1. 直接赋值、浅拷贝和深拷贝的区别

对于基本数据类型，使用=做赋值操作的时候，实际上是拷贝它的值，但是对于对象而言，实际赋值的只是这个对象的引用，将原对象的引用传递过去，他们实际上还是指向的同一个对象。浅拷贝和深拷贝就是在这个基础上做的区分，如果在拷贝这个对象的时候，只对基本数据类型进行了拷贝，而对引用数据类型只是进行了引用的传递，而没有真实的创建一个新的对象，则认为是浅拷贝。反之，在对引用数据类型进行拷贝的时候，创建了一个新的对象，并且复制其内的成员变量，则认为是深拷贝。

直接赋值：其实就是对象的引用（别名）

浅拷贝：拷贝父对象，不会拷贝对象的内部的子对象。浅拷贝仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么复制出来的对象也会相应改变。

深拷贝：copy模块的deepcopy方法，完全拷贝了父对象及其子对象。在计算机中开辟了一块新的内存地址用于存放复制的对象。

1. java中的引用

先说C++中的引用是同一块内存的不同名字，而Java的引用是指向一个对象，引用本身也占用了内存

1. java的值传递和引用传递

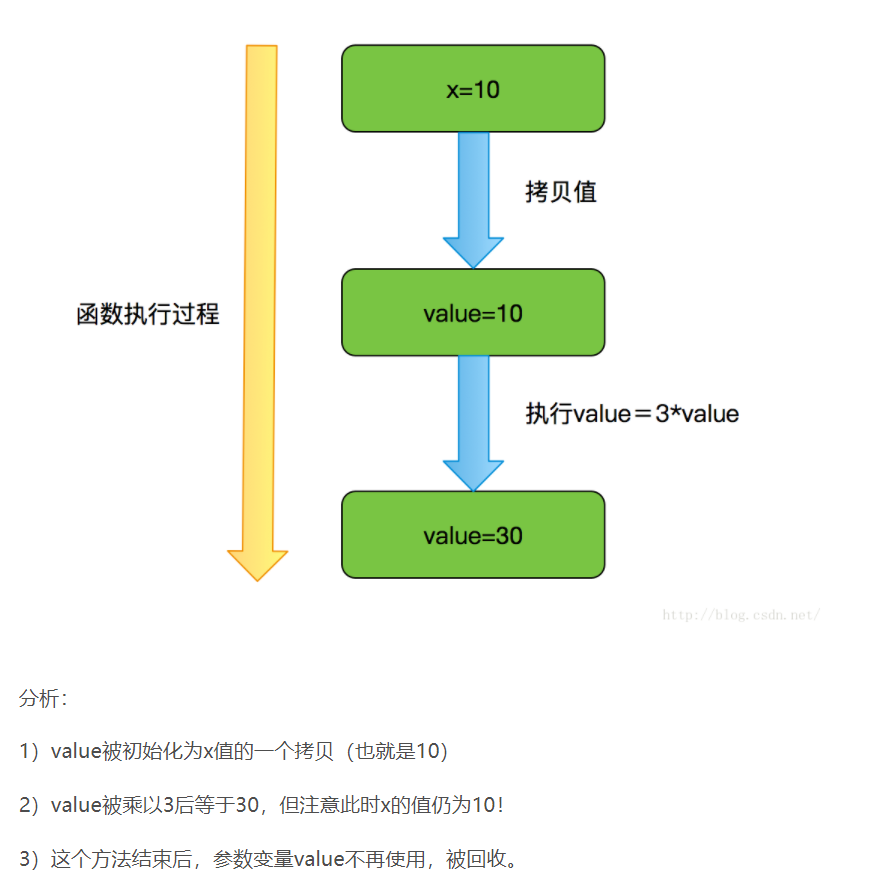
参数传递给方法的两种方式：  
（1）按值调用

（2）按引用调用

·按值调用表示方法接收的是调用者提供的值，而按引用调用则表示方法接收的是调用者提供的变量地址（在C语言中就是指针，但Java中没有指针的概念）。

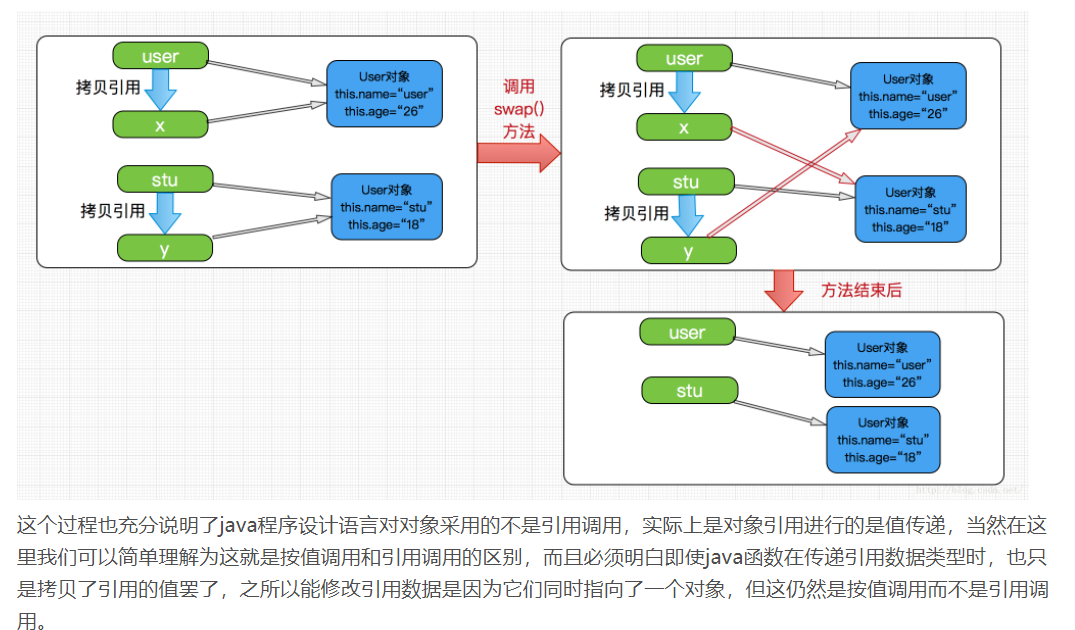
所以在Java中采用了按值调用，也就是说方法得到的是所有参数值的一个拷贝，方法并不能修改传递给它的任何参数变量的内容。

当传递方法参数类型为基本数据类型时，一个方法是不可能修改一个基本数据类型的参数。



当传递方法参数类型为引用数据类型时，一个方法将修改一个引用数据类型的参数所指向对象的值

user和stu的值并没有发生变化，也就是方法并没有改变存储在变量user和stu中的对象引用。swap方法的参数x和y被初始化为两个对象引用的拷贝，这个方法交换的是这两个拷贝的值而已，最终，所做的事都是白费力气罢了。在方法结束后x，y将被丢弃，而原来的变量user和stu仍然引用这个方法调用之前所引用的对象。



1. 多态，重写和重载

多态是指允许不同类的对象对同一消息做出响应，统一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。多态存在的三个必要条件：继承、重写、父类引用指向子类对象（子类到父类的类型转换。在多态中需要将子类的引用赋给父类对象，只有这样该引用才既能可以调用父类的方法，又能调用子类的方法）

Override重写一个方法，以实现不同的功能。一般是用于子类在继承父类时，重写父类中的方法。

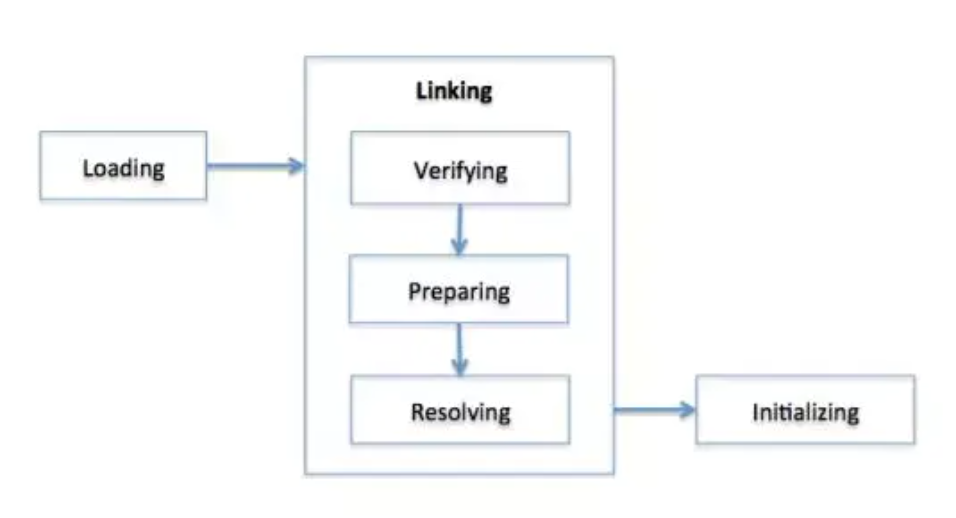
重写的规则：

1. 重写方法的参数列表必须完全与被重写的方法的相同，否则不能称其为重写而是重载
2. 重写方法的访问修饰符一定要大于被重写方法的访问修饰符（public>protected>default>private）
3. 重写的方法的返回值必须和被重写的方法的返回值一致
4. 重写的方法所抛出的异常必须和被重写方法所抛出的异常一致，或者是其子类
5. 被重写的方法不能为private,否则在其子类中只是新定义了一个方法，并没有对其进行重写
6. 静态方法不能被重写，可以被重载

Overload是重载，一般是用于在一个类内实现若干重载的方法，这些方法的名称相同而参数形式不同。

重载的规则：

1. 在使用重载时只能通过相同的方法名、不同的参数形式实现。不同的参数类型可以是不同的参数类型、不同的参数个数，不同的参数顺序
2. 不能通过访问权限、返回类型、抛出的异常进行重载
3. 方法的异常类型和数目不会对重载造成影响
4. JAVA类加载器



5.1装载(loading)

负责找到二进制字节码并加载至JVM中，JVM通过类名、类所在的包名、ClassLoader完成类的加载。因此，标识一个被加载了的类：类名 + 包名 + ClassLoader实例ID。完成以下三件事：

* 通过类的全限定名获取该类的二进制字节流。
* 将二进制字节流所代表的静态结构转化为方法区的运行时数据结构。
* 在内存中创建一个代表该类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。、

5.2链接(linking)—验证、准备、解析

负责对二进制字节码的格式进行校验、初始化装载类中的静态变量以及解析类中调用的接口。完成校验后，JVM初始化类中的静态变量，并将其赋值为默认值。最后对比类中的所有属性、方法进行验证，以确保要调用的属性、方法存在，以及具备访问权限

* + 1. 验证：确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。
    2. 准备

准备阶段是正式为类变量（或称“静态成员变量”）分配内存并设置初始值的阶段。这些变量（不包括实例变量）所使用的内存都在方法区中进行分配。

* + 1. 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

5.3初始化(initializing)

负责执行类中的静态初始化代码、构造器代码以及静态属性的初始化，以下四种情况初始化会被触发

* 在遇到 new、putstatic、getstatic、invokestatic 字节码指令时，如果类尚未初始化，则需要先触发其初始化。
* 对类进行反射调用时，如果类还没有初始化，则需要先触发其初始化。
* 初始化一个类时，如果其父类还没有初始化，则需要先初始化父类。
* 虚拟机启动时，用于需要指定一个包含 main() 方法的主类，虚拟机会先初始化这个主类。

来源：https://github.com/doocs/jvm/blob/master/docs/01-jvm-memory-structure.md

# 6.JVM内存结构

6.1程序计数器（PC计数器）

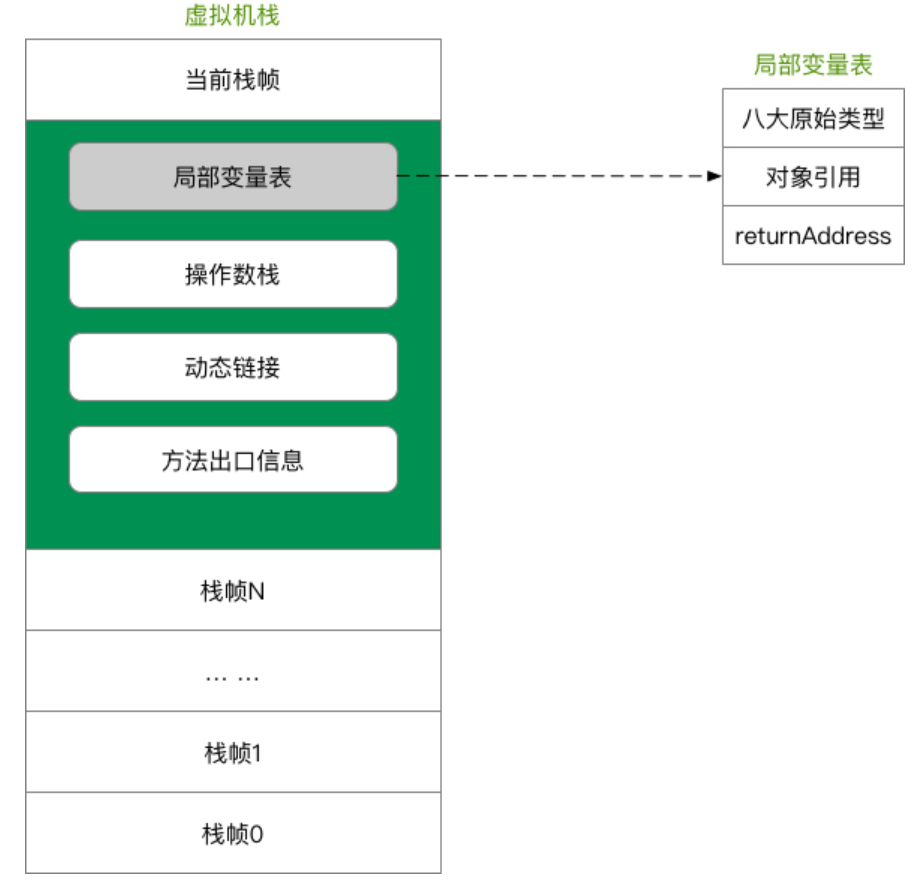
程序计数器是一块较小的内存空间，是当前线程正在执行的那条字节码指令的地址。若当前线程正在执行的是一个本地方法，那么此时程序计数器为undefined.

程序计数器的作用：字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制。在多线程情况下，程序计数器记录的是当前线程执行的位置，从而实现线程切换回来时，就知道上次线程执行到哪了。

程序计数器的特点：是一块较小的内存空间。线程私有，每条线程都有自己的程序计数器。它的生命周期随着线程的创建而创建，随着线程的结束而销毁。

6.2Java虚拟机栈

Java虚拟机栈是描述Java方法运行过程的内存模型。栈会为每一个即将运行的Java方法创建一块叫“栈帧”的区域，用于存放该方法运行过程中的一些信息：



局部变量表

操作数栈

动态链接

方法出口信息

6.2.1压栈出栈过程：

当方法运行过程中需要创建局部变量时，就将局部变量的值存入栈帧中的局部变量表中。Java虚拟机栈的栈顶的栈帧时正在执行的活动栈，也就是当前正在执行的方法，PC寄存器也会指向这个地址。只有这个活动的栈帧的本地变量可以被操作数栈使用，当在这个栈帧中调用另一个方法，与之对应的栈帧又会被创建，新创建的栈帧压入栈顶，变为当前的活动栈帧。

方法结束后，当前栈帧被移出，栈帧的返回值编程心得活动栈帧中操作数栈的一个操作数。如果没有返回值，那么新的活动栈帧中操作数栈的操作数没有变化。

6.2.2Java虚拟机栈的特点

局部变量表随着栈帧的创建而创建，它的大小在编译时确定，创建时只需分配事先规定的大小即可。在方法运行过程中，局部变量表的大小不会发生改变。Java虚拟机栈会出现两种异常：

1. Stackoverflow.若Java虚拟机栈的大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度时，抛出异常
2. Outofmemoryerror,若允许动态扩展，那么当线程请求栈时内存用完了。无法再动态扩展时，抛出异常

6.3本地方法栈

本地方法栈是为JVM运行Native方法准备的空间，由于很多native方法都是用C语言实现的，所以又称为C栈。它与Java虚拟机栈实现的功能类似，只不过本地方法栈是描述本地方法运行过程的内存模型。

本地方法被执行时，在本地方法栈也会创建一块栈帧，用于存放还犯法的局部变量表、操作数栈、动态链接、方法出口信息等。方法执行结束后，相应的栈帧也会出栈，并释放内存空间。

6.4堆

堆是用来存放对象的内存空间，几乎所有的对象都存储在堆中。

特点：

1. 线程共享，整个Java虚拟机只有一个堆，所有的线程都访问同一个堆。而程序计数器、Java虚拟机栈、本地方法都是一个线程对应一个。
2. 在虚拟机启动时创建
3. 是垃圾回收的主要场所

Java堆所使用的内存不需要保证是连续的，而由于堆是被所有线程共享的，所以对它的访问需要注意同步问题，方法和对应的属性都需要保证一致性。

6..5方法区

Java虚拟机规范中定义方法区是堆的一个逻辑部分。方法区存放以下信息：

已经被虚拟机加载的类信息

常量：放在常量池中。

静态变量

即时编译器编译后的代码

6.5.1方法区的特点

线程共享。方法区是堆的一个逻辑部分，因此和堆一样，都是线程共享的。整个虚拟机中只有一个方法区。

永久代。方法区中的信息一般需要长期存在，而且它又是堆的逻辑分区，因此用堆的划分方法，把方法区称为永久代。

内存回收效率低。

# 7.背包问题

7.1 0-1背包

F(j) = max{ f[ j - c[ i ] ]+w[ i ], f[ j ] }

范围:1<=i<=n, c[i] <= j <=m(注意此时取得值，初值为m , j >= c[i] , j--

初始状态：F[0] = 0

目标：F[m]

7.2 完全背包问题



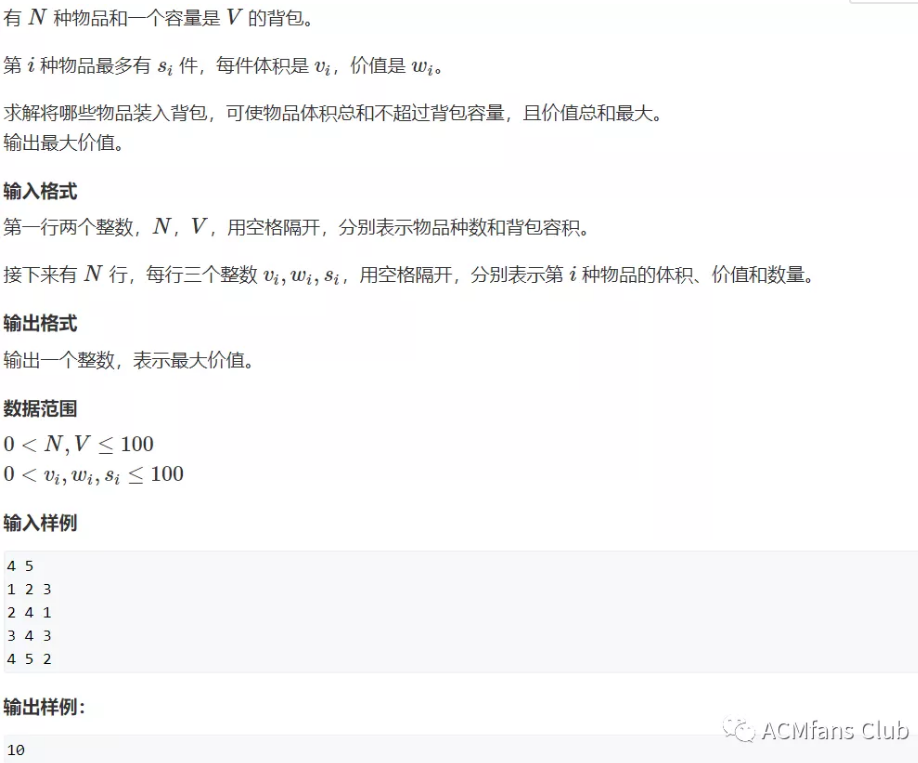
F(j) = max{ f[j], f[j-w[i]] + v[i]}

范围:1<=i<=N , w[i]<=j<=V

初始状态：F[0] = 0

目标：f[V]

7.3多重背包问题



转换成0-1背包与完全背包

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

{

For(j = V ; j >= w[i] ; j--)

{

For(k=0; k<s[i] and k\*w[i] <=j;k++)

{

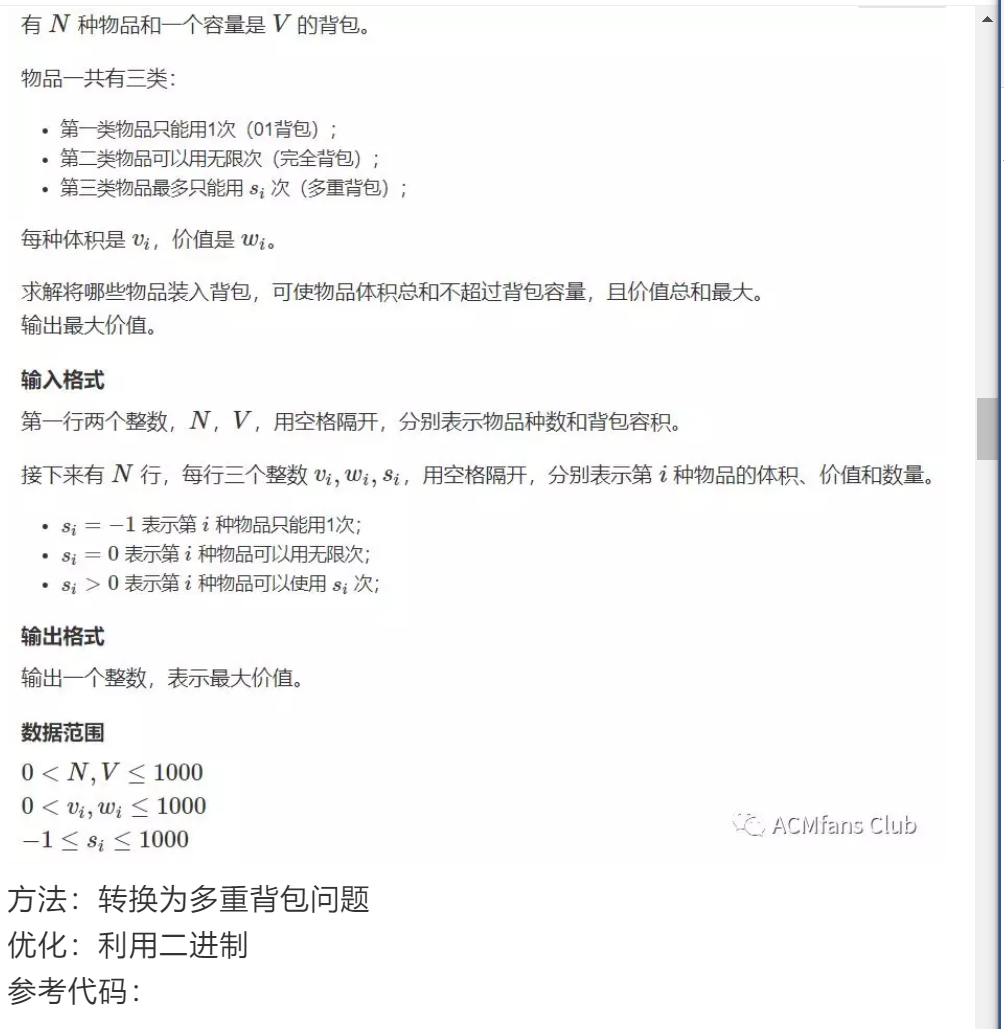
F[j] = max(f[j],f[j-k\*w[i]] +k\*v[i]);

}

}

}

7.4混合背包



转换成多重背包问题

7.5 二维费用的背包问题



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

{

scanf("%d%d%d",&v,&m,&w);

for(int k = V;k >= v;k--)

{

for(int u = W;u >= m;u--)

{

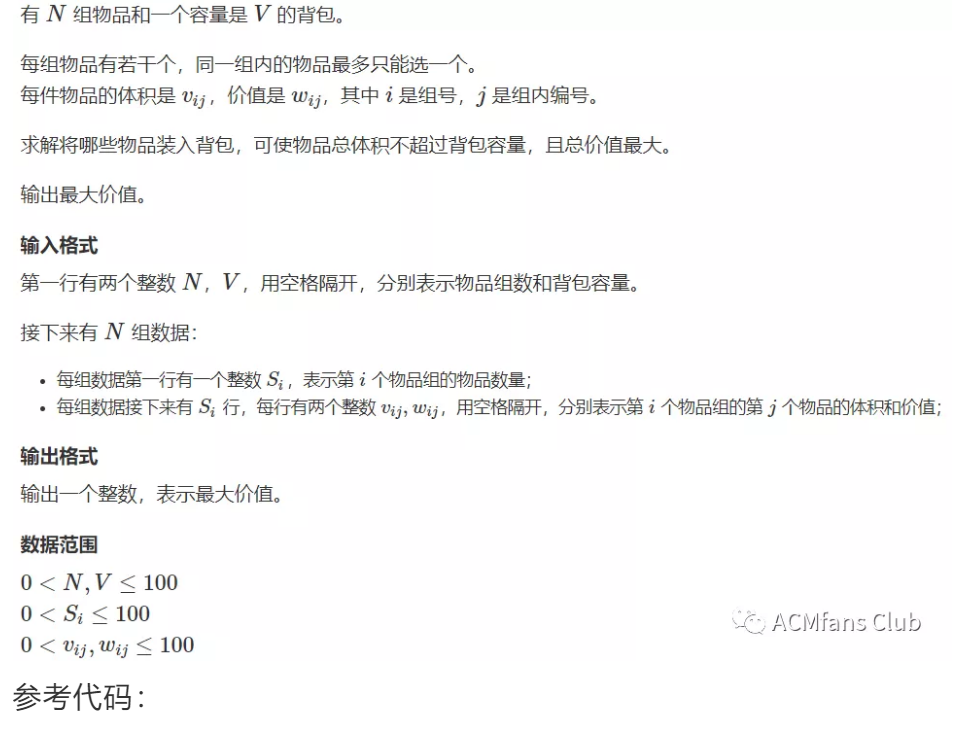
f[k][u] = max(f[k][u],f[ k-v ][ u-m ] + w);

}

}

}

7.6分组背包问题





8.

‘’’8.乐观锁、悲观锁

8.1乐观锁

乐观锁就是在操作时很乐观，认为操作不会产生并发问题（不会有其他线程对数据进行修改），因此不会上锁。但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或者CAS算法实现。

8.1.1版本号机制：

实现套路：

取出记录时，获取当前version

更新时，带上这个version

执行更新时，set version = newVersion where version = oldVersion

如果version不对，就更新失败

8.1.2CAS算法

当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

CAS操作中包含三个操作数：

需要读写的内存位置v

进行比较的预期原值 A

拟写入的新值B

如果内存位置v的值与预期原值A相匹配，那么处理器会自动将该位置更新为新值B.否则处理器不做任何操作。

8.2悲观锁

总是假设最坏的情况，每次取数据时都认为其他进程会修改，所以都会加悲观锁。一旦加锁，不同线程同时执行时，只能有一个线程执行，其他的线程在入口处等待，直到锁被释放。

悲观锁在MySQL、Java中有广泛的使用：

MySQL的读锁、写锁等

Java的synchronized关键字’’’

# 8.堆排序

排序

void HeapSort(int a[],int len)

{

build(a,len);

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

{

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

adjust(a,1,i-1);

}

}

建堆

void build(int a[],int len)

{

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

{

adjust(a,i,len);

}

}

调整节点

void adjust(int a[],int k,int len)

{

a[0] = a[k];

for(int i = k\*2;i<=len;i=i\*2)

{

if(i<len && a[i]<a[i+1])

{

i+=1;

}

if(a[i]<a[0])

{

break;

}

else{

a[k] = a[i];

k = i;

}

}

a[k] = a[0];

}