本书将从算法、代码调优、并行编程、网络等方面，介绍如何打造高性能系统的方方面面。本章关注代码调优，以 C++语言为代表和例子。

inline 函数

inline 是 C++中的关键字。普通函数调用时存在压栈、退栈等操作，存在一定的开销。将函数声明为 inline 后，在调用该函数时，将用函数体替换之，避免这些开销。例如：

代码清单 x-y

```
inline int add(int a,int b)
{
    return a+b;
}
```

inline 可以用于解释 STL 中的 sort 为什么比 C 语言标准库函数 qsort 高效。由于 qsort 需要通过函数指针形式传入比较函数 cmp，对 cmp 的频繁函数调用产生较大开销；而 STL 中的 sort 则通过 inline 避免了这个问题。当然，sort 的高效还可能和基于快速排序实现时对小数组使用插入排序有关。

使用 inline 的时候需要注意如下事项：

1， 由于每次对函数的调用都替换为函数体，因此使用 inline 后程序的目标代码大小可能显著增大。

2， inline 只是对编译器的一种申请和建议，并不是强制。编译器可能并不理会，事实上，函数如果含有复杂的循环、递归等代码，几乎不会被 inline。因此适合将逻辑简单、语句较少的函数 inline。

3， inline 和宏表面上看效果相同，但其实有本质的区别。宏替换是预处理器(preprocessor)操作的结果，并不在编译链接过程中完成。由于宏本质上不是函数，因此在涉及运算优先级的情况下往往需要通过加入（大量）额外的括号来保证正确性。另外，宏容易产生副作用，尤其是涉及到增量运算符时。例如：

代码清单 x-y

```
#define MAX(a,b) (a>b)?a:b
int x=2;
int y=1;
cout<<MAX(x,y)<<endl;//line 1
```

cout<<MAX(++x,y)<<endl;//line 2

line 1 对 MAX 的使用将得到正确的结果；而在 line 2 中，由于宏的机制是机械替换，因此 line2 将被更改为 cout<<(++x>y)?++x:y<<endl;导致 x 自增两次，值从 2 变为 4 而非 3。

Cache 命中率优化

现代计算机的存储设备一般包括硬盘、内存、Cache。三者相比较，容量依次减少，速度依次提升，Cache 容量最小、速度最快、价格最高。由于容量有限，因此一般只在 cache 中存储近期频繁操作的数据，这样，当访问这些数据的时候，避免了从内存里载入数据这样相对昂贵的操作。当 Cache 满时，需要一定的策略和算法，将数据替换出去。而当需要访问的数据不在 Cache 中时，我们称 Cache 未命中；否则称 Cache 命中。Cache 命中率对一个程序的性能影响很大。

以按行和按列两种方式来访问二维数组是分析讨论 Cache 命中率对程序影响的一个经典例子。

代码清单 x-y

```
int f(int a[][],int m,int n) //m*n
{
int sum=0;
for(int i=0;i<m;i++)
{
   for(int j=0;j<n;j++)
{
      Sum+=a[i][j];
}
}
return sum;
}
int g(int a[][],int m,int n) //m*n
{
int sum=0;
for(int j=0;j<n;j++)
{
   for(int i=0;i<m;i++)
```

```
{
    Sum+=a[i][j];
}
}
return sum;
}
```

函数 f 和 g 都将返回二维数组 a 中所有元素之和。而 g 按照列的方式来访问数组，而 C++中的数组元素是按照行顺序来组织存放的。这样，当在 g 中访问 a[i][j]后，a[i][j],a[i][j+1],a[i][j+2]等少量数据可能被存放到 Cache 中，此时按列访问 a[i+1][j],a[i+2][j]等很可能发生 Cache 不命中(极端情况下，g 的 Cache 命中率为 0)，这大大影响了程序性能。因此，虽然 f 和 g 的时间复杂度都是 O(mn)，运算的次数也都相同，但是运行时间可能差别极大。

作为分析 Cache 命中对程序性能影响的另外一个例子，我们考察顺序搜索和二分查找算法。

```
bool binarySearch(int a[],int n,int target)
{
    int left=0;
    int right=n-1;
    while(left<=right)
{
        int mid=(left+right)/2;
        if(a[mid]<target) left=mid+1;
        else if(a[mid]>target) right=mid-1;
        else return true;
}
        return false;
}
bool bruteForce(int a[],int n,int target)// a and b are both ordered.
{
    for(int i=0;i<n;i++)
    {
```

```
            if(a[i]==target) return true;
        }
        return false;
    }
```

由于顺序查找连续访问数组 a，因此程序局部性较好；而二分查找是跳跃访问数组 a，因此 Cache 命中率可能不如顺序查找。因此，当 n 较小即数组 a 中元素较少时，时间复杂度较高的顺序查找(O(n))性能往往优于时间复杂度较低的二分搜索算法(O(lgn))。

这里顺便指出，正确、高效地实现二分搜索是一个很值得思考的问题，涉及到很多细节。例如，mid=(left+right)/2 可能存在溢出、移位代替除法等等，有兴趣的朋友可以参考其他书籍。

位运算

位运算主要包括与运算、或运算、非运算、异或运算、左移和右移等。位运算在不使用临时变量交换两个整数、murmurhash2 哈希函数、加密算法、去重、八皇后问题等场景中得到了广泛的应用。我们知道，将一个无符号整数乘以(除以)2，可以通过将该整数左(右)移 1 位来实现。由于乘法、除法操作较为昂贵，而移位运算较为高效，因此，通过位运算来优化相关计算是一种可行的方法。

有人对用左移代替乘法是否能改善性能表示怀疑，理由是认为开启编译器优化之后使用乘法和移位生成的机器代码可能是一样的(编译器可能会将乘法运算转为移位操作)。但是，移位本身作为一种技术，还是值得学习的，而且我们设计和实现代码时不能过分依赖编译器。

第三章将结合一个实际例子来介绍位运算的用处。

避免多余计算

程序中反复使用的、固定不变的一些数值，我们只需要计算一次，并用变量保存起来即可。需要使用的时候，直接读取该值，避免重复计算。

代码清单 x-y

```
void f(char *p, char *q)
{
    for(int i=0;i<strlen(p);i++)
        for(int j=0;j<strlen(q);j++)
            //use p[i] and q[j] here
}
```

由于循环每次迭代时，p 和 q 的长度不变，因此 f 中存在多余的计算，可以通过如下的方式优化:

代码清单 x-y

```
void g(char *p, char *q)
{
    int m=strlen(p);
    int n=strlen(q);
    for(int i=0;i<m;i++)
        for(int j=0;j<n;j++)
            //use p[i] and q[j] here
}
```

这种方法也可以视为空间换时间技术：以 O(1)常数空间换来了 O(n)时间，其中 n 是字符串长度。

分支预测优化

由于现代计算机所普遍采用的指令流水线技术，当分支预测失败时，需要清空流水线，重新载入正确分支的那些指令，成本巨大。因此，应尽可能降低分支预测失败的几率。具体的途径包括：合理安排 if-else 语句、避免多余 if 判断、合并判断条件等。

合理安排 if-else 语句：在写包含多个 if 语句的代码时，应把最不可能成立的情况写在最后，最可能的情况放在最先测试。例如，在二分搜索中，我们也可以这么实现：

```
bool f(int a[],int n,int target)
{
    int left=0;
    int right=n-1;
    while(left<=right)
{
        int mid=(left+right)/2;
        if(a[mid]==target) return true;//line 1
        else if(a[mid]>target) right=mid-1;//line 2
        else left=mid+1; //line 3
```

```
    }
        return false;
}
```

在 f 的循环体中，我们需要比较 a[mid]和 target 的大小关系。如果我们事先知道在很大的数组 a 中只有一个(很少)元素等于 target，我们期望 a[mid]==target 这个测试成立的机会很少，因此，为了优化分支预测，我们需要把这种最不可能发生的情况放在最后测试。

避免多余 if 判断：多余的代码总是应该避免的，多余的 if 判断更应该去除，尤其是当它出现在循环内部。例如：

代码清单 x-y

```
void f(int a[],int b[],int size)
{
    for(int i=0;i<size;i++)
    {
        if(x+y>n+m)
            //use a[i]
        else
            //use b[i]
     }
}
```

由于循环体内的 if 表达式不依赖于中间状态，在整个循环中的真值(true or false)保持不变，因此可以提取到循环外面，如下所示

```
void f(int a[],int b[],int size)
{
    if(x+y>n+m)
   {
    for(int i=0;i<size;i++)
    {
        //use a[i]
     }
```

```
    }
    else
    {
       for(int i=0;i<size;i++)
       {
            //use b[i]
       }
    }
    }
```

合并判断条件:对于如下形式的分支判断，可以进行相应的优化。

代码清单 x-y

```
if(a!=0 && b!=0 && c!=0)
{
     //…
}
```

部分编译器会对代码清单 x-y 中的 if 判断生成三条 cmp 汇编指令和相应分支代码，影响了效率。因此，可以改写为：

代码清单 x-y

```
int flag=a&&b&&c;
if(flag!=0)
{
     //…
}
```

循环展开

一般说来，适当地将循环展开可以避免管道阻塞和增加指令级并行性。例如，

代码清单 x-y

```
for(int i=0;i<=99999995;i+=4)
{
    a[i]=2*a[i]+3;
```

```
    a[i+1]=2*a[i+1]+3;
    a[i+2]=2*a[i+2]+3;
    a[i+3]=2*a[i+3]+3;
}
```

这里，循环展开的步长是 4。在实际系统中，一般需要根据实验等方法确定最佳步长。有时候，循环展开并不能带来性能提升。另外，当数组大小不能整除步长时，特别需要留意边界元素处理。

避免频繁 new/delete

在 C++中我们可以通过 new 来动态分配内存，但有时候频繁分配内存会成为程序瓶颈，应予以避免。考虑如下代码

```
int* f(int n)
{
    int *p=new int(n);
    //do with p
    return p;
}
void g()
{
   //…
   f();
   //…
}
```

如果 f 是一个性能需求较高的函数(可能直接和用户交互)并且被频繁调用，在 f 中的 new 操作容易造成 f 执行时间的大大增加。因为我们知道，new 运算符在被使用的时候，将进行两个动作:

1.调用堆(heap)内存分配函数获得一块内存空间。

2.初始化内存空间。如果该空间用于存放用户自定义对象，则类的构造函数将被调用。

而且，如此实现 f 还存在一个(具有争议的)接口设计问题：返回的动态内存分配空间，该由谁归还？显然，f 不会完成这个动作。那么调用 f 的函数 g 可能会忘记这件事情，毕竟，内存不是 g 分配的。因此，如果 f 是性能攸关并被频繁调用的函数，那么可以改写为:

```
void f( int *& p ,int n)
{
      //do with p
      return ;
}
void g()
{
    //…
    int *p=new int(n);
    f(p,n);
    delete [] p;// garbage collection
    //…
}
```

避免在性能攸关的函数中频繁使用 new 分配内存的另一种方法是使用内存池。程序在启动后，即分配一定量的内存，构成内存池。内存池中的内存，可能按照程序需求，分为 2KB、16KB、1MB、2MB 等不同大小的块(可能有 5 个 2KB 大小的块，3 个 1MB 大小的块，等等)。当函数需要内存时，由专门的一个程序负责从内存池中选出一个合适大小的内存块，分配给该函数。

通过内存池等方式来手工分配和回收内存的技术，在很多大型项目中都有使用到。程序员不允许自己手动地通过 new 和 delete 来分配、回收堆内存，系统所使用的内存分配回收，由一个专门的全局内存管理器负责。当程序需要内存时，向管理器发送请求即可。手工管理内存的好处是显而易见的：它解放了程序员，程序员不需要为内存分配和回收花费巨大精力，担心内存泄露；同时，由于应用程序最清楚自己的行为，因此这种定制化的方法可能会使程序可以更灵活、高效地掌握、使用内存资源。

手工管理内存需要精心设计和实现块大小分配、回收算法，处理内存碎片等。

避免临时对象

在编写 C++程序中，临时对象就像幽灵一样，防不胜防。临时对象不存在于纸面上，看不见摸不着，但是编译器可能暗中生成它，因此我们得为此付出相应的构造函数、析构函数调用成本等，这在很多时候将严重影响程序效率。这节中，我们对避免临时对象相关方法，进行全面的讨论。

代码清单 x-y

```
class ToyClass
{
public:
    ToyClass()
    {
        this->x = 0;
        this->y = 0;
    }
    ToyClass(const ToyClass& a)
    {
        this->x = a.x;
        this->y = a.y;
    }
private:
    int x;
    int y;
};
ToyClass operator + (const ToyClass& a, const ToyClass& b);
```

如果我们调用 ToyClass c=a+b;在未优化的情况下，编译器将使用临时对象存储 a+b 的值，然后通过 ToyClass 的拷贝构造函数(注意到这里是初始化，不是赋值)，完成对象 c 的创造，接着调用析构函数，临时对象被销毁。下面介绍如何通过各种方法来避免临时对象的生成。

用+=代替+

一种非常容易想到的方法是用+=来代替+。

代码清单 x-y

```
class ToyClass
{
public:
    ToyClass operator += (const ToyClass& a)
    {
```

```
        this->x += a.x;
        this->y += a.y;
    }
private:
    int x;
    int y;
};
```

改写 ToyClass c=a+b;为

代码清单 x-y

```
ToyClass c;
c+=a;
c+=b;
```

改写后，临时对象将不存在。这种方法的缺点是明显的：本来只需要一行代码完成的功能，现在却通过三行来实现，程序的可读性和可维护性大大降低，因此我们可以考虑用其他方法。

NRV 优化

为了尽可能让编译器实施一种称为 NRV 的优化，我们可以这么实现 operator +

代码清单 x-y

```
class ToyClass
{
public:
    ToyClass()
    {
        this->x = 0;
        this->y = 0;
    }
    ToyClass(int x,int y)
    {
        this->x = x;
        this->y = y;
```

```
    }
    friend ToyClass operator + (const ToyClass& a, const ToyClass& b);
private:
    int x;
    int y;
};
ToyClass operator + (const ToyClass& a, const ToyClass& b)
{
    return ToyClass(a.x + b.x, a.y + b.y); //末尾调用构造函数直接返回
}
```

当我们调用 ToyClass c=a+b;时，编译器开启 NRV 优化后，编译器会对我们的代码进行调整和优化，优化后的代码类似于：

代码清单 x-y

```
void operator + (const ToyClass& a, const ToyClass& b, ToyClass &out) //多了一个输出参数
{
    out.x = a.x + b.x;
    out.y = a.y + b.y;
}
```

同时，ToyClass c=a+b;将被改写为类似于如下形式:

```
ToyClass c;
operator +(a,b,c);//直接将结果作用于 c 上
```

这种方法较为依赖编译器。毕竟，编译器是否开启优化、优化到什么程度我们不一定心里有数。而且，它的可移植性也相对较低，因为可能不是所有的编译器都会实施这种优化。

利用移动构造函数避免临时对象

准确说，利用移动构造函数这种方法并没有避免临时对象：临时对象还是生成了，但是我们在创建新对象的时候，将直接利用临时对象的资源，将资源“窃取”出来。也就是说，此时，资源的拥有者和控制权发生了变化，从临时对象神奇地转移到新对象上，然后临时对象被析构。偷梁换柱！因此，这节的标题可以理解为“利用移动构造函数避免临时对象带来的开销”。

在具体描述做法之前，我们有必要学习下 C++ 11 中的右值引用以及复习下左值以及左值引用。

什么是左值？通常有两种定义和描述：

左值是可以放在赋值运算符左边的东西

左值是可以寻址(可以实施取地址运算符&)的东西

一般说来，左值标示一个对象，而右值标示一个对象的值。变量是常见的左值，而常数和临时对象是常见的右值。

左值引用就是必须绑定到左值的引用；右值引用就是必须绑定到右值的引用。

例如:

代码清单 x-y

```
int    x;//x 是左值

const int y=3;//3 是右值

int& y=x;//y 是左值引用，用一个“&”

int&& z=3;//z 是右值引用，用两个“&&”
```

由于右值引用可以绑定到临时对象，因此，可以利用这点性质来高效完成一些事情。实际上，C++ 11 提供了移动构造函数。顾名思义，和拷贝构造函数相比，它并不拷贝资源，而仅仅是移动资源、“窃取”资源。为了更好的描述移动构造函数的作用，我们使用另外一个例子。

……因版权问题……. 后面就不发了吧…….看到这里的都是真爱…….