**C++后台发开面试宝典**

记录源：牛客网2018-08-13 15:28:03至2019-1-29面筋，补充ing

记录者：一个专注于微波炉制造业的烟酒生

记录时间：2019年1月27日-至今

目录

[面试基础问题 6](#_Toc4337196)

[C++基础知识 6](#_Toc4337197)

[智能指针 6](#_Toc4337198)

[static关键字 9](#_Toc4337199)

[C/C++程序编译的整个过程 9](#_Toc4337200)

[虚函数实现机制 10](#_Toc4337201)

[析构函数为什么需要用虚函数 10](#_Toc4337202)

[构造函数为什么不能为虚函数 10](#_Toc4337203)

[纯虚函数和虚函数 11](#_Toc4337204)

[虚函数表 11](#_Toc4337205)

[C++多态如何实现 11](#_Toc4337206)

[C和C++区别 12](#_Toc4337207)

[i++和++i的区别 12](#_Toc4337208)

[C++中4种类型转换方式 12](#_Toc4337209)

[C++常用的设计模式 12](#_Toc4337210)

[C/C++内存模型，每个区放什么 13](#_Toc4337211)

[内存中的栈和堆分配 13](#_Toc4337212)

[static和const可以修饰同一个变量么 14](#_Toc4337213)

[const和#define的区别 14](#_Toc4337214)

[指针和引用的区别 14](#_Toc4337215)

[静态绑定和动态绑定是怎么实现的 15](#_Toc4337216)

[内存溢出和内存泄漏 15](#_Toc4337217)

[面向对象理解 16](#_Toc4337218)

[重载、重写和隐藏 16](#_Toc4337219)

[new/delete和malloc/free的用法和区别 16](#_Toc4337220)

[C++11新特性-待加 17](#_Toc4337221)

[C++左值、右值引用和move语义 18](#_Toc4337222)

[vector 18](#_Toc4337223)

[list 22](#_Toc4337224)

[deque 28](#_Toc4337225)

[stl中set、map底层实现，与hashtalbe差异 28](#_Toc4337226)

[read函数和fread函数 29](#_Toc4337227)

[C++异常安全 30](#_Toc4337228)

[strcpy和memcpy区别 30](#_Toc4337229)

[静态链接库和动态链接库 30](#_Toc4337230)

[数据结构和算法 32](#_Toc4337231)

[几种重要的排序 32](#_Toc4337232)

[AVL树 39](#_Toc4337233)

[B树和B+树 44](#_Toc4337234)

[哈夫曼树 53](#_Toc4337235)

[红黑树 53](#_Toc4337236)

[红黑树和AVL树差别 60](#_Toc4337237)

[HashMap是否线程安全 60](#_Toc4337238)

[如何解决hash冲突 61](#_Toc4337239)

[字典树 61](#_Toc4337240)

[二叉树、完全二叉树、满二叉树、平衡二叉树 63](#_Toc4337241)

[跳表 63](#_Toc4337242)

[最小生成树算法 64](#_Toc4337243)

[最短路径算法 69](#_Toc4337244)

[拓扑排序 81](#_Toc4337245)

[关键路径算法（仅作了解） 84](#_Toc4337246)

[计算机网络 85](#_Toc4337247)

[OSI七层模型 85](#_Toc4337248)

[TCP/IP五层模型 85](#_Toc4337249)

[TCP/IP协议总结 87](#_Toc4337250)

[TCP、UDP区别 95](#_Toc4337251)

[TCP三次握手、四次挥手 96](#_Toc4337252)

[TCP状态转换图 97](#_Toc4337253)

[TCP状态中TIME\_WAIT作用 97](#_Toc4337254)

[TCP连接建立需要为什么不是两次握手 97](#_Toc4337255)

[TCP第三次握手失败会出现什么 98](#_Toc4337256)

[TCP长连接和短链接及优缺点 98](#_Toc4337257)

[TCP拥塞控制-慢启动、拥塞避免、快重传、快启动 98](#_Toc4337258)

[TCP如何保证可靠性传输 100](#_Toc4337259)

[TCP如何解决粘包、拆包问题 103](#_Toc4337260)

[TCP为什么可靠 105](#_Toc4337261)

[UDP如何实现TCP可靠传输 105](#_Toc4337262)

[IP知识点总结 105](#_Toc4337263)

[IP地址和子网掩码 113](#_Toc4337264)

[ARP解析过程 114](#_Toc4337265)

[DNS原理 114](#_Toc4337266)

[HTTP1.0、HTTP1.1、HTTP2.0区别 115](#_Toc4337267)

[HTTP和HTTPS区别 116](#_Toc4337268)

[HTTP状态码 117](#_Toc4337269)

[Nagle算法 118](#_Toc4337270)

[ping命令 119](#_Toc4337271)

[数据库 120](#_Toc4337272)

[数据库类别 120](#_Toc4337273)

[关系型数据库和非关系型数据库区别 121](#_Toc4337274)

[SQL常见语句（分好很重要） 122](#_Toc4337275)

[MySQL内链接，外链接（左链接、右链接、全链接） 129](#_Toc4337276)

[MySQL索引类型和原理 130](#_Toc4337277)

[MySQL数据存储引擎 135](#_Toc4337278)

[Sql优化思路 137](#_Toc4337279)

[数据库高并发解决方法总结 138](#_Toc4337280)

[网络编程 140](#_Toc4337281)

[网络编程API有哪些 140](#_Toc4337282)

[Linux 143](#_Toc4337283)

[top命令 143](#_Toc4337284)

[ps命令 144](#_Toc4337285)

[netstat命令 145](#_Toc4337286)

[awk命令 146](#_Toc4337287)

[find命令 146](#_Toc4337288)

[grep命令 148](#_Toc4337289)

[wc命令 149](#_Toc4337290)

[sed命令 149](#_Toc4337291)

[head和tail命令 150](#_Toc4337292)

[正则表达式 150](#_Toc4337293)

[如何查找出现频率最高的100个IP地址 152](#_Toc4337294)

[linux如何统计文件中某个字符串出现的频率 153](#_Toc4337295)

[linux启动的第一个进程 153](#_Toc4337296)

[linux查看端口占用 153](#_Toc4337297)

[linux查看CPU和内存使用 153](#_Toc4337298)

[Linux查看系统负载命令 154](#_Toc4337299)

[Linux调试程序 154](#_Toc4337300)

[Linux硬链接和软连接 154](#_Toc4337301)

[core dump 155](#_Toc4337302)

[cmake和makefile-待加 155](#_Toc4337303)

[操作系统、进程和线程 156](#_Toc4337304)

[提升磁盘IO性能的几个技巧 156](#_Toc4337305)

[物理内存和虚拟内存（见深入理解操作系统P559） 156](#_Toc4337306)

[缓存IO和直接IO 160](#_Toc4337307)

[作业调度算法 161](#_Toc4337308)

[进程地址空间分布 161](#_Toc4337309)

[线程和进程 162](#_Toc4337310)

[进程和线程的调度 163](#_Toc4337311)

[线程的创建和结束 165](#_Toc4337312)

[线程状态 166](#_Toc4337313)

[线程间通信与线程同步-待加 168](#_Toc4337314)

[互斥锁和信号量-待加 168](#_Toc4337315)

[多线程程序如何调试 168](#_Toc4337316)

[C++实现线程安全的队列 169](#_Toc4337317)

[线程池 169](#_Toc4337318)

[进程的创建和结束 170](#_Toc4337319)

[如何创建新进程，内核做了啥 170](#_Toc4337320)

[孤儿进程、僵尸进程和守护进程 170](#_Toc4337321)

[多进程 171](#_Toc4337322)

[进程间通信方式 171](#_Toc4337323)

[怎样实现进程同步 199](#_Toc4337324)

[fork()返回值 200](#_Toc4337325)

[LT（水平触发）和ET（边缘触发）-待加 201](#_Toc4337326)

[epoll与select/poll区别-待加 201](#_Toc4337327)

[IO模型-待加 201](#_Toc4337328)

[生产者、消费者模式问题-待加 201](#_Toc4337329)

[单例模式-代码-待加 201](#_Toc4337330)

[死锁 201](#_Toc4337331)

[如何降低锁的竞争 202](#_Toc4337332)

[并发和并行 202](#_Toc4337333)

[场景题 203](#_Toc4337334)

[手撕代码 206](#_Toc4337335)

[其他 215](#_Toc4337336)

[干货帖子 216](#_Toc4337337)

# 面试基础问题

## C++基础知识

### 智能指针

auto\_ptr(C++11弃用),shared\_ptr,unique\_ptr,weak\_ptr

**shared\_ptr**：可被多个指针共享，它使用计数机制来表明资源被几个指针共享，可以通过成员函数use\_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto\_ptr、unique\_ptr、weak\_ptr来构造。调用release()释放资源所有权，计数减一。

**unique\_ptr**：是取代C++98的auto\_ptr产物。他是一个独享所有权的智能指针，它提供了严格意义上的所有权，包括：1.它指向的对象。2.无法进行复制构造，无法进行复制赋值操作。但可以进行移动构造和移动赋值构造std.move()。3.保存指向某个对象的指针。

**weak\_ptr**：解决shared\_ptr相互（循环）引用的问题（参见muduo库）。shared\_ptr可以直接赋值给它，它可以通过lock函数来获得shared\_ptr。(通常搭配shared\_ptr使用)

详情见：<https://www.cnblogs.com/TenosDoIt/p/3456704.html>。

基于引用计数的智能指针简单实现：

#include <iostream>

#include <memory>

template<typename T>

class SmartPtr

{

pivate:

T\* \_ptr;

sze\_t\* \_count;

public:

SmartPtr(T\* ptr=nullptr):\_ptr(ptr)

{

if(\_ptr)

{

\_count=new size\_t(1);

}

else

{

\_count=new size\_t(0);

}

}

~SmartPtr()

{

(\*this->\_count)--;

if(\*this->\_count==0)

{

delete this->\_ptr;

delete this->\_count;

}

}

SmartPtr(const SmartPtr& ptr) //拷贝构造

{

if(this!=&ptr)

{

this->\_ptr=ptr.\_ptr;

this->\_count=ptr.\_count;

(\*this->\_count)++;

}

}

SmartPtr& operator=(const SmartPtr& ptr) //赋值运算符重载

{

if(this->\_ptr==ptr.\_ptr)

{

return \*this;

}

if(this->\_ptr)

{

(\*this->\_count)--;

if(this->\_count==0)

{

delete this->\_ptr;

delete this->\_count;

}

}

this->\_ptr=ptr.\_ptr;

this->\_count=ptr.\_count;

(\*this->\_count)++;

return \*this;

}

T& operator\*()

{

assert(this->\_ptr==nullptr);

return \*(this->\_ptr);

}

T\* operator->()

{

assert(this->\_ptr==nullptr);

return this->\_ptr;

}

size\_t use\_count()

{

return \*this->count;

}

};

详情见: <https://www.cnblogs.com/wxquare/p/4759020.html>。

### static关键字

作用：

1：隐藏（静态全局变量和函数）。加static前缀的全局变量和函数不具有全局可见性（源文件间）。

2：保持变量内容的持久（静态局部变量）。存储在静态存储区（另一个变量是static），会在程序开始就完成初始化。PS：如果static局部变量在函数中定义，它的生存期为整个源程序，但是作用域仍与自动变量相同。

3：默认初始化为0。

4：C++中类成员申明为static。（1）**类的静态成员函数是属于整个类而非类的对象，没有this指针，只能访问类的静态数据和静态成员函数**。（2）**不能将静态成员函数定义为虚函数**。（3）静态数据成员是静态存储的，**必须进行初始化**。<数据类型><类名>::<静态数据成员名>=<值>。（4）**防止父类的影响，可以在子类定义一个与父类相同的静态变量，以屏蔽父类的影响**。

详情见：<https://www.cnblogs.com/songdanzju/p/7422380.html>。

### static和const可以修饰同一个变量么

static const作为预编译声明使用时，**相当于C的#define**，但是需要编译器**进行类型检查**，保证程序的健壮性，此时只能用于基本数据类型。

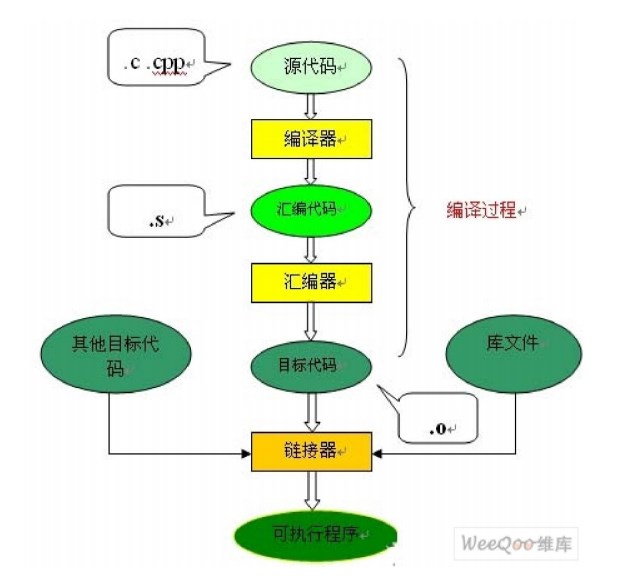
static const作为类的常量属性，因为常量区是在程序真正执行代码前进行初始化，所以其也必须是基本数据类型。

static const存在于常量区，const存在于动态区（stack、heap），static存在于静态区（全局区）。

详情见：<https://blog.csdn.net/sevenshal/article/details/9173861>。

### C/C++程序编译的整个过程

C/C++编译过程主要分为4个过程：1.编译预处理（处理以#开头的指令）。2.编译、优化阶段。3.汇编过程。4.链接程序。



详情见：<https://blog.csdn.net/u012662731/article/details/78520349>。

### 虚函数实现机制

C++中虚函数的作用主要是为了实现多态和继承（用父类型的指针指向其子类的实例对象，然后通过父类的指针调用子类的成员函数）的机制。

C++中虚函数是通过一张虚函数表来实现的。虚函数表主要是一个类的虚函数的地址表，这张表解决了继承、覆盖的问题。在有虚函数的类的实例中，这个表被分配在了这个实例对象的内存中。虚函数表在类继承中的会有所变化。

详情见：<https://blog.csdn.net/neiloid/article/details/6934135>。

### 析构函数为什么需要用虚函数

是因为多态的存在。当你的基类的析构函数不为虚的话，其子类中所有的成员变量的类中分配的内存也将可能泄漏。（父类行指针指向一个子类型对象，析构销毁时，调用的是父类型的析构函数，而子类中定义的成员变量内存泄漏）

例子见：<https://blog.csdn.net/yue_jijun/article/details/82290853>。

### 构造函数为什么不能为虚函数

1：从存储空间角度

虚函数对应一个vtale,这个表的地址是存储在对象的内存空间的。如果将构造函数设置为虚函数，就需要到vtable中调用，可是对象还没有实例化，没有内存空间分配，如何调用。（悖论）

2：从使用角度

**虚函数主要用于在信息不全的情况下，能使重载的函数得到对应的调用**。构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。

3：构造函数不需要是虚函数，也不允许是虚函数，因为创建一个对象时我们总是要明确指定对象的类型，尽管我们可能通过实验室的基类的指针或引用去访问它。但析构却不一定，我们往往通过基类的指针来销毁对象。这时候如果析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。

4：**从实现上看，vbtl在构造函数调用后才建立，因而构造函数不可能成为虚函数。**

从实际含义上看，在调用构造函数时还不能确定对象的真实类型（因为子类会调父类的构造函数）；而且构造函数的作用是提供初始化，在对象生命期只执行一次，不是对象的动态行为，也没有太大的必要成为虚函数

### 纯虚函数和虚函数

使用纯虚函数的类成为抽象类，该类不提供方法，只提供接口，它也是一种‘运行时多态’，可以理解为一个API。

虚函数是为了实现多态。

详情见：<https://www.cnblogs.com/steven66/p/5013695.html>。

### 虚函数表

每个包含了虚函数的类都包含一个虚表。

**虚表属于类，而不是某个具体的对象，一个类只需要一个虚表就行。为了指定对象的虚表，对象内部包含一个虚表的指针，来指向自己所使用的虚表。**

详情见：[https://blog.csdn.net/prieprime/article/details/80776625](https://blog.csdn.net/primeprime/article/details/80776625)。

### C++多态如何实现

在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据**对象**的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。

1：用virtual关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数。

2：存在虚函数的类都有一个一维的虚函数表，类的对象有一个指向虚表开始的虚指针。**虚表和类对应的，虚表指针是和对象对应的。**

3：多态是一个接口的多种实现。

4：多态用虚函数来实现，结合动态绑定。

5：抽象类是指至少包括一个纯虚函数的类。

详情见：<https://www.cnblogs.com/cxq0017/p/6074247.html>。

### C和C++区别

1：C语言是面向过程的编程，它的特点是函数，C++是面向对象的编程，类是它的主要特点。

2：C语言主要用于嵌入式领域、驱动开发等与硬件直接打交道的领域，C++可以用于应用层开发，用户界面开发等操作系统打交道的领域。

3：C++即继承了C强大的底层操作特性，又被赋予了面向对象机制。

4：C++是对C的“增强”，如：类型检查更严格。增加了面向对象的机制。增加了**泛型编程**的机制。增加了**异常**处理。增加了**运算符重载**。增加了**命名空间**，避免全局命名冲突。

详情见：<https://www.cnblogs.com/Amedeo/p/6246013.html>。

### i++和++i的区别

1：i++是指先使用i，后再将i的值加1；++i是将i的值先加1，然后在使用i。

2：如果i是一个**内建数据类型**时，效率没有区别；如果是**自定义数据类型**（包括迭代器）时，++i效率高。

详情见：<https://blog.csdn.net/xuantaiyang888/article/details/80886861>。

### C++中4种类型转换方式

1：static\_cast：可以实现C++中内置**基本数据类型**之间的相互转换。它不能进行无关类型（如非基类和子类）指针之间的转换。

例如：int c=static\_cast<int>(7.987);

2：const\_cast：const\_cast操作不能在不同的种类间转换。相反，它仅仅把一个它作用的表达式**转换成常量**。它可以使一个本来不是const类型的数据转换成const类型，或者吧const属性去掉。

3：reinterpret\_cast：它可以转换任何内置的数据类型为其他任何的数据类型，无需考虑类型安全或者常量的情形。

4：dynamic\_cast：其他三种都是编译时完成的，它是运行时处理的。

详情见：<https://www.cnblogs.com/carsonzhu/p/5251012.html>。

### C++常用的设计模式

**工厂模式**；简单工厂模式、工厂方法模式、抽象工厂模式。

简单工厂模式：主要特点是需要在工厂类中做判断，从而创造相应的产品，当增加新产品是，需要修改工厂类。

工厂方法模式：是指定义一个创建对象的接口，让子类决定实例化那一个类，Facktory Method使一个类的实例化延迟到其子类。

抽象工厂模式：提供一个创建一系列相关或互相依赖的对象接口，而无需指定它们的具体类。

**策略模式**：是指定义一系列的算法，把它们一个个分装起来，并且是它们可以互相替换。使得算法可以独立于使用它们的客户而变化，也就是说这些算法所完成的功能是一样的，只是各自现实上存在差异。（泛型算法）

**适配器模式**：将一个类的接口转换成客户希望的另一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。（容器适配器）

**单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。参见muduo中base，Singelton类。**

详情见：<https://www.cnblogs.com/chengjundu/p/8473564.html>。

### C/C++内存模型，每个区放什么

C分为四个区：堆、栈、静态全局变量区、常量区。

由C++内存分为以下几部分

1：栈区（stack）-由编译器自动分配释放，存放函数参数值，局部变量等。其操作类似于数据机构中栈。

2：堆区（heap）-一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中堆是两回事，分配方式类似于链表。

3：全局区（静态区static）-全局变量和静态变量存取释放在一快，初始化全区变量和静态变量在一块区域。未初始化全局变量和未初始化静态变量在相邻另一块区域，程序结束后由系统释放。

4：文字常量区（coment）-常量字符串就是放在这里。程序结束后由系统释放

5：程序代码区（code）-存放函数体二进制代码。

详情见：<https://www.cnblogs.com/Stultz-Lee/p/6751522.html>。

### 内存中的栈和堆分配

C分为四个区：堆、栈、全局（静态）变量区、常量区。

C++内存分为以下几部分

1：栈区（stack）-由编译器自动分配释放，存放函数参数值，局部变量等。其操作类似于数据机构中栈。

2：堆区（heap）-一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中堆是两回事，分配方式类似于链表。

3：全局区（静态区static）-全局变量和静态变量存取释放在一快，初始化全区变量和静态变量在一块区域。未初始化全局变量和未初始化静态变量在相邻另一块区域，程序结束后由系统释放。

4：文字常量区（coment）-常量字符串就是放在这里。程序结束后由系统释放

5：程序代码区（code）-存放函数体二进制代码。

**区别：**

1：申请方式

stack:

由**系统自动分配**。 例如，声明在函数中一个局部变量 int b; 系统自动在栈中为b开辟空间

heap:

需要**程序员自己申请**，并指明大小，在c中malloc函数

如p1 = (char \*)malloc(10);

在C++中用new运算符

如p2 = (char \*)malloc(10);

但是注意p1、p2本身是在栈中。

**2：申请后系统响应**

栈：只要栈剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址链表，当系统收到程序申请时，会遍历该链表，寻找第一个空间大于所申请空间堆结点，然后将该结点从空闲结点链表中删除，并将该结点空间分配给程序，另外，对于大多数系统，会在这块内存空间中首地址处记录本次分配大小，这样，代码中delete语句才能正确释放本内存空间。另外，由于找到堆结点大小不一定正好等于申请大小，系统会自动将多余那部分重新放入空闲链表中。

**3：申请大小限制**

**栈：在Windows下,栈是向低地址扩展数据结构，是一块连续内存区域**。这句话意思是栈顶地址和栈最大容量是系统预先规定好，在WINDOWS下，栈大小是2M（也有说是1M，总之是一个编译时就确定常数），如果申请空间超过栈剩余空间时，将提示overflow。因此，能从栈获得空间较小。

**堆：堆是向高地址扩展数据结构，是不连续内存区域**。这是由于系统是用链表来存储空闲内存地址，自然是不连续，而链表遍历方向是由低地址向高地址。堆大小受限于计算机系统中有效虚拟内存。由此可见，堆获得空间比较灵活，也比较大。

4：申请效率比较

**栈由系统自动分配，速度较快。但程序员是无法控制。**

**堆是由new分配内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。**

另外，在WINDOWS下，最好方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈是直接在进程地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。

5：存储内容

栈： 在函数调用时，第一个进栈是主函数中后下一条指令（函数调用语句下一条可执行语句）地址，然后是函数各个参数，在大多数C编译器中，参数是由右往左入栈，然后是函数中局部变量。注意静态变量是不入栈。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存地址，也就是主函数中下一条指令，程序由该点继续运行。

堆：一般是在堆头部用一个字节存放堆大小。堆中具体内容有程序员安排。

详情见：<https://www.cnblogs.com/yyxt/p/4268304.html>。

### const和#define的区别

区别：

1：就作用的阶段而言：#define是在编译的预处理阶段起作用，const是在编译、运行的时候起作用。

2：就起作用的方式而言：#define只是简单的字符串互换，没有类型检查，而const有对应的数据类型，是要进行判断的，可以避免一些低级的错误。

3：就存储方式而言：#define只是进行展开，有多少地方使用，就替换多少次，它定义的宏常量在内存中有若干个备份，const定义的只读变量在程序运行过程中只有一份备份。

4：从代码调试的方便程度而言，const常量可以进行调试的，#define不可以，因为在预编译阶段就已经替换了。

const优点：

1：const常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。

2：有些集成化的调试工具可以对const常量进行调试，但是不能对宏常量进行调试。

3：const可节省空间，避免不必要的内存分配，提高效率。

详情见：<https://blog.csdn.net/weibo1230123/article/details/81981384>。

### 指针和引用的区别

每一种编程语言都使用指针，而不同的是c++将指针暴露给了用户（程序员）。

**区别：**

1：现象上看：指针在运行是可以改变其所指向的值，而引用一旦和某个对象绑定后就不再改变。

2：从内存分配上看，程序为指针变量分配内存区域，而**不为引用分配内存区域**。

4：**不存在指向空值的引用**。

5：理论上，对于指针的级数没有限制，但是引用只能是一级。

**特别之处：const**

**指针传递和引用传递**

常量指针VS常量引用

指针常量VS引用常量

常量指针常量VS常量引用常量

指针传递和引用传递

详情见：<https://www.cnblogs.com/gxcdream/p/4805612.html>。

### 顶层const和底层const

待加

### 静态绑定和动态绑定是怎么实现的

1：对象的静态类型：对象在声明是采用的类型。是在编译期确定的。

2：对象的动态类型：目前所指对象的类型。是在运行期决定的。对象的动态类型可以更改，但是静态类型无法更改。

3：静态绑定：绑定的是对象的静态类型，某特性（比如函数）依赖于对象的静态类型，发生在编译期。

4：动态绑定：绑定的是对象的静态类型，某特性（比如函数）依赖于对象的动态类型，发生在运行期。

**区别：**

1：静态绑定发生在编译期，动态绑定发生在运行期；

2：**对象的动态类型可以更改，但是静态类型无法更改；**

3：**想要实现多态，必须使用动态绑定；**

4：**在继承体系中只有虚函数使用的是动态绑定，其他都是静态绑定。**

详情见：<https://www.cnblogs.com/lizhenghn/p/3657717.html>。

### 内存溢出和内存泄漏

内存溢出：是指程序在申请内存时，没有足够的内存空间供其使用。

内存泄漏：是指程序在申请内存后，无法释放已申请的内存空间。

内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。

  内存泄漏是指你向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。

### 面向对象理解

面向对象编程，即OPP，是一种编程范式，满足面向对象编程的语言，一般会提供类、封装、继承等语法和概念来辅助我们进行面向对象编程。

**四大特征：继承、多态、封装、抽象。**

**继承：子类继承父类的特征和行为。**子类可以有父类的方法，属性（非private）。子类也可以对父类进行扩展，也可以重写父类的方法。缺点就是提高了代码之间的耦合。final可以修饰类，方法，属性，修饰的类不能继承，修饰的方法不能重写，修饰的属性不能修改。

**多态：多态就是不同继承类的对象，对同一消息做出不同响应就叫多态。**

**封装：**封装就是把**过程和数据包围起来**，对数据的访问只能通过特定的界面（**接口**）。**能降低耦合性**。

**抽象：**分为过程抽象和数据抽象。

详情见：<https://www.cnblogs.com/msn-z/p/7802576.html>。

### 重载、重写和隐藏

**重载：**是指同一可访问区内被声明几个具有不同参数列的同名函数，根据参数列表确定调用哪个参数，重载不关心函数返回类型。

**隐藏：**是指派生类的函数屏蔽了与其同名的基类函数，主要只要同名函数，不管参数列表是否相同，基类函数都会被隐藏。

**重写(覆盖)：**是指派生类中存在重新定义的函数。**其函数名，参数列表，返回值类型，所有都必须同基类中被重写的函数一致，只有函数体不同**，派生类调用时会调用派生类的重写函数，不会调用被重写函数。重写的基类中被重写的函数必须有virtual修饰。

**重写和重载的区别：**

1：范围区别：重写和被重写的函数在不同的类中，重载和被重载的函数在同一类中。

2：参数区别：重写与被重写函数的函数参数列表一定相同，重载和被重载的函数参数列表一定不同。

3：virtual的区别：重写的基类必须有virtual修饰，重载函数和被重载函数可以被virtual修饰，也可以没有。

**隐藏和重写，重载的区别：**

1：与重载范围不同，隐藏函数和被隐藏函数在不同类中。

2：参数的区别，隐藏函数和被隐藏函数参数列表可以相同，也可以不同，但函数名一定相同，当参数不同时，无论基类中的函数是否被virtual修饰，积累函数都是被隐藏，而不是重写。

详情见：<https://www.cnblogs.com/zhangjxblog/p/8723291.html>。

### new/delete和malloc/free的用法和区别

new/delete用法：

int \*pi=new int；

int \*pi=new int()；

int \*pi=new int (1024)；

delete pi；

此时pi尽管没有定义，但仍然存在了它所指向对象的地址，然而pi所指向的内存已经被释放，因此pi不再有效。建议一旦删除指针所指向的对象，立即将指针置位0。

malloc和free用法：

函数原型：

void \*malloc(size\_t size）；

void free(void \*pointer);

语法：

int \*p=(int \*)malloc(100);

free(p);

**区别：**

1：**属性**---new/delete是C++**关键字**，需要编译器支持；malloc/free是**库函数**，需要头文件支持C。

2：**参数**---使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。malloc则需要显示地指出所需内存的尺寸。

3：**返回类型**---new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符，而malloc内存分配成功则是返回void\*，需要通过**强制类型转换**将void\*指针转换成我们需要的类型。

4：**分配失败**---new内存分配失败时，会抛出bac\_alloc异常。malloc分配内存失败时返回NULL；

5：**自定义类型实现流程-**--new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现），然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。Malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构函数。

6：**重载**---C++允许重载new/delete操作符，而malloc不允许重载。

PS：C++中，内存区分为5个区，堆、栈、全局/静态区、文字常量区、程序代码区。而C内存分为堆、栈、全局/静态区、文字常量区，文字常量区。

详情见：<https://www.cnblogs.com/maluning/p/7944231.html>。

### C++11新特性-待加

lambda表达式。1：可以定义匿名函数。2：编译器会把其转换成函数。3：“闭包”限制了别人的访问，更私有。

auto（自动类型推导，也是新的关键字）和decltype（可以从一个表达式中“俘获”其结果的类型并“返回”）。

统一初始化方式，大括号{}统一天下。

**delete函数**（=delete禁止拷贝）和**default函数**（=default 指示编译器生成该函数的默认实现）。

nullptr（空指针常量）。(0,NULL)

**右值引用-**右值引用类型，它是绑定到右值的，如临时对象或字面量。增加右值引用的主要原因是为了**实现move语义**。

C++11的标准库-除TR1包含的新容器(unordered\_set,unordered\_map,unordered\_multiset和unordered\_multimap),还有一些新的库，如正则表达式，tuple，函数对象封装器等。（线程库、新的智能指针类、新的算法）

序列for循环-例如for（auto p:m）。

详情见：<https://blog.csdn.net/caogenwangbaoqiang/article/details/79438279>。

**std::placeholds占位符**

std::function和std::bind工具使用

**std::function类模版**是一种通用、多态的函数封装。std::function对象是对C++中现有的可调用实体的一种类型安全的包裹

**std::bind函数将可调用对象(用法中所述6类)和可调用对象的参数进行绑定**，返回新的可调用对象(std::function类型，参数列表可能改变)

详情见：<https://blog.csdn.net/u014303647/article/details/88362337>。

and：<https://blog.csdn.net/u014303647/article/details/88362791>。

### C++左值、右值引用和move语义

左值：既可以放在等号左侧，也可以放在等号右侧的变量。

右值：只可以放在等号右侧的变量。

综上：“**左值持久，左值短暂**”。左值持久存在，右值将在运算结束后销毁。

move语义：所有权转移。

详情见：<https://blog.csdn.net/gg_18826075157/article/details/71889042>。

### vector

**介绍：**

1：vector表示可变大小数组的序列容器。

2：vector也采用**连续的存储空间**来存储元素。

3：vector使用**动态分配数组**来存储它的元素。

4：vector空间分配策略-vector会额外分配一些空间以适应可能的增长。

5：vector**占用了更多的存储空间（比实际存储的空间）**，为了获得管理存储空间的能力，并且以一种有效的方式动态增长。

6：与其他动态序列容器相比（deques，lists，forward\_lists）v**ector在访问元素的时候更加高效，对于其它不在末尾的删除和插入操作，效率更低。**

详情见：<https://www.cnblogs.com/zhonghuasong/p/5975979.html>。

vector简单实现见：

#include <iostream>

#include <cstring>

using namespace std;

template <typename T>

class Vector

{

public:

Vector(int size=0):theSize(size),theCapaticty(1+SPACE\_CAPACOTY)

{

objects=new T[theCapacity];

}

//拷贝构造

Vector(const Vector& rhs):obsjects(NULL)

{

Operator=(rhs);

}

~Vector()

{

delete[] objects;

}

//重载=运算符

const Vector& operator=(const Vector& rhs)

{

theCapacity=rhs.theCapacity;

theSize=rhs.theSize;

objects=new objects[this->theCapacity];

for(int i=0;i<this->theSize;i++)

objects[i]=rhs.objects[i];

return \*this;

}

//设置容器元素数目

void resize(int newSize)

{

if(newSize>theCapacity)

reserve(newSize\*2+1);

theSize=newSize;

}

//分配容器容纳内存大小

void reserve(int newCapacity)

{

if(newCapacity<theSize)

return;

T\* oldObject=objects;

objects=new T[newCapacity];

theCapacity=newCapacity;

for(int i=0;i<theSize;i++)

objects[i]=oldObject[i];

delete[] objects;

}

//重载[]运算符

T& operator[](int index)

{

return \*(objects+index);

}

const T& operator[](int index)const

{

return \*(objects+index);

}

bool isEmpty() const

{

return theSize==0;

}

int capacity() const

{

return theCapacity;

}

int size() const

{

return theSize;

}

void push\_back(T t)

{

if(theSize==theCapacity)

reserve(theCapacity\*2+1);

Objects[theSize++]=t;

}

void pop\_back()

{

theSize--;

}

T& back()

{

return object[theSize-1];

}

const T& back() const

{

return objects[theSize-1];

}

//迭代器类，非私有

typedef T\* iterater;

typedef const T\* const\_iterater;

iterater begin()

{

return objects;

}

const\_iterater begin() const

{

return objects;

}

iterater end()

{

return (objects+theSize);

}

const\_iterater begin() const

{

return (objects+theSize);

}

enum {SPACE\_CAPACITY=16};

private:

T\* objects;

int theSize;

int theCapacity;

};

vector简单实现见：<https://blog.csdn.net/u011408355/article/details/47957481>。

### list

list是一个线性双向链表，它的**每个节点都包括一个信息快、一个前驱、一个后驱指针**。**它存储在非连续的内存空间中**，并且由指针将有序的元素连接起来。

**特点：**

1：**不使用连续的内存空间，这样可以随意进行动态操作**。

2：可以在内部任何位置快速插入或删除，当然也可以在两端push和pop。

3：**不能进行内部的随机访问**，不支持[]操作和vector.at();

4：**相对于vector占用更多的内存。**

**list简单实现：**

//#program once：避免同一个文件被include多次。

//详情见：<https://blog.csdn.net/fengbingchun/article/details/78696814>。

#program once

#include<list>

Template<class T>

Struct \_ListNode

{

\_ListNode<T>\* \_next;

\_ListNode<T>\* \_prev;

T \_data;

\_ListNode(const T& x)

:\_data(x)

:\_next(NULL)

:\_prev(NULL);

{}

};

//迭代器类实现

template<class T,class Ref,class Ptr>

struct \_ListIterator

{

typedef \_ListNode<T> Node;

typedef \_ListIterator<T,Ref,Ptr> Self;

Node\* \_node;

\_ListIterator(Node\* x)

:\_node(x)

{}

Ref operator\*()

{

return \_node->\_data;

}

Ref operator\*()

{

return \_node->\_data;

}

Ptr operator->()

{

return &(operator\*());

}

bool operator!=(const Self& s)

{

return \_node!=s.\_node;

}

bool operator==(const Self& s)

{

return \_node==s.\_node;

}

Self& operator++()

{

\_node=\_node->\_next;

Return \*this;

}

Self operator++(int)

{

Self tmp(\*this);

\_node=\_node->\_next;

}

Self operator—()

{

\_node=\_node->\_prev;

return \*this;

}

Self operator—(int)

{

Self tmp(\*this);

\_node=\_node->\_prev;

return tmp;

}

};

template<class T>

class List

{

Typedef \_ListNode<T> Node;

Public:

Typedef \_ListIterator<T,T&,T\*> Iterator;

Typedef \_ListIterator<T,const T&,const T\*> ConstIterator;

List()

{

\_head=new Node(T());

\_head->new=\_head;

\_head->\_prev=\_head;

}

~List()

{

Clear();

delete \_head;

\_head=NULL;

}

void Clear()

{

Iterator it=begin();

while(it!=end())

{

Node\* del=it.\_node;

++it;

delete del;

}

\_head->\_next=\_head;

\_head->\_prev=\_head;

}

Iterator Begin()

{

return \_head->\_next;

}

Iterator End()

{

return \_head;

}

ConstIterator Begin()

{

return \_head->\_next;

}

ConstIterator End()

{

return \_head;

}

void PushBack(const T& x)

{

insert(End(),x);

}

void PushFront（const T& x）

{

Insert(Begin(),x);

}

void PopBack()

{

Erase(--End());

}

Void PopFront()

{

Erase(Begin());

}

Void PushFront()

{

Erase(Begin());

}

Void Insert(Iterator pos,const T& x)

{

Node\* cur=pos.\_node;

Node\* prev=cur->\_prev;

Node\* tmp=new Node(x);

prev->next=tmp;

tmp->\_next=prev;

tmp->next=cur;

cur->\_prev=tmp;

}

Iterator Erase(Iterator& pos)

{

Assert(pos->\_node!=\_head);

Node\* cur=pos->node:

Node\* prev=cur->\_prev;

Node\* next=cur->next;

prev->\_next=next;

next->\_prev=prev;

delete cur;

pos=next;

return next;

}

void Assigh(size\_t n,const T& val)

{

Clear();

for(size\_t i=0;i<n;++i)

{

PushBack(val);

}

}

Template<class InputIterator>

void Assigh(InputIterator first,InputIterator last)

{

Clear();

while(fist!=last)

{

PushBack(\*first);

++first;

}

}

protected:

Node\* \_head;

};

详情见：<https://blog.csdn.net/ego_bai/article/details/80206524>。

### deque

deque的元素数据采用分块的线性结构进行存储，deque分为若干线性存储块，成为deque块。块的大小一般为512个字节，元素的数据类型所占字节数决定每个deque块可容纳元素的个数。

所有的deque块使用一个map块管理。

**特性：**

1：与vector容器一样，在deque容器的中间insert或erase元素效率比较低。

2：不同于vector容器，deque容器提供高效的在其首部insert或erase操作，就像在容器尾一样。

3：与vector容器一样而不同于list容器的是，**deque容器支持对所有元素的随机访问**。

4：**在deque容器首部或尾部插入元素不会使任何迭代器失效，而在首部或尾部删除元素则只会使指向该元素的迭代器失效。在deque容器的任何其他位置的插入与删除操作将使指向该容器元素的所有迭代器失效。**

详情见：<http://www.cnblogs.com/SWiper/p/6617774.html>。

### stl中set、map底层实现，与hashtalbe差异

map是关联容器，以键值对的形式进行存储，方便进行查找。关键词起到索引的作用，值则表示与索引相关联的数据，以红黑树的结构实现，插入删除等操作都是在O(logn)时间内完成。

set是关联容器，set中每个元素只包含一个关键字，set支持高效的关键字查询操作—检查一个给定的关键字是否在set中，set也是以红黑树的结构实现的，支持高效插入、删除等操作。

**常见问题：**

1：为何map 和set的插入效率比用其他容器高？

C++ STL中标准关联容器set, multiset, map, multimap内部采用的就是一种非常高效的平衡检索二叉树：红黑树，也成为RB树(Red-Black Tree)。RB树的统计性能要好于一般的平衡二叉树，很简单，因为对于关联容器来说，不需要做内存拷贝和内存移动。

**map和set容器内所有元素都是以红黑树节点（如下）的方式来存储，其节点，结构和链表差不多，指向父节点和子节点。**

2：为何每次insert之后，以前保存的iterator不会失效?

**iterator这里就相当于指向节点的指针，内存没有变，指向内存的指针怎么会失效呢。**

相对于vector来说，每一次删除和插入，指针都有可能失效，调用push\_back在尾部插入也是如此。因为为了保证内部数据的连续存放，iterator指向的那块内存在删除和插入过程中可能已经被其他内存覆盖或者内存已经被释放了。即使时push\_back的时候，容器内部空间可能不够，需要一块新的更大的内存，只有把以前的内存释放，申请新的更大的内存，复制已有的数据元素到新的内存，最后把需要插入的元素放到最后，那么以前的内存指针自然就不可用了。

3：map和set不能像vector一样有个reserve函数来预分配数据？

map和set内部存储的已经不是元素本身了，而是包含元素的节点。

详情见：<https://www.cnblogs.com/LUO77/p/5774476.html>。

源码分析--详情见：<https://blog.csdn.net/terence1212/article/details/52487656>。

**hashtable、hashset、hashmap、set、map区别**

hashtable：可以提供任何有名项的存取和删除操作，由于操作对象都是有名项，故可被视为一种字典结构。 用意是在常数时间内提供基本操作。主要思想是通过hash函数，把对象银蛇到一个较小的容器里面，并且保证时间复杂度。

hashset重点在于set几个字母上，主要实现集合功能，只不过底层函数由hashtable提供，而set是由红黑树做提供，两者唯一的区别就是set默认是排好序的，而hashset不是。

Hashmap重点在于map几个字母上，主要提供对象间的一一映射功能，区别同上。

**总结：加有hash几个字的一些结构，底层都是由hashtalbe来提供的，不加的都是由红。**

**黑树来提供。**

详情见：<http://blog.sina.com.cn/s/blog_178a6cbd50102xwzh.html>。

### read函数和fread函数

read：ssize\_t read（int fd，void\* buf，size\_t count）；

read用于从文件描述符对应的文件读取数据，调用成功返回读出的字节数；buf为读取数据的缓冲区，count为每次读取的字节数，出错返回-1，EOF返回0；

fread：size\_t fread（void\* ptr，size\_t size，size\_t nmemb，FILE\* stream）；

ptr为指向缓冲区保存或读取的数据，size为控制记录大小，nmemb为记录数，函数返回读取或会写的记录数。

**差别：**

**效率：fread为封装好的库函数，而read为系统函数，一般来说，fread效率更高。**

**读取文件差别：fread功能更强大，可以读结构体的二进制文件。如果是底层的操作，用到文件描述符，read更好。**

详情见：<https://blog.csdn.net/qq_33832591/article/details/52268477>。

### C++异常安全

“异常安全”并不是说不能抛出异常，更不是说“非常安全的意思”，而是说即使在抛出异常的时候，程序回退的过程是干净的。

“异常安全”分为三个级别：

1：基本级别：可能发生异常，但是彼时代码保证做了任何必要的清理工作。

2：事物级别：可能发生异常，但是彼时代码保证他对数据的任何修改都被回滚。

3：没有异常：不可能发生异常。

异常安全函数即使发生了异常也不会泄露资源或允许任何数据结构破坏。这样的函数区分为三种可能的保证：基本型、强烈型、不抛异常性。

详情见：<https://www.cnblogs.com/feng9exe/p/7245542.html>。

### strcpy和memcpy区别

复制的内容不同：strcpy只能复制字符串，而且还会复制字符串的结束符。而memcpy可以复制任意类型的内容。

复制的方法不同：strcpy不需要指定长度，遇到结束符‘\0’才会结束，所以容易溢出。memcpy则是根据第三个参数决定复制的长度。

用途不同：通常在复制字符串时用strcpy，在复制其他类型数据时一般用memcpy。

### 静态链接库和动态链接库

这两者都是共享代码的一种方式。（节省空间）

**静态链接库的优点：**

1：代码装载速度块，执行速度略比动态链接库快。**原因：**你一开始就调用了整个静态链接库，so，不用一个个装载当前用的函数或者方法；

2：只需保证开发者的计算机中有正确的.LIB文件，再已二进制形式发布程序时不再考虑在用户的计算机上.LIB文件是否存在及版本问题。

**动态链接库的优点：**

1：更加节省内存并减少页面交换。**原因：**仅仅调用自己需要的，不需要的就不调用，所以减少内存（减少页面交换同理）；

2：DLL文件与EXE文件独立，只要输出接口不变，更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性。

3：不同的编程语言编写的程序只要按照函数调用约定就可以调用同一个DLL函数。

4：.DLL适用于大规模的软件开发，使开发过程独立、耦合度小，便于不同开发者和开发组织之间进行开发和测试。

不足：

1：使用静态链接生成的可执行文件体积较大，包含相同的公共代码，造成浪费。

2：使用动态链接库的应用程序不是自完备的，它依赖的DLL模块也要存在，

简单来说，就是：.LIB用空间交换时间，.DLL用时间交换空间。

详情见：<https://blog.csdn.net/qq_33346538/article/details/77506517>。

### 优先队列

## 数据结构和算法

### 几种重要的排序

八大排序算法总体比较：



**快排**

1：基本思想-选择一个基准元素，将比基准元素小的元素放在其前面，比基准元素大的元素放在其后面，然后:再将小于其基准值元素的子数列和大于基准元素的子数列按原来的方法排序。

2：优点-极快，数据移动少。缺点-不稳定。

3：效率分析-此排序算法的效率在序列越乱的时候，效率越高。在数据有序时，会退化成冒泡排序。

4：对于基准的选择-三数（low、mid、high）中取、随机选取基准。

5：优化方法-a.当待排序序列的长度分割到一定大小后，使用插入排序（对于很小和部分有序的数组，快排不如[插排](https://www.baidu.com/s?wd=%E6%8F%92%E6%8E%92&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)好）。b.再一次分割结束后，可以把与key相等的元素聚集在一起，继续下次分割时，不必再对于key相等的元素分割。

6：应用场景：求数组中第k小的数。

7：实现

（1）挖坑法

#include<iostream>

using namespace std;

int Partition(int arr[],int low,int high)

{

int x=arr[hight];

int i=low-1;

for(int j=low;j<high;j++)

{

if(a[j]<x)

{

i++;

swap(arr[i],arr[j]);

}

}

swap(arr[++i],arr[hight]);

return i;

}

void QuickSort(int a[],int low,int high)

{

if(low<high)

{

int q=Partition(a,low,high);

QuickSort(a,low,q-1);

QuickSort(a,q+1,high);

}

}

8：单链表的快排实现（leetcode148）

struct Node

{

int key;

Node\* next;

Node(int nKey, Node\* pNext)

: key(nKey)

, next(pNext)

{}

};

Node\* GetPartion(Node\* pBegin, Node\* pEnd)

{

int key = pBegin->key;

Node\* p = pBegin;

Node\* q = p->next;

while(q != pEnd)

{

if(q->key < key)

{

p = p->next;

swap(p->key,q->key);

}

q = q->next;

}

swap(p->key,pBegin->key);

return p;

}

void QuickSort(Node\* pBeign, Node\* pEnd)

{

if(pBeign != pEnd)

{

Node\* partion = GetPartion(pBeign,pEnd);

QuickSort(pBeign,partion);

QuickSort(partion->next,pEnd);

}

}

**堆排序**

1：二叉堆定义-二叉堆是完全二叉树或近似完全二叉树。满足特性a.父节点的键值总大于或等于（小于或等于）任何一个子节点的键值。b.每个节点的左子树和右子树都是一个二叉堆。

2：堆的存储-一般用数组来表示堆，i结点的父节点下标是（i-1）/2，它的左右节点的下标分别是2\*i+1和2\*i+2。

3：应用-寻找M个数中前k个最小的数并保持有序。时间复杂度：O(K)[创建K个元素最大堆的时间复杂度] +（M-K）\*log(K)[对剩余M-K个数据进行比较并每次对最大堆进行从新最大堆化]。

4：实现（构建堆，调整堆）

#include<iostream>

using namespace std;

//构建堆

void adjust(vector<int>& arr,int len,int index)

{

int left=2\*index+1;

int right=2\*index+2;

int maxIdx=index;

if(left<len&&arr[left]>arr[maxIdx])

maxIdx=left;

if(right<len&&arr[right]>arr[maxIdx])

maxIdx=right;

if(maxIdx!=index)

{

swap(arr[maxIdx],arr[index];

adjust(arr,len,maxIdx);

}

}

//堆排序

void heapsort(vector<int>& arr,int size)

{

//构建大根堆

for(int i=size/2-1;i>=0;i--)

adjust(arr,size,i);

//调整大根堆

for(int i=size-1;i>=1;i--)

{

swap(arr[0],arr[i]);

adjust(arr,i,0);

}

}

**归并排序**

1：基本思想-首先将初始序列的n个记录看成n个有序的子序列，每个子序列的长度为1，然后两两合并，得到n/2个长度为2的有序子序列，…以此类推，直到得到一个长度为n的有序序列为止。

2：适用场景-若n较大，并且要求排序稳定，则可以选择归并排序。

3：实现

分治法

#include<iostream>

using namespace std;

void Merge(int arr[],int reg[],int start,int end)

{

if(start>=end) return ;

int len=end-start,mid=(len>>1)+start; //移位符在 二进制位值中 时常用

int start1=start,endl=mid;

int start2=mid+1,end2=end;

Merge(arr,reg,start1,end1);

Merge(arr,reg,start2,end2);

int k=start;

while(start1<=endl&&start2<=end2)

reg[k++]=arr[start]<arr[start2]?arr[start1++]:arr[start2++];

while(start1<=endl)

reg[k++]=arr[start1++];

while(start2<=end2)

reg[k++]=arr[start2++];

for(k=start;k<=end;k++)

arr[k]=reg[k];

}

void MergeSort(int arr[],const int len)

{

Int reg[len];

Merge(arr,reg,0,len-1);

}

**冒泡排序**

1：基本原理在要排序的一组数中，对目前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较，让较大的数往下沉，较小的往上冒。

2：优点:-稳定。缺点-慢。

3：实现

#include<iostream>

using namespace std;

void BubbleSort(int a[],int n)

{

bool flag=true;

int k=n;

while(flag)

{

flag=false;

for(int i=1;i<k;I++)

{

if(a[i-1]>a[i])

{

swap(a[i-1],a[i]);

flag=true;

}

}

k--;

}

}

**直接插入排序**

1：基本思想-将一个记录插入到已排序好的有序表中，从而得到一个新的，记录数增1的有序表。

2：优点-稳定，快。缺点-比较次数不一定，比较次数较少，插入后点的数据移动较多，特别是数据量庞大的时候。

3：实现

#include<iostream>

using namespace std;

void InsertSort(int a[],int n)

{

int i,j;

for(i=1;i<n;i++)

{

for(j=i-1;j>=0;j--)

{

if(a[j+1]<a[j])

swap(a[j],a[j+1]);

}

}

}

**希尔排序**

1：基本思想-先将整个待排序元素序列分割成若干个子序列分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序时，在对全体元素进行一次直接插入排序（因为直接插入排序在元素基本有序的情况下，效率很高）。

2：适用场景-比较在希尔排序中是最主要的操作，不是交换。用已知最好的步长序列的希尔排序比直接插入排序要快，甚至在小数组中比快速排序和堆排序还快，但在涉及大量数据是希尔排序还是不如快排。  
 3：实现

#include<iostream>

using namespace std;

void ShellSort(int a[],int n)

{

int i,j,gap;

for(gap=n>>1;gap>0;gap>>1)

{

for(i=gap;j<m;j+=gap)

{

int x=a[i];

j=i-gap;

while(j>=0&&x<a[j])

{

a[j+gap]=a[j];

j-=gap;

}

A[j+gap]=x;

}

}

}

**简单选择排序**

1:实现

#include<iostream>

using namespace std;

void SelectSort(int a[],int n)

{

int i,j;

for(i=0;i<n;i++)

{

int minIndex=i;

for(j=i+1;j<n;j++)

{

if(a[j]<a[minIndex])

{

minIndex=j;

}

}

swap(a[i],a[minIndex]);

}

}

**交换排序**

详情见：<https://blog.csdn.net/m0_37962600/article/details/81475585>。

### AVL树

**定义：**AVL树本质上是一颗二叉搜索树，但是它又具有以下特点：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为平衡二叉树。

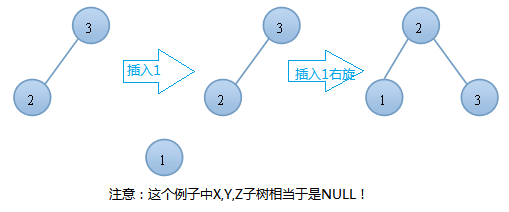
平衡因子(bf)：结点的左子树的深度减去右子树的深度，那么显然-1<=bf<=1(avl。

**作用：**减小各操作时间复杂度。

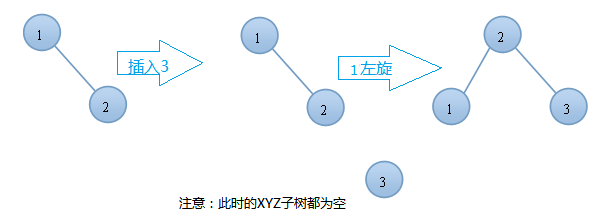
**操作**

单旋转：

右旋(左子树高于右子树，且在左子树的左子树插入)

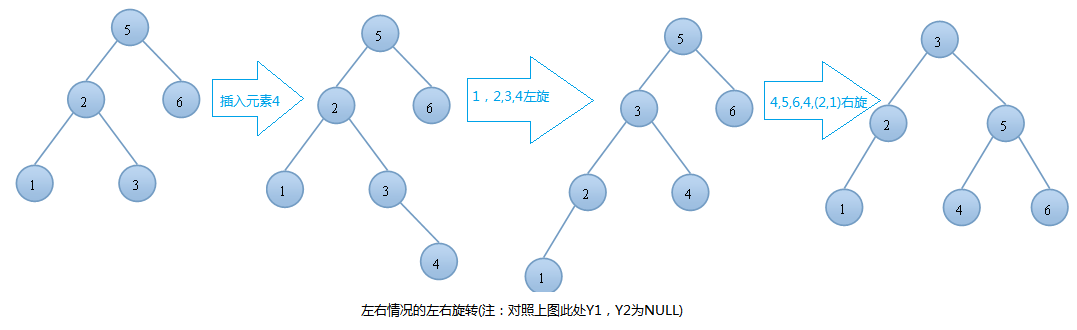


左旋(右子树高于左边子树，且在右子树的右子树插入)

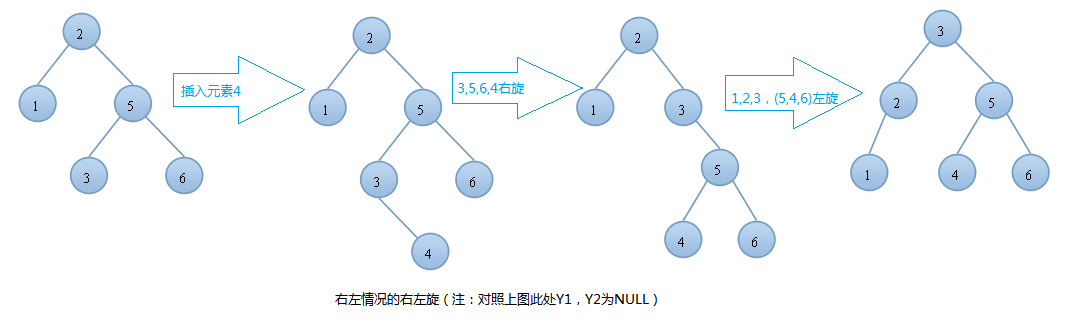


双旋转：

左右（先旋转左子树，后旋转右子树）旋(左子树高于右子树，且在左子树的右子树插入)



右左（先旋转左子树，后旋转右子树）旋(右子树高于左边子树，且在右子树的左子树插入)



AVL插入代码实现（了解哈。。。）

#include <iostream>

using namespace std;

#define DataType int

/\*

定义AVL树的结构体，链式

\*/

typedef struct AvlNode{

DataType data;

AvlNode \* m\_pLeft;

AvlNode \* m\_pRight;

int height;

}\*AvlTree,\*Position,AvlNode;

//求两个数的最大值

int Max(int a,int b)

{

return a>b?a:b;

}

//求树的高度

int Height( AvlTree T)

{

if(NULL == T)

return -1;

else

return T->height;

}

//单旋转右旋

AvlTree singleRotateWithRight(AvlTree T)

{

AvlTree L = T->m\_pLeft;

T->m\_pLeft = L->m\_pRight;

L->m\_pRight = T;

T->height = Max( Height(T->m\_pLeft),Height(T->m\_pRight) ) + 1;

L->height = Max( Height(L->m\_pLeft),Height(L->m\_pRight) ) + 1;

return L; //此时L成为根节点了（可参考AVL的插入的左左情况的右旋图）

}

//单旋转左旋

AvlTree singleRotateWithLeft(AvlTree T)

{

AvlTree R = T->m\_pRight;

T->m\_pRight = R->m\_pLeft;

R->m\_pLeft = T;

T->height = Max( Height(T->m\_pLeft),Height(T->m\_pRight) ) + 1;

R->height = Max( Height(R->m\_pLeft),Height(R->m\_pRight) ) + 1;

return R; //此时R成为根节点了（可参考AVL的插入的左左情况的左旋图）

}

//双旋转，先左后右

AvlTree doubleRotateWithLeft(AvlTree T) //先左后右

{

T->m\_pLeft = singleRotateWithLeft(T->m\_pLeft);

return singleRotateWithRight(T);

}

//双旋转，先右后左

AvlTree doubleRotateWithRight(AvlTree T) //先右后左

{

T->m\_pRight = singleRotateWithRight(T->m\_pRight);

return singleRotateWithLeft(T);

}

AvlTree AvlTreeInsert(AvlTree T, DataType x)

{

if(T == NULL) //如果树为空

{

T = (AvlNode \*)malloc(sizeof(struct AvlNode));

if(T)

{

T->data = x;

T->m\_pLeft = NULL;

T->m\_pRight = NULL;

T->height = 0;

}

else

{

cout << "空间不够" << endl;

exit(0);

}

}

else if( x < T->data) //如果插入到T结点的左子树上

{

T->m\_pLeft = AvlTreeInsert(T->m\_pLeft,x); //先插入，后旋转

if(Height(T->m\_pLeft) - Height(T->m\_pRight) == 2) //只有可能是这个

{

if(x < T->m\_pLeft->data) //左左情况，只需要右旋转

{

T = singleRotateWithRight( T );

}

else //左右情况，双旋转,先左

{

T = doubleRotateWithLeft( T );

}

}

}

else if( x > T->data )

{

T->m\_pRight = AvlTreeInsert(T->m\_pRight,x);

if(Height(T->m\_pRight) - Height(T->m\_pLeft) == 2)

{

if(x > T->m\_pRight->data) //右右情况，进行左旋

{

T = singleRotateWithLeft( T );

}

else //左右情况，双旋转,先右

{

T = doubleRotateWithRight( T );

}

}

}

//如果这个数已经存在，那么不进行插入

T->height = Max(Height(T->m\_pLeft),Height(T->m\_pRight)) + 1;

return T;

}

//递归实现中序遍历

void inOrderVisitUseRecur(const AvlTree pCurrent)

{

if(pCurrent)

{

inOrderVisitUseRecur(pCurrent->m\_pLeft);

cout << pCurrent->data << " ";

if(pCurrent->m\_pLeft)

cout << " leftChild: "<<pCurrent->m\_pLeft->data;

else

cout << " leftChild: "<<"NULL" ;

if(pCurrent->m\_pRight)

cout << " rightChild: "<<pCurrent->m\_pRight->data;

else

cout << " rightChild: "<< "NULL";

cout << endl;

inOrderVisitUseRecur(pCurrent->m\_pRight);

}

}

/\*

int main()

{

AvlTree root = NULL;

root = AvlTreeInsert(root,1);

root = AvlTreeInsert(root,2);

root = AvlTreeInsert(root,3);

root = AvlTreeInsert(root,4);

root = AvlTreeInsert(root,5);

root = AvlTreeInsert(root,6);

root = AvlTreeInsert(root,7);

root = AvlTreeInsert(root,8);

root = AvlTreeInsert(root,9);

root = AvlTreeInsert(root,10);

root = AvlTreeInsert(root,11);

root = AvlTreeInsert(root,12);

root = AvlTreeInsert(root,13);

root = AvlTreeInsert(root,14);

root = AvlTreeInsert(root,15);

inOrderVisitUseRecur(root);

return 0;

}

\*/

详情见：<https://www.cnblogs.com/zhuwbox/p/3636783.html>。

### B树和B+树

B树是为实现高效的**磁盘存取**而设计的多叉平衡搜索树。这个概念在文**件系统，数据库**系统中非常重要。

**B树**

**1：基本原理**

首先，简单说一下B树产生的原因。B树是一种查找树，我们知道，这一类树（比如二叉查找树，红黑树等等）最初生成的目的都是为了解决某种系统中，查找效率低的问题。B树也是如此，它最初启发于二叉查找树，二叉查找树的特点是每个非叶节点都只有两个孩子节点。然而这种做法会导致当数据量非常大时，二叉查找树的深度过深，搜索算法自根节点向下搜索时，需要访问的节点也就变的相当多。如果这些节点存储在外存储器中，每访问一个节点，相当于就是进行了一次I/O操作，随着树高度的增加，频繁的I/O操作一定会降低查询的效率。

这里有一个基本的概念，就是说我们从外存储器中读取信息的步骤，简单来分，大致有两步：

1：找到存储这个数据所对应的磁盘页面，这个过程是机械化的过程，需要依靠磁臂的转动，找到对应磁道，所以耗时长。

2：读取数据进内存，并实施运算，这是电子化的过程，相当快。

综上，对于外存储器的信息读取最大的时间消耗在于寻找磁盘页面。那么一个基本的想法就是能不能减少这种读取的次数，在一个磁盘页面上，多存储一些索引信息。B树的基本逻辑就是这个思路，它要改二叉为多叉，每个节点存储更多的指针信息，以降低I/O操作数。

**2：基本结构**

B树的定义：不同的资料在表述上有所差别。我在这里采用《算导》中的定义，用**最小度t**来定义B树。一棵最小度为t的B树是满足如下四个条件的平衡多叉树：

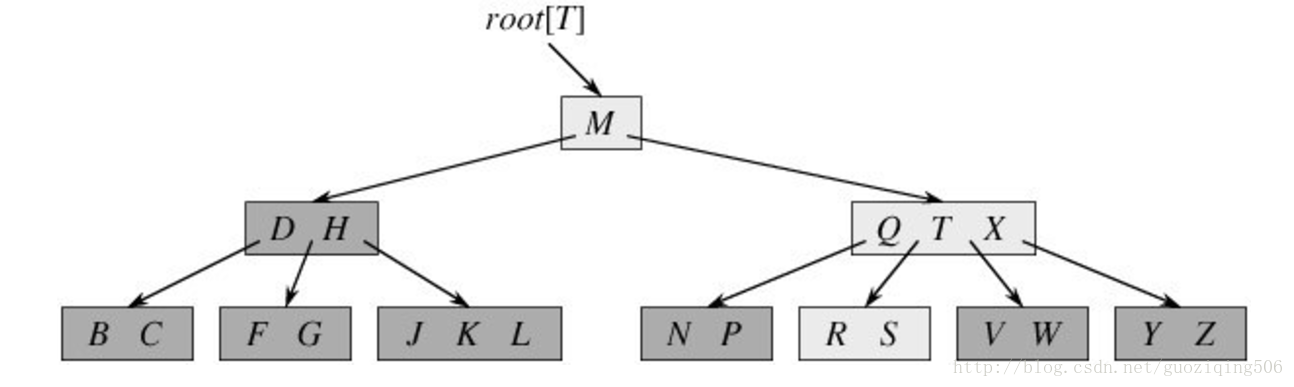
1：每个节点最多包含2t−1个关键字；除根节点外的每个节点至少有t−1个关键字（t≤2），根节点至少有一个关键字；

2：一个节点u中的关键字按非降序排列：u.key1≤u.key2≤…u.keyn ；

3：每个节点的关键字对其子树的范围分割。设节点u有n+1个指针，指向其n+1棵子树，指针为u.p1,…u.pn关键字ki为u.pi所指的子树中的关键字，有k1≤u.key1≤k2≤u.key2…成立；

4：所有叶子节点具有相同的深度，即树的高度h。这表明B树是平衡的。平衡性其实正是B树名字的来源，B表示的正是单词Balanced。

一个标准的B树如下图：



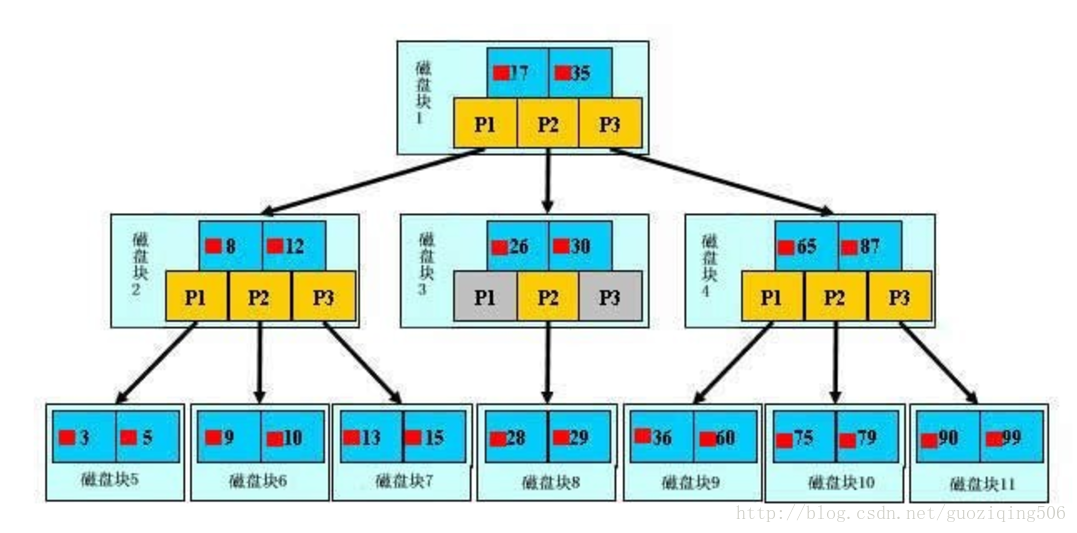
B树的高度

我直接给出结论了：对于一个包含n个关键字(n≥1)，最小度数t≥2的B树T，其高度h hh满足如下规律：

在搜索B树时，很明显，访问节点（即读取磁盘）的次数与树的高度呈正比，而B树与红黑树和普通的二叉查找树相比，虽然高度都是对数数量级，但是显然B树中log loglog函数的底可以比2更大，因此，和二叉树相比，极大地减少了磁盘读取的次数。

**3：搜索算法**

一棵已经建立好的B树如下图所示，我们的目的是查找关键字为29的文件：



先简单对上图说明一下：

1：图中的小红方块表示对应关键字所代表的文件的存储位置，实际上可以看做是一个地址，比如根节点中17旁边的小红块表示的就是关键字17所对应的文件在硬盘中的存储地址；

2：P是指针，不用多说了，需要注意的是：指针，关键字，以及关键字所代表的文件地址这三样东西合起来构成了B树的一个节点，这个节点存储在一个磁盘块上。

下面，看看搜索关键字的29的文件的过程：

1：从根节点开始，读取根节点信息，根节点有2个关键字：17和35。因为17 < 29 <35，所以找到指针P2指向的子树，也就是磁盘块3（1次I/0操作）

2：读取当前节点信息，当前节点有2个关键字：26和30。26 < 29 < 30，找到指针P2指向的子树，也就是磁盘块8（2次I/0操作）

3：读取当前节点信息，当前节点有2个关键字：28和29。找到了！（3次I/0操作）

由上面的过程可见，同样的操作，如果使用平衡二叉树，那么需要至少4次I/O操作，B树比之二叉树的这种优势，还会随着节点数的增加而增加。另外，因为B树节点中的关键字都是排序好的，所以，在节点中的信息被读入内存之后，可以采用二分查找这种快速的查找方式，更进一步减少了读入内存之后的计算时间，由此更能说明对于外存数据结构来说，I/O次数是其查找信息中最大的时间消耗，而我们要做的所有努力就是尽量在搜索过程中减少I/O操作的次数。

**4：B树中插入关键字**

向B树种插入关键字的过程与向二叉查找树中插入关键字的过程类似，但是要稍微复杂一点，因为根据上面B树的定义，我们可以看出，B树每个节点中关键字的个数是有范围要求的，同时，B树是平衡的，所以，如果像二叉查找树那样，直接找到相关的叶子，插入关键字，有可能会导致B树的结构发生变化而这种变化会使得B树不再是B树。

所以，我们这样来设计B树种对新关键字的插入：首先找到要插入的关键字应该插入的叶子节点（为方便描述，设这个叶子节点为u），如果u是满的（恰好有2t−1个关键字），那么由于不能将一个关键字插入满的节点，我们需要对u按其当前排在中间关键字u.keyt进行分裂，分裂成两个节点u1,u2 ；同时，作为分裂标准的关键字u.keyt会被上移到u的父节点中，在u.keyt插入前，如果u的父节点未满，则直接插入即可；如果u的父节点已满，则按照上面的方法对u的父节点分裂，这个过程如果一直不停止的话，最终会导致B树的根节点分裂，B树的高度增加一层。

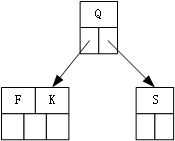
用《算导》中的一个题目展示一下这种插入关键字的过程：

现在我们要将关键字序列：F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y依次插入一棵最小度为2的B树中。也就是说，这棵树的节点中，最多有3个关键字，最少有1个关键字。

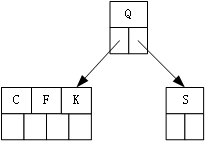
**第1步**，F, S, Q可以被插入一个节点（也就是根节点）

https://img-blog.csdn.net/20170320110812239?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvZ3VvemlxaW5nNTA2/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

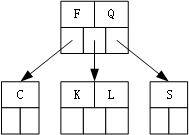
**第2步**，插入关键字K，因为节点已满，所以在插入前，发生分裂，中间关键字Q上移，建立了一个新的根节点：



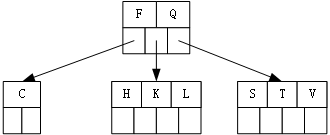
**第3步**，插入关键字C:



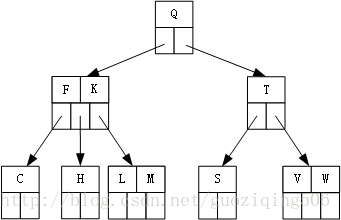
**第4步**，插入关键字L，L应该被插入到根节点的左侧的孩子中，因为此时该节点已满，所以在插入前，发生分裂：



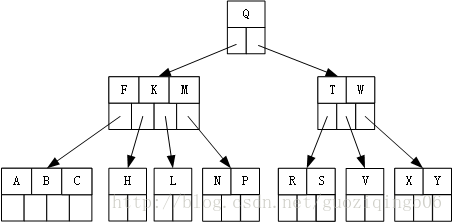
**第5步**，插入关键字H, T, V，这个过程没有发生节点的分裂：



**第6步**，插入关键字W，W应该被插入到根节点的最右侧的孩子中，因为此时该节点已满，所以在插入前，关键字T上移，最右端的叶子节点发生分裂：



**第7步**，插入关键字M，M应该被插入到根节点的左起第2个孩子中，因为此时该节点已满，所以在插入前，发生分裂，分裂之后，中间关键字K上移，导致根节点发生分裂，树高增加1：



**第8步**，同样的道理，插入关键字R, N, P, A, B, X, Y：最终得到的B树如下：

**5：B树中删除关键字**

删除操作的基本思想和插入操作是一样的，都是不能因为关键字的改变而改变B树的结构。插入操作主要防止的是某个节点中关键字的个数太多，所以采用了分裂；删除则是要防止某个节点中，因删除了关键字而导致这个节点的关键字个数太少，所以采用了合并操作。

下面分三种情况来讨论下删除操作是如何工作的，这个过程的顺序是自根节点起向下遍历B树

\*\*Case - 1：\*\*如果要删除的关键字k kk在节点u uu中，而且u uu是叶子节点，那么直接删除k；

\*\*Case - 2：\*\*如果要删除的关键字k kk在节点u uu中，而且u uu是内部节点，那么分以下3种情况讨论：

1：如果u中前于k的子节点u1中至少含有t tt个关键字，则找出k kk在以u1为根的子树中的前驱k′（前驱的意思是u1中比k kk小的关键字中最大的），然后在以u1为根的子树中删除k′，并在u中以k′替代 k；

2：如果上面的条件(1)不成立，也就是说，前于k:的子节点中关键字的个数小于t了，那么就去找后于k的子节点，记为u2。若u2中至少含有t tt个关键字，则找出k在以u2为根的子树中的后继k′(大于k kk的关键字中最小的)，然后在以u2为根的子树中删除k′，并在u中以k′替代k。可以看出（2）是（1）的一个对称过程；

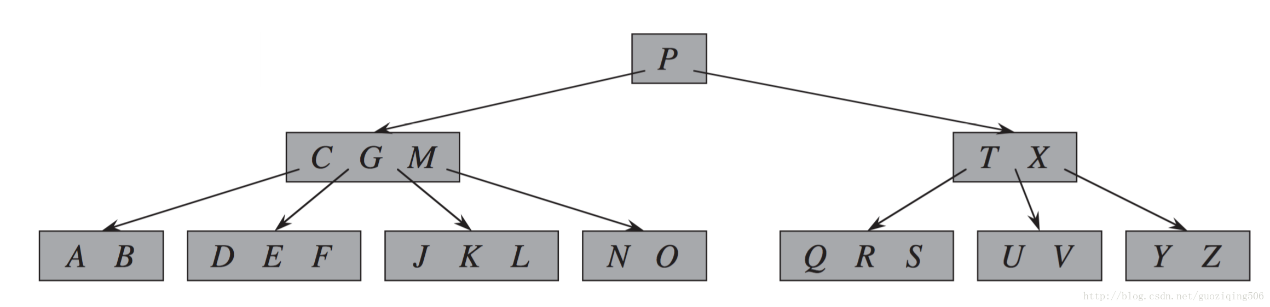
(3) 如果u1,u2中的关键字个数都是t−1 t - 1t−1，则将k和u2合并后并入u1，这样u就失去了k和指向u2的指针，最后递归地从u1中删除k。

\*\*Case - 3：\*\*如果要删除的关键字k不在当前节点u中，而且u是内部节点（如果自上而下扫描到叶子都没有这个关键字的话，那就说明要删除的关键字根本就不存在，所以此处只考虑u是内部节点的情况），则首先确定包含k的u的子树，我们这里设为u.pi。如果u.pi中至少含有t个关键字，那么继续扫描，寻找下一个要被扫描的子树；如果u.pi中只含有t−1 t - 1t−1个关键字，则需要分下面两种情况进行操作：

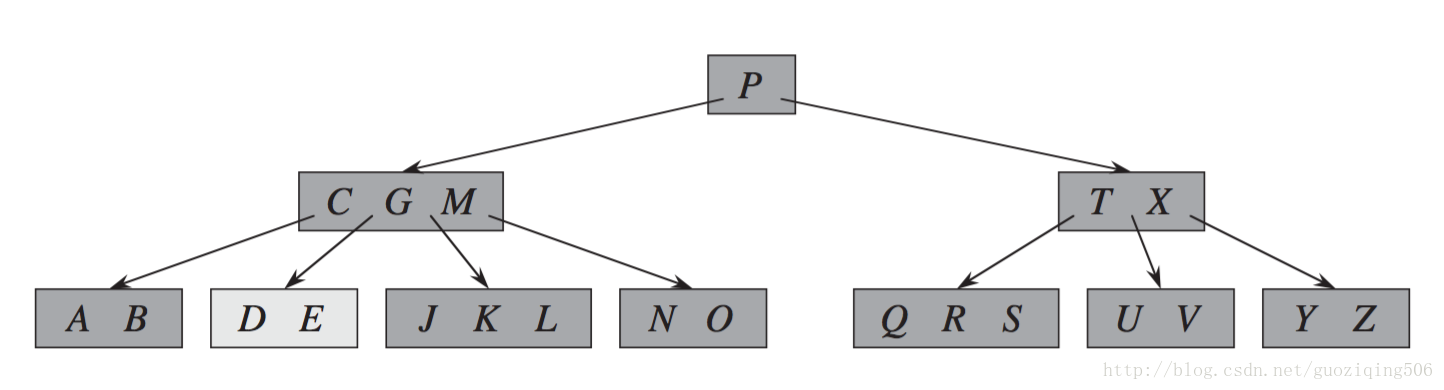
1：如果u.pi至少有一个相邻的兄弟比较“丰满”（即这个兄弟至少有t个关键字）。则将u uu中的一个关键字降至u.pi，同时令u.pi的最“丰满”的兄弟中升一个关键至u uu。然后继续扫描B树，寻找k；

2：如果u.pi的两个相邻的兄弟都不“丰满”（都只有t−1个关键字）。则令u.pi和其一个兄弟合并，再将u的一个关键字降至新合并的节点。使之成为该节点的中间关键字。

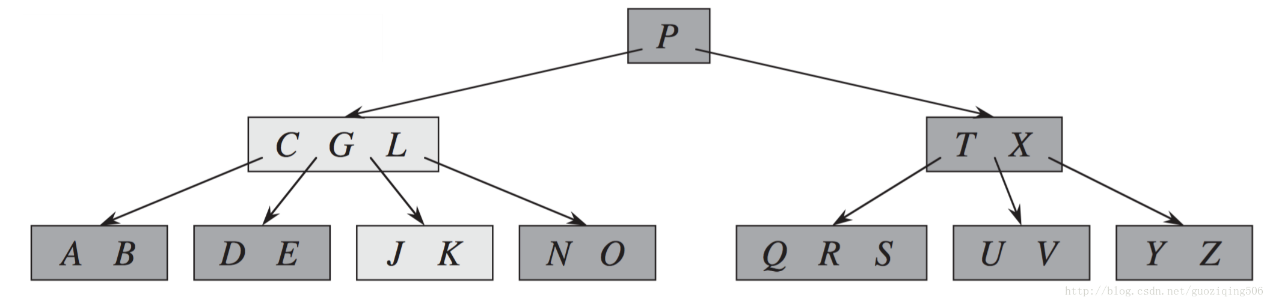
举个例子，就可以清晰看到上面说的这几种删除的情况。拿下图所示的最小度为3的B树为例（即树中除根和叶子之外的节点只能有2，3，4，5四种情况的关键字个数）：



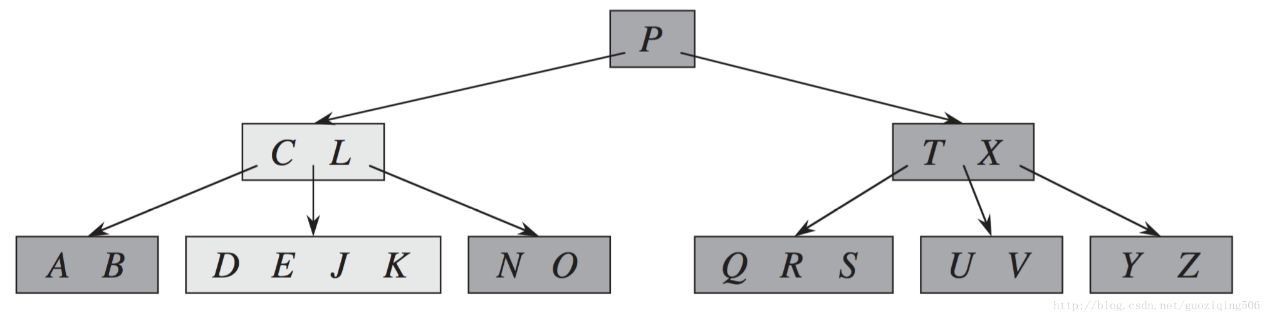
Step **1:** 删除上图中的关键字F，过程如下：先扫描根节点（含P），再扫描其左孩子（含CGM），发现丰满，继续扫描到左起第二个叶子，然后就是符合**Case - 1**的情况了。结果如下图所示：



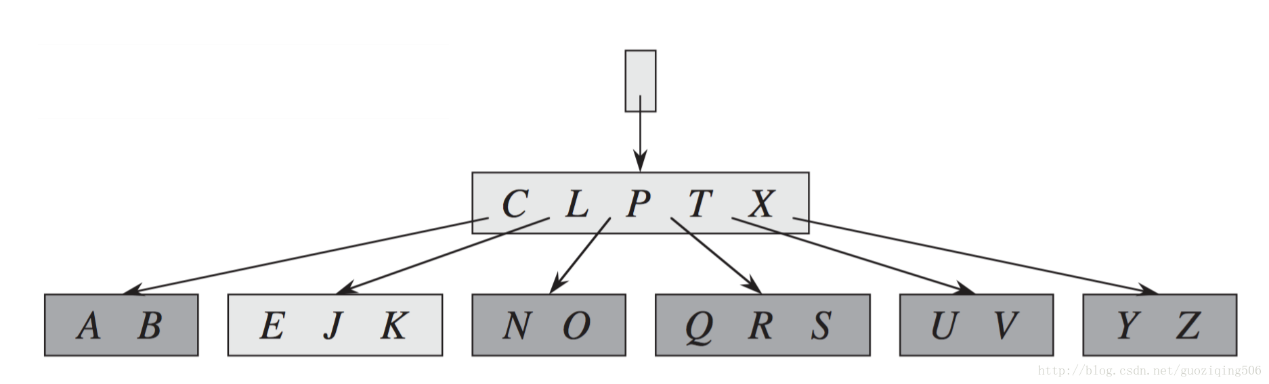
**Step 2**: 再删除M，此时遇到\*\*Case - 2(1)\*\*的情况，结果如下图所示：



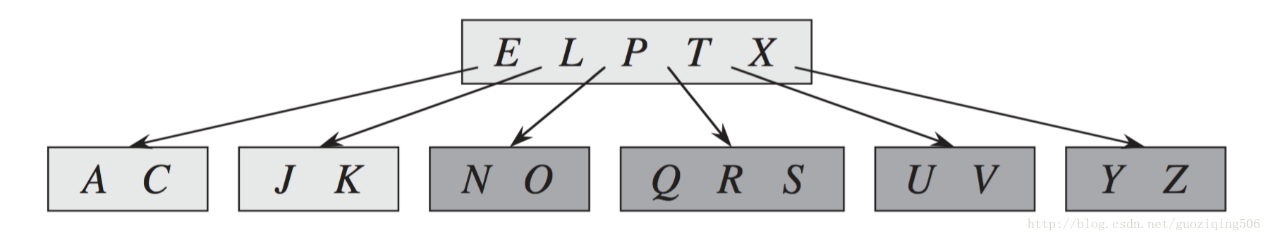
**Step 3:** 再删除G，G的前驱、后驱都是不丰满的。也就是\*\*Case - 2(3)\*\*的情况，结果如下图所示：



**Step 4:** 再删除D，扫描至含CL的节点后，发现它不丰满，且他的兄弟也不丰满。则将节点CL和TX合并，并降关键字P至新合并的节点。也就是\*\*Case - 3(2)\*\*的情况，结果如下图所示，此时，树高减1：



**Step 5:** 再删除B，也就是\*\*Case - 3(1)\*\*的情况，结果如下图所示：



下面总结一下B树的删除原理：

1：基本原则是不能破坏关键字个数的限制；

2：如果在当前节点中，找到了要删的关键字，且当前节点为内部节点。那么，如果有比较丰满的前驱或后继，借一个上来，再把要删的关键字降下去，在子树中递归删除；如果没有比较丰满的前驱或后继，则令前驱与后继合并，把要删的关键字降下去，递归删除；

3：如果在当前节点中，还未找到要删的关键字，且当前节点为内部节点。那么去找下一步应该扫描的孩子，并判断这个孩子是否丰满，如果丰满，继续扫描；如果不丰满，则看其有无丰满的兄弟，有的话，从父亲那里接一个，父亲再找其最丰满的兄弟借一个；如果没有丰满的兄弟，则合并，再令父亲下降，以保证B树的结构。

**B+树**

**定义**

B+树是B树的一种变形，它更适合实际应用中操作系统的文件索引和数据库索引。定义如下：（**为和大多资料保持一致，这里使用阶数m来定义B+树，而不像之前的B树中，使用的是最小度t来定义**）

1：除根节点外的内部节点，每个节点最多有m个关键字，最少有[]个关键字。其中每个关键字对应一个子树（也就是最多有m棵子树，最少有[]棵子树）；

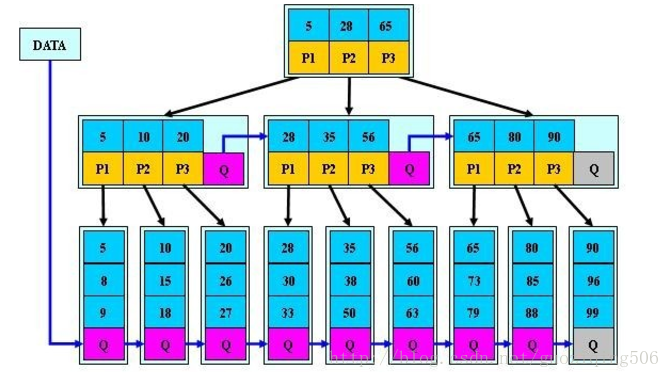
2：根节点要么没有子树，要么至少有2棵子树；

3：所有的**叶子节点**包含了全部的关键字以及这些关键字指向文件的指针，并且：

* 所有叶子节点中的关键字按大小顺序排列
* 相邻的叶子节点顺序链接（相当于是构成了一个顺序链表）
* 所有叶子节点在同一层
* 所有分支节点的关键字都是对应子树中关键字的最大值

4：所有分支节点的关键字都是对应子树中关键字的最大值

比如，下图就是一个非常典型的B+树的例子。



B+树和B树相比，主要的**不同点**在以下3项：

1：内部节点中，关键字的个数与其子树的个数相同，不像B树种，子树的个数总比关键字个数多1个

2：所有指向文件的关键字及其指针都在叶子节点中，不像B树，有的指向文件的关键字是在内部节点中。换句话说，B+树中，内部节点仅仅起到**索引**的作用，

3：在搜索过程中，如果查询和内部节点的关键字一致，那么搜索过程不停止，而是继续向下搜索这个分支。

根据B+树的结构，我们可以发现B+树相比于B树，在文件系统，数据库系统当中，更有**优势**，原因如下：

**1：B+树的磁盘读写代价更低没看懂？？？**

B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说I/O读写次数也就降低了。

**2：B+树的查询效率更加稳定**

由于内部结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

**3：B+树更有利于对数据库的扫描**

B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题，而B+树只需要遍历叶子节点就可以解决对全部关键字信息的扫描，所以对于数据库中频繁使用的range query，B+树有着更高的性能。

详情见：<https://blog.csdn.net/guoziqing506/article/details/64122287>。

### 哈夫曼树

定义：哈夫曼树，又称最优树，是一类带权路径长度最短的树。首先有几个概念需要清楚：

1：路径和路径长度-从树中一个结点到另一个结点之间的分支构成两个结点的路径，路径上的分支数目叫做路径长度。树的路径长度是从树根到每一个结点的路径长度之和。

2：结点的带权路径长度为从该结点到树根之间的路径长度与结点上权的乘积。树的带权路径长度为树中所有叶子结点的带权路径长度之和，通常记作WPL。

若有n个权值为w1,w2,...,wn的结点构成一棵有n个叶子结点的二叉树，则树的带权路径最小的二叉树叫做哈夫曼树或最优二叉树。



在上图中，3棵二叉树都有4个叶子结点a、b、c、d，分别带权7、5、2、4，则它们的带权路径长度为

（a）WPL = 7\*2 + 5\*2 + 2\*2 + 4\*2 = 36

（b）WPL = 4\*2 + 7\*3 + 5\*3 + 2\*1 = 46

（c）WPL = 7\*1 + 5\*2 + 2\*3 + 4\*3 = 35

其中（c）的WPL最小，可以验证，（c）恰为哈夫曼树。

详情见：<https://www.cnblogs.com/sench/p/7798064.html>。

### 红黑树

**红黑树与AVL树的比较：**

1.AVL树的时间复杂度虽然优于红黑树，但是对于现在的计算机，CPU太快，可以忽略性能差异

2.红黑树的插入删除比AVL树更便于控制操作

3.红黑树整体性能略优于AVL树（红黑树旋转情况少于AVL树）

红黑树的性质：

**红黑树是一棵二叉搜索树**，它在每个节点增加了一个存储位记录节点的颜色，可以是RED,也可以是BLACK；通过任意一条从根到叶子简单路径上颜色的约束，红黑树保证最长路径不超过最短路径的二倍，因而近似平衡。

具体性质如下：

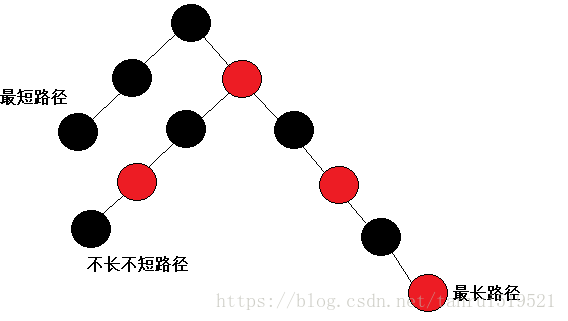
1：每个节点颜色不是黑色，就是红色

2：根节点是黑色的

3：如果一个节点是红色，那么它的两个子节点就是黑色的（**没有连续的红节点**）

4：对于每个节点，**从该节点到其后代叶节点的简单路径上，均包含相同数目的黑色节点**

如图解释：



你的最短路径就是全黑节点，最长路径就是一个红节点一个黑节点，最后黑色节点相同时，最长路径刚好是最短路径的两倍。

红黑树的插入：

红黑树插入节点过程大致分析：

RBTree为二叉搜索树，我们按照二叉搜索树的方法对其进行节点插入

RBTree有颜色约束性质，因此我们在插入新节点之后要进行颜色调整

具体步骤如下：

根节点为NULL，直接插入新节点并将其颜色置为黑色

根节点不为NULL，找到要插入新节点的位置

插入新节点

判断新插入节点对全树颜色的影响，更新调整颜色

首先红黑树的插入其实不是那么容易实现的，以前搜索树的插入我们很容易理解，现在我们首先思考一个问题，你插入节点的默认颜色是RED或BLACK？ 这里我们需要根据性质来思考，首先如果插入黑节点，这个可以直接插入无论它的父亲是什么颜色，但是红黑树的性质是每条路径的黑色节点数目相同这个时候你再想想那其他路径的黑色节点数目一定比你现在少一个节点，所以调整起来是非常繁琐的。插入红节点不需要调整其他路径，如果它的父亲为黑，那么直接插入，如果他的父亲为红那么在该路径上面开始分情况调整。所以插入节点默认颜色一定要为红。如果为黑调节成本太大了。

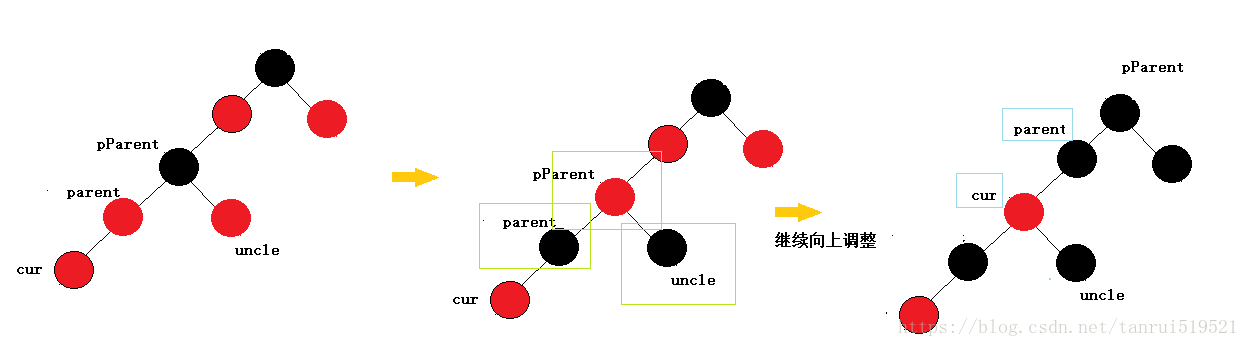
接下来开始插入节点如果插入节点的父亲为黑那么直接插入后返回不需要做任何调整.。但是如果插入节点的父亲为红，那么就需要调整了。具体的调整过程可以分为三个情况：

1：第一种情况

cur为红，parent为红，pParent为黑，uncle存在且为红

则将parent,uncle改为黑，pParent改为红，然后把pParent当成cur，继续向上调整。

此时对该子树进行操作，parent节点和uncle节点变为黑，pParent节点变为红，这样我们就保证该子树中每条路径中黑色节点相同并且没有连续的红节点，然后再让cur等于pParent节点继续往上继续调整。

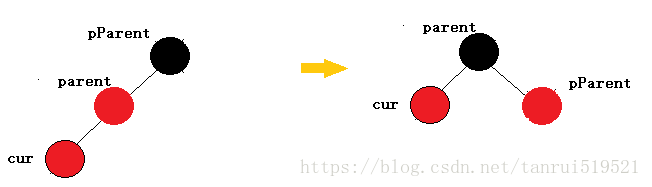


2：第二种情况

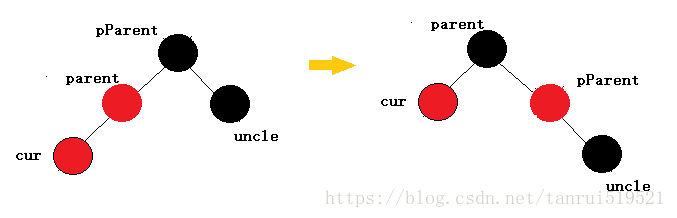
cur为红，parent为红，pParent为黑，uncle不存在/u为黑

parent为pParent的左孩子，cur为parent的左孩子，则进行右单旋转；

uncle不存在：

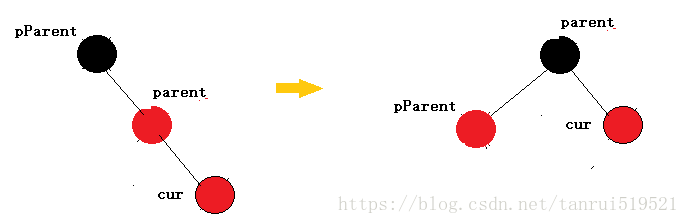


uncle存在且为黑：

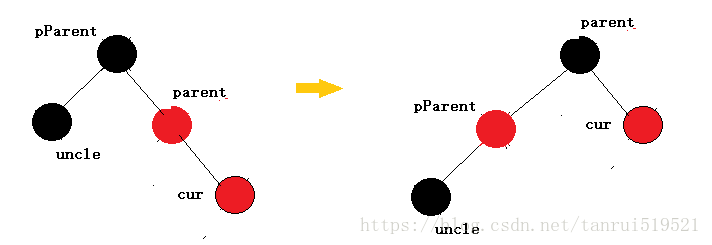


相反，parent为pParent的右孩子，cur为parent的右孩子，则进行左单旋转；   
p、g变色–p变黑，g变红。

uncle不存在：



uncle存在且为黑：



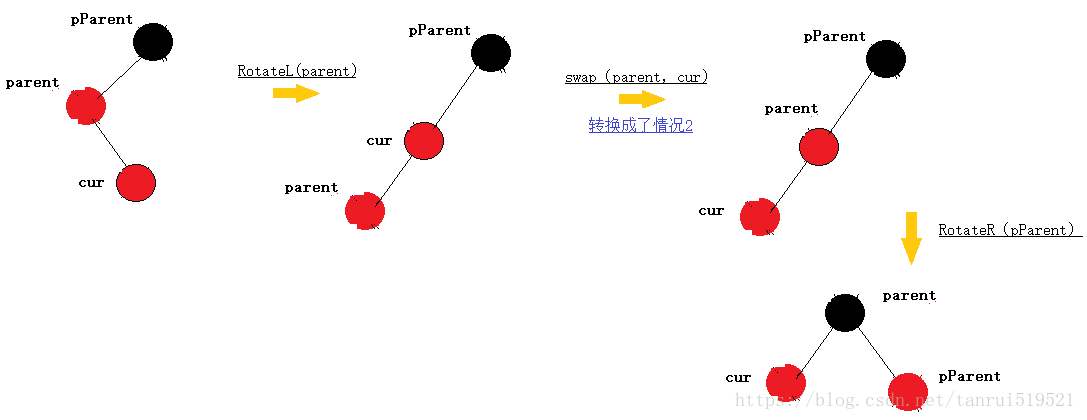
3：第三种情况

cur为红，p为红，g为黑，u不存在/u为黑

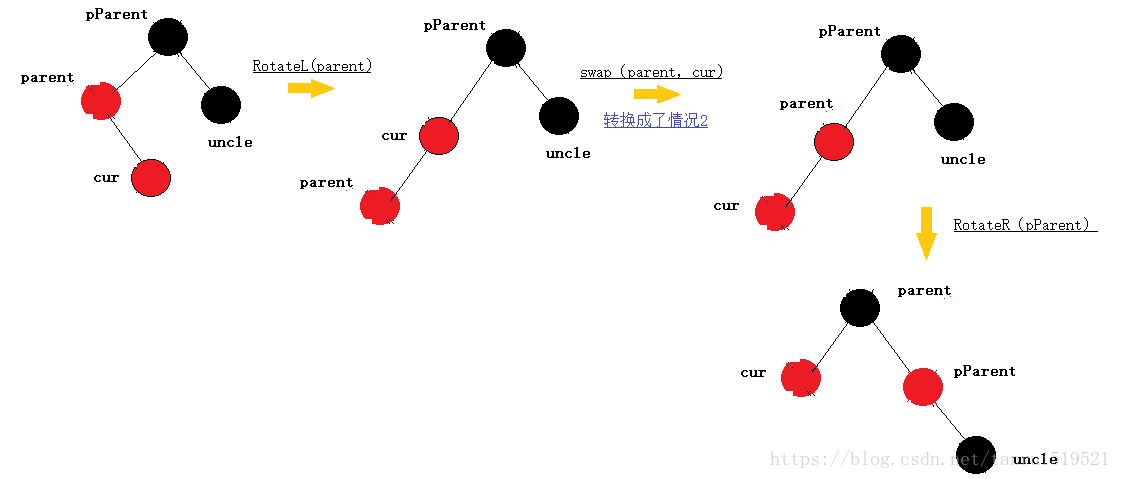
p为g的左孩子，cur为p的右孩子，则针对p做左单旋转

则转换成了情况2

uncle不存在：



uncle存在且为黑：



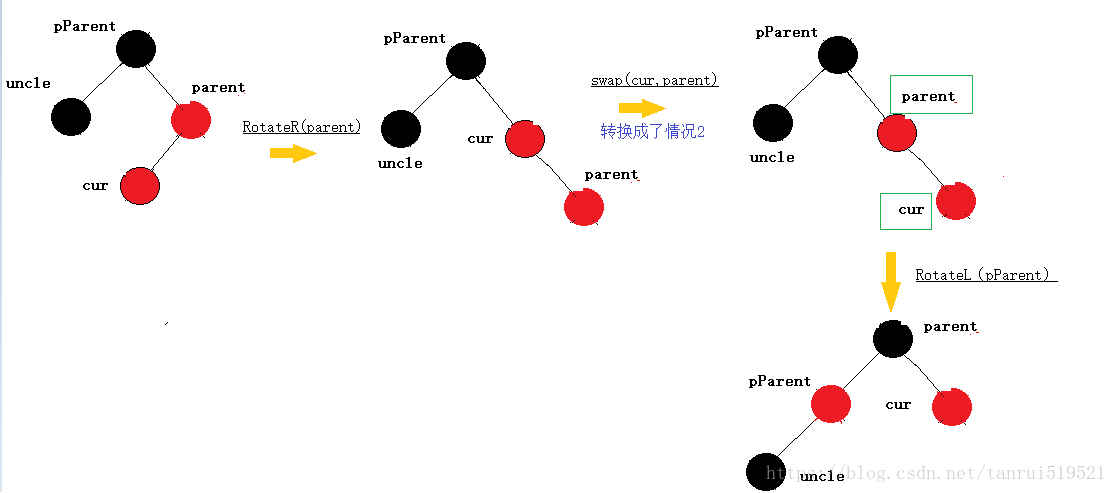
相反，p为g的右孩子，cur为p的左孩子，则针对p做右单旋转

则转换成了情况2

uncle不存在：



uncle存在且为黑：



红黑树插入节点代码实现如下：（仅作了解）

bool Insert(const K& key, const V& value)

{

//根节点为空

if (\_root == NULL)

{

\_root = new Node(key,value);

\_root->\_color = BLACK;

return true;

}

//根节点不为空

//找到新节点插入位置

Node\* parent = NULL;

Node\* cur = \_root;

while (cur)

{

if (cur->\_key < key)

{

parent = cur;

cur = cur->\_right;

}

else if (cur->\_key >key)

{

parent = cur;

cur = cur->\_left;

}

else

return false;

}

//插入新节点

cur = new Node(key, value);

cur->\_color = RED;

if (parent->\_key > key)

{

parent->\_left = cur;

cur->\_parent = parent;

}

else//parent->\_key < key

{

parent->\_right = cur;

cur->\_parent = parent;

}

//插入节点后颜色的调整

while (parent && parent->\_color == RED)

{

Node\* grandfather = parent->\_parent;//grandfather颜色一定为黑色

if (parent == grandfather->\_left)

{

Node\* uncle = grandfather->\_right;

//uncle存在且为红

if (uncle && uncle->\_color == RED)

{

parent->\_color = uncle->\_color = BLACK;

grandfather->\_color = RED;

cur = grandfather;

parent = cur->\_parent;

}

else//uncle不存在/uncle存在且为黑

{

if (cur == parent->\_right)

{

RotateL(parent);

swap(parent, cur);

}

RotateR(grandfather);

parent->\_color = BLACK;

grandfather->\_color = RED;

}

}

else//grandfather->\_right==parent

{

Node\* uncle = grandfather->\_left;

//uncle存在且为红

if (uncle && uncle->\_color == RED)

{

parent->\_color = uncle->\_color = BLACK;

grandfather->\_color = RED;

cur = grandfather;

parent = cur->\_parent;

}

else//不存在/存在且为黑

{

if (cur == parent->\_left)

{

RotateR(parent);

swap(cur, parent);

}

RotateL(grandfather);

parent->\_color = BLACK;

grandfather->\_color = RED;

}

}

}//end while (parent && parent->\_color == RED)

\_root->\_color = BLACK;

return true;

}

### 红黑树和AVL树差别

红黑树和平衡二叉树（AVL树）类似，都是在进行插入和删除操作时通过特定（旋转）操作保持二叉查找树的平衡，从而获得较高的查找性能。自从红黑树出来后，AVL树就被放到了博物馆里，据说是红黑树有更好的效率，更高的统计性能。

红黑树和AVL树的**区别**在于它使用颜色来标识结点的高度，它所追求的是**局部平衡**而不是AVL树中的非常严格的平衡。

红黑树就是平衡二叉树，只不过它的每个节点多加了一个标志属性，这个标志是在增加和删除节点时用的。

对一个平衡二叉树做几次增加删除节点的操作，它就变成非平衡的了，这不利于查找。所以每次增加、删除节点后都要进行调整，调整的算法就是按红黑树的规则“红节点的孩子不能是红节点”。**对一个n个节点的红黑树做一次这样的调整最多需要log(n)步。**

详情见：<https://blog.csdn.net/rain_qingtian/article/details/11105777>。

### HashMap是否线程安全

HashMap是线程不安全的：

HashMap在并发时可能出现的问题主要是两方面,首先**如果多个线程同时使用put方法添加元素，而且假设正好存在两个put的key发生了碰撞(hash值一样)**，那么根据HashMap的实现，这两个key会添加到数组的同一个位置，这样最终就会发生其中一个线程的put的数据被覆盖。第二就是如果多个线程同时检测到元素个数超过数组大小\*loadFactor，这样就会发生多个线程同时对Node数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给table，也就是说其他线程的都会丢失，并且各自线程put的数据也丢失。

### 如何解决hash冲突

哈希函数：一般情况下，需要在关键字与它在表中的[存储](http://www.07net01.com/storage_networking/)位置之间建立一个函数关系，以f(key)作为关键字为key的记录在表中的位置，通常称这个函数f(key)为哈希函数。

**hash :**翻译为“散列”，就是把任意长度的输入，通过散列[算法](http://lib.csdn.net/base/datastructure)，变成固定长度的输出，该输出就是散列值。

     这种转换是一种压缩映射，散列值的[空间](http://www.07net01.com/tags-%E7%A9%BA%E9%97%B4-0.html)通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到莫伊固定长度的消息摘要的函数。`

**hash冲突：就是根据key即经过一个函数f(key)得到的结果的作为地址去存放当前的key value键值对(这个是hashmap的存值方式)，但是却发现算出来的地址上已经有人先来了。**就是说这个地方要挤一挤啦。这就是所谓的hash冲突啦

**解决方法：1.开放定址发。2.链地址法。3.再哈希法。4.建立一个公共溢出区。**

1：开放地址法

这种方法也称再散列法，其基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。这种方法有一个通用的再散列函数形式：

Hi=（H（key）+di）% m   i=1，2，…，n

2：再哈希发

这种方法是**同时构造多个不同的哈希函数**：

Hi=RH1（key）  i=1，2，…，k

**当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。**

3：链地址发

**这种方法的基本思想是将所有哈希地址为i（相同）的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。**

4：简历公共溢出区

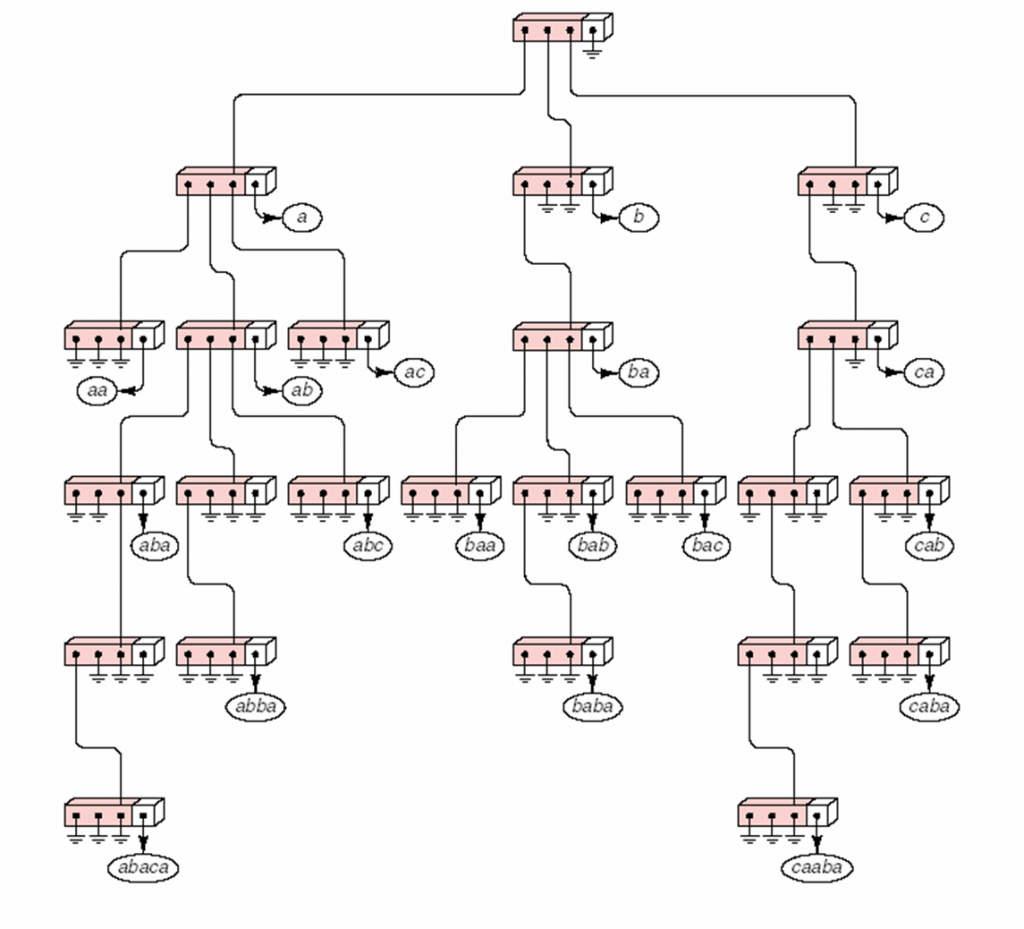
**这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。**

详情见：<https://blog.csdn.net/yeiweilan/article/details/73412438>。

### 字典树

**定义**-Trie，又称字典树，是一种用于快速检索的二十六叉树结构。典型的空间换时间。

**结构**



**原理**-Trie把要查找的关键词看作一个字符序列，并根据构成关键词字符的先后顺序检索树结构；

**特别地：**和二叉查找树不同，在Trie树中，每个结点上并非存储一个元素。

**性质**

1：**利用串的公共前缀**，节约内存

    2：在trie树上进行检索总是始于根结点

    3：**根节点不包含字符**，**除根节点外的每一个节点都只代表一个字母，这里不是包含。**

    4：**从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。**

5：每个节点的所有子节点包含的字符都不相同。

**应用**

**1：字典树在串的快速检索中的应用。**

给出N个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。在这道题中，我们可以用字典树，先把熟词建一棵树，然后读入文章进行比较，这种方法效率是比较高的。

**2：字典树在“串”排序方面的应用**

给定N个互不相同的仅由一个单词构成的英文名，让你将他们按字典序从小到大输出用字典树进行排序，采用数组的方式创建字典树，这棵树的每个结点的所有儿子很显然地按照其字母大小排序。对这棵树进行**先序遍历**即可。

**3：字典树在最长公共前缀问题的应用**

对所有串建立字典树，对于两个串的最长公共前缀的长度即他们所在的结点的公共祖先个数，于是，问题就转化为最近公共祖先问题。

详情见：<https://blog.csdn.net/lucky_greenegg/article/details/9141759>。

### 二叉树、完全二叉树、满二叉树、平衡二叉树

概念介绍

**度：**指的是一个节点拥有子节点的个数。深度：树的层数。叶子结点：度为0的节点，即没有子节点的节点。

树：树中的每一个节点，可以有n（后续节点）个子节点，但是每个节点只有一个前驱节点。

二叉树：除了叶子节点外，每个节点只有两个分支，左子树和右子树，每个节点的最大度数为2.

满二叉树：除了叶结点外每一个结点都有左右[子叶](https://www.baidu.com/s?wd=%E5%AD%90%E5%8F%B6&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)且叶结点都处在最底层的二叉树。

完全二叉树：只有最下面的两层结点度小于2，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树。

也就是说，在满叉树的基础上，我在最底层从右往左删去若干节点，得到的都是完全二叉树。完全二叉树包含满二叉树

平衡二叉树：树的左右子树的高度差不超过1的数，空树也是平衡二叉树的一种。常用算法有：[红黑树](http://baike.baidu.com/view/133754.htm)、[AVL](http://baike.baidu.com/view/414610.htm)树、[Treap](http://baike.baidu.com/view/956602.htm)等。

二叉树的性质：

1：在二叉树中，第i层的结点总数不超过2^([i-1](https://www.baidu.com/s?wd=i-1&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd))；

2：深度为h的二叉树最多有2^h-1个结点(h>=1)，最少有h个结点（链表）；

3：对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，

则N0=N2+1；

4：具有n个结点的完全二叉树的深度为int（log2n）+1

5：有N个结点的完全二叉树各结点如果用顺序方式存储，则结点之间有如下关系：

若I为结点编号则 如果I<>1，则其父结点的编号为I/2；

如果2\*I<=N，则其左儿子（即左子树的根结点）的编号为2\*I；若2\*I>N，则无左儿子；

如果2\*I+1<=N，则其右儿子的结点编号为2\*I+1；若2\*I+1>N，则无右儿子。

6：给定N个节点，能构成h(N)种不同的二叉树。

h(N)为卡特兰数的第N项。h(n)=C(n,2\*n)/(n+1)。

详情见：<https://blog.csdn.net/yuwushuang11/article/details/78628071>。

### 跳表

结合链表和二分法的特点，将链表进行加工，创造一个二者的结合体：  
1.链表从头节点到尾节点是有序的  
2.可以进行跳跃查找（形如二分法）

跳表节点定义

template<class T>

class skipNode<T>

{

public:

skipNode(const T& theElement, size\_t size):

element(theElement)

{

next = new skipNode<T>\*[size+1];

}

T element;

skipNode<T> \*\*next;

};

详情见：<https://www.cnblogs.com/haoliuhust/p/4420887.html>。

### 最小生成树算法

1：生成树的概念

**所有的顶点和边都属于图G的图称为G的子图。含有G的所有顶点的子图称为G的生成子图。**

如果T为G的生成子图且T为树，图T称为无向图G的生成树（spanning tree）。

2：求图的最小生成树算法

**严格来说，如果图G=（V，E）是一个连通的无向图，则把它的全部顶点V和一部分边E’构成一个子图G’，即G’=（V， E’），且边集E’能将图中所有顶点连通又不形成回路，则称子图G’是图G的一棵生成树。**

对于加权连通图，生成树的权即为生成树中所有边上的权值总和，权值最小的生成树称为图的**最小生成树**。

求图的最小生成树具有很高的实际应用价值，比如下面的这个例题。

例1、城市公交网

[问题描述]

有一张城市地图，图中的顶点为城市，无向边代表两个城市间的连通关系，边上的权为在这两个城市之间修建高速公路的造价，研究后发现，这个地图有一个特点，即任一对城市都是连通的。现在的问题是，要修建若干高速公路把所有城市联系起来，问如何设计可使得工程的总造价最少。

[输入]

n（城市数，1<=n<=100）

e（边数）

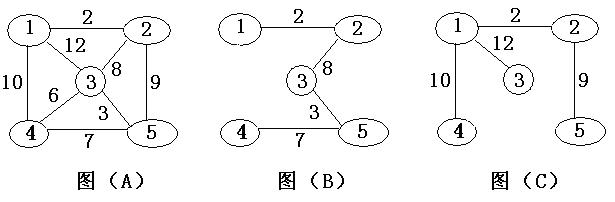
以下e行，每行3个数i,j,wij，表示在城市i,j之间修建高速公路的造价。

 [输出]

n-1行，每行为两个城市的序号，表明这两个城市间建一条高速公路。

 [举例]

下面的图（A）表示一个5个城市的地图，图（B）、（C）是对图（A）分别进行**深度优先遍历和广度优先遍历**得到的一棵生成树，其权和分别为20和33，前者比后者好一些，但并不是最小生成树，最小生成树的权和为19。



[问题分析]

出发点：具有n个顶点的带权连通图，其对应的生成树有n-1条边。

那么选哪n-1条边呢？设图G的度为n，G=（V，E），我们介绍两种基于贪心的算法，Prim算法和Kruskal算法。

Prim算法

普里姆算法的基本思想：1：清空生成树，任取一个顶点加入生成树；

2：在那些一个端点在生成树里，另一个端点不在生成树里的边中，选取一条权最小的边，将它和另一个端点加进生成树； 3：重复步骤2，直到所有的顶点都进入了生成树为止，此时的生成树就是最小生成树。

即:从连通网络 N = { V, E }中的某一顶点 u0 出发，选择与它关联的具有最小权值的边(u0, v)，将其顶点v加入到生成树的顶点集合U中。以后每一步从一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的边(u, v),把它的顶点 v加入到集合U中。如此继续下去，直到网络中的所有顶点都加入到生成树顶点集合U中为止。

#include<iostream>

using namespace std;

#define MAX 100

#define MAXCOST 0x7fffffff

int graph[MAX][MAX];

int Prim(int graph[][MAX], int n)//二维数组作为参数如何使用？

{

int sta[MAX];//存放某一条边的起点值

int lowcost[MAX];//存放以i为终点的的边的最小的权值

int min,minid,sum=0; //min用来存放最小权值，minid用来存放权值最小的边所对应的终点

for(int i=2;i<=n;i++)

{

lowcost[i]=graph[1][i]; //初始化lowcost[i]，并把他们的初始值都看作是从节点1出发到i的权值

sta[i]=1;//起点赋值为1

}

sta[1]=0; //节点1进入最小生成树

for(int h=2;h<=n;h++)

{

min=MAXCOST; //找到最小的，先来个较大值

for(int j=2;j<=n;j++)

{

if(lowcost[j]<min&&lowcost[j]!=0) //如果找到权值较小的就赋值给min，并把终点j赋值给minid。

{ min=lowcost[j]; minid=j;}

}

lowcost[minid]=0; //这条边已经进入最小生成树，所以把值置为0

sum+=min;

for(int s=2;s<=n;s++)

{

if(lowcost[s]<graph[minid][s]) //如果原先的lowcost[j]的值大于以minid为起点到终点j的权值，则更新它，并把起点更新为minid

{

lowcost[s]=graph[minid][s];

sta[s]=minid;

}

}

}

return sum;

}

int main()

{

int m,n,x,y,cost;

cout<<"请输入节点数目和边的数目："<<endl;

cin>>m>>n;

for(int i=1;i<=m;i++)

for(int j=1;j<=m;j++)

graph[i][j]=MAXCOST;

for(int k=1;k<=n;k++)

{

cin>>x>>y>>cost;

graph[x][y]=graph[y][x]=cost;

}

cost= Prim(graph,n);

cout<<cost<<endl;

return 0;

}

Kruskal算法1：对所有边进行从小到大的排序。2：每次选一条边（最小的边），如果如果形成环，就不加入(u,v)中，否则加入。那么加入的(u,v)一定是最佳的。

#include <iostream>

#include<algorithm>

using namespace std;

#define MAX 100

struct edge

{

int x,y;

int w;

}e[MAX];

int fa[MAX];

int rank[MAX];

int sum;

int cmp(edge a,edge b)//排序函数

{

if(a.w!=b.w)

return a.w<b.w;

else

{

return a.x<b.x;

}

}

void make\_set(int x)//初始化节点

{

fa[x]=x;

rank[x]=0;

}

int find(int x)//查找父节点

{

return fa[x]==x?x:fa[x]=find(fa[x]);

}

void union\_set(int x,int y,int w)//合并节点

{

if(rank[x]>rank[y])

{

rank[y]=x;

}

else if(rank[x]<rank[y])

{

rank[x]=y;

}

else

{

rank[x]++;

rank[y]=x;

}

sum+=w;//总权值加上w

}

int main()

{

int x,y,w;

int m,n;//n是点,m是边

cin>>n>>m;

for(int i=0;i<m;i++)

{

cin>>x>>y>>w;

e[i].x=x;

e[i].y=y;

e[i].w=w;

make\_set(x);

make\_set(y);

}

sort(e,e+m,cmp);

sum=0;

for(int i=0;i<n;i++)

{

x=find(e[i].x);

y=find(e[i].y);

w=e[i].w;

if(x!=y)

{

union\_set(x,y,w);

}

}

cout<<sum<<endl;

return 0;

}

详情见：<https://www.cnblogs.com/GHzz/p/9148279.html>。

### 最短路径算法

熟悉的最短路算法就四种：深度或广度优先搜索算法，弗洛伊德算法，迪杰斯特拉算法，Bellman-Ford算法

**1：深度或广度优先搜索算法（解决单源最短路径）**

从起始结点开始访问所有的深度遍历路径或广度优先路径，则到达终点结点的路径有多条，取其中路径权值最短的一条则为最短路径。

下面是核心代码：

void dfs(int cur, int dst){

/\*\*\*operation\*\*\*/

/\*\*\*operation\*\*\*/

if(minPath < dst) return;//当前走过路径大于之前最短路径，没必要再走下去

if(cur == n){//临界条件

if(minPath > dst) minPath = dst;

return;

}

else{

int i;

for(i = 1; i <= n; i++){

if(edge[cur][i] != inf && edge[cur][i] != 0 && mark[i] == 0){

mark[i] = 1;

dfs(i, dst+edge[cur][i]);

mark[i] = 0; //需要在深度遍历返回时将访问标志置0

}

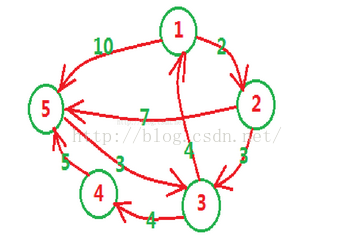
}

return;

}

}

例1：下面是城市的地图，注意是单向图，求城市1到城市5的最短距离。(引用的是上次总结的图论（一）中1）的例2)



实现

/\*\*\*先输入n个结点，m条边，之后输入有向图的m条边，边的前两元素表示起始结点，第三个值表权值，输出1号城市到n号城市的最短距离\*\*\*/

/\*\*\*算法的思路是访问所有的深度遍历路径，需要在深度遍历返回时将访问标志置0\*\*\*/

#include <iostream>

#include <iomanip>

#define nmax 110

#define inf 999999999

using namespace std;

int n, m, minPath, edge[nmax][nmax], mark[nmax];//结点数，边数，最小路径，邻接矩阵，结点访问标记

void dfs(int cur, int dst){

/\*\*\*operation\*\*\*/

/\*\*\*operation\*\*\*/

if(minPath < dst) return;//当前走过路径大于之前最短路径，没必要再走下去

if(cur == n){//临界条件

if(minPath > dst) minPath = dst;

return;

}

else{

int i;

for(i = 1; i <= n; i++){

if(edge[cur][i] != inf && edge[cur][i] != 0 && mark[i] == 0){

mark[i] = 1;

dfs(i, dst+edge[cur][i]);

mark[i] = 0;

}

}

return;

}

}

int main(){

while(cin >> n >> m && n != 0){

//初始化邻接矩阵

int i, j;

for(i = 1; i <= n; i++){

for(j = 1; j <= n; j++){

edge[i][j] = inf;

}

edge[i][i] = 0;

}

int a, b;

while(m--){

cin >> a >> b;

cin >> edge[a][b];

}

//以dnf(1)为起点开始递归遍历

memset(mark, 0, sizeof(mark));

minPath = inf;

mark[1] = 1;

dfs(1, 0);

cout << minPath << endl;

}

return 0;

}

**2：弗洛伊德算法（解决多源最短路径）：时间复杂度O(n^3),空间复杂度O(n^2)**

基本思想：**最开始只允许经过1号顶点进行中转，接下来只允许经过1号和2号顶点进行中转......允许经过1~n号所有顶点进行中转，来不断动态更新任意两点之间的最短路程。**即求从i号顶点到j号顶点只经过前k号点的最短路程。

分析如下：

1：首先构建邻接矩阵Floyd[n+1][n+1]，假如现在只允许经过1号结点，求任意两点间的最短路程，很显然Floyd[i][j] = min{Floyd[i][j], Floyd[i][1]+Floyd[1][j]}，代码如下：

for(i = 1; i <= n; i++){

for(j = 1; j <= n; j++){

if(Floyd[i][j] > Floyd[i][1] + Floyd[1][j])

Floyd[i][j] = Floyd[i][1] + Floyd[1][j];

}

}

2：接下来继续求在只允许经过1和2号两个顶点的情况下任意两点之间的最短距离，在已经实现了从i号顶点到j号顶点只经过前1号点的最短路程的前提下，现在再插入第2号结点，来看看能不能更新更短路径，故只需在步骤1求得的Floyd[n+1][n+1]基础上，进行Floyd[i][j] = min{Floyd[i][j], Floyd[i][2]+Floyd[2][j]}；......

3：很显然，需要n次这样的更新，表示依次插入了1号，2号......n号结点，最后求得的Floyd[n+1][n+1]是从i号顶点到j号顶点只经过前n号点的最短路程。故核心代码如下：

#define inf 99999999

for(k = 1; k <= n; k++){

for(i = 1; i <= n; i++){

for(j = 1; j <= n; j++){

if(Floyd[i][k] < inf && Floyd[k][j] < inf && Floyd[i][j] > Floyd[i][k] + Floyd[k][j])

Floyd[i][j] = Floyd[i][k] + Floyd[k][j];

}

}

}

例1：寻找最短的从商店到赛场的路线。其中商店在1号结点处，赛场在n号结点处，1~n结点中有m条线路双向连接。

/\*\*\*先输入n，m，再输入m个三元组，n为路口数，m表示有几条路其中1为商店，n为赛场，三元组分别表起点，终点，该路径长，输出1到n的最短路径\*\*\*/

#include <iostream>

using namespace std;

#define inf 99999999

#define nmax 110

int edge[nmax][nmax], n, m;

int main(){

while(cin >> n >> m && n!= 0){

//构建邻接矩阵

int i, j;

for(i = 1; i <= n; i++){

for(j = 1; j <= n; j++){

edge[i][j] = inf;

}

edge[i][i] = 0;

}

while(m--){

cin >> i >> j;

cin >> edge[i][j];

edge[j][i] = edge[i][j];

}

//使用弗洛伊德算法

int k;

for(k = 1; k <= n; k++){

for(i = 1; i <= n; i++){

for(j = 1; j <= n; j++){

if(edge[i][k] < inf && edge[k][j] < inf && edge[i][j] > edge[i][k] + edge[k][j])

edge[i][j] = edge[i][k] + edge[k][j];

}

}

}

cout << edge[1][n] << endl;

}

return 0;

}

**3：迪杰斯特拉算法（解决单源最短路径）**

基本思想：每次找到离源点（如1号结点）最近的一个顶点，然后以该顶点为中心进行扩展，最终得到源点到其余所有点的最短路径。

基本步骤：

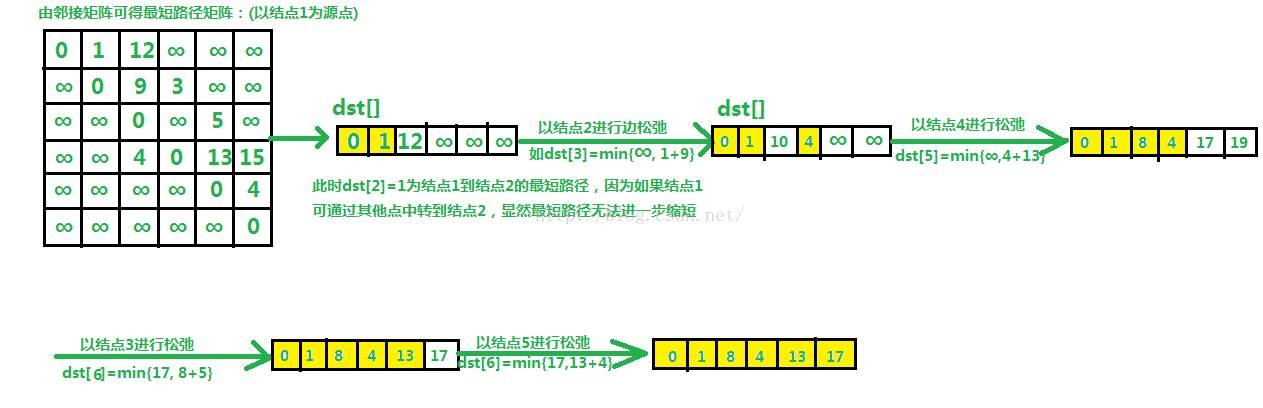
1：设置标记数组book[]：将所有的顶点分为两部分,已知最短路径的顶点集合P和未知最短路径的顶点集合Q，很显然最开始集合P只有源点一个顶点。book[i]为1表示在集合P中；

2：设置最短路径数组dst[]并不断更新：初始状态下，令dst[i] = edge[s][i](s为源点，edge为邻接矩阵)，很显然此时dst[s]=0，book[s]=1。此时，在集合Q中可选择一个离源点s最近的顶点u加入到P中。并依据以u为新的中心点，对每一条边进行松弛操作(松弛是指由结点s-->j的途中可以经过点u，并令dst[j]=min{dst[j], dst[u]+edge[u][j]})，并令book[u]=1;

3：在集合Q中再次选择一个离源点s最近的顶点v加入到P中。并依据v为新的中心点，对每一条边进行松弛操作(即dst[j]=min{dst[j], dst[v]+edge[v][j]}),并令book[v]=1;

4：重复3，直至集合Q为空。

以下是图示：



核心代码如下所示：

#define inf 99999999

/\*\*\*构建邻接矩阵edge[][],且1为源点\*\*\*/

for(i = 1; i <= n; i++) dst[i] = edge[1][s]；

for(i = 1; i <= n; i++) book[i] = 0;

book[1] = 1;

for(i = 1; i <= n-1; i++){

//找到离源点最近的顶点u，称它为新中心点

min = inf;

for(j = 1; j <= n; j++){

if(book[j] == 0 && dst[j] < min){

min = dst[j];

u = j;

}

}

book[u] = 1;

//更新最短路径数组

for(k = 1; k <= n; k++){

if(edge[u][k] < inf && book[k] == 0){

if(dst[k] > dst[u] + edge[u][k])

dst[k] = dst[u] + edge[u][k];

}

}

}

例1：给你n个点，m条无向边，每条边都有长度d和花费p，给你起点s，终点t，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入：输入n,m，点的编号是1~n,然后是m行，每行4个数 a,b,d,p，表示a和b之间有一条边，且其长度为d，花费为p。最后一行是两个数s,t;起点s，终点 t。n和m为 0 时输入结束。(1<n<=1000, 0<m<100000, s != t)

输出：输出一行，有两个数， 最短距离及其花费。

分析：由于每条边有长度d和花费p，最好构建边结构体存放，此外可以使用邻接链表，使用邻接链表时需要将上面的核心代码修改几个地方：

1，初始化dst[]时使用结点1的邻接链表；

2，更新最短路径数组时，k的范围由1~n变为1~edge[u].size()。先采用邻接矩阵解决此题，再使用邻接表解决此题，两种方法的思路都一样：初始化邻接矩阵或邻接链表，并

初始化最短路径数组dst ----> n-1轮边的松弛中，先找到离新源点最近的中心点u，之后根据中心点u为转折点来更新路径数组。

使用邻接矩阵求解：

/\*\*\*对于无向图，输入n,m，点的编号是1~n,然后是m行，每行4个数 a,b,d,p，表示a和b之间有一条边，且其长度为d，花费为p。最后一行是两个数s,t;起点s，终点 t。\*\*\*/

/\*\*\*n和m为 0 时输入结束。(1<n<=1000, 0<m<100000, s != t) 输出：输出一行，有两个数， 最短距离及其花费。\*\*\*/

#include <iostream>

#include <iomanip>

using namespace std;

#define nmax 1001

#define inf 99999999

struct Edge{

int len;

int cost;

};

Edge edge[nmax][nmax];

int dst[nmax], spend[nmax], book[nmax], n, m, stNode, enNode;

int main(){

while(cin >> n >> m && n != 0 && m != 0){

int a, b, i, j;

//构建邻接矩阵和最短路径数组

for(i = 1; i <= n; i++){

for(j = 1; j <= n; j++){

edge[i][j].cost = 0;

edge[i][j].len = inf;

}

edge[i][i].len = 0;

}

while(m--){

cin >> a >> b;

cin >> edge[a][b].len >> edge[a][b].cost;

edge[b][a].len = edge[a][b].len;

edge[b][a].cost = edge[a][b].cost;

}

cin >> stNode >> enNode;

for(i = 1; i <= n; i++){

dst[i] = edge[stNode][i].len;

spend[i] = edge[stNode][i].cost;

}

memset(book, 0, sizeof(book));

book[stNode] = 1;

//开始迪杰斯特拉算法，进行剩余n-1次松弛

int k;

for(k = 1; k <= n-1; k++){

//找离源点最近的顶点u

int minNode, min = inf;

for(i = 1; i <= n; i++){

if(book[i] == 0 && min > dst[i] /\* || min == dst[i]&& edge[stNode][min].cost > edge[stNode][i].cost\*/){

min = dst[i];

minNode = i;

}

}

//cout << setw(2) << minNode;

book[minNode] = 1;//易错点1，错写成book[i]=1

//以中心点u为转折点来更新路径数组和花费数组

for(i = 1; i <= n; i++){

if(book[i] == 0 && dst[i] > dst[minNode] + edge[minNode][i].len || dst[i] == dst[minNode] + edge[minNode][i].len && spend[i] > spend[minNode] + edge[minNode][i].cost){

dst[i] = dst[minNode] + edge[minNode][i].len;//易错点2，错写成dst[i]+

spend[i] = spend[minNode] + edge[minNode][i].cost;

}

}

}

cout << dst[enNode] << setw(3) << spend[enNode] << endl;

}

return 0;

}

使用邻接链表求解：

/\*\*\*对于无向图，输入n,m，点的编号是1~n,然后是m行，每行4个数 a,b,d,p，表示a和b之间有一条边，且其长度为d，花费为p。最后一行是两个数s,t;起点s，终点 t。\*\*\*/

/\*\*\*n和m为 0 时输入结束。(1<n<=1000, 0<m<100000, s != t) 输出：输出一行，有两个数， 最短距离及其花费。\*\*\*/

#include <iostream>

#include <iomanip>

#include <vector>

using namespace std;

#define nmax 1001

#define inf 99999999

struct Edge{

int len;

int cost;

int next;

};

vector<Edge> edge[nmax];

int dst[nmax], spend[nmax], book[nmax], n, m, stNode, enNode;

int main(){

while(cin >> n >> m && n != 0 && m != 0){

int a, b, i, j;

//构建邻接表和最短路径数组

for(i = 1; i <= n; i++) edge[i].clear();

while(m--){

Edge tmp;

cin >> a >> b;

tmp.next = b;

cin >> tmp.len >> tmp.cost;

edge[a].push\_back(tmp);

tmp.next = a;

edge[b].push\_back(tmp);

}

cin >> stNode >> enNode;

for(i = 1; i <= n; i++) dst[i] = inf; //注意2，别忘记写此句来初始化dst[]

for(i = 0; i < edge[stNode].size(); i++){//注意1，从下标0开始存元素，误写成i <= edge[stNode].size()

dst[edge[stNode][i].next] = edge[stNode][i].len;

//cout << dst[2] << endl;

spend[edge[stNode][i].next] = edge[stNode][i].cost;

}

memset(book, 0, sizeof(book));

book[stNode] = 1;

//开始迪杰斯特拉算法，进行剩余n-1次松弛

int k;

for(k = 1; k <= n-1; k++){

//找离源点最近的顶点u

int minnode, min = inf;

for(i = 1; i <= n; i++){

if(book[i] == 0 && min > dst[i] /\* || min == dst[i]&& edge[stnode][min].cost > edge[stnode][i].cost\*/){

min = dst[i];

minnode = i;

}

}

//cout << setw(2) << minnode;

book[minnode] = 1;//易错点1，错写成book[i]=1

//以中心点u为转折点来更新路径数组和花费数组

for(i = 0; i < edge[minnode].size(); i++){

int t = edge[minnode][i].next;//别忘了加此句，表示与结点minnode相邻的点

if(book[t] == 0 && dst[t] > dst[minnode] + edge[minnode][i].len || dst[t] == dst[minnode] + edge[minnode][i].len && spend[t] > spend[minnode] + edge[minnode][i].cost){

dst[t] = dst[minnode] + edge[minnode][i].len;

spend[t] = spend[minnode] + edge[minnode][i].cost;

}

}

}

cout << dst[enNode] << setw(3) << spend[enNode] << endl;

}

return 0;

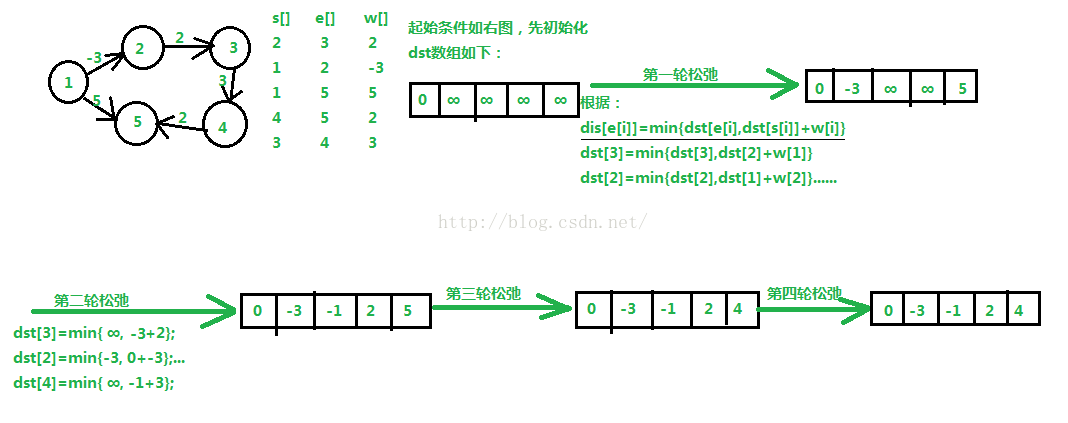
}

使用邻接表时，注意更新dst[]，book[]时要使用邻接表元素对应下标中的next成员，而涉及到权值加减时时需要使用邻接表中的对应下标来取得权值；而使用邻接矩阵就没这么多顾虑了，因为这时候邻接矩阵对应下标和dst[]要更新元素的下标正好一致，都是从1开始编号。

**4：Bellman-Ford算法(解决负权边，解决单源最短路径，前几种方法不能求含负权边的图)：：时间复杂度O(nm),空间复杂度O(m)**

主要思想：对所有的边进行n-1轮松弛操作，因为在一个含有n个顶点的图中，任意两点之间的最短路径最多包含n-1边。换句话说，第1轮在对所有的边进行松弛后，得到的是从1号顶点只能经过一条边到达其余各定点的最短路径长度。第2轮在对所有的边进行松弛后，得到的是从1号顶点只能经过两条边到达其余各定点的最短路径长度，......

以下是图示：



此外，Bellman\_Ford还可以检测一个图是否含有负权回路：如果在进行n-1轮松弛后仍然存在dst[e[i]] > dst[s[i]]+w[i]。算法核心代码如下：

#define inf 999999999

for(i = 1; i <= n; i++) dst[i] = inf;

dst[1] = 0;

for(k = 1; k <= n-1; k++){

for(i = 1; i <= m; i++){

if(dst[e[i]] > dst[s[i]] + w[i])

dst[e[i]] = dst[s[i]] + w[i];

}

}

//检测负权回路

flag = 0;

for(i = 1; i <= m; i++){

if(dst[e[i]] > dst[s[i]] + w[i])

flag = 1;

}

if(flag) cout << "此图含有负权回路";

例1：对图示中含负权的有向图，输出从结点1到各结点的最短路径，并判断有无负权回路。

/\*\*\*先输入n，m，分别表结点数和边数，之后输入m个三元组，各表起点，终点，边权，输出1号结点到各结点的最短路径\*\*\*\*/

#include <iostream>

#include <iomanip>

using namespace std;

#define nmax 1001

#define inf 99999999

int n, m, s[nmax], e[nmax], w[nmax], dst[nmax];

int main(){

while(cin >> n >> m && n != 0 && m != 0){

int i, j;

//初始化三个数组：起点数组s[],终点数组e[],权值数组w[],最短路径数组dst[]

for(i = 1; i <= m; i++)

cin >> s[i] >> e[i] >> w[i];

for(i = 1; i <= n; i++)

dst[i] = inf;

dst[1] = 0;

//使用Bellman\_Ford算法

for(j = 1; j <= n-1; j++){

for(i = 1; i <= m; i++){

if(dst[e[i]] > dst[s[i]] + w[i])

dst[e[i]] = dst[s[i]] + w[i];

}

}

//测试是否有负权回路并输出

int flag = 0;

for(i = 1; i <= m; i++)

if(dst[e[i]] > dst[s[i]] + w[i])

flag = 1;

if(flag) cout << "此图含有负权回路\n";

else{

for(i = 1; i <= n; i++){

if(i == 1)

cout << dst[i];

else

cout << setw(3) << dst[i];

}

cout << endl;

}

}

return 0;

}

详情见：[https://blog.csdn.net/qibofang/article/details/51594673#](https://blog.csdn.net/qibofang/article/details/51594673)。

### 拓扑排序

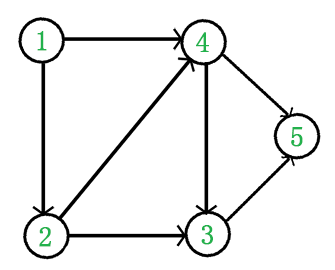
**定义**

图论中，拓扑排序（Topological Sorting）是一个有向无环图（DAG, Directed Acyclic Graph）的所有顶点的线性序列。且该序列必须满足下面两个条件：

1：每个顶点出现且只出现一次。

2：若存在一条从顶点 A 到顶点 B 的路径，那么在序列中顶点 A 出现在顶点 B 的前面。

有向无环图（DAG）才有拓扑排序，非DAG图没有拓扑排序一说。



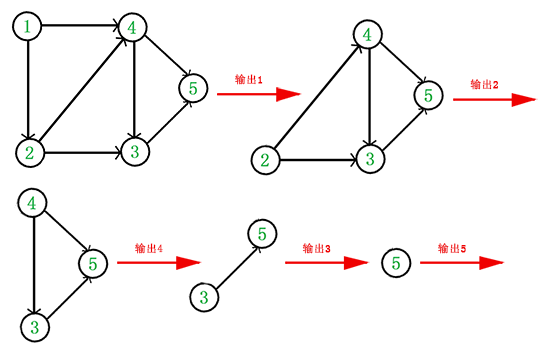
如图它是一个 DAG 图，那么如何写出它的拓扑排序呢？这里说一种比较常用的方法：

**1：从 DAG 图中选择一个没有前驱（即入度为0）的顶点并输出。**

**2：从图中删除该顶点和所有以它为起点的有向边。**

**3：重复 1 和 2 直到当前的 DAG 图为空或当前图中不存在无前驱的顶点为止。**

**4：后一种情况说明有向图中必然存在环。**



于是，得到拓扑排序后的结果是 { 1, 2, 4, 3, 5 }。

通常，一个有向无环图可以有一个或多个拓扑排序序列。

**应用**

拓扑排序通常用来“排序”具有依赖关系的任务。

比如，如果用一个DAG图来表示一个工程，其中每个顶点表示工程中的一个任务，用有向边表示在做任务 B 之前必须先完成任务 A。故在这个工程中，任意两个任务要么具有确定的先后关系，要么是没有关系，绝对不存在互相矛盾的关系（即环路）。

**实现**

#include<iostream>

#include <list>

#include <queue>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*类声明\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Graph

{

int V; // 顶点个数

list<int> \*adj; // 邻接表

queue<int> q; // 维护一个入度为0的顶点的集合

int\* indegree; // 记录每个顶点的入度

public:

Graph(int V); // 构造函数

~Graph(); // 析构函数

void addEdge(int v, int w); // 添加边

bool topological\_sort(); // 拓扑排序

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*类定义\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

indegree = new int[V]; // 入度全部初始化为0

for(int i=0; i<V; ++i)

indegree[i] = 0;

}

Graph::~Graph()

{

delete [] adj;

delete [] indegree;

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

++indegree[w];

}

bool Graph::topological\_sort()

{

for(int i=0; i<V; ++i)

if(indegree[i] == 0)

q.push(i); // 将所有入度为0的顶点入队

int count = 0; // 计数，记录当前已经输出的顶点数

while(!q.empty())

{

int v = q.front(); // 从队列中取出一个顶点

q.pop();

cout << v << " "; // 输出该顶点

++count;

// 将所有v指向的顶点的入度减1，并将入度减为0的顶点入栈

list<int>::iterator beg = adj[v].begin();

for( ; beg!=adj[v].end(); ++beg)

if(!(--indegree[\*beg]))

q.push(\*beg); // 若入度为0，则入栈

}

if(count < V)

return false; // 没有输出全部顶点，有向图中有回路

else

return true; // 拓扑排序成功

}

详情见：<https://blog.csdn.net/qq_24028753/article/details/77692447>。

### 关键路径算法（仅作了解）

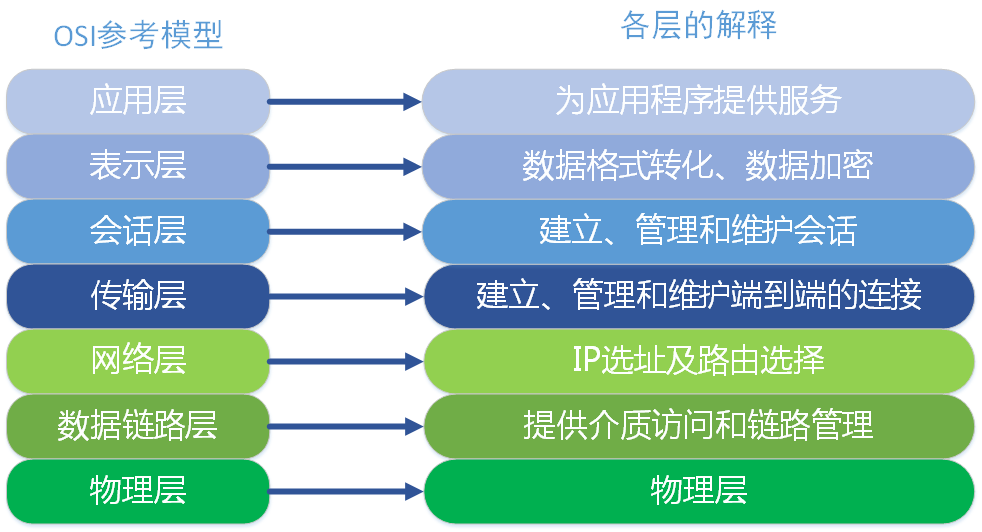
详情见：<https://blog.csdn.net/qq_35644234/article/details/52664108>。

## 计算机网络

### OSI七层模型

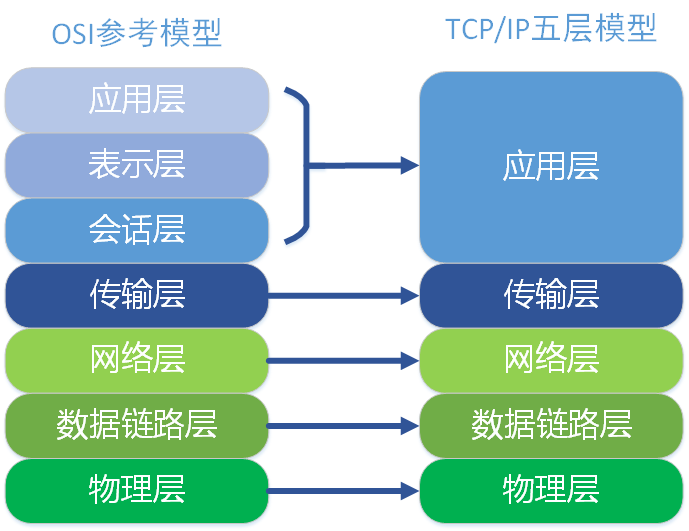
介绍：开放式系统互联，一般都叫OSI参考模型。

七层模型划分：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层，每一层实现各自的功能和协议，并完成与相邻层的接口通信。

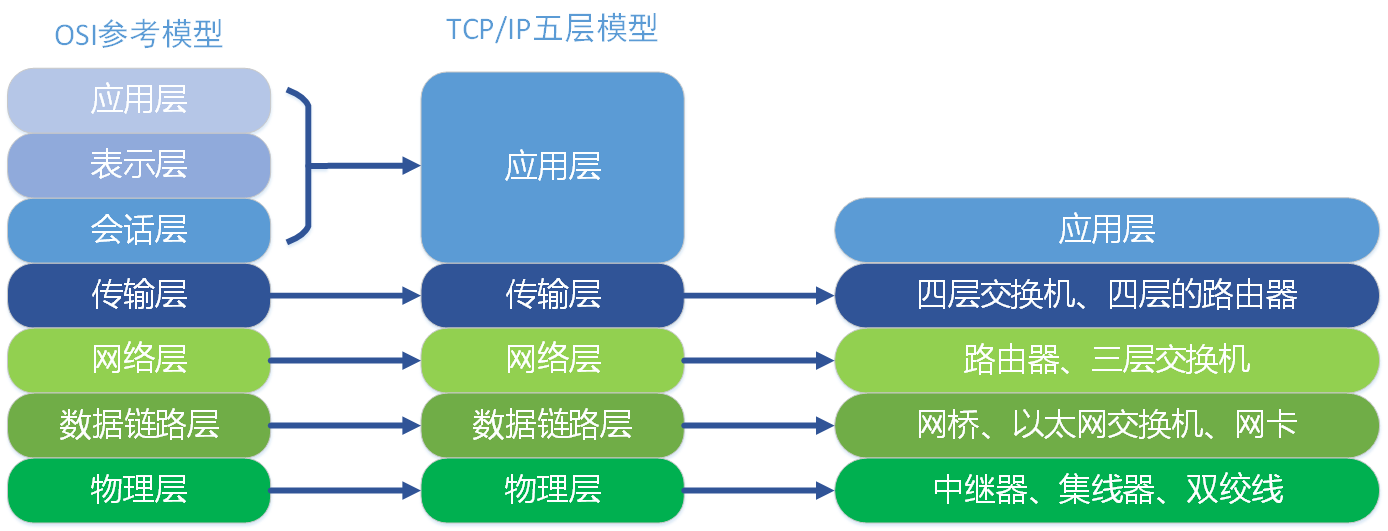


### TCP/IP五层模型

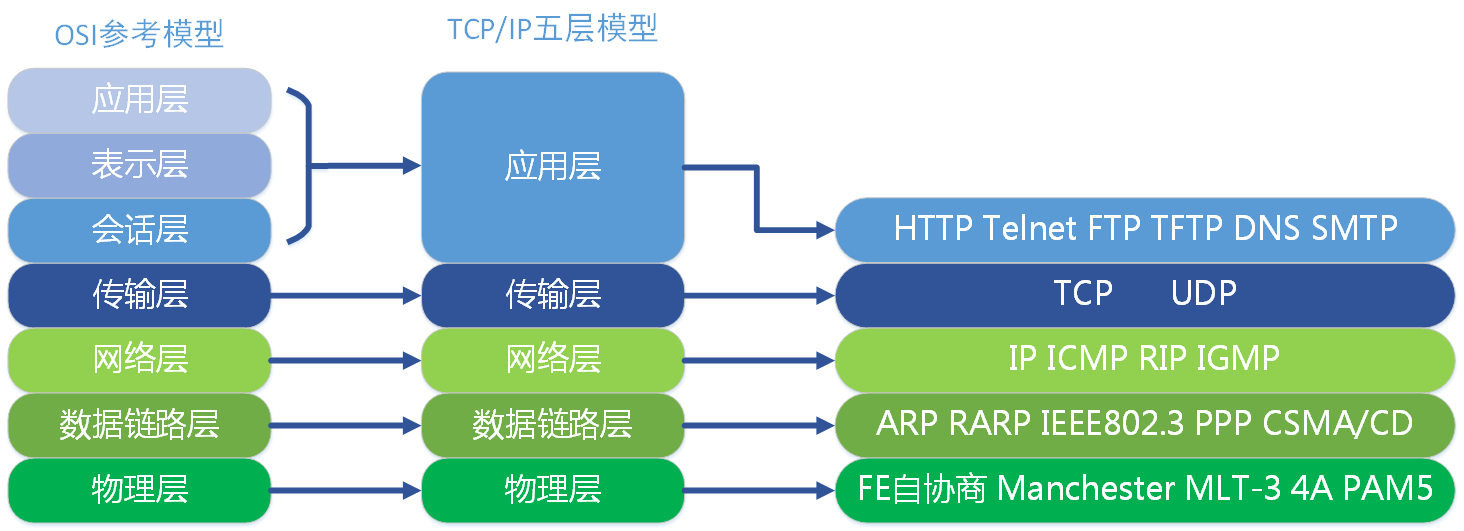
TCP/IP五层模型和OSI七层模型对应关系：



在每一层都工作者不同的设备，如：交换机就工作在数据链路层的，一般的路由器是工作在网络层的。如下图：

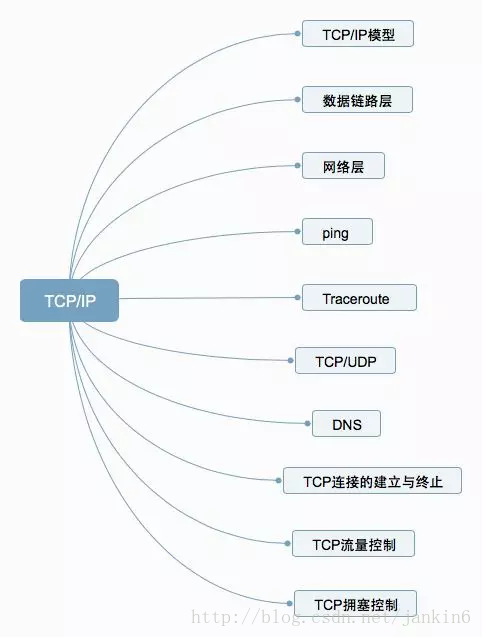


各层协议:



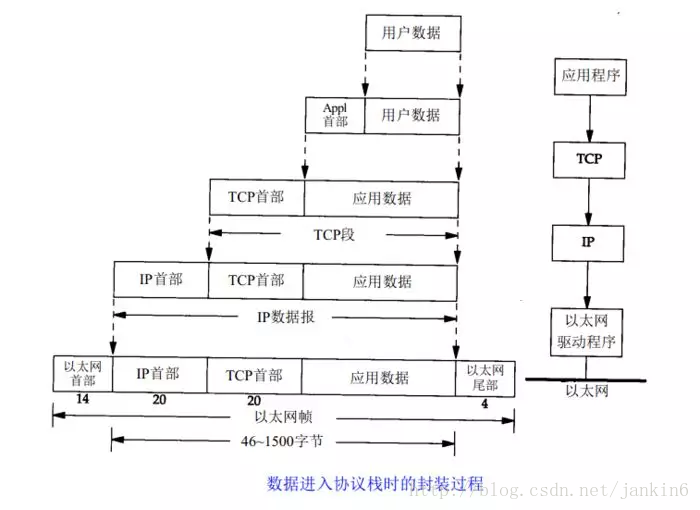
### TCP/IP协议总结

TCP/IP协议簇中需要必知必会的十大问题。



**1：TCP/IP模型**如上已介绍

如下是数据进入协议栈的封装



APP首部、TCP首部、IP首部、以太网首部

**2：数据链路层：**

特点：

1： 封装成**帧**：把网络层数据报加头和尾，封装成帧，帧头中包括**源MAC地址和目的MAC地址**。

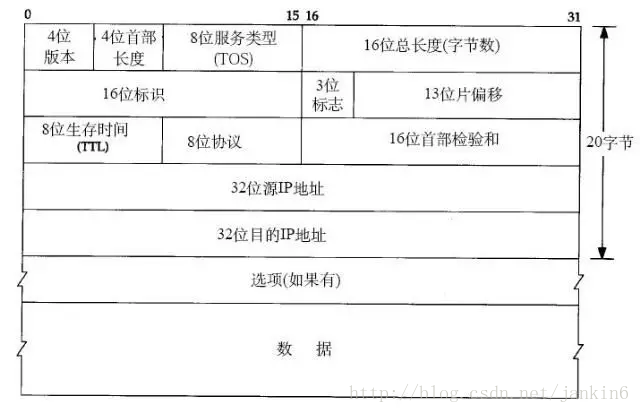
2： 透明传输：零比特传输、转义字符

3： **可靠传输**： 在出错率很低的链路上很少用，但是无线链路WLAN会保证可靠传输。

4： **差错检验（CRC）**：接收者检测错误，如果发现差错，丢弃该帧。

**3：网络层：**

IP协议头格式：



**ARP协议：根据IP地址获取到MAC地址的一种协议。**

ARP（地址解析）协议是一种解析协议，本来主机是完全不知道这个IP对应的是哪个主机的哪个接口，**当主机要发送一个IP包的时候，会首先查一下自己的ARP高速缓存（就是一个IP-MAC地址对应表缓存）。**

**如果查询的IP－MAC值对不存在，那么主机就向网络发送一个ARP协议广播包，这个广播包里面就有待查询的IP地址，而直接收到这份广播的包的所有主机都会查询自己的IP地址，如果收到广播包的某一个主机发现自己符合条件，那么就准备好一个包含自己的MAC地址的ARP包传送给发送ARP广播的主机。**

**而广播主机拿到ARP包后会更新自己的ARP缓存**（就是存放IP-MAC对应表的地方）。发送广播的主机就会用新的ARP缓存数据准备好数据链路层的的数据包发送工作。

**ICMP协议（网络层协议）：**

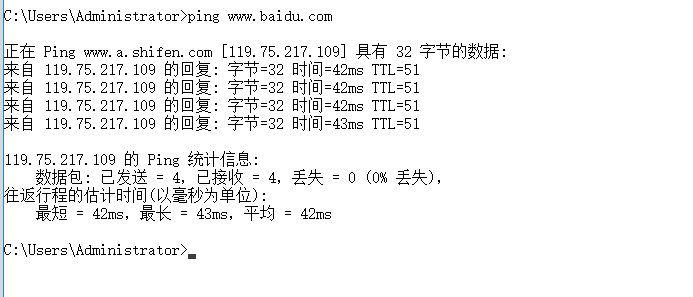
IP协议并不是一个可靠的协议，它不保证数据被送达，那么，自然的，保证数据送达的工作应该由其他的模块来完成。其中一个重要的模块就是**ICMP(网络控制报文)协议**。**ICMP不是高层协议，而是IP层的协议。**

**当传送IP数据包发生错误**。比如主机不可达，路由不可达等等，**ICMP协议将会把错误信息封包，然后传送回给主机。**给主机一个处理错误的机会，这也就是为什么说建立在IP层以上的协议是可能做到安全的原因。

**4：ping协议：**

ping可以说是ICMP的最著名的应用，是TCP/IP协议的一部分。利用“ping”命令可以检查网络是否连通，可以很好地帮助我们分析和判定网络故障。

例如：



**6：TCP和UDP（传输层协议）**

TCP和UDP都是是传输层协议，但是两者具有不同的特性，同时也具有不同的应用场景，下面以图表的形式对比分析。



**TCP面向字节流和UDP面向报文：**

UDP面向报文：发送方的UDP对应用程序交下来的报文，在添加首部后就向下交付IP层。UDP对应用层交下来的报文，**既不合并，也不拆分，而是保留这些报文的边界**。也就是说，应用层交给UDP多长的报文，UDP就照样发送，即一次发送一个报文。 在接收方的UDP，对IP层交上来的UDP用户数据报，在去除首部后就原封不动地交付上层的应用进程。也就是说，UDP一次交付一个完整的报文。因此，应用程序必须选择合适大小的长度。若报文太长，UDP把它交给IP层后，IP层在传递时可能要进行分片，这会降低IP层的效率。反之，若报文太短，UDP把它交给IP后，会使IP数据包的首部的相对长度太大，这也降低了IP层的效率。

TCP面向字节流：TCP中的”流”指的是流入到进程和从进程流出的字节序列。 “面向字节流”的含义时：**虽然应用程序和TCP的交换是一次一个数据块（大小不等），但TCP把应用程序交下来的数据看成仅仅是一连串的无结构的字节流。**TCP并不知道所传送的字节流的含义。 **TCP并不保证接收方应用程序所收到的数据块和发送方应用程序发送的数据块具有对应的大小关系。**（例如，发送方应用程序交给发送方的TCP工10个数据块，但接收方的TCP肯可能只用了4个数据块就把接收到的字节流交付上层的应用程序。）但是接收方应用程序收到的字节流必须和发送方应用程序发出的字节流完全一样。

TCP和UDP在发送报文时所采用的方式完全不同，TCP并不关心应用进程一次把多长的报文发送到TCP的缓存中，而是**根据对方给出的窗口值和当前网络拥塞的程度**来**决定一个报文段应包含多少个字节**（UDP发送的报文长度是应用程序给出的）。如果应用程序传送到TCP缓存的数据块太长，TCP可以把它划分短一些在传送，如果应用程序每一次只发送一个字节，TCP也可以等待积累有足够多的字节后再构成报文段发送出去。



TCP和UDP协议的一些应用：

TCP: SMTP(电子邮件)，TELNET（远程终端接入），HTTP（万维网），FTP（文件传输）

UDP：DNS（域名转换），TFTP（文件传输），SNMP(网络管理)，NFS（远程文件服务器）

什么时候应该使用TCP？

当对**网络通讯质量有要求（通讯效率低）的时候**，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如HTTP、HTTPS、FTP等传输文件的协议，POP、SMTP等邮件传输的协议。

什么时候应该使用UDP？

当对**网络通讯质量要求不高的时候**，**要求网络通讯速度能尽量的快**，这时就可以使用UDP。

**7：DNS**

DNS(Domain Name System，域名系统)，因特网上作为**域名和IP地址相互映射的一个分布式数据库**，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串，最终得到该主机对应的IP地址的过程叫做域名解析（或主机名解析）。DNS协议运行在UDP协议之上，使用端口号53。

**8：TCP连接的建立与终止**

参见如下。

为什么需要三次握手：

车小胖回答：<https://www.zhihu.com/question/24853633>

TCP作为一种可靠传输控制协议，其核心思想：既要保证数据可靠传输，又要提高传输的效率，而用三次恰恰可以满足以上两方面的需求！

为什么不是四次握手呢？

四次握手的过程：

1.1： A发送同步信号 SYN + A’s Initial Sequence Number

1.2： B确认收到A的同步信息，并记录A’s ISN 到本地，命名（发送）B‘s ACK sequence number

1.3： B发送同步信号SYN + B’s Initial Sequence Number

1.4： A确认收到B的同步信号，并记录B’s ISN到本地，命名（发送）A‘s ACK sequence number

很明显1.2和1.3这两个步骤可以合并，只需要三次握手，可以提高连接的速度和效率。

为什么不是二次握手？

2.1： A发送同步信号SYN + A’s initial sequence number

2.2： B发送同步信号SYN + B’s initial sequence number + B’s ack sequence number

这里有一个问题，A与B就A的初始序号达成了一致，这里是1000。但是B无法知道A是否已经接收到自己的同步信号，如果这个同步信号丢失了，A和B就B的初始序列号将无法达成一致。

补充阅读：

第一个包，即A发给B的SYN 中途被丢，没有到达B

答：A会周期性重传，直到收到B的去确认。

第二个包：即B发给A的SYN +ACK中途被丢，没有到达A

答：B会周期性重传，直到收到A的确认

第三个包：即A发给B的ACK中途丢弃，没有到达B

答：A发完ACK，单方面认为TCP为established状态，而B显然认为TCP为Active状态。

a: 假定此时双方都没有数据发送，B会周期性超时重传，直到收到A的确认，收到之后B的TCP连接也为established状态，双向可以发包。

b: 假定此时A有数据发送，B收到A的data + ack，自然会切换为established状态，并接受A的data

c: 假定B有数据发送，数据发送不了，会一直周期性超时重传SYN + ACK，直到收到A的确认才可以发送数据。

TCP连接终止



TCP终止一个连接需要4个分节

1: 某个应用程序首先调用close，我们称该端执行主动关闭（active close）。该端的TCP发送一个FIN分节，表示数据发送完毕。

2： 接收到这个FIN的对端执行被动关闭（passive close）。这个FIN由TCP确认。它的接收也作为一个文件结束符（end of file）传递给接收端应用进程（放在已排队等候该应用进程接收的任何其他数据之后），因为FIN的接受意味着接收端应用进程在相应连接上再无额外数据可接收。

3： 一段时间后，接收到这个文件描述符的应用进程将调用close关闭它的套接字。这导致它的TCP发送一个FIN。

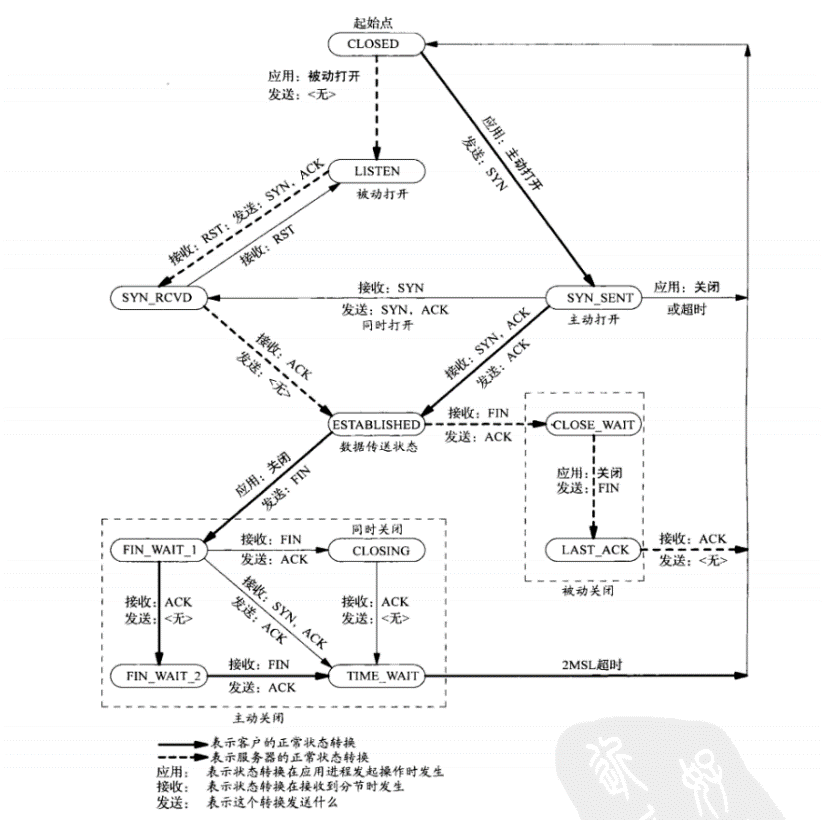
4：接收这个最终FIN的原发送端TCP（即执行主动关闭的那一端）确认这个FIN。

TCP状态转换图

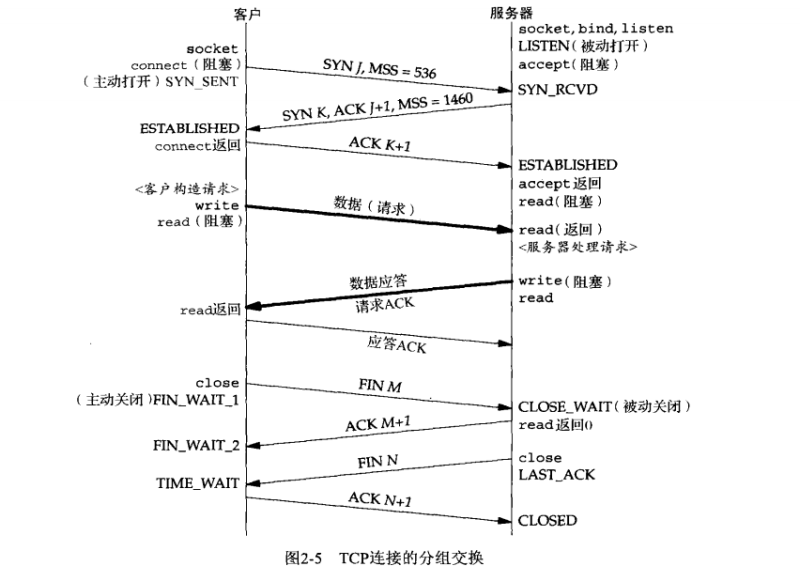
11种状态：

三次握手：CLOSED、LISTEN、SYN\_SENT、SYN\_RCVD、ESTABLISHED。

四次挥手：FIN\_WAIT\_1，CLOSE\_WAIT、FIN\_WAIT\_2、LAST\_ACK、CLOSING、TIME\_WAIT。



TCP连接的分组交换



TIME\_WAIT状态

在主动关闭的那端经历了这个状态，该断点停留在这个状态的持续时间是最长分节生命期(maximum segment lifetime，MSL)的两倍，有时候称之为2MSL。 MSL是任何IP数据报能够在因特网中存活的最长时间。

参考 UNIX网络编程P37

TIME\_WAIT状态存在有两个存在的理由：

1：可靠地实现TCP全双工连接的终止。

2：允许老的重复分节在网络中消逝。

第一个理由可以通过图2-5并假设最终的ACK丢失了来解释。服务器将重新发送它的那个最终那个FIN，因此客户必须维护状态消息，以允许它发送最终那个ACK，要是客户不维护状态消息，它将响应一个RST（另外一种类型的TCP分节），该分解被服务器解释为一个错误。这个例子也说明了为什么执行主动关闭的那一端是处于TIME\_WAIT状态的那一端：因为可能不得不重传最终那个ACK的就是那一端。

第二个理由是允许老的重复分节在网络中消逝。假设 12.106.32.254的1500端口和206.168.112.219的21端口之间有一个TCP连接。我们关闭这个连接，过一段时间后在相同的IP地址和端口之间建立另外一个连接。后一个连接称为前一个连接的化身，因为他们的IP地址和端口号都相同.TCP必须防止来自某个连接的老的分组在该连接已终止后再现，从而被误解成属于同一个连接的某个新的化身。为做到这一点，TCP将不给处于TIME\_WAIT状态的连接发起新的化身。既然TIME\_WAIT的状态的持续时间是MSL的2倍，这就足以让某个方向上的分组最多存活MSL秒即被丢弃，另外一个方向上的应答最多存活MSL秒也被丢弃。

这样就能保证每成功建立一个TCP连接时，来自该连接先前化身的老的重复分组都已经在网络中消逝了。

详情见：<https://blog.csdn.net/jankin6/article/details/79192095>。

### TCP、UDP区别



TCP优点：**可考、稳定**，可靠性体现在TCP传递数据前，会有三次握手来建立**连接**，而且在数据传递时，有确认、窗口、重传、拥塞控制机制、在数据传完后，还会断开连接用来节约系统资源。

TCP缺点：**慢、效率低、占用系统资源多，易被攻击**。建立连接、确认机制、重传机制、拥塞控制机制等消耗时间，每台设备上维护所有传输连接，每个连接占用系统CPU、内存等资源。TCP有确认机制、三次握手机制，导致TCP容易别人利用，实现DOS、DDOS、CC等攻击。

UDP优点：**快，比TCP稍安全**，UDP没有TCP的握手、确认、窗口、重传、拥塞控制等机制，UDP是一个无状态的传输协议，故传输数据非常快。**没有TCP这些机制，UDP较TCP被攻击、利用的漏洞少。**

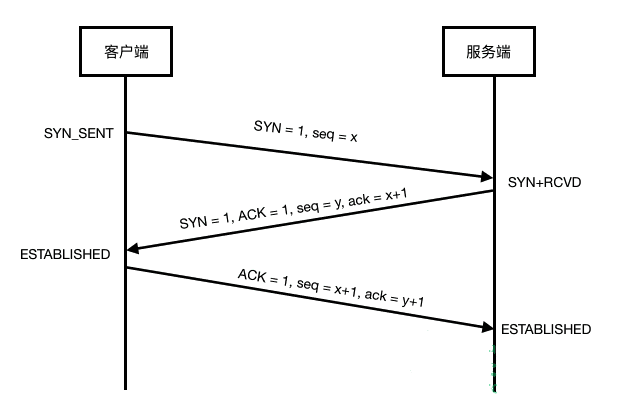
UDP缺点：**不可靠、不稳定**。没有TCP那些可靠的机制，如果网络质量不好，容易丢包。

TCP和UDP适用场景：对网络通讯质量有要求的时候，使用TCP：如HTTP、HTTPS、FTP等文件传输协议，POP、SMTP等邮件传输协议。当对网络通讯质量要求不高的时候，要求网络通讯速度尽量快，这时就可以使用UDP:比如QQ语音、QQ视频。

TCP、UDP区别：1.TCP面向连接；UDP是无连接的，发送数据前不需要建立连接。2.TCP提供可靠服务，**传输的数据无差错、不丢失、不重复、且按序到达**。UDP尽可能努力交付，**不保证可靠交付**。3.**TCP面向字节流**，实际上是**TCP把数据看成一连串无结构的字节流；UDP面向报文**。4.**UDP没有拥塞控制、流量控制机制**，因此网络出现拥塞不会使源主机的发送效率降低。5**.每一条TCP连接只能是点到点的**。**UDP支持一对一，一对多，多对一和多对多的交互通信**。6.TCP首部开销20字节。UDP首部开销8个字节。7.TCP的逻辑通信通道是全双工的可靠通道，UDP是不可靠通道。

### TCP三次握手、四次挥手

TCP是面向连接的，在传输报文段之前先要建立连接。发起连接请求的一方叫**客户端**，想要连接请求的一方叫**服务端**。



**三次握手**

**第一次握手：**

客户端向服务端发送**请求连接报文(SYN)**；其中**报头控制位SYN=1**，**初始序列号seq=x**。并进入**SYN\_SENT(SYN-sent)**状态，等待服务器确认；

**第二次握手：**

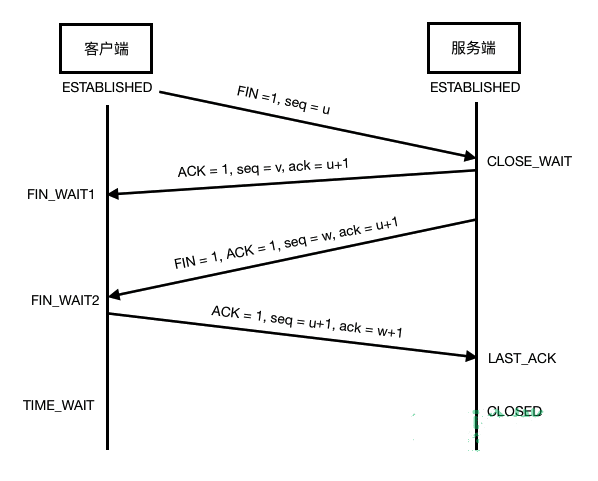
服务端收到**请求连接报文(SYN)**后，向客户端发送**确认报文(SYN+ACK包)**。确认报文段的首部中，**ACK=1、SYN=1**。**确认序号：ack=x+1**，同时为自己选择一个**初始序列号seq=y**。此时，服务器进入**SYN\_RCVD（SYN-received）**状态；

**第三次握手：**

客户端收到服务器的**确认报文(SYN＋ACK包)**后，还要再向服务端发送一个**确认报文(ACK)**。该确认报文段首部中，**ACK=1，确认号是ack=y+1，自己的序列号是seq=x+1**。此包发送完毕，客户端进入**ESTABLISHED(established)**状态，**服务端收到确认报文后**，也进入**ESTABLISHED(established)**状态。

**至此TCP连接建立。**

**三次握手过程中传送的包里不携带数据**。三次握手完毕后，客户端与服务端才正式开始传输数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。



**四次挥手**

断开一个TCP连接则需要“四次挥手”。

此处，客户端——主动关闭方，服务端——被动关闭方。

**第一次挥手：**

此时两端还都处于ESTABLISHED状态，**客户端停止发送数据，并发送一个请求断开连接报文**(FIN)。

该报文首部中，**FIN=1，序列号seq=u**。

服务端接收到请求报文后，进入**CLOSE\_WAIT**(关闭等待)状态。

**也就是告诉服务端，我的数据发送完了，不会再给你发数据了。**

(另，在FIN包发送之前发送出去的数据，如果没有收到服务端对应的ACK确认报文，主动关闭方依然会重发这些数据)；**此时，客户端还可以接受数据**。

**第二次挥手：**

服务端收到**请求断开连接报文(FIN)**后，回复**确认报文(ACK)**，**确认号：ack=u+1，序列号seq=v。**

客户端收到**确认报文(ACK)**后，进入**FIN-WAIT-1(终止等待-1)状态**。

现在TCP连接处于**半开半闭**状态，服务端如果继续发送数据，客户端依然接收。

**第三次挥手：**

服务端发送一个**请求断开连接报文(FIN)**，用来服务端到客户端的数据传送。

该报文段首部中，FIN=1，ACK=1，**确认序列号ack=u+1，序列号seq=w。**

客户端接收到**确认断开连接报文(ACK、FIN)**后，进入**FIN-WAIT-2(终止等待-2)**状态。

也就是告诉客户端，**我的数据也发送完了，不会再给你发数据了**。

**第四次挥手：**

客户端收到**确认断开连接报文(ACK、FIN)**后，回复**确认报文(ACK)**给服务端。

确认序号ack=w+1。序列号=u+1。

然后进入**TIME\_WAIT(时间等待)**状态。

注意：

**此时，TCP连接还没有被释放，需要时间等待状态结束后(2MSL)，两端才会进入CLOSED状态。**

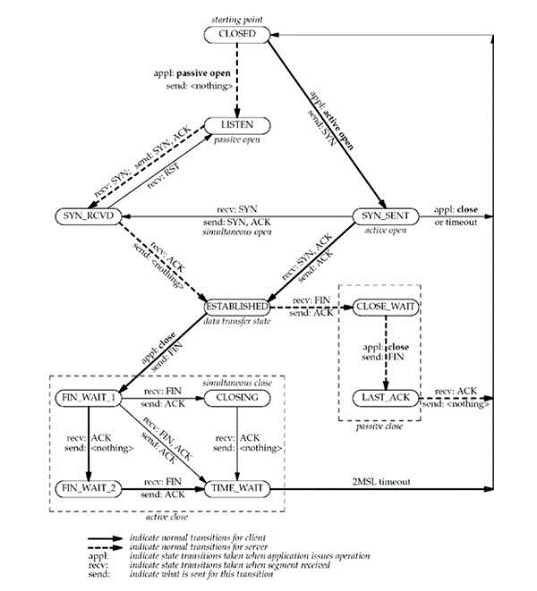
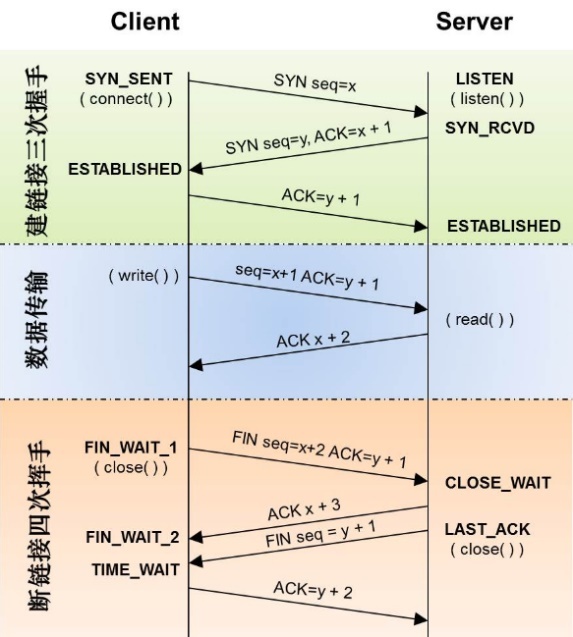
**设置时间等待是因为——最后一个确认报文可能会丢失，而需要重传。**

详情见：<https://www.cnblogs.com/huenchao/p/6266352.html>。

### TCP状态转换图

**三次握手：CLOSED、LISTEN、SYN\_SENT、SYN\_RCVD、ESTABLISHED。**

**四次挥手：FIN\_WAIT\_1，CLOSE\_WAIT、FIN\_WAIT\_2、LAST\_ACK、CLOSING、TIME\_WAIT。**



详情见：<https://blog.csdn.net/wenqian1991/article/details/40110703>。

### TCP状态中TIME\_WAIT作用

客户端接收到服务端的FIN报文后进入状态，此时并不是直接进入CLOSED状态，还需要等待一个时间计时器设置的时间，理由如下：

**1.确保最后一个确认报文段能够到达，如果B没有收到A发送来的确认报文段，那么就会重新发送连接释放请求报文段，A等待一段时间就是为了处理这种情况发生。**

**2.可能存在’已失效的连接请求报文段’，为了防止这种报文段出现在本次连接之外，需要等待一段时间。**

详情见；<https://www.cnblogs.com/huenchao/p/6266352.html>。

### TCP连接建立需要为什么不是两次握手

**防止失效的连接请求报文段被服务端接受，从而产生错误。**

PS：失效的连接请求：若客户端向服务端发送的连接请求丢失，客户端等待应答超时后就会再次发送连接请求，此时，上一个连接请求就是『失效的』。

若建立连接只需两次握手，客户端并没有太大的变化，仍然需要获得服务端的应答后才进入ESTABLISHED状态，而服务端在收到连接请求后就进入ESTABLISHED状态。此时如果网络拥塞，客户端发送的连接请求迟迟到不了服务端，客户端便超时重发请求，如果服务端正确接收并确认应答，双方便开始通信，通信结束后释放连接。此时，如果那个失效的连接请求抵达了服务端，由于只有两次握手，**服务端收到请求就会进入ESTABLISHED状态，等待发送数据或主动发送数据。但此时的客户端早已进入CLOSED状态，服务端将会一直等待下去，这样浪费服务端连接资源。**

详情见：<https://www.cnblogs.com/cenglinjinran/p/8482412.html>。

### TCP第三次握手失败会出现什么

**当失败时，服务器并不会重传ack报文，而是直接发送RTS报文段，进入CLOSED状态，防止SYN洪泛攻击。**

详情见：<https://blog.csdn.net/libaineu2004/article/details/79020031>。

### TCP长连接和短链接及优缺点

长连接：

从生命周期的角度来解释，长连接的生命周期和程序的生命周期一样，程序启动的时候会建立好连接，然后程序结束后才主动关闭这个连接。这里注意用到了一个关键词“主动”，也就是长连接过程中会有可能因为其它因素导致这个连接被动断开，这就要求我们在长连接中用什么方法保证连接还存在：

短连接：

同样从生命在后期来考虑，短连接的生命周期跟栈区的变量一样，好比某个函数里面的局部变量，当外面调用进入到某个函数内部，建立好连接，发送或者接收数据，离开函数后，该连接就关闭。

从上面的介绍得知，一个TCP的连接建立和关闭，通常是需要七个分节的。所以长连接和短连接的优缺点也就突显了出来。



总结：

1：如果业务来往比较频繁，则选择长连接。

2：如果server要主动给client发数据，则选择长连接。

详情见：<https://www.jianshu.com/p/813e89e115a8>。

### TCP拥塞控制-慢启动、拥塞避免、快重传、快启动

一般原理：发生拥塞控制的原因：资源(带宽、交换节点的缓存、处理机)的需求>可用资源。

**作用：拥塞控制就是为了防止过多的数据注入到网络中，这样可以使网络中的路由器或者链路不至于过载。**拥塞控制要做的都有一个前提：就是网络能够承受现有的网络负荷。

对比流量控制：拥塞控制是一个全局的过程，涉及到所有的主机、路由器、以及降低网络相关的所有因素。流量控制往往指点对点通信量的控制。是端对端的问题。

**拥塞窗口**

发送方为一个动态变化的窗口叫做拥塞窗口，拥塞窗口的大小取决于网络的拥塞程度。发送方让自己的发送窗口=拥塞窗口，但是发送窗口不是一直等于拥塞窗口的，在网络情况好的时候，拥塞窗口不断的增加，发送方的窗口自然也随着增加，但是接受方的接受能力有限，在发送方的窗口达到某个大小时就不在发生变化了。

发送方如果知道网络拥塞了呢？发送方发送一些报文段时，如果发送方没有在时间间隔内收到接收方的确认报文段，则就可以认为网络出现了拥塞。

**慢启动**

主机开发发送数据报时，如果立即将大量的数据注入到网络中，可能会出现网络的拥塞。**慢启动算法就是在主机刚开始发送数据报的时候先探测一下网络的状况**，如果网络状况良好，发送方每发送一次报文段都能正确的接受确认报文段。**那么就从小到大的增加拥塞窗口的大小，即增加发送窗口的大小。**

例子：开始发送方先设置cwnd（拥塞窗口）=1,发送第一个报文段M1，接收方接收到M1后，发送方接收到接收方的确认后，把cwnd增加到2，接着发送方发送M2、M3，发送方接收到接收方发送的确认后cwnd增加到4，慢启动算法每经过一个传输轮次（认为发送方都成功接收接收方的确认），拥塞窗口cwnd就加倍。

**拥塞控制**

为了防止cwnd增加过快而导致网络拥塞，所以需要设置一个慢启动开始门限ssthresh状态变量（我也不知道这个到底是什么，就认为他是一个拥塞控制的标识）,它的用法：

当cwnd < ssthresh,使用慢启动算法，

当cwnd > ssthresh,使用拥塞控制算法，停用慢启动算法。

当cwnd = ssthresh，这两个算法都可以。

拥塞避免的思路：是让cwnd缓慢的增加而不是加倍的增长，每经历过一次往返时间就使cwnd增加1，而不是加倍，这样使cwnd缓慢的增长，比慢启动要慢的多。

无论是慢启动算法还是拥塞避免算法，只要判断网络出现拥塞，就要把慢启动开始门限(ssthresh)设置为设置为发送窗口的一半（>=2），cwnd(拥塞窗口)设置为1，然后在使用慢启动算法，这样做的目的能迅速的减少主机向网络中传输数据，使发生拥塞的路由器能够把队列中堆积的分组处理完毕。**拥塞窗口是按照线性的规律增长，比慢启动算法拥塞窗口增长块的多。**

实例：  
TCP连接进行初始化的时候，cwnd=1,ssthresh=16。  
在慢启动算法开始时，cwnd的初始值是1，每次发送方收到一个ACK拥塞窗口就增加1，当ssthresh =cwnd时，就启动拥塞控制算法，拥塞窗口按照规律增长，  
当cwnd=24时，网络出现超时，发送方收不到确认ACK，此时设置ssthresh=12,(二分之一cwnd),设置cwnd=1,然后开始慢启动算法，当cwnd=ssthresh=12,慢启动算法变为拥塞控制算法，cwnd按照线性的速度进行增长。

AIMD(加法增大乘法减小)

乘法减小：无论在慢启动阶段还是在拥塞控制阶段，只要网络出现超时，就是将cwnd置为1，ssthresh置为cwnd的一半，然后开始执行慢启动算法（cwnd<ssthresh）。

加法增大：当网络频发出现超时情况时，ssthresh就下降的很快，为了减少注入到网络中的分组数，而加法增大是指执行拥塞避免算法后，是拥塞窗口缓慢的增大，以防止网络过早出现拥塞。  
这两个结合起来就是AIMD算法，是使用最广泛的算法。拥塞避免算法不能够完全的避免网络拥塞，通过控制拥塞窗口的大小只能使网络不易出现拥塞。

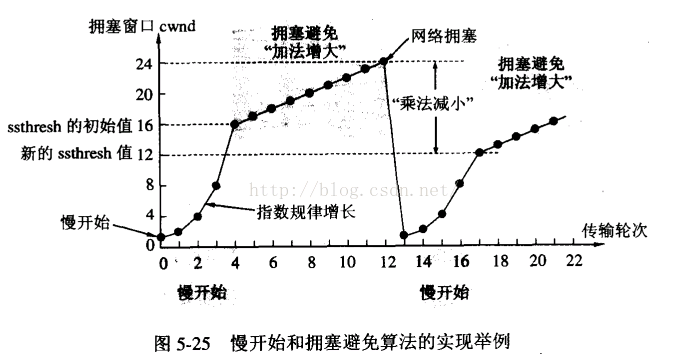
**快重传**

快重传算法要求首先接收方收到一个失序的报文段后就立刻发出重复确认，而不要等待自己发送数据时才进行捎带确认。接收方成功的接受了发送方发送来的M1、M2并且分别给发送了ACK，现在接收方没有收到M3，而接收到了M4，显然接收方不能确认M4，因为M4是失序的报文段。如果根据可靠性传输原理接收方什么都不做，但是按照快速重传算法，在收到M4、M5等报文段的时候，不断重复的向发送方发送M2的ACK,如果接收方一连收到三个重复的ACK,那么发送方不必等待重传计时器到期，由发送方尽早重传未被确认的报文段。

**快恢复**

1：当发送发连续接收到三个确认时，就执行乘法减小算法，把慢启动开始门限（ssthresh）减半，但是接下来并不执行慢开始算法。

2：此时不执行慢启动算法，而是把cwnd设置为ssthresh的一半， 然后执行拥塞避免算法，使拥塞窗口缓慢增大。



详情见：<http://www.cnblogs.com/bincoding/p/8976157.html>。

### TCP如何保证可靠性传输

确保传输可靠性的方式

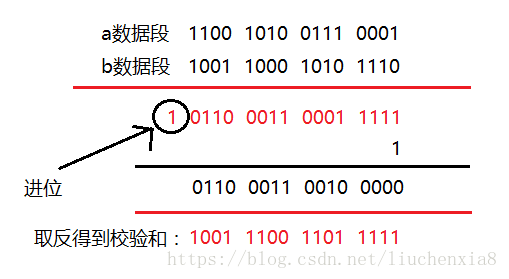
TCP协议保证数据传输可靠性的方式主要有：**校验和、**[**序列号**](https://www.baidu.com/s?wd=%E5%BA%8F%E5%88%97%E5%8F%B7&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)、**确认应答、超时重传**、**连接管理**、**流量控制**、**拥塞控制。**

**校验和**

计算方式：在数据传输的过程中，将发送的数据段都当做一个16位的整数。将这些整数加起来。并且前面的进位不能丢弃，补在后面，最后取反，得到校验和。

发送方：在发送数据之前计算检验和，并进行校验和的填充。

接收方：收到数据后，对数据以同样的方式进行计算，求出校验和，与发送方的进行比对。



注意：如果接收方比对校验和与发送方不一致，那么数据一定传输有误。但是如果接收方比对校验和与发送方一致，**数据不一定传输成功。**

**确认应答与序列号**

序列号：TCP传输时将每个字节的数据都进行了编号，这就是序列号。

确认应答：TCP传输的过程中，每次接收方收到数据后，都会对传输方进行确认应答。也就是发送ACK报文。这个ACK报文当中带有对应的确认序列号，告诉发送方，接收到了哪些数据，下一次的数据从哪里发。



序列号的作用不仅仅是应答的作用，有了序列号能够将接收到的数据根据序列号排序，并且去掉重复序列号的数据。这也是TCP传输可靠性的保证之一。

**超时重传**

在进行TCP传输时，由于确认应答与序列号机制，也就是说发送方发送一部分数据后，都会等待接收方发送的ACK报文，并解析ACK报文，判断数据是否传输成功。如果发送方发送完数据后，迟迟没有等到接收方的ACK报文，这该怎么办呢？而没有收到ACK报文的原因可能是什么呢？

首先，发送方没有介绍到响应的ACK报文原因可能有两点：

数据在传输过程中由于网络原因等直接全体丢包，接收方根本没有接收到。

接收方接收到了响应的数据，但是发送的ACK报文响应却由于网络原因丢包了。

TCP在解决这个问题的时候引入了一个新的机制，叫做超时重传机制。**简单理解就是发送方在发送完数据后等待一个时间，时间到达没有接收到ACK报文，那么对刚才发送的数据进行重新发送。**如果是刚才第一个原因，接收方收到二次重发的数据后，便进行ACK应答。如果是第二个原因，接收方发现接收的数据已存在（判断存在的根据就是序列号，所以上面说序列号还有去除重复数据的作用），那么直接丢弃，仍旧发送ACK应答。

那么发送方发送完毕后等待的时间是多少呢？如果这个等待的时间过长，那么会影响TCP传输的整体效率，如果等待时间过短，又会导致频繁的发送重复的包。如何权衡？

由于TCP传输时保证能够在任何环境下都有一个高性能的通信，因此这个最大超时时间（也就是等待的时间）是动态计算的。

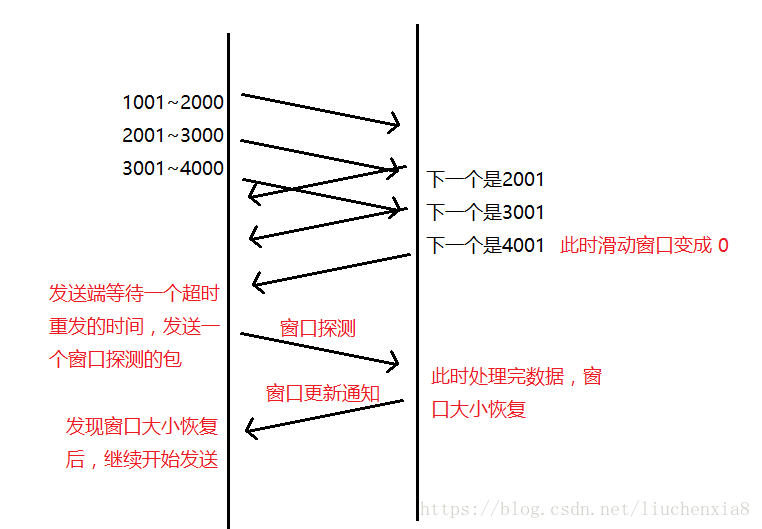
**连接管理**

**连接管理就是三次握手与四次挥手的过程**，在前面详细讲过这个过程，这里不再赘述。保证可靠的连接，是保证可靠性的前提。

**流量控制**

接收端在接收到数据后，对其进行处理。如果发送端的发送速度太快，导致接收端的结束缓冲区很快的填充满了。此时如果发送端仍旧发送数据，那么接下来发送的数据都会丢包，继而导致丢包的一系列连锁反应，超时重传呀什么的。**而TCP根据接收端对数据的处理能力，决定发送端的发送速度，这个机制就是流量控制。**

在TCP协议的报头信息当中，有一个16位字段的窗口大小。在介绍这个窗口大小时我们知道，窗口大小的内容实际上是接收端接收数据缓冲区的剩余大小。这个数字越大，证明接收端接收缓冲区的剩余空间越大，网络的吞吐量越大。接收端会在确认应答发送ACK报文时，将自己的即时窗口大小填入，并跟随ACK报文一起发送过去。而发送方根据ACK报文里的窗口大小的值的改变进而改变自己的发送速度。如果接收到窗口大小的值为0，那么发送方将停止发送数据。并定期的向接收端发送窗口探测数据段，让接收端把窗口大小告诉发送端。



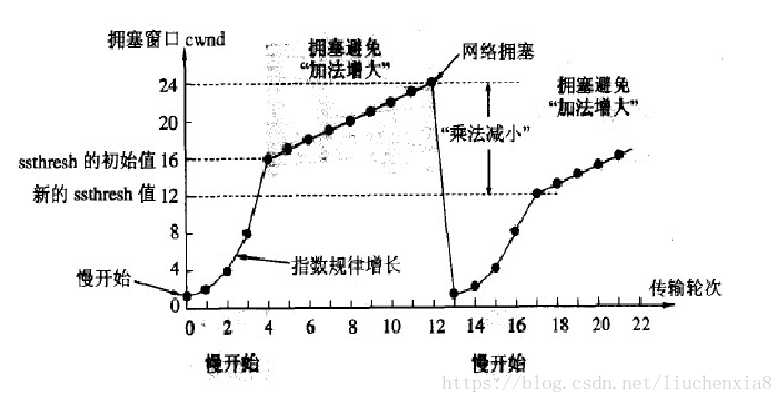
注：16位的窗口大小最大能表示65535个字节（64K），但是TCP的窗口大小最大并不是64K。在TCP首部中40个字节的选项中还包含了一个窗口扩大因子M，实际的窗口大小就是16为窗口字段的值左移M位。每移一位，扩大两倍。

**拥塞控制**

TCP传输的过程中，发送端开始发送数据的时候，如果刚开始就发送大量的数据，那么就可能造成一些问题。网络可能在开始的时候就很拥堵，如果给网络中在扔出大量数据，那么这个拥堵就会加剧。拥堵的加剧就会产生大量的丢包，就对大量的超时重传，严重影响传输。

所以TCP引入了慢启动的机制，在开始发送数据时，先发送少量的数据探路。探清当前的网络状态如何，再决定多大的速度进行传输。这时候就引入一个叫做拥塞窗口的概念。发送刚开始定义拥塞窗口为 1，每次收到ACK应答，拥塞窗口加 1。在发送数据之前，首先将拥塞窗口与接收端反馈的窗口大小比对，取较小的值作为实际发送的窗口。

拥塞窗口的增长是指数级别的。慢启动的机制只是说明在开始的时候发送的少，发送的慢，但是增长的速度是非常快的。为了控制拥塞窗口的增长，不能使拥塞窗口单纯的加倍，设置一个拥塞窗口的阈值，当拥塞窗口大小超过阈值时，不能再按照指数来增长，而是线性的增长。在慢启动开始的时候，慢启动的阈值等于窗口的最大值，一旦造成网络拥塞，发生超时重传时，慢启动的阈值会为原来的一半（这里的原来指的是发生网络拥塞时拥塞窗口的大小），同时拥塞窗口重置为 1。



**拥塞控制是TCP在传输时尽可能快的将数据传输，并且避免拥塞造成的一系列问题。是可靠性的保证，同时也是维护了传输的高效性。**

详情见：<https://blog.csdn.net/liuchenxia8/article/details/80428157>。

### TCP如何解决粘包、拆包问题

我们都知道TCP属于传输层的协议，传输层除了有TCP协议外还有UDP协议。那么UDP是否会发生粘包或拆包的现象呢？答案是不会。UDP是基于报文发送的，**从UDP的帧结构可以看出，在UDP首部采用了16bit来指示UDP数据报文的长度**，因此在应用层能很好的将不同的数据报文区分开，从而避免粘包和拆包的问题。而TCP是基于字节流的，虽然应用层和TCP传输层之间的数据交互是大小不等的数据块，**但是TCP把这些数据块仅仅看成一连串无结构的字节流，没有边界**；**另外从TCP的帧结构也可以看出，在TCP的首部没有表示数据长度的字段**，基于上面两点，在使用TCP传输数据时，才有粘包或者拆包现象发生的可能。

粘包、拆包表现形式

现在假设客户端向服务端连续发送了两个数据包，用packet1和packet2来表示，那么服务端收到的数据可以分为三种，现列举如下：

第一种情况，**接收端正常收到两个数据包，即没有发生拆包和粘包的现象**，此种情况不在本文的讨论范围内。



第二种情况，接收端只收到一个数据包，由于TCP是不会出现丢包的，所以**这一个数据包中包含了发送端发送的两个数据包的信息，这种现象即为粘包**。这种情况由于接收端不知道这两个数据包的界限，所以对于接收端来说很难处理。



第三种情况，这种情况有两种表现形式，如下图。**接收端收到了两个数据包，但是这两个数据包要么是不完整的，要么就是多出来一块，这种情况即发生了拆包和粘包**。这两种情况如果不加特殊处理，对于接收端同样是不好处理的。



粘包、拆包发生原因

发生TCP粘包或拆包有很多原因，现列出常见的几点：

1：要发送的数据大于TCP**发送缓冲区剩余空间**大小，将会发生拆包。

2：待发送数据大于MSS（**最大报文长度**），TCP在传输前将进行拆包。

3：**要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包。**

4：**接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包**。等等。。。

粘包、拆包解决办法

通过以上分析，我们清楚了粘包或拆包发生的原因，那么如何解决这个问题呢？解决问题的关键在于如何给每个数据包添加边界信息，常用的方法有如下几个：

1、**发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度**，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。

2、**发送端将每个数据包封装为固定长度（不够的可以通过补0填充），**这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。

3、**可以在数据包之间设置边界**，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。等等。。。

### UDP如何实现TCP可靠传输

**设计方法：1.添加seq/ack机制，确保数据发送到对端。2.添加发送和接受缓冲区，主要是用于超时重传。3.添加超时重传机制。**

详细说明：发送端发送数据时，生成一个随机seq=x，然后每一片按照数据大小分配seq。数据到达接收端后接收端放入缓存，并发送一个ack=x的包，表示对方收到了数据。发送端收到了ack包后，删除缓冲区对应的数据。**并且时间到达后，定时任务检查是否需要重传数据。**

### IP知识点总结

IP协议负责将数据包发送给最终目标计算机，能够让世界上任何两台计算机进行通信。  
文章介绍IP的主要功能及其协议具体内容的知识点。

**定义**

IP（Internet Protocol）即网络之间互连的协议。

IP相当于OSI参考模型的第三层——网络层。网络层的功能是基于IP地址进行不同网络系统间的路径选择。

IP作用：对不同数据链路的相异特性进行抽象化，在复杂的网络环境中将数据包发给最终的目标地址。

如果以上内容感觉有点不知所言，可以参考下OSI模型中网络层与数据链路层的关系：

网络层的下一层——数据链路层的作用是在同一数据链路的节点之间进行数据传递。通过使用接收系统的MAC地址来寻址，将比特信息封装成数据帧Frame。但是不同网络，如以太网、分组交换网等，由于它们所传送数据的基本单元（“帧”）的格式不同，它们相互之间不能互通。而IP协议把各种不同“帧”统一转换成“网协数据包”格式，使所有各种计算机都能在因特网上实现互通。

数据链路层提供直连**同一网络下的两个设备**之间的通信功能。网络层的IP则负责在**两个不同网络网络**之间进行通信传输。

**IP的两大功能**

路由控制

含义：指分组数据发送到最终目标地址的功能。

作用：即使网络非常复杂，通过路由控制就可以确定到达目标地址的通路。

例子：坐火车，每到一站再向车站工作人员打听接下来该坐什么车。“各跳之间无计划传输”。

火车-数据链路，旅客-IP数据包，车站工作人员-路由器。

路由控制表：所有主机都维护着一张路由控制表（Routing Table），记录了IP数据在下一步应该发给哪个路由器。IP包根据这个路由表在各个数据链路上传输。

数据链路的抽象化

**IP对不同数据链路进行了抽象。**那么不同数据链路之间最大的区别是：他们各自的最大传输单位（MTU，Maximum Transmission Unit）不同。IP会进行分片处理（IP Fragmentation），将较大的IP包分成多个较小的IP包。

从网络层上看，可以忽略数据包在各个数据链路上的MTU，只需要按照原地址发送的长度接收数据包。

**IP属于面向无连接型**

无连接型含义：发包之前，不需要建立与对端目标地址之间的连接。上次如果有需要发送给IP的数据，该数据会立即被压缩成IP包发出去。

缺点：产生冗余的通信。

为什么IP要采用面向无连接？

简化：面向连接相对复杂，管理每个连接本身就是一个很繁琐的事情。

提速：每次通信之前都要事先建立连接，会降低处理速度。需要有连接时，可以委托上一层提供此项服务。

**IP地址的基础知识**

IP地址（IPv4地址）

定义：用32位二进制数来表示。以每8位为一组，分成4组，每组用“.”隔开，再将每组数转成十进制数。

规则：

最多允许43亿台（2^32）计算机连接到网络。

每一块网卡（NIC）都得设置IP地址。

IP地址由网络和主机两部分标识而成。

网络标识在数据链路的每个段配置不同的值，同段内的主机的网络地址是相同的。

主机标识不允许在同一网段内重复出现。

通过设置网络地址和主机地址，保证了整个互联网中主机的IP地址都是\**唯一*的。

IP地址的分类

A类地址

定义：首位0开头，后24位是主机标识  
网络地址范围：0.0.0.0~127.0.0.0  
一个网段内可以容纳的主机上限是2^24-2个

B类地址

定义：前两位是10的地址，后16位是主机标识  
网络地址范围：128.0.0.1~191.255.0.0  
一个网段可以容纳的主机上限是2^16-2个

C类地址

定义：前三位是110的地址，前24位是网络标识，后8位是主机标识。  
网络地址范围：192.0.0.0~239.255.255.0  
一个网段内可容纳的主机地址上限是2^8-2个(254个)

D类地址

定义：前四位是1110的地址，前32位是网络标识  
网络地址范围：224.0.0.0~239.255.255.255  
D类地址没有主机标识，常被用于多播

关于分配IP主机地址的注意事项：

主机地址不可以全部为0或全部为1。  
全部为0标识对应的网络地址或IP地址不可获知。  
全部为1的主机地址通常作为广播地址。  
所以以上每个网段的主机数量上限是2^n-2个。

**广播地址**

广播地址定义：IP地址的主机标识全部设置为1。

广播定义：向某个网段的广播地址发送IP包，这个网段的所有主机都能收到这个包，由主机IP之上的一层去判断是否接收数据。

缺点：给毫无关系的网络或主机带来影响，造成不必要的流量。

**子网掩码**

作用

理论上B类网络一个链路允许65000多台计算机连接。然而实际网络架构中，一般不会有在同一连路上连接65000多台计算机的情况。

因此，直接使用A\B\C类地址，有点浪费资源，人们开始采用一种新的组合方式减少地址空间的浪费。

**子网掩码可以通过子网网络地址细分出比A\B\C类更小粒度的网络**。

定义

子网掩码用二进制方式表示，也是32位二进制数字，

规则

1表示IP地址的网络地址对应位。

0表示IP地址的主机地址对应位。

特点

子网掩码可以灵活指定网络标识的长度，从而网络粒度更细。

**路由控制**

作用：仅仅有IP地址不足以将数据发向目标地址，在数据发送过程中需要“指明路由器或主机”的信息，以便真正发往目标地址。保存这种信息的就是路由控制表（Routing Table）。该表是由一个“路由协议”（有别于IP协议）制作而成的。

路由控制与IP地址

默认路由

主机路由

环回地址

同一台计算机的程序之间进行网络通信，使用一个特殊IP地址 127.0.0.1作为环回地址，与该地址等价的是localhost主机名。使用这个IP或主机名，数据包不会发向网络。

路由控制表的聚合

又称路由汇总（Aggregation）。

作用：路由表越大，管理它需要的内存和CPU也就越大，而且查找路由表的时间就越长，导致转发IP数据包的性能下降。如果要构建大规模、高性能网络，需要尽可能削减路由表的大小。

利用网络地址的比特分步进行分层配置。对内即使有多个子网掩码，对外呈现出的也是同一个网络地址。

IP对数据的处理

背景&作用

每种数据链路的MTU之所以不同，是因为每个不同类型的数据链路的使用目的不同，可承载的MTU也就不同。

IP数据传递到下一层数据链路层时，可能需要进行分片处理（IP Fragmentation)。

规则

按照路径中存在的所有数据链路中 最小的MTU发送，可以避免中途的路由器进行分片处理。

**IPv6**

定义

**8个16位，共128比特，是IPv4长度的4倍。**

作用

从根本上解决IPv4地址耗尽的问题。

特点

依旧适应互联网分层构造，路由控制上也尽可能避免路由表膨大

性能提升，简化首部，路由器不再分片，通过路径MTU只由发送端主机进行分片处理

即插即用

采用认证与加密功能

多播、Moblie IP比IPv4更顺利使用

规则

每16位比特为一组，用":"隔开。出现连续的0时可以省略，用"::"表示。一个IP地址只允许出现一次“::”。

**IP包的具体内容**

IPv4首部

IP首部包含着 用IP协议进行发包控制时 所有的必要信息。  
首部以每32比特（8字节）为一个单位：

版本号：4比特，IPv4即值为4

首部长度（IHL：Internet Header Length）：IP首部大小。4比特，单位是4字节。值为5时，IP首部长度为4\*5=20字节。

区分服务（TOS：Type Of Service）：表明服务质量。8比特，现已划分为DSCP和ECN两个字段。

总长度（Total Length）：表示IP首部与数据部分的总字节数。该字段长16比特。因此IP包最大长度是65535字节。

标识（ID：Identification）:用于分片重组，该字段长16比特。同一分片的标识值相同，不同分片的标识值不同。

标志（Flags）:表示包被分片的相关信息，该字段长3比特。可以表示 是否使用分片、是否分片的最后一个包。

片偏移（FO：Fragment Offset）:标识被分片的每一个分段相对于原始数据的位置。该字段长13比特。单位是8字节，因此最大可以表示8\* 2^13 =65536字节的位置。

生存时间（TTL：Time To Live）：指可以中转多少个路由器，没经过一个路由器，TTL会减1，该字段长度为8比特。因此一个包的中转路由次数不会超过2^8=256次。由此可以避免IP包在网络内无限传递的问题。

协议（Protocol）：表示IP首部的下一个首部属于哪个协议（其实就是上层协议）。该字段长度8比特。

首部校验和（Header Checksum）：按照一定算法校验数据报的首部，不校验数据部分，主要来确保IP数据报不被破坏。该字段长度16比特。

源地址（Source Address）：32比特，表示发送端IP地址。

目标地址（Destination Address）：32比特，表示接收端IP地址。

可选项（Options）:  
长度可变，通常只用于实验或诊断时使用。包含如下几点信息：

安全级别

源路径

路径记录

时间戳

填充（Padding）：再有可选项的情况下，首部长度可能不是32比特的整数倍。为此，通过向该字段填充0，调整为32比特的整数倍。

IP包数据部分

OSI模型网络层之下 数据链路层的数据帧 就是IP包的数据部分。IP首部是对数据进行包装，增加了网络层的控制信息。

**IP协议相关技术**

IP协议除了让最终目标主机收到数据包，还要求能够解析主机名称和MAC地址，以及数据包在发送过程中异常情况处理的功能。

本章介绍了作为IP的辅助和扩展规范的DNS、ARP、ICMP以及DHCP等协议。

仅凭IP无法完成通信

需要将IP上层的应用层用到的地址映射为IP地址。

同时还需要知悉IP下层的数据链路层的MAC地址。

ICMP

辅助IP的ICMP

背景

架构IP网络时需要注意：确认网络是否正常工作，以及遇到异常时进行问题诊断。

作用

确定IP包是否成功送达目标地址

通知发在发送过程中IP包被废弃的原因

改善网络设置  
根据以上，就可以获得网路是否正常、设置是否有误以及设备有何异常等信息，从而便于进行网络上的问题诊断。

网络的设置可以包括很多内容：网线、IP地址和子网掩码的设置、路由表的和值、DNS服务器的设置、邮件服务器的设置以及代理服务器的设置等。而ICMP只负责与IP相关的设置。

定义

ICMP的包以明文的形式像TCP、UDP一样通过IP进行传输。然而，ICMP所承担的功能并非传输层的补充，而应该**把它视为IP的一部分**。

**ICMP的消息分为两类：**

**通知出错原因的错误消息**

**用于诊断的查询消息**

例子

ICMP超时消息

IP包首部有一个字段TTL，每经过一个路由器就会减1，直到减到0时IP报就会被丢弃。此时，IP路由器将会发送一个ICMP超时的消息给发送端，通知该包已被丢弃。

ICMP回送消息

用于进行通信的主机或路由器之间，**判断所发送的数据包是否已经成功到达对端**的一种消息。

向对端主机发送 回送请求消息（ICMP Echo Request Message）,接收对端主机发回来的 回送应答消息（ICMP Echo Reply Message）。

ping命令（Packet InterNetwork Groper） 就是利用这个消息实现的。

还有ICMP路由器探索消息用于发现与自己相连网络中的路由器、ICMP地址掩码消息获取子网掩码的信息。

ICMPv6

在IPv6中，ICMP作用被扩大，如果没有ICMPv6，IPv6就无法正常通信。ICMPv6集成了IPv4的ARP、ICMP重定向以及ICMP路由器选择消息等功能。

详情见：<https://www.jianshu.com/p/1183208dd5e3>。

### IP地址和子网掩码

网际协议IP是TCP/IP体系中最主要的两个协议之一。与IP协议配套使用的还有三个协议：

1：地址解析协议ARP

2：网际控制报文协议ICMP

3：网际组管理协议IGMP

4：逆地址解析协议RARP

每一类地址都由两个固定的长度的字段表示，其中第一个字段是网络号（net-id），它标识主机（或路由器）所连接到的网络。一个网络号在整个因特网范围内必须是唯一的。第二个字段是主机号（host-id），它标识该主机（或路由器），一个主机号在它前面的网络号所指明的网络范围内必须是唯一的。由此可见，一个ip地址在整个因特网范围是是唯一的。

使用子网掩码的好处是：不管网络有没有划分子网，只要把子网掩码和IP地址进行逐位的“与”运算（AND），就立刻得出**网络地址**来。这样路由器在处理到来的分组时就可以采用同样的算法。

### ARP解析过程

ARP是一种能够实现IP地址到物理地址的转化协议，以目标地址为线索用来定义下一位应该接收数据分包的网络设备对应的MAC地址，如果目标主机不在同一个数据链路层的话，可以通过ARP查找下一跳路由器的MAC地址，**ARP只识用于IPV4，IPV6可以使用ICMPV6替代。**

ARP工作过程：

**ARP借助ARP请求包和响应包来确定MAC地址。**

例如：处在同一链路上的主机A 向B发送一个IP请求包，且互不知MAC地址：主机为了获取主机B的 MAC地址，起初通过广播发送一个**ARP请求包**，此包包含想要获取MAC地址主机的IP地址，因此ARP的请求包中会被同一链路上的所有主机或路由解析，如果ARP请求包中的目标IP与自己的IP地址相同，那么此节点就将自己MAC地址塞入ARP响应包，返回给主机A。

ARP缓存表：

如果每发送一次IP数据包就进行一次ARP请求获取MAC地址，会造成不必要的网络流量。

通常做法就是把获取的MAC地址缓存一端时间，即把第一次通过ARP获取的MAC地址作为IP对MAC的映射关系记忆，下一次如果若向该IP发送数据的时候，直接使用缓存表中的多应的MAC地址进行通信即可。当然每执行一次ARP请求去，其对应的ARP都会被清除。

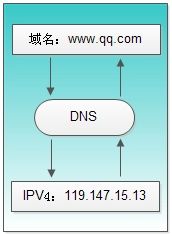
### DNS原理

为什么需要DNS解析域名为IP地址？

**网络通讯大部分是基于TCP/IP的，而TCP/IP是基于IP地址的**，所以计算机在网络上进行通讯时只能识别如“202.96.134.133”之类的IP地址，而不能认识域名。我们无法记住10个以上IP地址的网站，所以我们访问网站时，更多的是在浏览器地址栏中输入域名，就能看到所需要的页面，这是因为有一个叫“DNS服务器”的计算机自动把我们的域名“翻译”成了相应的IP地址，然后调出IP地址所对应的网页。

具体什么是DNS？

**DNS( Domain Name System)是“域名系统”的英文缩写，是一种组织成域层次结构的计算机和网络服务命名系统，它用于TCP/IP网络，它所提供的服务是用来将主机名和域名转换为IP地址的工作。**DNS就是这样的一位“翻译官”，它的基本工作原理可用下图来表示。



DNS 的过程？

关于DNS的获取流程：

DNS是应用层协议，事实上他是为其他应用层协议工作的，包括不限于HTTP和SMTP以及FTP，用于将用户提供的主机名解析为ip地址。

具体过程如下：

1：用户主机上运行着DNS的客户端，就是我们的PC机或者手机客户端运行着DNS客户端了

2：浏览器将接收到的url中抽取出域名字段，就是访问的主机名，比如http://www.baidu.com/, 并将这个主机名传送给DNS应用的客户端

3：DNS客户机端向DNS服务器端发送一份查询报文，报文中包含着要访问的主机名字段（中间包括一些列缓存查询以及分布式DNS集群的工作）

4：该DNS客户机最终会收到一份回答报文，其中包含有该主机名对应的IP地址

5：一旦该浏览器收到来自DNS的IP地址，就可以向该IP地址定位的HTTP服务器发起TCP连接

详情见：<https://www.cnblogs.com/gopark/p/8430916.html>。

### HTTP1.0、HTTP1.1、HTTP2.0区别

**HTTP1.0、HTTP 1.1主要区别**

      长链接

HTTP 1.0需要使用keep-alive参数来告知服务器端要建立一个长连接，而HTTP1.1默认支持长连接。

  HTTP是基于TCP/IP协议的，创建一个TCP连接是需要经过三次握手的,有一定的开销，如果每次通讯都要重新建立连接的话，对性能有影响。因此最好能维持一个长连接，可以用个长连接来发多个请求。

节约带宽

HTTP 1.1支持只发送header信息(不带任何body信息)，如果服务器认为客户端有权限请求服务器，则返回100，否则返回401。客户端如果接受到100，才开始把请求body发送到服务器。

这样当服务器返回401的时候，客户端就可以不用发送请求body了，节约了带宽。

另外HTTP还支持传送内容的一部分。这样当客户端已经有一部分的资源后，只需要跟服务器请求另外的部分资源即可。这是支持文件断点续传的基础

HOST域

现在可以web server例如tomat，设置虚拟站点是非常常见的，也即是说，web serve上的多个虚拟站点可以共享同一个ip和端口。

HTTP1.0是没有host域的，HTTP1.1才支持这个参数。

**HTTP 1.1、HTTP2.0主要区别**

**多路复用**

**HTTP2.0使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。**

当然HTTP1.1也可以多建立几个TCP连接，来支持处理更多并发的请求，但是创建TCP连接本身也是有开销的。

TCP连接有一个预热和保护的过程，先检查数据是否传送成功，一旦成功过，则慢慢加大传输速度。因此对应瞬时并发的连接，服务器的响应就会变慢。所以最好能使用一个建立好的连接，并且这个连接可以支持瞬时并发的请求。

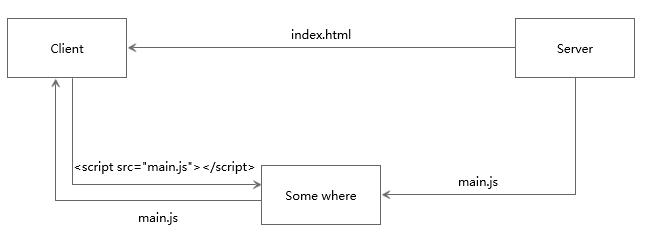
**数据压缩**

      HTTP1.1不支持header数据的压缩，HTTP2.0使用HPACK算法对header的数据进行压缩，这样数据体积小了，在网络上传输就会更快。

**服务器推送**

  当我们对支持HTTP2.0的web server请求数据的时候，服务器会顺便把一些客户端需要的资源一起推送到客户端，免得客户端再次创建连接发送请求到服务器端获取。这种方式非常合适加载静态资源。

  服务器端推送的这些资源其实存在客户端的某处地方，客户端直接从本地加载这些资源就可以了，不用走网络，速度自然是快很多的。



服务端推送过来的资源，会统一放在一个网络与http缓存之间的一个地方，在这里可以理解为“本地”。当客户端把index.html解析完以后，会向本地请求这个资源。由于资源已经本地化，所以这个请求的速度非常快，这也是服务端推送性能优势的体现之一。

### HTTP和HTTPS区别

**HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本体游览器的传输协议**，它可以是游览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的HTTP通道，**简单讲是HTTP的安全版，即HTTP下加入SSL层，因此加密的详细内容就需要SSL。**

**区别：1.HTTPS协议需要到CA申请证书，需要一定费用。2.HTTP是超文本传输协议，信息是明文传输，HTTPS则是安全性的SSL加密传输协议。3. HTTP和HTTPS使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。**

详情见：<https://www.cnblogs.com/wqhwe/p/5407468.html>。

### HTTP状态码

状态码的职责是当客户端向[服务器](https://www.baidu.com/s?wd=%E6%9C%8D%E5%8A%A1%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)发送请求时，描述返回的请求结果。借助状态码，用户可以知道服务器端是正常处理了请求还是出现了错误。

2XX——表明请求被正常处理了

1：200 OK：请求已正常处理。

2：204 No Content：请求处理成功，但没有任何资源可以返回给客户端，一般在只需要从客户端往服务器发送信息，而对客户端不需要发送新信息内容的情况下使用。

3：206 Partial Content：是对资源某一部分的请求，该状态码表示客户端进行了范围请求，而服务器成功执行了这部分的GET请求。响应报文中包含由Content-Range指定范围的实体内容。

3XX——表明浏览器需要执行某些特殊的处理以正确处理请求

4：301 Moved Permanently：资源的uri已更新，你也更新下你的书签引用吧。永久性重定向，请求的资源已经被分配了新的URI，以后应使用资源现在所指的URI。

5：302 Found：资源的URI已临时定位到其他位置了，姑且算你已经知道了这个情况了。临时性重定向。和301相似，但302代表的资源不是永久性移动，只是临时性性质的。换句话说，已移动的资源对应的URI将来还有可能发生改变。

6：303 See Other：资源的URI已更新，你是否能临时按新的URI访问。该状态码表示由于请求对应的资源存在着另一个URL，应使用GET方法定向获取请求的资源。303状态码和302状态码有着相同的功能，但303状态码明确表示客户端应当采用GET方法获取资源，这点与302状态码有区别。

当301,302,303响应状态码返回时，几乎所有的浏览器都会把POST改成GET，并删除请求报文内的主体，之后请求会自动再次发送。

7：304 Not Modified：资源已找到，但未符合条件请求。该状态码表示客户端发送附带条件的请求时（采用GET方法的请求报文中包含If-Match，If-Modified-Since，If-None-Match，If-Range，If-Unmodified-Since中任一首部）服务端允许请求访问资源，但因发生请求未满足条件的情况后，直接返回304.。

8：307 Temporary Redirect：临时重定向。与302有相同的含义。

4XX——表明客户端是发生错误的原因所在。

9：400 Bad Request：服务器端无法理解客户端发送的请求，请求报文中可能存在语法错误。

10：401 Unauthorized：该状态码表示发送的请求需要有通过HTTP认证（BASIC认证，DIGEST认证）的认证信息。

11：403 Forbidden：不允许访问那个资源。该状态码表明对请求资源的访问被服务器拒绝了。（权限，未授权IP等）

12：404 Not Found：服务器上没有请求的资源。路径错误等。

5XX——服务器本身发生错误

13：500 Internal Server Error：貌似内部资源出故障了。该状态码表明服务器端在执行请求时发生了错误。也有可能是web应用存在bug或某些临时故障。

14：503 Service Unavailable：抱歉，我现在正在忙着。该状态码表明服务器暂时处于超负载或正在停机维护，现在无法处理请求。

详情见；<https://blog.csdn.net/qq_35689573/article/details/82120851>。

### Nagle算法

**Nagle算法就是为了尽可能发送大块数据，避免网络中充斥着许多小数据快**。它的基本定义是任意时刻，最多只能有一个未被确认的小段。所谓小段，指的是小于MSS尺寸的数据块，所谓“未被确认”，是指一个数据块发送出去后，没有收到对方发送的ACK确认该数据已收到。

Nagle算法的规则（可参考tcp\_output.c文件里tcp\_nagle\_check函数注释）：

      （1）如果包长度达到MSS，则允许发送；

      （2）如果该包含有FIN，则允许发送；

      （3）设置了TCP\_NODELAY选项，则允许发送；

      （4）未设置TCP\_CORK选项时，若所有发出去的小数据包（包长度小于MSS）均被确认，则允许发送；

      （5）上述条件都未满足，但发生了超时（一般为200ms），则立即发送。

        Nagle算法只允许一个未被ACK的包存在于网络，它并不管包的大小，因此它事实上就是一个扩展的停-等协议，只不过它是基于包停-等的，而不是基于字节停-等的。Nagle算法完全由TCP协议的ACK机制决定，这会带来一些问题，比如如果对端ACK回复很快的话，Nagle事实上不会拼接太多的数据包，虽然避免了网络拥塞，网络总体的利用率依然很低。

        Nagle算法是silly window syndrome(SWS)预防算法的一个半集。SWS算法预防发送少量的数据，Nagle算法是其在发送方的实现，而接收方要做的时不要通告缓冲空间的很小增长，不通知小窗口，除非缓冲区空间有显著的增长。这里显著的增长定义为完全大小的段（MSS）或增长到大于最大窗口的一半。

详情见：<https://blog.csdn.net/occupy8/article/details/48174307>。

## 数据库

### 数据库类别

数据库种类大体分为：关系型数据库和非关系型数据库

**关系型数据库**

**关系型数据库模型是把复杂的**[**数据结构**](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=19&is_app=0&jk=358e3e6e069fd65f&k=%CA%FD%BE%DD%BD%E1%B9%B9&k0=%CA%FD%BE%DD%BD%E1%B9%B9&kdi0=0&luki=3&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=5fd69f066e3e8e35&ssp2=1&stid=0&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http%3A%2F%2Fwww.bubuko.com%2Finfodetail-469185.html&urlid=0)**归结为简单的二元关系（即二维表格形式）**。在关系型数据库中，对数据的操作几乎全部建立在一个或多个关系表格上，通过对这些关联的表格分类、合并、连接或选取等运算来实现数据库的管理。  
典型产品：Mysql、Oracle、DB2、Sqlserver

**非关系型数据库**

NoSQL是非关系型数据库的广义定义，如下小结：

NOSQL不是否定关系数据库，而是作为关系数据库的一个重要补充。

NOSQL为了高性能、高并发而生，忽略影响高性能、高并发的功能。

NOSQL典型产品memcached（纯内存），redis（持久化缓存），mongodb（面向文档）

**1：键值存储数据库（key-value）**

  键值数据库就类似传统语言中使用的哈希表。可以通过key来添加、查询或者删除数据库，因为使用key主键访问，所以会获得很高的性能及扩展性。

键值数据库主要使用一个哈希表，这个表中有一个特定的键和一个指针指向特定的数据。Key/value模型对于IT系统来说的优势在于简单、易部署、高并发。

典型产品：Memcached、**Redis**、MemcacheDB:

2：列存储（Column-oriented）数据库:

列存储数据库将数据存储在列族中，一个列族存储经常被一起查询的相关数据，比如人类，我们经常会查询某个人的姓名和年龄，而不是薪资。这种情况下姓名和年龄会被放到一个列族中，薪资会被放到另一个列族中。

这种数据库通常用来应对分布式存储海量数据。

典型产品：Cassandra、**HBase**

3：面向文档（Document-Oriented）数据库:

文档型数据库的灵感是来自于Lotus Notes办公软件，而且它同第一种键值数据库类似。该类型的数据模型是版本化的文档，半结构化的文档以特定的格式存储，比如JSON。文档型数据库可以 看作是键值数据库的升级版，允许之间嵌套键值。而且文档型数据库比键值数据库的查询效率更高。

面向文档数据库会将数据以文档形式存储。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名词与对应值，值既可以是简单的数据类型，如字符串、数字和日期等；也可以是复杂的类型，如有序列表和关联对象。数据存储的最小单位是文档，同一个表中存储的文档属性可以是不同的，数据可以使用XML、JSON或JSONB等多种形式存储。

典型产品：MongoDB、CouchDB

4：图形数据库

图形数据库允许我们将数据以图的方式存储。实体会被作为顶点，而实体之间的关系则会被作为边。比如我们有三个实体，Steve Jobs、Apple和Next，则会有两个“Founded by”的边将Apple和Next连接到Steve Jobs。

典型产品：Neo4J、InforGrid

### 关系型数据库和非关系型数据库区别

**关系型数据库最典型的数据结构是表，由二维表及其之间的联系所组成的一个数据组织。**

优点：

1：易于维护：**都是使用表结构**，格式一致；

2：使用方便：**SQL语言通用**，可用于复杂查询；

3：复杂操作：支持SQL，可用于一个表以及多个表之间非常复杂的查询。

缺点：

1：**读写性能比较差，尤其是海量数据的高效率读写；**

2：**固定的表结构，灵活度稍欠；**

3：**高并发读写需求**，传统关系型数据库来说，硬盘I/O是一个很大的瓶颈。

**非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方式的集合，可以是文档或者键值对等。**

优点：

1：**格式灵活**：存储数据的格式可以是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，使用灵活，应用场景广泛，而关系型数据库则只支持基础类型。

2：**速度快：nosql可以使用硬盘或者随机存储器作为载体，而关系型数据库只能使用硬盘；**

3：**高扩展性**；

4：**成本低**：nosql数据库部署简单，基本都是**开源**软件。

缺点：

1：**不提供sql支持**，学习和使用成本较高；

2：无事务处理；

3：数据结构相对复杂，复杂查询方面稍欠。

详情见：<https://blog.csdn.net/aaronthon/article/details/81714528>。

### SQL常见语句（每天敲一敲））

数据库

# 查看所有的数据库

SHOW DATABASES ;

# 创建一个数据库

CREATE DATABASE name;

# 删除一个数据库

DROP DATABASE name;

# 使用这个数据库

USE name;

表

# 查看所有的表

SHOW TABLES ;

# 创建一个表

CREATE TABLE n(id INT, name VARCHAR(10));

CREATE TABLE m(id INT, name VARCHAR(10), PRIMARY KEY (id), FOREIGN KEY (id) REFERENCES n(id), UNIQUE (name));

CREATE TABLE m(id INT, name VARCHAR(10));

# 直接将查询结果导入或复制到新创建的表

CREATE TABLE n SELECT \* FROM m;

# 新创建的表与一个存在的表的数据结构类似

CREATE TABLE m LIKE n;

# 创建一个临时表

# 临时表将在你连接MySQL期间存在。当断开连接时，MySQL将自动删除表并释放所用的空间。也可手动删除。

CREATE TEMPORARY TABLE l(id INT, name VARCHAR(10));

# 直接将查询结果导入或复制到新创建的临时表

CREATE TEMPORARY TABLE tt SELECT \* FROM n;

# 删除一个存在表

DROP TABLE IF EXISTS m;

# 更改存在表的名称

ALTER TABLE n RENAME m;

RENAME TABLE n TO m;

# 查看表的结构(以下五条语句效果相同）

DESC n; # 因为简单，所以建议使用（DESC表示descend降序，ASC表示ascend升序）

DESCRIBE n; #（discribe）

SHOW COLUMNS IN n;

SHOW COLUMNS FROM n;

EXPLAIN n;

# 查看表的创建语句

SHOW CREATE TABLE n;

表的结构

# 添加字段

ALTER TABLE n ADD age VARCHAR(2) ;

# 删除字段

ALTER TABLE n DROP age;

# 更改字段属性和属性

ALTER TABLE n CHANGE age a INT;

# 只更改字段属性

ALTER TABLE n MODIFY age VARCHAR(7) ;

表的数据

# 增加数据

INSERT INTO n VALUES (1, 'tom', '23'), (2, 'john', '22');

INSERT INTO n SELECT \* FROM n; # 把数据复制一遍重新插入

# 删除数据

DELETE FROM n WHERE id = 2;

# 更改数据

UPDATE n SET name = 'tom' WHERE id = 2;

# 数据查找

SELECT \* FROM n WHERE name LIKE '%h%';

# 数据排序(反序)

SELECT \* FROM n ORDER BY name, id DESC ;

键

# 添加主键

ALTER TABLE n ADD PRIMARY KEY (id);

ALTER TABLE n ADD CONSTRAINT pk\_n PRIMARY KEY (id); # 主键只有一个，所以定义键名似乎也没有什么用

# 删除主键

ALTER TABLE n DROP PRIMARY KEY ;

# 添加外键

ALTER TABLE m ADD FOREIGN KEY (id) REFERENCES n(id); # 自动生成键名m\_ibfk\_1

ALTER TABLE m ADD CONSTRAINT fk\_id FOREIGN KEY (id) REFERENCES n(id); # 使用定义的键名fk\_id

# 删除外键

ALTER TABLE m DROP FOREIGN KEY `fk\_id`;

# 修改外键

ALTER TABLE m DROP FOREIGN KEY `fk\_id`, ADD CONSTRAINT fk\_id2 FOREIGN KEY (id) REFERENCES n(id); # 删除之后从新建

# 添加唯一键

ALTER TABLE n ADD UNIQUE (name);

ALTER TABLE n ADD UNIQUE u\_name (name);

ALTER TABLE n ADD UNIQUE INDEX u\_name (name);

ALTER TABLE n ADD CONSTRAINT u\_name UNIQUE (name);

CREATE UNIQUE INDEX u\_name ON n(name);

# 添加索引

ALTER TABLE n ADD INDEX (age);

ALTER TABLE n ADD INDEX i\_age (age);

CREATE INDEX i\_age ON n(age);

# 删除索引或唯一键

DROP INDEX u\_name ON n;

DROP INDEX i\_age ON n;

视图

# 创建视图

CREATE VIEW v AS SELECT id, name FROM n;

CREATE VIEW v(id, name) AS SELECT id, name FROM n;

# 查看视图(与表操作类似)

SELECT \* FROM v;

DESC v;

# 查看创建视图语句

SHOW CREATE VIEW v;

# 更改视图

CREATE OR REPLACE VIEW v AS SELECT name, age FROM n;

ALTER VIEW v AS SELECT name FROM n ;

# 删除视图

DROP VIEW IF EXISTS v;

链接

# 内联接

SELECT \* FROM m INNER JOIN n ON m.id = n.id;

# 左外联接

SELECT \* FROM m LEFT JOIN n ON m.id = n.id;

# 右外联接

SELECT \* FROM m RIGHT JOIN n ON m.id = n.id;

# 交叉联接

SELECT \* FROM m CROSS JOIN n; # 标准写法

SELECT \* FROM m, n;

# 类似全连接full join的联接用法

SELECT id,name FROM m

UNION

SELECT id,name FROM n;

函数

# 聚合函数

SELECT count(id) AS total FROM n; # 总数

SELECT sum(age) AS all\_age FROM n; # 总和

SELECT avg(age) AS all\_age FROM n; # 平均值

SELECT max(age) AS all\_age FROM n; # 最大值

SELECT min(age) AS all\_age FROM n; # 最小值

# 数学函数

SELECT abs(-5); # 绝对值

SELECT bin(15), oct(15), hex(15); # 二进制，八进制，十六进制

SELECT pi(); # 圆周率3.141593

SELECT ceil(5.5); # 大于x的最小整数值6

SELECT floor(5.5); # 小于x的最大整数值5

SELECT greatest(3,1,4,1,5,9,2,6); # 返回集合中最大的值9

SELECT least(3,1,4,1,5,9,2,6); # 返回集合中最小的值1

SELECT mod(5,3); # 余数2

SELECT rand(); # 返回０到１内的随机值，每次不一样

SELECT rand(5); # 提供一个参数(种子)使RAND()随机数生成器生成一个指定的值。

SELECT round(1415.1415); # 四舍五入1415

SELECT round(1415.1415, 3); # 四舍五入三位数1415.142

SELECT round(1415.1415, -1); # 四舍五入整数位数1420

SELECT truncate(1415.1415, 3); # 截短为3位小数1415.141

SELECT truncate(1415.1415, -1); # 截短为-1位小数1410

SELECT sign(-5); # 符号的值负数-1

SELECT sign(5); # 符号的值正数1

SELECT sqrt(9); # 平方根3

SELECT sqrt(9); # 平方根3

# 字符串函数

SELECT concat('a', 'p', 'p', 'le'); # 连接字符串-apple

SELECT concat\_ws(',', 'a', 'p', 'p', 'le'); # 连接用','分割字符串-a,p,p,le

SELECT insert('chinese', 3, 2, 'IN'); # 将字符串'chinese'从3位置开始的2个字符替换为'IN'-chINese

SELECT left('chinese', 4); # 返回字符串'chinese'左边的4个字符-chin

SELECT right('chinese', 3); # 返回字符串'chinese'右边的3个字符-ese

SELECT substring('chinese', 3); # 返回字符串'chinese'第三个字符之后的子字符串-inese

SELECT substring('chinese', -3); # 返回字符串'chinese'倒数第三个字符之后的子字符串-ese

SELECT substring('chinese', 3, 2); # 返回字符串'chinese'第三个字符之后的两个字符-in

SELECT trim(' chinese '); # 切割字符串' chinese '两边的空字符-'chinese'

SELECT ltrim(' chinese '); # 切割字符串' chinese '两边的空字符-'chinese '

SELECT rtrim(' chinese '); # 切割字符串' chinese '两边的空字符-' chinese'

SELECT repeat('boy', 3); # 重复字符'boy'三次-'boyboyboy'

SELECT reverse('chinese'); # 反向排序-'esenihc'

SELECT length('chinese'); # 返回字符串的长度-7

SELECT upper('chINese'), lower('chINese'); # 大写小写 CHINESE chinese

SELECT ucase('chINese'), lcase('chINese'); # 大写小写 CHINESE chinese

SELECT position('i' IN 'chinese'); # 返回'i'在'chinese'的第一个位置-3

SELECT position('e' IN 'chinese'); # 返回'i'在'chinese'的第一个位置-5

SELECT strcmp('abc', 'abd'); # 比较字符串，第一个参数小于第二个返回负数- -1

SELECT strcmp('abc', 'abb'); # 比较字符串，第一个参数大于第二个返回正数- 1

# 时间函数

SELECT current\_date, current\_time, now(); # 2018-01-13 12:33:43 2018-01-13 12:33:43

SELECT hour(current\_time), minute(current\_time), second(current\_time); # 12 31 34

SELECT year(current\_date), month(current\_date), week(current\_date); # 2018 1 1

SELECT quarter(current\_date); # 1

SELECT monthname(current\_date), dayname(current\_date); # January Saturday

SELECT dayofweek(current\_date), dayofmonth(current\_date), dayofyear(current\_date); # 7 13 13

# 控制流函数

SELECT if(3>2, 't', 'f'), if(3<2, 't', 'f'); # t f

SELECT ifnull(NULL, 't'), ifnull(2, 't'); # t 2

SELECT isnull(1), isnull(1/0); # 0 1 是null返回1，不是null返回0

SELECT nullif('a', 'a'), nullif('a', 'b'); # null a 参数相同或成立返回null，不同或不成立则返回第一个参数

SELECT CASE 2

WHEN 1 THEN 'first'

WHEN 2 THEN 'second'

WHEN 3 THEN 'third'

ELSE 'other'

END ; # second

# 系统信息函数

SELECT database(); # 当前数据库名-test

SELECT connection\_id(); # 当前用户id-306

SELECT user(); # 当前用户-root@localhost

SELECT version(); # 当前mysql版本

SELECT found\_rows(); # 返回上次查询的检索行数

用户

# 增加用户

CREATE USER 'test'@'localhost' IDENTIFIED BY 'test';

INSERT INTO mysql.user(Host, User, Password) VALUES ('localhost', 'test', Password('test')); # 在用户表中插入用户信息，直接操作User表不推荐

# 删除用户

DROP USER 'test'@'localhost';

DELETE FROM mysql.user WHERE User='test' AND Host='localhost';

FLUSH PRIVILEGES ;

# 更改用户密码

SET PASSWORD FOR 'test'@'localhost' = PASSWORD('test');

UPDATE mysql.user SET Password=Password('t') WHERE User='test' AND Host='localhost';

FLUSH PRIVILEGES ;

# 用户授权

GRANT ALL PRIVILEGES ON \*.\* TO test@localhost IDENTIFIED BY 'test';

# 授予用'test'密码登陆成功的test@localhost用户操作所有数据库的所有表的所有的权限

FLUSH PRIVILEGES ; # 刷新系统权限表,使授予权限生效

# 撤销用户授权

REVOKE DELETE ON \*.\* FROM 'test'@'localhost'; # 取消该用户的删除权限

存储过程

# 创建存储过程

DELIMITER // # 无参数

CREATE PROCEDURE getDates()

BEGIN

SELECT \* FROM test ;

END //

CREATE PROCEDURE getDates\_2(IN id INT) # in参数

BEGIN

SELECT \* FROM test WHERE a = id;

END //

CREATE PROCEDURE getDates\_3(OUT sum INT) # out参数

BEGIN

SET sum = (SELECT count(\*) FROM test);

END //

CREATE PROCEDURE getDates\_4(INOUT i INT) # inout参数

BEGIN

SET i = i + 1;

END //

DELIMITER ;

# 删除存储过程

DROP PROCEDURE IF EXISTS getDates;

# 修改存储过程的特性

ALTER PROCEDURE getDates MODIFIES SQL DATA ;

# 修改存储过程语句（删除再重建）略

# 查看存储过程

SHOW PROCEDURE STATUS LIKE 'getDates'; # 状态

SHOW CREATE PROCEDURE getDates\_3; # 语句

# 调用存储过程

CALL getDates();

CALL getDates\_2(1);

CALL getDates\_3(@s);

SELECT @s;

SET @i = 1;

CALL getDates\_4(@i);

SELECT @i; # @i = 2

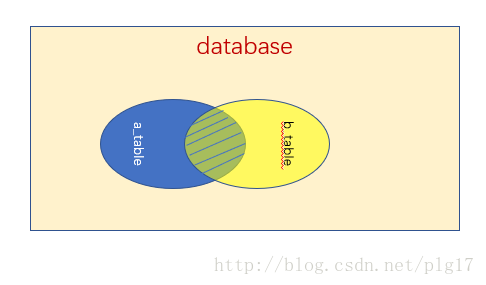
详情见：<https://blog.csdn.net/c361604199/article/details/79479398>。

### MySQL内链接，外链接（左链接、右链接、全链接）

内链接：（相交部分）

关键字：inner join on

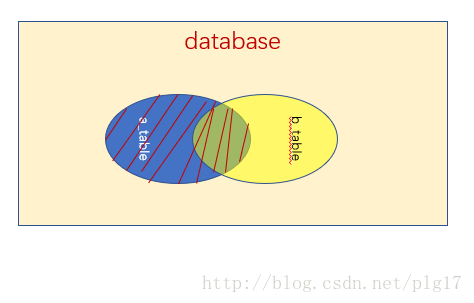
语句：select \* from a\_table a inner join b\_table b on a.a\_id = b.b\_id;



左链接：（左边部分）

关键字：left join on / left outer join on

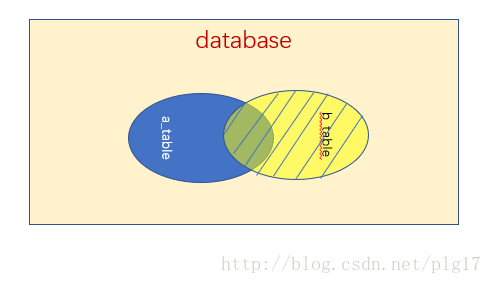
语句：select \* from a\_table a left join b\_table b on a.a\_id = b.b\_id;



右链接：（右边部分）

关键字：right join on/right outer join on

语句：select \* from a\_table a right outer join b\_table b on a.a\_id = b.b\_id;



全链接：目前MySQL不支持

详情见：<https://blog.csdn.net/plg17/article/details/78758593>。

### MySQL索引类型（逻辑角度）和原理

**MySQL目前主要有以下几种索引类型：普通索引、唯一索引、主键索引、组合索引、全文索引。**

普通索引：仅加速查询

唯一索引：加速查询 + 列值唯一（可以有null）

主键索引：加速查询 + 列值唯一（不可以有null）+ 表中只有一个

组合索引：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并

全文索引：对文本的内容进行分词，进行搜索

1：**普通索引**-是最基本的索引，它没有任何限制。它有以下几种创建方式：

直接创建索引

CREATE INDEX index\_name ON table(column(length))

修改表结构的方式添加索引

ALTER TABLE table\_name ADD INDEX index\_name ON (column(length))

创建表的时候同时创建索引

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) CHARACTER NOT NULL ,

`content` text CHARACTER NULL ,

`time` int(10) NULL DEFAULT NULL ,

PRIMARY KEY (`id`),

INDEX index\_name (title(length))

)

删除索引

DROP INDEX index\_name ON table

2：**唯一索引-**与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。它有以下几种创建方式：

创建唯一索引

CREATE UNIQUE INDEX indexName ON table(column(length))

修改表结构

ALTER TABLE table\_name ADD UNIQUE indexName ON (column(length))

创建表的时候直接指定

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) CHARACTER NOT NULL ,

`content` text CHARACTER NULL ,

`time` int(10) NULL DEFAULT NULL ,

UNIQUE indexName (title(length))

);

3：**主键索引**-是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。一般是在建表的时候同时创建主键索引：

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) NOT NULL ,

PRIMARY KEY (`id`)

);

4：**组合索引**-指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀集合

ALTER TABLE `table` ADD INDEX name\_city\_age (name,city,age);

5：**全文索引**-主要用来查找文本中的关键字，而不是直接与索引中的值相比较。fulltext索引跟其它索引大不相同，**它更像是一个搜索引擎**，而不是简单的where语句的参数匹配。fulltext索引配合match against操作使用，而不是一般的where语句加like。它可以在create table，alter table ，create index使用，不过目前只有char、varchar，text 列上可以创建全文索引。值得一提的是，在数据量较大时候，现将数据放入一个没有全局索引的表中，然后再用CREATE index创建fulltext索引，要比先为一张表建立fulltext然后再将数据写入的速度快很多。  
 创建表的适合添加全文索引

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) CHARACTER NOT NULL ,

`content` text CHARACTER NULL ,

`time` int(10) NULL DEFAULT NULL ,

PRIMARY KEY (`id`),

FULLTEXT (content)

);

修改表结构添加全文索引

ALTER TABLE article ADD FULLTEXT index\_content(content)

直接创建索引

CREATE FULLTEXT INDEX index\_content ON article(content)

**MySQL索引原理（底层实现角度）**

MySQL支持诸多存储引擎，而各种存储引擎对索引的支持也各不相同，因此MySQL数据库支持多种索引类型，如BTree索引，B+Tree索引，哈希索引，全文索引等等，

**1：哈希索引**

**只有memory（内存）存储引擎支持哈希索引**，哈希索引用索引列的值计算该值的hashCode，然后在hashCode相应的位置存执该值所在行数据的物理位置，因为使用散列算法，因此访问速度非常快，但是一个值只能对应一个hashCode，而且是散列的分布方式，因此哈希索引不支持范围查找和排序的功能。

2：全文索引

**FULLTEXT（全文）索引，仅可用于MyISAM和InnoDB，针对较大的数据，生成全文索引非常的消耗时间和空间。**对于文本的大对象，或者较大的CHAR类型的数据，如果使用普通索引，那么匹配文本前几个字符还是可行的，但是想要匹配文本中间的几个单词，那么就要使用LIKE %word%来匹配，这样需要很长的时间来处理，响应时间会大大增加，这种情况，就可使用时FULLTEXT索引了，在生成FULLTEXT索引时，会为文本生成一份单词的清单，在索引时及根据这个单词的清单来索引。FULLTEXT可以在创建表的时候创建，也可以在需要的时候用ALTER或者CREATE INDEX来添加：

//创建表的时候添加FULLTEXT索引

CTREATE TABLE my\_table(

id INT(10) PRIMARY KEY,

name VARCHAR(10) NOT NULL,

my\_text text CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL,

FULLTEXT(my\_text));

//创建表以后，在需要的时候添加FULLTEXT索引

ALTER my\_table ADD FULLTEXT ft\_index(my\_text);

CREATE INDEX ft\_index ON my\_table(my\_text);

\*对于较大的数据集，把数据添加到一个没有FULLTEXT索引的表，然后添加FULLTEXT索引的速度比把数据添加到一个已经有FULLTEXT索引的表快。

\*MySQL自带的全文索引只能用于MyISAM存储引擎，如果是其它数据引擎，那么全文索引不会生效。

\*在MySQL中，全文索引支队英文有用，目前对中文还不支持。

\*在MySQL中，如果检索的字符串太短则无法检索得到预期的结果，检索的字符串长度至少为4字节，此外，如果检索的字符包括停止词，那么停止词会被忽略。

3：BTree索引和B+Tree索引

BTree索引

BTree是平衡搜索多叉树，设树的度为d（d>1），高度为h，那么BTree要满足以一下条件：

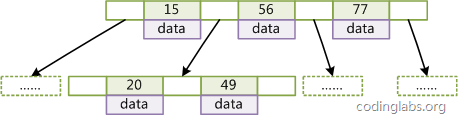
每个叶子结点的高度一样，等于h；

每个非叶子结点由n-1个key和n个指针point组成，其中d<=n<=2d,key和point相互间隔，结点两端一定是key；

叶子结点指针都为null；

非叶子结点的key都是[key,data]二元组，其中key表示作为索引的键，data为键值所在行的数据；

BTree的结构如下：



在BTree的机构下，就可以使用二分查找的查找方式，查找复杂度为h\*log(n)，一般来说树的高度是很小的，一般为3左右，因此BTree是一个非常高效的查找结构。

B+Tree索引

B+Tree是BTree的一个变种，设d为树的度数，h为树的高度，B+Tree和BTree的不同主要在于：

B+Tree中的非叶子结点不存储数据，只存储键值；

B+Tree的叶子结点没有指针，所有键值都会出现在叶子结点上，且key存储的键值对应的数据的物理地址；

B+Tree的结构如下：

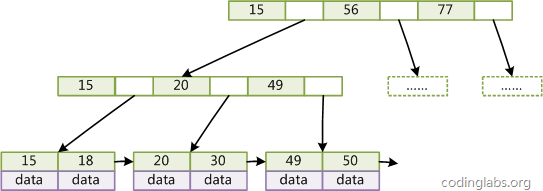


一般来说B+Tree比BTree更适合实现外存的索引结构，因为存储引擎的设计专家巧妙的利用了外存（磁盘）的存储结构，即磁盘的一个扇区是整数倍的page（页），页是存储中的一个单位，通常默认为4K，因此索引结构的节点被设计为一个页的大小，然后利用外存的“预读取”原则，每次读取的时候，把整个节点的数据读取到内存中，然后在内存中查找，已知内存的读取速度是外存读取I/O速度的几百倍，那么提升查找速度的关键就在于尽可能少的磁盘I/O，那么可以知道，每个节点中的key个数越多，那么树的高度越小，需要I/O的次数越少，因此一般来说B+Tree比BTree更快，因为B+Tree的非叶节点中不存储data，就可以存储更多的key。

带顺序索引的B+TREE

很多存储引擎在B+Tree的基础上进行了优化，添加了指向相邻叶节点的指针，形成了带有顺序访问指针的B+Tree，这样做是为了提高区间查找的效率，只要找到第一个值那么就可以顺序的查找后面的值。

B+Tree的结构如下：



详情见：<https://www.cnblogs.com/luyucheng/p/6289714.html>。

<https://blog.csdn.net/tongdanping/article/details/79878302>。

### MySQL数据存储引擎

**简单来说，存储引擎就是指表的类型以及表在计算机上的存储方式。**

存储引擎的概念是MySQL的特点，Oracle中没有专门的存储引擎的概念，Oracle有OLTP和OLAP模式的区分。不同的存储引擎决定了MySQL数据库中的表可以用不同的方式来存储。我们可以根据数据的特点来选择不同的存储引擎。

在MySQL中的存储引擎有很多种，可以通过“SHOW ENGINES”语句来查看。下面重点关注InnoDB、MyISAM、MEMORY这三种。

1：InnoDB存储引擎

InnoDB给MySQL的表提供了事务处理、回滚、崩溃修复能力和多版本并发控制的事务安全。在MySQL从3.23.34a开始包含InnnoDB。它是MySQL上第一个提供外键约束的表引擎。而且InnoDB对事务处理的能力，也是其他存储引擎不能比拟的。靠后版本的MySQL的默认存储引擎就是InnoDB。

InnoDB存储引擎总支持AUTO\_INCREMENT。自动增长列的值不能为空，并且值必须唯一。MySQL中规定自增列必须为主键。在插入值的时候，如果自动增长列不输入值，则插入的值为自动增长后的值；如果输入的值为0或空（NULL），则插入的值也是自动增长后的值；如果插入某个确定的值，且该值在前面没有出现过，就可以直接插入。

InnoDB还支持外键（FOREIGN KEY）。外键所在的表叫做子表，外键所依赖（REFERENCES）的表叫做父表。父表中被字表外键关联的字段必须为主键。当删除、更新父表中的某条信息时，子表也必须有相应的改变，这是数据库的参照完整性规则。

InnoDB中，创建的表的表结构存储在.frm文件中（我觉得是frame的缩写吧）。数据和索引存储在innodb\_data\_home\_dir和innodb\_data\_file\_path定义的表空间中。

InnoDB的优势在于提供了良好的事务处理、崩溃修复能力和并发控制。缺点是读写效率较差，占用的数据空间相对较大。

2：MyISAM存储引擎

MyISAM是MySQL中常见的存储引擎，曾经是MySQL的默认存储引擎。MyISAM是基于ISAM引擎发展起来的，增加了许多有用的扩展。

MyISAM的表存储成3个文件。文件的名字与表名相同。拓展名为frm、MYD、MYI。其实，frm文件存储表的结构；MYD文件存储数据，是MYData的缩写；MYI文件存储索引，是MYIndex的缩写。

基于MyISAM存储引擎的表支持3种不同的存储格式。包括静态型、动态型和压缩型。其中，静态型是MyISAM的默认存储格式，它的字段是固定长度的；动态型包含变长字段，记录的长度不是固定的；压缩型需要用到myisampack工具，占用的磁盘空间较小。

MyISAM的优势在于占用空间小，处理速度快。缺点是不支持事务的完整性和并发性。

3：MEMORY存储引擎

memory存储引擎是MySQL中的一类特殊的存储引擎。其使用存储在内存中的内容来创建表，而且所有数据也放在内存中。这些特性都与InnoDB，MyISAM存储引擎不同。

OK，这里我们讲解一些memory存储引擎的文件存储形式，索引类型，存储周期和优缺点。

每个基于memory存储引擎的表实际对应一个磁盘文件，该文件的文件名与表名相同，类型为frm类型。该文件只存储表的结构，而其数据文件，都是存储在内存中的，这样有利于对数据的快速的处理，提高整个表的处理效率。

值得注意的是：服务器需要有足够的内存来维持memory存储引擎的表的使用。如果不需要了，可以释放这些内存，甚至可以删除不需要的表。

Memory存储引擎默认使用哈希(HASH)索引，其速度比使用B型树(BTREE)索引快。如果我们需要使用B型树索引，可以在创建索引时选择使用。

这里来整理一个小的技巧：

Memory存储引擎通常很少用到，至少我是没有用到过。因为Memory表的所有数据都是存储在内存上的，如果内存出现异常会影响到数据的完整性。

如果重启机器或者关机，表中的所有数据都将消失，因此，基于Memory存储引擎的表的生命周期都比较短，一般都是一次性的。

Memory表的大小是受到限制的，表的大小主要取决于2个参数，分别是max\_rows和max\_heap\_table\_size。其中，max\_rows可以在创建表时指定，max\_heap\_table\_size的大小默认为16MB，可以按需要进行扩大。

因此，其基于内存中的特性，这类表的处理速度会非常快，但是，其数据易丢失，生命周期短。基于其这个缺陷，选择Memory存储引擎时需要特别小心。

4：怎样选择存储引擎

在实际工作中，选择一个合适的存储引擎是一个比较复杂的问题。每种存储引擎都有自己的优缺点，不能笼统地说谁比谁好



InnoDB：支持事务处理，支持外键，支持崩溃修复能力和并发控制。如果需要对事务的完整性要求比较高（比如银行），要求实现并发控制（比如售票），那选择InnoDB有很大的优势。如果需要频繁的更新、删除操作的数据库，也可以选择InnoDB，因为支持事务的提交（commit）和回滚（rollback），还有需要主键自增或者外键的需求也需要InnoDB。

MyISAM：插入数据快，空间和内存使用比较低。如果表主要是用于插入新记录和读出记录，那么选择MyISAM能实现处理高效率。如果应用的完整性、并发性要求比 较低，也可以使用。

MEMORY：所有的数据都在内存中，数据的处理速度快，但是安全性不高。如果需要很快的读写速度，对数据的安全性要求较低，可以选择MEMOEY。它对表的大小有要求，不能建立太大的表。所以，这类数据库只使用在相对较小的数据库表。

注意，同一个数据库也可以使用多种存储引擎的表。如果一个表要求比较高的事务处理，可以选择InnoDB。这个数据库中可以将查询要求比较高的表选择MyISAM存储。如果该数据库需要一个用于查询的临时表，可以选择MEMORY存储引擎。

5：InnoDB 索引到底是b+树还是b树 :

先从数据结构的角度来答。

题主应该知道B-树和B+树最重要的一个区别就是**B+树只有叶节点存放数据，其余节点用来索引，而B树是每个索引节点都会有Data域**。

这就决定了**B+树更适合用来存储外部数据，也就是所谓的磁盘数据。**

**从Mysql（Inoodb）的角度来看，B+树是用来充当索引的，一般来说索引非常大，尤其是关系性数据库这种数据量大的索引能达到亿级别，所以为了减少内存的占用，索引也会被存储在磁盘上。**

那么Mysql如何衡量查询效率呢？磁盘IO次数，B-树（B类树）的特定就是每层节点数目非常多，层数很少，目的就是为了就少磁盘IO次数，当查询数据的时候，最好的情况就是很快找到目标索引，然后读取数据，使用B+树就能很好的完成这个目的，但是B-树的每个节点都有data域（指针），这无疑增大了节点大小，说白了增加了磁盘IO次数（磁盘IO一次读出的数据量大小是固定的，单个数据变大，每次读出的就少，IO次数增多，一次IO多耗时啊！），而B+树除了叶子节点其它节点并不存储数据，节点小，磁盘IO次数就少。这是优点之一。

另一个优点是什么，B+树所有的Data域在叶子节点，一般来说都会进行一个优化，就是将所有的叶子节点用指针串起来。这样遍历叶子节点就能获得全部数据，这样就能进行区间访问啦。

至于MongoDB为什么使用B树而不是B+树，可以从它的设计角度来考虑，它并不是传统的关系性数据库，而是以Json格式作为存储的nosql，目的就是高性能，高可用，易扩展。首先它摆脱了关系模型，上面所述的优点2需求就没那么强烈了，其次Mysql由于使用B+树，数据都在叶节点上，每次查询都需要访问到叶节点，而MongoDB使用B-树，所有节点都有Data域，只要找到指定索引就可以进行访问，无疑单次查询平均快于Mysql（但侧面来看Mysql至少平均查询耗时差不多）。

总体来说，Mysql选用B+树和MongoDB选用B-树还是以自己的需求来选择的。

详情见：<https://blog.csdn.net/yifansj/article/details/79233726>。

### Sql优化思路

**SQL语句优化：**

1：**SQL语句全部大写**（所有SQL语句执行时都会转换成大写）

2：**避免使用星号 \* ,用具体字段替代**（使用星号会在查询时增加一个查询列的操作）

3：**尽量避免在 where 子句中对字段进行 null 值判断，最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库**（不然会进行全表扫描，影响效率）

-反例：select id from t where num is null

-可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：select id from t where num = 0

4：应尽量避免在 where 子句中使用 != 或 <> ，in 或 not in 操作符 （同上）

5：**能用between就不要使用in，或者用exists替换in**

-select num from a where num in(select num from b)

用下面的语句替换：select num from a where exists(select 1 from b where num=a.num)

6：**尽可能的使用 varchar/nvarchar（长度可变） 代替 char/nchar（长度不可变） （节省字段存储空间）**

**Mysql分页优化**

Mysql在使用limit进行分页查询时有明显的缺陷，比如：

SELECT \* FROM table ORDER BY id LIMIT x,y;

当x越来越大时，因为会查询出x跳记录然后过滤出需要的y条，查询效率会显著降低，为此，有几下集中解决方案：

SELECT \* FROM table WHERE id>= (SELECT id FROM table LIMIT1000000, 1) LIMIT 10`

如果id是连续的，建议使用between

SELECT \* FROM table WHERE id BETWEEN 1000000 AND 1000010;

还有一种，可以在查询下一页时把上一页的行id作为参数传递给客户端程序，然后sql就改成了

select \* from table where id>3000000 limit 10;

详情见： <https://blog.csdn.net/m0_37896272/article/details/78177484>。

### 数据库高并发解决方法总结

一个项目刚开始的时候是为了实现基本功能，随着版本和功能的迭代，[大数据](http://lib.csdn.net/base/hadoop)和高并发成了软件设计必须考虑的问题！本质很简单，一个是慢，一个是等。两者是相互关联的，因为慢，所以要等，因为等，所以慢，解决了慢，也就解决了等，解决了等，也就解决了慢。

关键是如何解决慢和等，核心一个是短，一个是少，一个是分流，最后一个是集群/横向扩张/读写分离/建立主从。

**短是指路径要短：**

典型的mvc结构是请求->controller->model->dao->view，然后把页面返回给用户。要想短的话，

1：页面静态化- 用户可以直接获取页面，不用走那么多流程，比较适用于页面不频繁更新。

2：使用缓存- 第一次获取数据从[数据库](http://lib.csdn.net/base/mysql)准提取，然后保存在缓存中，以后就可以直接从缓存提取数据。不过需要有机制维持缓存和数据库的一致性。

3：使用储存过程-那些处理一次请求需要多次访问数据库的操作，可以把操作整合到储存过程，这样只要一次数据库访问就可以了。

4：批量读取 - 高并发情况下，可以把多个请求的查询合并到一次进行，以减少数据库的访问次数

5：延迟修改 - 高并发情况下，可以把多次修改请求，先保存在缓存中，然后定时将缓存中的数据保存到数据库中，风险是可能会断电丢失缓存中的数据，

6：使用索引 - 索引可以看作是特殊的缓存，尽量使用索引就要求where字句中精确的给出索引列的值。

**少是指查询的数据要少**

1：分表 - 把本来同一张表的内容，可以按照地区，类别等分成多张表，很简单的一个思路，但是要尽量避免分出来的多表关联查询。

2：分离活跃数据 - 例如登录用户业务，注册用户很多，但是活跃的登录用户很少，可以把活跃用户专门保存一张表，查询是先查询活跃表，没有的话再查总表，这也类似与缓存啦。

3：分块 - 数据库层面的优化，对程序是透明的，查询大数据只用找到相应块就行。

**分流三种:**

1：集群 - 将并发请求分配到不同的服务器上，可以是业务服务器，也可以是数据库服务器。

2：分布式 - 分布式是把单次请求的多项业务逻辑分配到多个服务器上，这样可以同步处理很多逻辑，一般使用与特别复杂的业务请求。

3：CDN - 在域名解析层面的分流，例如将华南地区的用户请求分配到华南的服务器，华中地区的用户请求分配到华中的服务器。

详情见：<https://www.cnblogs.com/zxz1987/p/6538462.html>。

## 网络编程

### 网络编程API有哪些

一、大端字节序和小端字节序

      计算机硬件有两种储存数据的方式：大端字节序（big endian）和小端字节序（little endian）。举例来说，数值0x2211使用两个字节储存：高位字节是0x22，低位字节是0x11。

**大端字节序：高位字节排放在内存的低地址端，低位字节排放在内存的高地址端**。又被称为网络字节序，方便程序员检查。

**小端字节序：低位字节排放在内存的低地址端，高位字节排放在内存的高地址端**。现代PC大多采用小端字节序，所以其又被称为主机字节序。

二、核心API

1：创建socket

#include<sys/typs.h>

#include<sys/socket.h>

int socket(int domain, int type, int protocol)

domain:告诉系统使用哪个低层协议族；

type:指服务类型（面向流的还是字节的）；

protocol:默认值为0。

socket创建成功时返回一个socket文件描述符，失败则返回-1并设置error。

2：命名socket

      将一个socket与socket绑定称为给socket命名。

int bind(int sockfd, const struct sockaddr\* my\_addr, socklen\_t addrlen)

bind将my\_addr所指的socket地址分配给未命名的sockfd文件描述符，addrlen是socket地址的长度。

      bind成功则返回0，失败则返回-1并设置error。

3：监听socket

      首先需要系统调用来创建一个监听队列以存放待处理的客户连接：

#include<sys/socket.h>

int listen(int sockfd, intbacklog)

sockfd参数指定被监听的socket。backlog参数提示内核监听队列的最大长度。

listen成功则返回0，失败则返回-1并设置error。

4：接受连接

#include<sys/typs.h>

#include<sys/socket.h>

int accept(int sockfd, const sockaddr\*addr，socklen\_t \*addrlen);

sockfd参数是执行过listen系统调用的监听socket。addr参数用来获取被接受连接的远端socket地址，该socket地址的长度由addrlen参数指出。Accept成功时返回一个新的连接socket，该socket唯一标识了被接受的这个连接，服务器可通过读写该socket来与被接受连接对应的客户通信。失败则返回-1并设置error。

5：发起连接

服务器通过listen调用被动接受连接，那么客户端需要通过connect的系统调用来主动与服务器建立连接：

#include<sys/typs.h>

#include<sys/socket.h>

int connect(int sockfd, const struct sockaddr\*serv\_addr，socklen\_t addrlen);

 sockfd 参数由socket 系统调用返回一个socket。serv\_addr参数是服务器监听的socket地址，addrlen参数则指定这个地址的长度。

 Connect成功时返回0。一旦成功建立连接，sockfd就唯一地标识了这个连接，客户端就可以通过读写socket来与服务器通信。

6：关闭连接

关闭一个连接就是关闭该连接对应的socket，可以通过关闭普通文件描述符的系统调用来完成：

#include<unistd.h>

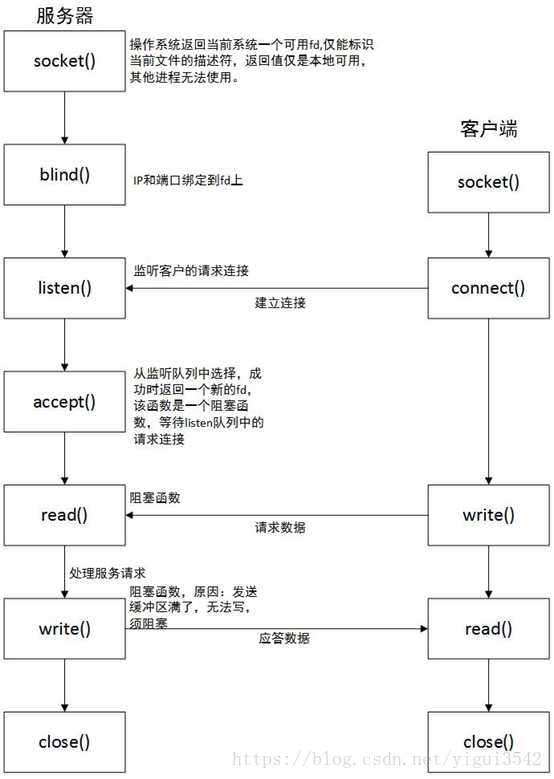
int close(int fd)

7：数据读写

Recv读取sockfd 上的数据，buf、len参数分别指定读缓冲区的位置和大小，flags通常置为0。rece成功时返回实际读取到的数据长度。recv可能返回0，这意味着通信对方已经关闭连接了。失败则返回-1并设置error。

send往sockfd上写数据，buf、len参数分别指定读缓冲区的位置和大小。send成功时返回实际读取到的数据长度，失败则返回-1并设置error。

8：整体流程图



总结：

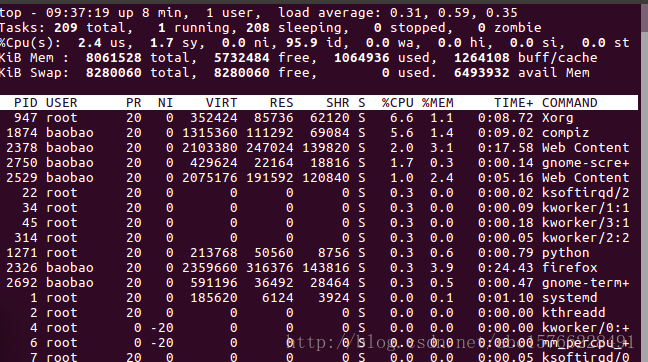
**创建套接字、命名套接字、监听套接字、接受连接、发起连接、关闭链接、数据读写**

详情见：<https://blog.csdn.net/yigui3542/article/details/80755970>。

## Linux

### top命令

终端输入top之后，就是下面这样啦



前面是参数，后面就是进程和进程号之类的了

下面，我们一行一行看

第一行：当前时间，系统运行时间，登录用户数量，平均负载（分别在5，10，15分钟内）

这里是：早上9：37：19，系统运行了8分钟，1一个用户

第二行：显示了系统的进程总数，后面是相应的状态下的进程

这里是：一共209个进程，1个是running状态，208个sleeping状态，0个stopped，0个zombie

关于进程的状态，这里解释一下zombie：这个是僵尸进程，就是，这个进程其实已经结束了，它仅仅在进程列表中保留一个位置，记载该进程的状态信息等，僵尸进程不再占有内存空间，没有可执行程序，也不能被调用。。这个进程中存储着进程的各种信息，占用cpu啊，运行时间之类的。。。这个进程会被其父进程收集它的信息。。。

第三行：就是cpu的各种信息了

参数说明如下：

us：用户空间占cpu百分比

sy：内核空间占cpu百分比

ni：用户进程空间内改变过优先级的进程占用cpu百分比

id：空闲cpu百分比

wa：等待输入输出的cpu时间百分比

hi：硬中断（处理硬件中断的cpu时间）

si：软中断（处理软件中断的cpu时间）

第四行、第五行：内存使用

第一行：物理内存的使用，第二行：虚拟内存（交换空间）的使用。   
每一行的后面四个参数是：总的内存，已经使用的内存，空闲内存，缓冲内存

第六行：表头，具体解释如下：

PID： 进程ID进程的唯一标识符

USER：进程ID 进程的唯一标识符

PR：进程调度优先级，一个拥有更高进程优先级的进程拥有更大的机率得到处理器的处理。，”tr”值代表这些进程运行在实时态

NI：进程的nice值（优先值）。越小意味着越高的优先级。

VIRT：系统使用的虚拟内存

RES：驻留内存大小，驻留内存是任务使用的非交换物理内存大小

SHR：是进程使用的共享内存

S：进程状态：

D：不可中断的睡眠态

R：运行态

S：睡眠态

T：被跟踪或已停止

Z：僵尸态

%CPU 自从上一次更新时到现在任务所使用的CPU时间百分比。

%MEM 进程使用的可用物理内存百分比

TIME+ 任务启动后到现在所使用的全部CPU时间，精确到百分之一秒。

COMMAND 进程所使用的命令。

详情见：<https://blog.csdn.net/abc15766228491/article/details/79339208>。

### ps命令

ps命令是最基本的进程查看命令，使用该命令可以确定有哪些进程正在运行和运行的状态、进程是否结束、进程有没有僵尸、哪些进程占用了过多的资源等等。**ps是显示瞬间进程的状态，并不动态连续；如果想对进程进行实时监控应该用top命令。**

参数：

* -A ：所有的进程均显示出来，与 -e 具有同样的效用；
* -a ： 显示现行终端机下的所有进程，包括其他用户的进程；
* -u ：以用户为主的进程状态 ；
* x ：通常与 a 这个参数一起使用，可列出较完整信息。

输出格式规划：

* l ：较长、较详细的将该PID 的的信息列出；
* j ：工作的格式 (jobs format)
* -f ：做一个更为完整的输出

详情见：<https://www.cnblogs.com/wxgblogs/p/6591980.html>。

### netstat命令

**netstat命令用于显示与IP、TCP、UDP和ICMP协议相关的统计数据，一般用于检验本机各端口的网络连接情况。**netstat是在内核中访问网络及相关信息的程序，它能提供TCP连接，TCP和UDP监听，进程内存管理的相关报告。

例子

列出所有端口情况

[root@xiesshavip002 ~]# netstat -a # 列出所有端口

[root@xiesshavip002 ~]# netstat -at # 列出所有TCP端口

[root@xiesshavip002 ~]# netstat -au # 列出所有UDP端口

列出所有处于监听状态的 Sockets

[root@xiesshavip002 ~]# netstat -l # 只显示监听端口

[root@xiesshavip002 ~]# netstat -lt # 显示监听TCP端口

[root@xiesshavip002 ~]# netstat -lu # 显示监听UDP端口

[root@xiesshavip002 ~]# netstat -lx # 显示监听UNIX端口

显示每个协议的统计信息

[root@xiesshavip002 ~]# netstat -s # 显示所有端口的统计信息

[root@xiesshavip002 ~]# netstat -st # 显示所有TCP的统计信息

[root@xiesshavip002 ~]# netstat -su # 显示所有UDP的统计信息

显示 PID 和进程名称

[root@xiesshavip002 ~]# netstat -p

显示核心路由信息

[root@xiesshavip002 ~]# netstat -r

Kernel IP routing table

Destination Gateway Genmask Flags MSS Window irtt Iface

default gateway 0.0.0.0 UG 0 0 0 eth0

192.168.130.0 0.0.0.0 255.255.255.0 U 0 0 0 eth0

[root@xiesshavip002 ~]# netstat -rn # 显示数字格式，不查询主机名称

Kernel IP routing table

Destination Gateway Genmask Flags MSS Window irtt Iface

0.0.0.0 192.168.130.1 0.0.0.0 UG 0 0 0 eth0

192.168.130.0 0.0.0.0 255.255.255.0 U 0 0 0 eth0

详情见：<https://www.cnblogs.com/ftl1012/p/netstat.html>。

<https://www.cnblogs.com/xieshengsen/p/6618993.html>。

### awk命令

awk是一个强大的文本分析工具。

相对于grep的查找，sed的编辑，awk在其对数据分析并生成报告时，显得尤为强大。**简单来说awk就是把文件逐行的读入，以空格为默认分隔符将每行切片，切开的部分再进行各种分析处理。**

使用方法：awk ‘{patten + action}’ {filename}

尽管操作可能会很复杂，但语法总是这样，其中 pattern 表示 AWK 在数据中查找的内容，而 action 是在找到匹配内容时所执行的一系列命令。花括号（{}）不需要在程序中始终出现，但它们用于根据特定的模式对一系列指令进行分组。 pattern就是要表示的正则表达式，用斜杠括起来。

awk语言的最基本功能是在文件或者字符串中基于指定规则浏览和抽取信息，awk抽取信息后，才能进行其他文本操作。完整的awk脚本通常用来格式化文本文件中的信息。

通常，awk是以文件的一行为处理单位的。awk每接收文件的一行，然后执行相应的命令，来处理文本。

详细操作介绍见：<https://www.cnblogs.com/sahara/p/6874918.html>。

详情见：<https://www.cnblogs.com/xiaoleiel/p/8349487.html>。

### find命令

查找文件  
find ./ -type f  
  
查找目录  
find ./ -type d  
  
查找名字为test的文件或目录  
find ./ -name test  
  
查找名字符合正则表达式的文件,注意前面的‘.\*’(查找到的文件带有目录)  
find ./ -regex .\*so.\*\.gz  
  
查找文件名匹配\*.c的文件  
find ./ -name \*.c  
  
查找文件更新日时在距现在时刻二天以内的文件  
find ./ -mtime -2  
  
查找文件更新日时在距现在时刻二天以上的文件  
find ./ -mtime +2  
  
查找文件更新日时在距现在时刻一天以上二天以内的文件  
find ./ -mtime 2  
  
查找文件更新日时在距现在时刻二分以内的文件  
find ./ -mmin -2  
  
查找文件更新日时在距现在时刻二分以上的文件  
find ./ -mmin +2  
  
查找文件更新日时在距现在时刻一分以上二分以内的文件  
find ./ -mmin 2  
  
查找空文件或空目录  
find ./ -empty  
  
查找权限为644的文件或目录(需完全符合)  
find ./ -perm 664  
  
查找用户/组权限为读写，其他用户权限为读(其他权限不限)的文件或目录  
find ./ -perm -664  
  
查找用户有写权限或者组用户有写权限的文件或目录  
find ./ -perm /220  
find ./ -perm /u+w,g+w  
find ./ -perm /u=w,g=w  
  
查找所有者权限有读权限的目录或文件  
find ./ -perm -u=r  
  
查找用户组权限有读权限的目录或文件  
find ./ -perm -g=r  
  
查找其它用户权限有读权限的目录或文件  
find ./ -perm -o=r  
  
查找所有者为lzj的文件或目录  
find ./ -user lzj  
  
查找组名为gname的文件或目录  
find ./ -group gname  
  
查找文件的用户ID不存在的文件  
find ./ -nouser  
  
查找文件的组ID不存在的文件  
find ./ -nogroup  
  
查找文件size小于10个字节的文件或目录  
find ./ -size -10c  
  
查找文件size等于10个字节的文件或目录  
find ./ -size 10c  
  
查找文件size大于10个字节的文件或目录  
find ./ -size +10c  
  
查找文件size小于10k的文件或目录  
find ./ -size -10k  
  
查找文件size小于10M的文件或目录  
find ./ -size -10M  
  
查找文件size小于10G的文件或目录  
find ./ -size -10G

详细用法见：<https://www.cnblogs.com/bigbean/p/3669739.html>。

### grep命令

Linux系统中grep命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。

使用格式：grep [OPTIONS] PATTERN [FILE...]

例如：grep -i "s" /etc/passwd

grep

常用参数：

-c： 打印符合要求的行数（数目）

-i ：忽略大小写

-n：输出行和行号

-v：打印不符合要求的行，即反选

-A：后跟数字(有无空格都可以)，例如-A2 表示打印筛选行及前2行

-B：后跟数字，例如-B2表示打印筛选行及后2行

-C：后跟数字，例如-C2表示打印筛选行及前后各2行

-o：只打印符合要求的内容，而非整行

详情见<http://www.cnblogs.com/csj2018/p/9158963.html>。

### wc命令

wc命令的功能为统计指定文件中的字节数、字数、行数, 并将统计结果显示输出。

语法：wc [选项] 文件…

说明：该命令统计给定文件中的字节数、字数、行数。如果没有给出文件名，则从标准输入读取。wc同时也给出所有指定文件的总统计数。字是由空格字符区分开的最大字符串。

该命令各选项含义如下：

- c 统计字节数。

- l 统计行数。

- w 统计字数。

这些选项可以组合使用。

输出列的顺序和数目不受选项的顺序和数目的影响。总是按下述顺序显示并且每项最多一列。

行数、字数、字节数、文件名

如果命令行中没有文件名，则输出中不出现文件名。

统计指定文件中的字节数、字数、行数，并将统计结果显示输出。

详情见：<https://blog.csdn.net/freeking101/article/details/53404897>。

### sed命令

sed是一个很好的文件处理工具，本身是一个管道命令，主要是以行为单位进行处理，可以将数据行进行替换、删除、新增、选取等特定工作，下面先了解一下sed的用法

sed命令行格式为：

sed [-nefri] ‘command’ 输入文本

常用选项：

-n∶使用安静(silent)模式。在一般 sed 的用法中，所有来自 STDIN的资料一般都会被列出到萤幕上。但如果加上 -n 参数后，则只有经过sed 特殊处理的那一行(或者动作)才会被列出来。

-e：直接在指令列模式上进行 sed 的动作编辑；

-f：直接将 sed 的动作写在一个档案内， -f filename 则可以执行 filename 内的sed 动作；

-r：sed 的动作支援的是延伸型正规表示法的语法。(预设是基础正规表示法语法)

-i：直接修改读取的档案内容，而不是由萤幕输出。

常用命令：

a：新增，a 的后面可以接字串，而这些字串会在新的一行出现(目前的下一行)～

c：取代，c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！

d：删除，因为是删除啊，所以 d 后面通常不接任何咚咚；

i：插入， i 的后面可以接字串，而这些字串会在新的一行出现(目前的上一行)；

p：列印，亦即将某个选择的资料印出。通常 p 会与参数 sed -n 一起运作～

s：取代，可以直接进行取代的工作哩！通常这个 s 的动作可以搭配正规表示法！例如 sed ‘1,20s/old/new/g’ 就是啦！

详情见：<https://www.cnblogs.com/ginvip/p/6376049.html>。

### head和tail命令

head [option] …filename…

选项：-n num 显示文件的前num行。

-c num 显示文件的前num个字节。

-c -n 显示文件除了最后n个字节的其他内容。

-q 隐藏文件名。

-v 显示文件名。

tail [option] …filename…

选项：-n num 显示文件的后num行。

-r num 逆序显示filename最后10行。

-f 检视filename的尾部内容（相当于-n 10）。

### 正则表达式

**基本正则表达式：Basic REGEXP**

|  |  |
| --- | --- |
| 元字符 | 描述 |
| ？ | 匹配任意单个字符 |
| \* | 匹配任意字符 |
| [] | 匹配指定**范围内**的任意单个字符 |
| [^] | 匹配指定**范围外**的任意单个字符 |
| [:lower:] | 小写字母 |
| [:upper:] | 大写字母 |
| [:alpha:] | 所有字母 |
| [:digit:] | 数字 |
| [:alnum:] | 所有数字和字母 |
| [:punct:] | 标点符号 |
| [:space:] | 空白字符 |
| \? | 匹配其前面的字符1次或0次 |
| \{m,n\} | 匹配其前面的字符至少m次，至多n次 |
| ^ | 铆定行首，此字符后面的任意内容必须出现在行首 |
| $ | 铆定行尾，此字符前面的任意内容必须出现在行尾 |
| ^$ | 表示空白行 |
| \<或\b | 铆定词首，其后面的任意字符必须作为单词的首部出现 |
| \>或\b | 铆定词尾，其前面的任意字符必须作为单词的尾部出现 |
| \(\) | 分组 |
| \(ab\)\* | ab作为一个整体，可以出现任意次 |
| \(ab\).\*\1 | 引用第一个左括号以及与之对应的右括号所包括的所有内容 |
| \(ab\).\*\2 | 引用第二个左括号以及与之对应的右括号所包括的所有内容 |

**扩展正则表达式：Extended REGEXP**

|  |  |
| --- | --- |
| 字符匹配 | |
| . | 匹配任意单个字符 |
| [] | 匹配指定范围内的任意单个字符 |
| [^] | 匹配指定范围外的任意单个字符 |
| 次数匹配 | |
| \* | 匹配其前字符任意次 |
| ? | 匹配其前字符0次或1次 |
| + | 匹配其前字符至少1次，类似于基本正则表达式\{1,\} |
| {m,n} | 匹配其前面的字符至少m次，至多n次 |
| 位置铆定 | |
| ^ | 行首 |
| $ | 行尾 |
| \<或\b | 词首 |
| \>或\b | 词尾 |
| 分组 | |
| ().\*\1\2\3 |  |
| 或者 | |
| | | or  a|b ，a或者b ，有一个就行 |
| C|cat--> C或cat  (C|c)at-->Cat或cat |

### 如何查找出现频率最高的100个IP地址

查看日志中访问次数最多的前10个IP

cat access\_log | cut –d ‘ ’ –f 1 | sort | uniq –c | sort –nr | awk ‘{print $0}’ | head –n 100 | less

查看日志中出现100次以上的IP

cat access\_log | cut –d ‘ ’ –f 1 |sort |uniq -c | awk ‘{if($1>100) print $0}’ | sort –nr | less

等等。。。

详情见：<https://blog.csdn.net/qq43599939/article/details/78873150>

### linux如何统计文件中某个字符串出现的频率

1：**grep+wc**

grep –o targetStr finename | wc –l #单个字符串

grep –o ‘targetStr1\|targetStr2’ finename | wc –l #多个字符串

2：awk

awk –v RS=”@#$j” ‘{print gsub(/targetstr/,”$”}’ filename

### linux启动的第一个进程

**init进程**是内核启动的第一个进程，它是**后续进程的发起者**。

内核启动init进程的过程如下：

1：打开标准输入、标准输出、标准错误文件。

2：如果radmdisk\_execute\_command指定了要运行的程序，则启动它。

3：如果excute\_command指定了要运行的程序，则启动它。

4：依次尝试执行/sbin/init、/etc/init、/bin/init、/bin/sh。

详情见：<https://blog.csdn.net/chuhongcai/article/details/53931371>

### linux查看端口占用

1：lsof –i #查看所有的服务端口。

2：lsof –i:端口号 #查看占用端口

3：netstat –a #查看所有的服务端口。

4：netstat –an | grep 端口号 #检验下是不是已经打开了某端口。

详情见：<https://www.cnblogs.com/CEO-H/p/7794306.html>。

### linux查看CPU和内存使用

ps命令可以实时的现实各个进程的内存使用情况。

top命令提供了实时的运行中的程序的资源使用统计。

atop命令是一个终端环境的监控命令。

/proc/meminfo查看RAM使用情况最简单的方法是通过查看/proc/meminfo文件。

free命令是一个快速查看内存使用情况的方法，它是对/proc/meminfo收集到的信息的一个概述。

详情见：<https://www.cnblogs.com/zhuiluoyu/p/6154898.html>。

### Linux查看系统负载命令

top

uptime

w

vmstat

详情见：<https://blog.csdn.net/qq_36357820/article/details/76606113>。

### Linux调试程序

1：printf语句。

2：查询(cpu信息，内存容量)。

3：跟踪工具- strace的和ltrace是两个在Linux中用来追踪程序的执行细节的跟踪工具。

4：GDB-来自自由软件基金会的调试器。当被调试的程序运行时，它给用户控制权去执行各种动作。比如:

启动程序

停在指定位置

停在指定的条件

检查所需信息

改变程序中的数据。

你也可以将一个崩溃的程序coredump附着到GDB并分析故障的原因。

详情见：<https://www.cnblogs.com/bokeyuan-dlam/articles/9157857.html>。

### Linux硬链接和软连接

**硬链接总结：**

**1：具有相同inode（索引节点）号的多个文件互为硬链接文件；**

**2：删除硬链接文件或者删除源文件任意之一，文件实体并未被删除；**

3：只有删除了源文件和所有对应的硬链接文件，文件实体才会被删除；

4：硬链接文件是文件的另一个入口；

5：可以通过给文件设置硬链接文件来防止重要文件被误删；

6：创建硬链接命令 ln 源文件 硬链接文件；

7：硬链接文件是普通文件，可以用rm删除；

8：对于静态文件（没有进程正在调用），当硬链接数为0时文件就被删除。注意：如果有进程正在调用，则无法删除或者即使文件名被删除但空间不会释放。

**软连接总结：**

1：软链接类似windows系统的快捷方式；

2：软链接里面存放的是源文件的路径，指向源文件；

3：**删除源文件，软链接依然存在，但无法访问源文件内容；**

4：软链接失效时一般是白字红底闪烁；

5：创建软链接命令 ln -s 源文件 软链接文件；

6：软链接和源文件是不同的文件，文件类型也不同，inode号也不同；

7：软链接的文件类型是“l”，可以用rm删除。

详情见：<https://www.cnblogs.com/Peter2014/p/7594504.html>。

### core dump

**当程序运行过程中异常终止或崩溃，操作系统会将程序当时的内存状态记录下来，保存在一个文件中，这种行为就叫做core dump（核心转储）。**但实际上，除了**内存信息**之外，还有些关键的**程序运行状态**也会同时 dump 下来，例如**寄存器信息（包括程序指针、栈指针等）、内存管理信息、其他处理器和操作系统状态和信息**。**core dump文件可以再现程序出错时的情景。**

### cmake和makefile-待加

## 操作系统、进程和线程

### 提升磁盘IO性能的几个技巧

1.  文件越多读取越慢：如果可以的话，将多个小文件合并成一个文件。

2. 读写次数越多读取越慢：一次多读一些数据到内存。

3. 将读写操作分配到不同的硬盘上。

4. 磁盘[RAID0](https://www.baidu.com/s?wd=RAID0&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)比[RAID5](https://www.baidu.com/s?wd=RAID5&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)读写速度快很多。

详情见：<https://blog.csdn.net/guowenyan001/article/details/9190585>。

### 物理内存和虚拟内存（见深入理解操作系统P559）

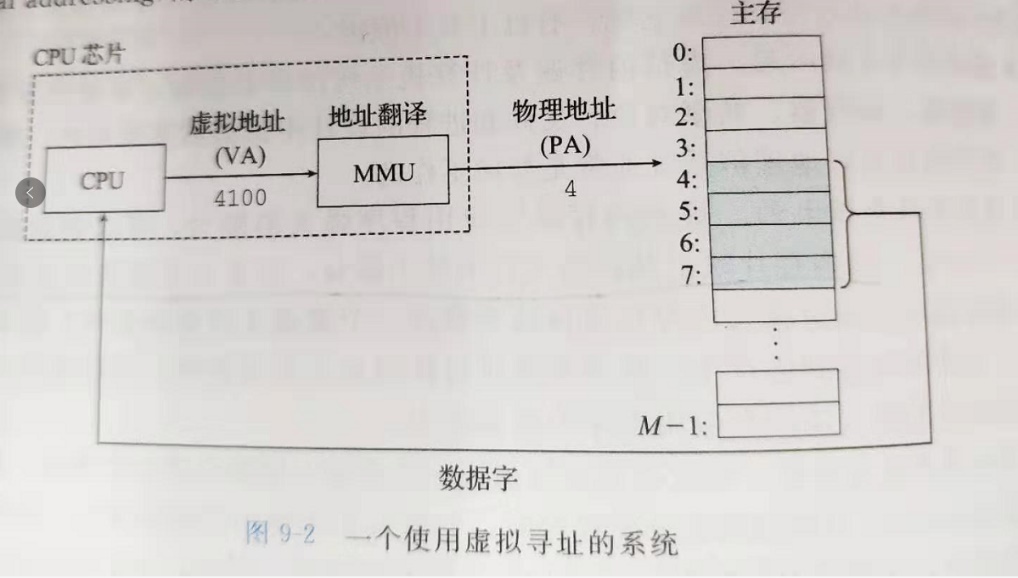
虚拟内存提供了三个重要的能力：

1： 它将主存看成是一个存储在磁盘上的地址空间的高速缓存，在主存中只保存活动区域，并根据需要在磁盘和主存之间来回传递数据，通过这种方式，它高效地使用了主存。

2： 它为每个进程提供了一致的地址空间，从而简化了内存管理。

3: 它保护了每个进程的地址空间不被其他进程破坏。

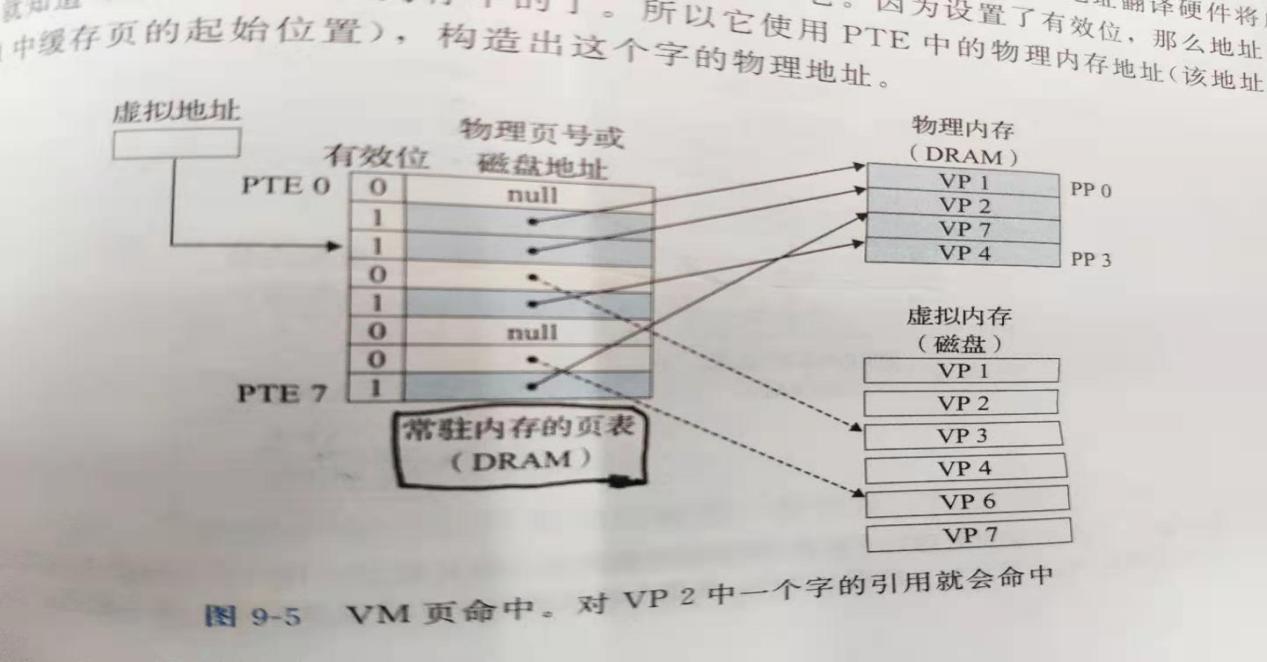
计算机系统的主存被组织成一个由M个连续的字节大小的单元组成的数组，每字节都有一个唯一的物理地址。第一个字节的地址为0，给定这种简单的结构，CPU访问内存的最自然的方式就是使用物理地址，我们把这种方式成为物理寻址。



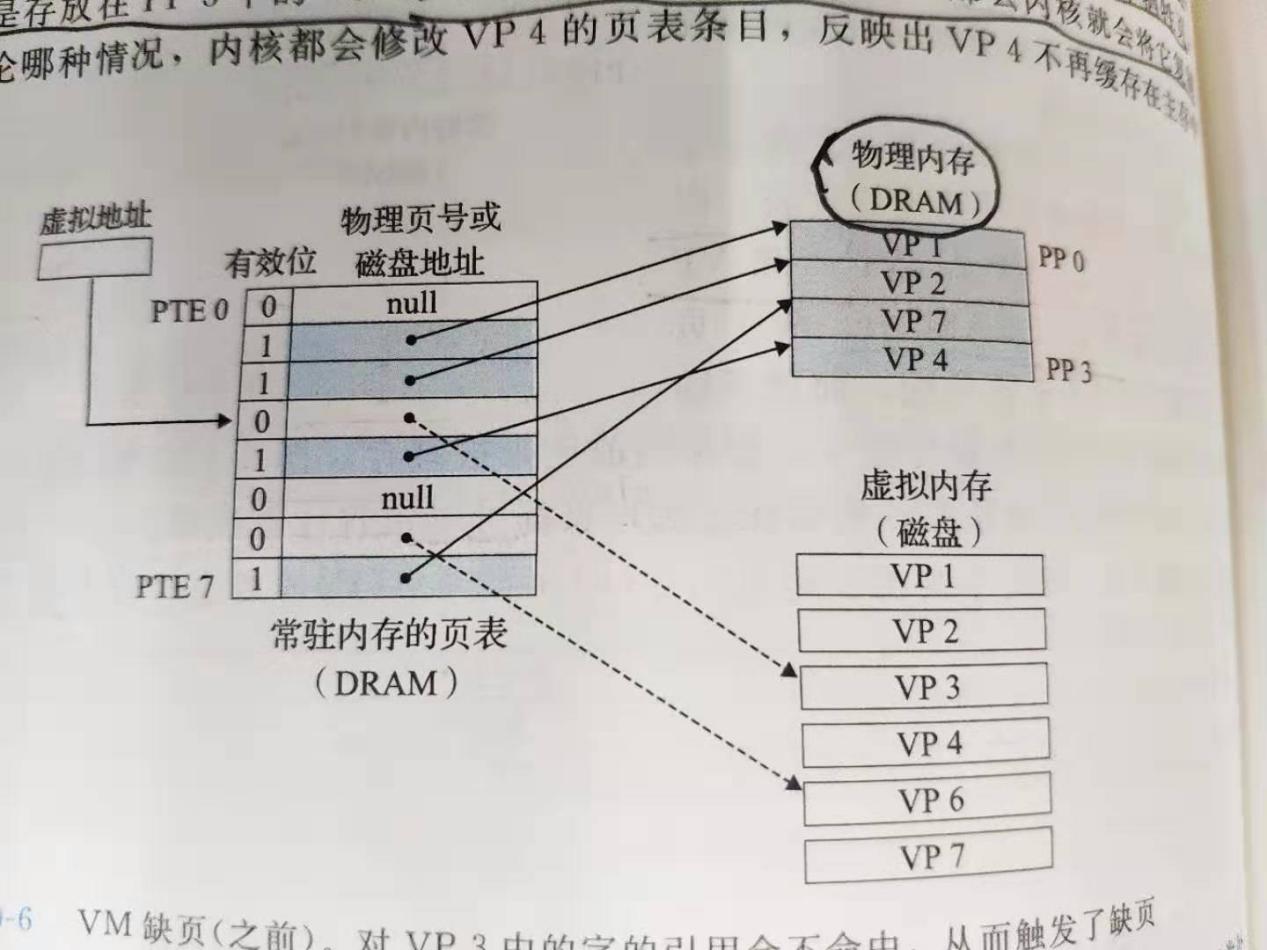
使用虚拟地址，CPU通过生成一个虚拟地址（Virtual Address，VA）来访问主存，这个虚拟地址在被送到内存之前先转换成适当的物理地址。将一个虚拟地址转换成物理地址的任务叫做地址翻译。CPU芯片上叫做内存管理单的专用硬件，利用存放在主存中的查询表来动态翻译虚拟地址，该表的内容由操作系统管理。

出现页命中和缺页等名词

页命中



缺页



这样那个就会出现一系列页面置换算法

页面置换算法：

1：最佳置换算法（OPT）

理想状况下的页面置换算法，但实际上是不可能实现的 思想是：发生缺页时，有些页面在内存中，其中有一页将很快被访问，而其他页面可能要等到10，100,1000条指令后才会被访问，每个页面都可以用在该页面首次被访问前所要执行的指令数进行标记。最佳页面置换算法规定：标记最大的页应该被置换。

2：先进先出置换算法（FIFO）

最简单的页面置换算法是先入先出（FIFO）法。这种算法的实质是，总是选择在主存中停留时间最长（即最老）的一页置换，即先进入内存的页，先退出内存。理由是：最早被调入内存的页，其不再被使用的可能性比刚调入内存的可能性大。

3：最近最久未使用（LRU）算法

FIFO算法和OPT算法之间的主要差别是，FIFO算法利用页面进入内存后的时间长短作为置换依据，而OPT算法的依据是将来使用页面的时间，如果以最近的过去作为不久将来的近似，那么就可以把过去最长一段时间里不曾被使用的页面置换掉。它的实质是：当需要置换一页时，选择在最近一段时间里最近没有使用过的页面予以置换。

LRU算法实现： (Leetcode 146)

代码链接：[LRU算法](https://paste.ubuntu.com/p/syWcCW5n7s/)

struct CacheNode

{

int key, value;

CacheNode(int k, int v) :key(k), value(v){}

};

class LRUCache

{

public:

LRUCache(int capacity)

{

size = capacity;

}

int get(int key)

{

/\*如果找不到，则返回-1

若找到，则更新，并且把这个页面放到栈顶，并且更新unordered\_map\*/

if (lru\_map.find(key) == lru\_map.end()) return -1;

else

{

lru\_list.splice(lru\_list.begin(), lru\_list, lru\_map[key]);

lru\_map[key] = lru\_list.begin();

return lru\_list.front().value;

}

}

void put(int key, int value)

{

/\*

分为两种情况

1：如果不存在，

容量超过限制：删除最近最久未使用的，然后把心的放到list头部，并且更新unordered\_map

没有超过限制，则直接放在头部

2：如果存在，则放到list头部，并且更新unordered\_map

\*/

if (lru\_map.find(key) == lru\_map.end())

{

int list\_size = lru\_list.size();

if (list\_size >= size)

{

lru\_map.erase(lru\_list.back().key);

lru\_list.pop\_back();

CacheNode cache\_node(key, value);

lru\_list.push\_front(cache\_node);

lru\_map[key] = lru\_list.begin();

}

else

{

CacheNode cache\_node(key, value);

lru\_list.push\_front(cache\_node);

lru\_map[key] = lru\_list.begin();

}

}

else

{

//最后要把值更新一下，因为可能页面内容会换

lru\_list.splice(lru\_list.begin(), lru\_list, lru\_map[key]);

lru\_map[key] = lru\_list.begin();

lru\_list.front().value = value;

}

}

private:

list<CacheNode> lru\_list; //双向链表的容器

typedef list<CacheNode>::iterator Iter;

unordered\_map<int, Iter> lru\_map;

int size;

};

/\*\*

\* Your LRUCache object will be instantiated and called as such:

\* LRUCache obj = new LRUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

详情见：<https://blog.csdn.net/zhyfxy/article/details/70157248>。

### 缓存IO和直接IO

缓存I/O又称为标准I/O，大多数文件系统的默认I/O操作都是缓存I/O。在Linux的缓存I/O机制中，操作系统会将I/O的数据缓存在文件系统的页缓存中，即数据会先被拷贝到[操作系统内核](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E5%86%85%E6%A0%B8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。

缓存I/O的优点：

1：缓存I/O使用了操作系统内核缓冲区，在一定程度上分离了应用程序空间和实际的物理设备。

2：缓存I/O可以减少读盘的次数，提高性能。

直接I/O技术：

省略掉缓存I/O技术中操作系统内核缓冲区的使用，数据直接在应用程序地址空间和磁盘之间进行传输，从而使得自缓存应用程序可以省略掉复杂的系统级别的缓存结构。

直接I/O优点：

直接I/O技术的优点：减少操作系统内核缓冲区和应用程序地址空间的数据拷贝的次数，降低了对文件读取和写入时所带来的CPU的使用以及内存带宽的占用。如果传输的数据量很大，使用直接I/O的方式进行数据传输，而不需要操作系统内核地址空间拷贝数据操作的参与，提高性能。

缺点：开销非常大。

详情见：<https://blog.csdn.net/sinat_29844779/article/details/72878828>。

### 作业调度算法

1：先来先服务算法（FCFS调度算法）

先进入输入井的作业先被服务，值得注意的是，并不是先来的就一定会被先放入主存区，必须要有合适的而资源才行。

缺点：当作业周转时间长的作业占用主存资源时，主存区资源长时间不能被释放，后面周转时间少的作业只能处于等待状态，这无疑减少了系统的吞吐量。

2：短作业优先算法（SJF调度算法）

短作业调度算法的核心在于，运行时间越短的作业就先被执行。

有两个值得注意的地方：

1：首先第一到达的作业一定先运行

2：要注意，在后续比较作业时间长短时，要看作业是否到达，就是上图的到达时间，没有到达的作业是不能比较的。

3：高响应比优先调度算法

变换上式可以得到：优先权 = 1 + （等待时间 / 要求服务时间）

综合了作业的等待时间和作业的运行时间。

4：优先级调度算法

这种算法使用优先级来决定谁先被服务。 设定优先级会有两种方法，第一是用户自己设定，第二种是系统设定。系统设定会根据作业的类型，作业的等待时间，计算时间，作业的缓急程度等等各方面的因素来判断是否应该被服务，有些系统还能在输入井根据各方面因素来动态改变优先级。

### 进程地址空间分布

在32位操作系统中，内存空间拥有4GB的寻址能力。操作系统会把高地址的空间分配给内核，称为内核空间。

（1）内核空间：默认情况下，Windows将高地址的2GB空间分配给内核，Linux将高地址的1GB空间分配给内核。剩下的2GB或3GB的内存空间称为用户空间。

在用户空间里，有许多地址区间有特殊的地位，一般来讲，应用程序使用的内存空间里有如下"默认"的区域。

（2）栈：用于维护函数调用的上下文。栈通常在用户空间的最高地址处分配，通常有数兆字节的大小。

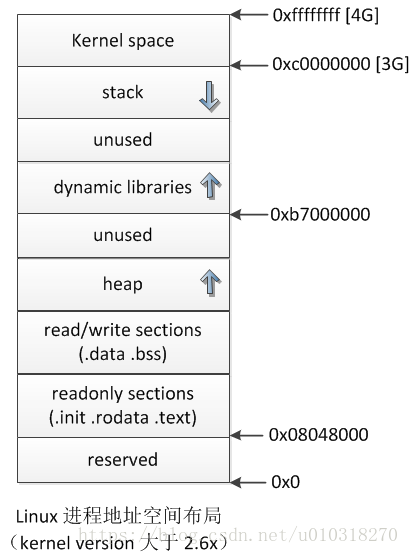
（3）动态链接库映射区：用于映射装载的动态链接库，在linux中，如果可执行文件依赖于其他共享库，系统将在0xbfxxxxxx附近分配地址(kernel > 2.6) ，并将共享库载入到该空间。[kernel = 2.4x，从0x40000000开始分配]

（4）堆：用来容纳应用程序动态分配的内存区域，malloc或new分配内存时，得到的内存来自堆里。堆通常存在于栈的下方(低地址方向)，可以有几十或数百兆字节。

（5）可执行文件映像：存储着可执行文件在内存里的映像，由装载器在装载时将可执行文件的内存读取或映射到这里。包括read/write sections和readonly sections。

（6）保留区：对内存中受到保护而禁止访问的内存区域的总称。

栈向低地址增长，堆向高地址增长。当栈或堆现有的大小不够用时，它将按照图中的增长方向扩大自身的尺寸，直到预留的空间被用完为止。



Q：32位系统一个进程最多申请多大堆

A：2.6版的linux，malloc的最大空间申请数应该在2.9G左右。其中可执行文件占去一部分，0x08048000之前的地址占去一部分，栈占去一部分，共享库占去一部分。

详情见：<https://blog.csdn.net/u010318270/article/details/81058090>。

### 线程和进程

线程：是程序的最小执行单位，行程中的实际运作单位。

进程：是指程序执行中的一个实例。

区别：1.线程程序任务调度和执行的最小单位。进程是资源分配的最小单位2.进程拥有独立的栈堆空间和数据段，启动一个新的进程必须分配给它独立的地址空间，系统开销大。线程拥有独立的栈堆空间，但是共享数据段，开销小，切换速度快，效率高。3.进程间相对独立，安全性高。一个线程死掉等于整个进程死掉。4.进程间相对独立，通信机制较复杂。线程通信机制由于共享数据段，通信机制方便。5.同进程的所有线程贡献该进程的所有资源，包括文件描述符，而不同进程相对独立。6.线程必定只能属于一个进程，而进程可以拥有多个线程而且至少拥有一个线程。

线程和进程场景选择：1.创建和销毁一个进程代价很大，需要频繁创建销毁优先使用线程。2.线程切换速度快，在需要大量计算、切换频繁时用线程，耗时的操作使用线程可提高应用程序的响应。3.对CPU系统的效率上线程占有，所以可能要发展到多级分布的用进程、多喝分布用线程。4.并行操作用线程。5.需要更稳定安全时，选择进程，需要速度时，选线程。

对于线程弄清两点是非常重要的。

线程之间有无先后访问顺序（线程依赖关系）

多个线程共享访问一个变量（同步互斥问题）

### 进程和线程的调度

对于多道程序设计的系统，就会有多个进程或者线程在同时竞争CPU。对于单核系统，调度问题，就是选择下一个要运行的进程或者线程是哪一个。

线程的调度与进程类似，对于按内核级别的调度，与线程所属的进程基本没有关系。

进程切换的代价是比较大的，包括用户态到内核态的切换、保存当前进程的状态、内存映像的改变、调度程序以及载入新进程的状态；另外，会导致高速缓存的失效。

调度程序要考虑的要素：

1：进程是CPU密集型还是I/O密集型。I/O密集型可能会需要尽快运行，并且很快就阻塞。CPU密集型可能会长期占用CPU。

2：何时调度。创建进程、进程退出、进程阻塞以及I/O中断发生。

3：是否支持抢占。支持抢占的调度，可以不必等待当前进程运行完或阻塞，直接上CPU运行。

调度算法的分类。

不同的环境需要不同的调度算法，也有不同的考量指标：

1：批处理。批处理系统下，不需要特别快的响应速度，所以可以考虑非抢占以及每个进程都有长时间的抢占算法。指标主要是：吞吐量、周转时间以及CPU利用率。

2：交互式。交互式系统下，抢占是必须的，因为操作者会希望比较快的得到响应。考虑的指标主要是最小响应时间、均衡性。

3：实时。实时系统下，最重要的是满足截止时间要求，即在一定的时间内完成任务。

批处理系统中的调度

1：先来先服务（first-come-first-serve）。最简单的方式，非抢占式的先来先服务，从字面意思就可以看出，把任务作为一个队列，先来先服务。缺点是，无法体现任务的轻重缓急，达不到理想的指标性能。

2：最短作业优先。非抢占式下，即当一个任务完成时，从任务队列中挑选最短的一个作业执行。相对于先来先服务，提高了一些性能。运行时间必须提前掌握。

3：最短剩余时间优先。即最短作业优先的抢占版本。调度程序总是选择剩余运行时间最短的作业执行。每当一个新的作业到达，如果运行时间比当前进程的剩余运行时间短，就挂起当前进程并切换到新的进程。

交互式系统中的调度

1：轮转调度。最简单且最公平的方法，给每个进程分配一个时间片。时间片耗尽时，进程会下CPU并加入到就绪队列的末尾。问题的关键是选择合适的时间片。

2：优先级调度。进程有轻重缓急，于是给进程设置不同的优先级，每次调度优先级最高的进程运行。当然，这样可能会导致低优先级进程饥饿。解决方案是，实行奖惩机制，高优先级进程耗尽时间片时会降低它的优先级。

3：多级队列。给不同优先级的进程队列，设置不同单位的时间片。同时，也实行奖惩机制，耗尽时间片会改变进程的队列级别。

线程的调度

线程的调度，取决于支持的是内核级线程还是用户级线程。

对于用户级线程，内核不知道线程的存在，就给了进程很大的自主权。内核只是调度进程，进程中的调度程序选择哪个线程来运行。

对于内核级线程，线程的调度就交给了系统完成。

Linux中的进程与线程调度

首先明确一个概念，Linux系统中甚至没有真正的线程。不过，可以认为Linux是系统的线程是内核线程，所以调度是基于线程的。

一个进程由于其运行空间的不同, 从而有内核线程和用户进程的区分, 内核线程运行在内核空间, 之所以称之为线程是因为它没有虚拟地址空间, 只能访问内核的代码和数据, 而用户进程则运行在用户空间, 但是可以通过中断, 系统调用等方式从用户态陷入内核态。

用户进程运行在用户空间上, 而一些通过共享资源实现的一组进程我们称之为线程组, Linux下内核其实本质上没有线程的概念, Linux下线程其实上是与其他进程共享某些资源的进程而已。但是我们习惯上还是称他们为线程或者轻量级进程。

因此, Linux上进程分3种，内核线程（或者叫核心进程）、用户进程、用户线程, 当然如果更严谨的，也可以认为用户进程和用户线程都是用户进程。

Linux中，进程和线程都被维护为一个task\_struct结构，线程和进程被同等对待来进行调度。

Linux将线程区分为3类：

1：实时先入先出。

2：实时轮转。

3：分时。

实时先入先出有最高的优先级，不会被其他线程抢占，除非是另外一个刚刚准备好的且优先级更高的实时先入先出线程。

实时轮转线程与实时先入先出类似，不过有一个轮转体系，即分配一个时间片，时间到了就可以被抢占。时间片消耗完就进入实时轮转线程列表的末尾。其实，这两种都不是真的实时，因为执行的最后期限无法确定，只是比分时线程有更高的优先级。

实时线程的优先级从0-99,0是实时线程的最高优先级，99是实时线程的最低优先级。

传统的非实时线程，优先级从100-139。Linux系统根据非实时线程的优先级分配时间量。

Linux使用一个重要的结构，调度队列。每个CPU有自己的调度队列，包括两个数组：活动的和过期失效的。每个数组包括了140个链表头，对应140个优先级的链表。

调度器从正在活动数组中选择一个优先级最高的任务，如果时间片耗尽失效，就加入到过期失效数组中。如果进程在时间片内被阻塞，那么在时间片失效之前，等待的事件发生就可以继续运行，放回到正在活动的数组中。如果活动数组没有任务了，调度器交换指针，使得活动数组和失效数组调换。

不同的优先级被赋予不同的时间片长度，优先级越高的进程，时间片越长。

Linux采用静态优先级与动态优先级结合的方式。Linux采用了奖惩机制，目的在于奖励互动进程以及惩罚占用CPU的进程。一个进程初始被赋予了一个优先级，耗尽和阻塞会改变nice值，活动与过期进程转换时，动态改变进程优先级。

另外，对于多核处理器，运行队列数据结构与某一个处理器相对应，调度器尽量进行亲和调度，即将之前在某个处理器上运行过的任务再次调入该处理器。

调度器只考虑可以运行的任务，不可运行的任务和正在等待各种I/O操作的或内核事件的任务被放入等待队列中。每一种等待某种事件的任务组成一个等待队列。等待队列的头部包含一个指向任务链表的指针及一个自旋锁。

详情见：<https://www.cnblogs.com/lustar/p/7716165.html>。

### 线程的创建和结束

背景知识：

在一个文件内的多个函数通常都是按照main函数中出现的顺序来执行，但是在分时系统下，我们可以让每个函数都作为一个逻辑流并发执行，最简单的方式就是采用多线程策略。在main函数中调用多线程接口创建线程，每个线程对应特定的函数（操作），这样就可以不按照main函数中各个函数出现的顺序来执行，避免了忙等的情况。线程基本操作的接口如下。

1： 创建线程

int pthread\_create(pthread\_t \*pthread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*agr);

创建一个线程pthread和start\_routine不可或缺，分别用于标识线程和执行入口，其他可以填NULL。

· pthread：用来返回线程的id， \*pthread值即为tid，类型为 pthread\_t = unsigned long int

· attr：指向线程属性结构体的指针，用于改变所创建线程的属性，填NULL使用默认值

· start\_routine：线程执行函数的首地址，传入函数指针

· arg: 通过地址传递来传递函数参数，这里是无符号类型指针，可以传任意类型变量的地址，在被传入函数中先强制类型转换成所需类型即可。

2：获取线程ID: pthread\_t pthread\_self();

3：等待线程结束 int pthread\_join(pthread\_t tid, void\*\* reval);

主线程调用，等待子线程退出并回收资源，类似于进程中wait/waitpid回收僵尸进程，调用pthread\_join线程会被阻塞

· tid：创建线程时通过指针得到tid值

· reval：指向返回值的指针

4：线程结束： pthread\_exit(void \*retval);

子线程执行，用来结束当前线程并通过retval传递返回值，该返回值可通过pthread\_join获得。

5：分离线程：int pthread\_detach(pthread\_t tid);

主线程、子线程均可调用。主线程中pthread\_detach(tid),子线程中pthread\_detach(pthread\_self()),调用后和主线程分离，子线程结束时自己立即回收资源。

线程属性对象类型为pthread\_attr\_t，结构体定义如下：

typedef struct{

int etachstate; // 线程分离的状态

int schedpolicy; // 线程调度策略

struct sched\_param schedparam; // 线程的调度参数

int inheritsched; // 线程的继承性

int scope; // 线程的作用域

// 以下为线程栈的设置

size\_t guardsize; // 线程栈末尾警戒缓冲大小

int stackaddr\_set; // 线程的栈设置

void \* stackaddr; // 线程栈的位置

size\_t stacksize; // 线程栈大小

}pthread\_arrt\_t；

### 线程状态

新建状态、就绪状态、运行状态、阻塞状态(等待阻塞、同步阻塞、其他阻塞)、死亡状态。

1：新建状态(New)：新创建了一个线程对象。

2：就绪状态(Runnable)：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于“可运行线程池”中，变得可运行，只等待获取CPU的使用权。即在就绪状态的进程除CPU之外，其它的运行所需资源都已全部获得。

3：运行状态(Running)：就绪状态的线程获取了CPU，执行程序代码。

4：阻塞状态(Blocked)：阻塞状态是线程因某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

阻塞的情况分三种：

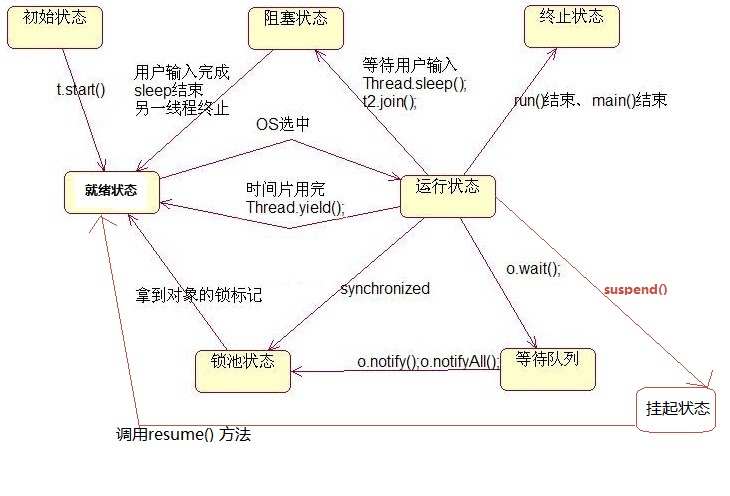
等待阻塞：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，唤醒后进入“锁池”中，通过获取锁状态来判断是否进入就绪状态

同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。

其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5：死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

线程变换的状态转化图如下：



注：拿到对象的锁标记，即为获得了对该对象(临界区)的使用权限。即该线程获得了运行所需的资源，进入“就绪状态”，只需获得CPU，就可以运行。因为当调用wait()后，线程会释放掉它所占有的“锁标志”，所以线程只有在此获取资源才能进入就绪状态。

下面小小的作下解释：

1：线程的实现有两种方式，一是继承Thread类，二是实现Runnable接口，但不管怎样，  当我们new了这个对象后，线程就进入了初始状态；

2：当该对象调用了start()方法，就进入就绪状态；

3：进入就绪后，当该对象被操作系统选中，获得CPU时间片就会进入运行状态；

4：进入运行状态后情况就比较复杂了

4.1、run()方法或main()方法结束后，线程就进入终止状态；

    4.2、当线程调用了自身的sleep()方法或其他线程的join()方法，进程让出CPU，然后就会进入阻塞状态（该状态既停止当前线程，但并不释放所占有的资源即调用sleep ()函数后，线程不会释放它的“锁标志”。）。当sleep()结束或join()结束后，该线程进入可运行状态，继续等待OS分配CPU时间片。典型地，sleep()被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。

    4.3、线程调用了yield()方法，意思是放弃当前获得的CPU时间片，回到就绪状态，这时与其他进程处于同等竞争状态，OS有可能会接着又让这个进程进入运行状态； 调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间片从而需要转到另一个线程。yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行。

   4.4、当线程刚进入可运行状态（注意，还没运行），发现将要调用的资源被synchroniza（同步），获取不到锁标记，将会立即进入锁池状态，等待获取锁标记（这时的锁池里也许已经有了其他线程在等待获取锁标记，这时它们处于队列状态，既先到先得），一旦线程获得锁标记后，就转入就绪状态，等待OS分配CPU时间片；

    4.5、suspend() 和 resume()方法：两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume()被调用，才能使得线程重新进入可执行状态。典型地，suspend()和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume()使其恢复。

    4.6、wait()和 notify() 方法：当线程调用wait()方法后会进入等待队列（进入这个状态会释放所占有的所有资源，与阻塞状态不同），进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒（由于notify()只是唤醒一个线程，但我们由不能确定具体唤醒的是哪一个线程，也许我们需要唤醒的线程不能够被唤醒，因此在实际使用时，一般都用notifyAll()方法，唤醒有所线程），线程被唤醒后会进入锁池，等待获取锁标记。

wait() 使得线程进入等待队列中，它有两种形式：

一种允许指定以毫秒为单位的一段时间作为参数；另一种没有参数。前者当对应的 notify()被调用或者超出指定时间时线程重新进入可执行状态即就绪状态，后者则必须对应的 notify()被调用。当调用wait()后，线程会释放掉它所占有的“锁标志”，从而使线程所在对象中的其它synchronized数据可被别的线程使用。waite()和notify()因为会对对象的“锁标志”进行操作，所以它们必须在synchronized函数或synchronizedblock中进行调用。如果在non-synchronized函数或non-synchronizedblock中进行调用，虽然能编译通过，但在运行时会发生IllegalMonitorStateException的异常。

详情见：<https://blog.csdn.net/Jacoob1024/article/details/81097721>。

### 线程间通信与线程同步-待加

详情见：<https://blog.csdn.net/zcyzsy/article/details/52947558>。

### 互斥锁和信号量-待加

### 多线程程序如何调试

一：关于设置断点和单步执行

很多同学非常依赖于调试器的断点功能和单步功能。这在单线程情况下倒还好（不过有些单线程但涉及GUI的程序，也会有点麻烦）。至于多线程程序的调试，这两种手段简直就是噩梦的开始。多线程造成的主要问题大都和竞态条件（Race Condition，详细解释看“这里 ”）有关。

而设置断点或单步跟踪可能会严重干扰 多线程之间的竞争状态。导致你看到的是一个假象。比如本来有两个线程并发执行，存在某些不和谐的Bug（由竞态引起）。一旦你在某一个线程设置了断点，该线程在断点处停住了，只剩下另一个线程在跑。这时候，并发的场景已经完全被破坏了，你通过调试器看到的可能 是一个和谐的场景。

二：关于Log输出

既然断点和单步不好用。那咋办捏？一个替代方案是输出log日志。它可以有效减轻断点和单步所导致的（针对竞态条件的）副作用。

1：传统Log机制的问题

传统的log输出主要是打印到屏幕或者输出到文件。对于C++而言，标准库内置的类和函数（比如cout、printf、fputs）可能会有线程安全的问题（和编译器的具体实现有关）。尤其是标准流类库（iostream）的八个全局对象，更是要小心慎用。轻则输出的log文本混杂，重则导致程序崩溃。

鉴于上述原因，应该尽量使用第三方线程库内置的log机制来搞定log输出功能。比如ACE内置的ACE\_Log\_Msg等。

2：Log函数要短小精悍

很多情况下，我们会包装一个公用的函数来实现log输出功能。然后在该函数内部调用线程库的log类/函数。为了不影响线程的竞态条件，这个log函数要尽可能简单轻便：不要涉及太多杂七杂八的琐事、千万别进行耗时的操作、尽量不操作一些全局的变量。

3：Log的副作用

不过捏，即使log函数再短小精悍，也还是有可能影响竞态条件（毕竟log也有开销，也要消耗CPU时间）。

万一竞态条件受到log的影响，那就比较棘手了。我以前就碰到过这种情况：加了log，程序没有问题；去掉log，程序随机崩溃。这种情况一般有两种可能：要么是log功能本身有问题，要么是程序的竞态条件非常敏感（连log的开销都会有影响）。

这时候你能依靠的就只有肉眼和人脑了。先把相关的代码和文档仔细看上几遍（最好再找其他有经验的人一起Code Review），然后大家一起开动脑筋使劲琢磨。

三：GDB

gdb是Unix下用来调试C和C++程序的常用的调试器. 它使你能在程序运行时观察程序的内部结构和内存的使用情况。但在多线程编程过程中很多程序问题出在启动阶段，而且很难使用gdb进行调试，我们可以采用手工插入以下辅助代码暂停程序运行

四：gcc的内置函数打印程序调用的帧栈

由于程序中函数的调用是存储在内存栈中，本章通过介绍使用gcc的内置函数(见以下)来打印函数栈，来显示函数的调用的层次关系，来调试程序。

### C++实现线程安全的队列

std::queue这个容器提供了pop(),push(),empty(),等这些读写操作容器的函数，只要对这些函数上面加个锁，就可以使其线程安全。

详情见：<https://blog.csdn.net/u011726005/article/details/82670730>。

### 线程池

线程池是一种多线程处理形式，将任务添加到队列中，然后在创建线程后，自动启动这些任务。线程池线程都是后台线程。

使用线程池理由：1.频繁创建、销毁加大系统开销，影响处理效率。2.限制线程最大并发数。3.管理线程。

### 进程的创建和结束

背景知识：

进程创建有两种方式，一种是操作系统创建的，一种是父进程创建的。

从计算机启动到终端执行程序的过程为：0号进程 -> 1号内核进程 -> 1号用户进程(init进程) -> getty进程 -> shell进程 -> 命令行执行进程。所以我们在命令行中通过 ./program执行可执行文件时，所有创建的进程都是shell进程的子进程，这也就是为什么shell一关闭，在shell中执行的进程都自动被关闭的原因。

相关接口：

· 创建进程：pid\_t fork(void);

返回值：出错返回-1；父进程中返回pdi>0；子进程中pid ==0

· 结束进程：void exit(int status)

status是退出状态，保存在全局变量，通常0表示正常退出

· 获取PID： pid\_t getpid(void)

返回调用者pid

· 获得父进程pid： pid\_t getppid(void)

返回父进程pid

### 如何创建新进程，内核做了啥

在linux中主要提供了fork，vfork，clone三个进程创建方法。

在linux源码中这是哪个调用的执行过程是执行fork,vfork,clone时，通过一个系统调用表映射到sys\_fork(),sys\_vfork(),sys\_clone()，再在这三个函数中去调用do\_fork()去做具体的创建进程工作。

详情见：<https://blog.csdn.net/wujiafei_njgcxy/article/details/77116175>

### 孤儿进程、僵尸进程和守护进程

父进程在调用fork接口之后和子进程独立开，之后子进程和父进程就以未知的顺序向下执行（异步过程）。所以父进程和子进程都有可能先执行完。当父进程先结束，子进程此时就会变成孤儿进程，不过这种情况问题不大，孤儿进程会自动向上被init进程收养，init进程完成对状态收集工作。而且这种过继的方式也是守护进程能够实现的因素。如果子进程先结束，父进程并未调用wait或者waitpid获取进程状态信息，那么子进程描述符就会一直保存在系统里，这种进程称为僵尸进程。

相关接口：

回收进程（1）： pid\_wait(int \*status)

一旦调用wait（），就会立刻阻塞自己，wait（）自动分析某个子进程是否退出，如果找到僵尸进程就会负责收集和销毁，如果没有找到就一直阻塞在这里。

status：指向子进程结束状态值。

回收进程（2）： pid\_t waitpid(pid\_t pid,int \*status,int options)

返回值：返回pid：返回收集的子进程id。返回-1：出错 返回0：没有被收集的子进程

· pid：子进程识别码，控制等待哪些子进程。

1. pid < -1,等待进程组识别码为pid绝对值的任何进程
2. Pid = -1，等待任何子进程
3. Pid = 0，等待进程组识别码与目前进程相同的任何子进程
4. Pid>0 ，等待任何子进程识别码为pid的子进程

· status： 指向返回码的指针

· options： 选项决定父进程调用waitpid后的状态

a options = WNOHANG, 即使没有子进程退出也立即返回。

b options = WUNYRACED, 子进程进入暂停马上返回，但结束状态不予理会。

守护进程：

定义：守护进程是脱离终端并在后台运行的进程，执行过程中信息不会显示在终端上并且也不会被终端发出的信号打断。

操作步骤：

创建子进程，父进程退出：fork() + if(pid > 0){exit(0);}，使子进程称为孤儿进程被init进程收养。

·在子进程中创建新会话：setsid()。

·改变当前目录结构为根：chdir("/")。

·重设文件掩码：umask(0)。

·关闭文件描述符：for(int i = 0; i < 65535; ++i){close(i);}。

### 多进程

每一个进程是资源分配的基本单位。进程结构由以下几个部分组成：代码段、堆栈段、数据段。代码段是静态的二进制代码，多个程序可以共享。实际上在父进程创建子进程之后，父、子进程除了pid外，几乎所有的部分几乎一样，子进程创建时拷贝父进程PCB中大部分内容，而PCB的内容实际上是各种数据、代码的地址或索引表地址，所以复制了PCB中这些指针实际就等于获取了全部父进程可访问数据。所以简单来说，创建新进程需要复制整个PCB，之后操作系统将PCB添加到进程核心堆栈底部，这样就可以被操作系统感知和调度了。

父、子进程共享全部数据，但并不是说他们就是对同一块数据进行操作，子进程在读写数据时会通过写时复制机制将公共的数据重新拷贝一份，之后在拷贝出的数据上进行操作。如果子进程想要运行自己的代码段，还可以通过调用execv()函数重新加载新的代码段，之后就和父进程独立开了。我们在shell中执行程序就是通过shell进程先fork()一个子进程再通过execv()重新加载新的代码段的过程。

### 进程间通信方式

进程间通信（IPC，InterProcess Communication）是指在不同进程之间传播或交换信息。

IPC的方式通常有管道（包括无名管道和命名管道）、消息队列、信号量、共享存储、Socket、Streams等。其中 Socket和Streams支持不同主机上的两个进程IPC。

管道，通常指无名管道，是 UNIX 系统IPC最古老的形式。

1：特点

它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。

它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。

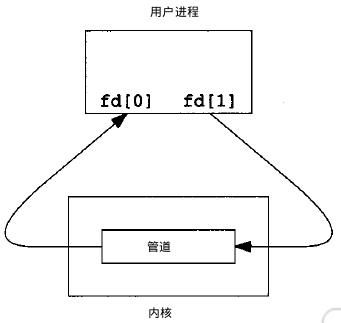
它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

#include <unistd.h>

int pipe(int fd[2]); // 返回值：若成功返回0，失败返回-1

2：原型

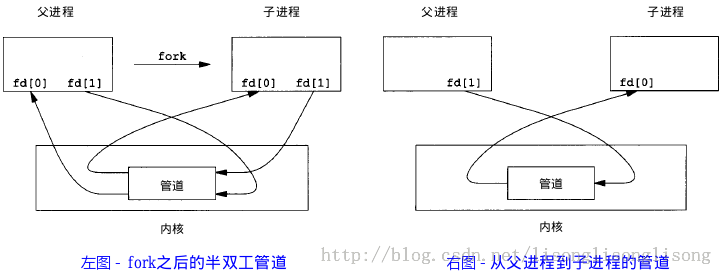
当一个管道建立时，它会创建两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。如下图：



要关闭管道只需将这两个文件描述符关闭即可。

3：例子

单个进程中的管道几乎没有任何用处。所以，通常调用 pipe 的进程接着调用 fork，这样就创建了父进程与子进程之间的 IPC 通道。如下图所示：



若要数据流从父进程流向子进程，则关闭父进程的读端（fd[0]）与子进程的写端（fd[1]）；反之，则可以使数据流从子进程流向父进程。

#include<stdio.h>

#include<unistd.h>

int main()

{

int fd[2]; // 两个文件描述符

pid\_t pid;

char buff[20];

if(pipe(fd) < 0) // 创建管道

printf("Create Pipe Error!\n");

if((pid = fork()) < 0) // 创建子进程

printf("Fork Error!\n");

else if(pid > 0) // 父进程

{

close(fd[0]); // 关闭读端

write(fd[1], "hello world\n", 12);

}

else

{

close(fd[1]); // 关闭写端

read(fd[0], buff, 20);

printf("%s", buff);

}

return 0;

}

FIFO，也称为命名管道，它是一种文件类型。

1：特点

FIFO可以在无关的进程之间交换数据，与无名管道不同。

FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2：原型

#include <sys/stat.h>

// 返回值：成功返回0，出错返回-1

int mkfifo(const char \*pathname, mode\_t mode);

其中的 mode 参数与open函数中的 mode 相同。一旦创建了一个 FIFO，就可以用一般的文件I/O函数操作它。

当 open 一个FIFO时，是否设置非阻塞标志（O\_NONBLOCK）的区别：

若没有指定O\_NONBLOCK（默认），只读 open 要阻塞到某个其他进程为写而打开此 FIFO。类似的，只写 open 要阻塞到某个其他进程为读而打开它。

若指定了O\_NONBLOCK，则只读 open 立即返回。而只写 open 将出错返回 -1 如果没有进程已经为读而打开该 FIFO，其errno置ENXIO。

3：例子

FIFO的通信方式类似于在进程中使用文件来传输数据，只不过FIFO类型文件同时具有管道的特性。在数据读出时，FIFO管道中同时清除数据，并且“先进先出”。下面的例子演示了使用 FIFO 进行 IPC 的过程：

write\_fifo.c

#include<stdio.h>

#include<stdlib.h> // exit

#include<fcntl.h> // O\_WRONLY

#include<sys/stat.h>

#include<time.h> // time

int main()

{

int fd;

int n, i;

char buf[1024];

time\_t tp;

printf("I am %d process.\n", getpid()); // 说明进程ID

if((fd = open("fifo1", O\_WRONLY)) < 0) // 以写打开一个FIFO

{

perror("Open FIFO Failed");

exit(1);

}

for(i=0; i<10; ++i)

{

time(&tp); // 取系统当前时间

n=sprintf(buf,"Process %d's time is %s",getpid(),ctime(&tp));

printf("Send message: %s", buf); // 打印

if(write(fd, buf, n+1) < 0) // 写入到FIFO中

{

perror("Write FIFO Failed");

close(fd);

exit(1);

}

sleep(1); // 休眠1秒

}

close(fd); // 关闭FIFO文件

return 0;

}

read\_fifo.c

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<fcntl.h>

#include<sys/stat.h>

int main()

{

int fd;

int len;

char buf[1024];

if(mkfifo("fifo1", 0666) < 0 && errno!=EEXIST) // 创建FIFO管道

perror("Create FIFO Failed");

if((fd = open("fifo1", O\_RDONLY)) < 0) // 以读打开FIFO

{

perror("Open FIFO Failed");

exit(1);

}

while((len = read(fd, buf, 1024)) > 0) // 读取FIFO管道

printf("Read message: %s", buf);

close(fd); // 关闭FIFO文件

return 0;

}

在两个终端里用 gcc 分别编译运行上面两个文件，可以看到输出结果如下：

[cheesezh@localhost]$ ./write\_fifo

I am 5954 process.

Send message: Process 5954's time is Mon Apr 20 12:37:28 2015

Send message: Process 5954's time is Mon Apr 20 12:37:29 2015

Send message: Process 5954's time is Mon Apr 20 12:37:30 2015

Send message: Process 5954's time is Mon Apr 20 12:37:31 2015

Send message: Process 5954's time is Mon Apr 20 12:37:32 2015

Send message: Process 5954's time is Mon Apr 20 12:37:33 2015

Send message: Process 5954's time is Mon Apr 20 12:37:34 2015

Send message: Process 5954's time is Mon Apr 20 12:37:35 2015

Send message: Process 5954's time is Mon Apr 20 12:37:36 2015

Send message: Process 5954's time is Mon Apr 20 12:37:37 2015

[cheesezh@localhost]$ ./read\_fifo

Read message: Process 5954's time is Mon Apr 20 12:37:28 2015

Read message: Process 5954's time is Mon Apr 20 12:37:29 2015

Read message: Process 5954's time is Mon Apr 20 12:37:30 2015

Read message: Process 5954's time is Mon Apr 20 12:37:31 2015

Read message: Process 5954's time is Mon Apr 20 12:37:32 2015

Read message: Process 5954's time is Mon Apr 20 12:37:33 2015

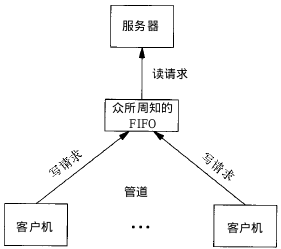
Read message: Process 5954's time is Mon Apr 20 12:37:34 2015

Read message: Process 5954's time is Mon Apr 20 12:37:35 2015

Read message: Process 5954's time is Mon Apr 20 12:37:36 2015

Read message: Process 5954's time is Mon Apr 20 12:37:37 2015

上述例子可以扩展成 客户进程—服务器进程 通信的实例，write\_fifo的作用类似于客户端，可以打开多个客户端向一个服务器发送请求信息，read\_fifo类似于服务器，它适时监控着FIFO的读端，当有数据时，读出并进行处理，但是有一个关键的问题是，每一个客户端必须预先知道服务器提供的FIFO接口，下图显示了这种安排：



消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

1：特点

消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

2：原型

#include <sys/msg.h>

// 创建或打开消息队列：成功返回队列ID，失败返回-1

int msgget(key\_t key, int flag);

// 添加消息：成功返回0，失败返回-1

int msgsnd(int msqid, const void \*ptr, size\_t size, int flag);

// 读取消息：成功返回消息数据的长度，失败返回-1

int msgrcv(int msqid, void \*ptr, size\_t size, long type,int flag);

// 控制消息队列：成功返回0，失败返回-1

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

在以下两种情况下，msgget将创建一个新的消息队列：

如果没有与键值key相对应的消息队列，并且flag中包含了IPC\_CREAT标志位。

key参数为IPC\_PRIVATE。

函数msgrcv在读取消息队列时，type参数有下面几种情况：

type == 0，返回队列中的第一个消息；

type > 0，返回队列中消息类型为 type 的第一个消息；

type < 0，返回队列中消息类型值小于或等于 type 绝对值的消息，如果有多个，则取类型值最小的消息。

可以看出，type值非 0 时用于以非先进先出次序读消息。也可以把 type 看做优先级的权值。

3：例子

下面写了一个简单的使用消息队列进行IPC的例子，服务端程序一直在等待特定类型的消息，当收到该类型的消息以后，发送另一种特定类型的消息作为反馈，客户端读取该反馈并打印出来。

msg\_server.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/msg.h>

// 用于创建一个唯一的key

#define MSG\_FILE "/etc/passwd"

// 消息结构

struct msg\_form {

long mtype;

char mtext[256];

};

int main()

{

int msqid;

key\_t key;

struct msg\_form msg;

// 获取key值

if((key = ftok(MSG\_FILE,'z')) < 0)

{

perror("ftok error");

exit(1);

}

// 打印key值

printf("Message Queue - Server key is: %d.\n", key);

// 创建消息队列

if ((msqid = msgget(key, IPC\_CREAT|0777)) == -1)

{

perror("msgget error");

exit(1);

}

// 打印消息队列ID及进程ID

printf("My msqid is: %d.\n", msqid);

printf("My pid is: %d.\n", getpid());

// 循环读取消息

for(;;)

{

msgrcv(msqid, &msg, 256, 888, 0);// 返回类型为888的第一个消息

printf("Server: receive msg.mtext is: %s.\n", msg.mtext);

printf("Server: receive msg.mtype is: %d.\n", msg.mtype);

msg.mtype = 999; // 客户端接收的消息类型

sprintf(msg.mtext, "hello, I'm server %d", getpid());

msgsnd(msqid, &msg, sizeof(msg.mtext), 0);

}

return 0;

}

msg\_client.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/msg.h>

// 用于创建一个唯一的key

#define MSG\_FILE "/etc/passwd"

// 消息结构

struct msg\_form {

long mtype;

char mtext[256];

};

int main()

{

int msqid;

key\_t key;

struct msg\_form msg;

// 获取key值

if ((key = ftok(MSG\_FILE, 'z')) < 0)

{

perror("ftok error");

exit(1);

}

// 打印key值

printf("Message Queue - Client key is: %d.\n", key);

// 打开消息队列

if ((msqid = msgget(key, IPC\_CREAT|0777)) == -1)

{

perror("msgget error");

exit(1);

}

// 打印消息队列ID及进程ID

printf("My msqid is: %d.\n", msqid);

printf("My pid is: %d.\n", getpid());

// 添加消息，类型为888

msg.mtype = 888;

sprintf(msg.mtext, "hello, I'm client %d", getpid());

msgsnd(msqid, &msg, sizeof(msg.mtext), 0);

// 读取类型为777的消息

msgrcv(msqid, &msg, 256, 999, 0);

printf("Client: receive msg.mtext is: %s.\n", msg.mtext);

printf("Client: receive msg.mtype is: %d.\n", msg.mtype);

return 0;

}

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

1：特点

信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

2：原型

最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二值信号量（Binary Semaphore）。而可以取多个正整数的信号量被称为通用信号量。

Linux 下的信号量函数都是在通用的信号量数组上进行操作，而不是在一个单一的二值信号量上进行操作。

#include <sys/sem.h>

// 创建或获取一个信号量组：若成功返回信号量集ID，失败返回-1

int semget(key\_t key, int num\_sems, int sem\_flags);

// 对信号量组进行操作，改变信号量的值：成功返回0，失败返回-1

int semop(int semid, struct sembuf semoparray[], size\_t numops);

// 控制信号量的相关信息

int semctl(int semid, int sem\_num, int cmd, ...);

当semget创建新的信号量集合时，必须指定集合中信号量的个数（即num\_sems），通常为1； 如果是引用一个现有的集合，则将num\_sems指定为 0 。

在semop函数中，sembuf结构的定义如下：

struct sembuf

{

short sem\_num; // 信号量组中对应的序号，0～sem\_nums-1

short sem\_op; // 信号量值在一次操作中的改变量

short sem\_flg; // IPC\_NOWAIT, SEM\_UNDO

}

其中 sem\_op 是一次操作中的信号量的改变量：

* 若sem\_op > 0，表示进程释放相应的资源数，将 sem\_op 的值加到信号量的值上。如果有进程正在休眠等待此信号量，则换行它们。
* 若sem\_op < 0，请求 sem\_op 的绝对值的资源。
  + 如果相应的资源数可以满足请求，则将该信号量的值减去sem\_op的绝对值，函数成功返回。
  + 当相应的资源数不能满足请求时，这个操作与sem\_flg有关。
    - sem\_flg 指定IPC\_NOWAIT，则semop函数出错返回EAGAIN。
    - sem\_flg 没有指定IPC\_NOWAIT，则将该信号量的semncnt值加1，然后进程挂起直到下述情况发生：
      1. 当相应的资源数可以满足请求，此信号量的semncnt值减1，该信号量的值减去sem\_op的绝对值。成功返回；
      2. 此信号量被删除，函数smeop出错返回EIDRM；
      3. 进程捕捉到信号，并从信号处理函数返回，此情况下将此信号量的semncnt值减1，函数semop出错返回EINTR
* 若sem\_op == 0，进程阻塞直到信号量的相应值为0：
  + 当信号量已经为0，函数立即返回。
  + 如果信号量的值不为0，则依据sem\_flg决定函数动作：
    - sem\_flg指定IPC\_NOWAIT，则出错返回EAGAIN。
    - sem\_flg没有指定IPC\_NOWAIT，则将该信号量的semncnt值加1，然后进程挂起直到下述情况发生：
      1. 信号量值为0，将信号量的semzcnt的值减1，函数semop成功返回；
      2. 此信号量被删除，函数smeop出错返回EIDRM；
      3. 进程捕捉到信号，并从信号处理函数返回，在此情况将此信号量的semncnt值减1，函数semop出错返回EINTR

在semctl函数中的命令有多种，这里就说两个常用的：

SETVAL：用于初始化信号量为一个已知的值。所需要的值作为联合semun的val成员来传递。在信号量第一次使用之前需要设置信号量。

IPC\_RMID：删除一个信号量集合。如果不删除信号量，它将继续在系统中存在，即使程序已经退出，它可能在你下次运行此程序时引发问题，而且信号量是一种有限的资源。

#include<stdio.h>

#include<stdlib.h>

#include<sys/sem.h>

// 联合体，用于semctl初始化

union semun

{

intval; /\*for SETVAL\*/

struct semid\_ds \*buf;

unsigned short \*array;

};

// 初始化信号量

int init\_sem(int sem\_id, int value)

{

union semun tmp;

tmp.val = value;

if(semctl(sem\_id, 0, SETVAL, tmp) == -1)

{

perror("Init Semaphore Error");

return -1;

}

return 0;

}

// P操作:

// 若信号量值为1，获取资源并将信号量值-1

// 若信号量值为0，进程挂起等待

int sem\_p(int sem\_id)

{

struct sembuf sbuf;

sbuf.sem\_num = 0; /\*序号\*/

sbuf.sem\_op = -1; /\*P操作\*/

sbuf.sem\_flg = SEM\_UNDO;

if(semop(sem\_id, &sbuf, 1) == -1)

{

perror("P operation Error");

return -1;

}

return 0;

}

// V操作：

// 释放资源并将信号量值+1

// 如果有进程正在挂起等待，则唤醒它们

int sem\_v(int sem\_id)

{

struct sembuf sbuf;

sbuf.sem\_num = 0; /\*序号\*/

sbuf.sem\_op = 1; /\*V操作\*/

sbuf.sem\_flg = SEM\_UNDO;

if(semop(sem\_id, &sbuf, 1) == -1)

{

perror("V operation Error");

return -1;

}

return 0;

}

// 删除信号量集

int del\_sem(int sem\_id)

{

union semun tmp;

if(semctl(sem\_id, 0, IPC\_RMID, tmp) == -1)

{

perror("Delete Semaphore Error");

return -1;

}

return 0;

}

int main()

{

int sem\_id; // 信号量集ID

key\_t key;

pid\_t pid;

// 获取key值

if((key = ftok(".", 'z')) < 0)

{

perror("ftok error");

exit(1);

}

// 创建信号量集，其中只有一个信号量

if((sem\_id = semget(key, 1, IPC\_CREAT|0666)) == -1)

{

perror("semget error");

exit(1);

}

// 初始化：初值设为0资源被占用

init\_sem(sem\_id, 0);

if((pid = fork()) == -1)

perror("Fork Error");

else if(pid == 0) /\*子进程\*/

{

sleep(2);

printf("Process child: pid=%d\n", getpid());

sem\_v(sem\_id); /\*释放资源\*/

}

else /\*父进程\*/

{

sem\_p(sem\_id); /\*等待资源\*/

printf("Process father: pid=%d\n", getpid());

sem\_v(sem\_id); /\*释放资源\*/

del\_sem(sem\_id); /\*删除信号量集\*/

}

return 0;

}

上面的例子如果不加信号量，则父进程会先执行完毕。这里加了信号量让父进程等待子进程执行完以后再执行。

共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。

1：特点

共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。

因为多个进程可以同时操作，所以需要进行同步。

信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

2：原型

#include <sys/shm.h>

// 创建或获取一个共享内存：成功返回共享内存ID，失败返回-1

int shmget(key\_t key, size\_t size, int flag);

// 连接共享内存到当前进程的地址空间：成功返回指向共享内存的指针，失败返回-1

void \*shmat(int shm\_id, const void \*addr, int flag);

// 断开与共享内存的连接：成功返回0，失败返回-1

int shmdt(void \*addr);

// 控制共享内存的相关信息：成功返回0，失败返回-1

int shmctl(int shm\_id, int cmd, struct shmid\_ds \*buf);

当用shmget函数创建一段共享内存时，必须指定其 size；而如果引用一个已存在的共享内存，则将 size 指定为0 。

当一段共享内存被创建以后，它并不能被任何进程访问。必须使用shmat函数连接该共享内存到当前进程的地址空间，连接成功后把共享内存区对象映射到调用进程的地址空间，随后可像本地空间一样访问。

shmdt函数是用来断开shmat建立的连接的。注意，这并不是从系统中删除该共享内存，只是当前进程不能再访问该共享内存而已。

shmctl函数可以对共享内存执行多种操作，根据参数 cmd 执行相应的操作。常用的是IPC\_RMID（从系统中删除该共享内存）。

3：例子

下面这个例子，使用了【共享内存+信号量+消息队列】的组合来实现服务器进程与客户进程间的通信。

共享内存用来传递数据；

信号量用来同步；

消息队列用来 在客户端修改了共享内存后 通知服务器读取。

server.c

#include<stdio.h>

#include<stdlib.h>

#include<sys/shm.h> // shared memory

#include<sys/sem.h> // semaphore

#include<sys/msg.h> // message queue

#include<string.h> // memcpy

// 消息队列结构

struct msg\_form {

long mtype;

char mtext;

};

// 联合体，用于semctl初始化

union semun

{

int val; /\*for SETVAL\*/

struct semid\_ds \*buf;

unsigned short \*array;

};

// 初始化信号量

int init\_sem(int sem\_id, int value)

{

union semun tmp;

tmp.val = value;

if(semctl(sem\_id, 0, SETVAL, tmp) == -1)

{

perror("Init Semaphore Error");

return -1;

}

return 0;

}

// P操作:

// 若信号量值为1，获取资源并将信号量值-1

// 若信号量值为0，进程挂起等待

int sem\_p(int sem\_id)

{

struct sembuf sbuf;

sbuf.sem\_num = 0; /\*序号\*/

sbuf.sem\_op = -1; /\*P操作\*/

sbuf.sem\_flg = SEM\_UNDO;

if(semop(sem\_id, &sbuf, 1) == -1)

{

perror("P operation Error");

return -1;

}

return 0;

}

// V操作：

// 释放资源并将信号量值+1

// 如果有进程正在挂起等待，则唤醒它们

int sem\_v(int sem\_id)

{

struct sembuf sbuf;

sbuf.sem\_num = 0; /\*序号\*/

sbuf.sem\_op = 1; /\*V操作\*/

sbuf.sem\_flg = SEM\_UNDO;

if(semop(sem\_id, &sbuf, 1) == -1)

{

perror("V operation Error");

return -1;

}

return 0;

}

// 删除信号量集

int del\_sem(int sem\_id)

{

union semun tmp;

if(semctl(sem\_id, 0, IPC\_RMID, tmp) == -1)

{

perror("Delete Semaphore Error");

return -1;

}

return 0;

}

// 创建一个信号量集

int creat\_sem(key\_t key)

{

int sem\_id;

if((sem\_id = semget(key, 1, IPC\_CREAT|0666)) == -1)

{

perror("semget error");

exit(-1);

}

init\_sem(sem\_id, 1); /\*初值设为1资源未占用\*/

return sem\_id;

}

int main()

{

key\_t key;

int shmid, semid, msqid;

char \*shm;

char data[] = "this is server";

struct shmid\_ds buf1; /\*用于删除共享内存\*/

struct msqid\_ds buf2; /\*用于删除消息队列\*/

struct msg\_form msg; /\*消息队列用于通知对方更新了共享内存\*/

// 获取key值

if((key = ftok(".", 'z')) < 0)

{

perror("ftok error");

exit(1);

}

// 创建共享内存

if((shmid = shmget(key, 1024, IPC\_CREAT|0666)) == -1)

{

perror("Create Shared Memory Error");

exit(1);

}

// 连接共享内存

shm = (char\*)shmat(shmid, 0, 0);

if((int)shm == -1)

{

perror("Attach Shared Memory Error");

exit(1);

}

// 创建消息队列

if ((msqid = msgget(key, IPC\_CREAT|0777)) == -1)

{

perror("msgget error");

exit(1);

}

// 创建信号量

semid = creat\_sem(key);

// 读数据

while(1)

{

msgrcv(msqid, &msg, 1, 888, 0); /\*读取类型为888的消息\*/

if(msg.mtext == 'q') /\*quit - 跳出循环\*/

break;

if(msg.mtext == 'r') /\*read - 读共享内存\*/

{

sem\_p(semid);

printf("%s\n",shm);

sem\_v(semid);

}

}

// 断开连接

shmdt(shm);

/\*删除共享内存、消息队列、信号量\*/

shmctl(shmid, IPC\_RMID, &buf1);

msgctl(msqid, IPC\_RMID, &buf2);

del\_sem(semid);

return 0;

}

client.c

#include<stdio.h>

#include<stdlib.h>

#include<sys/shm.h> // shared memory

#include<sys/sem.h> // semaphore

#include<sys/msg.h> // message queue

#include<string.h> // memcpy

// 消息队列结构

struct msg\_form {

long mtype;

char mtext;

};

// 联合体，用于semctl初始化

union semun

{

int val; /\*for SETVAL\*/

struct semid\_ds \*buf;

unsigned short \*array;

};

// P操作:

// 若信号量值为1，获取资源并将信号量值-1

// 若信号量值为0，进程挂起等待

int sem\_p(int sem\_id)

{

struct sembuf sbuf;

sbuf.sem\_num = 0; /\*序号\*/

sbuf.sem\_op = -1; /\*P操作\*/

sbuf.sem\_flg = SEM\_UNDO;

if(semop(sem\_id, &sbuf, 1) == -1)

{

perror("P operation Error");

return -1;

}

return 0;

}

// V操作：

// 释放资源并将信号量值+1

// 如果有进程正在挂起等待，则唤醒它们

int sem\_v(int sem\_id)

{

struct sembuf sbuf;

sbuf.sem\_num = 0; /\*序号\*/

sbuf.sem\_op = 1; /\*V操作\*/

sbuf.sem\_flg = SEM\_UNDO;

if(semop(sem\_id, &sbuf, 1) == -1)

{

perror("V operation Error");

return -1;

}

return 0;

}

int main()

{

key\_t key;

int shmid, semid, msqid;

char \*shm;

struct msg\_form msg;

int flag = 1; /\*while循环条件\*/

// 获取key值

if((key = ftok(".", 'z')) < 0)

{

perror("ftok error");

exit(1);

}

// 获取共享内存

if((shmid = shmget(key, 1024, 0)) == -1)

{

perror("shmget error");

exit(1);

}

// 连接共享内存

shm = (char\*)shmat(shmid, 0, 0);

if((int)shm == -1)

{

perror("Attach Shared Memory Error");

exit(1);

}

// 创建消息队列

if ((msqid = msgget(key, 0)) == -1)

{

perror("msgget error");

exit(1);

}

// 获取信号量

if((semid = semget(key, 0, 0)) == -1)

{

perror("semget error");

exit(1);

}

// 写数据

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\* IPC \*\n");

printf("\* Input r to send data to server. \*\n");

printf("\* Input q to quit. \*\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

while(flag)

{

char c;

printf("Please input command: ");

scanf("%c", &c);

switch(c)

{

case 'r':

printf("Data to send: ");

sem\_p(semid); /\*访问资源\*/

scanf("%s", shm);

sem\_v(semid); /\*释放资源\*/

/\*清空标准输入缓冲区\*/

while((c=getchar())!='\n' && c!=EOF);

msg.mtype = 888;

msg.mtext = 'r'; /\*发送消息通知服务器读数据\*/

msgsnd(msqid, &msg, sizeof(msg.mtext), 0);

break;

case 'q':

msg.mtype = 888;

msg.mtext = 'q';

msgsnd(msqid, &msg, sizeof(msg.mtext), 0);

flag = 0;

break;

default:

printf("Wrong input!\n");

/\*清空标准输入缓冲区\*/

while((c=getchar())!='\n' && c!=EOF);

}

}

// 断开连接

shmdt(shm);

return 0;

}

五种通讯方式总结

管道：速度慢，容量有限，只有父子进程能通讯

FIFO：任何进程间都能通讯，但速度慢

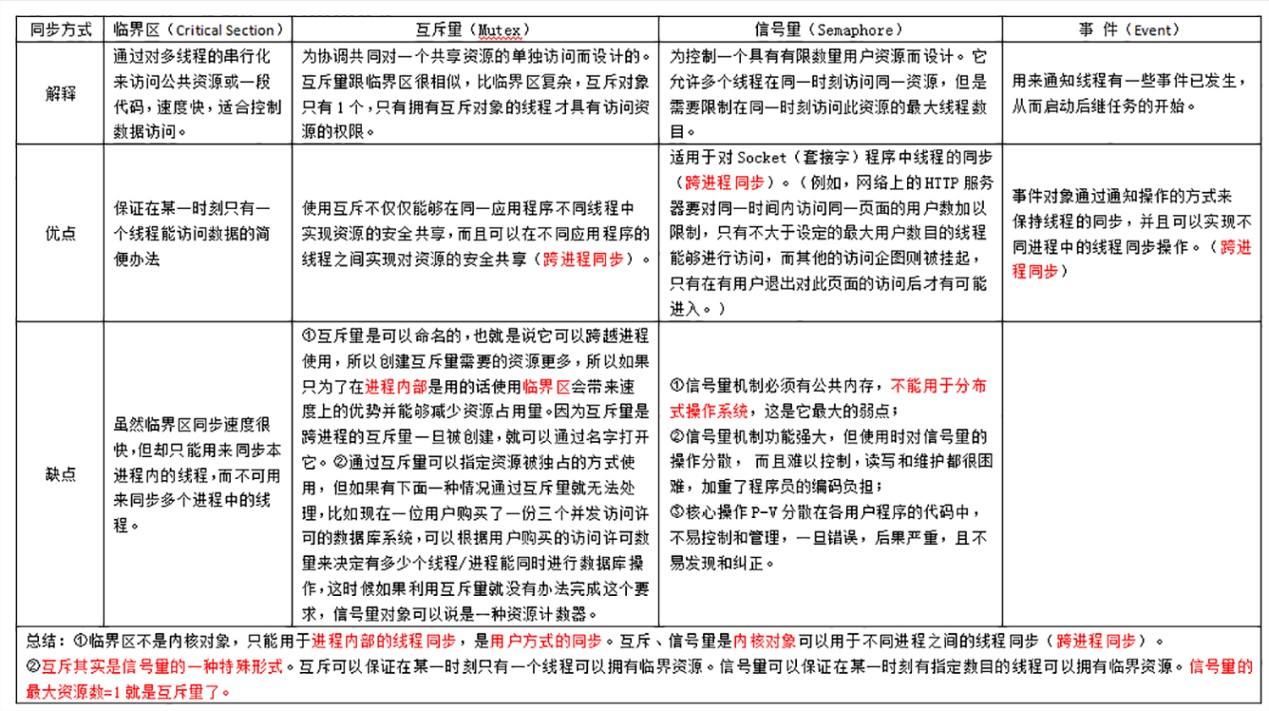
消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题

信号量：不能传递复杂消息，只能用来同步

共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

详情见：<http://www.cnblogs.com/zgq0/p/8780893.html>。

### 怎样实现进程同步



进程同步的四种方法

临界区（Critical Section）:通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。

优点：保证在某一时刻只有一个线程能访问数据的简便办法

缺点：虽然临界区同步速度很快，但却只能用来同步本进程内的线程，而不可用来同步多个进程中的线程。

互斥量（Mutex）:为协调共同对一个共享资源的单独访问而设计的。

互斥量跟临界区很相似，比临界区复杂，互斥对象只有一个，只有拥有互斥对象的线程才具有访问资源的权限。

优点：使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。

缺点：

互斥量是可以命名的，也就是说它可以跨越进程使用，所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。

通过互斥量可以指定资源被独占的方式使用，但如果有下面一种情况通过互斥量就无法处理，比如现在一位用户购买了一份三个并发访问许可的数据库系统，可以根据用户购买的访问许可数量来决定有多少个线程/进程能同时进行数据库操作，这时候如果利用互斥量就没有办法完成这个要求，信号量对象可以说是一种资源计数器。

信号量（Semaphore）:为控制一个具有有限数量用户资源而设计。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。互斥量是信号量的一种特殊情况，当信号量的最大资源数=1就是互斥量了。

优点：适用于对Socket（套接字）程序中线程的同步。（例如，网络上的HTTP服务器要对同一时间内访问同一页面的用户数加以限制，只有不大于设定的最大用户数目的线程能够进行访问，而其他的访问企图则被挂起，只有在有用户退出对此页面的访问后才有可能进入。）

缺点：

信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱点；

信号量机制功能强大，但使用时对信号量的操作分散， 而且难以控制，读写和维护都很困难，加重了程序员的编码负担；

核心操作P-V分散在各用户程序的代码中，不易控制和管理，一旦错误，后果严重，且不易发现和纠正。

事件（Event）: 用来通知线程有一些事件已发生，从而启动后继任务的开始。

优点：事件对象通过通知操作的方式来保持线程的同步，并且可以实现不同进程中的线程同步操作。

缺点：

临界区不是内核对象，只能用于进程内部的线程同步，是用户方式的同步。互斥、信号量是内核对象可以用于不同进程之间的线程同步（跨进程同步）。

互斥其实是信号量的一种特殊形式。互斥可以保证在某一时刻只有一个线程可以拥有临界资源。信号量可以保证在某一时刻有指定数目的线程可以拥有临界资源。

详情见：<https://blog.csdn.net/qq_38211852/article/details/80211169>。

### fork()返回值

fork()之前，只有一个进程在执行这段代码，但是在该条语句后，就变成了两个进程在执行了，这两个进程代码部分完全相同。Fork仅仅被调用一次，但却能够返回两次，它有三种不同的返回值：1.父进程中，fork返回新创建的子进程ID。2.子进程中，fork返回0。3.出现错误，fork返回负值。

详情见：<https://blog.csdn.net/lovenankai/article/details/6874475>。

### LT（水平触发）和ET（边缘触发）-待加

详情见：<https://blog.csdn.net/misszhoudandan/article/details/81173046>。

总结非常完美。

### epoll与select/poll区别-待加

详情见：<https://blog.csdn.net/xsf50717/article/details/47282545>。

### IO模型-待加

种类：阻塞IO模型、非阻塞IO模型、IO复用模型(select、poll、epoll)、信号驱动IO模型、异步IO模型。

IO多路复用模型,select、poll、epoll调用过程和优缺点。

详情见：<https://www.jianshu.com/p/6a6845464770>。

### 生产者、消费者模式问题-待加

### 单例模式-代码-待加

### 死锁

定义

死锁是指多个进程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。例如，在某一个计算机系统中只有一台打印机和一台输入 设备，进程P1正占用输入设备，同时又提出使用打印机的请求，但此时打印机正被进程P2 所占用，而P2在未释放打印机之前，又提出请求使用正被P1占用着的输入设备。这样两个进程相互无休止地等待下去，均无法继续执行，此时两个进程陷入死锁状态。

产生原因

系统资源的竞争

系统资源的竞争导致系统资源不足，以及资源分配不当，导致死锁。

 进程运行推进顺序不合适

进程在运行过程中，请求和释放资源的顺序不当，会导致死锁。

死锁的四个必要条件

互斥条件：进程对于所分配到的资源具有排它性，即一个资源只能被一个进程占用，直到被该进程释放   
请求和保持条件：一个进程因请求被占用资源而发生阻塞时，对已获得的资源保持不放。   
不剥夺条件：任何一个资源在没被该进程释放之前，任何其他进程都无法对他剥夺占用   
循环等待条件：当发生死锁时，所等待的进程必定会形成一个环路（类似于死循环），造成永久阻塞。

避免死锁：

[加锁顺序](http://ifeve.com/deadlock-prevention/#ordering)：当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。

[加锁时限](http://ifeve.com/deadlock-prevention/#timeout)：另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。这段随机的等待时间让其它线程有机会尝试获取相同的这些锁，并且让该应用在没有获得锁的时候可以继续运行

[死锁检测](http://ifeve.com/deadlock-prevention/#detection)：死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

详情见：<https://www.cnblogs.com/hadoop-dev/p/6899171.html>。

<https://www.cnblogs.com/bopo/p/9228834.html>。

### 如何降低锁的竞争

方法：1.减少锁的持有时间。2.降低锁的请求频率。3.使用一些协调机制。

详情见：<https://segmentfault.com/a/1190000013652087>。

### 并发和并行

并发：指应用能够交替执行不同的任务。

并行：指引用能够同时执行不同的任务。

### Reacktor模式

# 场景题

设计一个抽奖系统，每个人抽到的都不是自己的。

设计一个高并发定时执行任务：实现一个方法，接受一个任务以及它要开始执行的时间，定时执行这些任务。

TCP：拔掉网线后连接是否还存在，为什么？（类似的心跳检测机制）

操作系统如何识别TCP连接

TCP半连接，客户端如何接收消息

千万级数据，设计数据结构，同时支持先入先出，后入后出？

给定10G数据，内存只有几十M，怎样将这些数据进行排序（分治+外部排序）

设计一个线程池

服务器大量请求如何同步。

session校检服务器该如何设计

对于线程A和线程B，要求两个线程交替输出ABABAB，有几种实现方法。

设计一个处理高并发请求的系统

3TB IP数据，怎么统计个IP出现次数？

有1000亿条数据，每条数据包括url，ip，time三个字段，现在搜索某个时间段的某个URL访问的IP，怎么设计。

访问量统计系统设计

2TB数据，都是单词，输入一个单词，输出出现的次数。

定时触发系统怎么设计更高效。

微信如何设计

高并发场景，如何防止商品超卖

有三个线程都返回一个list，把这三个list做汇总。

秒杀系统设计。

详情见：<https://www.nowcoder.com/discuss/148246?type=2&order=3&pos=230&page=1>。

如何判断2个文件里的内容是否一致，内容是4000万条的订单号、金额及交易状态。

浏览器中输入一个网址，回车后发生了什么（一次完成的http请求）

详情见：<https://blog.csdn.net/baoendemao/article/details/40659229>。

扫码登录整个流程

详情见：<https://blog.csdn.net/huangshulang1234/article/details/79840047>。

手写strstr、strcpy

有一个数据结构，存用户和得分，想快速得到300到500分的所有用户（去检查找），用什么数据机构

十台主机，A主机一个文件，想要同步到其他九台主机，如何实现？如何优化？

给几百万个网址，如何高效找出特定网址是否在其中（布隆过滤器）

1G内存，4Gurl，求重复的url

多个用户线上共同编辑一个文件

一万个对象，一百个线程，如何做到高并发访问

如何实现分布式集群的每个url的访问量统计

一个网站有成千上万的访问量，如何处理请求（代理）

两个文件各有100亿个URL，如何找到两个文件中相同的URL

微信获取权限的协议

大文件分配处理1G 1T 100G数据的统计处理

1GB的文件中包含了以换行分割的10000以下的数字，给一个4核的CPU，设计方案最快的对所有数字进行求和。

淘宝不用遍历查找1000-2000的商品

一个进程有10个线程，如果down掉一个线程，会不会对其他的有影响

微信发送一张图片，然后都做了什么

内存超过一个G，如何快速读取

# 手撕代码

跳台阶（递归和动态规划解法）

详情见：<https://www.cnblogs.com/evenleee/p/8474465.html>。注意跳台阶拓展。

括号匹配（（）、[]、{}）

利用栈来实现。

在翻转的排好序的数组（123456789——>891234567）中找到最小值小标

二分查找加特殊情况判断。（时间复杂度logn）

<https://blog.csdn.net/u012243115/article/details/41923667>。

字符串去驼峰处理（分析时间复杂度）：例如aba输出b，abcba输出c。

回文串

回文串

<https://www.cnblogs.com/Kobe10/p/6365676.html>。

手写层次遍历二叉树（DFS和BFS）

DFS递归，使用数组保存，BFS用队列实现

手写Z字形遍历二叉树（层与层之间交替）

Leetcode103题

二叉树的三种遍历

<https://www.cnblogs.com/joyeehe/p/7878698.html>。

100W个数中求最大的1000个

堆排序（构建最大堆、交换元素、调整堆）

二叉树最远的两个节点距离

<http://www.cnblogs.com/miloyip/archive/2010/02/25/1673114.html>。（分情况讨论，递归）

堆中寻找中位数

构建最大堆和最小堆<https://blog.csdn.net/lidalong0408/article/details/7671562>。

二叉树和为X的序列（根节点到叶子结点）

递归

手写快排

挖坑法

设计一个可以满足高效率获取第k大和前k个大的元素的数据结构  
堆实现？

找出一个从小到大排序数组中小于0最大数的索引和大于0最小数的索引，如果数据没排序怎么找。

二分查找。索引数组

手撕HashMap（important）（开链法解决hash冲突）

<https://www.cnblogs.com/myd620/p/6349552.html>。

单例模式实现（important）（static实现）

<https://www.cnblogs.com/CheeseZH/p/5264519.html>。

Hash数据结构设计---重点关注

<https://www.cnblogs.com/peformer/p/8044371.html>。

给一个数组，能不能分成两个数组，它们平均值相等，能的话返回两个数组。（不会做）

Leetcode805.

写程序判断大小端(union法和直接读取法)

<https://blog.csdn.net/kit_9875507/article/details/44264663>。

输入一个数组，取一些相邻数出来，使和最大

Leetcode53

给定10G数据，内存只有几个M，怎么讲这些数据进行排序（分治+外部排序）

<https://www.cnblogs.com/codeMedita/p/7425291.html>。

第一步：（分治+内部排序）

1、采用适当的内部排序方法对输入文件的每个片段进行排序，将排好序的片段（成为归并段）写到外部存储器中（通常由一个可用的磁盘作为临时缓冲区），这样临时缓冲区中的每个归并段的内容是有序的。

2、利用归并算法，归并第一阶段生成的归并段，直到只剩下一个归并段为止。

第二步：（外部排序）

1、将内存空间划分为三份，每份大小250个记录，其中两个用作输入缓冲区，另外一个用作输出缓冲区。首先对Segment\_1和Segment\_2进行归并，先从每个归并段中读取250个记录到输入缓冲区，对其归并，归并结果放到输出缓冲区，当输出缓冲区满后，将其写道临时缓冲区内，如果某个输入缓冲区空了，则从相应的归并段中再读取250个记录进行继续归并，反复以上步骤，直至Segment\_1和Segment\_2全都排好序，形成一个大小为1500的记录，然后对Segment\_3和Segment\_4、Segment\_5和Segment\_6进行同样的操作。

2、对归并好的大小为1500的记录进行如同步骤1一样的操作，进行继续排序，直至最后形成大小为4500的归并段，至此，排序结束。

一个单词变成另一个单词。Leetcode127

详情见：<https://www.nowcoder.com/discuss/148246?type=2&order=3&pos=230&page=1>。

版本号比较

Leetcode165

二叉树深度、宽度

Max(左子树，右子树)，深度遍历（dfs）。

Max(每一层宽度)，宽度遍历（bfs）

二叉搜索树、双向链表互相转换

<https://blog.csdn.net/qq_37964547/article/details/80629461>。

二叉搜索树->双向链表

原先指向左子结点的指针调整为链表中指向前一个结点的指针，原先指向右子结点的指针调整为链表中指向下一个结点的指针。

双向链表->二叉搜索树

1、首先遍历链表，统计节点个数

2、首先我们通过下标找出中间的那个数作为根节点，假设下表为x

3、那么下标0~x-1的数肯定比x小，x+1~n-1的数比x大，将其分成三段

4、新建根节点root的值为下标X的数值，然后root的左孩子的值就是等于下标0~x-1的中间的下标的值，而右孩子就是x+1~n中间的下标的值，去到这一步大家都知道其实就是重复着2跟3，不停的递归下去，只要设置好递归的退出就能轻易转化好这个BST

大小端转换（先按位与与在移位，最后按位或）

<https://blog.csdn.net/cyq0318/article/details/78581646>。

LRU实现、优化leetcode146

<https://blog.csdn.net/tinyway/article/details/24536327>。

反转链表（方法1，方法2）

<https://www.jianshu.com/p/84117123f709>。

调整二叉树，要求所有节点的右子树的最大值大于左子树的最大值

用两种积木，2\*1型、1\*1型，摆满n行n列有多少种摆法。  
动态规划

矩阵中的最长递增路径长度（leetcode 329）

回溯

一个二维矩阵，从左到右是升序，从上到下是降序，找一个数是否存在于矩阵中。

时间复杂度O(n) <https://blog.csdn.net/zzuchengming/article/details/49475101>。

一个链表，假设第一个节点我们定位下标为1，第二个为2，那么下标为奇数的节点是升序排序，偶数的节点是降序排序，如何让整个链表有序？

（分离链表，合并两个有序链表）

C++实现消息队列（了解、了解、了解）

<https://blog.csdn.net/u012778714/article/details/80556628>。

m个n大小的有序数组求并集，m路归并。（重点）

<https://blog.csdn.net/caroline_wendy/article/details/39205383>。

电梯调度算法（了解、了解、了解）

<https://www.cnblogs.com/mengzhang/p/5329050.html>。

N个红白蓝三种小球，乱序排在一起，两两交换，要求从左至右排序成红白蓝。

Leetcode75

栈和队列相互实现

<https://www.cnblogs.com/fengziwei/p/8157557.html>。

用两个栈实现队列。用两个队列实现栈。

循环队列和链式队列实现（重点）

详情见：<https://www.cnblogs.com/ytz1996/p/6339162.html>。

汉诺塔算法

详情见：<https://www.cnblogs.com/wuzhenbo/p/3496054.html>。

void hannoi (int n, char A, char B, char C)　　// 把A盘里面的圆圈转移到C盘里面【A--C】。

{

if (n == 1)

{

cout << "移动圆圈" << n << "从盘" << A << "盘" << C << endl;　　//把最后一个圆环从起点盘移动到目标盘。

}

else

{

hannoi (n-1, A, C, B);　　// 把N-1个圆环从起点盘移动到（当前）没有任何圆环的过度盘；通过B、C盘在此函数调用中调用位置的互换，来实现把N-1个圆环从A盘到B盘的转移【A--B】。

cout << "移动圆圈" << n << "从盘" << A << "盘" << C << endl;

hannoi (n-1, B, A, C);　　// 把N-1个圆环从国度盘移动到目标盘（模仿1和2的操作方法来实现）；通过A、B盘在此函数调用中位置的互换，来实现N-1个圆环从B盘到C盘的转移【B--C】。

}

}

链表的环路判断以及环路的入口点

详情见：<https://www.jianshu.com/p/19d73fb32f2b>。

Leetcode 160题

最长无重复字符子串

Leeetcode3题

走象棋[m][n]大小的期盼，起始位置为（x，y）的马（走日字）要走到（a，b）最少需要几步（常见）

<https://blog.csdn.net/chengchencheng/article/details/78550755>。（不加障碍物）

<https://blog.csdn.net/y_aozhen/article/details/79749266>。（加障碍物）

给定一个二叉树，返回节点个数，递归版和非递归版。

<https://blog.csdn.net/y396397735/article/details/51148348>。

给定一个二叉树，返回叶子结点个数，递归版。

<https://www.cnblogs.com/xiaofengkang/archive/2011/06/01/2065357.html>。

10亿个32位整数（占用4G内存），如何快速找出不重复元素的个数，内存4G

<https://blog.csdn.net/kingyuan666/article/details/84584017>。

方法一：分治法

采用hash的方法，把这2.5亿个数划分到更小的文件中，从而保证每个文件的大小不超过可用内存的大小。然后对于每个小文件而言，所有的数据可以一次性被加载到内存中，因此可以使用字典或set来找到每个小文件中不重复的数。当处理完所有的文件后就可以找出这2.5亿个整数中所有的不重复的数。

方法二：位图法

对于整数相关的算法的求解，位图法是一种非常实用的算法。如果可用的内存空间超过1GB就可以使用这种方法。具体思路：假设整数占用4B（如果占用8B，那么求解思路类似，只不过需要占用更大的内存），4B也就32位，可以表示的整数的个数为2^32.由于题目中只查找不重复的数，而不关心具体数字出现的次数，因此可以分别使用2bit来表示各个数字的状态：用00表示这个数字没有出现过，01表示出现过一次，10表示出现过多次，11暂不使用。

序列中有一个书出现次数超过三分之一次，如何快速找到。

<https://blog.csdn.net/ouyangjinbin/article/details/51079569>。

1、找出数组中出现次数超过一半的元素

int majorityElement(const vector<int>& nums) {

int m;

int count = 0;

for (auto num : nums) {

if (0 == count) {

m = num;

++count;

}

else {

if (m == num)

++count;

else

--count;

}

}

return m;

}

2、找出数组中出现次数超过三分之一的元素

pair<int, int> majorityElement(const vector<int>& nums) {

int m, n; //最多存在2个出现次数超过 1/3 的元素

int cm, cn; //对应 m 和 n 的统计

for (auto num : nums) {

if (cm == 0 || num == m) {

m = num;

++cm;

}

else if (cn == 0 || num == n) {

n = num;

++cn;

}

else {

--cm;

--cn;

}

}

//若事先确定序列中存在超过1/3的元素，则无需验证，m一定是目标元素

//若不确定，则需要遍历一遍这个数组，统计其出现的次数，验证是否超过1/3

return make\_pair(m, n);

}

判断一棵树是否是平衡二叉树

递归

二叉树的广度优先遍历

队列

利用快排对一个单链表进行排序

快排的两种思路<https://blog.csdn.net/ricardo18/article/details/78867143>。

两个链表的公共节点

Leetcode160题

长宽为n和m的咖啡店，店里有顾客和障碍，寻找一个出发点，使得到所有顾客的距离之和最短，上下左右走（不会做）

1-7的随机数生成1-12的随机数

<https://blog.csdn.net/MDreamlove/article/details/48599107>。

最长回文子串

<https://www.cnblogs.com/mini-coconut/p/9074315.html>。（important）

100层楼扔两个鸡蛋的问题

<https://www.jianshu.com/p/bef8997e7e90>。精妙绝伦的解答

1：最笨的方法——只用一个鸡蛋遍历——N次尝试

2、二分查找——两个鸡蛋，鸡蛋A用来二分尝试，鸡蛋B用来在A摔碎后做局部遍历尝试

3、平均分割楼层法——假设总共扔X次，其中鸡蛋A扔了X1次，鸡蛋B扔了X2次

4、假设法——假设最多允许尝试X次，问能尝试到的最高的楼层。

5、动态规划法——找最优解常用方法

3n+1个数，AAABBBCCCx，如何快速找到x，设计实现3进制加法

异或

大数相加、相乘

字符串

手写memcpy（）（重要哦）

<https://www.cnblogs.com/chuanfengzhang/p/8447251.html>。

数组（平面）中包含n个点，如何找出距离某个点最近的m个点

堆排序

输入一个正整数，得到和等于这个正整数的全部正整数连续序列

<https://blog.csdn.net/chenkjiang/article/details/16967379>。

二叉树的宽度

层次遍历

一个长为x，宽为y的矩阵，四个角游动，一个台球从某点出发，不考虑摩擦力无限运动下去，求它会进入哪个洞里面

一个数组左边之和最接近右边之和的节点

先求和，再遍历

1到10000的质数（时间复杂度O（n））（important）

<https://blog.csdn.net/github_39363510/article/details/75674945>。

给一个数组，每个数都在1-n之间，不使用其他空间，找出只出现一次的所有数字

<https://www.cnblogs.com/CarrieCui/p/5119597.html>。

有序的数组，其他数都出现两次，一个数只出现一次，找出来

设计timer函数，传入时间和函数指针，到指定时间后调用函数。（设计一个定时任务模块，支持大量、不同时间的定时任务）

参见muduo库

最小堆实现

最小（最大）二叉堆实现

字符串中第一个出现一次的字符

map来做

正则表达式匹配ip地址

|  |  |  |
| --- | --- | --- |
| 匹配 | 正则表达式 | 说明 |
| 0～9 | /d | 单个数字 |
| 10～99 | [1-9]/d | 两位数 |
| 100～199 | 1/d/d | 百位为1的三位数 |
| 200～249 | 2[0-4]/d | 三位数，百位是2，十位是0～9 |
| 250～255 | 25[0-5] | 三位数，百位是2，十位是5，个位是0～5 |

<https://blog.csdn.net/asb2010/article/details/43055261>。

((25[0-5])|(2[0-4]/d)|(1/d/d)|([1-9]/d)|/d)(/.((25[0-5])|(2[0-4]/d)|(1/d/d)|([1-9]/d)|/d)){3}

洗牌算法

<https://blog.csdn.net/qq_26399665/article/details/79831490>。

给一副有序牌，设计一个打乱算法使得每张牌到每个位置的概率一样

给一个无限长链表，怎么随机抽k个数，保证每个数抽到概率一样（蓄水池抽样）

二叉树的直径（2个节点的最长距离）

而某节点的直径是左孩子直径，右孩子直径，左孩子高和右孩子高之和的最大值

找到两个字符串最长公共子串

<https://blog.csdn.net/qq_25800311/article/details/81607168>。

A和B整数集合，找出两个集合中相等的数（要求最优解决办法）；若每个集合都是亿呢

（bitmap）

<https://blog.csdn.net/jie1991liu/article/details/13168255>。

二叉树的恢复（前序遍历和中序遍历 后序遍历和中序遍历）

<https://blog.csdn.net/u014303647/article/details/88357985>。

手撕线程池的代码

参见muduo库

一个m\*n的网络，中间有个别网格标记上X不能走，问从左下角到右上角最短路径长度（如果只能向上和向右呢）

动态规划

给你一个数组，数组中第i个数是num[i]，表示从当前位置i最多能跳num[i]步,判断是否能到达最后一个数字，并且求出最小跳数。

Leetcode55题

二叉树的镜像

Leetcode226

N皇后问题

Leetcode51题

# 其他

实习做什么

实验室做什么

大气遥感，而我做的是模拟卫星微波观测资料，以及数值预报模式和资料同化同化

项目如何从0到1，项目难点

项目并发量

项目压力测试

介绍一个你曾经解决的最棘手的技术难题

讨论下研究方向，和研究方向的应用场景

github/git的使用和各种命令

UTF-8编码

分布式系统

分布式存储

redis

gdb

斐波那契数列

reactor模式

nginx服务器

# 干货帖子

<https://www.nowcoder.com/discuss/152227?type=2&order=3&pos=47&page=2>。

<https://www.nowcoder.com/discuss/147923?type=2&order=3&pos=224&page=1>。

<https://www.nowcoder.com/discuss/147801?type=2&order=3&pos=240&page=0>。

<https://www.nowcoder.com/discuss/147923?type=2&order=3&pos=233&page=1>。

<https://www.nowcoder.com/discuss/146655?type=2&order=3&pos=270&page=1>。

<https://www.nowcoder.com/discuss/90907>。

<https://www.nowcoder.com/discuss/144159?type=2&order=3&pos=333&page=1>。

<https://www.nowcoder.com/discuss/138476?type=2&order=3&pos=464&page=1>。

<https://www.nowcoder.com/discuss/132903?type=2&order=3&pos=619&page=1>。

<https://www.nowcoder.com/discuss/124724?type=2&order=3&pos=897&page=1>。

<https://www.nowcoder.com/discuss/125248?type=2&order=3&pos=869&page=2>。

<https://www.polarxiong.com/archives/%E6%88%91%E7%9A%842018%E5%B9%B4%E8%AE%A1%E7%AE%97%E6%9C%BA%E5%AE%9E%E4%B9%A0%E5%92%8C%E7%A7%8B%E6%8B%9B%E7%BB%8F%E5%8E%86.html>。

<https://www.nowcoder.com/discuss/55353>。

<https://www.nowcoder.com/discuss/122760?type=2&order=3&pos=953&page=1>。

<https://www.nowcoder.com/discuss/59394>。

<https://www.nowcoder.com/discuss/118716>。

<https://www.nowcoder.com/discuss/118716?type=2&order=3&pos=1153&page=1>。