

北京亚嵌教育研究中心

多媒体项目实训营

中级阶段--理论教材

## 北京亚嵌（AKAE）教育研究中心简介

北京亚嵌（AKAE）教育研究中心是国内最早从事嵌入式技术教育培训的专业研究机构。亚嵌（AKAE）发源于 1998 年在清华大学成立的 AKA 组织 (www.aka.org.cn)，秉承 AKA “自由、协作、创造”的技术理念，始终致力于嵌入式技术的人才培养、项目研发、技术推广和咨询服务工作。

中心具有博士以上学历 12 人，硕士学历近 20 人，同时汇聚了 30 多名来自于清华、北大、中科院等科研院校的一线研发主力作为专家师资团队，聘请了龙芯、红旗、中兴、华为、大唐电信、IBM、AMD 等各大公司的技术负责人和项目经理作为中心的技术顾问。

嵌入式技术凝聚了计算机和信息技术精华，嵌入式技术人才是具备透视计算机和信息技术奥秘的高级专业人才。作为国内最早开展嵌入式技术的教育研究机构，亚嵌（AKAE）将嵌入式技术人才培养做为自己的终身事业，把最新的技术发展和行业需求结合起来，为广大学员提供最完善、最有效的技术咨询培训服务。凭着多年技术积累和教学经验，凭着雄厚的师资、优越的课程体系和严格的教学管理，亚嵌学员的就业率一直保持在 100%。亚嵌已为国内外各大公司和科研院所培养嵌入式技术人才上千名，真正成为国内嵌入式技术高级专业人才的摇篮。

多年来，亚嵌教育得到了中科院软件所、ARM 中国、华为通信、神州龙芯、联想集团、飞漫软件、中星微电子、中科红旗等公司以及清华大学信息学院、北航出版社、北航软件学院、中国软件行业协会嵌入式分会、广东省技术推广站等科研院所、政府部门的大力支持。

亚嵌走过了近五年卓有成绩的发展道路，坚持不懈地努力架设一座高校教育与社会需要之间信息沟通、技术服务和人才培养的桥梁。本着“自由、协作、创造”的技术理念，亚嵌汇聚了一批年轻有朝气的师资团队，将“传道、授业、解惑”作为自己的神圣职责，在传授知识的同时，引导学员树立正确的价值观和高尚的职业道德，使得每一位来到亚嵌学习的学员能够在这里亲身感受到一种奋发向上的力量，一种无所畏惧的勇气，一种积极乐观的精神和一种自强不息的信念。

展望未来，亚嵌有着自己独特的使命和愿景--成为“中国嵌入式技术的黄埔军校”。黄埔军校在革命年代“构建了无数个生长着崇高追求的精神世界，塑造了无数个怀抱着伟大使命的高尚人格”，这是黄埔军校的成功所在，也是亚嵌教育的终极理想。

只有胸怀崇高追求才能放弃个人享乐，只有肩负伟大使命才能成为国家栋梁！任重道远，矢志不移，永不言败，“黄埔精神”将会在每一期亚嵌学员的身上发扬光大！

第 1 章 数码相框项目概述 .....	8
1、何谓数码相框? .....	8
1.1 数码相框产品起源 .....	8
1.2 数码相框产品发展现状 .....	8
1.3 数码相框产品发展前景 .....	9
2、哪里需要图形处理-数码相框技术 .....	10
2.1 图形处理应用领域 .....	10
2.2 数码相框相关技术岗位需求 .....	10
3、数码相框项目成果演示 .....	12
第 2 章 Framebuffer 编程 .....	15
1、文件 IO .....	15
1.1 汇编程序的 Hello world .....	15
1.2 C 标准 I/O 库函数与 Unbuffered I/O 函数 .....	17
1.3 open/close .....	19
1.4 read/write .....	23
1.5 lseek .....	30
1.6 fcntl .....	31
1.8 ioctl .....	36
1.9 mmap .....	37
2、FrameBuffer 编程的基础知识 .....	41
2.1 FrameBuffer 工作原理 .....	41
2.2 FrameBuffer 相关数据结构分析 .....	43
2.3 Framebuffer 的使用 .....	44
第 3 章 鼠标的工作原理 .....	49
0、简介 .....	49
1、鼠标的设备文件 .....	49
2、辅助代码声明 .....	50
3、标准鼠标的数据格式 .....	51
4、鼠标的绘制 .....	53
5、完整的代码实例 .....	54
第 4 章 五子棋算法 .....	60
1 导言 .....	60
1.1 电脑五子棋简介 .....	60
1.2 设计思路总介绍 .....	60
1.3 本文架构 .....	61
2 数据结构 .....	61
2.1 棋局的表示 .....	61
3 涉及算法 .....	62
3.1 五子棋搜索树种的极大极小搜索 .....	62
3.2 $\alpha$ - $\beta$ 剪枝 .....	64
3.3 优化估值函数 .....	66
3.4 禁手特征计算 .....	67
4 结论及展望未来 .....	67
4.1 总结 .....	67

4.2 未来展望.....	67
第 5 章 图像生成 .....	69
1、 图形图像知识基础.....	69
1.1 图形图像基础学习资料.....	69
2、直线的生成算法.....	74
2.1 简介.....	74
2.2 直线的 Bresenham 算法.....	74
3、圆的生成算法.....	78
3.1 Bresenham 画圆算法描述.....	78
4、常见的图片格式.....	80
4.1 jpeg 图片格式.....	80
4.2 bmp 图片格式.....	81
4.3 png 图片格式.....	82
5. libjpeg 库简介.....	83
第 6 章 字体显示 .....	86
1、 漫谈字符集和编码.....	86
2、 utf-8 编码详解 .....	91
2.1 什么是 UCS 和 ISO 10646? .....	91
2.2 什么是组合字符? .....	92
2.3 什么是 UCS 实现级别? .....	93
2.4 什么是 Unicode? .....	94
2.5 什么是 UTF-8? .....	95
2.6 什么编程语言支持 Unicode? .....	96
2.7 在 Linux 下该如何使用 Unicode? .....	97
2.7 我该如何修改我的软件? .....	98
2.8 我怎样才能得到 UTF-8 版本的 xterm? .....	101
2.9 我在哪儿能找到 ISO 10646-1 X11 字体? .....	103
2.10 我怎样才能找出一个 X 字体里有哪些字形? .....	105
2.11 与 UTF-8 终端模拟器相关的问题是什么? .....	105
2.12 已经有哪些支持 UTF-8 的应用程序了? .....	106
2.13 X11 的剪切与粘贴工作在 UTF-8 时是如何完成的? .....	107
2.14 更多参考 .....	107
第 7 章 触摸屏应用 .....	110
1、 触摸屏工作原理.....	110
1.1 触摸屏的基本原理.....	110
1.2 触摸屏的控制实现.....	111
2、电阻式触摸屏.....	116
3、电容式触摸屏.....	122
4、触摸屏日常维护.....	123
4.1 触摸屏的内部结构以及工作原理.....	123
4.2 触摸屏的分类以及日常使用维护.....	123
4.3 触摸屏的日常维护重点.....	124
4.4 针对触摸屏的分类也有不同的养护方式: .....	124
5. iphone 触摸屏和 5800 触摸屏工作原理.....	125

第 8 章 多进程编程 .....	133
1、 引言.....	133
2、环境变量.....	134
3、进程控制.....	139
3.1. fork 函数 .....	139
3.2. exec 函数 .....	143
4、进程间通讯.....	151
4.1. 管道.....	152
4.2. 其它 IPC 机制 .....	156
第 9 章 多线程编程 .....	159
1. 线程的概念.....	159
2. 线程控制.....	160
2.1. 创建线程.....	160
2.2. 终止线程.....	162
3. 线程间同步.....	166
3.1. mutex.....	166
3.2. Condition Variable .....	173
3.3. Semaphore.....	176
3.4. 其它线程间同步机制.....	179
4. 编程练习.....	179
第 10 章 音乐播放 .....	182
1、 数字音频格式简介 .....	182
2、 mp3 编码技术.....	186
2.1. mp3 .....	186
2.2.掩蔽效应.....	187
2.3 提前说 mp3 的版权控制。 .....	188
2.4 mpeg audio layer .....	188
3、 mp3 编码格式.....	188
附录 A 重要参考资料.....	192
HTML 语法 .....	192
1、img 标签 .....	192
2、img 标签语法 .....	192
3、示例.....	193
4、网页设计中常见的图像格式简介.....	193
图片压缩知识基础.....	194
1、数据压缩引言.....	194
2、JPEG 算法的主要计算步骤.....	195
TTF 文件分析与使用.....	198
1 引言.....	198
2 TTF 字体设计思想.....	199
3 TTF 字体结构.....	199
4 TTF 字体的创建.....	199
5 TTF 字体在面向对象程序中的引用 .....	200

## 为什么我们需要项目实训

### 背景

时光如梭，亚嵌嵌入式培训中心已走过了 6 年的成功发展之路。遥想当年，独辟嵌入式教育之先河，逐步摸索实践，时至今日已形成一套成熟的教育培训模式。亚嵌--这座中国嵌入式教育的摇篮，已经培养出了数以千计的嵌入式行业人才，他们作为中国嵌入式行业的中坚力量正发挥着越来越重要的作用。

2009 年是亚嵌飞速发展的一年，在培训模式、课程改革、教学手段方面都取得了丰硕的成果，更是凭借其在嵌入式教育领域的业绩与良好口碑，成为工业和信息化部设立的国家信息技术紧缺人才培养工程（NITE）首个嵌入式人才培养实训基地并在年底荣膺“网易 2009 中国教育年度大选-十佳 IT 培训机构”称号。

传统的教学模式“重理论，轻技能”，导致学生在知识、能力和素质结构方面缺乏特点，尤其是缺乏专业人才应有的比较强的操作和实践能力，与实际企业对人才的需求存在巨大差距。传统的教学模式中，缺少对市场动向的足够关注，较难将现有的专业知识与就业需求结合起来，学生对于今后的职业发展没有明确的方向和规划。

针对传统教学模式的特点，在国家培养创新人才的社会背景下，改革教育模式已势在必行，亚嵌项目实训应运而生。

### 特点和目标

亚嵌项目实训将以理论知识为基础、紧扣嵌入式市场脉搏以实际项目为依托、专业规范为目标，以更接近于企业工作模式的方法，着重培养学员分析问题、解决问题的能力。我们将突破固有的“教师教、学生练”的教学模式，变学员被动学习为主动思考。

项目实训过程中，将提倡学员大胆质疑、多向思考、科学想象，通过项目分析、课程讨论等形式，培养学员的学习兴趣，让其参与到创新的环境中来，使学生形成探求创新的心理愿望和性格特征，形成一种以创新精神汲取知识、应用知识的习惯，并积极培养学员清晰思维、表达和写作的能力；收集、甄别、综合、评价资料信息的能力；独立思考与人合作的能力；勇于创造独立工作的能力。

## 项目实训包含哪些内容？

从以上培养目标出发，我们特编写项目实训学员指导手册。手册中明晰了项目内容、项目任务、目标方向，改变了传统教材中“知识要点+模拟练习”的内容，手册只指明了项目内容、项目目标和评价标准，鼓励学员在明确目标的前提下，积极思考，搜集资料，立足于现有的知识结构，分析问题，大胆尝试，最终通过个人及团队的智慧完成项目任务。

此外，在手册中我们列出了完成项目必需的知识要点并在参考资料部分进行了知识要点的讲解，通过教师必要的帮助和学员自身的努力，在掌握基本知识的基础上，完成实际项目的开发。

我们力求使本手册成为学员完成实训项目的路线图，通过本手册的指引使学员能够在各个问题阵地上插上自己胜利的小红旗并最终挖掘出项目成果宝藏。

## 学生和老师在项目实训中的角色定位？

在项目实训中，教师将更加深入到学员中来，成为项目实训这场“寻宝盛宴”的组织者和参与者，转变原有重在“解惑”的职业定位，而更加注重“传道、授业”，关注师生间的教学互动、思想交流，引导学员更有目的、更自觉地发现问题并提出解决问题的方法，激活学员的创造性潜能和创新的主动性，增强他们对新知识的渴望和敏感性。

学员也将变原来的被动学习为主动思考，使自己成为学习过程中的主导，积极思考、勇于提问，提高自身分析问题、解决问题的能力。

学员应该对技术抱有积极的态度及热情，在技术探索的过程中不免会遇到很多困难，应该能够刻苦钻研、积极探索，自身激发出的对技术孜孜以求的态度，将会成为学员完成项目的最大动力。

## 第 1 章 数码相框概述

---

课程内容:

- ✧ 何谓数码相框
- ✧ 哪里需要图形处理-数码相框相关技术?
- ✧ 数码相框项目成果演示



# 第 1 章 数码相框项目概述

## 1、何谓数码相框？

### 1.1 数码相框产品起源

随着信息化，智能化，网络化的发展，嵌入式系统的广泛应用已经渗入到我们日常生活的各个方面。在手机、MP3、PDA、数码相机、电视机，甚至电饭锅、手表里都有嵌入式系统的身影，工业自动化控制、仪器仪表、汽车、航空航天等领域更是嵌入式系统的天下。据估计，每年全球嵌入式系统带来的相关工业产值已超过 1 万亿美元。随着多功能手机、便携式多媒体播放机、数码相机、HDTV 和机顶盒等新兴产品逐渐获得市场的认可，嵌入式系统的市场正在以每年 30% 的速度递增。

数码相框正是这样一种嵌入式技术应用的代表产品。数码相框由概念型产品进入市场至今，已经经历了 5、6 个年头。作为伴随数码相机及互连网不断飞速发展的衍生产物，在今天也已经被愈来愈多的普通消费者所接受。

### 1.2 数码相框产品发展现状

数码相框产品是 2001 年开始出现的，但由于当时消费者的接受度及价格过高的因素，使这一市场一直到 2003 年都很低迷。随着主要器件价格的下降，数码相框的价格也逐步下降，市场在 2004 年开始有了起色，尤其在 2005 年，数码相框产品开始在欧美热销，2006 年、2007 年产品销量均有大幅增长，据预测，到 2011 年出货量将达到 4000 万台。

在中国，2006 年以前，中国生产的数码相框绝大多数出口国外。2005 年底 Philips 率先将数码相框在中国推广，在礼品市场上取得了一些成绩，但由于销售价格较高，销售量仅有 1.7 万台。2006 年下半年，开始有更多的国内厂商在中国市场推出数码相框，因而也带动了此产品价格的下降，到 2007 年下半年，业界才感到这个一直处于培育期的市场，开始了真正的起飞，这得益于对数码相框产品认知度的提高、价格的下滑和需求量的提高。

从 IT 厂商来看，仅 2007 年下半年以来，就有惠普、三星、优派、AOC、明基、柯达、长城等众多新军加入数码相框阵营。其中，巨头惠普 2007 年 7 月底在美国宣布进入数码相框市场，2007 年数码相框出货量设定为 50 万台。除新军外，数码相框老牌劲旅的出货量表现也令业界振奋，如飞利浦 2006 年数码相框

出货量达到 50 万台，而 2007 年上半年出货量已达去年总和，2007 年全年出货量达 150 万台。

### 1.3 数码相框产品发展前景

随着嵌入式技术的不断发展以及数码相框市场的不断拓展，2008 年及今后几年将为处在数码相框产品供应链的各企业带来巨大的商机。未来几年，数码相框的市场将处在逐渐走向成熟期的阶段，其产销量和市场需求依然将保持大幅度的增长，在未来的三年内，市场尚无萎缩的可能。但数码相框拓展中国市场需克服两大难点：一是价格；二是拓展应用空间。价格下降是必然趋势，中国消费者接受只是时间的问题，而根据中国消费者的需求和特点，开发出相应的应用产品，则是需要数码相框厂商亟待解决的问题。

从技术上来说，未来数码相框的发展将向两极分化。一部分产品着重强调基本功能和低成本，整合家庭中的闹钟、日历和装饰功能，这些产品走的是低成本路线，以展示照片为主，追求图像的品质及幻灯片播放特效，已成为 DC/DV（数码相机/数码摄像机）的附属物；另一部分产品将会添加一些新的功能，如 Wi-Fi、DVB-T，还可即时报告天气、股票等信息，从而有望成为“桌面信息中心”。



图 1.1.1 项目应用示例图

美国 IDC 公布的一预测认为，数码相框全球市场规模 2011 年将扩大至 06 年的 15 倍。从长远来远，2009 年及今后几年将为处在数码相框产品供应链各段的企业带来巨大的商机。未来几年，数码相框的市场将处在逐渐走向成熟期的阶段，其产销量和市场需求依然将保持大幅度的增长。随着数码相框市场的不断扩大，对行业人才的需求也愈发旺盛，据各类招聘网站显示，目前与数码相框产品相关的职位数量已经达到上千条。在此背景下，我们开展数码相框项目实训，对

于增加学员的技术经验、迎合市场需求具有其现实意义。

## 2、哪里需要图形处理-数码相框技术？

### 2.1 图形处理应用领域

随着娱乐多媒体时代的到来，嵌入式领域的发展也正经历着巨大的技术变革。嵌入式技术不单在工业控制、航天航空等领域发展迅猛，在个人消费电子产品领域更是一枝独秀。巨大的个人消费电子产品市场，更加带动了嵌入式技术、多媒体技术、网络技术、图形图像处理技术的发展。随着新技术层出不穷，各类消费电子产品更是改变功能单一的状况，向多元、集成方向发展。

为了追求完美的消费体验以及全新的视觉享受，消费者对于娱乐功能的要求近乎苛刻。大家希望电子产品能够播放视频电影、高速浏览互联网、拍摄和显示高清晰度照片、收发多媒体邮件、玩互动多媒体游戏、打可视电话，甚至希望通过掌上移动产品收看实时卫星或地面电视节目。图形图像处理技术的发展，使得满足消费者期望成为可能，图形图像处理更是成为目前炙手可热的技术方向。

据预测，到 2012 年我国消费电子市场规模将突破万亿，与图形图像处理相关的产品市场规模更是不可小觑，飞利浦、三星、华旗、纽曼、优可视、长虹等国内外知名企业纷纷涉足此领域，相信随着技术的发展、产业链的完善以及市场的不断成熟，数码相框产品作为新兴的产业分支，必将焕发出勃勃生机与活力。

### 2.2 数码相框相关技术岗位需求

基于目前数码相框产业强劲的发展势头，各公司对于相关技术人才的需求、储备也进行得如火如荼。如表 1.2.1 显示了部分招聘企业信息及岗位技能要求。

表 1.2.1 近 6 个月职位需求表

日期	2010-01-03
公司名称	佳的美电子科技有限公司
职位名称	嵌入式开发工程师
公司简介	佳的美电子科技有限公司创始于 1999 年，是一家集科研、制造、营销为一体的大型股份制、高新技术企业。公司坚持以市场为导向、以视听及多媒体产品为主导，主要产品有：电脑电视接收机（TVBOX）、多媒体影音中心、多媒体液晶电视（LCDTV）、车载液晶电视、数字机顶盒等，拥有电脑电视等共六大类五百多个不同型号的产品，形成了多层面、宽领域、高定位的产品线。历经

	耕耘，产品现已覆盖全国并远销欧美、日本、东南亚、中东等 100 多个国家和地区；现已具备 1500 万台视听及多媒体产品的年生产能力，是全球最大电脑电视接收机生产基地之一。
<b>职位描述</b>	职位描述： 1.嵌入式软件的详细设计和技术文档编写 2.嵌入式软件驱动程序和应用程序代码编写 3.嵌入式软件产品的升级和维护 任职资格： 1.本科以上学历，计算机、应用数学专业 2.2 年以上嵌入式软件开发工作经验，熟悉 ARM 或 MIPS 原理及其体系结构，熟练运用 C 语言和 C++语言 3.具有数字电视和模拟电视的系统知识，熟悉操作系统并能够利用一种以上平台进行程序设计 4.具有数码相框、MP3、MP4、机顶盒嵌入式软件开发经验
<b>日期</b>	2009-12-30
<b>公司名称</b>	晶晨半导体（上海）有限公司
<b>职位名称</b>	软件工程师
<b>公司简介</b>	晶晨半导体是一家业界领先的半导体公司，在视频、音频和图像处理领域提供先进的产品解决方案，广泛应用于数字电视、数码相框、家庭媒体中心和机顶盒等消费电子产品中。晶晨半导体为客户提供一整套综合的方案，以协助他们与时俱进地为消费者推出高品质的电子产品。在这种良好合作关系下，使晶晨半导体的客户有信心和能力专注于他们的核心业务。晶晨半导体先进的网络技术进一步扩展了用户使用电子产品接入互联网的感受，开创了一个消费类电子产品连接到每个家庭和每个人的新时代。
<b>职位描述</b>	职位描述： 负责公司数码相框项目的软件设计开发工作 任职资格： 1.大学本科及以上学历，电子计算机，自动控制等相关专业毕业 2.熟悉嵌入式操作系统和 C 语言 3.有两年以上软件编程经验 4.有消费类产品设计经验优先 5.有硬件基础或设计经验优先 6.有 DPF（数码相框）相关产品应用领域工作经验者优先

日期	2009-12-30
公司名称	深圳市艾佳美电子有限公司
职位名称	数码相框开发工程师
公司简介	深圳市艾佳美电子有限公司是一家集数码相框、电子产品、电子礼品的开发、生产、贸易为一体的高科技公司。公司专业生产/销售：迷你数码相框系列和大屏幕数码相框系列。因为专注，所以更专业，艾佳美数码相框经过卓实发展，在国外和国内都取得了显著成绩，承接了大量的 OEM 订单和 ODM 订单，并在河北、湖北、杭州等十多个省市成立了办事处和合作销售点。
职位描述	职位描述： 负责公司数码相框产品的软件设计开发工作 任职资格： 1.三年以上 MP3、数码相框、便携式媒体播放器等产品嵌入式软件开发经验 2.精通汇编语言，C 语言 3.熟悉 ARM、MIPS 等 32 位嵌入式平台，熟练使用相对应的开发与调试工具，熟悉 uC/OS，uCLinux 等嵌入式操作系统 4.有操作系统优化、裁减、移植等实际项目经验 5.有嵌入式应用软件开发经验 6.精通 USB，NANDFLASH 等嵌入式操作系统的底层硬件设备的驱动开发

### 3、数码相框项目成果演示

本项目中我们将模拟实现市场现有的数码相框产品图片显示功能。通过解析图片信息获取图片名称及图片说明信息，分析图片类型确定图片格式，调用 libjpeg、libpng 库文件实现 jpeg、png 图片解码，调用字体库实现图片说明信息的处理，通过对 FrameBuffer 帧缓冲设备的使用，进行图片、字形的播放显示。

图 1.3.1、1.3.2 为数码相框项目成果演示图，从图中可以看到在背景图片的映衬下前景图片在不断循环播放，同时还能显示每张图片的说明文字。





图 1.3.1 数码相框项目完成效果演示图 1



图 1.3.2 数码相框项目完成效果演示图 2

## 第 2 章 FrameBuffer 编程

---

课程内容：

- ✧ 文件 IO
- ✧ FrameBuffer 编程的基础知识

## 第 2 章 Framebuffer 编程

### 1、文件 IO

#### 1.1 汇编程序的 Hello world

之前我们学习了如何用 C 标准 I/O 库读写文件, 本章详细讲解这些 I/O 操作是怎么实现的。所有 I/O 操作最终都是在内核中做的, 以前我们用的 C 标准 I/O 库函数最终也是通过系统调用把 I/O 操作从用户空间传给内核, 然后 让内核去做 I/O 操作, 本章和下一章会介绍内核中 I/O 子系统的工作原理。首先看一个打印 Hello world 的汇编程序, 了解 I/O 操作是怎样通过系统调用传给内核的。

例 1.1. 汇编程序的 Hello world

```
.data                                # section declaration

msg:
    .ascii    "Hello, world!\n"  # our dear string
    len = . - msg                # length of our dear string

.text                                # section declaration

    # we must export the entry point to the ELF linker or
    .global _start # loader. They conventionally recognize _start as their
    # entry point. Use ld -e foo to override the default.

_start:

# write our string to stdout

    movl    $len,%edx    # third argument: message length
    movl    $msg,%ecx    # second argument: pointer to message to write
    movl    $1,%ebx      # first argument: file handle (stdout)
    movl    $4,%eax      # system call number (sys_write)
    int     $0x80        # call kernel

# and exit

    movl    $0,%ebx      # first argument: exit code
    movl    $1,%eax      # system call number (sys_exit)
```



```
int $0x80      # call kernel
```

像 以前一样，汇编、链接、运行：

```
$ as -o hello.o hello.s
$ ld -o hello hello.o
$ ./hello
Hello, world!
```

这段汇编相当于以下 C 代码：

```
#include <unistd.h>

char msg[14] = "Hello, world!\n";
#define len 14

int main(void)
{
    write(1, msg, len);
    _exit(0);
}
```

.data 段有一个标号 msg，代表字符串"Hello, world!\n"的首地址，相当于 C 程序的一个全局变量。注意 在 C 语言中字符串的末尾隐含有一个'\0'，而汇编指示符.ascii 定义的字符串末尾没有隐含的'\0'。汇编程序中的 len 代表一个常量，它的值由当前地址减去符号 msg 所代表的地址得到，换句话说就是字符串"Hello, world!\n"的长度。现在解释一下这行代码中的“.”，汇编器总是从前到 后把汇编代码转换成目标文件，在这个过程中维护一个地址计数器，当处理到每个段的开头时把地址计数器置成 0，然后每处理一条汇编指示或指令就把地址计数器 增加相应的字节数，在汇编程序中用“.”可以取出当前地址计数器的值，该值是一个常量。

在\_start 中调了两个系统调用，第一个是 write 系 统调用，第二个是以前讲过的\_exit 系统调用。在调 write 系统调用时，eax 寄存器保存着 write 的系统调用号 4，ebx、ecx、edx 寄存器分别保存着 write 系统调用需要的三个参数。ebx 保 存着文件描述符，进程中每个打开的文件都用一个编号来标识，称为文件描述符，文件描述符 1 表示标准输出，对应于 C 标准 I/O 库的 stdout。ecx 保存着输出缓冲区的首 地址。edx 保存着输出的字节数。write 系 统调用把从 msg 开始的 len 个 字节写到标准输出。

C 代码中的 `write` 函数是系统调用的包装函数，其内部实现就是把传进来的三个参数分别赋给 `ebx`、`ecx`、`edx` 寄存器，然后执行 `movl $4,%eax` 和 `int $0x80` 两条指令。这个函数不可能完全用 C 代码来写，因为任何 C 代码都不会编译生成 `int` 指令，所以这个函数有可能是完全用汇编写的，也可能是用 C 内联汇编写的，甚至可能是一个宏定义（省了参数入栈出栈的步骤）。`_exit` 函数也是如此，我们讲过这些系统调用的包装函数位于 `Man Page` 的第 2 个 `Section`。

## 1.2 C 标准 I/O 库函数与 Unbuffered I/O 函数

现在看看 C 标准 I/O 库函数是如何用系统调用实现的。

### ✓ `fopen(3)`

调用 `open(2)` 打开指定的文件，返回一个文件描述符（就是一个 `int` 类型的编号），分配一个 `FILE` 结构体，其中包含该文件的描述符、I/O 缓冲区和当前读写位置等信息，返回这个 `FILE` 结构体的地址。

### ✓ `fgetc(3)`

通过传入的 `FILE *` 参数找到该文件的描述符、I/O 缓冲区和当前读写位置，判断能否从 I/O 缓冲区中读到下一个字符，如果能读到就直接返回该字符，否则调用 `read(2)`，把文件描述符传进去，让内核读取该文件的数据到 I/O 缓冲区，然后返回下一个字符。注意，对于 C 标准 I/O 库来说，打开的文件由 `FILE *` 指针标识，而对于内核来说，打开的文件由文件描述符标识，文件描述符从 `open` 系统调用获得，在使用 `read`、`write`、`close` 系统调用时都需要传文件描述符。

### ✓ `fputc(3)`

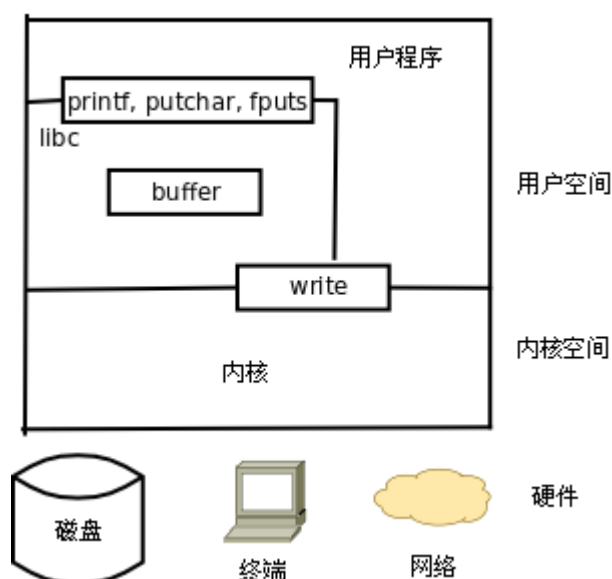
判断该文件的 I/O 缓冲区是否有空间再存放一个字符，如果有空间则直接保存在 I/O 缓冲区中并返回，如果 I/O 缓冲区已满就调用 `write(2)`，让内核把 I/O 缓冲区的内容写回文件。

### ✓ `fclose(3)`

如果 I/O 缓冲区中还有数据没写回文件，就调用 `write(2)` 写回文件，然后调用 `close(2)` 关闭文件，释放 `FILE` 结构体和 I/O 缓冲区。

以写文件为例，C 标准 I/O 库函数（`printf(3)`、`putchar(3)`、`fputs(3)`）与系统调用 `write(2)` 的关系如下图所示。

图 1.1. 库函数与系统调用的层次关系



open、read、write、close 等系统函数称为无缓冲 I/O (Unbuffered I/O) 函数，因为它们位于 C 标准库的 I/O 缓冲区的底层<sup>[46]</sup>。用户程序在读写文件时既可以调用 C 标准 I/O 库函数，也可以直接调用底层的 Unbuffered I/O 函数，那么用哪一组函数好呢？

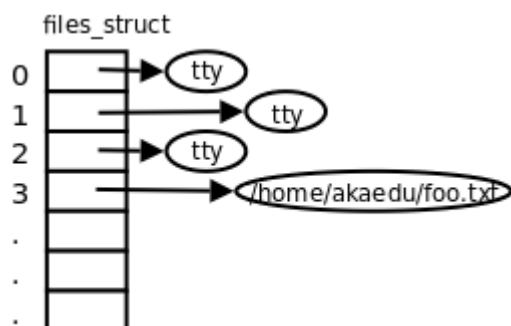
- ✚ 用 Unbuffered I/O 函数每次读写都要进内核，调一个系统调用比调一个用户空间的函数要慢很多，所以在用户空间开辟 I/O 缓冲区还是必要的，用 C 标准 I/O 库函数就比较方便，省去了自己管理 I/O 缓冲区的麻烦。
- ✚ 用 C 标准 I/O 库函数要时刻注意 I/O 缓冲区和实际文件有可能不一致，在必要时需调用 `fflush(3)`。
- ✚ 我们知道 UNIX 的传统是 Everything is a file，I/O 函数不仅用于读写常规文件，也用于读写设备，比如终端或网络设备。在读写设备时通常是不希望有缓冲的，例如向代表网络设备的文件写数据就是希望数据通过网络设备发送出去，而不希望只写到缓冲区里就算完事儿了，当网络设备接收到数据时应用程序也希望第一时间被通知到，所以网络编程通常直接调用 Unbuffered I/O 函数。

C 标准库函数是 C 标准的一部分，而 Unbuffered I/O 函数是 UNIX 标准的一部分，在所有支持 C 语言的平台上应该都可以用 C 标准库函数（除了有些平台的 C 编译器没有完全符合 C 标准之外），而只有在 UNIX 平台上才能使用 Unbuffered I/O 函数，所以 C 标准 I/O 库函数在头文件 `stdio.h` 中声明，而 `read`、`write` 等函数在头文件 `unistd.h` 中声明。在支持 C 语言的非 UNIX 操作系统上，标准 I/O 库的底层可能由另外一组系统函数支持，例如 Windows 系统的底层是 Win32 API，其中读写文件的系统函数是 `ReadFile`、`WriteFile`。

现在该说说文件描述符了。每个进程在 Linux 内核中都有一个 `task_struct`

结构体来维护进程相关的信息，称为进程描述符（Process Descriptor），而在操作系统理论中称为进程控制块（PCB，Process Control Block）。task\_struct 中有一个指针指向 files\_struct 结构体，称为文件描述符表，其中每个表项包含一个指向已打开的文件的指针，如下图所示。

图 1.2. 文件描述符表



至于已打开的文件在内核中用什么结构体表示，我们将在下一章详细介绍，目前我们在画图时用一个圈表示。用户程序不能直接访问内核中的文件描述符表，而只能使用文件描述符表的索引（即 0、1、2、3 这些数字），这些索引就称为文件描述符（File Descriptor），用 int 型变量保存。当调用 open 打开一个文件或创建一个新文件时，内核分配一个文件描述符并返回给用户程序，该文件描述符表项中的指针指向新打开的文件。当读写文件时，用户程序把文件描述符传给 read 或 write，内核根据文件描述符找到相应的表项，再通过表项中的指针找到相应的文件。

我们知道，程序启动时会自动打开三个文件：标准输入、标准输出和标准错误输出。在 C 标准库中分别用 FILE \* 指针 stdin、stdout 和 stderr 表示。这三个文件的描述符分别是 0、1、2，保存在相应的 FILE 结构体中。头文件 unistd.h 中有如下的宏定义来表示这三个文件描述符：

```
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

### 1.3 open/close

open 函数可以打开或创建一个文件。

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
返回值：成功返回新分配的文件描述符，出错返回-1 并设置 errno
```




在 Man Page 中 open 函数有两种形式，一种带两个参数，一种带三个参数，其实在 C 代码中 open 函数是这样声明的：

```
int open(const char *pathname, int flags, ...);
```





最后的可变参数可以是 0 个或 1 个，由 flags 参数中的标志位决定，见下面的详细说明。

pathname 参数是要打开或创建的文件名，和 fopen 一样，pathname 既可以是相对路径也可以是绝对路径。flags 参数有一系列常数值可供选择，可以同时选择多个常数用按位或运算符连接起来，所以这些常数的宏定义都以 O\_开头，表示 or。

必选项：以下三个常数中必须指定一个，且仅允许指定一个。

-  O\_RDONLY 只读打开
-  O\_WRONLY 只写打开
-  O\_RDWR 可读可写打开

以下可选项可以同时指定 0 个或多个，和必选项按位或起来作为 flags 参数。可选项有很多，这里只介绍一部分，其它选项可参考 open(2)的 Man Page：

-  O\_APPEND 表示追加。如果文件已有内容，这次打开文件所写的数据附加到文件的末尾而不覆盖原来的内容。
-  O\_CREAT 若此文件不存在则创建它。使用此选项时需要提供第三个参数 mode，表示该文件的访问权限。
-  O\_EXCL 如果同时指定了 O\_CREAT，并且文件已存在，则出错返回。
-  O\_TRUNC 如果文件已存在，并且以只写或可读可写方式打开，则将其长度截断（Truncate）为 0 字节。



O\_NONBLOCK 对于设备文件，以 O\_NONBLOCK 方式打开可以做非阻塞 I/O (Nonblock I/O)，非阻塞 I/O 在下一节详细讲解。

注意 open 函数与 C 标准 I/O 库的 fopen 函数有些细微的区别：



以可写的方式 fopen 一个文件时，如果文件不存在会自动创建，而 open 一个文件时必须明确指定 O\_CREAT 才会创建文件，否则文件不存在就出错返回。



以 w 或 w+ 方式 fopen 一个文件时，如果文件已存在就截断为 0 字节，而 open 一个文件时必须明确指定 O\_TRUNC 才会截断文件，否则直接在原来的数据上改写。

第三个参数 mode 指定文件权限，可以用八进制数表示，比如 0644 表示 -rw-r--r--，也可以用 S\_IRUSR、S\_IWUSR 等宏定义按位或起来表示，详见 open(2) 的 Man Page。要注意的是，文件权限由 open 的 mode 参数和当前进程的 umask 掩码共同决定。

补充说明一下 Shell 的 umask 命令。Shell 进程的 umask 掩码可以用 umask 命令查看：

```
$ umask
0022
```

用 touch 命令创建一个文件时，创建权限是 0666，而 touch 进程继承了 Shell 进程的 umask 掩码，所以最终的文件权限是  $0666 \& \sim 022 = 0644$ 。

```
$ touch file123
$ ls -l file123
-rw-r--r-- 1 akaedu akaedu 0 2009-03-08 15:07 file123
```

同样道理，用 gcc 编译生成一个可执行文件时，创建权限是 0777，而最终的文件权限是  $0777 \& \sim 022 = 0755$ 。

```
$ gcc main.c
$ ls -l a.out
-rwxr-xr-x 1 akaedu akaedu 6483 2009-03-08 15:07 a.out
```

我们看到的都是被 umask 掩码修改之后的权限，那么如何证明 touch 或 gcc

创建文件的权限本来应该是 0666 和 0777 呢？我们可以把 Shell 进程的 umask 改成 0，再重复上述实验：

```
$ umask 0
$ touch file123
$ rm file123 a.out
$ touch file123
$ ls -l file123
-rw-rw-rw- 1 akaedu akaedu 0 2009-03-08 15:09 file123
$ gcc main.c
$ ls -l a.out
-rwxrwxrwx 1 akaedu akaedu 6483 2009-03-08 15:09 a.out
```

现在我们自己写一个程序，在其中调用 `open("somefile", O_WRONLY|O_CREAT, 0664);` 创建文件，然后在 Shell 中运行并查看结果：

```
$ umask 022
$ ./a.out
$ ls -l somefile
-rw-r--r-- 1 akaedu akaedu 6483 2009-03-08 15:11 somefile
```

不出所料，文件 somefile 的权限是  $0664 \& \sim 022 = 0644$ 。有几个问题现在我没有解释：为什么被 Shell 启动的进程可以继承 Shell 进程的 umask 掩码？为什么 umask 命令可以读写 Shell 进程的 umask 掩码？这些问题将在第 1 节“引言”解释。

close 函数关闭一个已打开的文件：

```
#include <unistd.h>

int close(int fd);
返回值：成功返回 0，出错返回-1 并设置 errno
```

参数 fd 是要关闭的文件描述符。需要说明的是，当一个进程终止时，内核对该进程所有尚未关闭的文件描述符调用 close 关闭，所以即使用户程序不调用







close, 在终止时内核也会自动关闭它打开的所有文件。但是对于一个长年累月运行的程序（比如网络服务器），打开的文件描述符一定要记得关闭，否则随着打开的文件越来越多，会占用大量文件描述符和系统资源。

由 open 返回的文件描述符一定是该进程尚未使用的最小描述符。由于程序启动时自动打开文件描述符 0、1、2，因此第一次调用 open 打开文件通常会返回描述符 3，再调用 open 就会返回 4。可以利用这一点在标准输入、标准输出或标准错误输出上打开一个新文件，实现重定向的功能。例如，首先调用 close 关闭文件描述符 1，然后调用 open 打开一个常规文件，则一定会返回文件描述符 1，这时候标准输出就不再是终端，而是一个常规文件了，再调用 printf 就不会打印到屏幕上，而是写到这个文件中了。后面要讲的 dup2 函数提供了另外一种办法在指定的文件描述符上打开文件。

### ✓ 习题

1、在系统头文件中查找 flags 和 mode 参数用到的这些宏定义的值是多少。把这些宏定义按位或起来是什么效果？为什么必选项只能选一个而可选项可以选择多个？

2、请按照下述要求分别写出相应的 open 调用。

-  打开文件/home/akae.txt 用于写操作，以追加方式打开
-  打开文件/home/akae.txt 用于写操作，如果该文件不存在则创建它
-  打开文件/home/akae.txt 用于写操作，如果该文件已存在则截断为 0 字节，如果该文件不存在则创建它
-  打开文件/home/akae.txt 用于写操作，如果该文件已存在则报错退出，如果该文件不存在则创建它

## 1.4 read/write

read 函数从打开的设备或文件中读取数据。




```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```



返回值：成功返回读取的字节数，出错返回-1 并设置 `errno`  
如果在调 `read` 之前已到达文件末尾，则这次 `read` 返回 0

参数 `count` 是请求读取的字节数，读上来的数据保存在缓冲区 `buf` 中，同时文件的当前读写位置向后移。注意这个读写位置和使用 C 标准 I/O 库时的读写位置有可能不同，这个读写位置是记在内核中的，而使用 C 标准 I/O 库时的读写位置是用户空间 I/O 缓冲区中的位置。比如用 `fgetc` 读一个字节，`fgetc` 有可能从内核中预读 1024 个字节到 I/O 缓冲区中，再返回第一个字节，这时该文件在内核中记录的读写位置是 1024，而在 `FILE` 结构体中记录的读写位置是 1。注意返回值类型是 `ssize_t`，表示有符号的 `size_t`，这样既可以返回正的字节数、0（表示到达文件末尾）也可以返回负值-1（表示出错）。`read` 函数返回时，返回值说明了 `buf` 中前多少个字节是刚读上来的。有些情况下，实际读到的字节数（返回值）会小于请求读的字节数 `count`，例如：

-  读常规文件时，在读到 `count` 个字节之前已到达文件末尾。例如，距文件末尾还有 30 个字节而请求读 100 个字节，则 `read` 返回 30，下次 `read` 将返回 0。
-  从终端设备读，通常以行为单位，读到换行符就返回了。
-  从网络读，根据不同的传输层协议和内核缓存机制，返回值可能小于请求的字节数，后面 `socket` 编程部分会详细讲解。

`write` 函数向打开的设备或文件中写数据。

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

返回值：成功返回写入的字节数，出错返回-1 并设置 `errno`

写常规文件时，`write` 的返回值通常等于请求写的字节数 `count`，而向终端设备或网络写则不一定。

读常规文件是不会阻塞的，不管读多少字节，`read` 一定会在有限的时间内返回。从终端设备或网络读则不一定，如果从终端输入的数据没有换行符，调用 `read` 读终端设备就会阻塞，如果网络上没有接收到数据包，调用 `read` 从网络读就会阻塞，至于会阻塞多长时间也是不确定的，如果一直没有数据到达就一直阻塞在那里。同样，写常规文件是不会阻塞的，而向终端设备或网络写则不一定。

现在明确一下阻塞（Block）这个概念。当进程调用一个阻塞的系统函数时，该进程被置于睡眠（Sleep）状态，这时内核调度其它进程运行，直到该进程等待的事件发生了（比如网络上接收到数据包，或者调用 sleep 指定的睡眠时间到了）它才有可能继续运行。与睡眠状态相对的是运行（Running）状态，在 Linux 内核中，处于运行状态的进程分为两种情况：

🚦 正在被调度执行。CPU 处于该进程的上下文环境中，程序计数器（eip）里保存着该进程的指令地址，通用寄存器里保存着该进程运算过程的中间结果，正在执行该进程的指令，正在读写该进程的地址空间。

🚦 就绪状态。该进程不需要等待什么事件发生，随时都可以执行，但 CPU 暂时还在执行另一个进程，所以该进程在一个就绪队列中等待被内核调度。系统中可能同时有多个就绪的进程，那么该调度谁执行呢？内核的调度算法是基于优先级和时间片的，而且会根据每个进程的运行情况动态调整它的优先级和时间片，让每个进程都能比较公平地得到机会执行，同时要兼顾用户体验，不能让和用户交互的进程响应太慢。

下面这个小程序从终端读数据再写回终端。

### 例 1.2. 阻塞读终端

```
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    char buf[10];
    int n;
    n = read(STDIN_FILENO, buf, 10);
    if (n < 0) {
        perror("read STDIN_FILENO");
        exit(1);
    }
    write(STDOUT_FILENO, buf, n);
    return 0;
}
```

执行结果如下：

```
$ ./a.out
```

```
hello (回车)
hello
$ ./a.out
hello world (回车)
hello worl$d
bash: d: command not found
```

第一次执行 `a.out` 的结果很正常，而第二次执行的过程有点特殊，现在分析一下：

1. Shell 进程创建 `a.out` 进程，`a.out` 进程开始执行，而 Shell 进程睡眠等待 `a.out` 进程退出。
2. `a.out` 调用 `read` 时睡眠等待，直到终端设备输入了换行符才从 `read` 返回，`read` 只读走 10 个字符，剩下的字符仍然保存在内核的终端设备输入缓冲区中。
3. `a.out` 进程打印并退出，这时 Shell 进程恢复运行，Shell 继续从终端读取用户输入的命令，于是读走了终端设备输入缓冲区中剩下的字符 `d` 和换行符，把它当成一条命令解释执行，结果发现执行不了，没有 `d` 这个命令。

如果在 `open` 一个设备时指定了 `O_NONBLOCK` 标志，`read/write` 就不会阻塞。以 `read` 为例，如果设备暂时没有数据可读就返回 -1，同时置 `errno` 为 `EWOULDBLOCK`（或者 `EAGAIN`，这两个宏定义的值相同），表示本来应该阻塞在这里（`would block`，虚拟语气），事实上并没有阻塞而是直接返回错误，调用者应该试着再读一次（`again`）。这种行为方式称为轮询（Poll），调用者只是查询一下，而不是阻塞在这里死等，这样可以同时监视多个设备：

```
while(1) {
    非阻塞 read(设备 1);
    if(设备 1 有数据到达)
        处理数据;
    非阻塞 read(设备 2);
    if(设备 2 有数据到达)
        处理数据;
    ...
}
```

如果 `read(设备 1)` 是阻塞的，那么只要设备 1 没有数据到达就会一直阻塞在设备 1 的 `read` 调用上，即使设备 2 有数据到达也不能处理，使用非阻塞 I/O 就可

以避免设备 2 得不到及时处理。

非阻塞 I/O 有一个缺点，如果所有设备都一直没有数据到达，调用者需要反复查询做无用功，如果阻塞在那里，操作系统可以调度别的进程执行，就不会做无用功了。在使用非阻塞 I/O 时，通常不会在一个 while 循环中一直不停地查询（这称为 Tight Loop），而是每延迟等待一会儿来查询一下，以免做太多无用功，在延迟等待的时候可以调度其它进程执行。

```
while(1) {  
    非阻塞 read(设备 1);  
    if(设备 1 有数据到达)  
        处理数据;  
    非阻塞 read(设备 2);  
    if(设备 2 有数据到达)  
        处理数据;  
    ...  
    sleep(n);  
}
```

这样做的问题是，设备 1 有数据到达时可能不能及时处理，最长需延迟  $n$  秒才能处理，而且反复查询还是做了很多无用功。以后要学习的 `select(2)` 函数可以阻塞地同时监视多个设备，还可以设定阻塞等待的超时时间，从而圆满地解决了这个问题。

以下是一个非阻塞 I/O 的例子。目前我们学过的可能引起阻塞的设备只有终端，所以我们用终端来做这个实验。程序开始执行时在 0、1、2 文件描述符上自动打开的文件就是终端，但是没有 `O_NONBLOCK` 标志。所以就像例 28.2 “阻塞读终端”一样，读标准输入是阻塞的。我们可以重新打开一遍设备文件 `/dev/tty`（表示当前终端），在打开时指定 `O_NONBLOCK` 标志。

### 例 1.3. 非阻塞读终端

```
#include <unistd.h>  
#include <fcntl.h>  
#include <errno.h>  
#include <string.h>  
#include <stdlib.h>
```

```
#define MSG_TRY "try again\n"

int main(void)
{
    char buf[10];
    int fd, n;
    fd = open("/dev/tty", O_RDONLY|O_NONBLOCK);
    if(fd<0) {
        perror("open /dev/tty");
        exit(1);
    }
tryagain:
    n = read(fd, buf, 10);
    if (n < 0) {
        if (errno == EAGAIN) {
            sleep(1);
            write(STDOUT_FILENO, MSG_TRY, strlen(MSG_TRY));
            goto tryagain;
        }
        perror("read /dev/tty");
        exit(1);
    }
    write(STDOUT_FILENO, buf, n);
    close(fd);
    return 0;
}
```

以下是用非阻塞 I/O 实现等待超时的例子。既保证了超时退出的逻辑又保证了有数据到达时处理延迟较小。

#### 例 1.4. 非阻塞读终端和等待超时

```
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

#define MSG_TRY "try again\n"
#define MSG_TIMEOUT "timeout\n"

int main(void)
```

```
{
    char buf[10];
    int fd, n, i;
    fd = open("/dev/tty", O_RDONLY|O_NONBLOCK);
    if(fd<0) {
        perror("open /dev/tty");
        exit(1);
    }
    for(i=0; i<5; i++) {
        n = read(fd, buf, 10);
        if(n>=0)
            break;
        if(errno!=EAGAIN) {
            perror("read /dev/tty");
            exit(1);
        }
        sleep(1);
        write(STDOUT_FILENO, MSG_TRY, strlen(MSG_TRY));
    }
    if(i==5)
        write(STDOUT_FILENO, MSG_TIMEOUT, strlen(MSG_TIMEOUT));
    else
        write(STDOUT_FILENO, buf, n);
    close(fd);
    return 0;
}
```

以下是利用 open/close, read/write 实现的一个简单版本的 cp 命令

### 例 1.5. cp 命令实现

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define MAX 4096
/* ./cp 1.c 2.c */
int main(int argc, char * argv[])
{
    /* step 1: open file */
    int fd_in = open(argv[1], O_RDONLY);
    int fd_out = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0644);
```

```
/* step 2: read and write file */
char buf[MAX];
int n;
while((n = read(fd_in, buf, MAX)) > 0)
    write(fd_out, buf, n);

/* step 3: close file */
close(fd_in);
close(fd_out);
return 0;
}
```

注意,在真正的项目中没有容错处理的代码,可以认为是垃圾代码.在本书为了让诸位看到一些系统调用的基本用法,没有加容错处理,并不表示不需要容错,以下类似.

## 1.5 lseek

每个打开的文件都记录着当前读写位置,打开文件时读写位置是 0,表示文件开头,通常读写多少个字节就会将读写位置往后移多少个字节.但是有一个例外,如果以 O\_APPEND 方式打开,每次写操作都会在文件末尾追加数据,然后将读写位置移到新的文件末尾。lseek 和标准 I/O 库的 fseek 函数类似,可以移动当前读写位置(或者叫偏移量)。

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

参数 offset 和 whence 的含义和 fseek 函数完全相同。只不过第一个参数换成了文件描述符。和 fseek 一样,偏移量允许超过文件末尾,这种情况下对该文件的下一次写操作将延长文件,中间空洞的部分读出来都是 0。

若 lseek 成功执行,则返回新的偏移量,因此可用以下方法确定一个打开文件的当前偏移量:

```
off_t curpos;
```

```
currpos = lseek(fd, 0, SEEK_CUR);
```

这种方法也可用来确定文件或设备是否可以设置偏移量，常规文件都可以设置偏移量，而设备一般是不可以设置偏移量的。如果设备不支持 `lseek`，则 `lseek` 返回-1，并将 `errno` 设置为 `ESPIPE`。注意 `fseek` 和 `lseek` 在返回值上有细微的差别，`fseek` 成功时返回 0 失败时返回-1，要返回当前偏移量需调用 `ftell`，而 `lseek` 成功时返回当前偏移量失败时返回-1。

## 1.6 fcntl

先前我们以 `read` 终端设备为例介绍了非阻塞 I/O，为什么我们不直接对 `STDIN_FILENO` 做非阻塞 `read`，而要重新 `open` 一遍 `/dev/tty` 呢？因为 `STDIN_FILENO` 在程序启动时已经被自动打开了，而我们需要在调用 `open` 时指定 `O_NONBLOCK` 标志。这里介绍另外一种办法，可以用 `fcntl` 函数改变一个已打开的文件的属性，可以重新设置读、写、追加、非阻塞等标志（这些标志称为 File Status Flag），而不必重新 `open` 文件。

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

这个函数和 `open` 一样，也是用可变参数实现的，可变参数的类型和个数取决于前面的 `cmd` 参数。下面的例子使用 `F_GETFL` 和 `F_SETFL` 这两种 `fcntl` 命令改变 `STDIN_FILENO` 的属性，加上 `O_NONBLOCK` 选项，实现和例 28.3 “非阻塞读终端”同样的功能。

### 例 2.6. 用 `fcntl` 改变 File Status Flag

```
#include <unistd.h>
#include <fcntl.h>
```



```
#include <errno.h>
#include <string.h>
#include <stdlib.h>

#define MSG_TRY "try again\n"

int main(void)
{
    char buf[10];
    int n;
    int flags;
    flags = fcntl(STDIN_FILENO, F_GETFL);
    flags |= O_NONBLOCK;
    if (fcntl(STDIN_FILENO, F_SETFL, flags) == -1) {
        perror("fcntl");
        exit(1);
    }
tryagain:
    n = read(STDIN_FILENO, buf, 10);
    if (n < 0) {
        if (errno == EAGAIN) {
            sleep(1);
            write(STDOUT_FILENO, MSG_TRY, strlen(MSG_TRY));
            goto tryagain;
        }
        perror("read stdin");
        exit(1);
    }
    write(STDOUT_FILENO, buf, n);
    return 0;
}
```

以下程序通过命令行的第一个参数指定一个文件描述符，同时利用 Shell 的重定向功能在该描述符上打开文件，然后用 fcntl 的 F\_GETFL 命令取出 File Status Flag 并打印。

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
{
    int val;
    if (argc != 2) {
        fputs("usage: a.out <descriptor#>\n", stderr);
        exit(1);
    }
    if ((val = fcntl(atoi(argv[1]), F_GETFL)) < 0) {
        printf("fcntl error for fd %d\n", atoi(argv[1]));
        exit(1);
    }
    switch(val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;
    case O_WRONLY:
        printf("write only");
        break;
    case O_RDWR:
        printf("read write");
        break;
    default:
        fputs("invalid access mode\n", stderr);
        exit(1);
    }
    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
    putchar('\n');
    return 0;
}
```

运行该程序的几种情况解释如下。

```
$ ./a.out 0 < /dev/tty
read only
```

Shell 在执行 a.out 时将它的标准输入重定向到/dev/tty, 并且是只读的。argv[1]

是 0，因此取出文件描述符 0（也就是标准输入）的 File Status Flag，用掩码 O\_ACCMODE 取出它的读写位，结果是 O\_RDONLY。注意，Shell 的重定向语法不属于程序的命令行参数，这个命令只有两个参数，argv[0]是"./a.out"，argv[1]是"0"，重定向由 Shell 解释，在启动程序时已经生效，程序在运行时并不知道标准输入被重定向了。

```
$ ./a.out 1 > temp.foo  
$ cat temp.foo  
write only
```

Shell 在执行 a.out 时将它的标准输出重定向到文件 temp.foo，并且是只写的。程序取出文件描述符 1 的 File Status Flag，发现是只写的，于是打印 write only，但是打印不到屏幕上而是打印到 temp.foo 这个文件中了。

```
$ ./a.out 2 2>>temp.foo  
write only, append
```

Shell 在执行 a.out 时将它的标准错误输出重定向到文件 temp.foo，并且是只写和追加方式。程序取出文件描述符 2 的 File Status Flag，发现是只写和追加方式的。

```
$ ./a.out 5 5<>temp.foo  
read write
```

Shell 在执行 a.out 时在它的文件描述符 5 上打开文件 temp.foo，并且是可读可写的。程序取出文件描述符 5 的 File Status Flag，发现是可读可写的。

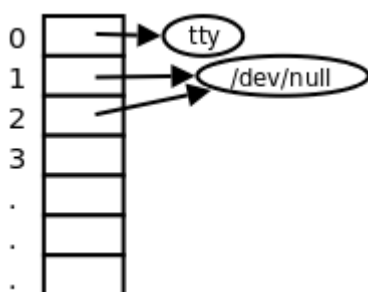
我们看到一种新的 Shell 重定向语法，如果在<、>、>>、<>前面添一个数字，该数字就表示在哪个文件描述符上打开文件，例如 2>>temp.foo 表示将标准错误输出重定向到文件 temp.foo 并且以追加方式写入文件，注意 2 和>>之间不能有空格，否则 2 就被解释成命令行参数了。文件描述符数字还可以出现在重定向符

号右边，例如：

```
$ command > /dev/null 2>&1
```

首先将某个命令 `command` 的标准输出重定向到 `/dev/null`，然后将该命令可能产生的错误信息（标准错误输出）也重定向到和标准输出（用 `&1` 标识）相同的文件，即 `/dev/null`，如下图所示。

图 2.3. 重定向之后的文件描述符表



重定向之后的文件描述符表

`/dev/null` 设备文件只有一个作用，往它里面写任何数据都被直接丢弃。因此保证了该命令执行时屏幕上没有任何输出，既不打印正常信息也不打印错误信息，让命令安静地执行，这种写法在 `Shell` 脚本中很常见。注意，文件描述符数字写在重定向符号右边需要加 `&` 号，否则就被解释成文件名了，`2>&1` 其中的 `>` 左右两边都不能有空格。

除了 `F_GETFL` 和 `F_SETFL` 命令之外，`fcntl` 还有很多命令做其它操作，例如设置文件记录锁等。可以通过 `fcntl` 设置的都是当前进程如何访问设备或文件的访问控制属性，例如读、写、追加、非阻塞、加锁等，但并不设置文件或设备本身的属性，例如文件的读写权限、串口波特率等。下一节要介绍的 `ioctl` 函数用于设置某些设备本身的属性，例如串口波特率、终端窗口大小，注意区分这两个函数的作用。

## 1.8 ioctl

ioctl 用于向设备发控制和配置命令，有些命令也需要读写一些数据，但这些数据是不能用 read/write 读写的，称为 Out-of-band 数据。也就是说，read/write 读写的数据是 in-band 数据，是 I/O 操作的主体，而 ioctl 命令传送的是控制信息，其中的数据是辅助的数据。例如，在串口线上收发数据通过 read/write 操作，而串口的波特率、校验位、停止位通过 ioctl 设置，A/D 转换的结果通过 read 读取，而 A/D 转换的精度和工作频率通过 ioctl 设置。

```
#include <sys/ioctl.h>

int ioctl(int d, int request, ...);
```

d 是某个设备的文件描述符。request 是 ioctl 的命令，可变参数取决于 request，通常是一个指向变量 或结构体的指针。若出错则返回 -1，若成功则返回其他值，返回值也是取决于 request。

### 例 2.7. 使用 FBIOGET\_VSCREENINFO 命令获得液晶屏的窗口大

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/fb.h>

int main(int argc, char *argv[])
{
    struct fb_var_screeninfo fb_var;

    int fd = open("/dev/fb0", O_RDWR);
    ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
    printf("width : %d\t", fb_var.xres);
    printf("high: %d\t", fb_var.yres);
    printf("bpp: %d\n", fb_var.bits_per_pixel);

    close(fd);
}
```

```
return 0;  
}
```

注意: 访问 "/dev/fb0"显示器的设备文件,在新版本的系统系统上需要 root 用户权限.

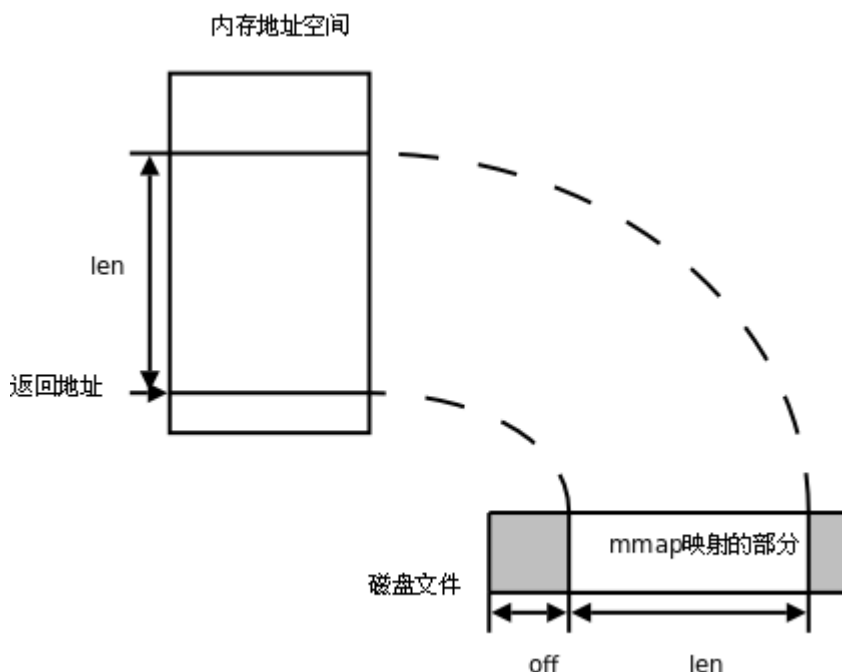
## 1.9 mmap

mmap 可以把磁盘文件的一部分直接映射到内存,这样文件中的位置直接就有对应的内存地址,对文件的读写可以直接用指针来做而不需要 read/write 函数。

```
#include <sys/mman.h>  
  
void *mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off);  
int munmap(void *addr, size_t len);
```

该函数各参数的作用图示如下：

图 2.4. mmap 函数



如果 `addr` 参数为 `NULL`，内核会自己在进程地址空间中选择合适的地址建立映射。如果 `addr` 不是 `NULL`，则给内核一个提示，应该从什么地址开始映射，内核会选择 `addr` 之上的某个合适的地址开始映射。建立映射后，真正的映射首地址通过返回值可以得到。`len` 参数是需要映射的那一部分文件的长度。`off` 参数是从文件的什么位置开始映射，必须是页大小的整数倍（在 32 位体系结构上通常是 4K）。`filedes` 是代表该文件的描述符。

`prot` 参数有四种取值：

- 🚩 `PROT_EXEC` 表示映射的这一段可执行，例如映射共享库
- 🚩 `PROT_READ` 表示映射的这一段可读
- 🚩 `PROT_WRITE` 表示映射的这一段可写
- 🚩 `PROT_NONE` 表示映射的这一段不可访问

`flag` 参数有很多种取值，这里只讲两种，其它取值可查看 `mmap(2)`

- 🚩 `MAP_SHARED` 多个进程对同一个文件的映射是共享的，一个进程对映射的内存做了修改，另一个进程也会看到这种变化。
- 🚩 `MAP_PRIVATE` 多个进程对同一个文件的映射不是共享的，一个进程对映射的内存做了修改，另一个进程并不会看到这种变化，也不会真的写到文件中去。

如果 `mmap` 成功则返回映射首地址，如果出错则返回常数 `MAP_FAILED`。当进程终止时，该进程的映射内存会自动解除，也可以调用 `munmap` 解除映射。`munmap` 成功返回 0，出错返回-1。

下面做一个简单的实验。

```
$ vi hello
（编辑该文件的内容为“hello”）
$ od -tx1 -tc hello
00000000 68 65 6c 6c 6f 0a
          h  e  l  l  o  \n
00000006
```

现在用如下程序操作这个文件（注意，把 `fd` 关掉并不影响该文件已建立的映射，仍然可以对文件进行读写）。

```
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>

int main(void)
{
    int *p;
    int fd = open("hello", O_RDWR);
    if (fd < 0) {
        perror("open hello");
        exit(1);
    }
    p = mmap(NULL, 6, PROT_WRITE, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
    close(fd);
    p[0] = 0x30313233;
    munmap(p, 6);
    return 0;
}
```



然后再查看这个文件的内容：

```
$ od -tx1 -tc hello
00000000 33 32 31 30 6f 0a
          3  2  1  0   o  \n
00000006
```

请读者自己分析一下实验结果。

mmap 函数的底层也是一个系统调用，在执行程序时经常要用到这个系统调用来映射共享库到该进程的地址空间。例如一个很简单的 hello world 程序：

```
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 0;
}
```

用 strace 命令执行该程序，跟踪该程序执行过程中用到的所有系统调用的参数及返回值：

```
$ strace ./a.out
execve("./a.out", [ "./a.out" ], [ /* 38 vars */ ]) = 0
brk(0)                                = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)     = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fca000
access("/etc/ld.so.preload", R_OK)     = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)     = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=63628, ...}) = 0
mmap2(NULL, 63628, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fba000
```

```

close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\260a\1"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1339816, ...}) = 0
mmap2(NULL, 1349136, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e70000
mmap2(0xb7fb4000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xb7fb4000) = 0xb7fb4000
mmap2(0xb7fb7000, 9744, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7fb7000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7e6f000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e6f6b0, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
mprotect(0xb7fb4000, 4096, PROT_READ) = 0
munmap(0xb7fba000, 63628) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fc9000
write(1, "hello world\n", 12hello world
) = 12
exit_group(0) = ?
Process 8572 detached
  
```

可以看到, 执行这个程序要映射共享库/lib/tls/i686/cmov/libc.so.6 到进程地址空间。也可以看到, printf 函数的底层确实是调用 write。

## 2、Framebuffer 编程的基础知识

### 2.1 FrameBuffer 工作原理

Framebuffer 是出现在 2.2.xx 内核当中的一种驱动程序接口。

Linux 是工作在保护模式下, 所以用户态进程是无法象 DOS 那样使用显卡 BIOS 里提供的中断调用来实现直接写屏, Linux 抽象出 FrameBuffer 这个设备来供用户态进程实现直接写屏。Framebuffer 机制模仿显卡的功能, 将显卡硬

件结构抽象掉, 可以通过 `Framebuffer` 的读写直接对显存进行操作。用户可以将 `Framebuffer` 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应在屏幕上。这种操作是抽象的, 统一的。用户不必关心物理显存的位置、换页机制等等具体细节。这些都是由 `Framebuffer` 设备驱动来完成的。

但 `Framebuffer` 本身不具备任何运算数据的能力, 就只好比是一个暂时存放水的水池。CPU 将运算后的结果放到这个水池, 水池再将结果流到显示器。中间不会对数据做处理。应用程序也可以直接读写这个水池的内容。在这种机制下, 尽管 `Framebuffer` 需要真正的显卡驱动的支持, 但所有显示任务都有 CPU 完成, 因此 CPU 负担很重

`framebuffer` 的设备文件一般是 `/dev/fb0`、`/dev/fb1` 等等。

可以用命令: `#dd if=/dev/zero of=/dev/fb` 清空屏幕。

如果显示模式是 `1024x768-8` 位色, 用命令: `$ dd if=/dev/zero of=/dev/fb0 bs=1024 count=768` 清空屏幕;

用命令: `#dd if=/dev/fb of=fbfile` 可以将 `fb` 中的内容保存下来;

可以重新写回屏幕: `#dd if=fbfile of=/dev/fb;`

在使用 `Framebuffer` 时, Linux 是将显卡置于图形模式下的。

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 `1024x768-8` 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); //这个命令应该只有在 root 可以执行
```

`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、像素结构、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。

通过 `FrameBuffer` 设备, 还可以获得当前内核所支持的加速显示卡的类型 (通过固定信息得到), 这种类型通常是和特定显示芯片相关的。比如目前最新的内核 (2.4.9) 中, 就 包含有对 `S3`、`Matrox`、`nVidia`、`3Dfx` 等等流行显示

芯片的加速支持。在获得了加速芯片类型之后，应用程序就可以将 PCI 设备的内存 I/O（memio）映射到进程的地址空间。这些 memio 一般是用来控制显示卡的寄存器，通过对这些寄存器的操作，应用程序就可以控制特定显卡的加速功能。

PCI 设备可以将自己的控制寄存器映射到物理内存空间，而后，对这些控制寄存器的访问，就变成了对物理内存的访问。因此，这些寄存器又被称为“memio”。一旦被映射到物理内存，Linux 的普通进程就可以通过 mmap 将这些内存 I/O 映射到进程地址空间，这样就可以直接访问这些寄存器了。

当然，因为不同的显示芯片具有不同的加速能力，对 memio 的使用和定义也各自不同，这时，就需要针对加速芯片的不同类型来编写实现不同的加速功能。比如大多数芯片都提供了对矩形填充的硬件加速支持，但不同的芯片实现方式不同，这时，就需要针对不同的芯片类型编写不同的用来完成填充矩形的函数。

FrameBuffer 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，还需要自己动手完成其他许多工作。

## 2.2 FrameBuffer 相关数据结构分析

FrameBuffer 设备驱动基于如下两个文件：

- 1) linux/include/linux/fb.h
- 2) linux/drivers/video/fbmem.c

下面就 fb.h 中的主要结构进行分析

### 1、fb\_var\_screeninfo

这个结构描述了显卡的特性

说明：\_\_u32 代表 unsigned 无符号 32 bits 数据类型，其余类推。

这是 Linux 内核中所用到的数据类型，如果是开发用户空间(user-space)的程序，可以根据具体计算机平台的情况，用 unsigned long 等来代替

```

struct fb_var_screeninfo
{
    __u32 xres;                /* visible resolution */
    __u32 yres;                /* visible resolution */
    __u32 xres_virtual;        /* virtual resolution */
    __u32 yres_virtual;        /* virtual resolution */
    __u32 xoffset;              /* offset from virtual to visible resolution */

```

```

__u32 yoffset;

__u32 bits_per_pixel;          /* guess what */
__u32 grayscale;              /* != 0 Gray levels instead of colors */

struct fb_bitfield red;        /* bitfield in fb mem if true color, */
struct fb_bitfield green;      /* else only length is significant */
struct fb_bitfield blue;
struct fb_bitfield transp;     /* transparency */
__u32 nonstd;                  /* != 0 Non standard pixel format */

__u32 activate;                /* see FB_ACTIVATE_ */

__u32 height;                  /* height of picture in mm */
__u32 width;                    /* width of picture in mm */

__u32 accel_flags;             /* acceleration flags (hints) */

/* Timing: All values in pixclocks, except pixclock (of course) */

__u32 pixclock;                /* pixel clock in ps (pico seconds) */
__u32 left_margin;             /* time from sync to picture */
__u32 right_margin;            /* time from picture to sync */
__u32 upper_margin;            /* time from sync to picture */
__u32 lower_margin;
__u32 hsync_len;               /* length of horizontal sync */
__u32 vsync_len;               /* length of vertical sync */
__u32 sync;                     /* see FB_SYNC_ */
__u32 vmode;                    /* see FB_VMODE_ */
__u32 reserved[6];             /* Reserved for future compatibility */
};

```

## 2.3 Framebuffer 的使用

Framebuffer 主要是根据 VESA 标准的实现的，所以只能实现最简单的功能。

由于涉及内核的问题，Framebuffer 是不允许在系统起来后修改显示模式等一系列操作。（好象很多人都想要这样干，这是不被允许的，当然如果你自己写驱动的话，是可以实现的）。

对 Framebuffer 的操作，会直接影响到本机的所有控制台的输出，包括 XWIN 的图形界面。在新版本 ubuntu9.10 后的 Linux 系统上，操作 Framebuffer，对

Xwin 的影响系统已经处理的很好。基本不影响 Xwin。

好，现在可以让我们开始实现直接写屏：

- 1、打开一个 FrameBuffer 设备
- 2、通过 mmap 调用把显卡的物理内存空间映射到用户空间
- 3、直接写内存。

```
/******  
File name : fbtools.h  
*/  
#ifndef _FBTOOLS_H_  
#define _FBTOOLS_H_  
#include <linux/fb.h>  
//a framebuffer device structure;  
typedef struct fbdev{  
    int fb;  
    unsigned long fb_mem_offset;  
    unsigned long fb_mem;  
    struct fb_fix_screeninfo fb_fix;  
    struct fb_var_screeninfo fb_var;  
    char dev[20];  
} FBDEV, *PFBDEV;  
//open & init a frame buffer  
//to use this function,  
//you must set FBDEV.dev="/dev/fb0"  
//or "/dev/fbX"  
//it's your frame buffer.  
int fb_open(PFBDEV pFbdev);  
  
//close a frame buffer  
int fb_close(PFBDEV pFbdev);  
  
//get display depth  
int get_display_depth(PFBDEV pFbdev);  
  
//full screen clear  
void fb_memset(void *addr, int c, size_t len);  
#endif  
  
/******  
File name : fbtools.c  
*/  
#include <stdio.h>
```

```

#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <asm/page.h>
#include "fbtools.h"
#define TRUE      1
#define FALSE     0
#define MAX(x,y)  ((x)>(y)?(x):y)
#define MIN(x,y)  ((x)<(y)?(x):y)

//open & init a frame buffer
int fb_open(PFBDEV pFbdev)
{
    pFbdev->fb = open(pFbdev->dev, O_RDWR);
    if(pFbdev->fb < 0)
    {
        printf("Error opening %s: %m. Check kernel config\n", pFbdev->dev);
        return FALSE;
    }
    if(-1==ioctl(pFbdev->fb,FBIOGET_VSCREENINFO,&(pFbdev->fb_var)))
    {
        printf("ioctl FBIOGET_VSCREENINFO\n");
        return FALSE;
    }
    if (-1 == ioctl(pFbdev->fb,FBIOGET_FSCREENINFO,&(pFbdev->fb_fix)))
    {
        printf("ioctl FBIOGET_FSCREENINFO\n");
        return FALSE;
    }
    //map physics address to virtual address
    pFbdev->fb_mem_offset = (unsigned long)(pFbdev->fb_fix.smem_start) &
(~PAGE_MASK);
    pFbdev->fb_mem = (unsigned long int)mmap(NULL, pFbdev->fb_fix.smem_len +
pFbdev->fb_mem_offset,          PROT_READ | PROT_WRITE, MAP_SHARED,
pFbdev->fb, 0);
    if (-1L == (long) pFbdev->fb_mem)
    {
        printf("mmap error! mem:%d offset:%d\n", pFbdev->fb_mem,
pFbdev->fb_mem_offset);
        return FALSE;
    }
}

```



```
        return TRUE;
    }
    //close frame buffer
    int fb_close(PFBDEV pFbdev)
    {
        close(pFbdev->fb);
        pFbdev->fb=-1;
    }
    //get display depth
    int get_display_depth(PFBDEV pFbdev);
    {
        if(pFbdev->fb<=0)
        {
            printf("fb device not open, open it first\n");
            return FALSE;
        }
        return pFbdev->fb_var.bits_per_pixel;
    }
    //full screen clear
    void fb_memset (void *addr, int c, size_t len)
    {
        memset(addr, c, len);
    }
    //use by test
    #define DEBUG
    #ifdef DEBUG
    main()
    {
        FBDEV fbdev;
        memset(&fbdev, 0, sizeof(FBDEV));
        strcpy(fbdev.dev, "/dev/fb0");
        if(fb_open(&fbdev)==FALSE)
        {
            printf("open frame buffer error\n");
            return;
        }
        fb_memset(fbdev.fb_mem + fbdev.fb_mem_offset, 0, fbdev.fb_fix.smem_len);
        fb_close(&fbdev);
    }
}
```

## 第 3 章 鼠标的工作原理

---

课程内容：

- ✧ 鼠标的设备文件
- ✧ 辅助代码声明
- ✧ 标准鼠标的数据格式
- ✧ 滚轮鼠标的数据格式
- ✧ 鼠标的绘制
- ✧ 完整的代码实例

## 第 3 章 鼠标的工作原理

### 0、简介

以下信息含代码在 ubuntu10.04 下验证过。

目前最常见的鼠标有 PS/2 鼠标和 USB 鼠标。但从我们应用层去看这两种鼠标其实是没有太大区别的，那我们就以 PS/2 鼠标为例，来谈谈在应用层如何来操作鼠标好了，

PS/2 鼠标有 4 种工作模式，具体如下：

1. 复位模式。当上电后或接收到复位命令 FF 后鼠标即处于此模式。鼠标进行自检和初始化，再向主机发送 0xFA, 0xAA 和 0x00，一些参数将恢复到默认值，即采样率为 100sample/s 非自动流速、流模式、分辨率为 4 计数/mm、禁止状态。

2. 流模式。如果有按键或滚轮动作，即向系统发送信息，最大发送速率就是可编程的采样率。

3. 遥控模式。只有主机发送了模式设置指令 0xF0 后，鼠标才进入这种模式。

4. 这种模式只用于检测鼠标与主机是否连接正确，在该模式下鼠标收到什么就返回什么，除非收到退出卷绕指令 0xEC 或复位指令 0xFF。

流模式就是默认的工作模式。我们在应用层探讨的操作鼠标也就是工作在这种模式下，鼠标的任何动作都会报告给主机。也可以使用遥控模式，主机使用 0xEB 命令请求数据，鼠标进行应答。

下面我就分这几个方面来探讨，在 linux 下对鼠标的操作（捕获）；

### 1、鼠标的设备文件

鼠标的设备文件是：

```
$ ls -l /dev/input/mice
```

执行如下命令，你可以看到鼠标的工作

```
$ cat /dev/input/mice
```

通过执行 `cat` 命令，当鼠标有动作(移动、按键、滚轴)的时候，你可以看到屏幕有很多信息显示当然，多数看到的是乱码，主要原因是你读到的鼠标动作数据，不一定是 0-127 之间的可见字符。

注意：对设备文件的操作通常需要 `root` 用户权限，如果你不是 `root` 用户，你的上述操作，可能会提示 “Permission denied”

```
$ sudo cat /dev/input/mice
```

但为了方便，建议你更改鼠标设备文件的操作权限：

```
$ sudo chmod 777 /dev/input/mice
```

这样以普通用户的权限，也可以操作了。

## 2、辅助代码声明

在下面所用到的类型、变量名分别表示如下：

```
typedef unsigned char u8_t;  
typedef signed char s8_t;
```

```
typedef struct
{
    int dx;                // * mice move left and right /
    int dy;                // * mice move up and down /
    int dz;                // * wheel
    int button;
}mevent_t;

int mice_fd = open("/dev/input/mice", O_RDWR); // mice_fd, 鼠标设备的文件描述符

mevent_t mevent;          // 用来存储鼠标的数据信息
```

### 3、标准鼠标的数据格式

标准的 PS/2 鼠标支持左右移动和三个按键，协议数据格式为 3 字节，如下表所示。鼠标的按键和移动信息都采用这种格式汇报给主机。

- 第一字节 鼠标按键信息(详细如下)， 按位表示
- 第二字节 鼠标横向移动信息(x 方向)， x 方向相对增量，补码表示，自左向右取正值
- 第三字节 鼠标纵向移动信息(y 方向)， y 方向相对增量，补码表示，自下向上取正值

对于鼠标的按键信息分别按位描述如下：

D7	D6	D5	D4	D3	D2	D1	D0
Y overflow	X overflow	Y sign	X sign	1	middle button	right button	left button

每一位值的含意如下：

- D0 左键的按键信息， 1： 按下； 0： No
- D1 右键的按键信息， 1： 按下； 0： No
- D2 中键的按键信息， 1： 按下； 0： No
- D4 鼠标移动方向， X 坐标方向， 1： 自右向左移动； 0： 相反方向
- D5 鼠标移动方向， Y 坐标方向， 1： 自上向下移动； 0： 相反方向

获取鼠标信息的关键代码如下

```
// 注意，本例子， buf 为有符号数
s8_t buf[N]; //从网络的资料来看，要求 N > 3 即可，

//读取鼠标信息
read(mice_fd, buf, N);

// 获取鼠标的按键信息， 1：左键，2：右键，4：中建
mevent.button = buf[0] & 0x07; //数组下标从 0 开始

// dx, x 方向增量，右移为正
mevent.dx = buf[1];
// dy, y 方向增量， 下移为正
mevent.dy = -buf[2];
// dy 之所以要加负号， 通常坐标系我一般以左下角为坐标原点，向右 x 为正，向上 y 为
正
// 而我目前的屏幕坐标系，是以左上角为坐标原点，向右 x 为正，向下 y 为正，故 dx 取
原值，而 dy 取相反值
```

在网络上通常还看到另外一种对 dx, dy 的解析方式：

// 注意： 本例子， buf 为无符号数

```
u8_t buf[N];
read(mice_fd, buf, N)

mevent.button = buf[0] & 0x07;
mevent.dx = (buf[0] & 0x10 ? buf[1] - 256 : buf[1];
mevent.dy = (buf[0] & 0x20 ? -(buf[2] - 256) : -buf[2];
```

其实，上述这两种方法是一样的，之所以你看到以 256 参与运算，这只是向补码转换为源码而已，有兴趣的话，你可以验证一下，对 8 为补码 n， 做取反加一，和 n-256，是否相同。

最关键的是我们读到的数据表示的是 x（或 y）方向的相对增量，其值用补码表示，至于你用哪一种解析，结果是一样的；

## 4、鼠标的绘制

通过上述对鼠标协议数据的解析，我们获取了鼠标的按键事件，和 x、y 方向的相对增量，那么我们只要声明一组变量，用来累加 dx，dy 即可，我们就可以获取鼠标在屏幕的绝对坐标了。

```
int m_x = 0; // 初始化，鼠标起点在原点，但通常初始化为屏幕中心
int m_y = 0;

while(1){
    // ... 读取鼠标数据，并解析
    m_x += mevent.dx;
    m_y += mevent.dy;
}
```

这样有了鼠标在屏幕上的位置（绝对坐标），只要在相应的位置绘制一个鼠标的标示，那么对鼠标的操控就完成了，

要做到这一地点需要解决如下几个问题：

🎨 要绘制的鼠标的表示是什么？

🎨 当鼠标移动到一个新的位置上的时候，除了要在新的位置上，绘制表示，还要在恢复旧位置的原有图像，如何做到？

我们可以定义一个鼠标的标示，在屏幕上绘制鼠标标示前，先保存屏幕相应位置（鼠标标示大小）的原有图像，而后在绘制鼠标的标示。那么当鼠标移动到一个新位置时，在原来位置就可以根据保存的原有图像恢复，而在新位置，继续重复保存、绘制即可，以后只要重复 恢复-保存-绘制，这三部曲即可。



## 5、完整的代码实例

```

#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

#include <fcntl.h>
#include <unistd.h>
#include "common.h"

/* ***** .h start ***** */
typedef struct
{
    int dx;                /* mice move left and right */
    int dy;                /* mice move up and down */
    int dz;                /* wheel */
    int button;
}mevent_t;

extern int mouse_open(const char *mdev);
extern int mouse_parse(int fd, mevent_t * mevent);

extern int mouse_draw(const pinfo_t fb, int x, int y);
extern int mouse_restore(const pinfo_t fb, int x, int y);

extern int mouse_test(pinfo_t fb);
/* ***** .h end ***** */

#define C_WIDTH  10
#define C_HEIGHT 17
#define T_____ 0xFFFFFFFF
#define BORD      0x0
#define X_____ 0xFFFF

static u32_t cursor_pixel[C_WIDTH * C_HEIGHT] = {
    BORD, T____, T____, T____, T____, T____, T____, T____, T____, T____,
    BORD, BORD, T____, T____, T____, T____, T____, T____, T____, T____,
    BORD, X____, BORD, T____, T____, T____, T____, T____, T____, T____,
    BORD, X____, X____, BORD, T____, T____, T____, T____, T____, T____,
    BORD, X____, X____, X____, BORD, T____, T____, T____, T____, T____,
    BORD, X____, X____, X____, X____, BORD, T____, T____, T____, T____,

```

```

    BORD, X____, X____, X____, X____, X____, BORD, T____, T____, T____,
    BORD, X____, X____, X____, X____, X____, X____, BORD, T____, T____,
    BORD, X____, X____, X____, X____, X____, X____, X____, BORD, T____,
    BORD, X____, X____, X____, X____, X____, X____, X____, X____, BORD,
    BORD, X____, X____, X____, X____, X____, BORD, BORD, BORD, BORD,
    BORD, X____, X____, BORD, X____, X____, BORD, T____, T____, T____,
    BORD, X____, BORD, T____, BORD, X____, X____, BORD, T____, T____,
    BORD, BORD, T____, T____, BORD, X____, X____, BORD, T____, T____,
    T____, T____, T____, T____, T____, BORD, X____, X____, BORD, T____,
    T____, T____, T____, T____, T____, BORD, X____, X____, BORD, T____,
    T____, T____, T____, T____, T____, T____, BORD, BORD, T____, T____
  };

static u32_t save_cursor[C_WIDTH * C_HEIGHT];

/* Mouse Test, demo for use mouse operation */
int mouse_test(pinfo_t fb)
{
    int fd;
    if((fd = mouse_open("/dev/input/mice")) < 0){
        fprintf(stderr, "Error mouse open:%s\n",
                strerror(errno));
        exit(1);
    }
    mevent_t mevent;

    /* mouse absolute coordinate */
    int m_x = fb->w / 2;
    int m_y = fb->h / 2;
    mouse_draw(fb, m_x, m_y);

    /* 向鼠标发送 " 命令信息 " ， 启动鼠标滚轮功能 */
    u8_t buf[] = {0xF3, 0xC8, 0xF3, 0x64, 0xF3, 0x50};
    if(write(fd, buf, sizeof(buf)) < sizeof(buf)){
        fprintf(stderr, "Error write to mice devie:%s\n",
                strerror(errno));
        fprintf(stderr, "鼠标将不支持滚轮\n");
    }

    while(1){
        if(mouse_parse(fd, &mevent) == 0){
            printf("dx= %d\tdy=%d\tdz=%d\t",
                    mevent.dx, mevent.dy, mevent.dz);

```

```

        mouse_restore(fb, m_x, m_y);

        m_x += mevent.dx;
        m_y += mevent.dy;

        mouse_draw(fb, m_x, m_y);
        printf("mx= %d\tny=%d\n",
               m_x, m_y);

        switch(mevent.button){
        case 1:
            printf("left button\n");
            break;
        case 2:
            printf("right button\n");
            break;
        case 4:
            printf("middle button\n");
            break;
        case 0:
            printf("no button\n");
            break;
        default:
            break;
        }
    }
    else
        ; /* Error read mouse */

}
close(fd);
return 0;
}

/* return mice device fd */
int mouse_open(const char *mdev)
{
    if (mdev == NULL)
        mdev = "/dev/input/mice";
    return (open(mdev, O_RDWR | O_NONBLOCK));
}

/* 0: read mouse Success
   * -1: error for read mouse

```

```

*/
#define READ_MOUSE 8
int mouse_parse(int fd, mevent_t * mevent)
{
    s8_t buf[READ_MOUSE];
    int n;
    if((n = read(fd, buf, READ_MOUSE)) > 0){
        /* 1: left key    * 2: right key
        * 4: middle key  * 0: no button key */
        mevent->button = buf[0] & 0x07;
        /* 获取 x, y, 滚轮的增量值 */
        mevent->dx = buf[1];
        mevent->dy = - buf[2];
        mevent->dz = buf[3];
    }
    else
        return -1;

    return 0;
}

static int mouse_save(const pinfo_t fb, int x, int y)
{
    int i,j;

    for(j = 0; j < C_HEIGHT; ++j)
        for(i = 0; i < C_WIDTH; ++i)
            save_cursor[i + j * C_WIDTH] =
                *(u32_t*)(fb->fb_mem + ((x+i) + (y+j) * fb->w)*fb->bpp/8);

    return 0;
}

int mouse_draw(const pinfo_t fb, int x, int y)
{
    int i,j;
    mouse_save(fb, x, y);

    for(j = 0; j < C_HEIGHT; ++j)
        for(i = 0; i < C_WIDTH; ++i)
            if(cursor_pixel[i + j * C_WIDTH] != T____)
                fb_pixel(fb, x+i, y+j,

```

```
                                &cursor_pixel[i + j * C_WIDTH]);

    return 0;
}

int mouse_restore(const pinfo_t fb, int x, int y)
{
    int i,j;

    for(j = 0; j < C_HEIGHT; ++j)
        for(i = 0; i < C_WIDTH; ++i)
            fb_pixel(fb, x+i, y+j,
                    &save_cursor[i + j * C_WIDTH]);

    return 0;
}
```

## 第 4 章 五子棋算法

---

课程内容：

- ✧ 引言
- ✧ 数据结构
- ✧ 设计算法
- ✧ 结论及展望未来

## 第 4 章 五子棋算法

### 1 引言

在人类文明发展的初期，人们便开始进行棋类博弈的游戏了。在人工智能领域内，博弈是很重要的一个研究分支，很多实际问题可以在博弈的研究中得到解决，并且使计算机智能更加靠近人类智能。电脑博弈是人工智能研究的一个方向，到了近 50 年前，随着电子计算机的诞生，科学家们开始通过电脑模拟人的智能逐步向人类智能发起挑战，香农(1950)与图灵(1953)提出了对棋类博弈程序的描述，随着电脑硬件和软件的高速发展，从 1980 开始，电脑博弈便开始逐渐大规模地向人的智能发起了挑战，到了 1997 年，IBM 超级电脑 Deeper Blue 击败了当时国际象棋世界冠军卡斯帕罗夫，成为了人工智能挑战人类智能发展的一个重要里程碑。

#### 1.1 电脑五子棋简介

五子棋是起源于中国古代的传统黑白棋种之一。现代五子棋日文称之为“连珠”，英译为“Ren-ju”，英文称之为“Gobang”或“FIR”(Five in a Row 的缩写)，亦有“连五子”、“五子连”、“串珠”、“五目”、“五目碰”、“五格”等多种称谓。五子棋不仅能增强思维能力，提高智力，而且变化多端，非常富有趣味性和消遣性，因此为人民群众所喜闻乐见。本文在研究博弈机器人系统过程中，对五子棋博弈算法进行了一些有效的研究，设计和实现一个人机对弈的五子棋程序。五子棋属于黑白棋的一种，它的博弈树复杂度为  $10^6$ ，状态空间复杂度为  $10^7$ 。因此盘面预测的计算量是非常大的，比如对于五子棋的中盘走法中，如果要预测四步的局面数的话可以达到一百万。

#### 1.2 设计思路总介绍

本文是对五子棋算法的设计原理和实现方法进行探讨和研究，主要包括数据结构、搜索算法和优劣评价函数组成，主要的特点包括快速的数据结构设计实现、以及高效率的搜索算法和尽可能的模拟人类的智能。



## 1.3 本文架构

第一章将会阐述计算机博弈发展以及对电脑五子棋的介绍、电脑五子棋的状态空间复杂度以及文章架构。

第二章首先描述了电脑象棋表示的一些基本数据结构。

第三章将重点讲述博弈树的搜索方法，包括博弈树构建、各算法介绍、以及五子棋规则的实现，最后对整体搜索框架进行总结。

最后，第四章是全文的总结及展望。

## 2 数据结构

### 2.1 棋局的表示

我们知道五子棋的走法中有优先和禁手，如连四肯定是没有三四优先，而如果是黑方出现三三（包括“四、三、三”）、四四（包括“四、四、三”），而黑方只能以四三取胜，如果黑方走出禁手则是输；五连与禁手同时形成，先五为胜，等等的规矩。但是电脑毕竟不是人类，可以类人但是却不可以自己思考，那么就需要给电脑一个它可以明白的评判标准。

首先介绍一个五子棋中的常用术语：

阳线：棋盘上可见的横纵直线。阴线：棋盘上无实线连接的隐形斜线。

活 3：由于一方走一着在无子交叉点上形成一个“活三”的局面，也就是在放一子就可行成 4 连了。

活 4：在棋盘某一条阳线或阴线上有同色 4 子不间隔地紧紧相连，且在此 4 子两端延长线上各有一个无子的交叉点与此 4 子紧密相连。

冲四：除“活四”外的，再下一着棋便可形成五连，并且存在五连的可能性的局面。

由于篇幅有限不能将所有的规则讲完，只是提出了对讲算法有用的几点加以叙述。

如何让电脑知道该落子在哪一点呢，在这方面。电脑要做得和人一样，判断棋盘上每一点的重要度。比如冲四比冲强，冲三比造二强，遇到四三如果是对方的，堵死，如果是自己的，优先落子。遇到双三，如果是黑棋，黑方输，如果是白棋，优先等级仅次于四三。我们看到电脑在运算的时候，同种棋型在敌我双方的优先等级并不相同，由于禁手的缘故黑棋和白棋的处理也不相同。同样是四三，如果是玩家的，则电脑不会冲自己的，而是先堵玩家的。禁手的处理比较复杂，我们稍候讨论。

下面，我列出基本的棋型优先顺序：

造一个二<.....<造四个二<冲三<.....<冲两个二和一个三<冲双三<冲四<冲四三。

之所以这么设置是由于五子棋的下法所致，只要构成 5 颗一线就赢了，所以说一条线上构成的越多那么就有优势，当然就是棋数越多，分越多。

那么为了让电脑明确的知道是不是应该落子，就给它一个标准：分值。用程序语言表示：

```
enum Value {FAILED=-99999, SKIP=20, LENG=10, TWO =100, THREE
=1000, IF0UR =10000, FOUR =50000, SF0UR=70000, WIN=100000};
```

如果某一点导致失败，则分值为 FAILED，由于它足够小，所以无论任何棋型与它的组合都小于 0，SKIP 之所以给它 20 分，是为了避免电脑出现无棋可下的情况，LENG 是有可能发生长连的点，这有可能导致禁手，TWO、THREE、FOUR、WIN 观名知意，TFOUR 和 SF0UR 分别是弱四和强四。强四的点比普通的冲四重要的多，比如一个活四，意味着即将判出胜负。弱四是为了提供更大的灵活。也许有某种冲四并不一定比冲三重要，我在这里并没有使用，以后可以扩充。

## 3 涉及算法

### 3.1 五子棋搜索树种的极大极小搜索

在五子棋乃至所有博弈类游戏的研究中[1-4]，最理想的方法是让电脑不论在对手走那种棋路的时候都可以根据对手的走法来预测下一步，以至于使电脑立于不败之地，但是五子棋博弈树的复杂度为  $10^6$ ，状态空间的复杂度为  $10^7$ ，因此即使电脑的运算速度已经很快了，又有强有力的启发式搜索技术，但是要完成如此复杂的搜索也是不可取的。这种情况下搜索深度可根据实际情况进行调整，从局部搜索树中选取一步最好的走步，等对方走步后再寻找下一步好棋，通过结果来判断以后的走法。我们可以通过极大极小搜索方法来实现这种搜索策略。

为了使谈论更有条理性[5]，将博弈的双方分别命名为 MAX 和 MIN。而搜索的任务是为 MAX 找最佳的移动。假设 MAX 先移动(此时暂时不考虑五子棋的禁手等规则)，然后 2 个博弈者轮流移动。因此，深度为偶数的节点，对应于 MAX 下一步移动的位置，称为 MAX 节点；深度为奇数的节点对应于 MIN 下一步移动的位置，称为 MIN 节点(博弈树的顶节点深度为 0)。k 层包括深度为 2k 和 2k+1 的节点。通常用层数表示博弈树的搜索深度。他可以表示出向前预测的 MAX 和 MIN 交替运动的回合数。对于复杂的博弈，博弈者必须认识到由于博弈树的复杂程度所以搜索到终点是不可能的(除了在博弈快结束时)。所以，常采用在有限

范围搜索方法，这里使用启发式搜索。在启发式搜索的过程中，关键的一步是如何确定下一个要考察的节点，在确定节点时只要能充分利用与问题有关的信息，估计出节点的重要性，就能在搜索时选择重要性较高的节点，以利于博弈者以较快的速度求出最佳的棋步。

在这里我采用静态评估函数[6-13]，用评估函数  $h(i)$  衡量每一个叶节点位置的“值”。一个最佳首步可以由一个最小最大化过程产生。假设轮到 MAX 从搜索树的叶节点中选取，他肯定选择拥有最大值的节点。因此，MIN 叶节点的一个 MAX 节点双亲的倒推值就等于叶节点的静态评估值中的最大值。另一方面，MIN 从叶节点中选取时，必然选取最小的节点(即最负的值)。既然如此，MAX 叶节点的 MIN 双亲节点被分配一个倒推值，他等于叶节点静态评估值的最小值。在所有叶节点的父节点被赋予倒推值后，开始倒推另一层，假定 MAX 将选择有最大倒推值的 MIN 的后继节点，而 MIN 会选择有最小倒推值的 MAX 后继节点。继续逐层对节点评估，直到最后开始节点的后继者被赋予倒推值。MAX 将选择有最大倒推值的节点作为他的首步。整个过程的有效性基于这样的假设。用整个棋盘估值的函数  $h(n)$  为静态估值函数。设想当前棋局 S 为轮到计算机方下棋(用方框表示)，任选一空点作为计算机方的下棋位置(可有若干种选择)，接着考虑在此情况下游戏者一方下棋的棋局(用圆圈表示)；从某一个圆圈棋局出发，任选一空点作为游戏者一方的落子处(又有若干种选择)。再次形成计算机方下棋的方框棋局；依此类推，这样可形成一棵以 S 为根结点的博弈树，该树以圆圈棋局为第 2 层子结点，以方框棋局为第 3 层子结点等等。如果继续向前搜索，可形成多层子结点，现在以向前搜索 3 层子结点为例来说明极大极小值的过程。对第 3 层子结点的某一棋局 n，求出其估计值  $h(n)$ ，假设有一博弈树已形成，如图 1 所示[2]， $h(n)$  的值由各结点旁的数值给出。

根据极小极大化分析法，先计算第 3 层子结点  $h(n)$  值，然后第 2 层子结点的估计值取他的各后继子结点的极小值，根结点的估计值取他的各子结点的极大值。这个取得最大估计值的子结点即为从 S 出发的计算机方的最佳落子方案。棋盘上某一行、某一列或某一对角线为一路，这里使用的棋盘为 19 行 19 列，因此，行和列方向上共有  $19+19=38$  路；从左下到右上方向的对角线有 37 路，同样，从左上到右下方向的对角线也有 37 路。但对于五子棋来说必须在一条直线上有连续五个棋子才能赢。因此，在对角线上就可以减少 8 路。所以，整个棋盘路的总数为： $38+(37-8)+(37-8)=96$  路。

对某一棋局 n[14-15]，第 i 路得分： $h(i)=h(i)-h(i)[1]$ 。其中： $h(i)$  为计算机方在第 i 路估计值得分， $h(i)$  为玩家一方在第 i 路得分。对的得分规则作如下说明。规则中+代表一空点；O 代表对方棋子；\*代表有计算机方棋子；++表示连

续 2 点为空点；+++\*++表示连续 3 个空点接一个计算机方棋子，再接两空点。为了使电脑可以对棋局进行判断，那就需要告诉电脑什么时候的局势好过什么时候的局势，我们可以通过给不同局势时不同分值的办法来让电脑明白。我给出的局势评估如下：

+++\*++:  $h(i)=10$ ;

++\*\*++:  $h(i)=100$ ;

+\*\*\*++:  $h(i)=1000$ ;

O\*\*\*\*+:  $h(i)=10000$ ;

+\*\*\*\*\*:  $h(i)=50000$ ;

\*\*\*\*\*的时候落子方都已经是胜利了： $h(i)=100000$ （之所以给分 200 是为了以后其他棋局的时候好判断）。之所以要给出如此大的分值差距，主要是考虑以后的全局判断的时候不会因为分数的累加使得电脑判断错误。而电脑在判断双方的分值的时候，就是在落子时对自己的落子点涉及分值和对方的分值进行对比来进行之后落子的判断。

## 3.2 $\alpha$ - $\beta$ 剪枝

我们不用把每个节点都搜索一遍也可获得和极大极小搜索同样结果的走步，不搜索分支节点而舍弃该分支的过程叫剪枝。常用的剪枝方法是  $\alpha$ - $\beta$  剪枝算法。

在极大极小搜索的过程中，存在着一定程度的数据冗余。如下图 2 所示左半部的一棵极大极小树的片断，节点下面为该节点的值，设 A 为极大值节点，节点 B 的值为 18，节点 D 的值为 16，由此可以判断节点 C 的值将小于等于 16(取极小值)；而节点 A 的值为节点  $\text{Max}(B, C)$ ，是 18，也就是说不再需要估节点 C 的其他子节点如 E、F 的值就可以得出父节点 A 的值了。这样将节点 D 的后继兄弟节点减去称为 Alpha 剪枝( $\alpha$  剪枝)。如图 2:

同样道理在图 3 右半部一棵极大极小树的片段中，设 A 为极小值节点，节点 B 的估值为 8，节点 D 的估值为 18，由此可以判断节点 C 的值将大于等于 18(取极大值)；而节点 A 的值为  $\text{Min}(B, c)$ ，是 8。也就是说不再要求节点 C 的其他子节点如 E、F 值就可以得出父节点 A 的值了。这样将节点 D 的后继兄弟节点剪去称为 Beta 剪枝( $\beta$  剪枝)。如图 3:

将 Alpha 剪枝和 Beta 剪枝加入极大极小搜索，就得到  $\alpha$ - $\beta$  搜索算法。

经过了上面的介绍，我们再回到五子棋问题上来。

上边讲到了极大极小搜索，以及搜索时产生子结点的顺序。五子棋的静态估

值函数有多种方法得到，具体如何实现除了要有一定的五子棋知识外，还要能运用这些知识正确地给出对一个五子棋当前局面的估值。可以进行全局搜索，对棋盘的每一个点判断棋形计算得分，最后综合得到全局得分，优先选择分值最高的走步。若对五子棋比较熟悉，可以在程序中规定几个下棋定式，根据定式判断当前局势得分。

根据极大极小搜索中给出的判断棋形所设的得分值，搜索整个棋盘得出共有多少个活一，死一，活二，……最后把各部分得分求和，得到当前局势的评价值。这个静态估值函数的好坏直接决定了程序智能的高低。可以在实践中不断调整得分值，使其智能更高。

而对于搜索时产生子结点的顺序，这关系到程序运行的快慢，倘若在  $\alpha$ - $\beta$  剪枝过程中，很早就得到一个较大的值，则在其后的搜索过程中，剪枝数目增加，程序效率也会提高。产生子结点的顺序，光靠说不行，还需要在实际中慢慢体会，得出一个较优方案。在这里我暂且假设它从(0, 0)点开始到(14, 14)为止产生结点，对于每一次预测走步，我们假设白棋从(0, 0)开始的第一个无子的点开始预测(假设电脑为白棋)，再轮到黑棋走，再轮到白棋走，一直向后预测  $n$  步后用静态估值函数对整个局面打分，再选择另一个位置，再为局面打分，重复此过程对深度为  $n$  的搜索树进行  $\alpha$ - $\beta$  剪枝搜索。

现在假设五子棋问题深度为 3 的搜索树给出  $\alpha$ - $\beta$  剪枝算法，用类 C 语言对其进行描述：

设电脑为 Max，人为 Min，初始时  $\alpha$  为  $-\infty$ ， $\beta$  为  $+\infty$ ；函数 Evaluate() 返回对当前局面的估值

```
int Max(point p,  $\alpha$ ,  $\beta$ )
{
    .....
    在 p 处下白子；
    if(已到达搜索的最后一层) return Evaluate();
    for(int i=0; i<15; i++)
        for(int j=0; j<15; j++)
            {point pt=(i, j);
                 $\alpha$ =max ( $\alpha$ , Min(pt,  $\alpha$ ,  $\beta$ ))
            }
    if( $\alpha$ >  $\beta$ ) return  $\beta$ ;)
    return  $\alpha$ ;)
MIN 结点的  $\beta$  剪枝函数：
int Min (point p,  $\alpha$ ,  $\beta$ )
{
    .....
    在 p 处下黑子；
```



```

if(已到达搜索的最后一层) return Evalute();
for(int i= 0; i< 15; i++ )
for(int j=0; j< 15; j++ )
{point pt=(i,j);
 $\beta$ =min( $\beta$ , Max (pt,  $\alpha$ ,  $\beta$ ))
if( $\beta$ <= $\alpha$ ) return  $\alpha$ ;)
returnp  $\beta$ ;)
  
```

如果 Max(point p, $\alpha$ , $\beta$ )函数, 则返回对 Max 结点最有利的走步的局势静态估值函数值。反之 Min (point p,  $\alpha$ ,  $\beta$ )函数, 则返回对 Min 结点最有利的局势静态估值函数值。两种互为递归调用, 可以动态改变搜索深度。

### 3.3 优化估值函数

通过以上的研究可以看出, 在博弈的程序中电脑的行为都是以估值函数为依据的, 所以说估值函数在很大的程度上是决定了电脑的棋力高低。我们可以采用遗传算法来改进静态估值。遗传算法的估计值在很大的程度上也依赖于实施者的经验。但是可以利用一些高水平名局棋谱或同其他博弈程序对弈, 看使用某一组参数时取胜的机率有多大来进行较验。经过几次的试验一般就可以得到较好的静态估值。

传统的算法一般只能维护一组较好的参数。遗传算法同传统的算法相比可以同时维护几组较好的参数, 通过向其中添加一组新参数, 通常是将几组老参数中选出的值组合在一起做一点变化。然后同几组老的参数比较孰优孰劣, 将最差的一组从中去除。这里面较重要的操作是交叉和变异操作。交叉通过随机交换父代个体的信息来继承已有参数中的优良内容; 变异通过随机改变个体中的基因而增加种群的多样性, 避免优化的因局部震荡而失败。

遗传算法的优点:

(1)由于搜索算法是从问题的解开始的, 而不是一组参数。所以被局部震荡干扰导致求最优解失败的可能性大大减小。

(2)此算法能将搜索重点集中于性能高的部分, 能较快地求出最佳的参数。

遗传算法的优化估值参数的过程: 首先是①随机产生几组参数作为初始种群; 接着②对种群的参数进行收敛判断, 收敛则③输出优化结果并且评估过程结束, 否则就④执行复制操作产生一组新参数; 然后⑤取 0、1 之间的随机数, 让随机数和交叉概率进行比较; 若大于⑥则执行交叉操作改变新参数, 若小于就不用执行这一步; ⑦接着取 0、1 之间的随机数, 随机数和变异概率进行比较, 若⑧大于就执行变异操作改变新参数, 小于就不执行这一步; 接着是⑨把较差的一

组参数从种群中去除；接着跳回第二步判断是否已经收敛，如此循环就能将搜索重点集中于性能高的部分，能较快地求出最佳的参数。画图 2 表示如下：

### 3.4 禁手特征计算

五子棋中还有项规则就是是禁手的判断，在上边已经给出了 3 颗棋子的时候的分值为 1000。如果是双三则需要乘以 2，即 2000。电脑在分值表中将会判断双三点比一个三的点更为重要。这在电脑进行全局判断的时候也是正确的，可是如果电脑执黑子，这一点是不能落子的，否则将导致失败。但是电脑不能真的会自己去判断，只有认为的给他一个判断的标准，处理的方法是：在累加分值的时候增加一个判断语句，如果正好形成双三，那么这一点不加入分值表同时判断此点落棋将判为负，其他禁手按同样的方法处理。

## 4 结论及展望未来

### 4.1 总结

本文介绍了应用人工智能设计五子棋的总体方法。用极大极小值的过程以及启发式搜索的方法，采用静态估值函数对各节点的代价进行估计，应用遗传算法对估计的值进行了优化，克服了人为主观因素的片面性。对博弈的研究具有一定的借鉴意义。在实际五子棋系统的设计中。应用本方法只向前搜索了三步，就可以取得很好的效果。也可以增加搜索的深度来提高系统的判断能力，但是是以牺牲电脑的运算速度为前提的，毕竟加深搜索的话就需要更大的运算量。此外，也可以通过增加机器学习，比如电脑对棋局进行记忆，在对手落子之前对棋局和之前记忆的棋局进行比较，先判断出更优的走法，就可以进一步提高系统的智能。

### 4.2 未来展望

随着 Intel HP(超线程技术)的实现和将来多处理器 PC 机的普及. 对于数据计算量大的人机对弈问题必然要求应用并行的思想去处理。超快搜索速度和必要的复盘“反思”必然带给下棋电脑更多智慧。



## 第 5 章 图像生成

---

课程内容：

- ✧ 图形图像知识基础
- ✧ 直线的生成算法
- ✧ 圆的生成算法
- ✧ 图片格式简介
- ✧ **libjpeg** 库简介

## 第 5 章 图像生成

### 1、图形图像知识基础

#### 1.1 图形图像基础学习资料

##### 图形 (graphic):

和图像与视频不同，有一种说法是图形就是自然界的客观世界不存在的图案。对于计算机中的图形研究，有专门的计算机图形学，主要的研究对象是点、线、面等抽象事物。目前所谓的计算机显卡 3D 技术支持，主要就是图形技术相关的范畴。关于图形方面的开发，好象 OpenGL 是其中比较有名的 3D 图形库。

##### ✓ 图像 (image):

和图形相反，图像可以定位为自然界中客观存在的图案。图像处理和我们有关系的大致是图像滤波处理和图像压缩。目前用得最多的静止图像压缩算法就是 jpeg 了，大家应该都很熟悉。而对图像的其他处理，一般称之为对图像进行滤波，图像处理方面，photoshop 软件很多人应该都很熟悉，它图像处理的功能十分强大。在视频行业，主要是关注消隔行滤波器、去除摄像头白噪声滤波器、去除块效应 (deblock) 滤波器等。

##### ✓ 视频 (video):

视频我的理解就是连续的图像，被称为视频。对视频图像的处理，核心是压缩，其他的就是采集、传输、显示和录像了。视频图像如果不压缩的话，传输和录像的成本都太高了。

##### ✓ RGB 格式:

RGB 指的是红绿蓝，应用还是很广泛的，比如显示器显示，BMP 文件格式中的像素值等；而 YUV 主要指亮度和两个色差信号，被称为 luminance 和 chrominance 他们的转化关系可以自己去看一下，我们视频里面基本上都是用 YUV 格式。

### ✓ YUV 格式:

YUV 文件格式又分很多种, 如果算上存储格式, 就更多了, 比如 YUV444、YUV422、YUV411、YUV420 等等, 视频压缩用到的是 420 格式, 这是因为人眼对亮度更敏感些, 对色度相对要差些。另外要注意几个英文单词的意思, 比如: packet、planar、interlace、 progressive 等。

### ✓ 帧率

每秒钟图像的刷新速度。PAL 制式的电视, 帧率是 25 帧每秒, NTSC 制式的电视帧率是 29.97 帧每秒。我们常用的电脑也有刷新率, 一般来说, 电脑的刷新率要在 75 赫兹以上, 人眼才不会觉得闪。

### ✓ 隔行扫描(interlace)和逐行扫描(progressive)

一般的电视上都是隔行扫描, 而显示器都是逐行扫描。这里有一个场的概念, 隔行扫描是一帧等于两场, 而逐行扫描则是一帧就是一场。

### ✓ 码率

它的单位是 bit per second, 一般所有描述带宽的概念, 单位都是 bit, 描述存储容量的单位一般都是大 B, 也就是 BYTE (字节)。

### ✓ 分辨率

图像的分辨率指的是它的像素数, 一般用得最多的是 CIF, 也就是 352\*288, 4cif 自然就是指 704\*576, 而 D1 的分辨率严格意义上是 720\*576, 大小来说和 4cif 差不多了。当然现在还有很多高清的分辨率, 这些我不是太了解, 大家感兴趣可以查一下。另外, 国外很多时候, 对 cif 的高度取 240, 这是因为他们的帧率比我们高 (29.97hz), 自然, 高度要小一些了。

### ✓ 实时与非实时

主要用来形容编码器, 它含有两个意思, 一个是要保证帧率, 也就是每秒 25 帧, 另一个是“live”的意思, 意味着直播, 所谓的“实况转播”的“实”。

### ✓ 延时

也是形容编码器的一个重要指标，一般来说，200ms 到 300ms 人的感觉不会很明显，到了 500 毫秒的话，还是可以很明显感觉到的。

### ✓ 音视频同步

作为视频会议的应用，一般要求做到所谓的“唇同步”。基本的保证音视频同步的手段就是时间戳（time stamp）。

### ✓ 复合视频和 S-Video

NTSC 和 PAL 彩色视频信号是这样构成的--首先有一个基本的黑白视频信号，然后在每个水平同步脉冲之后，加入一个颜色脉冲和一个亮度信号。因为彩色信号是由多种数据“叠加”起来的，故称之为“复合视频”。S-Video 则是一种信号质量更高的视频接口，它取消了信号叠加的方法，可有效避免一些无谓的质量损失。它的功能是将 RGB 三原色和亮度进行分离处理。

### ✓ NTSC、PAL 和 SECAM

基带视频是一种简单的模拟信号，由视频模拟数据和视频同步数据构成，用于接收端正确地显示图像。信号的细节取决于应用的视频标准或者“制式”--NTSC（美国全国电视标准委员会，National Television Standards Committee）、PAL（逐行倒相，Phase Alternate Line）以及 SECAM（顺序传送与存储彩色电视系统，法国采用的一种电视制式，SEquential Couleur Avec Memoire）。

中国的电视信号一般都是 PAL，而美日则是 NTSC。这 2 个制式的帧率，图像尺寸都有所不同。

### ✓ 线数

我们在买摄像头的时候，经常会提到一个叫线数的概念，它其实就是分辨率中的高（height）。举个例子：PAL 制式的 D1 图像，线数就是 576。

## ✓ 亮度、饱和度和对比度

英文名分别是：brightness、saturation 和 contrast。这是三个表示图像的重要指标。

## ✓ 数据压缩

数据压缩，通俗地说，就是用最少的数码来表示信号。其作用是：能较快地传输各种信号，如传真、Modem 通信等；在现有的通信干线并行开通更多的多媒体业务，如各种增值业务；紧缩数据存储容量，如 CD-ROM、VCD 和 DVD 等；降低发信机功率，这对于多媒体移动通信系统尤为重要。由此看来，通信时间、传输带宽、存储空间甚至发射能量，都可能成为数据压缩的对象。

## ✓ 为何进行数据压缩

如今在 Internet 上，传统基于字符界面的应用逐渐被能够浏览图象信息的万维网方式所取代。尽管图片漂亮，但是也带来了一个问题：图象信息的数据量太大，使得本来就非常紧张的网络带宽变得更加不堪重负，使得 World Wide Web 变成了 World Wide Wait。

总之，大数据量的图象信息会给存储器的存储容量，通信干线信道的带宽，以及计算机的处理速度增加极大的压力。单纯靠增加存储器容量，提高信道带宽以及计算机的处理速度等方法来解决这个问题是不现实的，这时就要考虑压缩。

图片信息因其数据量大，在进行传输时必将占据宝贵的网络资源，如何在保证传输质量的前提下减少传输数据，就成为一项亟待解决的课题。

在图片中往往存在色彩均匀的背景信息等相关数据，因此，有可能利用某些变换来尽可能地去掉这些相关性，来减少图片数据量，此种方法也被称为图片压缩技术。

## ✓ 像素

通常所说的像素，就是 CCD 上光电感应元件的数量，一个感光元件经过感光，光电信号转换，A/D 转换等步骤以后，在输出的照片上就形成一个点，我们如果把影像放大数倍，会发现这些连续色调其实是由许多色彩相近的小方点所组成，这些小方点就是构成影像的最小单位“像素”--pixel。

## ✓ 分辨率

图像的分辨率是指单位长度上的像素数量，即直观看到的图像的清晰与模糊程度。分辨率可分为多种，如显示分辨率、屏幕分辨率等。

显示分辨率用于确定屏幕上显示图像的区域的大小。显示分辨率有最大显示分辨率和当前显示分辨率之分。最大显示分辨率是由物理参数，即显示器和显示卡决定的；而当前显示分辨率是由当前设置的参数决定的。


图像分辨率用于确定组成一幅图像的像素数目，是组成一幅图像的像素密度的度量方法，图像分辨率用每英寸点数(dpi)表示。对同样大小的一幅原图，如果数字化时图像分辨率高，则组成该图的像素点数目越多，看起来就越逼真。图像分辨率在图像输入/输出时起作用，它决定图像的点阵数。不同的分辨率会产生不同的图像清晰度。

### ✓ 常见压缩算法简介


压缩可分为两大类：第一类压缩过程是可逆的，也就是说，从压缩后的图像能够完全恢复出原来的图像，信息没有任何丢失，称为无损压缩；第二类压缩过程是不可逆的，无法完全恢复出原图像，信息有一定的丢失，称为有损压缩。


图像压缩一般通过改变图像的表示方式来达到，因此压缩和编码是分不开的。图像压缩的主要应用是图像信息的传输和存储，可广泛地应用于广播电视、电视会议、计算机通讯、传真、多媒体系统、医学图象、卫星图象等领域。

压缩编码的方法有很多，主要分成以下四大类：

 像素编码

 预测编码

 变换编码

 其它方法

所谓像素编码是指，编码时对每个像素单独处理，不考虑像素之间的相关性。

所谓预测编码是指，去除相邻像素之间的相关性和冗余性，只对新的信息进行编码。举个简单的例子，因为像素的灰度是连续的，所以在一片区域中，相邻像素之间灰度值的差别可能很小。如果我们只记录第一个像素的灰度，其它像素的灰度都用它与前一个像素灰度之差来表示，就能起到压缩的目的。

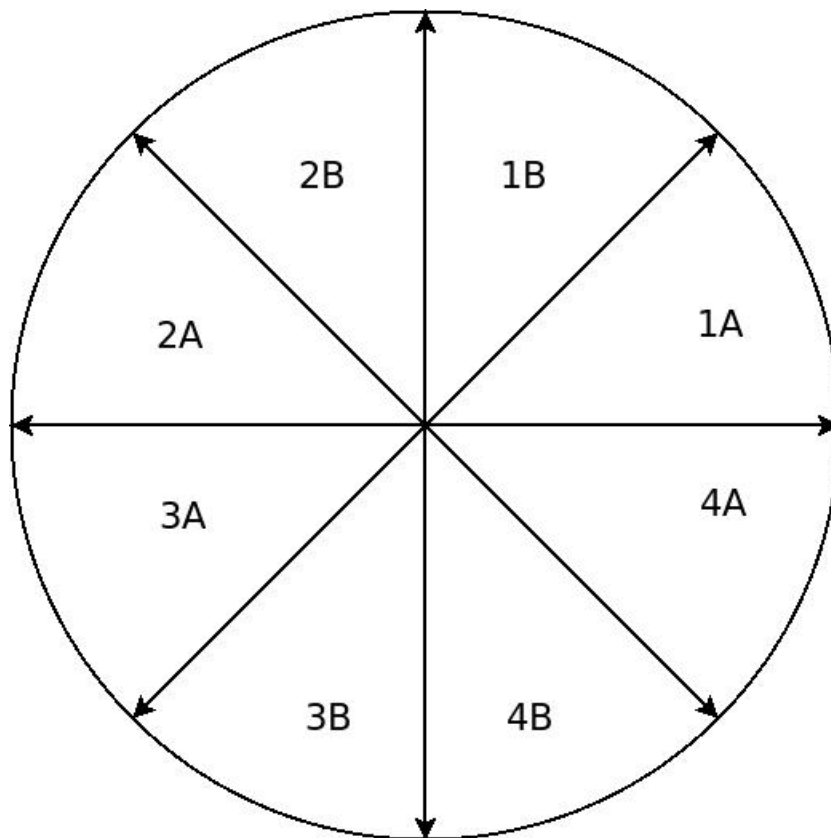
所谓变换编码是指，将给定的图象变换到另一个数据域(如频域)上，使得大量的信息能用较少的数据来表示从而达到压缩的目的。

值得注意的是，近些年来出现了很多新的压缩编码方法，如使用人工神经网络(Artificial Neural Network，简称 ANN)的压缩编码算法、分形(Fractal)、小波(Wavelet)、基于对象(Object Based)的压缩编码算法、基于模型(Model-Based)的压缩编码算法(应用在 MPEG4 及未来的视频压缩编码标准中)。

## 2、直线的生成算法

### 2.1 简介

在光栅显示器的荧光屏上生成一个对象，实质上是往帧缓存寄存器的相应单元中填入数据。画一条从  $(x_1, y_1)$  到  $(x_2, y_2)$  的直线，实质上是一个发现最佳逼近直线的象素序列，并填入色彩数据的过程。这个过程也称为直线光栅化。



直线方向的 8 个象限

### 2.2 直线的 Bresenham 算法

本算法由 Bresenham 在 1965 年提出。设直线从起点  $(x_1, y_1)$  到终点  $(x_2, y_2)$ 。直线可表示为方程  $y=mx+b$ 。其中  $b = y_1 - m * x_1$ ,  $m = (y_2 - y_1) / (x_2 - x_1) = dy/dx$

我们的讨论先将直线方向限于 1a 象限（图 2.1.1）在这种情况下，当直线光栅化时， $x$  每次都增加 1 个单元，即



$x_{i+1}=x_i+1$ 。而  $y$  的相应增加应当小于 1。为了光栅化,  $y_{i+1}$  只可能选择如下两种位置之一。

$y_{i+1}$  的位置选择  $y_{i+1}=y_i$  或者  $y_{i+1}=y_i+1$ 。选择的原则是看精确值  $y$  与  $y_i$  及  $y_{i+1}$  的距离  $d_1$  及  $d_2$  的大小而定。计算式为:

$$y=m(x_{i+1})+b \quad (2.1.1)$$

$$d_1=y-y_i \quad (2.1.2)$$

$$d_2=y_{i+1}-y \quad (2.1.3)$$

如果  $d_1-d_2>0$ , 则  $y_{i+1}=y_i+1$ , 否则  $y_{i+1}=y_i$ 。因此算法的关键在于简便地求出  $d_1-d_2$  的符号。将式 (2.1.1)、(2.1.2)、(2.1.3) 代入  $d_1-d_2$ , 得

$$d_1-d_2=2y-2y_i-1=2 \left( \frac{dy}{dx} \right) (x_{i+1})-2y_i+2b-1$$

用  $dx$  乘等式两边, 并以  $P_i=dx(d_1-d_2)$  代入上述等式, 得

$$P_i=2x_idy-2y_idx+2dy+dx(2b-1) \quad (2.1.4)$$

$d_1-d_2$  是我们用以判断符号的误差。由于在 1a 象限,  $dx$  总大于 0, 所以  $P_i$  仍旧可以用作判断符号的误差。 $P_{i-1}$  为:

$$P_{i+1}=P_i+2dy-2dx(y_{i+1}-y_i) \quad (2.1.5)$$

误差的初值  $P_1$ , 可将  $x_1, y_1$ , 和  $b$  代入式 (2.1.4) 中的  $x_i, y_i$  而得到:

$$P_1=2dy-dx$$

综述上面的推导, 第 1a 象限内的直线 Bresenham 算法思想如下:

1. 画点( $x_1, y_1$ );  $dx=x_2-x_1$ ;  $dy=y_2-y_1$ ;

计算误差初值  $P_1=2dy-dx$ ;  $i=1$ ;



2. 求直线的下一点位置:

$x_{i+1}=x_i+1$ ;

if  $P_i > 0$  则  $y_{i+1}=y_i+1$ ;

否则  $y_{i+1}=y_i$ ;

3. 画点  $(x_{i+1}, y_{i+1})$ ;

4. 求下一个误差  $P_{i+1}$ ;

if  $P_i > 0$  则  $P_{i+1}=P_i+2dy-2dx$ ;

否则  $P_{i+1}=P_i+2dy$ ;

5.  $i=i+1$ ; if  $i < dx+1$  则转 2; 否则 end。

Bresenham 算法的优点是:

1. 不必计算直线之斜率, 因此不做除法;
2. 不用浮点数, 只用整数;
3. 只做整数加减法和乘 2 运算, 而乘 2 运算可以用硬件移位实现。

Bresenham 算法速度很快, 并适于用硬件实现。

由上述算法思想编制的程序如程序 2.1.2。这个程序适用于所有 8 个方向的直线(图 2.1.1)的生成。程序用色彩 C 画出一条端点为  $(x_1, y_1)$  和  $(x_2, y_2)$  的直线。其中变量的含义是: P 是误差; const1 和 const2, 是误差的逐点变化量; inc 是 y 的单位递变量, 值为 1 或 -1; tmp 是用作象限变换时的临时变量。程序以判断  $|dx| > |dy|$  为分支, 并分别将 2a, 3a 象限的直线和 3b, 4b 象限的直线变换到 1a, 4a 和 2b, 1b 方向去, 以求得程序处理的简洁。

上述思路来自网上:

依据以上思想实现的程序如下

```

int fb_line(const pfb_info pfb_inf, int x1, int y1, int x2, int y2, u32_t color)
{
    int dx = x2 - x1;
    int dy = y2 - y1;
    int p;
    int inc;

    /* 准备 x 或 y 的单位递变值 */
    inc = ((dx * dy) >= 0) ? 1 : -1;

    if (abs(dx) > abs(dy)){          /* dx > dy 位于 A 区域 */
        if (dx < 0){ // 将 2a, 3a 象限方向的直线转换到 1a, 4a */
            swap(&x1, &x2);    swap(&y1, &y2);
            dx = -dx; dy = -dy;
        }
        // 注意此时的误差变化参数取值.
        dy = (dy > 0) ? dy : -dy;
        p = 2 * dy - dx;
        while (x1++ <= x2){
            fb_pixel(pfb_inf, x1-1, y1, (u8_t*)&color);
            // 注意此时的误差变化参数取值. */
            if (p < 0)    p += 2 * dy;
            else{
                y1 += inc;
                p += 2 * (dy - dx);
            }
        }
    }
    else { // dy > dx 位于 B 区域 */
        if (dy < 0){ // 将 3b, 4b 象限方向的直线变换到 2b, 1b 象限方向去. */
            swap(&x1, &x2);    swap(&y1, &y2);
            dx = -dx; dy = -dy;
        }
        dx = (dx > 0) ? dx : -dx;
        // 注意此时的误差变化参数取值. */
        p = 2 * dx - dy;
        while (y1++ < y2){
            fb_pixel(pfb_inf, x1, y1-1, (u8_t*)&color);
            if(p<0)    p += 2 * dx;
            else{
                x1 += inc;
                p += 2 * (dx - dy);
            }
        }
    }
}

```

```

    }
  }
  return 0;
}

```

## 3、圆的生成算法

### 3.1 Bresenham 画圆算法描述

以下算法描述，用于绘制八分之一圆弧，其余类似，

1. 求误差初值:

\*  $p_1 = 3 - 2r$ ;

\*  $i = 1$ ;

2. 画点( $x(i)$ ,  $y(i)$ );

3. 求下一个光栅位置:

\*  $x(i+1) = x(i) + 1$ ;

\* 而  $y(i+1)$  的取值则有两种可能:

o  $p_i < 0$ ; 则  $y(i+1) = y(i)$ ;

o  $p_i \geq 0$ ; 则  $y(i+1) = y(i) - 1$ ;

4. 计算下一个误差:

\*  $p_i < 0$ ; 则  $p(i+1) = p(i) + 4 * x(i) + 6$ ;

\*  $p_i \geq 0$ ; 则  $p(i+1) = p(i) + 4 * (x(i) - y(i)) + 10$ ;

5.  $i++$ ;

\* if ( $x == y$ ) 则 end;

\* if ( $x \neq y$ ); 返回到 第 2 步。

算法优点:

计算量小，效率较高

实现代码如下:

至于绘制实心圆，还是绘制空心圆，只要传入 `circle_line`，或者 `circle_point` 函数，就 ok 了

```

/*
 * 同时绘制 关于 x 轴, y 轴对称 4 个点: (x, y); (x, -y); (-x, y); (-x, -y);
 * x0 y0: 圆心坐标
 * x, y: 圆周上的点坐标
 */
static inline int fb_pixel_symmetric(const pinfo_t fb, int x0, int y0, int x, int y, u32_t c)
{
    fb_pixel(fb, x0 + x, y0 + y, &c);
    fb_pixel(fb, x0 + x, y0 - y, &c);
    fb_pixel(fb, x0 - x, y0 + y, &c);
    fb_pixel(fb, x0 - x, y0 - y, &c);

    return 0;
}

/*
 * draw point in circle, draw hollow circle
 * 同时绘制关于 |x| = |y| 对称的点
 * x0 y0: 圆心坐标
 * x, y: 圆周上的点坐标
 */
int circle_point(const pinfo_t fb, int x0, int y0, int x, int y, u32_t color)
{
    fb_pixel_symmetric(fb, x0, y0, x, y, color);
    fb_pixel_symmetric(fb, x0, y0, y, x, color);
    return 0;
}

/*
 * draw line in circle, draw solid circle
 * x0, y0, 圆心坐标
 * x, y 圆周上坐标
 */
int circle_line(const pinfo_t fb, int x0, int y0, int x, int y, u32_t color)
{
    fb_line_x(fb, x0-x, y0+y, 2*x, color);
    fb_line_x(fb, x0-x, y0-y, 2*x, color);

    fb_line_x(fb, x0-y, y0+x, 2*y, color);
    fb_line_x(fb, x0-y, y0-x, 2*y, color);
  
```

```

    return 0;
}

/*
 * draw circle, use Bresenham
 * Hollow circle, circle_point
 * solid circle, circle_line
 */
int fb_circle(const pinfo_t fb, int x0, int y0, int r, u32_t color, func_t circle)
{
    int x = 0;
    int y = r;
    int p = 3 - 2 * r;

    while (x <= y) { /*画 八分之一 的圆弧*/
        circle(fb, x0, y0, x, y, color);
        if (p < 0) {
            p += 4 * x + 6;
        }
        else{
            p += 4 * (x - y) + 10;
            y--;
        }
        x++;
    }
    return 0;
}

```

## 4、常见的图片格式

### 4.1 jpeg 图片格式

JPEG 是 Joint Photographic Experts Group(联合图像专家组)的缩写,文件后缀名为".jpg"或".jpeg",是最常用的图像文件格式,由一个软件开发联合会组织制定,是一种有损压缩格式,能够将图像压缩在很小的储存空间,图像中重复或不重要的资料会被丢失,因此容易造成图像数据的损伤。尤其是使用过高的压缩比例,将使最终解压缩后恢复的图像质量明显降低,如果追求高品质图像,不宜采用过高压缩比例。但是 jpeg 压缩技术十分先进,它用有损压缩方式去除冗余的图像数据,在获得极高的压缩率的同时能展现十分丰富生动的图像,换句话说,

就是可以用最少的磁盘空间得到较好的图像品质。而且 jpeg 是一种很灵活的格式，具有调节图像质量的功能，允许用不同的压缩比例对文件进行压缩，支持多种压缩级别，压缩比率通常在 10: 1 到 40: 1 之间，压缩比越大，品质就越低；相反地，压缩比越小，品质就越好。比如可以把 1.37Mb 的 BMP 位图文件压缩至 20.3KB。当然也可以在图像质量和文件尺寸之间找到平衡点。JPEG 格式压缩的主要是高频信息，对色彩的信息保留较好，适合应用于互联网，可减少图像的传输时间，可以支持 24bit 真彩色，也普遍应用于需要连续色调的图像。

jpeg 文件大体上可以分成以下两个部分：标记码(tag)加压缩数据。先介绍标记码部分。

标记码部分给出了 jpeg 图像的所有信息，如图像的宽、高、Huffman 表、量化表等等。标记码有很多，但绝大多数的 jpeg 文件只包含几种，其中标记码 SOI(Start of Image)的结构如下：

标记结构	字节数
0XFF	1
0XD8	1

可作为 jpeg 格式的判据。jpeg 文件数据内容如图 3.3.3 所示：

```

00000000h: FF D8 FF EO 00 10 4A 46 49 46 00 01 02 00 00 64 ; 00000001h: 00 64 00 00 FF EC 00 11 44 75 63 6B 79 00 01 00 ; .d.. ?..Ducky...
00000002h: 04 00 00 00 3C 00 00 FF EE 00 0E 41 64 6F 62 65 ; ....<.. ?..Adobe
00000003h: 00 64 C0 00 00 00 01 FF DB 00 84 00 06 04 04 04 ; .d?.... ??....
00000004h: 05 04 06 05 05 06 09 06 05 06 09 0B 08 06 06 08 ; .....
00000005h: 0B 0C 0A 0A 0B 0A 0A 0C 10 0C 0C 0C 0C 0C 10 ; .....
00000006h: 0C 0E 0F 10 0F 0E 0C 13 13 14 14 13 13 1C 1B 1B ; .....
00000007h: 1B 1C 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 01 07 07 07 ; .....
00000008h: 0D 0C 0D 18 10 10 18 1A 15 11 15 1A 1F 1F 1F 1F ; .....
00000009h: 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F ; .....
0000000a0h: 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F ; .....
0000000b0h: 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F FF C0 00 ; ..... ?
0000000c0h: 11 08 03 00 04 00 03 01 11 00 02 11 01 03 11 01 ; .....
0000000d0h: FF C4 00 B2 00 00 01 05 01 01 01 00 00 00 00 00 ; ??.....
0000000e0h: 00 00 00 00 00 00 01 04 05 06 07 03 02 08 01 00 ; .....
0000000f0h: 02 03 01 01 01 00 00 00 00 00 00 00 00 00 00 ; .....
00000100h: 01 02 03 04 05 06 07 10 00 01 03 03 03 02 04 04 ; .....
00000110h: 03 07 03 02 04 05 00 0B 01 02 03 04 00 11 05 21 ; .....!
00000120h: 12 06 31 41 51 61 22 13 71 32 14 07 81 42 23 91 ; ..1AQa".q2...T#?
00000130h: A1 52 62 33 24 15 B1 C1 72 D1 43 82 53 34 16 E1 ; b3$.绑r表係4.2

```

图 3.1.3 jpeg 图片文件示例图

## 4.2 bmp 图片格式

bmp 是一种与硬件设备无关的图像文件格式，使用非常广。它采用位映射存储格式，除了图像深度可选以外，不采用其他任何压缩，因此，bmp 文件所占用的空间很大。bmp 文件的图像深度可选 1bit、4bit、8bit 及 24bit。bmp 文件存储数据时，图像的扫描方式是按从左到右、从下到上的顺序。

典型的 bmp 图像文件由三部分组成：位图文件头数据结构，它包含 bmp 图像文件的类型、显示内容等信息；位图信息数据结构，它包含有 bmp 图像的宽、高、压缩方法，以及定义颜色等信息，bmp 图片具体内容如图 3.3.4：

00000000h:	42	4D	6A	9E	0C	00	00	00	00	00	36	00	00	00	28	00	; BM ? ..
00000010h:	00	00	23	02	00	00	F7	01	00	00	01	00	18	00	00	00	; ..#...
00000020h:	00	00	34	9E	0C	00	00	00	00	00	00	00	00	00	00	00	; ..4?..
00000030h:	00	00	00	00	00	00	FF	FO	E5	FF	FO	E5	FF	FO	E5	FF	; .....
00000040h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000050h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000060h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000070h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000080h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000090h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
000000a0h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
000000b0h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
000000c0h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
000000d0h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
000000e0h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
000000f0h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000100h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000110h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000120h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....
00000130h:	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	; .....

图 3.1.4 bmp 图片文件示例图

以上图为例，文件第一个字节为 424Dh='BM'，表示是 bmp 格式，接下来的内容包括文件大小、文件开始到位图数据之间的偏移量、位图宽高等信息。

若想判断图片文件是否为 bmp 格式，可以根据位图文件头内容是否为 424Dh='BM'来确定。

### 4.3 png 图片格式

png(Portable Network Graphics)的原名称为"可移植性网络图像"，是网上接受的最新图像文件格式。png 能够提供长度比 GIF 小 30%的无损压缩图像文件。它同时提供 24 位和 48 位真彩色图像支持以及其他诸多技术性支持。由于 png 非常新，所以目前并不是所有的程序都可以用它来存储图像文件，但 Photoshop 可以处理 png 图像文件，也可以用 png 图像文件格式存储。

对于一个 PNG 文件来说，其文件头总是由固定的字节来描述的(十六进制):

89 50 4E 47 0D 0A 1A 0A

其中第一个字节 0x89 超出了 ASCII 字符的范围，这是为了避免某些软件将 png 文件当做文本文件来处理。以图 3.3.5 为例：

00000000h:	89	50	4E	47	0D	0A	1A	0A	00	00	00	0D	49	48	44	52	; PNG.... IHDR
00000010h:	00	00	00	08	00	00	00	08	04	03	00	00	00	36	21	A3	; .....6!?
00000020h:	B8	00	00	00	03	73	42	49	54	08	08	08	DB	E1	4F	E0	; ...sBIT... 垠o?
00000030h:	00	00	00	27	50	4C	54	45	FF	FF	00	FF	ED	00	FF	C1	; ...'PLTE . ? ?
00000040h:	00	FF	99	00	FF	66	00	FF	3B	00	FF	0F	00	E2	00	15	; . ? f. ;. ...?
00000050h:	B7	00	34	8B	00	54	60	00	73	33	00	99	09	00	B2	5F	; ?4?T`.s3.? 瞋
00000060h:	F5	BB	DD	00	00	00	09	70	48	59	73	00	00	0B	12	00	; 寔?...pHYs....
00000070h:	00	0B	12	01	D2	DD	7E	FC	00	00	00	20	74	45	58	74	; ....逸~?... tEXt
00000080h:	53	6F	66	74	77	61	72	65	00	4D	61	63	72	6F	6D	65	; Software.Macrome
00000090h:	64	69	61	20	46	69	72	65	77	6F	72	6B	73	20	4D	58	; dia Fireworks MX
000000a0h:	BB	91	2A	24	00	00	00	16	74	45	58	74	43	72	65	61	; 彝*\$....tEXtCrea
000000b0h:	74	69	6F	6E	20	54	69	6D	65	00	30	34	2F	30	31	2F	; tion Time.04/01/
000000c0h:	30	35	22	44	50	99	00	00	00	27	49	44	41	54	78	9C	; 05"DP?... IDATx?
000000d0h:	63	38	BD	B2	3D	95	61	D7	8C	B2	10	06	20	C3	99	01	; c8讲=弄讓?. 朕.
000000e0h:	C8	30	62	00	32	14	19	80	0C	01	06	10	83	01	C4	00	; ?b.2..e....??
000000f0h:	00	24	A7	0B	A4	DA	12	06	A5	00	00	00	00	49	45	4E	; .\$?~...IEN
00000100h:	44	AE	42	60	82												; D 珣`口

图 3.1.5 png 图片文件示例图



我们可以通过判断文件的前八个字节来确定该文件是否为 png 图片。

## 5. libjpeg 库简介

libjpeg 是一个被广泛使用的 jpeg 压缩/解压缩函数库，它能够读写 JFIF 格式的 jpeg 图像文件，通常这类文件是以.jpg 或者.jpeg 为后缀名的。通过 libjpeg 库，应用程序可以每次从 jpeg 压缩图像中读取一个或多个扫描线（scanline，所谓扫描线，是指由一行像素点构成的一条图像线条），而诸如颜色空间转换、降采样/增采样、颜色量化之类的工作则都由 libjpeg 去完成了。

要使用 libjpeg，需要读者对数字图像的基本知识有初步的了解。对于 libjpeg 而言，图像数据是一个二维的像素矩阵。对于彩色图像，每个像素通常用三个分量表示，即 R(Red)、G(Green)、B(Blue)三个分量，每个分量用一个字节表示，因此每个分量的取值范围从 0 到 255；对于灰度图像，每个像素通常用一个分量表示，一个分量同样由一个字节表示，取值范围从 0 到 255。

在 libjpeg 中，图像数据是以扫描线的形式存放的。每一条扫描线由一行像素点构成，像素点沿着扫描线从左到右依次排列。对于彩色图像，每个分量由三个字节组成，因此这三个字节以 R、G、B 的顺序构成扫描线上的一个像素点。一个典型的扫描线形式如下：

R,G,B,R,G,B,R,G,B,...

通过 libjpeg 解压出来的图像数据也是以扫描线的形式存放的。

### ✓ libjpeg 解压方法

#### 1、分配并初始化一个 jpeg 解压对象

```
struct jpeg_decompress_struct cinfo;
struct jpeg_error_mgr jerr;
...
cinfo.err = jpeg_std_error(&jerr);
jpeg_create_decompress(&cinfo);
```

#### 2、指定要解压缩的图像文件

```
FILE * infile;
..
if ((infile = fopen(filename, "rb")) == NULL)
{
    fprintf(stderr, "can't open %s\n", filename);
    exit(1);
}
jpeg_stdio_src(&cinfo, infile);
```

#### 3、调用 jpeg\_read\_header()获取图像信息



```
jpeg_read_header(&cinfo, TRUE);
```

4、设置 jpeg 解压缩对象 cinfo 的一些参数

这是一个可选步骤，本文忽略

5、调用 jpeg\_start\_decompress() 开始解压过程

```
jpeg_start_decompress(&cinfo);
```

调用 jpeg\_start\_decompress() 函数之后，jpeg 解压缩对象 cinfo 中的下面这几个字段将会比较有用：

output\_width                      这是图像输出的宽度

output\_height                      这是图像输出的高度

output\_components                  每个像素的分量数，也即字节数

这是因为在调用 jpeg\_start\_decompress() 之后往往需要为解压后的扫描线上的所有像素点分配存储空间，这个空间的大小可以通过 output\_width \* output\_components 确定，而要读取的扫描线的总数为 output\_height 行。

6、读取一行或者多行扫描线数据并处理

通常的代码是这样的：

```
while (cinfo.output_scanline < cinfo.output_height)
{
    jpeg_read_scanlines();           /* deal with scanlines */
}
```

对扫描线的读取是按照从上到下的顺序进行的，也就是说图像最上方的扫描线最先被 jpeg\_read\_scanlines() 读入存储空间中，紧接着是第二个扫描线，最后是图像底边的扫描线被读入存储空间中。

7、调用 jpeg\_finish\_decompress() 完成解压过程

```
jpeg_finish_decompress(&cinfo);
```

8、调用 jpeg\_destroy\_decompress() 释放 jpeg 解压对象 cinfo

```
jpeg_destroy_decompress(&cinfo);
```

以上就是通过 libjpeg 函数解压 jpeg 压缩图像的基本过程，另外需要说明的是：由于本项目中所用的 FrameBuffer 设备的颜色深度为 16 位，颜色格式为 5-6-5 格式——即 R（红色）在 16bit 中占据高 5 位，G（绿色）在 16bit 中占据中间 6 位，B（蓝色）在 16bit 中占据低 5 位；而 libjpeg 解压出来的图像数据为 24 位 RGB 格式，因此必须进行转换。对于 24 位的 RGB，每个字节表示一个颜色分量，因此转换的方式为：对于 R 字节，右移 3 位，对于 G 字节，右移 2 位，对于 B 字节，右移 3 位，然后将右移得到的值拼接起来，就得到了 16 位的颜色值。

## 第 6 章 字体显示

---

课程内容：

- ✧ 漫谈字符集和编码
- ✧ **utf-8** 编码详解

## 第 6 章 字体显示

### 1、漫谈字符集和编码

很久很久以前，有一群人，他们决定用 8 个可以开合的晶体管来组合成不同的状态，以表示世界上的万物。他们看到 8 个开关状态是好的，于是他们把这称为"字节"。

再后来，他们又做了一些可以处理这些字节的机器，机器开动了，可以用字节来组合出很多状态，状态开始变来变去。他们看到这样是好的，于是它们就这机器称为"计算机"。

开始计算机只在美国用。八位的字节一共可以组合出  $256(2 \text{ 的 } 8 \text{ 次方})$  种不同的状态。

他们把其中的编号从 0 开始的 32 种状态分别规定了特殊的用途，一但终端、打印机遇上约定好的这些字节被传过来时，就要做一些约定的动作。遇上 `00x10`，终端就换行，遇上 `0x07`，终端就向人们嘟嘟叫，例好遇上 `0x1b`，打印机就打印反白的字，或者终端就用彩色显示字母。他们看到这样很好，于是就把这些 `0x20` 以下的字节状态称为"控制码"。

他们又把所有的空格、标点符号、数字、大小写字母分别用连续的字节状态表示，一直编到了第 127 号，这样计算机就可以用不同字节来存储英语的文字了。大家看到这样，都感觉很好，于是大家都把这个方案叫做 ANSI 的"Ascii"编码（American Standard Code for Information Interchange，美国信息互换标准代码）。当时世界上所有的计算机都用同样的 ASCII 方案来保存英文文字。

后来，就像建造巴比伦塔一样，世界各地的都开始使用计算机，但是很多国家用的不是英文，他们的字母里有许多是 ASCII 里没有的，为了可以在计算机保存他们的文字，他们决定采用 127 号之后的空位来表示这些新的字母、符号，还加入了很多画表格时需要用下到的横线、竖线、交叉等形状，一直把序号编到了最后一个状态 255。从 128 到 255 这一页的字符集被称"扩展字符集"。从此之后，贪婪的人类再没有新的状态可以用了，美帝国主义可能没有想到还有第三世界国家的人们也希望能用到计算机吧！

等中国人们得到计算机时，已经没有可以利用的字节状态来表示汉字，况且有 6000 多个常用汉字需要保存呢。但是这难不倒智慧的中国人民，我们不客气地把那些 127 号之后的奇异符号们直接取消掉，规定：一个小于 127 的字符的意义与原来相同，但两个大于 127 的字符连在一起时，就表示一个汉字，前面的一

个字节（他称之为高字节）从 0xA1 用到 0xF7，后面一个字节（低字节）从 0xA1 到 0xFE，这样我们就可以组合出大约 7000 多个简体汉字了。在这些编码里，我们还把数学符号、罗马希腊的字母、日文的假名们都编进去了，连在 ASCII 里本来就有的数字、标点、字母都统统重新编了两个字节长的编码，这就是常说的“全角”字符，而原来在 127 号以下的那些就叫“半角”字符了。

中国人民看到这样很不错，于是就把这种汉字方案叫做“GB2312”。GB2312 是对 ASCII 的中文扩展。

但是中国的汉字太多了，我们很快就发现有许多人的姓名没有办法在这里打出来，特别是某些很会麻烦别人的国家领导人。于是我们不得不继续把 GB2312 没有用到的码位找出来老实不客气地用上。

后来还是不够用，于是干脆不再要求低字节一定是 127 号之后的内码，只要第一个字节是大于 127 就固定表示这是一个汉字的开始，不管后面跟的是不是扩展字符集里的内容。结果扩展之后的编码方案被称为 GBK 标准，GBK 包括了 GB2312 的所有内容，同时又增加了近 20000 个新的汉字（包括繁体字）和符号。

后来少数民族也要用电脑了，于是我们再扩展，又加了几千个新的少数民族的字，GBK 扩成了 GB18030。从此之后，中华民族的文化就可以在计算机时代中传承了。

中国的程序员们看到这一系列汉字编码的标准是好的，于是通称他们叫做“DBCS”（Double Byte Character Set 双字节字符集）。在 DBCS 系列标准里，最大的特点是两字节长的汉字字符和一字节长的英文字符并存于同一套编码方案里，因此他们写的程序为了支持中文处理，必须要注意字符串里的每一个字节的值，如果这个值是大于 127 的，那么就认为一个双字节字符集里的字符出现了。那时候凡是受过加持，会编程的计算机僧侣们都要每天念下面这个咒语数百遍：

“一个汉字算两个英文字符！一个汉字算两个英文字符……”

因为当时各个国家都像中国这样搞出一套自己的编码标准，结果互相之间谁

也不懂谁的编码,谁也不支持别人的编码,连大陆和台湾这样只相隔了 150 海里,使用着同一种语言的兄弟地区,也分别采用了不同的 DBCS 编码方案??当时的中国人想让电脑显示汉字,就必须装上一个"汉字系统",专门用来处理汉字的显示、输入的问题,但是那个台湾的愚昧封建人士写的算命程序就必须加装另一套支持 BIG5 编码的什么"倚天汉字系统"才可以用,装错了字符系统,显示就会乱了套!这怎么办?而且世界民族之林中还有那些一时用不上电脑的穷苦人民,他们的文字又怎么办?

真是计算机的巴比伦塔命题啊!

正在这时,大天使加百列及时出现了??一个叫 ISO (国际标准化组织)的国际组织决定着手解决这个问题。他们采用的方法很简单:废了所有的地区性编码方案,重新搞一个包括了地球上所有文化、所有字母和符号的编码!他们打算叫它 "Universal Multiple- Octet Coded Character Set",简称 UCS,俗称 "UNICODE".

UNICODE 开始制订时,计算机的存储器容量极大地发展了,空间再也不成为问题了。于是 ISO 就直接规定必须用两个字节,也就是 16 位来统一表示所有的字符,对于 ascii 里的那些“半角”字符,UNICODE 保持其原编码不变,只是将其长度由原来的 8 位扩展为 16 位,而其他文化和语言的字符则全部重新统一编码。由于“半角”英文符号只需要用到低 8 位,所以其高 8 位永远是 0,因此这种大气的方案在保存英文文本时会多浪费一倍的空间。

这时候,从旧社会里走过来的程序员开始发现一个奇怪的现象:他们的 strlen 函数靠不住了,一个汉字不再是相当于两个字符了,而是一个!是的,从 UNICODE 开始,无论是半角的英文字母,还是全角的汉字,它们都是统一的“一个字符”!同时,也都是统一的“两个字节”,请注意“字符”和“字节”两个术语的不同,“字节”是一个 8 位的物理存储单元,而“字符”则是一个文化相关的符号。在 UNICODE 中,一个字符就是两个字节。一个汉字算两个英文字符的时代已经快过去了。

从前多种字符集存在时,那些做多语言软件的公司遇上过很大麻烦,他们为了在不同的国家销售同一套软件,就不得不在区域化软件时也加持那个双字节字符集咒语,不仅要处处小心不要搞错,还要把软件中的文字在不同的字符集中转来转去。UNICODE 对于他们来说是一个很好的一揽子解决方案,于是从

Windows NT 开始，MS 趁机把它们的操作系统改了一遍，把所有的核心代码都改成了用 UNICODE 方式工作的版本，从这时开始，WINDOWS 系统终于不需要加装各种本土语言系统，就可以显示全世界上所有文化的字符了。

但是，UNICODE 在制订时没有考虑与任何一种现有的编码方案保持兼容，这使得 GBK 与 UNICODE 在汉字的内码编排上完全是不一样的，没有一种简单的算术方法可以把文本内容从 UNICODE 编码和另一种编码进行转换，这种转换必须通过查表来进行。

如前所述，UNICODE 是用两个字节来表示为一个字符，他总共可以组合出 65535 不同的字符，这大概已经可以覆盖世界上所有文化的符号。如果还不够也没有关系，ISO 已经准备了 UCS-4 方案，说简单了就是四个字节来表示一个字符，这样我们就可以组合出 21 亿个不同的字符出来（最高位有其他用途），这大概可以用到银河联邦成立那一天吧！

UNICODE 来到时，一起到来的还有计算机网络的兴起，UNICODE 如何在网络上传输也是一个必须考虑的问题，于是面向传输的众多 UTF（UCS Transfer Format）标准出现了，顾名思义，UTF8 就是每次 8 个位传输数据，而 UTF16 就是每次 16 个位，只不过为了传输时的可靠性，从 UNICODE 到 UTF 时并不是直接的对应，而是要过一些算法和规则来转换。

受到过网络编程加持的计算机僧侣们都知道，在网络里传递信息时有一个很重要的问题，就是对于数据高低位的解读方式，一些计算机是采用低位先发送的方法，例如我们 PC 机采用的 INTEL 架构，而另一些是采用高位先发送的方式，在网络中交换数据时，为了核对双方对于高低位的认识是否是一致的，采用了一种很简便的方法，就是在文本流的开始时向对方发送一个标志符??如果之后的文本是高位在位，那就发送"FEFF"，反之，则发送"FFFE"。不信你可以用二进制方式打开一个 UTF-X 格式的文件，看看开头两个字节是不是这两个字节？

讲到这里，我们再顺便说说一个很著名的奇怪现象：当你在 windows 的记事本里新建一个文件，输入"联通"两个字之后，保存，关闭，然后再次打开，你会发现这两个字已经消失了，代之的是几个乱码！呵呵，有人说这就是联通之所以拼不过移动的原因。



其实这是因为 GB2312 编码与 UTF8 编码产生了编码冲撞的原因。

从网上引来一段从 UNICODE 到 UTF8 的转换规则：

Unicode

UTF-8

0000 - 007F

0xxxxxxx

0080 - 07FF

110xxxxx 10xxxxxx

0800 - FFFF

1110xxxx 10xxxxxx 10xxxxxx

例如"汉"字的 Unicode 编码是 6C49。6C49 在 0800-FFFF 之间，所以要用 3 字节模板：1110xxxx 10xxxxxx 10xxxxxx。将 6C49 写成二进制是：0110 1100 0100 1001，将这个比特流按三字节模板的分段方法分为 0110 110001 001001，依次代替模板中的 x，得到：1110-0110 10-110001 10-001001，即 E6 B1 89，这就是其 UTF8 的编码。

而当你新建一个文本文件时，记事本的编码默认是 ANSI，如果你在 ANSI 的编码输入汉字，那么他实际就是 GB 系列的编码方式，在这种编码下，"联通"的内码是：

c1 1100 0001

aa 1010 1010

cd 1100 1101

a8 1010 1000

注意到了吗？第一二字节、第三四个字节的起始部分的都是"110"和"10"，

正好与 UTF8 规则里的两字节模板是一致的，于是再次打开记事本时，记事本就误认为这是一个 UTF8 编码的文件，让我们把第一个字节的 110 和第二个字节的 10 去掉，我们就得到了"00001 101010"，再把各位对齐，补上前导的 0，就得到了"0000 0000 0110 1010"，不好意思，这是 UNICODE 的 006A，也就是小写的字母"j"，而之后的两字节用 UTF8 解码之后是 0368，这个字符什么也不是。这就是只有"联通"两个字的文件没有办法在记事本里正常显示的原因。

而如果你在"联通"之后多输入几个字，其他的字的编码不见得又恰好是 110 和 10 开始的字节，这样再次打开时，记事本就不会坚持这是一个 utf8 编码的文件，而会用 ANSI 的方式解读之，这时乱码又不出现了。

好了，终于可以回答 NICO 的问题了，在数据库里，有 n 前缀的字串类型就是 UNICODE 类型，这种类型中，固定用两个字节来表示一个字符，无论这个字符是汉字还是英文字母，或是别的什么。

如果你要测试"abc 汉字"这个串的长度，在没有 n 前缀的数据类型里，这个字串是 7 个字符的长度，因为一个汉字相当于两个字符。而在有 n 前缀的数据类型里，同样的测试串长度的函数将会告诉你 5 个字符，因为一个汉字就是一个字符。

## 2、utf-8 编码详解

在将来不远的几年里，Unicode 已经很接近于取代 ASCII 与 Latin-1 编码的位置了。它不仅允许你处理处理事实上存在于地球上的任何语言文字，而且提供了一个全面的数学与技术符号集，因此可以简化科学信息交换。

UTF-8 编码提供了一种简便而向后兼容的方法，使得那种完全围绕 ASCII 设计的操作系统，比如 Unix，也可以使用 Unicode。UTF-8 就是 Unix, Linux 已经类似的系统使用 Unicode 的方式。现在是你了解它的时候了。

### 2.1 什么是 UCS 和 ISO 10646?

国际标准 ISO 10646 定义了 通用字符集 (Universal Character Set, UCS)。UCS 是所有其他字符集标准的一个超集。它保证与其他字符集是双向兼容的。就是说，如果你将任何文本字符串翻译到 UCS 格式，然后再翻译回原编码，你不会丢失任何信息。

UCS 包含了用于表达所有已知语言的字符。不仅包括拉丁语，希腊语，斯拉



夫语,希伯来语,阿拉伯语,亚美尼亚语和乔治亚语的描述,还包括中文,日文和韩文这样的象形文字,以及平假名,片假名,孟加拉语,旁遮普语果鲁穆奇字符 (Gurmukhi), 泰米尔语,印.埃纳德语 (Kannada), Malayalam, 泰国语, 老挝语, 汉语拼音 (Bopomofo), Hangul, Devangari, Gujarati, Oriya, Telugu 以及其他数也数不清的语. 对于还没有加入的语言, 由于正在研究怎样在计算机中最好地编码它们, 因而最终它们都将被加入. 这些语言包括 Tibetan, 高棉语, Runic(古代北欧文字), 埃塞俄比亚语, 其他象形文字, 以及各种各样的印-欧语系的语言, 还包括挑选出来的艺术语言比如 Tengwar, Cirth 和 克林贡语(Klingon). UCS 还包括大量的图形的, 印刷用的, 数学用的和科学用的符号, 包括所有由 TeX, Postscript, MS-DOS, MS-Windows, Macintosh, OCR 字体, 以及许多其他字处理和出版系统提供的字符.

ISO 10646 定义了一个 31 位的字符集. 然而, 在这巨大的编码空间中, 迄今为止只分配了前 65534 个码位 (0x0000 到 0xFFFFD). 这个 UCS 的 16 位子集称为 基本多语言面 (Basic Multilingual Plane, BMP). 将被编码在 16 位 BMP 以外的字符都属于非常特殊的字符(比如象形文字), 且只有专家在历史和科学领域里才会用到它们. 按当前的计划, 将来也许再也不会再有字符被分配到从 0x000000 到 0x10FFFF 这个覆盖了超过 100 万个潜在的未来字符的 21 位的编码空间以外去了. ISO 10646-1 标准第一次发表于 1993 年, 定义了字符集与 BMP 中内容的架构. 定义 BMP 以外的字符编码的第二部分 ISO 10646-2 正在准备中, 但也许要过好几年才能完成. 新的字符仍源源不断地加入到 BMP 中, 但已经存在的字符是稳定的且不会再改变了.

UCS 不仅给每个字符分配一个代码, 而且赋予了一个正式的名字. 表示一个 UCS 或 Unicode 值的十六进制数, 通常在前面加上 "U+", 就象 U+0041 代表字符"拉丁大写字母 A". UCS 字符 U+0000 到 U+007F 与 US-ASCII(ISO 646) 是一致的, U+0000 到 U+00FF 与 ISO 8859-1(Latin-1) 也是一致的. 从 U+E000 到 U+F8FF, 已经 BMP 以外的大范围的编码是为私用保留的.

## 2.2 什么是组合字符?

UCS 里有些编码点分配给了 组合字符. 它们类似于打字机上的无间隔重音键. 单个的组合字符不是一个完整的字符. 它是一个类似于重音符或其他指示标记, 加在前一个字符后面. 因而, 重音符可以加在任何字符后面. 那些最重要的

被加重的字符,就象普通语言的正字法(orthographies of common languages)里用到的那种,在 UCS 里都有自己的位置,以确保同老的字符集的向后兼容性. 既有自己的编码位置,又可以表示为一个普通字符跟随一个组合字符的被加重字符,被称为 预作字符(precomposed characters). UCS 里的预作字符是为了同没有预作字符的旧编码,比如 ISO 8859,保持向后兼容性而设的. 组合字符机制允许在任何字符后加上重音符或其他指示标记,这在科学符号中特别有用,比如数学方程式和国际音标字母,可能会需要在一个基本字符后组合上一个或多个指示标记.

组合字符跟随着被修饰的字符. 比如,德语中的元音变音字符 ("拉丁大写字母 A 加上分音符"),既可以表示为 UCS 码 U+00C4 的预作字符,也可以表示成一个普通 "拉丁大写字母 A" 跟着一个"组合分音符":U+0041 U+0308 这样的组合. 当需要堆叠多个重音符,或在一个基本字符的上面和下面都要加上组合标记时,可以使用多个组合字符. 比如在泰国文中,一个基本字符最多可加上两个组合字符.

## 2.3 什么是 UCS 实现级别?

不是所有的系统都需要支持象组合字符这样的 UCS 里所有的先进机制. 因此 ISO 10646 指定了下列三种实现级别:

### ✓ 级别 1

不支持组合字符和 Hangul Jamo 字符 (一种特别的,更加复杂的韩国文的编码,使用两个或三个子字符来编码一个韩文音节)

### ✓ 级别 2

类似于级别 1,但在某些文字中,允许一系列固定的组合字符 (例如,希伯来文,阿拉伯文,Devangari,孟加拉语,果鲁穆奇语, Gujarati, Oriya, 泰米尔语, Telugo, 印.埃纳德语, Malayalam, 泰国语和老挝语). 如果没有这最起码的几个组合字符, UCS 就不能完整地表达这些语言.

### ✓ 级别 3

支持所有的 UCS 字符,例如数学家可以在任意一个字符上加上一个 tilde(颞化符号,西班牙语字母上面的~)或一个箭头(或两者都加).

## 2.4 什么是 Unicode?

历史上, 有两个独立的, 创立单一字符集的尝试. 一个是国际标准化组织 (ISO) 的 ISO 10646 项目, 另一个是由(一开始大多是美国的)多语言软件制造商组成的协会组织的 Unicode 项目. 幸运的是, 1991 年前后, 两个项目的参与者都认识到, 世界不需要两个不同的单一字符集. 它们合并双方的工作成果, 并为创立一个单一编码表而协同工作. 两个项目仍都存在并独立地公布各自的标准, 但 Unicode 协会和 ISO/IEC JTC1/SC2 都同意保持 Unicode 和 ISO 10646 标准的码表兼容, 并紧密地共同调整任何未来的扩展.

那么 Unicode 和 ISO 10646 不同在什么地方?

Unicode 协会公布的 Unicode 标准 严密地包含了 ISO 10646-1 实现级别 3 的基本多语言面. 在两个标准里所有的字符都在相同的位置并且有相同的名字.

Unicode 标准额外定义了许多与字符有关的语义符号学, 一般而言是对于实现高质量的印刷出版系统的更好的参考. Unicode 详细说明了绘制某些语言(比如阿拉伯语)表达形式的算法, 处理双向文字(比如拉丁与希伯来文混合文字)的算法和 排序与字符串比较 所需的算法, 以及其他许多东西.

另一方面, ISO 10646 标准, 就象广为人知的 ISO 8859 标准一样, 只不过是一个简单的字符集表. 它指定了一些与标准有关的术语, 定义了一些编码的别名, 并包括了规范说明, 指定了怎样使用 UCS 连接其他 ISO 标准的实现, 比如 ISO 6429 和 ISO 2022. 还有一些与 ISO 紧密相关的, 比如 ISO 14651 是关于 UCS 字符串排序的.

考虑到 Unicode 标准有一个易记的名字, 且在任何好的书店里的 Addison-Wesley 里有, 只花费 ISO 版本的一小部分, 且包括更多的辅助信息, 因而它成为使用广泛得多的参考也就不足为奇了. 然而, 一般认为, 用于打印 ISO 10646-1 标准的字体在某些方面的质量要高于用于打印 Unicode 2.0 的. 专业字体设计者总是被建议说要两个标准都实现, 但一些提供的样例字形有显著的区别. ISO 10646-1 标准同样使用四种不同的风格变体来显示表意文字如中文, 日文和韩文 (CJK), 而 Unicode 2.0 的表里只有中文的变体. 这导致了普遍的认为 Unicode 对日本用户来说是不可接收的传说, 尽管是错误的.

## 2.5 什么是 UTF-8?

首先 UCS 和 Unicode 只是分配整数给字符的编码表. 现在存在好几种将一串字符表示为一串字节的方法. 最显而易见的两种方法是将 Unicode 文本存储为 2 个或 4 个字节序列的串. 这两种方法的正式名称分别为 UCS-2 和 UCS-4. 除非另外指定, 否则大多数的字节都是这样的(Bigendian convention). 将一个 ASCII 或 Latin-1 的文件转换成 UCS-2 只需简单地在每个 ASCII 字节前插入 0x00. 如果要转换成 UCS-4, 则必须在每个 ASCII 字节前插入三个 0x00.

在 Unix 下使用 UCS-2 (或 UCS-4) 会导致非常严重的问题. 用这些编码的字符串会包含一些特殊的字符, 比如 '\0' 或 '\/', 它们在文件名和其他 C 库函数参数里都有特别的含义. 另外, 大多数使用 ASCII 文件的 UNIX 下的工具, 如果不进行重大修改是无法读取 16 位的字符的. 基于这些原因, 在文件名, 文本文件, 环境变量等地方, UCS-2 不适合作为 Unicode 的外部编码.

在 ISO 10646-1 Annex R 和 RFC 2279 里定义的 UTF-8 编码没有这些问题. 它是在 Unix 风格的操作系统下使用 Unicode 的明显的方法.

UTF-8 有以下特性:

UCS 字符 U+0000 到 U+007F (ASCII) 被编码为字节 0x00 到 0x7F (ASCII 兼容). 这意味着只包含 7 位 ASCII 字符的文件在 ASCII 和 UTF-8 两种编码方式下是一样的.

所有 >U+007F 的 UCS 字符被编码为一个或多个字节的串, 每个字节都有标记位集. 因此, ASCII 字节 (0x00-0x7F) 不可能作为任何其他字符的一部分.

表示非 ASCII 字符的多字节串的第一个字节总是在 0xC0 到 0xFD 的范围内, 并指出这个字符包含多少个字节. 多字节串的其余字节都在 0x80 到 0xBF 范围内. 这使得重新同步非常容易, 并使编码无国界, 且很少受丢失字节的影响.

可以编入所有可能的 231 个 UCS 代码

UTF-8 编码字符理论上可以最多到 6 个字节长, 然而 16 位 BMP 字符最多只用到 3 字节长.

Bigendian UCS-4 字节串的排列顺序是预定的.

字节 0xFE 和 0xFF 在 UTF-8 编码中从未用到.

下列字节串用来表示一个字符. 用到哪个串取决于该字符在 Unicode 中的

序号.

U-00000000 - U-0000007F: 0xxxxxxx  
U-00000080 - U-000007FF: 110xxxxx 10xxxxxx  
U-00000800 - U-0000FFFF: 1110xxxx 10xxxxxx 10xxxxxx  
U-00010000 - U-001FFFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx  
U-00200000 - U-03FFFFFF: 111110xx 10xxxxxx 10xxxxxx 10xxxxxx  
10xxxxxx  
U-04000000 - U-7FFFFFFF: 1111110x 10xxxxxx 10xxxxxx 10xxxxxx  
10xxxxxx 10xxxxxx

xxx 的位置由字符编码数的二进制表示的位填入. 越靠右的 x 具有越少的特殊意义. 只用最短的那个足够表达一个字符编码数的多字节串. 注意在多字节串中, 第一个字节的开头"1"的数目就是整个串中字节的数目.

例如: Unicode 字符 U+00A9 = 1010 1001 (版权符号) 在 UTF-8 里的编码为:

11000010 10101001 = 0xC2 0xA9

而字符 U+2260 = 0010 0010 0110 0000 (不等于) 编码为:

11100010 10001001 10100000 = 0xE2 0x89 0xA0

这种编码的官方名字拼写为 UTF-8, 其中 UTF 代表 UCS Transformation Format. 请勿在任何文档中用其他名字 (比如 utf8 或 UTF\_8) 来表示 UTF-8, 当然除非你指的是一个变量名而不是这种编码本身.

## 2.6 什么编程语言支持 Unicode?








在大约 1993 年之后开发的大多数现代编程语言都有一个特别的数据类型, 叫做 Unicode/ISO 10646-1 字符. 在 Ada95 中叫 Wide\_Character, 在 Java 中叫 char.

ISO C 也详细说明了处理多字节编码和宽字符 (wide characters) 的机制, 1994 年 9 月 Amendment 1 to ISO C 发表时又加入了更多. 这些机制主要是为各类东亚编码而设计的, 它们比处理 UCS 所需的要健壮得多. UTF-8 是 ISO C 标准调用多字节字符串的编码的一个例子, `wchar_t` 类型可以用来存放 Unicode 字符.

## 2.7 在 Linux 下该如何使用 Unicode?

在 UTF-8 之前, 不同地区的 Linux 用户使用各种各样的 ASCII 扩展. 最普遍的欧洲编码是 ISO 8859-1 和 ISO 8859-2, 希腊编码 ISO 8859-7, 俄国编码 KOI-8, 日本编码 EUC 和 Shift-JIS, 等等. 这使得 文件的交换非常困难, 且应用软件必须特别关心这些编码的不同之处.

最终, Unicode 将取代所有这些编码, 主要通过 UTF-8 的形式. UTF-8 将应用在

-  文本文件 (源代码, HTML 文件, email 消息, 等等)
-  文件名
-  标准输入与标准输出, 管道
-  环境变量
-  剪切与粘贴选择缓冲区
-  telnet, modem 和到终端模拟器的串口连接
-  以及其他地方以前用 ASCII 来表示的字节串

在 UTF-8 模式下, 终端模拟器, 比如 `xterm` 或 `Linux console driver`, 将每次按键转换成相应的 UTF-8 串, 然后发送到前台进程的 `stdin` 里. 类似的, 任何进程在 `stdout` 上的输出都将发送到终端模拟器, 在那里用一个 UTF-8 解码器进行处理, 之后再以一种 16 位的字体显示出来.

只有在功能完善的多语言字处理器包里才可能有完全的 Unicode 功能支持. 而广泛用在 Linux 里用于取代 ASCII 和其他 8 位字符集的方案则要简单得多. 第一步, Linux 终端模拟器和命令行工具将只是转变到 UTF-8. 这意味着只用到级别 1 的 ISO 10646-1 实现 (没有组合字符), 且只支持那些不需要更多处理的语言象 拉丁, 希腊, 斯拉夫 和许多科学用符号. 在这个级别上, UCS 支持与 ISO 8859 支持类似, 唯一显著的区别是现在我们有几千种字符可以用了, 其中



的字符可以用多字节串来表示.

总有一天 Linux 会当然地支持组合字符, 但即便如此, 对于组合字符串, 预作字符 (如何可用的话) 仍将是首选的. 更正式地, 在 Linux 下用 Unicode 对文本编码的首选的方法应该是定义在 Unicode Technical Report #15 里的 Normalization Form C.

在今后的一个阶段, 人们可以考虑增加在日文和中文里用到的双字节字符的支持 (他们相对比较简单), 组合字符支持, 甚至也许对从右至左书写的语言如希伯来文 (他们可不是那么简单的) 的支持. 但对这些高级功能的支持不应该阻碍简单的平板 UTF-8 在 拉丁, 希腊, 斯拉夫和科学用符号方面的快速应用, 以取代大量的欧洲 8 位编码, 并提供一个象样的科学用符号集.

## 2.7 我该如何修改我的软件?

有两种途径可以支持 UTF-8, 我称之为软转换与硬转换. 软转换时, 各处的数据均保存为 UTF-8 形式, 因而需要修改的软件很少. 在硬转换时, 程序将读入的 UTF-8 数据转换成宽字符数组, 以在应用程序内部处理. 在输出时, 再把字符串转换回 UTF-8 形式.

大多数应用程序只用软转换就可以工作得很好了. 这使得将 UTF-8 引入 Unix 成为切实可行的. 例如, 象 cat 和 echo 这样的程序根本不需要修改. 他们仍然可以对输入输出的是 ISO 8859-2 还是 UTF-8 一无所知, 因为它们只是搬运字节流而没有处理它们. 它们只能识别 ASCII 字符和象 '\n' 这样的控制码, 而这在 UTF-8 下也没有任何改变. 因此, 这些应用程序的 UTF-8 编码与解码将完全在终端模拟器里完成.

而那些通过数字字节数来获知字符数量的程序则需要一些小修改. 在 UTF-8 模式下, 它们必须不数入 0x80 到 0xBF 范围内的字节, 因为这些只是跟随字节, 它们本身并不是字符. 例如, ls 程序就必须修改, 因为它通过数文件名中字符数来排放给用户的目录表格布局. 类似地, 所有的假定其输出为定宽字体, 并因此而格式化它们的程序, 必须学会怎样数 UTF-8 文本中的字符数. 编辑器的功能, 如删除单个字符, 必须要作轻微的修改, 以删除可能属于该字符的所有字节. 受影响有编辑器 (vi, emacs, 等等) 以及使用 ncurses 库的程序.

Linux 核心使用软转换也可以工作得很好, 只需要非常微小的修改以支持 UTF-8. 大多数处理字符串的核心功能 (例如: 文件名, 环境变量, 等等) 都不受影响. 下列地方也许必须修改:

控制台显示与键盘驱动程序 (另一个 VT100 模拟器) 必须能编码和解码 UTF-8, 必须要起码支持 Unicode 字符集的几个子集. 从 Linux 1.2 起这些功能已经有了.

外部文件系统驱动程序, 例如 VFAT 和 WinNT 必须转换文件名字符编码. UTF-8 已经加入可用的转换选项的列表里了, 因此 mount 命令必须告诉核心驱动程序用户进程希望看到 UTF-8 文件名. 既然 VFAT 和 WinNT 无论如何至少已经用了 Unicode 了, 那么 UTF-8 在这里就可以发挥其优势, 以保证转换中无信息损失.

POSIX 系统的 tty 驱动程序支持一种 "cooked" 模式, 有一些原始的行编辑功能. 为了让字符删除功能工作正常, stty 必须在 tty 驱动程序里设置 UTF-8 模式, 因此它就不会把 0x80 到 0xBF 范围内的跟随字符也数进去了. Bruno Haible 那里已经有了一些 stty 和核心 tty 驱动程序的 Linux 补丁了.

#### C 对 Unicode 和 UTF-8 的支持

从 GNU glibc 2.1 开始, wchar\_t 类型已经正式定为只存放独立于当前 locale 的, 32 位的 ISO 10646 值. glibc 2.2 开始将完全支持 ISO C 中的多字节转换函数 (wprintf(), mbstowcs(), 等等), 这些函数可以用于在 wchar\_t 和包括 UTF-8 在内的任何依赖于 locale 的多字节编码间进行转换.

例如, 你可以写

```
wprintf(L"Sch 鰈 e Gr e!\n");
```

然后, 你的软件将按照你的用户在环境变量 LC\_CTYPE (例如, en\_US.UTF-8 或 de\_DE.ISO\_8859-1) 中选择的 locale 所指定的编码来打印这段文字. 你的编译器必须运行在与该 C 源文件所用编码相应的 locale 中, 在目标文件中以上的宽字符串将改为 wchar\_t 字符串存储. 在输出时, 运行时库将把 wchar\_t 字符串转换回与程序执行时的 locale 相应的编码.

注意, 类似这样的操作:

```
char c = L"a";
```



只允许从 U+0000 到 U+007F (7 位 ASCII) 范围里的字符. 对于非 ASCII 字符, 不能直接从 `wchar_t` 到 `char` 转换.

现在, 象 `readline()` 这样的函数在 UTF-8 locale 下也能工作了.

怎样激活 UTF-8 模式?

如果你的应用程序既支持 8 位字符集 (ISO 8859-\*, KOI-8, 等等), 也支持 UTF-8, 那么它必须通过某种方法以得知是否应使用 UTF-8 模式. 幸运的是, 在未来的几年里, 人们将只使用 UTF-8, 因此你可以将它作为默认, 但即使如此, 你还是得既支持传统 8 位字符集, 也支持 UTF-8.

当前的应用程序使用许许多多的不同的命令行开关来激活它们各自的 UTF-8 模式, 例如:

`xterm` 命令行选项 `"-u8"` 和 X resource `"XTerm*utf8:1"`

`gnat/gcc` 命令行选项 `"-gnatW8"`

`stty` 命令行选项 `"iutf8"`

`minied` 命令行选项 `"-U"`

`xemacs elisp` 包裹 以在 UTF-8 和内部使用的 MULE 编码间转换

`vim` `'fileencoding'` 选项

`less` 环境变量 `LESSCHARSET=utf-8`

记住每一个应用程序的命令行选项或其他配置方法是非常单调乏味的, 因此急需某种标准方法.

如果你在你的应用程序里使用硬转换, 并使用某种特定的 C 库函数来处理外部字符编码和内部使用的 `wchar_t` 编码的转换工作, 那么 C 库会帮你处理模式切换的问题. 你只需将环境变量 `LC_CTYPE` 设为正确的 locale, 例如, 如果你使用 UTF-8, 那就是 `en.UTF-8`, 而如果是 Latin-1, 并需要英语的转换, 则设为 `en.ISO_8859-1`.

然而, 大多数现存软件的维护者选择用软转换来代替, 而不使用 `libc` 的宽字符函数, 不仅因为它们还未得到广泛应用, 还因为这会使得软件进行大规模修改. 在这种情况下, 你的应用程序必须自己来获知何时使用 UTF-8 模式. 一种方式是做以下工作:

按照环境变量 `LC_ALL`, `LC_CTYPE`, `LANG` 的顺序, 寻找第一个有值的变量. 如果该值包含 UTF-8 子串 (也许是小写或没有"-") 则默认为 UTF-8 模式 (仍然可以用命令行开关来重设), 因为这个值可靠又恰当地指示了 C 库应该使用一种 UTF-8 locale.

提供一个命令行选项 (或者如果是 X 客户程序则用 X resource 的值) 将仍然是有用的, 可以用来重设由 `LC_CTYPE` 等环境变量指定的默认值.

## 2.8 我怎样才能得到 UTF-8 版本的 xterm?

在 XFree86 里带的 xterm 版本最近已经由 Thomas E. Dickey 加入了支持 UTF-8 的扩展. 使用方法是, 获取 xterm patch #119 (1999-10-16) 或更新版本, 用 `./configure --enable-wide-chars ; make` 来编译, 然后用命令行选项 `-u8` 来调用 xterm, 使它将输入输出转换为 UTF-8. 在 UTF-8 模式里使用一个 \*-ISO10646-1 字体. 当你在 ISO 8859-1 模式里时也可以使用 \*-ISO10646-1 字体, 因为 ISO 10646-1 字体与 ISO 8859-1 字体是完全向后兼容的.

新的支持 UTF-8 的 xterm 版本, 以及一些 ISO 10646-1 字体, 将被收录入 XFree86 4.0 版里.

### ✓ xterm 支持组合字符吗?

Xterm 当前只支持级别 1 的 ISO 10646-1, 就是说, 不提供组合字符的支持. 当前, 组合字符将被当作空格字符对待. xterm 将来的修订版很有可能加入某些简单的组合字符支持, 就是仅仅将那个有一个或多个组合字符的基字符加粗 (logical OR-ing). 对于在基线以下的和在小字符上方的重音符来说, 这样处理的结果还是可以接受的. 对于象泰国文字体那样使用特别设计的加粗字符的文字, 这样处理也能工作的很好. 然而, 对于某些字体里, 在较高的字符上方组合上的重音符, 特别是对于 "fixed" 字体族, 产生的结果就不完全令人满意了. 因此, 在可用的地方, 应该继续优先使用预作字符.

### ✓ xterm 支持半宽与全宽 CJK 字体吗?

Xterm 当前只支持那种所有字形都等宽的 cell-spaced 的字体. 将来的修订版很有可能为 CJK 语言加入半宽与全宽字符支持, 类似于 kterm 提供的那种.

如果选择的普通字体是  $X \times Y$  像素大小, 且宽字符模式是打开的, 那么 `xterm` 会试图装入另外的一个  $2X \times Y$  像素大小的字体 (同样的 `XLFD`, 只是 `AVERAGE_WIDTH` 属性的值翻倍). 它会用这个字体来显示所有在 Unicode Technical Report #11 里被分配了 East Asian Wide (W) 或 East Asian FullWidth (F) 宽度属性的 Unicode 字符. 下面这个 C 函数用来测试一个 Unicode 字符是否是宽字符并需要用覆盖两个字符单元的字形来显示:

```
/* This function tests, whether the ISO 10646/Unicode character code
 * ucs belongs into the East Asian Wide (W) or East Asian FullWidth
 * (F) category as defined in Unicode Technical Report #11. In this
 * case, the terminal emulator should represent the character using a
 * a glyph from a double-wide font that covers two normal (Latin)
 * character cells. */

int iswide(int ucs)
{
    if (ucs < 0x1100)
        return 0;

    return
        (ucs >= 0x1100 && ucs <= 0x115f) || /* Hangul Jamo */
        (ucs >= 0x2e80 && ucs <= 0xa4cf && (ucs & ~0x0011) != 0x300a &&
         ucs != 0x303f) || /* CJK ... Yi */
        (ucs >= 0xac00 && ucs <= 0xd7a3) || /* Hangul Syllables */
        (ucs >= 0xf900 && ucs <= 0xfaff) || /* CJK Compatibility Ideographs */
        (ucs >= 0xfe30 && ucs <= 0xfe6f) || /* CJK Compatibility Forms */
        (ucs >= 0xff00 && ucs <= 0xff5f) || /* Fullwidth Forms */
        (ucs >= 0xffe0 && ucs <= 0xffe6);
}
```

某些 C 库也提供了函数

```
#include <wchar.h>
int wewidth(wchar_t wc);
int wcswidth(const wchar_t *pwcs, size_t n);
```

用来测定该宽字符 `wc` 或由 `pwcs` 指向的字符串中的 `n` 个宽字符码 (或者

少于  $n$  个宽字符码, 如果在  $n$  个宽字符码之前遇到一个空宽字符的话) 所要求的列位置的数量. 这些函数定义在 Open Group 的 Single UNIX Specification 里. 一个拉丁/希腊/斯拉夫/等等的字符要求一个列位置, 一个 CJK 象形文字要求两个, 一个组合字符要求零个。

## ✓ 2.9 最终 xterm 是否会支持从右到左的书写?

此刻还没有给 xterm 增加从右到左功能的计划. 希伯来与阿拉伯用户因此不得不靠应用程序在将希伯来文与阿拉伯文字符串送到终端前按左方向翻转它们, 换句话说, 双向处理必须在应用程序里完成, 而不是在 xterm 里. 至少, 希伯来与阿拉伯文在预作字形的可用性的形式上, 以及提示表格上的支持, 比 ISO 8859 要有所改进. 现在还远没有决定 xterm 是否支持双向文字以及该怎样工作. ISO 6429 = ECMA-48 和 Unicode bidi algorithm 都提供了可供选择的开始点, 可以参考 ECMA Technical

Report TR/53. Xterm 也不处理阿拉伯文, Hangul 或 印度文本的格式化算法, 而且现在还不太清楚在 VT100 模拟器里处理是否可行和值得, 或者应该留给应用软件去处理. 如果你打算在你的应用程序里支持双向文字输出, 看一下 FriBidi, Dov Grobgeld 的 Unicode 双向算法的自由实现。

## 2.9 我在哪儿能找到 ISO 10646-1 X11 字体?

在过去的几个月里出现了相当多的 X11 的 Unicode 字体, 并且还在快速增多.

Markus Kuhn 正和其他许多志愿者一起工作于手动将旧的 -misc-fixed-\*-iso8859-1 字体扩展到覆盖所有的欧洲字符表 (拉丁, 希腊, 斯拉夫, 国际音标字母表. 数学与技术符号, 某些字体里甚至有亚美尼亚语, 乔治亚语, 片假名等). 更多信息请参考 Unicode fonts and tools for X11 页. 这些字体将与 XFree86 一起分发. 例如字体

-misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso10646-1

(旧的 xterm 的 fixed 缺省字体的一个扩展, 包括超过 3000 个字符) 已经是 XFree86 3.9 snapshot 的一部分了.

Markus 也做好了 X11R6.4 distribution 里所有的 Adobe 和 B&H BDF 字

体的 ISO 10646 版本. 这些字体已经包含了全部 Postscript 字体表 (大约 30 个额外的字符, 大部分也被 CP1252 MS-Windows 使用, 如 smart quotes, dashes 等), 在 ISO 8859-1 编码下是没有的. 它们在 ISO 10646-1 版本里是完全可用的.

XFree86 4.0 将携带一个集成的 TrueType 字体引擎, 这使得你的 X 应用程序可以将任何 Apple/Microsoft 字体用于 ISO 10646-1 编码.

将来的 XFree86 版本很有可能从分发版中去除大多数旧的 BDF 字体, 取而代之的是 ISO 10646-1 编码的版本. X 服务器则会增加一个自动编码转换器, 只有当旧的 8 位软件请求一个类似于 ISO 8859-\* 编码的字体时, 才虚拟地从 ISO 10646-1 字体文件中创建一个这样的字体. 现代软件应该优先地直接使用 ISO 10646-1 字体编码.

ClearlyU (cu12) 是一个非常有用的 X11 的 12 点阵, 100 dpi 的 proportional ISO 10646-1 BDF 字体, 包含超过 3700 个字符, 由 Mark Leisher 提供 (样例图象).

Roman Czyborra 的 GNU Unicode font 项目工作于收集一个完整的与免费的 8×16/16×16 pixel Unicode 字体. 目前已经覆盖了 34000 个字符.

etl-unicode 是一个 ISO 10646-1 BDF 字体, 由 Primoz Peterlin 提供.

Unicode X11 字体名字以 -ISO10646-1 结尾. 这个 X 逻辑字体描述器 (X Logical Font Descriptor, XLFD) 的 CHARSET\_REGISTRY 和 CHARSET\_ENCODING 域里的值已经为所有 Unicode 和 ISO 10646-1 的 16 位字体而正式地注册了. 每个 \*-ISO10646-1 字体都包含了整个 Unicode 字符集里的某几个子集, 而用户必须弄清楚他们选择的字体覆盖哪几个他们需要的字符子集.

\*-ISO10646-1 字体通常也指定一个 DEFAULT\_CHAR 值, 指向一个非 Unicode 字形, 用来表示所有在该字体里不可用的字符 (通常是一个虚线框, 一个 H 的大小, 位于 0x1F 或 0xFFFE). 这使得用户至少能知道这儿有一个不支持的字符. xterm 用的小的定宽字体比如 6x13 等, 将永远无法覆盖所有的 Unicode, 因为许多文字比如日本汉字只能用比欧洲用户广泛使用的大的象素尺寸才能表示. 欧洲使用的典型的 Unicode 字体将只包含大约 1000 到 3000 个字符的子集.

## 2.10 我怎样才能找出一个 X 字体里有哪些字形？

X 协议无法让一个应用程序方便地找出一个 cell-spaced 字体提供哪些字形，它没有为字体提供这样的量度。因此 Mark Leisher 和 Erik van de Poel (Netscape) 指定了一个新的 `_XFREE86_GLYPH_RANGES` BDF 属性，告诉应用程序该 BDF 字体实现了哪个 Unicode 子集。Mark Leisher 提供了一些样例代码以产生并扫描这个属性，而 Xmbdfed 3.9 以及更高版本将自动将其加入到由它产生的每个 BDF 文件里。

## 2.11 与 UTF-8 终端模拟器相关的问题是什么？

VT100 终端模拟器接受 ISO 2022 (=ECMA-35) ESC 序列，用于在不同的字符集间切换。

UTF-8 在 ISO 2022 的意义里是一个 "其他编码系统" (参考 ECMA 35 的 15.4 节)。UTF-8 是在 ISO 2022 SS2/SS3/G0/G1/G2 /G3 世界之外的，因此如果你从 ISO 2022 切换到 UTF-8，所有的 SS2/SS3/G0/G1/G2/G3 状态都变得没有意义了，直到你离开 UTF-8 并切换回 ISO 2022。UTF-8 是一个没有国家的编码，也就是一个自我终结的短字节序列完全决定了它代表什么字符，独立于任何国家的切换。G0 与 G1 在 ISO 10646 里与在 ISO 8859-1 里相同，而 G2/G3 在 ISO 10646 里不存在，因为任何字符都有固定的位置，因而不会发声切换。在 UTF-8 模式下，你的终端不会因为你偶然地装入一个二进制文件而切换入一种奇怪图形字符模式。这使得一个终端在 UTF-8 模式下比在 ISO 2022 模式下要健壮得多，而且因此可以有办法将终端锁在 UTF-8 模式里，而不会偶然地回到 ISO 2022 世界里。

ISO 2022 标准指定了一系列的 ESC % 序列，以离开 ISO 2022 世界 (指定其他的编码系统, DOCS)，用于 UTF-8 的许多这样的序列已经注册进了 ISO 2375 International Register of Coded Character Sets:

ESC %G 从 ISO 2022 里激活一个未指定实现级别的 UTF-8 模式且允许再返回 ISO 2022。

ESC %@ 从 UTF-8 回到 ISO 2022，条件是通过 ESC %G 进入的 UTF-8  
ESC %G 切换进 UTF-8 级别 1 且不返回。

ESC %H 切换进 UTF-8 级别 2 且不返回。



ESC %I 切换进 UTF-8 级别 3 且不返回。

当一个终端模拟器在 UTF-8 模式时, 任何 ISO 2022 逃脱码序列例如用于切换 G2/G3 等的都被忽略. 一个在 UTF-8 模式下的终端模拟器唯一会执行的 ISO 2022 序列是 ESC %@ 以从 UTF-8 返回 ISO 2022 方案.

UTF-8 仍然允许你使用象 CSI 这样的 C1 控制字符, 尽管 UTF-8 也使用 0x80-0x9F 范围里的字节. 重要的是必须理解在 UTF-8 模式下的终端模拟器必须在执行任何控制字符前对收到的字节流运用 UTF-8 解码器. C1 字符与其他任何大于 U+007F 的字符一样需先经过 UTF-8 解码.

## 2.12 已经有哪些支持 UTF-8 的应用程序了?

Yudit 是 Gaspar Sinai 的自由 X11 Unicode 编辑器

Mined 98 由 Thomas Wolff 提供, 是一个可以处理 UTF-8 的文本编辑器.

less 版本 346 或更高, 支持 UTF-8

C-Kermit 7.0 在传输, 终端, 及文件字符集方面支持 UTF-8.

Sam 是 Plan9 的 UTF-8 编辑器, 类似于 vi, 也可用于 Linux 和 Win32. (Plan9 是第一个完全转向 UTF-8, 将其作为字符编码的操作系统.)

9term 由 Matty Farrow 提供, 是一个 Plan9 操作系统的 Unicode/UTF-8 终端模拟器的 Unix 移植.

Wily 是一个 Plan9 Acme 编辑器的 Unix 实现.

ucm-0.1 是 Juliusz Chroboczek 的 Unicode 字符映射表, 一个小工具, 使你可以选中任何一个 Unicode 字符并粘贴进你的应用程序.

有哪些用于改善 UTF-8 支持的补丁?

Robert Brady 提供了一个 patch for less 340 (现在已经合并进了 less 344)

Bruno Haible 提供了用于 stty, Linux 核心 tty 等的 多个补丁.

Otfried Cheong 编写了 Unicode encoding for GNU Emacs 工具箱, 使 Mule 能够处理 UTF-8 文件.

Postscript 字形的名字与 UCS 代码是怎么关联的?

参考 Adobe 的 Unicode and Glyph Names 指南.



## 2.13 X11 的剪切与粘贴工作在 UTF-8 时是如何完成的?

参考 Juliusz Chroboczek 的客户机间 Unicode 文本的交换 草案, 对 ICCCM 的一个扩充的建议, 用一个新的可用于属性类型(property type)和选中(selection)目标的原子 UTF8\_STRING 来处理 UTF-8 的选中.

现在有没有用于处理 Unicode 的免费的库?

IBM Classes for Unicode

Mark Leisher 的 UCData Unicode 字符属性库和 wchar\_t 支持测试码.

各种 X widget 对 Unicode 支持的现状如何?

GScript - Unicode 与复杂文本处理 是一个为 GTK+ 增加全功能的 Unicode 支持的项目.

Qt 2.0 现在支持使用 \*-ISO10646-1 字体了.

FriBidi 是 Dov Grobgeld 的 Unicode 双向算法的免费实现.

有什么关于这个话题的好的邮件列表?

你确实应该订阅的是 unicode@unicode.org 邮件列表, 这是发现标准的作者和其他许多领袖的话语的最好办法. 订阅方法是, 用 "subscribe" 作为标题, "subscribe YOUR@EMAIL.ADDRESS unicode" 作为正文, 发一条消息到 unicode-request@unicode.org.

也有一个专注与改进通常用于 GNU/Linux 系统上应用程序的 UTF-8 支持的邮件列表 linux-utf8@nl.linux.org. 订阅方法是, 以 "subscribe linux-utf8" 为内容, 发送消息到 majordomo@nl.linux.org. 你也可以浏览 linux-utf8 archive

其他相关的还有 XFree86 组的 "字体" 与 "i18n" 列表, 但你必须成为一名正式的开发人员才能订阅.

## 2.14 更多参考

Bruno Haible's Unicode HOWTO.

The Unicode Standard, Version 2.0

Unicode Technical Reports

Mark Davis' Unicode FAQ

ISO/IEC 10646-1:1993

Frank Tang's International Secrets

Unicode Support in the Solaris 7 Operating Environment

The USENIX paper by Rob Pike and Ken Thompson on the introduction of UTF-8 under Plan9 reports about the first operating system that migrated already in 1992 completely to UTF-8 (which was at the time still called UTF-2).

Li18nux is a project initiated by several Linux distributors to enhance Unicode support for Linux.

The Online Single Unix Specification contains definitions of all the ISO C Amendment 1 function, plus extensions such as `wcwidth()`.

The Open Group's summary of ISO C Amendment 1.

GNU libc

The Linux Console Tools

The Unicode Consortium character database and character set conversion tables are an essential resource for anyone developing Unicode related tools.

Other conversion tables are available from Microsoft and Keld Simonsen's WG15 archive.

Michael Everson's ISO10646-1 archive contains online versions of many of the more recent ISO 10646-1 amendments, plus many other goodies. See also his Roadmaps to the Universal Character Set.

An introduction into The Universal Character Set (UCS).

Otfried Cheong's essay on Han Unification in Unicode

The AMS STIX project is working on revising and extending the mathematical characters for Unicode 4.0 and ISO 10646-2.

Jukka Korpela's Soft hyphen (SHY) - a hard problem? is an excellent discussion of the controversy surrounding U+00AD.

James Briggs' Perl, Unicode and I18N FAQ.

## 第 7 章 触摸屏应用

---

### 课程内容：

- ✧ 触摸屏工作原理
- ✧ 电阻式触摸屏
- ✧ 电容式触摸屏
- ✧ 触摸屏日常维护
- ✧ **iphone 触摸屏和 5800 触摸屏工作原理**

## 第 7 章 触摸屏应用

### 1、触摸屏工作原理

#### 1.1 触摸屏的基本原理

典型触摸屏的工作部分一般由三部分组成，如图 1 所示：两层透明的阻性导体层、两层导体之间的隔离层、电极。阻性导体层选用阻性材料，如铟锡氧化物 (ITO) 涂在衬底上构成，上层衬底用塑料，下层衬底用玻璃。隔离层为粘性绝缘液体材料，如聚脂薄膜。电极选用导电性能极好的材料(如银粉墨)构成，其导电性能大约为 ITO 的 1000 倍。

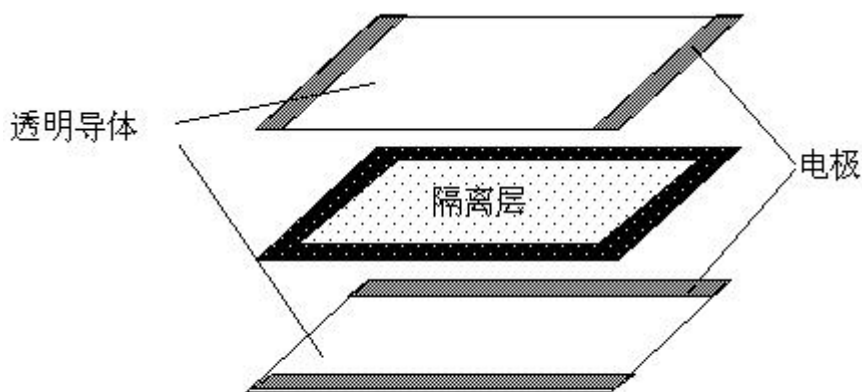


图 1 触摸屏结构

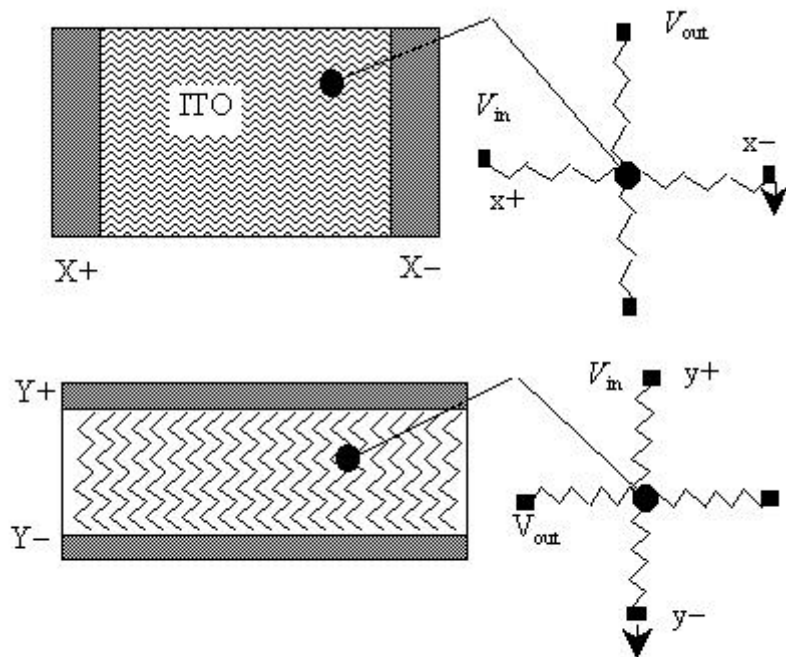


图 2 工作时的导体层

触摸屏工作时，上下导体层相当于电阻网络，如图 2 所示。当某一层电极加上电压时，会在该网络上形成电压梯度。如有外力使得上下两层在某一点接触，则在电极未加电压的另一层可以测得接触点处的电压，从而知道接触点处的坐标。比如，在顶层的电极(X+,X-)上加上电压，则在顶层导体层上形成电压梯度，当有外力使得上下两层在某一点接触，在底层就可以测得接触点处的电压，再根据该电压与电极(X+)之间的距离关系，知道该处的 X 坐标。然后，将电压切换到底层电极(Y+,Y-)上，并在顶层测量接触点处的电压，从而知道 Y 坐标。

## 1.2 触摸屏的控制实现

现在很多 PDA 应用中，将触摸屏作为一个输入设备，对触摸屏的控制也有专门的芯片。很显然，触摸屏的控制芯片要完成两件事情：**其一，是完成电极电压的切换；其二，是采集接触点处的电压值(即 A/D)**。本文以 BB (Burr-Brown) 公司生产的芯片 ADS7843 为例，介绍触摸屏控制的实现。

## ✓ ADS7843 的基本特性与典型应用

ADS7843 是一个内置 12 位模数转换、低导通电阻模拟开关的串行接口芯片。供电电压 2.7~5 V，参考电压 VREF 为 1 V~+VCC，转换电压的输入范围为 0~VREF，最高转换速率为 125 kHz。ADS7843 的引脚配置如图 3 所示。表 1 为引脚功能说明，图 4 为典型应用。

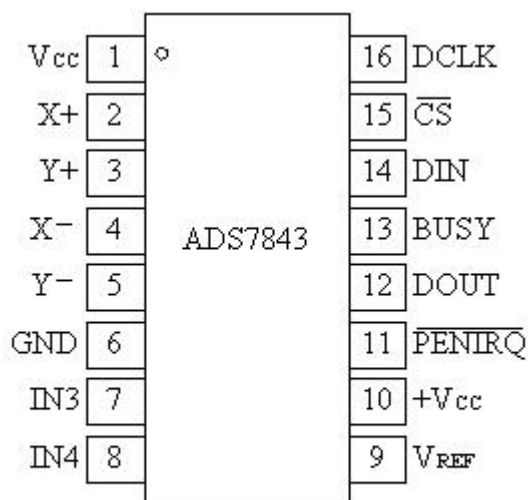


图 3 ADS7843 引脚

表 1 引脚功能说明

引脚号	引脚名	功能描述
1,10	+V <sub>CC</sub>	供电电源 2.7~5 V
2,3	X+,Y+	接触触摸屏正电极，内部 A/D 通道
4,5	X-,Y-	接触触摸屏负电极
6	GND	电源地
7,8	IN3,IN4	两个附属 A/D 输入通道
9	V <sub>REF</sub>	A/D 参考电压输入
11	PENIRQ	中断输出，须接外拉电阻 (10 kΩ 或 100 kΩ)
12,14,16	DOUT,DIN,DCLK	串行接口引脚，在时钟下降沿数据移出，上升沿移进
13	BUSY	忙指示，低电平有效
15	CS	片选

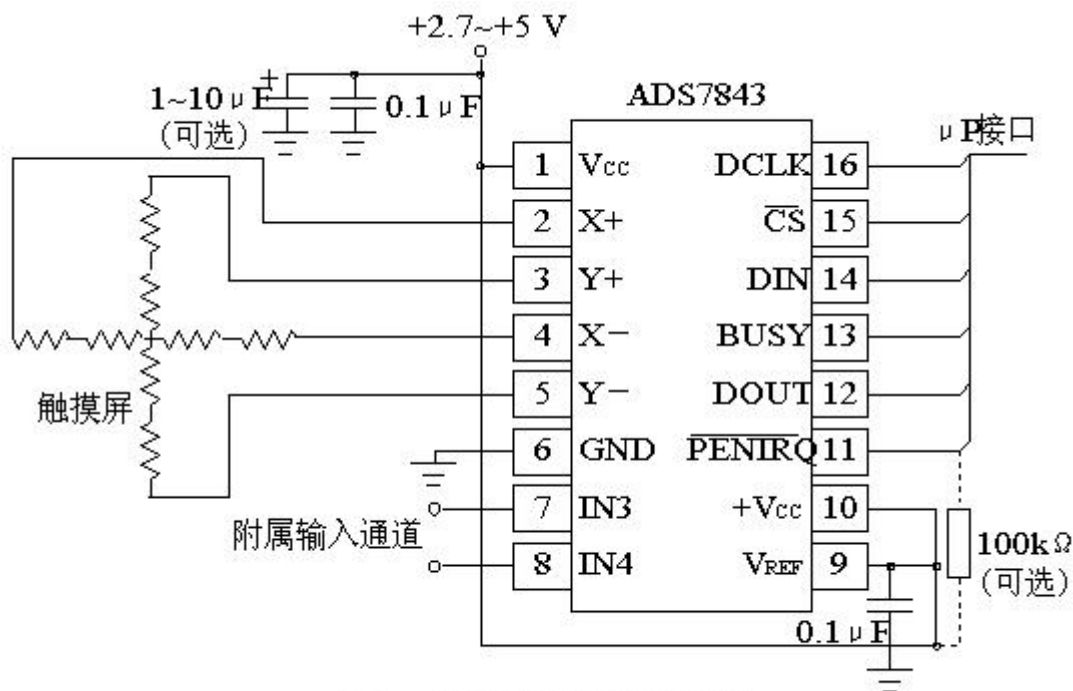


图4 ADS7843的典型应用

#### ✓ ADS7843的内部结构及参考电压模式选择

ADS7843之所以能实现对触摸屏的控制，是因为其内部结构很容易实现电极电压的切换，并能进行快速A/D转换。图5所示为其内部结构，A2~A0和SER/DFR为控制寄存器中的控制位，用来进行开关切换和参考电压的选择。

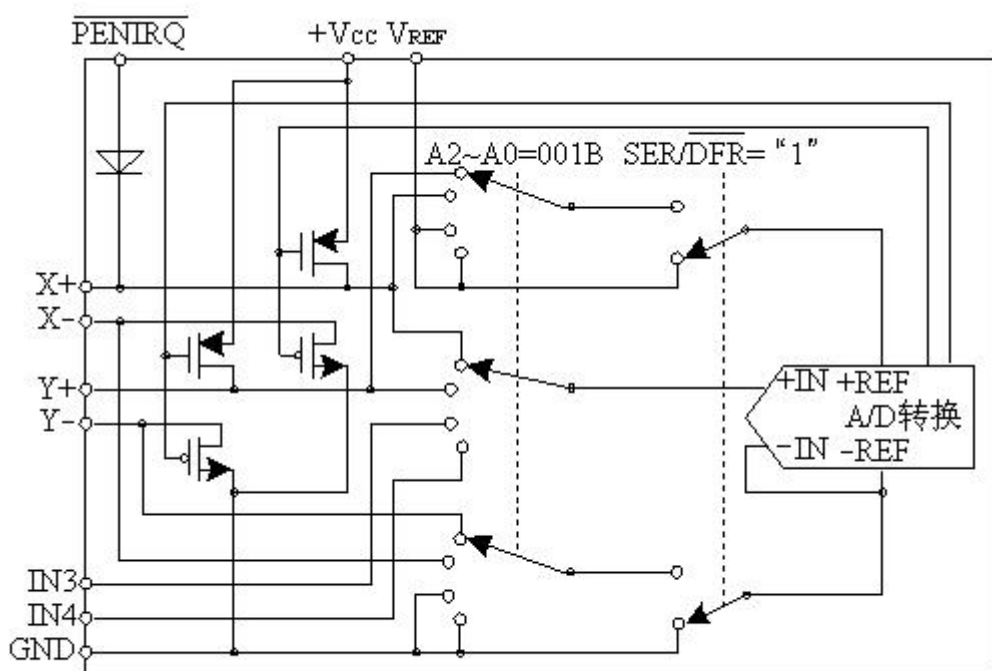


图5 ADS7843内部结构



ADS7843 支持两种参考电压输入模式：一种是参考电压固定为  $V_{REF}$ ，另一种采取差动模式，参考电压来自驱动电极。这两种模式分别如图 6(a)、(b)所示。采用图 6(b)的差动模式可以消除开关导通压降带来的影响。表 2 和表 3 为两种参考电压输入模式所对应的内部开关状况。

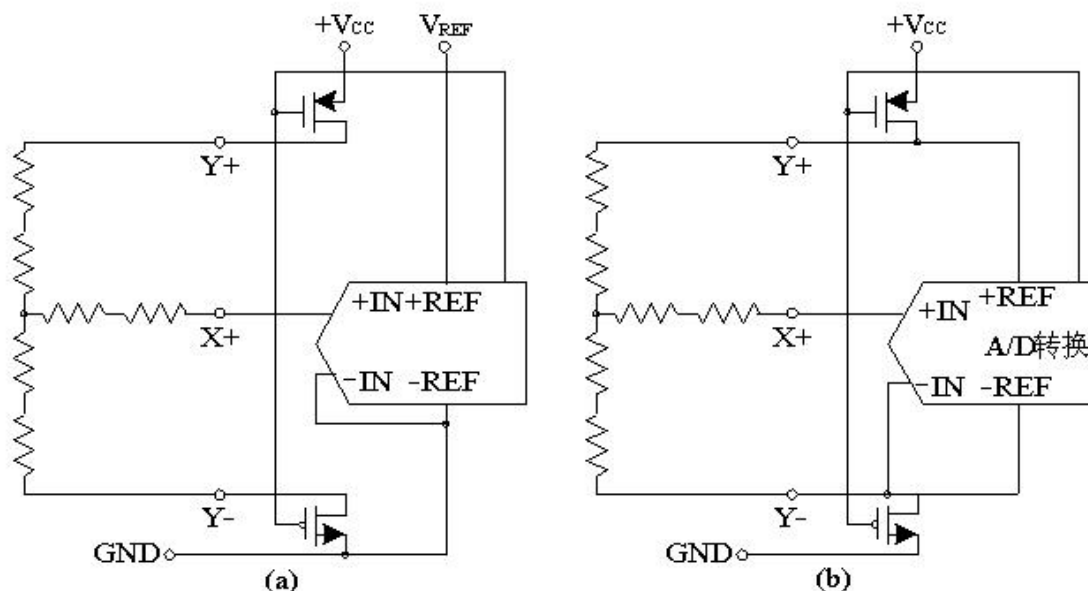


图 6 参考电压输入模式

表 3 参考电压差动输入模式 (SER/DFR = "0")

A2	A1	A0	X+	Y+	IN3	IN4	-IN	X 开关	Y 开关	+ REF	-REF
0	0	1	+IN				-Y	OFF	ON	+Y	-Y
1	0	1		+IN			-X	ON	OFF	+X	-X
0	1	0			+IN		GND	OFF	OFF	+V <sub>REF</sub>	GND
1	1	0				+IN	GND	OFF	OFF	+V <sub>REF</sub>	GND

表 4 ADS7843 的控制字

bit7(MSB)	bit6	bit5	bit4	bit3	bit2	bit1	bit0
S	A2	A1	A0	MODE	SER/DFR	PD1	PD0

### ✓ ADS7843 的控制字及数据传输格式

ADS7843 的控制字如表 4 所列，其中 S 为数据传输起始标志位，该位必为 "1"。A2~A0 进行通道选择(见表 2 和 3)。

MODE 用来选择 A/D 转换的精度，"1"选择 8 位，"0"选择 12 位。

SER/选择参考电压的输入模式(见表 2 和 3)。PD1、PD0 选择省电模式：

- ✚ "00"省电模式允许，在两次 A/D 转换之间掉电，且中断允许；
- ✚ "01"同"00"，只是不允许中断；
- ✚ "10"保留；
- ✚ "11"禁止省电模式。

为了完成一次电极电压切换和 A/D 转换，需要先通过串口往 ADS7843 发送控制字，转换完成后再通过串口读出电压转换值。标准的一次转换需要 24 个时钟周期，如图 7 所示。由于串口支持双向同时进行传送，并且在一次读数与下一次发控制字之间可以重叠，所以转换速率可以提高到每次 16 个时钟周期，如图 8 所示。如果条件允许，CPU 可以产生 15 个 CLK 的话(比如 FPGAs 和 ASICs)，转换速率还可以提高到每次 15 个时钟周期，如图 9 所示。

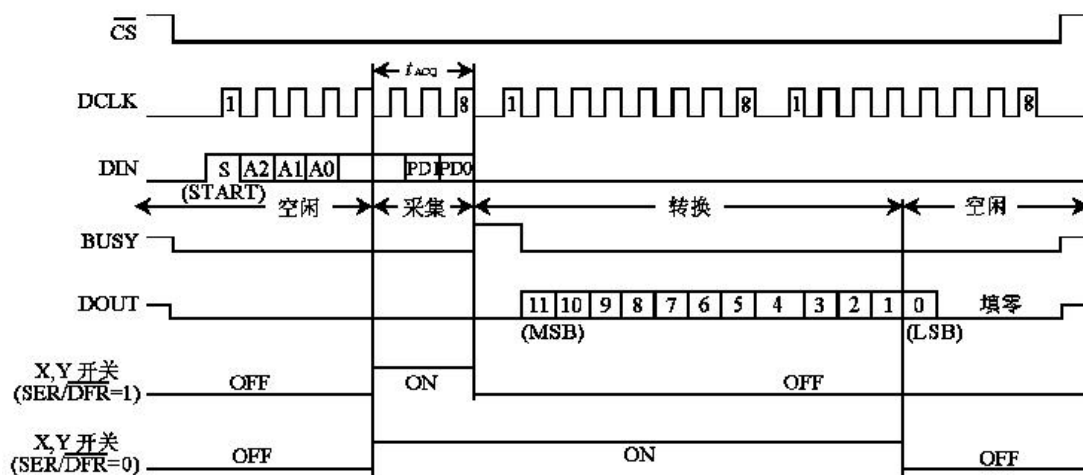


图 7 A/D 转换时序 (每次转换需 24 个时钟周期)

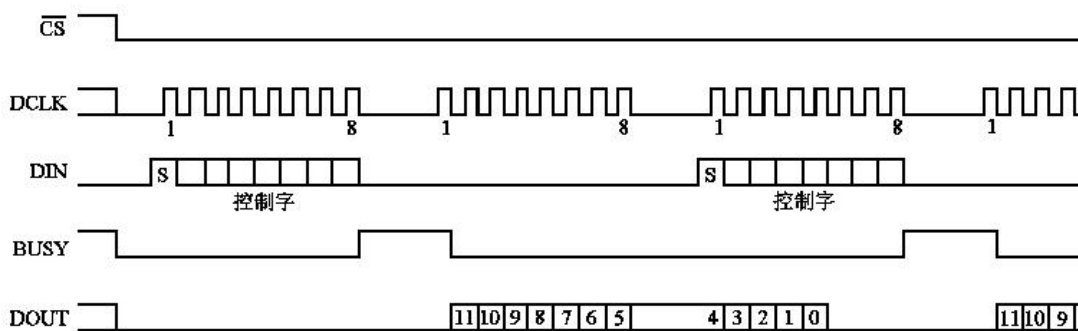


图 8 A/D 转换时序 (每次转换需 16 个时钟周期)

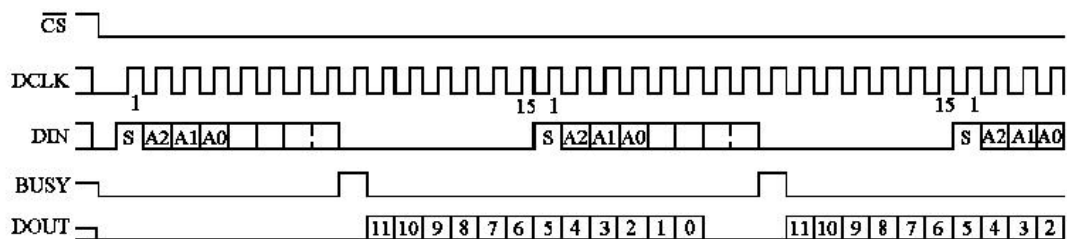


图9 A/D 转换时序（每次转换需 15 个时钟周期）

## 2、电阻式触摸屏

### ✓ 简介

很多 LCD 模块都采用了电阻式触摸屏，这些触摸屏等效于将物理位置转换为代表 X、Y 坐标的电压值的传感器。通常有 4 线、5 线、7 线和 8 线触摸屏来实现，本文详细介绍了 SAR 结构、四种触摸屏的组成结构和实现原理，以及检测触摸的方法。

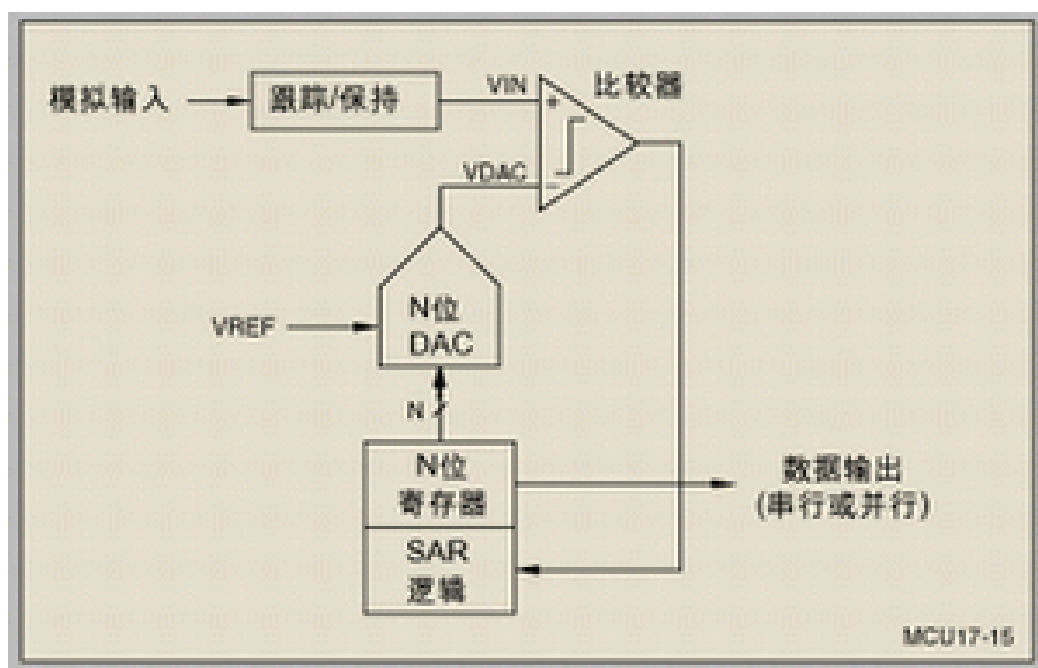


图 1: 简化的 N 位 SAR 结构

电阻式触摸屏是一种传感器，它将矩形区域中触摸点(X,Y)的物理位置转换为代表 X 坐标和 Y 坐标的电压。很多 LCD 模块都采用了电阻式触摸屏，这种屏幕可以用四线、五线、七线或八线来产生屏幕偏置电压，同时读回触摸点的电压。

过去, 为了将电阻式触摸屏上的触摸点坐标读入微控制器, 需要使用一个专用的触摸屏控制器芯片, 或者利用一个复杂的外部开关网络来连接微控制器的片上模数转换器(ADC)。夏普公司的 LH75400/01/10/11 系列和 LH7A404 等微控制器都带有一个内含触摸屏偏置电路的片上 ADC, 该 ADC 采用了一种逐次逼近寄存器(SAR)类型的转换器。采用这些控制器可以实现在触摸屏传感器和微控制器之间进行直接接口, 无需 CPU 介入的情况下控制所有的触摸屏偏置电压, 并记录全部测量结果。本文将详细介绍四线、五线、七线和八线触摸屏的结构和实现原理, 在下期的文章中将介绍触摸屏与 ADC 的接口与编程。

## SAR 结构

SAR 的实现方法很多, 但它的基本结构很简单, 参见图 1。该结构将模拟输入电压(VIN)保存在一个跟踪/保持器中, N 位寄存器被设置为中间值(即  $100\dots0$ , 其中最高位被设置为 1), 以执行二进制查找算法。因此, 数模转换器(DAC)的输出(VDAC)为 VREF 的二分之一, 这里 VREF 为 ADC 的参考电压。之后, 再执行一个比较操作, 以决定 VIN 小于还是大于 VDAC:

✚ 如果 VIN 小于 VDAC, 比较器输出逻辑低, N 位寄存器的最高位清 0。

✚ 如果 VIN 大于 VDAC, 比较器输出逻辑高(或 1), N 位寄存器的最高位保持为 1。

其后, SAR 的控制逻辑移动到下一位, 将该位强制置为高, 再执行下一次比较。SAR 控制逻辑将重复上述顺序操作, 直到最后一位。当转换完成时, 寄存器中就得到了一个 N 位数据字。

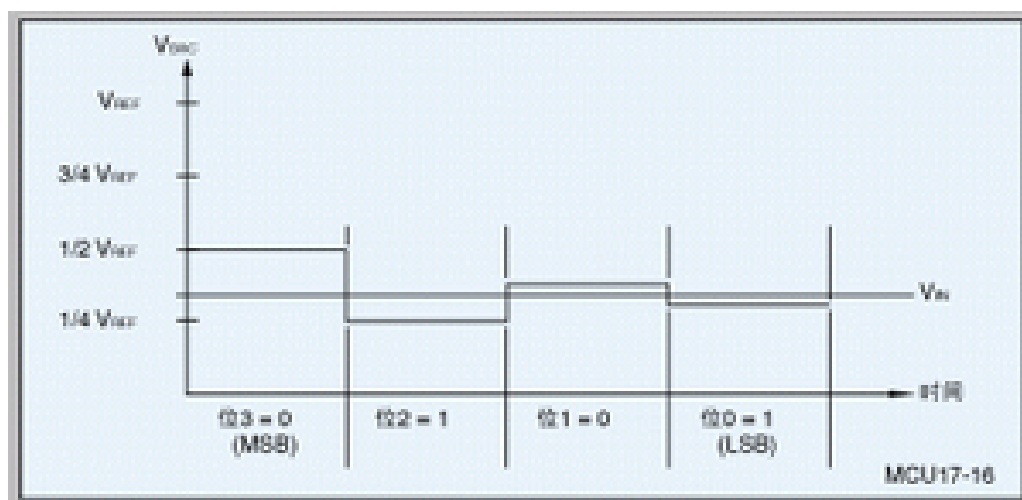


图 2: 4 位 SAR ADC 工作示意图

图 2 显示了一个 4 位转换过程的例子，图中 Y 轴和粗线表示 DAC 的输出电压。在本例中：

- ✚ 第一次比较显示  $V_{IN}$  小于  $V_{DAC}$ ，因此位[3]被置 0。随后 DAC 被设置为 0b0100 并执行第二次比较。
- ✚ 在第二次比较中， $V_{IN}$  大于  $V_{DAC}$ ，因此位[2]保持为 1。随后，DAC 被设置为 0b0110 并执行第三次比较。
- ✚ 在第三次比较中，位[1]被置 0。DAC 随后被设置为 0b0101，并执行最后一次比较。
- ✚ 在最后一次比较中，由于  $V_{IN}$  大于  $V_{DAC}$ ，位[0]保持为 1。

### ✓ 触摸屏原理

触摸屏包含上下叠合的两个透明层，四线和八线触摸屏由两层具有相同表面电阻的透明阻性材料组成，五线和七线触摸屏由一个阻性层和一个导电层组成，通常还要用一种弹性材料来将两层隔开。当触摸屏表面受到的压力(如通过笔尖或手指进行按压)足够大时，顶层与底层之间会产生接触。所有的电阻式触摸屏都采用分压器原理来产生代表 X 坐标和 Y 坐标的电压。如图 3 所示，分压器是通过将两个电阻进行串联来实现的。上面的电阻( $R_1$ )连接正参考电压( $V_{REF}$ )，下面的电阻( $R_2$ )接地。两个电阻连接点处的电压测量值与下面那个电阻的阻值成正比。

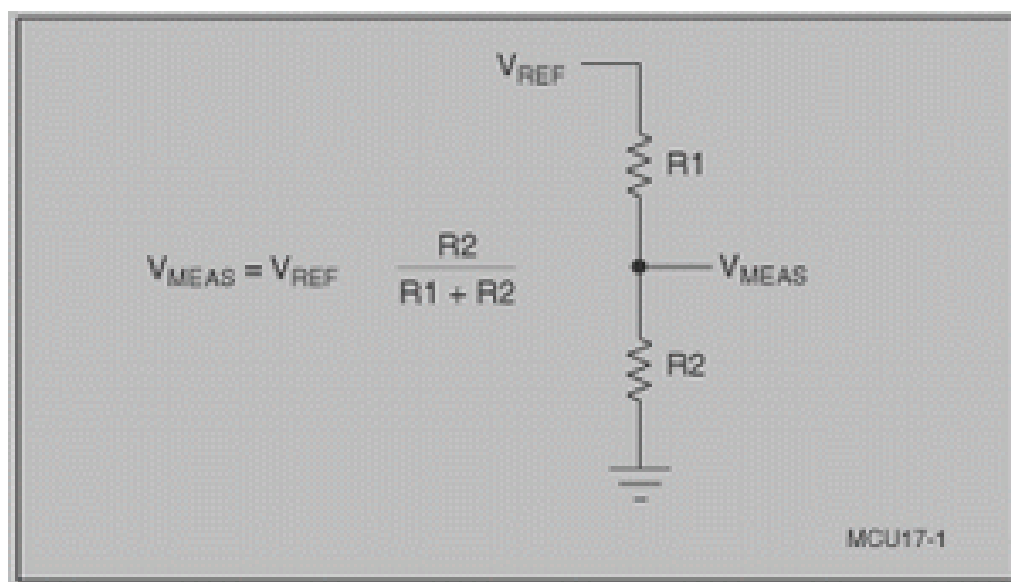


图 3：等效电压分压器电路

为了在电阻式触摸屏上的特定方向测量一个坐标，需要对一个阻性层进行偏置：将它的一边接  $V_{REF}$ ，另一边接地。同时，将未偏置的那一层连接到一个

ADC 的高阻抗输入端。当触摸屏上的压力足够大，使两层之间发生接触时，电阻性表面被分隔为两个电阻。它们的阻值与触摸点到偏置边缘的距离成正比。触摸点与接地边之间的电阻相当于分压器中下面的那个电阻。因此，在未偏置层上测得的电压与触摸点到接地边之间的距离成正比。

#### ✓ 四线触摸屏

四线触摸屏包含两个阻性层。其中一层在屏幕的左右边缘各有一条垂直总线，另一层在屏幕的底部和顶部各有一条水平总线，见图 4。为了在 X 轴方向进行测量，将左侧总线偏置为 0V，右侧总线偏置为 VREF。将顶部或底部总线连接到 ADC，当顶层和底层相接触时即可作一次测量。

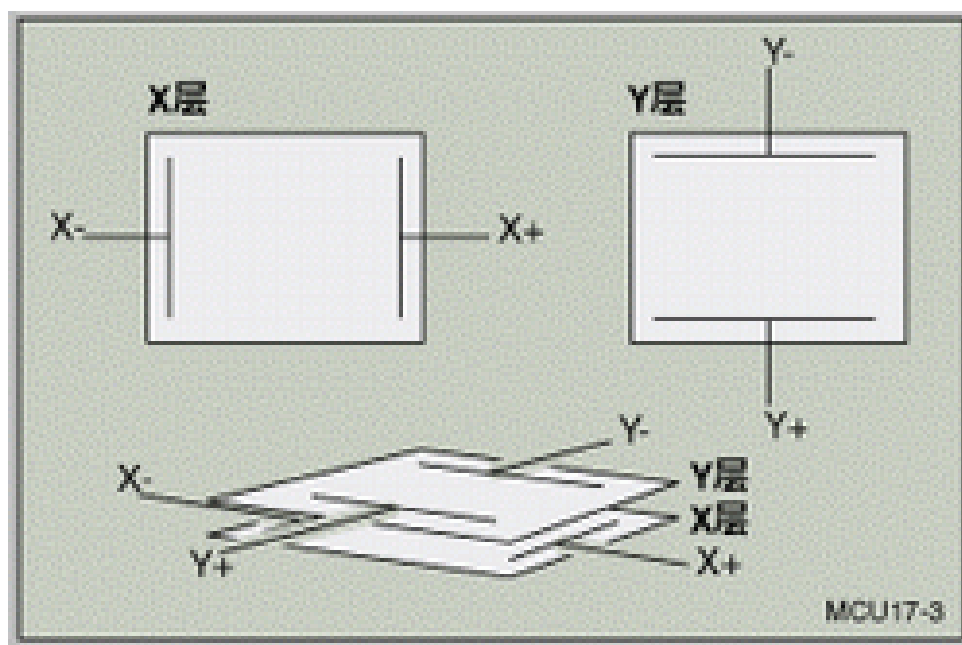


图 4：四线触摸屏

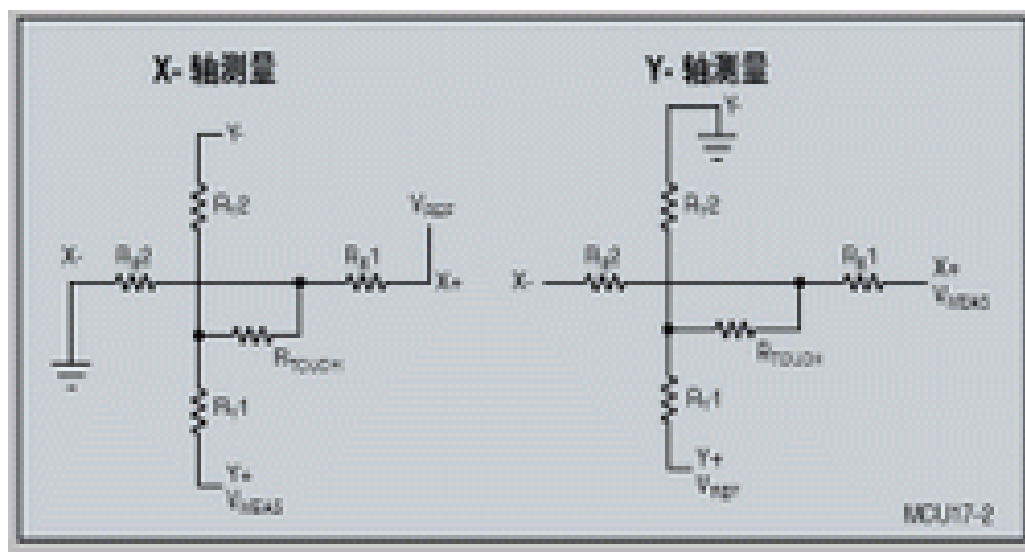


图 5：四线触摸屏在两层相接触时的简化模型

为了在 Y 轴方向进行测量，将顶部总线偏置为 VREF，底部总线偏置为 0V。将 ADC 输入端接左侧总线或右侧总线，当顶层与底层相接触时即可对电压进行测量。图 5 显示了四线触摸屏在两层相接触时的简化模型。对于四线触摸屏，最理想的连接方法是将偏置为 VREF 的总线接 ADC 的正参考输入端，并将设置为 0V 的总线接 ADC 的负参考输入端。

#### ✓ 五线触摸屏

五线触摸屏使用了一个阻性层和一个导电层。导电层有一个触点，通常在其一侧的边缘。阻性层的四个角上各有一个触点。为了在 X 轴方向进行测量，将左上角和左下角偏置到 VREF，右上角和右下角接地。由于左、右角为同一电压，其效果与连接左右侧的总线差不多，类似于四线触摸屏中采用的方法。

为了沿 Y 轴方向进行测量，将左上角和右上角偏置为 VREF，左下角和右下角偏置为 0V。由于上、下角分别为同一电压，其效果与连接顶部和底部边缘的总线大致相同，类似于在四线触摸屏中采用的方法。这种测量算法的优点在于它使左上角和右下角的电压保持不变；但如果采用栅格坐标，X 轴和 Y 轴需要反向。对于五线触摸屏，最佳的连接方法是将左上角(偏置为 VREF)接 ADC 的正参考输入端，将左下角(偏置为 0V)接 ADC 的负参考输入端。图 5：四线触摸屏等效电路。

#### ✓ 七线触摸屏

七线触摸屏的实现方法除了在左上角和右下角各增加一根线之外，与五线触



触摸屏相同。执行屏幕测量时，将左上角的一根线连到VREF，另一根线接SAR ADC的正参考端。同时，右下角的一根线接0V，另一根线连接SAR ADC的负参考端。导电层仍用来测量分压器的电压。

### ✓ 八线触摸屏

除了每条总线上各增加一根线之外，八线触摸屏的实现方法与四线触摸屏相同。对于VREF总线，将一根线用来连接VREF，另一根线作为SAR ADC的数模转换器的正参考输入。对于0V总线，将一根线用来连接0V，另一根线作为SAR ADC的数模转换器的负参考输入。未偏置层上的四根线中，任何一根都可用来测量分压器的电压。

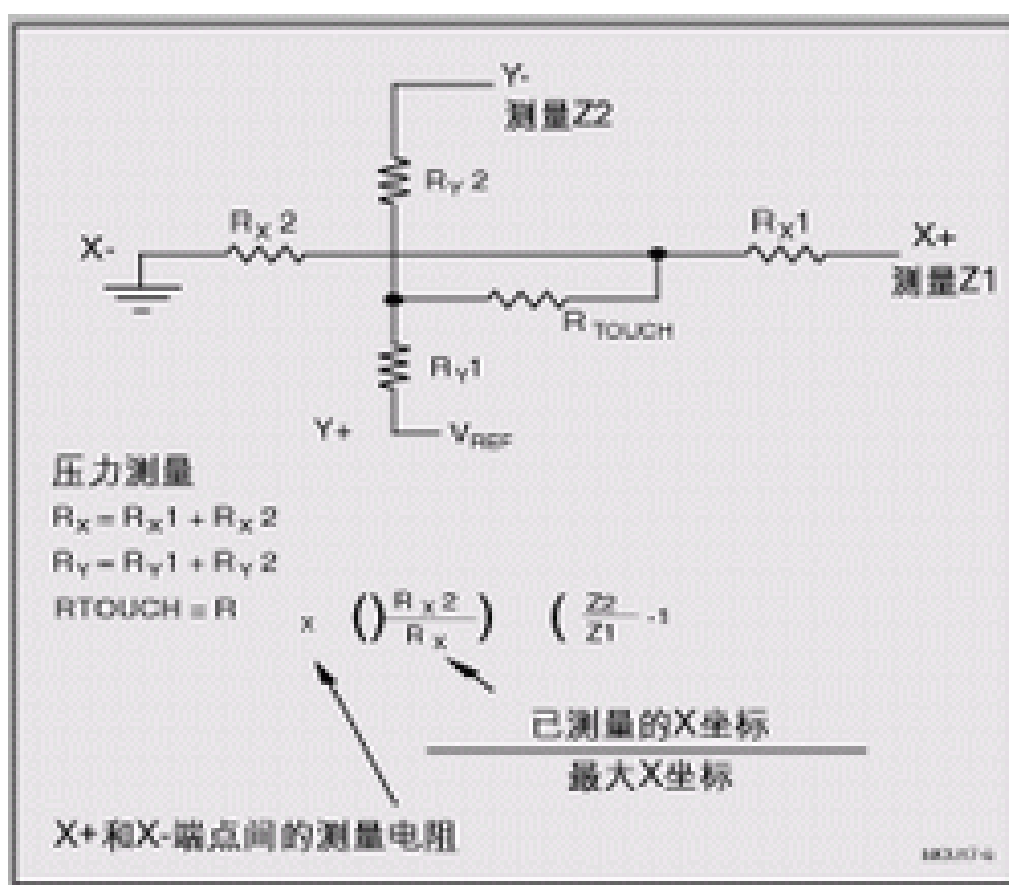


图 6：四线触摸屏压力计算。

### ✓ 检测有无接触

所有的触摸屏都能检测到是否有触摸发生，其方法是用一个弱上拉电阻将其中一层上拉，而用一个强下拉电阻来将另一层下拉。如果上拉层的测量电压大于某个逻辑阈值，就表明没有触摸，反之则有触摸。这种方法存在的问题在于触摸屏是一个巨大的电容器，此外还可能增加触摸屏引线的电容，以便滤除LCD

引入的噪声。弱上拉电阻与大电容器相连会使上升时间变长,可能导致检测到虚假的触摸。

四线和八线触摸屏可以测量出接触电阻,即图 5 中的 RTOUCH。RTOUCH 与触摸压力近似成正比。要测量触摸压力,需要知道触摸屏中一层或两层的电阻。图 6 中的公式给出了计算方法。需要注意的是,如果 Z1 的测量值接近或等于 0(在测量过程中当触摸点靠近接地的 X 总线时),计算将出现一些问题,通过采用弱上拉方法可以有效改善这个问题。

### 3、电容式触摸屏

与电阻式触摸屏不同,电容式触摸屏是利用人体的电流感应进行工作的。

电容式触摸屏的感应屏是一块四层复合玻璃屏,玻璃屏的内表面和夹层各涂有一层导电层,最外层是一薄层矽土玻璃保护层。当我们用手指触摸在感应屏上的时候,人体的电场让手指和和触摸屏表面形成一个耦合电容,对于高频电流来说,电容是直接导体,于是手指从接触点吸走一个很小的电流。这个电流分从触摸屏的四角上的电极中流出,并且流经这四个电极的电流与手指到四角的距离成正比,控制器通过对这四个电流比例的精确计算,得出触摸点的位置。

相比传统的电阻式触摸屏,电容式触摸屏的优势主要有以下几个方面:

- ✚ 操作新奇. 电容式触摸屏支持多点触控,操作更加直观、更具趣味性。
- ✚ 不易误触. 由于电容式触摸屏需要感应到人体的电流,只有人体才能对其进行操作,用其他物体触碰时并不会有所相应,所以基本避免了误触的可能。
- ✚ 耐用度高. 比起电阻式触摸屏,电容式触摸屏在防尘、防水、耐磨等方面有更好的表现。

作为目前正当红的触摸屏技术,电容式触摸屏虽然具有界面华丽、多点触控、只对人体感应等优势,但与此同时,它也有以下几个缺点:

- ✚ 精度不高. 由于技术原因,电容式触摸屏的精度比起电阻式触摸屏还有所欠缺. 而且只能使用手指进行输入,在小屏幕上还很难实现辨识比较复杂的手写输入。
- ✚ 易受环境影响. 温度和湿度等环境因素发生改变时,也会引起电容式触摸屏的不稳定甚至漂移. 例如用户在使用时将身体靠近屏幕就可能引起漂移,甚至在拥挤的人群中操作也会引起漂移. 这主要是由于电容式触摸屏技术的工作原理所致,虽然用户的手指距离屏幕更近,但屏幕附近还有很多体积远大于手指的电场同时作用,这样就会影响到触摸位置的判断。
- ✚ 成本偏高. 此外,当前电容式触摸屏在触控板贴附到 LCD 面板的步骤中还存在

一定的技术困难,良品率并不高,所以无形中也增加了电容式触摸屏的成本。

## 4、触摸屏日常维护

### 4.1 触摸屏的内部结构以及工作原理

触摸屏简单的工作原理实际上就是用手指或其他物体触摸安装在显示器前端的触摸屏时,所触摸的位置(以坐标形式)由触摸屏控制器检测,并通过接口(如 RS-232 串行口)送到 CPU,从而确定输入的信息。

首先来简要介绍一下手机触摸显示屏的大致结构和原理。我们把触摸显示屏分为两个部分:触摸屏和显示屏。触摸屏显示器主要组件触摸屏和显示器集成设备而且具有输入输出设备的功能,按照 4 线触摸屏、6 线触摸屏、8 线触摸屏、红外线式触摸屏、表面声波触摸屏、电容式触摸屏。触摸屏显示器也分有 CRT 触摸屏显示器和 LCD 触摸屏显示器的基本两种触摸屏显示器。

触摸屏又分为上下两块触摸板。为了便以说明,我们在这里称它 A 板和 B 板。A 板一般以有机玻璃作为基层,表面盖有一层经过硬化处理、光滑防刮的塑料层;背面涂有一层透明的 OTI(氧化铟)导电涂层,并由两侧两条导电银胶至 A 板底部的接口线引出,与主板通信。B 板一般采用强化玻璃为材料,表面经过精密的网络格式附上横竖两个方向的 OTI 导电涂层,并由上下两条边的两条银胶连接到 B 板底部的接口。每个网络点到接口引出线的阻值都是递增或递减的。

A、B 两板 OTI 导电涂层之间距离仅为  $2.5\mu\text{m}$ ,并以细小的透明隔离点隔开。我们把 A 板的两条引出线命名为  $x_1$  和  $Y_1$ ,主 B 板的两条引出线命名为  $X_2$  和  $Y_2$ 。触摸显示屏工作时, $x_1$  和  $Y_1$  分别为  $x$  轴和  $y$  轴的基准工作电压输入端(这个基准电压各种机型不尽相同,例如 MOT0388C 采用  $2\text{V}$  电压;波导 E858、868 采用  $2.75\text{V}$ ,CEC2800 采用  $3\text{V}$  电压等)。

当手指接触屏幕, $X_1$  通过 A 板导电层和 B 板上的网络点形成回路,从  $X_2$  输出一个触摸点的  $X$  轴坐标电压;同理, $Y_1$  也同时经过  $Y_2$  输出  $Y$  轴的坐标电压,CPU 通过这两个电压准确计算出触摸点的位置并执行操做,实现用户的触摸功能。这就是触摸显示屏的基本工作原理。由以上原理得知,我们可以通过测量接口引出线  $X_1$ 、 $X_2$ 、 $Y_1$ 、 $Y_2$  之间的阻值来判断触摸显示屏的物理性能。

### 4.2 触摸屏的分类以及日常使用维护

不论何种触摸屏,也不论在日常生活和工作的任何场合使用,都存在正

确使用和维护保养的问题。

- ✚ 表面声波触摸屏：适用于任何非露天的未知使用对象的场合。尤其适合于环境较干净、灰尘少的场合。表面声波触摸屏的感应介质是手指（非指甲、戴手套也可）、橡皮等较软的能与玻璃完全吻合的物品。
- ✚ 电阻压力触摸屏：电阻压力触摸屏的缺陷是怕划伤，因此该触摸屏适合已知对象的固定人员操作使用。电阻压力触摸屏只要给它压力就行，不管是手指、笔杆和其他物品均可触摸，但不能用尖锐和锋利的物品操作。
- ✚ 电容感应触摸屏：不适合在有电磁场干扰和要求精密的场合使用。该触摸屏仅能用手指（非指甲）和肉体接触操作。
- ✚ 红外感应触摸屏：适合于多种非露天的未知使用对象的场合。红外触摸屏的感应介质是任何可阻挡光线的物品，如手指、笔杆、小棍棒等等。

### 4.3 触摸屏的日常维护重点




触摸屏也同普通机器一样需要定期保养维护。并且由于触摸屏是多种电器设备高度集成的触控一体机，所以在使用和维护时应注意以下的一些问题。

- ✚ 应用玻璃清洁剂清洗触摸屏上的脏指印和油污。
- ✚ 水滴或饮料落在屏幕上，会使软件停止反应，这是由于水滴和手指具有相似的特性，需把水滴擦去。特别是当需要贴膜的时候，会有维修人员先喷水，再贴膜，其实这样会严重降低触摸屏的灵敏性，应当注意不要让水进入触摸屏。
- ✚ 触摸屏控制器能自动判断灰尘，但积尘太多会降低触摸屏的敏感性，只需用干布把屏幕擦拭干净。
- ✚ 定期打开机头清洁触摸屏的反射条纹和内表面。具体的方法是：在机内两侧打开盖板，可以找到松开扣住机头前部锁舌的机关，打开机关即可松开锁舌。抬起机头前部，可以看到触摸屏控制卡，拔下触摸屏电缆，向后退机头可卸下机头和触摸屏。仔细看清楚固定触摸屏的方法后，卸下触摸屏清洗，注意不要使用硬纸或硬布，不要划伤反射条纹。最后，按相反顺序和原结构将机头复原。

### 4.4 针对触摸屏的分类也有不同的养护方式：

- ✚ 表面声波触摸屏：在日常使用中应尽量保持触摸屏表面的清洁干净，可用清洁剂及潮湿的抹布将触摸屏表面擦干净。如触摸屏的反射条纹裸露在外，可定期用毛刷清洁发射条纹；如触摸屏安装在显示器里，应经常用毛刷清洁触

触摸屏与显示器边框接触处的灰尘，如果出现触摸反映迟钝或触摸不准的现象，可将一张名片插在触摸屏与显示器外框之间的夹缝中约 3 厘米深，用手轻轻按住触摸屏，将名片沿显示器外框四周反复移动几圈即可（尽量让名片靠紧触摸屏），这样触摸屏即可灵敏如初。

-  电阻压力触摸屏：在日常使用中，主要是避免其被划伤。如果在使用中发现触摸坐标总是固定在某一点，可能是显示器的外框在此处压住了触摸屏，应打开显示器松动一些螺钉，以避免触摸屏受力（一般应专业人员处理）。如触摸屏外表过脏，可用湿抹布清洁。五线、六线电阻压力触摸屏的外层划伤后，一般还能正常使用；四线电阻压力触摸屏的外层一旦划伤触摸屏就很难使用了。另一方面，触摸屏的某一位置由于长时间触摸，会造成触摸屏此处透光度明显降低或表层邹软、起泡，这是只有改动触摸软件的触摸按键，使触摸按键离开此位置。
-  电容感应触摸屏：在日常使用中，应避免用硬物敲击触摸屏，否则，敲击点可能会出现一个小洞，洞的周围相当大的面积将不可触摸。如触摸屏外表过脏，也可用湿抹布清洁。电容感应触摸屏在使用过程中会经常出现“漂移”——触摸不准的现象，应经常执行校准定位程序。
-  红外感应触摸屏：红外触摸屏应放置在不受红外线干扰的场合。红外感应触摸屏如果反映不灵敏，应将触摸屏四周透光部分清洁干净。如触摸屏外表过脏，也可用湿抹布清洁。注意：任何触摸屏一旦屏体破碎或划伤，即失去了免费保修的权利，因此每个使用者都应爱惜触摸屏。

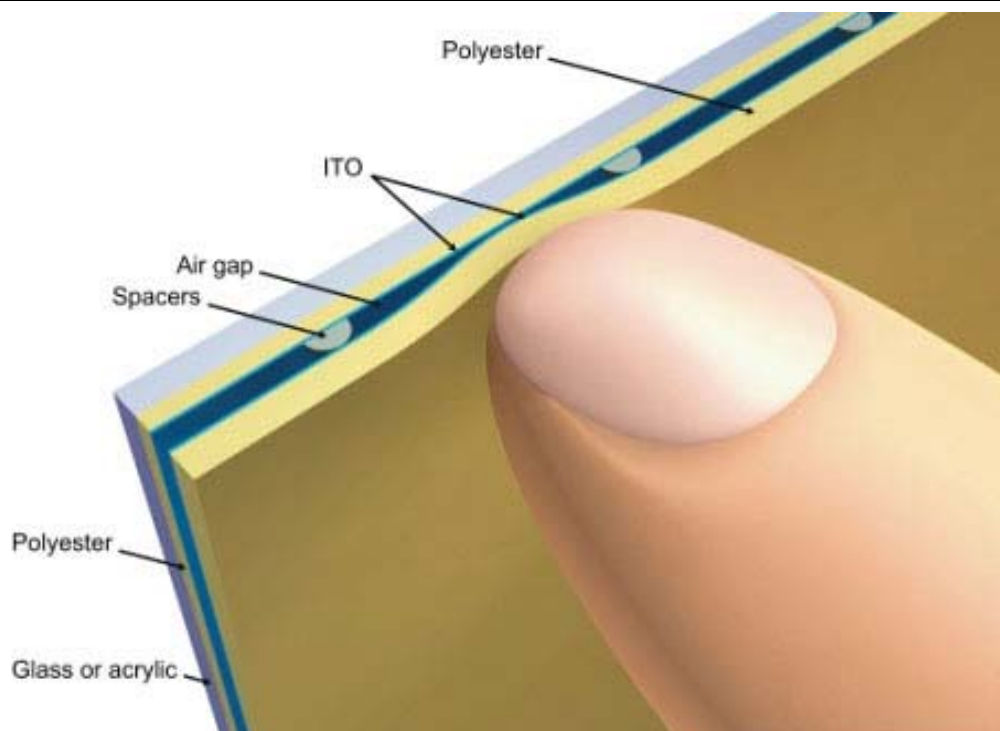
## 5. iphone 触摸屏和 5800 触摸屏工作原理

首先我们要知道 5800 和 iPhone 使用的是两种不同的工作原理。

5800 采用的是电阻式触摸屏，利用压力感应进行控制的，而 iPhone 采用的电容式触摸屏，通过人体的感应电流来工作。

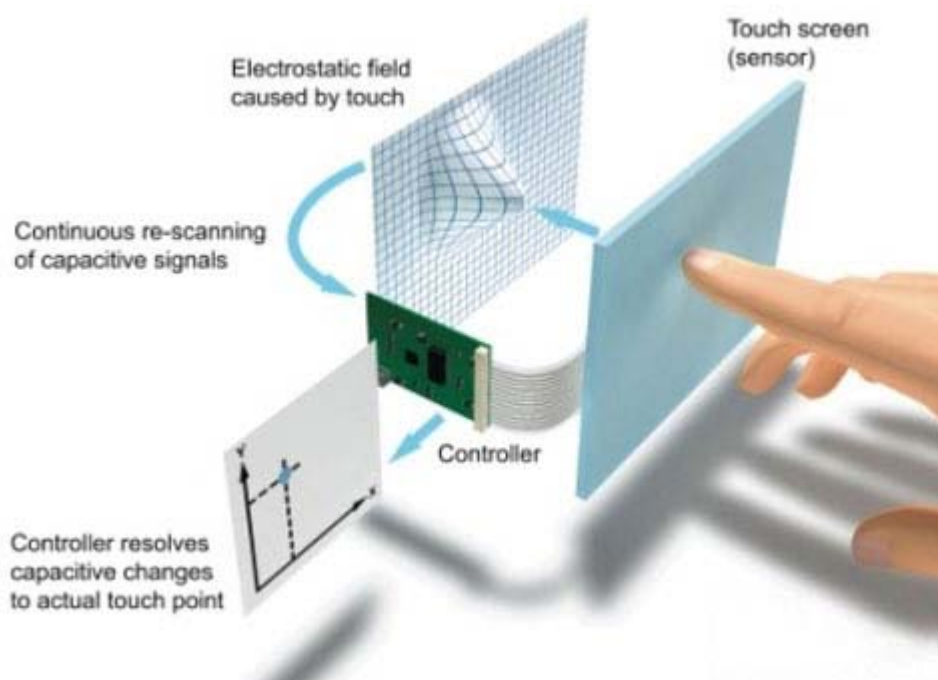
电阻式触摸屏的构成是显示屏及一块与显示屏紧密贴合的电阻薄膜屏。这个电阻薄膜屏通常分为两层，一层是由玻璃或有机玻璃构成的基层，其表面涂有透明的导电层；基层外面压着我们平时直接接触的经过硬化及防刮处理的塑料层，塑料层内部同样有一层导电层，两个导电层之间是分离的。当我们用手指或其他物体触摸屏幕的时候，两个导电层发生接触，电阻产生变化，控制器则根据电阻的具体变化来判断接触点的坐标并进行相应的操作。





而 iPhone 则采用的是电容式触摸屏，它是通过人体的感应电流来进行工作的。

普通电容式触摸屏的感应屏是一块四层复合玻璃屏，玻璃屏的内表面和夹层各涂有一层导电层，最外层是一薄层矽土玻璃保护层。当我们用手指触摸在感应屏上的时候，人体的电场让手指和和触摸屏表面形成一个耦合电容，对于高频电流来说，电容是直接导体，于是手指从接触点吸走一个很小的电流。这个电流分从触摸屏的四角上的电极中流出，并且流经这四个电极的电流与手指到四角的距离成正比，控制器通过对这四个电流比例的精确计算，得出触摸点的位置。



电容式触摸屏与传统的电阻式触摸屏有很大区别。电阻式触控屏幕在工作时每次只能判断一个触控点，如果触控点在两个以上，就不能做出正确的判断了，所以电阻式触摸屏仅适用于点击、拖拽等一些简单动作的判断。而电容式触摸屏的多点触控，则可以将用户的触摸分解为采集多点信号及判断信号意义两个工作，完成对复杂动作的判断。电容式触摸屏也有以下几个缺点：

- ✚ 精度不高。
- ✚ 易受环境影响。
- ✚ 成本偏高。

大多数情况下，这些系统都能正确探测到触摸的精确位置。但如果您试着同时触摸屏幕的好几个地方，结果就可能出错。有些屏幕只能对您第一次触摸到的地方作出反应。还有些屏幕可以同时探测到好几处触点，但软件无法计算出每次触摸的精确位置。其原因如下：

- ✚ 很多系统沿着轴线或者某个特定的方向探测变化，而不是探测屏幕的每个点。
- ✚ 有些触摸屏用系统内触点的平均值来探测触摸位置。
- ✚ 有些系统在测量时首先建立一道基线，当您触摸屏幕时，您的触摸产生了一道新的基线。所以如果同时触摸多处就会导致系统使用错误的基线作为起点进行测量。

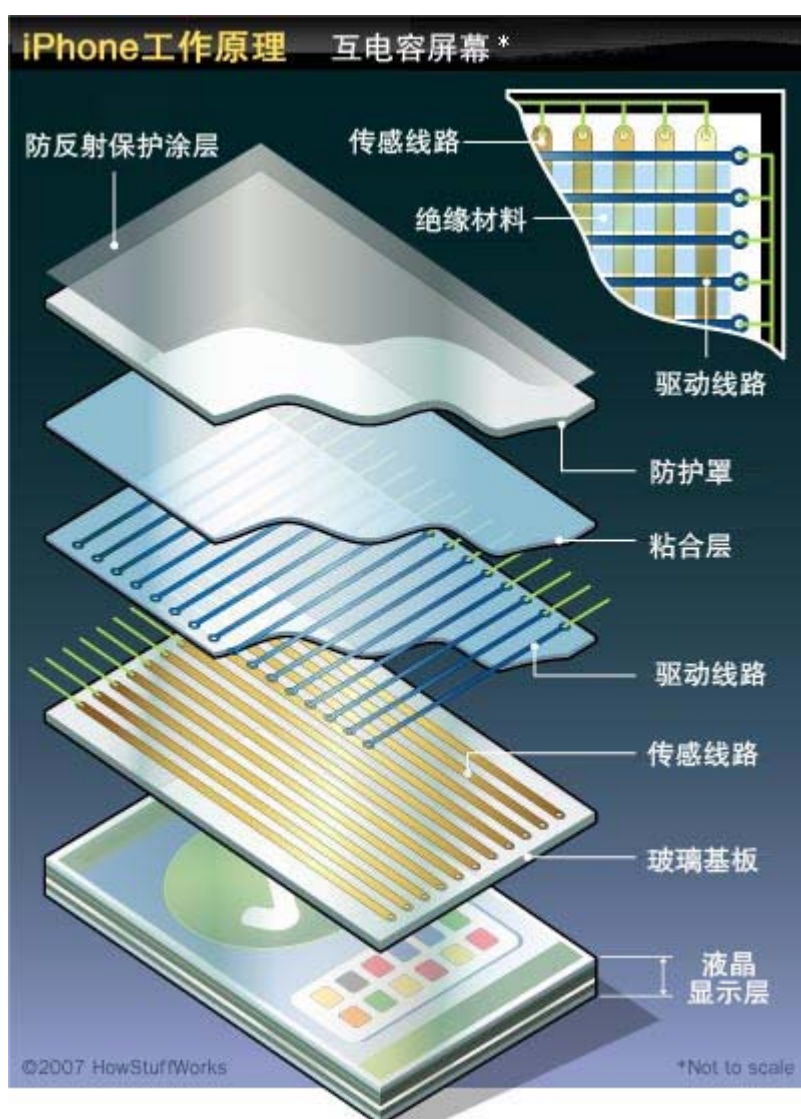
以下引用 Discovery 旗下网站原创文章对 iPhone 触摸原理的解析，我们通过



对比发现目前通过软件 5800 不可能达到 iPhone 多点触控的高度。

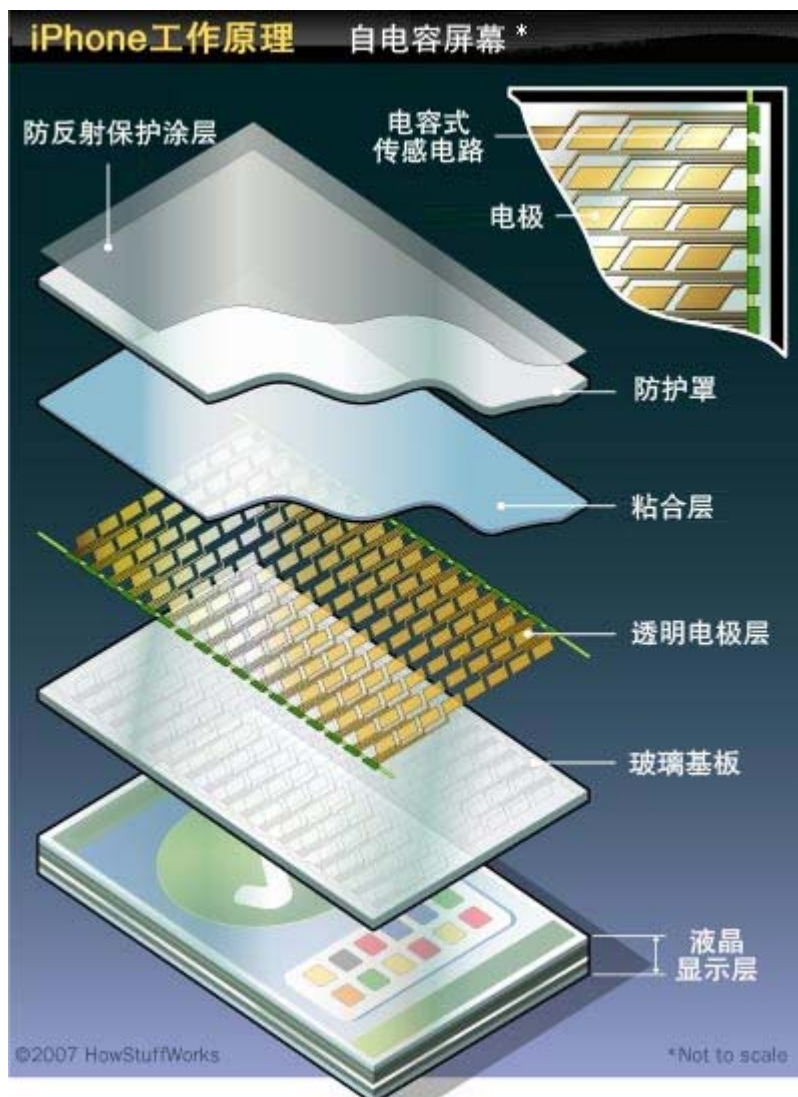
为了能让用户输入多触点的命令，iPhone 对已有技术做出了全新改进。和其它很多触摸屏一样，它的触摸屏含有一层电容材料。但是 iPhone 的电容器是根据一个坐标系来设计的。电容器的电路能够感应到沿线各点所发生的变化。也就是说，所有的点在被触摸时都能生成自己的信号，然后将信号传送给 iPhone 的处理器。这使得 iPhone 能够确定在多个点同时发生触摸的位置和运动方向。由于 iPhone 是依靠电容材料来工作的，因此您必须用手指去触摸它，用触控笔或者带着手套去触摸它都是无法操作的。

✓ 互耦合电容式触摸屏包括了一排的驱动线和一排的检测线



互电容触摸屏包含一个由传感线路和驱动线路组成的坐标系，以确定用户触摸了什么地方。

✓ 自耦合电容式传感器检测到包括检测电路和电极



自电容屏幕包含传感电路和电极，以确定用户触摸了什么地方。

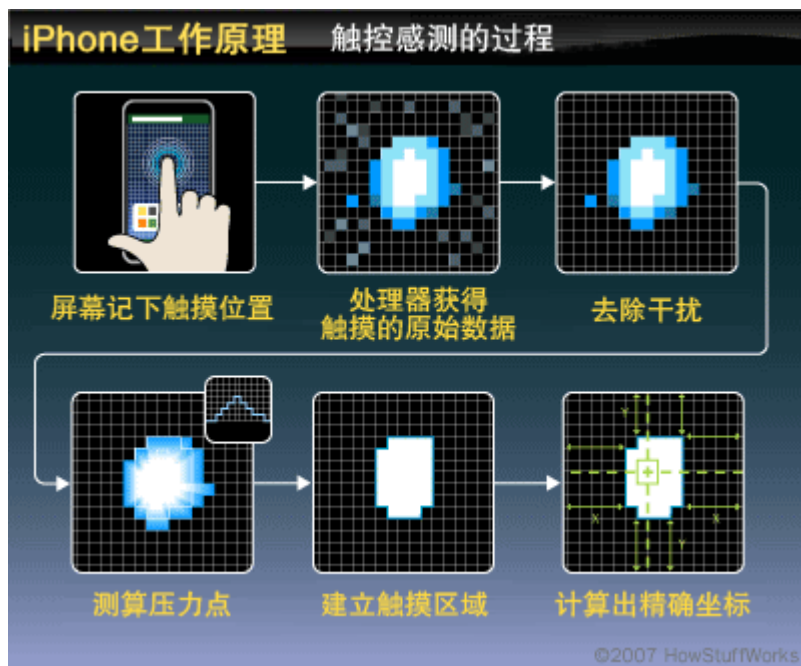
iPhone 的触摸屏使用互电容或自电容来探测触摸位置。互电容中，电容电路需要两层不同的材料，一层含有携带电流的驱动线路，一层含有传感线路，用于探测在节点的电流。自电容使用一层单独的电极，与电容感应电路相连。这两种方法都可以将触摸数据发送成电脉冲。

✓ iPhone 处理器

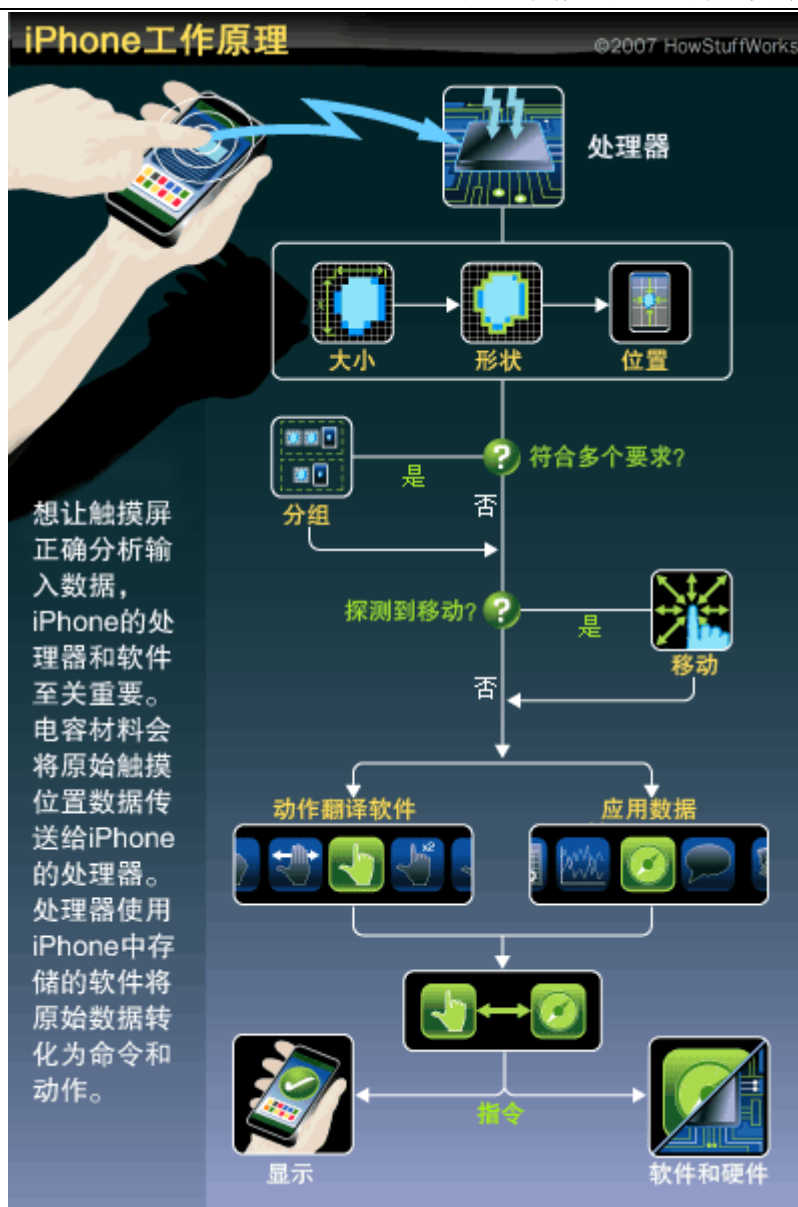
想让触摸屏正确分析输入数据，iPhone 的处理器和软件至关重要。电容材料会将原始触摸位置数据传送给 iPhone 的处理器。处理器使用 iPhone 内存储的软

件将原始数据转化为命令和动作。下面是转换过程：

- ✚ 信号以电脉冲的形式从触摸屏传送到处理器。
- ✚ 处理器使用软件分析数据，确定每次触摸是为了使用什么功能。这一过程包含确定屏幕上被触摸的区域大小、形状和位置。如果有必要，处理器会将相似的触摸整理分组。如果用户移动手指，处理器就会计算用户触摸的起点和终点间的差别。



- ✚ 处理器使用动作转换软件来确定用户的动作指令。它将用户的手指运动与用户在使用哪种应用程序的信息、用户触摸屏幕时应用程序在做什么联系起来。
- ✚ 处理器将用户的指令传送给使用中的程序。如果有必要的话，它还会将命令发送给 iPhone 的屏幕和其它硬件。如果原始数据与任何有用的动作或命令都不相符的话，iPhone 会认为这是一次无效触摸。



所有上述步骤都是在瞬间发生的——您在触摸屏上输入后，马上就能看见屏幕发生了变化。这样，您只需要用手指发出指令，就可以打开并使用 iPhone 的所有应用程序。

## 第 8 章 多进程编程

---

课程内容：

- ✧ 引言
- ✧ 环境变量
- ✧ 进程控制
- ✧ 进程间通讯














## 第 8 章 多进程编程

### 1、引言

我们知道，每个进程在内核中都有一个进程控制块（PCB）来维护进程相关的信息，Linux 内核的进程控制块是 `task_struct` 结构体。现在我们全面了解一下其中都有哪些信息。

进程 id。系统中每个进程有唯一的 id，在 C 语言中用 `pid_t` 类型表示，其实就是一个非负整数。

-  进程的状态，有运行、挂起、停止、僵尸等状态。
-  进程切换时需要保存和恢复的一些 CPU 寄存器。
-  描述虚拟地址空间的信息。
-  描述控制终端的信息。
-  当前工作目录（Current Working Directory）。
-  `umask` 掩码。
-  文件描述符表，包含很多指向 `file` 结构体的指针。
-  和信号相关的信息。
-  用户 id 和组 id。
-  控制终端、Session 和进程组。
-  进程可以使用的资源上限（Resource Limit）。

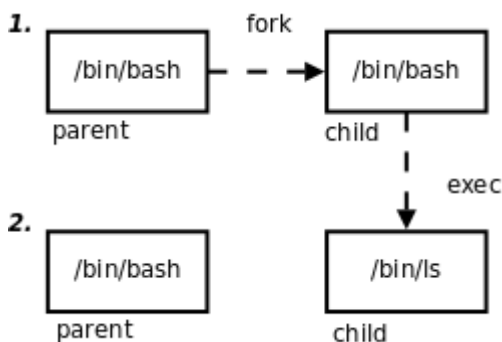
目前读者并不需要理解这些信息的细节，在随后几章中讲到某一项时会再次提醒读者它是保存在 PCB 中的。

`fork` 和 `exec` 是本章要介绍的两个重要的系统调用。`fork` 的作用是根据一个现有的进程复制出一个新进程，原来的进程称为父进程（Parent Process），新进程称为子进程（Child Process）。系统中同时运行着很多进程，这些进程都是从最初只有一个进程开始一个一个复制出来的。在 Shell 下输入命令可以运行一个程序，是因为 Shell 进程在读取用户输入的命令之后会调用 `fork` 复制出一个新的 Shell 进程，然后新的 Shell 进程调用 `exec` 执行新的程序。

我们知道一个程序可以多次加载到内存，成为同时运行的多个进程，例如可

以同时开多个终端窗口运行/bin/bash，另一方面，一个进程在调用 exec 前后也可以分别执行两个不同的程序，例如在 Shell 提示符下输入命令 ls，首先 fork 创建子进程，这时子进程仍在执行/bin/bash 程序，然后子进程调用 exec 执行新的程序/bin/ls，如下图所示。

图 8.1. fork/exec



在第 3 节“open/close”中我们做过一个实验：用 umask 命令设置 Shell 进程的 umask 掩码，然后运行程序 a.out，结果 a.out 进程的 umask 掩码也和 Shell 进程一样。现在可以解释了，因为 a.out 进程是 Shell 进程的子进程，子进程的 PCB 是根据父进程复制而来的，所以其中的 umask 掩码也和父进程一样。同样道理，子进程的当前工作目录也和父进程一样，所以我们可以用 cd 命令改变 Shell 进程的当前目录，然后用 ls 命令列出那个目录下的文件，ls 进程其实是在列自己的当前目录，而不是 Shell 进程的当前目录，只不过 ls 进程的当前目录正好和 Shell 进程相同。有一个例外，子进程 PCB 中的进程 id 和父进程是不同的。

## 2、环境变量

先前讲过，exec 系统调用执行新程序时会把命令行参数和环境变量表传递给 main 函数，它们在整个进程地址空间中的位置如下图所示。



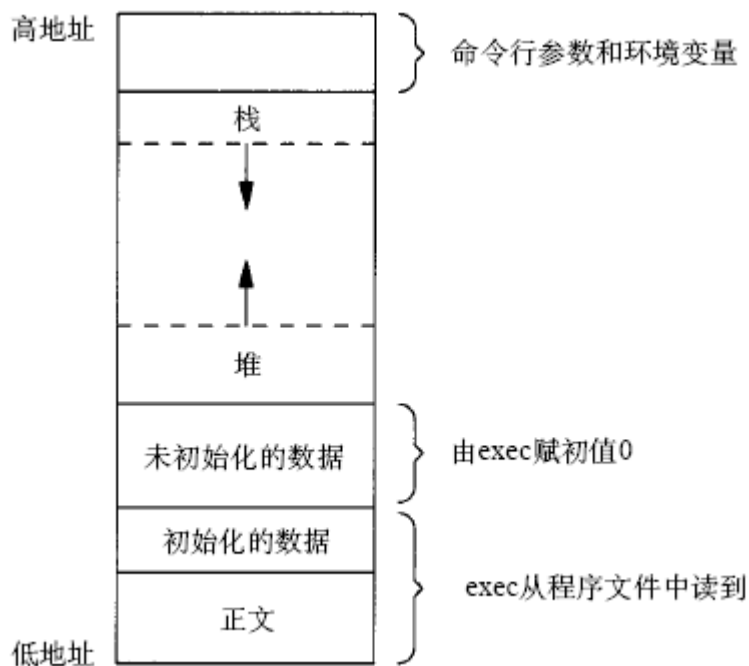


图 8.2. 进程地址空间

和命令行参数 `argv` 类似，环境变量表也是一组字符串，如下图所示。

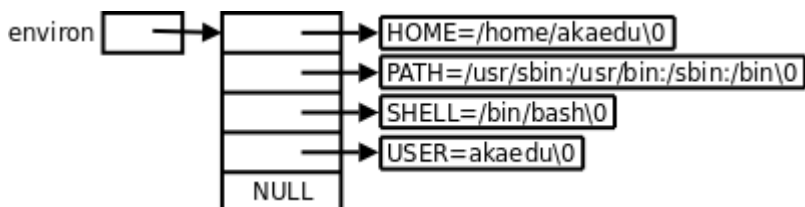


图 8.3. 环境变量

`libc` 中定义的全局变量 `environ` 指向环境变量表，`environ` 没有包含在任何头文件中，所以在使用时要用 `extern` 声明。例如：

#### 例 8.1. 打印环境变量

```
#include <stdio.h>

int main(void)
{
    extern char **environ;
    int i;
    for(i=0; environ[i]!=NULL; i++)
```

```
printf("%s\n", environ[i]);  
return 0;  
}
```

执行结果为

```
$ ./a.out  
SSH_AGENT_PID=5717  
SHELL=/bin/bash  
DESKTOP_STARTUP_ID=  
TERM=xterm  
...
```

由于父进程在调用 `fork` 创建子进程时会把自己的环境变量表也复制给子进程，所以 `a.out` 打印的环境变量和 `Shell` 进程的环境变量是相同的。

按照惯例，环境变量字符串都是“`key=value`”这样的形式，大多数 `name` 由大写字母加下划线组成，一般把 `name` 的部分叫做环境变量，`value` 的部分则是环境变量的值。环境变量定义了进程的运行环境，一些比较重要的环境变量的含义如下：

### ✓ PATH

可执行文件的搜索路径。`ls` 命令也是一个程序，执行它不需要提供完整的路径名 `bin/ls`，然而通常我们执行当前目录下的程序 `a.out` 却需要提供完整的路径名 `./a.out`，这是因为 `PATH` 环境变量的值里面包含了 `ls` 命令所在的目录 `bin`，却不包含 `a.out` 所在的目录。`PATH` 环境变量的值可以包含多个目录，用冒号隔开。在 `Shell` 中用 `echo` 命令可以查看这个环境变量的值：

```
$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

## ✓ SHELL

当前 Shell，它的值通常是/bin/bash。

## ✓ TERM

当前终端类型，在图形界面终端下它的值通常是 **xterm**，终端类型决定了一些程序的输出显示方式，比如图形界面终端可以显示汉字，而字符终端一般不行。

## ✓ LANG

语言和 locale，决定了字符编码以及时间、货币等信息的显示格式。

## ✓ HOME

当前用户主目录的路径，很多程序需要在主目录下保存配置文件，使得每个用户在运行该程序时都有自己的一套配置。

用 **environ** 指针可以查看所有环境变量字符串，但是不够方便，如果给出 **name** 要在环境变量表中查找它对应的 **value**，可以用 **getenv** 函数。

```
#include <stdlib.h>
char *getenv(const char *name);

getenv 的返回值是指向 value 的指针，若未找到则为 NULL。
```

修改环境变量可以用以下函数

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);
```

putenv 和 setenv 函数若成功则返回为 0，若出错则返回非 0。

setenv 将环境变量 name 的值设置为 value。如果已存在环境变量 name，那么

若 rewrite 非 0，则覆盖原来的定义；

若 rewrite 为 0，则不覆盖原来的定义，也不返回错误。

unsetenv 删除 name 的定义。即使 name 没有定义也不返回错误。

### 例 8.2. 修改环境变量

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    printf("PATH=%s\n", getenv("PATH"));
    setenv("PATH", "hello", 1);
    printf("PATH=%s\n", getenv("PATH"));
    return 0;
}
```

```
$ ./a.out
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PATH=hello
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

可以看出，Shell 进程的环境变量 PATH 传给了 a.out，然后 a.out 修改了 PATH 的值，在 a.out 中能打印出修改后的值，但在 Shell 进程中 PATH 的值没变。父进程在创建子进程时会复制一份环境变量给予进程，但此后二者的环境变量互不影响。

## 3、进程控制

### 3.1. fork 函数

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

fork 调用失败则返回-1，调用成功的返回值见下面的解释。我们通过一个例子来理解 fork 是怎样创建新进程的。

#### 例 30.3. fork

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
```

```
sleep(1);
}
return 0;
}
```

```
$ ./a.out
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

这个程序的运行过程如下图所示。

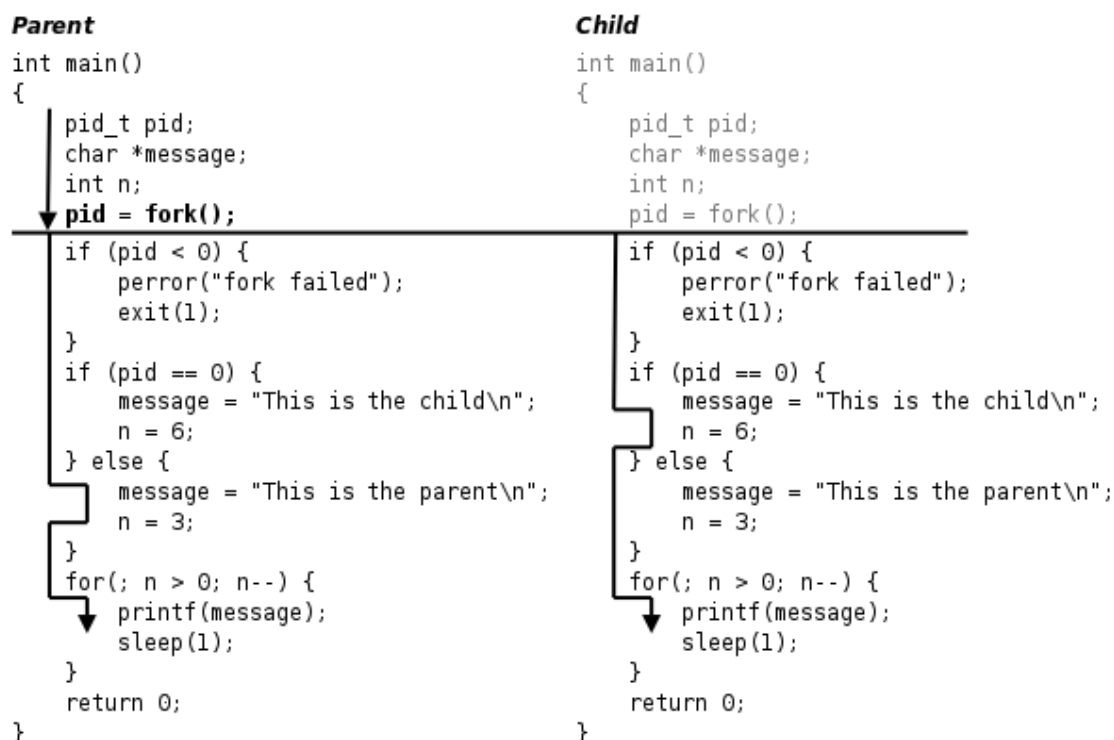


图 30.4. fork

1. 父进程初始化。
2. 父进程调用 fork，这是一个系统调用，因此进入内核。

3. 内核根据父进程复制出一个子进程，父进程和子进程的 PCB 信息相同，用户态代码和数据也相同。因此，子进程现在的状态看起来和父进程一样，做完了初始化，刚调用了 `fork` 进入内核，还没有从内核返回。

4. 现在有两个一模一样的进程看起来都调用了 `fork` 进入内核等待从内核返回（实际上 `fork` 只调用了一次），此外系统中还有很多别的进程也等待从内核返回。是父进程先返回还是子进程先返回，还是这两个进程都等待，先去调度执行别的进程，这都不一定，取决于内核的调度算法。

5. 如果某个时刻父进程被调度执行了，从内核返回后就从 `fork` 函数返回，保存在变量 `pid` 中的返回值是子进程的 `id`，是一个大于 0 的整数，因此执下面的 `else` 分支，然后执行 `for` 循环，打印 "This is the parent\n" 三次之后终止。

6. 如果某个时刻子进程被调度执行了，从内核返回后就从 `fork` 函数返回，保存在变量 `pid` 中的返回值是 0，因此执行下面的 `if (pid == 0)` 分支，然后执行 `for` 循环，打印 "This is the child\n" 六次之后终止。`fork` 调用把父进程的数据复制一份给子进程，但此后二者互不影响，在这个例子中，`fork` 调用之后父进程和子进程的变量 `message` 和 `n` 被赋予不同的值，互不影响。

7. 父进程每打印一条消息就睡眠 1 秒，这时内核调度别的进程执行，在 1 秒这么长的间隙里（对于计算机来说 1 秒很长了）子进程很有可能被调度到。同样地，子进程每打印一条消息就睡眠 1 秒，在这 1 秒期间父进程也有可能被调度到。所以程序运行的结果基本上是父子进程交替打印，但这也不是一定的，取决于系统中其它进程的运行情况和内核的调度算法，如果系统中其它进程非常繁忙则有可能观察到不同的结果。另外，读者也可以把 `sleep(1)` 去掉看程序的运行结果如何。

8. 这个程序是在 Shell 下运行的，因此 Shell 进程是父进程的父进程。父进程运行时 Shell 进程处于等待状态（第 3.3 节“`wait` 和 `waitpid` 函数”会讲到这种等待是怎么实现的），当父进程终止时 Shell 进程认为命令执行结束了，于是打印 Shell 提示符，而事实上子进程这时还没结束，所以子进程的消息打印到了 Shell 提示符后面。最后光标停在 `This is the child` 的下一行，这时用户仍然可以敲命令，即使命令不是紧跟在提示符后面，Shell 也能正确读取。

`fork` 函数的特点概括起来就是“调用一次，返回两次”，在父进程中调用一次，在父进程和子进程中各返回一次。从上图可以看出，一开始是一个控制流程，调用 `fork` 之后发生了分叉，变成两个控制流程，这也就是“`fork`”（分叉）这个名字的由来了。子进程中 `fork` 的返回值是 0，而父进程中 `fork` 的返回值则是子进程的 `id`（从根本上说 `fork` 是从内核返回的，内核自有办法让父进程和子进程返回不同的值），这样当 `fork` 函数返回后，程序员可以根据返回值的不同让父进程和



子进程执行不同的代码。

fork 的返回值这样规定是有道理的。fork 在子进程中返回 0，子进程仍可以调用 getpid 函数得到自己的进程 id，也可以调用 getppid 函数得到父进程的 id。在父进程中用 getpid 可以得到自己的进程 id，然而要想得到子进程的 id，只有将 fork 的返回值记录下来，别无它法。

fork 的另一个特性是所有由父进程打开的描述符都被复制到子进程中。父、子进程中相同编号的文件描述符在内核中指向同一个 file 结构体，也就是说，file 结构体的引用计数要增加。

用 gdb 调试多进程的程序会遇到困难，gdb 只能跟踪一个进程（默认是跟踪父进程），而不能同时跟踪多个进程，但可以设置 gdb 在 fork 之后跟踪父进程还是子进程。以上面的程序为例：

```
$ gcc main.c -g
$ gdb a.out
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) l
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void)
7  {
8      pid_t pid;
9      char *message;
10     int n;
11     pid = fork();
(gdb)
12     if(pid<0) {
13         perror("fork failed");
14         exit(1);
```

```

15     }
16     if(pid==0) {
17         message = "This is the child\n";
18         n = 6;
19     } else {
20         message = "This is the parent\n";
21         n = 3;
(gdb) b 17
Breakpoint 1 at 0x8048481: file main.c, line 17.
(gdb) set follow-fork-mode child
(gdb) r
Starting program: /home/akaedu/a.out
This is the parent
[Switching to process 30725]

Breakpoint 1, main () at main.c:17
17         message = "This is the child\n";
(gdb) This is the parent
This is the parent
  
```

set follow-fork-mode child 命令设置 gdb 在 fork 之后跟踪子进程（set follow-fork-mode parent 则是跟踪父进程），然后用 run 命令，看到的现象是父进程一直在运行，在(gdb)提示符下打印消息，而子进程被先前设的断点打断了。

### 3.2. exec 函数

用 fork 创建子进程后执行的是和父进程相同的程序（但有可能执行不同的代码分支），子进程往往要调用一种 exec 函数以执行另一个程序。当进程调用一种 exec 函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行。调用 exec 并不创建新进程，所以调用 exec 前后该进程的 id 并未改变。

其实有六种以 exec 开头的函数，统称 exec 函数：

```

#include <unistd.h>

int execl(const char *path, const char *arg, ...);
  
```

```
int execlp(const char *file, const char *arg, ...);
int execl(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

这些函数如果调用成功则加载新的程序从启动代码开始执行，不再返回，如果调用出错则返回-1，所以 `exec` 函数只有出错的返回值而没有成功的返回值。

这些函数原型看起来很容易混，但只要掌握了规律就很好记。不带字母 `p` (表示 `path`) 的 `exec` 函数第一个参数必须是程序的相对路径或绝对路径，例如 `"/bin/ls"` 或 `"/a.out"`，而不能是 `"ls"` 或 `"a.out"`。对于带字母 `p` 的函数：

如果参数中包含 `/`，则将其视为路径名。

否则视为不带路径的程序名，在 `PATH` 环境变量的目录列表中搜索这个程序。

带有字母 `l` (表示 `list`) 的 `exec` 函数要求将新程序的每个命令行参数都当作一个参数传给它，命令行参数的个数是可变的，因此函数原型中有 `...`，`...` 中的最后一个可变参数应该是 `NULL`，起 `sentinel` 的作用。对于带有字母 `v` (表示 `vector`) 的函数，则应该先构造一个指向各参数的指针数组，然后将该数组的首地址当作参数传给它，数组中的最后一个指针也应该是 `NULL`，就像 `main` 函数的 `argv` 参数或者环境变量表一样。

对于以 `e` (表示 `environment`) 结尾的 `exec` 函数，可以把一份新的环境变量表传给它，其他 `exec` 函数仍使用当前的环境变量表执行新程序。

`exec` 调用举例如下：

```
char *const ps_argv[]={"ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL};
char *const ps_envp[]={"PATH=/bin:/usr/bin", "TERM=console", NULL};
execl("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
execv("/bin/ps", ps_argv);
execl("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL, ps_envp);
execve("/bin/ps", ps_argv, ps_envp);
execlp("ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
execvp("ps", ps_argv);
```

事实上，只有 `execve` 是真正的系统调用，其它五个函数最终都调用 `execve`，所以 `execve` 在 man 手册第 2 节，其它函数在 man 手册第 3 节。这些函数之间的关系如下图所示。

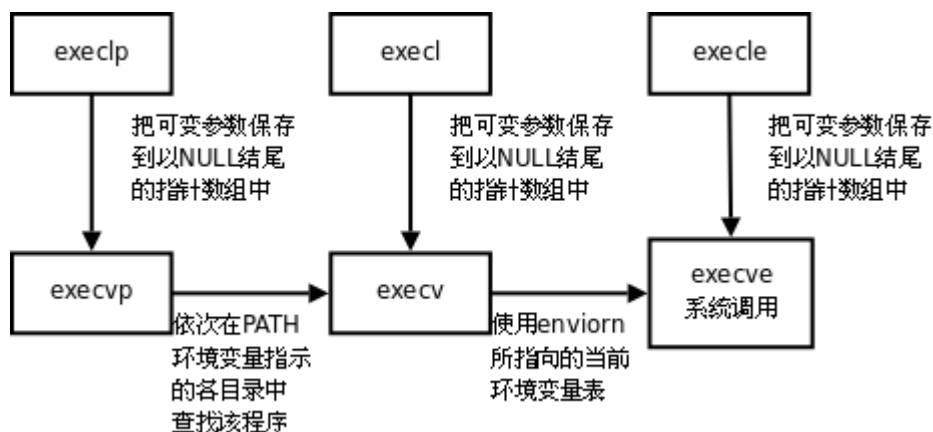


图 8.5. exec 函数族

一个完整的例子：

```

#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    execlp("ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
    perror("exec ps");
    exit(1);
}

```

执行此程序则得到：

```

$ ./a.out
  PID  PPID  PGRP  SESS TPGID COMMAND
 6614  6608  6614  6614  7199  bash
 7199  6614  7199  6614  7199  ps

```

由于 `exec` 函数只有错误返回值，只要返回了一一定是出错了，所以不需要判断它的返回值，直接在后面调用 `perror` 即可。注意在调用 `execlp` 时传了两个"ps"参数，第一个"ps"是程序名，`execlp` 函数要在 `PATH` 环境变量中找到这个程序并执行它，而第二个"ps"是第一个命令行参数，`execlp` 函数并不关心它的值，只是简单地把它传给 `ps` 程序，`ps` 程序可以通过 `main` 函数的 `argv[0]` 取到这个参数。

调用 `exec` 后，原来打开的文件描述符仍然是打开的[47]。利用这一点可以实现 I/O 重定向。先看一个简单的例子，把标准输入转成大写然后打印到标准输出：

#### 例 8.4. upper

```
/* upper.c */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    return 0;
}
```

运行结果如下：

```
$ ./upper
hello THERE
HELLO THERE
（按 Ctrl-D 表示 EOF）
$

使用 Shell 重定向：

$ cat file.txt
```

```
this is the file, file.txt, it is all lower case.  
$ ./upper < file.txt  
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

如果希望把待转换的文件名放在命令行参数中，而不是借助于输入重定向，我们可以利用 `upper` 程序的现有功能，再写一个包装程序 `wrapper`。

### 例 30.5. wrapper

```
/* wrapper.c */  
#include <unistd.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include <fcntl.h>  
  
int main(int argc, char *argv[])  
{  
    int fd;  
    if (argc != 2) {  
        fputs("usage: wrapper file\n", stderr);  
        exit(1);  
    }  
    fd = open(argv[1], O_RDONLY);  
    if (fd < 0) {  
        perror("open");  
        exit(1);  
    }  
    dup2(fd, STDIN_FILENO);  
    close(fd);  
    execl("./upper", "upper", NULL);  
    perror("exec ./upper");  
    exit(1);  
}
```

`wrapper` 程序将命令行参数当作文件名打开，将标准输入重定向到这个文件，然后调用 `exec` 执行 `upper` 程序，这时原来打开的文件描述符仍然是打开的，`upper` 程序只负责从标准输入读入字符转成大写，并不关心标准输入对应的是文件还是终端。运行结果如下：

```
$ ./wrapper file.txt  
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

### ✓ 3.3. wait 和 waitpid 函数

一个进程在终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的 PCB 还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用 `wait` 或 `waitpid` 获取这些信息，然后彻底清除掉这个进程。我们知道一个进程的退出状态可以在 Shell 中用特殊变量 `$?` 查看，因为 Shell 是它的父进程，当它终止时 Shell 调用 `wait` 或 `waitpid` 得到它的退出状态同时彻底清除掉这个进程。

如果一个进程已经终止，但是它的父进程尚未调用 `wait` 或 `waitpid` 对它进行清理，这时的进程状态称为僵尸（Zombie）进程。任何进程在刚终止时都是僵尸进程，正常情况下，僵尸进程都立刻被父进程清理了，为了观察到僵尸进程，我们自己写一个不正常的程序，父进程 `fork` 出子进程，子进程终止，而父进程既不终止也不调用 `wait` 清理子进程：

```
#include <unistd.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    pid_t pid=fork();  
    if(pid<0) {  
        perror("fork");  
        exit(1);  
    }  
    if(pid>0) { /* parent */  
        while(1);  
    }  
    /* child */  
    return 0;  
}
```



在后台运行这个程序，然后用 ps 命令查看：

```

$ ./a.out &
[1] 6130
$ ps u
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
akaedu       6016  0.0  0.3   5724  3140 pts/0    Ss   08:41   0:00 bash
akaedu       6130 97.2  0.0   1536   284 pts/0    R    08:44  14:33 ./a.out
akaedu       6131  0.0  0.0      0     0 pts/0    Z    08:44   0:00 [a.out]
<defunct>
akaedu       6163  0.0  0.0   2620  1000 pts/0    R+   08:59   0:00 ps u
  
```

在 ./a.out 命令后面加个 & 表示后台运行，Shell 不等待这个进程终止就立刻打印提示符并等待用户输入命令。现在 Shell 是位于前台的，用户在终端的输入会被 Shell 读取，后台进程是读不到终端输入的。第二条命令 ps u 是在前台运行的，在此期间 Shell 进程和 ./a.out 进程都在后台运行，等到 ps u 命令结束时 Shell 进程又重新回到前台。在第 33 章 信号和第 34 章 终端、作业控制与守护进程将会进一步解释前台（Foreground）和后台（Background）的概念。

父进程的 pid 是 6130，子进程是僵尸进程，pid 是 6131，ps 命令显示僵尸进程的状态为 Z，在命令行一栏还显示 <defunct>。

如果一个父进程终止，而它的子进程还存在（这些子进程或者仍在运行，或者已经是僵尸进程了），则这些子进程的父进程改为 init 进程。init 是系统中的一个特殊进程，通常程序文件是 /sbin/init，进程 id 是 1，在系统启动时负责启动各种系统服务，之后就负责清理子进程，只要有子进程终止，init 就会调用 wait 函数清理它。




僵尸进程是不能用 kill 命令清除掉的，因为 kill 命令只是用来终止进程的，而僵尸进程已经终止了。思考一下，用什么办法可以清除掉僵尸进程？

wait 和 waitpid 函数的原型是：



```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

若调用成功则返回清理掉的子进程 `id`，若调用出错则返回-1。父进程调用 `wait` 或 `waitpid` 时可能会：

-  阻塞（如果它的所有子进程都还在运行）。
-  带子进程的终止信息立即返回（如果一个子进程已终止，正等待父进程读取其终止信息）。
-  出错立即返回（如果它没有任何子进程）。

这两个函数的区别是：

-  如果父进程的所有子进程都还在运行，调用 `wait` 将使父进程阻塞，而调用 `waitpid` 时如果在 `options` 参数中指定 `WNOHANG` 可以使父进程不阻塞而立即返回 0。
-  `wait` 等待第一个终止的子进程，而 `waitpid` 可以通过 `pid` 参数指定等待哪一个子进程。

可见，调用 `wait` 和 `waitpid` 不仅可以获得子进程的终止信息，还可以使父进程阻塞等待子进程终止，起到进程间同步的作用。如果参数 `status` 不是空指针，则子进程的终止信息通过这个参数传出，如果只是为了同步而不关心子进程的终止信息，可以将 `status` 参数指定为 `NULL`。

### 例 30.6. waitpid

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t pid;
```

```
pid = fork();
if (pid < 0) {
    perror("fork failed");
    exit(1);
}
if (pid == 0) {
    int i;
    for (i = 3; i > 0; i--) {
        printf("This is the child\n");
        sleep(1);
    }
    exit(3);
} else {
    int stat_val;
    waitpid(pid, &stat_val, 0);
    if (WIFEXITED(stat_val))
        printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
    else if (WIFSIGNALED(stat_val))
        printf("Child terminated abnormally, signal %d\n", WTERMSIG(stat_val));
}
return 0;
}
```

子进程的终止信息在一个 `int` 中包含了多个字段，用宏定义可以取出其中的每个字段：如果子进程是正常终止的，`WIFEXITED` 取出的字段值非零，`WEXITSTATUS` 取出的字段值就是子进程的退出状态；如果子进程是收到信号而异常终止的，`WIFSIGNALED` 取出的字段值非零，`WTERMSIG` 取出的字段值就是信号的编号。作为练习，请读者从头文件里查一下这些宏做了什么运算，是如何取出字段值的。

### ✓ 习题

1、请读者修改例 8.6 “waitpid”的代码和实验条件，使它产生“Child terminated abnormally”的输出。

## 4、进程间通讯

每个进程各自有不同的用户地址空间，任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核，在内核中开辟一块缓冲

区，进程 1 把数据从用户空间拷到内核缓冲区，进程 2 再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信（IPC，InterProcess Communication）。如下图所示。

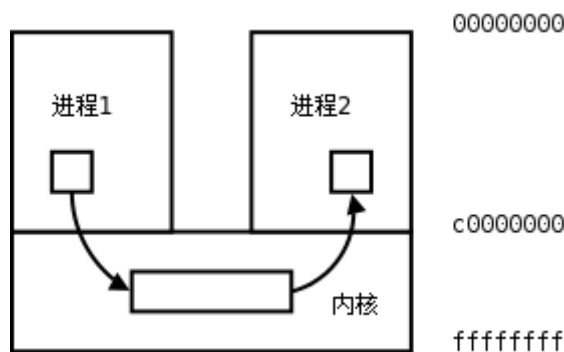


图 8.6. 进程间通信

## 4.1. 管道

管道是一种最基本的 IPC 机制，由 pipe 函数创建：

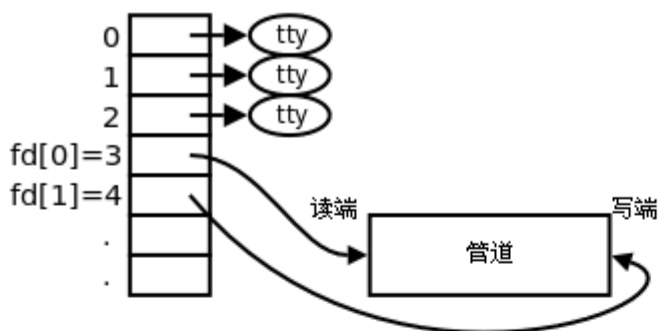
```
#include <unistd.h>

int pipe(int filedes[2]);
```

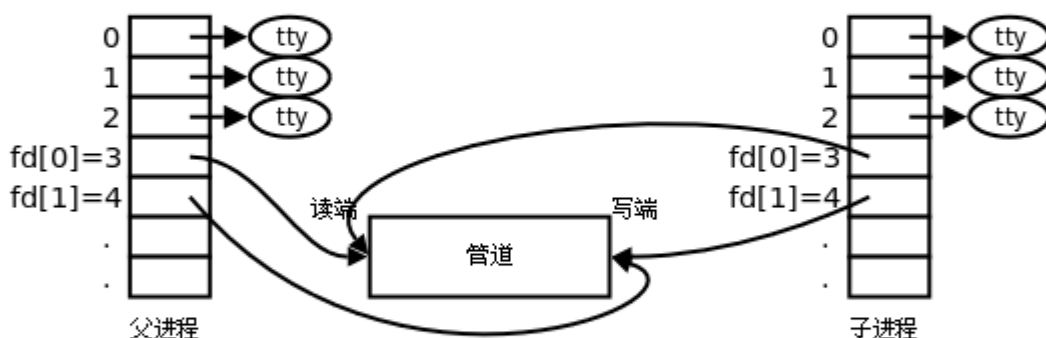
调用 pipe 函数时在内核中开辟一块缓冲区（称为管道）用于通信，它有一个读端一个写端，然后通过 filedes 参数传出给用户程序两个文件描述符，filedes[0] 指向管道的读端，filedes[1] 指向管道的写端（很好记，就像 0 是标准输入 1 是标准输出一样）。所以管道在用户程序看起来就像一个打开的文件，通过 read(filedes[0]); 或者 write(filedes[1]); 向这个文件读写数据其实是在读写内核缓冲区。pipe 函数调用成功返回 0，调用失败返回 -1。

开辟了管道之后如何实现两个进程间的通信呢？比如可以按下面的步骤通信。

### 1. 父进程创建管道



### 2. 父进程fork出子进程



### 3. 父进程关闭fd[0]，子进程关闭fd[1]

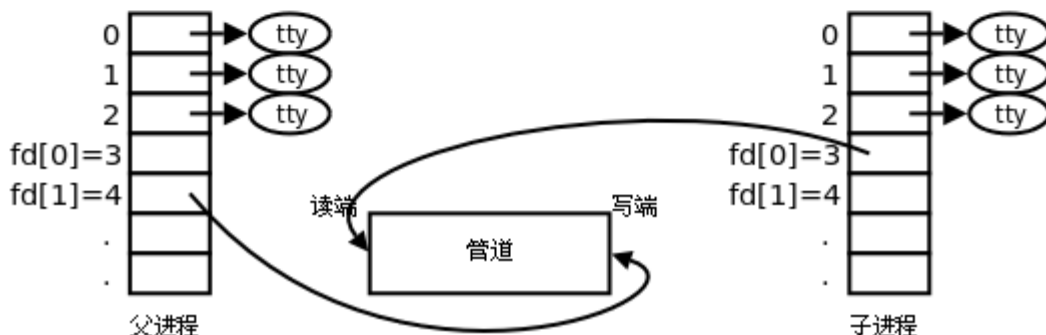


图 30.7. 管道

1. 父进程调用 `pipe` 开辟管道，得到两个文件描述符指向管道的两端。
2. 父进程调用 `fork` 创建子进程，那么子进程也有两个文件描述符指向同一管道。
3. 父进程关闭管道读端，子进程关闭管道写端。父进程可以往管道里写，子进程可以从管道里读，管道是用环形队列实现的，数据从写端流入从

读端流出，这样就实现了进程间通信。

### 例 30.7. 管道

```
#include <stdlib.h>
#include <unistd.h>
#define MAXLINE 80

int main(void)
{
    int n;
    int fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0) {
        perror("pipe");
        exit(1);
    }
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }
    if (pid > 0) { /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
        wait(NULL);
    } else { /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    return 0;
}
```

使用管道有一些限制：

两个进程通过一个管道只能实现单向通信，比如上面的例子，父进程写子进程读，如果有时候也需要子进程写父进程读，就必须另开一个管道。请读者思考，如果只开一个管道，但是父进程不关闭读端，子进程也不关闭写端，双方都有读端和写端，为什么不能实现双向通信？

管道的读写端通过打开的文件描述符来传递，因此要通信的两个进程必须从它们的公共祖先那里继承管道文件描述符。上面的例子是父进程把文件描述符传给子进程之后父子进程之间通信，也可以父进程 `fork` 两次，把文件描述符传给两个子进程，然后两个子进程之间通信，总之需要通过 `fork` 传递文件描述符使两个进程都能访问同一管道，它们才能通信。

使用管道需要注意以下 4 种特殊情况（假设都是阻塞 I/O 操作，没有设置 `O_NONBLOCK` 标志）：

1. 如果所有指向管道写端的文件描述符都关闭了（管道写端的引用计数等于 0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次 `read` 会返回 0，就像读到文件末尾一样。
2. 如果有指向管道写端的文件描述符没关闭（管道写端的引用计数大于 0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次 `read` 会阻塞，直到管道中有数据可读了才读取数据并返回。
3. 如果所有指向管道读端的文件描述符都关闭了（管道读端的引用计数等于 0），这时有进程向管道的写端 `write`，那么该进程会收到信号 `SIGPIPE`，通常会导致进程异常终止。在第 33 章 信号会讲到怎样使 `SIGPIPE` 信号不终止进程。
4. 如果有指向管道读端的文件描述符没关闭（管道读端的引用计数大于 0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次 `write` 会阻塞，直到管道中有空位置了才写入数据并返回。

管道的这四种特殊情况具有普遍意义。在第 37 章 `socket` 编程要讲的 `TCP socket` 也具有管道的这些特性。

## ✓ 习题

1、在例 8.7 “管道”中，父进程只用到写端，因而把读端关闭，子进程只用到读端，因而把写端关闭，然后互相通信，不使用的读端或写端必须关闭，请读者想一想如果不关闭会有什么问题。

2、请读者修改例 8.7 “管道”的代码和实验条件，验证我上面所说的四种特



殊情况。

## 4.2. 其它 IPC 机制

进程间通信必须通过内核提供的通道，而且必须有一种办法在进程中标识内核提供的某个通道，上一节讲的管道是用打开的文件描述符来标识的。如果要互相通信的几个进程没有从公共祖先那里继承文件描述符，它们怎么通信呢？内核提供一条通道不成问题，问题是如何标识这条通道才能使各进程都可以访问它？文件系统中的路径名是全局的，各进程都可以访问，因此可以用文件系统中的路径名来标识一个 IPC 通道。

FIFO 和 UNIX Domain Socket 这两种 IPC 机制都是利用文件系统中的特殊文件来标识的。可以用 `mkfifo` 命令创建一个 FIFO 文件：










```
$ mkfifo hello
$ ls -l hello
prw-r--r-- 1 akaedu akaedu 0 2008-10-30 10:44 hello
```

FIFO 文件在磁盘上没有数据块，仅用来标识内核中的一条通道，各进程可以打开这个文件进行 `read/write`，实际上是在读写内核通道（根本原因在于这个 `file` 结构体所指向的 `read`、`write` 函数和常规文件不一样），这样就实现了进程间通信。UNIX Domain Socket 和 FIFO 的原理类似，也需要一个特殊的 `socket` 文件来标识内核中的通道，例如 `/var/run` 目录下有很多系统服务的 `socket` 文件：

```
$ ls -l /var/run/
total 52
srw-rw-rw- 1 root      root      0 2008-10-30 00:24 acpid.socket
...
srw-rw-rw- 1 root      root      0 2008-10-30 00:25 gdm_socket
...
srw-rw-rw- 1 root      root      0 2008-10-30 00:24 sdp
...
srwxr-xr-x 1 root      root      0 2008-10-30 00:42 synaptic.socket
```

文件类型 `s` 表示 `socket`，这些文件在磁盘上也没有数据块。UNIX Domain Socket 是目前最广泛使用的 IPC 机制，到后面讲 `socket` 编程时再详细介绍。

现在把进程之间传递信息的各种途径（包括各种 IPC 机制）总结如下：

-  父进程通过 `fork` 可以将打开文件的描述符传递给子进程
-  子进程结束时，父进程调用 `wait` 可以得到子进程的终止信息
-  几个进程可以在文件系统中读写某个共享文件，也可以通过给文件加锁来实现进程间同步
-  进程之间互发信号，一般使用 `SIGUSR1` 和 `SIGUSR2` 实现用户自定义功能
-  管道
-  FIFO
-  `mmap` 函数，几个进程可以映射同一内存区
-  `SYS V IPC`，以前的 `SYS V UNIX` 系统实现的 IPC 机制，包括消息队列、信号量和共享内存，现在已经基本废弃
-  UNIX Domain Socket，目前最广泛使用的 IPC 机制

## 第 9 章 多线程编程

---

课程内容：





- ✧ 线程的概念
- ✧ 线程控制
- ✧ 线程间同步
- ✧ 编程练习

## 第 9 章 多线程编程







### 1. 线程的概念

我们知道，进程在各自独立的地址空间中运行，进程之间共享数据需要用 `mmap` 或者进程间通信机制，本节我们学习如何在一个进程的地址空间中执行多个线程。有些情况需要在一个进程中同时执行多个控制流程，这时候线程就派上了用场，比如实现一个图形界面的下载软件，一方面需要和用户交互，等待和处理用户的鼠标键盘事件，另一方面又需要同时下载多个文件，等待和处理从多个网络主机发来的数据，这些任务都需要一个“等待-处理”的循环，可以用多线程实现，一个线程专门负责与用户交互，另外几个线程每个线程负责和一个网络主机通信。

以前我们讲过，`main` 函数和信号处理函数是同一个进程地址空间中的多个控制流程，多线程也是如此，但是比信号处理函数更加灵活，信号处理函数的控制流程只是在信号递达时产生，在处理完信号之后就结束，而多线程的控制流程可以长期并存，操作系统会在各线程之间调度和切换，就像在多个进程之间调度和切换一样。由于同一进程的多个线程共享同一地址空间，因此 `Text Segment`、`Data Segment` 都是共享的，如果定义一个函数，在各线程中都可以调用，如果定义一个全局变量，在各线程中都可以访问到，除此之外，各线程还共享以下进程资源和环境：

-  文件描述符表
-  每种信号的处理方式（`SIG_IGN`、`SIG_DFL` 或者自定义的信号处理函数）
-  当前工作目录
-  用户 `id` 和组 `id`

但有些资源是每个线程各有一份的：

-  线程 `id`
-  上下文，包括各种寄存器的值、程序计数器和栈指针
-  栈空间
-  `errno` 变量
-  信号屏蔽字
-  调度优先级

我们将要学习的线程库函数是由 `POSIX` 标准定义的，称为 `POSIX thread` 或

者 pthread。在 Linux 上线程函数位于 libpthread 共享库中，因此在编译时要加上 -lpthread 选项。

## 2. 线程控制

### 2.1. 创建线程

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *restrict thread,  
const pthread_attr_t *restrict attr,  
void *(*start_routine)(void*), void *restrict arg);
```

返回值：成功返回 0，失败返回错误号。以前学过的系统函数都是成功返回 0，失败返回 -1，而错误号保存在全局变量 `errno` 中，而 `pthread` 库的函数都是通过返回值返回错误号，虽然每个线程也都有一个 `errno`，但这是为了兼容其它函数接口而提供的，`pthread` 库本身并不使用它，通过返回值返回错误码更加清晰。

在一个线程中调用 `pthread_create()` 创建新的线程后，当前线程从 `pthread_create()` 返回继续往下执行，而新的线程所执行的代码由我们传给 `pthread_create` 的函数指针 `start_routine` 决定。`start_routine` 函数接收一个参数，是通过 `pthread_create` 的 `arg` 参数传递给它的，该参数的类型为 `void *`，这个指针按什么类型解释由调用者自己定义。`start_routine` 的返回值类型也是 `void *`，这个指针的含义同样由调用者自己定义。`start_routine` 返回时，这个线程就退出了，其它线程可以调用 `pthread_join` 得到 `start_routine` 的返回值，类似于父进程调用 `wait(2)` 得到子进程的退出状态，稍后详细介绍 `pthread_join`。

`pthread_create` 成功返回后，新创建的线程的 `id` 被填写到 `thread` 参数所指向的内存单元。我们知道进程 `id` 的类型是 `pid_t`，每个进程的 `id` 在整个系统中是唯一的，调用 `getpid(2)` 可以获得当前进程的 `id`，是一个正整数值。线程 `id` 的类型是 `thread_t`，它只在当前进程中保证是唯一的，在不同的系统中 `thread_t` 这个类型有不同的实现，它可能是一个整数值，也可能是一个结构体，也可能是一个地址，所以不能简单地当成整数用 `printf` 打印，调用 `pthread_self(3)` 可以获得当前线

程的 id。

attr 参数表示线程属性, 本章不深入讨论线程属性, 所有代码例子都传 NULL 给 attr 参数, 表示线程属性取缺省值, 感兴趣的读者可以参考[APUE2e]。首先看一个简单的例子:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_t ntid;

void printids(const char *s)
{
    pid_t    pid;
    pthread_t tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
        (unsigned int)tid, (unsigned int)tid);
}

void *thr_fn(void *arg)
{
    printids(arg);
    return NULL;
}

int main(void)
{
    int err;

    err = pthread_create(&ntid, NULL, thr_fn, "new thread: ");
    if (err != 0) {
        fprintf(stderr, "can't create thread: %s\n", strerror(err));
        exit(1);
    }
    printids("main thread:");
    sleep(1);
}
```

```
return 0;  
}
```

编译运行结果如下：

```
$ gcc main.c -lpthread  
$ ./a.out  
main thread: pid 7398 tid 3084450496 (0xb7d8fac0)  
new thread:  pid 7398 tid 3084446608 (0xb7d8eb90)
```

可知在 Linux 上，`thread_t` 类型是一个地址值，属于同一进程的多个线程调用 `getpid(2)` 可以得到相同的进程号，而调用 `pthread_self(3)` 得到的线程号各不相同。


由于 `pthread_create` 的错误码不保存在 `errno` 中，因此不能直接用 `perror(3)` 打印错误信息，可以先用 `strerror(3)` 把错误码转换成错误信息再打印。

如果任意一个线程调用了 `exit` 或 `_exit`，则整个进程的所有线程都终止，由于从 `main` 函数 `return` 也相当于调用 `exit`，为了防止新创建的线程还没有得到执行就终止，我们在 `main` 函数 `return` 之前延时 1 秒，这只是一种权宜之计，即使主线程等待 1 秒，内核也不一定会调度新创建的线程执行，下一节我们会看到更好的办法。

思考题：主线程在一个全局变量 `ntid` 中保存了新创建的线程的 `id`，如果新创建的线程不调用 `pthread_self` 而是直接打印这个 `ntid`，能不能达到同样的效果？

## 2.2. 终止线程

如果需要只终止某个线程而不终止整个进程，可以有三种方法：

 从线程函数 `return`。这种方法对主线程不适用，从 `main` 函数 `return` 相当于调用 `exit`。

 一个线程可以调用 `pthread_cancel` 终止同一进程中的另一个线程。





线程可以调用 `pthread_exit` 终止自己。

用 `pthread_cancel` 终止一个线程分同步和异步两种情况，比较复杂，本章不打算详细介绍，读者可以参考[APUE2e]。下面介绍 `pthread_exit` 的和 `pthread_join` 的用法。

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

`value_ptr` 是 `void *`类型，和线程函数返回值的用法一样，其它线程可以调用 `pthread_join` 获得这个指针。

需要注意，`pthread_exit` 或者 `return` 返回的指针所指向的内存单元必须是全局的或者是用 `malloc` 分配的，不能在线程函数的栈上分配，因为当其它线程得到这个返回指针时线程函数已经退出了。

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

返回值：成功返回 0，失败返回错误号

调用该函数的线程将挂起等待，直到 id 为 `thread` 的线程终止。`thread` 线程以不同的方法终止，通过 `pthread_join` 得到的终止状态是不同的，总结如下：



如果 `thread` 线程通过 `return` 返回，`value_ptr` 所指向的单元里存放的是 `thread` 线程函数的返回值。



如果 `thread` 线程被别的线程调用 `pthread_cancel` 异常终止掉，`value_ptr` 所指向的单元里存放的是常数 `PTHREAD_CANCELED`。



如果 `thread` 线程是自己调用 `pthread_exit` 终止的，`value_ptr` 所指向的单元存放的是传给 `pthread_exit` 的参数。

如果对 `thread` 线程的终止状态不感兴趣，可以传 `NULL` 给 `value_ptr` 参数。

看下面的例子（省略了出错处理）：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void *thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return (void *)1;
}

void *thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

void *thr_fn3(void *arg)
{
    while(1) {
        printf("thread 3 writing\n");
        sleep(1);
    }
}

int main(void)
{
    pthread_t    tid;
    void         *tret;

    pthread_create(&tid, NULL, thr_fn1, NULL);
    pthread_join(tid, &tret);
    printf("thread 1 exit code %d\n", (int)tret);

    pthread_create(&tid, NULL, thr_fn2, NULL);
    pthread_join(tid, &tret);
    printf("thread 2 exit code %d\n", (int)tret);

    pthread_create(&tid, NULL, thr_fn3, NULL);
```

```
sleep(3);
pthread_cancel(tid);
pthread_join(tid, &tret);
printf("thread 3 exit code %d\n", (int)tret);

return 0;
}
```

运行结果是：

```
$ ./a.out
thread 1 returning
thread 1 exit code 1
thread 2 exiting
thread 2 exit code 2
thread 3 writing
thread 3 writing
thread 3 writing
thread 3 exit code -1
```

可见在 Linux 的 pthread 库中常数 PTHREAD\_CANCELED 的值是-1。可以在头文件 pthread.h 中找到它的定义：

```
#define PTHREAD_CANCELED ((void *) -1)
```

一般情况下，线程终止后，其终止状态一直保留到其它线程调用 pthread\_join 获取它的状态为止。但是线程也可以被置为 detach 状态，这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。不能对一个已经处于 detach 状态的线程调用 pthread\_join，这样的调用将返回 EINVAL。对一个尚未 detach 的线程调用 pthread\_join 或 pthread\_detach 都可以把该线程置为 detach 状态，也就是说，不能对同一线程调用两次 pthread\_join，或者如果已经对一个线程调用了 pthread\_detach 就不能再调用 pthread\_join 了。

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t tid);
```

返回值：成功返回 0，失败返回错误号。

### 3. 线程间同步

#### 3.1. mutex

多个线程同时访问共享数据时可能会冲突，这跟前面讲信号时所说的可重入性是同样的问题。比如两个线程都要把某个全局变量增加 1，这个操作在某平台需要三条指令完成：

1. 从内存读变量值到寄存器
2. 寄存器的值加 1
3. 将寄存器的值写回内存

假设两个线程在多处理器平台上同时执行这三条指令，则可能导致下图所示的结果，最后变量只加了一次而非两次。

CPU1执行 线程A的指令	CPU2执行 线程A的指令	变量i的内存 单元的值
mov 0x8049540, %eax (eax = 5)	其它指令	5
add \$0x1, %eax (eax = 6)	mov 0x8049540, %eax (eax = 5)	5
mov %eax, 0x8049540 (eax = 6)	add \$0x1, %eax (eax = 6)	6
其它指令	mov %eax, 0x8049540 (eax = 6)	6

图 35.1. 并行访问冲突

思考一下，如果这两个线程在单处理器平台上执行，能够避免这样的问题吗？

我们通过一个简单的程序观察这一现象。上图所描述的现象从理论上是存在这种可能的，但实际运行程序时很难观察到，为了使现象更容易观察到，我们把上述三条指令做的事情用更多条指令来做：

```
val = counter;
printf("%x: %d\n", (unsigned int)pthread_self(), val + 1);
counter = val + 1;
```

我们在“读取变量的值”和“把变量的新值保存回去”这两步操作之间插入一个 printf 调用，它会执行 write 系统调用进内核，为内核调度别的线程执行提供了一个很好的时机。我们在一个循环中重复上述操作几千次，就会观察到访问冲突的现象。

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NLOOP 5000

int counter;                /* incremented by threads */

void *doit(void *);

int main(int argc, char **argv)
{
    pthread_t tidA, tidB;

    pthread_create(&tidA, NULL, &doit, NULL);
    pthread_create(&tidB, NULL, &doit, NULL);

    /* wait for both threads to terminate */
    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);

    return 0;
}

void *doit(void *vptr)
{
    int    i, val;

    /*
```

```
* Each thread fetches, prints, and increments the counter NLOOP times.
* The value of the counter should increase monotonically.
*/

for (i = 0; i < NLOOP; i++) {
    val = counter;
    printf("%x: %d\n", (unsigned int)pthread_self(), val + 1);
    counter = val + 1;
}

return NULL;
}
```

我们创建两个线程，各自把 counter 增加 5000 次，正常情况下最后 counter 应该等于 10000，但事实上每次运行该程序的结果都不一样，有时候数到 5000 多，有时候数到 6000 多。

```
$ ./a.out
b76acb90: 1
b76acb90: 2
b76acb90: 3
b76acb90: 4
b76acb90: 5
b7eadb90: 1
b7eadb90: 2
b7eadb90: 3
b7eadb90: 4
b7eadb90: 5
b76acb90: 6
b76acb90: 7
b7eadb90: 6
b76acb90: 8
...
```

对于多线程的程序，访问冲突的问题是很普遍的，解决的办法是引入互斥锁（Mutex，Mutual Exclusive Lock），获得锁的线程可以完成“读-修改-写”的操作，然后释放锁给其它线程，没有获得锁的线程只能等待而不能访问共享数据，这样“读-修改-写”三步操作组成一个原子操作，要么都执行，要么都不执行，不会执

行到中间被打断，也不会和其它处理器上并行做这个操作。

Mutex 用 `pthread_mutex_t` 类型的变量表示，可以这样初始化和销毁：

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

返回值：成功返回 0，失败返回错误号。

`pthread_mutex_init` 函数对 Mutex 做初始化，参数 `attr` 设定 Mutex 的属性，如果 `attr` 为 NULL 则表示缺省属性，本章不详细介绍 Mutex 属性，感兴趣的读者可以参考[APUE2e]。用 `pthread_mutex_init` 函数初始化的 Mutex 可以用 `pthread_mutex_destroy` 销毁。如果 Mutex 变量是静态分配的（全局变量或 static 变量），也可以用宏定义 `PTHREAD_MUTEX_INITIALIZER` 来初始化，相当于用 `pthread_mutex_init` 初始化并且 `attr` 参数为 NULL。Mutex 的加锁和解锁操作可以用下列函数：

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

返回值：成功返回 0，失败返回错误号。

一个线程可以调用 `pthread_mutex_lock` 获得 Mutex，如果这时另一个线程已经调用 `pthread_mutex_lock` 获得了该 Mutex，则当前线程需要挂起等待，直到另一个线程调用 `pthread_mutex_unlock` 释放 Mutex，当前线程被唤醒，才能获得该 Mutex 并继续执行。

如果一个线程既想获得锁，又不想挂起等待，可以调用



pthread\_mutex\_trylock, 如果 Mutex 已经被另一个线程获得, 这个函数会失败返回 EBUSY, 而不会使线程挂起等待。

现在我们用 Mutex 解决先前的问题:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NLOOP 5000

int counter;          /* incremented by threads */
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void *doit(void *);

int main(int argc, char **argv)
{
    pthread_t tidA, tidB;

    pthread_create(&tidA, NULL, doit, NULL);
    pthread_create(&tidB, NULL, doit, NULL);

    /* wait for both threads to terminate */
    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);

    return 0;
}

void *doit(void *vptr)
{
    int i, val;

    /*
     * Each thread fetches, prints, and increments the counter NLOOP times.
     * The value of the counter should increase monotonically.
     */

    for (i = 0; i < NLOOP; i++) {
        pthread_mutex_lock(&counter_mutex);
```

```
    val = counter;
    printf("%x: %d\n", (unsigned int)pthread_self(), val + 1);
    counter = val + 1;

    pthread_mutex_unlock(&counter_mutex);
}

return NULL;
}
```

这样运行结果就正常了，每次运行都能数到 10000。

看到这里，读者一定会好奇：Mutex 的两个基本操作 lock 和 unlock 是如何实现的呢？假设 Mutex 变量的值为 1 表示互斥锁空闲，这时某个进程调用 lock 可以获得锁，而 Mutex 的值为 0 表示互斥锁已经被某个线程获得，其它线程再调用 lock 只能挂起等待。那么 lock 和 unlock 的伪代码如下：

```
lock:
    if(mutex > 0){
        mutex = 0;
        return 0;
    } else
        挂起等待;
    goto lock;

unlock:
    mutex = 1;
    唤醒等待 Mutex 的线程;
    return 0;
```

unlock 操作中唤醒等待线程的步骤可以有不同的实现，可以只唤醒一个等待线程，也可以唤醒所有等待该 Mutex 的线程，然后让被唤醒的这些线程去竞争获得这个 Mutex，竞争失败的线程继续挂起等待。

细心的读者应该已经看出问题了：对 Mutex 变量的读取、判断和修改不是原子操作。如果两个线程同时调用 lock，这时 Mutex 是 1，两个线程都判断 mutex>0

成立，然后其中一个线程置 `mutex=0`，而另一个线程并不知道这一情况，也置 `mutex=0`，于是两个线程都以为自己获得了锁。

为了实现互斥锁操作，大多数体系结构都提供了 `swap` 或 `exchange` 指令，该指令的作用是把寄存器和内存单元的数据相交换，由于只有一条指令，保证了原子性，即使是多处理器平台，访问内存的总线周期也有先后，一个处理器上的交换指令执行时另一个处理器的交换指令只能等待总线周期。现在我们把 `lock` 和 `unlock` 的伪代码改一下（以 x86 的 `xchgb` 指令为例）：

```
lock:
    movb $0, %al
    xchgb %al, mutex
    if(al 寄存器的内容 > 0){
        return 0;
    } else
        挂起等待;
    goto lock;

unlock:
    movb $1, mutex
    唤醒等待 Mutex 的线程;
    return 0;
```

`unlock` 中的释放锁操作同样只用一条指令实现，以保证它的原子性。

也许还有读者好奇，“挂起等待”和“唤醒等待线程”的操作如何实现？每个 `Mutex` 有一个等待队列，一个线程要在 `Mutex` 上挂起等待，首先在把自己加入等待队列中，然后置线程状态为睡眠，然后调用调度器函数切换到别的线程。一个线程要唤醒等待队列中的其它线程，只需从等待队列中取出一项，它的状态从睡眠改为就绪，加入就绪队列，那么下次调度器函数执行时就有可能切换到被唤醒的线程。

一般情况下，如果同一个线程先后两次调用 `lock`，在第二次调用时，由于锁已经被占用，该线程会挂起等待别的线程释放锁，然而锁正是被自己占用着的，该线程又被挂起而没有机会释放锁，因此就永远处于挂起等待状态了，这叫做死锁（Deadlock）。另一种典型的死锁情形是这样：线程 A 获得了锁 1，线程 B 获

得了锁 2，这时线程 A 调用 lock 试图获得锁 2，结果是需要挂起等待线程 B 释放锁 2，而这时线程 B 也调用 lock 试图获得锁 1，结果是需要挂起等待线程 A 释放锁 1，于是线程 A 和 B 都永远处于挂起状态了。不难想象，如果涉及到更多的线程和更多的锁，有没有可能死锁的问题将会变得复杂和难以判断。

写程序时应该尽量避免同时获得多个锁，如果一定有必要这么做，则有一个原则：如果所有线程在需要多个锁时都按相同的先后顺序（常见的是按 Mutex 变量的地址顺序）获得锁，则不会出现死锁。比如一个程序中用到锁 1、锁 2、锁 3，它们所对应的 Mutex 变量的地址是锁 1<锁 2<锁 3，那么所有线程在需要同时获得 2 个或 3 个锁时都应该按锁 1、锁 2、锁 3 的顺序获得。如果要为所有的锁确定一个先后顺序比较困难，则应该尽量使用 pthread\_mutex\_trylock 调用代替 pthread\_mutex\_lock 调用，以免死锁。

### 3.2. Condition Variable

线程间的同步还有这样一种情况：线程 A 需要等某个条件成立才能继续往下执行，现在这个条件不成立，线程 A 就阻塞等待，而线程 B 在执行过程中使这个条件成立了，就唤醒线程 A 继续执行。在 pthread 库中通过条件变量（Condition Variable）来阻塞等待一个条件，或者唤醒等待这个条件的线程。Condition Variable 用 pthread\_cond\_t 类型的变量表示，可以这样初始化和销毁：

```
#include <pthread.h>
```

```
int pthread_cond_destroy(pthread_cond_t *cond);  
int pthread_cond_init(pthread_cond_t *restrict cond,  
    const pthread_condattr_t *restrict attr);  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

返回值：成功返回 0，失败返回错误号。

和 Mutex 的初始化和销毁类似，pthread\_cond\_init 函数初始化一个 Condition Variable，attr 参数为 NULL 则表示缺省属性，pthread\_cond\_destroy 函数销毁一个 Condition Variable。如果 Condition Variable 是静态分配的，也可以用宏定义 PTHREAD\_COND\_INITIALIZER 初始化，相当于用 pthread\_cond\_init 函数初始化并且 attr 参数为 NULL。Condition Variable 的操作可以用下列函数：

```
#include <pthread.h>
```

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,  
    pthread_mutex_t *restrict mutex,  
    const struct timespec *restrict abstime);  
int pthread_cond_wait(pthread_cond_t *restrict cond,  
    pthread_mutex_t *restrict mutex);  
int pthread_cond_broadcast(pthread_cond_t *cond);  
int pthread_cond_signal(pthread_cond_t *cond);
```

返回值：成功返回 0，失败返回错误号。

可见，一个 Condition Variable 总是和一个 Mutex 搭配使用的。一个线程可以调用 `pthread_cond_wait` 在一个 Condition Variable 上阻塞等待，这个函数做以下三步操作：

1. 释放 Mutex
2. 阻塞等待
3. 当被唤醒时，重新获得 Mutex 并返回

`pthread_cond_timedwait` 函数还有一个额外的参数可以设定等待超时，如果到达了 `abstime` 所指定的时刻仍然没有别的线程来唤醒当前线程，就返回 `ETIMEDOUT`。一个线程可以调用 `pthread_cond_signal` 唤醒在某个 Condition Variable 上等待的另一个线程，也可以调用 `pthread_cond_broadcast` 唤醒在这个 Condition Variable 上等待的所有线程。

下面的程序演示了一个生产者-消费者的例子，生产者生产一个结构体串在链表的表头上，消费者从表头取走结构体。

```
#include <stdlib.h>  
#include <pthread.h>  
#include <stdio.h>  
  
struct msg {  
    struct msg *next;  
    int num;
```

```
};

struct msg *head;
pthread_cond_t has_product = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *consumer(void *p)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&lock);
        while (head == NULL)
            pthread_cond_wait(&has_product, &lock);
        mp = head;
        head = mp->next;
        pthread_mutex_unlock(&lock);
        printf("Consume %d\n", mp->num);
        free(mp);
        sleep(rand() % 5);
    }
}

void *producer(void *p)
{
    struct msg *mp;
    for (;;) {
        mp = malloc(sizeof(struct msg));
        mp->num = rand() % 1000 + 1;
        printf("Produce %d\n", mp->num);
        pthread_mutex_lock(&lock);
        mp->next = head;
        head = mp;
        pthread_mutex_unlock(&lock);
        pthread_cond_signal(&has_product);
        sleep(rand() % 5);
    }
}

int main(int argc, char *argv[])
{
    pthread_t pid, cid;

    srand(time(NULL));
```

```
pthread_create(&pid, NULL, producer, NULL);  
pthread_create(&cid, NULL, consumer, NULL);  
pthread_join(pid, NULL);  
pthread_join(cid, NULL);  
return 0;  
}
```

执行结果如下：

```
$ ./a.out  
Produce 744  
Consume 744  
Produce 567  
Produce 881  
Consume 881  
Produce 911  
Consume 911  
Consume 567  
Produce 698  
Consume 698
```

### ✓ 习题

1、在本节的例子中，生产者和消费者访问链表的顺序是 LIFO 的，请修改程序，把访问顺序改成 FIFO。

## 3.3. Semaphore

Mutex 变量是非 0 即 1 的，可看作一种资源的可用数量，初始化时 Mutex 是 1，表示有一个可用资源，加锁时获得该资源，将 Mutex 减到 0，表示不再有可用资源，解锁时释放该资源，将 Mutex 重新加到 1，表示又有了一个可用资源。

信号量（Semaphore）和 Mutex 类似，表示可用资源的数量，和 Mutex 不同的是这个数量可以大于 1。



本节介绍的是 POSIX semaphore 库函数，详见 `sem_overview(7)`，这种信号量不仅可用于同一进程的线程间同步，也可用于不同进程间的同步。

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

semaphore 变量的类型为 `sem_t`，`sem_init()` 初始化一个 semaphore 变量，`value` 参数表示可用资源的数量，`pshared` 参数为 0 表示信号量用于同一进程的线程间同步，本节只介绍这种情况。在用完 semaphore 变量之后应该调用 `sem_destroy()` 释放与 semaphore 相关的资源。

调用 `sem_wait()` 可以获得资源，使 semaphore 的值减 1，如果调用 `sem_wait()` 时 semaphore 的值已经是 0，则挂起等待。如果不希望挂起等待，可以调用 `sem_trywait()`。调用 `sem_post()` 可以释放资源，使 semaphore 的值加 1，同时唤醒挂起等待的线程。

上一节生产者—消费者的例子是基于链表的，其空间可以动态分配，现在基于固定大小的环形队列重写这个程序：

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>

#define NUM 5
int queue[NUM];
sem_t blank_number, product_number;

void *producer(void *arg)
{
    int p = 0;
    while (1) {
```

```
        sem_wait(&blank_number);
        queue[p] = rand() % 1000 + 1;
        printf("Produce %d\n", queue[p]);
        sem_post(&product_number);
        p = (p+1)%NUM;
        sleep(rand()%5);
    }
}

void *consumer(void *arg)
{
    int c = 0;
    while (1) {
        sem_wait(&product_number);
        printf("Consume %d\n", queue[c]);
        queue[c] = 0;
        sem_post(&blank_number);
        c = (c+1)%NUM;
        sleep(rand()%5);
    }
}

int main(int argc, char *argv[])
{
    pthread_t pid, cid;

    sem_init(&blank_number, 0, NUM);
    sem_init(&product_number, 0, 0);
    pthread_create(&pid, NULL, producer, NULL);
    pthread_create(&cid, NULL, consumer, NULL);
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
    sem_destroy(&blank_number);
    sem_destroy(&product_number);
    return 0;
}
```

## ✓ 习题

1、本节和上一节的例子给出一个重要的提示：用 Condition Variable 可以实

现 Semaphore。请用 Condition Variable 实现 Semaphore，然后用自己实现的 Semaphore 重写本节的程序。

### 3.4. 其它线程间同步机制

如果共享数据是只读的，那么各线程读到的数据应该总是一致的，不会出现访问冲突。只要有一个线程可以改写数据，就必须考虑线程间同步的问题。由此引出了读者写者锁（Reader-Writer Lock）的概念，Reader 之间并不互斥，可以同时读共享数据，而 Writer 是独占的(exclusive)，在 Writer 修改数据时其它 Reader 或 Writer 不能访问数据，可见 Reader-Writer Lock 比 Mutex 具有更好的并发性。

用挂起等待的方式解决访问冲突不见得是最好的办法，因为这样毕竟会影响系统的并发性，在某些情况下解决访问冲突的问题可以尽量避免挂起某个线程，例如 Linux 内核的 Seqlock、RCU (read-copy-update) 等机制。

关于这些同步机制的细节，有兴趣的读者可以参考[APUE2e]和[ULK]。

## 4. 编程练习

哲学家就餐问题。这是由计算机科学家 Dijkstra 提出的经典死锁场景。

原版的故事里有五个哲学家(不过我们写的程序可以有 N 个哲学家)，这些哲学家们只做两件事——思考和吃饭，他们思考的时候不需要任何共享资源，但是吃饭的时候就必须使用餐具，而餐桌上的餐具是有限的，原版的故事里，餐具是叉子，吃饭的时候要用两把叉子把面条从碗里捞出来。很显然把叉子换成筷子会更合理，所以：一个哲学家需要两根筷子才能吃饭。

现在引入问题的关键：这些哲学家很穷，只买得起五根筷子。他们坐成一圈，两个人的中间放一根筷子。哲学家吃饭的时候必须同时得到左手边和右手边的筷子。如果他身边的任何一位正在使用筷子，那他只有等着。

假设哲学家的编号是 A、B、C、D、E，筷子编号是 1、2、3、4、5，哲学家和筷子围成一圈如下图所示：

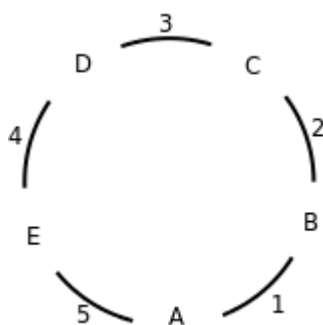


图 35.2. 哲学家问题

每个哲学家都是一个单独的线程，每个线程循环做以下动作：思考  $\text{rand()} \% 10$  秒，然后先拿左手边的筷子再拿右手边的筷子（筷子这种资源可以用 `mutex` 表示），有任何一边拿不到就一直等着，全拿到就吃饭  $\text{rand()} \% 10$  秒，然后放下筷子。

编写程序仿真哲学家就餐的场景：

```
Philosopher A fetches chopstick 5
Philosopher B fetches chopstick 1
Philosopher B fetches chopstick 2
Philosopher D fetches chopstick 3
Philosopher B releases chopsticks 1 2
Philosopher A fetches chopstick 1
Philosopher C fetches chopstick 2
Philosopher A releases chopsticks 5 1
...
```

分析一下，这个过程有没有可能产生死锁？调用 `usleep(3)` 函数可以实现微秒级的延时，试着用 `usleep(3)` 加快仿真的速度，看能不能观察到死锁现象。然后修改上述算法避免产生死锁。

## 第 10 章 音乐播放

---

课程内容：

- ✧ 数字音频格式简介
- ✧ mp3 编码技术
- ✧ mp3 编码格式

## 第 10 章 音乐播放

### 1、数字音频格式简介

首先，我们来明确一下数字音频的概念，它是指一个用来表示声音强弱的数据序列，由模拟声音经抽样、量化和编码后得到的。简单地说，数字音频的编码方式就是数字音频格式，我们所使用的不同的数字音频设备一般都对应着不同的音频文件格式。常见的数字音频格式有：

#### ✓ WAV

WAV 格式，是微软公司开发的一种声音文件格式，也叫波形声音文件，是最早的数字音频格式，被 Windows 平台及其应用程序广泛支持。WAV 格式支持许多压缩算法，支持多种音频位数、采样频率和声道，采用 44.1kHz 的采样频率，16 位量化位数，跟 CD 一样，对存储空间需求太大不便于交流和传播。

#### ✓ MIDI

MIDI 是 Musical Instrument Digital Interface 的缩写，又称作乐器数字接口，是数字音乐/电子合成乐器的统一国际标准。它定义了计算机音乐程序、数字合成器及其它电子设备交换音乐信号的方式，规定了不同厂家的电子乐器与计算机连接的电缆和硬件及设备间数据传输的协议，可以模拟多种乐器的声音。MIDI 文件就是 MIDI 格式的文件，在 MIDI 文件中存储的是一些指令。把这些指令发送给声卡，由声卡按照指令将声音合成出来。

#### ✓ CD

大家都很熟悉 CD 这种音乐格式了，扩展名 CDA，其取样频率为 44.1kHz，16 位量化位数，跟 WAV 一样，但 CD 存储采用了音轨的形式，又叫“红皮书”格式，记录的是波形流，是一种近似无损的格式。

#### ✓ MP3

MP3 全称是 MPEG-1 Audio Layer 3，它在 1992 年合并至 MPEG 规范中。MP3 能够以高音质、低采样率对数字音频文件进行压缩。换句话说，音频文件(主要是大型文件，比如 WAV 文件)能够在音质丢失很小的情况下(人耳根本无法察觉这种音质损失)把文件压缩到更小的程度。

### ✓ **MP3Pro**

MP3Pro 是由瑞典 Coding 科技公司开发的, 其中包含了两大技术: 一是来自于 Coding 科技公司所特有的解码技术, 二是由 MP3 的专利持有者法国汤姆森多媒体公司和德国 Fraunhofer 集成电路协会共同研究的一项译码技术。MP3Pro 可以在基本不改变文件大小的前提下改善原先的 MP3 音乐音质。它能够在用较低的比特率压缩音频文件的情况下, 最大程度地保持压缩前的音质。

### ✓ **WMA**

WMA (Windows Media Audio)是微软在互联网音频、视频领域的力作。WMA 格式是以减少数据流量但保持音质的方法来达到更高的压缩率目的, 其压缩率一般可以达到 1:18。此外, WMA 还可以通过 DRM (Digital Rights Management) 方案加入防止拷贝, 或者加入限制播放时间和播放次数, 甚至是播放机器的限制, 可有力地防止盗版。

### ✓ **MP4**

MP4 采用的是美国电话电报公司(AT&T)所研发的以“知觉编码”为关键技术的 a2b 音乐压缩技术, 由美国网络技术公司(GMO)及 RIAA 联合公布的一种新的音乐格式。MP4 在文件中采用了保护版权的编码技术, 只有特定的用户才可以播放, 有效地保证了音乐版权的合法性。另外 MP4 的压缩比达到了 1:15, 体积较 MP3 更小, 但音质却没有下降。不过因为只有特定的用户才能播放这种文件, 因此其流传与 MP3 相比差距甚远。

### ✓ **SACD**

SACD (SA=SuperAudio)是由 Sony 公司正式发布的。它的采样率为 CD 格式的 64 倍, 即 2.8224MHz。SACD 重放频率带宽达 100kHz, 为 CD 格式的 5 倍, 24 位量化位数, 远远超过 CD, 声音的细节表现更为丰富、清晰。

### ✓ **QuickTime**

QuickTime 是苹果公司于 1991 年推出的一种数字流媒体, 它面向视频编辑、Web 网站创建和媒体技术平台, QuickTime 支持几乎所有主流的个人计算平台, 可以通过互联网提供实时的数字化信息流、工作流与文件回放功能。现有版本为 QuickTime 1.0、2.0、3.0、4.0 和 5.0, 在 5.0 版本中还融合了支持最高 A/V 播放质量的播放器等多项新技术。



## ✓ VQF

VQF 格式是由 YAMAHA 和 NTT 共同开发的一种音频压缩技术，它的压缩率能够达到 1:18, 因此相同情况下压缩后 VQF 的文件体积比 MP3 小 30%~50%, 更便于网上传播, 同时音质极佳, 接近 CD 音质(16 位 44.1kHz 立体声)。但 VQF 未公开技术标准, 至今未能流行开来。

## ✓ DVD Audio

DVD Audio 是新一代的数字音频格式, 与 DVD Video 尺寸以及容量相同, 为音乐格式的 DVD 光碟, 取样频率为“48kHz/96kHz/192kHz”和“44.1kHz/88.2kHz/176.4kHz”可选择, 量化位数可以为 16、20 或 24 比特, 它们之间可自由地进行组合。低采样率的 192kHz、176.4kHz 虽然是 2 声道重播专用, 但它最多可收录到 6 声道。而以 2 声道 192kHz/24b 或 6 声道 96kHz/24b 收录声音, 可容纳 74 分钟以上的录音, 动态范围达 144dB, 整体效果出类拔萃。

## ✓ MD

Sony 公司的 MD (MiniDisc) 大家都很熟悉了。MD 之所以能在一张小小的盘中存储 60~80 分钟采用 44.1kHz 采样的立体声音乐, 就是因为使用了 ATRAC 算法 (自适应声学转换编码) 压缩音源。这是一套基于心理声学原理的音响译码系统, 它可以把 CD 唱片的音频压缩到原来数据量的大约 1/5 而声音质量没有明显的损失。ATRAC 利用人耳听觉的心理声学特性 (频谱掩蔽特性和时间掩蔽特性) 以及人耳对信号幅度、频率、时间的有限分辨能力, 编码时将人耳感觉不到的成分不编码, 不传送, 这样就可以相应减少某些数据量的存储, 从而既保证音质又达到缩小体积的目的。

### RealAudio

是由 Real Networks 公司推出的一种文件格式, 最大的特点就是可以实时传输音频信息, 尤其是在网速较慢的情况下, 仍然可以较为流畅地传送数据, 因此 RealAudio 主要适用于网络上的在线播放。现在的 RealAudio 文件格式主要有 RA(RealAudio)、RM (RealMedia, RealAudio G2)、RMX(RealAudio Secured) 等三种, 这些文件的共同性在于随着网络带宽的不同而改变声音的质量, 在保证大多数人听到流畅声音的前提下, 令带宽较宽敞的听众获得较好的音质。

## ✓ Liquid Audio

Liquid Audio 是一家提供付费音乐下载的网站。它通过在音乐中采用自己独有的音频编码格式来提供对音乐的版权保护。Liquid Audio 的音频格式就是所谓

的 LQT。如果想在 PC 中播放这种格式的音乐，你就必须使用 Liquid Player 和 Real Jukebox 其中的一种播放器。这些文件也不能够转换成 MP3 和 WAV 格式，因此这使得采用这种格式的音频文件无法被共享和刻录到 CD 中。如果非要把 Liquid Audio 文件刻录到 CD 中的话，就必须使用支持这种格式的刻录软件和 CD 刻录机。

#### ✓ Audible

Audible 拥有四种不同的格式：Audible1、2、3、4。Audible.com 网站主要是在互联网上贩卖有声书籍，并对它们所销售商品、文件通过四种 Audible.com 专用音频格式中的一种提供保护。每一种格式主要考虑音频源以及所使用的收听的设备。格式 1、2 和 3 采用不同级别的语音压缩，而格式 4 采用更低的采样率和 MP3 相同的解码方式，所得到语音吐辞更清楚，而且可以更有效地从网上进行下载。Audible 所采用的是他们自己的桌面播放工具，这就是 Audible Manager，使用这种播放器就可以播放存放在 PC 或者是传输到便携式播放器上的 Audible 格式文件。

#### ✓ VOC

VOC 文件，在 DOS 程序和游戏中常会遇到这种文件，它是随声霸卡一起产生的数字声音文件，与 WAV 文件的结构相似，可以通过一些工具软件方便地互相转换。

#### ✓ AU

AU 文件，在 Internet 上的多媒体声音主要使用该种文件。AU 文件是 UNIX 操作系统下的数字声音文件，由于早期 Internet 上的 Web 服务器主要是基于 UNIX 的，所以这种文件成为 WWW 上唯一使用的标准声音文件。

#### ✓ AIFF(.AIF)

AIFF(.AIF) 是苹果公司开发的声音文件格式，被 Macintosh 平台和应用程序所支持。

#### ✓ Amiga 声音(.SVX)

Amiga 声音(.SVX)：Commodore 所开发的声音文件格式，被 Amiga 平台和应用程序所支持，不支持压缩。

## ✓ MAC 声音(.snd)

MAC 声音(.snd) : Apple 计算机公司所开发的声音文件格式, 被 Macintosh 平台和多种 Macintosh 应用程序所支持, 支持某些压缩。

## ✓ S48

S48(stereo、48kHz)采用 MPEG-1 layer 1、MPEG-1 layer 2 (简称 Mp1,Mp2) 声音压缩格式, 由于其易于编辑、剪切, 所以在广播电台应用较广。

## ✓ AAC

AAC 实际上是高级音频编码的缩写。AAC 是由 Fraunhofer IIS-A、杜比和 AT&T 共同开发的一种音频格式, 它是 MPEG-2 规范的一部分。AAC 所采用的运算法则与 MP3 的运算法则有所不同, AAC 通过结合其他的功能 来提高编码效率。AAC 的音频算法在压缩能力上远远超过了以前的一些压缩算法(比如 MP3 等)。它还同时支持多达 48 个音轨、15 个低频音轨、更多种采样率和比特率、多种语言的兼容能力、更高的解码效率。总之, AAC 可以在比 MP3 文件缩小 30% 的前提下提供更好的音质。

数字音频以音质优秀、传播无损耗、可进行多种编辑和转换而成为主流, 并且应用于各个方面。

## 2、mp3 编码技术

### 2.1. mp3

我们所说的 mp3, 对应着 3 个版本, mpeg1 audio, mpeg2 audio, mpeg2.5 audio, 其中 mpeg1 audio 只对应双声道或者单声道, 而采样频率只有 32kHz, 44.1kHz, 48kHz。然后, 在 mpeg2 中扩展了 22kHz, 24kHz, 16kHz, 紧接着在最后的 mpeg2.5 中扩展了 11kHz, 12kHz, 8kHz。这说明, 不同的采样率, 对应的 mpeg 标准是不同的, mpeg2 中不仅仅扩展了采样率, 同时扩展了 mpeg1 中 audio layer 3 的通道数量, 从而支持 5.1, 7.1 声道。还有就是将 mpeg1 audio 中的速率由 32-384kbps 到 8-640bps, 这里可以知道, 如果是多声道的 mp3, 则其最大速率为 640kbps, 而不是大家所认为的 320kbps。mpeg2 中的 audio layer 1, 2, 3 部分是同 mpeg1 audio 兼容的, 但是因为 mpeg2 中提出了 aac, 这部分的格式与 mpeg1 的不相兼容, 所以 mpeg2 统称为非后向兼容, 因为它只是一部分兼容了 mpeg1 audio。

## 2.2.掩蔽效应

从 mpeg1 中的 audio 部分开始,主要采用了 3 个听觉系统的感知特性:响度,音调,掩蔽效应。

响度:在听阈(最小响度)与痛域(最大响度)之间的响度,也就人耳的听觉范围,这里涉及到两个概念,客观测量值与主观测量值。前者不必多说,即认为用仪器测量为准。而后者,主观测量值,则是用人耳来判断两个不同频率的声音,响度是否相同,也就是说,人耳认为相同响度的两个声音,客观测量值则完全不同。例如:1kHz 的 10dB 声音,同 200Hz 的 30dB 的声音,人耳听起来具有相同的响度。而 10dB 同 30dB,却恰恰是客观的物理的数值。

音调:即所谓声音的高低。也就是频率低的声音,音调低,反之,音调高。这里,也分为客观音高与主观音高。关系可以列入一个约等号公式。这样,人耳对于频率也就有了一个范围,20-18000Hz。

现在有了两个范围,响度范围,频率范围。

掩蔽效应:一个强的纯音(单一频率声音称之为纯音)会掩蔽其附近同时发生的弱纯音,这种特性称之为频域掩蔽。不仅仅音频本身会掩蔽,噪声的存在,如果条件达到了一定,同样会对音频产生掩蔽,偶就不废话了,不过从而引入一个概念:临界带宽。

同时,还存在着第二种掩蔽,时域掩蔽。顾名思义,在时间上,相邻的声音也存在掩蔽现象。分为:超前,滞后掩蔽。

稍微总结一下,其实虽然有 3 种方式的掩蔽,但是 mpeg 中的 audio layer 1,2,3 三层却有很大的区别。而其中只有 3 才完全利用了上面的所有掩蔽效应,这也是为什么在 1,2,3 中,只有 3 在低速率下实现了近似 cd 的效果。

这里要说明,mpeg audio 的 1, 2, 3 层,由于各式的关系,无法实现高于 cd 的效果。因为它的采样速率小于等于 48kHz,而样本精度小于等于 16bit(在 layer3 中,对于样本的精度是可变的,但是最大为 16bit(另一说法是 24bit),而 1, 2 中则为定长)。尽管如此,据 mpeg 小组所作的实验表明,48kHz,16bit,256kbps 的声音压缩数据,专业的测试人员也无法分清。但并不是说压缩音频无法超越 cd 的标准格式,因为从 mpeg2 中的 aac 开始,采样率范围扩展为 8--96kHz,同时支持 48 个主声道,16 个低频加强通道,16 个配音声道以及 16 个数据流,这是很诱人的,自然超过 cd 也是不在话下。其实 aac 也并不是以提供 cd 音质为目

标，而是以提供原始声音为目标的。

## 2.3 提前说 mp3 的版权控制。

mpeg audio layer 123 压缩格式为帧结构。帧头为 32bit 即 4byte。其中：

第 3 位为版权标志，0 表示原版媒体的复制，1 表示原版媒体。

第 8 位为私有 bit。

之所以先说这两个标志，因为通过他们，可以实现完全类似于 netmd 中录入的 3 次限制以及版权保护。

第 15 位到 12 位为 vbr 标示，也就是说，近年被大家广泛谈论的压缩方式，早在 93 年刚刚发布 mpeg 的时候就已经准备好了。可惜这么晚才让它发扬光大，不过也说明了格式的制定是有前瞻性的。所以当前的 aac 虽然看起来支持多声道显得无所作为，但是相信终究会让大家体验到他的威力的一天。

## 2.4 mpeg audio layer

mpeg audio layer 1,2,3 编码（mpeg1,mpeg2,mpeg2.5 的 layer 部分的编码基本上是一致的，后两者主要在采样率上作了扩展从而更广泛的适应各种场合的需要，所以下面简称为 layer 1,2,3）。

简要的介绍一下声音数据的压缩原理，声音的数据量两方面决定：采样频率，样本精度。根据乃奎斯特理论，如果要想不是真的重建信号，采样频率不能低于 40KHz,但是由于实际中使用的滤波器都不可能是理想的，并且考虑到各国所使用的交流电源频率，所以采样率一般不能低于 44.1KHz.所以压缩只能从第二个方面，也就是“样本精度”来考虑了。

## 3、 mp3 编码格式

将 CD 唱片、影碟上的音频数据或者其他音频文件转换为 MP3 文件，实际上就是一个将音频数据重新以 MP3 压缩格式编码的过程。而 MP3 编码的质量和编码的速度都取决于编码器，换言之 MP3 编码器是决定我们是否能够制作出高品质 MP3 的关键。目前有许多种 MP3 编码器可供选择，不过对于 MP3 压缩技术有一定了解的音乐爱好者都会毫不犹豫地选择 Lame 编码器。Lame 编码独创的听觉心理模型结合了 VBR、ABR 等多种编码模式，可以让你自由地选择自己需要的 MP3 编码效果和文件大小。它能够与众多音频处理软件协同工作，让你



在常用的数字音频处理软件上通过调用外部编码器的方式直接将 CD 唱片上的歌曲或者其他音频格式的文件编码为 MP3。

Lame 开发者（[www.mp3dev.org](http://www.mp3dev.org)）只提供了编码器的源代码，需要可运行版本的话可以到 [http://www.hot.ee/smpman /mp3/](http://www.hot.ee/smpman/mp3/) 下载。编码器可运行在命令行状态下或供其他音频处理软件调用，而要使用哪种编码模式和比特率完全由参数决定。由于 Lame 编码器有数十个可选的参数，要按照这些参数自己定义出一套合适的编码方案会相当复杂。因此，这里笔者不准备逐一为你介绍这些编码参数，只介绍 Lame 开发者为编码器预设的几组参数。虽然在一些音频处理软件上你可以通过软件提供的菜单直观地定义编码参数，但在命令行状态下和大部分音频处理软件上你将仍然需要自行指定参数，为了获得最佳的 MP3 编码效果，本文中所有制作 MP3 的方法都强调使用 Lame 编码器。

### ✓ CBR 预设参数

CBR（Constant Bit Rate）编码模式采用常数比特率，所谓常数比特率，也就是说使用这种编码模式的 MP3 文件每秒钟的数据流量是固定的，这种编码模式的优点是压缩速度快，缺点是压缩效率并不高。常用的 CBR 预设参数有以下三种：“--alt-preset insane”和“--alt-preset cbr 320”参数的作用一样，都是让编码器采用 320kbps 的常数比特率，由于 320kbps 是 Lame 编码的最高比特率，因此使用该参数编码后的 MP3 文件品质最高，但文件体积也将会是最大的；“--alt-preset cbr <比特率>”则可由用户自定义比特率，可选比特率有 80、96、112、128、160、192、224、256、320。

### ✓ VBR 预设参数

VBR（Variable Bit Rate）模式采用动态比特率，也就是说这种编码模式每秒钟的数据流量是可以变化的，编码时编码器根据音频数据确定使用什么比特率，音频数据波型比较复杂时采用高用比特率进行编码，比较简单时就用比较低的比特率。其缺点是编码的速度较慢，优点是压缩效率较高，在保证质量的同时兼顾了文件大小，是目前最流行的 MP3 编码模式。常用的 VBR 预设参数有以下四种：“--alt-preset extreme”是 220 至 270 kbps 左右的 VBR 编码方案，音质接近最高品质的 320kbps CBR 编码文件，但文件要小 25%；“--alt-preset standard”是 180 至 220 kbps 左右的 VBR 编码方案，兼顾了音质和文件大小；“--alt-preset fast extreme”介于上面两种方案之间，音质较第一种方案差，文件比第二种方案大；参数“--alt-preset fast standard”与之相似，比“--alt-preset standard”方案压缩的速度快点，文件也大点。

✓ **ABR 预设参数**

ABR (Average Bit Rate) 模式采用平均比特率, 这种编码模式在指定的平均比特率内, 以每 50 帧 (30 帧约 1 秒) 为一段, 对编码内容的低频和不敏感频率的段落使用相对较低的比特率, 高频和大动态表现的段落使用高比特率。在编码的过程中大部分音频数据使用指定的比特率编码, 剩余的部分通过动态调整进行优化。在同一比特率下, ABR 编码的 MP3 文件与 CBR 编码的文件大小相差不多, 但音质却提高不少。而由于大部分内容使用指定的比特率, 编码的速度比 VBR 编码模式要快很多, 可以作为 VBR 和 CBR 的一种折衷选择。“--alt-preset <比特率>” 是用户自定义比特率的 ABR 参数, 可选的比特率有 80、96、112、128、160、192、224、256、320。



## 附录 重要的参考资料

---

课程内容：

- ✧ HTML 语法
- ✧ 图片压缩知识基础
- ✧ TTF 文件格式和使用

## 附录 A 重要参考资料

### HTML 语法

图片信息格式基础

可以为一个图像指定链接 HTML 图像通常使用 `img` 标签，如：

```
<a href="http://www.baidu.com/" title="baidu">

</a>
<a href="http://www.google.com/" title="google">


</a>
```


#### 1、img 标签


`img` 标签：代表 HTML 图像


`img` 标签是单独出现的，`<img />`

✓ 属性：

 `alt`：代表图像的替代文字

 `src`：代表一个图像源(就是图像的位置)

 `height`：代表一个图像的高度

 `width`：代表一个图像的宽度

`img` 是 `image`(图像)的缩写


#### 2、img 标签语法

```

```

`img` 标签说明

 `img`：是图像的标签

 `src`：属性告诉浏览器图片的具体位置，就像连接的 `href` 标签一样告诉浏览器要链接到的文件，可以用 `URI` 表示，`URI` 表示法见相对路径和绝对路径

alt: 图片的替代文字。有些浏览者不想看到图片(比如由于网速太慢), 有些早期的浏览器也不支持图片, 还有一种可能'你把图片的具体位置写错了', 这些情况浏览者是看不到图片的, 这时 alt 可以在图片的位置上显示出代替的文字, 这是非常有用的, 记得一定要加上

title: 图片的提示文字, 当鼠标停留到图片上时, 会提示相关文字

上面的例子中并没有指定图像的宽度 width 与高度 height, 其实浏览器会自动识别出图像的高度与宽度, 一般不用我们指定, 除非有其它的目的, 比如有意的增大或缩小图片

### 3、示例

```

```

### 4、网页设计中常见的图像格式简介

设计网页时经常使用的图片有三种, 它们的相同点是都经过了压缩, 压缩比越高, 图像品质越差。

#### 1).GIF

GIF(Graphics Interchange Format): 最多支持 256 色, 支持透明,支持多帧动画显示效果。(可以使用在品质较差或者需要透明或动画显示的情况)

#### 2).JPEG

JPEG(Joint Photographic Experts Group): 支持多种颜色, 可以有很高的压缩比, 使用了有损压缩, 不支持透明, 不支持动画效果。(可以使用在品质要求较高的场合: 风景, 写真等)

#### 3).PNG

PNG(Portable Network Graphics): 是一种新的图片技术, 可以表现品质比较高的图片, 使用了无损压缩, 支持透明, 不支持动画。(是 w3c 推荐的 <http://www.w3.org/Graphics/PNG/>)

## 图片压缩知识基础

### 1、数据压缩引言

随着多媒体技术和通讯技术的不断发展，多媒体娱乐、信息高速公路等不断对信息数据的存储和传输提出了更高的要求，也给现有的有限带宽以严峻的考验，特别是具有庞大数据量的数字图像通信，更难以传输和存储，极大地制约了图像通信的发展，因此图像压缩技术受到了越来越多的关注。图像压缩的目的就是把原来较大的图像用尽量少的字节表示和传输，并且要求复原图像有较好的质量。利用图像压缩，可以减轻图像存储和传输的负担，使图像在网络上实现快速传输和实时处理。

图像压缩编码技术可以追溯到 1948 年提出的电视信号数字化，到今天已经有 50 多年的历史了。在此期间出现了很多种图像压缩编码方法，特别是到了 80 年代后期以后，由于小波变换理论，分形理论，人工神经网络理论，视觉仿真理论的建立，图像压缩技术得到了前所未有的发展，其中分形图像压缩和小波图像压缩是当前研究的热点。以下内容将对当前最为广泛使用的图像压缩算法进行阐述并讨论它们的优缺点及发展前景。

#### JPEG 压缩详述

JPEG (Joint Photographic Experts Group) 是一个由 ISO 和 IEC 两个组织机构联合组成的一个专家组，负责制定静态的数字图像数据压缩编码标准，这个专家组开发的算法称为 JPEG 算法，并且成为国际上通用的标准，因此又称为 JPEG 标准。JPEG 是一个适用范围很广的静态图像数据压缩标准，既可用于灰度图像又可用于彩色图像。

JPEG 专家组开发了两种基本的压缩算法，一种是采用以离散余弦变换 (Discrete Cosine Transform, DCT) 为基础的有损压缩算法，另一种是采用以预测技术为基础的无损压缩算法。使用有损压缩算法时，在压缩比为 25:1 的情况下，压缩后还原得到的图像与原始图像相比较，非图像专家难于找出它们之间的区别，因此得到了广泛的应用。例如，在 VCD 和 DVD-Video 电视图像压缩技术中，就使用 JPEG 的有损压缩算法来取消空间方向上的冗余数据。为了在保证图像质量的前提下进一步提高压缩比，近年来 JPEG 专家组正在制定 JPEG2000 标准，这个标准中将采用小波变换 (Wavelet) 算法。

JPEG 压缩是有损压缩，它利用了人的视角系统的特性，使用量化和无损压缩编码相结合来去掉视角的冗余信息和数据本身的冗余信息。JPEG 算法框图如图：

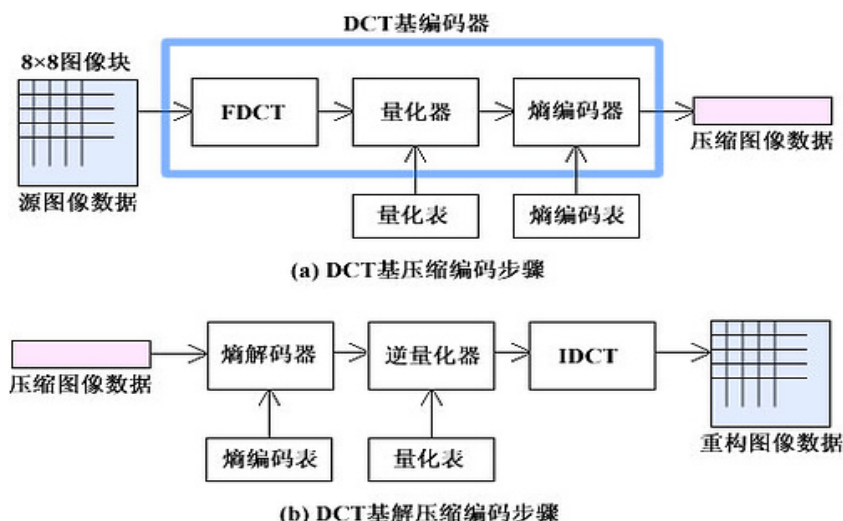


图 1 JPEG 压缩算法框图

压缩编码大致分成三个步骤：

1、使用正向离散余弦变换（Forward Discrete Cosine Transform, FDCT）把空间域表示的图变换成频率域表示的图。

2、使用加权函数对 DCT 系数进行量化，这个加权函数对于人的视觉系统是最佳的。

3、使用霍夫曼可变字长编码器对量化系数进行编码。

译码或者叫做解压缩的过程与压缩编码过程正好相反。

JPEG 算法与彩色空间无关，因此“RGB 到 YUV 变换”和“YUV 到 RGB 变换”不包含在 JPEG 算法中。JPEG 算法处理的彩色图像是单独的彩色分量图像，因此它可以压缩来自不同彩色空间的数据，如 RGB, YCbCr 和 CMYK。

## 2、JPEG 算法的主要计算步骤

JPEG 压缩编码算法的主要计算步骤如下：

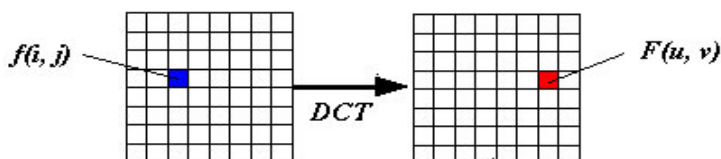
- 1.正向离散余弦变换（FDCT）
- 2.量化（Quantization）
- 3.Z 字形编码（Zigzag Scan）
- 4.使用差分脉冲编码调制（Differential Pulse Code Modulation, DPCM)对直流系数(DC)进行编码
- 5.使用行程长度编码（Run-Length Encoding, RLE）对交流系数（AC）进行编码
- 6.熵编码（Entropy Eoding）

### ✓ 1、正向离散余弦变换

下面对正向离散余弦变换(FDCT)变换作几点说明。

- 1).对每个单独的彩色图像分量，把整个分量图像分成若干个 8×8 的图像块，如图所示，

并作为两维离散余弦变换 DCT 的输入。通过 DCT 变换，把能量集中在少数几个系数上。



2). DCT 变换使用下式计算:

$$F(u, v) = \frac{1}{4} C(u) C(v) \left[ \sum_{i=0}^7 \sum_{j=0}^7 f(i, j) \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} \right]$$

它的逆变换使用下式计算:

$$f(i, j) = \frac{1}{4} C(u) C(v) \left[ \sum_{u=0}^7 \sum_{v=0}^7 F(u, v) \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} \right]$$

上面两式中,

$C(u), C(v) = (2)^{-1/2}$ , 当  $u, v = 0$ ;

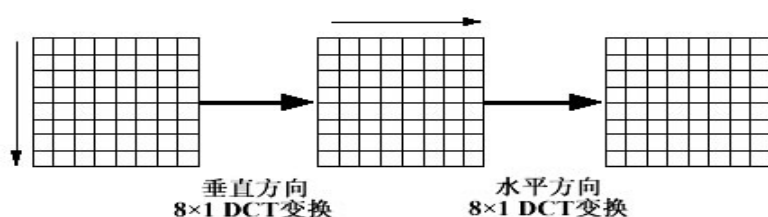
$C(u), C(v) = 1$ , 其他。

$f(i, j)$  经 DCT 变换之后,  $F(0, 0)$  是直流系数, 其他为交流系数。

3). 在计算两维的 DCT 变换时, 可使用下面的计算式把两维的 DCT 变换变成一维的 DCT 变换:

$$F(u, v) = \frac{1}{2} C(u) \left[ \sum_{i=0}^7 G(i, v) \cos \frac{(2i+1)u\pi}{16} \right]$$

$$G(i, v) = \frac{1}{2} C(v) \left[ \sum_{j=0}^7 f(i, j) \cos \frac{(2j+1)v\pi}{16} \right]$$

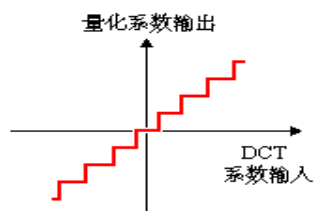


## ✓ 2、量化

量化是对经过 FDCT 变换后的频率系数进行量化。量化的目的是减小非“0”系数的幅度以及增加“0”值系数的数目。量化是图像质量下降的最主要原因。

对于有损压缩算法, JPEG 算法使用如下图所示的均匀量化器进行量化, 量化步距是按照系数所在的位置和每种颜色分量的色调值来确定。因为人眼对亮度信号比对色差信号更敏感, 因此使用了两种量化表: 亮度量化值和色差量化值。此外, 由于人眼对低频分量的图像比对高频分量的图像更敏感, 因此图中的左上角的量化步距要比右下角的量化步距小。下面

2 个表中的数值对 CCIR 601 标准电视图像已经是最佳的。如果不使用这两种表，你也可以把自己的量化表替换它们。



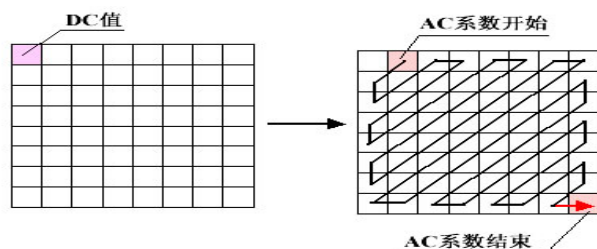
亮度量化值表和色度量化值表：

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

### ✓ 3、Z 字形编排

量化后的系数要重新编排，目的是为了增加连续的“0”系数的个数，就是“0”的游程长度，方法是按照 Z 字形的式样编排，如下图所示。这样就把一个 8×8 的矩阵变成一个 1×64 的矢量，频率较低的系数放在矢量的顶部。



量化 DCT 系数序号

0	1	5	6	4	5	7	25
2	4	7	3	6	6	9	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63



#### ✓ 4、直流系数的编码

8×8 图像块经过 DCT 变换之后得到的 DC 直流系数有两个特点，一是系数的数值比较大，二是相邻 8×8 图像块的 DC 系数值变化不大。根据这个特点，JPEG 算法使用了差分脉冲调制编码（DPCM）技术，对相邻图像块之间量化 DC 系数的差值（Delta）进行编码。

$$\Delta = DC(0,0)_k - DC(0,0)_{k-1}$$

#### ✓ 5、交流系数的编码

量化 AC 系数的特点是 1×64 矢量中包含有许多“0”系数，并且许多“0”是连续的，因此使用非常简单和直观的游程长度编码(RLE)对它们进行编码。

JPEG 使用了 1 个字节的高 4 位来表示连续“0”的个数，而使用它的低 4 位来表示编码下一个非“0”系数所需要的位数，跟在它后面的是量化 AC 系数的数值。

#### ✓ 6、熵编码

使用熵编码还可以对 DPCM 编码后的直流 DC 系数和 RLE 编码后的交流 AC 系数作进一步的压缩。

在 JPEG 有损压缩算法中，使用霍夫曼编码器来减少熵。使用霍夫曼编码器的理由是可以使用很简单的查表（Lookup Table）方法进行编码。压缩数据符号时，霍夫曼编码器对出现频度比较高的符号分配比较短的代码，而对出现频度较低的符号分配比较长的代码。这种可变长度的霍夫曼码表可以事先进行定义。

#### ✓ 7、组成位数据流

JPEG 编码的最后一个步骤是把各种标记代码和编码后的图像数据组成一帧一帧的数据，这样做的目的是为了便于传输、存储和译码器进行译码，这样的组织的数据通常称为 JPEG 位数据流（JPEG bitstream）。

## TTF 文件分析与使用

TTF 字体是 Windows 等操作系统通用的字体文件，本文从使用角度对 TTF 文件的设计思想、文件结构做了粗浅的剖析，并通过实例演示了该字体文件的使用。

### 1 引言

TTF(TrueType Font)是 Apple 公司和 Microsoft 公司共同推出的字体文件格式，随着 windows 的流行，已经变成最常用的一种字体文件表示方式，在一些特

殊的场合，系统字符集不包含你要用的字体，这时候必须使用自己的字体文件，如甲骨文等古文字处理，但是很多 windows 的使用者对 TTF 字体文件结构、生成和显示方式并不了解，本文从使用角度对 TTF 文件做一个粗浅的剖析，为人们能够掌握这种字体的使用方法。

## 2 TTF 字体设计思想

TTF 是一种向量字体格式，其设计思想是：用一系列点构造字型轮廓，在此基础上用一系列指令调节，使轮廓线变的平滑，从而得到良好的显示效果。

## 3 TTF 字体结构

TTF 文件结构分三部分，如图 2 所示：

文件名(12Bytes)
描述表目录(每个 16Bytes)
描述表数据

图 2

文件头中定义描述表的数目以及文件版本号等信息，描述表目录每项占 16Bytes，按字母顺序列出文件中包含的描述表的信息，包括描述表的名称、在文件中的位置、长度。所以我们根据描述表目录就可以了解 TTF 文件中都包括哪些描述表，这些描述表在文件中什么位置。第三部分描述表数据是每一个描述表的数据区。他们的位置、长度在描述表目录中定义。下面对重要的描述表作简要的描述。

emap 是代码映射表，它给出了从字符代码到文件内部文字序号的映射。字符的编码通过 emap 转换成文件内部文字轮廓描述信息的人口地址，使用何种字符编码由平台 ID(Platform ID)和编码 ID(Encoding ID)决定，如 PlatformID = 3 EncodingID = 1 表示使用 Unicode 编码。glyf 是轮廓数据表，它包括每个文字的轮廓数据信息和指令描述信息。head 表中给出了 TTF 字体的制造时间、最后修改时间等属性信息。name 是名字表，名字表包含版权信息、字体名称、字体系列名称以及风格名称。这些名称信息可以用英、汉两种语言描述。要了解更多的 TTF 文件格式信息请查看相关的资料。

## 4 TTF 字体的创建

现在已有许多 TTF 字体的创建、编辑软件，他们为用户构建 TTF 字体提供了

极大的方便。其中 Font CreatorProgram 是个很好的软件，它提供可视化的操作方式，使用鼠标拖拉就可以编辑、创建字体，可以对版权信息、字体名称、字体版本号等信息进行编辑，它还提供了将图形转化为 TTF 字体的功能。假设我们创建了一个字体文件，字体文件名是 JGW. ttf，字体名称是“甲骨文”，其中包含“你好”这两个字，下面我们对这个文件进行操作。

## 5 TTF 字体在面向对象程序中的引用

windows 给面向对象程序引用 TTF 字体提供了方便的接口，用户可以不必要了解 TTF 文件的内部结构，运用 windows API 透明地引用 TTF 字体。在面向对象的编程中 windows 使用 LOGFONT 结构定义字体文件的属性，LOGFONT 结构原形为：

```
typedef struct tagLOGFONT{ // If
    LONG lfHeight; // 字体高度
    LONG lfWidth; , // 字体平均宽度
    LONG lfEscapement; // 指定移位向量和设备 x 轴之间的角度
    LONG lfOrientation; // 指定每个字符基线和设备 x 轴之间的角度
    LONG lfWeight; // 指定字体的权值，如 400 表示标准体，700 表示黑(粗)
    体
    BYTE lfItalic; // 是否为斜体
    BYTE lfUnderline; // 是否要下画线
    BYTE lfStrikeOut; // 设置为 TRUE，则指定 strikeout 字体
    BYTE lfCharSet; // 使用的字符集
    BYTE lfOutPrecision; // 输出精度
    BYTE lfClipPrecision; // 裁剪精度
    BYTE lfQuality; // 输出质量
    BYTE lfPitchAndFamily; // 指定字体间距
    TCHAR lfFaceName[LF-FACESIZE]; // 字体名称
}LOGFONT;
```

上面注释部分对结构的内容作了简要的解释，详细内容请查看 Windows MSDN 中关于 SDK 部分的说明。此结构是 windows 定义好的，在程序设计中不用自行定义，只需对其直接引用即可，以在屏幕上显示 JGW . ttf 中的“你好”为例，说明在面向对象程序设计中 TrF 字体文件是如何引用的。

下面用面向的对象程序设计语言 VC 给出了实现的程序片段。

## ✓ (1) 定义字体结构变量

```
LOGFONT lfont;
```

## ✓ (2) 给字体结构变量赋值

```
lfont. lfHeight=20; // 高度值  
lfont. lfWidth=20; // 宽度值, 为 0 系统将自动寻找和高度匹配的值。  
lfont. lfEscapement=0; // 缺省值  
lfont. lfOrientation=0; // 缺省值  
lfont. lfWeight=FW-NORMAL; // 标准体  
lfont. lfItalic=FALSE; // 不使用斜体  
lfont. lfUnderline=FALSE; // 不加下画线  
lfont. lfStrikeOut=FALSE; // 缺省值  
lfont. lfCharSet=GB2312 — CHARSET; // 使用简体中文字符集  
lfont. lfOutPrecision: OUT-STROKE. PRECIS; // 缺省值  
lfont. lfClipPrecision=CLIP—STROKE —PRECIS; // 缺省值  
lfont. lfQuality=DRAFT-QUALITY; // 缺省值  
lfont. lfPitchAndFamily = VARIABLE—PITCH I FF—MOD—ERN; // 缺省
```

值

```
strcpy(lfont. lfFaceName, “甲骨文”); // 使用 TTF 字体的名称
```

字体的引用只需对字体高度、名称、是否倾斜、字符集等几个值按需要改变, 其他值使用系统缺省值。关于标准字(如: FW—NORMAL、GB2312 — CHARSET)的含义 MSDN 中有详细说明。由于篇幅原因, 在此不加一一说明。

## ✓ (3) 引用字体实例

```
CClientDC dc(this); // 创建设备对象  
CString str= “你好”; // 定义字符串内容  
CFont myfont; // 创建字体对象  
myfont. CreateFontIndirect(&lfont); // 把字体结构装入字体对象  
dc. SelectObject(&myfont); // 设备对象选择该字体对象  
dc. TextOut(10, 10, str); // 设备对象输出该字符串
```

在 VC 中为了显示字体, 必须创建字体对象, 将上面定义的字体结构 lfont 装入字体对象, 然后设备对象选择此字体对象, 设备对象输出字符串对象。要使显示字符的格式符合(2)中字体结构的描述, 需要所选择的 TTF 字库中含有要显示的字符。也就是说只有 JGW . ttf 字库中含有“你好”这两个字符的映射, “你好”才能正确的显示在屏幕上。