第一部分　Linux GUI 编程框架及编程基础

第1章　Linux软件开发概述

1.1　关于Linux

Linux于1991年诞生于芬兰。大学生Linus Torvalds，由于没有足够的钱购买昂贵的商用操作系统，于是自己编写了一个小的操作系统内核，这就是 Linux的前身。Linus Torvalds将操作系统的源代码在Internet上公布，受到了计算机爱好者的热烈欢迎。各种各样的计算机高手不断地为它添加新的特性，并不断地提高它的稳定性。　1994年，Linux 1.0正式发布。现在，Linux已经成为一个功能强劲的32位的操作系统。

严格地说，Linux只是一个操作系统内核。比较正式的称呼是　GNU操作系统，它使用Linux内核。GNU的意思是GNU’s not Unix(GNU不是Unix)——一种诙谐的说法，意指GNU是一种类Unix的操作系统。GNU计划是由自由软件的创始人 Stallman在20世纪80年代提出的一个庞大的项目，目的是提供一个免费的类　Unix的操作系统以及在上面运行的应用程序。GNU项目在初期进展并不顺利，特别是操作系统内核方面。 Linux适时而出，由于它出色的性能，使它成为GNU项目的操作系统的内核。从此以后，GNU项目进展非常迅速：全世界的计算机高手已经为它贡献了非常多的应用程序和源代码。

Linux是遵从GPL协议的软件，也就是说，只要遵从 GPL协议，就可以免费得到它的软件和源代码，并对它进行自由地修改。然而，对一般用户来说，从 Internet或者其他途径获得这些源代码，然后对它们进行编译和安装是技术难度很高的工作。一些应用程序的安装也都非常复杂。因而，有一些公司如Red Hat、VA等开始介入Linux的业务。它们将Linux操作系统以及一些重要的应用程序打包，并提供较方便的安装界面。同时，还提供一些有偿的商业服务如技术支持等。这些公司所提供的产品一般称为 Linux的发布版本。目前比较著名的 Linux发布版本有以下几种：

Red Hat——最著名的Linux服务提供商，Intel、Dell等大公司都对其有较大投资，该公司前不久收购了开放源代码工具供应商Cygnus公司。

SlackWare——历史比较悠久，有一定的用户基础。

SUSE——在欧洲知名度较大。

TurboLinux——在亚洲，特别是日本用户较多。该公司在中国推出了　TurboLinux 4.0、4.02和6.0的中文版，汉化做得很出色。

Debain——完全由计算机爱好者和Linux社区的计算机高手维护的Linux发布版本。

Linux进入中国后，在我国计算机界引起了强烈的反响，最近两年，也出现了许多汉化的Linux发布版本，影响较大的有以下几种：

XteamLinux——北京冲浪平台公司推出的产品，中国第一套汉化的 Linux发布版本。

BluePoint——1999年底正式推出的产品，内核汉化技术颇受瞩目。

红旗Linux——中国科学院软件研究所和北大方正推出的Linux发布版本。

从本质上来说，上面所有发布版本使用的都是同样的内核(或者版本略有不同)，因而，它们在使用上基本上没有什么区别。但它们的安装界面不一样，所包含的应用程序也有所不同。

Linux之所以大受欢迎，不仅仅因为它是免费的，而且还有以下原因：

1) Linux是一个真正的抢占式多任务、多线程、多用户的操作系统。

2) Linux性能非常稳定，功能强劲，可以与最新的商用操作系统媲美。

3) Linux有非常广泛的平台适应性。它在基于Intel公司的x86(也包括AMD、Cyrix、IDT)的计算机、基于Alpha的计算机，以及苹果、Sun、SGI等公司的计算机上都有相应的发布版本，甚至在AS/400这样的机器上都能找到相应的版本。 Linux还可以在许多PDA和掌上电脑以及嵌入式设备上运行。

4) 已有非常多的应用程序可以在Linux上运行，大多数为SCO Unix开发的应用程序都能在Linux上运行(借助于iBCS软件包)，甚至还比在SCO Unix上运行速度更快。借助Dosemu，可以运行许多DOS应用程序，而借助Wabi或Wine，还可以运行许多为Windows设计的软件。

5) Linux是公开源代码的，也就是说，不用担心某公司会在系统中留下后门 (软件开发商或程序员预留的，可以绕开正常安全机制进入系统的入口)。

6) 只要遵从GPL协议，就可以自由地对Linux进行修改和剪裁。

当然，Linux的优点决不止于此。对计算机专业人员来说， Linux及其相关应用程序也是学习编程的绝好材料，因为这些软件都提供了完整的源代码。

Linux的出现为我国软件产业赶超世界先进水平提供了极好的机遇，也为我国软件产业反对微软的垄断提供了有力的武器。

1.2 关于Linux的桌面环境

目前使用Linux主要在于服务器端。在Internet上有很多服务器都在使用Linux。但是，一个操作系统要想得到普及，并占据一定的市场份额，必须要使非计算机专业人士都可以轻松掌握这种系统。而Linux作为一种类Unix操作系统，对它的操作一般都是通过复杂的Shell命令进行的。因而，应该有一种简便易学的图形用户接口(Graphics User Interface，GUI)，使用户使用鼠标就可以完成大多数工作。

在Linux中，GUI由以下几个部分组成：

- 窗口系统—组织显示屏上的图形输出并执行基本的文本和绘图功能。
- 窗口管理器—负责对窗口的操作(比如最小化、最大化、关闭按钮的形状，窗口边框外观等)以及输入焦点的管理。
- 工具包—带有明确定义的编程界面的常规库。
- 风格—指定应用程序的用户界面外观和行为。

在Linux发展的初期，众多的计算机专家为它贡献了多种图形用户接口，如 FVWM95、AfterStep等。这些接口模仿了Windows 95、Macintosh、NestStep、Amiga、Unix CDE等桌面环境。这些GUI在一定程度上来说只是其他图形接口的仿制品，不能提供优秀的操作系统所需要的特性。其后，自由软件社区的一批计算机专家开始了KDE项目(K Desktop Environment，K桌面环境)，目的是提供一个开放源代码的图形用户接口和开发环境。该项目取得了极大的

成功，KDE成为许多Linux发布版本的首选桌面环境。 GNU/Linux项目因此而得到蓬勃发展。但是，KDE是基于Troll Technologies公司的Qt库的。Qt库是一个跨平台的C++类库，可以用于多种Unix、Linux、Win32等操作系统。Qt并不是遵从GPL或LGPL协议的软件包。它的许可条件是：如果使用它的免费版本开发应用程序或程序库，则所开发的软件必须开放源代码；如果使用它的商用版本，则可以用以开发私有的商用软件。另外， Qt库是属于Troll公司的产品，一旦Troll公司破产，或者被收购，自由软件事业将受到严重打击。

1997年由墨西哥国立自治大学的 Miguel de Icaza领导的项目组开始了 Gnome开发计划。Gnome是GNU Network Object Model Environment(GNU，网络对象模型环境)的缩写。该计划的最初目的是创建一种基于应用程序对象的架构，类似于微软公司的 OLE和 COM技术。然而，随着项目的进展，项目的范围也迅速地扩大；项目开发过程中有数百名程序员加入进来，编写了成千上万行的源代码。该项目进展很快， 1998年发布了Gnome 1.0。目前的最新版本是于1999年10月发布的October Gnome。现在，Gnome已成为一个强劲的GUI应用程序开发框架，并且可以在任何一种Unix系统下运行。Gnome使用的图形库是Gtk+——最初为了编写GIMP而创建的一套构件库，它是基于 LGPL创建的，可以用它来开发开放源代码的自由软件，也可以开发不开放源代码的商用软件。 Gnome的界面与KDE的界面是类似的(Gnome的目的之一就是创建一套类似KDE的桌面环境)，熟悉KDE的用户无需学习就能够使用 Gnome。由于以上几个原因，Gnome已经成为大多数Linux发布版本的首选桌面环境。

由于Gnome项目的成功，1998年11月Qt库的开发者Troll公司宣布修改许可证协议， Qt库将成为自由软件。但是获取 Qt库的许可证很不方便，况且 Gnome的进展也很不错，因而，只要有可能，应该避免使用Qt库以及KDE。

从用户的角度看， Gnome是一个集成桌面环境和应用程序的套件。从程序员的角度看，它是一个应用程序开发框架(由数目众多的实用函数库组成)。即使用户不运行Gnome桌面环境，用Gnome编写的应用程序也可以正常运行，但是这些应用程序是可以很好地和 Gnome桌面环境集成的。 Gnome桌面环境包含文件管理器，它用于任务切换、启动程序以及放置其他程序的“ 面板”、“ 控制中心” (包括配置系统的程序以及一些小东西)等。这些程序在易用的图形界面背后隐藏了传统的 UNIX Shell。Gnome的开发结构使开发一致的、易用的和可互相操作的应用程序成为可能。

1.3 Linux系统中的软件开发

1.3.1 开发所使用的库

在Linux下开发GUI程序的首要问题是采用什么样的图形库。在 Linux的发展历史中曾经出现过多种图形库，但是由于自由软件的特点(没有技术方面的承诺)，使得无人继续对它们进行维护，或者其他方面的原因，这些库都已慢慢地被人遗忘了。

Gtk+(GIMP ToolKit，GIMP工具包)是一个用于创造图形用户接口的图形库。 Gtk+是基于LGPL授权的，因此可以用 Gtk+开发开放源码软件、自由软件，甚至商业的、非自由的软件，并且不需要为授权费或版权费花费一分钱。之所以被称为 GIMP工具包因为它最初用于开发“ 通用图片处理程序” (General Image Manipulation Program，GIMP)，但是 Gtk已在大量软件项目，包括Gnome中得到了广泛应用。 Gtk+是在Gdk(GIMP Drawing Kit，GIMP绘图包)的基

础上创建的。Gdk是对低级窗口函数的包装（对X window系统来说就是Xlib)。

读者可能会看到，在本书中既有GTK，又出现了Gtk+。一般用GTK代表软件包和共享库，用Gtk+代表GTK的图形构件集。

GTK的主要作者是：

Peter Mattis petm@xcf.berkeley.edu
Spencer Kimball spencer@xcf.berkeley.edu
Josh MacDonald jmacd@xcf.berkeley.edu

Gtk+图形库使用一系列称为“ 构件” 的对象来创建应用程序的图形用户接口。它提供了窗口、标签、命令按钮、开关按钮、检查按钮、无线按钮、框架、列表框、组合框、树、列表视图、笔记本、状态条等构件。可以用它们来构造非常丰富的用户界面。

在用Gtk+开发Gnome的过程中，由于实际需要，在上面的构件基础上，又开发了一些新构件。一般把这些构件称为 Gnome构件(与Gtk+构件相对应)。这些构件都是 Gtk+构件库的补充，它们提供了许多 Gtk+构件没有的功能。从本质上来说， Gtk+构件和Gnome构件是完全类似的东西。

GTK本质上是面向对象的应用程序编程接口 (API)。虽然完全是用C写成的，但它仍然是用类和回调函数(指向函数的指针)的方法实现的。

1.3.2 Gnome的开发结构

只使用 Gtk+构件也可以开发出优秀的 Linux应用程序，但是 Gnome构件，特别是GnomeApp、GnomeUIInfo等，使开发界面一致的应用程序变得更加容易。 Gnome的一些新特性，如popt参数分析，保存应用程序设置等也是 Gtk+构件所没有的。

Gnome的应用程序开发结构核心是一套库 ,都是由通用的ANSI C语言编写的，并且倾向于使用在类UNIX的系统上。其中涉及图形的库依赖于 XWindow系统。Gnome差不多对任何语言都提供了Gnome API接口，其中包括 Ada、Scheme、Python、 Perl、 Tom、Eiffel、Dylan以及Objective C等。至少有三种不同的C++封装。本书只介绍有关库的C语言接口，不过，对使用其他语言绑定的用户来说，它也很有用，因为从 C到其他语言之间的转换都是非常直接的。本书包含Gnome库1.0版本（包括兼容的bug补丁版，比如 1.0.9——所有 1.0.x版本都是兼容的)。

Gnome的开发架构包含以下一些内容：

1. 非Gnome 库

Gnome并不是从头开始的，它充分继承了自由软件的传统——其中许多内容来自于Gnome项目开始之前的一些函数库。其中一些库 Gnome应用程序开发架构的一部分，但是不属于Gnome库——我们称之为非Gnome库。可以在Gnome环境中使用这些库函数。主要有以下几种：

Glib Glib是Gnome的基础，它是一个C工具库，提供了创建和操作常用数据结构的实用函数。它也涉及到了可移植性问题，例如 ,许多系统缺乏snprintf()函数，但是glib包含了一个，称为g_snprintf()，它能保证在所有平台上使用，并且比 snprintf()更安全 (它总是将目标字符串以NULL结尾)。Gnome 1.0中使用glib的 1.2版本，可以和任何 1.2系列的glib一起工作(1.2.1、1.2.2，等等)。

Gtk+ Gtk+(GIMP Toolkit的缩写)，是在Gnome应用程序中使用的GUI工具包。Gtk+最初是为了设计GIMP而引入的(GNU 图片处理程序)，但是现在已变成通用的库。Gtk+依赖于glib。Gtk+包中包含了Gdk，它是对底层的 X Window系统库Xlib的简化。由于Gtk+使用了Gdk而不是直接调用Xlib，因此Gdk的移植版本允许Gtk+运行在不同于X 但只有相对较少的修改的窗口系统上。Gtk+和Gimp已经移植到了Win32平台(32位的Windows平台，包括Windows 95/98、Windows NT/2000)上。

对Gnome应用程序来说，Gtk+具有以下的特性：

1) 动态类型系统。

2) 用C语言编写的对象系统，可实现继承、类型检验，以及信号 /回调函数的基础结构。

3) 类型和对象系统不是特别针对GUI的。

4) GtkWidget对象使用对象系统,它定义了Gtk+的图形组件的使用接口。

5) 大量的GtkWidget子类(构件)。

Gnome在基本Gtk+构件集合的基础上添加了许多其他构件。 Gnome 1.0是在Gtk+ 1.2版本的基础上完成的。

ORBit ORBit是 一个用C开发的CORBA 2.2 ORB。和其他ORB相比，它短小精悍，但速度更快，同时还支持 C语言映射。ORBit是以一整套库函数的方式实现的。 CORBA，或称作通用对象请求中介构架(Common Object Request Broker Architecture)，是一套对象请求中介，或称为ORB的规范。一个ORB更类似于动态链接程序，但是它以对象的方式操作，而非子程序调用。在执行过程中，程序能够请求一个特定的对象服务； ORB可定位对象并且创建对象和程序连接。例如,一个电子邮件程序可以请求 addressbook对象，并且利用它查找人名。与动态链接库不同，CORBA可以在网络内很好地运行，并且允许不同编程语言和操作系统之间进行交互。如果熟悉Windows操作系统下的DCOM, 那么CORBA与之类似。

Imlib Imlib(图片库)提供一些例程，其中包括加载、存储、显示，以及定绘制各种流行的图像格式 (包括 GIF、JPEG、PNG以及TIFF)的函数。它包括两种版本： Xlib-only版本和基于Gdk的版本。Gnome使用 Gdk版本。

2. Gnome库

下面所介绍的库是Gnome-libs包的一部分，并且是专门为Gnome项目开发的。

libgnome libgnome是一些与图形用户接口无关的函数集合， Gnome应用程序可以调用其中的函数。它包含分析配置文件的代码，也包含与一些外部实用程序的接口，比如国际化编程接口 (通过GNU gettext包)、变量解析 (通过popt包)、声音编程接口 (通过Enlightenment Daemon，esound)等。Gnome-libs包考虑了与外部库之间的交互，因此程序员无需关心库的实现或可用性。

libgnomeui libgnomeui包含了与GUI相关的Gnome代码。它由为增强和扩展Gtk+功能而设计的构件组成。 Gnome构件通常使用用户接口策略，以提供更方便的 API函数(这样程序员需要指定的东西较少)。当然，这也让应用程序界面更一致。

libgnomeui主要包含：

1) GnomeApp构件 一般用来为应用程序创建主窗口。它使用 GnomeDock构件，允许用户重新排列工具栏，还可以将工具条从窗口上拖开。

2) GnomeCanvas构件 用来编写复杂的、无闪烁的定制构件。

3) Gnome 内置的pixmap(包括打开、关闭、保存以及其他操作的图标） 用于创建和使用对话框的例程。GnomePixmap构件比GtkPixmap功能更多。

libgnomeui 中还有几种其他构件，如 GnomeEntry、GnomeFilePicker等。这些构件都比Gtk+构件库中的构件功能更强，也更方便。

libgnorba libgnorba提供与CORBA相关的实用程序，包括安全机制和对象激活。对象激活是指获得实现给定接口对象的引用过程，它包括执行服务器程序，加载共享库模块，或为已有程序请求新的对象实例等。

libzvt 这个库包含一个终端构件(ZvtTerm)，可以在Gnome程序中使用它。

libart_lgp 这个库包含由Raph Levien编写的图形绘制例程。在这里包含的是在 LGPL许可下发布的，用在GnomeCanvas构件中的，Raph Levien也销售它的增强版本。实质上它是一个矢量图形光栅图形库，功能类似于 PostScript语言。

3. 其他库

这些库一般使用在Gnome应用程序中，但它不是Gnome-libs 专属的部分。

Gnome-print Gnome-print目前还是实验性的，但是非常有前途。它使用 libart_lgpl库，可以和GnomeCanvas一起工作得很好。它提供一个虚拟输出设备 (称“ 打印上下文”)，因此一段代码能输出到一个打印预览构件或 PostScript文件，还可以输出到其他打印机格式。Gnome-print也包含与打印相关的GUI元素,例如打印设置对话框、虚拟字体接口 (处理 X字体不可打印的问题)。

Gnome-xml Gnome-xml是还未经验证的XML引擎，它由WWW协会的Daniel Veillard编写。它能按照树状结构分析 XML，也能按照XML输出树状结构。它对任何需要加载和保存结构化数据的应用程序来说是很有用的，许多 Gnome应用程序把它作为文件格式使用。这个库不依赖于任何其他的库(甚至 glib)，所以它只是在名义上是一个 Gnome库。然而，可以认为大多数Gnome用户都安装了它，因此如果应用程序使用了这个库，对用户来说也没有什么不方便。

Guile Guile是Scheme编程语言在一个库中的实现，它使任何应用程序都能带有一个嵌入式的Sheme解释器。它是GNU项目的正式扩展语言，并且有一些 Gnome应用程序也使用它。为应用程序添加扩展语言听起来挺复杂，但是有了 Guile后就微不足道了。一些 Gnome应用程序也支持 Perl和Python，一旦实现了应用程序，同时支持几种语言就会变得很容易。 Guile在Gnome开发者心目中有着特殊的地位。

Bonobo Bonobo是一种对象嵌入式结构，类似于 Microsoft的OLE。例如，它允许你在电子表格中嵌入图表。它将在 Gnome中普遍使用。任何应用程序将能通过适当的 Bonobo组件调用Gnome库，显示 MIME类型数据，例如纯文本、 HTML或图像。

1.4 开发Linux应用程序的编程语言和编程工具

Linux是一种类Unix的操作系统。传统 Unix下的开发语言是 C语言。因为C语言是平台适应性最强的语言，差不多每种平台上都会有一个 C编译器。 C语言也更易移植，因而，在Linux下编程的最佳语言应该是 C语言，Linux上的很多应用程序就是用 C语言写的。当然，也可以使用其他语言。

因为Gtk+和Gnome是用C语言编写的，所以在开发 Linux下的GUI程序时使用C语言是非常

方便的。但是 Gtk+也提供与许多其他语言的接口，如 Ada、Scheme、Python、Perl、Tom、Eiffel、Dylan以及Objective C等。如果用C++语言开发基于Gtk+应用程序，可以使用一个名为Gtk--的函数库，它是GTK工具包的C++风格的封装。如果要用 Gtk+库和其他语言，最好参考相应的文档。本书只介绍使用 C语言开发Linux程序。

一般的Linux发布版本中都提供了 C编译器gcc或egcs。使用gcc或egcs可以编译C和C++源代码，编译出的目标代码质量非常好，编译速度也很快。

各种C编译器都要使用一些C语言实用函数。为了保证程序的可移植性， gcc没有使用通用的C函数库，而是使用一种称为 glib的函数库。 glib也是Gtk+的基础。它包含一些标准函数的替代函数(如字符串处理函数)和基本数据结构的实现(单向链表、双向链表、树、哈希表等)。glib中所包含的函数消除了某些函数的安全漏洞，使其更加可靠，在不同平台上移植也更加方便。

还有许多使用工具可以提高Linux下的编程效率，如gdb是优秀的C语言调试器，有非常丰富的调试指令； automake和autoconf用于由源代码结构配置编译选项，生成编译所需的Makefile文件。

到在目前为止，还没有像Windows平台上的Visual Basic、Delphi等一样的可视化的快速应用程序开发工具。开发 Linux应用需要用文本编辑器书写源代码，然后再用编译器生成应用程序。眼下有一个开发小组正致力于开发一个Linux下的类似于Visual Basic的开发工具——gBasic，另外，预计Inprise公司(即Borland)的Delphi for Linux也即将面市。

有几种正在开发的RAD(Rapid Application Development)工具，其中最有希望的是 Glade——一种GUI生成器，可以快速生成创建界面的C源程序。

1.5 本书的结构

本书包含了以下内容：

- Gnome应用程序开发的框架。
- 开发Linux应用程序的方法和步骤。
- glib，Gtk+/Gnome构件的使用方法。
- GDK基础知识。
- Linux常用编程工具：调试工具gdb，GUI生成器Glade。

本书附录包含 Gtk+和Gnome对象介绍(每个对象有一个简短描述)，还有一些在线编程资源。

本书提供了大量可供参考的源代码。如果觉得内容不够充分，请参考相应的头文件。实际上，Gtk+/Gnome和glib的头文件都是非常简单易懂的，从函数名称就可以猜到它的用处和用法。如果还觉得不够，可以钻研它们的源代码，这对了解它们的实现方法也是极有好处的。

第2章 Gtk+/Gnome开发简介

2.1 安装Gtk+/Gnome库

要想用Gtk+/Gnome编程，首先要保证系统中已经安装了 Gtk+和Gnome库。

一般情况下，Linux发布版本的光盘中都应该包含了所需要的库的源代码。例如 Red Hat Linux 6.0/6.1和TurboLinux 4.0中都有Gtk+和Gnome的最新版本。在安装系统的过程中，当提示安装类型时，一般有“ 服务器”、“ 工作站”、“ 开发工作站”、“ 自定义” 和“ 完全安装” 等几个选项。选择“ 开发工作站” 或“ 完全安装”，完全安装大多数用于软件开发的库、头文件和实用程序，如Gtk+库、Gnome库、automake、autoconfig、gcc编译器、gdb调试器等。

如果觉得系统中已有的库的版本已经过时，可以从 Internet上下载最新版本，然后安装。可以从Gtk+的Web站点http://www.gtk.org下载最新版本的Gtk+。Gtk+所使用的版本号与Linux的版本编号方法类似，偶数版本号（如 1.0和1.2）表示稳定的版本，而奇数版本号（如 0.9和1.1）表示正在开发的版本。有时还增加一个附加数字表示对这一版本进行了修正，如 1.2.1。当前Gtk的最新版本是1.2.3。

从网上下载的文件名一般是 gtk+1.2.3.tar.gz或者其他类似形式，文件名中包含了该软件包的名称和版本号信息。因为它的后缀是 .tar.gz，所以它是一个归档的压缩文件。用 gunzip命令对它解压缩：

```
gunzip gtk+1.2.3.tar.gz
```

将会产生一个解压缩的以.tar结尾的归档文件。用tar命令将它扩展为它的目录结构：

```
tar -xvf gtk+1.2.3.tar
```

这个命令建立了建库所需要的目录结构。进入上面建立的目录，执行 configure脚本生成编译所需的makefile：

```
./configure
```

下面可以建立库了。输入make命令：

```
make
```

建库后，需要安装刚才建立的库。输入以下命令：

```
make install
```

然后，还需要运行/sbin/ldconfig以使Gtk+能正常工作。

当然，完成上面工作的前提是系统上已经安装了 glib。不过，一般情况下都可以保证做到这一点。如果需要安装新版本的 glib库，操作步骤和安装Gtk+库一样。

在Gtk+源代码目录下的examples子目录下，有很多 Gtk+构件示例。这些代码都有很详细的注释，通读这些代码对学习 Gtk+编程也是很有帮助的。本书中关于 Gtk+构件的演示代码都来自这些示例。

Gnome的最新版本可以从 http://www.gnome.org下载。取得新版本软件后，解压缩和安装的方法与Gtk+类似。

2.2 第一个Gtk+应用程序

2.2.1 一个什么也不能做的窗口

用Gtk+库编程和同时使用Gtk+/Gnome库编程的方法完全相同，只是细节上略有不同。我们将从一个最简单的程序开始介绍GTK，然后用一个简单的例子介绍Gnome库编程的方法。

本程序将创建一个200× 200像素的窗口，除了用shell命令kill以外，没有其他的退出程序的方法。

```
/* 例子开始 base.c */
#include <gtk/gtk.h>
int main( int argc, char *argv[] )
{
GtkWidget *window;
gtk_init (&argc, &argv);
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_show (window);
gtk_main ();
return(0);}
/*示例结束 */
```

上面的源代码中的/*和*/之间的内容都是注释。Linux中常用的C编译器能够识别/* */和//两种格式的注释。

可以用gcc编译上述程序：

```
gcc base.c -o base `gtk-config --cflags --libs`
```

注意，gtk-config --cflags--libs选项左右的符号不是单引号，而是反引号（键盘上“ 1”键左边的键）。编译选项的含义选项将在下面编译“ Hello World”时解释。

编译结束后，输入以下命令来执行上面创建的应用程序(结果如图2-1所示)：

```
./base
```

图2-1 第一个Gtk应用程序，它什么也不能做

2.2.2 示例代码的含义

在程序的源代码中，所有用到的函数和数据类型以及结构等都应该声明。也就是说，如果使用了一个按钮构件，应该包含按钮所对应的gtkbutton.h头文件。如果在程序中使用了多种构件和函数库，包含这些头文件将会是一件很麻烦的事，也很容易出错。有一个特殊的头文件，gtk/gtk.h，其中包含了Gtk+编程中所有需要的头文件，也包含gdk.h和glib.h等。在源文件的前面包含这个文件就可以了。后面会看到，如果要用到 Gnome的构件和库函数，包含gnome.h就可以了。在gnome.h中已经包含了gtk.h文件和其他相关头文件。

第一行 #include <gtk/gtk.h> 包含了所有需要的头文件。

下一行:

```
gtk_init (&argc, &argv);
```

先调用函数gtk_init(gint *argc, gchar ***argv)。所有GTK应用程序都要调用该函数。它为我们设置一些缺省值例如视觉和颜色映射，然后调用 gdk_init(gint *argc, gchar ***argv)。这

个函数将函数库初始化,设置缺省的信号处理函数，并检查通过命令行传递给应用程序的参数。

主要检查下面所列的一些值：

```
--gtk-module
--g-fatal-warnings
--gtk-debug
--gtk-no-debug
--gdk-debug
--gdk-no-debug
--display
--sync
--no-xshm
--name
--class
```

它从变量表中删除以上参数，并分离出所有不识别的值，应用程序可以分析或忽略这些值，由此生成一些Gtk应用程序可以接受的标准参数。

下面两行代码将创建和显示一个窗口：

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_show (window);
```

GTK_WINDOW_TOPLEVEL参数指明让窗口使用“窗口管理程序”指定的状态设置和位置布置。没有子窗口的窗口缺省设置为200× 200像素大小，而不是0× 0大小。gtk_widget_show()函数让 Gtk知道那我们已经设置该构件（窗口）的属性，现在可以显示它了。

最后一行代码进入 Gtk主处理循环：

```
gtk_main ();
```

gtk_main()是在每个Gtk应用程序都要调用的函数。当程序运行到这里时，Gtk将进入等待状态，等候 X事件(比如点击按钮或按下键盘的某个按键)、Timeout或文件输入/输出发生。

在这个简单例子里，所有事件都被忽略。用鼠标点击窗口右上角的“×”按钮也不能将窗口关闭。唯一关闭它的办法就是使用 Shell命令kill删除它的进程。

这里顺便指出：在代码中可以使用C语言风格的注释（/* */）和C++语言风格的注释（//）。不过，最好能够使用C语言风格注释，否则可能在某些平台上编译时会出现错误。

2.2.3 GTK的Hello World

上面的例子实在是太简陋了，它甚至什么也不能做。下面我们创建一个 Gtk版本的“Hello World”以进一步说明Gtk+的编程方法。在这个例子中，我们在窗口里面放一个按钮构件。下面是示例的源代码。

1. 源代码

```
/*示例开始 helloworld helloworld.c */
#include <gtk/gtk.h>
/*回调函数在本例中忽略了传递给程序的所有参数。下面是回调函数 */
void hello( GtkWidget *widget, gpointer data )
{
g_print ("Hello World\n");
```

```
}
gint delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
/*如果在"delete_event"处理程序中返回FALSE，GTK 将引发一个"destroy"
*信号，返回TRUE意味着你不想关闭窗口。
* 这些在弹出"你真的要退出?"对话框时很有作用*/
g_print ("delete event occurred\n");
/* 将TRUE改为FALSE，主窗口就会用一个"delete_event"信号，然后退出*/
return(TRUE);
}
/* 另一个回调函数*/
void destroy( GtkWidget *widget, gpointer data )
{
gtk_main_quit();
}
int main( int argc, char *argv[] )
{
/* GtkWidge是构件的存储类型*/
GtkWidget *window;
GtkWidget *button;
/* 在所有的Gtk应用程序中都应该调用。它的作用是解析由命令行传递
 * 进来的参数并将它返回给应用程序*/
    gtk_init(&argc, &argv);
/* 创建一个主窗口*/
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
/* 当给窗口一个"delete_event"信号时(这个信号是由窗口管
* 理器发出的，通常是在点击窗口标题条右边的"×"按钮，或
* 者在标题条的快捷菜单上选择"close"选项时发出的)，我们
* 要求调用上面定义的delete_event()函数传递给这个回调函数
* 的数据是NULL，回调函数会忽略这个参数*/
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete_event), NULL);
    /* 这里，我们给"destory"事件连接一个信号处理函数，
     * 当我们在窗口上调用gtk_widget_destroy()函数
     * 或者在"delete_event"事件的回调函数中返回FALSE
     * 时会发生这个事件*/
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (destroy), NULL);
    /* 设置窗口的边框宽度 */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    /* 创建一个标题为"Hello World"的按钮 */
    button = gtk_button_new_with_label ("Hello World");
    /* 当按钮接收到"clicked"时，它会调用hello()函数，
     * 传递的参数为NULL。函数hello()是在上面定义的 */
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (hello), NULL);
    /* 当点击按钮时，通过调用gtk_widget_destroy(window)函数销毁窗口。
     * 另外，"destory"信号可以从这里发出，也可以来自于窗口管理器*/
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC (gtk_widget_destroy),
                               GTK_OBJECT (window));
```

```
    /* 将按钮组装到窗口中（一个gtk容器中）*/
    gtk_container_add (GTK_CONTAINER (window), button);
    /* 最后一步就是显示新创建的构件 */
    gtk_widget_show (button);
 /* 显示窗口 */
    gtk_widget_show (window);
    /* 所有的GTK应用程序都应该有一个gtk_main()函数。
* 程序的控制权停在这里并等着事件的发生（比如一次按键或鼠标事件）*/
gtk_main ();
return(0);}
/* 示例结束*/
```

2. 编译“ Hello World” 应用程序

用以下语法编译：

```
gcc -Wall -g helloworld.c -o helloworld `gtk-config --cflags` \
    `gtk-config --libs`
```

上面使用了Gtk自带的gtk-config程序。这个程序“ 知道” 用Gtk编译应用程序时需要使用什么编译程序开关选项。

gtk-config --cflags将输出一个包含编译器需要的查找路径的列表，并且 gtk-config --libs将输出编译程序要链接的库的列表和找到这些库的路径。在上面的例子中，这些选项可以写在一句话里，比如 `gtk-config --cflags --libs`。

编译中通常要链接的库包含下面这些：

GTK库 (-lgtk)：构件库,建立在Gdk的基础上。

GDK库 (-lgdk)：包装的Xlib库。

gmodule库 (-lgmodule)：用于在加载运行时扩展函数。

glib库 (-lglib)：包含了各种实用函数。在这里仅仅使用了 g_print()。Gtk是在Glib的基础上创建的，所以需要使用glib库。

Xlib库 (-lX11)：GDK库要调用该库。

Xext库 (-lXext)：包含为共享内存中的pixmaps图片和其他X扩展的代码。

数学库 (-lm)：被GTK调用。

2.2.4 Gtk+的信号和回调函数原理

在详述helloworld的细节之前，我们先讨论信号和回调函数。GTK是一种事件驱动工具包，这意味着它将在gtk_main函数中一直等待，直到事件发生和控制权被传递给相应的函数。

1. 信号（signal）

控制权的传递是使用“ 信号” 的方法。一旦事件发生，比如鼠标器按钮被按下，被按下的构件（按钮）将引发适当的信号。这是GTK实现其绝大多数工作的方法。

有一些信号是大多数构件都具备的，比如 destory，还有一些是某些构件专有的，比如在开关按钮（togglebutton）的toggled信号。

要让一个按钮执行一个操作，我们需要建立一段信号处理程序，以捕获它的信号，然后调用相应的函数。这由类似以下所示的函数实现：

```
gint gtk_signal_connect( GtkObject *object, gchar *name,
                         GtkSignalFunc func, gpointer func_data );
```

第一个参数*object是将要发出信号的构件，第二个参数 *name是希望捕获的信号的名称，第三个参数func是捕获信号时要调用的函数，第四个参数 func_data是要传递给函数的用户数据参数。

在第三个参数里指定的函数称为“ 回调函数” ,它的形式通常是：

```
void callback_func( GtkWidget *widget, gpointer callback_data );
```

这个函数的第一个参数是一个指向发出信号的构件的指针，第二个参数是一个指向传递给回调函数的用户数据的指针。

注意 上述对“ 信号”的回调函数的声明仅仅是一个通用的规则，因为一些构件的特殊“ 信号”产生不同调用参数。例如, Clist构件的select_row构件同时提供行和列参数。

另一个在 helloworld例子里使用的函数调用是：

```
gint gtk_signal_connect_object( GtkObject *object,
                                gchar *name,
                                GtkSignalFunc func,
                                GtkObject *slot_object );
```

gtk_signal_connect_object()跟gtk_signal_connect()一样，除了回调函数仅仅使用一个参数：指向GTK对象的指针。

这样，当用这个函数连接到一个信号时，回调函数将以下面的形式出现：

```
void callback_func( GtkObject *object);
```

在这里，参数object通常是一个构件。

不过，我们通常不为gtk_signal_connect_object()设置回调函数。它们通常用于调用接受信号或对象作为参数的Gtk函数，就像在helloworld里的那样。有两种函数能连接到信号的目的仅仅是允许回调函数可以有不同数量的参数。

因为在GTK库里许多函数仅仅接受单个 GtkWidget指针作为参数，所以可能想用gtk_signal_connect_object()连接到某些回调函数。在这种情况下，需要提供附加的数据传递到回调函数。

2. 事件

除了上面描述的信号机制之外，还有一套与 X事件（X Window中发生的动作，比如鼠标某个按键按下或弹起，鼠标移动等）机制相对应的事件。回调函数也可以与这些事件连接起来。

这些事件是:

button_press_event	button_release_event
motion_notify_event	delete_event
destroy_event	expose_event
key_press_event	key_release_event
enter_notify_event	leave_notify_event
configure_event	focus_in_event
focus_out_event	map_event unmap

要将回调函数连接到上面的某一个事件，需要使用 gtk_signal_connect函数，并使用上面的事件名称作为命名参数。事件的回调函数与信号的回调函数在形式上略有不同：

```
Void func(
     GtkWidget *widget, GdkEvent *event,
     gpointer callback_data );
```

GdkEvent是 C中的联合体结构，其类型依赖于发生的事件是哪一个事件。要想知道哪一个事件已经引发，可以查看类型参数，因为每个可能的可选事件都有一个反映引发事件的类型参数。

事件结构的另一个组件依赖于事件类型。事件类型的可能取值有：

GDK_NOTHING	GDK_DELETE	GDK_DESTROY
GDK_EXPOSE	GDK_MOTION_NOTIFY	GDK_BUTTON_PRESS
GDK_2BUTTON_PRESS	GDK_3BUTTON_PRESS	GDK_BUTTON_RELEASE
GDK_KEY_PRESS	GDK_KEY_RELEASE	GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY	GDK_FOCUS_CHANGE	GDK_CONFIGURE
GDK_MAP	GDK_UNMAP	GDK_PROPERTY_NOTIFY
GDK_SELECTION_CLEAR	GDK_SELECTION_REQUEST	GDK_SELECTION_NOTIFY
GDK_PROXIMITY_IN	GDK_PROXIMITY_OUT	GDK_DRAG_BEGIN
GDK_DRAG_REQUEST	GDK_DROP_ENTER	GDK_DROP_LEAVE
GDK_DROP_DATA_AVAIL	GDK_CLIENT_EVENT	GDK_VISIBILITY_NOTIFY
GDK_NO_EXPOSE		

GDK_OTHER_EVENT /* 最好不要使用它 */

因此，要将回调函数与一个事件连接起来，需要使用以下形式的函数：

```
gtk_signal_connect( GTK_OBJECT(button),
    "button_press_event",
    GTK_SIGNAL_FUNC(button_press_callback),NULL);
```

这里假定button是一个按钮构件。现在，当鼠标移动到按钮上方，鼠标按钮按下时， 将调用button_press_callback函数。

这个回调函数可以作如下声明：

```
static gint button_press_callback( GtkWidget *widget,
         GdkEventButton *event, gpointer data);
```

注意，我们可以将第二个参数声明为 GdkEventButton类型，因为对被调用的函数来说，它已经知道将发生什么类型的事件。

函数返回的值指示事件是否由GTK的事件处理机制做进一步的传播。

返回TRUE指示事件已被处理，并且不会进一步传播。

返回FALSE将继续正常的事件处理。

2.2.5 Hello World代码解释

我们已解释了 Gtk+编程中的事件、信号以及回调函数的机制，下面我们将详细解释helloworld程序。

下面是当clicked按钮时调用的回调函数。在这个例子里，我们忽略了构件和参数 ,但是实际处理它们并不难。在下一个例子中应用数据参数将告诉我们按下了哪一个按钮。

```
void hello( GtkWidget *widget, gpointer data )
```

```
{
g_print ("Hello World\n");
}
```

下一个回调函数有一点特殊。当窗口管理程序将事件传递给应用程序时， delete_event事件发生。在这里，我们有机会选择对这些事件做些什么动作。我们可以忽略它们，也可以做出一些响应，或简单地退出应用程序。

回调函数返回的值让 GTK知道发生了什么动作。返回 TRUE，意味着不想引发 destory信号，让应用程序继续运行。返回 FALSE，要求引发destory信号，然后调用 destory的信号处理函数。

```
gint delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{ g_print ("destory occurred\n");
return(TRUE);
}
```

下面是另一个回调函数，通过调用 gtk_main_quit()使应用程序退出。这个函数告诉 GTK当控制被交回时将从gtk_main退出。

```
void destory( GtkWidget *widget, gpointer data )
{ gtk_main_quit ();
}
```

main()是应用程序的入口。与其他任何 C语言程序一样，该程序应该有一个 main()函数作为入口。

```
int main( int argc,char *argv[] ) {
```

下一步声明了两个指向 GtkWidget类型结构的指针。它们用于创建一个窗口和一个按钮：

```
GtkWidget *window;
GtkWidget *button;
```

下面又是gtk_init函数。和以前介绍的一样，这个函数初始化 GTK，并且分析在命令行中传递进来的参数。命令行中传递过来的任何参数，只要是它能识别的，都会从列表中删除，并且修改 argc和 argv的值，就像这些参数从不存在一样，然后应用程序分析剩余的参数。

```
gtk_init (&argc, &argv);
```

下面的语句创建新窗口。这是相当直接了当的，因为 GtkWidget *window结构分配内存，现在它指向了一个有效的结构类型。到这里为止，已经创建了一个新窗口，但是直到在程序结束前调用gtk_widget_show(window)，它才会显示出来。

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

下面是将对象（窗口）和信号处理函数连接起来的两个例子。在此处捕获了 delete_event和destory两个信号。当我们用窗口管理程序关闭窗口时，或当我们在窗口的某个构件中调用gtk_widget_destroy()销毁窗口时，将引发第一个信号。在 delete_event处理函数中返回FALSE时，将引发第二个信号。 GTK_OBJECT和 GTK_SIGNAL_FUNC是用于执行类型转换和检查的宏，它们还提高了代码的可读性。

```
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                    GTK_SIGNAL_FUNC (delete_event),NULL);
gtk_signal_connect (GTK_OBJECT (window), "destory",
                    GTK_SIGNAL_FUNC (destory),NULL);
```

下面这个函数用于设置放容器对象的属性。这里将设置窗口的属性，让它内部没有构件

占据的位置有一个宽度为 10像素的空白区域。在这一节我们还会看到其他一些类似的设置构件属性的函数。其中GTK_CONTAINER是用于类型转换的宏。

```
gtk_container_set_border_width (GTK_CONTAINER (window), 10);
```

下面的函数将创建一个新按钮。它在内存中为一个新的 GtkWidget结构类型分配空间，然后对它进行初始化，并且让按钮指针指向它。按钮显示时，它上面显示“ Hello World”标签。

```
button = gtk_button_new_with_label ("Hello World");
```

在这里，我们创建了一个按钮 ,并且让它做点儿有用的事。我们在按钮上添加一个信号处理函数，当它引发clicked信号时，调用hello()函数。在这里我们不想向函数传递参数，因而我们简单传递一个NULL给hello()回调函数。很明显，当我们用鼠标点击按钮时就会引发 clicked信号。

```
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (hello),NULL);
```

我们也用这个按钮退出应用程序。这说明了 destroy信号可以来自窗口管理程序，也可以来自于应用程序。

当按钮被按下后，同上面所述一样，它首先调用 hello()回调函数，然后按设置的次序调用其他函数。根据需要，可以有多个回调函数。它们会依据它们的连接次序依次执行。因为gtk_widget_destroy()函数仅仅接受 GtkWidget *widget作为参数，在这里我们用gtk_signal_connect_object() 函数代替gtk_signal_connect()。

```
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),
                           GTK_OBJECT (window));
```

下面的函数调用是一个组装调用，后面将会专门解释。它是相当容易理解的。它简单地告诉GTK，按钮应该放置在窗户中，并且可以在窗口中显示。要注意，一个 GTK容器构件仅仅能容纳一个子构件。还有一些其他构件，它们作为容器可以用多种方法容纳多个构件。

```
gtk_container_add (GTK_CONTAINER (window),btutton);
```

现在，我们已经拥有了创建一个应用程序所需要的所有方法。当设置了全部信号处理函数，并且将按钮放在窗口中恰当的地方后，我们要求 GTK在屏幕上“ 显示”所有的构件。最好让窗口构件最后显示，这样整个窗口－包括里面的所有构件－将一起弹出来，而不是窗口先弹出来，然后窗口内的按钮再显示出来。不过，在这个简单的例子里 ,很难注意到这种区别。

```
gtk_widget_show (button);
gtk_widget_show (window);
```

然后,我们调用gtk_main()，开始等候来自X服务器的事件发生。这些事件的发生，会使某个构件引发信号。

```
gtk_main ();
```

最后是main()函数的返回值。当调用gtk_quit()时，控制返回到这里。

```
return (0);
```

现在,当我们在GTK按钮上点击鼠标按键时，构件引发 clicked信号。要使用这些信息，我们的程序中应设置一个信号处理函数以捕获这个信号。在本例子里，当我们创建的按钮被

“点击”时，将调用hello()函数，并给它传递一个NULL参数，然后调用这个信号的下一个处理函数。调用gtk_widget_destroy()函数，将窗口构件作为它的参数，运行结果是销毁窗口构件。这导致窗口引发destroy信号，当这个信号被捕获时，调用destroy()回调函数，简单退出GTK。

另一种方法是用窗口管理程序关闭窗口时，会引发delete_event事件。它调用delete_event处理函数。如果返回TRUE，将保留窗口，就像什么也没有发生一样。返回FALSE将导致GTK引发destroy信号，调用destroy信号的回调函数，退出GTK。

2.2.6 运行helloworld

上面介绍了helloworld中代码的含义。编译完成后，在应用程序所在目录下输入以下命令，运行helloworld(结果如图2-2所示)：

```
./helloworld
```

尝试一下，将鼠标放在窗口的边框处，按下鼠标左键，拖动鼠标以改变窗口大小，按钮会随之而改变大小。点击“Hello World”按钮会退出应用程序。

图2-2 Gtk版本的“Hello World”

2.3 Gnome应用程序

上面我们通过两个例子介绍了使用Gtk+库创建Linux应用程序的步骤。实际上，如果要用到Gnome库，在代码中只有一些细微的区别。

首先，应该在代码的头部包含gnome.h而不是gtk.h：

```
#include <gnome.h>
```

gnome.h头文件中已经包含了gtk.h文件。

其次，应该用gnome_init()函数替换gtk_init()函数。Gnome_init()的声明如下：

```
gnome_init(const char* app_id,
           const char* app_version,
           int argc,
           char** argv)
```

gnome_init()函数会在内部调用gtk_init()。gnome_init()函数的第一个参数是应用程序的名称，第二个参数是代表应用程序版本的字符串。这些参数是Gnome库内部使用的(例如，由参数分析程序提供的一些缺省信息)。

与gtk_init()函数一样，gnome_init()分析命令行参数；但是与gtk_init()不一样的是，它不会改变argc和argv的值。如果想分析特定的选项，应该使用gnome_init_with_popt_table()函数。我们会在后面的内容中介绍这个函数。

如果初始化失败，gnome_init()函数的返回值应该是一个非0的值，但在当前的Gnome版本中它总返回0(如果有什么问题，比如说没有找到X服务器，gnome_init()会简单地异常终止)。通常的惯例是忽略它的返回值，至少在Gnome 1.0中是这样的，但是，无论如何，检查它的返回值是一个好主意，这样可以防止未来的Gnome版本返回一个错误。

最后，在Gnome应用程序中所有用户可见的字符串都应该为翻译做标记。翻译是用GNU的gettext实用程序完成的，这称为“国际化”。在程序的开头，必须调用bindtextdomain()和

textdomain() 函数。这两个函数的调用形式如下：

```
bindtextdomain (PACKAGE, PACKAGE_LOCALE_DIR);
textdomain (PACKAGE);
```

对国际化问题，我们将在后面另辟专题进行介绍。

除了上面的几点以外，使用 Gnome库和Gtk库没有什么两样，比如创建构件、连接信号和回调函数、主循环等等。

2.4 GNU C 编译器

目前最常用的GNU C编译器(gcc)是一个全功能的ANSI C兼容编译器。如果熟悉其他操作系统或硬件平台上的一种 C编译器，将能很快地掌握 gcc。gcc可以用于编译 C语言和用C++语言编的代码，不管它使用的是什么函数库和构件库。下面简要介绍如何使用 gcc 和一些 gcc 编译器最常用的选项。

2.4.1 使用 gcc

通常后跟一些选项和文件名来使用 gcc编译器。 gcc 命令的基本用法如下：

```
gcc [options] [filenames]
```

命令行选项指定的操作将在命令行上每个给出的文件上执行。下面将介绍一些最常用到的选项。

2.4.2 gcc 选项

gcc 有超过100个的编译选项可用。这些选项中的许多选项一般根本不会用到，但一些主要的选项将会频繁用到。很多的 gcc选项包括一个以上的字符。因此必须为每个选项指定各自的连字符，并且就像大多数 Linux命令一样，不能在一个单独的连字符后跟一组选项。 例如，下面的两个命令是不同的：

```
gcc -p -g test.c
gcc -pg test.c
```

第一条命令告诉gcc编译test.c时为prof命令建立剖析信息并且把调试信息加入到可执行的文件里。第二条命令只告诉gcc为gprof命令建立剖析信息。

如果编译一个程序时不使用任何选项， gcc 将会建立(假定编译成功)一个名为 a.out 的可执行文件。例如，下面的命令将在当前目录下产生一个名为 a.out的文件：

```
gcc test.c
```

可以使用-o编译选项来为即将产生的可执行文件指定一个文件名来代替 a.out。例如，将一个叫count.c的C程序编译为名叫count的可执行文件，可以输入下面的命令：

```
gcc -o count count.c
```

gcc 同样有指定编译器处理多少的编译选项。 -c选项告诉 gcc 仅把源代码编译为目标代码而跳过汇编和连接的步骤。这个选项使用得非常频繁，因为它使编译多个 C程序时的速度更快并且更易于管理。缺省时 gcc建立的目标代码文件有一个 .o 的扩展名。

-S 编译选项告诉gcc在为C代码产生了汇编语言文件后停止编译。 gcc 产生的汇编语言文件的缺省扩展名是 .s。-E选项指示编译器仅对输入文件进行预处理。当使用这个选项时，预处

理器的输出被送到标准输出而不是储存在文件里。

1. 优化选项

用gcc编译C代码时，它会尝试用最少的时间完成编译并且使编译后的代码易于调试。易于调试意味着编译后的代码与源代码有同样的执行次序，编译后的代码没有经过优化。有很多选项可用于告诉 gcc 在耗费更多编译时间和牺牲易调试性的基础上产生更小更快的可执行文件。这些选项中最典型的是-O和-O2选项。

-O选项告诉gcc对源代码进行基本优化。这些优化在大多数情况下都会使程序执行得更快。-O2选项告诉gcc产生尽可能小和尽可能快的代码。-O2选项将使编译的速度比使用-O时慢。但通常产生的代码执行速度会更快。

除了-O和-O2优化选项外，还有一些低级选项用于产生更快的代码。这些选项非常特殊，而且最好只有完全理解这些选项将会对编译后的代码产生什么样的效果时再去使用它们。

2. 调试和剖析选项

gcc 支持数种调试和剖析选项。在这些选项里最常用到的是 -g和-pg选项。

-g选项告诉gcc产生能被GNU调试器gdb使用的调试信息以便调试程序。 gcc提供了一个很多其他C编译器里没有的特性，在gcc里可以将-g和-O (产生优化代码)联用。这一点非常有用，因为这样可以在与最终产品尽可能相近的情况下调试代码。在同时使用这两个选项时必须清楚所写的某些代码已经在优化时被 gcc做了改动。

-pg选项告诉gcc在程序里加入额外的代码，执行时，将产生 gprof用的剖析信息以显示程序的耗时情况。

上面介绍的都是关于 gcc的最简单的知识。如果想详细了解 gcc，请参考GCC-HOWTO，或者在shell提示符下输入：

```
man gcc
```

这样可以浏览gcc的手册页。

当然，对规模较大的程序来说，仅仅一行编译指令是远远不够的。有多种 GNU实用工具可以帮助完成编译选项的设置，如 autoconf、automake和libtool等。GUI生成器Glade所生成的代码中有一个autogen.sh脚本，也可以用来完成这件工作。

2.5 初始化库

前面已经介绍过，在程序的 main函数一开始，应用程序必须调用 gtk_init()函数初始化Gtk+库。对Gnome应用程序，要用gnome_init()函数代替gtk_init()函数(gnome_init()会在内部调用gtk_init())。

gnome_init()函数的第一个参数是应用程序的名称，第二个参数是代表应用程序版本的字符串。这些参数是Gnome库内部使用的(例如，由参数分析程序提供的一些缺省信息)。

函数列表：初始化Gnome

```
#include <libgnomeui/gnome-init.h>
int gnome_init(const char* app_id,
          const char* app_version,
          int argc,
          char** argv)
```

与gtk_init()函数一样，gnome_init()分析命令行参数；但是与 gtk_init()不一样的是，它不

会改变argc和argv的值。如果想分析特定的选项，应该使用 gnome_init_with_popt_table()函数。

如果初始化失败，gnome_init()函数的返回值应该是一个非0的值，但目前它总返回0(如果有什么问题，比如说没有找到X服务器，gnome_init()会简单地异常终止)。通常的惯例是忽略它的返回值。但是，无论如何，检查它的返回值是一个好主意，这样可以防止未来的 Gnome 版本产生错误。

2.6 用popt分析参数

对基于Gnome的应用程序，如果允许用户从命令行带参数启动，应该在初始化的时候对传递到应用程序的参数和选项进行分析。

2.6.1 参数分析方法

Gnome使用一个强劲的选项分析库 popt来实现这一点。 popt处理所有的缺省Gnome选项。要查看Gnome的缺省选项，只需将 --help选项传递给任何Gnome应用程序即可。还可以用定制的选项添加一个“ popt表”。要做到这一点，用 gnome_init_with_popt_table()函数代替gnome_init()函数。

函数列表：初始化库，进行参数分析

```
#include <libgnomeui/gnome-init.h>
int gnome_init_with_popt_table(const char* app_id,
                        const char* app_version,
                        int argc,
                        char** argv,
                        const struct poptOption* options,
                        int flags,
                        poptContext* return_ctx)
```

popt表是一个poptOption结构数组。下面是其定义：

```
struct poptOption {
  const char* longName;
  char shortName;
  int argInfo;
  void* arg;
  int val;
  char* descrip;
  char* argDescrip;
};
```

下面解释poptOption结构中各成员的含义。

前面的两个部分 longName和shortName是选项的长名字和短名字，例如“ help”和“ h”对应于命令行选项的help和-h。如果只想有一个选项名，它们可以设为 NULL和' \0 '。

第三个成员arginfo告知表输入项是什么类型。可以是以下几种取值：

- POPT_ARG_NONE 表明选项只是一个简单的开关，它没有参数。
- POPT_ARG_STRING表明选项有一个字符串参数，比如说 --geometry="300×300+50+100"。
- POPT_ARG_INT 表明选项带一个整数的参数，例如 --columns=10.

- POPT_ARG_LONG 表明选项带一个长整型的参数。
- POPT_ARG_INCLUDE_TABLE表明poptOption结构不指定一个选项，但是要包含另一个popt表。
- POPT_ARG_CALLBACK表明poptOption结构并不指定选项，而是一个分析表中选项的回调函数。入口种类应该在表的开头。
- POPT_ARG_INTL_DOMAIN表明poptOption结构指定这个表和任何子表的翻译范围。

arg的意义依赖于arginfo成员。对一个带参数的选项来说， arg应该指向一个参数类型的变量。popt会用参数填充所指向的变量。对 POPT_ARG_NONE，如果选项能在命令行上找到，*arg设置为TRUE。对所有的情况，arg都可以设置为NULL，popt会忽略它。

对POPT_ARG_INCLUDE_TABLE，arg指向所包含的表；对 POPT_ARG_CALLBACK，它指向要调用的回调函数；对 POPT_ARG_INTL_DOMAIN，它应该是翻译范围字符串。

val成员是每个成员的标识符。它在 Gnome应用程序中一般没有什么用，但是如果使用一个回调函数，则它在回调函数中是有用的。如果不想用它，把它设置为 0。

最后两个成员用于对 --help选项自动生成输出信息。 descrip用于描述选项； argDescrip 用于描述选项的参数。例如， --display选项的帮助是这个样子：

```
  --display=DISPLAY                X display to use
```

在这里，argDescrip是“ DISPLAY” ；descrip是“ X display to use”。要用_宏为这两个字符串标记以便翻译。

对POPT_ARG_INCLUDE_TABLE，descrip意义略有不同。在这种情况下，它在帮助输出中为一“ 组” 选项加标题。例如，在下面输出中的“ 帮助选项” ：

帮助选项
-?, --help 显示帮助信息
--usage 显示简要的用法消息

如果在popt表的开头放一个POPT_ARG_CALLBACK 类型条目，会对命令行中的每个选项的信息调用一个用户定义的回调函数。下面是回调函数应该有的类型：

```
typedef void (*poptCallbackType)(poptContext con,
                                 enum poptCallbackReason reason,
                                 const struct poptOption* opt,
                                 const char* arg,
                                 void* data);
```

poptContext对象是不透明的，它包含所有的 popt状态。这样就有可能在同一个程序中多次使用popt，或者同时分析一套以上的选项。还可以用由 popt提供的函数从popyContext中提取出当前分析状态的信息。

poptCallbackReason可取以下值：

- POPT_CALLBACK_REASON_PRE
- POPT_CALLBACK_REASON_POST
- POPT_CALLBACK_REASON_OPTION

回调函数将POPT_CALLBACK_REASON_OPTION 作为“ reson” 参数，对命令行中的每个选项调用一次。

根据要求，也可以在参数分析之前或之后调用。在这些情况下，“ reson” 参数会是

POPT_CALLBACK_REASON_PRE或 POPT_CALLBACK_REASON_POST。要指定回调函数在参数分析之前或之后调用，必须将上面两个标志与 POPT_ARG_CALLBACK 结合起来使用。例如，下面的poptOption结构初始化程序指定在参数分析之前和之后都调用回调函数：

```
{ NULL,‘ \0’ , POPT_ARG_CALLBACK|POPT_CBFLAG_PRE|POPT_CBFLAG_POST,
  &parse_an_arg_callback, 0, NULL}
```

回调函数的opt参数就是对应于最近看见的命令行选项的 poptOption结构。可以访问这个结构的val成员以判定只进行查找的选项是哪一个。 arg参数是传递到命令行选项的任何参数的原文； data参数是回调函数的数据，这个数据就是在回调函数里指定的 popyOption结构的descrip成员。

在Gnome环境中gnome_init_with_popt_table()的flags参数基本上可以忽略掉，这个“ flags”用处不大。

如果给gnome_init_with_popt_table()函数的最后一个参数return_ctx传递一个非空的指针，会返回当前的环境。可以用这个环境来提取那些不是选项的部分，比如文件名。这是使用poptGetArgs()完成的。下面是一个例子：

```
char** args;
poptContext ctx;
int i;

bindtextdomain (PACKAGE, GNOMELOCALEDIR);
textdomain (PACKAGE);
gnome_init_with_popt_table(APPNAME, VERSION, argc, argv,
                           options, 0, &ctx);

args = poptGetArgs(ctx);

if (args != NULL)
  {
    i = 0;
    while (args[i] != NULL)
      {
        /* Do something with each argument */
        ++i;
      }
  }

poptFreeContext(ctx);
```

注意，必须释放前面使用过的 poptContext变量。然而，如果对 return_ctx传递一个NULL值，库函数会释放它。还要记住，如果命令行没有参数， poptGetArgs()会返回NULL。

2.6.2 GnomeHello程序的参数分析

下面的示例代码来自于一个名为 GnomeHello的程序。这个程序中集中演示了 Gnome编程的各种技巧。GnomeHello的源代码见附录。

如果带有help启动程序，GnomeHello将输出下面的信息：

```
$ ./hello --help
```

```
Usage: hello [OPTION...]
GNOME Options
  --disable-sound              Disable sound server usage
  --enable-sound               Enable sound server usage
  --espeaker=HOSTNAME:PORT     Host:port on which the sound server to use is
                               running

Help options
  -?, --help                   Show this help message
  --usage                      Display brief usage message

GTK options
  --gdk-debug=FLAGS            Gdk debugging flags to set
  --gdk-no-debug=FLAGS         Gdk debugging flags to unset
  --display=DISPLAY            X display to use
  --sync                       Make X calls synchronous
  --no-xshm                    Don't use X shared memory extension
  --name=NAME                  Program name as used by the window manager
  --class=CLASS                Program class as used by the window manager
  --gxid_host=HOST
  --gxid_port=PORT
  --xim-preedit=STYLE
  --xim-status=STYLE
  --gtk-debug=FLAGS            Gtk+ debugging flags to set
  --gtk-no-debug=FLAGS         Gtk+ debugging flags to unset
  --g-fatal-warnings           Make all warnings fatal
  --gtk-module=MODULE          Load an additional Gtk module

GNOME GUI options
  -V, --version

Help options
  -?, --help                   Show this help message
  --usage                      Display brief usage message

Session management options
  --sm-client-id=ID            Specify session management ID
  --sm-config-prefix=PREFIX    Specify prefix of saved configuration
  --sm-disable                 Disable connection to session manager

GnomeHello options
  -g, --greet                  Say hello to specific people listed on the
                               command line
  -m, --message=MESSAGE        Specify a message other than "Hello, World!"
  --geometry=GEOMETRY          Specify the geometry of the main window
$
```

对所有Gnome应用程序来说，所有这些选项差不多都是相同的，只有最后三个，标为“GnomeHello options”的是GnomeHello特有的。--greet或-g选项打开“问候模式”；GnomeHello期望在命令行上输入姓名列表，创建一个对话框向输入的每一个名字打招呼。--message选项期望输入一个字符串参数代替常见的“Hello, World!”消息；--geometry

选项期待一个标准的X几何字符串，指定主窗口的尺寸和位置。

下面是GnomeHello用作参数分析的变量和popt表：

```
static int greet_mode = FALSE;
static char* message  = NULL;
static char* geometry = NULL;
struct poptOption options[] = {
  {
    "greet",
    ' g' ,
    POPT_ARG_NONE,
    &greet_mode,
    0,
    N_("Say hello to specific people listed on the command line"),
    NULL
  },
  {
    "message",
    ' m' ,
    POPT_ARG_STRING,
    &message,
    0,
    N_("Specify a message other than \"Hello, World!\""),
    N_("MESSAGE")
  },
  {
    "geometry",
    ' \0' ,
    POPT_ARG_STRING,
    &geometry,
    0,
    N_("Specify the geometry of the main window"),
    N_("GEOMETRY")
  },
  {
    NULL,
    ' \0' ,
    0,
    NULL,
    0,
    NULL,
    NULL
  }
};
```

下面是main()的第一部分，在这里 GnomeHello检验参数是否已经正确组合并装配成一个要欢迎的人员名单列表：

```
  GtkWidget* app;
  poptContext pctx;
  char** args;
  int i;
```

```
GSList* greet = NULL;
GnomeClient* client;

bindtextdomain(PACKAGE, GNOMELOCALEDIR);
textdomain(PACKAGE);

gnome_init_with_popt_table(PACKAGE, VERSION, argc, argv,
                           options, 0, &pctx);

/* Argument parsing */

args = poptGetArgs(pctx);

if (greet_mode && args)
  {
    i = 0;
    while (args[i] != NULL)
      {
        greet = g_slist_prepend(greet, args[i]);
        ++i;
      }
    /* Put them in order */
    greet = g_slist_reverse(greet);
  }
else if (greet_mode && args == NULL)
  {
    g_error(_("You must specify someone to greet."));
  }
else if (args != NULL)
  {
    g_error(_("Command line arguments are only allowed with --greet."));
  }
else
  {
    g_assert(!greet_mode && args == NULL);
  }

poptFreeContext(pctx);
```

2.7 国际化

在Gnome应用程序中所有用户可见的字符串都应该为翻译做标记。翻译是用 GNU的gettext实用程序完成的。gettext只是一个简单的消息分类，它存储了键 /值对，键是程序的硬编码字符串，值是翻译过的字符串(如果有的话)，或者只有键（如果没有翻译，或者键的语言已经是正确的)。

作为程序员，不一定要提供软件的多国语言版本。不过，强烈建议将其中的字符串作为翻译标记，这样，gettext脚本能够提取出一个要翻译的字符串表。程序的用户可以根据他的实际情况决定是否编译一个对应于当前语言的版本。

宏列表：翻译宏

```
#include <libgnome/gnome-i18n.h>
_(string)
N_(string)
```

Gnome使用上面的两个宏来实现翻译标记。宏 _()对字符串做翻译标记的同时还进行信息类别查找。应该在任何C准许函数调用的环境中使用它。宏 N_()不做后一项操作，仅仅是为字符串做翻译标记。可以在不允许进行函数调用的场合使用它，例如在静态数组初始化程序中。如果使用宏N_()为字符串做翻译标记，最后必须对它调用宏 _()以做实际的查找。

下面是一个简单的例子：

```
#include <gnome.h>
static char* a[] =
  N_("Translate Me"),
  N_("Me Too")
};
int main(int argc, char** argv)
{
  bindtextdomain(PACKAGE, GNOMELOCALEDIR);
  textdomain(PACKAGE);
  printf(_("Translated String\n"));
  printf(_(a[0]));
  printf(_(a[1]));
  return 0;
}
```

注意到字符串“ Translate Me”和“ Me Too”已经做了标志，这样，gettext就能发现它们，并产生一个要翻译的字符串列表。翻译程序将用这个表生成实际的翻译字符串。以后，宏 _()包含了一个函数调用，用以对数组的每个成员进行翻译查找。因为当字符串文字“ Translated String”引入时允许进行函数调用，所有的工作都是在一步内完成的。

在程序的开头，必须调用 bindtextdomain()和textdomain()函数，如同上面的例子显示的那样。在上面的代码中，PACKAGE是一个字符串，它代表程序中找到的字符串包，典型情况下它是在 config.h中定义的。 还必须定义 GNOMELOCALEDIR目录，典型情况下它是在Makefile.am中定义的(标准值应该是$(prefix)/share/locale，或$(datadir)/locale)。翻译是存储在GNOMELOCALEDIR路径下的。

当用字符串为翻译做标记时，必须保证字符串是可翻译的。要避免在运行时通过连接多个字符串生成一个字符串。例如，不要像下面这样做：

```
gchar* message = g_strconcat(_("There is an error on device "),
                             device, NULL);
```

问题是：在一些语言中，应该将设备名称放在前面 (或放在中间)。如果使用g_snprintf()或者g_strdup_printf()函数而不是字符串连接函数，翻译程序就能够改变单词的次序。下面就是正确的方法：

```
gchar* message = g_strdup_printf(_("There is an error on device %s"),
                                 device);
```

现在，翻译程序能够根据需要移动 %s的位置。

应该尽可能避免复杂的语法。例如，要翻译下面的字符串就很困难：

```
printf(_("There %s %d dog%s\n"),
       n_dogs > 1 ? _("were") : _("was"),
       n_dogs,
       n_dogs > 1 ? _("s") : "");
```

最好将条件移动到printf()的外面：

```
if (n_dogs > 0)
  printf(_("There were %d dogs\n"), n_dogs);
else
  printf(_("There was 1 dog\n"));
```

然而，即使这样也不一定能够正常工作。一些语言还有比“正好一个”和“一个以上”更多的分类(也就是，在英语的单复数形式以外，它们还有“正好两个”的说法)。可以使用以下做法：

```
static const char* ndogs_phrases[] = {
  N_("There were no dogs.\n"),
  N_("There was one dog.\n"),
  N_("There were two dogs.\n"),
  N_("There were three dogs.\n")
};
```

可是这样处理也太麻烦了。如果可能应该尽量避免出现这种情况。

当分析和显示特定种类的数据（包括日期和十进制数）时，也必须考虑国际化。通常， C库函数提供了足够的实用函数以处理这些情况，使用 strftime()、strcoll()等函数处理这种情况。GlibGDate的实用函数在内部也是用 strftime()函数处理日期数据的。

应避免的常见错误：当读和写文件时，不要使用依赖于特定地区的函数。例如， printf()和scanf()函数根据地区调整它们的小数位格式，所以不能在文件中使用这种格式。如果这样，欧洲的用户将不能读出在美国常见的文件。

2.8 保存配置信息

有时候，需要保留一些应用程序的配置信息。例如，在文件菜单里保存一个“最近打开的文件”列表，应用程序是否显示工具条、状态条等。这些功能都可以使用 Gnome API函数实现。不过，有些设置值，比如上次窗口打开的位置、尺寸等，一般不这么处理，而是使用会话管理功能实现。

libgnome库有在普通文本配置文件里保存简单的键 /值对的能力。提供的实用程序能够处理数值型和布尔型的值，可以透明地将变量值转换为文本文件，或者相反，从文本文件读出。Gnome配置文件的标准位置是 ~/.gnome，库函数文件用这个位置作为缺省位置。不过，库函数也可以使用任何其他文件。每个函数还有相应的变体函数，它们能将配置文件存到~/.gnome_private下，这个目录具有用户权限设置。通常也将这个 libgnome模块称为gnome-config。不要将这个gnome-config与gnome-config脚本混淆，后者是Gnome程序用来报告编译和链接标志的。

gnome-config函数用路径作为参数。路径由三部分组成：

- 要用到的文件名，在~/.gnome或~/.gnome_private目录下。按惯例是应用程序的名字。
- 一个节，相关配置信息的逻辑子类。
- 一个键，键/值对的一半。键实际上是与一块配置数据相联系的。

路径作为一个字符串传给Gnome，并使用“ /filename/section/key”的形式。如果想用一个不在标准Gnome目录下的文件名，可以将整个路径用“ =”隔开，它将被解释为绝对路径。甚至可以将配置文件作为简单的数据文件格式(可以用作.desktop文件——设置在里面的文件会出现在Gnome的主菜单上)。然而，XML(也许要使用gnome-xml软件包)才是这种情况的更好的选择。对存储某些种类的配置信息，XML也许是一个更好的选择，libgnome配置函数库的主要优点就是简单。

gnome-config已经有很长的历史了，它最初是为 WINE——Windows仿真器项目写的，然后用在GNU Midnight Commander文件管理器上，最后移植到Gnome库上。当前的计划是在下一个Gnome版本中用更强劲的库函数取代 gnome-config，主要想支持按主机配置、 LDAP(轻量级目录存取协议)后端，以及一些其他特性。然而，即使下层的引擎 (函数库)发生剧烈变化，gnome-config API函数也总是会得到支持的。

2.8.1 读出存储的配置数据

从文件中获得数据很简单，只要调用一个函数获取给定键的值就可以了。取值的函数接受一个路径作为参数。例如，你可能询问用户是否想要看到一个对话框：

```
gboolean show_dialog;
show_dialog =
  gnome_config_get_bool("/myapp/General/dialog");
```

如果配置文件还不存在，或没有键名与你提供的键匹配，函数返回 0、FALSE或NULL。返回字符串的函数会返回一段分配的内存，应该用 g_free()函数将字符串释放。字符串矢量函数返回一个已分配空间的字符串数组 (释放该矢量的最容易的方法是调用 g_strfreev()函数)。

如果给定键不存在，可以指定一个缺省值，在路径后加一个“ =value”。例如：

```
gboolean show_dialog;
show_dialog =
gnome_config_get_bool("/myapp/General/dialog=true");
```

每个函数都有一个 with_default式样的变体，这些函数告诉你返回的值是从配置文件取得的还是从指定的缺省值取得的。例如：

```
gboolean show_dialog;
gboolean used_default;
show_dialog =
gnome_config_get_bool_with_default("/myapp/General/dialog=true",
                                   &used_default);
if (used_default)
  printf("Default value used for show_dialog\n");
```

gnome_config_push_prefix()和gnome_config_pop_prefix()函数可以用于避免每次都要指定整个路径。例如：

```
gboolean show_dialog;
gnome_config_push_prefix("/myapp/General/");
show_dialog = gnome_config_get_bool("dialog=true");
gnome_config_pop_prefix();
```

这些函数在保存值时也起作用。

在名字中带一个 private的配置函数使用具有权限限制的 .gnome_private目录，如上面所讨论的。带translated_string后缀的函数限定用当前场所的名字限定给定的键，这些函数一般用

于Gnome读.desktop文件，对应用程序没有什么用。

函数列表：从配置文件获取数据

```
#include <libgnome/gnome-config.h>
gchar* gnome_config_get_string(const gchar* path)
gchar* gnome_config_get_translated_string(const gchar* path)
gint gnome_config_get_int(const gchar* path)
gdouble gnome_config_get_float(const gchar* path)
gboolean gnome_config_get_bool(const gchar* path)
void gnome_config_get_vector(const gchar* path,
                      gint* argcp,
                      gchar*** argvp)
gchar* gnome_config_private_get_string(const gchar* path)
gchar* gnome_config_private_get_translated_string(const gchar* path)
gint gnome_config_private_get_int(const gchar* path)
gdouble gnome_config_private_get_float(const gchar* path)
gboolean gnome_config_private_get_bool(const gchar* path)
void gnome_config_private_get_vector(const gchar* path,
                             gint* argcp,
                             gchar*** argvp)
gchar* gnome_config_get_string_with_default(const gchar* path,
                                    gboolean* was_default)
gchar* gnome_config_get_translated_string_with_default(const gchar* path,
                                              gboolean* was_default)
gint gnome_config_get_int_with_default(const gchar* path,
                                gboolean* was_default)
gdouble gnome_config_get_float_with_default(const gchar* path,
                                   gboolean* was_default)

gboolean gnome_config_get_bool_with_default(const gchar* path,
                                  gboolean* was_default)
void gnome_config_get_vector_with_default(const gchar* path,
                                   gint* argcp,
                                   gchar*** argvp,
                                   gboolean* was_default)

gchar* gnome_config_private_get_string_with_default(const gchar* path,
                                           gboolean* was_default)
gchar* gnome_config_private_get_translated_string_with_default(const gchar*
path,
                                                     gboolean* was_default)
gint gnome_config_private_get_int_with_default(const gchar* path,
                                       gboolean* was_default)
gdouble gnome_config_private_get_float_with_default(const gchar* path,
                                          gboolean* was_default)

gboolean gnome_config_private_get_bool_with_default(const gchar* path,
                                        gboolean* was_default)

void gnome_config_private_get_vector_with_default(const gchar* path,
                                          gint* argcp,
```

```
                                        gchar*** argvp,
                                        gboolean* was_default)
```

2.8.2 在配置文件中存储数据

保存数据是与获取数据相反的过程，用同样的方法给出一个路径“ /file/section/key”，连带要存储的值。数据并不是立即写入的，必须调用 gnome_config_sync()函数以保证文件写到磁盘上了。

函数列表：保存数据到配置文件

```
#include <libgnome/gnome-config.h>
void gnome_config_set_string(const gchar* path,
                             const gchar* value)
void gnome_config_set_translated_string(const gchar* path,
                                        const gchar* value)

void gnome_config_set_int(const gchar* path,
                          gint value)

void gnome_config_set_float(const gchar* path,
                            gdouble value)

void gnome_config_set_bool(const gchar* path,
                           gboolean value)

void gnome_config_set_vector(const gchar* path,
                             gint argc,
                             const gchar* const argv[])

void gnome_config_private_set_string(const gchar* path,
                                     const gchar* value)

void gnome_config_private_set_translated_string(const gchar* path,
                                                const gchar* value)

void gnome_config_private_set_int(const gchar* path,
                                  gint value)

void gnome_config_private_set_float(const gchar* path,
                                    gdouble value)

void gnome_config_private_set_bool(const gchar* path,
                                   gboolean value)

void gnome_config_private_set_vector(const gchar* path,
                                     gint argc,
                                     const gchar* const argv[])
```

2.8.3 配置文件迭代器

迭代器用于在给定文件中扫描节，或在给定节中扫描键。应用程序可以用这个特性存储

数据列表。具体实现方法是通过动态生成键或节的名字以保存数据，随后迭代它们以检查保存了些什么。

迭代器是一种不透明的数据类型；可以将“节”的名称传给 gnome_config_init_iterator()函数，迭代一遍，并依次接收一个迭代器。然后调用 gnome_config_iterator_next()函数从该节中获得键/值对。从gnome_config_iterator_next()函数返回的键/值对必须用g_free()函数释放，gnome_config_iterator_next()函数返回的值是一个指向下一个迭代器的指针。当函数gnome_config_iterator_next()返回NULL时，说明已经遍历了所有的键/值对。

gnome-apt程序中的迭代示例

下面是一个来自gnome-apt的迭代示例，在Debian发布版本中用来管理软件包的C++应用程序。gnome-apt在一个树状视图中保存和加载一些栏的位置。栏是用GAptPkgTree::Column Type枚举类型标识的。GAptPkgTree::ColumnTypeEnd是栏枚举类型的最后一个元素，它等于有效栏类型的数目。

```
static void
load_column_order(vector<GAptPkgTree::ColumnType> & columns)
{
  gpointer config_iterator;
  guint loaded = 0;

  config_iterator = gnome_config_init_iterator("/gnome-apt/ColumnOrder");

  if (config_iterator != 0)
    {
      gchar * col, * pos;
      columns.reserve(GAptPkgTree::ColumnTypeEnd);

      loaded = 0;
      while ((config_iterator =
              gnome_config_iterator_next(config_iterator,
                                         &col, &pos)))
        {
          // shouldn't happen, but I'm paranoid
          if (pos == 0 || col == 0)
            {
              if (pos) g_free(pos);
              if (col) g_free(col);
              continue;
            }

          GAptPkgTree::ColumnType ct = string_to_column(col);

          gint index = atoi(pos);

          g_free(pos); pos = 0;
          g_free(col); col = 0;

          // the user could mangle the config file to make this happen
          if (static_cast<guint>(index) >= columns.size())
```

```
            continue;

          columns[index] = ct;

          ++loaded;
        }
    }

  if (loaded != static_cast<guint>(GAptPkgTree::ColumnTypeEnd))
    {
      // Either there was no saved order, or something is busted - use
      // default order
      columns.clear();

      int i = 0;
      while (i < GAptPkgTree::ColumnTypeEnd)
        {
          columns.push_back(static_cast<GAptPkgTree::ColumnType>(i));
          ++i;
        }

      // Clean the section - otherwise an old entry could
      //  remain forever and keep screwing us up in the future.
      gnome_config_clean_section("/gnome-apt/ColumnOrder");
      gnome_config_sync();
    }

  g_return_if_fail(columns.size() ==
                   static_cast<guint>(GAptPkgTree::ColumnTypeEnd));
}
```

下面是用于保存“列”位置的函数：

```
static void
save_column_order(const vector<GAptPkgTree::ColumnType> & columns)
{
  g_return_if_fail(columns.size() ==
                   static_cast<guint>(GAptPkgTree::ColumnTypeEnd));

  int position = 0;
  vector<GAptPkgTree::ColumnType>::const_iterator i = columns.begin();
  while (i != columns.end())
    {
      gchar key[256];
      g_snprintf(key,255,"/gnome-apt/ColumnOrder/%s",column_to_string(*i));
      gchar val[30];
      g_snprintf(val,29,"%d",position);
      gnome_config_set_string(key, val);

      ++position;
      ++i;
```

```
        }

      gnome_config_sync();
    }
```

在这段代码中，将枚举值保存为字符串而不是整数值。 column_to_string()和string_to_column()函数使用了一个简单的、由枚举值索引的栏名数组，用于来回转换。这么做有两个理由：在程序的未来版本中枚举值发生变化时，代码不会因此而发生故障，同时，它还使配置文件可以人工修改。

栏位置是用gnome_config_set_string()而不是用gnome_config_set_int()存储的。这是因为gnome_config_iterator_next()返回一个代表存储信息的字符串。更可能的情况是：gnome_config_set_int()函数将整数存储为atoi()函数能理解的字符串(它确实能理解)，但是从技术上来说，API函数并未保证这一点。如果代码中使用了 gnome_config_set_int()函数，它将从gnome_config_iterator_next()函数获得一个唯一的键，然后调用 gnome_config_get_int()函数获得整数值。对获得的字符串值使用 atoi()函数会对gnome-config 的实现产生没有根据的假设。

2.8.4 节迭代器

gnome_config_init_iterator_sections()允许在一个文件中迭代所有的节，而不是在一节中迭代键。迭代节时，gnome_config_iterator_next()函数忽略它的值参数并将节名放在键参数位置上。

函数列表：配置文件迭代器

```
#include <libgnome/gnome-config.h>
void* gnome_config_init_iterator(const gchar* path)
void* gnome_config_private_init_iterator(const gchar* path)
void* gnome_config_init_iterator_sections(const gchar* path)
void* gnome_config_private_init_iterator_sections(const gchar* path)
void* gnome_config_iterator_next(void* iterator_handle,
                         gchar** key,
                         gchar** value)
```

2.8.5 其他的配置文件操作

下面的函数列表列举了一些可用于操作配置文件的其他操作函数。这些函数中最重要的已经介绍过了。gnome_config_sync()将配置文件写到磁盘上，gnome_config_push_prefix()允许缩短传递到其他 gnome-config函数中的路径长度。还有一些布尔测试函数，用于询问gnome-config给定的节是否存在。

这里面有两个新函数。“ 删除”一个文件或节意味着忘掉任何存储在内存中的相关信息，包括从文件加载的缓存值和还没有用 gnome_config_sync()函数保存到磁盘上的值。要“ 清除”一个文件、节或键意味着将它的值清除，所以，一旦调用 gnome_config_sync()函数，文件、节或键都不再存在。

gnome_config_sync()函数自动调用 gnome_config_drop_all()函数并释放所有的 gnome-config资源，因为信息已经安全地在磁盘上存在了。

还有可以从gnome-config路径取得一个配置文件的实际(文件系统级)路径的函数。这些在应用程序中没有多大用处。

函数列表：其他配置文件函数

```
#include <libgnome/gnome-config.h>
gboolean gnome_config_has_section(const gchar* path)
gboolean gnome_config_private_has_section(const gchar* path)
void gnome_config_drop_all()
void gnome_config_sync()
void gnome_config_sync_file(const gchar* path)
void gnome_config_private_sync_file(const gchar* path)
void gnome_config_drop_file(const gchar* path)
void gnome_config_private_drop_file(const gchar* path)
void gnome_config_clean_file(const gchar* path)
void gnome_config_private_clean_file(const gchar* path)
void gnome_config_clean_section(const gchar* path)
void gnome_config_private_clean_section(const gchar* path)
void gnome_config_clean_key(const gchar* path)
void gnome_config_private_clean_key(const gchar* path)
gchar* gnome_config_get_real_path(const gchar* path)
gchar* gnome_config_private_get_real_path(const gchar* path)
void gnome_config_push_prefix(const gchar* path)
void gnome_config_pop_prefix()
```

2.9 会话管理

术语“会话”是指用户桌面状态的快照：哪个应用程序是打开的，窗口放在桌面的什么地方，每个应用程序打开了什么窗口，这些窗口的尺寸是多大，打开了什么文档，当前鼠标光标的位置等等。用户能在退出前保存这些会话，并且在下次登录时尽可能地自动恢复上次的会话。要做到这一点，应用程序必须有能力记录和恢复其状态特征，这些状态特征不是由窗口管理器控制的。

当应用程序应该保存状态时，由一个称为会话管理器的特殊程序通知应用程序。 Gnome桌面环境带有一个称为gnome-session的会话管理器，但是Gnome用的X会话管理规范已经过时好几年了。 CDE(通用桌面环境)使用同样的规范， KDE(K桌面环境)也准备采用这个规范。一个通过Gnome接口实现了会话管理的应用程序应该在任何会话管理的桌面环境上正常工作。Gnome确实对基本的规范(特别是启动“优先级”)做了一些扩展，但是这些应该不会中断其他的会话管理器，并且KDE也可能会实现这些规范。

使用GnomeClient对象

Gnome将应用程序与原始的、随X系统自带的会话管理接口屏蔽开来。这是通过一个称为GnomeClient的GtkObject对象做到的。 GnomeClient代表应用程序与会话管理器的连接。Gnome管理会话管理的大多数细节。对大多数应用程序来说，只需要对两个请求响应。

- 当保存一个会话时，会话管理器会要求每个客户保存足够的信息，以便在下次登录时恢复状态。应用程序应该保存尽可能多的状态：当前打开的文档、鼠标光标的位置、命令历史等等。应用程序不用保存当前窗口的几何参数，窗口管理器会做这些事。

- 有时，会话管理器会要求客户关闭并退出（典型情况是当用户退出时）。当接收到这样的信息时，应该完成一些必要的工作，然后退出应用程序。

当会话管理器要求应用程序完成某个动作时， GnomeClient对象引发一个适当的信号。两个重要的信号是save_yourself和die。当应用程序保存它的状态时会引发 save_yourself信号，当应用程序应该退出时，引发 die信号。 save_yourself信号的回调函数相当复杂，有好几个参数。die信号的回调函数很简单。

下面是来自于GnomeHello中的代码。GnomeHello获得一个指向GnomeClient对象的指针，并设置了一个信号回调函数：

```
client = gnome_master_client ();
gtk_signal_connect (GTK_OBJECT (client), "save_yourself",
                    GTK_SIGNAL_FUNC (save_session), argv[0]);
gtk_signal_connect (GTK_OBJECT (client), "die",
                    GTK_SIGNAL_FUNC (session_die), NULL);
```

argv[0]将用于save_yourself信号的回调函数。

下面是GnomeHello中的die回调函数：

```
static void
session_die(GnomeClient* client, gpointer client_data)
{
  gtk_main_quit ();
}
```

它的作用是退出应用程序。

下面是save_yourself的回调函数：

```
static gint
save_session (GnomeClient *client, gint phase, GnomeSaveStyle save_style,
              gint is_shutdown, GnomeInteractStyle interact_style,
              gint is_fast, gpointer client_data)
{
  gchar** argv;
  guint argc;
  /* allocate 0-filled, so it will be NULL-terminated */
  argv = g_malloc0(sizeof(gchar*)*4);
  argc = 1;
  argv[0] = client_data;

  if (message)
    {
      argv[1] = "--message";
      argv[2] = message;
      argc = 3;
    }
  gnome_client_set_clone_command (client, argc, argv);
  gnome_client_set_restart_command (client, argc, argv);

  return TRUE;
}
```

save_yourself信号必须告诉会话管理器怎样重新启动并克隆应用程序 (创建一个新的实例)。重新启动的应用程序应该记住尽可能多的状态，在 GnomeHello例子中，它记住了显示的消息。保存应用程序状态最简单的方法是生成一个命令行，就像 GnomeHello所做的。可以要求GnomeClient使用gnome-config API函数做这些工作，还可以将信息保存在一个按会话配置的文件中。带有重要状态信息的应用程序需要使用这种方法。

2.10 Gtk+的主循环

Gtk+主循环的首要目的就是在连接到 X服务器的文件描述符上监听事件，并将事件转发到构件上。本节解释怎样使用主循环，怎样给主循环添加新功能：当主循环在指定的时间间隔内空闲时、当一个文件描述符已经读或写就绪、以及当主循环退出时调用一个函数。

2.10.1 主循环基本知识

从根本上来说，主循环是由 glib实现的。Gtk+将glib主循环连接到Gdk的X服务器，并提供一个方便的接口(glib循环是比Gtk+的循环更低层的)。

gtk_main()函数运行主循环。直到调用 gtk_main_quit()函数，gtk_main()才会退出。gtk_main()函数可以递归调用，每次调用一个 gtk_main_quit()就退出gtk_main()函数的一个实例。gtk_main_level()函数返回递归的层次，也就是：如果没有 gtk_main()运行，返回0；如果一个gtk_main()函数在运行，返回1，等等。

gtk_main()函数的所有实例功能都是一样的，它们都监视同一个与 X服务器的连接，都对同样的事件队列起作用。gtk_main()实例用于阻塞、遮断一个函数的控制流直到满足某些条件。所有的Gtk+程序都用这个技巧使应用程序正在运行时main()函数不能退出去。gnome_dialog_run()函数使用了一个递归的主循环，此循环直到用户点击对话框的按钮时它才会返回。

有时候想处理一些事件，又不想将控制交给 gtk_main()，可以调用gtk_main_iteration()函数对主循环进行迭代。例如，这样可以处理单独的一个事件，它依赖于想将什么任务挂起。可以检查是否有任何事件需要通过调用 gtk_events_pending()函数处理。同样地，这两个函数允许临时将控制交还给 Gtk+。例如，在一个很长的计算中，想显示一个进度条，必须允许Gtk+主循环周期性地返回，让Gtk+能重画进度条。可以使用下面的代码：

```
while (gtk_events_pending())
  gtk_main_iteration();
```

下面是有关主循环的函数：

```
#include <gtk/gtkmain.h>
void gtk_main()
void gtk_main_quit()
void gtk_main_iteration()
gint gtk_events_pending()
guint gtk_main_level()
```

2.10.2 退出函数

退出函数就是当调用gtk_main_quit()函数时要调用的回调函数。换句话说，回调函数只在

gtk_main()返回之前运行。回调函数应该是一个像下面这样定义的 GtkFunction：

```
typedef gint (*GtkFunction) (gpointer data);
```

退出函数是用 gtk_quit_add()添加进去的。添加退出函数时，必须指定一个由gtk_main_level()返回的主循环的级别。第二个和第三个参数指定一个回调函数和回调数据。

回调函数的返回值说明了回调函数是否应该再次调用。只要回调函数返回 TRUE，它会被重复调用。只要它返回 FALSE，将取消与主循环的连接，并且不会再次调用。所有的退出函数都返回FALSE时，gtk_main()会返回。

gtk_quit_add()函数返回一个ID号码，可以用于用gtk_quit_remove()函数删除该退出函数。还可以通过将回调数据传递给 gtk_quit_remove_by_data()函数来删除退出函数。

函数列表：退出函数

```
#include <gtk/gtkmain.h>
guint gtk_quit_add(guint main_level,
            GtkFunction function,
            gpointer data)
void gtk_quit_remove(guint quit_handler_id)
void gtk_quit_remove_by_data(gpointer data)
```

2.10.3 Timeout函数

有时候可能想应该在 gtk_main主循环中怎样让 GTK做点什么。这时可以创建一个定时（Timeout）函数，隔一定时间(毫秒)就调用一次。 Timeout类似于其他编程环境中的定时器控件。下面的函数用于添加一个 Timeout函数。

```
#include <gtk/gtkmain.h>
gint gtk_timeout_add( guint32     interval,
                      GtkFunction function,
                      gpointer    data );
```

第一个参数调用定时函数的时间间隔，以毫秒计。第二个参数是要调用的函数，第三个是要传递给函数的参数。函数返回一个整数值“ 标志”。可以用下面的函数停止调用定时函数：

```
#include <gtk/gtkmain.h>
void gtk_timeout_remove( gint tag );
```

其中tag参数是前一个函数返回的“ 标志” 值。

还可以让回调函数返回FALSE或0来停止调用定时函数。也就是说，要想让函数继续调用，必须让它返回一个非0值或TRUE。

定期调用的回调函数声明应该是下面的形式：

```
gint timeout_callback( gpointer data );
```

可以看到，Timeout函数类似于许多可视化编程工具中的 Timer控件（计时器）。

2.10.4 idle函数

当Gtk+主循环没有其他事情做时， idle函数连续运行。只有在事件队列是空的，并且主循环正常空闲着，正等待着有什么事情发生时， idle函数才会运行。只要它们返回 TRUE，这个函数就会一次又一次地调用；当它们返回 FALSE时，函数会被删除，就像调用了

gtk_idle_remove()函数一样。

列在下面函数列表中的idle函数API，与timeout以及退出函数API是一样的。不过，不能在idle函数中调用gtk_idle_remove()函数，因为它会破坏Gtk+的函数列表。要返回FALSE来删除idle函数。

idle函数在对“只此一次”的代码排队时通常是很有用的，这样的代码一般在所有的事件已经处理之后才运行。相对昂贵（耗费系统资源较多）的操作，比如Gtk+的大小协商以及GnomeCanvas重绘一般在返回FALSE的idle函数中发生。这保证了代价昂贵的操作只会执行一次，即使多个连续的事件独立地要求重新执行。

Gtk+的主循环包含一个简单的调度程序。idle函数有一个分配给它们的优先级，就像UNIX进程所做的一样，也可以分配一个非缺省的优先级给idle函数。

函数列表：idle函数

```
#include <gtk/gtkmain.h>
guint gtk_idle_add(GtkFunction function,
            gpointer data)
void gtk_idle_remove(guint idle_handler_id)
void gtk_idle_remove_by_data(gpointer data)
```

2.10.5 输入函数

输入函数用于检查文件描述符的数据(由open(2)或socket(2)返回的文件描述符)。输入函数是在Gdk级处理的。当给定的文件描述符已经读写就绪时，会调用输入函数。它们对网络应用程序特别有用。关于文件描述符请参考Unix编程方面的参考书。

要添加一个输入函数，需要指定要监视的文件描述符、要等待的状态(读或写就绪)，以及一个回调函数/数据对。下面的函数列表列出了这些API。该函数可以用gdk_input_add()返回的标识符删除。不像quit、timeout和idle函数，从输入函数里面调用gdk_input_remove()函数删除该函数是安全的，Gtk+不会处于输入函数列表的迭代状态。

要指定等待的条件，使用GdkInputCondition标志：

GDK_INPUT_READ

GDK_INPUT_WRITE

GDK_INPUT_EXCEPTION

可以用OR将一个以上的标志连在一起。这些标志对应于传到select()系统调用的三种文件描述符集。要了解详细的内容，请参考UNIX的编程参考书。如果所有条件都得到满足，就会调用输入函数。

回调函数应该是这个样子：

```
typedef void (*GdkInputFunction) (gpointer data,
                                  gint source_fd,
                                  GdkInputCondition condition);
```

它接受回调数据、被监视的文件描述符以及要满足的条件(可能是一个正在监视的条件的子集)。

函数列表：输入函数

```
#include <gdk/gdk.h>
```

```
gint gdk_input_add(gint source_fd,
                   GdkInputCondition condition,
                   GdkInputFunction function,
                   gpointer data)
void gdk_input_remove(gint tag)
```

2.11 编译应用程序

用前面所介绍的基本概念，已经可以编译全功能的 Gtk+/Gnome应用程序了。但是还有一个大问题：如何配置编译选项？一些实用工具如 automake、autoconf、libtool等，可以用来简化这一过程。

为了方便维护，同时，也是为了便于使用这些实用工具，应该在编写代码时遵从一些约定。如果要将程序发布为自由软件，最好能使程序源代码的目录结构遵从“ GNU项目编码标准”。即使应用程序是私有的商用程序，不想公开源代码，从技术上来说，这么做也是一个非常好的选择，因为这些标准都是经过实践检验，能够让你节省大量的时间和精力。另外还应该在程序代码中包含INSTALL、README的文件。

2.11.1 生成源代码树

差不多所有的Gnome应用程序都使用同样的基于GNU工具automake、autoconf和libtool的编译系统。Gtk+和Gnome提供了一套autoconf宏，用于生成可移植的、符合标准的编译设置。我们用一个称为GnomeHello的应用程序来演示Gnome的特性。

Gnome应用程序遵从一系列的约定来生成源代码树和发布的 tar文件，大多数约定被自由软件社区广泛使用。这些约定的许多方面已经在“ GNU项目编码标准” (GNU Project’s Coding Standards：http://www.gnu.org/prep/standards_toc.html)和Linux文件系统层次标准(Linux Filesystem Hierarchy Standard：http://www.pathname.com/fhs/)中正式化了。

GNU工具集，包括automake和autoconf使遵从这些标准变得很容易。然而，有时候你可能不想使用GNU工具集，例如，你也许需要一个统一的在 Windows和MacOS平台上都能工作的编译工具(一些工具确实能在Windows平台上工作，它们使用Cygnus的“ Cygwin” 环境，参看http://sourceware.cygnus.com/cygwin)。

如果使用了autoconf和automake，除了编译应用程序，用户并不需要有这些工具。使用这些工具的目的是创建能在用户环境使用的、可移植的 shell脚本和Makefile文件。

Autoconf实际上是一个工具集，其中包含 aclocal、autoheader和autoconf等可执行文件。这些工具生成一个可移植的shell脚本——configure，configure和软件包一起发布给用户。它探查编译系统，生成Makefile文件和一个特殊的头文件 config.h。由configure生成的文件能适应用户系统的特定环境。configure脚本从一个称为Makefile.in的模板文件生成每个Makefile文件。automake由一个手写的Makefile.am生成Makefile.in文件。 Makefile.in文件随软件一同发布，当用户运行configure时会自动生成Makefile。

Libtool软件包是第三个重要的 GNU工具，它的作用是确定共享库在特定平台上的特性。因为共享库在不同平台上可能会有所不同。

下面有一些Gnome软件包应该具有的特征：

- 一个README文件，介绍软件包。

- 一个INSTALL文件，解释怎样编译、安装软件包。
- 一个configure脚本，能使程序自动适应特定平台的特征（或者该平台所缺乏的特性）。configure可以带一个参数--prefix，指定要安装的软件包的位置。
- 标准的make目标，比如clean等等。
- 一个COPYING文件，包含软件包的版权信息。
- 一个ChangeLog文件，记录了软件的变化。
- 打包文件，一般用gzip压缩，在名字中包含软件包的版本(例如foo-0.2.1.tar.gz)。它们应该解开到单个目录中，目录应该以软件包及其版本命名，比如 foo-0.2.1。
- 国际化是由GNU gettext软件包提供的。将gettext软件包随应用程序提供给用户，这样用户没有gettext也能够实现国际化。

下面是创建Gtk+/Gnome应用程序源代码树框架的重要步骤：

1) 创建一个顶级目录，用以容纳应用程序的所有组件，包括编译文件、文档以及翻译文件。

2) 通常在顶级目录下创建一个 src子目录，将所有的源代码放在该目录下，并与其他文件分开。

3) 在顶级目录下，创建 AUTHORS、NEWS、COPYING和README文件。还可以创建一个空的ChangeLog文件。

4) 写一个configure.in文件；configure.in文件的主要作用是决定使用什么样的编译器、编译标志以及链接标志。 configure.in还可以使用#define符号反映当前平台的特征；它把这些优先放在自动生成的config.h文件里。

5) 写一个acconfig.h文件。它是config.h.in文件要使用的模板文件。这个文件应该撤销每个可能在 config.h中定义了的符号以避免重复定义（一般在 config.h中用#define定义，用#undef撤销定义）。autoheader程序基于acconfig.h创建config.h.in文件， autoconf程序创建config.h文件。autoheader是autoconf软件包中的实用程序。

6) 创建一个空的stamp.h.in文件。在configure.in中的AM_CONFIG_HEADER宏会用到它。

7) 在顶级目录下，写一个 Makefile.am文件，在其中列出每个包含源代码的子目录；在每个子目录中也写一个Makefile.am文件。

8) 运行gettext软件包中的gettextize程序。这样可以创建 intl和po目录，这是软件国际化所需要的。在intl目录中包含GNU gettext源代码。如果编译程序的用户没有 gettext，它们可以在执行configure脚本时传一个 --with-included-gettext参数，让configure在intl目录下自动编译一个gettext的静态版本。在po容纳了翻译文件后， gettextize也会创建一个称为po/Makefile.in.in的文件，用于编译翻译文件。

9) 创建一个po/POTFILES.in文件，在其中列出应该扫描字符串以便翻译的源文件。最初的POTFILES.in文件可以是空的。

10) 从其他的Gnome模块中复制一个autogen.sh文件和它的宏目录。必须根据自己的软件包的名称修改 autogen.sh文件。运行 autogen.sh文件将调用 libtoolize、aclocal、autoheader、automake以及autoconf。

11) autogen.sh用--add-missing参数调用文件automake。这会添加一些文件，比如带有通用安装指导的 INSTALL文件。编辑 INSTALL，在其中包含任何针对应用程序的安装指南。

autogen.sh会在每个目录下创建一个Makefile。

2.11.2 configure.in文件

autoconf处理configure.in文件，生成一个configure脚本。configure是一个可移植的shell脚本，它检查编译环境以决定哪些库可用，所用平台有什么特征，哪些库和头文件已经找到等等。基于这些信息，它修改编译标记，生成 Makefile文件，并/或输出一个包含已定义的预处理符号的config.h文件。configure并不需要运行autoconf，所以在发布应用程序之前生成这个文件，这样，用户就不必有autoconf软件包。

眼前的任务就是写一个configure.in文件。文件基本上是一系列的m4宏，根据传递给它们的参数，这些宏扩展为 shell脚本代码段。还可以手工书写 shell代码。要真正理解怎样写configure.in文件要求有一些m4的知识以及一些Bourne shell的知识。幸运的是，有省事的方法；可以找一个已有的configure.in文件，然后修改它以适应你的应用程序。还有一个很全面的autoconf 手册，里面介绍了很多随autoconf发布的预先写好的宏。

Gtk+和Gnome的开发者已经进一步简化了这些工作，提供了一些宏用于在用户的系统中定位Gtk+和Gnome。

下面是一个简单的configure.in文件，来自于Gnome版的“ Hello, World”：

```
AC_INIT(src/hello.c)
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(GnomeHello, 0.1)
AM_MAINTAINER_MODE
AM_ACLOCAL_INCLUDE(macros)
GNOME_INIT
AC_PROG_CC
AC_ISC_POSIX
AC_HEADER_STDC
AC_ARG_PROGRAM
AM_PROG_LIBTOOL
GNOME_COMPILE_WARNINGS
ALL_LINGUAS="de es fr no ru sv fi"
AM_GNU_GETTEXT
AC_SUBST(CFLAGS)
AC_SUBST(CPPFLAGS)
AC_SUBST(LDFLAGS)
AC_OUTPUT([
Makefile
macros/Makefile
src/Makefile
intl/Makefile
po/Makefile.in
pixmaps/Makefile
doc/Makefile
doc/C/Makefile
doc/es/Makefile
])
```

上面以AC开头的宏来自 autoconf，以AM开头的宏来自 automake。要从autoconf或

automake中寻求帮助，这一点很有用。以GNOME开头的宏来自于Gnomemacros目录。这些宏都是用m4宏语言写的。如果将autoconf和automake安装在/usr目录下，autoconf和automake中的标准宏一般放在/usr/share/aclocal目录下。

- AC_INIT总是configure.in中的第一个宏。它扩展为许多可由其他 configure脚本共享的模板文件代码。这些代码解析传到 configure中的命令行参数。这个宏的一个参数是一个文件名，这个文件应该在源代码目录中，它用于健全性检查，以保证 configure脚本已正确定位源文件目录。
- AM_CONFIG_HEADER指定了要创建的头文件，差不多总是 config.h。创建的头文件包含由configure定义的C预处理符号。最低限度应该定义 PACKAGE和VERSION符号，这样可以将应用程序名称和版本传送到代码中，而无须对它们硬编码 (非公用的源文件应该包含config.h(#include <config.h>)以利用这些定义。然而，不要将 config.h文件安装到系统中，因为它有可能与其他的软件包冲突)。
- AM_INIT_AUTOMAKE初始化automake。传到这个宏里的参数是要编译的应用程序的名称和版本号(这些参数成为config.h中定义的PACKAGE和VERSION值)。
- AM_MAINTAINER_MODE关闭缺省时仅供程序维护者使用的 makefile目标，并修改以使configure能理解--enable-maintainer-mode选项。--enable-maintainer-mode将maintainer-only目标重新打开。仅供维护者使用的 makefile目标允许最终用户清除自动生成的文件，比如configure，这意味着要修复编译故障，必须安装有 autoconf 和automake软件。注意，因为 autogen.sh脚本主要是给开发人员用的， autogen.sh会自动传递一个 --enable-maintainer-mode选项给configure。
- AM_ACLOCAL_INCLUDE指定一个附加的目录，用于搜索 m4宏。在这里，它指定为macros子目录。在这个目录中应该有 Gnome宏的拷贝。
- GNOME_INIT给configure添加一个与Gnome相关的命令行参数个数，并为 Gnome程序定义一些makefile变量，这些变量中包含了必要的预处理程序和链接程序标志。这些标志是由gnome-config脚本取得的。安装 gnome-libs时会安装gnome-config脚本。
- AC_PROG_CC定位C编译器。
- AC_ISC_POSIX 添加一些在某些平台上实现 POSIX兼容需要的标志。
- AC_HEADER_STDC检查当前平台上是否有标准的 ANSI头文件，如果有，则定义 STDC_HEADERS。
- AC_ARG_PROGRAM添加一些选项到configure中，让用户能够修改安装程序的名称 (如果在用户系统上碰巧有一个与要安装的程序名称相同的程序，这是很有用的)。
- AM_PROG_LIBTOOL是由automake用来设置libtool的用途的。只在计划编译共享库或动态可加载模块时才需要设置这个值。
- GNOME_COMPILE_WARNINGS给gcc命令行添加许多警告选项，但是在其他绝大多数的编译器上什么也不做。
- ALL_LINGUAS=“ es”不是一个宏，只是一句 shell代码。它包含一个由空格分隔的语言种类缩写表，对应于 po子目录下的.po文件。 .po文件包含翻译成其他语言的文本，所以ALL_LINGUAS应该列出程序已经被翻译成的所有语言。
- AM_GNU_GETTEXT由automake使用，但是这个宏会随 gettext软件包发布。它让

automake执行一些与国际化相关的任务。

- AC_SUBST输出一个变量到由configure生成的文件中。具体内容将在后面说明。
- AC_OUTPUT列出由configure脚本创建的文件。这些文件都是由带 .in后缀的同名文件生成的。例如，src/Makefile是由src/Makefile.in生成的，config.h是由config.h.in生成的。

在执行 AC_OUTPUT宏时，configure脚本处理包含有两个 @符号标志的变量 (例如@PACKAGE@)的文件。只有用 AC_SUBST输出了变量，它才能识别这些变量 (许多在上面讨论过的预先写好的宏都用 AC_SUBST定义变量)。这些特征用于将一个 Makefile.in文件转换成一个Makefile文件。典型情况下， Makefile.in是由automake从Makefile.am生成的(不过，你可以只用autoconf，而不用automake，自己编写一个Makefile.in)。

2.11.3 Makefile.am文件

automake处理Makefile.am，生成一个符合标准的 Makefile.in文件。 automake会做很多工作：例如，它维护源文件之间的依赖关系；生成所有的标准目标，比如 install和clean；它还生成更复杂的目标：如果 Makefile.am是正确的，简单输入 make dist就会创建一个标准的 .tar.gz文件。

一般情况是在最上层目录下写一个 Makefile.am，然后在每一个子目录下分别写一个Makefile.am文件。 automake会从最上层开始递归处理各个 Makefile.am，然后生成一个Makefile.in。

在最上层目录的Makefile.am通常都很简单，下面是一个例子：

```
SUBDIRS = macros po intl src pixmaps doc
EXTRA_DIST = \
        gnome-hello.desktop
Applicationsdir = $(datadir)/gnome/apps/Applications
Applications_DATA = gnome-hello.desktop
```

上面程序的第一行通知 automake在给定的子目录中递归查找 Makefile.am文件。在src子目录的Makefile.am是这样的：

```
INCLUDES = -I$(top_srcdir) -I$(includedir) $(GNOME_INCLUDEDIR) \
         -DG_LOG_DOMAIN=\"GnomeHello\"
-DGNOMELOCALEDIR=\""$(datadir)/locale"\" \
-I../intl -I$(top_srcdir)/intl
bin_PROGRAMS = gnome-hello
gnome_hello_SOURCES =          \
        app.c            \
        hello.c          \
        menus.c         \
app.h           \
hello.h         \
menus.h
gnome_hello_LDADD = $(GNOMEUI_LIBS) $(GNOME_LIBDIR) $(INTLLIBS)
```

automake能够理解许多“ 不可思议的变量”，并用这些变量创建 Makefile.in文件。在上面的小例子中，用到了下面的变量：

- INCLUDES指定了在编译阶段 (与连接阶段相对)中传递给C编译器的标志。这一行用到的变量来自于2.11.2节中的configure.in文件。

- bin_PROGRAMS列出了要编译的程序。
- hello_SOURCES列出了要编译和连接的文件，这些文件是依赖生成 hello程序的。程序名必须列在bin_PROGRAMS中。在这个变量中的所有文件都被自动包含在发布包中。
- hello_LDADD列出了要传递给连接程序的标志。在这个例子中是由 configure决定的Gnome库标志。
- INCLUDES行中有几个在所有的 Gnome程序中都应该用到的元素。应该定义G_LOG_DOMAIN，来自与校验和断言代码中的错误信息会报告这个值，这样就能够判定错误是发生在什么地方(在代码中，还是在一个库中)。GNOMELOCALEDIR用于定位翻译文件。intl目录被添加到了头文件的搜索路径，这样应用程序就能够找到 intl头文件。

在Makefile.am中还可以做很多复杂的事，特别是，可以添加两端带有 @符号的、能带入到configure脚本中的变量。可以有条件地包含基于 configure校验的Makefile文件中的一部分，还可以建立库。automake的手册介绍了细节内容。

表2-1概括了由automake生成的最常见的 make目标。当然，缺省的 make目标是all，它编译整个程序。GNU代码标准(http://www.gnu.org/prep/standards_toc.html)中有这些make目标和GNU Makefile文件的详细信息。

表2-1 make目标

标准make目标	目 标 介 绍
Dist	建立一个用于发布的tar压缩文件(.tar.gz)
Distcheck	建立一个用于发布的压缩文件，然后尝试编译
Clean	删除编译结果(目标文件和可执行文件)，但是也许不会删除一些由机器生成的文件
Install	如果需要，创建安装目录，将软件复制到目录里
Uninstall	反安装(删除已经安装文件)
Distclean	将执行configure脚本以及所有 make过程的效果按相反的过程重做一遍，也就是，将一个tar文件还原为它原来的状态
Mostlyclean	和clean差不多，但是将那些极有可能不需重建的，目标文件留下来
Maintainer-clean	比clean更彻底，也许会删除一些需要由特殊工具软件重建的文件，比如由机器生成的源代码
TAGS	重建一个tag表，Emacs可以使用它
Check	如果有，则运行一个测试组件

2.11.4 安装支持文件

完整的Gnome应用程序还有许多代码以外的东西。它们有在线帮助(要列在 Gnome的主菜单上)，有界面翻译，还有一个桌面图标。它们也许带一个 pixmap以及一个用在“ 关于” 对话框上的徽标、一个用于“ 向导” 的图形或者一个用以帮助用户快速区别菜单项或列表元素的小图标。下面的内容介绍怎样发布这些文件。

1. 安装数据文件：文档和pixmap

文档和pixmap的安装方法是差不多的。 automake允许你将数据文件安装到任意位置，可以用配置文件中定义的变量决定将它们安装到哪里。

(1) pixmap

要从Makefile.am中安装数据文件，只需简单地为安装目标指定一个名字（pixmap 就不错）然后为该目录和要安装到目录里的文件分别创建一个变量。例如：

```
EXTRA_DIST = gnome-hello-logo.png
pixmapdir = $(datadir)/pixmaps
pixmap_DATA = gnome-hello-logo.png
fill
```

“pixmap”字符串将pixmap_DATA变量和pixmapdir变量连接起来。automake解释_DATA前缀，并在Makefile.in中生成适当的安装规则。这个Makefile.am片断将gnome-hello-logo.png文件安装到$(datadir)/pixmaps目录下，$(datadir)是由configure分配的变量。典型情况下，$(datadir)是/usr/local/share（更精确的说，是$(prefix)/share），这是独立于体系结构的数据文件（也就是，几个具有不同二进制文件格式的系统共享的文件）的标准位置。

Gnome的pixmap图片的标准位置是 $(datadir)/pixmaps，所以在例子中我们这样用。Gnome项目鼓励在所有的pixmap图片中使用PNG格式，这个格式是gdk_imlib(Gnome图象加载库)支持的。它的文件尺寸小，速度快，也不存在专利问题。

(2) 文档

安装文档使用同样的原则，不过稍有一点复杂。 Gnome文档通常是用DocBook写的。DocBook是一个SGML DTD (Document Type Definition，文档类型定义)，就像HTML一样。然而，DocBook的文档标签是为技术文档设计的。用DocBook 写的文档可以转换为其他格式，包括PostScript和HTML。依照标准，应该安装HTML格式的文档，用户就可以用 Web浏览器或Gnome帮助浏览器阅读文档。

Gnome库和帮助浏览器能理解一个名为topic.dat的文件，这个文件只是一个含有相应 URL的帮助主题列表。它起应用程序帮助主题索引的作用。下面是一个例子，只有两条：

```
gnome-hello.html        GnomeHello manual
advanced.html           Advanced Topics
```

URL路径相对于所安装的帮助文件的目录。

应该预先考虑文档可能会翻译为其他语言。最好为每一个地区建立一个子目录，例如，缺省地区(C)或es(西班牙语)。使用这种方法翻译程序不会引起混乱。一般将 Gnome帮助安装在以地区开头的目录下，这种做法用其他观点来看也是很方便的。文档目录看起来也许和GnomeHello示例程序差不多：

```
doc/
  Makefile.am
  C/
    Makefile.am
    gnome-hello.sgml
    topic.dat
  es/
    Makefile.am
    gnome-hello.sgml
    topic.dat
```

下面是doc/C/Makefile.am：

```
gnome_hello_helpdir = $(datadir)/gnome/help/gnome-hello/C
```

```
gnome_hello_help_DATA =          \
        gnome-hello.html         \
        topic.dat
SGML_FILES =                      \
        gnome-hello.sgml
# files that aren't in a binary/data/library target have to be listed here
# to be included in the tarball when you make dist
EXTRA_DIST = \
        topic.dat                \
        $(SGML_FILES)
## The - before the command means to ignore it if it fails.  That way
## people can still build the software without the docbook tools
all:
gnome-hello.html: gnome-hello/gnome-hello.html
-cp gnome-hello/gnome-hello.html .
gnome-hello/gnome-hello.html: $(SGML_FILES)
-db2html gnome-hello.sgml
## when we make dist, we include the generated HTML so people don't
## have to have the docbook tools
dist-hook:
mkdir $(distdir)/gnome-hello
-cp gnome-hello/*.html gnome-hello/*.css $(distdir)/gnome-hello
-cp gnome-hello.html $(distdir)
install-data-local: gnome-hello.html
$(mkinstalldirs) $(gnome_hello_helpdir)/images
-for file in $(srcdir)/gnome-hello/*.html $(srcdir)/gnome-hello/*.css; do \
basefile=`basename $$file`; \
$(INSTALL_DATA) $(srcdir)/$$file $(gnome_hello_helpdir)/$$basefile; \
done
gnome-hello.ps: gnome-hello.sgml
-db2ps $<
gnome-hello.rtf: gnome-hello.sgml
-db2rtf $<
```

需要特别注意的是生成的 HTML文件的安装目录：$(datadir)/gnome/help/gnome-hello/C。Gnome库在这里查找帮助文件。每个应用程序的帮助都放在 $(datadir)/gnome/help下的它自己的目录下。每个地区的文档都安装在应用程序目录的对应于地区的子目录下。

2. .desktop入口

Gnome程序带有一个.desktop入口，它是一个以desktop为后缀的文本文件，描述应用程序应该放在Gnome主菜单的什么位置。安装一个 .desktop入口文件可以让应用程序显示在 Gnome面板菜单（主菜单）中。下面是 gnome-hello.desktop文件：

```
[Desktop Entry]
Name=Gnome Hello
Name[es]=Gnome Hola
Name[fi]=GNOME-hei
Name[no]=Gnome hallo
Name[sv]=Gnome Hej
Comment=Hello World
Comment[es]=Hola Mundo
```

```
Comment[fi]=Hei, maailma
Comment[sv]=Hej Världen
Comment[no]=Hallo verden
Exec=gnome-hello
Icon=gnome-hello-logo.png
Terminal=0
Type=Application
```

这个文件由键－值对组成。 Name键指定在缺省地区场合的名称；任何键都可以有一个用于标明地区的字符串，放在一对方括号里面。当登录到 X窗口时，如果指定了不同的地区，系统会根据语言从这里读取相应的字符串。 Comment键是一个“ 工具提示”或“ 暗示”，用以更详细地描述应用程序。 Exec是用来执行程序的命令行。 Terminal是布尔类型，如果为非 0值，程序在一个终端内运行。在这里 Type应该是“ Application”

安装一个.desktop条目很简单。下面是GnomeHello中的最顶层的Makefile.am文件：

```
SUBDIRS = macros po intl src pixmaps doc
EXTRA_DIST = \
        gnome-hello.desktop
Applicationsdir = $(datadir)/gnome/apps/Applications
Applications_DATA = gnome-hello.desktop
```

注意，在$(datadir)/gnome/apps/下有一个目录树，可以将应用程序安装到下面子目录所对应的类别中。GnomeHello将自己安装在“ Applications”类中，而其他的应用程序也许会选择Games、Graphics、Internet或者其他合适的地方。尽量选择一个已经存在的类别，而不是自己建一个。

Makefile.am中的EXTRA_DIST变量列出了需要包含在发布软件包 (压缩的tar文件)中的文件。最重要的文件会自动包含进来，例如，所有作为二进制或库的源文件的文件都会自动包含。然而，automake并不认识.desktop文件，或SGML文档，所有这些文件必须列在 EXTRA_DIST中。如果将这些文件留在EXTRA_DIST 之外，用make distcheck命令编译发布程序通常会失败。

第二部分　Linux编程常用C语言函数库及构件库

第3章　glib库简介

glib库是Linux平台下最常用的C语言函数库，它具有很好的可移植性和实用性。 glib是Gtk+库和Gnome的基础。glib可以在多个平台下使用，比如Linux、Unix、Windows等。glib为许多标准的、常用的C语言结构提供了相应的替代物。如果有什么东西本书没有介绍到，请参考glib的头文件：glib.h。glib.h中的头文件很容易理解，很多函数从字面上都能猜出它的用处和用法。如果有兴趣，glib的源代码也是非常好的学习材料。

glib的各种实用程序具有一致的接口。它的编码风格是半面向对象，标识符加了一个前缀“ g”，这也是一种通行的命名约定。

使用glib库的程序都应该包含glib的头文件glib.h。如果程序已经包含了gtk.h或gnome.h，则不需要再包含glib.h。

3.1　类型定义

glib的类型定义不是使用C的标准类型，它自己有一套类型系统。它们比常用的 C语言的类型更丰富，也更安全可靠。引进这套系统是为了多种原因。例如， gint32能保证是32位的整数，一些不是标准C的类型也能保证。有一些仅仅是为了输入方便，比如 guint 比unsigned更容易输入。还有一些仅仅是为了保持一致的命名规则，比如， gchar和char是完全一样的。

以下是glib基本类型定义：

整数类型：gint8、guint8、gint16、guint16、gint32、guint32、gint64、guint64。其中gint8是8位的整数，guint8是8位的无符号整数，其他依此类推。这些整数类型能够保证大小。不是所有的平台都提供64位整型，如果一个平台有这些，glib会定义G_HAVE_GINT64。

整数类型gshort、glong、gint和short、long、int完全等价。

布尔类型gboolean：它可使代码更易读，因为普通C没有布尔类型。 Gboolean可以取两个值：TRUE和FALSE。实际上FALSE定义为0，而TRUE定义为非零值。

字符型gchar和char完全一样，只是为了保持一致的命名。

浮点类型gfloat、gdouble和float、double完全等价。

指针gpointer对应于标准C的void *，但是比void *更方便。

指针gconstpointer对应于标准C的const void *（注意，将const void *定义为const gpointer是行不通的）。

3.2　glib的宏

3.2.1　常用宏

glib定义了一些在C程序中常见的宏，详见下面的列表。 TRUE/FALSE/NULL就是

1/0/((void*)0)。MIN()/MAX()返回更小或更大的参数。 ABS()返回绝对值。 CLAMP(x, low, high)若X在[low, high]范围内，则等于X；如果X小于low，则返回low；如果X大于high，则返回high。

一些常用的宏列表

```
#include <glib.h>
TRUE
FALSE
NULL
MAX(a, b)
MIN(a, b)
ABS(x)
CLAMP(x, low, high)
```

有些宏只有glib拥有，例如在后面要介绍的gpointer-to-gint 和 gpointer-to-guint。

大多数glib的数据结构都设计成存储一个 gpointer。如果想存储指针来动态分配对象，可以这样做。然而，有时还是想存储一列整数而不想动态地分配它们。虽然C标准不能严格保证，但是在多数glib支持的平台上，在gpointer变量中存储gint或guint仍是可能的。在某些情况下，需要使用中间类型转换。

下面是示例：

```
gint my_int;
gpointer my_pointer;
my_int = 5;
my_pointer = GINT_TO_POINTER(my_int);
printf("We are storing %d\n", GPOINTER_TO_INT(my_pointer));
```

这些宏允许在一个指针中存储一个整数，但在一个整数中存储一个指针是不行的。如果要实现的话，必须在一个长整型中存储指针。

宏列表：在指针中存储整数的宏

```
#include <glib.h>
GINT_TO_POINTER(p)
GPOINTER_TO_INT(p)
GUINT_TO_POINTER(p)
GPOINTER_TO_UINT(p)
```

3.2.2 调试宏

glib提供了一整套宏，在你的代码中使用它们可以强制执行不变式和前置条件。这些宏很稳定，也容易使用，因而Gtk+大量使用它们。定义了G_DISABLE_CHECKS 或G_DISABLE_ASSERT之后，编译时它们就会消失，所以在软件代码中使用它们不会有性能损失。大量使用它们能够更快速地发现程序的错误。发现错误后，为确保错误不会在以后的版本中出现，可以添加断言和检查。特别是当编写的代码被其他程序员当作黑盒子使用时，这种检查很有用。用户会立刻知道在调用你的代码时发生了什么错误，而不是猜测你的代码中有什么缺陷。

当然，应该确保代码不是依赖于一些只用于调试的语句才能正常工作。如果一些语句在生成代码时要取消，这些语句不应该有任何副作用。

宏列表：前提条件检查

```
#include <glib.h>
g_return_if_fail(condition)
g_return_val_if_fail(condition, retval)
```

这个宏列表列出了glib的预条件检查宏。对g_return_if_fail()，如果条件为假，则打印一个警告信息并且从当前函数立刻返回。 g_return_val_if_fail()与前一个宏类似，但是允许返回一个值。毫无疑问，这些宏很有用——如果大量使用它们，特别是结合Gtk+的实时类型检查，会节省大量的查找指针和类型错误的时间。

使用这些函数很简单，下面的例子是glib中哈希表的实现：

```
void
g_hash_table_foreach (GHashTable *hash_table,
                      GHFunc      func,
                      gpointer    user_data)
{
  GHashNode *node;
  gint i;

  g_return_if_fail (hash_table != NULL);
  g_return_if_fail (func != NULL);

  for (i = 0; i < hash_table->size; i++)
    for (node = hash_table->nodes[i]; node; node = node->next)
      (* func) (node->key, node->value, user_data);
}
```

如果不检查，这个程序把NULL作为参数时将导致一个奇怪的错误。库函数的使用者可能要通过调试器找出错误出现在哪里，甚至要到glib的源代码中查找代码的错误是什么。使用这种前提条件检查，他们将得到一个很不错的错误信息，告之不允许使用NULL参数。

宏列表：断言

```
#include <glib.h>
g_assert(condition)
g_assert_not_reached()
```

glib也有更传统的断言函数。g_assert()基本上与assert()一样，但是对G_DISABLE_ASSERT响应（如果定义了G_DISABLE_ASSERT，则这些语句在编译时不编译进去），以及在所有平台上行为都是一致的。还有一个g_assert_not_reached()，如果执行到这个语句，它会调用abort()退出程序并且(如果环境支持)转储一个可用于调试的core文件。

应该断言用来检查函数或库内部的一致性。 g_return_if_fail()确保传递到程序模块的公用接口的值是合法的。也就是说，如果断言失败，将返回一条信息，通常应该在包含断言的模块中查找错误；如果g_return_if_fail()检查失败，通常要在调用这个模块的代码中查找错误。这也是断言与前提条件检查的区别。

下面glib日历计算模块的代码说明了这种差别：

```
GDate*
g_date_new_dmy (GDateDay day, GDateMonth m, GDateYear y)
{
  GDate *d;
  g_return_val_if_fail (g_date_valid_dmy (day, m, y), NULL);
  d = g_new (GDate, 1);
```

```
    d->julian = FALSE;
    d->dmy    = TRUE;

    d->month = m;
    d->day   = day;
    d->year  = y;

    g_assert (g_date_valid (d));

    return d;
}
```

开始的预条件检查确保用户传递合理的年月日值；结尾的断言确保 glib构造一个健全的对象，输出健全的值。

断言函数g_assert_not_reached() 用来标识“ 不可能” 的情况，通常用来检测不能处理的所有可能枚举值的switch语句：

```
switch (val)
  {
    case FOO_ONE:
      break;
    case FOO_TWO:
      break;
    default:
      /* 无效枚举值 */
      g_assert_not_reached();
      break;
  }
```

所有调试宏使用glib的g_log()输出警告信息，g_log()的警告信息包含发生错误的应用程序或库函数名字，并且还可以使用一个替代的警告打印例程。例如，可以将所有警告信息发送到对话框或log文件而不是输出到控制台。

3.3 内存管理

glib用自己的g_变体包装了标准的malloc()和free()，即g_malloc() 和 g_free()。它们有以下几个小优点：

- g_malloc()总是返回gpointer，而不是char*，所以不必转换返回值。
- 如果低层的malloc()失败，g_malloc()将退出程序，所以不必检查返回值是否是 NULL。
- g_malloc() 对于分配0字节返回NULL。
- g_free()忽略任何传递给它的NULL指针。

除了这些次要的便利， g_malloc() 和 g_free()支持各种内存调试和剖析。如果将 enable-mem-check选项传递给glib的configure脚本，在释放同一个指针两次时， g_free()将发出警告。enable-mem-profile选项使代码使用统计来维护内存。调用 g_mem_profile()时，信息会输出到控制台上。最后，还可以定义 USE_DMALLOC，GLIB内存封装函数会使用 malloc()。调试宏在某些平台上在dmalloc.h中定义。

函数列表： glib内存分配

```
#include <glib.h>
gpointer g_malloc(gulong size)
void g_free(gpointer mem)
gpointer g_realloc(gpointer mem,
         gulong size)
gpointer g_memdup(gconstpointer mem,
         guint bytesize)
```

用g_free()和g_malloc(), malloc()和free()，以及(如果正在使用C++)new 和 delete匹配是很重要的，否则，由于这些内存分配函数使用不同内存池 (new/delete调用构造函数和解构函数)，不匹配将会发生很糟糕的事。

另外，g_realloc()和realloc()是等价的。还有一个很方便的函数 g_malloc0()，它将分配的内存每一位都设置为0；另一个函数g_memdup()返回一个从mem开始的字节数为bytesize的拷贝。为了与 g_malloc()一致， g_realloc()和g_malloc0()都可以分配 0字节内存。不过，g_memdup()不能这样做。 g_malloc0()在分配的原始内存中填充未设置的位，而不是设置为数值0。偶尔会有人期望得到初始化为0.0的浮点数组，但这样是做不到的。

最后，还有一些指定类型内存分配的宏，见下面的宏列表。这些宏中的每一个 type参数都是数据类型名，count参数是指分配字节数。这些宏能节省大量的输入和操纵数据类型的时间，还可以减少错误。它们会自动转换为目标指针类型，所以试图将分配的内存赋给错误的指针类型，应该触发一个编译器警告。

宏列表：内存分配宏

```
#include <glib.h>
g_new(type, count)
g_new0(type, count)
g_renew(type, mem, count)
```

3.4 字符串处理

glib提供了很丰富的字符串处理函数，其中有一些是 glib独有的，一些用于解决移植问题。它们都能与glib内存分配例程很好地互操作。

如果需要比gchar *更好的字符串，glib提供了一个GString类型。

函数列表：字符串操作

```
#include <glib.h>
gint g_snprintf(gchar* buf,
                gulong n,
                const gchar* format,
                ...)
gint g_strcasecmp(const gchar* s1,
                const gchar* s2)
gint g_strncasecmp(const gchar* s1,
                const gchar* s2,
                guint n)
```

上面的函数列表显示了一些ANSI C函数的glib替代品，这些函数在ANSI C中是扩展函数，一般都已经实现，但不可移植。对普通的 C函数库，其中的sprintf()函数有安全漏洞，容易造成程序崩溃，而相对安全并得到充分实现的 snprintf()函数一般都是软件供应商的扩展版本。

在含有snprintf()的平台上，g_snprintf()封装了一个本地的snprintf()，并且比原有实现更稳定、安全。以往的snprintf()不保证它所填充的缓冲是以 NULL结束的，但g_snprintf()保证了这一点。

g_snprintf函数在buf参数中生成一个最大长度为 n的字符串。其中 format是格式字符串，后面的“ ...”是要插入的参数。

g_strcasecmp()和g_strncasecmp()实现两个字符串大小写不敏感的比较，后者可指定需比较的最大长度。 strcasecmp()在多个平台上都是可用的，但是有的平台并没有，所以建议使用glib的相应函数。

下面的函数列表中的函数在合适的位置上修改字符串：第一个将字符串转换为小写，第二个将字符串全部转换为大写。 g_strreverse()将字符串颠倒过来。 g_strchug()和g_strchomp()，前者去掉字符串前的空格，后者去掉结尾的空格。宏 g_strstrip()结合这两个函数，删除字符串前后的空格。

函数列表：修改字符串

```
#include <glib.h>
void g_strdown(gchar* string)
void g_strup(gchar* string)
void g_strreverse(gchar* string)
gchar* g_strchug(gchar* string)
gchar* g_strchomp(gchar* string)
```

下面的函数列表显示了几个半标准函数的 glib封装。g_strtod类似于strtod()，它把字符串nptr转换为gdouble。*endptr设置为第一个未转换字符，例如，数字后的任何文本。如果转换失败，*endptr设置为nptr值。*endptr可以是NULL，这样函数会忽略这个参数。 g_strerror()和g_strsignal()与前面没有“ g_”的函数是等价的，但是它们是可移植的，它们返回错误号或警号数的字符串描述。

函数列表：字符串转换

```
#include <glib.h>
gdouble  g_strtod(const gchar* nptr,
         gchar** endptr)
gchar*  g_strerror(gint errnum)
gchar*  g_strsignal(gint signum)
```

下面的函数列表显示了glib中的字符串分配函数。

g_strdup()和g_strndup()返回一个已分配内存的字符串或字符串前 n个字符的拷贝。为与glib内存分配函数一致，如果向函数中传递一个 NULL指针，它们返回NULL。

printf()返回带格式的字符串。 g_strescape在它的参数前面通过插入另一个“ \”，将后面的字符转义，返回被转义的字符串。 g_strnfill()根据length参数返回填充fill_char字符的字符串.

g_strdup_printf()值得特别注意，它是处理下面代码更简单的方法：

```
gchar* str = g_malloc(256);
g_snprintf(str, 256, "%d printf-style %s", 1, "format");
```

用下面的代码，不需计算缓冲区的大小：

```
gchar* str = g_strdup_printf("%d printf-style %", 1, "format");
```

函数列表：分配字符串

```
#include <glib.h>
gchar*
g_strdup(const gchar* str)
gchar* g_strndup(const gchar* format,
         guint n)
gchar* g_strdup_printf(const gchar* format,
                ...)
gchar* g_strdup_vprintf(const gchar* format,
                va_list args)
gchar* g_strescape(gchar* string)
gchar* g_strnfill(guint length,
          gchar fill_char)
```

g_strconcat() 返回由连接每个参数字符串生成的新字符串，最后一个参数必须是 NULL，让g_strconcat()知道何时结束。g_strjoin()与它类似，但是在每个字符串之间插入由 separator指定的分隔符。如果separator是NULL，则不会插入分隔符。

下面是glib提供的连接字符串的函数。

函数列表：连接字符串的函数

```
#include <glib.h>
gchar* g_strconcat(const gchar* string1,
           ...)
gchar* g_strjoin(const gchar* separator,
         ...)
```

最后，下面的函数列表总结了几个处理以 NULL结束的字符串数组的例程。 g_strsplit()在每个分隔符处分割字符串，返回一个新分配的字符串数组。 g_strjoinv()用可选的分隔符连接字符串数组，返回一个已分配好的字符串。 g_strfreev()释放数组中每个字符串，然后释放数组本身。

函数列表：处理以NULL结尾的字符串向量

```
#include <glib.h>
gchar** g_strsplit(const gchar* string,
          const gchar* delimiter,
          gint max_tokens)
gchar* g_strjoinv(const gchar* separator,
          gchar** str_array)
void g_strfreev(gchar** str_array)
```

3.5 数据结构

glib实现了许多通用数据结构，如单向链表、双向链表、树和哈希表等。下面的内容介绍glib链表、排序二叉树、N-ARY 树以及哈希表的实现。

3.5.1 链表

glib提供了普通的单向链表和双向链表，分别是 GSList 和 GList。这些是由gpointer链表实现的，可以使用GINT_TO_POINTER 和GPOINTER_TO_INT 宏在链表中保存整数。GSList 和 GList有一样的API接口，除了有g_list_previous() 函数外没有g_slist_previous()函数。本节讨论GSList的所有函数，这些也适用于双向链表。

在 glib实现中，空链表只是一个 NULL指针。因为它是一个长度为 0的链表，所以向链表函数传递NULL总是安全的。以下是创建链表、添加一个元素的代码：

```
GSList* list = NULL;
gchar* element = g_strdup("a string");
list = g_slist_append(list, element);
```

glib的链表明显受Lisp的影响，因此，空链表是一个特殊的“ 空” 值。 g_slist_prepend()操作很像一个恒定时间的操作：把新元素添加到链表前面的操作所花的时间都是一样的。

注意，必须将链表用链表修改函数返回的值替换，以防链表头发生变化。 Glib会处理链表的内存问题，根据需要释放和分配链表链接。

例如，以下的代码删除上面添加的元素并清空链表：

```
list = g_slist_remove(list, element);
```

链表 list现在是 NULL。当然，仍需自己释放元素。为了清除整个链表，可使用 g_slist_free()，它会快速删除所有的链接。因为 g_slist_free()函数总是将链表置为 NULL，它不会返回值；并且，如果愿意，可以直接为链表赋值。显然，g_slist_free()只释放链表的单元，它并不知道怎样操作链表内容。

为了访问链表的元素，可以直接访问 GSList结构：

```
gchar* my_data = list->data;
```

为了遍历整个链表，可以如下操作：

```
GSList* tmp = list;
while (tmp != NULL)
  {
    printf("List data: %p\n", tmp->data);
    tmp = g_slist_next(tmp);
  }
```

下面的列表显示了用于操作 GSList元素的基本函数。对所有这些函数，必须将函数返回值赋给链表指针，以防链表头发生变化。注意， glib不存储指向链表尾的指针，所以前插（prepend）操作是一个恒定时间的操作，而追加（ append)、插入和删除所需时间与链表大小成正比。

这意味着用g_slist_append()构造一个链表是一个很糟糕的主意。当需要一个特殊顺序的列表项时，可以先调用g_slist_prepend()前插数据，然后调用g_slist_reverse()将链表颠倒过来。如果预计会频繁向链表中追加列表项，也要为最后的元素保留一个指针。下面的代码可以用来有效地向链表中添加数据：

```
void
efficient_append(GSList** list, GSList** list_end, gpointer data)
{
  g_return_if_fail(list != NULL);
  g_return_if_fail(list_end != NULL);
  if (*list == NULL)
    {
      g_assert(*list_end == NULL);
      *list = g_slist_append(*list, data);
      *list_end = *list;
    }
```

```
    else
      {
        *list_end = g_slist_append(*list_end, data)->next;
      }
  }
```

要使用这个函数，应该在其他地方存储指向链表和链表尾的指针，并将地址传递给efficient_append()：

```
  GSList* list = NULL;
  GSList* list_end = NULL;
  efficient_append(&list, &list_end, g_strdup("Foo"));
  efficient_append(&list, &list_end, g_strdup("Bar"));
  efficient_append(&list, &list_end, g_strdup("Baz"));
```

当然，应该尽量不使用任何改变链表尾但不更新 list_end的链表函数。

函数列表：改变链表内容

```
#include <glib.h>
/* 向链表最后追加数据，应将修改过的链表赋给链表指针 */
GSList* g_slist_append(GSList* list,
                       gpointer data)
/* 向链表最前面添加数据，应将修改过的链表赋给链表指针 */
GSList* g_slist_prepend(GSList* list,
                        gpointer data)
/* 在链表的position位置向链表插入数据，应将修改过的链表赋给链表指针 */
GSList* g_slist_insert(GSList* list,
                       gpointer data,
                       gint position)
/*删除链表中的data元素，应将修改过的链表赋给链表指针 */
GSList* g_slist_remove(GSList* list,
                       gpointer data)
```

访问链表元素可以使用下面的函数列表中的函数。这些函数都不改变链表的结构。g_slist_foreach()对链表的每一项调用Gfunc函数。Gfunc函数是像下面这样定义的：

```
typedef void (*GFunc)(gpointer data, gpointer user_data);
```

在g_slist_foreach()中，Gfunc函数会对链表的每个list->data调用一次，将user_data传递到g_slist_foreach()函数中。

例如，有一个字符串链表，并且想创建一个类似的链表，让每个字符串做一些变换。下面是相应的代码，使用了前面例子中的 efficient_append()函数。

```
typedef struct _AppendContext AppendContext;
struct _AppendContext {
  GSList* list;
  GSList* list_end;
  const gchar* append;
};
static void
append_foreach(gpointer data, gpointer user_data)
{
  AppendContext* ac = (AppendContext*) user_data;
  gchar* oldstring = (gchar*) data;
```

```
  efficient_append(&ac->list, &ac->list_end,
                   g_strconcat(oldstring, ac->append, NULL));
}
GSList*
copy_with_append(GSList* list_of_strings, const gchar* append)
{
  AppendContext ac;
  ac.list = NULL;
  ac.list_end = NULL;
  ac.append = append;
  g_slist_foreach(list_of_strings, append_foreach, &ac);
  return ac.list;
}
```

函数列表：访问链表中的数据

```
#include <glib.h>
GSList* g_slist_find(GSList* list,
            gpointer data)
GSList* g_slist_nth(GSList* list,
           guint n)
gpointer g_slist_nth_data(GSList* list,
                guint n)
GSList* g_slist_last(GSList* list)
gint g_slist_index(GSList* list,
             gpointer data)
void g_slist_foreach(GSList* list,
               GFunc func,
               gpointer user_data)
```

还有一些很方便的操纵链表的函数，列在下面的函数列表中。除了 g_slist_copy()函数，所有这些函数都影响相应的链表。也就是，必须将返回值赋给链表或某个变量，就像向链表中添加和删除元素时所做的那样。而 g_slist_copy()返回一个新分配的链表，所以能够继续使用两个链表，最后必须将两个链表都释放。

函数列表：操纵链表

```
#include <glib.h>
/* 返回链表的长度 */
guint g_slist_length(GSList* list)
/* 将list1和list2两个链表连接成一个新链表*/
GSList* g_slist_concat(GSList* list1,
              GSList* list2)
/*将链表的元素颠倒次序*/
GSList* g_slist_reverse(GSList* list)
/*返回链表list的一个拷贝*/
GSList* g_slist_copy(GSList* list)
```

最后，还有一些用于对链表排序的函数，见下面的函数列表。要使用这些函数，必须写一个比较函数GcompareFunc，就像标准C里面的qsort()函数一样。在glib里面，比较函数是这个样子：

```
typedef gint (*GCompareFunc) (gconstpointer a, gconstpointer b);
```

如果a < b，函数应该返回一个负值；如果 a > b，返回一个正值；如果 a = b，返回0。

一旦有了比较函数，就可以将一个元素插入到一个已经排序的链表中，或者对整个链表排序。链表是按升序排序的。使用 g_slist_find_custom()函数，甚至能够循环使用GcompareFunc来发现链表元素。注意，在 glib中，GcompareFunc的使用是不一致的。有时glib需要一个等式判定式，而不是一个 qsort()风格的函数。不过，在链表 API中，它的用法是一致的。

不要随意对链表排序，滥用它们很快就会变得效率低下。例如， g_slist_insert_sorted()函数将数据插入到链表，同时进行排序，它是一个 O(n)复杂度的操作。但是如果在一个循环中插入多个元素，则每次插入都会进行一次排序，循环的运行时间就是指数级的。较好的方法是先将元素前插，然后调用g_slist_sort()函数对链表排序。

函数列表：对链表排序

```
#include <glib.h>
GSList* g_slist_insert_sorted(GSList* list,
                    gpointer data,
                    GCompareFunc func)
GSList* g_slist_sort(GSList* list,
            GCompareFunc func)
GSList* g_slist_find_custom(GSList* list,
                  gpointer data,
                  GCompareFunc func)
```

3.5.2 树

树是一种非常重要的数据结构。在 glib中有两种不同的树： GTree是基本的平衡二叉树，它将存储按键值排序成对键值； GNode存储任意的树结构数据，比如分析树或分类树。

1. GTree

使用下面的函数创建和销毁一个 Gtree。其中GCompareFunc是类似GSList的qsort()风格的比较函数。在这里，用它来比较树的键值。

函数列表：创建和销毁平衡二叉树

```
#include <glib.h>
GTree* g_tree_new(GCompareFunc key_compare_func)
void g_tree_destroy(GTree* tree)
```

操作树中数据的函数列在下面的函数列表中。这些函数可以从字面上理解它们的用处和用法。g_tree_insert()覆盖任何已有值，所以如果存在的值是指向已分配内存区域的唯一指针，使用时要当心。如果 g_tree_lookup()没有找到所需的键，返回NULL，否则返回相应的值。键和值都是gpointer类型，但是要使用整数，并用GPOINTER_TO_INT()和GPOINTER_TO_UINT()宏进行转换。

函数列表：操纵Gtree数据

```
#include <glib.h>
void g_tree_insert(GTree* tree,
            gpointer key,
            gpointer value)
void g_tree_remove(GTree* tree,
            gpointer key)
gpointer g_tree_lookup(GTree* tree,
```

```
                gpointer key)
```

下面的函数可以确定树的大小。

函数列表：获得GTree的大小

```
#include <glib.h>
/*获得树的节点数*/
gint g_tree_nnodes(GTree* tree)
/*获得树的高度*/
gint g_tree_height(GTree* tree)
```

使用g_tree_traverse()函数可以遍历整棵树。要使用它，需要一个 GtraverseFunc遍历函数，它用来给g_tree_traverse()函数传递每一对键值对和数据参数。只要GTraverseFunc返回FALSE，遍历继续；返回 TRUE时，遍历停止。可以用 GTraverseFunc函数按值搜索整棵树。以下是GTraverseFunc的定义：

```
typedef gint (*GTraverseFunc)(gpointer key, gpointer value, gpointer data);
```

GTraverseType是枚举型，它有四种可能的值。下面是它们在 Gtree中各自的意思：

- G_IN_ORDER(中序遍历)首先递归左子树节点 (通过GCompareFunc比较后,较小的键)，然后对当前节点的键值对调用遍历函数，最后递归右子树。这种遍历方法是根据使用GCompareFunc函数从最小到最大遍历。
- G_PRE_ORDER(先序遍历)对当前节点的键值对调用遍历函数，然后递归左子树，最后递归右子树。
- G_POST_ORDER(后序遍历)先递归左子树，然后递归右子树，最后对当前节点的键值对调用遍历函数。
- G_LEVEL_ORDER(水平遍历)在GTree中不允许使用，只能用在Gnode中。

函数列表：遍历GTree

```
#include <glib.h>
void g_tree_traverse(GTree* tree,
                GTraverseFunc traverse_func,
                GTraverseType traverse_type,
                gpointer data)
```

2. GNode

一个GNode是一棵N维的树，由双链表(父和子链表)实现。这样，大多数链表操作函数在Gnode API中都有对等的函数。可以用多种方式遍历。以下是一个 GNode的声明：

```
typedef struct _GNode GNode;
struct _GNode
{
  gpointer data;
  GNode   *next;
  GNode   *prev;
  GNode   *parent;
  GNode   *children;
};
```

有一些用来访问GNode成员的宏，见下面的宏列表。作为一个 Glist，其中的data成员可以直接使用。这些宏分别返回 next、prev和children成员，在将GList解除参照以前，这些宏也检查参数是否为NULL，如果是，则返回NULL。

宏列表：访问GNode成员

```
#include <glib.h>
/*返回GNode的前一个节点*/
g_node_prev_sibling(node)
/*返回GNode的下一个节点*/
g_node_next_sibling(node)
/*返回GNode的第一个子节点*/
g_node_first_child(node)
```

用g_node_new()函数创建一个新节点。g_node_new()创建一个包含数据，并且无子节点、无父节点的Gnode节点。通常仅用g_node_new()创建根节点，还有一些宏可以根据需要自动创建新节点。

函数列表：创建一个GNode

```
#include <glib.h>
GNode* g_node_new(gpointer data)
```

要创建一棵树，可以用下面函数列表中的函数。为方便循环或递归树，每个操作都返回刚刚添加的节点。

函数列表：创建一棵GNode树

```
#include <glib.h>
/*在父节点parent的position处插入节点node*/
GNode* g_node_insert(GNode* parent,
             gint position,
             GNode* node)
/*在父节点parent中的sibling节点之前插入节点node*/
GNode* g_node_insert_before(GNode* parent,
                    GNode* sibling,
                    GNode* node)
/*在父节点parent最前面插入节点node*/
GNode* g_node_prepend(GNode* parent,
              GNode* node)
```

下面的宏列表列出了一些常用的宏，用于实现对 Gnode的操作。g_node_append()和g_node_prepend()类似，其余的宏则带一个 data参数，自动分配节点，并且调用相关的基本操作函数。

宏列表：向Gnode添加、插入数据

```
#include <glib.h>
g_node_append(parent, node)
g_node_insert_data(parent, position, data)
g_node_insert_data_before(parent, sibling, data)
g_node_prepend_data(parent, data)
g_node_append_data(parent, data)
```

有两个函数可以从一棵树中删除一个节点。g_node_destroy()从树中删除一个节点，销毁它以及它的子节点。g_node_unlink()将一个节点删除，并将它转换为一个根节点，也就是，它将一棵子树转换为一棵独立的树。

函数列表：销毁GNode

```
#include <glib.h>
void g_node_destroy(GNode* root)
```

```
void g_node_unlink(GNode* node)
```

下面宏列表中的两个宏用来检查一个节点是否是最顶部的节点或最底部的节点。根节点没有父节点和兄弟节点，叶节点没有子节点。

宏列表：判断Gnode的类型

```
#include <glib.h>
G_NODE_IS_ROOT(node)
G_NODE_IS_LEAF(node)
```

下面函数列表中的函数返回Gnode的一些有用信息，包括它的节点数、根节点、深度以及含有特定数据指针的节点。其中的遍历类型 GtraverseType在Gtree中介绍过。下面是在 Gnode中它的可能取值：

- G_IN_ORDER 先递归节点最左边的子树，并访问节点本身，然后递归节点子树的其他部分。这不是很有用，因为多数情况用于 Gtree中。
- G_PRE_ORDER 访问当前节点，然后递归每一个子树。
- G_POST_ORDER 按序递归每个子树，然后访问当前节点。
- G_LEVEL_ORDER 首先访问节点本身，然后每个子树，然后子树的子树，然后子树的子树的子树，以次类推。也就是说，它先访问深度为 0的节点，然后是深度为 1，然后是深度为2，等等。

GNode的树遍历函数有一个 GTraverseFlags参数。这是一个位域，用来改变遍历的种类。当前仅有三个标志——只访问叶节点，非叶节点，或者所有节点：

- G_TRAVERSE_LEAFS 指仅遍历叶节点。
- G_TRAVERSE_NON_LEAFS 指仅遍历非叶节点。
- G_TRAVERSE_ALL 只是指(G_TRAVERSE_LEAFS | G_TRAVERSE_NON_LEAFS)快捷方式。

函数列表：取得GNode属性

```
#include <glib.h>
guint g_node_n_nodes(GNode* root,
              GTraverseFlags flags)
GNode* g_node_get_root(GNode* node)
Gboolean g_node_is_ancestor(GNode* node,
                 GNode* descendant)
Guint g_node_depth(GNode* node)
GNode* g_node_find(GNode* root,
           GTraverseType order,
           GTraverseFlags flags,
           gpointer data)
```

其他GNode函数都很简单，它们大多数是对树的节点表进行操作，见下面的函数列表。GNode有两个独有的函数类型定义：

```
typedef gboolean (*GNodeTraverseFunc) (GNode* node, gpointer data);
typedef void (*GNodeForeachFunc) (GNode* node, gpointer data);
```

这些函数调用以要访问的节点指针以及用户数据作为参数。 GNodeTraverseFunc返回TRUE，停止任何正在进行的遍历，这样就能将 GnodeTraverseFunc与g_node_traverse()结合起来按值搜索树。

函数列表：访问GNode

```
#include <glib.h>
/*对Gnode进行遍历*/
void g_node_traverse(GNode* root,
                GTraverseType order,
                GTraverseFlags flags,
                gint max_depth,
                GNodeTraverseFunc func,
                gpointer data)
/*返回GNode的最大高度*/
guint g_node_max_height(GNode* root)
/*对Gnode的每个子节点调用一次func函数*/
void g_node_children_foreach(GNode* node,
                        GTraverseFlags flags,
                        GNodeForeachFunc func,
                        gpointer data)
/*颠倒node的子节点顺序*/
void g_node_reverse_children(GNode* node)
/*返回节点node的子节点个数*/
guint g_node_n_children(GNode* node)
/*返回node的第n个子节点*/
GNode* g_node_nth_child(GNode* node,
                 guint n)
/*返回node的最后一个子节点*/
GNode* g_node_last_child(GNode* node)
/*在node中查找值为date的节点*/
GNode* g_node_find_child(GNode* node,
                  GTraverseFlags flags,
                  gpointer data)
/*返回子节点child在node中的位置*/
gint g_node_child_position(GNode* node,
                     GNode* child)
/*返回数据data在node中的索引号*/
gint g_node_child_index(GNode* node,
                   gpointer data)
/*以子节点形式返回node的第一个兄弟节点*/
GNode* g_node_first_sibling(GNode* node)
/*以子节点形式返回node的第一个兄弟节点*/
GNode* g_node_last_sibling(GNode* node)
```

3.5.3 哈希表

GHashTable是一个简单的哈希表实现，提供一个带有连续时间查寻的关联数组。要使用哈希表，必须提供一个GhashFunc函数，当向它传递一个哈希值时，会返回正整数：

typedef guint (*GHashFunc) (gconstpointer key);

返回的每个guint数值(表字节数的模数)对应于一个哈希表中的一个“存取窗口”或者“哈希表元”。GHashTable通过在每个“存取窗口”中存储一个键值对的链表来处理冲突。因而，GhashFunc函数返回的无符号整数值必须在可能取值中尽可能平均地分配，否则哈希表将

退化为一个链表。GHashFunc也必须快，因为每次查找都要用到它。

除了GhashFunc，还需要一个GcompareFunc比较函数用来测试关键字是否相等。不过，虽然GCompareFunc函数原型是一样的，但它在GHashTable中的用法和在GSList、Gtree中的用法不一样。在GHashTable中可以将GcompareFunc看作是等式操作符，如果参数是相等的，则返回TRUE。当哈希冲突导致在相同的“哈希表元”中有多个关键字－值对时，键比较函数用来找到正确的键值对。

使用下面函数列表中的函数创建和销毁一个GHashTable。注意，glib并不知道怎样销毁哈希表中保存的数据，它只销毁表本身。

函数列表：GHashTable

```
#include <glib.h>
GHashTable* g_hash_table_new(GHashFunc hash_func,
                GCompareFunc key_compare_func)
void g_hash_table_destroy(GHashTable* hash_table)
```

哈希表和比较函数支持最常用的几种键：整数、指针和字符串。这些都列在下面的函数列表中。针对整数的函数接收一个指向gint类型的指针，而不是gint整数值。如果将NULL作为哈希函数的参数传递给g_hash_table_new()，缺省情况下会使用g_direct_hash()函数。如果给键比较函数传递NULL参数，那么会使用简单的指针比较函数(等同于g_direct_equal()，但是没有函数调用)。

函数列表：哈希表/比较函数

```
#include <glib.h>
guint g_int_hash(gconstpointer v)
gint g_int_equal(gconstpointer v1,
           gconstpointer v2)
guint g_direct_hash(gconstpointer v)
gint g_direct_equal(gconstpointer v1,
              gconstpointer v2)
guint g_str_hash(gconstpointer v)

gint g_str_equal(gconstpointer v1,
           gconstpointer v2)
```

操纵哈希表很简单。插入函数不复制键或值，只是将给定的键值准确插入到哈希表，会覆盖任何已存在的具有相同键的键值对(记住，“相同”是由哈希表和比较函数决定的)。如果这样做有问题，在插入前必须查找或删除哈希表的键或值。如果动态分配键或值，需要特别注意。

如果g_hash_table_lookup()发现了与键相关联的值，返回这个值，否则，返回NULL。但有时不能这么做。例如，NULL可能本身就是一个有效的值。如果使用字符串，特别是动态分配字符串作为键，知道表里的一个键或许并不够，或许想检索出哈希表用来代表“ foo”键的确切的gchar*值。在这种情况下，可以使用g_hash_table_lookup_extended()。如果检索成功，g_hash_table_lookup_extended()返回TRUE；如果返回TRUE，则将它发现的键－值对放在给定的位置。

函数列表：处理GHashTable

```
#include <glib.h>
```

```
void g_hash_table_insert(GHashTable* hash_table,
                  gpointer key,
                  gpointer value)
void g_hash_table_remove(GHashTable * hash_table,
                  gconstpointer key)
gpointer g_hash_table_lookup(GHashTable * hash_table,
                  gconstpointer key)
gboolean g_hash_table_lookup_extended(GHashTable* hash_table,
                        gconstpointer lookup_key,
                        gpointer* orig_key,
                        gpointer* value)
```

GHashTable 保存一个内部数组，它的大小是质数。它保存存储在表中的键 -值对数的合计。如果每个有用的“ 哈希表元” 中的键值对平均个数降到 0.3以下，数组会变小；如果在 3以上，数组会变大以便减少冲突。不论何时从表中插入或删除键值对，数组都会自动调整大小。这确保了哈希表的内存使用是最优化的。然而，如果正在做大量的插入或删除，会反复重建哈希表，这会急剧降低效率。为了解决这个问题，哈希表可以被“ 冻结”，即临时禁止调整数组大小。当添加和删除条目已经完成时，简单地“ 解冻” 表，这时会进行一次优化计算。注意，如果添加大量数据，由于哈希冲突，“ 冻结” 的表会“ 死” 掉。在做任何查找以前将表“ 解冻” 就会一切正常。

函数列表：冻结和解冻GHashTable

```
#include <glib.h>
/**冻结哈希表/
void g_hash_table_freeze(GHashTable* hash_table)
/*将哈希表解冻*/
void g_hash_table_thaw(GHashTable* hash_table)
```

3.6 GString

除了使用gchar *进行字符串处理以外，Glib还定义了一种新的数据类型：GString。它类似于标准C的字符串类型，但是GString能够自动增长。它的字符串数据是以NULL结尾的。这些特性可以防止程序中的缓冲溢出。这是一种非常重要的特性。下面是 GString的定义：

```
struct GString
{
  gchar *str; /* Points to the string's current \0-terminated value. */
  gint len; /* Current length */
};
```

用下面的函数创建新的GString变量：

```
GString *g_string_new( gchar *init );
```

这个函数创建一个 GString，将字符串值 init复制到GString中，返回一个指向它的指针。如果init参数是NULL，创建一个空GString。

```
void g_string_free( GString *string,
                  gint      free_segment );
```

这个函数释放 string所占据的内存。 free_segment参数是一个布尔类型变量。如果free_segment参数是TRUE，它还释放其中的字符数据。

```
GString *g_string_assign( GString      *lval,
                          const gchar *rval );
```

这个函数将字符从rval复制到lval，销毁lval的原有内容。注意，如有必要， lval会被加长以容纳字符串的内容。这一点和标准的字符串复制函数 strcpy()相同。

下面的函数的意义都是显而易见的。其中以 _c结尾的函数接受一个字符，而不是字符串。

截取string字符串，生成一个长度为len的子串：

```
GString *g_string_truncate( GString *string,
                            gint     len );
```

将字符串val追加在string后面，返回一个新字符串：

```
GString *g_string_append( GString *string,
                          gchar    *val );
```

将字符c追加到string后面，返回一个新的字符串：

```
GString *g_string_append_c( GString *string,
                            gchar    c );
```

将字符串val插入到string前面，生成一个新字符串：

```
GString *g_string_prepend( GString *string,
                           gchar    *val );
```

将字符c插入到string前面，生成一个新字符串：

```
GString *g_string_prepend_c( GString *string,
                             gchar    c );
```

将一个格式化的字符串写到string中，类似于标准的sprintf函数：

```
void g_string_sprintf( GString *string,
                       gchar   *fmt,
                       ...);
```

将一个格式化字符串追加到string后面，与上一个函数略有不同：

```
void g_string_sprintfa ( GString *string,
                         gchar   *fmt,
                         ... );
```

3.7 计时器函数

计时器函数可以用于为操作计时（例如，记录某项操作用了多长时间）。使用它的第一步是用g_timer_new()函数创建一个计时器，然后使用 g_timer_start()函数开始对操作计时，使用g_timer_stop()函数停止对操作计时，用 g_timer_elapsed()函数判定计时器的运行时间。

创建一个新的计时器：

```
GTimer *g_timer_new( void );
```

销毁计时器：

```
void g_timer_destroy( GTimer *timer );
```

开始计时：

```
void g_timer_start( GTimer  *timer );
```

停止计时：

```
void g_timer_stop( GTimer  *timer );
```

计时重新置零：

```
void g_timer_reset( GTimer  *timer );
```

获取计时器流逝的时间：

```
gdouble g_timer_elapsed( GTimer *timer,
                         gulong *microseconds );
```

3.8 错误处理函数

```
gchar *g_strerror( gint errnum );
```

返回一条对应于给定错误代码的错误字符串信息，例如“ no such process”等。输出结果一般采用下面这种形式：+

程序名：发生错误的函数名:文件或者描述：strerror

下面是一个使用g_strerror函数的例子：

```
g_print("hello_world:open:%s:%s\n", filename, g_strerror(errno));
void g_error( gchar *format, ... );
```

打印一条错误信息。格式与printf函数类似，但是它在信息前面添加“ ** ERROR **: ”，然后退出程序。它只用于致命错误。

```
void g_warning( gchar *format, ... );
```

与上面的函数类似，在信息前面添加“ ** WARNING **:”，不退出应用程序。它可以用于不太严重的错误。

```
void g_message( gchar *format, ... );
```

在字符串前添加“ message: ”，用于显示一条信息。

```
gchar *g_strsignal( gint signum );
```

打印给定信号号码的Linux系统信号的名称。在通用信号处理函数中很有用。

3.9 其他实用函数

glib还提供了一系列实用函数，可以用于获取程序名称、当前目录、临时目录等。这些函数都是在glib.h中定义的。

```
/*返回应用程序的名称*/
gchar* g_get_prgname (void);
/*设置应用程序的名称*/
void  g_set_prgname (const gchar *prgname);
/*返回当前用户的名称*/
gchar* g_get_user_name (void);
/*返回用户的真实名称。该名称来自“ passwd”文件。返回当前用户的主目录*/
gchar* g_get_real_name (void);
/*返回当前使用的临时目录，它按环境变量TMPDIR、TMP and TEMP的顺序查找。如果上面的环境变量都
  没有定义，返回“ /tmp”*/
gchar* g_get_home_dir (void);  gchar* g_get_tmp_dir (void);
```

```
/*返回当前目录。返回的字符串不再需要时应该用 g_free ( )释放*/
gchar* g_get_current_dir (void);
/*获得文件名的不带任何前导目录部分的名称。它返回一个指向给定文件名字符串的指针 */
gchar* g_basename (const gchar *file_name);
/*返回文件名的目录部分。如果文件名不包含目录部分，返回“ .”。返回的字符串不再使用时应该用 g_free
  ( ) 函数释放*/
gchar* g_dirname (const gchar *file_name);
/*如果给定的file_name是绝对文件名（包含从根目录开始的完整路径，比如 /usr/local)，返回TRUE*/
gboolean g_path_is_absolute (const gchar *file_name);
/*返回一个指向文件名的根部标志（ /”）之后部分的指针。如果文件名 file_name不是一个绝对路径，返回
  NULL*/
gchar* g_path_skip_root (gchar *file_name);
/*指定一个在正常程序终止时要执行的函数 */
void g_atexit (GVoidFunc func);
```

上面介绍的只是 glib库中的一小部分， glib的特性远远不止这些。如果想了解其他内容，请参考glib.h文件。这里面的绝大多数函数都是简明易懂的。另外， http://www.gtk.org上的glib文档也是极好的资源。

如果你需要一些通用的函数，但 glib中还没有，考虑写一个 glib风格的例程，将它贡献到glib库中！你自己，以及全世界的 glib使用者，都将因为你的出色工作而受益。