第3章

列表

上一章介绍的向量结构中，各数据项的物理存放位置与逻辑次序完全对应，故可通过秩直接访问对应的元素，此即所谓“循秩访问”（call-by-rank）。这种访问方式，如同根据具体的城市名、街道名和门牌号，直接找到某人。本章将要介绍的列表，与向量同属序列结构的范畴，其中的元素也构成一个线性逻辑次序；但与向量极为不同的是，元素的物理地址可以任意。

为保证对列表元素访问的可行性，逻辑上互为前驱和后继的元素之间，应维护某种索引关系。这种索引关系，可抽象地理解为被索引元素的位置（position），故列表元素是“循位置访问”（call-by-position）的；也可形象地理解为通往被索引元素的链接（link），故亦称作“循链接访问”（call-by-link）。这种访问方式，如同通过你的某位亲朋，找到他/她的亲朋、亲朋的亲朋、...。注意，向量中的秩同时对应于逻辑和物理次序，而位置仅对应于逻辑次序。

本章的讲解，将围绕列表结构的高效实现逐步展开，包括其ADT接口规范以及对应的算法。此外还将针对有序列表，系统地介绍排序等经典算法，并就其性能做一分析和对比。

§3.1 从向量到列表

不同数据结构内部的存储与组织方式各异，其操作接口的使用方式及时空性能也不尽相同。在设计或选用数据结构时，应从实际应用的需求出发，先确定功能规范及性能指标。比如，引入列表结构的目的，就在于弥补向量结构在解决某些应用问题时，在功能及性能方面的不足。二者之间的差异，表面上体现于对外的操作方式，但根源则在于其内部存储方式的不同。

3.1.1 从静态到动态

数据结构支持的操作，通常无非静态和动态两类：前者仅从中获取信息，后者则会修改数据结构的局部甚至整体。以第2章基于数组实现的向量结构为例，其size()和get()等静态操作均可在常数时间内完成，而insert()和remove()等动态操作却都可能需要线性时间。究其原因，在于“各元素物理地址连续”的约定——此即所谓的“静态存储”策略。

得益于这种策略，可在$\mathcal{O}(1)$时间内由秩确定向量元素的物理地址；但反过来，在添加（删除）元素之前（之后），又不得不移动$\mathcal{O}(n)$个后继元素。可见，尽管如此可使静态操作的效率达到极致，但就动态操作而言，局部的修改可能引起大范围甚至整个数据结构的调整。

列表（list）结构尽管也要求各元素在逻辑上具有线性次序，但对其物理地址却未作任何限制——此即所谓“动态存储”策略。具体地，在其生命期内，此类数据结构将随着内部数据的需要，相应地分配或回收局部的数据空间。如此，元素之间的逻辑关系得以延续，却不必与其物理次序相关。作为补偿，此类结构将通过指针或引用等机制，来确定各元素的实际物理地址。

例如，链表（linked list）就是一种典型的动态存储结构。其中的数据，分散为一系列称作节点（node）的单位，节点之间通过指针相互索引和访问。为了引入新节点或删除原有节点，只需在局部，调整少量相关节点之间的指针。这就意味着，采用动态存储策略，至少可以大大降低动态操作的成本。

3.1.2 由秩到位置

改用以上动态存储策略之后，在提高动态操作效率的同时，却又不得不舍弃原静态存储策略中循秩访问的方式，从而造成静态操作性能的下降。

以采用动态存储策略的线性结构（比如链表）为例。尽管按照逻辑次序，每个数据元素依然具有秩这一指标，但为了访问秩为r的元素，我们只能顺着相邻元素之间的指针，从某一端出发逐个扫描各元素，经过r步迭代后才能确定该元素的物理存储位置。这意味着，原先只需$O(1)$时间的静态操作，此时的复杂度也将线性正比于被访问元素的秩，在最坏情况下等于元素总数n；即便在各元素被访问概率相等的情况下，平均而言也需要$O(n)$时间。

对数据结构的访问方式，应与其存储策略相一致。此时，既然继续延用循秩访问的方式已非上策，就应更多地习惯于通过位置，来指代并访问动态存储结构中的数据元素。与向量中秩的地位与功能类似，列表中的位置也是指代各数据元素的一个标识性指标，借助它可以便捷地（比如在常数时间内）得到元素的物理存储地址。各元素的位置，通常可表示和实现为联接于元素之间的指针或引用。因此，基于此类结构设计算法时，应更多地借助逻辑上相邻元素之间的位置索引，以实现对目标元素的快速定位和访问，并进而提高算法的整体效率。

3.1.3 列表

与向量一样，列表也是由具有线性逻辑次序的一组元素构成的集合：

$$L = \{ a_0, a_1, \ldots, a_{n-1} \}$$

列表是链表结构的一般化推广，其中的元素称作节点（node），分别由特定的位置或链接指代。与向量一样，在元素之间，也可定义前驱、直接前驱，以及后继、直接后继等关系；相对于任意元素，也有定义对应的前缀、后缀等子集。

§3.2 接口

如上所述，作为列表的基本组成单位，列表节点除需保存对应的数据项，还应记录其前驱和后继的位置，故需将这些信息及相关操作组成列表节点对象，然后参与列表的构建。

本节将给出列表节点类与列表类的接口模板类描述，稍后逐一讲解各接口的具体实现。

3.2.1 列表节点

■ ADT接口

作为一种抽象数据类型，列表节点对象应支持以下操作接口。

表3.1 列表节点ADT支持的操作接口

操 作 接 口	功 能
data()	当前节点所存数据对象
pred()	当前节点前驱节点的位置
succ()	当前节点后继节点的位置
insertAsPred(e)	插入前驱节点，存入被引用对象e，返回新节点位置
insertAsSucc(e)	插入后继节点，存入被引用对象e，返回新节点位置

■ ListNode模板类

按照表3.1所定义的ADT接口，可定义列表节点模板类如代码3.1所示。出于简洁与效率的考虑，这里并未对ListNode对象做封装处理。列表节点数据项的类型，通过模板参数T指定。

```
1  typedef int Rank; //秩
2  #define ListNodePosi(T) ListNode<T>* //列表节点位置
3
4  template <typename T> struct ListNode { //列表节点模板类（以双向链表形式实现）
5  // 成员
6     T data; ListNodePosi(T) pred; ListNodePosi(T) succ; //数值、前驱、后继
7  // 构造函数
8     ListNode() {} //针对header和trailer的构造
9     ListNode ( T e, ListNodePosi(T) p = NULL, ListNodePosi(T) s = NULL )
10       : data ( e ), pred ( p ), succ ( s ) {} //默认构造器
11 // 操作接口
12    ListNodePosi(T) insertAsPred ( T const& e ); //紧靠当前节点之前插入新节点
13    ListNodePosi(T) insertAsSucc ( T const& e ); //紧随当前节点之后插入新节点
14 };
```

代码3.1 列表节点模板类[①]

每个节点都存有数据对象data。为保证叙述简洁，在不致歧义的前提下，本书将不再区分节点及其对应的data对象。此外，每个节点还设有指针pred和succ，分别指向其前驱和后继。

为了创建一个列表节点对象，只需根据所提供的参数，分别设置节点内部的各个变量。其中前驱、后继节点的位置指针若未予指定，则默认取作NULL。

3.2.2 列表

■ ADT接口

作为一种抽象数据类型，列表对象应支持以下操作接口。

表3.2 列表ADT支持的操作接口

操 作 接 口	功 能	适 用 对 象
size()	报告列表当前的规模（节点总数）	列表
first()、last()	返回首、末节点的位置	列表
insertAsFirst(e) insertAsLast(e)	将e当作首、末节点插入	列表
insertA(p, e) insertB(p, e)	将e当作节点p的直接后继、前驱插入	列表

① 请注意，这里所“定义”的ListNodePosi(T)并非真正意义上“列表节点位置”类型。
巧合的是，就在本书第1版即将付印之际，C++.0x标准终于被ISO接纳。
新标准所拓展的特性之一，就是对模板别名（template alias）等语法形式的支持。因此可以期望在不久的将来，C++编译器将能够支持如下更为直接和简明的描述和实现：

```
template <typename T> typedef ListNode<T>* ListNodePosi;
```

操 作 接 口	功 能	适 用 对 象
remove(p)	删除位置p处的节点，返回其数值	列表
disordered()	判断所有节点是否已按非降序排列	列表
sort()	调整各节点的位置，使之按非降序排列	列表
find(e)	查找目标元素e，失败时返回NULL	列表
search(e)	查找目标元素e，返回不大于e且秩最大的节点	有序列表
deduplicate()	剔除重复节点	列表
uniquify()	剔除重复节点	有序列表
traverse()	遍历并统一处理所有节点，处理方法由函数对象指定	列表

请留意用以指示插入和删除操作位置的节点p。这里约定，它或者在此前经查找已经确定，或者从此前的其它操作返回或沿用。这些也是列表类结构的典型操作方式。

这里也设置一个disordered()接口，以判断列表是否已经有序。同时，也分别针对有序和无序列表，提供了去重操作的两个版本（deduplicate()和uniquify()），以及查找操作的两个版本（find()和search()）。与向量一样，有序列表的唯一化，比无序列表效率更高。然而正如我们将要看到的，由于只能通过位置指针以局部移动的方式访问节点，尽管有序列表中节点在逻辑上始终按照大小次序排列，其查找操作的效率并没有实质改进（习题[3-1]）。

List模板类

按照表3.2定义的ADT接口，可定义List模板类如下。

```
#include "listNode.h" //引入列表节点类

template <typename T> class List { //列表模板类

private:
   int _size; ListNodePosi(T) header; ListNodePosi(T) trailer; //规模、头哨兵、尾哨兵

protected:
   void init(); //列表创建时的初始化
   int clear(); //清除所有节点
   void copyNodes ( ListNodePosi(T), int ); //复制列表中自位置p起的n项
   void merge ( ListNodePosi(T)&, int, List<T>&, ListNodePosi(T), int ); //归并
   void mergeSort ( ListNodePosi(T)&, int ); //对从p开始连续的n个节点归并排序
   void selectionSort ( ListNodePosi(T), int ); //对从p开始连续的n个节点选择排序
   void insertionSort ( ListNodePosi(T), int ); //对从p开始连续的n个节点插入排序

public:
// 构造函数
   List() { init(); } //默认
   List ( List<T> const& L ); //整体复制列表L
   List ( List<T> const& L, Rank r, int n ); //复制列表L中自第r项起的n项
   List ( ListNodePosi(T) p, int n ); //复制列表中自位置p起的n项
```

```
23 // 析构函数
24    ~List(); //释放（包含头、尾哨兵在内的）所有节点
25 // 只读访问接口
26    Rank size() const { return _size; } //规模
27    bool empty() const { return _size <= 0; } //判空
28    T& operator[] ( Rank r ) const; //重载，支持循秩访问（效率低）
29    ListNodePosi(T) first() const { return header->succ; } //首节点位置
30    ListNodePosi(T) last() const { return trailer->pred; } //末节点位置
31    bool valid ( ListNodePosi(T) p ) //判断位置p是否对外合法
32    { return p && ( trailer != p ) && ( header != p ); } //将头、尾节点等同于NULL
33    int disordered() const; //判断列表是否已排序
34    ListNodePosi(T) find ( T const& e ) const //无序列表查找
35    { return find ( e, _size, trailer ); }
36    ListNodePosi(T) find ( T const& e, int n, ListNodePosi(T) p ) const; //无序区间查找
37    ListNodePosi(T) search ( T const& e ) const //有序列表查找
38    { return search ( e, _size, trailer ); }
39    ListNodePosi(T) search ( T const& e, int n, ListNodePosi(T) p ) const; //有序区间查找
40    ListNodePosi(T) selectMax ( ListNodePosi(T) p, int n ); //在p及其n-1个后继中选出最大者
41    ListNodePosi(T) selectMax() { return selectMax ( header->succ, _size ); } //整体最大者
42 // 可写访问接口
43    ListNodePosi(T) insertAsFirst ( T const& e ); //将e当作首节点插入
44    ListNodePosi(T) insertAsLast ( T const& e ); //将e当作末节点插入
45    ListNodePosi(T) insertA ( ListNodePosi(T) p, T const& e ); //将e当作p的后继插入
46    ListNodePosi(T) insertB ( ListNodePosi(T) p, T const& e ); //将e当作p的前驱插入
47    T remove ( ListNodePosi(T) p ); //删除合法位置p处的节点,返回被删除节点
48    void merge ( List<T>& L ) { merge ( first(), size, L, L.first(), L._size ); } //全列表归并
49    void sort ( ListNodePosi(T) p, int n ); //列表区间排序
50    void sort() { sort ( first(), _size ); } //列表整体排序
51    int deduplicate(); //无序去重
52    int uniquify(); //有序去重
53    void reverse(); //前后倒置（习题）
54 // 遍历
55    void traverse ( void (* ) ( T& ) ); //遍历，依次实施visit操作（函数指针，只读或局部性修改）
56    template <typename VST> //操作器
57    void traverse ( VST& ); //遍历，依次实施visit操作（函数对象，可全局性修改）
58 }; //List
```

代码3.2 列表模板类

由代码3.2可见，列表结构的实现方式与第2章的向量结构颇为相似：通过模板参数T指定列表元素的类型（同时亦为代码3.1中列表节点数据项的类型）；在内部设置私有变量以记录当前规模等状态信息；基于多种排序算法提供统一的sort()接口，以将列表转化为有序列表。

以下，分别介绍列表的内部结构、基本接口，以及主要算法的具体实现。

§3.3 列表

3.3.1 头、尾节点

List对象的内部组成及逻辑结构如图3.1所示，其中私有的头节点（header）和尾节点（trailer）始终存在，但对外并不可见。对外部可见的数据节点如果存在，则其中的第一个和最后一个节点分别称作首节点（first node）和末节点（last node）。

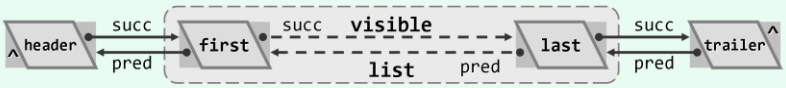

图3.1 首（末）节点是头（尾）节点的直接后继（前驱）

就内部结构而言，头节点紧邻于首节点之前，尾节点紧邻于末节点之后。这类经封装之后从外部不可见的节点，称作哨兵节点（sentinel node）。由代码3.2中List::valid()关于合法节点位置的判别准则可见，此处的两个哨兵节点从外部被等效地视作NULL。

设置哨兵节点之后，对于从外部可见的任一节点而言，其前驱和后继在列表内部都必然存在，故可简化算法的描述与实现。比如，在代码3.2中为实现first()和last()操作，只需直接返回header->succ或trailer->pred。此外更重要地，哨兵节点的引入，也使得相关算法不必再对各种边界退化情况做专门的处理，从而避免出错的可能，我们稍后将对此有更实际的体会。

尽管哨兵节点也需占用一定的空间，但只不过是常数规模，其成本远远低于由此带来的便利。

3.3.2 默认构造方法

创建List对象时，默认构造方法将调用如代码3.3所示的统一初始化过程init()，在列表内部创建一对头、尾哨兵节点，并适当地设置其前驱、后继指针构成一个双向链表。

```
1 template <typename T> void List<T>::init() { //列表初始化，在创建列表对象时统一调用
2    header = new ListNode<T>; //创建头哨兵节点
3    trailer = new ListNode<T>; //创建尾哨兵节点
4    header->succ = trailer; header->pred = NULL;
5    trailer->pred = header; trailer->succ = NULL;
6    _size = 0; //记录规模
7 }
```

代码3.3 列表类内部方法init()

如图3.2所示，该链表对外的有效部分初始为空，哨兵节点对外不可见，此后引入的新节点都将陆续插入于这一对哨兵节点之间。

图3.2 刚创建的List对象

在列表的其它构造方法中，内部变量的初始化过程与此相同，因此都可统一调用init()过程。该过程仅涉及常数次基本操作，共需运行常数时间。

3.3.3 由秩到位置的转换

鉴于偶尔可能需要通过秩来指定列表节点，可通过重载操作符“[]”，提供一个转换接口。

```
1 template <typename T> //重载下标操作符，以通过秩直接访问列表节点（虽方便，效率低，需慎用）
2 T& List<T>::operator[] ( Rank r ) const { //assert: 0 <= r < size
3    ListNodePosi(T) p = first(); //从首节点出发
4    while ( 0 < r-- ) p = p->succ; //顺数第r个节点即是
5    return p->data; //目标节点，返回其中所存元素
6 }
```

代码3.4 重载列表类的下标操作符

具体地如代码3.4所示，为将任意指定的秩r转换为列表中对应的元素，可从首节点出发，顺着后继指针前进r步。只要秩r合法，该算法的正确性即一目了然。其中每步迭代仅需常数时间，故该算法的总体运行时间应为$\mathcal{O}(r + 1)$，线性正比于目标节点的秩。

相对于向量同类接口的$\mathcal{O}(1)$复杂度，列表的这一效率十分低下——其根源在于，列表元素的存储和访问方式已与向量截然不同。诚然，当r大于n/2时，从trailer出发沿pred指针逆行查找，可以在一定程度上减少迭代次数，但就总体的平均效率而言，这一改进并无实质意义。

3.3.4 查找

■ 实现

在代码3.2中，列表ADT针对整体和区间查找，重载了操作接口find(e)和find(e, p, n)。其中，前者作为特例，可以直接调用后者。因此，只需如代码3.5所示，实现后一接口。

```
1 template <typename T> //在无序列表内节点p（可能是trailer）的n个（真）前驱中，找到等于e的最后者
2 ListNodePosi(T) List<T>::find ( T const& e, int n, ListNodePosi(T) p ) const {
3    while ( 0 < n-- ) //（0 <= n <= rank(p) < _size）对于p的最近的n个前驱，从右向左
4       if ( e == ( p = p->pred )->data ) return p; //逐个比对，直至命中或范围越界
5    return NULL; //p越出左边界意味着区间内不含e，查找失败
6 } //失败时，返回NULL
```

代码3.5 无序列表元素查找接口find()

■ 复杂度

以上算法的思路及过程，与无序向量的顺序查找算法Vector::find()（代码2.10）相仿，故时间复杂度也应是$\mathcal{O}(n)$，线性正比于查找区间的宽度。

3.3.5 插入

■ 接口

为将节点插至列表，可视具体要求的不同，在代码3.6所提供的多种接口中灵活选用。

```
1 template <typename T> ListNodePosi(T) List<T>::insertAsFirst ( T const& e )
2 {  _size++; return header->insertAsSucc ( e );  } //e当作首节点插入
3
```

```
4  template <typename T> ListNodePosi(T) List<T>::insertAsLast ( T const& e )
5  {  _size++; return trailer->insertAsPred ( e );  } //e当作末节点插入
6
7  template <typename T> ListNodePosi(T) List<T>::insertA ( ListNodePosi(T) p, T const& e )
8  {  _size++; return p->insertAsSucc ( e );  } //e当作p的后继插入（After）
9
10 template <typename T> ListNodePosi(T) List<T>::insertB ( ListNodePosi(T) p, T const& e )
11 {  _size++; return p->insertAsPred ( e );  } //e当作p的前驱插入（Before）
```

代码3.6 列表节点插入接口

可见，这些接口的实现，都可转化为列表节点对象的前插入或后插入接口。

■ 前插入

将新元素e作为当前节点的前驱插至列表的过程，可描述和实现如代码3.7所示。

```
1  template <typename T> //将e紧靠当前节点之前插入于当前节点所属列表（设有哨兵头节点header）
2  ListNodePosi(T) ListNode<T>::insertAsPred ( T const& e ) {
3     ListNodePosi(T) x = new ListNode ( e, pred, this ); //创建新节点
4     pred->succ = x; pred = x; //设置正向链接
5     return x; //返回新节点的位置
6  }
```

代码3.7 ListNode::insertAsPred()算法

图3.3给出了整个操作的具体过程。插入新节点之前，列表局部的当前节点及其前驱如图(a)所示。该算法首先如图(b)所示创建新节点new，构造函数同时将其数据项置为e，并令其后继链接succ指向当前节点，令其前驱链接pred指向当前节点的前驱节点。随后如图(c)所示，使new成为当前节点前驱节点的后继，使new成为当前节点的前驱（次序不能颠倒）。最终如图(d)所示，经过如此调整，新节点即被顺利地插至列表的这一局部。

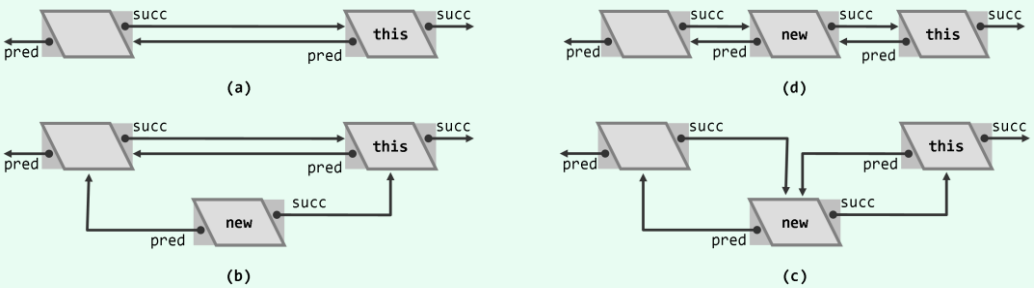

图3.3 ListNode::insertAsPred()算法

请注意，列表规模记录的更新由代码3.6中的上层调用者负责。另外，得益于头哨兵节点的存在，即便当前节点为列表的首节点，其前驱也如图(a)所示必然存在，故不必另做特殊处理。当然，在当前节点即首节点时，前插入接口等效于List::insertAsFirst()。

■ 后插入

将新元素e作为当前节点的后继插至列表的过程，可描述和实现如代码3.8所示。

```
1 template <typename T> //将e紧随当前节点之后插入于当前节点所属列表（设有哨兵尾节点trailer）
2 ListNodePosi(T) ListNode<T>::insertAsSucc ( T const& e ) {
3    ListNodePosi(T) x = new ListNode ( e, this, succ ); //创建新节点
4    succ->pred = x; succ = x; //设置逆向链接
5    return x; //返回新节点的位置
6 }
```

代码3.8 ListNode::insertAsSucc()算法

后插入的操作过程以及最终效果与前插入完全对称，不再赘述。

■ **复杂度**

上述两种插入操作过程，仅涉及局部的两个原有节点和一个新节点，且不含任何迭代或递归。若假设当前节点已经定位，不计入此前的查找所消耗的时间，则它们都可在常数时间内完成。

3.3.6 基于复制的构造

与向量一样，列表的内部结构也是动态创建的，故利用默认的构造方法并不能真正地完成新列表的复制创建。为此，需要专门编写相应的构造方法，通过复制某一已有列表来构造新列表。

■ **copyNodes()**

尽管这里提供了多种形式，以允许对原列表的整体或局部复制，但其实质过程均大同小异，都可概括和转化为如代码3.9所示的底层内部方法copyNodes()。在输入参数合法的前提下，copyNodes()首先调用init()方法，创建头、尾哨兵节点并做相应的初始化处理，然后自p所指节点起，从原列表中取出n个相邻的节点，并逐一作为末节点插至新列表中。

```
1 template <typename T> //列表内部方法：复制列表中自位置p起的n项
2 void List<T>::copyNodes ( ListNodePosi(T) p, int n ) { //p合法，且至少有n-1个真后继节点
3    init(); //创建头、尾哨兵节点并做初始化
4    while ( n-- ) { insertAsLast ( p->data ); p = p->succ; } //将起自p的n项依次作为末节点插入
5 }
```

代码3.9 列表类内部方法copyNodes()

根据此前的分析，init()操作以及各步迭代中的插入操作均只需常数时间，故copyNodes()过程总体的运行时间应为$\mathcal{O}(n + 1)$，线性正比于待复制列表区间的长度n。

■ **基于复制的构造**

如代码3.10所示，基于上述copyNodes()方法可以实现多种接口，通过复制已有列表的区间或整体，构造出新列表。其中，为了复制列表L中自秩r起的n个相邻节点，List(L, r, n)需借助重载后的下标操作符，找到待复制区间起始节点的位置，然后再以此节点作为参数调用copyNodes()。根据3.3.3节的分析结论，需要花费$\mathcal{O}(r + 1)$的时间才能将r转换为起始节点的位置，故该复制接口的总体复杂度应为$\mathcal{O}(r + n + 1)$，线性正比于被复制节点的最高秩。由此也可再次看出，在诸如列表之类采用动态存储策略的结构中，循秩访问远非有效的方式。

```
1 template <typename T> //复制列表中自位置p起的n项（assert: p为合法位置，且至少有n-1个后继节点）
2 List<T>::List ( ListNodePosi(T) p, int n ) { copyNodes ( p, n ); }
```

```
3
4 template <typename T> //整体复制列表L
5 List<T>::List ( List<T> const& L ) { copyNodes ( L.first(), L._size ); }
6
7 template <typename T> //复制L中自第r项起的n项（assert: r+n <= L._size）
8 List<T>::List ( List<T> const& L, int r, int n ) { copyNodes ( L[r], n ); }
```

代码3.10 基于复制的列表构造方法

3.3.7 删除

■ **实现**

在列表中删除指定节点p的算法，可以描述并实现如代码3.11所示。

```
1 template <typename T> T List<T>::remove ( ListNodePosi(T) p ) { //删除合法节点p，返回其数值
2    T e = p->data; //备份待删除节点的数值（假定T类型可直接赋值）
3    p->pred->succ = p->succ; p->succ->pred = p->pred; //后继、前驱
4    delete p; _size--; //释放节点，更新规模
5    return e; //返回备份的数值
6 }
```

代码3.11 列表节点删除接口remove()

图3.4给出了整个操作的具体过程。删除节点之前，列表在位置p附近的局部如图(a)所示。为了删除位置p处的节点，首先如图(b)所示，令其前驱节点与后继节点相互链接。然后如图(c)所示，释放掉已经孤立出来的节点p，同时相应地更新列表规模计数器_size。最终如图(d)所示，经过如此调整之后，原节点p即被顺利地从列表中摘除。

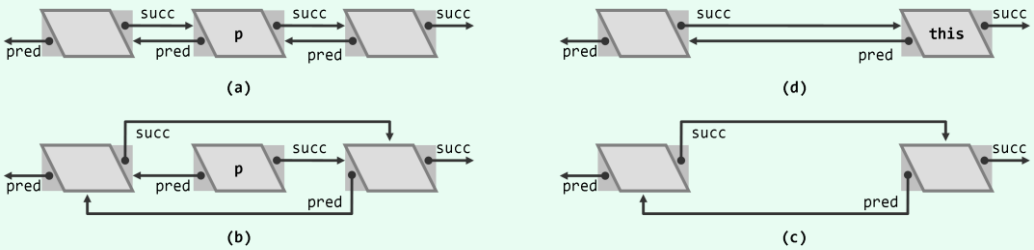

图3.4 List::remove()算法

这里，可以清晰地体会到哨兵节点的作用。不难验证，即便p所指的是列表中唯一对外有效的节点（其前驱和后继都是哨兵节点），remove()算法依然可以正常运转。

■ **复杂度**

以上过程仅涉及常数次基本操作，故若不计入此前为查找并确定位置p所消耗的时间，列表的节点删除操作可在常数时间内完成。

3.3.8 析构

■ 释放资源及清除节点

与所有对象一样，列表对象析构时也需如代码3.12所示，将其所占用的资源归还操作系统。

```
1 template <typename T> List<T>::~List() //列表析构器
2 { clear(); delete header; delete trailer; } //清空列表，释放头、尾哨兵节点
```

代码3.12 列表析构方法

可见，列表的析构需首先调用clear()接口删除并释放所有对外部有效的节点，然后释放内部的头、尾哨兵节点。而clear()过程则可描述和实现如代码3.13所示。

```
1 template <typename T> int List<T>::clear() { //清空列表
2    int oldSize = _size;
3    while ( 0 < _size ) remove ( header->succ ); //反复删除首节点，直至列表变空
4    return oldSize;
5 }
```

代码3.13 列表清空方法clear()

■ 复杂度

这里的时间消耗主要来自clear()操作，该操作通过remove()接口反复删除列表的首节点。因此，clear()方法以及整个析构方法的运行时间应为$\mathcal{O}(n)$，线性正比于列表原先的规模。

3.3.9 唯一化

■ 实现

旨在剔除无序列表中重复元素的接口deduplicate()，可实现如代码3.14所示。

```
1 template <typename T> int List<T>::deduplicate() { //剔除无序列表中的重复节点
2    if ( _size < 2 ) return 0; //平凡列表自然无重复
3    int oldSize = _size; //记录原规模
4    ListNodePosi(T) p = header; Rank r = 0; //p从首节点开始
5    while ( trailer != ( p = p->succ ) ) { //依次直到末节点
6       ListNodePosi(T) q = find ( p->data, r, p ); //在p的r个（真）前驱中查找雷同者
7       q ? remove ( q ) : r++; //若的确存在，则删除之；否则秩加一
8    } //assert: 循环过程中的任意时刻，p的所有前驱互不相同
9    return oldSize - _size; //列表规模变化量，即被删除元素总数
10 }
```

代码3.14 无序列表剔除重复节点接口deduplicate()

与算法Vector::deduplicate()（42页代码2.14）类似，这里也是自前向后依次处理各节点p，一旦通过find()接口在p的前驱中查到雷同者，则随即调用remove()接口将其删除。

■ 正确性

向量与列表中元素的逻辑次序一致，故二者的deduplicate()算法亦具有类似的不变性和单调性（习题[3-4]），故正确性均可保证。

■ 复杂度

与无序向量的去重算法一样，该算法总共需做$\mathcal{O}(n)$步迭代。由3.3.4节的分析结论，每一步迭代中find()操作所需的时间线性正比于查找区间宽度，即当前节点的秩；由3.3.7节的分析结论，列表节点每次remove()操作仅需常数时间。因此，总体执行时间应为：

$$1 + 2 + 3 + \ldots + n = n \cdot (n + 1) / 2 = \mathcal{O}(n^2)$$

相对于无序向量，尽管此处节点删除操作所需的时间减少，但总体渐进复杂度并无改进。

3.3.10 遍历

列表也提供支持节点批量式访问（习题[3-5]）的遍历接口，其实现如代码3.15所示。

```
1 template <typename T> void List<T>::traverse ( void ( *visit ) ( T& ) ) //借助函数指针机制遍历
2 {  for ( ListNodePosi(T) p = header->succ; p != trailer; p = p->succ ) visit ( p->data );  }
3
4 template <typename T> template <typename VST> //元素类型、操作器
5 void List<T>::traverse ( VST& visit ) //借助函数对象机制遍历
6 {  for ( ListNodePosi(T) p = header->succ; p != trailer; p = p->succ ) visit ( p->data );  }
```

代码3.15 列表遍历接口traverse()

该接口的设计思路与实现方式，与向量的对应接口（2.5.8节）如出一辙，复杂度也相同。

§3.4 有序列表

若列表中所有节点的逻辑次序与其大小次序完全一致，则称作有序列表（sorted list）。为保证节点之间可以定义次序，依然假定元素类型T直接支持大小比较，或已重载相关操作符。与有序向量一致地，这里依然约定采用非降次序。

3.4.1 唯一化

与有序向量同理，有序列表中的雷同节点也必然（在逻辑上）彼此紧邻。利用这一特性，可实现重复节点删除算法如代码3.16所示。位置指针p和q分别指向每一对相邻的节点，若二者雷同则删除q，否则转向下一对相邻节点。如此反复迭代，直至检查过所有节点。

```
1 template <typename T> int List<T>::uniquify() { //成批剔除重复元素，效率更高
2    if ( _size < 2 ) return 0; //平凡列表自然无重复
3    int oldSize = _size; //记录原规模
4    ListNodePosi(T) p = first(); ListNodePosi(T) q; //p为各区段起点，q为其后继
5    while ( trailer != ( q = p->succ ) ) //反复考查紧邻的节点对(p, q)
6       if ( p->data != q->data ) p = q; //若互异，则转向下一区段
7       else remove ( q ); //否则（雷同），删除后者
8    return oldSize - _size; //列表规模变化量，即被删除元素总数
9 }
```

代码3.16 有序列表剔除重复节点接口uniquify()

整个过程的运行时间为$\mathcal{O}(_size) = \mathcal{O}(n)$，线性正比于列表原先的规模。

3.4.2 查找

■ 实现

有序列表的节点查找算法，可实现如代码3.17所示。

```
1 template <typename T> //在有序列表内节点p（可能是trailer）的n个（真）前驱中，找到不大于e的最后者
2 ListNodePosi(T) List<T>::search ( T const& e, int n, ListNodePosi(T) p ) const {
3 // assert: 0 <= n <= rank(p) < _size
4    while ( 0 <= n-- ) //对于p的最近的n个前驱，从右向左逐个比较
5       if ( ( ( p = p->pred )->data ) <= e ) break; //直至命中、数值越界或范围越界
6 // assert: 至此位置p必符合输出语义约定——尽管此前最后一次关键码比较可能没有意义（等效于与-inf比较）
7    return p; //返回查找终止的位置
8 } //失败时，返回区间左边界的前驱（可能是header）——调用者可通过valid()判断成功与否
```

代码3.17 有序列表查找接口search()

与有序向量类似，无论查找成功与否，返回的位置都应便于后续（插入等）操作的实施。

■ 顺序查找

与2.6.5节至2.6.8节有序向量的各种查找算法相比，该算法完全不同；反过来，除了循环终止条件的细微差异，多数部分反倒与3.3.4节无序列表的顺序查找算法几乎一样。

究其原因在于，尽管有序列表中的节点已在逻辑上按次序单调排列，但在动态存储策略中，节点的物理地址与逻辑次序毫无关系，故无法像有序向量那样自如地应用减治策略，从而不得不继续沿用无序列表的顺序查找策略。

■ 复杂度

与无序向量的查找算法同理：最好情况下的运行时间为$O(1)$，最坏情况下为$O(n)$。在等概率的前提下，平均运行时间也是$O(n)$，线性正比于查找区间的宽度。

§3.5 排序器

3.5.1 统一入口

与无序向量一样，针对无序列表任意合法区间的排序需求，这里也如代码3.18所示，设置了一个统一的排序操作接口。

```
1 template <typename T> void List<T>::sort ( ListNodePosi(T) p, int n ) { //列表区间排序
2    switch ( rand() % 3 ) { //随机选取排序算法。可根据具体问题的特点灵活选取或扩充
3       case 1:  insertionSort ( p, n ); break; //插入排序
4       case 2:  selectionSort ( p, n ); break; //选择排序
5       default: mergeSort ( p, n ); break; //归并排序
6    }
7 }
```

代码3.18 有序列表基于排序的构造方法

这里提供了插入排序、选择排序和归并排序三种算法，并依然以随机方式确定每次调用的具体算法，以便测试和对比。以下，将依次地讲解这几种算法的原理、实现，并分析其复杂度。

3.5.2 插入排序

■ 构思

插入排序（insertionsort）算法适用于包括向量与列表在内的任何序列结构。

算法的思路可简要描述为：始终将整个序列视作并切分为两部分：有序的前缀，无序的后缀；通过迭代，反复地将后缀的首元素转移至前缀中。由此亦可看出插入排序算法的不变性：

在任何时刻，相对于当前节点e = S[r]，前缀S[0, r)总是业已有序

算法开始时该前缀为空，不变性自然满足。

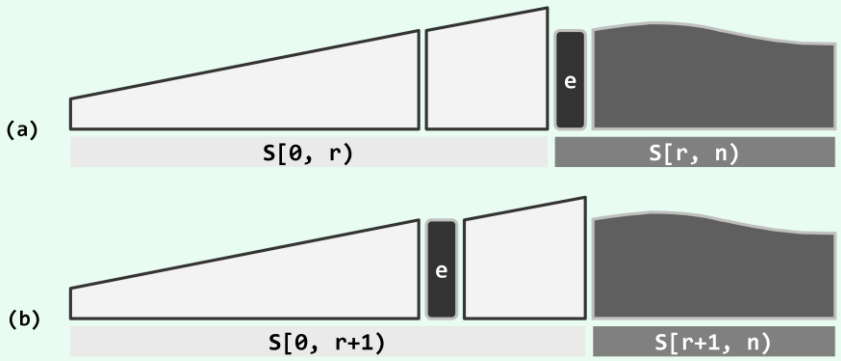

图3.5 序列的插入排序

假设如图3.5(a)所示，前缀S[0, r)已经有序。接下来，借助有序序列的查找算法，可在该前缀中定位到不大于e的最大元素。于是只需将e从无序后缀中取出，并紧邻于查找返回的位置之后插入，即可如图(b)所示，使得有序前缀的范围扩大至S[0, r]。

如此，该前缀的范围可不断拓展。当其最终覆盖整个序列时，亦即整体有序。

■ 实例

如表3.3所示，即为序列插入排序算法的一个实例。

表3.3 插入排序算法实例

#迭代	前缀有序子序列	后缀无序子序列
0	^	[5] 2 7 4 6 3 1
1	[5]	[2] 7 4 6 3 1
2	[2] 5	[7] 4 6 3 1
3	2 5 [7]	[4] 6 3 1
4	2 [4] 5 7	[6] 3 1
5	2 4 5 [6] 7	[3] 1
6	2 [3] 4 5 6 7	[1]
7	[1] 2 3 4 5 6 7	^

这里，前后共经7步迭代。输入序列中的7个元素以秩为序，先后作为首元素被取出，并插至有序前缀子序列中的适当位置。新近插入的元素均以方框注明，为确定其插入位置而在查找操作过程中接受过大小比较的元素以下划线示意。

■ 实现

依照以上思路，可针对列表实现插入排序算法如代码3.19所示。

```
1 template <typename T> //列表的插入排序算法：对起始于位置p的n个元素排序
2 void List<T>::insertionSort ( ListNodePosi(T) p, int n ) { //valid(p) && rank(p) + n <= size
3    for ( int r = 0; r < n; r++ ) { //逐一为各节点
4       insertA ( search ( p->data, r, p ), p->data ); //查找适当的位置并插入
5       p = p->succ; remove ( p->pred ); //转向下一节点
6    }
7 }
```

代码3.19 列表的插入排序

按3.4.2节的约定，有多个元素命中时search()接口将返回其中最靠后者，排序之后重复元素将保持其原有次序，故以上插入排序算法属于稳定算法。

■ 复杂度

插入排序算法共由n步迭代组成，故其运行时间应取决于，各步迭代中所执行的查找、删除及插入操作的效率。根据此前3.3.5节和3.3.7节的结论，插入操作insertAfter()和删除操作remove()均只需$\mathcal{O}(1)$时间；而由3.4.2节的结论，查找操作search()所需时间可在$\mathcal{O}(1)$至$\mathcal{O}(n)$之间浮动（从如表3.3所示的实例，也可看出这一点）。

不难验证，当输入序列已经有序时，该算法中的每次search()操作均仅需$\mathcal{O}(1)$时间，总体运行时间为$\mathcal{O}(n)$。但反过来，若输出序列完全逆序，则各次search()操作所需时间将线性递增，累计共需$\mathcal{O}(n^2)$时间。在等概率条件下，平均仍需要$\mathcal{O}(n^2)$时间（习题[3-10]）。

3.5.3 选择排序

选择排序（selectionsort）也适用于向量与列表之类的序列结构。

■ 构思

与插入排序类似，该算法也将序列划分为无序前缀和有序后缀两部分；此外，还要求前缀不大于后后缀。如此，每次只需从前缀中选出最大者，并作为最小元素转移至后缀中，即可使有序部分的范围不断扩张。

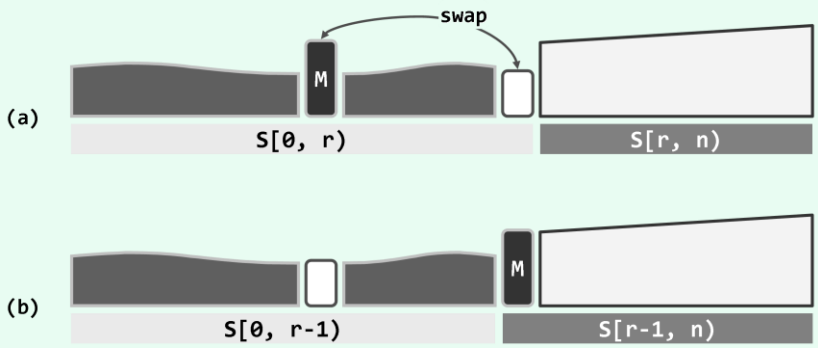

图3.6 序列的选择排序

同样地，上述描述也给出了选择排序算法过程所具有的不变性：

在任何时刻，后缀S[r, n)已经有序，且不小于前缀S[0, r)

在算法的初始时刻，后缀为空，不变性自然满足。如图3.6(a)所示，假设不变性已满足。于是，可调用无序序列的查找算法，从前缀中找出最大者M。接下来，只需将M从前缀中取出并作为首元素插入后缀，即可如图(b)所示，使得后缀的范围扩大，并继续保持有序。

如此，该后缀的范围可不断拓展。当其最终覆盖整个序列时，亦即整体有序。

■ 实例

表3.4 选择排序算法实例

#迭代	前缀无序子序列	后缀有序子序列
0	5 2 [7] 4 6 3 1	^
1	5 2 4 [6] 3 1	[7]
2	[5] 2 4 3 1	[6] 7
3	2 [4] 3 1	[5] 6 7
4	2 [3] 1	[4] 5 6 7
5	[2] 1	[3] 4 5 6 7
6	[1]	[2] 3 4 5 6 7
7	^	[1] 2 3 4 5 6 7

序列选择排序算法的一个实例如表3.4所示。其中，前后共经7步迭代，输入序列中的7个元素按由大到小的次序，依次被从无序前缀子序列中取出，并作为首元素插至初始为空的有序后缀序列中。无序子序列在各步迭代中的最大元素用方框注明。

■ 实现

依照以上思路，可针对列表实现选择排序算法如代码3.20所示。

```
1  template <typename T> //列表的选择排序算法：对起始于位置p的n个元素排序
2  void List<T>::selectionSort ( ListNodePosi(T) p, int n ) { //valid(p) && rank(p) + n <= size
3     ListNodePosi(T) head = p->pred; ListNodePosi(T) tail = p;
4     for ( int i = 0; i < n; i++ ) tail = tail->succ; //待排序区间为(head, tail)
5     while ( 1 < n ) { //在至少还剩两个节点之前，在待排序区间内
6        ListNodePosi(T) max = selectMax ( head->succ, n ); //找出最大者（歧义时后者优先）
7        insertB ( tail, remove ( max ) ); //将其移至无序区间末尾（作为有序区间新的首元素）
8        tail = tail->pred; n--;
9     }
10 }
```

代码3.20 列表的选择排序

其中的selectMax()接口用于在无序列表中定位最大节点，其实现如代码3.21所示。

```
template <typename T> //从起始于位置p的n个元素中选出最大者
ListNodePosi(T) List<T>::selectMax ( ListNodePosi(T) p, int n ) {
   ListNodePosi(T) max = p; //最大者暂定为首节点p
   for ( ListNodePosi(T) cur = p; 1 < n; n-- ) //从首节点p出发，将后续节点逐一与max比较
      if ( !lt ( ( cur = cur->succ )->data, max->data ) ) //若当前元素不小于max，则
         max = cur; //更新最大元素位置记录
   return max; //返回最大节点位置
}
```

代码3.21 列表最大节点的定位

■ **复杂度**

与插入排序类似地，选择排序亦由n步迭代组成，故其运行时间取决于各步迭代中查找及插入操作的效率。根据3.3.5和3.3.7节的结论，insertB()和remove()均只需$\mathcal{O}(1)$时间。selectMax()每次必须遍历整个无序前缀，耗时应线性正比于前缀长度；全程累计耗时$\mathcal{O}(n^2)$。

实际上进一步地仔细观察之后不难发现，无论输入序列中各元素的大小次序如何，以上n次selectMax()调用的累计耗时总是$\Theta(n^2)$。因此与插入排序算法不同，以上选择排序算法的时间复杂度为固定的$\Theta(n^2)$。也就是说，其最好和最坏情况下的渐进效率相同。

选择排序属于CBA式算法，故相对于2.7.5节所给出的$\Omega(n\log n)$下界，$\Theta(n^2)$的效率应有很大的改进空间。正如我们将在10.2.5节看到的，借助更为高级的数据结构，可以令单次selectMax()操作的复杂度降至$\mathcal{O}(\log n)$，从而使选择排序的整体效率提高至$\mathcal{O}(n\log n)$。

3.5.4 归并排序

2.8.3节介绍过基于二路归并的向量排序算法，其构思也同样适用于列表结构。实际上，有序列表的二路归并不仅可以实现，而且能够达到与有序向量二路归并同样高的效率。

■ **二路归并算法的实现**

代码3.22针对有序列表结构，给出了二路归并算法的一种实现。

```
template <typename T> //有序列表的归并：当前列表中自p起的n个元素，与列表L中自q起的m个元素归并
void List<T>::merge ( ListNodePosi(T) & p, int n, List<T>& L, ListNodePosi(T) q, int m ) {
// assert:  this.valid(p) && rank(p) + n <= size && this.sorted(p, n)
//          L.valid(q) && rank(q) + m <= L._size && L.sorted(q, m)
// 注意：在归并排序之类的场合，有可能 this == L && rank(p) + n = rank(q)
   ListNodePosi(T) pp = p->pred; //借助前驱（可能是header），以便返回前 ...
   while ( 0 < m ) //在q尚未移出区间之前
      if ( ( 0 < n ) && ( p->data <= q->data ) ) //若p仍在区间内且v(p) <= v(q)，则
         { if ( q == ( p = p->succ ) ) break; n--; } //p归入合并的列表，并替换为其直接后继
      else //若p已超出右界或v(q) < v(p)，则
         { insertB ( p, L.remove ( ( q = q->succ )->pred ) ); m--; } //将q转移至p之前
   p = pp->succ; //确定归并后区间的（新）起点
}
```

代码3.22 有序列表的二路归并

作为有序列表的内部接口，List::merge()可以将另一有序列表L中起始于节点q、长度为m的子列表，与当前有序列表中起始于节点p、长度为n的子列表做二路归并。

为便于递归地实现上层的归并排序，在二路归并的这一版本中，归并所得的有序列表依然起始于节点p。在更为通用的场合，不见得需要采用这一约定。

■ 归并时间

代码3.22中二路归并算法merge()的时间成本主要消耗于其中的迭代。该迭代反复地比较两个子列表的首节点p和q，并视其大小相应地令p指向其后继，或将节点q取出并作为p的前驱插入前一子列表。当且仅当后一子列表中所有节点均处理完毕时，迭代才会终止。因此，在最好情况下，共需迭代m次；而在最坏情况下，则需迭代n次。

总体而言，共需$\mathcal{O}(n + m)$时间，线性正比于两个子列表的长度之和。

■ 特例

在List模板类（70页代码3.2）中，作为以上二路归并通用接口的一个特例，还重载并开放了另一个接口List::merge(L)，用以将有序列表L完整地归并到当前有序列表中。

请注意，以上二路归并算法的通用接口，对列表L没有过多的限定，因此同样作为一个特例，该算法也适用于L同为当前列表的情形。此时，待归并的列表实际上是来自同一列表的两个子列表（当然，此时的两个子列表不得相互重叠。也就是说，在两个首节点中，p应是q的前驱，且二者的间距不得小于n）。对以下归并排序算法的简捷实现而言，这一特性至关重要。

■ 分治策略

仿照向量的归并排序算法mergesort()（62页代码2.28），采用分治策略并基于以上有序列表的二路归并算法，可如代码3.23所示，递归地描述和实现列表的归并排序算法。

```
1 template <typename T> //列表的归并排序算法：对起始于位置p的n个元素排序
2 void List<T>::mergeSort ( ListNodePosi(T) & p, int n ) { //valid(p) && rank(p) + n <= size
3    if ( n < 2 ) return; //若待排序范围已足够小，则直接返回；否则...
4    int m = n >> 1; //以中点为界
5    ListNodePosi(T) q = p; for ( int i = 0; i < m; i++ ) q = q->succ; //均分列表
6    mergeSort ( p, m ); mergeSort ( q, n - m ); //对前、后子列表分别排序
7    merge ( p, m, *this, q, n - m ); //归并
8 } //注意：排序后，p依然指向归并后区间的（新）起点
```

代码3.23 列表的归并排序

■ 排序时间

根据该算法的流程，为对长度为n的列表做归并排序，首先需要花费线性时间确定居中的切分节点，然后递归地对长度均为n/2的两个子列表做归并排序，最后还需花费线性的时间做二路归并。因此，仿照2.8.3节对向量归并排序算法的分析方法，同样可知其复杂度应为$\mathcal{O}(n\log n)$。另外，以上列表归并排序算法的递归跟踪过程，与如图2.19所示的向量版本别无二致。故从递归跟踪的角度，亦可得出同样的结论。

请注意，在子序列的划分阶段，向量与列表归并排序算法之间存在细微但本质的区别。前者支持循秩访问的方式，故可在$\mathcal{O}(1)$时间内确定切分中点；后者仅支持循位置访问的方式，故不得不为此花费$\mathcal{O}(n)$时间。幸好在有序子序列的合并阶段二者均需$\mathcal{O}(n)$时间，故二者的渐进时间

复杂度依然相等。

最后，尽管二路归并算法并未对子列表的长度做出任何限制，但这里出于整体效率的考虑，在划分子列表时宁可花费$O(n)$时间使得二者尽可能接近于等长。反之，若为省略这部分时间而不保证划分的均衡性，则反而可能导致整体效率的下降（习题[3-16]）。