# c++

## c++发展

开始是c，后来加上了一些面向对象的特性，c with classes

后面经拓展成为一个语言联邦

1. C
2. 面向对象的c：类，封装，继承，多态，
3. 模板
4. SLT

## union

1.1 定义

union即为联合，它是一种特殊的类。通过关键字union进行定义，一个union可以有多个数据成员。例如

union Token{

char cval;

int ival;

double dval;

};

以上代码定义了一个名为Token的联合，该联合中包含了3个数据成员。

互斥赋值，为成员指定长度

## 堆栈区别

　1、管理方式不同；

　　2、空间大小不同；

　　3、能否产生碎片不同；

　　4、生长方向不同；

　　5、分配方式不同；

　　6、分配效率不同；

管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。

　　空间大小：一般来讲在32位系统下，堆内存可以达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M（好像是，记不清楚了）。当然，我们可以修改：

　　打开工程，依次操作菜单如下：Project->Setting->Link，在Category 中选中Output，然后在Reserve中设定堆栈的最大值和commit。

　　注意：reserve最小值为4Byte；commit是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。

　　碎片问题：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构，这里我们就不再一一讨论了。

　　生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

　　分配方式：堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

　　分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

　　从这里我们可以看到，堆和栈相比，由于大量new/delete的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址，EBP和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。

　　虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。

无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候debug可是相当困难的：

## 内存管理

https://blog.csdn.net/caogenwangbaoqiang/article/details/79788368 C++内存管理（超长，例子很详细，排版很好）

在C++中，内存分成5个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

　　栈，在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

　　堆，就是那些由new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

　　自由存储区，就是那些由malloc等分配的内存块，他和堆是十分相似的，不过它是用free来结束自己的生命的。

　　全局/静态存储区，全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量又分为初始化的和未初始化的，在C++里面没有这个区分了，他们共同占用同一块内存区。

常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改。

从技术上来说，堆（heap）是C语言和操作系统的术语。堆是操作系统所维护的一块特殊内存，它提供了动态分配的功能，当运行程序调用malloc()时就会从中分配，稍后调用free可把内存交还。而自由存储是C++中通过new和delete动态分配和释放对象的抽象概念，通过new来申请的内存区域可称为自由存储区。基本上，所有的C++编译器默认使用堆来实现自由存储，也即是缺省的全局运算符new和delete也许会按照malloc和free的方式来被实现，这时藉由new运算符分配的对象，说它在堆上也对，说它在自由存储区上也正确。但程序员也可以通过重载操作符，改用其他内存来实现自由存储，例如全局变量做的对象池，这时自由存储区就区别于堆了。我们所需要记住的就是：

堆是操作系统维护的一块内存，而自由存储是C++中通过new与delete动态分配和释放对象的抽象概念。堆与自由存储区并不等价。

## extern

基本解释：extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。此外extern也可用来进行链接指定。

也就是说extern有两个作用，第一个,当它与"C"一起连用时，如: extern "C" void fun(int a, int b);则告诉编译器在编译fun这个函数名时按着C的规则去翻译相应的函数名而不是C++的，

C++的规则在翻译这个函数名时会把fun这个名字变得面目全非，可能是fun@aBc\_int\_int#%$也可能是别的，这要看编译器的"脾气"了(不同的编译器采用的方法不一样)，

为什么这么做呢，因为C++支持函数的重载啊，在这里不去过多的论述这个问题，如果你有兴趣可以去网上搜索，相信你可以得到满意的解释!

第二，当extern不与"C"在一起修饰变量或函数时，如在头文件中: extern int g\_Int; 它的作用就是声明函数或全局变量的作用范围的关键字，其声明的函数和变量可以在本模块活其他模块中使用，记住它是一个声明不是定义!也就是说B模块(编译单元)要是引用模块(编译单元)A中定义的全局变量或函数时，它只要包含A模块的头文件即可,在编译阶段，模块B虽然找不到该函数或变量，但它不会报错，它会在连接时从模块A生成的目标代码中找到此函数。

2 问题：extern 变量

　　在一个源文件里定义了一个数组：char a[6];

　　在另外一个文件里用下列语句进行了声明：extern char \*a；

　　请问，这样可以吗？

　　答案与分析：

　　1)、不可以，程序运行时会告诉你非法访问。原因在于，指向类型T的指针并不等价于类型T的数组。extern char \*a声明的是一个指针变量而不是字符数组，因此与实际的定义不同，从而造成运行时非法访问。应该将声明改为extern char a[ ]。

　　2)、例子分析如下，如果a[] = "abcd",则外部变量a=0x61626364 (abcd的ASCII码值)，\*a显然没有意义

　　显然a指向的空间（0x61626364）没有意义，易出现非法内存访问。

　　3)、这提示我们，在使用extern时候要严格对应声明时的格式，在实际编程中，这样的错误屡见不鲜。

　　4)、extern用在变量声明中常常有这样一个作用，你在\*.c文件中声明了一个全局的变量，这个全局的变量如果要被引用，就放在\*.h中并用extern来声明。

3 问题：当方面修改extern 函数原型

　　当函数提供方单方面修改函数原型时，如果使用方不知情继续沿用原来的extern申明，这样编译时编译器不会报错。但是在运行过程中，因为少了或者多了输入参数，往往会照成系统错误，这种情况应该如何解决？

　　答案与分析：

　　目前业界针对这种情况的处理没有一个很完美的方案，通常的做法是提供方在自己的xxx\_pub.h中提供对外部接口的声明，然后调用方include该头文件，从而省去extern这一步。以避免这种错误。

　　宝剑有双锋，对extern的应用，不同的场合应该选择不同的做法。

4 问题：extern “C”

　　在C++环境下使用C函数的时候，常常会出现编译器无法找到obj模块中的C函数定义，从而导致链接失败的情况，应该如何解决这种情况呢？

　　答案与分析：

　　C++语言在编译的时候为了解决函数的多态问题，会将函数名和参数联合起来生成一个中间的函数名称，而C语言则不会，因此会造成链接时找不到对应函数的情况，此时C函数就需要用extern “C”进行链接指定，这告诉编译器，请保持我的名称，不要给我生成用于链接的中间函数名。

　　下面是一个标准的写法：

//在.h文件的头上

#ifdef \_\_cplusplus

#if \_\_cplusplus

extern "C"{

　#endif

　#endif /\* \_\_cplusplus \*/

　…

　…

　//.h文件结束的地方

　#ifdef \_\_cplusplus

　#if \_\_cplusplus

}

#endif

#endif /\* \_\_cplusplus \*/

5 问题：extern 函数声明

　　常常见extern放在函数的前面成为函数声明的一部分，那么，C语言的关键字extern在函数的声明中起什么作用？

　　答案与分析：

　　如果函数的声明中带有关键字extern，仅仅是暗示这个函数可能在别的源文件里定义，没有其它作用。即下述两个函数声明没有明显的区别：

extern int f(); 和int f();

　　当然，这样的用处还是有的，就是在程序中取代include “\*.h”来声明函数，在一些复杂的项目中，我比较习惯在所有的函数声明前添加extern修饰。关于这样做的原因和利弊可见下面的这个例子：“用extern修饰的全局变量”

(1) 在test1.h中有下列声明:

#ifndef TEST1H

#define TEST1H

extern char g\_str[]; // 声明全局变量g\_str

void fun1();

#endif

(2) 在test1.cpp中

#include "test1.h"

char g\_str[] = "123456"; // 定义全局变量g\_str

void fun1() { cout << g\_str << endl; }

(3) 以上是test1模块， 它的编译和连接都可以通过,如果我们还有test2模块也想使用g\_str,只需要在原文件中引用就可以了

#include "test1.h"

void fun2() { cout << g\_str << endl; }

以上test1和test2可以同时编译连接通过，如果你感兴趣的话可以用ultraEdit打开test1.obj,你可以在里面找到"123456"这个字符串,但是你却不能在test2.obj里面找到，这是因为g\_str是整个工程的全局变量，在内存中只存在一份,test2.obj这个编译单元不需要再有一份了，不然会在连接时报告重复定义这个错误!

(4) 有些人喜欢把全局变量的声明和定义放在一起，这样可以防止忘记了定义，如把上面test1.h改为

extern char g\_str[] = "123456"; // 这个时候相当于没有extern

然后把test1.cpp中的g\_str的定义去掉,这个时候再编译连接test1和test2两个模块时，会报连接错误，这是因为你把全局变量g\_str的定义放在了头文件之后，test1.cpp这个模块包含了test1.h所以定义了一次g\_str,而test2.cpp也包含了test1.h所以再一次定义了g\_str,这个时候连接器在连接test1和test2时发现两个g\_str。如果你非要把g\_str的定义放在test1.h中的话，那么就把test2的代码中#include "test1.h"去掉 换成:

extern char g\_str[];

void fun2() { cout << g\_str << endl; }

这个时候编译器就知道g\_str是引自于外部的一个编译模块了，不会在本模块中再重复定义一个出来，但是我想说这样做非常糟糕，因为你由于无法在test2.cpp中使用#include "test1.h",那么test1.h中声明的其他函数你也无法使用了，除非也用都用extern修饰，这样的话你光声明的函数就要一大串，而且头文件的作用就是要给外部提供接口使用的，所以 请记住， 只在头文件中做声明，真理总是这么简单

6. extern 和 static

(1) extern 表明该变量在别的地方已经定义过了,在这里要使用那个变量.

(2) static 表示静态的变量，分配内存的时候, 存储在静态区,不存储在栈上面.

static 作用范围是内部连接的关系, 和extern有点相反.它和对象本身是分开存储的,extern也是分开存储的,但是extern可以被其他的对象用extern 引用,而static 不可以,只允许对象本身用它. 具体差别首先，static与extern是一对“水火不容”的家伙，也就是说extern和static不能同时修饰一个变量；其次，static修饰的全局变量声明与定义同时进行，也就是说当你在头文件中使用static声明了全局变量后，它也同时被定义了；最后，static修饰全局变量的作用域只能是本身的编译单元，也就是说它的“全局”只对本编译单元有效，其他编译单元则看不到它,如:

(1) test1.h:

#ifndef TEST1H

#define TEST1H

static char g\_str[] = "123456";

void fun1();

#endif

(2) test1.cpp:

#include "test1.h"

void fun1() { cout << g\_str << endl; }

(3) test2.cpp

#include "test1.h"

void fun2() { cout << g\_str << endl; }

以上两个编译单元可以连接成功, 当你打开test1.obj时，你可以在它里面找到字符串"123456",同时你也可以在test2.obj中找到它们，它们之所以可以连接成功而没有报重复定义的错误是因为虽然它们有相同的内容，但是存储的物理地址并不一样，就像是两个不同变量赋了相同的值一样，而这两个变量分别作用于它们各自的编译单元。 也许你比较较真，自己偷偷的跟踪调试上面的代码,结果你发现两个编译单元（test1,test2）的g\_str的内存地址相同，于是你下结论static修饰的变量也可以作用于其他模块，但是我要告诉你，那是你的编译器在欺骗你，大多数编译器都对代码都有优化功能，以达到生成的目标程序更节省内存，执行效率更高，当编译器在连接各个编译单元的时候，它会把相同内容的内存只拷贝一份，比如上面的"123456", 位于两个编译单元中的变量都是同样的内容，那么在连接的时候它在内存中就只会存在一份了，如果你把上面的代码改成下面的样子，你马上就可以拆穿编译器的谎言:

(1) test1.cpp:

#include "test1.h"

void fun1()

{

g\_str[0] = ''a'';

cout << g\_str << endl;

}

(2) test2.cpp

#include "test1.h"

void fun2() { cout << g\_str << endl; }

(3) void main() {

fun1(); // a23456

fun2(); // 123456

}

这个时候你在跟踪代码时，就会发现两个编译单元中的g\_str地址并不相同，因为你在一处修改了它，所以编译器被强行的恢复内存的原貌，在内存中存在了两份拷贝给两个模块中的变量使用。正是因为static有以上的特性，所以一般定义static全局变量时，都把它放在原文件中而不是头文件，这样就不会给其他模块造成不必要的信息污染，同样记住这个原则吧！

## const

C++ const 允许指定一个语义约束，编译器会强制实施这个约束，允许程序员告诉编译器某值是保持不变的。如果在编程中确实有某个值保持不变，就应该明确使用const，这样可以获得编译器的帮助。

const修饰指针变量时：

　　(1)只有一个const，如果const位于\*左侧，表示指针所指数据是常量，不能通过解引用修改该数据；指针本身是变量，可以指向其他的内存单元。

　　(2)只有一个const，如果const位于\*右侧，表示指针本身是常量，不能指向其他内存地址；指针所指的数据可以通过解引用修改。

　　(3)两个const，\*左右各一个，表示指针和指针所指数据都不能修改。

2.const修饰函数参数

　　传递过来的参数在函数内不可以改变，与上面修饰变量时的性质一样。

void testModifyConst(const int \_x) {

\_x=5;　　　///编译出错

}

3.const修饰成员函数

(1)const修饰的成员函数不能修改任何的成员变量(mutable修饰的变量除外)

1. const成员函数不能调用非onst成员函数，因为非const成员函数可以会修改成员变量4.const修饰函数返回值
2. (1)指针传递
3. 如果返回const data,non-const pointer，返回值也必须赋给const data,non-const pointer。因为指针指向的数据是常量不能修改。(2)值传递

如果函数返回值采用“值传递方式”，由于函数会把返回值复制到外部临时的存储单元中，加const 修饰没有任何价值。所以，对于值传递来说，加const没有太多意义。

所以：

不要把函数int GetInt(void) 写成const int GetInt(void)。不要把函数A GetA(void) 写成const A GetA(void)，其中A 为用户自定义的数据类型。

onst 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。而对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(边际效应) 。

#define是定义宏变量，它其实是在编译之前，由预处理指令把代码里面的宏变量用指定的字符串替换，它不做语法检查，而constant 则是定义含有变量类型的常量。一般说来推荐使用constant定义常量，它在编译时会做语法检查。

## static

1. 先来介绍它的第一条也是最重要的一条：隐藏。（static函数，static变量均可）当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性。如果加了static，就会对其它源文件隐藏。例如在a和msg的定义前加上static，main.c就看不到它们了。利用这一特性可以在不同的文件中定义同名函数和同名变量，而不必担心命名冲突。static可以用作函数和变量的前缀，对于函数来讲，static的作用仅限于隐藏
2. static的第二个作用是保持变量内容的持久。（static变量中的记忆功能和全局生存期）

存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存储区：全局变量和static变量，只不过和全局变量比起来，static可以控制变量的可见范围，说到底static还是用来隐藏的。虽然这种用法不常见PS：如果作为static局部变量在函数内定义，它的生存期为整个源程序，但是其作用域仍与自动变量相同，只能在定义该变量的函数内使用该变量。退出该函数后， 尽管该变量还继续存在，但不能使用它。

C++的static有两种用法：面向过程程序设计中的static和面向对象程序设计中的static。前者应用于普通变量和函数，不涉及类；后者主要说明static在类中的作用。

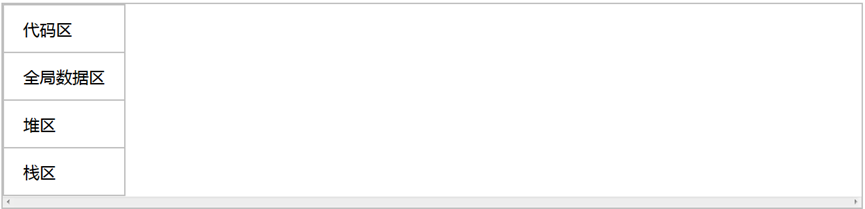
静态全局变量有以下特点：

该变量在全局数据区分配内存；

未经初始化的静态全局变量会被程序自动初始化为0（自动变量的值是随机的，除非它被显式初始化）；

静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的；

静态变量都在全局数据区分配内存，包括后面将要提到的静态局部变量。对于一个完整的程序，在内存中的分布情况如下图：



一般程序的由new产生的动态数据存放在堆区，函数内部的自动变量存放在栈区。自动变量一般会随着函数的退出而释放空间，静态数据（即使是函数内部的静态局部变量）也存放在全局数据区。全局数据区的数据并不会因为函数的退出而释放空间。

的确，定义全局变量就可以实现变量在文件中的共享，但定义静态全局变量还有以下好处：

静态全局变量不能被其它文件所用；

其它文件中可以定义相同名字的变量，不会发生冲突；

静态局部变量有以下特点：

（1）该变量在全局数据区分配内存；

（2）静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；

（3）静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为0；

（4）它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束；

3、静态函数

　　在函数的返回类型前加上static关键字,函数即被定义为静态函数。静态函数与普通函数不同，它只能在声明它的文件当中可见，不能被其它文件使用。

定义静态函数的好处：

静态函数不能被其它文件所用；

其它文件中可以定义相同名字的函数，不会发生冲突

静态数据成员有以下特点：

对于非静态数据成员，每个类对象都有自己的拷贝。而静态数据成员被当作是类的成员。无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份拷贝，由该类型的所有对象共享访问。也就是说，静态数据成员是该类的所有对象所共有的。对该类的多个对象来说，静态数据成员只分配一次内存，供所有对象共用。所以，静态数据成员的值对每个对象都是一样的，它的值可以更新；

静态数据成员存储在全局数据区。静态数据成员定义时要分配空间，所以不能在类声明中定义。在Example 5中，语句int Myclass::Sum=0;是定义静态数据成员；

静态数据成员和普通数据成员一样遵从public,protected,private访问规则；

因为静态数据成员在全局数据区分配内存，属于本类的所有对象共享，所以，它不属于特定的类对象，在没有产生类对象时其作用域就可见，即在没有产生类的实例时，我们就可以操作它；

静态数据成员初始化与一般数据成员初始化不同。静态数据成员初始化的格式为：

＜数据类型＞＜类名＞::＜静态数据成员名＞=＜值＞

类的静态数据成员有两种访问形式：

＜类对象名＞.＜静态数据成员名＞ 或 ＜类类型名＞::＜静态数据成员名＞

如果静态数据成员的访问权限允许的话（即public的成员），可在程序中，按上述格式来引用静态数据成员 ；

静态数据成员主要用在各个对象都有相同的某项属性的时候。比如对于一个存款类，每个实例的利息都是相同的。所以，应该把利息设为存款类的静态数据成员。这有两个好处，第一，不管定义多少个存款类对象，利息数据成员都共享分配在全局数据区的内存，所以节省存储空间。第二，一旦利息需要改变时，只要改变一次，则所有存款类对象的利息全改变过来了；

同全局变量相比，使用静态数据成员有两个优势：

静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其它全局名字冲突的可能性；

可以实现信息隐藏。静态数据成员可以是private成员，而全局变量不能

与静态数据成员一样，我们也可以创建一个静态成员函数，它为类的全部服务而不是为某一个类的具体对象服务。静态成员函数与静态数据成员一样，都是类的内部实现，属于类定义的一部分。普通的成员函数一般都隐含了一个this指针，this指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的。通常情况下，this是缺省的。如函数fn()实际上是this->fn()。但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有this指针。从这个意义上讲，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数。下面举个静态成员函数的例子。关于静态成员函数，可以总结为以下几点：

出现在类体外的函数定义不能指定关键字static；

静态成员之间可以相互访问，包括静态成员函数访问静态数据成员和访问静态成员函数；

非静态成员函数可以任意地访问静态成员函数和静态数据成员；

静态成员函数不能访问非静态成员函数和非静态数据成员；

由于没有this指针的额外开销，因此静态成员函数与类的全局函数相比速度上会有少许的增长；

调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指针调用静态成员函数，也可以直接使用如下格式：

＜类名＞::＜静态成员函数名＞（＜参数表＞）

调用类的静态成员函数。

## new与malloc

| **特征** | **new/delete** | **malloc/free** |
| --- | --- | --- |
| 分配内存的位置 | 自由存储区 | 堆 |
| 内存分配失败返回值 | 完整类型指针 | void\* |
| 内存分配失败返回值 | 默认抛出异常 | 返回NULL |
| 分配内存的大小 | 由编译器根据类型计算得出 | 必须显式指定字节数 |
| 处理数组 | 有处理数组的new版本new[] | 需要用户计算数组的大小后进行内存分配 |
| 已分配内存的扩充 | 无法直观地处理 | 使用realloc简单完成 |
| 是否相互调用 | 可以，看具体的operator new/delete实现 | 不可调用new |
| 分配内存时内存不足 | 客户能够指定处理函数或重新制定分配器 | 无法通过用户代码进行处理 |
| 函数重载 | 允许 | 不允许 |
| 构造函数与析构函数 | 调用 | 不调用 |

## inline

函数调用在执行时，首先要在栈中为形参和局部变量分配存储空间，然后还要将实参的值复制给形参，接下来还要将函数的返回地址（该地址指明了函数执行结束后，程序应该回到哪里继续执行）放入栈中，最后才跳转到函数内部执行。这个过程是要耗费时间的。

另外，函数执行 return 语句返回时，需要从栈中回收形参和局部变量占用的存储空间，然后从栈中取出返回地址，再跳转到该地址继续执行，这个过程也要耗费时间。

在c/c++中，为了解决一些频繁调用的小函数大量消耗栈空间（栈内存）的问题，特别的引入了inline修饰符，表示为内联函数。

当编译器处理调用内联函数的语句时，不会将该语句编译成函数调用的指令，而是直接将整个函数体的代码插人调用语句处，就像整个函数体在调用处被重写了一遍一样。

有了内联函数，就能像调用一个函数那样方便地重复使用一段代码，而不需要付出执行函数调用的额外开销。

栈空间就是指放置程序的局部数据（也就是函数内数据）的内存空间。

在系统下，栈空间是有限的，假如频繁大量的使用就会造成因栈空间不足而导致程序出错的问题，如，函数的死循环递归调用的最终结果就是导致栈内存空间枯竭

内联函数的优势在于做参数类型检查，而宏定义不会，宏只是简单的文本替换

内联函数特点：

1，C++ 编译器可以将一个函数进行内联编译；

2，被 C++ 编译器内联编译的函数叫做内联函数；

1，内联函数在 C++ 中的地位是用来替换 C 中的宏代码块；

3，C++ 编译器直接将函数体插入函数调用的地方；

4，内联函数没有普通函数调用时的额外开销（压栈，跳转，返回）；

1，C++ 编译器不一定满足函数的内敛请求；

1. 此时内联函数变为普通函数，行为同普通函数

编译器将内联函数展开到了函数的调用出，并减少了函数调用产生的效率和压栈操作，所以效率比普通函数调用要高一些

但是比起宏直接转换为立即数肯定还是要低点，毕竟内联函数展开后，代码量都比宏展开的多，很明显。而宏的话在预编译的时候就已经被展开了，在编译时期就是

当成一个普通表达式处理，而无关任何函数调用的事情

## cast

1) static\_cast

使用方法：static\_cast < type-id > ( expression )

功能：把expression转换为type-id类型，无运行时类型检查，转换存在不安全性；

使用场景：

情况1：void指针->其他类型指针

情况2：改变通常的标准转换

情况3：避免出现可能多种转换的歧义

它主要有如下几种用法：

① 用于类层次结构中基类和子类之间指针或引用的转换。进行上行转换（把子类的指针或引用转换成基类表示）是安全的；进行下行转换（把基类指针或引用转换成子类指针或引用）时，由于没有动态类型检查，所以是不安全的。

② 用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。

③ 把void指针转换成目标类型的指针(不安全!!)

④ 把任何类型的表达式转换成void类型。

注意：static\_cast不能转换掉expression的const、volitale、或者\_\_unaligned属性

---------------------

2） dynamic\_cast

使用方法：dynamic\_cast < type-id > ( expression )

说明：该运算符把expression转换成type-id类型的对象。Type-id必须是类的指针、类的引用或者void \*；如果type-id是类指针类型，那么expression也必须是一个指针，如果type-id是一个引用，那么expression也必须是一个引用。

使用场景：dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的；在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。

注意事项：

① dynamic\_cast是动态转换，只有在基类指针转换为子类指针时才有意义。

② dynamic\_cast<>需要类成为多态，即包括“虚”函数，并因此而不能成为void\*。

③ static\_cast和dynamic\_cast可以执行指针到指针的转换，或实例本身到实例本身的转换，但不能在实例和指针之间转换。static\_cast只能提供编译时的类型安全，而dynamic\_cast可以提供运行时类型安全。

3）reinpreter\_cast

使用方法：reinpreter\_cast<type-id> (expression)

说明：type-id必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。

reinterpret\_cast可以转换任意一个32bit整数，包括所有的指针和整数。可以把任何整数转成指针，也可以把任何指针转成整数，以及把指针转化为任意类型的指针，威力最为强大！但不能将非32bit的实例转成指针。总之，只要是32bit的东东，怎么转都行！

因为任何指针可以被转换到void\*，而void\*可以被向后转换到任何指针（对于static\_cast<> 和 reinterpret\_cast<>转换都可以这样做），如果没有小心处理的话错误可能发生。

---------------------

4） const\_cast

用法：const\_cast<type\_id>(expression)

使用场景：该运算符用来修改类型的const或volatile属性。除了const或volatile修饰外，type\_id和expression的类型是一样的。

注意事项：常量指针被转化成非常量指针，并且仍然指向原来的对象；常量引用被转换成非常量引用，并且仍然指向原来的对象；常量对象被转换成非常量对象。

## 虚函数

virtual关键字可以修饰普通的成员函数，也可以修饰析构函数，但并不是没有限制

virtual在函数中的使用限制

普通函数不能是虚函数，也就是说这个函数必须是某一个类的成员函数，不可以是一个全局函数，否则会导致编译错误。

静态成员函数不能是虚函数 static成员函数是和类同生共处的，他不属于任何对象，使用virtual也将导致错误。

内联函数不能是虚函数 如果修饰内联函数 如果内联函数被virtual修饰，计算机会忽略inline使它变成存粹的虚函数。

构造函数不能是虚函数，否则会出现编译错误。

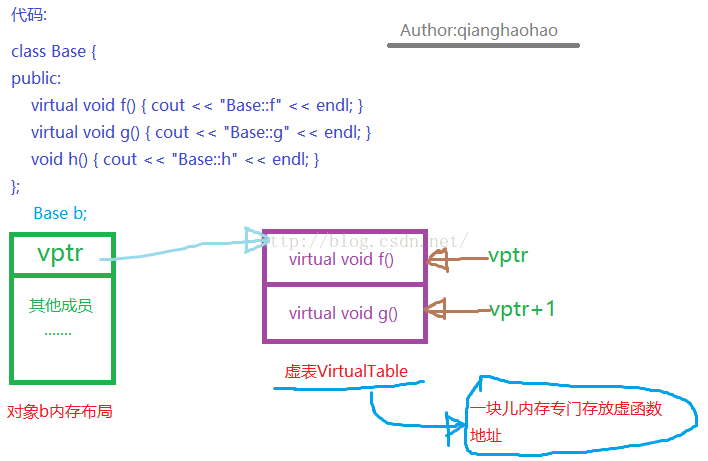
关于多态，简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”，这是一种泛型技术。所谓泛型技术，说白了就是试图使用不变的代码来实现可变的算法。比如：模板技术，RTTI技术，虚函数技术，要么是试图做到在编译时决议，要么试图做到运行时决议

虚函数表

**虚函数表**

对C++ 了解的人都应该知道虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。简称为V-Table。在这个表中，主是要一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。这样，在有虚函数的类的实例中这个表被分配在了这个实例的内存中，所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

这里我们着重看一下这张虚函数表。C++的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。



class Base {

public:

virtual void f() { cout << "Base::f" << endl; }

virtual void g() { cout << "Base::g" << endl; }

void h() { cout << "Base::h" << endl; }

};

typedef void(\*Fun)(void); //函数指针

int main()

{

Base b;

// 这里指针操作比较混乱,在此稍微解析下:

// \*\*\*\*\*printf("虚表地址:%p\n", \*(int \*)&b); 解析\*\*\*\*\*:

// 1.&b代表对象b的起始地址

// 2.(int \*)&b 强转成int \*类型,为了后面取b对象的前四个字节,前四个字节是虚表指针

// 3.\*(int \*)&b 取前四个字节,即vptr虚表地址

//

// \*\*\*\*\*printf("第一个虚函数地址:%p\n", \*(int \*)\*(int \*)&b);\*\*\*\*\*:

// 根据上面的解析我们知道\*(int \*)&b是vptr,即虚表指针.并且虚表是存放虚函数指针的

// 所以虚表中每个元素(虚函数指针)在32位编译器下是4个字节,因此(int \*)\*(int \*)&b

// 这样强转后为了后面的取四个字节.所以\*(int \*)\*(int \*)&b就是虚表的第一个元素.

// 即f()的地址.

// 那么接下来的取第二个虚函数地址也就依次类推. 始终记着vptr指向的是一块内存,

// 这块内存存放着虚函数地址,这块内存就是我们所说的虚表.

//

printf("虚表地址:%p\n", \*(int \*)&b);

printf("第一个虚函数地址:%p\n", \*(int \*)\*(int \*)&b);

printf("第二个虚函数地址:%p\n", \*((int \*)\*(int \*)(&b) + 1));

Fun pfun = (Fun)\*((int \*)\*(int \*)(&b)); //vitural f();

printf("f():%p\n", pfun);

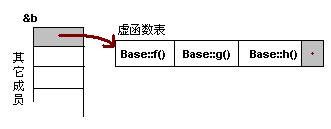
pfun();

pfun = (Fun)(\*((int \*)\*(int \*)(&b) + 1)); //vitural g();

printf("g():%p\n", pfun);

pfun();

}



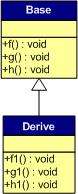
<?xml:namespace prefix = v />

*注意：在上面这个图中，我在虚函数表的最后多加了一个结点，这是虚函数表的结束结点，就像字符串的结束符“*/0*”一样，其标志了虚函数表的结束。这个结束标志的值在不同的编译器下是不同的。在*WinXP+VS2003*下，这个值是*NULL*。而在*Ubuntu 7.10 + Linux 2.6.22 + GCC 4.1.3*下，这个值是如果*1*，表示还有下一个虚函数表，如果值是*0*，表示是最后一个虚函数表。*

*下面，我将分别说明“无覆盖”和“有覆盖”时的虚函数表的样子。没有覆盖父类的虚函数是毫无意义的。我之所以要讲述没有覆盖的情况，主要目的是为了给一个对比。在比较之下，我们可以更加清楚地知道其内部的具体实现。*

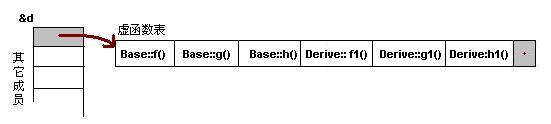
***一般继承（无虚函数覆盖）***

*下面，再让我们来看看继承时的虚函数表是什么样的。假设有如下所示的一个继承关系：*



*请注意，在这个继承关系中，子类没有重载任何父类的函数。那么，在派生类的实例中，其虚函数表如下所示*

*对于实例：*Derive d; *的虚函数表如下：*



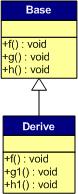
*我们可以看到下面几点：*

1）*虚函数按照其声明顺序放于表中。*

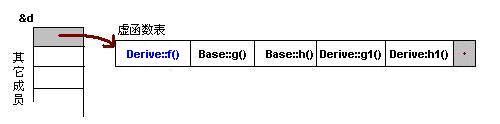
2）*父类的虚函数在子类的虚函数前面。*

***一般继承（有虚函数覆盖）***

*覆盖父类的虚函数是很显然的事情，不然，虚函数就变得毫无意义。下面，我们来看一下，如果子类中有虚函数重载了父类的虚函数，会是一个什么样子？假设，我们有下面这样的一个继承关系。*



*为了让大家看到被继承过后的效果，在这个类的设计中，我只覆盖了父类的一个函数：*f()*。那么，对于派生类的实例，其虚函数表会是下面的一个样子：*



*我们从表中可以看到下面几点，*

1）*覆盖的*f()*函数被放到了虚表中原来父类虚函数的位置。*

2）*没有被覆盖的函数依旧。*

*这样，我们就可以看到对于下面这样的程序，*

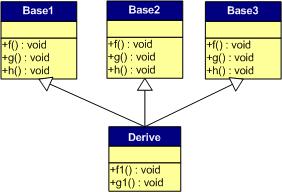
            Base \*b = new Derive();

            b->f();

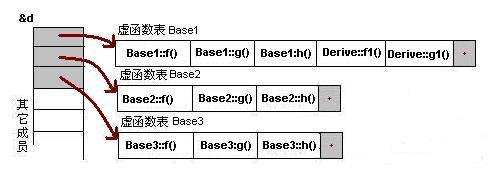
*由*b*所指的内存中的虚函数表的*f()*的位置已经被*Derive::f()*函数地址所取代，于是在实际调用发生时，是*Derive::f()*被调用了。这就实现了多态。*

***多重继承（无虚函数覆盖）***

*下面，再让我们来看看多重继承中的情况，假设有下面这样一个类的继承关系。注意：子类并没有覆盖父类的函数*



*对于子类实例中的虚函数表，是下面这个样子：*



*我们可以看到：*

1）  *每个父类都有自己的虚表。*

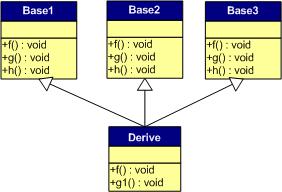
2）  *子类的成员函数被放到了第一个父类的表中。（所谓的第一个父类是按照声明顺序来判断的）*

*这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。*

***多重继承（有虚函数覆盖）***

*下面我们再来看看，如果发生虚函数覆盖的情况。*

*下图中，我们在子类中覆盖了父类的*f()*函数。*



*下面是对于子类实例中的虚函数表的图：*



*我们可以看见，三个父类虚函数表中的*f()*的位置被替换成了子类的函数指针。这样，我们就可以任一静态类型的父类来指向子类，并调用子类的*f()*了。如：*

            Derive d;

            Base1 \*b1 = &d;

            Base2 \*b2 = &d;

            Base3 \*b3 = &d;

            b1->f(); //Derive::f()

            b2->f(); //Derive::f()

            b3->f(); //Derive::f()

            b1->g(); //Base1::g()

            b2->g(); //Base2::g()

            b3->g(); //Base3::g()

一、通过父类型的指针访问子类自己的虚函数

我们知道，子类没有重载父类的虚函数是一件毫无意义的事情。因为多态也是要基于函数重载的。虽然在上面的图中我们可以看到Base1的虚表中有Derive的虚函数，但我们根本不可能使用下面的语句来调用子类的自有虚函数：

Base1 \*b1 = new Derive();

b1->f1(); //编译出错

任何妄图使用父类指针想调用子类中的未覆盖父类的成员函数的行为都会被编译器视为非法，所以，这样的程序根本无法编译通过。但在运行时，我们可以通过指针的方式访问虚函数表来达到违反C++语义的行为

***二、访问*non-public*的虚函数***

*另外，如果父类的虚函数是*private*或是*protected*的，但这些非*public*的虚函数同样会存在于虚函数表中，所以，我们同样可以使用访问虚函数表的方式来访问这些*no

n-public*的虚函数，这是很容易做到的。*

*如：*

class Base {

    private:

            virtual void f() { cout << "Base::f" << endl;

};

class Derive : public Base{

};

typedef void(\*Fun)(void);

void main() {

    Derive d;

    Fun  pFun = (Fun)\*((int\*)\*(int\*)(&d)+0);

    pFun();

}

虚函数表中为什么就能准确查找相应的函数指针呢？

因为在类设计的时候，虚函数表直接从基类也继承过来，如果派生类中成员函数覆盖了其中的某个虚函数，那么虚函数表的指针就会被替换为该成员函数，因此可以根据指针准确找到该调用哪个函数。

虚函数的作用？

虚函数用于实现多态

但是虚函数在设计上还具有封装和抽象的作用。比如抽象工厂模式。

虚函数表是针对类的还是针对对象的？同一个类的两个对象的虚函数表是怎么维护的？

C++的函数调用默认不使用动态绑定。要触发动态绑定，必须满足两个条件：

只有指定为虚函数的成员函数才能进行动态绑定

必须通过基类类型的引用或指针进行函数调用

虚函数的缺省参数是静态绑定的

因为每个派生类对象中都拥有基类部分，所以可以使用基类类型的指针或引用来引用派生类对象

编译器为每一个类维护一个虚函数表，每个对象的首地址保存着该虚函数表的指针，同一个类的不同对象实际上指向同一张虚函数表。

纯虚函数如何定义，为什么对于存在虚函数的类中析构函数要定义成虚函数

为了实现多态进行动态绑定，将派生类对象指针绑定到基类指针上，对象销毁时，如果析构函数没有定义为析构函数，则会调用基类的析构函数，显然只能销毁部分数据。如果要调用对象的析构函数，就需要将该对象的析构函数定义为虚函数，销毁时通过虚函数表找到对应的析构函数。

在很多情况下，基类中不能对虚函数给出有意义的实现，而把它们说明为纯虚函数，它的实现留给该基类的派生类去做。这就是纯虚函数的作用。

虚函数与纯虚函数的区别在于：纯虚函数是虚函数的一个子集，用于抽象类，含有纯虚函数的类就是抽象类，它不能生成对象。

定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

应该在什么情况下使用纯虚函数？

　　1、当想在基类中抽象出一个方法，且该基类只做能被继承，而不能被实例化；

　　2、这个方法必须在派生类（derived class）中被实现。

二、实现方式：

纯虚函数的声明，是在虚函数声明的结尾加 = 0，没有函数体。在派生类没有重新定义虚函数之前是不能调用的

## 多态

多态分为静态多态和动态多态。

静态多态是通过重载和模板技术实现，在编译的时候确定。

动态多态通过虚函数和继承关系来实现，执行动态绑定，在运行的时候确定。

动态多态实现有几个条件：

(1) 虚函数；

(2) 一个基类的指针或引用指向派生类的对象；

基类指针在调用成员函数(虚函数)时，就会去查找该对象的虚函数表。虚函数表的地址在每个对象的首地址。查找该虚函数表中该函数的指针进行调用。

每个对象中保存的只是一个虚函数表的指针，C++内部为每一个类维持一个虚函数表，该类的对象的都指向这同一个虚函数表。

## cpp->exe

**一个程序从源代码到可执行程序的过程**

https://blog.csdn.net/qq\_39755395/article/details/78293733

一个源程序到一个可执行程序的过程：预编译、编译、汇编、链接。   
其中，编译是主要部分，其中又分为六个部分：词法分析、语法分析、语义分析、中间代码生成、目标代码生成和优化。   
链接中，分为静态链接和动态链接，本文主要是静态链接。

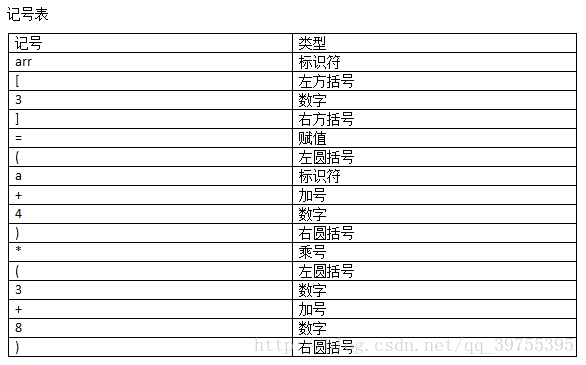
**一、预编译：主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下**   
1.删除所有的#define，展开所有的宏定义。   
2.处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。   
3.处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件。   
4.删除所有的注释，“//”和“/\*\*/”。   
5.保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重复引用。   
6.添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是能够显示行号。

**C语言的宏替换和文件包含的工作，不归入编译器的范围，而是交给独立的预处理器。**   
C语言中源代码文件的文件扩展名为.c，头文件的文件扩展名为.h，经预编译之后，生成xxx.i文件。   
在C++，源代码文件的扩展名是.cpp或.cxx，头文件的文件扩展名为.hpp，经预编译之后，生成xxx.ii文件。

**二、编译：把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。**

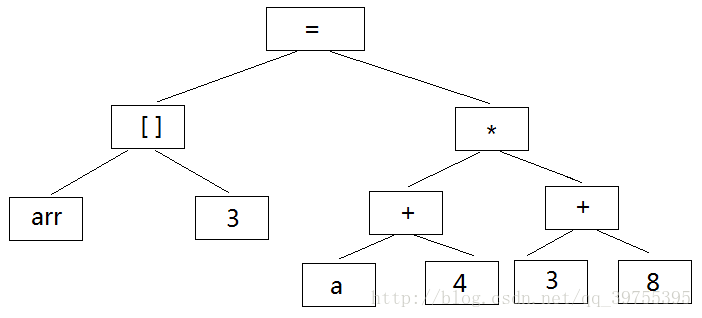
(结合程序来说明编译的几个步骤)   
有C语言的源代码如下：   
arr[3] = (a+4)\*(3+8);

**1.词法分析：**利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的***字符序列分割成一系列的记号***。   
以上的一行C语言程序，一共有16个空字符，经扫描机扫描之后，产生了16个记号。lex可以实现词法分析。见下表：



见上图：   
词法分析产生的**记号分类**有：关键字、标识符、字面量(数字、字符串)、特殊符号(加号、等号等)

**2.语法分析：**语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的***语法树是一种以表达式为节点的树***。上述的代码就是   
各种表达式的组合：赋值表达式、加法表达式、乘法表达式、数组表达式和括号表达式组成的复杂表达式。yacc可以实现语法分析，根据用户给定的规则(不同的编程语言对应不同的语法规则)对记号表进行解析。

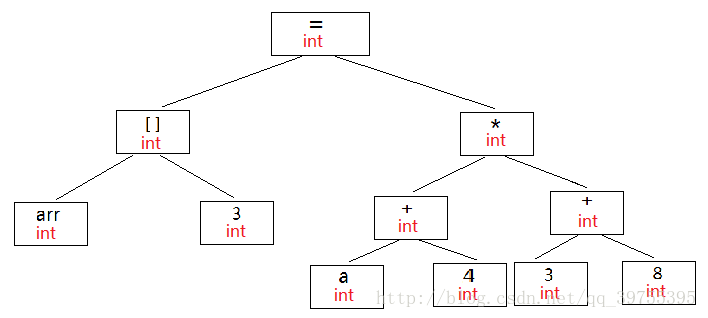


见上图：   
整个语句被看作是一个“赋值表达式”，“=”左边是一个“数组表达式”，右边是一个“乘法表达式”。数组表达式又由两个符号表达式组成，符号表达式就是最小的表达式，之后同理。

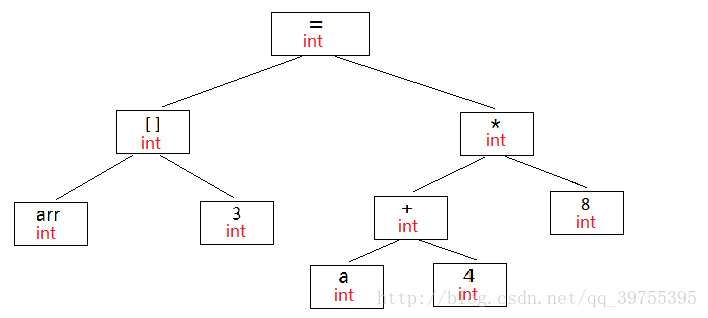
**在语法分析的同时，就把运算符的优先级确定了下来，如果出现表达式不合法，——各种括号不匹配、表达式中缺少操作，编译器就会报错。**

**3.语义分析：**语法分析器只是完成了对表达式语法层面的分析，语义分析器则对***表达式是否有意义***进行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定的语义。   
其中，***静态语义通常包括：声明和类型的匹配，类型的转换***，那么语义分析就会对这些方面进行检查，例如将一个int型赋值给int\*型时，**语义分析程序会发现这个类型不匹配，编译器就会报错。**

经过语义分析阶段之后，所有的符号都被标识了类型(如果有些类型需要做隐式转化，语义分析程序会在语法树中插入相应的转换节点)，见下图：

  
这个语句中的类型都是int型，无须做转换。

**4.优化:*\****源代码级别的一个优化过程*\**，例如该语句中的(3+8)的值可以在编译期确定，源代码优化器会将整个语法树转换成中间代码——语法树的顺序表示，十分接近目标代码。   
中间代码有很多种类型，最常见的是“***三地址码***”和“P-代码”,其中三地址码的基本形式为：***x = y op z***，表示将变量y和z进行op操作后，赋值给x，op操作可以是加减乘除等。   
经优化之后的语法树为：



该语句的三地址码：   
t1 = 3 + 8;   
t2 = a + 4;   
t3 = t2 \* t1;   
arr[3] = t3;

t1由数字11代替，省去t3，经优化或的三地址码为：   
t2 = a +4;   
t2 = t2 + 11;   
arr[3] = t2;

另一个关于中间代码的要点：***中间代码使得编译器可以被分成前端和后端***，编译器前端负责产生与机器无关的中间代码，编译器后端将中间代码转换为机器代码。   
源代码优化去产生中间代码标志着下面的过程都属于编译器后端，后端主要包括：代码生成器和目标代码优化器。

**5.目标代码生成：**由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示。

**6.目标代码优化：**目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等。

上述的六个步骤完毕之后，***编译过程***也就告一段落了。***最终产生了由汇编语言编写的目标代码***。

gcc把预编译和编译两个步骤合并成一个步骤。对于C语言的代码，是用“cc1”这个程序来完成这两步，对于C++代码，对应的程序为“cc1plus”。***gcc这个命令只是后台程序的包装***，根据不同的参数去调用：预编译编译程序——cc1，汇编器——as，连接器——ld。

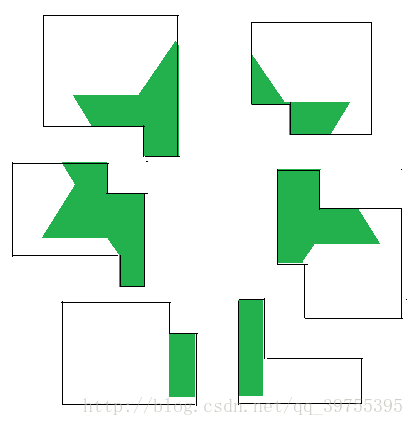
C语言的代码，经编译后产生的文件名为xxx.s。

**三、汇编：将汇编代码转变成机器可以执行的指令(机器码文件)。**   
汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过来，汇编过程有汇编器as完成。

经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Windows下)、xxx.obj(Linux下)。

但是，经过预编译、编译、汇编之后，生成机器可以执行的目标文件之后，还有***一个问题——变量a和数组arr的地址还没有确定***。这就需要链接器来搞定啦~

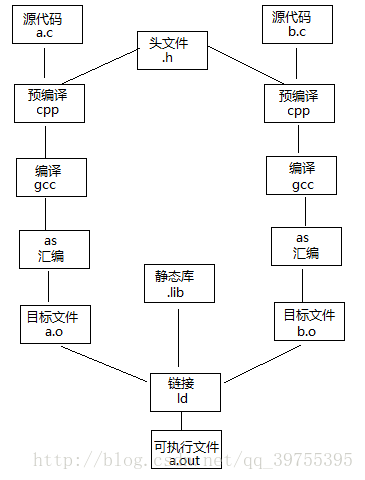
**四、链接：**   
**1、历史过程：**曾经，程序猿门在编程时，使用纸带作为最原始的存储设备，每当程序需要修改时，都要重新扎一条纸带，扎孔的表示1，不扎的是0，一串串1和0就组成了各种各样的指令——跳转等等….   
每一次的修改都非常痛苦，所以先知们就发明了汇编语言，这种编程语言方便之处在于符号的引用，表示跳转指令不再需要记住一串串0和1，终于可以使用符号——foo来表示这个动作了！   
随着汇编语言的普及，程序的代码量也就开始快速膨胀了，汇编语言说它也撑不住了….不过还好，高级编程语言Fortran、C、C++等一个接一个地问世，语言越来越方便了，追求perfect的人们就想：代码咋写更好呢？可不可以把代码按照功能的不同，分成不同的部分，便于日后的修改和重复使用呢？   
有了这个启发，程序猿们越来越得心应手，他们开始把代码按照功能和性质划分，分别形成不同的功能模块，不同的模块之间又按照各种结构来组织。   
发展到如今，软件的规模越来越大，代码动辄数百万行代码，放在一个模块那是万万不行的，维护起来会非常麻烦，所有现在的大型软件往往拥有成千上万的模块，   
模块之间相互独立又相互依赖。   
新的问题来了，一个程序被分割成这么多模块，最后要怎么把这些模块组合形成一个单一的程序?   
答案就是：***模块之间，符号的引用***！   
这就像是一张画有大树的拼图，叶子、枝干、根系都零散的分布在那些拼图碎片上，想要看到完整的大树，我们就会耐心地把那些碎片拼合在一起。



这些模块之间同样如此，它们依靠那些凸起和凹陷联系在一起，最终组合成一个完整的程序，这样的过程称为——链接。

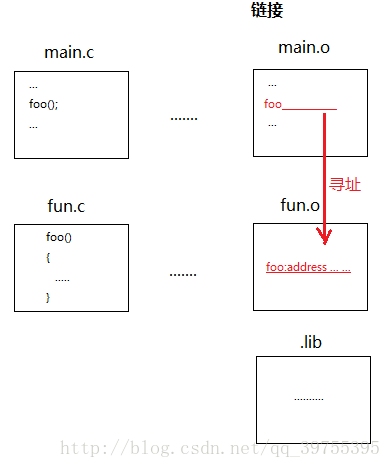
这样基于符号的模块化，使得链接过程在整个程序开发中显得十分重要和突出…..

**2、下面就静态链接，进行分析。**   
1.链接：“组装”模块的过程。   
2.链接的内容：把各个模块之间相互引用的部分都处理好，使得各个模块之间能够正确地衔接。(就像拼图，凸起和凹槽的位置一定一一对应，否则…)   
3.链接的过程：地址和空间的分配、符号决议(也叫“符号绑定”，倾向于动态链接)和重定位   
以gcc编译器为例，看基本的链接过程：



.c文件经过编译器、汇编器之后得到目标文件.o，目标文件再与库进行链接得到可执行文件.out。   
***库其实就是一组目标文件的打包***，这些目标文件中都是一些常用的代码。

我们在fun.c模块中定义了函数foo()，在main.c模块中引用了foo()函数，在编译过程当中，编译器并不知道main.c中foo()的地址，所以将调用foo()的指令的目标地址部分搁置，   
等到了链接的阶段，链接器会去找到foo()定义的那个模块，在main.o中填入正确的函数地址，这个修改地址的过程被叫做“***重定位***”，每个被修正的地方叫“重定位入口”。



以上就是一个程序从源代码到可执行程序的大致过程，这是博主根据《程序员的自我修养——链接、装载与库》来整理的，有兴趣的同学可以自己去琢磨琢磨~

## volatile

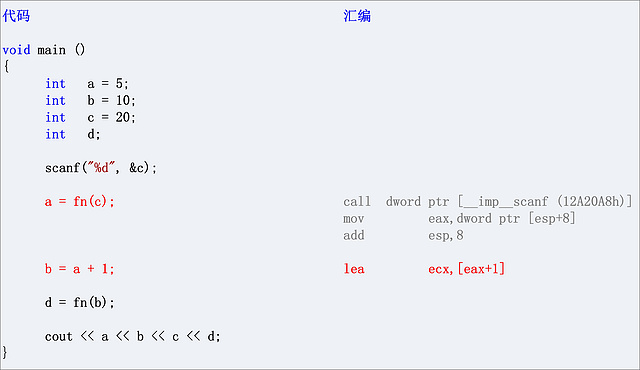
**1.易变性**

**1.1概念**

　　编译器对volatile修饰的变量，当要读取这个变量时，任何情况下都会从内存中读取，而不会从寄存器缓存中读取（因为每次都从内存中读取体现出变量的“易变”）

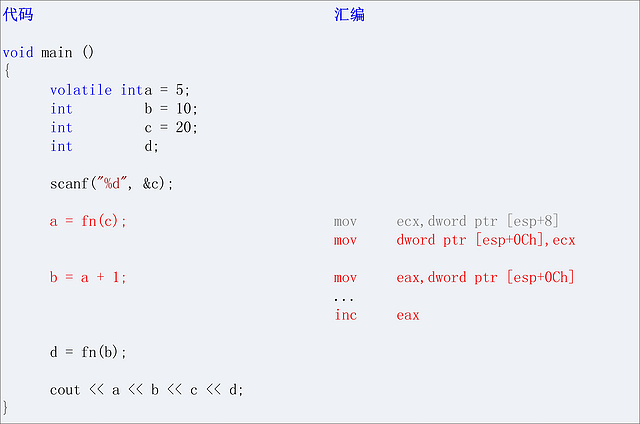
**1.2测试代码（VS 2008编译出来的Release版本）**

1）非volatile变量



b = a + 1;这条语句，对应的汇编指令是：lea ecx, [eax + 1]。由于变量a，在前一条语句a = fn(c)执行时，被缓存在了寄存器eax中，因此b = a + 1；语句，可以**直接使用仍旧在寄存器eax中的内容**，来进行计算，对应的也就是汇编：[eax + 1]

2）volatile变量



与测试用例一唯一的不同之处，是变量a被设置为volatile属性，一个小小的变化，带来的是汇编代码上很大的变化。a = fn(c)执行后，寄存器ecx中的a，被写回内存：mov dword ptr [esp+0Ch], ecx。然后，在执行b = a + 1；语句时，**变量a有重新被从内存中读取出来**：mov eax, dword ptr [esp + 0Ch]，而**不再直接使用寄存器ecx中的内容**

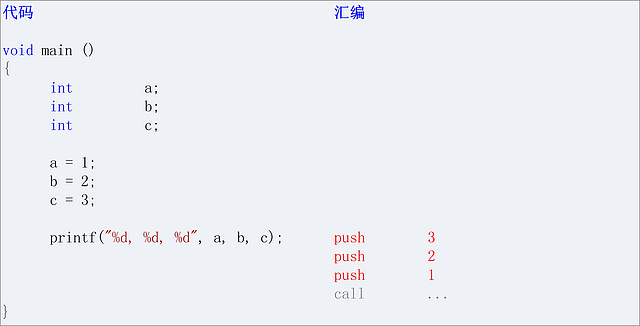
**2.不可优化性**

**2.1概念**

　　编译器不会对volatile修饰的变量进行任何优化

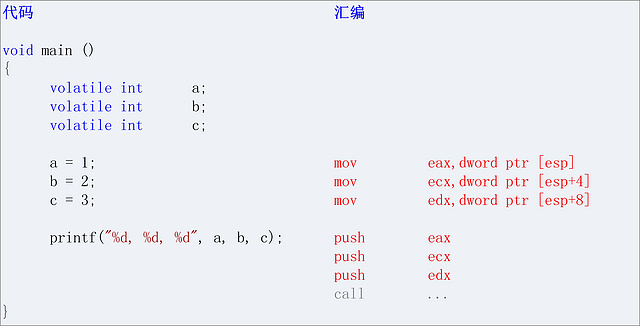
**2.2测试代码（VS 2008编译出来的Release版本）**

1）非volatile变量



在这个用例中，非volatile变量a，b，c全部被编译器优化掉了 (optimize out)，因为编译器通过分析，发觉a，b，c三个变量直接用立即数替换，不必存在内存中。最后的汇编代码相当简介，高效率

2）volatile变量



在这个用例中，a、b、c三个变量，都是volatile变量。在汇编语言中，这三个变量存到了内存中，在使用a、b、c时需要将三个变量从内存读入到寄存器之中，然后再调用printf()函数

**3.顺序性**

1）程序的乱序优化：保证一段代码的输出结果的前提下，将各条代码的实际执行顺序进行优化调整

volatile变量与volatile变量间代码的顺序，编译器不会进行乱序优化，但是volatile变量与非volatile变量代码的顺序，编译器不保证顺序，可能会进行乱序优化

在单任务的环境中，一个函数体内部，如果在两次读取变量的值之间的语句没有对变量的值进行修改，那么编译器就会设法对可执行代码进行优化。由于访问寄存器的速度要快过RAM（从RAM中读取变量的值到寄存器），以后只要变量的值没有改变，就一直从寄存器中读取变量的值，而不对RAM进行访问。

而在多任务环境中，虽然在一个函数体内部，在两次读取变量之间没有对变量的值进行修改，但是该变量仍然有可能被其他的程序（如中断程序、另外的线程等）所修改。如果这时还是从寄存器而不是从RAM中读取，就会出现被修改了的变量值不能得到及时反应的问题

多线程应用中被多个任务共享的变量。

（1）并行设备的硬件寄存器

（2）一个中断服务子程序中访问到的变量；

变更程序变量i，使在main函数中调用dosomething函数，但是，由于编译器判断在main函数里面没有修改过i，因此可能只执行一次对从i到某寄存器的读操作，然后每次if判断都只使用这个寄存器里面的“i副本”，导致dosomething永远不会被调用。如果将变量i加上volatile修饰，则编译器保证对变量i的读写操作都不会被优化，从而保证了变量i被外部程序更改后能及时在原程序中得到感知

(3)当多个线程共享某一个变量时，该变量的值会被某一个线程更改，应该用 volatile 声明。作用是防止编译器优化把变量从内存装入CPU寄存器中，当一个线程更改变量后，未及时同步到其它线程中导致程序出错。volatile的意思是让编译器每次操作该变量时一定要从内存中真正取出，而不是使用已经存在寄存器中的值。

## 内存对齐

所谓内存对齐，是为了让内存存取更有效率而采用的一种编译阶段优化内存存取的手段。

比如对于int x;（这里假设sizeof(int)==4），因为cpu对内存的读取操作是对齐的，如果x的地址不是4的倍数，那么读取这个x，需要读取两次共8个字节，然后还要将其拼接成一个int，这比存取对齐过的x要麻烦很多。

二，怎么算内存对齐大小（理论）？

对于简单类型，如int,char,float等，其对齐大小为其本身大小，即align(int) == sizeof(int)，align(char)==sizeof(char)，等等。

对于复合类型，如struct,class，其本身并无所谓对齐，因为CPU没有直接存取一个struct的指令。对于struct而言，它的对齐指的是它里面的所有成员变量都是对齐的，class同理。

下面就讲讲struct对齐是怎么回事。

首先要明白三个点：

1，内存对齐是指首地址对齐，而不是说每个变量大小对齐；

2，结构体内存对齐要求结构体内每一个成员变量都是内存对齐的；

3，结构体对齐除了第2点之外还要求结构体数组也必须是对齐的，也就是说每个相邻的结构体内部都是对齐的。

存对齐，通常也称为数据对齐，是计算机对数据类型合法地址做出了一些限制，要求某种类型对象的地址必须是某个值K(通常使2、4、8、16、32或64)的倍数。

现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

内存对齐原因：

(1). 平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

(2). 性能原因：数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

为了让处理器快速读写内存里面的数据，默认情况，编译器会把：

(1). 1个字节的变量，例如char类型的变量，放在任意地址的位置上；

(2). 2个字节的变量，例如short类型的变量，放在2的整数倍的地址上；

(3). 4个字节的变量，例如long/float类型的变量，放在4的整数倍地址上；

(4). 8个字节的变量，例如long long/uint64\_t或double类型的变量，放在8的整数倍地址上；

(5). 16个字节的变量，放在8的整数倍地址上，因为默认的对齐方式是 8。

变量在内存里面的顺序，和定义变量的顺序相同。为了符合对齐方式，就会在变量之间加入填充字节(padding)，让后面的变量放在按照对齐方式的规则的地址上。

strcut/class/union内存对齐规则：

1. 没有#pragma pack宏的对齐规则：

(1). 结构体的起始存储位置必须是能够被该结构体中最大的数据类型所整除。

(2). 每个数据成员存储的起始位置是自身大小的整数倍(比如int在32位机为4字节，则int型成员要从4的整数倍地址开始存储)。

(3). 结构体总大小(也就是sizeof的结果)，必须是该结构体成员中最大的对齐模数的整数倍。若不满足，会根据需要自动填充空缺的字节。

(4). 结构体包含另一个结构体成员，则被包含的结构体成员要从其原始结构体内部最大对齐模数的整数倍地址开始存储(比如struct a里存有struct b，b里有char,int,double等元素,那b应该从8的整数倍开始存储)。

(5). 结构体包含数组成员，比如char a[3],它的对齐方式和分别写3个char是一样的，也就是说它还是按一个字节对齐。如果写：typedef char Array[3], Array这种类型的对齐方式还是按一个字节对齐，而不是按它的长度3对齐。

(6). 结构体包含共用体成员，则该共用体成员要从其原始共用体内部最大对齐模数的整数倍地址开始存储。

2. 存在#pragma pack宏的对齐：

(1). #pragma pack (n) // 编译器将按照n个字节对齐

(2). #pragma pack () //取消自定义字节对齐方式

## 空类

等价于:

class Empty

{

public:

Empty(); //缺省构造函数

Empty(const Empty &rhs); //拷贝构造函数

~Empty(); //析构函数

Empty& operator=(const Empty &rhs); //赋值运算符

Empty\* operator&(); //取址运算符

const Empty\* operator&() const; //取址运算符(const版本)

};

使用时的调用情况:

Empty \*e = new Empty(); //缺省构造函数

delete e; //析构函数

Empty e1; //缺省构造函数

Empty e2(e1); //拷贝构造函数

e2 = e1; //赋值运算符

Empty \*pe1 = &e1; //取址运算符(非const)

const Empty \*pe2 = &e2; //取址运算符(const)

## 智能指针

### auto\_ptr

**要点**

（1）**auto\_ptr对象被拷贝或者被赋值后，已经失去了对原指针的所有权，**此时，对这个auto\_ptr的读取操作是不安全的。使用const 引用传递则可以阻止在函数体内对auto\_ptr对象的所有权转移。如果不得不使用auto\_ptr对象作为函数参数时，尽量使用const引用传递参数。

（2）**可能导致对同一块堆空间进行多次delete。**

当两个智能指针都指向同一个堆空间时，每个智能指针都会delete一下这个堆空间，这会导致未定义行为。

针对这个问题有3种策略：

1、进行深度复制，有几个指针就复制几个对象；

2、制定指针专有权的概念。即，只有一个智能指针能真正指向一个特定的对象，也只有该指针能析构这个对象所占用的空间，直到把这个指针赋给另一个指针，后一个指针才能真正指向这个对象，而前一个指针就不再起作用了，从而避免了两次delete而导致的未定义行为。这个概念比较适合auto\_ptr和unique\_ptr，但后者要求更严格；

3、记录性智能指针。即，有一个智能指针指向某对象，就把这个对象上的智能指针数加1，有一个指针不再指向该对象，就把这个对象上的智能指针数减1。只有当最后一个智能指针生命期结束了，才真正释放对象空间。

### scoped\_ptr/unique\_ptr

1.独占对象，被摧毁时，对象也被摧毁，可以通过release()放弃所有权(注意防止指针丢失)，或reset()转移

2.不支持拷贝和赋值，例外：可以从函数返回一个unique\_ptr,可以拷贝或赋值一个将要被摧毁的unique\_ptr

3.支持数组

### shared\_ptr

1.对被管理的资源进行引用计数，当一个shared\_ptr对象要共享这个资源的时候，该资源的引用计数加1，当这个对象生命期结束的时候，再把该引用技术减少1。这样当最后一个引用它的对象被释放的时候，资源的引用计数减少到0，此时释放该资源。

2.不支持数组，要自定义删除器

3.临时shared\_ptr作为函数参数传递会导致引用计数为0，产生空悬指针

4.存在循环引用问题（通过weak\_ptr解决）

循环引用：对于两个类，类A的成员中有类B的智能指针，类B的成员中有类A的智能指针，若A的对象对B中改成员进行了共享，同时B的对象赋值给了A的成员，那么此时引用计数为2，无法析构，导致内存泄漏

## struct和class

面向过程的编程认为，数据和数据操作是分开的。然而当struct进入面向对象的c++时，其特性也有了新发展，就拿上面的错误函数来说，在c++中就能运行，因为在c++中认为数据和数据对象是一个整体，不应该分开，这就是struct在c和c++两个时代的差别。

在C++中struct得到了很大的扩充：

1.struct可以包括成员函数

2.struct可以实现继承

3.struct可以实现多态

区别：

**1.默认的继承访问权。**class默认的是private,strcut默认的是public。

当然，到底默认是public继承还是private继承，取决于子类而不是基类。我的意思是，struct可以继承class，同样class也可以继承struct，那么默认的继承访问权限是看子类到底是用的struct还是class。如下：

struct A

{

int a;

};

struct B: A //共有继承

{

int b;

};

class C: A //私有继承

{

int c

};

**2.默认访问权限**：struct作为数据结构的实现体，它默认的数据访问控制是public的，而class作为对象的实现体，它默认的成员变量访问控制是private的。

**3.“class”这个关键字还用于定义模板参数，**就像“typename”。但关键字“struct”不用于定义模板参数

**4.class和struct在使用大括号{ }上的区别**

关于使用大括号初始化

1.）class和struct如果定义了构造函数的话，都不能用大括号进行初始化

　　2.）如果没有定义构造函数，struct可以用大括号初始化。

　　3.）如果没有定义构造函数，且所有成员变量全是public的话，class可以用大括号初始化

# 操作系统

## 进程线程

#### 相同点

无论是进程还是线程，对于程序员而言，都是用来实现多任务并发的技术手段。二者都可以独立调度，因此在多任务环境下，功能上并无差异。并且二者都具有各自的实体，是系统独立管理的对象个体。所以在系统层面，都可以通过技术手段实现二者的控制。而且二者所具有的状态都非常相似。而且，在多任务程序中，子进程(子线程)的调度一般与父进程(父线程)平等竞争。

其实在Linux内核2.4版以前，线程的实现和管理方式就是完全按照进程方式实现的。在2.6版内核以后才有了单独的线程实现

#### 实现差异

进程是资源分配的基本单位，线程是调度的基本单位。进程的个体间是完全独立的，而线程间是彼此依存的。多进程环境中，任何一个进程的终止，不会影响到其他进程。而多线程环境中，父线程终止，全部子线程被迫终止(没有了资源)。而任何一个子线程终止一般不会影响其他线程，除非子线程执行了exit()系统调用。任何一个子线程执行exit()，全部线程同时灭亡。

其实，也没有人写出只有线程而没有进程的程序。多线程程序中至少有一个主线程，而这个主线程其实就是有main函数的进程。它是整个程序的进程，所有线程都是它的子线程。我们通常把具有多线程的主进程称之为主线程。

#### 通信方式不同

进程间的通信方式有这样几种：

A.共享内存 B.消息队列 C.信号量 D.有名管道 E.无名管道 F.信号

G.文件 H.socket

线程间的通信方式上述进程间的方式都可沿用，且还有自己独特的几种：

A.互斥量 B.自旋锁 C.条件变量 D.读写锁 E.线程信号 G.全局变量

而且，进程间采用的通信方式要么需要切换内核上下文，要么要与外设访问(有名管道，文件)。所以速度会比较慢。而线程采用自己特有的通信方式的话，基本都在自己的进程空间内完成，不存在切换，所以通信速度会较快。也就是说，进程间与线程间分别采用的通信方式，除了种类的区别外，还有速度上的区别。

#### 父子关系差异

进程的备份关系森严，在父进程没有结束前，所有的子进程都尊从父子关系，也就是说A创建了B，则A与B是父子关系，B又创建了C，则B与C也是父子关系，A与C构成爷孙关系，也就是说C是A的孙子进程。在系统上使用pstree命令打印进程树，可以清晰看到备份关系。

多线程间的关系没有那么严格，不管是父线程还是子线程创建了新的线程，都是共享父线程的资源，所以，都可以说是父线程的子线程，也就是只存在一个父线程，其余线程都是父线程的子线程。

#### 进程池与线程池的技术实现差别

## 进程间通信

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。

以Linux中的C语言编程为例。

**一、管道**

管道，通常指无名管道，是 UNIX 系统IPC最古老的形式。

**1、特点：**

1. 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。
2. 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）。
3. 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。

**一、管道**

管道，通常指无名管道，是 UNIX 系统IPC最古老的形式。

**1、特点：**

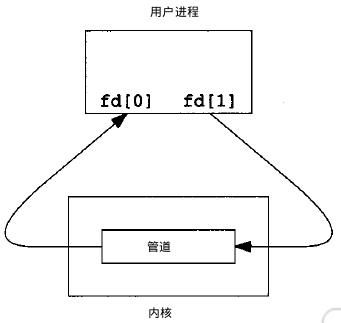
1. 它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。
2. 它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。
3. 它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

**2、原型：**

1 #include <unistd.h>

2 int pipe(int fd[2]); // 返回值：若成功返回0，失败返回-1

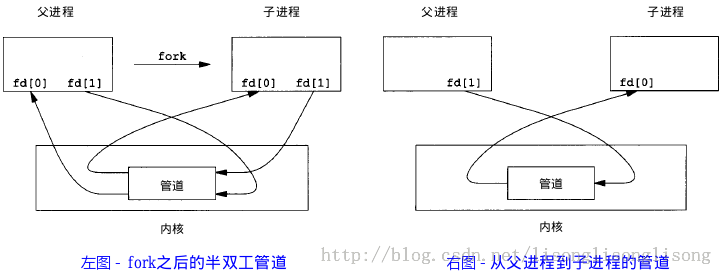
当一个管道建立时，它会创建两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。如下图：



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。

**二、FIFO**

FIFO，也称为命名管道，它是一种文件类型。

**1、特点**

1. FIFO可以在无关的进程之间交换数据，与无名管道不同。
2. 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**

1 #include <sys/stat.h>

2 // 返回值：成功返回0，出错返回-1

3 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。

**三、消息队列**

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。

**四、信号量**

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**

1. 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。
2. 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。
3. 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。
4. 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>

2 // 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

3 int semget(key\_t key, int num\_sems, int sem\_flags);

4 // 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

5 int semop(int semid, struct sembuf semoparray[], size\_t numops);

6 // 控制信号量的相关信息

7 int semctl(int semid, int sem\_num, int cmd, ...);

复制代码

复制代码

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：

复制代码

复制代码

1 struct sembuf

2 {

3 short sem\_num; // 信号量组中对应的序号，0～sem\_nums-1

4 short sem\_op; // 信号量值在一次操作中的改变量

5 short sem\_flg; // IPC\_NOWAIT, SEM\_UNDO

6 }

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

五、共享内存

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

**套接字：**这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛

## C++多线程编程

https://blog.csdn.net/ajioy/article/details/7620780

不管是辅助线程还是用户接口线程，在存取共享资源时，都需要保护共享资源，以免引起冲突，造成错误。处理方法类似于Win32 API函数的使用，但MFC为我们提供了几个同步对象C++类，即CSyncObject、CMutex、CSemaphore、CEvent、CCriticalSection。这里，CSyncObject为其它四个类的基类，后四个类分别对应前面所讲的四个Win32 API同步对象。

　　通常，我们在C++对象的成员函数中使用共享资源，或者把共享资源封装在C++类的内部。我们可将线程同步操作封装在对象类的实现函数当中，这样在应用中的线程使用C++对象时，就可以像一般对象一样使用它，简化了使用部分代码的编写，这正是面向对象编程的思想。这样编写的类被称作"线程安全类"。在设计线程安全类时，首先应根据具体情况在类中加入一个同步对象类数据成员。然后，在类的成员函数中，凡是所有修改公共数据或者读取公共数据的地方均要加入相应的同步调用。一般的处理步骤是：创建一个CSingleLock或者CMultiLock对象，然后调用其Lock函数。当对象结束时，自动在析构函数中调用Unlock函数，当然也可以在任何希望的地方调用Unlock函数。

　　如果不是在特定的C++对象中使用共享资源，而是在特定的函数中使用共享资源（这样的函数称为"线程安全函数"），那么还是按照前面介绍的办法去做：先建立同步对象，然后调用等待函数，直到可以访问资源，最后释放对同步对象的控制。

　　下面我们讨论四个同步对象分别适用的场合：

　　（1）如果某个线程必须等待某些事件发生后才能存取相应资源，则用CEvent；

　　（2）如果一个应用同时可以有多个线程存取相应资源，则用CSemaphore；

　　（3）如果有多个应用(多个进程)同时存取相应资源，则用CMutex，否则用CCriticalSection。

　　使用线程安全类或者线程安全函数进行编程，比不考虑线程安全的编程要复杂，尤其在进行调试时情况更为复杂，我们必须灵活使用Visual C++提供的调试工具，以保证共享资源的安全存取。线程安全编程的另一缺点是运行效率相对要低些，即使在单个线程运行的情况下也会损失一些效率。所以，我们在实际工作中应具体问题具体分析，以选择合适的编程方法。

         OS中的解说及详细编程使用：

      1.临界区: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。

　 2.互斥量:为协调共同对一个共享资源的单独访问而设计的。

　 3.信号量:为控制一个具有有限数量用户资源而设计。

　 4.事 件: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。

#### 临界区（Critical Section）

　　保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。

　　临界区包含两个操作原语：EnterCriticalSection（） 进入临界区LeaveCriticalSection（） 离开临界区

　　EnterCriticalSection（）语句执行后代码将进入临界区以后无论发生什么，必须确保与之匹配的LeaveCriticalSection（）都能够被执行到。否则临界区保护的共享资源将永远不会被释放。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。

　　MFC提供了很多功能完备的类，我用MFC实现了临界区。MFC为临界区提供有一个CCriticalSection类，使用该类进行线程同步处理是非常简单的。只需在线程函数中用CCriticalSection类成员函数Lock（）和UnLock（）标定出被保护代码片段即可。Lock（）后代码用到的资源自动被视为临界区内的资源被保护。UnLock后别的线程才能访问这些资源。

　　//CriticalSection

　　CCriticalSection global\_CriticalSection;

　　// 共享资源

　　char global\_Array[256];

　　//初始化共享资源

　　void InitializeArray()

　　{

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=I;

　　 }

　　}

　　//写线程

　　UINT Global\_ThreadWrite(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 //进入临界区

　　global\_CriticalSection.Lock();

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=W;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　//离开临界区

　　 global\_CriticalSection.Unlock();

　　 return 0;

　　}

　　//删除线程

　　UINT Global\_ThreadDelete(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 //进入临界区

　　 global\_CriticalSection.Lock();

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=D;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　//离开临界区

　　 global\_CriticalSection.Unlock();

　　 return 0;

　　}

　　//创建线程并启动线程

　　void CCriticalSectionsDlg::OnBnClickedButtonLock()

　　{

　　 //Start the first Thread

　　 CWinThread \*ptrWrite = AfxBeginThread(Global\_ThreadWrite,

　　 &m\_Write,

　　 THREAD\_PRIORITY\_NORMAL,

　　 0,

　　 CREATE\_SUSPENDED);

　　 ptrWrite->ResumeThread();

　　 //Start the second Thread

　　 CWinThread \*ptrDelete = AfxBeginThread(Global\_ThreadDelete,

　　 &m\_Delete,

　　 THREAD\_PRIORITY\_NORMAL,

　　 0,

　　 CREATE\_SUSPENDED);

　　 ptrDelete->ResumeThread();

　　}

　　在测试程序中，Lock UnLock两个按钮分别实现，在有临界区保护共享资源的执行状态，和没有临界区保护共享资源的执行状态。

　　程序运行结果

#### **互斥量（Mutex）**

　　互斥量跟临界区很相似，只有拥有互斥对象的线程才具有访问资源的权限，由于互斥对象只有一个，因此就决定了任何情况下此共享资源都不会同时被多个线程所访问。当前占据资源的线程在任务处理完后应将拥有的互斥对象交出，以便其他线程在获得后得以访问资源。互斥量比临界区复杂。因为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。

　　互斥量包含的几个操作原语：

　　CreateMutex（） 创建一个互斥量

　　OpenMutex（） 打开一个互斥量

　　ReleaseMutex（） 释放互斥量

　　WaitForMultipleObjects（） 等待互斥量对象

　　同样MFC为互斥量提供有一个CMutex类。使用CMutex类实现互斥量操作非常简单，但是要特别注意对CMutex的构造函数的调用

　　CMutex( BOOL bInitiallyOwn = FALSE, LPCTSTR lpszName = NULL, LPSECURITY\_ATTRIBUTES lpsaAttribute = NULL)

　　不用的参数不能乱填，乱填会出现一些意想不到的运行结果。

　　//创建互斥量

　　CMutex global\_Mutex(0,0,0);

　　// 共享资源

　　char global\_Array[256];

　　void InitializeArray()

　　{

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=I;

　　 }

　　}

　　UINT Global\_ThreadWrite(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 global\_Mutex.Lock();

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=W;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　 global\_Mutex.Unlock();

　　 return 0;

　　}

　　UINT Global\_ThreadDelete(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 global\_Mutex.Lock();

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=D;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　 global\_Mutex.Unlock();

　　 return 0;

　　}

　　同样在测试程序中，Lock UnLock两个按钮分别实现，在有互斥量保护共享资源的执行状态，和没有互斥量保护共享资源的执行状态。

　　程序运行结果

#### 信号量（Semaphores）

　　信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的PV操作相同。它指出了同时访问共享资源的线程最大数目。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。在用CreateSemaphore（）创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减1，只要当前可用资源计数是大于0的，就可以发出信号量信号。但是当前可用计数减小到0时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过ReleaseSemaphore（）函数将当前可用资源计数加1。在任何时候当前可用资源计数决不可能大于最大资源计数。

　　PV操作及信号量的概念都是由荷兰科学家E.W.Dijkstra提出的。信号量S是一个整数，S大于等于零时代表可供并发进程使用的资源实体数，但S小于零时则表示正在等待使用共享资源的进程数。

　　P操作申请资源：

　　（1）S减1；

　　（2）若S减1后仍大于等于零，则进程继续执行；

　　（3）若S减1后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转入进程调度。

　　V操作 释放资源：

　　（1）S加1；

　　（2）若相加结果大于零，则进程继续执行；

　　（3）若相加结果小于等于零，则从该信号的等待队列中唤醒一个等待进程，然后再返回原进程继续执行或转入进程调度。

　　信号量包含的几个操作原语：

　　CreateSemaphore（） 创建一个信号量

　　OpenSemaphore（） 打开一个信号量

　　ReleaseSemaphore（） 释放信号量

　　WaitForSingleObject（） 等待信号量

　　//信号量句柄

　　HANDLE global\_Semephore;

　　// 共享资源

　　char global\_Array[256];

　　void InitializeArray()

　　{

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=I;

　　 }

　　}

　　//线程1

　　UINT Global\_ThreadOne(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 //等待对共享资源请求被通过 等于 P操作

　　WaitForSingleObject(global\_Semephore, INFINITE);

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=O;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　//释放共享资源 等于 V操作

　　 ReleaseSemaphore(global\_Semephore, 1, NULL);

　　 return 0;

　　}

　　UINT Global\_ThreadTwo(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 WaitForSingleObject(global\_Semephore, INFINITE);

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=T;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　 ReleaseSemaphore(global\_Semephore, 1, NULL);

　　 return 0;

　　}

　　UINT Global\_ThreadThree(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 WaitForSingleObject(global\_Semephore, INFINITE);

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=H;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　 ReleaseSemaphore(global\_Semephore, 1, NULL);

　　 return 0;

　　}

　　void CSemaphoreDlg::OnBnClickedButtonOne()

　　{

　　//设置信号量 1 个资源 1同时只可以有一个线程访问

　　 global\_Semephore= CreateSemaphore(NULL, 1, 1, NULL);

　　 this->StartThread();

　　// TODO: Add your control notification handler code here

　　}

　　void CSemaphoreDlg::OnBnClickedButtonTwo()

　　{

　　//设置信号量 2 个资源 2 同时只可以有两个线程访问

　　 global\_Semephore= CreateSemaphore(NULL, 2, 2, NULL);

　　 this->StartThread();

　　// TODO: Add your control notification handler code here

　　}

　　void CSemaphoreDlg::OnBnClickedButtonThree()

　　{

　　//设置信号量 3 个资源 3 同时只可以有三个线程访问

　　 global\_Semephore= CreateSemaphore(NULL, 3, 3, NULL);

　　 this->StartThread();

　　// TODO: Add your control notification handler code here

　　}

　　信号量的使用特点使其更适用于对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，这时可以为每一个用户对服务器的页面请求设置一个线程，而页面则是待保护的共享资源，通过使用信号量对线程的同步作用可以确保在任一时刻无论有多少用户对某一页面进行访问，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。

　　程序运行结果

#### **事件（Event）**

　　事件对象也可以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。

　　信号量包含的几个操作原语：

　　CreateEvent（） 创建一个信号量

　　OpenEvent（） 打开一个事件

　　SetEvent（） 回置事件

　　WaitForSingleObject（） 等待一个事件

　　WaitForMultipleObjects（）

　　等待多个事件

　　WaitForMultipleObjects 函数原型：

　　WaitForMultipleObjects（

　　IN DWORD nCount, // 等待句柄数

　　IN CONST HANDLE \*lpHandles, //指向句柄数组

　　IN BOOL bWaitAll, //是否完全等待标志

　　IN DWORD dwMilliseconds //等待时间

　　）

　　参数nCount指定了要等待的内核对象的数目，存放这些内核对象的数组由lpHandles来指向。fWaitAll对指定的这nCount个内核对象的两种等待方式进行了指定，为TRUE时当所有对象都被通知时函数才会返回，为FALSE则只要其中任何一个得到通知就可以返回。dwMilliseconds在这里的作用与在WaitForSingleObject（）中的作用是完全一致的。如果等待超时，函数将返回WAIT\_TIMEOUT。

　　//事件数组

　　HANDLE global\_Events[2];

　　// 共享资源

　　char global\_Array[256];

　　void InitializeArray()

　　{

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=I;

　　 }

　　}

　　UINT Global\_ThreadOne(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=O;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　//回置事件

　　 SetEvent(global\_Events[0]);

　　 return 0;

　　}

　　UINT Global\_ThreadTwo(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=T;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　//回置事件

　　 SetEvent(global\_Events[1]);

　　 return 0;

　　}

　　UINT Global\_ThreadThree(LPVOID pParam)

　　{

　　 CEdit \*ptr=(CEdit \*)pParam;

　　 ptr->SetWindowText("");

　　//等待两个事件都被回置

　　 WaitForMultipleObjects(2, global\_Events, true, INFINITE);

　　 for(int i = 0;i<256;i++)

　　 {

　　 global\_Array[i]=H;

　　 ptr->SetWindowText(global\_Array);

　　 Sleep(10);

　　 }

　　 return 0;

　　}

　　void CEventDlg::OnBnClickedButtonStart()

　　{

　　 for (int i = 0; i < 2; i++)

　　 {

　　//实例化事件

　　 global\_Events[i]=CreateEvent(NULL,false,false,NULL);

　　 }

　　 CWinThread \*ptrOne = AfxBeginThread(Global\_ThreadOne,

　　 &m\_One,

　　 THREAD\_PRIORITY\_NORMAL,

　　 0,

　　 CREATE\_SUSPENDED);

　　 ptrOne->ResumeThread();

　　//Start the second Thread

　　 CWinThread \*ptrTwo = AfxBeginThread(Global\_ThreadTwo,

　　 &m\_Two,

　　 THREAD\_PRIORITY\_NORMAL,

　　 0,

　　 CREATE\_SUSPENDED);

　　 ptrTwo->ResumeThread();

　　//Start the Third Thread

　　 CWinThread \*ptrThree = AfxBeginThread(Global\_ThreadThree,

　　 &m\_Three,

　　 THREAD\_PRIORITY\_NORMAL,

　　 0,

　　 CREATE\_SUSPENDED);

　　 ptrThree->ResumeThread();

　　// TODO: Add your control notification handler code here

　　}

　　事件可以实现不同进程中的线程同步操作，并且可以方便的实现多个线程的优先比较等待操作，例如写多个WaitForSingleObject来代替WaitForMultipleObjects从而使编程更加灵活。

　　程序运行结果

#### 总结：

　　1． 互斥量与临界区的作用非常相似，但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。

　　2． 互斥量（Mutex），信号灯（Semaphore），事件（Event）都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用WaitForSingleObject来等待进程和线程退出。

　　3． 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，信号灯对象可以说是一种资源计数器。

————————————————

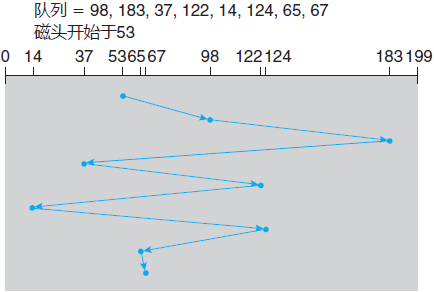
## 磁盘调度

**FCFS 调度**

磁盘调度的最简单形式当然是先来先服务（FCFS）算法。虽然这种算法比较公平，但是它通常并不提供最快的服务。  
  
例如，考虑一个磁盘队列，其 I/O 请求块的柱面的顺序如下：

98,183,37,122,14,124,65,67

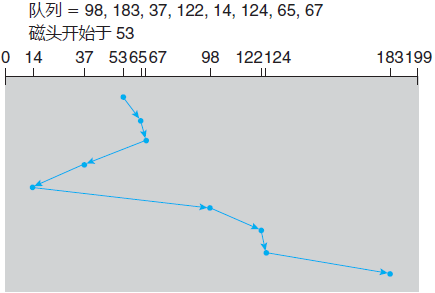
如果磁头开始位于柱面 53，那么它首先从 53 移到 98，接着再到 183、37、122、14、124、65，最后到 67，磁头移动柱面的总数为 640。这种调度如图 1 所示。

  
图 1 FCFS 磁盘调度

从 122 到 14 再到 124 的大摆动说明了这种调度的问题。如果对柱面 37 和 14 的请求一起处理，不管是在 122 和 124 之前或之后，总的磁头移动会大大减少，并且性能也会因此得以改善。

**SSTF调度**

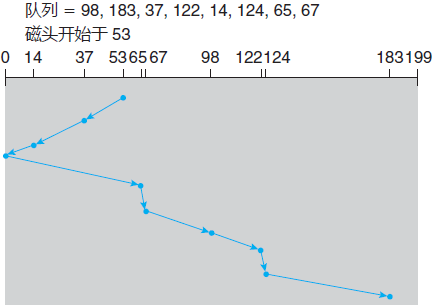
在移动磁头到别处以便处理其他请求之前，处理靠近当前磁头位置的所有请求可能较为合理。这个假设是最短寻道时间优先（SSTF）算法的基础。  
  
SSTF 算法选择处理距离当前磁头位置的最短寻道时间的请求。换句话说，SSTF 选择最接近磁头位置的待处理请求。  
  
对于上面请求队列的示例，与开始磁头位置（53）的最近请求位于柱面 65。一旦位于柱面 65，下个最近请求位于柱面 67。从那里，由于柱面 37 比 98 还要近，所以下次处理 37。如此，会处理位于柱面 14 的请求，接着 98，122，124，最后183（图 2）。

  
图 2 SSTF 磁盘调度

这种调度算法的磁头移动只有 236 个柱面，约为 FCFS 调度算法的磁头移动总数的三分之一多一点。显然，这种算法大大提高了性能。  
  
SSTF 调度本质上是一种最短作业优先（SJF）调度；与 SJF 调度一样，它可能会导致一些请求的饥饿。请记住，请求可能随时到达。假设在队列中有两个请求，分别针对柱面 14 和 186，而当处理来自 14 的请求时，另一个靠近 14 的请求来了，这个新的请求会下次处理，这样位于 186 的请求需要等待。当处理该请求时，另一个 14 附近的请求可能到达。  
  
理论上，相互接近的一些请求会连续不断地到达，这样位于 186 上的请求可能永远得不到服务。当等待处理请求队列较长时，这种情况就很可能出现了。  
  
虽然 SSTF 算法比 FCFS 算法有了相当改进，但是并非最优的。对于这个例子，还可以做得更好：移动磁头从 53 到 37（虽然 37 并不是最近的），再到 14，再到 65、67、98、122、124、183。这种策略的磁头移动的柱面总数为 208。

**SCAN 调度**

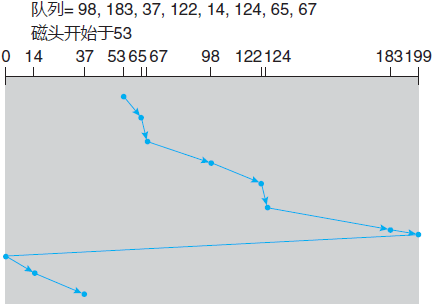
对于扫描算法，磁臂从磁盘的一端开始，向另一端移动；在移过每个柱面时，处理请求。当到达磁盘的另一端时，磁头移动方向反转，并继续处理。磁头连续来回扫描磁盘。SCAN 算法有时称为电梯算法，因为磁头的行为就像大楼里面的电梯，先处理所有向上的请求，然后再处理相反方向的请求。  
  
下面回到前面的例子来说明。在采用 SCAN 来调度柱面 98、183、37、122、14、124、65 和 67 的请求之前，除了磁头的当前位置，还需知道磁头的移动方向。

  
图 3 SCAN磁盘调度

假设磁头朝 0 移动并且磁头初始位置还是 53，磁头接下来处理 37，然后 14。在柱面 0 时，磁头会反转，移向磁盘的另一端，并处理柱面 65、67、98、122、124、183（图 3）上的请求。如果请求刚好在磁头前方加入队列，则它几乎马上就会得到服务；如果请求刚好在磁头后方加入队列，则它必须等待，直到磁头移到磁盘的另一端，反转方向，并返回。  
  
假设请求柱面的分布是均匀的，考虑当磁头移到磁盘一端并且反转方向时的请求密度。这时，紧靠磁头前方的请求相对较少，因为最近处理过这些柱面。磁盘另一端的请求密度却是最多。这些请求的等待时间也最长，那么为什么不先去那里？这就是下一个算法的想法。

**C-SCAN 调度**

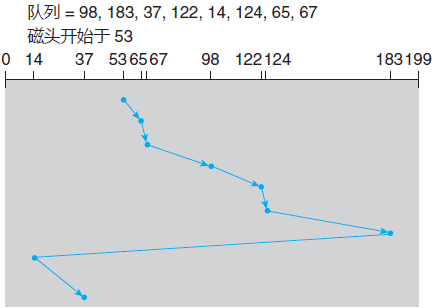
循环扫描（C-SCAN）调度是 SCAN 的一个变种，以提供更均匀的等待时间。像 SCAN 一样，C-SCAN 移动磁头从磁盘一端到磁盘另一端，并且处理行程上的请求。然而，当磁头到达另一端时，它立即返回到磁盘的开头，而并不处理任何回程上的请求（图 4）。

  
图 4 C-SCAN磁盘调度

C-SCAN 调度算法基本上将这些柱面作为一个环链，将最后柱面连到首个柱面。

**LOOK 调度**

正如以上所述，SCAN 和 C-SCAN 在磁盘的整个宽度内移动磁臂。实际上，这两种算法通常都不是按这种方式实施的。更常见的是，磁臂只需移到一个方向的最远请求为止。  
  
遵循这种模式的 SCAN 算法和 C-SCAN 算法分别称为 LOOK 和 C-LOOK 调度，因为它们在向特定方向移动时查看是否会有请求（图 5）。

  
图 5 C-LOOK 磁盘调度

**磁盘调度算法的选择**

给出了如此多的磁盘调度算法，如何选择最佳的呢？  
  
SSTF 是常见的，并且具有自然的吸引力，因为它比 FCFS 具有更好的性能。对于磁盘负荷较大的系统，SCAN 和 C-SCAN 表现更好，因为它们不太可能造成饥饿问题。对于任何特定的请求列表，可以定义最佳的执行顺序，但是计算最佳调度的所需时间可能得不到补偿。然而，对于任何调度算法，性能在很大程度上取决于请求的数量和类型。

## 虚拟内存

### 概述

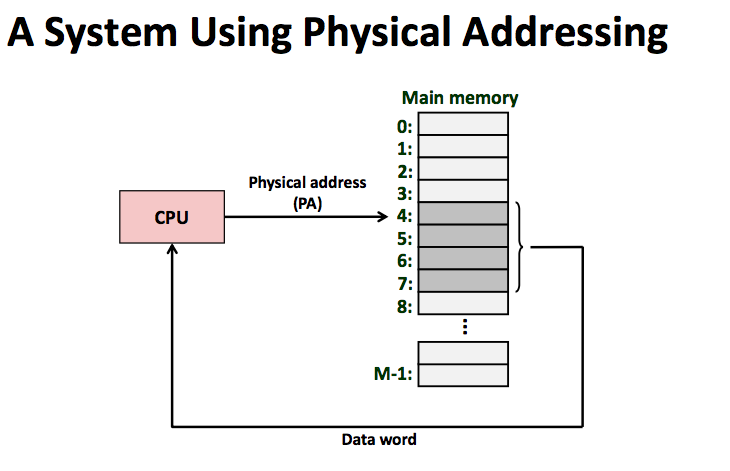
现代操作系统了提供了一种对主存的抽象概念，叫做虚拟内存。它为每个进程提供了一个非常大的，一致的和私有的地址空间。虚拟内存提供了以下的三个关键能力：

1. 它将主存看成是一个存储在磁盘空间上的地址空间的高速缓存，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送数据。
2. 它为内阁进程提供了一致的地址空间,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。

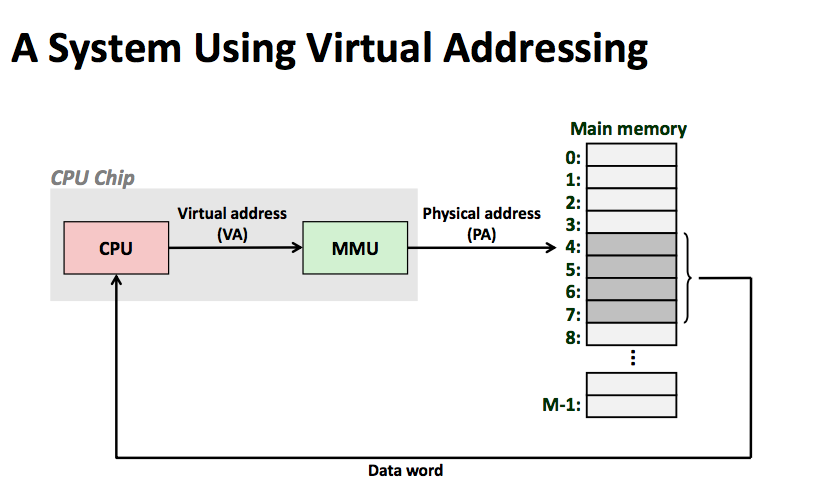
### 物理和虚拟寻址

计算机的主存可以看做是一个由 M 个连续的字节大小的单元组成的数组。每个字节都有一个唯一的物理地址（Physical Address,PA）。第一个字节的地址为 0，接下来的地址为 1，以此类推。CPU 访问内存的最简单的方式是使用物理寻址（physical addressing）

该图例的上下文是一条加载指令，塔读取从物理地址 4 处开始的 4 字节字。CPU 在执行这条指令的时候，生成一个有效物理地址，通过内存总线，把这个物理地址传递给主存，主存取出从物理地址4处开始的 4 个字节字，然后将它返回给 CPU，CPU 将它存放在一个寄存器里。早期使用物理寻址的系统通常都比较简单，比如数字信号处理器，计算器等嵌入设备

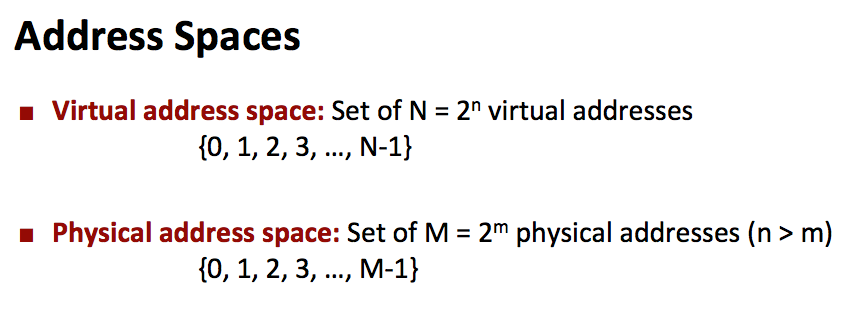


现在处理器采用的是一个程序虚拟寻址（virtual addressing）的寻址方式，如下图所示。CPU 通过生成一个虚拟地址（virtual address,VA）来访问主存，这个虚拟地址在被送到主存之前会先转换成一个物理地址。将虚拟地址转换成物理地址的任务叫做地址翻译（address translation），地址翻译需要 CPU 硬件和操作系统之间的配合。 CPU 芯片上叫做内存管理单元（Menory Management Unit, MMU）的专用硬件，利用存放在主存中的查询表来动态翻译虚拟地址，该表的内容由操作系统管理。



### 地址空间

地址空间（address space）是一个非负整数 地址 的有序合集：{0，1，2，... }



在一个带虚拟内存的系统中，CPU 从一个有 N= 2 的 n 次方 个地址的地址空间中生成虚拟地址，这个地址空间就称为虚拟地址空间：

{0，1，2，3，...., N-1}。

一个系统还有一个地理地址空间，对应于系统中物理内存的 M 个字节： {0，1，2，3，..., M-1}。

一个地址空间的大小通常是由表示最大地址所需要的位数来描述的，比如，一个包含 N = 2 的 n 次方 个地址的虚拟地址空间就叫做一个 n 位地址空间，现代操作系统通常支持 32 位或者 64 位虚拟地址空间

### 虚拟内存做为缓存的工具

从概念上来说，虚拟内存被组织成为一个由存放在磁盘上的 N 个连续的字节大小的单元组成的数组，也就是字节数组。每个字节都有一个唯一的虚拟地址作为数组的索引。磁盘上活动的数组内容被缓存在主存中。在存储器结构中，较低层次上的磁盘的数据被分割成块，这些块作为和较高层次的主存之间的传输单元。主存作为虚拟内存的缓存。

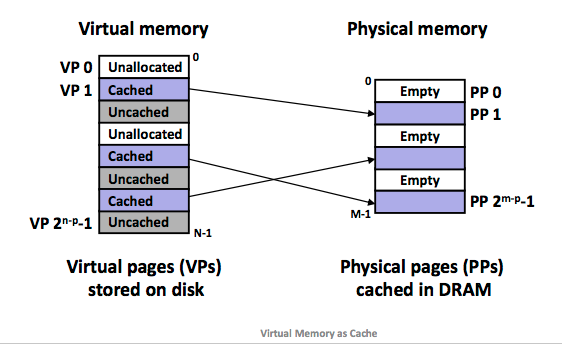
虚拟内存（VM）系统将虚拟内存分割成称为虚拟页（Virtual Page,VP）的大小固定的块，每个虚拟页的大小为 P = 2 的 p 次方 字节。同样的，物理内存被分割为物理页（Physical Page,PP）,大小也为 P 字节（物理页也称作页帧（page frame））。

在任意时刻，虚拟页面的集合都分为三个不相交的子集：

未分配的，VM 系统还未分配（或者创建）的页，未分配的页没有任何数据和它们关联，因此不占用任何内存空间。

缓存的，当前已缓存在物理内存中的已分配页。

未缓存的，未缓存在物理内存中的已分配页。



### VA 是通过什么方式转成 PA 呢 ？

同任何缓存一样，虚拟内存系统必须有某种方法来判定一个虚拟也是否缓存在 DRAM 的某个地方。如果命中缓存，那么虚拟内存系统还必须确认这个虚拟页存在哪个物理页中。如果没有命中缓存，那么虚拟内存系统必须判断虚拟页存放在磁盘的哪个位置，在物理内存中选择一个牺牲页，并将虚拟页从磁盘复制到 DRAM，替换这个牺牲页。

这些功能由软硬件联合提供，包括操作系统软件，MMU 中的地址翻译硬件和一个存放在物理内存中叫页表（page table）的数据结构，页表将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换成物理地址时都会读取页表。

### 页面置换算法

1.最佳（Optimal）置换算法

其所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面，是理想化的算法（因为我们很难预知未来要访问哪些页面），可以用来评测其他实际应用算法的好坏。

2.先进先出（FIFO）置换算法

总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面给予淘汰（由于与页面的使用规律不符，可能是性能最差的算法）。

3.最近最久未使用LRU（Least Recently Used）置换算法

选择最近最久未使用的页面予以淘汰。

4.最少使用LFU（Least Frequently Used）置换算法

选择在最近时期使用最少的页面作为淘汰页（当使用次数相同时，谁最先来淘汰谁）。

5.Clocks置换算法

# 算法

## avl和红黑树对比

AVL树（http://baike.baidu.com/view/593144.htm?fr=aladdin），又称（严格）高度平衡的二叉搜索树。其他的平衡树还有：红黑树、Treap、伸展树、SBT。

注：使用 "nil 叶子"或"空(null)叶子"，它不包含数据而只充当树在此结束的指示。这些节点在绘图中经常被省略，导致了这些树好象同上述原则相矛盾，而实际上不是这样。与此有关的结论是所有节点都有两个子节点，尽管其中的一个或两个可能是空叶子。

红黑树（http://baike.baidu.com/view/133754.htm?fr=aladdin），红黑树也是二叉查找树，统计性能要好于平衡二叉树。典型的用途是实现关联数组。因此，红黑树在很多地方都有应用。在C++ STL中，很多部分(目前包括set,multiset, map, multimap)应用了红黑树的变体(SGI STL中的红黑树有一些变化，这些修改提供了更好的性能，以及对set操作的支持)。

红黑树的关键性质: 从根到叶子的最长的可能路径不多于（<=）最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

注：最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。

红黑树与AVL树的比较：

**AVL是严格的平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多；**

红黑树是用非严格的平衡来换取增删节点时候旋转次数的降低开销；

所以简单说，如果你的应用中，搜索的次数远远大于插入和删除，那么选择AVL树，

如果搜索，插入删除次数几乎差不多，应选择红黑树。即，有时仅为了排序（建立-遍历-删除），不查找或查找次数很少，R-B树合算一些。

红黑树与AVL树的调整平衡的实现机制不同，AVL靠平衡因子和旋转，红黑树靠节点颜色以及一些约定再加上旋转。因此，存在去掉颜色的红黑树后它不是AVL树，比如左子树都是黑的，右子树都是红黑相间的，这样整个树高度2n的时候，根节点的左右层数差可以到n。

**红黑树能够以O(*log2* n) 的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构 能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。红黑树的算法时间复杂度和AVL相同，但统计性能比AVL树更高。当然，红黑树并不适应所有应用树的领域。如果数据基本上是静态的，那么让他们待在他们能够插入，并且不影响平衡的地方会具有更好的性能。如果数据完全是静态的，例如，做一个哈希表，性能可能会更好一些。*红黑树是牺牲了严格的高度平衡的优越条件*为 代价红黑树能够以O(log2 n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构 能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。红黑树的算法时间复杂度和AVL相同，但统计性能比AVL树更高.**

***AVL树在顺序插入和删除时有20%左右的性能优势，但随机性能反而落后15%左右，现实应用当然一般都是随机情况，所以红黑树得到了更广泛的应用。***

## 大数据topk

**大数据topk**

top K问题

在大规模数据处理中，经常会遇到的一类问题：在海量数据中找出出现频率最好的前k个数，或者从海量数据中找出最大的前k个数，这类问题通常被称为top K问题。例如，在搜索引擎中，统计搜索最热门的10个查询词；在歌曲库中统计下载最高的前10首歌等。

针对top K类问题，通常比较好的方案是分治+Trie树/hash+小顶堆（就是上面提到的最小堆），即先将数据集按照Hash方法分解成多个小数据集，然后使用Trie树活着Hash统计每个小数据集中的query词频，之后用小顶堆求出每个数据集中出现频率最高的前K个数，最后在所有top K中求出最终的top K。

(1)：有1亿个浮点数，如果找出期中最大的10000个？

最容易想到的方法是将数据全部排序，然后在排序后的集合中进行查找，最快的排序算法的时间复杂度一般为O（nlogn），如快速排序。但是在32位的机器上，每个float类型占4个字节，1亿个浮点数就要占用400MB的存储空间，对于一些可用内存小于400M的计算机而言，很显然是不能一次将全部数据读入内存进行排序的。其实即使内存能够满足要求（我机器内存都是8GB），该方法也并不高效，因为题目的目的是寻找出最大的10000个数即可，而排序却是将所有的元素都排序了，做了很多的无用功。

第二种方法为局部淘汰法，该方法与排序方法类似，用一个容器保存前10000个数，然后将剩余的所有数字——与容器内的最小数字相比，如果所有后续的元素都比容器内的10000个数还小，那么容器内这个10000个数就是最大10000个数。如果某一后续元素比容器内最小数字大，则删掉容器内最小元素，并将该元素插入容器，最后遍历完这1亿个数，得到的结果容器中保存的数即为最终结果了。此时的时间复杂度为O（n+m^2），其中m为容器的大小，即10000。

第三种方法是分治法，将1亿个数据分成100份，每份100万个数据，找到每份数据中最大的10000个，最后在剩下的100\*10000个数据里面找出最大的10000个。如果100万数据选择足够理想，那么可以过滤掉1亿数据里面99%的数据。100万个数据里面查找最大的10000个数据的方法如下：用快速排序的方法，将数据分为2堆，如果大的那堆个数N大于10000个，继续对大堆快速排序一次分成2堆，如果大的那堆个数N大于10000个，继续对大堆快速排序一次分成2堆，如果大堆个数N小于10000个，就在小的那堆里面快速排序一次，找第10000-n大的数字；递归以上过程，就可以找到第1w大的数。参考上面的找出第1w大数字，就可以类似的方法找到前10000大数字了。此种方法需要每次的内存空间为10^6\*4=4MB，一共需要101次这样的比较。

第四种方法是Hash法。如果这1亿个书里面有很多重复的数，先通过Hash法，把这1亿个数字去重复，这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间，然后通过分治法或最小堆法查找最大的10000个数。

第五种方法采用最小堆。首先读入前10000个数来创建大小为10000的最小堆，建堆的时间复杂度为O（mlogm）（m为数组的大小即为10000），然后遍历后续的数字，并于堆顶（最小）数字进行比较。如果比最小的数小，则继续读取后续数字；如果比堆顶数字大，则替换堆顶元素并重新调整堆为最小堆。整个过程直至1亿个数全部遍历完为止。然后按照中序遍历的方式输出当前堆中的所有10000个数字。该算法的时间复杂度为O（nmlogm），空间复杂度是10000（常数）。

实际运行：

实际上，最优的解决方案应该是最符合实际设计需求的方案，在时间应用中，可能有足够大的内存，那么直接将数据扔到内存中一次性处理即可，也可能机器有多个核，这样可以采用多线程处理整个数据集。

下面针对不容的应用场景，分析了适合相应应用场景的解决方案。

（1）单机+单核+足够大内存

如果需要查找10亿个查询次（每个占8B）中出现频率最高的10个，考虑到每个查询词占8B，则10亿个查询次所需的内存大约是10^9 \* 8B=8GB内存。如果有这么大内存，直接在内存中对查询次进行排序，顺序遍历找出10个出现频率最大的即可。这种方法简单快速，使用。然后，也可以先用HashMap求出每个词出现的频率，然后求出频率最大的10个词。

（2）单机+多核+足够大内存

这时可以直接在内存总使用Hash方法将数据划分成n个partition，每个partition交给一个线程处理，线程的处理逻辑同（1）类似，最后一个线程将结果归并。

该方法存在一个瓶颈会明显影响效率，即数据倾斜。每个线程的处理速度可能不同，快的线程需要等待慢的线程，最终的处理速度取决于慢的线程。而针对此问题，解决的方法是，将数据划分成c×n个partition（c>1），每个线程处理完当前partition后主动取下一个partition继续处理，知道所有数据处理完毕，最后由一个线程进行归并。

（3）单机+单核+受限内存

这种情况下，需要将原数据文件切割成一个一个小文件，如次啊用hash(x)%M，将原文件中的数据切割成M小文件，如果小文件仍大于内存大小，继续采用Hash的方法对数据文件进行分割，知道每个小文件小于内存大小，这样每个文件可放到内存中处理。采用（1）的方法依次处理每个小文件。

（4）多机+受限内存

这种情况，为了合理利用多台机器的资源，可将数据分发到多台机器上，每台机器采用（3）中的策略解决本地的数据。可采用hash+socket方法进行数据分发。

从实际应用的角度考虑，（1）（2）（3）（4）方案并不可行，因为在大规模数据处理环境下，作业效率并不是首要考虑的问题，算法的扩展性和容错性才是首要考虑的。算法应该具有良好的扩展性，以便数据量进一步加大（随着业务的发展，数据量加大是必然的）时，在不修改算法框架的前提下，可达到近似的线性比；算法应该具有容错性，即当前某个文件处理失败后，能自动将其交给另外一个线程继续处理，而不是从头开始处理。

top K问题很适合采用MapReduce框架解决，用户只需编写一个Map函数和两个Reduce 函数，然后提交到Hadoop（采用Mapchain和Reducechain）上即可解决该问题。具体而言，就是首先根据数据值或者把数据hash(MD5)后的值按照范围划分到不同的机器上，最好可以让数据划分后一次读入内存，这样不同的机器负责处理不同的数值范围，实际上就是Map。得到结果后，各个机器只需拿出各自出现次数最多的前N个数据，然后汇总，选出所有的数据中出现次数最多的前N个数据，这实际上就是Reduce过程。对于Map函数，采用Hash算法，将Hash值相同的数据交给同一个Reduce task；对于第一个Reduce函数，采用HashMap统计出每个词出现的频率，对于第二个Reduce 函数，统计所有Reduce task，输出数据中的top K即可。

直接将数据均分到不同的机器上进行处理是无法得到正确的结果的。因为一个数据可能被均分到不同的机器上，而另一个则可能完全聚集到一个机器上，同时还可能存在具有相同数目的数据。

## 大数据搜索

**[1. Bloom Filter](http://blog.csdn.net/hguisu/article/details/7866173)（判断存在性）**

[【Bloom Filter】](http://blog.csdn.net/hguisu/article/details/7866173)Bloom Filter（BF）是一种空间效率很高的随机**[数据结构](http://lib.csdn.net/base/datastructure)**，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。它是一个判断元素是否存在集合的快速的概率算法。Bloom Filter有可能会出现错误判断，但不会漏掉判断。也就是Bloom Filter判断元素不再集合，那肯定不在。如果判断元素存在集合中，有一定的概率判断错误。因此，Bloom Filter不适合那些“零错误”的应用场合。

而在能容忍低错误率的应用场合下，Bloom Filter比其他常见的算法（如hash，折半查找）极大节省了空间。

Bloom Filter的详细介绍：[海量数据处理之Bloom Filter详解](http://blog.csdn.net/hguisu/article/details/7866173)

【适用范围】   
可以用来实现数据字典，进行数据的判重，或者集合求交集

【基本原理及要点】

原理要点：一是位数组， 而是k个独立hash函数。

**1）位数组：**

假设Bloom Filter使用一个m比特的数组来保存信息，初始状态时，Bloom Filter是一个包含m位的位数组，每一位都置为0，即BF整个数组的元素都设置为0。

2）k个独立hash函数

      为了表达S={x1, x2,…,xn}这样一个n个元素的集合，Bloom Filter使用k个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。

          当我们往Bloom Filter中增加任意一个元素x时候，我们使用k个哈希函数得到k个哈希值，然后将数组中对应的比特位设置为1。即第i个哈希函数映射的位置hashi(x)就会被置为1（1≤i≤k）。 注意，如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个哈希函数选中同一个位置（从左边数第五位，即第二个“1“处）。

 3）判断元素是否存在集合

    在判断y是否属于这个集合时，我们只需要对y使用k个哈希函数得到k个哈希值，如果所有hashi(y)的位置都是1（1≤i≤k），即k个位置都被设置为1了，那么我们就认为y是集合中的元素，否则就认为y不是集合中的元素。下图中y1就不是集合中的元素（因为y1有一处指向了“0”位）。y2或者属于这个集合，或者刚好是一个false positive。

      显然这 个判断并不保证查找的结果是100%正确的。

Bloom Filter的缺点：

       1）Bloom Filter无法从Bloom Filter集合中删除一个元素。因为该元素对应的位会牵动到其他的元素。所以一个简单的改进就是 counting Bloom filter，用一个counter数组代替位数组，就可以支持删除了。 此外，Bloom Filter的hash函数选择会影响算法的效果。

       2）还有一个比较重要的问题，如何根据输入元素个数n，确定位数组m的大小及hash函数个数，即hash函数选择会影响算法的效果**。**当hash函数个数k=(ln2)\*(m/n)时错误率最小。在错误率不大于E的情况 下，m至少要等于n\*lg(1/E) 才能表示任意n个元素的集合。但m还应该更大些，因为还要保证bit数组里至少一半为0，则m应 该>=nlg(1/E)\*lge ，大概就是nlg(1/E)1.44倍(lg表示以2为底的对数)。

举个例子我们假设错误率为0.01，则此时m应大概是n的13倍。这样k大概是8个。

 注意：

         这里m与n的单位不同，m是bit为单位，而n则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多bit的。所以使用bloom filter内存上通常都是节省的。

       一般BF可以与一些key-value的**[数据库](http://lib.csdn.net/base/mysql)**一起使用，来加快查询。由于BF所用的空间非常小，所有BF可以常驻内存。这样子的话，对于大部分不存在的元素，我们只需要访问内存中的BF就可以判断出来了，只有一小部分，我们需要访问在硬盘上的key-value数据库。从而大大地提高了效率。

【扩展】   
Bloom filter将集合中的元素映射到位数组中，用k（k为哈希函数个数）个映射位是否全1表示元素在不在这个集合中。Counting bloom filter（CBF）将位数组中的每一位扩展为一个counter，从而支持了元素的删除操作。Spectral Bloom Filter（SBF）将其与集合元素的出现次数关联。SBF采用counter中的最小值来近似表示元素的出现频率。

【问题实例】   
给你A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，让你找出A,B文件共同的URL。如果是三个乃至n个文件呢？   
根据这个问题我们来计算下内存的占用，4G=2^32大概是40亿\*8大概是340亿bit，n=50亿，如果按出错率0.01算需要的大概是650亿个bit。 现在可用的是340亿，相差并不多，这样可能会使出错率上升些。另外如果这些urlip是一一对应的，就可以转换成ip，则大大简单了。

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

**2. Hash**

【什么是Hash】   
       Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射， pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间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，而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某一固定长度的消息摘要的函数。   
       HASH主要用于信息安全领域中加密算法，它把一些不同长度的信息转化成杂乱的128位的编码,这些编码值叫做HASH值. 也可以说，hash就是找到一种数据内容和数据存放地址之间的映射关系。   
      数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表，哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法——拉链法，（也是树的一种存储结构，称为二叉链表）我们可以理解为“链表的数组”，如图：

                                                    
     左边很明显是个数组，数组的每个成员包括一个指针，指向一个链表的头，当然这个链表可能为空，也可能元素很多。我们根据元素的一些特征把元素分配到不同的链表中去，也是根据这些特征，找到正确的链表，再从链表中找出这个元素。   
元素特征转变为数组下标的方法就是散列法。

散列法当然不止一种，下面列出三种比较常用的：  
1，除法散列法 （求模数）  
最直观的一种，上图使用的就是这种散列法，公式：   
index = value % 16   
学过汇编的都知道，求模数其实是通过一个除法运算得到的，所以叫“除法散列法”。   
2，平方散列法   
求index是非常频繁的操作，而乘法的运算要比除法来得省时（对现在的CPU来说，估计我们感觉不出来），所以我们考虑把除法换成乘法和一个位移操作。公式：   
index = (value \* value) >> 28   
如果数值分配比较均匀的话这种方法能得到不错的结果，但我上面画的那个图的各个元素的值算出来的index都是0——非常失败。也许你还有个问题，value如果很大，value \* value不会溢出吗？答案是会的，但我们这个乘法不关心溢出，因为我们根本不是为了获取相乘结果，而是为了获取index。   
3，斐波那契（Fibonacci）散列法   
平方散列法的缺点是显而易见的，所以我们能不能找出一个理想的乘数，而不是拿value本身当作乘数呢？答案是肯定的。   
1，对于16位整数而言，这个乘数是40503   
2，对于32位整数而言，这个乘数是2654435769   
3，对于64位整数而言，这个乘数是11400714819323198485   
这几个“理想乘数”是如何得出来的呢？这跟一个法则有关，叫黄金分割法则，而描述黄金分割法则的最经典表达式无疑就是著名的斐波那契数列，如果你还有兴趣，就到网上查找一下“斐波那契数列”等关键字，我数学水平有限，不知道怎么描述清楚为什么，另外斐波那契数列的值居然和太阳系八大行星的轨道半径的比例出奇吻合，很神奇，对么？  
对我们常见的32位整数而言，公式：   
i ndex = (value \* 2654435769) >> 28   
如果用这种斐波那契散列法的话，那我上面的图就变成这样了：

很明显，用斐波那契散列法调整之后要比原来的取摸散列法好很多。   
【适用范围】   
快速查找，删除的基本数据结构，通常需要总数据量可以放入内存。   
【基本原理及要点】   
hash函数选择，针对字符串，整数，排列，具体相应的hash方法。   
碰撞处理：

一种是open hashing，也称为拉链法；

另一种就是closed hashing，也称开地址法，opened addressing。   
【扩展】   
d-left hashing中的d是多个的意思，我们先简化这个问题，看一看2-left hashing。2-left hashing指的是将一个哈希表分成长度相等的两半，分别叫做T1和T2，给T1和T2分别配备一个哈希函数，h1和h2。在存储一个新的key时，同 时用两个哈希函数进行计算，得出两个地址h1[key]和h2[key]。这时需要检查T1中的h1[key]位置和T2中的h2[key]位置，哪一个 位置已经存储的（有碰撞的）key比较多，然后将新key存储在负载少的位置。如果两边一样多，比如两个位置都为空或者都存储了一个key，就把新key 存储在左边的T1子表中，2-left也由此而来。在查找一个key时，必须进行两次hash，同时查找两个位置。   
【问题实例】   
1).海量日志数据，提取出某日访问百度次数最多的那个IP。   
IP的数目还是有限的，最多2^32个，所以可以考虑使用hash将ip直接存入内存，然后进行统计。

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

**3. Bit-map**

【什么是Bit-map】   
        所谓的[Bit-map](http://blog.csdn.net/hguisu/article/details/7880288)就是用一个bit位来标记某个元素对应的Value， 而Key即是该元素。由于采用了Bit为单位来存储数据，因此在存储空间方面，可以大大节省。

        如果说了这么多还没明白什么是Bit-map，那么我们来看一个具体的例子，假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复）。那么我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0(如下图：)

然后遍历这5个元素，首先第一个元素是4，那么就把4对应的位置为1（可以这样操作 p+(i/8)|(0x01<<(i%8)) 当然了这里的操作涉及到Big-ending和Little-ending的情况，这里默认为Big-ending）,因为是从零开始的，所以要把第五位置为一（如下图）：

然后再处理第二个元素7，将第八位置为1,，接着再处理第三个元素，一直到最后处理完所有的元素，将相应的位置为1，这时候的内存的Bit位的状态如下：

然后我们现在遍历一遍Bit区域，将该位是一的位的编号输出（2，3，4，5，7），这样就达到了排序的目的。下面的代码给出了一个BitMap的用法：排序。

C代码

复制代码

1 //定义每个Byte中有8个Bit位

2 #include ＜memory.h＞

3 #define BYTESIZE 8

4 void SetBit(char \*p, int posi)

5 {

6 for(int i=0; i ＜ (posi/BYTESIZE); i++)

7 {

8 p++;

9 }

10

11 \*p = \*p|(0x01＜＜(posi%BYTESIZE));//将该Bit位赋值1

12 return;

13 }

14

15 void BitMapSortDemo()

16 {

17 //为了简单起见，我们不考虑负数

18 int num[] = {3,5,2,10,6,12,8,14,9};

19

20 //BufferLen这个值是根据待排序的数据中最大值确定的

21 //待排序中的最大值是14，因此只需要2个Bytes(16个Bit)

22 //就可以了。

23 const int BufferLen = 2;

24 char \*pBuffer = new char[BufferLen];

25

26 //要将所有的Bit位置为0，否则结果不可预知。

27 memset(pBuffer,0,BufferLen);

28 for(int i=0;i＜9;i++)

29 {

30 //首先将相应Bit位上置为1

31 SetBit(pBuffer,num[i]);

32 }

33

34 //输出排序结果

35 for(int i=0;i＜BufferLen;i++)//每次处理一个字节(Byte)

36 {

37 for(int j=0;j＜BYTESIZE;j++)//处理该字节中的每个Bit位

38 {

39 //判断该位上是否是1，进行输出，这里的判断比较笨。

40 //首先得到该第j位的掩码（0x01＜＜j），将内存区中的

41 //位和此掩码作与操作。最后判断掩码是否和处理后的

42 //结果相同

43 if((\*pBuffer&(0x01＜＜j)) == (0x01＜＜j))

44 {

45 printf("%d ",i\*BYTESIZE + j);

46 }

47 }

48 pBuffer++;

49 }

50 }

51

52 int \_tmain(int argc, \_TCHAR\* argv[])

53 {

54 BitMapSortDemo();

55 return 0;

56 }

复制代码

【适用范围】   
  
可进行数据的快速查找，判重，删除，一般来说数据范围是int的10倍以下

【基本原理及要点】  
使用bit数组来表示某些元素是否存在，比如8位电话号码   
  
【扩展】   
  
[Bloom filter](http://blog.csdn.net/hguisu/article/details/7866173)可以看做是对[bit-map](http://blog.csdn.net/hguisu/article/details/7880288)的扩展   
  
【问题实例】   
  
1)已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。   
  
8位最多99 999 999，大概需要99m个bit（1024\*1024 \*99个bit ），大概10几m字节的内存即可。

申请内存空间的大小为:int a[1 + N/32] =（(99 999 999/32 +1)\*4 个字节/1024/1024 = 12M

（可以理解为从0-99 999 999的数字，每个数字对应一个Bit位，所以只需要99M个Bit==12MBytes，这样，就用了小小的12M左右的内存表示了所有的8位数的电话）   
  
2)2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。   
  
将bit-map扩展一下，用2bit表示一个数即可，0表示未出现，1表示出现一次，2表示出现2次及以上，在遍历这些数的时候，如果对应位置的值是0，则将其置为1；如果是1，将其置为2；如果是2，则保持不变。或者我们不用2bit来进行表示，我们用两个bit-map即可模拟实现这个2bit-map，都是一样的道理。

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

### 4. 堆【前n大，前n小，中位数】

【什么是堆】

在八大排序里面有堆 的详细介绍：[八大排序算法](http://blog.csdn.net/hguisu/article/details/7776068" \l "t4)  
概念：堆是一种特殊的二叉树，具备以下两种性质  
1）每个节点的值都大于（或者都小于，称为最小堆）其子节点的值  
2）树是完全平衡的，并且最后一层的树叶都在最左边  
这样就定义了一个最大堆。如下图用一个数组来表示堆：

那么下面介绍二叉堆：二叉堆是一种完全二叉树，其任意子树的左右节点（如果有的话）的键值一定比根节点大，上图其实就是一个二叉堆。

你一定发觉了，最小的一个元素就是数组第一个元素，那么二叉堆这种有序队列如何入队呢？看图：

假设要在这个二叉堆里入队一个单元，键值为2，那只需在数组末尾加入这个元素，然后尽可能把这个元素往上挪，直到挪不动，经过了这种复杂度为Ο(logn)的操作，二叉堆还是二叉堆。

那如何出队呢？也不难，看图

出队一定是出数组的第一个元素，这么来第一个元素以前的位置就成了空位，我们需要把这个空位挪至叶子节点，然后把数组最后一个元素插入这个空位，把这个“空位”尽量往上挪。这种操作的复杂度也是Ο(logn)。

【适用范围】  
海量数据前n大，并且n比较小，堆可以放入内存

【基本原理及要点】  
最大堆求前n小，最小堆求前n大。方法，比如求前n小，我们比较当前元素与最大堆里的最大元素，如果它小于最大元素，则应该替换那个最大元 素。这样最后得到的n个元素就是最小的n个。适合**[大数据](http://lib.csdn.net/base/hadoop)**量，求前n小，n的大小比较小的情况，这样可以扫描一遍即可得到所有的前n元素，效率很高。

【扩展】  
双堆，一个最大堆与一个最小堆结合，可以用来维护中位数。

【问题实例】  
1)100w个数中找最大的前100个数。  
用一个100个元素大小的最小堆即可。

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

**5. 双层桶**

【什么是双层桶】    
事实上，与其说双层桶划分是一种数据结构，不如说它是一种算法设计思想。面对一堆大量的数据我们无法处理的时候，我们可以将其分成一个个小的单元，然后根据一定的策略来处理这些小单元，从而达到目的。

【适用范围】   
第k大，中位数，不重复或重复的数字

【基本原理及要点】   
因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。可以通过多次缩小，双层只是一个例子，分治才是其根本（只是“只分不治”）。

【扩展】   
当有时候需要用一个小范围的数据来构造一个大数据，也是可以利用这种思想，相比之下不同的，只是其中的逆过程。

【问题实例】   
1).2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。

有 点像鸽巢原理，整数个数为2^32,也就是，我们可以将这2^32个数，划分为2^8=256个区域(比如用单个文件代表一个区域)，然后将数据分离到不同的区 域，然后不同的区域在利用bitmap就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。 当然这个题也可以用我们前面讲过的BitMap方法解决，正所谓条条大道通罗马~~~

2).5亿个int找它们的中位数。

这个例子比上面那个更明显。首先我们将int划分为2^16个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

实 际上，如果不是int是int64，我们可以经过3次这样的划分即可降低到可以接受的程度。即可以先将int64分成2^24个区域，然后确定区域的第几 大数，在将该区域分成2^20个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有2^20，就可以直接利用direct addr table进行统计了。

3).现在有一个0-30000的随机数生成器。请根据这个随机数生成器，设计一个抽奖范围是0-350000彩票中奖号码列表，其中要包含20000个中奖号码。

这个题刚好和上面两个思想相反，一个0到3万的随机数生成器要生成一个0到35万的随机数。那么我们完全可以将0-35万的区间分成35/3=12个区 间，然后每个区间的长度都小于等于3万，这样我们就可以用题目给的随机数生成器来生成了，然后再加上该区间的基数。那么要每个区间生成多少个随机数呢？计算公式就是：区间长度\*随机数密度，在本题目中就是30000\*（20000/350000）。最后要注意一点，该题目是有隐含条件的：彩票，这意味着你 生成的随机数里面不能有重复，这也是我为什么用双层桶划分思想的另外一个原因。

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

**6. 数据库索引及优化**

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。

数据库索引

什么是索引

　　数据库索引好比是一本书前面的目录，能加快数据库的查询速度。  
　　例如这样一个查询：select \* from table1 where id=44。如果没有索引，必须遍历整个表，直到ID等于44的这一行被找到为止；有了索引之后(必须是在ID这一列上建立的索引)，直接在索引里面找44（也就是在ID这一列找），就可以得知这一行的位置，也就是找到了这一行。可见，索引是用来定位的。  
　　索引分为聚簇索引和非聚簇索引两种，聚簇索引 是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

概述

　　建立索引的目的是加快对表中记录的查找或排序。  
　　为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

B树索引-Sql Server索引方式

为什么要创建索引

　　创建索引可以大大提高系统的性能。  
　　　　第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。  
　　　　第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。  
　　　　第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。  
　　　　第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。  
　　　　第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。  
　　也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。  
　　　　第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。  
　　　　第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。  
　　　　第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

在哪建索引

　　索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。一般来说，应该在这些列上创建索引：  
　　在经常需要搜索的列上，可以加快搜索的速度；  
　　在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；  
　　在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；  
　　在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；  
　　在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。  
　　同样，对于有些列不应该创建索引。一般来说，不应该创建索引的的这些列具有下列特点：  
　　第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。  
　　第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。  
　　第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少,不利于使用索引。  
　　第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改操作远远多于检索操作时，不应该创建索引。

数据库优化

　　此外，除了数据库索引之外，在LAMP结果如此流行的今天，数据库（尤其是**[MySQL](http://lib.csdn.net/base/mysql)**）性能优化也是海量数据处理的一个热点。下面就结合自己的经验，聊一聊MySQL数据库优化的几个方面。  
　　首先，在数据库设计的时候，要能够充分的利用索引带来的性能提升，至于如何建立索引，建立什么样的索引，在哪些字段上建立索引，上面已经讲的很清楚了，这里不在赘述。另外就是设计数据库的原则就是尽可能少的进行数据库写操作（插入，更新，删除等），查询越简单越好。如下：

**数据库设计：**

. 创建索引

. 查询语句

1）查询越简单越好:单表查询 > inner join >其他

        2）更新越少越好

　　其次，配置缓存是必不可少的，配置缓存可以有效的降低数据库查询读取次数，从而缓解数据库服务器压力，达到优化的目的，一定程度上来讲，这算是一个“围魏救赵”的办法。可配置的缓存包括索引缓存(key\_buffer)，排序缓存(sort\_buffer)，查询缓存(query\_buffer)，表描述符缓存(table\_cache)，如下：

**配置缓存：**

. 索引缓存（key\_buffer）

. 排序缓存 （sort\_buffer）

. 查询缓存  （query\_buffer）

. 表描述符缓存（table\_cache）

　　第三，切表，切表也是一种比较流行的数据库优化法。分表包括两种方式：横向分表和纵向分表，其中，横向分表比较有使用意义，故名思议，横向切表就是指把记录分到不同的表中，而每条记录仍旧是完整的（纵向切表后每条记录是不完整的），例如原始表中有100条记录，我要切成2个表，那么最简单也是最常用的方法就是ID取摸切表法，本例中，就把ID为1,3,5,7。。。的记录存在一个表中，ID为2,4,6,8,。。。的记录存在另一张表中。虽然横向切表可以减少查询强度，但是它也破坏了原始表的完整性，如果该表的统计操作比较多，那么就不适合横向切表。横向切表有个非常典型的用法，就是用户数据：每个用户的用户数据一般都比较庞大，但是每个用户数据之间的关系不大，因此这里很适合横向切表。最后，要记住一句话就是：分表会造成查询的负担，因此在数据库设计之初，要想好是否真的适合切表的优化：

**切表分表：**

. 纵向 ：字段较多时可以考虑，一般用处不到

. 横向 ：1）能有效降低表的大小，减少由于枷锁导致的等待

             2）查询会变得复杂，尤其是需要排序的查询

第四，日志分析，在数据库运行了较长一段时间以后，会积累大量的LOG日志，其实这里面的蕴涵的有用的信息量还是很大的。通过分析日志，可以找到系统性能的瓶颈，从而进一步寻找优化方案。

**数据库性能分析：**

. 查询吞吐量，数据量监控

. 慢查询分析：索引，I/O,cpu等。

以上讲的都是单机MySQL的性能优化的一些经验，但是随着信息大爆炸，单机的数据库服务器已经不能满足我们的需求，于是，多多节点，分布式数据库网络出现了，其一般的结构如下：

分布式数据库结构

这种分布式集群的技术关键就是“同步复制”。。。

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

**7. 倒排索引(搜索引擎之基石)**

引言：

在信息大爆炸的今天，有了**[搜索引擎](http://lib.csdn.net/base/searchengine)**的帮助，使得我们能够快速，便捷的找到所求。提到搜索引擎，就不得不说VSM模型，说到VSM，就不得不聊倒排索引。可以毫不夸张的讲，倒排索引是搜索引擎的基石。

VSM检索模型

VSM全称是Vector Space Model(向量空间模型)，是IR(Information Retrieval信息检索)模型中的一种，由于其简单，直观，高效，所以被广泛的应用到搜索引擎的**[架构](http://lib.csdn.net/base/architecture)**中。98年的Google就是凭借这样的一个模型，开始了它的疯狂扩张之路。废话不多说，让我们来看看到底VSM是一个什么东东。

在开始之前，我默认大家对线性代数里面的向量(Vector)有一定了解的。向量是既有大小又有方向的量，通常用有向线段表示，向量有：加、减、倍数、内积、距离、模、夹角的运算。

文档(Document)：一个完整的信息单元，对应的搜索引擎系统里，就是指一个个的网页。

标引项(Term)：文档的基本构成单位，例如在英文中可以看做是一个单词，在中文中可以看作一个词语。

查询(Query)：一个用户的输入，一般由多个Term构成。

那么用一句话概况搜索引擎所做的事情就是：对于用户输入的Query，找到最相似的Document返回给用户。而这正是IR模型所解决的问题：

信息检索模型是指如何对查询和文档进行表示，然后对它们进行相似度计算的框架和方法。

举个简单的例子：

现在有两篇文章(Document)分别是 “春风来了，春天的脚步近了” 和 “春风不度玉门关”。然后输入的Query是“春风”，从直观上感觉，前者和输入的查询更相关一些，因为它包含有2个春，但这只是我们的直观感觉，如何量化呢，要知道计算机是门严谨的学科^\_^。这个时候，我们前面讲的Term和VSM模型就派上用场了。

首先我们要确定向量的维数，这时候就需要一个字典库，字典库的大小，即是向量的维数。在该例中，字典为{春风,来了,春天, 的,脚步,近了,不度,玉门关} ，文档向量，查询向量如下图：

VSM模型示例

PS:为了简单起见，这里分词的粒度很大。

将Query和Document都量化为向量以后，那么就可以计算用户的查询和哪个文档相似性更大了。简单的计算结果是D1和D2同Query的内积都是1，囧。当然了，如果分词粒度再细一些，查询的结果就是另外一个样子了，因此分词的粒度也是会对查询结果（主要是召回率和准确率）造成影响的。

上述的例子是用一个很简单的例子来说明VSM模型的，计算文档相似度的时候也是采用最原始的内积的方法，并且只考虑了词频(TF)影响因子，而没有考虑反词频(IDF)，而现在比较常用的是cos夹角法，影响因子也非常多，据传Google的影响因子有100+之多。  
大名鼎鼎的Lucene项目就是采用VSM模型构建的，VSM的核心公式如下（由cos夹角法演变，此处省去推导过程）

VSM模型公式

从上面的例子不难看出，如果向量的维度(对汉语来将，这个值一般在30w-45w)变大，而且文档数量(通常都是海量的)变多，那么计算一次相关性，开销是非常大的，如何解决这个问题呢？不要忘记了我们这节的主题就是 倒排索引，主角终于粉墨登场了！！！

倒排索引非常类似我们前面提到的Hash结构。以下内容来自维基百科：

倒排索引（英语：Inverted index），也常被称为反向索引、置入档案或反向档案，是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。

有两种不同的反向索引形式：

* 一条记录的水平反向索引（或者反向档案索引）包含每个引用单词的文档的列表。
* 一个单词的水平反向索引（或者完全反向索引）又包含每个单词在一个文档中的位置。

后者的形式提供了更多的兼容性（比如短语搜索），但是需要更多的时间和空间来创建。

由上面的定义可以知道，一个倒排索引包含一个字典的索引和所有词的列表。其中字典索引中包含了所有的Term(通俗理解为文档中的词)，索引后面跟的列表则保存该词的信息(出现的文档号，甚至包含在每个文档中的位置信息)。下面我们还采用上面的方法举一个简单的例子来说明倒排索引。

例如现在我们要对三篇文档建立索引(实际应用中，文档的数量是海量的)：

文档1(D1)：中国移动互联网发展迅速

文档2(D2)：移动互联网未来的潜力巨大

文档3(D3)：中华民族是个勤劳的民族

那么文档中的词典集合为：{中国，移动，互联网，发展，迅速，未来，的，潜力，巨大，中华，民族，是，个，勤劳}

建好的索引如下图：

倒排索引

在上面的索引中，存储了两个信息，文档号和出现的次数。建立好索引以后，我们就可以开始查询了。例如现在有一个Query是”中国移动”。首先分词得到Term集合{中国，移动}，查倒排索引，分别计算query和d1,d2,d3的距离。有没有发现，倒排表建立好以后，就不需要在检索整个文档库，而是直接从字典集合中找到“中国”和“移动”，然后遍历后面的列表直接计算。

对倒排索引结构我们已经有了初步的了解，但在实际应用中还有些需要解决的问题(主要是由海量数据引起的)。笔者列举一些问题，并给出相应的解决方案，抛砖以引玉，希望大家可以展开讨论：

1.左侧的索引表如何建立?怎么做才能最高效？

可能有人不假思索回答：左侧的索引当然要采取hash结构啊，这样可以快速的定位到字典项。但是这样问题又来了，hash函数如何选取呢？而且hash是有碰撞的，但是倒排表似乎又是不允许碰撞的存在的。事实上，虽然倒排表和hash异常的相思，但是两者还是有很大区别的，其实在这里我们可以采用前面提到的Bitmap的思想，每个Term(单词)对应一个位置(当然了，这里不是一个比特位)，而且是一一对应的。如何能够做到呢，一般在文字处理中，有很多的编码，汉字中的GBK编码基本上就可以包含所有用到的汉字，每个汉字的GBK编码是确定的，因此一个Term的”ID”也就确定了，从而可以做到快速定位。注：得到一个汉字的GBK号是非常快的过程，可以理解为O(1)的时间复杂度。

2.如何快速的添加删除更新索引？

有经验的码农都知道，一般在系统的“做加法”的代价比“做减法”的代价要低很多，在搜索引擎中中也不例外。因此，在倒排表中，遇到要删除一个文档，其实不是真正的删除，而是将其标记删除。这样一个减法操作的代价就比较小了。

3.那么多的海量文档，如果存储呢？有么有什么备份策略呢？

当然了，一台机器是存储不下的，分布式存储是采取的。一般的备份保存3份就足够了。

好了，倒排索引终于完工了，不足的地方请指正。谢谢

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

### 8. 外排序【内存不够】

适用范围：

大数据的排序，去重  
 基本原理及要点：

外部排序的两个独立阶段：

1）首先按内存大小，将外存上含n个记录的文件分成若干长度L的子文件或段。依次读入内存并利用有效的内部排序对他们进行排序，并将排序后得到的有序字文件重新写入外存，通常称这些子文件为归并段。

2）对这些归并段进行逐趟归并，使归并段逐渐由小到大，直至得到整个有序文件为之。

外排序的归并方法，置换选择 败者树原理，最优归并树  
 扩展：  
 问题实例：  
 1).有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16个字节，内存限制大小是1M。返回频数最高的100个词  
这个数据具有很明显的特点，词的大小为16个字节，但是内存只有1m做hash有些不够，所以可以用来排序。内存可以当输入缓冲区使用。

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

### 9. trie树【字典树，字符串去重，高频】

 适用范围：

数据量大，重复多，但是数据种类小可以放入内存

基本原理及要点：

实现方式，节点孩子的表示方式  
 扩展：

压缩实现。  
 问题实例：  
 1).有10个文件，每个文件1G， 每个文件的每一行都存放的是用户的query，每个文件的query都可能重复。要你按照query的频度排序 。  
 2).1000万字符串，其中有些是相同的(重复),需要把重复的全部去掉，保留没有重复的字符串。请问怎么设计和实现？  
 3).寻找热门查询：查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个，每个不超过255字节。

[Return Top](https://www.cnblogs.com/ECJTUACM-873284962/p/6910842.html" \l "_labelTop)

**10. 分布式处理 mapreduce**

基本原理及要点：

将数据交给不同的机器去处理，数据划分，结果归约。

扩 展：   
问题实例：   
  
1).The canonical example application of MapReduce is a process to count the appearances of   
  
each different word in a set of documents:   
void map(String name, String document):   
// name: document name   
// document: document contents   
for each word w in document:   
EmitIntermediate(w, 1);   
  
void reduce(String word, Iterator partialCounts):   
// key: a word   
// values: a list of aggregated partial counts   
int result = 0;   
for each v in partialCounts:   
result += ParseInt(v);   
Emit(result);   
Here, each document is split in words, and each word is counted initially with a "1" value by   
  
the Map function, using the word as the result key. The framework puts together all the pairs   
  
with the same key and feeds them to the same call to Reduce, thus this function just needs to   
  
sum all of its input values to find the total appearances of that word.   
  
2). 海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10。   
  
3).一共有N个机器，每个机器上有N个数。每个机器最多存 O(N)个数并对它们操作。如何找到N^2个数的中数(median)？   
  
  
经典问题分析   
  
上千万or亿数据（有 重复），统计其中出现次数最多的前N个数据,分两种情况：可一次读入内存，不可一次读入。   
  
可用思路：trie树+堆，数据库索引，划分 子集分别统计，hash，分布式计算，近似统计，外排序   
  
所谓的是否能一次读入内存，实际上应该指去除重复后的数据量。如果去重后数据可 以放入内存，我们可以为数据建立字典，比如通过 map，hashmap，trie，然后直接进行统计即可。当然在更新每条数据的出现次数的时候，我们可以利用一个堆来维护出现次数最多的前N个数据，当然这样导致维护次数增加，不如完全统计后在求前N大效率高。   
  
如果数据无法放入内存。一方面我们可以考虑上面的字典方法能否被改进以适应这种情形，可以做的改变就是将字典存放到硬盘上，而不是内存，这可以参考数据库的存储方法。   
  
当然还有更好的方法，就是可以采用分布式计算，基本上就是map-reduce过程，首先可以根据数据值或者把数据hash(md5)后的值，将数据按照范围划分到不同的机子，最好可以让数据划分后可以一次读入内存，这样不同的机子负责处理各种的数值范围，实际上就是map。得到结果后，各个机子只需拿出各自的出现次数最多的前N个数据，然后汇总，选出所有的数据中出现次数最多的前N个数据，这实际上就是reduce过程。   
  
实际上可能想直接将数据均分到不同的机子上进行处理，这样是无法得到正确的解的。因为一个数据可能被均分到不同的机子上，而另一个则可能完全聚集到一个机子上，同时还可能存在具有相同数目的数据。比如我们要找出现次数最多的前100个，我们将1000万的数据分布到10台机器上，找到每台出现次数最多的前 100个，归并之后这样不能保证找到真正的第100个，因为比如出现次数最多的第100个可能有1万个，但是它被分到了10台机子，这样在每台上只有1千个，假设这些机子排名在1000个之前的那些都是单独分布在一台机子上的，比如有1001个，这样本来具有1万个的这个就会被淘汰，即使我们让每台机子选出出现次数最多的1000个再归并，仍然会出错，因为可能存在大量个数为1001个的发生聚集。因此不能将数据随便均分到不同机子上，而是要根据hash 后的值将它们映射到不同的机子上处理，让不同的机器处理一个数值范围。   
  
而外排序的方法会消耗大量的IO，效率不会很高。而上面的分布式方法，也可以用于单机版本，也就是将总的数据根据值的范围，划分成多个不同的子文件，然后逐个处理。处理完毕之后再对这些单词的及其出现频率进行一个归并。实际上就可以利用一个外排序的归并过程。   
  
另外还可以考虑近似计算，也就是我们可以通过结合自然语言属性，只将那些真正实际中出现最多的那些词作为一个字典，使得这个规模可以放入内存。

## 排序算法



希尔排序的时间复杂度与其增量序列有关，目前最好的是塞奇威克增量序列，O(n^4/3)

最坏O(n^(1.5))

# 计算机网络

## HTTP

HTTP是一种能够获取如 HTML 这样的网络资源的 protocol(通讯协议)。它是在 Web 上进行数据交换的基础，是一种 client-server 协议，也就是说，请求通常是由像浏览器这样的接受方发起的。一个完整的Web文档通常是由不同的子文档拼接而成的，像是文本、布局描述、图片、视频、脚本等等。

### HTTP版本区别

**缓存处理**，在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。

**带宽优化及网络连接的使用**，HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

**错误通知的管理，**在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

**Host头处理，**在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。

**长连接，**HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection： keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

### 构成

**请求**

1.**一个HTTP的method**，经常是由一个动词像GET, POST 或者一个名词像OPTIONS，HEAD来定义客户端的动作行为。通常客户端的操作都是获取资源（GET方法）或者发送HTML form表单值（POST方法），虽然在一些情况下也会有其他操作。

**2.要获取的资源的路径**，通常是上下文中就很明显的元素资源的URL，它没有protocol（http://），domain（developer.mozilla.org），或是TCP的port（HTTP一般在80端口）。

HTTP协议版本号。

3.为服务端表达其他信息的可选头部headers。

4.对于一些像POST这样的方法，报文的body就包含了发送的资源，这与回应报文的body类似

**回应报文**

HTTP协议版本号。

一个状态码（status code），来告知对应请求执行成功或失败，以及失败的原因。

一个状态信息，这个信息是非权威的状态码描述信息，可以由服务端自行设定。

HTTP headers，与请求头部类似。

可选项，比起请求报文，响应报文中更常见地包含获取的资源body。

### HTTP状态码

当浏览者访问一个网页时，浏览者的浏览器会向网页所在服务器发出请求。当浏览器接收并显示网页前，此网页所在的服务器会返回一个包含HTTP状态码的信息头（server header）用以响应浏览器的请求。

HTTP状态码的英文为HTTP Status Code。

下面是常见的HTTP状态码：

101 Switching Protocols 切换协议。服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到HTTP的新版本协议

200 - 请求成功

301 - 资源（网页等）被永久转移到其它URL

400 Bad Request 客户端请求的语法错误，服务器无法理解

401 Unauthorized 请求要求用户的身份认证

402 Payment Required 保留，将来使用

403 Forbidden 服务器理解请求客户端的请求，但是拒绝执行此请求

404 Not Found 请求的资源（网页等）不存在

500 - 内部服务器错误

|  |  |
| --- | --- |
| 分类 | 分类描述 |
| 1\*\* | 信息，服务器收到请求，需要请求者继续执行操作 |
| 2\*\* | 成功，操作被成功接收并处理 |
| 3\*\* | 重定向，需要进一步的操作以完成请求 |
| 4\*\* | 客户端错误，请求包含语法错误或无法完成请求 |
| 5\*\* | 服务器错误，服务器在处理请求的过程中发生了错误 |

### 过程

**1.DNS解析**

DNS解析的过程就是寻找哪台机器上有你需要资源的过程，寻找的过程遵循就近原则。

输入一个网址并按回车的时候浏览器会根据输入的URL去查找对应的IP，具体过程如下：

（1）首先是查找**浏览器缓存**，浏览器会保存一段时间内访问过的一些网址的DNS信息，不同浏览器保存的时常不等。

（2）如果没有找到对应的记录，这个时候浏览器会尝试调用**操作系统缓存**来继续查找这个网址的对应DNS信息。

（3）如果还是没找到对应的IP，那么接着会发送一个请求到**路由器**上，然后路由器在自己的路由器缓存上查找记录，路由器一般也存有DNS信息。

（4）如果还是没有，这个请求就会被发送到ISP（注：Internet Service Provider，互联网服务提供商，就是网络运营商，中国电信中国移动等），ISP也会有相应的ISP DNS服务器，就是本地**DNS服务器**，请求的域名基本上都能在这里找得到。

（5）如果还是没有的话， ISP的DNS服务器会将请求发向**根域名服务器**进行搜索。根域名服务器就是面向全球的顶级DNS服务器，共有13台逻辑上的服务器，从A到M命名，真正的实体服务器则有几百台，分布于全球各大洲。

（6）如果到了这里还是找不到域名的对应信息，那只能说明一个问题：这个域名本来就不存在，它没有在网上正式注册过。或者域名过期了。

这也就是为什么有时候打开一个新页面会有点慢，因为如果本地没什么缓存，查找域名的过程要这样递归地查询下去，查找完还要一层层的向上返回。例如"mp3.baidu.com"，域名先是解析出这是个.com的域名，然后跑到管理.com域的服务器上进行进一步查询，然后是.baidu，最后是mp3， 所以域名结构为：三级域名.二级域名.一级域名。

所以DNS根据域名查询IP地址的过程为：浏览器缓存 --> 操作系统缓存 --> 路由器缓存-->本地（ISP）域名服务器缓存 --> 根域名服务器。

**2.进行TCP连接**

浏览器终于得到了IP以后，向服务器发送TCP连接，TCP连接经过三次握手。

**3.浏览器发送HTTP请求**

浏览器和服务器建立连接以后，浏览器接着给这个IP地址给服务器发送一个http请求，方式为get，例如访问www.baidu.com。其本质是在建立起的TCP连接中，按照HTTP协议标准发送一个索要网页的请求。

这个get请求包含了主机（Host）、用户代理(User-Agent)，用户代理就是自己的浏览器，它是你的"代理人"，Connection（连接属性）中的keep-alive表示浏览器告诉对方服务器在传输完现在请求的内容后不要断开连接，不断开的话下次继续连接速度就很快了。可能还会有Cookies，Cookies保存了用户的登陆信息，一般保存的是用户的JSESSIONID，在每次向服务器发送请求的时候会重复发送给服务器。

在建立连接发送请求时每个服务端需要和客户端保持通信，有很多客户端都会和服务器进行通信。服务器为了识别是哪个客户端与它通信，就必须用一个标识记录客户端的信息。客户端首次访问服务器，服务端返回响应时通过附带一个记录的客户端信息的标识来返回给客户端，这个标识就是JSESSIONID，JSESSIONID就放在了客户端的Cookies里。当客户端再次向服务器发送请求时上就使用上次记录的Cookies里面的JSESSIONID，这样服务器就知道是哪个浏览器了。这样他们之间就能保持通信了。

发送完请求接下来就是等待回应了，

**4.服务器处理请求**

发送完请求接下来就是等待回应了，如下图：

服务器收到浏览器的请求以后），会解析这个请求（读请求头），然后生成一个响应头和具体响应内容。接着服务器会传回来一个响应头和一个响应，响应头告诉了浏览器一些必要的信息，例如重要的Status Code，2开头如200表示一切正常，3开头表示重定向，4开头是客户端错误，如404表示请求的资源不存在，5开头表示服务器端错误。响应就是具体的要请求的页面内容。

**5.浏览器解析渲染页面**

（1）浏览器显示HTML

当服务器返回响应之后，浏览器读取关于这个响应的说明书（响应头），然后开始解析这个响应并在页面上显示出来。

浏览器打开一个网址的时候会慢慢加载这个页面，一部分一部分的显示，直到完全显示，知道最后的旋转进度条停止。因此在浏览器没有完整接受全部HTML文档时，它就已经开始显示这个页面了。

（2）浏览器向服务器发送请求获取嵌入在HTML中的对象

在浏览器显示HTML时，打开一个网页的过程中，主页（index）页面框架传送过来以后，浏览器还会因页面上的静态资源多次发起连接请求，需要获取嵌入在HTML中的其他地址的资源。这时，浏览器会发送一些请求来获取这些文件。这些内容也要一点点地请求过来，所以标签栏转啊转，内容刷啊刷，最后全部请求并加载好了就终于好了。

这时请求的内容是主页里面包含的一些资源，如图片，视频，css样式，JavaScript文件等等。

这在文件属于静态文件，首次访问会留在浏览器的缓存中，过期才会从服务器去取。缓存的内容通常不会保存很久，因为难保网站不会被改动。

静态的文件一般会从CDN中去取，CDN根据请求获取资源的时候可能还会用到负载均衡。

（3）浏览器发送异步（AJAX）请求

对于那些动态的请求，动态网页等就必须要从服务器获取了。对于静态的页面内容，浏览器通常会进行缓存，而对于动态的内容，浏览器通常不会进行缓存。对于这些动态请求，Nginx可能会专门设置一些服务器用来处理这些访问动态页面的请求。

**6.关闭TCP连接**

当数据完成请求到返回的过程之后，根据Connection的Keep-Alive属性可以选择是否断开TCP连接，HTTP/1.1一般支持同一个TCP多个请求，而不是1.0版本下的完成一次请求就发生断开。TCP的断开与连接不一样，断开可以分为主动关闭和被动关闭，需要经过4次握手。

当浏览器需要的全部数据都已经加载完毕，一个页面就显示完了。