高效编程十八式

王伟冰

目录

导言：编程五大原则

本文讨论的是如何提高编程的质量和效率，涉及编程的十八个方面：类，函数，泛型，委托，封装，约束，继承，多态，宏思想，语法糖，命名，陷阱，异常，性能优化，多线程，代码编辑，测试，调试。

示例代码主要是用 C++写的，但是我所讨论的原则同样适用于其它语言。

我根据自己实际编程的经验，以及阅读过的编程书籍，总结出编写代码的五条基本原则：简洁，安全，快速，灵活，清晰。下面用一些小小的例子，来接触一下这五个原则。

我们在编程的时候，经常会遇到常量，比如说圆周率，我们可以这样写：

```
const double PI=3.14159265;
```

这样做的第一个好处就是，每一次用到圆周率时，只需要写 PI，就不用写什么 3.14…一大串了，这就引出了第一个原则：

简洁原则：写出来的代码要尽量简洁，避免重复。

使用 PI 的另一个好处是，不容易出错。如果我们每一次都写 3.14…，那么有可能某一次会不小心写错了哪位数字，导致计算结果不正确，这种错误很难被发现。但是用 PI 的话，即使我们不小心把 PI 拼错了，编译器也会提醒我们。这就是第二个原则：

安全原则：写出来的代码要不易出错，易于查错。

那么，为什么要写成 const 呢？直接写 double PI=3.14159265 不也可以吗？这涉及到性能问题，const 型的变量编译时就能确定它的值，而非 const 型的变量在运行才能得到它的值，执行速度会慢一些。于是引出第三个原则：

快速原则：写出来的代码要快速运行，尽量省时。

再来看另一个例子，比如你想输入 10 个数，然后排好序输出来：（省略具体过程）

```
const int Length=10;
int a[Length];
for(int i=0;i< Length;i++) ……//逐个输入
……//排序
for(int i=0;i< Length;i++) ……//逐个输出
```

为什么要用 Length 来代替 10 呢？这不是违反了简洁原则吗？但是想想，假如你还想试一试输入 20 个数的情况，那么你只需要把第一句中的 10 改成 20 就可以了。如果全部都直接写 10，那就一个个都要改成 20，麻烦不？所以就有第四个原则：

灵活原则：写出来的代码要易于更改，易于扩展。

那么，为什么要用 Length 呢？直接写 L 不是更简洁吗？简洁是简洁，但是写成 L 不能准确体现出长度这个意义，让人无法一眼看出这个变量的作用。这是第五个原则：

清晰原则：写出来的代码要意图清晰，易于阅读。

以上五个原则，不分优先级，同等重要。如果它们出现冲突，该如何取舍，取决于实际情况。随后的章节会详细地介绍这些原则在具体编程中的应用。

复数运算：类与函数

有时候我们会在程序中用到复数运算，我们一般会先定义一个复数类：

```
class complex{
public:
    double x,y;
};
```

假如我们需要把两个复数相加，比如说有三个 complex 型变量 a、b、c，要把 a 和 b 相加的和赋给 c。我们可以这样做：

```
c.x=a.x+b.x;
c.y=a.y+b.y;
```

假如我们每次做复数加法都要写这么两行，那显然太麻烦，不妨写一个函数：

```
void add(const complex &a,const complex &b, complex &c){
    c.x=a.x+b.x;
    c.y=a.y+b.y;
}
```

这样每次做加法只要写 add(a,b,c)就可以了，所以，**把经常用到的代码段写成一个函数，可以简化代码。（简洁原则 1）**（请注意红色的定语，如果不是经常用到，就没必要写成函数，写成函数反而会制造麻烦。后面还会有这种红色的定语，就不一一解释了。）

用 const 修饰的函数参数表示这个函数不会修改此参数的值。

也可以把 add 写成 complex 类的一个成员函数：

```
class complex{
public:
    double x,y;
    void add(const complex &a,const complex &b){
        x=a.x+b.x;
        y=a.y+b.y;
    }
};
```

这样就可以用 c.add(a,b)来做加法了，对比 add 函数的两种实现，你可以发现，后一种更简洁一些，因为不用再传 complex &c 参数，而且 c.x 和 c.y 也可以直接写成 x 和 y。所以**把函数定义为与之密切相关的类的成员函数，可以简化函数实现的代码。（简洁原则 2）**

无论是 add(a,b,c)还是 c.add(a,b)，都已经够简洁了，但是从清晰原则的角度来看，它们还不合格，因为你无法一眼看出是哪两个加起来等于第三个。如果是 c=add(a,b)，是不是比较直观呢？

```
complex add(const complex &a,const complex &b){
    complex c;
    c.x=a.x+b.x;
    c.y=a.y+b.y;
    return c;
}
```

甚至可以用运算符重载：

```
complex operator +(const complex &a,const complex &b){
    ……//同上
```

这样就可以用 c=a+b 来表示复数加法了，这是最简洁也最清晰的一种形式。所以，**一个函数的函数名、参数个数和类型、返回值类型应该能够自然地体现出这个函数所要实现的操作。（清晰原则 1）**

上面的方法似乎已经非常完美了，但是其实还有问题：性能问题。在 add 函数里，我们定义了一个局部变量 c，然后把 c 作为返回值，实际上返回的是 c 的一个副本，这种复制导致了不必要的开销。尽管复制一个复数所需的时间微不足道，但是推而广之，假如是做矩阵加法呢？矩阵 C=add(A,B)，如果 C 是 100*100 的矩阵，那么就要把 C 的 10000 元素都进行复制，这时间可就不能忽视了。其实，我们在传入参数 a 和 b 时，用的是引用传参（complex &a,complex &b），而不是按值传参（complex a,complex b），就是为了避免在传参数过程中多余的复制运算。所以要**尽量减少对变量的复制。（快速原则 1）**

如何解决返回时多余的复制操作？这是 C++独有的一个缺陷，Java 和 C#并不存在这个问题。在 Java 和 C#里，返回的只是引用，并不需要复制。C++也可以返回引用啊，比如这样：

```
complex& add(const complex &a,const complex &b){
    ……//同上
```

但是，这样做并不能如愿，因为返回的是对局部变量 c 的引用，局部变量在函数返回的时候就会自动销毁，也就是说，返回的是对一个不复存在的变量的引用！这样带来的后果，是不可预测的。所以，**要保证每个指针或引用不指向实际并不存在的变量。（安全原则 1）**（引用的实质就是指针）

局部变量是在栈中创建的，所以函数一返回就自动销毁，那可不可以在堆中创建变量 c？比如这样：

```
complex& add(const complex &a,const complex &b){
    complex* pc=new complex();//在堆中创建变量 c
    pc->x=a.x+b.x;
    pc->y=a.y+b.y;
    return *pc;
}
```

实际上，Java 和 C#就是这样做的。但是 C++这样做会有一个严重问题：你在堆中创建了变量，什么时候回收？如果不回收，则会浪费内存（或者叫内存泄漏），如果要回收，则每次用完这个返回值之后都要自己手动回收，比如说：

```
complex& c=add(a,b);
……//用 c 进行其它运算
delete &c; //手动回收
```

这简直就是麻烦。Java 和 C#以及其它很多高级语言都有自动垃圾回收机制，所以不需要手动回收，但是 C++却只能这样。总之，**在堆中创建的变量，要保证能回收。（安全原则 2）**

现有 C++很难解决这个问题，不过，即将发布的 C++新标准，加入了“右值引用”的新特性，可以完美地解决这个问题。网上有很多资料，具体我就不介绍了。

在现有的情况下，只能根据实际需要进行折衷设计，一般牺牲快速原则用 c=a+b。

也许你会觉得我无聊，不就是一个复数加法吗，几行代码就完事，用得着讨论这么多吗？其实我只是想借用这个最简单例子来阐述一些普适性的原则。当你运用这些原则解决了复数加法的问题之后，你会发现同样的方法可以用来实现复数减法、乘法、除法，甚至矩阵的加减乘除，集合的交、并、差，……很多很多。

数据统计：泛型与委托

我们常常会遇到这样的问题，比如说，统计一个班的学生中数学成绩大于 60 分的人数。假如说所有学生的成绩储存在一个 int 型数组中，那么我们可以定义这样的函数：

```
int count(int scores[],int n){
    int m=0;
    for(int i=0;i<n;i++)
        if(scores[i]>60)m++;
    return m;
}
```

其中 scores 中储存学生成绩的数组，n 是数组的长度。

于是我们就可以用诸如 count(a,30)这样的代码来统计一个 30 人的班上的及格人数了。

但是，这样的一个函数，限死了只能统计大于 60 分的人数，不能统计大于 70 分、80 分的人数，所以我们可以把函数改成这样：

```
int count(int scores[],int n,int min){
    int m=0;
    for(int i=0;i<n;i++)
        if(scores[i]>min)m++;
    return m;
}
```

这样不就提高了灵活性了吗？所以，**对于有可能改变的数值，不要写死，可以作为函数参数传进来。（灵活原则 1）**

但是可能过几天之后，你发现有的学生的成绩不是整数，可惜你的这个函数只能处理整数的情况，只好另写一个：

```
double count(double scores[],int n,double min){
    ……
```

麻不麻烦？其实，如果从一开始就预料到类型可能会发生改变，那么不妨把函数定义成模板：

```
template<class T>
int count(T scores[],int n,T min){
    int m=0;
    for(int i=0;i<n;i++)
        if(scores[i]>min)m++;
    return m;
}
```

其中 T 称为模板参数，如果 T 是 int，那么就得到前面 int 版本的 count 函数；如果 T 是 double，就得到前面 double 版本的 count 函数。比如这样调用：

```
m=count<int>(a,30,60); //调用 int 版本
m=count<double>(b,40,60); //调用 double 型版本
m=count(c,30,60); //编译器会根据 c 的类型自动推断出用哪个版本
```

所以，**对于有可能改变的类型，不要写死，可以写成模板，把类型作为模板参数传进来。包括类模板和函数模板。（灵活原则 2）**这就是 “泛型编程”。而且这样也体现了简洁

原则，因为不需要对每种类型分别写一个单独的版本。

但是，问题还没有完，有时我们不仅需要统计成绩大于某个值的人数，还要统计小于某个值的人数，或者是介于某两个值之间的人数。考虑到要**尽量利用已有的类和函数来构造新的功能，而不改变类和函数的内部代码。（灵活原则 3）**你可以这样子来实现：

```
m=30-count(a,30,60); //总人数减去大于 60 的人数得到小于 60 的人数
m=count(a,30,60)-count(a,30,70); //大于 60 的人数减去大于 70 的人数得到介于 60 和 70 之间的人数
```

尽管这种方法很巧妙，在很多情况下是一种实用的思路，但是这毕竟是一种治标不治本的做法（而且性能不好），假如我要你统计出成绩是奇数或偶数的人数呢？无能为力了吧。根本的方法是使用函数指针：

```
template<class T>
int count(T scores[],int n,bool func(T)){ //func 代表一个函数指针
    int m=0;
    for(int i=0;i<n;i++)
        if( func(scores[i]) )m++;
    return m;
}
bool func1(double x){ //判断 x 是否大于 60
    return x>60;
}
bool func2(int x){ //判断 x 是否是偶数
    return x%2==0;
}
```

这样你就可以这样子做统计：

```
m=count(a,30, func1); //统计分数大于 60 的人数
m=count(a,30, func2); //统计分数是偶数的人数
```

其中 bool func(T)表示一个名为 func 的函数指针，它指向一个函数，这个函数的参数为 T 型，返回值为 bool 型。当你把 func1（或 func2）作为参数传给 count 函数时，func 指针就指向了 func1（或 func2）函数。当执行到 func(scores[i])时，实际上就执行了 func1(scores[i])（或 func2(scores[i])）。

所以，**当一个函数里有一段有可能改变的代码时，不要写死，可以把这段代码委托给一个传进来的函数指针去执行。（灵活原则 4）**这种思想就叫做“委托”。

回顾前面的灵活原则 1，我们用一个 min 参数来代替 60，从而提高了灵活性和简洁性。我们希望 func1 函数也能做到这一点，能够判断 x 是否大于一个设定的数值 min.于是对它进行了改造：

```
bool func1(double x,double min){
    return x>min;
}
```

这样做对吗？那调用 count 怎么调用？count(a,30,func(,60))？？？

没有任何语言支持这种语法，count 要求传入的函数指针必须是 bool func(double)型的，也就是接受一个 double 型参数并返回 bool 型的，所以 func1 不能接受两个参数。解决方法

可以是这样：

```
double min;
bool func1(double x){
    return x>min;
}
```

我们可以在调用 count 之前先设定 min 的值：

```
min=60;
m=count(a,30,func1);
```

可惜的是，min 变量变成了一个全局变量，所有的函数都可以访问它，通常我们要**严格控制变量作用域，只有真正需要用到某个变量的代码段才可以访问到这个变量。（清晰原则 2）**你可能觉得这没什么大不了，但是在大项目里，过多不必要的全局变量可能会导致不经意的命名冲突，使得在别的地方对另外一个变量的访问不小心变成了对这个变量的访问。即使不存在任何同名现象，在多线程的程序里对全局变量的异步访问也会导致意想不到的结果，这个后面还会说到。

更好的实现应该使用函数对象，我们需要更改 count 和 func1 的定义：

```
template<class T,class T1>
int count(T scores[],int n,T1 func){ //func 的类型未定
    int m=0;
    for(int i=0;i<n;i++)
        if( func(scores[i]) )m++;
    return m;
}
class Func1{
    int min;
public:
    Func1(int x){ //构造函数
        min=x;
    }
    bool operator()(double x){ //重载()运算符，判断 x 是否大于 min
        return x>min;
    }
}
```

于是就可以这样子：

```
m=count(a,30,Func1(60)); //统计分数大于 60 的人数
```

怎么这么复杂？这个过程发生了什么事？或许这样写会清楚一些：

```
Func1 func1(60);
m=count<double,Func1>(a,30,func1);
```

首先，Func1 是一个类型，而不是一个函数。Func1 func1(60)调用了 Func1 类的构造函数，新建了一个 Func1 型的对象 func1，并把它的成员变量 min 设为 60。然后把 func1 传给 count 函数的第三个参数，即 T1 func。所以 T1 就是 Func1，func 就是 func1。

当执行 func(scores[i])的时候，就相当于执行 func1(scores[i])，即调用了 Func1 类的()运算符，把 score[i]作为参数 x 传入，返回 x>min 即 score[i]>60 的值。在这里，func1 对象就叫做函数对象，因为它本质是一个对象，却表现得像一个函数指针。

太麻烦了。为了增加灵活性得写这么长一个 Func1 类。有没有简单一点的办法？有。可以利用 C++内置的函数对象（需要#include <functional>）：

m=count(a,30,bind2nd(greater<double>(),60));

这样就不用写 Func1 类了。但是这个实在很难看懂，我也不打算在这里解释它的意思……你猜，如果要统计 60 到 70 间的人数该怎么写？

m=count(a,30,logical_and(bind2nd(greater<double>(),60),bind2nd(less_equal<double>(),70)));

也许有的人会觉得这个很酷，但是你看一下 C#是怎么写的：

m=a.Count((x)=> x>60 && x<=70);

简洁吧。“(x)=> x>60 && x<=70”这个东西叫做“lambda 表达式”，代表一个匿名的函数，前面的“(x)”表示这个函数有一个参数 x，“=>”是 lambda 表达式的标志，后面表示函数返回 x>60 && x<=70 的计算结果。上面的写法是简化的写法，完整的是：

m=a.Count((double x)=>{ return x>60 && x<=70; })

所以 lambda 表达式是一个非常有用的东西，我认为现代的编程语言都应该包含 lambda 表达式。事实上，C++新标准也包含了 lambda 表达式，你可以这样写：

m=count(a,30,[](double x){ return x>60 && x<=70; });

可以看到，和 C#的完整写法其实大同小异，只是“=>”变成了“[]”，并且提到前面而已。总之，**如果需要对函数指针进行更加灵活的定制，可以使用函数对象或者 lambda 表达式。（灵活原则 5）**

矩阵类：封装与约束

矩阵的元素可以用一个数组来储存，我们希望建立一个矩阵类，来封装各种矩阵操作（比如加减乘、求逆、转置等等）。我们可以这么写：

```
class matrix{
    int width,height; //矩阵的宽度和高度
    double* data; //储存矩阵数据的数组
public:
    matrix(int w1,int h1){ //构造函数
        width=w1;
        height=h1;
        data=new double[w1*h1];
    }
    void add(const matrix& m1,const matrix& m2){ //加法
        for(int i=0;i<width*height;i++)
            data[i]=m1.data[i]+m2.data[i];
    }
    ……//其它矩阵操作，不一一写出
};
```

在这里我们把 width、height 和 data 都设为私有变量，因为我们不希望使用 matrix 类的代码对这些变量作随意的更改，**如果外界代码对类的某个成员变量或函数的随意访问可能导致不好的结果，则应该把它设为私有成员。（安全原则 3）**随意更改矩阵的高度、宽度或数据的储存位置，都可能导致原有数据的丢失。

现在我们还不能对矩阵的具体元素进行访问，可以通过在 matrix 类中重载()运算符来实现：

```
double& operator () (int i,int j){ //返回第 i 行第 j 列元素
    return data[i*width+j];
}
```

由于返回类型是引用类型，所以不仅可以读取，还可以对它进行赋值：

```
matrix m(3,3);
double x=m(0,1); //读取第 0 行第 1 列元素
m(0,1)=x; //设置第 0 行第 1 列元素
```

这样，我们提供了对矩阵数据的封装，虽然外部代码无法直接访问矩阵数据，但是可以通过()运算符对它进行访问。

现在我们的矩阵类看似可以做很多事情了，但实际上它有很多漏洞。比如，在构造函数里面用 new double[w1*h1]分配了内存，但是却没有考虑它的回收，我们可以在 matrix 类中定义析构函数，让它自动回收：

```
~matrix(){
    delete[] data;
}
```

当一个 matrix 型变量销毁时（比如说，局部变量在函数返回时会自动销毁），就会调用析构函数~matrix()，释放 data 指向的内存。

然而这样还不够，假如我这样用，会发生什么事？

```
matrix m1(3,3);
matrix m2(m1); //调用 matrix 类的复制构造函数
m2=m1; //调用 matrix 类的=运算符
```

等等，我们并没有定义什么“复制构造函数”和“=运算符”啊？这是 C++为每一个类自动生成的，这两个函数都会自动地复制 m1 中变量的值到 m2 里去。上面代码的本意应该是把 m1 的数据复制给 m2，但实际上只是复制 data 指针的值，也就是说，m1、m2 里的 data 指针指向的是同一个地址，更改任一个矩阵的元素值也会同时更改另一个矩阵的元素值。而且，当这两个 matrix 对象销毁时，它们的析构函数会分别被调用，执行 delete data 操作，同一个 data 地址被 delete 两次，这样很可能会使程序崩溃。

所以 C++自动生成的这两个函数并不是什么好东西。解决方法是把它们显式地实现为复制数据，而不是复制指针：

```
public:
    matrix(const matrix& m1){
        width=m1.width; //复制宽度
        height=m1.height; //复制高度
        data=new double[width*height]; //新分配内存
        for(int i=0;i<width*height;i++)
            data[i]=m1.data[i]; //复制数据
    }
    void operator = (const matrix& m1){
        delete[] data; //先删除原来的数据所占用的内存
        ……//同上
    }
```

所以，**复制或传递函数参数时，要清楚复制或传递的是指针（引用）还是数值。（安全原则 4）**在 C#中，对于 struct 是复制数值，而对于 class 是复制引用。Java 中除了内置类型，所有都是引用。

还有一个漏洞是，访问矩阵元素时可能越界，比如：

```
matrix m(3,3);
m(0,3)=0; //列数越界
m(-1,0)=0; //行数越界
```

我们希望它在越界的时候能够提醒我们，以便我们检查代码中是否有错误：

```
double& operator () (int i,int j){
    assert(i>=0 && i<width && j>=0 && j<height);
    return data[i*width+j];
}
```

assert 的作用就是，如果传给它的参数为 true，那么什么事都没发生，如果为 false，那么程序就会终止，或者进入调试状态。（需要包含 assert.h 头文件）

所以，**要确保每个数组的下标访问不会越界，每个函数的输入参数都合法。（安全原则 5）**

void add(const matrix& m1,const matrix& m2)这个函数的输入参数就有可能不合法，要保证两个输入矩阵的宽与高都和当前矩阵一样：

```
    assert(m1.width==width && m1.height==height
```

```
        && m2.width==width && m2.height==height);
```

总之，为了防止外部代码随意访问而可能导致的不良后果，我们应当对这些访问加以约束。

再来考虑一下矩阵的转置，比如：

```
void transpose(matrix& m1){
    assert(m1.width==height && m1.height==width);
    for(int i=0;i<height;i++)
        for(int j=0;j<width;j++)
            data[i*width+j]=m1.data[j*height+i]; //把 m1(j,i)复制到 当前矩阵(i,j)里
}
```

把 m1 的转置矩阵储存到当前矩阵里，这样做没错吧？可是，如果想把一个矩阵的转置储存到它自身（m 是 n*n 矩阵）：

```
m.transpose(m);
```

并不能得到正确的结果。因为在 transpose 函数里，data 和 m1.data 引用的是同一个地址。考虑 m(0,1)和 m(1,0)，即 data[1]和 data[n]这两个元素的对调，先是 data[1] = data[n]，然后再 data[n] = data[1]，结果两个数都是 data[n]原来的值。这就好比交换 a 和 b，不能 a=b,b=a，而应该引入一个临时变量：t=a,a=b,b=t。所以这里也要引入一个临时变量：

```
matrix m1=m;
m.transpose(m1);
```

所以**同时使用两个指针（或引用）时，要考虑两个指针（或引用）引用同一个变量的情况。（安全原则 6）**这里就是因为 data 和 m1.data 引用的是同一个地址。

于是这个矩阵的使用就多了一个规定：不能将自身对象传入 transpose 函数。可是让一个矩阵变成自己的转置这是一个很自然的事情，这样子的规定就给人感觉很别扭。这种规定我们把它称为“无理规定”。其实我们只需稍改一下 transpose 函数：

```
void transpose(matrix& m1){
    matrix* pm1=&m1;
    if(&m1==this) //如果 m1 就是自身，那么就把 m1 复制一份出来
        pm1=new matrix(m1);
    ……//同上，m1.data 改为 pm1->data1
    if(&m1==this) //记得删除这个副本
        delete pm1;
}
```

就可以让自身转置了。总之，**尽量减少对类和函数的用法的无理规定。（清晰原则 3）**

注意，这里 transpose 函数设计成这样，只是为了引出上面的两条原则，这种设计并不是好的设计，好的设计应该是像 m=m1.transpose()这样。

你有没有发现，本节提到的安全原则，基本上都是以牺牲灵活性为代价的，要么是限制访问类的某个成员，要么是限制数组下标或函数参数的范围，要么是限制两个指针不能指向同一变量。

面向对象程序设计的教科书往往很推崇这种限制，或者叫做“封装”。但是我认为，这种限制有时候是没有必要的。比如我写这个 matrix 类只是为了给我自己用，我自己很清楚它是怎么实现的，所以我也不会笨到去乱改它的成员变量，不会笨到去传给它一个不合法的参数，那我干嘛还要自己限制自己呢？

而如果这个 matrix 类是给别人用的，就如同 printf、cout 是别人写给我们用的一样，那么封装限制就是必要的了。我们无法保证别人是否会合法地使用这个类，所以必须加以限制。

所以，**对于只在模块内部使用到的代码，访问限制松一些，参数检查松一些，类和变量的命名随意一些。会被外部使用到的代码则相反。（灵活原则 6）**

事物总有好处和坏处，对一个类进行封装和约束，虽然麻烦，但是以后你就可以在上层代码放心使用这个类了；不进行封装和约束，你在上层代码就得小心使用，不过只要你自己有把握，就可以更灵活地使用这个类。这就好比 C/C++中的指针，其它高级语言都没有指针，因为使用指针很容易出问题；但是如果你指针用得很熟了，你可以用指针实现其它语言无法实现的强大功能。

形体建模：继承与多态

我们经常会用计算机对一些现实的事物进行建模，比如说用代码来描述一些形体。我们可以用边长来描述一个立方体，这样就可以求出它的体积：

```
class cube{
public:
    double length;
    double volume(){
        return length*length*length;
    }
};
```

我们同样可以用半径来描述一个球体：

```
class sphere{
public:
    double radius;
    double volume(){
        return 4*PI*radius*radius*radius/3;
    }
};
```

现在我们有若干个立方体和若干个球体，假设我们要求出它们的总体积，一种方法是分别求立方体和球的总体积，再把这两个总体积加起来。但是这样比较麻烦，我们希望能够统一地对待立方体和球体，所以我们可以为它们建立一个共同的基类 object：

```
class object{
public:
    virtual double volume();
};
class cube:public object{ //让 cube 继承 object
……
};
class sphere:public object{ //让 sphere 继承 object
……
};
```

然后就可以用 object 类来统一代表 cube 和 sphere 了。比如：

```
cube c1;
object* p=&c1; //将 cube 类指针转换成 object 类指针
double v=p->volume(); //调用 cube 类的 volume()函数
sphere s1;
p=&s1; //将 sphere 类指针转换成 object 类指针
v=p->volume(); //调用 sphere 类的 volume()函数
```

也就是说，通过基类的指针可以调用不同子类的函数，这种特点称为“多态性”。但是实现多态性有两个前提，一是基类中要给函数加上“virtual”修饰符，这样的函数称为“虚函数”；二是必须通过指针或引用调用虚函数，如果不用指针或引用，比如 object p=c1;p.volume();只是简单地调用基类中的 volume 函数。

现在我们可以把所有立方体和球的指针一起放到一个 object*型的数组里：

```
object* objects[10];
……//其它操作
double v=0;
for(int i=0;i<10;i++)
    v+=objects[i]->volume();
```

这样，我们用简单的几句代码求出了所有物体的总体积，而不需要管每个物体究竟是什么形状。因此，**对于内部实现不同但对外接口相同的类成员函数，可以建立共同的基类虚函数（或 Java 和 C#中的接口），从而可以在外部代码中统一处理，简化外部代码。（简洁原则 3）**

现在我们的形体类都只有形状信息。我们可以还希望给它增加密度信息，这样可以求出质量：

```
class object{
public:
    double density; //密度
    double mass(){ //返回质量
        return volume()*density;
    }
    virtual double volume();
};
```

注意到 mass 是“实函数”，并不用加 virtual，因为对于每一种形体，计算质量的方法都是一样的：密度乘体积。所以我们不需要在 cube 类和 sphere 类中分别定义 density 和 mass()，而是把它们放在基类 object 中定义，于是在子类中就都可以使用这些变量和函数了。所以，**对于内部实现完全一致的类成员变量和函数，可以放在基类中统一实现，简化内部代码。（简洁原则 4）**

于是我们看到，继承实际上分为两种，一种是实继承，继承的是内部实现，用于简化内部代码；一种是虚继承，继承的是对外接口，用于简化外部代码。

但是，继承的时候一定要注意，当类 B 继承类 A 时，类 B 会全盘继承类 A 的变量、实函数和虚函数。所以，当且仅当类 B 确实需要 A 的所有变量、实函数和虚函数时，才可以让 B 继承 A。如果 B 只需要 A 的一部分，应该让 B 和 A 继承自一个共同的基类。

尽管面向对象的程序设计思想受到了很多人的推崇，但是程序中的类和现实中的事物毕竟不是完全对应。在现实中 B 属于 A 并不代表着 B 就必须继承自 A。比如正方形属于长方形，但是让正方形类继承自长方形类并不是一种好的设计。正方形类只需要边长一个变量，而长方形却需要长和宽两个变量，让正方形全盘继承这两个变量是一种多余，而且还得采取必要的手段确保这两个变量始终相等。如果想要统一地对正方形和长方形求面积，不如让正方形和长方形共同继承一个可以求面积的虚函数：

```
class has_area{
public:
    virtual double area();
}
```

不过，让“带有颜色的长方形”去继承“长方形”就不会错，因为前者完全是后者的扩展，所以需要继承后者的所有信息。

类的继承和多态，只有能够真正简化代码时，才是有用的。一些面向对象程序设计的教材总喜欢举一些不切实际的例子。比如商店里买东西，每一种不同的商品都继承自“商品”这个类，每种商品都定义了自己的 getname()函数和 getprice()函数来获取它的名称和价格。问题是我何必这么麻烦？我只需要一个商品类就可以了：

```
class commodity{
public:
    char name[10];
    double price;
}
```

然后每种不同的商品就是 commodity 类的不同实例，具有不同的 name 和 price 而已。没错，“牙刷”是一种“商品”，但“牙刷”不一定要成为“商品”的子类，它可以是“商品”类的一个实例，因为“牙刷”和其它的商品的区别只是数据不同（name 和 price 都只是数据），而不是函数不同（立方体和球就是函数不同，因为它们求体积用的函数根本不一样）。仅仅是函数不同还不代表一定要用多态（因为使用函数指针也可以把不同的函数统一对待），还要求这里的函数不是全局函数而是类的实例成员函数，它和类的其它成员有着紧密的关联（比如立方体求体积与其边长有紧密关联），这种情况下使用多态是最合适的。TopLanguage 上不时有 GP（泛型编程）和 OO（面向对象）之争，我认为全局函数用 GP，如第 2 节所示，类的实例成员函数用 OO，如本节所示，而类模板（或者说泛型类）则是 GP 与 OO 的结合。最后，Java 没有函数指针或委托，所以 Java 只能使用多态。

宏思想与语法糖

在 C 程序里我们会常用到宏定义，比如导言中 PI 的例子，也可以写成宏定义：

#define PI 3.14159265

如同导言所讨论的，宏定义也有同样的好处：简化书写、便于修改、便于理解等。然而，正如许多 C/C++书籍所强调的，宏定义不安全，因为它只是执行简单的文本替换。比如#define f(x) x*x，那么 f(1+2)会变成 1+2*1+2 而不是(1+2)*(1+2)。所以我们不提倡在代码中随便使用宏。但是，我们依然需要运用宏的思想，来做到简化书写、便于修改、便于理解的目的。C++的许多语言特性运用到了宏的思想，同时又避免了宏的不安全性：

1. const 常量

这个我们在导言中已经讨论过。

2. typedef

比如我们要使用 C++内置的复数类，完整的名称应该是 std::complex<double>，这样写显然太麻烦，或许可以用宏：

#define comp std::complex<double>

然而这样可能不太安全，更保险的是用 typedef：

typedef std::complex<double> comp;

表明 comp 类型是 std::complex<double>的别名。

3. 内联函数

比如我们在代码中多次用到三次方运算，我们可以定义一个宏：

#define cube(x) ((x)*(x)*(x))

为了避免上面所提到的安全问题，我们加了好几个括号。但是这样并不代表着就完全没问题了，因为宏不会对输入参数进行类型检查，更好的方法是定义内联函数：

```
inline double cube(double x){
    return x*x*x;
}
```

内联函数和函数有什么区别？看起来就是多了个 inline 修饰符而已。但是内联函数在运行时是不存在的，它只是在编译的时候把调用 cube(x)的地方替换成 x*x*x 而已，跟宏一样，只不过它会进行参数的类型检查，而且它保证正确运算次序，不需要像宏一样加括号才能保证。

内联函数比起函数的好处就是性能，因为函数的调用需要进行参数的复制、控制的跳转、返回值的复制等多余操作，而内联函数只是简单的语句替换，所以它的性能会更好。

4. **sizeof**

sizeof 语句常用于内存分配的语句中：

```
int* p=(int*)malloc(sizeof(int)*10); //分配 10 个整数的空间
```

我们明明知道 sizeof(int)是 4，但还是要写成 sizeof(int)，一方面是使表意清晰，另一方面是使代码可移植，如果我们换了一台机器来编译，说不定 sizeof(int)不再是 4 了，而我们用 sizeof 不需要做任何更改。如果我们直接写成 4，那么移植的时候就要把每个 4 都改成别的，麻烦。

5. 枚举

枚举常用于描述选项之类的信息，比如第四节的立方体与球体求体积，也可以这样实现：

```
enum shape{ cube,sphere };
class object{
public:
    shape type; //标志物体属于哪种形状，有两个选项：cube 和 sphere
    double length; //如果是立方体则表示边长，是球体则表示半径
    double volume(){
        switch(type){
        case cube: //如果是立方体
            return length*length*length;
        case sphere: //如果是球体
            return 4*PI*length*length*length/3;
        }
    }
};
```

枚举值本质上就是整数值，上面的 cube 实际上就是 0，而 sphere 就是 1。下面的代码和上面的本质上是一样的：

```
    int type;
    double volume(){
        switch(type){
        case 0: //如果是立方体
            return length*length*length;
        case 1: //如果是球体
            return 4*PI*length*length*length/3;
        }
    }
```

既然直接写 0 和 1 就可以解决问题，那为什么还要使用枚举呢？因为枚举含义清晰，比较好记，上例中的枚举只有两个可能值，假如说有 10 个可能值，那你就得时时刻刻记住哪个数对应哪个意思，稍有不慎就会弄错。还有，枚举可以随意排列，把上面的定义语句改成 enum shape{ sphere,cube }，其它代码都不需要修改；但是如果直接用数字，你想让 0 表示球体，让 1 表示立方体，那就所有代码都要改。

6. 条件编译

条件编译常用于程序版本的控制，比如说，你希望你的程序在测试版本中输出一些中间信息，以检查程序运行过程是否有问题，而在正式版本中你不想要这些信息，你可以一条一条手工地删除这些代码，但是这样做太麻烦，所以可以用条件编译：

```
#define TEST
……
#ifdef TEST
    cout<<信息 1;
#endif
```

……

```
#ifdef TEST
    cout<<信息 2;
#endif
```

当你不想输出这些信息的时候，只要把#define TEST 这一句注释掉就可以了。

综上所述，我们得到了**宏思想原则：运用宏、常量、typedef、内联函数、sizeof、枚举、条件编译之类的语言特性，可以使程序便于书写，便于修改，便于理解。**

而从另外一个角度来看，宏思想体现了一个原则：**将需要经常改动或将来有可能改动的因素所对应的代码量减到最少。（灵活原则 7）**

然后再来讲语法糖。什么是语法糖？就是那些没有为计算机语言增加新功能，而只是用来简化代码书写的语法。比如说，我们可以用 p[i]来表示*(p+i)，这样的写法更为简洁和清晰，但并没有增加什么实际的功能。又如运算符重载，c=a+b 和 c=add(a,b)并没有本质区别，只是看起来更清晰明了而已。

这一次我不用 C++作为例子，而用 C#，因为 C#有很多语法糖。但是在这里我只是展示一下这些语法糖的效果，而不会具体地讲它们如何实现。如果你对它感兴趣可以自己去查 MSDN。

1. 属性

假设你在一个类中储存有月份的信息，你不能让外部代码直接去设它的值，因为你要确保月份的值在 1 到 12 之间，那么在 C++里你可以这样做：

```
int month;
public:
    int getMonth(){
        return month;
    }
    void setMonth(int m){
        assert(m>0 && m<13);
        month=m;
    }
```

那么你在用的时候就必须这样：

```
int m=a.getMonth(); //获取月份
a.setMonth(10); //设置月份
```

这时我们称 Month 为一个属性，属性就是提供一个 get 函数和一个 set 函数去约束对一个私有变量的访问。如果提供 get 函数，则称为只读属性。在 C#里，提供了用于专门定义属性的语法，使得你可以这样使用属性：

```
int m=a.Month;
a.Month=10;
```

看起来 Month 好像是一个成员变量，但它实际上还是一个 get 函数和一个 set 函数。

2. 索引器

假设 Student 类用来储存一个学生的信息，而 Class 类用来储存一个班的信息，class1

是 Class 类的一个实例。那么 Class 类里应该有一个 Student 数组，假设数组名为 students，为了获取这个班的第一个学生，我们应该这样做：

```
Student stu=class1.students[0];
```

C#提供了一索引器的语法，定义了索引器之后你可以这样做：

```
Student stu=class1[0];
```

就好像重载[]运算符一样。还可以这样做：

```
Student stu=class1[“张三”];
```

可以找出名为张三的学生。而且索引器参数还可以不止一个：

```
Student stu=class1[1,3];
```

找出坐在第 1 排第 3 列的学生（假如我们记录了每个学生的座位）。

索引器本质上也是一个 get 函数和 set 函数：

```
Student stu=class1[1,3]; //相当于 class1.get(1,3)
class1[1,3]=stu; //相当于 class1.set(1,3,stu)
```

3. 扩展方法

比如在 C++中，你定义了一个 print 函数来输出一个整数：

```
void print(int x){
    cout<<x<<endl;
}
```

然后你使用这个函数：

```
print(10);
```

C#提供了一种定义函数的语法，使得你可以这样调用它：

```
10.print();
```

仿佛 print 是 int 的成员函数，但本质上并不是。

4. 可变个数参数

考虑一个求和函数，在 C++里你可以这样：

```
int sum(int a[],int n){
    int s=0;
    for(int i=0;i<n;i++)
        s+=a[i];
    return s;
}
```

然后你在代码中使用这个函数：

```
int a[3]={1,2,3};
int s=sum(a,2); //得到 3
s=sum(a,3); //得到 6
```

C#提供了一种语法，能够定义参数个数可变的函数，然后你就可以这样：

```
int s=sum(1,2); //得到 3
s=sum(1,2,3); //得到 6
```

传任意多个参数都可以。这种函数本质上是按照输入的参数生成一个数组，然后把这个数组传给函数。

5. var 关键字

var 关键字可以使你定义变量的时候不需要写类型。比如：

```
var x=10; //x 就是 int 型
var x=”abcd”; //x 就是 string 型
```

有些类型的全称很长，用 var 就可以简化书写。

C++新标准提出 auto 关键字，和 var 有同样的功能，比如：

```
vector<int> vec;
auto iter=vec.begin(); //相当于 vector<int>::iterator iter=vec.begin();
```

除了这里提及的，语法糖还有很多，包括其它语言里的。语法糖可以简化书写，这自然的好事，但是请不要太过依赖语法糖，真正决定编程质量和效率的，是整个程序的架构设计是否合理。也不要凭语法糖多少去评价一种编程语言的好坏。最后总结一下，**应当充分利用语法糖来简化书写，但也不要太过依赖。（简洁原则 5）**

命名、陷阱与异常

命名

命名其实包括两方面的内容，一个是起什么样的名字，一个是在什么地方定义。通常我们认为，一个变量、函数或类的名字应该能够清晰地表达出它的作用，但是有时候要想出一个合适的名字就很费时间，所以比较实际的做法是，**一个变量、函数或类的重要性越大，它的名字就越应该清晰地表达出它的作用。（清晰原则 4）**衡量重要性有两个方面，一方面是本征重要性：类>函数>变量；另一方面是使用次数，一个变量、函数或类在代码中出现的次数越多，它的重要性就越大。所以从某种角度来读，一种语言内置的库函数的重要性是最大的，因为有无数人的无数代码要使用它。所以像 scanf、printf 这样的名字简直罪大恶极，每一个学 C 语言的同学都必须被迫接受这样一些不是单词的单词……呵呵开玩笑啦。

大型项目需要避免的一个重要问题就是同名问题。解决同名的一个方法就是减少全局变量。比如你在程序中用到 100 个坐标，你可以这样定义：

```
int x[100];
int y[100];
```

也可以这样定义：

```
struct point{ int x,y; };
point p[100];
```

把关联密切的一组变量组合到一个结构或类里面去，这样就减少了全局变量。

有些变量根本没必要是全局变量，比如这样：

```
int t;
void swap(int& a,int& b){
    t=a,a=b,b=t;
}
```

完全可以把 t 放到 swap 里定义。

另一个方法是使用命名空间。比如：

```
namespace A{
    int x;
    void f(){}
}
namespace B{
    int x;
    void f(){}
}
```

那么 A::x 和 B::x 就代表不同的变量，A::f()和 B::f()代表不同的函数。在 A 中使用 x 变量就是使用 A::x 变量。通常，我们把一组功能较为接近，关联较为密切的变量和函数放到同一个命名空间里。我本人喜欢把我自己写的代码放在一个以我的名字缩写为名的命名空间里。

还有一种方法是使用静态变量和静态函数。有些全局变量（或全局函数）只和某个类

有关，可以把它定义为这个类的静态变量（或静态函数）。比如第四节的 sphere 类中，用到了 PI 常量，我们发现这个 PI 常量只在 sphere 类用到，在其它代码中不会用到。为了防止其它代码中也定义了这个变量而产生冲突（尤其是在合作编程中，每个人定义自己要的东西，合到一起编译时就冲突了），可以把这个量定义为 sphere 类的静态变量：

```
class sphere{
public:
    static const double PI; //声明
    ……
}
const double sphere::PI=3.14159265; //定义
```

静态变量在类中声明之后，还需要在类外定义。

声明为静态变量后，外界要访问 PI，就必须写成 sphere::PI。如果 PI 在类中是声明为 private 的，那么外界根本就无法访问 PI。

但是，静态变量本质上还是一个全局变量。无论这个类创建了多少个实例，静态变量都只有一个。只不过是它的可访问性与全局变量不同而已。

静态变量不必要是 const 的，上面只是一个例子而已。还可以定义静态的函数：

```
class sphere{
public:
    static double PI(){
        return 3.14159265;
    }
    ……
}
```

同样，静态函数本质上还是全局函数，只是要访问它得用 sphere::PI()而已。

综上可看出，**通过减少全局变量，使用命名空间和静态成员，可以避免命名冲突，同时使代码之间的关系更为清晰。（清晰原则 5）**

可能你注意到了，上面我多次地强调“功能相近”，“关联密切”之类的词语，**把功能相近、关联密切、调用层次相近的函数放到同一个类、同一个命名空间或同一个文件里，可以使整个程序的架构清晰。（清晰原则 6）**相反，如果关联不太密切的函数被乱七八糟地放在一起，那程序就显得混乱。调用层次可以这样理解：A 调用 B，那么 A 的调用层次比 B 高。所以在一个程序里，main 函数的层次最高，汇编指令的层次最低。

陷阱

陷阱就是指那些容易引起错误的语言细节。下面列举一些常见的 C++陷阱：

1. 把“==”误写成“=”。
2. 某些运算符的优先级，如*p++到底是(*p)++还是*(p++)？没把握时还是加括号吧。
3. 运算可能会溢出，比如 int 型 20 亿加 20 亿会得到一个负数。
4. 在 switch 里一个 case 完了忘了加 break，结果继续到下一个 case。

……

总之，**写代码时注意避免语言陷阱。（安全原则 7）**

有些错误其实不是语言的问题，而是程序员自己的疏忽，像这个求和函数：

```
int s=0;
```

```
int sum(int a[],int n){
    for(int i=0;i<n;i++)
        s+=a[i];
    return s;
}
```

每次调用 sum 的时候没有对 s 进行重新初始化，导致每一次求和的结果都积累到下一次。所以，**每次进入函数调用或进入循环体的时候，要记得对相关的变量重新初始化。（安全原则 8）**对于上面的例子，把 s 声明为局部变量就可以解决问题，但有时有些变量必须是全局变量，所以就要重新给它们赋初值。

异常

异常通常是指一些程序无法控制的意外事件，比如用户指定打开一个文件，但是磁盘上却没有这个文件；用户指定要访问某个网络上的资源，但是断网了。一个健壮的程序应该能够对这些意外事件做出恰当的处理。

考虑一个简单的程序：输入一个数，输出它的倒数。那么我们得处理用户输入是 0 的情况，可能有三种处理方式，一种是断言（assert）：

```
double reciprocal(double x){
    assert(x!=0);
    return 1/x;
}
void main(){
    while(true){
        double x;
        cin>>x;
        cout<<reciprocal(x)<<endl;
    }
}
```

一种是异常机制(try-catch)：

```
double reciprocal(double x){
    if(x==0)throw 1; //抛出变量 1
    return 1/x;
}
void main(){
    while(true){
        double x;
        cin>>x;
        try{
            cout<<reciprocal(x);
        }
        catch(int flag){ //捕获变量 1
            if(flag==1)cout<<"0 没有倒数！";
        }
    }
```

```
}
```
一种是返回错误代码：
```
bool reciprocal(double x,double& y){
    if(x==0)return 1; //失败返回 1
    y=1/x;
    return 0; //成功返回 0
}
void main(){
    while(true){
        double x,y;
        cin>>x;
        if(reciprocal(x,y))cout<<"0 没有倒数！ ";
        else cout<<y;
    }
}
```
我们看到，用断言最简单，但是最不负责，因为一输入 0 程序就崩溃了。断言适合用来检查程序潜在的 bug，但是不适合用来处理异常情况。异常情况并不是 bug，而是由于程序无法控制的外部因素引起的，比如用户的非法输入，比如操作系统或硬件的不支持，等等。所以处理异常情况还是要用异常机制或错误代码的方式。

异常机制的好处是，不破坏原有函数的结构，而且可以跨越多层函数调用，而缺点是 try-catch 这样的语句写起来比较烦琐，而且你在一个函数里 throw 就要保证上层代码要有 try-catch，如果上层没有的话，程序还是会崩溃。尤其是合作编程时，如果没有事先约定，有的人在自己的代码里 throw，其它人不知道，没有用 try-catch 处理，那就会出问题。

返回错误代码的好处就是处理起来比较方便，而且你不处理的话程序也还照样运行。但是返回错误代码破坏了原有函数的结构，为了返回一个 bool 值，原来的返回值只能变成一个引用参数 double& y。还有就是无法跨越多层调用，假设 A 调用 B，B 调用 C，C 调用 D，然后 D 中检测到一个异常情况，为了在 A 中处理这个情况，必须把错误代码一层一层传上去，这就太麻烦了。

TopLanguage 上有过不少关于这两者的争论，我觉得两种方式各有优缺点，选择哪一种就取决于个人喜好了。不过，在一个团队里一定要统一，不能有的人用这种，有的人用那种。总之，**要运用异常机制或错误代码的方式，来处理可能遇到的异常情况。（安全原则 9）**

性能优化

第 1 节中已经提到过快速原则 1：尽量减少对数据的复制。在这一节将讨论更多的提高程序执行速度的方法。

快速原则 2：尽量利用先前算出的结果。

比如说计算数列：1，1，2，3，5，8，13……的第 n 位，即 f(0)=f(1)=1，f(n)=f(n-1)+f(n-2)，最直观的代码就是这样：

```
int f(int n){
    if(n==0 || n==1)return 1;
    else return f(n-1)+f(n-2);
}
```

但是这样做效率极低，比如要算 f(10)，就要算 f(9)+f(8)，而算这里的 f(9)又要算 f(8)+f(7)，所以 f(8)的值就重复计算了，算 f(8)又要算 f(7)+f(6)，所以 f(7)重复计算了……最后出现无数的重复计算，尤其是 f(0)。好的做法是不用递归，每个值只算一次：

```
int f(int n){
    int* p=new int[n+1];
    p[0]=p[1]=1;
    for(int i=2;i<=n;i++)
        p[i]=p[i-1]+p[i-2];
    int t=p[n]; //用临时变量保存一下结果，因为 delete p 后 p 就不能用了
    delete p;
    return t;
}
```

这种思想叫做“动态规划”。凡是具有 f(n)=F(f(n-1),f(n-2),…,f(0))这种形式的函数关系，都可以考虑用动态规划法。其中 F 可以是任意函数，f 不一定要一元，可以是多元函数，但是参数必须是整数。

动态规划的思想有很大的实用价值，不过，尽量利用先前算出的结果的原则更具有普遍性。

上面的例子还可以再改进，我们注意到，当算完 f(2)之后，f(0)的值就不会再用到了，那么可不可以把 f(2)的计算结果直接保存到 f(0)所在的位置呢？同样，f(3)的计算结果也可以直接保存在 f(1)里……最后我们只需要 f(0)和 f(1)这两个位置就够了：

```
int f(int n){
    int p[2]={1,1};
    for(int i=2;i<=n;i++)
        p[i%2]=p[0]+p[1];
    return p[n%2];
}
```

这个改进，在时间上不会快多少，但是它节省了空间，所以，**尽量不让用不到的变量继续占有空间（省空间原则）**。

快速原则 3：尽量减少对用不到的数据的计算。

考虑一个函数 int f(int n)，我们且不管这个函数具体是算什么，只知道它每次计算都需

要很长时间。而在我们的程序里，只需要用到 n<100 的情况，而且有多次会用到相同的 n，那么根据上一个原则，我们可以预先把 n<100 的结果全部算好，把结果保存在一个数组里，要用到的时候直接到数组里读取就可以。这样对每个 n 都只算一次，避免重复的计算。这种思想叫“预加载”。

但是，也许我们并不会用到 n<100 的所有结果，可能我们的程序整个运行过程只用到了 n=1,4,23,46,53,79 这几种情况，但是我们却把 n<100 的全算了，所以这是一种浪费。所以我们采取延迟计算的方式：

```
bool flag[100];
int result[100];
```

一开始所有的 flag 都设为 false，表示尚未计算。需要用到某个 f(n)，就用以下代码：

```
int F(int n){
    if(!flag[n]){ //如果没有现成结果，就计算结果
        result[n]=f(n);
        flag[n]=true;
    }
    return result[n];
}
```

这种思想叫“延迟加载”。预加载体现了原则 2，延迟加载体现了原则 3，两种思想结合，就得到上面的最佳方案。

可以把 flag 和 result 定义为函数内部的静态变量：

```
int F(int n){
    static bool flag[100];
    static int result[100];
    ……
}
```

静态变量本质上还是全局变量，但是只有函数 F 才能访问它们，这样就避免了可能和其它变量同名的问题。

快速原则 4：尽量减少循环内的重复计算。

比如我们有一个已知半径的球 ball1，现在要计算这个球在多种不同密度下的质量：

```
for(int i=0;i<n;i++)
    mass[i]=density[i]*ball1.volume();
```

其中 volume()函数就是算 4*PI*r*r*r/3。

但是在这个循环里，ball1.volume()的值被反复地计算，这是浪费，应该这样：

```
double v=ball1.volume();
for(int i=0;i<n;i++)
    mass[i]=density[i]*v;
```

这样就只算一次。其实这个原则和原则 2 没有太大差别，只是这个比较具体一些，比较容易做到。有时候即使你自己没做到，编译器也会帮你优化。但是，像 volume 这样的函数调用，编译器无法肯定每一次调用都会出来同样的结果，所以未必会进行优化。

快速原则 5：尽量减少不必要的函数调用，或使用内联函数。

其实第五节谈内联函数的时候已经说到。内联函数性能比普通函数好就是因为它不需要真正的函数调用的开销。不过，内联函数里面不能有复杂结构，比如循环、条件结构，

也不能有递归，否则，编译器会把这个函数编译成普通函数，而无视 inline 修饰符。

快速原则 6：尽量使用局部变量。

还是考虑求和函数：

```
int s;
int sum(int a[],int n){
    s=0;
    for(int i=0;i<n;i++)
        s+=a[i];
    return s;
}
```

和另一种方式

```
int sum(int a[],int n){
    int s=0;
    ……
}
```

后一种可能性能更高，因为 s 是局部变量，生存期短，编译器可能把它优化到 CPU 的寄存器里储存，而不是储存在内存里，而 CPU 寄存器的访问速度要比内存快得多。但如果 s 是全局变量，整个程序运行期间它都存在，寄存器的数目是有限的，不可能让一个变量长期占着不放，所以 s 只能储存在内存。

不过如果你对局部变量取了地址：

```
int s=0;
int* ps=s;
```

那么 s 就只能储存在内存里了，因为寄存器里的变量是无法取地址的。

快速原则 7：提高数据访问的时空局部性。

时空局部性，就是指在短时间内访问的数据地址基本集中在一个小的区间内。比如遍历一个二维数组 a，可以这样：

```
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
        a[i][j]=0;
```

也可以这样：

```
for(int j=0;j<n;j++)
    for(int i=0;i<n;i++)
        a[i][j]=0;
```

唯一的不同就是 i 和 j 在循环嵌套中的次序不同。那么第一种方法有更好的时空局部性，比如对于 3*3 的矩阵，按第一种的访问次序是 123456789，按第二种的访问次序是 147258369，更具有跳跃性。

为什么内存访问有跳跃性不好？因为计算机的内存是多级缓存机制的。比如一块 1M 大的内存，配上一个 1k 大的缓存，缓存的访问速度比内存快得多，假如我们集中访问某个地址区间中的内存地址，那么计算机就自动把这个地址区间的数据复制到缓存里，让我们直接访问缓存就可以，这样就提高了数据访问的速度。这其实跟前面的预加载是同样的思想。但是，如果我们跳跃式地访问内存，尤其是跳跃的幅度超过了缓存的大小的时候，计算机就根本不知道该把哪一块内存加载来缓存里，刚加载一块地址区间进来，马上又跳到

别的地址区间去访问了。所以提高数据访问的时空局部性可以提高性能。实际的缓存机制比这里说的要复杂得多，但是基本思想是一致的。

最后强调一下，决定程序运行速度最根本的因素是算法与数据结构，而不是编程语言或者是编译器的优化。只是算法与数据结构太博大精深，这里就不作讨论了。

多线程

什么是多线程？多线程有什么用？先来看一个例子。

假设我们要写一个播放在线视频的程序，那么肯定会有两个步骤：下载视频，然后再播放视频。简单用代码表示就是这样：

```
Download();
Play();
```

然而我们必须等到整部视频下载完了才能播放，我们希望下载和播放能够同时进行，下多少播多少。而且，由于网络的原因，下载的速度我们是无法控制的，有时我们下载完50%的时候，才播了 10%，有时却已经播了 40%。无论如何，只要下载完的比已经播完的要多，我们就可以继续播放而无需等待。所以，我们把下载工作和播放工作分别放在两个线程里执行（下面是 C#代码，因为用 C++操作线程比较烦琐）：

```
Thread thread1=new Thread(Download); //创建新线程
thread1.Start(); //启动新线程，开始在新线程中执行 Download()
Play(); //在原来的线程中执行 Play()
```

线程的特点就是，在单个线程里，你可以肯定任何两个事件发生的时间顺序，比如先下后播时，下载完毕这个事件总比开始播放这个事件早发生；而在两个不同线程中的两个事件，一般无法预测其发生的先后顺序，比如边下边播时，我们无法判断下载完 50%这个事件和播放完 30%这个事件谁先谁后。

运用多线程，可以让多个操作同时进行，节省不必要的等待时间。（快速原则 8）我们可以边浏览网页，边听音乐，边下载东西，杀毒软件还会在后台帮我们扫描磁盘，因为这些操作都处在不同的线程之中。

单核的 CPU 只能顺序地执行一条一条的指令，为了实现多线程，操作系统必须把 CPU 运行的时间划分成一段一段的时间片，比如以 1ms（毫秒）为单位。在某个 1ms 内，CPU 执行线程 A 的指令，在下一个 1ms 内，CPU 执行线程 B 的指令，而在我们用户的眼中，就好像 A 和 B 同时在进行。

多核的 CPU 就不一样，比如双核，线程 A 在这个核，线程 B 在那个核，两个核执行指令互不相干，是真正的并行运行。比如我们要复制一个大数组，在单线程的情况下可能需要一秒时间，我们可以把数组分成两部分，让两个线程同时来复制，两个线程在不同的核中执行，那么就只需要半秒的时间。所以**在多核的情况下使用多线程，可以提高程序的性能。（快速原则 9）**

当然，对于上面的播放视频的程序，还需要考虑下载速度比播放速度还慢的情况。假如我们用 nDown 表示已经下载的数据量，用 nPlay 表示已经播放的数据量，那么只有 nDown 比 nPlay 大时，才可以继续播放。Play 函数可以这样写：

```
void Play(){
    while(nPlay<nAll){ //nAll 就是整部视频的数据量
        while(nPlay<nDown); //等待 nPlay 小于 nDown
        PlayPart(nPlay,nDown); //播放 nPlay 到 nDown 的片段
    }
}
```

在这里，我们假设 Download 函数会自动修改 nDown，PlayPart 函数播完片段之后会自动修改 nPlay。

在不同线程中的两个事件发生的先后顺序一般是所谓的，但是有些特定事件的先后顺序还是要保证的，比如上面例子需要保证下载到 nDown 之后才能播放到 nDown，这种保证称为“线程同步”。在上面的例子中，我们用了一种最原始的方法 while(nPlay<nDown);来实现线程同步，不断地进行条件测试直到 Download 线程修改了 nDown 的值使它大于 nPlay 为止。这种方法称为“轮询”。但是这样做浪费很多无谓的时间在条件测试上。一种更好的方案是使用 Sleep：

```
while(nPlay<nDown)Thread.Sleep(100);
```

只要测试条件不满足，就让当前线程睡眠 100ms，把 CPU 让给别的线程去用。这样就不会浪费大量时间在轮询上了。

不过这样做还有一个小问题，比如某一次测试条件不满足，于是 Sleep(100)。但是在睡了 50ms 之后，Download 线程已经下载了一大堆新的数据，可以继续播放了，但 Play 线程还是得再睡 50ms 才能醒过来继续播放，睡过头了。我们希望当 Download 线程积累了足够的未播放数据（比如说 10000 字节）的时候，能够主动通知 Play 线程，叫它别睡懒觉了起来播东西了。我们可以用同步事件来做到这一点：

```
AutoResetEvent event1=new AutoResetEvent(false); //创建一个全局的同步事件
void Download(){
    while(nDown<nAll){ //nAll 就是整部视频的数据量
        DownloadPart(); //新下载一段数据，并修改 nDown
        if(nDown>nPlay+10000)
            event1.Set(); //通知 Play 线程起来工作了
    }
}
void Play(){
    while(nPlay<nAll){
        event1.WaitOne(); //睡觉，等待 Download 线程来叫醒自己
        PlayPart(nPlay,nDown); //播放 nPlay 到 nDown 的片段
    }
}
```

现在 Play 线程就舒心了，既不用忙个不停，也不用担心睡过头。总之，**避免用轮询来实现线程同步，而是用睡眠或同步事件。（快速原则 10）**Java 中用 wait 和 notify 来实现同步事件。

再来看另一个多线程的例子，假设我们要处理一些银行帐户，可以定义下面的类：

```
class Account{
    double money; //当前帐户的余额
    public void add(double x){ //存入 x 元
        money+=x;
    }
}
```

对于一些大公司的帐户，可能会有很多客户同时往这个帐户汇款，为了避免这些人相互等待，可以为每一个汇款人分配一个线程，每个线程负责接待一位客户，并且最后调用 add 函数存入用户指定的数额。

但是这样做却可能会导致错误的结果，比如一个线程存入 10 元，另一个线程存入 20 元，结果 money 值却只增加 10 元，为什么会这样？

我们来看 money+=x 这一句代码，看起来是一个操作，但是实际运行时是三条指令：

从内存读入 money 的值，放在寄存器中；

从内存读入 x 的值，加到 money 所在的寄存器上。

把寄存器中的 money 值写入内存。

我们不妨将寄存器记为 eax，把指令的赋值操作记为<-，那么上面指令可以简写成：

eax<-money

eax<-eax+x

money<-eax

假设 money 原来为 0，线程 1 和线程 2 同时调用 add 函数，x 分别为 10 和 20，由于线程间指令执行顺序是不确定的，所以可能出现以下顺序（用 123456 表示）：

指令	线程 1	线程 2
eax<-money	1	3
eax<-eax+x	2	4
money<-eax	6	5

1、2：线程 1 执行前两条指令，此时 eax 的值为 10；

3、4、5：切换到线程 2，线程 2 执行三条指令，此时 money 的值为 20，注意不同线程有各自的寄存器变量，线程 2 修改了自己的 eax 但不会修改线程 1 的 eax；

6：切换到线程 1，执行最后一条指令，线程 1 的 eax 值仍为 10，所以 money 的值被设为 10。

由于每一个线程都共享 money 变量，不同线程对 money 变量的访问次序是无法预测的，所以出现了这样的现象。解决这个问题的方法是对共享变量加同步锁：

```
class Account{
    double money;
    object moneyLock=new object(); //代表对 money 的锁
    public void add(double x){
        lock(moneyLock){ //锁定 moneyLock 对象
            money+=x;
        }
    }
}
```

被 lock 语句的括号括住的代码区域就像是一个带锁的房间，moneyLock 就是它的锁。假设线程 1 先进入 add 函数，执行 lock(moneyLock)，进入房间并把房门锁住。然后线程 2 进入 add 函数，也想执行 lock(moneyLock)，但房门已经被锁住，它进不去，只能等待线程 1 执行完 money+=x 语句，开锁出了房间，线程 2 才能进去。这样就保证了，任意时候只有一个线程占有这个房间，执行里面的代码。

原则上 lock 语句可以锁定任何对象，但是最好是锁定私有对象，比如上例中的 moneyLock，以防止外部代码也对它进行锁定。

所以，**修改线程的共享变量时，最好对它加锁。（安全原则 10）**C#中的 lock 语句、C++中的临界区、Java 中的 synchronized 关键字都可以用来加锁。另外，Mutex（互斥）也可以用来加锁，而且不仅可用于线程间加锁，还可以用于进程间加锁，但是系统开销比较大。

如果所有的线程都只对共享变量进行读取，则不需要加锁。

虽然加锁可以保证对共享变量的正确访问，但是不恰当的加锁可能会导致一些问题，

比如线程 1 执行以下代码：

```
lock(a){
    lock(b){
        ……
    }
}
```

线程 2 执行以下代码

```
lock(b){
    lock(a){
        ……
    }
}
```

可能会发生什么事？假如线程 1 执行完 lock(a)，锁住了 a 对象；这时刚好切换到线程 2，执行 lock(b)，锁住了 b 对象，想再执行 lock(a)，但是 a 已经被线程 1 锁住了，于是线程 2 等待；于是切换到线程 1，想执行 lock(b)，但是 b 对象已经被线程 2 锁住了，所以线程 1 也等待。两个线程就这样永远相互等待下去，这种情况叫做“死锁”。

一些方法可以避免死琐，比如说，有多个锁的时候，每个线程都按同样的次序加锁和解锁；不要把不会修改到共享变量的代码也写到加锁的区域中；减小加锁的“粒度”，比如说共享一个数组，如果对整个数组加锁，那么线程 1 访问数组时线程 2 就得等待，如果对数组的每一个元素加锁，那么一个线程 1 访问某个元素时线程 2 还可以访问另一个元素。

总之，**在使用同步事件或同步锁的时候，要防止死锁。（安全原则 11）**

代码编辑

下面列举一些代码编辑器常用的功能，如果你正在使用的代码编辑器有此功能，请充分利用，这样可以提高写代码的效率。没有的话就算了。

添加：

自动补全：键入单词的前几个字母，自动插入整个单词。

代码模板：插入一些常用的代码结构或程序框架。

方便查看：

自动格式化：不用你自己敲 Tab 缩进代码，自动缩进。

折叠代码块：一个文件太多行时，把一些类和函数的代码折叠起来。

跳转：

类视图：列出当前工程的所有类和成员，可随意跳转到任一个类或成员所在的代码。

书签：在经常改动的代码放一个书签，要找就比较容易。

转到定义或声明：选中一个变量或函数或类，跳转到它定义或声明的地方。

键盘跳转：最典型如 VIM，通过丰富的命令来实现各种快捷的跳转。

重构：

重构，指的是不改变代码的功能，但改变其具体实现。前面几节讨论的很多例子其实都涉及重构。比如复数类，把 c.add(a,b)改成 c=add(a,b)，功能没变，但实现方式变了，看起来更为清晰，这就是一种重构。又如数据统计，把 m=count(a,30,60)改成 m=count(a,30,Func1(60))，也是一种重构，功能没变，但是使用起来可以更灵活了，这也是一种重构。

重构是整个程序开发过程要不断做的事情。因为很多情况下你往往不能在一开始就清楚什么样的设计是最佳的，而且将会使用这个程序的用户的需求也往往是变化的。所以如果只是一味地向程序添加新功能，那么整个程序的架构可能会越来越混乱，越来越难以维护。就好比我们电脑磁盘中的文件，如果我们总是随意存放新文件，那么时间一长就会很乱，有些东西你就忘记放在哪个文件夹里了，所以时不时要整理整理。写程序也是，**时不时要进行重构，虽然没有改变它的功能，但是可以使它变得更清晰、更灵活，方便添加新功能。**

大多数的重构还是要手工完成的，但有些简单的可以用代码编辑器来做：

文本替换：经常用于变量、类、函数的重命名，但是要非常小心，因为这只是简单地做文本替换，把变量 f 替换成 F 会把 printf 也替换成 printF。

智能重命名：有些编辑器可以做到，比如上例，通过推理识别出哪些 f 才代表真正的变量 f，就不会出现上面的问题了。

重新排列函数参数的顺序：重排后，所有调用这个函数的代码都会自动重新排列。

提升局部变量为函数参数：比如第二节的最开始的部分，把 60 提升为函数参数 min，有些编辑器能自动把调用 count 函数的代码修改过来，比如原来是 count(a,30)，就改成 count(a,30,60)。

……

尽管如此，代码编辑器所能够做的重构还是很有限。我认为未来的代码编辑器应该往更为智能的重构方面发展。

我认为代码编辑器还应该支持重构的导入和导出，但是似乎现在还没有哪个代码编辑器做到这一点。比如 A 同学写了一个类库，把它发布到网上，B 同学在自己的程序里使用到了 A 的类库。后来，A 同学对类库进行了升级，改进了架构，但是某些类的接口发生了变化，比如有些函数的名字发生了变化，参数个数发生了变化。B 同学如果想让原来的程序使用上 A 同学的新类库，那就要对程序里所有使用到这些类的代码进行修改，那显然很麻烦。所以编程的接口一旦制定，是很难修改的。Win32 的 API 函数，十几年了还是那个样子，因为一旦修改，很多程序就无法在 Windows 上运行了。但是如果重构可以导入和导出，A 同学就可以把它对类库接口所做的更改导出为一个文件，随新类库一起发布，B 同学让代码编辑器导入这个文件，自动对 B 的程序中的代码进行更改，然后重新编译就行了。

这一节的题目叫“代码编辑”，不过好像没有太多可说的。不如说一些别的：C++模板元编程。

什么是元编程？来看一个例子，比如我们要算 n 的阶乘，可以用下面的递归函数：

```
void factorial(int n){
    if(n==0)return 1;
    else return n*factorial(n-1);
}
```

但是你还可以用模板来实现：

```
template<int N>
struct factorial{ //默认情况下用这个版本
    static const int value=N*factorial<N-1>::value;
};
template<>
struct factorial<0>{ //N=0 时用这个特化版本
    static const int value=1;
};
```

上面的代码的意思就是：

如果 N=0，则使用 factorial 的特化版本，factorial<N>::value=1;

否则使用 factorial 的普通版本，factorial<N>::value =N*factorial<N-1>::value;

所以，factorial<3>::value 就表示 3 的阶乘。

用模板来计算阶乘，和用函数来计算阶乘，有什么不同呢？不同就在于，前者是在程序编译的时候进行计算的，而后者是在程序运行的时候进行计算的，经过编译之后，factorial<3>::value 就会变成 6。所以你不能这样：

```
int x=3;
int y=factorial<x>::value;
```

因为 x 的值只有在运行的时候才知道，所以上面的代码编译不通过。

这种在编译时进行计算的程序就称为“元程序”。通过把一些运行时的计算提前到编译时来完成，可以提高程序运行时的性能（当然，编译速度就会变慢）。甚至有人证明，C++的模板元编程是图灵完备的，也就是说，所有普通程序能够完成的计算，元程序也能完成。

除了能够对数值进行计算，元编程还可以对类型进行计算，这是普通编程做不到的。比如判断两个类型是否相等，你总不能用 if(int==double)…吧！但是下面的代码可以对类型进行比较：

```
template<class T1,class T2>
struct is_same{ //默认情况下用这个版本
    static const bool value=false;
};
template<class T1>
struct is_same<T1,T1>{ //T1和T2相等时用这个版本
    static const bool value=true;
};
```

上面的代码的意思就是：

如果 T1 和 T2 相等，则使用 is_same 的特化版本，is_same <T1,T2>::value=true;

否则使用 is_same 的普通版本，is_same <T1,T2>::value=false;

于是is_same<int,doube>::value就是false，is_same<int,int>::value的值就是true。

可是类型计算有什么意义？难道我不知道int和double相不相等？看一个应用的例子。假设我们写了一个模板类class1，这个类具体做什么不重要，我们只知道它的某个函数func里，需要对浮点型的数值进行特别的处理，而对其它的类型不需要特别处理，那么可以这样：

```
template<class T>
class class1{
    ……
    void func(){
        if(is_same<T,double>::value || is_same<T,float>::value){
            ……//如果T是double或float型，则做一些特别处理
        }
        ……//默认情况下做的操作
    }
};
```

C++模板元编程，是 C++最高深的部分，这里举的都是最简单的例子，涉及的都只是皮毛。C++模板强大的类型运算能力，弥补了 C++无法在运行时获取类型信息的缺陷。对于能够在运行时获取类型信息的语言，像 C#，提供了对元编程的原生支持，如 typeof(int)==typeof(double)，很直观地比较两个类型。当然，由于是在运行时实现的，它的性能肯定不如 C++。

测试

测试就是指检验程序是否能运行得出正确的结果。“黑盒测试”是指不考虑程序内部如何实现，只检查程序能否对各种不同的输入得出正确的输出结果；“白盒测试”是指深入程序代码内部，测试每一个操作的执行是否正确。测试的方法有很多，而且有专门用来测试的软件，这里都不讲，这里只讲一些测试的原则。

1. 自动化原则：尽量把测试所要做的人工操作减到最少。

比如你想测试程序对于 10 种不同的输入的结果是否正确，你可以重复启动 10 次你的程序，每次输入不同的结果，看结果是否正确，但是这样是不是很麻烦？你可以把 10 次的输入和结果都先放在数组里，然后在主函数里用一个循环，每一次循环就调用你程序的主计算的代码，自动输入，并检查输出是否与预期的一致。这样只需运行一次，结果对不对就自动都出来了。

当然，这样做破坏了原来的代码。如果不想破坏原来的代码，可以另外写一个测试程序，或者直接用测试软件来测试。

2. 写到哪就测到哪。

如果等到整个程序写完再测试，那出了问题上哪找原因？问题可能出在代码的任一个角落，找起来就很麻烦。所以，每写完一个小单元，就要测试一下，确保没有问题了，再写下一个小单元。一个单元就是一个函数或一个类。一般情况下，经过充分测试的小单元存在 bug 的概率要比没有经过测试的小得多，所以当某次测试结果出错的时候，那么问题很有可能就在最新写进去那个单元里。

写写测测，是不是太麻烦了？但是，如果不测试，那以后出了问题，找起来那可要麻烦得多。相比之下，写写测测的效率还是高一些。

3. 合理安排测试的顺序。

一般情况下，先测试核心功能，再测试其它功能；先测试没有把握的代码，再测试有把握的代码。然后，既然是写到哪测到哪，那么测试的顺序也就是写的顺序。

总之，如果 A 单元的执行正确性依赖于 B 单元的正确性，那么 B 单元应该比 A 单元先测试。如果彼此不依赖，那次序就无所谓了。

4. 测试不同的平台和环境。

如果你的程序想在不同的操作系统下都能运行，那么就要确保它在每个操作系统下都正确运行。对于网页上的程序，则要测试它在不同浏览器下的正确性。对于汇编程序，则要测试不同的机器。

如果你的程序和操作系统的互动比较多，那么不同平台上的测试工作最好早点进行，除非是很有把握。如果一直都在某个平台上测试，等到最后快完工了再测试其它的平台，可能会突然间冒出很多的错误。

5. 测试集尽量多样化。

测试集就是测试时输入数据的集合。输入数据可能有很多种可以取的值，无法一一测试，所以会找出一些代表，组成测试集。比如测试一个一元函数 f(n)，n 可能取 1 到 100 中的整数，要取 10 个代表，那么找哪些代表好呢？1，2，3，……10，这样都是些小数值，

缺乏代表性；10，20，30，……100，这样都是 10 的倍数，也不太好；还是多样化一些比较好，比如：1，8，31，42，……，85，100。

6. 要测试非法输入和边界值附近的输入。

还是上面的例子 f(n)，n 可取 1 到 100 的整数。除了测试输入 1 到 100 的值能否得到正确结果，还要测试输入非法数值会出现什么事，比如说输入-10，输入 200，输入 4.5，输入 abc，看看是什么结果。可能程序会崩溃，可能程序会算出一个莫名其妙的结果，可能程序会死循环，可能程序会输出一条友好的信息提示输入有误请重新输入。不管怎么样，结果要符合你的预期才对。

还要测试边界值附近的输入，比如上面 1 和 100 是边界值，那么一定要特别测试一下 0、1、100、101 这四个输入会得到的结果，因为有可能程序里检查边界的代码有误，比如 <=100 被写成<100，那么输入 100 会被错误地认为是非法输入。

调试

调试可以深入程序内部，观察运行时各个变量的值。但是，并不是一出现 bug 就要调试。调试最适合用来探究一些自己不太熟悉的语言特性或者是技术。比如你对 C++某些语句的作用不太熟悉，对某个库函数的作用不太熟悉，调试一下，就可以看得清清楚楚了。如果程序只是逻辑出错误，最好的方法是测试，通过逐个单元的测试，找出问题的所在。为什么测试的效率更高？因为测试可以是自动化的，你可以编写测试代码，一次性地完成很多测试，但调试只能一步一步地来。调试的好处是可以直接看变量的值，而测试的话，必须写额外的代码把变量的值输出到控制台或者日志文件里。下面说一些调试的技巧。

断点

最简单的一种，设置一个断点，程序执行到那一句就自动中断进入调试状态。

单步执行

有三种，一种是每次执行一行；一种是每次执行一行，但遇到函数调用就会跳到被调用的函数里；一种是直接执行当前函数里剩下的指令，返回上一级函数。在 Visual Studio 中，上面三种方法对应的快捷键分别为 F10、F11、Shift+F11。

监视

调试器可能会自动列出一些相关变量的值，但是你可能还关心其它变量的值，可以添加对这些变量的监视。还可以监视一个表达式的值，比如 a+b。但是，这个表达式最好不要修改变量的值，比如监视 a++都会导致监视时修改了 a 的值，影响了程序的运行结果。

条件中断

假如你有这样的循环：

```
for(int i=0;i<100;i++){
    for(int j=0;j<100;j++){
        ……
    }
}
```

你怀疑当 i=10 且 j=10 的时候执行有问题，那如何调试？用断点的话，从 i=0 的初始状态，需要中断 10 次才能到 i=10，然后从 j=0 也需要再中断 10 次，才能到 j=10 的状态。所以想进入 i=10 且 j=10 的状态，需要中断 20 次，这太麻烦了。可以使用条件中断：

```
for(int i=0;i<100;i++){
    for(int j=0;j<100;j++){
        if(i==10 && j==10){
            ; //空语句
        }
        ……
    }
}
```

在空语句的那一行设置断点就可以了。

上面的 if 结构太占地方，还可以用 assert：

assert(i!=10 || j!=10);

断言 i 不为 10 或 j 不为 10，那么当 i=10 且 j=10 的时候，断言就不成立，程序就会中断，进入调试状态。

有时候用 throw 也可以中断：

if(i==10 && j==10)throw;

但是最好不要这样做，调试器不一定会在 throw 的地方中断。

控制变量法

其实这已经不算是调试的内容了，但是也是一种找出 bug 原因的手段，所以还是在这里说。

控制变量法常用于科学研究中，比如说，研究牛顿第二定律 a=F/m，a 与 F 和 m 都有关，那么可以先固定 m，研究 a 与 F 的关系；然后固定 F，研究 m 与 a 的关系。

对于一个程序来说，一个 bug 可能跟多处代码有关。假如你怀疑这个 bug 与某些语句有关，可以把这些语句注释掉，或者是改一改，看看 bug 是否还存在，如果不存在，说明确实跟这些语句有关。（当然，要保证程序少了这些语句之后还可以顺利运行。）如果 bug 还存在，就说明它跟这些语句无关。

有些时候我们缺乏调试工具，比如在网页上运行的程序，在特殊设备上运行的程序，那么控制变量法是一种很有用的代替手段。

二分法

二分法是控制变量法的进一步扩展。

在数学上，二分法用于求一个连续函数的根。比如一个函数 f(x)，如果 $f(x_1)>0$ 且 $f(x_2)<0$，那么在区间 x_1 和 x_2 之间，必定存在一个 x，使 f(x)=0。然后我们再考察区间的中点 $x_3=(x_1+x_2)/2$，如果 $f(x_3)>0$，则函数的根就在区间 x_3 和 x_2 之间，如果 $f(x_3)<0$，那么函数的根就在区间 x_1 和 x_3 之间。如此不断地把区间一分为二，最后锁定函数的根。

对于一个程序来说，如果当前情况是有 bug 的，那就好比是 $f(x_1)>0$；如果你把 main 函数里所有的操作都注释掉，那么程序什么都不做，就不可能有 bug，那就好比是 $f(x_2)<0$；于是在这两种状态之中，肯定存在一些临界的语句，当这些语句改动的时候，就会使程序在有 bug 和无 bug 状态间切换，这些语句就是 bug 的原因所在。运用二分法的思想可以锁定这些临界语句。一开始先对程序做一些大刀阔斧的改动，比如说，程序的主循环会循环 10 次，就改成 1 次；程序有 10 个功能，就关掉 5 个功能。看看哪些改动，可以让程序由有 bug 状态切换到无 bug 状态。找到这样改动后，就把这个改动再细分成几个小改动，比如关掉 5 个功能，就细分为关掉一两个功能，再看看哪些小改动可以让程序由有 bug 状态切换到无 bug 状态。如此一步一步缩小包围圈，就后锁定一个无法再分的小改动，这个改动就是 bug 的原因所在。

同步法

有些 bug 是由于多线程而产生的。因为在不同线程里的操作我们无法预测其发生的顺序，可能当它们按某种次序进行时，bug 不会出现，当它们按另一种次序进行时，bug 就出现了。比如多线程那一节说到的那个银行帐户，如果没有加同步锁，就会出现这种 bug。对这种 bug 的调试是很困难的，有时你运行程序发现了 bug，而在进行调试的时候，由于执行顺序不同了，bug 又不出现了。

为了解决这个问题，我想了一个办法，就是利用同步事件，强行把多线程的程序按照预定好的顺序去执行。比如说有两个线程，一开始就让线程 1 运行，线程 2 睡觉，线程 1

运行到某个特定的点后，就换线程 2 运行，线程 1 睡觉。任何时候，都只有一个线程可以运行。我们可以在多次运行的过程中使用不同的执行顺序，如果按某种执行顺序运行之后 bug 浮现了，那么就把这种顺序记录下来。然后按照这种顺序进入调试，找出 bug。

用这种方法一定要谨慎，如果你的程序里本来就有线程同步的代码，再加上这些强制的同步，可以会导致死锁。

总结

简洁原则：写出来的代码要尽量简洁，避免重复。

1 把经常用到的代码段写成一个函数，可以简化代码。
2 把函数定义为与之密切相关的类的成员函数，可以简化函数实现的代码。
3 对于内部实现不同但对外接口相同的类成员函数，可以建立共同的基类虚函数（或 Java 和 C#中的接口），从而可以在外部代码中统一处理，简化外部代码。
4 对于内部实现完全一致的类成员变量和函数，可以放在基类中统一实现，简化内部代码。
5 应当充分利用语法糖来简化书写，但也不要太过依赖。

安全原则：写出来的代码要不易出错，易于查错。

1 要保证每个指针或引用不指向实际并不存在的变量。
2 在堆中创建的变量，要保证能回收。
3 如果外界代码对类的某个成员变量或函数的随意访问可能导致不好的结果，则应该把它设为私有成员。
4 复制或传递函数参数时，要清楚复制或传递的是指针（引用）还是数值。
5 要确保每个数组的下标访问不会越界，每个函数的输入参数都合法。
6 同时使用两个指针（或引用）时，要考虑两个指针（或引用）引用同一个变量的情况。
7 写代码时注意避免语言陷阱。
8 每次进入函数调用或进入循环体的时候，要记得对相关的变量重新初始化。
9 要运用异常机制或错误代码的方式，来处理可能遇到的异常情况。
10 修改线程的共享变量时，最好对它加锁。
11 在使用同步事件或同步锁的时候，要防止死锁。

快速原则：写出来的代码要快速运行，尽量省时。

1 尽量减少对变量的复制。
2 尽量利用先前算出的结果。
3 尽量减少对用不到的数据的计算。
4 尽量减少循环内的重复计算。
5 尽量减少不必要的函数调用，或使用内联函数。
6 尽量使用局部变量。
7 提高数据访问的时空局部性。
8 运用多线程，可以让多个操作同时进行，节省不必要的等待时间。
9 在多核的情况下使用多线程，可以提高程序的性能。
10 避免用轮询来实现线程同步，而是用睡眠或同步事件。

灵活原则：写出来的代码要易于更改，易于扩展。

1 对于有可能改变的数值，不要写死，可以作为函数参数传进来。
2 对于有可能改变的类型，不要写死，可以写成模板，把类型作为模板参数传进来。包括

类模板和函数模板。
3 尽量利用已有的类和函数来构造新的功能，而不改变类和函数的内部代码。
4 当一个函数里有一段有可能改变的代码时，不要写死，可以把这段代码委托给一个传进来的函数指针去执行。
5 如果需要对函数指针进行更加灵活的定制，可以使用函数对象或者 lambda 表达式。
6 对于只在模块内部使用到的代码，访问限制松一些，参数检查松一些，类和变量的命名随意一些。会被外部使用到的代码则相反。
7 将需要经常改动或将来有可能改动的因素所对应的代码量减到最少。

清晰原则：写出来的代码要意图清晰，易于阅读。

1 一个函数的函数名、参数个数和类型、返回值类型应该能够自然地体现出这个函数所要实现的操作。
2 严格控制变量作用域，只有真正需要用到某个变量的代码段才可以访问到这个变量。
3 尽量减少对类和函数的用法的无理规定。
4 一个变量、函数或类的重要性越大，它的名字就越应该清晰地表达出它的作用。
5 通过减少全局变量，使用命名空间和静态成员，可以避免命名冲突，同时使代码之间的关系更为清晰。
6 把功能相近、关联密切、调用层次相近的函数放到同一个类、同一个命名空间或同一个文件里，可以使整个程序的架构清晰。

测试

1 自动化原则：尽量把测试所要做的人工操作减到最少。
2 写到哪就测到哪。
3 合理安排测试的顺序。
4 测试不同的平台和环境。
5 测试集尽量多样化。
6 要测试非法输入和边界值附近的输入。

其它

宏思想原则：运用宏、常量、typedef、内联函数、sizeof、枚举、条件编译之类的语言特性，可以使程序便于书写，便于修改，便于理解。
省空间原则：尽量不让用不到的变量继续占有空间。
重构原则：时不时要进行重构，虽然没有改变它的功能，但是可以使它变得更清晰、更灵活，方便添加新功能。
调试技巧：断点、单步执行、监视、条件中断、控制变量法、二分法、同步法。

后记

首先，我为什么要写《高效编程十八式》。其实就是想把自己编程过程中学到的、领悟到的东西整理整理，发出来大家讨论讨论。我不是计算机专业的学生，我喜欢编程，但是在我所在的学院里像我这样的人并不多。而且我这个人交际面不广，也没有认识多少其它院系的人。所以大学里的很多时候我都是一个人在学，看编程的书，做一些自己喜欢的小项目。我不知道我现在到底属于什么水平，因为没有人可以跟我一起讨论或参照。我越来越觉得不能再这样下去，所以就想到把我学到的东西写下来，跟大家一起分享讨论。初学者看了，可能会学到一些有用的东西；牛人看了，可以帮我指出其中的不足。

我认为编程的原则比具体的语言特性重要。语言可以有很多种，原则却是通用的。不过为了用实例来阐述这些原则，在文章里我还是介绍到了 C++的不少语言特性。我尽量地避免太多地去讨论 C++的语言特性，以免喧宾夺主，至于效果如何，我就不知道了。从所有文章来看，C++语言的各个方面我基本上都涉及到了，但是都只是讲了最简单的情况，没有深入讨论。所以要学 C++的话，看我这个是没有用的，要看专门介绍 C++的书才有用。

本来我打算所有的代码都用 C++写，但是由于 C++语言本身的缺陷，有些东西用 C++写起来不太直观，所以有的地方会用 C#写。之所以选择 C#，一方面是因为我对它比较熟，另一方面是它的语法和 C++很接近，所以能保持总体上大致统一的风格。我尽量地避免涉及到它与 C++的不同之处，使这些 C#代码看起来和 C++的没有太大的差别，但是本质上，C#和 C++还是很不同的。

我没有讲到设计模式，因为我认为设计原则比设计模式重要。采取什么模式取决于具体问题，但是原则适用于所有问题。面对一个具体问题，只要朝着原则所认可的方向去设计，最佳的模式就自然而然地会出来。

虽然我举了这么多例子来阐述编程原则，但是我认为只有在实际的编程过程中才能真正掌握这些原则。我写的这些东西，可能会有一些启发作用吧。

我几乎没有谈到算法与数据结构，只有在性能优化那一节中提到了一点。但是这并不是说算法不重要，而是它太重要了，以至于需要另外讨论。我觉得算法可以分为两种：确定性的和不确定的性的。确定性的算法有一个明确的目标，必须把这个目标准确求出来。而不确定性的只要求出近似的解出来，甚至连要求解的目标都是不明确的。

有一些思想常常可以用来求解确定性的问题，比如贪心法、动态规划、分治、延迟加载、快速查询表、筛法等等。但是这些思想仍然依赖于具体问题，所以单靠这些思想还是会有很多问题无法解决。性能优化那一节中提到的几个原则，比如减少对数据的复制、减少重复的计算、减少对用不到的数据的计算，对于算法的设计有普遍的适用性，但是这些原则太过笼统，面对一个实际问题，你很难由这些原则直接得到一个有效的算法。所以，算法的研究依然是很必要的，也是很有挑战性的。这不仅是计算机领域的问题，也是数学领域的问题。

不确定性的问题常见于人工智能领域。像贝叶斯网络、人工神经网络、遗传算法、微粒群算法等，都是用来解决不确定性问题的算法。虽然这些算法各不相同，但我觉得，还是存在一些共同的思想的。比如离散的逻辑连续化、操作随机化、采用非线性的函数关系、群体相互作用、结果反馈等等。但是同样，这些思想还不足以解决所有问题。

有的人喜欢用严格的数学方法去分析一个智能算法的行为，我不太赞同这种做法。如果一个算法能够用数学严格分析出它的行为，那只能说明这个方法不够智能。假定你能够

获取我现在所有的脑电波信息，你能够用数学的方法预测出我 1 分钟之后会做什么事？我觉得要想得到一个足够聪明的算法，从生物学或者心理学的角度去寻找答案可能会更好。

我想写一些关于算法的总结，但是我现在还写不出来。因为算法实在太博大精深了，我所学的还太少。我会把今后的学习重点放在算法上，包括确定性的和不确定性的算法。希望以后的某一天，我能够写出关于算法的总结。

最后，感谢 gullibility、Geo、sweating 等同学指正了文章中的不少错漏之处，感谢我 mm 帮我修改了不少错别字和病句。也欢迎大家提出更多的意见。

王伟冰

2010 年 3 月 21 号 于北京大学