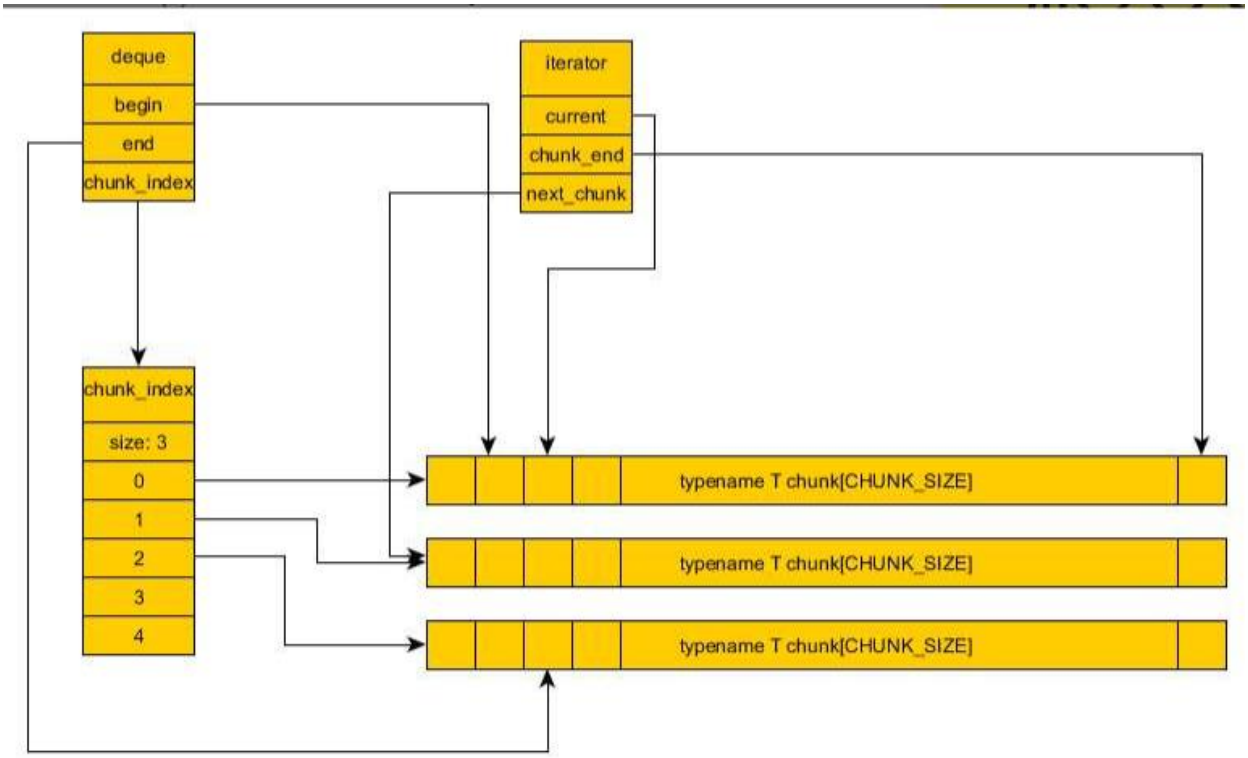
https://blog.csdn.net/ww32zz/article/details/48224941
https://www.zhihu.com/question/35521556
https://cloud.tencent.com/developer/article/1338355

deque：序列容器，一种双向开口的连续线性空间，存储数据也是连续的，和vector相似，区别在于在序列的头部插入和删除操作也是常数时间复杂度，可以在任何位置插入新元素，有随机访问功能。

1、在开头或末尾插入、删除元素的时间复杂度为 O(1)；
2、随机访问的时间复杂度为 O(1)；



deque的数据实际存储在分段数组中，每个分段数组的长度是一致的，大小是固定的，所以我们可以说deque在内存上是不连续的；这些分段数组的首地址存储在一个索引数组中，索引数组是一段连续的内存空间，如上图chunk_index；deque每次扩充的时候，扩充一整个分段数组；这个数据结构有以下特性：

1、从头部或尾部添加元素，现有元素的内存地址不会改变，因此指针和引用不失效;
2、从头部或尾部添加元素，可能需要分配新的 chunk, 这会使迭代器失效;
3、从中间添加元素，需要移动其它元素，这会使迭代器、指针和引用失效;

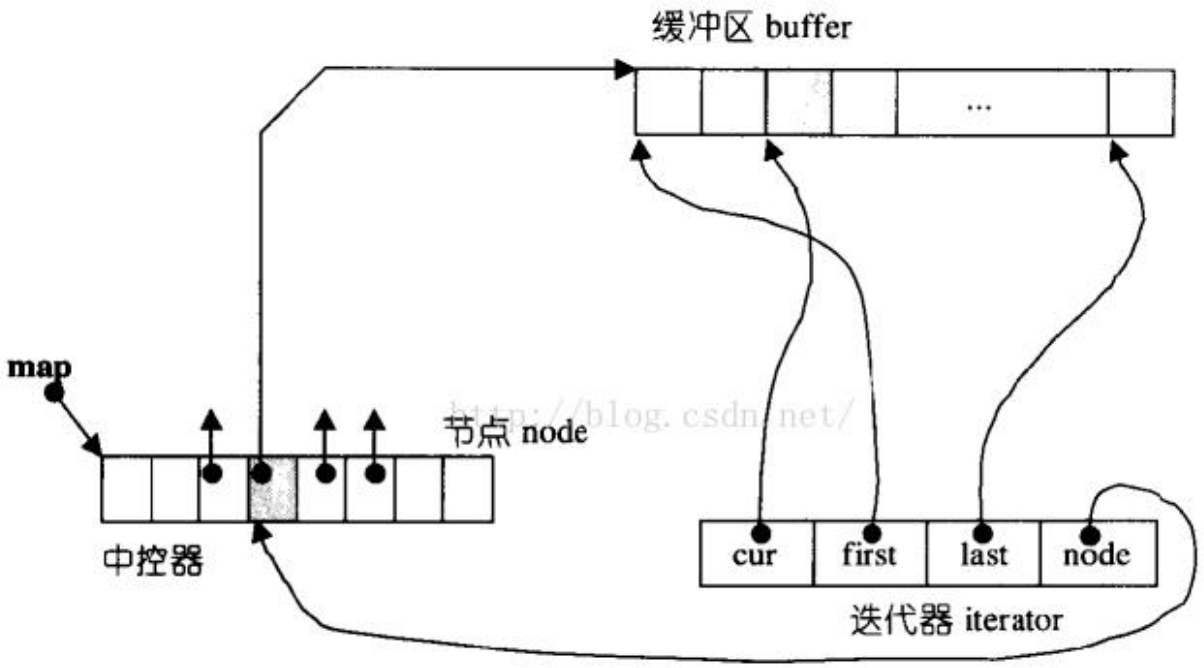


图 4-11 deque 的中控器、缓冲区、迭代器的相互关系

对迭代器it的解引用得到的是cur位置处对应的元素。deque要求map中前后各预留一个node节点，以便扩充时可用。插入元素时，导致node节点不足，于是引起map的重分配，原来的迭代器的node指向的map节点被释放，也就无法找到对应的元素，故原迭代器失效。而由于这个过程中内存并未发生改变，故其他元素的引用、指针仍然有效；

插入

①头尾：

可能指向其他元素的迭代器失效，但指针、引用仍有效；

②其他位置：

指向其他元素的迭代器、指针、引用失效；

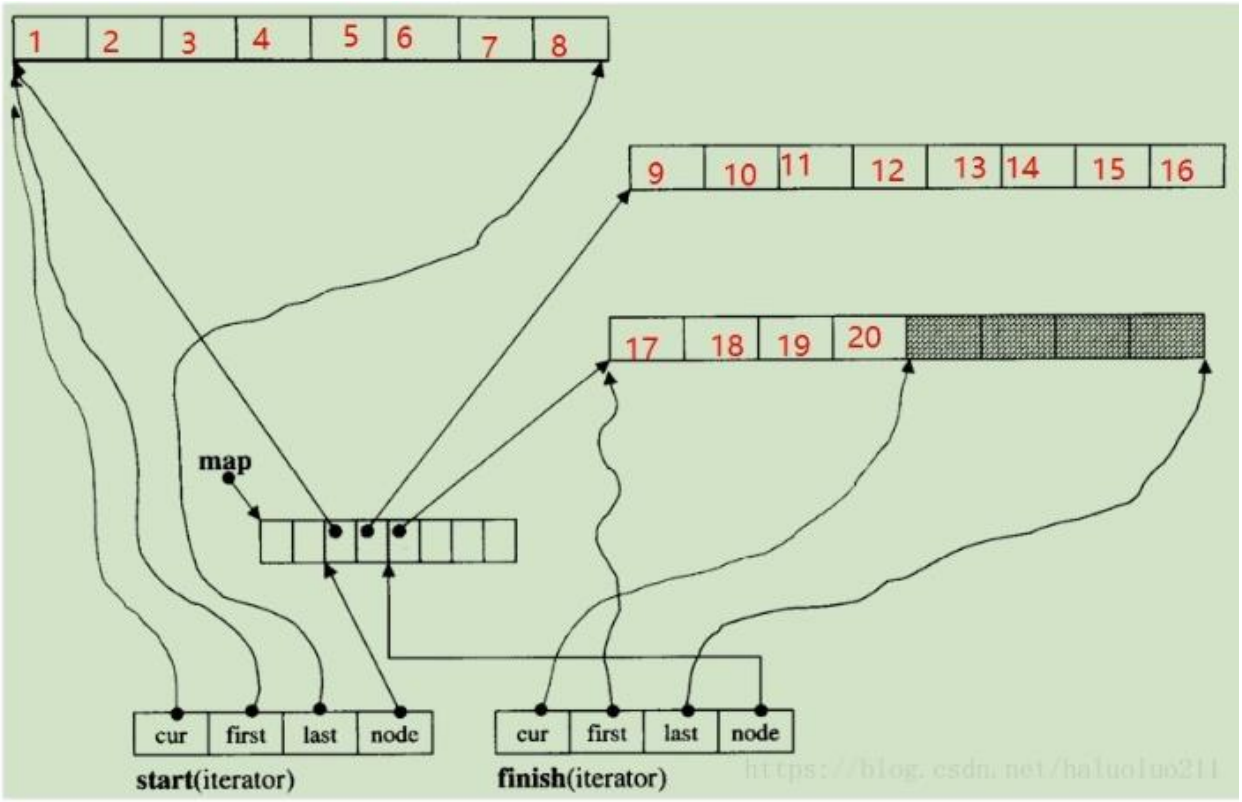
删除

①头尾：

指向其他元素的迭代器、指针、引用仍然有效；

②其他位置：

指向其他元素的迭代器、指针、引用失效；



deque迭代器设计的最大问题在于：当指针++ ，--，+n 不能简单的指针前进，后退。因为迭代器可能会遇到缓冲区的边缘，一旦遇到缓冲区边缘，要特别当心，视前进或后退而定，可能需要调用set_node函数跳一个缓冲区。

```
self& operator++(){
    ++cur;
    if (cur == last){                //如果已达所在缓冲区的尾端
        set_node(node + 1);          //切换至下一个节点(缓冲区)
        cur = first;                 //cur指向first
    }
    return *this;
}

self& operator--(){
    if(cur == first){                //已经是缓冲区的头部
        set_node(node - 1);          //切换至前一个节点
        cur = last;                  //cur指向last
    }
    --cur;
    return *this;
}
```

特性:

有序：序列容器中的元素以严格的线性顺序排列。单个元素按其顺序通过其位置访问;

动态数组：允许直接访问序列中的任何元素，甚至通过指针算术，并在序列结尾提供相对快速的添加/删除元素；
自动存储：容器使用allocator对象动态处理其存储需求；

头文件：

```
#include<deque>
```

基本接口：

```
deque.size();                //返回容器中，元素的数量
deque.max_size();            //容器能够存储的元素最大数量，受限于内存

//调整deque容器中的元素个数到n个；如果n比容器中现有元素数量少，容器中只保留前n个元素，其他元素全部弹出deque，并销毁，不改变容器占用内存大小；如果n比容器中现有元素数量多，则在容器尾部扩展元素直到满足n个元素的要求，扩展容器占用内存大小；
deque.resize(size_type n, value_type val = value_type());

//申请容器改变占用的内存大小，使占用内存大小适配当前容器中得元素个数，没有严格要求一定要使内存适配当前元素个数，各个实现库策略会不同；耗时有点久；
deque.shrink_to_fit();

//返回的均为引用，可以通过引用修改该元素的值
deque.at(size_type n);       //返回容器中位置n的元素的引用，
deque.front();               //返回容器中第一个元素的引用；
deque.back();                //返回容器中最后一个元素的引用；

//可以通过迭代器修改指向元素的值：*it = xxx;
deque.begin();          //返回指向第一个元素的迭代器
deque.end();            //返回指向最后一个元素的迭代器

deque.push_back(const value_type& val);          //插入元素到队列尾部
deque.push_front(const value_type& val);         //插入元素到队首
deque.pop_back();                                //将队列尾部的元素移出队列
deque.pop_front();                               //将队列首部的元素移出队列

//插入元素到指定位置的前面；返回值为迭代器，该迭代器指向新插入的第一个元素；
deque.insert(iterator position, const value_type& val);
deque.insert(iterator position, size_type n, const value_type& val);
deque.insert(iterator position, InputIterator first, InputIterator last);
```

//删除容器中元素，返回值为一个迭代器，该迭代器指向最后一个删除元素(position)后面的一个元素重新布局后的位置；所以可以通过erase的返回值安全的迭代删除所有元素；
deque.erase(iterator position);
deque.erase(iterator first, iterator last);

```
//use iterator to delete all element safely
for (deque<int>::iterator iter =deque.begin(); iter != deque.end();){
    iter = deque.erase(iter);
}
```

//删除容器中得所有元素，不会改变占用的内存大小
deque.clear();

deque迭代器iterator有效性问题：
增加任何元素都将使deque的迭代器失效。在deque的中间删除元素将使迭代器失效。在deque的头或尾删除元素时，只有指向该元素的迭代器失效。

In case the container shrinks, all iterators, pointers and references to elements that have not been removed remain valid after the resize and refer to the same elements they were referring to before the call; If the container expands, all iterators are invalidated, but existing pointers and references remain valid, referring to the same elements they were referring to before.
resize();

All iterators related to this container are invalidated. Pointers and references to elements in the container remain valid, referring to the same elements they were referring to before the call.
push_back();
push_front;

The iterators, pointers and references referring to the removed element are invalidated. Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.
pop_back();
pop_front();

If the insertion happens at the beginning or the end of the sequence, all iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call; If the insertion happens anywhere

else in the deque, all iterators, pointers and references related to this container are invalidated.

insert();

If the erasure operation includes the last element in the sequence, the end iterator and the iterators, pointers and references referring to the erased elements are invalidated; If the erasure includes the first element but not the last, only those referring to the erased elements are invalidated; If it happens anywhere else in the deque, all iterators, pointers and references related to the container are invalidated.

erase();

All iterators, pointers and references related to this container are invalidated.

clear();