精通嵌入式 Linux 编程

——构建自己的 GUI 环境

李玉东　著

（第零版）

http://www.lgui.net

版本说明

本书尚在初稿阶段，有些章节尚未全部完成。鉴于有些网友的要求，先将这一版本发布。由于时间关系，其中难免有很多错误与不足之处。如果你有任何建议，一定要告诉我。我希望在后续的版本中能将“构建嵌入式 Linux 的 GUI 环境”这个问题讲得更加透彻一点。

master@lgui.net 或 lguitech@126.com

源代码下载：

http://www.lgui.net/download/lgui_0.3.0.tgz

作者简介

李玉东：1969 年生人，获北京大学软件硕士学位

目录

序言

第一个问题，为什么要写这本书？

现在很多面向嵌入式 Linux 的编程书籍大都泛泛而谈，并不针对某一个领域，对于解决某一个领域的特定问题没有指导性。例如，C 语言的书一大堆，但我们不是只为学习 C 语言而学习 C 语言，阅读者并不仅仅是想掌握 C 语言的编程规范，而是用 C 语言在某一个平台上实现某种功能、解决某种问题。

利用嵌入式 Linux 来构造系统，一般来说以便携式终端产品居多，就软件方面而言，这类产品主要有两种环节需要把握：一是 Linux 的内核（包括驱动）的移植；另一方面就是 GUI 层与应用层软件的设计。

有些人说，嵌入式 Linux 的最大问题是其 GUI 没有统一标准，我不知道这是他的缺点还是优点，是否应该由如 Microsoft 或 Nokia 这种级别的公司在这个操作系统平台上构造一个全世界都一样的用户界面，然后我们大家都它的 API 来开发应用程序呢？这个问题我不想讨论。据我所知的情况，基于一个原则，就是嵌入式产品对于界面的需要简直是太不一样了，MP3、MP4、导航仪、电视机顶盒、手机等等，五花八门，如果所有的界面都从“开始”菜单开始，我不知道操作起来是不是都很方便，是用手指、遥控器、鼠标还是别的什么东西。所以，对于嵌入式产品，我认为个性化用户界面才是合适的，那么——为什么自己不去构造一个属于自己的 GUI 系统呢？例如，你所在的公司是做手机的，或你所在的公司是做机顶盒的，那么，为什么不开发一个小型的 GUI 库呢，为什么要说**用某某某种** GUI 系统，既然在嵌入式环境对于 GUI 系统的需求千差万别，而任何一个 GUI 都不可能有如此好的适应性和可配置性，那么把一个 PDA 风格的 GUI 系统移植到机顶盒上到底有什么意义，把一个手机风格的 GUI 移植到工控机里又有什么意义？所以，最简单的办法，就是自己构造一个小型的 GUI 环境，只针对你的应用，与其他系统无关。

那么，可能有人会说，量体裁衣开发一个适合于自有项目的 GUI 环境固然很好，但这会不会很复杂，是不是会使项目周期拉长？这本书正是要告诉你，开发一个小型的嵌入式 GUI 系统其实很容易！何况网络上有如此之多的开源代码可供参考。当然无偿 Copy 开源软件用于商业目的是不允许的，但思想是自由的，这一点我想谁也否认不了——“我怀疑你看过某某代码，所以要 GPL”的说法我认为是荒唐的！

另外，现在已经开发完并开源的面向嵌入式 Linux 的 GUI 系统固然很多，而且还有一些技痒的人又在开发这个“柜”那个“柜”的，但没有人仔细讨论到底

一个嵌入式 Linux 的 GUI 系统的体系结构如何，能让使用者从全局把握系统，从而能开发出自己的 GUI 环境，而不是 e 柜、f 柜、g 柜……这样开发下去。我如果说这是个“鱼”和“渔”的关系，不知大家认不认可！

所以我写了这本书——通过嵌入式 Linux 特定环节的应用实例，即中间件层的 GUI 软件，来精通 Linux 开发，同时对于消息驱动的、轻量级窗口系统的实现有较为彻底的理解。

第二个问题，这本书都什么特点？

我只针对 GUI 这个环节讨论其中的技术问题，讨论如何在嵌入式 Linux 上实现，用到了 Linux 开发的那些技术细节，所以第一个特点是针对性强；

另外，我不想为了凑页数而把这本书搞成一个 Linux 编程的百科全书，讲清楚一个问题是最重要的，所以第二个特点是没有废话。

最后，这本书写出来，先放到网上，谁想看都行，不为赚钱，只希望对大家都所帮助。只有大家愿意承认我做的工作有意义就行。

——李玉东

第1章 概论

我想我们没有必要如同所有 Linux 书籍一样，从 Linux 的来源说起，并引用一些 Linus 本人的话来装点门面。但我想，介绍一下 GUI 在嵌入式 Linux 系统中所处的位置以及目前有什么系统可供我们借鉴，这个步骤还是不能省略，如果你对这个环节比较了解，麻烦你转到其他章节阅读。

1.1 嵌入式系统的基本概念

在现在日益信息化的社会中，计算机与网络已经渗透到我们日常生活的每一个方面，而嵌入式系统，正是这个渗透过程的主要推动力量。与我们生活息息相关的家用电器、汽车电子、我们随身携带的手机、MP3、手表、PDA、数码相机、数码录像机，这一切都与嵌入式系统密切相关；而在工业领域，使用嵌入式设备控制的生产流水线、数字机床、智能工具也正在其中扮演着极其重要的角色。

嵌入式系统的一般定义是： “以应用为中心、以计算机技术为基础、软件硬件可裁剪、适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统”。

广义上讲，凡是带有微处理器的专用软硬件系统都可以称为嵌入式系统。所以也有人简单的说:“嵌入式系统是指操作系统和功能软件集成于计算机硬件系统之中。”

狭义上讲，人们更加强调那些使用嵌入式微处理器构成独立系统，具有自己的操作系统并且具有某些特定功能的系统。

1.2 嵌入式系统的特征

与通用计算机不同，嵌入式系统是针对具体应用的专用系统，一般具有成本敏感性，它的硬件和软件必须高效地设计，好的嵌入式系统是完成目标功能的最小系统。

嵌入式系统一般要求高的可靠性，例如在高温、高压、电磁干扰严重的工业环境就对嵌入式系统有很高的要求；

嵌入式处理器的功耗、体积、处理能力在具体应用中也有很高的要求，这在

消费类电子产品方面的表现非常明显。嵌入式处理器要针对用户的具体需求，对芯片配置进行裁减和添加，才能达到理想的效果。

嵌入式系统软件与嵌入式应用软件也与通用计算机有所不同。一般嵌入式软件要求高质量的代码与高可靠性。另外，许多嵌入式应用系统要求系统软件具有实时处理能力，在多任务嵌入式系统中，对重要性各不相同的任务进行统筹兼顾的合理调度是保证每个任务及时执行的关键。

1.3 选择 Linux 构造嵌入式系统

Linux 从 1991 年问世到现在，短短的十几年时间已经发展成为功能强大、设计完善的操作系统之一，不仅可以与各种传统的商业操作系统分庭抗争，在新兴的嵌入式操作系统领域内也获得了飞速发展。

嵌入式 Linux 的开发和研究是操作系统领域中的一个热点，目前已经开发成功的嵌入式系统中，大约有一半使用的是 Linux。Linux 之所以能在嵌入式系统市场上取得如此辉煌的成果，与其自身的优良特性是分不开的。

一、广泛的硬件支持。Linux 能够支持 x86、ARM、MIPS、ALPHA、PowerPC 等多种体系结构，目前已经成功移植到数十种硬件平台，几乎能够运行在所有流行的 CPU 上。Linux 有着异常丰富的驱动程序资源，支持各种主流硬件设备和最新硬件技术，甚至可以在没有存储管理单元（MMU）的处理器上运行，这些都进一步促进了 Linux 在嵌入式系统中的应用。

二、内核高效稳定。Linux 内核的高效和稳定已经在各个领域内得到了大量事实的验证，Linux 的内核设计非常精巧，分成进程调度、内存管理、进程间通信、虚拟文件系统和网络接口五大部分，其独特的模块机制可以根据用户的需要，实时地将某些模块插入到内核或从内核中移走。这些特性使得 Linux 系统内核可以裁剪得非常小巧，很适合于嵌入式系统的需要。

三、开放源码，软件丰富。Linux 是开放源代码的自由操作系统，它为用户提供了最大限度的自由度，由于嵌入式系统千差万别，往往需要针对具体的应用进行修改和优化，因而获得源代码就变得至关重要了。Linux 的软件资源十分丰富，每一种通用程序在 Linux 上几乎都可以找到，并且数量还在不断增加。在 Linux 上开发嵌入式应用软件一般不用从头做起，而是可以选择一个类似的自由软件做为原型，在其上进行二次开发。

四、优秀的开发工具。开发嵌入式系统的关键是需要有一套完善的开发和调试工具。传统的嵌入式开发调试工具是在线仿真器（In-Circuit Emulator，ICE），它通过取代目标板的微处理器，给目标程序提供一个完整的仿真环境，从而使开

发者能够非常清楚地了解到程序在目标板上的工作状态，便于监视和调试程序。在线仿真器的价格非常昂贵，而且只适合做非常底层的调试，如果使用的是嵌入式 Linux，一旦软硬件能够支持正常的串口功能时，即使不用在线仿真器也可以很好地进行开发和调试工作，从而节省了一笔不小的开发费用。嵌入式 Linux 为开发者提供了一套完整的工具链（Tool Chain），它利用 GNU 的 gcc 做编译器，用 gdb、kgdb、xgdb 做调试工具，能够很方便地实现从操作系统到应用软件各个级别的调试。

五、完善的网络通信和文件管理机制。Linux 从诞生之日起就与 Internet 密不可分，支持所有标准的 Internet 网络协议，并且很容易移植到嵌入式系统当中。此外，Linux 还支持 ext2、fat16、fat32、romfs 等文件系统，这些都为开发嵌入式系统应用打下了很好的基础。

1.4 GUI 在嵌入式 Linux 系统中的地位及要求

随着近年来手持式和家用型消费类电子产品的发展，人们对这些产品的用户界面产生了新的需求，例如：手机、PDA、便携式媒体播放器、家庭多媒体娱乐中心、数字机顶盒、DVD 播放器等等。以前，这类产品的用户界面都比较简单，而现在，我们可以看到，大部分产品都需要有漂亮的图形用户界面，甚至要求能够支持全功能的浏览器，使得用户能够随时随地进行网络信息的浏览。但是，由于消费类电子的成本敏感性特点，这些产品大多数希望建立在一个有限占用系统资源的轻量级 GUI 系统之上，这与 PC 机中 GUI 系统有根本性的区别。

另外一个轻量级 GUI 系统的需求存在于工业控制领域、由于工业控制领域对实时性的要求比较高，所以这些系统也不希望建立在庞大的、响应迟缓的 GUI 系统之上。尤其是在实时 Linux 系统出现以后，由于 Linux 系统的稳定性、可靠性、易移植性以及其广泛的软硬件支持，Linux 系统在工业领域也得到越来越多的应用，而一个轻量级的 GUI 系统也正是这类系统所需要的。

从用户的观点来看，GUI 是系统的一个最至关重要的方面：用户通过 GUI 与系统进行交互，所以 GUI 应该易于使用并且非常可靠，而且它还需要有内存意识，可以在内存受限的、微型嵌入式设备上运行。

从二次开发者的角度看，GUI 是一个友好的开发环境，开发者无需经过艰苦的学习就能适应开发过程，这样才能使得基于此平台的应用很快地丰富起来。对

于二次开发商而言，也才有兴趣使用此产品为终端产品制造商提供解决方案。

另外，必须清楚的是，嵌入式系统往往是一种定制设备，它们对 GUI 的需求也各不相同。有的系统只要求一些图形功能，而有些系统要求完备的 GUI 支持。因此，GUI 也必须是可定制的。

从系统的体系结构来看,GUI 系统属于应用层的软件系统，但通常而言，GUI 有别于一个简单的图形库，一个 GUI 系统通常会有自己的应用开发模式，从这个意义上讲，GUI 应该属于中间件的范畴。

GUI 在整个系统中所处的位置如下图所示：

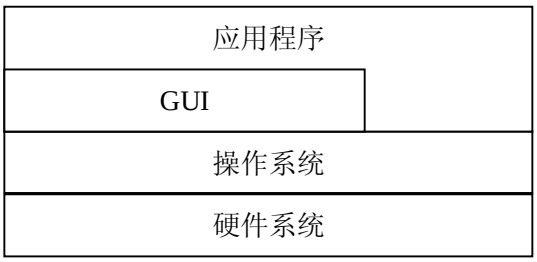

图 1-1 GUI 在系统中所处的位置

1.5 用户界面概况

1.5.1 用户界面的历史

计算机用户界面是指计算机与其使用者之间的对话接口，是计算机系统的重要组成部分。计算机的发展史不仅是计算机本身处理速度和存储容量飞速提高的历史，而且是计算用户界面不断改进的历史。早期的计算机是通过面板上的指示灯来显示二进制数据和指令，人们则通过面板上的开关、扳键及穿孔纸带送入各种数据和命令。50 年代中、后期，由于采用了作业控制语言(JCL)及控制台打字机等，使计算机可以批处理多个计算任务，从而代替了原来笨拙的手工扳键方式，提高了计算机的使用效率。

1963 年，美国麻省理工学院在 709/7090 计算机上成功地开发出第一个分时系统 CTSS，该系统连接了多个分时终端，并最早使用了文本编辑程序。从此，以命令行形式对话的多用户分时终端成为 70 年代乃至 80 年代用户界面的主流。

80 年代初，由美国 Xerox 公司 Alto 计算机首先使用的 Smalltalk—80 程序

设计开发环境，以及后来的 Lisa、Macintosh 等计算机，将用户界面推向图形用户界面的新阶段。随之而来的用户界面管理系统和智能界面的研究均推动了用户界面的发展。用户界面已经从过去的人去适应笨拙的计算机，发展到今天的计算机不断地适应人的需求。

用户界面的重要性在于它极大地影响了最终用户的使用，影响了计算机的推广应用，甚至影响了人们的工作和生活。由于开发用户界面的工作量极大，加上不同用户对界面的要求也不尽相同，因此，用户界面已成为计算机软件研制中最困难的部分之一。当前，Internet 的发展异常迅猛，虚拟现实、科学计算可视化及多媒体技术等对用户界面提出了更高的要求。

1.5.2 图形用户界面的特征

图形用户界面（GUI）的广泛流行是当今计算机技术的重大成就之一，它极大地方便了非专业用户的使用，人们不再需要死记硬背大量的命令，而可以通过窗口、菜单方便地进行操作 。“Visual”已成为当前最流行的形容词，如 Visual Basic、Visual C＋＋等。为什么图形用户界面受到如此青睐？它的主要特征是什么？

WIMP

其 中：

W(Windows) 指窗口，是用户或系统的一个工作区域。一个屏幕上可以有多个窗口。

I(Icons)指图符，是形象化的图形标志。

M(Menu)指菜单，可供用户选择的功能提示。

P(Pointing Devices) 指鼠标器等，便于用户直接对屏幕对象进行操作。

用户模型

GUI 采用了不少 Desktop 桌面办公的隐喻，使应用者共享一个直观的界面框架。由于人们熟悉办公桌的情况，因而对计算机显示的图符的含义容易理解，诸如：文件夹、收件箱、画笔、工作簿、钥匙及时钟等。

直接操作

过去的界面不仅需要记忆大量命令，而且需要指定操作对象的位置，如行号、空格数、X 及 Y 的坐标等。采用 GUI 后，用户可直接对屏幕上的对象进行操作，如拖动、删除、插入以至放大和旋转等。用户执行操作后，屏幕能立即给出反馈信息或结果，因而称为所见即所得(What You See Is What You Get) 。用视、点(鼠 标)代替了记、击(键盘) ， 给用户带来了方便。

1.5.3 图形用户界面系统的结构模型

一个图形用户界面系统通常由三个基本层次组成。它们是显示模型，窗口模型和用户模型。用户模型包含了显示和交互的主要特征，因此图形用户界面这一术语有时也仅指用户模型。图 2-1 给出了图形用户界面系统的层次结构。

图 2-1 中的最底层是计算机硬件平台。硬件平台的上面是计算机的操作系统。大多数图形用户界面系统都只能在一两种操作系统上运行，只有少数的产品例外。

操作系统之上是图形用户界面的显示模型。它决定了图形在屏幕上的基本显示方式。不同的图形用户界面系统所采用的显示模型各不相同。例如大多数在 UNIX 之上运行的图形用户界面系统都采用 X 窗口作显示模型；MS Windows 则采用 Microsoft 公司自己设计的图形设备接口(GDI)作显示模型。

图 1-2 图形用户界面系统的层次结构

显示模型之上是图形用户界面系统的窗口模型。窗口模型确定窗口如何在屏幕上显示，如何改变大小，如何移动，及窗口的层次关系等。它通常包括两个部分：一是编程工具；二是对如何移动、输出和读取屏幕显示信息的说明。因为 X 窗口不但规定了如何显示基本图形对象也规定了如何显示窗口，所以它不但可以充当图形用户界面的显示模型，也可以充当它的窗口模型。

窗口模型之上是用户模型，图形用户界面的用户模型又称为图形用户界面的视感。它也包括两个部分：一是构造用户界面的工具；二是对于如何在屏幕上组织各种图形对象，以及这些对象之间如何交互的说明。比如，每个图形用户界面模型都会说明它支持什么样的菜单和什么样的显示方式。

图形用户界面系统的应用程序接口由其显示模型，窗口模型和用户模型的应用程序接口共同组成。例如 OSF/Motif 的应用程序接口就是由它的显示模型和窗口模型的应用程序接口 Xlib 和用户模型的应用程序接口 Xt Intrinsics 及 Motif

Toolkit 共同组成的。

1.5.4 用户界面的发展：GUI＋新人机交互技术

人机交互是研究人、计算机以及它们相互影响的技术。随着计算机处理、存储能力的飞速提高和体积、成本的降低，人们已将注意力逐渐转移到改善人机交互的手段和界面方面，越来越期望计算机来适应人的习惯和要求。人们对于无所不在的计算要求，使手持移动计算逐渐成为当今主流的计算模式之一，人机交互的效率和自然性已经成为了移动计算普及和应用中最为核心的问题之一。能够使用听、说、写等日常生活中的基本技能于计算机操作，是提高计算机的可用性、友好性和自然性的重要方面。

随着虚拟现实、科学计算可视化及多媒体技术的飞速发展，新的人机交互技术不断出现，更加自然的交互方式将逐渐为人们所重视，尤其体现在手持设备里面。手持设备自身的物理特征决定了纯粹基于 WIMP 界面的交互方式很难在移动计算环境中取得良好的效果。交互设备和交互手段的限制降低了信息输入效率，而过小的屏幕又给输出信息的呈现造成了困难。与交互效率降低并存的还有交互自然性的下降，用户使用指点工具在过小的屏幕上进行点击、选择等精确操作，增加了交互过程中的认知负担，从而容易导致用户的厌烦心理。因此，提高移动系统人机交互的效率和自然性，是一个值得关注和研究的问题。

多通道人机交互(multi-modal human-computer interaction) 是一种使用多种通道与计算机通信的人机交互方式。多通道人机交互是提高交互效率和自然性的有效途径[16]。通道（modality,也有译为模态、模式）涵盖了用户表达意图、执行动作或感知反馈信息的各种通信方法，如言语、眼神、脸部表情、唇动、手动、手势、头动、肢体姿势、触觉、嗅觉或味觉等。多通道人机交互是提高交互效率和自然性的有效途径。单一的交互方式是导致交互效率低下的一个重要原因，多通道用户界面允许用户通过各种不同的交互通道以及它们之间的相互组合、协作来完成交互任务，这正好弥补了单一通道交互给用户带来的限制和负担。

多通道人机交互比传统 WIMP 界面适用于更多领域的应用程序以及更广泛的用户群[17]。笔和按键是目前手持设备上使用最广泛的输入手段，它们都是单一通道的输入模式，而单纯的图形文本显示也是一种单通道的输出方式。而单一的交互方式是导致交互效率低下的一个重要原因，多通道用户界面允许用户通过各种不同的交互通道以及它们之间的相互组合、协作来完成交互任务，这正好弥补了单一通道交互给用户带来的限制和负担。同时，移动设备所访问和处理的信息由各种不同的媒体承载，在这样一个混合媒体的环境中，系统需要多个通道来对

应各种不同的信息类型。

1.6 各种嵌入式 Linux 上的图形及 GUI 系统介绍

1.6.1 Linux 基本图形系统（函数库）

这些系统（或者函数库）包括：X Window、SVGALib、FrameBuffer 等等。这些系统（或者函数库）一般作为其他高级图形或者图形应用程序的基本函数库。

1. X Window

X Window 是由麻省理工学院（MIT）推出的窗口系统，简称 X，它旨在建立不依赖于特定硬件系统的图形和文字显示窗口系统的标准。1987 年 9 月，MIT 推出了 X 系统的 11 版，称为 X11，它的出现标志着计算机工作站的一个新时代的到来。现在几乎所有的工作站都采用了 X 窗口的标准，几乎所有的工作站上的应用软件都采用了基于 X Window 的软件平台。同时，微机的 X 系统也日益增多。

X 窗口系统之所以能受到人们的广泛青睐，是与其优越的特点分不开的。首先，它不依赖与硬件系统的特点，使我们在任意一种计算机上用 X 系统开发的图形软件，可以不需任何修改或只需极少改动就能移植到几十种其它类型的计算机上。其次，X 是一种基于网络的窗口系统，采用 X 的应用软件可以在由不同机器组成的网络上运行。我们能方便地在远程计算机上运行软件，而将结果显示到本机上。

基于 X 的应用软件是通过调用 X 的一系列 C 语言函数实现其各种功能的。这些函数称为 Xlib（X 库），它提供了建立窗口、画图、处理用户操作事件等基本功能。Xlib 是一种底层库，用它来编写图形和交互界面程序虽非常灵活，但却比较复杂甚至繁琐。为此又发展出了一些比 Xlib 更高层的库函数，称作工具包(Toolkits)，它们将一些常用的界面图形（如窗口、菜单、按键等，通常称作工具包中的组件（widgets））按面向对象编程的方式组织到一起供应用软件使用，而工具包的 Intrinsics(内在、本质)允许我们在它们之上建立新的 widget。

Xlib、X Intrinsic 以及 Toolkits 之间的关系是：Xlib 控制 X 协议以及网络问题，对开发者而言只是一些非常原始的接口函数，只提供基本的 C 语言 API。在 Xlib 上是 X Intrinsic，X Intrinsic 为高层的 Toolkits 提供面向对象的框架，如果需要自己开发一套 Toolkits，可以从这里开始。

再上就是 Toolkits 库。Toolkits 则提供完整的用户界面开发包，里面有了“菜单”、“按钮”等基本的窗口对象，这些常被称作 Widget。开发者编写的程序

可以基于上面三种的任何一种进行开发：其于 Xlib 库（工作量很大），基于 X Intrinsic（仍然很困难），Toolkit/Widgets（种类较多，开发起来也相对容易得多）。基本的结构如图所示：

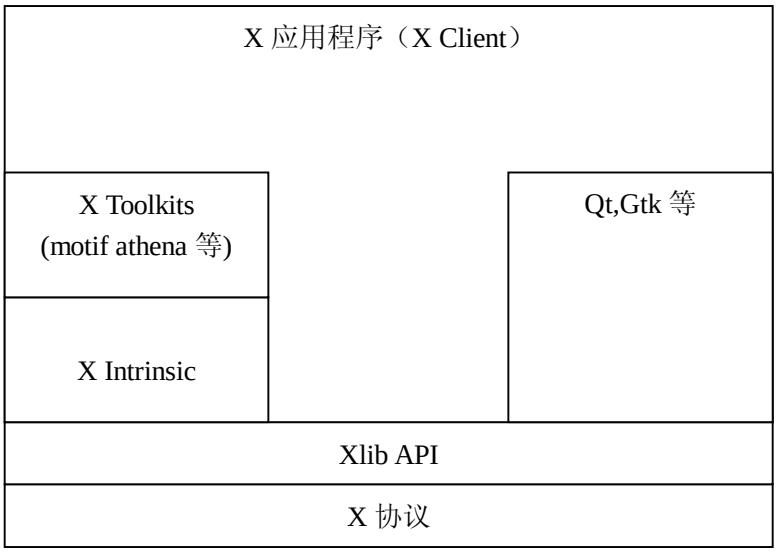

图 1-3 开发 X 系统时的基本结构图

X 下的程序设计并不困难，但如果只是基于 Xlib，则相当于使用汇编语言开发程序，工作量比较大。如果界面要求不复杂，注重效率，可以使用这种方式。如果需要开发工具有完整 Windows 风格的程序，最好还是选用其他方法。基于 X Toolkits 进行开发以前是 X 程序开发的主流，不过 X Toolkits 提供的面向对象特征并不强，而且调用函数多，概念多，不容易上手。Qt、Gtk 是在自由软件浪潮中发展起来的，具有非常明显的面向对象的特点，而且直接基于 Xlib，封装完整，特别是 Qt，采用 C++类作为接口 API，则于它界面美观（可以很像 Windows）、开发时间短（可以使用 VC 一类的图形工具画窗口）、运行效率高（直接基于 Xlib）等特点，已经成为目前进行 X Window 程序设计的首选。

对用户来讲，所接触的 X 系统首先是在计算机显示器上的窗口系统，事实上，X 系统的概念是多方面的，X Window 是一种协议，通过它，应用程序可以在支持位图显示和能接受输入的计算机上产生输出。X Window 系统建立在一种客户—服务器模型之上，这里，应用程序是客户，它通过 X 协议与服务器联系，服务器承担直接向显示产生输出和接受输入的工作。另外，X 系统是一种基于网络的窗口系统。一个基于 X 系统的应用程序既可以在本机上运行，也可以在另外一台机器

上运行，通过网络（TCP/IP 或 DECnet 网络）将输出与输入的工作交到任一指定的显示终端上。这里，控制显示的程序称作 Server 或 X Server（X 服务器），它是直接与计算机硬件平台打交道的程序。X 服务器起着在本地机或远程机上运行的用户应用程序和执行输出输入的本地机的资源之间的桥梁作用。

X Window 系统中经常被混淆的一个关键概念是服务器与客户之间的区别。在计算机网络中，服务器指的是向其他机器传输文件的机器，然而 X Window 系统中的服务器起的是完全不同的作用。X Window 系统中，服务器是从用户处接收输入并向用户显示输出的硬件（或）软件。例如，用户面前的键盘、鼠标和显示器就是服务器的一部分，即他们图形化地向用户提供信息“服务”。客户是指连接到服务器的应用程序。

在 X Window 系统中，服务器与客户可以存在于同一个工作站或者计算机上，使用进程间通信（IPC）机制，如 UNIX 管道与套接字，在它们之间传递信息。本地客户是指运行在用

户面前的机器上的应用程序。远程客户是指运行在通过网络连接到用户的服务器上的应用程序。不管是本地客户还是远程客户，对于 X Window 系统的用户来讲，外观与感受都是完全一样的。

图 2-3 描绘了三个客户应用程序 X、Y、Z，它们均在 X Window 系统的服务器上显示输出。每个应用程序都在不同的机器上运行。远程客户 X 运行在通过局域网连接到 X Window 系统的机器上；远程客户 Y 运行在通过广域网连接到 X Window 系统的机器上；本地客户 Z 直接运行在作为服务器的工作站上，使用 UNIX 套接字在服务器显示器上显示输出请求。

X Window 作为一个图形环境是成功的，它上面运行着包括 CAD 建模工具和办公套件在内的大量应用程序。但是，由于 X Window 在体系接口上的原因，限制了其对游戏、多媒体的支持能力。

Tiny-X 是 XServer 在嵌入式系统的小巧实现，它由 Xfree86 Core Team 的 Keith Packard 开发。它的目标是运行于小内存系统环境。典型的运行于 X86 CPU 上的 Tiny-X Server 尺寸接近（小于）1MB。

上的 Tiny-X Server 尺寸接近（小于）1MB。

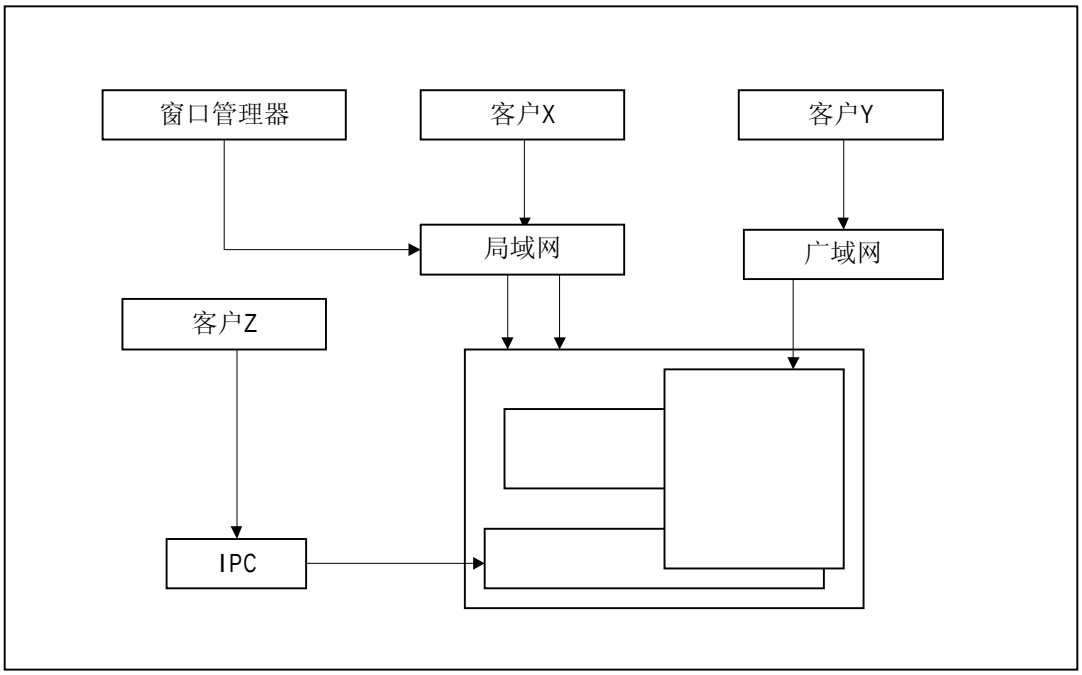

图 1-4 X Window 客户与服务器拓朴图

2. SVGALib[8]

SVGALib 是 Linux 上底层的图形库，也是 Linux 系统中最早出现的非 X 图形支持库，它支持标准的 VGA 图形模式和一些其他的模式，SVGALib 的缺点是程序必须以 root 权限登录，并且它是基于图形卡的，所以不是所有的硬件都支持它。自从 framebuffer 这个孪生姐妹诞生后，许多软件由只支持 SVGALib 改变为同时支持两者，甚至一些流行的高层函数库如 QT 和 GTK 只支持 Framebuffer，作为一个老的图形支持库，SVGALib 目前的应用范围越来越小，尤其是在 Linux 内核增加了 FrameBuffer 驱动支持之后。

3. FrameBuffer

FrameBuffer 是出现在 2.2.xx 内核当中的一种驱动程序接口。这种接口将显示设备抽象为帧缓冲区。用户可以将它看成是显示内存的一个映像，将其映射到进程地址空间之后，就可以直接进行读写操作，而写操作可以立即反映在屏幕上。该驱动程序的设备文件一般是 /dev/fb0、/dev/fb1 等等。

在应用程序中，一般通过将 FrameBuffer 设备映射到进程地址空间的方式使用，比如下面的程序就打开 /dev/fb0 设备，并通过 mmap 系统调用进行地址映射，随后用 memset 将屏幕清空（这里假设显示模式是 1024x766-8 位色模式，线性内存模式）：

```
int fb;
unsigned char* fb_mem;
fb = open ("/dev/fb0", O_RDWR);
fb_mem = mmap (NULL, 1024*768,
     PROT_READ|PROT_WRITE,MAP_SHARED,fb,0);
memset (fb_mem, 0, 1024*768);
```

图 1-5 Framebuffer 的调用方法

FrameBuffer 设备还提供了若干 ioctl 命令，通过这些命令，可以获得显示设备的一些固定信息（比如显示内存大小）、与显示模式相关的可变信息（比如分辨率、象素结构、每扫描线的字节宽度），以及伪彩色模式下的调色板信息等等。

FrameBuffer 实际上只是一个提供显示内存和显示芯片寄存器从物理内存映射到进程地址空间中的设备。所以，对于应用程序而言，如果希望在 FrameBuffer 之上进行图形编程，还需要完成其他许多工作。FrameBuffer 就像一张画布，使用什么样子的画笔，如何画画，还需要你自己动手完成。

4. LibGGI

GGI，即 General Graphics Interface，是新一代的图形支持库。

GGI 的主要功能特性有：

可在 FrameBuffer，SVGALib，X 等设备上运行，在这些设备上是二进制兼容的；

在所有平台上提供了一致的输入设备接口，比如鼠标和键盘；

与 LinuxThreads 线程库兼容，接口线程安全；

提供异步绘制模式，可提高屏幕刷新速度；

提供良好的颜色处理接口；

接口简单易用；

采用共享库机制，实现底层支持库的动态装载；

GGI 的主要不足在于安装和配置较为复杂。

1.6.2 面向嵌入式 Linux 系统的图形用户界面

1. Qt/Embedded

Qt 是著名的 Qt 库开发商 TrollTech 发布是一个跨平台的 C++图形界面应用程序框架。它实际上是一个类库，里面包括了大量的可重用类，其中既有按钮、窗口等这些可见类，也有定时器这样的不可见类和一些抽象类。Qt 是完全面向对象，拥有良好的扩展性与稳定性，并支持模块化编程。

利用 Qt 编程，可以充分利用其高度面向对象和模块化的特征，从繁琐的 X 编程中解脱出来，专注于程序本身的内容，使 Linux 下窗口程序设计成为一件非常轻松的事情。

Qt 中的各种窗口可以根据参数以几种不同的风格显示，例如 MS Windows 风格、CDE 风格等。几乎所有的控件都可以在 Qt 中找到相应的类。

关于对象间通信的问题，Qt 采取了一种被称作“Signal-Slot”的方式，这是 Qt 的重要特征之一。在 MS-Windows 中，程序通过消息队列和消息循环的方式进行消息的传递与事件的触发，而 Qt 的“Signal-Slot”机制采取了这样的方式：一个类可以定义多个 Signal 和多个 Slot，Signal 就好像是“事件”，而 Slot 则是响应事件的“方法”，并且和一般的成员函数没有太大的区别。如需要实现它们之间的通信时，就将某个类的 Slot 与另一个类的 Signal“连接”起来，从而实现“事件驱动”。例如，如果需要实现一个功能，当按下某个按钮，程序弹出一个对话框。做法就是：当按下一个按钮时，发出一个 Signal，而这个 signal 是事先和某个类的弹出对话框的方法（一个 Slot）相连接的。

Qt/Embedded 是 Qt 的嵌入式版本。因为 Qt 是 KDE 等项目使用的 GUI 支持库，所以有许多基于 Qt 的 X Window 程序可以非常方便地移植到 Qt/Embedded 版本上。因此，自从 Qt/Embedded 以 GPL 条款形式发布以来，就有大量的嵌入式 Linux 开发商转到了 Qt/Embedded 系统上。

但是 Qt/Embedded 的问题还是过于庞大，尽管 Qt/Embedded 声称可以裁剪到最少 630K，但这时的 Qt/Embedded 库已经基本上失去了使用价值。低的程序效率、大的资源消耗也对运行 Qt/Embedded 的硬件提出了更高的要求。

Qt/Embedded 库目前主要针对手持式信息终端。因为对硬件加速支持的匮乏，很难应用到对图形速度、功能和效率要求较高的嵌入式系统当中，比如机顶盒、游戏终端等等。

Qt/Embedded 提供的控件集风格沿用了 PC 风格，并不太适合许多手持设备的操作要求。

Qt/Embedded 的结构过于复杂，很难进行底层的扩充、定制和移植，尤其是用来实现 signal/slot 机制的 moc 文件。

2. MicroWindows/NanoX

MicroWindows 是一个开放源码的项目，目前由美国 Century Software 公司主持开发。

MicroWindows 能够在没有任何操作系统或其他图形系统支持的情况下运行，它能对裸设备进行直接操作。这样 MicroWindows 就显得十分小巧，便于移植到各种硬件和软件系统上。MicroWindows 现在除了可以运行在拥有 FrameBuffer 驱动的 32 位的 Linux 系统上，还可以运行在 SVGALib 库上，也能运行在 16 位的 LinuxELKS 和实模式的 MSDOS 上。MicroWindows 已经有了 1、2、4、8、16、32 位色彩显示驱动，而且 MicroWindows 的图形引擎使其能够运行在任何支持 readpixel、writepixel、drawhozline、drawvertline 和 setpalette 的系统上。

MicroWindows 的设计是分层的，这样的设计便于用户按自己的需要来修改、删减和增加。它分三层：最底层是 screen、mouse/touchpad 和 keyboard 驱动程序，它们直接与显示和输入硬件打交道；中间层是一个可移植的图形引擎层，它使用最底层提供的功能完成对画线、区域填充、文本、多边形、区域裁剪、色彩等的支持,最上层是 API,提供给图形化程序调用。目前这些 API 支持 Win32 和 Nano X 接口。这样一来，它们就与 Win32 和 X Window 窗口系统保持了兼容，在这些系统间移植应用软件就要容易得多。因为 WinCE API 是 Win32 API 的子集，所以 MicroWindows 也与 Win CE 在应用接口一级兼容，可在 MicroWindows 的嵌入式系统上运行 Win CE 应用。

该项目的主要特色在于提供了类似 X 的客户/服务器体系结构，并提供了相对完善的图形功能，包括一些高级的功能，比如 Alpha 混合，三维支持，TrueType 字体支持等。

3. MiniGUI

MiniGUI 是国内的一个自由软件项目,目前，MiniGUI 由北京飞漫软件公司负责开发。

MiniGUI 有两个不同架构的版本。最初的 MiniGUI 运行在 PThread 库之上，这个版本适合于功能单一的嵌入式系统，但存在系统健壮性不够的缺点。在 0.9.98 版本中，引入了 MiniGUI-Lite 版本，这个版本允许在不同的进程中创建应用程序，但同时只能有一个进程运行。下面的讨论针对其多线程的版本。

MiniGUI 有如下特点：

(1)微客户/服务器结构：因为 MiniGUI 客户/服务器体系在一个进程中实现，所以称之为微客户/服务器结构。在 MiniGUI 中，有一个线程，即服务器线程负责

维护全局的窗口列表，而其他线程不能直接修改这些全局的数据结构。而是通过请求——服务的模式来完成。例如，当一个线程要求桌面线程创建一个窗口时，该线程通过向桌面线程发送消息、然后等待桌面线程的响应，由桌面线程完成请求的任务后再通知请求线程这样一种方式来实现。

(2)多线程多窗口：MiniGUI 的窗口包括：主窗口、子窗口、对话框、控件。MiniGUI 的主窗口与附属主窗口对应于一个单独的线程，通过函数调用可建立主窗口以及对应的线程，每个线程都有一个消息队列，属于同一线程的所有主窗口从这一消息队列中获取消息并由注册的窗口过程进行处理。

(3)消息与消息循环：MiniGUI 是典型的消息驱动的系统。拥有单独线程与消息队列的窗口自创建后就处于消息循环中，读取消息队列中的消息并处理消息，直到接收到特定的消息为至。

MiniGUI 系统中的消息主要分为两类：邮寄消息和通知消息。在任何一个窗口消息队列中，邮寄消息队列的长度是固定的，这意味着在系统繁忙的时候，这类消息有丢失的可能；另外一种消息是通知消息，这类消息是通过链表进行存储的，是不允许丢失的。MiniGUI 支持多种消息的传递方式。

“PostMessage”，消息发送到消息队列后立即返回，在目的窗口的邮寄消息队列满的情况下，消息会丢失；

“PostSyncMessage”，发送同步消息：只有消息被处理后，函数才能返回；

“SendMessage”,该函数可以向任意一个窗口发送消息，消息处理完之后，函数返回。如果目标窗口所在线程和调用线程是同一个线程，该函数直接调用窗口过程，否则调用 PostSyncMessage 函数发送同步消息。

“SendNotifyMessage”,该函数向指定的窗口发送通知消息，将消息发送到对方消息队列后立即返回。

“SendAsyncMessage”,利用该函数发送的消息称为异步消息，系统直接调用目标窗口的窗口过程。

(4)图形抽象层与输入抽象层，MiniGUI 中定义了一组不依赖于任何特定硬件的抽象接口，所有顶层的图形操作和输入处理都建立在抽象接口上。而用于实现这一抽象接口的底层代码称为“图形引擎”或“输入引擎”。这种设计方式很大程度上方便了程序的移植。MiniGUI 目前包含三个图形引擎，SVGALib、LibGGI 以及直接基于 Linux FrameBuffer 的 Native Engine，利用 LibGGI 时，可在 X Window 上运行 MiniGUI 应用程序，并可非常方便地进行调试。

4. OpenGUI

OpenGUI 在 Linux 系统上存在已经很长时间了。最初的名字叫 FastGL，支持多种操作系统平台，比如 MS-DOS、QNX 和 Linux 等等，不过目前只支持 x86 硬

件平台。OpenGUI 也分为三层。最底层是由汇编编写的快速图形引擎；中间层提供了图形绘制 API，包括线条、矩形、圆弧等，并且兼容于 Borland 的 BGI API。第三层用 C++ 编写，提供了完整的 GUI 对象集。

OpenGUI 采用 LGPL 条款发布。OpenGUI 比较适合于基于 x86 平台的实时系统，可移植性稍差，目前的发展也基本停滞。

综上所述，面向嵌入式 Linux 的 GUI 系统已经发展了很长时间，有些已经是比较成熟的产品了，同时也得到了较为广泛的利用，例如 Qt/Embedded 目前已使用到 PDA 和手机产品中，Motorola 公司于 2003 年推出的手机产品中就使用了 Qt/Embedded。而 Microwindows 等轻量级的 GUI 系统在工控机、机顶盒等产品中也得以发挥其价值。

开发新的嵌入式 Linux 的 GUI 系统，首先是建立在对 GUI 系统一种新的设计思路上。系统的优劣是相对而言的，只是面向不同的应用领域其区别于其他系统的优越性才能体现出来。当然，现有的系统存在一些固有的缺陷，如 Qt/Embedded 来源于 PC 系统的 Qt，尽管经过了裁减，系统依然比较庞大，静态空间占用在 10M 以上，而动态空间占用一般在 16 或 32M 的空间上才能比较流畅运行，另外 Qt/Embedded 的运行效率不高，无法在较低端的系统上运行；而 MiniGUI 为了降低系统设计的难度，采取了一些不利于二次开发的策略，同时对应用作了诸多限制。

1.7 一个嵌入式 LinuxGUI 系统开发的实例

自 2002 年以来，我所从事的工作都与嵌入式 Linux 密切相关，包括刚开始所从事的基于 Linux 的智能手机的研发工作，到后来所从事的手持多媒体娱乐与导航系统、MP4 等产品的研发工作。这些系统都需要一个 GUI 系统的支持，而且，作为嵌入式环境中的运行与开发支持平台，要求 GUI 必须是轻量级的、高效的、稳定的，并且是可移植、可定制的、支持二次开发的 GUI 系统。

正是从那时开始，我对嵌入式 Linux 的 GUI 产生了浓厚的兴趣，并开始阅读有关材料和代码、浏览有关网站与论坛，从中寻找开发 GUI 的思路和一些关键问题的解决方法。

当时我所接触到的面向嵌入式 Linux 的 GUI 系统，主要有挪威 TrollTech 公司提供的 Qt/Embedded（准确讲 Qt/Embedded 是一个 C++类库，除此之外，TrollTech 提供的产品包括基于 Qt/Embedded 的 GUI 环境 Qtopia 及开发包）、Microwindows、MiniGUI 以及业界并不知名、由南京移软公司开发的 mGUI。

从技术层面来说，这些产品各有优缺点，也各有不同的适用环境。这也与其

面向的应用领域有密切关系。如果说开发一个新的系统能够摈弃这些产品的所有缺点而吸收这些产品的所有优点，那是不现实的。但是在了解现有系统一些实现方法的基础上，针对轻量级、多进程、多窗口的嵌入式 LinuxGUI 系统的现实需求，开发一个新的系统，至少为实现这类系统提供一个新的思路，为这一类产品提供一个新的选择。

此后，我一直利用所有可以利用的时间来设计和完善这个 GUI 系统，主要是系统的窗口管理、消息管理、进程管理。在 2004 的时候，这个系统已经可以稳定地在 PC 及一些移植的平台上（如 Intel Xscale PXA255）运行起来。基本上实现了多进程、多窗口的功能。我将这个系统命名为 LGUI。“L”是“Light”轻量级的意思。

此后，有一些同伴加入到开发队伍中，为 LGUI 移植了部分控件；也有的为 LGUI 改写了部分 GDI 函数。目前，文本方面可支持多种点阵字体；而二维图形方面的功能支持也较为全面，包括：点、线、矩形、圆、椭圆、多边形的绘制、多边形的剪切、多边形的填充等等；还有的为 LGUI 增加了图形文件的支持，主要是 JPEG 文件的支持；另外包括拼音输入法的的移植等。

2004 后半年，我们开始将 LGUI 投入商用。主要应用于车载多媒体娱乐与导航系统、手持多媒体娱乐与导航系统、家庭多媒体娱乐中心、MP4 等系统方案中，系统的运行是移定和高效的。

另外，鉴于 GUI 在应用系统中所处的特殊地位，为使 LGU 得到更多应用，我们在 2005 年已将 LGUI 在 GPL 的条款下公开源码。

1.7.1 开发 LGUI 系统主要考虑的问题

1. 多进程还是单进程

如果嵌入式 Linux 的 GUI 系统和 GUI 支持的应用都在一个进程中运行，系统实现会简单很多，因为在这种情况下不必考虑进程间的通讯问题、进程间的同步问题以及复杂的进程间窗口的剪切、输出等管理，GUI 系统在实现过程中面临的一些难题都将迎刃而解，系统的复杂性将大为降低。这种实现的好处是系统比较简单，效率高，但缺点也是明显的：首先系统的任何一个二次开发的应用系统与系统中负责协调调度的桌面系统都在一个进程中运行，任何一个程序的错误都将导致系统崩溃；其次是二次开发的问题，应用系统与桌面系统在一个进程中运行，必然使得二次开发者要对系统的机制有充分了解，这对二次开发人员提出了很高的要求。

使用多个进程，即负责其他进程启动终止、进程间消息传递、进程间协调机

制的桌面或者叫做服务器系统单独在一个进程空间中运行，而其他应用系统也单独在自己的进程空间中运行，这种组织方式在一定程度上会增加系统的开销，同时增加 GUI 系统开发的复杂度，但好处就是方便了二次开发。二次开发者不用了解桌面进程是如何工作的，他只需按照 API 文档的要求编写应用程序，而不用考虑与其他应用程序之间以及与桌面之间的通讯问题与同步问题。而且，在 GUI 系统的开发过程中，采用一定的技巧，可使得进程之间消息的传递与处理所消耗的系统资源在合理的范围之内。

另外，GUI 位于操作系统与应用程序之间，GUI 隐藏了操作系统的一些细节并向上层应用程序提供更为简捷的应用开发接口，从而使得基于 GUI 的应用开发变得更加方便，从这个角度讲，GUI 系统并不是一个单纯的应用程序，而是一个中间件。所以，一个好的嵌入式 GUI 系统应该使二次开发更为方便。

多进程的 GUI 系统首先要解决的问题是如何有效地、尽可能少地在进程间传递数据。由于不同的进程具有不同的地址空间，而且相互之间不可访问，这就使得其机制与同一个进程内部线程之间传递数据截然不同。由于 GUI 系统在进行窗口的剪切、覆盖、重绘等处理时会有大量的数据要在不同的窗口拥有者之间传递，基于进程的 GUI 系统就必须考虑进程之间数据的传递带来的系统的效率问题。

2．进程内多窗口还是进程内单一窗口

有些轻量级的 GUI 系统，其占用的系统资源很小，但本身功能也是有限的。例如，一个应用程序只能拥有一个窗口。在这种情况下，系统在设计时相对也会简单地多，设计者不必考虑主子窗口以及子窗口之间的相互关系。对于一些界面要求比较简单的 GUI 系统，例如单一界面的工控系统等，这样的选择当然也有其合理性。但对于稍复杂的系统，例如：手机、PDA 等，这样的设计是不足以满足其要求的，因而其应用价值也是有限的。

进程内多窗口需要解决的问题是父窗口与子窗口之间、子窗口与子窗口之间、窗口与其上的控件之间剪切关系的维护、消息的传递、消息的处理等问题。还有窗口的激活问题、控件的焦点问题等等。

3．Framebuffer 还是第三方图形库

首先，选择 GUI 系统基于的底层支持环境，是开发系统之前需要考虑的主要问题。如上文如述，Framebuffer 作为显示设备的一个抽象，由于接口简单、易于使用，目前正得到越来越广泛的应用，特别是在嵌入式 Linux 系统之中。而且 Framebuffer 已经在 2.2.xx 及后续的 Linux 内核版本中得到支持，所以，使用 Framebuffer 作为底层基本的图形环境，是比较明智的选择；另外， Framebuffer 作为抽象的显示设备，屏蔽了显示设备之间的差异，这使得 GUI 系统的移植变得更加容易。

但是，选择 Framebuffer 作为基本的图形环境也有一些问题，那就是基本的图形引擎就需要重新进行开发。

4. 实时性问题

Linux 本身并不是任务可抢占的实时性操作系统，但现在有一些公司通过改造 Linux，使之变成一种 RTOS，一般采取的策略是重新编写任务调度程度，将 Linux 作为实时系统的一个任务来调度。这种处理方式大大提高了 OS 的响应实时任务的速度，一般情况下可以满足一些应用系统对于实时性的要求。

GUI 系统作为操作系统上运行的一个应用级系统，由于其所处的特殊位置，对系统的实时性具有很大的影响。如何有效地组织 GUI 系统的消息传递机制，对于 GUI 系统的实时性有决定性的影响，从而对整个系统的实时性具有不可低估的影响。

1.7.2 LGUI 的特点

完成一个支持二次开发的多进程、多窗口，面向嵌入式 Linux 的 GUI 系统，工作量是很大的，另外，在窗口系统理论已非常成熟的今天，要独创一个完全新颖的窗口实现思路，也是不太可能的，如果说我们的工作有所创新的话，主要是指在系统资源受限的情况下，为以客户/服务器为基本实现模式的多窗口、多进程的嵌入式 LinuxGUI 系统提供了一个实现思路。

这里需要说明的是，在 LGUI 中，消息管理包括消息队列与消息循环的思想来源于 Windows，剪切域等内容的实现参考了其他开源项目。

其中，进程之间窗口剪切域管理的思想，主要指应用主窗口初始剪切域的计算与传递是 LGUI 的独特之处。这个思想的初衷是服务器进程只维护每个应用进程主窗体的列表，即只有应用进程主窗体的创建过程需要通知桌面进程，其他窗体的创建、销毁、显示、隐藏则不必通知桌面进程，而前提是一个应用进程中所有子窗体、对话框的边界均不得超出主窗体的显示范围（实际上子窗体、对话框、控件在显示时会被主窗体的边界剪切）。在这种情况下，每个应用进程独自维护自己所有窗体的剪切域，所以任何一个应用进程在对屏幕进行输出时不需要了解其他进程的信息，只有在当前活动的进程发生了切换，或当前活动进程的主窗体的位置或显示/隐藏的状态发生改变时，桌面进程在得到请求后才将所有进程的主窗口的初始剪切域重新计算并通过 IPC 发送到各个进程。这种方式一定程度上减少了进程之间的交互，而且进程在屏幕输出时只考虑自身的剪切域状态，所以提高了显示了速度。

1.7.3 LGUI 作为后续讲解实例

首先，我当然不是说 LGUI 是多么的好！实际上，LGUI 有一些不足之处，例如代码没有经过充分的优化，结构方面清晰度不够（如果网友能够发现更多不足之处，也正好作为嵌入式 LinuxGUI 系统的反面教材！）。我这里也不是要推介 LGUI 给大家用，这对我没有意义。使用 LGUI 作为讲解实例，使得阅读者能够比照这本书中所讲的理论性的内容与实际代码，相信对于 GUI 的开发会有很多帮助。

第2章 Linux 高级程序设计简介

2.1 Linux IPC 介绍

在 LGUI 中，其客户/服务器的模式首先是基于 Linux 的 IPC 来实现的。

2.1.1 信号

信号是为了使进程获得某项重要的通知而发送给它的重要事件，这时进程必须停止当前的工作，转而处理该信号，每一个信号都用一个整数代表信号的类型，这些信号定义在/usr/include/asm/signal.h 中。我们日常使用 Linux 的过程中经常接触到信号操作，比如当某个程序正在运行时，为了终止程序的运行按下 Ctrl-C 键，或者使用 kill 命令将进程杀掉，实际上就是向进程发送了信号。

进程在接收到信号后，处理的方式有五种：一种为忽略这个信号；一种为执行处理该信号的函数；还有的是暂停进程的执行；也有的是重新启动刚才被暂停的进程；最常见的是采用系统默认的行动，大部分信号的默认操作都是终止进程的执行。

在 LGUI 中，使用信号主要是为了结束进程，即当桌面进程退出时，桌面进程要通过 Kill 函数发送 SIGTERM 信号到所有的客户进程，以结束当前正在运行的进程。在此之前，客户进程启动时，首先要注册一个 SIGTERM 信号的处理函数，当客户进程接收到这个信号后，就调用注册的函数作一些退出前的工作。

2.1.2 管道

管道是父进程与子进程之间单向的通信机制，即一个进程发送数据到管道，另外一个进程从管道中读取数据，如果需要双向的通信机制，则需要建立两个管道。

上面所说的是匿名管道，它们是与进程密切相关的，只有有关系的进程才能使用它们，如子进程和父进程之间或者是多个子进程之间。如果不相关的进程需要通信，则可以通过使用有名字的管道，即命名管道。命名管道又称 FIFO(First

In First Out)，它是文件系统中的特殊文件。不同的进程通过打开命名管道读写也可以实现类似匿名管道的数据通信，匿名管道的读写也是阻塞的，比如当某个进程读数据时，如果没有数据存在，那么读数据的进程被阻塞，直到有一个进程向命名管道中写入数据。

2.1.3 消息队列

使用管道传输数据的缺点是显而易见的。特别是管道中的数据必须是 FIFO 次序的，进程必须读完所有的数据，才能找到所需要的数据段，除了管道之外，Linux 上还有许多进程间的通信方式，比如：消息队列、信号量、共享内存等。

消息队列（Message Queue）是存放消息的队列。消息是指含有消息类型（是一个数字）和消息数据的信息。它可以是私有的，也可以是公有的。如果它是私有的，它只能被创建消息队列的进程和它的子进程访问到。如果它是公有的，它可以被系统上任何一个进程访问到。不同的进程可以向同一个消息队列中写消息或者读消息，消息可以按类型访问，因此不必要使用 FIFO 的次序。

2.1.4 信号量

在多进程程序中，多进程的同步是一个比较麻烦的问题。尽管我们可以使用管道、消息队列可以作到同步，但是我们有时需要同步多个进程，甚至是多个进程对数据资源访问的同步，信号量（Semaphore）便是解决这类问题的进程间的通信方法。

信号量是一个含有整数的资源，它允许进程通过检测和设置它的值来实现同步。即进程在检测和设置它的值时，保证了其他进程在此期间不能做类似操作。

2.1.5 共享内存

上面所讲的进程间的通信工具，不论是管道、消息队列还是信号量，它们都使用了顺序共享数据的通信方法。如果我们可以自由地访问和共享某些资源，则上述方法都不能满足要求。共享内存（Share Memory）正好满足了这方面的需要。

在 Linux 下每一个进程都拥有自己的内存空间，如果一个进程进入另一个进程的内存空间，则很容易引起错误。共享内存则是开辟了一段内存空间，它允许多个进程同时拥有这个空间。即许多进程都可以对这一内存空间进行读写操作。

在 LGUI 中，桌面进程启动以后，首先会创建一个共享内存，并将字库、系统预定义 GDI 对象、鼠标的当前状态存放到共享内存中，以方便与其他客户进程访问。

2.1.6 Domain Socket

Linux 下有 Domain Socket 和 Berkely Socket 两种套接字，其中 Domain Socket 是进程间的通信方式之一，而 Berkely Socket 则支持 Unix、Windows、OS/2、Macintosh 以及其他的计算机系统。

Domain Socket 与 Berkely Socket 的接口是非常类似的，只是它们使用的协议族不同。LGUI 正是使用 Domain Socket 实现桌面进程与应用进程之间的通信。

2.1.7 SYSTEM V IPC 与 POSIX IPC 的区别

上面提到的除了管道与信号之外，都是 SYSTEM V IPC 中的内容。在早期的 Linux 内核中，SYSTEM V IPC 的实现是很不完善的。在后续的版本中，Linux 开始支持 POSIX IPC，与 SYSTEM V IPC 相比，POSIX IPC 实现接口更为简单，也更为方便。比如经常用于实现消费者/生产者问题的信号量，SYSTEM V IPC 与 POSIX IPC 提供的接口就完全不同，使用 POSIX IPC 非常简单，而 SYSTEM V IPC 则比较复杂。

2.2 Linux 多线程编程介绍

LGUI 是一个消息驱动的软件系统，线程在其中扮演着非常重要的角色。例如：桌面进程对于鼠标事件、键盘事件都是通过单独的线程进行监视的。

多线程指的是一个独立的程序看起来像是同一时间执行多个任务的功能。这里，“任务”指一个计算单元，对应于一个线程。例如，一个程序在加载数据文件的同时读入用户输入就是进行两个计算单元，就可以用多线程的程序来实现。

一个进程中执行的所有线程称为线程组。它们共享同一块内存区域，所以可以访问同样一些全局变量、堆内存及文件描述符等等。

使用线程组与使用一个顺序执行的程序的优势在于：很多操作可以并行执行，所以事件在它们到达后立即得到处理。在单处理器的机器上，线程的并行执行就是在不同的时间片执行不同的线程；而在多处理器机器上，同一进程的不同线程

可以分配到不同的处理器上，真正实现并行执行，所以多线程程序可以充分利用机器上的所有处理器，其执行速度比单线程要快。

使用线程组与使用进程组的优势在于：线程间的运行环境切换比进程间的运行切换要快得多。同样，线程间的通信比进程间的通信更快更容易。但另一方面，由于线程组中的所有线程使用同一块内存空间，如果一个线程破坏了内存中的内容，其他线程也会面对同样的后果，对于进程，操作系统通常会保护每个进程使用的内存空间，一个进程破坏了它自己的内存空间中的内容，并不会殃及其他进程。

LGUI 中大量地使用了线程技术来提高系统性能，同时使得其能快速响应外部输入、进程间消息等。

（本章详细内容还在撰写之中……）

第3章 LGUI 的基本体系结构

3.1 基础知识

3.1.1 嵌入式 Linux 的 GUI 到底有什么用？

我们在前面讲了 GUI 在嵌入式 Linux 系统中所处的位置，讲了现在常用的一些 GUI 系统及其特点，但是初入门的开发者对 GUI 到底是什么东西没有直观的认识，通常情况下，他们问到的第一个问题就是：GUI 到底有什么用？

我们知道：Windows 启动以后会有一个桌面环境，上面会有工具栏，桌面上会有一些与应用程序相关的图标，我们点击其中一个图标，就会启动一个对应的程序，这个程序启动后，通常会出现对应的窗口，在窗口中可以显示程序运行的状态，程序的输出等等。

Linux 启动以后，会进入一个命令行状态，我们通过输入命令调用系统中提供的程序或我们编写的程序来完成某种功能。而使用 GUI 的目的，就是使得嵌入式 Linux 系统启动以后，进入一个类似的桌面环境，我们在这个易于操作的环境中与系统进行交互。例如当手机启动后，我们希望看到的是一个易于操作的图形界面，而不是一个需要高超的专业水平才能操作的命令行界面。

更为关键的是，GUI 系统应该提供一个二次开发的模式、支持二次开发的 API 函数集，使得基于此可以方便地开发应用程序，而不必对 GUI 整体的体系结构了如指掌。就像 Windows API 编程一样，可能很多 Windows API 程序的开发人员并不了解 Window 内部窗口之间是如何管理的，但这并不影响他使用 Windows API 编出漂亮的图形界面。与此类似，我们一般在嵌入式 LinuxGUI 中，也提供二次开发的 API 函数集以及二次开发的模式。开发人员在不了解 GUI 体系结构，内核部分功能以及处理方法的情况下，根据 GUI 的 API 函数接口以及二次开发模式，完全可以开发出应用程序。

在这样一个前提下，如果要构造嵌入式图形界面，可以先构造自己的 GUI 环境，包括一个小型的可定制风格的桌面环境，然后提供支持二次开发的开发工具——API 函数集与应用程序的模式，应用层的开发人员就可以轻松进行上层软件的开发了。这就是构造自己的 GUI 环境的最终目的——实现快速的、统一结构的应用程序开发模式，加速嵌入式 Linux 项目的实施。相反情况下，GUI 层没有统

一的接口与标准，要么对于应用开发人员就有很高的要求，他必须非常了解系统的整体结构，否则无法开发出应用层的软件，或者时刻关注于用户界面的内容，而无法将更多的精力集中于程序逻辑，导致项目无法顺利进行。同时还会面临一个问题，那就是所有应用层的程序其结构百花齐放，为后续的集成、维护带来困难。

所以，此后的章节就是在你了解了 Linux 高级编程（主要指多进程、多线程编程）的前提下，告诉你如何构造属于自己的面向嵌入式 Linux 的 GUI 环境，相信会给你的工作带来很大帮助。而在描述过程中，就以 LGUI 作为例子，这样您在阅读的过程中，可以随时浏览相关代码，以对整个系统有更深入了解。

3.1.2 LGUI 的基本体系结构是什么？

总体而言，LGUI 是一个多进程、多线程支持的客户/服务器系统。

其中多进程是指，当 LGUI 启动后，所显示的特征是一个桌面环境显示在屏幕上（如 Windows 一样），实际上是启动了一个服务器端进程。为什么讲是服务器端进程呢？因为从这个进程在整个系统中所担负的角色来看，它是为其他应用程序所对应的进程服务的。相应的，其他应用程序所对应的进程即可以称之为客户进程。LGUI 的特点是多个客户进程可以同时存在，系统自动维护各个进程之间的窗口关系，所以这就构成一种多对一的客户机/服务器模式。在后面的讲述中，我将服务器进程称为桌面进程，将应用程序对应的进程称为应用进程。

在嵌入式系统中，如何保证系统在资源受限的情况下稳定高效运行是至关重要的，客户/服务器模式是系统的总体模式，而客户端向服务器端发送什么，服务器端如何处理，服务器端向客户端响应什么，这些设计细节对系统性能有决定性的影响。

因为任何一个客户进程并不知道当前其他进程的信息，所以他在进行输出时便无法兼顾。在这种情况下直接对屏幕进行输出，势必会影响或破坏其他窗口的框架。但是另一方面，如果所有客户进程对于屏幕的输出全部发送到服务器进程，系统的效率势必会大打折扣。

LGUI 在这方面作了一些努力，想了一些办法：客户进程并不是将输出请求发送到服务器端，由服务器端完成输出，而只将必要的信息发送到服务器端，而实际对屏幕的输出都由自己维护，这在很大程度上减少了进程之间数据的传递，减少了系统的资源消耗，从而在很大程序上提高了系统的性能。

3.1.3 为什么是客户机/服务器结构？

为什么是客户机/服务器结构呢？为什么不让所有进程都互相平等呢？而要分出个服务器进程与客户机进程？根本上讲，这是为了维护窗口剪切的需要。

假如两个程序同时启动了，那么就是两个不同的进程，这两个进程分别要做不同的事情，分别要在屏幕上进行输出。我们知道，两个进程可能是完全不同的应用程序，相互之间不可能事先“通个气”，告诉对方“我”在屏幕的什么位置输出信息。这必然会导致两个程序互相竞争屏幕资源，看到的结果会是：屏幕上一些乱七八糟的内容，我们无法了解两个进程分别要“说什么”。这个情形可以用开会来比喻：比如我们召集一个技术部的技术讨论会，一上来大家就各说各话，每个人都认为自己说的没有问题，但一个旁观者根本听不到任何有用的信息。这时候就需要一个“技术部经理”出来说话，他来协调每个人发言的时间，以便每个人表达的信息都能为别人所了解。那么这个协调与被协调的关系算不算是一个客户机/服务器结构呢？一般意义上讲应该说不是，因为所谓客户机/服务器结构应该是：客户机发出请求，服务器进行处理，并将处理的结果返回到客户机。技术部开会的时候并不是每个工程师发请求到技术部经理，由技术部经理完成处理后返回信息到工程师。在这个系统中，技术部经理只是一个协调者的角色，而不是服务者的角度，所以并不是通常意义上讲的客户机/服务器结构。但是另一方面，客户端有胖瘦之分，客户端要求服务器端处理的事情可能很复杂，也可能很简单。在很复杂的情况下，客户端很少自己做事情，大部分事情都由服务器端完成；相反，客户端可能要求服务器做很少的事情，大部分事情由自己完成。无论何种情况，他们之间有一个请求与被请求的关系，协调与被协调的关系。所以，在这里我们不必过多讨论这是不是严格意义上的客户机/服务器结构，我们姑且认为协调者的角色就是服务器的角度，被协调者的角度就是客户机的角色。

在多个进程同时运行的情况下，任何一个进程在对屏幕进行输出的时候需要了解当前屏幕上的哪些区域是可以输出的，哪些区域是不可以输出的， 从具体实现的时候，有两种方法：一是所有的输出都由一个服务进程来完成，由这个服务进程来确定当前对于哪些屏幕区域的输出请求是允许的，哪些是不允许的，这样就避免了多个进程对于屏幕区域的竞争；另一种方法就是其他进程只从服务进程那里请求并得到允许输出的区域，而具体的输出操作自己完成。前一种方法面临问题就是需要在进程之间不停地传递大量数据。我们知道，不同进程之间除非通过 IPC，否则因为不同的进程空间不允许互相访问数据，大块的数据需要在进程之间传递，这是非常耗费资源的操作，在嵌入式环境中更是不可取。而后一种方

法需要输出的进程只请求允许输出的屏幕区域，输出的操作由进程自己完成，相对而言，效率会有很大提高。而 LGUI 就是采取了这种方式。

3.1.4 为什么要多进程？

从 GUI 的角度讲多进程，实际上是讲多个进程对于屏幕的输出管理，如果有很多进程在同时运行，但并没有屏幕输出的要求，这就谈不上多进程的管理。

我们说 LGUI 是一个多进程、多线程支持的客户机/服务器系统。那为什么是多进程呢？单个进程不是更简单吗？

当然，并不是所有的嵌入式环境都要求多个进程周时运行，或者同时要求进行屏幕输出。例如一个机顶盒的 GUI 系统，就不会这样复杂。但在一些复杂的嵌入式环境中，多进程是必须的，例如 PDA 等。我们不能要求用户在 PDA 中添加一项功能，就重新将系统编译一下吧！

LGUI 的体系结构是支持多进程的，但如果你只想构造一个简单的单进程的系统，那么你在了解了这些相关的内容后，自己可以剪裁 LGUI 的功能，从而构造一个自己的 GUI 系统。而了解多进程的体系结构，对于构造自己的系统是有帮助的。

3.1.5 为什么要多线程？

从前面的内容我们可以看到，线程是一种轻量级的进程，是进程内的进程。即一个进程内可以有多个线程同时运行，线程之间可以共享数据。在多 CPU 的系统中，多个线程可能分别在不同的 CPU 上运行；在单 CPU 系统中，多个线程会分时占用 CPU 时间片。

将程序设计成多线程的系统，有两个方面原因，一是多线程可以更大程度上发挥 CPU 的效率，所以程序运行更快；另一方面是程序逻辑的需要，有时复杂的系统可能需要多个程序逻辑并行，并要求实现同步。例如在 LGUI 中，一个应用程序启动后每个显示的窗口都会对应一个线程，这个线程的主要作用就是进行消息队列的循环处理，即查看消息队列中是否有需要处理的消息，如果没有消息，就会进入睡眠状态。但这个线程的阻塞并不影响其他窗口接收、处理消息，而且阻塞的线程在收到消息后会自动激活并处理消息。如果将系统设计成单一线程，那么一个耗时的操作就会阻塞其他所有操作。所以说除了效率的问题，单一线程对于某些特定的程序逻辑也是无法支持的。

3.2 LGUI 体系结构综述

3.2.1 客户机与服务器之间的通讯通道

首先，客户与服务器之间的连接是通过 Domain Socket 来实现的。Domain Socket 是 Linux/Unix 系统中进程之间通讯的一种手段，Domain Socket 的接口与 TCP/IP socket 通讯的接口是非常类似的。

在服务器端，针对每一个客户端的连接请求都会单独创建一个 Socket 并通过这个 Socket 与客户端单独进行通讯，也就是说，服务器与每个客户端都是使用单独的“通道”传递数据，从而使得任何一个客户端都不会“干扰”其他客户端与服务器端的交互。

如图 3-1 所示：

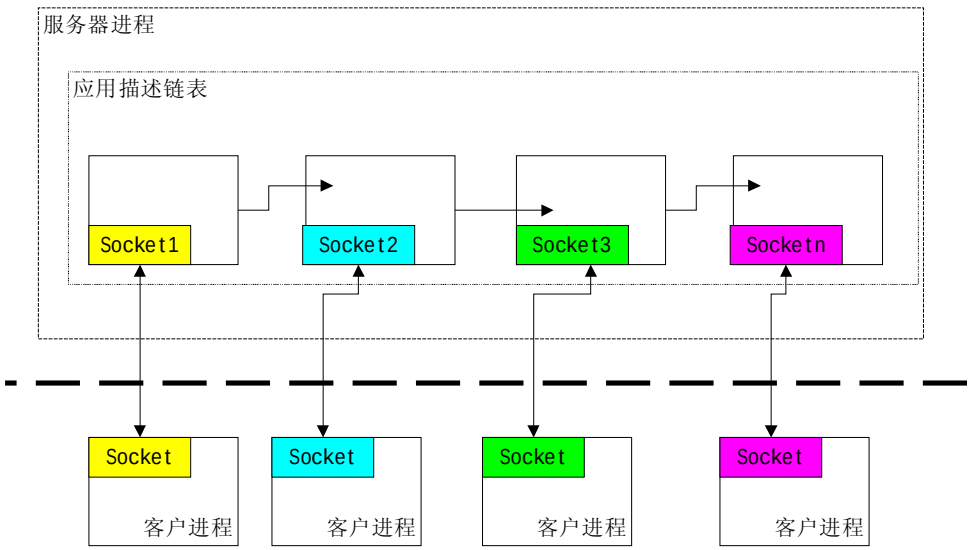

图 3-1 LGUI 中客户/服务器 Domain Socket 连接示意图

这种结构使得系统稳定性更好。首先，服务器与客户在不同的进程空间中运行，由于进程具有独立的地址空间，客户进程出现问题不会导致服务器进程的崩溃；另外，任何一个客户进程出现问题也不会波及其他进程；最后，不论客户进

程正常关闭或不正常关闭，服务器进程总会收到一个关闭 Socket 的消息，于是服务器进程可以在这个消息处理过程中将对应的客户进程占有的资源进行清理。

从图中还可以看到，在服务器端，保存有一个应用描述的链表。当前系统中运行的所有应用进程，在服务器端（桌面进程）会有一个对应的描述节点，多个节点构成一个链表。每个描述节点中包括有应用进程的名称、应用进程的进程 ID，应用进程主窗口的边界矩形（这个数据是非常重要的，是桌面进程维护多进程窗口输出的根本依据，在后面的章节中会详细讨论这个问题）。

那么这个链表是如何形成的呢？

桌面进程（服务器进程）启动后，首先会启动一个线程专门用于侦听应用进程（客户进程）的连接请求，而每个应用进程启动后的第一件事也是首先向桌面进程发出连接请求，只有连接到桌面进程，应用进程的启动才能继续下去。当桌面进程侦听到应用进程的一个连接请求后，就会接收这个请求并建立一个新的 Domain Socket 连接，然后创建一个新的线程来读取发送到这个 Socket 端口的数据。这样做的目的就是防止某一个应用进程的崩溃波及到其他进程。例如：当某一个应用进程出错退出后，桌面进程会收到一个 Socket 的关闭或出错消息，在对这个消息的处理过程中，桌面进程就可以关闭对应的 Socket 并销毁线程、刷新该进程主窗口占用的屏幕并释放进程描述信息等资源。

当客户端进程启动以后，也会创建一个单独的线程循环读取桌面发送到 Socket 端口的数据，并将读到的消息发送到主窗口的消息队列中去。

那为什么要建立这么多的线程呢？建立线程当然也会消耗系统资源，但应该注意的是：对于 socket 的读是一种阻塞读，即在没有数据到达的情况下，线程会被阻塞在这个地方，只有当数据到达后，线程才会被唤醒。这样的话，线程消耗的资源便是有限的。相反情况，如果不建立这些线程，对于客户端来说，对于消息的响应可能会变得比较迟缓；而对于桌面进程来讲，除了对于消息的响应会出现与客户端的同样情形以外，客户端的错误很可能会波及到桌面进程，导致桌面进程不再能响应其他进程的消息。

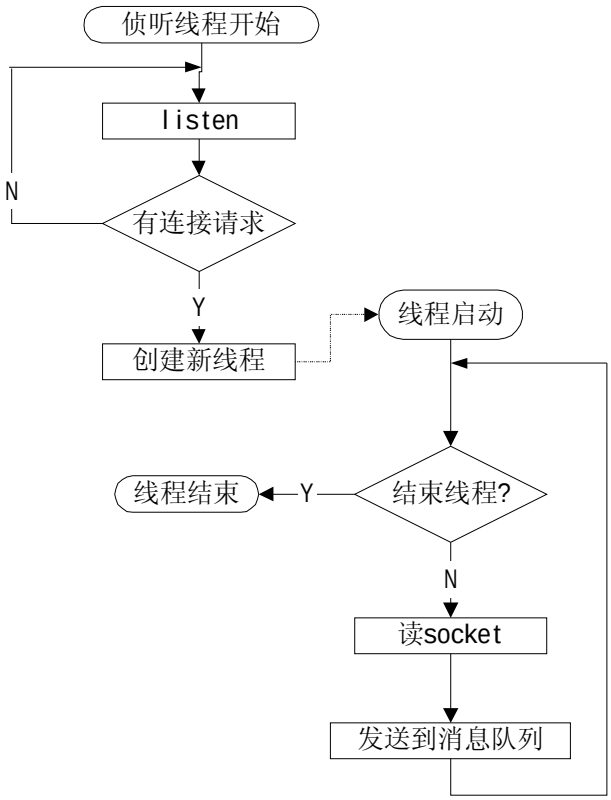

图 3-2 LGUI 多线程 Domain Socket 连接过程示意图

3.2.2 客户机需要与服务器交换什么信息？

前面我们提到，客户机/服务器模式是系统的总体模式，但客户机向服务器提出什么请求，服务器如何响应客户机的请求，服务器如何把请求的结果回传给客户机，也就是说：客户机与服务器之间交换什么信息，这是系统总体模式的核心问题。

要回答这个问题，首先要有“剪切域”的概念，这个概念我在后面的章节里会详细讲述。在这里，为了说明客户机/服务器之间交换信息的内容，我必须要对

“剪切域”这个概念提前介绍一下。

假设当前有两个窗口，其中 A 窗口在下面，B 窗口在上面，所谓在上面是指离观察者更近，如果两个窗口有重叠，则在上面的窗口会覆盖在下面窗口的内容。我们一般用“Z 序”来表示窗口的上下关系，如果 Z 序值小，则在下面，反之，则在上面。如下图所示：

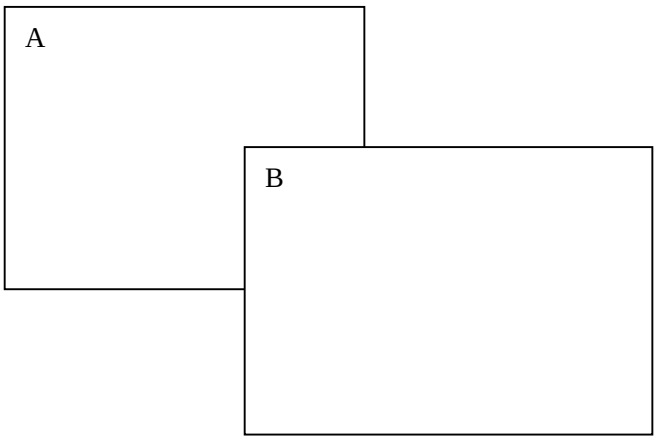

图 3-3 窗口的 Z 序及剪切关系

因为 A 窗口在下面，那么这个窗口区域里被 B 窗口所覆盖的部分 A 窗口就不能进行输出，否则就会破坏 B 窗口中的内容。那么 A 窗口中那些区域是可以输出的呢？这就是剪切域的概念：一个窗口中可以输出的有效区域总和称为这个窗口的剪切域。

从图 3-3 中可以看出，A 窗口被 B 窗口剪切以后，其有效区域是一个多边形，在 LGUI 中，这个多边形是通过多个矩形组成的链表来表示的。如下图所示，A 窗口的剪切域分成了两个矩形：

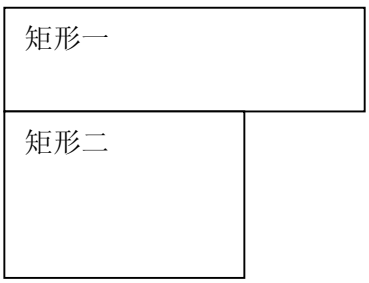

图 3-4 窗口剪切域多边形由多个矩形组成

也就是说，对于 A 窗口来说，这两个矩形范围是可以输出的有效范围。在 LGUI 里，一个窗口的剪切域正是用矩形链表来表示的。

回到这一小节的题目：客户机需要与服务器交换什么信息，答案就是剪切域！那么窗口的剪切域，是如何交换的，是在什么时机交换的，这是我们马上要问到的又一个问题？

我先把答案告诉你，然后再详细讨论——交换的剪切域为初始剪切域，交换的时机是每一个应用进程初始启动的时候。在这里我们暂时先不展开讨论剪切域的问题，在后续的章节中会详细讨论这个问题。

任何一个应用程序启动时，会将自己主窗口的边界矩形告诉桌面进程，桌面进程根据当前所有进程的情况计算一个初始的剪切域并发送到应用进程中；如果一个程序启动时其主窗口与当前已经运行的进程主窗口有交叉，则需要对这些进程发送初始剪切域变化消息，各进程在接到这个消息后重新计算其内部窗口的剪切域；同样，如果不同进程主窗口在屏幕上的 Z 序发生了变化，桌面进程也要计算受到影响的进程的初始剪切域，并将计算结果发送到相应进程中去。

当各个进程收到其有效的初始剪切域后，各进程就知道这个剪切域范围就是它可以进行输出的区域，除此之外的区域就是不允许输出的区域。各进程根据这个要求进行输出，就不会破坏屏幕上其他进程的输出，大家都便可确保相安无事，相互之间不会发生冲突。

我们前面讨论过多进程的效率问题，那么如果所有进程只是在第一次启动时与桌面进程进行一次交互，并在其他进程启动的时候被动地更新一次自身的剪切域。此后的操作都与桌面进程无关，与其他进程也没有关系。那么可以说，进程之间交互的数据是非常有限的，在这种情况下，效率就会有大幅度的提升。LGUI 正是采取了这种措施，使得其效率在支持多进程的情况下依然能得到保证。

在这里有一个潜在的限制条件，即应用进程启动时传递到桌面进程的主窗口边界矩形必须是这个进程中所有窗口边界矩形的超集，即进程内其他所有窗口边界矩形都完全包含在主窗口边界矩形之内，这样才能保证初始剪切域对于每个进程有效的。我认为，在嵌入式环境中，这种限制是可以忍受的，以这种限制换来性能的提升是值得的。

3.2.3 服务器对客户进程的管理

服务器进程用一个链表来表示当前与其连接的所有客户进程的信息，链表中的一个节点表示一个当前与服务器相连的客户进程。其结构如图 3-2 所示：

```
char        pAppName[256];  //应用名称
RECT        rc;             //应用主窗口位置
PClipRegion pClipRgn;       //主窗口剪切域
BOOL        bVisible;       //是否可视
int         fdSocket;       //Socket 描述符
pthread_t   tdSocket;       //socket 线程 ID
```

图 3-5　服务器端客户进程管理链表节点结构

客户端与服务器端之间发送的消息及发送消息的时机包括以下几个方面：

客户端创建主窗口时，向服务器端发送消息：LMSG_IPC_CREATEAPP。

客户端进程启动时，首先要与服务器端建立 Domain Socket 连接，连接建立以后，就会发送创建应用消息到服务器端。该消息的附加数据为客户进程的描述，主要包括两项内容：一是客户进程的进程号(PID)，二是主窗口的矩形边界。

服务器端保存有客户端信息的一个列表，收到客户端的创建消息后，将消息进行解析，然后在列表中增加一个节点。

客户端显示主窗口时，向服务器端发送消息 LMSG_IPC_SHOWMAINWIN。

客户端程序要求按 win32 的程序格式书写，所以一般的格式为：

```
CreateWindow(……);
ShowWindow(……);
UpdateWindow(……);
While(1){
        //消息循环体，取消息，分发消息
}
```

所以客户端发送的第二个消息就是显示窗口。但显示窗口的功能并不是由服务器进程来完成，因为显示窗口的过程中包括窗口客户区的绘制代码，所以服务器进程并知道应该怎样完成客户端主窗口的显示。客户端发送显示窗口消息到服务器进程的目的只是为了通知服务器：当前主窗口对应的区域将成为这个客户进程的显示区域，以便服务器进程再通知其他的应用进程应该怎么改变剪切域。

服务器端在得到客户端要求输出主窗口的请求后，首先根据得到的“需要输

出的窗口区域”，生成一个初始剪切域，并将这个初始剪切域作为响应消息的附加数据，传送到客户端。客户端在得到这个响应消息后，会复制这个初始的剪切域，并进行窗口的输出。

新建立的应用程序，其主窗口总在最上面，那为什么还会有初始的剪切域需要进行计算呢？如果在最上面，那初始的剪切域不就是窗口的外部框架吗？原因是这样的：因为桌面上的系统状态栏、软键盘、开始菜单、输入法窗口等永远都在所有应用窗口的最上面，所以无论应用窗口的 Z 序大小如何，首先必须接受这几个窗口的剪切，否则，应用窗口的输出将会破坏桌面上状态栏等等窗口的框架。

这里有一个潜在的问题，即服务器端在回复客户端同意其输出主窗口并将初始剪切域发送到客户端之前，将会对当前启动的所有的客户进程重新计算剪切域，并将新的剪切域发送到对应的客户进程中去，但是，为了避免系统过于复杂，服务器端并没有等待其他客户进程发回确认消息后再给准备输出主窗口的新启动进程发送确认消息，由于进程执行的不确定性，这有可能使得新窗口的输出，在其他客户进程因没有来得及改变剪切域而碰巧有输出的情况下被破坏。

这个问题其实解决起来没有太大的难度，进程之间的互斥量就可以解决这个问题，实际上一个客户进程准备输出主窗口时发送请求到服务器端后，就会锁定一个互斥量，而由服务器通过解锁互斥量来激活等待的客户进程。但是通过这种方式来解决上面提到的“潜在的问题”，则有可能引发另外一个“潜在的问题”，即如果服务器等待其他客户程序发回确认消息后再将确认消息发送到请求输出主窗口的客户程序时，而恰巧某一个客户程序崩溃了，则系统将会一直处于挂起状态，而不能继续运行了。权衡利弊，因为“由于其他客户进程因为没来得及改变剪切域而碰巧输出的情况”，只有该客户进程窗口是以动态输出为主要任务的情况下才有可能出现，所以还是采用了实际上更为安全的这种“鸵鸟算法”。

上述的这个过程可用下图表示：

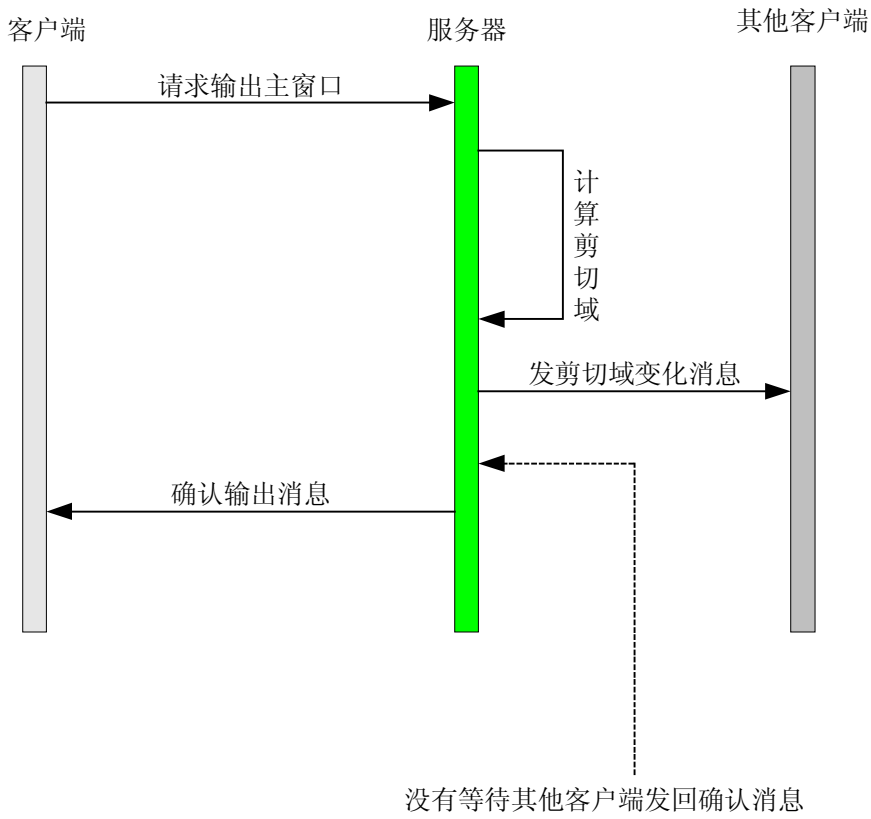

图 3-6　客户端输出主窗口的过程

客户端隐藏主窗口时，发送消息 LMSG_IPC_HIDEMAINWIN

当客户端隐藏主窗口的时候，客户端所有窗口都将被隐藏，所以客户端必须通知服务器进程，使得服务器进程重新计算自身的剪切域，如果有其他客户进程存在的话，要计算其他客户进程的初始剪切域，并将其他进程的初始剪切域发送到相应的进程中去。然后，服务器进程的窗口自身也要进行重绘，同时需要向其他进程发送重绘消息。

隐藏主窗口主要用于将应用最小化的时候。

客户端销毁主窗口的时候（LMSG_IPC_DESTROYMAINWIN）

当一个应用结束的时候，就会发送这个消息到服务器进程，除了要将应用链表中客户进程对应的节点进行销毁以外，此时剪切域方面的处理过程类似于客户端主窗口最小化时的处理过程

3.3 LGUI 进程创建与进程的管理

在 LGUI 里，所有的应用进程都是由桌面进程创建的，而创建进程的方法就是我们所熟知的 fork()函数。

以下是 LGUI 启动一个新程序时所使用的代码：

```
BOOL
LaunchApp(
        char* pFileName
)
{
        pid_t child;
        char* args[]={NULL};
        if((child=fork())==-1){
                printerror("fork error");
                exit(EXIT_FAILURE);
        }
        else if(child==0){
                execve(pFileName,args,environ);
        }

        DisactiveWindow(_lGUI_pWindowsTree);
        return true;
}
```

图 3-7 桌面进程启动一个新的应用进程

从代码中可以看出，桌面进程首先通过 fork()创建一个新的进程，然后调用 execve 启动文件名为 pFileName 的程序。这是启动程序常用的办法。其中 environ 是为了使新的进程能够继承父进程的环境变量。

应用描述链表的结构如下所示：

```
typedef struct taglGUIAppStat{
    char            pAppName[256];   //应用程序名称
    RECT            rc;              //主窗口外接矩形
    PClipRegion     pClipRgn;        //初始剪切域
    BOOL            bVisible;        //是否可视
    int             fdSocket;        //domain socket 句柄
    pthread_t       tdSocket;        //socket 线程句柄
    struct taglGUIAppStat* pNext;    //指向下一节点的指针
} lGUIAppStat;
typedef lGUIAppStat* PlGUIAppStat;
```

图 3-8 应用描述链表结构

每当桌面进程得到一个连接请求后，就会创建一个节点，加入到这个链表中。通过遍历这个链表就可以获取当前应用进程的所有信息。

第4章 LGUI 中多窗口的设计与实现

4.1 窗口树

有些嵌入式设备对于界面需求比较简单，也许有一个窗口就足以解决问题。但大多数情况下，要求有多级窗口。在 LGUI 中，一个应用进程中包括有三级窗口，即主窗口、子窗口、控件。其中控件具有大部分窗口的特性，所以也统称为窗口。

这三级窗口形成一个树状关系，就是这里要讨论的窗口树。

窗口树的结构如图 4-1 所示：

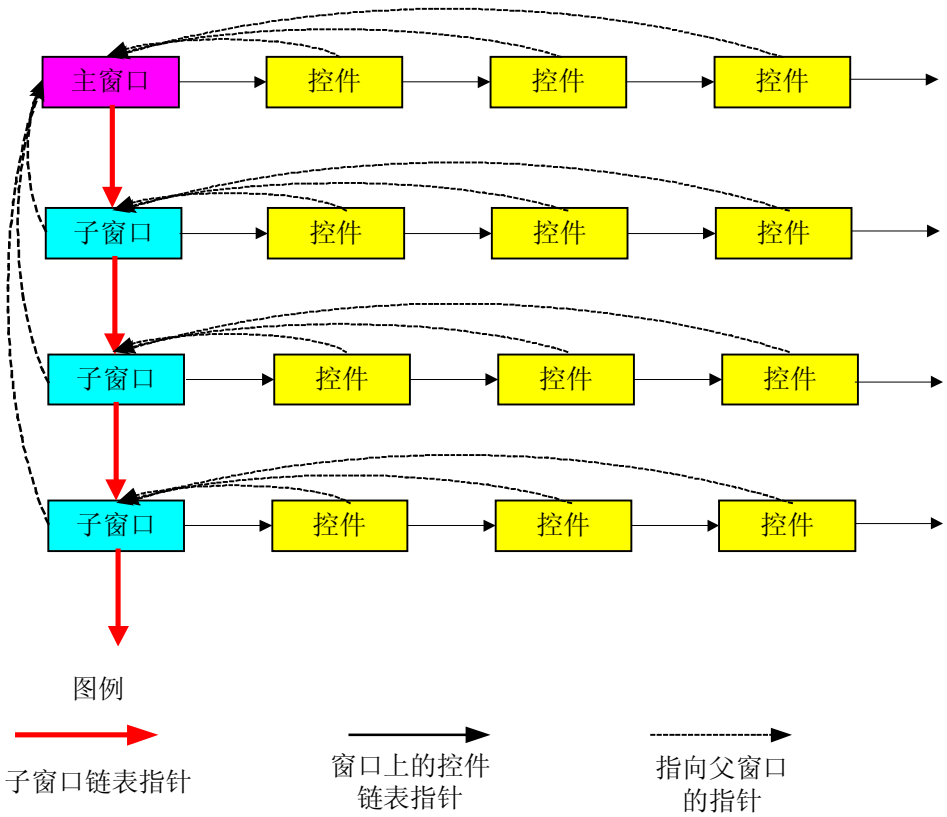

图 4-1 LGUI 中的窗口树

从图 4-1 中可以看出，主窗口可能会含有多个子窗口，同时也可能会包含有

多个控件；而每个子窗口也可能会包含有多个控件。

同一父窗口的控件或子窗口是用链表连接在一起的，同时，除主窗口以外，所有子窗口/控件都有指向父窗口的指针，这样非常方便窗口之间的互相查找。

从图中也可以看出，为了使系统简化，控件之间没有嵌套关系。

```
typedef struct tagWindowsTree{
    char                lpszClassName[MAXLEN_CLASSNAME];
                        //窗口类名
    char                lpszCaption[MAXLEN_WINCAPTION];
                        //窗口标题
    DWORD               dwData;         //窗口附加数据
    DWORD               dwAddData;      //窗口附加数据
    RECT                rect;           //窗口外接矩形
    HMENU               hMenu;          //子窗口标识
    DWORD               dwStyle;        //窗口样式
    int                 iZOrder;        //窗口 Z 序
    PMsgQueue           pMsgQueue;      //消息队列
    PClipRegion         pClipRgn;       //剪切域
    PClipRegion         pBackClipRgn;   //备份剪切域
    PInvalidRegion      pInvRgn;        //无效域
    pthread_t           threadid;       //线程 ID
    LPSCROLLINFO        pHScroll;       //横向滚动条
    LPSCROLLINFO        pVScroll;       //竖向滚动条
    LPSCROLLCURSTATE    pHCurState;     //横向滚动条状态
    LPSCROLLCURSTATE    pVCurState;     //竖向滚动条状态
    PCARETINFO          pCaretInfo;     //光标信息
    struct tagWindowsTree* pFocus;      //当前焦点控件
    struct tagWindowsTree* pParent;     //父窗口
    struct tagWindowsTree* pControlHead;//窗口上控件头指针
    struct tagWindowsTree* pControlTail;//窗口上控件尾指针
    struct tagWindowsTree* pChildHead;  //子窗口头指针
    struct tagWindowsTree* pChildTail;  //子窗口尾指针
    struct tagWindowsTree* pNext;       //兄弟窗口中下一个窗口
    struct tagWindowsTree* pPrev;       //兄弟窗口中前一个窗口
} WindowsTree;
typedef WindowsTree* PWindowsTree;
```

图 4-2 LGUI 中的窗口结构

从图 4-2 中可以看出，窗口树是 LGUI 中非常重要的结构，其中的窗口节点基本涵盖了窗口的所有信息。在任何时候，只要能获取到窗口节点的指针，即可以得到该窗口的详细的信息。在 LGUI 中，窗口的句柄就是窗口节点的指针，所以，给定了窗口句柄，便可以操纵窗口。

从图 4-1 与图 4-2 的数据结构描述中还可以看出。主窗口上的子窗口是以儿子/兄弟的方式链接，即主窗口有一个指针指向它的第一个子窗口，这个子窗口再通过兄弟链表将其他子窗口链接起来。主窗口与子窗口上的控件也是同样的链接方式。

另外，除主窗口之外，所有的子窗口与控件都有一个指针指向其父窗口，通过这个指针，可以非常方便地得到父窗口的信息，例如某一个窗口上有一个按钮，按下这个按钮后，需要向这个按钮所在的窗口发送消息，有了父窗口指针，就很容易解决问题。

通过浏览源代码可以看出来，当调用函数 CreateWindow 时，就会创建窗口树中的节点，并根据 CreateWindow 传递的父窗口句柄将当前窗口的节点插入到窗口树的合适位置。

需要提及的是：这个窗口树的结构对于服务器端（即桌面进程）和客户机端（即应用程序端）都是一样的。对于桌面进程而言，它也有一个主窗口，这个主窗口会包含一些控件，包括：桌面上代表应用程序的图标、桌面的标题栏、软键盘、输入法以及开始菜单等。

4.2 窗口的 Z 序

窗口的 Z 序是窗口在屏幕上体现出来的前后关系，Z 序越大，则对应的窗口越靠近上层。所以说，Z 序大的窗口将会剪切 Z 序值比它小的所有窗口。如图 3-4 所示：

3 号窗口 Z 序值最大，如果当前屏幕上只有三个窗口，则 1 号窗口要被 2、3 号窗口剪切，而 2 号窗口要被 3 号窗口剪切，而 3 号窗口则不会被任何窗口剪切。

在 LGUI 系统中，窗口之间的 Z 序是通过窗口链表之间的前后关系体现的，如果在链表的前面，表明窗口的 Z 序也靠前，否则靠后。

窗口之间的这种 Z 序关系是非常重要的，因为它是计算窗口剪切关系的依据。

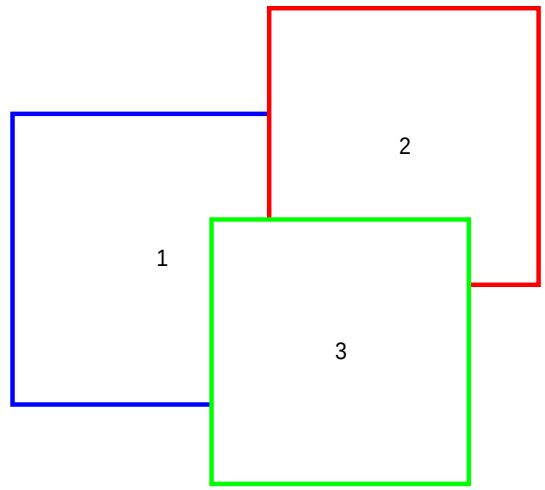

图 4-3 LGUI 窗口 Z 序

4.3 窗口的剪切与剪切域

4.3.1 如何生成窗口剪切域

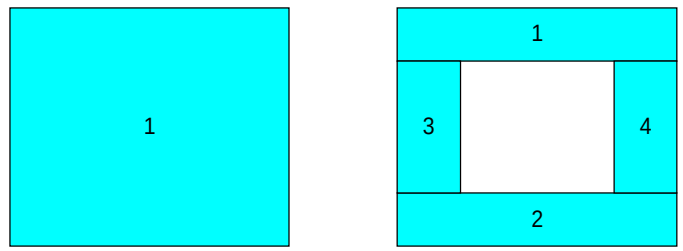

图 4-4 剪切域的生成过程

在窗口系统中，为什么要计算和维护窗口的剪切域呢？这是因为，窗口系统本身就要求对于一个窗口的输出，以不破坏其他窗口为前提，否则，不受限制的任意输出，窗口系统也就不能称其为窗口系统了。这就要求任何一个窗口，必须知道当前它可以在哪些地方输出，而哪些地方又是不允许输出的。这就是所谓的

剪切域。

在 LGUI 系统中，每个窗口必须保存自身的剪切域，剪切域目前实现的实际是剪切矩形，并以链表的形式进行存储，即每个窗口存储它当前可以进行输出的所有矩形区域的集合。

剪切域的生成可用图 4-4 表示。

在图 4-4 中，左图是原始的窗口，它的剪切域即是窗口的矩形，在这个矩形范围内，对于窗口的输出是不受限制的。

右图是屏幕上新输出一个窗口后的情形。新输出一个窗口后，原来窗口的剪切域会分裂成如右图所示的四个部分，而原来中间的那部分，则不包括在窗口的剪切域范围内。因为针对窗口被覆盖部分的输出受到限制，从而也使上面的窗口不会被破坏。当然上层窗口的输出也会只限制在它本身剪切域的范围内，从而也不会破坏下层的窗口。

4.3.2 LGUI 中窗口/控件剪切域的生成过程

以上是剪切域的简单的示例，实际实现时，因为要考虑主窗口、子窗口、控件之间的相互关系，剪切域的运算还是比较复杂的。

桌面主窗体的剪切域是这样计算的：

桌面边界矩形剪去输入法窗口边界矩形；

剪去软键盘窗口边界矩形；

剪去所有应用程序的主窗口边界矩形

剪去桌面上所有可见图标的窗口边界矩形

得到最后的剪切域

主窗口的剪切域是这样计算的：

得到当前进程的初始剪切域；

剪去所有的子窗口的边界矩形；

剪去主窗口上的控件。

得到最后的剪切域

子窗口的剪切域是这样计算的：

得到当前进程的初始剪切域；

与主窗口客户区边界矩形相交；

与子窗口的边界矩形相交；

剪去 Z 序大于该子窗口的兄弟窗口边界矩形；

剪去该子窗口上的控件边界矩形；

得到取后的剪切域。

控件剪切域的生成因其父窗口的类型不同而有所不同。

桌面上控件的剪切域是这样计算的：

控件的外接矩形为初始矩形；

剪去输入法窗口的边界矩形；

剪去软键盘窗口的边界矩形；

与桌面边界矩形相交；

剪去所有应用主窗口边界矩形；

剪去 Z 序大于该控件的兄弟控件边界矩形；

得到最后的剪切域。

主窗口上控件的剪切域是这样计算的：

控件的外接矩形为初始矩形；

与主窗口客户区边界矩形相交；

剪去所有子窗口的边界矩形；

剪去 Z 序大于该控件的兄弟控件边界矩形；

得到最后的剪切域。

子窗口上控件的剪切域是这样计算的：

1．控件的外接矩形为初始矩形；

2．与主窗口客户区边界矩形相交；

3．与父窗口客户区边界矩形相交；

4．减去其他子窗口的边界矩形（这些子窗口是当前控件父窗口的兄弟，并且这些子窗口的 Z 序要大于当前控件父窗口）

5．剪去 Z 序大于该控件的兄弟控件边界矩形；

6．得到最后的剪切域。

注：以上所叙述的操作过程都是指对剪切域的矩形链表进行操作。操作主要的两种：一是剪切，二是求交集。其中剪切是用一个矩形剪切一个矩形链表，要分别对链表中的矩形进行剪切操作，剪切操作之后的结果仍然是一个链表；第二种操作是求交集，也是一个矩形与一个矩形链表进行操作，操作过程是该矩形与链表中的每一个矩形分别进行交集操作，操作的结果仍然保存为一个链表。

另外需要注意的是，这两种操作结束后，链表中的部分节点可能会变成不含有矩形的“空”节点，所以每次操作完后要过滤一遍，删除这样的节点。

4.3.3 LGUI 中窗口剪切域的存储方法

```
typedef struct _ClipRect
{
    RECT            rect;
    struct _ClipRect* pNext;
} ClipRect;
typedef ClipRect* PClipRect;

// Clip Region
typedef struct _ClipRegion
{
    RECT            rcBound;    // bound rect of clip region
    PclipRect       pHead;      // clip rect list head
    PclipRect       pTail;      // clip rect list tail
    PprivateHeap    pHeap;      // heap of clip rect
} ClipRegion;
typedef ClipRegion* PClipRegion;
```

图 4-5 LGUI 中剪切域的存储方法

从图 4-5 中可以看出，LGUI 中剪切域就是以矩形链表的形式存储。为了加快内存分配速度，同时为了避免内存碎块化，我们将矩形链表存入在一个预申请的堆中。详细内容可以参考源代码。

4.4 进程主窗口的初始剪切域与进程内窗体剪切域

我们知道，作为服务器端进程的桌面进程是一个与众不同的进程，它除了维护它自身的剪切域及其控件的剪切域之外，还负责为每一个应用进程维护其主窗口的初始剪切域。在上文中说到，每个应用启动以后，首先与服务器端建立连接，连接完成后，就会将自身的一些信息发送到服务器端，这些信息主要包括：进程的 ID 号、应用主窗体的矩形位置。

而在服务器端，时刻保存有当前与自己连接的所有客户进程的一个链表，并维护着这些应用之间的 Z 序关系（即哪个应用在最上层，哪个在最下层？）同窗

体之间的剪切关系一样，应用之间的 Z 序也是桌面进程为每个客户进程生成初始剪切域的依据。

桌面进程接收到客户进程发送的显示主窗口的消息以后，首先会重新计算桌面本身以及桌面上控件的剪切域（因为客户进程显示的主窗体当然首先会剪切桌面本身和桌面图标），随之，马上会为每个当前运行的进程重新计算主窗体的初始剪切域，并将计算结果发送到对应的进程。而客户进程在收到剪切域变化消息以后，必须以主窗体的初始剪切域为基础，重新计算其中每一个窗体的剪切域。因为新启动一个应用进程以后，默认将处于最上层，所以它将影响所有现有的进程。

为什么叫进程的初始剪切域呢？因为对于任何一个进程而言，它的任何一个窗体在计算剪切域时，首先必须是在由桌面剪切域传递过来的剪切域范围之内进行，所以称之为初始剪切域。

4.5 客户端对剪切域的管理

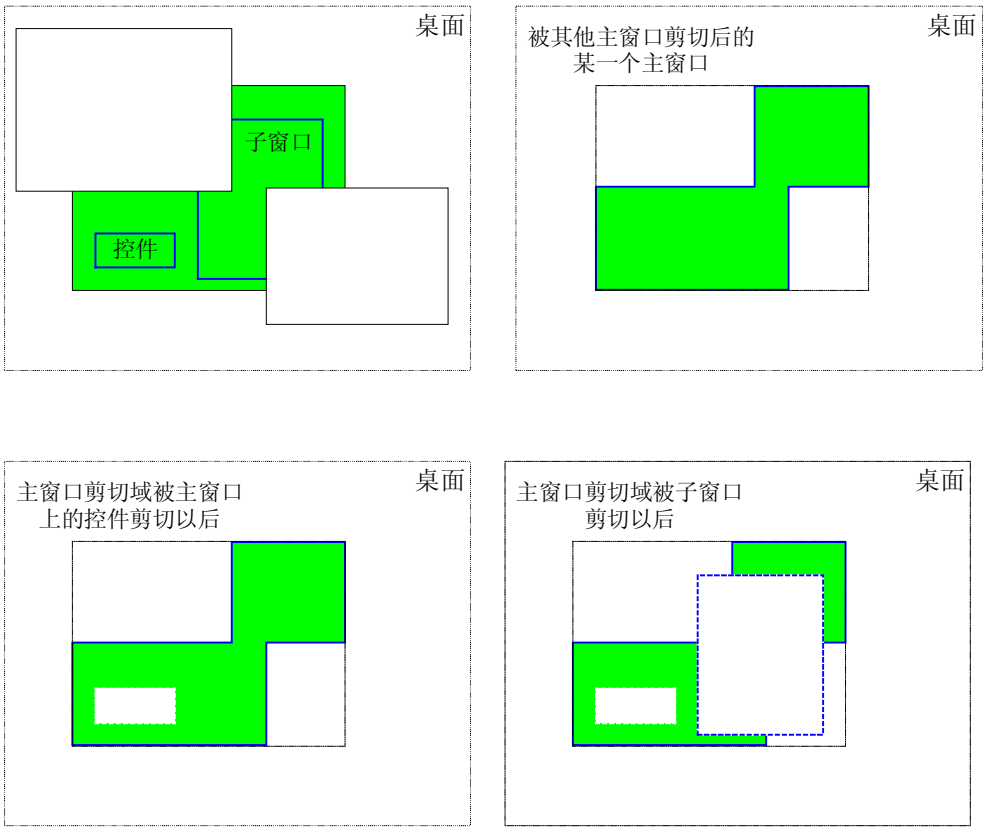

图 4-5　主窗口剪切域生成过程

首先，无论是服务器进程还是客户进程，其中的每一个窗口都会保存自己的剪切域。在窗口的位置、Z 序等发生变化的时候，系统都会为每一个与此变化相关的窗口重新计算剪切域。

这里需要着重说明的是，在 LGUI 的客户/服务器模式中，当客户在需要显示主窗口时，发送请求并在得到服务器端的响应以后，将主窗口进行显示。而同时，其他客户端进程将收到变化了的主窗口的剪切域。

其他客户端收到消息以后，需要作一系列的处理：

1．主窗口剪切域的计算

由服务器发送过来的初始剪切域只是主窗口被桌面软键盘、状态栏、开始菜单以及其它进程的主窗口剪切以后的结果，主窗口还需要被主窗口上的控件以及子窗口进行剪切，才能生成真正的主窗口剪切域。如图 4-4 所示：

2．客户进程中其他窗体剪切域的计算

客户进程内其他窗口剪切域的计算都是以主窗口的剪切域为基础进行的，包括：主窗口上的控件、子窗口、子窗口上的控件。实际上，客户进程在收到由于其他进程主窗体的变化而引起的当前进程主窗体的初始剪切域的变化消息后，不得不对当前进程中所有窗体的剪切域重新进行计算。根据前文叙述，剪切域全部以矩形链表的形式进行组织与存储，任意两个窗口（包括控件）之间的相交，都将产生剪切的关系，所以如果一个客户进程中的窗口关系过于复杂，则剪切域的计算也会消耗一定的系统资源，当然，这是一个窗口系统必须付出的代价。

客户进程内除主窗口外其他窗口剪切域的计算过程如下：

主窗口上控件的剪切域是主窗口的初始剪切域与控件的交集（即互相重叠部分），再剪掉 Z 序比当前控件大的兄弟控件，再剪掉当前进程中的所有子窗口而形成的剪切域；

子窗口的剪切域是主窗口初始剪切域与子窗口的交集剪去 Z 序比它大的其他子窗口和本身窗口上的控件形成的剪切域；

子窗口上控件的剪切域是主窗口的初始剪切域与当前控件的交集，剪掉 Z 序比其父窗口大的所有子窗口，再剪掉 Z 序比其大的所有兄弟控件而形成的剪切域。

在这里要说明的是，这种剪切域管理方式实现的前提是任何一个应用中，主窗口是所有其他窗口（包括控件）的最大边界，即在一个应用中没有任何窗口或控件的边界会超出主窗口的边界。这虽然是一种折衷方案带来的限制，但这种限制在面向嵌入式设备的 GUI 环境中是可以接受的，而且它带来的运行效率的提高与资源消耗的降低是非常明显的。实际上，对于嵌入式 GUI 环境来讲，使用者并不会对这种限制感到不便。

4.6 窗口类的注册管理

在 Windows 编程的时候，我们为了创建一个窗口，首先需要向系统创建一个窗口类，然后再根据窗口类创建窗口。在 LGUI 中，也有类似的机制，不论是主窗口、子窗口、控件，或者对于 LGUI 系统内部的桌面窗口、桌面上的图标控件、桌面上的软键盘等等都是需要先注册一个窗口类。对于 LGUI 系统内部的窗口，LGUI 在启动时会自动进行注册，对于基于 LGUI 的应用程序，在创建自己的窗口时就需要先注册窗口类。

4.6.1 为什么要注册窗口类

注册窗口类的目的主要是为了实现面向对象的特征。例如对于普通的按钮，如果每一个按钮都要写一大堆代码，那岂不是非常繁琐。所以我们就把按钮注册为一类窗口。根据类创建的按钮都是这个按钮类的实例，这样保证所有按钮的外观特征及行为特征是类似的。

4.6.2 如何注册窗口类

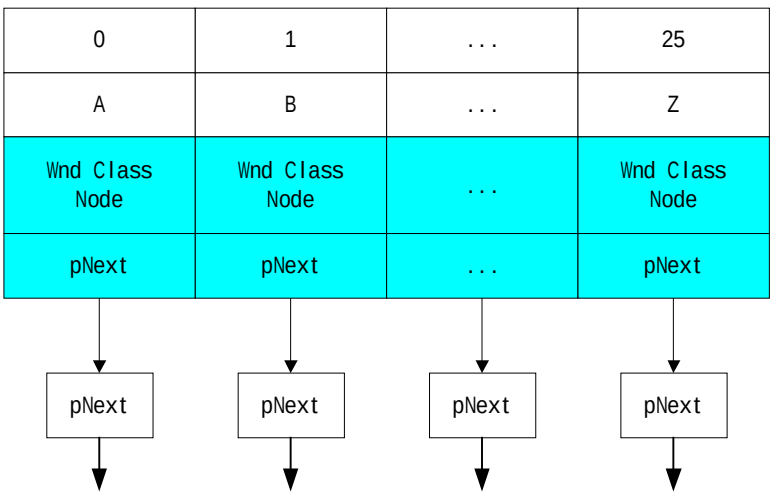

图 4-6 LGUI 窗口注册表的管理方法

窗口的注册在系统中通过一个 Hash 表进行管理，如图 4-6 所示。

从图中可以看出，以某一字母开头的所有注册的窗体类的节点在一个链表上。注册窗口类时，限制窗口类的名字以字母开头。注册或查找窗口类时，先根据注册的窗口类名称的第一个字母找到链表头，然后根据链表的情况进行后续的处理。

通过上面的内容我们知道窗口类是如何注册的，关键的问题是我们还需要知道窗口类到底注册了什么内容？

```
typedef struct tagWNDCLASSEX {
    ……
    WNDPROC                 lpfnWndProc; //窗口消息处理函数
    ……
    HBRUSH                  hbrBackground;//窗口背景画刷
    ……
    char*                   lpszClassName;//类名称
    struct tagWNDCLASSEX*   pNext;
} WNDCLASSEX;
typedef    WNDCLASSEX* PWNDCLASSEX;
```

图 4-7　窗口类注册的主要内容

在 LGUI 中，窗口类注册的内容主要有三项：一是窗口的背景画刷，主要是窗口在绘制的时候用什么画刷来刷新背景；二是窗口的类名称，窗口类在一个进程内唯一；三是窗口消息处理函数，即发送到这个窗口有消息用什么函数来处理。

```
    WNDCLASSEX    wcex;
    HWND          hWnd;
    wcex.lpfnWndProc        = (WNDPROC)WndProc;
    wcex.hbrBackground      = CreateSolidBrush(RGB(147,222,252));
    wcex.lpszClassName      = "mainwin";
    RegisterClass(&wcex);
    hWnd = CreateWindow("mainwin", "main",
            WS_MAIN | WS_VISIBLE | WS_THINBORDER,
            10, 100,150, 80, NULL, NULL, NULL, NULL);
    ShowWindow(hWnd, true);
```

图 4-7　典型的注册窗口类的过程

图 4-7 所示代码开始先给 wcex 的一个字段 lpfnWndProc 赋值，这个字段在结构中被定义为一个函数指针，所以在这里赋值为消息处理函数的函数名称；然后为背景画刷赋值；再为窗口类名赋值，最后调用 RegisterClass 注册窗口类。

随后创建窗口类时，第一个参数就是窗口类的名称，系统正是根据这个类名来创建合适的窗口。

```
LRESULT
WndProc(
    HWND        hWnd,
    UINT        message,
    WPARAM      wParam,
    LPARAM      lParam
)
{
    switch(message)
    {
        case LMSG_CREATE:
            break;
        case LMSG_PENDOWN:
            break;
        ……
        case LMSG_PAINT:
            break;
        case LMSG_DESTROY:
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return true;
}
```

图 4-8 LGUI 中典型的消息处理函数

如图 4-8 是一个典型的消息处理函数的模板，在 LGUI 中所有的消息处理函数都是这样的风格。这个函数的主体就是一个条件分支，根据不同的消息值进行不同的处理，如果其中没有某一个消息的处理分支，则调用系统提供的默认的处理

函数 DefWindowProc()

4.6.3 注册窗口类如何发挥作用

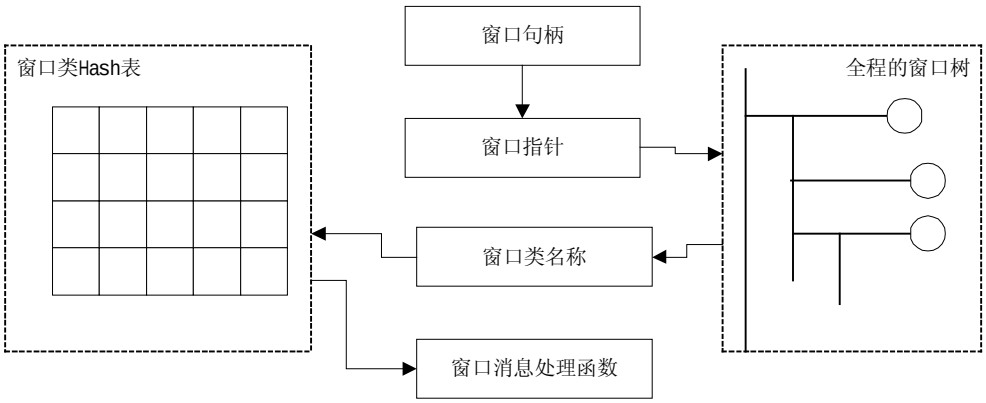

图 4-9 窗口消息处理函数的查找过程

从图中可以看到，通过窗口句柄查找到窗口类名称，再通过窗口类名称查找到窗口的消息处理函数。因为这个过程涉及到窗口消息处理的过程，如果你不太明白其中的含义，请在阅读完 LGUI 的消息管理之后，查看消息处理的过程。

第5章 LGUI 中的消息管理

在任何 GUI 系统中，均有事件与消息驱动的概念。在消息驱动的系统中，计算机外设发生的事件，都由支持系统收集，将其按约定的格式翻译为特定的消息。应用程序一般有自己的消息队列，系统将消息发送到应用程序的消息队列中。

与传统的程序不同，以消息驱动的程序启动后就会一直处于循环状态，在这个循环当中，程序从外部输入设备获取某些事件，比如用户的按键或者鼠标的移动，然后根据这些事件做出某种响应，并完成一定的功能，这个循环直到程序接受到某个消息为止。所以消息传递与消息处理是消息（事件）驱动的 GUI 系统的核心。

通过阅读 LGUI 的源代码，我们可以看到，在 LGUI 系统里有两类消息，第一类是进程之间的消息，第二类是进程内部的消息。进程之间的消息在应用进程与桌面进程之间发送；进程内部的消息是由窗口之间互相发送。

对于进程之间的消息，除了前面章节所述当应用进程创建主窗口、显示主窗口、隐藏主窗口、销毁主窗口时向桌面进程发送消息之外，还有一项重要内容，就是外部事件的处理，主要指按键、鼠标事件。因为我们知道，当同时有多个应用进程在运行的时候，需要桌面进程收集这些消息并根据当前窗口的激活情况将消息发送到相应的进程中去。

5.1 外部事件收集与分发

桌面进程启动后，首先会初始化环境，包括创建专门用于收集外部事件的键盘线程与鼠标线程，然后创建桌面窗口以及桌面上的按钮、标题栏、软键盘、输入法窗口等，之后主线程就会进行循环，等待消息，当收到消息后就处理这些消息。

键盘线程与鼠标线程启动后，会循环读取当前的事件，得到一个事件后，就直接发送到桌面进程的消息队列中去，在 LGUI 里，这些消息名称都以 RAW 开头，表示为原始消息。

桌面进程在收到任何一个消息后，首先要根据当前状态判断这个消息是由自身来处理还是转发到应用进程中去。这个过程可用图 5-1 表示：

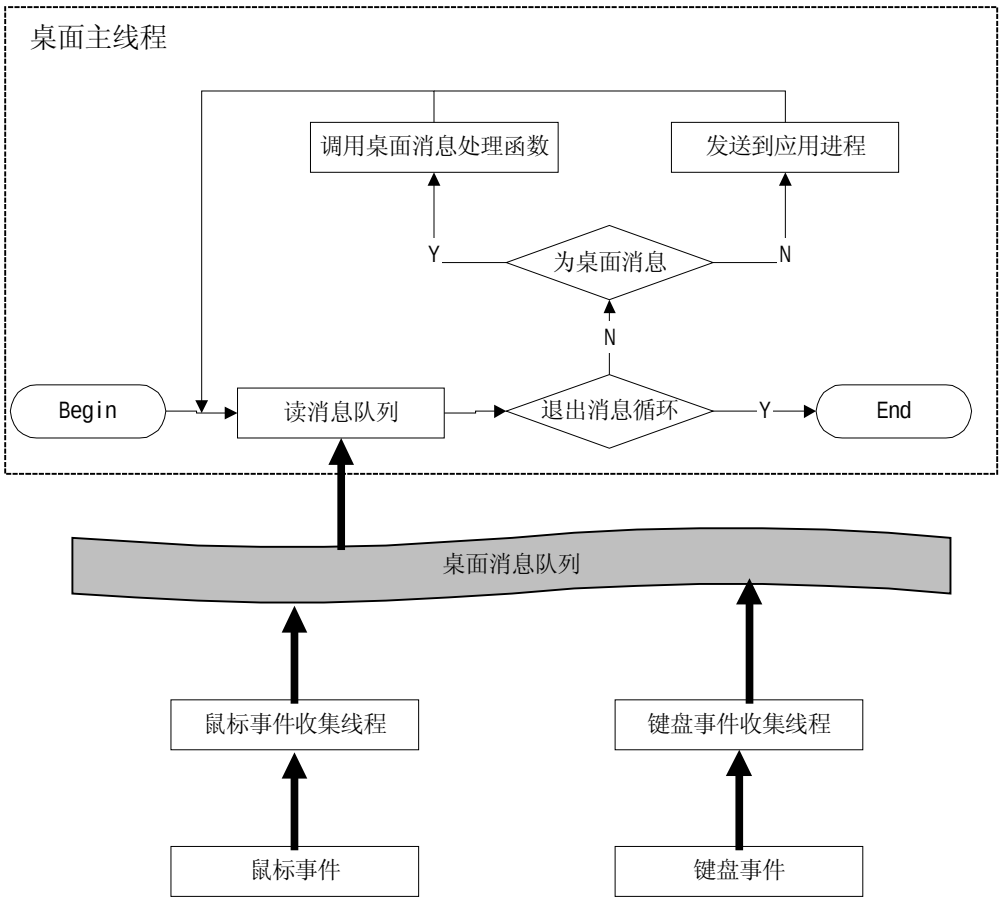

图 5-1　外部消息的收集与分发

其中，由桌面进程自己处理还是转发到其他的应用进程中去，是根据当前鼠标的位置、由桌面进程管理的应用进程主窗口当前的激活状态等情况来进行判断的。例如，如果当前没有应用进程在运行，那边任何外部消息肯会由桌面进程自己来处理；如果有一个应用进程在运行并处于激活状态，则按键消息肯定需要发送到应用进程中去，鼠标消息则要根据鼠标点击的位置来判断，如果点击位置在应用进程主窗口的范围之外，则鼠标消息由桌面进程自己来处理，否则发送到应用进程中去。如果当前至多于一个应用进程在运行，当通过鼠标点击一个处于 Z 序下层的的应用进程的窗口时，桌面进程不仅要将鼠标消息发送到被点击窗口所对应的应用进程中去，而且还要做一系统处理使得被点击窗口所对应的进程处于 Z 序的最上层，并处于激活状态，而点击之前处于 Z 序最上层的应用进程则下降一层，同时不再处于激活状态。

如果消息被发送到应用进程，应用进程还需要根据目前应用进程内各窗体的活动状态将外部消息转发到相应窗口的消息队列中去。例如，假设当前应用进程只有一个主窗口，没有子窗口。如果主窗口上没有控件，则这个消息肯定由主窗口来处理，如果有控件，则根据剪切域判断是发给其中的一个控件还是发给主窗口；有子窗口的情况下，处理过程是类似的。总之，是根据当前的状态判断接收消息的控件或窗口。另外如果应用进程收到的消息是鼠标消息，应用进程还需要根据鼠标的点击位置以及进程中当前的状态判断是否在激活其中一个子窗口还是激活主窗口，窗口的激活状态是很重要的，因为按键消息只会发送到当前激活的窗口。

5.2 消息队列

消息队列是消息的“暂存处”，在 GUI 系统中，消息是与窗口紧密联系在一起的。实际上，在 LGUI 中，除了控件以外，主窗口、子窗口都有自己的消息队列，同时有专门的线程来处理消息。

不论消息队列的物理存储结构有何不同，例如是一个数组，或是一个链表，本质上就是先入先出的队列结构。当需要向一个窗口发送消息的时候，发送者就把消息放到目的窗口的消息队列的队尾，如果目的窗口的消息处理线程当前因没有消息可供处理而处于睡眠状态，则发送消息的一方有义务唤醒它，以使消息得到处理。

所以，在消息驱动的系统，在没有消息处理的时候，所有的窗口都处于等待状态（这种等待属于非忙等待，与传统的循环、查询是否有任务需要处理有本质不同）。当有消息需要处理时，发送方唤醒处理一方来处理消息，这就是消息驱动的根本意义。

5.3 LGUI 的消息

在 LGUI 中，除了桌面进程与客户进程之间传递并互相响应的固定格式的进程间消息以外，其他的消息都以窗口（不包括控件）为单位来管理。所谓以窗口为单位来管理，是指每一个窗口都有一个消息队列，并且每一个窗口都有一个消息处理线程专门用来处理消息。

窗口的数据结构中，包括有一个消息队列的结构，所有发送到窗口的消息，将会保存到窗口的消息队列中。

在 LGUI 中，将消息进行了分类，分别是：通知消息、邮寄消息、同步消息以及绘制消息。根据不同的消息分别建立消息队列。

通知消息是以链表方式存储的消息，是不允许丢失的重要消息。在 LGUI 中，这类消息是通过函数 SendNofityMessage 来发送的。

邮寄消息是不太重要的消息，允许丢失，这种消息不是以链表方式存储，而是以固定长度的数组来存储，当数组已满时，这种消息就会丢失。这类消息主要以鼠标消息和键盘消息为主。设置这类消息的目的是使系统在非常繁忙时可以丢弃一些不太重要的消息，以免因消息来不及处理而毫无意义地占用大量的内存空间。在 LGUI 中，这类消息是以 PostMessage 函数来发送的。

同步消息是要求系统马上响应的消息。消息发送者在发送同步消息后会处于阻塞状态，只有消息处理一方处理完消息后，调用者才能继续执行下面的工作。同步消息队列中有一个信号量来实现这种功能，消息发送者在发送消息后，通过 V 操作阻塞，消息处理者处理完消息后，通过 P 操作唤醒发送者。在 LGUI 中，这类消息是以 PostSyncMessage 函数来发送的。

绘制消息前面已有讨论，实际上，因为窗体已有无效区链表用于保存无效区，绘制消息链表只是要求绘制的消息链表而已。在 MiniGUI 中，并没有绘制消息链表，而只是通过在消息队列中设置一个绘制标志。在 LGUI 中，设置成一个队列的目的是为了保存控件的绘制消息。为什么要这样做呢？这是因为在 LGUI 中，控件是没有消息队列的，因为有消息队列就需要有单独的线程处理消息，如果每增加一个控件就要增加一个线程，则系统肯定会不堪重负。但是如果在窗体中不保存该窗体上控件的绘制消息，则对于控件的绘制就必须立即处理，这将不符合我们尽量提高系统实时性的要求。将控件的绘制消息都保存在其父窗口的消息队列中，可以保证窗口在没有其他消息的情况下才执行绘制消息，以此提高系统对于更为重要消息的响应速度。在 LGUI 中，这类消息是以 SendPaintMessage 函数来发送的。

下面分别就几类消息的处理方法进行说明：

5.3.1 LGUI 的消息队列结构

结构中第一个成员变量为互斥量 mutex,互斥量的作用是为了使不同线程操作同一队列是起到互斥作用。便如线程 A 发送消息到消息队列，线程 B 同时从队列中取消息。则线程 A 在往队列中写入消息时首先通过互斥量锁定，这时如果线程 B 试图通过操作消息队列就会进入等待状态，只有当线程 A 完成操作并解锁后，线程 B 才可以继续操作消息队列。

```
//LGUI 的消息队列结构
typedef struct _MsgQueue
{
    pthread_mutex_t    mutex;          //互斥量
    sem_t              sem;            //信号量
    DWORD              dwState;        //消息队列状态
    PSyncMsgLink       pHeadSyncMsg;   //同步消息队列头
    PSyncMsgLink       pTailSyncMsg;   //同步消息队列尾
    PNtfMsgLink        pHeadNtfMsg;    //通知消息队列头
    PNtfMsgLink        pTailNtfMsg;    //通知消息队列尾
    PNtfMsgLink        pHeadPntMsg;    //绘制消息队列头
    PNtfMsgLink        pTailPntMsg;    //绘制消息队列尾
    WndMailBox         wndMailBox;     //邮寄消息队列数组
    HWND               TimerOwner[NUM_WIN_TIMERS];
    int                TimerID[NUM_WIN_TIMERS];
    WORD               TimerMask;
}   MsgQueue;
typedef MsgQueue*      PMsgQueue;
```

图 5-2 LGUI 的消息队列结构

线程 A 操作过程如下所示：

```
pthread_mutex_lock(&msgqueue.mutex);     //锁定互斥锁
……                                       //往队列中放入消息
pthread_mutex_unlock(&msgqueue.mutex);   //解锁互斥锁
```

线程 B 操作过程如下所示：

```
pthread_mutex_lock(&msgqueue.mutex);     //锁定互斥锁
……                                       //从队列中取出消息
pthread_mutex_unlock(&msgqueue.mutex);   //解锁互斥锁
```

我们知道，互斥锁的作用就象一把门锁，第一个人进入一间屋子的时候，将门锁上，这样其他的人都无法进入；只有当他打开锁从屋子里出来的时候，后面的人才可能进入。如果所有准备进入屋子的人都遵守这样的规则，就可以保证任何时间屋子里只有一个人。互斥锁是线程同步的重要工具。

为什么对于消息队列的操作要通过互斥锁来进行保护呢？原因是不同线程同

时操作一个队列可能引发数据不完整的问题，例如当其中一个线程操作到中途，由于线程时间片已到，CPU 转而由第二个线程控制，并对同一数据进行操作，这样导致操作的结果的不确定，引起系统问题。我们在前面的讲述里说过，在 LGUI 里，每个窗口分别对应于一个线程，这些线程的主要任务就是处理消息，而在处理消息的过程中就有可能向其他窗口发送消息，这就涉及到线程同步的问题，而使用互斥量就可以很好地解决这个问题。

消息队列结构的第二个变量是信号量，信号量也是线程同步的重要工具。那么在消息队列里，信号量主要起到什么作用呢？我们通过发送消息我获取消息的一些关键代码便能消晰地了解到。

```
BOOL GUIAPI
GetMessage(
        PMSG    pMsg,
        HWND    hWnd
)
{
        PMsgQueue pMsgQueue;                    //定义消息队列变量
        pMsgQueue = GetMsgQueue(hWnd);          //得到消息队列指针
loop:
        ……                                      //操作消息队列
        sem_wait (&pMsgQueue->sem);             //信号量 V 操作
        goto loop;                              //循环操作
        return 1;                               //返回
}
```

图 5-3　获取消息的关键代码

从图 5-3 获取消息的关键代码我们可以看到，GetMessage 函数是一个死循环。当然，在省略的操作消息队列的代码中的情况是：如果获取到一条消息，就会退出这个循环。

这个循环的最后一步都是一个信号量的 P 操作，我们知道，P 操作的过程是这样的：先使信号量的值减一，如果减完以后信号量的值等于零，则线程就会在这里进入睡眠状态，只到被其他线程唤醒。这里我们可能会有一个疑问：为什么要进行一次 P 操作呢？也就是说为什么要使线程进行睡眠状呢？原因就是为了消息的非忙等待。

我们从 LGUI 的代码中可以看出，所有主窗口对应于进程的主线程，进程启动

后，主线程就会进行消息循环，不停地取消息、分发消息；对于一个子窗口对应的线程也是类似的，取消息、分发消息。如图 5-3 所示：

```
while (GetMessage(&msg,hWnd)){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

图 5-4　线程的主要任务是取消息、分发消息

所以，如果线程始终处于忙碌的查询消息状态，那对于系统资源的消耗就会比较大。我们采取了信号量来使线程在没有消息可取的消息下处于睡眠状态。通过这种设计，既便在 LGUI 中，一个窗口对应于一个消息队列，对应于一个线程，如果 LGUI 启动后，同时启动了数个进程，则同时运行的线程会比较多，但能过合理的设计，很多线程同时运行，并不会过多消耗系统资源。

回到 GetMessage 函数的功能方面。在函数里，每次循环都会对信号量进行一次 P 操作，一次操作使得信号量的值减小一，在信号量的值变成零之前，线程是不会被阻塞的。但当信号量的值变为零之后，此时消息队列中没有消息可供处理，线程于是进行睡眠状态。

线程的唤醒是通的发送消息函数，通过查看 LGUI 的源代码，发送消息函数，例如 PostMessage 函数、PostSyncMessage 函数、SendNotifyMessage 函数，在完成必要的操作之后，最后总有一个对于消息队列信号量的 V 操作，V 操作的结果是信号量的值增加一，同时会唤醒等待的线程继续进行处理。

我们知道，互斥量操作与信号量操作是现代操作系统理论中非常重要的思想。在 LGUI 中的消息队列的处理中，正是使用了互斥量与信号量来使得多个线程能够实现同步。

5.3.2 通知消息（NotifyMessage）

通知消息有如下的一个队列：

从图 5-5 可以看到，通知消息队列是一个链表结构，由于链表没有长度限制，所以重要的、不允许丢失的消息就发送到通知消息队列中。通知消息使用 SendNotifyMessage 函数发送。SendNotifyMessage 函数的代码如图 5-5 所示。

从代码中可以看到，SendNotifyMessage 的主要功能就是从私有堆中申请一

个节点的空间，赋值后插入到消息队列的链表中去。在操作消息队列链表的时候，首先要锁定互斥量，最后要解锁互斥量。在代码的最后，是对信号量的操作：先取得信号量的值，如果信号量的值等于零，说明取消息的线程目前正处于睡眠状态，这时就要能过 V 操作来唤醒线程。

```
//消息结构
typedef struct _MSG {
    HWND        hWnd;
    UINT        message;
    WPARAM      wParam;
    LPARAM      lParam;
    void*       pData;
} MSG;
typedef MSG* PMSG;
//通知消息队列
typedef struct _NtfMsgLink
{
    MSG                 msg;
    struct _NtfMsgLink* pNext;
} NtfMsgLink;
typedef NtfMsgLink* PNtfMsgLink;
```

图 5-5　通知消息的数据结构

```
BOOL GUIAPI
SendNotifyMessage(
    HWND hWnd,
    int iMsg,
    WPARAM wParam,
    LPARAM lParam
)
{
    PmsgQueue    pMsgQueue;
    PntfMsgLink  pLinkNode;
    Int          sem_value;
    pMsgQueue = GetMsgQueue(hWnd);          //消息队列指针
    pLinkNode = HeapAlloc(&MsgQueueHeap);   //从堆中分配空间
    pthread_mutex_lock(&pMsgQueue->mutex);  //锁定互斥量
    pLinkNode->msg.hWnd = hWnd;
    pLinkNode->msg.message = iMsg;
    pLinkNode->msg.wParam = wParam;
    pLinkNode->msg.lParam = lParam;
    pLinkNode->msg.pData = NULL;
    pLinkNode->pNext = NULL;
    if(!pMsgQueue->pHeadNtfMsg){
        pMsgQueue->pHeadNtfMsg =
            pMsgQueue->pTailNtfMsg = pLinkNode;
    }else{
        pMsgQueue->pTailNtfMsg->pNext = pLinkNode;
        pMsgQueue->pTailNtfMsg = pLinkNode;
    }
    pMsgQueue->dwState |= QS_NOTIFYMSG;     //设置状态位
    pthread_mutex_unlock (&pMsgQueue->mutex); //解锁互斥量
    sem_getvalue (&pMsgQueue->sem, &sem_value);//取信号量值
    if (sem_value == 0)
        sem_post(&pMsgQueue->sem);//信号量 V 操作
    return true;
}
```

图 5-6 SendNotifyMessage 函数的代码

5.3.3 邮寄消息

邮寄消息的消息队列格式如图 5-7 所示

```
typedef struct _WndMailBox
{
    MSG         msg[SIZE_ MAILBOX];
    int         iReadPos;
    int         iWritePos;
} WndMailBox;
typedef WndMailBox* PWndMailBox;
```

图 5-7 邮寄消息队列格式

邮寄消息的消息队列是一个固定大小的静态数组，其中有两个指针指示当前读的位置与写的位置。

当需要邮寄消息时，调用函数 PostMessage。PostMessage 的代码如图 5-8 所示。从代码中可以看出，在操作数组之前，首先会锁定互斥量，然后判断队列是否已满，如果已满，则直接退出，否则将消息写入到数组之中，再解锁互斥量。函数退出之前的操作与 SendNotifyMessage 的操作是类似的。先测试信号量的值是否等于零，如果为零，表时取消息的线程处于睡眠状态，需要通过 V 操作来唤醒。

邮寄消息正如他的名字一样，调用邮寄消息函数的线程并不确认消息是否已存入到目的窗口的消息队列中，正如我们日常生活中发送邮件一样，我们写好信后丢如邮筒，以后的事情我们无法控制，全靠邮政公司的服务水平了。

在 LGUI 中设置这种消息的主要原因是为了处理多余的按键、鼠标消息。如果系统处于忙的状态，无法处理用户的操作请求。这时如果将大量的操作消息存储起来，等系统空间时再处理，在实际应用中是没有意义的，只会无谓消耗系统空间。而且等系统从忙转到空闲，可能需要比较长的时间，这时再去处理用户不久前的操作请求，已经是没有意义的，而且会引起不必要的误解，所以最的办法就是在系统忙的时候，直接丢弃多余的操作消息。

```
BOOL GUIAPI
PostMessage(
    HWND hWnd,
    int iMsg,
    WPARAM wParam,
    LPARAM lParam
)
{
    PmsgQueue    pMsgQueue;   //消息队列指针
    PMSG         pMsg;        //消息指针
    Int          sem_value;   //信号量
    pMsgQueue = GetMsgQueue(hWnd);              //获取消息队列指针
    pthread_mutex_lock(&pMsgQueue->mutex);      //锁定互斥量
    if(iMsg == LMSG_QUIT){   //如果为退出消息，设置标志位
        pMsgQueue->dwState |= QS_QUITMSG;
    }else{
        if((pMsgQueue->wndMailBox.iWritePos + 1) % SIZE_MAILBOX
            == pMsgQueue->wndMailBox.iReadPos)
            return false;    //消息队列满，退出
        pMsg = &(pMsgQueue->wndMailBox.msg[
                pMsgQueue->wndMailBox.iWritePos]);
        pMsg->hWnd = hWnd;
        pMsg->message = iMsg;
        pMsg->wParam = wParam;
        pMsg->lParam = lParam;
        pMsgQueue->wndMailBox.iWritePos ++;
        if(pMsgQueue->wndMailBox.iWritePos >= SIZE_MAILBOX)
            pMsgQueue->wndMailBox.iWritePos = 0;
            pMsgQueue->dwState |= QS_POSTMSG;    //设定标志位
    }
    pthread_mutex_unlock (&pMsgQueue->mutex);        //解锁互斥量
    sem_getvalue (&pMsgQueue->sem, &sem_value);      //获取信号量值
    if (sem_value == 0)
        sem_post(&pMsgQueue->sem);    //如果信号量等于零，V 操作
    return true;
```

图 5-8 PostMessage 的关键代码

5.3.4 同步消息

同步消息的结构如图 5-9 所示：

```
typedef struct _SyncMsgLink
{
    MSG         msg;
    int         iRetValue;
    sem_t       sem;
    struct _SyncMsgLink* pNext;
} SyncMsgLink;
```

图 5-9 同步消息队列格式

同步消息是要求系统马上响应的消息。消息发送者在发送同步消息后会处于阻塞状态，只有消息处理一方处理完消息后，调用者才能继续执行下面的工作。同步消息队列中有一个信号量用来实现这种功能，消息发送者在发送消息后，通过 P 操作阻塞，消息处理者处理完消息后，通过 V 操作唤醒发送者。在 LGUI 中，这类消息是以 PostSyncMessage 函数来发送的。

PostSyncMessage 函数的关键代码如图 5-10 所示：

从代码中可以看到，函数将消息赋值并发送到消息队列、初始化信号量、设置标志以后，通过 V 操作唤醒等待的线程处理线程。然后会等待消息被处理，当处理完成后，再销毁信号量。

这里的技巧在于对同步消息队列结构中信号量的操作：每次发送同步消息之前，先初始化这个信号量，发送完消息后，唤醒处理线程，同时通过对这个信号量的 P 操作使线程阻塞并等待处理完成。而处理消息的线程在处理完一个同步消息后，需要对同样的信号量作 V 操作，以唤醒等待中的发送消息线程。能过这种方式，实现同步消息的处理。

5.3.5 绘制消息

在 LGUI 中，为了提高系统对于其他实时性要求更高的消息的响应速度，将对于窗口的绘制消息单独形成队列，并且在取消息的过程中，将绘制消息的处理优

先级设置为最低，即只在没有其他消息的情况下才处理绘制消息。详细内容可以参考 GetMessage 函数。

```
int
PostSyncMessage(
        HWND hWnd,
        int msg,
        WPARAM wParam,
        LPARAM lParam
)
{
        PmsgQueue           pMsgQueue;
        SyncMsgLink         syncMsgLink;
        Int                 sem_value;
        if(!(pMsgQueue = GetMsgQueue(hWnd)))        //获取消息队列指针
                return -1;
        pthread_mutex_lock(&pMsgQueue->mutex);      //锁定互斥量
        syncMsgLink.msg.hWnd = hWnd;                //消息赋值
        syncMsgLink.msg.message = msg;
        syncMsgLink.msg.wParam = wParam;
        syncMsgLink.msg.lParam = lParam;
        syncMsgLink.pNext = NULL;
        sem_init(&syncMsgLink.sem,0,0);             //初始化信号量
        ……                                          //操作消息队列
        pMsgQueue->dwState |= QS_SYNCMSG;           //设置消息标志
        pthread_mutex_unlock (&pMsgQueue->mutex);   //解锁互斥量
        sem_getvalue (&pMsgQueue->sem, &sem_value);//获取信号量值
        if (sem_value == 0)
                sem_post(&pMsgQueue->sem);          //激活等待的线程
        sem_wait(&syncMsgLink.sem);                 //等待处理
        sem_destroy(&syncMsgLink.sem);              //销毁信号量
        return syncMsgLink.iRetValue;
}
```

图 5-10 PostSyncMessage 的关键代码

绘制消息与窗口的无效区的管理密切相关，在后面的章节要进行详细讨论。

因为窗体已有无效区链表用于保存无效区，绘制消息链表只是要求绘制的消息链表而已。在有些开源的 Linux 的 GUI 中，并没有绘制消息链表，而只是通过在消息队列中设置一个绘制标志。在 LGUI 中，设置成一个队列的目的是为了保存控件的绘制消息。为什么要这样做呢？这是因为在 LGUI 中，控件是没有消息队列的，因为有消息队列就需要有单独的线程处理消息，如果每增加一个控件就要增加一个线程，则系统肯定会不堪重负，而且本身也没有任何必要性。但是如果在窗体中不保存该窗体上控件的绘制消息，则对于控件的绘制就必须立即处理，这将不符合我们尽量提高系统实时性的要求。将控件的绘制消息都保存在其父窗口的消息队列中，可以保证窗口在没有其他消息的情况下才执行绘制消息，以此提高系统对于更为重要消息的响应速度。在 LGUI 中，这类消息是以 SendPaintMessage 函数来发送的。

SendPaintMessage 的关键代码参见图 5-12

从代码中可以看到，SendPaintMessage 的操作与其他发送通知消息（SendNotifyMessage）的操作是类似的，锁定互斥量后对消息队列进行操作，操作完成后解锁互斥量，然后通过 V 操作唤醒等待的队列。

5.3.6 其他消息发送方式

我们从 LGUI 中代码中，可以看到一个特别的函数，SendMessage。这个函数主要有两个功能：如果消息发送线程与消息处理线程是同一线程，则直接调用消息接收窗口的消息处理函数，如果不是同一线程，则发送同步消息。

```
if(
        ((iWinType == WS_DESKTOP) ||
        (iWinType == WS_MAIN) ||
        (iWinType == WS_CHILD))
        &&
        (pWin->threadid != pthread_self())
      )
        return PostSyncMessage (hWnd, iMsg, wParam, lParam);
WndProc = GetWndProc(hWnd);
return (*WndProc)(hWnd, iMsg, wParam, lParam);
```

图 5-11 SendMessage 的关键代码

```
BOOL
SendPaintMessage(
    HWND hWnd
)
{
    PmsgQueue    pMsgQueue;
    PntfMsgLink  pLinkNode;
    int          sem_value;
    pMsgQueue = GetMsgQueue(hWnd);                  //获取消息队列
    if(!pMsgQueue)
        return false;
    pLinkNode = HeapAlloc(&MsgQueueHeap);
    pthread_mutex_lock(&pMsgQueue->mutex);          //锁定互斥量
    pLinkNode->msg.hWnd = hWnd;                     //消息赋值
    pLinkNode->msg.message = LMSG_PAINT;
    pLinkNode->msg.wParam = (WPARAM)NULL;
    pLinkNode->msg.lParam = (LPARAM)NULL;
    pLinkNode->msg.pData = NULL;
    pLinkNode->pNext = NULL;
    if(!pMsgQueue->pHeadPntMsg){                    //加入到队列中
        pMsgQueue->pHeadPntMsg =
        pMsgQueue->pTailPntMsg = pLinkNode;
    }else{
        pMsgQueue->pTailPntMsg->pNext = pLinkNode;
        pMsgQueue->pTailPntMsg = pLinkNode;
    }
    pMsgQueue->dwState |= QS_PAINTMSG;              //设置标志位
    pthread_mutex_unlock (&pMsgQueue->mutex);       //解锁互斥量
    sem_getvalue (&pMsgQueue->sem, &sem_value);     //获取信号量值
    if (sem_value == 0)
        sem_post(&pMsgQueue->sem);                  //信号量 V 操作
    return true;
}
```

图 5-12 SendPaintMessage 的关键代码

5.4 消息处理

这一节的主要任务是回答上一章最后一个问题，注册窗口类如何发挥作用。

```
typedef struct _MSG {
    HWND        hWnd;
    UINT        message;
    WPARAM      wParam;
    LPARAM      lParam;
    void*       pData;
} MSG;
typedef MSG* PMSG;
```

图 5-13 消息结构

如图 5-13 所示，LGUI 中消息结构中包含以下几个字段，hWnd 是指消息的窗口句柄，即这个消息应该由那个窗口来处理；第二个 message 是消息值，表明这是一个什么消息；第三、第四分别是消息的附加值，可以根据消息来定义附加值的具体含义，在 LGUI 一些固定的消息中，这两个附加值的意义是固定的；最后一是 pData 也是一个附加值的指针，在特定的消息里有意义。

注册窗口类发挥作用的过程用一个流程可以说明：

窗口线程在消息循环过程中取到一个消息。如何处理这个消息呢？

首先根据窗口句柄得到窗口的指针（LGUI 中，窗口句柄本身就是窗口指针），这个指针指向窗口树中的一个节点，这个节点中包含窗口描述的详细信息，其中一个信息就是窗口类的名称，得到这个类名称后，就可以在窗口注册表的 Hash 表中得到这个窗口的注册节点，根据注册节点得到消息处理函数指针，然后用 wParam,lParam 两个附加值作为参数调用这个消息处理函数。这就是消息处理的详细过程。

在这里有一个问题需要明确一下。因为我们知道，消息队列与窗口并不一一对应，即控件没有消息队列，而只有主、子窗口有消息队列。所以 LGUI 内部在处理时，将控件的消息发送到其父窗口的消息队列中。那么在消息循环的过程中进行消息处理时也是一样的，取到一个消息后，根据窗口句柄（包括主、子窗口和控件）得到其注册的处理函数，然后进行调用。

第6章 窗口输出及无效区的管理

6.1 窗口的客户区与非客户区

在讲解窗口的输出及无效区的管理之前，先说明一下窗口的客户区与非客户区。如图 6-1 所示：

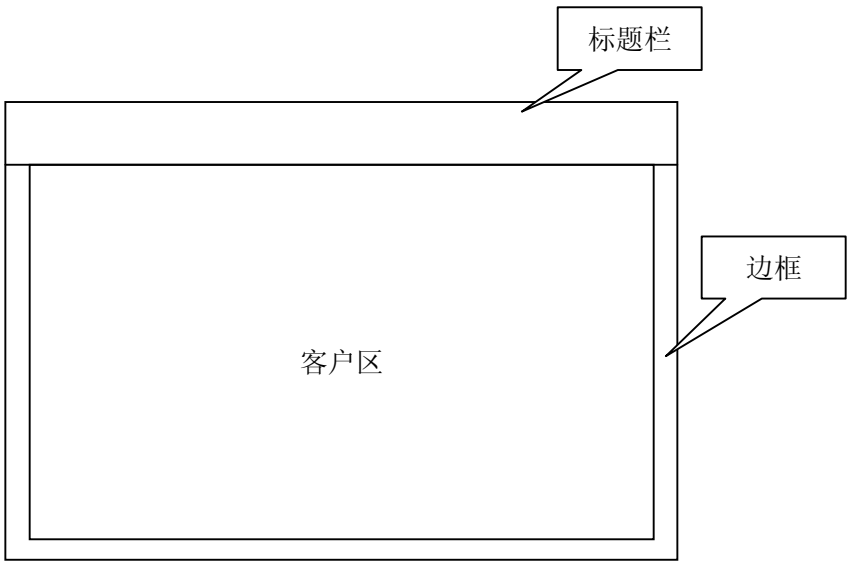

图 6-1 窗口的客户区与非客户区

在一个窗口中，标题栏、边框（或滚动条）就是窗口的非客户区，中间区域称为客户区。我们通过 GDI 函数在窗口上进行绘制时所使用的区域就是客户区。

6.2 坐标系统

在 LGUI 中，有三种坐标系统，相互之间是可以转换的。这三种坐标系统分别是：屏幕坐标、窗口坐标、客户坐标。其中屏幕坐标以屏幕左上角为坐标原点，水平方向为 X 值、垂直方向为 Y 值构成坐标系；窗口坐标以窗口左上角为坐标原点，水平方向为 X 值、垂直方向为 Y 值构成坐标系；客户坐标以窗口客户区左上角为坐标原点，水平方向为 X 值、垂直方向为 Y 值构成坐标系。如图 6-2 所示。

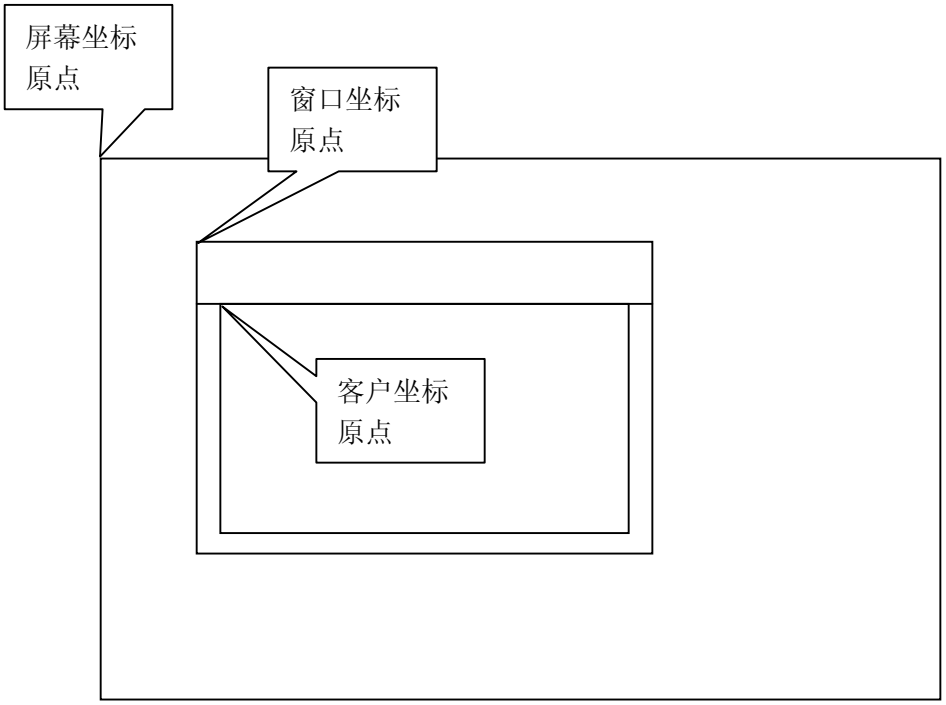

图 6-2 三种坐标系统示意

在 LGUI 实现时，经常需要在这几种坐标系之间转换，而对于应用开发者来讲，他只需根据客户坐标进行图形的绘制。

6.3 输出管理机制

在窗体的输出管理上，首先面临一个空间消耗与输出速度的平衡问题。按照基础软件的设计原则，系统提供的应当是机制，而不是策略。也就是说，选择牺牲速度、还是选择牺牲空间，应当是应用软件的事情。基础软件提供的机制应当满足牺牲速度、还是选择牺牲空间的两种可能性。LGUI 的输出管理满足这种原则。

在一般的 GUI 上，包括 MS Windows、QT、Microwindows 等系统都提供了 PAINT 重绘消息，如果你选择牺牲空间，则在 PAINT 中做复杂的绘图；如果你选择牺牲空间换取速度，你可以在内存设备上下文(MemoryDC)上绘制，然后整块输出到屏幕上。

LGUI 在这方面也为上层应用提供了选择。即在 Paint 消息处理时，可以选择直接绘制或是相反。若是直接绘制，则直接输出到屏幕，这时每一个点都需要进

行剪切域的判断，速度很慢，但好处是，一个窗体任何时候占用的内存资源是有限的；相反情况下，如果选择了非直接输出，则在开始输出之前，自动参照当前的设备上下文，创建一个内存设备上下文(MemoryDC)，然后，后续的绘图操作就定向到这个新创建的内存设备上下文。因为针对内存设备上下文的输出不必计算剪切域，所以速度是非常快的。当绘制完成后，再根据剪切域，分矩形块输出到屏幕上。这个过程可用图 6-3 表示：

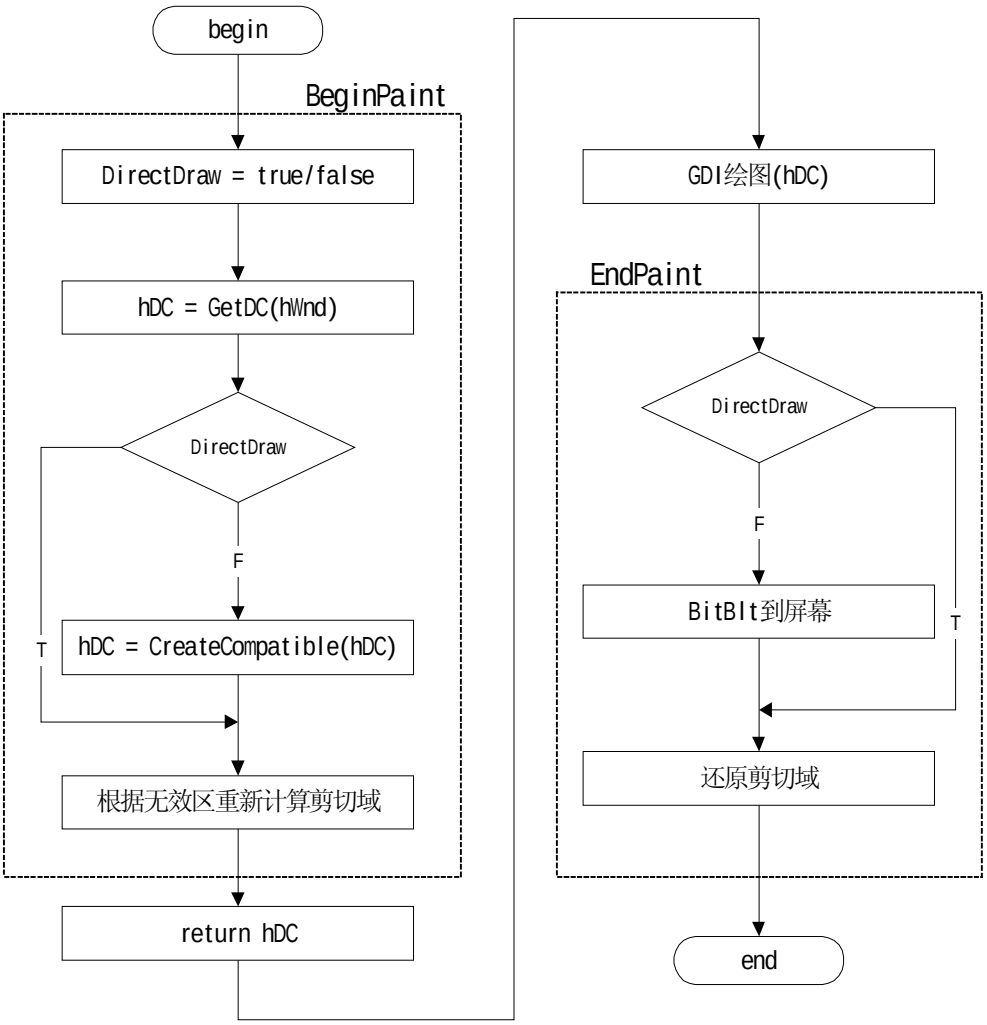

图 6-3　LGUI 的屏幕输出过程

其中 BeginPain、EndPaint 两个库函数对于上层应用来说是透明的，上层应用只需要在 Paint 消息处理的开始位置，设置一下 DirectDraw=true 或 DirectDraw=false。

在 BeginPaint 函数中，如果应用设置的 DirectDraw=false，则要创建一个新的内存型设备上下文，然后将该设备上下文句柄返回。否则，返回普通的设备上下文句柄。

在 BeginPaint 后，紧跟着是 GDI 的绘图，GDI 绘图函数根据设备上下文的类型自动将图形绘制到内存或屏幕。

在 EndPaint 函数中，根据设备上下文的类型，选择是否将内存中的图形整块输出到屏幕上去。

6.4 无效区

在窗口系统中，当某一窗口的某一部分需要重新输出时，会通过系统调用设置该区域无效的方式，使该区域得到重新绘制。

重新绘制消息可以立即执行，可以按照先来先服务的方式顺序执行，也可以等待其他消息执行完了以后再执行。由于窗口的输出占用更多的系统资源，所以为了保证系统对于其他更为紧急的消息的处理，一般要将绘制消息放在消息队列的后面，或者是在没有其他消息的情况下才处理绘制消息。

在 LGUI 中，每一个窗体（不包括控件）都有一个消息队列与之对应。为了使得系统具有较好的实时性，在 LGUI 中，将绘制消息单独形成一个队列，包括在整个窗体的消息队列中。

另外，在窗口的数据结构中，保存有当前窗口无效区的一个链表。窗口重新输出以后，这个链表会被清空。而每次调用 InvalidateRect 要求重新输出口体的一部分时，系统会在这个窗体的无效区链表中加入一个节点。

为什么要形成一个链表呢？因为如前面所述，对于绘制消息，可能会滞后处理。在此期间，可能会有多个绘制消息需要处理，这就需要通过链表的方式保存这些无效区域。

保存无效区的最终目的是为了在开始绘制的时候，形成真正的剪切域。在 LGUI 中，在 BeginPaint 时，窗体的剪切域将与无效区链表进行交运算，形成真正的剪切域。在 BeginPaint 后面的 GDI 函数的输出就是根据这个重新生成的剪切域进行输出，而在 EndPaint 的时候，剪切域将恢复到原来窗体的剪切域。这个过程可用图 6-4 表示：

为什么要将剪切域与无效区进行交运算生成新的剪切域呢？首先无效区必须

限制在剪切域的范围之内，否则，直接输出无效区就有可能破坏其他窗体；另外，如果对于任何一个窗体无效的消息，都根据剪切域全部输出一遍，则资源消耗大，而且窗口会有闪烁现象。无效域链表使得每次输出时，只将声明为无效的那些区域块重新输出到屏幕上。这也就是无效区管理的最终目的。

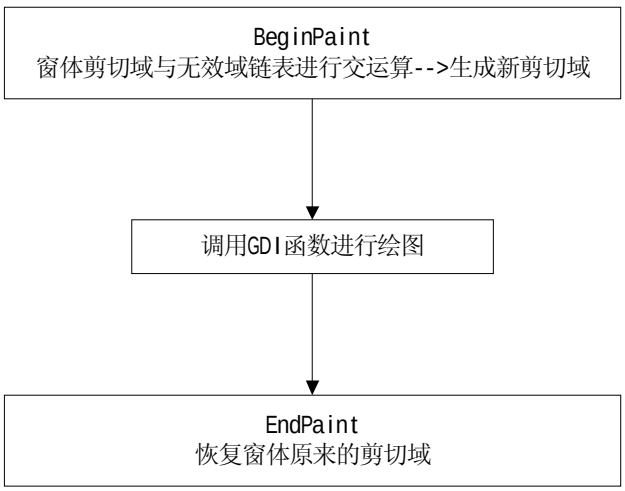

图 6-4　窗口 Paint 消息的处理过程

在这里，对于无效域链表的管理作进一步的说明：如果我们在程序运行的过程中由于某些原因需要重新输出窗口的某一块区域，例如原来被其他窗口覆盖的区域因为其他窗口销毁或移动使得被覆盖的区域需要重新输出，或者我们在窗口的客户区绘制了一些图形后，需要将该区域更新。而为了保证 GUI 系统有更高的实时性，我们设置对于窗口输出消息响应优先级为最低。这就使得可能会在一定时间会有多个对于窗口无效区的输出请求，所以就需要一个结构来保留这些无效区域。在 LGUI 中，我们通过一个链表来保存这些无效区，等到系统真正处理 Paint 消息时，会对一个窗口的无效区域一次性全部输出。我们知道，窗口还有一个表示剪切域的链表，只有这个链表内的矩形区域才是可以输出的，所以，输出无效区时，要将无效区与剪切域进行交运算，交集区域才是真正可以输出并且需要输出的区域。总而言之，剪切域是“可以”输出的区域， 而无效区是“需要”输出的区域，所以一次 Paint 消息的响应，要对两个区域进行交集运算。

第7章 DC 与 GDI 的设计与实现

7.1 LGUI 中设备上下文 DC 的描述

我们在 Windows 编程时，如果要操作一个窗口客户区，例如在客户区绘制图形，首先需要得到一个该窗口的设备上下文句柄，通过这个设备上下文句柄我们可以得到当前设备上下文的详细情况，如当前所使用的字体，字的前景色、背景色、所使用的画刷、所使用的线型等等。需要的时候可以将一些 GDI 对象选择到当前设备上下文中。例如，创建一种线型，并将该线型选择到当前设备上下文中，则此后针对该设备上下文的画线操作就会使用这一线型，画刷、字体等等的操作是类假的，那么在一个小型的窗口系统中，如何支持这种功能呢？

在 LGUI 中，设备上下文（Device Context，以下简称 DC）分为两类，一类是普通的，另一类是内存型的。对于内存型的 DC，在 DC 的描述结构中有一个指针 pData 指向对应的内存地址。LGUI 中虽然有位图 GDI 对象，但对于内存型 DC，仍然单独申请空间，大多数情况下位图对象用来保存该 DC 中频繁调用的位图内容。

在 LGUI 中，支持内存型设备上下文与普通型设备上下文目的是为了给使用者或于次开发者一种选择，如果需要输出的内存很少，例如只在窗口上画几个点，我们可以选择普通的设备上下文，这时，对于窗口的输出就会直接反应到屏幕上。相反情况下，如果有大量的绘图操作或者有成片区域需要绘制，我们一般会选择内存型的设备上下文。先将绘制的内容绘制到内存里，绘制完成后一次性将整个区域输出到屏幕上，这样速度会有极大提高。

在这里需要说明的是：LGUI 中内存型的设备上下文与窗口无效区的输出密切相关。根据图 6-3 所示，普通的设备上下文的输出直接针对屏幕，与窗口无效区没有关系。

LGUI 的 DC 的描述结构中存有当前的文本样式、各种 GDI 对象，包括：画笔、画刷、字体、位图对象的句柄。因为在该 DC 上进行输出时需要参考窗口的剪切域，所以设备上下文的结构中也包括对应窗口的句柄。

其中 DC 的数据结构中有一个重要的字段：pData。这个字段是一个指针。如果我们创建的设备上下文是一个内存型的设备上下文，则这个指针会指向一个大小与窗口大小一致的内存区域。例如在一个色彩深度为 3*8=24 位的系统，如果窗口的大小为非 100*100，则每次创建这个窗口的内存型设备上下文时，会申请一

个 100*100*24Byte 的内存区域。针对此设备上下文的输出都会先输出到这块内存区域。

```
typedef struct tagDC{
    int         iType;          //type(window/memory)
    COLORREF    crTextBkColor;  //text background color
    COLORREF    crTextColor;    //text color
    BOOL        isTextBkTrans ; //the text is transparent ?
    POINT       point;          //current point

    HPEN        hPen;
    HBRUSH      hBrush;
    HFONT       hFont;
    HBITMAP     hBitmap;
    void*       pData;          //memory device context only
    HWND        hWnd;           //handle of window
} DC;
typedef DC* HDC;
```

图 7-1　LGUI 中设备上下文的结构定义

创建普通型设备上下文的过程如图 7-2 所示：

从代码中可以看到，创建普通型设备上下文的过程就是创建一个指向 DC 结构的指针，同时创建结构中的 Pen、Brush、Font、Bitmap 等内容。然后返回这个指针，这个指针就是设备上下文的句柄。

创建内存型的设备上下文的代码较长，请参考源码，函数名称为：CreateCompatibleDC()。

GetDC、CreateCompatibleDC 这两个 API 函数在进行二次开发时可以显式进行调用，以创建普通的或内存型的设备上下文。实际上在任何一个窗口有消息处理函数框架中，对于 Paint 消息的处理已经包含了相关内容，如图 7-3 所示。当外理 Paint 消息时，首先会调用 BeginPaint 函数，在这个函数里，会根据参数 ps 的设置，确定是否创建内存型的设备上下文（请参考 BeginPaint 函数源代码）。然后返回一个设备上下文句柄。

```
ps. BPaintDirect = false;
hDC = BeginPaint(hWnd, &ps);
```

```
HDC GUIAPI
GetDC(
        HWND hWnd
)
{
      HDC hDC;
      BITMAP *pBitmap;
      hDC=malloc(sizeof(DC));
      if(!hDC)
            return NULL;
      memset(hDC,0,sizeof(DC));
      pBitmap=malloc(sizeof(BITMAP));
      if(!pBitmap)
            return NULL;

      memset(pBitmap,0,sizeof(BITMAP));

      hDC->iType=DC_TYPE_WIN;
      hDC->iCoordType = DC_COORDTYPE_CLIENT;
      hDC->hWnd=hWnd;
      hDC->isTextBkTrans     =true;
      hDC->crTextBkColor     =RGB(0xff, 0xff, 0xff);
      hDC->crTextColor  =RGB(0x00, 0x00, 0x00);
      hDC->hPen     =(HPEN)(GetStockObject(BLACK_PEN));
      hDC->hBrush =(HBRUSH)(GetStockObject(NULL_BRUSH));
      hDC->hFont    =(HFONT)(GetStockObject(SYS_DEFAULT_FONT));
      hDC->hBitmap=(HBITMAP)pBitmap;
      hDC->pData    =NULL;

      return hDC;
}
```

图 7-2　创建普通型的设备上下文

```
case LMSG_PAINT:
    ps.bPaintDirect=false;
    hDC=BeginPaint(hWnd, &ps);
    if(!hDC){
        return true;
    }

    //draw something

    EndPaint(hWnd, &ps);
    break;
```

图 7-3 Paint 消息处理代码

对于内存型的设备上下文，真正输出到屏幕上的操作是通过 EndPaint 来实现的，因为在 BeginPaint 时已经备份了窗口的剪切域，并将剪切域与无效域进行了交集操作生成输出时的有效剪切域。所以 EndPaint 就根据这个剪切域按矩形块输出到屏幕上，之后恢复剪切域释放设备上下文所占用的空间。

所以，在 LGUI 实现时，设备上下文不仅包含了当前所使用的线型、画刷、字体、字色等设置信息，还包含了一个用于缓冲输出的内存区域以支持屏幕的快速输出，这一点是非常重要的。

7.2 预定义 GDI 对象的实现

LGUI 预定义了常用的 GDI 对象，主要有：固定颜色、线型、宽度的画笔、固定颜色的画刷、字体等。

首先，这些 GDI 对象是由桌面进程在启动时创建的，并将他们全部保存在共享内存中，其他进程可以将这块共享内存映射到进程内部的地址空间中。

GDI 对象在共享内存中是通过以下的方式进行管理的：在固定的起始位置，创建一个所有 GDI 对象的索引，通过索引可得到 GDI 对象的指针。这个过程可用下图表示：

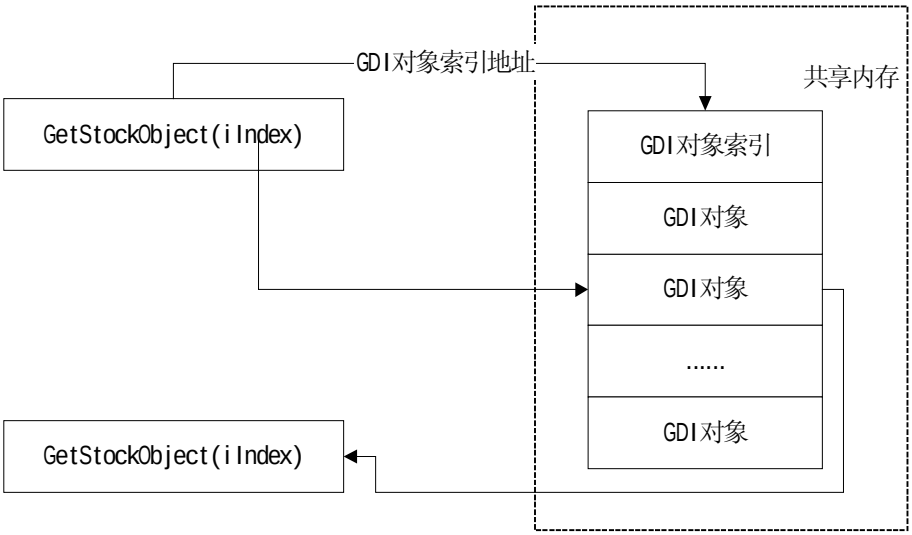

图 7-4 LGUI 预定义 GDI 对象的获取过程

在 LGUI 中，点阵字库也是放置在共享内存中，任何一个进程都可通过系统提供的 API 函数访问到字库中的内容，目前 LGUI 支持 24*24、16*16、14*14、12*12 三种点阵字体。

7.3 GDI 对象的描述结构及创建方法

GDI 对象的描述结构保持了与 Windows 的基本一致，如图 7-5 所示。

创建 GDI 对象的过程是比较简单的，给定一些参数，返回一个指向该 GDI 对象结构的指针。

目前 LGUI 中可以通过以下函数创建对应的 GDI 对象，包括：CreatePen、CreateBrush、CreateBitmap、CreateFont。

```
typedef struct tagPEN{
    GDITYPE     iGdiType;
    int         iPenStyle;      //
    int         iPenWidth;      // pen width
    COLORREF    crPenColor;     // pen color
} PEN;
typedef PEN* PPEN;

typedef struct tagBRUSH{
    GDITYPE     iGdiType;
    int         iBrushStyle;    // brush style
    int         iHatchStyle;    // hatch style
    COLORREF    crBrushColor;   // color value
} BRUSH;
typedef BRUSH* PBRUSH;

typedef struct tagFONT{
    GDITYPE     iGdiType;
    int iFontStyle;             //字体
    int iOffset;                //共享内存偏移地址
    FONTLIBHEADER FontLibHeader;//字库头结构
} FONT;
typedef FONT* PFONT;
```

图 7-5　LGUI 中 GDI 对象的描述结构

7.4 将 GDI 对象选入 DC 中

创建 GDI 对象以后，需要将这些 GDI 对象选入 DC 中，才能使定义的这些 GDI 对象在随后的输出中发挥作用。在 LGUI 中，这个功能通过函数 SelectObject 来实现。因为创建 DC 时其中会生成默认的 GDI 对象，所以 SelectObject 只需将 GDI 对象整体拷贝到 DC 描述结构中去即可。

7.5 GDI 函数的实现

```
//字库结构
typedef struct tagFONTLIBHEADER{
    int iSize;          //size of this struction
    int iAscWidth;      //width of Ascii character
    int iAscHeight;     //height of Ascii character
    int iAscBytes;      //bytes of a Ascii character used
    int iChnWidth;      //width of chinese character
    int iChnHeight;     //height of chinese character
    int iChnBytes;      //bytes of a chinese character used
    int iAscOffset;     //offset address of Ascii character
    int iChnSymOffset;  //offset address of chinese symbol
    int iChnOffset;     //offset address of chinese characte
} FONTLIBHEADER;
typedef FONTLIBHEADER* PFONTLIBHEADER;
```

图 7-6 LGUI 中的字库头结构

Linux FrameBuffer 本质上只是提供了对图形设备的硬件抽象，在开发者看来，FrameBuffer 是一块显示缓存，往显示缓存中写入特定格式的数据就意味着向屏幕输出内容。所以说 FrameBuffer 就是一块白板。例如对于初始化为 16 位色的 FrameBuffer 来说， FrameBuffer 中的两个字节代表屏幕上一个点，从上到下，从左至右，屏幕位置与内存地址是顺序的线性关系。

由于 LGUI 没有使用第三方图形库，直接基于 FrameBuffer，所以 GDI 函数的实现就有很大的工作量。在这方面，LGUI 在实现了基本的绘图函数 SetPixel 的基础上，移植了很多第三方图形源码，包括计算机图形学中的一些代码。主要包括以下几个方面。

基本的二维图形的绘制：包括画线、画圆、画椭圆、画矩形、画多边形。其中包括复杂的多边形的剪切与填充，这些函数最终都通过调用画点函数 SetPixel 来实现。因为这方面只是涉及到计算机图形学的一些基本算法，不再多述。

文本的输出：LGUI 中，系统字库的点阵信息由服务器进程初始化时保存到共享内存中，各个进程包括服务器进程都可以访问得到其中的内容。LGUI 中自己定义了点阵字库格式，在字库的字库头部分，保存有固定长度的字库参数，如图 3-11

所示。如果要在某个 DC 上输出一个英文或汉字字符，因为在该 DC 中保存有当前选入该 DC 的字体信息，其中就有这个头结构，所以通过公式计算就可以得到相应点阵的地址，根据点阵在就可以在该 DC 上进行输出。

图形文件的支持：就是指图形文件的解码，目前支持 24 位色及 8 位色的 BMP 文件、JPEG 文件。其中 24 色及 8 位色的 BMP 文件格式比较简单，而 JPEG 文件的解码则是通过移植第三方开源的代码来完成。

第8章 LGUI 应用开发模式

一般意义上，人们将 GUI 系统定位为一个中间件系统。中间件（middleware）是基础软件的一大类，属于可复用软件的范畴。顾名思义，中间件处于操作系统软件与用户的应用软件的中间。中间件在操作系统、网络和数据库之上，应用软件的下层，总的作用是为处于自己上层的应用软件提供运行与开发的环境，帮助用户灵活、高效地开发和集成复杂的应用软件。

LGUI 面向嵌入式 Linux，提供了一个桌面环境，包括桌面的图标、状态栏、键盘、鼠标等，同时提供二次开发的 API 函数库以及对二次开发模式的定义。

8.1 应用开发的模式

毋庸讳言，Linux 是操作系统的后起之秀，Unix 也不曾占领过桌面计算的市场，而对 Windows 熟悉的程序员在我们的周围占有更多的比例。所以在二次开发方面，LGUI 选择了力争与 Windows 兼容的策略。

首先是 API 函数的设计，API 函数设计成与 Windows API 函数基本相同的函数，如 CreateWindow、ShowWindow 之类。函数的功能也与 Windows API 函数基本一致。

当然，在保证 API 函数的定义与功能实现与 Windows 基本一致之外，更为重要的是二次开发的应用程序的模式。因为基于消息驱动的程序与过程式程序的不同必然要求有一个消息处理的模式。而对于过程式程序提供 API 函数实际上涉及不到二次开发程序的模式问题。LGUI 应用程序的模式如图 3-5 所示：

其实这种程序结构正好是 Win32 程序的模式，以注册窗口类开始，然后创建窗口、显示窗口，然后进行消息循环，从消息队列中取消息、分发消息。

如果应用开发人员遵守这个开发模式的要求，则他在不了解 LGUI 系统结构原理的前提下也是完全能够开发出基于 LGUI 的应用程序，就像 Windows 程序员也许根本不了解 Windows 的窗口关系、消息队列等的原理与实现方式，但仍然可以方便地开发应用程序。只需按照系统的要求发送消息、处理消息，系统就会正常地运行起来。

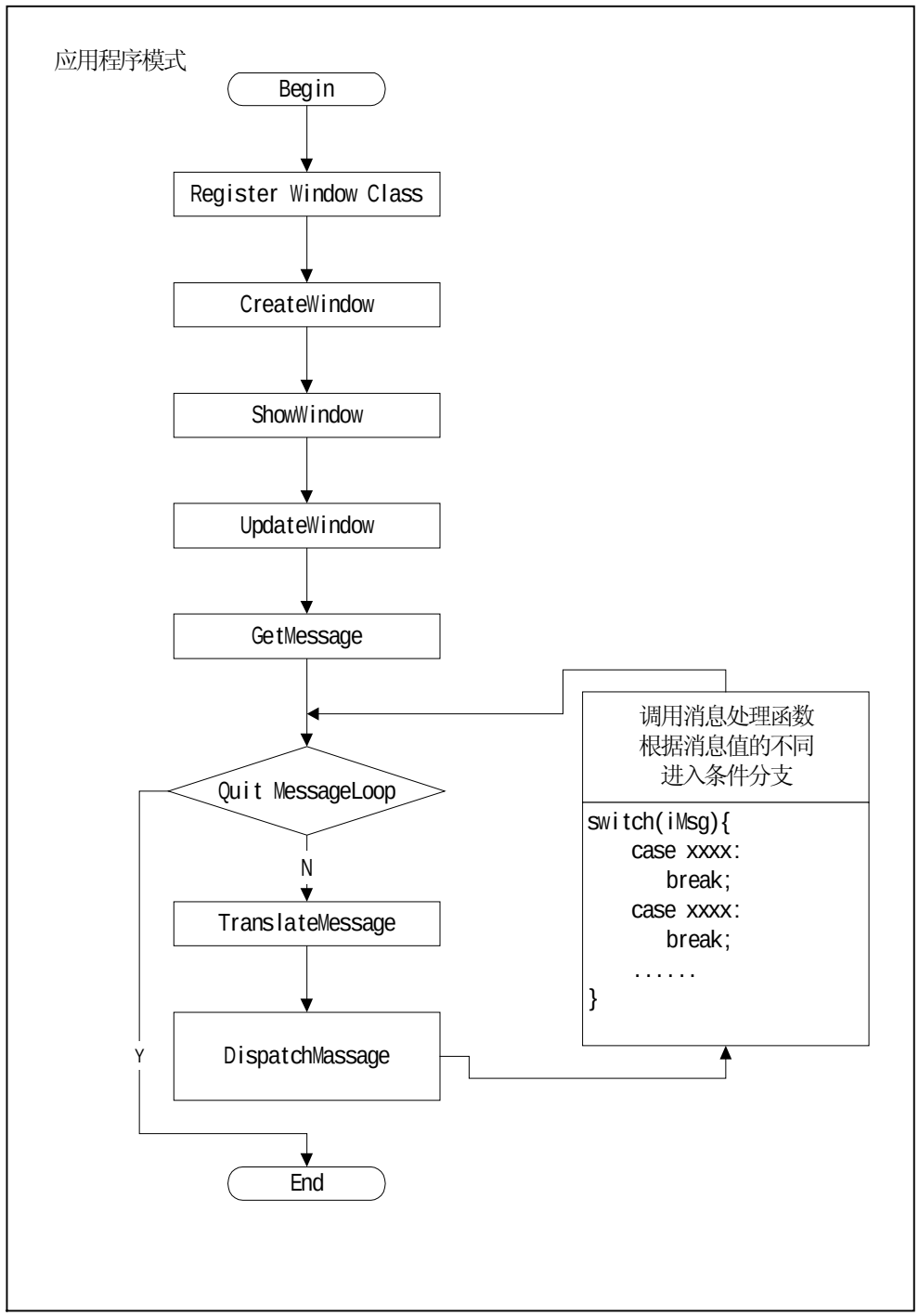

图 8-1 应用程序的模式

这个应用程序模式的基础是窗体的注册机制，与一个控件的注册机制一样，应用程序需要创建自己的窗口类并进行注册。窗口的注册最主要目的是注册窗口的消息处理函数（当然还有其他一些任务），这样系统在分发这个窗体的消息时，就知道该调用什么函数对这个消息进行处理。

这个过程如图 8-2 所示：

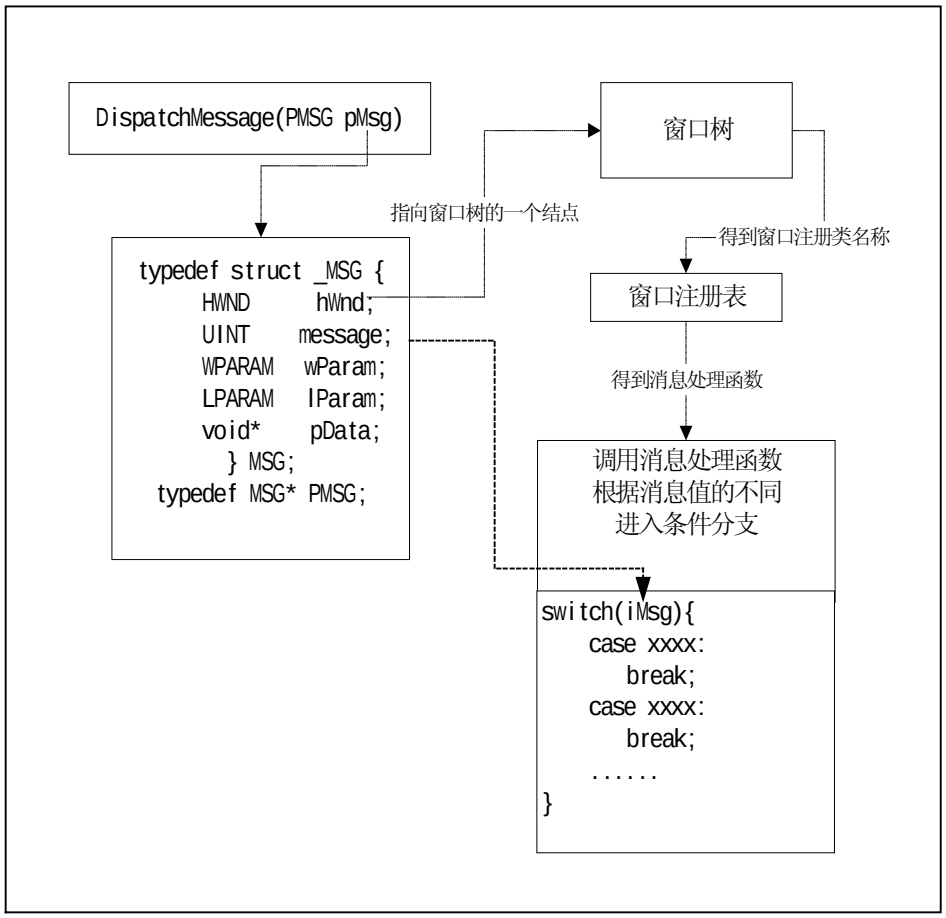

图 8-2　LGUI 应用程序消息分发过程

这个过程就是：根据消息对应的窗口句柄，从窗口树中得到窗口的类名称，根据类名称从窗口注册表中得到消息处理函数，然后用消息值调用消息处理函数。

其中，窗口树与窗口注册表在前面已经有过叙述，所谓窗口树，是由窗口的主子关系形成的一个树状表；窗口注册表是窗口向系统注册的过程中形成的一个

表，在 LGUI 中，是使用 Hash 表进行管理的。

其中由指向兄弟窗口的指针与指向子窗口的指针以及指向控件的指针形成一个树状的窗口集合，而任何一个进程包括服务器与客户进程都会拥有一个自己的窗口树。

从这个结构中我们可以看出，通过窗口指针，我们可以得到所有与窗口相关的信息，包括：窗口类、剪切域、无效域、消息队列等等。

窗口类是一个字符串，通过在窗口注册表中进行搜索，我们便可得到窗口的注册信息，注册信息中主要包括消息处理函数。因为结构中定义的是一个函数指针，赋值时会给这个字段赋一个函数名，实际上这就是通常所说的回调函数。

LGUI 应用程序的格式是固定的，应用开发者的工作便主要集中在消息处理函数上，包括对各种系统消息以及自定义消息的响应。

8.2 开发调试方法

LGUI 在设计时考虑到了调试的问题，由于 LGUI 是使用 Domain Socket 来连接客户端与服务器端，所以在 Linux 的 PC 环境下，可以用多个控制台进行调试。例如：用第一个控制台启动桌面进程，然后切换到第二个控制台，使用 gdb 等工具单步执行应用程序，可以比较方便地发现程序中的 Bug。这种调试方法比普通的通过 print 的方法要方便很多。

8.3 基于 LGUI 的应用程序简例

基于 LGUI 的地图引擎及 GPS 定位导航系统：

目前基于 LGUI 的系统方案主要有：车载多媒体娱乐与 GPS 定位导航系统、手持多媒体娱乐与 GPS 定位导航系统以及家庭多媒体中心，其中与定位导航相关的系统中都有一个基于 LGUI 的关键应用：地图引擎 LGIS。下面将 LGIS 一些关键算法及 LGUI 对 LGIS 的功能支持进行说明。

地图引擎的开发有一些较为复杂的算法，主要包括地图对象的空间索引问题、多边形的剪切问题、地图数据的转换问题。

其中，空间索引是地图引擎的核心，在 LGIS 中，空间索引是通过 R-tree 的索引算法实现的。R-tree 索引依据对象的外接矩形建立类 B+树的索引表。

在地图引擎中定义了比较复杂的数据结构，以描述当前工作空间、图层以及图形对象的属性。LGIS 中定义的图形对象主要包括：文本、点、线、面、折线、

多边形、矩形、椭圆等。地图引擎中目前尚未实现对象之间的空间关系。

地图在显示时，系统对每一个图层中的所有地图对象首先会进行一次查询，如果与当前地图窗口相交，则要进行输出，否则不必输出。由于使用了高效的 R-tree 索引算法，而不是简单地逐个图形对象进行比较，所以这个查询过程是非常高效的。

地图引擎加上 GPS 定位设备就构成一个 GPS 导航系统，这个系统的结构如图 7-3 所示：

从图中可以看出，导航系统中主机系统与 GPS 模组通过标准的 RS232 口进行连接。地图引擎在初始化 GPS 设备以后，通过创建一个串口监视线程来接收 GPS 设备返回的数据。当接收的内容根据 GPS 的数据格式定义确定为有效的一帧数据后，就向地图引擎发送消息，地图引擎进行显示、移动等相应的处理。

LGUI 对 LGIS 的功能支持：

(1) LGIS 启动与退出过程：

LGUI 的应用程序是动态可安装的，一个应用程序的安装包包括有应用程序描述文件：xxx.desktop、桌面上显示的图标文件 xxx.bmp、二进制应用程序 xxx.bin。应用程序安装后，在 LGUI 桌面上会创建一个图标。点击该图标可启动对应的应用程序。

当用户点击 LGIS 图标后，LGIS 即可启动，显示主窗口，如果用户点击主窗口右上角的关闭按钮，主窗口就会收到 destroy 消息，主窗口退出。

(2) 设备上下文(DC)与图形设备接口(GDI)：

在 LGIS 中，任何一个图层的都包含有一个该图层的对象链表，而其中的每一个节点都详细描述了该图形对象的有关信息，例如对一个多边形来说，包含的信息有：多边形各节点坐标、边界画笔、填充画刷；对于一个文本对象，其描述信息则包含：文本内容、文本位置、字体。当 LGIS 主窗口的内容需要重绘时，首先要创建一个内存型的设备上下文，然后根据 R-tree 查找到的图形对象经过地理坐标到屏幕坐标的转换后调用 LGUI 的 GDI 函数进行绘制。由于图形对象种类多、且同类图形对象其线型、填充类型均可能不同，所以调用 GDI 函数进行绘制之前，首先根据对象的描述信息创建画笔与画刷或者字体，并将这些 GDI 对象选入设备上下文，然后调用 GDI 函数进行绘制。当然绘制过程中还包括线与多边形的剪切以及多边形的填充，这些都是调用 LGUI 的 GDI 函数来完成的。绘制完成后再将这个内存型设备上下文的内容整块输出到屏幕上，由于使用了内存型设备上下文，输出的速度是很快的。

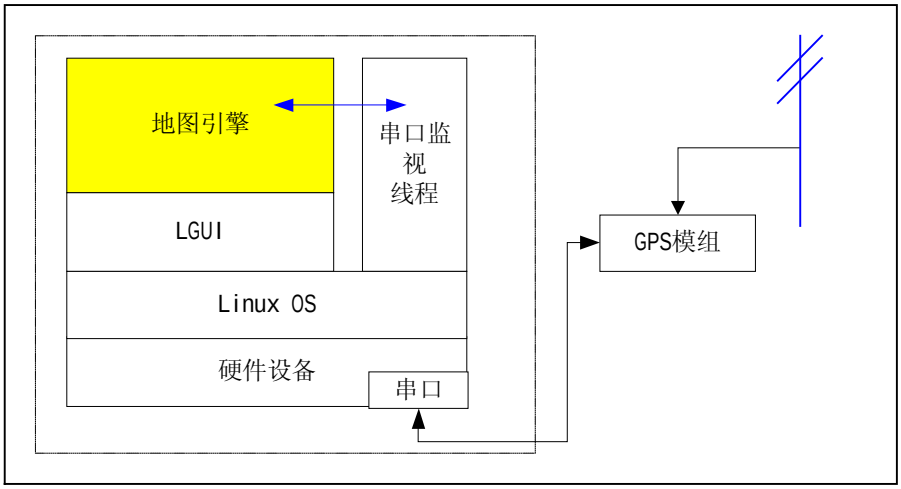

图 8-3 基于 LGUI 的 GPS 导航系统

(3) LGIS 中子窗口及控件的使用

LGIS 主窗口上的控件包括静态文本框与按钮，静态文本框主要显示当前坐标位置。在移植到 SigmaDesign 8510 平台的 LGIS 中，其中还包括编辑文本框，用于输入地址，以进行路径的查找与规划。还有可动态显示与隐藏的子窗口用于显示属性信息。

(4) LGIS 中的消息队列与消息循环：

LGIS 的主窗口首先会建立一个消息队列，并在显示完主窗口后进入消息循环，而且 LGIS 在启动时还会创建一个单独的线程来监视 GPS 信息。当用户按下主窗口上的按钮时，按钮会以其 ID 为参数发送通知消息到其父窗口，在主窗口的消息处理函数中就可对该事件进行处理。另外监视 GPS 信息的线程在收到 GPS 信号后，会将收到的内容进行解析，然后生成相应的消息发送到 LGIS 主窗口的消息队列中去，主窗口在收到这个消息后对该消息进行处理。

LGUI 运行过程截图

后记——LGUI 开发的一些体会

当前，手持设备、智能终端、信息家电等等嵌入式系统正得到蓬勃的发展。现在出现的市场只是冰山一角。就像十多年前人们的梦想已变成活生生的现实一样，一个网络化、智能化的生活形态已经越来越清晰地展现在了我们的面前。而嵌入式系统的发展，正在为这个大潮起着推波助澜的作用。

与面向桌面计算的 PC 机不同，嵌入式系统的最大特色是“个性化”。没有固定模式的嵌入式系统，尤其是用户界面。这注定了“定制”将会无处不在。这也使得嵌入式系统的市场将在很大程序上会“碎块化”，没有哪家公司可以统一市场，形成事实上的标准。

嵌入式系统定制化的特点本身就要求系统是开放的，可量身定做的，这也就是为什么有很多公司从 WinCE 转向 Linux 的原因。另外，Linux 的低成本特点，可大大降低最终嵌入式系统的成本。

嵌入式系统的蓬勃发展，使得很多业内的大公司推出了专门针对嵌入式设备的专用芯片。由于 linux 开放源码、易于移植，并且由于 linux 目前在嵌入式系统中所占据的举足轻重的地位，几乎所有的芯片制造商在推出新的芯片时，都会投入大量的人力、物力来移植 Linux，并提供丰富的开发工具。有些公司甚至会提供包括 CPU、SDRAM、FLASH、LCD、TouchPanel、KeyBoard 在内的功能齐全的测试板和所有这个外部设备的基于 Linux 的驱动程序。这使得嵌入式 Linux 的 GUI 系统成为一个完整产品构架中非常重要的环节，如果在 GUI 基础上进一步构造面向某一行业的应用，为行业用户提供解决方案，将为嵌入式 Linux 的发展起到一定的推动作用。

在开发的过程中，我认为开发一个中等规模的系统，首先需要有全局意识，但除此之上，对一个完全陌生的系统，并不完全是“需求——设计——编码——调试”的过程。需要不断地反复，需要不断地通过编写代码来证明你的设计是可以实现的，且该种实现方法的非功能指标是满足设计要求的，然后再重新调整最初的框架。这正是一个软件工程中所讲的“迭代”过程。

“唯一满足工程设计标准的文档，就是源代码清单”，对于一个需求明确、功能复杂的系统，可以通过一些图表展示系统设计，但经过验证的框架性的代码也许才是最好的设计文档。我认为：在对系统功能、实现的环境都了解有限的情况下，凭空进行设计，绝大多数情况下是没有任何意义的。设计者必须对系统需求与实现环境有非常深入的了解，而且应该通过书写代码来验证自己的设计，并最终将这些代码作为设计的结果来提交，这才是有意义的设计。

在有些情况下，甚至项目的总体目标是模糊的，只有通过多次的“迭代”，才能逐渐逼近目标，同时由于在开发的过程中对项目的了解越来越多，对项目目标的把握也越来越清晰，从而使得项目的实现更加接近于项目的根本需求。

另外，做研究开发，借鉴别人的思想是很重要的，不能闭门造车。尤其是作为软件开发者。现在的软件技术一日千里，不学习别人的先进技术，借鉴别人的先进的思想，只凭个人的想象，软件完成之日也许就是丢弃之时。充分利用网络资源是一个软件工作者的好习惯，在 Internet 上，遍布着各种各样的资源，学会挖掘这种资源，将在很大程度上推动项目进展。

最后，如果这本书能对您开发嵌入式 Linux 的 GUI 系统有所帮助，那我将感到非常高兴！

——李玉东

参考文献

[1].董士海：用户界面的今天和明天 计算机世界 1997 年第 7 期

[2].Anand K Santhanam,Vishal kulmarni,嵌入式设备上的 Linux 系统开发，http://www-900.ibm.com/developerWorks/cn/linux/embed/embdev/index.shtml

[3].Microwindows, http://microwindows.censoft.com，2004-8

[4].OpenGUI, http://www.tutok.sk/fastgl/，2004-8

[5].Qt/Embedded, http://www.trolltech.com/，2004-8

[6].MiniGUI, http://www.minigui.org/，2004-8

[7].SVGALIB, http://www.svgalib.org/，2004-8

[8].GGI, http://www.ggi-project.org/，2004-8

[9].王悦，岳玮宁，王衡，汪国平，董士海，手持移动计算中的多通道交互，软件学报 2005,16(1):29-36

[10].董士海，王坚，戴国忠，人机交互和多通道用户界面，科学出版社

[11].王衡，董士海，汪国平，走向和谐自然的人机交互，北京大学 2003 年信息科学技术学院学术研讨会论文集 289-296，2003

[12].Charles Petzold（美）， Windows 程序设计，北京大学出版社