

# Linux 环境搭建

## 本章概要

1. 认识 Linux, 了解 Linux 的相关背景
2. 学会如何使用云服务器
3. 掌握使用远程终端工具 xshell 登陆 Linux 服务器

## 1. Linux 背景介绍

### 发展史

本门课程学习Linux系统编程，你可能要问Linux从哪里来？它是怎么发展的？在这里简要介绍Linux的发展史。要说Linux，还得从UNIX说起。

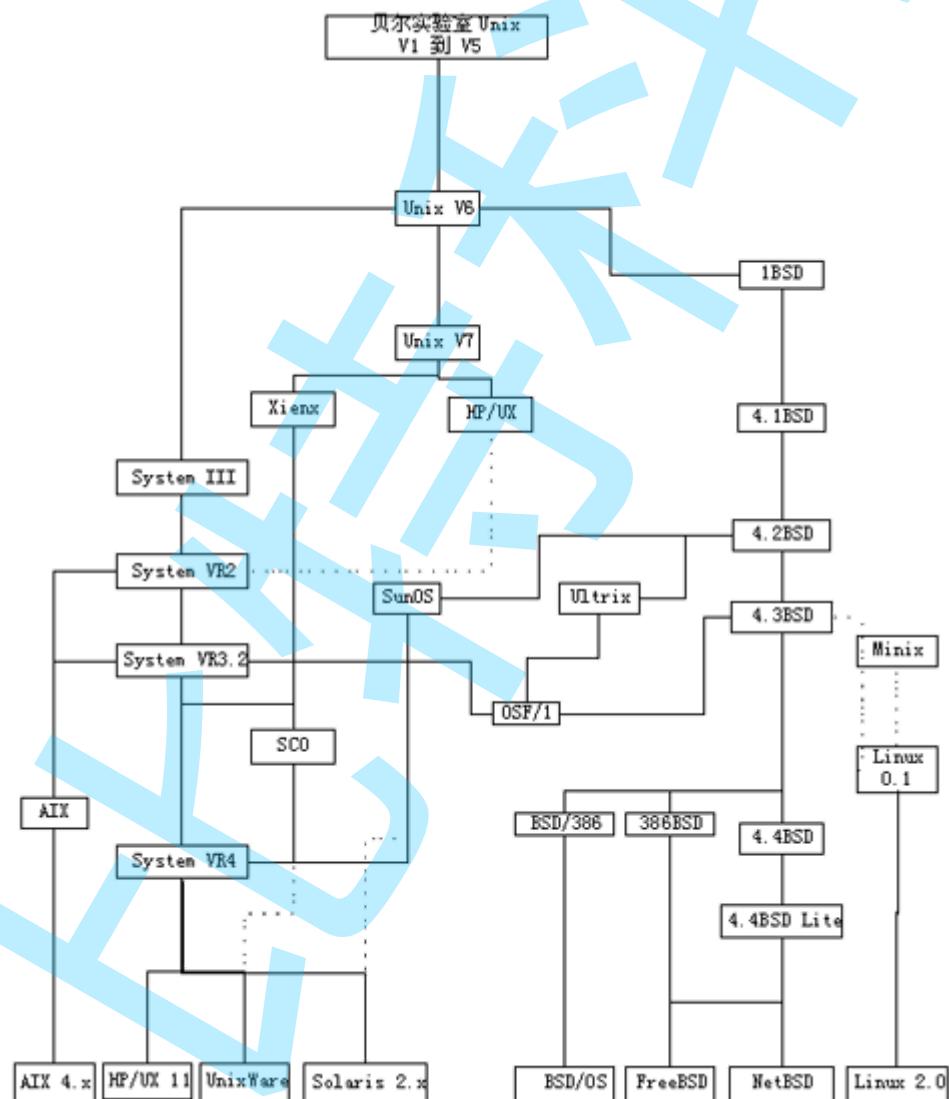
#### UNIX发展的历史

- 1968年，一些来自通用电器公司、贝尔实验室和麻省理工学院的研究人员开发了一个名叫Multics的特殊操作系统。Multics在多任务文件管理和用户连接中综合了许多新概念。
- 1969 - 1970年，AT&T的贝尔实验室研究人员Ken Thompson和Dennis Ritchie，在采用很多Multics特点的基础上开发了UNIX系统。它运行在小型机上，满足了系统对科研环境的要求。从产生开始，UNIX就是一个有价值的、高效的、多用户和多任务的操作系统。
- UNIX从满足个人的设计需求开始，逐步成长为由许多不同开发商所支持的标准软件产品。
- 第一个UNIX版本是免费给许多知名大学的计算机系使用的。
- 1972年，贝尔实验室开始发放商业版本，并且给不同的用户授权使用这个系统，使用者之一是加州大学伯克莱分校的计算机系。伯克莱给系统增加了许多新的特点，后来成为了标准。
- 1975年伯克莱由下属部门BSD发行了自己的UNIX版本。UNIX的BSD版本成为AT&T贝尔实验室版本的主要竞争者，而其它的独立开发出的UNIX版本也开始萌生。
- 1980年微软公司开发了叫做Xenix的UNIX PC版本。AT&T发行了第一个商业版本。名叫System III，后来被成为对商用软件产品良好支持的System V所替代。
- 同时UNIX的BSD版本不断发展，在70年代末期，BSD UNIX成为了国防部的高科技研究机构科研项目的基础。其结果，伯克莱发行了一个叫做BSD Release 4.2的有效版本。
- 它包括了高级的文件管理器和基于TCP/IP网络协议的网络特点。现在TCP/IP被Internet所使用。BSD Release 4.2被许多厂商所采用，例如SUN Microsystem。
- UNIX不同版本的出现导致了UNIX标准的需要，软件开发商不知道他们的程序运行在哪些版本上比较合适。
- 到80年代中期，两个竞争的标准出现了，一个是基于AT&T的UNIX版本，另一个是BSD版本。在今天的书店里你能发现分别适用于这两个版本的不同的UNIX的书，一些是System V，另一些集中在BSD UNIX。
- AT&T建立了一个叫UNIX系统实验室的新组织，它的作用就是综合UNIX的不同版本，集中开发一个标准系统。
- 1991年，UNIX综合实验室综合了System V Release3, SUN OS和Xenix的所有特点，发行了System V Release 4。为了与System V Release 4 竞争，一些其它公司，如IBM和惠普Open Software Foundation (OSF) 去产生自己的UNIX标准版本，继而出现了两个标准商业版本OSF版本和System Release 4。
- 1993年，AT&T把它的UNIX转卖给Novell公司。UNIX系统实验室成为了Novell的UNIX系统小组的一部分。Novell发行了基于System V Release 4的自己的UNIX版本UNIXWare，它可以和Novell公司的Netware系统相联。SUN公司已经把System V Release 4 融进了它的SUN OS，发行了Solaris。两个相互竞争的UNIX使用的图

形用户界面（一个叫Motif，另一个叫Openlook），已经合并为一个新的工作平台标准，叫做通用平台环境（CDE）。

## Linux发展历史

- 1991年10月5日，赫尔辛基大学的一名研究生Linus Benedict Torvalds在一个Usenet新闻组（comp.os.minix）中宣布他编制出了一种类似UNIX的小操作系统，叫Linux。新的操作系统是受到另一个UNIX的小操作系统——Minix的启发，该系统是由一名叫Andrew S Tanenbaum的教师开发的。读者也许猜想所发布的这个系统应该是Linux的0.01版本，实际上不是这样。真正的Linux 0.01版本并没有被发布，原因是0.01版本不实用。Linus仅仅在第一个Linux的FTP站点（ftp://nic.funet.fi）上提供过这个版本的源代码。
- Torvalds于10月5日发布的这个Linux版本被称为0.02版，它能够运行GNU Bourne Again Shell(bash)和GNU的C编译程序（gcc）以及为数不多的其它语言。Torvalds绝对没有想到他设想的一种能够针对高级业余爱好者和黑客们的操作系统已经产生，这就是人们所称的Linux。；
- Linux发布时的版本是0.02，后来又有0.03版，然后又跳到0.10版。因为世界各地越来越多的程序员都开始开发Linux，它已经达到0.95版。这就意味着正是公布1.0版本的时间已经为期不远了。正式的1.0版本是在1994年公布的



## 2. 开源

- Linux是一种自由和开放源代码的类UNIX操作系统，该操作系统的内核由林纳斯托瓦兹在1991年首次发布，之后，在加上用户空间的应用程序之后，就成为了Linux操作系统。严格来讲，Linux只是操作系统内核本身，但通常采用“Linux内核”来表达该意思。而Linux则常用来指基于Linux内核的完整操作系统，它包括GUI组件和许多其他实用工具。
- GNU通用公共许可协议（GNU General Public License，简称GNU GPL或GPL），是一个广泛被使用的自由软件许可协议条款，最初由理查德斯托曼为GNU计划而撰写，GPL给予了计算机程序自由软件的定义，任何基于GPL软件开发衍生的产品在发布时必须采用GPL许可证方式，且必须公开源代码，
- Linux是自由软件和开放源代码软件发展中最著名的例子。只要遵循GNU通用公共许可证，任何个人和机构都可以自由地使用Linux的所有底层源代码，也可以自由地修改和再发布。随着Linux操作系统飞速发展，各种集成在Linux上的开源软件和实用工具也得到了应用和普及，因此，Linux也成为了开源软件的代名词。

### 3. 官网

- [kernel官网](#)

### 4. 企业应用现状

- **Linux在服务器领域的发展**

随着开源软件在世界范围内影响力日益增强，Linux服务器操作系统在整个服务器操作系统市场格局中占据了越来越多的市场份额，已经形成了大规模市场应用的局面。并且保持着快速的增长率。尤其在政府、金融、农业、交通、电信等国家关键领域。此外，考虑到Linux的快速成长性以及国家相关政策的扶持力度，Linux服务器产品一定能够冲击更大的服务器市场。

据权威部门统计，目前Linux在服务器领域已经占据75%的市场份额，同时，Linux在服务器市场的迅速崛起，已经引起全球IT产业的高度关注，并以强劲的势头成为服务器操作系统领域中的中坚力量。

- **Linux在桌面领域的发展**

近年来，特别在国内市场，Linux桌面操作系统的发展趋势非常迅猛。国内如中标麒麟Linux、红旗Linux、深度Linux等系统软件厂商都推出的Linux桌面操作系统，目前已经在政府、企业、OEM等领域得到了广泛应用。另外SUSE、Ubuntu也相继推出了基于Linux的桌面系统，特别是Ubuntu Linux，已经积累了大量社区用户。但是，从系统的整体功能、性能来看，Linux桌面系统与Windows系列相比还有一定的差距，主要表现在系统易用性、系统管理、软硬件兼容性、软件的丰富程度等方面。

- **Linux在移动嵌入式领域的发展**

Linux的低成本、强大的定制功能以及良好的移植性能，使得Linux在嵌入式系统方面也得到广泛应用，目前Linux已广泛应用于手机、平板电脑、路由器、电视和电子游戏机等领域。在移动设备上广泛使用的Android操作系统就是创建在Linux内核之上的。目前，Android已经成为全球最流行的智能手机操作系统，据2015年权威部门最新统计，Android操作系统的全球市场份额已达84.6%。

此外，思科在网络防火墙和路由器也使用了定制的Linux，阿里云也开发了一套基于Linux的操作系统“YunOS”，可用于智能手机、平板电脑和网络电视；常见的数字视频录像机、舞台灯光控制系统等都在逐渐采用定制版本的Linux来实现，而这一切均归功于Linux与开源的力量。

- **Linux在云计算/大数据领域的发展**

互联网产业的迅猛发展，促使云计算、大数据产业的形成并快速发展，云计算、大数据作为一个基于开源软件的平台，Linux占据了核心优势；据Linux基金会的研究，86%的企业已经使用Linux操作系统进行云计算、大数据平台的构建，目前，Linux已开始取代Unix成为最受青睐的云计算、大数据平台操作系统。

### 5. 发行版本

- Debian

Debian运行起来极其稳定，这使得它非常适合用于服务器。debian这款操作系统无疑并不适合新手用户，而是适合系统管理员和高级用户。

- Ubuntu  
Ubuntu是Debian的一款衍生版，也是当今最受欢迎的免费操作系统。Ubuntu侧重于它在这个市场的应用，在服务器、云计算、甚至一些运行Ubuntu Linux的移动设备上很常见。Ubuntu是新手用户肯定爱不释手的一款操作系统。
- 红帽企业级Linux 这是第一款面向商业市场的Linux发行版。它有服务器版本，支持众多处理器架构，包括x86和x86\_64。红帽公司通过课程红帽认证系统管理员/红帽认证工程师（RHCSA/RHCE），对系统管理员进行培训和认证。
- CentOS  
CentOS是一款企业级Linux发行版，它使用红帽企业级Linux中的免费源代码重新构建而成。这款重构版完全去掉了注册商标以及Binary程序包方面一个非常细微的变化。有些人不想支付一大笔钱，又能领略红帽企业级Linux；对他们来说，CentOS值得一试。此外，CentOS的外观和行为似乎与母发行版红帽企业级Linux如出一辙。CentOS使用YUM来管理软件包。
- Fedora  
小巧的Fedora适合那些人：想尝试最先进的技术，等不及程序的稳定版出来。其实，Fedora就是红帽公司的一个测试平台；产品在成为企业级发行版之前，在该平台上进行开发和测试。Fedora是一款非常好的发行版，有庞大的用户论坛，软件库中还有为数不少的软件包。
- Kali Linux  
Kali Linux是Debian的一款衍生版。Kali旨在用于渗透测试。Kali的前身是Backtrack。用于Debian的所有Binary软件包都可以安装到Kali Linux上，而Kali的魅力或威力就来自于此。此外，支持Debian的用户论坛为Kali加分不少。Kali随带许多的渗透测试工具，无论是Wifi、数据库还是其他任何工具，都设计成立马可以使用。Kali使用APT来管理软件包。  
毫无疑问，Kali Linux是一款渗透测试工具，或者是文明黑客（我不想谈论恶意黑客）青睐的操作系统。
- ...



## 2. 搭建 Linux 环境

### Linux 环境的搭建方式

主要有三种

1. 直接安装在物理机上。但是由于Linux桌面使用起来非常不友好，不推荐。
2. 使用虚拟机软件，将Linux搭建在虚拟机上。但是由于当前的虚拟机软件(如VMWare之类的)存在一些bug，会导致环境上出现各种莫名其妙的问题，比较折腾。
3. 使用云服务器，可以直接在腾讯云、阿里云或华为云等服务器厂商处直接购买一个云服务器。

如腾讯云阿里云等为在校学生提供了优惠, 只要通过学生认证, 最低可以 10 块钱一个月. 还是非常划算的.

甚至同学们可以 4 , 5 个人共用一台服务器, 平均下来一个人一个月 2 块钱.

使用云服务器不仅环境搭建简单, 避免折腾, 同时还有一个最大的好处, 部署在云服务器上的项目可以直接被外网访问到, 这个时候就和一个公司发布一个正式的网站没有任何区别. 也就能让我们自己写的程序真的去给别人去使用.

## 购买云服务器

我们以腾讯云为例, 其他的服务器厂商也是类似.

1. 进入官方网站 <https://cloud.tencent.com/act/campus> (直接在百度上搜 "腾讯云校园计划")
2. 登陆网站(可以使用 qq 或者 微信 登陆)

The screenshot shows the Tencent Cloud website's student discount section. At the top, there are several navigation links: '校园优惠套餐' (Student Discount Plan), '云从业者培训及认证' (Cloud Practitioner Training and Certification), '在线学习中心' (Online Learning Center) with a '小程序' (mini-program) badge, '玩转人工智能' (Play with Artificial Intelligence), and '云开发实验室' (Cloud Development Lab). Below this, a large banner for the '校园优惠套餐' (Student Discount Plan) is displayed, featuring a '25岁以下免学生认证' (Under 25 years old免 student certification) badge. The main content area shows a configuration for a '云服务器' (Cloud Server) with the following details:

- CPU: 1核 (1 Core)
- 内存: 2G (2 GB)
- 带宽: 1Mbps (1 Mbps)
- 云硬盘: 50GB (High-performance cloud disk)

On the right side, there are dropdown menus for '活动地域' (Activity Region) set to '广州三区' (Guangzhou Three Areas) and '北京三区' (Beijing Three Areas), '操作系统' (Operating System) set to 'CentOS 7.6 64位', and '购买时长' (Purchase Duration) with options for 1 year, 6 months, 3 months, 2 months, and 1 month. The price is listed as '120元 4398元'. A prominent blue button at the bottom right says '立即购买' (Buy Now).

3. 右侧的操作系统选择 CentOS 7.6 64位. 购买时长根据需要选择(建议1年), 点击立即购买即可. 这个步骤需要实名认证, 否则会提示



点击立即认证, 按照系统提示, 完成实名认证即可(认证速度很快).

4. 购买完成后, 可以在控制台中找到自己买的服务器. 点进去能够看到服务器的 IP 地址.

搜索

中国站 备案 控制台

技术大咖线下与你畅聊  
技术发展趋势

技术沙龙

腾讯云权威认证能力,  
助力事业发展

云认证

携手技术专家共同打造  
开发者生态

TVP

11.11 智惠上云

域名特惠

11.11 智惠上云

11.11 智惠上云

腾讯云产品

通过名称/关键字查找产品 (例如: 云服务器、数据库等)

最近访问

云服务器 私有网络

云产品 使用中 全部

云服务器 1 台

私有网络 1 个

查看更多

使用文档

查看更多

云服务器 云数据库 MySQL

私有网络 内容分发网络

The screenshot shows the Tencent Cloud console interface under the 'Cloud Servers' section. The left sidebar includes options like Overview, Instances, Dedicated Hosts, Placement Groups, Mirrors, Auto Scaling, Cloud Disks, Snapshots, SSH Keys, Security Groups, Elastic Public IPs, Service Migration, and Recycling Bin. The main area displays a list of instances with one entry highlighted by a red box. The highlighted row contains the following information:

ID/Name	Region	Type	Config	IP Address	Charge Mode	Network Mode	Project
ins-qgucsx9d 比特科技教学	北京三区	Standard Type S2	1 vCPU 2GB 1Mbps System Disk: High-performance Cloud Disk Network: Default-VPC	49.233.172.121 (Public) 172.21.0.12 (Internal)	Annual Billing 2020-11-08 17:34:21 Due	Bandwidth Billing Monthly Billing	Default Project

蓝色方框为公网 ip 地址, 稍后我们就会使用这个 ip 登陆服务器.

5. 设置 root 密码: 勾选服务器, 点击重置密码(这个环节可能需要手机短信验证). root 密码建议设置的稍微复杂一些, 否则容易被黑客入侵.

This screenshot shows the same instance list as the previous one, but with a blue box highlighting the 'Reset Password' button in the top navigation bar. The instance table remains the same, showing the single entry for 'ins-qgucsx9d'.

小结:

在这个环节我们最重要的是得到三个信息:

1. 服务器的外网 IP
2. 服务器的管理员账户 (固定为 root)
3. 管理员账户密码(在腾讯云网站上设置的)

通过这三个信息就可以使用 XShell 远程登陆了.

### 3. 使用 XShell 远程登陆到 Linux

#### 关于 Linux 桌面

很多同学的 Linux 启动进入图形化的桌面. 这个东西大家以后就可以忘记了. 以后的工作中 **没有机会** 使用图形界面.

**思考:** 为什么不使用图形界面?

## 下载安装 XShell

XShell 是一个远程终端软件. 下载官网

[https://www.netsarang.com/products/xsh\\_overview.html](https://www.netsarang.com/products/xsh_overview.html)

下载安装的时候选择 "home/school" 则为免费版本.

## 查看 Linux 主机 ip

参考上面的 "购买云服务器" 部分

## 使用 XShell 登陆主机

在 XShell 终端下敲

```
ssh [ip]
```

ip 为刚才看到的 ifconfig 结果.

如果网络畅通, 将会提示输入用户名密码. 输入即可正确登陆

备注: 这里的用户名密码都是在最初购买服务器的时候设置的用户名密码

## XShell 下的复制粘贴

**复制:** `ctrl + insert` (有些同学的 `insert` 需要配合 `fn` 来按)

**粘贴:** `shift + insert`

`ctrl + c / ctrl + v` 是不行的.

# 本节目目标

1. 初始Linux操作系统
2. 初识shell命令，了解若干背景知识。
3. 使用常用Linux命令
4. 了解Linux权限概念与思想，能深度理解“权限”
5. 初步了解Linux 目录结构含义

## Linux背景

### 1. 发展史

本门课程学习Linux系统编程，你可能要问Linux从哪里来？它是怎么发展的？在这里简要介绍Linux的发展史。要说Linux，还得从UNIX说起。

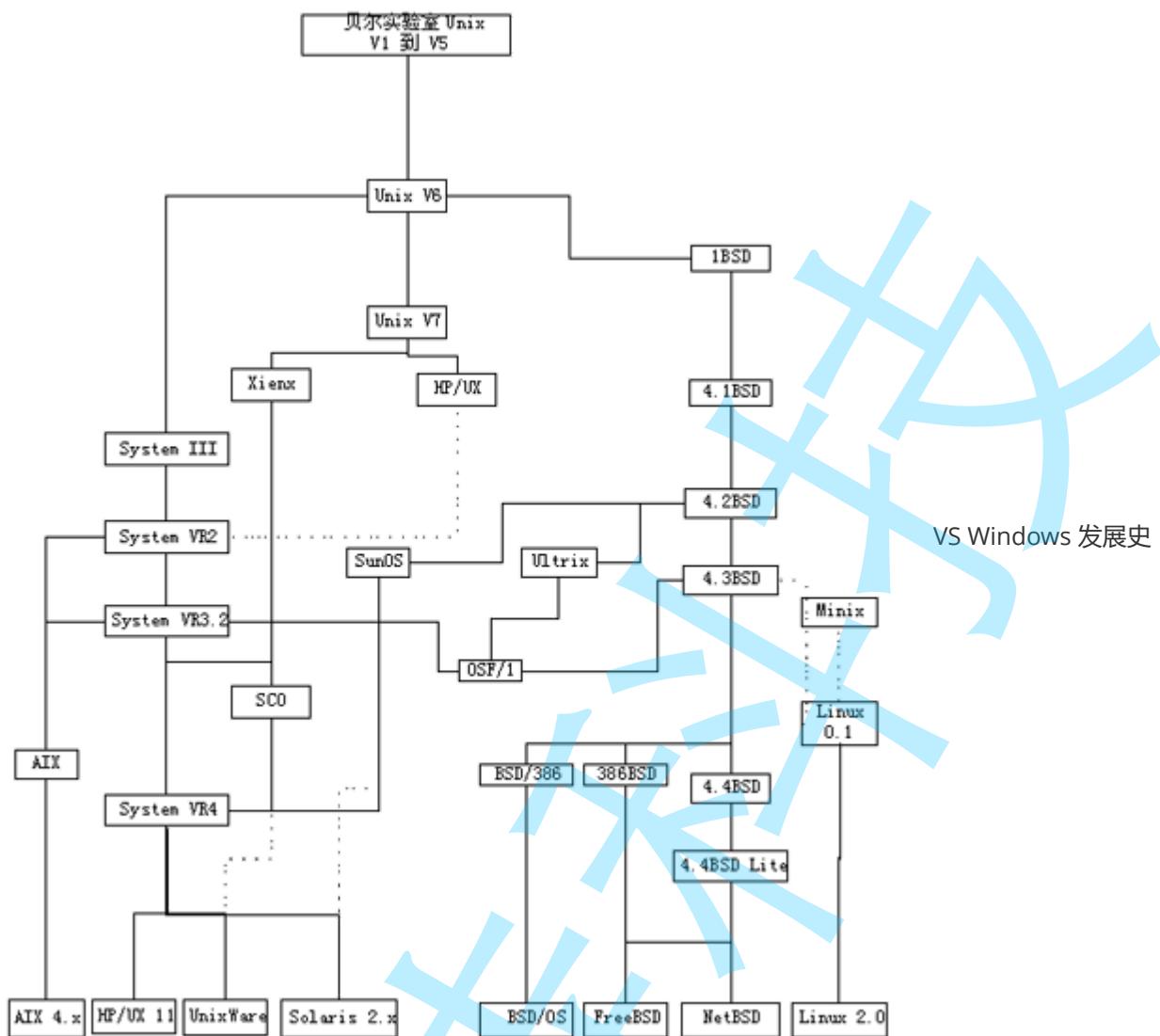
#### UNIX发展的历史

- 1968年，一些来自通用电器公司、贝尔实验室和麻省理工学院的研究人员开发了一个名叫Multics的特殊操作系统。Multics在多任务文件管理和用户连接中综合了许多新概念。
- 1969 - 1970年，AT&T的贝尔实验室研究人员Ken Thompson和Dennis Ritchie，在采用很多Multics特点的基础上开发了UNIX系统。它运行在小型机上，满足了系统对科研环境的要求。从产生开始，UNIX就是一个有价值的、高效的、多用户和多任务的操作系统。
- UNIX从满足个人的设计需求开始，逐步成长为由许多不同开发商所支持的标准软件产品。
- 第一个UNIX版本是免费给许多知名大学的计算机系使用的。
- 1972年，贝尔实验室开始发放商业版本，并且给不同的用户授权使用这个系统，使用者之一是加州大学伯克利分校的计算机系。伯克莱给系统增加了许多新的特点，后来成为了标准。
- 1975年伯克莱由下属部门BSD发行了自己的UNIX版本。UNIX的BSD版本成为AT&T贝尔实验室版本的主要竞争者，而其它的独立开发出的UNIX版本也开始萌生。
- 1980年微软公司开发了叫做Xenix的UNIX PC版本。AT&T发行了第一个商业版本。名叫System III，后来被成为对商用软件产品良好支持的System V所替代。
- 同时UNIX的BSD版本不断发展，在70年代末期，BSD UNIX成为了国防部的高科技研究机构科研项目的基础。其结果，伯克莱发行了一个叫做BSD Release 4.2的有效版本。
- 它包括了高级的文件管理器和基于TCP/IP网络协议的网络特点。现在TCP/IP被Internet所使用。BSD Release 4.2被许多厂商所采用，例如SUN Microsystem。
- UNIX不同版本的出现导致了UNIX标准的需要，软件开发商不知道他们的程序运行在哪些版本上比较合适。
- 到80年代中期，两个竞争的标准出现了，一个是基于AT&T的UNIX版本，另一个是BSD版本。在今天的书店里你能发现分别适用于这两个版本的不同的UNIX的书，一些是System V，另一些集中在BSD UNIX。
- AT&T建立了一个叫UNIX系统实验室的新组织，它的作用就是综合UNIX的不同版本，集中开发一个标准系统。
- 1991年，UNIX综合实验室综合了System V Release3, SUN OS和Xenix的所有特点，发行了System V Release 4。为了与System V Release 4 竞争，一些其它公司，如IBM和惠普Open Software Foundation (OSF) 去产生自己的UNIX标准版本，继而出现了两个标准商业版本OSF版本和System Release 4。
- 1993年，AT&T把它的UNIX转卖给Novell公司。UNIX系统实验室成为了Novell的UNIX系统小组的一部分。Novell发行了基于System V Release 4的自己的UNIX版本UNIXWare，它可以和Novell公司的Netware系统相联。SUN公司已经把System V Release 4 融进了它的SUN OS，发行了Solaris。两个相互竞争的UNIX使用的图形用户界面（一个叫Motif，另一个叫Openlook），已经合并为一个新的工作平台标准，叫做通用平台环境 (CDE) 。

## Linux发展历史

- 1991年10月5日，赫尔辛基大学的一名研究生Linus Benedict Torvalds在一个Usenet新闻组 (comp.os.minix) 中宣布他编制出了一种类似UNIX的小操作系统，叫Linux。新的操作系统是受到另一个UNIX的小操作系统——Minix的启发，该系统是由一名叫Andrew S Tanenbaum的教师开发的。读者也许猜想所发布的这个系统应该是Linux的0.01版本，实际上不是这样。真正的Linux 0.01版本并没有被发布，原因是0.01版本不实用。Linus仅仅在第一个Linux的FTP站点 (<ftp://nic.funet.fi>) 上提供过这个版本的源代码。
- Torvalds于10月5日发布的这个Linux版本被称为0.02版，它能够运行GNU Bourne Again Shell(bash)和GNU的C编译程序 (gcc) 以及为数不多的其它语言。Torvalds绝对没有想到他设想的一种能够针对高级业余爱好者和黑客们的操作系统已经产生，这就是人们所称的Linux。；
- Linux发布时的版本是0.02，后来又有0.03版，然后又跳到0.10版。因为世界各地越来越多的程序员都开始开发Linux，它已经达到0.95版。这就意味着正是公布1.0版本的时间已经为期不远了。正式的1.0版本是在1994年公布的





## 2. 开源

- Linux是一种自由和开放源代码的类UNIX操作系统，该操作系统的内核由林纳斯托瓦兹在1991年首次发布，之后，在加上用户空间的应用程序之后，就成为了Linux操作系统。严格来讲，Linux只是操作系统

内核本身，但通常采用“Linux内核”来表达该意思。而Linux则常用来指基于Linux内核的完整操作系统，它包括GUI组件和许多其他实用工具。

- GNU通用公共许可协议 (GNU General Public License, 简称GNU GPL或GPL)，是一个广泛被使用的自由软件许可协议条款，最初由理查德斯托曼为GNU计划而撰写，GPL给予了计算机程序自由软件的定义，任何基于GPL软件开发衍生的产品在发布时必须采用GPL许可证方式，且必须公开源代码，
- Linux是自由软件和开放源代码软件发展中最著名的例子。只要遵循GNU通用公共许可证，任何个人和机构都可以自由地使用Linux的所有底层源代码，也可以自由地修改和再发布。随着Linux操作系统飞速发展，各种集成在Linux上的开源软件和实用工具也得到了应用和普及，因此，Linux也成为了开源软件的代名词。

### 3. 官网

- [kernel官网](#)

### 4. 企业应用现状

- **Linux在服务器领域的发展**

随着开源软件在世界范围内影响力日益增强，Linux服务器操作系统在整个服务器操作系统市场格局中占据了越来越多的市场份额，已经形成了大规模市场应用的局面。并且保持着快速的增长率。尤其在政府、金融、农业、交通、电信等国家关键领域。此外，考虑到Linux的快速成长性以及国家相关政策的扶持力度，Linux服务器产品一定能够冲击更大的服务器市场。

据权威部门统计，目前Linux在服务器领域已经占据75%的市场份额，同时，Linux在服务器市场的迅速崛起，已经引起全球IT产业的高度关注，并以强劲的势头成为服务器操作系统领域中的中坚力量。

- **Linux在桌面领域的发展**

近年来，特别在国内市场，Linux桌面操作系统的发展趋势非常迅猛。国内如中标麒麟Linux、红旗Linux、深度Linux等系统软件厂商都推出的Linux桌面操作系统，目前已经在全国政府、企业、OEM等领域得到了广泛应用。另外SUSE、Ubuntu也相继推出了基于Linux的桌面系统，特别是Ubuntu Linux，已经积累了大量社区用户。但是，从系统的整体功能、性能来看，Linux桌面系统与Windows系列相比还有一定的差距，主要表现在系统易用性、系统管理、软硬件兼容性、软件的丰富程度等方面。

- **Linux在移动嵌入式领域的发展**

Linux的低成本、强大的定制功能以及良好的移植性能，使得Linux在嵌入式系统方面也得到广泛应用，目前Linux已广泛应用于手机、平板电脑、路由器、电视和电子游戏机等领域。在移动设备上广泛使用的Android操作系统就是创建在Linux内核之上的。目前，Android已经成为全球最流行的智能手机操作系统，据2015年权威部门最新统计，Android操作系统的全球市场份额已达84.6%。

此外，思科在网络防火墙和路由器也使用了定制的Linux，阿里云也开发了一套基于Linux的操作系统“YunOS”，可用于智能手机、平板电脑和网络电视；常见的数字视频录像机、舞台灯光控制系统等都在逐渐采用定制版本的Linux来实现，而这一切均归功于Linux与开源的力量。

- **Linux在云计算/大数据领域的发展**

互联网产业的迅猛发展，促使云计算、大数据产业的形成并快速发展，云计算、大数据作为一个基于开源软件的平台，Linux占据了核心优势；据Linux基金会的研究，86%的企业已经使用Linux操作系统进行云计算、大数据平台的构建，目前，Linux已经开始取代Unix成为最受青睐的云计算、大数据平台操作系统。

### 5. 发行版本

- Debian

Debian运行起来极其稳定，这使得它非常适合用于服务器。 debian这款操作系统无疑并不适合新手用户，而是适合系统管理员和高级用户。

- Ubuntu

Ubuntu是Debian的一款衍生版，也是当今最受欢迎的免费操作系统。Ubuntu侧重于它在这个市场的应用，在服务器、云计算、甚至一些运行Ubuntu Linux的移动设备上很常见。Ubuntu是新手用户肯定爱不释手的一款操作系统。

- 红帽企业级Linux

这是第一款面向商业市场的Linux发行版。它有服务器版本，支持众多处理器架构，包括x86和x86\_64。红帽公司通过课程红帽认证系统管理员/红帽认证工程师（RHCSA/RHCE），对系统管理员进行培训和认证。

- CentOS

CentOS是一款企业级Linux发行版，它使用红帽企业级Linux中的免费源代码重新构建而成。这款重构版完全去掉了注册商标以及Binary程序包方面一个非常细微的变化。有些人不想支付一大笔钱，又能领略红帽企业级Linux；对他们来说，CentOS值得一试。此外，CentOS的外观和行为似乎与母发行版红帽企业级Linux如出一辙。CentOS使用YUM来管理软件包。

- Fedora

小巧的Fedora适合那些人：想尝试最先进的技术，等不及程序的稳定版出来。其实，Fedora就是红帽公司的一个测试平台；产品在成为企业级发行版之前，在该平台上进行开发和测试。Fedora是一款非常好的发行版，有庞大的用户论坛，软件库中还有为数不少的软件包。

- Kali Linux

Kali Linux是Debian的一款衍生版。Kali旨在用于渗透测试。Kali的前身是Backtrack。用于Debian的所有Binary软件包都可以安装到Kali Linux上，而Kali的魅力或威力就来自于此。此外，支持Debian的用户论坛为Kali加分不少。Kali随带许多的渗透测试工具，无论是Wifi、数据库还是其他任何工具，都设计成立马可以使用。Kali使用APT来管理软件包。

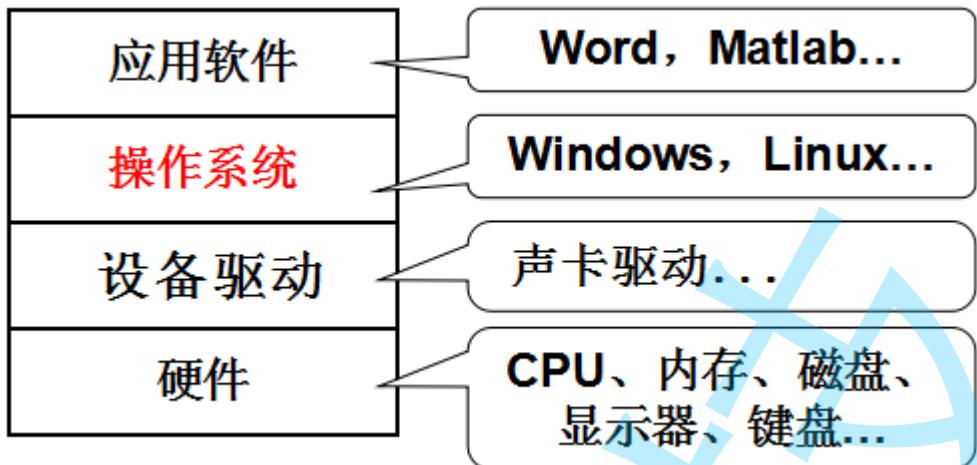
毫无疑问，Kali Linux是一款渗透测试工具，或者是文明黑客（我不想谈论恶意黑客）青睐的操作系统。

- ... ...



## 6. os概念，定位

使计算机更好用！这是操作系统的根本要义！！



## 使用 XShell 远程登录 Linux

### 关于 Linux 桌面

很多同学的 Linux 启动进入图形化的桌面. 这个东西大家以后就可以忘记了. 以后的工作中 **没有机会** 使用图形界面.

**思考:** 为什么不使用图形界面?

### 下载安装 XShell

XShell 是一个远程终端软件. 下载官网

[https://www.netsarang.com/products/xsh\\_overview.html](https://www.netsarang.com/products/xsh_overview.html)

下载安装的时候选择 "home/school" 则为免费版本.

### 查看 Linux 主机 ip

在终端下敲 `ifconfig` 指令, 查看到 ip 地址.

### 使用 XShell 登陆主机

在 XShell 终端下敲

```
ssh [ip]
```

ip 为刚才看到的 ifconfig 结果.

如果网络畅通, 将会提示输入用户名密码. 输入即可正确登陆

### XShell 下的复制粘贴

**复制:** `ctrl + insert` (有些同学的 `insert` 需要配合 `fn` 来按)

**粘贴:** `shift + insert`

`ctrl + c / ctrl + v` 是不行的.

## Linux下基本指令

## 01. ls 指令

语法: ls [选项][目录或文件]

功能: 对于目录, 该命令列出该目录下的所有子目录与文件。对于文件, 将列出文件名以及其他信息。

常用选项:

- -a 列出目录下的所有文件, 包括以 . 开头的隐含文件。
- -d 将目录象文件一样显示, 而不是显示其下的文件。如: ls -d 指定目录
- -i 输出文件的 i 节点的索引信息。如 ls -ai 指定文件
- -k 以 k 字节的形式表示文件的大小。ls -alk 指定文件
- -l 列出文件的详细信息。
- -n 用数字的 UID,GID 代替名称。 (介绍 UID, GID)
- -F 在每个文件名后附上一个字符以说明该文件的类型, "\*"表示可执行的普通文件; "/"表示目录; "@"表示符号链接; "|"表示FIFOs; "="表示套接字(sockets)。 (目录类型识别)
- -r 对目录反向排序。
- -t 以时间排序。
- -s 在文件名后输出该文件的大小。 (大小排序, 如何找到目录下最大的文件)
- -R 列出所有子目录下的文件。(递归)
- -1 一行只输出一个文件。

举例:

```
ls -l
```

## 02. pwd 命令

语法: pwd

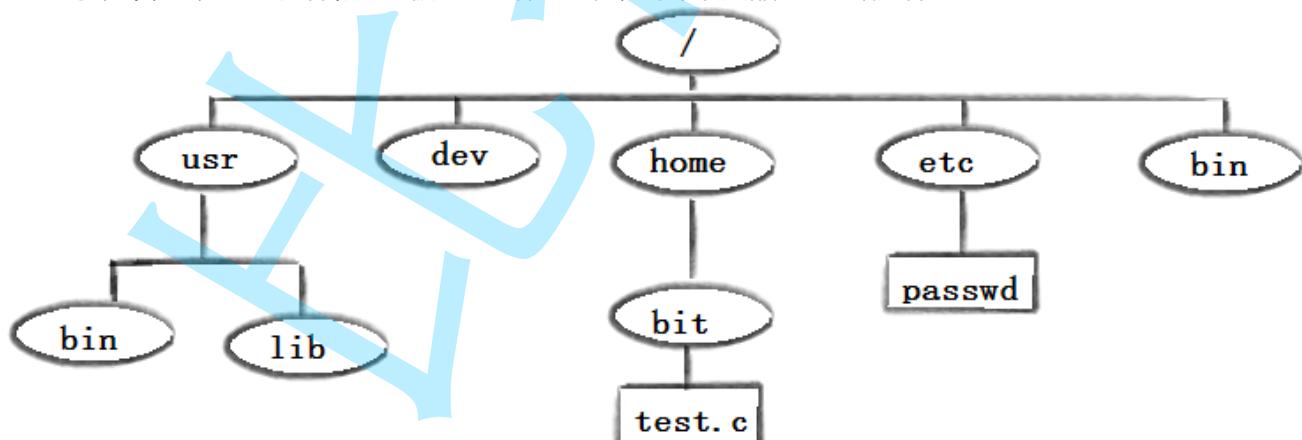
功能: 显示用户当前所在的目录

常用选项:

无

## 03. cd 指令

Linux系统中, 磁盘上的文件和目录被组成一棵目录树, 每个节点都是目录或文件。



语法: cd 目录名

功能: 改变工作目录。将当前工作目录改变到指定的目录下。

举例:

```
cd .. : 返回上级目录  
cd /home/litao/linux/ : 绝对路径  
cd ../day02/ : 相对路径  
cd ~: 进入用户家目  
cd -: 返回最近访问目录
```

## 04. touch指令

**语法:** touch [选项]... 文件...

**功能:** touch命令参数可更改文档或目录的日期时间，包括存取时间和更改时间，或者新建一个不存在的文件。

**常用选项:**

- -a 或--time=atime或--time=access或--time=use只更改存取时间。
- -c 或--no-create 不建立任何文档。
- -d 使用指定的日期时间，而非现在的时间。
- -f 此参数将忽略不予处理，仅负责解决BSD版本touch指令的兼容性问题。
- -m 或--time=mtime或--time=modify 只改变变动时间。
- -r 把指定文档或目录的日期时间，统统设成和参考文档或目录的日期时间相同。
- -t 使用指定的日期时间，而非现在的时间。

## 05.mkdir指令（重要）：

**语法:** mkdir [选项] dirname...

**功能:** 在当前目录下创建一个名为“dirname”的目录

**常用选项:**

- -p, --parents 可以是一个路径名称。此时若路径中的某些目录尚不存在,加上此选项后,系统将自动建立好那些尚不存在的目录,即一次可以建立多个目录;

**举例:**

```
mkdir -p test/test1 : 递归建立多个目录
```

参考文档

## 06.rmdir指令 && rm 指令（重要）：

**rmdir**是一个与**mkdir**相对应的命令。**mkdir**是建立目录，而**rmdir**是删除命令。

**语法:** rmdir [-p][dirName]

**适用对象:** 具有当前目录操作权限的所有使用者

**功能:** 删除空目录

**常用选项:**

- -p 当子目录被删除后如果父目录也变成空目录的话，就连带父目录一起删除。

**rm**命令可以同时删除文件或目录

**语法:** rm [-f-i-r-v][dirName/dir]

**适用对象:** 所有使用者

**功能:** 删除文件或目录

### 常用选项：

- -f 即使文件属性为只读(即写保护), 亦直接删除
- -i 删之前逐一询问确认
- -r 删目录及其下所有文件

## 07.man指令 (重要) :

Linux的命令有很多参数, 我们不可能全记住, 我们可以通过查看联机手册获取帮助。访问Linux手册页的命令是  
man 语法: man [选项] 命令

### 常用选项

- -k 根据关键字搜索联机帮助
- num 只在第num章节找
- -a 将所有章节都显示出来, 比如 man printf 它缺省从第一章开始搜索, 知道就停止, 用a选项, 当按下q退出, 他会继续往后面搜索, 直到所有章节都搜索完毕。

解释一下, 面手册分为8章

1 是普通的命令

2 是系统调用, 如open, write之类的(通过这个, 至少可以很方便的查到调用这个函数, 需要加什么头文件)

3 是库函数, 如printf, fread等是特殊文件, 也就是/dev下的各种设备文件

5 是指文件的格式, 比如passwd, 就会说明这个文件中各个字段的含义

6 是给游戏留的, 由各个游戏自己定义

7 是附件还有一些变量, 比如向environ这种全局变量在这里就有说明

8 是系统管理用的命令, 这些命令只能由root使用, 如ifconfig

## 07.cp指令 (重要) :

语法: cp [选项] 源文件或目录 目标文件或目录

功能: 复制文件或目录

说明: cp指令用于复制文件或目录, 如同时指定两个以上的文件或目录, 且最后的目的地是一个已经存在的目录, 则它会把前面指定的所有文件或目录复制到此目录中。若同时指定多个文件或目录, 而最后的目的地并非一个已存在的目录, 则会出现错误信息

### 常用选项:

- -f 或 --force 强行复制文件或目录, 不论目的文件或目录是否已经存在
- -i 或 --interactive 覆盖文件之前先询问用户
- -r 递归处理, 将指定目录下的文件与子目录一并处理。若源文件或目录的形态, 不属于目录或符号链接, 则一律视为普通文件处理
- -R 或 --recursive 递归处理, 将指定目录下的文件及子目录一并处理

## 08mv指令 (重要) :

**mv**命令是move的缩写，可以用来移动文件或者将文件改名（move (rename) files），是Linux系统下常用的命令，经常用来备份文件或者目录。

**语法:** mv [选项] 源文件或目录 目标文件或目录

**功能:**

- 视mv命令中第二个参数类型的不同（是目标文件还是目标目录），mv命令将文件重命名或将其移至一个新的目录中。
- 当第二个参数类型是文件时，mv命令完成文件重命名，此时，源文件只能有一个（也可以是源目录名），它将所给的源文件或目录重命名为给定的目标文件名。
- 当第二个参数是已存在的目录名称时，源文件或目录参数可以有多个，mv命令将各参数指定的源文件均移至目标目录中。

**常用选项:**

- -f：force 强制的意思，如果目标文件已经存在，不会询问而直接覆盖
- -i：若目标文件(destination) 已经存在时，就会询问是否覆盖！

**举例:**

```
将rm改造成mv
vim ~/.bashrc #修改这个文件
mkdir -p ~/.trash
alias rm=trash
alias ur=undelfile
undelfile() {
    mv -i ~/.trash/\$@ .
}
trash() {
    mv $@ ~/.trash/
}
```

## 09 cat

**语法:** cat [选项][文件]

**功能:** 查看目标文件的内容

**常用选项:**

- -b 对非空输出行编号
- -n 对输出的所有行编号
- -s 不输出多行空行

## 10.more指令

**语法:** more [选项][文件]

**功能:** more命令，功能类似 cat

**常用选项:**

- -n 对输出的所有行编号
- q 退出more

**举例:**

```
[atong@LiWenTong ~]$ ls -l / | more

total 162
drwxr-xr-x 2 root root 4096 Apr 25 05:39 bin
drwxr-xr-x 4 root root 1024 Apr 25 04:11 boot
drwxr-xr-x 9 root root 3820 May  4 23:20 dev
drwxr-xr-x 84 root root 4096 May  5 00:37 etc
```

## 11.less指令 (重要)

- less 工具也是对文件或其它输出进行分页显示的工具，应该说是linux正统查看文件内容的工具，功能极其强大。
- less 的用法比起 more 更加的有弹性。在 more 的时候，我们并没有办法向前面翻，只能往后面看
- 但若使用了 less 时，就可以使用 [pageup][pagedown] 等按键的功能来往前往后翻看文件，更容易用来查看一个文件的内容！
- 除此之外，在 less 里头可以拥有更多的搜索功能，不止可以向下搜，也可以向上搜。

**语法:** less [参数] 文件

**功能:**

less与more类似，但使用less可以随意浏览文件，而more仅能向前移动，却不能向后移动，而且less在查看之前不会加载整个文件。

**选项:**

- -i 忽略搜索时的大小写
- -N 显示每行的行号
- /字符串：向下搜索“字符串”的功能
- ?字符串：向上搜索“字符串”的功能
- n：重复前一个搜索（与 / 或 ? 有关）
- N：反向重复前一个搜索（与 / 或 ? 有关）
- q:quit

## 12.head指令

head 与 tail 就像它的名字一样的浅显易懂，它是用来显示开头或结尾某个数量的文字区块，head 用来显示档案的开头至标准输出中，而 tail 想当然尔就是看档案的结尾。

**语法:** head [参数]... [文件]...

**功能:**

head 用来显示档案的开头至标准输出中，默认head命令打印其相应文件的开头10行。

**选项:**

- -n<行数> 显示的行数

## 13.tail指令

tail 命令从指定点开始将文件写到标准输出。使用tail命令的-f选项可以方便的查阅正在改变的日志文件,tail -f filename会把filename里最尾部的内容显示在屏幕上,并且不但刷新,使你看到最新的文件内容.

**语法:** tail[必要参数][选择参数][文件]

**功能:** 用于显示指定文件末尾内容，不指定文件时，作为输入信息进行处理。常用查看日志文件。

**选项:**

- -f 循环读取
- -n<行数> 显示行数

举例：（简单讲解重定向和管道）

```
有一个文件共有100行内容, 请取出第50行内容<br>
seq 1 100 > test # 生成1到100的序列装入test
方法1 head -n50 test > tmp #将前50行装入临时文件tmp
tail -n1 tmp #得到中建行
方法2 head -n50 test | tail -n1
```

## 14.时间相关的指令

### date显示

date 指定格式显示时间: date +%Y:%m:%d

date 用法: date [OPTION]... [+FORMAT]

1.在显示方面, 使用者可以设定欲显示的格式, 格式设定为一个加号后接数个标记, 其中常用的标记列表如下

- %H : 小时(00..23)
- %M : 分钟(00..59)
- %S : 秒(00..61)
- %X : 相当于 %H:%M:%S
- %d : 日 (01..31)
- %m : 月份 (01..12)
- %Y : 完整年份 (0000..9999)
- %F : 相当于 %Y-%m-%d

### 2.在设定时间方面

- date -s //设置当前时间, 只有root权限才能设置, 其他只能查看。
- date -s 20080523 //设置成20080523, 这样会把具体时间设置成空00:00:00
- date -s 01:01:01 //设置具体时间, 不会对日期做更改
- date -s "01:01:01 2008-05-23" //这样可以设置全部时间
- date -s "01:01:01 20080523" //这样可以设置全部时间
- date -s "2008-05-23 01:01:01" //这样可以设置全部时间
- date -s "20080523 01:01:01" //这样可以设置全部时间

### 3.时间戳

时间->时间戳: date +%s

时间戳->时间: date -d@1508749502

Unix时间戳（英文为Unix epoch, Unix time, POSIX time 或 Unix timestamp）是从1970年1月1日（UTC/GMT的午夜）开始所经过的秒数，不考虑闰秒。

## 15.Cal指令

cal命令可以用来显示公历（阳历）日历。公历是现在国际通用的历法，又称格列历，通称阳历。“阳历”又名“太阳历”，系以地球绕行太阳一周为一年，为西方各国所通用，故又名“西历”。

**命令格式：** cal [参数][月份][年份]

**功能：** 用于查看日历等时间信息，如只有一个参数，则表示年份(1-9999)，如有两个参数，则表示月份和年份

**常用选项：**

- -3 显示系统前一个月，当前月，下一个月的月历
- -j 显示在当年中的第几天（一年日期按天算，从1月1号算起，默认显示当前月在一年中的天数）
- -y 显示当前年份的日历

**举例：**

```
Cal -y 2018
```

## 16.find指令：（灰常重要） -name

- Linux下find命令在目录结构中搜索文件，并执行指定的操作。
- Linux下find命令提供了相当多的查找条件，功能很强大。由于find具有强大的功能，所以它的选项也很多，其中大部分选项都值得我们花时间来了解一下。
- 即使系统中含有网络文件系统(NFS)，find命令在该文件系统中同样有效，只要你具有相应的权限。
- 在运行一个非常消耗资源的find命令时，很多人都倾向于把它放在后台执行，因为遍历一个大的文件系统可能会花费很长的时间(这里是指30G字节以上的文件系统)。

**语法：** find pathname -options

**功能：** 用于在文件树种查找文件，并作出相应的处理（可能访问磁盘）

**常用选项：**

- -name 按照文件名查找文件。

## 17.grep指令

[grep参考文档](#)

**语法：** grep [选项] 搜索字符串 文件

**功能：** 在文件中搜索字符串，将找到的行打印出来

**常用选项：**

- -i：忽略大小写的不同，所以大小写视为相同
- -n：顺便输出行号
- -v：反向选择，亦即显示出没有‘搜索字符串’内容的那一行

## 18.zip/unzip指令：

**语法：** zip 压缩文件.zip 目录或文件

**功能：** 将目录或文件压缩成zip格式

**常用选项：**

- -r 递归处理，将指定目录下的所有文件和子目录一并处理

**举例：**

```
将test2目录压缩: zip test2.zip test2/*
解压到tmp目录: unzip test2.zip -d /tmp
```

## 19.tar指令（重要）：打包/解包，不打开它，直接看内容

**tar [-cxtzjvf] 文件与目录 .... 参数:**

- -c : 建立一个压缩文件的参数指令(create 的意思);
- -x : 解开一个压缩文件的参数指令!
- -t : 查看 tarfile 里面的文件!
- -z : 是否同时具有 gzip 的属性? 亦即是否需要用 gzip 压缩?
- -j : 是否同时具有 bzip2 的属性? 亦即是否需要用 bzip2 压缩?
- -v : 压缩的过程中显示文件! 这个常用, 但不建议用在背景执行过程!
- -f : 使用档名, 请留意, 在 f 之后要立即接档名喔! 不要再加参数!
- -C : 解压到指定目录

**案例:**

范例一: 将整个 /etc 目录下的文件全部打包成为 `/tmp/etc.tar`

```
[root@linux ~]# tar -cvf /tmp/etc.tar /etc<==仅打包, 不压缩!
```

```
[root@linux ~]# tar -zcvf /tmp/etc.tar.gz /etc <==打包后, 以 gzip 压缩
```

```
[root@linux ~]# tar -jcvf /tmp/etc.tar.bz2 /etc <==打包后, 以 bzip2 压缩
```

特别注意, 在参数 f 之后的文件档名是自己取的, 我们习惯上都用 .tar 来作为辨识。

如果加 z 参数, 则以 .tar.gz 或 .tgz 来代表 gzip 压缩过的 tar file ~

如果加 j 参数, 则以 .tar.bz2 来作为附档名啊~

上述指令在执行的时候, 会显示一个警告讯息:

『`tar: Removing leading '/' from member names`』那是關於绝对路径的特殊设定。

范例二: 查阅上述 /tmp/etc.tar.gz 文件内有哪些文件?

```
[root@linux ~]# tar -ztf /tmp/etc.tar.gz
```

由於我们使用 gzip 压缩, 所以要查阅该 tar file 内的文件时, 就得要加上 z 这个参数了! 这很重要的!

范例三: 将 /tmp/etc.tar.gz 文件解压缩在 /usr/local/src 底下

```
[root@linux ~]# cd /usr/local/src
```

```
[root@linux src]# tar -zxf /tmp/etc.tar.gz
```

在预设的情况下, 我们可以将压缩档在任何地方解开的! 以这个范例来说,

我先将工作目录变换到 /usr/local/src 底下, 并且解开 /tmp/etc.tar.gz ,

则解开的目录会在 /usr/local/src/etc 呢! 另外, 如果您进入 /usr/local/src/etc

则会发现, 该目录下的文件属性与 /etc/ 可能会有所不同喔!

范例四: 在 /tmp 底下, 我只想要将 /tmp/etc.tar.gz 内的 etc/passwd 解开而已

```
[root@linux ~]# cd /tmp
```

```
[root@linux tmp]# tar -zxf /tmp/etc.tar.gz etc/passwd
```

我可以透过 tar -ztf 来查阅 tarfile 内的文件名称, 如果单只要一个文件,

就可以透过这个方式来下达! 注意到! etc.tar.gz 内的根目录 / 是被拿掉了!

范例五: 将 /etc/ 内的所有文件备份下来, 并且保存其权限!

```
[root@linux ~]# tar -zvfp /tmp/etc.tar.gz /etc
```

这个 -p 的属性是很重要的, 尤其是当您要保留原本文件的属性时!

范例六: 在 /home 当中, 比 2005/06/01 新的文件才备份

```
[root@linux ~]# tar -N "2005/06/01" -zcvf home.tar.gz /home
```

范例七：我要备份 /home, /etc，但不要 /home/dmotsai  
[root@linux ~]# tar --exclude /home/dmotsai -zcvf myfile.tar.gz /home/\* /etc

范例八：将 /etc/ 打包后直接解开在 /tmp 底下，而不产生文件！

```
[root@linux ~]# cd /tmp  
[root@linux tmp]# tar -cvf - /etc | tar -xvf -  
这个动作有点像是 cp -r /etc /tmp 啦～依旧是很有其用途的！  
要注意的地方在於输出档变成 - 而输入档也变成 -，又有一个 | 存在～  
这分别代表 standard output, standard input 与管线命令啦！
```

## 20.bc指令：

bc命令可以很方便的进行浮点运算

## 21.uname -r指令：

语法：uname [选项]

功能：uname用来获取电脑和操作系统的相关信息。

补充说明：uname可显示linux主机所用的操作系统的版本、硬件的名称等基本信息。

常用选项：

- -a或-all 详细输出所有信息，依次为内核名称，主机名，内核版本号，内核版本，硬件名，处理器类型，硬件平台类型，操作系统名称

## 22.重要的几个热键[Tab],[ctrl]-c, [ctrl]-d

- [Tab]按键---具有『命令补全』和『档案补齐』的功能
- [Ctrl]-c按键---让当前的程序『停掉』
- [Ctrl]-d按键---通常代表着：『键盘输入结束(End Of File, EOF 或 End Of Input)』的意思；另外，他也可以用来取代exit

## 23.关机

语法：shutdown [选项] \*\* 常见选项：\*\*

- -h : 将系统的服务停掉后，立即关机。
- -r : 在将系统的服务停掉之后就重新启动
- -t sec : -t 后面加秒数，亦即『过几秒后关机』的意思

## 以下命令作为扩展：

- 安装和登录命令：login、shutdown、halt、reboot、install、mount、umount、chsh、exit、last；
- 文件处理命令：file、mkdir、grep、dd、find、mv、ls、diff、cat、ln；
- 系统管理相关命令：df、top、free、quota、at、lp、adduser、groupadd、kill、crontab；
- 网络操作命令：ifconfig、ip、ping、netstat、telnet、ftp、route、rlogin、rcp、finger、mail、nslookup；
- 系统安全相关命令：passwd、su、umask、chgrp、chmod、chown、chattr、sudo ps、who；
- 其它命令：tar、unzip、gunzip、unarj、mtools、man、uncompress、uudecode。

# shell命令以及运行原理

Linux严格意义上说的是一个操作系统，我们称之为“核心（kernel）”，但我们一般用户，不能直接使用kernel。而是通过kernel的“外壳”程序，也就是所谓的shell，来与kernel沟通。如何理解？为什么不能直接使用kernel？

从技术角度，Shell的最简单定义：命令行解释器（command Interpreter）主要包含：

- 将使用者的命令翻译给核心（kernel）处理。
- 同时，将核心的处理结果翻译给使用者。

**对比windows GUI**，我们操作windows不是直接操作windows内核，而是通过图形接口，点击，从而完成我们的操作（比如进入D盘的操作，我们通常是双击D盘图标或者运行起来一个应用程序）。

**shell对于Linux**，有相同的作用，主要是对我们的指令进行解析，解析指令给Linux内核。反馈结果在通过内核运行出结果，通过shell解析给用户。

- 帮助理解：如果说你是一个闷骚且害羞的程序员，那shell就像媒婆，操作系统内核就是你们村头漂亮的且有让你心动的MM小花。你看上了小花，但是有不好意思直接表白，那就让你家人找媒婆帮你提亲，所有的事情你都直接跟媒婆沟通，由媒婆转达你的意思给小花，而我们找到媒婆姓王，所以我们叫它王婆，它对应我们常使用的bash。

## Linux权限的概念

Linux下有两种用户：**超级用户（root）**、**普通用户**。

- 超级用户：可以在linux系统下做任何事情，不受限制
- 普通用户：在linux下做有限的事情。
- 超级用户的命令提示符是“#”，普通用户的命令提示符是“\$”。

**命令：**su [用户名]

**功能：**切换用户。

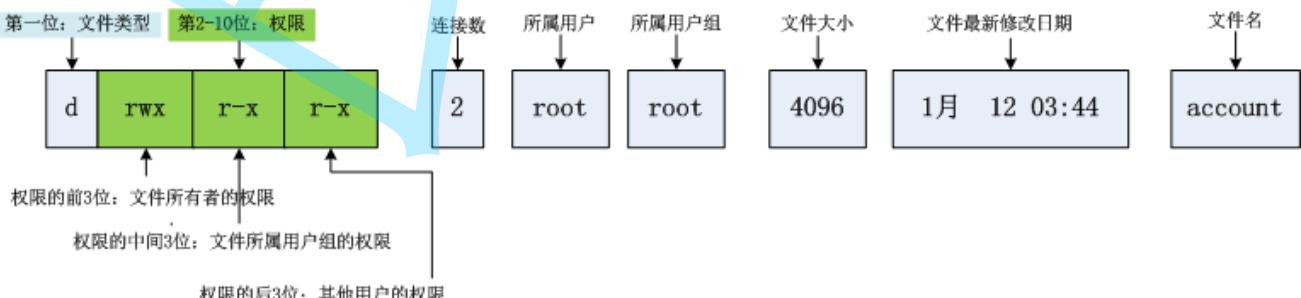
例如，要从root用户切换到普通用户user，则使用 su user。要从普通用户user切换到root用户则使用 su root (root可以省略)，此时系统会提示输入root用户的口令。

## Linux权限管理

### 01.文件访问者的分类（人）

- 文件和文件目录的所有者：u---User（中国平民法律问题）
- 文件和文件目录的所有者所在的组的用户：g---Group（不多说）
- 其它用户：o---Others（外国人）

### 02.文件类型和访问权限（事物属性）



### a) 文件类型

d: 文件夹  
-: 普通文件  
l: 软链接 (类似Windows的快捷方式)  
b: 块设备文件 (例如硬盘、光驱等)  
p: 管道文件  
c: 字符设备文件 (例如屏幕等串口设备)  
s: 套接口文件

### b) 基本权限

- i. 读 (r/4) : Read对文件而言，具有读取文件内容的权限；对目录来说，具有浏览该目录信息的权限
- ii. 写 (w/2) : Write对文件而言，具有修改文件内容的权限；对目录来说具有删除移动目录内文件的权限
- iii. 执行 (x/1) : execute对文件而言，具有执行文件的权限；对目录来说，具有进入目录的权限
- iv. “-”表示不具有该项权限

## 03. 文件权限值的表示方法

### a) 字符表示方法

Linux表示	说明	Linux表示	说明
r--	只读	-w-	仅可写
--x	仅可执行	rw-	可读可写
-wx	可写和可执行	r-x	可读可执行
rwx	可读可写可执行	---	无权限

### b) 8进制数值表示方法

权限符号 (读写执行)	八进制	二进制
r	4	100
w	2	010
x	1	001
rw	6	110
rx	5	101
wx	3	011
rwx	7	111
---	0	000

## 04. 文件访问权限的相关设置方法

## a) chmod

**功能：**设置文件的访问权限

**格式：**chmod [参数] 权限 文件名

**常用选项：**

- R -> 递归修改目录文件的权限
- 说明：只有文件的拥有者和root才可以改变文件的权限

*chmod命令权限值的格式*

① 用户表示符+/-=权限字符

- +:向权限范围增加权限代号所表示的权限
- -:向权限范围取消权限代号所表示的权限
- =:向权限范围赋予权限代号所表示的权限
- 用户符号：
  - u: 拥有者
  - g: 拥有者同组用
  - o: 其它用户
  - a: 所有用户

**实例：**

```
# chmod u+w /home/abc.txt  
# chmod o-x /home/abc.txt
```

## chmod a=x /home/abc.txt

②三位8进制数字

**实例：**

```
# chmod 664 /home/abc.txt  
# chmod 640 /home/abc.txt
```

## b) chown

**功能：**修改文件的拥有者

**格式：**chown [参数] 用户名 文件名

**实例：**

```
# chown user1 f1  
# chown -R user1 filegroup1
```

## c) chgrp

**功能：**修改文件或目录的所属组

**格式：**chgrp [参数] 用户组名 文件名

**常用选项：**-R 递归修改文件或目录的所属组

**实例：**

# chgrp users /abc/f2

## d)umask

功能：

查看或修改文件掩码

新建文件夹默认权限=0666

新建目录默认权限=0777

但实际上你所创建的文件和目录，看到的权限往往不是上面这个值。原因就是创建文件或目录的时候还要受到 umask 的影响。假设默认权限是 mask，则实际创建出来的文件权限是：mask & ~umask

格式：umask 权限值

说明：将现有的存取权限减去权限掩码后，即可产生建立文件时预设权限。超级用户默认掩码值为0022，普通用户默认为0002。

实例：

```
# umask 755  
# umask //查看  
# umask 044//设置
```

## file指令：

功能说明：辨识文件类型。

语法：file [选项] 文件或目录...

常用选项：

- -c 详细显示指令执行过程，便于排错或分析程序执行的情形。
- -z 尝试去解读压缩文件的内容。

## 使用 sudo 分配权限

(1) 修改/etc/sudoers 文件分配文件

```
# chmod 740 /etc/sudoers  
# vi /etc/sudoer
```

格式：接受权限的用户登陆的主机 = (执行命令的用户) 命令

(2) 使用 sudo 调用授权的命令

```
$ sudo -u 用户名 命令
```

实例：

```
$ sudo -u root /usr/sbin/useradd u2
```

## 目录的权限

- 可执行权限：如果目录没有可执行权限，则无法cd到目录中。

- 可读权限: 如果目录没有可读权限, 则无法用ls等命令查看目录中的文件内容.
- 可写权限: 如果目录没有可写权限, 则无法在目录中创建文件, 也无法在目录中删除文件.

于是, 问题来了~~

换句话来讲, 就是只要用户具有目录的写权限, 用户就可以删除目录中的文件, 而不论这个用户是否有这个文件的写权限.

这好像不太科学啊, 我张三创建的一个文件, 凭什么被你李四可以删掉? 我们用下面的过程印证一下.

```
[root@localhost ~]# chmod 0777 /home/  
[root@localhost ~]# ls /home/ -ld  
drwxrwxrwx. 3 root root 4096 9月 19 15:58 /home/  
[root@localhost ~]# touch /home/root.c  
[root@localhost ~]# ls -l /home/  
总用量 4  
-rw-r--r--. 1 root root 0 9月 19 15:58 abc.c  
drwxr-xr-x. 27 litao litao 4096 9月 19 15:53 litao  
-rw-r--r--. 1 root root 0 9月 19 15:59 root.c  
  
[root@localhost ~]# su - litao  
[litao@localhost ~]$ rm /home/root.c #litao可以删除root创建的文件  
rm: 是否删除有写保护的普通空文件 "/home/root.c"? y  
  
[litao@localhost ~]$ exit  
logout
```

为了解决这个不科学的问题, Linux引入了粘滞位的概念.

## 粘滞位

```
[root@localhost ~]# chmod +t /home/ # 加上粘滞位  
[root@localhost ~]# ls -ld /home/  
drwxrwxrwt. 3 root root 4096 9月 19 16:00 /home/  
[root@localhost ~]# su - litao  
[litao@localhost ~]$ rm /home/abc.c #litao不能删除别人的文件  
rm: 是否删除有写保护的普通空文件 "/home/abc.c"? y  
rm: 无法删除"/home/abc.c": 不允许的操作
```

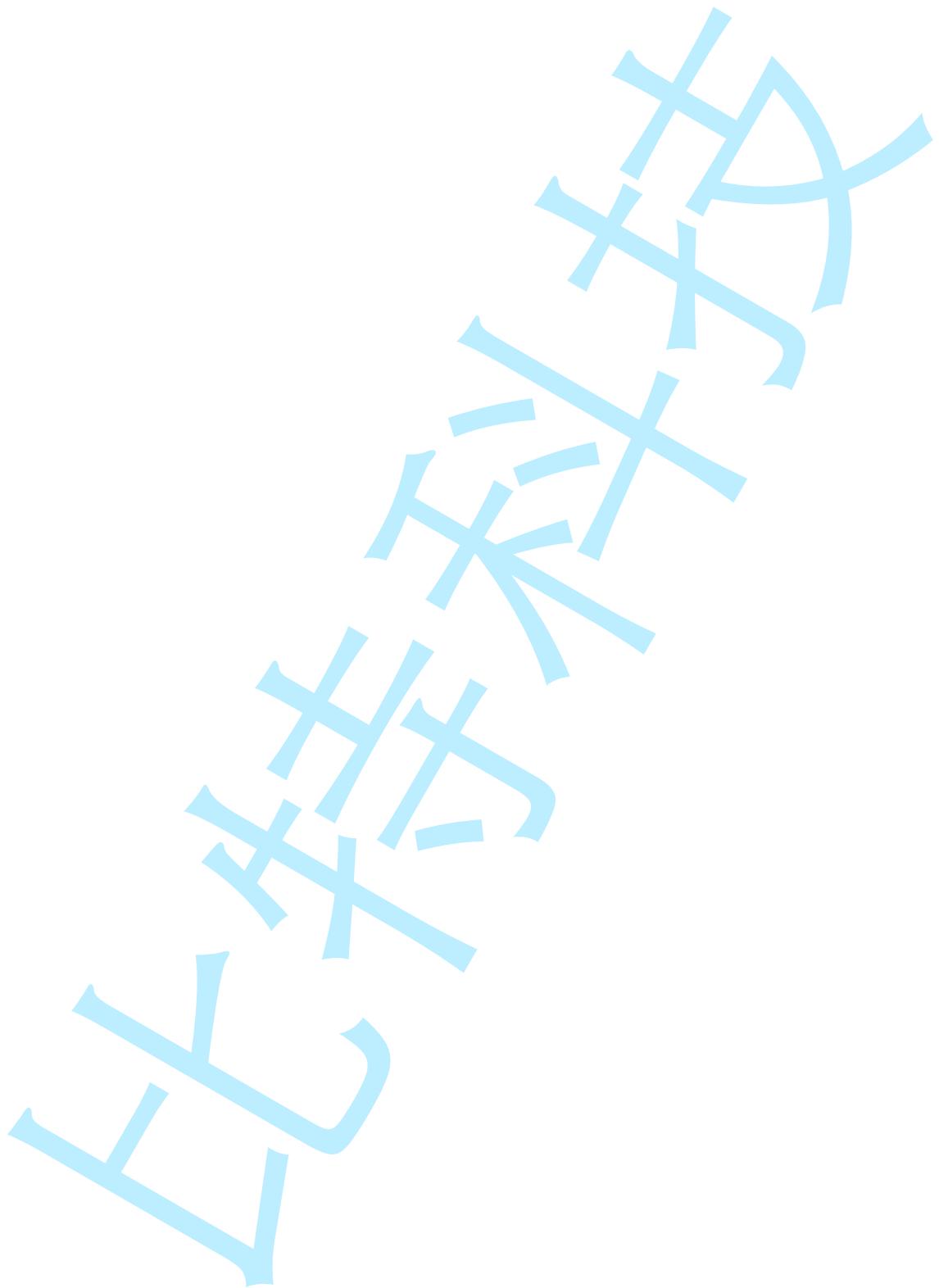
当一个目录被设置为"粘滞位"(用chmod +t), 则该目录下的文件只能由

- 一、超级管理员删除
- 二、该目录的所有者删除
- 三、该文件的所有者删除

## 关于权限的总结

- 目录的可执行权限是表示你可否在目录下执行命令。
- 如果目录没有-x权限, 则无法对目录执行任何命令, 甚至无法cd进入目, 即使目录仍然有-r 读权限 (这个地方很容易犯错, 认为有读权限就可以进入目录读取目录下的文件)
- 而如果目录具有-x权限, 但没有-r权限, 则用户可以执行命令, 可以cd进入目录。但由于没有目录的读权限

- 所以在目录下，即使可以执行ls命令，但仍然没有权限读出目录下的文档。



# 本节目标:

1. 学习yum工具, 进行软件安装
2. 掌握vim编辑器使用, 学会vim的简单配置
3. 掌握gcc/g++编译器的使用, 并了解其过程, 原理
4. 掌握简单gdb使用于调试
5. 掌握简单的Makefile编写, 了解其运行思想
6. 编写自己的第一个Linux 程序: 进度条
7. 学习 git 命令行的简单操作, 能够将代码上传到 Github 上

## Linux 软件包管理器 yum

### 什么是软件包

- 在Linux下安装软件, 一个通常的办法是下载到程序的源代码, 并进行编译, 得到可执行程序.
- 但是这样太麻烦了, 于是有些人把一些常用的软件提前编译好, 做成软件包(可以理解成windows上的安装程序)放在一个服务器上, 通过包管理器可以很方便的获取到这个编译好的软件包, 直接进行安装.
- 软件包和软件包管理器, 就好比 "App" 和 "应用商店" 这样的关系.
- yum(Yellow dog Updater, Modified)是Linux下非常常用的一种包管理器. 主要应用在Fedora, RedHat, Centos等发行版上.

### 关于 rzs2

这个工具用于 windows 机器和远端的 Linux 机器通过 XShell 传输文件.

安装完毕之后可以通过拖拽的方式将文件上传过去.

### 注意事项

关于 yum 的所有操作必须保证主机(虚拟机)网络畅通!!!

可以通过 ping 指令验证

```
ping www.baidu.com
```

### 查看软件包

通过 yum list 命令可以罗列出当前一共有哪些软件包. 由于包的数目可能非常之多, 这里我们需要使用 grep 命令只筛选出我们关注的包. 例如:

```
yum list | grep lrzs2
```

结果如下:

```
lrzs2.x86_64
```

```
0.12.20-36.el7
```

```
@base
```

### 注意事项:

- 软件包名称: 主版本号.次版本号.源程序发行号-软件包的发行号.主机平台.cpu架构.
- "x86\_64" 后缀表示64位系统的安装包, "i686" 后缀表示32位系统安装包. 选择包时要和系统匹配.
- "el7" 表示操作系统发行版的版本. "el7" 表示的是 centos7/redhat7. "el6" 表示 centos6/redhat6.
- 最后一列, base 表示的是 "软件源" 的名称, 类似于 "小米应用商店", "华为应用商店" 这样的概念.

## 如何安装软件

通过 yum, 我们可以通过很简单的一条命令完成 gcc 的安装.

```
sudo yum install lrzs
```

yum 会自动找到都有哪些软件包需要下载, 这时候敲 "y" 确认安装.

出现 "complete" 字样, 说明安装完成.

### 注意事项:

- 安装软件时由于需要向系统目录中写入内容, 一般需要 sudo 或者切到 root 账户下才能完成.
- yum安装软件只能一个装完了再装另一个. 正在yum安装一个软件的过程中, 如果再尝试用yum安装另外一个软件, yum会报错.
- 如果 yum 报错, 请自行百度.

## 如何卸载软件

仍然是一条命令:

```
sudo yum remove lrzs
```

## Linux开发工具

- IDE例子

## vi / vim 键盘图



## Linux编辑器-vim使用

vi/vim的区别简单点来说，它们都是多模式编辑器，不同的是vim是vi的升级版本，它不仅兼容vi的所有指令，而且还有一些新的特性在里面。例如语法加亮，可视化操作不仅可以在终端运行，也可以运行于x window、 mac os、 windows。我们课堂上，统一按照vim来进行讲解。





## 1. vim的基本概念

课堂上我们讲解vim的三种模式(其实有好多模式，目前掌握这3种即可),分别是命令模式 (command mode) 、插入模式 (Insert mode) 和底行模式 (last line mode) , 各模式的功能区分如下:

- 正常/普通/命令模式(Normal mode)

控制屏幕光标的移动，字符、字或行的删除，移动复制某区段及进入Insert mode下，或者到 last line mode

- 插入模式(Insert mode)

只有在Insert mode下，才可以做文字输入，按「ESC」键可回到命令行模式。该模式是我们后面用的最频繁的编辑模式。

- 末行模式(last line mode)

文件保存或退出，也可以进行文件替换，找字符串，列出行号等操作。在命令模式下，`shift+:`即可进入该模式。要查看你的所有模式：打开vim，底行模式直接输入

`:help vim-modes`

我这里一共有12种模式:six BASIC modes和six ADDITIONAL modes.

## 2. vim的基本操作

- 进入vim,在系统提示符号输入vim及文件名称后，就进入vim全屏幕编辑画面:

`$ vim test.c`

不过有一点要特别注意，就是你进入vim之后，是处于[正常模式]，你要切换到[插入模式]才能够输入文字。

- [正常模式]切换至[插入模式]

- 输入a
- 输入i
- 输入o

- [插入模式]切换至[正常模式]

- 目前处于[插入模式]，就只能一直输入文字，如果发现输错了字,想用光标键往回移动，将该字删除，可以先按一下「ESC」键转到[正常模式]再删除文字。当然，也可以直接删除。

- [正常模式]切换至[末行模式]

- 「`shift + ;`」, 其实就是输入「`:`」

- 退出vim及保存文件,在[正常模式]下，按一下「`:`」冒号键进入「Last line mode」,例如:

- `:w` (保存当前文件)

- :wq (输入「wq」,存盘并退出vim)
- :q! (输入q!,不存盘强制退出vim)

### 3. vim正常模式命令集

- 插入模式
  - 按「i」切换进入插入模式「insert mode」，按“i”进入插入模式后是从光标当前位置开始输入文件；
  - 按「a」进入插入模式后，是从目前光标所在位置的下一个位置开始输入文字；
  - 按「o」进入插入模式后，是插入新的一行，从行首开始输入文字。
- 从插入模式切换为命令模式
  - 按「ESC」键。
- 移动光标
  - vim可以直接用键盘上的光标来上下左右移动，但正规的vim是用小写英文字母「h」、「j」、「k」、「l」，分别控制光标左、下、上、右移一格
  - 按「G」：移动到文章的最后
  - 按「\$」：移动到光标所在行的“行尾”
  - 按「^」：移动到光标所在行的“行首”
  - 按「w」：光标跳到下个字的开头
  - 按「e」：光标跳到下个字的字尾
  - 按「b」：光标回到上个字的开头
  - 按「#l」：光标移到该行的第#个位置，如：5l,56l
  - 按 [gg]：进入到文本开始
  - 按 [shift + g]：进入到文本末端
  - 按「ctrl」+「b」：屏幕往“后”移动一页
  - 按「ctrl」+「f」：屏幕往“前”移动一页
  - 按「ctrl」+「u」：屏幕往“后”移动半页
  - 按「ctrl」+「d」：屏幕往“前”移动半页
- 删除文字
  - 「x」：每按一次，删除光标所在位置的一个字符
  - 「#x」：例如，「6x」表示删除光标所在位置的“后面（包含自己在内）”6个字符
  - 「X」：大写的X，每按一次，删除光标所在位置的“前面”一个字符
  - 「#X」：例如，「20X」表示删除光标所在位置的“前面”20个字符
  - 「dd」：删除光标所在行
  - 「#dd」：从光标所在行开始删除#行
- 复制
  - 「yw」：将光标所在之处到字尾的字符复制到缓冲区中。
  - 「#yw」：复制#个字到缓冲区
  - 「yy」：复制光标所在行到缓冲区。
  - 「#yy」：例如，「6yy」表示拷贝从光标所在的该行“往下数”6行文字。
  - 「p」：将缓冲区内的字符贴到光标所在位置。注意：所有与“y”有关的复制命令都必须与“p”配合才能完成复制与粘贴功能。
- 替换
  - 「r」：替换光标所在处的字符。

- 「R」：替换光标所到之处的字符，直到按下「ESC」键为止。
- 撤销上一次操作
  - 「u」：如果您误执行一个命令，可以马上按下「u」，回到上一个操作。按多次“u”可以执行多次恢复。
  - 「ctrl + r」：撤销的恢复
- 更改
  - 「cw」：更改光标所在处的字到字尾处
  - 「c#w」：例如，「c3w」表示更改3个字
- 跳至指定的行
  - 「ctrl」+「g」列出光标所在行的行号。
  - 「#G」：例如，「15G」，表示移动光标至文章的第15行行首。

## 4. vim末行模式命令集

在使用末行模式之前，请记住先按「ESC」键确定您已经处于正常模式，再按「:」冒号即可进入末行模式。

- 列出行号
  - 「set nu」：输入「set nu」后，会在文件中的每一行前面列出行号。
- 跳到文件中的某一行
  - 「#」：「#」号表示一个数字，在冒号后输入一个数字，再按回车键就会跳到该行了，如输入数字15，再回车，就会跳到文章的第15行。
- 查找字符
  - 「/关键字」：先按「/」键，再输入您想寻找的字符，如果第一次找的关键字不是您想要的，可以一直按「n」会往后寻找到您要的关键字为止。
  - 「?关键字」：先按「?」键，再输入您想寻找的字符，如果第一次找的关键字不是您想要的，可以一直按「n」会往前寻找到您要的关键字为止。
  - 问题： / 和 ? 查找有和区别？操作实验一下
- 保存文件
  - 「w」：在冒号输入字母「w」就可以将文件保存起来
- 离开vim
  - 「q」：按「q」就是退出，如果无法离开vim，可以在「q」后跟一个「!」强制离开vim。
  - 「wq」：一般建议离开时，搭配「w」一起使用，这样在退出的时候还可以保存文件。

## 5. vim操作总结

- 三种模式
  - 正常模式
  - 插入模式
  - 底行模式
- 我们一共有12种总模式，大家下来可以研究一下

- vim操作
  - 打开,关闭,查看,查询,插入,删除,替换,撤销,复制等等操作。
- 练习:当堂口头模式切换练习

## 6. 简单vim配置

### 配置文件的位置

- 在目录 /etc/ 下面,有个名为vimrc的文件,这是系统中公共的vim配置文件,对所有用户都有效。
- 而在每个用户的主目录下,都可以自己建立私有的配置文件,命名为: ".vimrc"。例如, /root目录下,通常已经存在一个.vimrc文件,如果不存在,则创建之。
- 切换用户成为自己执行 `su`,进入自己的主工作目录,执行 `cd ~`
- 打开自己目录下的.vimrc文件,执行 `vim .vimrc`

### 常用配置选项,用来测试

- 设置语法高亮: `syntax on`
- 显示行号: `set nu`
- 设置缩进的空格数为4: `set shiftwidth=4`

### 使用插件

要配置好看的vim,原生的配置可能功能不全,可以选择安装插件来完善配置,保证用户是你要配置的用户,接下来:

- 安装TagList插件,下载taglist\_xx.zip,解压完成,将解压出来的doc的内容放到~/.vim/doc,将解压出来的plugin下的内容拷贝到~/.vim/plugin
- 在~/.vimrc中添加: `let Tlist_Show_One_File=1 let Tlist_Exit_OnlyWindow=1 let Tlist_Use_Right_Window=1`
- 安装文件浏览器和窗口管理器插件: WinManager
- 下载winmanager.zip, 2.X版本以上的
- 解压winmanager.zip, 将解压出来的doc的内容放到~/.vim/doc,将解压出来的plugin下的内容拷贝到~/.vim/plugin
- 在~/.vimrc中添加 `let g:winManagerWindowLayout='FileExplorer|TagList' nmap wm :WMToggle<cr>`
- 然后重启vim,打开~/XXX.c或~/XXX.cpp,在normal状态下输入"wm",你将看到上图的效果。

更具体移步: [点我](#),其他手册,请执行 `vimtutor` 命令。

```

3 * dandan@ ~/home/dandan
4 *= ~/home/dandan/test/
5 ..
6 include/
7 src/
8 test.c
9 test.h

File Listl 5,1 Bot:
" Press <F1> to display
- test.c (/home/dandan/test)
  - Function
    - fun
    - main

1 #include <stdio.h>
2
3 int fun(int a, int b)
4 {
5     int c=0;
6     return 0;
7 }
8
9
10 int main()
11 {
12     int val_1 = 10;
13     int val_2 = 20;
14     fun(val_1, val_2);
15     return 0;
16 }

Tag_List__ 3,1 All test.c [R0] Z,0-1 all
WMToggle

```

# 参考资料

[Vim从入门到牛逼\(vim from zero to hero\)](#)

## Linux编译器-gcc/g++使用

### 1. 背景知识

1. 预处理 (进行宏替换)
2. 编译 (生成汇编)
3. 汇编 (生成机器可识别代码)
4. 连接 (生成可执行文件或库文件)

### 2. gcc如何完成

格式 `gcc [选项] 要编译的文件 [选项] [目标文件]`

#### 预处理(进行宏替换)

- 预处理功能主要包括宏定义,文件包含,条件编译,去注释等。
- 预处理指令是以#号开头的代码行。
- 实例: `gcc -E hello.c -o hello.i`
- 选项“-E”,该选项的作用是让 gcc 在预处理结束后停止编译过程。
- 选项“-o”是指目标文件,“.i”文件为已经过预处理的C原始程序。

#### 编译 (生成汇编)

- 在这个阶段中,gcc 首先要检查代码的规范性、是否有语法错误等,以确定代码的实际要做的工作,在检查无误后,gcc 把代码翻译成汇编语言。
- 用户可以使用“-S”选项来进行查看,该选项只进行编译而不进行汇编,生成汇编代码。
- 实例: `gcc -S hello.i -o hello.s`

#### 汇编 (生成机器可识别代码)

- 汇编阶段是把编译阶段生成的“.s”文件转成目标文件
- 读者在此可使用选项“-c”就可看到汇编代码已转化为“.o”的二进制目标代码了
- 实例: `gcc -c hello.s -o hello.o`

#### 连接 (生成可执行文件或库文件)

- 在成功编译之后,就进入了链接阶段。
- 实例: `gcc hello.o -o hello`

### 在这里涉及到一个重要的概念:函数库

- 我们的C程序中, 并没有定义“printf”的函数实现,且在预编译中包含的“stdio.h”中也只有该函数的声明,而没有定义函数的实现,那么,是在哪里实“printf”函数的呢?
- 最后的答案是:系统把这些函数实现都被放到名为 `libc.so.6` 的库文件中去了,在没有特别指定时,gcc 会到系统默认的搜索路径“`/usr/lib`”下进行查找,也就是链接到 `libc.so.6` 库函数中去,这样就能实现函数“printf”了,而这也就是链接的作用

函数库一般分为静态库和动态库两种。

- 静态库是指编译链接时,把库文件的代码全部加入到可执行文件中,因此生成的文件比较大,但在运行时也就不再需要库文件了。其后缀名一般为“.a”
- 动态库与之相反,在编译链接时并没有把库文件的代码加入到可执行文件中,而是在程序执行时由运行时链接文件加载库,这样可以节省系统的开销。动态库一般后缀名为“.so”,如前面所述的 libc.so.6 就是动态库。gcc 在编译时默认使用动态库。完成了链接之后,gcc 就可以生成可执行文件,如下所示。`gcc hello.o -o hello`
- gcc默认生成的二进制程序, 是动态链接的, 这点可以通过 `file` 命令验证。

## gcc选项

- `-E` 只激活预处理,这个不生成文件,你需要把它重定向到一个输出文件里面
- `-S` 编译到汇编语言不进行汇编和链接
- `-c` 编译到目标代码
- `-o` 文件输出到文件
- `-static` 此选项对生成的文件采用静态链接
- `-g` 生成调试信息。GNU 调试器可利用该信息。
- `-shared` 此选项将尽量使用动态库, 所以生成文件比较小, 但是需要系统由动态库.
- `-O0`
- `-O1`
- `-O2`
- `-O3` 编译器的优化选项的4个级别, `-O0`表示没有优化,`-O1`为缺省值, `-O3`优化级别最高
- `-w` 不生成任何警告信息。
- `-Wall` 生成所有警告信息。

## gcc选项记忆

- esc,iso例子

# Linux调试器-gdb使用

## 1. 背景

- 程序的发布方式有两种, debug模式和release模式
- Linux gcc/g++出来的二进制程序, 默认是release模式
- 要使用gdb调试, 必须在源代码生成二进制程序的时候, 加上 `-g` 选项

## 2. 开始使用

`gdb binFile` 退出: `ctrl + d` 或 `quit` 调试命令:

- `list / l 行号`: 显示binFile源代码, 接着上次的位置往下列, 每次列10行。
- `list / l 函数名`: 列出某个函数的源代码。
- `r或run`: 运行程序。
- `n 或 next`: 单条执行。
- `s或step`: 进入函数调用
- `break(b) 行号`: 在某一行设置断点
- `break 函数名`: 在某个函数开头设置断点
- `info break`: 查看断点信息。
- `finish`: 执行到当前函数返回, 然后挺下来等待命令
- `print(p)`: 打印表达式的值, 通过表达式可以修改变量的值或者调用函数

- p 变量: 打印变量值。
- set var: 修改变量的值
- continue(或c): 从当前位置开始连续而非单步执行程序
- run(或r): 从开始连续而非单步执行程序
- delete breakpoints: 删除所有断点
- delete breakpoints n: 删除序号为n的断点
- disable breakpoints: 禁用断点
- enable breakpoints: 启用断点
- info(或i) breakpoints: 参看当前设置了哪些断点
- display 变量名: 跟踪查看一个变量, 每次停下来都显示它的值
- undisplay: 取消对先前设置的那些变量的跟踪
- until X行号: 跳至X行
- breaktrace(或bt): 查看各级函数调用及参数
- info (i) locals: 查看当前栈帧局部变量的值
- quit: 退出gdb

### 3. 理解

- 和windows IDE对应例子

## Linux项目自动化构建工具-make/Makefile

### 背景

- 会不会写makefile, 从一个侧面说明了一个人是否具备完成大型工程的能力
- 一个工程中的源文件不计数, 其按类型、功能、模块分别放在若干个目录中, makefile定义了一系列的规则来指定, 哪些文件需要先编译, 哪些文件需要后编译, 哪些文件需要重新编译, 甚至于进行更复杂的功能操作
- makefile带来的好处就是——“自动化编译”, 一旦写好, 只需要一个make命令, 整个工程完全自动编译, 极大的提高了软件开发的效率。
- make是一个命令工具, 是一个解释makefile中指令的命令工具, 一般来说, 大多数的IDE都有这个命令, 比如: Delphi的make, Visual C++的nmake, Linux下GNU的make。可见, makefile都成为了一种在工程方面的编译方法。
- make是一条命令, makefile是一个文件, 两个搭配使用, 完成项目自动化构建。

### 理解

- 依赖例子

### 实例代码

C代码

```
#include <stdio.h>

int main()
{
    printf("hello Makefile!\n");
    return 0;
}
```

Makefile文件 hello:hello.o gcc hello.o -o hello hello.o:hello.s gcc -c hello.s -o hello.o hello.s:hello.i gcc -S hello.i -o hello.s hello.i:hello.c gcc -E hello.c -o hello.i

```
.PHONY:clean
clean:
    rm -f hello.i hello.s hello.o hello
```

## 依赖关系

- 上面的文件 `hello`, 它依赖 `hello.o`
- `hello.o`, 它依赖 `hello.s`
- `hello.s`, 它依赖 `hello.i`
- `hello.i`, 它依赖 `hello.c`

## 依赖方法

- `gcc hello.* -option hello.*`, 就是与之对应的依赖关系

## 原理

- make是如何工作的,在默认的方式下,也就是我们只输入make命令。那么,
  - make会在当前目录下找名字叫“Makefile”或“makefile”的文件。
  - 如果找到,它会找文件中的第一个目标文件 (target), 在上面的例子中,他会找到“hello”这个文件,并把这个文件作为最终的目标文件。
  - 如果hello文件不存在,或是hello所依赖的后面的hello.o文件的文件修改时间要比hello这个文件新(可以用`touch`测试),那么,他就会执行后面所定义的命令来生成hello这个文件。
  - 如果hello所依赖的hello.o文件不存在,那么make会在当前文件中找目标为hello.o文件的依赖性,如果找到则再根据那个规则生成hello.o文件。(这有点像一个堆栈的过程)
  - 当然,你的C文件和H文件是存在的啦,于是make会生成hello.o文件,然后再用hello.o文件声明make的终极任务,也就是执行文件hello了。
  - 这就是整个make的依赖性,make会一层又一层地去找文件的依赖关系,直到最终编译出第一个目标文件。
  - 在找寻的过程中,如果出现错误,比如最后被依赖的文件找不到,那么make就会直接退出,并报错,而对于所定义的命令的错误,或是编译不成功,make根本不理。
  - make只管文件的依赖性,即,如果在我找了依赖关系之后,冒号后面的文件还是不在,那么对不起,我就不工作啦。

## 项目清理

- 工程是需要被清理的

- 像clean这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要make执行。即命令——“make clean”，以此来清除所有的目标文件，以便重编译。
- 但是一般我们这种clean的目标文件，我们将它设置为伪目标，用`.PHONY`修饰，伪目标的特性是，总是被执行的。
- 可以将我们的`hello`目标文件声明成伪目标，测试一下。

## Linux第一个小程序 - 进度条

### \r&&\n

- 回车概念
- 换行概念
- 老式打字机的例子

### 行缓冲区概念

什么现象？

```
#include <stdio.h>

int main()
{
    printf("hello Makefile!\n");
    sleep(3);
    return 0;
}
```

什么现象？？

```
#include <stdio.h>

int main()
{
    printf("hello Makefile!");
    sleep(3);
    return 0;
}
```

什么现象？？？

```
#include <stdio.h>

int main()
{
    printf("hello Makefile!");
    fflush(stdout);
    sleep(3);
    return 0;
}
```

## 进度条代码

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

int main()
{
    int i = 0;
    char bar[102];
    memset(bar, 0 ,sizeof(bar));
    const char *lable="/-\\";;
    while(i <= 100 ){
        printf("[%-100s][%d%%][%c]\r", bar, i, lable[i%4]);
        fflush(stdout);
        bar[i++] = '#';
        usleep(10000);
    }
    printf("\n");
    return 0;
}
```

```
[hb@MiWiFi-R1CL-srv test]$ ./hello
[########################################] [17%] [/]
```

## 使用 git 命令行

### 安装 git

```
yum install git
```

### 在 Github 创建项目

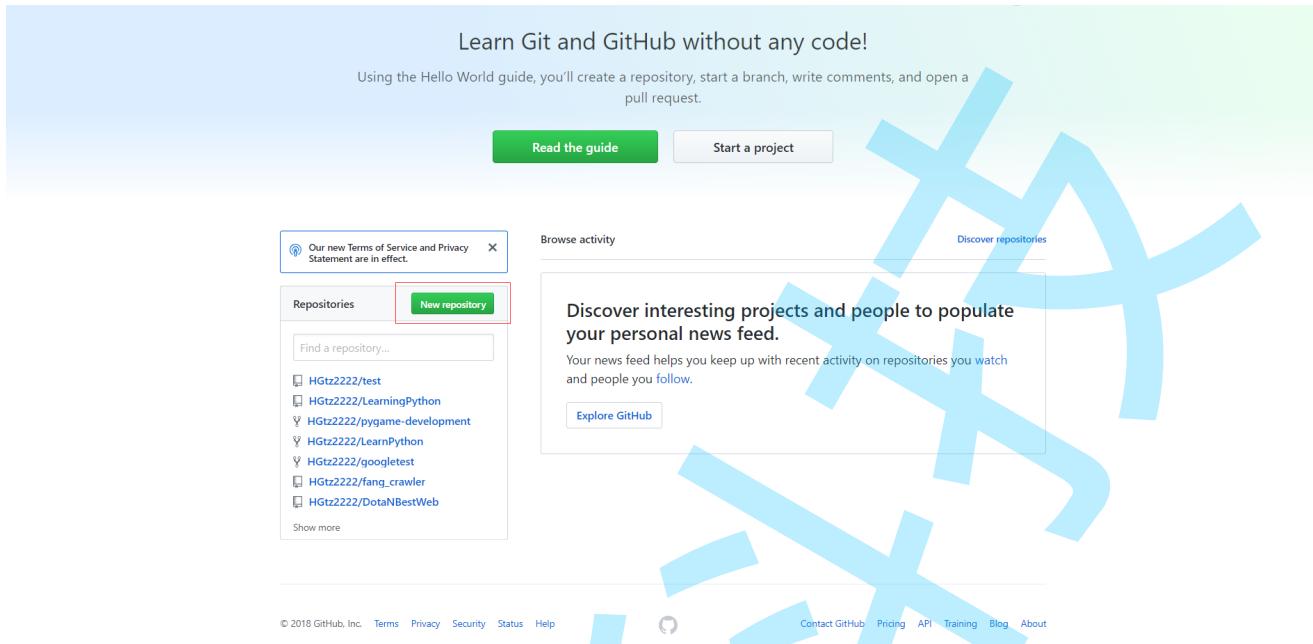
### 使用 Github 创建项目

## 注册账号

这个比较简单, 参考着官网提示即可. 需要进行邮箱校验.

# 创建项目

1. 登陆成功后, 进入个人主页, 点击左下方的 New repository 按钮新建项目



2. 然后跳转到的新页面中输入项目名称(注意, 名称不能重复, 系统会自动校验. 校验过程可能会花费几秒钟). 校验完毕后, 点击下方的 Create repository 按钮确认创建.

## Create a new repository

A repository contains all the files for your project, including the revision history.

Owner  
 HGtz2222 ▾

Repository name

Great repository names are short and memorable. Need inspiration? How about [super-duper-eureka](#).

Description (optional)

 Public

Anyone can see this repository. You choose who can commit.

 Private

You choose who can see and commit to this repository.

Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None ▾

Add a license: None ▾



**Create repository**

3. 在创建好的项目页面中复制项目的链接, 以备接下来进行下载.

if you've done this kind of thing before

or  HTTPS  SSH <https://github.com/HGtz2222/test2.git>



repository include a [README](#), [LICENSE](#), and [.gitignore](#).

## 下载项目到本地

创建好一个放置代码的目录.

```
git clone [url]
```

这里的 url 就是刚刚建立好的 项目 的链接.

## 三板斧第一招: git add

将代码放到刚才下载好的目录中

```
git add [文件名]
```

将需要用 git 管理的文件告知 git

## 三板斧第二招: git commit

提交改动到本地

```
git commit .
```

最后的 "." 表示当前目录

提交的时候应该注明提交日志, 描述改动的详细内容.

## 三板斧第三招: git push

同步到远端服务器上

```
git push
```

需要填入用户名密码. 同步成功后, 刷新 Github 页面就能看到代码改动了.

配置免密码提交

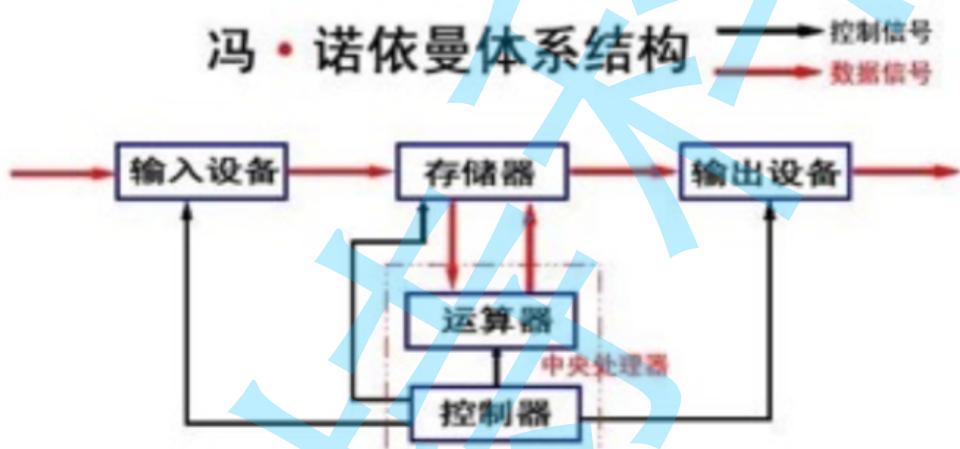
<https://blog.csdn.net/camillezj/article/details/55103149>

## 本节重点：

- 认识冯诺依曼系统
- 操作系统概念与定位
- 深入理解进程概念，了解PCB
- 学习进程状态，学会创建进程，掌握僵尸进程和孤儿进程，及其形成原因和危害
- 了解进程调度，Linux进程优先级，理解进程竞争性与独立性，理解并行与并发
- 理解环境变量，熟悉常见环境变量及相关指令，getenv/setenv函数
- 理解C内存空间分配规律，了解进程内存映像和应用程序区别，认识地址空间。
- 选学Linux2.6 kernel, O(1)调度算法架构

## 冯诺依曼体系结构

我们常见的计算机，如笔记本。我们不常见的计算机，如服务器，大部分都遵守冯诺依曼体系。



截至目前，我们所认识的计算机，都是有一个个的硬件组件组成

- 输入单元：包括键盘，鼠标，扫描仪，写板等
- 中央处理器(CPU)：含有运算器和控制器等
- 输出单元：显示器，打印机等

### 关于冯诺依曼，必须强调几点：

- 这里的存储器指的是内存
- 不考虑缓存情况，这里的CPU能且只能对内存进行读写，不能访问外设(输入或输出设备)
- 外设(输入或输出设备)要输入或者输出数据，也只能写入内存或者从内存中读取。
- 一句话，所有设备都只能直接和内存打交道。

对冯诺依曼的理解，不能停留在概念上，要深入到对软件数据流理解上，请解释，从你登录上qq开始和某位朋友聊天开始，数据的流动过程。从你打开窗口，开始给他发消息，到他的到消息之后的数据流动过程。如果是在qq上发送文件呢？

# 操作系统(Operator System)

## 概念

任何计算机系统都包含一个基本的程序集合，称为操作系统(OS)。笼统的理解，操作系统包括：

- 内核（进程管理，内存管理，文件管理，驱动管理）
- 其他程序（例如函数库，shell程序等等）

## 设计OS的目的

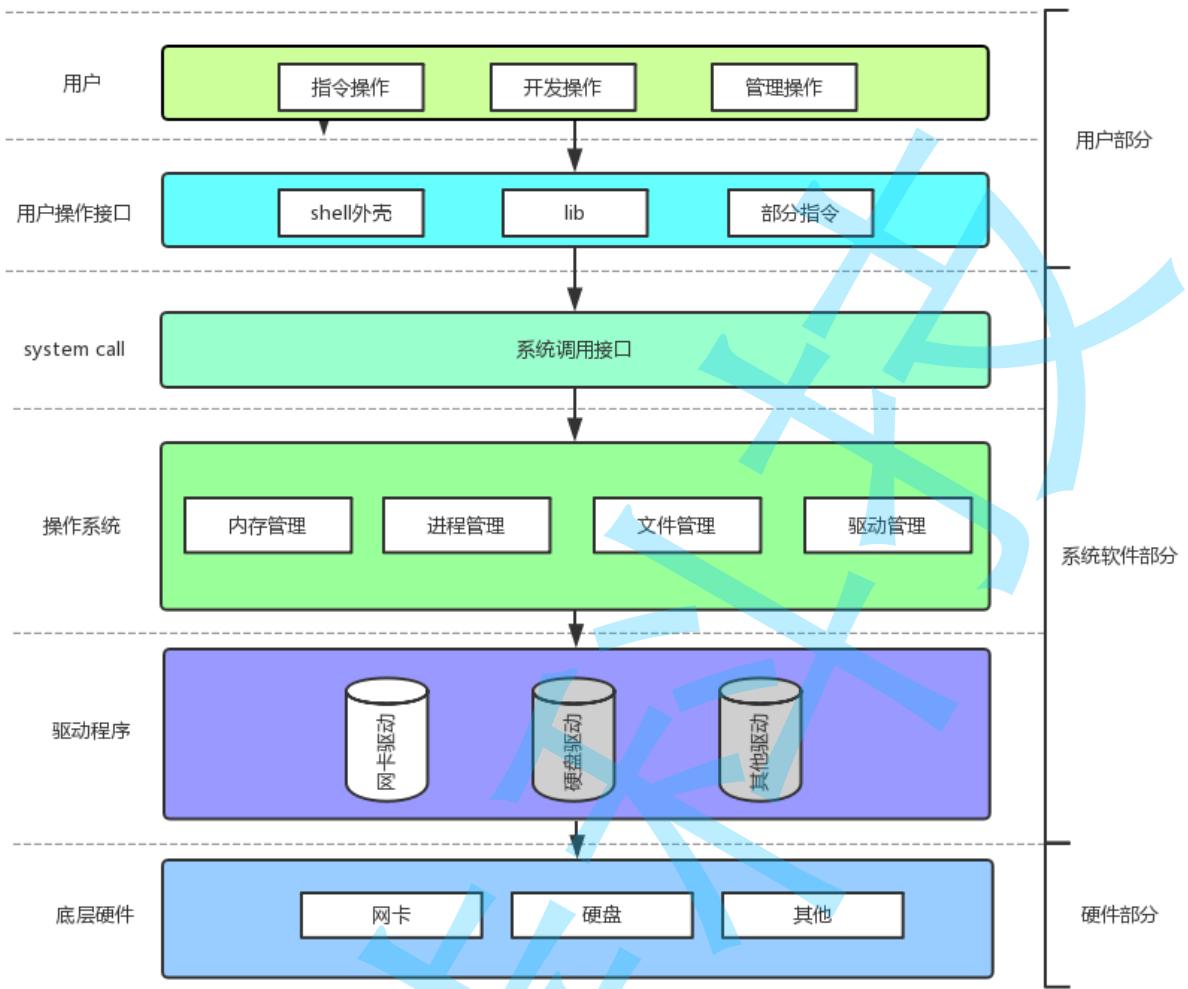
- 与硬件交互，管理所有的软硬件资源
- 为用户程序（应用程序）提供一个良好的执行环境

## 定位

- 在整个计算机软硬件架构中，操作系统的定位是：一款纯正的“搞管理”的软件

## 如何理解 “管理”

- 管理的例子
- 描述被管理对象
- 组织被管理对象



## 总结

计算机管理硬件

1. 描述起来，用struct结构体
2. 组织起来，用链表或其他高效的数据结构

## 系统调用和库函数概念

- 在开发角度，操作系统对外会表现为一个整体，但是会暴露自己的部分接口，供上层开发使用，这部分由操作系统提供的接口，叫做系统调用。
- 系统调用在使用上，功能比较基础，对用户的要求相对也比较高，所以，有心的开发者可以对部分系统调用进行适度封装，从而形成库，有了库，就很有利于更上层用户或者开发者进行二次开发。

## 承上启下

那在还没有学习进程之前，就问大家，操作系统是怎么管理进程的呢？很简单，先把进程描述起来，再把进程组织起来！

## 进程

# 基本概念

- 课本概念：程序的一个执行实例，正在执行的程序等
- 内核观点：担当分配系统资源（CPU时间，内存）的实体。

## 描述进程-PCB

- 进程信息被放在一个叫做进程控制块的数据结构中，可以理解为进程属性的集合。
- 课本上称之为PCB (process control block) , Linux操作系统下的PCB是: [task\\_struct](#)

### task\_struct-PCB的一种

- 在Linux中描述进程的结构体叫做task\_struct。
- task\_struct是Linux内核的一种数据结构，它会被装载到RAM(内存)里并且包含着进程的信息。

### task\_struct内容分类

- 标示符: 描述本进程的唯一标示符，用来区别其他进程。
- 状态: 任务状态，退出代码，退出信号等。
- 优先级: 相对于其他进程的优先级。
- 程序计数器: 程序中即将被执行的下一条指令的地址。
- 内存指针: 包括程序代码和进程相关数据的指针，还有和其他进程共享的内存块的指针
- 上下文数据:** 进程执行时处理器的寄存器中的数据[休学例子，要加图CPU，寄存器]。
- I / O状态信息: 包括显示的I/O请求,分配给进程的I / O设备和被进程使用的文件列表。
- 记账信息: 可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号等。
- 其他信息

## 组织进程

可以在内核源代码里找到它。所有运行在系统里的进程都以task\_struct链表的形式存在内核里。

## 查看进程

进程的信息可以通过 `/proc` 系统文件夹查看

- 如：要获取PID为1的进程信息，你需要查看 `/proc/1` 这个文件夹。

```
[root@localhost hb]# ls /proc/
1 1581 1720 1886 21 2218 2390 2448 2477 2539 2754 2856 43
10 16 1723 19 2102 2224 24 2450 2479 2542 2758 2890 44
11 168 1751 1922 2107 2278 2402 2453 2481 2554 2768 29 45
12 161 1761 2 2109 2289 2403 2456 2486 26 2771 3 5
1282 1611 1762 20 2113 2295 2412 2457 2493 27 2772 30 6
13 162 1797 2035 2117 23 2427 2460 2494 2722 28 31 612
1303 1683 18 2058 2121 2347 2429 2461 2497 2728 280 32 7
14 1695 1809 2059 2128 2357 2433 2462 25 2731 281 375 76
15 17 1813 2061 2131 2365 2435 2466 2530 2733 2837 4 77
152 1701 1830 2074 2148 2366 2437 2469 2537 2737 2838 40 8
153 1716 1865 2088 22 2386 2447 2474 2538 2739 2844 41 804
805  cpuinfo  iomem  locks  partitions  sysvipc
9  crypto  iports  mdstat  sched_debug  timer_list
914  devices  irq  meminfo  schedstat  timer_stats
915  diskstats  kallsyms  misc  scsi  tty
965  dma  kcore  modules  self  uptime
acpi  driver  keys  mounts  slabinfo  version
asound  execdomains  key-users  mpt  softirqs  vmallocinfo
buddyinfo  fb  kms  mtd  stat  vmstat
bus  filesystems  kpagecount  mttr  swaps  zoneinfo
cgroups  fs  kpageflags  net  sys
cmdline  interrupts  loadavg  pagetypeinfo  sysrq-trigger
```

- 大多数进程信息同样可以使用top和ps这些用户级工具来获取

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

int main()
{
    while(1){
        sleep(1);
    }
    return 0;
}
```

```
[root@localhost test]# ps aux | grep test | grep -v grep
root 3239 0.0 0.0 1864 284 pts/0 S 03:40 0:00 ./test
[root@localhost test]#
```

## 通过系统调用获取进程标识符

- 进程id (PID)
- 父进程id (PPID)

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

int main()
{
    printf("pid: %d\n", getpid());
    printf("ppid: %d\n", getppid());
    return 0;
}
```

## 通过系统调用创建进程-fork初识

- 运行 `man fork` 认识fork
- fork有两个返回值
- 父子进程代码共享，数据各自开辟空间，私有一份（采用写时拷贝）

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

int main()
{
    int ret = fork();
    printf("hello proc : %d!, ret: %d\n", getpid(), ret);
    sleep(1);
    return 0;
}
```

- fork 之后通常要用 `if` 进行分流

```

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

int main()
{
    int ret = fork();
    if(ret < 0){
        perror("fork");
        return 1;
    }
    else if(ret == 0){ //child
        printf("I am child : %d!, ret: %d\n", getpid(), ret);
    }else{ //father
        printf("I am father : %d!, ret: %d\n", getpid(), ret);
    }
    sleep(1);
    return 0;
}

```

## 进程状态

### 看看Linux内核源代码怎么说

- 为了弄明白正在运行的进程是什么意思，我们需要知道进程的不同状态。一个进程可以有几个状态（在Linux内核里，进程有时候也叫做任务）。
- 下面的状态在kernel源代码里定义：

```

/*
 * The task state array is a strange "bitmap" of
 * reasons to sleep. Thus "running" is zero, and
 * you can test for combinations of others with
 * simple bit tests.
 */
static const char * const task_state_array[] = {
    "R (running)", /* 0 */
    "S (sleeping)", /* 1 */
    "D (disk sleep)", /* 2 */
    "T (stopped)", /* 4 */
    "t (tracing stop)", /* 8 */
    "X (dead)", /* 16 */
    "Z (zombie)", /* 32 */
};

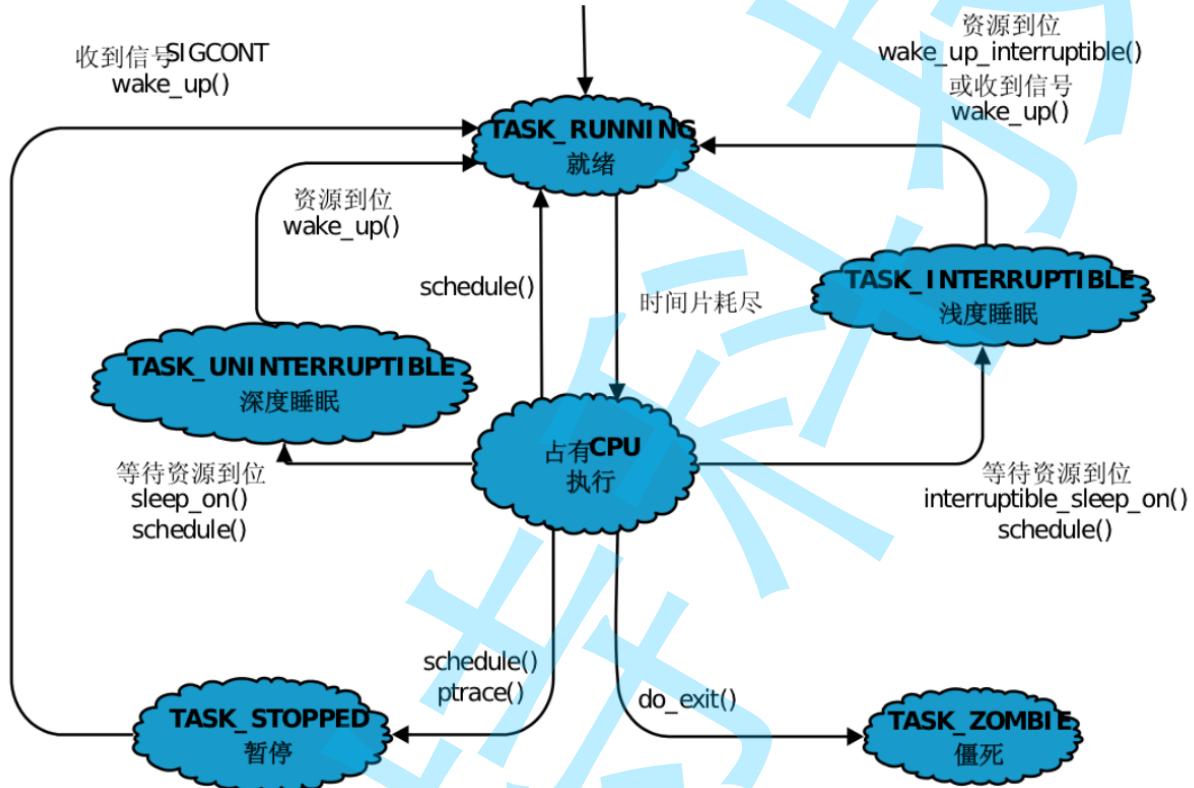
```

- R运行状态 (running)：并不意味着进程一定在运行中，它表明进程要么是在运行中要么在运行队列里。
- S睡眠状态 (sleeping)：意味着进程在等待事件完成（这里的睡眠有时候也叫做可中断睡眠（interruptible sleep））。

- D磁盘休眠状态 (Disk sleep) 有时候也叫不可中断睡眠状态 (uninterruptible sleep)，在这个状态的进程通常会等待IO的结束。
- T停止状态 (stopped)：可以通过发送 SIGSTOP 信号给进程来停止 (T) 进程。这个被暂停的进程可以通过发送 SIGCONT 信号让进程继续运行。
- X死亡状态 (dead)：这个状态只是一个返回状态，你不会在任务列表里看到这个状态。

## 进程状态查看

ps aux / ps axj 命令



## Z(zombie)-僵尸进程

- 僵死状态 (Zombies) 是一个比较特殊的状态。当进程退出并且父进程（使用wait()系统调用,后面讲）没有读取到子进程退出的返回代码时就会产生僵尸(尸)进程
- 僵死进程会以终止状态保持在进程表中，并且会一直在等待父进程读取退出状态代码。
- 所以，只要子进程退出，父进程还在运行，但父进程没有读取子进程状态，子进程进入Z状态

来一个创建维持30秒的僵尸进程例子：

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

int main()
{
    pid_t id = fork();
    if(id < 0){
        perror("fork");
        return 1;
    }
}
```

```

    }
    else if(id > 0){ //parent
        printf("parent[%d] is sleeping...\n", getpid());
        sleep(30);
    }else{
        printf("child[%d] is begin Z...\n", getpid());
        sleep(5);
        exit(EXIT_SUCCESS);
    }
    return 0;
}

```

编译并在另一个终端下启动监控

```

File Edit View Search Terminal Help
[root@MiWiFi-RICL-srv test]# ls
Makefile test test.c
[root@MiWiFi-RICL-srv test]# ./test

```

```

File Edit View Search Terminal Help
[root@MiWiFi-RICL-srv test]# while :; do ps aux | grep test | grep -v grep;
sleep 1; echo "#####"; done
#####
```

开始测试

```

File Edit View Search Terminal Help
[root@MiWiFi-RICL-srv test]# ls
Makefile test test.c
[root@MiWiFi-RICL-srv test]# ./test
parent[3872] is sleeping...
child[3873] is begin Z...

```

```

File Edit View Search Terminal Help
[root@MiWiFi-RICL-srv test]# while :; do ps aux | grep test | grep -v grep;
sleep 1; echo "#####"; done
#####
#####
#####
#####
root 3872 0.0 0.0 1868 372 pts/0 S+ 19:48 0:00 ./test
root 3873 0.0 0.0 0 0 pts/0 Z+ 19:48 0:00 [test] <defunct>
```

看到结果

```

#####
root 3872 0.0 0.0 1868 372 pts/0 S+ 19:48 0:00 ./test
root 3873 0.0 0.0 0 0 pts/0 Z+ 19:48 0:00 [test] <defunct>
```

[ptrace系统调用追踪进程运行，有兴趣研究一下](#)

## 僵尸进程危害

- 进程的退出状态必须被维持下去，因为他要告诉关心它的进程（父进程），你交给我的任务，我办的怎么样了。可父进程如果一直不读取，那子进程就一直处于Z状态？是的！
- 维护退出状态本身就是要用数据维护，也属于进程基本信息，所以保存在task\_struct(PCB)中，换句话说，Z状态一直不退出，PCB一直都要维护？是的！
- 那一个父进程创建了很多子进程，就是不回收，是不是就会造成内存资源的浪费？是的！因为数据结构对象本身就要占用内存，想想C中定义一个结构体变量（对象），是要在内存的某个位置进行开辟空间！
- 内存泄漏？是的！
- 如何避免？后面讲

## 进程状态总结

- 至此，值得关注的进程状态全部讲解完成，下面来认识另一种进程

## 孤儿进程

- 父进程如果提前退出，那么子进程后退出，进入Z之后，那该如何处理呢？
- 父进程先退出，子进程就称之为“孤儿进程”
- 孤儿进程被1号init进程领养，当然要有init进程回收喽。

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

int main()
{
    pid_t id = fork();
    if(id < 0){
        perror("fork");
        return 1;
    }
    else if(id == 0){ //child
        printf("I am child, pid : %d\n", getpid());
        sleep(10);
    }else{ //parent
        printf("I am parent, pid: %d\n", getpid());
        sleep(3);
        exit(0);
    }
    return 0;
}
```

来段代码：

```
[root@MiWiFi-R1CL-srv test]# ls  
Makefile test test.c  
[root@MiWiFi-R1CL-srv test]# ./test  
I am parent, pid: 4416  
I am child, pid : 4417  
[root@MiWiFi-R1CL-srv test]#
```

## 进程优先级

## 基本概念

- CPU资源分配的先后顺序，就是指进程的优先权（priority）。
  - 优先权高的进程有优先执行权利。配置进程优先权对多任务环境的Linux很有用，可以改善系统性能。
  - 还可以把进程运行到指定的CPU上，这样一来，把不重要的进程安排到某个CPU，可以大大改善系统整体性能。

## 查看系统进程

在linux或者unix系统中，用ps -l命令则会类似输出以下几个内容：

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	0	4226	4201	0	80	0	-	2121	-	pts/0	00:00:00	su
4	S	0	4241	4226	0	80	0	-	1314	-	pts/0	00:00:00	bash
4	S	0	4556	4241	0	80	0	-	1896	-	pts/0	00:00:00	su
4	S	0			A				1896	-	ntsc/A	00:00:00	cii

我们很容易注意到其中的几个重要信息，有下：

- UID : 代表执行者的身份
- PID : 代表这个进程的代号
- PPID : 代表这个进程是由哪个进程发展衍生而来的，亦即父进程的代号
- PRI : 代表这个进程可被执行的优先级，其值越小越早被执行
- NI : 代表这个进程的nice值

## PRI and NI

- PRI也还是比较好理解的，即进程的优先级，或者通俗点说就是程序被CPU执行的先后顺序，此值越小进程的优先级别越高
- 那NI呢？就是我们所要说的nice值了，其表示进程可被执行的优先级的修正数值
- PRI值越小越快被执行，那么加入nice值后，将会使得PRI变为： $PRI(new) = PRI(old) + nice$
- 这样，当nice值为负值的时候，那么该程序将会优先级值将变小，即其优先级会变高，则其越快被执行
- 所以，调整进程优先级，在Linux下，就是调整进程nice值
- nice其取值范围是-20至19，一共40个级别。

## PRI vs NI

- 需要强调一点的是，进程的nice值不是进程的优先级，他们不是一个概念，但是进程nice值会影响到进程的优先级变化。
- 可以理解nice值是进程优先级的修正修正数据

## 查看进程优先级的命令

### 用top命令更改已存在进程的nice：

- top
- 进入top后按“r”->输入进程PID->输入nice值

## 其他概念

- 竞争性：系统进程数目众多，而CPU资源只有少量，甚至1个，所以进程之间是具有竞争属性的。为了高效完成任务，更合理竞争相关资源，便具有了优先级
- 独立性：多进程运行，需要独享各种资源，多进程运行期间互不干扰
- 并行：多个进程在多个CPU下分别，同时进行运行，这称之为并行
- 并发：多个进程在一个CPU下采用进程切换的方式，在一段时间之内，让多个进程都得以推进，称之为并发

## 环境变量

### 基本概念

- 环境变量(environment variables)一般是指在操作系统中用来指定操作系统运行环境的一些参数
- 如：我们在编写C/C++代码的时候，在链接的时候，从来不知道我们的所链接的动态静态库在哪里，但是照样可以链接成功，生成可执行程序，原因就是有相关环境变量帮助编译器进行查找。
- 环境变量通常具有某些特殊用途，还有在系统当中通常具有全局特性

## 常见环境变量

- PATH : 指定命令的搜索路径
- HOME : 指定用户的主工作目录(即用户登陆到Linux系统中时,默认的目录)
- SHELL : 当前Shell,它的值通常是/bin/bash。

## 查看环境变量方法

```
echo $NAME //NAME:你的环境变量名称
```

## 测试PATH

1. 创建hello.c文件

```
#include <stdio.h>

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

2. 对比./hello执行和之间hello执行
3. 为什么有些指令可以直接执行，不需要带路径，而我们的二进制程序需要带路径才能执行？
4. 将我们的程序所在路径加入环境变量PATH当中, export PATH=\$PATH:hello程序所在路径
5. 对比测试
6. 还有什么方法可以不用带路径，直接就可以运行呢？

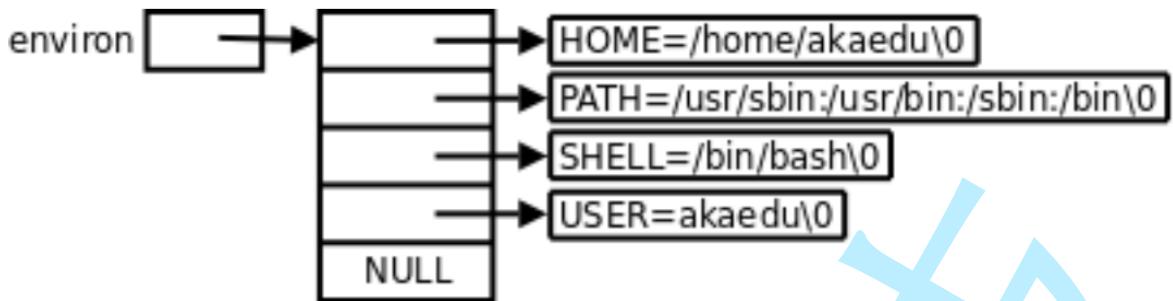
## 测试HOME

1. 用root和普通用户，分别执行 echo \$HOME ,对比差异  
. 执行 cd ~; pwd ,对应 ~ 和 HOME 的关系

## 和环境变量相关的命令

1. echo: 显示某个环境变量值
2. export: 设置一个新的环境变量
3. env: 显示所有环境变量
4. unset: 清除环境变量
5. set: 显示本地定义的shell变量和环境变量

## 环境变量的组织方式



每个程序都会收到一张环境表，环境表是一个字符指针数组，每个指针指向一个以'\0'结尾的环境字符串

## 通过代码如何获取环境变量

- 命令行第三个参数

```
#include <stdio.h>

int main(int argc, char *argv[], char *env[])
{
    int i = 0;
    for(; env[i]; i++){
        printf("%s\n", env[i]);
    }
    return 0;
}
```

- 通过第三方变量environ获取

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    extern char **environ;
    int i = 0;
    for(; environ[i]; i++){
        printf("%s\n", environ[i]);
    }
    return 0;
}
```

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。

## 通过系统调用获取或设置环境变量

- `putenv`, 后面讲解
- `getenv`, 本次讲解

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

int main()
{
    printf("%s\n", getenv("PATH"));
    return 0;
}
```

常用getenv和putenv函数来访问特定的环境变量。

## 环境变量通常是具有全局属性的

- 环境变量通常具有全局属性，可以被子进程继承下去

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

int main()
{
    char * env = getenv("MYENV");
    if(env){
        printf("%s\n", env);
    }
    return 0;
}
```

直接查看，发现没有结果，说明该环境变量根本不存在

- 导出环境变量  
`export MYENV="hello world"`
- 再次运行程序，发现结果有了！说明：环境变量是可以被子进程继承下去的！想想为什么？

## 实验

- 如果只进行 `MYENV = "helloworld"`，不调用export导出，在用我们的程序查看，会有什么结果？为什么？
- 普通变量

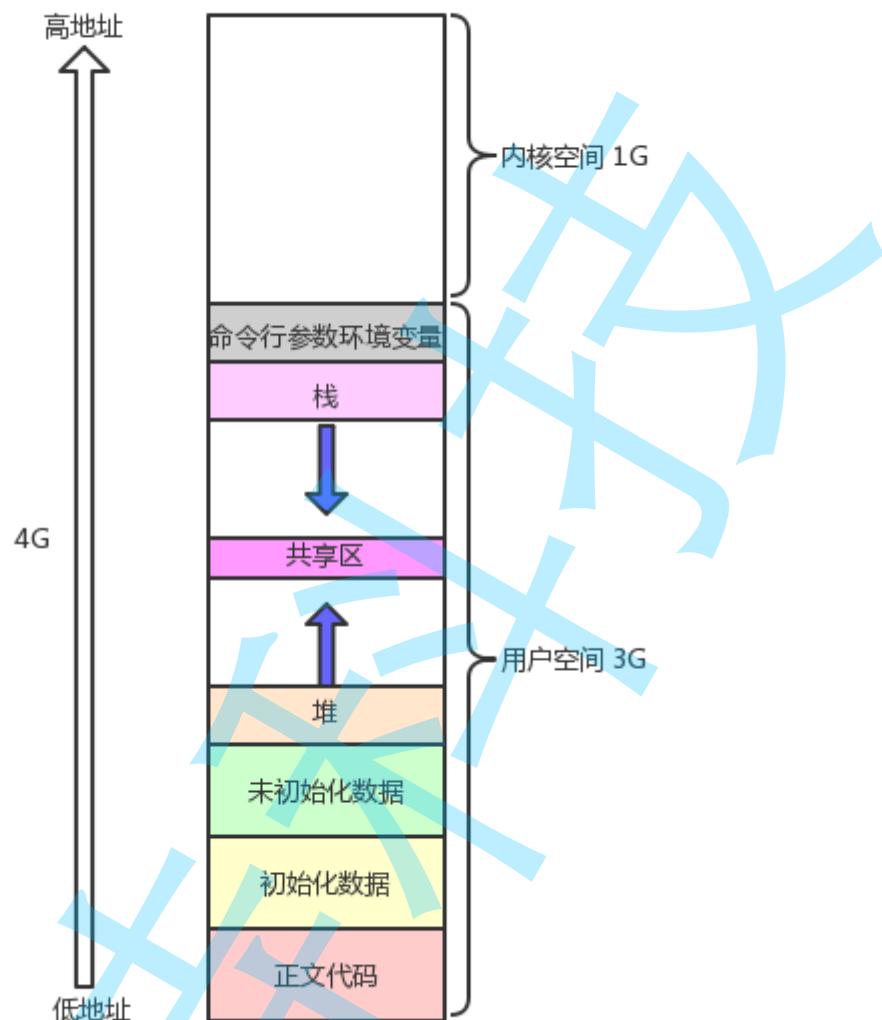
## 程序地址空间

### 研究背景

- kernel 2.6.32
- 32位平台

## 程序地址空间回顾

我们在讲C语言的时候，老师给大家画过这样的空间布局图



可是我们对他并不理解!

## 来段代码感受一下

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

int g_val = 0;

int main()
{
    pid_t id = fork();
    if(id < 0){
        perror("fork");
        return 0;
    }
    else if(id == 0){ //child
        ...
    }
}
```

```
    printf("child[%d]: %d : %p\n", getpid(), g_val, &g_val);
}else{ //parent
    printf("parent[%d]: %d : %p\n", getpid(), g_val, &g_val);
}
sleep(1);
return 0;
}
```

输出

```
//与环境相关，观察现象即可
parent[2995]: 0 : 0x80497d8
child[2996]: 0 : 0x80497d8
```

我们发现，输出出来的变量值和地址是一模一样的，很好理解呀，因为子进程按照父进程为模板，父子并没有对变量进行任何修改。可是将代码稍加改动：

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

int g_val = 0;

int main()
{
    pid_t id = fork();
    if(id < 0){
        perror("fork");
        return 0;
    }
    else if(id == 0){ //child, 子进程肯定先跑完，也就是子进程先修改，完成之后，父进程再读取
        g_val=100;
        printf("child[%d]: %d : %p\n", getpid(), g_val, &g_val);
    }else{ //parent
        sleep(3);
        printf("parent[%d]: %d : %p\n", getpid(), g_val, &g_val);
    }
    sleep(1);
    return 0;
}
```

输出结果：

```
//与环境相关，观察现象即可
child[3046]: 100 : 0x80497e8
parent[3045]: 0 : 0x80497e8
```

我们发现，父子进程，输出地址是一致的，但是变量内容不一样！能得出如下结论：

- 变量内容不一样，所以父子进程输出的变量绝对不是同一个变量

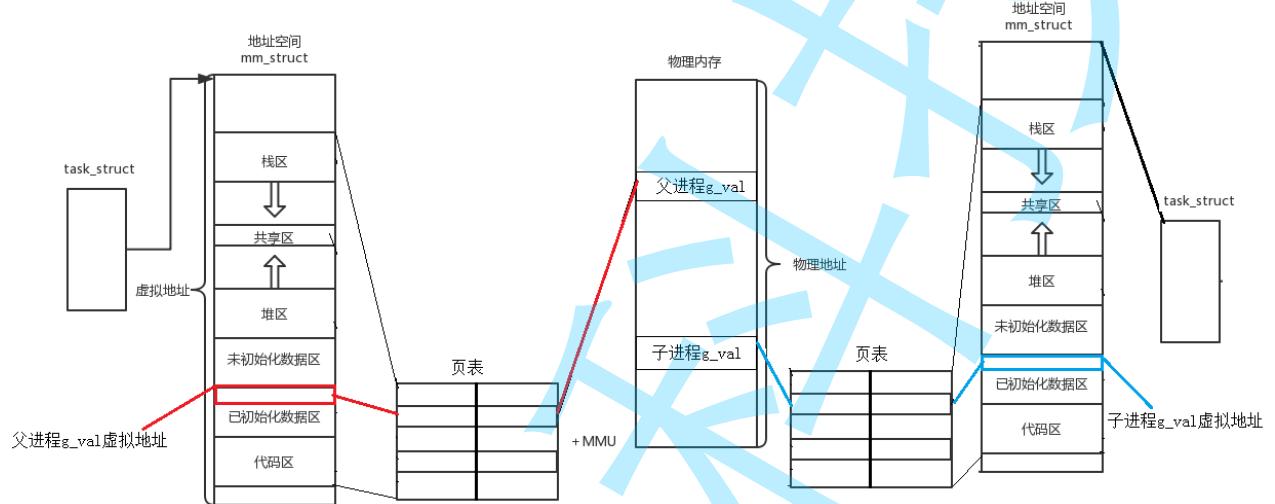
- 但地址值是一样的，说明，该地址绝对不是物理地址！
- 在Linux地址下，这种地址叫做 **虚拟地址**
- 我们在用C/C++语言所看到的地址，全部都是虚拟地址！物理地址，用户一概看不到，由OS统一管理

OS必须负责将 **虚拟地址** 转化成 **物理地址**。

## 进程地址空间

所以之前说‘程序的地址空间’是不准确的，准确的应该说成 **进程地址空间**，那该如何理解呢？看图：

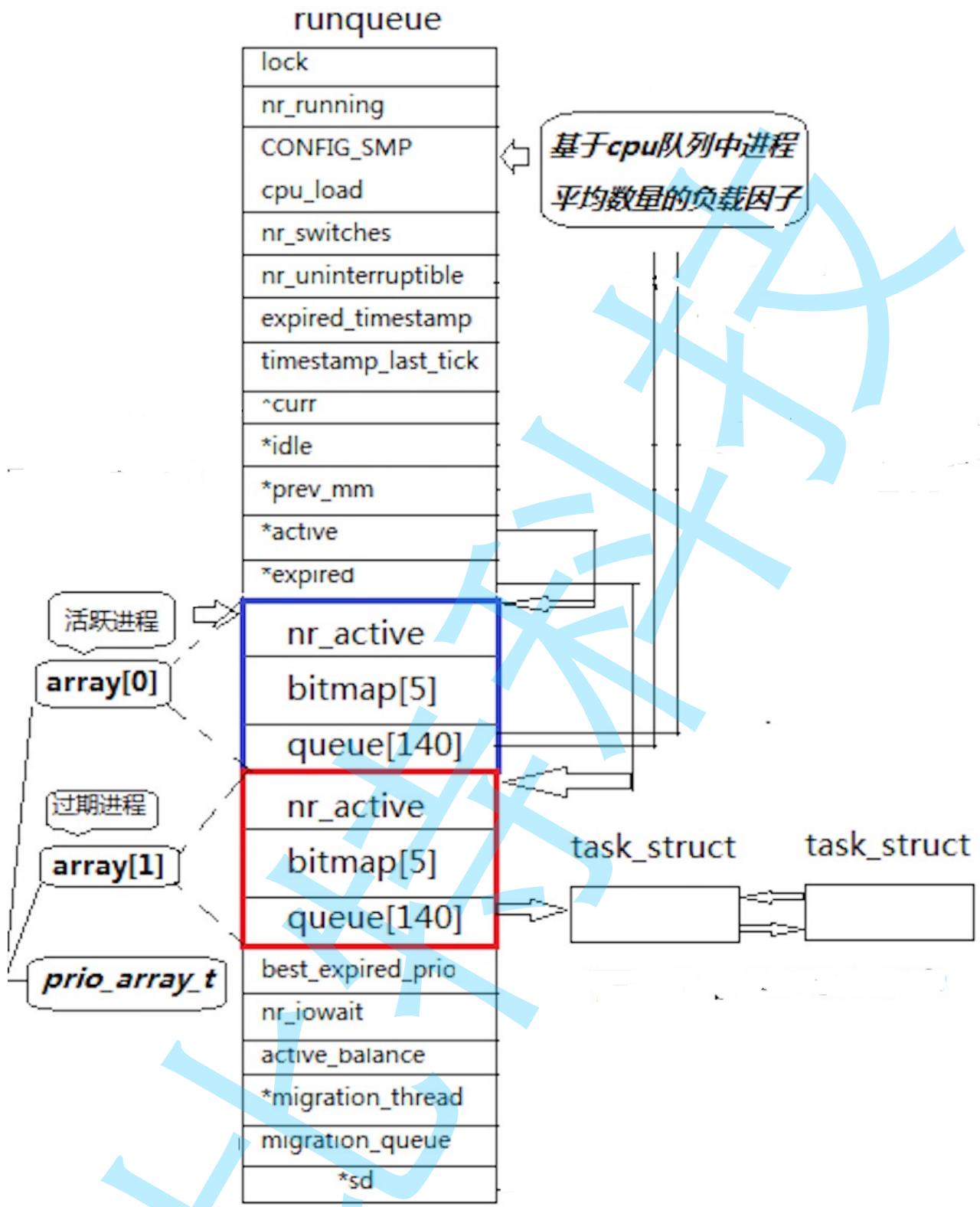
### 分页&虚拟地址空间



说明：

- 上面的图就足矣说名问题，同一个变量，地址相同，其实是虚拟地址相同，内容不同其实是被映射到了不同的物理地址！

## Linux2.6内核进程调度队列-选学



上图是Linux2.6内核中进程队列的数据结构，之间关系也已经给大家画出来，方便大家理解

## 一个CPU拥有一个runqueue

- 如果有多个CPU就要考虑进程个数的负载均衡问题

## 优先级

- 普通优先级：100 ~ 139（我们都是普通的优先级，想想nice值的取值范围，可与之对应！）
- 实时优先级：0 ~ 99（不关心）

## 活动队列

- 时间片还没有结束的所有进程都按照优先级放在该队列
- nr\_active: 总共有多少个运行状态的进程
- queue[140]: 一个元素就是一个进程队列，相同优先级的进程按照FIFO规则进行排队调度，所以，数组下标就是优先级！
- 从该结构中，选择一个最合适的进程，过程是怎么的呢？
  - 从0下表开始遍历queue[140]
  - 找到第一个非空队列，该队列必定为优先级最高的队列
  - 拿到选中队列的第一个进程，开始运行，调度完成！
  - 遍历queue[140]时间复杂度是常数！但还是太低效了！
- bitmap[5]:一共140个优先级，一共140个进程队列，为了提高查找非空队列的效率，就可以用5\*32个比特位表示队列是否为空，这样，便可以大大提高查找效率！

## 过期队列

- 过期队列和活动队列结构一模一样
- 过期队列上放置的进程，都是时间片耗尽的进程
- 当活动队列上的进程都被处理完毕之后，对过期队列的进程进行时间片重新计算

## active指针和expired指针

- active指针永远指向活动队列
- expired指针永远指向过期队列
- 可是活动队列上的进程会越来越少，过期队列上的进程会越来越多，因为进程时间片到期时一直都存在的。
- 没关系，在合适的时候，只要能够交换active指针和expired指针的内容，就相当于有具有一批新的活动进程！

## 总结

- 在系统当中查找一个最合适调度的进程的时间复杂度是一个常数，不随着进程增多而导致时间成本增加，我们称之为进程调度O(1)算法！

[更多文档](#)

## 本节重点：

- 学习进程创建,fork/vfork
- 学习到进程等待
- 学习到进程程序替换,微型shell, 重新认识shell运行原理
- 学习到进程终止,认识\$?

## 进程创建

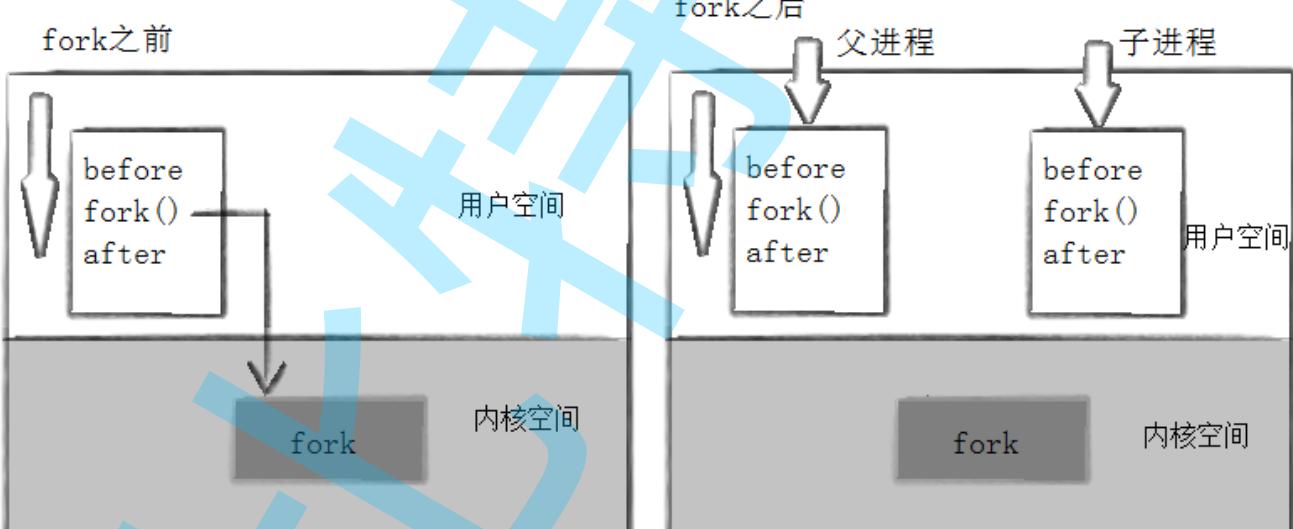
### fork函数初识

在linux中fork函数时非常重要的函数，它从已存在进程中创建一个新进程。新进程为子进程，而原进程为父进程。

```
#include <unistd.h>
pid_t fork(void);
返回值：自进程中返回0，父进程返回子进程id，出错返回-1
```

进程调用fork，当控制转移到内核中的fork代码后，内核做：

- 分配新的内存块和内核数据结构给子进程
- 将父进程部分数据结构内容拷贝至子进程
- 添加子进程到系统进程列表当中
- fork返回，开始调度器调度



当一个进程调用fork之后，就有两个二进制代码相同的进程。而且它们都运行到相同的地方。但每个进程都将可以开始它们自己的旅程，看如下程序。

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

    printf("Before: pid is %d\n", getpid());
```

```

if ( (pid=fork()) == -1 ) perror("fork()");
printf("After:pid is %d, fork return %d\n", getpid(), pid);
sleep(1);
return 0;
}

```

运行结果：

```

[root@localhost linux]# ./a.out
Before: pid is 43676
After:pid is 43676, fork return 43677
After:pid is 43677, fork return 0

```

这里看到了三行输出，一行before，两行after。进程43676先打印before消息，然后它有打印after。另一个after消息有43677打印的。注意到进程43677没有打印before，为什么呢？如下图所示



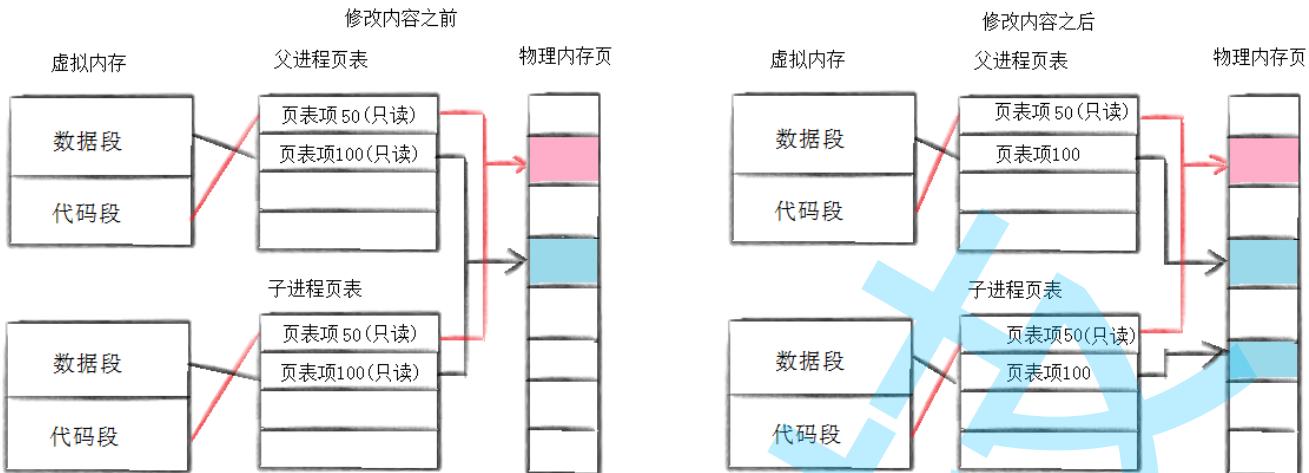
所以，`fork`之前父进程独立执行，`fork`之后，父子两个执行流分别执行。注意，`fork`之后，谁先执行完全由调度器决定。

## fork函数返回值

- 子进程返回0，
- 父进程返回的是子进程的pid。

## 写时拷贝

通常，父子代码共享，父子再不写入时，数据也是共享的，当任意一方试图写入，便以写时拷贝的方式各自一份副本。具体见下图：



## fork常规用法

- 一个父进程希望复制自己，使父子进程同时执行不同的代码段。例如，父进程等待客户端请求，生成子进程来处理请求。
- 一个进程要执行一个不同的程序。例如子进程从fork返回后，调用exec函数。

## fork调用失败的原因

- 系统中有太多的进程
- 实际用户的进程数超过了限制

## 进程终止

### 进程退出场景

- 代码运行完毕，结果正确
- 代码运行完毕，结果不正确
- 代码异常终止

### 进程常见退出方法

正常终止（可以通过 `echo $?` 查看进程退出码）：

- 从main返回
- 调用exit
- `_exit`

异常退出：

- `ctrl + c`, 信号终止

### `_exit`函数

```
#include <unistd.h>
void _exit(int status);
```

参数：status 定义了进程的终止状态，父进程通过wait来获取该值

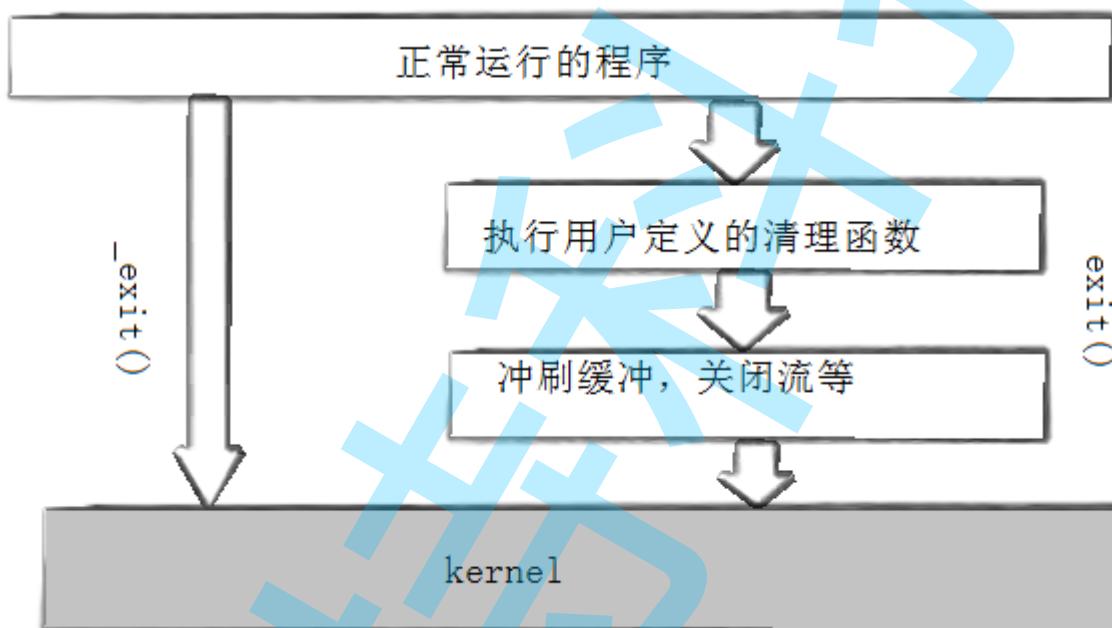
- 说明：虽然status是int，但是仅有低8位可以被父进程所用。所以\_exit(-1)时，在终端执行\$?发现返回值是255。

## exit函数

```
#include <unistd.h>
void exit(int status);
```

exit最后也会调用exit,但在调用exit之前,还做了其他工作:

1. 执行用户通过 atexit或on\_exit定义的清理函数。
2. 关闭所有打开的流，所有的缓存数据均被写入
3. 调用\_exit



实例:

```
int main()
{
    printf("hello");
    exit(0);
}
```

运行结果:

```
[root@localhost linux]# ./a.out
hello[root@localhost linux]#
```

```
int main()
{
    printf("hello");
    _exit(0);
}
```

运行结果:

```
[root@localhost linux]# ./a.out
[root@localhost linux]#
```

## return退出

return是一种更常见的退出进程方法。执行return n等同于执行exit(n),因为调用main的运行时函数会将main的返回值当做exit的参数。

## 进程等待

### 进程等待必要性

- 之前讲过，子进程退出，父进程如果不管不顾，就可能造成‘僵尸进程’的问题，进而造成内存泄漏。
- 另外，进程一旦变成僵尸状态，那就刀枪不入，“杀人不眨眼”的kill -9 也无能为力，因为谁也没有办法杀死一个已经死去的进程。
- 最后，父进程派给子进程的任务完成的如何，我们需要知道。如，子进程运行完成，结果对还是不对，或者是否正常退出。
- 父进程通过进程等待的方式，回收子进程资源，获取子进程退出信息

### 进程等待的方法

#### wait方法

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

pid_t wait(int*status);

返回值:
    成功返回被等待进程pid, 失败返回-1。
参数:
    输出型参数, 获取子进程退出状态, 不关心则可以设置成为NULL
```

#### waitpid方法

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

返回值:

当正常返回的时候waitpid返回收集到的子进程的进程ID;  
如果设置了选项WNOHANG,而调用中waitpid发现没有已退出的子进程可收集,则返回0;  
如果调用中出错,则返回-1,这时errno会被设置成相应的值以指示错误所在;

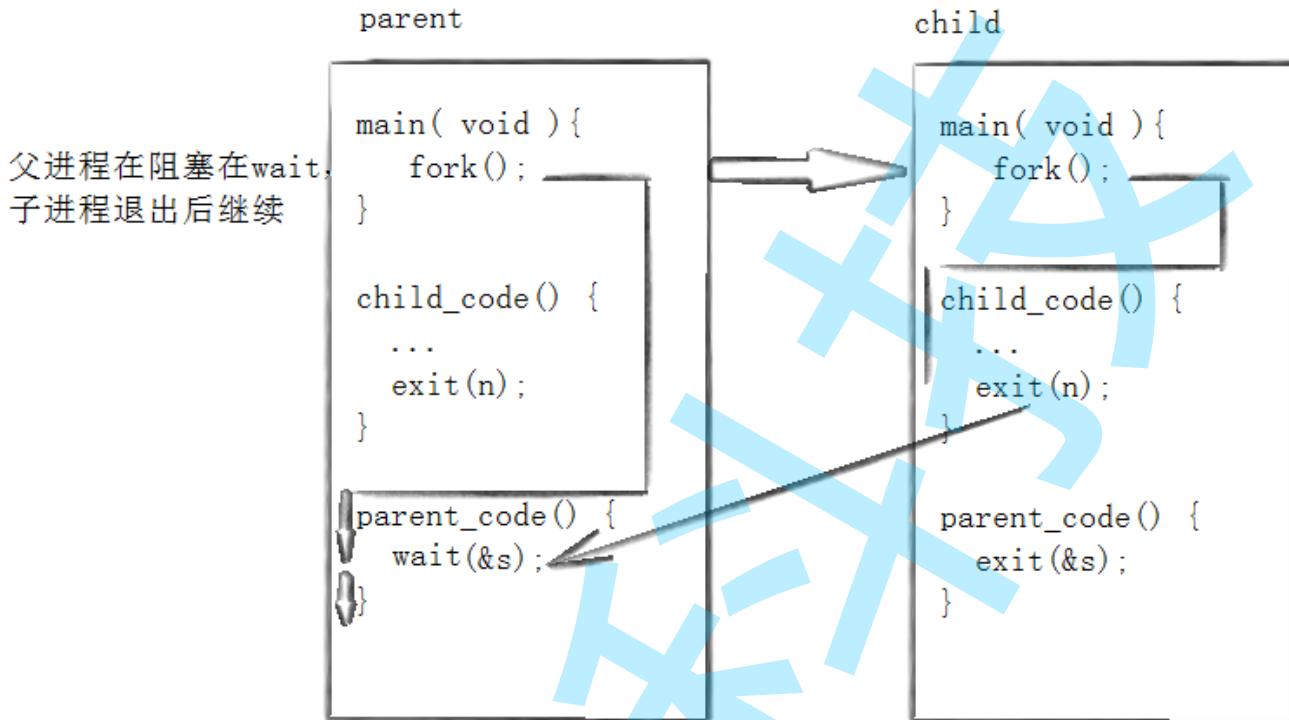
参数:

pid:  
 Pid=-1,等待任一个子进程。与wait等效。  
 Pid>0,等待其进程ID与pid相等的子进程。

status:  
 WIFEXITED(status): 若为正常终止子进程返回的状态, 则为真。 (查看进程是否是正常退出)  
 WEXITSTATUS(status): 若WIFEXITED非零, 提取子进程退出码。 (查看进程的退出码)

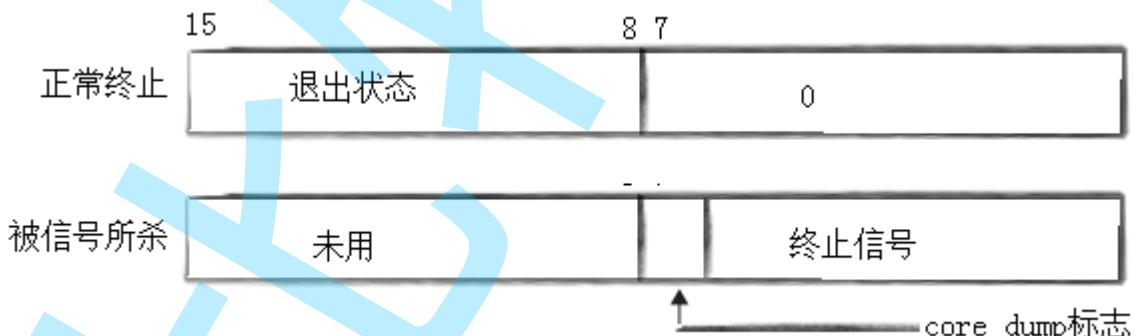
options:  
 WNOHANG: 若pid指定的子进程没有结束, 则waitpid()函数返回0, 不予以等待。若正常结束, 则返回该子进程的ID。

- 如果子进程已经退出，调用wait/waitpid时，wait/waitpid会立即返回，并且释放资源，获得子进程退出信息。
- 如果在任意时刻调用wait/waitpid，子进程存在且正常运行，则进程可能阻塞。
- 如果不存在该子进程，则立即出错返回。



## 获取子进程status

- wait和waitpid，都有一个status参数，该参数是一个输出型参数，由操作系统填充。
- 如果传递NULL，表示不关心子进程的退出状态信息。
- 否则，操作系统会根据该参数，将子进程的退出信息反馈给父进程。
- status不能简单的当作整形来看待，可以当作位图来看待，具体细节如下图（只研究status低16比特位）：



测试代码：

```

#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main( void )

```

```

{
    pid_t pid;

    if ( (pid=fork()) == -1 )
        perror("fork"),exit(1);

    if ( pid == 0 ){
        sleep(20);
        exit(10);
    } else {
        int st;
        int ret = wait(&st);

        if ( ret > 0 && ( st & 0X7F ) == 0 ){ // 正常退出
            printf("child exit code:%d\n", (st>>8)&0xFF);
        } else if( ret > 0 ) { // 异常退出
            printf("sig code : %d\n", st&0X7F );
        }
    }
}

```

测试结果：

```

[root@localhost linux]# ./a.out #等20秒退出
child exit code:10
[root@localhost linux]# ./a.out #在其他终端kill掉
sig code : 9

```

## 具体代码实现

- 进程的阻塞等待方式：

```

int main()
{
    pid_t pid;
    pid = fork();
    if(pid < 0){
        printf("%s fork error\n",__FUNCTION__);
        return 1;
    } else if( pid == 0 ){ //child
        printf("child is run, pid is : %d\n",getpid());
        sleep(5);
        exit(257);
    } else{
        int status = 0;
        pid_t ret = waitpid(-1, &status, 0); //阻塞式等待，等待5S
        printf("this is test for wait\n");
        if( WIFEXITED(status) && ret == pid ){
            printf("wait child 5s success, child return code is :%d.\n",WEXITSTATUS(status));
        }else{
            printf("wait child failed, return.\n");
            return 1;
        }
    }
}

```

```
    }
    return 0;
}
```

运行结果：

```
[root@localhost linux]# ./a.out
child is run, pid is : 45110
this is test for wait
wait child 5s success, child return code is :1.
```

- 进程的非阻塞等待方式：

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

int main()
{
    pid_t pid;

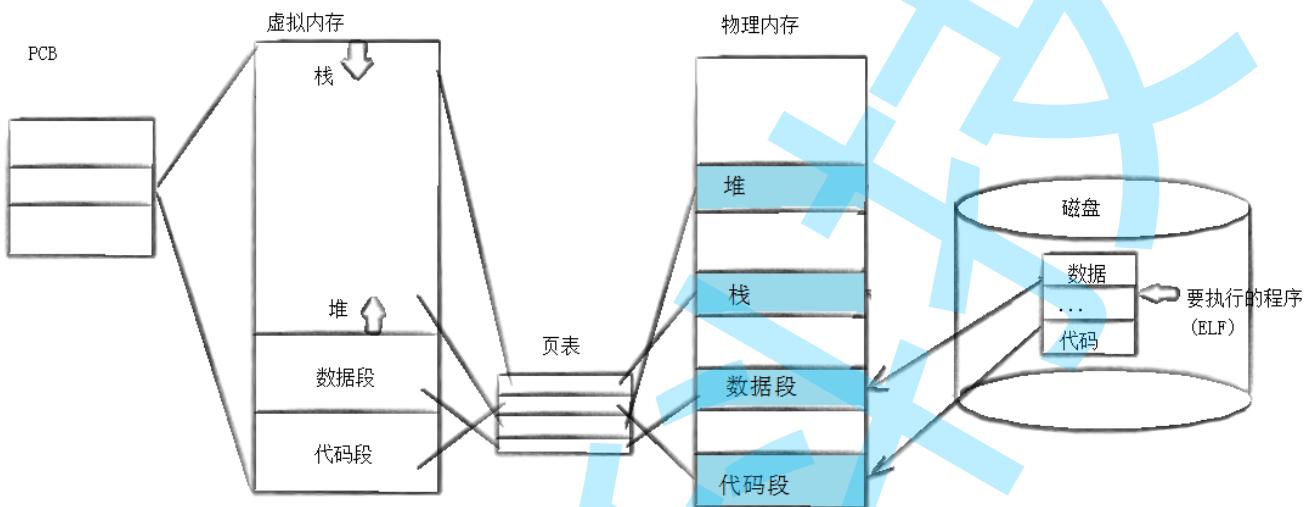
    pid = fork();
    if(pid < 0){
        printf("%s fork error\n",__FUNCTION__);
        return 1;
    }else if( pid == 0 ){ //child
        printf("child is run, pid is : %d\n",getpid());
        sleep(5);
        exit(1);
    } else{
        int status = 0;
        pid_t ret = 0;
        do
        {
            ret = waitpid(-1, &status, WNOHANG); //非阻塞式等待
            if( ret == 0 ){
                printf("child is running\n");
            }
            sleep(1);
        }while(ret == 0);

        if( WIFEXITED(status) && ret == pid ){
            printf("wait child 5s success, child return code is :%d.\n",WEXITSTATUS(status));
        }else{
            printf("wait child failed, return.\n");
            return 1;
        }
    }
    return 0;
}
```

# 进程序替换

## 替换原理

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。



## 替换函数

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 execle(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[]);
```

## 函数解释

- 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。

## 命名理解

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。

- l(list) : 表示参数采用列表
- v(vector) : 参数用数组
- p(path) : 有p自动搜索环境变量PATH
- e(env) : 表示自己维护环境变量

函数名	参数格式	是否带路径	是否使用当前环境变量
exec	列表	不是	是
execp	列表	是	是
execle	列表	不是	不是, 须自己组装环境变量
execv	数组	不是	是
execvp	数组	是	是
execve	数组	不是	不是, 须自己组装环境变量

exec调用举例如下:

```
#include <unistd.h>

int main()
{
    char *const argv[] = {"ps", "-ef", NULL};
    char *const envp[] = {"PATH=/bin:/usr/bin", "TERM=console", NULL};

    execl("/bin/ps", "ps", "-ef", NULL);
    // 带p的, 可以使用环境变量PATH, 无需写全路径
    execlp("ps", "ps", "-ef", NULL);

    // 带e的, 需要自己组装环境变量
    execle("ps", "ps", "-ef", NULL, envp);

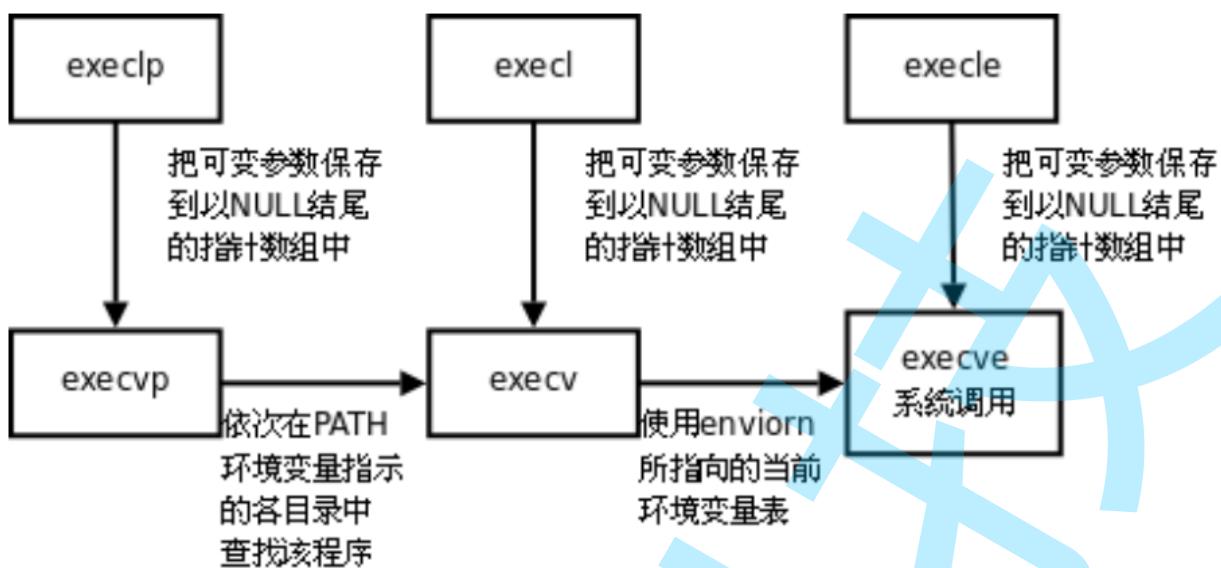
    execv("/bin/ps", argv);
    // 带p的, 可以使用环境变量PATH, 无需写全路径
    execvp("ps", argv);

    // 带e的, 需要自己组装环境变量
    execve("/bin/ps", argv, envp);

    exit(0);
}
```

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。

下图exec函数族一个完整的例子:

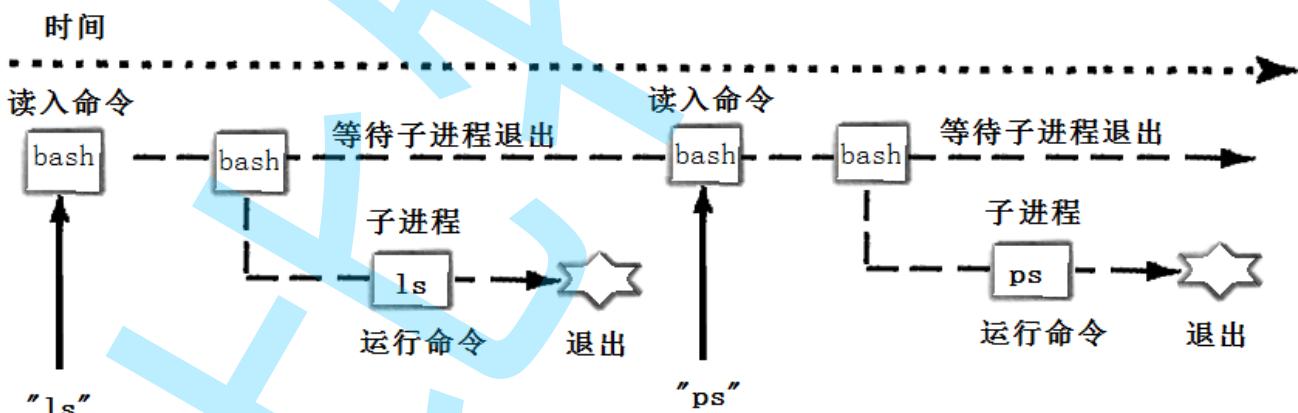


## 我们可以综合前面的知识，做一个简易的shell

考虑下面这个与shell典型的互动：

```
[root@localhost epoll]# ls
client.cpp  readme.md  server.cpp  utility.h
[root@localhost epoll]# ps
 PID TTY      TIME CMD
 3451 pts/0    00:00:00 bash
 3514 pts/0    00:00:00 ps
```

用下图的时间轴来表示事件的发生次序。其中时间从左向右。shell由标识为sh的方块代表，它随着时间的流逝从左向右移动。shell从用户读入字符串"ls"。shell建立一个新的进程，然后在那个进程中运行ls程序并等待那个进程结束。



然后shell读取新的一行输入，建立一个新的进程，在这个进程中运行程序 并等待这个进程结束。  
所以要写一个shell，需要循环以下过程：

1. 获取命令行
2. 解析命令行
3. 建立一个子进程 (fork)
4. 替换子进程 (execvp)

## 5. 父进程等待子进程退出 (wait)

根据这些思路，和我们前面学的技术，就可以自己来实现一个shell了。

实现代码：

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

#define MAX_CMD 1024
char command[MAX_CMD];

int do_face()
{
    memset(command, 0x00, MAX_CMD);
    printf("minishell$ ");
    fflush(stdout);
    if (scanf("%[^\\n]*%c", command) == 0) {
        getchar();
        return -1;
    }
    return 0;
}

char **do_parse(char *buff)
{
    int argc = 0;
    static char *argv[32];
    char *ptr = buff;

    while(*ptr != '\0') {
        if (!isspace(*ptr)) {
            argv[argc++] = ptr;
            while((!isspace(*ptr)) && (*ptr) != '\0') {
                ptr++;
            }
        } else {
            while(isspace(*ptr)) {
                *ptr = '\0';
                ptr++;
            }
        }
    }
    argv[argc] = NULL;
    return argv;
}

int do_exec(char *buff)
{
    char **argv = {NULL};

    int pid = fork();
```

```

if (pid == 0) {
    argv = do_parse(buff);
    if (argv[0] == NULL) {
        exit(-1);
    }
    execvp(argv[0], argv);
} else {
    waitpid(pid, NULL, 0);
}

return 0;
}
int main(int argc, char *argv[])
{
    while(1) {
        if (do_face() < 0)
            continue;
        do_exec(command);
    }
    return 0;
}

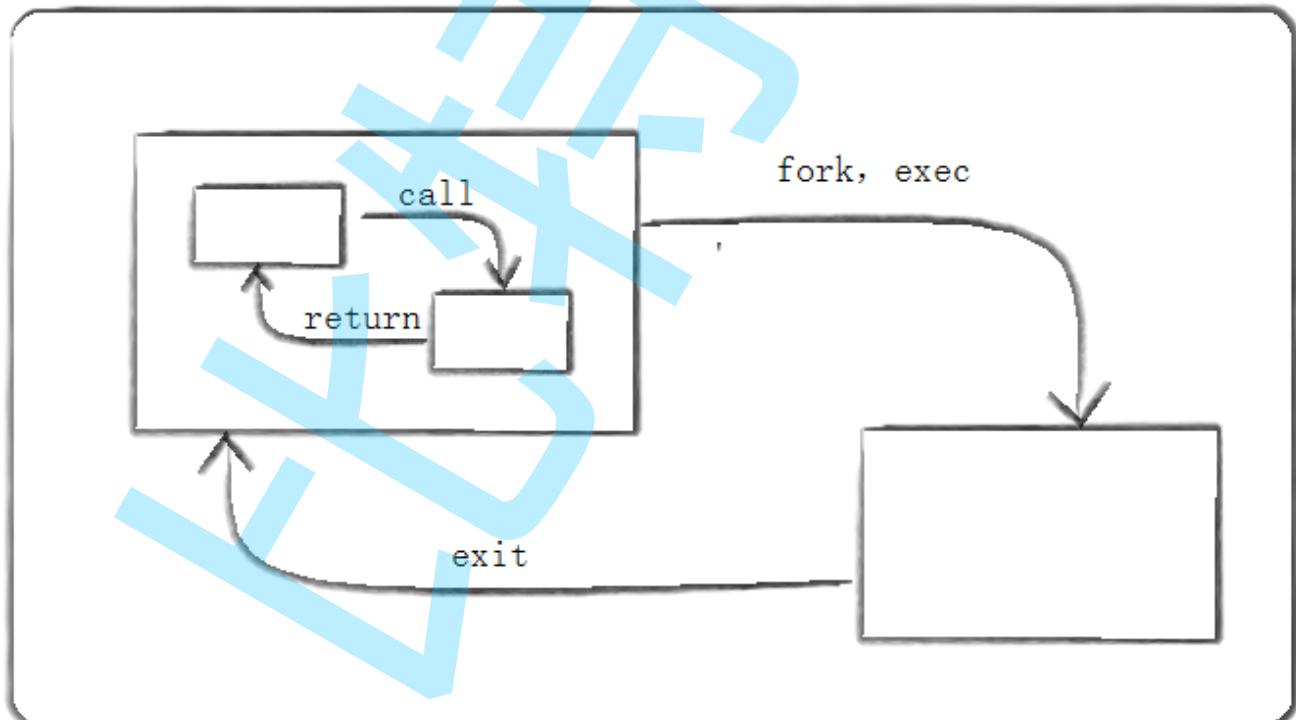
```

在继续学习新知识前，我们来思考函数和进程之间的相似性

exec/exit就像call/return

一个C程序有很多函数组成。一个函数可以调用另外一个函数，同时传递给它一些参数。被调用的函数执行一定的操作，然后返回一个值。每个函数都有他的局部变量，不同的函数通过call/return系统进行通信。

这种通过参数和返回值在拥有私有数据的函数间通信的模式是结构化程序设计的基础。Linux鼓励将这种应用于程序之内的模式扩展到程序之间。如下图



一个C程序可以fork/exec另一个程序，并传给它一些参数。这个被调用的程序执行一定的操作，然后通过exit(n)来返回值。调用它的进程可以通过wait (&ret) 来获取exit的返回值。

## 本节重点

- 复习C文件IO相关操作
- 认识文件相关系统调用接口
- 认识文件描述符,理解重定向
- 对比fd和FILE, 理解系统调用和库函数的关系
- 理解文件系统中inode的概念
- 认识软硬链接, 对比区别
- 认识动态静态库, 学会结合gcc选项, 制作动静态库

## 先来段代码回顾C文件接口

### hello.c写文件

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

int main()
{
    FILE *fp = fopen("myfile", "w");
    if(!fp){
        printf("fopen error!\n");
    }

    const char *msg = "hello bit!\n";
    int count = 5;
    while(count--){
        fwrite(msg, strlen(msg), 1, fp);
    }

    fclose(fp);

    return 0;
}
```

### hello.c读文件

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

int main()
{
    FILE *fp = fopen("myfile", "r");
    if(!fp){
        printf("fopen error!\n");
    }

    char buf[1024];
    const char *msg = "hello bit!\n";
```

```

while(1){
    //注意返回值和参数，此处有坑，仔细查看man手册关于该函数的说明
    ssize_t s = fread(buf, 1, strlen(msg), fp);
    if(s > 0){
        buf[s] = 0;
        printf("%s", buf);
    }
    if(feof(fp)){
        break;
    }
}

fclose(fp);
return 0;
}

```

## 输出信息到显示器，你有哪些方法

```

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

int main()
{
    const char *msg = "hello fwrite\n";
    fwrite(msg, strlen(msg), 1, stdout);

    printf("hello printf\n");
    fprintf(stdout, "hello fprintf\n");
    return 0;
}

```

## stdin & stdout & stderr

- C默认会打开三个输入输出流，分别是stdin, stdout, stderr
- 仔细观察发现，这三个流的类型都是FILE\*, fopen返回值类型，文件指针

## 总结

- 打开文件的方式

r	Open text file for reading. The stream is positioned at the beginning of the file.
r+	Open for reading and writing. The stream is positioned at the beginning of the file.
w	Truncate(缩短) file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
w+	Open for reading and writing.

The file is created if it does not exist, otherwise it is truncated.  
The stream is positioned at the beginning of the file.

- a Open for appending (writing at end of file).  
The file is created if it does not exist.  
The stream is positioned at the end of the file.

- a+ Open for reading and appending (writing at end of file).  
The file is created if it does not exist. The initial file position  
for reading is at the beginning of the file,  
but output is always appended to the end of the file.

如上，是我们之前学的文件相关操作。还有 `fseek` `ftell` `rewind` 的函数，在C部分已经有所涉猎，请同学们自行复习。

## 系统文件I/O

操作文件，除了上述C接口（当然，C++也有接口，其他语言也有），我们还可以采用系统接口来进行文件访问，先来直接以代码的形式，实现和上面一模一样的代码：

hello.c 写文件：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main()
{
    umask(0);
    int fd = open("myfile", O_WRONLY|O_CREAT, 0644);
    if(fd < 0){
        perror("open");
        return 1;
    }

    int count = 5;
    const char *msg = "hello bit!\n";
    int len = strlen(msg);

    while(count--){
        write(fd, msg, len); //fd: 后面讲, msg: 缓冲区首地址, len: 本次读取, 期望写入多少个字节的数据
    }

    close(fd);
    return 0;
}
```

## hello.c读文件

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int fd = open("myfile", O_RDONLY);
    if(fd < 0){
        perror("open");
        return 1;
    }

    const char *msg = "hello bit!\n";
    char buf[1024];
    while(1){
        ssize_t s = read(fd, buf, strlen(msg)); //类比write
        if(s > 0){
            printf("%s", buf);
        }else{
            break;
        }
    }

    close(fd);
    return 0;
}
```

## 接口介绍

open man open

```
#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);
```

pathname: 要打开或创建的目标文件

flags: 打开文件时, 可以传入多个参数选项, 用下面的一个或者多个常量进行“或”运算, 构成flags。

参数:

O\_RDONLY: 只读打开

O\_WRONLY: 只写打开

O\_RDWR : 读, 写打开

这三个常量, 必须指定一个且只能指定一个

O\_CREAT : 若文件不存在, 则创建它。需要使用mode选项, 来指明新文件的访问权限

O\_APPEND: 追加写

返回值：

成功：新打开的文件描述符

失败： -1

mode\_t理解：直接 man 手册，比什么都清楚。

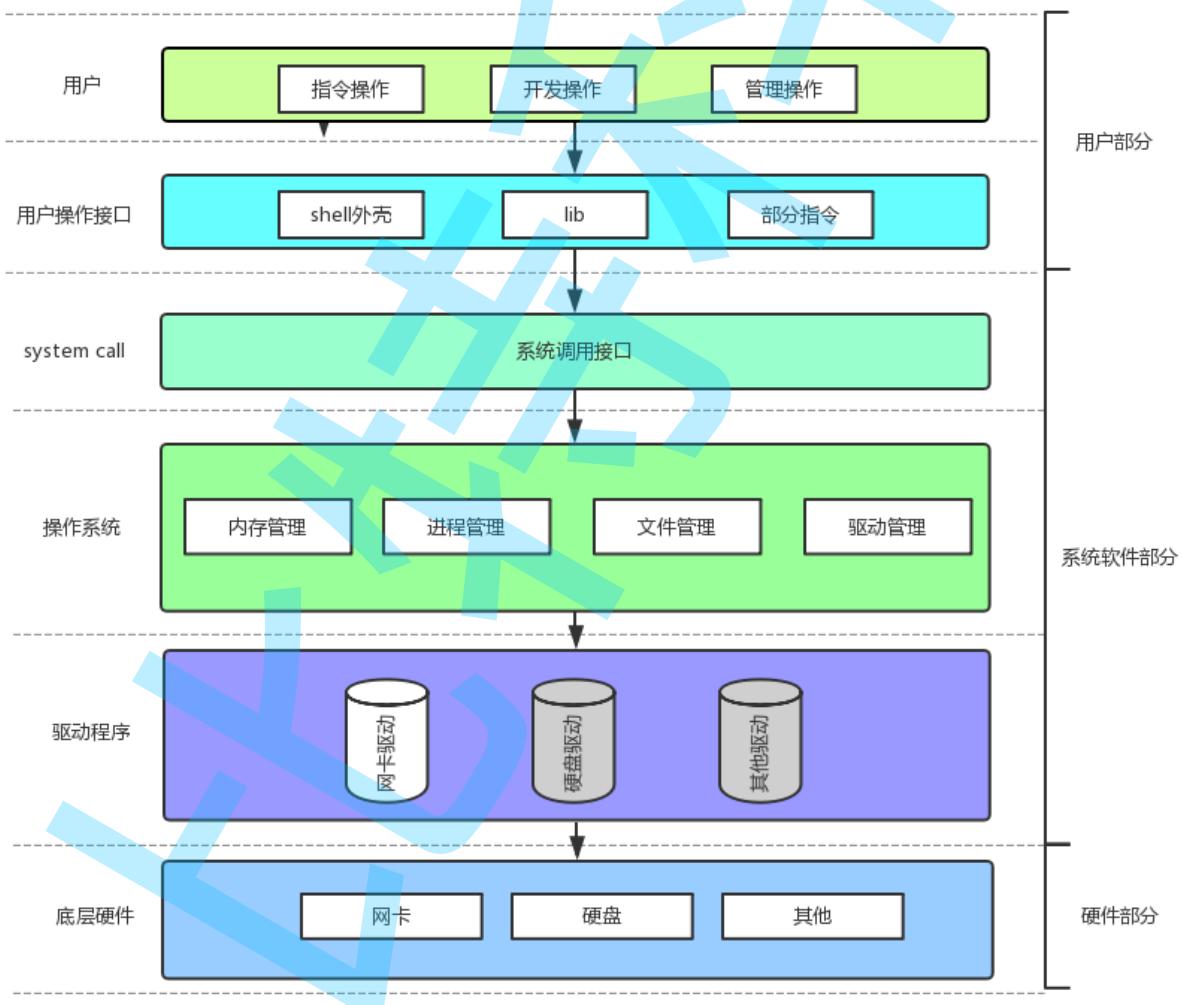
open 函数具体使用哪个，和具体应用场景相关，如目标文件不存在，需要open创建，则第三个参数表示创建文件的默认权限，否则，使用两个参数的open。

`write` `read` `close` `lseek`，类比C文件相关接口。

## open函数返回值

在认识返回值之前，先来认识一下两个概念：`系统调用` 和 `库函数`

- 上面的 `fopen` `fclose` `fread` `fwrite` 都是C标准库当中的函数，我们称之为库函数（libc）。
- 而，`open` `close` `read` `write` `lseek` 都属于系统提供的接口，称之为系统调用接口
- 回忆一下我们讲操作系统概念时，画的一张图



系统调用接口和库函数的关系，一目了然。

所以，可以认为，f#系列的函数，都是对系统调用的封装，方便二次开发。

## 文件描述符fd

- 通过对open函数的学习，我们知道了文件描述符就是一个小整数

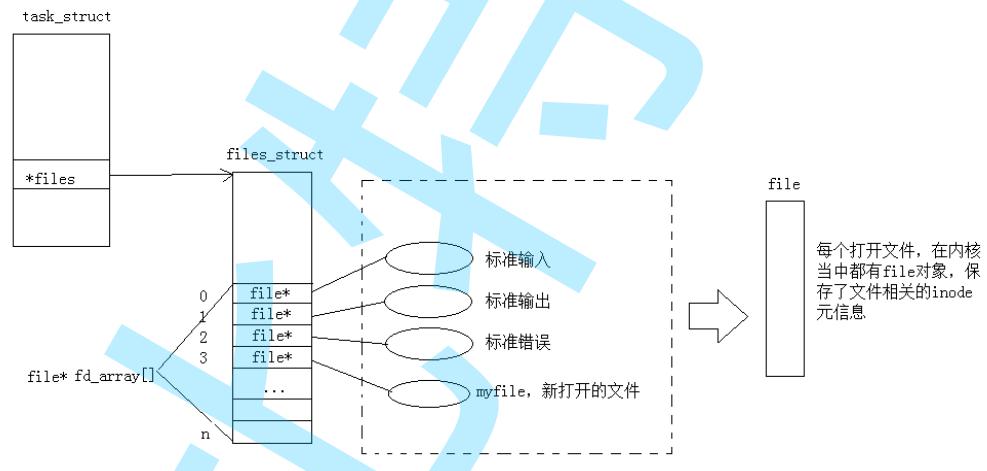
## 0 & 1 & 2

- Linux进程默认情况下会有3个缺省打开的文件描述符，分别是标准输入0，标准输出1，标准错误2。
- 0,1,2对应的物理设备一般是：键盘，显示器，显示器

所以输入输出还可以采用如下方式：

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

int main()
{
    char buf[1024];
    ssize_t s = read(0, buf, sizeof(buf));
    if(s > 0){
        buf[s] = 0;
        write(1, buf, strlen(buf));
        write(2, buf, strlen(buf));
    }
    return 0;
}
```



而现在知道，文件描述符就是从0开始的小整数。当我们打开文件时，操作系统在内存中要创建相应的数据结构来描述目标文件。于是就有了file结构体。表示一个已经打开的文件对象。而进程执行open系统调用，所以必须让进程和文件关联起来。每个进程都有一个指针`*files`, 指向一张表`files_struct`, 该表最重要的部分就是包涵一个指针数组，每个元素都是一个指向打开文件的指针！所以，本质上，文件描述符就是该数组的下标。所以，只要拿着文件描述符，就可以找到对应的文件

## 文件描述符的分配规则

直接看代码：

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

int main()
{
    int fd = open("myfile", O_RDONLY);
    if(fd < 0){
        perror("open");
        return 1;
    }
    printf("fd: %d\n", fd);

    close(fd);
    return 0;
}
```

输出发现是 fd: 3

关闭0或者2，在看

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

int main()
{
    close(0);
    //close(2);
    int fd = open("myfile", O_RDONLY);
    if(fd < 0){
        perror("open");
        return 1;
    }
    printf("fd: %d\n", fd);

    close(fd);
    return 0;
}
```

发现结果是： fd: 0 或者 fd 2 可见，文件描述符的分配规则：在files\_struct数组当中，找到当前没有被使用的最小的一个下标，作为新的文件描述符。

## 重定向

那如果关闭1呢？看代码：

```

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

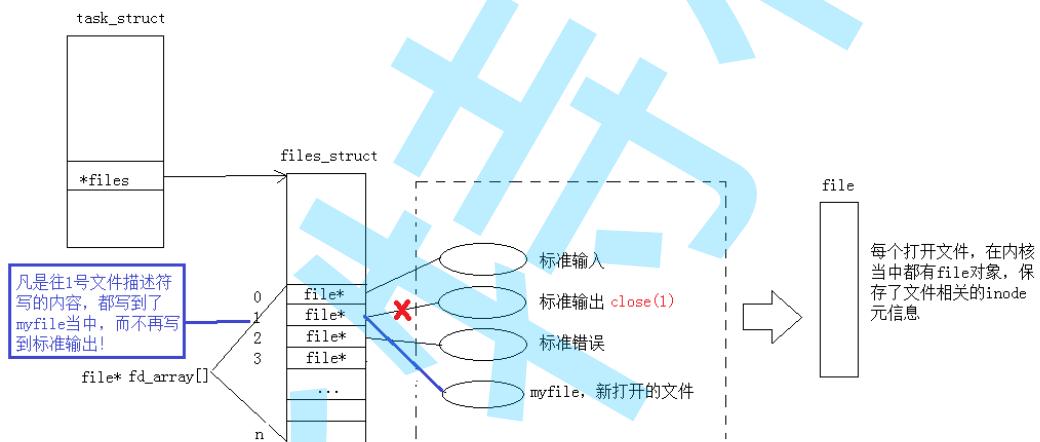
int main()
{
    close(1);
    int fd = open("myfile", O_WRONLY|O_CREAT, 00644);
    if(fd < 0){
        perror("open");
        return 1;
    }
    printf("fd: %d\n", fd);
    fflush(stdout);

    close(fd);
    exit(0);
}

```

此时，我们发现，本来应该输出到显示器上的内容，输出到了文件 `myfile` 当中，其中，`fd = 1`。这种现象叫做输出重定向。常见的重定向有`>`, `>>`, `<`

那重定向的本质是什么呢？



## 使用 dup2 系统调用

函数原型如下：

```

#include <unistd.h>

int dup2(int oldfd, int newfd);

```

示例代码

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

int main() {
    int fd = open("./log", O_CREAT | O_RDWR);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    close(1);
    dup2(fd, 1);
    for (;;) {
        char buf[1024] = {0};
        ssize_t read_size = read(0, buf, sizeof(buf) - 1);
        if (read_size < 0) {
            perror("read");
            break;
        }
        printf("%s", buf);
        fflush(stdout);
    }
    return 0;
}
```

### 例子1. 在minishell中添加重定向功能:

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

# define MAX_CMD 1024
char command[MAX_CMD];

int do_face()
{
    memset(command, 0x00, MAX_CMD);
    printf("minishell$ ");
    fflush(stdout);
    if (scanf("%[^\\n]*c", command) == 0) {
        getchar();
        return -1;
    }
    return 0;
}
char **do_parse(char *buff)
{
    int argc = 0;
    static char *argv[32];
    char *ptr = buff;
```

```
while(*ptr != '\0') {
    if (!isspace(*ptr)) {
        argv[argc++] = ptr;
        while((!isspace(*ptr)) && (*ptr) != '\0') {
            ptr++;
        }
    } else {
        while(isspace(*ptr)) {
            *ptr = '\0';
            ptr++;
        }
    }
}
argv[argc] = NULL;
return argv;
}

int do_redirect(char *buff)
{
    char *ptr = buff, *file = NULL;
    int type = 0, fd, redirect_type = -1;
    while(*ptr != '\0') {
        if (*ptr == '>') {
            *ptr++ = '\0';
            redirect_type++;
            if (*ptr == '>') {
                *ptr++ = '\0';
                redirect_type++;
            }
        }
        while(isspace(*ptr)) {
            ptr++;
        }
        file = ptr;
        while((!isspace(*ptr)) && *ptr != '\0') {
            ptr++;
        }
        *ptr = '\0';
        if (redirect_type == 0) {
            fd = open(file, O_CREAT|O_TRUNC|O_WRONLY, 0664);
        } else {
            fd = open(file, O_CREAT|O_APPEND|O_WRONLY, 0664);
        }
        dup2(fd, 1);
    }
    ptr++;
}
return 0;
}

int do_exec(char *buff)
{
    char **argv = {NULL};

    int pid = fork();
```

```

if (pid == 0) {
    do_redirect(buff);
    argv = do_parse(buff);
    if (argv[0] == NULL) {
        exit(-1);
    }
    execvp(argv[0], argv);
} else {
    waitpid(pid, NULL, 0);
}

return 0;
}

int main(int argc, char *argv[])
{
    while(1) {
        if (do_face() < 0)
            continue;
        do_exec(command);
    }
    return 0;
}

```

printf是C库当中的IO函数，一般往 `stdout` 中输出，但是`stdout`底层访问文件的时候，找的还是`fd:1`，但此时，`fd:1`下标所表示内容，已经变成了myfile的地址，不再是显示器文件的地址，所以，输出的任何消息都会往文件中写入，进而完成输出重定向。那追加和输入重定向如何完成呢？请同学们自行研究。

## FILE

- 因为IO相关函数与系统调用接口对应，并且库函数封装系统调用，所以本质上，访问文件都是通过`fd`访问的。
- 所以C库当中的FILE结构体内部，必定封装了`fd`。

来段代码研究一下：

```

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

int main()
{
    const char *msg0="hello printf\n";
    const char *msg1="hello fwrite\n";
    const char *msg2="hello write\n";

    printf("%s", msg0);
    fwrite(msg1, strlen(msg0), 1, stdout);
    write(1, msg2, strlen(msg2));

    fork();

    return 0;
}

```

```
}
```

运行出结果：

```
hello printf  
hello fwrite  
hello write
```

但如果对进程实现输出重定向呢？`./hello > file`，我们发现结果变成了：

```
hello write  
hello printf  
hello fwrite  
hello printf  
hello fwrite
```

我们发现 `printf` 和 `fwrite`（库函数）都输出了2次，而 `write` 只输出了一次（系统调用）。为什么呢？肯定和 `fork` 有关！

- 一般C库函数写入文件时是全缓冲的，而写入显示器是行缓冲。
- `printf` `fwrite` 库函数会自带缓冲区（进度条例子就可以说明），当发生重定向到普通文件时，数据的缓冲方式由行缓冲变成了全缓冲。
- 而我们放在缓冲区中的数据，就不会被立即刷新，甚至 `fork` 之后。
- 但是进程退出之后，会统一刷新，写入文件当中。
- 但是 `fork` 的时候，父子数据会发生写时拷贝，所以当你父进程准备刷新的时候，子进程也就有了同样的数据，随即产生两份数据。
- `write` 没有变化，说明没有所谓的缓冲。

综上：`printf` `fwrite` 库函数会自带缓冲区，而 `write` 系统调用没有带缓冲区。另外，我们这里所说的缓冲区，都是用户级缓冲区。其实为了提升整机性能，OS也会提供相关内核级缓冲区，不过不再我们讨论范围之内。

那这个缓冲区谁提供呢？`printf` `fwrite` 是库函数，`write` 是系统调用，库函数在系统调用的“上层”，是对系统调用的“封装”，但是 `write` 没有缓冲区，而 `printf` `fwrite` 有，足以说明，该缓冲区是二次加上的，又因为是 C，所以由 C 标准库提供。

如果有兴趣，可以看看 FILE 结构体：

```
typedef struct _IO_FILE FILE; 在/usr/include/stdio.h
```

```
在/usr/include/libio.h  
struct _IO_FILE {  
    int _flags;          /* High-order word is _IO_MAGIC; rest is flags. */  
#define _IO_file_flags _flags  
  
    //缓冲区相关  
    /* The following pointers correspond to the C++ streambuf protocol. */  
    /* Note: Tk uses the _IO_read_ptr and _IO_read_end fields directly. */  
    char* _IO_read_ptr;    /* Current read pointer */  
    char* _IO_read_end;   /* End of get area. */  
    char* _IO_read_base;  /* Start of putback+get area. */  
  
    char* _IO_write_base; /* Start of put area. */
```

```
char* _IO_write_ptr; /* Current put pointer. */
char* _IO_write_end; /* End of put area. */
char* _IO_buf_base; /* Start of reserve area. */
char* _IO_buf_end; /* End of reserve area. */
/* The following fields are used to support backing up and undo. */
char *_IO_save_base; /* Pointer to start of non-current get area. */
char *_IO_backup_base; /* Pointer to first valid character of backup area */
char *_IO_save_end; /* Pointer to end of non-current get area. */

struct _IO_marker *_markers;

struct _IO_FILE *_chain;

int _fileno; //封装的文件描述符

#if 0
    int _blksize;
#else
    int _flags2;
#endif
_IO_offset_t _old_offset; /* This used to be _offset but it's too small. */

#define __HAVE_COLUMN /* temporary */
/* 1+column number of pbase(); 0 is unknown. */
unsigned short _cur_column;
signed char _vtable_offset;
char _shortbuf[1];

/* char* _save_gptr; char* _save_egptr; */

_IO_lock_t *_lock;
#endif _IO_USE_OLD_IO_FILE
};
```

## 理解文件系统

我们使用ls -l的时候看到的除了看到文件名，还看到了文件元数据。

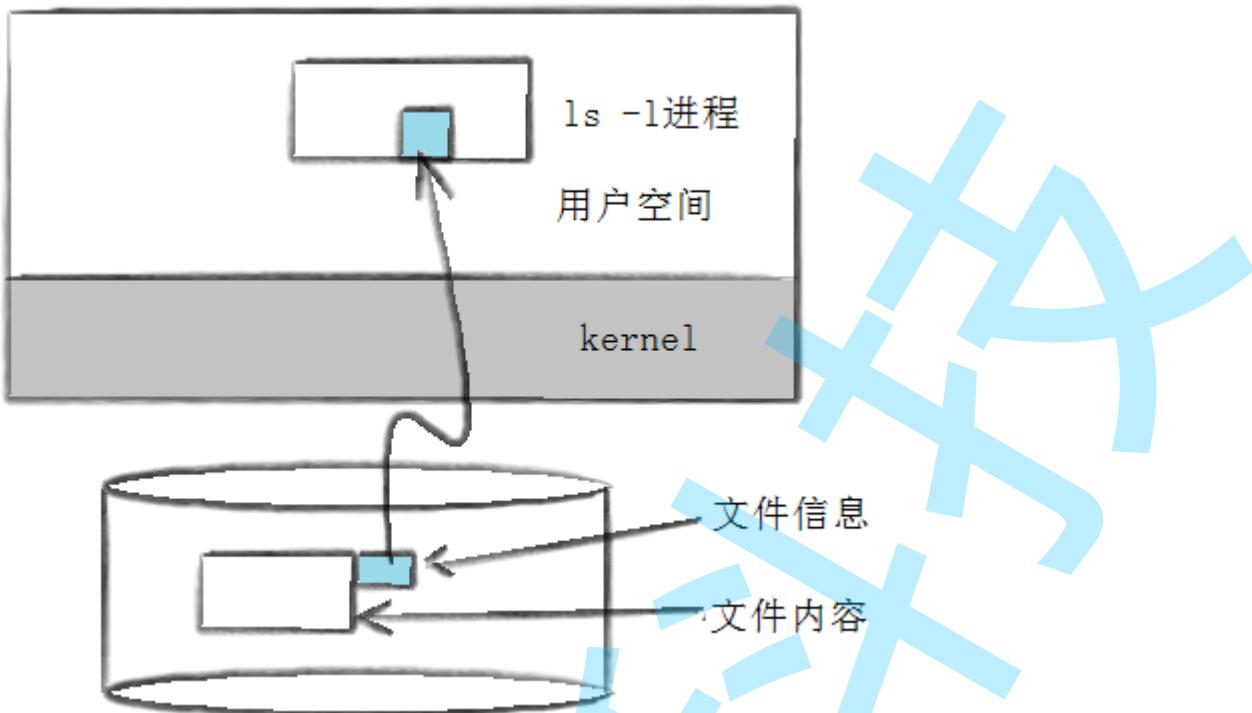
```
[root@localhost linux]# ls -l
总用量 12
-rwxr-xr-x. 1 root root 7438 "9月 13 14:56" a.out
-rw-r--r--. 1 root root 654 "9月 13 14:56" test.c
```

每行包含7列：

- 模式
- 硬链接数
- 文件所有者
- 组
- 大小
- 最后修改时间

- 文件名

ls -l读取存储在磁盘上的文件信息，然后显示出来



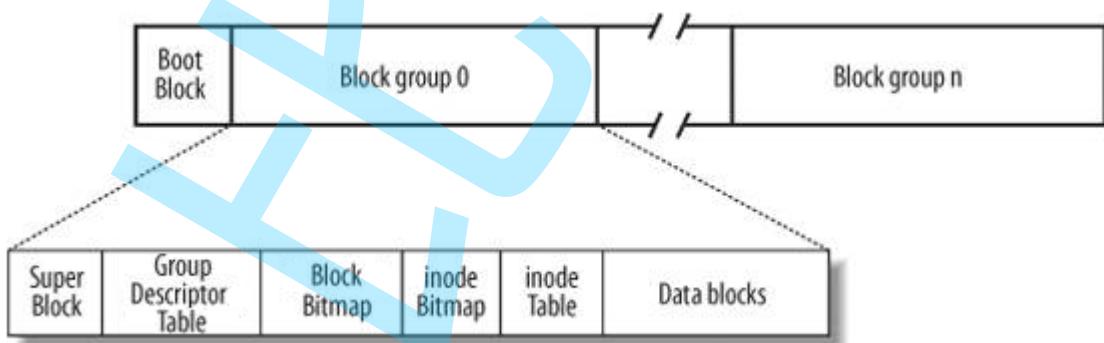
其实这个信息除了通过这种方式来读取，还有一个stat命令能够看到更多信息

```
[root@localhost linux]# stat test.c
  File: "test.c"
  Size: 654          Blocks: 8          IO Block: 4096   普通文件
Device: 802h/2050d  Inode: 263715      Links: 1
Access: (0644/-rw-r--r--) Uid: (    0/    root)  Gid: (    0/    root)
Access: 2017-09-13 14:56:57.059012947 +0800
Modify: 2017-09-13 14:56:40.067012944 +0800
Change: 2017-09-13 14:56:40.069012948 +0800
```

上面的执行结果有几个信息需要解释清楚

### inode

为了能解释清楚inode我们先简单了解一下文件系统



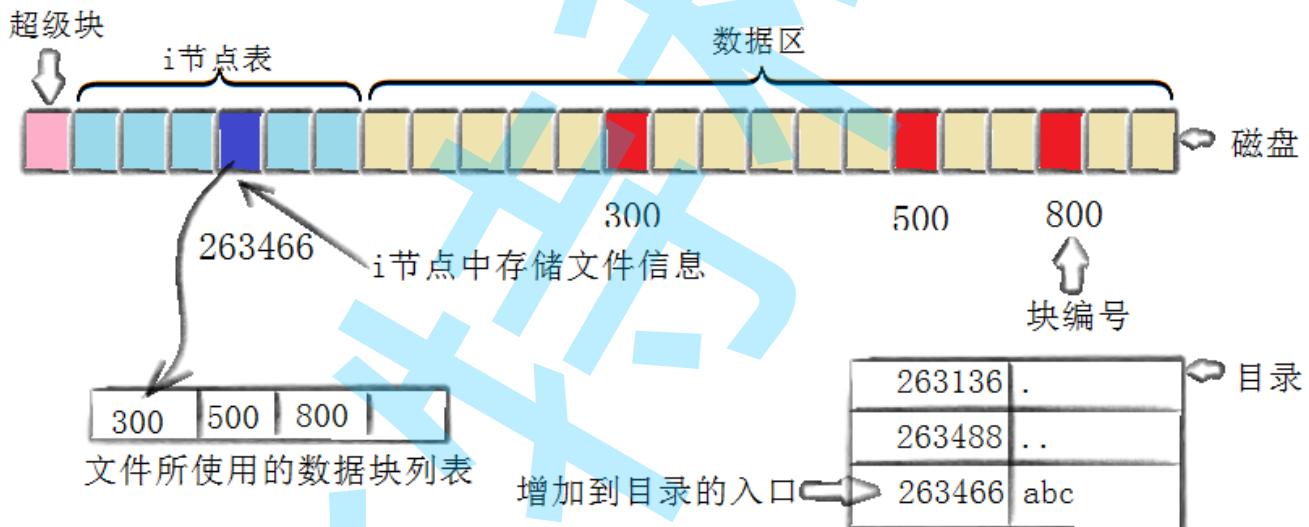
Linux ext2文件系统，上图为磁盘文件系统图（内核内存映像肯定有所不同），磁盘是典型的块设备，硬盘分区被划分为一个个的block。一个block的大小是由格式化的时候确定的，并且不可以更改。例如mke2fs的-b选项可以设定block大小为1024、2048或4096字节。而上图中启动块（Boot Block）的大小是确定的，

- Block Group: ext2文件系统会根据分区的大小划分为数个Block Group。而每个Block Group都有着相同的结构组成。政府管理各区的例子
- 超级块 (Super Block) : 存放文件系统本身的结构信息。记录的信息主要有: block 和 inode的总量, 未使用的block和inode的数量, 一个block和inode的大小, 最近一次挂载的时间, 最近一次写入数据的时间, 最近一次检验磁盘的时间等其他文件系统的相关信息。Super Block的信息被破坏, 可以说整个文件系统结构就被破坏了
- GDT, Group Descriptor Table: 块组描述符, 描述块组属性信息, 有兴趣的同学可以在了解一下
- 块位图 (Block Bitmap) : Block Bitmap中记录着Data Block中哪个数据块已经被占用, 哪个数据块没有被占用
- inode位图 (inode Bitmap) : 每个bit表示一个inode是否空闲可用。
- i节点表:存放文件属性 如文件大小, 所有者, 最近修改时间等
- 数据区: 存放文件内容

将属性和数据分开存放的想法看起来很简单, 但实际上是如何工作的呢? 我们通过touch一个新文件来看看如何工作。

```
[root@localhost linux]# touch abc
[root@localhost linux]# ls -i abc
263466 abc
```

为了说明问题, 我们将上图简化:



创建一个新文件主要有一下4个操作:

1. 存储属性  
内核先找到一个空闲的i节点 (这里是263466)。内核把文件信息记录到其中。
2. 存储数据  
该文件需要存储在三个磁盘块, 内核找到了三个空闲块: 300,500, 800。将内核缓冲区的第一块数据复制到300, 下一块复制到500, 以此类推。
3. 记录分配情况  
文件内容按顺序300,500,800存放。内核在inode上的磁盘分布区记录了上述块列表。
4. 添加文件名到目录

新的文件名abc。linux如何在当前的目录中记录这个文件? 内核将入口 (263466, abc) 添加到目录文件。文件名和inode之间的对应关系将文件名和文件的内容及属性连接起来。

## 理解硬链接

我们看到，真正找到磁盘上文件的并不是文件名，而是inode。 其实在linux中可以让多个文件名对应于同一个inode。 [root@localhost linux]# touch abc [root@localhost linux]# ln abc def [root@localhost linux]# ls -li abc def 263466 abc 263466 def

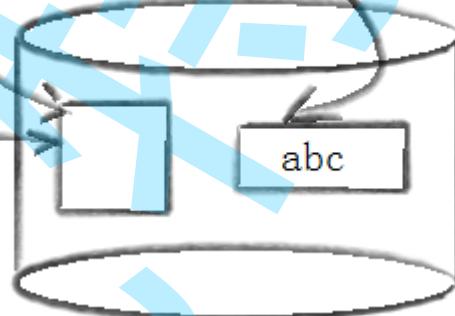
- abc和def的链接状态完全相同，他们被称为指向文件的硬链接。内核记录了这个连接数，inode 263466 的硬连接数为2。
- 我们在删除文件时干了两件事情：1.在目录中将对应的记录删除，2.将硬连接数-1，如果为0，则将对应的磁盘释放。

## 软链接

硬链接是通过inode引用另外一个文件，软链接是通过名字引用另外一个文件，在shell中的做法

```
263563 -rw-r--r--. 2 root root 0 9月 15 17:45 abc
261678 lrwxrwxrwx. 1 root root 3 9月 15 17:53 abc.s -> abc
263563 -rw-r--r--. 2 root root 0 9月 15 17:45 def
```

263136	.
263488	..
263563	abc
263563	def
261678	abc.s



## aclm

下面解释一下文件的三个时间：

- Access 最后访问时间
- Modify 文件内容最后修改时间
- Change 属性最后修改时间

## 动态库和静态库

### 静态库与动态库

- 静态库 (.a) : 程序在编译链接的时候把库的代码链接到可执行文件中。程序运行的时候将不再需要静态库
- 动态库 (.so) : 程序在运行的时候才去链接动态库的代码，多个程序共享使用库的代码。
- 一个与动态库链接的可执行文件仅仅包含它用到的函数入口地址的一个表，而不是外部函数所在目标文件的整个机器码
- 在可执行文件开始运行以前，外部函数的机器码由操作系统从磁盘上的该动态库中复制到内存中，这个过程称为动态链接 (dynamic linking)
- 动态库可以在多个程序间共享，所以动态链接使得可执行文件更小，节省了磁盘空间。操作系统采用虚拟内存机制允许物理内存中的一份动态库被要用到该库的所有进程共用，节省了内存和磁盘空间。

#### 测试程序

```
//////////add.h///////////
#ifndef __ADD_H__
#define __ADD_H__
int add(int a, int b);
#endif // __ADD_H__
//////////add.c/////////
#include "add.h"
int add(int a, int b)
{
    return a + b;
}

//////////sub.h/////////
#ifndef __SUB_H__
#define __SUB_H__
int sub(int a, int b);
#endif // __SUB_H__
//////////sub.c/////////
#include "add.h"
int sub(int a, int b)
{
    return a - b;
}

//////////main.c/////////
#include <stdio.h>
#include "add.h"
#include "sub.h"

int main( void )
{
    int a = 10;
    int b = 20;
    printf("add(10, 20)=%d\n", a, b, add(a, b));
    a = 100;
    b = 20;
    printf("sub(%d,%d)=%d\n", a, b, sub(a, b));
}
```

#### 生成静态库

```
[root@localhost linux]# ls  
add.c add.h main.c sub.c sub.h  
[root@localhost linux]# gcc -c add.c -o add.o  
[root@localhost linux]# gcc -c sub.c -o sub.o
```

#### 生成静态库

```
[root@localhost linux]# ar -rc libmymath.a add.o sub.o  
ar是gnu归档工具, rc表示(replace and create)
```

#### 查看静态库中的目录列表

```
[root@localhost linux]# ar -tv libmymath.a  
rw-r--r-- 0/0 1240 Sep 15 16:53 2017 add.o  
rw-r--r-- 0/0 1240 Sep 15 16:53 2017 sub.o
```

t:列出静态库中的文件

v:verbose 详细信息

```
[root@localhost linux]# gcc main.c -L. -lmymath
```

-L 指定库路径

-l 指定库名

测试目标文件生成后, 静态库删掉, 程序照样可以运行。

### 库搜索路径

- 从左到右搜索-L指定的目录。
- 由环境变量指定的目录 (LIBRARY\_PATH)
- 由系统指定的目录
  - /usr/lib
  - /usr/local/lib

### 生成动态库

- shared: 表示生成共享库格式
- fPIC: 产生位置无关码(position independent code)
- 库名规则: libxxx.so

示例: [root@localhost linux]# gcc -fPIC -c sub.c add.c [root@localhost linux]# gcc -shared -o libmymath.so \*.o [root@localhost linux]# ls add.c add.h add.o libmymath.so main.c sub.c sub.h sub.o

### 使用动态库

#### 编译选项

- I: 链接动态库, 只要库名即可(去掉lib以及版本号)
- L: 链接库所在的路径.

示例: gcc main.o -o main -L. -lhello

### 运行动态库

- 1、拷贝.so文件到系统共享库路径下, 一般指/usr/lib
- 2、更改LD\_LIBRARY\_PATH

```
[root@localhost linux]# export LD_LIBRARY_PATH=.
[root@localhost linux]# gcc main.c -lmymath
[root@localhost linux]# ./a.out
add(10, 20)=30
sub(100, 20)=80
```

- 3、ldconfig 配置/etc/ld.so.conf.d/，ldconfig更新

```
[root@localhost linux]# cat /etc/ld.so.conf.d/bit.conf
/root/tools/linux
[root@localhost linux]# ldconfig
```

## 使用外部库

系统中其实有很多库，它们通常由一组互相关联的用来完成某项常见工作的函数构成。比如用来处理屏幕显示情况的函数（ncurses库）

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double x = pow(2.0, 3.0);
    printf("The cubed is %f\n", x);
    return 0;
}
gcc -Wall calc.c -o calc -lm
```

-lm表示要链接libm.so或者libm.a库文件

库文件名称和引入库的名称

如：libc.so -> c库，去掉前缀lib，去掉后缀.so,.a

# 进程间通信

## 本节重点：

- 进程间通信介绍
- 管道
- 消息队列
- 共享内存
- 信号量

## 进程间通信介绍

### 进程间通信目的

- 数据传输：一个进程需要将它的数据发送给另一个进程
- 资源共享：多个进程之间共享同样的资源。
- 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
- 进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

### 进程间通信发展

- 管道
- System V进程间通信
- POSIX进程间通信

### 进程间通信分类

#### 管道

- 匿名管道pipe
- 命名管道

#### System V IPC

- System V 消息队列
- System V 共享内存
- System V 信号量

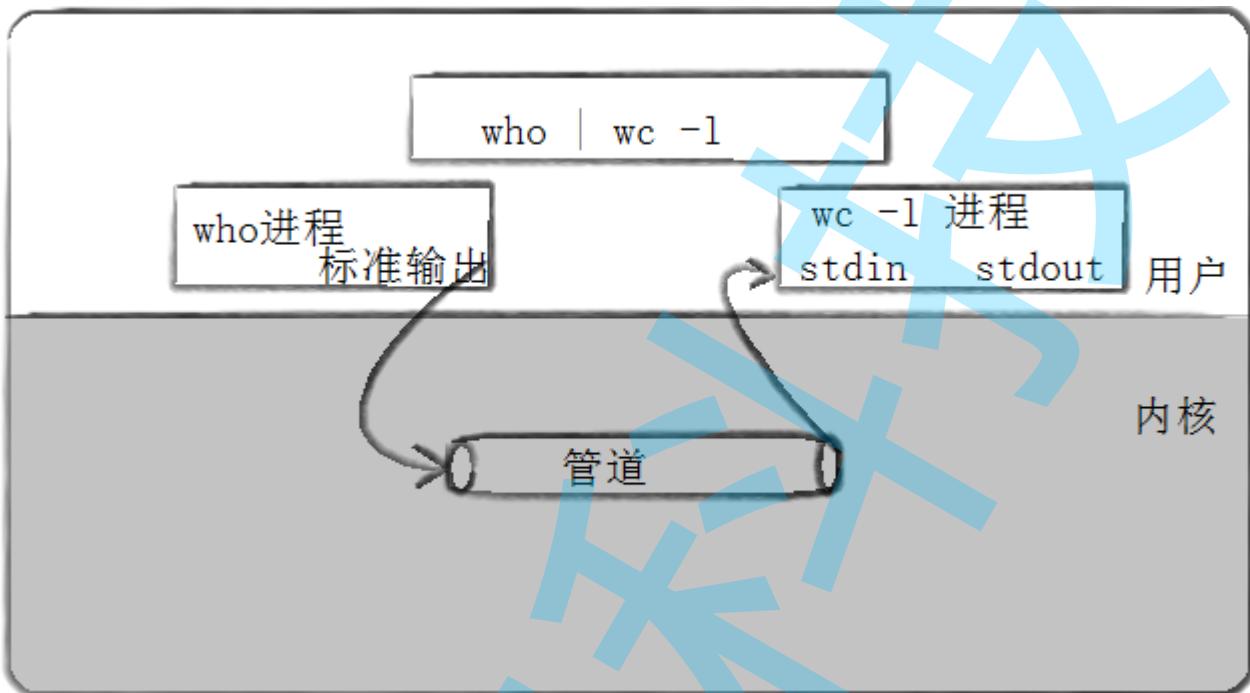
#### POSIX IPC

- 消息队列
- 共享内存
- 信号量
- 互斥量
- 条件变量
- 读写锁

# 管道

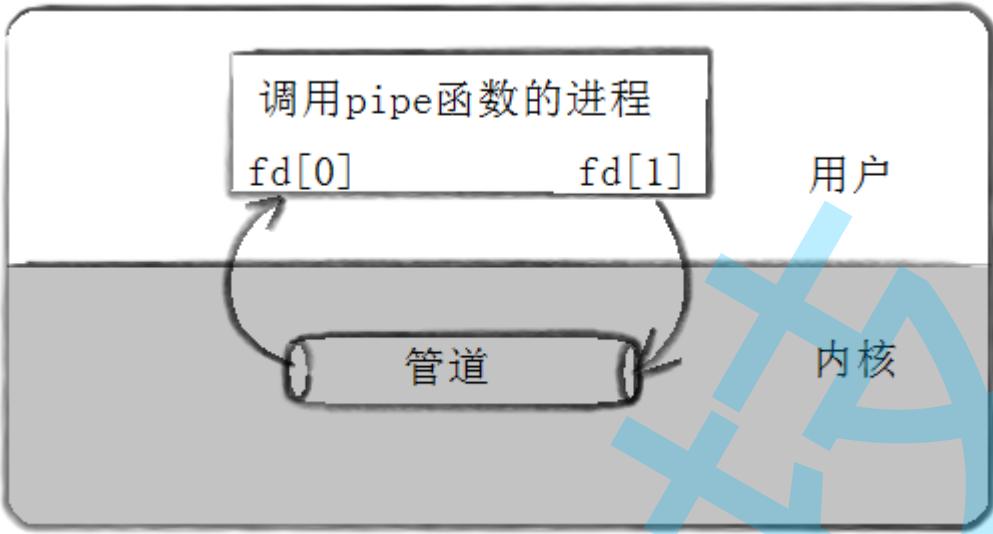
## 什么是管道

- 管道是Unix中最古老的进程间通信的形式。
- 我们把从一个进程连接到另一个进程的一个数据流称为一个“管道”



## 匿名管道

```
#include <unistd.h>
功能:创建一无名管道
原型
int pipe(int fd[2]);
参数
fd: 文件描述符数组,其中fd[0]表示读端, fd[1]表示写端
返回值:成功返回0, 失败返回错误代码
```



## 实例代码

例子：从键盘读取数据，写入管道，读取管道，写到屏幕

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

int main( void )
{
    int fds[2];
    char buf[100];
    int len;

    if ( pipe(fds) == -1 )
        perror("make pipe"), exit(1);

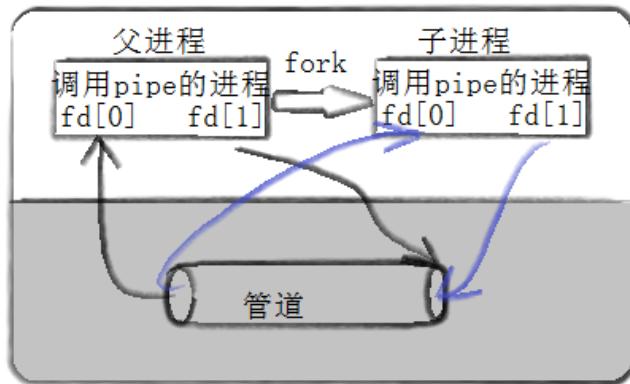
    // read from stdin
    while ( fgets(buf, 100, stdin) ) {
        len = strlen(buf);
        // write into pipe
        if ( write(fds[1], buf, len) != len ) {
            perror("write to pipe");
            break;
        }
        memset(buf, 0x00, sizeof(buf));

        // read from pipe
        if ( (len=read(fds[0], buf, 100)) == -1 ) {
            perror("read from pipe");
            break;
        }

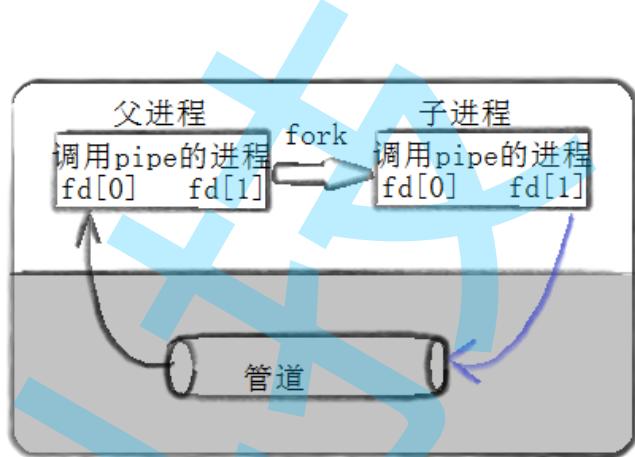
        // write to stdout
        if ( write(1, buf, len) != len ) {
            perror("write to stdout");
        }
    }
}
```

```
        break;  
    }  
}  
}
```

## 用fork来共享管道原理



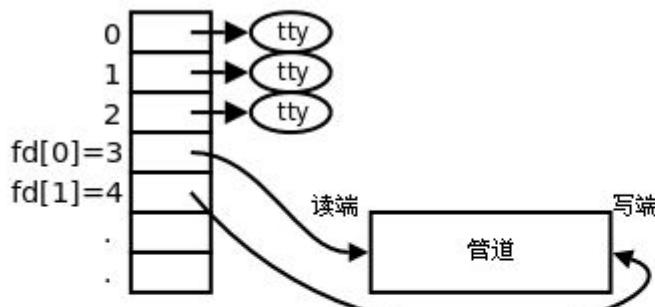
fork之后



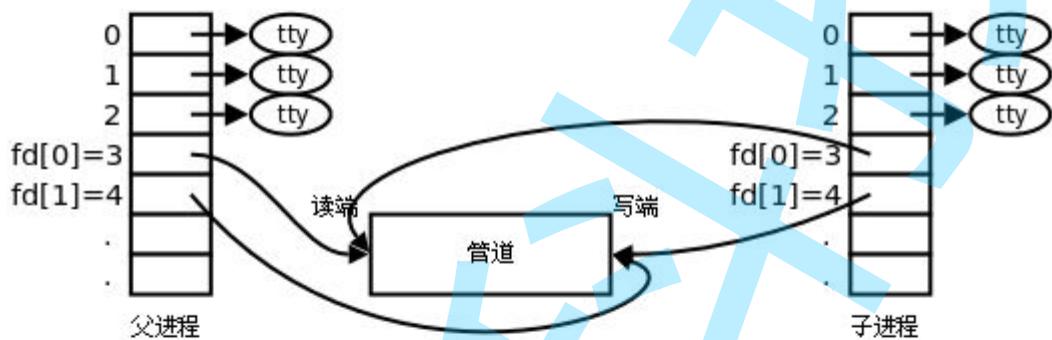
fork之后各自关掉不用的描述符

## 站在文件描述符角度-深度理解管道

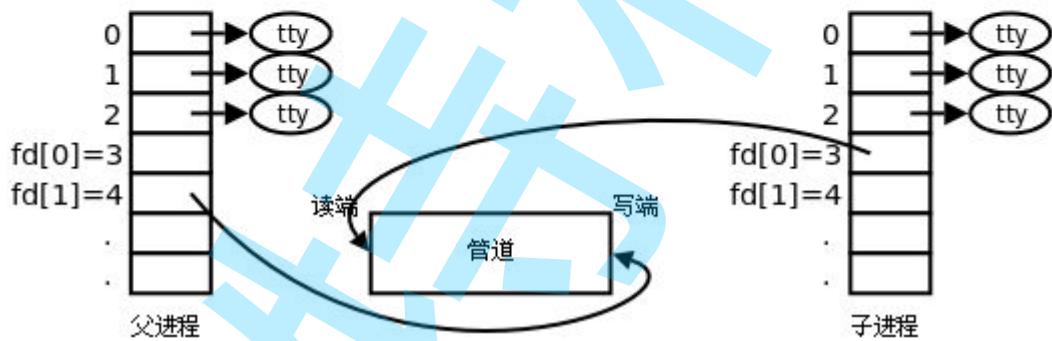
### 1. 父进程创建管道



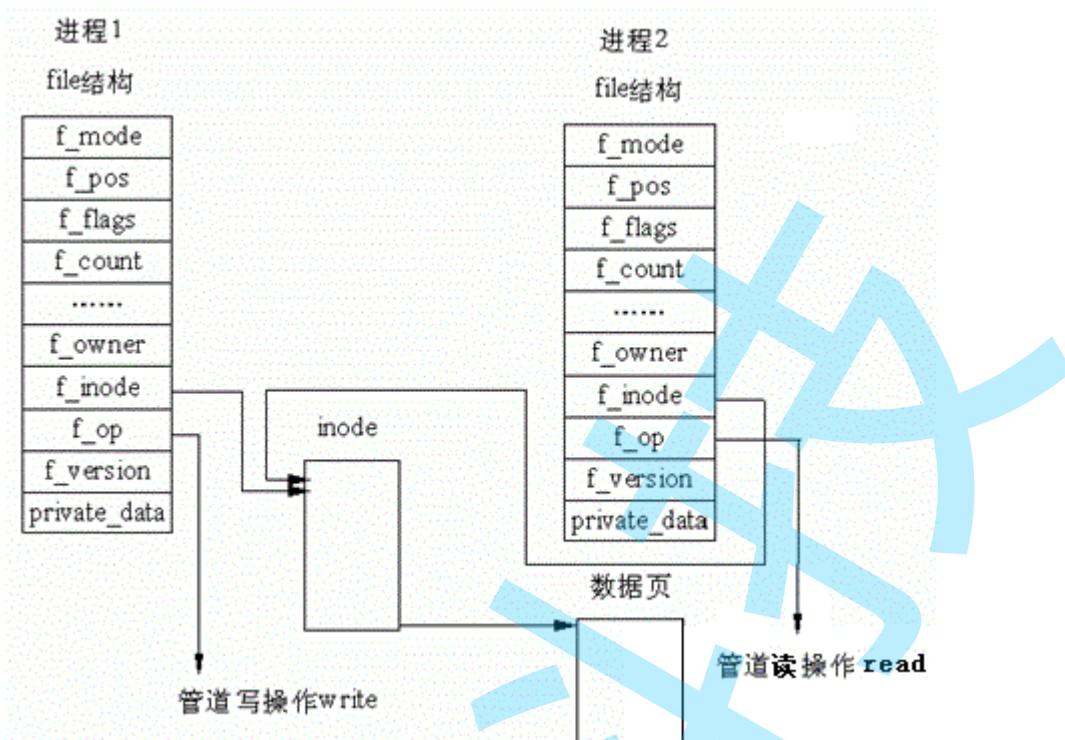
### 2. 父进程 fork 出子进程



### 3. 父进程关闭 fd[0]，子进程关闭 fd[1]



站在内核角度-管道本质



所以，看待管道，就如同看待文件一样！管道的使用和文件一致，迎合了“Linux一切皆文件思想”。

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

#define ERR_EXIT(m) do { perror(m); exit(EXIT_FAILURE); } while(0)

int main(int argc, char *argv[]) {
    int pipefd[2];
    if (pipe(pipefd) == -1) ERR_EXIT("pipe error");

    pid_t pid;
    pid = fork();
    if (pid == -1)
        ERR_EXIT("fork error");

    if (pid == 0) {
        close(pipefd[0]);
        write(pipefd[1], "hello", 5);
        close(pipefd[1]);
        exit(EXIT_SUCCESS);
    }

    close(pipefd[1]);
    char buf[10] = {0};
    read(pipefd[0], buf, 10);
    printf("buf=%s\n", buf);

    return 0;
}
```

## 例子1. 在minishell中添加管道的实现：

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

# define MAX_CMD 1024
char command[MAX_CMD];
int do_face()
{
    memset(command, 0x00, MAX_CMD);
    printf("minishell$ ");
    fflush(stdout);
    if (scanf("%[\n%c", command) == 0) {
        getchar();
        return -1;
    }
    return 0;
}
char **do_parse(char *buff)
{
    int argc = 0;
    static char *argv[32];
    char *ptr = buff;

    while(*ptr != '\0') {
        if (!isspace(*ptr)) {
            argv[argc++] = ptr;
            while((!isspace(*ptr)) && (*ptr) != '\0') {
                ptr++;
            }
        }else {
            while(isspace(*ptr)) {
                *ptr = '\0';
                ptr++;
            }
        }
    }
    argv[argc] = NULL;
    return argv;
}
int do_redirect(char *buff)
{
    char *ptr = buff, *file = NULL;
    int type = 0, fd, redirect_type = -1;
    while(*ptr != '\0') {
        if (*ptr == '>') {
            *ptr++ = '\0';

            redirect_type++;


```

```
        if (*ptr == '>') {
            *ptr++ = '\0';
            redirect_type++;
        }
        while(ispace(*ptr)) {
            ptr++;
        }
        file = ptr;
        while(!ispace(*ptr) && *ptr != '\0') {
            ptr++;
        }
        *ptr = '\0';
        if (redirect_type == 0) {
            fd = open(file, O_CREAT|O_TRUNC|O_WRONLY, 0664);
        }else {
            fd = open(file, O_CREAT|O_APPEND|O_WRONLY, 0664);
        }
        dup2(fd, 1);
    }
    ptr++;
}
return 0;
}
int do_command(char *buff)
{
    int pipe_num = 0, i;
    char *ptr = buff;
    int pipefd[32][2] = {{-1}};
    int pid = -1;

    pipe_command[pipe_num] = ptr;
    while(*ptr != '\0') {
        if (*ptr == '|') {
            pipe_num++;
            *ptr++ = '\0';
            pipe_command[pipe_num] = ptr;
            continue;
        }
        ptr++;
    }
    pipe_command[pipe_num + 1] = NULL;
    return pipe_num;
}
int do_pipe(int pipe_num)
{
    int pid = 0, i;
    int pipefd[10][2] = {{0}};
    char **argv = {NULL};

    for (i = 0; i <= pipe_num; i++) {
        pipe(pipefd[i]);
    }

    for (i = 0; i <= pipe_num; i++) {
```

```

pid = fork();
if (pid == 0) {
    do_redirect(pipe_command[i]);
    argv = do_parse(pipe_command[i]);
    if (i != 0) {
        close(pipefd[i][1]);
        dup2(pipefd[i][0], 0);
    }
    if (i != pipe_num) {
        close(pipefd[i + 1][0]);
        dup2(pipefd[i + 1][1], 1);
    }
    execvp(argv[0], argv);
} else {
    close(pipefd[i][0]);
    close(pipefd[i][1]);
    waitpid(pid, NULL, 0);
}
}

int main(int argc, char *argv[])
{
    int num = 0;
    while(1) {
        if (do_face() < 0)
            continue;
        num = do_command(command);
        do_pipe(num);
    }
    return 0;
}

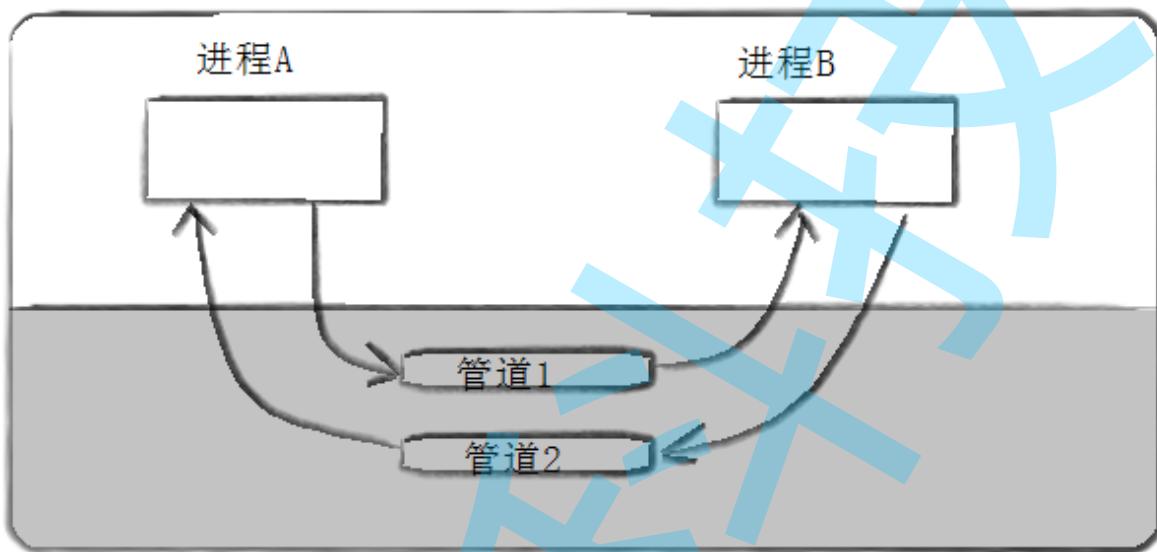
```

## 管道读写规则

- 当没有数据可读时
  - O\_NONBLOCK disable: read调用阻塞, 即进程暂停执行, 一直等到有数据来到为止。
  - O\_NONBLOCK enable: read调用返回-1, errno值为EAGAIN。
- 当管道满的时候
  - O\_NONBLOCK disable: write调用阻塞, 直到有进程读走数据
  - O\_NONBLOCK enable: 调用返回-1, errno值为EAGAIN
- 如果所有管道写端对应的文件描述符被关闭, 则read返回0
- 如果所有管道读端对应的文件描述符被关闭, 则write操作会产生信号SIGPIPE,进而可能导致write进程退出
- 当要写入的数据量不大于PIPE\_BUF时, linux将保证写入的原子性。
- 当要写入的数据量大于PIPE\_BUF时, linux将不再保证写入的原子性。

## 管道特点

- 只能用于具有共同祖先的进程（具有亲缘关系的进程）之间进行通信；通常，一个管道由一个进程创建，然后该进程调用fork，此后父、子进程之间就可应用该管道。
- 管道提供流式服务
- 一般而言，进程退出，管道释放，所以管道的生命周期随进程
- 一般而言，内核会对管道操作进行同步与互斥
- 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道



利用两个管道实现双向通信

## 命名管道

- 管道应用的一个限制就是只能在具有共同祖先（具有亲缘关系）的进程间通信。
- 如果我们想在不相关的进程之间交换数据，可以使用FIFO文件来做这项工作，它经常被称为命名管道。
- 命名管道是一种特殊类型的文件

## 创建一个命名管道

- 命名管道可以从命令行上创建，命令行方法是使用下面这个命令：

```
$ mkfifo filename
```

- 命名管道也可以从程序里创建，相关函数有：

```
int mkfifo(const char *filename, mode_t mode);
```

创建命名管道：

```
int main(int argc, char *argv[])
{
    mkfifo("p2", 0644);
    return 0;
}
```

## 匿名管道与命名管道的区别

- 匿名管道由pipe函数创建并打开。
- 命名管道由mkfifo函数创建，打开用open
- FIFO（命名管道）与pipe（匿名管道）之间唯一的区别在它们创建与打开的方式不同，一但这些工作完成之后，它们具有相同的语义。

## 命名管道的打开规则

- 如果当前打开操作是为读而打开FIFO时
  - O\_NONBLOCK disable：阻塞直到有相应进程为写而打开该FIFO
  - O\_NONBLOCK enable：立刻返回成功
- 如果当前打开操作是为写而打开FIFO时
  - O\_NONBLOCK disable：阻塞直到有相应进程为读而打开该FIFO
  - O\_NONBLOCK enable：立刻返回失败，错误码为ENXIO

## 例子1-用命名管道实现文件拷贝

读取文件，写入命名管道：

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

#define ERR_EXIT(m) \
    do \
    { \
        perror(m); \
        exit(EXIT_FAILURE); \
    } while(0)
```

```
int main(int argc, char *argv[]) { mkfifo("tp", 0644); int infd; infd = open("abc", O_RDONLY); if (infd == -1) \
ERR_EXIT("open");
```

```
    int outfd;
    outfd = open("tp", O_WRONLY);
    if (outfd == -1) ERR_EXIT("open");

    char buf[1024];
    int n;
    while ((n=read(infd, buf, 1024))>0)
    {
        write(outfd, buf, n);
    }

    close(infd);
```

```
    close(outfd);
    return 0;
}
```

读取管道, 写入目标文件:

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

#define ERR_EXIT(m) \
    do \
    { \
        perror(m); \
        exit(EXIT_FAILURE); \
    } while(0)
```

```
int main(int argc, char *argv[]) { int outfd; outfd = open("abc.bak", O_WRONLY | O_CREAT | O_TRUNC,
0644); if (outfd == -1) ERR_EXIT("open");
```

```
int infd;
infd = open("tp", O_RDONLY);
if (infd == -1)
    ERR_EXIT("open");

char buf[1024];
int n;
while ((n=read(infd, buf, 1024))>0)
{
    write(outfd, buf, n);
}
close(infd);
close(outfd);
unlink("tp");
return 0;
}
```

## 例子2-用命名管道实现server&client通信

```
# 11
total 12
-rw-r--r--. 1 root root 46 Sep 18 22:37 clientPipe.c
-rw-r--r--. 1 root root 164 Sep 18 22:37 Makefile
-rw-r--r--. 1 root root 46 Sep 18 22:38 serverPipe.c

# cat Makefile
.PHONY:all
all:clientPipe serverPipe
```

```
clientPipe:clientPipe.c
    gcc -o $@ $^
serverPipe:serverPipe.c
    gcc -o $@ $^

.PHONY:clean
clean:
    rm -f clientPipe serverPipe
```

## serverPipe.c

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

#define ERR_EXIT(m) \
do{\
    perror(m);\
    exit(EXIT_FAILURE);\
}while(0)

int main()
{
    umask(0);
    if(mkfifo("mypipe", 0644) < 0){
        ERR_EXIT("mkfifo");
    }
    int rfd = open("mypipe", O_RDONLY);
    if(rfd < 0){
        ERR_EXIT("open");
    }

    char buf[1024];
    while(1){
        buf[0] = 0;
        printf("Please wait...\n");
        ssize_t s = read(rfd, buf, sizeof(buf)-1);
        if(s > 0 ){
            buf[s-1] = 0;
            printf("client say# %s\n", buf);
        }else if(s == 0){
            printf("client quit, exit now!\n");
            exit(EXIT_SUCCESS);
        }else{
            ERR_EXIT("read");
        }
    }
}
```

```
    close(rfd);
    return 0;
}
```

## clientPipe.c

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

#define ERR_EXIT(m) \
do{\
    perror(m);\
    exit(EXIT_FAILURE);\
}while(0)

int main()
{
    int wfd = open("mypipe", O_WRONLY);
    if(wfd < 0){
        ERR_EXIT("open");
    }

    char buf[1024];
    while(1){
        buf[0] = 0;
        printf("Please Enter# ");
        fflush(stdout);
        ssize_t s = read(0, buf, sizeof(buf)-1);
        if(s > 0 ){
            buf[s] = 0;
            write(wfd, buf, strlen(buf));
        }else if(s <= 0){
            ERR_EXIT("read");
        }
    }

    close(wfd);
    return 0;
}
```

结果:

```

hb@MiWiFi-R1CL-srv:/home/hb/bit-code/listen_class/pipe
File Edit View Search Terminal Help
[root@MiWiFi-R1CL-srv pipe]# ./serverPipe
Please wait...
client say# hello world
Please wait...
client say# nihao
Please wait...
client say# good
Please wait...
client quit, exit now!

```

```

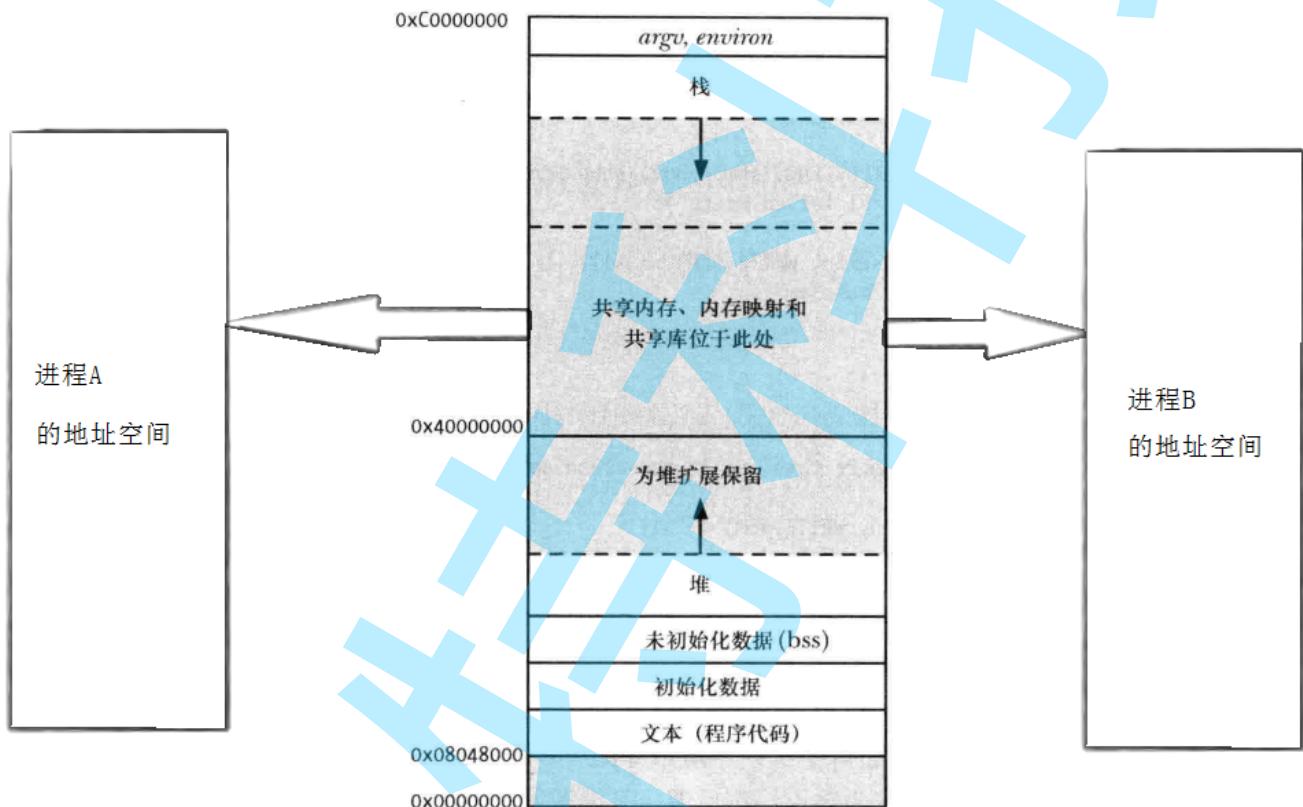
hb@MiWiFi-R1CL-srv:/home/hb/bit-code/listen_cl
File Edit View Search Terminal Help
[root@MiWiFi-R1CL-srv pipe]# ./clientPipe
Please Enter# hello world
Please Enter# nihao
Please Enter# good
Please Enter# ^C
[root@MiWiFi-R1CL-srv pipe]#

```

## system V共享内存

共享内存区是最快的IPC形式。一旦这样的内存映射到共享它的进程的地址空间，这些进程间数据传递不再涉及到内核，换句话说进程不再通过执行进入内核的系统调用来传递彼此的数据。

### 共享内存示意图



### 共享内存数据结构

```

struct shmid_ds {
    struct ipc_perm      shm_perm; /* operation perms */
    int                 shm_segsz; /* size of segment (bytes) */
    __kernel_time_t     shm_atime; /* last attach time */
    __kernel_time_t     shm_dtime; /* last detach time */
    __kernel_time_t     shm_ctime; /* last change time */
    __kernel_ipc_pid_t shm_cpid; /* pid of creator */
    __kernel_ipc_pid_t shm_lpid; /* pid of last operator */
    unsigned short      shm_nattch; /* no. of current attaches */
    unsigned short      shm_unused; /* compatibility */
    void                *shm_unused2; /* ditto - used by DIPC */
    void                *shm_unused3; /* unused */
};


```

## 共享内存函数

### shmget函数

功能：用来创建共享内存

原型

```
int shmget(key_t key, size_t size, int shmflg);
```

参数

key:这个共享内存段名字

size:共享内存大小

shmflg:由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的

返回值：成功返回一个非负整数，即该共享内存段的标识码；失败返回-1

### shmat函数

功能：将共享内存段连接到进程地址空间

原型

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

参数

shmid: 共享内存标识

shmaddr:指定连接的地址

shmflg:它的两个可能取值是SHM\_RND和SHM\_RDONLY

返回值：成功返回一个指针，指向共享内存第一个节；失败返回-1

- 说明：

shmaddr为NULL，核心自动选择一个地址

shmaddr不为NULL且shmflg无SHM\_RND标记，则以shmaddr为连接地址。

shmaddr不为NULL且shmflg设置了SHM\_RND标记，则连接的地址会自动向下调整为SHMLBA的整数倍。公式：shmaddr - (shmaddr % SHMLBA)

shmflg=SHM\_RDONLY，表示连接操作用来只读共享内存

### shmdt函数

功能：将共享内存段与当前进程脱离

原型

```
int shmdt(const void *shmaddr);
```

参数

shmaddr：由shmat所返回的指针

返回值：成功返回0；失败返回-1

注意：将共享内存段与当前进程脱离不等于删除共享内存段

## shmctl函数

功能：用于控制共享内存

原型

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

参数

shmid：由shmget返回的共享内存标识码

cmd：将要采取的动作（有三个可取值）

buf：指向一个保存着共享内存的模式状态和访问权限的数据结构

返回值：成功返回0；失败返回-1

命令	说明
IPC_STAT	把shmid_ds结构中的数据设置为共享内存的当前关联值
IPC_SET	在进程有足够的权限的前提下，把共享内存的当前关联值设置为shmid_ds数据结构中给出的值
IPC_RMID	删除共享内存段

## 实例代码

测试代码结构

```
# ls
client.c  comm.c  comm.h  Makefile  server.c
# cat Makefile

.PHONY:all
all:server client

client:client.c comm.c
        gcc -o $@ $^
server:server.c comm.c
        gcc -o $@ $^
.PHONY:clean
clean:
        rm -f client server
```

comm.h #ifndef COMM\_H #define COMM\_H

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define PATHNAME "."
#define PROJ_ID 0x6666

int createShm(int size);
int destroyShm(int shmid);
int getShm(int size);

#endif
```

comm.c #include "comm.h"

```
static int commShm(int size, int flags)
{
    key_t _key = ftok(PATHNAME, PROJ_ID);
    if(_key < 0){
        perror("ftok");
        return -1;
    }
    int shmid = 0;
    if( (shmid = shmget(_key, size, flags)) < 0){
        perror("shmget");
        return -2;
    }
    return shmid;
}

int destroyShm(int shmid)
{
    if(shmctl(shmid, IPC_RMID, NULL) < 0){
        perror("shmctl");
        return -1;
    }
    return 0;
}

int createShm(int size)
{
    return commShm(size, IPC_CREAT|IPC_EXCL|0666);
}

int getShm(int size)
{
    return commShm(size, IPC_CREAT);
}
```

server.c

```
#include "comm.h"

int main()
{
    int shmid = createShm(4096);

    char *addr = shmat(shmid, NULL, 0);
    sleep(2);
    int i = 0;
    while(i++<26){
        printf("client# %s\n", addr);
        sleep(1);
    }

    shmdt(addr);
    sleep(2);
    destroyShm(shmid);
    return 0;
}
```

client.c

```
#include "comm.h"

int main()
{
    int shmid = getShm(4096);
    sleep(1);
    char *addr = shmat(shmid, NULL, 0);
    sleep(2);
    int i = 0;
    while(i<26){
        addr[i] = 'A'+i;
        i++;
        addr[i] = 0;
        sleep(1);
    }

    shmdt(addr);
    sleep(2);
    return 0;
}
```

结果演示

hb@MiWiFi-R1CL-srv:/home/hb/bit-code/L2\_class/l1\_class/shm

```
[root@MiWiFi-R1CL-srv shm]# ./server
client#
client#
client# A
client# AB
client# ABC
client# ABCD
client# ABCDE
client# ABCDEF
client# ABCDEFG
client# ABCDEFGH
client# ABCDEFGHI
client# ABCDEFGHIJ
client# ABCDEFGHIJK
client# ABCDEFGHIJKL
client# ABCDEFGHIJKLM
client# ABCDEFGHIJKLMN
client# ABCDEFGHIJKLMNO
client# ABCDEFGHIJKLMNO
```

hb@MiWiFi-R1CL-srv:/home/hb/bit-code/L2\_class/l1\_class/shm

```
[root@MiWiFi-R1CL-srv shm]# ./client
^C
[root@MiWiFi-R1CL-srv shm]#
```

ctrl+c终止进程,再次重启

```
# ./server
shmget: File exists
# ipcs -m
----- Shared Memory Segments -----
key      shmid     owner      perms      bytes  nattch  status
0x66026a25 688145    root       666        4096      0

# ipcrm -m 688145 #删除shm ipc资源, 注意, 不是必须通过手动来删除, 这里只为演示相关指令, 删除IPC资源是进程该做的事情
```

## bytes 和 nattch 有时间可以研究一下

注意: 共享内存没有进行同步与互斥!

## system V消息队列 - 选学了解即可

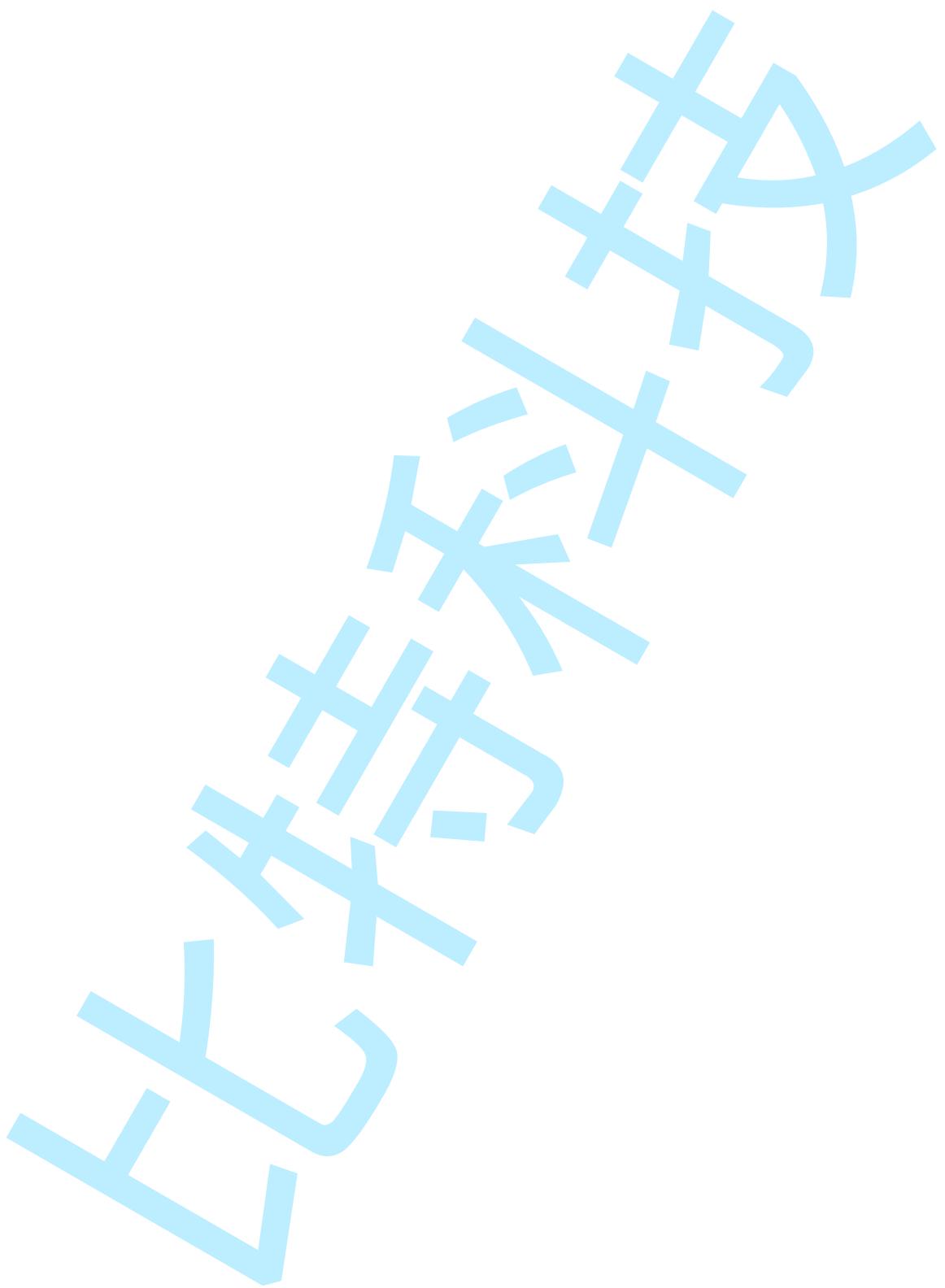
- 消息队列提供了一个从一个进程向另外一个进程发送一块数据的方法
- 每个数据块都被认为是有一个类型, 接收者进程接收的数据块可以有不同的类型值
- 特性方面
  - IPC资源必须删除, 否则不会自动清除, 除非重启, 所以system V IPC资源的生命周期随内核

## system V信号量 - 选学了解即可

信号量主要用于同步和互斥的, 下面先来看看什么是同步和互斥。

## 进程互斥

- 由于各进程要求共享资源, 而且有些资源需要互斥使用, 因此各进程间竞争使用这些资源, 进程的这种关系为进程的互斥
- 系统中某些资源一次只允许一个进程使用, 称这样的资源为临界资源或互斥资源。
- 在进程中涉及到互斥资源的程序段叫临界区
- 特性方面
  - IPC资源必须删除, 否则不会自动清除, 除非重启, 所以system V IPC资源的生命周期随内核



## 本节重点：

1. 掌握Linux信号的基本概念
2. 掌握信号产生的一般方式
3. 理解信号递达和阻塞的概念，原理。
4. 掌握信号捕捉的一般方式。
5. 重新了解可重入函数的概念。
6. 了解竞态条件的情景和处理方式
7. 了解SIGCHLD信号，重新编写信号处理函数的一般处理机制

## 信号入门

### 1. 生活角度的信号

- 你在网上买了很多件商品，再等待不同商品快递的到来。但即便快递没有到来，你也知道快递来临时，你该怎么处理快递。也就是你能“识别快递”
- 当快递员到了你楼下，你也收到快递到来的通知，但是你正在打游戏，需5min之后才能去取快递。那么在在这5min之内，你并没有下去去取快递，但是你是知道有快递到来了。也就是取快递的行为并不是一定要立即执行，可以理解成“在合适的时候去取”。
- 在收到通知，再到你拿到快递期间，是有一个时间窗口的，在这段时间，你并没有拿到快递，但是你知道有一个快递已经来了。本质上是你“记住了有一个快递要去取”
- 当你时间合适，顺利拿到快递之后，就要开始处理快递了。而处理快递一般方式有三种：1. 执行默认动作（幸福的打开快递，使用商品）2. 执行自定义动作（快递是零食，你要送给你你的女朋友）3. 忽略快递（快递拿上来之后，扔掉床头，继续开一把游戏）
- 快递到来的整个过程，对你来讲是异步的，你不能准确断定快递员什么时候给你打电话

### 2. 技术应用角度的信号

1. 用户输入命令，在Shell下启动一个前台进程。
  - . 用户按“Ctrl+C”这个键盘输入产生一个硬件中断，被OS获取，解释成信号，发送给目标前台进程
  - . 前台进程因为收到信号，进而引起进程退出

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>

int main()
{
    while(1){
        printf("I am a process, I am waiting signal!\n");
        sleep(1);
    }
}

[hb@localhost code_test]$ ./sig
I am a process, I am waiting signal!
I am a process, I am waiting signal!
I am a process, I am waiting signal!

^C
```

```
[hb@localhost code_test]$
```

- 请将生活例子和 `Ctrl-C` 信号处理过程相结合，解释一下信号处理过程
- 进程就是你，操作系统就是快递员，信号就是快递

### 3. 注意

1. `Ctrl-C` 产生的信号只能发给前台进程。一个命令后面加个`&`可以放到后台运行,这样Shell不必等待进程结束就可以接受新的命令,启动新的进程。
2. Shell可以同时运行一个前台进程和任意多个后台进程,只有前台进程才能接到像 `Ctrl-C` 这种控制键产生的信号。
3. 前台进程在运行过程中用户随时可能按下 `Ctrl-C` 而产生一个信号,也就是说该进程的用户空间代码执行到任何地方都有可能收到 `SIGINT` 信号而终止,所以信号相对于进程的控制流程来说是异步(Asynchronous)的。

### 4. 信号概念

- 信号是进程之间事件异步通知的一种方式，属于软中断。

### 5. 用 `kill -l` 命令可以察看系统定义的信号列表

```
[root@localhost ~]# kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD    18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU    23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF    28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN   35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- 每个信号都有一个编号和一个宏定义名称,这些宏定义可以在 `signal.h` 中找到,例如其中有定义 `#define SIGINT 2`
- 编号34以上的是实时信号,本章只讨论编号34以下的信号,不讨论实时信号。这些信号各自在什么条件下产生,默认的处理动作是什么,在 `signal(7)` 中都有详细说明: `man 7 signal`

SIGNAL(7) Linux Programmer's Manual SIGNAL(7)

**NAME**

signal - overview of signals

**DESCRIPTION**

Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

**Signal Dispositions**

Each signal has a current **disposition**, which determines how the process behaves when it is delivered the signal.

The entries in the "Action" column of the tables below specify the default disposition for each signal, as follows:

Term	Default action is to terminate the process.
Ign	Default action is to ignore the signal.
Core	Default action is to terminate the process and dump core (see core(5)).
Stop	Default action is to stop the process.
:	

## 6. 信号处理常见方式概览

(sigaction函数稍后详细介绍),可选的处理动作有以下三种:

1. 忽略此信号。
2. 执行该信号的默认处理动作。
3. 提供一个信号处理函数,要求内核在处理该信号时切换到用户态执行这个处理函数,这种方式称为捕捉(Catch)一个信号。

## 产生信号

### 1. 通过终端按键产生信号

SIGINT的默认处理动作是终止进程,SIGQUIT的默认处理动作是终止进程并且Core Dump,现在我们来验证一下。

#### Core Dump

首先解释什么是Core Dump。当一个进程要异常终止时,可以选择把进程的用户空间内存数据全部保存到磁盘上,文件名通常是core,这叫做Core Dump。进程异常终止通常是因为有Bug,比如非法内存访问导致段错误,事后可以用调试器检查core文件以查清错误原因,这叫做Post-mortem Debug (事后调试)。一个进程允许产生多大的core文件取决于进程的Resource Limit(这个信息保存在PCB中)。默认是不允许产生core文件的,因为core文件中可能包含用户密码等敏感信息,不安全。在开发调试阶段可以用ulimit命令改变这个限制,允许产生core文件。首先用ulimit命令改变Shell进程的Resource Limit,允许core文件最大为1024K: \$ ulimit -c 1024

```
[root@localhost test11]# ulimit -c 1024
[root@localhost test11]# ulimit -a
core file size          (blocks, -c) 1024
data seg size            (kbytes, -d) unlimited
scheduling priority      (-e) 0
file size                (blocks, -f) unlimited
pending signals           (-i) 7908
max locked memory        (kbytes, -l) 64
max memory size          (kbytes, -m) unlimited
open files               (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority        (-r) 0
stack size               (kbytes, -s) 10240
cpu time                 (seconds, -t) unlimited
```

然后写一个死循环程序:

```
[root@localhost test11]# cat test.c
#include <stdio.h>
int main()
{
    printf("pid is : %d\n",getpid());
    while(1);
    return 0;
}
```

前台运行这个程序,然后在终端键入Ctrl-C (貌似不行) 或Ctrl-\ (介个可以) :

```
[root@localhost test11]# ./test
pid is : 4506
^CQuit (core dumped)
[root@localhost test11]# ll
total 92
-rw-----. 1 root root 159744 Apr 21 18:04 core.4506
-rw-r--r--. 1 root root   1 Apr 21 18:00 makefile
-rwxr-xr-x. 1 root root  4766 Apr 21 18:03 test
-rw-r--r--. 1 hb   hb    92 Apr 21 18:03 test.c
-rw-r--r--. 1 root root  627 Apr 21 17:36 test.cpp
[root@localhost test11]#
```

ulimit命令改变了Shell进程的Resource Limit,test进程的PCB由Shell进程复制而来,所以也具有和Shell进程相同的Resource Limit值,这样就可以产生Core Dump了。 使用core文件:

```
total 16
-rw-r--r--. 1 root root  62 Feb 28 17:32 Makefile
-rwxr-xr-x. 1 root root 5870 Feb 28 18:03 test
-rw-r--r--. 1 root root 139 Feb 28 17:30 test.c
[root@bit_tech test78]# ./test
Segmentation fault (core dumped)
[root@bit_tech test78]# ls
core.2884 Makefile test test.c
[root@bit_tech test78]# gdb test
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-75.el6)
Copyright (C) 2010 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 "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /BIT/project/test78/test...done.
(gdb) core-file core.2884
[New Thread 2884]
Missing separate debuginfo for
Try: yum --enablerepo='*-debug*' install /usr/lib/debug/.build-id/d5/942f21cf3002919e55180a33793d3c25a060f3
Reading symbols from /lib/libc-2.12.so...Reading symbols from /usr/lib/debug/lib/libc-2.12.so.debug...done.
done.
Loaded symbols for /lib/libc-2.12.so
Reading symbols from /lib/ld-2.12.so...Reading symbols from /usr/lib/debug/lib/ld-2.12.so.debug...done.
done.
Loaded symbols for /lib/ld-2.12.so
Core was generated by `./test'.
Program terminated with signal 11, Segmentation fault.
#0 0x080483ad in fun () at test.c:8
8          *p = 100;
(gdb) [REDACTED]
```

## 2. 调用系统函数向进程发信号

首先在后台执行死循环程序,然后用kill命令给它发SIGSEGV信号。

```
[root@localhost test11]# ./test &
[3] 4568
[root@localhost test11]# kill -SIGSEGV 4568 → 多按一次回车
[root@localhost test11]# [REDACTED]
[3] Segmentation fault (core dumped) ./test
[root@localhost test11]# cat test.c
#include <stdio.h>
int main()
{
    while(1);
    return 0;
}
```

- 4568是test进程的id。之所以要再次回车才显示Segmentation fault,是因为在4568进程终止掉之前已经回到了Shell提示符等待用户输入下一条命令,Shell不希望Segmentation fault信息和用户的输入交错在一起,所以等用户输入命令之后才显示。
- 指定发送某种信号的kill命令可以有多种写法,上面的命令还可以写成kill -SIGSEGV 4568或kill -11 4568,11是信号SIGSEGV的编号。以往遇到的段错误都是由非法内存访问产生的,而这个程序本身没错,给它发SIGSEGV也能产生段错误。

kill命令是调用kill函数实现的。kill函数可以给一个指定的进程发送指定的信号。raise函数可以给当前进程发送指定的信号(自己给自己发信号)。

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
这两个函数都是成功返回0,错误返回-1。
```

abort函数使当前进程接收到信号而异常终止。

```
#include <stdlib.h>
void abort(void);
就像exit函数一样,abort函数总是会成功的,所以没有返回值。
```

### 3. 由软件条件产生信号

SIGPIPE是一种由软件条件产生的信号,在“管道”中已经介绍过了。本节主要介绍alarm函数 和SIGALRM信号。

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
调用alarm函数可以设定一个闹钟,也就是告诉内核在seconds秒之后给当前进程发SIGALRM信号,该信号的默认处理动作是终止当前进程。
```

这个函数的返回值是0或者是以前设定的闹钟时间还余下的秒数。打个比方,某人要小睡一觉,设定闹钟为30分钟之后响,20分钟后被人吵醒了,还想多睡一会儿,于是重新设定闹钟为15分钟之后响,“以前设定的闹钟时间还余下的时间”就是10分钟。如果seconds值为0,表示取消以前设定的闹钟,函数的返回值仍然是以前设定的闹钟时间还余下的秒数(自己验证一下?)

例 alarm

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     int count = 14;
7     alarm(1);
8     for(;1;count++){
9         printf("count = %d\n",count);
10    }
11
12    return 0;
13 }
```

这个程序的作用是1秒钟之内不停地数数,1秒钟到了就被SIGALRM信号终止。

### 4. 硬件异常产生信号

硬件异常被硬件以某种方式被硬件检测到并通知内核,然后内核向当前进程发送适当的信号。例如当前进程执行了除以0的指令,CPU的运算单元会产生异常,内核将这个异常解释为SIGFPE信号发送给进程。再比如当前进程访问了非法内存地址,,MMU会产生异常,内核将这个异常解释为SIGSEGV信号发送给进程。

## 信号捕捉初识

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

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
}

int main()
{
    signal(2, handler); //前文提到过，信号是可以被自定义捕捉的，sigal函数就是来进行信号捕捉的，提前了解一下
    while(1);
    return 0;
}

[hb@localhost code_test]$ ./sig
^Ccatch a sig : 2
^Quit (core dumped)
[hb@localhost code_test]$
```

## 模拟一下野指针异常

```
//默认行为
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
}

int main()
{
    //signal(SIGSEGV, handler);
    sleep(1);
    int *p = NULL;
    *p = 100;

    while(1);
    return 0;
}

[hb@localhost code_test]$ ./sig
Segmentation fault (core dumped)
[hb@localhost code_test]$
```

//捕捉行为

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
}

int main()
{
    //signal(SIGSEGV, handler);
    sleep(1);
    int *p = NULL;
    *p = 100;

    while(1);
    return 0;
}
[hb@localhost code_test]$ ./sig
[hb@localhost code_test]$ ./sig
catch a sig : 11
catch a sig : 11
catch a sig : 11
```

由此可以确认，我们在C/C++当中除零，内存越界等异常，在系统层面上，是被当成信号处理的。

## 总结思考一下

- 上面所说的所有信号产生，最终都要有OS来进行执行，为什么？OS是进程的管理者
- 信号的处理是否是立即处理的？在合适的时候
- 信号如果不是被立即处理，那么信号是否需要暂时被进程记录下来？记录在哪里最合适呢？
- 一个进程在没有收到信号的时候，能否知道，自己应该对合法信号作何处理呢？
- 如何理解OS向进程发送信号？能否描述一下完整的发送处理过程？

## 阻塞信号

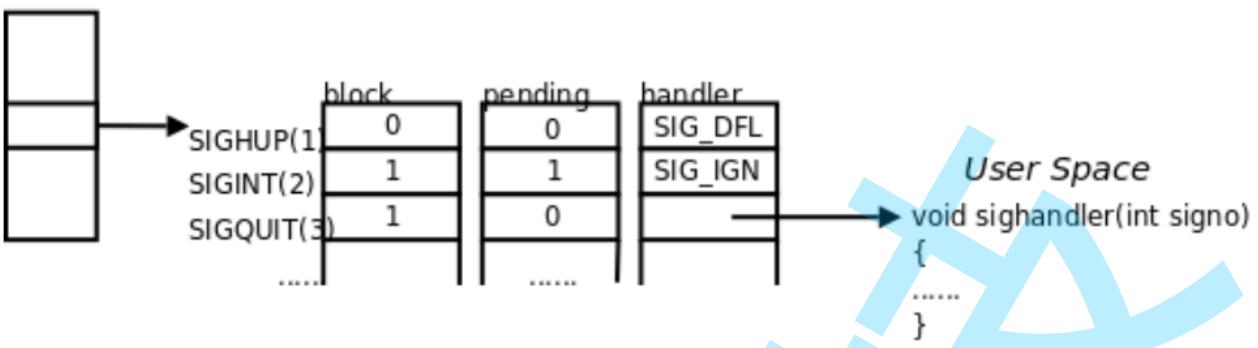
### 1. 信号其他相关常见概念

- 实际执行信号的处理动作称为信号递达(Delivery)
- 信号从产生到递达之间的状态，称为信号未决(Pending)。
- 进程可以选择阻塞(Block)某个信号。
- 被阻塞的信号产生时将保持在未决状态，直到进程解除对此信号的阻塞，才执行递达的动作。
- 注意，阻塞和忽略是不同的，只要信号被阻塞就不会递达，而忽略是在递达之后可选的一种处理动作。

### 2. 在内核中的表示

信号在内核中的表示示意图

## task\_struct



- 每个信号都有两个标志位分别表示阻塞(block)和未决(pending),还有一个函数指针表示处理动作。信号产生时,内核在进程控制块中设置该信号的未决标志,直到信号递达才清除该标志。在上图的例子中,SIGHUP信号未阻塞也未产生过,当它递达时执行默认处理动作。
- SIGINT信号产生过,但正在被阻塞,所以暂时不能递达。虽然它的处理动作是忽略,但在没有解除阻塞之前不能忽略这个信号,因为进程仍有机会改变处理动作之后再解除阻塞。
- SIGQUIT信号未产生过,一旦产生SIGQUIT信号将被阻塞,它的处理动作是用户自定义函数sighandler。如果在进程解除对某信号的阻塞之前这种信号产生过多次,将如何处理?POSIX.1允许系统递送该信号一次或多次。Linux是这样实现的:常规信号在递达之前产生多次只计一次,而实时信号在递达之前产生多次可以依次放在一个队列里。本章不讨论实时信号。

## 3. sigset\_t

从上图来看,每个信号只有一个bit的未决标志,非0即1,不记录该信号产生了多少次,阻塞标志也是这样表示的。因此,未决和阻塞标志可以用相同的数据类型sigset\_t来存储,sigset\_t称为信号集,这个类型可以表示每个信号的“有效”或“无效”状态,在阻塞信号集中“有效”和“无效”的含义是该信号是否被阻塞,而在未决信号集中“有效”和“无效”的含义是该信号是否处于未决状态。下一节将详细介绍信号集的各种操作。阻塞信号集也叫做当前进程的信号屏蔽字(Signal Mask),这里的“屏蔽”应该理解为阻塞而不是忽略。

## 4. 信号集操作函数

sigset\_t类型对于每种信号用一个bit表示“有效”或“无效”状态,至于这个类型内部如何存储这些bit则依赖于系统实现,从使用者的角度是不必关心的,使用者只能调用以下函数来操作sigset\_t变量,而不应该对它的内部数据做任何解释,比如用printf直接打印sigset\_t变量是没有意义的

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember (const sigset_t *set, int signo);
```

- 函数sigemptyset初始化set所指向的信号集,使其中所有信号的对应bit清零,表示该信号集不包含任何有效信号。
- 函数sigfillset初始化set所指向的信号集,使其中所有信号的对应bit置位,表示该信号集的有效信号包括系统支持的所有信号。
- 注意,在使用sigset\_t类型的变量之前,一定要调用sigemptyset或sigfillset做初始化,使信号集处于确定的状态。初始化sigset\_t变量之后就可以在调用sigaddset和sigdelset在该信号集中添加或删除某种有效信

号。

这四个函数都是成功返回0,出错返回-1。 sigismember是一个布尔函数,用于判断一个信号集的有效信号中是否包含某种信号,若包含则返回1,不包含则返回0,出错返回-1。

## sigprocmask

调用函数sigprocmask可以读取或更改进程的信号屏蔽字(阻塞信号集)。

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

返回值:若成功则为0,若出错则为-1

如果oset是非空指针,则读取进程的当前信号屏蔽字通过oset参数传出。如果set是非空指针,则更改进程的信号屏蔽字,参数how指示如何更改。如果oset和set都是非空指针,则先将原来的信号屏蔽字备份到oset里,然后根据set和how参数更改信号屏蔽字。假设当前的信号屏蔽字为mask,下表说明了how参数的可选值。

SIG_BLOCK	set包含了我们希望添加到当前信号屏蔽字的信号,相当于 $mask=mask set$
SIG_UNBLOCK	set包含了我们希望从当前信号屏蔽字中解除阻塞的信号,相当于 $mask=mask&\sim set$
SIG_SETMASK	设置当前信号屏蔽字为set所指向的值,相当于 $mask=set$

如果调用sigprocmask解除了对当前若干个未决信号的阻塞,则在sigprocmask返回前,至少将其中一个信号递达。

## sigpending

```
#include <signal.h>
sigpending
```

读取当前进程的未决信号集,通过set参数传出。调用成功则返回0,出错则返回-1。下面用刚学的几个函数做个实验。程序如下:

```

1 #include <stdio.h>
2 #include <signal.h>
3 #include <unistd.h>
4
5 void printsigset(sigset_t *set)
6 {
7     int i=0;
8     for(;i<32;i++){
9         if ( sigismember(set, i)){
10             putchar('1');
11         }else{
12             putchar('0');
13         }
14     }
15     puts("");
16 }
17 int main()
18 {
19     sigset_t s, p;
20     sigemptyset(&s);
21     sigaddset(&s, SIGINT);
22     sigprocmask(SIG_BLOCK, &s, NULL);
23     while(1){
24         if(siginpending(&p)){
25             printsigset(&p);
26             sleep(1);
27         }
28     }
29     return 0;
30 }

```

判断指定信号是否在目标集合中

定义信号集对象，并清空初始化

ctrl + c

设置阻塞信号集，阻塞SIGINT信号

获取未决信号集

```

[root@localhost sig]# ./test
10000000000000000000000000000000
10000000000000000000000000000000
10000000000000000000000000000000
10000000000000000000000000000000
^C1010000000000000000000000000000
10100000000000000000000000000000
10100000000000000000000000000000
^\\Quit (core dumped)

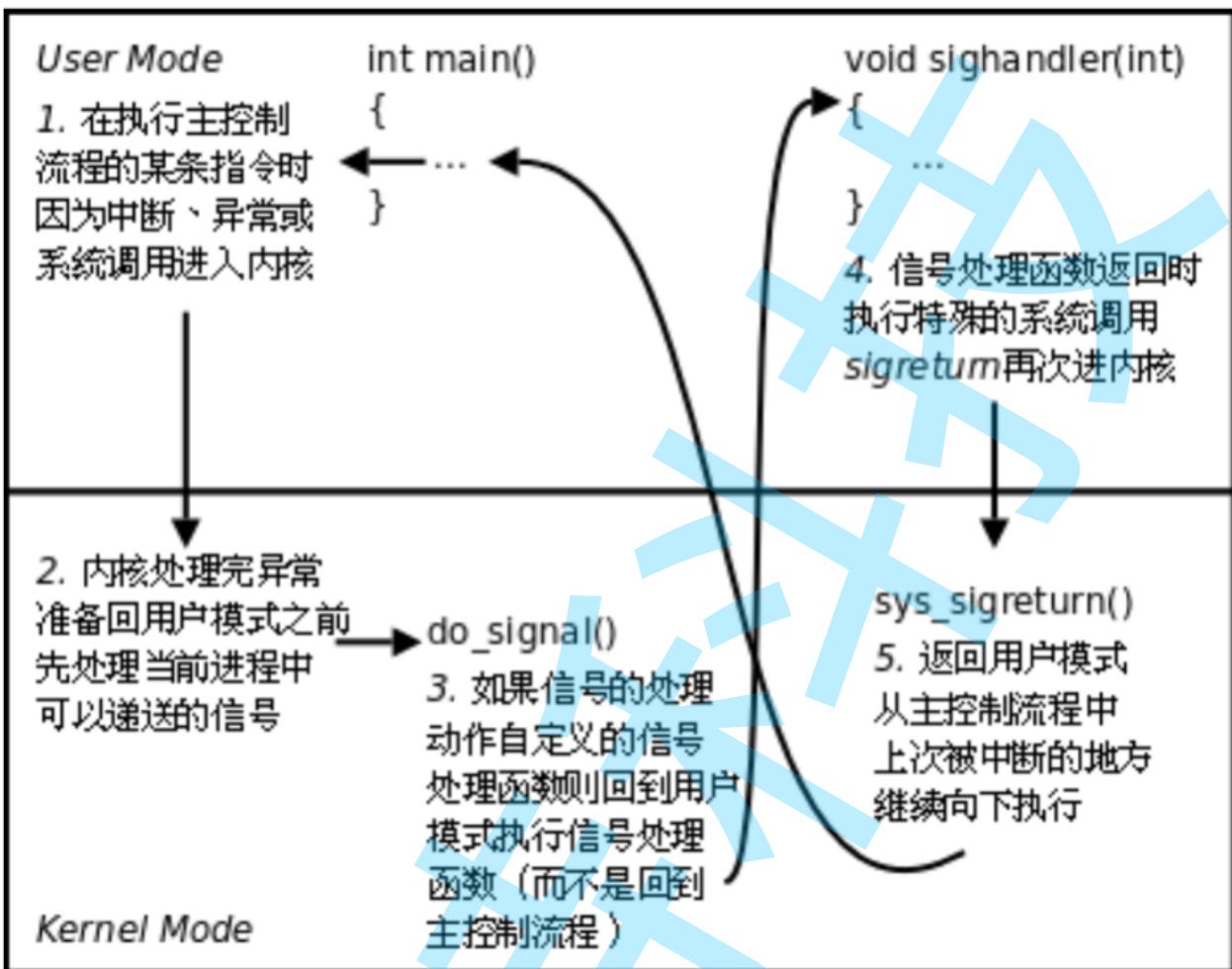
```

键入 ctrl + c (SIGINT) ,  
该信号被test阻塞，所以一直  
处于未决状态，不被处理

程序运行时,每秒钟把各信号的未决状态打印一遍,由于我们阻塞了SIGINT信号,按Ctrl-C将会使SIGINT信号处于未决状态,按Ctrl-\仍然可以终止程序,因为SIGQUIT信号没有阻塞。

## 捕捉信号

图 33.2. 信号的捕捉



## 1. 内核如何实现信号的捕捉

如果信号的处理动作是用户自定义函数，在信号递达时就调用这个函数，这称为捕捉信号。由于信号处理函数的代码是在用户空间的，处理过程比较复杂，举例如下：用户程序注册了SIGQUIT信号的处理函数sighandler。当前正在执行main函数，这时发生中断或异常切换到内核态。在中断处理完毕后要返回用户态的main函数之前检查到有信号SIGQUIT递达。内核决定返回用户态后不是恢复main函数的上下文继续执行，而是执行sighandler函数，sighandler和main函数使用不同的堆栈空间，它们之间不存在调用和被调用的关系，是两个独立的控制流程。sighandler函数返回后自动执行特殊的系统调用sigreturn再次进入内核态。如果没有新的信号要递达，这次再返回用户态就是恢复main函数的上下文继续执行了。

## 2. sigaction

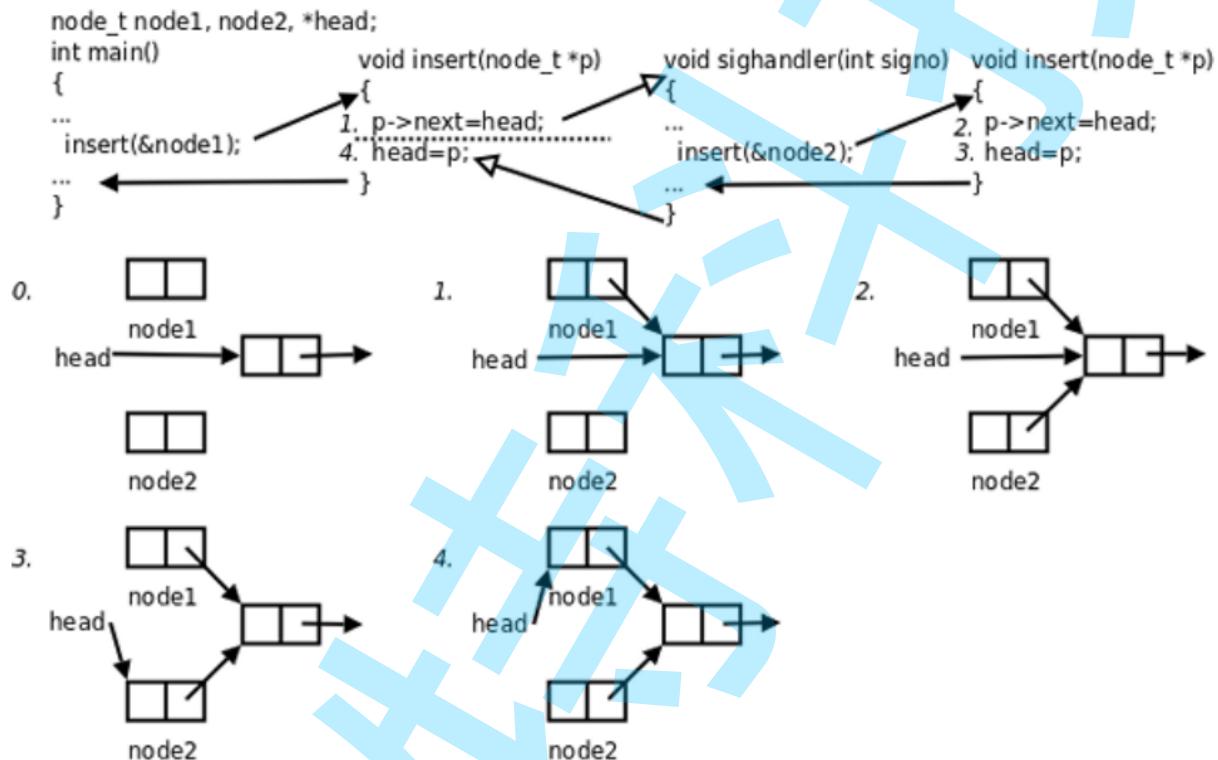
```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

- `sigaction`函数可以读取和修改与指定信号相关联的处理动作。调用成功则返回0，出错则返回-1。`signo`是指定信号的编号。若`act`指针非空，则根据`act`修改该信号的处理动作。若`oact`指针非空，则通过`oact`传出该信号原来的处理动作。`act`和`oact`指向`sigaction`结构体：

- 将sa\_handler赋值为常数SIG\_IGN传给sigaction表示忽略信号,赋值为常数SIG\_DFL表示执行系统默认动作,赋值为一个函数指针表示用自定义函数捕捉信号,或者说向内核注册了一个信号处理函数,该函数返回值为void,可以带一个int参数,通过参数可以得知当前信号的编号,这样就可以用同一个函数处理多种信号。显然,这也是一个回调函数,不是被main函数调用,而是被系统所调用。

当某个信号的处理函数被调用时,内核自动将当前信号加入进程的信号屏蔽字,当信号处理函数返回时自动恢复原来的信号屏蔽字,这样就保证了在处理某个信号时,如果这种信号再次产生,那么它会被阻塞到当前处理结束为止。如果在调用信号处理函数时,除了当前信号被自动屏蔽之外,还希望自动屏蔽另外一些信号,则用sa\_mask字段说明这些需要额外屏蔽的信号,当信号处理函数返回时自动恢复原来的信号屏蔽字。sa\_flags字段包含一些选项,本章的代码都把sa\_flags设为0,sa\_sigaction是实时信号的处理函数,本章不详细解释这两个字段,有兴趣的同学可以在了解一下。

## 可重入函数



- main函数调用insert函数向一个链表head中插入节点node1,插入操作分为两步,刚做完第一步的时候,因为硬件中断使进程切换到内核,再次回用户态之前检查到有信号待处理,于是切换到sighandler函数,sighandler也调用insert函数向同一个链表head中插入节点node2,插入操作的两步都做完之后从sighandler返回内核态,再次回到用户态就从main函数调用的insert函数中继续往下执行,先前做第一步之后被打断,现在继续做完第二步。结果是,main函数和sighandler先后向链表中插入两个节点,而最后只有一个节点真正插入链表中了。
- 像上例这样,insert函数被不同的控制流程调用,有可能在第一次调用还没返回时就再次进入该函数,这称为重入,insert函数访问一个全局链表,有可能因为重入而造成错乱,像这样的函数称为不可重入函数,反之,如果一个函数只访问自己的局部变量或参数,则称为可重入(Reentrant)函数。想一下,为什么两个不同的控制流程调用同一个函数,访问它的同一个局部变量或参数就不会造成错乱?

如果一个函数符合以下条件之一则是不可重入的:

- 调用了malloc或free,因为malloc也是用全局链表来管理堆的。
- 调用了标准I/O库函数。标准I/O库的很多实现都以不可重入的方式使用全局数据结构。

# volatile

- 该关键字在C当中我们已经有所涉猎，今天我们站在信号的角度重新理解一下

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

int flag = 0;

void handler(int sig)
{
    printf("change flag 0 to 1\n");
    flag = 1;
}

int main()
{
    signal(2, handler);
    while(!flag);
    printf("process quit normal\n");
    return 0;
}
```

```
[hb@localhost code_test]$ cat Makefile
sig: sig.c
    gcc -o sig sig.c #-02
.PHONY:clean
clean:
    rm -f sig
```

```
[hb@localhost code_test]$ ./sig
^Cchange flag 0 to 1
process quit normal
```

标准情况下，键入 `CTRL-C`, 2号信号被捕捉，执行自定义动作，修改 `flag = 1`，`while` 条件不满足，退出循环，进程退出

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

int flag = 0;

void handler(int sig)
{
    printf("change flag 0 to 1\n");
    flag = 1;
}

int main()
```

```
{  
    signal(2, handler);  
    while(!flag);  
    printf("process quit normal\n");  
    return 0;  
}  
  
[hb@localhost code_test]$ cat Makefile  
sig: sig.c  
    gcc -o sig sig.c -O2  
.PHONY: clean  
clean:  
    rm -f sig  
  
[hb@localhost code_test]$ ./sig  
^Cchange flag 0 to 1  
^Cchange flag 0 to 1  
^Cchange flag 0 to 1
```

优化情况下，键入 `CTRL-C`, 2号信号被捕捉，执行自定义动作，修改 `flag = 1`，但是 `while` 条件依旧满足，进程继续运行！但是很明显 `flag` 肯定已经被修改了，但是为何循环依旧执行？很明显，`while` 循环检查的 `flag`，并不是内存中最新的 `flag`，这就存在了数据二异性的的问题。`while` 检测的 `flag` 其实已经因为优化，被放在了 CPU 寄存器当中。如何解决呢？很明显需要 `volatile`

```
[hb@localhost code_test]$ cat sig.c  
#include <stdio.h>  
#include <signal.h>  
  
volatile int flag = 0;  
  
void handler(int sig)  
{  
    printf("change flag 0 to 1\n");  
    flag = 1;  
}  
  
int main()  
{  
    signal(2, handler);  
    while(!flag);  
    printf("process quit normal\n");  
    return 0;  
}  
  
[hb@localhost code_test]$ cat Makefile  
sig: sig.c  
    gcc -o sig sig.c -O2  
.PHONY: clean  
clean:  
    rm -f sig  
  
[hb@localhost code_test]$ ./sig
```

```
^Cchage flag 0 to 1
process quit normal
```

- `volatile` 作用：保持内存的可见性，告知编译器，被该关键字修饰的变量，不允许被优化，对该变量的任何操作，都必须在真实的内存中进行操作

## SIGCHLD信号 - 选学了解

进程一章讲过用`wait`和`waitpid`函数清理僵尸进程，父进程可以阻塞等待子进程结束，也可以非阻塞地查询是否有子进程结束等待清理（也就是轮询的方式）。采用第一种方式，父进程阻塞了就不能处理自己的工作了；采用第二种方式，父进程在处理自己的工作的同时还要记得时不时地轮询一下，程序实现复杂。

其实，子进程在终止时会给父进程发SIGCHLD信号，该信号的默认处理动作是忽略，父进程可以自定义SIGCHLD信号的处理函数，这样父进程只需专心处理自己的工作，不必关心子进程了，子进程终止时会通知父进程，父进程在信号处理函数中调用`wait`清理子进程即可。

请编写一个程序完成以下功能：父进程fork出子进程，子进程调用`exit(2)`终止，父进程自定义SIGCHLD信号的处理函数，在其中调用`wait`获得子进程的退出状态并打印。

事实上，由于UNIX的历史原因，要想不产生僵尸进程还有另外一种办法：父进程调用`sigaction`将SIGCHLD的处理动作置为SIG\_IGN，这样fork出来的子进程在终止时会自动清理掉，不会产生僵尸进程，也不会通知父进程。系统默认的忽略动作和用户用`sigaction`函数自定义的忽略通常是没有区别的，但这是一个特例。此方法对于Linux可用，但不能保证在其它UNIX系统上都可用。请编写程序验证这样做不会产生僵尸进程。

测试代码

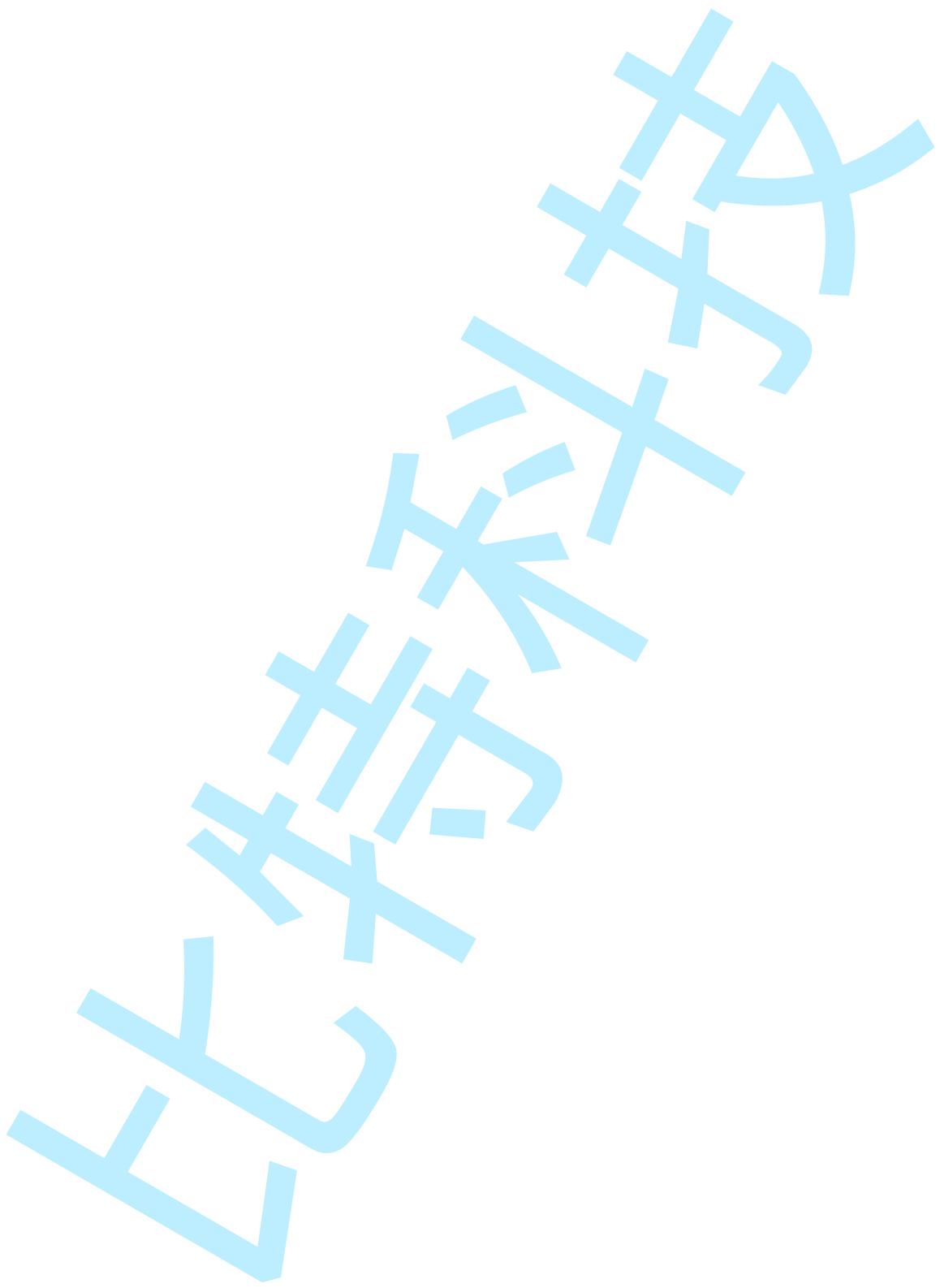
```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void handler(int sig)
{
    pid_t id;
    while( (id = waitpid(-1, NULL, WNOHANG)) > 0){
        printf("wait child success: %d\n", id);
    }
    printf("child is quit! %d\n", getpid());
}

int main()
{
    signal(SIGCHLD, handler);
    pid_t cid;
    if((cid = fork()) == 0){ //child
        printf("child : %d\n", getpid());
        sleep(3);
        exit(1);
    }

    while(1){
        printf("father proc is doing some thing!\n");
        sleep(1);
    }
}
```

```
    }  
    return 0;  
}
```



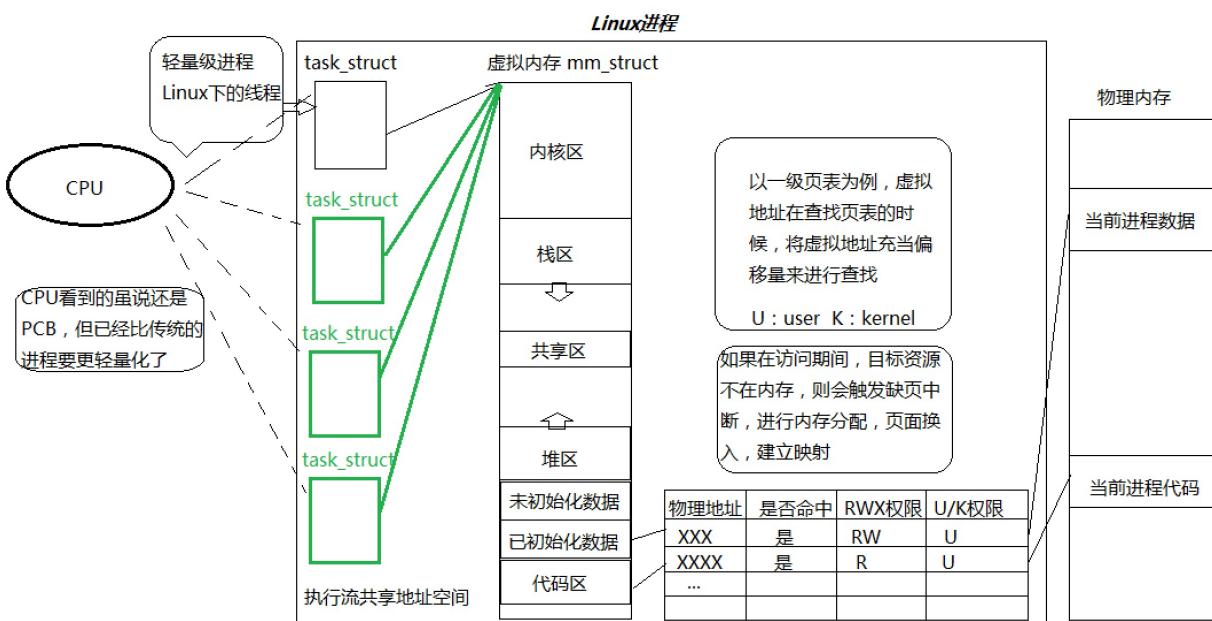
### ###本节重点：

1. 了解线程概念，理解线程与进程区别与联系。
2. 学会线程控制，线程创建，线程终止，线程等待。
3. 了解线程分离与线程安全概念。
4. 学会线程同步。
5. 学会使用互斥量，条件变量，posix信号量，以及读写锁。
6. 理解基于读写锁的读者写者问题。

### ##1. Linux线程概念

#### ###什么是线程

- 在一个程序里的一个执行路线就叫做线程（thread）。更准确的定义是：线程是“一个进程内部的控制序列”
- 一切进程至少都有一个执行线程
- 线程在进程内部运行，本质是在进程地址空间内运行
- 在Linux系统中，在CPU眼中，看到的PCB都要比传统的进程更加轻量化
- 透过进程虚拟地址空间，可以看到进程的大部分资源，将进程资源合理分配给每个执行流，就形成了线程执行流



## 线程的优点

- 创建一个新线程的代价要比创建一个新进程小得多
- 与进程之间的切换相比，线程之间的切换需要操作系统做的工作要少很多
- 线程占用的资源要比进程少很多
- 能充分利用多处理器的可并行数量
- 在等待慢速I/O操作结束的同时，程序可执行其他的计算任务
- 计算密集型应用，为了能在多处理器系统上运行，将计算分解到多个线程中实现
- I/O密集型应用，为了提高性能，将I/O操作重叠。线程可以同时等待不同的I/O操作。

## 线程的缺点

- 性能损失
  - 一个很少被外部事件阻塞的计算密集型线程往往无法与其它线程共享同一个处理器。如果计算密集型线程的数量比可用的处理器多，那么可能会有较大的性能损失，这里的性能损失指的是增加了额外的同步和调度开销，而可用的资源不变。
- 健壮性降低
  - 编写多线程需要更全面更深入的考虑，在一个多线程程序里，因时间分配上的细微偏差或者因共享了不该共享的变量而造成不良影响的可能性是很大的，换句话说线程之间是缺乏保护的。
- 缺乏访问控制
  - 进程是访问控制的基本粒度，在一个线程中调用某些OS函数会对整个进程造成影响。
- 编程难度提高
  - 编写与调试一个多线程程序比单线程程序困难得多

## 线程异常

- 单个线程如果出现除零，野指针问题导致线程崩溃，进程也会随着崩溃
- 线程是进程的执行分支，线程出异常，就类似进程出异常，进而触发信号机制，终止进程，进程终止，该进程内的所有线程也就随即退出

## 线程用途

- 合理的使用多线程，能提高CPU密集型程序的执行效率
- 合理的使用多线程，能提高IO密集型程序的用户体验（如生活中我们一边写代码一边下载开发工具，就是多线程运行的一种表现）

### ##2. Linux进程VS线程

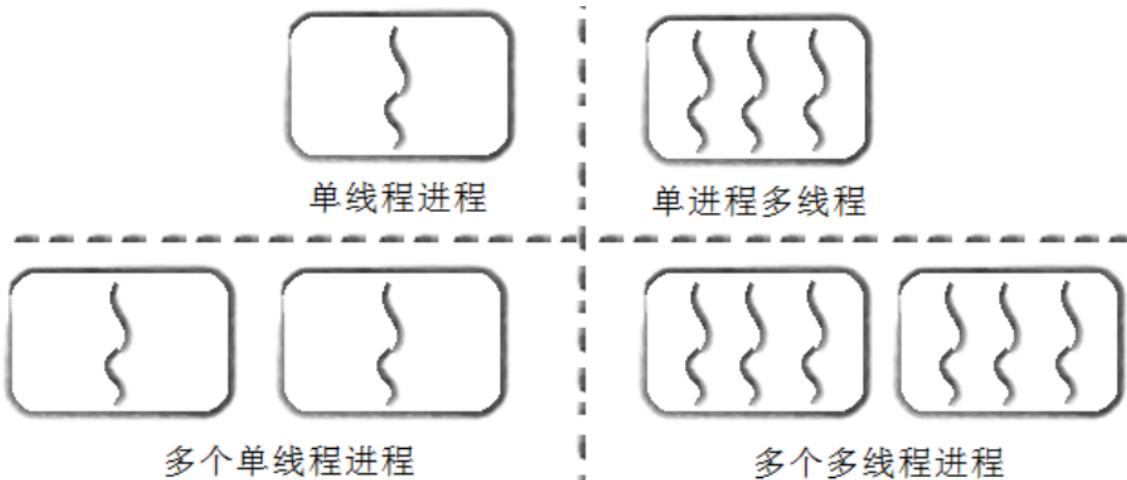
#### ###进程和线程

- 进程是资源分配的基本单位
- 线程是调度的基本单位
- 线程共享进程数据，但也拥有自己的一部分数据：
  - 线程ID
  - 一组寄存器
  - 栈
  - errno
  - 信号屏蔽字
  - 调度优先级

###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

进程和线程的关系如下图：



## 关于进程线程的问题

- 如何看待之前学习的单进程？具有一个线程执行流的进程

##3. Linux线程控制

###POSIX线程库

- 与线程有关的函数构成了一个完整的系列，绝大多数函数的名字都是以“pthread\_”打头的
- 要使用这些函数库，要通过引入头文<pthread.h>
- 链接这些线程函数库时要使用编译器命令的“-lpthread”选项

###创建线程

功能：创建一个新的线程

原型

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

参数

thread: 返回线程ID  
attr: 设置线程的属性, attr为NULL表示使用默认属性  
start\_routine: 是个函数地址, 线程启动后要执行的函数  
arg: 传给线程启动函数的参数

返回值：成功返回0；失败返回错误码

错误检查：

- 传统的一些函数是，成功返回0，失败返回-1，并且对全局变量errno赋值以指示错误。
- pthread函数出错时不会设置全局变量errno（而大部分其他POSIX函数会这样做）。而是将错误代码通过返回值返回
- pthread同样也提供了线程内的errno变量，以支持其它使用errno的代码。对于pthread函数的错误，建议通过返回值来判定，因为读取返回值要比读取线程内的errno变量的开销更小

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

```

#include <string.h>
#include <pthread.h>

void *rout(void *arg) {
    int i;
    for( ; ; ) {
        printf("I'am thread 1\n");
        sleep(1);
    }
}

int main( void )
{
    pthread_t tid;
    int ret;
    if ( (ret(pthread_create(&tid, NULL, rout, NULL)) != 0) {
        fprintf(stderr, "pthread_create : %s\n", strerror(ret));
        exit(EXIT_FAILURE);
    }

    int i;
    for( ; ; ) {
        printf("I'am main thread\n");
        sleep(1);
    }
}

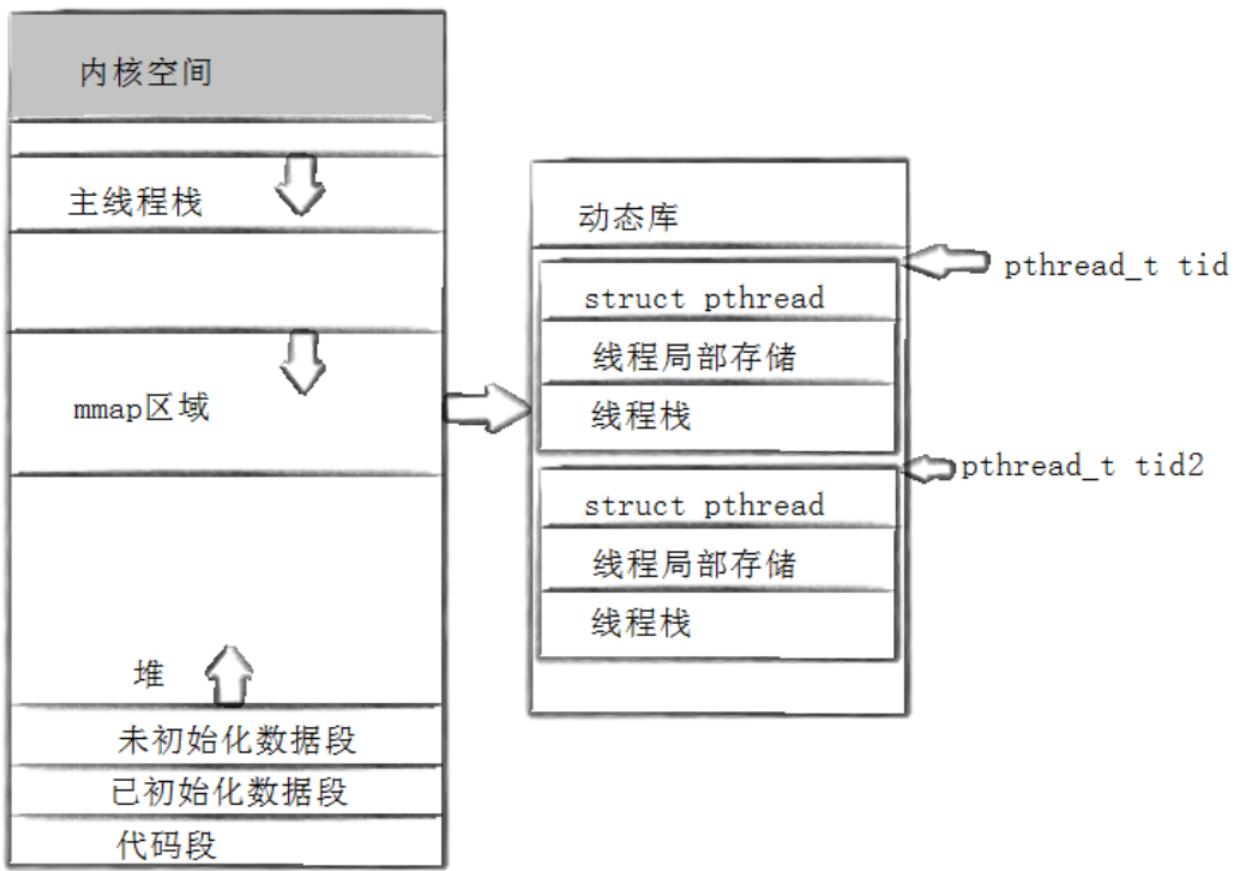
```

### ###线程ID及进程地址空间布局

- `pthread_create`函数会产生一个线程ID，存放在第一个参数指向的地址中。该线程ID和前面说的线程ID不是一回事。
- 前面讲的线程ID属于进程调度的范畴。因为线程是轻量级进程，是操作系统调度器的最小单位，所以需要一个数值来唯一表示该线程。
- `pthread_create`函数第一个参数指向一个虚拟内存单元，该内存单元的地址即为新创建线程的线程ID，属于NPTL线程库的范畴。线程库的后续操作，就是根据该线程ID来操作线程的。
- 线程库NPTL提供了`pthread_self`函数，可以获得线程自身的ID：

```
pthread_t pthread_self(void);
```

`pthread_t` 到底是什么类型呢？取决于实现。对于Linux目前实现的NPTL实现而言，`pthread_t`类型的线程ID，本质上就是一个进程地址空间上的一个地址。



### ###线程终止

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:

1. 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。
2. 线程可以调用`pthread_exit`终止自己。
3. 一个线程可以调用`pthread_cancel`终止同一进程中的另一个线程。

### `pthread_exit`函数

功能: 线程终止

原型

```
void pthread_exit(void *value_ptr);
```

参数

`value_ptr`: `value_ptr`不要指向一个局部变量。

返回值: 无返回值, 跟进程一样, 线程结束的时候无法返回到它的调用者 (自身)

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。

### `pthread_cancel`函数

功能: 取消一个执行中的线程

原型

```
int pthread_cancel(pthread_t thread);
```

参数

thread:线程ID

返回值: 成功返回0; 失败返回错误码

### ###线程等待 为什么需要线程等待?

- 已经退出的线程, 其空间没有被释放, 仍然在进程的地址空间内。
- 创建新的线程不会复用刚才退出线程的地址空间。

功能: 等待线程结束

原型

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

参数

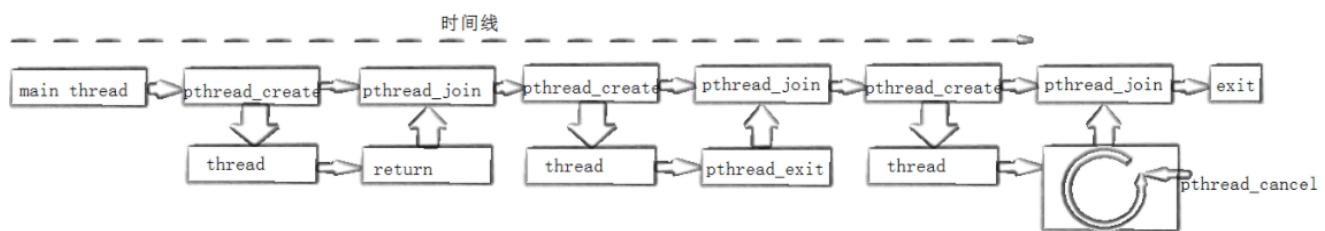
thread:线程ID

value\_ptr: 它指向一个指针, 后者指向线程的返回值

返回值: 成功返回0; 失败返回错误码

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:

1. 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。
2. 如果thread线程被别的线程调用pthread\_cancel异常终掉,value\_ptr所指向的单元里存放的是常数PTHREAD\_CANCELED。
3. 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。
4. 如果对thread线程的终止状态不感兴趣, 可以传NULL给value\_ptr参数。



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

void *thread1( void *arg )
{
    printf("thread 1 returning ... \n");
    int *p = (int*)malloc(sizeof(int));
    *p = 1;
    return (void*)p;
}
```

```

void *thread2( void *arg )
{
    printf("thread 2 exiting ...\\n");
    int *p = (int*)malloc(sizeof(int));
    *p = 2;
    pthread_exit((void*)p);
}

void *thread3( void *arg )
{
    while ( 1 ){ //
        printf("thread 3 is running ...\\n");
        sleep(1);
    }
    return NULL;
}

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

    // thread 1 return
    pthread_create(&tid, NULL, thread1, NULL);
    pthread_join(tid, &ret);
    printf("thread return, thread id %x, return code:%d\\n", tid, *(int*)ret);
    free(ret);

    // thread 2 exit
    pthread_create(&tid, NULL, thread2, NULL);
    pthread_join(tid, &ret);
    printf("thread return, thread id %x, return code:%d\\n", tid, *(int*)ret);
    free(ret);

    // thread 3 cancel by other
    pthread_create(&tid, NULL, thread3, NULL);
    sleep(3);
    pthread_cancel(tid);
    pthread_join(tid, &ret);
    if ( ret == PTHREAD_CANCELED )
        printf("thread return, thread id %x, return code:PTHREAD_CANCELED\\n", tid);
    else
        printf("thread return, thread id %x, return code:NULL\\n", tid);
}

```

运行结果:

```

[root@localhost linux]# ./a.out
thread 1 returning ...
thread return, thread id 5AA79700, return code:1
thread 2 exiting ...
thread return, thread id 5AA79700, return code:2
thread 3 is running ...
thread 3 is running ...

```

```
thread 3 is running ...
thread return, thread id 5AA79700, return code: PTHREAD_CANCELED
```

#### ##4. 分离线程

- 默认情况下，新创建的线程是joinable的，线程退出后，需要对其进行pthread\_join操作，否则无法释放资源，从而造成系统泄漏。
- 如果不关心线程的返回值，join是一种负担，这个时候，我们可以告诉系统，当线程退出时，自动释放线程资源。

```
int pthread_detach(pthread_t thread);
```

可以是线程组内其他线程对目标线程进行分离，也可以是线程自己分离：

```
pthread_detach(pthread_self());
```

joinable和分离是冲突的，一个线程不能既是joinable又是分离的。

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

void *thread_run( void * arg )
{
    pthread_detach(pthread_self());
    printf("%s\n", (char*)arg);
    return NULL;
}

int main( void )
{
    pthread_t tid;
    if ( pthread_create(&tid, NULL, thread_run, "thread1 run...") != 0 ) {
        printf("create thread error\n");
        return 1;
    }

    int ret = 0;
    sleep(1); //很重要，要让线程先分离，再等待

    if ( pthread_join(tid, NULL) == 0 ) {
        printf("pthread wait success\n");
        ret = 0;
    } else {
        printf("pthread wait failed\n");
        ret = 1;
    }
    return ret;
}
```

## ##5. Linux线程互斥

### ###进程线程间的互斥相关背景概念

- 临界资源：多线程执行流共享的资源就叫做临界资源
- 临界区：每个线程内部，访问临界资源的代码，就叫做临界区
- 互斥：任何时刻，互斥保证有且只有一个执行流进入临界区，访问临界资源，通常对临界资源起保护作用
- 原子性（后面讨论如何实现）：不会被任何调度机制打断的操作，该操作只有两态，要么完成，要么未完成

### ###互斥量mutex

- 大部分情况，线程使用的数据都是局部变量，变量的地址空间在线程栈空间内，这种情况，变量归属单个线程，其他线程无法获得这种变量。
- 但有时候，很多变量都需要在线程间共享，这样的变量称为共享变量，可以通过数据的共享，完成线程之间的交互。
- 多个线程并发的操作共享变量，会带来一些问题。

```
// 操作共享变量会有问题的售票系统代码
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

int ticket = 100;

void *route(void *arg)
{
    char *id = (char*)arg;
    while (1) {
        if (ticket > 0) {
            usleep(1000);
            printf("%s sells ticket:%d\n", id, ticket);
            ticket--;
        } else {
            break;
        }
    }
}

int main( void )
{
    pthread_t t1, t2, t3, t4;

    pthread_create(&t1, NULL, route, "thread 1");
    pthread_create(&t2, NULL, route, "thread 2");
    pthread_create(&t3, NULL, route, "thread 3");
    pthread_create(&t4, NULL, route, "thread 4");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
```

```
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
}
```

一次执行结果：

```
thread 4 sells ticket:100
...
thread 4 sells ticket:1
thread 2 sells ticket:0
thread 1 sells ticket:-1
thread 3 sells ticket:-2
```

为什么可能无法获得争取结果？

- `if` 语句判断条件为真以后，代码可以并发的切换到其他线程
- `usleep` 这个模拟漫长业务的过程，在这个漫长的业务过程中，可能有很多个线程会进入该代码段
- `--ticket` 操作本身不是一个原子操作

取出`ticket--`部分的汇编代码

```
objdump -d a.out > test.objdump
152  40064b:  8b 05 e3 04 20 00      mov    0x2004e3(%rip),%eax      # 600b34 <ticket>
153  400651:  83 e8 01                sub    $0x1,%eax
154  400654:  89 05 da 04 20 00      mov    %eax,0x2004da(%rip)    # 600b34 <ticket>
```

`--`操作并不是原子操作，而是对应三条汇编指令：

- `load`：将共享变量`ticket`从内存加载到寄存器中
- `update`：更新寄存器里面的值，执行`-1`操作
- `store`：将新值，从寄存器写回共享变量`ticket`的内存地址

要解决以上问题，需要做到三点：

- 代码必须要有互斥行为：当代码进入临界区执行时，不允许其他线程进入该临界区。
- 如果多个线程同时要求执行临界区的代码，并且临界区没有线程在执行，那么只能允许一个线程进入该临界区。
- 如果线程不在临界区中执行，那么该线程不能阻止其他线程进入临界区。

要做到这三点，本质上就是需要一把锁。Linux上提供的这把锁叫互斥量。



## 互斥量的接口

### 初始化互斥量

初始化互斥量有两种方法:

- 方法1, 静态分配:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```

- 方法2, 动态分配:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
```

参数:

mutex: 要初始化的互斥量

attr: NULL

### 销毁互斥量

销毁互斥量需要注意:

- 使用 PTHREAD\_MUTEX\_INITIALIZER 初始化的互斥量不需要销毁
- 不要销毁一个已经加锁的互斥量
- 已经销毁的互斥量, 要确保后面不会有线程再尝试加锁

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

### 互斥量加锁和解锁

```
int pthread_mutex_lock(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

调用 `pthread_lock` 时, 可能会遇到以下情况:

- 互斥量处于未锁状态，该函数会将互斥量锁定，同时返回成功
- 发起函数调用时，其他线程已经锁定互斥量，或者存在其他线程同时申请互斥量，但没有竞争到互斥量，那么pthread\_lock调用会陷入阻塞(执行流被挂起)，等待互斥量解锁。

改进上面的售票系统：

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

int ticket = 100;
pthread_mutex_t mutex;

void *route(void *arg)
{
    char *id = (char*)arg;
    while (1) {
        pthread_mutex_lock(&mutex);
        if (ticket > 0) {
            usleep(1000);
            printf("%s sells ticket:%d\n", id, ticket);
            ticket--;
            pthread_mutex_unlock(&mutex);
            // sched_yield(); 放弃CPU
        } else {
            pthread_mutex_unlock(&mutex);
            break;
        }
    }
}

int main( void )
{
    pthread_t t1, t2, t3, t4;

    pthread_mutex_init(&mutex, NULL);

    pthread_create(&t1, NULL, route, "thread 1");
    pthread_create(&t2, NULL, route, "thread 2");
    pthread_create(&t3, NULL, route, "thread 3");
    pthread_create(&t4, NULL, route, "thread 4");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    pthread_mutex_destroy(&mutex);
}
```

###互斥量实现原理探究

- 经过上面的例子，大家已经意识到单纯的 `i++` 或者 `++i` 都不是原子的，有可能会有数据一致性问题
- 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

```

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;

```

####可重入VS线程安全

####概念

- 线程安全：多个线程并发同一段代码时，不会出现不同的结果。常见对全局变量或者静态变量进行操作，并且没有锁保护的情况下，会出现该问题。
- 重入：同一个函数被不同的执行流调用，当前一个流程还没有执行完，就有其他的执行流再次进入，我们称之为重入。一个函数在重入的情况下，运行结果不会出现任何不同或者任何问题，则该函数被称为可重入函数，否则，是不可重入函数。

####常见的线程不安全的情况

- 不保护共享变量的函数
- 函数状态随着被调用，状态发生变化的函数
- 返回指向静态变量指针的函数
- 调用线程不安全函数的函数

## 常见的线程安全的情况

- 每个线程对全局变量或者静态变量只有读取的权限，而没有写入的权限，一般来说这些线程是安全的
- 类或者接口对于线程来说都是原子操作
- 多个线程之间的切换不会导致该接口的执行结果存在二义性

## 常见不可重入的情况

- 调用了 `malloc/free` 函数，因为 `malloc` 函数是用全局链表来管理堆的
- 调用了标准 I/O 库函数，标准 I/O 库的很多实现都以不可重入的方式使用全局数据结构
- 可重入函数体内使用了静态的数据结构

## 常见可重入的情况

- 不使用全局变量或静态变量
- 不使用用malloc或者new开辟出的空间
- 不调用不可重入函数
- 不返回静态或全局数据，所有数据都有函数的调用者提供
- 使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据

## 可重入与线程安全联系

- 函数是可重入的，那就是线程安全的
- 函数是不可重入的，那就不能由多个线程使用，有可能引发线程安全问题
- 如果一个函数中有全局变量，那么这个函数既不是线程安全也不是可重入的。

## 可重入与线程安全区别

- 可重入函数是线程安全函数的一种
- 线程安全不一定是可重入的，而可重入函数则一定是线程安全的。
- 如果将对临界资源的访问加上锁，则这个函数是线程安全的，但如果这个重入函数若锁还未释放则会产生死锁，因此是不可重入的。

# 6. 常见锁概念

## 死锁

- 死锁是指在一组进程中的各个进程均占有不会释放的资源，但因互相申请被其他进程所占用而处于的一种永久等待状态。

### ###死锁四个必要条件

- 互斥条件：一个资源每次只能被一个执行流使用
- 请求与保持条件：一个执行流因请求资源而阻塞时，对已获得的资源保持不放
- 不剥夺条件：一个执行流已获得的资源，在未使用完之前，不能强行剥夺
- 循环等待条件：若干执行流之间形成一种头尾相接的循环等待资源的关系

### ###避免死锁

- 破坏死锁的四个必要条件
- 加锁顺序一致
- 避免锁未释放的场景
- 资源一次性分配

### ###避免死锁算法

- 死锁检测算法(了解)
- 银行家算法（了解）

## ##7. Linux线程同步

### ###条件变量

- 当一个线程互斥地访问某个变量时，它可能发现在其它线程改变状态之前，它什么也做不了。
- 例如一个线程访问队列时，发现队列为空，它只能等待，直到其它线程将一个节点添加到队列中。这种情况就需要用到条件变量。

### ####同步概念与竞态条件

- 同步：在保证数据安全的前提下，让线程能够按照某种特定的顺序访问临界资源，从而有效避免饥饿问题，叫做同步
- 竞态条件：因为时序问题，而导致程序异常，我们称之为竞态条件。在线程场景下，这种问题也不难理解

#### ####条件变量函数 初始化

```
int pthread_cond_init(pthread_cond_t *restrict cond,const pthread_condattr_t *restrict attr);
```

参数：

cond: 要初始化的条件变量  
attr: NULL

#### 销毁

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

#### 等待条件满足

```
int pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t *restrict mutex);
```

参数：

cond: 要在这个条件变量上等待  
mutex: 互斥量，后面详细解释

#### 唤醒等待

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

#### 简单案例：

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

pthread_cond_t cond;
pthread_mutex_t mutex;

void *r1( void *arg )
{
    while ( 1 ){
        pthread_cond_wait(&cond, &mutex);
        printf("活动\n");
    }
}

void *r2(void *arg )
{
    while ( 1 ) {
```

```

    pthread_cond_signal(&cond);
    sleep(1);
}

int main( void )
{
    pthread_t t1, t2;

    pthread_cond_init(&cond, NULL);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&t1, NULL, r1, NULL);
    pthread_create(&t2, NULL, r2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

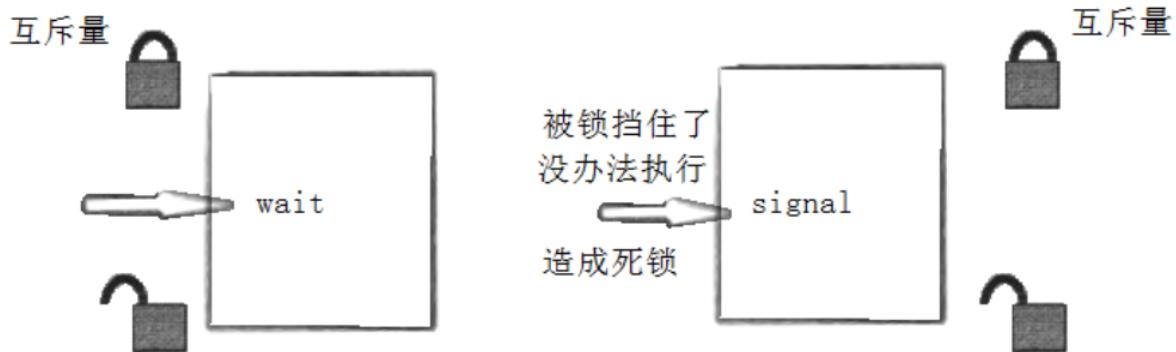
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
}

```

[root@localhost linux]# ./a.out  
活动  
活动  
活动

### 为什么 `pthread_cond_wait` 需要互斥量？

- 条件等待是线程间同步的一种手段，如果只有一个线程，条件不满足，一直等下去都不会满足，所以必须要有一个线程通过某些操作，改变共享变量，使原先不满足的条件变得满足，并且友好的通知等待在条件变量上的线程。
- 条件不会无缘无故的突然变得满足了，必然会牵扯到共享数据的变化。所以一定要用互斥锁来保护。没有互斥锁就无法安全的获取和修改共享数据。



- 按照上面的说法，我们设计出如下的代码：先上锁，发现条件不满足，解锁，然后等待在条件变量上不行了，如下代码：

```
// 错误的设计
pthread_mutex_lock(&mutex);
while (condition_is_false) {
    pthread_mutex_unlock(&mutex);
    //解锁之后，等待之前，条件可能已经满足，信号已经发出，但是该信号可能被错过
    pthread_cond_wait(&cond);
    pthread_mutex_lock(&mutex);
}
pthread_mutex_unlock(&mutex);
```

- 由于解锁和等待不是原子操作。调用解锁之后，`pthread_cond_wait`之前，如果有其他线程获取到互斥量，摒弃条件满足，发送了信号，那么`pthread_cond_wait`将错过这个信号，可能会导致线程永远阻塞在这个`pthread_cond_wait`。所以解锁和等待必须是一个原子操作。
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t * mutex);`进入该函数后，会去看条件量等于0不？等于，就把互斥量变成1，直到`cond_wait`返回，把条件量改成1，把互斥量恢复成原样。

### ### 条件变量使用规范

- 等待条件代码

```
pthread_mutex_lock(&mutex);
while (条件为假)
    pthread_cond_wait(cond, mutex);
修改条件
pthread_mutex_unlock(&mutex);
```

- 给条件发送信号代码

```
pthread_mutex_lock(&mutex);
设置条件为真
pthread_cond_signal(cond);
pthread_mutex_unlock(&mutex);
```

## 8. 生产者消费者模型

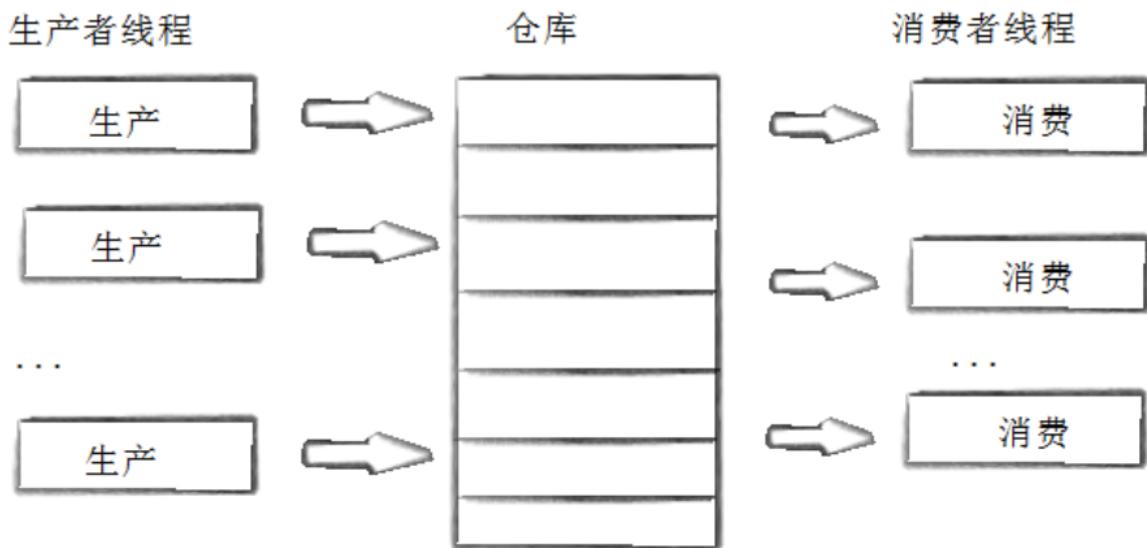
- 321原则(便于记忆)

### 为什么要使用生产者消费者模型

生产者消费者模式就是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而是通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费者解耦的。

### 生产者消费者模型优点

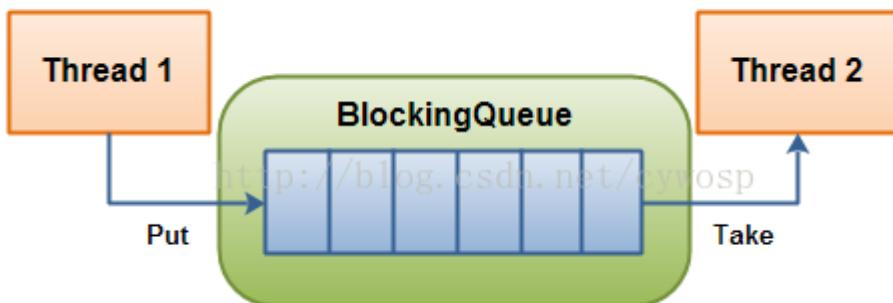
- 解耦
- 支持并发
- 支持忙闲不均



## 基于BlockingQueue的生产者消费者模型

### BlockingQueue

在多线程编程中阻塞队列(Blocking Queue)是一种常用于实现生产者和消费者模型的数据结构。其与普通的队列区别在于，当队列为空时，从队列获取元素的操作将会被阻塞，直到队列中被放入了元素；当队列满时，往队列里存放元素的操作也会被阻塞，直到有元素被从队列中取出(以上的操作都是基于不同的线程来说的，线程在对阻塞队列进程操作时会被阻塞)



## C++ queue模拟阻塞队列的生产消费模型

代码：

- 为了便于同学们理解，我们以单生产者，单消费者，来进行讲解。

```
#include <iostream>
#include <queue>
#include <stdlib.h>
#include <pthread.h>

#define NUM 8

class BlockQueue{
    private:
```

```

    std::queue<int> q;
    int cap;
    pthread_mutex_t lock;
    pthread_cond_t full;
    pthread_cond_t empty;

private:
    void LockQueue()
    {
        pthread_mutex_lock(&lock);
    }
    void unLockQueue()
    {
        pthread_mutex_unlock(&lock);
    }
    void ProductWait()
    {
        pthread_cond_wait(&full, &lock);
    }
    void Consumewait()
    {
        pthread_cond_wait(&empty, &lock);
    }
    void NotifyProduct()
    {
        pthread_cond_signal(&full);
    }
    void NotifyConsume()
    {
        pthread_cond_signal(&empty);
    }
    bool IsEmpty()
    {
        return ( q.size() == 0 ? true : false );
    }
    bool IsFull()
    {
        return ( q.size() == cap ? true : false );
    }

public:
    BlockQueue(int _cap = NUM):cap(_cap)
    {
        pthread_mutex_init(&lock, NULL);
        pthread_cond_init(&full, NULL);
        pthread_cond_init(&empty, NULL);
    }
    void PushData(const int &data)
    {
        LockQueue();
        while(IsFull()){
            NotifyConsume();
            std::cout << "queue full, notify consume data, product stop." << std::endl;

```

```

        Productwait();
    }
    q.push(data);
//    NotifyConsume();
    UnLockQueue();
}
void PopData(int &data)
{
    LockQueue();
    while(IsEmpty()){
        NotifyProduct();
        std::cout << "queue empty, notify product data, consume stop." << std::endl;
        Consumewait();
    }
    data = q.front();
    q.pop();
//    NotifyProduct();
    UnLockQueue();
}
~BlockQueue()
{
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&full);
    pthread_cond_destroy(&empty);
}
};

void *consumer(void *arg)
{
    BlockQueue *bqp = (BlockQueue*)arg;
    int data;
    for( ; ; ){
        bqp->PopData(data);
        std::cout << "Consume data done : " << data << std::endl;
    }
}

//more faster
void *producter(void *arg)
{
    BlockQueue *bqp = (BlockQueue*)arg;
    srand((unsigned long)time(NULL));
    for( ; ; ){
        int data = rand() % 1024;
        bqp->PushData(data);
        std::cout << "Product data done: " << data << std::endl;
//        sleep(1);
    }
}

int main()
{
    BlockQueue bq;

```

```
pthread_t c,p;
pthread_create(&c, NULL, consumer, (void*)&bq);
pthread_create(&p, NULL, producter, (void*)&bq);

pthread_join(c, NULL);
pthread_join(p, NULL);
return 0;
}
```

## POSIX信号量

POSIX信号量和SystemV信号量作用相同，都是用于同步操作，达到无冲突的访问共享资源目的。但POSIX可以用于线程间同步。

### 初始化信号量

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

参数：

- pshared:0表示线程间共享，非零表示进程间共享
- value: 信号量初始值

### 销毁信号量

```
int sem_destroy(sem_t *sem);
```

### 等待信号量

功能：等待信号量，会将信号量的值减1

```
int sem_wait(sem_t *sem); //P()
```

### 发布信号量

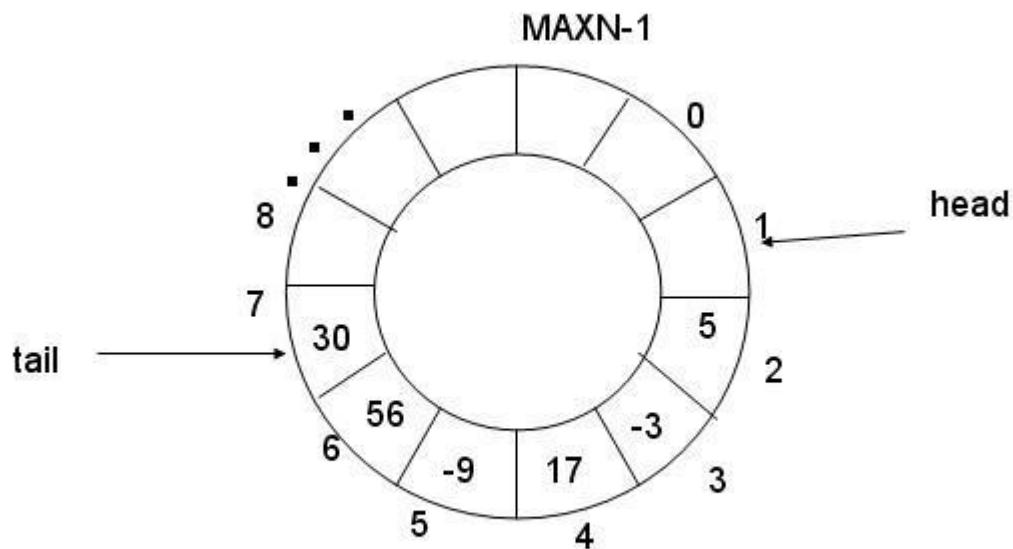
功能：发布信号量，表示资源使用完毕，可以归还资源了。将信号量值加1。

```
int sem_post(sem_t *sem); //V()
```

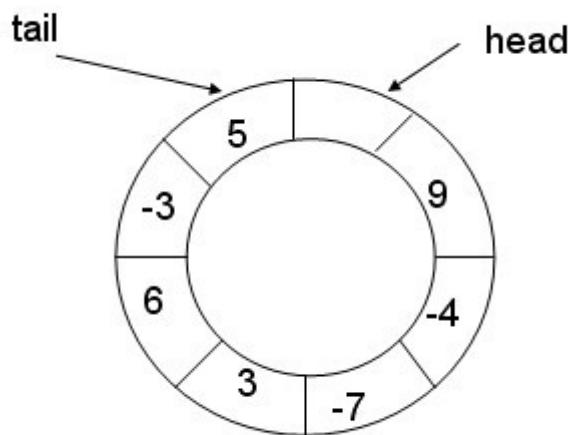
上一节生产者-消费者的例子是基于queue的，其空间可以动态分配，现在基于固定大小的环形队列重写这个程序（POSIX信号量）：

## 基于环形队列的生产消费模型

- 环形队列采用数组模拟，用模运算来模拟环状特性



- 环形结构起始状态和结束状态都是一样的，不好判断为空或者为满，所以可以通过加计数器或者标记位来判断满或者空。另外也可以预留一个空的位置，作为满的状态



- 但是我们现在有信号量这个计数器，就很简单的进行多线程间的同步过程

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

#define NUM 16

class RingQueue{
```

```

private:
    std::vector<int> q;
    int cap;
    sem_t data_sem;
    sem_t space_sem;
    int consume_step;
    int product_step;

public:
    RingQueue(int _cap = NUM):q(_cap),cap(_cap)
    {
        sem_init(&data_sem, 0, 0);
        sem_init(&space_sem, 0, cap);
        consume_step = 0;
        product_step = 0;
    }
    void PutData(const int &data)
    {
        sem_wait(&space_sem); // P
        q[consume_step] = data;
        consume_step++;
        consume_step %= cap;
        sem_post(&data_sem); //V
    }
    void GetData(int &data)
    {
        sem_wait(&data_sem);
        data = q[product_step];
        product_step++;
        product_step %= cap;
        sem_post(&space_sem);
    }
    ~RingQueue()
    {
        sem_destroy(&data_sem);
        sem_destroy(&space_sem);
    }
};

void *consumer(void *arg)
{
    RingQueue *rqp = (RingQueue*)arg;
    int data;
    for( ; ; ){
        rqp->GetData(data);
        std::cout << "Consume data done : " << data << std::endl;
        sleep(1);
    }
}

//more faster
void *producter(void *arg)
{
    RingQueue *rqp = (RingQueue*)arg;
}

```

```

    srand((unsigned long)time(NULL));
    for( ; ; ){
        int data = rand() % 1024;
        rqp->PutData(data);
        std::cout << "Prodoct data done: " << data << std::endl;
        // sleep(1);
    }
}

int main()
{
    RingQueue rq;
    pthread_t c,p;
    pthread_create(&c, NULL, consumer, (void*)&rq);
    pthread_create(&p, NULL, producter, (void*)&rq);

    pthread_join(c, NULL);
    pthread_join(p, NULL);
}

```

## 9. 线程池

```

/*threadpool.h*/
/* 线程池：
 *      一种线程使用模式。线程过多会带来调度开销，进而影响缓存局部性和整体性能。而线程池维护着多个线程，等待着
 *      监督管理者分配可并发执行的任务。这避免了在处理短时间任务时创建与销毁线程的代价。线程池不仅能够保证内核的充分利
 *      用，还能防止过分调度。可用线程数量应该取决于可用的并发处理器、处理器内核、内存、网络sockets等的数量。
 *      线程池的应用场景：
 *          1. 需要大量的线程来完成任务，且完成任务的时间比较短。 WEB服务器完成网页请求这样的任务，使用线程池技
 *              术是非常合适的。因为单个任务小，而任务数量巨大，你可以想象一个热门网站的点击次数。但对于长时间的任务，比如一个
 *              Telnet连接请求，线程池的优点就不明显了。因为Telnet会话时间比线程的创建时间大多了。
 *          2. 对性能要求苛刻的应用，比如要求服务器迅速响应客户请求。
 *          3. 接受突发性的大量请求，但不至于使服务器因此产生大量线程的应用。突发性大量客户请求，在没有线程池情
 *      况下，将产生大量线程，虽然理论上大部分操作系统线程数目最大值不是问题，短时间内产生大量线程可能使内存到达极限，
 *      出现错误。
 *      线程池的种类：
 *      线程池示例：
 *          1. 创建固定数量线程池，循环从任务队列中获取任务对象，
 *          2. 获取到任务对象后，执行任务对象中的任务接口
 */

```

```

/*threadpool.hpp*/
#ifndef __M_TP_H__
#define __M_TP_H__
#include <iostream>
#include <queue>
#include <pthread.h>

#define MAX_THREAD 5
typedef bool (*handler_t)(int);
class ThreadTask
{

```

```

private:
    int _data;
    handler_t _handler;
public:
    ThreadTask():_data(-1), _handler(NULL) {}
    ThreadTask(int data, handler_t handler) {
        _data= data;
        _handler = handler;
    }
    void SetTask(int data, handler_t handler) {
        _data = data;
        _handler = handler;
    }
    void Run() {
        _handler(_data);
    }
};

class ThreadPool
{
private:
    int _thread_max;
    int _thread_cur;
    bool _tp_quit;
    std::queue<ThreadTask *> _task_queue;
    pthread_mutex_t _lock;
    pthread_cond_t _cond;
private:
    void LockQueue() {
        pthread_mutex_lock(&_lock);
    }
    void UnLockQueue() {
        pthread_mutex_unlock(&_lock);
    }
    void WakeupOne() {
        pthread_cond_signal(&_cond);
    }
    void WakeupAll() {
        pthread_cond_broadcast(&_cond);
    }
    void ThreadQuit() {
        _thread_cur--;
        UnLockQueue();
        pthread_exit(NULL);
    }
    void ThreadWait(){
        if (_tp_quit) {
            ThreadQuit();
        }
        pthread_cond_wait(&_cond, &_lock);
    }
    bool IsEmpty() {
        return _task_queue.empty();
    }
};

```

```

    }

    static void *thr_start(void *arg) {
        ThreadPool *tp = (ThreadPool*)arg;
        while(1) {
            tp->LockQueue();
            while(tp->IsEmpty()) {
                tp->Threadwait();
            }
            ThreadTask *tt;
            tp->PopTask(&tt);
            tp->UnLockQueue();
            tt->Run();
            delete tt;
        }
        return NULL;
    }

public:
    ThreadPool(int max=MAX_THREAD):_thread_max(max), _thread_cur(max),
    _tp_quit(false) {
        pthread_mutex_init(&_lock, NULL);
        pthread_cond_init(&_cond, NULL);
    }

    ~ThreadPool() {
        pthread_mutex_destroy(&_lock);
        pthread_cond_destroy(&_cond);
    }

    bool PoolInit() {
        pthread_t tid;
        for (int i = 0; i < _thread_max; i++) {
            int ret = pthread_create(&tid, NULL, thr_start, this);
            if (ret != 0) {
                std::cout<<"create pool thread error\n";
                return false;
            }
        }
        return true;
    }

    bool PushTask(ThreadTask *tt) {
        LockQueue();
        if (_tp_quit) {
            UnLockQueue();
            return false;
        }
        _task_queue.push(tt);
        WakeUpOne();
        UnLockQueue();
        return true;
    }

    bool PopTask(ThreadTask **tt) {
        *tt = _task_queue.front();
        _task_queue.pop();
        return true;
    }
}

```

```

        bool Poolquit() {
            LockQueue();
            _tp_quit = true;
            UnLockQueue();
            while(_thread_cur > 0) {
                WakeUpAll();
                usleep(1000);
            }
            return true;
        }
    };
#endif

/*main.cpp*/
bool handler(int data)
{
    srand(time(NULL));
    int n = rand() % 5;
    printf("Thread: %p Run Task: %d--sleep %d sec\n", pthread_self(), data, n);
    sleep(n);
    return true;
}
int main()
{
    int i;

    ThreadPool pool;
    pool.PoolInit();
    for (i = 0; i < 10; i++) {
        ThreadTask *tt = new ThreadTask(i, handler);
        pool.PushTask(tt);
    }

    pool.Poolquit();
    return 0;
}

```

```
g++ -std=c++0x test.cpp -o test -lpthread -lrt
```

## 10. 线程安全的单例模式

### 什么是单例模式

单例模式是一种 "经典的, 常用的, 常考的" **设计模式**.

### 什么是设计模式

IT行业这么火, 涌入的人很多. 俗话说林子大了啥鸟都有. 大佬和菜鸡们两极分化的越来越严重. 为了让菜鸡们不太拖大佬的后腿, 于是大佬们针对一些经典的常见的场景, 给定了一些对应的解决方案, 这个就是 **设计模式**

## 单例模式的特点

某些类, 只应该具有一个对象(实例), 就称之为单例.

例如一个男人只能有一个媳妇.

在很多服务器开发场景中, 经常需要让服务器加载很多的数据 (上百G) 到内存中. 此时往往要用一个单例的类来管理这些数据.

## 饿汉实现方式和懒汉实现方式

[洗完的例子]

吃完饭, 立刻洗碗, 这种就是饿汉方式. 因为下一顿吃的时候可以立刻拿着碗就能吃饭.

吃完饭, 先把碗放下, 然后下一顿饭用到这个碗了再洗碗, 就是懒汉方式.

懒汉方式最核心的思想是 "延时加载". 从而能够优化服务器的启动速度.

## 饿汉方式实现单例模式

```
template <typename T>
class Singleton {
    static T data;
public:
    static T* GetInstance() {
        return &data;
    }
};
```

只要通过 Singleton 这个包装类来使用 T 对象, 则一个进程中只有一个 T 对象的实例.

## 懒汉方式实现单例模式

```
template <typename T>
class Singleton {
    static T* inst;
public:
    static T* GetInstance() {
        if (inst == NULL) {
            inst = new T();
        }
        return inst;
    }
};
```

存在一个严重的问题, 线程不安全.

第一次调用 GetInstance 的时候, 如果两个线程同时调用, 可能会创建出两份 T 对象的实例.

但是后续再次调用, 就没有问题了.

## 懒汉方式实现单例模式(线程安全版本)

```

// 懒汉模式，线程安全
template <typename T>
class Singleton {
    volatile static T* inst; // 需要设置 volatile 关键字，否则可能被编译器优化.
    static std::mutex lock;
public:
    static T* GetInstance() {
        if (inst == NULL) { // 双重判定空指针，降低锁冲突的概率，提高性能.
            lock.lock(); // 使用互斥锁，保证多线程情况下也只调用一次 new.
            if (inst == NULL) {
                inst = new T();
            }
            lock.unlock();
        }
        return inst;
    }
};

```

注意事项:

1. 加锁解锁的位置
2. 双重 if 判定, 避免不必要的锁竞争
3. volatile关键字防止过度优化

## 11. STL,智能指针和线程安全

### STL中的容器是否是线程安全的?

不是.

原因是, STL 的设计初衷是将性能挖掘到极致, 而一旦涉及到加锁保证线程安全, 会对性能造成巨大的影响.

而且对于不同的容器, 加锁方式的不同, 性能可能也不同(例如hash表的锁表和锁桶).

因此 STL 默认不是线程安全. 如果需要在多线程环境下使用, 往往需要调用者自行保证线程安全.

### 智能指针是否是线程安全的?

对于 unique\_ptr, 由于只是在当前代码块范围内生效, 因此不涉及线程安全问题.

对于 shared\_ptr, 多个对象需要共用一个引用计数变量, 所以会存在线程安全问题. 但是标准库实现的时候考虑到了这个问题, 基于原子操作(CAS)的方式保证 shared\_ptr 能够高效, 原子的操作引用计数.

## 12. 其他常见的各种锁

- 悲观锁: 在每次取数据时, 总是担心数据会被其他线程修改, 所以会在取数据前先加锁 (读锁, 写锁, 行锁等), 当其他线程想要访问数据时, 被阻塞挂起。
- 乐观锁: 每次取数据时候, 总是乐观的认为数据不会被其他线程修改, 因此不上锁。但是在更新数据前, 会判断其他数据在更新前有没有对数据进行修改。主要采用两种方式: 版本号机制和CAS操作。
- CAS操作: 当需要更新数据时, 判断当前内存值和之前取得的值是否相等。如果相等则用新值更新。若不等则失败, 失败则重试, 一般是一个自旋的过程, 即不断重试。
- 自旋锁, 公平锁, 非公平锁?

# 13. 读者写者问题 [选学]

## 读写锁

在编写多线程的时候，有一种情况是十分常见的。那就是，有些公共数据修改的机会比较少。相比较改写，它们读的机会反而高的多。通常而言，在读的过程中，往往伴随着查找的操作，中间耗时很长。给这种代码段加锁，会极大地降低我们程序的效率。那么有没有一种方法，可以专门处理这种多读少写的情况呢？有，那就是读写锁。

读写锁的行为

当前锁状态	读锁请求	写锁请求
无锁	可以	可以
读锁	可以	阻塞
写锁	阻塞	阻塞

- 注意：写独占，读共享，读锁优先级高

### 读写锁接口

#### 设置读写优先

```
int pthread_rwlockattr_setkind_np(pthread_rwlockattr_t *attr, int pref);  
/*  
pref 共有 3 种选择
```

PTHREAD\_RWLOCK\_PREFER\_READER\_NP (默认设置) 读者优先，可能会导致写者饥饿情况

PTHREAD\_RWLOCK\_PREFER\_WRITER\_NP 写者优先，目前有 BUG，导致表现行为和 PTHREAD\_RWLOCK\_PREFER\_READER\_NP 一致

PTHREAD\_RWLOCK\_PREFER\_WRITER\_NONRECURSIVE\_NP 写者优先，但写者不能递归加锁  
\*/

#### 初始化

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,const pthread_rwlockattr_t *restrict attr);
```

#### 销毁

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

#### 加锁和解锁

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);  
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);  
  
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

#### 读写锁案例：

```
#include <vector>
#include <sstream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <pthread.h>

volatile int ticket = 1000;
pthread_rwlock_t rwlock;

void * reader(void * arg)
{
    char *id = (char *)arg;
    while (1) {
        pthread_rwlock_rdlock(&rwlock);
        if (ticket <= 0) {
            pthread_rwlock_unlock(&rwlock);
            break;
        }
        printf("%s: %d\n", id, ticket);
        pthread_rwlock_unlock(&rwlock);
        usleep(1);
    }
    return nullptr;
}

void * writer(void * arg)
{
    char *id = (char *)arg;
    while (1) {
        pthread_rwlock_wrlock(&rwlock);
        if (ticket <= 0) {
            pthread_rwlock_unlock(&rwlock);
            break;
        }
        printf("%s: %d\n", id, --ticket);
        pthread_rwlock_unlock(&rwlock);
        usleep(1);
    }
    return nullptr;
}

struct ThreadAttr
{
    pthread_t tid;
    std::string id;
```

```
};

std::string create_reader_id(std::size_t i)
{
    // 利用 ostringstream 进行 string 拼接
    std::ostringstream oss("thread reader ", std::ios_base::ate);
    oss << i;
    return oss.str();
}

std::string create_writer_id(std::size_t i)
{
    // 利用 ostringstream 进行 string 拼接
    std::ostringstream oss("thread writer ", std::ios_base::ate);
    oss << i;
    return oss.str();
}

void init_readers(std::vector<ThreadAttr>& vec)
{
    for (std::size_t i = 0; i < vec.size(); ++i) {
        vec[i].id = create_reader_id(i);

        pthread_create(&vec[i].tid, nullptr, reader, (void *)vec[i].id.c_str());
    }
}

void init_writers(std::vector<ThreadAttr>& vec)
{
    for (std::size_t i = 0; i < vec.size(); ++i) {
        vec[i].id = create_writer_id(i);

        pthread_create(&vec[i].tid, nullptr, writer, (void *)vec[i].id.c_str());
    }
}

void join_threads(std::vector<ThreadAttr> const& vec)
{
    // 我们按创建的 逆序 来进行线程的回收
    for (std::vector<ThreadAttr>::const_reverse_iterator it = vec.rbegin(); it != vec.rend(); ++it) {
        pthread_t const& tid = it->tid;
        pthread_join(tid, nullptr);
    }
}

void init_rwlock()
```

```
{  
#if 0    // 写优先  
    pthread_rwlockattr_t attr;  
    pthread_rwlockattr_init(&attr);  
    pthread_rwlockattr_setkind_np(&attr, PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP);  
    pthread_rwlock_init(&rwlock, &attr);  
    pthread_rwlockattr_destroy(&attr);  
#else    // 读优先, 会造成写饥饿  
    pthread_rwlock_init(&rwlock, nullptr);  
#endif  
}  
  
int main()  
{  
    // 测试效果不明显的情况下, 可以加大 reader_nr  
    // 但也不能太大, 超过一定阈值后系统就调度不了主线程了  
    const std::size_t reader_nr = 1000;  
    const std::size_t writer_nr = 2;  
  
    std::vector<ThreadAttr> readers(reader_nr);  
    std::vector<ThreadAttr> writers(writer_nr);  
  
    init_rwlock();  
  
    init_readers(readers);  
    init_writers(writers);  
  
    join_threads(writers);  
    join_threads(readers);  
  
    pthread_rwlock_destroy(&rwlock);  
}
```

```
main: main.cpp  
g++ -std=c++11 -Wall -Werror $^ -o $@ -lpthread
```

```
thread reader 180: 1000
thread reader 929: 1000
thread reader 285: 1000
thread reader 916: 1000
thread reader 93: 1000
thread reader 172: 1000
thread reader 752: 1000
thread reader 589: 1000
thread reader 727: 1000
thread reader 701: 1000
thread reader 375: 1000
thread reader 255: 1000
thread reader 213: 1000
thread reader 499: 1000
thread reader 318: 1000
thread reader 126: 1000
thread reader 158: 1000
thread reader 256: 1000
thread reader 444: 1000
thread reader 829: 1000
```

# 网络基础(一)

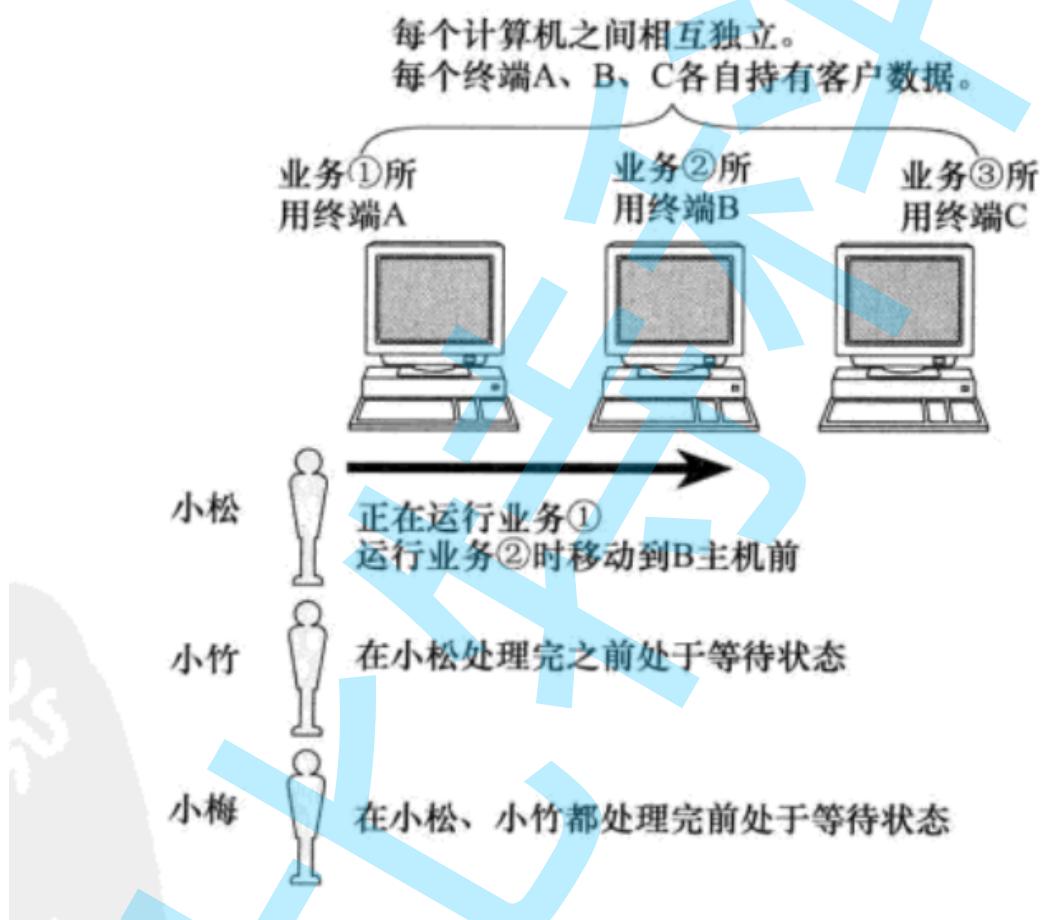
## 本节重点

- 了解网络发展背景, 对局域网/广域网的概念有基本认识;
- 了解网络协议的意义, 重点理解TCP/IP五层结构模型;
- 学习网络传输的基本流程, 理解封装和分用;

## 计算机网络背景

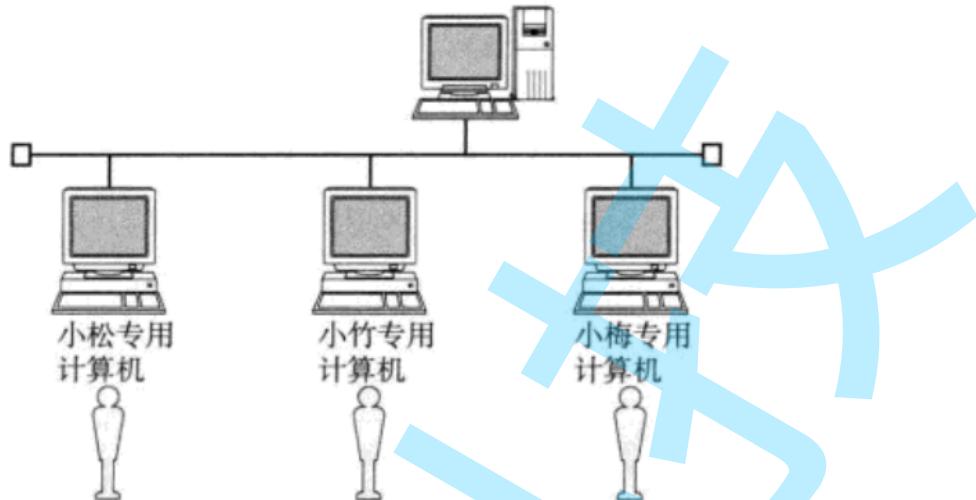
### 网络发展

独立模式: 计算机之间相互独立;



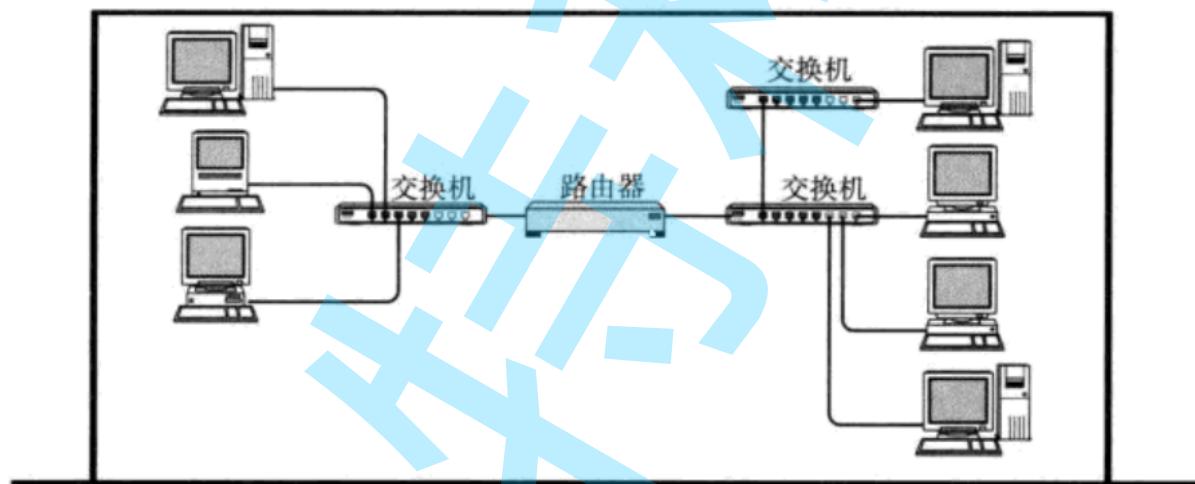
网络互联: 多台计算机连接在一起, 完成数据共享;

业务①~③所用服务器

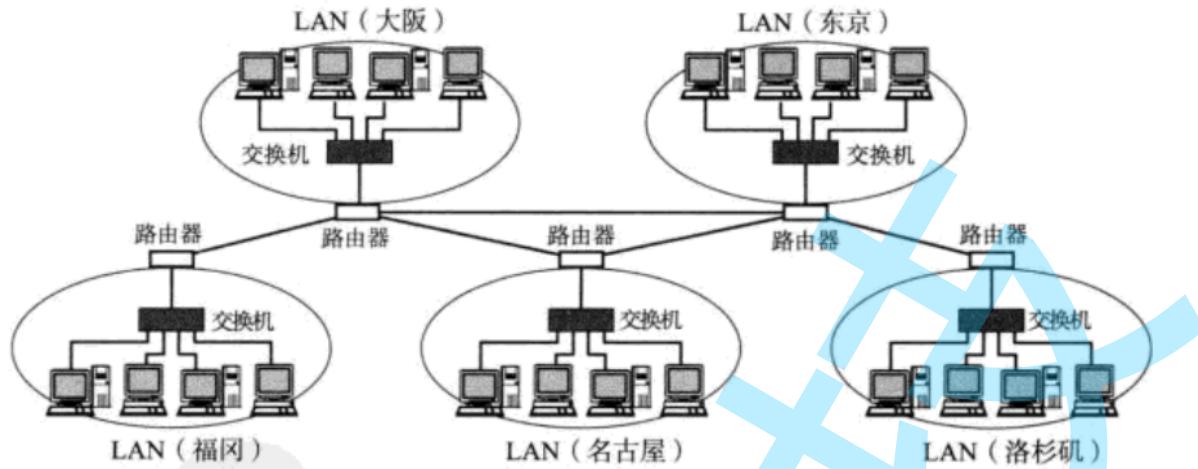


每个人都使用各自独立的计算机，业务①、②、③之间随时自由切换。  
共享数据由服务器集中管理。

局域网LAN: 计算机数量更多了, 通过交换机和路由器连接在一起;



广域网WAN: 将远隔千里的计算机都连在一起;



所谓 "局域网" 和 "广域网" 只是一个相对的概念。比如, 我们有 "天朝特色" 的广域网, 也可以看做一个比较大的局域网。

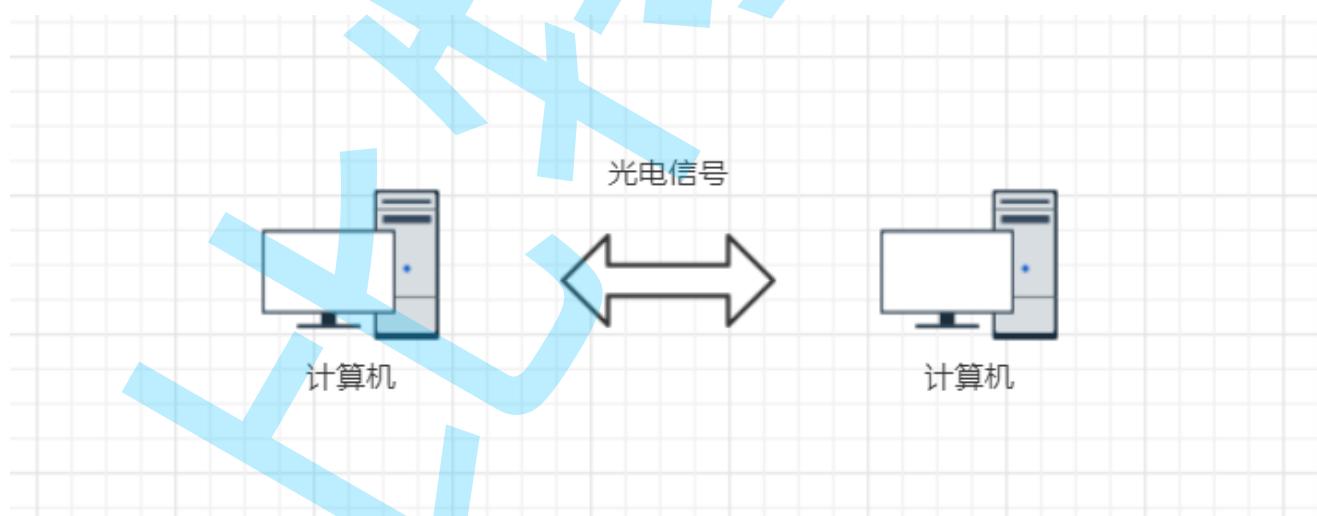


你知道的太多了

## 认识 "协议"

"协议" 是一种约定。

### 高小琴例子



计算机之间的传输媒介是光信号和电信号。通过 "频率" 和 "强弱" 来表示 0 和 1 这样的信息。要想传递各种不同的信息, 就需要约定好双方的数据格式。

思考: 只要通信的两台主机, 约定好协议就可以了么?

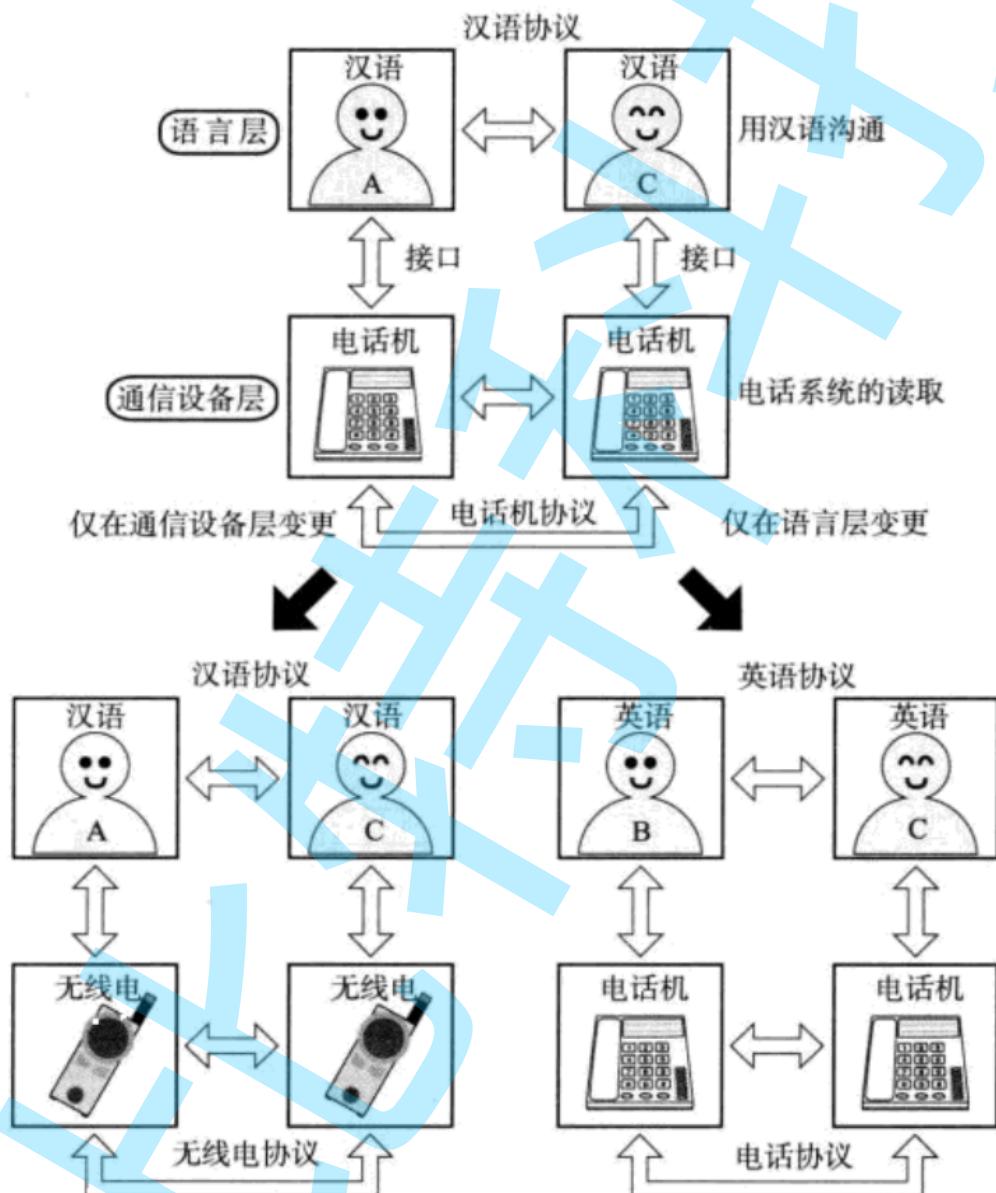
### 方言例子

- 计算机生产厂商有很多;
- 计算机操作系统, 也有很多;
- 计算机网络硬件设备, 还是有很多;
- 如何让这些不同厂商之间生产的计算机能够相互顺畅的通信? 就需要有人站出来, 约定一个共同的标准, 大家都来遵守, 这就是 **网络协议**;

## 网络协议初识

### 协议分层

打电话例子

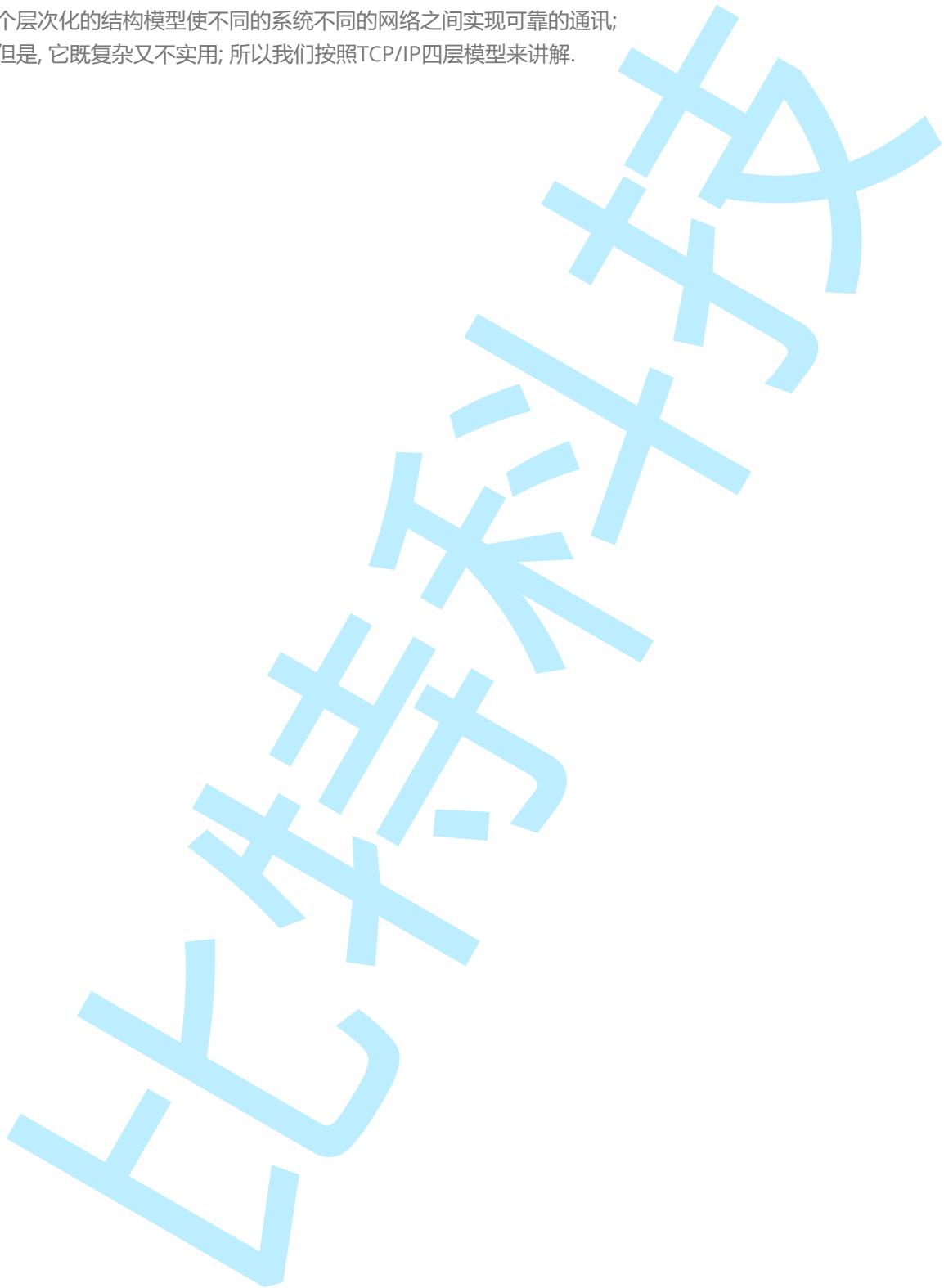


在这个例子中, 我们的协议只有两层; 但是实际的网络通信会更加复杂, 需要分更多的层次.

分层最大的好处在于 "封装" . 面向对象例子

### OSI七层模型

- OSI (Open System Interconnection, 开放系统互连) 七层网络模型称为开放式系统互联参考模型，是一个逻辑上的定义和规范；
- 把网络从逻辑上分为了7层. 每一层都有相关、相对应的物理设备，比如路由器，交换机；
- OSI 七层模型是一种框架性的设计方法，其最主要的功能使就是帮助不同类型的主机实现数据传输；
- 它的最大优点是将服务、接口和协议这三个概念明确地区分开来，概念清楚，理论也比较完整. 通过七个层次化的结构模型使不同的系统不同的网络之间实现可靠的通讯；
- 但是，它既复杂又不实用；所以我们按照TCP/IP四层模型来讲解.



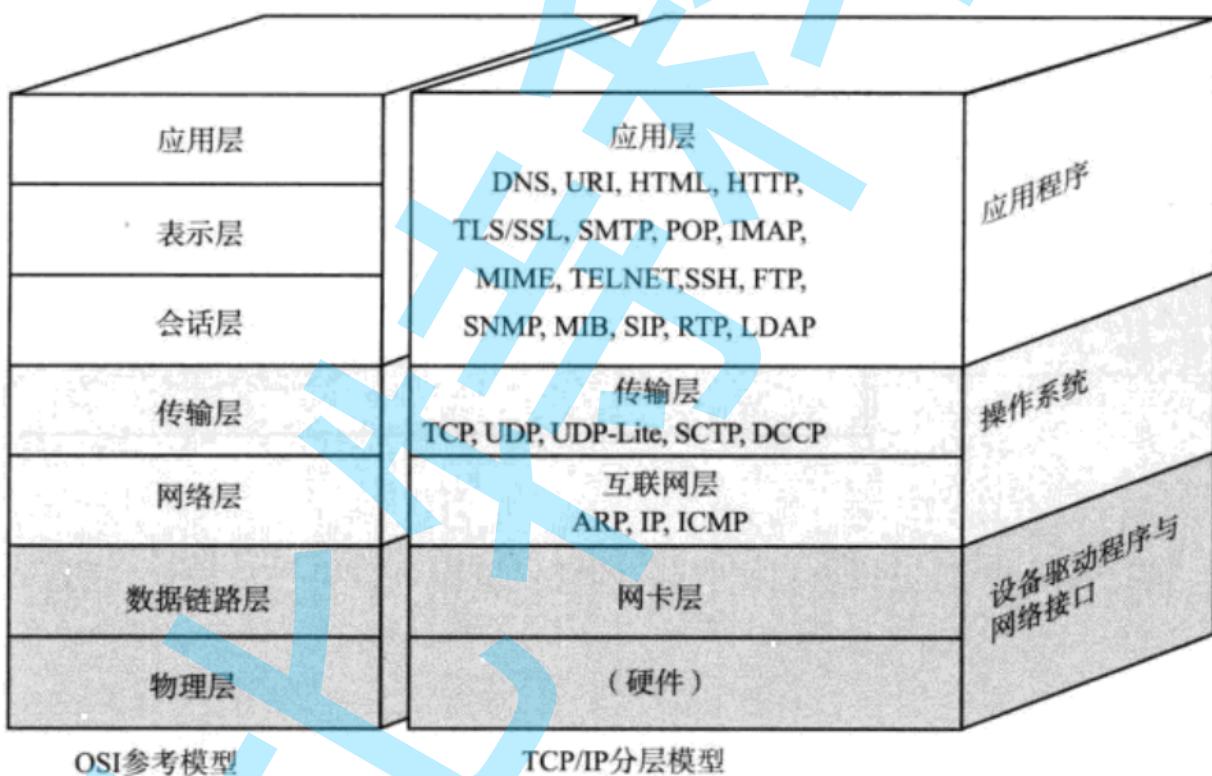
	分层名称	功    能	每层功能概览
7	应用层	针对特定应用的协议。	<p>针对每个应用的协议</p> <p>电子邮件 ↔ 电子邮件协议</p> <p>远程登录 ↔ 远程登录协议</p> <p>文件传输 ↔ 文件传输协议</p>
6	表示层	设备固有数据格式和网络标准数据格式的转换。	<p>接收不同表现形式的信息，如文字流、图像、声音等</p>
5	会话层	通信管理。负责建立和断开通信连接（数据流动的逻辑通路）。 管理传输层以下的分层。	<p>何时建立连接，何时断开连接以及保持多久的连接？</p>
4	传输层	管理两个节点之间的数据传输。负责可靠传输（确保数据被可靠地传递到目标地址）。	<p>是否有数据丢失？</p>
3	网络层	地址管理与路由选择。	<p>经过哪个路由传递到目标地址？</p>
2	数据链路层	互连设备之间传送和识别数据帧。	<p>数据帧与比特流之间的转换</p> <p>分段转发</p>
1	物理层	以“0”、“1”代表电压的高低、灯光的闪灭。 界定连接器和网线的规格。	<p>0101 → ┌┐ ┌┐ ┌┐ → 0101</p> <p>比特流与电子信号之间的切换</p>

## TCP/IP五层(或四层)模型

TCP/IP是一组协议的代名词，它还包括许多协议，组成了TCP/IP协议簇。

TCP/IP通讯协议采用了5层的层级结构，每一层都呼叫它的下一层所提供的网络来完成自己的需求。

- 物理层**: 负责光/电信号的传递方式。比如现在以太网通用的网线(双绞线)、早期以太网采用的同轴电缆(现在主要用于有线电视)、光纤, 现在的wifi无线网使用电磁波等都属于物理层的概念。物理层的能力决定了最大传输速率、传输距离、抗干扰性等。集线器(Hub)工作在物理层。
- 数据链路层**: 负责设备之间的数据帧的传送和识别。例如网卡设备的驱动、帧同步(就是说从网上检测到什么信号算作新帧的开始)、冲突检测(如果检测到冲突就自动重发)、数据差错校验等工作。有以太网、令牌环网, 无线LAN等标准。交换机(Switch)工作在数据链路层。
- 网络层**: 负责地址管理和路由选择。例如在IP协议中, 通过IP地址来标识一台主机, 并通过路由表的方式规划出两台主机之间的数据传输的线路(路由)。路由器(Router)工作在网络层。
- 传输层**: 负责两台主机之间的数据传输。如传输控制协议(TCP), 能够确保数据可靠的从源主机发送到目标主机。
- 应用层**: 负责应用程序间沟通, 如简单电子邮件传输(SMTP)、文件传输协议(FTP)、网络远程访问协议(Telnet)等。我们的网络编程主要就是针对应用层。



物理层我们考虑的比较少。因此很多时候也可以称为TCP/IP四层模型。

一般而言

- 对于一台主机, 它的操作系统内核实现了从传输层到物理层的内容;
- 对于一台路由器, 它实现了从网络层到物理层;
- 对于一台交换机, 它实现了从数据链路层到物理层;
- 对于集线器, 它只实现了物理层;

但是并不绝对. 很多交换机也实现了网络层的转发; 很多路由器也实现了部分传输层的内容(比如端口转发);

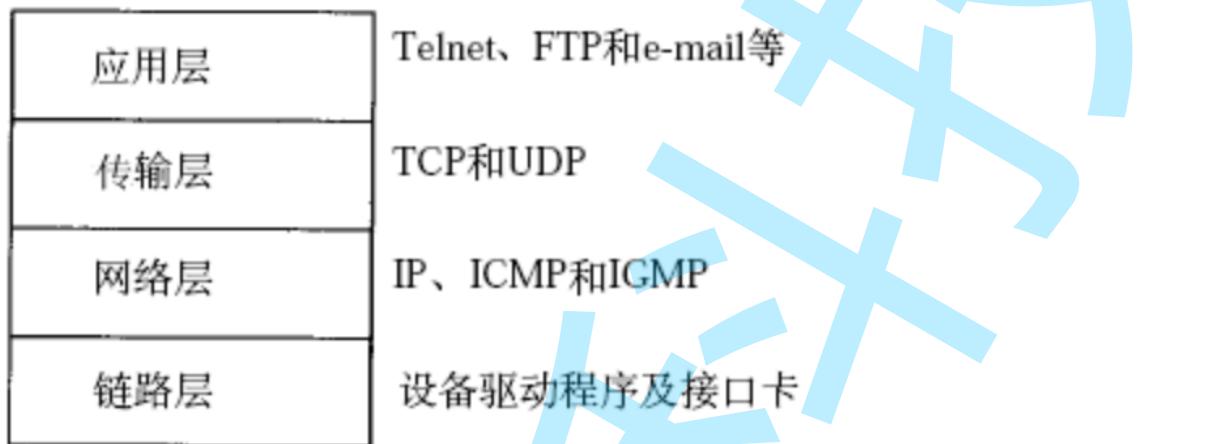
## 参考资料

[TCP/IP四层模型和OSI七层模型的概念](#)

# 网络传输基本流程

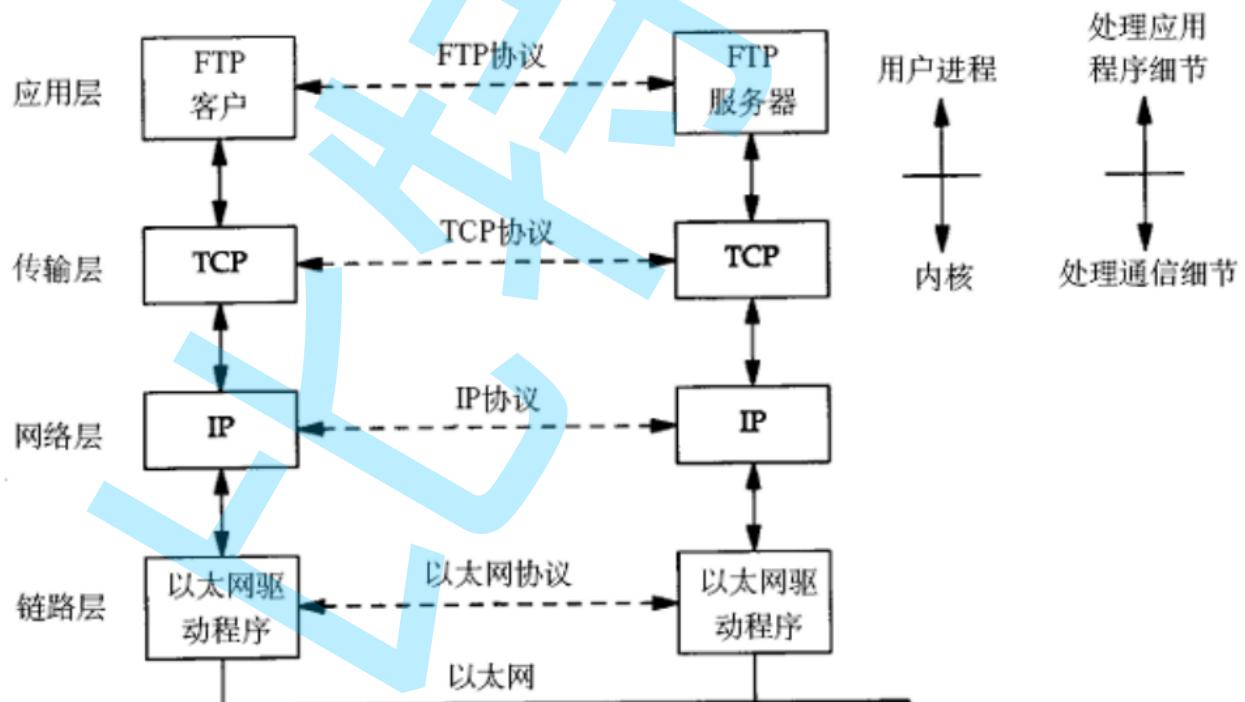
## 网络传输流程图

同一个网段内的两台主机进行文件传输.

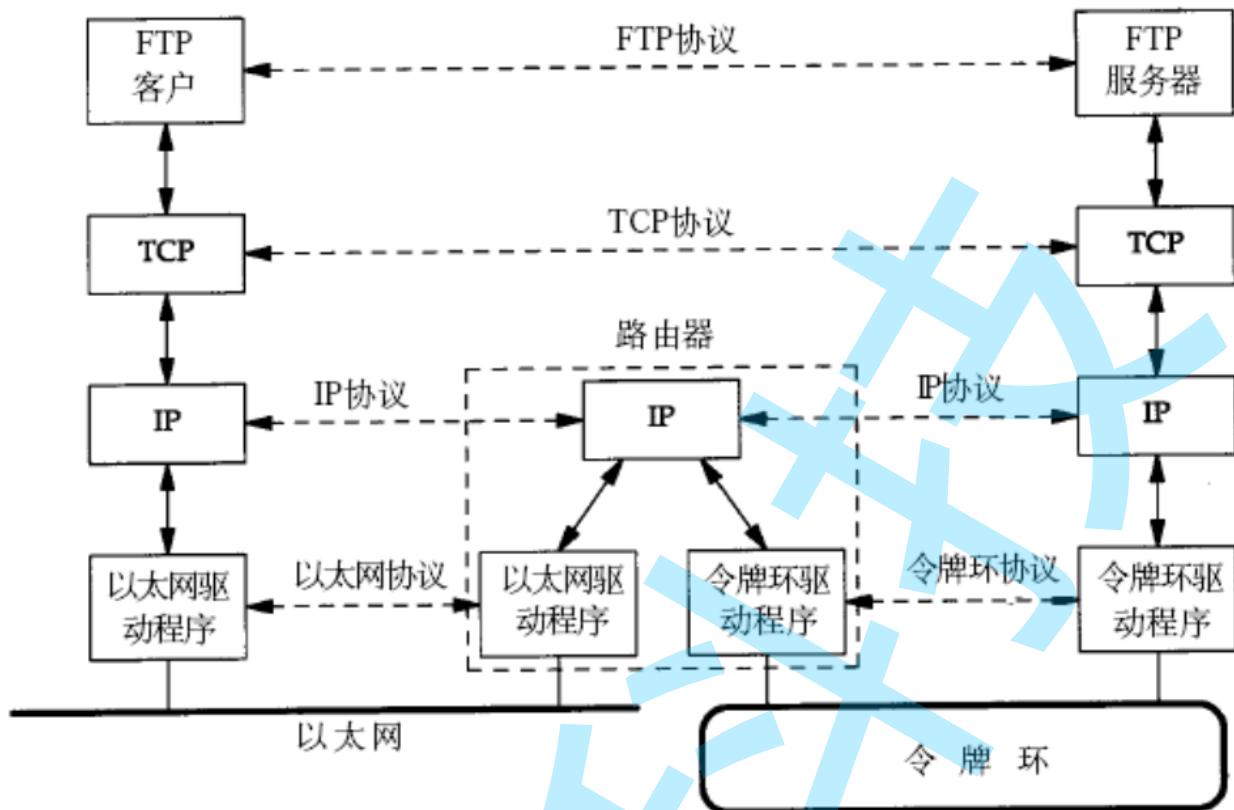


两台计算机通过TCP/IP协议通讯的过程如下所示

TCP/IP通讯过程



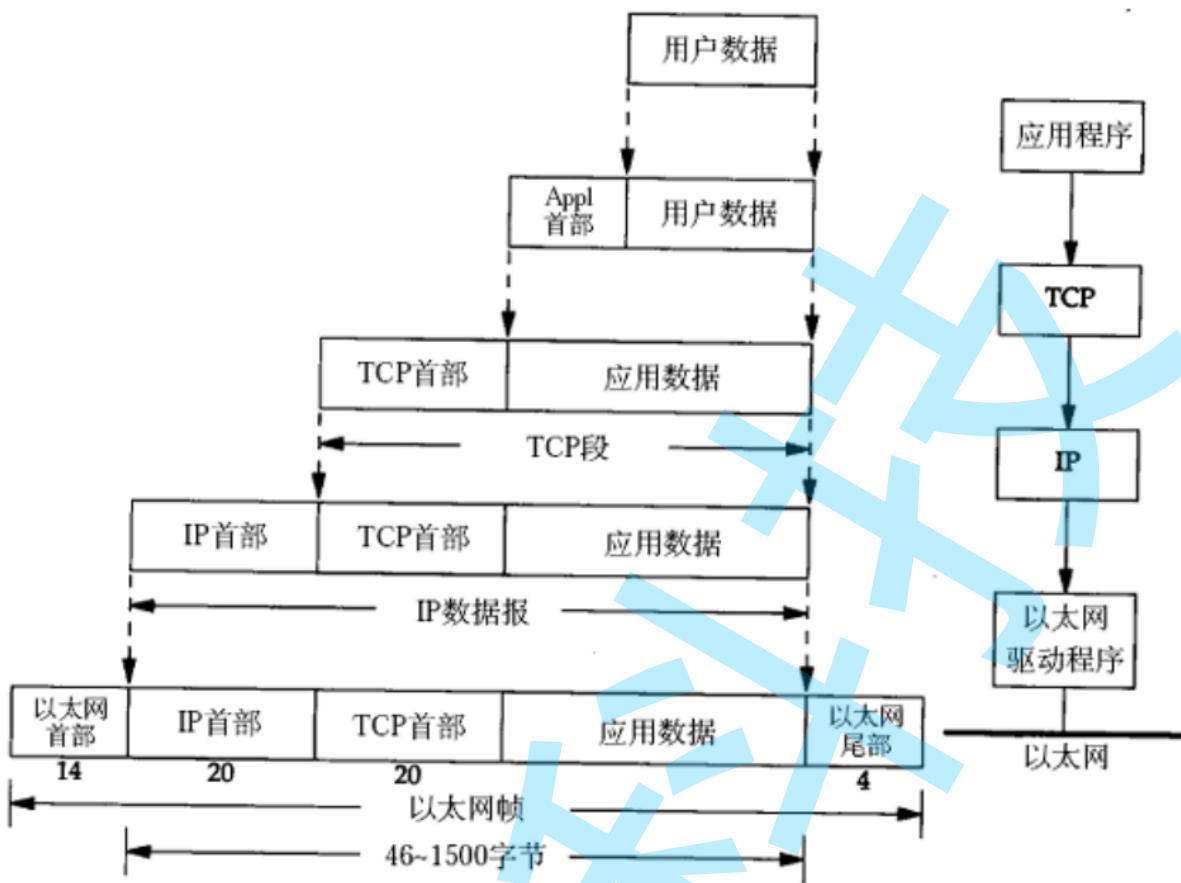
跨网段的主机的文件传输. 数据从一台计算机到另一台计算机传输过程中要经过一个或多个路由器.



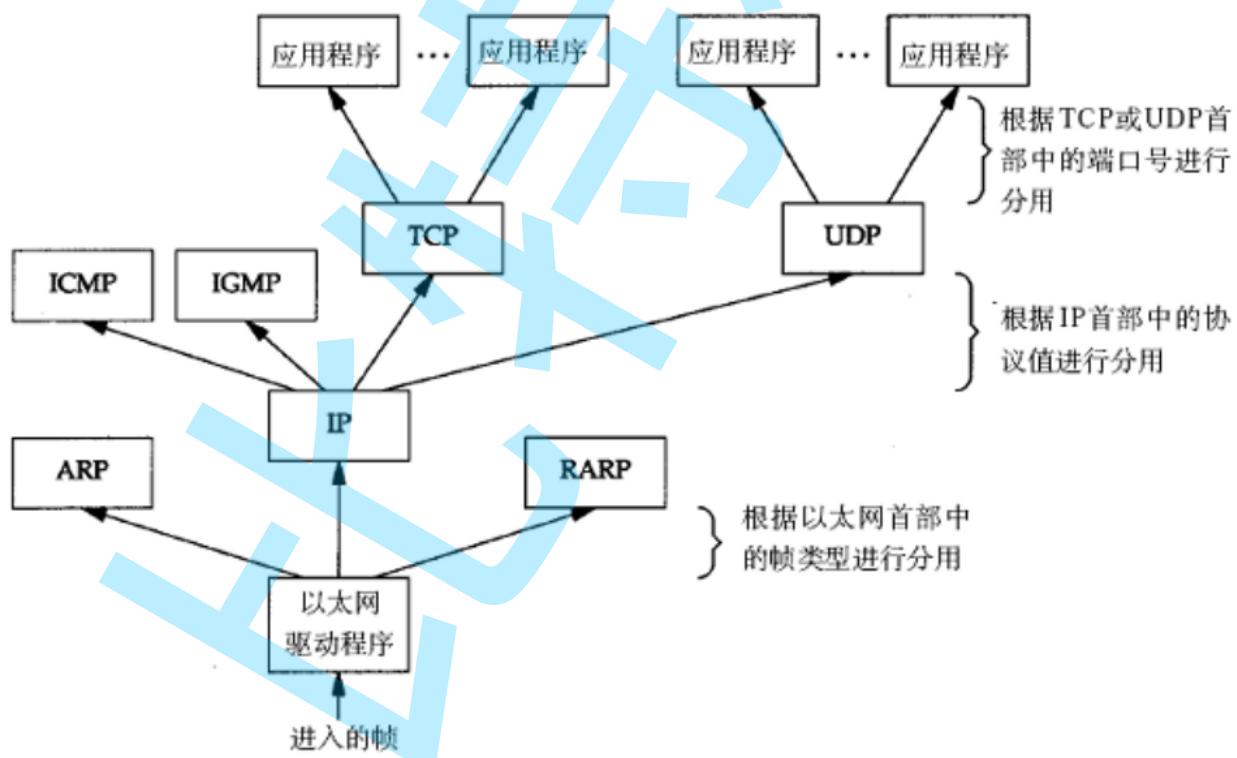
## 数据包封装和分用

- 不同的协议层对数据包有不同的称谓,在传输层叫做段(segment),在网络层叫做数据报(datagram),在链路层叫做帧(frame).
- 应用层数数据通过协议栈发到网络上时,每层协议都要加上一个数据首部(header),称为封装(Encapsulation).
- 首部信息中包含了一些类似于首部有多长,载荷(payload)有多长,上层协议是什么等信息.
- 数据封装成帧后发到传输介质上,到达目的主机后每层协议再剥掉相应的首部,根据首部中的"上层协议"字段将数据交给对应的上层协议处理.

下图为数据封装的过程



下图为数据分用的过程



# 网络中的地址管理

## 认识IP地址

IP协议有两个版本, IPv4和IPv6. 我们整个的课程, 凡是提到IP协议, 没有特殊说明的, 默认都是指IPv4

- IP地址是在IP协议中, 用来标识网络中不同主机的地址;
- 对于IPv4来说, IP地址是一个4字节, 32位的整数;
- 我们通常也使用 "点分十进制" 的字符串表示IP地址, 例如 192.168.0.1 ; 用点分割的每一个数字表示一个字节, 范围是 0 - 255;

## 认识MAC地址

- MAC地址用来识别数据链路层中相连的节点;
- 长度为48位, 及6个字节. 一般用16进制数字加上冒号的形式来表示(例如: 08:00:27:03:fb:19)
- 在网卡出厂时就确定了, 不能修改. mac地址通常是唯一的(虚拟机中的mac地址不是真实的mac地址, 可能会冲突; 也有些网卡支持用户配置mac地址).

# 网络编程套接字

## 本节重点

- 认识IP地址, 端口号, 网络字节序等网络编程中的基本概念;
- 学习socket api的基本用法;
- 能够实现一个简单的udp客户端/服务器;
- 能够实现一个简单的tcp客户端/服务器(单连接版本, 多进程版本, 多线程版本);
- 理解tcp服务器建立连接, 发送数据, 断开连接的流程;

## 预备知识

### 理解源IP地址和目的IP地址

#### 唐僧例子1

在IP数据包头部中, 有两个IP地址, 分别叫做源IP地址, 和目的IP地址.

思考: 我们光有IP地址就可以完成通信了嘛? 想象一下发qq消息的例子, 有了IP地址能够把消息发送到对方的机器上, 但是还需要有一个其他的标识来区分出, 这个数据要给哪个程序进行解析.

### 认识端口号

端口号(port)是传输层协议的内容.

- 端口号是一个2字节16位的整数;
- 端口号用来标识一个进程, 告诉操作系统, 当前的这个数据要交给哪一个进程来处理;
- IP地址 + 端口号能够标识网络上的某一台主机的某一个进程;
- 一个端口号只能被一个进程占用.

### 理解 "端口号" 和 "进程ID"

我们之前在学习系统编程的时候, 学习了 pid 表示唯一一个进程; 此处我们的端口号也是唯一表示一个进程. 那么这两者之间是怎样的关系?

#### 10086例子

另外, 一个进程可以绑定多个端口号; 但是一个端口号不能被多个进程绑定;

### 理解源端口号和目的端口号

#### 唐僧例子2

#### 送快递例子

传输层协议(TCP和UDP)的数据段中有两个端口号, 分别叫做源端口号和目的端口号. 就是在描述 "数据是谁发的, 要发给谁";

## 认识TCP协议

此处我们先对TCP(Transmission Control Protocol 传输控制协议)有一个直观的认识; 后面我们再详细讨论TCP的一些细节问题.

- 传输层协议
- 有连接
- 可靠传输
- 面向字节流

## 认识UDP协议

此处我们也是对UDP(User Datagram Protocol 用户数据报协议)有一个直观的认识; 后面再详细讨论.

- 传输层协议
- 无连接
- 不可靠传输
- 面向数据报

## 网络字节序

我们已经知道,内存中的多字节数据相对于内存地址有大端和小端之分, 磁盘文件中的多字节数据相对于文件中的偏移地址也有大端小端之分, 网络数据流同样有大端小端之分. 那么如何定义网络数据流的地址呢?

- 发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出;
- 接收主机把从网络上接到的字节依次保存在接收缓冲区中, 也是按内存地址从低到高的顺序保存;
- 因此, 网络数据流的地址应这样规定: 先发出的数据是低地址, 后发出的数据是高地址.
- TCP/IP协议规定, 网络数据流应采用大端字节序, 即低地址高字节.
- 不管这台主机是大端机还是小端机, 都会按照这个TCP/IP规定的网络字节序来发送/接收数据;
- 如果当前发送主机是小端, 就需要先将数据转成大端; 否则就忽略, 直接发送即可;

将0x1234abcd写入到以0x0000开始的内存中, 则结果为

	big-endian	little-endian
0x0000	0x12	0xcd
0x0001	0x23	0xab
0x0002	0xab	0x34
0x0003	0xcd	0x12

为使网络程序具有可移植性, 使同样的C代码在大端和小端计算机上编译后都能正常运行, 可以调用以下库函数做网络字节序和主机字节序的转换。

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

- 这些函数名很好记, h表示host,n表示network,l表示32位长整数,s表示16位短整数。
- 例如htonl表示将32位的长整数从主机字节序转换为网络字节序, 例如将IP地址转换后准备发送。
- 如果主机是小端字节序, 这些函数将参数做相应的大小端转换然后返回;
- 如果主机是大端字节序, 这些函数不做转换, 将参数原封不动地返回。

# socket编程接口

## socket 常见API

```
// 创建 socket 文件描述符 (TCP/UDP, 客户端 + 服务器)
int socket(int domain, int type, int protocol);

// 绑定端口号 (TCP/UDP, 服务器)
int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);

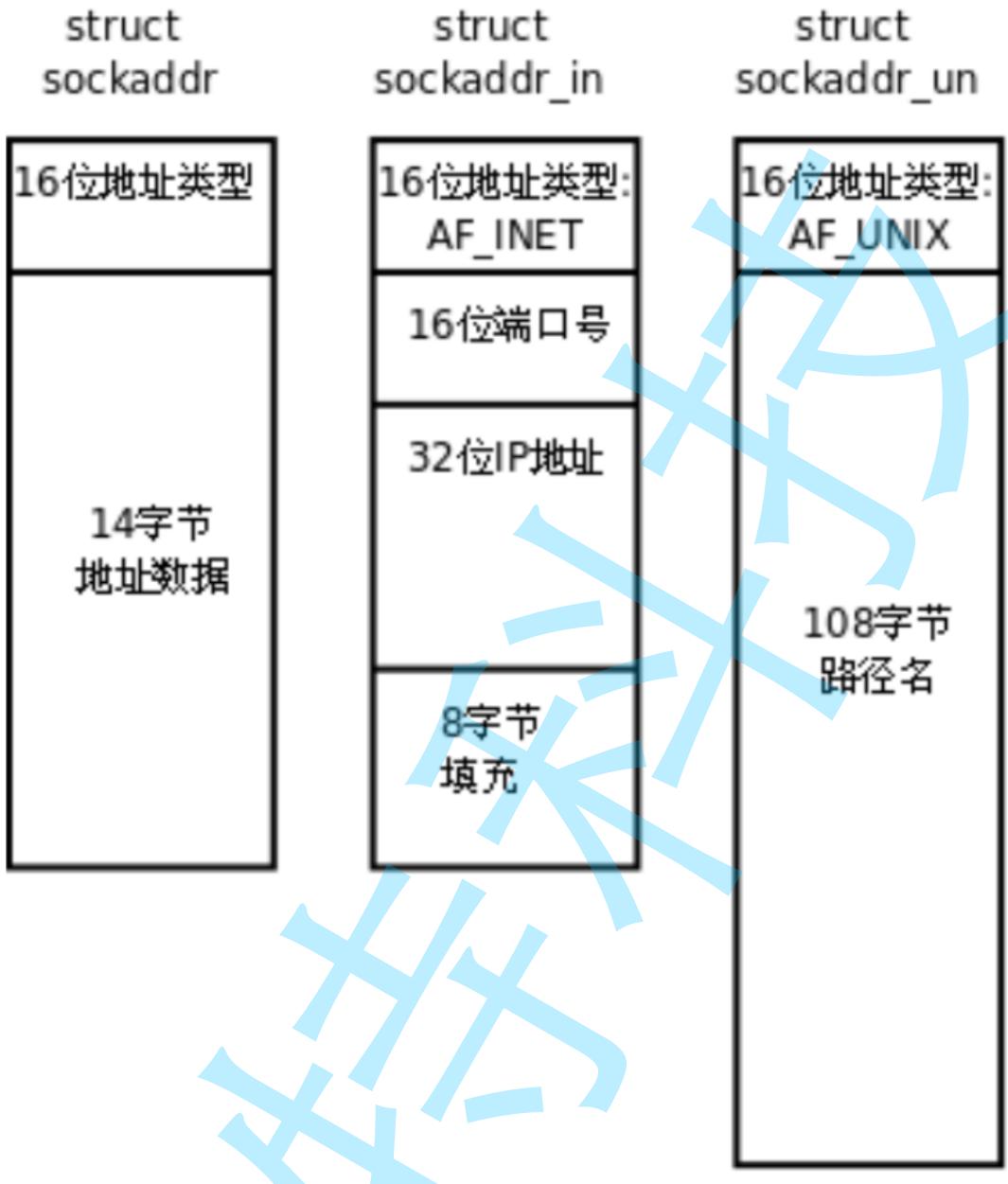
// 开始监听socket (TCP, 服务器)
int listen(int socket, int backlog);

// 接收请求 (TCP, 服务器)
int accept(int socket, struct sockaddr* address,
           socklen_t* address_len);

// 建立连接 (TCP, 客户端)
int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

## sockaddr结构

socket API是一层抽象的网络编程接口,适用于各种底层网络协议,如IPv4、IPv6,以及后面要讲的UNIX Domain Socket. 然而, 各种网络协议的地址格式并不相同.



- IPv4和IPv6的地址格式定义在netinet/in.h中,IPv4地址用sockaddr\_in结构体表示,包括16位地址类型, 16位端口号和32位IP地址.
- IPv4、IPv6地址类型分别定义为常数AF\_INET、AF\_INET6. 这样,只要取得某种sockaddr结构体的首地址, 不需要知道具体是哪种类型的sockaddr结构体, 就可以根据地址类型字段确定结构体中的内容.
- socket API可以都用struct sockaddr \*类型表示, 在使用的时候需要强制转化成sockaddr\_in; 这样的好处是程序的通用性, 可以接收IPv4, IPv6, 以及UNIX Domain Socket各种类型的sockaddr结构体指针做为参数;

### sockaddr 结构

```

148 struct sockaddr
149 {
150     __SOCKADDR_COMMON (sa_); /* Common data: address family and length. */
151     char sa_data[14]; /* Address data. */
152 };

```

## sockaddr\_in 结构

```
237 /* Structure describing an Internet socket address. */
238 struct sockaddr_in
239 {
240     __SOCKADDR_COMMON (sin_);
241     in_port_t sin_port;      /* Port number. */
242     struct in_addr sin_addr; /* Internet address. */
243
244     /* Pad to size of `struct sockaddr'. */
245     unsigned char sin_zero[sizeof (struct sockaddr) -
246         __SOCKADDR_COMMON_SIZE -
247         sizeof (in_port_t) -
248         sizeof (struct in_addr)];
249 };
```

虽然socket api的接口是sockaddr, 但是我们真正在基于IPv4编程时, 使用的数据结构是sockaddr\_in; 这个结构里主要有三部分信息: 地址类型, 端口号, IP地址.

## in\_addr结构

```
30 /* Internet address. */
31 typedef uint32_t in_addr_t;
32 struct in_addr
33 {
34     in_addr_t s_addr;
35 };
36
```

in\_addr用来表示一个IPv4的IP地址. 其实就是一个32位的整数;

## 简单的UDP网络程序

实现一个简单的英译汉的功能

备注: 代码中会用到 地址转换函数 . 参考接下来的章节.

## 封装 UdpSocket

udp\_socket.hpp

```
#pragma once
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cassert>
#include <string>

#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
typedef struct sockaddr sockaddr;
typedef struct sockaddr_in sockaddr_in;

class UdpSocket {
public:
```

```
}

bool Socket() {
    fd_ = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd_ < 0) {
        perror("socket");
        return false;
    }
    return true;
}

bool Close() {
    close(fd_);
    return true;
}

bool Bind(const std::string& ip, uint16_t port) {
    sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(ip.c_str());
    addr.sin_port = htons(port);
    int ret = bind(fd_, (sockaddr*)&addr, sizeof(addr));
    if (ret < 0) {
        perror("bind");
        return false;
    }
    return true;
}

bool RecvFrom(std::string* buf, std::string* ip = NULL, uint16_t* port = NULL) {
    char tmp[1024 * 10] = {0};
    sockaddr_in peer;
    socklen_t len = sizeof(peer);
    ssize_t read_size = recvfrom(fd_, tmp,
        sizeof(tmp) - 1, 0, (sockaddr*)&peer, &len);
    if (read_size < 0) {
        perror("recvfrom");
        return false;
    }
    // 将读到的缓冲区内容放到输出参数中
    buf->assign(tmp, read_size);
    if (ip != NULL) {
        *ip = inet_ntoa(peer.sin_addr);
    }
    if (port != NULL) {
        *port = ntohs(peer.sin_port);
    }
    return true;
}

bool SendTo(const std::string& buf, const std::string& ip, uint16_t port) {
    sockaddr_in addr;
```

```
addr.sin_addr.s_addr = inet_addr(ip.c_str());
addr.sin_port = htons(port);
ssize_t write_size = sendto(fd_, buf.data(), buf.size(), 0, (sockaddr*)&addr, sizeof(addr));
if (write_size < 0) {
    perror("sendto");
    return false;
}
return true;
}

private:
    int fd_;
};
```

## UDP通用服务器

udp\_server.hpp

```
#pragma once
#include "udp_socket.hpp"

// C 式写法
// typedef void (*Handler)(const std::string& req, std::string* resp);
// C++ 11 式写法，能够兼容函数指针，仿函数，和 lambda
#include <functional>
typedef std::function<void (const std::string&, std::string*)> Handler;

class UdpServer {
public:
    UdpServer() {
        assert(sock_.Socket());
    }

    ~UdpServer() {
        sock_.Close();
    }

    bool Start(const std::string& ip, uint16_t port, Handler handler) {
        // 1. 创建 socket
        // 2. 绑定端口号
        bool ret = sock_.Bind(ip, port);
        if (!ret) {
            return false;
        }
        // 3. 进入事件循环
        for (;;) {
            // 4. 尝试读取请求
            std::string req;
            std::string remote_ip;
            uint16_t remote_port = 0;
            bool ret = sock_.RecvFrom(&req, &remote_ip, &remote_port);
            if (!ret) {
```

```

        continue;
    }
    std::string resp;
    // 5. 根据请求计算响应
    handler(req, &resp);
    // 6. 返回响应给客户端
    sock_.SendTo(resp, remote_ip, remote_port);
    printf("[%s:%d] req: %s, resp: %s\n", remote_ip.c_str(), remote_port,
           req.c_str(), resp.c_str());
}
sock_.Close();
return true;
}

private:
    UdpSocket sock_;
};
```

## 实现英译汉服务器

以上代码是对 udp 服务器进行通用接口的封装. 基于以上封装, 实现一个查字典的服务器就很容易了.

dict\_server.cc

```

#include "udp_server.hpp"
#include <unordered_map>
#include <iostream>

std::unordered_map<std::string, std::string> g_dict;

void Translate(const std::string& req, std::string* resp) {
    auto it = g_dict.find(req);
    if (it == g_dict.end()) {
        *resp = "未查到!";
        return;
    }
    *resp = it->second;
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Usage ./dict_server [ip] [port]\n");
        return 1;
    }
    // 1. 数据初始化
    g_dict.insert(std::make_pair("hello", "你好"));
    g_dict.insert(std::make_pair("world", "世界"));
    g_dict.insert(std::make_pair("c++", "最好的编程语言"));
    g_dict.insert(std::make_pair("bit", "特别NB"));
    // 2. 启动服务器
    UdpServer server;

    server.Start(argv[1], atoi(argv[2]), Translate);
```

```
    return 0;  
}
```

## UDP通用客户端

udp\_client.hpp

```
#pragma once  
#include "udp_socket.hpp"  
  
class UdpClient {  
public:  
    UdpClient(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {  
        assert(sock_.Socket());  
    }  
  
    ~UdpClient() {  
        sock_.Close();  
    }  
  
    bool RecvFrom(std::string* buf) {  
        return sock_.RecvFrom(buf);  
    }  
  
    bool SendTo(const std::string& buf) {  
        return sock_.SendTo(buf, ip_, port_);  
    }  
private:  
    UdpSocket sock_;  
    // 服务器端的 IP 和 端口号  
    std::string ip_;  
    uint16_t port_;  
};
```

## 实现英译汉客户端

```
#include "udp_client.hpp"  
#include <iostream>  
  
int main(int argc, char* argv[]) {  
    if (argc != 3) {  
        printf("Usage ./dict_client [ip] [port]\n");  
        return 1;  
    }  
    UdpClient client(argv[1], atoi(argv[2]));  
    for (;;) {  
        std::string word;  
        std::cout << "请输入您要查的单词: ";  
        std::cin >> word;  
        if (!std::cin) {  
            std::cout << "Good Bye" << std::endl;
```

```

        break;
    }
    client.SendTo(word);
    std::string result;
    client.RecvFrom(&result);
    std::cout << word << " 意思是 " << result << std::endl;
}
return 0;
}

```

## 地址转换函数

本节只介绍基于IPv4的socket网络编程,sockaddr\_in中的成员struct in\_addr sin\_addr表示32位的IP地址但是我们通常用点分十进制的字符串表示IP地址,以下函数可以在字符串表示和in\_addr表示之间转换;

字符串转in\_addr的函数:

```

#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);
in_addr_t inet_addr(const char *strptr);
int inet_pton(int family, const char *strptr, void *addrptr);

```

in\_addr转字符串的函数:

```

char *inet_ntoa(struct in_addr inaddr);
const char *inet_ntop(int family, const void *addrptr, char *strptr,
size_t len);

```

其中inet\_pton和inet\_ntop不仅可以转换IPv4的in\_addr,还可以转换IPv6的in6\_addr,因此函数接口是void \*addrptr。

代码示例:

```

1 #include <stdio.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5
6 int main() {
7     struct sockaddr_in addr;
8     inet_aton("127.0.0.1", &addr.sin_addr);
9     uint32_t* ptr = (uint32_t*)(&addr.sin_addr);
10    printf("addr: %x\n", *ptr);
11    printf("addr_str: %s\n", inet_ntoa(addr.sin_addr));
12    return 0;
13 }

```

## 关于inet\_ntoa

inet\_ntoa这个函数返回了一个char\*,很显然是这个函数自己在内部为我们申请了一块内存来保存ip的结果.那么是否需要调用者手动释放呢?

```
The inet_ntoa() function converts the Internet host address in, given in network byte order, to a string in IPv4 dotted-decimal notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.
```

man手册上说, inet\_ntoa函数, 是把这个返回结果放到了静态存储区. 这个时候不需要我们手动进行释放.

那么问题来了, 如果我们调用多次这个函数, 会有什么样的效果呢? 参见如下代码:

```
1 #include <stdio.h>
2 #include <netinet/in.h>
3 #include <arpa/inet.h>
4
5 int main() {
6     struct sockaddr_in addr1;
7     struct sockaddr_in addr2;
8     addr1.sin_addr.s_addr = 0;
9     addr2.sin_addr.s_addr = 0xffffffff;
10    char* ptr1 = inet_ntoa(addr1.sin_addr);
11    char* ptr2 = inet_ntoa(addr2.sin_addr);
12    printf("ptr1: %s, ptr2: %s\n", ptr1, ptr2);
13    return 0;
14 }
```

运行结果如下:

```
[tangzhong@tz addr_convert]$ ./a.out
ptr1: 255.255.255.255, ptr2: 255.255.255.255
```

因为inet\_ntoa把结果放到自己内部的一个静态存储区, 这样第二次调用时的结果会覆盖掉上一次的结果.

- 思考: 如果有多个线程调用 inet\_ntoa, 是否会出现异常情况呢?
- 在APUE中, 明确提出inet\_ntoa不是线程安全的函数;
- 但是在centos7上测试, 并没有出现问题, 可能内部的实现加了互斥锁;
- 同学们课后自己写程序验证一下在自己的机器上inet\_ntoa是否会出现多线程的问题;
- 在多线程环境下, 推荐使用inet\_ntop, 这个函数由调用者提供一个缓冲区保存结果, 可以规避线程安全问题;

多线程调用inet\_ntoa代码示例如下(同学们课后自己测试):

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>

void* Func1(void* p) {
    struct sockaddr_in* addr = (struct sockaddr_in*)p;
    while (1) {
        char* ptr = inet_ntoa(addr->sin_addr);
        printf("addr1: %s\n", ptr);
    }
    return NULL;
}

void* Func2(void* p) {
    struct sockaddr_in* addr = (struct sockaddr_in*)p;
    while (1) {

        char* ptr = inet_ntoa(addr->sin_addr);
```

```

    printf("addr2: %s\n", ptr);
}
return NULL;
}

int main() {
pthread_t tid1 = 0;
struct sockaddr_in addr1;
struct sockaddr_in addr2;
addr1.sin_addr.s_addr = 0;
addr2.sin_addr.s_addr = 0xffffffff;
pthread_create(&tid1, NULL, Func1, &addr1);
pthread_t tid2 = 0;
pthread_create(&tid2, NULL, Func2, &addr2);
pthread_join(tid1, NULL);
pthread_join(tid2, NULL);
return 0;
}

```

## 简单的TCP网络程序

和刚才UDP类似. 实现一个简单的英译汉的功能

### TCP socket API 详解

下面介绍程序中用到的socket API,这些函数都在sys/socket.h中。

**socket():**

```

NAME      socket - create an endpoint for communication
SYNOPSIS
  #include <sys/types.h>
  #include <sys/socket.h>          /* See NOTES */
  int socket(int domain, int type, int protocol);

```

- socket()打开一个网络通讯端口,如果成功的话,就像open()一样返回一个文件描述符;
- 应用程序可以像读写文件一样用read/write在网络上收发数据;
- 如果socket()调用出错则返回-1;
- 对于IPv4, family参数指定为AF\_INET;
- 对于TCP协议,type参数指定为SOCK\_STREAM, 表示面向流的传输协议
- protocol参数的介绍从略,指定为0即可。

**bind():**

```
NAME
    bind - bind a name to a socket

SYNOPSIS
    #include <sys/types.h>           /* See NOTES */
    #include <sys/socket.h>

    int bind(int sockfd, const struct sockaddr *addr,
             socklen_t addrlen);
```

- 服务器程序所监听的网络地址和端口号通常是固定不变的,客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接; 服务器需要调用bind绑定一个固定的网络地址和端口号;
- bind()成功返回0,失败返回-1。
- bind()的作用是将参数sockfd和myaddr绑定在一起,使sockfd这个用于网络通讯的文件描述符监听myaddr所描述的地址和端口号;
- 前面讲过,struct sockaddr \*是一个通用指针类型,myaddr参数实际上可以接受多种协议的sockaddr结构体,而它们的长度各不相同,所以需要第三个参数addrlen指定结构体的长度;

我们的程序中对myaddr参数是这样初始化的:

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
```

1. 将整个结构体清零;
2. 设置地址类型为AF\_INET;
3. 网络地址为INADDR\_ANY, 这个宏表示本地的任意IP地址,因为服务器可能有多个网卡,每个网卡也可能绑定多个IP地址,这样设置可以在所有的IP地址上监听,直到与某个客户端建立了连接时才确定下来到底用哪个IP地址;
4. 端口号为SERV\_PORT, 我们定义为9999;

listen():

```
NAME
    listen - listen for connections on a socket

SYNOPSIS
    #include <sys/types.h>           /* See NOTES */
    #include <sys/socket.h>

    int listen(int sockfd, int backlog);
```

- listen()声明sockfd处于监听状态, 并且最多允许有backlog个客户端处于连接等待状态, 如果接收到更多的连接请求就忽略, 这里设置不会太大(一般是5), 具体细节同学们课后深入研究;
- listen()成功返回0,失败返回-1;

accept():

```
NAME
    accept - accept a connection on a socket

SYNOPSIS
    #include <sys/types.h>           /* See NOTES */
    #include <sys/socket.h>

    int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- 三次握手完成后, 服务器调用accept()接受连接;
- 如果服务器调用accept()时还没有客户端的连接请求, 就阻塞等待直到有客户端连接上来;
- addr是一个传出参数, accept()返回时传出客户端的地址和端口号;
- 如果给addr参数传NULL, 表示不关心客户端的地址;
- addrlen参数是一个传入传出参数(value-result argument), 传入的是调用者提供的, 缓冲区addr的长度以避免缓冲区溢出问题, 传出的是客户端地址结构体的实际长度(有可能没有占满调用者提供的缓冲区);

我们的服务器程序结构是这样的:

```
while (1) {
    cliaddr_len = sizeof(cliaddr);
    connfd = accept(listenfd,
                    (struct sockaddr *)&cliaddr, &cliaddr_len);
    n = read(connfd, buf, MAXLINE);
    ...
    close(connfd);
}
```

**理解accecp的返回值: 饭店拉客例子**

connect

```
NAME      connect - initiate a connection on a socket
SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
/* See NOTES */
int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

- 客户端需要调用connect()连接服务器;
- connect和bind的参数形式一致, 区别在于bind的参数是自己的地址, 而connect的参数是对方的地址;
- connect()成功返回0, 出错返回-1;

## 封装 TCP socket

tcp\_socket.hpp

```
#pragma once
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <string>
#include <cassert>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
typedef struct sockaddr sockaddr;
typedef struct sockaddr_in sockaddr_in;

#define CHECK_RET(exp) if (!(exp)) { \
    return false; \
}
```

```
class TcpSocket {
public:
    TcpSocket() : fd_(-1) { }
    TcpSocket(int fd) : fd_(fd) { }

    bool Socket() {
        fd_ = socket(AF_INET, SOCK_STREAM, 0);
        if (fd_ < 0) {
            perror("socket");
            return false;
        }
        printf("open fd = %d\n", fd_);
        return true;
    }

    bool Close() const {
        close(fd_);
        printf("close fd = %d\n", fd_);
        return true;
    }

    bool Bind(const std::string& ip, uint16_t port) const {
        sockaddr_in addr;
        addr.sin_family = AF_INET;
        addr.sin_addr.s_addr = inet_addr(ip.c_str());
        addr.sin_port = htons(port);
        int ret = bind(fd_, (sockaddr*)&addr, sizeof(addr));
        if (ret < 0) {
            perror("bind");
            return false;
        }
        return true;
    }

    bool Listen(int num) const {
        int ret = listen(fd_, num);
        if (ret < 0) {
            perror("listen");
            return false;
        }
        return true;
    }

    bool Accept(TcpSocket* peer, std::string* ip = NULL, uint16_t* port = NULL) const {
        sockaddr_in peer_addr;
        socklen_t len = sizeof(peer_addr);
        int new_sock = accept(fd_, (sockaddr*)&peer_addr, &len);
        if (new_sock < 0) {
            perror("accept");
            return false;
        }
        printf("accept fd = %d\n", new_sock);
    }
}
```

```
peer->fd_ = new_sock;
if (ip != NULL) {
    *ip = inet_ntoa(peer_addr.sin_addr);
}
if (port != NULL) {
    *port = ntohs(peer_addr.sin_port);
}
return true;
}

bool Recv(std::string* buf) const {
buf->clear();
char tmp[1024 * 10] = {0};
// [注意!] 这里的读并不算很严谨，因为一次 recv 并不能保证把所有的数据都全部读完
// 参考 man 手册 MSG_WAITALL 节。
ssize_t read_size = recv(fd_, tmp, sizeof(tmp), 0);
if (read_size < 0) {
    perror("recv");
    return false;
}
if (read_size == 0) {
    return false;
}
buf->assign(tmp, read_size);
return true;
}

bool Send(const std::string& buf) const {
ssize_t write_size = send(fd_, buf.data(), buf.size(), 0);
if (write_size < 0) {
    perror("send");
    return false;
}
return true;
}

bool Connect(const std::string& ip, uint16_t port) const {
sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr(ip.c_str());
addr.sin_port = htons(port);
int ret = connect(fd_, (sockaddr*)&addr, sizeof(addr));
if (ret < 0) {
    perror("connect");
    return false;
}
return true;
}

int GetFd() const {
return fd_;
}
```

```
private:  
    int fd_;  
};
```

## TCP通用服务器

tcp\_server.hpp

```
#pragma once  
#include <functional>  
#include "tcp_socket.hpp"  
  
typedef std::function<void (const std::string& req, std::string* resp)> Handler;  
  
class TcpServer {  
public:  
    TcpServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {}  
  
    bool Start(Handler handler) {  
        // 1. 创建 socket;  
        CHECK_RET(listen_sock_.Socket());  
        // 2. 绑定端口号  
        CHECK_RET(listen_sock_.Bind(ip_, port_));  
        // 3. 进行监听  
        CHECK_RET(listen_sock_.Listen(5));  
        // 4. 进入事件循环  
        for (;;) {  
            // 5. 进行 accept  
            TcpSocket new_sock;  
            std::string ip;  
            uint16_t port = 0;  
            if (!listen_sock_.Accept(&new_sock, &ip, &port)) {  
                continue;  
            }  
            printf("[client %s:%d] connect!\n", ip.c_str(), port);  
            // 6. 进行循环读写  
            for (;;) {  
                std::string req;  
                // 7. 读取请求. 读取失败则结束循环  
                bool ret = new_sock.Recv(&req);  
                if (!ret) {  
                    printf("[client %s:%d] disconnect!\n", ip.c_str(), port);  
                    // [注意!] 需要关闭 socket  
                    new_sock.Close();  
                    break;  
                }  
                // 8. 计算响应  
                std::string resp;  
                handler(req, &resp);  
                // 9. 写回响应  
                new_sock.Send(resp);  
            }  
        }  
    }  
};
```

```

        printf("[%s:%d] req: %s, resp: %s\n", ip.c_str(), port,
               req.c_str(), resp.c_str());
    }
}
return true;
}

private:
    TcpSocket listen_sock_;
    std::string ip_;
    uint64_t port_;
};

```

## 英译汉服务器

```

#include <unordered_map>
#include "tcp_server.hpp"

std::unordered_map<std::string, std::string> g_dict;

void Translate(const std::string& req, std::string* resp) {
    auto it = g_dict.find(req);
    if (it == g_dict.end()) {
        *resp = "未找到";
        return;
    }
    *resp = it->second;
    return;
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Usage ./dict_server [ip] [port]\n");
        return 1;
    }
    // 1. 初始化词典
    g_dict.insert(std::make_pair("hello", "你好"));
    g_dict.insert(std::make_pair("world", "世界"));
    g_dict.insert(std::make_pair("bit", "贼NB"));
    // 2. 启动服务器
    TcpServer server(argv[1], atoi(argv[2]));
    server.Start(Translate);
    return 0;
}

```

## TCP通用客户端

tcp\_client.hpp

```
#pragma once
#include "tcp_socket.hpp"

class TcpClient {
public:
    TcpClient(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {
        // [注意!!] 需要先创建好 socket
        sock_.Socket();
    }

    ~TcpClient() {
        sock_.Close();
    }

    bool Connect() {
        return sock_.Connect(ip_, port_);
    }

    bool Recv(std::string* buf) {
        return sock_.Recv(buf);
    }

    bool Send(const std::string& buf) {
        return sock_.Send(buf);
    }

private:
    TcpSocket sock_;
    std::string ip_;
    uint16_t port_;
};
```

## 英译汉客户端

dict\_client.cc

```
#include "tcp_client.hpp"
#include <iostream>

int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Usage ./dict_client [ip] [port]\n");
        return 1;
    }
    TcpClient client(argv[1], atoi(argv[2]));
    bool ret = client.Connect();
    if (!ret) {
        return 1;
    }
    for (;;) {
```

```

    std::cout << "请输入要查询的单词:" << std::endl;
    std::string word;
    std::cin >> word;
    if (!std::cin) {
        break;
    }
    client.Send(word);
    std::string result;
    client.Recv(&result);
    std::cout << result << std::endl;
}
return 0;
}

```

由于客户端不需要固定的端口号,因此不必调用bind(),客户端的端口号由内核自动分配.

#### 注意:

- 客户端不是不允许调用bind(), 只是没有必要调用bind()固定一个端口号. 否则如果在同一台机器上启动多个客户端, 就会出现端口号被占用导致不能正确建立连接;
- 服务器也不是必须调用bind(), 但如果服务器不调用bind(), 内核会自动给服务器分配监听端口, 每次启动服务器时端口号都不一样, 客户端要连接服务器就会遇到麻烦;

## 测试多个连接的情况

再启动一个客户端, 尝试连接服务器, 发现第二个客户端, 不能正确的和服务器进行通信.

分析原因, 是因为我们accept了一个请求之后, 就在一直while循环尝试read, 没有继续调用到accept, 导致不能接受新的请求.

我们当前的这个TCP, 只能处理一个连接, 这是不科学的.

## 简单的TCP网络程序(多进程版本)

通过每个请求, 创建子进程的方式来支持多连接;

tcp\_process\_server.hpp

```

#pragma once
#include <functional>
#include <signal.h>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string& req, std::string* resp)> Handler;

// 多进程版本的 Tcp 服务器
class TcpProcessServer {
public:
    TcpProcessServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {
        // 需要处理子进程
        signal(SIGCHLD, SIG_IGN);
    }

    void ProcessConnect(const TcpSocket& new_sock, const std::string& ip, uint16_t port,

```

```
        Handler handler) {  
    int ret = fork();  
    if (ret > 0) {  
        // father  
        // 父进程不需要做额外的操作，直接返回即可。  
        // 思考，这里能否使用 wait 进行进程等待?  
        // 如果使用 wait，会导致父进程不能快速再次调用到 accept，仍然没法处理多个请求  
        // [注意！！] 父进程需要关闭 new_sock  
        new_sock.Close();  
        return;  
    } else if (ret == 0) {  
        // child  
        // 处理具体的连接过程。每个连接一个子进程  
        for (;;) {  
            std::string req;  
            bool ret = new_sock.Recv(&req);  
            if (!ret) {  
                // 当前的请求处理完了，可以退出子进程了。注意，socket 的关闭在析构函数中就完成了  
                printf("[client %s:%d] disconnected!\n", ip.c_str(), port);  
                exit(0);  
            }  
            std::string resp;  
            handler(req, &resp);  
            new_sock.Send(resp);  
            printf("[client %s:%d] req: %s, resp: %s\n", ip.c_str(), port,  
                   req.c_str(), resp.c_str());  
        }  
    } else {  
        perror("fork");  
    }  
}  
  
bool Start(Handler handler) {  
    // 1. 创建 socket;  
    CHECK_RET(listen_sock_.Socket());  
    // 2. 绑定端口号  
    CHECK_RET(listen_sock_.Bind(ip_, port_));  
    // 3. 进行监听  
    CHECK_RET(listen_sock_.Listen(5));  
    // 4. 进入事件循环  
    for (;;) {  
        // 5. 进行 accept  
        TcpSocket new_sock;  
        std::string ip;  
        uint16_t port = 0;  
        if (!listen_sock_.Accept(&new_sock, &ip, &port)) {  
            continue;  
        }  
        printf("[client %s:%d] connect!\n", ip.c_str(), port);  
        ProcessConnect(new_sock, ip, port, handler);  
    }  
    return true;  
}
```

```
private:  
    TcpSocket listen_sock_;  
    std::string ip_;  
    uint64_t port_;  
};
```

dict\_server.cc 稍加修改

将 TcpServer 类改成 TcpProcessServer 类即可

## 简单的TCP网络程序(多线程版本)

通过每个请求, 创建一个线程的方式来支持多连接;

tcp\_thread\_server.hpp

```
#pragma once  
#include <functional>  
#include <pthread.h>  
#include "tcp_socket.hpp"  
  
typedef std::function<void (const std::string&, std::string*)> Handler;  
  
struct ThreadArg {  
    TcpSocket new_sock;  
    std::string ip;  
    uint16_t port;  
    Handler handler;  
};  
  
class TcpThreadServer {  
public:  
    TcpThreadServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {}  
  
    bool Start(Handler handler) {  
        // 1. 创建 socket;  
        CHECK_RET(listen_sock_.Socket());  
        // 2. 绑定端口号  
        CHECK_RET(listen_sock_.Bind(ip_, port_));  
        // 3. 进行监听  
        CHECK_RET(listen_sock_.Listen(5));  
        // 4. 进入循环  
        for (;;) {  
            // 5. 进行 accept  
            ThreadArg* arg = new ThreadArg();  
            arg->handler = handler;  
            bool ret = listen_sock_.Accept(&arg->new_sock, &arg->ip, &arg->port);  
            if (!ret) {  
                continue;  
            }  
            // 处理连接逻辑  
            arg->handler(arg->ip, arg->port);  
        }  
    }  
};
```

```

    }
    printf("[client %s:%d] connect\n", arg->ip.c_str(), arg->port);
    // 6. 创建新的线程完成具体操作
    pthread_t tid;
    pthread_create(&tid, NULL, ThreadEntry, arg);
    pthread_detach(tid);
}
return true;
}

// 这里的成员函数为啥非得是 static?
static void* ThreadEntry(void* arg) {
    // C++ 的四种类型转换都是什么?
    ThreadArg* p = reinterpret_cast<ThreadArg*>(arg);
    ProcessConnect(p);
    // 一定要记得释放内存!!! 也要记得关闭文件描述符
    p->new_sock.Close();
    delete p;
    return NULL;
}

// 处理单次连接. 这个函数也得是 static
static void ProcessConnect(ThreadArg* arg) {
    // 1. 循环进行读写
    for (;;) {
        std::string req;
        // 2. 读取请求
        bool ret = arg->new_sock.Recv(&req);
        if (!ret) {
            printf("[client %s:%d] disconnected!\n", arg->ip.c_str(), arg->port);
            break;
        }
        std::string resp;
        // 3. 根据请求计算响应
        arg->handler(req, &resp);
        // 4. 发送响应
        arg->new_sock.Send(resp);
        printf("[client %s:%d] req: %s, resp: %s\n",
               arg->ip.c_str(), arg->port, req.c_str(), resp.c_str());
    }
}
private:
    TcpSocket listen_sock_;
    std::string ip_;
    uint16_t port_;
};

```



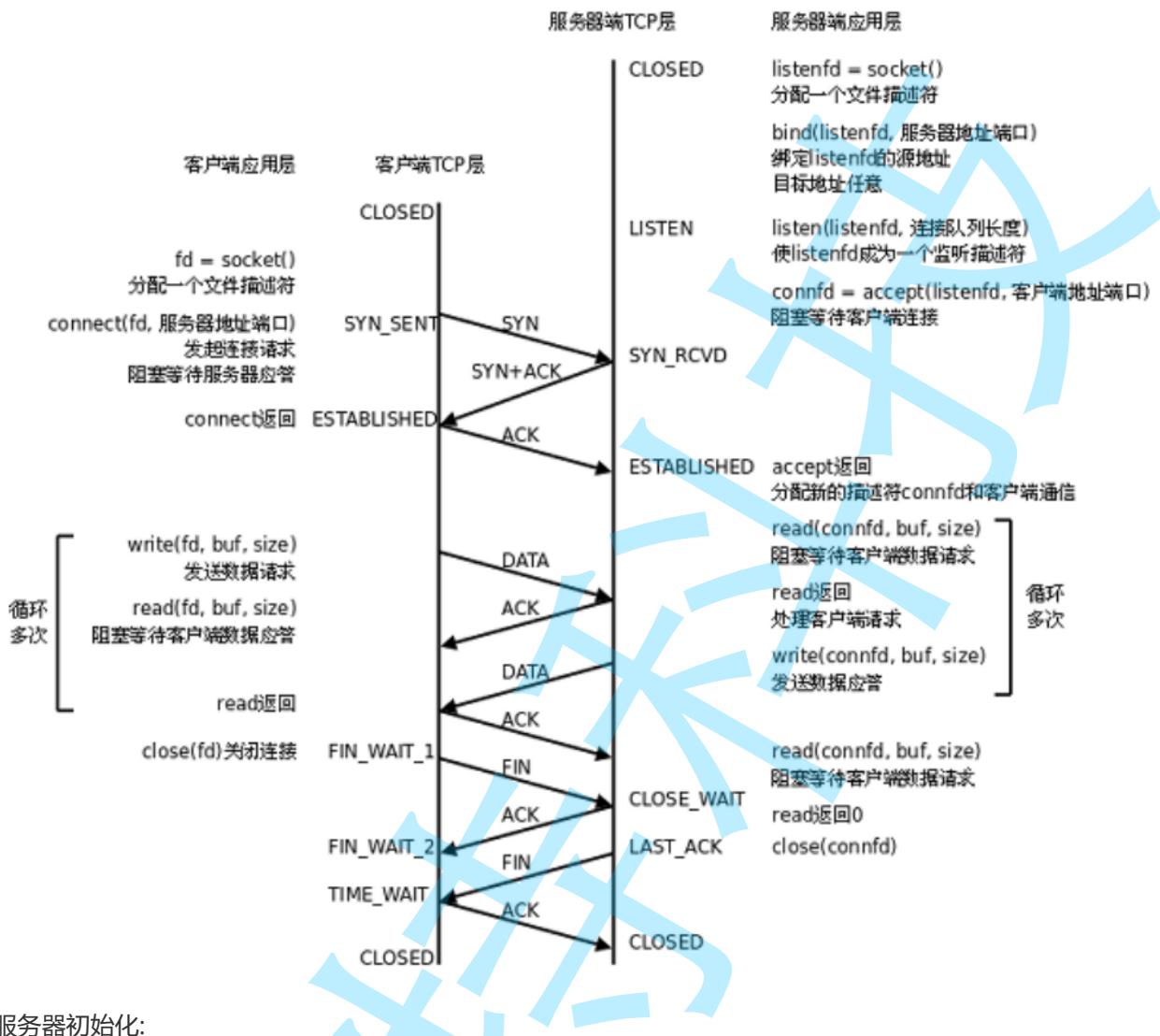
## 线程池版本的 TCP 服务器

同学们根据之前学过的线程池, 自行修改服务器为线程池版本.

## TCP 协议通讯流程

## 谈恋爱例子

下图是基于TCP协议的客户端/服务器程序的一般流程:



服务器初始化:

- 调用socket, 创建文件描述符;
- 调用bind, 将当前的文件描述符和ip/port绑定在一起; 如果这个端口已经被其他进程占用了, 就会bind失败;
- 调用listen, 声明当前这个文件描述符作为一个服务器的文件描述符, 为后面的accept做好准备;
- 调用accept, 并阻塞, 等待客户端连接过来;

建立连接的过程:

- 调用socket, 创建文件描述符;
- 调用connect, 向服务器发起连接请求;
- connect会发出SYN段并阻塞等待服务器应答; (第一次)
- 服务器收到客户端的SYN, 会应答一个SYN-ACK段表示"同意建立连接"; (第二次)
- 客户端收到SYN-ACK后会从connect()返回, 同时应答一个ACK段; (第三次)

这个建立连接的过程, 通常称为 **三次握手**;

数据传输的过程

- 建立连接后,TCP协议提供全双工的通信服务; 所谓全双工的意思是, 在同一条连接中, 同一时刻, 通信双方可以同时写数据; 相对的概念叫做半双工, 同一条连接在同一时刻, 只能由一方来写数据;
- 服务器从accept()返回后立刻调用read(), 读socket就像读管道一样, 如果没有数据到达就阻塞等待;
- 这时客户端调用write()发送请求给服务器, 服务器收到后从read()返回, 对客户端的请求进行处理, 在此期间客户端调用read()阻塞等待服务器的应答;
- 服务器调用write()将处理结果发回给客户端, 再次调用read()阻塞等待下一条请求;
- 客户端收到后从read()返回, 发送下一条请求, 如此循环下去;

断开连接的过程:

- 如果客户端没有更多的请求了, 就调用close()关闭连接, 客户端会向服务器发送FIN段(第一次);
- 此时服务器收到FIN后, 会回应一个ACK, 同时read会返回0 (第二次);
- read返回之后, 服务器就知道客户端关闭了连接, 也调用close关闭连接, 这个时候服务器会向客户端发送一个FIN; (第三次)
- 客户端收到FIN, 再返回一个ACK给服务器; (第四次)

这个断开连接的过程, 通常称为 **四次挥手**

在学习socket API时要注意应用程序和TCP协议层是如何交互的:

- 应用程序调用某个socket函数时TCP协议层完成什么动作, 比如调用connect()会发出SYN段
- 应用程序如何知道TCP协议层的状态变化, 比如从某个阻塞的socket函数返回就表明TCP协议收到了某些段, 再比如read()返回0就表明收到了FIN段

## TCP 和 UDP 对比

- 可靠传输 vs 不可靠传输
- 有连接 vs 无连接
- 字节流 vs 数据报

# 网络基础2

## 本节重点

- 理解应用层的作用, 初识HTTP协议
- 理解传输层的作用, 深入理解TCP的各项特性和机制
- 对整个TCP/IP协议有系统的理解
- 对TCP/IP协议体系下的其他重要协议和技术有一定的了解
- 学会使用一些分析网络问题的工具和方法

**注意!! 注意!! 注意!!**

- 本课是网络编程的**理论基础**.
- 是一个服务器开发程序员的**重要基本功**.
- 是整个Linux课程中的**重点和难点**.
- 也是各大公司笔试面试的**核心考点**.

## 应用层

我们程序员写的一个个解决我们实际问题, 满足我们日常需求的网络程序, 都是在应用层.

## 再谈 "协议"

协议是一种 "约定". socket api的接口, 在读写数据时, 都是按 "字符串" 的方式来发送接收的. 如果我们要传输一些 "结构化的数据" 怎么办呢?

## 网络版计算器

例如, 我们需要实现一个服务器版的加法器. 我们需要**客户端**把要**计算**的两个加数发过去, 然后由**服务器**进行计算, 最后再把结果返回给**客户端**.

约定方案一:

- 客户端发送一个形如"1+1"的字符串;
- 这个字符串中有两个操作数, 都是整形;
- 两个数字之间会有一个字符是运算符, 运算符只能是 +;
- 数字和运算符之间没有空格;
- ...

约定方案二:

- 定义**结构体**来表示我们需要交互的信息;
- 发送数据时将这个**结构体**按照一个规则转换成字符串, 接收到数据的时候再按照相同的规则把字符串转化回**结构体**;
- 这个过程叫做 "**序列化**" 和 "**反序列化**"

```
// proto.h 定义通信的结构体
```

```
typedef struct Request {
```

```

int a;
int b;
} Request;

typedef struct Response {
    int sum;
} Response;

// client.c 客户端核心代码
Request request;
Response response;

scanf("%d,%d", &request.a, &request.b);
write(fd, &request, sizeof(Request));
read(fd, &response, sizeof(Response));

// server.c 服务端核心代码
Request request;
read(client_fd, &request, sizeof(request));
Response response;
response.sum = request.a + request.b;
write(client_fd, &response, sizeof(response));

```

无论我们采用方案一, 还是方案二, 还是其他的方案, 只要保证一端发送时构造的数据, 在另一端能够正确的进行解析, 就是ok的. 这种约定, 就是 **应用层协议**

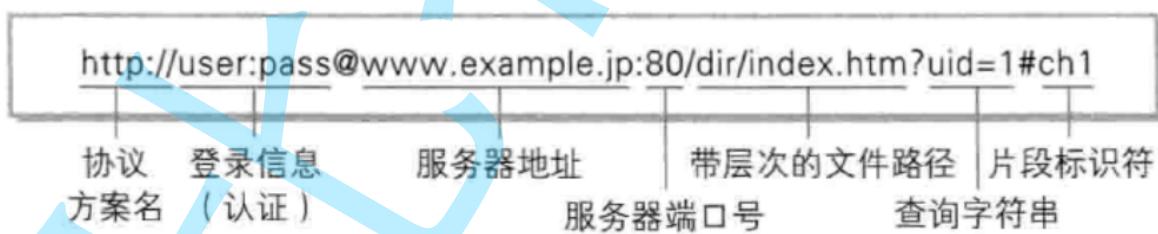
## HTTP协议

虽然我们说, 应用层协议是我们程序猿自己定的.

但实际上, 已经有大佬们定义了一些现成的, 又非常好用的应用层协议, 供我们直接参考使用. HTTP(超文本传输协议)就是其中之一.

## 认识URL

平时我们俗称的 "网址" 其实就是说的 URL



## urlencode和urldecode

像 / ? : 等这样的字符, 已经被url当做特殊意义理解了. 因此这些字符不能随意出现.

比如, 某个参数中需要带有这些特殊字符, 就必须先对特殊字符进行转义.

转义的规则如下:

将需要转码的字符转为16进制, 然后从右到左, 取4位(不足4位直接处理), 每2位做一位, 前面加上%, 编码成%XY格式

例如：

Baidu search results for "C++". The URL in the address bar is https://www.baidu.com/s?wd=c%2B%2B&rsv\_spt=1&rsv\_iqid=0x8d1f3c2a0000475c. The search term "C++" is highlighted in red.

百度为您找到相关结果约30,500,000个

[C++\\_百度百科](#)

**C++**是C语言的继承，它既可以进行C语言的过程化程序设计，又可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。**C++**擅长面向对象程序设计的同时，还可以进行基于过程的程序设计，因而**C++**就适应的问题规模而论，大小由之。**C++**不仅拥有计算机高效运行...

[发展历程](#) [编程开发](#) [语言特点](#) [工作原理](#) [学习指南](#) [更多>>](#)

baike.baidu.com/

[C++ 教程 | 菜鸟教程](#)

"+" 被转义成了 "%2B"

urlencode就是urldecode的逆过程；

[urlencode工具](#)

## HTTP协议格式

HTTP请求

```
POST http://job.xjtu.edu.cn/companyLogin.do HTTP/1.1
Host: job.xjtu.edu.cn
Connection: keep-alive
Content-Length: 36
Cache-Control: max-age=0
Origin: http://job.xjtu.edu.cn
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Referer: http://job.xjtu.edu.cn/companyLogin.do
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.8
Cookie: JSESSIONID=D628A75845A74D29D991DB47A461E4FC;
Hm_lvt_783e83ce0ee350e23a9d389df580f658=1504963710,1506661798;
Hm_lpvt_783e83ce0ee350e23a9d389df580f658=1506661802
username=hgtz2222&password=22222222
```

- 首行: [方法] + [url] + [版本]
- Header: 请求的属性, 冒号分割的键值对; 每组属性之间使用\n分隔; 遇到空行表示Header部分结束

- Body: 空行后面的内容都是Body. Body允许为空字符串. 如果Body存在, 则在Header中会有一个Content-Length属性来标识Body的长度;

## HTTP响应

```
HTTP/1.1 200 OK
Server: YxlinkWAF
Content-Type: text/html; charset=UTF-8
Content-Language: zh-CN
Transfer-Encoding: chunked
Date: Fri, 29 Sep 2017 05:10:13 GMT

<!DOCTYPE html>
<html>
<head>
<title>西安交通大学就业网</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="IE=Edge">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="shortcut icon" href="/renovation/images/icon.ico">
<link href="/renovation/css/main.css" rel="stylesheet" media="screen" />
<link href="/renovation/css/art_default.css" rel="stylesheet" media="screen" />
<link href="/renovation/css/font-awesome.css" rel="stylesheet" media="screen" />
<script type="text/javascript" src="/renovation/js/jquery1.7.1.min.js"></script>
<script type="text/javascript" src="/renovation/js/main.js"></script><!--main-->
<link href="/style/warmTipsstyle.css" rel="stylesheet" type="text/css">
</head>
```

- 首行: [版本号] + [状态码] + [状态码解释]
- Header: 请求的属性, 冒号分割的键值对; 每组属性之间使用\n分隔; 遇到空行表示Header部分结束
- Body: 空行后面的内容都是Body. Body允许为空字符串. 如果Body存在, 则在Header中会有一个Content-Length属性来标识Body的长度; 如果服务器返回了一个html页面, 那么html页面内容就是在body中.

## HTTP的方法

方法	说明	支持的HTTP协议版本
GET	获取资源	1.0、1.1
POST	传输实体主体	1.0、1.1
PUT	传输文件	1.0、1.1
HEAD	获得报文首部	1.0、1.1
DELETE	删除文件	1.0、1.1
OPTIONS	询问支持的方法	1.1
TRACE	追踪路径	1.1
CONNECT	要求用隧道协议连接代理	1.1
LINK	建立和资源之间的联系	1.0
UNLINK	断开连接关系	1.0

其中最常用的就是GET方法和POST方法.

## HTTP的状态码

类别	原因短语
1XX	Informational ( 信息性状态码 )
2XX	Success ( 成功状态码 )
3XX	Redirection ( 重定向状态码 )
4XX	Client Error ( 客户端错误状态码 )
5XX	Server Error ( 服务器错误状态码 )

最常见的状态码, 比如 200(OK), 404(Not Found), 403(FORBIDDEN), 302(Redirect, 重定向), 504(BAD GATEWAY)

## HTTP常见Header

- Content-Type: 数据类型(text/html等)
- Content-Length: Body的长度
- Host: 客户端告知服务器, 所请求的资源是在哪个主机的那个端口上;
- User-Agent: 声明用户的操作系统和浏览器版本信息;
- referer: 当前页面是从哪个页面跳转过来的;
- location: 搭配3xx状态码使用, 告诉客户端接下来要去哪里访问;
- Cookie: 用于在客户端存储少量信息. 通常用于实现会话(session)的功能;

### User-Agent里的历史故事

## 最简单的HTTP服务器

实现一个最简单的HTTP服务器, 只在网页上输出 "hello world"; 只要我们按照HTTP协议的要求构造数据, 就很容易能做到;

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void Usage() {
    printf("usage: ./server [ip] [port]\n");
}

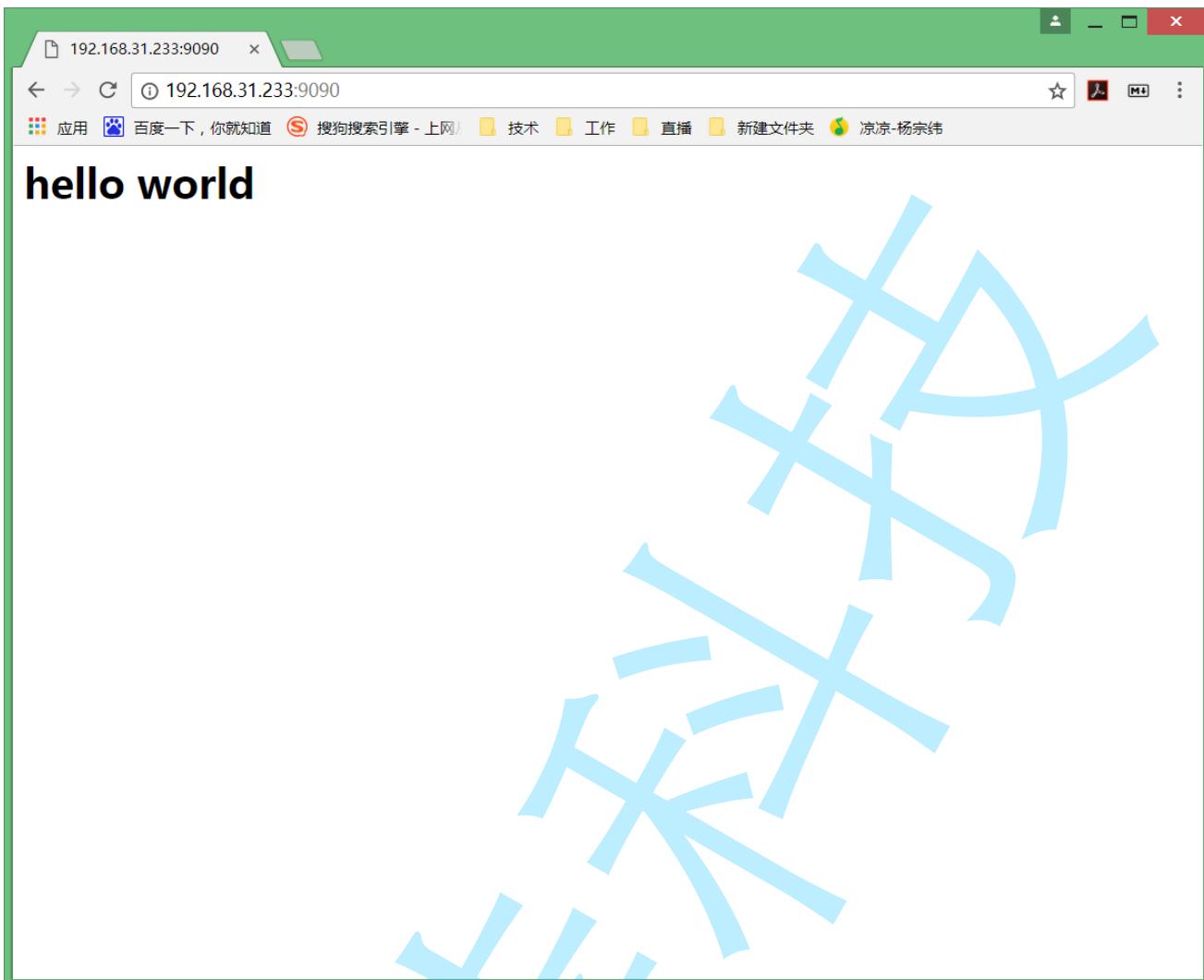
int main(int argc, char* argv[]) {
    if (argc != 3) {
        Usage();
        return 1;
}
```

```
}

int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
    perror("socket");
    return 1;
}
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr(argv[1]);
addr.sin_port = htons(atoi(argv[2]));

int ret = bind(fd, (struct sockaddr*)&addr, sizeof(addr));
if (ret < 0) {
    perror("bind");
    return 1;
}
ret = listen(fd, 10);
if (ret < 0) {
    perror("listen");
    return 1;
}
for (;;) {
    struct sockaddr_in client_addr;
    socklen_t len;
    int client_fd = accept(fd, (struct sockaddr*)&client_addr, &len);
    if (client_fd < 0) {
        perror("accept");
        continue;
    }
    char input_buf[1024 * 10] = {0}; // 用一个足够大的缓冲区直接把数据读完.
    ssize_t read_size = read(client_fd, input_buf, sizeof(input_buf) - 1);
    if (read_size < 0) {
        return 1;
    }
    printf("[Request] %s", input_buf);
    char buf[1024] = {0};
    const char* hello = "<h1>hello world</h1>";
    sprintf(buf, "HTTP/1.0 200 OK\nContent-Length:%lu\n\n%s", strlen(hello), hello);
    write(client_fd, buf, strlen(buf));
}
return 0;
}
```

编译, 启动服务. 在浏览器中输入 `http://[ip]:[port]`, 就能看到显示的结果 "Hello World"



```
[tangzhong@tz http]$ ./server 0 9090
GET / HTTP/1.1
Host: 192.168.31.233:9090
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100
Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.8
```

#### 备注:

此处我们使用 9090 端口号启动了HTTP服务器. 虽然HTTP服务器一般使用80端口, 但这只是一个通用的习惯. 并不是说HTTP服务器就不能使用其他的端口号.  
使用chrome测试我们的服务器时, 可以看到服务器打出的请求中还有一个 `GET /favicon.ico HTTP/1.1` 这样的请求.

同学们自行查找资料, 去理解favicon.ico的作用.

#### 实验

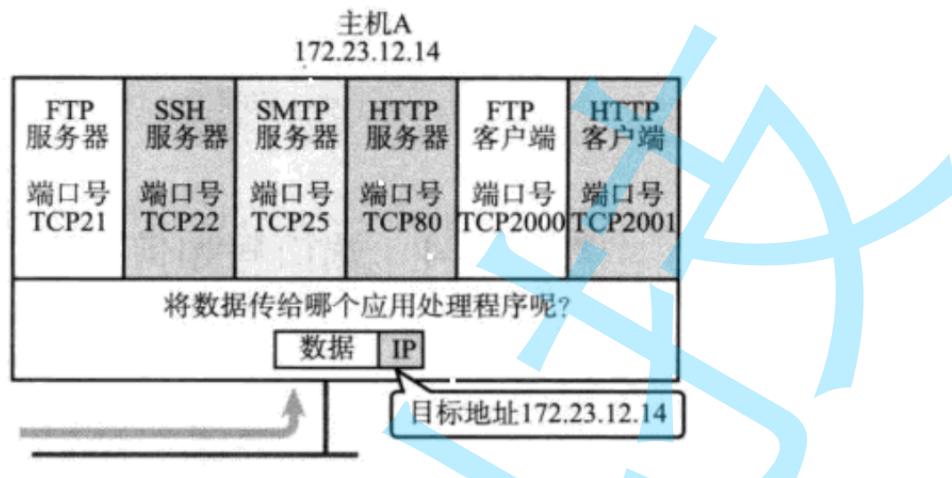
把返回的状态码改成404, 403, 504等, 看浏览器上分别会出现什么样的效果.

## 传输层

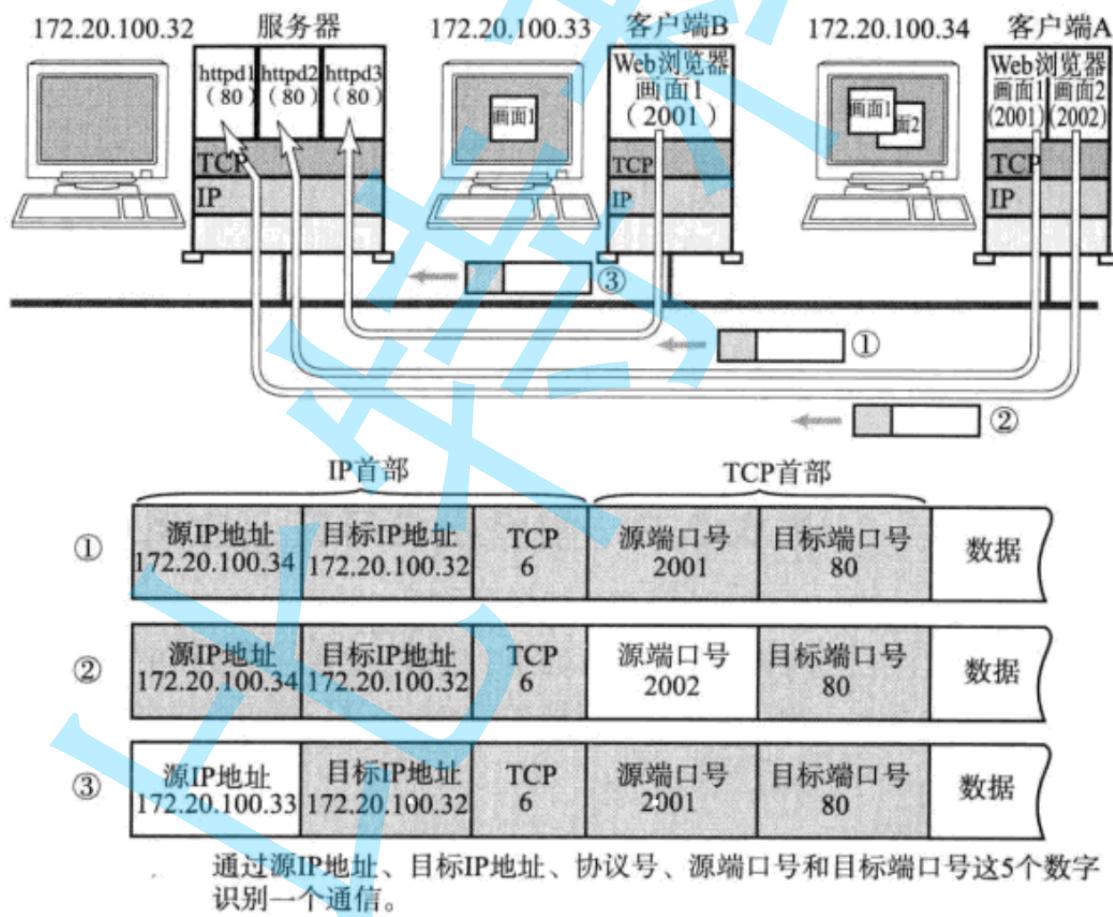
负责数据能够从发送端传输接收端.

## 再谈端口号

端口号(Port)标识了一个主机上进行通信的不同的应用程序;



在TCP/IP协议中,用"源IP", "源端口号", "目的IP", "目的端口号", "协议号"这样一个五元组来标识一个通信(可以通过netstat -n查看);



## 端口号范围划分

- 0 - 1023: 知名端口号, HTTP, FTP, SSH等这些广为使用的应用层协议, 他们的端口号都是固定的.

- 1024 - 65535: 操作系统动态分配的端口号. 客户端程序的端口号, 就是由操作系统从这个范围分配的.

## 认识知名端口号(Well-Known Port Number)

有些服务器是非常常用的, 为了使用方便, 人们约定一些常用的服务器, 都是用以下这些固定的端口号:

- ssh服务器, 使用22端口
- ftp服务器, 使用21端口
- telnet服务器, 使用23端口
- http服务器, 使用80端口
- https服务器, 使用443

执行下面的命令, 可以看到知名端口号

```
cat /etc/services
```

我们自己写一个程序使用端口号时, 要避开这些知名端口号.

## 两个问题

1. 一个进程是否可以bind多个端口号?
2. 一个端口号是否可以被多个进程bind?

## netstat

netstat是一个用来查看网络状态的重要工具.

语法: netstat [选项]

功能: 查看网络状态

常用选项:

- n 拒绝显示别名, 能显示数字的全部转化成数字
- l 仅列出有在 Listen (监听) 的服务状态
- p 显示建立相关链接的程序名
- t (tcp)仅显示tcp相关选项
- u (udp)仅显示udp相关选项
- a (all)显示所有选项, 默认不显示LISTEN相关

## pidof

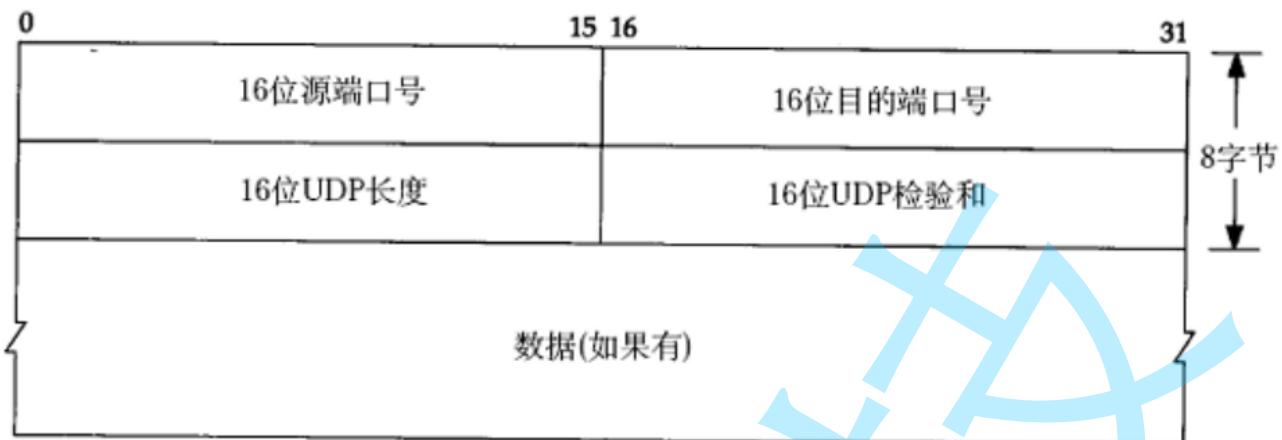
在查看服务器的进程id时非常方便.

语法: pidof [进程名]

功能: 通过进程名, 查看进程id

## UDP协议

### UDP协议端格式



- 16位UDP长度, 表示整个数据报(UDP首部+UDP数据)的最大长度;
- 如果校验和出错, 就会直接丢弃;

## UDP的特点

UDP传输的过程类似于寄信.

- 无连接: 知道对端的IP和端口号就直接进行传输, 不需要建立连接;
- 不可靠: 没有确认机制, 没有重传机制; 如果因为网络故障该段无法发到对方, UDP协议层也不会给应用层返回任何错误信息;
- 面向数据报: 不能够灵活的控制读写数据的次数和数量;

## 面向数据报

应用层交给UDP多长的报文, UDP原样发送, 既不会拆分, 也不会合并;

用UDP传输100个字节的数据:

- 如果发送端调用一次sendto, 发送100个字节, 那么接收端也必须调用对应的一次recvfrom, 接收100个字节; 而不能循环调用10次recvfrom, 每次接收10个字节;

## UDP的缓冲区

- UDP没有真正意义上的**发送缓冲区**. 调用sendto会直接交给内核, 由内核将数据传给网络层协议进行后续的传输动作;
- UDP具有**接收缓冲区**. 但是这个接收缓冲区不能保证收到的UDP报的顺序和发送UDP报的顺序一致; 如果缓冲区满了, 再到达的UDP数据就会被丢弃;

UDP的socket既能读, 也能写, 这个概念叫做**全双工**

## UDP使用注意事项

我们注意到, UDP协议首部中有一个16位的最大长度. 也就是说一个UDP能传输的数据最大长度是64K(包含UDP首部).

然而64K在当今的互联网环境下, 是一个非常小的数字.

如果我们需要传输的数据超过64K, 就需要在应用层手动的分包, 多次发送, 并在接收端手动拼装;

## 基于UDP的应用层协议

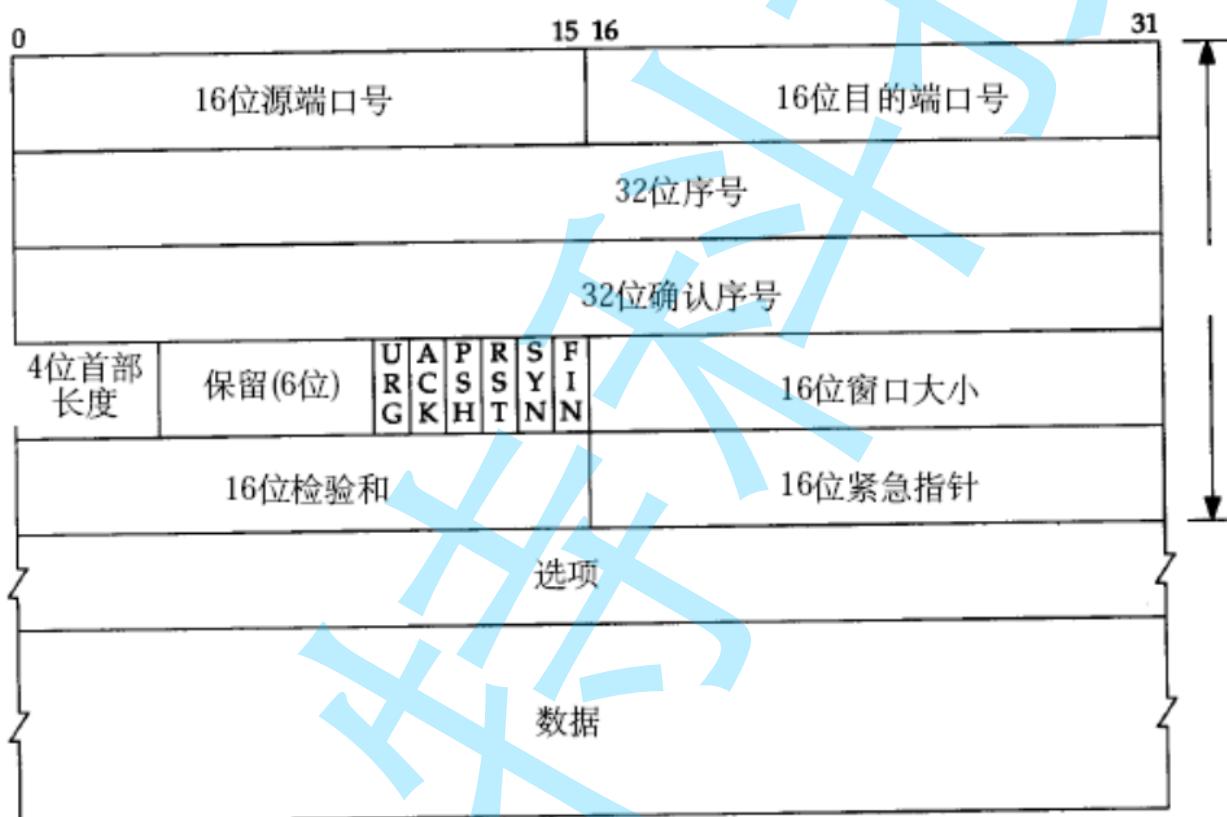
- NFS: 网络文件系统
- TFTP: 简单文件传输协议
- DHCP: 动态主机配置协议
- BOOTP: 启动协议(用于无盘设备启动)
- DNS: 域名解析协议

当然, 也包括你自己写UDP程序时自定义的应用层协议;

## TCP协议

TCP全称为 "传输控制协议(Transmission Control Protocol)". 人如其名, 要对数据的传输进行一个详细的控制;

### TCP协议段格式

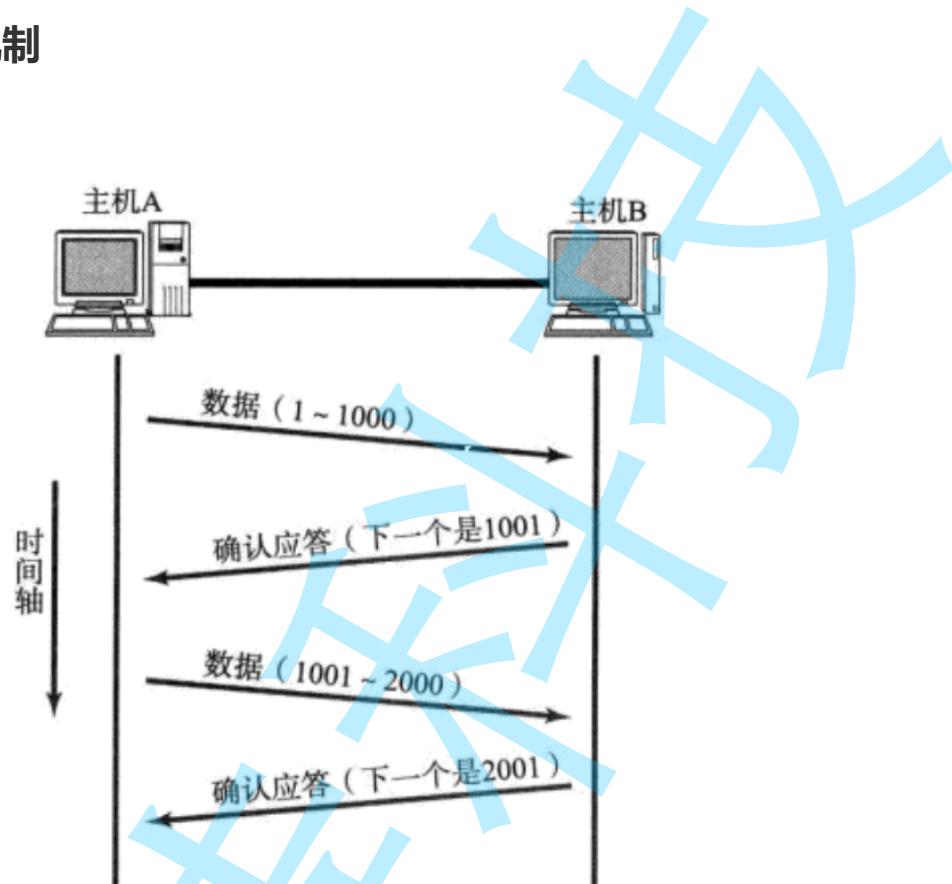


- 源/目的端口号: 表示数据是从哪个进程来, 到哪个进程去;
- 32位序号/32位确认号: 后面详细讲;
- 4位TCP报头长度: 表示该TCP头部有多少个32位bit(有多少个4字节); 所以TCP头部最大长度是 $15 * 4 = 60$
- 6位标志位:
  - URG: 紧急指针是否有效
  - ACK: 确认号是否有效
  - PSH: 提示接收端应用程序立刻从TCP缓冲区把数据读走
  - RST: 对方要求重新建立连接; 我们把携带RST标识的称为**复位报文段**
  - SYN: 请求建立连接; 我们把携带SYN标识的称为**同步报文段**
  - FIN: 通知对方, 本端要关闭了, 我们称携带FIN标识的为**结束报文段**
- 16位窗口大小: 后面再说

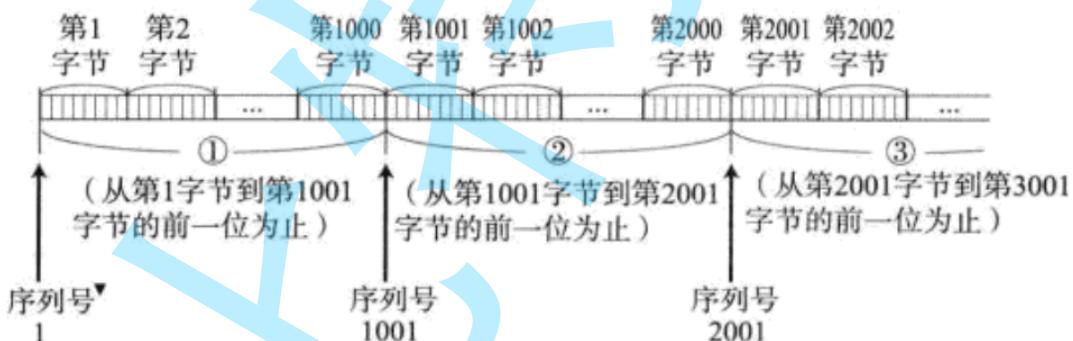
- 16位校验和: 发送端填充, CRC校验. 接收端校验不通过, 则认为数据有问题. 此处的检验和不光包含TCP首部, 也包含TCP数据部分.
- 16位紧急指针: 标识哪部分数据是紧急数据;
- 40字节头部选项: 暂时忽略;

## 确认应答(ACK)机制

[唐僧讲经例子]

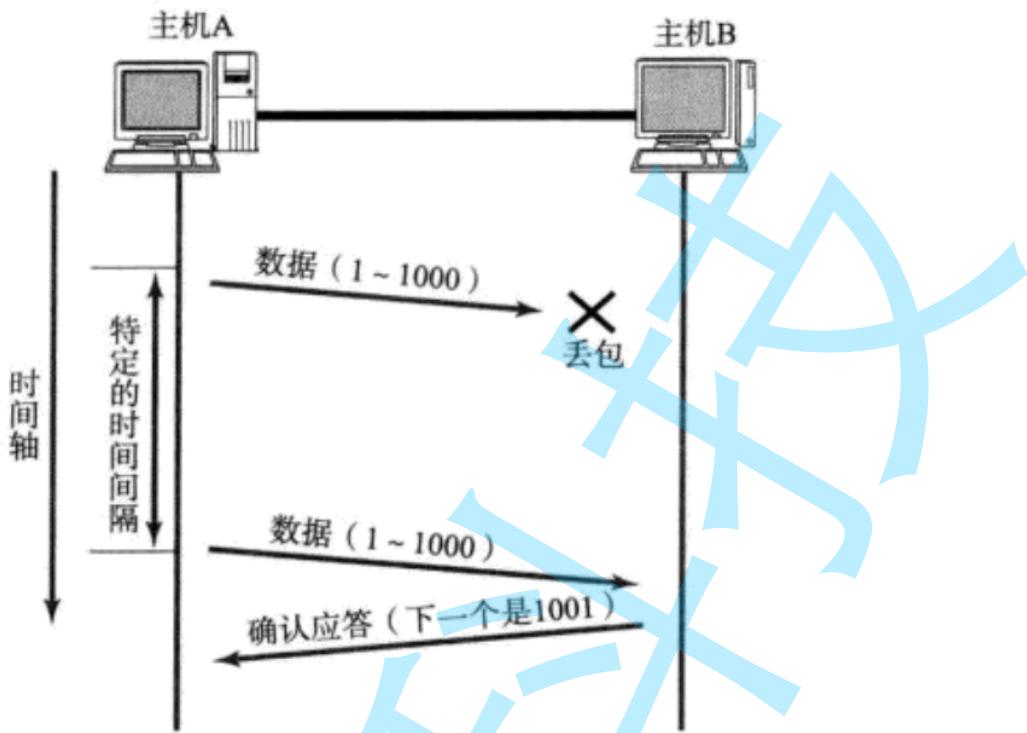


TCP将每个字节的数据都进行了编号. 即为序列号.



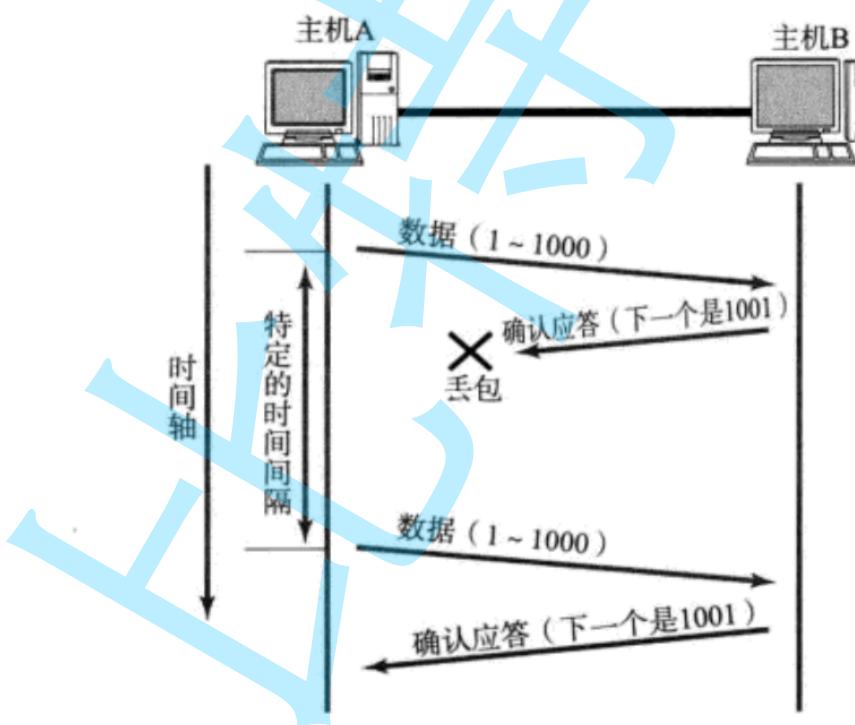
每一个ACK都带有对应的确认序列号, 意思是告诉发送者, 我已经收到了哪些数据; 下一次你从哪里开始发.

## 超时重传机制



- 主机A发送数据给B之后, 可能因为网络拥堵等原因, 数据无法到达主机B;
- 如果主机A在一个特定时间间隔内没有收到B发来的确认应答, 就会进行重发;

但是, 主机A未收到B发来的确认应答, 也可能是因为ACK丢失了;



因此主机B会收到很多重复数据. 那么TCP协议需要能够识别出那些包是重复的包, 并且把重复的丢弃掉. 这时候我们可以利用前面提到的序列号, 就可以很容易做到去重的效果.

那么, 如果超时的时间如何确定?

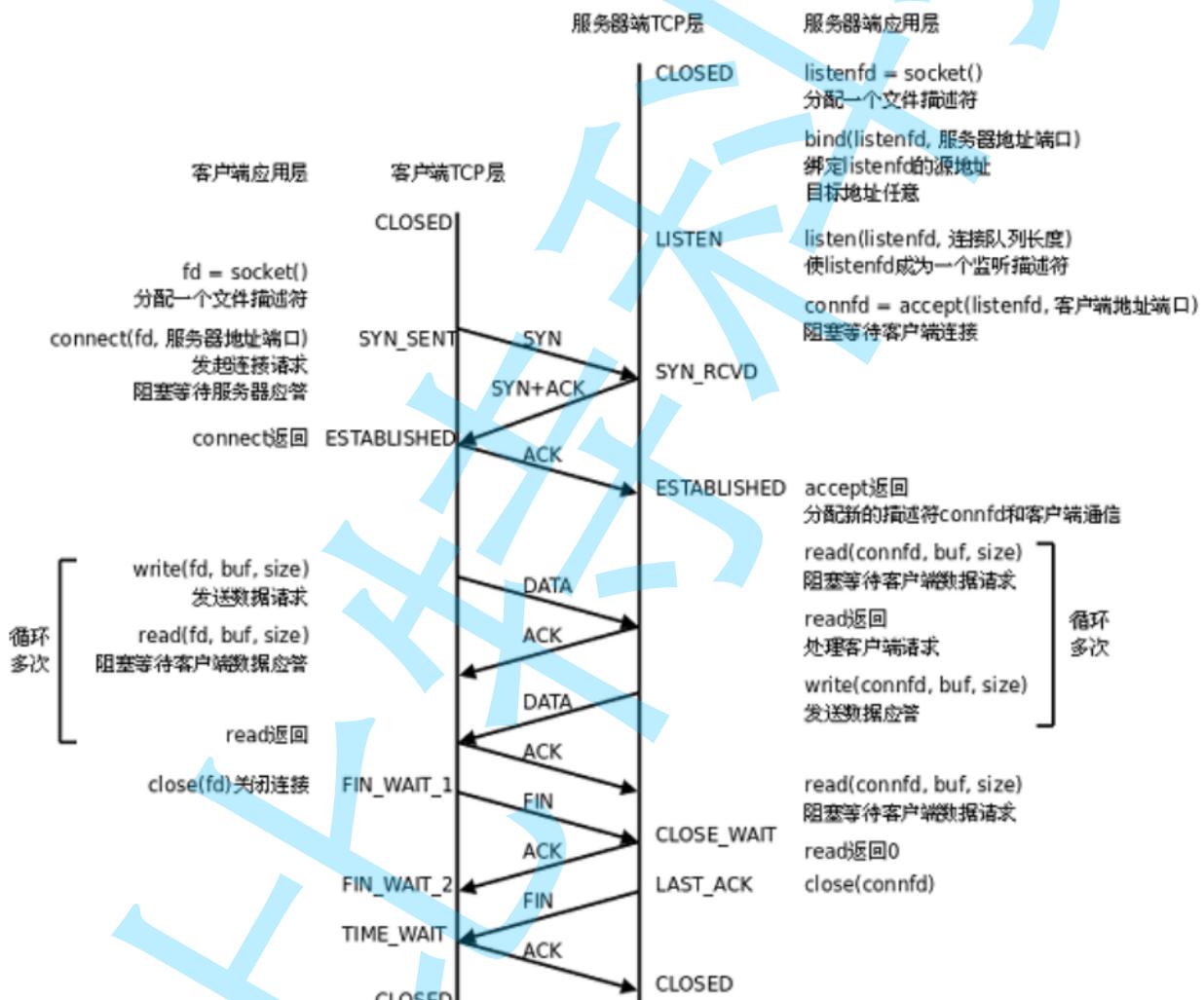
- 最理想的情况下, 找到一个最小的时间, 保证 "确认应答一定能在这个时间内返回".
- 但是这个时间的长短, 随着网络环境的不同, 是有差异的.
- 如果超时时间设的太长, 会影响整体的重传效率;
- 如果超时时间设的太短, 有可能会频繁发送重复的包;

TCP为了保证无论在任何环境下都能比较高性能的通信, 因此会动态计算这个最大超时时间.

- Linux中(BSD Unix和Windows也是如此), 超时以500ms为一个单位进行控制, 每次判定超时重发的超时时间都是500ms的整数倍.
- 如果重发一次之后, 仍然得不到应答, 等待  $2 \times 500\text{ms}$  后再进行重传.
- 如果仍然得不到应答, 等待  $4 \times 500\text{ms}$  进行重传. 依次类推, 以指数形式递增.
- 累计到一定的重传次数, TCP认为网络或者对端主机出现异常, 强制关闭连接.

## 连接管理机制

在正常情况下, TCP要经过三次握手建立连接, 四次挥手断开连接



服务端状态转化:

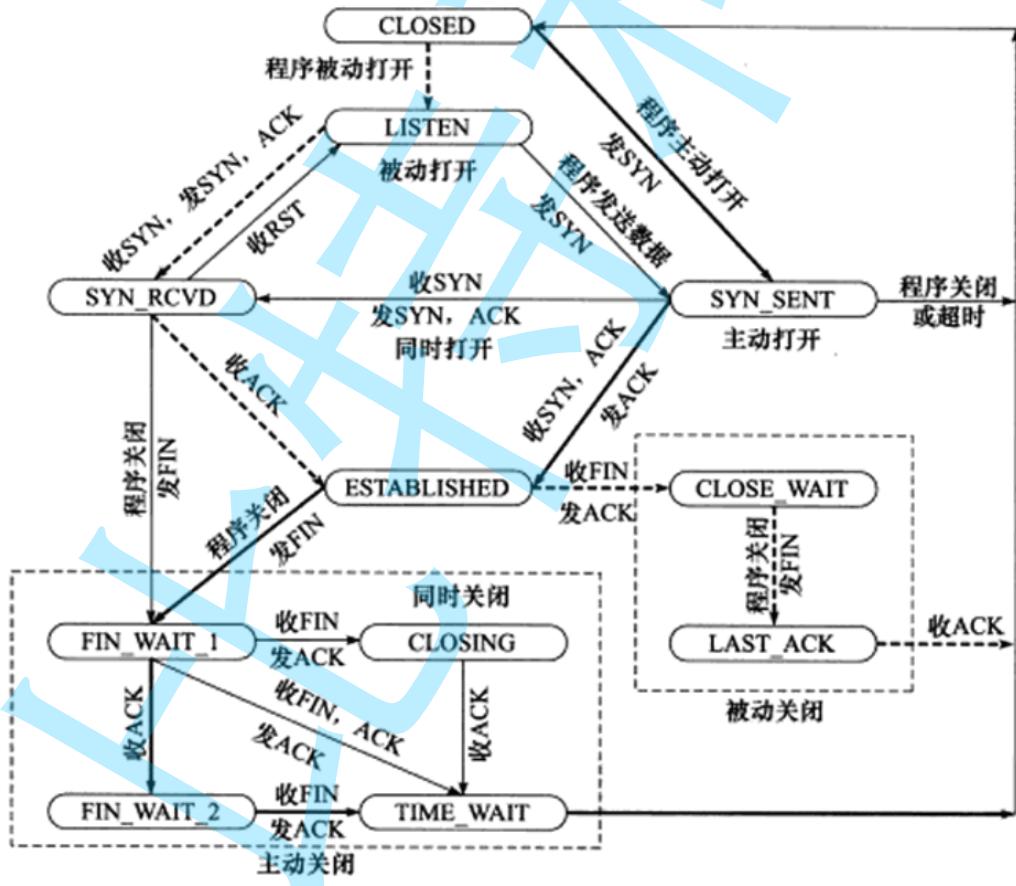
- [CLOSED -> LISTEN] 服务器端调用listen后进入LISTEN状态, 等待客户端连接;
- [LISTEN -> SYN\_RCV] 一旦监听到连接请求(同步报文段), 就将该连接放入内核等待队列中, 并向客户端发送SYN确认报文.

- [SYN\_RECV -> ESTABLISHED] 服务端一旦收到客户端的确认报文, 就进入ESTABLISHED状态, 可以进行读写数据了.
- [ESTABLISHED -> CLOSE\_WAIT] 当客户端主动关闭连接(调用close), 服务器会收到结束报文段, 服务器返回确认报文段并进入CLOSE\_WAIT;
- [CLOSE\_WAIT -> LAST\_ACK] 进入CLOSE\_WAIT后说明服务器准备关闭连接(需要处理完之前的数据); 当服务器真正调用close关闭连接时, 会向客户端发送FIN, 此时服务器进入LAST\_ACK状态, 等待最后一个ACK到来(这个ACK是客户端确认收到了FIN)
- [LAST\_ACK -> CLOSED] 服务器收到了对FIN的ACK, 彻底关闭连接.

客户端状态转化:

- [CLOSED -> SYN\_SENT] 客户端调用connect, 发送同步报文段;
- [SYN\_SENT -> ESTABLISHED] connect调用成功, 则进入ESTABLISHED状态, 开始读写数据;
- [ESTABLISHED -> FIN\_WAIT\_1] 客户端主动调用close时, 向服务器发送结束报文段, 同时进入FIN\_WAIT\_1;
- [FIN\_WAIT\_1 -> FIN\_WAIT\_2] 客户端收到服务器对结束报文段的确认, 则进入FIN\_WAIT\_2, 开始等待服务器的结束报文段;
- [FIN\_WAIT\_2 -> TIME\_WAIT] 客户端收到服务器发来的结束报文段, 进入TIME\_WAIT, 并发出LAST\_ACK;
- [TIME\_WAIT -> CLOSED] 客户端要等待一个2MSL(Max Segment Life, 报文最大生存时间)的时间, 才会进入CLOSED状态.

下图是TCP状态转换的一个汇总:



- 较粗的虚线表示服务端的状态变化情况;
- 较粗的实线表示客户端的状态变化情况;
- CLOSED是一个假想的起始点, 不是真实状态;

## 关于 "半关闭", 男女朋友分手例子

关于CLOSING状态. 同学们可以课后调研一下.

## 理解TIME\_WAIT状态

现在做一个测试,首先启动server,然后启动client,然后用Ctrl-C使server终止,这时马上再运行server,结果是:

```
$ ./server  
bind error: Address already in use
```

这是因为,虽然server的应用程序终止了,但TCP协议层的连接并没有完全断开,因此不能再次监听同样的server端口. 我们用netstat命令查看一下:

```
$ netstat -apn | grep 8000  
tcp        1      0 127.0.0.1:33498          127.0.0.1:8000  
CLOSE_WAIT 10830/client  
tcp        0      0 127.0.0.1:8000          127.0.0.1:33498  
FIN_WAIT2   -
```

- TCP协议规定,主动关闭连接的一方要处于TIME\_WAIT状态,等待两个MSL(maximum segment lifetime)的时间后才能回到CLOSED状态.
- 我们使用Ctrl-C终止了server, 所以server是主动关闭连接的一方, 在TIME\_WAIT期间仍然不能再次监听同样的server端口;
- MSL在RFC1122中规定为两分钟,但是各操作系统的实现不同, 在Centos7上默认配置的值是60s;
- 可以通过 `cat /proc/sys/net/ipv4/tcp_fin_timeout` 查看msl的值;
- 规定TIME\_WAIT的时间请读者参考UNP 2.7节;

```
[tangzhong@tz http]$ cat /proc/sys/net/ipv4/tcp_fin_timeout  
60
```

想一想, 为什么是TIME\_WAIT的时间是2MSL?

- MSL是TCP报文的最大生存时间, 因此TIME\_WAIT持续存在2MSL的话
- 就能保证在两个传输方向上的尚未被接收或迟到的报文段都已经消失(否则服务器立刻重启, 可能会收到来自上一个进程的迟到的数据, 但是这种数据很可能是错误的);
- 同时也是在理论上保证最后一个报文可靠到达(假设最后一个ACK丢失, 那么服务器会再重发一个FIN. 这时虽然客户端的进程不在了, 但是TCP连接还在, 仍然可以重发LAST\_ACK);

## 解决TIME\_WAIT状态引起bind失败的方法(作业)

在server的TCP连接没有完全断开之前不允许重新监听, 某些情况下可能是不合理的

- 服务器需要处理非常大量的客户端的连接(每个连接的生存时间可能很短, 但是每秒都有很大数量的客户端来请求).
- 这个时候如果由服务器端主动关闭连接(比如某些客户端不活跃, 就需要被服务器端主动清理掉), 就会产生大量TIME\_WAIT连接.
- 由于我们的请求量很大, 就可能导致TIME\_WAIT的连接数很多, 每个连接都会占用一个通信五元组(源ip, 源端口, 目的ip, 目的端口, 协议). 其中服务器的ip和端口和协议是固定的. 如果新来的客户端连接的ip和端口号和TIME\_WAIT占用的链接重复了, 就会出现问题.

使用setsockopt()设置socket描述符的 选项SO\_REUSEADDR为1, 表示允许创建端口号相同但IP地址不同的多个socket描述符

```
int opt = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

## 理解 CLOSE\_WAIT 状态

以之前写过的 TCP 服务器为例, 我们稍加修改

将 `new_sock.Close();` 这个代码去掉.

```
#pragma once
#include <functional>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string& req, std::string* resp)> Handler;

class TcpServer {
public:
    TcpServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {}

    bool Start(Handler handler) {
        // 1. 创建 socket;
        CHECK_RET(listen_sock_.Socket());
        // 2. 绑定端口号
        CHECK_RET(listen_sock_.Bind(ip_, port_));
        // 3. 进行监听
        CHECK_RET(listen_sock_.Listen(5));
        // 4. 进入事件循环
        for (;;) {
            // 5. 进行 accept
            TcpSocket new_sock;
            std::string ip;
            uint16_t port = 0;
            if (!listen_sock_.Accept(&new_sock, &ip, &port)) {
                continue;
            }
            printf("[client %s:%d] connect!\n", ip.c_str(), port);
            // 6. 进行循环读写
            for (;;) {
                std::string req;
                // 7. 读取请求. 读取失败则结束循环
                bool ret = new_sock.Recv(&req);
                if (!ret) {
                    printf("[client %s:%d] disconnect!\n", ip.c_str(), port);
                    // [注意!] 将此处的关闭 socket 去掉
                    // new_sock.Close();
                    break;
                }
            }
        }
    }
}
```

```

    // 8. 计算响应
    std::string resp;
    handler(req, &resp);
    // 9. 写回响应
    new_sock.Send(resp);
    printf("[%s:%d] req: %s, resp: %s\n", ip.c_str(), port,
           req.c_str(), resp.c_str());
}
}

return true;
}

private:
    TcpSocket listen_sock_;
    std::string ip_;
    uint64_t port_;
};

```

我们编译运行服务器. 启动客户端链接, 查看 TCP 状态, 客户端服务器都为 ESTABLISHED 状态, 没有问题.

然后我们关闭客户端程序, 观察 TCP 状态

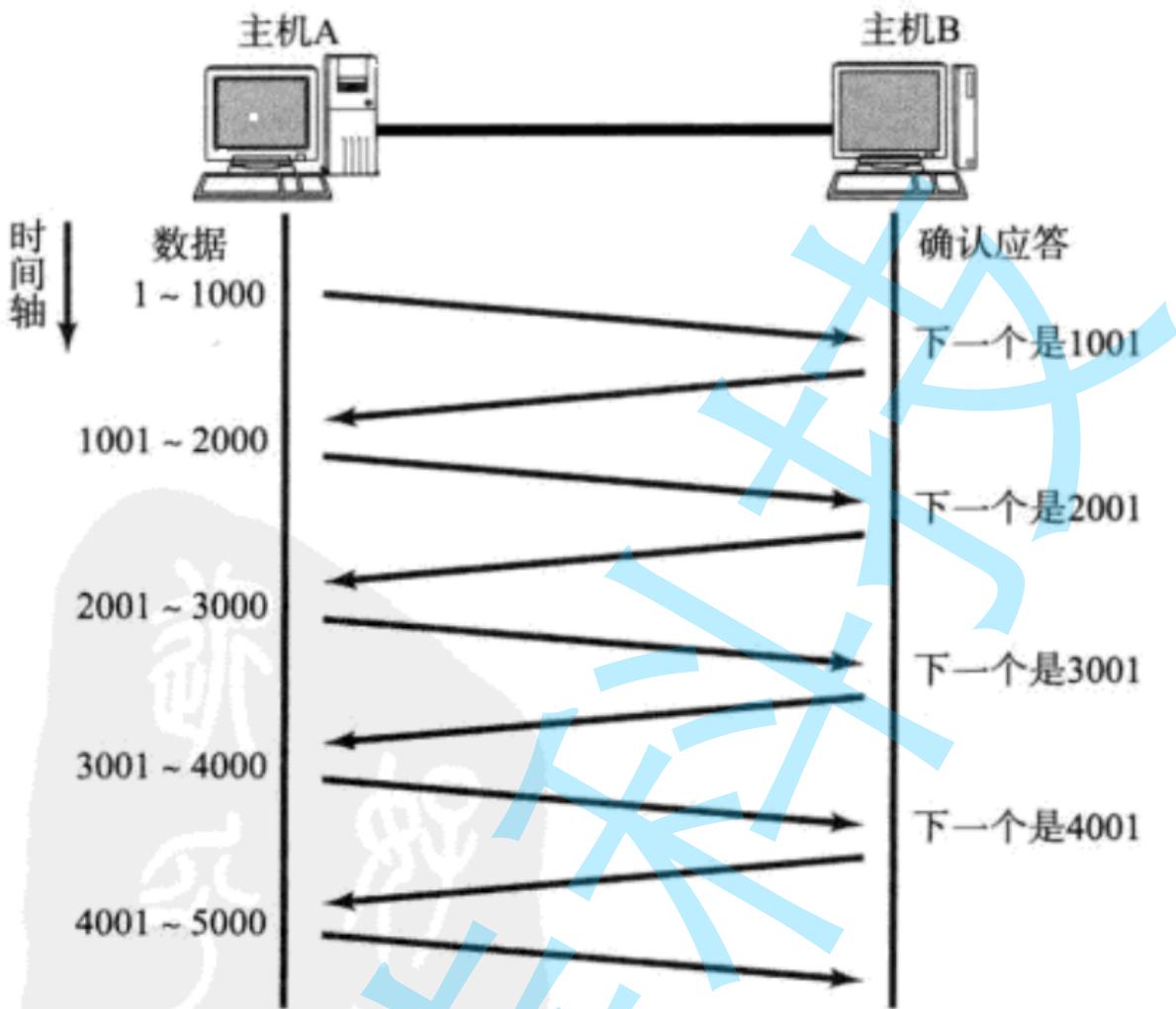
tcp	0	0 0.0.0.0:9090	0.0.0.0:*	LISTEN
5038./dict_server				
tcp	0	0 127.0.0.1:49958	127.0.0.1:9090	FIN_WAIT2 -
tcp	0	0 127.0.0.1:9090	127.0.0.1:49958	CLOSE_WAIT
5038./dict_server				

此时服务器进入了 CLOSE\_WAIT 状态, 结合我们四次挥手的流程图, 可以认为四次挥手没有正确完成.

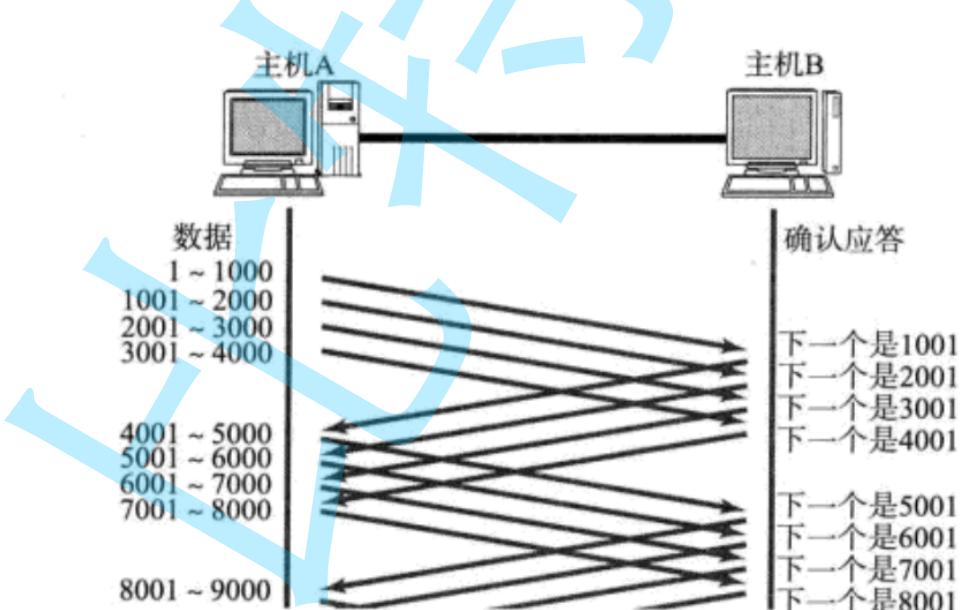
**小结:** 对于服务器上出现大量的 CLOSE\_WAIT 状态, 原因就是服务器没有正确的关闭 socket, 导致四次挥手没有正确完成. 这是一个 BUG. 只需要加上对应的 close 即可解决问题.

## 滑动窗口

刚才我们讨论了确认应答策略, 对每一个发送的数据段, 都要给一个ACK确认应答. 收到ACK后再发送下一个数据段. 这样做有一个比较大的缺点, 就是性能较差. 尤其是数据往返的时间较长的时候.

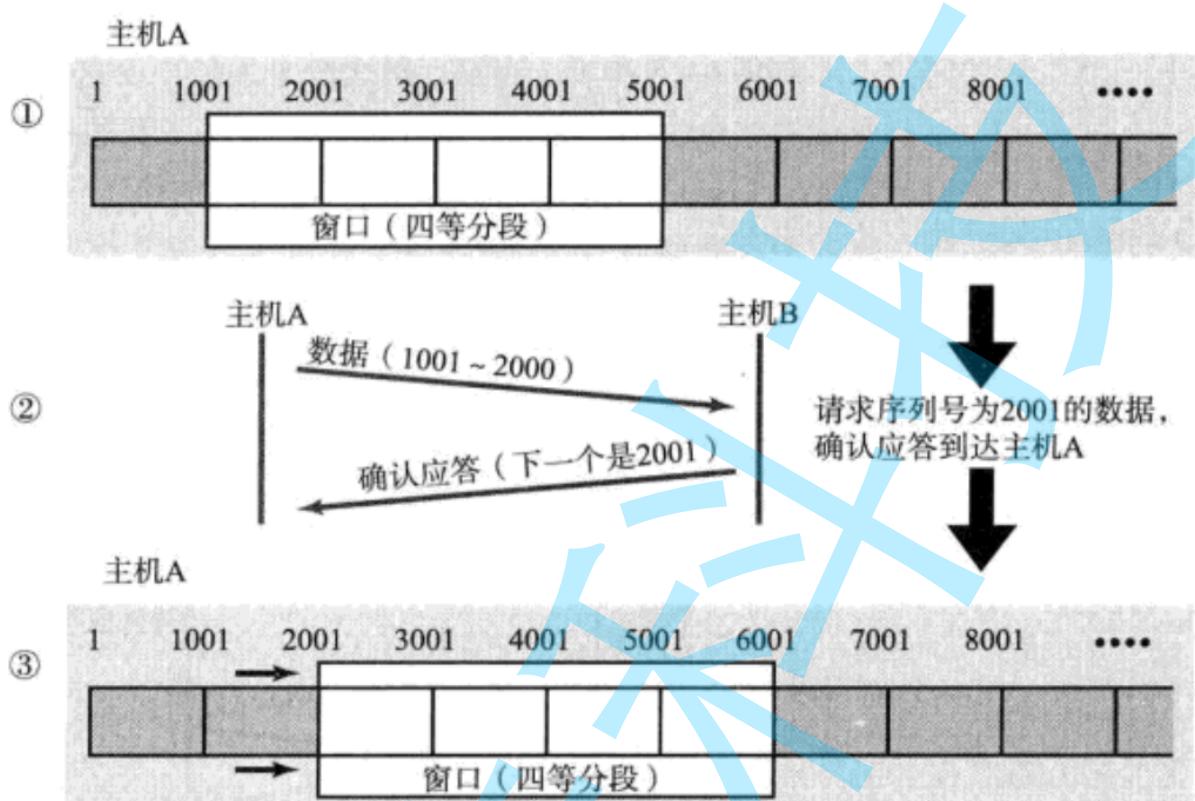


既然这样一一发一收的方式性能较低, 那么我们一次发送多条数据, 就可以大大的提高性能(其实是将多个段的等待时间重叠在一起了).



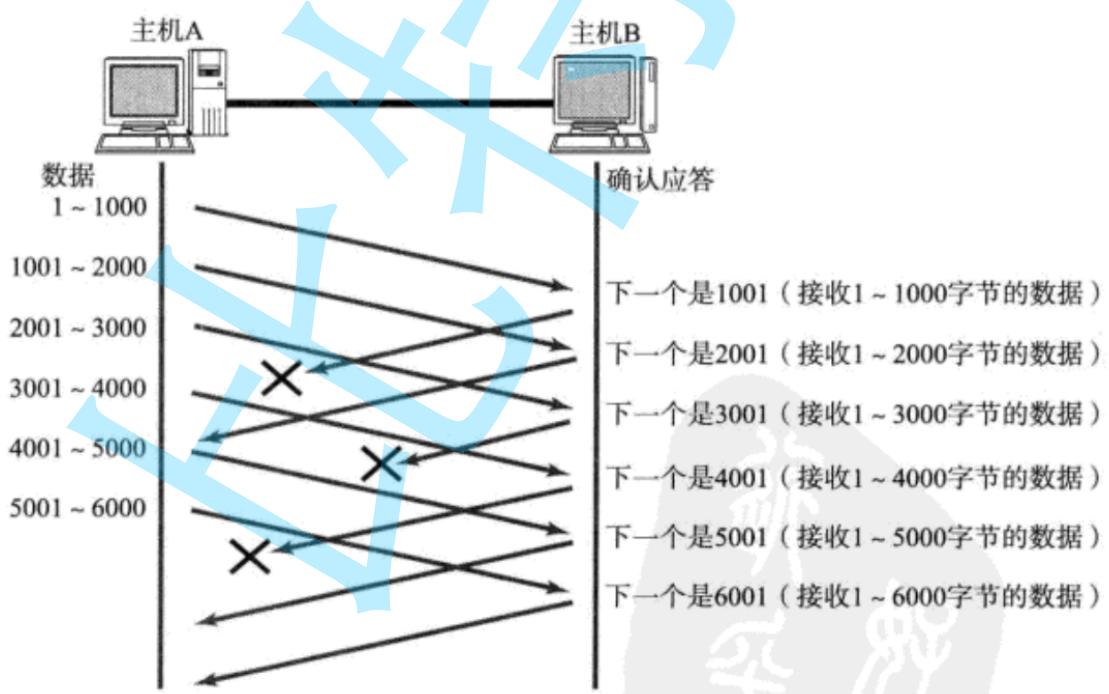
- 窗口大小指的是无需等待确认应答而可以继续发送数据的最大值. 上图的窗口大小就是4000个字节(四个段).
- 发送前四个段的时候, 不需要等待任何ACK, 直接发送;

- 收到第一个ACK后, 滑动窗口向后移动, 继续发送第五个段的数据; 依次类推;
- 操作系统内核为了维护这个滑动窗口, 需要开辟 **发送缓冲区** 来记录当前还有哪些数据没有应答; 只有确认应答过的数据, 才能从缓冲区删掉;
- 窗口越大, 则网络的吞吐率就越高;



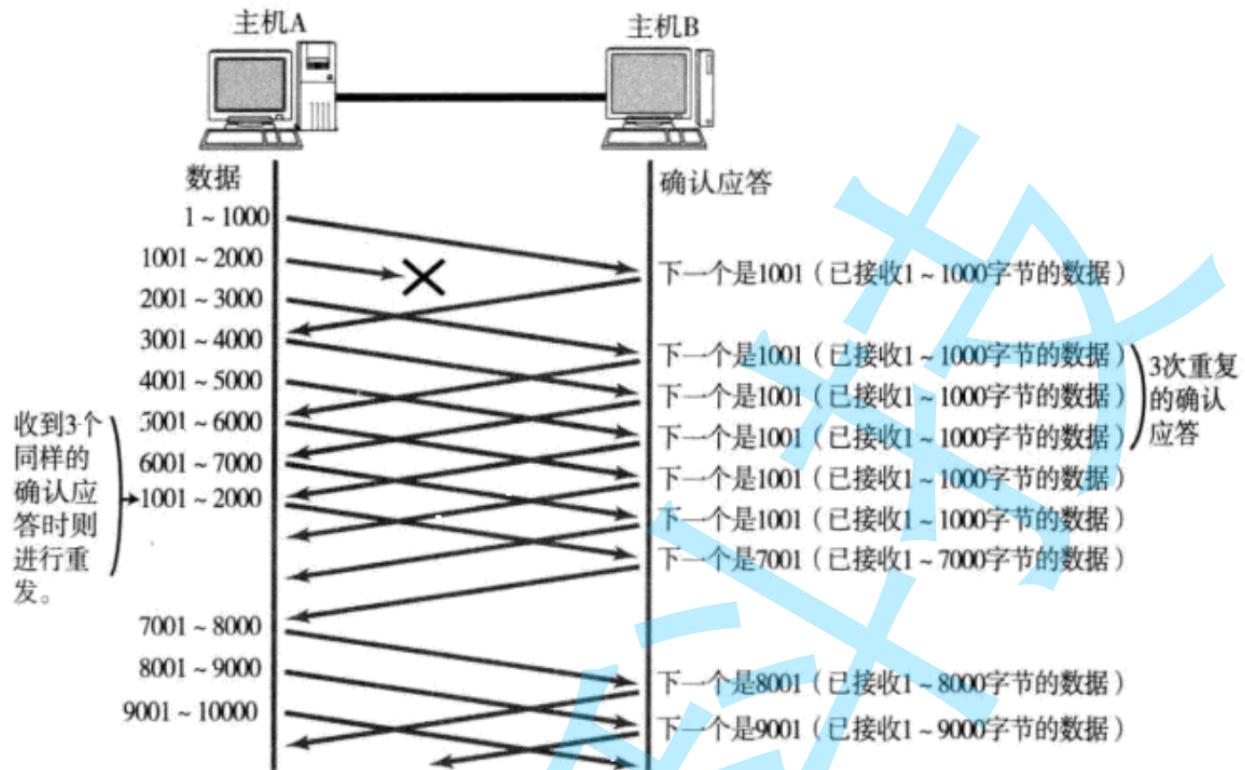
那么如果出现了丢包, 如何进行重传? 这里分两种情况讨论.

**情况一:** 数据包已经抵达, ACK被丢了.



这种情况下, 部分ACK丢了并不要紧, 因为可以通过后续的ACK进行确认;

**情况二:** 数据包就直接丢了.



- 当某一段报文段丢失之后, 发送端会一直收到 1001 这样的ACK, 就像是在提醒发送端 "我想要的是 1001" 一样;
- 如果发送端主机连续三次收到了同一个 "1001" 这样的应答, 就会将对应的数据 1001 - 2000 重新发送;
- 这个时候接收端收到了 1001 之后, 再次返回的ACK就是7001了(因为2001 - 7000)接收端其实之前就已经收到了, 被放到了接收端操作系统内核的接收缓冲区中;

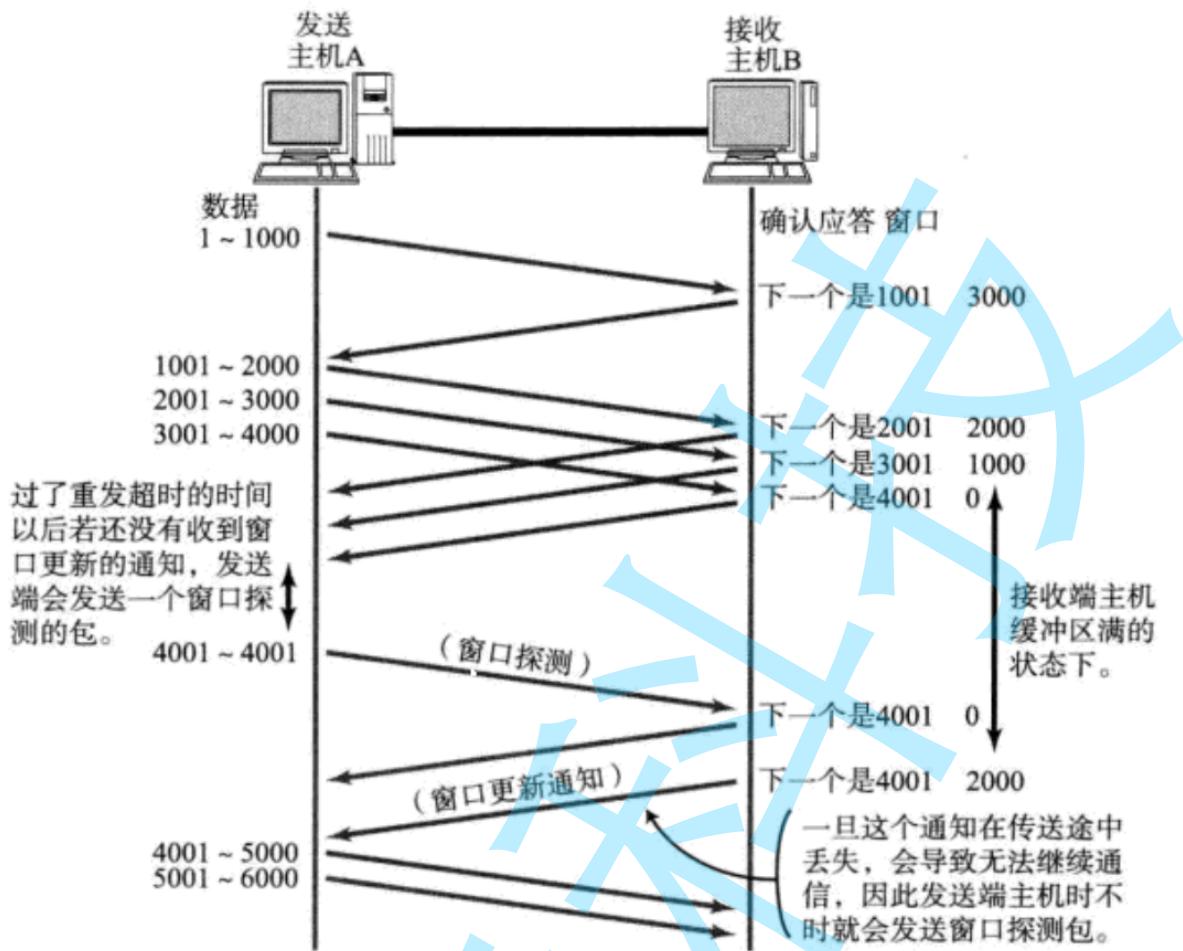
这种机制被称为 "高速重发控制"(也叫 "快重传").

## 流量控制

接收端处理数据的速度是有限的. 如果发送端发的太快, 导致接收端的缓冲区被打满, 这个时候如果发送端继续发送, 就会造成丢包, 继而引起丢包重传等等一系列连锁反应.

因此TCP支持根据接收端的处理能力, 来决定发送端的发送速度. 这个机制就叫做**流量控制(Flow Control)**:

- 接收端将自己可以接收的缓冲区大小放入 TCP 首部中的 "窗口大小" 字段, 通过ACK端通知发送端;
- 窗口大小字段越大, 说明网络的吞吐量越高;
- 接收端一旦发现自己的缓冲区快满了, 就会将窗口大小设置成一个更小的值通知给发送端;
- 发送端接受到这个窗口之后, 就会减慢自己的发送速度;
- 如果接收端缓冲区满了, 就会将窗口置为0; 这时发送方不再发送数据, 但是需要定期发送一个窗口探测数据段, 使接收端把窗口大小告诉发送端.



接收端如何把窗口大小告诉发送端呢？回忆我们的TCP首部中，有一个16位窗口字段，就是存放了窗口大小信息；那么问题来了，16位数字最大表示65535，那么TCP窗口最大就是65535字节么？

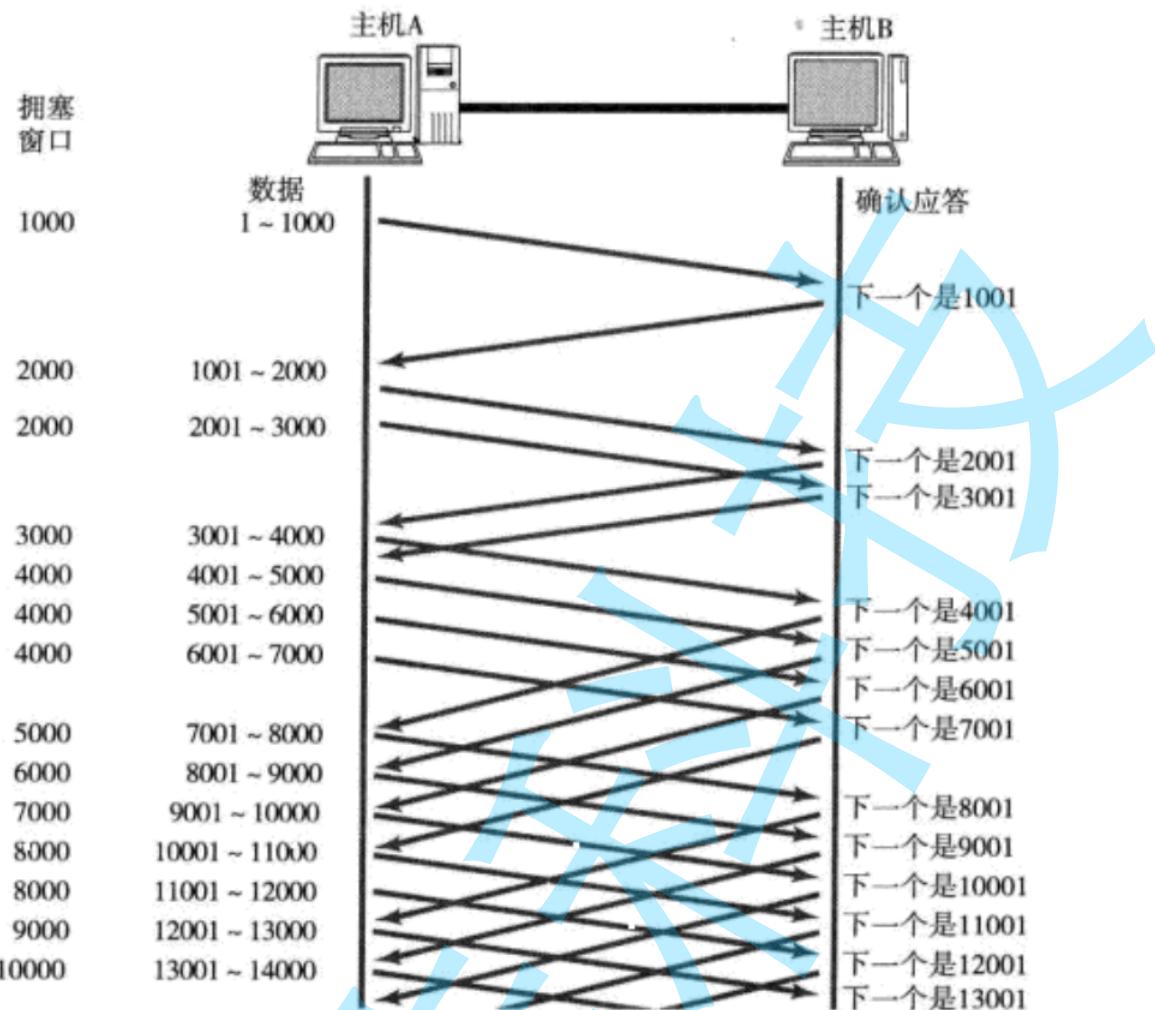
实际上，TCP首部40字节选项中还包含了一个窗口扩大因子M，实际窗口大小是窗口字段的值左移 M 位；

## 拥塞控制

虽然TCP有了滑动窗口这个大杀器，能够高效可靠的发送大量的数据。但是如果在刚开始阶段就发送大量的数据，仍然可能引发问题。

因为网络上有很多的计算机，可能当前的网络状态就已经比较拥堵。在不清楚当前网络状态下，贸然发送大量的数据，是很有可能引起雪上加霜的。

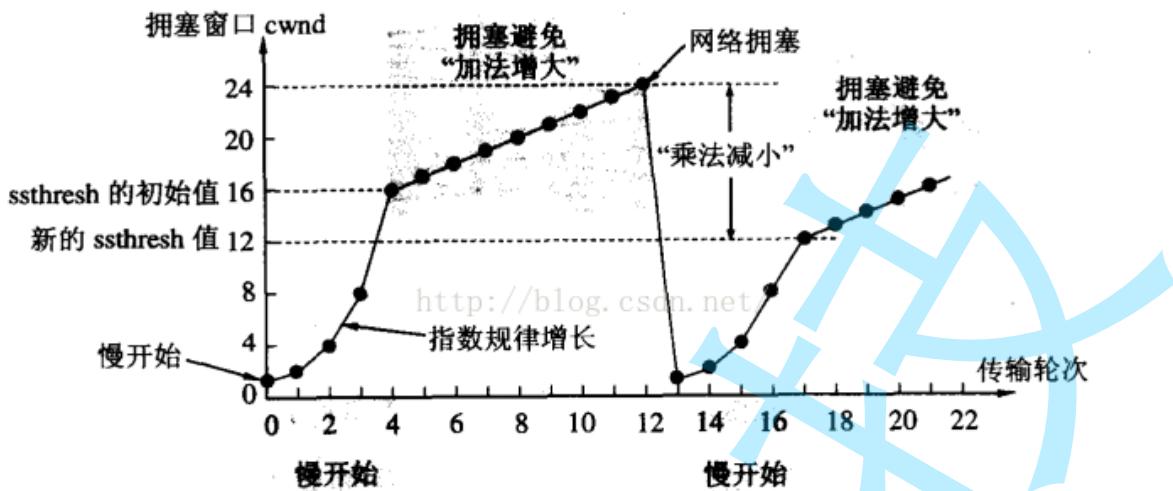
TCP引入 慢启动 机制，先发少量的数据，探路，摸清当前的网络拥堵状态，再决定按照多大的速度传输数据；



- 此处引入一个概念程为**拥塞窗口**
- 发送开始的时候, 定义拥塞窗口大小为1;
- 每次收到一个ACK应答, 拥塞窗口加1;
- 每次发送数据包的时候, 将**拥塞窗口**和**接收端主机反馈的窗口大小**做比较, 取较小的值作为实际发送的窗口;

像上面这样的拥塞窗口增长速度, 是指指数级别的. "慢启动" 只是指初使时慢, 但是增长速度非常快.

- 为了不增长的那么快, 因此不能使拥塞窗口单纯的加倍.
- 此处引入一个叫做慢启动的阈值
- 当拥塞窗口超过这个阈值的时候, 不再按照指数方式增长, 而是按照线性方式增长



- 当TCP开始启动的时候, 慢启动阈值等于窗口最大值;
- 在每次超时重发的时候, 慢启动阈值会变成原来的一半, 同时拥塞窗口置回1;

少量的丢包, 我们仅仅是触发超时重传; 大量的丢包, 我们就认为网络拥塞;

当TCP通信开始后, 网络吞吐量会逐渐上升; 随着网络发生拥堵, 吞吐量会立刻下降;

拥塞控制, 归根结底是TCP协议想尽可能快的把数据传输给对方, 但是又要避免给网络造成太大压力的折中方案.

TCP拥塞控制这样的过程, 就好像 **热恋的感觉**

## 延迟应答

如果接收数据的主机立刻返回ACK应答, 这时候返回的窗口可能比较小.

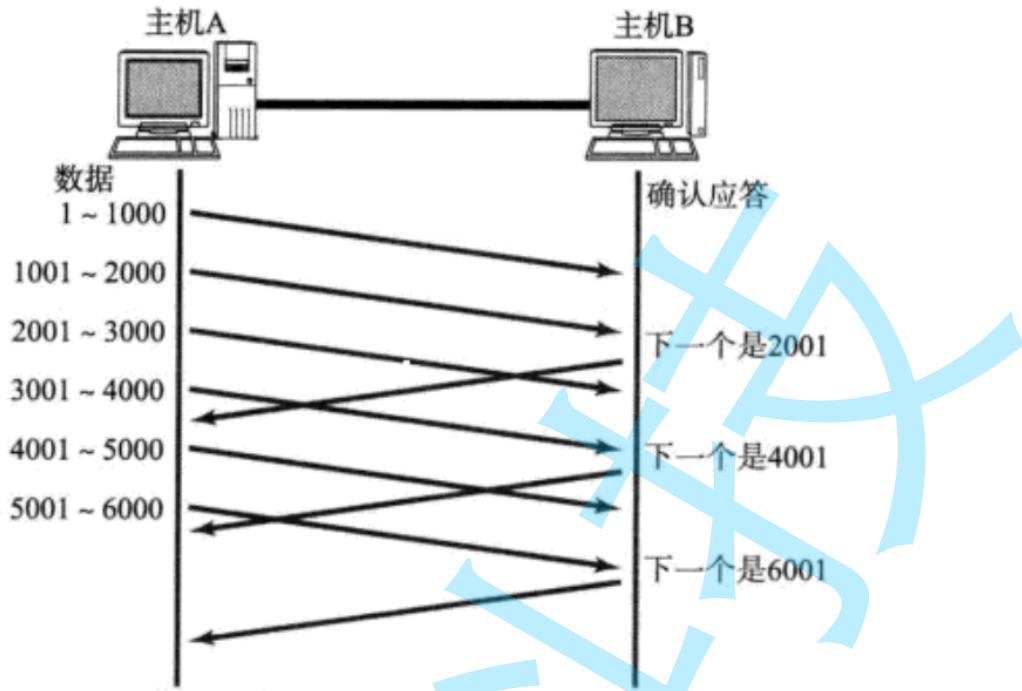
- 假设接收端缓冲区为1M. 一次收到了500K的数据; 如果立刻应答, 返回的窗口就是500K;
- 但实际上可能处理端处理的速度很快, 10ms之内就把500K数据从缓冲区消费掉了;
- 在这种情况下, 接收端处理还远没有达到自己的极限, 即使窗口再放大一些, 也能处理过来;
- 如果接收端稍微等一会再应答, 比如等待200ms再应答, 那么这个时候返回的窗口大小就是1M;

一定要记得, 窗口越大, 网络吞吐量就越大, 传输效率就越高. 我们的目标是在保证网络不拥塞的情况下尽量提高传输效率;

那么所有的包都可以延迟应答么? 肯定也不是;

- 数量限制: 每隔N个包就应答一次;
- 时间限制: 超过最大延迟时间就应答一次;

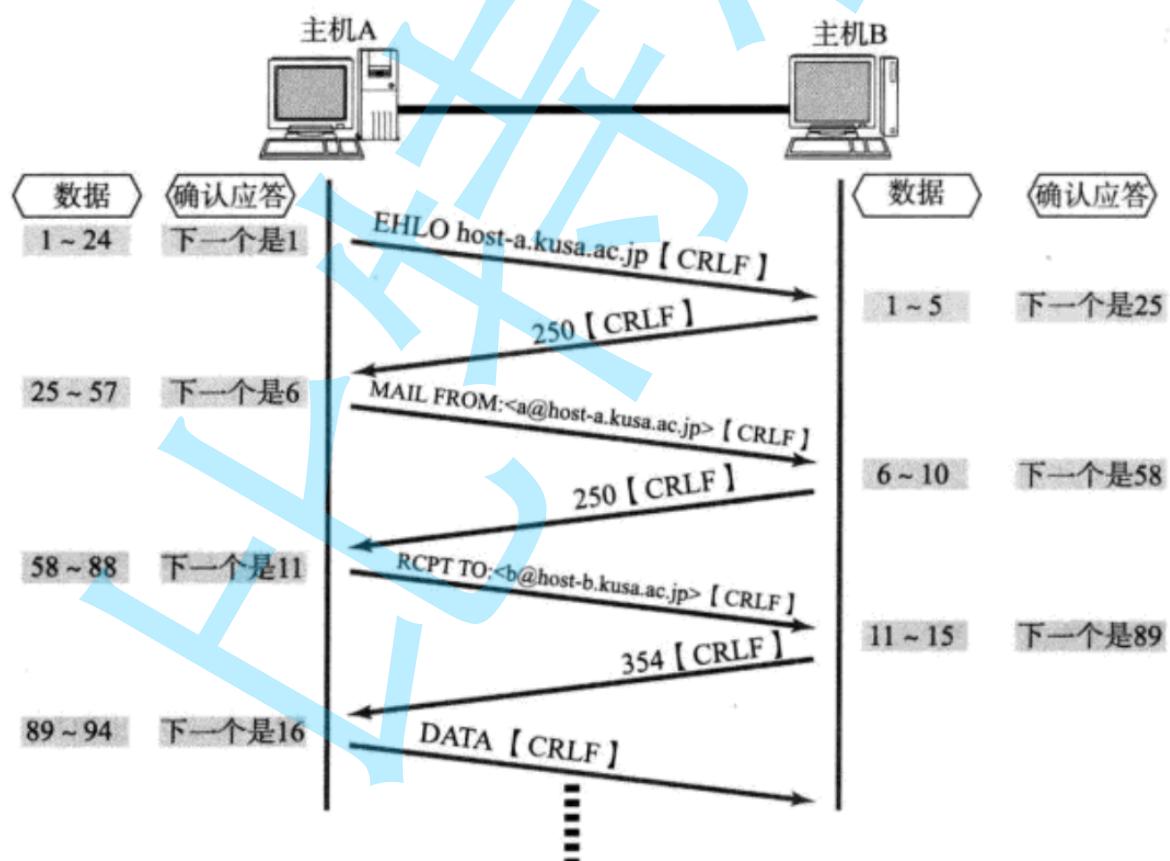
具体的数量和超时时间, 依操作系统不同也有差异; 一般N取2, 超时时间取200ms;



## 捎带应答

在延迟应答的基础上, 我们发现, 很多情况下, 客户端服务器在应用层也是 "一发一收" 的. 意味着客户端给服务器说了 "How are you", 服务器也会给客户端回一个 "Fine, thank you";

那么这个时候ACK就可以搭顺风车, 和服务器回应的 "Fine, thank you" 一起回给客户端



## 面向字节流

创建一个TCP的socket, 同时在内核中创建一个 **发送缓冲区** 和一个 **接收缓冲区**;

- 调用write时, 数据会先写入发送缓冲区中;
- 如果发送的字节数太长, 会被拆分成多个TCP的数据包发出;
- 如果发送的字节数太短, 就会先在缓冲区里等待, 等到缓冲区长度差不多了, 或者其他合适的时机发送出去;
- 接收数据的时候, 数据也是从网卡驱动程序到达内核的接收缓冲区;
- 然后应用程序可以调用read从接收缓冲区拿数据;
- 另一方面, TCP的一个连接, 既有发送缓冲区, 也有接收缓冲区, 那么对于这一个连接, 既可以读数据, 也可以写数据. 这个概念叫做 **全双工**

由于缓冲区的存在, TCP程序的读和写不需要一一匹配, 例如:

- 写100个字节数据时, 可以调用一次write写100个字节, 也可以调用100次write, 每次写一个字节;
- 读100个字节数据时, 也完全不需要考虑写的时候是怎么写的, 既可以一次read 100个字节, 也可以一次read一个字节, 重复100次;

## 粘包问题

### [八戒吃馒头例子]

- 首先要明确, 粘包问题中的 "包", 是指的应用层的数据包.
- 在TCP的协议头中, 没有如同UDP一样的 "报文长度" 这样的字段, 但是有一个序号这样的字段.
- 站在传输层的角度, TCP是一个一个报文过来的. 按照序号排好序放在缓冲区中.
- 站在应用层的角度, 看到的只是一串连续的字节数据.
- 那么应用程序看到了这么一连串的字节数据, 就不知道从哪个部分开始到哪个部分, 是一个完整的应用层数据包.

那么如何避免粘包问题呢? 归根结底就是一句话, **明确两个包之间的边界**.

- 对于定长的包, 保证每次都按固定大小读取即可; 例如上面的Request结构, 是固定大小的, 那么就从缓冲区从头开始按sizeof(Request)依次读取即可;
- 对于变长的包, 可以在包头的位置, 约定一个包总长度的字段, 从而就知道了包的结束位置;
- 对于变长的包, 还可以在包和包之间使用明确的分隔符(应用层协议, 是程序员自己来定的, 只要保证分隔符不和正文冲突即可);

思考: 对于UDP协议来说, 是否也存在 "粘包问题" 呢?

- 对于UDP, 如果还没有上层交付数据, UDP的报文长度仍然在. 同时, UDP是一个一个把数据交付给应用层. 就有很明确的数据边界.
- 站在应用层的角度, 使用UDP的时候, 要么收到完整的UDP报文, 要么不收. 不会出现"半个"的情况.

## TCP异常情况

进程终止: 进程终止会释放文件描述符, 仍然可以发送FIN. 和正常关闭没有什么区别.

机器重启: 和进程终止的情况相同.

机器掉电/网线断开: 接收端认为连接还在, 一旦接收端有写入操作, 接收端发现连接已经不在了, 就会进行reset. 即使没有写入操作, TCP自己也内置了一个保活定时器, 会定期询问对方是否还在. 如果对方不在, 也会把连接释放.

另外, 应用层的某些协议, 也有一些这样的检测机制. 例如HTTP长连接中, 也会定期检测对方的状态. 例如QQ, 在QQ断线之后, 也会定期尝试重新连接.

## TCP小结

为什么TCP这么复杂? 因为要保证可靠性, 同时又尽可能的提高性能.

可靠性:

- 校验和
- 序列号(按序到达)
- 确认应答
- 超时重发
- 连接管理
- 流量控制
- 拥塞控制

提高性能:

- 滑动窗口
- 快速重传
- 延迟应答
- 捎带应答

其他:

- 定时器(超时重传定时器, 保活定时器, TIME\_WAIT定时器等)

## 基于TCP应用层协议

- HTTP
- HTTPS
- SSH
- Telnet
- FTP
- SMTP

当然, 也包括你自己写TCP程序时自定义的应用层协议;

## TCP/UDP对比

我们说了TCP是可靠连接, 那么是不是TCP一定就优于UDP呢? TCP和UDP之间的优点和缺点, 不能简单, 绝对的进行比较

- TCP用于可靠传输的情况, 应用于文件传输, 重要状态更新等场景;
- UDP用于对高速传输和实时性要求较高的通信领域, 例如, 早期的QQ, 视频传输等. 另外UDP可以用于广播;

归根结底, TCP和UDP都是程序员的工具, 什么时机用, 具体怎么用, 还是要根据具体的需求场景去判定.

## 用UDP实现可靠传输(经典面试题)

参考TCP的可靠性机制, 在应用层实现类似的逻辑;

例如:

- 引入序列号, 保证数据顺序;

- 引入确认应答, 确保对端收到了数据;
- 引入超时重传, 如果隔一段时间没有应答, 就重发数据;
- .....

## TCP 相关实验

### 理解 listen 的第二个参数

基于刚才封装的 TcpSocket 实现以下测试代码

对于服务器, listen 的第二个参数设置为 2, 并且不调用 accept

test\_server.cc

```
#include "tcp_socket.hpp"

int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Usage ./test_server [ip] [port]\n");
        return 1;
    }
    TcpSocket sock;
    bool ret = sock.Bind(argv[1], atoi(argv[2]));
    if (!ret) {
        return 1;
    }
    ret = sock.Listen(2);
    if (!ret) {
        return 1;
    }
    // 客户端不进行 accept
    while (1) {
        sleep(1);
    }
    return 0;
}
```

test\_client.cc

```
#include "tcp_socket.hpp"

int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Usage ./test_client [ip] [port]\n");
        return 1;
    }
    TcpSocket sock;
    bool ret = sock.Connect(argv[1], atoi(argv[2]));
    if (ret) {
        printf("connect ok\n");
    }
}
```

```

} else {
    printf("connect failed\n");
}
while (1) {
    sleep(1);
}
return 0;
}

```

此时启动 3 个客户端同时连接服务器, 用 netstat 查看服务器状态, 一切正常.

但是启动第四个客户端时, 发现服务器对于第四个连接的状态存在问题了

tcp	3	0 0.0.0.0:9090	0.0.0.0:*	LISTEN
9084./test_server	0	0 127.0.0.1:9090	127.0.0.1:48178	SYN_RECV -
tcp	0	0 127.0.0.1:9090	127.0.0.1:48176	ESTABLISHED -
tcp	0	0 127.0.0.1:48178	127.0.0.1:9090	ESTABLISHED
9140./test_client	0	0 127.0.0.1:48174	127.0.0.1:9090	ESTABLISHED
tcp	0	0 127.0.0.1:48174	127.0.0.1:9090	ESTABLISHED
9087./test_client	0	0 127.0.0.1:48176	127.0.0.1:9090	ESTABLISHED
tcp	0	0 127.0.0.1:48172	127.0.0.1:9090	ESTABLISHED
9086./test_client	0	0 127.0.0.1:9090	127.0.0.1:48174	ESTABLISHED -
tcp	0	0 127.0.0.1:9090	127.0.0.1:48172	ESTABLISHED -

客户端状态正常, 但是服务器端出现了 SYN\_RECV 状态, 而不是 ESTABLISHED 状态

这是因为, Linux内核协议栈为一个tcp连接管理使用两个队列:

1. 半链接队列 (用来保存处于SYN\_SENT和SYN\_RECV状态的请求)
2. 全连接队列 (accpetd队列) (用来保存处于established状态, 但是应用层没有调用accept取走的请求)

而全连接队列的长度会受到 listen 第二个参数的影响.

全连接队列满了的时候, 就无法继续让当前连接的状态进入 established 状态了.

这个队列的长度通过上述实验可知, 是 listen 的第二个参数 + 1.

## 使用 wireshark 分析 TCP 通信流程

wireshark是 windows 下的一个网络抓包工具. 虽然 Linux 命令行中有 tcpdump 工具同样能完成抓包, 但是 tcpdump 是纯命令行界面, 使用起来不如 wireshark 方便.

**下载 wireshark**

<https://1.na.dl.wireshark.org/win64/Wireshark-win64-2.6.3.exe>

或者

链接: <https://pan.baidu.com/s/159UUloZ8b7guWDeuAHoF9A> 提取码: k79r

## 安装 wireshark

直接双击安装, 没啥太多注意的.

## 启用 telnet 客户端

参考 <https://jingyan.baidu.com/article/95c9d20d96ba4aec4f756154.html>

## 启动 wireshark 并设置过滤器

由于机器上的网络数据报可能较多, 我们只需要关注我们需要的. 因此需要设置过滤器

在过滤器栏中写入

```
ip.addr == [服务器 ip]
```

则只抓取指定ip的数据包.



或者在过滤器中写入

```
tcp.port == 9090
```

则只关注 9090 端口的数据

更多过滤器的设置, 参考

[https://blog.csdn.net/donot\\_worry\\_be\\_happy/article/details/80786241](https://blog.csdn.net/donot_worry_be_happy/article/details/80786241)

## 观察三次握手过程

启动好服务器.

使用 telnet 作为客户端连接上服务器

```
telnet [ip] [port]
```

抓包结果如下:

No.	Time	Source	Destination	Protocol	Length	Info
613	26.491563	192.168.0.107	47.98.116.42	TCP	66	50024 → 9090 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
614	26.536996	47.98.116.42	192.168.0.107	TCP	66	9090 → 50024 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1412 SACK_PERM=1 WS=128
615	26.537064	192.168.0.107	47.98.116.42	TCP	54	50024 → 9090 [ACK] Seq=1 Ack=1 Win=66304 Len=0

观察三个报文各自的序列号和确认序号的规律.

在中间部分可以看到 TCP 报文详细信息

```

    > Transmission Control Protocol, Src Port: 50024, Dst Port: 9090, Seq: 0, Len: 0
      Source Port: 50024
      Destination Port: 9090
      [Stream index: 23]
      [TCP Segment Len: 0]
      Sequence number: 0 (relative sequence number)
      [Next sequence number: 0 (relative sequence number)]
      Acknowledgment number: 0
      1000 .... = Header Length: 32 bytes (8)
    > Flags: 0x002 (SYN)
      Window size value: 64240
      [Calculated window size: 64240]
      Checksum: 0x770e [unverified]
      [Checksum Status: Unverified]
      Urgent pointer: 0

```

## 观察确认应答

在 telnet 中输入一个字符

No.	Time	Source	Destination	Protocol	Length	Info
40	2.798906	192.168.0.107	47.98.116.42	TCP	55	50024 → 9090 [PSH, ACK] Seq=1 Ack=1 Win=259 Len=1
41	2.842804	47.98.116.42	192.168.0.107	TCP	63	9090 → 50024 [PSH, ACK] Seq=1 Ack=2 Win=229 Len=9
42	2.889264	192.168.0.107	47.98.116.42	TCP	54	50024 → 9090 [ACK] Seq=2 Ack=10 Win=259 Len=0

可以看到客户端发送一个长度为 1 字节的数据, 此时服务器返回了一个 ACK 以及一个 9 个字节的响应(捎带应答), 然后客户端再反馈一个 ACK(注意观察 序列号和确认序号)

## 观察四次挥手

在 telnet 中输入 ctrl + ], 回到 telnet 控制界面, 输入 quit 退出.

No.	Time	Source	Destination	Protocol	Length	Info
91	7.353143	192.168.0.107	47.98.116.42	TCP	54	50024 → 9090 [FIN, ACK] Seq=1 Ack=1 Win=259 Len=0
92	7.394765	47.98.116.42	192.168.0.107	TCP	54	9090 → 50024 [FIN, ACK] Seq=1 Ack=2 Win=229 Len=0
93	7.394806	192.168.0.107	47.98.116.42	TCP	54	50024 → 9090 [ACK] Seq=2 Ack=2 Win=259 Len=0

实际上是 "三次挥手", 由于捎带应答, 导致其中的两次重合在了一起.

## 注意事项

如果使用虚拟机部署服务器, 建议使用 "桥接网卡" 的方式连接网络. NAT 方式下由于进行了 ip 和 port 的替换.

使用云服务器测试, 更加直观方便.

# 网络基础3

## 本节重点

- 理解网络层的作用, 深入理解IP协议的基本原理
- 理解数据链路层的作用, 了解ARP协议
- 对整个TCP/IP协议有系统的理解
- 对TCP/IP协议体系下的其他重要协议和技术有一定的了解
- 学会使用一些分析网络问题的工具和方法

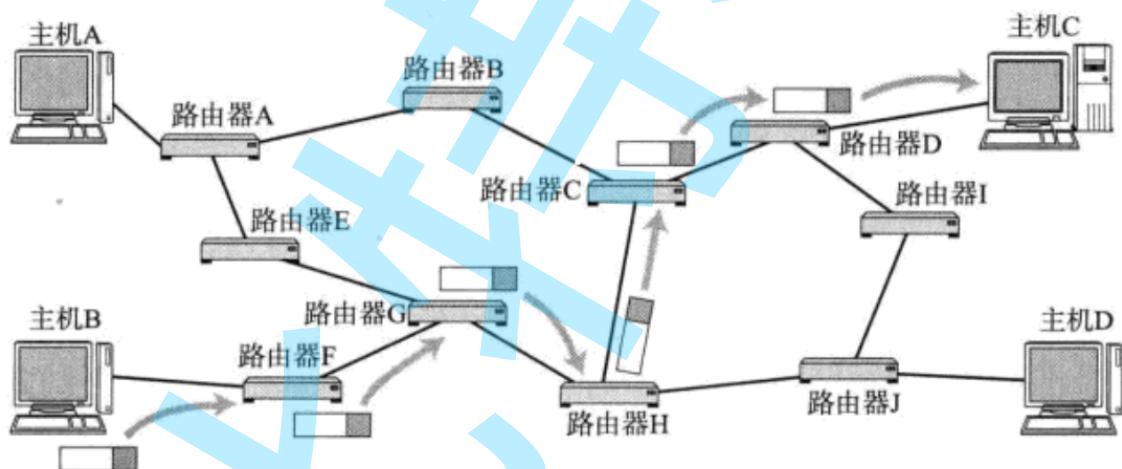
注意!! 注意!! 注意!!

- 本课是网络编程的理论基础.
- 是一个服务器开发程序员的重要基本功.
- 是整个Linux课程中的重点和难点.
- 也是各大公司笔试面试的核心考点.

## 网络层

在复杂的网络环境中确定一个合适的路径.

## IP协议



## 基本概念

主机: 配有IP地址, 但是不进行路由控制的设备; 路由器: 即配有IP地址, 又能进行路由控制; 节点: 主机和路由器的统称;

## 协议头格式



- 4位版本号(version): 指定IP协议的版本, 对于IPv4来说, 就是4.
- 4位头部长度(header length): IP头部的长度是多少个32bit, 也就是  $\text{length} * 4$  的字节数. 4bit表示最大的数字是15, 因此IP头部最大长度是60字节.
- 8位服务类型(Type Of Service): 3位优先权字段(已经弃用), 4位TOS字段, 和1位保留字段(必须置为0). 4位TOS分别表示: 最小延时, 最大吞吐量, 最高可靠性, 最小成本. 这四者相互冲突, 只能选择一个. 对于ssh/telnet这样的应用程序, 最小延时比较重要; 对于ftp这样的程序, 最大吞吐量比较重要.
- 16位总长度(total length): IP数据报整体占多少个字节.
- 16位标识(id): 唯一的标识主机发送的报文. 如果IP报文在数据链路层被分片了, 那么每一个片里面的这个id都是相同的.
- 3位标志字段: 第一位保留(保留的意思是现在不用, 但是还没想好说不定以后要用到). 第二位置为1表示禁止分片, 这时候如果报文长度超过MTU, IP模块就会丢弃报文. 第三位表示"更多分片", 如果分片了的话, 最后一个分片置为1, 其他是0. 类似于一个结束标记.
- 13位分片偏移(fragment offset): 是分片相对于原始IP报文开始处的偏移. 其实就是在表示当前分片在原报文中处在哪个位置. 实际偏移的字节数是这个值 \* 8 得到的. 因此, 除了最后一个报文之外, 其他报文的长度必须是8的整数倍(否则报文就不连续了).
- 8位生存时间(Time To Live, TTL): 数据报到达目的地的最大报文跳数. 一般是64. 每次经过一个路由, TTL -= 1, 一直减到0还没到达, 那么就丢弃了. 这个字段主要是用来防止出现路由循环

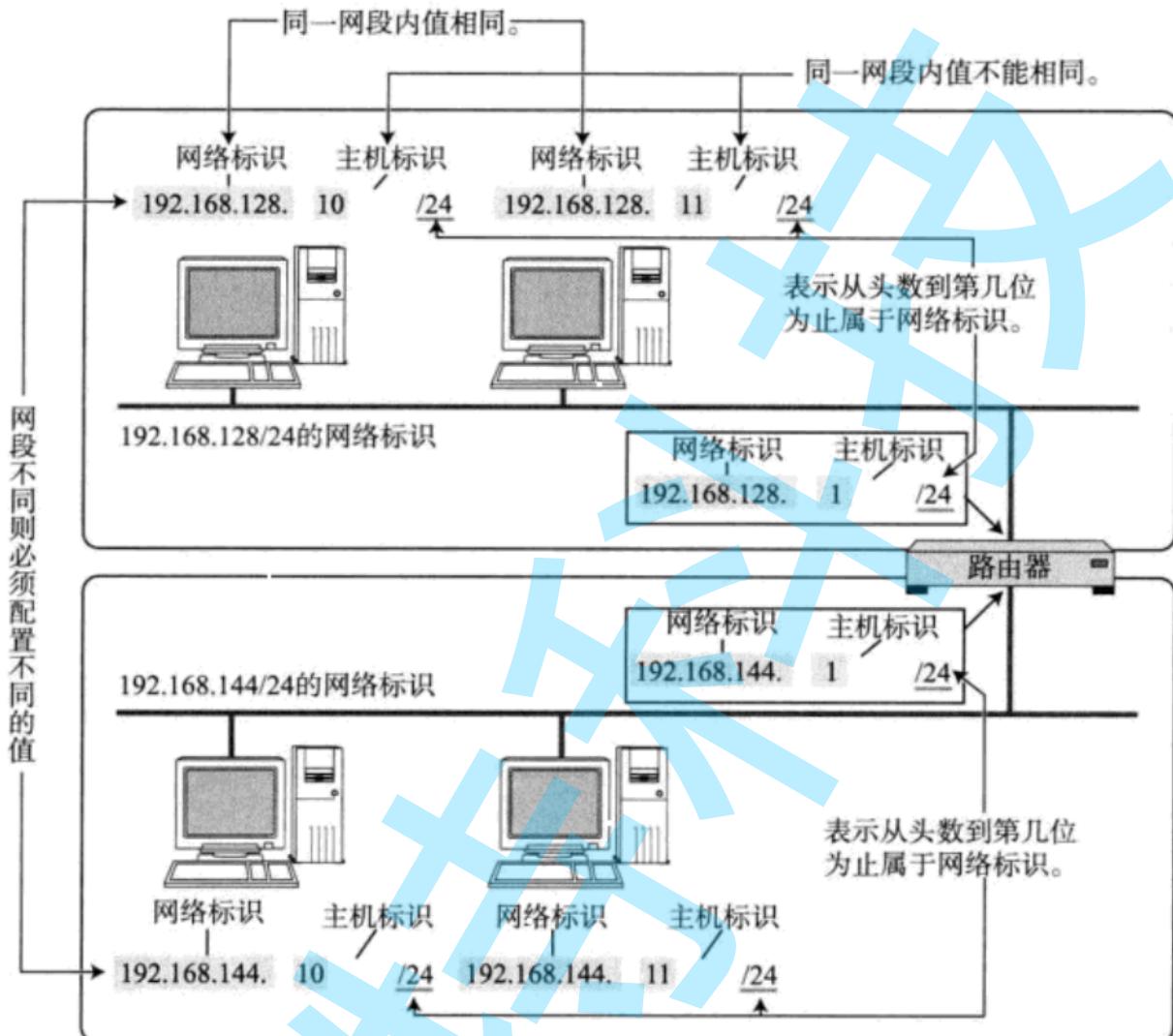
8位协议: 表示上层协议的类型

- 16位头部校验和: 使用CRC进行校验, 来鉴别头部是否损坏.
- 32位源地址和32位目标地址: 表示发送端和接收端.
- 选项字段(不定长, 最多40字节): 略

## 网段划分(重要)

IP地址分为两个部分，网络号和主机号

- 网络号：保证相互连接的两个网段具有不同的标识；
- 主机号：同一网段内，主机之间具有相同的网络号，但是必须有不同的主机号；



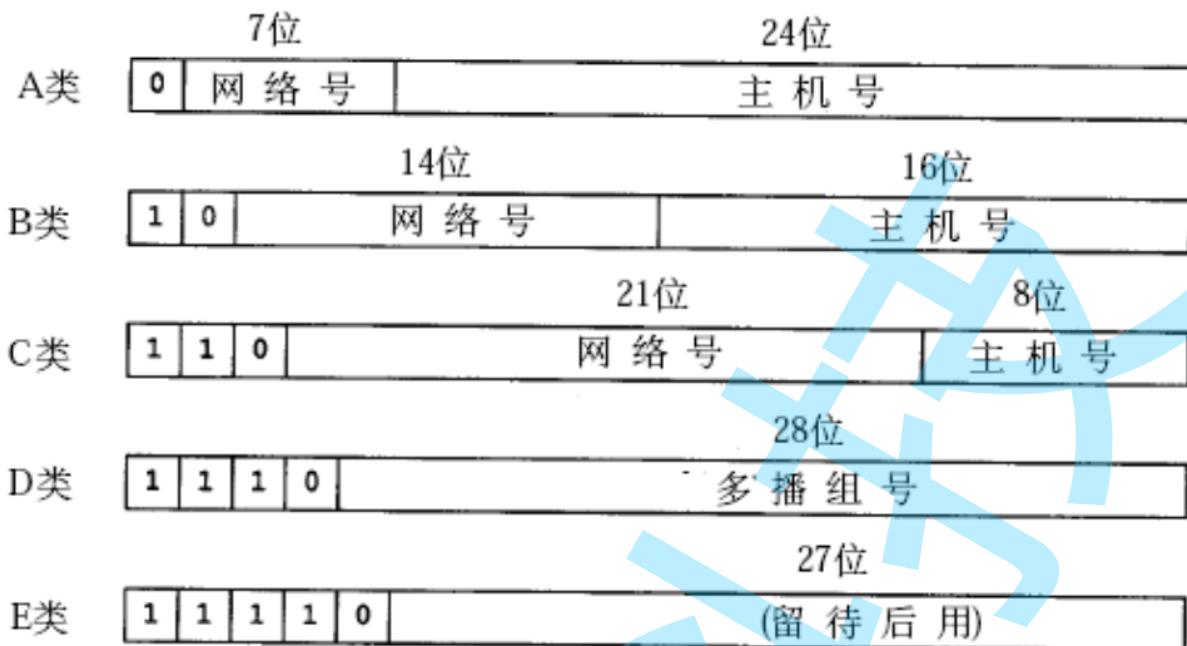
- 不同的子网其实就是把网络号相同的主机放到一起。
- 如果在子网中新增一台主机，则这台主机的网络号和这个子网的网络号一致，但是主机号必须不能和子网中的其他主机重复。

通过合理设置主机号和网络号，就可以保证在相互连接的网络中，每台主机的IP地址都不相同。

那么问题来了，手动管理子网内的IP，是一个相当麻烦的事情。

- 有一种技术叫做DHCP，能够自动的给子网内新增主机节点分配IP地址，避免了手动管理IP的不便。
- 一般的路由器都带有DHCP功能，因此路由器也可以看做一个DHCP服务器。

过去曾经提出一种划分网络号和主机号的方案，把所有IP地址分为五类，如下图所示(该图出自[TCPIP])。



- A类 0.0.0.0到127.255.255.255
- B类 128.0.0.0到191.255.255.255
- C类 192.0.0.0到223.255.255.255
- D类 224.0.0.0到239.255.255.255
- E类 240.0.0.0到247.255.255.255

随着Internet的飞速发展,这种划分方案的局限性很快显现出来,大多数组织都申请B类网络地址,导致B类地址很快就分配完了,而A类却浪费了大量地址;

- 例如,申请了一个B类地址,理论上一个子网内能允许6万5千多个主机.A类地址的子网内的主机数更多.
- 然而实际网络架设中,不会存在一个子网内有这么多的情况.因此大量的IP地址都被浪费掉了.

针对这种情况提出了新的划分方案,称为CIDR(Classless Interdomain Routing):

- 引入一个额外的子网掩码(subnet mask)来区分网络号和主机号;
- 子网掩码也是一个32位的正整数.通常用一串"0"来结尾;
- 将IP地址和子网掩码进行"按位与"操作,得到的结果就是网络号;
- 网络号和主机号的划分与这个IP地址是A类、B类还是C类无关;

下面举两个例子:

划分子网的例子1

IP地址	140.252.20.68	8C FC 14 44
子网掩码	255.255.255.0	FF FF FF 00
网络号	140.252.20.0	8C FC 14 00
子网地址范围	140.252.20.0~140.252.20.255	

划分子网的例子2

IP地址	140.252.20.68	8C FC 14 44
子网掩码	255.255.255.240	FF FF FF F0
网络号	140.252.20.64	8C FC 14 40
子网地址范围	140.252.20.64~140.252.20.79	

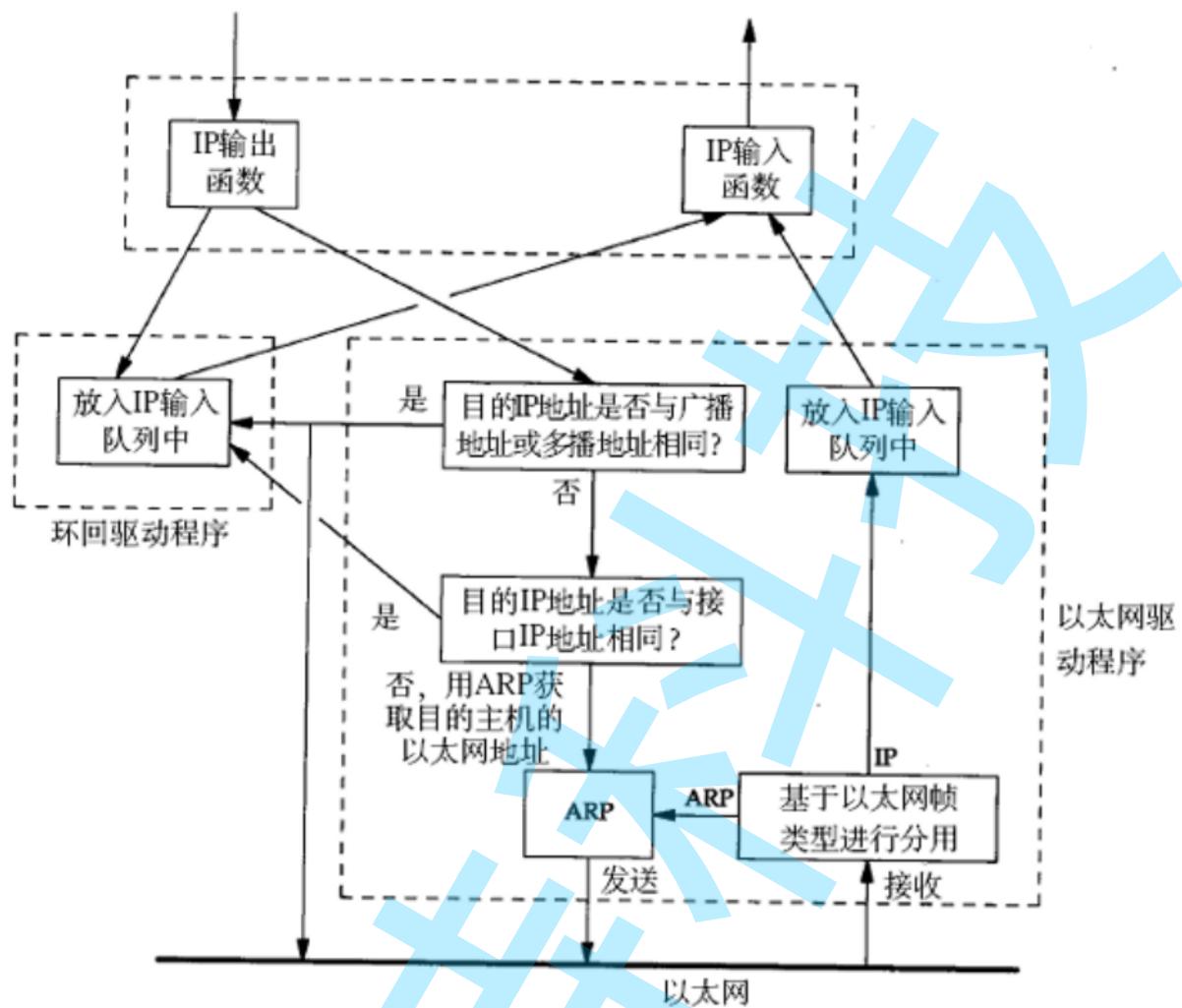
可见,IP地址与子网掩码做与运算可以得到网络号,主机号从全0到全1就是子网的地址范围;

IP地址和子网掩码还有一种更简洁的表示方法,例如140.252.20.68/24,表示IP地址为140.252.20.68,子网掩码的高24位是1,也就是255.255.255.0

## 特殊的IP地址

- 将IP地址中的主机地址全部设为0,就成为了网络号,代表这个局域网;
- 将IP地址中的主机地址全部设为1,就成为了广播地址,用于给同一个链路中相互连接的所有主机发送数据包;
- 127.\*的IP地址用于本机环回(loop back)测试,通常是127.0.0.1

## loopback设备



## IP地址的数量限制

我们知道, IP地址(IPv4)是一个4字节32位的正整数. 那么一共只有  $2^{32}$  个IP地址, 大概是43亿左右. 而TCP/IP协议规定, 每个主机都需要有一个IP地址.

这意味着, 一共只有43亿台主机能接入网络么?

实际上, 由于一些特殊的IP地址的存在, 数量远不足43亿; 另外IP地址并非是按照主机台数来配置的, 而是每一个网卡都需要配置一个或多个IP地址.

CIDR在一定程度上缓解了IP地址不够用的问题(提高了利用率, 减少了浪费, 但是IP地址的绝对上限并没有增加), 仍然不是很够用. 这时候有三种方式来解决:

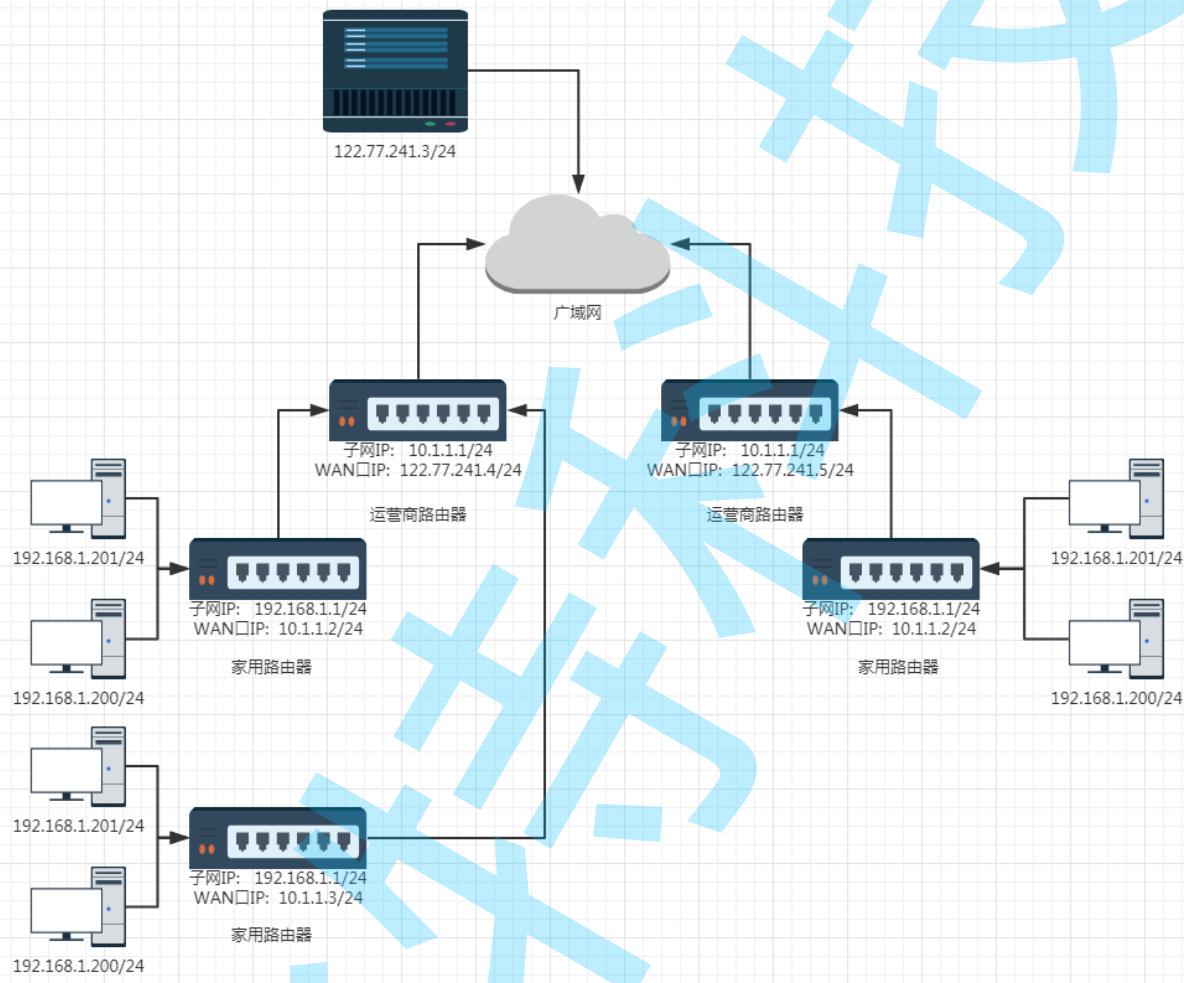
- 动态分配IP地址: 只给接入网络的设备分配IP地址. 因此同一个MAC地址的设备, 每次接入互联网中, 得到的IP地址不一定是相同的;
- NAT技术(后面会重点介绍);
- IPv6: IPv6并不是IPv4的简单升级版. 这是互不相干的两个协议, 彼此并不兼容; IPv6用16字节128位来表示一个IP地址; 但是目前IPv6还没有普及;

## 私有IP地址和公网IP地址

如果一个组织内部组建局域网,IP地址只用于局域网内的通信,而不直接连到Internet上,理论上 使用任意的IP地址都可以,但是RFC 1918规定了用于组建局域网的私有IP地址

- 10.\*,前8位是网络号,共16,777,216个地址
- 172.16.到172.31.,前12位是网络号,共1,048,576个地址
- 192.168.\*,前16位是网络号,共65,536个地址

包含在这个范围中的, 都成为私有IP, 其余的则称为全局IP(或公网IP);



- 一个路由器可以配置两个IP地址, 一个是WAN口IP, 一个是LAN口IP(子网IP).
- 路由器LAN口连接的主机, 都从属于当前这个路由器的子网中.
- 不同的路由器, 子网IP其实都是一样的(通常都是192.168.1.1). 子网内的主机IP地址不能重复. 但是子网之间的IP地址就可以重复了.
- 每一个家用路由器, 其实又作为运营商路由器的子网中的一个节点. 这样的运营商路由器可能会有很多级, 最外层的运营商路由器, WAN口IP就是一个公网IP了.
- 子网内的主机需要和外网进行通信时, 路由器将IP首部中的IP地址进行替换(替换成WAN口IP), 这样逐级替换, 最终数据包中的IP地址成为一个公网IP. 这种技术称为NAT(Network Address Translation, 网络地址转换).
- 如果希望我们自己实现的服务器程序, 能够在公网上被访问到, 就需要把程序部署在一台具有外网IP的服务器上. 这样的服务器可以在阿里云/腾讯云上进行购买.

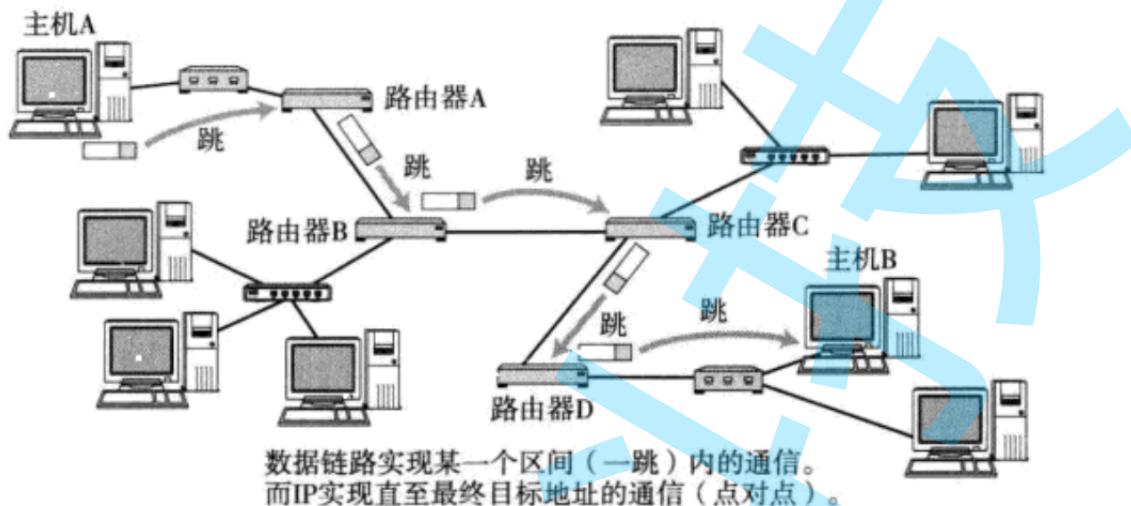
## 路由

在复杂的网络结构中, 找出一条通往终点的路线;

### [唐僧问路例子1]

路由的过程, 就是这样一跳一跳(Hop by Hop) "问路" 的过程.

所谓 "一跳" 就是数据链路层中的一个区间. 具体在以太网中指从源MAC地址到目的MAC地址之间的帧传输区间.

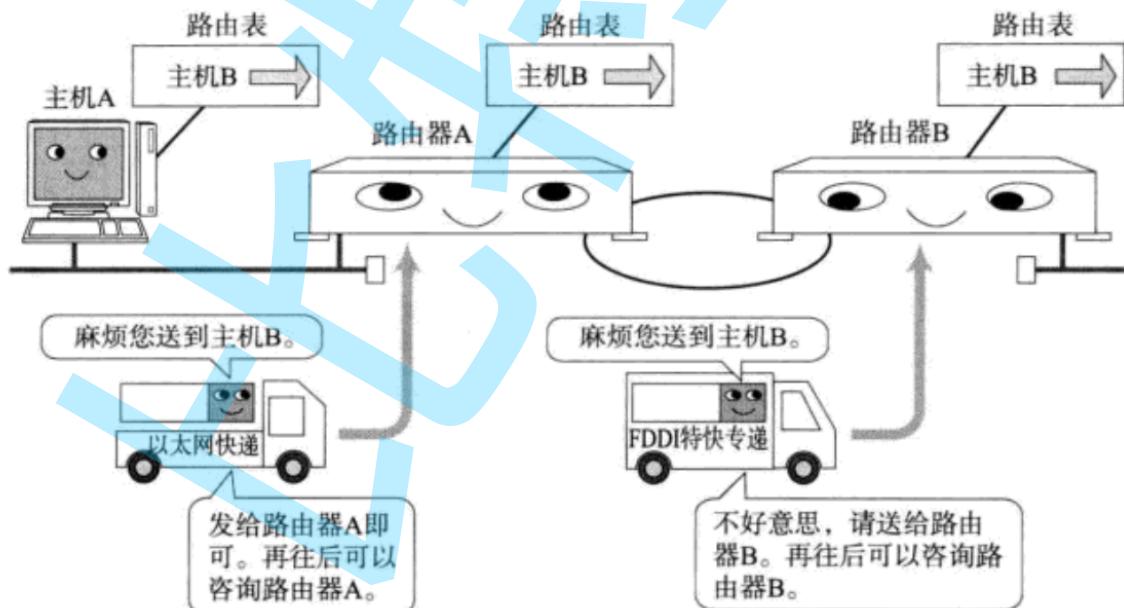


IP数据包的传输过程也和问路一样.

- 当IP数据包, 到达路由器时, 路由器会先查看目的IP;
- 路由器决定这个数据包是能直接发送给目标主机, 还是需要发送给下一个路由器;
- 依次反复, 一直到达目标IP地址;

那么如何判定当前这个数据包该发送到哪里呢? 这个就依靠每个节点内部维护一个路由表;

### [唐僧问路例子2]



- 路由表可以使用route命令查看

- 如果目的IP命中了路由表,就直接转发即可;
- 路由表中的最后一行,主要由下一跳地址和发送接口两部分组成,当目的地址与路由表中其它行都不匹配时,就按缺省路由条目规定的接口发送到下一跳地址。

假设某主机上的网络接口配置和路由表如下:

Destination	Gateway	Genmask	Flags	Metric	Ref
Use Iface	*				
192.168.10.0 0 eth0	*	255.255.255.0	U	0	0
192.168.56.0 0 eth1	*	255.255.255.0	U	0	0
127.0.0.0 0 lo	*	255.0.0.0	U	0	0
default 0 eth0	192.168.10.1	0.0.0.0	UG	0	0

- 这台主机有两个网络接口,一个网络接口连到192.168.10.0/24网络,另一个网络接口连到192.168.56.0/24网络;
- 路由表的Destination是目的网络地址,Genmask是子网掩码,Gateway是下一跳地址,Iface是发送接口,Flags中的U标志表示此条目有效(可以禁用某些条目),G标志表示此条目的下一跳地址是某个路由器的地址,没有G标志的条目表示目的网络地址是与本机接口直接相连的网络,不必经路由器转发;

转发过程例1: 如果要发送的数据包的目的地址是192.168.56.3

- 跟第一行的子网掩码做与运算得到192.168.56.0,与第一行的目的网络地址不符
- 再跟第二行的子网掩码做与运算得到192.168.56.0,正是第二行的目的网络地址,因此从eth1接口发送出去;
- 由于192.168.56.0/24正是与eth1接口直接相连的网络,因此可以直接发到目的主机,不需要经路由器转发;

转发过程例2: 如果要发送的数据包的目的地址是202.10.1.2

- 依次和路由表前几项进行对比,发现都不匹配;
- 按缺省路由条目,从eth0接口发出去,发往192.168.10.1路由器;
- 由192.168.10.1路由器根据它的路由表决定下一跳地址;

## 路由表生成算法(选学)

路由表可以由网络管理员手动维护(静态路由),也可以通过一些算法自动生成(动态路由).  
请同学们课后自己调研一些相关的生成算法,例如距离向量算法,LS算法,Dijkstra算法等.

## 数据链路层

用于两个设备(同一种数据链路节点)之间进行传递.

## 对比理解 "数据链路层" 和 "网络层"

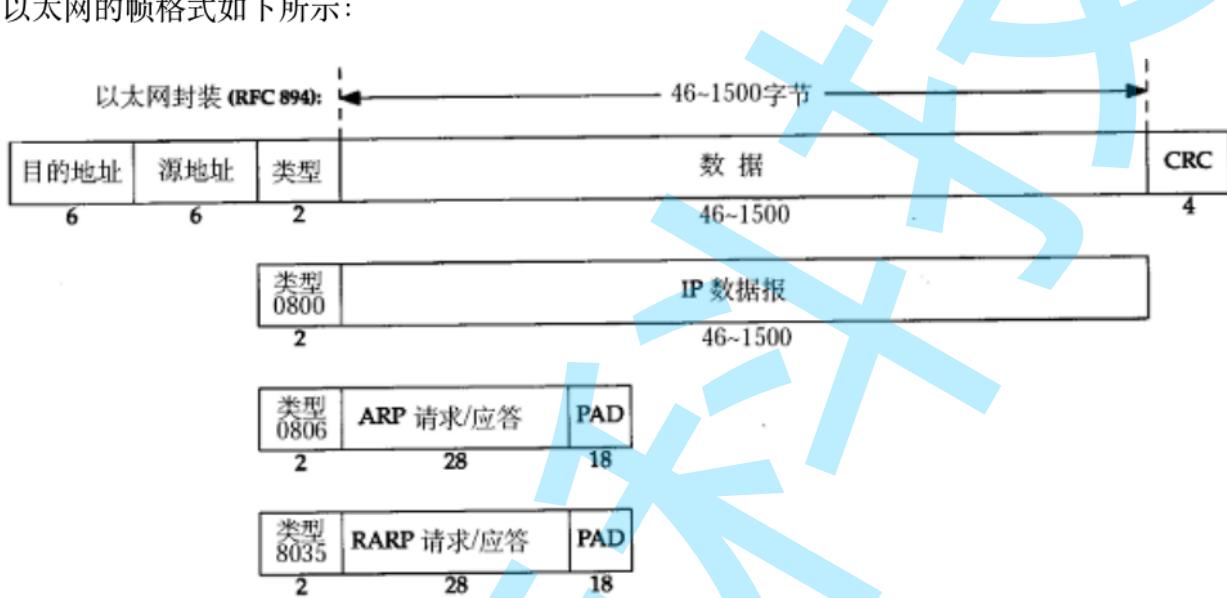
[唐僧例子之白龙马]

## 认识以太网

- "以太网"不是一种具体的网络,而是一种技术标准;既包含了数据链路层的内容,也包含了一些物理层的内容.例如: 规定了网络拓扑结构,访问控制方式,传输速率等;
- 例如以太网中的网线必须使用双绞线; 传输速率有10M, 100M, 1000M等;
- 以太网是当前应用最广泛的局域网技术; 和以太网并列的还有令牌环网, 无线LAN等;

## 以太网帧格式

以太网的帧格式如下所示:



- 源地址和目的地址是指网卡的硬件地址(也叫MAC地址), 长度是48位,是在网卡出厂时固化的;
- 帧协议类型字段有三种值,分别对应IP、ARP、RARP;
- 帧末尾是CRC校验码。

## 认识MAC地址

- MAC地址用来识别数据链路层中相连的节点;
- 长度为48位, 及6个字节. 一般用16进制数字加上冒号的形式来表示(例如: 08:00:27:03:fb:19)
- 在网卡出厂时就确定了, 不能修改. mac地址通常是唯一的(虚拟机中的mac地址不是真实的mac地址, 可能会冲突; 也有些网卡支持用户配置mac地址).

## 对比理解MAC地址和IP地址

还是 [唐僧例子之白龙马]

- IP地址描述的是路途总体的 起点 和 终点;
- MAC地址描述的是路途上的每一个区间的起点和终点;

## 认识MTU

MTU相当于发快递时对包裹尺寸的限制. 这个限制是不同的数据链路对应的物理层, 产生的限制.

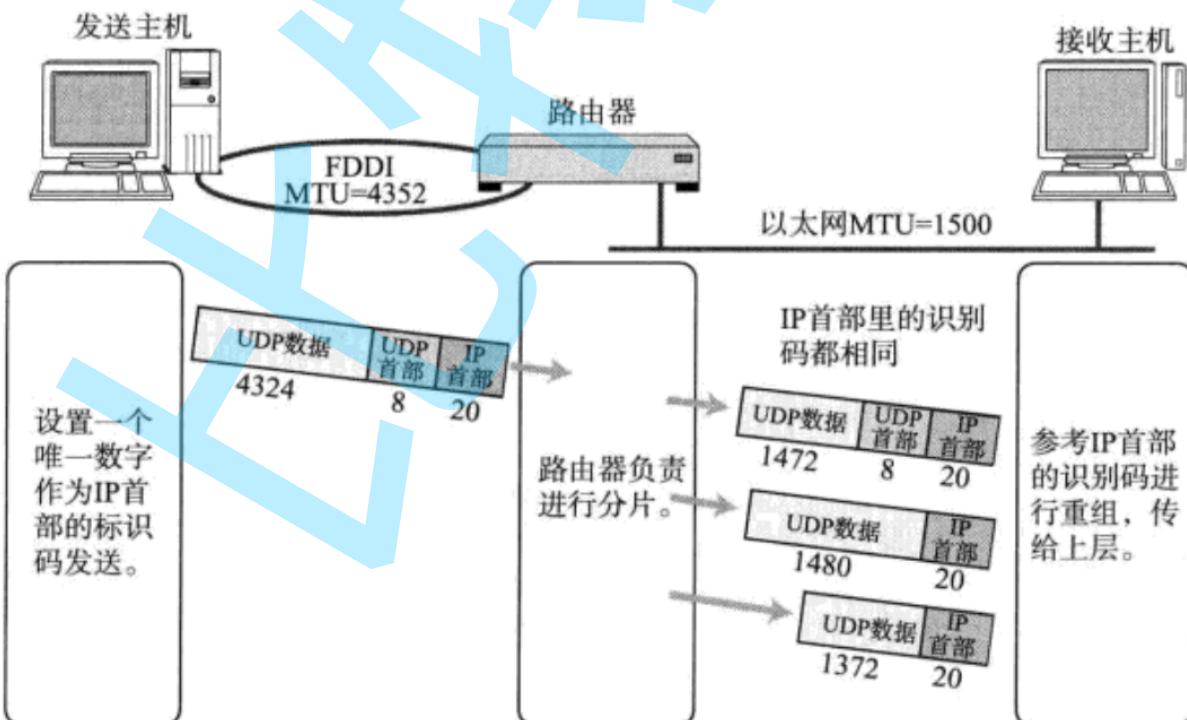
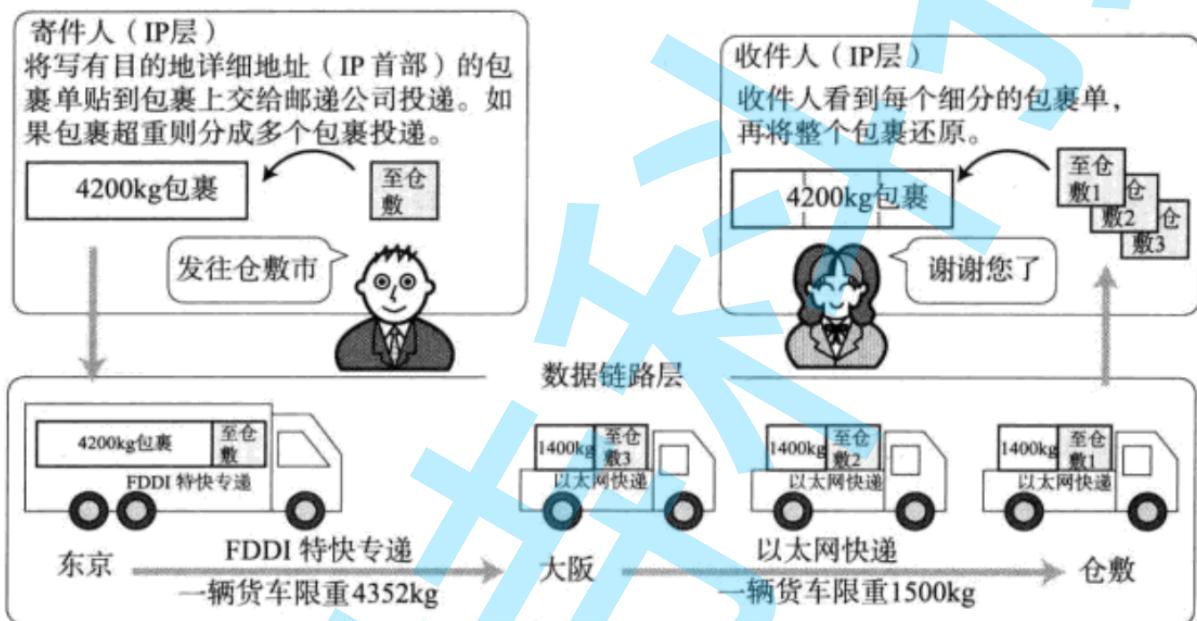
- 以太网帧中的数据长度规定最小46字节,最大1500字节,ARP数据包的长度不够46字节,要在后面补填充位;
- 最大值1500称为以太网的最大传输单元(MTU),不同的网络类型有不同的MTU;

- 如果一个数据包从以太网路由到拨号链路上,数据包长度大于拨号链路的MTU了,则需要对数据包进行分片(fragmentation);
- 不同的数据链路层标准的MTU是不同的;

## MTU对IP协议的影响

由于数据链路层MTU的限制,对于较大的IP数据包要进行分包.

- 将较大的IP包分成多个小包,并给每个小包打上标签;
- 每个小包IP协议头的16位标识(id)都是相同的;
- 每个小包的IP协议头的3位标志字段中,第2位置为0,表示允许分片,第3位来表示结束标记(当前是否是最后一个小包,是的话置为1,否则置为0);
- 到达对端时再将这些小包,会按顺序重组,拼装到一起返回给传输层;
- 一旦这些小包中任意一个小包丢失,接收端的重组就会失败.但是IP层不会负责重新传输数据;



## MTU对UDP协议的影响

让我们回顾一下UDP协议:

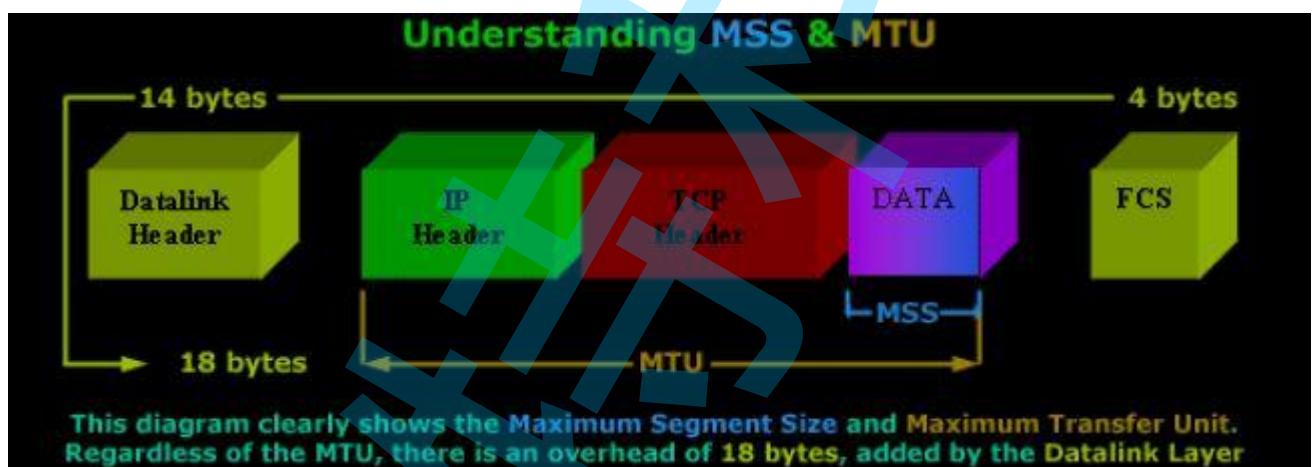
- 一旦UDP携带的数据超过1472(1500 - 20(IP首部) - 8(UDP首部)), 那么就会在网络层分成多个IP数据报.
- 这多个IP数据报有任何一个丢失, 都会引起接收端网络层重组失败. 那么这就意味着, 如果UDP数据报在网络层被分片, 整个数据被丢失的概率就大大增加了.

## MTU对于TCP协议的影响

让我们再回顾一下TCP协议:

- TCP的一个数据报也不能无限大, 还是受制于MTU. TCP的单个数据报的最大消息长度, 称为MSS(Max Segment Size);
- TCP在建立连接的过程中, 通信双方会进行MSS协商.
- 最理想的情况下, MSS的值正好是在IP不会被分片处理的最大长度(这个长度仍然是受制于数据链路层的MTU).
- 双方在发送SYN的时候会在TCP头部写入自己能支持的MSS值.
- 然后双方得知对方的MSS值之后, 选择较小的作为最终MSS.
- MSS的值就是在TCP首部的40字节变长选项中(kind=2);

MSS和MTU的关系



## 查看硬件地址和MTU

```
[tangzhong@tz ~]$ ifconfig  
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
        inet 192.168.1.108 netmask 255.255.255.0 broadcast 192.168.1.255  
        inet6 fe80::fa5f:d814:637d:a6a3 prefixlen 64 scopeid 0x20<link>  
          ether 08:00:27:03:fb:19 txqueuelen 1000 (Ethernet)  
            RX packets 400 bytes 37722 (36.8 KiB)  
            RX errors 0 dropped 0 overruns 0 frame 0  
            TX packets 269 bytes 37356 (36.4 KiB)  
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

使用ifconfig命令, 即可查看ip地址, mac地址, 和MTU;

# ARP协议

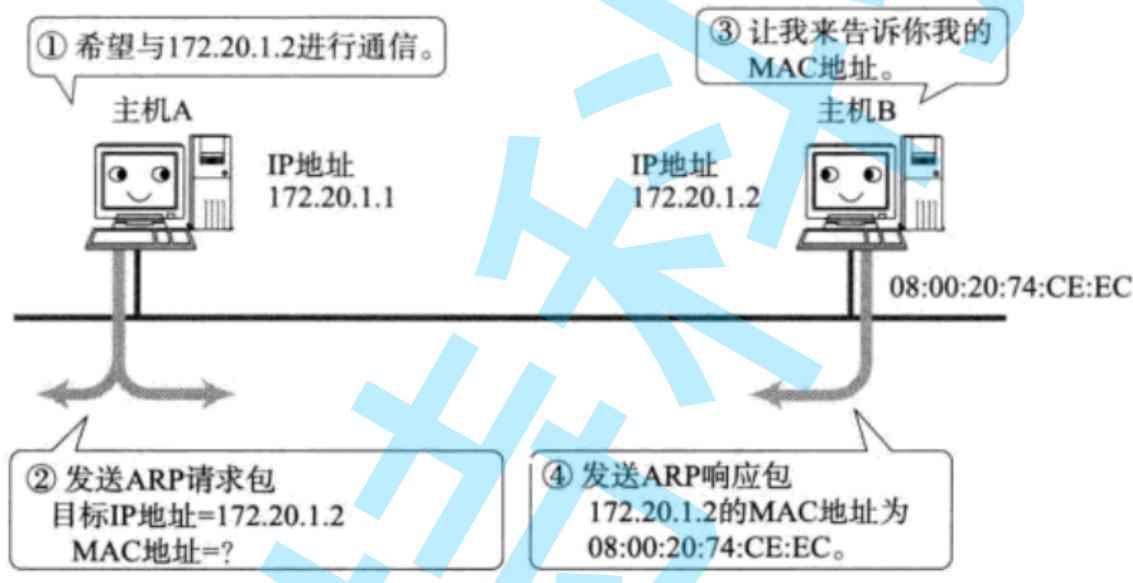
虽然我们在这里介绍ARP协议，但是需要强调，ARP不是一个单纯的数据链路层的协议，而是一个介于数据链路层和网络层之间的协议；

## ARP协议的作用

ARP协议建立了主机 IP地址 和 MAC地址 的映射关系。

- 在网络通讯时，源主机的应用程序知道目的主机的IP地址和端口号，却不知道目的主机的硬件地址；
- 数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃；
- 因此在通讯前必须获得目的主机的硬件地址；

## ARP协议的工作流程



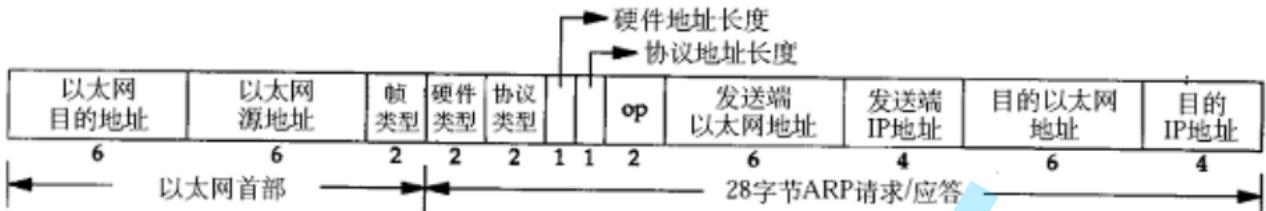
- 源主机发出ARP请求，询问“IP地址是192.168.0.1的主机的硬件地址是多少”，并将这个请求广播到本地网段（以太网帧首部的硬件地址填FF:FF:FF:FF:FF表示广播）；
- 目的主机接收到广播的ARP请求，发现其中的IP地址与本机相符，则发送一个ARP应答数据包给源主机，将自己的硬件地址填写在应答包中；
- 每台主机都维护一个ARP缓存表，可以用arp -a命令查看。缓存表中的表项有过期时间（一般为20分钟），如果20分钟内没有再次使用某个表项，则该表项失效，下次还要发ARP请求来获得目的主机的硬件地址

```
[tangzhong@tz ~]$ arp -a
? (192.168.1.107) at e4:f8:9c:be:af:41 [ether] on enp0s3
gateway (192.168.1.1) at 10:bd:18:08:af:62 [ether] on enp0s3
```

想一想，为什么要有缓存表？为什么表项要有过期时间而不是一直有效？

再想一想，结合我们刚才讲的工作流程，ARP的数据报应该是一个什么样的格式？

## ARP数据报的格式



- 注意到源MAC地址、目的MAC地址在以太网首部和ARP请求中各出现一次,对于链路层为以太网的情况是多余的,但如果链路层是其它类型的网络则有可能是必要的。
- 硬件类型指链路层网络类型,1为以太网;
- 协议类型指要转换的地址类型,0x0800为IP地址;
- 硬件地址长度对于以太网地址为6字节;
- 协议地址长度对于IP地址为4字节;
- op字段为1表示ARP请求,op字段为2表示ARP应答。

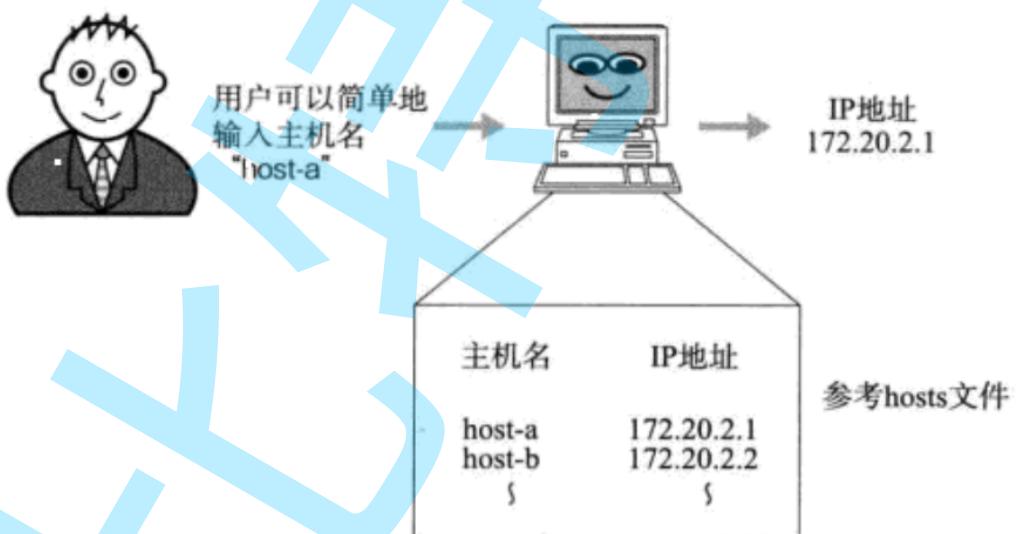
## 其他重要协议或技术

### DNS(Domain Name System)

DNS是一整套从域名映射到IP的系统

#### DNS背景

TCP/IP中使用IP地址和端口号来确定网络上的一台主机的一个程序.但是IP地址不方便记忆.  
于是人们发明了一种叫主机名的东西,是一个字符串,并且使用hosts文件来描述主机名和IP地址的关系.



最初,通过互连网信息中心(SRI-NIC)来管理这个hosts文件的.

- 如果一个新计算机要接入网络,或者某个计算机IP变更,都需要到信息中心申请变更hosts文件.
- 其他计算机也需要定期下载更新新版本的hosts文件才能正确上网.

这样就太麻烦了,于是产生了DNS系统.

- 一个组织的系统管理机构,维护系统内的每个主机的IP和主机名的对应关系.

- 如果新计算机接入网络, 将这个信息注册到数据库中;
- 用户输入域名的时候, 会自动查询DNS服务器, 由DNS服务器检索数据库, 得到对应的IP地址.

至今, 我们的计算机上仍然保留了hosts文件. 在域名解析的过程中仍然会优先查找hosts文件的内容.

```
cat /etc/hosts
```

## 域名简介

主域名是用来识别主机名称和主机所属的组织机构的一种分层结构的名称.

```
www.baidu.com
```

域名使用 . 连接

- com: 一级域名. 表示这是一个企业域名. 同级的还有 "net"(网络提供商), "org"(非盈利组织) 等.
- baidu: 二级域名, 公司名.
- www: 只是一种习惯用法. 之前人们在使用域名时, 往往命名成类似于ftp.xxx.xxx/[www.xxx.xxx](#)这样的格式, 来表示主机支持的协议.

## 域名解析过程(选学)

大家自行搜索资料. 可以参考 <<图解TCP/IP>> 相关章节

## 使用 dig 工具分析 DNS 过程

安装 dig 工具

```
yum install bind-utils
```

之后就可以使用 dig 指令查看域名解析过程了.

```
dig www.baidu.com
```

结果形如

```
; <>> DiG 9.9.4-RedHat-9.9.4-61.el7_5.1 <>> www.baidu.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 41628
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 0
;;
;; QUESTION SECTION:
;www.baidu.com.      IN  A
;;
;; ANSWER SECTION:
www.baidu.com.      1057     IN  CNAME   www.a.shifen.com.
www.a.shifen.com.    40      IN  A     115.239.210.27
```

```
www.a.shifen.com. 40 IN A 115.239.211.112
```

```
; ; Query time: 0 msec  
; ; SERVER: 100.100.2.136#53(100.100.2.136)  
; ; WHEN: Wed Sep 26 00:05:25 CST 2018  
; ; MSG SIZE rcvd: 90
```

## 结果解释

1. 开头位置是 dig 指令的版本号
2. 第二部分是服务器返回的详情, 重要的是 status 参数, NOERROR 表示查询成功
3. QUESTION SECTION 表示要查询的域名是什么
4. ANSWER SECTION 表示查询结果是什么. 这个结果先将 [www.baidu.com](#) 查询成了 [www.a.shifen.com](#), 再将 [www.a.shifen.com](#) 查询成了两个 ip 地址.
5. 最下面是一些结果统计, 包含查询时间和 DNS 服务器的地址等.

更多 dig 的使用方法, 参见

[https://www.imooc.com/article/26971?block\\_id=tuijian\\_wz](https://www.imooc.com/article/26971?block_id=tuijian_wz)

## 浏览器中输入url后, 发生的事情. (作业)

这是一个经典的面试题. 没有固定答案, 越详细越好. 可以参考:

[浏览器中输入url后发生的事情](#)

## ICMP协议

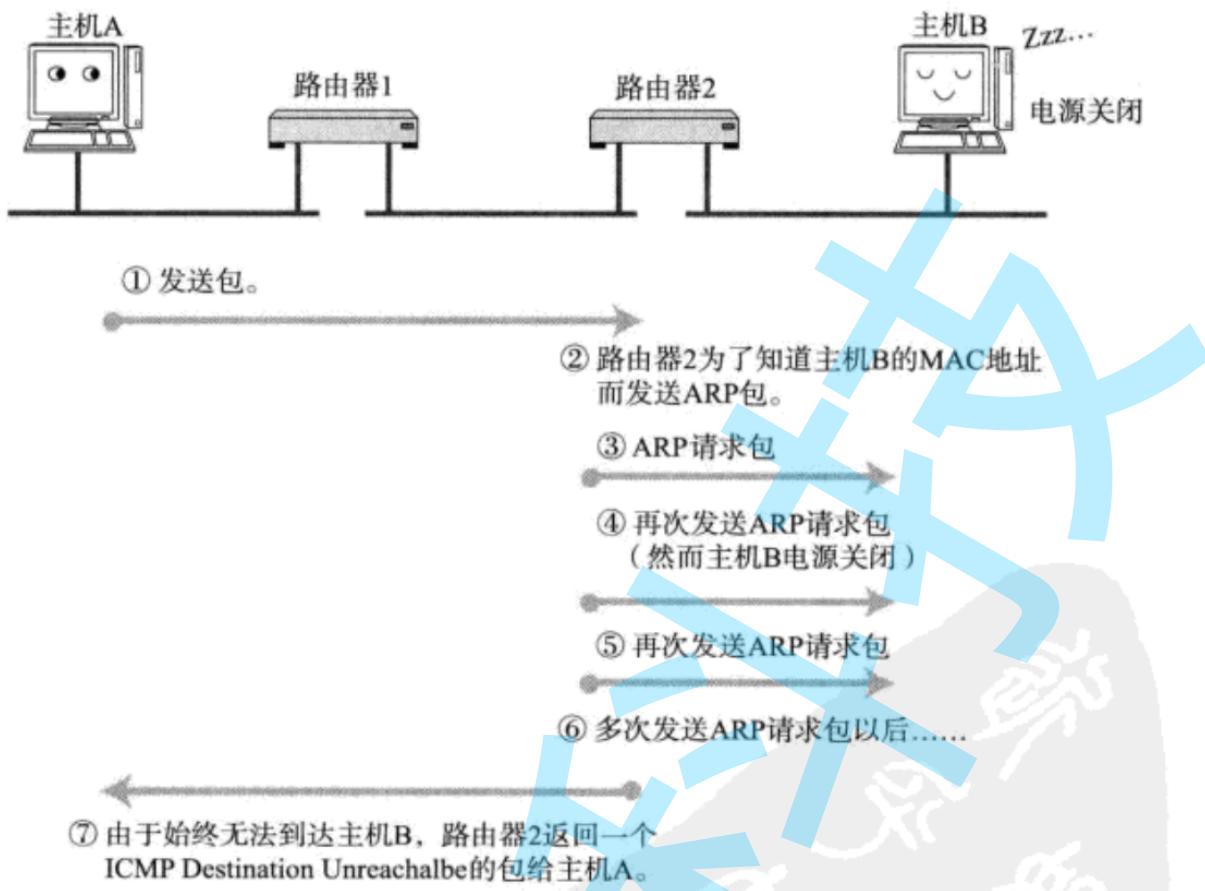
ICMP协议是一个 **网络层协议**

一个新搭建好的网络, 往往需要先进行一个简单的测试, 来验证网络是否畅通; 但是IP协议并不提供可靠传输. 如果丢包了, IP协议并不能通知传输层是否丢包以及丢包的原因.

## ICMP功能

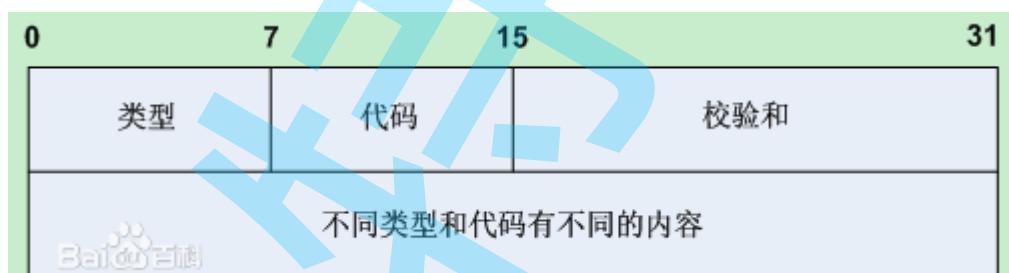
ICMP正是提供这种功能的协议; ICMP主要功能包括:

- 确认IP包是否成功到达目标地址.
- 通知在发送过程中IP包被丢弃的原因.
- ICMP也是基于IP协议工作的. 但是它并不是传输层的功能, 因此人们仍然把它归结为网络层协议;
- ICMP只能搭配IPv4使用. 如果是IPv6的情况下, 需要是用ICMPv6;



## ICMP的报文格式 (选学)

关于报文格式, 我们并不打算重点关注, 大家稍微有个了解即可.



ICMP大概分为两类报文:

- 一类是通知出错原因
- 一类是用于诊断查询

类型 (十进制数)	内 容
0	回送应答 (Echo Reply)
3	目标不可达 (Destination Unreachable)
4	原点抑制 (Source Quench)
5	重定向或改变路由 (Redirect)
8	回送请求 (Echo Request)
9	路由器公告 (Router Advertisement)
10	路由器请求 (Router Solicitation)
11	超时 (Time Exceeded)
17	地址子网请求 (Address Mask Request)
18	地址子网应答 (Address Mask Reply)

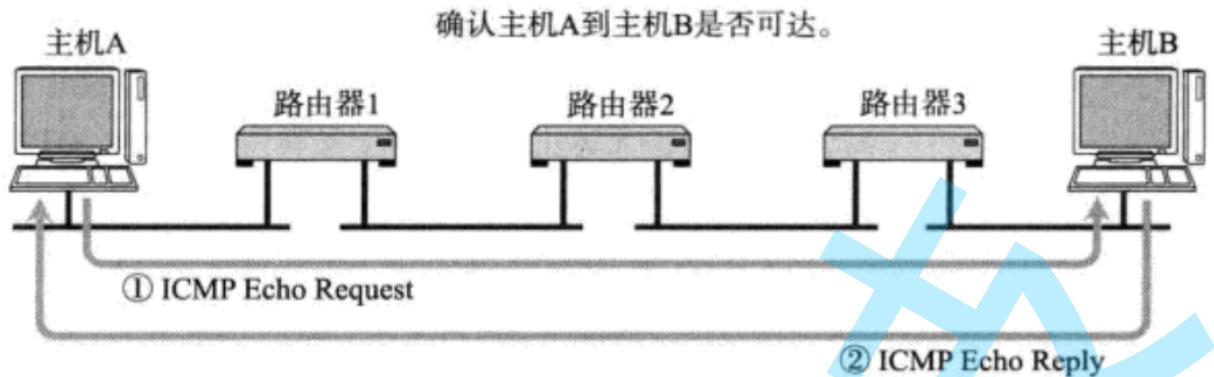
## ping命令

```
C:\Users\HGtz>ping www.baidu.com

正在 Ping www.a.shifen.com [61.135.169.121] 具有 32 字节的数据:
来自 61.135.169.121 的回复: 字节=32 时间=61ms TTL=52
来自 61.135.169.121 的回复: 字节=32 时间=28ms TTL=52
来自 61.135.169.121 的回复: 字节=32 时间=66ms TTL=52
来自 61.135.169.121 的回复: 字节=32 时间=44ms TTL=52

61.135.169.121 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 28ms, 最长 = 66ms, 平均 = 49ms
```

- 注意, 此处 ping 的是域名, 而不是url! 一个域名可以通过DNS解析成IP地址.
- ping命令不仅能验证网络的连通性, 同时也会统计响应时间和TTL(IP包中的Time To Live, 生存周期).
- ping命令会先发送一个 ICMP Echo Request给对端;
- 对端接收到之后, 会返回一个ICMP Echo Reply;



## 一个值得注意的坑

有些面试官可能会问: telnet是23端口, ssh是22端口, 那么ping是什么端口?

千万注意!!! 这是面试官的圈套



ping命令基于ICMP, 是在网络层. 而端口号, 是传输层的内容. 在ICMP中根本就不关注端口号这样的信息.

## traceroute命令

也是基于ICMP协议实现, 能够打印出可执行程序主机, 一直到目标主机之前经历多少路由器.

```
[tangzhong@tz ~]$ traceroute www.baidu.com
traceroute to www.baidu.com (61.135.169.121), 30 hops max, 60 byte packets
 1 * * *
 2 10.254.1.13 (10.254.1.13)  24.307 ms  32.617 ms  32.634 ms
 3 10.254.1.69 (10.254.1.69)  32.600 ms  32.512 ms  39.239 ms
 4 10.254.1.53 (10.254.1.53)  32.475 ms  32.459 ms  32.402 ms
 5 123.139.1.193 (123.139.1.193)  32.355 ms  32.325 ms  32.307 ms
 6 221.11.0.1 (221.11.0.1)  32.301 ms  23.785 ms  28.480 ms
 7 221.11.0.97 (221.11.0.97)  61.396 ms  221.11.0.85 (221.11.0.85)  56.582 ms  63.129 ms
 8 219.158.112.17 (219.158.112.17)  53.336 ms  219.158.112.21 (219.158.112.21)  53.269 ms  53.194 ms
 9 124.65.194.158 (124.65.194.158)  53.178 ms  53.149 ms  53.598 ms
10 124.65.58.54 (124.65.58.54)  56.066 ms  124.65.58.62 (124.65.58.62)  47.815 ms  124.65.59.114 (124.65.59.114)
38.873 ms
11 202.106.48.18 (202.106.48.18)  53.095 ms  61.49.168.110 (61.49.168.110)  57.682 ms  123.125.248.98 (123.125.248
.98)  46.452 ms
12 * * *
13 * * *
```

# NAT技术

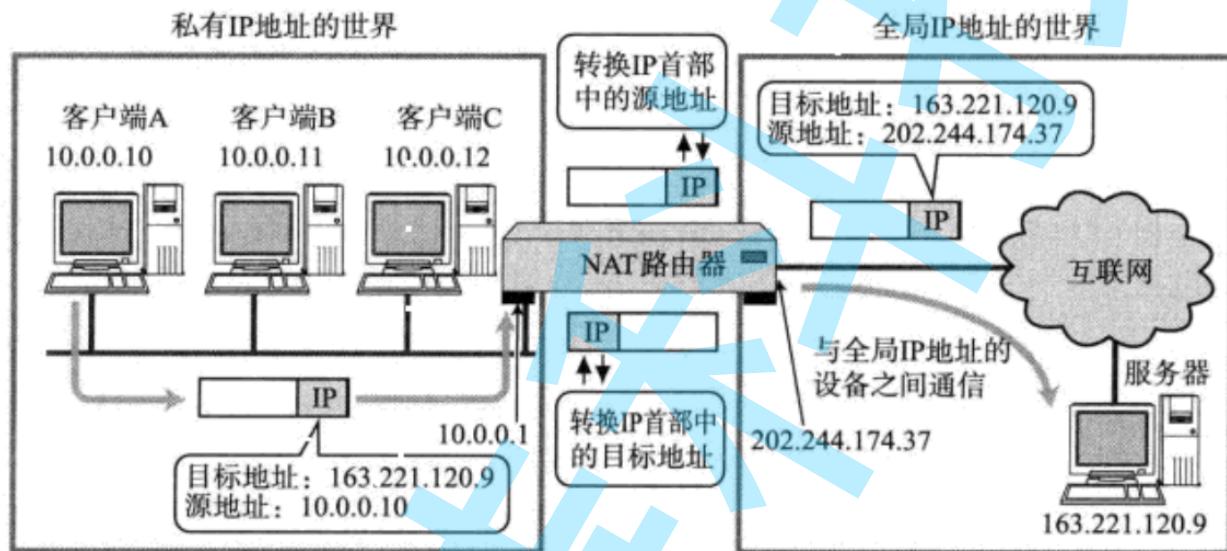
## NAT技术背景

之前我们讨论了, IPv4协议中, IP地址数量不足的问题

NAT技术当前解决IP地址不够用的主要手段, 是路由器的一个重要功能;

- NAT能够将私有IP对外通信时转为全局IP. 也就是就是一种将私有IP和全局IP相互转化的技术方法;
- 很多学校, 家庭, 公司内部采用每个终端设置私有IP, 而在路由器或必要的服务器上设置全局IP;
- 全局IP要求唯一, 但是私有IP不需要; 在不同的局域网中出现相同的私有IP是完全不影响的;

## NAT IP转换过程

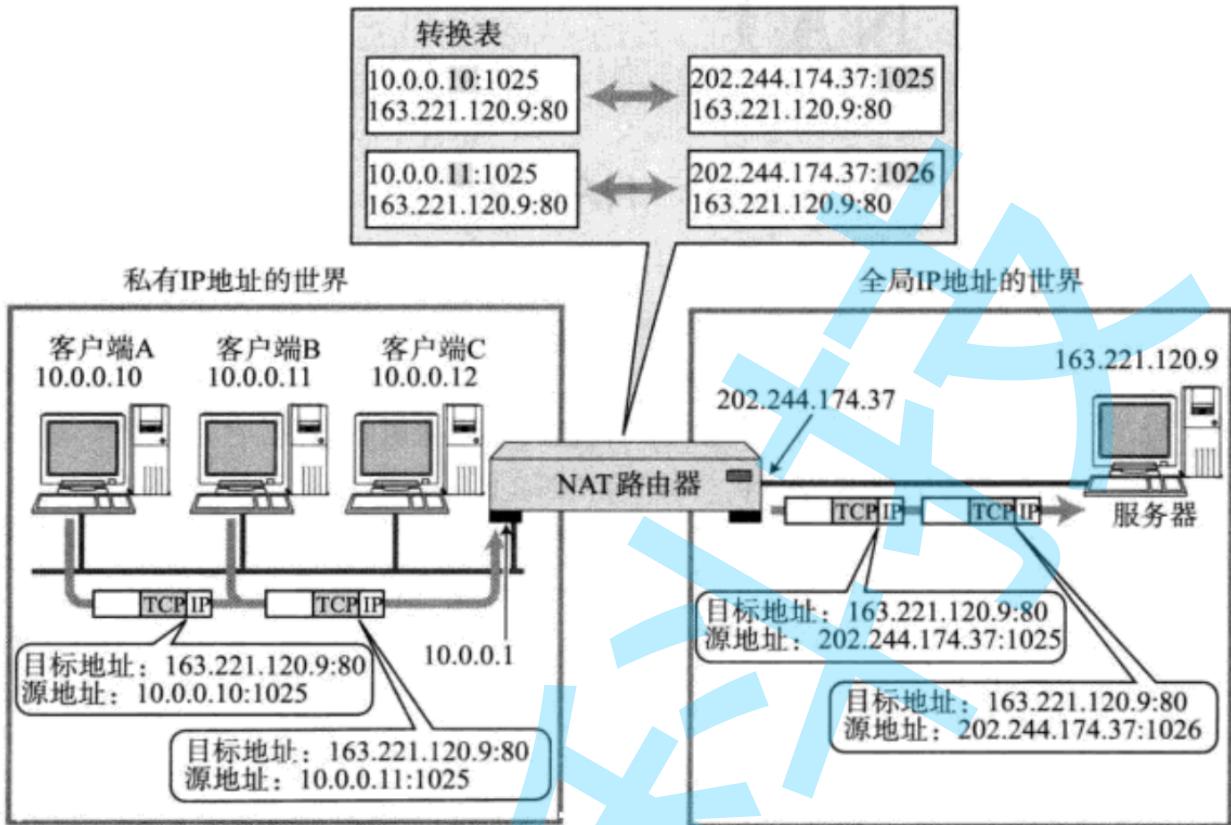


- NAT路由器将源地址从10.0.0.10替换成全局的IP 202.244.174.37;
- NAT路由器收到外部的数据时, 又会把目标IP从202.244.174.37替换成10.0.0.10;
- 在NAT路由器内部, 有一张自动生成的, 用于地址转换的表;
- 当10.0.0.10第一次向163.221.120.9发送数据时就会生成表中的映射关系;

## NAPT

那么问题来了, 如果局域网内, 有多个主机都访问同一个外网服务器, 那么对于服务器返回的数据中, 目的IP都是相同的. 那么NAT路由器如何判定将这个数据包转发给哪个局域网的主机?

这时候NAPT来解决这个问题了. 使用IP+port来建立这个关联关系



这种关联关系也是由NAT路由器自动维护的。例如在TCP的情况下，建立连接时，就会生成这个表项；在断开连接后，就会删除这个表项。

## NAT技术的缺陷

由于NAT依赖这个转换表，所以有诸多限制：

- 无法从NAT外部向内部服务器建立连接；
- 转换表的生成和销毁都需要额外开销；
- 通信过程中一旦NAT设备异常，即使存在热备，所有的TCP连接也都会断开；

课外调研：NAT穿越

## NAT和代理服务器

路由器往往都具备NAT设备的功能，通过NAT设备进行中转，完成子网设备和其他子网设备的通信过程。

代理服务器看起来和NAT设备有一点像。客户端像代理服务器发送请求，代理服务器将请求转发给真正要请求的服务器；服务器返回结果后，代理服务器又把结果回传给客户端。

那么NAT和代理服务器的区别有哪些呢？

- 从应用上讲，NAT设备是网络基础设备之一，解决的是IP不足的问题。代理服务器则是更贴近具体应用，比如通过代理服务器进行翻墙，另外像迅游这样的加速器，也是使用代理服务器。
- 从底层实现上讲，NAT是工作在网络层，直接对IP地址进行替换。代理服务器往往工作在应用层。
- 从使用范围上讲，NAT一般在局域网的出口部署，代理服务器可以在局域网做，也可以在广域网做，也可以跨网。

- 从部署位置上看, NAT一般集成在防火墙, 路由器等硬件设备上, 代理服务器则是一个软件程序, 需要部署在服务器上.

代理服务器是一种应用比较广的技术.

- 翻墙: 广域网中的代理.
- 负载均衡: 局域网中的代理.

代理服务器又分为正向代理和反向代理.

## 代购例子

花王尿不湿是一个很经典的尿不湿品牌, 产自日本.

我自己去日本买尿不湿比较不方便, 但是可以让我在日本工作的表姐去超市买了快递给我. 此时超市看到的买家是我表姐, 我的表姐就是 "正向代理";

后来找我表姐买尿不湿的人太多了, 我表姐觉得天天去超市太麻烦, 干脆去超市买了一大批尿不湿屯在家里, 如果有人来找她代购, 就直接把屯在家里的货发出去, 而不必再去超市. 此时我表姐就是 "反向代理"

正向代理用于请求的转发(例如借助代理绕过反爬虫).

反向代理往往作为一个缓存.

## 总结

## 数据链路层

- 数据链路层的作用: 两个设备(同一种数据链路节点)之间进行传递数据
- 以太网是一种技术标准; 既包含了数据链路层的内容, 也包含了一些物理层的内容. 例如: 规定了网络拓扑结构, 访问控制方式, 传输速率等;
- 以太网帧格式
- 理解mac地址
- 理解arp协议
- 理解MTU

## 网络层

- 网络层的作用: 在复杂的网络环境中确定一个合适的路径.
- 理解IP地址, 理解IP地址和MAC地址的区别.
- 理解IP协议格式.
- 了解网段划分方法
- 理解如何解决IP数目不足的问题, 掌握网段划分的两种方案. 理解私有IP和公网IP
- 理解网络层的IP地址路由过程. 理解一个数据包如何跨越网段到达最终目的地.
- 理解IP数据包分包的原因.
- 了解ICMP协议.
- 了解NAT设备的工作原理.

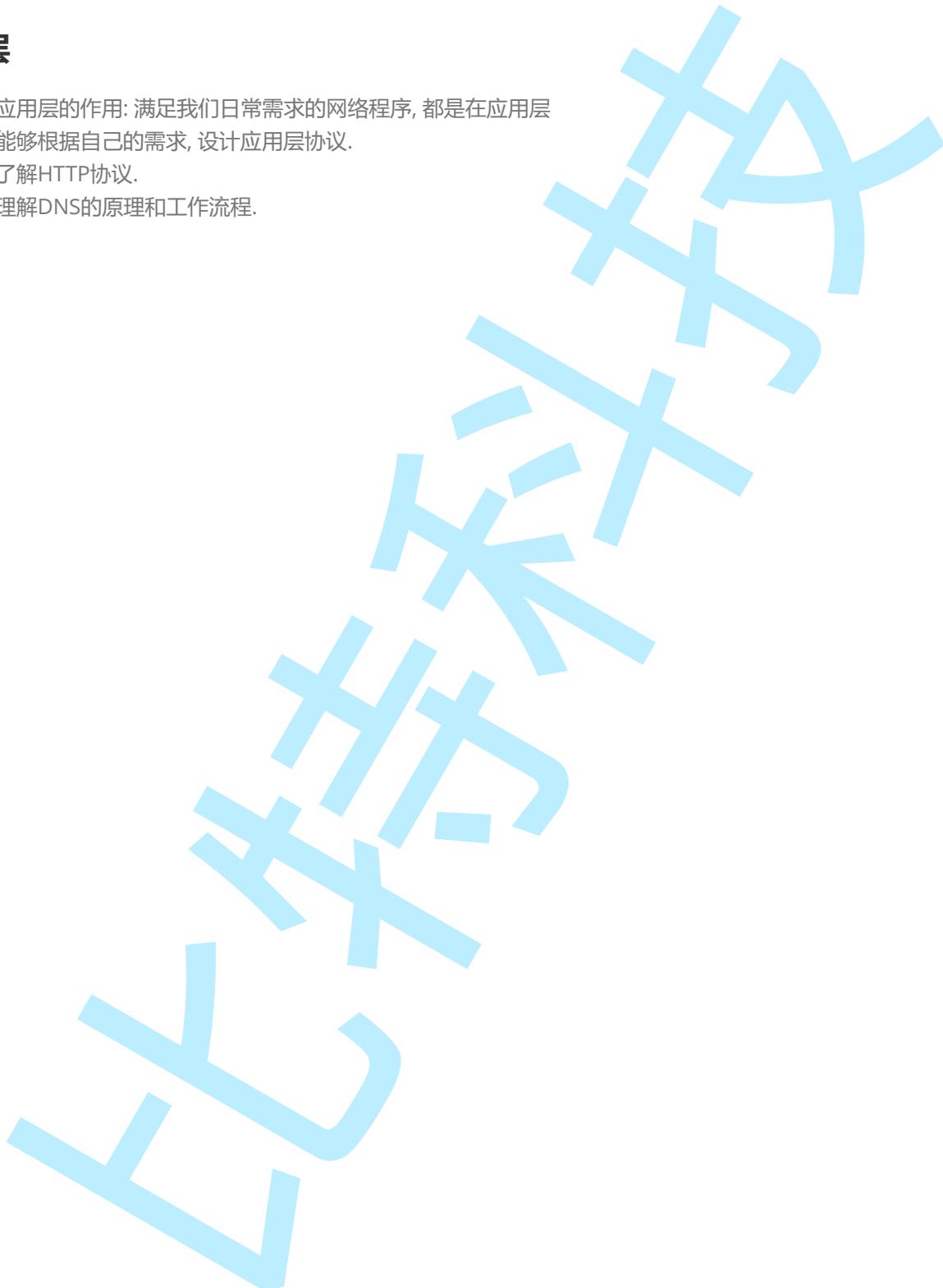
## 传输层

- 传输层的作用: 负责数据能够从发送端传输接收端.
- 理解端口号的概念.
- 认识UDP协议, 了解UDP协议的特点.

- 认识TCP协议, 理解TCP协议的可靠性. 理解TCP协议的状态转化.
- 掌握TCP的连接管理, 确认应答, 超时重传, 滑动窗口, 流量控制, 拥塞控制, 延迟应答, 搞带应答特性.
- 理解TCP面向字节流, 理解粘包问题和解决方案.
- 能够基于UDP实现可靠传输.
- 理解MTU对UDP/TCP的影响.

## 应用层

- 应用层的作用: 满足我们日常需求的网络程序, 都是在应用层
- 能够根据自己的需求, 设计应用层协议.
- 了解HTTP协议.
- 理解DNS的原理和工作流程.



# 高级IO

## 本节重点

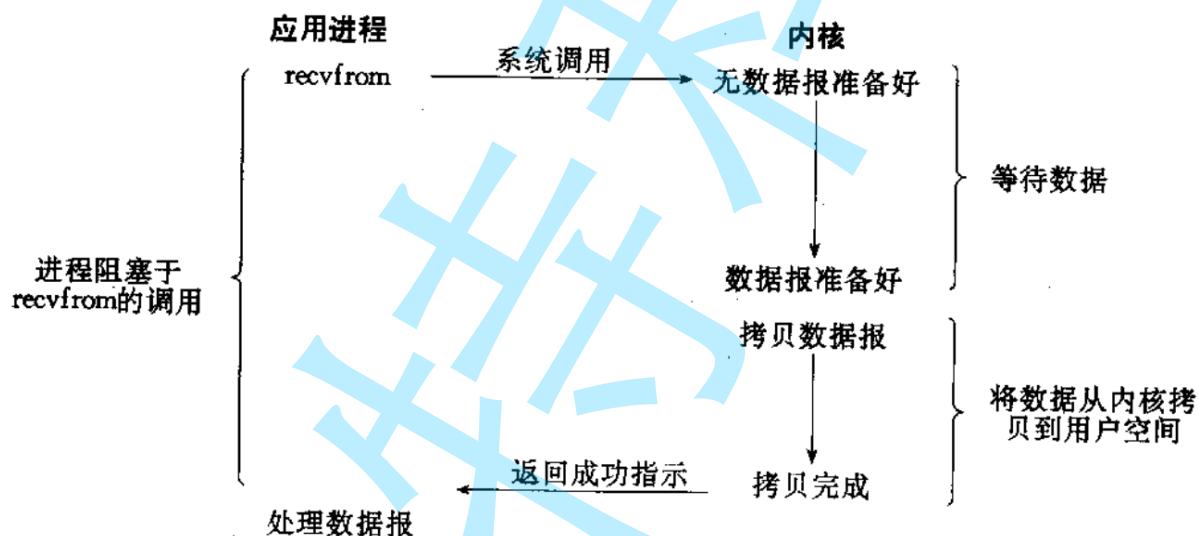
- 理解五种IO模型的基本概念, 重点是IO多路转接.
- 掌握select编程模型, 能够实现select版本的TCP服务器.
- 掌握poll编程模型, 能够实现poll版本的TCP服务器.
- 掌握epoll编程模型, 能够实现epoll版本的TCP服务器.
- 理解epoll的LT模式和ET模式.
- 理解select和epoll的优缺点对比.

## 五种IO模型

### [钓鱼例子]

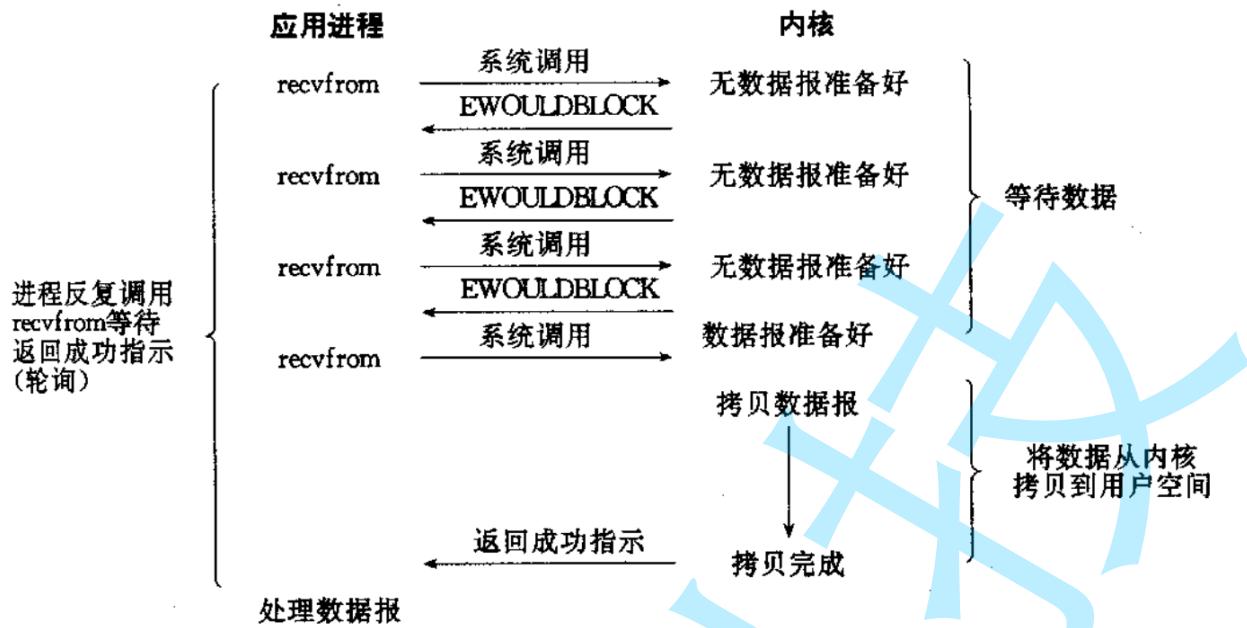
- 阻塞IO: 在内核将数据准备好之前, 系统调用会一直等待. 所有的套接字, 默认都是阻塞方式.

阻塞IO是最常见的IO模型.

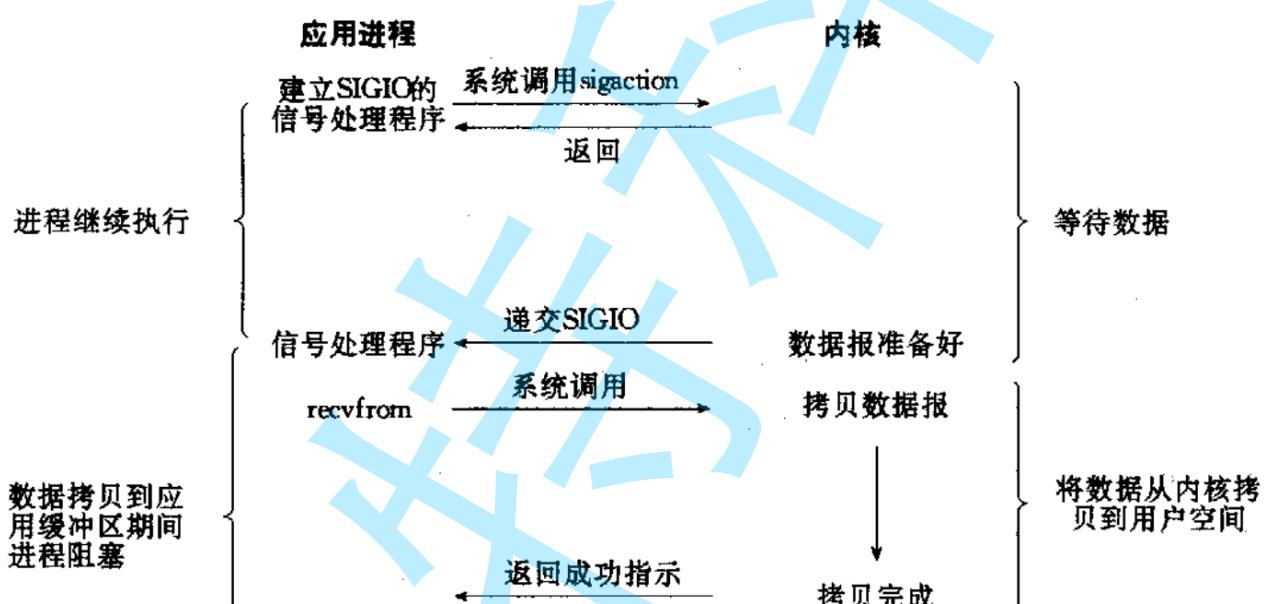


- 非阻塞IO: 如果内核还未将数据准备好, 系统调用仍然会直接返回, 并且返回`EWOULDBLOCK`错误码.

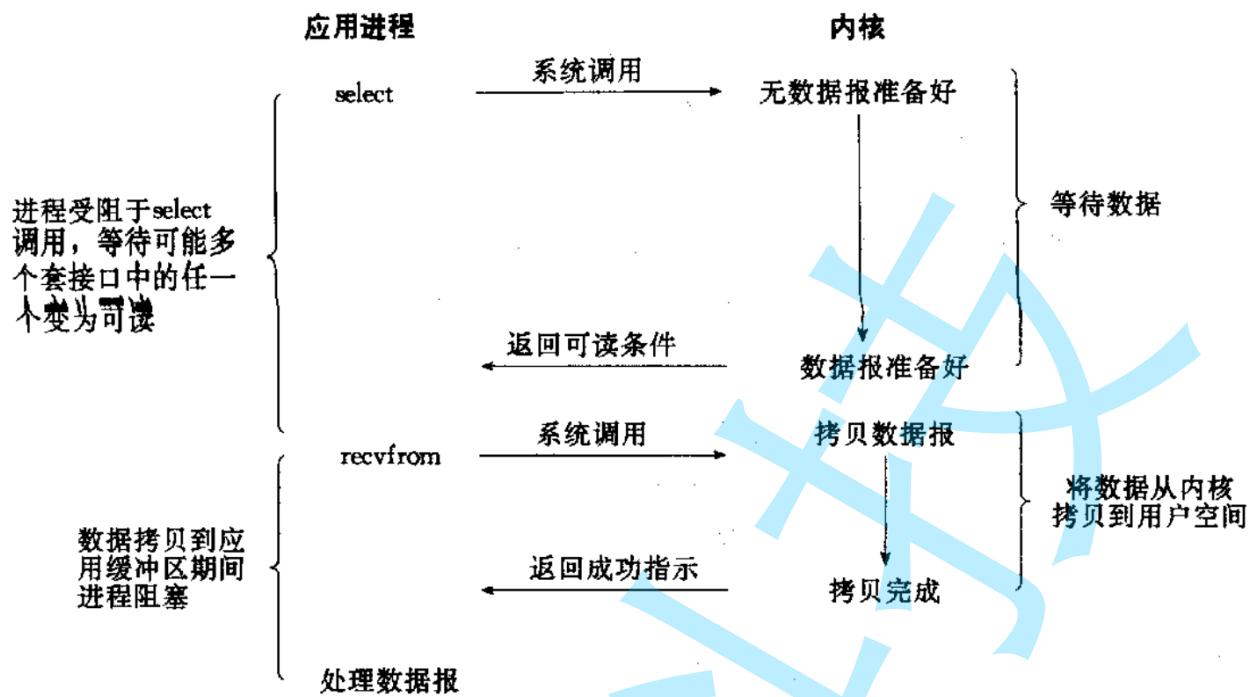
非阻塞IO往往需要程序员循环的方式反复尝试读写文件描述符, 这个过程称为**轮询**. 这对CPU来说是较大的浪费, 一般只有特定场景下才使用.



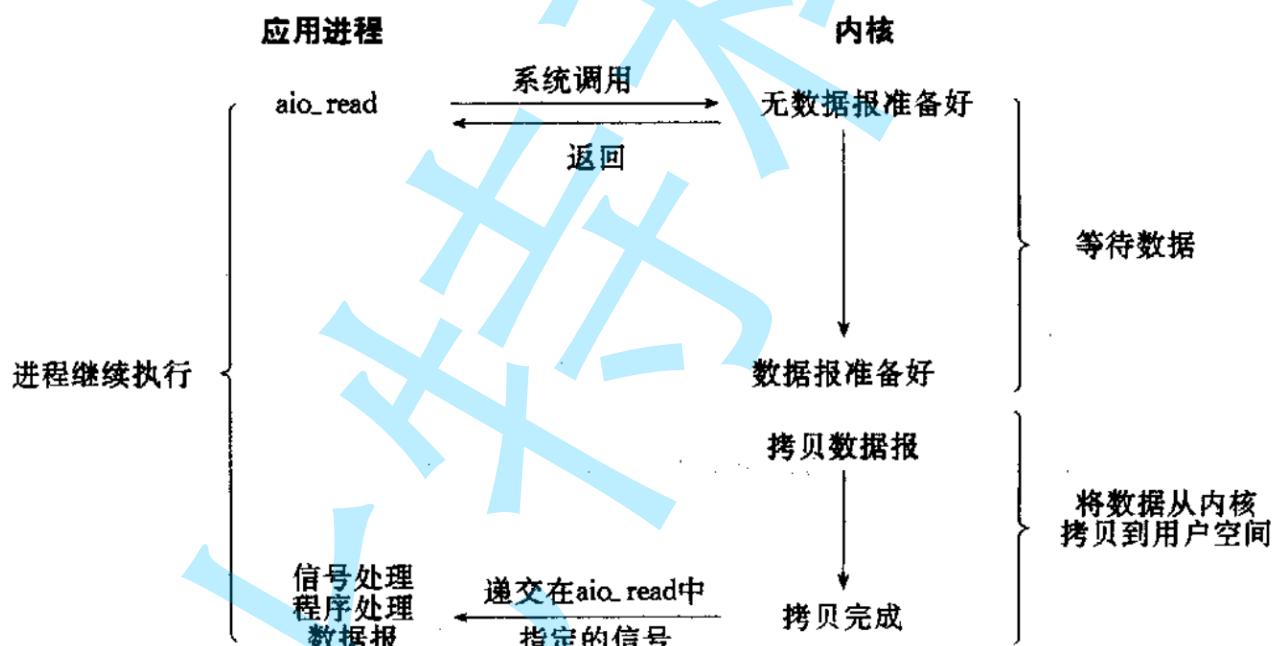
- 信号驱动IO: 内核将数据准备好的时候, 使用SIGIO信号通知应用程序进行IO操作.



- IO多路转接: 虽然从流程图上看起来和阻塞IO类似. 实际上最核心在于IO多路转接能够同时等待多个文件描述符的就绪状态.



- 异步IO: 由内核在数据拷贝完成时, 通知应用程序(而信号驱动是告诉应用程序何时可以开始拷贝数据).



## 小结

- 任何IO过程中, 都包含两个步骤. 第一是等待, 第二是拷贝. 而且在实际的应用场景中, 等待消耗的时间往往都远远高于拷贝的时间. 让IO更高效, 最核心的办法就是让等待的时间尽量少.

## 高级IO重要概念

在这里, 我们要强调几个概念

# 同步通信 vs 异步通信(synchronous communication/ asynchronous communication)

同步和异步关注的是消息通信机制.

- 所谓同步，就是在发出一个调用时，在没有得到结果之前，该调用就不返回. 但是一旦调用返回，就得  
到返回值了；换句话说，就是由调用者主动等待这个调用的结果；
- 异步则是相反，调用在发出之后，这个调用就直接返回了，所以没有返回结果；换句话说，当一个异步  
过程调用发出后，调用者不会立刻得到结果；而是在调用发出后，被调用者通过状态、通知来通知调用  
者，或通过回调函数处理这个调用。

另外，我们回忆在讲多进程多线程的时候，也提到同步和互斥。这里的同步通信和进程之间的同步是完全不想干的概念。

- 进程/线程同步也是进程/线程之间直接的制约关系
- 是为完成某种任务而建立的两个或多个线程，这个线程需要在某些位置上协调他们的工作次序而等待、  
传递信息所产生的制约关系. 尤其是在访问临界资源的时候。

同学们以后在看到 "同步" 这个词，一定要先搞清楚大背景是什么. 这个同步，是同步通信异步通信的同步，还是同步  
与互斥的同步。

## 阻塞 vs 非阻塞

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。

- 阻塞调用是指调用结果返回之前，当前线程会被挂起. 调用线程只有在得到结果之后才会返回。
- 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

## 理解这四者的关系

[妖怪蒸唐僧的例子]

## 其他高级IO

非阻塞IO，纪录锁，系统V流机制，I/O多路转接（也叫I/O多路复用），readv和writev函数以及存储映射  
IO（mmap），这些统称为高级IO。

我们此处重点讨论的是I/O多路转接

## 非阻塞IO

### fcntl

一个文件描述符，默认都是阻塞IO。

函数原型如下。

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

int fcntl(int fd, int cmd, ... /* arg */ );
```

传入的cmd的值不同, 后面追加的参数也不相同.

fcntl函数有5种功能:

- 复制一个现有的描述符 (cmd=F\_DUPFD) .
- 获得/设置文件描述符标记(cmd=F\_GETFD或F\_SETFD).
- 获得/设置文件状态标记(cmd=F\_GETFL或F\_SETFL).
- 获得/设置异步I/O所有权(cmd=F\_GETOWN或F\_SETOWN).
- 获得/设置记录锁(cmd=F\_GETLK,F\_SETLK或F\_SETLKW).

我们此处只是用第三种功能, 获取/设置文件状态标记, 就可以将一个文件描述符设置为非阻塞.

## 实现函数SetNoBlock

基于fcntl, 我们实现一个SetNoBlock函数, 将文件描述符设置为非阻塞.

```
void SetNoBlock(int fd) {  
    int fl = fcntl(fd, F_GETFL);  
    if (fl < 0) {  
        perror("fcntl");  
        return;  
    }  
    fcntl(fd, F_SETFL, fl | O_NONBLOCK);  
}
```

- 使用F\_GETFL将当前的文件描述符的属性取出来(这是一个位图).
- 然后再使用F\_SETFL将文件描述符设置回去. 设置回去的同时, 加上一个O\_NONBLOCK参数.

## 轮询方式读取标准输入

```
#include <stdio.h>  
#include <unistd.h>  
#include <fcntl.h>  
  
void SetNoBlock(int fd) {  
    int fl = fcntl(fd, F_GETFL);  
    if (fl < 0) {  
        perror("fcntl");  
        return;  
    }  
    fcntl(fd, F_SETFL, fl | O_NONBLOCK);  
}  
  
int main() {  
    SetNoBlock(0);  
    while (1) {  
        char buf[1024] = {0};  
        ssize_t read_size = read(0, buf, sizeof(buf) - 1);  
        if (read_size < 0) {  
            perror("read");  
            sleep(1);  
            continue;  
        }  
        // Process the received data  
    }  
}
```

```

    }
    printf("input:%s\n", buf);
}
return 0;
}

```

## I/O多路转接之select

### 初识select

系统提供select函数来实现多路复用输入/输出模型.

- select系统调用是用来让我们的程序监视多个文件描述符的状态变化的;
- 程序会停在select这里等待, 直到被监视的文件描述符有一个或多个发生了状态改变;

### select函数原型

select的函数原型如下: #include <sys/select.h>

```

int select(int nfds, fd_set *readfds, fd_set *writefds,
          fd_set *exceptfds, struct timeval *timeout);

```

#### 参数解释:

- 参数nfds是需要监视的最大的文件描述符值+1;
- rdset,wrset,exset分别对应于需要检测的可读文件描述符的集合, 可写文件描述符的集合及异常文件描述符的集合;
- 参数timeout为结构timeval, 用来设置select()的等待时间

#### 参数timeout取值:

- NULL: 则表示select () 没有timeout, select将一直被阻塞, 直到某个文件描述符上发生了事件;
- 0: 仅检测描述符集合的状态, 然后立即返回, 并不等待外部事件的发生。
- 特定的时间值: 如果在指定的时间段里没有事件发生, select将超时返回。

#### 关于fd\_set结构

```

62 /* fd_set for select and pselect. */
63 #ifndef _USE_XOPEN
64 #define __FD_SETSIZE 1024
65 #endif
66 /* XPG4.2 requires this member name. Otherwise avoid the name
67   from the global namespace. */
68 #if !defined __USE_XOPEN
69     __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
70 # define __FDS_BITS(set) ((set)->__fds_bits)
71 #else
72     __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
73 # define __FDS_BITS(set) ((set)->__fds_bits)
74 #endif
75 } fd_set;

```

```

53 /* The fd_set member is required to be an array of longs. */
54 #typedef long int __fd_mask;
55

```

其实这个结构就是一个整数数组, 更严格的说, 是一个 "位图". 使用位图中对应的位来表示要监视的文件描述符.

提供了一组操作fd\_set的接口, 来比较方便的操作位图.

```
void FD_CLR(int fd, fd_set *set);      // 用来清除描述词组set中相关fd 的位  
int FD_ISSET(int fd, fd_set *set);     // 用来测试描述词组set中相关fd 的位是否为真  
void FD_SET(int fd, fd_set *set);      // 用来设置描述词组set中相关fd的位  
void FD_ZERO(fd_set *set);             // 用来清除描述词组set的全部位
```

### 关于timeval结构

timeval结构用于描述一段时间长度, 如果在这个时间内, 需要监视的描述符没有事件发生则函数返回, 返回值为0。

```
28 /* A time value that is accurate to the nearest  
29   microsecond but also has a range of years. */  
30 struct timeval  
31 {  
32   __time_t tv_sec;    /* Seconds. */  
33   __suseconds_t tv_usec; /* Microseconds. */  
34 };
```

### 函数返回值:

- 执行成功则返回文件描述词状态已改变的个数
- 如果返回0代表在描述词状态改变前已超过timeout时间, 没有返回
- 当有错误发生时则返回-1, 错误原因存于errno, 此时参数readfds, writefds, exceptfds和timeout的值变成不可预测。

### 错误值可能为:

- EBADF 文件描述词为无效的或该文件已关闭
- EINTR 此调用被信号所中断
- EINVAL 参数n 为负值。
- ENOMEM 核心内存不足

### 常见的程序片段如下:

```
fs_set readset;  
FD_SET(fd,&readset);  
select(fd+1,&readset,NULL,NULL,NULL);  
if(FD_ISSET(fd,readset)){.....}
```

## 理解select执行过程

理解select模型的关键在于理解fd\_set, 为说明方便, 取fd\_set长度为1字节, fd\_set中的每一bit可以对应一个文件描述符fd。则1字节长的fd\_set最大可以对应8个fd。

\* (1) 执行fd\_set set; FD\_ZERO(&set); 则set用位表示是0000,0000。 \* (2) 若fd = 5, 执行FD\_SET(fd,&set); 后set变为0001,0000(第5位置为1) \* (3) 若再加入fd = 2, fd=1, 则set变为0001,0011 \* (4) 执行select(6,&set,0,0,0)阻塞等待 \* (5) 若fd=1,fd=2上都发生可读事件, 则select返回, 此时set变为0000,0011。注意: 没有事件发生的fd=5被清空。

## socket就绪条件

### 读就绪

- socket内核中, 接收缓冲区中的字节数, 大于等于低水位标记SO\_RCVLOWAT. 此时可以无阻塞的读该文件描述符, 并且返回值大于0;
- socket TCP通信中, 对端关闭连接, 此时对该socket读, 则返回0;
- 监听的socket上有新的连接请求;
- socket上有未处理的错误;

## 写就绪

- socket内核中, 发送缓冲区中的可用字节数(发送缓冲区的空闲位置大小), 大于等于低水位标记SO\_SNDDLOWAT, 此时可以无阻塞的写, 并且返回值大于0;
- socket的写操作被关闭(close或者shutdown). 对一个写操作被关闭的socket进行写操作, 会触发SIGPIPE信号;
- socket使用非阻塞connect连接成功或失败之后;
- socket上有未读取的错误;

## 异常就绪(选学)

- socket上收到带外数据. 关于带外数据, 和TCP紧急模式相关(回忆TCP协议头中, 有一个紧急指针的字段), 同学们课后自己收集相关资料.

## select的特点

- 可监控的文件描述符个数取决于sizeof(fd\_set)的值. 我这边服务器上sizeof(fd\_set) = 512, 每bit表示一个文件描述符, 则我服务器上支持的最大文件描述符是 $512 \times 8 = 4096$ .
- 将fd加入select监控集的同时, 还要再使用一个数据结构array保存放到select监控集中的fd,
  - 一是用于在select返回后, array作为源数据和fd\_set进行FD\_ISSET判断。
  - 二是select返回后会把以前加入的但并无事件发生的fd清空, 则每次开始select前都要重新从array取得fd逐一加入(FD\_ZERO最先), 扫描array的同时取得fd最大值maxfd, 用于select的第一个参数。

备注: fd\_set的大小可以调整, 可能涉及到重新编译内核. 感兴趣的同学可以自己去收集相关资料.

## select缺点

- 每次调用select, 都需要手动设置fd集合, 从接口使用角度来说也非常不便.
- 每次调用select, 都需要把fd集合从用户态拷贝到内核态, 这个开销在fd很多时会很大
- 同时每次调用select都需要在内核遍历传递进来的所有fd, 这个开销在fd很多时也很大
- select支持的文件描述符数量太小.

## select使用示例: 检测标准输入输出

只检测标准输入:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/select.h>

int main() {
    fd_set read_fds;
    FD_ZERO(&read_fds);
    FD_SET(0, &read_fds);
```

```
for (;;) {
    printf("> ");
    fflush(stdout);
    int ret = select(1, &read_fds, NULL, NULL, NULL);
    if (ret < 0) {
        perror("select");
        continue;
    }
    if (FD_ISSET(0, &read_fds)) {
        char buf[1024] = {0};
        read(0, buf, sizeof(buf) - 1);
        printf("input: %s", buf);
    } else {
        printf("error! invalid fd\n");
        continue;
    }
    FD_ZERO(&read_fds);
    FD_SET(0, &read_fds);
}
return 0;
}
```

说明:

- 当只检测文件描述符0（标准输入）时，因为输入条件只有在你有输入信息的时候，才成立，所以如果不输入，就会产生超时信息。

## select使用示例

使用 select 实现字典服务器

tcp\_select\_server.hpp

```
#pragma once
#include <vector>
#include <unordered_map>
#include <functional>
#include <sys/select.h>
#include "tcp_socket.hpp"

// 必要的调试函数
inline void PrintFdSet(fd_set* fds, int max_fd) {
    printf("select fds: ");
    for (int i = 0; i < max_fd + 1; ++i) {
        if (!FD_ISSET(i, fds)) {
            continue;
        }
        printf("%d ", i);
    }
    printf("\n");
}

typedef std::function<void (const std::string& req, std::string* resp)> Handler;
```

```
// 把 Select 封装成一个类. 这个类虽然保存很多 TcpSocket 对象指针，但是不管理内存
class Selector {
public:
    Selector() {
        // [注意!] 初始化千万别忘了!!
        max_fd_ = 0;
        FD_ZERO(&read_fds_);
    }

    bool Add(const TcpSocket& sock) {
        int fd = sock.GetFd();
        printf("[Selector::Add] %d\n", fd);
        if (fd_map_.find(fd) != fd_map_.end()) {
            printf("Add failed! fd has in Selector!\n");
            return false;
        }
        fd_map_[fd] = sock;
        FD_SET(fd, &read_fds_);
        if (fd > max_fd_) {
            max_fd_ = fd;
        }
        return true;
    }

    bool Del(const TcpSocket& sock) {
        int fd = sock.GetFd();
        printf("[Selector::Del] %d\n", fd);
        if (fd_map_.find(fd) == fd_map_.end()) {
            printf("Del failed! fd has not in Selector!\n");
            return false;
        }
        fd_map_.erase(fd);
        FD_CLR(fd, &read_fds_);

        // 重新找到最大的文件描述符，从右往左找比较快
        for (int i = max_fd_; i >= 0; --i) {
            if (!FD_ISSET(i, &read_fds_)) {
                continue;
            }
            max_fd_ = i;
            break;
        }
        return true;
    }

    // 返回读就绪的文件描述符集
    bool Wait(std::vector<TcpSocket>*& output) {
        output->clear();
        // [注意] 此处必须要创建一个临时变量，否则原来的结果会被覆盖掉
        fd_set tmp = read_fds_;
        // DEBUG
    }
}
```

```
PrintFdSet(&tmp, max_fd_);  
  
int nfds = select(max_fd_ + 1, &tmp, NULL, NULL, NULL);  
if (nfds < 0) {  
    perror("select");  
    return false;  
}  
// [注意!] 此处的循环条件必须是 i < max_fd_ + 1  
for (int i = 0; i < max_fd_ + 1; ++i) {  
    if (!FD_ISSET(i, &tmp)) {  
        continue;  
    }  
    output->push_back(fd_map_[i]);  
}  
return true;  
}  
  
private:  
fd_set read_fds_;  
int max_fd_;  
// 文件描述符和 socket 对象的映射关系  
std::unordered_map<int, TcpSocket> fd_map_  
};  
  
class TcpSelectServer {  
public:  
    TcpSelectServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {}  
  
    bool Start(Handler handler) const {  
        // 1. 创建 socket  
        TcpSocket listen_sock;  
        bool ret = listen_sock.Socket();  
        if (!ret) {  
            return false;  
        }  
        // 2. 绑定端口号  
        ret = listen_sock.Bind(ip_, port_);  
        if (!ret) {  
            return false;  
        }  
        // 3. 进行监听  
        ret = listen_sock.Listen(5);  
        if (!ret) {  
            return false;  
        }  
        // 4. 创建 Selector 对象  
        Selector selector;  
        selector.Add(listen_sock);  
        // 5. 进入事件循环  
        for (;;) {  
  
            std::vector<TcpSocket> output;
```

```

bool ret = selector.Wait(&output);
if (!ret) {
    continue;
}
// 6. 根据就绪的文件描述符的差别，决定后续的处理逻辑
for (size_t i = 0; i < output.size(); ++i) {
    if (output[i].GetFd() == listen_sock.GetFd()) {
        // 如果就绪的文件描述符是 listen_sock，就执行 accept，并加入到 select 中
        TcpSocket new_sock;
        listen_sock.Accept(&new_sock, NULL, NULL);
        selector.Add(new_sock);
    } else {
        // 如果就绪的文件描述符是 new_sock，就进行一次请求的处理
        std::string req, resp;
        bool ret = output[i].Recv(&req);
        if (!ret) {
            selector.Del(output[i]);
            // [注意!] 需要关闭 socket
            output[i].Close();
            continue;
        }
        // 调用业务函数计算响应
        handler(req, &resp);
        // 将结果写回到客户端
        output[i].Send(resp);
    }
} // end for
} // end for (;;)
return true;
}

private:
    std::string ip_;
    uint16_t port_;
};

```

dict\_server.cc

这个代码和之前相同，只是把里面的 server 对象改成 TcpSelectServer 类即可。

客户端和之前的客户端完全相同，无需单独开发。

## I/O多路转接之poll [选学]

### poll函数接口

```

#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

// pollfd结构
struct pollfd {
    int fd;          /* file descriptor */
    short events;    /* requested events */
    short revents;   /* returned events */
};


```

## 参数说明

- fds是一个poll函数监听的结构列表. 每一个元素中, 包含了三部分内容: 文件描述符, 监听的事件集合, 返回的事件集合.
- nfds表示fds数组的长度.
- timeout表示poll函数的超时时间, 单位是毫秒(ms).

events和revents的取值:

事件	描述	是否可作为输入	是否可作为输出
POLLIN	数据(包括普通数据和优先数据)可读	是	是
POLLRDNORM	普通数据可读	是	是
POLLRDBAND	优先级带数据可读(Linux不支持)	是	是
POLLPRI	高优先级数据可读, 比如TCP带外数据	是	是
POLLOUT	数据(包括普通数据和优先数据)可写	是	是
POLLWRNORM	普通数据可写	是	是
POLLWRBAND	优先级带数据可写	是	是
POLLRDHUP	TCP连接被对方关闭, 或者对方关闭了写操作. 它由GNU引入	是	是
POLLERR	错误	否	是
POLLHUP	挂起. 比如管道的写端被关闭后, 读端描述符上将收到POLLHUP事件	否	是
POLLNVAL	文件描述符没有打开	否	是

## 返回结果

- 返回值小于0, 表示出错;
- 返回值等于0, 表示poll函数等待超时;
- 返回值大于0, 表示poll由于监听的文件描述符就绪而返回.

## socket就绪条件

同select

## poll的优点

不同与select使用三个位图来表示三个fdset的方式, poll使用一个pollfd的指针实现.

- pollfd结构包含了要监视的event和发生的event, 不再使用select“参数-值”传递的方式. 接口使用比select更方便.

- poll并没有最大数量限制(但是数量过大后性能也是会下降).

## poll的缺点

poll中监听的文件描述符数目增多时

- 和select函数一样, poll返回后, 需要轮询pollfd来获取就绪的描述符.
- 每次调用poll都需要把大量的pollfd结构从用户态拷贝到内核中.
- 同时连接的大量客户端在一时刻可能只有很少的处于就绪状态, 因此随着监视的描述符数量的增长, 其效率也会线性下降.

## poll示例: 使用poll监控标准输入

```
#include <poll.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    struct pollfd poll_fd;
    poll_fd.fd = 0;
    poll_fd.events = POLLIN;

    for (;;) {
        int ret = poll(&poll_fd, 1, 1000);
        if (ret < 0) {
            perror("poll");
            continue;
        }
        if (ret == 0) {
            printf("poll timeout\n");
            continue;
        }
        if (poll_fd.revents == POLLIN) {
            char buf[1024] = {0};
            read(0, buf, sizeof(buf) - 1);
            printf("stdin:%s", buf);
        }
    }
}
```

## I/O多路转接之epoll

### epoll初识

按照man手册的说法: 是为处理大批量句柄而作了改进的poll.

它是在2.5.44内核中被引进的(epoll(4) is a new API introduced in Linux kernel 2.5.44)

它几乎具备了之前所说的一切优点, 被公认为Linux2.6下性能最好的多路I/O就绪通知方法.

### epoll的相关系统调用

epoll 有3个相关的系统调用.

## epoll\_create

```
int epoll_create(int size);
```

创建一个epoll的句柄.

- 自从linux2.6.8之后, size参数是被忽略的.
- 用完之后, 必须调用close()关闭.

## epoll\_ctl

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

epoll的事件注册函数.

- 它不同于select()是在监听事件时告诉内核要监听什么类型的事件, 而是在这里先注册要监听的事件类型.
- 第一个参数是epoll\_create()的返回值(epoll的句柄).
- 第二个参数表示动作, 用三个宏来表示.
- 第三个参数是需要监听的fd.
- 第四个参数是告诉内核需要监听什么事.

第二个参数的取值:

- EPOLL\_CTL\_ADD : 注册新的fd到epfd中;
- EPOLL\_CTL\_MOD : 修改已经注册的fd的监听事件;
- EPOLL\_CTL\_DEL : 从epfd中删除一个fd;

struct epoll\_event结构如下:

```

typedef union epoll_data
{
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

struct epoll_event
{
    uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
} __EPOLL_PACKED;

```

events可以是以下几个宏的集合:

- EPOLLIN : 表示对应的文件描述符可以读 (包括对端SOCKET正常关闭);
- EPOLLOUT : 表示对应的文件描述符可以写;
- EPOLLPRI : 表示对应的文件描述符有紧急的数据可读 (这里应该表示有带外数据到来);
- EPOLLERR : 表示对应的文件描述符发生错误;
- EPOLLHUP : 表示对应的文件描述符被挂断;
- EPOLLET : 将EPOLL设为边缘触发(Edge Triggered)模式, 这是相对于水平触发(Level Triggered)来说的.
- EPOLLONESHOT: 只监听一次事件, 当监听完这次事件之后, 如果还需要继续监听这个socket的话, 需要再次把这个socket加入到EPOLL队列里.

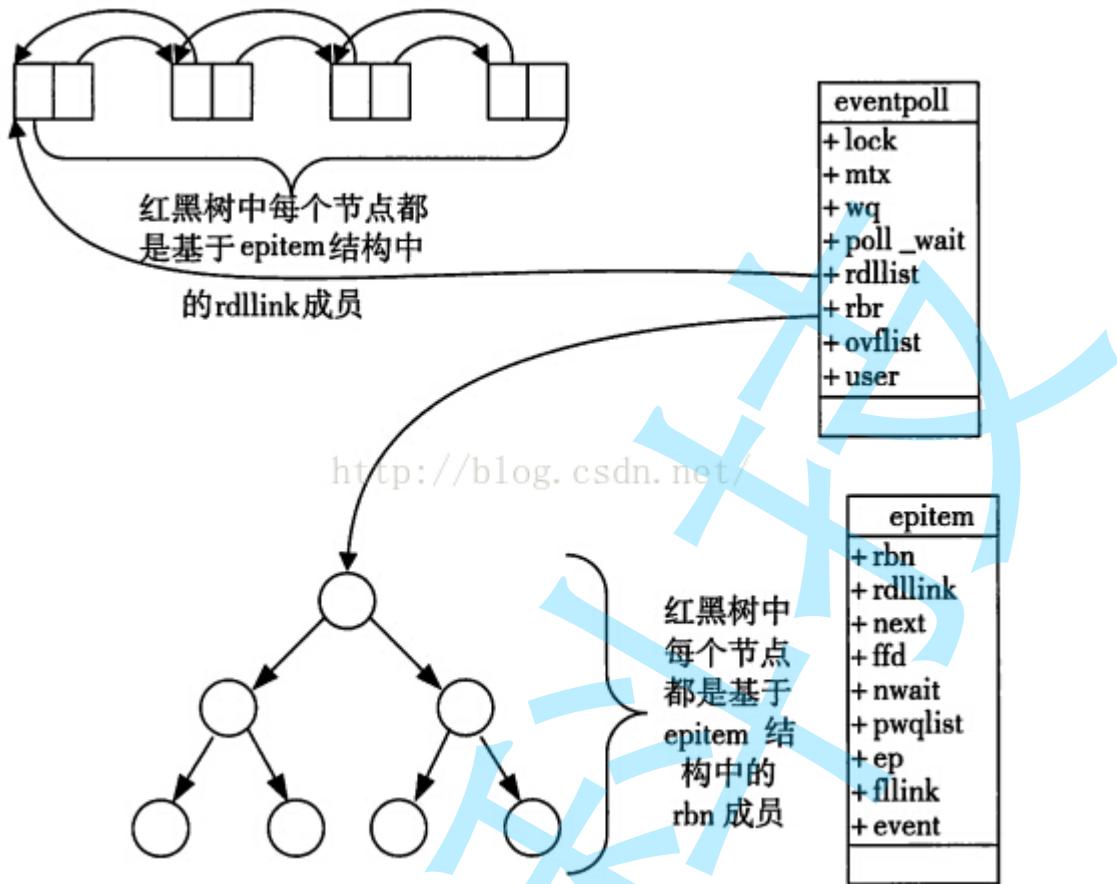
## epoll\_wait

```
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

收集在epoll监控的事件中已经发送的事件.

- 参数events是分配好的epoll\_event结构体数组.
- epoll将会把发生的事件赋值到events数组中 (events不可以是空指针, 内核只负责把数据复制到这个events数组中, 不会去帮助我们在用户态中分配内存).
- maxevents告之内核这个events有多大, 这个 maxevents的值不能大于创建epoll\_create()时的size.
- 参数timeout是超时时间 (毫秒, 0会立即返回, -1是永久阻塞).
- 如果函数调用成功, 返回对应I/O上已准备好的文件描述符数目, 如返回0表示已超时, 返回小于0表示函数失败.

## epoll工作原理



- 当某一进程调用epoll\_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。

```
struct eventpoll{
    ...
    /*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件*/
    struct rb_root rbr;
    /*双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ...
};
```

- 每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll\_ctl方法向epoll对象中添加进来的事件。
- 这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是 $O(\log n)$ , 其中n为树的高度)。
- 而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当响应的事件发生时会调用这个回调方法。
- 这个回调方法在内核中叫ep\_poll\_callback,它会将发生的事件添加到rdlist双链表中。
- 在epoll中，对于每一个事件，都会建立一个epitem结构体。

```

struct epitem{
    struct rb_node rbn; //红黑树节点
    struct list_head rdllink; //双向链表节点
    struct epoll_filefd ffd; //事件句柄信息
    struct eventpoll *ep; //指向其所属的eventpoll对象
    struct epoll_event event; //期待发生的事件类型
}

```

- 当调用epoll\_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。
- 如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。这个操作的时间复杂度是O(1)。

总结一下，epoll的使用过程就是三部曲：

- 调用epoll\_create创建一个epoll句柄；
- 调用epoll\_ctl，将要监控的文件描述符进行注册；
- 调用epoll\_wait，等待文件描述符就绪；

## epoll的优点(和 select 的缺点对应)

- 接口使用方便：虽然拆分成了三个函数，但是反而使用起来更方便高效。不需要每次循环都设置关注的文件描述符，也做到了输入输出参数分离。
- 数据拷贝轻量：只在合适的时候调用 `EPOLL_CTL_ADD` 将文件描述符结构拷贝到内核中，这个操作并不频繁（而select/poll都是每次循环都要进行拷贝）。
- 事件回调机制：避免使用遍历，而是使用回调函数的方式，将就绪的文件描述符结构加入到就绪队列中，`epoll_wait` 返回直接访问就绪队列就知道哪些文件描述符就绪。这个操作时间复杂度O(1)。即使文件描述符数目很多，效率也不会受到影响。
- 没有数量限制：文件描述符数目无上限。

### 注意!!

网上有些博客说，epoll中使用了内存映射机制

- 内存映射机制：内核直接将就绪队列通过mmap的方式映射到用户态。避免了拷贝内存这样的额外性能开销。

这种说法是不准确的。我们定义的`struct epoll_event`是在用户空间中分配好的内存。势必还是需要将内核的数据拷贝到这个用户空间的内存中的。

请同学们对比总结select, poll, epoll之间的优点和缺点(重要，面试中常见)。

## epoll工作方式

### 你妈喊你吃饭的例子

你正在吃鸡，眼看进入了决赛圈，`你妈饭做好了，喊你吃饭`的时候有两种方式：

1. 如果你妈喊你一次，你没动，那么你妈会继续喊你第二次，第三次... (亲妈，水平触发)
2. 如果你妈喊你一次，你没动，你妈就不管了(后妈，边缘触发)

epoll有2种工作方式-水平触发(LT)和边缘触发(ET)

假如有这样一个例子:

- 我们已经把一个tcp socket添加到epoll描述符
- 这个时候socket的另一端被写入了2KB的数据
- 调用epoll\_wait, 并且它会返回. 说明它已经准备好读取操作
- 然后调用read, 只读取了1KB的数据
- 继续调用epoll\_wait.....

### 水平触发Level Triggered 工作模式

epoll默认状态下就是LT工作模式.

- 当epoll检测到socket上事件就绪的时候, 可以不立刻进行处理. 或者只处理一部分.
- 如上面的例子, 由于只读了1K数据, 缓冲区中还剩1K数据, 在第二次调用 epoll\_wait 时, epoll\_wait 仍然会立刻返回并通知socket读事件就绪.
- 直到缓冲区上所有的数据都被处理完, epoll\_wait 才不会立刻返回.
- 支持阻塞读写和非阻塞读写

### 边缘触发Edge Triggered工作模式

如果我们在第1步将socket添加到epoll描述符的时候使用了EPOLLET标志, epoll进入ET工作模式.

- 当epoll检测到socket上事件就绪时, 必须立刻处理.
- 如上面的例子, 虽然只读了1K的数据, 缓冲区还剩1K的数据, 在第二次调用 epoll\_wait 的时候, epoll\_wait 不会再返回了.
- 也就是说, ET模式下, 文件描述符上的事件就绪后, 只有一次处理机会.
- ET的性能比LT性能更高( epoll\_wait 返回的次数少了很多). Nginx默认采用ET模式使用epoll.
- 只支持非阻塞的读写

select和poll其实也是工作在LT模式下. epoll既可以支持LT, 也可以支持ET.

## 对比LT和ET

LT是 epoll 的默认行为. 使用 ET 能够减少 epoll 触发的次数. 但是代价就是强逼着程序猿一次响应就绪过程中就把所有的数据都处理完.

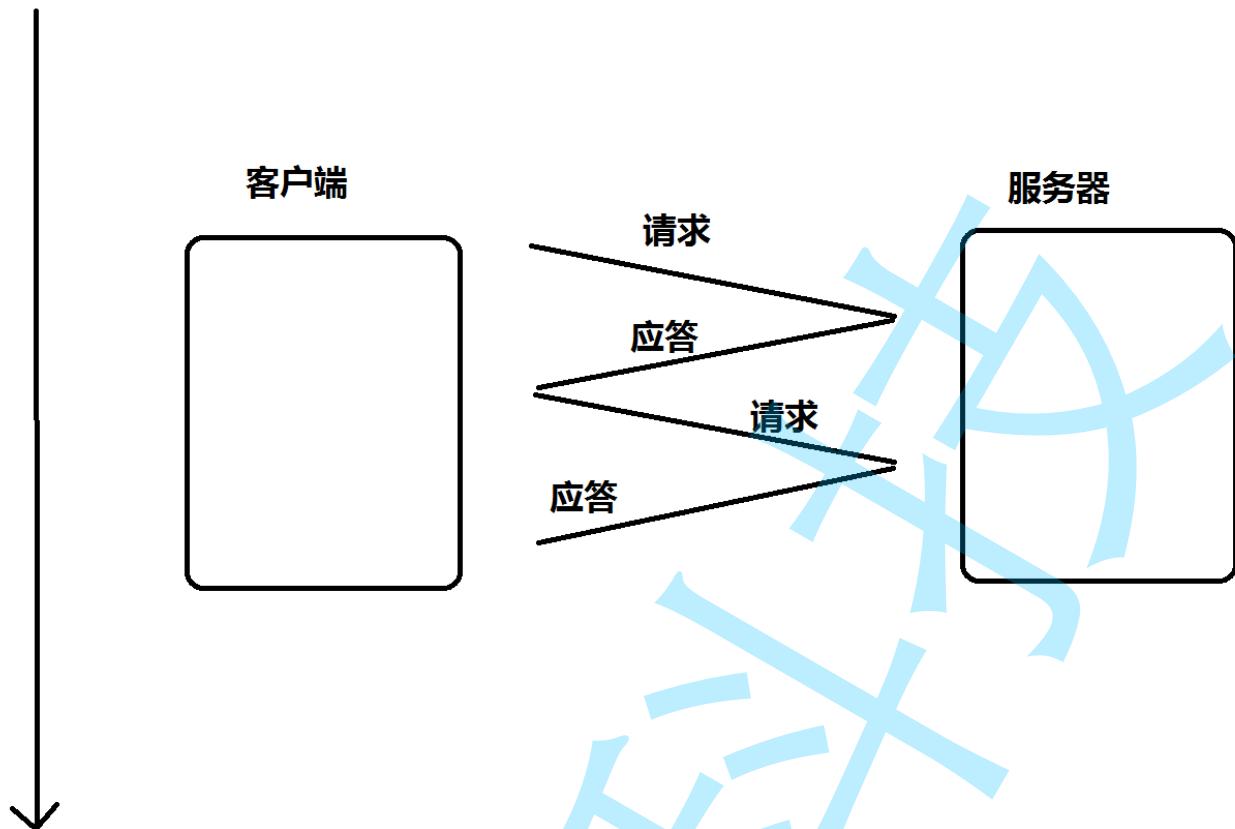
相当于一个文件描述符就绪之后, 不会反复被提示就绪, 看起来就比 LT 更高效一些. 但是在 LT 情况下如果也能做到每次就绪的文件描述符都立刻处理, 不让这个就绪被重复提示的话, 其实性能也是一样的.

另一方面, ET 的代码复杂程度更高了.

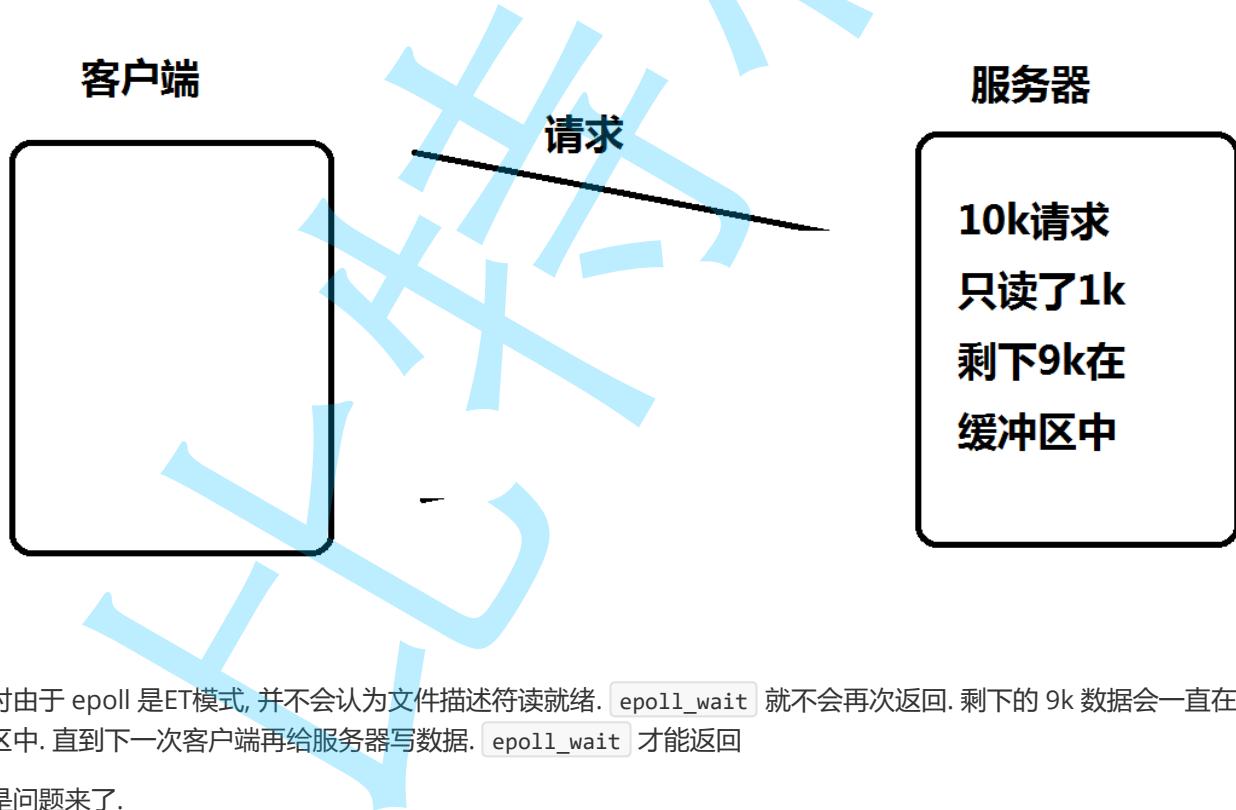
## 理解ET模式和非阻塞文件描述符

使用 ET 模式的 epoll, 需要将文件描述设置为非阻塞. 这个不是接口上的要求, 而是 "工程实践" 上的要求.

假设这样的场景: 服务器接收到一个10k的请求, 会向客户端返回一个应答数据. 如果客户端收不到应答, 不会发送第二个10k请求.



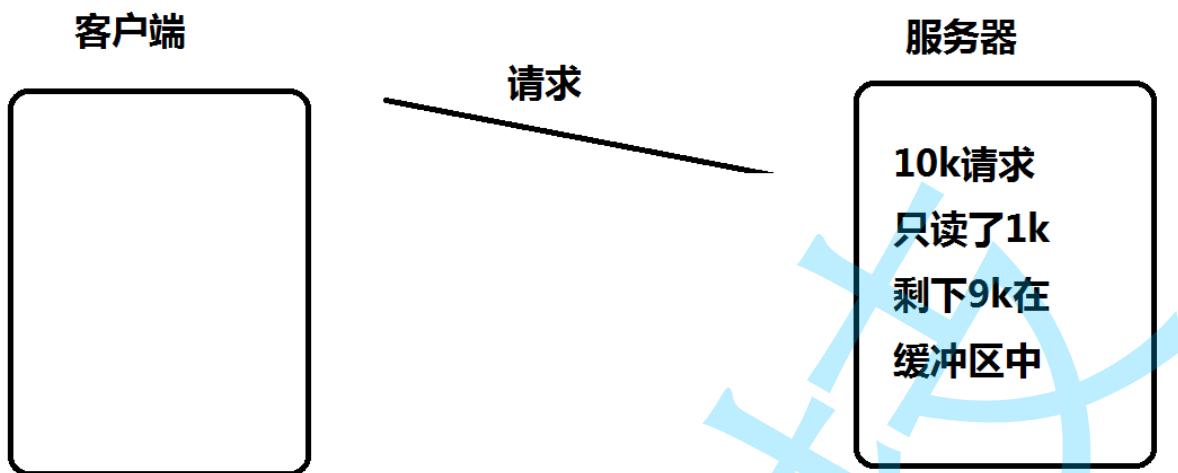
如果服务端写的代码是阻塞式的read, 并且一次只 read 1k 数据的话(read不能保证一次就把所有的数据都读出来, 参考 man 手册的说明, 可能被信号打断), 剩下的9k数据就会待在缓冲区中.



此时由于 epoll 是ET模式, 并不会认为文件描述符读就绪. `epoll_wait` 就不会再次返回. 剩下的 9k 数据会一直在缓冲区中. 直到下一次客户端再给服务器写数据. `epoll_wait` 才能返回

但是问题来了.

- 服务器只读到1k个数据, 要10k读完才会给客户端返回响应数据.
- 客户端要读到服务器的响应, 才会发送下一个请求
- 客户端发送了下一个请求, `epoll_wait` 才会返回, 才能去读缓冲区中剩余的数据.



**客户端: 你要把10k数据都读完  
给我发个响应, 我才能发下一个  
数据啊嘤嘤嘤**

**服务器: 客户端快给我  
再发一个数据, 我好把  
剩余的9k数据读出来**

所以, 为了解决上述问题(阻塞read不一定能一下把完整的请求读完), 于是就可以使用非阻塞轮训的方式来读缓冲区, 保证一定能把完整的请求都读出来.

而如果是LT没这个问题. 只要缓冲区中的数据没读完, 就能够让 `epoll_wait` 返回文件描述符读就绪.

## epoll的使用场景

epoll的高性能, 是有一定的特定场景的. 如果场景选择的不适宜, epoll的性能可能适得其反.

- 对于多连接, 且多连接中只有一部分连接比较活跃时, 比较适合使用epoll.

例如, 典型的一个需要处理上万个客户端的服务器, 例如各种互联网APP的入口服务器, 这样的服务器就很适合epoll.

如果只是系统内部, 服务器和服务器之间进行通信, 只有少数的几个连接, 这种情况下用epoll就并不合适. 具体要根据需求和场景特点来决定使用哪种IO模型.

## epoll中的惊群问题(选学)

惊群问题有些面试官可能会问到. 建议同学们课后自己查阅资料了解一下问题的解决方案.

参考 <http://blog.csdn.net/fsmiy/article/details/36873357>

## epoll示例: epoll服务器(LT模式)

`tcp_epoll_server.hpp`

```
//////////  
// 封装一个 Epoll 服务器, 只考虑读就绪的情况  
//////////  
  
#pragma once  
#include <vector>  
#include <functional>
```

```
#include <sys/epoll.h>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string&, std::string* resp)> Handler;

class Epoll {
public:
    Epoll() {
        epoll_fd_ = epoll_create(10);
    }

    ~Epoll() {
        close(epoll_fd_);
    }

    bool Add(const TcpSocket& sock) const {
        int fd = sock.GetFd();
        printf("[Epoll Add] fd = %d\n", fd);
        epoll_event ev;
        ev.data.fd = fd;
        ev.events = EPOLLIN;
        int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, fd, &ev);
        if (ret < 0) {
            perror("epoll_ctl ADD");
            return false;
        }
        return true;
    }

    bool Del(const TcpSocket& sock) const {
        int fd = sock.GetFd();
        printf("[Epoll Del] fd = %d\n", fd);
        int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_DEL, fd, NULL);
        if (ret < 0) {
            perror("epoll_ctl DEL");
            return false;
        }
        return true;
    }

    bool Wait(std::vector<TcpSocket*>* output) const {
        output->clear();
        epoll_event events[1000];
        int nfds = epoll_wait(epoll_fd_, events, sizeof(events) / sizeof(events[0]), -1);
        if (nfds < 0) {
            perror("epoll_wait");
            return false;
        }
        // [注意!] 此处必须是循环到 nfds, 不能多循环
        for (int i = 0; i < nfds; ++i) {
            TcpSocket sock(events[i].data.fd);
            output->push_back(sock);
        }
    }
}
```

```
    return true;
}

private:
    int epoll_fd_;
};

class TcpEpollServer {
public:
    TcpEpollServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {

    }

    bool Start(Handler handler) {
        // 1. 创建 socket
        TcpSocket listen_sock;
        CHECK_RET(listen_sock.Socket());
        // 2. 绑定
        CHECK_RET(listen_sock.Bind(ip_, port_));
        // 3. 监听
        CHECK_RET(listen_sock.Listen(5));
        // 4. 创建 Epoll 对象，并将 listen_sock 加入进去
        Epoll epoll;
        epoll.Add(listen_sock);
        // 5. 进入事件循环
        for (;;) {
            // 6. 进行 epoll_wait
            std::vector<TcpSocket> output;
            if (!epoll.Wait(&output)) {
                continue;
            }
            // 7. 根据就绪的文件描述符的种类决定如何处理
            for (size_t i = 0; i < output.size(); ++i) {
                if (output[i].GetFd() == listen_sock.GetFd()) {
                    // 如果是 listen_sock，就调用 accept
                    TcpSocket new_sock;
                    listen_sock.Accept(&new_sock);
                    epoll.Add(new_sock);
                } else {
                    // 如果是 new_sock，就进行一次读写
                    std::string req, resp;
                    bool ret = output[i].Recv(&req);
                    if (!ret) {
                        // [注意！！] 需要把不用的 socket 关闭
                        // 先后顺序别搞反。不过在 epoll 删除的时候其实就已经关闭 socket 了
                        epoll.Del(output[i]);
                        output[i].Close();
                        continue;
                    }
                    handler(req, &resp);
                    output[i].Send(resp);
                }
            }
        } // end for
    } // end for (;;)
}
```

```

    }
    return true;
}

private:
    std::string ip_;
    uint16_t port_;
};

}

```

dict\_server.cc 只需要将 server 对象的类型改成 TcpEpollServer 即可.

## epoll示例: epoll服务器(ET模式)

基于 LT 版本稍加修改即可

1. 修改 tcp\_socket.hpp, 新增非阻塞读和非阻塞写接口
2. 对于 accept 返回的 new\_sock 加上 EPOLLET 这样的选项

**注意:** 此代码暂时未考虑 listen\_sock ET 的情况. 如果将 listen\_sock 设为 ET, 则需要非阻塞轮询的方式 accept. 否则会导致同一时刻大量的客户端同时连接的时候, 只能 accept 一次的问题.

tcp\_socket.hpp

```

// 以下代码添加在 TcpSocket 类中
// 非阻塞 IO 接口
bool SetNoBlock() {
    int fl = fcntl(fd_, F_GETFL);
    if (fl < 0) {
        perror("fcntl F_GETFL");
        return false;
    }
    int ret = fcntl(fd_, F_SETFL, fl | O_NONBLOCK);
    if (ret < 0) {
        perror("fcntl F_SETFL");
        return false;
    }
    return true;
}

bool RecvNoBlock(std::string* buf) const {
    // 对于非阻塞 IO 读数据, 如果 TCP 接受缓冲区为空, 就会返回错误
    // 错误码为 EAGAIN 或者 EWOULDBLOCK, 这种情况也是意料之中, 需要重试
    // 如果当前读到的数据长度小于尝试读的缓冲区的长度, 就退出循环
    // 这种写法其实不算特别严谨(没有考虑粘包问题)
    buf->clear();
    char tmp[1024 * 10] = {0};
    for (;;) {
        ssize_t read_size = recv(fd_, tmp, sizeof(tmp) - 1, 0);
        if (read_size < 0) {
            if (errno == EWOULDBLOCK || errno == EAGAIN) {
                continue;
            }
            perror("recv");
        }
    }
}

```

```

        return false;
    }
    if (read_size == 0) {
        // 对端关闭, 返回 false
        return false;
    }
    tmp[read_size] = '\0';
    *buf += tmp;
    if (read_size < (ssize_t)sizeof(tmp) - 1) {
        break;
    }
}
return true;
}

bool SendNoBlock(const std::string& buf) const {
    // 对于非阻塞 IO 的写入, 如果 TCP 的发送缓冲区已经满了, 就会出现出错的情况
    // 此时的错误号是 EAGAIN 或者 EWOULDBLOCK. 这种情况下不应放弃治疗
    // 而要进行重试
    ssize_t cur_pos = 0; // 记录当前写到的位置
    ssize_t left_size = buf.size();
    for (;;) {
        ssize_t write_size = send(fd_, buf.data() + cur_pos, left_size, 0);
        if (write_size < 0) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                // 重试写入
                continue;
            }
            return false;
        }
        cur_pos += write_size;
        left_size -= write_size;
        // 这个条件说明写完需要的数据了
        if (left_size <= 0) {
            break;
        }
    }
    return true;
}

```

## tcp\_epoll\_server.hpp

```

/////////
// 封装一个 Epoll ET 服务器
// 修改点:
// 1. 对于 new_sock, 加上 EPOLLET 标记
// 2. 修改 TcpSocket 支持非阻塞读写
// [注意!] listen_sock 如果设置成 ET, 就需要非阻塞调用 accept 了
// 稍微麻烦一点, 此处暂时不实现
/////////

```

```
#pragma once
```

```
#include <vector>
#include <functional>
#include <sys/epoll.h>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string&, std::string* resp)> Handler;

class Epoll {
public:
    Epoll() {
        epoll_fd_ = epoll_create(10);
    }

    ~Epoll() {
        close(epoll_fd_);
    }

    bool Add(const TcpSocket& sock, bool epoll_et = false) const {
        int fd = sock.GetFd();
        printf("[Epoll Add] fd = %d\n", fd);
        epoll_event ev;
        ev.data.fd = fd;
        if (epoll_et) {
            ev.events = EPOLLIN | EPOLLET;
        } else {
            ev.events = EPOLLIN;
        }
        int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, fd, &ev);
        if (ret < 0) {
            perror("epoll_ctl ADD");
            return false;
        }
        return true;
    }

    bool Del(const TcpSocket& sock) const {
        int fd = sock.GetFd();
        printf("[Epoll Del] fd = %d\n", fd);
        int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_DEL, fd, NULL);
        if (ret < 0) {
            perror("epoll_ctl DEL");
            return false;
        }
        return true;
    }

    bool Wait(std::vector<TcpSocket>*>* output) const {
        output->clear();
        epoll_event events[1000];
        int nfds = epoll_wait(epoll_fd_, events, sizeof(events) / sizeof(events[0]), -1);
        if (nfds < 0) {
            perror("epoll_wait");
        }

        return false;
    }
}
```

```
    }
    // [注意!] 此处必须是循环到 nfds, 不能多循环
    for (int i = 0; i < nfds; ++i) {
        TcpSocket sock(events[i].data.fd);
        output->push_back(sock);
    }
    return true;
}

private:
    int epoll_fd_;
};

class TcpEpollServer {
public:
    TcpEpollServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {

    }

    bool Start(Handler handler) {
        // 1. 创建 socket
        TcpSocket listen_sock;
        CHECK_RET(listen_sock.Socket());
        // 2. 绑定
        CHECK_RET(listen_sock.Bind(ip_, port_));
        // 3. 监听
        CHECK_RET(listen_sock.Listen(5));
        // 4. 创建 Epoll 对象, 并将 listen_sock 加入进去
        Epoll epoll;
        epoll.Add(listen_sock);
        // 5. 进入事件循环
        for (;;) {
            // 6. 进行 epoll_wait
            std::vector<TcpSocket> output;
            if (!epoll.Wait(&output)) {
                continue;
            }
            // 7. 根据就绪的文件描述符的种类决定如何处理
            for (size_t i = 0; i < output.size(); ++i) {
                if (output[i].GetFd() == listen_sock.GetFd()) {
                    // 如果是 listen_sock, 就调用 accept
                    TcpSocket new_sock;
                    listen_sock.Accept(&new_sock);
                    epoll.Add(new_sock, true);
                } else {
                    // 如果是 new_sock, 就进行一次读写
                    std::string req, resp;
                    bool ret = output[i].RecvNoBlock(&req);
                    if (!ret) {
                        // [注意!!] 需要把不用的 socket 关闭
                        // 先后顺序别搞反. 不过在 epoll 删除的时候其实就已经关闭 socket 了
                        epoll.Del(output[i]);
                    }
                    output[i].Close();
                }
            }
        }
    }
}
```

```
        continue;
    }
    handler(req, &resp);
    output[i].SendNoBlock(resp);
    printf("[client %d] req: %s, resp: %s\n", output[i].GetFd(),
           req.c_str(), resp.c_str());
} // end for
} // end for (;;)
}
return true;
}

private:
std::string ip_;
uint16_t port_;
};
```

## 参考资料

[epoll详解](#)

[apache/nginx网络模型](#)