内存地址对齐，是一种在计算机内存中排列数据、访问数据的一种方式，包含了两种相互独立又相互关联的部分：基本数据对齐和结构体数据对齐。当今的计算机在计算机内存中读写数据时都是按字(word)大小块来进行操作的(在32位系统中，数据总线宽度为32，每次能读取4字节，地址总线宽度为32，因此最大的寻址空间为2^32=4GB，但是最低2位A[0]，A[1]是不用于寻址，A[2-31]才能存储器相连，因此只能访问4的倍数地址空间，但是总的寻址空间还是2^30\*字长=4GB，因此在内存中所有存放的基本类型数据的首地址的最低两位都是0，除结构体中的成员变量)。基本类型数据对齐就是数据在内存中的偏移地址必须等于一个字的倍数，按这种存储数据的方式，可以提升系统在读取数据时的性能。为了对齐数据，可能必须在上一个数据结束和下一个数据开始的地方插入一些没有用处字节，这就是结构体数据对齐。

举个例子，假设计算机的字大小为4个字节，因此变量在内存中的首地址都是满足4地址对齐，CPU只能对4的倍数的地址进行读取，而每次能读取4个字节大小的数据。假设有一个整型的数据a的首地址不是4的倍数(如下图所示)，不妨设为0X00FFFFF3，则该整型数据存储在地址范围为0X00FFFFF3~0X00FFFFF6的存储空间中，而CPU每次只能对4的倍数内存地址进行读取，因此想读取a的数据，CPU要分别在0X00FFFFF0和0X00FFFFF4进行两次内存读取，而且还要对两次读取的数据进行处理才能得到a的数据，而一个程序的瓶颈往往不是CPU的速度，而是取决于内存的带宽，因为CPU得处理速度要远大于从内存中读取数据的速度，因此减少对内存空间的访问是提高程序性能的关键。从上例可以看出，采取内存地址对齐策略是提高程序性能的关键。

内存地址对齐 - Stupid Bastard - 学习者的博客

结构体(struct)是C语言中非常有用的用户自定义数据类型，而结构体类型的变量以及其各成员在内存中的又是怎样布局的呢？怎样对齐的呢？很显然结构体变量首地址必须是4字节对齐的，但是结构体的每个成员有各自默认的对齐方式，结构体中各成员在内存中出现的位置是随它们的声明顺序依次递增的，并且第一个成员的首地址等于整个结构体变量的首地址。下面列出了在Microsoft，Borland，GNU上对于X86架构32位系统的结构体成员各种类型的默认对齐方式。

char(1字节)，1字节对齐

short(2字节)，2字节对齐

int(4字节)，4字节对齐

float(4字节)，4字节对齐

double(8字节)，Windows系统中8字节对齐，Linux系统中4字节对齐。

当结构体某一成员后面紧跟一个要求比较大的地址对齐成员时(例如char成员变量后面跟一个double成员变量)，这时要插入一些没有实际意义的填充(Padding)。而且总的结构体大小必须为最大对齐的倍数。

下面是一个有char，int，short三种类型，4个成员组成的结构体，该结构体在还未编译之前是大小占8个字节。

struct AlignData{

char a;

short b;

int c;

char d;

};

编译之后，为了保持结构体中的每个成员都是按照各自的对齐，编译器会在一些成员之间插入一些padding，因此编译后得到如下的结构体：

struct AlignData {

char a;

char Padding0[1];

short b;

int c;

char d;

char Padding1[3];

};

编译后该结构体的大小为12个字节，最后一个成员d后面填充的字节数要使该结构体的总大小是其成员类型中拥有最大字节数的倍数(int拥有最大字节数)，因此d后面要填充3个字节。下面举一些结构体例子来说明结构体的填充方式：

例子1：

struct struct1{

char a1;

char b1;

};

结构体struct1的大小为2字节，因为char在结构体中的默认对齐是1，因此在a1和b1之间没有数据填充，而且其成员中占用字节最大的类型为char，因此结构体结束处和b1之间也没有数据填充。

例子2:

struct struct2{

char a2;

short b2;

};

结构体struct2的大小为4字节，b2的是按2字节对齐，因此在b2于a2之间填充一个字节，而其成员中占用字节最大的类型为short，因此该结构体结束处和b2之间没有任何数据填充。

例子3:

Struct struct3{

double a3;

char b3;

};

结构体struct3的大小为16字节，因为b3是按1字节对齐，所以b3与a3之间没有数据填充，而其成员中占用字节最大的类型为double，在Windows平台下是8字节对齐，因此该结构体结束处和b3之间有7个字节的数据填充。

填充字节的大小和新的偏移地址有如下计算公式：

padding = (align - (offset mod align)) mod align

new offset = offset + ((align - (offset mod align)) mod align)

例如求成员a，b之间的填充字节，b的默认对齐为align=2个字节，b的未填充之前的偏移量offset=1，因此填充字节数padding=(2-(1 mod 2)) mod 2 = 1字节。如果要算接下来的成员之间的填充数，已经填充的字节也要算上，不然在算偏移量的时候会出错编译后的结构体比未编译之前多出了3个字节，有没有什么办法可以在保持各成员地址对齐的前提下，又能减少结构体的大小？答案是肯定的！

如果把struct AlignData的成员顺序调整成如下形式：

struct AlignData{

char a;

char d;

short b;

int c;

};

那么编译后不用填充字节就能保持所有的成员都按各自默认的地址对齐。这样可以节约不少内存！一般的结构体成员按照默认对齐字节数递增或是递减的顺序排放，会使总的填充字节数最少。基本数据类型数组在内存中的布局并不是每个数组的元素都是按照4字节对齐的，但是数组的首地址必须是按照4字节对齐，而且每个元素之间没有填充，为什么没有填充呢？地址对齐和填充的目的是减少内存读取的次数，但现在只要数组的首地址按4字节对齐，任何小于等于4字节的类型数组(char, short, int)中的任意数组元素都能通过一次内存读取来获得(假设该数据没有加载到高速缓存)，任何大于4字节类型数组(double)中的任意数组元素都能通过两次内存读取来获得任何大于4字节类型数组(double)中的任意数组元素都能通过两次内存读取来获得。因此要求每个数组元素都是按照4字节对齐是没有必要，浪费空间的。

结构体数组在内存中的布局，只要保持结构体数组的首地址是按照4字节对齐，而且每个数组元素同样也不必按照4字节地址对齐，就能尽量使内存的读取次数降到最低，因为只要每个结构体元素自己内部的填充和对齐都是上述的方式，那么同样也能达到既能减少内存访问的次数，又能节约不必要的内存浪费。但是有人会有这样的疑问，既然每个结构体首地址按照4字节对齐，为什么结构体内部每种数据类型还要各自默认的对齐大小进行对齐？其实其目的同样也是减少内存访问的次数，因为结构体是用户自定义的类型，内部还是由一些基本数据类型组成的！以上的对齐方式都是Windows默认的对齐方式，用户可以根据需求来设置自己的对齐方式，特别是在一些内存受限的系统中，内存比速度更重要！但是建议用户还是不要轻易来设置自己的对齐方式，如果用得不恰当的话，可能会造成大量冗余的内存读取，而且可能会出现不兼容的问题。可以用#pragma pack指令来对其进行设置。由内存地址对齐而引发的对减少内存访问次数的思考当今的CPU的处理速度远比内存访问的速度快，程序的执行速度的瓶颈往往不是CPU的处理速度不够，而是内存访问的延迟，虽然当今CPU中加入了高速缓存用来掩盖内存访问的延迟，但是如果高密集的内存访问，一种延迟是无可避免的。内存地址对齐给程序带来了很大的性能提升，在windows等系统了，编译器都提供了自动地址对齐，给程序员带来了很大的方便。但是减少对内存访问还是值得探讨的问题。

调整结构体成员变量的布局是减少内存访问次数的途径之一。下面分别介绍两种不同的结构体数据成员调整方案，都能得到很好的性能提升。

1. 按成员内存对齐大小按升序或是降序排序，减少结构体的大小。看如下两个结构体：

struct BeforeAdjust{

char a;

short b;

int c;

char d;

};

struct AfterAdjust{

char a;

char d;

short b;

int c;

};

从表面上看结构体BeforeAdjust和AfterAdjust成员都一样，就是成员布置的顺序有差异，因此造成了这两种类型数据占据空间大小有所不同，BeforeAdjust大小占12个字节，AfterAdjust大小占8个字节，因此从读取一个BeforeAdjust类型的数据要进行3次内存读取操作，而AfterAdjust类型的数据要进行2次内存读取操作。下面我分别对大小为1000万的这两种结构体的动态数组进行初始化，然后依次读取数组数据对每个数据成员做求和操作，得到的测试时间如下表。

内存地址对齐 - Stupid Bastard - 学习者的博客

从上面的测试数据可以看出，同样的数据成员，就是因为摆放的顺序不同而造成性能有28.127%的差异。因此调整好结构体内的数据成员的摆放顺序既可以减少内存的使用，又可以提高程序的性能。

2. 把一些字节数占用比较少的成员合并到字节数占用大的成员。首先看如下两个结构体：

struct UnMergeMember{

int a;

int b;

char c;

};

struct MergeMember{

int a;

union{

int b;

char c;};

};

UnMergeMember结构体由三个成员变量a，b，c，分别是int，int，char类型，按照地址对齐的规则，该结构体占用12个字节。因此初始化UnMergeMember类型变量涉及到3次内存读操作，3次赋值操作，3次内存写操作。MergeMember结构体由一个int类型的成员和一个联合体变量组成，按照地址对齐规则，该结构体占用8个字节。

联合体union{int b; char c;}占用4个字节，高位3个字节保存变量b(前提是用3字节能足够表示b的数据范围)，最低位1个字节保存变量c。假设定义一个MergeMember类型的变量为merge，初始化每个成员变量如下：

merge.a = some integer;

merge.b = some integer;

merge.b <<= 8;

merge.c = some char;

初始化一个MergeMember类型的数据只涉及到2次的内存读操作、3次赋值操作、1次位移操作，2次内存写操作。从上述可以看出初始化一个UnMergeMember类型的变量比MergeMember类型变量多了1次读操作和写操作，少了1次位移操作。下面我分别对大小为1000万的这两种结构体的动态数组进行初始化，然后依次读取数组数据对每个数据成员做求和操作，得到的测试时间如下表。

内存地址对齐 - Stupid Bastard - 学习者的博客

从上面的测试数据可以看出，在结构体中把小数据归并到大数据可以减少内存读取的次数，虽然多了一些CPU的操作，但是用CPU的操作换取内存数据读取次数，程序性能肯定能得到提高。上面的测试程序可以得到43.5%的性能提高(基本等于内存读取次数减少比(6-4)/4=50%)，在对性能要求特别高的系统中，这么大幅度的性能是相当可观。上述的例子也可以通过位段实现(Bit-fields)，但是位段只能对整数进行操作，如果把浮点数于和int类型的数据放在一起用位段实现显然不行，但是通过位移的方法也可以把char类型的数据并入浮点数float或是double中。

3. 通过位段(Bit-fields)的方式把一些整形数据按照各自需求的字段数来分配。这种方式可以大大节省空间，TCP协议的首部的定义就是采用位段的方式来定义的。位段的使用比较简单，这里我就不赘述了，可以参考相关的资料。但是值得注意的是，使用位段的方式的对齐方式也要遵守上述结构体对齐的方式，看下面一些结构体以及相应的大小：

struct BitField1{

char a:1;

char b:2;

char c:3;

char d:2;

};

struct BitField2{

char a:1;

char b:2;

char c:3;

char d:2;

int e:4;

};

Sizeof(BitField1)等于1(char大小的倍数)，sizeof(BitField2)等于8(int大小的倍数)。