

## Lesson3--顺序表和链表

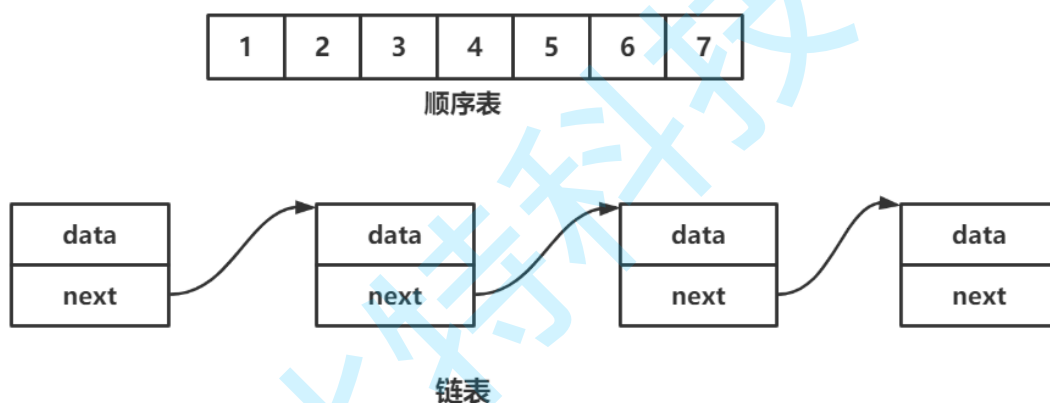
### 【本节目标】

- 1.线性表
- 2.顺序表
- 3.链表
- 4.顺序表和链表的区别和联系

### 1.线性表

线性表 (*linear list*) 是n个具有相同特性的数据元素的有限序列。线性表是一种在实际中广泛使用的数据结构，常见的线性表：顺序表、链表、栈、队列、字符串...

线性表在逻辑上是线性结构，也就说是连续的一条直线。但是在物理结构上并不一定是连续的，线性表在物理上存储时，通常以数组和链式结构的形式存储。



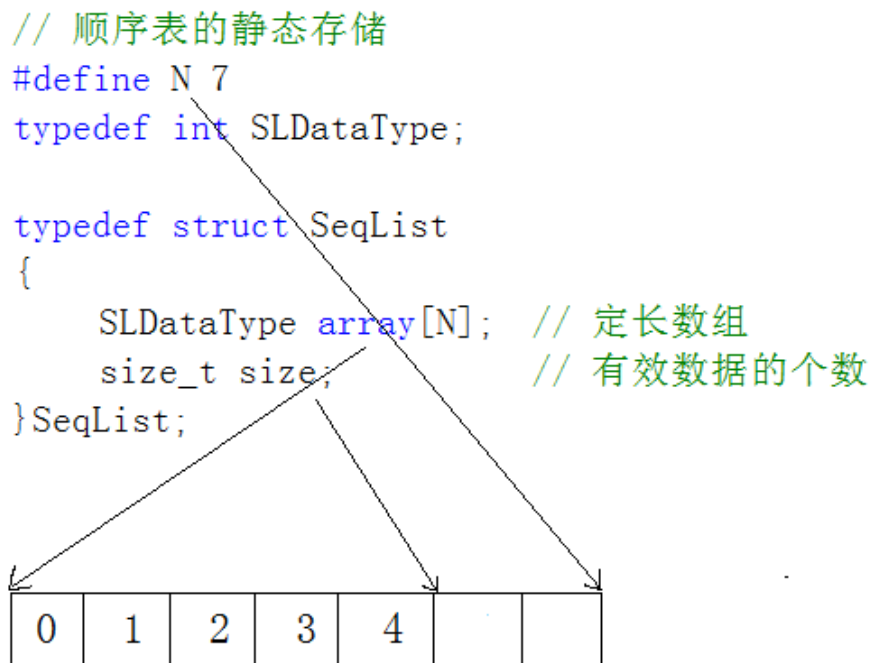
### 2.顺序表

#### 2.1概念及结构

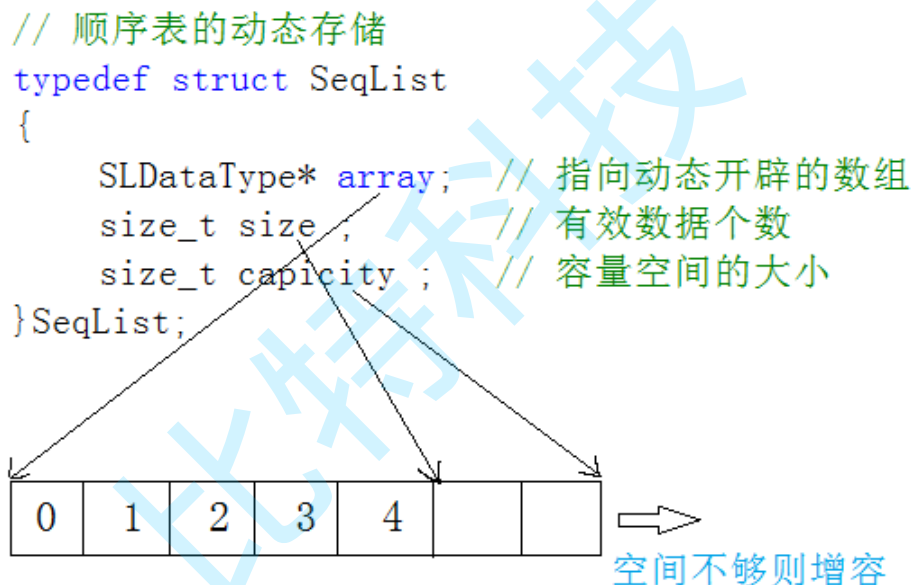
顺序表是用一段**物理地址连续**的存储单元依次存储数据元素的线性结构，一般情况下采用数组存储。在数组上完成数据的增删查改。

顺序表一般可以分为：

1. 静态顺序表：使用定长数组存储元素。



2. 动态顺序表：使用动态开辟的数组存储。



## 2.2 接口实现

静态顺序表只适用于确定知道需要存多少数据的场景。静态顺序表的定长数组导致N定大了，空间开多了浪费，开少了不够用。所以现实中基本都是使用动态顺序表，根据需要动态的分配空间大小，所以下面我们实现动态顺序表。

```
typedef int SLDataType;

// 顺序表的动态存储
typedef struct SeqList
{
    SLDataType* array; // 指向动态开辟的数组
    size_t size; // 有效数据个数
    size_t capacity; // 容量空间的大小
}SeqList;

// 基本增删查改接口
// 顺序表初始化
```

```

void SeqListInit(SeqList* ps1);
// 检查空间, 如果满了, 进行增容
void CheckCapacity(SeqList* ps1);
// 顺序表尾插
void SeqListPushBack(SeqList* ps1, SLDataType x);
// 顺序表尾删
void SeqListPopBack(SeqList* ps1);
// 顺序表头插
void SeqListPushFront(SeqList* ps1, SLDataType x);
// 顺序表头删
void SeqListPopFront(SeqList* ps1);
// 顺序表查找
int SeqListFind(SeqList* ps1, SLDataType x);
// 顺序表在pos位置插入x
void SeqListInsert(SeqList* ps1, size_t pos, SLDataType x);
// 顺序表删除pos位置的值
void SeqListErase(SeqList* ps1, size_t pos);
// 顺序表销毁
void SeqListDestory(SeqList* ps1);
// 顺序表打印
void SeqListPrint(SeqList* ps1);

```

## 2.3 数组相关面试题

1. 原地移除数组中所有的元素val, 要求时间复杂度为 $O(N)$ , 空间复杂度为 $O(1)$ 。[OJ链接](#)
2. 删除排序数组中的重复项。[OJ链接](#)
3. 合并两个有序数组。[OJ链接](#)

## 2.4 顺序表的问题及思考

问题:

1. 中间/头部的插入删除, 时间复杂度为 $O(N)$
2. 增容需要申请新空间, 拷贝数据, 释放旧空间。会有不小的消耗。
3. 增容一般是呈2倍的增长, 势必会有一定的空间浪费。例如当前容量为100, 满了以后增容到200, 我们再继续插入了5个数据, 后面没有数据插入了, 那么就浪费了95个数据空间。

思考: 如何解决以上问题呢? 下面给出了链表的结构来看看。

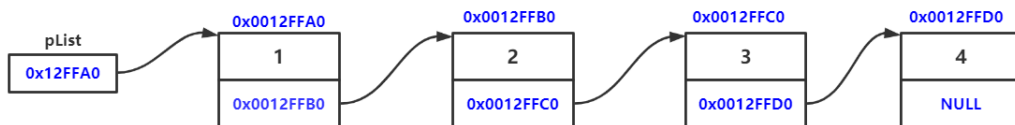
## 3.链表

### 3.1 链表的概念及结构

概念: 链表是一种物理存储结构上非连续、非顺序的存储结构, 数据元素的逻辑顺序是通过链表中的指针链接次序实现的。



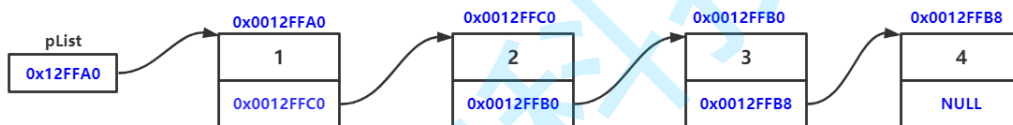
### 现实中 数据结构中



注意:

1. 从上图可看出，链式结构在逻辑上是连续的，但是在物理上不一定连续
2. 现实中的结点一般都是从堆上申请出来的
3. 从堆上申请的空间，是按照一定的策略来分配的，两次申请的空间可能连续，也可能不连续

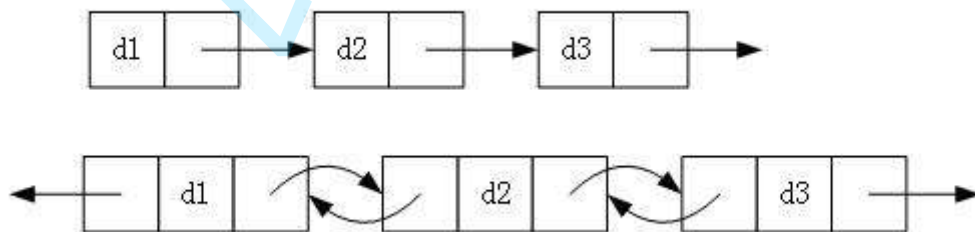
假设在32位系统上，结点中值域为int类型，则一个节点的大小为8个字节，则也可能有下述链表:



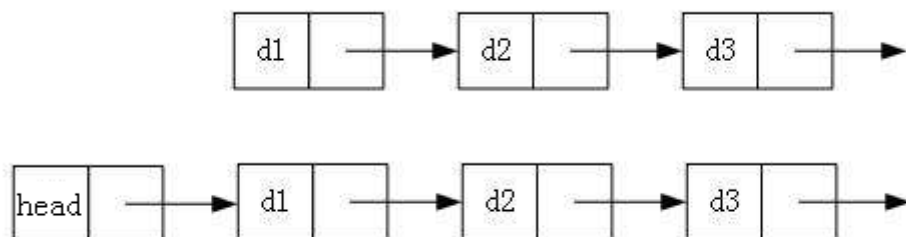
## 3.2 链表的分类

实际中链表的结构非常多样，以下情况组合起来就有8种链表结构:

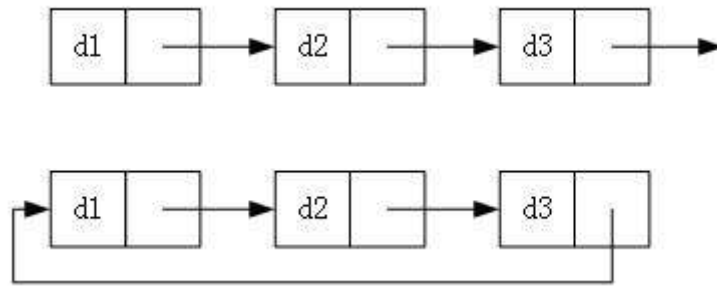
### 1. 单向或者双向



### 2. 带头或者不带头

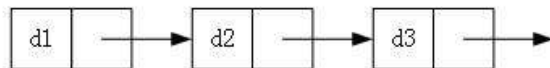


### 3. 循环或者非循环

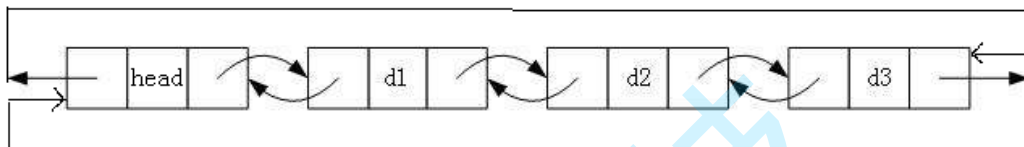


虽然有这么多的链表的结构，但是我们实际中最常用还是两种结构：

### 无头单向非循环链表



### 带头双向循环链表



1. 无头单向非循环链表：结构简单，一般不会单独用来存数据。实际中更多是作为其他数据结构的子结构，如哈希桶、图的邻接表等等。另外这种结构在笔试面试中出现很多。
2. 带头双向循环链表：结构最复杂，一般用在单独存储数据。实际中使用的链表数据结构，都是带头双向循环链表。另外这个结构虽然结构复杂，但是使用代码实现以后会发现结构会带来很多优势，实现反而简单了，后面我们代码实现了就知道了。

## 3.3 链表的实现

```
// 1、无头+单向+非循环链表增删查改实现
typedef int SLTDataType;
typedef struct SListNode
{
    SLTDataType data;
    struct SListNode* next;
}SListNode;

// 动态申请一个结点
SListNode* BuySListNode(SLTDataType x);
// 单链表打印
void SListPrint(SListNode* plist);
// 单链表尾插
void SListPushBack(SListNode** pplist, SLTDataType x);
// 单链表的头插
void SListPushFront(SListNode** pplist, SLTDataType x);
// 单链表的尾删
void SListPopBack(SListNode** pplist);
// 单链表头删
void SListPopFront(SListNode** pplist);
// 单链表查找
SListNode* SListFind(SListNode* plist, SLTDataType x);
// 单链表在pos位置之后插入x
// 分析思考为什么不在pos位置之前插入？
```

```
void SListInsertAfter(SListNode* pos, SLTDataType x);
// 单链表删除pos位置之后的值
// 分析思考为什么不删除pos位置?
void SListEraseAfter(SListNode* pos);
```

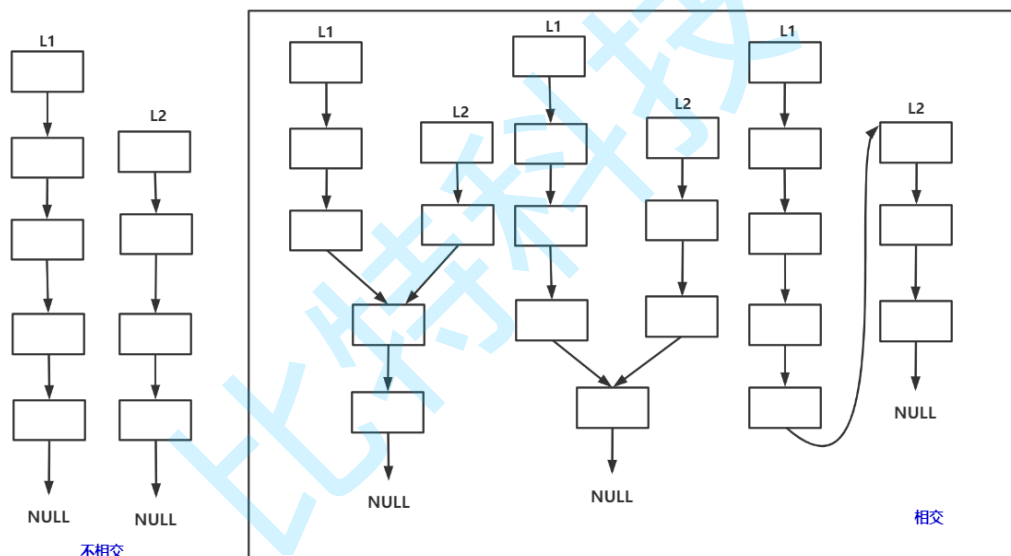
### 3.4 链表面试题

如果题目里没说，则默认不带哨兵位

1. 删除链表中等于给定值 **val** 的所有结点。 [OJ链接](#)
2. 反转一个单链表。 [OJ链接](#)
3. 给定一个带有头结点 head 的非空单链表，返回链表的中间结点。如果有两个中间结点，则返回第二个中间结点。 [OJ链接](#)
4. 输入一个链表，输出该链表中倒数第k个结点。 [OJ链接](#)
5. 将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有结点组成的。 [OJ链接](#)
6. 编写代码，以给定值x为基准将链表分割成两部分，所有小于x的结点排在大于或等于x的结点之前。 [OJ链接](#)
7. 链表的回文结构。 [OJ链接](#)
8. 输入两个链表，找出它们的第一个公共结点。 [OJ链接](#)

数组的回文判断也可以用左右两侧的快慢指针逼近后是否相遇来判断

一个结点只有一个next，因此两个链表不可能交叉



9. 给定一个链表，判断链表中是否有环。 [OJ链接](#)

#### 【思路】

快慢指针，即慢指针一次走一步，快指针一次走两步，两个指针从链表起始位置开始运行，如果链表

带环则一定会在环中相遇，否则快指针率先走到链表的末尾。比如：陪女朋友到操场跑步减肥。

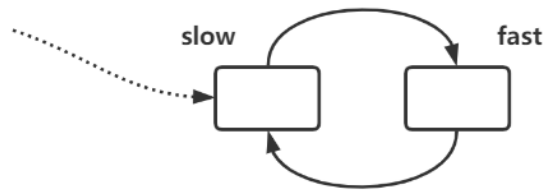
#### 【扩展问题】

- 为什么快指针每次走两步，慢指针走一步可以？

假设链表带环，两个指针最后都会进入环，快指针先进环，慢指针后进环。当慢指针刚进环时，可能就和快指针相遇了，最差情况下两个指针之间的距离刚好就是环的长度。此时，两个指针每移动一次，之间的距离就缩小一步，不会出现每次刚好是套圈的情况，因此：在满指针走到一圈之前，快指针肯定是可以追上慢指针的，即相遇。

- 快指针一次走3步，走4步，...n步行吗？

假设：快指针每次走3步，慢指针每次走一步，此时快指针肯定先进环，慢指针后来才进环。假设慢指针进环时候，快指针的位置如图所示：



此时按照上述方法来绕环移动，每次快指针走3步，慢指针走1步，是永远不会相遇的，快指针刚好将慢指针套圈了，因此不行。

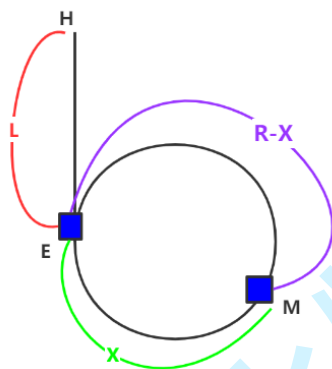
只有快指针走2步，慢指针走一步才可以，因为环的最小长度是1，即使套圈了两个也在相同的位置。

10. 给定一个链表，返回链表开始入环的第一个结点。如果链表无环，则返回 NULL [OJ链接](#)

#### • 结论

让一个指针从链表起始位置开始遍历链表，同时让一个指针从判环时相遇点的位置开始绕环运行，两个指针都是每次均走一步，最终肯定会在入口点的位置相遇。

#### • 证明



说明：

H为链表的起始点，E为环入口点，M为判环时相遇点

设：

环的长度为R，H到E的距离为L，E到M的距离为X

则：M到E的距离为R - X

在判环时，快慢指针相遇时所走的路径长度：

fast:  $L + X + nR$

slow:  $L + X$

注意：

1. 当慢指针进入环时，快指针可能已经在环中绕了n圈了，n至少为1  
因为：快指针先进环走到M的位置，最后又在M的位置与慢指针相遇

2. 慢指针进环之后，快指针肯定会在慢指针走一圈之内追上慢指针  
因为：慢指针进环后，快慢指针之间的距离最多就是环的长度，而两个指针在移动时，每次它们至今的距离都缩减一步，因此在慢指针移动一圈之前快指针肯定是可以追上慢指针的

而快指针速度是慢指针的两倍，因此有如下关系是：

$$2 * (L + X) = L + X + nR$$

$$L + X = nR$$

$$L = nR - X \quad (n \text{ 为 } 1, 2, 3, 4, \dots, n \text{ 的大小取决于环的大小，环越小 } n \text{ 越大})$$

极端情况下，假设  $n = 1$ ，此时：  $L = R - X$

即：一个指针从链表起始位置运行，一个指针从相遇点位置绕环，每次都走一步，两个指针最终会在入口点的位置相遇

11. 给定一个链表，每个结点包含一个额外增加的随机指针，该指针可以指向链表中的任何结点或空结点。

要求返回这个链表的深度拷贝。 [OJ链接](#)

12. 其他。ps：链表的题当前因为难度及知识面等等原因还不适合我们当前学习，以后大家自己下去以后 [Leetcode OJ链接](#) + [生客 OJ链接](#)

### 3.5 双向链表的实现

```
// 2、带头+双向+循环链表增删查改实现
typedef int LTDataType;
typedef struct ListNode
{
    LTDataType _data;
    struct ListNode* next;
```

```

    struct ListNode* prev;
}ListNode;

// 创建返回链表的头结点.
ListNode* ListCreate();
// 双向链表销毁
void ListDestory(ListNode* plist);
// 双向链表打印
void ListPrint(ListNode* plist);
// 双向链表尾插
void ListPushBack(ListNode* plist, LTDataType x);
// 双向链表尾删
void ListPopBack(ListNode* plist);
// 双向链表头插
void ListPushFront(ListNode* plist, LTDataType x);
// 双向链表头删
void ListPopFront(ListNode* plist);
// 双向链表查找
ListNode* ListFind(ListNode* plist, LTDataType x);
// 双向链表在pos的前面进行插入
void ListInsert(ListNode* pos, LTDataType x);
// 双向链表删除pos位置的结点
void ListErase(ListNode* pos);

```

#### 4.顺序表和链表的区别

不同点	顺序表	链表
存储空间上	物理上一定连续	逻辑上连续，但物理上不一定连续
随机访问	支持O(1)	不支持：O(N)
任意位置插入或者删除元素	可能需要搬移元素，效率低O(N)	只需修改指针指向
插入	动态顺序表，空间不够时需要扩容	没有容量的概念
应用场景	元素高效存储+频繁访问	任意位置插入和删除频繁
缓存利用率	高	低

备注：缓存利用率参考存储体系结构 以及 局部原理性。



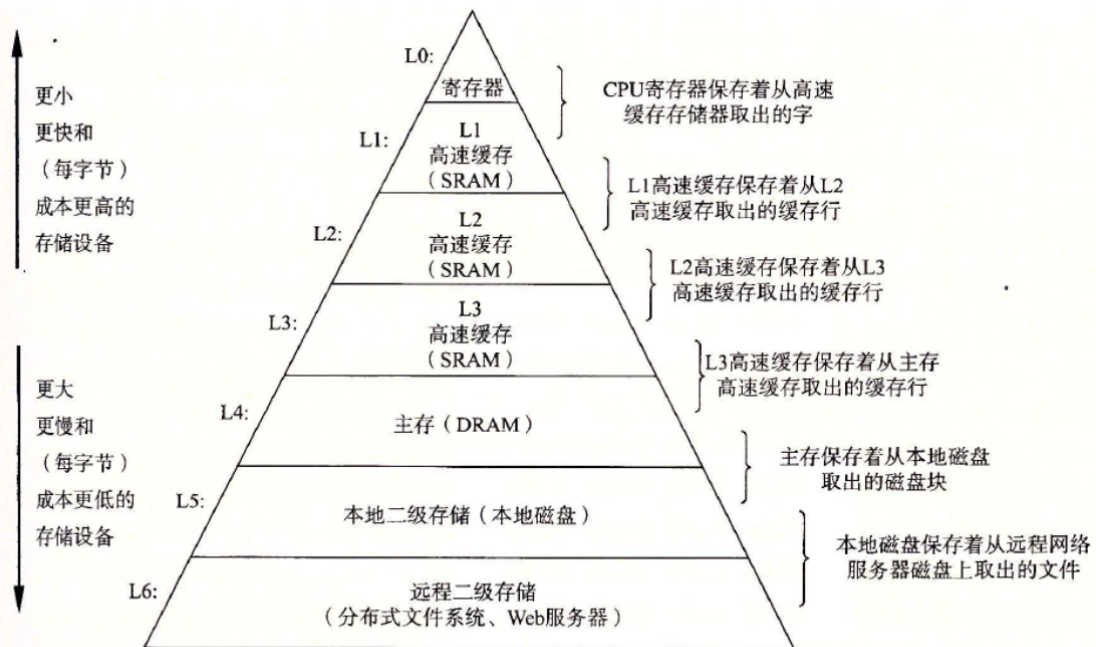
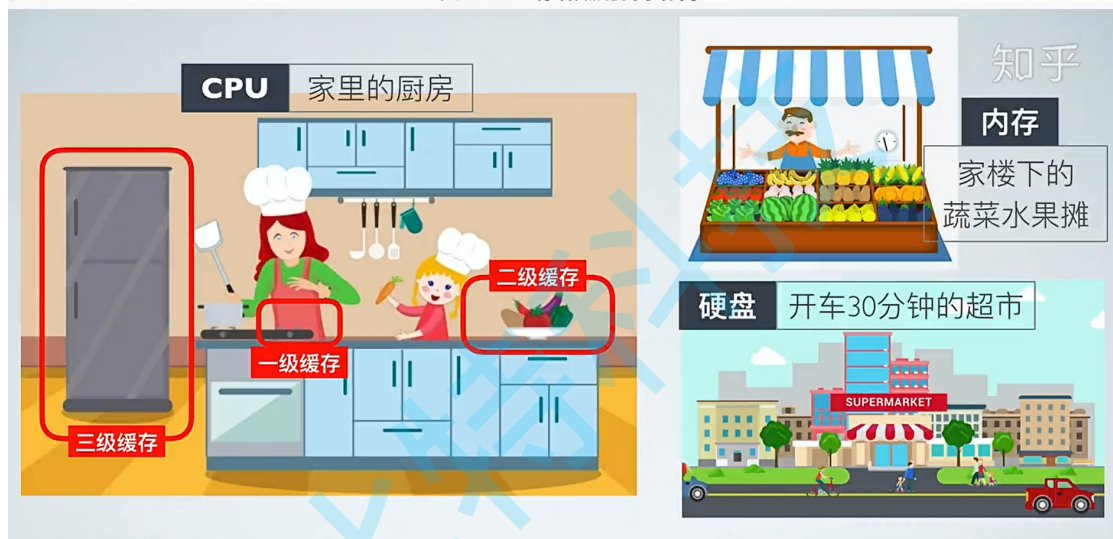


图 6-21 存储器层次结构



[与程序员相关的CPU缓存知识](#)