1 list

list概述

list底层为非连续区间，即链表（实质上是一个双向循环链表）

list每次插入或者删除一个元素，就配置和释放一个元素空间，对于任何位置的原属插入或原属移除，list永远为常数时间。

list的节点

首先要知道，list本身和list的节点是不同的，如果我们声明一个list，里面放了100W个元素，然后执行sizeof，会发现sizeof的结果并不是100W。这就是因为容器所管理的内存空间大小和容器本身的大小是不一样的。

又如下测试代码：

```
#include <list>

int main(){
    std::list<int> test (100);
    printf("sizeof: %d | size: %d | sizeof(int *) : %d\n", sizeof(test),
test.size(), sizeof(int*));
    // ans: sizeof: 24 | size: 100 | sizeof(int *) : 8
}
```

我的电脑为64位机，指针大小为8，可以看出，对list进行sizeof操作，其为24，而不是100，这也就能映衬上文所说的“容器所管理的内存空间大小和容器本身的大小是不一样的”。那么这24个byte是什么呢？这就得慢慢分析源代码了。

list的节点（node）定义如下，由一个指向前一个节点的指针，指向后一个节点的指针，和数据三个部分构成：

```
template <class T>
struct __list_node {
    typedef void* void_pointer;
    void_pointer next;
    void_pointer prev;
    T data;
};
```

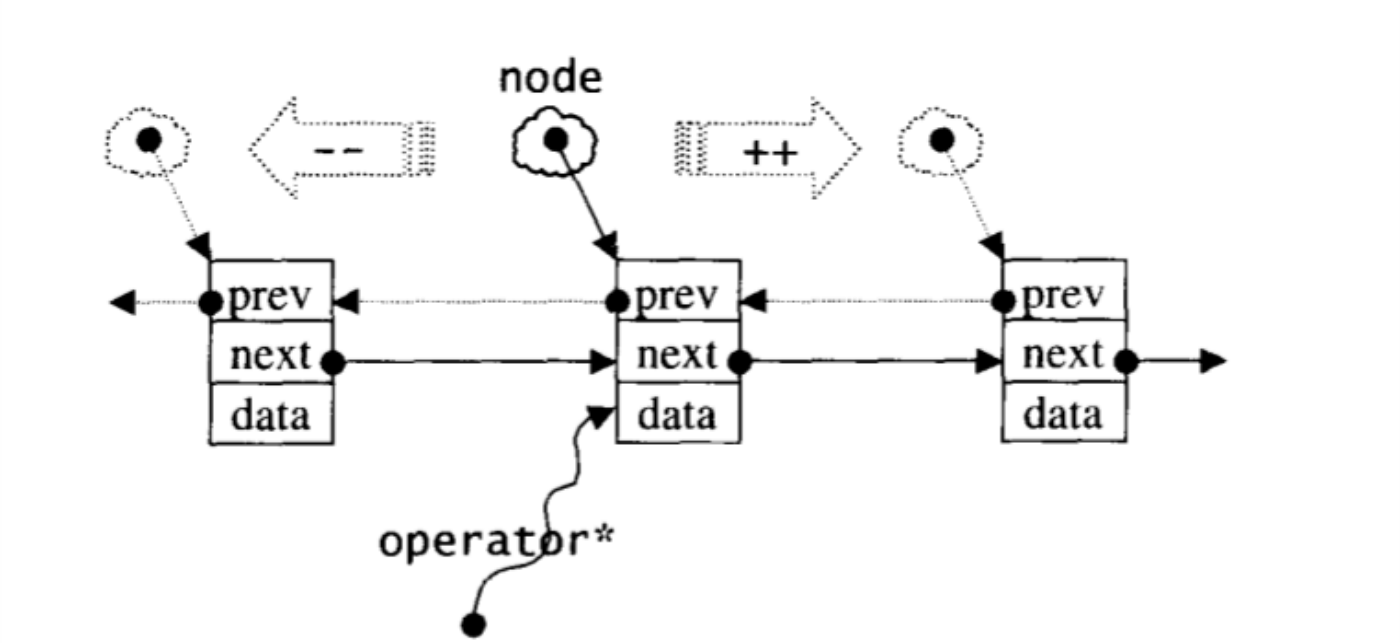
list的迭代器

list的底层节点不能保证其在内存中连续存在，因此list的迭代器是不可能实现随机访问的，只能靠next和prev两根指针的移动来进行操作。这样的迭代器是双向迭代器（bidirectional_iterator），即只能向前和向后移动。

```
template<class T, class Ref, class Ptr>
struct __list_iterator {
    // 定义相应型别
    typedef __list_iterator<T, T&, T*>              iterator;
    typedef __list_iterator<T, Ref, Ptr>            self;

    typedef bidirectional_iterator_tag iterator_category;
    typedef T value_type;
    typedef Ptr pointer;
    typedef Ref reference;
    typedef __list_node<T>* link_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    // 拥有一个指向对应结点的指针
    link_type node;

    // 构造函数
    __list_iterator() {}
    __list_iterator(link_type x) : node(x) {}
    __list_iterator(const iterator& x) : node(x.node) {}

    // 重载了iterator必须的操作符
    // 解引用，取数据
    reference operator*() const { return (*node).data; }
    // 指针使用->访问数据成员
    pointer operator->() const { return &(operator*()); }
    // ++iter，iter通过next指向下一个元素
    self& operator++() {
        node = (link_type)((*node).next);
        return *this;
    }
    self operator++(int){
        self tmp = *this;
        ++*this;
        return tmp;
    }
    // --iter，iter通过prev指向上一个元素
    self& operator--() {
        node = (link_type)((*node).prev);
        return *this;
    }
    self operator--(int){
        self tmp = *this;
        --*this;
        return tmp;
```

```
    }

    bool operator==(const self& x) const { return node == x.node; }
    bool operator!=(const self& x) const { return node != x.node; }

};
```

list有一个重要的性质：插入（insert）和接合（splice）操作不回造成原有的list迭代器失效，这在vector是不成立的。list的元素删除操作（erase）也只会让指向“被删除元素”的迭代器失效，其他迭代器不受影响。

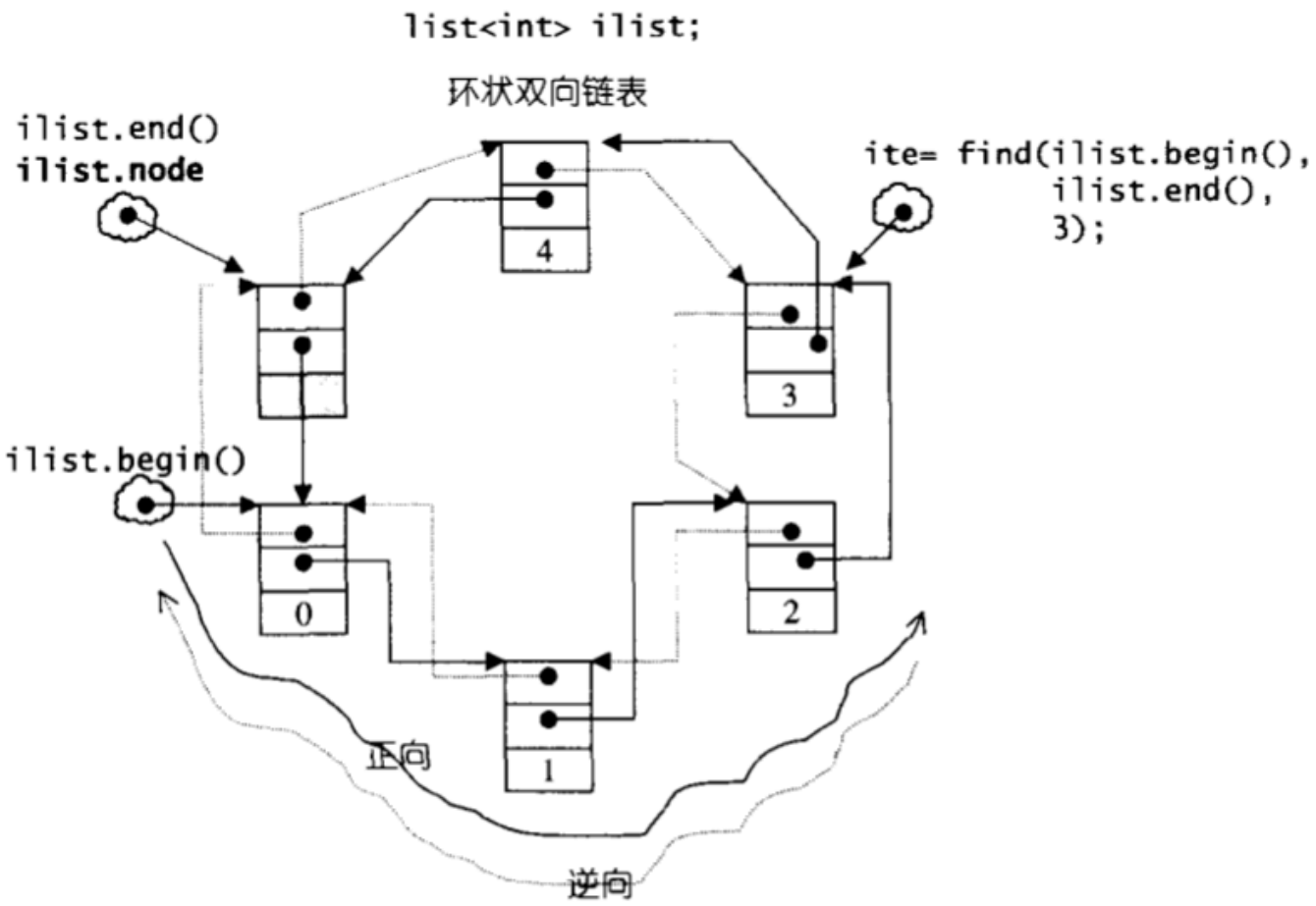


list的数据结构

list的底层数据结构就是一个双向循环链表，要表示这个双向循环链表十分的简单，就用一个节点（node）即可（这样可以说明为什么上面sizeof(list)=24了，一个指针8字节（64位机），prev、next2个指针和一个，这里有个坑，以后再补呜呜呜）。我们在数据结构中学过，一般链表有一个头节点，在list中也是一样的，只不过为了满足STL迭代器前闭后开这个特性，使得begin()为node->next，end()为node。可结合代码与图一起理解。

```
template <class T, class Alloc = alloc>
class list {
protected:
    typedef void* void_pointer;
    typedef __list_node<T> list_node;
    typedef simple_alloc<list_node, Alloc> list_node_allocator;
public:
    typedef T value_type;
    typedef value_type* pointer;
    typedef value_type& reference;
    typedef list_node* link_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
public:
    // 定义迭代器类型
    typedef __list_iterator<T, T&, T*> iterator;
```

```
protected:
    link_type node;  // 空白结点  链表尾结点
    // ...
};
```



根据上图，不难得出关于迭代器的几个操作如下：

```
// node 指向尾节点的下一位置，因此 node 符合STL对 end 的定义。
iterator begin() { return (link_type)((*node).next); }
iterator end() { return node; }
bool empty() const { return node->next == node; }
size_type size() const {
    size_type result = 0;
    distance(begin(), end(), result);  // 全局函数，求begin()和end()之间的距离，有对于bidirectional_iterator的特化版本
    return result;
}
// 取头节点的内容
reference front() { return *begin(); }
// 取尾节点的内容
reference back() { return *(--end()); }

```

list的构造与析构

直接上构造函数和析构函数吧，看代码就能说明问题！

```
template <class T, class Alloc = alloc>
class list {
public:
    // 默认构造函数，产生一个空链表
    list() { empty_initialize(); }
protected:
    // 初始化
    void empty_initialize() {
        node = get_node();  // 配置一個节点空间，令 node 指向它。
        node->next = node;  // 令node 头尾都指向自己，不设元素值。
        node->prev = node;
    }

    // 析构函数
    ~list() {
        clear(); // 清楚所有节点
        put_node(node); // 把list里边的node释放掉
    }

    // 实现在下面，清除所有节点
    void clear();

    // 为结点分配内存
    link_type get_node() { return list_node_allocator::allocate(); }
    // 回收内存
    void put_node(link_type p) { list_node_allocator::deallocate(p); }
    // 构造node
    link_type create_node(const T& x) {
        link_type p = get_node();
        construct(&p->data, x);
        return p;
    }
    // 销毁node
    void destroy_node(link_type p) {
        destroy(&p->data);
        put_node(p);
    }
};

// 清除所有节点
template <class T, class Alloc>
void list<T, Alloc>::clear()
{
  link_type cur = (link_type) node->next; // begin()
  while (cur != node) { // 访问每一个节点
    link_type tmp = cur;
    cur = (link_type) cur->next;
```

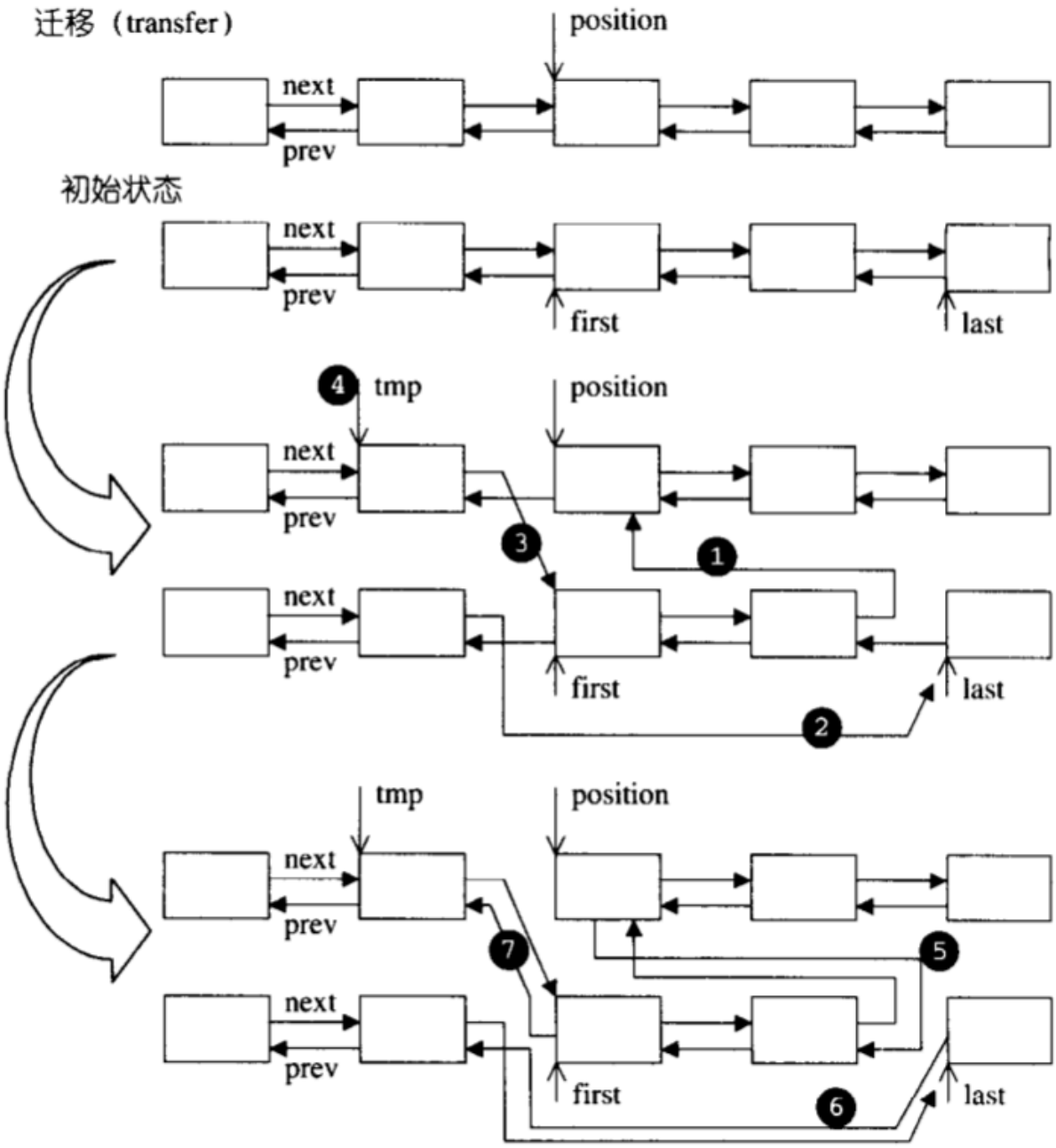
```
    destroy_node(tmp);  // 摧毁（析构并释放）一个节点
  }
  // 恢复 node 原始状态
  node->next = node;
  node->prev = node;
}

```

list的其他操作

list成员函数的实现其实就是对环状双向链表的操作。

首先是insert、erase、transfer的实现，关于插入删除大部分都调用这三个函数，实际上就是改变结点pre跟next指针的指向。

```
iterator insert(iterator position, const T& x) {
    link_type tmp = create_node(x);
    // 改变四个指针的指向 实际就是双向链表元素的插入
    tmp->next = position.node;
    tmp->prev = position.node->prev;
    (link_type(position.node->prev))->next = tmp;
    position.node->prev = tmp;
    return tmp;
}

iterator erase(iterator position) {
    // 改变四个指针的指向 实际就是双向链表的元素删除
    link_type next_node = link_type(position.node->next);
    link_type prev_node = link_type(position.node->prev);
    prev_node->next = next_node;
    next_node->prev = prev_node;
    destroy_node(position.node);
    return iterator(next_node);
}

// 将[first, last)插入到position位置(可以是同一个链表)
void transfer(iterator position, iterator first, iterator last) {
    if (position != last) {
        // 实际上也是改变双向链表结点指针的指向 具体操作看下图
        (*(link_type((*last.node).prev))).next = position.node;
        (*(link_type((*first.node).prev))).next = last.node;
        (*(link_type((*position.node).prev))).next = first.node;
        link_type tmp = link_type((*position.node).prev);
        (*position.node).prev = (*last.node).prev;
        (*last.node).prev = (*first.node).prev;
        (*first.node).prev = tmp;
    }
}
```



list的对外接口：

```
void push_front(const T& x) { insert(begin(), x); }
void push_back(const T& x) { insert(end(), x); }
void pop_front() { erase(begin()); }
void pop_back() {
    iterator tmp = end();
    erase(--tmp);
}

void swap(list<T, Alloc>& x) { __STD::swap(node, x.node); }

```

```cpp
// splice有很多重载版本
// 将 x 接合於 position 所指位置之前。x 必须不同於 *this。
void splice(iterator position, list& x) {
    if (!x.empty())
        transfer(position, x.begin(), x.end());
}
// 将 i 所指元素接合於 position 所指位置之前。position 和i 可指向同一個list。
void splice(iterator position, list&, iterator i) {
    iterator j = i;
    ++j;
    if (position == i || position == j) return;
    transfer(position, i, j);
}
// 将 [first,last) 內的所有元素接合於 position 所指位置之前。
// position 和[first,last)可指向同一個list,
// 但position不能位於[first,last)之內。
void splice(iterator position, list&, iterator first, iterator last)  {
    if (first != last)
        transfer(position, first, last);
}


// merge函数实现跟归并排序中合并的操作类似
template <class T, class Alloc>
void list<T, Alloc>::merge(list<T, Alloc>& x) {
    iterator first1 = begin();
    iterator last1 = end();
    iterator first2 = x.begin();
    iterator last2 = x.end();

    // 注意：前提是，两个list都递增排列
    while (first1 != last1 && first2 != last2)
        if (*first2 < *first1) {
            iterator next = first2;
            transfer(first1, first2, ++next);
            first2 = next;
        }
    else
        ++first1;
    if (first2 != last2) transfer(last1, first2, last2);
}

// reserse函数每次都调用transfer将结点插入到begin()之前
template <class T, class Alloc>
void list<T, Alloc>::reverse() {
    if (node->next == node || link_type(node->next)->next == node) return;
    iterator first = begin();
    ++first;
    while (first != end()) {
        iterator old = first;
        ++first;
        transfer(begin(), old, first);
```

```
    }
}

// list必须使用自己的sort()成员函数 因为STL算法中的sort()只接受RamdonAccessIterator
// 该函数采用的是quick sort
template <class T, class Alloc>
void list<T, Alloc>::sort() {
    // 空串和长度为1的 不用排序
    if (node->next == node || link_type(node->next)->next == node) return;

    // 一些新的 lists，暂存
    list<T, Alloc> carry;
    list<T, Alloc> counter[64];
    int fill = 0;
    while (!empty()) {
        carry.splice(carry.begin(), *this, begin());
        int i = 0;
        while(i < fill && !counter[i].empty()) {
            counter[i].merge(carry);
            carry.swap(counter[i++]);
        }
        carry.swap(counter[i]);
        if (i == fill) ++fill;
    }

    for (int i = 1; i < fill; ++i)
        counter[i].merge(counter[i-1]);
    swap(counter[fill-1]);
}
```