961 复习大纲

第一部分 数据结构与算法

考试题型：问答、分析、编程

总分：60 分

一、栈（Stack）、队列（Queue）和向量（Vector）(感谢 idea)

1、单链表,双向链表,环形链表,带哨兵节点的链表;

答：**链表相关概念及性质**

链表是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每个节点里存到下一个节点的指针。由于不须按顺序存储，链表在插入的时候可以达到 O(1)的复杂度，比顺序表 O(logn)快得多，但是查找一个节点或者访问特定编号的节点则需要 O(n)的时间，而顺序表的时间复杂度是 O(1)。

链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

链表由一连串节点组成，每个节点包含任意的实例数据(data fields)和一或两个用来指向明上一个/或下一个节点的位置的链接（"links"）。链表是一种自我指示数据类型，因为它包含指向另一个相同类型的数据的指针。链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。

单向链表或者单链表

1. 单向链表，它包含两个域，一个信息域和一个指针域。
2. 单向链表，链接指向表中的下一个节点，而最后一个节点则指向一个空值 NULL。
3. 单向链表只可向一个方向遍历。
4. 查找一个节点的时候需要从第一个节点开始每次访问下一个节点，一直访问到需要的位

置。

5. 也可以提前把一个节点的位置另外保存起来，然后直接访问。

双向链表,也叫双链表

1. 双向链表中不仅有指向后一个节点的指针，还有指向前一个节点的指针。
2. 双向链表第一个节点的"前连接"指向 NULL，最后一个节点的"后连接"指向 NULL。
3. 双向链表可以从任何一个节点访问前一个节点,也可以访问后一个节点,以至整个链表。
4. 双向链表一般是在需要大批量的另外储存数据在链表中的位置的时候用。
5. 双向链表由于另外储存了指向链表内容的指针，并且可能会修改相邻的节点，有的时候第一个节点可能会被删双向链表除或者在之前添加一个新的节点。这时候就要修改指向首个节点的指针。
6. 双向链表有一种方便的可以消除这种特殊情况的方法是在最后一个节点之后、第一个节点之前储存一个永远不会被删除或者移动的虚拟节点，形成一个循环链表。这个虚拟节点之后的节点就是真正的第一个节点。这种情况通常可以用这个虚拟节点直接表示这个链表。

循环链表（环形链表）

1. 循环链表,首节点和末节点被连接在一起。这种方式在单向和双向链表中皆可实现。
2. 要转换一个循环链表,你开始于任意一个节点然后沿着列表的任一方向直到返回开始的节点。
3. 循环链表可以被视为"无头无尾"。循环链表中第一个节点之前就是最后一个节点，反之亦然。循环链表的无边界使得在这样的链表上设计算法会比普通链表更加容易。
4. 对于新加入的节点应该是在第一个节点之前还是最后一个节点之后可以根据实际要求

灵活处理，区别不大。

5. 另外有一种模拟的循环链表，就是在访问到最后一个节点之后的时候，手工跳转到第一个节点。访问到第一个节点之前的时候也一样。这样也可以实现循环链表的功能，在直接用循环链表比较麻烦或者可能会出现问题的时候可以用。

带哨兵节点链表

1. 不带哨兵节点的双向链表在做查找删除节点等操作的时候，免不了要判断边界条件，比如 node==NULL 等。每次判断边界条件，虽然不会从根本上增加时间复杂度，但是对其常数项还是有影响的
2. 带哨兵节点构成的双向循环链表，则可以省去这些问题。我们使用一个“哑的”NIL 节点来代替之前的 head 头指针，NIL 节点的 key 值没有实际的意义，主要关注它的 next 和 pre，初始的时候，链表只有一个 NIL 节点，NIL.next 指向自己，NIL.pre 也指向自己。当添加了若干个节点之后，NIL.next 指向头节点，而 NIL.pre 则指向尾节点；而同样的，这时头节点的 pre 不再是 NULL 而是指向 NIL，尾节点的 next 也不再是 NULL，也是指向 NIL。
3. 不带哨兵节点的双向链表这样的好处在于，我们判断边界条件的时候，不需要再判断是否为空，尤其在删除节点的时候。
4. 带哨兵节点构成的双向循环链表，要额外分配空间来存储 NIL 节点，如果对于多个比较短的链表而言，这样可能会代码比较大的冗余空间。

2、栈的基本概念和性质,栈 ADT 及其顺序,链接实现;栈的应用;栈与递归;

答：**栈的概念和性质**

栈是 LIFO（Last In First Out），先存进去的数据只能最后被取出来，进出顺序逆序，即先进后出，后进先出。

ADT

ADT Stack{

数据对象：D={ai|ai 属于 ElemSet，i=1,2,...,n，n>=0}

数据关系：R1={<ai-1,ai>|ai-1, ai 属于 D，i=2，...，n}

约定 an 端为栈顶，a1 端为栈底。

基本操作：

InitStack(&S)　操作结果：构造一个空栈 S。

DestroyStack(&S)　初始条件：栈 S 已存在　操作结果：栈 S 被销毁

ClearStack(&S)　初始条件：栈 S 已存在　操作结果：将 S 清为空栈

StackEmpty(S)　初始条件：栈 S 已存在　操作结果：若栈 S 为空栈，则返回 TRUE，否则 FALSE

StackLength(S)　初始条件：栈 S 已存在　操作结果：返回 S 的元素个数，即栈的长度

GetTop(S, &e)　初始条件：栈 S 已存在且非空　操作结果：用 e 返回 S 的栈顶元素

Push(&S, e)　初始条件：栈 S 已存在　操作结果：插入元素 e 为新的栈顶元素

Pop(&S, &e)　初始条件：栈 S 已存在且非空　操作结果：删除 S 的栈顶元素，并用 e 返回其值

StackTraverse(S, visit())　初始条件：栈 S 已存在并非空

操作结果：从栈底到栈顶依次对 S 的每个数据元素调用函数 visit()，一旦 visit()失败，则返回操作失败。

}ADT Stack　(出自《数据结构(C 语言版)》严蔚敏、吴伟民著)

栈的应用举例　有数制转换、括号匹配的检验等。

3、队列的基本概念和性质,队列 ADT 及其顺序,链接实现;队列的应用;

答：队列的基本概念和性质：

1. 队列是一种线性集合，其元素一端加入，从另一端删除，因此我们说队列元素是按先进先出（FIFO）方式处理。

2. 队列的处理过程：通常队列会画成水平，其中一端作为队列的前端（front）也称队首（head）,另一端作为队列的末端（rear）也称队尾（tail）。元素都是从队列末端进入，从队列前端退出。

3. 在队列中，其处理过程可在队列的两端进行，而在栈中，其处理过程只在栈的一端进行，但两者也有相似之处，与栈类似，队列中也没有操作能让用户“抵达”队列中部，同样

也没有操作允许用户重组或删除多个元素。

ADT

ADT Queue{

数据对象：D={ai|ai 属于 ElemSet，i=1,2,...,n，n>=0}

数据关系：R1={<ai-1,ai>|ai-1, ai 属于 D，i=2，...，n}

约定 a1 端为队列头，an 端为队列尾。

基本操作：

InitQueue(&Q)　　操作结果：构造一个空队列 Q。

DestroyQueue(&Q) 初始条件：队列 Q 已存在　　操作结果：队列 Q 被销毁

ClearQueue(&Q)　　初始条件：队列 Q 已存在　　操作结果：队列 Q 清为空栈

QueueEmpty(Q)　　初始条件：队列 Q 已存在　　操作结果:若队列 Q 为空栈,则返回 TRUE，否则 FALSE

QueueLength(Q)　　初始条件：队列 Q 已存在　　操作结果：返回 Q 的元素个数，即队列的长度

GetHead(Q, &e)　　初始条件：队列 Q 非空　　操作结果：用 e 返回 Q 的对头元素

EnQueue(&Q, e)　　初始条件：队列 Q 已存在　　操作结果：插入元素 e 为 Q 的新队尾元素

DeQueue(&Q, &e)　　初始条件：队列 Q 已存在且非空　　操作结果：删除 Q 的队头元素，并用 e 返回其值

QueueTraverse(S, visit())　　初始条件：队列 Q 已存在并非空

操作结果：从队头到队尾，依次对 Q 的每个数据元素调用函数 visit()，一旦 visit()失败，则返回操作失败。

}ADT Queue　(出自《数据结构(C 语言版)》严蔚敏、吴伟民著)

队列链表与数组（顺序）的实现

1.　链表实现队列：

队列与栈的区别在于，我们必须要操作链表的两端。因此，除了一个指向链表首元素的引用外，还需要跟踪另一个指向链表末元素的引用。再增加一个整形变量 count 来跟踪队列中的元素个数。综合考虑，我们使用末端入列，前端出列。

2.　数组实现队列：

固定数组的实现在栈中是很高效的，是因为所有的操作（增删等）都是在集合的一端进行的，因而也是在数组的一端进行的，但在队列的实现中则不是这样，因为我们是在两端对队列进行操作的，因此固定数组的实现效率不高。

队列的应用实例：模拟售票口

4、向量基本概念和性质;向量 ADT 及其数组、链接实现;

答：向量基本概念和性质：

1. 对数组结构进行抽象与扩展之后，就可以得到向量结构，因此向量也称作数组列表（Array list）。
2. 向量提供一些访问方法，使得我们可以通过下标直接访问序列中的元素，也可以将指定下标处的元素删除，或将新元素插入至指定下标。为了与通常数组结构的下标（Index）概念区分开来，我们通常将序列的下标称为秩（Rank）。
3. 假定集合 S 由 n 个元素组成，它们按照线性次序存放，于是我们就可以直接访问其中的第一个元素、第二个元素、第三个元素……。也就是说，通过[0, n-1]之间的每一个整数，都可以直接访问到唯一的元素 e，而这个整数就等于 S 中位于 e 之前的元素个数——在此，我们称之为该元素的秩（Rank）。
4. 不难看出，若元素 e 的秩为 r，则只要 e 的直接前驱（或直接后继）存在，其秩就是 r-1（或 r+1）。
5. 支持通过秩直接访问其中元素的序列，称作向量（Vector）或数组列表（Array list）。
6. 实际上，秩这一直观概念的功能非常强大——它可以直接指定插入或删除元素的位置。

ADT

表2.1 向量ADT支持的操作接口

操作接口	功能	适用对象
size()	报告向量当前的规模（元素总数）	向量
get(r)	获取秩为r的元素	向量
put(r, e)	用e替换秩为r元素的数值	向量
insert(r, e)	e作为秩为r元素插入，原后继元素依次后移	向量
remove(r)	删除秩为r的元素，返回该元素中原存放的对象	向量
disordered()	判断所有元素是否已按非降序排列	向量
sort()	调整各元素的位置，使之按非降序排列	向量
find(e)	查找等于e且秩最大的元素	向量
search(e)	查找目标元素e，返回不大于e且秩最大的元素	有序向量
deduplicate()	剔除重复元素	向量
uniquify()	剔除重复元素	有序向量
traverse()	遍历向量并统一处理所有元素，处理方法由函数对象指定	向量

(出自《数据结构(C++语言版)第三版》邓俊辉著)

二、树 （感谢黑曼巴）

1、 树的基本概念和术语;树的前序,中序,后序,层次序遍历;

概念：树（Tree）是 n（n>=0）个结点的有限集

术语：

节点的度：一个节点含有的子树的个数称为该节点的度；

叶节点或终端节点：度为 0 的节点称为叶节点；

非终端节点或分支节点：度不为 0 的节点；

双亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点；

孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点；

兄弟节点：具有相同父节点的节点互称为兄弟节点；

树的度：一棵树中，最大的节点的度称为树的度；

节点的层次：从根开始定义起，根为第 1 层，根的子节点为第 2 层，以此类推；

树的高度或深度：树中节点的最大层次；

堂兄弟节点：双亲在同一层的节点互为堂兄弟；

节点的祖先：从根到该节点所经分支上的所有节点；

子孙：以某节点为根的子树中任一节点都称为该节点的子孙。

森林：由 m（m>=0）棵互不相交的树的集合称为森林；

二叉树的遍历（traversing binary tree）按某种搜索路径巡访树中每个结点，使每个结点均被访问一次仅且一次

（1）先序遍历二叉树　　(根>左>右)

（2）中序遍历二叉树　　(左>根>右)

（3）后序遍历二叉树　　(左>右>根)

（4）层次遍历二叉树

2、二叉树及其性质;普通树与二叉树的转换;

定义：

是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。

性质：

性质 1：二叉树第 i 层上的结点数目最多为 2^{i-1} (i≥1)。

性质 2：深度为 k 的二叉树至多有 2^{k}-1 个结点(k≥1)。

性质 3：包含 n 个结点的二叉树的高度至少为 log2 (n+1)。

性质 4：在任意一棵二叉树中，若终端结点的个数为 n0，度为 2 的结点数为 n2，则 n0=n2+1

满二叉树：

1）一棵深度为 k 且有 2^k-1 个结点的二叉树

2）可以对满二叉树的结点进行连续编号，约定编号从根开始，自上而下，自左而右

完全二叉树:

深度为 k 的，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 到 n 的结点一一对应时，称为完全二叉树

完全二叉树特点：

1）叶子结点只可能出现在层次最大的两层上

2）对任一结点，若其右分支下子孙的最大层次为 l，其左下分支的子孙的最大层次必为 l 或者 l + 1

3）深度为 k 的完全二叉树第 k 层最少 1 个结点，最多 2k−1−1 个结点；整棵树最少 2k−1 个结点，最多 2k−2 个结点

4）具有 n 个结点的完全二叉树的深度为[log2n]+1

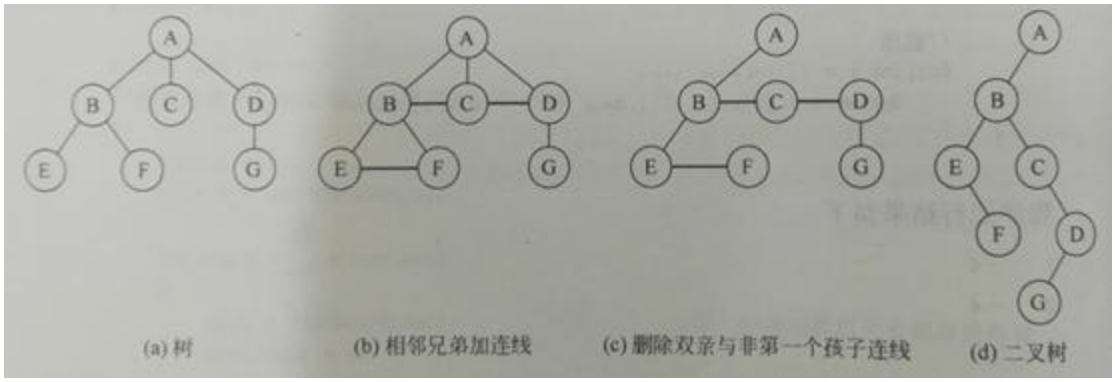
二叉树的转换

树转换为二叉树过程：

（1）树中所有相同双亲结点的兄弟结点之间加一条连线；

（2）对树中不是双亲结点的第一个孩子的结点，只保留新添加的该结点与左兄弟结点之间的连线，删去该结点与双亲结点之间的连线；

（3）整理所有保留的连线，根据连线摆放成二叉树的结构，转换完成

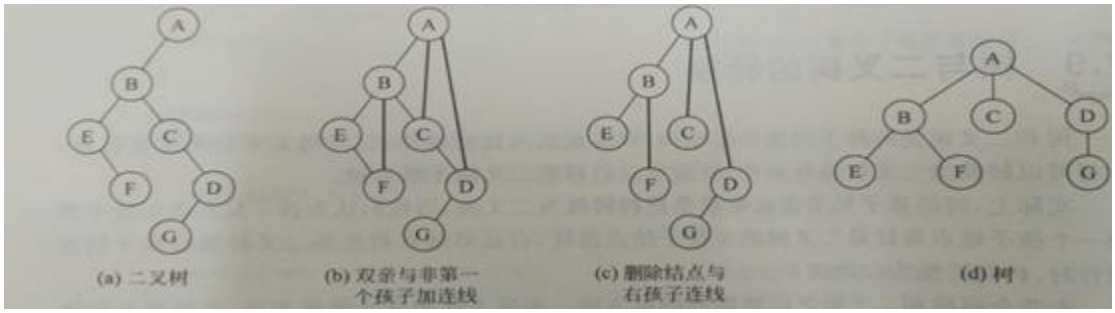


二叉树转换为树过程：

（1）若某结点是其双亲结点的左孩子，则把该结点的右孩子，右孩子的右孩子都与该结点的双亲结点用线连起来；

（2）删除原二叉树中所有双亲结点与右孩子结点的连线；

（3），根据连线摆放成树的结构，转换完成。



3、树的存储结构,标准形式;完全树(complete tree)的数组形式存储;

1） **双亲表示法定义：**假设以一组连续空间存储数的结点，同时在每个结点中，附设一个指示器指示其双亲结点到链表中的位置。

data(数据域)	parent(指针域)
存储结点的数据信息	存储该结点的双亲所在数组中的下标

2） **孩子表示法：**把每个结点的孩子结点排列起来，以单链表作为存储结构，则 n 个结点有 n 个孩子链表，如果是叶子结点则此单链表为空。然后 n 个头指针又组成一个线性表，采用顺序存储结构，存放进一个一维数组中。

- 孩子链表的孩子结点

child(数据域)	next(指针域)
存储某个结点在表头数组中的下标	存储指向某结点的下一个孩子结点的指针

- 表头数组的表头结点

data(数据域)	firstchild(头指针域)
存储某个结点的数据信息	存储该结点的孩子链表的头指针

3） **孩子兄弟表示法：**任意一棵树，它的结点的第一个孩子如果存在就是唯一的，它的右兄弟存在也是唯一的。因此，设置两个指针，分别指向该结点的第一个孩子和此结点的右兄弟。

data(数据域)	firstchild(指针域)	rightsib(指针域)
存储结点的数据信息	存储该结点的第一个孩子的存储地址	存储该结点的右兄弟结点的存储地址

4、树的应用,Huffman 树的定义与应用;

Huffman 树：是一类带权路径长度最短的树

基本概念：

1）树的路径长度：从根到每一结点的路径长度之和。

2）结点的带权路径长度：从该结点到树根之间的路径长度与结点上权的乘积。

3）树的带权路径长度：树中所有叶子结点的带权路径长度之和，通常记做 WPL。

Huffman 算法：

(1) 由给定的 n 个权值{w0, w1, w2, …, wn-1}，构造具有 n 棵二叉树的集合 F = {T0, T1, T2, …, Tn-1}，其中每一棵二叉树 Ti 只有一个带有权值 wi 的根结点，其左、右子树均为空。

(2) 在 F 中选取两棵根结点的权值最小的二叉树，做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

(3) 在 F 中删去这两棵二叉树，加入新得的树。

(4) 重复(2) (3)，直到 F 只含一棵树为止。这棵树就是赫夫曼树

三、查找(search)（感谢小涛同学）

1、查找的基本概念;对线性关系结构的查找,顺序查找,二分查找;

答:

查找定义:

根据给定的某个值（Key），在查找表中确定一个其关键字等于给定值的数据元素（或记录）。

查找算法分类:

1） 静态查找和动态查找;

注:静态或者动态都是针对查找表而言的。动态表指查找表中有删除和插入操作的表。

2） 无序查找和有序查找。

无序查找:被查找数列有序无序均可;

有序查找:被查找数列必须为有序数列。

平均查找长度（Average Search Length，ASL）:需和指定 key 进行比较的关键字的个数的期望值，称为查找算法在查找成功时的平均查找长度。

对于含有 n 个数据元素的查找表，查找成功的平均查找长度为:ASL = Pi*Ci 的和。

Pi:查找表中第 i 个数据元素的概率。 Ci:找到第 i 个数据元素时已经比较过的次数。

顺序查找:

说明:

顺序查找适合于存储结构为顺序存储或链接存储的线性表。

基本思想:

顺序查找也称为线形查找，属于无序查找算法。从数据结构线形表的一端开始，顺序扫描，依次将扫描到的结点关键字与给定值 k 相比较，若相等则表示查找成功;若扫描结束仍没有找到关键字等于 k 的结点，表示查找失败。

复杂度分析:

查找成功时的平均查找长度为:(假设每个数据元素的概率相等） ASL = 1/n(1+2+3+…+n) = (n+1)/2 ;当查找不成功时，需要 n+1 次比较，时间复杂度为 O(n);

所以，顺序查找的时间复杂度为 O(n)。

二分查找:

说明:

元素必须是有序的，如果是无序的则要先进行排序操作。

基本思想:

也称为是折半查找，属于有序查找算法。用给定值 k 先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功;若不相等，再根据 k 与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有

这样的结点。

复杂度分析：

最坏情况下，关键词比较次数为 log2(n+1)，且期望时间复杂度为 O(log2n)；

注：折半查找的前提条件是需要有序表顺序存储，对于静态查找表，一次排序后不再变化，折半查找能得到不错的效率。但对于需要频繁执行插入或删除操作的数据集来说，维护有序的排序会带来不小的工作量，那就不建议使用。

2、Hash 查找法,常见的 Hash 函数(直接定址法,随机数法),hash 冲突的概念，解决冲突的方法(开散列方法/拉链法,闭散列方法/开址定址法),二次聚集现象;

答：

Hash 查找法：

通常我们查找数据都是通过一个一个地比较来进行，有一种方法，要寻找的数据与其在数据集中的位置存在一种对应的关系，通过这种关系就能找到数据的位置。这个对应关系成为散列函数（哈希函数），因此建立的表为散列表（哈希表）。

散列查找是关键字与在数据集中的位置一一对应，通过这种对应关系能快速地找到数据，散列查找中散列函数的构造和处理冲突的方法尤为重要

常见的 Hash 函数

散列函数的构造

构造哈希表的前提是要有哈希函数，并且这个函数尽可能地减小冲突

（1）直接定址法（考纲点明）

可以取关键字的某个线性函数值为散列地址，即：

$$f(key) = a*key + b$$

这样的哈希函数简单均匀，不会产生冲突，但问题是这需要事先知道关键字的分布情况，适合查找表较小且连续的情况。

（2）数字分析法

该方法在知道关键字的情况下，取关键字的尽量不重复的几位值组成散列地址。

（3）平方取中法

取关键字平方后的中间几位为散列地址

（4）折叠法

将关键字分为位数相等的几部分，最后一部分的位数可以不等，然后把这几部分的值（舍去进位）相加作为散列地址。

（5）除留余数法

该方法为最常用的构造哈希函数方法，对于散列表长为 m 的散列函数公式为

$$f(key) = key \bmod p \ (p<=m)$$

使用除留余数法的一个经验是，若散列表表长为 m，通常 p 为小于或等于表长的最小质数或不包含小于 20 质因子的合数。

实践证明，当 p 取小于散列表长的最大质数时，函数较好。

（6）随机数法（考纲点明）

选择一个随机函数，取关键字的随机函数值作为散列地址。

hash 冲突的概念：

对于不同的关键字可能得到同一哈希地址，即 key1 ≠ key2，而(key1) = (key2)，这种现象称为冲突。

解决冲突的方法：

(1）开放定址法（考纲点明）

一旦发生冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入，公式：

$$fi(key) = (f(key)+di) \bmod m \ (di=1,2,3...m-1)$$

用开放定址法解决冲突的做法是：**当冲突发生时，使用某种探测技术在散列表中形成一个探测序列，沿此序列逐个单元地查找，直到找到给定的关键字，或者碰到一个开放的地址（该地址单元为空）为止(若要插入，在探查到开放的地址，则可将带插入的新节点存入该地址单元)。查找时探测到开放的地址则表明表中无待查的关键字，即查找失败**。

比如说，我们的关键字集合为{12,67,56,16,25,37,22,29,15,47,48,34}，表长为 12。我们用散列函数 f(key) = key mod 12。

当计算前 S 个数{12,67,56,16,25}时，都是没有冲突的散列地址，直接存入：

计算 key = 37 时，发现 f(37) = 1，此时就与 25 所在的位置冲突。

于是我们应用上面的公式 f(37) = (f(37)+1) mod 12 = 2。于是将 37 存入下标为 2 的位置。这其实就是房子被人买了于是买下一间的作法：。

接下来 22,29,15,47 都没有冲突，正常的存入：

到了 key=48，我们计算得到 f(48) = 0，与 12 所在的 0 位置冲突了，不要紧，我们 f(48) = (f(48)+1) mod 12 = 1，此时又与 25 所在的位置冲突。于是 f(48) = (f(48)+2) mod 12=2，还是冲突……一直到 f(48) = (f(48)+6) mod 12 = 6 时，才有空位，机不可失，赶快存入：

我们把这种解决冲突的开放定址法称为线性探测法。

二次探测法

考虑深一步，如果发生这样的情况，当最后一个 key=34，f(key)=10，与 22 所在的

位置冲突，可是 22 后面没有空位置了，反而它的前面有一个空位置，尽管可以不断地求余数后得到结果，但效率很差。

因此我们可以改进 di = 1^2, -1^2, 2^2, -2^2,……, q^2, -q^2 (q <= m/2)，这样就等于是可以双向寻找到可能的空位置。

对于 34 来说，我们取 di 即可找到空位置了。另外增加平方运算的目的是为了不让关键字都聚集在某一块区域。我们称这种方法为二次探测法。

fi(key) = (f(key)+di) MOD m (di = 1^2, -1^2, 2^2, -2^2,……, q^2, -q^2, q <= m/2)

随机探测法

还有一种方法，是在冲突时，对于位移量 di 采用随机函数计算得到，我们称之为随机探测法。

此时一定会有人问，既然是随机，那么查找的时候不也随机生成吗？如何可以获得相同的地址呢？这是个问题。这里的随机其实是伪随机数。

伪随机数是说，如果我们设置随机种子相同，则不断调用随机函数可以生成不会重复的数列，我们在查找时，用同样的随机种子，它每次得到的数列是相同的，相同的 di 当然可以得到相同的散列地址。

fi(key) = (f(key)+di) MOD m (di 是一个随机数列)

总之，开放定址法只要在散列表未填满时，总是能找到不发生冲突的地址，是我们常用的解决冲突的办法。

(2) 再哈希法

再哈希法是当散列地址冲突时，用另外一个散列函数再计算一次，这种方法减少了冲突，但增加了计算的时间。

Hi = RHi(key)， i = 1, 2, … , k (k ⩽ m -1)

R Hi 均是不同的哈希函数，即在同义词产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生。这种方法不容易产生“聚集”，但是增加了计算时间。

(3) 链地址法（拉链法）（考纲点明）

链地址法解决冲突的做法是：将所有关键字散列地址相同的结点链接在同一个单链表中。若选定的散列表长度为 m，则可将散列表定义为一个由 m 个头指针组成的指针数组 T[0..m-1]。凡是散列地址为 i 的结点，均插入到以 T[i]为头指针的单链表中。T 中各分量的初值均应为空指针。在拉链法中，装填因子α可以大于 1，但一般均取α⩽1。

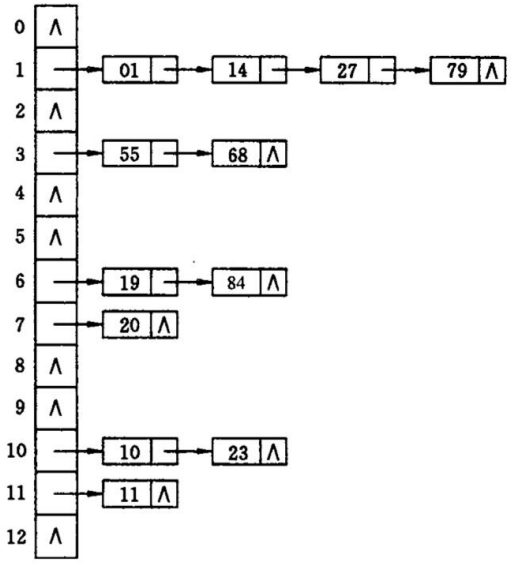


图 9.26　链地址法处理冲突时的哈希表
（同一链表中关键字自小至大有序）

前面我们谈到了散列冲突处理的开放定址法，它的思路就是一旦发生了冲突，就去寻找下一个空的散列地址。那么，有冲突就非要换地方吗？我们直接就在原地处理行不行呢？

可以的，于是我们就有了链地址法。

将所有关键字散列地址相同的记录存储在一个单链表中，我们称这种表为同义词子表，在散列表中只存储所有同义词子表的头指针。

(4) 建立公共溢出区

这种方法的基本思想是：将散列表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

二次聚集现象：

开放定址法会造成二次聚集的现象，对查找不利。

我们可以看到一个现象：当表中 i,i+1,i+2 位置上已经填有记录时，下一个哈希地址为 i,i+1,i+2 和 i+3 的记录都将填入 i+3 的位置，**这种在处理冲突过程中发生的两个第一个哈希地址不同的记录争夺同一个后继哈希地址的现象称为“二次聚集”**，即在处理同义词的冲突过程中又添加了非同义词的冲突。但另一方面，用线性探测再散列处理冲突可以保证做到：只要哈希表未填满，总能找到一个不发生冲突的地址 Hk。而二次探测再散列只有在哈希表长 m 为形如 4j+3（j 为整数）的素数时才可能。

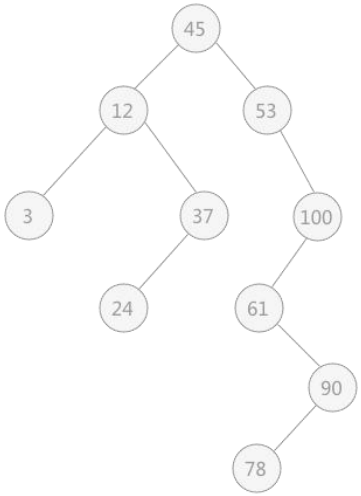
3、BST 树定义,性质,ADT 及其实现,BST 树查找,插入,删除算法;(感谢张寒张寒)

答：

BST 树定义、性质：

二叉排序树要么是空树，要么具有如下特点

- 二叉排序树中，如果其根节点有左子树，那么左子树上的所有节点都小于根节点的值
- 二叉排序树种，如果其根节点有右子树，那么右子树上的所有节点都大于根节点的值
- 二叉排序树的左右子树也要求都是二叉排序树



查找：

二叉排序树中查找某关键字时，查找过程类似于次优二叉树，在二叉排序树不为空树的前提下，首先将被查找值同树的根结点进行比较，会有 3 种不同的结果：

- 如果相等，查找成功；
- 如果比较结果为根结点的关键字值较大，则说明该关键字可能存在其左子树中；
- 如果比较结果为根结点的关键字值较小，则说明该关键字可能存在其右子树中；

插入：

二叉排序树本身是动态查找表的一种表示形式，有时会在查找过程中插入或者删除表中元素，当因为查找失败而需要插入数据元素时，该数据元素的插入位置一定位于二叉排序树的叶子结点，并且一定是查找失败时访问的最后一个结点的左孩子或者右孩子。

通过使用二叉排序树对动态查找表做查找和插入的操作，同时在中序遍历二叉排序树时，可以得到有关所有关键字的一个有序的序列。

一个无序序列可以通过构建一棵二叉排序树，从而变成一个有序序列。

删除：

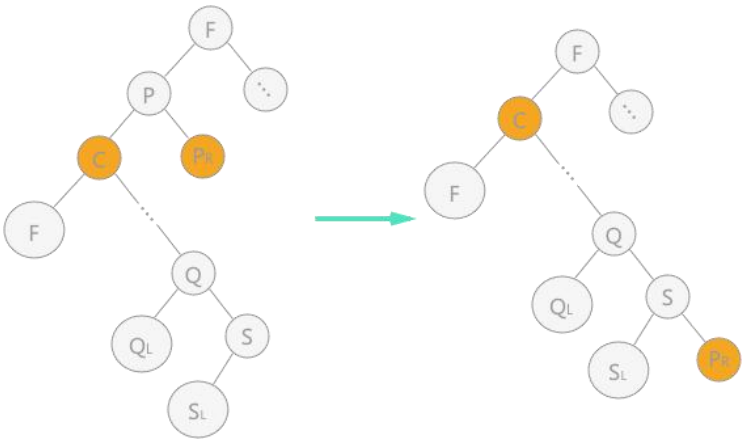
在查找过程中，如果在使用二叉排序树表示的动态查找表中删除某个数据元素时，需要在成功删除该结点的同时，依旧使这棵树为二叉排序树。

假设要删除的为结点 p，则对于二叉排序树来说，需要根据结点 p 所在不同的位置作不同的操作，有以下 3 种可能：

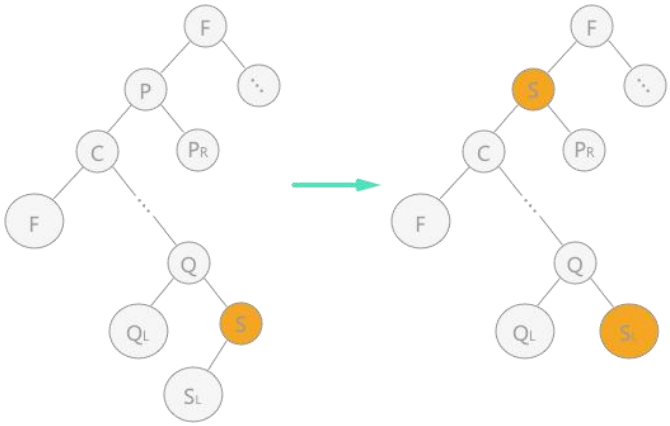
- 结点 p 为叶子结点，此时只需要删除该结点，并修改其双亲结点的指针即可；
- 结点 p 只有左子树或者只有右子树，此时只需要将其左子树或者右子树直接变为结点 p 双亲结点的左子树即可；
- 结点 p 左右子树都有

此时有两种处理方式：

1）令结点 p 的左子树为其双亲结点的左子树；结点 p 的右子树为其自身直接前驱结点的右子树



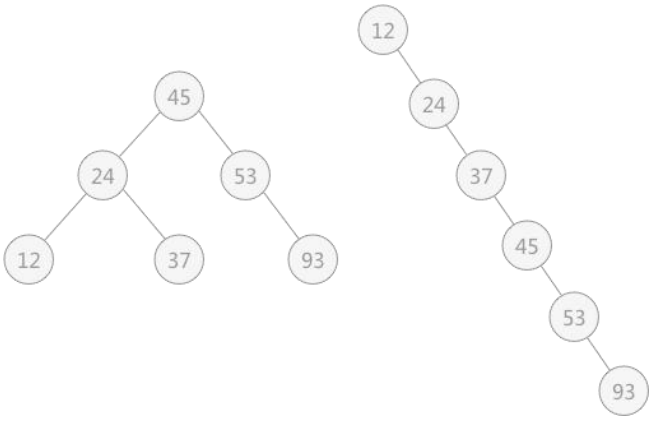
2）用结点 p 的直接前驱（或直接后继）来代替结点 p，同时在二叉排序树中对其直接前驱（或直接后继）做删除操作。为使用直接前驱代替结点 p：



总结

使用二叉排序树在查找表中做查找操作的时间复杂度同建立的二叉树本身的结构有关。即使查找表中各数据元素完全相同，但是不同的排列顺序，构建出的二叉排序树大不相同。

例如：查找表 {45，24，53，12，37，93} 和表 {12，24，37，45，53，93} 各自构建的二叉排序树图下图所示：



不同构造的二叉排序树

使用二叉排序树实现动态查找操作的过程，实际上就是从二叉排序树的根结点到查找元素结点的过程，所以时间复杂度同被查找元素所在的树的深度（层次数）有关。

为了弥补二叉排序树构造时产生如图 5 右侧所示的影响算法效率的因素，需要对二叉排序树做“平衡化”处理，使其成为一棵平衡二叉树。

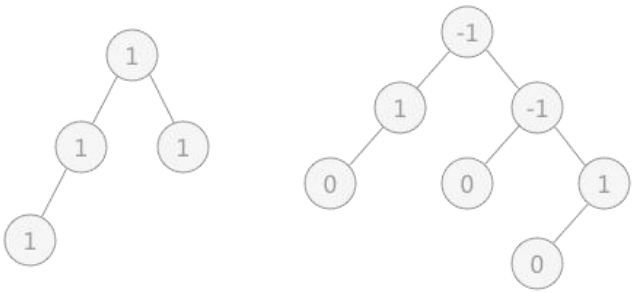
4、平衡树（AVL）的定义,性质,ADT 及其实现,平衡树查找,插入算法,平衡因子的概念;

答：

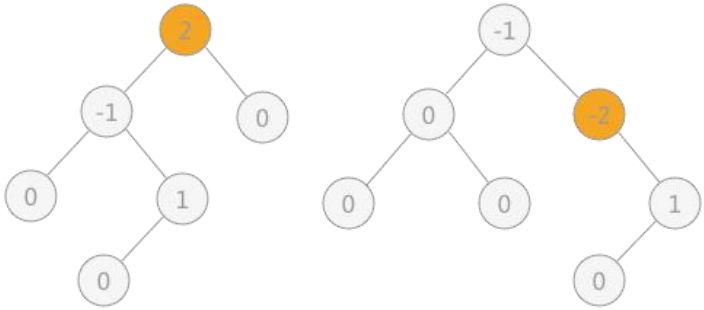
平衡二叉树是遵循以下两个特点的二叉树：

- 每棵子树中的左子树和右子树的深度差不能超过 1
- 二叉树中每棵子树都要求是平衡二叉树

其实就是在二叉树的基础上，使树中每棵子树都满足其左子树和右子树的深度差都不超过 1。



（a）平衡二叉树



（b）不平衡的二叉树

平衡因子：每个结点都有其各自的平衡因子，表示的就是其左子树深度同右子树深度的差。平衡二叉树中各结点平衡因子的取值只可能是：0、1 和 -1。

5、优先队列与堆,堆的定义,堆的生成,调整算法;范围查询;

答：

优先队列

队列是一个操作受限的线性表，数据只能在一端进入，另一端出来，具有先进先出的性质。**有时在队列中需要处理优先级的情况，即后面进入的数据需要提前出来，这里就需要优先队列**。

优先队列是至少能够提供插入和删除最小值这两种操作的数据结构。对应于队列的操作，插入相当于入队，删除最小相当于出队。

链表，二叉查找树，都可以提供插入和删除最小这两种操作。对于链表的实现，插入需要 O(1)，删除最小需要遍历链表，故需要 O(N)。对于二叉查找树，这两种操作都需要 O(logN)；而且随着不停的删除最小的操作，二叉查找树会变得非常不平衡；同时使用二叉查找树有些浪费，因此很多操作根本不需要。一种较好的实现优先队列的方式是二叉堆（下面简称堆）。

堆

堆实质上是满足如下性质的完全二叉树：

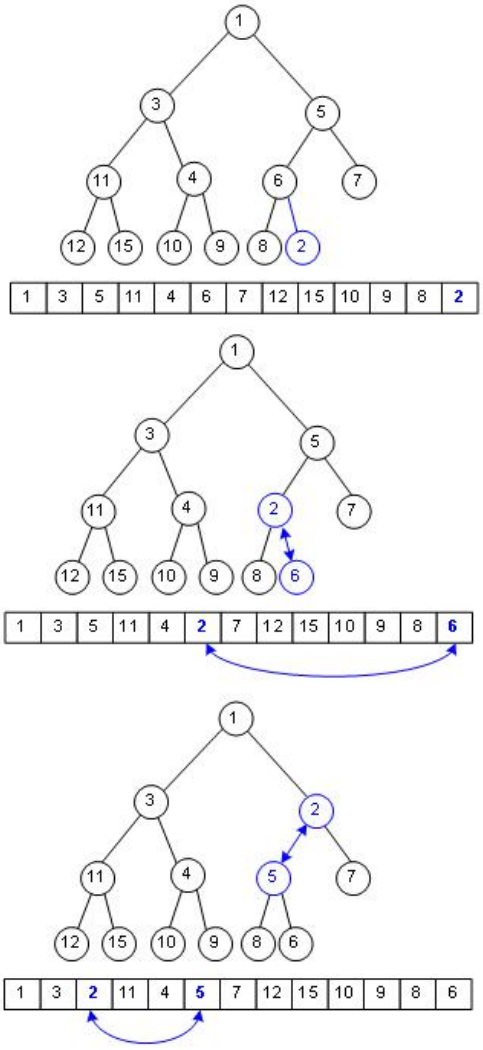
树中任一非叶结点的关键字均不大于（或不小于）其左右孩子（若存在）结点的关键字。首先堆是完全二叉树（只有最下面的两层结点度能够小于 2，并且最下面一层的结点都集中在

该层最左边的若干位置的二叉树），其次任意节点的左右孩子（若有）值都不小于其父亲，这是小根堆，即最小的永远在上面。相反的是大根堆，即大的在上面。

插入：

二叉堆就是一个简单的一维 Int 数组，故不需要初始化，直接插入便可

每次插入都将新数据放到数组的最后的位置



删除：

堆中每次都只能删除第 1 个数据。为了便于重建堆，实际的操作是将最后一个数据的值赋给根结点，然后再从根结点开始进行一次从上向下的调整。调整时先在左右儿子结点中找最小的，如果父结点比这个最小的子结点还小说明不需要调整了，反之将父结点和它交换后再考虑后面的结点。相当于从根结点将一个数据的“下沉”过程。

四、排序

1、 排序基本概念;

重排一个记录序列,使之成为按关键字有序

常见排序可分为以下五类:

- 插入排序（简单插入排序、希尔排序）
- 交换排序（冒泡排序、快速排序）
- 选择排序（简单选择排序、堆排序）
- 归并排序
- 计数排序（多关键字排序）

算法稳定性

- 假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且 r[i]在 r[j]之前，而在排序后的序列中，r[i]仍在 r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。
- 例如，对于如下冒泡排序算法，原本是稳定的排序算法，如果将记录交换的条件改成 r[j]>=r[j+1]，则两个相等的记录就会交换位置，从而变成不稳定的算法。
- 堆排序、快速排序、希尔排序、直接选择排序不是稳定的排序算法，而基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序是稳定的排序算法。

2、 插入排序,希尔排序,选择排序,快速排序,合并排序,基数排序等排序算法基本思想,算法代码及基本的时间复杂度分析

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数。

1.冒泡排序

- 交换排序的一种
- 依次比较相邻的两个待排序元素，若为逆序（递增或递减）则进行交换，将待排序元素从左至右比较一遍称为一趟“冒泡”
- 每趟冒泡都将待排序列中的最大关键字交换到最后（或最前）位置
- 直到全部元素有序为止/直到某次冒泡过程中没有发生交换为止

思路

- 第一次循坏遍历整个数组，找到数组中最大的一个元素，让其和数组中第一个元素交换位置，a[0]=最大元素
- 第二次循坏遍历整个数组，找到数组中次最大一个元素，让其和数组中第二个元素交换位置，a[1]=次最大元素
- 以此类推，遍历完整个数组，

```
public void sort(int[] arr) {
    int temp = 0;
    for (int j = 0; j < arr.length; j++) {
        for (int i = 0; i < arr.length - j - 1; i++) {
            //如果前一个元素大于后一个元素，则需要交换 2 个位置的数据 if
(arr[i] > arr[i + 1]) {
                temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
            }
        }
    }
}
```

2.插入排序

思路

- 创建一个空数组，存放排序的数据
- 从原数组中依次选择的数据
- 在新数组中寻找插入点，
- 如该点没有数据，就将数据插入到该点。否则需把该插入点后面的所有数据向后移动一位，空出位置，插入数据。

```
public void sort(int[] arr) {
    int cur,pIdx;
```

```
        for(int i=1;i<arr.length;i++){
            //基准元素
            cur=arr[i];
            pIdx=i-1;
            //如果前一个元素大于基准点值，不断向前寻找插入点，
                    while(pIdx>=0 && arr[pIdx]>cur){
                arr[pIdx+1]=arr[pIdx];
                pIdx--;
            }
            arr[pIdx+1]=cur;
        }
    }
```

3. 希尔排序

- 缩小增量排序
- 其是插入排序的改进版，改进点：减少了插入排序时移动元素次数。

基本思想

先将整个待排记录序列分割为若干子序列分别进行直接插入排序，待整个序列的记录“基本有序”时，再对全体记录进行一次直接插入排序

```
    public void sort(int[] arr) {
        //gap=arr.length/2, gap 以每次/2 的方式递减
            for(int gap=arr.length/2;gap>0;gap=gap/2){
             //以 gap 元素开始，对 gap 中的一组数据进行排序
                for(int i=gap;i<arr.length;i++){
                int cur=i;
                int temp=arr[cur];
                if(arr[cur]<arr[cur-gap]){
                    while(cur-gap>=0 && temp<arr[cur-gap]){
                        arr[cur]=arr[cur-gap];
                        cur-=gap;
                    }
                    arr[cur]=temp;
                }
            }
        }
```

```
    }
```

4.快速排序

基本概念

- 就平均时间而言，快速排序是目前被认为最好的一种内部排序方法,由 C. A. R. Hoare 发明
- 分治法(divide and conquer)思想的体现
- Unix 系统函数库所提供的标准排序方法
- C 标准函数库中的排序方法直接就命名为 qsort()
- 轴值(pivot):
  - 书上称枢轴
  - 用于将记录集"分割"为两个部分的那个键值
- 分割(partition):
  - 将记录集分为两个部分,前面部分记录的键值都比轴值小,后面部分的键值都比轴值大

基本思想

是对冒泡排序的一种改进 选取一个轴值,然后根据此轴值通过一趟排序对记录集进行一次分割,然后对分割后产生的两个记录子集分别进行快速排序

```
    @Overridepublic void sort(int[] arr) {
        qSort(arr,0,arr.length-1);
    }

    private void qSort(int[] arr,int l,int r){
        int i=l;
        int j=r;
        if(i>j){
            return;
        }
        int temp=arr[l];
        while(i!=j ){
            //如果当前元素大于基准元素，指针向左移动一位，直到找到一个小于基准元素为止，此时记录下标
                while(arr[j]>temp && i<j){
                 j--;
            }
```

```
            //如果当前元素小于基准元素，指针向右移动一位，直到找到一个大于基准元素为止，此时记录下标
                while(arr[i]<=temp && i<j){
                i++;
            }
            //交换 2 个下标的元素
                if(i<j){
                swap(arr,i,j);
            }
        }
        swap(arr,l,i);
        qSort(arr,l,i-1);
        qSort(arr,i+1,r);
    }

    private void swap(int[] arr,int i, int j){
        int temp=arr[i];
        arr[i]=arr[j];
        arr[j]=temp;
    }
```

5. 选择排序

基本思想

- 是每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。 选择排序是不稳定的排序方法
- 与冒泡排序不同点：减少了元素交换次数

```
/**
 * 选择排序算法，从小到大的方式开始排序
 * 实现思路：
 * 第一次循环找到数组的最小的一个元素，放在数组的第一个位置
 * 第二次循环找到数组中第二小的一个元素，放在数组的第二个位置
 * 第三次循环找到数组中第三小的一个元素，放在数组的第三个位置
 * 以此类推，直到走完数组中所有元素
 * 与冒泡最大区别，交换元素次数会较少
```

```
 */
public void sort(int[] arr) {
    int minIdx=0,temp=0;
    for(int i=0;i<arr.length;i++){
        //需要找到余下数组中最小的一个元素
            minIdx=i;
                for(int j=i+1;j<arr.length;j++){
            //假设第一个元素是最小值，余下元素如果比第一个元素小，
            // 记录小的元素下标，剩下元素再和这个最小元素比较，从而找到更小
的元素 if(arr[j]<arr[minIdx]){
                    minIdx=j;
            }
        }
        temp=arr[i];arr[i]=arr[minIdx];arr[minIdx]=temp;
    }
}
```

6.计数排序

- 是一种基于非比较的排序算法,该算法于 1954 年由 Harold H. Seward 提出。它的优势在于在对一定范围内的整数排序时，它的复杂度为O(n+k)（其中 k 是整数的范围），快于任何比较排序算法。 一种牺牲空间换取时间的做法。
- 只适合数字类型的排序

基本思想

- 开启额外的空间，来存储数组中元素
- 旧数组种元素（数字）作为新数组的下表，并记录相同元素的个数
- 最后将新数组反向输出，从而得到有序的数组

```
/**
 * 计数排序算法，从小到大的方式开始排序
 * 实现思路：
 * 只适合数字类型的排序
 * 会开启额外的空间，来存储数组中元素
 * 旧数组种元素（数字）作为新数组的下表，并记录相同元素的个数
 * 最后将新数组反向输出，从而得到有序的数组
 */
```

```
1.      private void sort(int[] arr,int maxVal){
1.          int[] newArr=new int[maxVal+1];
1.          for(int i=0;i<arr.length;i++){
1.              newArr[arr[i]]+=1;
1.          }
1.          int idx=0;
1.          //反向输出数组种的元素
1.              for(int i=0;i<newArr.length;i++){
1.               int cnt=newArr[i];
1.               while(cnt>0){
1.                   arr[idx++]=i;
i.                  cnt--;
ii.              }
iii.        }
iv.     }
```

7.桶排序

基本思想

将数组分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序）

```
/**
 * 桶排序算法，从小到大的方式开始排序
 * <ul>
 * 实现思路：将一个数组尽量拆分成一个个小数组（桶），在对桶里面元素进行排序，这个排序可以各种类型排序：插入，快排等等
 * 和计数排序不同：计数排序只能用于数字，每个元素作为新数组下标
 * </ul>
 * @param arr
 */
@Override
public void sort(int[] arr) {
    bucketSort(arr);
}

private void bucketSort(int[] arr){
```

```
        int minVal=arr[0];
        int maxVal=arr[0];
        //求出数组中最大值和最小值
            for(int i=0;i<arr.length;i++){
             if(arr[i]<minVal){
                    minVal=arr[i];
             }else if(arr[i]>maxVal){
                    maxVal=arr[i];
             }
        }
        //桶数
            int bucketNum = (maxVal - minVal) / arr.length + 1;
        //创建一个二维数组存放桶和桶中的元素
        ArrayList<ArrayList<Integer>> bucketArr = new
    ArrayList<>(bucketNum);
        for(int i = 0; i < bucketNum; i++){
              bucketArr.add(new ArrayList<Integer>());
        }
        //将每个元素放入桶
            for(int i = 0; i < arr.length; i++){
             //这样分法，尽量保证每个桶中分到元素，尽量的少 int num =
    (arr[i] - minVal) / (arr.length);
             bucketArr.get(num).add(arr[i]);
        }
        //对每个桶进行排序
            for(int i = 0; i < bucketArr.size(); i++){
             Collections.sort(bucketArr.get(i));
        }
        //最后输出桶种元素
        System.out.println(bucketArr.toString());
    }
```

8.归并排序

基本思想

- 利用归并的思想实现的排序方法，该算法采用经典的分治

（divide-and-conquer）策略（分治法将问题分(divide)成一些小的问题然后递归求解，而治(conquer)的阶段则将分的阶段得到的各答案"修补"在一起，即分而治之)

思路

- 分阶段
  - 类似于二分查找：将一个大数组折半拆分成二个数组
  - 再对这 2 个数组进行折半拆分 4 个小数组
  - ...
  - 直到小数组（长度=1）不可以再拆分
  - 最后交换左右 2 个子数组来排序
- 治阶段
  - 将 2 个子数组合并为一个有序数组
  - 创建一个临时 temp 数组，长度为 2 个子数组长度和
  - 最终完成 2 个子数组的合并

```
public static void sort(int []arr){
    int []temp = new int[arr.length];//在排序前，先建好一个长度等于原数组长度的临时数组，避免递归中频繁开辟空间
    sort(arr,0,arr.length-1,temp);
}
private static void sort(int[] arr,int left,int right,int []temp){
    if(left<right){
        int mid = (left+right)/2;
        sort(arr,left,mid,temp);//左边归并排序，使得左子序列有序
        sort(arr,mid+1,right,temp);//右边归并排序，使得右子序列有序
        merge(arr,left,mid,right,temp);//将两个有序子数组合并操作
    }
}
private static void merge(int[] arr,int left,int mid,int right,int[] temp){
    int i = left;//左序列指针
        int j = mid+1;//右序列指针
        int t = 0;//临时数组指针
    while (i<=mid && j<=right){
        if(arr[i]<=arr[j]){
            temp[t++] = arr[i++];
```

```
            }else {
                temp[t++] = arr[j++];
            }
        }
        while(i<=mid){//将左边剩余元素填充进 temp 中
            temp[t++] = arr[i++];
        }
        while(j<=right){//将右序列剩余元素填充进 temp 中
            temp[t++] = arr[j++];
        }
        t = 0;
        //将 temp 中的元素全部拷贝到原数组中
        while(left <= right){
            arr[left++] = temp[t++];
        }
    }
```

五、图

2. 图的基本概念

答：

顶点：使用图表示的每个数据元素称作顶点

顶点之间的关系有两种“有向图和无向图”，如图 1 中的两个图所示：

（a）中顶点 V1 和 V2 只有单方向的关系，只能通过 V1 找到 V2，反过来行不通，因此两顶点之间的关系表示为：<V1,V2>；

（b）中顶点之间具有双向的关系，之间用直线连通，对于 V1 和 V2 顶点来说，既可以通过 V1 找到 V2，也可以通过 V2 找到 V1，两顶点之间的关系表示为：(V1,V2)。

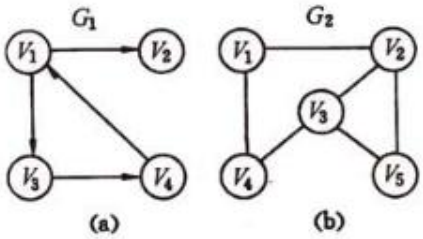


图 7.1 图的示例

（a）有向图 G_1；（b）无向图 G_2

图 1 有向图和无向图

“弧”和“边”：

在有向图中，<v,w>表示为从 v 到 w 的一条弧；在无向图中，(v,w)表示为顶点 v 和顶点 w 之间的一条边

完全图 ：

对于无向图来说，如果图中每个顶点都和除自身之外的所有顶点有关系，那么就称这样的无向图为完全图。如图 2 所示就是一个完全图。

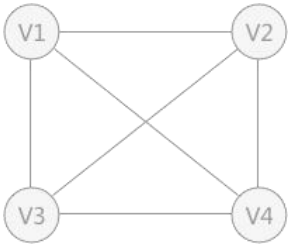


图 2 完全图

对于有 n 个顶点的完全图，其中的边的数目为： $\frac{1}{2}n(n-1)$

3. **图的存储结构,邻接矩阵,邻接表;**

答：

顺序存储结构+三种链式存储结构（邻接表，邻接多重表，十字链表）

4. **数组表示法：(邻接矩阵)**

使用数组存储图时，需要使用两个数组，一个数组存放图中顶点本身的数据(一维数组)，另外一个数组用于存储各顶点之间的关系（二维数组）。

不同类型的图，存储的方式略有不同，根据图有无权，可以将图划分为两大类：图和网

图，包括无向图和有向图。在使用二维数组存储图中顶点之间的关系时，如果顶点之间存在边或弧，在相应位置用 1 表示，反之用 0 表示。

$$G1.arcs=\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix},\quad G2.arcs=\begin{bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{bmatrix}$$

图 7.8 图的邻接矩阵

网，是指带权的图，包括无向网和有向网。使用二维数组存储网中顶点之间的关系，顶点之间如果有边或者弧的存在，在数组的相应位置存储其权值；反之用 ∞ 表示。

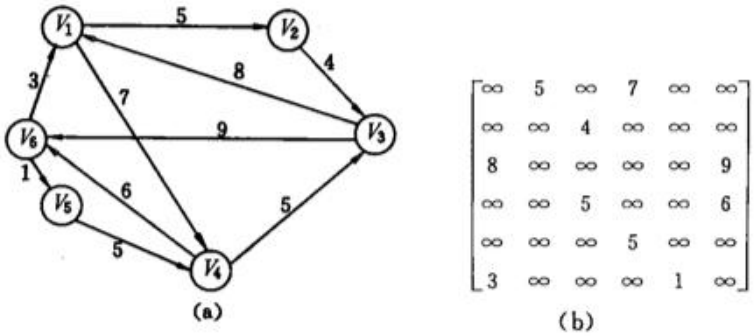


图 7.9　网及其邻接矩阵

(a) 网 N；(b) 邻接矩阵

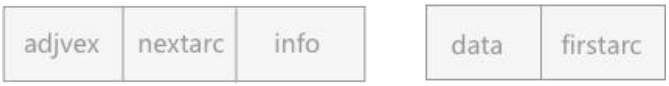
(2)邻接表

邻接表是图的一种链式存储结构。使用邻接表存储图时，对于图中的每一个顶点和它相关的邻接点，都存储到一个链表中。每个链表都配有头结点，头结点的数据域不为 NULL，而是用于存储顶点本身的数据；后续链表中的各个结点存储的是当前顶点的所有邻接点。

所以，采用邻接表存储图时，有多少顶点就会构建多少个链表，为了便于管理这些链表，常用的方法是将所有链表的链表头按照一定的顺序存储在一个数组中（也可以用链表串起来）。

在邻接表中，每个链表的头结点和其它结点的组成成分有略微的不同。

头结点需要存储每个顶点的数据和指向下一个结点的指针，由两部分构成：而在存储邻接点时，由于各个顶点的数据都存储在数组中，所以每个邻接点只需要存储自己在数组中的位置下标即可。另外还需要一个指向下一个结点的指针。除此之外，如果存储的是网，还需要一个记录权值的信息域。所以表头结点和其它结点的构造分别为：



（1）表中结点　　（2）表头结点

表结点结构

info 域对于无向图来说，本身不具备权值和其它相关信息，就可以根据需要将之删除。

例如，当存储图 2（A）所示的有向图时，构建的邻接表如图 2（B）所示：

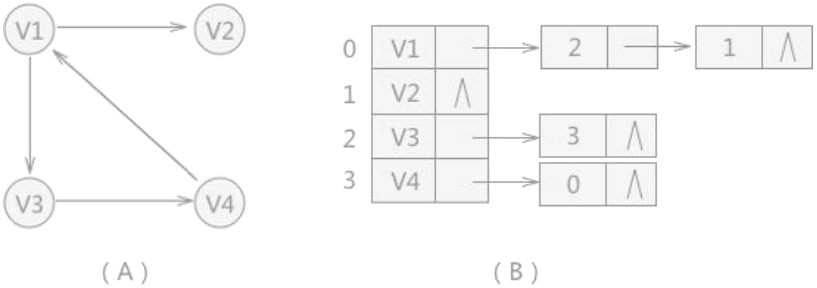
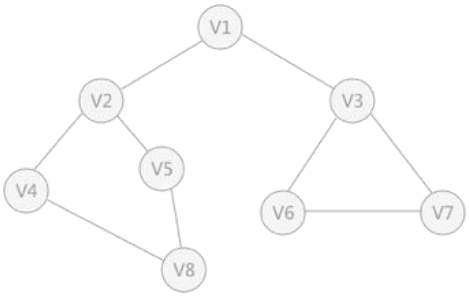


图 2 有向图和对应的邻接表

5. 图的遍历,广度度优先遍历和深度优先遍历;

答:

深度优先遍历（DFS）



深度优先搜索的过程类似于树的先序遍历，首先从例子中体会深度优先搜索。例如图 1 是一个无向图，采用深度优先算法遍历这个图的过程为:

6. 首先任意找一个未被遍历过的顶点，例如从 V1 开始，由于 V1 率先访问过了，所以，需要标记 V1 的状态为访问过;
7. 然后遍历 V1 的邻接点，例如访问 V2 ，并做标记，然后访问 V2 的邻接点，例如 V4（做标记），然后 V8 ，然后 V5 ；
8. 当继续遍历 V5 的邻接点时，根据之前做的标记显示，所有邻接点都被访问过了。此时，从 V5 回退到 V8 ，看 V8 是否有未被访问过的邻接点，如果没有，继续回退到 V4 ， V2 ， V1 ；
9. 通过查看 V1 ，找到一个未被访问过的顶点 V3 ，继续遍历，然后访问 V3 邻接点 V6 ，然后 V7 ；
10. 由于 V7 没有未被访问的邻接点，所有回退到 V6 ，继续回退至 V3 ，最后到达 V1 ，发现没有未被访问的;
11. 最后一步需要判断是否所有顶点都被访问，如果还有没被访问的，以未被访问的顶点为第一个顶点，继续依照上边的方式进行遍历。

所谓深度优先搜索，是从图中的一个顶点出发，每次遍历当前访问顶点的临界点，一直到访

问的顶点没有未被访问过的临界点为止。然后采用依次回退的方式，查看来的路上每一个顶点是否有其它未被访问的临界点。访问完成后，判断图中的顶点是否已经全部遍历完成，如果没有，以未访问的顶点为起始点，重复上述过程。

广度优先遍历

广度优先搜索类似于树的层次遍历。从图中的某一顶点出发，遍历每一个顶点时，依次遍历其所有的邻接点，然后再从这些邻接点出发，同样依次访问它们的邻接点。按照此过程，直到图中所有被访问过的顶点的邻接点都被访问到。

最后还需要做的操作就是查看图中是否存在尚未被访问的顶点，若有，则以该顶点为起始点，重复上述遍历过程。

还拿图 1 中的无向图为例，假设 V1 作为起始点，遍历其所有的邻接点 V2 和 V3 ，以 V2 为起始点，访问邻接点 V4 和 V5 ，以 V3 为起始点，访问邻接点 V6 、 V7 ，以 V4 为起始点访问 V8 ，以 V5 为起始点，由于 V5 所有的起始点已经全部被访问，所有直接略过， V6 和 V7 也是如此。

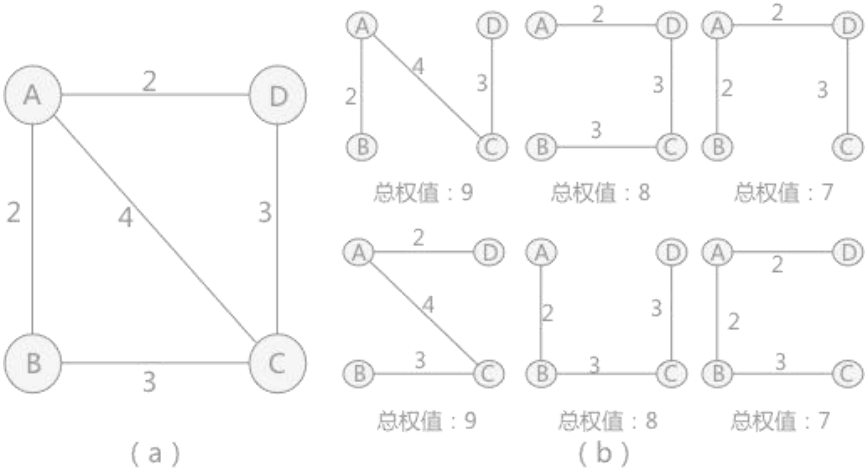
深度优先搜索算法的实现运用的主要是回溯法，类似于树的先序遍历算法。广度优先搜索算法借助队列的先

先出的特点，类似于树的层次遍历。

12. 最小生成树基本概念,Prim 算法,Kruskal 算法;最短路径问题,广度优先遍历算法,Dijkstra 算法,Floyd 算法;拓扑排序

答：

（1）最小生成树问题



假设通过综合分析，城市之间的权值如图 2（a）所示，对于（b）的方案中，选择权值总和为 7 的两种方案最节约经费

简单得理解就是给定一个带有权值的连通图（连通网），如何从众多的生成树中筛选出权值总和最小的生成树，即为该图的最小生成树

给定一个连通网，求最小生成树的方法有：普里姆（Prim）算法和克鲁斯卡尔（Kruskal）算法

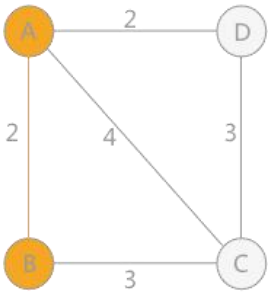
Prim 算法：

普里姆算法在找最小生成树时，将顶点分为两类，一类是在查找的过程中已经包含在树中的（假设为 A 类），剩下的是另一类（假设为 B 类）。

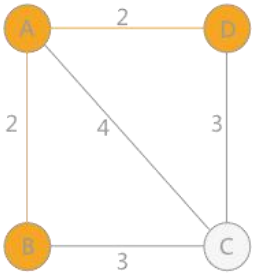
对于给定的连通网，起始状态全部顶点都归为 B 类。在找最小生成树时，选定任意一个顶点作为起始点，并将之从 B 类移至 A 类；然后找出 B 类中到 A 类中的顶点之间权值最小的顶点，将之从 B 类移至 A 类，如此重复，直到 B 类中没有顶点为止。所走过的顶点和边就是该连通图的最小生成树。

例如，通过普里姆算法查找图 2（a）的最小生成树的步骤为：

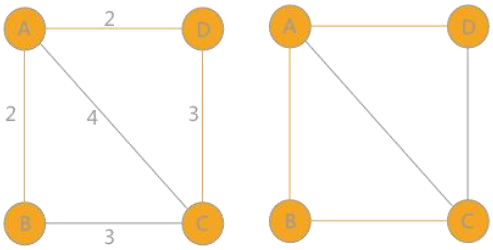
假如从顶点 A 出发，顶点 B、C、D 到顶点 A 的权值分别为 2、4、2，所以，对于顶点 A 来说，顶点 B 和顶点 D 到 A 的权值最小，假设先找到的顶点 B：



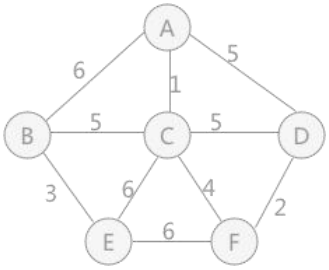
继续分析顶点 C 和 D，顶点 C 到 B 的权值为 3，到 A 的权值为 4；顶点 D 到 A 的权值为 2，到 B 的权值为无穷大（如果之间没有直接通路，设定权值为无穷大）。所以顶点 D 到 A 的权值最小：



最后，只剩下顶点 C，到 A 的权值为 4，到 B 的权值和到 D 的权值一样大，为 3。所以该连通图有两个最小生成树：



例子：



此图结果应为：A-C, C-F, F-D, C-B, B-E

普里姆算法的运行效率只与连通网中包含的顶点数相关，而和网所含的边数无关。所以普里姆算法适合于解决边稠密的网，该算法运行的时间复杂度为：O(n2)

Kruskal 算法：

克鲁斯卡尔算法的具体思路是：将所有边按照权值的大小进行升序排序，然后从小到大一一判断，条件为：如果这个边不会与之前选择的所有边组成回路，就可以作为最小生成树的一部分；反之，舍去。直到具有 n 个顶点的连通网筛选出来 n-1 条边为止。筛选出来的边和所有的顶点构成此连通网的最小生成树。

判断是否会产生回路的方法为：在初始状态下给每个顶点赋予不同的标记，对于遍历过程的每条边，其都有两个顶点，判断这两个顶点的标记是否一致，如果一致，说明它们本身就处在一棵树中，如果继续连接就会产生回路；如果不一致，说明它们之间还没有任何关系，可以连接。

假设遍历到一条由顶点 A 和 B 构成的边，而顶点 A 和顶点 B 标记不同，此时不仅需要

将顶点 A 的标记更新为顶点 B 的标记,还需要更改所有和顶点 A 标记相同的顶点的标记，全部改为顶点 B 的标记。

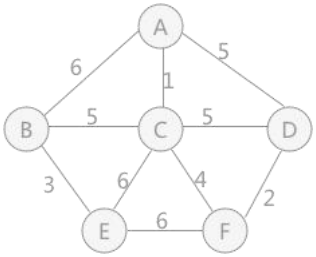
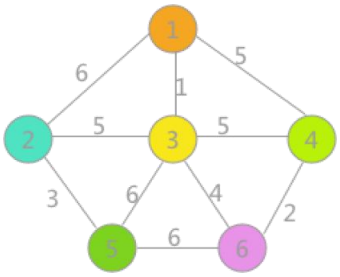


图 1 连通网

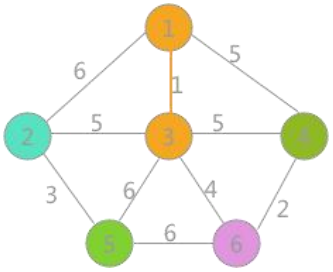
例如，使用克鲁斯卡尔算法找图 1 的最小生成树的过程为：

首先，在初始状态下，对各顶点赋予不同的标记（用颜色区别），如下图所示：



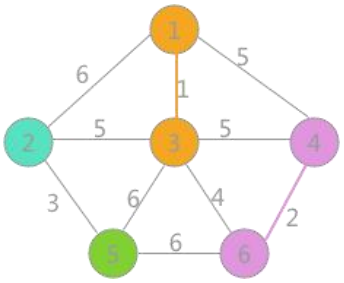
（1）

对所有边按照权值的大小进行排序，按照从小到大的顺序进行判断，首先是（1，3），由于顶点 1 和顶点 3 标记不同，所以可以构成生成树的一部分，遍历所有顶点，将与顶点 3 标记相同的全部更改为顶点 1 的标记，如（2）所示：



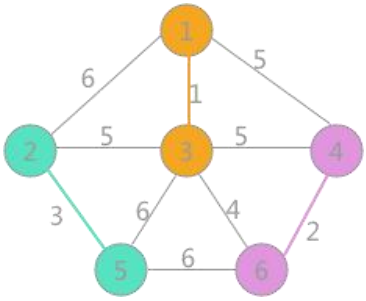
（2）

其次是（4，6）边，两顶点标记不同，所以可以构成生成树的一部分，更新所有顶点的标记为：



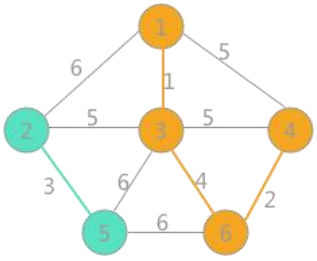
（3）

其次是（2，5）边，两顶点标记不同，可以构成生成树的一部分，更新所有顶点的标记为：



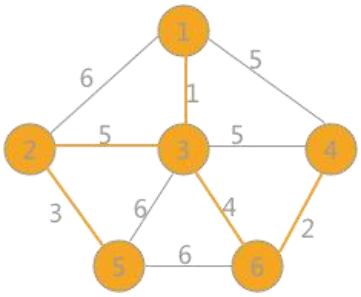
（4）

然后最小的是（3，6）边，两者标记不同，可以连接，遍历所有顶点，将与顶点 6 标记相同的所有顶点的标记更改为顶点 1 的标记：



（5）

继续选择权值最小的边，此时会发现，权值为 5 的边有 3 个，其中（1，4）和（3，4）各自两顶点的标记一样，如果连接会产生回路，所以舍去，而（2，3）标记不一样，可以选择，将所有与顶点 2 标记相同的顶点的标记全部改为同顶点 3 相同的标记：



（6）

当选取的边的数量相比与顶点的数量小 1 时，说明最小生成树已经生成。所以最终采用克鲁斯卡尔算法得到的最小生成树为（6）所示

总结：

普里姆算法。该算法从顶点的角度为出发点，时间复杂度为 $O(n^2)$，更适合与解决边的绸密度更高的连通网。

克鲁斯卡尔算法，从边的角度求网的最小生成树，时间复杂度为 O(eloge)。和普里姆算法恰恰相反，更适合于求边稀疏的网的最小生成树

（2）最短路径问题

在一个网（有权图）中，求一个顶点到另一个顶点的最短路径的计算方式有两种：迪杰斯特拉（Dijkstra 算法）和弗洛伊德（Floyd）算法。迪杰斯特拉算法计算的是有向网中的某个顶点到其余所有顶点的最短路径；弗洛伊德算法计算的是任意两顶点之间的最短路径。

迪杰斯特拉（Dijkstra 算法）:

迪杰斯特拉算法计算的是从网中一个顶点到其它顶点之间的最短路径问题。

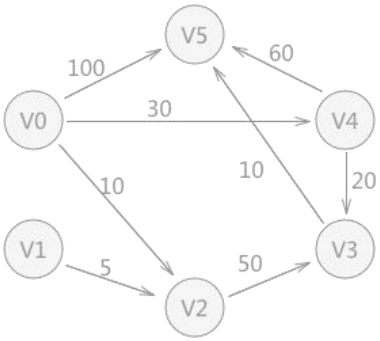


图 1 带权有向图

如图 1 所示是一个有向网，在计算 V0 到其它所有顶点之间的最小路径时，迪杰斯特拉算法的计算方式为：

从 V0 出发，由于可以直接到达 V2 和 V5 ，而其它顶点和 V0 之间没有弧的存在，所以之间的距离设定为无穷大，可以得到下面这个表格：

	V1	V2	V3	V4	V5
V0	∞	10	∞	30	100
涉及到的边	V0-V1	V0-V2	V0-V3	V0-V4	V0-V5

从表格中可以看到，V0 到 V2 的距离最近，所以迪杰斯特拉算法设定 V0-V2 为 V0 到 V2 之间的最短路径，最短路径的权值和为 10。

已经判断 V0-V2 是最短路径，所以以 V2 为起始点，判断 V2 到除了 V0 以外的其余各点之间的距离，如果对应的权值比前一张表格中记录的数值小，就说明网中有一条更短的路径，直接更新表格；反之表格中的数据不变。可以得到下面这个表格：

	V1	V2	V3	V4	V5
V0	∞	10	60	30	100
	V0-V1	V0-V2	V0V2-V3	V0-V4	V0-V5

例如，表格中 V0 到 V3 的距离，发现当通过 V2 到达 V3 的距离比之前的 ∞ 要小，所以更新表格。

更新之后，发现 V0-V4 的距离最近，设定 V0 到 V4 的最短路径的值为 30。之后从 V4 出发，判断到未确定最短路径的其它顶点的距离，继续更新表格：

	V1	V2	V3	V4	V5
V0	∞	10	50	30	90
	V0-V1	V0-V2	V0-V4-V3	V0-V4	V0-V4-V5

更新后确定从 V0 到 V3 的最短路径为 V0-V4-V3，权值为 50。然后从 V3 出发，继续判断：

	V1	V2	V3	V4	V5
V0	∞	10	50	30	60
	V0-V1	V0-V2	V0-V4-V3	V0-V4	V0-V4-V3-V5

对于 V5 来说，通过 V0-V4-V3-V5 的路径要比之前的权值 90 还要小，所以更新表格，更新后可以看到，V0-V5 的距离此时最短，可以确定 V0 到 V5 的最短路径为 60。

最后确定 V0-V1 的最短路径，由于从 V0 无法到达 V1 ，最终设定 V0 到 V1 的最短路径为 ∞（无穷大）。

在确定了 V0 与其他所有顶点的最短路径后，迪杰斯特拉算法才算结束。

事例中借用了图 1 的有向网对迪杰斯特拉算法进行了讲解，实际上无向网中的最短路径问题也可以使用迪杰斯特拉算法解决，解决过程和上述过程完全一致。

总结

迪杰斯特拉算法解决的是从网中的一个顶点到所有其它顶点之间的最短路径，算法整体的时间复杂度为 O(n2)。但是如果需要求任意两顶点之间的最短路径，使用迪杰斯特拉算法虽然最终虽然也能解决问题，但是大材小用，相比之下使用弗洛伊德算法解决此类问题会更合适。

弗洛伊德（Floyd）算法：

弗洛伊德的核心思想是：对于网中的任意两个顶点（例如顶点 A 到顶点 B）来说，之间的最短路径不外乎有 2 种情况：

13. 直接从顶点 A 到顶点 B 的弧的权值为顶点 A 到顶点 B 的最短路径；
14. 从顶点 A 开始，经过若干个顶点，最终达到顶点 B，期间经过的弧的权值和为顶点 A 到顶点 B 的最短路径

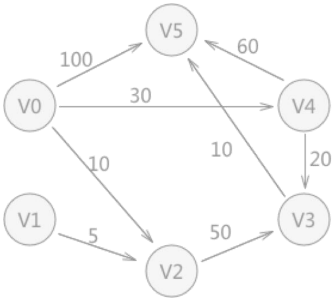


图 1 带权图

例如，在使用弗洛伊德算法计算图 1 中的任意两个顶点之间的最短路径时，具体实施步骤为：

首先，记录顶点之间初始的权值，如下表所示：

	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	∞	30	100
V1	∞	∞	5	∞	∞	∞
V2	∞	∞	∞	50	∞	∞
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	60
V5	∞	∞	∞	∞	∞	∞

依次遍历所有的顶点，假设从 V0 开始，将 V0 作为中间点，看每对顶点之间的距离值是否会更小。最终 V0 对于每对顶点之间的距离没有任何改善。

对于 V0 来说，由于该顶点只有出度，没有入度，所以没有作为中间点的可能。同理，V1 也没有可能。

将 V2 作为每对顶点的中间点，有影响的为 （V0，V3） 和 （V1，V3）：

例如，（V0，V3）权值为无穷大，而（V0，V2）+（V2，V3）= 60，比之前的值小，相比而言后者的路径更短；同理 （V1，V3）也是如此。

更新的表格为：

	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	60	30	100
V1	∞	∞	5	55	∞	∞
V2	∞	∞	∞	50	∞	∞
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	60
V5	∞	∞	∞	∞	∞	∞

以 V3 作为中间顶点遍历各队顶点，更新后的表格为：

	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	60	30	70
V1	∞	∞	5	55	∞	65
V2	∞	∞	∞	50	∞	60
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	30
V5	∞	∞	∞	∞	∞	∞

以 V4 作为中间顶点遍历各队顶点，更新后的表格为：

	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	50	30	60
V1	∞	∞	5	55	∞	65
V2	∞	∞	∞	50	∞	60
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	30
V5	∞	∞	∞	∞	∞	∞

而对于顶点 V5 来说，和顶点 V0 和 V1 相类似，所不同的是，V5 只有入度，没有出度，所以对各队顶点的距离不会产生影响。最终采用弗洛伊德算法求得的各个顶点之间的最短路径如上图所示。

该算法相比于使用迪杰斯特拉算法在解决此问题上的时间复杂度虽然相同，都为 $O(n^3)$，但是弗洛伊德算法的实现形式更简单

（3）拓扑排序

对有向无环图进行拓扑排序，只需要遵循两个原则：

15. 在图中选择一个没有前驱的顶点 V；
16. 从图中删除顶点 V 和所有以该顶点为尾的弧。

例如，在对图 1 中的左图进行拓扑排序时的步骤如图 2 所示：

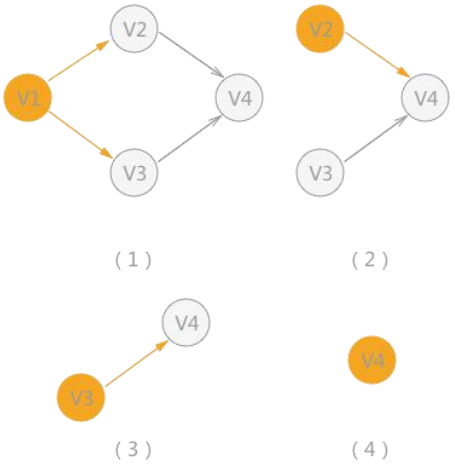


图 2 拓扑排序

有向无环图如果顶点本身具有某种实际意义，例如用有向无环图表示大学期间所学习的全部课程，每个顶点都表示一门课程，有向边表示课程学习的先后次序，例如要先学《程序设计基础》和《离散数学》，然后才能学习《数据结构》。所以用来表示某种活动间的优先关系的有向图简称为“AOV 网”。

进行拓扑排序时，首先找到没有前驱的顶点 V1，如图 2（1）所示；在删除顶点 V1 及以 V1 作为起点的弧后，继续查找没有前驱的顶点，此时， V2 和 V3 都符合条件，可以随机选择一个，例如图 2（2） 所示，选择 V2 ，然后继续重复以上的操作，直至最后找不到没有前驱的顶点。

所以，针对图 2 来说，拓扑排序最后得到的序列有两种：

- V1 -> V2 -> V3 -> V4
- V1 -> V3 -> V2 -> V4

第二部分计算机系统基础

考试题型：问答、分析、编程

总分：40 分

一、处理器体系结构 （感谢小鱼）

17. CPU 中的时序电路

答：CPU 中的时序电路：通过 RS 触发器控制 CPU 的时序。

2、单周期处理器的设计、

答：CPU 在处理指令时，一般需要经过以下几个步骤：

(1) **取指令(IF)**：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

(2) **指令译码(ID)**：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) **指令执行(EXE)**：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) **存储器访问(MEM)**：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) **结果写回(WB)**：指令执行的结果或者访问存储器中得到的数据写回

相应的目的寄存器中。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

18. 流水线处理器的基本原理

答：流水线(Pipeline)技术是指程序在执行时候多条指令重叠进行操作的一种准并行处理实现技术。通俗的讲将一个时序过程，分解成若干个子过程，每个过程都能有效的与其他子过程同时执行。旨在提高处理器处理效率，争取在一个时钟周期中完成一条指令。

将处理组织成阶段：取指、译码、执行、访存、写回。

通常一条指令包含很多操作，可以将它们组织成一定的阶段序列，从而便于放入一个通用框架来进行流水线处理。

参考：

流水线处理器的基本原理：

https://blog.csdn.net/pankul/article/details/8769979

4、Data Hazard 的处理

答：流水线给处理器带来了效率，当然也有问题。称之为流水线冒险(HaZard)。

1、结构冒险

由于处理器资源冲突，而无法实现某些指令或者阶段的组合实现，就称之为处理器有结构冒险。

比如，早期的处理器中，程序和数据是存储在一起的，那么容易出现下图的情况：在第四个 cycle 中，IF 和 MEM 同时访问存储器导致有一个操作要等待，此时 hazard 就出现了。现在的处理器已经解决了该问题：指令存储在 L1P cache 中，数据存储 L1D cache 中，单独访问，不会影响相互操作。

2、数据冒险

如果流水线中原来有先后顺序的指令同一时刻处理时，可能会导致出现访问了错误的数据的情况。

在汇编语句中，add R1,R2,R3 将寄存器 R2 和 R3 的和赋予 R1，改变 R1 的值；而紧接着下面的语句：add R4,R1,R5 则会使用 R1 的值，可是 R1 必须在第一条语句中的第 5 个 cycle 才能更新到寄存器中，语句二是在第 4 个 cycle 就要访问 R1，也就是说第二,条指令此时在使用错误的 R1 的值。这是数据 hazard 出现了。

解决方案：在两条指令中添加一条空指令：nop。但是会影响处理器的指令的执行效率。在现代处理器技术中，已经用 forwarding 的方式解决了。如下图，如果处理器在检测到当前指令的源操作数正好在流水线的 EX 或者 MEM 阶段，接直接将 ex 和 mem 寄存器的值传

递给 ALU 的输入，而不是再从寄存器堆中获取数据了。因为此时寄存器堆中的数据可能是没有被及时更新的。

3、控制冒险

在流水线中的的执行指令时，由于并行处理的关系，后面很多指令其实都在流水线中开始处理了，包括预取值和译码。那么，如果此时程序中出现一条跳转语句怎么办呢？因为程序已经跑到其他地址处执行，流水线中之前已经做好的预取值和译码动作都不能使用了。这些会被处理器的专有部件 flush 掉，重新开始新的流水线。此时我们可以称之为出现了控制 hazard。这种情况对于程序和效率来说是存在很大损失的。

解决方案：也就是在 jump 指令后面(不会被真正使用，但是会进入流水线)添加 nop。在 MIPS 程序中，经常在 jump 指令后面添加 nop 语句。

在 X86 架构中，是通过硬件来实现 flush，将无效的流水线排空，以保证正确运行流水线。这里会涉及到分支预测技术的使用。

在其他一些处理器中，用软件的方式来处理，添加 nop。同时在编译器中通过乱序的思想用有效指令代替 nop。这样也可以避免转跳带来的性能损失。

5、流水线设计中的其他问题

答： 1、每个阶段所用的硬件实际并不是相互独立的；增加的寄存器也会导致延迟增大；每阶段的周期划分也很难做到一致。

2、理想的流水线系统，每个阶段的时间都是相等的。实际上，各个阶段的时间是不等的。运行时钟是由最慢的阶段决定的。

3、另外流水线过深，寄存器的增加会造成延迟增大。当延迟增大到时钟周期的一定比例后，也会成为流水线吞吐量的一个制约因素。

二、优化程序性能 （感谢张寒张寒）

19. **优化程序性能**

答：

1、 程序优化的第一步就是消除不必要的内容，让代码尽可能有效地执行它期望的工作。这包括消除不必要的函数调用、条件测试和存储器引用。这些优化不依赖于目标机器的任何具体属性。

2、程序优化的第二步，利用处理器提供的指令级并行能力，同时执行多条指令。

3、最后对大型程序的优化，使用代码剖析程序，代码剖析程序是测量程序各个部分性能的工具这种分析能够帮助找到代码中低效率的地方，并且确定程序中应该着重优化的部分

4、Amdahl 定律，它量化了对系统某个部分进行优化所带来的整体效果

20. **优化编译器的能力和局限性以及表示程序性能、**

答：csapp 第五章 P325

优化编译器的能力：

现代编译器运用复杂精细的算法来确定一个程序中计算的是什么值，以及它们是被如何使用的。然后它们会利用一些机会来简化表达式，在几个不同的地方使用同一个计算，以及降低一个给定的计算必须被执行的次数。

优化编译器的局限性：

编译器必须很小心地对程序只是用安全的优化，也就是说对于程序可能遇到的所有可能的情况，在 C 语言标准提供的保证之下，优化后得到的程序和未优化的版本有一样的行为，限制编译器只进行安全的优化，消除了一些造成不希望的运行时行为的可能原因，但是这也意味着程序员必须花费更大的力气写出程序使编译器能够将之转换成有效机器代码，

两个指针可能指向同一个存储器位置的情况称为存储器别名使用（memory aliasing）。

这造成了一个主要的妨碍优化的因素，这也是可能严重限制编译器产生优化代码机会的程序的一个方面：如果编译器不能确定两个指针是否指向同一个位置，就必须假设什么情况都有可能，限制了可能的优化策略。

表示程序性能：

引入度量标准每元素的周期数（Cycles Per Element　CPE）,作为一种表示程序性能并指导我们改进代码的方法

使用最小二乘方拟合，得到一条形如 y=mx+b 的线，线性因子的系数 m 叫做每元素的周期数 CPE 的有效数

21. **特定体系结构或应用特性的性能优化、**

答：　1、简单地使用命令行选项如‘-O1’就会进行一些基本的优化。

2、消除循环的低效率：称作‘代码移动’，这类优化包括识别要执行多次（例如在循环里）但是计算结果不会改变的计算，因而将计算移动到代码前面不会被多次求值的部分

3、减少过程调用：修改代码减少函数的调用，不过会损害一些程序的模块性

4、循环展开：通过增加每次迭代计算的元素的数量，减少循环的迭代次数。从两个方面改程序的性能，首先它减少了不直接有助于程序结果的操作的数量，例如循环索引计算和条件分支。其次，它提供了一些方法，可以进一步变化代码，减少整个计算中关键路径上的操作数量

5、提高并行性：1.多个累积变量 2.重新结合变换

22. **限制因素**

答：

1、寄存器溢出：循环并行性的好处受到描述计算的汇编代码的能力限制。

特别地，IA32 指令集只有很少量的寄存器来存放积累的值（IA32 只有 4 个，x86-64 可以 12 个）。如果我们的并行度 p 超过了可用的寄存器数量，那么编译器会诉诸溢出（spilling），将某些临时值存放到栈中。一旦出现这种情况，性能会急剧下降。

23. 分支预测和预测错误处罚：当分支预测逻辑不能正确预测一个分支是否要跳转的时候，条 件分支可能会招致严重的预测错误处罚。

24. 对于这个问题没有简单的答案，有一些通用原则

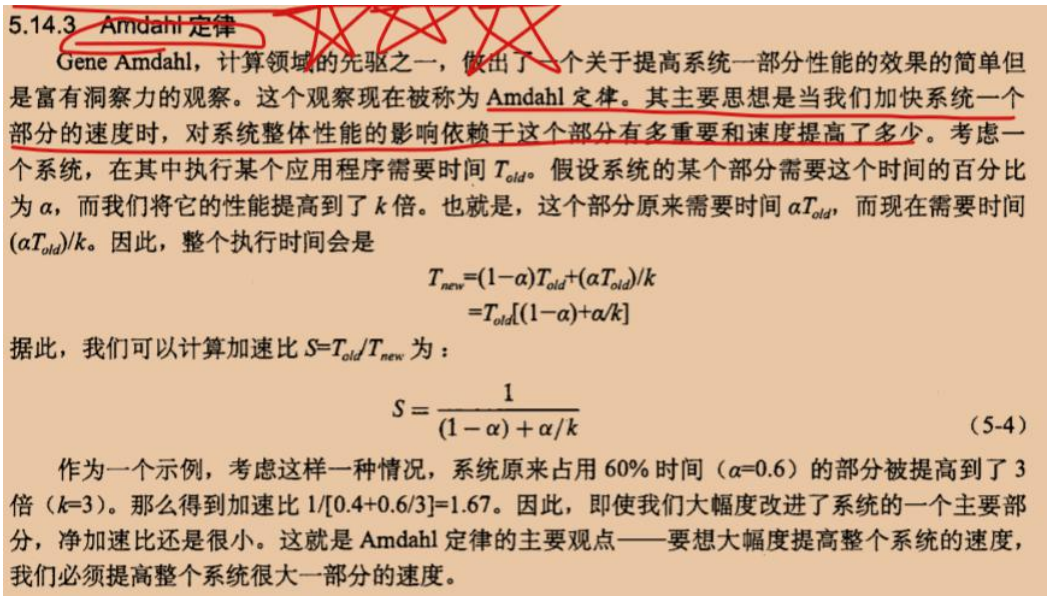
1.不要过分关心可预测的分支

2.书写适合用条件传送实现的代码

25. **确认和消除性能瓶颈**

答：处理大程序时连知道该优化什么地方都很困难。

- 程序剖析：程序剖析包括运行程序的一个版本，其中插入了工具代码，以确定程序的各个部分需要多少时间，这对于确认需要集中注意力优化的部分很有用，剖析的一个有力指出在于可以在现实的基准数据上运行实际程序的同时，进行剖析（Unix 系统提供了一个剖析程序 GPROF）。通常，假设在有代表性的数据上运行程序，剖析能帮助我们对典型的情况进行优化，但是我们还应该确保对所有可能的情况，程序都有相当的性能。这主要包括避免得到糟糕的渐近性能的算法（例如插入算法）和坏的编程实践
- Amdahl 定律：其主要思想是当我加快系统一个部分的速度时，对系统整体性能的影响依赖于这个部分有多重要和速度提高了多少。



5.14.3 Amdahl 定律

Gene Amdahl，计算领域的先驱之一，做出了一个关于提高系统一部分性能的效果的简单但是富有洞察力的观察。这个观察现在被称为 Amdahl 定律。其主要思想是当我们加快系统一个部分的速度时，对系统整体性能的影响依赖于这个部分有多重要和速度提高了多少。考虑一个系统，在其中执行某个应用程序需要时间 T_{old}。假设系统的某个部分需要这个时间的百分比为 α，而我们将它的性能提高到了 k 倍。也就是，这个部分原来需要时间 αT_{old}，而现在需要时间 $(\alpha T_{old})/k$。因此，整个执行时间会是

$$T_{new}=(1-\alpha)T_{old}+(\alpha T_{old})/k$$
$$=T_{old}[(1-\alpha)+\alpha/k]$$

据此，我们可以计算加速比 $S=T_{old}/T_{new}$ 为：

$$S=\frac{1}{(1-\alpha)+\alpha/k} \quad (5\text{-}4)$$

作为一个示例，考虑这样一种情况，系统原来占用 60% 时间（α=0.6）的部分被提高到了 3 倍（k=3）。那么得到加速比 1/[0.4+0.6/3]=1.67。因此，即使我们大幅度改进了系统的一个主要部分，净加速比还是很小。这就是 Amdahl 定律的主要观点——要想大幅度提高整个系统的速度，我们必须提高整个系统很大一部分的速度。

三、存储器结构及虚拟存储器 （感谢 gary）

1、局部性 9

答：

一个编写良好的计算机程序常具有良好的局部性。

引用邻近于其他最近引用过的数据项的数据项，或者最近引用过的数据项本身。这种倾向性，被称为局部性原理。

局部性两种形式：

时间局部性 被引用过一次的内存位置很可能在不远的将来再被多次引用。(通常在循环中)

空间局部性 一个内存位置被引用了一次，那么将来他附近的位置也会被引用。

局部性与性能的关系

有良好局部性的程序比局部性差的程序运行得更快。

1 局部性原理允许计算机设计者通过引入称为高速缓存存储器来保存最近被引用的指令和数据项，从而提高对主存的访问速度。

3 重复引用相同变量的程序有良好的时间局部性。

4 对于具有步长为 k 的引用模式的程序，步长越小，空间局部性越好。

5 具有步长为 l 的引用模式的程序有很好的空间局部性。

6 在内存中以大步长跳来跳去的程序空间局部性会很差。

7 对于取指令来说，循环有好的时间和空间局部性。

8 循环体越小，循环迭代次数越多，局部性越好。

2、存储器层级结构

答：

现在随着处理器和存储器在性能发展上的差异越来越大，存储器在容量尤其是访问延时方面的性能增长越来越跟不上处理器性能发展的需要。为了缩小存储器和处理器两者之间在性能方面的差距，通常在计算机内部采用层次化的存储器体系结构。

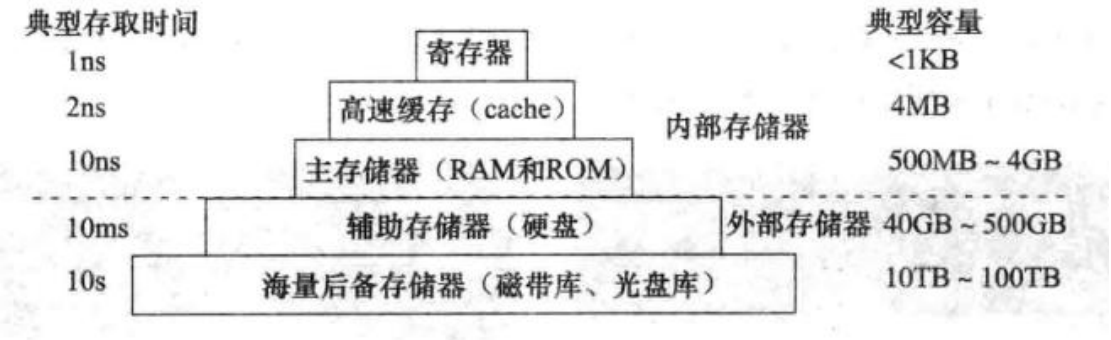


图 6.3　存储器层次化体系结构示意图

从图中可以看出，速度越快则容量越小、越靠近 CPU.CPU 可以直接访问内部存储器，而外部存储器的信息则要先取到主存，然后才能被 CPU 访问.CPU 执行指令时，需要的操作数大部分都来自寄存器；当需要从（向）存储器中取（存）数据时，先访问 cache，如果不在 cache 中，则访问主存，如果不在主存中，则访问硬盘，此时，操作数从硬盘中读出送到主存，然后从主存送到 cache 数据使用时一般只在相邻两层之间复制传送，而且总是从慢速存储器复制到快速存储器。

传送的单位是一个定长块，因此需要确定定长块的大小，并在相邻两层间建立块之间的映射关系。

3、计算机高速缓存器原理

答：

加快 CPU 访存速度的主要方式之一是在 CPU 和主存之间增加高速缓冲存储器（简称高速缓存或 cache）。

cache 是一种小容量高速缓冲存储器，由快速的 SRAM 组成，直接制作在 CPU 芯片内，速度较快，几乎与 CPU 处于同一个量级。

在 CPU 和主存之间设置 cache，总是把主存中被频繁访问的活跃程序块和数据块复制到 cache 中。由于程序访问的局部性，大多数情况下，CPU 能直接从 cache 中取得指令和数据，而不必访问慢速的主存。
为便于 cache 和主存间交换信息，cache 和主存空间都被划分为相等的区域。例如，将主存按照每 512 字节划分成一个区域，同时把 cache 也划分成同样大小的区域，这样主存中的信息就可按照 512 字节为单位送到 cache 中。
我们把主存中的区域称为块，也称为主存块，它是 cache 和主存之间的信息交换单位；cache 中存放一个主存块的区域称为行或槽，也称 cache 行。

4、高速缓存对性能的影响

答：影响 cache 性能的因素决定系统访存性能的重要因素之一是 cache 命中率，它与许多因素有关。
1、命中率与关联度有关。关联度越高，命中率越高。关联度反映一个主存块对应的 cache 行的个数，显然，直接映射的关联度为 1；2 路组相联映射的关联度为 2，4 路组相联映射的关联度为 4，全相联映射的关联度为 cache 行数。
2、命中率与 cache 容量有关。cache 容量越大，命中率就越高。
3、命中率与主存块的大小有关。采用大的交换单位能很好地利用空间局部性，但是，较大的主存块需要花费较多的时间来存取，因此，缺失损失会变大。由此可见，主存块的大小必须适中，不能太大，也不能太小。
此外，设计 cache 时还要考虑以下因素：
采用单级还是多级 cache、数据 cache 和指令 cache 是分开还是合在一起、主存一总线一 cache--CPI 之间采用什么架构等甚至主存 DRAM 芯片的内部结构、存储器总线的总线事务类型等，也都与 cache 设计有关，都会影响系统总体性能。
下面对这些问题进行简单分析说明。
目前 cache 基本上都在 CPU 芯片内，且使用 LI 和 I2 cache，甚至有 I3 cache,CPU 的访问顺序为 L1 cache、L2cache 和 L3 cache。
通常 L1 cache 采用分离 cache，即数据 cache 和指令 cache 分开设置，分别存放数据和指令。
指令 cache 有时称为代码 cache（code cache）.L2 cache 和 L3 cache 为联合 cache，即数据和指令放在一个 cache 中。
由于多级 cache 中各级 cache 所处的位置不同，使得对它们的设计目标有所不同。例如假定是两级 cache。那么，对于 L1 cache，通常更关注速度而不要求有很高的命中率，因为，即使不命中，还可以到 L2 cache 中访问，L2 cache 的速度比主存速度快得多；而对于 L2 cache，则要求尽量提高其命中率，因为若不命中，则必须到慢速的主存中访问，其缺失损失会很大而影响总体性能

5、地址空间

答：

每个高级语言源程序经编译、汇编、链接等处理生成可执行的二进制机器目标代码时，都被映射到同样的虚拟地址空间，因此，所有进程的虚拟地址空间是一致的，这简化了链接器的设计和实现，也简化了程序的加载过程。

虚拟存储机制为程序提供了一个极大的虚拟地址空间（也称为逻辑地址空间），它是主存和磁盘存储器的抽象。虚存机制带来了一个假象，使得每个进程好像都独占使用主存，并且主存空间极大。

这有三个好处：

- 每个进程具有一致的虚拟地址空间，从而可以简化存储管理
- 它把主存看成是磁盘存储器的一个缓存，在主存中仅保存当前活动的程序段和数据区，并根据需要在磁盘和主存之间进行信息交换，使有限的主存空间得到了有效利用；
- 每个进程的虚拟地址空间是私有的，因此，可以保护各自进程不被其他进程破坏。

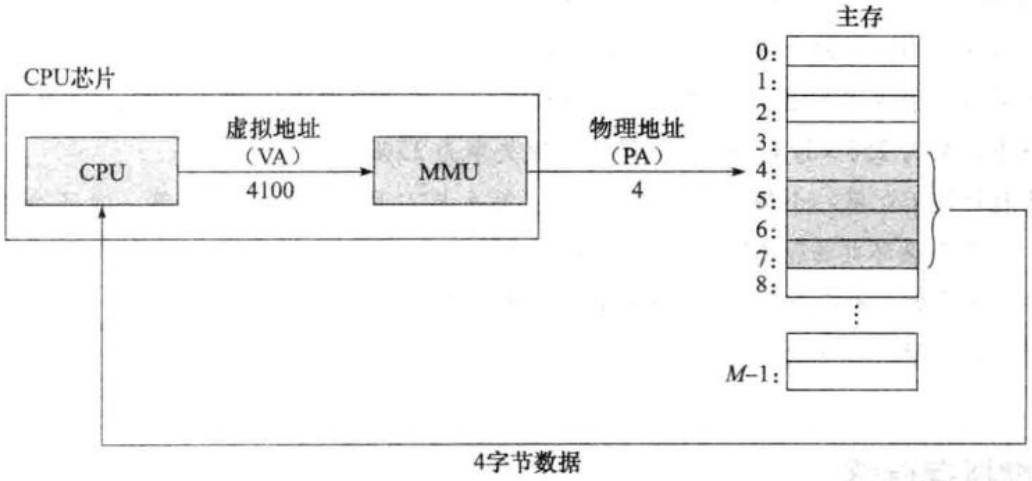
6、虚拟存储器

答：

一个系统中的进程是与其他进程共享 CPU 和主存资源的，然而，共享主存会形成一些特殊的情况，如果太多的进程需要太多的储存器，那么他们中的一些就根本无法运行。当一个程序没有空间可用的时候，那就是他运气不好。存储器还容易被迫害，如果一个进程不小心写了另一个进程使用的存储器，它就可能失去原先的逻辑。为了更有效的管理存储器，现在系统引入了一种对主存的抽象概念，叫做虚拟存储器。

目前，在服务器、台式机和笔记本等各类通用计算机系统中都采用虚拟存储技术。在采用虚拟存储技术的计算机中，指令执行时，通过存储器管理部件（Memory Management Unit，简称 MMU）将指令中的逻辑地址（也称虚拟地址或虚地址，简写为ＶA）转化为主存的物理地址（也称主存地址或实地址，简写为 PA）。在地址转换过程中由硬件检查是否发生了访问信息不在主存或地址越界或访问越权等情况。若发现信息不在主存，则由操作系统将数据从磁盘读到主存。若发生地址越界或访问越权，则由操作系统进行相应的异常处理。由此可以看出，虚拟存储技术既解决了编程空间受限的问题，又解决了多道程序共享主存带来的安全等问题。

图 6.29 是具有虚拟存储机制的 CPU 与主存的连接示意图，从图中可知，CPU 执行指令时所给出的是指令或操作数的虚拟地址，需要通过 MMU 将虚拟地址转换为主存的物理地址才能访问主存，MMU 包含在 CPU 芯片中。图中显示 MMU 将一个虚拟地址转换为物理地址 4，从而将第 4、5、6、7 这 4 个单元的数据组成 4 字节数据送到 CPU。图 6.29 仅是一个简单示意图，其中没有考虑 cache 等情况。



7、虚拟内存的管理

答：

- 请求分页存储管理

每次访问仅将当前需要的页面调入主存，而进程中其他不活跃的页面放在磁盘上。当访问某个信息所在页不在主存时发生缺页异常，此时，硬件将调出 OS 内核中的缺页处理程序，将缺失页面从磁盘调入主存。

与主存块大小相比，虚拟页的大小要大得多。因为 DRAM 比 SRAM 大约慢 10~100 倍，而磁盘比 DRAM 大约慢 100000 多倍，所以进行缺页处理所花的代价要比 cache 缺失损失大得多。

而且，根据磁盘的特性，磁盘扇区定位所用的时间要比磁盘读写一个数据的时间长大约 100000 倍，也即对扇区第一个数据的读写比随后数据的读写要慢 100000 倍。考虑到缺页代价的巨大和磁盘访问第一个数据的开销，通常将主存和磁盘之间交换的页的大小设定得比较大，典型的有 4KB 和 8KB 等，而且有越来越大的趋势。

因为缺页处理代价较大，所以提高命中率是关键，因此，在主存页框和虚拟页之间采用全相联映射方式。此外，当进行写操作时，由于磁盘访问速度很慢，所以，不能每次写操作都同时写 DRAM 和磁盘，因而，在处理一致性问题时，采用回

写方式，而不用全写方式。

- 请求分段存储管理

根据程序的模块化性质，可按程序的逻辑结构划分成多个相对独立的部分，例如，过程、数据表、数据阵列等。这些相对独立的部分被称为段，它们作为独立的逻辑单位可以被其他程序段调用，形成段间连接，从而产生规模较大的程序。段通常有段名、段起点、段长等。段名可用用户名、数据结构名或段号标识，以便于程序的编写、编译器的优化和操作系统的调度管理等。

可以把段作为基本信息单位在主存一辅存之间传送和定位。分段方式下，将主存空间按实际程序中的段来划分，每个段在主存中的位置记录在段表中，段的长度可变，所以段表中需有长度指示，即段长。每个进程有一个段表，每个段在段表中有一个段表项，用来指明对应段在主存中的位置、段长、访问权限、使用和装入情况等。段表本身也是一个可再定位段，可以存在外存中，需要时调入主存，但一般驻留在主存中。

在分段式虚拟存储器中，虚拟地址由段号和段内地址组成。通过段表把虚拟地址转换成主存物理地址，分段式管理系统的优点是段的分界与程序的自然分界相对应；段的逻辑独立性使它易于编译、管理、修改和保护，也便于多道程序共享；某些类型的段（如堆、栈、队列等）具有动态可变长度，允许自由调度以便有效利用主存空间。但是，由于段的长度各不相同，段的起点和终点不定，给主存空间分配带来麻烦，而且容易在主存中留下许多空白的零碎空间，造成浪费。

分段式和分页式存储管理各有优缺点，因此可采用两者相结合的段页式存储管理方式。

- 请求段页式存储管理

在段页式虚拟存储器中，程序按模块分段，段内再分页，用段表和页表（每段一个页表）进行两级定位管理。段表中每个表项对应一个段，每个段表项中包含一个指向该段页表起始位置的指针，以及该段其他的控制和存储保护信息，由页表指明该段各页在主存中的位置以及是否装入、修改等状态信息。

程序的调入调出按页进行，但它又可以按段实现共享和保护。因此，它兼有分页式和分段式存储管理的优点。它的缺点是在地址映象过程中需要多次查表。

8、翻译和映射

答：

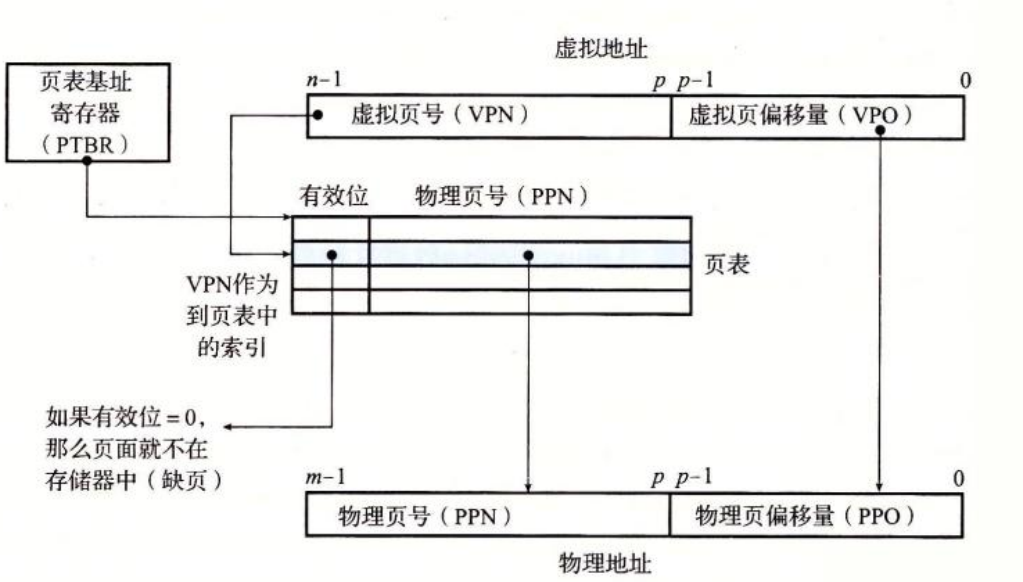


图 9-12　使用页表的地址翻译

9、TLB

答：

地址转换过程中，访存时首先要到主存查页表，然后才能根据转换得到的物理地址再访问主存以存取指令或数据。如果缺页，则还要进行页面替换、页表修改等，访问主的次数就更多。因此，采用虚拟存储机制后，使得访存次数增加了。为了减少访存次数，往往把页表中最活跃的几个页表项复制到高速缓存中，这种在高速缓存中的页表项组成的页表称为后备转换缓冲器（Translation Lookaside Buffer，简称 TLB），通常称为快表，相应地称主存中的页表为慢表。

10、动态存储器分配和垃圾收集

答：

我们可以通过 mmap 和 munmap 来创建和删除虚拟存储器区域，但对开发来说使用起来并不方便，况且没有很好的移植性，所以提出了使用动态存储分配器来管理进程空间中的堆区域。

动态存储分配器维护着一个进程的虚拟存储器区域，称为堆。堆是从低位地址向高位向上增长的，对于每个进程，内核维护着一个 brk，它指向堆的顶部。

分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟存储器片，要么是已分配的，要么是空闲的。已分配的显示地保留为供应应用程序使用。空闲块可用来分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显示执行的，要么是存储分配器隐式执行的，它们的都是

显式的来分配存储块的，不同之处在于由哪个实体来负责释放已分配的块。

a) 显式分配器，要求显式的释放已分配的块。如 C 标准库中的 malloc 和 free，C++中的 new 和 delete 操作符。

b) 隐式分配器，要求分配器检测一个已分配的块何时不再被程序使用，那么就释放这个块。隐式分配器也叫做垃圾收集器(Grabage collector)。如 java 语言就依赖于类似分配器。

下面我们看下 malloc 和 free 的实现是如何管理一个 C 程序的 16 字的小堆的。每个方框代表一个 4 字节的字。粗线标出的矩形对应于已分配块(有阴影)和空闲块(无阴影)，初始时，堆是由一个大小为 16 个字的、双字对齐的、空闲块组成的。

a) 程序请求一个 4 字的块，malloc 的响应是：从空闲块的前部切出一个 4 字的块，并返回一个指向这个块的第一个字的指针 p1

b) 程序请求一个 5 字的块，malloc 的响应是：从空闲块的前部分配一个 6 字的块，返回指针 p2,填充的一个额外字是为了保持空闲块是双字边界对齐的。

c) 程序请求一个 6 字的块，而 malloc 就从空闲块的前部切出一个 6 字的块。返回指针 p3

d) 程序释放在 b 中分配的那个 6 字的块。需要注意的是，在调用 free 返回之后，指针 p2 仍然指向被释放的块，在它被一个新的 malloc 调用重新初始化之前不能在程序中再使用 p2.

e) 程序请求一个 2 字的块。在这种情况下，malloc 分配在前一步中释放了的块的一部分，并返回指向新块的指针 p4.

垃圾收集器是一种动态存储器分配器，它自动释放程序不再需要的已分配块。这些块称为垃圾。自动回收堆存储的过程叫做垃圾收集。在 java 虚拟机中就使用了类似的机制，应用显式分配堆块，但是从不显式地释放他们。垃圾收集器定期识别垃圾块，并相应地调用 free，将这些块放回到空闲链表中。

垃圾收集器将存储器视为一张有向可达图，该图的节点被分成一组根节点和一组堆节点。每个堆节点对应于堆中的一个已分配块。有向边 p->q 意味着块 p 中的某个位置指向块 q 中的某个位置。根节点对应于这样一种不在堆中的位置，他们中包含指向堆中指针。这些位置可以是寄存器、栈里的变量，或者是虚拟存储器中读写数据区域内的全局变量。

当存在一条从任意根节点出发并到达 p 的有向路径时，我们说节点 p 是可达的。在任何时刻，不可达节点对应于垃圾，是不能被应用再次使用的。垃圾收集器的

角色就是维护可达图的某种表示，并通过释放不可达节点将它们返回给空闲链表，来定期地回收它们。

四、链接、进程及并发编程 （感谢小春）

26. **静态链接**

答：

01 将可重定位的文件和命令行变成完全链接的、可加载、可运行的目标文件；

02 可重定位目标文件由各代码和数据节组成；

完成静态链接，链接器要完成以下两个工作：

1 符号解析，将每一个符号引用正好和一个符号定义关联起来；

2 重定位：可重定位的目标文件地址都是从零开始的，连接器通过吧每个符号定义与一个内存位置关联起来，从而重定位这些节，然后修改所有对这些符号的引用，使得他们指向这个内存位置。

27. **目标文件**

答：

分类 可重定位目标文件；可执行目标文件；共享目标文件；

可重定位目标文件：包含二进制代码和数据，其形式可以在编译时与其他可重定位目标文件合并起来，创建一个可执行目标文件。

可执行目标文件：包含二进制代码和数据，其形式可以被直接复制到内存并执行。

共享目标文件：一种特殊类型的可重定位目标文件，可以在加载或者运行时被动态的加载进内存并链接。

28. **符号和符号表**

答：

定义：符号表记录了目标模块定义的符号和引用的符号信息。三种符号类型：

1.由模块 m 定义并能被其他模块引用的全局符号，非静态的 C 函数和全局变量；

2.由其他模块定义并被模块 m 引用的全局变量；

3.由目标模块定义和引用的局部符号，表现为静态全局变量和函数；

符号解析：链接符号引用与符号定义

a.全局符号的多重定义问题

强符号：已被初始化的全局变量和函数

弱符号：未被初始化的全局变量

b.规则

1:不允许有多个同名的强符号

2:如果有一个强符号和多个弱符号同名，选强符号

3:如果有多个弱符号同名，从其中任选一个

c.符号的地址由链接器确定，但符号的大小以及其类型在编译器就已经确定了，，链接器只负责

解析和重定位符号确认符号的地址，同名符号有且仅有一个地址

d.静态链接可选方式：

1.一组可重定位目标文件

2.所有相关的目标模块打包成一个单独文件---静态库（存档文件）

e.使用静态库来解析符号

过程：符号解析时，链接器从左到右按照他们在编译器驱动程序命令行上出现的顺序来烧描可

重定位目标文件和存档文件（如.c--.o）。链接器维护三个集合：所有目标模块集合 E；未定义的

集合 U；定义的集合 D；

烧描开始，链接器来判断输入 f 是什么，若是目标文件，f 添加到 E，修改 U/D 来反应 f 中的符号定

义和引用；若 f 是存档文件，链接器就尝试匹配 U 中未解析的符号和由存档文件成员定义的符号，

若存档文件成员 m 中有已定义的符号，m 放 E，修改 U/D；如果链接器完成扫描后，U 非空，则链

接器输出错误并终止，否则，它会合并和重定位 E 中的目标文件；

29. **重定位和加载**

答：

01. 重定位 （可结合静态链接的两个步骤作答）

需要的术语：

30. 重定位节和符号定义：聚合所有目标文件的相同节，链接器开始将运行时内存地址赋给每一个节和每一个符号 ☆☆☆

31. 重定位节中的符号引用，链接器修改代码节和数据节中的每个符号引用，使得他们执行正确的运行时地址。

02.过程：

32. 一个概念，当汇编器遇到未知的符号引用，就会生成一个重定位条目，代码的重定位条目存放在 .rel.text 中，已初始化数据的重定位条目放在.rel.data 中

重定位结构的结构：

```
typedef struct{
    long offset; //符号相对节的偏移
    long type:32;
    symbol:32;
    long addend;
}Elf64_Rela;
```

b.符号引用的重定位

只讲两种基本的重定位类型：

1.相对引用

2.绝对引用

03.加载 ☆☆☆

1.背景：elf 可执行文件的格式跟可重定位的格式是很相似的

2.可执行目标文件的加载

加载器将目标文件的代码和数据复制到内存中，然后跳转到入口点来运行即 _start 函数的地址；

33. **动态链接库**

答：

34. 出现的原因：为了解决静态库维护还是相对麻烦以及很多共用的库在内存中有很多碎片造成的内存浪费。

35. 动态链接：共享库在加载或运行时加载到任意内存位置，并和在内存中的程序链接起来，由 动态链接器完成。

36. 相关细节：

动态链接不会复制共享库的代码和数据段，仅会复制一些重定位信息和符号表；信息和符号表

共享库中会有一个 .interp 节，这个节包含动态链接器的路径名，动态链接器本身就是一个共享目标如（ld-linux.so）。

当运行一个可执行文件时，加载器会通过.interp 节的信息找到动态链接器来运行，而动态链接器重定位相关的.so 来完成链接任务：

位置无关代码 PIC（position independent code）：共享库若想要被进程共享就要求使用位置
无关代码，位置无关代码是可以加载到内存的任何位置，而无需链接器修改即可以加载和重定
位；

37. **异常和进程**

答：

异常就是控制流突变，用来响应处理器状态中的某个变化，一部分由硬件实现，一部分
由操作系统来实现，每个异常都会被分配一个异常号，一些由硬件设计者来分配，常
见的有：内存缺页，算术溢出，内存访问违规，除零，一些由内核设计者分配，常用的
有系统调用和外部 I/o 设备的信号，异常有如下分类：中断，故障，陷阱，终止。
其中中断是异步发生的，中断函数处理结束后返回下条指令，陷阱是同步的总是返回
下条指令，故障时同步的，由潜在可恢复的错误造成（缺页），返回当前的指令，终止
同步的，不可恢复，不会返回。

a.异常处理写过程调用的一些不同之处

01 返回地址不一样，异常返回地址需要根据异常类型来确定
02 异常处理时，处理器将一些额外的处理状态压入栈中
03 若是由用户态进入内核态，则异常处理便用内核栈
04 异常处理程序运行在内核态

b.四种不同类型

中断：不是由一条专门的指令造成的，它是处理器在每次执行一条指令后检测是否有中断产生：
陷阱和系统调用：陷阱是有意义的异常，像中断处理程序一样，陷阱处理程序将返回下一条指
令。陷阱最重要的作用是
在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。系统调用都有一个服务号，
指令“syscall n”来请求服务 n；

中断与系统调用的区别：系统调用采用陷阱门，中断进入中断服务 cpu 自动关闭中断 IF 清 0，防止
嵌套；陷阱门进入服务程序时
IF 不变，是开中断下进行的，所有系统调用可被中断；
系统调用与一般程序区别：调用的栈不同，一个运行在内核模式一个运行在用户模式；
故障：若故障能够被恢复，则返回到引起故障的指令，否则终止；
终止：不可恢复的错误。

进程：
a.定义：一个执行中的程序的实例。
进程与程序的区别：程序是一堆代码可以作为目标文件存在于磁盘上，或者作为段存在于地址空
间中，进程是执行程序的一个具体实例，程序总是运行在某个进程的上下文中。
b.进程提供应用程序的关键抽象：
一个独立的逻辑控制流，它提供一个假象，好像我们的程序独占地使用处理器。
一个私有的地址空间，它提供一个假象，好像我们的程序独占地使用内存系统。
c.逻辑控制流
PC 值的序列-逻辑流。
d.用户模式下访问内核
唯一的模式就是通过异常（中断、故障、陷阱）
处理器通常是用某个控制寄存器中的一个模式位，该寄存器描述了当前进程享有的特权。当设置
了模式位时，进程就运行。
在内核模式中（也叫超级用户模式）一个运行在内核模式的进程可以执行指令集中的任何指令，
并可以访问系统中的。
任何内存位置。没有设置，就是普通用户模式，不允许执行特权指令，比如停止处理器等。
e.上下文切换过程
每个进程维持一个上下文，上下文是内核重新启动一个被抢占的进程所需的状态。
过程：1 保存当前进程的上下文，2 恢复某个先前被抢占的进程被保存的上下文，3 将控制传递给
这个新恢复的进程。

38. **进程控制和信号**

答：

1.进程控制

a.获取进程号（进程有唯一的进程号 正数 ID/PID）

getpid；

b.创建进程和终止进程

程序员角度分三种

1 运行：进程在 cpu 运行或等待运行且最终被内核调度。

2 停止：进程的执行被挂起，且无触发条件不会被调度。

3 终止：永远停止了，如收到一个信号终止了进程，或调用 exit；

4 创建：fork、vfork；

fork：调用一次返回两次，和父进程用相同的地址空间，与父进程并发执行，与父共享文件，pid 不一样；

c.回收子进程

一个终止的进程若没有被父进程回收，会变成僵死进程，仍然会占用内存空间，直到被回收

若其父进程先终止，则另 init 进程收养、回收。init 进程 pid 为 1，是所有进程的祖先。

d.进程的休眠

sleep pause；

2.信号

a.一次软件形式的异常，一个信号就是一条消息，他通知进程系统中发生了一个某种类型

的事件。

底层的硬件异常是由内核异常处理程序处理，一般情况对用户而言是不可见的，但信号提供

了一种机制告诉用户进程系统发生了什么样的异常。

b.发生信号的原理：内核改变目的进程的上下文中某个状态来传递一个信号给目的进程；

（收发信号可简理解为：上下文中存与取）

接收信号后常见的信号处理行为有以下几种：

1 进程终止：SIGKILL

2 进程终止并转储内存
3 进程停止（挂起）知道被 SIGXONT 信号重启
4 进程忽略该信号
c.进程组的概念，每个进程都属于一个进程组，且父子进程同属于一个进程组。
d.阻塞和解除阻塞
隐式阻塞：阻塞任何与当前正在处理的信号类型的待处理的信号，只是等不用条件触发。
显式阻塞：明确的阻塞和解除阻塞选定的信号。
e.非本地跳转
一种用户级异常控制流，他可以将控制直接从一个函数转移到另一个当前正在执行的函数而不用
通过函数栈机制

39. **进程间的通信**

答：
指在不同的进程之间传播和交换信息
常见的通信手段：
40. 管道及有名管道 ：管道用于有亲缘关系进程间的通信，有名管道克服了管道没有名字的限
制，因此，有名管道可用于无亲缘关系进程间通信。
41. 信号：用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发信号给进
程本身。
42. 消息队列：是消息的链表，存放在内核中，并由消息队列标志符标示。克服了信号传输信息
少，管道只能承载无格式字节及缓存区大小受限等问题；
4 信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问；
5 共享内存：多个进程可以访问同一块内存空间，是最快的 ipc 形式，一般结合信号量使用；
6 套接字：更一般的进程间通信机制，可以完成本机或者跨机器的进程通信；

43. **进程间信号量的控制**

答：
信号量是一个计数器，是一个具有非负整值的全局变量，能做如下两个操作：

P（s）：如果 s 为非零，那么 P 将 s 减 1，并且立即返回。如果 s 为零，那么就挂起这个线程，直到
s 变为非零，而一个 V 操作会重启这个线程。重启后 P 操作将 s 减 1，并将控制返回给调用者。
V（s）：V 操作将 s 加 1，如果有任何线程阻塞在 P 操作等待 s 变成非零，那么 V 操作会重启这些
线程中的一个，然后该线程将 s 减 1，完成它的 P 操作。

44. **信号量**

答：

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行 P 和 V 操作，最简单
的信号量是只能取 0 和 1 的变量，这也是信号量最常用的一种形式，叫做二进制信号量。而可以取
多个常亮的信号量被称为通用信号量；

45. **各种并发编程模式**

答：

概念：如果逻辑控制流在时间上有重叠，那它们就是并发的；

场景：

1 访问慢速 I/O 设备

2 与人交互

3 推迟工作以降低延迟

4 服务多个网络客户端

5 在多核机器上并行计算

三种并发编程模式

1 基于进程的并发编程

每个逻辑控制流都是一个进程，由内核来调度和维护

优缺点：

优点：共享文件表，不共享用户空间，更加的安全。

缺点：开销大，共享信息困难，要使用显式的 IPC

2 基于 I/O 多路复用的并发编程

基本的思想就是使用 select 函数，要求挂起进程，只有在一个或多个 I/O 事件发生后，

才将控制返回给应用程序；
I/O 多路复用可以用做并发事件驱动程序的基础。
优缺点：
优点：程序员能更好的控制程序，共享数据简单
缺点：编码复杂，并且程度越小，复杂度越高；
3 基于线程并发模型
概念：线程就是运行在进程上下文中的逻辑流，线程也由内核调度，每个线程都有自己的线程上
下文，包括线程 ID、栈、栈指针、程序计数器、条件吗和通用目的寄存器。
线程的切换比进程快很多，线程是对等的，任何线程都可以访问共享虚拟内存的任何位置。
a.分离线程
线程要么是可结合的，要么是可分离的，区别：
1 可结合线程：可以被其他线程回收和杀死，被回收之前，它所占有的资源不会被释放；
2 可分离线程：不可以回收不可以杀死，它的内存资源在其终止时自动释放，默认创建的都是可
结合线程；

46. **共享变量和线程同步**
答：
1.线程内存模型：
每个线程都有自己独立的线程上下文、包括线程 ID、栈栈指针、程序计数器、条形码和通用目的
寄存器。线程与其他线程共享进程上下文的剩余部分。包括整个用户虚拟地址空间。
2.将变量映射到内存
全局变量：定义在函数之外的变量，运行时，全局变量只有一个实例，任何线程都可以引用。
本地自动变量：定义在函数内，但没有用 static 属性的变量，运行时每个线程的栈都包含它自己
的所有本地自动变量实例。
本地静态变量：函数内部用 static 属性的变量，运行时只有一个实例。
3.共享变量

我们说一个变量 v 是共享的，当且仅当它的一个实例被一个以上的线程引用。如果只有一个线程
引用就不是共享的。共享变量是简单的，但这种共享方式会引来同步错误，因为线程之间的运行
是竞争关系，没办法确认进程运行的顺序，为了解决这种错误，可以使用信号量来对共享资源加
锁，使其确定线程对该变量的互斥访问。

4.线程同步

确保每个线程在执行它的临界区中的指令时，拥有对共享变量的互斥访问。

通过对二元信号量的使用（二元信号量也称为互斥锁），对共享资源进行加锁，使得每次只会有
一个线程能够访问，直到访问完成，其他线程才能访问，这样就能对资源的互斥旺旺，达到线程
同步。

47. **其他并行问题**

答：

1.线程安全：如果一个函数被称为线程安全的，那当且仅当被多个并发线程反复地调用时，它会一直产生正确的结果。

四种线程不安全函数：

1 不保护共享变量的函数

2 保持跨越多个调用的状态的函数。

3 返回指向静态变量的指针函数。

4 调用线程不安全函数的函数。

2.可重入性

有一类重要的线程安全函数，叫可重入函数，当它们被多个线程调用时，不会引用任何共享数
据，尽管线程安全和可重入不等价，可重入函数属于线程安全函数；

显式可重入函数：不依赖调用者，所有的数据引用都是本地自动栈变量，且参数是传值传递的
（不是指针）；

隐式可重入函数：形参可以使指针，小心的传递指向非共享数据的指针；

3.LINUX 提供不安全的可重入版本（安全）函数名多以_r 结尾；

4.竞争：当程序的结果依赖一个线程要在另一个线程达到 y 点前到达它的控制流

中的 x 点时，就会产生竞争；

5.死锁：一个线程阻塞了，等待一个永远不会为真的条件。

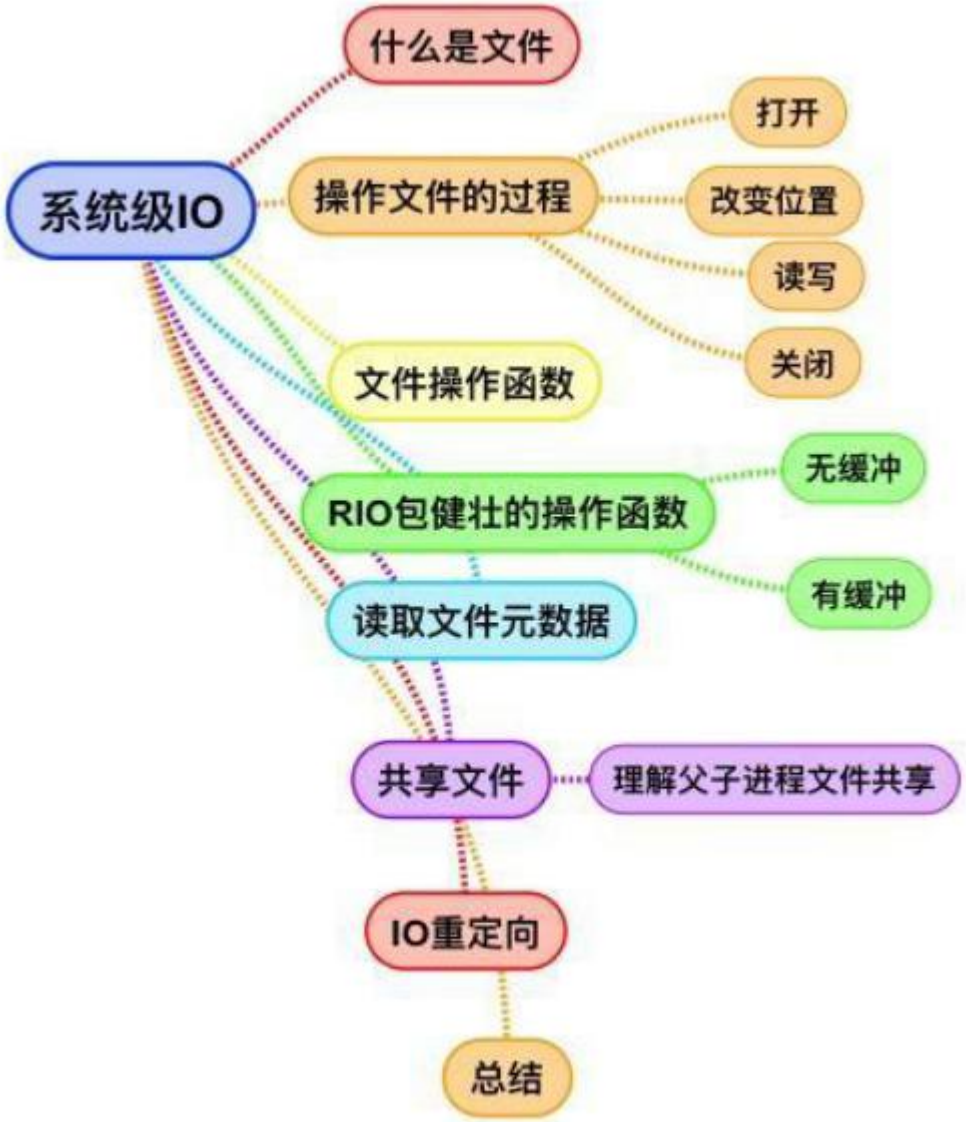
五、系统级 I/O 和网络编程 （感谢缤晴）

此部分全部来自深入理解计算机基础（第 3 版）

48. I/O 相关概念

为了更加有效地管理存储器并且少出错，现代系统提供了一种对主存的抽象概念输入/输出（I/O）是在主存（main memory）和外部设备（例如磁盘驱动器、终端和网络）之间拷贝数据的过程。输入操作是从 I/O 设备拷贝数据到内存，而输出操作则是从主存拷贝数据到 I/O 设备。

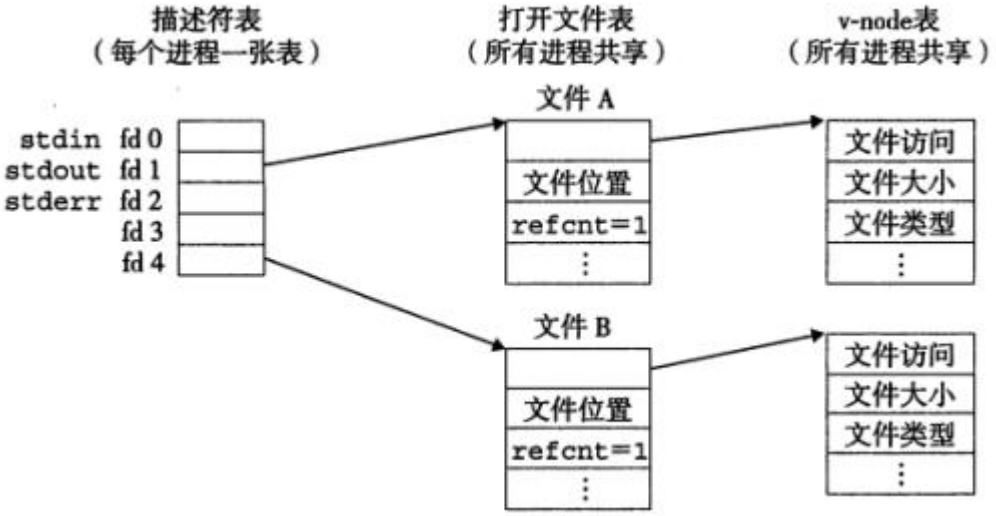
所有的运行时系统都提供执行 I/O 的较高级别的工具，例如，ANSI C 提供标准的 I/O 库，包含像 printf 和 scanf 这样执行带缓冲的 I/O 函数。C++语言用它的重载操作符<<（输入）和>>（输出）提供了类似的功能。在 Unix 系统，是通过使用内核提供的系统级 Unix I/O 函数来实现这戏较高级别的 I/O 函数的。



49. 文件及文件操作

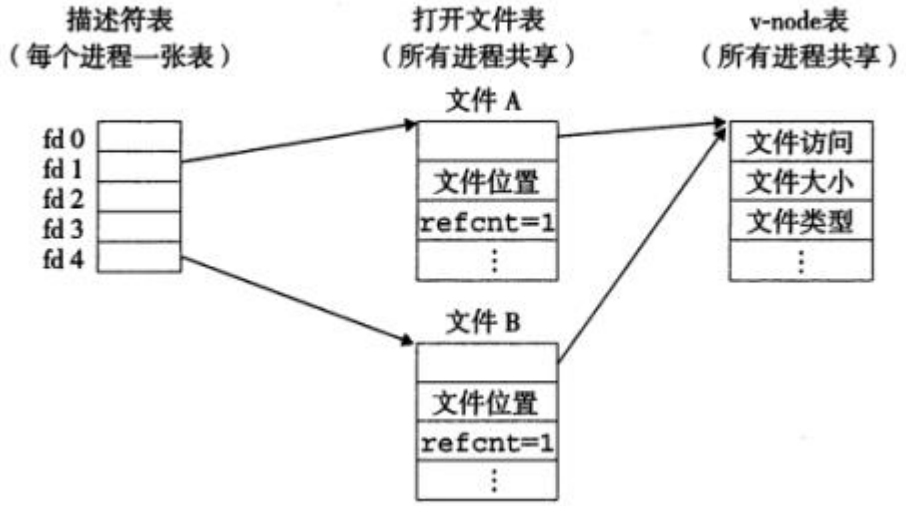
答： 一个 Unix 文件就是一个 m 字节的序列 B0,B1,⋯,Bk,B(m-1)
所有的 I/O 设备，例如网络、磁盘和中断，都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行。这种将设备优雅地映射为文件的方式，允许 Unix 内核以一种统一的且一致的方式来执行：
打开文件
改变当前的文件位置
读写文件
关闭文件

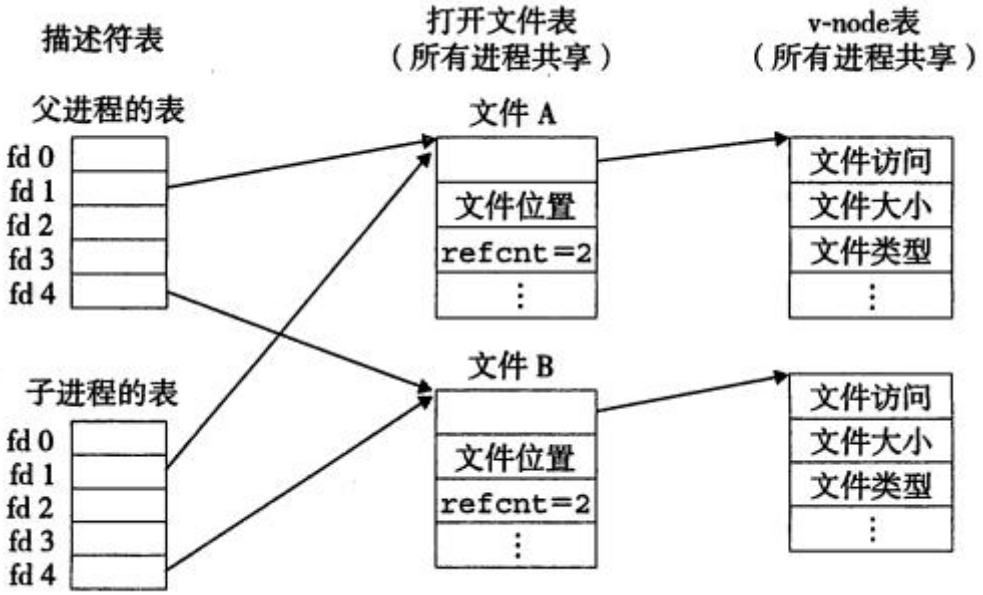
共享文件



无共享

同一个进程的不同表项，通过文件表指向了同一个位置





Unix 共享文件的内核用三种相关的数据结构来表示打开的文件：

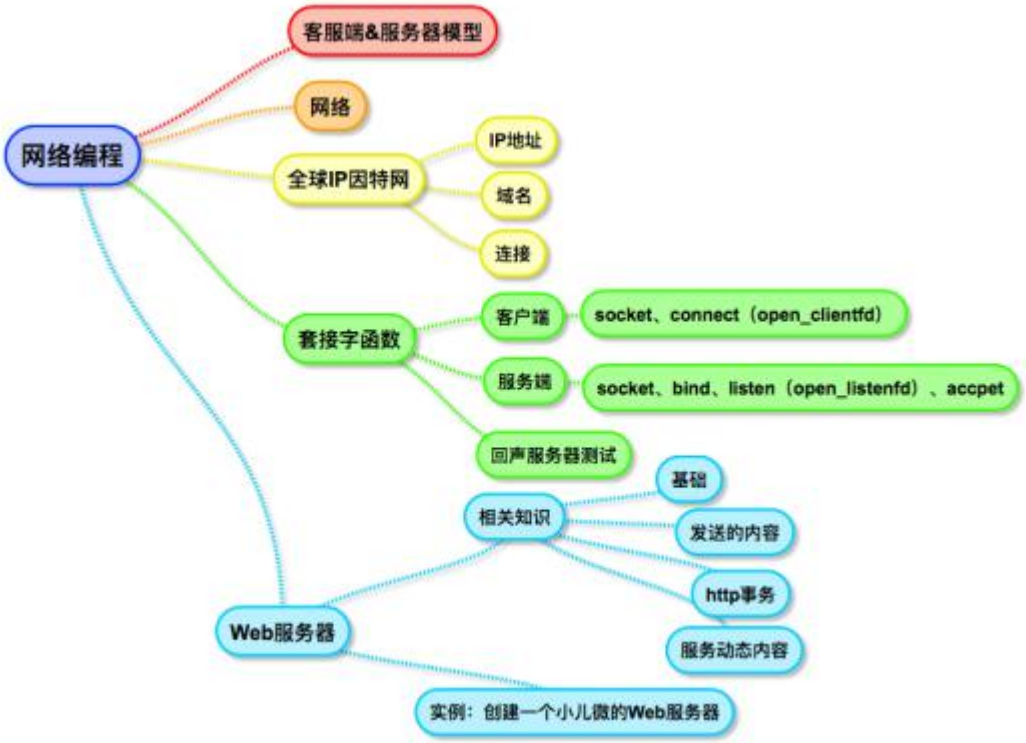
描述符表（descriptor table）：每个文件都有它独立的描述符表，它的表项是由进程打开的文件描述符来索引的。每个打开的描述符表项指向文件表中的一个表项。

文件表（file table）：打开文件的集合是由一张文件表来表示的，所有的进程共享这张表。

v-node 表（v-node table）：同文件表一样，所有的进程共享 v-node 表。

多个描述符也可以通过不同的文件表项来引用同一个文件。关键思想是每个描述符都有它自己的文件位置，所以对不同描述符的读操作可以从文件的不同位置获取数据。

50. 文件、网络编程



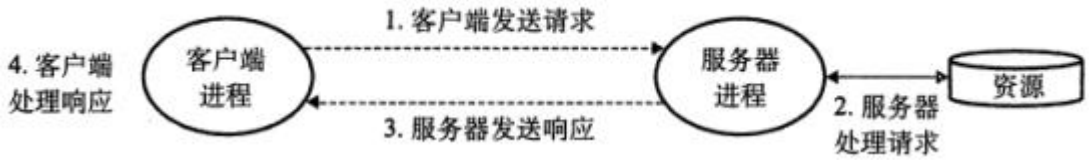
网络应用集成了我们已经学到的很多概念：进程、信号、字节顺序、存储器映射、动态分配等，同时客服端-服务器模型是一个新的知识，我们将所有的这些结合起来，创建一个微小的 Web 服务器，提供浏览器静态和动态的访问。

就是我们平常 TCP 和 UDP 两种编程

TCP 需要三次握手，才能成功连接

UDP 直接丢包，例如视频软件就用 UDP

51. 客户端-服务器模型



每个网络应用都是基于客户端-服务器模型的。

客户端-服务器模型中的基本操作是事务。一个客户端-服务器事务由以下四步组成

1）当一个客户端需要服务时，它向服务器发送一个请求，发起一个事务。

2）服务器收到请求后，解释它，并以适当的方式操作它的资源。

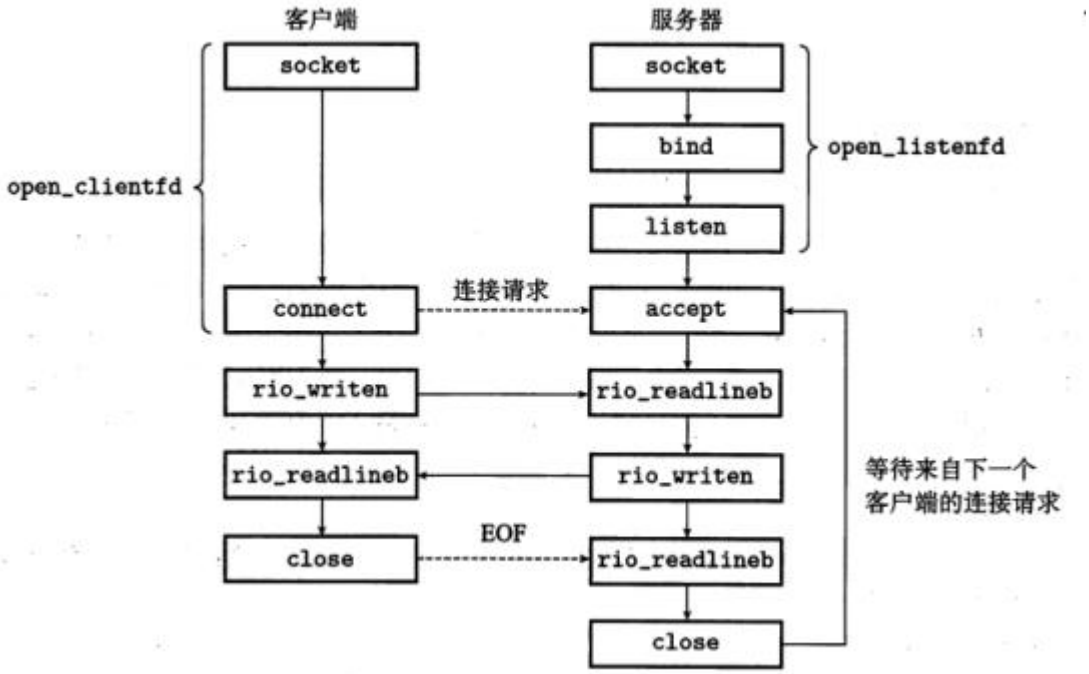
3）服务器给客户端发送一个响应，并等待下一个请求。

4）客户端收到响应并处理它

注意：客户端和服务器是进程，而不是机器或主机，因为一台主机可以同时运行不同的客户端和服务器

52. 套接字接口

答：



一个套接字是连接的一个端点。每个套接字都有相应的套接字地址，由一个因特网地址和一个 16 位的整数端口组成，用“地址：端口”表示

1.套接字接口是一组函数，它们和 Unix I/O 函数结合起来，用以创建网络应用。从 Linux 内核看，一个套接字就是通信的一个端点，从 Linux 程序看，套接字就是一个有相应描述符的打开文件

2.客户端和服务器使用 socket 函数来创建一个套接字描述符

例：对于 ipv4 创建 socket,其中使用 tcp 协议

```
sock = socket(AF_INET,SOCK_STREAM,0);
```

3. 客户端通过调用 connect 函数建立和服务器的连接

4. bind 函数告诉内核将 addr 中的服务器套接字地址和套接字描述符 sockfd 联系起来

5. listen 函数将 sockfd 从一个主动套接字转化为一个监听套接字。

6. TCP 服务器端依次调用 socket()、bind()、listen()之后，就会监听指定的 socket 地址了。TCP 客户端依次调用 socket()、connect()之后就向 TCP 服务器发送了一个连接请求。TCP 服务器监听到这个请求之后，就会调用 accept()函数取接收请求，

这样连接就建立好了。之后就可以开始网络 I/O 操作了，即类同于普通文件的读写 I/O 操作。

7. getaddrinfo 函数将主机名、主机地址、服务名和端口号得字符串表示转化成套接字地址结构，这个函数可重入的，适用于任何协议。

8. 将一个套接字地址结构转换成相应得主机和服务名字符串。

53. HTTP 请求

答：一个 HTTP 请求：一个请求行(request line) 后面跟随 0 个或多个请求报头(request header)，再跟随一个空的文本行来终止报头

请求行： `<method> <uri> <version>`

HTTP 支持许多方法，包括 GET，POST，PUT，DELETE，OPTIONS，HEAD，TRACE。

URI 是相应 URL 的后缀，包括文件名和可选参数

version 字段表示该请求所遵循的 HTTP 版本

请求报头：`<header name> : <header data>` 为服务器提供了额外的信息，例如浏览器的版本类型

HTTP 1.1 中 一个 IP 地址的服务器可以是 多宿主主机，例如 www.host1.com www.host2.com 可以存在于同一服务器上。

HTTP 1.1 中必须有 host 请求报头，如 host:www.google.com:80 如果没有这个 host 请求报头，每个主机名都只有唯一 IP，IP 地址很快将用尽。

54. Web 服务器

答：web 服务器以两种不同的方式向客服端提供内容：

1. 静态内容：取一个磁盘文件，并将它的内容返回给客户端
2. 动态内容：执行一个可执行文件，并将它的输出返回给客户端

第三部分软件工程

考试题型：概念问答题、实践案例题

总分：50 分

一、软件过程 （感谢小涛同学）

55. 软件过程的概念

答：1、软件过程描述为为了开发出客户需要的软件，什么人、在什么时候、做什么事以及怎样做这些事以实现某一个特定的具体目标。ISO9000 把过程定义为：“使用资源将输入转化为输出的活动所构成的系统”。（来自《软件工程导论》p14）

2、过程定义了运用方法的顺序、应该交付的文档资料、为保证软件质量和协调变化所需要采取的管理措施以及标志软件开发各个阶段任务完成的里程碑。（来自《软件工程导论》p14）

3、软件过程是软件生存周期中的一系列相关的过程。过程是活动的集合，活动是任务的集合。 （来自《复旦大学软件工程 ppt》）

软件过程有三层含义：

个体含义：即指软件产品或系统在生存周期中的某一类活动的集合，如软件开发过程，软件管理过程等；

整体含义：即指软件产品或系统在所有上述含义下的软件过程的总体；

工程含义：即指解决软件过程的工程，它应用软件工程的原则、方法来构造软件过程模型，并结合软件产品的具体要求进行实例化，以及在用户环境下的运作，以此进一步提高软件生产率，降低成本。

56. 经典软件过程模型的特点（瀑布模型、增量模型、演化模型、统一过程模型）；

答：

57. 瀑布模型：

- 瀑布模型将软件生命周期划分为需求分析、规格说明、设计、程序编写、软件测试和运行维护等六个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落。
- 瀑布模型强调文档的作用，并要求每个阶段都要仔细验证。
- 瀑布模型模型的线性过程太理想化，已不再适合现代的软件开发模式，几乎被业界抛弃，其主要问题在于：

58. 各个阶段的划分完全固定，阶段之间产生大量的文档，极大地增加了工作量；
59. 由于开发模型是线性的，用户只有等到整个过程的末期才能见到开发成果，从而增加了开发的风险；
60. 早期的错误可能要等到开发后期的测试阶段才能发现，进而带来严重的后果。

61. **增量模型：**

与建造大厦相同，软件也是一步一步建造起来的。

在增量模型中，软件被作为一系列的增量构件来设计、实现、集成和测试，每一个构件是由多种相互作用的模块所形成的提供特定功能的代码片段构成。

- 增量模型在各个阶段并不交付一个可运行的完整产品，而是交付满足客户需求的一个子集的可运行产品。
- 增量模型侧重于每个增量都提交一个可以运行都产品。
- 整个产品被分解成若干个构件，开发人员逐个构件地交付产品，这样做的好处是软件开发可以较好地适应变化，客户可以不断地看到所开发的软件，从而降低开发风险。

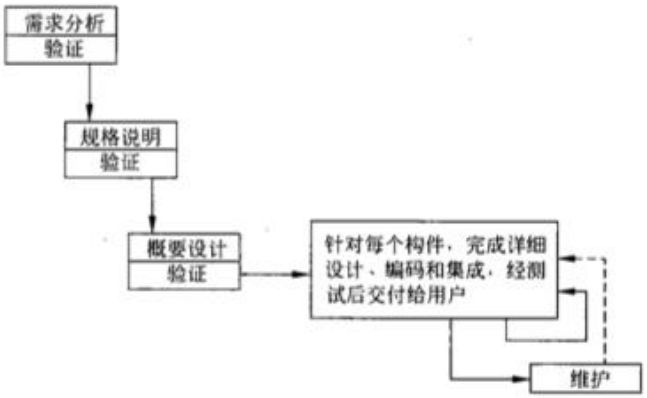


图 1.5 增量模型

但是，增量模型也存在以下缺陷：

62. 由于各个构件是逐渐并入已有的软件体系结构中的，所以加入构件必须不破坏已构造好的系统部分，这需要软件具备开放式的体系结构。

63. 在开发过程中，需求的变化是不可避免的。增量模型的灵活性可以使其适应这种变化的能力大大优于瀑布模型和快速原型模型，但也很容易退化为边做边改模型，从而是软件过程的控制失去整体性。

在使用增量模型时，第一个增量往往是实现基本需求的核心产品。核心产品交付用户使用后，经过评价形成下一个增量的开发计划，它包括对核心产品的修改和一些新功能的发布。这个过程在每个增量发布后不断重复，直到产生最终的完善产品。

64. **演化模型：**

演化模型是一种全局的软件（或产品）生存周期模型。属于迭代开发方法。

即根据用户的基本需求，通过快速分析构造出该软件的一个初始可运行版本，这个初始的软件通常称之为原型，然后根据用户在使用原型的过程中提出的意见和建议对原型进行改进，获得原型的新版本。重复这一过程，最终可得到令用户满意的软件产品。

采用演化模型的开发过程，实际上就是从初始的原型逐步演化成最终软件产品的过程。演化模型特别适用于对软件需求缺乏准确认识的情况。

缺点：

65. 如果所有的产品需求在一开始并不完全弄清楚的话，会给总体设计带来困难及削弱产品设计的完整性，并因而影响产品性能的优化及产品的可维护性。
66. 如果缺乏严格的过程管理的话，这个生命周期模型很可能退化为一种原始的无计划的“试－错－改”模式。

67. **Rational 统一过程模型：**

统一过程（RUP/UP，Rational Unified Process）是一种以用例驱动、以体系结构为核心、迭代及增量的软件过程模型，由 UML 方法和工具支持，广泛应用于各类面向对象项目。

RUP 是由 Rational 公司开发并维护，和一系列软件开发工具紧密集成。RUP 蕴含了大量优秀的实践方法，这些经验被称为“最佳实践”。

最佳实践包括：

- 迭代式软件开发、
- 需求管理、
- 基于构件的构架应用、
- 建立可视化的软件模型、
- 软件质量验证、
- 软件变更控制等。

RUP 的静态结构包括 6 个核心工作流（业务建模、需求、分析设计、实现、测试、部署）和 3 个核心支持工作流（配置与变更管理、项目管理和环境）。

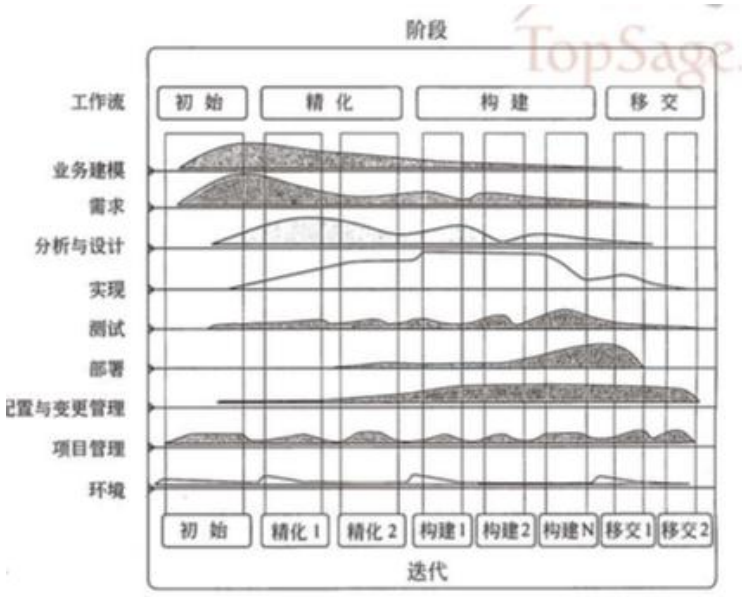


图 1.10 RUP 软件开发生命周期

RUP 把软件的生命周期划分成 4 个连续的阶段。每个阶段都有明确的目标，并且定义了用来评估是否达到这些目标的里程碑。每个阶段的目标通过一次或多次迭代来完成。

4 个阶段的工作目标包括：初始阶段、精华阶段、构建阶段、移交阶段。

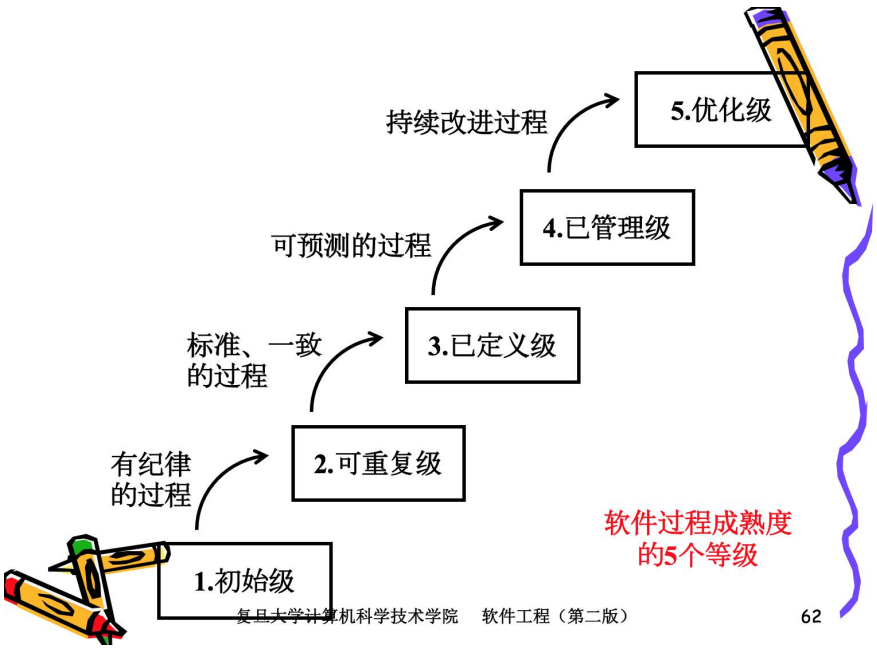
RUP 模型采用迭代开发，通过多次执行不同的开发工作流，逐步确定一部分需求分析和风险，在设计、实现并确认这部分后，再去做下一部分的需求分析、设计、实现和确认工作，依次进行下去，直到整个项目完成，这样能够在逐步集成中更好的理解需求，构建一个健壮的体系结构。

68. **过程评估与 CMM/CMMI 的基本概念；**

答：1、CMM（Capability Maturity Model）即能力成熟度模型，是美国卡耐基梅隆大学软件工程研究所（SEI）在美国国防部资助下于二十世纪八十年代末建立的，用于评价软件机构的软件过程能力成熟度的模型。此模型在建立和发展之初，主要目的在于提供一种评价软件承接方能力的方法，为大型软件项目的招投标活动提供一种全面而客观的评审依据。而发展到后来，又同时被软件组织用于改进其软件过程。（来自《复旦大学软件工程 ppt》）

CMM 提供了一个成熟度等级框架：

- 1 级-初始级
- 2 级-可重复级
- 3 级-已定义级
- 4 级-已管理级
- 5 级-优化级



2、美国国防部、美国国防工业委员会和 SEI/CMU 于 1998 年启动 CMMI 项目，希望 CMMI 是若干过程模型的综合和改进，是支持多个工程学科和领域的系统的、一致的过程改进框架，能适应现代工程的特点和需要，能提高过程的质量和工作效率。（来自《复旦大学软件工程 ppt》）

3、CMMI 模型为每个学科的组合都提供两种表示法：阶段式模型和连续式模型。

4、敏捷宣言与敏捷过程的特点。

答：

69. 敏捷宣言：（来自《复旦大学软件工程 ppt》）

1）个人和交互高于过程和工具

不是否定过程和工具的重要性，而是更强调软件开发中人的作用和交流的作用。

软件是由人组成的团队来开发的，与软件项目相关的各类人员通过充分的交流和有效的合作，才能成功地开发出得到用户满意的软件。

如果光有定义良好的过程和先进的工具，而人员的技能很差，又不能很好地交流和协作，软件是很难成功地开发的。

2）可运行软件高于详尽的文档

通过执行一个可运行的软件来了解软件做了什么，远比阅读厚厚的文档要容易得多。

敏捷软件开发强调不断地快速地向用户提交可运行的软件（不一定是完整的软件），以得到用户的认可。

好的必要的文档仍是需要的，它能帮助我们理解软件做什么，怎么做以及如何使用，但软件开发的

主要目标是创建可运行的软件。

3）与客户协作高于合同（契约）谈判

只有客户才能明确说明需要什么样的软件，然而，大量的实践表明，在开发的早期客户常常不能完整地表达他们的全部需求，有些早期确定的需求，以后也可能会改变。

要想通过合同谈判的方式，将需求固定下来常常是困难的。

敏捷软件开发强调与客户的协作，通过与客户的交流和紧密合作来发现用户的需求。

4）对变更及时做出反应高于遵循计划

任何软件项目的开发都应该制订一个项目计划，以确定各开发任务的优先顺序和起止日期。然而，随着项目的进展，需求、业务环境、技术等都可能变化，任务的优先顺序和起止日期也可能因种种原因会改变。

因此，项目计划应具有可塑性，有变动的余地。当出现变化时及时做出反应，修订计划以适应变化。

2、**敏捷过程的特点**（来自《复旦大学软件工程 ppt》）

（1）最优先的是通过尽早地和不断地提交有价值的软件使客户满意

（2）欢迎变化的需求，即使该变化出现在开发的后期，为了提升对客户的竞争优势，Agile 过程利用变化作为动力

（3）以几周到几个月为周期，尽快、不断地发布可运行软件

（4）在整个项目过程中，业务人员和开发人员必须天天一起工作

（5）以积极向上的员工为中心建立项目组，给予他们所需的环境和支持，对他们的工作予以充分的信任

（6）项目组内效率最高、最有效的信息传递方式是面对面的交流

（7）测量项目进展的首要依据是可运行的软件

（8）敏捷过程提倡可持续的开发，项目发起者、开发者和用户应能长期保持恒定的速度

（9） 应时刻关注技术上的精益求精和好的设计，以增强敏捷性

（10）简单化是必不可少的，这是尽可能减少不必要工作的艺术

（11）最好的构架、需求和设计出自于自我组织的团队

（12）团队要定期反思怎样才能更有效，并据此调整自己的行为

二、软件需求 （感谢 HRR）

70. 软件需求的概念；

答：**主观需求**：用户解决一个问题或达到一个目标所需要的一种状况或能力

客观需求：系统为了满足一种约定、标准、规格说明或其它正式文件而必须满足或拥有的一种状况或能力；

功能性需求：1.系统需要提供的服务或功能：如图书检索 ；2.系统对特定输入的处理方式：如对非法输入的提示；3.系统在特定环境下的行为：如长时间无操作时的屏保

非功能性需求 1.对系统功能或服务附加的质量约束，例如响应时间、容错性、安全性等——客户所关心的(外部质量)；2.从系统开发和维护角度出发的质量属性，例如可理解性、可扩展性、可配置性等——软件开发或维护者所关心的(内部质量、软件所特有)

71. **需求工程的基本过程；**

答：需求获取、需求分析与协商、系统建模、需求规约、需求验证、需求管理

72. **分层数据流模型；**

答：分层数据流图的设计方法 第一步，画子系统的输入输出　　把整个系统视为一个大的加工，然后根据数据系统从哪些外部实体接收数据流，以及系统发送数据流到那些外部实体，就可以画出输入输出图。这张图称为顶层图。第二步，画子系统的内部　　把顶层图的加工分解成若干个加工，并用数据流将这些加工连接起来，使得顶层图的输入数据经过若干加工处理后，变成顶层图的输出数据流。这张图称为 0 层图。从一个加工画出一张数据流图的过程就是对加工的分解。 可以用下述方法来确定加工：在数据流的组成或值发生变化的地方应该画出一个加工，这个加工的功能就是实现这一变化，也可以根据系统的功能决定加工。确定数据流的方法 用户把若干数据当作一个单位来处理（这些数据一起到达、一起处理）时，可以把这些数据看成一个数据流。 关于数据存储 对于一些以后某个时间要使用的数据，可以组织成为一个数据存储来表示。 第三步，画加工的内部　　把每个加工看作一个小系统，把加工的输入输出数据流看成小系统的输入输出流。于是可以象画 0 层图一样画出每个小系统的加工的 DFD 图。 第四步，画子加工的分解图　　对第三步分解出来的 DFD 图中的每个加工,重复第三步的分解过程，直到图中尚未分解的加工都是足够简单的（即不可再分解）。至此，得到了一套分层数据流图。 第五步，对数据流图和加工编号　　对于一个软件系统，其数据流图可能有许多层，每一层又有许多张图。为了区分不同的加工和不同的 DFD 子图，应该对每张图进行编号，以便于管理。

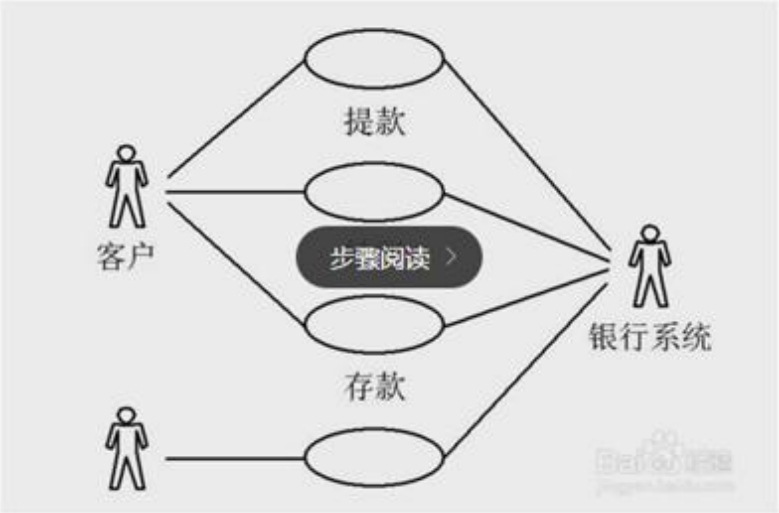
- 顶层图只有一张，图中的加工也只有一个，所以不必为其编号。

- 0 层图只有一张，图中的加工号分别是 0.1、0.2、…，或者 1， 2 。
- 子图就是父图中被分解的加工号。
- 子图中的加工号是由图号、圆点和序号组成，如：1.12，1.3 等等。

73. 用例和场景建模及其 UML 表达（用例图、活动图、泳道图、顺序图）

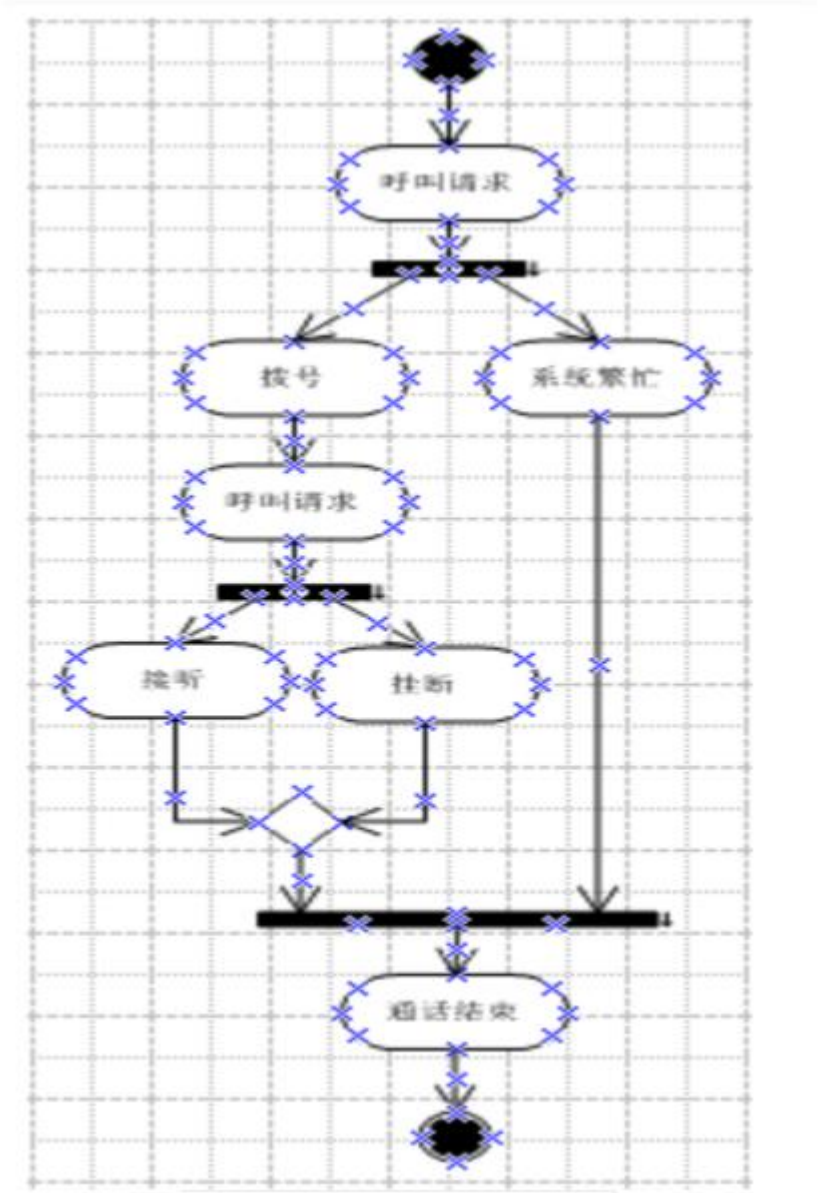
用例图：

用例是对一个活动者使用系统的一项功能时所进行的交互过程的一个文字描述序列。对系统的用户需求的描述，表达的是系统的功能和所提供的服务，它只描述活动者和系统在交互过程中做些什么，并不描述怎么做。



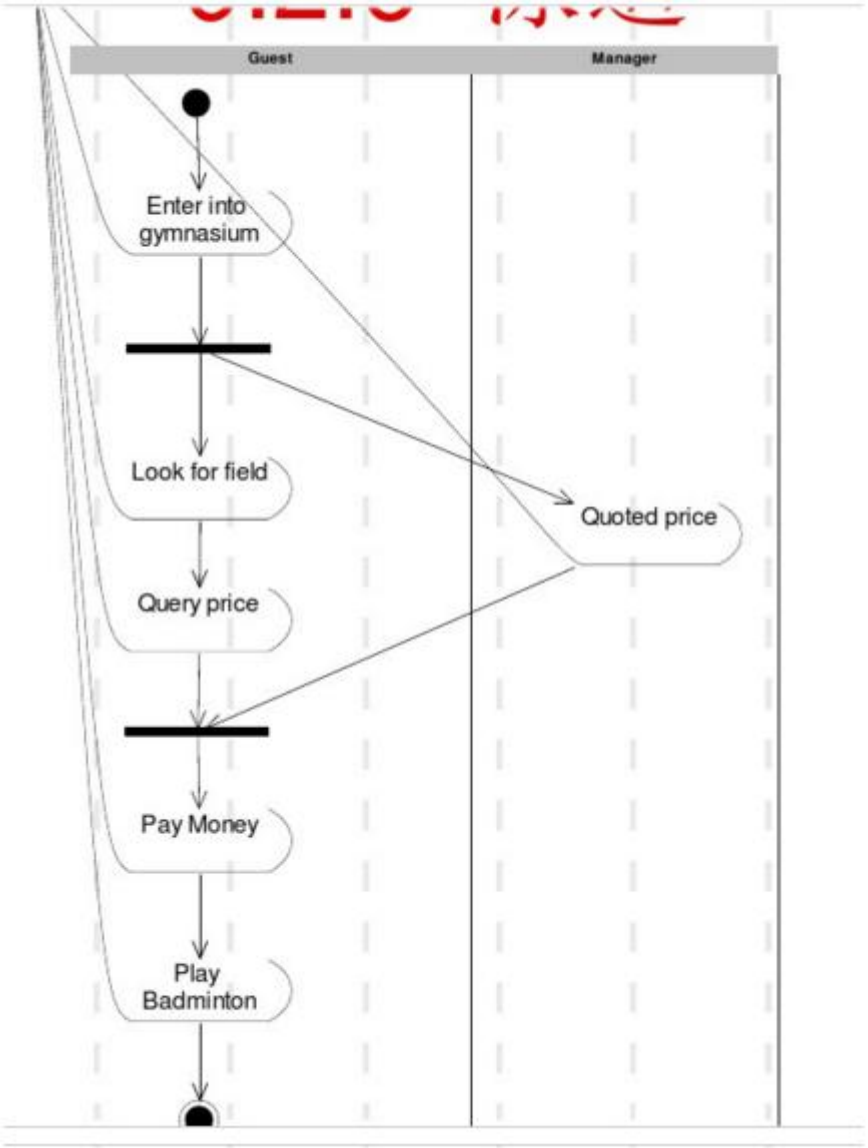
活动图：

活动图是状态图的一种特殊情况。用于简化描述一个过程或者操作的工作步骤。活动用圆角矩形表示——比状态图更窄，更接近椭圆。一个活动中的处理一旦完成，则自动引起下一个活动的发生。箭头表示从一个活动转移到下一个活动。和状态图类似，活动图中的起点用一个实心圆表示，终点用一个同心圆（内圆为实心圆）表示。在活动图中可以带判定点，即一组条件引发一条执行路径，另一组条件则引发另一条执行路径，并且这两条执行路径是互斥的。判定点常用小的菱形图标表示，同时在相关路径的附近指明引起这条路径被执行的条件，条件用方括号括起来。请用活动图描述打电话过程。



泳道图：泳道将活动图中的活动划分为若干组。并将每一组指定给负责这组活动的业务组织。

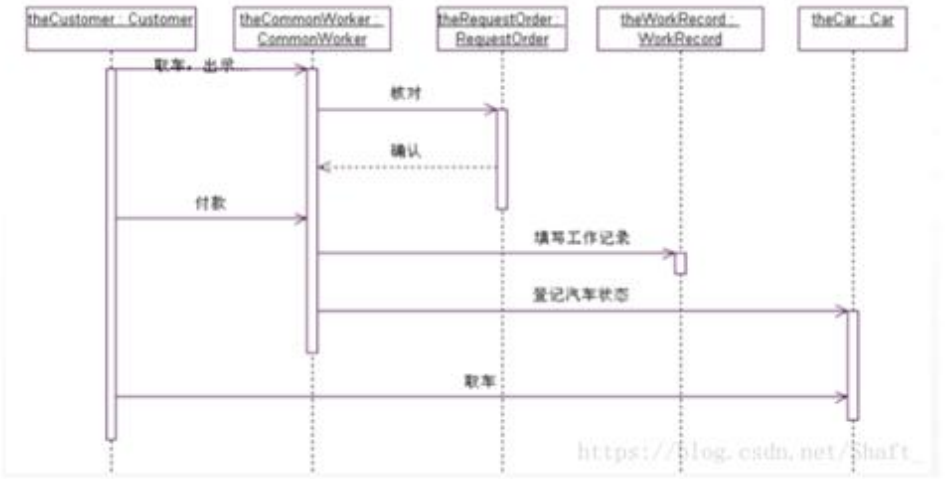
在活动图中，泳道使用垂直的实线绘制。


Guest
Manager
Enter into gymnasium
Look for field
Query price
Quoted price
Pay Money
Play Badminton

顺序图：

顺序图是强调消息时间的交互图，其描述了对象之间传送消息的时间顺序，用来表示用例中的行为顺序。在该二维图中，对象由左至右排列，消息则沿着纵轴由时间顺序排列。在构筑该图时，应布局简洁。

一、顺序图示例（购买小车简图）

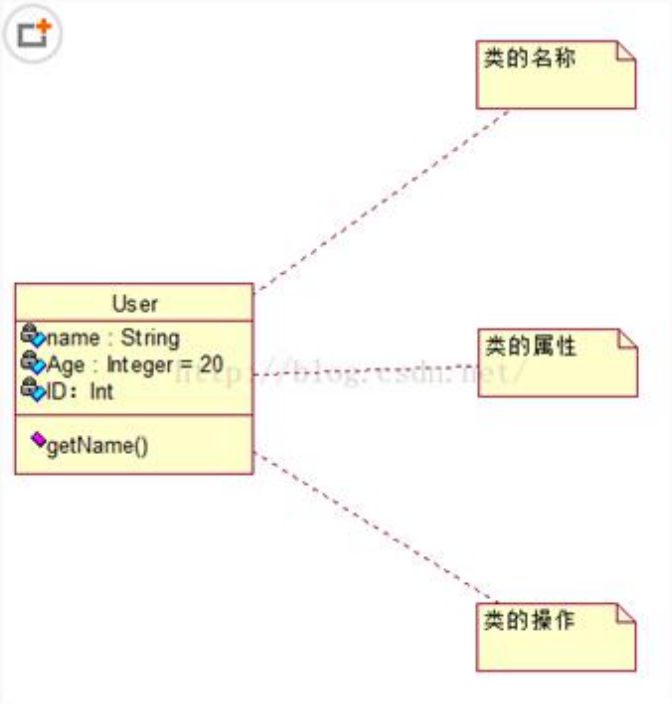


74. **数据模型建模及其 UML 表达（类图）**

答：类图（Class Diagram）是描述类、接口、协作以及它们之间关系的图，用来显示系统中各个类的静态结构。类图是定义其他图的基础，在类图基础上，可以使用状态图、协作图、组件图和配置图等进一步描述系统其他方面的特性。

在UML中，类被表述成为具有相同结构、行为和关系的一组对象的描述符号。所用的属性与操作都被附在类中。类定义了一组具有状态和行为的对象。其中，属性和关联用来描述状态。属性通常使用没有身份的数据值来表示，如数字和字符串。关联则使用有身份的对象之间的关系表示。行为由操作来描述，方法是操作的具体实现。对象的生命周期则由附加给类的状态机来描述。

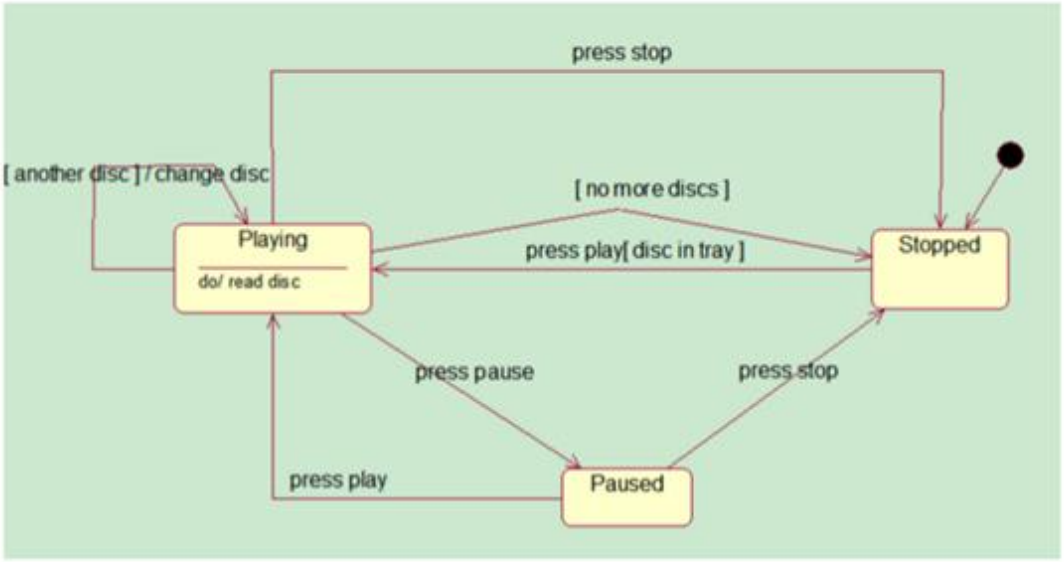
在UML的图形表示中，类的表示法是一个矩形，这个矩形由三个部分构成，分别是类的名称(Name)、类的属性(Attribute)和类的操作(Operation)。类的名称位于矩形的顶端，类的属性位于矩形的中间部位，而矩形的底部显示类的操作。中间部位不仅显示类的属性，还可以显示属性的类型以及属性的初始化值等。矩形的底部也可以显示操作的参数表和返回类型等，如图1所示。



在类的构成中还应当包含类的职责(Responsibility)、类的约束(Constraint)和类的注释(Note)等信息。

75. 行为模型建模及其 UML 表达（状态机图）

答：状态机图是用来为对象的状态及造成状态改变的事件建模。UML 的状态机图主要用于建立对象类或对象的动态行为模型，表现一个对象所经历的状态序列，引起状态或活动转移的事件，以及因状态或活动转移而伴随的动作。状态机图也可用于描述 Use Case,以及全系统的动态行为。



图表 12CD 播放器状态图

三、软件设计与构造 （感谢 yangxu）

76. **软件体系结构及体系结构风格的概念;**

答：**软件体系结构（百度版本）:**具有一定形式的结构化元素，即构件的集合，包括处理构件、数据构件和连接构件。处理构件负责对数据进行加工，数据构件是被加工的信息，连接构件把体系结构的不同部分组组合连接起来。这一定义注重区分处理构件、数据构件和连接构件，这一方法在其他的定义和方法中基本上得到保持。由于软件系统具有的一些共通特性，这种模型可以在多个系统之间传递，特别是可以应用到具有相似质量属性和功能需求的系统中，并能够促进大规模软件的系统级复用。

软件体系结构（指定教材版本）: 程序或计算机系统的软件体系结构是指系统的一个或者多个结构，它包括软件构件、构件的外部可以见属性以及它们之间的相互关系。

体系结构并非可运行的软件。它是一种表达，能达到以下三种目的：

77. 对设计在满足既定需求方面的有效性分析
78. 在设计变更相对容易的阶段，考虑体系结构可能的选择方案
79. 降低与软件构建相关的风险

体系结构风格:对软件体系结构风格的研究和实践促进了对设计的复用，一些经过实践证实的解决方案也可以可靠地用于解决新的问题。体系结构风格的不变部分使不同的系统可以共享同一个实现代码。只要系统是使用常用

的、规范的方法来组织，就可使别的设计者很容易地理解系统的体系结构。

例如，如果某人把系统描述为"客户/服务器"模式，则不必给出设计细节，

我们立刻就会明白系统是如何组织和工作的。

下面是 Garlan 和 Shaw 对通用体系结构风格的分类：

（1）数据流风格：批处理序列；管道/过滤器

（2）调用/返回风格：主程序/子程序；面向对象风格；层次结构

（3）独立构件风格：进程通讯；事件系统

（4）虚拟机风格：解释器；基于规则的系统

（5）仓库风格：数据库系统；超文本系统；黑板系统

80. **设计模式的概念；（来源百度）**

答：软件设计模式（Design pattern），又称设计模式，是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性、程序的重用性。

81. **模块化设计的基本思想及概念（抽象、分解、模块化、封装、信息隐藏、功能独立）；（来源教材）**

答：1 模块化：模块化设计思想是吧整个软件划分成几个独立命名的或者是独立访问的构成成分，这些模块集合起来就满足了问题的需要，就能把一个大而复杂的软件系统划分成易于理解的比较单纯的模块化结构。

模块化的要求：需要满足信息隐蔽原则，模块之间相互独立，并且尽量符合高内聚低耦合的要求

2 封装：是一种信息隐蔽技术，用户只能看到对象封装界面上的信息，对象的内部实现对用户是隐蔽的。封装的目的是使对象的使用和生产者分离。使对象的定义和实现分开。一个对象通常由对象名、属性和操作三部分组成

3 信息隐蔽：每个模块的实现细节对于其它模块来说是隐蔽的（不可访

问）也就是说，模块中所包含的信息（数据和过程）不允许其它不需要这些信息的模块使用。

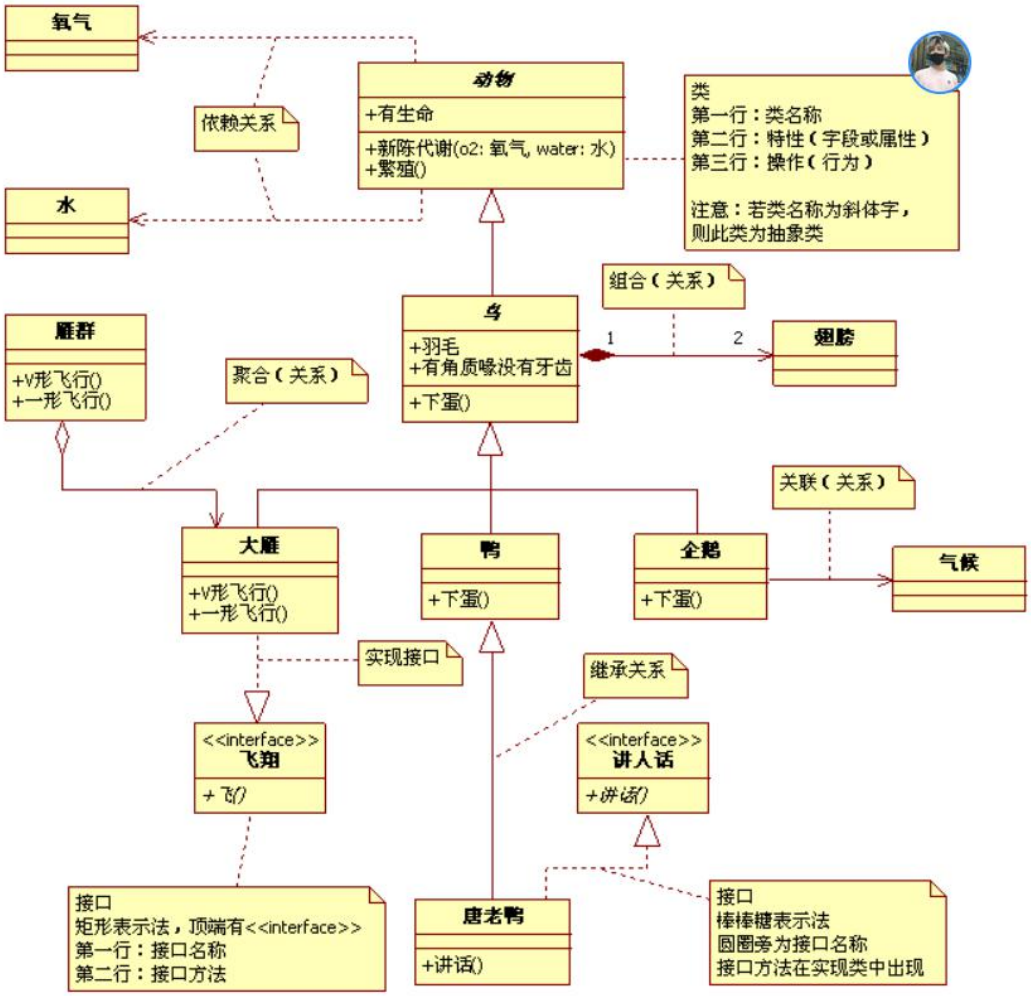
信息隐蔽作用：设计时把一些可能发生变化的因素隐蔽在某个模块中，以提高可维护性，并且可以减少错误向外传播

82. **软件重构的概念；软件体系结构的 UML 建模（包图、类图、构件图、顺序图、部署图）；（来源博客与教材）**

答：1.重构：以不改变代码外部行为而改进其内部结构的方式来修改软件系统的过程。这是一种净化代码以尽可能减少引入错误的严格方法。

2.包图（略，考察可能性极低考）：属于静态图的一种

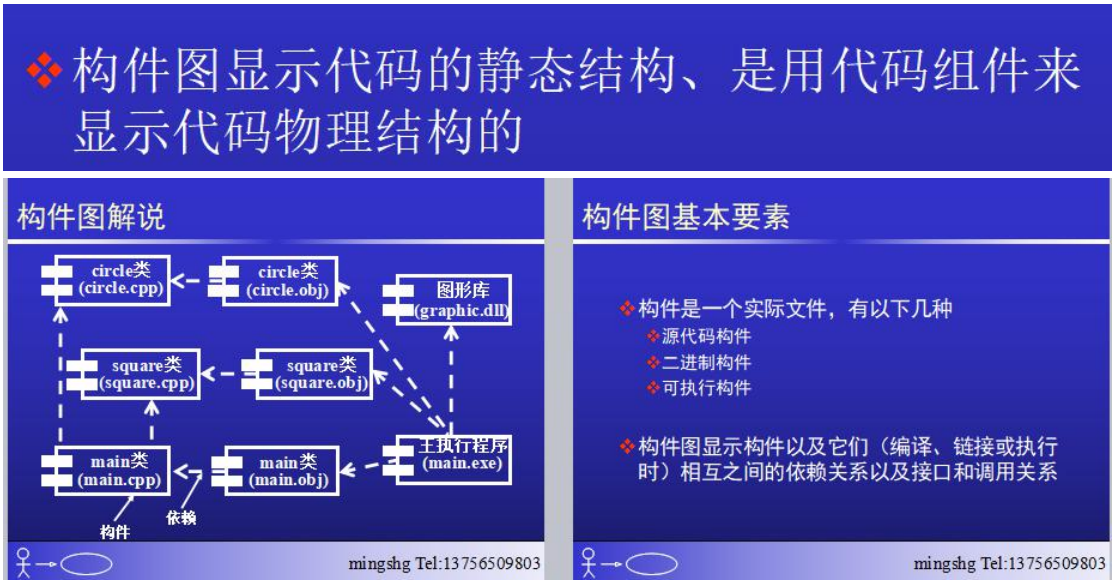
3.类图（重点!!!! 2018 考过）：展示系统中类的静态结构，即类与类之间的相互联系。



具体含义及其画法（看网站一学就会）

https://blog.csdn.net/monkey_d_meng/article/details/6005764（博客 ）

4.构件图



5.顺序图（重点）：用来显示对象之间发送消息的顺序，以及对象之间的交互。例题如下：

下面列出了打印文件时的工作流，请画出对应用于该工作流的顺序图。

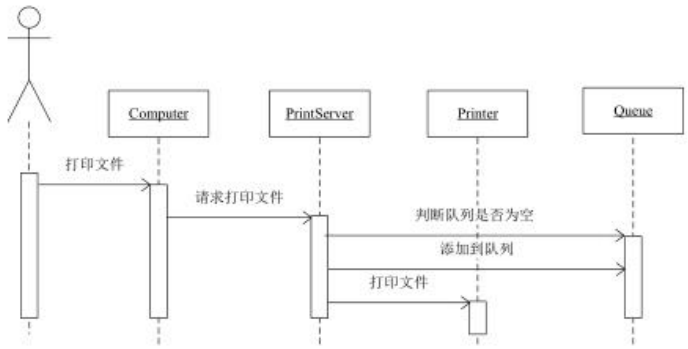
用户通过计算机指定要打印的文件。

打印服务器根据打印机是否空闲，操作打印机打印文件。

如果打印机空闲，则打印机打印文件；

如果打印机忙，则将打印消息存放在队列中等待。

经分析人员分析确认，该系统共有四个对象 Computer、PrintServer、Printer 和 Queue。



6.部署图(略，考察几率低):展现系统中硬件和软件的物理机构

画法：https://blog.csdn.net/wangyongxia921/article/details/8250129

83. 接口的概念；面向对象设计原则（开闭原则、Liskov 替换原则、依赖转置原则、接口隔离原则）；（来源学长资料）

答：开闭原则:

类的改动是通过增加代码进行的,而不是修改源代码

里氏代换原则:

任何抽象类出现的地方都可以用他的实现类进行替换，实际就是虚拟机制，语言

级别实现面向对象功能

依赖倒转原则:

依赖于抽象(接口),不要依赖具体的实现(类)，也就是针对接口编程

接口隔离原则:

不应该强迫用户的程序依赖他们不需要的接口方法。一个接口应该只提供一种对

外功能，不应该把所有操作都封装到一个接口中去。

84. 内聚与耦合的概念、常见的内聚和耦合类型。

答：“高内聚，低耦合”

起因：模块独立性指每个模块只完成系统要求的独立子功能，并且与其他模块的联系最少且接口简单，两个定性的度量标准——耦合性和内聚性。

耦合性也称块间联系。指软件系统结构中各模块间相互联系紧密程度的一种度量。模块之间联系越紧密，其耦合性就越强，模块的独立性则越差。模块间耦合高低取决于模块间接口的复杂性、调用的方式及传递的信息。

耦合性分类(低——高)：无直接耦合;数据耦合;标记耦合;控制耦合;公共耦合;内容耦合;

1 无直接耦合：两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的

2 数据耦合：指两个模块之间有调用关系，传递的是简单的数据值，相当于高级语言的值传递;

3 标记耦合：指两个模块之间传递的是数据结构，如高级语言中的数组名、记录名、文件名等这些名字即标记，其实传递的是这个数据结构的地址;

4 控制耦合：指一个模块调用另一个模块时，传递的是控制变量（如开关、标志等），被调模块通过该控制变量的值有选择地执行块内某一功能;

5 公共耦合：指通过一个公共数据环境相互作用的那些模块间的耦合。公共耦合的复杂程序随耦合模块的个数增加而增加。

6 内容耦合：这是最高程度的耦合，也是最差的耦合。当一个模块直接使用另一个模块的内部数据，或通过非正常入口而转入另一个模块内部。

内聚性又称块内联系。指模块的功能强度的度量，即一个模块内部各个元素彼此结合的紧密程度的度量。若一个模块内各元素（语名之间、程序段之间）联系的越紧密，则它的内聚性就越高。

内聚性分类(低——高)：偶然内聚;逻辑内聚;时间内聚;通信内聚;顺序内聚;功能内聚;

1 偶然内聚：指一个模块内的各处理元素之间没有任何联系。

2 逻辑内聚：指模块内执行几个逻辑上相似的功能，通过参数确定该模块完成哪一个功能。

3 时间内聚：把需要同时执行的动作组合在一起形成的模块为时间内聚模块。
4 通信内聚：指模块内所有处理元素都在同一个数据结构上操作（有时称之为信息内聚），或者指各处理使用相同的输入数据或者产生相同的输出数据。
5 顺序内聚：指一个模块中各个处理元素都密切相关于同一功能且必须顺序执行，前一功能元素输出就是下一功能元素的输入。
6 功能内聚：这是最强的内聚，指模块内所有元素共同完成一个功能，缺一不可。与其他模块的耦合是最弱的。

耦合性与内聚性是模块独立性的两个定性标准，将软件系统划分模块时，尽量做到高内聚低耦合，提高模块的独立性，为设计高质量的软件结构奠定基础。

四、软件测试 （感谢 Sky）

85. **软件测试及测试用例的概念；**

答：软件测试是在规定的条件下对程序进行操作，以发现程序错误，衡量软件质量，并对其是否能满足设计要求进行评估的过程。

测试用例是为某个特殊目标而编制的一组测试输入、执行条件以及预期结果，以便测试某个
程序路径或核实是否满足某个特定需求。

86. **单元测试、集成测试、确认测试、系统测试、回归测试的概念；**

答：单元测试又称模块测试，着重对软件设计的最小单位—程序模块进行验证。单元测试根据设计描述，对重要的控制路径进行测试，以发现各模块内部的错误。单元测试通常采用白盒测试，并且多个模块可以并行进行测试。

集成测试又称组装测试、联合测试，经单元测试后，每个模块都能独立工作，但把它们放在
一起往往不能正常工作。

确认测试以软件需求规格说明书为依据，检查软件的功能和性能及其它特性是否与用户的需
求一致，包括合同规定的全部功能和性能、文档资料（正确且合理）、其它需求（如可移植
性、兼容性、错误恢复能力、可维护性等）。

系统测试是将通过确认测试的软件，作为整个基于计算机系统的一个元素，与其它系统成分
（如硬件、外设、某些支持软件、数据和人员等）集成起来，在实际运行环境下，对计算机
系统进行一系列的集成测试和确认测试。系统测试的目的在于通过与系统

的需求定义作比
较，发现软件与系统的定义不符合或与之矛盾的地方。
回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码
产生错误。

87. **调试的概念、调试与测试的关系；**

答：测试的目的是发现错误，调试（也称排错）的目的是确定错误的原因和准确位置，并加以纠正。

调试与测试的关系：测试和调试在目标、方法和思路上都有所不同，测试是一种过程，目的是显示存在的软件错误，通常由软件测试工程师实施。而调试是一种方法，一种手段，目的是发现错误原因并解决，一般来说调试是测试后的活动，通常由开发工程师实施。

88. **测试覆盖度的概念；**

答：测试覆盖度评估是衡量阶段性软件测试执行状态的重要手段之一，来确定测试是否达到事先设定的测试任务完成的标准。测试覆盖率则是测试覆盖度评估中一种量化的表示方法，一般通过被测试的软件产品需求、功能点、测试用例数或程序代码行等来进行计算。

89. **白盒测试、黑盒测试的概念**

答：白盒测试又称结构测试，这种方法把测试对象看作一个透明的盒子，测试人员根据程序内部的逻辑结构及有关信息设计测试用例，检查程序中所有逻辑路径是否都按预定的要求正确地工作。

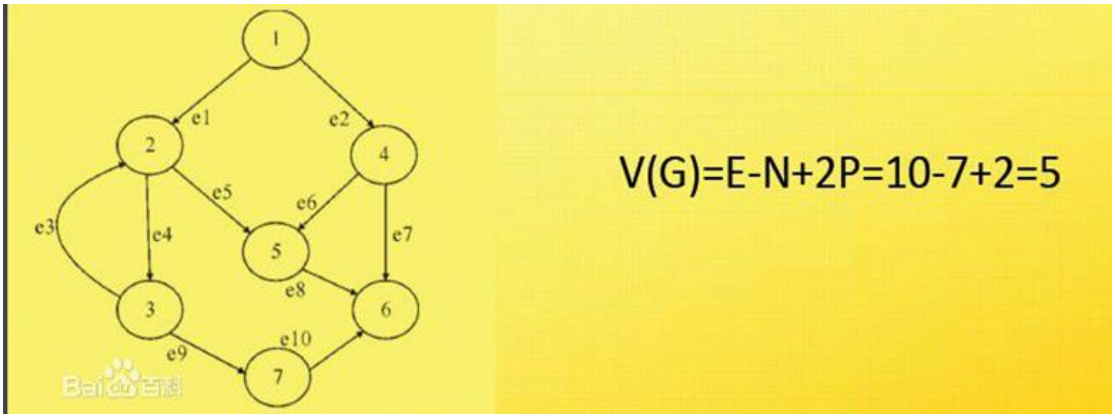
黑盒测试又叫做功能测试，这种方法是把测试对象看做一个黑盒子，测试人员完全不考虑程
序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它
的功能说明。

90. **代码圈复杂度的计算方法；**

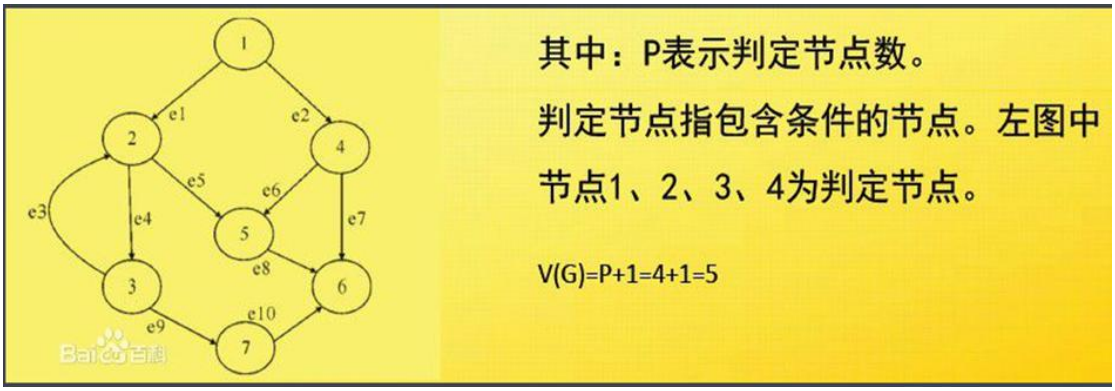
答：圈复杂度是一种代码复杂度的衡量标准。在软件测试的概念里，圈复杂度用来衡量一个模块判定结构的复杂程度，数量上表现为独立线性路径条数，即合理的预防错误所需测试的最少路径条数，圈复杂度大说明程序代码可能质

量低且难以测试和维护，根据经验，程序的可能错误和高的圈复杂度有着很大关系。它的计算方法为：

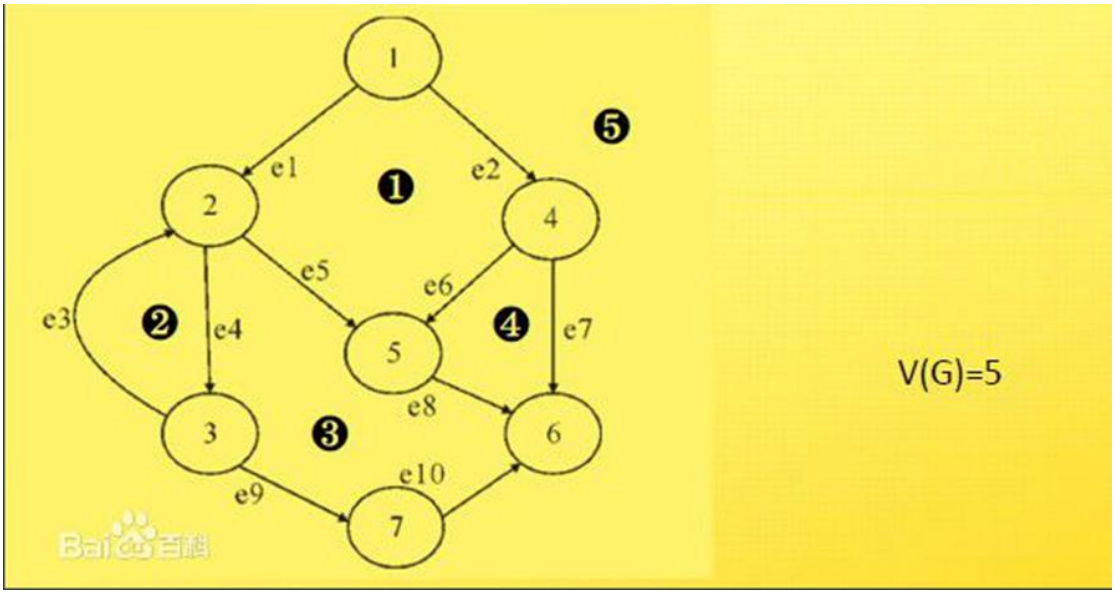
91. V(G)=e-n+2p。e 表示控制流图中边的数量，n 表示控制流图中节点的数量，p 表示图的连接组件数目（图的组件数是相连节点的最大集合），因为控制流图都是连通的，所以 p 永远为 1



92. V(G)=区域数=判定节点数+1



93. V(G)=R。R 代表平面被控制流图划分成的区域数



94. **白盒测试中的基本路径测试方法；**

答：基本路径测试是一种白盒测试技术，这种方法首先根据程序或设计图画出控制流图，并计算其区域数，然后确定一组独立的程序执行路径（称为基本路径），最后为每一条基本路径设计一个测试用例。在实际问题中，一个不太复杂的程序，其路径数可能很大，特别在包含循环时。因此要把覆盖的路径数压缩到一定的限度内。

95. **黑盒测试中的等价类划分方法。**

答：等价类划分是指由于不能穷举所有可能的输入数据来进行测试，所以只能选择少量有代表性的输入数据，来揭露尽可能多的程序错误，等价类划分方法将所有可能的输入数据划分成若干个等价类，然后在每个等价类中选取一个代表性的数据作为测试用例，测试用例由有效等价类和无效等价类的代表组成，从而保证测试用例具有完整性和代表性。