# 第一章、绪论

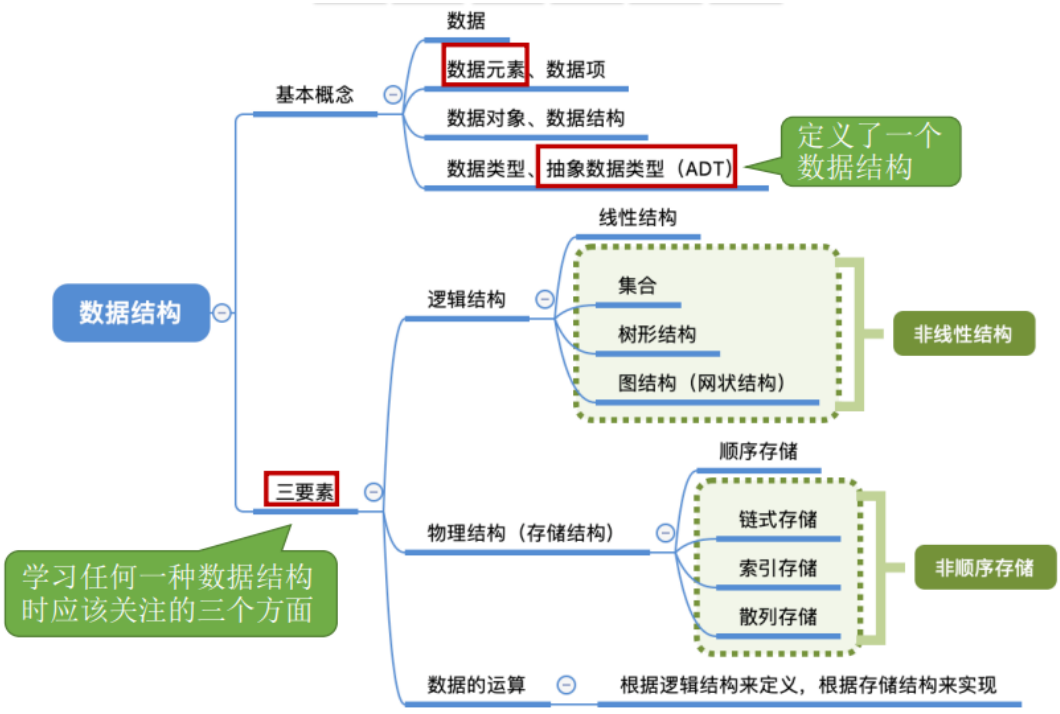
## 一、开篇——数据结构在学什么?

数据结构在学什么?

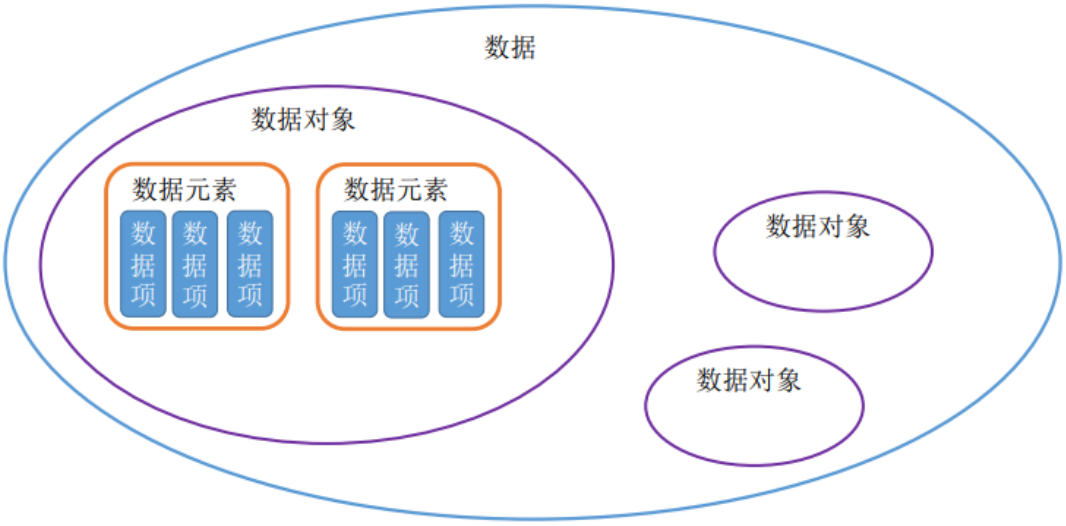
- 如何用程序代码把现实世界的问题信息化
- 如何用计算机高效地处理这些信息从而创造价值

## 二、数据结构的基本概念

知识总览

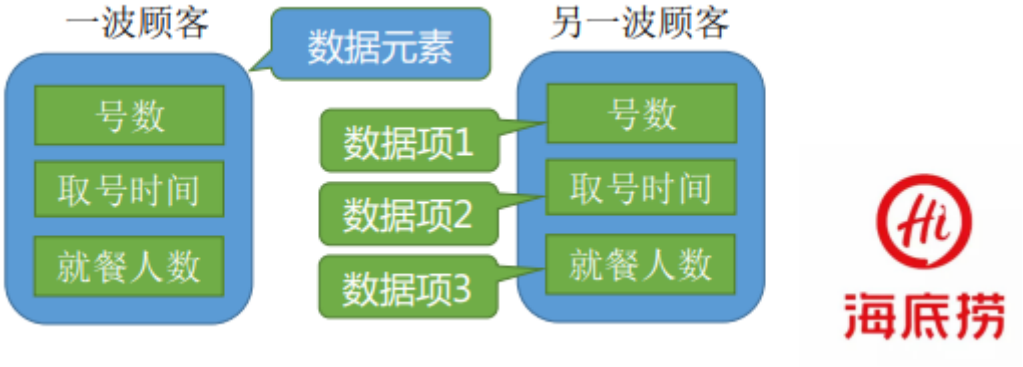



### 1. 基本概念




**数据**：数据是 *信息的载体*，是描述客观事物属性的数、字符及所有能输入到计算机中并被计算机程序识别和处理的符号的集合。数据是计算机程序加工的原料。

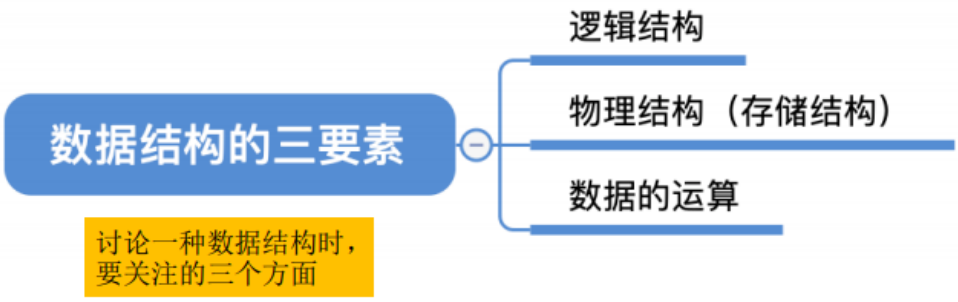
**数据元素**：*数据元素是数据的基本单位*，通常 *作为一个整体* 进行考虑和处理。一个数据元素可由若干数据项组成，数据项是构成数据元素的不可分割的 *最小单位*。



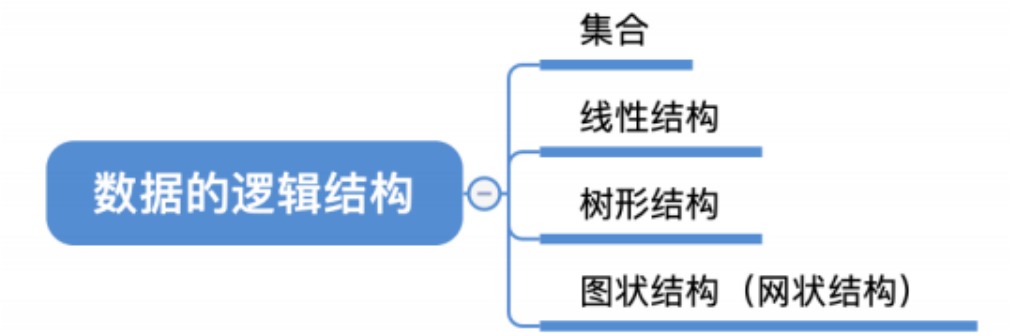

**数据结构**：数据结构是相互之间存在一种或多种 *特定关系* 的数据元素的集合。

**数据对象**：是具有 *相同性质* 的数据元素的集合，是数据的一个子集。

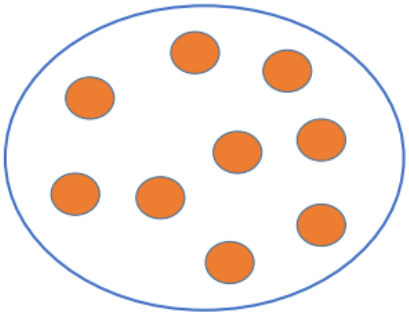
## 2. 数据结构的三要素




**逻辑结构**：数据元素之间的逻辑关系



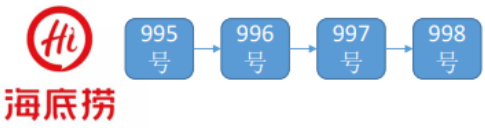

- *集合*：各个元素同属一个集合，别无其他关系



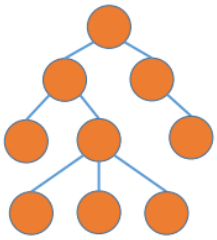
集合

- *线性结构*：数据元素之间是一对一的关系。 除了第一个元素，所有元素都有唯一前驱； 除了最后一个元素，所有元素都有唯一后继。

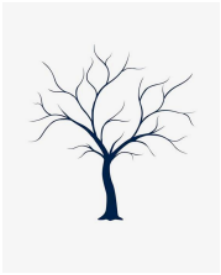
线性结构

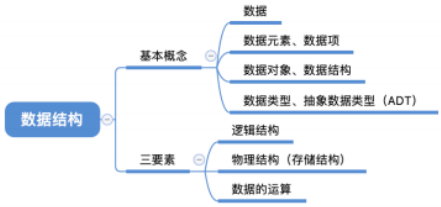


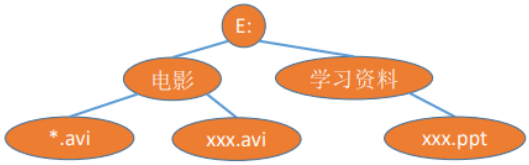

- *树形结构*：数据元素之间是 一对多的关系



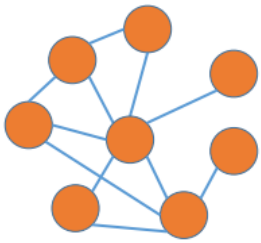
树形结构



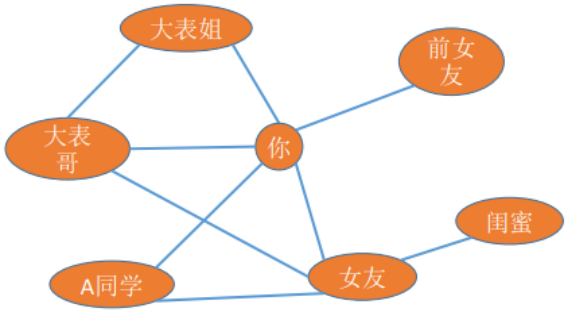



- *图结构*： 数据元素之间是 多对多的关系

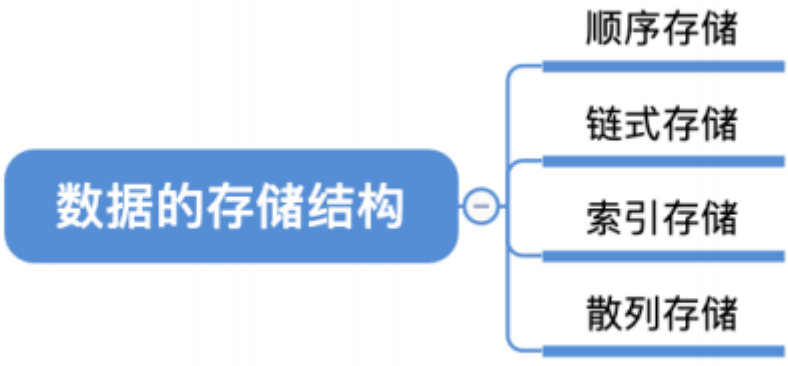


图结构




微信好友

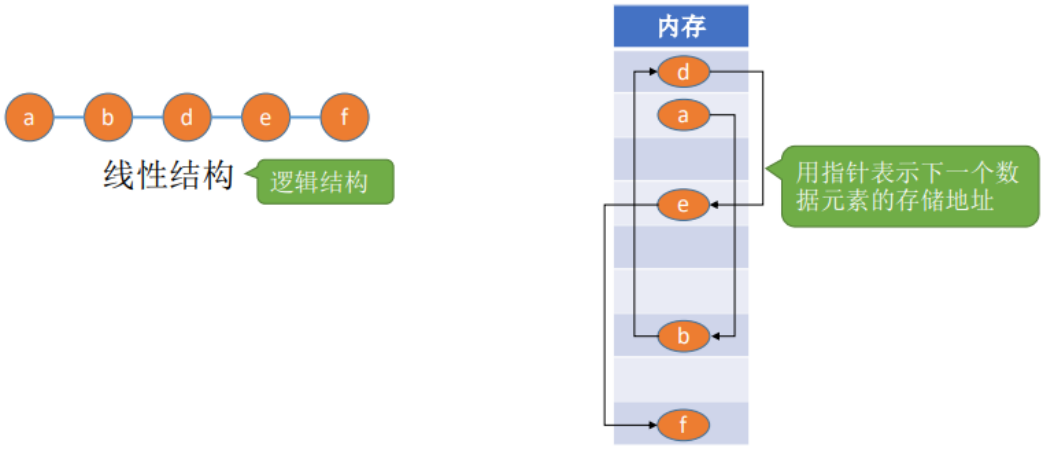
> **存储结构（物理结构）**： 计算机表示数据元素的逻辑关系



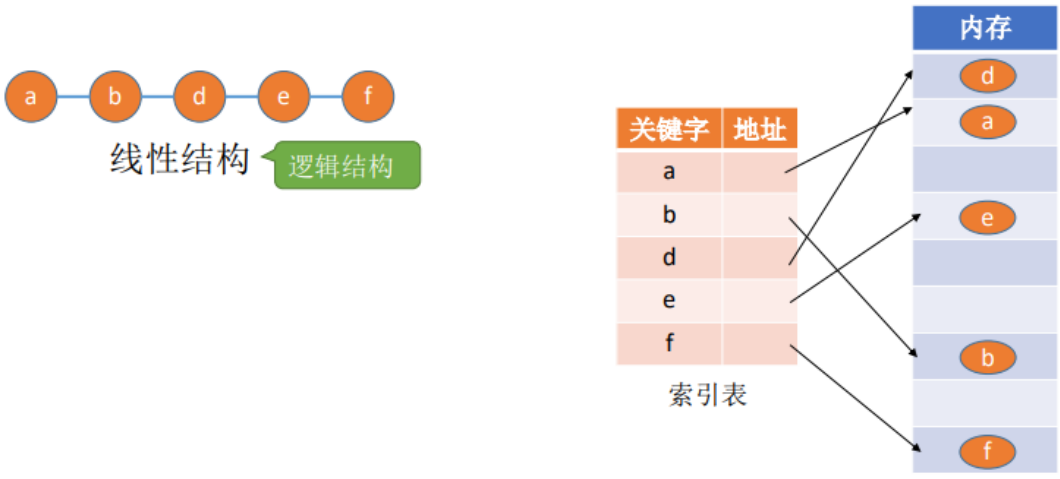

以线性结构来举例表示在计算机中的4种不同的存储结构

- *顺序存储*： 把 *逻辑上相邻的元素存储在物理位置上也相邻的存储单元中*， 元素之间的关系由存储单元的邻接关系来体现。


- *链式存储*： *逻辑上相邻的元素在物理位置上可以 不相邻*，借助指示元素存储地址的指针来表示元素之间的逻辑关系。




- *索引存储*：在存储元素信息的同时，还建立附加 的索引表。索引表中的每项称为索引项，索引项 的一般形式是（关键字，地址）




- *散列存储*：根据元素的关键字直接计算出该元素 的存储地址，又称 *哈希（Hash）存储*



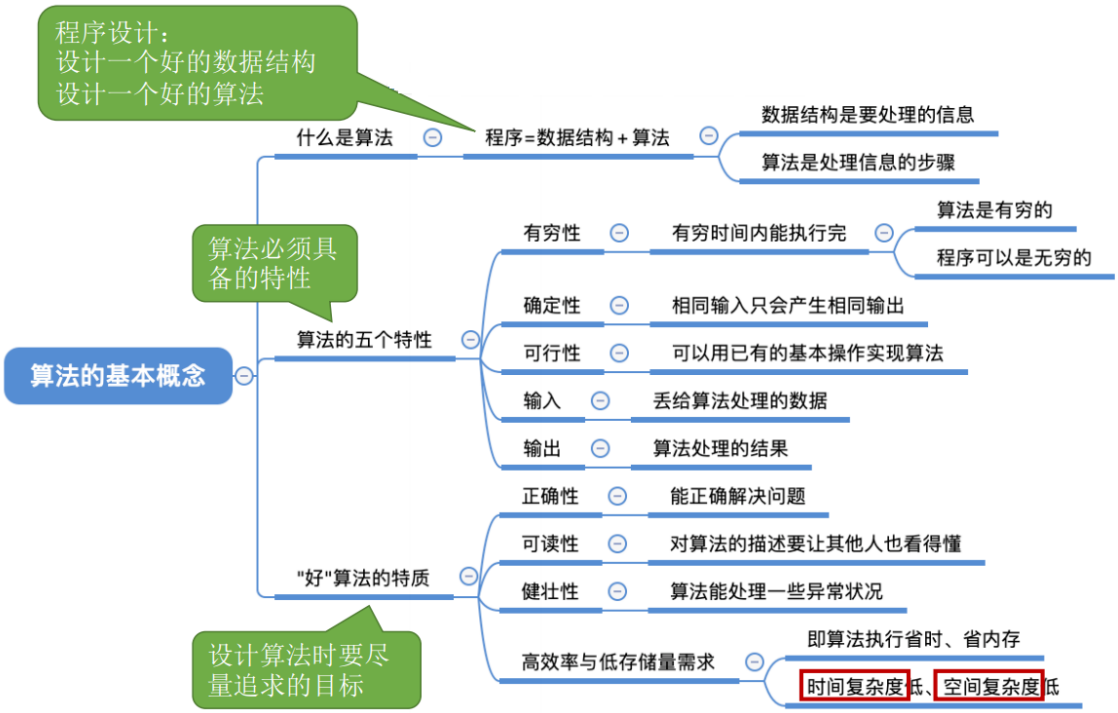
第六章，散列表


> **数据运算**：施加在数据上的运算包括运算的定义和实现。运算的定义是针对逻辑结构的，指出运算的功能；运算的实现是针对存储结构的，指出运算的具体操作步骤。

> **数据类型**：是一个值的集合和定义在此集合上的一组操作的总称。
>
> 1）原子类型。其值不可再分的数据类型。
>
> 2）结构类型。其值可以再分解为若干成分（分量）的数据类型。
>
> **抽象数据类型**：（Abstract Data Type，ADT）是抽象数据组织及与之相关的操作。

# 三、算法

## 1.基本概念



# 2.时间复杂度

> **算法时间复杂度**：事前预估算法时间开销T(n)与问题规模 n 的关系（T 表示 “time”）

a. *加法规则*：$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$

b. *乘法规则*：$T(n) = T_1(n) \times T_2(n) = O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

- 大小比较：

$$O(1) < O(log_2 n) < O(n) < O(nlog_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

例题1：

```
//算法1— 逐步递增型爱你
void loveYou(int n) {   //n 为问题规模
① int i=1;    //爱你的程度
② while(i<=n){
③     i++;    //每次+1
④     printf("I Love You %d\n", i);
  }
⑤ printf("I Love You More Than %d\n", n);
}
```

```
int main(){
  loveYou(3000);
}
```

```
I Love You 2994
I Love You 2995
I Love You 2996
I Love You 2997
I Love You 2998
I Love You 2999
I Love You 3000
I Love You 3001
I Love You More Than 300
```

语句频度：
①　——1次
②　——3001次
③④　——3000次
⑤　——1次

T(3000) = 1 + 3001 + 2*3000 + 1
时间开销与问题规模 n 的关系：
**T(n) = 3n+3 = O(n)**


问题1：是否可以忽略表达式某些部分？

只考虑阶数，用大O记法表示

问题2：如果有好几千行代码，按这种方法需要一行一行数？

只需考虑最深层循环的循环次数与 n 的关系

例题2：

```
//算法3— 指数递增型爱你
void loveYou(int n) {   //n 为问题规模
    int i=1;    //爱你的程度
    while(i<=n){
        i=i*2;  //每次翻倍
        printf("I Love You %d\n", i);
    }
    printf("I Love You More Than %d\n", n);
}
```

```
I Love You 32
I Love You 64
I Love You 128
I Love You 256
I Love You 512
I Love You 1024
I Love You 2048
I Love You 4096
I Love You More Than 3000
```

计算上述算法的时间复杂度 T(n)：
设最深层循环的语句频度（总共循环的次数）为 x，则
由循环条件可知，循环结束时刚好满足 $2^x > n$
$x = log_2 n + 1$

$T(n) = O(x) = O(log_2 n)$

例题3：

```c
//算法4—— 搜索数字型爱你
void loveYou(int flag[], int n) {   //n 为问题规模
    printf("I Am Iron Man\n");
    for(int i=0; i<n; i++){ //从第一个元素开始查找
        if(flag[i]==n){ //找到元素n
            printf("I Love You %d\n", n);
            break;  //找到后立即跳出循环
        }
    }
}
```

```c
//flag 数组中乱序存放了 1~n 这些数
int flag[n]={1...n};
loveYou(flag, n);
```

很多算法执行时间与输入的数据有关

计算上述算法的时间复杂度 T(n)

最好情况：元素n在第一个位置 ——最好时间复杂度 T(n)=O(1)
最坏情况：元素n在最后一个位置 ——最坏时间复杂度 T(n)=O(n)
平均情况：假设元素n在任意一个位置的概率相同为 $\frac{1}{n}$ ——平均时间复杂度 T(n)=O(n)

循环次数 $x = (1+2+3+...+n)\frac{1}{n} = (\frac{n(1+n)}{2})\frac{1}{n} = \frac{1+n}{2}$ T(n)=O(x)=O(n)

总结：


## 3.空间复杂度