相关的语法定义文件。
4. Source Insight 提供宏语言，用户可以自己编写宏脚本来扩展功能。
5. 支持外挂命令。下面我们就会把 Python 解释器和 Source Insight 连接起来，使用户在 Source Insight 里面就能够运行 Python 脚本。

我们看看如何让 Source Insight 支持 Python 语法：

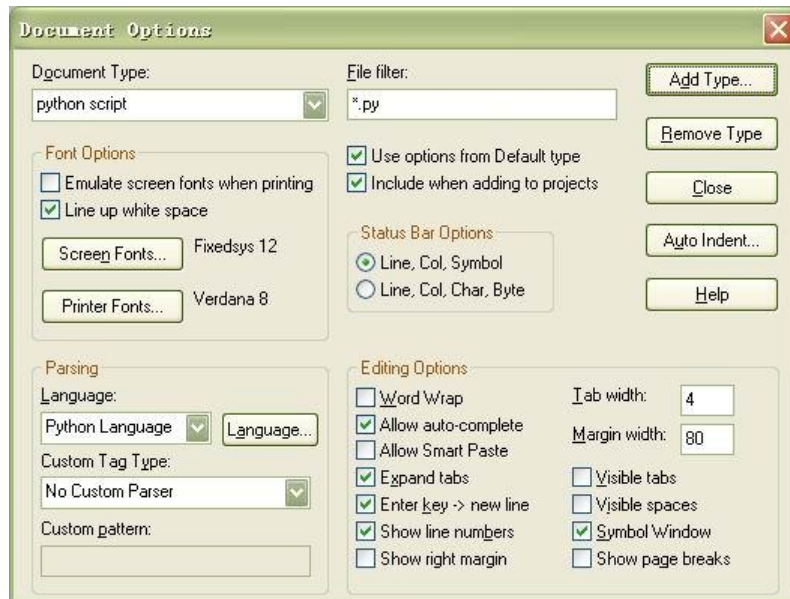
首先从作者网站上下载 Source Insight 支持 python 的语法文件 Python.CLF，将该文件复制到 Source Insight 安装目录下。



启动 SourceInsight，选择菜单“Options->Preferences”。在 Preferences 对话框中选择“Language”

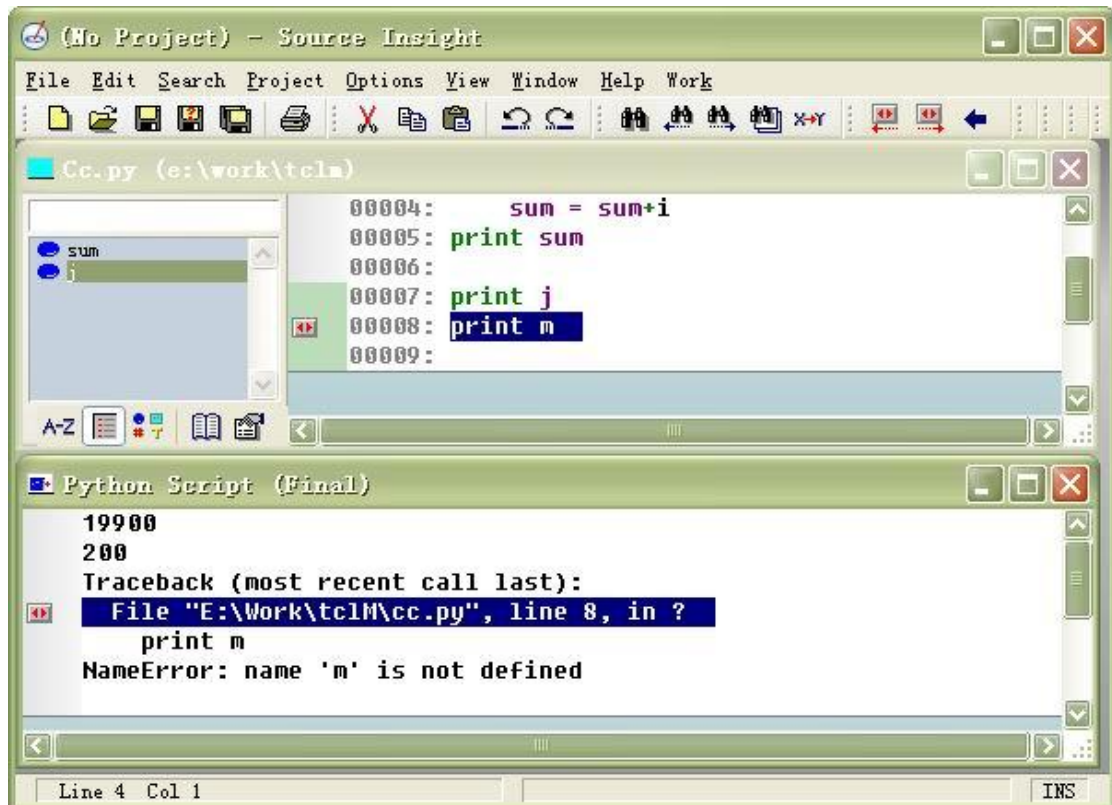
这一页。选择按钮“Import”，在弹出的对话框中选择刚才的 python.CLF。这时的界面如下所示，Language 列表中应该出现“Python Language”。

然后在上面的窗口中选择按钮“Doc Types...”，弹出下面的对话框：



在上面的对话框中选择“Add Type...”按钮，输入新增加的语言类型名字“python script”，然后在“File filter”中输入\*.py，表示所有的\*.py 文件都是 python 脚本。然后在“Language”下拉列表框中选择“Python Language”，至于对话框中其他选项，可以参考上面的设置。完成这些设置之后，你的 Source Insight 就可以完全支持 Python 脚本语言的语法了。

用户可以定制 Source Insight，直接在其中运行窗口中的 Python 脚本。并且可以捕获 Python 脚本执行结果，如果脚本中存在语法错误，Source Insight 可以根据你所指定的解析规则自动的找出你程序中出错的地方。这个功能可是非常的诱人，如下：



图中 Source Insight 的两个子窗口，上面是 cc.Py，下面是运行 cc.Py 之后的结果，可以看到该文件的第 8 行出现了错误。Source Insight 可以自动的根据错误信息帮你快速定位到出现错误的位置。选择 Source Insight 的菜单“Options”->“Cumtorn Command”，就可以加入定制的命令：



可以通过上面对话框中的“Add”增加我们自己的命令名字“Python Script”，其他选项按照上图一一设置，其中要注意的是“Pattern”，这里设置为“^\\s+File \"(.+\\.)\", line \\([0-9]+\\)”，这是一个正则表达式，用来告诉 Source Insight 如何从脚本执行结果中解析出出错文件名和出错的行号。什么是正则表达式，后面的 TCL 语言中会有详细的介绍。

通过上面对话框中的“Menu”按钮为我们的命令增加对应的菜单；通过“Keys”来设置对应的快捷键。我这里设置的是“Ctrl+F12”。

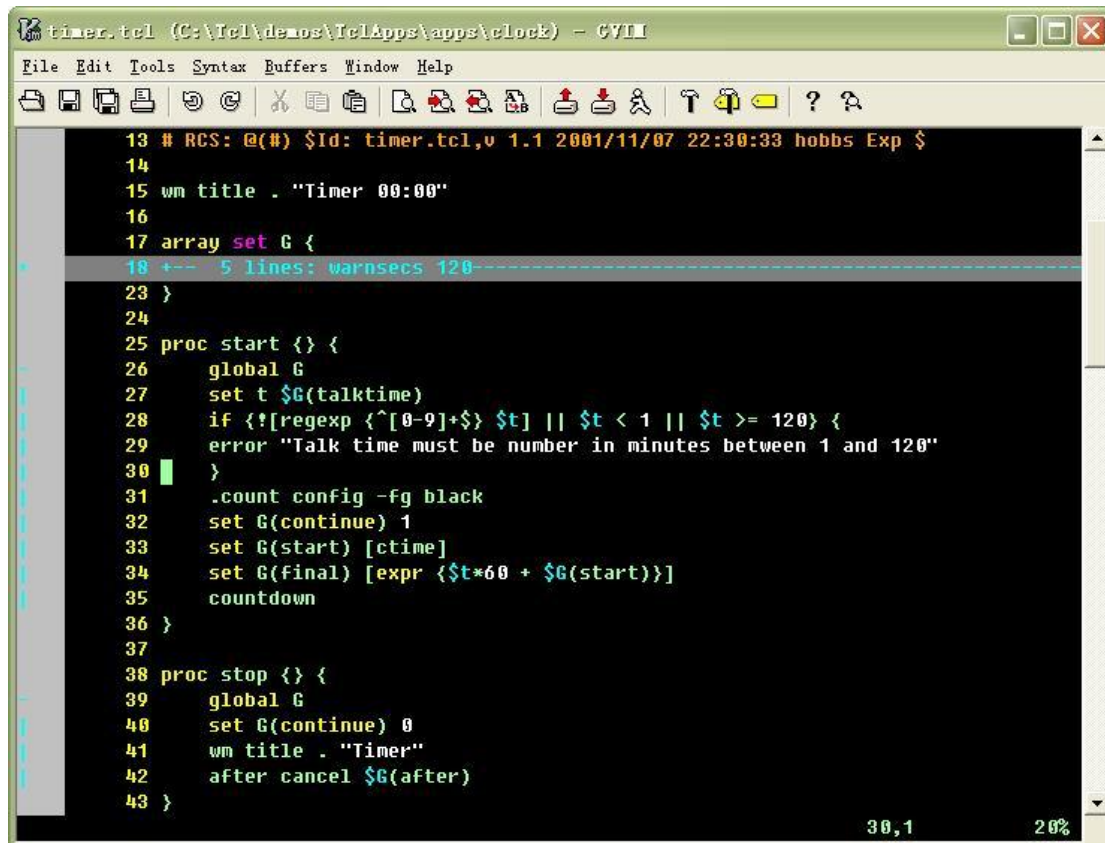
对于 TCL 脚本，设置方法和上面介绍的设置 Python 过程完全类似。

## gVim

VI 是一个起源于 Unix 操作系统的文本编辑器，gVim 是其 GUI 升级版本，已经移植到了现在几乎所有的操作系统上，并且是免费的。虽然免费，但并不影响其功能强大。这个编辑器和我们常见的文本编辑器（比如 UltraEdit）有很大的不同，要熟悉和掌握它的使用方法和技巧，需要花一定的时间和精力，但是一旦入门和熟悉之后，每次使用，几乎都能够让你找到一些惊喜：原来以前很多繁琐的功能在这里可以如此轻松的解决！

在网络里面存在一些热心的人们，他们正在把 vim 的使用手册翻译成中文，你可以到下面的地址里找到相关信息：<http://vcd.cosoft.org.cn/pwiki/index.php>。这里的中文帮助能够让你快速入门和熟悉 gVim，强烈建议你去看看。下面只介绍一下使用 gVim 的基本功能。

到 <http://www.vim.org> 下载最新的 gVim，直接安装后就可以运行。作者安装的是 6.2 版本，运行之后，打开一个 TCL 脚本，界面如下：



gVim 工作在几种模式下：插入模式、命令模式、可视模式等。几种模式可以互相转换。比如命令模式下输入 i, a, o 等命令就会进入插入模式，用户在插入模式下可以输入新的文本。插入模式下按 Esc 就进入命令模式。命令模式下用户输入的是 vim 操作命令。

1. gVim 支持丰富的光标移动，屏幕滚动命令；
2. gVim 支持的编辑功能异常丰富：插入，删除，拷贝，粘贴等；
3. gVim 支持颜色定制；输入命令:colorscheme murphy，就能够设置成 murphy 颜色模式，你可以设置成其他你喜欢的颜色模式。
4. gVim 支持多种语言的语法高亮度显示；包括 TCL 和 Python 脚本。
5. gVim 支持文件内容折叠显示。这对浏览较大的程序代码非常有帮助；比如上图中，代码 array set G {...} 中多行代码就被折叠起来。gVim 支持多种折叠方式，支持多种折叠。
6. 因为正则表达式的使用，gVim 支持强大的查找和替换功能。例如命令模式下在某一个字符串下面按下星号 “\*”，就开始前向查找该字符串。
7. gVim 支持同时编辑多个文件，支持在多个窗口中编辑同一个文件。
8. gVim 支持目录浏览和切换，文件打开功能。cd 命令用来改变目录，pwd 显示当前目录，edit <dir> 用来打开该目录，让你选择需要编辑的文件，并且可以按照各种属性排序。功能强大无比。
9. 通过和 ctags 程序结合，gVim 可以非常方便的在多个程序文件中查找某个函数或者变量标识符的定义。ctags 也是公开源代码的程序，支持多个操作系统。
10. gVim 是为程序员设计的编辑器，使用熟了，会给你的代码编写带来极大的便利。基本上代码编写过程中，可以不使用鼠标。对于某些编程高手而言，让手离开键盘去操控鼠标是不可忍受的行为。如果你属于这一类，那么恭喜你，gVim 就是为你而作的。

gVim 的使用方法本身可以编写成一本书。网络里面已经有相当多的资料，这里就不详细的介绍了。

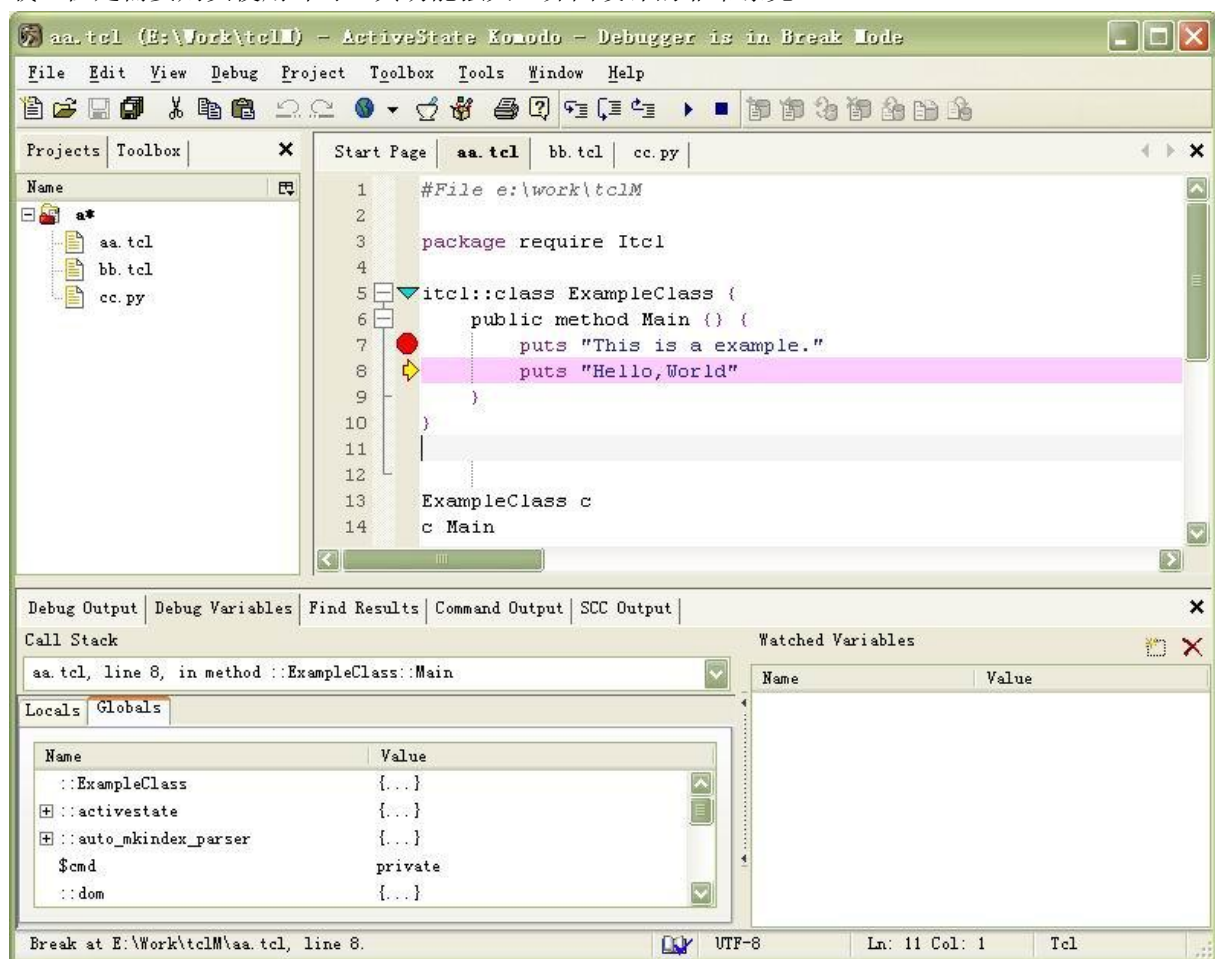


## 其他的工具

除了刚才我们介绍的解释器和文本编辑器，在网络上存在众多的针对脚本语言开发的工具。经常到 <http://www.tcl.tk> 和 <http://www.python.org> 这两个网站上去看看，基本上所有的新工具在这里都有简介和链接。下面列举其中一些做一个简单的介绍，读者如果有兴趣可以自行研究和探索。如果能够熟练使用这些工具，肯定会给你的工作带来极大的便利。

### Komodo

这是 ActiveState 公司开发的一款集成开发环境，是一款商业软件，可以到其公司主页上面去下载，但是需要购买使用许可。其功能强大，界面设计的非常漂亮：



该工具是跨平台的，支持多种语言（TCL 和 Python 肯定是支持的了）的调试以及远程调试。缺点就是占用资源太大了，而且对中文支持的不够理想。如果能够解决这些问题，将是一个非常好的 IDE。它的风格和 Microsoft Visual Studio.NET 的集成开发环境比较类似。

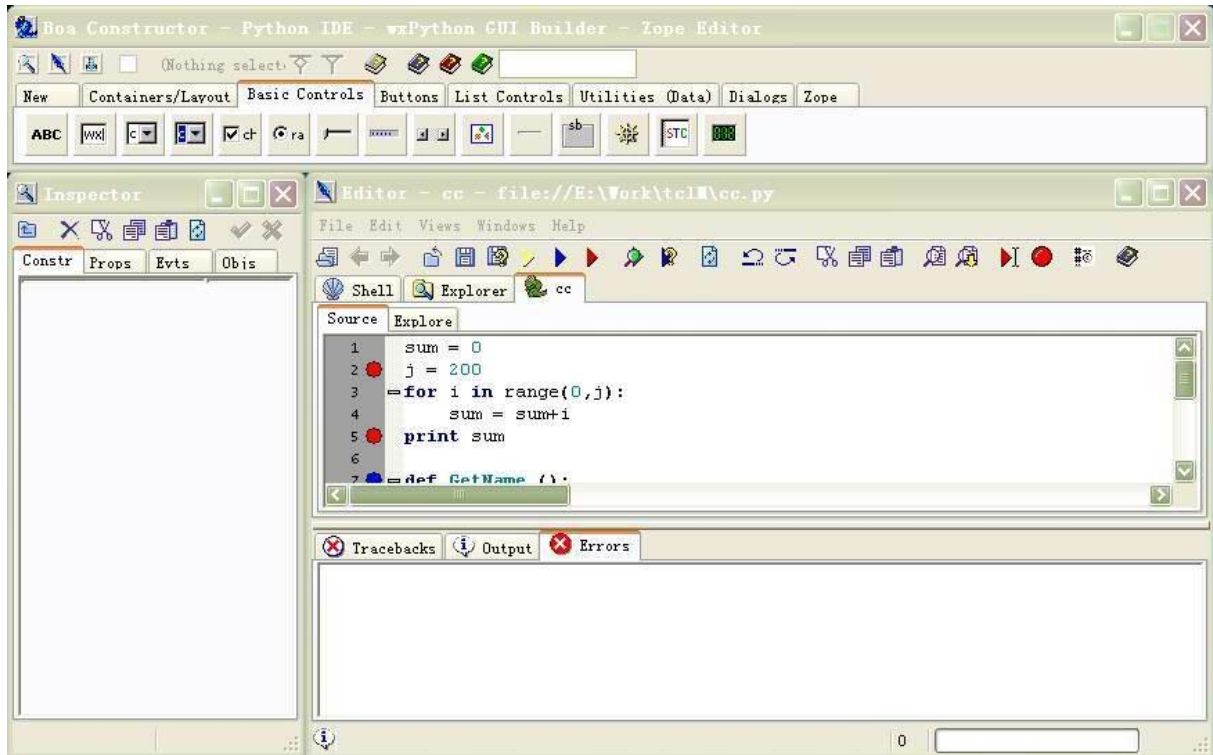
### boa Constructor

如果你用过 Delphi，肯定对其 IDE 环境有着深刻的记忆。boa Constructor 就是针对 Python 语言

开发的 Delphi。它本身就是采用 Python 开发的，所以跨平台。而且，它是开放源代码并且完全免费的！赶快去下载一个回来体会一下吧。下载地址：<http://boa-constructor.sourceforge.net>。

安装 boa Constructor 之前，必须先安装 Python 解释器和 wxPython，Python 解释器的版本应该在 2.1 及以上，wxPython 的版本应该在 2.4.0.7 及以上，否则 boa Constructor 不能工作。假设 Python 安装在 C:\Python23 目录下，安装完成之后在 C:\Python23\Lib\site-packages\wxPython\tools\boa 目录下会有一个文件 Boa.py 文件，使用 pythonw.exe 来运行这个程序，就启动了 Boa Constructor 了。

下面是 boa 的工作界面，是不是和 Delphi 极其类似？



boa Constructor 是一款很不错的 IDE，包含的功能相当多：

1. 所见即所得的图形界面程序开发；
2. 集成的程序代码编辑器和调试器；
3. 种类丰富的控件；
4. 集成的 Python 命令行窗口；

这个工具我认为值得尝试。特此推荐！

## wingIDE

wingIDE 是另一款针对 Python 的集成开发环境，并且在 Python 领域里面有着极高的评价：

*"The best Python IDE currently available."*

—— Stephen Scherer, Ed.D., Jan 2004

*"Zope debugging is an absolute breeze."*

——Mark Jeacocks, Advanced Data Integration, Sep 2002.

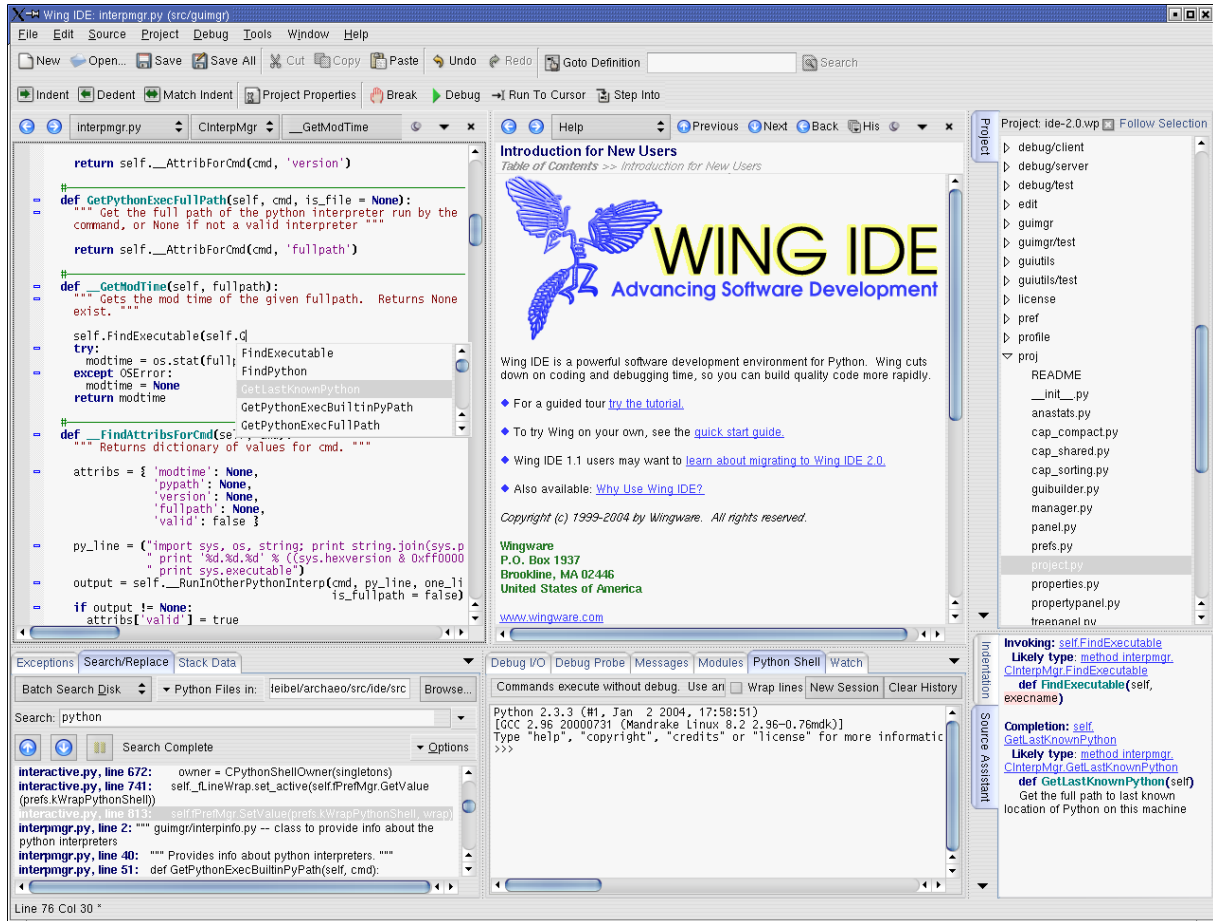
*For hardcore coding, there's nothing to beat Wing IDE.*

——InformIT, Sep 2001.

它所支持的关键特性：

1. 方便的调试器，支持远程调试；
2. 代码浏览功能；
3. 代码编写功能丰富；语法高亮度、自动完成、自动缩进、代码结构折叠、定义跳转；
4. 通过项目管理代码；
5. 跨平台实现；

可以到 <http://wingide.com> 去下载 wingIDE。现在稳定版本是 1.1.10，2.0.0 处于 beta 测试阶段。下面是 2.0.0 版本在 Linux 上的界面：



2.0.0 界面是相当的华丽，但是 1.1.10 的界面风格我不是很喜欢。所以不怎么使用。但是 2.0.0 版本却确实确实值得期待！



# TCL 脚本语言

## 目录

TCL 脚本语言 .....	1
了解它，一小时之内！ .....	5
启动解释器 .....	6
变量和表达式 .....	7
定义函数 .....	8
循环和控制 .....	8
列表和数组 .....	10
输入输出 .....	11
类和面向对象 .....	11
函数库程序包 .....	13
总结 .....	14
基本词法和概念 .....	15
一切都是命令和及其参数 .....	15
单词、引号、括号 .....	17
解释执行过程 .....	18
脚本注释 .....	20
没有 do...while? 自己实现 .....	21
置换：TCL 的灵魂 .....	23
变量置换 .....	23
命令置换 .....	25
反斜线置换 .....	26
subst 置换 .....	27
数学计算，怎么这么烦？ .....	29
语法描述 .....	29
操作符 .....	30
操作数 .....	31
数学函数 .....	32
类型、精度 .....	32
字符串计算 .....	33
计算性能方面的考虑 .....	33
简单变量和字符串处理 .....	35

创建和删除变量 .....	35
string: 一般处理 .....	36
format: 格式化 .....	42
scan: 解析扫描 .....	46
Chr ( ) 和 Ord ( ) .....	47
列表和数组: 高效强大的数据结构 .....	49
列表 .....	49
数组 .....	60
正则表达式 .....	67
正则表达式语法 .....	67
正则表达式匹配查找 .....	74
正则表达式匹配替换 .....	76
控制结构 .....	79
Boolean 类型 .....	79
条件判断 .....	79
循环控制 .....	82
break、continue .....	85
异常处理 .....	86
执行字符串 .....	90
使用 assert .....	97
过程和变量 .....	99
定义过程 .....	99
全局、局部 .....	100
参数默认值 .....	102
可变数量参数列表 .....	102
处理选项参数 .....	105
传引用还是传值? .....	110
再说 return .....	111
过程更名、删除 .....	112
变量跟踪 .....	113
名字空间 .....	117
创建名字空间 .....	117
variable 命令 .....	119
名字和名字解析 .....	120
引入命令 .....	122
输出命令 .....	123
其他名字空间命令 .....	125
面向对象编程 .....	129

定义类 .....	129
使用类和对象 .....	135
继承 .....	138
虚函数 .....	139
find: 查找类和对象 .....	140
程序库和程序包 .....	143
source: TCL 中的#include.....	143
unknown 方法 .....	143
auto_load: 加载程序库.....	144
创建程序库 (library) .....	146
交互模式下执行命令 .....	150
程序包 (package) .....	152
事件驱动 .....	165
after 命令 .....	165
vwait 进入事件循环 .....	166
输入输出系统 .....	169
操作文件系统 .....	169
读写文件 .....	175
匿名管道 .....	188



## 了解它，一小时之内！

TCL 语言非常简单，下面我们通过一个简单的例子，让我们对 TCL 有一个大概的映像和了解。如果你会使用 C Shell 脚本或者有过 C/C++ 的经验，那么 TCL 对你而言，应该非常容易。先看看下面的程序 complex.TCL：

```
#File complex.TCL
#
package require Itcl
package require control

itcl::class Complex {
    public variable m_r      ;#实数
    public variable m_i      ;#虚数

    constructor {r i} {
        set m_r $r
        set m_i $i
    }

    public method + { c } {      ;#复数的加法
        set r [expr "$m_r + [$c cget -m_r]" ]
        set i [expr "$m_i + [$c cget -m_i]" ]
        return [code [Complex #auto $r $i]]
    }

    public method GetReal { } { ;#复数的减法
        return $m_r
    }

    public method GetImag { } {
        return $m_i
    }

    public method - { c }
}
```

```
itcl::body Complex::- {c} {
    set r [expr "$m_r - [$c cget -m_r]" ]
    set i [expr "$m_i - [$c cget -m_i]" ]
    return [itcl::code [Complex #auto $r $i]]
}

pro main {} {      ;#定义了过程 main
    set r 100;set i 200
    Complex a $r $i
    Complex b 50 50
    set c [a - b]

    control::control assert enabled 1
    puts "c.real = [$c GetReal] ; c.imag = [$c GetImag]"
    control::assert "[$c GetReal]==50"
    control::assert "[$c GetImag]==150"
}

main                ;#调用过程 main
```

上面的代码实现了一个简单的复数类，并且实现了几个成员函数，完成加法和减法操作。最后的代码创建两个对象，并且相减得到另外一个对象。然后使用 `assert` 来断定我们的操作是正确的。

## 启动解释器

我们可以在 Dos 提示符中输入命令 `tclsh` 来启动 TCL 解释器，并且进入交互式模式，然后使用 `source` 命令来执行我们刚才创建的脚本，如下：

```
C:\>tclsh
% cd e:/work/script
% source complex.TCL
c.real = 50 ; c.imag = 150
%
```

交互式模式下，我们每输入一次命令，TCL 解释器就执行这命令，然后把命令的执行结果给打印出来。如果出现了语法错误或者异常，就把异常信息给打印出来。

我们也可以在 DOS 提示符下输入 `tclsh e:/work/script` 直接执行我们的脚本，如下：

```
C:\>tclsh e:/work/script/complex.tcl
c.real = 50 ; c.imag = 150

C:\>
```

ActiveTCL 软件包中还有一个程序 `tkCon`，可以在启动菜单中找到。这是一个使用 TCL/Tk 开发的图形界面的脚本解释器，使用起来更加方便。

## 变量和表达式

TCL 中也存在变量的概念，我们可以创建变量，赋值，引用，删除变量，比如上面的代码中：

```
% set r 100;set i 200          ;#创建了两个变量 r 和 i，并且初始化为 100 和 200
200
% Complex m $r $i              ;#引用变量 r 和 i，创建了一个 Complex 对象
m
%
```

TCL 中的变量没有类型，或者我们换一种说法，所有的变量都是同一种类型：“字符串”类型。比如上面例子种的变量 `r` 和 `i`，他们的值分别是 100 和 200，这里 100 和 200 都是字符串，没有整数这么一说。可能我们会迷惑了，如果我要计算这两个数的和，怎么办？

```
% set sum $r+$i                ;#企图计算 r 和 i 两个变量的和，但是事与愿违；
100+200
% set sum [expr $r+$i]          ;#只有这样才行
300
%
```

看看上面的例子就明白了，第一种方法计算 `sum`，结果 `sum` 的值是字符串“100+200”。只有通过第二种方式，使用 `expr` 命令来计算表达式 `$r+$i`，`sum` 的值才是我们期望的 300。在 TCL 中，所有的数学计算都是通过 `expr` 命令来实现的。例如通过计算正弦来得到 2 的平方根：

```
% set PI 3.1415926535897932    ;#创建变量 PI
3.1415926535897932
% expr "cos($PI/4)*2"           ;#计算得到 sqrt(2)
1.41421356237
% expr "sqrt(2)"                ;#直接计算得到 sqrt(2)
1.41421356237
```



TCL 中表达式的写法和 C 语言是比较类似的，并且支持常用的数学函数。

## 定义函数

Pascal、VB 语言中存在函数和过程的区别，并且函数和过程的定义方式不一样。在 TCL 语言中，函数和过程的定义方式没有差别，就像 C/C++ 一样，所以后面的章节中针对 TCL 而言，“函数”和“过程”两种提法是等价的。TCL 中的过程分成两类：

1. TCL 语言自带的核心命令。
2. 用户编写的扩展命令。

TCL 的核心命令只有 80 多条，比如 set 命令；我们还可以使用 TCL 脚本定义自己的过程，也可以使用 C/C++ 语言来实现一些和操作系统等紧密相关的过程。前面的例子代码中，我们就定义了一个过程 main；下面是另外一个过程定义：

```
#过程 Factorial，计算参数 n 的阶乘。
proc Factorial {n} {
    if {$n<=1} {
        return 1
    }
    return [expr $n*[Factorial [expr $n-1]]]
}

puts "10! = [Factorial 10]"      ;#调用过程 Factorial，计算 10!
puts "5! = [Factorial 5]"       ;#调用过程 Factorial，计算 5!
```

可见，TCL 中的过程是可以递归调用的。

## 循环和控制

TCL 的核心命令中提供了常见的控制结构。而且 TCL 和其他语言相比很大的不同是：你甚至可以编写你自己的控制结构！这是后话。TCL 中循环结构包括 for 循环和 while 循环，例如：

```
#使用 for 循环实现阶乘
proc Factorial1 {n} {
    set result 1
    for {set i 1} {$i<=$n} {incr i} {
        set result [expr $result*$i]
    }
}
```

```
        return $result
    }

    #使用 while 循环来实现阶乘
    proc Factorial2 {n} {
        set result 1
        set i 1
        while {$i<=$n} {
            set result [expr $result*$i]
            incr i
        }
        return $result
    }

    puts "10! = [Factorial1 10]"
    puts "5! = [Factorial2 5]"
```

还有一个很有用的 `foreach` 循环，功能和 VB 中的 `foreach` 类似，但是要强大和灵活的多。后面我们详细讨论；

可以使用 `if` 来进行判断分支结构，使用 `switch` 实现多路匹配选择。例如前面采用递归方式定义的 `Factorial` 函数中，就使用了 `if` 结构。`switch` 可以指定字符串匹配模式来进行匹配选择，例如：

```
set c "http://www.microsoft.com"

#根据正则表达式匹配方式来进行 switch 选择
switch -regexp $c {
    "http://.+"      {puts "$c is a http url"}
    ".+@.+"          {puts "$c is a email address"}
    "ftp://.+"       {puts "$c is a ftp url"}
    default          {puts "Other ..."}
}
```

和 C/C++ 类似，TCL 中也有 `break` 和 `continue` 语句。主要用来在循环中控制循环：`break` 跳出最里层的循环，`continue` 掠过本次循环中下面没有执行到的语句，继续下一次循环。要注意的是，`switch` 中各个子句中没有必要象 C/C++ 一样，加上一个 `break`。

## 列表和数组

TCL 中变量分成简单变量和组合变量。组合变量分成两种：列表和数组。这两种组合变量的数据结构虽然简单，但是功能强大。很多人抱怨，TCL 中怎么不能够象 C/C++ 那样定义 struct 或者 union 这样的结构？这是拿 C 语言的思维来使用高级脚本，是行不通的。实际上，有了列表和数组，基本上很少有实际问题无法解决。

列表是一个多个元素的有序的集合，每一个元素通过下标来进行操作，下标从 0 开始，列表中的元素可以是另外一个列表，例如：

```
set students {
    {LeiYuhou    Mail        27}
    {Lily        Femail     25}
    {Tiger       Mail        2}
}

set index 1
foreach s $students {
    foreach {name sex age} $s {
        puts "$index -> $name"      ;#打印出序号和名字
    }
    incr index
}
puts [lindex $students end]
```

上面的代码中，students 就是一个列表，里面三个元素同时也还是列表。通过 foreach 循环把名字给打印出来。最后的语句通过 lindex 命令取出列表的最后一个元素，并且打印出来。

TCL 中的数组则是多个元素的无序的集合，每一个元素都包含两个值：下标（key）和值（value）。在一个数组中，所有元素的下标都是互不相同的，元素通过下标来进行索引和操作。元素值可以是字符串，也可以是列表。例如：

```
array set DAY {
    0 Sunday
    6 Saturday
    5 Friday
    4 Thirsday
    3 Wednesday
    2 Tuesday
}
;#初始化数组变量 DAY
set Day(1) Monday ;#设置 DAY 数组的下标为 1 的元素
```

```
puts "5 - $DAY(5)"      ;#输出第五天的名字
puts "keys - [array names DAY]" ;#输出所有的下标
```

可以看到, TCL 中的数组和我们 C 语言中熟悉的数组完全不是一回事, 倒是和 VBScript 中的对象 Dictionary 非常的类似。TCL 中没有多维数组的概念, 但是后面我们会讲到如何使用多维数组。

## 输入输出

TCL 核心中提供了文件输入输出的命令, 其中标准输入和标准输出可以看成特殊的文件: 它们在进程启动的时候自动打开。TCL 中的输入命令是 `gets`, 输出命令是 `puts`:

```
set v 1
set fh [open "C:/a.txt" w]      ;#打开 C:\a.txt 文件
fconfigure $fh -translation crlf ;#配置成文本模式

while { $v!=""} {
    puts -nonewline "Please input you name:" ;#输出提示信息
    gets stdin v ;#从标准输入读入一个字符串
    puts $fh "your name $v" ;#写入到文件中
}

close $fh ;#关闭文件句柄;
```

`gets` 和 `puts` 不仅可以操作磁盘文件, 还可以操作串口, 操作 `socket` 句柄等。TCL 来源于 UNIX, 众所周知, UNIX 里面的文件是不区分文本和二进制的, 但是为了兼容多种操作系统, 这里增加了一个命令 `fconfigure` 用来配置文件句柄属性, 包括模式, 缓冲区大小等。

## 类和面向对象

这里我们不介绍面向对象编程的概念, 如果你还不知道什么是面向对象, 可以找一本 C/C++ 中相关部分的介绍先了解一下面向对象的基本概念。

TCL 是基于命令的语言, 一个 TCL 程序是由多个命令的线性组合, 本来它是不支持面向对象编程的。但是 TCL 的一个重要特点就是具有非常好的扩展性, 所以网络上出现了不少面向对象的扩展包, 其中最著名的就是 ITCL, 我们前面的例子中复数类就是采用 ITCL 扩展包写成的一个类。ITCL 使 TCL 具备了完备的面向对象特性, 并且在形式和语法上和 C/C++ 非常类似:

1. 封装: 每一个成员变量或者函数, 都可以指定三种保护方式的一种: `public`、`protected` 和 `private`; 其意义和 C++ 中完全一致;
  2. 继承: 一个类可以从多个类中派生, 也就是说, 一个类可以有几个基类;
  3. 多态: ITCL 中类的任何一个成员函数都是虚函数! 所以显然支持多态性了。
- 除了这三点, ITCL 类还支持构造函数和析构函数, 其功能和 C++ 的类似。  
看看下面一个多态性的例子:

```
#使用 ITCL 必须引入 ITCL 扩展包
package require Itcl
namespace import itcl::*

#定义了基类 CPerson
class CPerson {
    protected variable m_name      ;#成员变量, 保护类型, 可以被继承
    protected variable m_sex

    constructor {name sex} {       ;#构造函数
        set m_name $name
        set m_sex $sex
    }

    public method PrintInfo {} {    ;#public 方法, 输出对象信息, 可以被继承
        puts "CPerson [GetInfo]"   ;#调用了成员函数 GetInfo
    }

    public method GetInfo {} {
        return "name=$m_name; sex=$m_sex" ;#返回对象信息
    }
}

class CStudent {
    inherit CPerson                ;#表示本类从 CPerson 继承

    protected variable m_age

    constructor {name sex age} {    ;#构造函数
        CPerson::constructor $name $sex { ;#调用基类的构造函数
            set m_age $age
        }
    }
}
```

```
}

public method GetInfo {} {      ;#返回对象信息
    return "name=$m_name; sex=$m_sex; age=$m_age"
}
}

CPerson a "LeiYuhou" M          ;#构造 CPerson 对象实例
CStudent b "Lily" F 20          ;#构造 CStudent 对象实例

a PrintInfo                      ;#分别输出两个对象的信息
b PrintInfo
```

上面的代码中，声明了两个对象，CPerson 是 CStudent 类的基类，CPerson 基类中声明了一个函数 PrintInfo 用来输出一些信息，这个函数在 CStudent 中也可以被调用。这个函数在基类 CPerson 中定义，调用了另外的一个函数 GetInfo，这个函数在基类和继承类中都有各自不同的定义；PrintInfo 究竟应该调用哪一个 GetInfo，就看调用这个函数的对象类型。

上面的代码中，a 是 CPerson 类型，b 是 CStudent 类型，所以上面的代码执行结果是：

```
CPerson name=LeiYuhou; sex=M
CPerson name=Lily; sex=F; age=20
```

C++对象的多态性必须通过对象指针或者引用来体现，也就是说，只有通过对象指针来调用虚函数，C++的多态性才能够体现出来。这样的原因在于 C++是通过虚表（virtual table）的方式来实现多态性的。ITCL 中没有指针这么一说，那么多态性如何实现的呢？这个问题我没有深入研究，但是一般脚本语言的多态性都是通过查表回溯的方式来实现，比如 Python。TCL 应该也是采用类似的方法实现面向对象的多态性的。

## 函数库程序包

一个大型程序往往不是由一个文件组成的，一般需要很多其他的函数库。C/C++语言中，先逐个编译程序单元，再连接（link）所有的 obj 文件以及使用到的库文件，最后生成目标程序。TCL 有两种自己特有的处理方式：函数库和程序包。

函数库是比较古老的方式，和 C 语言的函数库比较类似。TCL 解释器在执行命令的时候，如果碰到不认识的命令，就会通过一定的索引方式在特定的位置寻找它。这种方式不支持版本升级，现在使用的比较少了，一般情况下我们使用另外一种方式：程序包。

程序包可以使用 TCL 语言编写，也可以使用 C/C++来实现；比如我们刚才提高的面向对象，就是通过扩展包 ITcl 来实现的。一个脚本在使用某一个扩展包之前，必须先将扩展包

引入，语法如下：

```
package require ?-exact? package ?version?
```

例如，为了使用 http 协议处理包：

```
package require -exact http 2.4.5
```

其中包名字参数是必须的。如果指定选项-exact，表示加载指定的版本。在网络上存在很多很多 Tcl 的扩展包，并且大部分是免费的，安装也非常方便：只要把相关的文件复制到特定的目录中即可。所以建议采用 package 的形式来开发 TCL 扩展代码。

## 总结

这一章我们介绍了 TCL 的基本语法和简单的用法，给大家一个对 TCL 粗略的认识。但是要真正掌握 TCL 的使用，还有很多细节性的东西需要进一步了解。后面的章节中我们针对这一章出现的各个概念进行详细的解释。



## 基本词法和概念

在进一步深入了解 TCL 的语法之前,弄清楚 TCL 脚本的几个基本概念以及 TCL 解释器解释执行一个脚本的基本流程,是很有必要的。

### 一切都是命令和及其参数

TCL 脚本语法的本质其实非常简单:

1. 一个脚本是由一个或多个命令以及其参数顺序排列而成;命令之间用换行字符或者分号分隔;TCL 中的一切都是命令及其参数。
  2. 一个命令语句包括一个命令字以及零个或多个该命令的参数;命令和参数以及参数之间用空格或者 Tab 分隔;
  3. 如果任何地方出现可以进行置换的操作,那么就会按照规则进行置换;
- 例如下面的一段代码实际上是由三个命令组成的:

```
class CPerson {                                ;#第一条命令 class 及其两个参数
    protected variable m_name                  ;#成员变量, 保护类型, 可以被继承
    protected variable m_sex

    constructor {name sex} {                   ;#构造函数
        set m_name $name
        set m_sex $sex
    }

    public method PrintInfo {} {                ;#public 方法, 输出对象信息, 可以被继承
        puts "CPerson [GetInfo]"              ;#调用了成员函数 GetInfo
    }

    public method GetInfo {} {
        return "name=$m_name; sex=$m_sex"      ;#返回对象信息
    }
}                                                ;#第一条命令结束。

CPerson a "Lei Yuhou" Male                     ;#第二条命令
a PrintInfo                                     ;#第三条命令
```

上面代码很长，但是实际上只有三个命令：第一个是 `class` 命令，带有两个参数，第一个是类名，第二个是类的定义体；第二个命令是 `CPerson` 命令，实际上刚才的 `class` 命令在执行之后就定义了一个新的 TCL 命令，命令名就是我们声明的类名；第三个命令的命令字是 `a`，这是我们刚才创建的对象，实际上 `CPerson` 命令在执行的时候又创建了一个 TCL 命令，其名字就是我们给出的对象名。

命令之间只有两种分隔符号：分号或者换行符号。所以如果我们要把两个命令写在一行，那么必须使用分号分隔它们。例如上面的后两个命令：

```
CPerson a "Lei Yuhou" Male ; a PrintInfo
```

要注意的是，并不是所有的分号和换行符号都是命令分隔符！！例如下面的代码：

```
if {$a>100} {  
    puts "a = $a"  
}
```

三行代码，三个换行符只有最后一个是 `if` 命令的分隔符；而前两个相对于 `if` 命令而言，只是其第二个参数中的字符而已。初学者往往会把上面的代码写成下面两种形式：

```
if {$a>100}{           ;#条件判断的}后面没有空格，直接跟着{  
    puts "a = $a"  
}  
  
if {$a>100}           ;#条件判断的后面直接回车，把执行体的{放到下一行  
{  
    puts "a = $a"  
}
```

两种写法都是错误的。错误代码分别如下：

```
extra characters after close-brace  
while executing  
"if {$a>100}{  
    puts "a = $a"  
}  
  
wrong # args: no script following "$a>100" argument  
while executing  
"if {$a>100}"
```

第一种情况，`{ $a>100 }`和后面的`{`之间没有空格，那么就会报错：因为不符合单词的语法。第二种情况，`{`直接拿到了 `if` 命令的第二行，那么 TCL 认为这里的 `if` 命令在本行就结束，那么 `if` 命令就只有一个参数，显然也会报错。

什么时候回车字符作为命令结束符，什么时候又不是？为什么会出现第一种错误？回答这些问题之前，我们必须弄清楚另外一个重要概念：单词！

## 单词、引号、括号

一个脚本是由多个命令顺序排列而成；而一个命令则是由多个单词组成。单词是由空格来进行分隔的一个字符串，例如下面的命令中：

```
set a 100
```

就包含了三个单词：set、a 和 100

但是也有一些特殊情况：

## 双引号

如果一个单词以一个双引号字符开始，那么这个单词会以下一个双引号作为结束。如果在这两个双引号之间出现了分号“;”、反方括号“]”、以及空格或者换行字符，那么这些字符都作为单词中的字面字符出现，它们不具备任何特殊含义。开始和结束的双引号不作为这个单词内的字符。例如：

set a 100	;#命令 1
"set" "a" "100"	;#命令 2
set b "First Second"	;#命令 3
set b First Second	;#命令 4，错误
set c "The first line The second line"	

上面代码中，前面两个命令是完全等价的，虽然命令 2 中三个单词都用双引号括起来，但是三个单词和命令 1 中的三个单词完全一样。执行后变量 a 的值都为 100；命令 3 中第三个单词用引号括起来，其中包含了空格，如果没有引号，象命令 4 中所示，那么就会出现错误。因为 set 命令不能够接受三个参数。最后一个命令中也包含三个单词，最后一个单词是用引号括起来的跨行字符串，里面包含了换行符号。

用双引号括起来的单词中，所有的置换类型都可以发生。具体我们后面在置换一节中详细解释。

## 花括号

如果一个单词以字符“{”开始，那么它必须以相对应的反括号字符“}”作为结束。单

词内部的花括号可以嵌套，但是它们必须配对出现；如果某一个花括号前面是反斜线，那么 TCL 解释器在寻找配对括号的时候该括号不计算在内。该单词是所有出现在花括号在内的原始字符，但是不包含开始和结束的花括号。例如下面的代码：

```
set a {
    {青山依旧在, }
    {几度夕阳红。}
}

set b { \{ Sat "Hello" \} }

puts $a
puts $b

set b { \{ Sat "Hello" } }
```

代码执行结果如下：

```
{青山依旧在, }
{几度夕阳红。}

\{ Sat "Hello" \}
wrong # args: should be "set varName ?newValue?"
while executing
"set b { \{ Sat "Hello" } }"
```

最后一行代码出错了，因为 set 第三个单词内，嵌套的花括号不能匹配上。如果一个单词是用花括号括起来的，那么里面所有的置换都不会发生。我们回归头来看看前面 CPerson 类的定义，类的定义部分就是一个用花括号括起来的，包含多行字符创的单词。

通过上面的分析，可以看到，下面的两块代码在“一定程度上”是等价的：

```
if {$a>100} {
    puts "a = $a"
}
```

```
if "$a>100" {
    puts \"a = $a\"
}
```

为什么说是“一定程度”呢，我们有必要了解 TCL 解释执行脚本的过程。

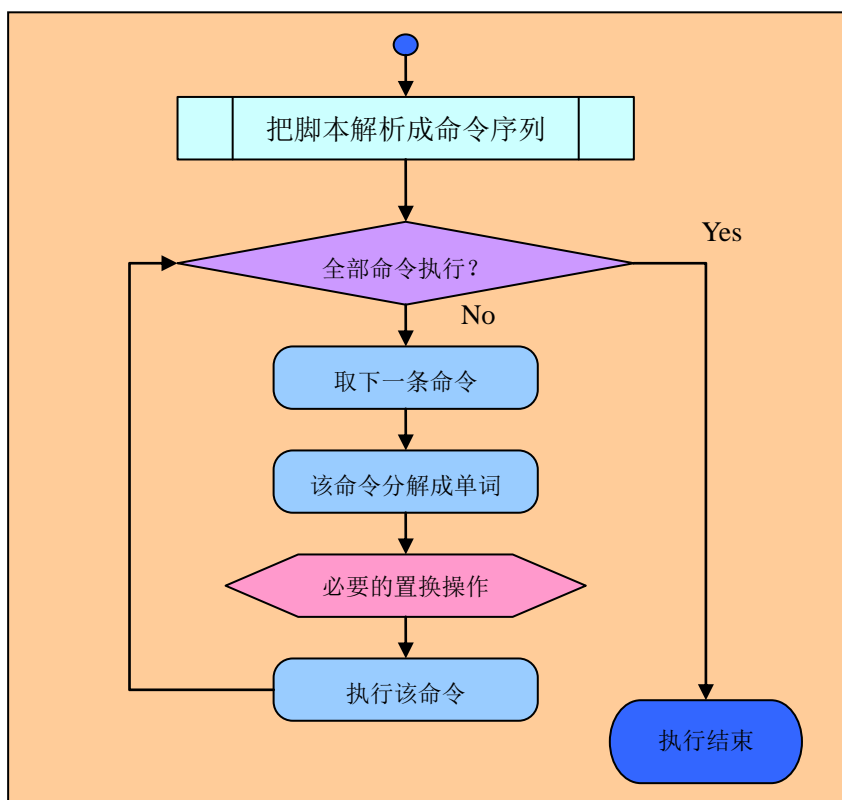
## 解释执行过程

TCL 解释器执行脚本的过程就是：

1. 根据前面介绍的命令分隔规则和单词划分规则，将脚本文件内容解析成多个命令的

序列；

2. 按照从上到下的顺序，逐个执行命令；
3. 执行一个命令的时候，
  - a) 如果组成命令的单词中存在变量、反斜线换或者命令置换，会根据单词的特点进行置换；
  - b) 置换后，把第一个单词作为命令字，到命令库中找到对应的命令执行体；
  - c) 如果找到，把后面的单词作为参数传入给命令，执行命令；
  - d) 如果没有找到，那么就通过一定的机制继续寻找，如果找不到，就抛出异常；
4. 所有的命令执行完，那么脚本执行就结束。



理解了上面的脚本执行过程，就可以理解上一节所说的“一定程度”是什么意思了：执行器在执行一条命令的之前，首先把命令划分成单词，然后根据单词的写法来进行置换。

在上一节左边的代码中，代码被划分成三个单词，但是后面两个单词都是用花括号括着的，所以没有会发生置换，直接把它们作为参数调用 if 命令。

而右边的代码中，同样是三个单词，只不过后面的两个单词被双引号括着，所以在执行 if 命令之前，解释器会先对这两个单词进行置换，把里面的 \$a 用 a 的值替代，然后把替换后的单词作为参数传递给 if 命令。

下面的代码会进入死循环，为什么？大家自行分析。

```
set a 100
set i 0
set sum 0

while "$i <= $a" {      ;#如果换成{$i<=$a}, 就不会死循环了。
    incr sum $i
    incr i
}

puts $sum
```

需要指出的是，上面解释执行过程中的第三步是可以嵌套的，也就是说，在执行一个过程的时候，在这个过程内部也可以通过这个过程来执行另外一块脚本代码。就比如上面的 `while` 循环，它的第一个参数是条件表达式“`$i<=$a`”，第二个参数是一段代码；在执行 `while` 命令的过程中，首先计算条件表达式，根据表达式的值确定是否执行下面的代码，如果条件满足，那么就执行。执行这一段代码的过程和执行 `while` 的类似：先划分成命令序列，再顺序执行每一条命令。这一段代码如何执行，是由 `while` 命令来控制，而不是 TCL 解释器所控制的。所以，`while` 不过是一个命令！

## 脚本注释

理解了脚本的执行过程，就不难理解 TCL 中的注释规则：

当 TCL 解释器在解释执行脚本的时候，如果期望下一个单词是一个命令，但是这个单词的第一个字符是“`#`”，那么从这个“`#`”开始直到这一行的结束，都被 TCL 当作注释而忽略过去。

下面是几个例子：

```
#这是一行合法的注释
set a 100      ;#创建变量 a，合法的注释
set a 100      #创建变量 a，不合法的注释；

for {set sum 0;set i 0;#初始化} {$i<=100} {incr i;#递增} {
    incr sum $i
}

puts $sum
```

第三行注释不正确，因为这里 `#` 以及后面的字符都被解释成了 `set` 命令的参数，所以会报错：

```
wrong # args: should be "set varName ?newValue?"  
while executing  
"set a 100"      #创建变量 a，不合法的注释"
```

上面的错误，是我们经常犯的错误。

上面例子中，for 循环命令中的注释，似乎和前面介绍的 TCL 注释规则相违背，注释并没有从“#”直到一行结束，而是只到花括号结束。仔细分析，实际上并不违背：例如 for 的第一个参数，包括里面的注释，是整个作为一个单词被传递给 for 命令来解释的，for 命令执行这个单词的时候，就会正确的解释其中的注释。

## 没有 do...while? 自己实现

TCL 中没有 do...while 循环，但是我们明白了上面的道理之后，完全可以自己实现一个 do 命令，作为我们自己的一种控制结构。这在其他的 Python、Pascal 等语言中是很难想象的：python 里面没有 switch，你试着在 python 实现一个 switch，看能否实现？

下面就是一个 do 命令的简单实现，使用 TCL 写成：

```
#定义 do 命令  
proc do { body while_key cond } {  
    if { $while_key != "while" } {  
        error "the second parameter must be \"while\"."  
    }  
  
    for {} {1} {} {      #使用 for 循环来执行循环体  
        set r [catch {uplevel $body} msg]      ;#捕获异常信息  
        switch $r {  
            0 {  
                #this is normal,do nothing  
            }  
            1 { this is a error,so throw the exception.  
                return -code error -errorinfo $::errorInfo -errorcode $::errorCode  
            }  
            2 { #this is return;  
                return -code return $msg  
            }  
            3 { #this is break;  
                break  
            }  
        }  
    }  
}
```

```
        4 {
            #this is continue,do nothing;
        }
    }
    set r [uplevel expr $cond]    #判断条件，是否应该继续执行。
    if { !$r } {
        break
    }
}

set a 0      ;set sum 0      ;#使用 do-while 来计算计算 1+2+3+...100
do {
    incr a
    incr sum $a
} while { $a<=100}

puts $sum    ;#打印出结果。
```

可以看到，这个 `do` 命令带有三个参数：循环体、`while` 关键字以及最后的判断条件。其内部是采用 `for` 命令实现循环。后面的章节中，我们会使用 C/C++ 来实现 TCL 中的 `do...while` 循环。



## 置换：TCL 的灵魂

我们已经知道了一个脚本的解释执行过程，其中在一个命令真正被执行之前，TCL 解释器会对组成这个命令的命令字和参数进行置换操作。深入了解 TCL 置换的规则和本质，有助于我们解决实际代码编写中的许多问题。TCL 中的置换操作有以下几种：变量置换，命令置换，反斜线置换和 subst 置换，下面我们一一道来。

### 变量置换

如果一个单词中包含”\$”符号，那么 TCL 就会执行变量置换；在置换的时候，\$符号连带后面的字符都会被后面字符所表示的变量的值所置换。变量置换有如下三种语法：

1. `$name`：其中 `name` 是一个变量的名字，这个名字只能由字母、数字、下划线或者名字空间分隔符（两个或两个以上的冒号）组成；
2. `${name}`：其中 `name` 是一个变量的名字，它可以包含任何字符，除了 “}” 字符；
3. `$name(index)`：其中 `name` 是一个数组的名字，`index` 是数组元素的下标；这种方式表示用指定数组中 `index` 所表示的元素的值来进行置换。其中 `name` 只能是字母、数字、下划线或者名字空间分隔符组成，也可以是空字符串。而在 `index` 上也可以先进行变量置换、命令置换、反斜线置换等。

三条规则，看起来头晕，还是看看几个例子吧：

```
set m_name "Leiyuhou"
puts "name is $m_name"           ;#第一种替换规则，最常用

set {her^name%} "Lily"          ;#变量名里面有^，%等字符
puts "my wife is ${her^name%}"  ;#第二种替换规则：${name}
#puts "my wife is $her^name%"   ;#如果采用第一种，则会出错；

array set day {0 Sunday 1 Monday 2 Tuesday} ;#创建名为 day 的数组
puts "2 -> $day(2)"              ;#第三种置换规则
set order 2
puts "2 -> $day($order)"         ;#第三种，其中下标还进行了变量置换；
puts "2 -> $day([set order])"    ;#第三种，其中下标部分进行了命令置换；
puts "2 -> $day(\x32)"           ;#第三种，其中下标部分进行了反斜线置换；
```

一般而言，第一种和第三种是我们最常用的。但是如果变量名字里面包含了字母、数字、下划线和名字空间分隔符之外的字符；那么我们就需要使用第二种置换规则了。就像上面的

例子中，变量名字为 `her^name%`，如果采用 `$her^name%`，那么就会报错：

```
can't read "her": no such variable
while executing
"puts "my wife is $her^name%""
(file "E:\Work\script\complex.tcl" line 7)
```

可见在使用第一种置换规则的时候，TCL 解释器会从\$后面的第一个字符开始开始往后搜索，当发现第一个不是字母、不是数字、下划线和名字空间分隔符的时候，就停止。然后把搜索到的字符串作为变量名字进行置换。上面的例子中，解释器搜索到了 `her`，然后寻找 `her` 这个变量，结果这个变量不存在，所以就报错。

在一个单词中可以有任意多个变量置换；如果单词是被花括号括起来的，那么任何变量置换都不会发生。请看下面的代码：

```
set m_name "Leiyuhou"
set {her^name%} "Lily"
array set day {0 Sunday 1 Monday 2 Tuesday}
set printf puts

puts "name = $m_name ; her name = ${her^name%} ; day(2)=$day(2)"
puts {name = $m_name ; her name = ${her^name%} ; day(2)=$day(2)}

puts "name = $m_name"           ;#发生置换
puts "{name = $m_name}"        ;#发生置换
puts {name = $m_name}          ;#不发生置换
$printf "name = $m_name"       ;#命令字中也可以进行变量置换
```

上面代码的执行结果是：

```
name = Leiyuhou ; her name = Lily ; day(2)=Tuesday
name = $m_name ; her name = ${her^name%} ; day(2)=$day(2)
name = Leiyuhou
{name = Leiyuhou}
name = $m_name
name = Leiyuhou
```

有了前面介绍的“单词”的概念，就不难理解上面的结果：所有被花括号括起来的单词都不会发生变量置换。

## 命令置换

如果一个单词中包含左方括号 “[”，那么 TCL 就会执行命令置换。在进行命令置换的时候，TCL 解释器把 “[” 后面的字符串当作 TCL 脚本来执行，这个脚本可以包含任意多个命令，并且必须用一个右方括号 “]” 结束。这一串脚本的执行结果会替换掉整个 “[...]”。在一个单词中可以包含任意多个命令置换，名且命令置换内部还可以包含命令置换；如果一个单词是花括号括起来的，那么变量置换就不会发生。

看看代码，一切都比较清楚了。

```
set a 100
set result "1+2+3+...[set a] = \
[set sum 0
for {set i 1} {$i<=$a} {incr i} {
    incr sum $i
}
set sum
]"

puts $result
puts "[set a]*$a = [set result [expr $a*$a]]"
puts {[set a]*$a = [set result [expr $a*$a]]}
```

这是执行结果：

```
1+2+3+...100 = 5050
100*100 = 10000
[set a]*$a = [set result [expr a*a]]
```

可以看到，第一个 set reult 命令的第二个参数中，包含了一个很长的命令置换，它们用方括号括起来，由三条独立的命令组成：set sum 0；for 循环命令；以及最后的 set sum。这组命令的最后一个命令（set sum）的执行结果就是整个命令置换的结果：5050。

最后的两个输出，第一个是用引号括起来，所以发生了命令置换；并且方括号中还可以嵌套方括号；第二个输出使用花括号括起来，结果没有发生任何置换，原样输出。

我们可以把命令置换类比其他语言中的函数调用，比如 C 语言中的：

```
int a = abs(c);
```

TCL 中就是：

```
set a [expr abs($c)]
```

## 反斜线置换

经常有人问题，TCL 中如何续行？实际上 TCL 的续行和 C 语言差不多，都是通过行末的反斜线来进行的。这在 TCL 中是反斜线置换的一种形式。

如果一个单词里面出现反斜线字符“\”，那么就会发生反斜线置换。根据反斜线字符后面跟随字符的不同，反斜线置换有着不同的处理方式。下面表格中列出了所有需要特殊处理的格式：

格式	说明
<code>\a</code>	响铃，等于字符(0x7
<code>\b</code>	退格字符，等于 0x8
<code>\f</code>	表格换行（Form feed），等于字符 0xc
<code>\n</code>	换行，NewLine，等于字符 0xa
<code>\r</code>	回车字符，Carriage-return，等于 0xd
<code>\t</code>	制表符，Tab 键，等于字符 0x9
<code>\v</code>	垂直制表符，Vertical tab，等于字符 0xb
<code>\&lt;newline&gt;whiteSpace</code>	表示反斜线后面紧跟换行符号，接着跟着很多空格或者 Tab；这种情况下，TCL 会把这部分用一个空格字符置换掉。这和我们 C/C++ 中的续行比较类似。这种置换即使在花括号内也能发生。
<code>\\</code>	表示单个反斜线字符。
<code>\ooo</code>	反斜线后面是一个八进制的数值，表示一个 Unicode 字符。这个字符的高 bit 为 0。
<code>\xhh</code>	hh 表示一个十六进制的数值，表示一个 Unicode 字符。这个字符的高 bit 为 0。注意的是 \x 后面可以跟任意多个 h，但是只有最后两个 h 被保留。 \xff 和 \x11ff 是等价的。这个 Unicode 字符的高字节为 0。
<code>\uhhhh</code>	hhhh 表示一个 Unicode 字符，这个字符是两字节，其中 hhhh 可以为 1 至 4 个。 \uaa 和 \u00aa 是等价的。

除了上面表格中列举出的情况需要特殊处理之外，其他任何情况下，反斜线“\”都是被简单的抛弃，后面的字符被当作原始的字符而插入到单词中。所以这种情况下，双引号，方括号和“\$”都可以出现在单词中，而不进行额外的置换处理。

要注意的是：如果反斜线所在的单词是被花括号括起来的，除了续行置换之外，其他的反斜线置换都不会发生。请看下面的代码：

```
set a 100
set b 200
```

```
puts " variable \"a\" = $a;\n\n      variable \"b\" = [set b]\n\n"
```

```
puts "$a , \ $a , \ [set a]"
```

```
puts "\m ^ \% \{ \}there"
```

```
variable "a" = 100; variable "b" = 200
100, $a, [set a]
m ^ % { }there
```

**subst 置换****subst ?-nobackslashes? ?-nocommands? ?-novariables? *string***

```
% set a 100
100
% set b a
a
% subst {[set b]}
$a          ;#这里的字符串用花括号括起来，所以第一次置换没有产生任何效果；
% subst "$[set b]"
100         ;#这里第一次置换，将[set b]置换成 a，然后 subst 进行$a 置换，结果为 100。
```

版权所有

一点要注意：一种类型的置换可以会导致另外一种置换的发生。假如 `subst` 指定了参数 `-novariables`，那么变量置换不会发生，但是命令置换依然会发生。如果在执行命令的时候需要进行变量置换，那么变量置换依然会发生。还是看看代码：

```
% set a 100
100
% subst -novariables {$a , [set c [expr sqrt($a)]]}
$a , 10.0           ;#我就不多解释了
```

下面是另外一个例子，它来自于 ActiveTCL 的 OnlineHelp：

```
proc b {} {return c}
array set a {c c [b] tricky}
subst -nocommands {[b] $a([b])}
```

`subst` 的结果是 `[b] c`，而不是 `[b] tricky`。因为 `$a([b])` 变量置换的时候必须执行 `[b]` 的命令置换。

还有几个特殊情况：

- 如果在 `subst` 置换的时候发生错误，那么 `subst` 就会抛出异常；
- 如果置换时发生 `break` 异常，那么在发生 `break` 之前的结果就是这个置换的结果；
- 如果发生 `continue` 异常，那么这个置换结果就是一个空字符串；
- 如果发生 `return` 异常，或者是 `return` 的其他代码，那么返回值就是置换结果；

看看下面的例子：

```
% subst {name = $name}
can't read "name": no such variable
% subst {name , [break] , LeiYuhou}
name ,
% subst {name , [set a 100;break;set b 200] , Leiyuhou}
name ,
% subst {name , [set a 100;continue;set b 200] , Leiyuhou}
name , , Leiyuhou
% subst {name , [set a 100;return LeiYuhou] , hehe}
name , LeiYuhou , hehe
% subst {name , [set a 100;return -code 10 LeiYuhou;set b 200] , hehe}
name , LeiYuhou , hehe
```

## 数学计算，怎么这么烦？

前面的快速入门教程里面已经提到了 `expr` 命令，TCL 中所有的数学计算都是通过 `expr` 命令来完成的。这一点和其他语言非常不同，例如 Python 中：

```
>>> from math import *
>>> pi
3.1415926535897931
>>> print cos(pi/4)*2 , sqrt(2)
1.41421356237 1.41421356237
>>>
```

Python 里面的表达式是非常直观的，其他语言例如 VBScript 等也是如此。但是 TCL 中，即使是计算两个数字相加，都得使用 `expr` 命令：

```
set a 100;      set b 200
puts "a+b = $a+$b"      ;#输出 a+b = 100+200
puts "a+b = [expr $a+$b]" ;#输出 a+b = 300
```

TCL 中，`expr` 加上一堆的 `$` 符号，方括号等字符，最简单的表达式，看起来都是非常不爽。这对于刚刚接触 TCL 的人来说，简直是一件无法忍受的事情，当初我也是这么认为的。后来在网上发现有人还真的通过扩展，在 TCL 实现类似 Python 的语法，比如原来的：

```
set a [expr $a+$b]
```

可以写成：

```
a = $a+$b
```

至于是怎么实现的，我们后面会进行详细的分析。但是我觉得，既然选择了 TCL，就要习惯其用法；要么当初就选择 Python 拉到，所以我不赞成上面的这种改进。而且习惯了 TCL 之后，也没有觉得它没有我们想象中的那么不方便。

## 语法描述

`expr` 命令是 TCL 的核心命令之一，其语法格如下：

```
expr arg ?arg arg ...?
```

该命令可以有一个或者多个参数，在执行的时候，把所有的参数连接起来成为一个数学



表达式字符串，然后计算这个表达式，计算结果作为命令结果返回。下面两条命令写法不一样，但是结果一样：

```
set PI 3.1415926535897932
set sqrt2_1 [expr "cos($PI/4)*2"]
set sqrt2_2 [expr cos ( $PI / 4 ) * 2]

puts "sqrt2_1 = $sqrt2_1\nsqrt2_2 = $sqrt2_2"
```

两个 `expr` 命令，第一个命令只有一个参数（所有的东西都被引号括起来），第二个则拆开了，参数个数一堆。但是计算结果都是一样的：

```
sqrt2_1 = 1.41421356237
sqrt2_2 = 1.41421356237
```

## 操作符

`expr` 命令所支持的计算操作符是 C 语言中操作符的一个子集。并且写法和优先级也和 C 语言完全一致。除了数学计算操作符之外，`expr` 还支持字符串的比较操作。下面是 `expr` 支持的操作符，按照优先级从高到低排列：

操作符	含义说明
<code>- + ~ !</code>	单目取负；单目正号；按位取反（not）；逻辑取反；
<code>* / %</code>	乘法；除法；取余；
<code>+ -</code>	加法；减法；
<code>&lt;&lt; &gt;&gt;</code>	向左移位，向右移位；
<code>&lt; &gt; &lt;= &gt;=</code>	比较操作符，小于、大于、小于等于、大于等于；结果 1 表示 true，0 表示 false；这几个操作符可以作用于字符串，进行大小写敏感的比较；
<code>== !=</code>	比较是否相等：等于、不等于；结果 1 和 0 分别表示 true 和 false。可以作用于所有类型的操作数；
<code>eq ne</code>	字符串的比较：等于、不等于；结果 1 和 0 表示 true 和 false。进行计算的时候，其操作数都被当成字符串进行比较。
<code>&amp;</code>	按位与操作（AND），只能够作用于整数；
<code>^</code>	按位异或操作（XOR），只能作用于整数；
<code> </code>	按位或操作（OR），只能作用于整数；
<code>&amp;&amp;</code>	逻辑与操作（AND），只有两个操作数都非零的时候，结果才是 1，否则为 0。只能够作用于 boolean 类型或者数字类型。
<code>  </code>	逻辑或操作（OR），只有两个操作数都为 0 的时候，结果才是 0，否则为 1；只能够作用于 boolean 或者数字类型。
<code>x?y:z</code>	If-then-else 操作，和 C 中的比较类似。如果 x 计算得出非零值，那么 y 的值就

是表达式的结果；否则，z 的值就是表达式的结果。x 的值必须是一个数值或者 boolean 值。

其中，&&，||，x?y:z 三个操作符和 C 语言中一样，会进行布尔短路计算：当表达式计算了一部分就知道整个结果的时候，那么其他部分就不会计算了。例如：

```
proc a { t } {  
    puts "PROC A . t=$t"  
    return [expr $t+1]  
}  
  
puts [expr {[a 100]>100 ? [a 200] : [a 300]}]
```

上面的代码执行输出为：

```
PROC A . t=100  
PROC A . t=200  
201
```

可以看到，[a 300]根本就不会执行。但是如果这样写：

```
puts [expr “[a 100]>100 ? [a 200] : [a 300]”]
```

把 expr 后面得表达式用双引号括起来，结果就不一样了，a 命令会执行三次！为什么，大家都是高手，自己分析。

## 操作数

数学表达式中肯定离不开操作数，操作数是操作符进行运算的对象（这简直是废话）。

expr 命令在解析操作数的时候，依照如下顺序进行：

1. 整数。expr 首先判断是否是整数，expr 表达式中的整数可以表示成普通的十进制，也可以表示成 8 进制（用 0 开始）和十六进制（用 0x 开始）。
2. 浮点数。如果操作数不具备上面描述的整数的格式，那么就试图将其解释成浮点数。expr 表达式中的浮点数和 ANSI-C 中的浮点数格式类似，后面不能够加上 f,F,L,l 这样的后缀，例如 2.1，3.，3.2E1.0 等都是正确的浮点数。
3. 字符串。如果既不能够解释成整数，也不能够解释成浮点数，那么就只有当成字符串了。

使用整数注意范围，8.4 版本的 TCL 中，整数的最大范围是 0x7FFFFFFFFFFFFFFF，也就是说，使用 8 个字节来表示的。但是 8.3 版本的 TCL 中，整数是 32 个字节来表示。

expr 的操作数可以是如下的形式：

1. 一个数值；
2. \$varName 格式的变量引用；

3. 双引号括起来的字符串；表达式解析器会对双引号之间的字符串进行置换处理，并且使用置换后的结果作为操作数；
4. 花括号括起来的字符串；其中的字符串不会进行任何的置换处理；
5. 方括号起来的命令；命令执行后的结果被作为操作数；
6. 数学函数，函数的参数可以是这 6 中形式中的任何一种。例如：`sin(cos($PI/[set a 2.0]))`；

## 数学函数

上面提到了 `expr` 表达式中可以包含数学函数。实际上，TCL 本身提供的数学函数虽然数目不多，但是基本上能够满足我们平常的需要：

<code>abs</code>	<code>cosh</code>	<code>log</code>	<code>sqrt</code>
<code>acos</code>	<code>double</code>	<code>log10</code>	<code>srand</code>
<code>asin</code>	<code>exp</code>	<code>pow</code>	<code>tan</code>
<code>atan</code>	<code>floor</code>	<code>rand</code>	<code>tanh</code>
<code>atan2</code>	<code>fmod</code>	<code>round</code>	<code>wide</code>
<code>ceil</code>	<code>hypot</code>	<code>sin</code>	<code>cos</code>
<code>int</code>	<code>sinh</code>		

具体每一个函数的我就不具体介绍了，基本上顾名思义，就可以猜出来。

如果你要是觉得上面的数学函数还不够用，想自己定义一个数学函数也能够使用，那么就需要使用 C/C++ 进行扩充了，需要用到的函数是 `Tcl_CreateMathFunc()`。这我们后面讲到 TCL 高级编程的时候会详细讲解。

## 类型、精度

TCL 使用 C 语言实现的，所以在 `expr` 计算的时候，所有的整数在内部都是被当作 C 语言的 `long` 类型，所有的浮点数都被当作了 C 语言的 `double` 类型。在进行计算的时候，整数、浮点数和字符串会进行自动的转换，如果有必要的话。例如：

```
% expr 5/4.0
1.25
% expr 5 / [string length "abcd"]
1
% expr 5 / ([string length "abcd")+0.0)
1.25
% expr 25.0/5.0
```

## 字符串计算

`expr` 也支持字符串的比较操作。除了一般的 `>`, `>=` 等之外, 还有 `eq`, `ne` 两个操作符:

```
expr {"0x03" > "2"}      ;#结果是 1
expr {"0x03" eq "3"}     ;#结果是 0
```

使用 `expr` 进行字符串比较要非常小心, 有些时候会自动的转换成整数进行比较, 那样结果就会出乎我们意料之外了。如果是比较字符串是否相等, 如下两个建议:

1. 使用 `ne` 和 `eq` 要比 `!=`, `==` 要好的多;
2. 使用 `string compare` 要比 `ne` 和 `eq` 又要好的多;

## 计算性能方面的考虑

程序员对于性能方面总有很多的需求, 如何写好 `TCL` 表达式, 对于整个脚本的性能性能影响是很大的。下面我们来看看几个例子:

```
set PI 3.1415926535897932
set FOUR 4
set TWO 2

set c [time {
    set d [expr sin($PI/$FOUR) * $TWO * sqrt($TWO)]
} 1000]
puts "$c , $d"

set c [time {
    set d [expr "sin($PI/$FOUR) * $TWO * sqrt($TWO)"]
} 1000]
puts "$c , $d"

set c [time {
    set d [expr {sin($PI/$FOUR) * $TWO * sqrt($TWO)}]
} 1000]
puts "$c , $d"
```

```
set c [time "  
    set d [expr {sin($PI/$FOUR) * $TWO * sqrt($TWO)}]  
" 1000]  
puts "$c , $d"
```

上面的代码通过两种方式计算出 $\sqrt{2}$ ，然后相乘得到结果 2。`time` 命令是 TCL 提供的核心命令之一，用来计算执行一段代码需要多少时间；下面是执行结果：

```
149 microseconds per iteration , 2.0  
129 microseconds per iteration , 2.0  
19 microseconds per iteration , 2.0  
3 microseconds per iteration , 2.0  
3 microseconds per iteration , 2.0
```

计算结果都是一样的，但是耗费的时间却大不一样。相差可以达到 50 倍。我们来一一分析怎么回事：

1. 1 和 2 两种写法的差别在于，2 中 `expr` 的参数只有一个，整个表达式用双引号括起来了；而 1 中的 `expr` 的参数却有多。从结果来看，2 比 1 要快一些！但是相差不大。
2. 2 和 3 两种写法的差别在于，3 中 `expr` 的表达式是用花括号括起来的，而 2 是用引号括起来的。两个 `expr` 的参数个数都是一个，但是性能却相差比较多了；
3. 3 和 4 的两种写法的差异在于，4 中把需要测试的语句块用双引号括起，而 2 中用花括号括起；第四种写法速度快得不可思议，难道真的这么快吗？非也，TCL 解释器在执行的时候，首先就把 `expr` 给计算了一遍，然后 `time` 进行计算时间的时候，实际上是在计算“`set d 2.0`”这条语句的执行时间，所以速度才这么快。看看最后的一句输出就知道了。

总结一下经验：

要想 `expr` 运行得最快，只需要把 `expr` 的表达式全部用花括号括起来即可。

## 简单变量和字符串处理

前面多次提到了，TCL 脚本语言是基于字符串的语言，其中的变量没有类型，或者说，只有一种类型，那就是字符串类型。其实 TCL 中的变量还是可以分类的，按照其结构不同，可以分成两大类：简单变量和组合变量。一个简单变量中就包含一个值，而一个组合变量中则可以包含很多的值。

这一章介绍简单变量的处理，至于组合变量，下一章进行详细介绍。

## 创建和删除变量

对于变量有两个概念很重要，就是变量的名字和变量的值。TCL 中的变量名字可以是任意的字符，甚至可以包含空格和引号等，前面我们讨论变量置换的时候已经看到了。为了使用的方便，我们还是按照 C/C++ 中的变量标识符命名规范来给 TCL 中的变量命名比较好。

我们通过 set 命令语法可以看出来：

```
set varName ?value?
```

set 命令可以有一个或者两个参数：

1. 当有两个参数的时候，它给名为 varName 的变量赋值为 value，如果变量 varName 不存在，那么就创建变量；
2. 有一个参数的时候，它返回变量 varName 的值；如果变量 varName 不存在，就产生错误；

unset 命令可以把一个或者多个变量删除，其格式如下：

```
unset ?-nocomplain? ?--? ?name name ...?
```

其中。-nocomplain 选项表示命令执行的时候任何错误都不会抛出异常；--选项表示选项结束；后面可以跟 0 个或者多个变量名字。看看下面的例子：

```
%set word "Hello world"
Hello World
%puts [set word]      ;#set 只有一个参数的时候，就把变量的值返回
Hello World!
%puts [set abc]       ;#如果查询的变量不存在，那么就产生错误。
can't read "abc": no such variable
%unset word
%puts $word
```

```
can't read "word": no such variable    ;#变量 word 已经消失了，所以产生错误；
```

除了 set 命令之外，还有一些命令可以创建变量，比如 append, foreach 等。例如：

```
append c "Hello"          ;#创建变量 c
append c ", World"        ;#往 c 后面追加字符串
puts $c                   ;#输出 c，结果为“Hello, World”
```

append 命令用来向一个变量后面追加内容，如果变量不存在，就创建这个变量。下面是 foreach 的例子：

```
set a {LeiYuhou 26591 ShenZhen}
foreach {name id address} $a {
}
puts "name = $name ; id = $id ; address = $address"
```

上面的脚本执行结果为：

```
name = LeiYuhou ; id = 26591 ; address = ShenZhen
```

看来，foreach 命令也可以创建变量，并且为它们赋值。

和其他语言中的变量一样，TCL 中变量也有作用域，局部变量在退出其作用域的时候会自动的销毁。具体和变量作用域有关的东西我们在后面介绍过程的时候详细介绍。

TCL 中所有变量都是字符串，它们究竟代表什么含义，怎样解释它们，是使用它们的命令的责任，所以在 TCL 中进行字符串处理是很频繁的事情，下面我们开始介绍 TCL 中字符串处理。

## string：一般处理

MFC 中处理字符串有 CString，STL 中有 string 类；TCL 中则有 string 命令，能够完成大部分常见的字符串操作。string 命令参数繁多，其一般语法格式如下：

```
string option arg ?arg ...?
```

根据 option 的不同，string 命令执行不同的操作。下面列举如下：

## 取字符串长度

string length string 以及 string bytlength string 两个选项，前者用来获得字符串的字符个数，后者用来获得字符串所占内存的字节数。下面是一个例子：

```
% set a {I Love You,中国}
I Love You,中国
```



```
% string length $a      ;#返回字符个数，为 13
13
% string bytelength $a   ;#返回字节个数，为 17
17
```

可以看到，同样的字符串变量 `a`，其字符格式为 13（不信你可以数一下，“中国”是两个字符）；但是在内存中，变量 `a` 却占用了 17 个字节的内存。因为在 TCL 内部字符串是使用 UTF-8 这种 Unicode 编码方式来保存字符串的，UTF-8 编码模式下，一个字符可能会占用 1 至 3 个字节。

在 TCL 中，需要知道字符串字节数的情况，非常罕见，至少我遇到的次数不多；

## 查找子串、Index 方法、替换、去头去尾

如果我要取出一个字符串变量 `a` 中指定下标位置的字符，该怎么办？C/C++中好说：

```
printf("the character in 10 is %c", a[10]);
```

看起来非常的直观，但是很可惜的是 TCL 不支持这种写法，只能这样写：

```
set a "仰天大笑出门去，我辈岂是蓬蒿人！"
puts "the character in 10 is [string index $a 10]"
```

这里采用的是 *string index stringvalue indexvalue* 的形式，其中 *stringvalue* 是字符串，*indexvalue* 就是字符所在下标。这里下标有三种表示方法：

index 语法	解释和例子
integer	一个从 0 开始的整数，首字符下标是 0；如果超出字符串长度，返回空；
end	end 用来表示字符串的最后一个字符；
end-integer	表示倒数第 integer+1 个字符，例如 end-1 表示倒数第二个字符

这种下标的表示方法在其他字符串操作中也会用到，TCL 的列表中元素下标也是采用相同的表示方法。例如：

```
% set a "Open the file"
Open the file
% string index $a 2
e
% string index $a end
e
% string index $a end-2
i
```

```
% string index $a "end - 1"      ;#end-1, 中间不要空格
bad index "end - 1": must be integer or end?-integer?
% string index $a end-0
e
% string index $a end+0          ;#语法不正确
bad index "end+0": must be integer or end?-integer? (looks like invalid octal number)
% string index $a end+1
bad index "end+1": must be integer or end?-integer?
%
```

TCL 中查找字符串有如下几种形式:

***string first string1 string2 ?startIndex?***

***string last string1 string2 ?lastIndex?***

两者都是在字符串 string2 当中查找子串 string1, 如果找到了, 就返回 string1 在 string2 中的起始下标; 否则返回 -1。如果指定了最后的可选参数, 那么就从这个指定下标开始查找。第一个命令是从前往后查找, 第二个则方向相反。例如:

```
% set a
0123456789abcdef abcdef
% string first a $a
10
% string first a $a 11
17
% string first A $a
-1
% string last a $a
17
% string last a $a 13
10
%
```

一般我们写程序的时候, 会先查找子串位置, 然后可能就需要取出一个子串, 这可以通过下面的命令实现:

***string range string first last***

请看几个例子:

```
% set a
0123456789abcdef abcdef
% string range $a 0 10
0123456789a
```

```
% string range $a 10 0      ;#如果 first>end, 返回空字符串
% string range $a 10 end
abcdef abcdef
% string range $a -2 40      ;#如果 first<0 或者 last>end, 就当成 0 或者 end
0123456789abcdef abcdef
%
```

替换字符串中的子串可以使用 `string replace` 命令来实现, 格式如下:

***string replace string first last ?newstring?***

该命令把 `string` 参数给出的字符串中从 `first` 到 `last` 之间的子串替换成为 `newstring` 参数表示的字符串, 其中 `first` 和 `last` 参数遵从 `index` 表示语法, 而 `newstring` 参数是可选的, 如果省略该参数, 那么 `string` 命令中 `first` 和 `last` 之间的字符串被删除。

```
% set a "0123456789ABCDEF"
0123456789ABCDEF
% set b [string replace $a 9 12]      ;#把 9—12 的字符删除
012345678DEF
% puts "$a  ; $b"
0123456789ABCDEF  ; 012345678DEF      ;#输出替换前和替换后的结果
% set b [string replace $a 10 end-3 " Hello "] ;#采用 end 等方法表示下标
0123456789 Hello DEF
% puts "$a  ; $b"
0123456789ABCDEF  ; 0123456789 Hello DEF
%
```

有时候我们希望去掉字符串头尾的一些字符, 比如空格, 可以使用下面的命令:

***string trim string ?chars?***

***string trimleft string ?chars?***

***string trimright string ?chars?***

三个命令分别用来去掉字符串两头、首部和尾部的指定字符。如果指定了参数 `chars`, 那么就去掉指定的字符, 否则就去掉空白字符 (空格、Tab、回车换行)。请看例子:

```
% set a
Hello,
World.

% string trim $a      ;#去掉了字符串 a 前后的空格, 回车等
Hello,
World.
```

```
% set a "abc DEFG 0123"
abc DEFG 0123
% string trim $a "abc23"           ;#去掉首尾出现在"abc23"里面的所有字符。
DEFG 01
```

## 字符串比较

前面我们讲解 `expr` 命令的时候，提到了该命令也能够进行字符串比较的操作，具体请参考前面的章节。使用 `string` 命令来比较字符串，是一种更好的选择：

***string compare ?-nocase? ?-length int? string1 string2***

***string equal ?-nocase? ?-length int? string1 string2***

上面两个命令都是用来进行字符串比较的，参数形式和意义完全一致，但是返回结果有差异。第一个命令：当 `string1` 和 `string2` 两个字符串比较相等的时候，返回 0；如果 `string1` 比 `string2` 小，则返回 -1；其他情况返回 1。第二个命令，只有当两个字符串相等的时候，才返回 1；其他情况都返回 0。

如果指定了选项参数 `-nocase`，则表示进行大小写无关的匹配，否则进行大小写敏感的匹配。如果指定了参数 `-length int`，表示只比较两个字符串的前面 `int` 个字符的子串。请看例子：

```
% set a "0123456789abcdef"
0123456789abcdef
% set b "0123456789ABCDEF"
0123456789ABCDEF
% string compare $a $b
1
% string compare -nocase $a $b
0
% string compare -length 10 $a $b
0
% string equal $a $b
0
% string equal -nocase $a $b
1
```

除了上面介绍的字符串比较之外，在 TCL 中还广泛使用“字符串匹配”。这是一种简单的模式匹配，但是在 TCL 中使用非常广泛，后面我们会反复看到。`string` 命令支持这种字符串模式匹配：

***string match ?-nocase? pattern string***

该命令用来判断参数 `string` 给出的字符串是否和模式 `pattern` 相匹配。如果两者匹配上，

那么命令返回 1，否则返回 0。pattern 可以是任意字符串，但是如果包含下面特殊的字符串，那么就有特殊的含义：

语法	含义
*	匹配任意字符串序列，包括空字符串
?	匹配任意单个字符
[chars]	匹配 chars 指定的集合中的任意单个字符；如果是 x-y 的形式，那么就匹配任意落在 x-y 之间单个字符。可以写成[a-zA-Z0-9]这样的形式。
\x	x 可以是*?[]中的某一个字符，表示该字符。

来看下面的例子：

```
% set a
Age : 34
% string match "Age*" $a      ;#字符*匹配 Age 后面的所有字符
1
% string match "Age ? \[0-9]\[0-9]" $a      ;#[0-9]匹配数字
1
% string match {Age ? [0-9][0-9]} $a
1
% string match { * ? [0-9][0-9]} $a
1
% string match -nocase {[a-z][a-z][a-z] ? [0-9][0-9]} $a      ;#[a-z]匹配任意字母
1
% string match {[a-z][a-z][a-z] ? [0-9][0-9]} $a
0
```

字符串匹配非常简单，但是功能却不弱，所以在 TCL 的 array，list，以及 switch 命令和文件操作等命令中得到了广泛的使用。

## 字符串映射

有些时候我们希望把某一个字符串中特定的一些子串全部替换成为另外的一些字符串，下面的命令可以轻松实现这一点。

***string map ?-nocase? charMap string***

先看看例子，再来讲解语法：

```
% string map {abc 1 ab 2 a 3 1 0} 1abcaababcbabababc
01321221
% string map {abc 1 11 *} "1abc"
11
```

这里参数 `charMap` 是一个列表，里面的元素是按照 `key, value, key, value……` 这样的序列出现，每一个 `key` 用来进行查找，对应的 `value` 就是要替换的字符串。所以上面例子中，`abc` 子串全部被替换成 1，`ab` 子串被替换成为 2，等等。

映射置换是有先后顺序的，如果存在多重替换的可能，那么 `charMap` 参数中先出现的 `key` 具有较高的优先级。比如上面的例子中，`abc` 可能会被映射成 1，也有可能成为 2c（`ab` 映射成 2），还有可能成为 3bc；但是实际成为 1，因为 `abc->1` 在最前面出现，优先级最高。

还有很重要的一点，映射的扫描过程只进行一遍。看上面例子中的第二个命令：`abc` 首先映射成为 1，那么最后的结果就是 11，虽然存在 `11->*` 的映射，但是该映射并不会发生。

## format: 格式化

C 语言中的 `sprintf` 函数以及 MFC 中类 `CString` 成员函数 `Format` 可以进行字符串格式化，在 TCL 中，`format` 可以完成类似的功能。其格式如下：

**format** *formatString* ?arg arg ...?

其实 `format` 命令参数形式和 C 语言的 `sprintf` 大同小异，不同的只是第一个参数的具体语法会有些差异。该参数用来控制最后输出的格式，我们先看看例子：

```
% format {%2$s ; %1$s ; %3$ 0*.3hf ;} first second 20 3
second ; first ; 000000000000003.000 ;
% format {%2$s ; %1$s ; %3$ 0*.3hd ;} first second 20 3
second ; first ; 003 ;
% format {%2$s ; %1$s ; %3$ 0*.3hd ;} first second 20 0x11223344
second ; first ; 13124 ;
% format {%2$s ; %1$s ; %3$ 0*.3hx ;} first second 20 0x11223344
second ; first ; 3344 ;
% format {%2$s ; %1$s ; %3$ 0*.3lx ;} first second 20 0x11223344
second ; first ; 11223344 ;
```

上面的例子也许大家看了就头晕；很多人写了一辈子的 C 语言代码，但是还没有弄清楚格式控制字符串的语法:-)。这里就给大家来一个详细的说明。

`formatString` 的格式控制字符串由普通字符串以及格式控制域组成，普通字符原样输出，但是格式控制字符则要把后面对应参数拿过来进行格式转换，然后放入到输出字符串中。每一个格式控制域由字符 `%` 开始，从左往右依次包含如下六个字段：

1. 参数位置限定符；
2. 前缀、对齐、填充、符号等控制字符；
3. 域宽度；
4. 精度控制；
5. 长度修饰字符；

## 6. 转换类型；

除了最后一个转换类型字段之外，其他的所有字段都可以省略。

## 参数位置限定

如果百分号%后面跟着一串数字以及一个\$字符，那么这就是参数位置限定。它表示本域需要转换的参数序号。例如"%3\$d"，表示取后面参数列表中的第三个参数进行转换。1表示首个参数。一般情况下，我们都省略位置字段，表示顺序选取后面的参数来转换。但是有时候，我们需要指定位置。上面例子中第一条命令：我们格式化的时候依次选择第二个、第一个和第三个来进行转换。

要注意的是：某一个域指定了位置限定之后，其他所有的域都必须使用位置限定，否则会出错：

```
% format {%2$s ; %s ;} first second
cannot mix "%" and "%n$" conversion specifiers
```

## 前缀对齐以及符号

该字段用来修饰该域的前缀、宽度等。可以是如下字符的任意顺序组合：

修饰符	描述
—	表示该域应该是左对齐的；
+	表示该域应该在前面加上表示正负的符号，即使该参数是整数；
space	如果该域的第一个字符不是正负符号的话，那么前面必须加上一个空格；
0	该域的左边应该使用字符 0 来填充，而不是默认的空格；
#	表示自动在前面加上修饰符；如果使用 o 输出 8 进制，前面自动加上 0；如果使用 x 或者 X 输出十六进制，自动加上 0x 或 0X

请看下面的对比：

命令	输出
format {%s : % d} LeiYuhou 100	LeiYuhou : 100
format {%s : %+ d} LeiYuhou 100	LeiYuhou : +100
format {%s : %d} LeiYuhou 100	LeiYuhou : 100
format {%s : %x} LeiYuhou 100	LeiYuhou : 64
format {%s : %#x} LeiYuhou 100	LeiYuhou : 0x64
format {%s : %#X} LeiYuhou 100	LeiYuhou : 0X64



## 域宽度

该字段指出了该域的最小宽度，当转换的时候如果小于最小宽度，就会采用默认空格填充，如果没有指定左对齐，并且指定了 0 填充，那么就会使用 0 填充。该域可以是一个数，也可以是\*字符，表示从参数列表中取宽度。请看例子：

命令	输出
<code>format {%s : %-010d;} LeiYuhou 100</code>	<code>LeiYuhou : 100      ;</code>
<code>format {%s : %010d;} LeiYuhou 100</code>	<code>LeiYuhou : 0000000100;</code>
<code>format {%s : %10d;} LeiYuhou 100</code>	<code>LeiYuhou :      100;</code>
<code>format {%s : %0*d;} LeiYuhou 10 100</code>	<code>LeiYuhou : 0000000100;</code>
<code>format {%s : %0*d;} LeiYuhou 15 100</code>	<code>LeiYuhou : 000000000000010;</code>

可以看到，最后两个命令中，使用了\*作为宽度，那么 `format` 命令在执行的时候，会从参数列表中取出对应的参数作为宽度，其后面紧跟的参数作为转换内容。这里两个宽度分别为 10 和 25。

## 精度控制

第四个字段表示输出域的精度，它是一个小数点加上数字组成；针对不同的转换类型，精度有不同的含义。下面分类说明：

1. 如果转换类型是 `e`、`E` 或者 `f`，那么该字段指出了小数点后面应该出现的数字个数；
2. 如果转换类型是 `g` 或者 `G`，表示所有应该出现的数字个数，包括小数点左右两边的数字；
3. 如果转换类型是整数，表示应该输出的最小数字；
4. 如果转换类型是 `s`（字符串），表示该字符串输出的最大长度；如果该字符串比指定精度要长，那么字符串后面超出的部分将会被截掉；

精度可以通过\*来指定，表示后面参数列表中对应该位置的整数是精度。例如：

```
% format "%.4s" abcdef
abcd
% format "%. *s" 12 abcdef
abcdef
% format "%. *s" 4 abcdef
abcd
% format "%. *d" 4 4
0004
%
```

## 长度修饰字符;

第五个字段是长度修饰符，必须是字母 **h** 或者字母 **l**。如果是 **h**，表示在转换之前将整数解释成 16bit 的；如果是 **l**，则解释成 64bit 的。如果没有该字段，那么解释成系统默认的宽度。该宽度可以通过查询 `tcl_platform(wordSize)` 得到，例如：

命令	结果
<code>% format "%ld" 0xff001122ee002233</code>	-72038752318381517
<code>% format "%lu" 0xff001122ee002233</code>	18374705321391170099
<code>% format "%u" 0xff001122ee002233</code>	integer value too large to represent
<code>% format "%hu" 0xffffffff</code>	65535
<code>% format "%hu" 0xfffffffffff</code>	integer value too large to represent

## 转换类型

最后一个字段就是转换类型，是一个单独的字母，解释如下：

字符	说明
<b>d</b>	把整数转换成带符号的十进制数值字符串
<b>u</b>	把整数转换成不带符号的字符串
<b>i</b>	转变成十进制字符串，数字可以是十进制、八进制(0 开始)或者十六进制(0x)
<b>o</b>	把整数转变成八进制字符串
<b>x、X</b>	把整数转换成十六进制的格式， <b>x</b> 表示小写 a-f， <b>X</b> 表示大写 A-F
<b>c</b>	把整数转换成它表示的 UniCode 字符
<b>s</b>	就是字符串本身，不作转换
<b>f</b>	把浮点数转换成字符串
<b>e、E</b>	把数字转换成科学记数法表示的格式
<b>g、G</b>	如果幂小于 -4，或者大于等于精度，那么就等价于 <b>e</b> ；否则等价于 <b>f</b>
<b>%</b>	插入一个百分号字符。

下面我们来看看几个简单的例子：

```
% format "%x , %X , %d , %u , %i" 244 245 -100 300 012
f4 , F5 , -100 , 300 , 10
% format "%c% %c" 0x30 0x30
0%c
```

## scan: 解析扫描

相比 `format` 命令, 我们使用 `sacn` 命令会比较少一些。该命令和 C 语言中的 `scanf` 命令比较类似, 命令格式如下所示:

```
scan string format ?varName varName ...?
```

该命令按照参数 `format` 中指定的格式, 从参数 `string` 所指定的字符串中解析出各个域, 如果后面指定了变量名列表, 那么解析出的结果就放到各个变量中, 否则把解析出来的结果作为一个列表返回。其中参数 `format` 是一个字符串, 用来指导 `scan` 具体的扫描活动: `scan` 同时扫描 `format` 和 `string`, 根据扫描到的 `format` 中的字符, 来决定如何解释 `string`。

如果 `format` 中下一个字符是空格或者 `Tab`, 那么它匹配 `string` 中的任意多个空格 (包括零)。如果不是 `%`, 那么 `string` 中扫描到的字符串必须和它精确匹配。如果是一个 `%`, 就表示这是一个转换字段, 每一个字段最多包括四个域:

1. 位置标识符; 和 `format` 中的位置标识符形式和意义完全一样。
2. 表示最大宽度的数字;
3. 字段大小修饰符; 可以是 `I` 或者 `L`。
4. 转换字符; 可以是 `d`、`o`、`x`、`i`、`u`、`c`、`s`、`e`、`f`、`g`、`[chars]`、`[^chars]`、`n`

如果 `%` 后面紧跟着的是一个星号 `*`, 表示扫描到的值会被丢弃, 不赋值给任意变量。如果转换字符是 `n`, 那么不消耗 `string`, 而是将当前已经扫描了的字符个数作为值返回。

例如:

```
% scan "ErrorCode is 13 20" "ErrorCode is %d %d" a b
2
% puts "a=$a , b=$b"
a=13 , b=20
% scan "ErrorCode is 13 20" "ErrorCode is %*d %d" a      ;#这里第一个字段丢掉
1
% puts "a = $a"
a = 20
% scan "ErrorCode is 13 20" "ErrorCode is %d %d"          ;#不指定参数列表, 返回列表
13 20
% scan "ErrorCode is 13 20 ." "ErrorCode is %d %d %n"
13 20 19          ;#这里%n, 转换出来的是 19, 表示已经扫描了 19 个字符
```

如果指定的参数列表中变量个数和 `format` 中转换域的个数不匹配, 那么就会出错。例如:

```
% scan "ErrorCode is 13 20 ." "ErrorCode is %d %d %n" a b
different numbers of variable names and field specifiers
```

## Chr ( ) 和 Ord ( )

这是一般语言里常有的函数，Chr 用来返回指定 ASCII 码所代表的字符，Ord 用来返回指定字符的 ASCII 码。但是一般 TCL 语言初学者往往找不到 TCL 对应的函数，事实上，format 和 scan 两个命令可以完成同样的功能，如下所示：

```
#-----  
#返回 ascii 所代表的字符  
proc Chr { ascii } {  
    if { $ascii<=255 && $ascii>=0 } {  
        return [format "%c" $ascii]  
    }  
    return  
}  
  
#-----  
#返回字符 char 的 ASCII 码  
proc Ord { char } {  
    scan $char "%c"  
}  
  
puts "Chr(100) = [Chr 100]"  
puts "Ord([Chr 100]) = [Ord [Chr 100]]"
```

上面的代码输出如下：

```
Chr(100) = d  
Ord(d) = 100
```



## 列表和数组：高效强大的数据结构

虽然字符串在 TCL 中扮演了重要的角色，但是如果 TCL 中只有字符串这种数据类型，那么其功能可能就要大打折扣了。列表和数组是 TCL 中两种重要的组合数据类型。我们先从下面的例子来看看列表和数组的使用。

### 列表

存在这么一个文本文件 `use.log`，里面每一行是一条记录，用来记录每一个用户使用服务器的时间，单位为秒。第一列是名字，第二列是时间。格式如下：

```
leiyuhou    30
wangtao     430
leiyuhou    89
lizhenghua  120
lizhenghua  340
.....
```

同时存在另外一个文件 `id.ini`，里面每一行都是记录，用来记录每一个用户的工号，第一列是名字，第二列是工号。格式如下：

```
leiyuhou    26591
hanzizhong  25565
.....
```

我们的任务是：统计出每一个用户使用服务器的总时间，次数，并且按照时间作为第一关键字，次数作为第二关键字进行排序输出，在输出的时候，同时输出工号，格式如下：

```
姓名      工号      时间      次数
leiyuhou   26591     125       3
wangfang   33123     120       4
.....
```

下面就是我们的代码，用来完成组合数据，排序和输出的工作：

```
1  #-----
2  #File      - listExample.tcl
3  #Author    - Leiyuhou
4  #Created   - 2004/12/9
5  #-----
```

```
6
7   set PATH {E:\Work\TCL、Python 和自动化测试\script\}    ;#数据文件存放
   路径
8
9   #-----
10  #将 use.log 的内容读入到 uselogbuf 缓冲区
11  set f [open "${PATH}use.log" r]
12  set uselogbuf [read $f]
13  close $f
14
15  #-----
16  #将 id.ini 的内容读入到 idbuf 缓冲区
17  set f [open "${PATH}id.ini" r]
18  set idbuf [read $f]
19  close $f
20
21  set uselog [split $uselogbuf "\n"] ;#每行都是一个列表元素
22  set idlist $idbuf
23
24  #创建一个 result 数组，key 为名字，value 为工号
25  foreach {name id} $idlist {
26      if {$name==""} continue
27      set result($name) $id
28  }
29
30  #-----
31  #将所有的数组整理到 result 数组当中去，下标为用户名字，
32  #对应的值是一个列表，格式为{工号，时间，次数}
33  foreach x $uselog {
34      foreach {name tm} $x {}    ;#取得姓名和时间，放到变量 name 和 tm 中去。
35
36      if {$name==""} { continue }
37
38      set val $result($name)
39      foreach {id totaltime count} $val {}
40      if {$totaltime==""} {
41          set totaltime $tm
42      } else {
```



```
43         incr totaltime $tm
44     }
45
46     if {$count==""} {
47         set count 1
48     } else {
49         incr count
50     }
51
52     set result($name) [list $id $totaltime $count]
53 }
54
55 set names [array names result] ;#取得所有的名字列表，存放在 names 中
56 set t ""
57 foreach name $names {
58     set tmp $result($name)
59     set tmp [linsert $tmp 0 $name] ;#在列表头加上名字
60     lappend t $tmp ;#追加到列表 t 的后面
61 }
62
63 #-----
64 #这是一个用于列表元素比较的回调函数。首先按照时间比较，然后按照次数
65 #比较
66 #如果两者完全相同，那么函数返回 0。
67 proc OnCompare {a b} {
68     foreach {name id tm1 cnt1} $a {} ;#取得变量 a 的时间和次数
69     foreach {name id tm2 cnt2} $b {} ;#取得变量 b 的时间和次数
70
71     if {$tm1<$tm2} {return -1} ;#首先按照时间比较
72     if {$tm1>$tm2} {return 1}
73
74     if {$cnt1<$cnt2} {return -1} ;#然后按照次数比较
75     if {$cnt1>$cnt2} {return 1}
76
77     return 0
78 }
79
80 set m [lsort -command OnCompare $t] ;#对列表 t 排序，结果放在 m 变量中。
```

```
80
81  foreach x $m {
82      puts $x      ;#输出列表中的所有元素。
83  }
99
```

下面我们分成两个章节来逐一讲解列表和数组的使用。

## 创建列表

列表的创建过程很简单，并且有很多方式，常见的有：

1. set listname “element list”
2. split string ?splitChars?
3. lappend listname element .....
4. list ?arg

可能会觉得纳闷，第一种方式是创建变量的方式，怎么又能够创建列表呢？事实上，TCL 语言中一个字符串和一个列表本质上几乎没有差别，上面代码中的第 18 行，读入文件的所有内容放到变量 idbuf 中。这时 idbuf 可以看成是一个字符串变量，也可以当作一个列表，看看随后第 25 行的 foreach 命令，就是把 idbuf 当作一个列表，逐个元素进行处理。下面的代码例子更加简单：

```
#演示字符串和列表
set alist "Hello world
I

am TCL"

puts [format "%40s" $alist]
puts "alist is a list.has [llength $alist] elements"
puts "[lindex $alist 1]"
```

变量 alist 可以使用 format 进行格式化处理，也可以用 llength 来就算列表长度。把一个字符串当作列表使用的使用，其中的连续空格（包括空格字符、Tab 和换行）作为列表元素的分隔。所以上面 alist 的元素个数为 5。

有些时候我们需要按照自己的需要把一个字符串分隔成列表，比如把字符串中的逗号作为分隔符号进行分隔，这时候我们就利用第二种方法，调用命令 split。例如如下的代码：

```
set a “www.google.com/index/img///a.img”
```

```
set b [split $a "."/]      ;#将字符串 a 按照.和/字符作为分隔符进行分割

puts $b
```

其运行结果如下：

```
www google com index img {} {} a img
```

`lappend` 命令则是另一个创建列表的命令，它和命令 `append` 比较类似。比如前面例子第 60 行代码，其命令格式如下：

`lappend varname ?val val.....?`

第一个参数是列表变量名，后面参数都可选。如果变量不存在，那么就创建该变量。否则后面的值作为列表元素追加到列表后面，看看下面的例子：

```
% lappend a "hello"
hello
% lappend a ","
hello ,
% lappend a "world"
hello , world
% lappend a I am TCL
hello , world I am TCL
% lappend a
hello , world I am TCL
%
```

最后要介绍的是 `list` 命令，其格式如下：

`list ?val val...?`

后面的参数都是可选的。如果没有参数，那么返回一个空列表。否则参数作为列表元素，看下面的例子：

```
% set a [list]
% puts $a

% set a [list 100 200 {300 400}]
100 200 {300 400}
%
```

## 拆分和连接

把拆分字符串成列表，上一节介绍了 `split` 命令，这里不作进一步的介绍。有时连接多个列表是有必要的，这里需要用到 `concat` 命令和 `join` 命令。`concat` 用来将多个列表连接成一个新的列表，例如：

```
% concat a b {c d e} f {g {h i}}
a b c d e f g {h i}
% set a [concat "a b {c  " d "  e} f"]      ;#这个命令需要注意！三个列表，共六个元素
a b {c d e} f      ;#这是 concat 后的结果
% foreach x $a {puts $x}                    ;#输出列表各个元素。concat 后只有五个元素!!
a
b
c d e
f
% set a [concat "a b {c  " d "  e f"}      ;#这里 e 后面少了一个花括号}
a b {c d e f
% foreach x $a {puts $x}                    ;#输出元素，会出错。
unmatched open brace in list
%
```

上面最后一个 `concat` 是不是让人很郁闷？分明 `concat` 运行成功，但是却是一个无效的列表。怎么办？好的方法就是避免字符串中不要出现没有配对的花括号。

`join` 命令用来将列表中的元素连接成字符串。例如：

```
% join "100 200 {300 400}"
100 200 300 400
% join "100 200 {300 400}" ","
100,200,300 400
% join "100 200 {300 400}" " " << "
100 << 200 << 300 400
%
```

`join` 可以看作 `split` 命令的逆过程。

## 取列表元素

TCL 中的列表是有序的元素集合，获取其中指定下标的元素，需要 `lindex` 命令。格式如下：

```
lindex list ?index...?
```

后面参数 `index` 表示下标，从 0 开始；是可选参数，可以有 0 个或者多个。。下标可以使用 `end` 或者 `end-integer` 的形式。如果有多个 `index` 表示取出嵌套列表中的元素。请看例子：

```
% set a [list 100 200 {300 { 400 500} 600}]
100 200 {300 { 400 500} 600}
% lindex $a          ;#没有 index 参数，则返回整个列表。
100 200 {300 { 400 500} 600}
% lindex $a 0        ;#返回列表头上的元素。
100
% lindex $a 2 1      ;#取出列表 a 的第二个元素作为列表，并且返回其第一个元素。
400 500
% lindex $a {2 1}    ;#同上，第二个元素是列表，然后取出其第一个元素。
400 500
% lindex [lindex $a 2] 1 ;#这是比较笨的写法。功能同前者一致。
400 500
% lindex $a end 0     ;#最后一个元素的第一个元素；end 表示最后一个元素
300
% lindex $a end-1     ;#end-1 表示倒数第二个元素。
200
```

很多使用 Tcl 编程多年的人，往往不知道 `lindex` 命令可以带有多个 `index` 参数，取出列表中列表的元素的时候，往往多次调用 `lindex` 命令。非常惭愧的是：我也是其中之一。对于元素个数很多并且嵌套较多的列表，一个 `lindex` 命令带上多个 `index`，能够获得更好的执行效率。

## 插入、替换和删除元素

向列表中插入元素的方法是 `linsert` 命令，格式如下：

```
linsert list index element ?element...?
```

其中参数 `list` 是需要插入元素的列表，`index` 是插入的位置，可以是整数或者 `end-integer` 的形式。可以这样来理解，`index` 表示我们插入元素之后，被插入的新元素在列表中的下标。`linsert` 命令可以一次插入多个元素。看看下面的例子代码：

```
% set city {Shanghai Nanjing Wuhan Shenzhen} ;#创建 city 列表
Shanghai Nanjing Wuhan Shenzhen
% linsert city 0 Beijing      ;#这里是 city，不是$city，看看返回结果
Beijing city
% linsert $city 0 Beijing    ;#这是插入 city 列表，用的是$city
Beijing Shanghai Nanjing Wuhan Shenzhen
% puts $city                ;#注意，city 变量内容没有任何变化
Shanghai Nanjing Wuhan Shenzhen
% linsert $city end Hongkong Taipei ;#向 city 的末尾追加元素
Shanghai Nanjing Wuhan Shenzhen Hongkong Taipei
% put city                  ;#city 列表的内容还是没有变化
Shanghai Nanjing Wuhan Shenzhen
%
```

`linsert` 的第一个参数是一个列表，而不是列表变量名，所以插入元素之后，命令返回新列表，但是原来的列表不会改变！！就如上面例子中所示。这一点是新人容易迷惑的地方：明明我插入了一个元素，可以列表内容为什么没有变化？要做到这一点很简单：

```
% set city [linsert $city end Hongkong Taipei] ;#向 city 的末尾追加元素
Shanghai Nanjing Wuhan Shenzhen Hongkong Taipei
% put $city
Shanghai Nanjing Wuhan Shenzhen Hongkong Taipei
```

列表元素替换和删除用的是同一个命令：`lreplace`。其格式如下：

`lreplace list first last ?element ...?`

如果后面的 `element` 参数个数为 0，那么就是把下标在 `first` 和 `last` 之间(包括 `first` 和 `last`) 所有元素都删除。否则，先删除，然后把所有的 `element` 参数插入到 `first` 位置。

```
% set city
Shanghai Nanjing Wuhan Shenzhen
% lreplace $city 1 2 Xian      ;#把下标为 1 和 2 之间的元素替换为 Xian
Shanghai Xian Shenzhen
% lreplace $city 2 end Guangzhou Qingdao
Shanghai Nanjing Guangzhou Qingdao
% lreplace $city 2 end
Shanghai Nanjing
```

和 `linsert` 类似，`lreplace` 不会改变参数 `list` 指定的列表，只是把新的列表作为结果返回。

## 排序

命令 `lsort` 用来对列表元素进行排序。其格式如下：

`lsort ?options? list`

该命令有几个可选的选项参数，用来控制排序的方式，常见的选项如下：

- ascii: 按照 ASCII 顺序进行元素排列；
- dictionary: 按照字典顺序进行排序；它与 `ascii` 的差异在于：a) 字典顺序中大小写无关，b) 元素中嵌入的数字按照数字大小比较，而不是字母；
- integer: 元素先转换成整数，然后按照数字比较排序；
- real: 元素转换成 `float` 类型，然后排序；
- increasing: 按照升序进行排序；
- decreasing: 按照降序进行排序；
- index n: 被排序的列表中，每一个元素都是一个列表，按照每一个元素中第 `n` 的元素进行比较排序。这里 `n` 是整数，或者 `end-integer` 的形式；
- command proc: `proc` 是用来进行元素比较的回调函数，它必须带有两个参数，表示比较大小的两个列表元素；`lsort` 比较元素大小的时候，就自动调用这个函数。该函数由我们自己来定义，控制比较结果。该函数通过返回 `>0`, `=0` 和 `<0` 的结果，表示第一个参数大于、等于和小于第二个参数。
- unique: 该选项表示列表中所有相同的元素，排序之后，只保留第一个。

下面是 `-ascii` 和 `-dictionary` 的比较：

```
% lsort -ascii {x9y x10y x11y}
x10y x11y x9y      ;#ascii 中，嵌入的数字也被当作字符进行比较
% lsort -dictionary {x9y x10y x11y}
x9y x10y x11y      #dictionary 中，输入当作数字比较
% lsort -dictionary {bigBoy bigbang bigboy}
bigbang bigBoy bigboy      ;#忽略大小写
% lsort -ascii {bigBoy bigbang bigboy}
bigBoy bigbang bigboy      ;#ascii 中，大小写敏感
```

选项 `-index n` 可以看成是 `-command` 的一种简化形式，何出此言？看看下面的例子！

```
#=====
# File - list compare with -index option and -command
# Author   - Leiyuhou
# Created  - 2004/12/26
#=====

set tmp {{First 24} {Second 18} {Third 30}}
```

```
set t [lsort -integer -index 1 $tmp] ;#使用-index 选项
puts $t

proc OnCompare {a b} {
    set first [lindex $a 1]
    set sec    [lindex $b 1]

    return [expr $first-$sec] ;#直接相减作为结果
}

set t [lsort -command OnCompare $tmp]
puts $t
```

使用-index 选项，一句话就行了，使用-command，还需要定义一个回调函数。但是使用-command 选项有一个很大的优点，就是我们可以完全控制排序的方式。比如我们在本章开头的例子中提到的：分别以“总时间”、“次数”和“名字”作为第一、二、三关键字来排序。这个时候，-command 选项就派上用场了。

## 查找

TCL 中列表的查找功能强大的一塌糊涂。但都是通过一个 lsearch 命令来完成，其格式如下：

`lsearch ?options? list pattern`

该命令在列表 list 中查找能够和模式 pattern 匹配得上的元素。如果找到了，就返回该元素的下标。如果没有找到，那么返回-1。选项参数 options 用来控制查找过程，常见的如下：

-all：以列表形式返回所有匹配元素的下标。如果没有该选项，那么只返回最先匹配上的一个元素的下标；

-ascii：查找匹配时，采用 ascii 方式比较（和前面 lsort 中的-ascii 类似）。该选项只能够和-exact 或者-sorted 选项一起使用；

-decreasing：表示列表是降序排列的，只能够和-sorted 一起使用才有意义；

-increasing：表示列表是升序排列的，只能够和-sorted 一起使用才有意义；

-dictionary：和 lsort 中的-dictionary 类似，指定查找时使用字典匹配；

-exact：表示查找匹配的时候，必须和 pattern 完全一致，才算查找成功；

-glob：表示 pattern 要按照 glob 解释，并且采用 glob 模式进行匹配；

-inline：指定 lsearch 命令应该直接返回匹配的元素，而不是下标；

-integer, -real：元素和模式解释成整数或者浮点数，然后进行比较匹配；



-not: 表示非匹配, 返回不能够匹配 pattern 的元素下标;

-regexp: 表示按照正则表达式来进行查找;

-sorted: 表示列表是排序过的, 如果没有指定-decreasing 或者-increasing 则默认为升序, 并且包含的都是-ascii 字符串; 通过指定-sorted, 查找效率会得到很大的提高!

-start index: 从下标 index 开始查找。

如果没有指定任何选项, 那么-glob 模式是默认模式。我们来看看下面的例子:

```
lsearch {a b c d e} c => 2
lsearch -all {a b c a b c} c => 2 5
lsearch -inline {a20 b35 c47} b* => b35
lsearch -inline -not {a20 b35 c47} b* => a20
lsearch -all -inline -not {a20 b35 c47} b* => a20 c47
lsearch -all -not {a20 b35 c47} b* => 0 2
lsearch -start 3 {a b c a b c} c => 5
```

## 取子列表

lrange 命令可以取出一个列表中的一段。其格式如下:

lrange list first last

返回的列表是参数 list 中从 first 开始到 last 结束的所有元素组成的列表, 包含 first 和 last。一个特殊的情况就是 first 和 last 相同, 这时它和 lindex list first 返回的结果实际上还不是一回事! 请看下面的例子:

```
% set city      =>Shanghai Nanjing Wuhan Shenzhen
% lrange $city 1 end-1  =>Nanjing Wuhan
% lrange $city 2 2      =>Wuhan
% lindex $city 2        =>Wuhan
```

看起来是一样的, 但是实际上, lrange list first first 返回的结果和 list [lindex list first]命令返回的结果才是一致的。lrange 返回的总是一个列表。

## lset: 更改某个元素的值

前面多次提到了, Tcl 中列表是多个元素的有序集合。如果我们想修改列表中某一个元素的值, 应该怎么办? 以前的 Tcl 中还真的没有专门的命令来完成这样简单的工作, lreplace 倒是一个选择! 这一点是让很多 Tcl 新手觉得非常不爽的地方: 想一想 Python:

```
a = [1,2,5,4]
```

```
a[2] = 3
```

简单又直观，和 C 语言的数组或者 C++ 中 STL 的 vector 操作看起来都一样。但是 Tcl 就只能使用 lreplace 了，如下：

```
set a [list 1 2 5 4]
set a [lreplace $a 2 2 3]
```

Oh, My God!! 是可忍孰不可忍？不过 Tcl8.4 版本中出现了一个新命令 lset，可以完成更改列表元素值的任务，了却了我们的的心愿。其格式如下：

```
lset varname ?index...? newValue
```

参数 varname 是一个 TCL 列表的变量名，可选参数 index 是下标，newValue 是新值。这里 index 可以有 0 个或者多个，多个参数时其意义和 lindex 的多个 index 意义一样，表示子列表中的下标。下面是一个例子：

```
% set a {{first 100 1} {second 200 2} {Third 300 3}} ;set b $a
{first 100 1} {second 200 2} {Third 300 3}
% lset a Hello      ;#这里没有 index 参数
Hello
% set a $b          ; lset a {} Hello      ;#这里也没有 index 参数
Hello
% set a $b          ; lset a {2 2} 400     ;#修改第二个元素的第二个元素。
{first 100 1} {second 200 2} {Third 300 400}
```

要注意的是，lset 只能更改列表元素的值，而不能更改列表的元素个数。并且只能在 8.4 以上版本中使用。

## 数组

TCL 中的数组是元素的无序集合，每一个元素都有一个名字。实际上，TCL 中的数组和其他语言中一些对象或者类型类似：VBScript 中的 dictionary 对象、ML 中的 record、以及 Python 中的 dict 对象类型。

## 创建和使用数组

TCL 创建数组的方式很简单，直接调用 set 命令即可，如下：

```
set arrayname(name) value
```

arrayname 就是数组变量的名字，name 则是新元素的名字，value 则为值，具体操作如下：

1. 如果该数组还不存在，那么就创建该数组变量；
2. 如果数组中名字为 `name` 的元素不存在，那么就向增加名字为 `name` 的元素，值为 `value`；
3. 如果名字为 `name` 的元素也存在，那么就更改其值为 `value`；

例如：

```
set city(Wuhan) "Hubei"           ;#创建数组 city，增加 Wuhan 元素
set city(Shenzhen) "Guangdong"    ;#增加 Shenzhen 元素
set city(Gui\ lin) "Guangxi"       ;#元素名字中的空格要注意
set city(Guangzhou) "Guangdong"

puts [array get city]
```

如果数组中存在很多元素，那么多次调用 `set` 就非常繁琐。使用 `array set` 可以避免这种情况：

`array set arrayname list`

其中参数 `arrayname` 是数组名字，`list` 则是一个列表，格式为 `{name1 val1 name2 val2...}`，例如创建上面同样的数组：

```
array set city {Guangzhou Guangdong "Gui lin" Guangxi Shenzhen Guangdong Wuhan
Hubei}
```

数字 `city` 如果不存在，那么就创建它；如果里面某一个元素已经存在，就更新其值；否则就创建它。注意后面的列表，元素个数必须是偶数个。

使用数组和使用普通变量没有太大的差别，例如上面的数组 `city`：

```
set cityname "Guangzhou"
puts "City $cityname lies in $city($cityname)" ;#找出其中名字为 Guangzhou 元素的
值；
```

代码运行结果：

```
City Guangzhou lies in Guangdong
```

使用 `$arrayname(elementname)`，可以获得元素中指定名字元素的值。如果没有 `elementname` 的元素，那么 TCL 会抛出异常！

如何判断数组中元素个数？使用命令 `array size arrayname`，例如：

```
% array size city           ;#结果为 4。
```

## 删除数组元素

如何判断是否存在一个数组变量？答案是命令 `array exist arrayname`，例如：

```
% array get city      ;#数组 city 存在
Guangzhou Guangdong {Gui lin} Guangxi Shenzhen Guangdong Wuhan Hubei
% array exist city     ;#返回 1，表示 city 数组存在
1
% unset city          ;#删除掉整个数组！
% array exist city     ;#再次查询，返回 0，表示不存在。
0
%
```

上面例子中可以看到，删除整个数组的方法和删除普通变量一样，调用 `unset` 即可。删除数组中元素，可以直接 `unset`，也可以调用 `array unset`。请看代码：

```
% array get city
Guangzhou Guangdong {Gui lin} Guangxi Shenzhen Guangdong Wuhan Hubei
% unset city(Guangzhou) city(Shenzhen)      ;#删除两个元素
% array get city
{Gui lin} Guangxi Wuhan Hubei
% array set city {{Gui lin} Guangxi Guangzhou Guangdong Wuhan Hubei Shenzhen
Gua
ngdong}
% array unset city G*      ;#删除所有名字以字符 G 开始的元素
% array get city
Shenzhen Guangdong Wuhan Hubei
%
```

`unset` 在删除单个元素的时候有优势，写起来方便。但是 `array unset` 在删除多个元素的时候要方便一些。尤其是根据通配符来删除名字的时候。其语法如下：`array unset arrayname ?Pattern?`

其中 `pattern` 是元素名字通配符，匹配该通配符字符串的元素被删除。如果省略该参数，就删除整个数组。如果通配符为`*`，那么就删除数组中所有元素，但是数组变量仍然存在。例如：

```
% array unset city *      ;#删除其中所有的元素
% array exist city        ;#数组变量仍然存在，只不过是一个空数组
1
% array unset city        ;#删除整个数组变量
```

```
% array exist city          ;#不再存在
0
```

array unset 命令中的 pattern 参数, 语法符合 string match 的模式, 也就是所谓的 glob 模式。

## 查找元素

如何判断某些元素在数组中是否存在? 前面我们已经接触到这个命令了, 就是 array get, 其命令语法如下:

```
array get arrayname ?pattern?
```

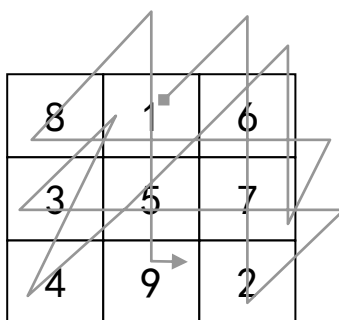
看起来和 array unset 很类似。可选参数 pattern 是模式字符串, 必须符合 glob 语法。凡是数组中名字和 pattern 能够匹配得上的元素, 都会返回。命令返回是一个列表, 是名字和值的列表。例如:

```
% array set city {"Gui lin" Guangxi Guangzhou Guangdong Wuhan Hubei Shenzhen
Guangdong}
% array get city          ;#找出 city 中所以元素
Guangzhou Guangdong {Gui lin} Guangxi Shenzhen Guangdong Wuhan Hubei
% array get city *        ;#和上面类似:
Guangzhou Guangdong {Gui lin} Guangxi Shenzhen Guangdong Wuhan Hubei
% array get city G*       ;#找出所有名字为 G 开始的元素
Guangzhou Guangdong {Gui lin} Guangxi
% array get city "Guangzhou" ;#查找元素 Guangzhou
Guangzhou Guangdong
% array get city "guangzhou" ;#不包含元素 guangzhou
%
```

可以看到, array get 命令所返回的列表, 和 array set 所需要的列表, 格式是完全一致的。

## 多维数组?

在 TCL 中提起多维数组是一个挺有趣的话题。因为一般的程序设计语言中, 数组是多个元素的有序集合, 但是 TCL 中数组则是元素的无序集合, 连顺序都谈不上, 如何谈得上维度? 但是 TCL 中却有一个巧妙的方式实现多维数组, 如下是一个简单的按照“规则摆数法”计算奇数阶幻方的 TCL 程序, 使得每一行, 每一列, 每一对角线上的所有数字相加都相等! 例如:



“规则摆数法”规则如下：先把 1 放在第一行中间，然后始终沿着右上角方向摆下一个数；如果碰到顶，就折向底；如果碰到右边，则折向左边，如果下一个地方已经有数或者碰上右上角，就退回到前一个方格的下方。具体走向如上图所示。下面是程序：

```
#=====
# File - 使用二维数组来实现 2n+1 阶的幻方
# Author - Leiyuhou
# Created - 2004/12/26
#=====

set N 7 ;#幻方的维度
for {set i 1} {$i<=$N} {incr i} {
    for {set j 1} {$j<=$N} {incr j} {
        set magic($i,$j) 0 ;#这里初始化二维矩阵
    }
}

set curx 1 ;#当前行号
set cury [expr {$N / 2 + 1}] ;#当前列号
set magic($curx,$cury) 1 ;#把 1 摆放在第一行中间

#过程 NextStep，将数字 n 拜访在矩阵 magic 中
proc NextStep { n } {
    global curx cury magic N
    set next_x [expr $curx-1] ;#计算下一步的位置
    set next_y [expr $cury+1]

    if { $next_x==0 && $next_y>$N } { ;#到了右上角
```

```

        incr curx
    } elseif { $next_x==0 } {      ;#到顶了
        set curx $N
        set cury $next_y
    } elseif { $next_y>$N } {      ;#到右端了
        set curx $next_x
        set cury 1
    } elseif { $magic($next_x,$next_y)!=0 } { ;#已经有数了
        incr curx
    } else {
        set curx $next_x
        set cury $next_y
    }

    set magic($curx,$cury) $n      ;#将数字 n 摆放到 curx 和 cury 中
}

#摆放 2-N*N 这几个数字
for {set i 2} {$i<=$N*$N} {incr i} {NextStep $i}

for {set i 1} {$i<=$N} {incr i} {
    set line ""
    for {set j 1} {$j<=$N} {incr j} {
        append line [format "%-4d" $magic($i,$j)] ;
    }
    puts $line      ;#打印结果
}

```

下面是执行结果，7 阶的幻方！

```

30  39  48  1   10  19  28
38  47  7   9   18  27  29
46  6   8   17  26  35  37
5   14  16  25  34  36  45
13  15  24  33  42  44  4
21  23  32  41  43  3   12
22  31  40  49  2   11  20

```

可以看到，我们使用 `magic(1,2)` 这样的形式来实现 TCL 中多维数组。事实上，TCL 并不理会数组是否多维，只是我们人为的把这样的形式理解成多维而已。在 TCL 看来，任意数组，都是一样的，不过是一个普通的字符串。所谓“多维”只存在于我们的脑袋里。我们用逗号分隔几个下标维度，如果你喜欢，你可以使用分号、冒号等等字符来分隔。

除此之外，我们还可以使用多阶列表来实现多维数组。自从 Tcl8.4 版本中引入 `lset` 命令之后，多阶列表的用处就有了极大的扩展。你可以使用二维列表来改写上面的程序。



## 正则表达式

首先要说的是，正则表达式本身就能够写成一本书，而且这样的书已经存在了。正则表达式是一种匹配模式，用来匹配部分特定的字符串。正则表达式在脚本语言中使用非常广泛，对于 TCL 脚本有其如此，因为 TCL 中一切都是字符串，而正则表达式是处理字符串的利器。TCL 中处理正则表达式的命令有两个：

**regexp** *?switches? exp string ?matchVar? ?subMatchVar subMatchVar ...?*

**regsub** *?switches? exp string subSpec ?varName?*

regexp 用来对字符串进行正则表达式的查找匹配；regsub 则是对字符串进行正则表达式的查找和替换操作。下面是两个例子：

```
% regexp -inline {0x[a-zA-Z0-9]+} "int a = 0x22FF;"
0x22FF
% regsub -all {(.)=(.)} "a=100" {\2=\1}
100=a
%
```

第一个例子，找出其中所有的十六进制数值；第二个，找出等号字符前后的串，并且前后互换。可以看到，这里最重要的就是正则表达式的语法。下面我们结合例子一一讲述。

## 正则表达式语法

正则表达式有着严格的语法规则，总结如下三条：

1. 正则表达式是用管道线符号“|”连接的一个或者多个分支（branch）组成；只要字符串能够匹配上其中任意一个分支，就算匹配整个表达式；
2. 每一个分支是由零个或多个约束（constraints）或者量化修饰的原子（atom）顺序连接而成；
3. 每一个原子包含了一个基本原子，后面跟着 0 个或者多个数量修饰串；如果没有数量修饰，那么就表示匹配一个原子；

## 数量限定

数量限定符号跟在原子后面，表示匹配前面的原子重复次数。可以有如下几种：

1. \*：表示匹配前面零个或者多个原子，等价于{0,}；
2. +：表示匹配前面的一个或者多个原子，等价于{1,}；
3. ?：表示匹配前面的零个或者一个原子，等价于{0,1}；

4. {m}: 表示匹配前面原子恰好 m 次重复;
5. {m,}: 表示匹配前面原子至少 m 次重复;
6. {m,n}: 表示匹配前面原子在 m 和 n 次之间重复, 包括 m 次和 n 次;
7. \*?、+?、??、{m}?, {m,}?, {m,n}?: 表示非贪婪匹配。默认情况下正则表达式是贪婪的, 往前尽量多的匹配; 在数量限定后面加上?之后, 表示尽量少的匹配。

下面是几个简单的例子:

```
% regexp -inline -all {ba?c} {bc bac baaac}    =>bc bac
% regexp -inline {ba*c} {bc bac baaac}          =>bc
% regexp -inline -all {ba*c} {bc bac baaac}      =>bc bac baaac
% regexp -inline -all {ba+c} {bc bac baaac}      =>bac baaac
% regexp -inline -all {ba{2,}c} {bc bac baaac}   =>baaac

#下面是贪婪和非贪婪的差别:
% regexp -inline -all {ba{1,}?} {bc bac baaac}  =>ba ba
% regexp -inline -all {ba{1,}} {bc bac baaac}    =>ba baaa
%% regexp -inline -all {ba|a+} {bc bac baaac}    =>ba ba aa
```

可以看到, `ba*c` 可以匹配 `bc`、`bac`、`baaac` 等; `ba+c` 则能匹配 `bac`、`baac` 等; 非贪婪则是尽可能少的进行匹配。“|”的优先级是最低的, 例如 `abcd+`, 匹配 `abc` 或者 `d`、`dd`; 但是不匹配 `abd`。

## 原子

原子有如下几种格式:

原子	含义
(re)	用括号括起来的一个正则表达式, 匹配 re。用来记录子串, 作为报告结果;
(?:re)	意义和(re)类似, 匹配 re, 但是不被记录作为子串来报告结果;
()	匹配一个空字符串; 记录下来作为报告结果
(?:)	意义同上, 匹配空串, 但是不被记录为子串;
[chars]	方括号表达式, 下面有详细的论述;
.	匹配任意的单个字符;
\k	k 是一个非数字和字母的字符; 表示原始字符, 例如\\表示反斜线;
\c	c 是字母或者数字, 表示转义符; 例如\d表示所有的数字;
{	后面跟数字, 表示数量限定; 否则表示字符{本身;
x	单个字符, 表示匹配这个字符本身;

例如:

```
% regexp -inline -all {[a-z]+} ([0-9]+) {a = 1000;b = 200}
{a = 1000} a 1000 {b = 200} b 200
```

上面的正则表达式([a-z]+) = ([0-9]+)。分别取出字符串中的赋值语句字符串，并且取出变量名和值；这里用到了简单的方括号语法[a-z]表示所有的小写字母，(re)也用到了，大家看到变量名和初始值都放在括号中，作为子串。

## 约束符号

约束符号用来进一步限定正则表达式的匹配。有如下几种：

^：表示匹配一行的开始；

\$：表示匹配一行的结束；

(?=re)：表示正的前项探索（positive lookahead），如果后面的串能够匹配 re，那么整个串就能够匹配上；

(?!re)：表示负的前向探索（negative lookahead），如果后面的串不能匹配 re，那么整个串就匹配成功；

\A：表示匹配一行的开始；它和^有着细微的不同；

\m：只匹配一个单词的开始位置；

\M：只匹配一个单词的结束位置；

\y：只匹配一个单词的开始或者结束位置；

\Y：匹配那些既不是单词的开始也不是结束的位置；

\Z：匹配一行的结尾，和\$类似，但是也存在不同；

\m：m 是一个非零数字，表示后向引用前面已经匹配的子串；

举一个前向探索的例子：如果我们需要匹配一个字符串中的“Windows”，但是只有在后面跟着 98 或者 2000 时才匹配，如果是其他则不匹配。那么就可以这样写：

```
% regexp -inline -all {Windows +(?=98|2000)\S+} "Windows 98 Windows xp Windows 2000"
{Windows 98} {Windows 2000}
% regexp -inline -all {Windows +(?!98|2000)\S+} "Windows 98 Windows xp Windows 2000"
{Windows xp}
```

第一个命令找出了两个 Windows，第二个则只匹配上了一个，就是 Windows xp 中的 Windows。

再举一个后向引用的例子：Html 文件中有<b>...</b>这样的字符串，如果我们要找出这样的串，可以这样写：

```
% regexp -inline -all {<([^\> ]+)>(.*)</\1>} "abc <h>Title1</h>"
```

```
<h>Title1</h> h Title1
```

这里的正则表达式中就使用\1 来表示后向引用，<([<^> ]+)>中括号匹配的字符串。所以上面例子中的字符串可以被匹配，但是<h>Title</b>则不能被匹配。

## 方括号表达式

方括号用来表示某些字符中的一个字符，例如[abc]匹配 abc 三个字母中的任意一个字母。[abc]+可以匹配 a, aacb, abc 等字符串，但是不能匹配 aec。它有几种表示方法：

1. [chars]: chars 是一些字符的列表，匹配这些字符中的任意一个；
2. [x-y]: x 和 y 是两个字符，那么表示匹配落在[x,y]区间内部的任意一个字符，例如[a-z]匹配所有的小写字母；[a-zA-Z0-9]则表示所有的字母和数字。
3. [^chars]: 表示非的意思；例如[^0-9]表示所有非数字的字符。
4. 特殊记法，表示某类字符，例如[:digit:]，等价于[0-9]

如果要在括号中使用“-”字符、“[”或者“]”字符，可以把它放在“\”后，例如\[ \]。下面我们看几个例子：

```
% regexp -inline {[+0-9]+} -100      ;#会出错
couldn't compile regular expression pattern: invalid character range
% regexp -inline -all {[+0-9]+} "-100,+100"    ;#把-放到方括号中作为第一个字符；
-100 +100
```

方括号还可以表示某一类的字符，例如[:alpha:]表示字母，[:upper:]表示大写字母。所有类别字符列举如下：

类别名字	含义
alpha	表示一个字母，等价于[a-zA-Z]
upper	大写字母，等价于[A-Z]
lower	小写字母，等价于[a-z]
digit	数字，等价于[0-9]
xdigit	十六进制数字，等价于[0-9a-fA-F]
alnum	字母或者数字，等价于[0-9a-zA-Z]
print	字母或者数字，等价于[:alnum:]
blank	空格字符或者 Tab 字符
space	显示为空格的字符

例如下面找出字符串中的十六进制数字：

```
% regexp -all -inline {0[xX][[:xdigit:]]+} "int a=0x3eF0"
0x3eF0
% regexp -all -inline {0[xX][[:digit:]]a-fA-F)+} "int a=0x3eF0"
```

0x3eF0

其中两个命令中的正则表达式是等价的。

## 反斜线

正则表达式中的反斜线的意义和普通 TCL 字符串中的反斜线含义一致，但是下面的反斜线及字符具有特殊的含义：

`\d`：表示数字，等价于`[0-9]`；`\D`：表示非数字字符，等价于`[^0-9]`。

`\s`：表示空格字符，等价于`[[:space:]]`；`\S`：表示非空格字符，等价于`[^[:space:]]`。

`\w`：表示字母或者数字或者下划线，等价于`[[:alnum:]]`；`\W`：等价于`[^[:alnum:]]`。

其中，`\d`、`\s`和`\w`可以用在方括号内，但是`\D`、`\S`和`\W`则不行，例如：

```
% regexp -all -inline {[\d]+} "int a=3450"
3450
% regexp -all -inline {[\D]+} "int a=3450"
couldn't compile regular expression pattern: invalid escape \ sequence%
```

## 元语法

正则表达式中还可以嵌入一些控制字符，用来控制正则表达式的一些匹配细节。这种记法叫做元语法（metaSyntax），具体语法格式为`(?xyz)`，其中`xyz`是控制字符。常见的控制字符有：

`c`：大小写敏感匹配，这一般是默认设定；

`i`：大小写不敏感；

`n`：换行敏感的匹配；

`p`：局部换行敏感的匹配；

`s`：换行不敏感的匹配；

`w`：翻转的局部换行敏感的匹配；

`t`：使用紧凑语法，这是一般的默认设定；

`x`：使用正则表达式扩展语法；

这里面需要强调的是`t`和`x`，其中`t`表示默认的紧凑语法，一般情况下不需要指定；`x`表示扩展语法，和紧凑语法的区别如下：

1. 所有的空格字符被忽略；
2. 所有在`#`字符和换行符之间的内容被忽略，所以可以在正则表达式中嵌入注释；
3. 但是，反斜线`\`后面的空格和`#`字符不被忽略；方括号中间的空格以及`#`不被忽略；并且在`(?:`这样的语法单元中的空格或者注释是非法的。同时，`(?#xxx)`表示一个嵌入

式的注释。

例如：

```
set c [regexp -inline -all {***:(?x)
    (int|long)      #this is type
    \s+            #the blanks
    ([a-zA-Z_]\w*)  #name of variable
    \s*=(?#COMMENT,the equal)\s*
    ([0-9]+)        #这是变量的初始化值
} {int age=2000} ]
puts $c
```

执行结果如下：

```
{int age=2000} int age 2000
```

上面的正则表达式就是要从“int age=2000”这一个字符串中解析出类型，变量名和初始化值。其中黑体的正则表达式中：

1. 开始的\*\*\*:表示这是一个 ARE。TCL 中的正则表达式默认是 ARE，所以咱们一般可以省略它；
2. 后面的(?x)是一个元语法，表示后面的表达式采用扩展语法；
3. 后面每一行最后都有一句注释，用#开始；表示这一行的含义；
4. 可以看到(?#COMMENT,the equal)，这是嵌入式的注释；

正则表达式要是写长了，可能连作者自己都搞不明白当初这样写是什么意思！所以在其中加入注释还是有必要的。使用了(?x)指定扩展语法之后，就可以加入注释了！

元语法中的指令，在 `regexp` 等命令中有对应的命令行参数。例如 `x`，对应的选项就是 `-expanded`。具体请参考 `regexp` 的帮助。

## 如何匹配换行

如何在正则表达式中匹配换行，是一个让人困惑的问题。比如正则表达式中既有`^`和`$`来表示行开始和结束，但是也有`\n`来表示换行，还有`\A``\Z`等都是什么意思？有时为什么`^`和`$`不管用，例如：

```
set c [regexp -inline -all {(int|long).*=D*(^[0-9]+$)} {
int   age   =
2000
} ]
puts "s = $c"
```

上面正则表达式中的黑体部分，我们想匹配下面的字符串 2000，并且要求这一串数字正号自成一（^和\$表示行头和行尾）；虽然后面被匹配字符串中的 2000 满足这一要求，但是执行结果却表示没有匹配上。我们把正则表达式中黑色部分的^和\$去掉，却能够匹配成功。实在让人困惑！！

要回答上面的问题，必须先明白正则表达式中的换行模式。TCL 中存在四种换行模式：

1. 换行不敏感模式：这是默认的模式，这种模式下，被匹配字符串中的回车换行符号没有任何特殊意义。字符.以及表达式[^...]可以匹配任意字符，包括换行字符；可以使用元语法(?s)来指定该模式；
2. 换行停止模式：这种模式和第一种模式的区别在于，表达式[^...]和字符.能够匹配换行符之外的其它任意字符；一旦碰到换行符，匹配会停止；可以通过(?p)来指定；
3. 换行停靠模式：和第一种模式的差别在于，字符^和\$的意义改变了，分别用来匹配一行的开头和结尾；可以通过(?w)来指定；
4. 换行敏感模式：是上面 2 和 3 两种模式的综合。^和\$匹配一行的开头和结尾；.以及[^...]不能够匹配合换行符；可以通过(?n)来指定；

明白了上面四点之后，问题迎刃而解：前面的正则表达式采用的是换行不敏感模式，所以表达式中的^和\$都是普通字符，因为在被匹配字符串中没有这样的字符，所以匹配失败。

下面再看看一个例子：

```
set dststring {
int    age    =
2000
}

set regs {
  {(?s)(int|long)[^0-9]+([0-9]+)}
  {(?n)(int|long)[^0-9]+([0-9]+)}
  {(?n)(int|long)[^0-9]+\n([0-9]+)}
  {(?n)(int|long)[^0-9]+\n^([0-9]+)$}
  {(?w)(int|long)[^0-9]+^([0-9]+)$}
}

foreach re $regs {
  set c [regexp -inline -all $re $dststring]
  puts "-----\n$c"
}
```

上面的代码执行结果如下，1 表示匹配成功，0 表示匹配失败

```
-----(?s)(int|long)[^0-9]+([0-9]+)----- : 1
```

```
-----(?n)(int|long)[^0-9]+([0-9]+)----- : 0
-----(?n)(int|long)[^0-9]+\n([0-9]+)----- : 1
-----(?n)(int|long)[^0-9]+\n^([0-9]+)$----- : 1
-----(?w)(int|long)[^0-9]+^([0-9]+)$----- : 1
```

为什么会出现各个结果，请大家自行分析。最后要说的就是\A 和\Z。很简单，它们分别匹配被匹配字符串的开头和结束。例如：

```
set dststring {
int    age    =
2000
}

set regs {
    {(?s)\A\n(int|long)[^0-9]+([0-9]+)\Z}
    {(?s)\A\n(int|long)[^0-9]+([0-9]+).*\Z}
}

foreach re $regs {
    set c [regexp -inline -all $re $dststring]
    puts "-----$re----- : [expr [llength $c]>0]"
}
```

上面代码执行结果如下：

```
-----(?s)\A\n(int|long)[^0-9]+([0-9]+)\Z----- : 0
-----(?s)\A\n(int|long)[^0-9]+([0-9]+).*\Z----- : 1
```

第一个表达式，\Z 表示字符串结束，但是因为[0-9]+匹配 2000 之后，后面还有一个\n 换行符没有匹配，所以\Z 就没有匹配上，整个表达式匹配失败。第二个表达式，在\Z 前面加上了.\*用来匹配回车符号，所以匹配成功！

## 正则表达式匹配查找

TCL 中使用命令 `regexp` 来完成正则表达式的查找匹配操作，我们上面也看到了它的简单用法，其完整语法如下：

```
regexp ?switches? exp string ?matchVar? ?subMatchVar subMatchVar ...?
```

命令可以有一个或者多个开关选项，用来控制命令的具体行为。第一个参数是我们用大量篇幅描述的正则表达式，第二个参数则是需要被匹配的字符串；后面的变量名是可选参数，用来接收匹配到的字符串以及各个子串。



命令一般情况下返回 1 或者 0，表示匹配成功或者失败。看看例子：

```
% regexp {(\S+)\s*=\s*(\d+)} {a=100;b=200}      => 1
% regexp {(\w+)\s*=\s*(\d+)} {a=100;b=200} v      => 1
% puts $v      => a=100
% regexp {(\w+)\s*=\s*(\d+)} {a=100;b=200} v a b      => 1
% puts "$v | $a | $b "      => a=100 | a | 100
```

命令有以下几个选项：

**-expanded**：表示后面的正则表达式采用扩展语法，等于在表达式中加入(?x)元语法；

**-line**，**-linestop**，**-lineanchor**：分别表示换行敏感、换行停止、换行停靠三种模式，分别等同于如下三种元语法：(?n)，(?p)，(?w)。如果三个都不指定，那么就是换行不敏感，等同于(?s)；

**-all**：在 string 中使用正则表达式进行尽量多的匹配，返回发现匹配的次数；如果命令中指定了结果变量，那么只有最后一次匹配的结果被放入到变量中。例如：

```
% regexp -all {(\w+)\s*=\s*(\d+)} {a=100;b=200} v a b      => 2
% puts "$v | $a | $b "      => b=200 | b | 200
```

**-inline**：命令返回一个列表；如果指定了该选项，那么 **regexp** 后面就不能给出结果变量名。并且把匹配结果字符串以及子串作为结果返回，例如：

```
% regexp -inline {(\w+)\s*=\s*(\d+)} {a=100;b=200} v a b
regexp match variables not allowed when using -inline
% regexp -inline {(\w+)\s*=\s*(\d+)} {a=100;b=200}
a=100 a 100
% regexp -inline -all {(\w+)\s*=\s*(\d+)} {a=100;b=200}
a=100 a 100 b=200 b 200
```

**-indices**：该选项用来更改 **subMatches** 的值类型，没有该选项时结果变量是字符串，有该选项后是列表，表示对应字符串的开始和结束字符的位置。例如：

```
% regexp -inline {(\w+)\s*=\s*(\d+)} {a=100;b=200}
a=100 a 100
% regexp -inline -indices {(\w+)\s*=\s*(\d+)} {a=100;b=200}
{0 4} {0 0} {2 4}
% regexp -indices {(\w+)\s*=\s*(\d+)} {a=100;b=200} v a b
1
% puts "$v | $a | $b"
0 4 | 0 0 | 2 4
```

## 正则表达式匹配替换

先从一个实际的问题说起。假如存在如下的文件，每一行都是一个记录，第一个字段是员工名字，第二个是工号，字段之间用空格隔开，如下：

```
王涛      1202212
李四      1303342
```

.....

现在要把内容变化一下，每一行变成名字+(工号)，如下：

```
王涛(1202212)
李四(1303342)
```

这个问题如何解决？嗯？使用 UltraEdit 等编译器打开后逐行的手工更改？这是最累的方法。事实上使用正则表达式可以非常简单的办到，代码如下：

```
set str {
王涛      100001
李四      100002
张三      100003
}

regsub -all -lineanchor {^\(S+\)s+(\d+)$} $str {\1(\2)} dst
puts $dst
```

执行结果如下：

```
王涛(100001)
李四(100002)
张三(100003)
```

命令 `regexp` 的语法如下：

```
regsub ?switches? exp string subSpec ?varName?
```

其中 `switches` 和命令 `regexp` 的选项类似，`exp` 是用来匹配的正则表达式，`string` 则是被替换的字符串，`subSpec` 是用来替换匹配部分的字符串。该命令在参数 `string` 中使用正则表达式进行搜索，能够匹配 `exp` 的部分，则用 `subSpec` 来替换掉。如果没有指定 `varName`，那么命令把替换后的字符串作为结果返回，否则就把替换后的结果放到变量 `varName` 中。选项中要重点提到的是 `-all`，表示所有能够匹配的都要替换；否则只替换出现的第一个匹配项。

参数 `subSpec` 中可以包含一些具有特殊意义的串：`\n` 或者 `&`，其中 `n` 在 `[0-9]` 之间。如果是 `\0` 或者 `&` 表示匹配到的目的字符串，如果是 `\1` 至 `\9` 表示查找到的当前匹配项中的子串。例如上面的例子中。正则表达式是 `^\(S+\)s+(\d+)$`，其中第一个子串 `\(S+\)` 用来匹配名字，第

二个子串用来匹配工号。参数 subSpec 为\1(2)，表示匹配到的第一个子串紧跟括号括起来的第二个子串。如果第一次匹配上了“王涛 100001”，那么两个子串分别为“王涛”和“10001”，然后整个被匹配部分被“王涛(10001)”所替换到，从而达成我们的目的。

其实，正则表达式查找和替换是大部分文本编辑器具有的功能，例如著名的 vim 和 Emacs，以及 UltraEdit，VC++ 的编辑器等。下面是 UltraEdit 的替换操作对话框：



这些编辑器中，正则表达式的具体语法会有较大的差别。大家在使用这些工具的正则表达式的时候需要注意。



## 控制结构

TCL 中使用命令来实现程序控制结构，我们还可以自己编写扩展命令来扩充控制结构。常见的控制结构包括如下几种：

循环控制、条件判断、异常处理和执行脚本。这些结构控制命令都比较简单，和 C/C++ 中的程序控制比较类似。但是在介绍控制结构之前，先有必要介绍一下 Tcl 中的布尔类型。

## Boolean 类型

再次强调，TCL 中一切都是字符串，没有类型。这里所说的 Boolean 类型，指的是在条件判断的时候，什么东西被当作 True，什么被当作 False。请记住如下规则：

1. 字符串：Yes, Y, True, T, On。不管大小写，当作条件来判断时，会被认为是 True；
2. 字符串：No, N, False, F, Off。不管大小写，当作条件来判断时，会被认为是 False；
3. 数字 0（整数或者浮点）会被当作 False，任何非 0 的整数或者浮点数都会被当作 True；

例如：

```
% while 0.00e23 "puts ccc;break;"      ;#False
% while 0. "puts ccc;break;"           ;#False
% while 0.1 "puts ccc;break;"          ;#True
ccc
% while T "puts ccc;break;"            ;#True
ccc
% while 1023 "puts ccc;break;"         ;#True
ccc
% while T "puts ccc;break;"            ;#True
ccc
```

## 条件判断

TCL 中的条件判断主要是两个命令：if 和 switch。

## if...else

if 命令的格式如下：

**if** *expr1* **?then?** *body1* **elseif** *expr2* **?then?** *body2* **elseif** ... **?else?** *bodyN*?

可以看见，if 命令的参数 then 是可以省略的。这给喜欢 Pascal 风格和 C 风格的不同的编程人员，都带来了怀旧的机会。Pascal 中 then 是必须的。C 中是没有 then 的。

if 命令中的条件表达式 *expr* 可以写成多行，只要是一个字符串都可以。如果写成多行，那么最好加上 then，这样看起来就比较方便。

这个命令的具体执行我就不详细解释了。要注意的是最后的 else，如果没有 else，那么就不能有 bodyN，如果有了 else，就必须有 bodyN。请看例子：

```
if {$vbl == 1} {  
    puts "vbl is one"  
} elseif {$vbl == 2} {  
    puts "vbl is two"  
} else {  
    puts "vbl is not one or two"  
}
```

## switch

相比 if...else，switch 命令则要复杂很多。它有两种语法形式：

1. switch *?options?* string pattern body *?pattern body ...?*
2. switch *?options?* string {pattern body *?pattern body ...?*}

两种形式功能上完全等价，我们一般建议采用第二种，它和 C 语言的语法比较接近：所有的候选值和对应的代码全部都放在一个花括号中。但是如果需要对 pattern 和 body 等进行替换，那么第一个形式就比较方便。

switch 的选项参数可以为：

1. -exact：这是默认情况，表示参数 string 和模式 pattern 的精确匹配；
2. -glob：表示采用 glob 模式进行匹配。匹配方法参考 string match 命令；
3. -regexp：表示采用正则表达式进行匹配，string 是字符串，而 pattern 是正则表达式；
4. --：表示选项参数到此为止，后面的是参数 string。这是为了防止参数 string 的第一个字符就是“-”而引起非法选项的错误。

该命令的执行过程：根据选项指定的匹配模式，逐个将 string 和 pattern 进行匹配，如果匹配成功，那么就执行 pattern 后面对应的代码，并且返回该代码的结果。如果最后的一个模式是 default，那么它匹配任意字符串。如果 string 没有一个匹配成功并且没有 default，那

么 switch 直接返回空。

如果某一个 pattern 对应的 body 为“-”，那么表示这个 pattern 和下一个 pattern 共享一个 body。这和 C 语言中的 case 1:case2:{...} 比较类似。

## switch 中的-regex

请看下面的几个例子：

```
set r "abbbcc"
switch -regex $r {
    ab{2}c { puts "1" }
    ab{3}c { puts "2" }
    ab{3}c{2} { puts "3" }
}
```

上面采用正则表达式进行匹配，显然，第三个表达式才是和 \$r 完全匹配的。但是上面的代码执行输出是“2”。这给我们一个重要的启示：

switch 中的-regex 匹配选项，实质上就是把 pattern 拿出来在 string 中进行正则表达式查找，也就是执行 regex 命令，如果 regex 返回 1（也就是在 string 中找到了 pattern），那么就算匹配成功，就开始执行后面的 body。而根本不管 pattern 是不是恰好匹配了整个 string！

上面这一点务必牢记。

对-glob 选项，则没有这样的问题。选项 pattern 和 string 只有完全匹配的时候才算匹配成功。也就是说执行命令 string match \$pattern \$string 返回 1，才算匹配成功。

## 注释的位置

如果使用 switch 的第二种语法形式，那么就要注意 switch 中注释的正确位置。注释只有放在各个 body 中才是正确的注释，如果放在其它地方，会出现语法错误。例如：

```
set r "abbbcc"
switch -glob $r {
    a {
        #匹配失败，不可能执行到这里
        puts "1"
        #正确的注释
    }
    a*cc { puts "2" }
    #放到这里，是错误的注释
}
```

```
ab??cc {  
    puts "3"  
}  
}
```

我使用的是 8.4.9 版本，出现的错误信息还非常的人性化：

```
extra switch pattern with no body, this may be due to a comment incorrectly placed  
outside of a switch body - see the "switch" documentation  
  
while executing
```

会提醒你可能是注释放错了地方，以前的版本就没有这么好了。错误信息会让你不知所云。

## 循环控制

TCL 中使用 `while`、`for`、`foreach` 命令来实现循环控制结构。除了这三个命令之外，`tcl` 程序包 `control` 还提供了 `do` 循环。别忘了，前面的章节中，我们还自己实现了一个简单的 `do...while` 循环。

## while 循环

`while` 循环命令的语法格式如下：

```
while test body
```

其中 `test` 被当作一个表达式来求值（和 `expr` 命令一样），结果必须是一个合法的布尔值。当 `test` 的值为 `True` 的时候，循环体 `body` 会被执行；`body` 执行完之后，`test` 会被再次求值，直到 `test` 为 `False` 为止。`while` 命令总是返回一个空的字符串。

如果 `body` 中包含 `continue`，那么当前这一次循环迭代会被中止，然后直接计算 `test`，根据结果判断是否继续下一次循环；如果 `body` 中包含 `break`，那么整个 `while` 循环立刻全部中止。

前面的章节中我们介绍 TCL 置换原理的时候强调了，`while` 命令的 `test` 条件必须放在花括号中，这样每次循环结束后才对 `test` 进行真正的求值。如果不放在花括号中，就可能会出现死循环。

有些初学者会写出如下的代码：

```
set x 100  
while {[expr $x>=50 && $x<=100 ]} {  
    .....  
}
```



```
}
```

这样的代码错了吗？没有错误，但是不好。`while` 循环中的判断条件，我们没有必要写在 `expr` 命令中来求值，因为 `while` 命令会自动的进行 `expr` 计算。所以如下的代码更好：

```
while {$x>=50 && $x<=100} {.....}
```

## for 循环

Tcl 中的 `for` 循环和 C 中的类似，其语法如下：

```
for start test next body
```

其中 `start`, `next` 和 `body` 都是可以执行的 TCL 脚本；`test` 和 `while` 中的 `test` 类似被当作一个表达式来求值，其结果作为循环是否继续的判断标准。`for` 循环的执行过程如下：

1. 首先执行 `start`；
2. 计算表达式 `test`；
3. 如果 `test` 的值是 `True`，那么就执行 `body`，然后再执行 `next`；然后跳到第 2 步；
4. 如果 `test` 的值是 `False`，那么就停止循环；

如果在 `body` 或者 `next` 中执行了 `break` 命令，那么 `for` 循环马上中止；如果在 `body` 中执行了 `continue` 命令，那么 `body` 中剩下的命令全部跳过而继续执行 `next`，然后再计算 `next`，根据结果判断是否继续循环；这两个命令在 `for` 循环中的表现和 C 语言中的 `break` 以及 `continue` 类似。

一般而言，`test` 都必须放在花括号中，这一点和 `while` 循环类似。下面是 `for` 循环的例子：

```
for {set x 1} {$x<=1024} {set x [expr {$x * 2}]} {  
    puts "x is $x"  
}
```

上面的代码打印出从 1 开始到 1024 中所有的 2 的 `n` 次方。

## foreach 循环

`foreach` 提供了一种简单的在一个或者多个列表上进行迭代的方法。它有两种形式：

```
foreach varname list body
```

```
foreach varlist list1 ?varlist2 list2...? body
```

第一种语法中，只利用一个变量对列表 `list` 进行迭代，每一次循环，都顺序取一个列表元素的值赋给 `varname`，然后执行 `body`。

第二种语法中，可以同时多个列表进行迭代。在迭代的时候，可以一次取出一个列表

多个元素。实际上，第一中形式只是第二种语法的一个特例而已。看看下面的几个例子：

```
set x {}
foreach {i j} {a b c d e f} {
    lappend x $j $i
}
#执行后，x 的值为"b a d c f e"；总共完成了 3 次迭代。

#下面的迭代中，同时针对两个列表进行迭代：
set x {}
foreach i {a b c} j {d e f g} {
    lappend x $i $j
}
#执行后，x 的值为"a d b e c f {} g"；总共完成了 4 的迭代。其中最后一次迭代中，
第一个
#列表已经迭代完，所以变量 i 为空。

set x {}
foreach i {a b c} {j k} {d e f g} {
    lappend x $i $j $k
}
#执行后，x 的值为"a d e b f g c {} {}"；总共完成 3 次迭代。其中最后一次迭代中，
因为
#第二个列表已经迭代完毕，所以 j 和 k 都为空。
```

仔细分析完上面的例子，基本上就掌握了 `foreach` 的用法。`foreach` 中也可以使用 `break` 和 `continue`，其功能和 `for` 循环中完全一致。

`foreach` 除了上面的迭代功能之外，还可以有另外一个用途，就是把列表的元素顺序解析到各个独立的变量中。一般我们取列表元素都是采用 `index` 的方法，例如：

```
set a {"TigerLei" "26000" "Male" 27}
set name [lindex $a 0]
set id [lindex $a 1]
set sex [lindex $a 2]
set age [lindex $a 3]
.....
```

这样的代码不仅难看，而且执行效率不高。使用 `foreach` 就可以这样写：

```
foreach {name id sex age} $a {}
```

一条语句就可以完成上面的四条语句的功能。而且执行效率要高出很多。

## do 循环

这里介绍的 do 循环不是前面章节中我们自己实现的 do 命令，而是 tclLib 中的程序包 control 提供的一个 do 命令。还是先看例子：

```
package require control      ;#加载程序包 control
namespace import control::do  ;#从 control 中引入命令 do

set sum 0 ; set i 1
do {
    incr sum $i
    incr i
} while {$i<=100}      ;#这里使用 do ...while 循环来计算。
puts $sum

set sum 0 ; set i 1
do {
    incr sum $i
    incr i
} until {$i>100}        ;#这里使用 do...until 循环来计算。
puts $sum
```

Pascal 语言中存在 repeat...until 循环，这里的 do...until 与之类似；do...while 循环则与 C 中的 do 循环类似。要区分的是这两种 do 循环中，判断条件的写法。do...while 中，只要条件为真，就一直执行循环；do...until 中，条件为假则一直执行循环，直到条件为真。

## break、continue

break 和 continue 两个命令我们前面已经见识过了，主要在循环体中使用：

1. break 跳出最靠近它的循环体；
2. continue 结束当前这一次循环，继续下一次的循环；

除此之外，它们在非循环体中也可以使用，例如字符串置换命令 subst。break 和 continue 还可以用在 Tk 的事件响应函数中。下面是 subst 的例子：

```
subst {abc,[break],def}
#返回字符串：`abc,"，而不是`abc,,def"
```

```
subst {abc,[continue;expr 1+2],def}  
#返回字符串: ``abc,,def'', 而不是``abc,3,def''
```

实际上, `break` 命令抛出一个 `TCL_BREAK` 的异常; 而 `continue` 抛出一个 `TCL_CONTINUE` 的异常。这两种异常都可以被 `catch` 命令所捕获。例如:

```
catch {break}      ;#catch 命令返回 3  
catch {continue}   ;#catch 命令返回 4
```

## 异常处理

程序在执行的时候, 可能会发生一些错误导致执行过程被停止。为了更好的处理程序中发生的各种错误, TCL 中也引入了异常处理机制。如果读者了解 C++ 中的异常处理方法, 那么 TCL 的异常处理方式就不难理解。在 TCL 中使用 `catch` 来捕获异常, 使用 `error` 来抛出异常。

## catch 命令

首先看看 `catch` 命令的格式:

```
catch script ?varname?
```

第一个参数是需要执行的程序块, 第二个参数可选的, 表示变量名。如果脚本 `script` 在执行的时候发生了错误, 那么错误处后面的脚本就不会被执行; `catch` 命令会捕获该错误, 并且根据错误的具体类型返回不同的整数值。TCL 已经定义好了五个不同的值, 表示不同的情况:

- 0: 表示 `script` 执行的时候没有抛出任何错误, 内部表示为 `TCL_OK`;
- 1: 表示 `script` 执行的时候发生了错误, 内部表示为 `TCL_ERROR`。这个最常用;
- 2: 表示 `script` 执行的时候发生了 `TCL_RETURN` 的异常; 这一般是执行了 `return` 命令引起;
- 3: 表示发生了 `TCL_BREAK` 异常, 这一般是执行了 `break` 命令引起;
- 4: 表示发生了 `TCL_CONTINUE` 异常, 一般是执行了 `continue` 引起;

除了上面 5 个已经定义好的返回代码之外, 我们还可以扩充我们自己的异常类型代码。其中 `TCL_ERROR` 异常类型是最常用的异常类型。

如果给出了参数 `varname`, 那么 `catch` 返回后该变量保存 `script` 的执行结果; 如果 `catch` 没有捕获任何异常, 那么 `varname` 变量中保存的是 `script` 正常执行结果; 如果返回的是 `TCL_ERROR`, 那么保存的是错误信息。

如果 `script` 执行时没有发生任何错误, 那么 `catch` 返回 0, `varname` 中保留脚本执行的结果。要注意的是 `catch` 捕获一切异常和错误, 包括 `break`、`continue` 和 `return` 等命令。唯一不

能够捕获的是编译脚本过程中发现的语法错误。这一点和 C++ 的 `try...catch` 不大一样。C++ 中的 `try...catch` 只有在最后指定 `catch(...)` 的时候才捕获一切异常信息。

先看一个使用 `catch` 的一般的例子：

```
proc ReadAll {path} {
    set f 0      ;#初始化文件句柄
    set r [catch {
        set f [open $path r]    ;#打开文件
        set buf [read $f]       ;#读出全部内容
    } msg ]

    if { $f!=0 } {
        catch {close $f}        ;#关闭文件
    }

    if { $r } {
        return ""               ;#发生了错误
    }
    return $buf
}
```

可能有朋友会问了，为什么不把上面代码中的 `close $f` 也放到第一个 `catch` 语句块中来，而专门拿到外面来进行关闭文件呢？答案是，如果把 `close $f` 拿到上面的 `read $f` 后面，那么有可能文件打开了之后不会被关闭。假如 `open` 打开文件正确，但是 `read` 出现了异常，那么后面的 `close` 就不会被执行。程序中的异常是任何一个程序员都需要重点考虑的问题，有人做过统计，一个大型系统中可能一半左右的代码都是在进行各种各样的错误和异常处理。

## 神奇的 return

TCL 中的异常处理和 `return` 命令紧密相关。`return` 最简单的用途就是从过程中返回，这一点和 C 语言中的 `return` 并无不同。但是除此之外，还有一些神奇的用法。我们再看另外的一个例子：

```
#这个过程实现自己的控制结构，根据 flag 的不同返回不同值，实现不同控制结构
proc MyControl { flag } {
    switch $flag {
        break {      return -code break      }
        continue { return -code continue    }
        return {     return -code return     }
    }
}
```

```
        error {      return -code error -errorinfo "A little Error!"  }
        goto {       return -code 100 GOTO   }
        default {    return -code ok $flag   }
    }
}

foreach ctl {break continue return error goto normalReturn} {
    set r [catch {
        MyControl $ctl    ;#调用每一个控制结构看看异常结果:
    } msg ]

    puts "-----$ctl : $r -----"
    puts "$msg"
}

set i 0;set sum 0          ;#下面累加 1..100 之间的偶数
while {$i<100} {
    if { ([incr i]%2)!=0 } {
        MyControl continue    ;#等同于直接调用 continue 命令
    }
    incr sum $i
}

puts "2+4+6+...100 = $sum"
```

下面是执行结果:

```
-----break : 3 -----
-----continue : 4 -----
-----return : 2 -----
-----error : 1 -----
-----goto : 100 -----
GOTO
-----normalReturn : 0 -----
normalReturn
```

```
2+4+6+...100 = 2550
```

上面的过程 `MyControl` 根据不同的 `flag` 参数执行不同的操作。`MyControl continue` 等同于直接调用 `continue` 命令，`MyControl return` 等同于在调用 `MyControl` 的上下文中直接调用 `return`。可以看到我们自己定义了一个异常类型 `goto`，其值是 100。我们没有实现这个具体的 `goto` 功能。事实上，TCL 提供的这几种异常类型已经能够满足我们编程的需要了。

还有一点需要强调的是，`catch` 中如果执行了 `return`，是不会直接从函数中返回的，而是抛出异常被 `catch` 捕获。这一点和 C++ 语言极为不同，请看下面的例子：

```
proc Test {a} {  
    catch {  
        if {$a==100} {  
            return 100  
        }  
        return 200  
    }  
    return 300  
}  
  
puts "[Test 50] , [Test 100] , [Test 300]"
```

上面代码输出是什么？乍一看应该是“200 , 100 , 300”，但是实际的执行结果却是“300,300,300”。初学者非常容易犯这样的错误，往往导致程序出现一些神秘的 `bug`。老实说，我以前也犯过，并且为此迷惑了很久才发现。

## error 命令

C++ 中抛出异常用的是关键字 `throw`，TCL 则是用命令 `error`。其命令语法如下：

```
error message ?info? ?code?
```

命令有三个参数：第一个参数是抛出的错误消息；第二个参数可选，如果指定了该参数，那么 TCL 在进行异常处理的时候，会在全局变量 `errorInfo` 中加入 `info` 所指定的信息；可选参数 `code` 我们一般很少使用，如果指定了该参数，TCL 会在将全局变量 `errorCode` 设置为该值。下面我们看一个例子：

```
proc Sqrt {a} {  
    if {$a<0} {  
        error "parameter could not <0" "Parameter Error"  
    }  
    return [expr sqrt($a)]  
}
```

```
}

proc Test {} {
    puts [Sqrt 2]
    puts [Sqrt -1]
}

catch Test msg
puts $msg
puts "-----"
puts $errorInfo
```

上面代码的执行输出是：

```
1.41421356237
parameter could not <0
-----
Parameter Error
    (procedure "Sqrt" line 1)
    invoked from within
    "Sqrt -1"
    (procedure "Test" line 3)
    invoked from within
    "Test"
```

可以看到，error 的第一个参数 message，就是 catch 命令的第二个参数指定的变量中的内容；而 error 的 info 参数，则放到了全局变量 errorInfo 中，errorInfo 给出了错误发生时的堆栈情况，方便我们定位问题。

## 执行字符串

TCL 是解释性语言，有些时候我们要在程序中解释执行一串动态生成的字符串，这个时候就要用到下面的两个命令了：eval 和 uplevel。

### eval

先拿出我们要解决的问题：模拟标准的 lappend 命令接口，写一个自己的 LApend 命令，只不过要求所有的元素是放在列表头。下面给出一个最容易想到的方法：



```
proc LAppend {varname args} {  
    upvar $varname lname      ;#传入的 varname 是变量名，这里引入到局部变量  
    中  
    lappend lname             ;#lappend，保证 lname 是一个列表  
    set lname [linsert $lname 0 $args]    ; ;#调用 linsert 插入到列表头  
}  
  
set a {100 200}  
puts [LAppend a 300 400]      ;#测试一下，输出的是 a 的内容  
puts [llength $a]            ;#看看 a 的长度
```

上面代码看起来没有什么问题，但是执行后就发现问题严重！我们期望得到 300 400 100 200。可是实际 LAppend 之后的列表是{300 400} 100 200；其第一个元素是列表{300 400}，列表 a 的长度变为 3，而不是 4。

怎么回事？问题出在 LAppend 命令的最后一句：linsert \$lname 0 \$args！TCL 在执行该语句的时候，首先对 \$lname 和 \$args 进行变量置换，但是要注意的是，TCL 中置换操作不改变单词边界！\$args 虽然被替换成为 300 400，但是 linsert 仍然只有三个参数：300 和 400 一起被当成一个元素插入到了列表头中。那么怎样才能实现我们想要的功能呢？有人会很快想出如下的方法：

```
proc LAppend2 {varname args} {  
    upvar $varname lname  
    lappend lname  
    set index 0      ;#插入元素时候的下标  
    foreach element $args {  
        set lname [linsert $lname $index $element] ;#循环插入各个 $args 中的各个  
        元素  
        incr index  
    }  
    return $lname  
}
```

经过测试验证，上面代码在功能实现倒是没有什么问题。但是这样实现实在是太复杂了，我们有更加简单的方法！见下面的代码：

```
proc LAppend3 {varname args} {  
    upvar $varname lname  
    lappend lname  
    set lname [eval [list linsert $lname 0] $args]  
}
```

这里面就用到了 `eval` 命令，该命令的语法如下：

```
eval arg ?arg.....?
```

该命令带有一个或者多个参数，这些参数能够组成一段 TCL 脚本。命令执行过程如下：

1. 首先把所有的参数连接起来，组成一个完整的字符串。连接的方式和命令 `concat` 完全一致！
2. 然后调用 TCL 脚本解释器来执行这个字符串；
3. 最后返回执行该字符串后得到的结果（包括产生的错误和异常）。

`eval` 命令是 TCL 中比较神秘的命令，初学者往往觉得不好掌握。其实掌握其执行过程就不难了。我们回过头来看看 `eval [list linsert $lname 0] $args` 的执行过程：

- 1) 首先 TCL 解释器会进行变量置换，上面的代码转换为：

```
eval [list linsert { 100 200} 0] {300 400}
```

- 2) 上面的代码中 `eval` 带有两个参数：`[list...]`和列表`{300 400}`。随后就进行命令置换，得到：

```
eval {linsert {100 200} 0} {300 400}
```

- 3) `eval` 采用 `concat` 的方式对参数进行连接，组装成的大字符串为：

```
linsert {100 200} 0 300 400
```

- 4) 调用 TCL 解释器，执行上面的这段代码，得到结果：`300 400 100 200`

初学者开始使用 `eval` 的时候，会觉得困惑！`[list...]`这个东东是用来作什么的？什么时候应该使用它？简而言之，`eval` 中的`[list...]`就是为了防止其中的列表被拆开。上面的例子中，如果我们没有这个`[list...]`，就会出错。

例如我们直接执行：`eval linsert {100 200} 0 {300 400}`。得到的结果是：

```
100 0 300 400
```

有人问了，如果我不用`[list...]`这样的括号形式，而是直接用引号可不可以，我们试试看：

```
% set a [list 100 200] ; set b [list 300 400]
300 400
% eval "linsert $a 0" $b
100 0 300 400
% eval "linsert {100 200} 0" $b
300 400 100 200
```

结果比较怪异，这给我们一个经验：要想 `eval` 的时候列表不被拆分，最好使用 `list` 命令括起来。

## uplevel

刚才介绍的 `eval` 是在当前所在的堆栈上下文空间中执行脚本，而 `uplevel` 是在当前堆栈的上层空间中执行脚本。这给我们实现自己的控制结构提供了可能。`uplevel` 的命令语法如

下：

`uplevel ?level? arg ?arg...?`

与 `eval` 相比，该命令多了一个可选的参数 `level`，用来指定在那一层栈空间中执行后面的代码。该参数必须遵从如下规则：

1. 可以是一个数字，表示在相对当前层次的上面第几层执行；
2. 可以是 `#n` 的形式，`n` 是整数，表示在第 `n` 层空间执行，最上层是 `#0`，表示全局空间；
3. 默认情况下是 `1`，表示在上面一层执行；
4. 如果后面的代码第一个字符是“`#`”或者数字，那么 `level` 不能省略；

我们看看下面的代码，弄清出栈空间层次的概念：

```
proc a {} {  
    puts "proc a .level [info level]"  
    b      ;#过程 a 调用过程 b  
}  
  
proc b {} {  
    puts "proc b .level [info level]"  
    puts "    local variable in b :[info local]"    ;#没有执行 c 之前的局部变量名  
    c      ;#调用过程 c  
    puts "    local variable in b :[info local]"    ;#调用 c 之后的局部变量名  
    puts "    x = $x"      ;#打印出变量 x 的值  
}  
  
proc c {} {  
    puts "proc c .level [info level]"  
    uplevel 1 {set x 100;d}      ;#参数 level 为 1，在上一层空间执行这段代码  
}  
  
proc d {} {  
    puts "proc d .level [info level]"  
    uplevel {set x 200}      ;#在上一层空间执行代码  
}  
  
puts "The Top level [info level]"    ;#打印当前的层次  
a      ;#调用过程 a
```

上面脚本的执行结果是：

```
The Top level 0
```

```
proc a .level 1
proc b .level 2
    local variable in b :
proc c .level 3
proc d .level 3
    local variable in b :x
    x = 200
```

这段代码中，a 调用 b，b 调用 c，在 c 中执行了 `uplevel {set x 100;d}`。这里 `set x 100` 和过程 d 调用都是在 c 的上一层栈空间中执行，所以 x 就成了过程 b 的局部变量；过程 d 也是在过程 b 的栈空间中执行，当 d 执行的时候，栈空间 c 就暂时不存在了。d 中也 `uplevel {set x 200}`，也是在 d 的上一层空间 b 中执行，所以就更改了 b 的局部变量 x，使之值为 200。上面的执行结果明确的证明了这一点。

最顶层的空间是 #0，这里只有全局变量是可见的。

## 写一个 try...catch

`uplevel` 能够把一段代码拿到任意的栈空间去执行，这在 C++ 等语言中难以想象。这种特性为我们提供了编写一些有趣代码的机会，比如前面我们就自己编写了一个 do 循环，其中就用到了 `uplevel` 命令。

TCL 中的 `catch` 捕获一切异常，这让部分 C++ 程序员赶到不爽，C++ 中的异常处理功能多么的优雅：能够根据不同的异常类型来进行不同的处理。没关系，我们试着为 TCL 写一个简单的 `try...catch` 来进行异常处理。

要注意的是，不象 C++ 或者 Python，TCL 中的异常没有类型这么一说。如果说一定有，那么就是 `TCL_ERROR`，`TCL_BREAK` 等。实际上我们处理最多的还是 `TCL_ERROR` 这么一种，这一种异常处理中，所有异常的差别往往在于消息的不同。而 `try...catch` 的优势在于能够针对不同的异常类型进行不同的处理，所以我们可以自己封装一个 `raise` 命令，来产生不同类型的异常。

下面是 `try...catch` 以及 `raise` 命令的一个简单实现：

```
#-----
# File : try.tcl
# Desc : 实现 try...catch 和 raise 过程，模拟 C++ 中的 try
# Author : LeiYuhou
#-----

#-----
# raise 过程，用来抛出特定类型的异常，用法：
```

```
# raise exceptType exceptMsg
# raise 无参数，在异常处理过程中使用，直接再次抛出当前异常
#-----
proc raise {args} {
    if {[info globals errorMsg] ne "errorMsg" } {
        set ::errorMsg ""
    }
    if {[llength $args]==2} {
        set expType [lindex $args 0]
        set expMsg [lindex $args 1]
    } elseif {[llength $args]==0} {
        set expType $::errorCode
        set expMsg $::errorMsg
    } else {
        error "Parameter error." "raise expType expMsg"
    }
    set ::errorMsg $expMsg
    error $expMsg "" $expType
}

package require control ;#引入包 control
control::control assert enabled 1

#-----
# 过程 try: 用来实现 try...catch
proc try {script args} {
    control::assert "[llength $args]%3 == 0" "Parameter error" "try script ?catch type
script?"
    set r [catch "uplevel 1 [list $script]" msg] ;#在上一层执行
    if {$r} {
        foreach {CATCH expType sc} $args {
            if {$CATCH ne "catch"} {
                error "Parameter Error." "try script ?catch exptype script?"
            }
            if {$expType eq $::errorCode || $expType eq "..."} {
                uplevel 1 $sc ;#在上一层执行
                break
            }
        }
    }
}
```

```
    }
  }
  return $r
}

#-----
#过程，抛出特定类型的过程，用来测试。
proc raiseError {errorType} {
    raise $errorType "This is a exception for testing .$errorType"
}

#-----
#下面是测试代码，用来测试 try...catch 工作是否正常
try {
    set a 100 ;    set b 200
    set c [expr $a+$b] ;    puts $c

    raiseError OSErrror

} catch IOError {
    puts "catch IOError..."
} catch AssertionError {
    puts "catch AssertionError"
} catch OSErrror {
    puts "OSErrror"
} catch NetError {
    puts "NetError"
} catch ... {
    puts "Any Error"
}
```

其中 `try` 命令的实现，就使用了 `uplevel` 命令。

我们只支持 `try...catch`，如果感兴趣，你还可以给他加上一个 `finally` 子句。就是不管抛出什么异常，`finally` 里面的代码都能够被执行。

上面的测试代码中，`try` 的语句块中调用 `raiseError` 抛出了一个 `OSErrror` 类型的异常，对应的异常处理函数中输出相应的异常类型。其执行结果是：

```
300
OSErrror
```

## 使用 assert

`assert` 是一个很有用的东西。在适当地方插入适当的 `assert` 进行断言，有助于提高程序的可测试性和可调试性，方便找出和定位问题。`assert` 几乎在所有的语言上都有实现，MFC 上就大量使用了 `ASSERT` 等宏。TCL 中也有 `assert` 的实现。

`assert` 通常使用在程序入口等地方，对一些我们预定的必须满足的条件进行断言。比如说我们上面定义的过程 `try`，其参数为 `{script args}`，其中 `script` 是可能会发生异常的代码，而 `args` 参数则是 `catch` 语句序列，显然，`args` 中元素个数必须是 3 的倍数，那么我们可以在 `try` 过程中开始的地方写上：

```
assert "[length $args]%3 == 0"
```

该语句断言：`args` 的元素个数必须是 3 的倍数！如果不是，那么就会产生错误！

并且 `assert` 行为可以被定制：在程序调试阶段，`assert` 发挥断言作用；程序发布之后，`assert` 不管表达式是否成立，都不作任何动作。要在 TCL 中使用 `assert`，必须首先从 `tcllib` 中引入扩展包 `control`。

除了上面的 `try`。下面是一个例子：

```
package require control

control::control assert enabled true      ;#启动 assert 的断言功能

proc Factorial {n} {
    control::assert "$n>=0"
    if {$n==1} {
        return 1
    }
    return [expr {$n*[expr $n-1]}]
}

puts [Factorial -10]      ;#测试一下 assert 的功能
```

下面是执行结果：

```
assertion failed: -10>=0
    while executing
    "control::assert "$n>=0""
    (procedure "Factorial" line 2)
    invoked from within
    "Factorial -10"
    invoked from within
```

```
"puts [Factorial -10]"  
(file "E:\Work\script\list.tcl" line 15)
```



## 过程和变量

前面的章节中我们对过程和变量已经有了较多的接触了，还定义了一些自己的过程，并且对列表和数组等有了深入的了解。本章节中我们将系统的深入了解 TCL 中的过程和变量相关的知识。首先从过程定义开始。

### 定义过程

TCL 中，“过程”和“函数”没有差别，这一点和 C 语言类似。在这本书当中，TCL 的“过程”和“函数”是两个完全等价，可以互相替换的概念和术语。

TCL 定义任何过程的语法如下：

```
proc name arglist body
```

`proc` 是标准的 TCL 核心命令之一，用来定义一个过程。它带有三个参数：过程名字、过程的参数列表以及过程体。

1. 过程名字可以是任意的字符串。当然了，本着易读和可维护性的原则，我们建议按照 C 语言语法惯例来命名。如果你喜欢中文，也可以给你定义的过程一个中文名字。如果定义的过程已经存在了，那么新定义的过程会替换掉原来的定义，这一点和 C 不同。在 C 中，函数是不能够被重复定义的。
2. 参数列表用来声明本过程的调用参数形式，它应该是一个 TCL 列表。列表的每一个元素就是一个过程的形式参数。当然列表可以为空，表示本过程不需要参数。
3. 过程体是一块 TCL 脚本，当本过程被调用的时候，就执行该脚本。
4. 过程返回值可以直接在 `body` 中调用 `return` 来返回，如果没有执行 `return` 就返回，那么 `body` 中最后被执行的那条命令的返回值就是过程的返回值。

TCL 中的过程支持递归，包括直接递归和间接递归。下面我们定义一个阶乘函数：

```
proc Factorial {n} {  
    if {$n==1} {  
        return 1  
    }  
  
    return [expr {$n * [Factorial [expr $n-1]]}]    ;#递归调用  
}  
puts [Factorial 4]
```

其中 `Factorial` 是过程名；`{n}` 是参数列表，只有一个元素，表示过程 `Factorial` 只有一个参数；过程内部通过递归调用来计算参数 `n` 的阶乘。

## 全局、局部

TCL 中的变量也有作用范围之说。有些变量是全局的，有些则是局部的。局部变量在过程执行完毕返回的时候自动的销毁，结束其声明周期。过程的形式参数在过程内部都是局部变量。请看下面的一个例子：

```
proc test {a b} {  
    if {$a>$b} {  
        set result $a    ;#result 变量是局部变量  
    } else {  
        set result $b  
    }  
  
    set ::globalResult $result    ;#globalResult 是新创建的全局变量  
  
    return $result  
}  
  
set c [test 100 200]  
puts $c  
puts $::globalResult
```

上面过程定义中，创建了一个局部变量 `result`，同时创建了一个全局变量 `globalResult`。它们之间的差别在于使用 `set` 命令创建变量的时候，`globalResult` 前面多了两个冒号。这两个冒号表示这个变量存在于全局名字空间中，所以是全局的变量。即使 `test` 过程退出了，该变量依然存在。

如何在过程中对全局变量进行操作？有两个方法。

第一个方法就是使用 `::` 作为名字空间限定符，来访问全局变量；

第二个方法是使用 `global` 命令来引用全局变量，该命令语法：

`global varname ?varname...?`

下面我们来看另外的一个例子：

```
set gFlag "GLOBAL variable"  
  
proc test0 {} {  
    puts "$gFlag"    ;#想直接输出全局变量，会出现异常  
}  
  
proc test1 {} {
```

```
set gFlag "Hello,LOCAL variable"
puts "$gFlag"           ;#输出局部变量
puts "$::gFlag"         ;#输出全局变量
}

proc test2 {} {
    global gFlag         ;#引用全局变量

    puts "$gFlag"         ;#输出全局变量
    set gFlag "ANOTHER value" ;#赋一个新值

    puts "$gFlag"         ;#输出该变量
    puts "$::gFlag"       ;#输出全局变量，和上面一致
}

test1
puts "-----"
test2
puts "-----"
puts $gFlag
```

我们看看上面的代码：

首先创建了一个全局变量 `gFlag`；过程 `test0` 中，通过 `$gFlag` 直接引用全局变量，这样的代码会出现错误；

在 `test1` 过程中，创建了一个同名的局部变量，然后分别引用输出，可以看到它们不会互相冲突；

在 `test2` 过程中，使用 `global` 命令声明 `gFlag` 是全局变量，后面针对 `gFlag` 的引用和操作都是针对全局变量进行的。上面代码的执行结果如下：

```
Hello,LOCAL variable
GLOBAL variable
-----
100
ANOTHER value
ANOTHER value
-----
ANOTHER value
```

## 参数默认值

和 C++ 语言类似，可以在定义 TCL 过程的时候为参数指定默认值。这样当调用过程的时候如果省略这些参数，那么就会使用默认值来进行调用。例如下面的过程：

```
#-----
#自己实现的一个 incr 函数，和标准 incr 类似
proc Incr { varname {inc 1} } {
    upvar $varname t
    set t [expr {$t+$inc}]
    return $t
}

set a 100
puts [incr a ; set a]      ;#省略了 inc，用默认值 1 来调用
puts [incr a 100 ; set a]  ;#用 100 来调用
```

过程 Incr 有两个参数，第一个是需要增加的变量名，第二个则是增幅。在参数列表中其本身是以列表的形式出现，第一个元素是参数名字，第二个参数是默认值。TCL 中所有具有默认值的参数都必须按照这样的格式定义。

定义过程的参数列表中，如果某一个参数定义了默认值，那么其后面出现的所有参数要么也具有默认值，要么就是特定的 args，例如下面的几种定义：

```
proc Say { count {what "Hello"} {who } }      ;#错误的定义；who 没有默认值
proc Say { count {what "Hello"} {who "Anyone"} args } ;#正确
```

TCL 的标准输出命令 puts 的语法如下：puts ?-nonewline? ?channelId? string。这里在参数 string 前面有两个选项参数，但是 string 是必须的，这和刚才介绍的原则不一致，可选参数出现在必须参数的前面，这样的过程应该如何定义呢？

这就需要看看下一节介绍 TCL 中的 args 参数。

## 可变数量参数列表

大家都知道 C 语言中的 printf 函数，调用它的时候参数个数是可变的。TCL 中也可以定义类似的过程，来处理个数可变的参数。包括上一节提到的 puts 命令。我们先看一个简单的例子，下面的一个过程将所有参数相加并且返回。

```
#-----
#累加所有的参数
proc Sum {args} {
```

```
set r 0
foreach i $args {    ;#这里 args 就是一个简单的 TCL 列表
    incr r $i
}
return $r
}

puts [Sum]                ;#输出 0
puts [Sum 100 200 300]    ;#输出 600
puts [Sum 1 2 -1]         ;#输出 2
```

看看上面的代码，过程 Sum 中参数 args 就是一个简单的 TCL 列表。我们在调用这个过程的时候，args 参数可以给出 0 个或者多个具体值，到了过程内部，args 就成了这些值的列表。

虽然 args 在过程内部成了列表，但是我们调用过程的时候，不能用列表的形式给其赋值，例如：

```
proc A {args} {
    return [llength $args]    ;#返回列表长度
}

puts [A]                    ;#输出 0
puts [A 100 200 300]        ;#输出 3
puts [A { 1 2 -1 } ] ;      ;#输出 1，因为参数{1 2 1}是一个列表值，而不是 3 个值
```

args 参数是独一无二的。曾经有兄弟问我，能不能自己给这样的参数取一个其他的名字，但是具有 args 参数同样的功能？答案是：不能！而且没有这样的必要。

而且同一个过程，最多只能有一个 args 参数，并且出现在参数列表的最后！

上面的原则，你可以违反，而且解释器不会出错，看看下面的代码：

```
#这里命令 A 有两个 args 参数
proc A {args a args} {
    puts "a = $a"
    puts "args = $args"
    return [llength $args]
}

puts [A 100 200 300 400 500]
```

上面代码执行结果如下：

```
a = 200
args = 100
1
```

是不是觉得不可思议？俗话说：“自作孽，不可活”。自己要挖一个坑然后往里跳，谁都挡不住！

下面我们试着写一个简单的 `putline` 命令，模拟标准的 `puts` 命令，这用到了 `args` 参数：

```
proc putline {args} {
    set newline 1      ;#是否换行
    set channel stdout ;#输出的通道，默认为 stdout
    set s ""           ;#要输出的内容

    if {[length $args]==1} {
        set s $args
    } elseif {[length $args]==2} {
        set arg1 [lindex $args 0]
        set s [lindex $args 1]
        if {$arg1!="-nonewline"} {
            set channel $arg1
        } else {
            set newline 0
        }
    } elseif {[length $args]==3} {
        set arg1 [lindex $args 0]
        if {$arg1!="-nonewline"} {
            error "Invalid option:$arg1 "
        }
        set newline 0
        set channel [lindex $args 1]
        set s [lindex $args 2]
    } else {
        error "wrong args count."
    }

    if {$newline} {
        puts $channel $s
    } else {
        puts -nonewline $channel $s
    }
}
```

```
    }  
}  
  
putline "Hello,world"  
putline stderr "Hello,wrong"  
putline -nonewline "Hello,no newline  - "  
putline -nonewline stdout "xxxx"
```

执行结果如下：

```
Hello,world  
Hello,wrong  
Hello,no newline  - xxxx
```

上面定义的过程 `putline` 模拟了过程 `puts` 的行为,可以发现其中大部分代码都是在对 `args` 参数进行了分析, 并且这样的代码比较复杂。类似 `puts` 这样具有选项参数的命令在 `TCL` 中使用的非常普遍。针对这样的选项参数, 有没有专门处理代码? 这是我们下一节要介绍的。

## 处理选项参数

选项参数是指那些用短横线开始的参数, 它可以是一个开关, 也可以是用来指定一个数值。选项使得 `TCL` 命令更加灵活方便, 因此在 `TCL` 中获得了广泛的使用, 在 `Tk` 中更甚。但是解析选项参数却不是一件让人愉快的事情, 上面我们已经看到了一个简单的 `putline` 命令, 就让我们花费了不少的精力。有没有比较统一的方式来解析选项参数呢? 回答是肯定的, `TclLib` 为我们提供了一个 `cmdline` 的扩展包, 可以方便的解析选项参数。

以我们前面介绍过的 `lsort` 命令为例子, 来说明这个扩展包的使用。

`lsort` 命令格式如下: `lsort ?options? list` 它具有很多选项参数:

`-ascii` , `-dictionary` , `-integer` , `-real`

`-command command` , `-increasing` , `-decreasing` , `-index index` , `-unique`

假定我们自己来实现 `lsort` 命令, 那么这个命令的参数解析应该如何实现? 看看下面的代码:

```
package require cmdline ;#引用程序包 cmdline  
  
#-----  
#定义自己的 listsort 过程, 实现类似 lsort 的功能  
proc listsort {args} {  
    #列表 options 中, 包含了选项参数的描述信息  
    set options {
```

```
{ascii          "sort by ascii order"}
{dictionary     "sort by dictionary order"}
{integer        "sort by integer number"}
{real           "sort by float number"}
{command.arg    ""  "sort by your command"}
{increasing     "sort increasing"}
{decreasing     "sort decreasing"}
{index.arg      -1  "sort by the index element "}
{unique         "get rid of duplicate elements"}
}

set comparetype 0      ;#排序的比较类型

#解析命令参数
set optlist [::cmdline::getoptions args $options ]
array set param $optlist

set l $args            ;#经过处理后，args 中最后剩下的应该被排序的列表
if {[length $l]==0} {
    return ""          ;#如果是空列表，就直接返回
}

puts "[string repeat - 15] listsort $l [string repeat - 15]"
foreach {opt val} $optlist {
    puts "$opt = $val"
}

#判断是否指定了 -command 选项?
if { $param(command)!=""} {
    if { $param(ascii)||$param(dictionary)||$param(integer)||$param(real)} {
        error "-command could not mixed with ascii,dictionary,integer or read."
    }
    # -command 方式排序
    return
}

#判断排序时候的比较方式
foreach type {ascii dictionary integer real} {
```



```
        if { $param($type) } {
            if { $comparetype==0 } {
                set comparetype $type
            } else {
                error "ascii ,dictionary,integer and real could not appear together."
            }
        }
    }

    if { $comparetype=="0" } {
        set comparetype ascii ;#如果都没有指定，那么就默认为 ascii 模式
    }

    #判断排序顺序，inc 和 dec 不能同时指定
    if { $param(increasing) && !$param(decreasing) } {
        set order increasing
    } elseif { !$param(increasing) && $param(decreasing) } {
        set order decreasing
    } elseif { !$param(increasing) && !$param(decreasing) } {
        set order increasing
    } else {
        error "-increasing and -decreasing could not exist together"
    }

    #其他的选项判断
    #.....
}

#测试一下上面的过程
listsort {a b c d e}
listsort -increasing {100 200 300}
listsort -increasing -ascii -real {100 200 300}
```

上面的代码执行结果如下：

```
----- listsort {a b c d e} -----
index = -1
real = 0
decreasing = 0
```

```
ascii = 0
integer = 0
dictionary = 0
unique = 0
increasing = 0
command =
----- listsort { 100 200 300} -----
index = -1
real = 0
decreasing = 0
ascii = 0
integer = 0
dictionary = 0
unique = 0
increasing = 1
command =
----- listsort { 100 200 300} -----
index = -1
real = 1
decreasing = 0
ascii = 1
integer = 0
dictionary = 0
unique = 0
increasing = 1
command =
ascii ,dictionary,integer and real could not appear together.
    while executing
"error "ascii ,dictionary,integer and real could not appear together.""
    (procedure "listsort" line 46)
    invoked from within
"listsort -increasing -ascii -real { 100 200 300}"
    (file "E:\Work\ script\list.tcl" line 79)
```

上面的代码看起来比较庞大，但是实际上逻辑非常简单，进行选项参数解析只用两条语句就完成了，这就是：

```
set optlist [::cmdline::getoptions args $options ]
array set param $optlist
```

其中第一条用来进行解析，第二条把结果放到一个数组中。命令 `getoptions` 是扩展包 `cmdline` 提供的一条选项解析命令，其语法格式如下：

`getoptions arglistVar optionslist ?usage?`

1. 第一个参数 `srglistVar` 是需要被解析的包含选项参数的参数列表的变量名；注意这里传入的是列表变量的名字，而不是列表本身。因为 `getoptions` 过程需要修改这个列表。
2. 第二个参数 `options` 是选项描述列表；它用来告诉 `getoptions` 如何解析选项参数。
3. 第三个参数可选，当出现解析错误的时候，它会被用上生成异常信息。

我们在代码中调用该命令的时候，参数 `arglistVar` 的值就是 `args`，也就是传递给过程 `listsort` 的可变参数列表。第三个参数为空。

参数 `options` 用来描述选项的语法，是一个二维列表。对照上面的例子：

```
set options {
    {ascii          "sort by ascii order"}
    {dictionary     "sort by dictionary order"}
    {integer        "sort by integer number"}
    {real           "sort by float number"}
    {command.arg    ""  "sort by your command"}
    {increasing     "sort increasing"}
    {decreasing     "sort decreasing"}
    {index.arg      -1  "sort by the index element "}
    {unique         "get rid of duplicate elements"}
}
```

其中每个元素描述一个选项：它本身就是一个列表：第一个元素是选项名字，最后一个选项的描述字符串；如果该选项是一个开关选项，那么就只有两个元素，例如上面的 `read` 和 `ascii` 等；如果该选项是通过紧随其后的参数来指定具体值的，例如上面的 `index`，那么必须：

1. 第一个元素，也就是选项名字后面必须加上 `.arg`，作为名字的结尾；表示这个选项使用紧随其后的参数作为值；
2. 第二个元素，是该选项的默认值，也就是说如果没有指定该选项，那么就使用该默认值；
3. 第三个元素才是选项参数描述信息字符串；

`getoptions` 命令如果执行成功，就返回一个列表。我们可以直接使用 `array set` 命令直接将它们转换到一个数组当中去。数组下标是选项名字，值则是选项的值（开关选项则是 1 和 0，否则就是具体值或者默认值）。如果解析失败，那么就抛出异常信息。

上面的例子中，`-command` 和 `-index` 是值选项，其它的都是开关选项。我们试着如下调用：

```
listsort -increasing -ascii -index 100 {100 200 300}
```

打印信息如下：

```
----- listsort {100 200 300} -----
```

```
index = 100
```

```
real = 0
```

```
decreasing = 0
```

```
ascii = 1
```

```
integer = 0
```

```
dictionary = 0
```

```
unique = 0
```

```
increasing = 1
```

```
command =
```

可以看到，开关选项-ascii 和-increasing 被指定了，对应的值都是 1，其他的开关选项都是 0；-index 选项被指定了，所以值是 100；-command 没有被指定，所以是默认值为空。

## 传引用还是传值？

其实“传值”和“传引用”是借鉴了 C++或者 Pascal 的概念，在 TCL 中本质上不存在这样的说法。TCL 中传递的一切参数，都是传值，都是字符串。那如果我们要通过参数从过程中带出一些信息，应该怎么办呢？命令 upvar 可以帮助我们解决这个问题。

在详细讲解 upvar 之前，我们先看看 incr 这个命令的使用：

```
set a 100          ;#创建一个变量 a，初始为 100
incr a 200         ;#给变量 a 加上 100，incr 返回后，a 为 200
```

上面的 incr 的参数 a，就有一点传引用的味道：因为 incr 返回之后，这个变量 a 的值也发生了变化，传值的参数是不具备这样的特性的。我们还注意到，这里调用 incr 的时候，是 incr a，而不是 incr \$a。也就是说，我们传递给 incr 的是变量名，而不是引用的变量值。原因很简单，我们要在 incr 中修改参数代表的变量，如果是\$a，就无法修改了。

我们自己来写一个简单的 increase 过程，模拟 incr 的功能：

```
proc increase { varname {i 1} } {
    upvar $varname t      ;#将上层变量名关联到局部变量 t
    set t [expr {$t+$i}]
}

set age 27
puts [increase age 2]
```

increase 的实现代码只有两行，当我们调用 increase age 2 的时候，参数 varname 的值就是字符串 age（肯定不是 27，因为不是\$age），里面的 upvar 命令经过替换后，变成 upvar age

t, 这个命令就是把上一层的 age 关联到局部变量 t, 对变量 t 做出的任何改动, 都会在 age 上完全体现出来。当过程返回后, 局部变量 t 销毁, 但是 age 依然存在, 并且所有改动都保留下来。这就实现了所谓的“传引用”的参数。其奥妙全部在于 upvar 命令, 该命令的语法格式如下:

```
upvar ?level? othervar myvar ?othervar myvar……?
```

其中选项参数 level 的含义以及取值, 与命令 uplevel 的参数 level 完全类似。如果省略就表示上一层栈空间的变量。othervar 是上层空间的变量名, myvar 是关联的局部变量。upvar 命令可以一次关联多个变量。上层变量可以是一个普通变量名, 可以是数组名, 也可以是数组的某一个元素。upvar 关联的上层变量, 可以先不用创建。在过程内部第一次创建关联的局部变量的时候, 会自动的创建上层变量。

总结一下: 要想在 TCL 中实现传引用参数, 需要传递的是变量名而不是变量值, 然后在过程内部使用 upvar 命令将这个变量名字和局部变量关联起来。

## 再说 return

对于 return 命令我们不再陌生, 前面在介绍 catch 的时候已经做了详细的介绍。TCL 过程的返回值有两种途径:

1. 如果不是 return 返回, 那么过程中最后一条被执行的命令的结果, 就是本过程的返回值;
2. 如果是 return 返回, 那么返回情况由 return 的参数来决定;

例如下面的过程:

```
#the proc Odd judge whether n is odd
proc Odd {n} {
    if {$n%2==1} {
        set i 1
    } else {
        set i 0
    }
}
```

上面过程判断 Odd 是否是奇数, 可以看到过程内部没有任何 return 命令, 只有一个 if 命令, 所以其返回值就应该是 if 命令的执行结果。对于 if 命令的返回值, TCL 帮助中是这样给出说明的: if 命令的返回值就是其所执行的分支脚本的执行结果, 如果没有一个分支被执行到, 那么就返回空字符串! 所以上面的过程的返回值根据参数 n 的具体取值分别是“set i 1”和“set i 0”两个命令的结果, 也分别是 1 或者 0。

作为一个编程规范方面的建议: 我们最好在自己定义的过程中都采用明确的 return 命令从过程中返回结果。

return 命令的格式如下：

**return** *?-code code? ?-errorinfo info? ?-errorcode code? ?string?*

其中-code 选项我们很少用到。要看看-code 怎样使用，请参考前面异常处理中的“神奇的 return”这一章。我们用的最多的形式就是 `return string`。直接从过程中返回。

## 过程更名、删除

TCL 中的过程可以被改名，即使是标准的过程也可以，但是不鼓励这样做。`rename` 命令可以实现过程改名的功能，例如：

```
#定义过程 Say
proc Say { args } {
    puts "Say : $args"
}

Say Hello world !                ;#调用这个过程

rename Say Shout                  ;#改名为 Shout
Shout hehe , Good morning !      ;#调用该过程
puts [catch {Say do you exist?} msg;set msg]    ;#调用原来的过程，打印错误信息
rename Shout ""                   ;#删除这个过程

Shout and you?                    ;#直接抛出异常
```

这是上面代码的输出：

```
Say : Hello world !
Say : hehe , Good morning !
invalid command name "Say"
invalid command name "Shout"
    while executing
    "Shout and you?"
(file "E:\Work\TCL、Python 和自动化测试\script\list.tcl" line 13)
```

过程改名之后，原来的过程名字就消失，如果调用老名字就会出错。如果 `rename` 改名的时候新名字为空字符串，那么就等同于删除这个命令。同一个名字的过程可以被定义多次，后面的定义会覆盖原来的定义，例如：

```
proc Say { args } {
    puts "Say : $args"
```

```
}

Say Hello world !

proc Say { who args } {
    puts "$who SAY: $args"
}

Say LeiYuhou hello
```

代码执行结果如下：

```
Say : Hello world !
LeiYuhou SAY: hello
```

## 变量跟踪

TCL 语言中一个比较神奇的功能是：我们可以在程序中来跟踪某个变量是如何被操作的。例如：假如我们希望某个变量被读取或者被修改的时候，能够自动执行一些程序，或者实现一个只读变量，那么这个功能就派上用场。命令 `trace` 可以实现该功能，其语法如下：

`trace add variable name opts command`

其中：

1. `name` 是需要跟踪的变量名，它可以是单个变量、数组名或者某一个数组元素；
2. `opts` 是需要跟踪的操作类型，可以是下列变量操作类型的列表：`array`、`read`、`write` 和 `unset`，分别表示使用 `array` 命令的操作、读变量、设置变量和删除变量；
3. 最后的参数 `command` 是一个脚本字符串，表示当被跟踪操作发生的时候需要触发执行的脚本。

当 `command` 语句被调用的时候，解释器会在其后添加如下的参数列表：

`command name1 name2 opt`

其含义如下：

1. `name1` 和 `name2` 表示变量名字。当跟踪的变量是数组中某个单独元素的时候，`name1` 表示数组名，`name2` 表示元素的下标；否则 `name2` 为空，`name1` 表示变量名；
2. `opt` 则是触发这次跟踪的操作类型，可以是 `array`、`read` 等；

一般情况下我们把 `command` 设置为某个过程的名字。它类似 C++ 中的回调函数，好比我们定义的一个触发器。当某处代码读取或者修改被跟踪变量的时候，该触发器就被触发。要注意的是：`command` 的执行上下文就在执行触发代码的上下文中。如果触发代码是在一个过程中执行的，那么 `command` 脚本能够直接操作这个过程局部变量！

先看看例子吧：

```
#=====
#File name   : traceTest.tcl
#Author      : LeiYuhou

array set m {1 100 2 200 3 300} ;#被跟踪的全局变量：数组 m

#增加跟踪操作，这里针对整个数组的所有操作都要跟踪
trace add variable m [list read write unset array] OnVarAccess

#增加对 m(3)这一个元素的写操作的跟踪
trace add variable m(3) write OnM3Write

#定义回调函数 OnM3Write
proc OnM3Write {n1 n2 op} {
    upvar $n1 v
    puts "In Trigger body.m(3) was writed,new value=$v($n2)"
}

#定义回调函数 OnVarAccess
proc OnVarAccess {n1 n2 op} {
    upvar $n1 v
    switch $op {
        read {
            incr v($n2) ;#读出的结果都会增加 1
        }
        write {
            if {$n2==1} {
                #下标为 1 的元素，是不能够更改的，否则出错
                return -code error \
                    -errorinfo "You could not change variable $n1"
            } else {
                incr v($n2) 100 ;#写的时候，其他元素都会增加 100
            }
        }
    }
    puts " !Triggers $n1 , $n2 , $op"
}
```



```
puts "-->array set operation : [array get m]\n"
puts "-->write element M(2) : [set m(2) 200]\n"
puts "-->after trigger,M(3)'s new value = [set m(3) 400 ]\n"

puts "-->write element M(1) : [set m(1) 101]\n"
```

上面代码的执行结果如下:

```
!Triggers m , , array
!Triggers m , 1 , read
!Triggers m , 2 , read
!Triggers m , 3 , read
-->array set operation : 1 101 2 201 3 301

!Triggers m , 2 , write
-->write element M(2) : 300

!Triggers m , 3 , write
In Trigger body.m(3) was writed,new value=500
-->after trigger,M(3)'s new value = 500

can't set "m(1)":
    while executing
    "set m(1) 101"
    invoked from within
    "puts "-->write element M(1) : [set m(1) 101]\n""
    (file "E:\Work\traceTest.tcl" line 42)
```

可见: 调用 `array get` 的时候, 触发了一个 `array` 操作和三个 `read` 操作。我们在响应 `read` 和 `write` 的时候, 做了一些有趣的事情:

可以在这个时候, 修改监视变量的值! 并且: 修改后的结果就作为最终读写操作的结果。例如上面的例子中:

1. 读的时候, 我们给每一个变量加一。这可以从结果中看到!
2. 写的时候, 会在设置下去的的基础上加上 100。例如上面的代码中设置 `m(2)` 为 200, 但是打印的结果是 300!

数组元素 `m(1)` 是只读变量, 这是如何实现的呢? 只要在回调函数对应的 `write` 操作中, 返回一个错误返回码即可。上面例子通过 `return -error` 来返回错误, 表示不可以修改! 如果某处代码想要修改该变量, 就会产生异常!

还有几点要注意:

1. 可以为同一个变量的同一个操作定义  $N$  个跟踪回调；当触发跟踪操作的时候，所有对应的回调按照它们被 `trace add` 的顺序依次执行。但是一旦某个脚本返回了错误，那么后面的脚本就不会执行；
2. 在 `trace add` 的回调代码 `command` 中，我们可以直接读写被跟踪变量！这样的读写操作不会触发跟踪。想想也是，否则就陷入一个无穷递归当中了。
3. 如果针对某个数组增加了一个跟踪操作 `a`，同时针对其中某个元素也定义了一个跟踪操作 `b`，那么当操作这个元素的时候，`a` 会在 `b` 之前被触发！这也可以从上面例子的 `m` 以及 `m(3)` 看出来。

命令 `trace info variable varname` 可以用来查看在某一个变量上定义的所有跟踪信息，返回的是一个列表。例如针对刚才的例子，执行如下命令：

```
puts [trace info variable m]
puts [trace info variable m(3)]
```

执行结果如下：

```
{ {array read write unset} OnVarAccess }
{ write OnM3Write }
```

`trace remove` 用来删除对变量的跟踪，其格式如下：

`trace remove variable name opList command`

`name` 是被跟踪的变量，`opList` 是跟踪的操作列表，`command` 则是回调命令。还是拿刚才的例子，下面的代码删除变量 `m` 上所有的跟踪：

```
foreach t [trace info variable m] {
#   eval trace remove variable m $t    ;#下面的语句也能完成同样的功能
    trace remove variable m [lindex $t 0] [lindex $t 1]
}
```

## 名字空间

TCL 为什么要引入名字空间的概念呢？这是因为随着程序规模的增加，参与代码编写的人员数量的增加，就有可能在全局变量名字，过程名字的命名上产生重复和冲突。对于 C 这样的语言还好，编译器会报告重复定义的编译错误或者链接错误；对于 TCL 而言，命令是可以重复定义的，这样一旦程序运行产生错误，查找起来将会非常的困难。

可能有人会说了，脚本语言开发程序，基本上都是我一个人搞定，怎么可能会有很多人参与呢？不要忘了，TCL 是与网络紧密结合的、开放源代码的语言，在网络上存在无数的人来为 TCL 开发各种各样的程序库和扩展包，他们互相之间既有可能根本不认识，所以命令同名是在太有可能发生了，比如象 Create、Initial 和 Close 等等这些都要被用滥了的名字。

为了解决这些问题，名字空间应运而生。名字空间 namespace，顾名思义，可以理解成有名字的空间，它包含了那些属于该空间的变量名和过程名。不同名字空间里面，即使包含同名的命令，TCL 解释器也能够区分它们，看待成不同的命令。

TCL 中所有和名字空间相关的命令只有一个，就是 namespace，命令的第一个参数决定了完成什么功能，下面我们逐一介绍。

## 创建名字空间

TCL 中没有 namespace create xxx 这样的命令来创建名字空间，使用的是另外一个命令：namespace eval，其语法如下：

```
namespace eval namespace arg ?arg?
```

其中第二个参数 namespace 是名字空间的名字；后面的参数是需要在这个名字空间中执行的代码，如果 arg 参数有多个，那么命令会先把它们用空格连接起来组装成一个字符串。该命令在名字空间 namespace 中执行后面的代码，如果参数 namespace 指定的名字空间还不存在，那么就会先创建这个名字空间。

下面是一个简单的名字空间，实现一个计数器以及相关的操作：

```
#-----
#创建名字空间 Counter
namespace eval Counter {
    namespace export *
    variable m_count 0 ;#定义名字空间内部变量

    #定义过程 Reset，将计数器复位
    proc Reset {} {
        variable m_count
```

```
        set m_count 0
    }
    ;      #!!! 这里应该有分号，否则会出错!!!
} {
    #这里定义过程 Increase，增加计数器，默认参数为 1
    proc Increase { {m 1} } {
        variable m_count
        incr m_count $m
    }
}

#名字空间已经 Counter 存在，继续在 Counter 中执行如下代码
namespace eval Counter {
    #定义过程 GetCounter，返回得到计数器的实际值
    proc GetCounter {} {
        variable m_count
        return $m_count
    }
    #定义子名字空间 Operator，里面定义了一个函数 Print
    namespace eval Operator {
        proc Print {} {
            variable ::Counter::m_count
            puts "The counter is $m_count"    ;#打印计数器当前值
        }
    }
}

#下面是测试代码，增加计数器技术，然后打印计数器的值
Counter::Reset
for {set i 0} {$i<10} {incr i} {
    Counter::Increase
}

puts [Counter::GetCounter]      ;#直接打印计数器的值
::Counter::Operator::Print      ;#调用内部定义过程来打印
```

第一个 namespace eval 命令带有三个参数：名字为 Counter，剩下的两个参数是两块代码，用来在 Counter 之中执行，分别定义了两个过程 Reset 和 Increase。因为该 namespace eval

执行的时候 `Counter` 还不存在，所以就创建了该名字空间。紧接着后面的一个 `namespace eval` 命令，用来在名字空间 `Counter` 中创建过程 `GetCounter`。再往下就是测试代码，用来检查一下这个计数器名字空间是否正常工作。最后的打印结果是“10”

细致的读者可能会发现，过程 `Reset` 定义结束后，后面跟了一个分号，有什么用处？分号的用处在于，如果后面有多个语句块，那么在将它们连接成一个字符串之后，不会因为没命令分隔而出现错误。一般情况下我们不赞成象上面的代码那样后面跟多个 `arg` 参数，而是把所有的代码全部放到一对花括号之中。

名字空间是可以嵌套的，一个名字空间可以包含多个孩子名字空间，例如上面的代码就定义了嵌套的名字空间 `Operator`，并且在其中定义了过程 `Print`。

上面的代码中还出现了一个我们没有见过的命令 `variable`，下面我们详细讨论：

## variable 命令

`variable` 一般用在 `namespace eval` 的语句块中，定义该名字空间内部的全局变量。这里要区分全局和局部的概念：局部（`local`）表示在过程内部生成的变量，过程外部的变量都是全局（`global`）的变量。但是在名字空间出现后，也存在一个全局的概念，这就是全局名字空间，比如 `TCL` 内建的命令 `set` 等命令都属于全局名字空间。这里不要搞混淆了。

每一个名字空间，都可以有自己的全局变量，`variable` 就可以用来定义以及引用这些全局变量。该命令的语法如下：

```
variable ?name value...? name ?value?
```

命令 `variable` 执行的时候，如果 `name` 指定的变量还不存在，那么就创建这个变量。在这种情况下，如果指定了 `value`，那么就给变量赋值为 `value`，否则变量处于未定义状态；如果变量已经存在，那么当 `value` 参数有的时候，就直接给该变量赋值，否则变量值保持不变。

如果 `variable` 是在某一个 `TCL` 过程中被调用，那么就会创建一个同名的局部变量，和对应的全局变量关联起来。在这种情况下，`variable` 命令和 `global` 非常的类似，它们的差别在于：`global` 关联的是全局名字空间（`::`）中的全局变量，而 `variable` 关联的是命令所在名字空间的全局变量。

例如上面的 `Counter` 中，就有如下两种语句：

1. `variable m_counter 0`
2. `variable m_counter`

第一个我们是在 `Counter` 中任何过程外面调用的，并且给出了初始值，那么该语句就在 `Counter` 中创建了一个全局变量 `m_counter` 并且初始值未 0；第二个则是在 `Counter` 内部定义的过程中调用的，没有给出 `value` 参数，那么它的功能就是在过程内部创建一个同名的局部变量并且和全局变量关联起来，然后我们就可以象使用局部变量一样使用全局变量。

`variable` 命令的 `name` 参数可以带有名字空间来修饰，例如上面代码中的 `Print` 过程，这个过程是在 `::Counter::Operator` 中定义的，其内部就使用了 `variable ::Counter::m_counter` 语句，用来声明一个局部变量 `m_counter`，和其他名字空间中的变量关联起来。我们一般不推荐这

种用法！

读者可能会问，如果我使用的变量是数组，应该如何给其赋初始值？可不可以象下面这样写？

```
variable m_days {1 Mon 2 Tru 3 Wed}
```

回答是可以这样下，但是结果不是我们要的，`m_days` 是一个列表！正确的方法是：使用 `variable` 来声明变量，省略掉后面的 `value` 参数，然后调用 `array set` 等方法来为数组变量赋值？例如：

```
variable m_days  
array set m_days {1 Mon 2 Tru 3 Wed }
```

还有读者会问：过程中调用 `variable` 关联变量的时候，这个变量必须存在吗？必须是用 `variable` 声明并且创建的吗？两个问题的答案都是否定的。例如下面的代码：

```
proc t {} {  
    variable m 100      ;#定义全局变量，初值为 100  
    puts "In procedure t : $m"  
}  
  
t      ;#调用该过程  
puts "Global variable m = $m"      ;#直接引用该全局变量
```

上面代码执行如下：

```
In procedure t : 100  
Global variable m = 100
```

`variable` 命令使用很灵活，常见用法，总结一下：

1. 在名字空间中创建全局变量，并且为其赋值！
2. 在名字空间的过程中调用 `variable`，可以象局部变量一样引用该变量！

总之，`variable` 使用非常灵活，我们要求把握的是该命令的本质！

## 名字和名字解析

前面我们看到名字空间是可以嵌套的，就好比 Unix 操作系统中的文件系统目录结构一样。Unix 中表示目录的方式是用 “/” 作为分隔，TCL 中表示嵌套名字空间的方式比较类似，只不过分隔符号换成了 “::”，两个连续的冒号。

每一个名字空间都必须有一个名字来标识，除了全局名字空间（`global namespace`），它只需要用两个冒号（`::`）就行了，就好比文件系统根目录用 “/” 表示。例如，`set` 命令是标准的 TCL 命令，存在于全局名字空间中，我们可以这样使用：

```
::set myage 23
```

不过这样总有画蛇添足的嫌疑。

例如前面代码中，为了在全局名字空间中调用 `Print` 过程，我们写出如下的代码：

```
::Counter::Operator::Print
```

如果我们直接写 `Print` 来调用，TCL 解释器是无法找到这个过程的。同样，如果需要在全局名字空间中引用 `Counter` 中的变量 `m_counter`，也需要这样写：

```
puts $::Counter::m_counter
```

名字空间中变量名和过程名的解析，有一定的规则。如果它们有名字空间来修饰，那么就按照指定的名字空间来定位寻找，否则，先在当前名字空间中寻找，如果没有找到，就到全局名字空间中寻找，如果还没有找到，那么就报告错误。下面是一个啰行的例子：

```
set m "Global ::m"      ;#全局的变量 m

proc print {args} {
    puts "I am ::print . Value is : $args"      ;#全局的过程 print
}

namespace eval a {
    set m " m in ::a"      ;#创建::a::m 变量，也是全局，但是属于 a

    proc print {args} {
        puts "I am a::print . Value is : $args"      ;#这是过程 a::print
    }

    namespace eval b {      ;#定义名字空间 b
        print $m      ;#直接打印变量 m

        proc print {args} {
            puts "I am a::b::print . Value is : $args"
        }

        variable m "m in ::a::b"      ;#定义变量::a::b::m
        print $m      ;#再次打印变量 m
    }
}
```

上面的代码执行结果如下：

```
I am ::print . Value is : {m in ::a}  
I am a::b::print . Value is : {m in ::a::b}
```

执行结果正号印证了上面介绍的关于调用过程以及变量名的查找规则，具体请读者自行分析。

命令 `namespace which` 可以告诉我们 TCL 解析到的具体结果，例如还是结合刚才的程序代码，在代码后面分别执行下面的语句：

```
puts [namespace eval ::a::b {namespace which -variable m}]  
puts [namespace eval ::a::b {namespace which -command print}]
```

这两条命令分别告诉调用者，在名字空间 `::a::b` 中，解析到的变量 `m` 和命令 `print` 的完整名字，执行结果如下：

```
::a::b::m  
::a::b::print
```

也请读者自行分析这两个结果。

## 引入命令

名字空间一般被用来创建程序库，有一些程序库中的命令使用得非常频繁，那么每次调用这些库中命令的时候都要写上一个带名字空间的命令，显得就比较繁琐了。例如我们前面提到过的 `assert` 命令，就放在名字空间 `control` 中。每次调用的时候都得这样：

```
control::assert $a>=0
```

如果直接调用就方便了，象下面这样：

```
assert $a>=0
```

这就好像调用本名字空间中的命令一样。那么，怎样才能把其他名字空间中的命令直接引入到当前名字空间中呢？可以使用 `namespace import` 来完成。该命令语法如下：

**namespace import** *?-force? ?pattern pattern ...?*

其中参数 `pattern` 是需要引入的带有名字空间修饰的名字，类似：`itcl::class`，或者 `itcl::*` 等；它包含需要被引入命令所在的名字空间的名字，和命令名模式，命令名可以是一个完整的命令，也可以通过 `glob` 模式的通配符指定一系列命令。例如 `itcl::*` 表示 `itcl` 名字空间中的所有输出命令；`itcl::c*` 表示所有 `itcl` 中以字母 `c` 开始的命令（什么是 `glob` 模式？本书前面章节有详细的章节论述）。`pattern` 中的名字空间不能带有通配符，例如：

```
namespace import contr??:*
```

就会引发错误：

```
unknown namespace in import pattern "contr??:*"
```



```
while executing
"namespace import contr??:*"
```

当命令执行时，所有符合 `pattern` 模式的命令都被引入到当前名字空间中来。实际上就是在当前名字空间中创建了一个新的同名命令，和原始名字空间中的命令建立连接。当这个新命令被调用的时候，实际上就是调用了原始的命令。

当引入命令的时候，如果当前名字空间中已经存在同名的命令，那么就会发生冲突，命令会抛出一个异常。如果要避免抛出这样的异常，我们需要指定 `-force` 选项参数！当指定该参数之后，如果当前名字空间已经存在同名的命令，那么 `namespace import` 命令会悄悄把原来的命令给替换掉！例如：

```
package require control
proc assert {args} {      ;#当前名字空间中定义一个自己的 assert 命令
    eval control::assert $args
}

namespace import control::assert ;#引入 control::assert 命令
assert {100>99}
```

上面的代码中，我们先定义了一个自己的 `assert` 命令；然后再次引入该命令，结果出现了如下的错误：

```
can't import command "assert": already exists
while executing
"namespace import control::assert"
```

如果换成 `namespace import -force control::assert`，则可以避免这样的异常。但是加上了 `-force` 选项之后，就有可能在我们不知情的情况下发生命令替换。这样就可以能发生错误，而且调试和定位起来非常困难。所以我们不推荐这个选项！

## 输出命令

当我们调用 `namespace import` 引入某个名字空间中的命令的时候，只有那些标明了可以输出的命令，才能够被引入。采用 `namespace export` 命令来输出名字空间中的命令，该命令格式如下：

**`namespace export ?-clear? ?pattern pattern ...?`**

该命令和 `namespace imprt` 很类似。`pattern` 也是一个 `glob` 模式的字符串，表示需要输出的命令模板；不同之处在于，`pattern` 中不能够包含名字空间，例如下面语句不正确：

```
namespace export Counter::*
```

拿我们前面的 Counter 名字空间为例,我们可以在 Counter 内部写上这么一句:namespace export \*; 它表示本名字空间内部所有命令都可以输出。然后在 Counter 外部调用 namespace import Counter::\*; 将 Counter 中的所有命令都引入到当前空间中,然后我们就可以直接调用 Reset, Increase, GetCounter 等命令了。再看下面的一个例子:

```
namespace eval A {
    namespace export m      ;#这里我们只输出命令 m
    proc m {} {puts "it is m"}
    proc n {} {puts "it is n"}
}

namespace import A::*      ;#引入 A 中所有输出的命令
m                          ;#可以调用成功
A::n                      ;#也可以调用成功
n                          ;#调用失败,找不到命令 n
```

执行结果如下:

```
it is m
it is n
invalid command name "n"
    while executing
    "n"
```

实际上,在每个空间中都存在一个输出模板的列表,我们每次调用 namespace export 的时候:

1. 如果没有-clear 选项参数,那么就直接将 pattern 添加到这个列表的末尾;
2. 如果有-clear 选项,则先清空该列表。然后再添加 pattern,如果有 pattern 的话;
3. 如果既没有-clear 选项,也没有 pattern 参数,那么就返回当前名字空间的输出模式列表;

比如接着刚才的代码,最后执行下面的语句:

```
namespace eval A {puts [namespace export]}
```

该语句的执行结果就是打印出: m

如果一个名字空间中某个命令不符合任何一个输出模式,那么我们就不能够 import 这个命令;但是我们仍然可以在其他名字空间中,通过指定完整的命令名字来调用它,就好比刚才例子中的语句: A::n。

## 其他名字空间命令

掌握前面几个小节介绍的名字空间命令的使用方法之后，基本上就能够满足我们平时的编码了。但是 TCL 还包含了一个和名字空间有关的命令，可以帮助我们解决一些其他方面的问题。对这些命令，我们只作简单的描述，具体请大家在实际编码中慢慢琢磨。

### **namespace children** *?namespace? ?pattern?*

本命令返回属于指定名字空间的所有子名字空间，如果指定 `pattern`，则只返回那些匹配 `pattern` 的名字空间，如果没有指定 `namespace`，那么就返回当前名字空间的孩子。例如：

```
puts [namespace children ::]      => ::ftp ::log ::pkg ::vfs
puts [namespace children Counter] => Operator
```

### **namespace current**

该命令返回当前所在的名字空间的名字。

### **namespace delete** *?namespace namespace ...?*

这个命令用来删除名字空间以及其中所有的命令、变量以及所有的子名字空间；

### **namespace exists** *namespace*

如果名字空间存在，那么就返回 1，否则返回 0

### **namespace forget** *?pattern pattern ...?*

如果 `pattern` 中包含名字空间修饰，比如 `A::*`，那么该过程就是 `import` 的逆过程，将能够和 `pattern` 匹配上命令从当前名字空间中删除掉。如果 `pattern` 就是一个简单的模式字符串，那么就在当前名字空间中搜索匹配得上的命令，如果这些命令是从其它名字空间中引入进来的，那么就将其删除。

### **namespace origin** *command*

本命令返回 `command` 的原始命令的全名。看看下面的例子：

```
namespace eval A {
    namespace export m n
    proc m {} {puts "it is m"}
    proc n {} {puts "[namespace current]"}
}
```

```
namespace eval B {
    namespace import ::A::*      ;#引入 A 中的所有命令
    namespace export *           ;#然后再输出它们
}

namespace import B::*           ;#从 B 中引入到全局 namespace 中
puts [namespace origin m]
puts [namespace origin set]
```

上面的代码，首先 A 中定义了命令 m、n；然后在 B 中，引入 A 中的所有命令，让它们又可以被输出；最后在全局名字空间中引入 B 中的所有命令（这当然包含 m 和 n）。我们要看看 m 命令就是在什么地方被定义的，调用 `namespace origin m` 即可。

执行结果如下：

```
::A::m
::set
```

对于那些不是被引入的命令，直接返回它们的带有名字空间的名字，例如 `::set`。

#### **namespace parent ?namespace?**

返回名字空间的父亲。如果没有指定 `namespace` 参数，那么就返回当前名字空间的父亲。如果紧接刚才上面的代码，执行下面的语句：

```
namespace eval B {puts [namespace parent]}      ;#返回 B 的父亲
```

结果是 “::”，表示是全局名字空间。

#### **namespace qualifiers string**

##### **namespace tail string**

返回一个带有修饰符的名字空间字符串的修饰符部分和结尾部分。例如：

`namespace qualifiers “::A::B::C”`      =>结果是： `::A::B`

`namespace tail “::A::B::C”`      =>结果是： `C`

#### **namespace which ?-command? ?-variable? name**

这个命令我们前面已经接触到了，用来返回一个命令或者变量经过 TCL 解析后，其所引用的命令或者变量名。这个命令和 `namespace origin` 有很大不同。还是紧接着刚才的代码，执行如下语句：

```
namespace eval B {puts [namespace which -command m]}      =>::B::m
puts [namespace which -command m]                        =>::m
```

因为 `namespace which` 所返回的是命令调用的是哪一个命令，而不是原始命令，所以上面的结果都是不难理解的。

**namespace code script**

这是一个很有用的命令，它会返回一个对 `script` 经过精心包装后的字符串，这个字符串可以在任意的名字空间中执行，当执行的时候，就好比在原来执行 `namespace code` 命令的名字空间中执行 `script`。说得很绕口，我们看看下面的例子：

```
namespace eval A {
    proc m {} {return [namespace code n]}
    proc n {args} {puts "in [namespace current]. args = $args"}
}

namespace eval B {
    set t [A::m]
    $t 1 2 3 4
}
```

执行结果如下：

```
in ::A. args = 1 2 3 4
```

我们看到在 `B` 中，首先执行 `A::m`，得到对命令 `A::n` 的包装字符串，然后在 `B` 中执行返回的这个字符串，并且传入了一堆参数，从结果来看，这个字符串虽然在 `B` 中直接执行，但是返回的仍然是在名字空间 `A` 中调用 `n`。



## 面向对象编程

在详细阅读下面的章节之前，我们假定读者已经掌握了面向对象编程（OOP）的基本知识，知道“封装”、“继承”和“多态”表示什么意思。如果还不具备这样的知识，建议先看看相关的书籍，了解 OOP 的相关知识。并且要求您具有 C++ 编程方面的经验。

TCL 自身的 80 多条核心命令是不支持面向对象的，但是存在很多扩展包，它们给 TCL 提供了面向对象的功能。这些扩展包中，面向对象支持最好最全面的是 Itcl。所以我们就围绕 Itcl 开展论述。

首先必须在脚本中包含 Itcl 程序包：

```
package require Itcl
```

## 定义类

使用 Itcl 定义类的方式和 C++ 非常类似。下面我们定义几个类来表示集合图形的类“形状”。

```
#-----
#File Name    : Shape.tcl
#Description : define the classes for shape.
#Author       : LeiYuhou
#Created      : 2005/4/2
#-----

package require Itcl

itcl::class CShape {
    #类的构造函数
    constructor {} {
        puts "Constructor"
        incr m_shapeCount    ;#增加形状计数
    }

    #类的析构函数
    destructor {
        puts "[Description] destructor..."
    }
}
```

```
incr m_shapeCount -1      ;#减少形状计数
}

#Draw 方法，画出该图形
public method Draw {} {
    puts "Draw [Description]"
}

#Description 方法，画出该图形
public method Description {} {
    return "A base shape"
}

public proc GetCounter {} {return $m_shapeCount} ;#查询当前总共有多少个形状
private common m_shapeCount 0      ;#类变量，形状计数器
}
```

定义类的命令是 `Itcl::class` 命令，它有两个参数：第一个是类名，第二个是类的定义体。在类的定义体中我们可以定义类的构造函数、析构函数、成员变量、成员函数。

类定义了之后，我们就可以创建其实例，也就是所说的对象，例如：

```
CShape s      ;#定义 CShape 的实例 s，对象名字为 s
s Draw        ;#调用对象方法 Draw
itcl::delete object s      ;#删除对象 s
```

可以看到，一个类定义成功之后，类名就成了一个普通的 TCL 命令。

定义类的实例后，实例名字也成为一个 TCL 命令。

## 构造函数

每一个类最多一个构造函数，每一个对象被生成的时候，构造函数都会被解释器自动的调用。其定义语法如下：

`constructor arglist ?init? body`

构造函数以关键字 `constructor` 开始，后面跟着构造函数的参数列表，然后是可选的初始化列表，最后是函数定义部分。

和普通的函数定义唯一的差别在于这里有一个可选的 `init` 部分：

1. 在这里我们可以直接调用基类的构造函数，对基类进行初始化。并且参数部分 `arglist` 中出现的变量可以在 `init` 部分直接使用。



2. 如果省略了 `init` 部分，并且该类有基类，那么解释器会自动用空的参数列表来调用基类的构造函数。如果基类构造函数不接受空的参数列表，那么就会抛出异常。
3. `init` 部分的代码总是在 `body` 之前被执行，这样将保证基类首先被构造好。

构造函数如果失败，那么就抛出异常。一般情况下，我们没有从其中返回什么东西，因为我们都用不着其返回值。

聪明的读者会问，C++中的构造函数可以重载，也就是说一个 C++类可以定义好多个不同参数列表的构造函数。TCL 是否也支持这样的特点？答案是不支持，TCL 中的一个类最多只能有一个构造函数。因为如果构造函数中使用 `args` 参数，就可以实现同样的功能。

## 析构函数

每一个 TCL 类可以有一个析构函数，当对象被删除的时候，会自动的调用析构函数。一般在析构函数中做一些清除资源的工作。其语法如下：

`destructor body`

析构函数总是以关键字 `destructor` 开始，后面跟函数体。要注意的是析构函数没有参数列表。对象被删除的时候，解释器会自动的先调用它自己的析构函数，然后按照顺序调用其基类的析构函数。

## 成员变量

和 C++一样，类中可以定义成员变量，按照类别可以分成两类：

1. 普通成员变量：类的每一个实例对象都有自己独立的一份变量数据；
2. 静态成员变量：类的所有实例对象都共享一份变量数据；

我们先介绍普通成员变量。其定义方式如下：

```
[public | protected | private] variable varname ?init? ?config?
```

关键字 `variable` 前面可以用 `public` 等来指明变量的可见性，其含义和 C++中含义完全一致。如果省略这些修饰符，那么变量就是 `private`，这一点和 C++完全一致。

`variable` 后面是变量名字，名字后是初始值 `init`，它是可选的。如果指定了 `init`，那么当生成新对象的时候，其成员变量就会初始化为该值。

最后是可选的 `config`，它是一个脚本块。当我们调用对象的 `configure` 方法来更改该变量值的时候，这一块脚本就会自动执行。例如下面是一个简单的类：

```
itcl::class A {  
  public variable m_a 0 { puts "Config : [set m_a]" }  
  variable m_b 1  
}
```

```
A a      ;#生成对象实例 a
a configure -m_a 200      ;#修改对象 a 的成员变量 m_a。
a configure -m_b 200      ;#发生错误，m_b 是 private 的
puts [a cget -m_b]        ;#错误，m_b 是 private 的
```

一个类的成员变量如果是 `public` 的，可以通过对象的 `configure` 和 `cget` 方法来设置和查询该成员变量的值。这个成员变量名字见面加上一个“-”，就成了该类的属性。

`configure` 的时候，如果变量有 `config` 属性，那么就会执行 `config` 语句块。例如刚才的例子中，`a configure -m_a 200` 语句，会把对象 `a` 的成员变量 `m_a` 设置为 200，并且会执行其对应的 `config` 代码，所以会打印出“Config:200”的结果。

成员变量可以直接在类的成员方法中使用，不用象名字空间中的过程那样：必须加上 `variable` 语句来说明这些变量。

关于初始化值，有读者会问：如果需要把成员变量初始化为一个数组，该当如何？怎样去写这个 `init` 部分？回答是“在使用 `variable` 定义变量的时候不需要给出 `init` 的值，而是在 `constructor` 中去设置”。在构造函数中你可以随心所欲的调用 `set` 和 `array set` 来设置成员变量的值。

## 静态成员变量

在 C++ 类中，可以定义 `static` 类型的成员变量。对于这样的变量，所有的类实例对象都共享一个变量。TCL 中也可以定义这样的变量，语法如下：

```
[public | protected | private] common varname ?init?
```

和 `variable` 类型变量定义类似，不同的是没有 `config` 部分了。因为对于 `common` 类型的变量，是不能通过对象的 `config` 方法来进行设置的。如果前面没有 `public` 等关键字，那么该变量就是 `private` 类型的。

类 `CShape` 中就定义了这样类型的一个变量 `m_shapeCount`。用来统计程序中形状对象的总数。可以看到，这样类型的变量和名字空间中的 `variable` 定义的变量比较类似，可以在其过程中直接使用而不用 `variable` 来声明。

如果变量 `m_shapeCount` 是 `public` 类型的，那么可以在对象外面直接使用该变量，比如我们可以在类 `CShape` 外面这样写：

```
puts $CShape::m_shapeCount
```

如果不是 `public` 类型，那么就会发生错误。

关于 `init` 初始化值的部分，读者可能又会问了。静态成员变量是所有对象共享的，如果要初始化为一个 TCL 数组，它的 `init` 部分应该如何写？总不能象 `variable` 变量那样在构造函数中去初始化吧。没错，不能在构造函数中作初始化，也没有必要。只需要在 `class` 中初始

化就行了，例如：

```
itcl::class CTest {  
    public common a      ;#定义变量 a  
    array set a {1 "ONE" 2 "TWO" 3 "THREE"} ;#初始化  
}  
  
puts [array get CTest::a]
```

执行上面的代码，结果如下：

```
1 ONE 2 TWO 3 THREE
```

还有读者会问：可不可以类的外面用专门的代码来进行初始化设置，就好比 C++ 中那样：

```
class CTest {  
    static int a[100];  
};  
  
int CTest::a[100] = {0,1,2,3,4};
```

对于 `public` 类型的变量，这样写是可以的。但是对于 `protected` 等类型，就不行了。最好的解决方法就是直接在类的定义过程中，调用 `array set` 或者 `set` 来完成。

## 对象方法

每一个类中可以定义多个方法，来对对象进行相关的操作。定义的语法如下：

```
[public | protected | private ] method methodName ?arglist? ?body?
```

和普通的过程定义类似，对象方法定义采用 `method` 关键字，前面是可选的方法可见性说明 `public` 等。如果省略 `public` 等关键字，那么该方法默认就是 `public` 类型。`method` 后面是方法的名字，然后就是参数列表，最后是过程体：这几点和前面介绍过的 `proc` 命令来定义过程的方法完全一样。

在 `class` 命令中定义对象方法的时候，可以省略掉最后的 `body`；或者将 `arglist` 和 `body` 两者一起省略。然后在类的外面使用 `itcl::body` 来定义方法的实现。这样就好比在 C++ 中，头文件定义类的接口，而在 `cpp` 文件中定义类的实现。

比如 `CShape` 中的过程 `GetCounter`，可以在 `class` 中如下定义：

```
public method GetCounter
```

然后在 `class` 外面定义其参数列表和实现代码，如下：

```
itcl::body CShape::GetCounter { } { return $m_shapeCount }
```

其中 `itcl::body` 命令的语法如下：

```
body className::methodName arglist methodbody
```

在使用 `body` 实现方法的时候，必须给出类名和方法名，这和 C++ 类似。否则解释器根本不知道这个方法是属于哪一个类的。

在 `method` 中，有一个 TCL 自动生成的变量可以使用，这就是“`this`”，其功能和 C++ 中的 `this` 指针完全一致。例如 `Shape` 中的方法 `Draw`，我们可以这样写：

```
public method Draw {} {  
    puts "Draw $this ->[$this Description]"  
}
```

因为 `this` 是一个变量，所以需要使用 `$` 字符来置换它，其值是本对象的名字。例如执行如下的代码：

```
CShape s;#定义 CShape 的实例 s，对象名字为 s  
s Draw      ;#调用对象方法 Draw  
itcl::delete object s      ;#删除对象 s
```

执行结果如下：

```
Draw ::s ->Base Shape  
CShape destructor...
```

可以看到，`$this` 被置换为对象名字：`::s`。同时我们可以在过程调用方法的时候，使用 `$this` 来进行调用，比如 `Draw` 里面调用 `Description` 就是这样处理，这和 C++ 中的：`this->Description()` 是完全类似的。

`this` 只能够在对象方法 `method` 中使用，在下面介绍的类方法中是无法使用的。

## 类方法

C++ 中定义方法的时候，如果在前面加上 `static` 进行修饰，那么这个方法就成了一个所谓的类方法（一般也叫做静态方法），这样的方法没有 `this` 参数，所以不能够直接操作类中的非静态成员变量和对象方法。也就是说，这样的方法不能够针对对象进行调用。在 Tcl 中也可以定义这样的方法，语法如下：

```
[public | protected | private ] proc procName ?arglist? ?body?
```

除了关键字 `method` 变成了 `proc` 之外，其他地方和 `method` 方法定义没有任何不一致的地方。也可以在定义的时候只给出名字，然后使用 `body` 来定义其实现过程。但还是存在如下差异：

1. 刚才介绍的 `this` 变量，在 `proc` 方法中不能够使用；

2. `proc` 方法中，不能够直接调用 `method` 类型的方法；
3. `proc` 方法中，不能够直接操作 `variable` 类型的成员变量；

例如 `CShape` 中的方法 `GetCounter`，就是 `proc` 方法。它简单的返回静态成员变量 `m_shapeCount` 的值。如果 `m_shapeCount` 是普通的成员变量，那么这个方法就会发生错误，抛出异常。类似，如果直接调用 `Description`，也会出错。

类方法可以在其他的 `method` 方法内部被直接调用，不过千万不要用 `$this` 来调用，因为我们一再强调了，类方法是不支持 `this` 的。如果我们这样写方法 `Draw`：

```
public method Draw {} {  
    puts "Draw [$this GetCounter] $this ->[$this Description]"  
}
```

上面代码通过 `$this` 来调用类方法 `GetCounter`，就会出错：

```
bad option "GetCounter": should be one of...  
s Description  
s Draw  
s cget -option  
s configure ?-option? ?value -option value...?  
s isa className  
  while executing  
"$this GetCounter"  
  (object "::s" method "::CShape::Draw" body line 2)  
  invoked from within  
"s Draw      "
```

类方法还可以在类外面调用，调用形式和 `namespace` 中过程一样，和 C++ 也比较类似，例如：

```
puts "Has total [CShape::GetCounter] Shapes"
```

会打印出：当前有多少个形状对象。

## 使用类和对象

类定义成功之后，就可以使用了，释放方式也就如下几种：创建对象、使用对象、删除对象和类。首先看看如何创建对象：

## 创建对象

创建对象的方法我们前面有过简单的介绍。其具体的语法如下所示：

```
className objName ?arg...?
```

`className` 是类名，后面跟着对象名字，然后是构造函数的参数列表。构造函数是由基类开始逐步调用的，`className` 的构造函数总是最后被调用。如果构造函数成功，那么在当前名字空间中，一个名字为 `objName` 的新命令就创建成功；并且该命令也将 `objName` 作为结果返回。如果在构造的时候发生错误，那么析构函数会被自动的调用，以前申请的资源会被释放，然后抛出异常。

为对象思索一个好的名字，很多时候是一件比较费脑筋的事情。特别是对对象比较多的时候，Itcl 为我们提供了“`#auto`”，自动取名字。例如：

```
set s [CShape #auto]      =>cShape0
set s [CShape sh_#auto]   =>sh_cShape1
```

可以直接使用`#auto` 的值作为对象的名字（第 1 条语句），也可以把`#auto` 嵌入到对象名字内部，作为名字的一部分（第 2 条语句）。ITcl 会进行相关处理，处理方法很简单，把`#auto` 进行置换，规则如下：类名的第一个字母小写，然后紧跟一个自动增加的计数器。

注意：上面两条语句执行后，变量 `s` 保存的值才是对象名字。很多人对这一点不明白，直接将 `s` 作为对象名字使用，结果就出错了。例如：

```
set s [CShape #auto]      =>cShape0
s Draw                    ;#会出错，s 是变量名字，其值才是对象名字；
$s Draw                   ;#正确；
```

通过上面方式创建的对象是在名字空间中创建的，具有类似全局对象的生命周期，即使你是在一个过程内部创建的，当过程退出的时候，对象也不会被自动的析构。怎样才能够定义一个类似 C++ 中的局部对象呢？ITCL 为我们提供了 `local` 命令，其语法如下，和直接创建对象很类似：

```
itcl::local className objName ?arg...?
```

请看下面的代码：

```
proc TestShape {} {
    CShape gShape      ;#全局的对象
    itcl::local CShape lShape      ;#局部对象
    lShape Draw
}

TestShape
itcl::delete object gShape
```

```
exit
```

上面的例子在过程 `TestShape` 中定义了两个 `CShape` 类型的对象，但是创建方式不一样：用 `itcl::local` 创建的是局部对象，在过程退出的时候会自动析构，但是 `gShape` 则不会，需要我们专门删除它。下面是执行结果：

```
Constructor.  
Constructor.  
Draw ->Base Shape:lShape  
Base Shape:::lShape destructor...  
Base Shape:::gShape destructor...
```

如果我们不调用 `itcl::delete object` 来删除对象 `gShape`，那么 `gShape` 的析构函数不会被调用，即使程序退出，也不会被自动调用。

## 使用对象

对象生成之后，使用对象的语法如下：

```
objName methodName ?arg...?
```

`objName` 就是我们刚才创建的对象名字，对象创建之后，在其被创建的名字空间中生成了一个名字就是对象名字的过程。参数 `methodName` 是对象类中所定义的方法名字，后面则是方法的参数。

`methodName` 除了可以是类中定义的方法之外，还可以是如下的内建方法：

1. `objName cget option`：读取对象成员变量的值，其中 `option` 的格式是“-varname”。
2. `objName configure ?option? ?value?...:` 如果给出了 `value` 参数，就设置成员变量的值；如果没有给出 `value`，那么就返回 `option` 选项的信息列表，格式如下：{选项名，初始值，当前值}。可以同时指定多个 `option-value` 为参数。

例如：

```
itcl::class A {  
    public variable a -1  
    public variable b  
}  
A m  
m cget -a      ;#返回-1  
m cget -b      ;#返回{undefined}  
m configure -a ;#返回列表{-a -1 -1}  
m configure -b ;#返回列表{-a <undefined> <undefined>}
```

```
m configure -b 200      ;#设置成员变量 b 的值为 200
m configure -b          ;#返回列表{-b <undefined> 200}
```

3. objName isa className: 判断一个对象是不是 className 类型。如果 objName 的类型名是 classNam, 或者是从 className 派生出来的, 那么该方法返回 1, 否则返回 0。

## 删除对象和类

可以使用 itcl::delete 命令删除类和对象。删除对象的时候命令如下:

itcl::delete object name ?name...?. 其中 name 是需要删除的对象名字, 可以一次删除多个对象;

删除类的格式如下:

itcl::delete class name ?name...?. 其中 name 是需要删除的类型, 可以一次删除多个类; 当删除某一个类的时候, 所有该类以及其子类的实例对象都会被析构。

delete 命令的例子, 我们到后面结合 find 命令一起来看看。

## 继承

ITCL 支持类的继承, 并且可以多重继承, 也就是说一个类可以有多个基类。多重继承一般会给我们带来一些麻烦, 所以在某些程序设计语言中不支持。我们也不推荐在 TCL 中使用多重继承。

下面是一个继承的例子, 我们从 CShape 中派生出 CPoint 类:

```
itcl::class CPoint {
    inherit CShape
    constructor { x y } {CShape::constructor} {      ;#构造函数
        set m_x $x
        set m_y $y
    }
    destructor { }      ;#析构函数

    public method GetX {} {return $m_x}
    public method GetY {} {return $m_y}
    public method Description {} {return "Point ($m_x,$m_y)"}

    public method Distance {pt}
```



```
protected variable m_x
protected variable m_y
}

#在类的外部定义成员函数 Distance，计算两点之间的距离
itcl::body CPoint::Distance { pt } {
    set x [$pt GetX]
    set y [$pt GetY]
    return [expr {sqrt(pow($x-$m_x,2) + pow($y-$m_y,2))} ]
}
```

在 class 定义中使用 inherit 命令来声明基类，C++中的继承有 public，protected 和 private 继承，但是实际上 public 之外的继承关系很少用到。在 TCL 中则都是 public 继承，其规则如下：

1. 基类中的 public 成员，在子类中都是 public 类型的；
2. 基类中的 protected 成员，在子类中都是 protected 类型的；
3. 基类中的 private 成员，在子类中则不可见，无法访问；

基类中定义的 common 类型以及 variable 类型的变量，如果在子类中可见，那么可以在子类的 method 方法中直接访问；但是如果要在类外部访问这些变量，必须通过基类名字来访问，例如：

```
puts $CShape::m_shapeCount    ;#正确（假如 m_shapeCount 是 public 类型）
puts $CPoint::m_shapeCount    ;#错误（即使 m_shapeCount 是 public 类型）
puts [CShape::GetCounter]     ;#正确
puts [CPoint::GetCounter]     ;#错误，没有定义这个方法。
```

如果在子类中定义了一个和基类某成员函数同名的函数，会出现什么情况？比如上面的 CPoint 类中定义了 Description 函数，但是基类 CShape 类中也有一个同名的函数。这就是我们下面要介绍的虚函数和面向对象的多态性。

## 虚函数

多态性是 OOP 的一个重要特征，而多态则通常是由虚函数来实现的。在 C++中，成员函数如果在声明前面加上 virtual 关键字，那么它就是虚函数。在 ITCL 中则没有这么复杂，ITCL 中所有的 method 类型的成员函数都是虚函数。如果按照 C++的思维，ITCL 中类的析构函数也是虚函数。

我们还是看看上面的例子：

基类 CShape 中有两个成员函数，Draw 以及 Description；类 CPoint 从 CShape 中继承，

并且也定义了成员函数 `Description`。当我们执行如下代码：

```
CPoint a 4 0      ;#定义 CPoint 的实例 a
a Draw            ;#调用 a 的 Draw 函数
```

因为 `Draw` 函数是在基类 `CShape` 中定义，所以 `CPoint` 继承了该函数，当 `CPoint` 的实例 `a` 调用 `Draw` 函数的时候，实际上也就是调用 `CShape::Draw` 函数，该函数内部调用了 `Description` 成员函数。问题来了：这里调用的 `Description` 是基类 `CShape` 的，还是子类 `CPoint` 的呢？

按照一般思维：基类 `CShape` 的函数 `Draw` 调用 `Description` 函数，肯定应该也是基类 `CShape` 的，不可能调用到 `CPoint` 的成员函数。但是实际上，上面两条语句的执行结果是：

```
Draw ->Point (4,0)
```

显然，这里调用的 `Description` 是 `CPoint::Description` 函数，因为我们通过 `CPoint` 的实例来调用 `Draw`。这，就是“多态性”。而 `Description` 就是虚函数。

对于虚函数，子类中的函数参数列表和基类中函数的参数列表应该一致吗？答案是“不一定”。但是作为一个约定俗成的规则，我们建议基类和子类中同名的函数，参数列表最好是一致的。例如上面例子中 `CPoint` 和 `CShape` 中的 `Description` 函数，参数都是空。如果我们把基类 `CShape` 的 `Description` 函数参数列表中加上几个参数，上面的两条命令也能够执行，但是这样会给我们带来不必要的麻烦。

## find：查找类和对象

`itcl::find` 命令可以找出当前已经定义的类或者对象名字。查找类的语法如下：

```
find classes ?pattern?
```

该命令会返回一个列表，其中包含了所有的 `itcl::class` 定义的类名；当前名字空间中的类会首先列出来，然后其他名字空间中的。如果给出了可选参数 `pattern`，那么就是采用 `string match` 匹配，只有能和 `pattern` 匹配上的名字才会出现在结果列表中。`pattern` 中可以包含名字空间修饰符，例如：

```
find classes ::*      ;#返回所有的类，带有名字空间修饰
```

查找对象的语法如下：

```
find objects ?pattern? ?-class className? ?-isa className?
```

这个命令用来查找符合条件的对象，返回的是对象名字的列表。后面的选项用来指定搜索条件：

1. `pattern` 表示匹配模式，如果指定该参数，那么将采用 `string match` 方式进行匹配，只有匹配上的对象名字才返回。
2. `-class className`：用来指定对象所属的类名，只有指定 `className` 类的实例才返回；
3. `-isa className`：用来指定对象的类名或者基类，如果对象是类 `className` 或者其子

类的实例，那么对象名字才返回。

请看看下面的例子：

```
CPoint a 4 0      ;#创建类 CPoint 实例 a
CPoint b 0 4      ;#创建类 CPoint 实例 b
CShape shape1
CShape shape2

puts "->All classes : [::itcl::find classes ::*]"
puts "->CPoint objects : [::itcl::find objects -class CPoint]"
puts "->CShape objects : [::itcl::find objects -class CShape]"
puts "->Objects belong CShape : [::itcl::find objects -isa CShape]"

itcl::delete class CPoint      ;#删除类 CPoint
puts [::itcl::find objects]    ;#打印出所有的对象
```

下面是程序的执行输出结果：

```
Constructor.
Constructor.
Constructor.
Constructor.
->All classes : ::CShape ::CPoint
->CPoint objects : a b
->CShape objects : shape1 shape2
->Objects belong CShape : a b shape1 shape2
Point (4,0) destructor...
Point (0,4) destructor...
shape1 shape2
```

可以看到::itcl::find objects -isa CShape 命令的结果，包括了 CShape 和 CPoint 两个类的实例，因为 CPoint 是 CShape 的子类。



## 程序库和程序包

一门语言附带程序库的功能是否强大，对其能否获得广泛应用其有着重要的影响。程序库是那些可以被多个应用程序使用的、公共的代码或者程序。例如 C/C++ 语言经过多年发展，有了 CRT Library、STL、boost 等通用库，以及 Windows 上的 MFC 等程序库。应该说正是这些功能强大无所不包的程序库，才使 C/C++ 语言到现在仍然是最具有活力的系统编程语言。现在正在流行的 C# 语言，其运行的 .net 平台中更是包含了一个巨大的程序库。

TCL 语言也有很多程序库。如何组织这些程序库，让多个程序可以调用？C/C++ 中的程序库是通过头文件和库文件来进行的，而 TCL 的程序库的变迁经历了两种形式：程序库（library）和程序包（package）。library 的方式比较原始和古老，现在一般都不采用了，package 方式则比较全面。我们在介绍这两种方式之前，先介绍一下 source 命令和 unknow 命令。

## source: TCL 中的#include

C/C++ 中使用库文件的方式，首先必须在源程序中#include 相关的头文件，编译通过后，在链接（link）阶段与相关的库文件（.lib、.obj 等）连接。在 TCL 中则没有所谓头文件和库文件，那么分散在其他多个文件的命令和程序，如何被我们的应用中加载引用呢？有一个命令可以完成类似#include 的功能，这就是 source。其命令语法如下：

source filename

唯一的参数是文件名，source 命令读取文件全部内容，然后交给 TCL 解释器来执行，source 命令的结果就是文件中最后一个命令的结果。如果执行的时候发生了异常，source 命令也会抛出异常。

source 命令在 library 和 package 两种模式中都得到了广泛的应用。

## unknow 方法

请先思考下面的问题：Windows 操作系统下，当我们执行命令 tclsh 启动一个 TCL 解释器之后，进入交互模式，然后在 TCL 的命令提示符下面输入 dir 命令，会怎样？得出结论后，然后动手实践一下来验证你的结论是否正确。

创建一个文件 test.tcl，其内容只有一行，就是 dir 命令，然后在命令提示符下面通过输入命令行 tclsh test.tcl 来执行这个 TCL 脚本，看看结果怎么样。

结果比较让人诧异：交互模式下，dir 命令能够执行，并且列出了当前目录下的所有内容，和一般情况下执行 dir 的结果并无差异；但是执行脚本，则报告了一个错误：invalid command name “dir”。怎么回事？dir 是标准的 DOS 内部命令，它不属于 Tcl 的命令。这样的命令按照常理是不能够被执行的，但是事实上在交互模式下却正确执行了，那为何在脚本

里面就不能执行了呢？其实，这都是 `unknown` 命令在作怪。

实际上，TCL 解释器在解释执行的时候，如果碰上了当前不认识的命令名字，都会调用一个命令：`unknown`；并且把这个未知命令连同其参数作为 `unknown` 命令的参数。`unknown` 命令一般是在 TCL 解释器初始化的时候来定义的。作为一个比较普遍的规则：

1. 每一个 TCL 应用程序都应该有一个 `init.tcl` 文件，该文件会做一些初始化的工作，这个文件都位于 `info library` 命令所返回的目录中。
2. `unknown` 命令就是在这个 `init.tcl` 文件中定义的。

例如我现在使用的 ActiveTcl，运行 `info library` 命令，返回解释器的库目录为：

```
% info library
C:/Tcl/lib/tcl8.4
```

在这个目录下存在文件 `init.tcl`，其中就定义了 `unknown` 命令。大家有兴趣可以直接看看这个命令的源代码。

TCL 解释器碰到不认识的命令之后，处理流程如下：

1. 首先看看 `unknown` 命令是否存在，不存在则抛出异常；
2. 然后调用 `unknown` 命令，未知命令及其参数都作为 `unknown` 的参数；
3. 检查命令字是否具有 “`namespace in scope ns cmd`” 这样的格式，如果是，则直接执行该命令，然后返回；
4. 如果没有定义全局变量 `auto_noload`，则调用 `auto_load` 命令来尝试从程序库中加载和寻找这个未知命令；如果找到则执行它，返回结果；
5. 如果 `auto_load` 没有找到该命令的实现，那么就判断是否是处在交互模式下；如果是交互模式：
  - a) 并且没有定义全局变量 `auto_noexec`，那么就调用 `auto_execok` 命令来判断这个未知命令是否是 DOS 内部命令或者是某一个可执行程序。如果是，则执行它们，并且返回执行结果。
  - b) 如果没有找到可执行命令文件，或者定义了 `auto_noexec`，那么就判断该未知命令是否是 “`!!`”、“`!(.+)$`” 等这样的格式；如果是，那么就调用对应的历史命令。
6. 如果不是交互模式，则抛出异常，退出 `unknown`。

上面是 `unknown` 命令的默认处理流程，我们可以自己重新编写这个命令，但是最好要保留原来的功能。因为其处理机制非常重要，保证了 TCL 程序库中定义的命令能够被自动的加载到 TCL 解释器中。完成这个功能的就是 `auto_load` 命令。

## auto\_load：加载程序库

`auto_load` 命令是在 `init.tcl` 中定义，在解释器初始化的时候被创建。当解释器碰到了未定义命令的时候，`unknown` 命令会被调用，在 `unknown` 中会调用 `auto_load` 命令来寻找这个

未定义命令。弄清楚 `auto_load` 的执行机制，会让我们更加清楚 TCL 程序库（library）的工作机制和自动加载的原理。

首先得介绍一下和 `auto_load` 相关的两个全局变量：

1. `auto_path`: 列表变量，其每一个元素是库文件所在的目录。这个变量在解释器初始化、执行 `init.tcl` 文件的时候被定义并且初始化，过程如下：
  - a) 如果有环境变量 `TCLLIBPATH`，其值将是 `auto_path` 的第一个元素；
  - b) 命令 `info library` 所返回的路径以及其父目录会加入到 `auto_path` 中；
  - c) 当前可执行文件所在目录的父目录下的 `lib` 子目录，会加入 `auto_path` 中；
  - d) 如果定义了变量 `tcl_pkgPath`，其中每一个元素也会加入到 `auto_path` 中。可见我们可以将库文件放到很多分散的目录中，只要在 `auto_path` 变量中有这些目录中就行了。
2. `auto_index`: 一个数组变量。其中每一个元素的下标是命令名字，对应值则是一条 TCL 语句。通过执行该语句能够定义对应的命令。例如：元素 `parrray` 对应的语句是“`source C:/Tcl/lib/tcl8.4/parrray.tcl`”。执行该语句后，`parrray` 命令就会被定义。一般而言，这个语句类似 `source xxxfile` 的格式。过程就是在 `xxxfile` 中被定义。

`auto_load` 的语法格式如下：`auto_load cmd ?namespace?`；其中的 `cmd` 就是解释器碰到的未知命令，当它被调用的时候：

1. 判断数组 `auto_index` 中是否存在下标为 `cmd` 的元素。如果有则执行对应的语句，然后判断命令 `cmd` 是否被定义；如果存在就返回 1，表示加载成功；
2. 接着判断 `auto_path` 变量是否存在，如果不存在就返回 0，表示加载失败；
3. 然后依次在 `auto_path` 所包含的每一个目录下，寻找名字为“`tclIndex`”的文件，判断这个文件是否是有效文件。如果有效，那么就执行这个文件。
4. 然后再次判断 `auto_index` 中是否存在下标为 `cmd` 的元素，如果存在，则执行对应的语句，然后判断命令 `cmd` 是否存在（是否被定义了）；如果存在，就返回 1。
5. 最后返回 0，表示自动加载失败。

这就是 `auto_load` 命令的全部执行过程。可以看到，除了前面介绍的 `auto_path` 和 `auto_index` 变量之外，还涉及到了一个文件 `tclIndex`。这个文件的内容很简单：就是用来设置 `auto_index` 变量的值。例如在我的系统上：

```
% foreach {k v} [array get auto_index] {puts "$k = $v"}
tcl_startOfNextWord = source C:/Tcl/lib/tcl8.4/word.tcl
parrray = source C:/Tcl/lib/tcl8.4/parrray.tcl
pkg_mkIndex = source C:/Tcl/lib/tcl8.4/package.tcl
::safe::Lappend = source C:/Tcl/lib/tcl8.4/safe.tcl
history = source C:/Tcl/lib/tcl8.4/history.tcl
::safe::AliasSubset = source C:/Tcl/lib/tcl8.4/safe.tcl
.....
% set auto_path
```

```
C:/Tcl/lib/tcl8.4 C:/Tcl/lib
```

上面是 `auto_index` 和 `auto_path` 两个变量的内容。在 `C:/tcl/lib/tcl8.4` 目录下存在一个文件 `tclIndex`，其内容如下：

```
# Tcl autoload index file, version 2.0
# This file is generated by the "auto_mkindex" command
# and sourced to set up indexing information for one or
# more commands. Typically each line is a command that
# sets an element in the auto_index array, where the
# element name is the name of a command and the value is
# a script that loads the command.

set auto_index(auto_reset) [list source [file join $dir auto.tcl]]
set auto_index(tcl_findLibrary) [list source [file join $dir auto.tcl]]
set auto_index(auto_mkindex) [list source [file join $dir auto.tcl]]
set auto_index(auto_mkindex_old) [list source [file join $dir auto.tcl]]
.....
```

里面就是一堆的 `set auto_index(cmd)` 这样的命令。可以看到，第一条 `auto_reset` 命令可以通过执行 “`source auto.tcl`” 语句来被定义。根据我们前面对 `unknown` 和 `auto_load` 命令的分析，如果解释器碰到了 `auto_reset` 命令并且该命令没有被定义，那么就会通过 `unknown` 和 `auto_load` 来执行 `source auto.tcl`，而 `auto_reset` 命令就是在文件 `auto.tcl` 中被定义。所以 `auto_load` 能够加载成功，并且在 `unknown` 中会执行这个 `auto_reset` 命令。如果再次调用 `auto_reset` 命令，就不会再次进行同样的加载了，因为这个命令已经被定义了，可以被直接调用。

`tclIndex` 文件内容是不是很简单？确实很简单，读者可能会觉得自己动手写一个也不成问题，确实可以自己动手写，不过如果需要手动编写 `tclIndex` 文件，一定要注意文件的第一行内容必须是：

```
# Tcl autoload index file, version 2.0
```

`auto_load` 就是根据第一行的内容来判断该文件是否是有效的 `tclIndex` 文件。

大部分情况下没有必要手动编写，因为 TCL 程序库中已经为我们定义了一个命令：`auto_mkindex`，来帮助我们自动生成 `tclIndex` 文件。

## 创建程序库（library）

经过前面的分析，相信大家对 TCL 自动加载 `library` 的过程有了深入的了解。下面我们来实践一下，自己动手来编写一个程序库。



## 创建库文件

创建库文件，是创建程序库最主要的工作，除了编写程序之外，还需要进行调试。我们计划在两个文件中分别定义几个不同的过程。

文件 `perf_func.tcl` 文件的内容如下：

```
#////////////////////////////////////  
#File Name : perf_func.tcl  
#Author    : LeiYuhou  
  
proc perf_Test1 {} {  
    puts "perf_Test1"  
}  
  
proc perf_Test2 {} {  
    puts "perf_Test2"  
}
```

文件 `perf_func.tcl` 文件的内容如下：

```
#////////////////////////////////////  
#File Name : test_func.tcl  
#Author    : LeiYuhou  
  
proc fTest1 {} {  
    puts "fTest1"  
}  
  
proc fTest2 {} {  
    puts "fTest2"  
}
```

我们在两个文件中总共定义了四个过程。目的仅仅是为了说明 `library` 的创建过程，所以四个过程看起来很弱智。

## 创建 `tclIndex` 文件

我的 `ActiveTcl` 安装在 `C:/tcl` 目录下。那么我们自己创建的库文件应该存放在什么地方

呢？答案是：任何你认为方便的地方都可以，只要你做好相关设置。这里我们把刚才创建的两个文件放到 E:/work/script 目录中。

创建 tclIndex 文件的方法有两种：自己动手写，或者自动生成。选择前者的人一般都有不可告人的目的；我们选择后者。

打开命令提示符，执行 tclsh 进入 TCL 的交互模式，然后执行命令 auto\_mkindex：

```
% auto_mkindex e:/work/script *.tcl
%
```

命令执行后，在 E:/work/script 目录下就生成文件 tclIndex。如果对 auto\_mkindex 命令不放心，可以打开这个文件检查一下。这个文件的内容如下：

```
# Tcl autoload index file, version 2.0
# This file is generated by the "auto_mkindex" command
# and sourced to set up indexing information for one or
# more commands. Typically each line is a command that
# sets an element in the auto_index array, where the
# element name is the name of a command and the value is
# a script that loads the command.

set auto_index(perf_Test1) [list source [file join $dir perf_test.tcl]]
set auto_index(perf_Test2) [list source [file join $dir perf_test.tcl]]
set auto_index(fTest1) [list source [file join $dir test_func.tcl]]
set auto_index(fTest2) [list source [file join $dir test_func.tcl]]
```

可以看到我们创建的四个命令，都会在全局变量 auto\_index 中有一席之地。

## 其他设置

事情还没完！还需要最后的设置，否则 auto\_load 无法找到我们定义的过程。怎样让 auto\_load 命令找到呢？答案很简单，就是将 tclIndex 文件所在的目录加入到全局变量 auto\_path 中。我们有几种方法供选择：

第一种，设置或者修改环境变量 TCLLIBPATH。如果这个环境变量原来没有定义，那么就设置它为我们库文件所在目录：“E:/work/script”；如果已经被设置了，那么就把我们库文件所在的目录添加到原来变量值的后面，与原来的值用空格隔开。

例如在我系统上，启动 tclsh，进入交互模式：

```
C:\>set TCLLIBPATH=          #取消环境变量 TCLLIBPATH 的定义

C:\>tclsh                    #启动 TCL 解释器，进入交互模式
```

```
% fTest1                #执行命令 fTest1
invalid command name "fTest1" #未定义，并且找不到，所以命令无效
% set auto_path          #看看 auto_path 变量
C:/Tcl/lib/tcl8.4 C:/Tcl/lib    #不包含目录 E:/work/script
% exit

C:\>set TCLLIBPATH="e:/work/script"    #设置环境变量

C:\>tclsh                #启动 TCL，进入交互模式
% fTest1                 #执行命令 fTest1，成功
fTest1
% fTest2                 #执行命令 fTest2，成功
fTest2
% set auto_path          #查看变量 auto_path，第一个元素
e:/work/script C:/Tcl/lib/tcl8.4 C:/Tcl/lib
% set auto_index(fTest1) #看看变量 auto_index，下标为 fTest1
source e:/work/script/test_func.tcl
% exit

C:\>
```

上面的过程在简单明了，我就不废话了。

除此之外的第二种方法是：将我们的库文件连通 `tclIndex` 文件移动到 `auto_path` 中的某一个目录中去。要注意的是，这个目录中必须没有同名的文件。例如在我的系统上：

```
E:\Work\script>copy *.* C:\tcl\lib    #复制到 c:\tcl\lib 目录中
.\perf_test.tcl
.\tclIndex
.\test_func.tcl
已复制          3 个文件。

E:\Work\script>set TCLLIBPATH=    #删除环境变量 TCLLIBPATH

E:\Work\script>tclsh            #启动解释器，进入交互模式
% fTest1                       #执行命令 fTest1，成功
fTest1
% set auto_path                #看看变量 auto_path
C:/Tcl/lib/tcl8.4 C:/Tcl/lib
% set auto_index(fTest1)       #看看变量 auto_index(fTest1)
```

```
source C:/Tcl/lib/test_func.tcl
%
```

这也是一个不错的方法，不过容易对原来的函数库造成干扰。所以我们推荐第一种方法，设置操作系统环境变量 `TCLLIBPATH` 的值。这是最简单直观的方法。

除了上面介绍的两种方法之外，还有很多其它的方法。只要你完全掌握了 `TCL` 自动加载函数库的原理，就可以想出自己的方法。下面是两种其他的考虑：

1. 修改系统自带的 `init.tcl` 文件，在其中修改 `auto_path` 变量，通过 `lappend` 命令将我们程序库的路径添加到 `auto_path` 列表的末尾；
2. 在解释器执行了 `init.tcl` 之后，开始执行应用程序脚本之前，修改 `auto_path` 变量，在其中添加一个元素；

上面是创建一个程序库的基本步骤。首先创建库文件，然后使用 `auto_mkindex` 来生成 `tclIndex` 文件，然后做一些相关的设置，使解释器的自动加载机制能够找到这个文件。这样我们的程序库就创建成功。

## 交互模式下执行命令

`TCL` 解释器处在交互模式下时，会表现出一些特殊的行为，前面我们做了一些介绍。它可以执行 `DOS` 内部命令，例如 `dir`，`cls` 和 `copy` 等，也可以执行外部的可执行文件，例如 `xcopy.exe`、`notepad.exe`、`ping.exe` 等。例如：

```
% ver

Microsoft Windows XP [版本 5.1.2600]

% date
当前日期: 2005-05-05 星期四
输入新日期: (年月日)

% ping 127.0.0.1
Pinging 127.0.0.1 with 32 bytes of data:

Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
```

```
Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

```
%
```

事实上，这些命令都不是 TCL 的命令，它们之所以能够被执行是因为 `unknown` 命令做了特殊的处理。具体流程请参考前面 `unknown` 方法的描述。只有 TCL 碰到不认识的命令，并且 `auto_load` 加载失败，并且解释器处在交互模式下的时候，才会判断这个命令是不是 DOS 外部命令或者可执行文件。这个判断过程是由 `auto_execok` 过程来完成的，其语法格式如下：

```
auto_execok cmdname
```

`cmdname` 是未定义的命令，如果 `auto_execok` 判断它是内部命令或者可执行文件，就会返回一个带有完整路径的字符串，否则返回空字符串。判断是否是可执行文件的时候，会在环境变量 `PATH` 中进行查找，判断是否存在这样的可执行文件。例如：

```
% auto_execok dir
C:/WINDOWS/system32/cmd.exe /c dir
% auto_execok ping
C:/WINDOWS/system32/ping.EXE
% auto_execok ping 127.0.0.1
wrong # args: should be "auto_execok name"
%
```

除了上面的内部命令和外部可执行文件之外，还可以有如下的简单记法：

1. `!!`：两个冒号，重复执行刚才执行的那条命令；
2. `!n`：重复执行数字 `n` 所指定的历史命令，`-1` 代表刚才执行的前一条命令，`1` 表示第一条命令
3. `^xxx^yyy^`：`xxx` 是一个正则表达式，它表示将刚才执行的前一条命令进行 `regsub` 替换，将所有 `xxx` 用 `yyy` 替换掉，然后执行替换后的命令。

看看下面的几个例子，就会非常清楚：

```
% set a 100
100
% set b 200
200
% set c [expr $a*$b]
20000
% history info           ;#查看命令历史记录
    1  set a 100
    2  set b 200
    3  set c [expr $a*$b]
    4  history info
% !!                    ;#重复执行刚才执行的命令
```

```
history info
1  set a 100
2  set b 200
3  set c [expr $a*$b]
4  history info
5  history info
% !1                ;#执行历史记录中第一条记录
set a 100
100
% !2                ;#执行历史记录中第二条记录
set b 200
200
% ^b^e^            ;#刚才的命令 set b 200, 把 b 替换成为 e 后执行
set e 200
200
%
```

有些时候我们并不想在交互式解释器中执行外部命令，通过设置 `auto_noexec` 这个全局变量就可以了。例如：

```
% set auto_noexec 1        ;#设置 auto_noexec 变量，可以是任意值
1
% dir                      ;#执行内部命令 dir，会失败
invalid command name "dir"
% ping.exe 127.0.0.1        ;#执行可执行文件 ping.exe，会失败
invalid command name "ping.exe"
%
```

## 程序包（package）

前面已经详细的介绍了 `library` 的使用原理和创建过程。这种 TCL 函数库的方式历史悠久，到现在仍然有使用，不过现在有一种更加强大和方便程序库的方式，就是 `package`，这种方式现在获得了广泛的应用。基本上网上很多共享程序库都是采用 `package` 的方式来发布的。与自动加载库的方式相比，`package` 提供了版本控制机制：同一个程序库可以同时提供多个不同的版本，让不同的应用程序各取所需。

## package 的版本号

Tcl 程序包的版本号是一个用 “.” 格开的多个数字组成的字符串，例如下面几个字符串都是合法的版本号：2，2.0，2.0.0，2.13.45。虽然 TCL 没有对版本号包含数字的个数进行限制，但是作为一个普遍规则，版本号最好不要超过 4 个数。一般情况下我们只需要两个数即可。

版本号是可以比较大小的。其中，最左边的数对版本号的大小影响最大，可以看成主版本，后面的是小版本，如果小版本缺失，则默认为 0。例如：2.3 和 2.3.0 是相同的；2.3.1 则大于 2.2.9。

版本号越大，表示该版本越新，功能也可能越强大，越稳定，bug 越少。但是要注意的是，新版本应该与老版本兼容。例如某个应用程序使用的是程序包 A 的 1.0 版本，如果有朝一日 A 的作者对 A 进行了较大的更改，发布了版本 2.0，作为一个道德准则：用 2.0 替换 1.0 的版本后，使用 A1.0 写成的应用程序应该不作任何更改就能够在 A2.0 上正常运行。这就是所谓的向后兼容。

一个包的不同版本，可以在一个解释器内部共存，供应用程序有针对性的选用。

## 使用 package

如果要使用其它程序包，只需要在调用程序包中的命令之前使用 `package require` 命令将程序包引入即可，就好比 C 语言中使用 `#include` 来包含头文件一样。这是比 `library` 方式要多做的一点工作。该命令的语法如下：

```
package require ?-exact? pkgname ?version?
```

其中各个参数的意义如下：

1. `pkgname` 是我们需要的程序包名字；包名是大小写敏感的；
2. `version` 是我们需要加载的版本号，其取值不同，结果也不同：
  - a) 如果省略该参数，TCL 会寻找 `pkgname` 的最大版本进行加载。
  - b) 如果指定该参数，那么主版本号与 `version` 相同并且整个版本比 `version` 大的版本，都是可以接受的。如果当前可用的最大版本都小于 `version`，那么就会加载失败；
  - c) 如果指定该参数，同时指定 `-exact` 参数，那么只有和 `version` 参数完全一致的程序包，才是可以接受的。

请看下面的例子：

```
C:\Tcl\lib\test1.0>tclsh
% package require itcl          ;#包名大小写敏感，所以加载失败
can't find package itcl
```

```
% package require Itcl      ;#省略版本加载 Itcl, 返回版本号 3.2
3.2
% package require Itcl 3.1   ;#加载 Itcl3.1, 也能成功, 返回 3.2
3.2
% package require Itcl 3.0   ;#加载 Itcl3.0, 也能成功
3.2
% package require Itcl 2.9   ;#加载 Itcl2.9, 则失败, 因为主版本不对
can't find package Itcl 2.9
% package require Itcl 3.3   ;#加载 Itcl3.3, 失败, 因为最大才 3.2
can't find package Itcl 3.3
% package require -exact Itcl 3.1 ;#-exact 选项加载 3.1, 失败
can't find package Itcl 3.1
% package require -exact Itcl 3.0
can't find package Itcl 3.0
% package require -exact Itcl 3.2 ;#-exact 选项加载 3.2, 成功
3.2
%
```

虽然这里介绍 `package require` 命令煞费苦心, 但是实际上我们用的时候, 都很少使用 `-exact` 选项和 `version` 参数, 只是简单的给出包名, TCL 会自动的加载最新的版本。

## 创建 package

这里我们将 `library` 中介绍的几个命令, 用 `package` 的方式来重写。创建其他程序包的过程与此也基本类似。在介绍的过程中, 我们来逐步了解 `package` 内部的工作机制和原理。

## 创建程序库文件

将原来的两个文件改写后如下:

```
#####
#File Name : perf_func.tcl
#Author    : LeiYuhou

puts "file perf_func.tcl Initialed." ;#输出一句话, 表示文件初始化

proc perf_Test1 {} { puts "perf_Test1" }
```



```
proc perf_Test2 {} { puts "perf_Test2" }

package provide test 1.0
```

文件 test\_func.tcl 改写后如下：

```
#####
#File Name : test_func.tcl
#Author    : LeiYuhou

puts "file test_func.tcl Initialed." ;#我们输出一句话，表示初始化

proc fTest1 {} { puts "fTest1" }
proc fTest2 {} { puts "fTest2" }

package provide test 1.0
```

与上一章 library 中的同名文件相比差异不大，最后多了一行命令：

```
package provide test 1.0
```

这一行命令的意思就是：本文件提供了程序包 test，其版本是 1.0。一般情况下，每一个库文件脚本中，需要有一个 package provide 命令。它可以出现在文件的任何地方，不过最好在开头或者最后。

除此之外，在文件开始还有一个 puts 语句，当文件被执行的时候，会打印这么一句话，表示本文件被解释器读取并且被执行了。

## pkgIndex.Tcl

光有上面两个文件还不行，我们必须将它们复制到 TCL 解释器能够找到的地方。我在 C:/tcl/lib 目录下创建子目录 test1.0，并且将我们刚才创建的两个文件复制到这个子目录中。然后我们要做的工作是在该目录中创建一个 pkgIndex.tcl 文件。

这个文件有什么用途的？

非常有用。当命令 package require test 1.0 被执行的时候，解释器会在 auto\_path 所包含的所有目录以及它们的直接子目录中寻找 pkgIndex.tcl 文件，并且执行所有找到的 pkgIndex.tcl 文件。在这些文件中，会告诉解释器如果你需要的程序包恰好是我提供的程序包，那么请执行我提供给你的脚本语句。执行它之后，你需要的程序包就找到了。

pkgIndex.tcl 文件的内容，决定了包加载的两种不同形式：直接加载和滞后加载。我们先看简单的直接加载。

## 直接加载

调用命令 `pkg_mkIndex` 可以创建 `pkgIndex.tcl` 文件，如下：

```
% pkg_mkIndex -direct "c:/tcl/lib/test1.0" *.tcl
%
```

其中选项参数 `-direct` 表示直接加载，第二个参数表示库文件所在的目录，第三个参数是文件名模板，这里是 `*.tcl`，所有能够和这个模板匹配上的文件，都会被 `pkg_mkIndex` 命令分析。命令执行成功之后，会在同一个目录下创建一个 `pkgIndex.tcl` 文件，内容如下：

```
# Tcl package index file, version 1.1
# This file is generated by the "pkg_mkIndex -direct" command
# and sourced either when an application starts up or
# by a "package unknown" script.  It invokes the
# "package ifneeded" command to set up package-related
# information so that packages will be loaded automatically
# in response to "package require" commands.  When this
# script is sourced, the variable $dir must contain the
# full path name of this file's directory.

package ifneeded test 1.0 "
[list source [file join $dir perf_test.tcl]]
[list source [file join $dir test_func.tcl]]"
```

文件前面是注释，后面只有一条命令：`package ifneeded`，跟着包名 `test`，版本号 `1.0`，以及一串 TCL 语句。它表示如果：如果解释器需要的是程序包 `test` 的 `1.0` 版本，那么就请执行最后面的 TCL 语句。执行之后，`test1.0` 就被加载了；如果解释器正在寻找的不是 `test1.0`，那么就直接跳过去，对后面的语句不需要理睬。

我们可以看到，后面的语句就是两条 `source` 命令，分别执行两个库文件。这两个文件通过 `source` 命令被解释器执行之后，程序包 `test1.0` 就已经存在了。程序包中提供的过程就能够被正常的调用了。

例如：

```
C:\>tclsh                                ;#启动解释器
% fTest1                                  ;#调用命令 fTest1，会失败
invalid command name "fTest1"
% package require test 1.0                ;#加载 test1.0 程序包
file perf_func.tcl Initialed.             ;#执行文件 perf_func.tcl
file test_func.tcl Initialed.             ;#执行文件 test_func.tcl
1.0                                        ;#返回 test 的版本号
```

```
% fTest1                ;#执行命令 fTest1，成功!!
fTest1
% perf_Test1            ;#执行命令 perf_Test1，成功!!
perf_Test1
%
```

从上面的结果可以看到，执行 `package require test 1.0` 命令的时候会马上加载两个库文件。这也是“直接加载”这种说法的来历。

其优点是：所有的库文件在 `package require` 的时候会全部加载到解释器中，简单方便；而且以后调用库中命令的时候，这些命令都已经是定义好的。

缺点在于：如果一个库由很多文件组成，包含很多命令，但是我们只需要调用其中一个文件中的某一个命令，那么“直接加载”方式会不管三七二十一，将所有用不着的文件都加载进来，增加了系统的负担和开销。

为了避免“直接加载”方式的缺点，TCL 提供了“滞后加载”的包加载模式，也可以称之为“按需加载”

## 滞后加载

只需要更换一下程序包中的 `pkgIndex.tcl`，就可以将加载方式更改为“滞后加载”。我们还是使用 `pkg_mkIndex` 命令来创建 `pkgIndex.tcl` 文件。请看：

```
C:\Tcl\lib\test1.0>tclsh
% pkg_mkIndex -lazy -verbose "c:/tcl/lib/test1.0" *.tcl
file perf_func.tcl Initialed.
successful sourcing of perf_test.tcl
commands provided were perf_Test1 perf_Test2
packages provided were {test 1.0}
processed perf_test.tcl
file test_func.tcl Initialed.
successful sourcing of test_func.tcl
commands provided were fTest1 fTest2
packages provided were {test 1.0}
processed test_func.tcl
%
```

和前一节相比，命令 `pkg_mkIndex` 的参数变了，多了一个 `-verbose` 参数，由 `-direct` 变成了 `-lazy`。`-verbose` 选项表示在创建 `pkgIndex.tcl` 文件的过程中输出相关信息；而 `-lazy` 参数则表示包加载模式是“滞后加载”。

生成的 `pkgIndex.tcl` 文件内容如下：

```
# Tcl package index file, version 1.1
```

```
# This file is generated by the "pkg_mkIndex -lazy" command
# and sourced either when an application starts up or
# by a "package unknown" script.  It invokes the
# "package ifneeded" command to set up package-related
# information so that packages will be loaded automatically
# in response to "package require" commands.  When this
# script is sourced, the variable $dir must contain the
# full path name of this file's directory.

package ifneeded test 1.0 {
    tclPkgSetup $dir test 1.0 {
        {perf_test.tcl source {perf_Test1 perf_Test2}}
        {test_func.tcl source {fTest1 fTest2}}
    }
}
```

看起来比上一节中的 `pkgIndex.tcl` 要复杂一些。主要差别在于 `package ifneeded` 的最后一个参数，就是提供程序包的 TCL 语句。这里用到了 `tclPkgSetup` 命令，其参数语法如下：

`tclPkgSetup dir pkgname version files`

参数 `dir` 是包含库文件的目录；`pkgname` 是程序库的名字；`version` 是程序库的版本号。参数 `files` 则是一个比较复杂的列表，其每一个元素也是列表，都符合如下的模板格式：`{ filename action proclist }`，其中 `filename` 是文件名，`action` 可以为“source”或者“load”，`proclist` 则是 `filename` 所表示文件中定义的所有过程的列表。如果 `filename` 是一个 TCL 脚本，那么 `action` 是 `source`，如果 `filename` 是一个用 C/C++ 语言写成的二进制程序库，那么 `action` 是 `load`。

`tclPkgSetup` 命令定义在 `package.tcl` 文件中，是 TCL 标准库提供的一个命令。我们可以看到它的源代码：

```
proc tclPkgSetup {dir pkg version files} {
    global auto_index

    package provide $pkg $version
    foreach fileInfo $files {
        set f [lindex $fileInfo 0]
        set type [lindex $fileInfo 1]
        foreach cmd [lindex $fileInfo 2] {
            if {[string equal $type "load"]} {
                set auto_index($cmd) [list load [file join $dir $f] $pkg]
            } else {
                set auto_index($cmd) [list source [file join $dir $f]]
            }
        }
    }
}
```

```
    }  
  }  
}
```

看懂这段代码不难，这个命令根据传入的参数修改了全局变量 `auto_index`。这是一个什么变量？怎么冒出来的？faint！回忆一下前面介绍的 `library` 的工作原理吧。程序包 `pkg` 中提供的所有命令，都会在 `auto_index` 数组中出现一个对应的元素，下标就是命令名字，对应的值则是一个脚本语句，比如“`souce xxx`”之类。

可以看到，当 `tclPkgSetup` 命令执行结束后，程序包文件并没有被马上 `source` 到解释器中，而只是仅仅把每一个命令添加到了 `auto_index` 数组中。只有当以后需要调用这些命令的时候，解释器发现这些命令没有被定义，于是就调用 `unknown` 过程，`unknown` 过程会调用 `auto_load` 过程，`auto_load` 会搜索 `auto_index` 变量，找到这个未知命令对应的值，然后执行这个“`source xxx`”或者“`load xxx`”语句。于是命令就被定义成功，然后就可以正常调用了。请看下面的例子：

```
C:\Tcl\lib\test1.0>tclsh  
% package require test 1.0      ;#加载程序包 test1.0  
1.0  
% fTest1                        ;#调用过程 fTest1  
file test_func.tcl Initialed.  ;#只有文件 test_func.tcl 被 source  
fTest1  
% set auto_index(fTest1)       ;#查看变量 auto_index(fTest1)  
source C:/Tcl/lib/test1.0/test_func.tcl  
% fTest2                        ;#调用同一文件中定义的 fTest2 过程  
fTest2  
% set auto_index(perf_Test1)    ;#查看 auto_index(perf_Test1)  
source C:/Tcl/lib/test1.0/perf_test.tcl  
% perf_Test1                   ;#调用 perf_Test1  
file perf_func.tcl Initialed.  ;#文件 perf_Test.tcl 被 source  
perf_Test1  
%
```

可以清楚的看到，“滞后加载”的程序包被 `package require` 的时候，两个文件都没有被执行。而只有在程序包中定义的某一个过程被调用的时候，才会加载定义这个过程的源程序，而包中的其它文件都不会被 `source` 加载。

这就是“按需加载”的程序包的实现机制和原理。它充分利用了 `unknown` 命令。

## 深入了解 package

前面我们已经详细了解了如何创建一个程序包，并且分析和介绍了“直接加载”和“按需加载”两种方式下，`pkgIndex.tcl` 文件的制作方式和加载原理。了解这么多之后就基本上可以很熟练的制作和使用程序包了，如果你不想继续深入了解程序包，可以跳过这个章节，本章节中会进一步了解 `package` 的内部机制。

在 `TCL` 解释器内部保留了一个 `ifneeded` 数据库。这个数据库的主键（`Key`）由“程序包名字”和“版本”共同组成，每一个 `Key` 对应一个特定版本程序包的相关信息，包括“加载它需要执行的脚本”、“是否已经加载”等。我们可以通过执行下面的脚本来查询出 `TCL` 解释器中这个数据库的内容：

```
#=====
#File Name   : pkgIfNeeded.tcl
#Author      : LeiYuhou

package require Itcl          ;##!! 注意这里 require 了一个程序库!!

proc pkgQuery {} {
    set index 0
    set pkgNames [package names]    ;#所有的 package 名字

    foreach pkg $pkgNames {
        set version [package versions $pkg]    ;#该程序包的所有版本号
        foreach v $version {
            set script [package ifneeded $pkg $v]    ;#对应的加载脚本
            puts "$index ==>($pkg , $v) : $script"
            incr index
        }
    }
}

pkgQuery
```

在我的 `ActiveTcl 8.4` 版本上运行上面的脚本，总共打印出了 187 条记录，下面是部分运行结果：

```
159 ==>(parser , 1.4) :   load C:/Tcl/lib/tclparser1.4/tclparser14.dll
160 ==>(md5 , 2.0.0) :   source C:/Tcl/lib/tcllib1.6/md5/md5x.tcl
161 ==>(md5 , 1.4.3) :   source C:/Tcl/lib/tcllib1.6/md5/md5.tcl
```

注意上面脚本的第一条命令是“`package require Itcl`”。试着去掉这条命令后重新执行这个脚本，会发现没有任何输出。看来一条简单的 `package require Itcl` 命令，背后所作的事情决不是想象中的那么简单！

每次 `package require` 命令被调用的时候，TCL 解释器都会首先看看，需要的这个程序包是不是已经被加载了，如果已经加载，那么就什么都不做，直接返回。否则就会到这个 `ifneeded` 数据库中寻找满足要求的 **Key**（需要加载的包名和版本）。这时分成两种情况：

1. 如果找到了 **Key**，那么就会判断这个包是否已经被加载了；如果已经加载，则直接返回版本号，如果没有加载，则执行对应的脚本语句来加载包。
2. 如果没有在库中找到符合要求的 **Key**，那么就会调用过程 `tclPkgUnknown`，该过程会搜索 `auto_path` 中所有目录及其一级子目录中的 `pkgIndex.tcl` 文件，并且使用 `source` 命令执行所有找到的 `pkgIndex.tcl` 文件。然后，再次到数据库中去寻找满足要求的 **Key**。如果还是找不到，那么就返回错误！

了解这一点之后，上面的执行情况就不难解释。实际上在解释器刚启动的时候，`ifneeded` 数据库中没有任何记录。所以上面的程序在去掉第一条命令“`package require Itcl`”之后，执行结果为空。但是加上这条命令之后，因为需要加载 `Itcl` 程序包，但是 `ifneeded` 数据库中没有任何对应的记录，所以 `tclPkgUnknown` 命令会被执行，这样一来 `auto_path` 中所有目录以及其一级子目录中的 `pkgIndex.tcl` 文件都会被执行一遍，`ifneeded` 数据库中就会增加所有能够找到的程序包和版本信息记录。所以就能够出现 180 多条记录了。

这样实现的好处是：下次加载其他的程序包的时候，首先在内存中查找这个数据库（基本上都能够找到对应的记录，除非你需要一个不存在的程序包），而不用再次根据 `auto_path` 重新查找文件系统，这样就大大提高了系统的效率。这样的设计实在是非常的巧妙，让人忍不住拍掌叫好！

过程 `tclPkgUnknown` 定义在 `C:/tcl/lib/tcl8.4/package.tcl` 文件中，感兴趣的读者可以直接查看其源代码，了解其工作机制！

## package 命令

`package` 命令是 TCL 提供的核心命令之一，我们前面已经详细了解了它的几个用法：`package require`、`package provide`、`package ifneeded`、`package names` 等。下面我们介绍它的其他用法：

`package forget ? package package .....?`

该命令会将参数 `package` 相关信息从解释器中完全删除，包括 `ifneeded` 和 `provide` 信息。如果某一个程序包被加载了之后然后调用 `forget` 来清除其相关信息，我们仍然可以调用这个程序包中定义的命令，因为 `forget` 只是清除了相关信息。例如：

```
% package ifneeded test 1.0      ;#还没有加载，所以返回空
% package require test           ;#加载 test 1.0
```



```
1.0
% package ifneeded test 1.0      ;#查看对应的初始化加载代码
    tclPkgSetup C:/Tcl/lib/test1.0 test 1.0 {
        {perf_test.tcl source {perf_Test1 perf_Test2}}
        {test_func.tcl source {fTest1 fTest2}}
    }
% package forget test 1.0        ;#调用 forget，清除相关信息
% fTest1                          ;#调用 test 中的命令 fTest1，成功
file test_func.tcl Initialed.
fTest1
% perf_Test1                      ;#调用 test 中的 perf_Test1，成功
file perf_func.tcl Initialed.
perf_Test1
% package ifneeded test 1.0      ;#再次查看 ifneeded 数据库，为空!!!
% package require test          ;#重新加载它试试看。
1.0
% package ifneeded test 1.0      ;#再次查看，还是为空!!! ???
%
```

可以看到，forget 程序包 test 之后，还是可以调用其中的 fTest1 等命令。因为这些命令要么在加载程序包的时候被定义，要们记录在 auto\_index 变量中了；而这些信息是无法忘记，无法 forget 的。

上面例子中的最后一个命令大家肯定觉得非常奇怪：明明才调用 package require test 成功，为什么查看其初始化加载脚本，却还是返回为空呢？仔细分析一下上面的操作过程，就不难理解了：在我们调用 forget 之后，马上调用了 test 程序包中的命令 fTest1，因为这是第一次调用 fTest1，所以文件 test\_func.tcl 被解释器 source 执行，而这个文件中最后一行代码就是“package provide test 1.0”，它告诉解释器：“我已经提供了程序包 test1.0”，相当于该程序包又被加载。所以后面再次调用“package require test”的时候，解释器会发现该程序包已经存在，所以就直接返回。根本不会去搜索 auto\_path 目录中的 pkgIndex.tcl 文件。所以会造成 ifneeded 数据库中仍然没有相关的记录。

package present ?-exact? pkgname ?version?

该命令的参数和 package require 类似，它返回解释器中 pkgname 包是不是已经存在了，如果存在就返回版本号，否则就抛出异常！例如：

```
% package require test          ;#加载程序包 test
1.0
% package present test          ;#看看当前是否有 test 程序包
1.0
% package present test 1.0
```



```
1.0
% package present test 2.0    ;#版本冲突
version conflict for package "test": have 1.0, need 2.0
% package present Itcl       ;#抛出异常
package Itcl is not present
%
```

package ifneeded package version ?script?

这个命令用来查询或者修改 ifneeded 数据库。如果给出了 script 可选参数，那么就是修改 ifneeded 数据库；否则则是查询数据库，返回 script。两种用法我们前面都用到了。一般而言这个命令只在每一个程序包的 pkgIndex.tcl 文件中使用。



## 事件驱动

事件驱动，顾名思义就是由各种事件驱动来驱动程序运行。这种类型程序的主体是一个无限循环，在这个循环中不停的等待各种事件的发生；当某种事件发生之后，主循环根据事件的类型来调用对应的事件处理函数。

在当初的 DOS 时代里，使用 Borland C++ 编写事件驱动程序，可不是一件容易的事情。后来 Borland 公司提供了一个 Turbo Vision 的程序库，使用它能够比较方便的在 DOS 字符界面下编写事件驱动程序。比如：著名的 Borland C++ 的 IDE 好像就是采用它写成的。Windows 下各种 GUI 类型程序基本上都是事件驱动的。

TCL 语言内建了事件驱动的机制，所以编写事件驱动程序非常简单。我们首先从一个 `after` 命令开始。

## after 命令

命令 `after` 用来在代码中进行延时相关的操作，它有如下几种命令格式：

1. `after ms`
2. `after ms ?script script script ...?`
3. `after cancel id`
4. `after cancel script script script ...`
5. `after idle ?script script script ...?`
6. `after info ?id?`

第一种格式中，参数 `ms` 必须是一个整数，表示延时的毫秒数。这个命令只是简单暂停解释器的执行，其效果等同于 Win32 的 API 函数 `::Sleep`。在解释器休眠期间，发生的任何事件都不能被处理。`after` 命令会阻塞直到超时才返回。

第二种格式中，后面跟着可选的 `script` 参数，如果指定了 `script`，那么它们会被 `concat` 连接起来作为一个脚本，当超时时间到了之后会执行这个脚本。要注意的是这个脚本只会执行一次，所以这种格式的 `after` 命令好比一个超时定时器。这个 `after` 执行后会立刻返回，返回值是一个 `afterID`，用来标识这个 `after` 定时器。`script` 会在全局层次上执行，可以理解成 `uplevel #0...` 的层次。

第三和第四种格式用来取消某个 `after` 超时操作。其中参数 `id` 则是前面某个 `after` 调用返回的标识。第四种格式中的所有 `script` 参数会被 `concat` 连接起来，然后和所有延迟的 `after` 的命令进行匹配，如果匹配上就删除这个延迟操作，如果没有匹配任何操作，那么这个 `after` 命令没有任何效果。

第五种格式，后面的参数会被连接起来作为一个脚本字符串，这个脚本只会被执行一次。只有当下次进入事件循环，并且恰好没有事件需要处理的时候，就会执行这个脚本。这个命

令也返回一个 ID。

最后一个 `after info` 命令用来返回某个还没有超时的 `after` 命令的信息。如果指定了 `id` 参数, 这个 `id` 所表示的 `after` 参数必须表示的是还没有超时或者被 `cancel` 的 `after` 操作返回的值; 如果没有指定 `id`, 那么返回一个列表, 每一个列表都是一个 `id` 信息。

第一个 `after ms` 命令在不是事件驱动的情况下也能够被执行, 但是第二种则必须是在事件驱动的程序中, 否则后面的脚本永远得不到执行的机会。例如我们进入 `tclsh` 交互式执行界面, 进行如下测试:

```
C:\>tclsh
% after 3000                                ;#延时等待 3 秒
% after 3000 "puts Hello"                  ;#三秒钟后执行 puts Hello
after#0
% after 3000 "puts Hello2"                 ;#三秒钟后执行 puts Hello2
after#1
% after info                               ;#现在快 10 秒过去了, 还没有看到 Hello
after#1 after#0
```

`tclsh` 启动的解释器不是事件驱动的, 所以上面我们看到延迟执行的命令都没有得到执行。如何进入事件驱动? 或者说如何构造一个事件循环呢? 这就需要用到一个 `vwait` 命令。

## vwait 进入事件循环

`vwait` 用来进入事件循环, 直到指定变量被设置。其语法如下:

`vwait varname`

`varname` 是一个全局变量的变量名。`vwait` 进入事件循环, 不停的响应各种事件, 直到某个事件处理程序设置了 `varname` 这个变量后, `vwait` 才返回。

下面我们来看看 `vwait` 和 `after` 结合, 实现两个定时器的程序例子:

```
#=====
#File    : timer.tcl
#Autor   : LeiYuhou

set v 0 ;#事件循环的控制变量
set c 0 ;#计数变量

#-----
#OnTimer 是 after 的响应操作过程, delay 参数是延迟的毫秒
proc OnTimer { delay } {
    global c v
```

```
incr c
puts "count = $c.TIMER = $delay"
if { $c==100 } {
    set v 0 ;#修改变量 v，退出事件循环
} else {
    after $delay "OnTimer $delay" ;#再次启动 after
}
}

after 200 OnTimer 200 ;#启动周期为 200ms 的定时器
after 500 OnTimer 500 ;#启动周期为 500ms 的定时器

#调用 vwait，进入事件循环
vwait v
```

上面代码中：变量 `c` 用来计数，表示总共发生了多少次 `after` 事件；变量 `v` 用来控制事件循环，一旦此变量被设置，那么就退出事件驱动；过程 `OnTimer` 是定时器的处理过程，在其内部对变量 `c` 递增，一旦达到 100 就修改变量 `v`，否则就继续启动下一个 `after`。最后调用了两次 `after` 命令，启动两个定时器；然后调用 `vwait` 来进入事件驱动。代码执行结果如下：

```
count = 1.TIMER = 200
count = 2.TIMER = 200
count = 3.TIMER = 500
.....
count = 98.TIMER = 200
count = 99.TIMER = 200
count = 100.TIMER = 500
```

退出 `vwait` 命令的条件是：其等待的变量被设置，而不是被修改。例如刚才的代码中，变量 `v` 初始值为 0，`c` 达到 100 之后我们设置 `v` 的值仍然是 0，结果 `vwait` 就退出了。

看到这里，是不是觉得 TCL 的事件驱动简单得让人难以置信？没错，脚本语言的优势就在这里。当别人陷入在 C/C++ 的指针泥潭中挣扎的时候，采用 TCL 的您，可能已经轻舟已过万重山了！



## 输入输出系统

输入和输出相比大家都不陌生。说起它，我就想起来大学第一次写 Pascal 程序的时候，总要写出这样的代码：

```
Program Test(input,output);
Begin
    Write('Hello,World!');
End.
```

老实说，第一行的 `input` 和 `output` 两个参数让我迷惑了很久，一直都不明白它们的用途。后来才慢慢的知道它们叫做输入和输出。

输入和输出总是和文件系统紧密相连。现代操作系统中，所有的设备几乎都被抽象成文件系统。但是不同操作系统下文件系统格式不同，例如 Unix 和 Windows、MAC 的目录等都有一定的差异。好在 TCL 语言比较好的解决了这个问题。本章节我们会详细的介绍如何在 TCL 中操作文件系统以及读写文件。

## 操作文件系统

TCL 中能够非常方便的操作文件、目录。包括：查询和设置文件属性、复制、删除以及路径名字的操作等。所有这些都是通过一个 `file` 命令来完成，其语法都非常简单，所以我们这里只是按照分类，列举出该命令的各种用法：

### 文件属性操作

命令	用法描述
<code>file atime name ?time?</code>	返回文件 <code>name</code> 最后被读取的时间；该命令在 FAT 文件系统上无效（返回数据可能不正确）；
<code>file mtime name ?time?</code>	设置或者修改文件 <code>name</code> 的最后被修改时间。 <code>time</code> 参数表示 1970/1/1 到现在的秒。设置最后修改时间，和那个著名的 <code>touch</code> 命令完成类似的功能；
<code>file attributes name</code> <code>file attributes name ?opt?</code> <code>file attributes name ?opt vla? ?opt val?...</code>	查询或者设置文件的属性。第一个命令查询所有的属性；第二个命令查询指定的属性；第三个设置属性的值。
<code>file executable name</code>	看看这个文件是否是可执行的，是就返回 1；

file readable name	返回文件 name 是否是可读的。
file writable name	返回文件是否是可写的。
file exists name	文件如果存在并且当前用户有搜索它所在目录的权限，那么就返回 1，否则返回 0；
file isdirectory name	name 是否是一个目录，是就返回 1，否则返回 0
file isfile name	如果 name 是一个文件，那么就返回 1，否则返回 0
file size name	返回文件的大小（字节数）
file stat name varname	查询得到文件 name 的相关属性，并且存放到数组 varname 中。
file system name	返回文件 name 所在文件系统的类型；
file type name	返回 name 所表示的文件类型；

上面的命令都是用来查询设置文件或者目录的属性的。不知大家有没有使用 C Runtime Library 编程的经历，在 C 语言的库函数中操作文件的函数基本上在这里都能够找到。下面是几个例子：

```
% set fp {d:\tmp\aa.txt}      ->d:\tmp\aa.txt
% file exist $fp              ->1
% file readable $fp           ->1
% file writable $fp           ->1
% file isdirectory $fp        ->0
% file isfile $fp             ->1
% file size $fp               ->442
% file stat $fp v ; parray v
v(ctime) = 1116761242
v(dev)   = 3
v(gid)   = 0
v(ino)   = 0
v(mode)  = 33188
v(mtime) = 1116545258
v(nlink) = 1
v(size)  = 442
v(type)  = file
v(uid)   = 0
% file system $fp             ->native NTFS #这是一个 NTFS 系统
```

下面我们再来看看 file attributes 命令：

```
% file attributes $fp
```



```

-archive 1 -hidden 0 -longname D:/tmp/aa.txt -readonly 0 -shortname D:/tmp/aa.txt
-system 0
% foreach {opt val} [file attributes $fp] {puts "  $opt = $val"}
  -archive = 1
  -hidden = 0
  -longname = D:/tmp/aa.txt
  -readonly = 0
  -shortname = D:/tmp/aa.txt
  -system = 0
% file attributes $fp -shortname      #查询短文件名
D:/tmp/aa.txt
% file attributes $fp -readonly 1    #让文件具有只读

```

## 路径操作

有些时候分析表示文件路径名的字符串，是很有必要的工作。C 语言中也包含了这样的库函数，TCL 中也有对应的实现：

命令格式	用法描述
<code>file dirname name</code>	返回 <code>name</code> 文件的所在目录名字。这只是一个字符串的处理，不管文件 <code>name</code> 是否存在！
<code>file extension name</code>	返回文件的扩展名；
<code>file join name ?name..?</code>	把 <code>name</code> 拼接起来组成一个完成的路径，目录分隔符号因不同操作系统而异；
<code>file nativename name</code>	返回一个符合操作系统特点的路径名；
<code>file normalize name</code>	返回标准化的路径名字；
<code>file rootname name</code>	返回除了最后扩展名的文件名字
<code>file split name</code>	和 <code>file join</code> 恰好相反，将路径 <code>name</code> 拆分层各个独立的目录单元；
<code>file tail name</code>	返回路径 <code>name</code> 中剔除目录名后的最后文件名

请看下面的例子：

```

% file dirname {d:\tmp\a.txt}      ->d:/tmp
% file extension {d:\tmp\a.txt}    ->.txt
% file split {d:\tmp\a.txt}        ->d:/  tmp  a.txt
% eval file join [file split {d:\tmp\a.txt}] ->d:/tmp/a.txt
% file join c:\\ windows system notepad.exe ->c:/windows/system/notepad.exe
% file nativename c:/windows/cmd.exe ->c:\windows\cmd.exe

```

```
% file normalize c:/windows/system32/././cmd.exe
C:/WINDOWS/cmd.exe
% file rootname c:/windows/cmd.exe      ->c:/windows/cmd
% file tail c:/windows/cmd.exe          ->cmd.exe
```

这里面需要解释的是 `file nativename` 命令。在 TCL 的目录结构中，采用字符“/”作为目录结构的分隔字符，这和 Unix 操作系统是一致的；但是 Windows 中则是采用“\”作为分隔。有些时候我们需要将 TCL 中的目录转换到操作系统特定的格式，那么 `file nativename` 则能完成这个功能。

## 操作文件目录

复制文件的操作采用 `file copy` 来完成，格式如下：

```
file copy ?-force? ?-? source target
```

```
file copy ?-force? ?-? source ?source .....? targetDir
```

第一个用来赋值单个文件到另外一个文件；第二个则是赋值 `n` 个文件 ( $n \geq 1$ ) 到另外一个目录。

```
file delete ?-force? ?-? pathname ?pathname ... ?
```

上面的命令用来删除一个或者多个文件、或者目录；通过指定 `-force` 选项，可以强制删除非空目录。对于只读文件，即使没有 `-force`，该命令也能 `delete` 掉它。

```
file make dir ?dir...?
```

用来在文件系统中创建目录结构。例如：

```
% cd c:\\      ->进入 C:\目录
% file mkdir aa bb cc      ->创建三个目录： C:\aa, C:\bb, C:\cc
% file delete aa bb cc      ->再删掉这三个目录
% file mkdir aa/bb/cc      ->一次创建三个目录： C:\aa\bb\cc
% cd aa/bb/cc      ->进入这个最深的目录 c:\aa\bb\cc
% file delete -force c:/aa      ->删除 C:\aa，也能成功
% pwd      ->C:\，删除的时候，当前目录会自动回退
```

```
file rename ?-force? ?-? source target
```

```
file rename ?-force? ?-? source ?source ...? targetDir
```

上面的两个命令用来修改文件名、目录名；或者移动文件或者目录；例如：

```
% pwd      #当前目录为： D:/tmp
```

```
% dir .
D:\tmp 的目录

2005-05-20  07:27                442 aa.txt
2005-05-20  07:27                442 bb.txt
2005-05-22  21:52    <DIR>                t
                2 个文件                884 字节
                3 个目录  4,141,834,240 可用字节
% file rename aa.txt bb.txt t    #移动文件到目录 t 中
% dir t
D:\tmp\t 的目录

2005-05-20  07:27                442 aa.txt
2005-05-20  07:27                442 bb.txt
                2 个文件                884 字节
                2 个目录  4,141,834,240 可用字节

% dir .
D:\tmp 的目录

2005-05-22  21:53    <DIR>                t
                0 个文件                0 字节
                3 个目录  4,141,834,240 可用字节
```

## 搜索目录

有些时候我们需要在目录中寻找匹配一定模式的文件或者子目录，这可以使用 `glob` 命令来完成。下面命令得到当前目录下所有的 `tcl` 脚本文件：

```
glob -directory . -types f -tails -- *.tcl
```

其中选项 `-directory .` 表示在当前目录下搜索；`-types f` 表示搜索文件，如果是 `-types d` 表示搜索目录名；`-tails` 表示只返回文件名，不要目录名。`*.tcl` 则是匹配模板，用来匹配文件名。

例如上面命令的执行结果是：

```
perf_test.tcl  pkgIfNeeded.tcl  test_func.tcl
```

返回一个列表，三个元素都是搜索到文件名，如果我们下发如下的命令：

```
glob -directory -types f *.tcl
```

和刚才命令相比，少了-tails 选项，那么结果中就会多出目录名，如下：

```
./perf_test.tcl ./pkgIfNeeded.tcl ./test_func.tcl
```

-types 选项的参数是一个列表，表示只搜索指定的类型。可以为：

类型字符	说明
b	block special file, 块文件
c	character special file, 字符文件
d	directory, 目录
f	plain file, 文件
l	symbolic link, 符号链接
p	named pipe, 命名管道
s	socket 文件
r	Readable, 具有可读权限的
w	writeable, 具有写权限的
x	executable, 可以被执行的
readonly	ReadOnly file, 只读的
hidden	hidden, 隐藏的

例如下面的命令：

```
glob -directory . -types {f d r w} *
```

表示当前目录下查找文件或者目录，并且具有可读和可写的权限。

当 glob 没有发现任何匹配目标的时候，会抛出异常。选项-nocomplain 可以避免异常抛出。如果没有发现任何匹配的目标，那么就返回空的列表。例如：

```
% catch {glob -directory c:/ *.tcl} msg ;#C:/下没有*.tcl 这样的文件
1
% puts $msg
no files matched glob pattern "*.tcl" ;#抛出了异常
% catch {glob -directory c:/ -nocomplain *.tcl} msg
0 ;#加上-nocomplain 之后，不抛出异常
% puts $msg ;#看看 glob 的执行结果，是一个空列表

%
```

有时候我们想搜索当前目录下所有子目录中的特定文件，例如当前在 C:/tcl/lib 目录下，里面有很多扩展包子目录，现在我们想找出各个扩展包的 pkgIndex 文件：

```
% foreach t [glob -tails -directory . -types f ./*/pkgIndex.*] {puts $t}
./as_style/pkgIndex.tcl
./Banking/pkgIndex.tcl
```

.....

`glob` 命令中使用的文件匹配模式，在 TCL 中叫做 `glob` 模式，这一点我们前面在介绍 `switch` 命令的时候已经介绍过了，这里不再罗嗦。要注意的是，一个 `glob` 命令的最后可以带有多个模式字符串参数，这样我们就可以一次搜索多种类型的文件。下面再举几个模式的例子：

1. `glob -directory . -types f abc[0-9].tcl net*.exe`；表示当前目录下搜索两类文件名，一个是以 `abc` 开始，后面跟一个数字的 `tcl` 文件；一类是 `net` 开始的 `exe` 文件。
2. `glob -directory . test{case,suit}.tcl`；表示搜索 `testcase.tcl` 或者 `testsuit.tcl` 两个文件。注意这里用到了 `glob` 模式的大括号语法，`{a,b,c...}` 其中用逗号分隔的多个字符串，表示可以匹配其中的任意一个。注意逗号两边不能够有不必要的空格，除非你期望匹配文件名中本身就有空格。如果有空格，整个模式必须用引号括起来。

这个命令的具体语法，还是去参考 TCL 的联机帮助手册。

## 读写文件

TCL 中读写文件的概念以及方式和 C 语言非常的类似，常见的操作有打开、关闭、读写、定位等等。我们逐一介绍：

### 打开和关闭

`open` 命令用来打开文件，其格式如下：

1. `open filename`
2. `open filename access ?permissions?`

其中，`filename` 表示需要打开的文件名；`access` 是需要进行的操作类型，可以是如下的字符串：

`r`：只读；使用该模式的时候文件必须已经存在了；`access` 省略时就默认为 `r`。

`r+`：读写；文件必须已经存在；

`w`：只写，如果文件已经存在，就覆盖它并且长度截为 0；否则就创建它；

`w+`：读写；如果文件已经存在，就覆盖它并且长度截为 0；否则就创建它；

`a`：追加写；打开后文件初始指针定位在文件末尾；如果不存在就创建新文件；

`a+`：读写；其它和 `a` 类似；

可选参数 `permissions` 用来设置文件的许可权限，如果 `open` 过程中需要创建新文件，那么这个参数就有意义。默认情况下为八进制 `0666`。表示所有人都可以读写这个文件，这个参数和 Unix 文件系统紧密相关，一般在 Windows 下面很少使用。

如果 `open` 命令执行成功，就返回打开的文件句柄。这个句柄得存起来留着后面使用。如果失败那么就抛出异常，可以使用 `catch` 命令来捕获。

打开的文件最后必须关闭，特别是那些打开写的文件，否则最终写入文件的数据可能不完整。关闭文件使用 `close` 命令，格式如下：

`close filehandle`

其中参数 `filehandle` 就是刚才 `open` 命令返回的文件句柄。当一个句柄被 `close` 之后，它就不能再用了，成为一个无效的句柄。当一个句柄被关闭的时候，所有被缓冲但是还没有被写入的输出数据都被强制写入，所有被缓冲但是还没有被读出的输入就绪数据会被丢弃。

事实上当进程退出，解释器销毁的时候，所有打开的文件句柄会自动关闭。但是一个有道德的程序员，都应改主动的关闭不用的文件句柄，让自己的进程消耗尽量少的系统资源。

下面看一个例子，创建一个空的文件 `a.txt`：

```
% set fd [open a.txt w+]
file8f8d10
% close $fd
% glob *.txt
a.txt
```

我们使用 “w+” 模式打开一个文件后，马上关闭。然后 `glob` 命令搜出该文件确认它确实存在。

## 文件读写命令

从文件中读出数据的命令有 `gets`、`read`。`gets` 用来读入一行数据，`read` 则可以读入指定大小的数据。

### gets 命令

`gets` 命令的格式如下：

`gets channelId ?varname?`

第一个参数是打开的文件句柄；如果指定第二个参数，它表示变量名，用来存放读入的数据，返回读入的字符个数；如果不指定 `varname`，那么读取的结果就作为命令结果返回。

我们可以把输入文件当成一个缓冲区，`gets` 命令每次读取一行数据，直到碰到了换行符号，但是返回的结果中会丢弃换行符号；如果在读取一行的过程中碰到了文件结尾，那么就返回所有当前已经得到的数据。

如果文件句柄是阻塞模式，`gets` 命令会一直阻塞直到读到了换行或者文件结果才返回，如果文件句柄不是阻塞模式并且当前没有一个完整得行（也就是缓冲区中没有换行符），那么 `gets` 命令马上返回，并且不消耗缓冲区的任何数据。

如果指定参数 `varname`，因为到达文件末尾或者没有一个完整行，那么 `varname` 将得到

空字符串并且 `gets` 命令会返回-1。

## read 命令

`read` 命令的有两种格式，格式如下：

1. `read ?-nonewline? channelId`
2. `read channelId numChars`

第一种格式，`read` 命令读取 `channelId` 中的所有数据，直到文件末尾。如果指定了选项 `-nonewline` 并且文件最后一个字符是换行符，那么就丢弃这个字符。

第二种格式，`read` 命令读取参数 `numChars` 所指定数量的字符并且返回，除非整个文件中剩余的字符数量比 `numChars` 少，在这种情况下，剩余多少个字符就返回多少个字符。要注意的是如果通道 `channelId` 被配置成多字节形式，那么读出的字符串的字符个数可能和字节数不一致。

命令的两种形式，都返回实际读取到的字符串内容。

如果通道 `channelId` 被配置成非阻塞模式，那么 `read` 的行为就会发生变化：`read` 不会读入用户所要求的字符数，而是读取当前输入缓冲区中所能够被读取的字符，然后返回。并且选项 `-nonewline` 选项只有在达到文件末尾的时候才生效。

## puts 命令

说完了读之后，看看写操作。只有一个 `puts` 命令供我们来写文件，格式如下：

`puts ?-nonewline? ?channelId? string`

它用来将参数 `string` 表示的内容写入到文件句柄 `channelId` 中，如果 `channelId` 省略，表示写入标准输出 `Stdout`。选项 `-nonewline` 表示写入 `string` 之后不要写入换行字符，如果不指定这个选项，每调用一次 `puts` 都会在写入 `string` 之后再自动的写入一个回车字符。

## flush 命令

TCL 会将写出的数据缓冲起来，例如调用 `puts` 往文件写入一些数据，`puts` 返回之后，这些数据可能不会马上出现在磁盘上，而是保留在内部缓冲区中。只有当缓冲区满了或者文件关闭的时候，缓冲区的数据才真正的写入到磁盘或者其它设备中。我们可以调用 `flush` 命令来强制将缓冲区的内容写入到设备。其格式如下：

`flush channelId`

参数 `channelId` 就是文件句柄。如果 `channelId` 配置成阻塞模式，那么 `flush` 会一直等到所有的数据都被写入到设备之后才返回；如果是非阻塞模式，那么 `flush` 会马上返回，被缓

冲的数据会在后台被尽快的写入到设备中。

## eof 命令

eof 用来判断某一个文件句柄的文件指针是否到了文件末尾。格式如下：

eof channelId

如果最近一次读操作达到了文件末尾，那么它就返回 1，否则返回 0；

下面我们看一个例子，它读入一个文本文件所有的行，然后将它们排序之后，写入到另外一个文件中。源代码如下：

```
#-----
#File name: sortfle.tcl
#Author : LeiYuhou

set INFILE "dircontent.txt"      #输入文件的名字
set OUTFILE "dircontent_sort.txt" #输出文件名

set buf ""
set fd [open $INFILE r]
while {[eof $fd]==0} {
    # 依次读取所有的行，放入列表 buf 中
    lappend buf [gets $fd]
}
close $fd

#对文件内容进行排序
set newbuf [lsort -ascii -increasing $buf]

set ofd [open $OUTFILE w]
foreach ln $newbuf {
    puts $ofd $ln
}
close $ofd
```

## 文件读写当前位置

一个打开的文件可以看成是一个流，对其读写都有一个当前位置的问题。例如当文件使用



r 方式打开的时候，当前位置就在文件头；当使用 read 读取了 20 个字节之后，当前位置就自动的后移到文件头后面 20 字节处。

跟文件读写位置有关的命令有两个：tell 和 seek。seek 用来设置读写位置，格式如下：

seek channelId offset ?origin?

参数 channelId 是一个打开的文件句柄，offset 表示偏移的字节数，可以为负数，参数 origin 表示参数 offset 的相对位置。可以为 start、current 和 end，分别表示文件开头，当前位置和文件结尾，如果省略则默认为 start。

命令 tell 的格式如下：

tell channelId

参数 channelId 和 seek 命令的参数 channelId 一样，是一个打开的文件句柄。这个命令返回这个文件的当前读写位置（相对于文件头的偏移字节数）。

看下面的例子：

```
% set fd [open a.txt r]          => file8f8d10
% tell $fd                      => 0
% gets $fd                      => 驱动器 C 中的卷是 SYSTEM
% tell $fd                      => 27
% gets $fd                      => 卷的序列号是 2003-3495
% tell $fd                      => 52
% seek $fd -52 current          #当前后移 52 字节，回到开头
% tell $fd                      => 0
% gets $fd                      => 驱动器 C 中的卷是 SYSTEM
% seek $fd 52 start
% gets $fd
% gets $fd                      => C:\ 的目录
% tell $fd                      => 67
```

要注意的是 seek 和 tell 中的位置偏移量都是以字节（byte）为单位，而不是字符。因为字符有单字节字符和多字节字符的分别。

seek 命令还有一个副作用：清除缓冲区。当 seek 被调用的时候，如果输出缓冲区中有数据，这些数据会被写入到文件中；如果输入缓冲区中有数据没有被读出，那么这些数据会被丢弃。

## 文件模式配置

在 Windows 操作系统下，文件有文本文件和二进制文件的差别，但是 Unix 操作系统中则没有这种差异。在 TCL 中，是否文本文件是可以通过命令来进行配置的；除了这一点之外，还可以对打开的文件句柄进行各种属性和模式的配置，达到一些特定的目的。这是通过

fconfigure 命令来完成的，该命令格式如下：

1. fconfigure channelId
2. fconfigure channelId name
3. fconfigure channelId name value ?name value ...?

其中参数 channelId 表示一个有效的文件句柄，name 表示模式名字，value 则表示模式的值。第一种格式，用列表的形式返回文件句柄 channelId 的所有配置信息；第二种，返回 name 所指定的配置信息；第三种，则是用来设置文件句柄的配置。

可以配置的属性有如下几种：

配置项	取值说明
-blocking	boolean 类型，true 表示阻塞式读写，false 表示异步式读写。
-buffering	表示文件缓冲的模式，可以取值为： full：只有内部缓冲区满了或者调用 flush，才真正输出； line：当缓冲区接收到了换行字符后，才自动写入； none：每一次输出操作的时候都会写入设备；
-buffersize	缓冲区的字节数；
-encoding	表示文件的编码。当读入或者写出的时候，TCL 会自动的在文件编码和 UniCode 编码之间自动转换；当文件打开时默认设置为操作系统相关的编码；
-eofchar	表示文件结束的字符。值可以是一个字符，也可以用两个元素的列表来分别为 input 和 output 指定。一般情况下为空。
-translation	其值格式有两种：mode、或者 {inMode,outMode}。前者是后者 in 和 out 一致的简化设置方式。它用来设置文件或者通道的换行识别以及转换方式，可以为：auto、binary、cr、lf 或者 crlf。通过设置该属性，可以将文件设置为所谓的“文本文件”或“二进制文件”。

我们通常用到的有 blocking、encoding 以及 translation 模式设置。encoding 模式我们放到后面的“高级应用”去详细论述。请看下面的例子：

```
% set fd [open c:/a.txt r+]      #打开文件
file8fd5c0
% fconfigure $fd
-blocking 1 -buffering full -buffersize 4096 -encoding cp936 -eofchar { {}
-translation {auto crlf}
% foreach {mode val} [fconfigure $fd] {puts "$mode = $val"} #所有属性
-blocking = 1
-buffering = full
-buffersize = 4096
-encoding = cp936
```

```
-eofchar = {}  
-translation = auto crlf  
% fconfigure $fd -translation      #查询-translation 属性  
auto crlf  
% fconfigure $fd -encoding binary  #设置 encoding 属性  
% fconfigure $fd -encoding        #查询 encoding 属性  
binary
```

我们看看标准的 stdout 通道的模式：

```
% foreach {mode val} [fconfigure stdout] {puts "$mode = $val"}  
-blocking = 1  
-buffering = line  
-buffersize = 4096  
-encoding = cp936  
-eofchar =  
-translation = crlf
```

我这里的系统是 Windows XP，可以看到这里标准输出的编码为 cp936，这也是我们打开其它文件的时候，默认的编码方式。-translation 模式为 crlf，也就是我们所说的“文本文件”。

## fcopy，通道间复制

fcopy 命令能够直接在两个通道之间复制数据，也就是说，它可以直接从一个通道读取数据，然后写入到另外一个通道中去。其格式如下：

```
fcopy inChannel outChannel ?-size Size? ?-command callbackCmd?
```

参数 inChannel 和 outChannel 分别表示源通道和宿通道；当从 inChannel 中读取了 Size 个字节或者达到 inChannel 结束的时候，复制过程结束。

当指定选项-command 之后，fcopy 命令会立刻返回，让拷贝过程在后台进行，拷贝结束之后会执行回调过程 callbackCmd。前提条件是，解释器必须通过 vwait 等进入事件驱动模式。

使用 fcopy 命令，我们可以简单的实现文件编码方式的转换，或者换行方式的转换。例如下面的脚本，可以将 Windows 系统下的文本文件中的 crlf 换行，直接转换为 Unix 下的 lf 换行：

```
#=====
#File      : fcopytest.tcl
#Author    : LeiYuhou
```

```

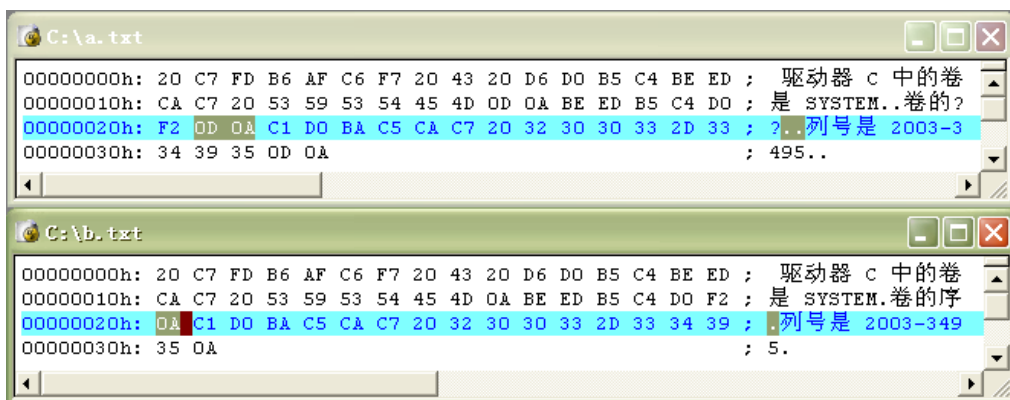
set fpath "c:/a.txt"
set opath "c:/b.txt"

set ifd [open $fpath "r"]
set ofd [open $opath "w"]
fconfigure $ifd -translation crlf
fconfigure $ofd -translation lf
fcopy $ifd $ofd

close $ifd ; close $ofd

```

这是在 UltraEdit 编辑器中使用十六进制模式打开 a.txt 和转换后得到的 b.txt 之后看到的情况：



可以清楚的看到，a.txt 中的所有\x0D\x0A 都被转换成为了\x0A。

使用 fcopy，还可以简单的实现编码转换，例如将 cp936 编码的文本文件转换为内容相同但是使用 UTF-8 编码的文本文件，请看例子：

```

#=====
#File      : fcopytest.tcl
#Author    : LeiYuhou

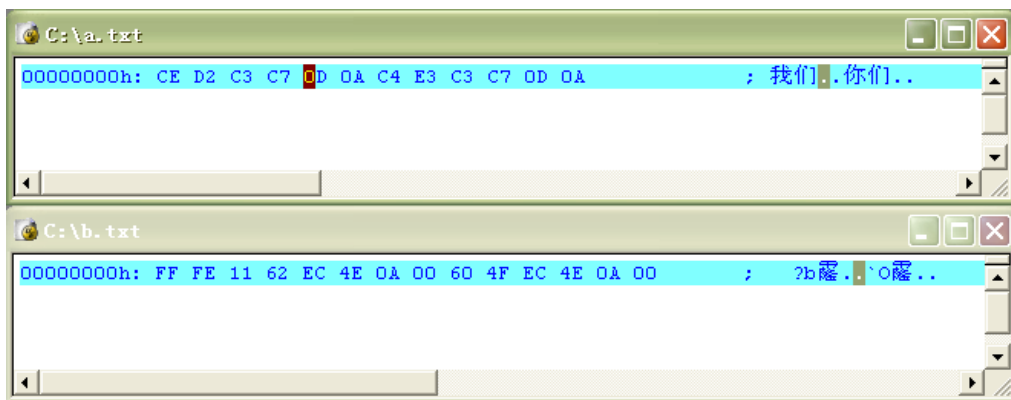
set fpath "c:/a.txt"
set opath "c:/b.txt"

set ifd [open $fpath "r"]
set ofd [open $opath "w"]
fconfigure $ifd -translation crlf -encoding cp936
fconfigure $ofd -translation lf -encoding utf-8
fcopy $ifd $ofd

```

```
close $fd  
close $ofd
```

执行结果如下，这是采用 UltraEdit 的十六进制模式打开两个文件后的窗口：



上面程序运行环境是中文 Windows XP SP2，a.txt 是一个采用 cp936 编码的文本文件。我们打开它的时候，将其配置成 `-encoding = cp936` 以及 `-translation=crlf`；然后打开输出文件 b.txt，并且配置成 `-encoding=utf-8`，`-translation=lf`。fcopy 执行之后，b.txt 的内容就成了采用 UTF-8 编码的内容相同的文件。

可以看到：在 cp936 编码中，字符“我”用 0xCE,0xD2 这两个字节来表示；但是使用 UTF-8 编码的时候，字符“我”用两个字节 0x11,0x62 来表示。可见：同样的一个字符，在不同的编码方案中，是用不同的编号来表示的。

## 通道句柄上的事件驱动

首先要强调的一点是，TCL 中的通道是一个非常广泛的概念，文件系统中的文件被打开之后的文件句柄只是通道的一种，除此之外，还有进行网络通信的 Socket 句柄、打开的串口句柄、匿名管道、以及内存通道等。不过因为 TCL 脚本良好的设计和封装，在脚本中读写这些通道的时候，我们根本不用关注这些通道的类型。

话虽然如此，但是这些通道之间也存在一定差别，比如文件系统中的文件句柄，读写起来速度就非常快，但是网络 socket，操作起来可能就比较慢。如果我们使用阻塞式的通道操作，那么应用程序可能会使用不少时间来等到读写操作完成，而用户界面的操作将被阻塞。例如在阻塞式的通道上调用 `gets` 或者 `read` 的时候，如果恰好缓冲区中没有足够的数据，那么这两个命令就会阻塞直到通道中收到了足够的数据。

非阻塞的通道上调用 `gets` 或者 `read` 从来不会阻塞。但是我们应该在什么时候去读写呢？如果应用程序只在有数据可读、或者通道可写的时候，才去读写通道，那将节约不少时间。实际上，TCL 已经为我们实现了这个功能，这就是通道事件。当通道上可以读，或者可以写

的时候，TCL 自动的触发事件，来执行我们设定的一段脚本，这样就实现了事件驱动。

TCL 中在通道上实现事件驱动的命令是 `fileevent`，语法如下：

```
fileevent channelId readable ?script?
```

```
fileevent channelId writable ?script?
```

这两个命令分别为参数 `channelId` 所指定的通道来设置事件响应脚本：

- 1) 如果给出了参数 `script`，那么就会为通道 `channelId` 设定对应的响应脚本；
- 2) 如果原来已经设定过了对应的脚本，那么新的 `script` 回代替原来的脚本；在某一时刻，通道的一个具体事件只能有一个对应的脚本；
- 3) 如果 `script` 是一个空字符串，那么将删除掉原来对应的脚本；
- 4) 如果没有给出参数 `script`，那么将返回本通道上已经设定的事件响应句柄；如果原来没有设定，那么返回空字符串；

TCL 支持两类事件，`readable` 表示通道可读了，`writable` 表示可写了。如下情况通道被认为是可读：

- 1) 在输入缓冲区中存在还没有读出的数据，那么会触发 `readable` 事件；
- 2) 如果最近一次读操作是 `gets` 但是没有读出一个完整行（没有发现换行符），那么即使缓冲区存在数据，那么也不会触发 `readable`。这是第一种情况的例外；
- 3) 到达了文件末尾；
- 4) 设备或者文件出错；

如下情况则被认为是可写，会触发 `writable` 事件：

- 1) 至少可以向文件中写一个字节；
- 2) 文件句柄没有阻塞；
- 3) 底层文件或者设备出错；

使用通道事件的时候，最好把通道设置在非阻塞模式下，这可以使用 `fconfigure` 命令做到。下面我们来看一个 Tk 的例子，例子中我们实现一个图形化的 `tracert` 工具，其内部使用了操作系统的命令 `tracert.exe`。

`tracert` 是用来干什么的？这个话题与我们的 TCL 无关，不过还是顺便介绍一下，这个命令是用来追踪达到目的主机的路由路径的。具体可以参考 TCP/IP 相关书籍。

请看代码：

```
# -----  
# Filename   : TraceRt.tcl  
# Author    : LeiYuhou  
# Create date: 2005-8-6  
# -----  
package require Iwidgets 4.0  
  
option add *textBackground seashell  
. configure -background white
```

```
iwidgets::Labeledframe .pr -labelpos nw -labeltext "TraceRt Options:"
set cs [.pr childsitem]

#创建输入框
iwidgets::entryfield $cs.target -labeltext "Target:" -labelpos w \
    -command { focus [.passwd component entry] } \
    -textvariable ::targetAddress

#创建按钮
iwidgets::buttonbox $cs.bb -pady 1
$cs.bb add Start -text "Start trace" -command "OnStart" -pady 1
$cs.bb add Stop -text "Stop" -command "OnStop" -pady 1
$cs.bb default Start

pack $cs.target -side left -fill x -expand 1
pack $cs.bb -side right
pack .pr -side top -fill x -expand 1 -anchor nw

#创建记录结果的文本框
iwidgets::scrolledtext .st -labeltext "TraceRoute result" -wrap none \
    -vscrollmode static -hscrollmode dynamic \
    -width 5i -height 2i -relief sunken

pack .st -side top -fill both -expand 1 -anchor n

set hPipe "" ;#匿名管道的句柄
set targetAddress "www.szptt.net.cn" ;#要追踪的目标地址

#按钮 Start 的响应函数
proc OnStart {} {
    if {$::hPipe!=""} { return }

    set cmdline "|tracert.exe $::targetAddress"
    #启动 tracert 子进程，创建匿名管道
    set ::hPipe [open $cmdline r]

    #将管道设置为非阻塞模式
```

```
fconfigure $::hPipe -blocking 0 -translation binary

#设置管道句柄的读事件响应脚本，执行过程 OnRead $::hPipe
fileevent $::hPipe readable "OnRead $::hPipe"
}

#按钮 Stop 的响应函数
proc OnStop {} {
    if { $::hPipe==""} { return }

    close $::hPipe
    set ::hPipe ""
}

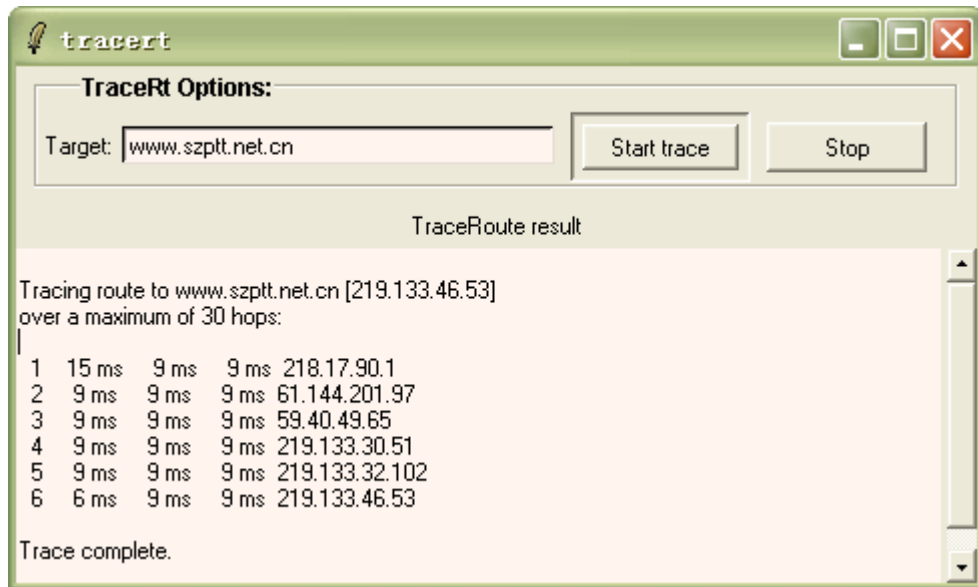
#本过程就是通道事件 readable 的响应函数。
proc OnRead {fpipe} {
    puts "readable Event.."
    if {[eof $fpipe]} { ;#判断是否文件结束，如果结束，则执行这里
        close $::hPipe ;#关闭管道，并且返回
        set ::hPipe ""
        puts "eof."
        return
    }

    set v [gets $fpipe buf] ;#从管道中读取一行

    if {$v != -1} {
        .st insert "end - 1 chars" "$buf\n" ;#记录结果
        puts "$v bytes data read."
    } else {
        puts "read nothing."
    }
}
```

在 Windows XP 操作系统上，执行 `tclsh.exe tracert.tcl`。运行的界面如下：





上面的过程 `OnRead` 代码中，我们使用 `puts` 来记录通道事件产生的情况。只要管道中有数据，那么 `TCL` 都会自动调动 `OnRead` 过程，`OnRead` 中首先判断文件是否到达末尾，然后读取一行，如果读到了完整行那么就写入到用户界面中。

下面是 `puts` 输出的部分日志：

```
46 bytes data read.
readable Event..
read nothing.
readable Event..
read nothing.
readable Event..
read nothing.
readable Event..
read nothing.
readable Event..
48 bytes data read.
```

可以看到，当通道中产生 `readable` 事件的时候，我们调用 `gets` 命令并不一定能够从通道中读取到一个完整的行。所以在代码中，我们使用了：

```
set v [gets $fpipe buf]
```

这样的读取操作方式。在通道处于 `nonblocking` 模式下，并且 `gets` 无法读到一个完整行或者通道被关闭的时候，`gets` 会返回-1。

上面程序就演示了一个典型的事件驱动程序。因为 `tracert` 在追踪路由路径的时候，需要耗费很长的执行时间。为了能够及时的在 `GUI` 上反映追踪的进度，同时又避免用户界面被

冻结，我们这里正好用上通道事件。代码中用到了 Tk 和 IWidget，这是用来创建 GUI 程序的程序库，我们后面会介绍。

## 匿名管道

在 Unix 系统的 Shell 编程中，管道是非常有用的一种机制，它可以将一个命令的输出结果作为另外命令的输入，从而将多个命令有机的结合起来。TCL 中可以非常方便地通过文件操作来创建匿名管道，从而实现 IPC（进程间通信）。

怎样创建匿名管道？很简单，使用 open 命令来打开需要执行的子进程可执行文件。和打开普通文件不同之处在于，文件名的最前面必须使用管道符号“|”开始，表示需要执行这个文件，并且返回匿名管道句柄。

下面的脚本实现了一个过程，来判断目的主机是否可达。这里通过执行 ping.exe 文件并且读取匿名管道来实现，请看代码：

```
#-----
#File Name   : ping.tcl
#Author      : LeiYuhou

#Proc name   : HostAvailable
#说明        : 返回目的 host 的可达性，返回介于 0—100 之间。
#参数 host   : 目的主机的 IP 地址
proc HostAvailable {host} {
    #执行 ping.exe，创建子进程，打开匿名管道
    set fd [open "|ping.exe $host" r]
    set result 0
    while {[eof $fd]==0} {
        set buf [gets $fd]    ;#读取管道内容

        #使用正则表达式进行解析
        if {[regexp {, Lost = \d+ \((\d+)% loss\),} $buf _t p]} {
            set result $p
        }
    }
    close $fd
    return [expr 100-$p]    ;#返回目的可达性
}

#测试本机 127.0.0.1 的可达性
```

```
set percent [HostAvailable 127.0.0.1]
puts "Host 127.0.0.1  -> $percent"
```

上面代码中，过程 `HostAvailable` 返回目的主机 `host` 的可达性，返回 0 表示完全不可达，100 表示完全可达。实际上就是 `ping.exe` 发送的多个 ICMP 包的回复率。在过程中使用 `open` 来执行了 `ping.exe`，并且通过匿名管道读取其输出内容，然后使用正则表达式解析出其丢包率。

在我的机器上，上面代码执行结果如下：

```
Host 127.0.0.1  -> 100
```

再次重复一遍，`open` 的第一个参数表示需要打开的文件名，如果该参数的第一个字符是管道字符“|”，那就表示执行后面的可执行文件（而不是简单打开它），并且创建当前进程和这个子进程之间的匿名管道，父子进程之间可以如下通信：

1. 子进程输出到标准输出的所有内容，都被写入到这个管道中。父进程都可以通过读取该管道来获得子进程的输出。
2. 父进程可以向管道写入数据，这些数据都会作为子进程的标准输入。

当然了，读写数据的时候我们可以使用前面介绍的 `fileevent` 来设置事件响应脚本，进行事件驱动的编程。

# TCL 高级进阶

应该说，掌握第二章介绍的 TCL 基础之后，我们能够完成大部分的编程任务，解决平常工作中遇到的常见问题了；也许也能够给周围同事一些答疑解惑了。在享受这些成就感的时候，我还是要提醒一下各位，TCL 是博大精深的，第二章介绍仅仅的是 TCL 的核心基础，离 TCL 高手需要达到的境界还相差很远。

这一章，我们将探索 TCL 各种高级应用，看看在各种应用场合，TCL 是如何解决复杂问题的。我们将分别介绍如下激动人心的内容：

1. **TclX**: TclX 是一个简单实用的扩展包，它提供的命令可以扩展和增强 TCL 的部分功能。例如对列表的扩展等。
2. **C 语言扩展和嵌入**：我们可以使用 C/C++ 语言来为 TCL 编写扩展包，也可以把 TCL 解释器嵌入到我们的程序中，使我们的程序可以象 Emacs, VIM 或者 Visual Studio IDE 一样通过脚本语言来定制和扩展程序的功能；也就是说，你不用重新编译整个系统，而只需要更改或者增加新的脚本即可。
3. **高级数据结构**：这一部分我们将介绍如何使用 TCL 来表达和操作树（tree）、图（graph）、列表（list）、集合（set）、矩阵（matrix）、栈（stack）、记录（record）、队列（queue）、优先级队列（prioqueue），跳跃式列表（skiplist）。

呵呵，这些术语的出现，是不是勾起了你对大学课堂的丝丝回忆。

4. **TCL 与 COM**：一门脚本语言要想在 Windows 的世界里生存下去，对 COM 技术的支持是必不可少的；否则就会败在 VBScript 和 JScript 手下。通过使用支撑包 tcom 之后，可以用 TCL 脚本作为客户端来控制自动化服务器；甚至可以来实现 COM 组件！

脚本实现 COM 组件？你没有发烧吧？喔没有，这是真的可以的。

5. **数据库**：介绍如何使用 TCL 脚本来操作数据库，包括创建数据库，查询数据等。我们将分别研究如何在 TCL 中使用 TclOdbc 以及 ADO 组件这两种方式，来操作数据库。
6. **虚拟文件系统**：TCL 中的文件系统都是虚拟的，包括本机的硬盘文件系统。除此之外，ZIP 压缩包，FTP 服务器，HTTP 服务器等都可以被虚拟成一个文件系统，从而可以通过常见的文件操作命令来操作它们，比如 open, gets 等。

是吗？那我是不是可以直接用 open 来打开一个 HTTP 服务器上的页面文件？真不可思议！没错，完全可以这样做，只是 open 之前我们需要进行一些小小的设置。

7. **多线程编程**：一个进程中可以存在多个线程，这样就有了多条执行路径，它们之间可以并行执行，通过通过同步对象（比如信号量）来同步。同时，同属一个进程的所有线程可以共享地址空间和全局变量。TCL 也支持多线程的编程，不过需

要我们做一些小小的工作：自己编译自己的 TCL 解释器。

8. **Socket 编程：**Socket 是传输层向应用层提供的编程接口。通过它，我们能够实现 TCP 服务器以及 TCP 客户端，从而实现 TCP/IP 应用层的网络编程。Socket 是一套复杂的接口，但是 TCL 却进行极大的简化，使用起来非常方便。
  9. **Ftp、Http 和 SNMP 等网络方面的编程：**Socket 编程属于比较底层的开发，而这一节我们介绍如何通过 Scotty、以及其它扩展包，来实现网络应用方面的编程。Scotty 实现了一系列应用层的协议，我们使用起来更加方便。您会吃惊的发现，使用 TCL 实现一个 web 客户端或者 FTP 客户端，竟然是如此的简单！
  10. **Tk 和 GUI 编程：**这里我们将介绍一下如何使用 Tk 来进行图形用户界面的开发。关于 Tk 本身可以写成一本书，我们这里只作简单的介绍。
- OK，让我们开始吧！

## 目录

TCL 高级进阶 .....	1
使用 TclX .....	7
一般命令 .....	7
目录栈 .....	7
增强的循环控制 .....	8
递归文件搜索 .....	9
异常控制 try_eval...catch...finally.....	10
list 扩展命令 .....	11
列表模拟集合 .....	12
添加和删除等 .....	12
lmatch 进行查找 .....	14
lassign 分解列表 .....	14
索引列表 (keyed list) .....	15
命令介绍 .....	16
和 array 性能比较 .....	17
文件操作命令 .....	18
lgets 读取列表 .....	19
read_file .....	20
for_file .....	20
文件扫描 .....	20
使用 tcltest 和 TclUnit 进行测试.....	24
tcltest .....	24
一个简单的用例 .....	24
匹配模式、准备和清除 .....	25
检查输出信息 .....	26
检查异常以及特殊返回码 .....	27
组织运行多个测试用例 .....	29
tclUnit .....	32
下载安装 tclUnit .....	32
tclUnit 用例入门 .....	33
增加一个用例 .....	35

写一个简单的测试构件 .....	36
使用断言 .....	39
动手扩充 tclUnit 的断言 .....	40
C 语言扩展和嵌入 .....	43
用 C 编写扩展命令 .....	43
load 命令 .....	43
创建 DLL 工程 .....	46
实现扩展命令函数 .....	46
库初始化函数 .....	52
安装库并且测试 .....	52
编写自己的数学函数 .....	54
实现为 package .....	57
摆脱 TCL 版本的限制 .....	59
解释器嵌入 .....	61
初始化 TCL .....	61
初始化 TCL 解释器 .....	65
嵌入解释器的用途 .....	73
高级数据结构 .....	74
列表 (list) .....	74
普通命令 .....	74
高阶函数 .....	76
离散对象池 (discrete items pool) .....	79
队列 (queue) .....	82
优先级队列 (prioQueue) .....	84
跳越式列表 (skipList) .....	85
栈 (stack) .....	87
集合 (set) .....	88
矩阵 (matrix) .....	90
创建矩阵对象 .....	90
其它矩阵操作 .....	92
关联数组 .....	94

树 (tree) .....	95
创建和销毁 .....	96
节点操作和查询命令 .....	96
操作节点属性 .....	99
树的遍历 .....	100
图 (graph) .....	103
构造一个图 .....	103
操作节点和边 .....	107
图的序列化 .....	108
图的遍历 .....	109
最短路径算法 .....	112
TCL 与 COM 组件 .....	115
COM 自动化简介 .....	115
使用 TCL 来操作 Excel .....	117
在 TCL 中使用组件举例 .....	118
关闭 Windows 系统 .....	119
列出当前所有进程以及相关信息 .....	119
Kill 某些进程 .....	121
改变磁盘卷标 .....	122
查询 CPU 信息 .....	123
tcom 详细使用方法 .....	125
了解和解析接口 .....	125
创建服务器对象 .....	133
调用方法和属性 .....	134
响应事件 .....	136
查看接口信息 .....	137
用 TCL 编写 COM 组件 .....	138
编写 IDL 文件 .....	138
生成 TLB 文件 .....	139
创建 TCL 程序包 .....	140
注册服务器 .....	144
在客户端 (JScript、Tcl 或者 C++) 中使用 .....	144
IActiveScript 接口 .....	148



JScript 与 VBScript 混合编程 .....	148
TCL for ActiveScript .....	149
通过 JScript/VBScript 来扩展 TCL.....	150
数据库编程 .....	155
TCL+Odbc.....	155
操作数据源 .....	155
数据库对象接口 .....	159
命令对象接口 .....	163
TCL+ADO .....	165
TCL+BerkeleyDB.....	167
下载和安装 .....	168
使用介绍 .....	169
TCL+SQLite .....	171
TCL+MySQL.....	171
TCL+XML .....	171
TCL+文本文件 .....	171
虚拟文件系统 .....	173
多线程 .....	175
Socket 编程 .....	175
Ftp、Http、SNMP 等网络编程.....	175
Tk 和 GUI .....	176

## 使用 TclX

TclX 的完整名字应该为“Extended TCL”，它在不改变标准 TCL 语法的前提下，增加了一系列的命令，从而提供了一些增强功能。TclX 可以看成是标准 TCL 的超集。在代码中灵活的使用 TclX，能够非常方便的解决一些比较繁琐的问题。我们选择最有意义的部分进行介绍，其它命令可以参考 TclX 的帮助。

要使用 TclX 中的命令，首先必须引入这个程序库，如下：

```
package require Tclx
```

## 一般命令

这一部分 TclX 命令包括目录栈、特定循环命令、递归文件搜索和异常控制这几个部分。我们首先从目录栈开始。

## 目录栈

大家实际编程的时候，肯定碰到过需要在几个目录之间来回切换的情况。特别是类似下面的操作，会经常碰到：

1. 首先用一个变量保存当前目录名字；
2. 切换到另外的一个目录，进行一些其它操作；
3. 切换回来原来的目录；

TclX 通过“目录栈”来简化这样的操作。在 TclX 内部实现了一个栈对象，栈里面的元素是目录名字。针对目录栈的操作命令有：dirs, popd, pushd 三种，其格式和用法如下。

1. dirs: 显示目录栈的内容，返回一个列表，其第 0 个元素是当前目录，后面则是栈的内容；元素序号越小则越靠近栈顶。
2. pushd dirname: 将当前目录压入栈，并且将 dirname 设置为当前目录；
3. popd: 将栈顶元素弹出栈，并且将其设置为当前目录；

提到 TCL 的目录操作，顺便介绍 TCL 的两个核心命令：cd 和 pwd。前者用来更改当前目录，后者用来查询当前目录。我们结合目录栈来一起看看下面的例子：

```
% pwd          ;#查询当前目录，为 D:/  
D:/  
% dirs         ;#查询当前“目录栈”的内容，返回一个 D:/，就是当前目录  
D:/  
% pushd c:/ ; pwd ;#更改当前目录为 C:/，pwd 返回 C:/，可见 pushd 确实更改了
```

```
C:/
% dirs          ;#查询“目录栈”的内容，里面有两个元素
C:/ D:/
% push e:/ ; pwd ;#更改当前目录为 e:/
E:/
% dirs          ;#显示“目录栈”的内容，有两个个元素，当前为 E:/
E:/ C:/ D:/
% cd c:/ ; dirs ;#用 cd 直接更改当前目录为 C:/，然后看栈内容
C:/ C:/ D:/
% popd ; dirs   ;#弹出栈顶元素。栈内只剩下一个元素 D:/
C:/ D:/
% popd ; dirs   ;#再次弹出栈，栈已经为空了。当前目录则为 D:/
D:/
% pwd
D:/
% popd          ;#使用 popd 再次弹出栈顶元素，会出错！
directory stack empty
```

要记住的是 `dirs` 返回的是：当前目录，栈内容。所以即使栈为空，`dirs` 也会返回一个元素，就是当前目录的名字。

## 增强的循环控制

TCL 的一个特点就是“一切都是命令和其参数，并且可以很方便的扩展”，这一点我们以前强调了很多次。如果你觉得 TCL 提供的循环控制不好用，可以自己编写循环控制命令，第二章中我们就自己写过一个 `do` 循环。TclX 为我们提供了几个循环命令：

1. `for_array_keys`：循环遍历数组元素；
2. `loop`：开始、结束和步长都可控的循环；类似 VB 的 `FOR i=s to n STEP m` 我们逐一介绍。

如果给你一个 TCL 数组变量 `students`，要求遍历数组中的每一个元素，你会如何操作？一般情况下我们会写出如下的代码：

```
array set students {1 Tom 2 Jack 3 Mike}
foreach id [array names students] {
    puts "$id -> $students($id)"
}
```

TclX 为我们提供了另外一个 `for_array_keys` 命令，来专门遍历数组。其语法如下：

```
for_array_keys var array_name code
```

例如刚才的代码可以如下改写：

```
for_array_keys id students {  
    puts "$id -> $students($id)"  
}
```

break 和 continue 命令都能够在 for\_array\_keys 中正常使用。

我们再看另外的一个问题：在 VBScript 中 for 循环可以这样写：

```
For counter = start To end [Step step]  
    [statements]  
Next
```

也就是说，可以指定循环的开始值，结束值和递增步长。这没有什么希罕的，TCL 中的 for 循环也行；不错，可以是可以，但是不够直观。TclX 提供了一个 loop 命令，它比 for 命令更加直观，同时更加高效，语法如下：

```
loop var start end ?step? body
```

其中 var 是一个循环变量的名字；start, end 和 step 必须都是整数，step 可以是负数，可以省略，如果省略那么就默认为 1。body 则是需要执行的循环体。请看例子：

```
package require Tclx  
loop i 0 4 2 { puts "$i" }    #这里只会输出 0, 2 两个数字
```

要注意的是：start 等数值只在 loop 循环之前计算一次；并且当 var>=end 的时候会马上退出循环。所以计算 1+2+..100 的时候必须写成：

```
set sum 0  
loop i 1 101 1 { incr sum $i }
```

这里的 end 为 101 而不是 100。

## 递归文件搜索

loop 循环先介绍到这里，我们再考虑另外一个常见问题：如果我们需要在一系列目录下（包括所有子目录）搜索那些匹配特定模式的文件，并且针对这些文件做一些操作，那么该当如何？大家首先想到的肯定是 glob 命令进行搜索。什么？你没有想到？嗯，这个……你应该想到的。如果没有想到 glob，那么先去看看第二章介绍的 glob。

用 glob 没错，但是如何递归搜索所有子目录？呵呵，是不是想起了什么“深度优先”和“广度优先”遍历这样的术语？可能还需要去翻一下《数据结构》……，其实用不着，TclX 为我们提供了命令 for\_recursive\_glob，其语法如下：

```
for_recursive_glob var dirlist globlist code
```

其中 `var` 是一个循环变量, `dirlist` 是需要搜索的目录列表, `globlist` 则是需要匹配的模板列表, `code` 是需要执行的循环体。请看例子:

```
for_recursive_glob fn {e:/Work e:/game} {*.tcl *.txt} {  
    puts $fn  
}
```

这个例子搜索 `E:/work` 和 `e:/game` 这两个目录下(包括所有子目录)下的`*.tcl` 和`*.txt` 文件, 并且打印出搜索到的完整文件名。循环变量中包含每次搜索到的文件路径名。

要提醒一下的是, 这里 `for_recursive_glob` 采用的是广度优先的遍历算法。

## 异常控制 `try_eval...catch...finally`

TCL 核心命令 `catch` 可以完成异常捕获, 但是用起来不是很直观, 特别是和 C++ 的异常处理语句 `try...catch...finally` 相对比, 部分 C++ 程序员对 `catch` 命令就非常不满, 现在好了, TclX 提供的 `try_eval` 正好满足这部分高手的需求, 而且用起来非常直观。其语法如下:

```
try_eval code catch ?finally?
```

其中参数 `code` 是可能引发异常的代码; `catch` 是用来处理异常的代码; `finally` 则是最终执行代码, 它是可选参数, 如果给出了 `finally` 参数, 那么不管 `code` 有没有引发异常, `finally` 都会被执行。请看例子:

```
proc try_eval_test {} {  
    set m [try_eval {      ;#Code 部分  
        set a 100  
        set b 200  
        set c [expr $a+$b +]      ;#多了一个加号, 引发异常  
    } {  
        puts "catch: $errorResult"      ;#Catch 部分  
    } {  
        puts "finally."      ;#Finally 部分  
    }]  
  
    puts "result: $m"      ;#打印出命令 try_eval 的结果  
}  
try_eval_test
```

执行结果如下:

```
catch: syntax error in expression "100+200 +": premature end of expression
finally.
result:
```

如果 Code 部分执行异常，那么全局变量 `errorResult`、`errorCode` 和 `errorInfo` 中会记录相关错误信息。这些变量可以直接在 `catch` 中使用。

## list 扩展命令

列表是 TCL 中一种非常重要的数据结构，第一部分我们已经做了详细的介绍。这里我们来总结一下它的特点：

1. 它是元素的有序集合，每一个元素都可以通过唯一序号来指定；
2. 长度没有限制，可以无限多（只受系统资源的限制）；
3. 元素类型不做限定，可以是嵌套的列表；

大家如果用过 VBScript，可能会觉得 TCL 列表和 VBScript 中的数组比较类似。事实上两者存在较大的差异，最明显的莫过于上面列出的第 2 点。VBScript 的数组虽然也可以通过 `ReDim` 来改变大小，但是用起来远远不如 TCL 列表方便，而且 VBScript 数组中不能插入元素。所以当我用熟了 TCL 之后再用 VBScript，就觉得有些数据结构要表达出来，非常不方便。

但是，TCL 的列表在使用上也有其一些不方便之处，例如我们向列表中插入一个元素的时候，需要如下调用 `linsert` 命令：

```
set a {100 200 300}
set a [linsert $a 0 "Insert Element"] ;#linsert 后必须重新赋值给 a
```

可以看到，插入元素不直观；同样，删除元素也不直观！为了解决这些问题，同时增强列表的功能，TclX 提供了一系列命令，见如下表格：

命令	功能说明
<code>intersect lista listb</code>	两个列表作为两个集合，返回它们的交集
<code>intersect3 lista listb</code>	两个列表作为集合，返回三个集合，分别为 <code>a-b</code> ， <code>a^b</code> ， <code>b-a</code>
<code>union lista listb</code>	返回两个集合的并集
<code>lcontain list element</code>	判断 <code>element</code> 是否存在于列表 <code>list</code> 中
<code>lempty list</code>	判断列表 <code>list</code> 是否是空
<code>lmatch</code>	返回列表中所有能够匹配特定模式的元素
<code>lrmdups list</code>	删除列表 <code>list</code> 中重复的元素
<code>lvarcat</code>	在列表的后面增加元素
<code>lvarpop</code>	删除列表中的某一个元素
<code>lvarpush</code>	向列表中插入元素
<code>lassign list var ?var?</code>	将列表元素的值分别赋值给后面的变量

## 列表模拟集合

通过 TclX，能够把列表模拟成集合来使用。所谓集合，是一系列不重复元素的无序的组合。虽然列表元素是有顺序的，但是我们可以忽略这种特性。请看示例：

```
% package require Tclx
8.4
% set a {Tom Mike Bush Jack}      ;# a 是一个集合
Tom Mike Bush Jack
% set b {Tom Tiger Rose}          ;# b 是另外一个集合
Tom Tiger Rose
% intersect $a $b                  ;#求两个集合的交集
Tom
% foreach s [intersect3 $a $b] {puts $s}    ;#换一种方式求交集，返回列表
Bush Jack Mike                    ;#第一个元素是 a-b
Tom                                ;#第二个元素是 a 和 b 的交集
Rose Tiger                        ;#第三个元素是 b-a
% lcontain $a Tom                  ;#判断 Tom 是否包含在列表 a 中
1
% lcontain $a tom                  ;#判断 tom 是否包含在列表 a 中
0
% lempty $a                        ;#判断列表 a 是否为空
0
% eval lvarcat a $a ; puts $a      ;#将 a 的所有元素添加在 a 的末尾
Tom Mike Bush Jack Tom Mike Bush Jack
% puts [lrmdups $a]                ;#删除列表 a 中重复元素后的结果
Bush Jack Mike Tom
% union $a $b                      ;#返回列表 a 和 b 的并集，结果是排序过后的列表
Bush Jack Mike Rose Tiger Tom
```

可以看到，这些命令把列表当成集合进行运算操作的时候，元素会被进行排序，并且重复的元素会被自动的删除掉。所以我们不能对结果中元素的顺序做出任何假设。

## 添加和删除等

TclX 提供的 lvarpush、lvarpop 和 lvarcat 命令能非常方便的向列表中插入和删除元素：

1. lvarpush var string ?indexExpr?

本命令将参数 `string` 所插入到列表 `var` 的指定位置 `indexExpr`。如果 `indexExpr` 省略，那么就插入到列表头，也就是默认为 0 的位置。

2. `lvarpop var ?indexExpr? ?string?`

本命令从列表 `var` 中位置为 `indexExpr` 的元素删除掉，如果给出了参数 `string`，那么就用 `string` 代替该元素；如果没有给出 `indexExpr` 参数，那么就默认为 0。

3. `lvarcat var string ?string ...?`

将所有的参数 `string` 作为列表元素添加到列表变量 `var` 后面。如果 `var` 不存在，那么就创建它。

请看下面的例子：

```
% lvarcat m Tom Mike      ;#变量 m 不存在，lvarcat 会创建它
Tom Mike
% lvarcat m Jack Tiger    ;#在 m 后面添加两个元素
Tom Mike Jack Tiger
% puts "[lvarpop m] ; $m"  ;#删除列表 m 的首元素
Tom ; Mike Jack Tiger
% lvarpop m 0 Bush        ;#用 Bush 替换首个元素，返回原来的元素 Mike
Mike
% puts $m                 ;#列表 m 的首个元素变为 Bush
Bush Jack Tiger
% lvarpop m 1 Tom         ;#用 Tom 替换 m 第一个元素，返回原来元素
Jack
% puts $m                 ;#第一个元素变为 Tom
Bush Tom Tiger
% lvarpush m Mike ; puts $m ;#将 Mick 插入到列表头部
Mike Bush Tom Tiger
% lvarpush m Mike end;puts $m ;#将 Mike 插入到列表 end 处
Mike Bush Tom Mike Tiger
% lvarpush m Mike len ; puts $m ;#将 Mike 插入到列表 len 处
Mike Bush Tom Mick Tiger Mike
% lvarpush m Mike len-1 ; puts $m ;#将 Mike 插入到列表的 len-1 处
Mike Bush Tom Mick Tiger Mike Mike
```

注意这里参数 `var` 是列表变量的名字，而不是列表变量的值。如下的代码会出错：

```
% lvarpop $m      ;#这里使用 $m 而不是 m，抛出异常！
can't read "Mike Bush Tom Mick Tiger Mike Mike": no such variable
```

命令参数 `indexExpr` 中可以是一个表达式，`lvarpop` 和 `lvarpush` 会自动计算它。表达式可以包含特定标志 `len` 和 `end`，其中 `len` 会被替换成列表的长度，`end` 会被替换成最后一个



元素的下标，等于“len-1”。请看例子：

```
% lvarcat m 100 200 300 400
100 200 300 400
% lvarpop m "2-1" 200000 ;#index 是表达式，用 200000 替换掉位置 1 的元素
200
% set m ;#替换后的 m
100 200000 300 400
% lvarpop m end ; set m ;#删除 end 位置的元素，可见最后的 400 被删除了
100 200000 300
% lvarpush m 400 len ; set m ;#将 400 插入到 len 位置，就是添加到最后
100 200000 300 400
% lvarpush m 350 end ; set m ;#将 350 插入到 end 位置，即最后元素的前面
100 200000 300 350 400
% lvarpush m 360 len-1 ; set m ;#将 400 插入到 len-1 位置，即 end 位置
100 200000 300 350 360 400
```

## lmatch 进行查找

lmatch 能在列表中搜索匹配特定模式的元素，它是 lsearch 的简化版。该命令格式如下：

lmatch ?mode? list pattern

其中 mode 可以为 -exact, -glob, -regexp, 分别表示精确匹配, glob 匹配和正则表达式匹配，默认是 glob 模式。list 是需要查找的列表，pattern 则是需要查找的模式。请看例子：

```
% set files [glob *] ;#当前目录下所有文件
% lmatch -glob $files tcl*.h ;#搜索 tcl*.h
tcl.h tclDecls.h tcldom-libxml2.h
% lmatch -exact $files tcl.h ;#查找 tcl.h
tcl.h
% lsearch -glob -all -inline $files tcl*.h ;#使用 lsearch 命令
tcl.h tclDecls.h tcldom-libxml2.h
```

可以看到，命令 lmatch -glob... 与 TCL 核心命令 lsearch -glob -all -inline... 完全等价。

## lassign 分解列表

首先看个问题：一个列表包含 n 个元素，如何把这 n 个元素值设置到 n 个变量中？这是我们编程时候经常碰到的一个问题。该题目太简单，部分大侠马上就能写出如下的代码：

```
lappend m {1 2 3}
set m1 [lindex $m 0]
set m2 [lindex $m 1]
set m3 [lindex $m 2]
```

没错，上面代码完成了我们要求的功能，但是不够直观。有高手写出如下的代码：

```
foreach {m1 m2 m3} $m {}
```

它可以完成同样的功能，将 `m` 中三个元素分别赋值给 `m1`，`m2` 和 `m3` 三个变量。不过 TclX 提供了 `lassign` 命令，它比 `foreach` 更加直观明了。格式如下：

`lassign list var ?var...?`

1. 其中参数 `list` 是列表，`var` 则是变量名字。它将 `list` 中各元素一次赋值给 `var` 列表
  2. 如果 `var` 参数个数小于 `list` 长度，那么命令会返回 `list` 中没有赋值的元素列表；
  3. 如果 `var` 参数大于 `list` 长度，那么只有前面的 `var` 变量被赋值，后面的变量为空；
- 例如：

```
% set m
100 200000 300 350 360 400
% lassign $m m1 m2      ;#前两个元素赋值给 m1 和 m2，返回剩下的列表
300 350 360 400
% puts "m1=$m1 , m2=$m2" ;#输出变量 m1 和 m2 的值
m1=100 , m2=200000
% lassign $m m1 m2 m3 m4 m5 m6 m7 m8 ;#变量个数超过列表长度
% puts "m7=$m7 , m8=$m8" ;#后面的 m7 和 m8 是空值。
m7= , m8=
```

如果熟悉 Python，可能会想了，这么简单的操作在 TCL 中如此麻烦？是啊，这个功能在 Python 中很简单就实现了，如下：

```
>>> a = [1,2,3]
>>> m1,m2,m3=a      #将 a 的元素分别赋值给 m1,m2,3 三个变量
>>> print m1,m2,m3
1 2 3
```

每一种语言都有其特色。这里就不进行 TCL 和 Python 的比较了，以免引起战争。

## 索引列表（keyed list）

索引列表是 TclX 引入的一种列表，其本质上也是一个普通的 TCL 列表。只不过在索引列表中，每一个元素包含两部分：索引（key）和值（value），它们两者组合成一个子列

表作为索引列表的一个元素，例如 `person` 是一个索引列表：

```
{ {NAME LeiYuhou} {EMAIL leiyuhou010@hotmail.com} }
```

它包含两个元素，索引分别为 `NAME` 和 `EMAIL`，后面是对应的值。可以看到，这种数据结构类似于 C 语言中的 `struct`，每一个元素对应于 `struct` 中的一个域（`field`）。

索引列表中的域可以嵌套。例如上面下面的 `person`：

```
{ {NAME { {FIRST Yuhou} {LAST Lei} } } {EMAIL leiyuhou010@hotmail.com} }
```

对应的嵌套域的索引为 `NAME.FIRST` 和 `NAME.LAST`，中间用 “.” 分隔。域嵌套的深度没有限制。

## 命令介绍

TclX 提供了如下的命令来操作索引列表：

1. `keylset listvar key value ?key2 value2...?`

用来设置索引列表 `listvar` 的值。如果 `listvar` 还不存在则创建它；如果 `key` 不存在，则添加这个域并且设置其值为 `value`；如果 `key` 对应的域已经存在，则修改其值；可以同时设置或者修改多个域。

2. `keylget listvar ?key? ?retvar | { }?`

1) 用来获取参数 `key` 指定域的值。如果没有指定 `key`，那么就返回 `listvar` 的所有索引；否则返回指定域的值。如果 `key` 域不存在，那么就抛出异常！

2) 如果给出了参数 `retvar`，那么域值放在变量 `retvar` 中返回，命令只是返回 1 或者 0，分别表示域 `key` 存在与否。给出了 `retvar` 之后，命令不会抛出异常；

3) 如果参数 `retvar` 给出了，但是是空，也就是 `{}`，那么命令仅仅返回 1 或者 0，表示域 `key` 是否存在；

3. `keylkeys listvar ?key?`

返回 `listvar` 中所有域的索引列表，如果指定了参数 `key`，那么就返回 `key` 这个域值中的所有子域的索引列表。

4. `keyldel listvar key`

从 `listvar` 中删除索引为 `key` 的域，包括索引和值都会被删除掉。

请看下面的例子：

```
% keylset lei NAME.FIRST Yuhou NAME.LAST Lei EMAIL leiyuhou@huawei.com
% puts $lei           ;#命令 keylset 创建索引列表变量 lei
{NAME {{FIRST Yuhou} {LAST Lei}}} {EMAIL leiyuhou@huawei.com}
% keylset lei ID 30000 ;#插入一个域 ID
% puts $lei
{NAME {{FIRST Yuhou} {LAST Lei}}} {EMAIL leiyuhou@huawei.com} {ID
30000}
```

```
% keylkeys lei           ;#返回 lei 的所有 key
NAME EMAIL ID
% keylkeys lei NAME      ;#返回 lei 的域 NAME 的值中所有的索引
FIRST LAST
% keylget lei            ;#返回 lei 的所有索引
NAME EMAIL ID
% keylget lei NAME       ;#返回 lei 的域 NAME 的值
{FIRST Yuhou} {LAST Lei}
% keylget lei NAME.FIRST ;#返回域 NAME.FIRST 的值，这是嵌套的索引
Yuhou
% keylget lei NAME.middle ;#查询一个不存在的域值，抛出异常
key "NAME.middle" not found in keyed list
% keylget lei NAME.middle "" ;#不存在的域值，但是给出 retval，则返回 0
0
```

## 和 array 性能比较

乍看一下，索引列表和 TCL 的数组比较类似，都是一个索引对应一个值的组合。但是索引列表比数组有如下优势：

1. 索引列表可以作为过程的返回值，但是数组不可以（其实可以返回数组名字）；
2. 索引列表可以嵌套，但是数组则比较困难；

所以在很多的时候，索引列表可以代替数组。但是在性能上，数组要比索引列表高，请看下面的测试代码：

```
#比较 keyed list 和 array 的性能
proc keyl_PerfTest {} {
    #初始化索引列表和数组
    for {set i 1} {$i<=10000} {incr i} {
        array set v_a "key_${i} value_${i}-[string repeat VAL_ 10]"
        keylset v_l key_${i} value_[string repeat VAL_ 10]
    }

    #分别查询不同下标的元素值，并且计算其性能值
    foreach i {1 10 50 100 500 1000 2000 4000 5000 8000 10000} {
        set k key_${i}
        set r1 [time {
            set v [set v_a($k)]
        }]
```

```
    } 100]    ;#取得数组的值，计算 100 次的平均值

    set r2 [time {
        set v [keylget v_l $k]
    } 100]
    #输出计算得到的微秒值
    puts "key_$i,[lindex [split $r1] 0],[lindex [split $r2] 0]"
}
keyl_Pertest
```

上面的代码中，分别创建了一个包含 10000 个元素的索引列表变量和数组变量，然后计算读取元素的时间。在我机器上的运行结果经过整理后，如下所示：

下标	数组	索引列表
key_1	5	8
key_10	4	10
key_50	4	20
key_100	4	36
key_500	4	138
key_1000	4	316
key_2000	4	741
key_4000	4	1727
key_5000	4	2149
key_8000	4	3451
key_10000	4	4855

显而易见，对于数组，任何一个元素的存取时间基本上是相同的，但是对于索引列表而言，存取事件随着下标的增加而线性增加。这是列表的线性结构所决定的。所以对于线性列表的使用，我们给出如下建议：

1. 不要使用它保存太多的域，否则性能会极大下降，如果需要保存大量元素，可以考虑数组；
2. 数组和索引列表结合，可以很好的描述数据库的一张表。表的主键作为数组下标，数组的值则是索引列表类型，用来描述一条记录；表的每一个字段名字对应索引列表的一个域。

## 文件操作命令

Tclx 提供了简化我们文件操作的命令，我们这里只介绍常用的 `lgets`, `read_file`, `for_file`。除此之外，还提供了一个扫描文件的机制，可以比较方便的对文件进行逐行扫描分析。下

面我们分别介绍。

## lgets 读取列表

gets 是 TCL 标准命令，用来从文本文件中读入一行内容。但是我们有些时候需要将读入的一行或者多行内容解析成一个列表，命令 lgets 正好可以完成这个功能，其格式如下：

```
lgets fileid ?listvar?
```

和 gets 命令的参数类似。fileid 是文件句柄，如果参数 listvar 省略，那么就将解析后的列表返回，否则就将列表存储在变量 listvar 中。

即使列表元素内容有换行，lgets 也能解析出来，这一点功能尤其可贵！否则我们自己手工编程解析，会有很大的麻烦。例如下面的一个文本文件：

```
{张三} 24 {研发部部长}  
{李四} 25 {销售部经理，  
    2000 年毕业}  
王五 30 "人力资源部部长  
    原来从事销售工作"
```

该文件由六行组成：最后一行是空行；第一行是一个完整列表；第二行和第三行一起组成一个完整列表，其最后一个元素用大括号括起来并且分行了；四五行一起组成一个列表，最后一个元素用双引号括起来。使用 lgets 可以方便地解析成三个列表，请看代码：

```
set fpath "c:/a.txt"  
set fh [open $fpath r]  
  
while {[eof $fh]==0} {  
    lgets $fh v ; puts "list : [lindex $v 0] , [llength $v]"  
}  
close $fh
```

上面代码执行结果如下：

```
list : 张三 , 3  
list : 李四 , 3  
list : 王五 , 3  
list : , 0
```

最后一个空行也被读出来成为一个空列表。这一个空行是必须的，否则 lgets 会无法解析最后一个列表。

## read\_file

`read_file` 命令可以直接读取一个文件的全部内容，它有两种格式：

```
read_file ?-nonewline? fileName
```

```
read_file fileName numBytes
```

第一种格式中，命令会读取 `fileName` 所表示文件的全部内容并且返回；如果指定了选项 `-nonewline`，那么文件最后一个字符如果是换行的话会被丢弃。第二种格式中，命令会直接读取文件 `fileName` 中的 `numBytes` 个字节的内容；如果文件字节数比 `numBytes` 少，那么就返回所有内容。

## for\_file

我们可能经常碰到这样的操作：打开文件，读取每一行内容，针对每一行内容执行一些操作，最后关闭文件。`TclX` 提供了 `for_file` 命令，将这种模式简化，其格式如下：

```
for_file var filepath code
```

其中参数 `var` 是变量名，`filepath` 是要读取的文件名，`code` 则是针对每一行要执行的代码。下面的例子用来打印整个文件：

```
for_file line c:/a.txt {puts $line}
```

下面的代码用来统计文件中的空行数量：

```
set emptyLines 0
for_file line c:/a.txt { if {[string length $line]==0} {incr emptyLines} }
puts "Empty lines : $emptyLines"
```

## 文件扫描

在介绍 `TclX` 的文件扫描功能之前，先介绍一下 `AWK`：一个神奇的 `Unix` 命令。它设计得非常精巧，经常被用来做文件扫描分析处理。而且它有自己的程序设计语言：`AWK`。在扫描文件的时候，能够完成很多分析和处理。事实上，`AWK` 的名字来源于其三个作者的首字母：`Aho`，`Weinberger` 和 `Kernighan`。

看下面的文本文件 `books`（这又是 `Windows` 带来的，`Unix` 下面没有文本和非文本差别）：

Author	Title	Prices
Smith	Ants	\$39.5
Jones	Cats	\$23.8
Brown	Dogs	\$19.5

我们可以执行如下的 `awk` 命令为每一行的前面增加一个行号：

```
awk '{printf "\t%03d\t%s\n", NR, $0}' books
```

该命令会在屏幕上打印出如下的内容：

001	Author	Title	Prices
002	Smith	Ants	\$39.5
003	Jones	Cats	\$23.8
004	Brown	Dogs	\$19.5

如果使用 `AWK` 语言，还能完成更加强大的分析转换功能。这也是 `Unix` 的迷人之处：在 `Unix` 世界中存在很多让人着迷的工具，包括本书介绍的 `TCL` 也是来源于 `Unix`。我们不介绍 `AWK`，关于它我们可以写成另外一本书。我们回到 `TCL`，`TclX` 提供了类似的文件扫描分析功能，但是在用法上和 `awk` 有很大不同。一般分成如下步骤：

1. 创建一个扫描上下文对象，得到该对象句柄；需要用到命令 `scancontext create`；
2. 设置对象的模式和对应对动作；用到命令 `scanmatch`；
3. 打开文件，开始扫描；使用命令 `scanfile`；
4. 删除扫描上下文对象，使用命令 `scancontext delete`

为扫描对象设置模式和动作的命令格式如下：

```
scanmatch ?-nocase? contexthandle ?regexp? command
```

其中 `contexthandle` 参数就是 `scancontext create` 返回的对象句柄，选项参数 `regexp` 表示用来进行匹配校验的正则表达式，`command` 则是需要执行的命令。如果省略 `regexp` 参数，那么 `command` 则是一个默认命令，只有这一行不跟任何模式匹配的时候才执行这个命令。

1. 我们可以为一个 `contexthandle` 多次调用 `scanmatch` 命令，设置不同模式的动作；
  2. 如果某一行能匹配多个模式，那么这些模式的对应命令会按照模式被设置的顺序来执行；
  3. 如果某命令中执行了 `continue` 命令，那么后面所有也能够匹配该行的命令会跳过；
  4. 如果某命令中执行了 `return` 命令，那么扫描过程会中止，并且返回 `return` 的结果；
- 设置好模式和命令之后，就可以调用 `scanfile` 来执行扫描分析了，格式如下：

```
scanfile ?-copyfile fileId? contexthandle fileId
```

参数 `contexthandle` 是扫描对象句柄，`fileId` 是我们需要扫描分析的文件句柄，选项参数 `-copyfile fileId` 用来指定一个文件句柄，那些没有匹配成任何一个模式（包括默认模式）的行，会被写入到这个文件中。

在匹配执行的过程中，数组变量 `matchInfo` 会包含相关的匹配信息：

`matchInfo(line)`：当前行的内容；

`matchInfo(offset)`：当前行的第一个字符在整个文件中的偏移字节数；

`matchInfo(linenum)`：当前扫描行的行号；

`matchInfo(context)`：当前使用的扫描上下文对象的句柄；



`matchInfo(handle)`: 当前扫描处理的文件的句柄;

`matchInfo(submatch0)`: 模式中第一个子串的匹配内容; 后面依次为 `submatch1` 等;

`matchInfo(subindex0)`: 第一个被匹配的子串内容的开始和结束位置, 是一个列表;

下面, 我们使用 TCL 来完成上面的扫描分析功能, 请看代码:

```
proc scanfile_test {} {  
    #创建一个扫描对象, 句柄保存在 h_scan 变量中  
    set h_scan [scancontext create]  
  
    #设置扫描模式, .+对应后面的 puts 命令  
    scanmatch $h_scan ".+" {puts "$matchInfo(linenum)    $matchInfo(line)"}  
  
    #打开文件, 开始扫描  
    set fh [open c:/books r]  
    scanfile $h_scan $fh  
  
    #删除扫描对象  
    scancontext delete $h_scan  
}  
scanfile_test
```

该段代码的执行结果和我们上面给出的 `awk` 命令执行结果相同, 分析处理后的结果直接打印到标准输出。不同之处在于: `awk` 要简单明了的多。



## 使用 tcltest 和 TclUnit 进行测试

很多程序员对自己的代码非常自信：我写的代码不可能有错误！系统只通过简单的联调就发布了。事实上并非如此，没有经过充分测试的系统中隐藏着很多未知的 bug，并且迟早会爆发。尽早发现 bug，就能够把损失尽可能降低。

软件测试按照阶段可以分成：单元测试，集成测试和系统测试等。测试活动的一个特点就是：重复！每次更新发布一个版本，都需要将测试进行一遍，所以测试自动化现在受到了广泛的关注和重视，要实现测试自动化，编写测试脚本都是必不可少的；TCL 脚本非常适合编写测试脚本，来进行自动化测试。

TCL 中有两个测试框架可以供我们选择，tcltest 和 tclUnit。前者随同 ActiveTcl 发布，后者则需要我们下载。下面我们分别介绍。

### tcltest

tcltest 是一个扩展包，它提供了一整套编写测试用例的命令和方法。这些方法都放在名字空间 tcltest 中，通过这些方法，我们可以编写许多测试用例并且组织成测试套。tcltest 整套体系比较复杂，命令和选项繁多，我们逐步介绍。

### 一个简单的用例

进行自动化测试需要编写测试脚本，测试脚本中必须至少包含如下几个要素：

1. 测试操作步骤：进行测试必须对被测试系统进行一些操作；
2. 操作的实际结果：操作后，脚本中应该获得实际的操作结果；
3. 期望的结果：对于刚才的操作，我们期望出现什么结果？
4. 比较结果：比较实际结果和期望结果，如果一致则该测试点测试通过；

下面我们测试一下 Tcl 的标准命令 incr，看看它是否真的实现了其功能规格。对应的测试代码如下，它保存为 incr\_test.test 文件：

```
001 #File Name : incr_test.test, Could be run by tclsh.exe
002 #Author    : LeiYuhou
003
004 package require tcltest
005
006 tcltest::configure -verbose {start body pass}
007
```

```
008  ::tcltest::test incr.1 "验证 incr 的功能"\  
009  -body {  
010      set m 100  
011      incr m 100  
012  } -result {200}
```

第 3 行，首先引入了 `tcltest` 扩展包；然后调用 `configure` 命令设置 `verbose` 选项，这是设置为 `start`, `body` 和 `pass`；第 6 行调用 `tcltest::test` 命令来定义测试用例，这个命令是重点，其参数解释如下：

1. 第一个参数 `incr.1` 为用例名字；
2. 第二个参数是用例描述字符串，描述该用例的相关信息；
3. `-body` 选项用来定义用例的“操作步骤”；它是一脚本，其执行结果就是实际结果；
4. `-result` 选项用来指定期望结果；`test` 会把两个结果进行比对；

除了 `body` 和 `result` 选项之外，`test` 命令还有一些其它选项，我们后面介绍。上面的代码执行结果如下：

```
---- incr.1 start  
++++ incr.1 PASSED
```

因为 `incr m 100` 是 `body` 的最后一个命令，所以其结果 200 就是操作步骤的实际结果；命令 `test` 将其和 `-result 200` 进行比对，两者完全一致，所以用例执行通过！

## 匹配模式、准备和清除

从刚才的例子可以看出，命令 `tcltest::test` 是 `tcltest` 的核心。其前两个参数分别是测试用例名字和描述字符串，后面则是一系列的选项参数，可以通过设置它们来设定用例属性。我们考虑如下的一个问题：

下面的情况是比较常见的：在执行用例的操作步骤之前，我们需要做一些准备工作；在执行完毕之后，要执行一些清除工作。怎样实现这些需求呢？请看例子文件 `format.test`：

```
001  #File Name : format.test, Could be run by tclsh.exe  
002  #Author    : LeiYuhou  
003  
004  package require tcltest  
005  tcltest::configure -verbose {start body pass}  
006  
007  ::tcltest::test format.1 "验证 format 的功能"\  
008  -setup {  
009      set name "LilyLei"
```

```
010      set age 28
011  } -body {
012      format "%s : %d" $name $age
013  } -cleanup {
014      unset name age
015  } -match regexp -result {\S+ : \d+}
```

这里除了 `-body` 和 `-result` 选项之外，出现了 `-setup`、`-cleanup` 和 `-match` 三个新的选项，其中：

1. `-setup` 用来指定测试前的准备操作代码，这部分代码在 `-body` 之前执行；一般用来构建测试环境，初始化变量等；
2. `-cleanup` 则用来指定操作执行之后的清除代码，它在 `-body` 之后执行；它一般是用清除和恢复测试环境，释放资源等；
3. `-match` 用来指定结果匹配模式，可以为 `exact`、`glob` 和 `regexp` 三种中的一个；如果不指定该选项，那么就默认为 `exact`，即“精确匹配”；

刚才的例子中，`-match` 指定匹配模式为 `regexp`，即正则表达式匹配，那么 `-result` 给出的期望结果应该是一个正则表达式。如果它能够和实际结果匹配，那么测试就成功。

## 检查输出信息

我们检查的结果是选项 `-body` 命令执行后的返回结果；除此之外，还可以指定 `-output` 和 `-errorOutput` 选项来给出期望的输出结果，即检查 `puts` 命令打印出来的信息！为此我们在刚才的 `format.test` 文件后面增加如下代码：

```
018  ::tcltest::test format.2 "验证 format 的功能 2" \
019  -setup {
020      set name "LilyLei" ; set age 28
021  } -body {
022      set r [format "%s=%d" $name $age]
023      puts $r ; puts stderr "Error:$r" ; set r
024  } \
025  -match regexp \
026  -result {\S+=\d+} \
027  -output {LilyLei=\d+} \
028  -errorOutput {Error:LilyLei=\D+}
```

这里的 `body` 中，我们分别向 `stdout` 和 `stderr` 输出了信息，并且最后的命令 `set r` 保证整个 `body` 的命令执行结果就是 `$r`。随后的几行代码中：

1. 25 行的 `-match` 指定结果匹配模式为 `regexp`；它既适合命令结果，也适合输出结果；

2. 26 行的 `-result` 指定了期望的命令执行结果，它是一个正则表达式；
3. 27 和 28 行用 `-output` 和 `-errorOutput` 选项分别指定了在 `stdout` 和 `stderr` 两个通道上的期望结果；`-body` 中所有向“标准输出”和“标准错误”两个通道输出的内容，分别和这两个选项内容进行匹配检查，判断是否匹配成功。

执行脚本文件 `format.test`，输出结果如下：

```
---- format.1 start
++++ format.1 PASSED
---- format.2 start

===== format.2 验证 format 的功能 2 FAILED
===== Contents of test case:

    set r [format "%s=%d" $name $age]
    puts $r ; puts stderr "Error:$r" ; set r

---- Error output was:
Error:LilyLei=28

---- Error output should have been (regexp matching):
Error:LilyLei=\D+
===== format.2 FAILED
```

从结果可以看到，用例 `format.1` 测试通过，但是 `format.2` 失败了；结果中打印出详细的错误信息，原因在于实际 `Error output` 和期望不匹配。要注意的是，只有 `::puts` 命令输出的内容才会被检查。

## 检查异常以及特殊返回码

先回顾一下 `incr` 命令，它用来改变一个变量的数值。其格式如下：

```
incr varname ?value?
```

它正常工作的前提是：

1. `varname` 是已经存在的变量名；
2. 变量 `varname` 的值必须是整数；
3. 参数 `value` 如果给出，必须是整数；

如果违反了上面几条，`incr` 命令就会抛出异常。假定这是 `incr` 命令的设计规格，现在我们需要验证这几点。如何使用 `tcltest` 来编写测试用例呢？`tcltest::test` 命令还提供了一个

选项 `-returnCodes`，通过指定该选项，可以对命令执行的 `return` 代码进行比较判断，这里的 `return` 值包括了 `return`，`break`，`error` 等情况。请看例子：

```
031  ::tcltest::test returnValue.1 "验证异常情况" \  
032      -body {  
033          set m 100.00      ;#变量 m 的初始值是一个浮点数  
034          incr m            ;#这里会抛出异常  
035      } \  
036      -returnCodes {error} \  
037      -match regexp \  
038      -result {expected integer but got ".*"}
```

这里 36 行，设置 `-returnCodes` 选项为 `error`，表示期望执行的 `return` 代码为 1，也就是“期望抛出异常”；37 行设置结果比较采用正则表达式进行比较；38 行则是设置抛出异常的具体信息，我们期望实际的异常信息能够和字符串 `expected integer but got ".*"` 匹配上。如果你对这里的“命令返回代码”和“返回结果”有疑问，可以参考本书前面对 `return` 命令的详细介绍。上面代码执行情况如下：

```
---- returnValue.1 start  
++++ returnValue.1 PASSED
```

测试通过！我们试着改动一下：将 36 行的 `returnCodes {error}` 修改成 `-returnCodes {ok}`，再次执行该脚本，结果如下：

```
---- returnValue.1 start  
  
===== returnValue.1 验证异常情况 FAILED  
===== Contents of test case:  
          set m 100.00  
          incr m  
  
---- Test generated error; Return code was: 1  
---- Return code should have been one of: 0  
===== returnValue.1 FAILED
```

测试用例执行失败（FAILED）了，失败原因在结果中指出：期待 `ReturnCode` 是 0（ok），但是实际的 `returnCode` 却是 1（error）。

总结一下，如果使用 `tcltest` 来测试命令应该抛出的异常，应该：

1. 指定 `-returnCodes` 选项为 `error`；
  2. 指定 `-result` 为期待的异常信息，也就是抛出异常后全局变量 `::errorMsg` 的值；
- 选项 `returnCodes` 的值可以是一个列表，只要命令返回代码能够匹配其中一项，那么

就算匹配上。如果不指定该选项，那么就默认为{ok return}。该选项可以取值为：ok、error、return、break 或者 continue。下面我们看看 break 的情况，请看代码：

```
042 #定义一个 breakProc 过程，只返回 break 代码
043 proc breakProc {msg} {
044     return -code break $msg
045 }
046
047 ::tcltest::test breakTest.1 "验证 break 情况" \
048     -body {
049         set m 100.00
050         breakProc "Break from here"
051     } \
052     -returnCodes {break} \
053     -match regexp \
054     -result { }
```

我们定义了过程 breakProc，它除了返回 break 返回码之外什么都不做。后面我们的测试用例中来验证其功能：-returnCodes 选项指定期望返回码为 break。这样设计的测试用例我们用到机会比较少。

## 组织运行多个测试用例

一个 tcltest::test 命令只能对一个脚本块进行检查；并且只能指定一个具体的期望结果。这对稍微复杂一些的被测试对象，都是远远不够的。比如我们刚才提到的 incr 命令，要想对其“什么情况下会抛出异常”这个规格进行验证，至少需要多个 test 命令才行。如果是更加复杂的被测试对象，需要的测试用例是成百上千的。而且测试用例是需要维护的，因为它被我们写出来之后不是运行一次就扔掉了，而是会被多次用到。所以如何来编写、组织测试用例，使其具有良好的可维护性、可读性以及可重用性是非常重要的。

## 组织测试用例

tcltest 提供了组织测试用例的机制，我们一般按照如下规格来组织：

1. 相关的多个 tcltest::test 命令可以放在一个.test 文件中；比如测试某个过程的所有测试用例，可以放在一个.test 文件中。
2. 多个相关的.test 文件可以放在同一个目录下；比如测试某一个模块所有函数的多个.test 文件。在该目录下生成一个 all.tcl 文件，将本目录下的所有.test 文件 source



进来;

3. 在最上层的目录下, 创建一个 `test.tcl` 文件, 在该文件中需要对测试过程进行配置, 然后执行命令 `runAllTests`, 来执行本目录以及所有子目录下的所有文件。

请看例子, 这是一个测试套的目录树结构:

```
M:.  
|  add.test  
|  format.test  
|  test.tcl          =>这是整个测试套测试的入口脚本  
|  
└─string  
    |  all.tcl        =>该目录下的 all.tcl  
    |  trim.test  
    |  
    └─incr  
        all.tcl  
        incr.test
```

两个问题我们有必要关注, 一个是各个子目录下的 `all.tcl` 文件:

```
001 #-----  
002 #File Name    : all.tcl  
003 #Description  : 执行本目录下的所有 .test 文件  
004 #Author       : LeiYuhou  
005  
006 package require tcltest  
007  
008 set d [file dirname [info script]]  
009 set testlist [glob -directory $d -types f *.test] ;#找出所有的*.test 文件  
010  
011 foreach testfile $testlist {  
012     source $testfile  
013 }  
014
```

上面代码很简单, 这里不作介绍。另外一个是最上层目录下的 `test.tcl`, 其内容如下:

```
001 #-----  
002 #File name     : test.tcl  
003 #Description   : 配置测试套, 执行所有测试用例  
004 #Author        : LeiYuhou
```

```
005
006 package require tcltest
007 tcltest::configure -singleproc 1 -verbose {pass skip start}
008 tcltest::configure -limitconstraint 0
009 tcltest::configure -testdir [file dirname [file normalize [info script]]]
010
011 tcltest::runAllTests
```

这个文件中 7—9 三行调用了 `configure` 命令对测试过程进行了一些配置，然后就调用了命令 `runAllTests`，这个命令根据配置情况来选择执行测试用例，其运行过程如下：

1. 在指定的目录[`configure -testdir`]下，寻找所有和[`configure -file`]中任何一个模式匹配但是不和[`configure -notfile`]中任何一个模式匹配的那些文件。
2. 如果[`configure -singleproc`]的值为 `true`，所有的测试文件将在主进程中被执行；否则，主进程会为每一个文件创建一个子进程，在子进程中执行测试文件。
3. 递归搜寻子目录，所有能够和[`configure -relatedir`]中任何一个匹配模式但是不和[`configure -asidefromdir`]中任何一个模式匹配的子目录，都会被搜寻。`runAllTests` 会在这些目录中查找 `all.tcl` 文件，如果 `all.tcl` 存在，那么就会被 `source` 到主进程的上下文中。

上面出现了很多[`configure -XXX`]，它表示配置项 `XXX` 的值。`runAllTests` 根据我们描述的这个运行过程，就能够把一个目录连同其所有的子目录中相关的`*.test` 文件都执行到。

## 配置测试过程

刚才看到的命令 `configure`，是用来配置测试过程，从而来具体控制 `runAllTests` 的具体执行方式。命令 `configure` 的格式如下：

```
tcltest::configure option value ?option value?
```

常见的配置项还有等价的简便命令，来达到同样的功能。常用配置选项有：

1. `configure -match ?patternList?`：指定模板的列表，只有名字能够和这个列表匹配的那些测试用例，才能够被执行；其简化命令为 `match ?patternList?`。该选项默认值为 “\*”
2. `configure -skip ?patternList?`：指定模板列表，名字和这个列表匹配的那些测试用例，都不能执行而跳过去。简化命令为 `skip ?patternList?`。默认值为空。
3. `configure -testdir directory`：指定测试文件所在的目录。`testAllTests` 会在这个目录中搜寻测试文件。
4. `configure -file ?patternList?`：指定模板列表，只有那些能够跟其中至少一个模板匹配的测试文件，才能够执行。简化命令为 `matchFiles ?patternlist?`，其默认值为 `*.test`。

5. `configure -notfile ?patternList?`: 指定模板列表, 那些名字和该列表中任意一个模式匹配的测试文件, 都被排斥在外, 不予执行。其简化命令为 `skipFile ?patternList?`, 该选项的默认值为 “`l*.test`”。
6. `configure -singleproc ?boolean?`: 该选项控制是否为每一个单独的测试文件创建独立的子进程, 然后在子进程中执行这个文件。简化命令是 `singleProcess ?bool?`, 默认情况为 `true`。

可以看到, 测试用例以及测试文件的命名最好也要符合一定的规则, 这样我们就可以方便的使用配置选项命令来选择需要测试的用例, 剔除暂时不需要测试的用例。

## tclUnit

如果读者使用过 `CPPUnit` 或者 `JavaUnit` 做过单元测试, 一定对这两种测试架构中独特的用例编写方式和用例组织方式映象深刻。事实上, 这种单元测试架构来源于所谓的“极限编程”, 最早在 `SmallTalk` 语言上实现, 随后就在极短的时间内风靡到了所有的语言中, 几乎在当今所有的语言上都实现了类似的测试架构, 我们统统称之为 `XUnit`。在下面的网址能够找到几乎所有的 `XUnit` 实现:

<http://c2.com/cgi/wiki?TestingFramework>

`XUnit` 架构在 `TCL` 语言上的实现就是 `tclUnit`, 使用过 `CPPUnit` 的高手能够在 `tclUnit` 上找到那种似曾相识的感觉。

## 下载安装 tclUnit

`tclUnit` 不随着 `ActiveTcl` 发布, 所以需要我们到下面的网址去下载:

<http://sourceforge.net/projects/tclunit/>

下载得到一个压缩文件, 本书写成的时候其版本号为 0.9。按照如下操作来安装它:

1. 将其解压缩到 `tclunit0.9` 目录下, 我机器上是 `E:\tclunit0.9`;
2. 得到了三个子目录: `extensions`、`samples` 和 `tests`, 我们不要管它。除此之外, 在 `E:\tclunit0.9` 目录下, 还有 11 个 `.Tcl` 文件以及一个 `tunit` 文件;
3. 在 `TCL` 解释器的 `lib` 目录下创建子目录 `tclunit-0.9`; 然后将刚才提到的 12 个文件全部复制到这个目录中; 我机器上的目录是: `C:\tcl\lib\tclunit-0.9`;

完成上面操作之后, 需要验证一下是否安装正确。我们直接使用 `tclunit` 提供的 `samples` 目录下的例子文件 `basic.tcl` 来验证, 请执行下面的命令:

```
tclsh.exe c:\tcl\lib\tclunit-0.9\tunit e:\tclunit0.9\basic.tcl
```

如果出现如下结果, 就说明安装正确! 否则就有问题 (希望不要发生这种悲剧):

```
<summary>
```

```
pass basic.tcl                      Total    1    Pass    1    Fail    0
</summary>
```

我们可以看到，执行 tclUnit 测试用例脚本的时候，需要用到脚本文件 tunit，这个文件是 tclUnit 安装包提供的。执行用例的一般方式如下：

```
tclsh.exe tunit unitCaseFile
```

tunit 是脚本解释器的第一个参数，而第二个参数是用例脚本文件的名称。

## tclUnit 用例入门

tclUnit 是采用 Itcl 面向对象的方式实现的，并且开放源代码。即使如此，它的不足之处仍然比较明显：tclUnit 的发布包中居然没有任何有用的文档！不知道这是不是开源项目独特的特点！没关系，我们从 tclUnit 提供的 samples 入手，来逐步掌握其运行机制和用例编写方式。

首先看看 samples/EvalTest.tcl 文件（我在源文件上加上注释）

```
001 #####
002 #File Name      : EvalTest.tcl
003 #Description    : a TestCase that evals its -script argument
004
005 if {[itcl::find class EvalTest] != ""} {
006     # 如果类 EvalTest 已经存在了，直接返回
007     return
008 }
009
010 #定义测试用例类 EvalTest
011 itcl::class EvalTest {
012     inherit ::tclunit::TestCase ;#从类 tclunit::TestCase 派生
013
014     #构造函数，传入配置参数
015     constructor {args} {
016         eval configure $args
017     }
018
019     #虚函数 runTest，用来描述测试过程
020     method runTest {} {
021         incr runCount ;#累加
022         eval $script ;#执行测试过程脚本
```

```
023     }
024
025     public variable script      ;#测试过程需要执行的脚本
026     common runCount 0          ;#执行次数
027 }
```

从第 11 行开始定义一个测试用例类 EvalTest，它从::tclunit::TestCase 类派生。记住：所有的测试用例都是一个 itcl 的类，并且从 TestCase 类派生出来。EvalTest 定义了构造函数，其参数用来进行初始化；同时定义了一个成员函数 runTest，这是一个在基类 TestCase 中定义的虚函数，我们在这里对其进行重载，其实现就是把 script 给执行一遍。

这样的用例是一个空架子，不能够执行。我们需要构造它，然后进行初始化设置后才能够执行，这些工作在脚本 samples/basic.tcl 中实现，我做了一些修改，请看：

```
001 #####
002 # File Name      : basic.tcl
003 # Description    : 一个简单的测试套，使用了 EvalTest 用例
004
005 # 首先需要加载用例类 EvalTest
006 source [file join [file dirname [info script]] EvalTest.tcl]
007
008 if {[itcl::find class basic] != ""} {return}
009
010 #定义类 basic，文件 basic.tcl 必须包含一个类"basic"
011 itcl::class basic {
012     proc suite {} {
013         #创建一个测试套类 TestSuite 的对象实例 suite
014         set suite [new TestSuite -name basic]
015
016         #向测试套实例 suite 中增加一个测试用例 EvalTest 的实例
017         $suite addTest [new EvalTest -name Whoville -script {
018             assertTrue "[expr 100+100]==200"
019         }]
020
021         #必须将测试套实例 suite 返回
022         return $suite
023     }
024 }
```

第 6 行首先执行同目录下的 EvalTest.tcl 文件，第 11 行开始定义类 basic，其中只定义

了一个 proc 类型的过程 suite，该过程中：

1. 14 行创建一个类 TestSuite 的实例，测试套命名为 basic；
2. 17 行创建了一个类 EvalTest 的实例，用例名字为 Whoville，脚本内容只是调用了  
一个断言 assertTrue，断言 100+100 等于 200；然后调用测试套 suite 的方法 addTest，  
将这个新的测试用例加入到测试套中；
3. 22 行，将新创建的测试套实例返回；

我们调用命令行： `tclsh.exe tuit basic.tcl` 来执行这个测试用例。结果如下：

```
<summary>
pass  basic.tcl                      Total    1  Pass    1  Fail    0
</summary>
```

说明测试用例执行成功。这里我们总结一下，使用 tunit 来运行测试脚本 xxx.tcl 的时候，其中测试用例的实现方法：

1. xxx.tcl 中必须定义一个类 xxx；
2. 在 xxx 中必须定义一个 proc 类型的成员函数 suite；
3. 在 suite 函数中创建类 TestSuite 的实例，并且向其中增加测试用例；其方法就是  
调用 addTest 方法向测试套中增加测试用例类的实例；
4. 一个测试用例类，就是从 TestCase 派生下来的类，重载其 runTest 函数即可；

## 增加一个用例

我们可以在刚才的 basic::suite 过程中第 21 行添加如下的代码，来增加另一个用例：

```
021      $suite addTest [new EvalTest -name Whoville2 -script {
022          assertTrue "100*100==20000"
023      }]
```

再次执行这个文件，结果如下：

```
FAIL  Whoville2 assertion failed: {100*100==20000}
<summary>
Failing Test          Error Message
-----
Whoville2             assertion failed: {100*100==20000}
-----

FAIL  basic.tcl                      Total    2  Pass    1  Fail    1
</summary>
```

刚才加入的第二个用例执行失败了。因为我们期望  $100*100==20000$ ，这怎么可能呢？

tclunit 中，测试用例被包含在测试套中。我们可以测试套类比为文件系统的目录，而测试用例则类似文件。测试套可以嵌套包含多个测试用例和多个测试套。下面我们来看看一个测试套嵌套的例子，这个例子在 sample/composite.tcl 文件中：

```
001 #####
002 # File Name      : composite.tcl
003 # Description    : 一个测试套，包含了另外一个测试套和一个用例
004
005 if {[itcl::find class composite] != ""} {return}
006
007 itcl::class composite {
008     proc suite {} {
009         #创建测试套实例 suite
010         set suite [new TestSuite -name composite]
011
012         #增加测试套 basic 和 测试用例 Horton
013         $suite addTestSuite basic
014         $suite addTest [new EvalTest -name Horton -script {
015             assertTrue "100==100"
016         }]
017
018         return $suite
019     }
020 }
```

composite.tcl 文件和 basic.tcl 文件非常类似，里面定义了类 compsite，它只有一个方法 suite，在这个方法中创建了一个测试套实例，然后调用 addTestSuite 和 addTest 分别将测试套 basic 和一个用例加入到这个新创建的测试套中。测试套 basic 就是我们上面在 basic.tcl 中实现的测试套，读者会问了：composite.tcl 文件中从来没有 source 这个 basic.tcl 文件，它是如何找到测试套 basic 的呢？这是 tclUnit 内部实现的一个查找功能。这里我们只需要知道 tclUnit 会在当前目录中查找 basic.tcl 文件即可。

## 写一个简单的测试构件

到现在为止，不知道你是不是对 tclUnit 丧失了信息和兴趣：这都什么东西？比起 tcltest 要复杂多了！如果真是这样，那我必须向你解释，因为我把一些复杂的东西放在前面先讲了，目的在于让我们一开始就能够对用例结构有一个比较深入的了解，下面我们来看看怎

样写一个简单的测试用例（其实其内部运行机制一点都不简单）。请看文件 fixture.tcl:

```
001 #####
002 #File Name      : fixture.tcl
003 #Description    :
004
005 #定义测试用例类 fixture
006 itcl::class fixture {
007     inherit ::tclunit::TestCase
008
009     private variable _answer 42
010     common setupCount 0    ;#记录 setUp 函数被执行的次数
011     common runCount 0      ;#记录运行了多少个用例
012
013     constructor {} {
014         puts "Constructor : $_answer" ;#构造函数
015     }
016
017     method setUp {} {      ;#用例执行前的准备工作
018         incr setupCount
019         puts "->setUp. order = $setupCount. answer=$_answer."
020     }
021     method tearDown {} {   ;#执行后的清除工作
022         puts "->tearDown."
023     }
024
025     method test1 {} {
026         incr runCount
027         assertEquals $_answer 42
028         set _answer 0      ;#修改成员变量的值
029     }
030     method testAnotherThing {} {
031         incr runCount
032         assertTrue {$_answer == 42}
033         set _answer -1
034     }
035     method testWhateverAndEver {} {
036         incr runCount
```



```
037         assertTrue {$runCount > 0}
038         assertTrue {"blue" == "blue"}
039     }
040 }
```

简单解释如下：文件 `fixture.tcl` 定义了一个从 `TestCase` 派生的类 `fixture`，要反复强调的一点是：类的名字必须和文件名完全一致，否则 `tclunit` 会无法找到正确的用例类。

第 17 行和 21 行分别定义了两个成员函数 `setUp` 和 `tearDown`，这两个 `method` 类型的成员函数分别在每一个测试用例函数运行前和运行后执行，完成准备和清除工作；这两个函数是基类 `TestCase` 类中定义的成员函数，也是虚函数。

第 25、30 和 35 行分别定义了三个名字以字符串“test”开始的测试函数，这三个函数代表三个测试用例。要强调的是：测试用例函数的函数名必须以字符串“test”作为开始，否则得不到执行机会。函数内部调用了 `assertTrue` 和 `assertEqual` 函数进行断言，这两个断言函数是在基类 `TestCase` 中定义的。

执行如下的命令行来对本用例进行测试：

```
tclsh.exe ../tunit -verbose pe., fixture.tcl
```

这里给出的选项参数 `-verbose`，用来控制执行过程中的输出打印信息，这里值为“pe.”，包含了四个字符，含义如下：

1. 字符 `e`：打印出执行失败的测试用例的 `errorInfo` 和 `errorCode`；
2. 字符 `p`：打印执行通过的用例；
3. 句号字符 `.`：开始执行用例时，打印开始信息；
4. 逗号字符 `,`：执行用例结束时，打印结束信息；

上面命令行的执行结果如下（每一行前面的行号是我加上去的）：

```
001 Constructor : 42
002 Constructor : 42
003 Constructor : 42
004 start fixture::test1
005 ->setUp. order = 1. answer=42.
006 ->tearDown.
007 end    fixture::test1 -result {} -code 0 -elapsed 0.002
008 pass  fixture::test1
009 start fixture::testAnotherThing
010 ->setUp. order = 2. answer=42.
011 ->tearDown.
012 end    fixture::testAnotherThing -result {} -code 0 -elapsed 0.002
013 pass  fixture::testAnotherThing
014 start fixture::testWhateverAndEver
```

```
015 ->setUp. order = 3. answer=42.
016 ->tearDown.
017 end    fixture::testWhateverAndEver -result {} -code 0 -elapsed 0.002
018 pass  fixture::testWhateverAndEver
019 <summary>
020 pass  fixture.tcl                                Total    3    Pass    3    Fail    0
021 </summary>
```

乍一看这个结果，觉得很纳闷：这个用例的执行过程究竟是怎样的呢？仔细分析后就不难明白其执行过程：

1. 因为这个 `fixture` 类中包含了三个 `test*`测试函数，所以在执行的时候，首先创建了三个 `fixture` 类的实例；这从构造函数被执行三遍可以得到证实，每一个实例被用来执行一个单独的 `test*`函数。
2. 依次来执行各个 `test` 函数；针对每一个 `test` 函数：
  - a) 首先调用类的 `setUp` 成员函数；如果没有发生异常，则继续下一步；否则本用例执行失败，跳过后面的所有步骤，执行下一个用例函数；
  - b) 然后调用执行 `test*`测试函数；
  - c) 最后调用 `tearDown` 函数；
3. 统计用例执行情况并且打印统计信息；

某个 `test` 函数对的应用例在执行过程中，只有 `setUp`、`test*`函数和 `tearDown` 函数都不发生异常，这个用例才算执行成功；否则就失败。

一个测试构建类中，如果一个用例函数执行失败了，对其它的用例函数有影响吗？回答是没有任何影响。其它的用例函数会照常执行。

细心的读者可能会问一个问题：函数 `test1` 的最后将类成员变量 `_answer` 设置为 0 了，但是下一个用例 `testAnotherThing` 函数中，`assertTrue {$_answer==42}` 居然断言通过，没有抛出异常！这是为什么呢？了解了上面描述的用例执行过程，这个问题就迎刃而解了：因为不同的 `test` 函数是在相互独立的 `fixture` 实例中执行的，而 `_answer` 是一个 `variable` 类型的成员变量，所以最后设置为 0 对其它的用例函数没有任何影响。如果是 `_answer` 是 `common` 类型的变量，那结果就不是这样了。

## 使用断言

我们以前介绍过 `struct::assert` 命令，在 `tclUnit` 中编写用例的时候，也需要使用断言来比较实际结果和期望结果是否一致。`tclUnit` 在类 `TestCase` 中定义了两个成员函数来帮助我们通过断言进行结果比较：`assertEqual` 和 `assertTrue`，其参数形式如下：

1. `assertEqual a b`
2. `assertTrue expr`

`assertEqual` 函数有两个参数，如果它们不相等就抛出异常；而 `assertTrue` 只有一个参数，它会被当成一个表达式被命令 `expr` 来求值，结果为 `False` 的时候就会抛出异常。

只要用例在执行过程中抛出了异常，那么用例就执行失败，不管这个异常是 `assert` 函数抛出的，还是其它类型的异常。

## 动手扩充 tclUnit 的断言

如果用过 `CPPUnit`，应该知道最新的 `CPPUnit` 中提供了非常丰富的 `ASSERT*`宏来进行结果比较，例如：`ASSERT_THROW`，就是用来断言语句应该抛出异常。我们可以在 `tclUnit` 中自己实现这个断言函数，将代码添加在文件 `TestCase.tcl` 的 73 行处，如下：

```
073  #-----
074  #定义断言函数 assertThrow，断言 body 参数应该抛出异常
075  #assertThrow ?-option? body ?errorInfo?
076  #option      : exact , glob , regexp
077  #?errorInfo? : error information expected
078  method assertThrow { args } {
079      package require cmdline
080      set options {
081          {exact      "use exact mode"}
082          {glob       "use glob mode"}
083          {regexp     "use regexp mode"}
084      }
085      array set params [::cmdline::getoptions args $options]
086      set body [lindex $args 0]
087      set expectInfo [lindex $args 1]
088
089      set ret [catch {uplevel $body} msg ]
090      if {$ret==0} {
091          error "No exception throw : $args"
092      }
093
094      if {$expectInfo eq ""} {
095          return;          #期望结果为空，不进行进一步匹配，直接返回
096      }
097
098      if { $params(exact) } {
```

```
099         set mode excat
100     } elseif { $params(glob) } {
101         set mode glob
102     } elseif { $params(regexp) } {
103         set mode regexp
104     } else {
105         set mode exact;      #默认为 exact 模式
106     }
107
108     switch -$mode -- $msg \
109         $expectInfo { return } \
110         default {error "$body throw exception:$msg,but not match"}
111     return
112 }
```

函数 `assertThrow` 的用法如下：

```
assertThrow ?-option? body ?errorInfo?
```

其中 `-option` 参数可以为 `glob`、`exact` 或 `regexp` 中的一个，表示对异常信息进行检查匹配的模式，如果省略则默认为 `exact`；`body` 则是需要执行的程序；选项参数 `errorInfo` 表示期望的异常信息，如果省略，则不进行异常信息匹配，只检查是否抛出了异常。

不知大家是否注意到了，这个函数内部使用了包 `cmdline` 来进行选项参数解析。函数 `assertThrow` 现在成了类 `TestCase` 的成员函数。我们在 `fixture.tcl` 文件中添加如下测试用例：

```
046     method testThrow {} {
047         assertThrow -regexp {
048             set v "Hello."
049             incr v
050         } {integer .*} ;#期望这个语句应该抛出异常
051
052         assertThrow -glob {
053             set v "Hello."
054             incr v
055         } {expected integer but got "*"} ;#期望这里应该抛出异常
056
057         assertThrow {set v ++;incr v} ;#期望这里抛出异常
058         assertThrow {set v 100 ; incr v} ;#断言失败，因为不会产生异常
059     }
```

该测试用例函数中调用了四次 `assertThrow` 函数来进行断言。前面三个断言都会成功，

应为里面的语句 `incr v` 确实会抛出异常；但是最后一个 `assertThrow` 则会失败，因为里面的语句是完全正确的，不会抛出异常，所以这个测试用例会失败，执行结果如下：

```
FAIL  fixture::testThrow No exception throw : {set v 100 ; incr v}
<summary>
Failing Test          Error Message
-----
fixture::testThrow    No exception throw : {set v 100 ; incr v}
-----

FAIL  fixture.tcl          Total    4  Pass    3  Fail    1
</summary>
```

结果中包含了详细的错误信息：`{set v 100;incr v}`没有抛出异常，这不符合我们用例中的期望，所以该用例失败了。

## C 语言扩展和嵌入

阅读这一章，读者必须具备 C 语言技能和相关经验；C++不是必须的，但是能够掌握最好。并且会编写 Windows 或者 Unix 操作系统中的动态链接库。

扩展和嵌入是两个不同的概念，但是实现原理类似。什么叫做“扩展”，就是当 TCL 本身提供的命令不足以满足我们工作的时候，我们使用 C/C++来编写扩展函数，并且将它们加入到 TCL 命令库中，使其成为 TCL 的一部分。我们在脚本里面可以象一般 TCL 命令那样调用它。

“嵌入”指的是将 TCL 解释器嵌入到你的应用程序中，使解释器成为应用程序的一部分，并且在应用程序中增加一些扩展命令。这样应用程序就可以执行 TCL 脚本来完成某些特定的工作。例如：Emacs 就嵌入了 lisp 语言，微软的 VC 6.0 开发系统中嵌入了 VBScript 语言等。

使用 C 和 Tcl 来混合编程是一个复杂的话题，ActiveTcl 的联机帮助手册中有章节来介绍 Tcl 提供的 C 函数接口。本章节不会介绍具体的函数，只介绍通用的原理和方法。

## 用 C 编写扩展命令

我们这里用 C 语言来编写一个 Max 扩展命令，其命令格式和规格说明如下：

```
Max val ?val...?
```

Max 命令带有至少一个参数，参数都是整数形式；命令返回所有参数的最大值。如果参数个数为 0 或者有参数不是整数，那么就抛出异常。我们使用 Visual C++ .net 来实现。里面的路径等都是我系统上的配置，和你的会有差异。在深入介绍动态链接库编写方法之前，首先要介绍一个重要的 load 命令。

## load 命令

load 命令是 TCL 的标准命令，主要用来将外部库文件加载到解释器中，使之成为解释器的一部分。这个库文件一般是动态链接库，在 Windows 系统上是.DLL 文件，在 Solaris 上则是.so 文件；库文件主要用来实现 TCL 的扩展命令。命令格式如下：

1. load filename
2. load filename packageName
3. load filename packageName interp

三种形式中，前两种使用比较普遍，第三种方式在多解释器环境中使用。参数 filename 表示动态链接库的文件名，可以包含完整的路径；packageName 则是需要加载的程序包

package 的名字；参数 `interp` 是解释器的名字；

`load` 命令在加载文件的时候，需要完成三个工作：

1. 如果动态链接库还没有映射到解释器进程的地址空间中，那么就将其加载进来；如果进程空间中已经有该文件的映象，那么就跳过该过程；在 Windows 系统中，可以使用 Win32API 函数 `LoadLibrary` 来完成这个操作；
2. 执行动态链接库的入口函数。如果是 Windows 操作系统，那么会执行 `DllMain` 这个函数，完成相关的初始化操作。
3. 执行扩展库初始化函数。注意这里的初始化函数和 `DllMain` 是完全不同；`DllMain` 是操作系统来执行的，而初始化函数是 TCL 解释器来调用执行的。这个函数必须在动态库中实现，如果没有的话就会加载失败。初始化函数一般是向解释器注册各类扩展命令或者程序包。

## 初始化函数的名字

一般情况下，扩展库初始化函数的名字由两个因素决定：

1. `load` 命令的参数 `packageName`；
2. `load` 命令的参数 `interp` 是否是一个安全解释器（Safe Interp）；

初始化函数的名字为 `pkg_Init`。其中 `pkg` 是将参数 `packageName` 的第一个字母变成大写，后面所有的字母变成小写后得到。假如 `packageName` 为 `skipList` 或者 `SkipList`，那么对应的初始化函数名字都为 `Skiplist`。

如果省略了 `interp` 参数，那么就默认为调用 `load` 命令的当前解释器。如果 `interp` 解释器是安全解释器，那么初始化函数名字就成了 `pkg_SafeInit`。其中 `pkg` 部分和上面一致。

如果省略了 `packageName` 参数或者它是空字符串怎么办？这个时候 Tcl 解释器会根据参数 `filename` 猜测程序包的名字。在大部分 Unix 系统中，Tcl 会从 `filename` 参数中，去掉所有的路径和目录，去掉后面的文件名后缀，得到文件名。如果文件名是 `lib` 开始，也去掉 `lib`。然后使用剩下文件名中字符下划线来构成 `package` 名字。例如，执行 `load /lib/libxyz4.2.so` 的时候，包的名字为 `xyz`；在 Windows 系统中与此类似。例如我们下面将要实现的动态链接库 `myMath.dll`，初始化函数名字为 `Mymath_Init`。

我们可以在一个库文件中实现多个包，每一个包包含自己特有的命令；在 `load` 的时候只需要给出相应的包名参数即可。

## Windows 下 load 失败的原因

在 Windows 操作系统下，执行 TCL 的 `load` 命令有时候会发生“library not found”的错误，很多读者碰到这个问题后显得束手无策，我以前也经常被其困扰，其实这个问题往往是因为动态链接库的依赖关系导致的。例如：我们在 `C:/tcl/lib/myLib` 目录下存放了三个



动态链接库文件，分别是 a.dll，b.dll 和 c.dll；其中 a.dll 是用来被 load 的库文件，实现了库初始化函数；而 b 和 c 两个库则对 a 提供支撑，实现了其它必须的函数；a 依赖于 b 和 c，也就是说 a 在被加载的时候，操作系统会自动的加载 b 和 c。如果加载 b 或者 c 失败，那么加载 a 的时候就会出现那个恼人的错误。

如果 load a.dll 失败，就需要检查 a.dll 所依赖的所有动态链接库是否都在操作系统的加载搜索路径中。请按照如下的步骤来逐一检查：

1. 看看 a.dll 是否对 tcl 动态链接库的版本有依赖关系。如果 a.dll 是使用 tcl83 的引入库链接的，那么一般情况下是无法用在 tcl84 版本的解释器中；如果一定要用，必须要重新编译链接这个动态链接库。
2. 除了 b 和 c，a.dll 是否还依赖其它非系统库。所谓系统库，就是操作系统提供的动态链接库，比如 user32.dll，kernel.dll，一般而言系统库文件是不会缺失的。如果有其它非系统库文件缺失，那么肯定就无法加载；
3. 到了这一步，说明你已经确认 TCL 版本无误并且没有任何文件缺失。那就可以确认是路径的问题。当 a.dll 被加载的时候，因为 a 依赖于 b 和 c，所以操作系统会首先自动搜寻 b.dll 和 c.dll，如果搜寻不到它们，a.dll 就无法加载，就会报错！那么操作系统怎么会搜寻不到呢？怎么可能搜不到呢？它们不是和 a.dll 在同一个目录下吗？要弄清楚这个问题，有必要解释一下 Windows 操作系统在加载动态链接库的时候，具体的搜寻方式是怎样的！

Windows 操作系统的 API 函数 LoadLibrary 或者 LoadLibraryEx 在执行的时候，按照如下顺序在各个目录中搜寻需要加载的动态链接库：

1. 当前进程的可执行文件所在的目录；
2. 进程的当前目录；我们可以用 pwd 命令来查询获得；
3. 系统目录，可以通过 WIN32API 函数 GetSystemDirectory 查询到；如果是 NT 或者 XP 系统，那么一般就是 C:/windows/system32；
4. 16 位系统目录，这个目录我们可以不用理会；
5. Windows 目录，可以通过 WIN32API 函数 GetWindowsDirectory 查询到；
6. 环境变量 PATH 中列出来的目录；

如果我们的 b.dll 或 c.dll，或者 b 和 c 所依赖的任何一个其它 DLL 文件没有在上面六个地方出现，就会出现错误。了解这个原理就比较好了，解决方法有：

1. 将 b.dll 和 c.dll 放到 TCL 解释器可执行文件所在的目录中去，例如 tclsh.exe 的所在目录为 C:/tcl/bin，可以将 b 和 c 放到这里；不过这样容易把一个完整的程序包搞得太过于分散，文件不够集中；不好维护。
2. 将 b.dll 和 c.dll 转移到 Windows 的系统目录或者 Windows 安装目录下；
3. 将 b.dll 所在的路径添加到 PATH 环境变量中，不过这个方法看起来比较蠢；
4. 如果所有依赖的库文件都在同一个目录下，那么可以修改 load a.dll 这条语句所在的脚本文件。在执行 load 命令之前，首先调用 cd 命令改变进程的当前目录到 a.dll 所在的目录，然后再执行 load，load 结束后再次将当前目录更改回去。我认为这是一个比较聪明的做法。例如 load b.dll 是在 tclIndex 文件，可以这样修改：



```
set auto_index(Max) [subst -nocommands {set t [pwd] } ;#保存当前目录
cd $dir ;#更改当前目录到包所在目录
load [file join $dir myMath.dll] ;#加载文件
cd [set t] ;#回到原来的目录
unset t}]
```

5. 如果保证了上面四点还不行，那就可以肯定的说，是某些动态链接库在执行入口函数 `DllMain` 的时候失败了。比如 `b.dll` 或者 `c.dll` 的 `DllMain` 函数。这是程序的问题，需要调试解决。

怎样查看一个 DLL 依赖哪些其它的动态链接库？VC 提供了两个工具：

1. 在命令提示符下执行 `dumpbin.exe -imports dllname`，就可以看到 `dllname` 这个文件所依赖的所有其它动态链接库的名字了；`dumpbin.exe` 是 VC 提供的一个程序；
2. 运行 `Depends.exe` 程序，会出现一个图形界面，打开我们需要查看的动态链接库，就可以非常直观的看到它所依赖的其它文件。

## 创建 DLL 工程

在 Visual C++ 中创建一个名为 `myMath` 的 Win32 DLL 工程，不需要 MFC。工程目录为 `E:\work\myMath`。我们还需要对工程进行一些设置，打开工程属性设置窗口：

1. 设置附加的头文件目录：我们用到了 `tcl.h` 这个文件，必须告诉编译器其所在路径。在工程属性窗口中，设置附加的 `include` 目录为 “`C:\Tcl\include`”，这是 `ActiveTcl` 安装目录下的 `include` 子目录；
2. 设置附加的库文件目录：link 的需要连接 TCL 的库，所以必须指定链接库所在的目录。这里指定为 “`C:\tcl\lib`”，这是安装目录下的子目录，里面包含了随着 TCL 一起发布的所有 `.lib` 文件；
3. 指定额外的链接库：link 的时候需要连接 `tcl` 的库，否则会连接出错；TCL 的库包含多种形式：动态的，静态的，还有桩形式的。我们这里选择动态引入库文件 `tcl84.lib`。在连接器的附加依赖库这一个选项中输入即可。

完成这几步之后，打开预编译头文件 `stdafx.h`，在最后面加入一行代码：

```
#include "tcl.h"
```

然后选择 “生成解决方案”，如果不出意外，应该能够编译链接成功。我们的主要工作在 `myMath.h` 和 `myMath.cpp` 这两个文件中展开。

## 实现扩展命令函数

这里是 `myMath.h` 文件的内容：

```
001 // myMath.h      : 相关宏和函数的定义
002 // Author        : LeiYuhou
003
004 #ifdef MYMATH_EXPORTS
005 #define MYMATH_API __declspec(dllexport) //这里表示引出
006 #else
007 #define MYMATH_API __declspec(dllimport) //这里表示引入
008 #endif
009
010 extern "C"{
011     MYMATH_API int Mymath_Init(Tcl_Interp* interp); //库文件初始化函数
012 }
```

这里是 myMath.cpp 文件的内容:

```
001 // myMath.cpp : 定义 DLL 应用程序的入口点和 TCL 扩展命令实现函数
002 // Author      : LeiYuhou
003
004 #include "stdafx.h"
005 #include "myMath.h"
006 #include "limits.h"
007
008 BOOL APIENTRY DllMain( HANDLE hModule,
009                       DWORD  ul_reason_for_call,
010                       LPVOID lpReserved
011                       )
012 {
013     switch (ul_reason_for_call)
014     {
015     case DLL_PROCESS_ATTACH:
016     case DLL_THREAD_ATTACH:
017     case DLL_THREAD_DETACH:
018     case DLL_PROCESS_DETACH:
019         break;
020     }
021     return TRUE;
022 }
023
024 //这是命令 max 的实现函数, 求几个整数的最大值
```

```
025 static int _tcl_Max(ClientData clientData,
026                     Tcl_Interp *interp,    //解释器指针
027                     int objc,              //命令和其参数个数
028                     Tcl_Obj *CONST objv[] //命令和参数内容
029                     )
030 {
031     if ( objc<2 )    //判断参数是否正确
032     {
033         Tcl_WrongNumArgs( interp , objc , objv , "val ?val...?" );
034         return TCL_ERROR; //参数个数不正确，必须返回错误
035     }
036
037     //来循环找出其中的最大值
038     long lmax = LONG_MIN;
039     for(int i=1;i<objc;i++)
040     {
041         long l; //保存转换的整数结果
042
043         //从 Tcl_Obj 对象得到 Long 类型
044         int r = Tcl_GetLongFromObj( interp , objv[i] , &l);
045         if ( r==TCL_ERROR ) //转换失败，不能转换为整数
046         {
047             return TCL_ERROR;
048         }
049         if ( l>lmax ) {
050             lmax = l;
051         }
052     }
053
054     //设置命令的返回结果
055     Tcl_Obj *result = Tcl_NewLongObj( lmax );
056     Tcl_SetObjResult( interp , result );
057     return TCL_OK;
058 }
059
060 //TCL 使用 load 加载该动态链接库的时候，会执行该函数进行初始化
061 MYMATH_API int Mymath_Init(Tcl_Interp* interp)
062 {
```

```
063      //向解释器中增加一个 Max 命令
064      Tcl_CreateObjCommand(interp,"Max",_tcl_Max,NULL,NULL);
065
066      return TCL_OK;    //初始化成功才返回 TCL_OK。
067  }
```

其中 myMath.cpp 文件中第 25 行开始定义的函数 `_tcl_Max` 就是命令 `Max` 的实现函数。当 TCL 解释器在执行 `Max` 命令的时候，就会调用这个函数。

## 函数原型

`_tcl_Max` 该函数是基于 TCL 的对象（Object）机制定义的，除了对象机制外，还有一种老的字符串（String）机制。两者差别在于：对象机制是 TCL8.0 新引入的，变量在解释器内部使用对象来保存，而 `String` 则是用字符串来保存；对象机制的速度比字符串机制要快得多。两种不同的机制下，扩展命令实现函数的接口不一样：

```
//这是对象机制下的命令实现函数的原型
typedef int Tcl_ObjCmdProc(ClientData clientData,Tcl_Interp *interp,
                           int objc,
                           Tcl_Obj *CONST objv[]);
//这是字符串机制下的命令实现函数的原型
typedef int Tcl_CmdProc(ClientData clientData, Tcl_Interp *interp,
                        int argc,
                        CONST char *argv[]);
```

它们的差别在于最后一个参数的类型。在 TCL8.4 中，两种接口都支持，但是后者我们不推荐使用。本书都是以对象机制来讲解。所有扩展命令实现函数的原型都是一致的，其参数意义如下：

1. **clientData**: 这是一个指针类型，指向一块用户申请的数据区域，我们一般不用；
2. **interp**: 这是指向解释器的指针，用来标识当前运行该命令的解释器；很多 Tcl 库函数都需要这个指针；
3. **objc**: 这是一个整数类型，表示传递给 `Max` 命令的参数个数，这个数字包括 `Max` 命令字本身；所以实际上是参数个数再加上一；
4. **objv**: 是一个 `Tcl_Obj*` 类型的数组，里面每一个元素都是一个 `Tcl_Obj` 类型的指针，表示一个参数。其中 `objv[0]` 表示命令字本身；

## 校验参数个数

函数首先必须校验传递给命令的参数个数等是否有效。根据 Max 命令的规格，它必须至少带一个参数！所以参数 objc 至少为 2。代码第 31 行就是用来判断参数个数，如果参数个数不正确，我们调用库函数 Tcl\_WrongNumArgs 来设置错误信息，并且马上返回错误代码 TCL\_ERROR；该函数的原型为：

```
void Tcl_WrongNumArgs(Tcl_Interp* interp, int objc, Tcl_Obj* objv[], char* message)
```

参数 interp 表示解释器指针，我们直接将函数 \_tcl\_Max 的参数 interp 传递给它即可；后面的 objc、objv 和 message 用来构造错误信息。例如：

objv 包含了如下信息：foo bar；参数 objc 为 1；message 为“fileName count”；那么产生的错误信息为：“wrong # args: should be "foo fileName count"”；如果参数 objc 为 2，产生的错误则为：“wrong # args: should be "foo bar fileName count"”。一般情况下 objc 为 1，就是只选择命令字部分；少数情况下可以为 2，例如 string 这样一个命令下有几个子命令的情况。

这个函数会根据上面描述的规则产生错误信息，并且放入 interp 指向的解释器的命令结果中；然后将该信息设置为异常信息的内容。比如我们的 \_tcl\_Max 函数中，当参数个数不正确的时候，产生的错误信息就是：

```
wrong # args: should be "Max val ?val...?"
```

## 解析参数值

完成参数个数校验后，我们开始查找最大值了。这里 Tcl\_Obj 有一点类似于 VC++ 中的 variant 数据类型，我们需要从 Tcl\_Obj 中解析出我们想要的数据类型。这里我们期望参数类型为 long 类型，所以使用了库函数 Tcl\_GetLongFromObj，其原型如下：

```
int Tcl_GetLongFromObj(Tcl_Interp*interp, Tcl_Obj* objPtr, long* longPtr)
```

参数 interp 我就不多说了；objPtr 就是保存参数对象的指针，这里是直接从 \_tcl\_Max 外部传递进来的；longPtr 是一个 long 变量的地址，用来存放得到的 long 类型的值；

如果这个函数正确的解析出了一个 long 类型的值，那么就返回 TCL\_OK，并且将结果存放在 longPtr 指向的地址；否则就返回 TCL\_ERROR，表示无法解析成 long，此时它也会自动生成一个错误信息，将其设置为解释器 interp 此时命令的结果和错误信息。所以我们一定要检查该函数的返回值。\_tcl\_Max 函数中，当它返回 TCL\_ERROR 的时候，我们只是简单的返回 TCL\_ERROR。

除了 Tcl\_GetLongFromObj 函数之外，TCL 库函数还提供了其它类型的转换函数，包括 String，Boolean，Double 和 List 等。

## 返回值

这里的返回值包含两个层面：一是函数 `_tcl_Max` 的返回值，其次是命令 `Max` 的返回值。函数本身的返回值比较简单，下面是几个简单的原则：

1. 一般的正常情况下就返回 `TCL_OK`，表示命令成功执行了；
2. 出现了错误，那么就返回 `TCL_ERROR`；这时解释器中的命令会抛出异常；
3. 其它的 `TCL_RETURN`、`TCL_BREAK` 和 `TCL_CONTINUE` 很少用到，除非你正在编写一个自己的控制命令，比如前面我们实现的 `do` 循环；

命令的返回值需要调用 `Tcl` 库函数来设置，`_tcl_Max` 函数需要返回的是一个 `long` 类型，所以在 55 行首先调用 `Tcl_NewLongObj` 函数创建了一个 `Tcl_Obj` 对象，并且用 `lmax` 来对其进行初始化，该对象内部保留的是一个 `Long` 类型的数值；紧接着调用 `Tcl_SetObjResult` 来将这个对象设置为命令执行结果即可。

命令执行错误（函数返回 `TCL_ERROR`）的时候，也需要设置返回值。大家可能会问了，你的 `_tcl_Max` 在返回 `TCL_ERROR` 之前可是什么都没做啊？没错，因为这里用到的几个 `Tcl` 库函数都已经自动的帮助我设置命令返回值了，比如 `Tcl_WrongNumArgs` 等。如果我们不用 `Tcl_WrongNumArgs` 函数，自己动手该怎么编写呢？我将参数个数不正确情况下的处理代码改写了一下，请看：

```
031     if ( objc<2 )
032     {
033 #if 0
034         Tcl_WrongNumArgs( interp , objc , objv , "val ?val...?" );
035 #else
036         //自己来设置错误信息和命令返回值
037         const char* errorInfo = "wrong # args : should be \"Max val ?val...?\"";
038         Tcl_AddErrorInfo( interp , errorInfo );
039         Tcl_Obj* result = Tcl_NewStringObj( errorInfo , -1 );
040         Tcl_SetObjResult( interp , result );
041 #endif
042         return TCL_ERROR; //参数个数不正确，必须返回错误
043     }
```

预编译指定 `#if 0` 部分是原来的代码，只用了一个 `Tcl_WrongNumArgs` 函数就搞定；`#else` 部分则是自己编写的处理代码。第 38 行调用函数 `Tcl_AddErrorInfo` 设置 `TCL` 解释器的异常错误信息，随后的 39 和 40 行创建一个字符串对象并且将其设置为命令返回值。

两种方法，显然采用 `Tcl_WrongNumArgs` 要快捷得多。但是如果在有些地方没有相应的库函数来帮我们设置错误信息和返回值的话，就需要我们自己动手了。

## 库初始化函数

刚才的 `_tcl_Max` 函数是扩展命令的实现函数，光有它还不够，还需要将这个扩展命令安装到解释器中，这个工作在初始化函数中完成，在我们工程中函数名字为 `Mymath_Init`。为什么是这个名字？如果你真忘了，请参考前面介绍的“load 命令”中初始化函数的命名。

函数的原型是固定的，如下：

```
typedef int Tcl_PackageInitProc(Tcl_Interp *interp);
```

参数 `interp` 是 TCL 解释器传入的指针；如果初始化成功那么就返回 `TCL_OK`，否则就要返回 `TCL_ERROR` 表示初始化失败。

在本函数中主要完成命令和程序包注册的任务。在例子中调用 `Tcl_CreateObjCommand` 函数来将 `Max` 命令注册到解释器当中。该函数的原型如下：

```
Tcl_Command Tcl_CreateObjCommand (Tcl_Interp* interp,  
    char* cmdName, Tcl_ObjCmdProc* proc,  
    ClientData clientData, Tcl_CmdDeleteProc deleteProc)
```

参数虽然多，但是并不复杂：

1. `interp` 我就不多说了，表示解释器指针；
2. `cmdName` 表示需要注册的命令名字，是字符串类型，它的值就是以后在脚本中调用的命令名字，这里我们传入的是“Max”；
3. `proc` 则是我们实现的命令扩展函数的指针，我们可以直接把函数名 `_tcl_Max` 传给它；它必须和 `cmdName` 对应，也就是说，不要张冠李戴了；
4. `clientData` 参数指向我们自己定义的一块数据，究竟怎么用？用户说了算；这里我们给它传入参数 `NULL`；
5. `deleteProc` 指向一个函数指针，当命令 `cmdName` 被删除的时候就会调用这个函数。这个函数的原型如下：`typedef void Tcl_CmdDeleteProc(ClientData clientData);`。参数 `clientData` 就是在调用 `Tcl_CreateObjCommand` 时传入的 `clientData`；

函数 `Tcl_CreateObjCommand` 在创建命令 `cmdName` 之前，会首先检查该命令是否已经存在了，如果存在就删除它。

动态链接库必须输出（Export）初始化函数，否则 TCL 解释器无法执行它。在 VC 中我们通过 `__declspec(dllexport)` 来修饰该函数，表示该函数在编译连接的时候，会被输出。扩展命令的实现函数 `_tcl_Max` 就不用输出了。

## 安装库并且测试

编译链接整个工程，在 `E:\work\myMath\Debug` 目录下生成 `myMath.dll`。我们需要测试



一下，看看是否实现了命令 Max 的设计规格。启动 TCL 解释器：

```
E:\Work\myMath\Debug>tclsh
% pwd                ;#查看当前目录
E:/Work/myMath/Debug
% load myMath.dll     ;#加载 myMath.dll 文件
% Max                 ;#不带参数调用 Max 命令，应该返回错误
wrong # args : should be "Max val ?val...?"
% Max 1 2 3           ;#应该返回 3
3
% Max 1 2 0x34        ;#应该返回 0x34 的值
52
% Max 30 tt           ;#参数 tt 不是整数，会返回错误
expected integer but got "tt"
```

看来一切都符合我们期望。但是每次使用前都要 load 一下就比较麻烦了。能不能让解释器自动来加载呢？答案是肯定的，我们在以前就介绍过 TCL 命令自动加载的机制。下面我们直接介绍操作步骤，想了解操作原理的请把书向前翻。

1. 设置系统环境变量 TCLLIBPATH 的值为 E:/work/myMath/Debug，也就是动态链接库所在的目录；
2. 在这个目录下创建一个名为 tclIndex 的文件，文件内容如下，注意它开头部分的一堆注释可千万不要删除或修改，否则 TCL 解释器不会加载这个文件：

```
# Tcl autoload index file, version 2.0
# This file is generated by the "auto_mkindex" command
# and sourced to set up indexing information for one or
# more commands. Typically each line is a command that
# sets an element in the auto_index array, where the
# element name is the name of a command and the value is
# a script that loads the command.

set auto_index(Max) [subst -nocommands {set t [pwd]
    cd $dir
    load [file join $dir myMath.dll]
    cd [set t]
    unset t}]
```

3. 现在可以让 TCL 自动加载了，测试一下：

```
E:\Work\myMath\Debug>set tcl      =>查看一下环境变量
TCLLIBPATH=E:/Work/myMath/Debug
```



```
E:\Work\myMath\Debug>tclsh
% Max      ;#直接执行 Max 命令，让 TCL 去自动加载
wrong # args : should be "Max val ?val...?"
% Max 1 2
2
```

一切都很顺利，不是吗？

## 编写自己的数学函数

除了编写扩展命令之外，我们还可以在动态链接库中编写自己的数学函数，这些函数能够作为参数出现在 `expr` 命令中。下面我们来实现一个求两个整数的最大公约数的数学函数，这个函数是依据欧里几德算法而得到，其原理如下：

如果用  $\text{gcd}(a,b)$  表示  $a$  和  $b$  的最大公约数，那么  $\text{gcd}(a,b) = \text{gcd}(b, a \bmod b)$ ；

证明这个定理不是本章节目的。我们还是在刚才的 `myMath` 工程中来实现一个 `gcd` 函数，完成求两个整数的最大公约数的数学函数。

## 分析例子代码

代码在 `myMath.cpp` 中完成，节选相关部分如下：

```
068 //函数 gcd 采用递归算法，计算 a 和 b 的最大公约数
069 static long gcd(long a , long b)
070 {
071     if ( a<b ) {
072         long tmp=a ; a=b ; b = tmp; //交换两个参数
073     }
074     if ( (a%b)==0 ) {
075         return b;
076     }
077     return gcd( b , a%b ); //递归
078 }
079
080 //Tcl_gcd，这是 TCL 的数学计算函数 gcd 的实现函数
081 static int Tcl_gcd( ClientData clientData, //自定义数据
082                    Tcl_Interp *interp,    //解释器指针
```

```
083         Tcl_Value *args,          //参数列表
084         Tcl_Value *resultPtr      //返回值的指针
085     )
086 {
087     long a = args[0].intValue;
088     long b = args[1].intValue;
089
090     //判断参数是否合法
091     if ( a<=0 || b<=0 ) {
092         Tcl_Obj* result = Tcl_NewStringObj(
093             "domain error: argument not in valid range" , -1 );
094         Tcl_SetObjResult( interp , result );
095         return TCL_ERROR;
096     }
097     resultPtr->type = TCL_INT; //结果类型
098     resultPtr->intValue = gcd( a , b ); //结果的值
099     return TCL_OK;
100 }
101
102 //TCL 使用 load 加载该动态链接库的时候，会执行该函数进行初始化
103 MYMATH_API int Mymath_Init(Tcl_Interp* interp)
104 {
105     //向解释器中增加一个 Max 命令
106     Tcl_CreateObjCommand(interp,"Max" , _tcl_Max , NULL , NULL);
107
108     //向解释器中注册数学函数 gcd
109     Tcl_ValueType argsType[] = {TCL_INT , TCL_INT }; //参数类型数组
110     Tcl_CreateMathFunc(interp , "gcd" , 2 , argsType , Tcl_gcd , NULL);
111
112     return TCL_OK;    //初始化成功才返回 TCL_OK。
113 }
```

68 行开始的 C 函数 gcd 采用递归方法实现，可能有人很瞧不起递归，但是我要说的是递归是一种非常优秀的思想，它是很多函数式程序设计语言的重要基石，例如 ML, Haskell 等。gcd 只是简单的返回参数 a 和 b 的最大公约数。

函数 Tcl\_gcd 则是我们要实现的 TCL 数学函数 gcd 的实现函数，其原型是固定的，带有四个参数，参数含义在代码注释中都已写清楚。其中参数 args 是一个数组的首地址，数组元素的个数就是 TCL 数学函数 gcd 的参数个数，可以看到，TCL 数学函数的参数个数

是固定的，不存在可变参数这么一说；每一个元素的类型 `Tcl_Value` 定义如下：

```
typedef struct Tcl_Value {  
    Tcl_ValueType type;    //值的类型  
    long intValue;         //type = TCL_INT 时有效，整数值  
    double doubleValue;    //type = TCL_DOUBLE 时有效，双精度浮点  
    Tcl_WideInt wideValue; //type=TCL_WIDE_INT 时有效，宽整数  
} Tcl_Value;
```

根据域 `type` 的取值不同，`Tcl_Value` 的实际值在下面三个字段中保存。这里为什么不将后面的三个字段定义成一个 `union` 呢？老实说，我也不知道为什么。最后的参数 `resultPtr` 是指向结果值的指针，类型和 `args` 一样。

读者会问了：函数 `Tcl_gcd` 中，你为什么没有判断 `args` 中各个元素的 `type`，就直接从 `intValue` 中取值了呢？答案在第 109 行，我们注册函数的时候指明了它只接受 `TCL_INT` 类型的参数，所以 TCL 解释器在执行 `gcd` 函数之前，会先进行类型转换，确保传入 `Tcl_gcd` 的参数是我们要求的类型。

随后的第 91 行，我们判断参数范围，如果不符合要求，我们就设置解释器的结果为一个错误信息，然后返回 `TCL_ERROR`。第 97 行，我们设置结果类型为 `TCL_INT`，结果为调用 C 函数 `gcd` 的结果，参数 `resultPtr` 指向的结构地址已经由 TCL 解释器申请到了，我们没有必要再次申请（你申请了也没有用，不仅带不出这个函数，还会造成内存泄漏）。

实现了 `Tcl_gcd` 还不够，必须向解释器注册，这在库初始化函数 `Mymath_Init` 中的第 109 行完成，首先定义了一个 `Tcl_ValueType` 类型的数组，它包含两个元素，分别用来描述 TCL 数学函数 `gcd` 的两个参数可以接受的数值类型。类型除了 `TCL_INT` 等三种之外，还可以是 `TCL_EITHER`，表示可以接受任意一种类型（如果是它，那我们就必须在 `Tcl_gcd` 里面来判断 `args` 各个元素的具体类型了）。然后调用 `Tcl_CreateMathFunc` 来进行注册，函数原型如下：

```
void Tcl_CreateMathFunc(Tcl_interp* interp,    //解释器指针  
    const char* name, //TCL 数学函数的名字，它出现在 expr 命令参数中  
    int numArgs,      //函数的参数个数，也是 argTypes 数组元素的个数  
    Tcl_ValueType* argTypes, //包含 numArgs 个元素的数组；  
    Tcl_MathProc proc,    //命令 name 对应的执行函数，这里是 Tcl_gcd  
    ClientData clientData) //客户数据
```

该函数向解释器中注册 TCL 数学函数，参数 `name` 给出函数名，`numArgs` 指定了参数个数，`argTypes` 是一个数组首地址，描述每一个参数的类型，`proc` 指定执行数学计算的函数指针。如果 `name` 指定的数学函数在解释器 `interp` 中已经存在，那么会首先删除原来的数学函数，然后再注册。

`Tcl_MathProc` 是一个函数类型的定义，如下：

```
typedef int Tcl_MathProc( ClientData clientData,
```

```
Tcl_Interp *interp,  
Tcl_Value *args,  
Tcl_Value *resultPtr);
```

TCL 中所有自定义的数学函数都必须按照这个原型定义，包括我们定义的 `Tcl_gcd` 函数。具体的参数含义我们已经解释过了，这里不再重复。

## 测试验证

重新编译工程 `myMath` 生成动态链接库 `myMath.dll`。然后启动 TCL 解释器进行测试：

```
E:\Work\myMath>tclsh           =>启动解释器  
% expr gcd(50,20)              ;#用 expr 命令来调用 gcd，会出错。为什么？  
unknown math function "gcd"  
% Max 1 2                      ;#调用 Max 命令，自动将 myMath.dll 加载进来  
2  
% expr gcd(50,20)              ;#再次计算 50 和 20 的最大公约数，成功  
10  
% expr gcd(50)                 ;#只用一个参数调用看看，应该报错  
too few arguments for math function  
% expr gcd(50,20,30)           ;#用三个参数来调用，也应该报错  
too many arguments for math function  
% expr gcd(50,20.23)           ;#用一个浮点数 20.23 来测试看看，会自动转换为 20  
10  
% expr gcd(50,-20.23)          ;#用一个负数来测试，应该报错；  
domain error: argument not in valid range
```

差不多了，一切正常！为什么在调用 `Max` 命令之前运算 `gcd` 函数会报错呢？因为这个时候动态链接库 `myMath.dll` 还没有被加载到解释器中，数学函数 `gcd` 还不存在。

## 实现为 package

前面我们使用的是 `tclIndex` 文件和 `auto_index` 机制将 `myMath.dll` 自动加载到解释器中，如果我们想把 `myMath.dll` 作为程序包 `MyMath` 来加载该怎么办呢？

第一种方法：将 `tclIndex` 文件换成 `pkgIndex.tcl` 文件，内容如下：

```
001 package ifneeded MyMath 1.0 [subst {  
002     load [file join $dir mymath.dll] ;#加载 DLL 文件
```

```
003         package provide MyMath 1.0           ;#声明提供 package 成功
004     }]
```

此时必须把 `tclIndex` 文件改名或者删除。我们试着测试一下：

```
E:\Work\myMath>tclsh
% Max           ;#直接调用 Max 命令，会失败；返回无效命令
invalid command name "Max"
% package require MyMath           ;#将包 MyMath 引入
1.0
% Max           ;#再次调用 Max，可以调用；返回参数错误信息
wrong # args : should be "Max val ?val...?"
% expr gcd(100,36)
4
```

一切都在期望之中。上面的 `pkgIndex.tcl` 文件的第 3 行非常重要，不能够省略！否则引入 `MyMath` 程序包会失败。如果一定要省略这一行，那就需要修改 `myMath` 的实现。我们只需要修改扩展包的初始化函数 `Mymath_Init` 即可，如下：

```
116     Tcl_CreateMathFunc(interp , "gcd" , 2 , argsType , Tcl_gcd , NULL);
117
118     if( TCL_ERROR==Tcl_PkgProvide( interp , "MyMath" , "1.0") )
119     {
120         return TCL_ERROR;
121     }
122
123     return TCL_OK;    //初始化成功才返回 TCL_OK。
124 }
```

我们在文件 `myMath.cpp` 的第 118 行，也就是初始化函数 `Mymath_Init` 的最后调用了函数 `Tcl_PkgProvide`，向解释器表明提供了 `MyMath 1.0` 版本。该函数调用和原来 `pkgIndex.tcl` 文件中的 `package provide MyMath 1.0` 的功能是完全一样的。

可见用 C 编写 `package` 也是非常方便的。

## 摆脱 TCL 版本的限制

迄今为止，我们的扩展库运行良好。但是 TCL 在不断发展进步，在我写这本书的时候，Tcl8.5 已经进入到 8.5a3 版本，里面又加入了诸多诱人的增强功能。动态链接库 myMath 中的 Max 和 gcd 能不能运行在 tcl8.5 版本中呢？我下载 Tcl8.5a3 源代码并且 make 成功后，启动 tcl85 解释器执行 Max 命令，结果出现了“程序异常，需要关闭”的对话框。为什么？

因为 myMath.dll 是采用 tcl84.lib 进行链接的，事实上 tcl84.lib 只是一个引入库，真正 TCL 的代码都在 tcl84.dll 这个动态链接库中。当 myMath.dll 被 load 的时候，Windows 操作系统首先判断 tcl84.dll 是不是已经存在于当前进程的地址空间中，如果不存在，那么操作系统就会去自动的搜寻并且加载 tcl84.dll。而在 tcl85 解释器运行时，是无法找到 tcl84.dll 这个文件的，所以会报错。即使找到了，tcl84.dll 和 tcl85.dll 能否共存于一个进程中，也是未知数。

怎么解决这个问题？最简单的方法就是使用 tcl85.lib 来重新编译连接 myMath.dll 这个文件。不过这样一来，每次 TCL 版本升级都需要重新编译 myMath.dll，就太繁琐了。能不能够不重新编译 myMath.dll，就可以直接在其它版本的解释器中使用呢？当然有！解决问题的钥匙就是 TCL 的桩机制（Stubs mechanism）。

下面我们就使用 TCL 桩来对我们的 myMath.dll 进行改进，使之一次编译，到处加载。具体操作步骤如下：

1. 修改扩展库的初始化函数 Mymath\_Init，在其内部调用任何 TCL 库函数的方法之前，必须首先调用 Tcl\_InitStubs 函数来初始化桩机制。代码如下：

```
103 MYMATH_API int Mymath_Init(Tcl_Interp* interp)
104 {
105     //初始化 Tcl 的桩
106     const char* lpVer = Tcl_InitStubs( interp , "8.3" , 0 );
107     if ( lpVer==NULL ) {
108         return TCL_ERROR;    //桩机制初始化失败
109     }
110
111     //向解释器中增加一个 Max 命令
112     .....
```

2. 定义宏 USE\_TCL\_STUBS，这个宏应该对整个工程中所有的源文件都起作用。一般是在工程的预定义宏中来定义它。
3. 更换链接库，原来我们连接是 tcl84.lib，现在要换成 tclstub84.lib。

完成上面三步后就可以重新编译链接，生成新的 myMath.dll。我们来测试一下它是否能够在两个不同的 TCL 版本中运行，过程如下：

```
C:\>set tcl
TCLLIBPATH=E:/Work/myMath/Debug          #环境变量 TCLLIBPATH

C:\>tclsh84
% set tcl_version          ;#检查当前解释器的版本, 为 8.4
8.4
% Max                      ;#调用 Max 失败, 但加载 myMath.dll 扩展库文件成功
wrong # args : should be "Max val ?val...?"
% expr gcd(100,90)
10
% exit                    ;#退出解释器

C:\>TCL85\bin\tclsh85.exe          ;#启动 tcl85 版本的解释器
% set tcl_version              ;#检查当前解释器的版本
8.5
% Max                          ;#调用 Max, 加载 myMath.dll 扩展库, 成功
wrong # args : should be "Max val ?val...?"
% expr gcd(100,90)             ;#计算 gcd 函数, 也成功
10
% exit
```

是不是感觉很爽？奥妙都在宏 `USE_TCL_STUBS` 和函数 `Tcl_InitStubs` 中，我们先看看这个函数的原型：

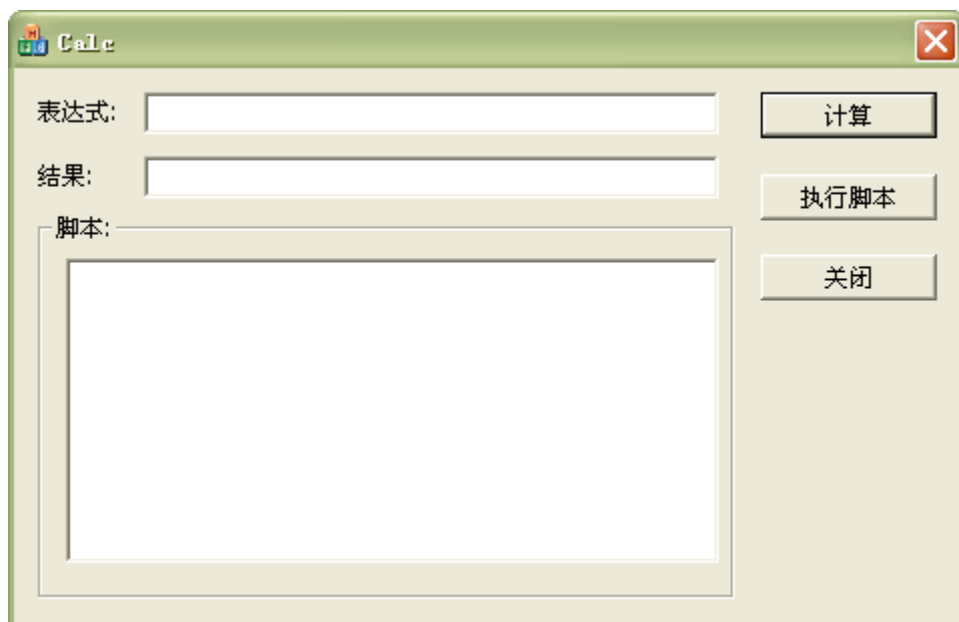
```
CONST char* Tcl_InitStubs( Tcl_Interp* interp , CONST char* version , int exact )
```

第二个参数是一个字符串，用来指定 TCL 解释器版本；第三个参数用来描述是否必须精确满足 `version` 指定的版本；如果 `exact` 为 1，那么只有和 `version` 完全一致的版本才是可以接受的；如果为 0，那么比 `version` 更新的版本也是可以接受的，只要它的主版本一致。我们的例子中，`version="8.3"`，`exact=0`，那么所有 `>=8.3` 并且 `<9.0` 的版本都是可以接受的，但是 9.0 或者 9.1 或者 8.2 的解释器就不行了。如果 `exact=1`，那么只有在 TCL8.3 中才能够加载成功。

如果桩机制初始化成功，函数就返回一个描述当前解释器版本的字符串；如果初始化失败，那么就返回 `NULL`。

## 解释器嵌入

我们通过一个简单例子来说明如何编写应用程序并且嵌入 Tcl 解释器。程序使用 Visual C++ 开发，主窗口是一个对话框，界面如下：



从上往下依次三个编辑框，分别映射成对话框类 CCalcDlg 的三个 CString 类型的成员变量 m\_strExpr、m\_strResult 和 m\_strScript，分别表示需要被计算的数学表达式，表达式的计算结果，以及需要被执行脚本内容。三个按钮：“计算”用来计算数学表达式，“执行脚本”则用来在解释器中执行 m\_strScript 的内容，“关闭”用来退出程序。

因为要嵌入 TCL 解释器，所以需要对这个工程的配置选项做一些修改，这些修改和编写 TCL 扩展库类似：

1. 将 C:\tcl\include 加入到附加的编译头文件目录中；
2. 将 C:\tcl\lib 加入到附加的引入库目录中；

## 初始化 TCL

将 TCL 解释器嵌入到应用程序中的时候，首先要做的事情就是考虑链接和加载 TCL 库。链接方式分成两种：静态链接和动态链接。

1. 静态链接最简单，直接链接库文件 tcl84s.lib 即可。
2. 动态链接方式需要链接 DLL 库文件，又可以细分成两类：使用和不使用桩机制：
  - a) 使用桩机制可以动态的寻找和链接满足一定条件的 tclxx.dll 文件，这样就可



以避免依赖某一个固定的版本；

- b) 如果不使用桩机制，那么应用程序一旦编译完成，它运行的时候必须依赖某一个确定的动态链接库版本；

我们先看看 Calc 应用程序的源文件 CalcDlg.cpp 中初始化 TCL 的代码：

```
075 #ifdef USE_TCL_STUBS
076 /*****
077 /*如果定义了宏 USE_TCL_STUBS */
078 /*使用动态加载和 Stub 机制来初始化 Tcl 系统，创建解释器 */
079 /*****/
080 static Tcl_Interp* InitialTcl()
081 {
082     Tcl_Interp* pInterp = NULL;
083
084     //定义 Tcl_CreateInterp 函数原型
085     typedef Tcl_Interp *(*LPFN_CREATEINTERP)();
086
087     HINSTANCE hTcl = NULL;
088     for( int minor=4 ; hTcl==NULL&&minor<=5 ; minor++ )
089     {
090         CHAR buf[128];
091         sprintf( buf , "TCL8%d.dll" , minor );
092         hTcl = LoadLibrary( buf ); //依次寻找并且加载动态链接库
093     }
094
095     //如果加载动态链接库成功
096     if (hTcl != NULL)
097     {
098         //得到 Tcl_CreateInterp 函数
099         LPFN_CREATEINTERP lpfn = (LPFN_CREATEINTERP)
100             GetProcAddress(hTcl, "Tcl_CreateInterp");
101         if (lpfn != NULL)
102         {
103             pInterp = lpfn(); //创建解释器
104             if (pInterp != NULL)
105             {
106                 //初始化桩
107                 Tcl_InitStubs(pInterp, "8.2", 0);
```

```
108
109             //初始化各子系统
110             Tcl_FindExecutable(GetCommandLine());
111
112             //注册 memory 等命令
113             Tcl_InitMemory(pInterp);
114
115             //执行 init.tcl 文件
116             Tcl_Init(pInterp);
117         }
118     }
119 }
120     return pInterp;
121 }
122 #else
123 /*****
124 /* 如果没有定义 USE_TCL_STUBS, 那么就不使用 Stubs 机制来初始化*/
125 *****/
126 static Tcl_Interp* InitialTcl()
127 {
128     Tcl_Interp* pInterp = NULL;
129
130     Tcl_FindExecutable( GetCommandLine() );//初始化各子系统
131     pInterp = Tcl_CreateInterp();        //创建解释器
132     Tcl_InitMemory(pInterp);            //注册 memory 等命令
133     Tcl_Init(pInterp);                  //初始化
134
135     return pInterp;
136 }
137 #endif
138
139 #ifdef USE_TCL_STUBS
140 #pragma comment(lib, "tclstub84.lib") //链接桩
141 #else
142 #pragma comment(lib, "tcl84.lib")     //链接引入库
143 #endif
144
145 // CCalcDlg 消息处理程序
```

```
146 BOOL CCalcDlg::OnInitDialog()
147 {
148     CDialog::OnInitDialog();
149     SetIcon(m_hIcon, TRUE);        //设置大图标
150     SetIcon(m_hIcon, FALSE);     //设置小图标
151
152     m_pInterp = InitialTcl();      //初始化系统
153     return TRUE;
154 }
```

第 80 行以及第 126 行分别定义了函数 `InitialTcl` 的两个实现，用宏 `USE_TCL_STUBS` 来对它们进行预编译选择。如果定义了宏 `USE_TCL_STUBS`，那么说明工程采用了 TCL 桩机制来进行动态加载链接；否则采用简单的动态链接。`InitialTcl` 除了初始化 TCL 之外，还创建了一个 TCL 解释器，将其指针返回。

在第 139 行我们根据宏开关 `USE_TCL_STUBS`，来决定链接哪一个库文件，这里使用了 Visual C++ 的编译指令：`#pragma comment`。如果打开了宏开关，则链接 `tclstub84.lib`；否则就链接引入库 `tcl84.lib`。

在对话框初始化函数 `OnInitDialog` 中 152 行，调用 `InitialTcl` 函数来初始化 TCL 系统。这样 TCL 解释器就嵌入到了应用程序 `Calc` 当中，可以执行 TCL 脚本了。

## 简单动态链接

文件 `CalcDlg.cpp` 第 126 行定义了初始化 TCL 系统的行数 `InitialTCL`。它首先调用了函数 `Tcl_FindExecutable`，其格式如下：

```
void Tcl_FindExecutable(argv0)
```

该函数非常重要。在程序初始化的时候，应该在所有其它 TCL 库函数被调用之前，首先调用这个函数，它用来设置应用程序的路径名，参数 `argv0` 用来指定本程序的名字，一般就是函数 `main` 的参数 `argv[0]`。TCL 命令 `info nameofexecutable` 返回的应用程序名字就是该函数设置下去的。

设置程序路径只是这个函数的功能之一，其最大的用处在于初始化 TCL 的各个子系统，包括 IO，编码和名字空间等等。如果不初始化，应用程序会出现各种错误。

紧接着调用 `Tcl_CreateInterp` 创建一个 TCL 解释器对象，该函数无参数，直接返回一个 `Tcl_Interp*` 类型的指针，对这个指针我们并不陌生，前面介绍的 `Tcl_CreateCommand` 等等函数都会用到它。

随后的函数 `Tcl_InitMemory` 向解释器注册两个命令：`memory` 和 `checkmem`。它们可以用来跟踪调试应用程序的内存使用情况，例如找出内存泄漏。不调用该函数也没有关系。

随后调用 `Tcl_Init` 来对刚创建的解析器进行初始化。该函数根据一套特定的算法来查

找 `init.tcl` 文件，然后执行 `init.tcl` 文件对解释器进行初始化。这套算法我们后面介绍。

`Tcl_Init` 成功执行后，整个 TCL 系统以及 TCL 解释器就初始化成功，我们就可以针对 TCL 解释器来进行常见的操作了，比如增加自己定义的扩展命令等。

## 使用桩机制动态链接

桩机制动态加载的方式稍微复杂一些，分成如下几步：

1. 加载 TCL 的动态链接库；
2. 初始化 TCL 桩机制；
3. 初始化 TCL 系统，创建解释器，初始化解释器；

在文件 `CalcDlg.cpp` 的第 87 至 93 行，用来完成第一个步骤。这里通过 `for` 循环来依次加载 `tcl84.dll` 和 `tcl85.dll`，只要加载成功就马上退出循环。在 Windows 操作系统中通过 `Win32 API` 函数 `LoadLibrary` 来加载动态链接库，加载成功就返回动态库的句柄，如果加载失败就返回 `NULL`。

第 99 行，通过 `Win32 API` 函数 `GetProcAddress` 从动态库中找到 `Tcl_CreateInterp` 函数的地址，放到函数指针类型变量 `lpfn` 中，然后调用该函数创建解释器。

第 107 行，调用 `Tcl_InitStubs` 初始化 TCL 桩机制。随后的三个函数调用来对 TCL 系统和创建的解释器进行初始化。这和刚才介绍的简单动态链接没有什么不同。

`Tcl_Init` 初始化解释器之后，我们就可以在后面直接使用这个解释器指针了。

## 初始化 TCL 解释器

刚才提到，函数 `Tcl_Init` 用来对 TCL 解释器进行初始化。其函数原型如下：

```
int Tcl_Init(Tcl_Interp* pInterp)
```

该函数的本质就是：按照特定顺序搜寻文件 `init.tcl`，然后在解释器 `pInterp` 中执行该脚本。`init.tcl` 文件的内容我们前面有介绍，它会初始化 `auto_path` 变量，创建 `unknown` 过程等，该文件非常重要，我们在将 TCL 嵌入到自己应用程序中的时候，也要有一个 `init.tcl` 文件，如果没有什么特殊的需求，可以将 ActiveTCL 自带的 `init.tcl` 复制过来。

## tcl\_library 和 tcl\_libPath

这是 TCL 中的两个全局变量，很多人不明白其用途，尤其是后者。先认识一下它们：

```
E:\Work\script>set TCL_LIBRARY          =>环境变量 TCL_LIBRARY
TCL_LIBRARY=E:/AAA/BBB/CCC
```

```
E:\Work\script>tclsh                                =>启动 Tcl 解释器
% foreach m $tcl_libPath {puts $m}                    =>打印变量 tcl_libPath 的内容
E:/AAA/BBB/CCC                                        =>和 TCL_LIBRARY 有关
E:/AAA/BBB/tcl8.4
C:/Tcl/lib/tcl8.4
C:/Tcl/lib/tcl8.4
C:/lib/tcl8.4
C:/Tcl/library
C:/library
C:/tcl8.4.6/library
% puts "tcl_library = $tcl_library ;\[info library\]=\[info library\]"
tcl_library = C:/Tcl/lib/tcl8.4 ;\[info library\]=C:/Tcl/lib/tcl8.4
```

可以看到, `tcl_library` 是一个目录, 其值和`[info library]`返回值一致; 但是 `tcl_libPath` 就看起来很奇怪了: 它是一个列表, 其值和环境变量 `TCL_LIBRARY` 有一些关系, 还包含有两个重复的元素, 还有一堆根本就不存在的目录。它是用来做什么的? 别急, 看完后面的内容就明白了。

## 搜索 `init.tcl`

回到原来的话题, 有一个问题: 这个 `init.tcl` 文件应该放到什么地方? 回答是: 放在`[info library]`返回的目录中? 问题依然存在, 我怎么知道 `info library` 会返回什么呢?

这就需要了解一下 `Tcl_Init` 函数是怎样搜寻到 `init.tcl` 文件的。

刚才介绍的函数 `Tcl_FindExecutable` 是用来初始化 TCL 各个子系统的, 除此之外, 它还会调用一个内部函数 `TclpInitLibraryPath`, 该函数会根据当前应用程序所在的目录、TCL 动态链接库所在目录, 以及环境变量 `TCL_LIBRARY` 等, 计算出一个包含许多目录名的列表, 并将这个列表保存在一个内部变量中。这个算法如下:

1. 首先将 `TCL_LIBRARY` 所指定的路径加入到列表;  
将`<TCL_LIBRARY>././tclx.y`所代表的路径加入到列表;
2. 将`<dldir>././lib/tclx.y`加入到列表末尾; `dldir` 是 `tclxy.dll` 所在目录;
3. 将`<bindir>././lib/tclx.y`加入到列表末尾; `bindir` 是应用程序所在目录;
4. 将`<bindir>././lib/clx.y`加入到列表末尾;
5. 将`<bindir>././library`加入到列表末尾;
6. 将`<bindir>././library`加入到列表末尾;
7. 将`<bindir>././tclx.y.z/library`加入到列表末尾;

这里的 `xyz` 表示版本号, 它编译时候确定; 例如我机器上就是 `tcl8.4.6`, 那么 `x`、`y` 和 `z` 分别等于 8, 4 和 6。

当 Tcl\_Init 被调用的时候，它会将该内部变量的值设置到 TCL 变量 tcl\_libPath 中，然后执行一个内部定义的脚本字符串，其内容如下：

```
01 if {[info proc tclInit]==""} {
02     proc tclInit {} {
03         global tcl_libPath tcl_library errorInfo
04         global env tclDefaultLibrary
05         rename tclInit {}
06         set errors {}
07         set dirs {}
08
09         if {[info exists tcl_library]} {
10             lappend dirs $tcl_library
11         } else {
12             if {[info exists env(TCL_LIBRARY)]} {
13                 lappend dirs $env(TCL_LIBRARY)
14             }
15             catch {
16                 lappend dirs $tclDefaultLibrary
17                 unset tclDefaultLibrary
18             }
19             set dirs [concat $dirs $tcl_libPath]
20         }
21         foreach i $dirs {
22             set tcl_library $i
23             set tclfile [file join $i init.tcl]
24             if {[file exists $tclfile]} {
25                 if {[catch {uplevel #0 [list source $tclfile]} msg]} {
26                     return
27                 } else {
28                     append errors "$tclfile: $msg\n$errorInfo"
29                 }
30             }
31         }
32
33         set msg "Can't find a usable init.tcl in the following directories: "
34         append msg "    $dirs\n"
35         append msg "$errors\n"
```

```
36         append msg "This probably means that Tcl wasn't installed properly."
37         error $msg
38     }
39 }
40 tclInit
```

这段代码是从 TCL 源程序中抠出来的，它搜索 init.tcl 文件并且执行它。其逻辑如下：

1. 首先判断变量 `tcl_library` 是否存在。如果不存在，就把环境变量 `TCL_LIBRARY` 表示的路径，变量 `tclDefaultLibrary` 以及 `tcl_libPath` 合并到列表 `dirs` 中。如果存在，那么将 `dirs` 变量设置为 `tcl_library` 的值。
2. 然后依次针对列表 `dirs` 中的每一个目录，在其中寻找 `init.tcl`。如果找到就执行它。如果执行 `init.tcl` 没有错误，那么就马上返回。如果出错，那么就继续寻找并且执行，直到找到一个执行无错的 `init.tcl` 后退出，或者在最后抛出异常。也就是说，我们可以在这里的每一个目录中放一个 `init.tcl` 文件，即使有几个 `init.tcl` 有错误也没有关系，只要有一个是正确的，那么就能够用它来将解释器初始化成功。

如果是第一次调用 `Tcl_Init`，此时变量 `tcl_library` 还不存在，所以总是从一堆目录中来搜索 `init.tcl` 文件，并且在找到之后将目录名赋值给 `tcl_library`。以后调用时就不用搜索了。变量 `tcl_library` 的值就是命令 `[info library]` 的返回值。从搜索顺序可以看到，搜索 `init.tcl` 的顺序如下：

1. 环境变量 `TCL_LIBRARY` 的优先级是最高的，最先被搜索；
2. 其次是相对动态链接库的目录： `<Tclxy.dll>/../lib/tclx.y`，这里 `x` 和 `y` 是版本号，例如我机器上 TCL 动态链接库文件在 `C:/tcl/bin/` 中，那么会搜索 `C:/tcl/lib/tcl8.4`；
3. 再次就是相对可执行文件所在的目录： `<bindir>/../lib/tclx.y`。对于 `ActiveTcl` 而言，这个目录和上一个目录是重复的，因为 `tclsh84.exe` 和 `tcl84.dll` 在同一个目录中。
4. 最后就是列表变量 `tcl_libPath` 中剩余的那些目录了。

正是因为上面的这种处理机制，变量 `tcl_libPath` 中才存在重复的、或者是完全不存在的目录。TCL 这样搜索一大堆目录的目的，可能就是：宁可错搜一千，不可漏掉一个。

我们正在开发的 Calc 应用程序中，`tcl_libPath` 的值和说明如下：

```
c:/tcl/lib/tcl8.4           =>根据 DLL 所在路径转换而来
e:/Work/myMath/Calc/lib/tcl8.4 =>根据 Exe 所在路径转换而来
e:/Work/myMath/lib/tcl8.4
e:/Work/myMath/Calc/library
e:/Work/myMath/library
e:/Work/myMath/tcl8.4.6/library
e:/Work/tcl8.4.6/library
```

在我系统的环境变量 `PATH` 中，包含了 `C:/tcl/bin` 这个目录，它就是 `ActiveTcl` 的可执行文件和 `Dll` 所在目录。当 `Calc` 启动加载 `tcl8x.dll` 的时候，操作系统会从 `PATH` 中来搜索

这个 DLL, 就会在目录 C:/tcl/bin 中找到动态链接库 tcl84.dll, 所以会将 C:/tcl/bin/../lib/tcl8.4 加入到 tcl\_libPath 中, 也就是 c:/tcl/lib/tcl8.4。恰好这个目录中存在一个正确的 init.tcl, 所以 tcl\_library 的值也是 c:/tcl/lib/tcl8.4。这也印证了一点: 全局变量 tcl\_library 的值是由 init.tcl 文件所在目录决定的。

## 执行脚本、计算表达式

TCL 系统和解释器的初始化之后, 就可以使用解释器来完成一些工作了, 下面我们为按钮“计算”和“执行脚本”添加事件响应函数:

```
156 //按下按钮 "计算" 后的处理函数
157 void CCalcDlg::OnBnClickedOk()
158 {
159     UpdateData( );
160     if ( m_strExpr.IsEmpty() ) {
161         MessageBox( "请输入一个正确的数学表达式" );
162         return;
163     }
164
165     if ( m_pInterp!=NULL ) {
166         Tcl_ExprString( m_pInterp , (LPCSTR)m_strExpr );//计算表达式
167         m_strResult = Tcl_GetStringResult( m_pInterp );//输出结果
168         Tcl_ResetResult( m_pInterp );
169         UpdateData( FALSE );
170     }
171 }
172
173 //按下按钮 "执行脚本" 后的处理函数
174 void CCalcDlg::OnBnClickedEvalscript()
175 {
176     UpdateData( TRUE );
177     if ( m_pInterp !=NULL ) {
178         Tcl_Preserve((ClientData) m_pInterp);
179         Tcl_Eval( m_pInterp , m_strScript ); //执行脚本
180         TRACE("%s\n" , Tcl_GetStringResult(m_pInterp) );
181         Tcl_ResetResult( m_pInterp );//清除命令结果
182         if ( Tcl_InterpDeleted(m_pInterp) ) {
```



```
183         Tcl_DeleteInterp(m_pInterp);
184         Tcl_Release((ClientData)m_pInterp);
185         m_pInterp = NULL;
186     } else {
187         Tcl_Release((ClientData)m_pInterp);
188     }
189 }
190 }
```

第 166 行函数 `Tcl_ExprString` 把用来进行数学计算，其原型如下：

```
int Tcl_ExprString( Tcl_Interp* interp , const char* expr );
```

字符串 `expr` 被当成一个数学表达式来进行计算，计算结果保存在解释器的结果缓冲区中，可以通过 `Tcl_GetStringResult` 来得到计算结果。除了这个函数之外，还存在系列计算表达式的函数，这些函数计算的结果类型存在一些差异：

```
int Tcl_ExprLong(Tcl_Interp* interp, const char* string, long* longPtr)
int Tcl_ExprDouble(Tcl_Interp* interp, const char* string, double* doublePtr)
int Tcl_ExprBoolean(Tcl_Interp* interp, const char* string, int* booleanPtr)
```

这三个函数的执行结果类型分别为：`Long`、`Double` 和 `Boolean` 类型，结果存放在最后一个参数所指向的内存地址中。如果某表达式执行的最终结果是浮点型，例如 `expr sqrt(2.0)` 的结果是 `1.414...`，但是采用 `Tcl_ExprLong` 函数来计算它，那么函数会自动的进行类型转换，`longPtr` 中的结果是 `1`。

如果结果是一个不能转换为数字的字符串，那么使用 `Tcl_ExprLong` 的时候就会出现错误。这个时候就只能让 `Tcl_ExprString` 出场了。

第 179 行函数 `Tcl_Eval` 将字符串参数作为一个脚本来执行，原型如下：

```
int Tcl_Eval( Tcl_Interp* interp , const char* script );
```

该函数将参数 `script` 作为脚本在解释器 `interp` 中执行，结果放在 `interp` 的结果缓冲区中。除了这个函数之外，还有系列 `Tcl_Eval*` 函数来执行脚本，例如：

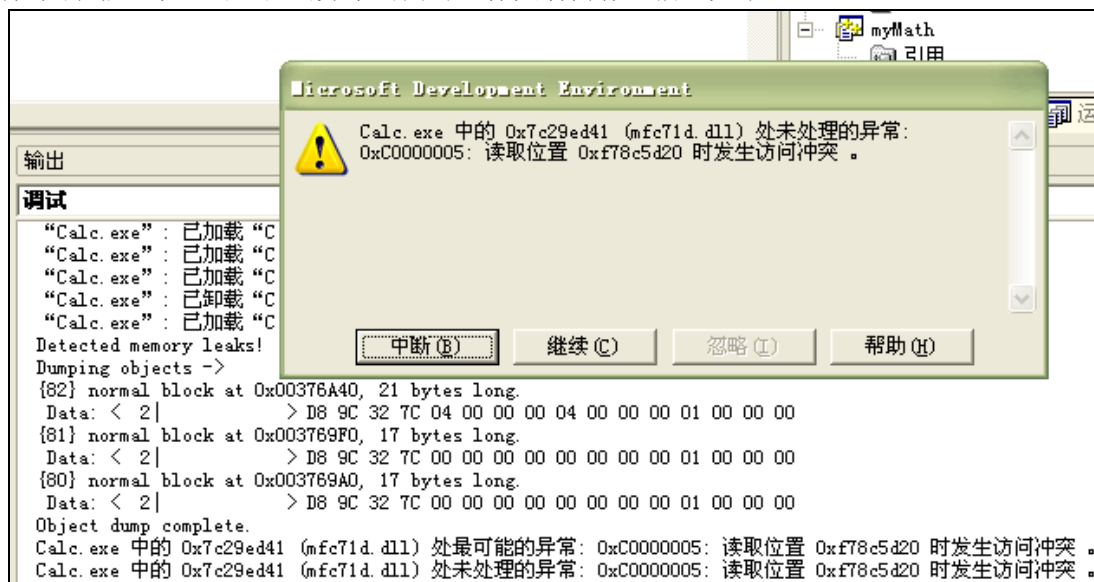
```
int Tcl_EvalFile(Tcl_Interp* interp, const char* fileName)
```

这个函数完成 `Tcl` 命令 `source` 的功能，执行 `fileName` 所表示的脚本文件。`Tcl_EvalEx` 函数可以给出更多的细节控制，它可以设置是否将脚本在 `Global Level` 上执行。具体请参考 `TCL` 联机帮助手册。

在 180 行使用 `Tcl_GetStringResult` 获取结果，通过 `TRACE` 宏打印在调试窗口中。

## 安全退出

程序还存在一个问题：如果我们在 Calc 的命令窗口中输入 exit 命令并且执行该脚本，会发现程序非正常退出，抛出异常的同时还伴随着内存泄漏，如下：



这是一个程序 Bug，那该怎么办？总不能够在用户手册中禁止用户执行 exit 命令吧？其实比较简单，我们将 exit 命令重写即可。代码做出如下修改：

首先我们在 OnInitDialog 函数前面定义自己的 exit 扩展命令，如下：

```

146 static int _my_exitProc(ClientData clientData,
147                          Tcl_Interp *interp,
148                          int objc,
149                          Tcl_Obj *CONST objv[])
150 {
151     int exitCode(0);    //退出代码
152     if ( objc==2 &&
153         Tcl_GetIntFromObj(interp,objv[1],&exitCode)==TCL_ERROR ) {
154         return TCL_ERROR;
155     } else if (objc>2) {
156         //参数个数错误
157         Tcl_WrongNumArgs( interp, 1 , objv , " ?exitcode?" );
158         return TCL_ERROR;
159     }
160

```

```
161    //给主窗口发送 WM_DESTROY 消息
162    AfxGetMainWnd()->DestroyWindow();
163
164    return TCL_OK;
165 }
```

函数 `_my_exitProc` 首先判断 `exit` 命令参数是否合法。然后调用 `DestroyWindows` 给主窗口发送 `WM_DESTROY` 消息，退出整个程序。

`CCalcDlg::OnInitDialog` 函数也需要做相应的修改，如下：

```
168 BOOL CCalcDlg::OnInitDialog()
169 {
170     //.....
171     m_pInterp = InitialTcl();    //初始化系统
172     //将原来的 exit 命令改名为 __exit__
173     Tcl_Eval( m_pInterp , "rename exit __exit__ ;");
174     //向解释器增加新的 exit 命令
175     Tcl_CreateObjCommand(m_pInterp, "exit" , _my_exitProc, NULL, NULL);
176     return TRUE;
177 }
```

在 176 行将 `exit` 改名，178 行将新的 `exit` 命令增加到解释器中。这样执行 `exit` 命令的时候就按照我们的意志来执行退出操作。

这样还没完，执行脚本的过程也需要一定的改动，如下：

```
200 void CCalcDlg::OnBnClickedEvalscript()
201 {
202     UpdateData( TRUE );
203     if ( m_pInterp != NULL ) {
204         Tcl_Preserve((ClientData) m_pInterp);
205         Tcl_Eval( m_pInterp , m_strScript ); //执行脚本
206         if ( m_pInterp == NULL ) { //解释器已经被 Destroy
207             return;
208         }
209         if ( Tcl_InterpDeleted(m_pInterp) ) {
210             Tcl_DeleteInterp(m_pInterp);
211             Tcl_Release((ClientData)m_pInterp);
212             Tcl_Finalize(); //清除
213             m_pInterp = NULL;
214         } else {
```

```
215         TRACE("%s\n", Tcl_GetStringResult(m_pInterp) );
216         Tcl_ResetResult( m_pInterp ); //清除命令结果
217         Tcl_Release((ClientData)m_pInterp);
218     }
219 }
220 }
```

最大的不同在于，`Tcl_Eval` 返回后，马上查看 `m_pInterp` 是不是被置为 `NULL`，以及是否被删除了。只有在没有被删除的时候，才能够从其中获取计算结果。

经过上面的一番改造，我们的程序在执行 `exit` 命令的时候就不会出现异常了。

## 嵌入解释器的用途

将 TCL 解释器嵌入到自己软件系统之后，就为系统提供了一个功能扩展的接口。我们可以使用 VB 为 Microsoft Visual Studio 开发一些嵌入式小工具，可以使用 lisp 语言为 EMacs 编写工具来增强其功能，满足自己特定的需要。同样，嵌入了 TCL 之后，我们就可以通过 TCL 语言来编写脚本，对原来的系统进行功能扩展和升级。这可以满足各种用户的特定需求，让比较高级的用户能够自由扩展。

一般的步骤如下：

1. 将 TCL 解释器嵌入到系统中，可以参考上面介绍的方法；
2. 将系统中各种功能模块抽象成 TCL 命令接口，并且将这些接口实现为 TCL 解释器的扩展命令；
3. 将这些接口文档化，提供给最终用户；
4. 提供一种编码扩展规范，比如所有的用户定制功能脚本，必须按照特定接口编写等；
5. 提供一种用户扩展机制，比如菜单项，工具条上按钮或者快捷键，让用户可以通过增加菜单条，并且将菜单条和用户自己编写的脚本联系起来；

这里就不给大家举例子了。除了 Emacs，Visual Studio 等工具之外，还有很多软件都具有类似的功能，比如 Source Insight 等等。大家可以参考其实现方法。

## 高级数据结构

Tcl 是脚本语言，字符串类型和数组、列表一般情况下能够应付普通编程。但是表达复杂数据时也有力不从心的时候，碰到这种情况，往往各位 TCL 高手就不管三七二十一，马上撸起袖子自己动手写一个库出来。作者以前就干过几次这样的事情，写完之后往往会非常得意，但是随后就发现在 tclLib 中已经实现了这些数据结构，而且还比我写得好，此时懊恼之情难以言表。

我们现在就来介绍 TclLib 提供的各种高级数据结构，避免各位高手犯类似的错误。

## 列表（list）

针对列表，TclX 已经提供了相关的扩展命令，前面已有介绍。但是 TclLib 中仍然提供了一堆扩展命令来操作列表，可见列表无疑是 TCL 中最具有生命力的数据结构。这里的列表命令通过包 struct 来提供，所以要使用它们之前必须执行 `package require struct`。

## 普通命令

这部分命令用来完成常见的列表操作，包含如下几个命令：

命令格式	功能说明
<code>::struct::list reverse sequence</code>	将一个列表颠倒后作为结果返回
<code>::struct::list assign sequence varname ?varname?...</code>	将一个列表中的元素分别赋值给后面的变量，并且返回列表剩下的内容
<code>::struct::list flatten ?-full? ?-? sequence</code>	将列表中嵌套的列表元素展开
<code>::struct::list shift listvar</code>	返回 listvar 的头一个元素并从中删除
<code>::struct::list iota n</code>	返回[0..n) 列表。和 Python 语言中的 range 类似
<code>::struct::list equal a b</code>	判断 a 和 b 两个列表是否相等
<code>::struct::list repeat size element1 ?element2 element3...?</code>	将元素 element1...elementn 重复 size 次后作为列表返回
<code>::struct::list repeatn value size...</code>	将 value 重复 size 次后作为列表返回

这些命令都比较简单，有些和 TclX 还有重复。请看下面的例子：

```
% ::struct::list reverse {1 2 3 4 5}      ;#倒转列表中的所有元素
5 4 3 2 1
% ::struct::list assign {1 2 3 4 5} a1 a2   ;#列表分解到变量 a1 a2，返回剩余元素
```

```
3 4 5
% puts "$a1 , $a2"      ;#输出 a1 和 a2 两个变量
1 , 2
% ::struct::list flatten {1 {2 3} 4 5}    ;#展开元素中的列表
1 2 3 4 5
% ::struct::list flatten {1 {2 3} {{4 5} 6} }    ;#只展开一级子列表
1 2 3 {4 5} 6
% ::struct::list flatten -full {1 {2 3} {{4 5} 6} }    ;#-full 参数表示全部展开
1 2 3 4 5 6
% set a {1 2 3 4 5} ; ::struct::list shift a    ;#列表 a 左移一个元素并且返回其值
1
% puts $a          ;#列表 a 的第一个元素已经被删除
2 3 4 5
% ::struct::list iota 10    ;#返回一个包含 0——9 这几个元素的列表
0 1 2 3 4 5 6 7 8 9
% ::struct::list equal {1 2 3 4 5} {1 2 3 {4 5}}    ;#判断两个列表是否相等
0
% ::struct::list equal {1 2 3 {4 5}} {1 2 3 {4 5}}    ;#判断两个列表是否相等
1
% ::struct::list repeat 3 1    ;#将数字 1 重复 3 次得到列表
1 1 1
% ::struct::list repeat 3 1 2    ;#将 1 和 2 都依次重复 3 次，得到列表
1 2 1 2 1 2
% ::struct::list repeat 3 {1 2}    ;#将列表{1 2}重复 3 次
{1 2} {1 2} {1 2}
% ::struct::list repeatn 100 3    ;#将 100 重复 3 次。repeatn 和 repeat 不同
100 100 100
% ::struct::list repeatn 100 3 4    ;#将 100 先重复 3 次，然后再次重复 4 次
{100 100 100} {100 100 100} {100 100 100} {100 100 100}
% ::struct::list repeatn 100 {3 4}    ;#和上面的命令类似
{100 100 100} {100 100 100} {100 100 100} {100 100 100}
```

这些命令能够给我们提供一些方便，比如 `iota` 命令，它可以完成类似 Python 语言中的 `range` 命令，返回一个 `[0..n)` 这样的列表。事实上在操作列表数据结构方面，Python 比 Tcl 要方便一些，例如 Python 中的 `range` 函数的原型：`range(start,[stop],[step])`，这也比 TCL 中的 `itoa` 要强大得多。

## 高阶函数

所谓高阶函数是来自于“函数式语言”中的一个概念，并且在 Python 这样的脚本中也得到了实现，例如 Python 中的 `map` 函数。在 TCL 中也可以非常方便的实现。所谓高阶函数，可以简单的解释为使用其它函数作为其参数的函数。这里实现的高阶函数有：

列表高阶函数名	功能说明
<code>::struct::list map sequence cmdprefix</code>	针对列表 <code>sequence</code> 中的每一个元素，来调用 <code>cmdprefix</code> 命令，返回所有结果组成的列表。
<code>::struct::list filter sequence cmdprefix</code>	针对列表 <code>sequence</code> 中的每一个元素，调用命令 <code>cmdprefix</code> ，返回结果为 1 的元素列表。实际上是一个过滤作用。
<code>::struct::list split sequence cmdprefix ?passVar failVar?</code>	也是过滤器，只不过将 <code>cmdprefix</code> 执行后返回 1 和 0 的元素分别在两个列表中返回。
<code>::struct::list fold sequence initialValue cmdprefix</code>	将列表 <code>sequence</code> 中的元素通过 <code>cmdprefix</code> 调用，折叠到一个值。

下面分别讲解。

## 列表映射（map）命令

上面的说明看起来很别扭，还是来看具体的 `map`（映射）例子：计算列表中每一个元素的平方值，结果以列表中返回。一般的方法如下（其实很简单）：

```
01 package require struct
02
03 #计算平方的函数
04 proc sqr {m} { return [expr $m*$m]}
05
06 set a [struct::list iota 10] ;#列表 0-9
07 puts "list a = $a"
08
09 #传统的方法，使用 for 循环
10 foreach m $a {
11     lappend b [sqr $m]
12 }
13 puts "list b = $b" ;#输出结果
```

使用高阶函数 `struct::list map` 来实现的方式如下：

```
15 #新方法, 调用 struct::list map 命令
16 puts "list b" = [struct::list map $a {sqr }]"
```

可以看到, `struct::list map` 命令的第一个参数是需要迭代执行的列表 `$a`, 第二个参数是需要执行的命令以及其参数。`map` 在执行的时候, 取出列表中的每一个元素的值, 分别附加到命令后面, 然后执行这个命令, 将命令结果作为其映射后的结果。这里输出的结果是:

```
list a = 0 1 2 3 4 5 6 7 8 9
list b = 0 1 4 9 16 25 36 49 64 81
list b1 = 0 1 4 9 16 25 36 49 64 81
```

刚才提到了: `map` 的第二个参数不仅仅可以是一个命令, 还可以是命令及其参数序列。下面是另外两个例子

```
18 #为列表中的每一个元素加上 5 后返回
19 puts [struct::list map $a {expr 5+}]
20
21 set a [struct::list repeat 3 {1 2}] ;#列表 a = {{1 2} {1 2} {1 2}}
22 #取出列表 a 中每一个子列表的第 1 个元素
23 puts [struct::list map $a [proc _t {l m} {return [lindex $m $l]} ;return "_t 0" ]]
```

第 19 行, `map` 的第二个参数是字符串 `"expr 5+"`, `map` 执行时直接把每一个元素值追加到其后接着执行。第 23 行比较复杂一些, 首先定义了一个 `_t` 命令, 然后返回 `"_t 0"`, 其目的就是取出列表的第 0 个元素。上面代码执行结果如下:

```
5 6 7 8 9 10 11 12 13 14
1 1 1
```

## 列表过滤 (filter) 命令

如何从一个列表中挑出所有满足一定条件的元素? `filter` 命令可以很方便的做到。请看下面的例子:

```
% package require struct
2.0
% set a [struct::list iota 20] ;#生成列表 a = [0-20)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
% proc odd {x} {return [expr {$x % 2}]} ;#函数 odd 判断是否是奇数
% proc even {x} {return [expr {$x%2==0}]} ;#函数 even 判断是否是偶数
% puts [struct::list filter $a odd] ;#过滤出列表 a 中的所有奇数
1 3 5 7 9 11 13 15 17 19
```



```
% puts [struct::list filter $a even]    ;#过滤出列表中的所有偶数
0 2 4 6 8 10 12 14 16 18
% puts [struct::list filter $a "expr 10<="]    ;#过滤出列表中所有>=10 的数字
10 11 12 13 14 15 16 17 18 19
% puts [struct::list filter $a "expr 10>"]    ;#过滤出列表中所有<10 的数字
0 1 2 3 4 5 6 7 8 9
```

`filter` 命令就是将列表元素带入到后面的命令中来执行，结果不为 `false` 的元素就放入结果列表中返回。

## 列表拆分 (`split`) 命令

`split` 命令和 `filter` 命令类似，不同之处在于将过滤通过和没有通过的元素都返回；它返回的是一个包含了两个子列表的列表，分别表示过滤通过和没有通过的元素集合。请看下面的例子：

```
% puts [struct::list split $a "expr 10>"]    ;#返回列表，一个<10，一个>=10
{0 1 2 3 4 5 6 7 8 9} {10 11 12 13 14 15 16 17 18 19}
% puts [struct::list split $a odd]    ;#分别返回奇数和偶数两个列表
{1 3 5 7 9 11 13 15 17 19} {0 2 4 6 8 10 12 14 16 18}
% struct::list split $a odd oddlist evenlist    ;#返回两个列表的长度
10 10
% puts "oddlist = $oddlist ; evenlist = $evenlist"    ;#列表内容在 oddlist 和 evenlist 中
oddlist = 1 3 5 7 9 11 13 15 17 19 ; evenlist = 0 2 4 6 8 10 12 14 16 18
```

`struct::list split` 命令有两种格式，一个是只带有列表和命令两个参数，这种情况下命令返回两个列表，分别表示过滤通过 (`Pass`) 和未通过 (`NoPass`) 的元素列表。还有一种情况则是后面多了两个参数，分别表示两个变量名字，用来存在 `Pass` 和 `NoPass` 的元素列表，这种情况下，`split` 命令返回两个数字，分别表示两个结果列表的长度。

## 列表折叠 (`fold`) 命令

`fold` 翻译成折叠并不一定合适，其含义就是将一个列表中的各个元素通过一定规则的验算最终归纳到一个值。请看例子：

```
01 package require struct
02
03 set a [struct::list iota 101]    ;#列表 a=[0,100]
```

```

04 proc + {i j} {return [expr $i+$j]} ;#命令+, 两个数的和
05 proc * {i j} {return [expr $i*$j]} ;#命令*, 两个数的积
06
07 #输出列表 a 中所有元素的和
08 puts "1+2+3+...100 = [struct::list fold $a 0 +]"
09 #输出 10 的阶乘
10 puts "10! = [struct::list fold {1 2 3 4 5 6 7 8 9 10} 1 *]"

```

struct::list fold 命令带有三个参数，如下：

struct::list fold sequence initValue cmdprefix

如果 sequence 中有 n 个元素，那么 cmdprefix 就会被执行 n 次；每次执行的时候，都会向 cmdprefix 后面增加两个值作为参数。以上面的“+”命令为例，其执行序列如下：

1. + \$initValue [lindex \$a 0] =>result
2. + \$result [lindex \$a 1] =>result
3. ....
- n + \$result [lindex \$a end] =>result

第一次执行的时候，使用参数 initValue 和第 0 个元素作为参数来调用 cmdprefix，结果放到内部变量 result 中；第二次，则使用第 1 个元素和 result 来调用，结果再次放到 result 中，这样直到最后一个元素。最后一次调用的结果就是整个 fold 命令的结果。

## 离散对象池（discrete items pool）

现在很多编程体系中出现了“线程池”以及“连接池”等这样的概念，其实现现实中类似的概念也是非常多的：比如停车场中停车位，DHCP 协议中可以用来分配的 IP 地址，图书馆中可以出借的书籍等。这些概念具有共同的特点：

1. 项目是离散的，项目之间没有任何关系；
2. 项目的总数是固定的，全部集中在某一个池子中；
3. 池子中的每个项目要么处于 free（空闲）状态，要么处于 allocated（被占用）状态；
4. 每一个项目可以被申请使用；被申请了之后除非被释放，否则不能被再次申请；

TclLib 提供了一个抽象数据结构 pool 来描述这种概念。它提供的命令如下：

命令语法格式	说明
::struct::pool ?poolName? ?maxsize?	创建一个池子对象，maxsize 指定大小
poolName add itemName1 ?itemName2...?	向池子中加入离散的对象，可以一次加入多个对象
poolName clear ?-force?	删除所有池子中的对象；
poolName destroy ?-force?	删除整个池子，包括其中的对象
poolName info type ?arg?	查询相关池子的相关信息

poolName maxsize ?maxsize?	查询池子的大小
poolName release itemName	将某一项释放，放回池中供下次申请
poolName remove itemName ?-force?	删除池子中的某一项，-force 选项强制删除
poolName request itemVar ?options?	向池子请求一个空闲的对象，成功返回 1， 申请不到则返回 0；

假如我们要使用 pool 来模拟一个停车场，这里用一个数组来描述每一个停车位的详细信息，然后使用 pool 来模拟车位使用情况。请看下面的代码：

```

052 array set parkInfo {
053     1 {size=3*1.8 , Underground 1 , for BMW}
054     2 {size=5*2 ,Underground 2 , for Benz}
055     3 {size=3*1.8 , Underground 1 ,for Toyota}
056     4 {size=3*2 , Underground 3 , for XiaLi}
057 }
058
059 proc CreateParkPool {} {
060     global parkInfo
061     ::struct::pool park 30      ;#最大 30 个车位
062     eval park add [array names parkInfo]  ;#目前加入所有的 4 个车位
063     puts "Free Pool : [park info freeitems]"
064 }
065 CreateParkPool

```

上面代码执行之后，一个名字为 park 的 pool 就创建了，并且其中加入了 4 个可用的车位。下面我们到交互式环境中执行如下命令：

```

% park request tom      ;#向 park 申请一个停车位，申请到的名字放入变量 tom 中
1
% set tom              ;#request 返回 1 表示成功，tom 的值为 4，表示申请到的是 4 号位
4
% park request mike    ;#再次申请，结果放入变量 mike 中
1
% set mike              ;#申请成功，这次申请到的是 1 号位
1
% park info freeitems  ;#查看现在还空闲的停车位，剩下 2 和 3
2 3
% park info allocstate ;#查看所有停车位的申请情况，返回列表，-1 表示空闲
4 dummyID 1 dummyID 2 -1 3 -1
% park request jack -prefer 1 ;#选项 prefer 1，表示希望申请 1 号停车位

```

```

0
% set jack    ;#request 返回 0，表示申请失败！因为你需要的 1 号已经被占用了
can't read "jack": no such variable
% park remove 4    ;#删除 4 号停车位，结果失败：因为 4 号处于 alloced 状态
Can't remove `4' because it is still allocated.
% park remove 4 -force    ;#加上选项-force，强制删除。
% park info allocstate    ;#再次查看申请状况，4 号已经消失了
1 dummyID 2 -1 3 -1
% park request jack -prefer 2 -allocID jack ;#选项-allocID 用来指定申请的 ID
1
% park info allocstate    ;#再次查看申请状况，可以看到 2 号被 jack 申请了
1 dummyID 2 jack 3 -1

```

从上面例子中可以看到，在调用 `request` 命令申请项目的时候可以有两个选项：

1. `-prefer item`：指出希望申请哪一个项目。如果该项目处于 `free` 状态，那么就会成功，返回 1（成功）；否则就返回 -1（失败）。比如你的车是加长的大奔驰，只有 2 号停车位才能够放下，那么就可以指定 `-prefer 2` 来申请该车位。如果没有指定选项 `-prefer`，那么就随即分配一个空闲的项目；
2. `-allocID allocID`：用来指定申请 ID，它可以使任意不是 -1 的字符串。如果没有该选项那么被申请项的 ID 会自动成为“DummyID”。

也可以看到，通过 `parkName info type args` 能够查询池子的相关信息：

type args 语法格式	说明
<code>allocID itemName</code>	查看 <code>itemName</code> 的 <code>allocID</code> 。如果在当初申请该项目的时候指定了选项 <code>-allocID</code> ，那么返回该选项，否则就是默认的 <code>DummyID</code>
<code>allitems</code>	返回池中所有的项目名字，结果是一个列表
<code>allocstate</code>	返回池中所有项目的申请情况，结果是类似 <code>array get</code> 的列表
<code>cursize</code>	返回池中当前所有项目的个数
<code>freeitems</code>	返回池中当前处于 <code>free</code> 状态的项目名字的列表
<code>maxsize</code>	返回池子的最大大小

请看下面的例子：

```

% park info allocstate
1 dummyID 2 jack 3 -1
% puts "[park info allocID 1], [park info allocID 2], [park info allocID 3]"
dummyID , jack , -1
% park info freeitems
3
% park info maxsize
30

```

```
% park add 4 5 6
% park info freeitems
3 4 5 6
```

## 队列（queue）

队列是一种简单的数据结构，实现了先进先出（FIFO）的功能，元素从队列尾部进入队列，从队列头部出队列。tcllib 中的 `struct::queue` 实现了队列的功能，它提供的命令如下：

命令语法格式	说明
<code>struct::queue ?queueName?</code>	创建一个名为 <code>queueName</code> 的队列；如果省略参数，那么命令自动的给队列起一个名字；队列创建后，实际上创建了一个名为 <code>queueName</code> 的 TCL 命令。
<code>queueName destroy</code>	删除名字为 <code>queueName</code> 的队列；
<code>queueName put item ?item ...?</code>	将 <code>item</code> 等数据项依次放入到队列的末尾；
<code>queueName get ?count?</code>	取出队列头部的 <code>count</code> 个元素并且返回，默认为 1 个；并且从队列中删除这些元素；
<code>queueName peek ?count?</code>	取出队列头部的 <code>count</code> 个元素，但是不从队列中删除它们；默认取出 1 个；
<code>queueName clear</code>	清空队列，删除所有的元素；
<code>queueName size</code>	返回队列中的元素的个数；

我们来看看下面的例子：

```
% package require struct      ;#必须首先将包 struct 引入进来
2.1
% ::struct::queue q          ;#创建名字为 q 的队列
::q
% for {set i 0} {$i<10} {incr i} {q put $i} ;#将 0-9 这 10 个元素依次放入队列中
% q peek 10                  ;#取出但是不删除
0 1 2 3 4 5 6 7 8 9
% q size                     ;#元素格式还是 10 个
10
% q get 2                    ;#取出两个元素，返回列表{0 1}
0 1
% q size                     ;#查看元素格式，剩下 8 个
8
% q get 7                    ;#再次取出 7 个元素，以列表的形式返回{2-8}
2 3 4 5 6 7 8
```

```
% q get          ;#再次取出一个，返回元素 9。注意返回的不是列表!!
9
% q get 3         ;#列表已经为空，再次 get 会抛出异常
insufficient items in queue to fill request
% q clear         ;#清空队列，删除所有的元素。但是队列依然存在。
% q size
0
% q destroy       ;#删除整个队列
```

这里提供的队列使用起来非常简单。前面我们在讲解 Tclx 的时候，提到了几个命令能够将普通的 TCL 列表模拟成队列，包括 lvarpush 和 lvarpop 等命令！下面我们就算作心血来潮，简单比较一下它们的性能，请看代码：

```
001 package require Tclx
002 package require struct
003
004 #测试 TclLib 中的 queue 的性能
005 proc testQueue {} {
006     set i 1000
007     struct::queue q1 ;#创建队列 q1
008
009     #将 1000 个元素放入到队列 q1 末尾
010     puts "Queue in tcllib.put -> [time {q1 put [incr i]} 1000]"
011     #依次从队列 q1 中取出这 1000 个元素
012     puts "Queue in tcllib.get -> [time {q1 get 1} 1000]"
013     q1 destroy
014 }
015
016 #测试 Tclx 中的列表实现队列的性能
017 proc testQueueTclx {} {
018     set l [list]
019     set i 1000
020
021     #lvarpush list item len，就是将元素放入列表末尾
022     puts "Queue in tclx .put -> [time {lvarpush l [incr i] len} 1000]"
023
024     #lvar pop list 就是将元素从列表头取出来
025     puts "Queue in tclx .get -> [time {lvarpop l} 1000]"
026 }
```

027

028 testQueue

029 testQueueTclx

作者的系统是运行在 PII 400M 的 Windows XP。执行结果如下：

```
Queue in tcllib.put -> 109 microseconds per iteration
```

```
Queue in tcllib.get -> 189 microseconds per iteration
```

```
Queue in tclx .put -> 43 microseconds per iteration
```

```
Queue in tclx .get -> 16 microseconds per iteration
```

看来使用列表要比 TclLib 的 queue 要快出很多倍！尤其是从队列中取元素的操作，Tclx 的 lvarpop 比 queue 的 get 要快出 10 倍以上。不过这并不妨碍我们使用 struct::queue，因为它的好处就是：写出的程序比较直观，易懂。

## 优先级队列（prioQueue）

优先级队列也是一种队列，元素从尾部进去，头部出来，但是与普通队列的不同之处在于：它并不完全遵守先进先出（FIFO）的规则。放入队列的每一个元素都有一个优先级，优先级相同的元素，遵循 FIFO 的规则，但是优先级高的元素可以插队，插入到所有优先级比它低的元素前面。

TclLib 提供的优先级队列的相关命令如下：

命令语法格式	功能说明
<code>::struct::prioqueue ?-ascii -dictionary -integer -real? ?prioqueueName?</code>	创建名字为 prioqueueName 的优先级队列，并且指定优先级比较的方式。默认的比较方式是 -integer。
<code>prioqueueName clear</code>	删除队列中的所有元素，保留队列
<code>prioqueueName destroy</code>	删除整个队列
<code>prioqueueName get ?count?</code>	取出头部的 count 个元素，默认为 1
<code>prioqueueName peek ?count?</code>	取出头部的 count 个元素，默认为 1；但是不从队列中删除它们
<code>prioqueueName peekpriority ?count?</code>	取出队列头部的 count 个元素的优先级，默认为 1 个。
<code>prioqueueName put item prio ?item prio ...?</code>	将后面的 item 根据各自的优先级，插入到队列当中去。
<code>prioqueueName size</code>	返回队列中元素的个数。

在创建优先级队列的时候，可以指定名字；如果不指定，那么会自动的为你创建一个名字。我们还可以指定优先级比较的方式，默认为 -integer。也就是说插入队列的每一个元素的优先级必须是整数才行。当比较方式是 integer 或者是 real（实数）的时候，元素按照

优先级由大到小的顺序排列；当比较方式是 `ascii` 或者 `dictionary` 的时候，元素按照优先级的由小到大的顺序排列。例如：

```
% ::struct::prioqueue q      ;#采用默认的-integer 比较方式
q
% q put a 100 b 200          ;#插入两个元素 a 和 b，优先级分别为 100 和 200
% q peek 2                  ;#取出元素，b 在 a 前面。b 的优先级 200 比 a 的 100 大。
b a
% ::struct::prioqueue -ascii r ;#创建队列 r，采用 -ascii 方式进行比较
r
% r put a 100 b 200          ;#插入同样的两个元素 a 和 b，优先级为 100 和 200
% r peek 2                  ;#取出元素，a 在 b 前面。这是 100 和 200 的 ascii 比较顺序
a b
```

优先级队列的使用和刚才介绍的普通队列相比差别不大，多了一个 `peekpriority` 命令，它只是简单的返回队列前面 `count` 个元素的优先级列表，我们简单的看一下：

```
% struct::prioqueue q
q
% q put a 100 b 200 c 300 d 50 e 150
% q peek 5
c b e a d
% q peekpriority 5          ;#查看优先级，返回列表对应{c b e a d}的优先级
300 200 150 100 50
% q clear                  ;#清空队列
% q destroy                ;#删除队列
```

优先级队列是一个很有用的数据结构，我们在后面求图（`graph`）中的最短路径的时候需要用到。

## 跳越式列表（`skipList`）

`skipList` 这种数据结构可以用来代替平衡树，它是在 1989 年由马里兰大学的 William Pugh 教授开发出来的。`SkipList` 既保留了二叉树搜索的高效率，同时输入数据的顺序对其效率影响甚微。`skipList` 是为了提高效率而出现的数据结构，但是 `tclLib` 中的 `skipList` 是采用 Tcl 语言实现的，所以效率上并不一定高。最好的方法就是使用 C 语言来编写扩展库实现 `skipList`。

有关 `skipList` 的原理，请参考论文《Skip lists: a probabilistic alternative to balanced trees》June 1990。在网络上也有很多 `skipList` 的介绍。下面我们看看如何使用它：



命令名和格式	功能说明
::struct::skiplist ?skiplistName?	创建一个 skipList 对象，名字为 skiplistName
skiplistName delete node ?node...?	删除列表中的某个节点
skiplistName destroy	删除整个列表
skiplistName insert key value	向列表中插入节点，索引为 key，值为 value
skiplistName search node	在列表中搜索指定的节点
skiplistName size	返回列表中元素的个数
skiplistName walk cmd	遍历列表，对每一个节点执行 cmd 命令

命令都很简单，请看下面的例子：

```
% package require struct
2.1
% struct::skiplist a      ;#创建名字为 a 的列表
a
% a insert LeiYuhou 100   => 1      ;#依次加入多个节点。
% a insert Lily 200       => 1
% a insert 1 30           => 2
% a insert 2 40           => 3
% a insert 1,30 40        => 1
% a insert 1.5 60         => 2
% a search 1              => 1 30
% a search 1.5            => 1 60
% proc OnVisit {k v} {puts "$k = $v"}      ;#定义遍历访问函数
% a walk OnVisit           ;#遍历列表 a
1 = 30
1,30 = 40
1.5 = 60
2 = 40
LeiYuhou = 100
Lily = 200
```

可以看到，skipList 中每一个节点都是由 key 和对应的 value 组成，这和 array 比较类似。当 insert 一个节点的时候，如果 key 已经存在，那么就更新该节点的 value 并且返回 0；否则就创建这个节点，并且返回这个节点的 level（一个内部数据）。

skiplist 内部的节点是按照 key 的大小来进行升序排列的。key 可以是任意字符串，比较大小的时候使用默认的“>”操作符来进行比较。当使用 walk 遍历的时候，针对每一个节点都会执行 cmd 参数所指定的命令，并且将节点的 key 和 value 作为参数附加其后。

我们可以将 skiplist 看成是一个排序过的 array。不过它们性能相比怎样呢？写一个小小的程序来验证一下：

```
001 package require struct
002
003 #插入 1000 个节点到 skipList 中
004 struct::skiplist a
005 set i 10000
006 puts "Insert skiplist: [time {a insert [incr i] "Value_$i"} 1000]"
007
008 #插入 1000 个节点到 array 中
009 set i 10000
010 array set b {}
011 puts "Insert array : [time {set b([incr i]) "Value_$i"} 1000]"
012
013 puts "size of skiplist = [a size]"
014 puts "size of array      = [array size b]"
015
016 #比较查询时间
017 puts "[time {a search 10800} 50]"
018 puts "[time {set b(10800)} 50]"
```

在我的系统上执行结果如下：

```
Insert skiplist: 785 microseconds per iteration
Insert array : 24 microseconds per iteration
size of skiplist = 1000
size of array      = 1000
748 microseconds per iteration
6 microseconds per iteration
```

可以看到，无论是插入还是删除，性能上两者不是一个数量级。原因在于：TCL 中的 skiplist 是使用 TCL 写成的；并且 skiplist 主要是用来代替平衡二叉树数据结构的，而 array 则是 TCL 内建的数据结构，使用 C 实现，它更加类似于一个散列表。所以即使用 C 重写 skiplist，其性能还是比不上 array。

## 栈 (stack)

栈是一种容器，元素先进后出，它的使用非常广泛。TclLib 的 struct 库实现了这种数据结构，我们来看看其命令格式：

命令格式	功能说明
struct::stack ?stackName?	创建栈对象，名字由 stackName 指定

stackName clear	删除栈中的所有元素
stackName destroy	删除栈对象
stackName peek ?count?	取得栈中的 count 个元素，但是不删除它们
stackName pop ?count?	从栈顶弹出 count 个元素
stackName push item ?item ...?	将 item 元素压入到栈顶中
stackName size	返回栈中的元素个数

栈的使用非常简单，请看例子：

```
% struct::stack a      ;# 创建栈对象 => ::a
% a push 1 2 3         ;# 依次将 1 2 3 压入栈
% a peek               ;# 查看栈顶元素 => 3
% a push 10 20 30      ;# 再次将 10 20 30 压入栈
% a peek 6             ;=> 30 20 10 3 2 1
% a push 40
% a peek               ;=> 40
% a size               ;#栈元素的个数 => 7
% a pop                ;#弹出栈顶元素 => 40
% a pop                ;#弹出栈顶元素 => 30
% a clear
% a destroy
```

除了 TclLib 提供的 stack 之外，前面我们介绍了 TclX 的列表命令，其中命令 lvarpop 和 lvarpush 能够将列表模拟成栈（前面我们介绍队列的时候也提到过：这两个命令还能够将列表模拟成队列）。看看怎样模拟：

```
% lvarpush a 1         ;#等价于 a push 1。
% lvarpush a 2         ;#等价于 a push 2
% lvarpush a 3         ;#等价于 a push 3
% lvarpop a            ;# 返回 3，等价于 a pop
% lvarpop a            ;# 返回 2
% lvarpop a            ;# 返回 1
```

使用 TclX 的 lvarpush 等命令的时候，栈顶就是列表头；0 号元素就是栈顶元素。

## 集合（set）

用数学语言来讲，集合是元素的无序组合，一个集合中不能够有重复元素。TclLib 提供的 struct::set 就可以实现常见的集合操作，命令如下：

命令格式	功能描述
------	------

::struct::set empty set	判断集合是否是空的，是空则返回 true
::struct::set size set	返回集合中元素的个数
::struct::set contains set item	判断集合 set 中是否包含元素 item
::struct::set union ?set1...?	求多个集合 set1...setn 的并集
::struct::set intersect ?set1...?	求多个集合 set1...setn 的交集
::struct::set difference set1 set2	返回 set1-set2 集合；
::struct::set symdiff set1 set2	返回(set1-set2) + (set2-set1)集合
::struct::set intersect3 set1 set2	返回一包含三个元素的列表，三个元素分别为 set1 * set2, set1-set2, 以及 set2-set1
::struct::set equal set1 set2	判断两个集合 set1 和 set2 是否相等。

和前面介绍的队列、栈相比，这里介绍的集合没有专门的命令来创建集合对象。它使用普通的列表变量作为集合。请看下面的例子：

```
% package require struct
2.1
% set s1 {1 2 3 0 4 2}      ;# s1 是列表，同时也是集合，里面有两个元素 2
1 2 3 0 4 2
% struct::set size $s1      ;#列表 s1 有 6 个元素，但是作为集合来计算则只有 5 个
5
% struct::set empty $s1     ;#判断集合 s1 是不是空的，返回 0 表示非空
0
% struct::set contains $s1 0 ;#判断集合 s1 中是不是包含元素 0
1
% struct::set contains $s1 5 ;#判断是否包含元素 5，返回 0 表示不包含
0
% set s2 { 3 4 6 10}        ;#第二个集合 s2
3 4 6 10
% ::struct::set union $s1 $s2 ;#求 s1 和 s2 的并集，以列表的形式返回
4 0 10 1 6 2 3
% ::struct::set intersect $s1 $s2 ;#求 s1 和 s2 的交集，以列表的形式返回
3 4
% ::struct::set difference $s1 $s2 ;#求 s1 中那些不在 s2 中的元素集合，即 s1-s2
0 1 2
% ::struct::set symdiff $s1 $s2 ;#求集合 (s1-s2) + (s2-s1)
0 10 1 6 2
% ::struct::set intersect3 $s1 $s2 ;#返回列表：s1*s2, s1-s2, s2-s1
{3 4} {0 1 2} {10 6}
% ::struct::set equal $s1 $s2 ;#两个集合是否相等
```

0

利用集合命令，我们可以很简单的删除一个列表中所有重复的元素，这可以使用 `union` 操作来实现，例如：

```
% for {set i 0} {$i<10} {incr i} {lappend a $i [expr $i +1]}
% set a      ;#列表 a 的内容
0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10
% ::struct::set union $a {} ;#求 a 和空集合的并集，结果就是删除重复元素的 a
8 4 0 10 9 5 1 6 2 7 3
% ::struct::set intersect $a $a ;#求 a 和 a 的并集，重复元素不删除
0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10
% package require Tclx      ;#使用 Tclx 中的列表命令 lrmdups
8.4
% lrmdups $a      ;#直接删除列表 a 中的重复元素
0 1 10 2 3 4 5 6 7 8 9
```

TclX 中提供了 `intersect`、`lcontain` 等命令也可以对列表进行集合操作，并且效率要高出很多。

## 矩阵（matrix）

矩阵也是一种常见的数据结构，元素以行和列的形式来组织，在 `tclLib` 的矩阵中，每一个元素都用两个数字来标识其位置：列（column）和行（row）。它们都从 0 开始计数，随着数字的增加，行表示向下增长，列表示向右增长。还有一种形式就是“`end[-n]`”，`end` 表示最右边的列或者最下边的行。`end-n` 则表示倒数第 `n` 行或者列。`matrix` 提供了较多的命令来操作矩阵，不过都比较简单，我们分类介绍。

## 创建矩阵对象

创建矩阵对象的命令如下：

```
::struct::matrix ?matrixName? ?=|:=|as|deserialize source?
```

可选参数 `matrixName` 是矩阵的名字，如果省略命令会自动取一个名字；`source` 参数是另外一个矩阵对象，如果没有省略，那么表示在构造新矩阵对象之后，马上讲 `source` 的内容复制到新矩阵中。

矩阵创建成功之后，这个矩阵的名字就成了一个普通的 `TCL` 命令，对矩阵对象的操作都是通过这个命令带上不同的参数来完成。如下命令删除矩阵对象：

```
matrixName destroy
```

下面的两个命令是互逆的操作，完成两个矩阵对象之间复制：

命令格式	说明
matrixName = sourcematrix	将 sourcematrix 的内容复制到 matrixName 中
matrixName --> destmatrix	将 matrixName 矩阵复制到矩阵 destmatrix 中。

矩阵被创建之后，里面一般没有任何元素，需要通过命令向里面加入行和列：

命令格式	说明
matrixName add column ?values?	加入一列到矩阵的最后; values 是列值列表
matrixName add row ?values?	加入一行到矩阵的最后, values 是行值列表
matrixName add columns n	向矩阵最后加入 n 列，列元素为空
matrixName add rows n	向矩阵最后加入 n 行，行元素为空
matrixName set cell column row value	设置某一个单元格的值
matrixName set column column values	设置某一列的值
matrixName set rect column row values	设置其中子矩阵的值
matrixName set row row values	设置某一行的值

矩阵对象还支持序列化操作，将矩阵的内容写入到一个列表中，这样就可以写入到文件或者网络中，以后矩阵对象可以从变量将其读进来。命令如下：

命令格式	说明
matrixName deserialize serialization	serialization 是序列化后的内容, 这个命令从其中解序列化，读入到矩阵 matrixName 中
matrixName     serialize     ?column_tl row_tl column_br row_br?	将矩阵 matrixName 进行序列化，后面的参数用来制定子矩阵。

来看看下面的例子，我们要创建下面的这样一个矩阵，代码如下：

```
001 package require struct
002 package require Tclx
003
004 proc CreateMatrix {} {
005     struct::matrix m1
006     m1 add columns 4 ;#首先要增加 4 个 columns
007     m1 add row {101 201 301 401}
008     m1 add row {102 202 0    302}
009     m1 add row {103 0    303 403}
010     puts "m1 = [m1 serialize]" ;输出内容
011
012     struct::matrix m2 as m1 ;#创建矩阵 m2，从 m1 中复制内容
013     m2 set row end {0 0}  设置最后一行的内容
```

$$\left\{ \begin{array}{cccc} 101 & 201 & 301 & 401 \\ 102 & 202 & 0 & 402 \\ 103 & 0 & 303 & 403 \end{array} \right\}$$

```

014     puts "m2 = [m2 serialize]"
015 }
016
017 CreateMatrix

```

上面代码的执行结果如下：

```

m1 = 3 4 {{101 201 301 401} {102 202 0 302} {103 0 303 403}}
m2 = 3 4 {{101 201 301 401} {102 202 0 302} {0 0 {} {}}}

```

这里输出的是矩阵 `serialize` 之后的内容：前面两个数字是行数和列数，后面则是各行元素的列表。可以看到我们调用 `set row end {0 0}` 来设置 `m2` 最后一行的内容的时候，这一行原来的元素会被全部删除，然后用新的值来代替。

## 其它矩阵操作

出了 `add`、`set` 操作之外，矩阵还支持 `delete`、`get` 和 `insert` 等操作。命令如下：

命令格式	命令功能说明
<code>matrixName columns</code>	返回矩阵的列数
<code>matrixName columnwidth column</code>	返回指定列中，最长的元素的长度
<code>matrixName rows</code>	返回矩阵的行数
<code>matrixName rowheight row</code>	返回指定行中，最高的元素的高度（元素的行数）
<code>matrixName delete column column</code>	删除矩阵中指定的列
<code>matrixName delete columns n</code>	从矩阵右边开始，删除 <code>n</code> 列
<code>matrixName delete row row</code>	删除矩阵中指定的行
<code>matrixName delete rows n</code>	从矩阵最底端开始，删除 <code>n</code> 行
<code>matrixName transpose</code>	将矩阵转置（行变成列，列变成行）
<code>matrixName cells</code>	返回矩阵中所有元素的个数
<code>matrixName cellsize column row</code>	返回指定位置元素的大小

接着上面的代码，我们来操作矩阵 `m2`。代码如下：

```

016     m2 transpose      ;#转置矩阵 m2
017     puts "m2 = [m2 serialize]"    ;#输出
018     m2 set cell 1 1 "1234567\n0\n0" ;#修改元素的值
019     foreach m {0 1 2 } {puts "width of column $m = [m2 columnwidth $m]"}
020     foreach m {0 1 2 3} {puts "height of row $m = [m2 rowheight $m]"}
021     m2 delete column 2 ;#删除 1 行
022     puts "m2 = [m2 serialize]"

```

执行结果如下：

```

m2 = 4 3 {{101 102 0} {201 202 0} {301 0 {}} {401 302 {}}}
width of column 0 = 3
width of column 1 = 7
width of column 2 = 1
height of row 0 = 1
height of row 1 = 3
height of row 2 = 1
height of row 3 = 1
m2 = 4 2 {{101 102} {201 {1234567
0
0}} {301 0} {401 302}}

```

这里解释一下命令 `rowheight` 和 `columnwidth`。`rowheight` 表示一行的高度，也就是这一行所有的元素中，元素值的行数最多的那个元素的行数。这里每一个元素都被当成一个字符串。`columnwidth` 则是一列所有元素中，宽度最宽的那个元素的宽度。例如上面的例子中，第一行第一列的元素为“1234567\n0\n0\n0”，其宽度为 7，高度（行数）为 3。所以其所在地行和列的高度和宽度分别为 7 和 3。

还有如下查询和插入类的命令：

命令格式	命令功能说明
<code>matrixName get cell column row</code>	查询指定单元格的值
<code>matrixName get column column</code>	返回 <code>column</code> 指定列的元素的列表
<code>matrixName get rect column_tl row_tl column_br row_br</code>	返回一个表示子矩阵的元素列表。后面的四个参数分别指定了矩阵地坐上角和右下角
<code>matrixName get row row</code>	返回 <code>row</code> 指定行的元素的列表
<code>matrixName insert column column ?values?</code>	向矩阵中插入一列元素，参数 <code>column</code> 指定列的位置；
<code>matrixName insert row row ?values?</code>	向矩阵中插入一行元素，参数 <code>row</code> 指定行的位置；

请看下面的例子：

```

%      struct::matrix m1
::m1
%      m1 add columns 4      ;#首先要增加 columns
%      m1 add row {101  201  301  401}
%      m1 add row {102  202   0   302}
%      m1 add row {103   0   303  403}
%      puts "m1 = [m1 serialize]"      ;#查看序列化后的结果
m1 = 3 4 {{101 201 301 401} {102 202 0 302} {103 0 303 403}}
% m1 get row 0      ;#查看第 0 行的内容

```



```
101 201 301 401
% m1 get column 0 ;#查看第 0 行的内容
101 102 103
% m1 get rect 0 0 2 end ;#得到子矩阵内容（左边三列的全部元素）
{101 201 301} {102 202 0} {103 0 303}
% m1 insert row 1 {new1 new2 new3 new4} ;#在第 1 行位置插入一行
% m1 get rect 0 0 2 end ;#再次查看左边三列的全部元素
{101 201 301} {new1 new2 new3} {102 202 0} {103 0 303}
```

还可以在矩阵中按照一定模式搜索元素，将某一行或者列进行排序，或者交换矩阵两行或者两列内容。搜索元素的命令如下：

1. matrixName search ?-nocase? ?-exact|-glob|-regexp? all pattern
2. matrixName search ?-nocase? ?-exact|-glob|-regexp? column column pattern
3. matrixName search ?-nocase? ?-exact|-glob|-regexp? row row pattern
4. matrixName search ?-nocase? ?-exact|-glob|-regexp? rect column\_tl row\_tl column\_br row\_br pattern

上面四个命令分别是在整个矩阵中、在某一行、某一列以及某一个子矩阵范围内进行搜索，搜索模式可以是 exact、glob 或者 regexp 三种模式。

矩阵中排序的命令如下：

1. matrixName sort columns ?-increasing|-decreasing? row ; 排序某一行
2. matrixName sort rows ?-increasing|-decreasing? column ; 排序某一列

下面的命令用来交换矩阵中的两列或者两行：

1. matrixName swap columns column\_a column\_b; 交换两列内容
2. matrixName swap rows row\_a row\_b; 交换两行内容

## 关联数组

可以将矩阵和 TCL 数组关联起来，通过数组下标 “column,row”，可以非常方便地读取和修改矩阵元素，注意这里是用逗号将列号和行号分开。例如：

```
% m1 link a ;#和数组 a 建立关联
% m1 get cell 0 0 => 101 ;#得到 0 行 0 列的元素
% puts $a(0,0) => 101 ;#查看数组 a 下标为 0,0
% parray a ;#打印出整个数组的内容
a(0,0) = 101
a(0,1) = 102
.....
a(3,2) = 403
```

```
% set a(0,0) 889      => 889      ;#修改数组元素 a(0,0)的值
% m1 get row 0        => 889 201 301 401 ;#直接反映到矩阵元素中
```

矩阵连接数组的命令有：

命令格式	功能说明
<code>matrixName link ?-transpose? arrayvar</code>	将矩阵和数组 <code>arrayvar</code> 建立关联，选项参数 <code>-transpose</code> 表示矩阵转置连接。
<code>matrixName links</code>	返回当前和矩阵建立了关联的所有数组名
<code>matrixName unlink arrayvar</code>	将矩阵和数组 <code>arrayvar</code> 解除关联

一个矩阵可以同时和多个数组变量建立关联。有如下特点：

1. 修改数组元素值的时候同时也修改矩阵元素的值，修改矩阵元素的同时也修改数组元素；
  2. 连接建立后，删除数组变量，不影响矩阵内容；
  3. 连接建立后，删除矩阵对象，也不影响数组内容；
  4. 先建立连接，然后删除连接，关联到的数组变量以及其中内容仍然保留；
- 不过在我发现了如下的一个 Bug。兄弟们在使用的时候要当心：

```
% m1 get rect 0 0 end end ;#m1 是一个 3 行 4 列的矩阵，这里查看全部内容
{ 101 201 301 401 } { 102 202 0 302 } { 103 0 303 403 }
% m1 link a      ;#将数组 a 关联到数组 m1
% m1 link b      ;#将数组 b 关联到数组 m1
% set a(0,0) 2000 ;#修改数组 a(0,0)的值
2000
% set b(0,0)      ;#数组 b(0,0)的值同步改变
2000
% m1 set cell 0 0 3000 ;#直接修改矩阵 m1 的元素 0,0
% set a(0,0) ; set b(0,0) ;#数组 a 和 b 的元素 0,0 也同步自动更改
3000
% unset a        ;#直接删除数组变量 a
% set b(0,0) 2000 ;#再设置 b(0,0) 的值，会出现 bug
can't set "b(0,0)": can't set "data(0,0)": can't read "link(a)": no such element in array
% m1 links       ;#查看和矩阵 m1 相关联的所有数组，只有 b
b
```

## 树 (tree)

现实中的很多实际都能够用树这种数据结构来进行抽象描述，`tclLib` 中提供的树功能非常强大。我们分类讲述：

## 创建和销毁

创建和销毁树的命令如下：

命令格式	功能说明
<code>::struct::tree ?treeName?</code> <code>? = as  deserialize source?</code>	创建一个树对象，树名 <code>treeName</code> 会成为一个 TCL 命令名字
<code>treeName = sourcetree</code>	将 <code>sourcetree</code> 所代表的树复制到 <code>treeName</code> 中
<code>treeName --&gt; desttree</code>	将树 <code>treeName</code> 的内容复制到 <code>desttree</code> 中
<code>treeName serialize ?node?</code>	将树 <code>treeName</code> 的节点 <code>node</code> 以及其所有子节点进行序列化后，将结果返回
<code>treeName deserialize serialization</code>	从序列化后的结果中解序列化
<code>treeName destroy</code>	删除树对象 <code>treeName</code>

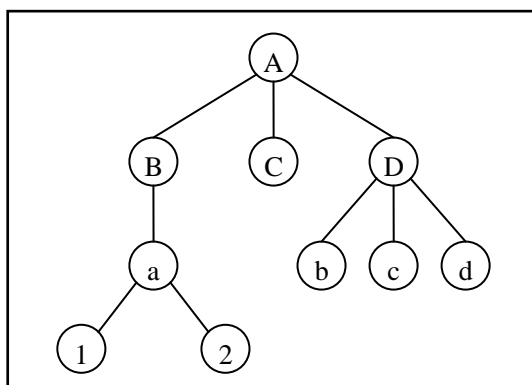
这里树的创建和删除操作，和前面介绍的 `matrix` 的创建和删除是非常类似的。只不过树对象序列化之后的列表结果有其特有的格式。具体我们在后面进行分析。

## 节点操作和查询命令

如下命令用来操作树中的节点，包括插入、删除和移动等等：

命令格式	功能说明
<code>treeName insert parent index ?child ?child ...??</code>	插入节点，作为 <code>parent</code> 的子节点
<code>treeName delete node ?node ...?</code>	删除节点（同时删除子节点）
<code>treeName cut node</code>	删除节点但是保留子节点，所有子节点会成为 <code>node</code> 父节点的子节点
<code>treeName move parent index node ?node ...?</code>	将节点移动，作为 <code>parent</code> 的节点
<code>treeName splice parent from ?to? ?child?</code>	插入一个节点 <code>child</code> 到 <code>parent</code> 的子节点 <code>from</code> 位置，并且 <code>from</code> 到 <code>to</code> 位置的所有节点成为 <code>child</code> 的子节点
<code>treeName swap node1 node2</code>	交换节点 <code>node1</code> 和 <code>node2</code>
<code>treeName rename node newname</code>	为节点 <code>node</code> 改名

在 `TclLib` 提供的树中，每一个节点的名字都是唯一的：在同一棵树中，不可能存在两个同名的节点。下面我们来创建一棵树：



```

030 proc CreateTree {} {
031     ::struct::tree t1      ;#创建树 t1
032     t1 rename root A      ;#为根结点 root 改名为 A
033     t1 insert A 0 B C D   ;#开始插入节点 B,C 和 D，作为 A 的子节点
034     t1 insert B 0 1 2     ;#将 1 和 2 作为 B 的子节点插入
035     t1 splice B 0 end a   ;#将 a 作为 B 子节点插入，1 和 2 降为 a 的子节点
036     t1 insert D 0 b c d   ;#插入 b,c 和 d 作为 D 子节点插入
037
038     t1 set B value 100     ;#设置 B 的属性 value 为 100
039     t1 set B length 2     ;#设置 B 的属性 length 为 2
040 }
041
042 CreateTree
043 foreach {x y z} [t1 serialize] {puts [list $x $y $z]} ;#序列化 t1

```

上面脚本的执行结果如下：

```

A {} {}
B 0 {value 100 length 2}
a 3 {}
1 6 {}
2 6 {}
C 0 {}
D 0 {}
b 18 {}
c 18 {}
d 18 {}

```

请注意这里插入 a 和 1、2 的方式。首先我们调用 insert 将 1 和 2 作为 B 的直接子节点

插入到树中，然后调用 `splice` 将节点 `a` 插入到 `B` 中，并且将原来 `B` 的子节点 1 和 2 降级为 `a` 的子节点。命令 `serialize` 用来将树对象进行序列化，序列化的结果是一个列表，列表元素个数等于“3×节点个数”。从第 0 个元素开始，每三个元素用来描述一个节点：分别为：

1. 节点名字；
2. 节点的父节点名字在结果列表中的序号；例如上例中，`a` 的父节点为 `B`，而 `B` 在列表中的位置为 3，所以序列化后的列表中，`a` 后面的值为 3，用来标出其父节点；
3. 该节点的属性名字和值的列表，其格式类似 `array get` 返回的结果。例如上面例子中，节点 `B` 有两个属性 `value` 和 `length`。所以在最后序列化后的结果中，第 5 个元素为一个属性列表；

对节点的查询类命令有如下几个：

命令格式	功能说明
<code>treeName children ?-all?</code>	返回节点 <code>node</code> 的孩子节点，如果指定 <code>-all</code> ，那么返回所有的
<code>node ?filter cmdprefix?</code>	子孙节点，参数 <code>filter</code> 用来指定一个过滤函数
<code>treeName exists node</code>	判断节点 <code>node</code> 是否存在
<code>treeName depth node</code>	从该节点 <code>node</code> 到根结点的步数，当作本节点的深度
<code>treeName index node</code>	返回节点在其父节点所有子节点列表中的位置
<code>treeName isleaf node</code>	判断节点 <code>node</code> 是不是叶子节点
<code>treeName numchildren node</code>	返回 <code>node</code> 的直接儿子结点的数量
<code>treeName parent node</code>	返回节点 <code>node</code> 的父亲节点
<code>treeName previous node</code>	返回节点 <code>node</code> 的前一个兄弟节点
<code>treeName next node</code>	返回节点 <code>node</code> 的后一个兄弟节点
<code>treeName rootname</code>	返回树的根结点的名字
<code>treeName size ?node?</code>	返回节点 <code>node</code> 的所有子孙节点的数量

这里需要进一步解释的是 `children` 命令。选项参数 `cmdprefix` 用来指定一个命令，该命令在执行的时候额外加入两个参数：树名和当前被测试的节点名。例如：

```

045 proc hasKeyValue {t n} {
046     if {[${t} keyexists $n value]} { ;#判断节点 n 是否有属性 value
047         return true
048     }
049     return false
050 }
051
052 puts [t1 children -all A filter hasKeyValue]
```

这里定义了一个过程 `hasKeyValue`，它用来判断节点是否具有属性 `value`，如果有则返回 `true`，否则返回 `false`。下面调用 `children` 来查询 `A` 的所有具有属性 `value` 的子节点。执行结果就是输出 `B`。因为整棵树种只有 `B` 具有属性 `value`。

如果一个节点有多个直接子节点，那么这些子节点之间是有顺序的：他们构成一个列

表，例如针对刚才创建的树 t1，执行如下的命令：

```
% t1 next b      => c      ;#查询 b 的下一个兄弟节点，返回 c
% t1 next c      => d      ;#查询 c 的下一个兄弟节点，返回 d
% t1 next d      => ""     ;#d 的下一个兄弟节点，为空。应为 d 是最后一个
% t1 previous d   => c      ;#查询 d 的前一个兄弟节点，为 c
% t1 depth b      => 2      ;#节点 b 的深度为 2，从 b->D->A，恰好两步
% t1 depth 2      => 3      ;#节点 2 的深度为 3，从 2->a->B->A，三步
```

## 操作节点属性

每一个节点都可以具有多个属性与之相关联。节点属性的用处很大，例如：如果我们用节点来表示文件系统中的文件，那么文件的属性（大小、创建时间等）就可以用节点属性来表示。设置节点属性的命令有：

命令格式	功能说明
treeName set node key ?value?	为节点 node 设置属性 key 的值；如果省略 value 参数，那么返回节点 node 的属性 key 的值；
treeName unset node key	删除节点 node 的属性 key；
treeName append node key value	将 value 添加到节点 node 的属性 key 的末尾；
treeName lappend node key value	将 node 的属性 key 作为列表，将 value 添加到最后

假设 t1 是文件系统中的某一个目录树，节点 1 和 2 是两个文件，我们来设置其属性：

```
% t1 set 1 created "2006-1-1"      ;#设置节点 1 的创建时间，属性名字为 created
2006-1-1
% t1 set 1 modified 2006-1-4        ;#设置节点 1 的修改时间，属性名字为 modified
2006-1-4
% t1 set 1 owner leiyuhou           ;#设置节点 1 的 owner，
leiyuhou
% t1 set 2 created 2006-1-2
2006-1-2
% t1 set 2 modified 2006-1-3
2006-1-3
% t1 set 2 owner lily
lily
% t1 set 2 owner                    ;#省略 value 参数，查询节点 2 的 owner 属性值
lily
```

与属性设置命令对应，还存在相关查询以及搜索命令：

命令格式	功能说明
------	------

<code>treeName attr key</code>	返回树中所有含有属性 <code>key</code> 的节点名字，以及该节点对应属性值的列表；
<code>treeName attr key -nodes list</code>	在节点列表 <code>list</code> 中，查找含有属性 <code>key</code> 的节点名字，返回名字以及对应属性值的列表；
<code>treeName attr key -glob globpattern</code>	在树中使用 <code>glob</code> 模式来查询节点名字，这些节点必须包含有属性 <code>key</code> 。返回节点名字和属性值的列表
<code>treeName attr key -regexp repattern</code>	在树中使用 <code>regexp</code> 来查询节点名字，这些节点必须包含有属性 <code>key</code> 。返回节点名字和属性值的列表；
<code>treeName getall node ?pattern?</code>	返回节点 <code>node</code> 的所有属性以及对应的值；如果指定 <code>pattern</code> ，那么只返回和它匹配（ <code>glob</code> 模式）的属性；
<code>treeName keys node ?pattern?</code>	返回节点 <code>node</code> 的所有属性名字；如果指定 <code>pattern</code> ，那么只返回和 <code>pattern</code> 匹配的属性名字；
<code>treeName keyexists node key</code>	判断节点 <code>node</code> 是否具有属性 <code>key</code>
<code>treeName get node key</code>	查询节点的属性 <code>key</code> 的值

请看下面的几个例子：

```
% t1 attr created      ;# t1 中包含属性 created 的节点
1 2006-1-1 2 2006-1-2

% t1 attr created -nodes {1 2 a b c} ;# 节点列表{1 2 a b c}中包含有 created 的节点
1 2006-1-1 2 2006-1-2

% t1 attr created -glob *   ;# t1 中名字和*能够 glob 匹配上且含有 created 的节点
1 2006-1-1 2 2006-1-2

% t1 getall 1      ;# 查询节点 1 的所有属性信息
created 2006-1-1 modified 2006-1-4 owner leiyuhou

% t1 keys 1 *ed    ;# 节点 1 中，名字和*ed 能够 glob 匹配上的属性名字
created modified

% t1 keyexists 1 owner ;# 在 t1 树中，节点 1 是否含有 owner 属性
1

% t1 get 1 owner    ;# t1 树中，查询节点 1 的属性 owner 的值
leiyuhou
```

## 树的遍历

树的一个重要操作就是遍历。对二叉树而言，有前序、中序和后序遍历三种方式，这也是很多公司招聘面试的考点:-)。对普通的树而言，则有广度优先和深度优先等几种，遍历树的命令如下：

```
treeName walk node ?-order order? ?-type type? loopvar script
```

参数 `node` 表示将 `node` 作为根开始遍历其所有的子孙节点；选项参数 `type` 可以取值为 `bfs` 和 `dfs`，分别表示“广度优先”和“深度优先”；选项参数 `order` 则用来指定访问节点的顺序，可以为以下几种：

1. `pre`：表示在所有子节点被访问之前，首先访问父节点；
2. `post`：在所有子节点被访问了之后，才访问父节点；
3. `both`：表示在访问子节点之前，首先访问一遍父节点；访问完所有子节点之后，再次访问父节点；
4. `in`：首先访问第一个子节点，然后访问父节点，然后再访问第二个以及其它子节点；这种模式主要应用在二叉树地中序遍历上。

参数 `loopvar` 和 `script` 用来指定循环变量和访问节点时需要执行的脚本。`loopvar` 可以是单个变量名字，每次访问节点的时候，该变量被赋值为节点的名字；也可以是两个变量组成的列表，每次访问的时候，分别为动作（`enter`、`leave` 或者 `visit`）和节点名字。而 `script` 则在调用者空间中被执行。

看到这里，可能比较晕了。我们先看看一个例子：

```
% t1 walk A -order pre -type dfs {a v} {lappend r "$a $v"};set r
{enter A} {enter B} {enter a} {enter 1} {enter 2} {enter C} {enter D} {enter b}
{enter c} {enter d}
```

上面命令中，使用 `pre` 顺序来对树进行深度优先的遍历。打印出来的结果表示了各个节点被访问的顺序以及访问的动作（这里是 `enter`）。可以看到首先访问的节点是 `A`。我们再看看 `post` 顺序的访问结果：

```
% t1 walk A -order post -type dfs {a v} {lappend r "$a $v"};set r
{leave 1} {leave 2} {leave a} {leave B} {leave C} {leave b} {leave c} {leave d}
{leave D} {leave A}
```

可以看到，使用 `post` 顺序来访问时，动作都是 `leave`，表示再完全离开这个节点以及其所有子节点的时候，再访问该节点。上面例子中，最后离开的是根结点 `A`。

最后看看顺序 `both` 的结果，我们更改遍历方法为 `bfs`（宽度优先）：

```
% t1 walk A -order both -type bfs {a v} {lappend r "$a $v"};set r
{enter A} {enter B} {enter C} {enter D} {enter a} {enter b} {enter c} {enter d}
{enter 1} {enter 2} {leave 2} {leave 1} {leave d} {leave c} {leave b} {leave a}
{leave D} {leave C} {leave B} {leave A}
```

看到这里，大家可能会想了：怎样用 `order` 和 `type` 进行组合来实现二叉树的前序中序后序等各种遍历方式呢？其实这三种遍历都是采用深度优先进行的，可以如下实现：

二叉树遍历方式	实现组合方式	
前序遍历	<code>-order pre</code>	<code>-type dfs</code>
中序遍历	<code>-order in</code>	<code>-type dfs</code>



---

后序遍历`-order post -type dfs`

---

## 图 (graph)

图 (Graph) 是一种重要而且比较复杂的数据结构, 分成“有向图”和“无向图”。事实上无向图可以看成是特殊的有向图 (两个节点之间有来回双向的边)。TCL 通过 `struct` 包对有向图提供了完美的支持。这里不介绍图的相关理论, 而且 `struct` 包中有关图的命令有 50 多条, 所以也不一一介绍。我们这里只通过一个简单的例子, 来说明如何在 TCL 脚本中使用图这种数据结构。

### 构造一个图

首先来构造下面的这个有向图。在 TCL 中创建图的命令如下:

```
::struct::graph ?graphname? ?=|:=|as|deserialize source?
```

可选参数 `graphname` 是需要创建的图的名字, 如果省略的话命令会自动的取一个名字。创建成功后, 这个名字实际上就成了一个 TCL 命令。可以用不同的参数来调用这个命令, 从而来对图进行一些操作, 如下:

```
graphname option ?arg...?
```

其中 `option` 参数表示针对图所作的不同的操作, `arg` 根据 `option` 不同而不同。

在创建图的时候, 后面可以加上 `= source`、`:= source` 或者 `as source`, 其中 `source` 是另外的一个图。例如命令: `::struct::graph a = b` 等价于:

```
::struct::graph a
a = b
```

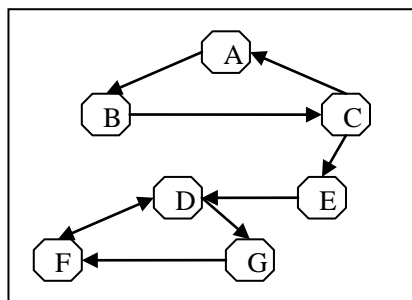
其含义就是将图 `b` 完全复制到新创建的图 `a` 中。这里的“=”就是一个子命令, 格式如下:

```
graphName = sourcegraph
```

图 `graphName` 中原来的内容会被删除, 然后替换成为 `sourcegraph` 中的内容, 它等价于如下的命令:

```
graphName deserialize [sourcegraph serialize]
```

其中的 `deserialize` 和 `serialize` 命令用来对图进行“持续化”操作。



## 插入节点和边

节点（node）是图的组成要素之一，另外一个要素是边（arc）。下面我们来构造上面图对象，首先创建节点，然后创建边。请看代码：

```
001 #=====
002 #File      : graph.tcl
003 #Author    : LeiYuhou
004 #Created   : 2005-11-27
005
006 package require struct 2.0
007
008 #创建图对象的过程
009 proc CreateGraph { } {
010     ::struct::graph gh1 ;#构造名字为 gh1 的图
011     set nodes {A B C D E F G} ;#节点名字列表
012     foreach node $nodes {
013         gh1 node insert $node ;#将节点加入到图中
014     }
015
016     #所有边列表,每一个元素表示边的起始和终结节点名字以及边权重
017     set arcs { {A B 1} {B C 2} {C A 1}
018               {C E 3} {E D 4} {D F 5}
019               {F D 4} {D G 2} {G F 2} }
020     foreach arc $arcs {
021         struct::list assign $arc start end weight
022
023         #将边加入到图中，最后一个参数是边的名字
024         gh1 arc insert $start $end "arc_$start->$end"
025
026         #设置该边的权重
027         gh1 arc set "arc_$start->$end" weight $weight
028     }
029     return gh1 ;#返回图的名字
030 }
031
032 #输出图的内容
```

```

033 proc Output {} {
034     set r [gh1 serialize]
035     foreach {node attr arcs} $r {
036         puts "$node : $attr : Arc : $arcs"
037     }
038 }
039
040 CreateGraph
041 Output
042 gh1 destroy ;#删除图 gh1

```

上面代码中，第 10 行构造了一个名字为 gh1 的图；第 13 行调用 gh1 insert node 将 A—G 七个节点插入到图中；第 24 行则调用 gh1 arc insert 将所有的边插入到图中；第 27 行为每一条边设置权重 weight。第 33 行定义的函数 Output 将图 gh1 的所有内容打印输出。

## 设置点和边的属性

对于 TclLib 中图的每个节点以及每一条边而言，都可以有若干个属性与之关联。每一个属性，由属性名字（key）和属性值（value）组成。这有一点类似于 TCL 数组结果。

上面代码中第 27 行调用了 gh1 arc set 来为每一条边设置权重：属性名字为 weight，属性值则在列表 arcs 中每一个元素的第三个元素给出，例如边 A—>B 的权重为 1。

设置、修改和查询边属性的命令如下（斜体部分是参数）：

命令	说明
graphName arc get arc key	返回边 arc 属性 key 的值
graphName arc attr key	返回图中包含有属性 key 的边名字以及对应的属性值；不包含该属性的边不会返回；
graphName arc attr key -glob globpattern	-arcs -glob -regexp 用来限定查询哪些边；
graphName arc attr key -regexp repattern	返回的格式类似于[array get]的返回形式；
graphName arc getall arc ?pattern?	返回边 arc 所有名字匹配 pattern 的属性的名字和对应值（类似于 array get 返回格式）
graphName arc keys arc ?pattern?	返回边 arc 所有名字匹配 pattern 的属性名字
graphName arc keyexists arc key	判断边 arc 是否有属性 key
graphName arc lappend arc key value	将 value 追加到边 arc 的属性 key 的值后面；属性值是一个列表
graphName arc append arc key value	将 value 追加到边 arc 的属性 key 的值后面
graphName arc set arc key ?value?	将 value 设置为边 arc 的属性 key 的值
graphName arc unset arc key	删除边 arc 的属性 key

看几个例子，执行刚才的 `graph.tcl` 之后，我们在解释器中交互执行如下命令：

```
% source graph.tcl    ;#下面的输出是 serialize 的描述信息
D : :Arc : {arc_D->F 9 {weight 5}} {arc_D->G 15 {weight 2}}
E : :Arc : {arc_E->D 0 {weight 4}}
A : :Arc : {arc_A->B 12 {weight 1}}
F : :Arc : {arc_F->D 0 {weight 4}}
B : :Arc : {arc_B->C 18 {weight 2}}
G : :Arc : {arc_G->F 9 {weight 2}}
C : :Arc : {arc_C->E 3 {weight 3}} {arc_C->A 6 {weight 1}}
: :Arc :
% CreateGraph    ;#重新创建图 gh1
gh1
% foreach {arc value} [gh1 arc attr weight] {puts "$arc $value"}
arc_C->E 3
arc_C->A 1
arc_G->F 2
arc_E->D 4
arc_B->C 2
arc_F->D 4
arc_D->F 5
arc_D->G 2
arc_A->B 1
% gh1 arc attr weight -glob arc_*->F    ;#返回名字匹配 arc_*->F 的边 weight 属性
arc_G->F 2 arc_D->F 5
% gh1 arc getall arc_D->F    ;#查询边 arc_D->F 的所有属性
weight 5
```

`graph` 还可以设置每一个节点的属性，有如下方法：

命令格式	命令说明
<code>graphName node attr key</code>	查询图中包含属性 <code>key</code> 的节点名字以及对应的属性值； <code>-nodes -glob -regexp</code> 用来限制查询哪些节点。
<code>graphName node attr key -nodes list</code>	
<code>graphName node attr key -glob globpattern</code>	
<code>graphName node attr key -regexp repattern</code>	
<code>graphName node get node key</code>	返回节点 <code>node</code> 的属性 <code>key</code> 的值
<code>graphName node getall node ?pattern?</code>	返回节点 <code>node</code> 所有名字匹配 <code>pattern</code> 的属性值，结果类似 <code>[array get]</code> 返回值格式
<code>graphName node keys node ?pattern?</code>	返回节点 <code>node</code> 所有匹配 <code>pattern</code> 的属性名字
<code>graphName node keyexists node key</code>	判断 <code>node</code> 是否包含属性 <code>key</code>

<code>graphName node append node key value</code>	将 value 增加到节点 node 的属性 key 的值的后面;
<code>graphName node lappend node key value</code>	将 value 以列表元素的形式增加到节点 node 的属性 key 的值的后面, 结果是一个列表
<code>graphName node set node key ?value?</code>	设置节点 node 的属性 key 的值为 value
<code>graphName node unset node key</code>	删除节点 node 的属性 key

## 操作节点和边

刚才介绍了属性设置和查询, `graph` 还提供了操作边和节点的函数。现在列举如下:

命令名字	命令功能说明
<code>graphName arc delete arc ?arc ...?</code>	删除参数 arc 指定的边
<code>graphName arc exists arc</code>	判断边 arc 是否存在
<code>graphName arc rename arc newname</code>	将边 arc 改名为 newname
<code>graphName arc source arc</code>	返回边 arc 的源节点
<code>graphName arc target arc</code>	返回边 arc 的宿节点
<code>graphName node degree ?-in/-out? node</code>	返回与节点 node 相邻边的数量
<code>graphName node delete node ?node...?</code>	删除节点 node
<code>graphName node exists node</code>	判断节点 node 是否存在
<code>graphName node opposite node arc</code>	返回边 arc 中相对于 node 的对端节点
<code>graphName node rename node newname</code>	将节点 node 的名字更改为 newname

还是以上面那个图为例, 请看下面的交互执行命令:

```
% gh1 node exists A      ;#判断节点 A 是否存在
1
% gh1 node opposite D arc_F->D  ;#查询边 arc_F->D 中, D 的对端节点
F
% gh1 node opposite F arc_F->D  ;#查询边 arc_F->D 中, F 的对端节点
D
% gh1 node delete D        ;#删除节点 D, 所有 D 的邻接边会被自动删除
% gh1 arcs                  ;#查询图的所有边
arc_C->E arc_C->A arc_G->F arc_B->C arc_A->B
% gh1 arc target arc_C->E    ;#查询边的目的节点
E
% gh1 arc source arc_C->E    ;#查询边的源节点
C
```

## 图的序列化

怎样将一个 graph 图对象保存到某个数据结构，写入到文件中？graph 提供了两个方法来实现这所谓的“序列化”操作：

```
graphName serialize ?node ...?
```

该命令返回节点列表 node...的生成图的描述信息。如果没有指定 node 参数，那么就返回整个图的描述信息。描述信息用一个列表来表示，它包含有  $3*n+1$  个元素，其格式比较复杂：

1. 最后一个元素是一个 dictionary 类型的列表，用来记录整个 graph 对象的属性名字和对应的值；
2. 从第 0 个元素开始，每三个元素可以看成是一个三元组，分别为：
  - a) 节点名字；
  - b) 包含该节点所有属性的 dictionary 型的列表；
  - c) 一个包含了从本节点出发的所有边的列表，数量为  $3*n$  个，可以划分为多个三元组，每一个三元组其结构如下：
    - i. 边的名字；
    - ii. 边的目的节点在 serialize 返回列表中的索引位置；
    - iii. 一个 dictionary 型的列表，描述了该边的所有属性名字和对应的值；

还是拿上面的图为例，我们在交互环境中添加一些属性，然后执行 serialize：

```
% CreateGraph      ;#创建图对象
gh1
% gh1 node set D time 100      ;#为节点 D 设置属性 time=100
100
% gh1 node set E data 300      ;#为节点 E 设置属性 data=300
300
% gh1 set author leiyuhou      ;#为图设置属性 author=leiyuhou
leiyuhou
% set r [gh1 serialize]      ;#序列化结果放到 r 中
% foreach {node attr arcs} [lrange $r 0 end-1] {puts "$node , $attr , $arcs"}
D , time 100 , {arc_D->F 9 {weight 5}} {arc_D->G 15 {weight 2}}
E , data 300 , {arc_E->D 0 {weight 4}}
A , , {arc_A->B 12 {weight 1}}
F , , {arc_F->D 0 {weight 4}}
B , , {arc_B->C 18 {weight 2}}
G , , {arc_G->F 9 {weight 2}}
C , , {arc_C->E 3 {weight 3}} {arc_C->A 6 {weight 1}}
```

```
% lindex $r end    ;#最后一个元素是整个图的属性列表
author leiyuhou
```

与 `serialize` 对应的则是 `deserialize`，格式如下：

```
graphName deserialize serialization
```

其中参数 `serialization` 就是刚才介绍的命令 `serialize` 的输出结果列表。命令执行后会根据 `serialization` 的内容重新设置调用本命令的图对象。

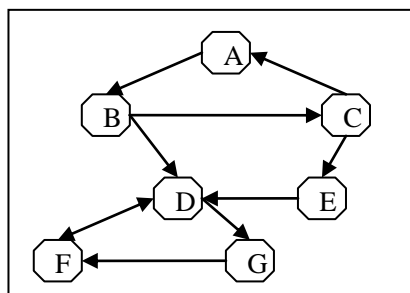
## 图的遍历

图的遍历是一个基本操作，分成“深度优先（DFS）”和“广度优先（BFS）”，其算法并不复杂，`graph` 已经为我们实现了其遍历算法，命令格式如下：

```
graphName walk node ?-order order? ?-type type? ?-dir direction? -command cmd
```

1. 参数 `node` 是遍历的起始节点名字；
2. 选项 `-order` 用来指定访问节点的顺序，可以取值为 `pre`、`post` 或者 `both`；
3. 选项 `-type` 用来指定遍历模式，可以取值为 `dfs` 或者 `bfs`，分别表示深度优先和广度优先模式；默认为 `dfs`；
4. 选项 `-dir` 用来指定遍历方向，可以为 `backward` 和 `forward`，分别表示逆入边方向和顺着出边的方向；
5. 参数 `-command` 用来指定访问节点时需要执行的过程名字，该过程在被执行的时候会加上三个参数：访问模式（`enter` 或者 `leave`）、图名字以及节点名字。

我们以右边的图为例，来看看深度优先和广度优先遍历的结果，请看代码：



```
001 #=====
002 #File      : graph_Walk.tcl
003 #Author    : LeiYuhou
004 #Created   : 2005-11-27
005
006 package require struct 2.0
007
008 #创建图对象的过程，返回图名字 gh2
009 proc CreateGraph {} {
010     ::struct::graph gh2    ;#构造名字未 gh2 的图
011     set nodes {A B C D E F G} ;#节点名字列表
```



```

012     foreach node $nodes {
013         gh2 node insert $node    ;#将节点加入到图中
014     }
015
016     #所有边的列表,每一元素表示边的起始和终结节点名字以及边的权重
017     set arcs { {A B 1} {B C 2} {C A 1} {B D 2}
018               {C E 3} {E D 4} {D F 5}
019               {F D 4} {D G 2} {G F 2} }
020     foreach arc $arcs {
021         struct::list assign $arc start end weight    ;#加入边
022
023         #将边加入到图中, 最后一个参数是边的名字
024         gh2 arc insert $start $end "arc_$start->$end"
025     }
026     return gh2    ;#返回图的名字
027 }
028
029 proc OnVisitNode {mode graph node} {
030     global r    ;#本过程就是访问节点时的过程
031     lappend r "$mode $node"
032 }
033
034 CreateGraph
035 puts "-----BFS walk-----" ; set r ""
036 gh2 walk A -order pre -type bfs -dir forward -command OnVisitNode;puts $r
037
038 puts "-----BFS walk-----" ; set r ""
039 gh2 walk A -order pre -type bfs -dir forward -command OnVisitNode;puts $r
040
041 puts "-----DFS walk-----" ; set r ""
042 gh2 walk A -order pre -type dfs -dir forward -command OnVisitNode;puts $r
043
044 gh2 destroy

```

上面代码中从 36 行开始, 分别采用三种不同的方式来遍历图, 执行结果如下:

```

-----BFS walk-----
{enter A} {enter B} {enter C} {enter D} {enter E} {enter F} {enter G}
-----BFS walk-----

```

```
{enter A} {enter B} {enter C} {enter D} {enter E} {enter F} {enter G}
-----DFS walk-----
{enter A} {enter B} {enter C} {enter E} {enter D} {enter F} {enter G} {enter D}
```

看起来似乎有些问题，采用 dfs 模式的时候，节点 D 居然被访问了两次！我们自己来写一个简单的 dfs 遍历算法吧：

```
034 #=====
035 #DfsWalk    - 过程，对图 gh 做深度优先的遍历
036 #参数
037 #    gh      - 图对象的名字
038 #    start    - 起始节点的名字
039 #    cmd      - 访问节点的回调方法
040 package require Tclx      ;#这里需要用到 Tclx 的列表操作
041 proc DfsWalk {gh start cmd} {
042     set stack ""           ;#栈对象
043     set vstlist ""        ;#保存已经访问节点列表
044
045     lvarpush stack $start
046     while {[lempty $stack]==0} {
047         set curVertex [lvarpop stack]      ;#栈顶节点
048         if {[lcontain $vstlist $curVertex]==0} {
049             $cmd enter $gh $curVertex ;#调用回调函数
050             lappend vstlist $curVertex    ;#放入已访问节点列表中
051         }
052
053         #寻找栈顶节点的所有邻接节点
054         set adjVertex [$gh nodes -out $curVertex]
055         foreach v $adjVertex {
056             if {[lcontain $vstlist $v]==0} {
057                 lvarpush stack $v    ;#压入栈顶
058             }
059         }
060     }
061 }
062
063 CreateGraph      ;#创建图对象
064 set r "" ; DfsWalk gh2 A OnVisitNode ; puts $r    ;#深度优先搜索
065 set r "" ; DfsWalk gh2 B OnVisitNode ; puts $r
```

上面的代码执行结果如下：

```
{enter A} {enter B} {enter D} {enter G} {enter F} {enter C} {enter E}
{enter B} {enter D} {enter G} {enter F} {enter C} {enter E} {enter A}
```

深度优先搜索的算法比较简单，需要一个栈和一个列表来保存中间数据，我们这里全部采用列表来实现，其中使用了 TclX 的一些列表命令。具体的图遍历算法可以参考相关数据结构和算法的书籍。

## 最短路径算法

在一个图中，从一个节点到达另外一个节点的路径有多种，假设路径经过的每条边都有一个权重，那么如何寻找一条权重之和最小的路径呢？这是一个非常常见的算法，但是 TCL 的 graph 图没有提供这样的算法，下面我们自己来实现一个。请看下面的代码：

```
066 #=====
067 # MinPath    - 实现最短路径算法，返回路径节点列表
068 # gh         - 图的名字
069 # start      - 开始节点的名字
070 # end        - 结束节点的名字
071 # keyname     - 边的权重属性的名字，默认为 weight
072 proc MinPath {gh start end {keyname weight}} {
073     set prioq [::struct::prioqueue -integer] ;#创建优先级队列
074     set vlist [list] ;#访问过的节点列表
075
076     set found 0 ;#是否已经找到的标志
077     $prioq put [list [list $start ] 0] 0 ;#加入起始节点，优先级为 0
078     while { $found==0 && [$prioq size]>0 } {
079         set path [$prioq get] ;#返回优先级最高的节点（路径最短的）
080         set cost [lindex $path end] ;#当前的权重
081         set endVertex [lindex $path 0 end] ;#最后的一个节点名字
082
083         #如果已经找到了路径，那么就设置 found=1，退出循环
084         if { $endVertex eq $end } {
085             set found 1
086         } else {
087             lappend vlist $endVertex ;#将节点放入到 vlist
088             set adjArcs [$gh arcs -out $endVertex] ;#所有的出边
089             foreach arc $adjArcs {
```

```

090          set vertex [$gh arc target $arc ]      ;#目的节点
091          set w [$gh arc get $arc $keyname] ;#该边的权重
092          if {[lcontain $vlist $vertex]==0} {
093              set newCost [expr $cost + $w] ;#新的权重
094              $prioq put [list [linsert [lindex $path 0] end $vertex] \
095                          $newCost] -$newCost
096          }
097      }
098  }
099  }
100
101  $prioq destroy ;#删除优先级队列
102  if {$found==1} {
103      return $path ;#返回找到的路径
104  }
105
106  return "No Path" ;#否则表示没有可达路径
107 }
108
109 CreateGraph ;#构造图对象
110 puts [MinPath gh2 A F] ;#寻找 A->F 的最短路径
111 puts [MinPath gh2 E A] ;#寻找 E->A 的最短路径
112 puts [MinPath gh2 E F] ;#寻找 E->F 的最短路径

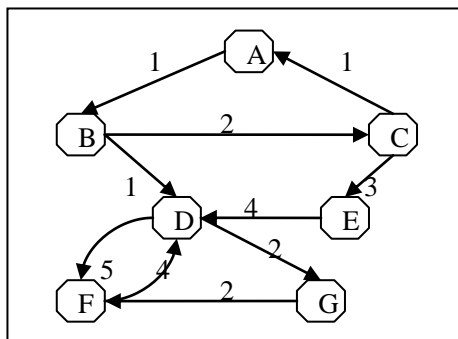
```

呵呵，“最短路径算法”稍微有一点复杂，如果不解释一下，不一定能够看懂上面的代码。上面函数 `MinPath` 有四个参数：`gh` 表示需要搜索的图对象的名字；`start` 和 `end` 分别表示要搜索的起始节点和结束节点；`keyname` 表示边的权重属性名字，默认为 `weight`。函数 `MiniPath` 的功能就是在图 `gh` 中搜索一条从 `start` 到 `end` 的最短路径。

函数内部使用了两个数据结构：一个优先级队列，和一个普通列表。优先级队列中的每一个元素都有一个优先级，当从队列中取出一个元素的时候，总是首先取出优先级最高的元素，在我们的算法里面，优先级被定为“路径长度的负数”（数字越大，优先级越高）。队列里面的数据元素都是列表，格式如下：

{ { 路径经过的节点列表 } 路径权重 }

算法开始的时候首先将 { `start` 0 } 节点放入到队列中，然后进入循环，每次都取出最短的一个路径，然后判断该路径的结束节点是不是 `end`，如



果是，那么就找到返回；如果不是，那么就将结束节点的所有邻接节点分别加到路径最后，各自构造出新的路径，然后加入到队列中（已经访问过的节点就省略不加入）。列表 `vlist` 用来保存已经被访问过的节点列表。

上面代码的 109 行创造如上所示的图，然后分别搜索 A->F，E->A 和 E->F 的最短路径，其执行结果如下所示：

```
{A B D G F} 7  
No Path  
{E D G F} 8
```

从 A 到 F 的最短路径为 ABDGF，权重为 7；从 E 到 A 根本就没有路径；从 E 到 F 的最短路径为 EDGF，权重为 8。

关于图，还有很多非常有趣的算法，例如求可达矩阵的 Warshall 算法。有兴趣的朋友可以参考相关书籍然后自行实现！

## TCL 与 COM 组件

如果你是一个纯粹的 Unix 程序员，讨厌 Windows 中的一切（也许游戏除外），甚至连 Windows 操作系统都没有，那么本章对你来说没有任何意义，请直接跳过。

COM 和自动化应该说是 Windows 系统中最激动人心的技术之一！因此也是值得每一个 Windows 下的程序员去学习的技术。Windows 操作系统中，土生土长的语言是 VBScript 以及 JScript。它们最大的优势就是对 COM 自动化的良好支持：

1. 这两种语言能够作为自动化客户端，来控制自动化服务器；
2. 还能够直接使用 VBScript 或者 JScript 来编写自动化服务器；

Windows 的很多系统功能和服务都通过组件方式来提供。TCL 要想这个充满组件的操作系统中生存下去，必须能够有效的支持 COM 自动化。而事实上 TCL 也做到了这一点。

这里出现了两个概念：自动化客户端和自动化服务器。在深入介绍 TCL 的自动化操作之前，有必须先介绍一下 COM 自动化的基本知识

## COM 自动化简介

COM 是微软发明的技术，它可不是我这里能够三言两语能够说清楚的。其初衷是为了解决应用程序之间数据交互的问题，到后来发展成了一套非常复杂而且庞大的体系。在 COM 最重要的概念就是接口，接口中包括了三种元素：方法、属性和事件。一个组件通过一个或者多个接口向外界提供功能和服务。两个接口非常的重要：

1. IUnknown：一切接口的基类，所有的接口都从该接口派生；
2. IDispatch：自动化接口，脚本语言可以通过该接口来使用组件的功能；该接口是自动化操作的核心接口。但是这里我们没有必要深入了解其工作机制。

下面我们先来解决一个问题：编写一个程序，通过它打开电子表格程序 Excel，然后创建一个电子表格，在 A1 单元格中写入公式 “=100+200”，验证其值是不是 300。

乍一看挺吓人的，老一代的程序员可以还会记得 dBase 数据库 dbf 文件的格式，那个年代只有深入了解了 dbf 文件格式，才有可能自己编程创建一个 dbf 文件。但是对于 Excel 而言，没有这个必要。Excel 本身就是一个巨大的 COM 组件，它提供了成百上千个接口，我们通过这些接口就可以轻松的完成这些操作。请看如下的 VBScript 代码：

```
001 '=====
002 'File : Exceltest.vbs
003
004 '创建一个 Excel 应用程序对象
005 Set a = CreateObject("Excel.Application")
006 a.workbooks.Add '新增一个 Workbook
```

```
007 a.visible = true '让应用程序处于可见状态
008
009 '设置 A1 单元格的公式
010 a.ActiveSheet.Range("A1").Formula = "=100+200"
011 '输出 A1 单元格的值
012 WScript.Echo "A1 value = ", a.ActiveSheet.Range("A1").Value
```

将上面的代码保存为 Exceltest.vbs 文件，这是一个 VBScript 脚本，真正的代码虽然只有 5 行，但是却完全解决了我们的问题。进入 Windows 的命令提示符，执行如下的命令：

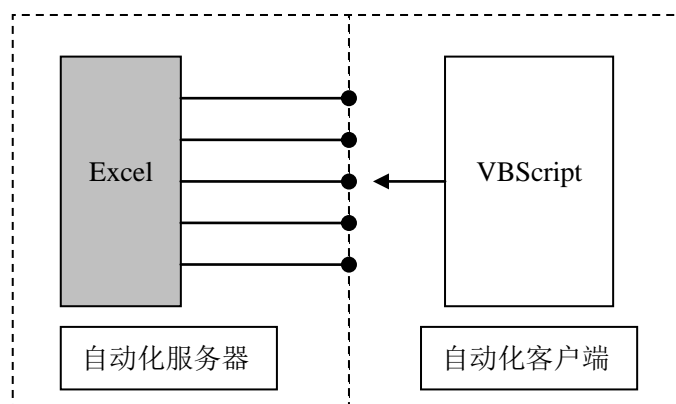
```
cscript.exe Exceltest.vbs
```

cscript.exe 是 Microsoft 提供的 Windows 脚本解释器程序，能够执行 VBScript 和 JScript，它一般位于 Windows 的系统目录下。在我的机器上执行结果如下：

```
Microsoft (R) Windows Script Host Version 5.6
版权所有(C) Microsoft Corporation 1996-2001。保留所有权利。

A1 value = 300
```

程序正确执行的前提之一是机器上必须安装了 Excel 程序。程序执行后除了打印上面的信息之外，还会打开 Excel 窗口，并且可以看到在一个新的电子表格页面上，A1 单元格的内容为：300。下面我们来解释一下这个过程：



Excel 在这里就是一个“自动化服务器”，它通过接口向外面提供了一些功能。而我们刚才的 VBScript 脚本程序就是“自动化客户端”，它通过这些接口使用了 Excel 提供的服务和功能。如果我们知道了服务器提供的接口规格，那么就可以在任何支持自动化客户端的语言中来控制自动化服务器，包括：

1. 调用接口中提供的方法；
2. 读取或者设置接口中提供的属性；
3. 响应对象接口中被触发的事件；

上面代码中：

第 5 行，调用函数 `CreateObject` 创建了一个 Excel 服务器对象，同时得到了该对象的句柄（本质上就是一个 `_Application` 接口）。该函数的参数是一个字符串，在 COM 中它被叫做“ProgID”，我们可以将其等价的看成对象的名字。得到接口后，一切就好办了；

第 6 行，`workbooks` 是对象的一个属性，返回另外一个接口 `Workbooks`，而 `add` 则是这个接口的一个方法，该方法创建一个新的电子表格（`Worksheet`）；

第 7 行，`visible` 是接口 `_Application` 的一个可读写属性，用来控制 Excel 主窗口的可见性。如果不将该属性设置为 `True`，那么 Excel 的主窗口就不可见。

上面的 VBScript 脚本就演示了一个最简单的操作 COM 自动化服务器的方法。客户端可以是任何支持 COM 自动化的语言，例如编译型语言 Delphi，C++；也可以是脚本语言，例如 VBScript，JScript，或者 Python，Ruby 以及我们要介绍的 Tcl。客户端和服务端可以采用不同的语言来开发。

客户端和服务端可以共存于同一个进程内（进程内服务器），也可以分别位于两个不同的进程中（本地服务器），甚至可以跨网络存在于两个不同的机器上（远程服务器）。其中进程内服务器速度是最快的，它一般采用动态链接库 DLL 来实现。刚才我们看到的 VBScript 操作 Excel，则是一个典型的进程外服务器的例子，因为 `cscript.exe` 和 `Excel.exe` 分别位于两个不同的进程中。

自动化服务器必须在系统注册表中注册，对于进程内服务器，可以调用 `regsvr32.exe` 来进行注册。注册就是将进程名字“ProgID”和 COM 类组件的 `classID` 以及实现组件的文件名等信息写入注册表。`CreateObject` 方法会根据 ProgID 在系统注册表中查找相关信息，对于进程内服务器则加载动态链接库并且创建类，对于进程外服务器，则是启动相关进程并且得到接口句柄。

## 使用 TCL 来操作 Excel

下面我们介绍如何使用 Tcl 脚本来完成同样的工作。请看代码：

```
001 #=====
002 #File      : tcl_excel.tcl
003 #Desc      : use tcl script to operate Excel application
004 #Author : LeiYuhou
005
006 package require tcom
007
008 #创建自动化服务器，得到其_Application 接口
009 set app [tcom::ref createobject -local "Excel.Application"]
010
011 [$app -get Workbooks] Add      ;#新建一个 Workbook
```



```
012 $app -set Visible true      ;#主窗口可见
013
014 #设置当前活动的 WorkSheet 的 A1 单元格的公式
015 [[ $app -get ActiveSheet] -get Range "A1"] -set Formula "=100+200"
016
017 #获取 A1 单元格的值，应该是 300
018 puts "A1 value = [[ $app -get ActiveSheet] Range A1] Value]"
019 unset app
```

这段代码就是将 ExcelTest.VBS 翻译成了 Tcl 脚本。它看起来似乎没有 VBScript 那样直观:-)，不过习惯就好，我现在就觉得非常清晰明了。使用 TCL 解释器执行它，结果如下：

```
A1 value = 300.0
```

同时也会看到 Excel 被打开，新创建电子表格的 A1 单元格值变成 100。下面我们对这段代码来逐行解释：

第 6 行引入了扩展包 tcom，该扩展包提供的命令可以使 Tcl 具备自动化客户端的功能。要想在 TCL 中操作自动化服务器，必须首先引入该扩展包。

第 9 行，调用了扩展包 tcom 中的命令：tcom::ref createobject，它和 VBS 中 CreateObject 功能类似。根据 ProgID 加载或者启动自动化服务器，然后得到其接口 \_Application，放入变量 app 中保存。

第 11 行，查询 \_Application 接口中属性 Workbooks，得到新接口 Workbooks，然后调用该方法 Add，该方法的目的就是新增一个 Workbook。

第 12 行，直接设置 \_Application 接口的属性 Visible 为 true，使窗口可见。

第 15 行乍一看很复杂，仔细一看却非常简单：首先通过读取 app 的 ActiveSheet 属性得到当前活动的 WorkSheet；然后读取属性 Range，A1 是属性 Range 的参数，从而得到一个 Range 接口；最后设置该接口的属性 Formula 为字符串 “=100+200”。

第 18 行与 15 行类似，得到 A1 单元格的属性 Value，该属性值是另一个属性 Formula 计算后的结果，300 恰好是 100+200 的结果。

第 19 行使用 unset 命令将 app 删除，从而释放接口 \_Application。

以上就是使用 Tcl 来操作自动化服务器的最基本步骤了，其它任何复杂的操作也不过如此。实际上 Microsoft Office 中的每一个程序都是功能强大的自动化服务器组件，我们只要得到自动化组件的接口规格，就可以使用他们。

## 在 TCL 中使用组件举例

Windows 中的很多系统服务都是通过 COM 组件的方式来提供的，有了 tcom，那么就可以在 TCL 中得心应手的使用第三方提供的各种自动化组件地服务，来完成各类复杂的操作，下面就举几个例子。这几个例子都使用了 Windows 操作系统自身提供的一些 COM

组件以及服务，这里对这些组件具体的接口不作进一步的分析，如果有兴趣可以去参考相关的接口手册。

## 关闭 Windows 系统

下面的脚本可以弹出关闭系统的对话框，类似操作“开始->关闭计算机”。

```
023 #关闭 Windows 系统，类似 start-shutdown 操作
024 proc test_Shutdown {} {
025     set shell [tcom::ref createobject "Shell.Application"]
026     $shell ShutdownWindows
027     unset shell
028 }
029
030 test_Shutdown
```

这里使用了自动化服务器 Shell.Application 的方法 ShutdownWindows，该组件是微软提供的。一般 Windows 操作系统安装好之后，该组件就可以被使用了。

## 列出当前所有进程以及相关信息

Unix 操作系统中有一个命令 ps，可以列出系统中所有进程，其参数个数无数，格式复杂；不过即是如此，它仍然是无数 Unix 程序员的最爱，尤其是在写 Shell 脚本的时候。而 Windows 操作系统中就缺少这样的命令行命令，不过下面的 TCL 脚本可以实现该功能：

```
031 #查询一个 Wbem 对象的属性，返回属性的值
032 proc GetWbemObjProp {obj prop} {
033     return [[[ $obj -get Properties_] -get Item $prop] -get Value]
034 }
035
036 #打印出所有的进程信息（进程 ID 名字,可执行路径）
037 proc ExportProcesses {} {
038     set computer "."
039     set objpath "winmgmts:{impersonationLevel=impersonate}"
040     append objpath "!\\\\$computer\\root\\cimv2"
041     set wmiObjs [tcom::ref getobject $objpath]
042     set allProcesses [$wmiObjs ExecQuery "select * from win32_process"]
043 }
```

```
044      ::tcom::foreach ps $allProcesses {
045          lappend result [list [GetWbemObjProp $ps ProcessId] \
046                          [GetWbemObjProp $ps Name] \
047                          [GetWbemObjProp $ps ExecutablePath]]
048      }
049
050      puts [format "%-5s %-25s %s" Id Name ExecuteablePath]
051      foreach ps $result {
052          lassign $ps id name execpath
053          puts [format "%-5d %-25s %s" $id $name $execpath]
054      }
055      return $result
056 }
057
058 ExportProcesses
```

上面的脚本使用了 WMI（Windows Management Instrumentation）的相关技术，该技术比较复杂，不过对于客户而言，它就是一套结构复杂的自动化组件对象体系。客户端脚本可以通过它们若干接口，来完成对 Windows 系统的控制。

上面的代码在我的 Windows 系统下执行结果如下：

Id	Name	ExecuteablePath
0	System Idle Process	
4	System	
500	smss.exe	C:\windows\System32\smss.exe
564	csrss.exe	
592	winlogon.exe	C:\windows\system32\winlogon.exe
640	services.exe	C:\windows\system32\services.exe
652	lsass.exe	C:\windows\system32\lsass.exe
812	ati2evxx.exe	C:\windows\system32\Ati2evxx.exe
824	svchost.exe	C:\windows\system32\svchost.exe
1084	svchost.exe	
1272	spoolsv.exe	C:\windows\system32\spoolsv.exe
1496	sqlservr.exe	D:\MSDEMSSQL\Binn\sqlservr.exe
1572	svchost.exe	C:\windows\system32\svchost.exe
1848	ati2evxx.exe	C:\windows\system32\Ati2evxx.exe
1996	explorer.exe	C:\windows\Explorer.EXE
2128	cmd.exe	C:\windows\system32\cmd.exe
3908	tclsh84.exe	D:\Tcl\bin\tclsh84.exe

这里我们只是输出了每一个进程的 ID，进程的名字，以及执行该进程的可执行文件的完整路径。如果你愿意，还可以输出其它信息：比如最大工作集、该进程的页面失效次数、使用的虚拟内存大小等等。

## Kill 某些进程

使用和刚才类似的技术，你可以 Kill 掉系统中的某些进程，实现类似 Unix 中 kill 的命令。请看下面的代码：

```
058 #=====
059 #杀掉系统中的某些进程，函数用法：
060 # KillProcess -id n : 杀掉进程 id 为 n 的进程
061 # KillProcess -name pattern : 杀掉名字匹配 pattern 的进程，模式字符串如下：
062 # [ ] specified range ([a=f]) or set ([abcdef]).
063 # ^ not within the range ([^a=f]) or set ([^abcdef].)
064 # % Any string of 0 (zero) or more characters.
065 # _ (underscore) Any one character.
066 proc KillProcess {flag idname} {
067     set objpath "winmgmts:{impersonationLevel=impersonate}"
068     append objpath "!\\\\.\\root\\cimv2"
069
070     #创建 wmi 自动化对象
071     set wmiObjs [tcom::ref getobject $objpath]
072
073     #解析参数，构造查询语句中的条件子句
074     switch -- $flag {
075         -id { set where "ProcessId = $idname" }
076         -name { set where "Name LIKE \"$idname\"" }
077         default { error "invalid flag: $flag" }
078     }
079
080     #查询得到符合条件的进程对象的集合
081     set pss [$wmiObjs ExecQuery \
082         "select * from win32_process WHERE $where"]
083
084     ::tcom::foreach ps $pss {
085         #构造输入参数，Reason 的值为 -1
```

```
086      set param [[[ $ps Methods_] Item "Terminate"] \
087              InParameters] SpawnInstance_]
088      [[ $param Properties_] Item "Reason"] -set Value -1
089      set psname [[[ $ps Properties_] Item "Name"] Value]
090
091      #调用对象的 Terminate 方法，中止该进程
092      set r [ $ps ExecMethod_ "Terminate" $param]
093      set result [[[ $r Properties_] Item ReturnValue] -get Value]
094      puts "Terminate $psname result = $result"
095  }
096 }
097
098 KillProcess -name mspaint.%      ;#干掉所有名字匹配 mspaint.%的进程
099 KillProcess -id 812              ;#干掉 id 为 812 的进程
```

这里定义了一个过程 KillProcess，有两种调用方式：

1. KillProcess -id n: n 是需要被 Kill 掉的进程 ID；
2. KillProcess -name pattern: pattern 是一个模式字符串，里面可以使用通配符。这里的通配符和 TCL 里面的 glob 匹配有一些差别，倒是和 SQL 语句中的 LIKE 字符串类似，这一点在函数注释中已经写出来了；

第 98 行和 99 行分别调用该函数来杀掉一些进程。我先运行两次 mspaint.exe 得到两个进程，然后执行本脚本，结果如下：

```
Terminate mspaint.exe result = 0
Terminate mspaint.exe result = 0
Terminate ati2evxx.exe result = 2
```

前两个 mspaint.exe 都被成功的 Kill，返回结果是 0；最后一个进程 ati2evxx.exe 是一个驱动程序，Kill 的时候返回 2，其含义是“Access Denied”。

本程序的核心思想也是利用了 WMI 中的自动化服务器对象。通过 tcom 来得到对象接口，然后调用接口中的方法和属性，这里主要是 Terminate，从而将进程 Kill 掉。

## 改变磁盘卷标

下面的函数 ModifyVolumn 可以修改逻辑磁盘的卷标。接受两个参数：磁盘盘符，新的卷标字符串。函数返回的是修改前的磁盘卷标。

```
101 proc ModifyVolumn {disk volumn} {
102     set objpath "winmgmts:{impersonationLevel=impersonate}"
```

```
103     append objpath "!Win32_LogicalDisk=\"$disk:\""
104     set disk [::tcom::ref getobject $objpath]
105
106     #得到原来的  volumn
107     set result [[[ $disk Properties_] Item VolumeName] -get Value]
108
109     #设置新的  volumn
110     set r [[[ $disk Properties_] Item VolumeName] -set Value $volumn]
111     $disk Put_
112
113     #返回原来的  volumnName
114     return $result
115 }
116
117 #修改 C 盘的卷标为 NewSystem，并且打印出原来的卷标
118 puts [set old [ModifyVolumn C "NewSystem"]]
119 puts [ModifyVolumn C $old] ;#修改回去
```

这里使用了 WMI 自动化服务器对象提供的服务接口：通过 tcom 直接修改磁盘的盘符。最后两行都是用来修改 C 盘的卷标。在我机器上运行结果：

```
System
NewSystem
```

我机器的 C 盘原来的卷标就是 “System”

## 查询 CPU 信息

如何在程序中获得 CPU 的一些制造信息，比如地址总线宽度，Family 信息，L2 Cache 的大小和速度？如果是 DOS 年代可以直接执行某些特殊的机器指令，但是在脚本语言中该当如何？这也可以通过 WMI 组件来得到，请看代码：

```
121 proc GetCpuInfo {id} {
122     set props [list AddressWidth \
123                 Architecture \
124                 Caption \
125                 CurrentClockSpeed \
126                 DataWidth \
127                 Description \
```

```
128             ExtClock \  
129             Family \  
130             L2CacheSize \  
131             L2CacheSpeed \  
132             Manufacturer \  
133             MaxClockSpeed \  
134             OtherFamilyDescription \  
135         ]  
136     set objpath "winmgmts:root\\cimv2:win32_processor=\"cpu$id\""  
137     set cpu [::tcom::ref getobject $objpath]  
138     set propobjs [$cpu Properties_  
139     foreach prop $props {  
140         set r($prop) [$propobjs Item $prop] -get Value]  
141     }  
142     return [array get r]  
143 }  
144  
145 foreach {cinfo v} [GetCpuInfo 0] {  
146     puts [format "%-22s = %s" $cinfo $v]  
147 }
```

我机器的 CPU 是 Intel Celeron D 336。运行上面的脚本结果如下：

```
DataWidth           = 32  
Architecture        = 0  
L2CacheSpeed        =  
Description          = x86 Family 15 Model 4 Stepping 1  
AddressWidth        = 32  
OtherFamilyDescription =  
CurrentClockSpeed    = 2814  
Family              = 2  
MaxClockSpeed        = 2814  
ExtClock             = 133  
L2CacheSize          = 0  
Manufacturer         = GenuineIntel  
Caption              = x86 Family 15 Model 4 Stepping 1
```

可以看到 CPU 的地址总线与数据总线宽度是 32 位，架构为 0 表示 X86 架构，主频是 2.8G，外部时钟时 133M，制造商是 Intel。事实上我们还能够获得更加丰富的信息，具体

属性数据，请参考 MSDN 中 WMI 的参考手册。

## tcom 详细使用方法

刚才给各位演示了一些眼花缭乱的 TCL 控制操作系统的例子。相信各位读者看了之后应该对在 Tcl 中使用 tcom 来控制自动化服务器的步骤和方法有了一个基本的了解，也看到了这种方法的广阔应用前景。同时也可能会产生一些疑问，例如：

1. 我怎么知道当前的系统中，有哪些自动化服务器可供使用？
2. 我怎么知道自动化服务器组件提供的接口？这些接口是怎么描述的？
3. 现在为止我看到了 getobject 和 createobject 两种方法，它们有什么差别？
4. 如何调用接口方法？如何设置和查询接口属性？
5. 如何响应服务器组件的事件？

首先回答第一个问题，没有什么更好的方法让你知道有哪些组件可以使用。当你有一个目的需要去达到的时候，就需要去寻找相关的办法和工具，这个寻找过程和任务执行者的经验和技能密切相关。平时做一个有心人比较重要，例如我们需要实现拼写语法检查的功能，那么我们就应该记起 Office 中的拼写检查功能，那么就应该去看看这个功能是不是通过自动化服务器组件提供了相关的接口，而事实上确实如此。另外平时多关注 MSDN 也是一个不错的方法。

其它的问题我们分段描述。

## 了解和解析接口

接口一般都是用 IDL（Interface define language）来进行描述，不过我们一般我们无法直接得到这个文件，它有些时候被编译成 tlb（type library）文件后，作为资源被一起编译到组件实现文件（一般是 dll 或者 exe 文件）中，有时 tlb 文件也会被一起发布。tlb 文件是一个二进制文件，即使如此，我们也有方法将其解析出来。下面举两个例子来说明

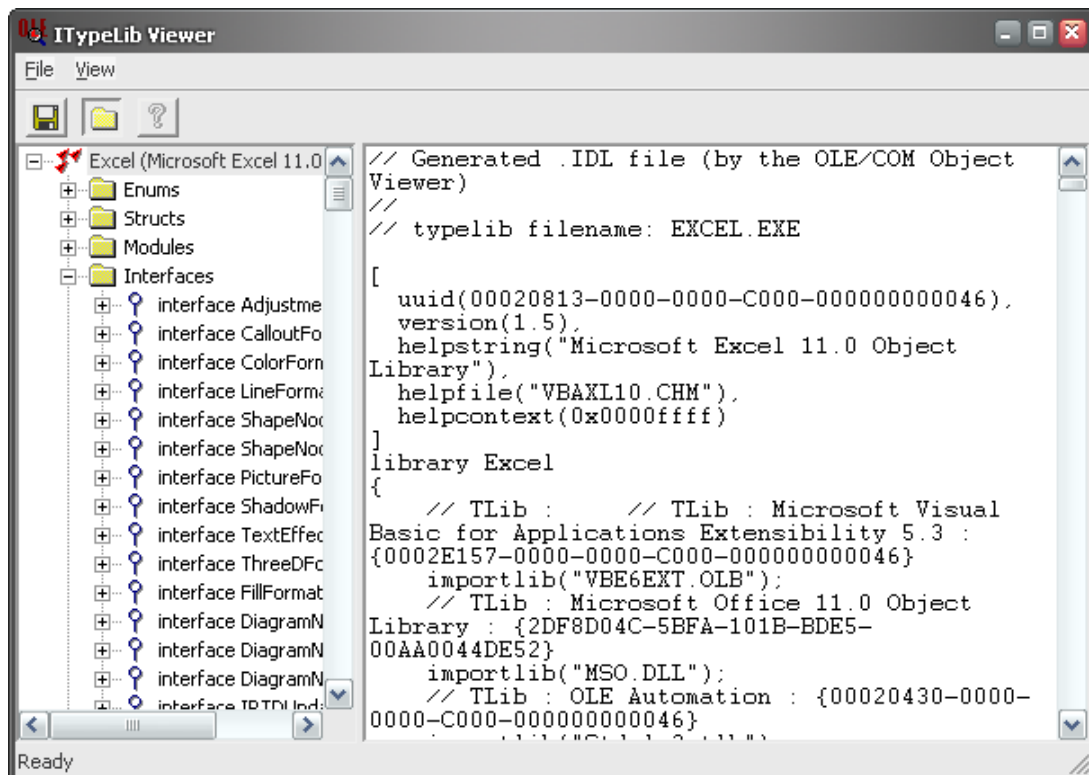
## 得到 Excel 接口和说明手册

前面我们举了一些和 Excel 相关的例子，那么有关 Excel 的接口规格在哪里？我们首先想到的就是 excel.exe 文件中是否包含了 Tlb 信息：

1. 打开 Visual Studio 中附带的程序 oleview.exe；
2. 然后选择其菜单“File->View typeLib...”，打开文件 excel.exe；

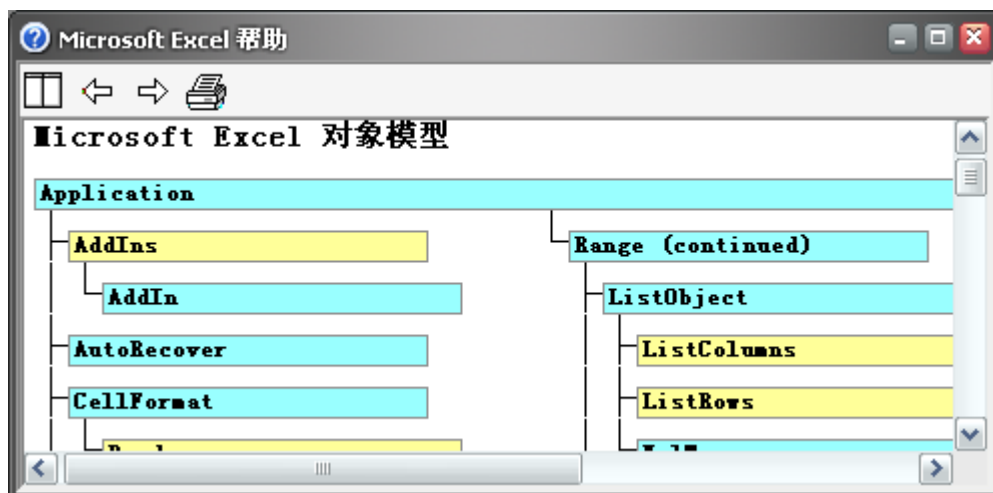
出现如下图所示的窗口内容：





这个窗口中出现了所有的接口信息。左边是一棵树：显示出该 TLB 文件中所有定义的常量名字、接口名字、以及自动化类名等信息；右边窗口中则是左边选中项目的详细信息。选中某个结点，可以把相应信息保存到其它的文本文件中。

以上只是得到了 IDL 信息，光凭这个东西还不能够给我们足够的指引。我们需要一份详细的接口功能说明，怎样得到？很简单：启动 Excel 文件，打开其帮助手册，选择“Microsoft Excel 对象说明”，就可以看到详细的对象结构，如图：



选中你感兴趣的对象，就可以马上看到对象的详细功能说明、以及对象的各个属性和

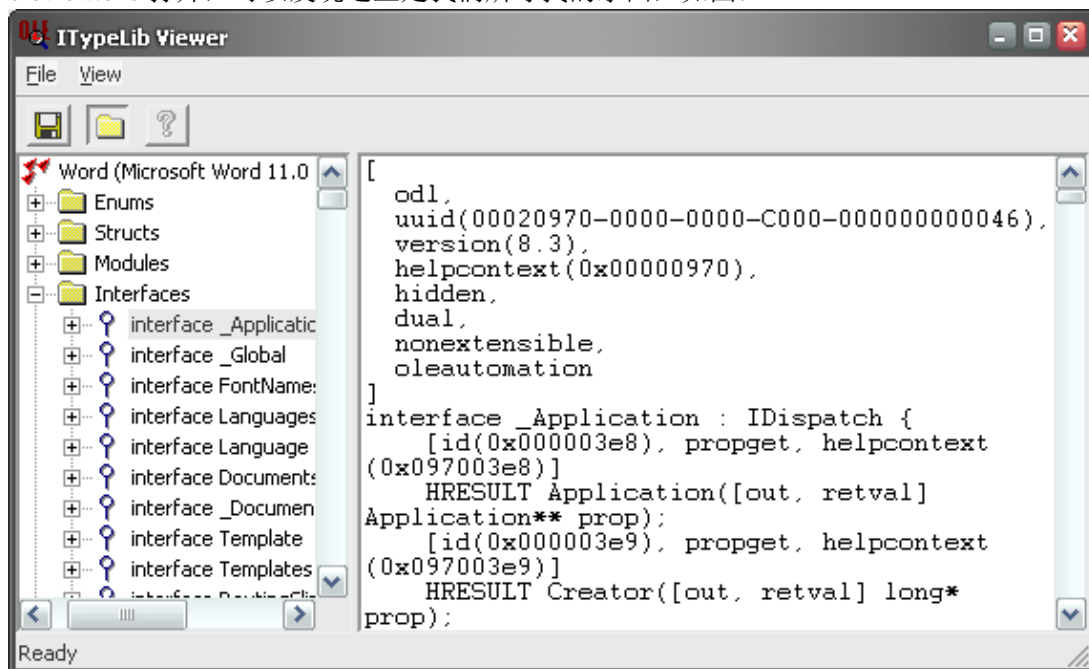
方法的功能、参数等等说明。这就是你的 COM 组件编程手册。

## 得到 Word 的接口说明

Word 是 Office 中另外一个应用程序，如何得到其完整的接口？如果采用和刚才 Excel 类似的方法进行操作，会出现如下的错误：



这说明 Tlb 信息没有编译到可执行文件中，那么这些信息在哪里？到 winword.exe 文件所在目录中仔细寻找一下，可以看到一个 MsWord.olb 文件，这个文件有点可疑，用 oleview.exe 打开，可以发现这正是我们所寻找的东西，如图：



打开 Word 的联机帮助文件，打开“Microsoft Word VB 参考”，里面详细的给出了各个对象、方法和属性的功能规格说明。虽然例子都是用 VBA 写成的，但是我们可以轻易的移植到 Tcl 中。

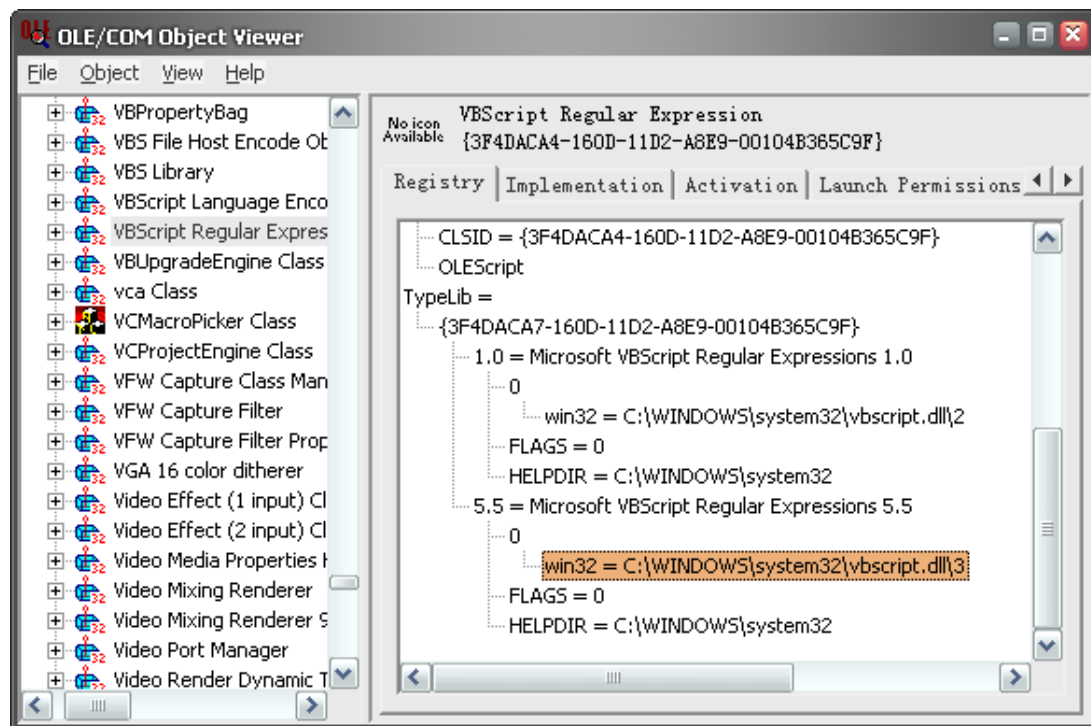
看到这里读者可能会问：你找到这些定义信息有什么用处？回答是：用处很大。好比我们使用 C 的运行时库的时候，必须有相应的.h 头文件一样，我们在使用 COM 组件的时

候, `tlb` 信息就好比头文件, 它详细描述了所有的接口信息。虽然在写脚本的时候、即使没有 `tlb` 信息也可以调用 COM 组件, 但是有了它之后我们可以用的更好, 关于这一点我们后面会有进一步的解释。

## 得到 VBScript 中正则表达式组件接口

刚才的两个例子都比较简单, 一下子就找到了 `tlb` 信息所在。现在这个例子比较复杂一些, 如果你在 VBScript 脚本中用过 `RegExp` 对象, 一定会被其精巧的对象结构所吸引。这个对象一般通过语句 `set r = new RegExp` 来创建, 它完成的功能和 Tcl 命令 `regexp` 类似。事实上它是一个自动化组件对象, 我们如何找到其 `Tlb` 信息? 这是一个小小的挑战。

我们首先想到的就是 `oleview.exe`, 它会显示出系统中所有的 COM 组件的信息:



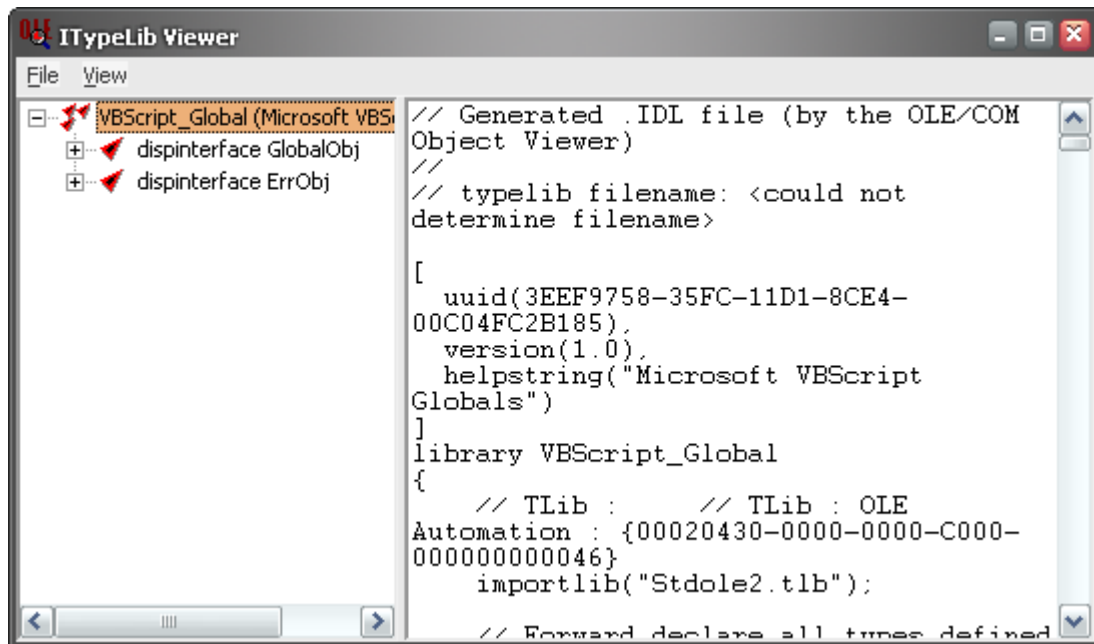
在左边的树中打开“`All Objects`”节点, 找到并且选中“`VBScript Regular Expression`”节点, 右边的窗口中马上出现了该对象的相关信息。首先可以肯定的是: 这个对象就是我们在 VBScript 中用到的 `RegExp` 对象。可以看到在 `Registry` 标签页看到该组件的注册信息, 其中就有 `TypeLib` 信息。上图中, 我们可以看到选中的“`win32=...`”。它表示该组件的 5.5 版本的 `Tlb` 信息保存在如下地方:

`C:\WINDOWS\system32\vbscript.dll#3`

这是一个比较奇怪的字符串, 前面是一个文件 `vbscript.dll` 的完整路径, 该文件是确实存在的, 并且 `RegExp` 的 `Tlb` 信息几乎可以肯定保存在这个文件中。但是文件名的最后还

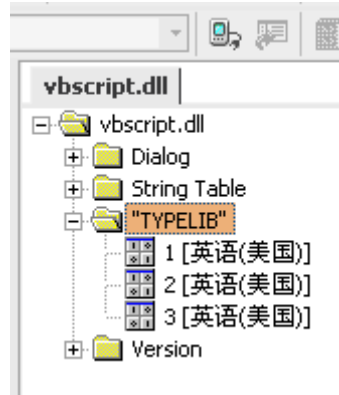
有一个“\3”，它是什么东西？表示什么意思？

先不管，使用 oleview.exe 的菜单“File->View Typelib...”打开这个 vbscript.dll 文件：



结果让我们很失望，这里没有发现任何 RegExp 对象的相关说明信息。问题在哪里？问题就在刚才的文件路径最后的那个“\3”。我们前面提到过：在 Exe 或者 Dll 文件中，tlb 信息是以资源形式存在的。一个 Dll 文件中可以存在多个 Tlb 资源。这里的“\3”是不是资源 ID？为了验证这一点，我们用 Visual studio 以资源的形式打开 vbscript.dll 文件，如图：

可以看到 VbScript.dll 文件中包含了三个 TYPELIB 类型的资源，我们可以将这三个 TYPELIB 类型的资源分别导出到 vbreg[1-3].tlb 这三个文件中。然后再分别用 Oleview.exe 打开，可以看到 vbreg1.tlb 文件的内容，就是我们直接打开 vbscript.dll 文件看到的内容。而 vbreg3.tlb 的内容则是 RegExp 对象的接口说明。



事实上 Microsoft 提供的 Automation 编程接口手册中，如下几个函数可以非常方便的寻找和加载 Tlb 信息：

```
HRESULT LoadTypeLib(const OLECHAR* szFile, ITypeLib** pptlib);
HRESULT LoadRegTypeLib(
    REFGUID rguid,
    unsigned short wVerMajor, unsigned short wVerMinor,
    LCID lcid,
```

```
    ITypeLib ** pptlib  
);
```

其中第一个函数 LoadTypeLib 的参数 szFile 就可以指定 dll 文件路径,如果该文件中存在多个 TypeLib 资源,可以在最后加上 “\n” 来指定序号为 n 的资源,这正和我们看到的 RegExp 对象注册信息的路径格式一致。关于 TypeLib 的相关 API 函数,可以参考 MSDN 中的相关资料。这里不进一步论述。

## 引入到 TCL 中

我们提到过, TypeLib 信息就好比 C 语言函数库的头文件。tcom 能够将 Tlb 文件引入到 TCL 解释器中,从而方便我们的使用。命令 tcom::import 可以完成该功能,格式如下:

```
::tcom::import typeLibrary ?namespace?
```

参数 typeLibrary 可以是一个 tlb 文件,可以是包含 tlb 资源的 Dll 或者 Exe 文件。参数 namespace 可选,表示将这个类型库内容导入到 Tcl 的这个名字空间中。如果你曾经在 Visual C++ 中使用过 #import 这个指令,那么就会很好理解 tcom 的这个命令。我们看一个例子:

```
001 package require tcom  
002 package require Tclx  
003  
004 #将 tlbPath 中的内容导入到 Tcl 中,并且输出相关内容  
005 proc OutputTlb {tlbPath {ns _x}} {  
006     tcom::import $tlbPath $ns    ;#将 tlb 引入到 $ns 中  
007  
008     puts "All commands in $ns:"  
009     foreach cmd [set i 0;info commands ${ns}::*] {  
010         puts "([incr i]) $cmd"  
011     }  
012  
013     puts "All variables in $ns:"  
014     foreach v [set i 0;info vars ${ns}::*] {  
015         puts "!!([incr i]) $v"    ;#数组名字  
016         parray $v                ;#打印数组内容  
017     }  
018 }  
019  
020 OutputTlb d:/vbreg3.tlb
```

```
021 OutputTlb {C:\Program Files\Common Files\System\ado\msado15.dll}
```

过程 OutputTlb 将参数 tlbPath 所表示的文件中的类型库信息导入到解释器的名字空间中，然后将名字空间中的内容给打印出来。tcom::import 命令会在名字空间中创建相关如下内容：

1. 为每一个接口和类创建一个同名的命令；
2. 为每一个枚举类型创建一个同名的数组，数组下标是枚举名字，对应值则是数字；

最后两行的代码将 vbreg3.tlb 文件和 msado15.dll 导入到解释器中。vbreg3.tlb 是我们在上一节获得的 VBScript.RegExp 对象的类型信息，msado15.dll 则包含了著名的 MSADO 对象的类型信息说明（ADO 是微软提供的一套数据库操作对象组件）。下面是部分执行结果：

```
All commands in vbreg:
(1) ::vbreg::ISubMatches
(2) ::vbreg::IMatchCollection2
(3) ::vbreg::IMatch2
(4) ::vbreg::IRegExp
.....
(11) ::vbreg::RegExp
All variables in vbreg:
!!(1) ::vbreg::__uuidof
::vbreg::__uuidof(IMatch)           = 3f4daca1-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(IMatch2)          = 3f4dacb1-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(IMatchCollection) = 3f4daca2-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(IMatchCollection2) = 3f4dacb2-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(IRegExp)           = 3f4daca0-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(IRegExp2)          = 3f4dacb0-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(ISubMatches)        = 3f4dacb3-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(Match)              = 3f4daca5-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(MatchCollection)    = 3f4daca6-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(RegExp)             = 3f4daca4-160d-11d2-a8e9-00104b365c9f
::vbreg::__uuidof(SubMatches)         = 3f4dacc0-160d-11d2-a8e9-00104b365c9f
All commands in ado:
(1) ::ado::Parameter
(2) ::ado::Fields20
(3) ::ado::Recordset15
.....
(39) ::ado::Command
(41) ::ado::Fields
(42) ::ado::_DynaCollection
```

```
All variables in ado:
!!(1) ::ado::FieldEnum
::ado::FieldEnum(adDefaultStream) = -1
::ado::FieldEnum(adRecordURL)      = -2
!!(2) ::ado::ResyncEnum
::ado::ResyncEnum(adResyncAllValues)      = 2
::ado::ResyncEnum(adResyncUnderlyingValues) = 1
!!(3) ::ado::StreamReadEnum
::ado::StreamReadEnum(adReadAll)  = -1
::ado::StreamReadEnum(adReadLine) = -2
.....
```

从上面的执行结果可以看到：对于每一个接口，`import` 都生成一个对应的命令，通过这个命令可以直接创建该接口所对应的对象。例如通过命令 `IRegExp` 可以创建一个 `RegExp` 对象。另外还有一个数组变量 `__uuidof`，每一个元素都是接口对应的 `UNID`。

除此之外，IDL 中定义的每一个枚举类型，`import` 指令都会生成一个数组，数组名字就是枚举类型的类型名，各个枚举值的名字就是对应的下标，对应的值则是枚举的值。例如 IDL 的枚举类型 `ExecuteOptionEnum` 定义如下：

```
typedef
[uuid(0000051E-0000-0010-8000-00AA006D2EA4), helpcontext(0x0012ec10)]
enum {
    adOptionUnspecified = -1,
    adAsyncExecute = 16,
    adAsyncFetch = 32,
    adAsyncFetchNonBlocking = 64,
    adExecuteNoRecords = 128,
    adExecuteStream = 1024,
    adExecuteRecord = 2048
} ExecuteOptionEnum;
```

指令 `import` 生成的对应数组变量如下：

```
::ado::ExecuteOptionEnum(adAsyncExecute)      = 16
::ado::ExecuteOptionEnum(adAsyncFetch)        = 32
::ado::ExecuteOptionEnum(adAsyncFetchNonBlocking) = 64
::ado::ExecuteOptionEnum(adExecuteNoRecords)  = 128
::ado::ExecuteOptionEnum(adExecuteRecord)     = 2048
::ado::ExecuteOptionEnum(adExecuteStream)     = 1024
::ado::ExecuteOptionEnum(adOptionUnspecified) = -1
```



我们可以通过变量 `ExecuteOptionEnum` 来使用对应的枚举值。

## 创建服务器对象

通过前面诸多例子可以看出，使用自动化对象的第一步就是创建自动化服务器对象。创建对象的方法有两种，一个是传统的 `createobject` 等方法，另一个是使用 `import` 指令生成的命令，下面分别讲解。

### 使用 `createobject` 方法创建

通过 `tcom::ref createobject` 和 `tcom::ref getobject` 命令可以得到自动化对象句柄，其命令如下：

```
::tcom::ref createobject ?-inproc? ?-local? ?-remote? ?-clsid? progID ?hostName?  
::tcom::ref getactiveobject ?-clsid? progID  
::tcom::ref getobject path
```

选项参数 `-inproc`、`-local` 和 `-remote` 分别表示创建进程内、本地和远程的自动化服务器对象。如果省略这些参数，系统会自动的根据注册表内的组件信息来创建对应的服务器对象，如果某一个组件同时注册了多种类型的服务器，那么系统会按照“进程内”、“本地”和“远程”由高到低的优先级顺序来创建对象，因为进程内服务器的速度和性能是最快的。

选项参数 `-clsid` 表示参数 `progID` 是 CLSID，而不应该当作 `ProgID` 来解释。CLSID 是一个 128 位的 ID，使用特殊的算法产生，足以保证全球在若干千年之内每一个组件的 ID 都是唯一的。但是这样的数字 ID 太难以记忆了，所以就出现了 `ProgID`，它是其一个方便记忆的名字。参数 `hostName` 只在创建远端服务器对象的时候才有用，一般情况下省略。

命令 `getactiveobject` 用来连接一个已经创建起来的自动化对象。例如我们可以先启动一个 Excel 实例，然后调用 `getactiveobject` 来得到其接口，并且对其进行控制。

命令 `getobject` 用来从参数 `path` 指定的文件中得到句柄。例如我们可以从一个.xls 电子表格文件中得到 `WorkBook` 或者 `WorkSheet` 对象接口，进而进行进一步的操作和控制。上一章节的例子中，我们为了查询系统中的进程，就调用 `getobject` 得到了一个对象接口，不过那种情况下的参数 `path` 不是一个磁盘文件，而是一个特殊格式的字符串。

下面我们来看一个 `getactiveobject` 的例子：

```
% #首先通过开始菜单，启动 Office 中的 Excel 电子表格程序  
% set h [::tcom::ref getactiveobject "Excel.Application"] ;#得到对象接口  
::tcom::handle0x009304F0  
% $h -get Visible ;#查询 Visible 属性  
1
```



```
% $h -set Visible false ;#隐藏 Excel 的主窗口！执行完之后会发现 Excel 消失了
% $h -get Visible      ;#再次查询 Visible 属性，为 0 表示隐藏了，不可见
0
```

下面我们来看看使用 `getObject` 方法从文件得到接口的例子：

```
% set h [tcom::ref getObject c:/book1.xls] ;#得到一个 Workbook 对象接口
::tcom::handle0x009157C0
% [$h ActiveSheet] Name ;#得到当前页面的名字
Sheet1
```

## 使用 `import` 导入生成的命令

`import` 会根据 TLB 信息在解释器中生成一些命令，下面我们以前面生成的 `vbreg3.tlb` 文件为例子，来试着在 TCL 脚本中使用 VBScript 的正则表达式对象，请看例子：

```
% package require tcom
3.9
% tcom::import i:/vbreg3.tlb vb ;#将 TLB 引入到 vb 名字空间中
VBScript_RegExp_55
% set r [vb::RegExp] ;#创建 RegExp 服务器对象，接口保存在 r 中
::tcom::handle0x009288F0
% $r Global true ;#设置 Global 属性为 True
% $r Pattern {0[xX][0-9a-fA-F]+} ;#设置正则表达式的模式，匹配十六进制数字
% $r Test "0x3ab + 0x33" ;#进行匹配测试，结果是 1
1
% $r Test "3ab + 33" ;#匹配另一个字符串，结果是 0
0
```

使用 `import` 引入的命令和枚举常量，程序看起来更加直观，易于维护。

## 调用方法和属性

创建自动化对象之后，就可以调用接口提供的方法；并且可以读取和设置对象属性。其格式有多种，如下：

```
handle ?-method? method ?argument ...?
handle -namedarg method ?argumentName argumentValue ...?
handle ?-get? ?-set? property ?index ...? ?value?
```

第一种格式我们已经用过了，要注意的是参数-method，一般情况下我们都省略。如果使用它，就表示我们在明确调用一个方法。

第二种格式我们叫做命名参数调用，是为了明确地给某一个参数赋值，这种方法只在特殊情况下使用。例如：使用过 VBA 编写过程序的读者可能见过下面的 VBA 的写法：

```
ActiveWorkbook.Sheets.Add Before:=Worksheets(Worksheets.Count)
```

这里 Sheets 集合的方法 Add 用来新增一个 Sheet 对象，其功能说明如下：

Sheets.Add(Before, After, Count, Type)

expression：必需。该表达式返回上面的对象之一；

Before：Variant 类型，可选。指定工作表对象，新建的工作表将置于此工作表之前；

After：Variant 类型，可选。指定工作表对象，新建的工作表将置于此工作表之后；

Count：Variant 类型，可选。要新建的工作表的数目。默认值为 1；

这里的几个参数都是可选的，并且 Before 和 After 不能够同时出现；如果我们需要在最后新增一个 WorkSheet，那么就必须使用 After 参数。在 VBA 中可以这样调用：

```
ActiveWorkbook.Sheets.Add After:=Worksheets(Worksheets.Count)
```

但是如果是 VBScript 脚本，那么就没有办法了：-(。不过 TCL 中可以做到，如下：

```
% set app [:tcom::ref createobject "Excel.Application"] ;#创建 Excel 服务器
:tcom::handle0x00916B30
% $app Visible 1 ;#设置为可见的
% set wkb [[$app Workbooks] Add] ;#新增加一个 Workbook
:tcom::handle0x00944310
#在最后面的一个 Worksheet 后面增加一个新的 Sheet
% set shtObj [[$wkb Sheets] -namedarg Add After [[$wkb Worksheets] Item 3]]
:tcom::handle0x0094E4E0
% $shtObj Name ;#查看该 Worksheet 的名字
Sheet4
```

第三种格式用来读取和设置属性，参数-get 和-set 可以省略。属性可以有参数。如果是设置属性，那么命令最后必须加上属性的新值；如果没有新值，那么就表示读取属性。例如刚才的 Visible 就是 ExcelApplication 对象的一个属性。对于 Excel 的 Workbook 对象而言，属性 Colors 表示该工作簿的调色板属性，它有参数，格式如下：

```
expression.Colors(Index)
```

返回或设置工作簿调色板中的颜色。调色板共有 56 项，每一项用一个 RGB 值表示。Variant 类型，可读写；

1. expression：必需。该表达式返回一个 Workbook 对象；
2. Index：Variant 类型，可选。颜色号（从 1 至 56）。如果未给出本参数，本属性返回包含调色板中所有 56 种颜色的数组；

我们试着来读取和设置该属性：

```
% $wkb Colors 21      ;#读取索引 21 的调色板颜色属性
6684774.0
% $wkb -get Colors 21   ;#同上，读取 21 号颜色
6684774.0
% $wkb -set Colors 21   ;#虽然是-set，但是缺少属性新值，所以还是读取属性
6684774.0
% $wkb -set Colors 21 255 ;#设置 21 号颜色值为 255
% $wkb -set Colors 21    ;#再次读取出来，应该等于刚才设置的 255
255.0
```

## 响应事件

自动化对象还可以触发事件，通过事件来通知客户端：某些事情发生了。TCL 为事件响应机制提供了支持，通过如下两个命令来实现：

```
::tcom::bind handle command ?eventIID?
::tcom::unbind handle
```

第一个 bind 命令用来将事件 eventIID 和命令 command 绑定起来，当句柄 handle 所表示的对象触发了 eventIID 接口所表示的事件的时候，就会执行 command 命令；命令 unbind 用来解除绑定关系。我们一般情况下省略 eventIID 参数，这时对象默认得事件源接口被绑定。事件可以有参数。请看下面的例子：

Excel 的 Workbook 对象支持 BeforeClose 和 Deactivate 事件，我们想通过响应这些事件来作一些自己的工作。代码如下：

```
% set wkb [$App Workbooks] Add]    ;#创建一个工作簿对象
::tcom::handle0x01D67A10
#定义事件响应函数，参数为事件 id 和额外的参数
% proc OnWkbEvent {id args} {puts "Event:$id.args=$args"}
#将 wkb 对象和函数绑定起来
% ::tcom::bind $wkb OnWkbEvent
% $wkb Close      ;#关闭 Workbook 对象，来触发一些事件
Event:BeforeClose.args=::tcom::arg_Cancel
Event:WindowDeactivate.args=::tcom::handle0x01D67B30
Event:Deactivate.args=
% ::tcom::unbind $wkb    ;#解除绑定
```

上面的例子中，我们定义了事件响应函数 OnWkbEvent，它有两个参数：id 表示接收

到的事件名字, args 则是事件的参数。当事件被触发的时候, Workbook 对象会自动的使用相关参数来调用该响应函数。

我们通过调用 Workbook 的方法 Close 来关闭工作簿, 并且触发了三个事件。最后使用 unbind 来解除事件绑定。

## 查看接口信息

如果我们已经的到了一个对象句柄, 可以通过 tcom::info 命令来查询其相关信息。首先调用 tcom::info interface handle, 得到一个接口描述句柄, 然后通过调用该句柄来进一步得到相关信息, 它支持如下参数:

参数	说明
handle iid	得到接口的 IID
handle methods	得到接口所支持的所有方法及其参数说明列表
handle name	得到接口的名字
handle properties	得到接口所支持的所有属性以及参数说明列表

例如:

```
% set r [::tcom::info interface $wkbs]    ;#得到接口$wkbs 的描述句柄
::tcom::handle0x00971BF0
% $r iid    ;#查询得到接口的 IID
000208db-0000-0000-c000-000000000046
% $r name    ;#查询得到接口的名字
Workbooks
% foreach m [$r methods] {puts $m}    ;#打印出 Workbooks 的所有方法
148 {Application *} Application {}
149 I4 Creator {}
150 DISPATCH Parent {}
181 {Workbook *} Add {{in VARIANT Template}}
277 VOID Close {}
118 I4 Count {}
170 {Workbook *} Item {{in VARIANT Index}}
.....
% foreach m [$r properties] {puts $m}    ;#打印出接口所有的属性
148 out {Application *} Application
149 out I4 Creator
150 out DISPATCH Parent
118 out I4 Count
```

```
170 out {Workbook *} Item {{in VARIANT Index}}
-4 out UNKNOWN _NewEnum
0 out {Workbook *} _Default {{in VARIANT Index}}
```

## 用 TCL 编写 COM 组件

前面章节所讲述的是如何使用 TCL 来操纵其它的 COM 服务器，这时 TCL 扮演客户端的角色。事实上有了 tcom，还可以使用 TCL 来编写 COM 服务器。

下面我们以 tcom 中附带的 Banking 组件为例子，各个步骤依次说明。

## 编写 IDL 文件

第一个步骤就是编写组件的接口说明文件 Banking.idl，其内容如下：

```
import "oidl.idl";
import "ocidl.idl";

[ object,
  uuid(0A0059C4-E0B0-11D2-942A-00C04F7040AB),
  dual , helpstring("IAccount Interface"), pointer_default(unique)
]
interface IAccount: IDispatch
{
    [id(1), propget, helpstring("property Balance")]
    HRESULT Balance([out, retval] long *pValue);
    [id(2), helpstring("method Deposit")]
    HRESULT Deposit([in] long amount);
    [id(3), helpstring("method Withdraw")]
    HRESULT Withdraw([in] long amount);
};

[ object,
  uuid(0A0059C4-E0B0-11D2-942A-00C04F7040AC),
  dual, helpstring("IBank Interface"), pointer_default(unique)
]
interface IBank: IDispatch
{
    [id(1), helpstring("method CreateAccount")]
```

```
HRESULT CreateAccount([out, retval] IAccount **ppAccount);
};

[
    uuid(0A0059B8-E0B0-11D2-942A-00C04F7040AB),
    version(1.0), helpstring("Banking 1.0 Type Library")
]
library Banking
{
    importlib("stdole32.tlb");

    [
        uuid(0A0059C5-E0B0-11D2-942A-00C04F7040AB),
        helpstring("Account Class")
    ]
    coclass Account
    {
        [default] interface IAccount;
    };

    [
        uuid(0A0059C5-E0B0-11D2-942A-00C04F7040AC),
        helpstring("Bank Class")
    ]
    coclass Bank
    {
        [default] interface IBank;
    };
};
```

这里定义了两个接口 `IAccount` 和 `IBank`，同时定义了两个对象类 `Account` 和 `Bank`，它们分别实现了接口 `IAccount` 和 `IBank`。两个接口都是从 `IDispatch` 派生出来，是双接口。

我们不允许客户可以直接创建 `Account` 对象。必须通过 `IBank` 的方法 `CreateAccount` 可以创建 `IAccount` 接口。

## 生成 TLB 文件

IDL 是文本文件，我们需要将其编译成为 `tlb` 文件，这里需要用到 `MIDL` 编译器，`MIDL` 是微软提供的 `Idl` 文件编译器，随 `Visual C++` 一起提供。打开命令提示符窗口，运行 `Visual`

C++提供的环境设置批处理文件 VCVARS32.BAT，对于 VC6.0 而言，它位于 VC98\BIN 目录下。运行结果如下：

```
D:\>"D:\Progra~1\Microsoft Visual Studio\VC98\Bin\VCVARS32.BAT"
Setting environment for using Microsoft Visual C++ tools.
D:\Tcl\lib\Banking>midl banking.idl
Microsoft (R) MIDL Compiler Version 5.01.0164
Copyright (c) Microsoft Corp 1991-1997. All rights reserved.
Processing .\banking.idl
banking.idl
Processing D:\PROGRA~1\MICROS~3\VC98\INCLUDE\oidl.idl
oidl.idl
.....
Processing D:\PROGRA~1\MICROS~3\VC98\INCLUDE\msxml.idl
msxml.idl

D:\Tcl\lib\Banking>dir *.tlb
D:\Tcl\lib\Banking 的目录

2006-03-12  22:54                2,396 Banking.tlb
                1 个文件                2,396 字节
                0 个目录 25,934,499,840 可用字节
```

可以看到 Midl 根据 Idl 文件生成了对应的 Banking.tlb 文件。

## 创建 TCL 程序包

tcom 实现 COM 服务器需要用到 package 机制，我们需要在 lib 目录下生成一个新的包，package 名字为 Banking，和 IDL 文件中的 library 的名字保持一致。

在 lib 目录下创建一个名字为 bank 的目录，将刚才生成的 Banking.tlb 文件复制到这个目录中。然后生成一个内容如下的 pkgIndex.tcl 的脚本文件：

```
package ifneeded Banking 1.0 [list source [file join $dir server.itcl]]
```

接着编写真正实现 COM 的 server.itcl 脚本，我们在 tcom 安装包提供的 server.tcl 文件基础上做了一些修改，其内容如下：

```
001 # $Id: server.itcl,v 1.7 2002/06/29 15:34:52 cthuang Exp $
002 # modified by leiyuhou to make-ready,send some bug.in 2006/3/17
003
```

```
004 package provide Banking 1.0
005
006 package require Itcl
007 package require tcom
008
009 ::tcom::import [file join [file dir $::info script]] Banking.tlb
010
011 itcl::class AccountImpl {
012     private variable balance 0 ;#帐号余额
013
014     public method _get_Balance {} {
015         return $balance
016     }
017
018     public method Deposit {amount} {
019         set balance [expr $balance + $amount]
020     }
021
022     public method Withdraw {amount} {
023         set balance [expr $balance - $amount]
024     }
025 }
026
027 itcl::class BankImpl {
028     public method CreateAccount {} {
029         set accountImpl [AccountImpl #auto]
030         return [::tcom::object create ::Banking::Account \
031             [itcl::code $accountImpl] [itcl::delete object]]
032     }
033 }
034
035 ::tcom::object registerfactory ::Banking::Bank \
036     {BankImpl #auto} [itcl::delete object]
```

我们这里使用了 ITcl 来实现组件。掌握其中有几个要点，我们就可以依样画葫芦，编写其它的组件了：

1. 第 4 行，表示我们实现了名字为 **Banking** 的 package；
2. 第 9 行，将 **Banking.tlb** 文件引入到默认得 **Banking** 名字空间中；



3. 第 11 行, 实现了名字为 `AccountImpl` 的类, 它实现了接口 `IAccount` 的方法和属性; 类的名字可是随意, 不一定是 `AccountImpl`, 你可以起一个其它的名字;
4. 第 14 行, `Itcl` 类方法 `_get_Balance` 实现了属性 `Balance` 的 `get` 操作; 这里要注意的是, 接口属性对应应在 `Itcl` 类中分别用一个 `method` 来实现其 `get` 和 `set` 操作, 并且 `method` 的名字是在属性名字前面加上 “`_get_`” 或者 “`_set_`” 字符串。例如这里属性为 `Balance`, 其对应的 `get` 操作的 `method` 的名字为 `_get_Balance`; 如果它可设置 (`writable`), 那么还应该有一个名字为 `_set_Balance` 的 `method` 来实现属性设置操作。因为该属性只读, 所以就没有 `set` 操作。
5. 第 18 行和 22 行分别用方法 `Deposit` 和 `Withdraw` 来实现接口 `IAccount` 中对应的同名方法。
6. 第 27 行定义接口 `IBank` 的实现类 `BankImpl`, 不过其方法 `CreateAccount` 的实现比较怪异。因为 `IBank` 接口的方法 `CreateAccount` 本身有其特殊之处: 它应该创建一个新的 `IAccount` 接口并且返回。如何在 TCL 中创建接口?
  - a) 首先在 29 行创建 `AccountImpl` 的对象实例, 该实例对应一个 `IAccount` 接口;
  - b) 然后调用 `tcom::object create` 方法来创建接口, 它有三个参数:
    - i. `Tlb` 被引入后生成的接口对象创建命令, 这里是 `Banking::Account`;
    - ii. 接口实例对象, 这里使用 `itcl::code` 将刚才创建的 `AccountImpl` 实例引出;
    - iii. 删除该对象的时候应该执行的语句;
  - c) 将 `tcom::object create` 方法返回值作为接口方法 `CreateAccount` 的返回值即可。
7. 第 35 行注册对象工厂。因为我们不希望客户可以直接创建 `Account` 对象, 所以我们只注册 `Bank` 对象。这里通过 `::tcom::object registerfactory` 来实现, 它也有三个参数, 格式和刚才使用的 `tcom::object create` 类似。分别是对应的命令名字、创建对象需要执行的脚本, 以及删除对象的时候需要执行的脚本。

至此我们的 COM 组件就基本上全部实现了。这里是采用面向对象的 ITcl 来实现, 实际上不用 ITcl 也可以, 我将经过我修改和注释过的非 ITcl 代码放在这里, 请大家自行研究:

```
001 # $Id: server.tcl,v 1.4 2003/03/07 00:03:00 cthuang Exp $
002 # Modified by leiyuhou , add some comment for ready
003 package provide Banking 1.0
004
005 package require tcom
006 ::tcom::import [file join [file dirname [info script]] Banking.tlb]
007
008 #COM 类 Account 的实现, args 是其所有方法的参数列表
009 proc accountImpl {method args} {
010     global balance    ;#全局变量, 帐户余额
011
012     switch -- $method {
```

```
013     _get_Balance {
014         #实现 IAccount 接口属性 Balance 的 get 操作
015         return $balance
016     }
017     Deposit {
018         #实现 IAccount 接口方法 Deposit
019         set amount [lindex $args 0] ;#取得方法参数
020         set balance [expr $balance + $amount]
021     }
022     Withdraw {
023         #实现 IAccount 接口方法 Withdraw
024         set amount [lindex $args 0]
025         set balance [expr $balance - $amount]
026     }
027     default {
028         error "unknown method $method $args" ;#出错
029     }
030 }
031 }
032
033 #COM 类 Bank 对应的实现
034 proc bankImpl {method args} {
035     global balance
036
037     switch -- $method {
038         CreateAccount {
039             set balance 0 ;#帐号余额初始化为 0
040             #这里 tcom::object create 只有两个参数:
041             #第 1 个为 import 后接口类对应的方法名字
042             #第 2 个为接口 IAccount 对应的实现方法 accountImpl
043             return [::tcom::object create ::Banking::Account accountImpl]
044         }
045         default {
046             error "unknown method $method $args"
047         }
048     }
049 }
050
```

```
051 #只需要注册类 Bank 的工厂
```

```
052 ::tcom::object registerfactory ::Banking::Bank {list bankImpl}
```

上面的代码是从 tcom 安装包中取出来的，文件名为 server.tcl。可以看到非 ITcl 实现是采用过程定义来实现的，并且使用全局变量 balance 来保存帐户余额。这里存在一个重大的 bug，稍后我们可以看到。

## 注册服务器

COM 组件必须先注册再使用，Tcl 编写的组件也不例外。我们需要 Tcl 环境下来注册刚刚完成的组件，打开 TCL 交互执行环境：

```
% package require tcom
3.9
% tcom::server register Banking.tlb
%
```

至此组件就处于可以被使用的状态。

## 在客户端（JScript、Tcl 或者 C++）中使用

这个用 TCL 脚本写成的组件可以在任何支持 COM 客户端的语言环境下正常运作，包括 VBScript、JScript 脚本和 Visual C++、Delphi 等。

## 脚本中使用本组件

我们在 JScript 中来对组件进行简单的测试。先保证 lib\Banking 目录下的 pkgIndex.tcl 文件中 source 的是 server.itcl 文件，也就是说，我们首先测试 ITcl 实现。test.js 内容如下：

```
001 //test.js ;run with cmdline: cscript.exe test.js
002 //Author :leiyuhou . 2006/3/17
003
004 var bank = new ActiveXObject("Banking.bank"); //创建组件对象
005 var a1 = bank.CreateAccount() //创建两个帐号
006 var a2 = bank.CreateAccount()
007
008 a1.Deposit(200) //分别存钱 200,300
009 a2.Deposit(300)
```

```
010 WScript.Echo("a1.balance=",a1.balance," , a2.balance=",a2.balance)
011
012 a1.Withdraw(50) //分别取出 50 和 100
013 a2.Withdraw(100)
014 WScript.Echo("a1.balance=",a1.balance," , a2.balance=",a2.balance)
```

打开命令提示符窗口，运行命令行 `cscript test.js` 来执行这段 JScript 脚本，结果如下：

```
a1.balance= 200 , a2.balance= 300
a1.balance= 150 , a2.balance= 200
```

我们修改 `pkgIndex.tcl` 文件，将 `server.itcl` 修改成 `server.tcl`，使组件成为非 Itcl 版本。再次运行 `test.js` 脚本，结果如下：

```
a1.balance= 500 , a2.balance= 500
a1.balance= 350 , a2.balance= 350
```

可见在非 Itcl 版本中，多个帐号对象使用的是同一个 `balance` 变量来保存余额。这就造成了如上一个严重问题，`a1` 和 `a2` 是两个不同的帐号，但是余额总是相同的。我没有尝试去修改这个 bug，因为我觉得可能比较复杂，而且没有必要（ITcl 版本工作得相当好）。

在 Tcl 中使用本组件的方法和使用其它组件没有任何不同，例如：

```
% package require tcom
3.9
% set bank [:tcom::ref createobject "Banking.bank"] ;#创建对象
::tcom::handle0x0094A7B0
% set a1 [$bank CreateAccount] ;#创建帐号
::tcom::handle0x0094A420
% $a1 Deposit 200 ;#存入 200
% $a1 Withdraw 100 ;#取出 100
% $a1 Balance ;#查询
100
%
```

## Visual C++中使用本组件

如何在 Visual C++中使用 TCL 编写的 COM 组件？有了 Tlb 文件一切都好办了。我们在 Visual C++中创建一个简单不使用 MFC 的 Console 工程，工程名字为 `TclComTest`。其中最主要的源程序文件的内容如下：

```
001 // TclComTest.cpp : Defines the entry point for the console application.
```

```
002 // Author : LeiYuhou
003 #include "stdafx.h"
004
005 /*****
006 /* 组件是双接口，我们通过宏开关 USE_DISP 分别用两种方式来调用组件*/
007 *****/
008 #define USE_DISP
009
010 //=====
011 //这一部分使用 IDispatch 接口来调用组件的属性和方法
012 #ifdef USE_DISP
013 #import "D:\Tcl\lib\Banking\Banking.tlb" no_dual_interfaces, \
014         rename_namespace("BANKING")
015 void test()
016 {
017     BANKING::IBankPtr bank;
018     if( FAILED(bank.CreateInstance(__uuidof(BANKING::Bank))) )
019     {
020         printf("Create bank failed.\n");
021         return;
022     }
023
024     BANKING::IAccountPtr a1 = bank->CreateAccount();
025     a1->Deposit( 200 );
026     a1->Withdraw( 30 );
027     printf("balance of a1 = %d\n", a1->Balance);
028 }
029 #else
030 //=====
031 //这里使用原始接口定义的方式来调用组件的属性和方法
032 #include "D:\Tcl\lib\Banking\banking.h"
033 void test()
034 {
035     IBank* pbank = NULL;
036     HRESULT hr=CoCreateInstance(CLSID_Bank, NULL, CLSCTX_ALL,
037                                IID_IBank, (LPVOID*)&pbank);
038     if ( FAILED(hr) )
039     {
```

```
040     printf("Create bank failed.\n");
041     return;
042 }
043 IAccount* pa = NULL;
044 hr = pbank->CreateAccount( &pa );
045 if ( FAILED(hr) )
046 {
047     printf("Create account interface failed.\n");
048     return;
049 }
050 pa->Deposit( 400 );
051 pa->Withdraw( 20 );
052 long bal = 0;
053
054 printf("balance of pa = %d\n", (pa->get_Balance(&bal),bal) );
055 pa->Release();
056 pbank->Release();
057 }
058 #endif
059
060 int main(int argc, char* argv[])
061 {
062     CoInitialize( NULL );
063
064     test();
065     puts("Press Enter to exit.");
066     getchar();
067
068     return 0;
069 }
```

在 Visual C++中使用这个组件有两种方式选择：

1. 使用 IDispatch 接口来进行调用。上面代码 13 行我们使用 #import 指令将 banking.tlb 文件导入的时候，设置了属性 no\_dual\_interfaces，该属性保证我们后面的 test 函数中对组件的方法调用都是通过 IDispatch 的方法 Invoke 方法来进行。
2. 通过 virtual table 来进行。代码第 32 行包含了一个文件 banking.h。这个文件从哪里来的？是我们前面使用 MIDL 编译 tlb 文件的时候产生的。第 33 行定义的函数中则是通过 virtual table 来调用组件方法和属性。这种方式下，还要将 MIDL 编译

生成的文件 `banking_i.c` 加入到工程中一起编译连接。

第二种方法是不是有点复杂？事实上，也可以通过 `#import` 指令达到通过 `virtual table` 来调用组件方法的目的。只需要将 `#import` 指令后面的 `no_dual_interfaces` 属性去掉即可。关于 Visual C++ 的 `#import` 指令，有很多的属性可以选择，这里就不详细讲解了。有兴趣的兄弟们可以去参考 MSDN 中详细文档。

一般情况下，通过 `virtual table` 来进行调用要比 `IDispatch` 要快一些。但是对于这个 TCL 脚本写成的组件，性能就不在我们考虑之列，还是怎样用起来方便就怎样使用吧。

## IActiveScript 接口

IActiveScript 接口是 Microsoft 公司为 Windows 操作系统设计的通用脚本技术。在讲解什么是 IActiveScript 接口技术之前，先让大家看看一个例子：

## JScript 与 VBScript 混合编程

Microsoft 为 Windows Script 实现了两种脚本语言 JScript 和 VBScript。两种语言各有千秋，我们先看一个简单的脚本文件 `tcl_asx.wsf`，让这两种语言来一次亲密接触！

```
001 <?XML version="1.0" encoding="gb2312" ?>
002 <!-- 演示 VBScript 和 JScript 互相调用 -->
003 <package>
004     <job id="Notepad">
005         <?job debug="true"?>
006             <script language="VBScript">
007                 Function CreateNotepad()
008                     Set sh = CreateObject("WScript.Shell")
009                     sh.Run getAppName() '函数 getAppName 是 JScript 函数
010                     WScript.Sleep 200
011                     sh.AppActivate "Notepad"
012                     WScript.Sleep 100
013                     Set CreateNotepad = sh
014                 End Function
015             </script>
016             <script language="JScript">
017                 var sh = CreateNotepad() //这里调用一个 VBScript 函数
018                 sh.SendKeys("Hello,world!{ENTER}")
019                 sh.SendKeys("I come from VBScript and JScript.")
```

```
020         sh.SendKeys("%{F4}")
021
022         function getAppName()
023         {
024             return "Notepad"
025         }
026     </script>
027 </job>
028 </package>
```

后缀为“wsf”，但是怎么看起来像 XML 格式，它可以执行吗？没错！这是一个可以执行的脚本文件，wsf 就是“Windows Script file”的缩写。这里我不打算详细介绍 wsf 文件的语法，MSDN 中有详细介绍。

在命令行中执行如下命令就可以执行本脚本：

```
cscript.exe tcl_axs.wsf
```

文件中用 VBScript 和 JScript 两种语言共同完成了一个简单的功能：打开你机器上的记事本（notepad.exe），输入一些文字，然后试图关闭它。6—15 行使用 VBScript 写成，它定义了一个名字为 CreateNotepad 的函数，里面调用了 22 行用 JScript 定义的函数 getAppName；16—22 行用 JScript 写成，里面调用了 CreateNotepad 函数。

Wsf 文件中的脚本语句块必须放在如下的 XML 元素之中：

```
<Script Language="LANGUAGEName">
.....
</Script>
```

其属性 Language 必须给出脚本语言的名字，这个名字可不能乱给，刚才我们已经看到了“VBScript”和“JScript”，这是 Microsoft 提供的两个语言。TCL 语言来源于 Unix，并且开放源代码，可以用在这里吗？

## TCL for ActiveScript

TCL 可以吗？当然可以！有很多热心肠的人已经为 TCL 语言实现了这样的插件，这里推荐我使用的基于 tcom 实现的 TclScript。如果你已经下载了 tcom 3.9 版本，那么可以在下载到的压缩文件包看到一个子目录 lib\TclScript，按照如下步骤安装：

1. 将 lib\TclScript 目录及其包含的文件一起解压缩到 TCL 的 lib 目录中；
2. 执行该目录中下的 register.tcl 脚本，将脚本引擎注册到系统中。

TCL 用在 Windows Script 中的时候，语言名字为“TclScript”。我们在刚才的 tcl\_asx.tcl 文件第 27 行后面插入如下的代码：



```
028     <job id="Calc">
029         <script language="TclScript">
030             package require tcom      ;#开始写 TCL 脚本
031             set sh [::tcom::ref createobject "WScript.Shell"]
032             $sh Run "Calc"            ;#启动计算器程序
033             WScript Sleep 200
034             $sh AppActivate "Calc"    ;#激活这个程序
035             WScript Sleep 200
036             $sh SendKeys "1{+}1="    ;#输入按键消息 1+1=
037         </script>
038     </job>
```

这是一段 TclScript 脚本。回到命令行提示符，执行如下的命令行：

```
cscript.Exe  tcl_asx.wsf  //Job:Calc
```

可以看到“计算器”程序被启动，并且会被要求计算出“1+1”的值。

ActiveScript 是 Microsoft 公司提供的一种通用脚本语言机制，只要为任何一种脚本语言提供几个接口，那么这些脚本语言就可以被 IE, IIS 等脚本宿主来使用。目前已知 Python, Ruby 等脚本都已经实现了对应的接口。

tcom 为 TCL 实现的这个 ActiveScript 接口目前还不够成熟，例如象刚才 VBScript 和 JScript 的互相调用，对 TCL 就不管用：如果在刚才的 TCL 脚本块中调用一个 JScript 函数，会出现严重错误导致程序崩溃。不过这部分的代码是公开的，如果有兴趣可以将它修理完善一下。

## 通过 JScript/VBScript 来扩展 TCL

前面章节中我们介绍了如何使用 C 语言编写扩展包来扩展 TCL 的功能，在了解了 tcom 之后，另外一条扩展 TCL 的阳光大道展现在我们眼前，那就是：编写 COM 组件来扩展。和传统的 C/C++ 扩展方式相比，它具有如下特点：

1. 实现语言和工具方面，具有更多的选择。例如 VB, Delphi 甚至 VBScript 脚本，都可以编写 COM 组件；实现的组件有更加广泛的用途（不仅在 TCL 中可以使用，其他编程环境下都可以使用）。
2. 环境限制，目前只有在 Windows 操作系统上，才可以编写 COM 组件；
3. 接口数据类型具有限制。只能通过 IDispatch 接口支持的数据类型来设计接口，TCL 中常见的列表，数组都不能够使用。损失了一些灵活性。

下面我们就用 JScript 脚本语言来实现一个 COM 组件，来为 TCL 进行扩展。读者会发问了：什么？用 JScript 来编写 COM 组件！你没有发烧吧？当然没有，TCL 通过 tcom 都可以编写 COM 组件，JScript 为什么不可以呢？只不过 JScript 不是通过 tcom，而是通过

微软的 WSC（Windows Script Component）技术。

我们生成一个内容如下的文本文件，文件名为 gcd.wsc。

```
001 <?XML version="1.0" encoding="gb2312"?>
002 <package>
003 <?component error="true" debug="true"?>
004     <!--
005         描述：这是用 JScript 写成的一个组件，来扩展 TCL
006         作者：雷雨后
007     -->
008 <component id="scriptlet">
009     <registration
010         progid="JSCOM.Math"
011         description="用 JScript 实现的一个 Math 组件，完成常见计算"
012         version="1.0"
013         clsid="{52476339-C7D6-4C2A-9193-C396Ef53F97A}"/>
014
015     <public>
016         <property name="PI" get/>    <!--定义属性 PI-->
017         <method name="gcd">          <!--定义方法 gcd，有两个参数-->
018             <parameter name="a"/>
019             <parameter name="b"/>
020         </method>
021         <event name="OnError"/>      <!--定义事件-->
022     </public>
023
024     <script language="JScript">
025         <![CDATA[
026             function get_PI()
027             {
028                 return 3.1415926535897932
029             }
030
031             //计算两个数的最大公约数
032             function gcd(a,b)
033             {
034                 if (a<=0 || b<=0) {
035                     fireEvent("OnError") //触发错误事件
```

```
036         return 0
037     }
038     var t
039     if (a<b) {
040         t=a ; a=b ; b=t ; //交换两个数
041     }
042     if ((a%b)==0) {
043         return b;
044     } else {
045         return gcd(b,a%b); //递归调用
046     }
047 }
048 ]]>
049 </script>
050 </component>
051 </package>
```

如果你看着眼熟，那就对了。WSC 文件也是采用 XML 语法写成的，和 WSF 文件类似。该组件的接口中定义了三个元素：属性 PI、方法 gcd 和事件 OnError。该文件写好之后我们要先注册才能够使用，命令行下执行如下命令：

```
regsvr32.exe c:\yourpath\gcd.wsc
```

如果文件中存在语法错误，regsvr32 会弹出错误信息对话框提示你去修改。如果没有错误，则会弹出对话框提示注册成功。

如何使用呢？照常使用即可！下面我们来看看如何在 TCL 中使用该组件。启动 TCL 解释环境，在交互环境下执行：

```
% package require tcom
3.9
% set m [::tcom::ref createobject "JSCom.Math"] ;#创建这个 JScript 写成的组件
::tcom::handle0x00927B70
% puts "PI = [$m PI]" ;#得到属性 PI
PI = 3.14159265359
% puts "gcd(100 , 26) = [$m gcd 100 26]" ;#执行方法 gcd
gcd(100 , 26) = 2
% proc OnErrorEvent {args} {puts "Error occus:$args"} ;#定义事件响应函数
% tcom::bind $m OnErrorEvent ;#绑定事件
% puts "gcd(100 , -5) = [$m gcd 100 -5]" ;#触发事件
Error occus:OnError
```

```
gcd(100, -5) = 0  
%
```

一切都在我们预料之中，是不是很神奇？微软的 WSC 技术非常的直观简单，比起庞大复杂的 MFC 和 ATL，通过它来编写 COM 组件实在是一件让人心旷神怡的事情。所有这一切都是基于 COM 技术的。想要深入了解 WSC 技术，请参考 MSDN 中相关资料。这里不作深入的介绍。

有心的读者可能会想了：可不可以在 WSC 技术的基础上，使用 TCL 来开发 com 组件呢？原则上是完全可以的，我们只需要为 TCL 脚本实现 IActiveScript 等接口即可。但是目前一些为 TCL 开发的这些扩展接口插件都还不完备，如果有兴趣的话，可以在 tcom 的 TclScript 插件基础之上继续开发完善。



## 数据库编程

如果一提起数据库你就想起了 ODBC，那说明我们受 Microsoft 的影响实在是太深了！不过这也难怪，在 Windows 世界中访问各种商业数据库的最好的接口就是 ODBC。但是这里我们介绍的是如何在 TCL 脚本中来访问数据库，面对各种不同的数据库，我们可以有很多的解决方案。本章我们还会介绍一些使用广泛的开源数据库，以及通过 TCL 操作它们的方法。

### TCL+Odbc

ODBC 是一套接口标准，很多数据库厂商都实现了其引擎。甚至有些厂家在 Unix 操作系统上也实现了 ODBC 接口。

所谓引擎，就是一套函数库，这些函数库实现了一套标准的接口。在理论上，即使底层数据库发生更改（例如从 Access 更改为 FoxPro），使用 ODBC 接口编写的应用程序可以不作任何改动，就可以兼容新的继续运行。这听起来是非常诱人的，只不过不同数据库厂家实现的 ODBC 引擎功能上可能存在一些差异，有些支撑得更加完备，有些则差强人意。

为了能够让 TCL 语言可以通过 ODBC 引擎，必须使用扩展包 tclOdbc，可以从如下的网址下载最新版本：<http://sourceforge.net/projects/tclodbc/>。从这里可以下载到两个版本：for Unix 和 for Windows。我们以 Windows 作为例子，目前最新版本是 2.3.1。下载之后解压缩到某临时目录中，然后在命令提示符下面执行如下的命令即可完成安装：

```
C:\> tclsh.exe d:\tmp\setup.tcl
```

安装程序在 Tcl 的 lib 目录下创建子目录 tclodbc2.3，其中子目录 doc 包含了使用手册和接口说明，子目录 samples 中包含部分例子代码。

为了讲述方便，我们使用 Microsoft 的 Access 附带的数据库 Northwind.mdb 作为例子。这个数据库可以在 Office 安装目录下找到。tclodbc 命令不多，但是功能却足够强大，下面我们逐一看来。

### 操作数据源

ODBC 的数据源看起来神秘，其本质上却是一个文本格式的配置文件。下面就是一个比较典型的 ODBC 数据源文件的内容：

```
[ODBC]
DRIVER=Microsoft Access Driver (*.mdb)
```

```
UID=admin
UserCommitSync=Yes
Threads=3
SafeTransactions=0
PageTimeout=5
MaxScanRows=8
MaxBufferSize=2048
FIL=MS Access
DriverId=25
DefaultDir=D:\Program Files\Microsoft Office\OFFICE11\SAMPLES
DBQ=D:\Program Files\Microsoft Office\OFFICE11\SAMPLES\Northwind.mdb
```

它描述了一个数据库连接的各种属性值。其实，即使没有数据源文件，我们也能够连接数据库。我以前就很少使用数据源文件来连接数据库。

tclobc 提供了如下的几个方法来创建、配置和删除数据源文件，这几个命令都是命令 database configure 的子命令，其格式如下：

```
database configure cmd drivename attributes
```

drivename 是 ODBC 驱动程序的名字，attributes 则是属性列表。

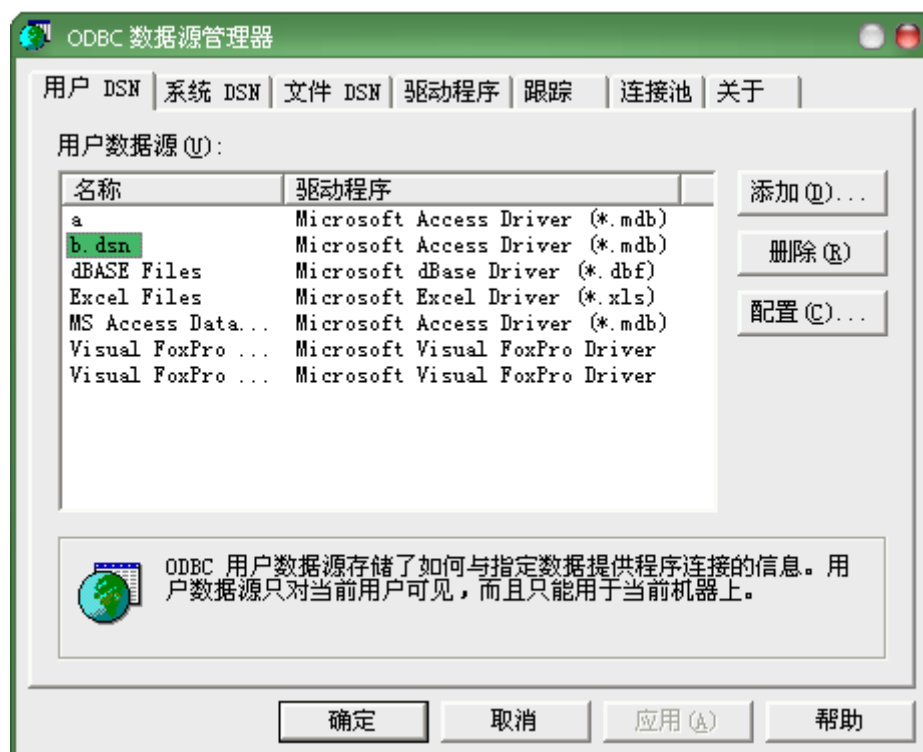
cmd 命令	功能说明
add_dsn	为当前用户增加一个数据源
config_dsn	配置当前用户的数据源
remove_dsn	删除当前用户的一个数据源
add_sys_dsn	为系统增加一个数据源
config_sys_dsn	配置某一个系统数据源的选项
remove_sys_dsn	删除某一个系统数据源

例如，下面的代码就增加一个用户数据源，该数据源和刚才那个数据源属性完全一致：

```
030 proc AddDataSrc {} {
031     #----数据库驱动程序的名字----#
032     set driver "Microsoft Access Driver (*.mdb)"
033
034     #----数据库连接的各种属性，其中名字为 b.dsn----#
035     set cfg [split {DSN=b.dsn
036         UID=admin
037         UserCommitSync=Yes
038         Threads=3
039         SafeTransactions=0
040         PageTimeout=5
```

```
041      MaxScanRows=8
042      MaxBufferSize=2048
043      FIL=MS Access
044      DriverId=25
045      DefaultDir=D:\SAMPLES
046      DBQ=D:\SAMPLES\Northwind.mdb}
047      "\n"]
048
049      #----增加一个用户数据库连接----#
050      puts [database configure add_dsn $driver $cfg]
051 }
```

直接调用该方法, 就可以帮我们生成一个数据源文件, 打开 ODBC 配置程序即可看到:



tclobdc 还可以列举出系统中当前所有已经安装的驱动程序以及数据源名字, 这是由命令 database drivers 和 database datasources 来实现的, 请看如下两个过程:

```
001 package require tclobdc
002 package require Tclx
003
004 #列出所有的驱动程序以及其属性
```



```
005 proc ListAllDrivers {} {
006     set drvs [database drivers]
007     set i 1
008     puts "\nAll Drivers in system."
009     foreach d $drvs {
010         lassign $d name attrs    ;#驱动程序的名字和属性
011         puts "$i : $name"
012         foreach attr $attrs {    ;#打印出所有的属性
013             puts "    $attr"
014         }
015         incr i
016     }
017     return [expr $i-1]
018 }
019
020 #列出系统中当前所有的数据源以及属性
021 proc ListAllSources {} {
022     set dses [database datasources]
023     set i 1
024     puts "\nAll datasources in system."
025     foreach ds $dses {
026         lassign $ds sourcename drivename
027         puts "$i : $sourcename : $drivename"
028         incr i
029     }
030 }
```

命令 `database` 还有一个子命令 `connect`，用来连接数据库并且返回一个数据库对象，它有两种格式，语法分别如下：

```
database connect id datasource ?userid? ?password?
database connect id connectionstring
```

第一种是采用 ODBC 数据源文件的连接方式；第二种则是采用“连接字符串”的方式直接连接，所谓“连接字符串”就是用分号连接起来的多个“`attr=value`”形式的字符串。

参数 `id` 是数据库连接的名字，连接成功创建之后，该 `id` 就成了一个 TCL 命令，代表了一个 TCL 数据库连接对象，通过它我们可以执行查询等各种操作。请看例子：

```
059 proc OpenDb {} {
060     set cns "DRIVER=Microsoft Access Driver (*.mdb);\n"
```

```
061      UID=admin;\
062      UserCommitSync=Yes;\
063      Threads=3;\
064      SafeTransactions=0;\
065      PageTimeout=5;\
066      MaxScanRows=8;\
067      MaxBufferSize=2048;\
068      FIL=MS Access;\
069      DriverId=25;\
070      DefaultDir=D:\\;\
071      DBQ=D:\\Northwind.mdb"
072
073      #打开数据库连接对象，名字为 db
074      database connect db $cns
075      #执行 SQL 语句进行查询，返回记录集列表
076      set rows [db "Select 产品名称, 类别 ID from 产品 WHERE 单价>=50"]
077
078      #打印出查询的结果
079      foreach row $rows {
080          puts [join $row "\t"]
081      }
082 }
```

第 74 行打开了数据库连接，并且其句柄为 `db`；随后执行一个 `SELECT` 语句，返回了满足查询条件的记录集列表。

## 数据库对象接口

命令 `database connect` 执行成功后，`id` 就是一个数据库对象，同时也成了 TCL 解释器中的一个命令，在前一章节有简单的介绍。一个数据库对象句柄支持如下子命令：

子命令格式	功能说明
<code>&lt;sql-clause&gt; ?argtype? ?args?</code>	执行一个 SQL 语句，查询结果以列表形式返回
<code>disconnect</code>	断开数据库的连接
<code>set option value</code>	设置数据库连接的属性
<code>get option</code>	查询数据库连接的属性
<code>commit</code>	提交事务
<code>rollback</code>	回滚当前的事务

tables ?pattern?	查询能够和 pattern 相匹配的表
columns ?tablename?	返回指定表中的字段名字和属性
indexes tablename	返回指定表上的索引信息
typeinfo typeid	返回 typeid 指定的 SQL 类型的相关信息
statement id <sql> ?argtypes?	创建一个命令对象，命令名字为 id
eval proc <sql> ?argtypes? ?args?	执行 Sql 语句，并且针对每一记录执行 proc 命令
read arrayspect <sql> ?types? ?args?	执行 Sql 语句，并且将结果组织到数组变量中去

第一个命令刚才我们已经看过了。如果 sql 语句是一个 SELECT 查询，那么返回的就是记录的列表，列表每一个元素同时也是一个列表（除非返回的只有一列）。如果 sql 是更新（UPDATE）或者插入（INSERT）命令，那么返回就是受影响的记录数量。基本上我们只需要这一个命令就能够完成常见的 90% 的数据库操作了。

disconnect 命令用来断开数据库连接，删除 connect 创建的 id 对象和 TCL 命令。好比申请了内存就必须释放，数据库连接也是资源，最后也必须释放。

set option value 和 get option 用来设置查询连接属性值。option 可以为“autocommit”和“cursortype”等，表示“是否自动提交”和“游标类型”，具体请参考 tclodbc 的联机手册。

commit 和 rollback 则用来提交和回滚事务。tables、columns 和 indexes 用来查询数据库的表、字段和索引信息。

下面我们先介绍 eval 命令。请看例子：

```
071 #简单的打印每一行的记录
072 proc printRow {pductName catalog} {
073     puts "    $pductName , $catalog"
074 }
075
076 proc testEval {} {
077     global cns    ;#连接字符串
078     database connect db $cns    ;#连接数据库
079
080     puts "不使用参数列表: "
081     set sqlclaus "Select 产品名称,类别 ID from 产品 where 单价>50"
082     db eval printRow $sqlclaus
083
084     puts "使用参数列表: "
085     set sqlclaus "Select 产品名称,类别 ID from 产品 where 单价 > ?"
086     db eval printRow $sqlclaus 50.0
087
088     db disconnect    ;#关闭数据库连接
089 }
```

```
090
```

```
091 testEval
```

第 72 行定义函数 `printRow`，带有两个参数，它只是查询出来的两个字段值。该函数简单打印参数值。第 78 行建立数据库连接。第 82 行通过 `eval printRow` 来执行 SQL 查询，其本质就是先执行查询，得到返回的数据记录之后，然后针对每一行数据记录来调用过程 `printRow`，每一行的每一个字段依次作为参数传递给过程作为实参。

第 85 行的 SQL 语句和 81 行相比，差别在于将最后的 50 换成了一个问号。这就是所谓的 SQL 语句的参数。在 `eval` 执行该语句的时候，需要在最后补上一个实参，如 85 行。上面脚本的执行结果如下：

```
不使用参数列表:
```

```
鸡 ,6
```

```
墨鱼 ,8
```

```
桂花糕 ,3
```

```
鸭肉 ,6
```

```
绿茶 ,1
```

```
猪肉干 ,7
```

```
光明奶酪 ,4
```

```
使用参数列表:
```

```
鸡 ,6
```

```
墨鱼 ,8
```

```
桂花糕 ,3
```

```
鸭肉 ,6
```

```
绿茶 ,1
```

```
猪肉干 ,7
```

```
光明奶酪 ,4
```

再看看 `read arrayspec` 命令。该命令执行 SQL 查询，并且将结果组织到一个或者多个 TCL 数组中，从而为数据查询提供方便。请看例子：

```
091 proc testReadArray {} {
```

```
092     global cns      ;#连接字符串
```

```
093     database connect db $cns      ;#连接数据库
```

```
094
```

```
095     #----将记录内容读入到两个数组中----
```

```
096     db read {prod_name price} \
```

```
097         "select 产品 ID,产品名称,单价 from 产品 where 单价>80"
```

```
098     parray prod_name      ;#输出数组内容
```

```
099     parray price
```

```
100      puts "-----"
101
102      #---将记录内容读入到一个数组中---
103      unset prod_name price
104      db read {price} \
105          "select 产品 ID,产品名称,单价,单位数量 from 产品 where 单价>80"
106      parray price
107      db disconnect
108 }
```

执行命令 testReadArray 的输出结果如下:

```
prod_name(20) = 桂花糕
prod_name(29) = 鸭肉
prod_name(38) = 绿茶
prod_name(9)  = 鸡
price(20) = 81.0000
price(29) = 123.7900
price(38) = 263.5000
price(9)  = 97.0000
-----
price(20,产品名称) = 桂花糕
price(20,单价)    = 81.0000
price(20,单位数量) = 每箱 30 盒
price(29,产品名称) = 鸭肉
price(29,单价)    = 123.7900
price(29,单位数量) = 每袋 3 公斤
price(38,产品名称) = 绿茶
price(38,单价)    = 263.5000
price(38,单位数量) = 每箱 24 瓶
price(9,产品名称)  = 鸡
price(9,单价)      = 97.0000
price(9,单位数量)  = 每袋 500 克
```

第 96 行查询三个字段“产品 ID”、“产品名称”和“单价”，结果存放到两个数组中。从数组内容来看，prod\_name 和 price 都是以第一个查询字段“产品 ID”作为数组下表，并且分别依次以“产品名称”和“单价”的值作为数组对应的值。例如：prod\_name(9)表示产品 ID 为 9 的产品名称，而 price(9)则是产品 ID 为 9 的产品单价。

第 104 行的查询则不同，它查询了四个数据库字段，结果放到一个数组变量 price 中。

从数组 `price` 的内容来看，这是一个所谓的“二维数组”：第一维是“产品 ID”的各个记录的值，第二维则分别是 `SELECT` 语句后面的三个字段名字。例如 `price(9,单价)` 的值就是“产品 ID”等于 9 的产品的单价，`price(9,产品名称)` 是“产品 ID”等于 9 的产品名称。

`read` 命令是不是用起来很爽？但是不能够滥用：如果记录数太多，那么 `read` 将浪费大量的内存资源；另一方面，`read` 将数据读入数组之后，如果其他用户更改了数据，本地的数据不能够同步的更新；另外，`SELECT` 语句的第一个字段必须是表的主键或者具有不可重复的索引属性，否则数组中的数据可能不完全正确。

最后还有一个 `statement` 命令，我们放到下面的章节来介绍。

## 命令对象接口

命令 `dbhandle statement` 可以用来构造一个数据库命令对象。创建成功之后，该命令对象可以当作一个普通的 TCL 命令来调用，并且可以传入参数，请看例子：

```
111 proc testStatement {} {
112     global cns      ;#连接字符串
113     database connect db $cns      ;#连接数据库
114
115     #----定义数据库命令对象 getPrice，带有一个 CHAR 类型参数
116     db statement getPrice \
117         "SELECT 单价 from 产品 where 产品名称=?" CHAR
118
119     #----分别查询如下几个产品的单价----
120     foreach prod {苹果汁 牛奶 盐} {
121         set price [getPrice $prod]
122         puts "<$prod>的单价为 $price"
123     }
124     db disconnect
125 }
```

第 116 行定义了一个数据库命令 `getPrice`，它执行一个 `SELECT` 查询，根据产品名称查询对应的单价并且作为命令结果返回。第 120 行分别查询了三种商品的单价。上面代码执行的结果如下：

```
<苹果汁>的单价为 18.0000
<牛奶>的单价为 19.0000
<盐>的单价为 22.0000
```

定义一个数据库命令的语法如下：

dbhandle statement id <sql>|tables ?argtypedefs?

其中:

1. dbhandle 是已经创建好的数据库连接句柄, 也就是数据库对象;
2. statement 是子命令字, 必须原封不动的出现在这里, 表示需要创建一个命令对象;
3. id 是需要创建的命令名字, 不要和 TCL 中已有的命令重复, 避免不必要的混乱;
4. 随后则是一个 SQL 语句, 或者是一个表名;
5. 最后则是 SQL 语句中出现的参数类型类标; 可以是 CHAR、NUMERIC、DECIMAL、DOUBLE、DATE 等类型, 具体参考 tclOdbc 手册;

命令定义之后, 我们可以直接在脚本中调用该命令, 每次根据需要给出不同的参数。命令返回结果和 SQL 语句有关: 如果是一个 SELECT 语句或者表名, 那么返回的就是一个 TCL 列表; 如果是 UPDATE 等语句, 则返回一个整数表示多少条记录受到本次操作的影响 (这种说法比较别扭, 简单的例子: 修改了多少条记录, 或者删除了多少条记录)。

一个创建好的数据库命令有如下调用方式:

调用语法格式	说明
run ?args?	传入参数, 运行该命令并且返回执行结果。一般情况下, run 被省略。
execute ?args?	运行该命令, 但是不返回结果集; 结果可以通过另外一个子命令 fetch 来获取
fetch ?arrayName? ?columnNames?	execute 执行成功后, 通过 fetch 来获取各条记录内容, 可以直接返回, 也可以放入数组中
rowcount	命令最后一次执行成功后, 影响 (插入、删除或者更新) 的记录条数
columns ?attrib?	得到查询结果各列的属性
drop	删除这个命令
eval proc ?args?	执行命令, 并且将每一条记录作为参数传入 proc 来调用, 其执行机制和 database 的 eval 命令类似
read arrayspect ?args?	用 args 执行命令, 并且将结果放入到数组 arrayspect 中, 具体用法和 database 的 read array 完全一致

我们修改一下刚才的 testStatement 过程, 在 124 行前面加入如下的代码行, 来演示上面表格中的重要命令, 代码如下:

```
125      #---定义另外的一个数据库命令---
126      db statement getAttr \
127          "SELECT * FROM 产品 WHERE 产品名称=?" CHAR
128      getAttr execute "苹果汁" ;#No return
129      while {[set row [getAttr fetch]]!=[]} {
130          puts $row
131      }
```

```
132
133     #----fetch 到一个数组 row 中----
134     getAttr execute "牛奶"
135     unset row ; getAttr fetch row
136     parray row
137
138     #----得到影响到的 rowcount
139     puts "RowCount = [getAttr rowcount]"
140
141     getAttr drop      ;#删除掉这个 TCL 命令
```

这一段代码的执行结果如下：

```
1 苹果汁 1 1 每箱 24 瓶 18.0000 39 0 10 1
row(中止)      = 0
row(产品 ID)   = 2
row(产品名称)  = 牛奶
row(供应商 ID) = 1
row(再订购量) = 25
row(单价)      = 19.0000
row(单位数量)  = 每箱 24 瓶
row(库存量)    = 17
row(类别 ID)   = 1
row(订购量)    = 40
RowCount = -1
```

这里的 rowcount 返回记录数为-1，这是因为我们使用的 Access 驱动程序不支持这个特性：返回 SELECT 操作查询得到的记录数。少数驱动程序支持这一点。

## TCL+ADO

TclOdbc 基本上能够满足我们常见的数据库操作。但是它只封装了很少一部分的 Odbc 接口，如果要对数据库进行进一步细致的控制，就有一点无能为力了。不过没关系，在 Windows 操作系统下，我们还有另一种访问数据库的机制，那就是 ADO。我们可以简单的将 ADO 当作一系列的 COM 对象，通过它们提供的接口即可以实现对数据库的操作。

ADO 和 ODBC 相比，最大的优势在于：我们可以在各种脚本语言中直接调用 ADO 的接口方法和属性，而 ODBC 接口则是一套复杂的 C 函数。所以一般而言，使用 ADO 操作数据库的程序要简单得多，在脚本语言中尤其如此。虽然形式简单，但是丝毫不影响 ADO 的功能强大。目前很多商用数据库都提供了 OLE-DB 接口形式的驱动程序，这样就



可以通过 ADO 来访问和操作这些数据库。

既然 ADO 是一套自动化组件，那么 TCL+tcom 扩展包就应该可以使用它来操作数据库，没错！下面我们来看一个简单的例子：

```
145 package require tcom
146 proc ado_Select {} {
147     set mdbpath "d:\\Northwind.mdb"
148     set cnstr "Provider=Microsoft.Jet.OLEDB.4.0;\\
149         Data Source=$mdbpath;\\
150         Persist Security Info=False"
151
152     #创建数据库连接对象
153     set cnn [::tcom::ref createobject "ADODB.Connection"]
154     $cnn Open $cnstr      ;#打开连接，cnstr 用来描述连接属性
155
156     #执行一条 SQL 的 SELECT 语句，返回一个 RecordSet 对象
157     set rst [$cnn Execute "SELECT * FROM 产品 WHERE 单价>50"]
158
159     if {![ $rst EOF]} {
160         set flds [$rst Fields]
161         #每一个字段的标题
162         for {set i 0} {$i<[$flds Count]} {incr i} {
163             append ln [format "%-10s" [[$flds Item $i] Name]]
164         }
165         puts $ln      ;#打印标题
166     }
167
168     while {![ $rst EOF]} {
169         set flds [$rst Fields]
170         set ln ""
171         for {set i 0} {$i<[$flds Count]} {incr i} {
172             #找出该记录的每一个字段的值
173             append ln [format "%-10s" [[$flds Item $i] Value]]
174         }
175         puts $ln      ;#打印每一行的内容
176         $rst MoveNext
177     }
178 }
```

```
179      #关闭数据库连接对象
180      $conn Close
181 }
182
183 ado_Select
```

上面的代码中第 153 行，通过 `createobject` 创建了一个 `Connection` 对象，在 ADO 编程中，`Connection` 对象表示应用程序和数据库之间的连接，一般而言都是必不可少的。通过调用该对象的 `Open` 方法，来打开连接。然后通过方法 `Execute` 来执行了一条 `SELECT` 语句，该方法返回一个 `Recordset` 对象，表示查询回来的记录集。然后下面我们通过该对象的 `Fields` 属性，找出了每一个字段的标题，以及每一条记录的值。打印完毕之后，就调用 `Close` 方法来关闭 `Connection` 对象。

现在一般 Windows 操作系统上都安装了 ADO 对象组件，它提供的对象结构比较简单，但是每一个对象的接口方法都非常丰富。关于 ADO 编程，本身可以写成一本厚厚的书，这里不做详细的介绍，具体可以参考 MSDN 中相关的技术文档和手册。

## TCL+BerkeleyDB

BerkeleyDB 系出名门，从名字就可以看出来。但是它不是一个关系数据库，而是一个类似 `key-value` 形式的数据记录集操作引擎，一般情况下被用作嵌入式的内存数据库。其性能非常出众，在众多的嵌入式系统（比如电信设备等）中有广泛的应用。

这是其官方网站：<http://www.sleepycat.com>。现在 BerkeleyDB 由 SleepyCat Software 公司来进行维护，并且提供客户支持。笔者写此书时最新版本为 4.4。

和众多的 C/S 结构的数据库服务器不同，BerkeleyDB 提供了丰富的 API 接口，应用程序直接调用这些接口来操作数据库，并且数据库引擎被一起编译到应用程序中，运行时和应用程序位于同一进程中，从而避免了进程间通信的性能消耗。除此之外，它还具有如下的特点：

- 支持本地、进程内的数据存储；
- 为并发系统进行特殊粒度优化的锁，可以配置；
- 最高可支持 4G 字节的记录容量；表可以支持 256TB 容量；
- 支持事务处理，支持嵌套式事务；
- 支持热备份、冷备份；
- 支持多种数据库复制模型；
- 支持多种操作系统（Windows、Linux、BSD Unix、VxWorks 等）；
- 支持多种编程语言接口（C/C++、Java、Perl、Python、Ruby 和 TCL）；
- 支持内存受限的环境（可以小到 350KB）；

总的来看，Berkeley DB 是一个非常不错的数据库系统，根据官方资料，它已经在 Google, SUN 等公司获得了广泛的应用。下面我们就来介绍如何在 TCL 语言环境下，来

安装和使用这个数据库系统。

## 下载和安装

BerkleyDB 是开放源代码的，如果需要的话，我们要对其进行重新编译。下面分别以 Windows 操作系统和 Linux 操作系统为例子，来介绍 BerkeleyDB 的安装。首先到其官方网站上下载对应版本。

## Windows 操作系统下的安装

SleepyCat 提供了 msi 格式的安装程序，下载到本地后直接运行即可完成初步的安装。我这里安装到了目录“C:\program files\SleepyCat Software\Berkeley DB 4.4.20”目录下，后面用\$BerkDb来表示这个安装目录；安装完毕之后，还不能够直接在 TCL 环境中直接使用。必须进行一下设置。

首先，到 TCL 的 lib 目录下创建子目录 berkbd。我系统上目录为“d:\tcl\lib\berkbd”；进入\$BerkDb\bin 目录，将该目录下的文件 libdb\_tcl44.dll 复制到刚才创建的目录中；执行 tclsh，进入 TCL 交互式环境，然后执行如下 TCL 命令：

```
% pkg_mkIndex d:\tcl\lib\berkbd libdb_tcl*.dll
```

然后看看这个目录下是不是多了一个 pkgIndex.tcl 文件？接着执行如下命令，应该出现如下的结果：

```
% package require Db_tcl
4.4
% berkbd version -string
Sleepycat Software: Berkeley DB 4.4.20: (January 10, 2006)
```

至此为止，我们才能够在 TCL 中正常调用 BerkeleyDB。使用之前，只需要加载 Db\_tcl 这个扩展包即可。

有些时候，有必要重新编译这两个动态库。操作过程很简单：

1. 在 VC6.0 中打开\$BerkDb\db-4.4.20\build\_win32\Berkeley\_DB.dsw，然后选择 build。
2. 在 VC 中选择项目“db\_tcl files”，修改其编译属性，将 D:\tcl\include 加入到 Additional Include Directory 中，将 D:\tcl\lib 加入到额外的库目录中，然后 build 该项目。
3. 完成这两步，就可以看到在 build\_win32\release 目录下生成了两个动态链接库文件。这就是我们重新编译得到的 BerkeleyDB 的库文件。

如果要发布应用程序，可以将这两个库一起发布；不过要注意其使用许可，否则可能

会造成侵权。因为开放源代码的软件，不一定是免费的。

## Linux 操作系统下的安装

Linux 下的安装也足够简单。本人系统是 Redhat Linux 9.0, 有 ActiveTCL 8.4 版本安装在 /usr/local/ActiveTcl 目录中。从官方网站下载 BerkeleyDB 的最新版本的压缩包, 文件名字为 db-4.4.20.NC.zip, 放到自己的 \$HOME 目录中, 依次执行如下的命令:

```
# cd ~
# unzip db-4.4.20.NC.zip -d .
# cd db-4.4.20.NC/build_unix
# ../dist/configure --enable-tcl --with-tcl=/usr/local/ActiveTcl/lib
# make
# make install
```

在调用 configure 脚本进行配置的时候, 必须使用参数 --enable-tcl 和 --with-tcl, 否则我们期望的 TCL 扩展包不会被自动编译出来。--with-tcl 参数给出 tclConfig.sh 所在的目录。

至此为止, BerkeleyDb 已经安装到了 /usr/local/BerkeleyDB4.4 目录中, 该目录下现在有 bin, include 和 lib 等子目录。但是现在还不能够在 TCL 中使用, 继续运行如下的命令:

```
# cd /usr/local/ActiveTcl/lib      ;#进入 Tcl 的库目录
# mkdir berkdb                    ;#创建 berkdb 子目录, 并且进入该目录
# cd berkdb
# ln -s /usr/local/BerkeleyDB4.4/lib/libdb_tcl-4.4.so .
# tclsh
% pkg_mkIndex *.so                ;# 建立 pkgIndex.tcl 文件
% package require Db_tcl          ;# 加载 Db_tcl 扩展包
4.4
% berkdb version -string          ;# 查询版本信息
Sleepycat Software : Berkeley DB 4.4.20: (Feb 11 , 2006)
```

至此安装结束, 我们就可以在 TCL 解释器中来使用 BerkeleyDB 了。

## 使用介绍

BerkeleyDB 中没有“表 (Table)”的概念, 或者说一个数据库就是一个表。每一个数据库由两个字段 (Field) 组成: 下标 (key) 和数据 (data), 两者之间一般情况下都是一一对应, 通过设置可以在一个数据库中存在 key 重复的记录。

先看一个简单的例子，将数据写入到一个数据库中，然后根据一个 key 读出结果。

```
001 package require Db_tcl
002
003 proc test_simple {} {
004     #打开数据库，文件为 emp.db
005     set db [berkdb open -btree -create -- emp.db]
006     array set databuf {
007         1 LeiYuhou
008         2 Lily
009         3 TigerLei
010         4 WangWu
011     }
012
013     for {set i 1} {$i<=4} {incr i} {
014         $db put $i $databuf($i)    ;#将 4 条记录写入数据库
015     }
016     $db sync    ;#写入到文件
017
018     #得到 key=1 的记录数据
019     puts [$db get 1]
020
021     $db close    ;#关闭数据库
022 }
023
024 test_simple
```

上面代码中：

在第 5 行调用 `berkdb open` 命令打开一个数据库，数据库句柄放在变量 `db` 中，`open` 成功之后返回的数据库句柄实际上是一个 TCL 命令；

1. 第 6 行定义数组 `databuf`，里面四条记录。每条记录的下标为数字，内容为字符串；
2. 第 14 行则调用 `put` 命令，将下标和对应的数据写入到数据库中。
3. 第 16 行调用 `sync` 命令将数据写入到文件；
4. 第 19 行调用 `get` 命令从数据库中取出数据，参数为 1 表示取出 `key=1` 的数据记录。  
`get` 返回的是一个 `{key value}` 的字符串。最后 21 行调用 `close` 关闭数据库。

BerkeleyDB 不是关系数据库，所以也不支持 SQL 语句，也不能够使用 `SELECT` 了，那么如何在一个数据库中查找符合要求的记录呢？

**TCL+SQLite**

**TCL+MySQL**

**TCL+XML**

**TCL+文本文件**



## 虚拟文件系统





## 多线程

## Socket 编程

## Ftp、Http、SNMP 等网络编程

## Tk 和 GUI