listvector：连续内存，插入时无需重新分配！

迈克尔·乔达基斯

2015 年 9 月 17 日 警察

std::vector 和 std::list 的组合，它不会复制或移动插入中的元素并且仍然具有连续内存

下载 main.zip - 4.2 KB

上帝模式开启

我刚刚发明了一种新型的内存，它总是连续的，但是当我执行**insert()**或**push_back()**时没有元素的重新定位。它由放射性砹 210 制成，半衰期为 32 毫秒，每 MB 将耗资 14.3 亿美元。

我即将在国际空间站举行的 2042 年计算机科学大会上展示它。如果你不能参加，那么你至少可以稍后看看6,023x10^23页的会议记录。

到那时我将获得图灵奖。还记得 **std::future()** 和 **std::promise()** 吗？是的，他们答应了我很多钱。不幸的是，实现这样的容器比我为证明著名的奖杯而必须做的事情便宜得多，也更容易:)

神模式关闭

好吧，我不是神。我只是喜欢尝试奇怪的东西。

反正也没有那么难。基本上，这是一个 Windows 文件映射技巧。容器中的每个元素都有自己的固定地址，当然，列表中的元素在内存中不是连续的（如 **std::list**），但每个项目还有另一个映射指向实际地址和你得到的迭代器包含这些指针地址。

你有两种类型的指针。一、内设。这是元素实际存储的地方，它像 std::list 一样管理：插入/删除是常量，因为没有移动，但内部集合不是连续的。

外部集合（用户通过迭代器看到的）只是映射（通过**MapViewOfFileEx**）到内部指针。在插入/删除时，它们会被重新映射，以便它们指向正确的元素，同时始终是连续的。

因此，您拥有连续内存和非常优化的插入/擦除！

-THE- 抓住

对齐为 65536 字节。哎哟。抱歉，这是 Win32 中的页面粒度。因此容器中的每个元素都需要 65536 字节的倍数，这意味着，为了使用我们的容器，您要么有大对象要存储（实际上，略低于 65536 的倍数），要么完全沉迷于性能.

好吧，两者都适用于我：P。我们走吧！

为什么要换新容器?

尝试扩展现有容器时会遇到困难。

- STL 容器并不意味着从（无虚拟函数/析构函数）等继承。
- 自定义分配器只能分配/释放，没有任何关于谁在调用、迭代器位置等的通知。

因此，我不能从**std::list**或 **std::vector**继承，也不能只编写自定义分配器。必须创建一个全新的类。

内件

每个元素都存储为一个 listvector::MAP 类。该结构包含一个指向实际内存（保持固定）的指针和一个指向映射内存的指针。

GetSystemInfo () 函数返回我们的最小复用分配粒度。在我的系统中它是 64KB。

InitFull () 创建句柄和映射（从构造函数调用）。

MemPerElement () 函数决定每个元素需要多少内存，与分配粒度对齐。

GetMaxMemoryNeeded() 根据容器的容量和每个元素所需的内存来决定我们需要多少内存。

MapTheList () 和 **UnmapTheList** () 将元素的实际、固定、非连续内存地址映射和取消映射到非固定连续内存地址，因此 **data()**、**at()**和 **operator[]**可以工作。映射是在**MapViewOfFileEx** 的帮助下通过读/写访问完成的， 它强制 Windows 为我们使用特定的映射地址。

SetMinAddress() 找到当前容量的最佳基映射地址。这就是 **data()** 返回的内容。

CreateH ()和 **DestroyH** ()实际上创建和销毁了文件映射对象和内存的实际映射。

如果容量必须增长，**recreate** () 将重新创建整个容器。

ClearAndDestroy () 清理（由 recreate() 和析构函数使用）。

FindMapAtPosition () 从实际地址中查找元素。**FindFreeAreaForElement** ()使用它 来为新元素创建固定内存。

smartadd () 根据当前和请求的大小更改容量。

CreateMemoryAtEnd() 由**push_back** 等函数调用，为最后插入创建内存。它是**CreateMemoryBeforePosition** ()的优化版本。

CreateMemoryBeforePosition () 被任何其他内存创建调用。

模板 <typename T> 列表向量

该类是**std::list**和**std::vector 的组合**，提供几乎相同的函数集，除了我们将要注意的一些例外情况。该类采用模板形式，这意味着它适用于有资格包含在 STL 容器中的所有类型。

如果某些 Win32 操作失败，该类将抛出 **std::bad_alloc** 。

构造函数、运算符 =() 和析构函数

C++ 复制代码

```
// Creates an empty listvector with capacity = 10
listvector();
```

```
// Creates a listvector with a specific size.
listvector(size_t);

// Constructs from an initializer_list.
listvector(const std::initializer_list<T>&);
listvector(std::initializer_list<T>&&);

// Constructs from a vector
listvector(const std::vector<T>&);
listvector(std::vector<T>&&);

// Constructs from a list
listvector(const std::list<T>&);
listvector(std::list<T>&&);

// Constructs from a listvector
listvector(const listvector<T>&);
listvector(listvector<T>&&);

// Destructor
~listvector();
```

每个构造函数都伴随着它的相关运算符 =()。

at()、operator[] 和位置请求

C++ 复制代码

```
T& at(size_t idx);
T& operator[](size_t idx);
T& back();
T& front();
```

类似于 STL。**at()** 执行边界检查（抛出**std::out_of_range**），而 operator[] 不执行。 **const**版本也在那里。

迭代器

C++ 复制代码

```
listvectoriterator<T> begin();
const listvectoriterator<T> cbegin() const;
listvectoriterator<T> rbegin();
const listvectoriterator<T> crbegin() const;
listvectoriterator<T> end();
const listvectoriterator<T> cend() const;
listvectoriterator<T> rend();
const listvectoriterator<T> rend() const;
```

类似于 STL。**const** 版本也在那里。迭代器总是指向元素的映射地址，这些地址在调整大小时会改变，但实际元素不会改变，因为它们的原始地址保持不变。

listvectoriterator 还提供了 ++,+,-,==,!= 等操作符。

insert()、emplace()、splice() 和 pushes

C++ 复制代码

```
template <class... Args>
        listvectoriterator<T> emplace(const listvectoriterator<T> position,Args&&... args);
template <class... Args>
        void emplace_back(Args&&... args);
template <class... Args>
        void emplace_front(Args&&... args);
listvectoriterator<T> insert(const listvectoriterator<T> position,const T& val);
listvectoriterator<T> insert(const listvectoriterator<T> position,T&& val);
void insert(const listvectoriterator<T> pos,size_t count,const T& value);
void insert(const listvectoriterator<T> pos,std::initializer_list<T> li);
template <class InputIterator>
        listvectoriterator<T> insert(const listvectoriterator<T> pos,InputIterator
first,InputIterator last);
void push_back(const T& val);
void push_back(T&& val);
void push_front(const T& val);
void push_front(T&& val);
void splice(const listvectoriterator<T> pos,listvector& x);
void splice(const listvectoriterator<T> pos,listvector&& x);
void splice(const listvectoriterator<T> pos,listvector& x,const listvectoriterator<T> i);
void splice(const listvectoriterator<T> pos,listvector&& x,const listvectoriterator<T> i);
void splice(const listvectoriterator<T> pos,listvector& x,const listvectoriterator<T>
first,const listvectoriterator<T> last);
void splice(const listvectoriterator<T> pos,listvector&& x,const listvectoriterator<T>
first,const listvectoriterator<T> last);
```

类似于 STL。发生插入时，容器不会移动/复制任何其他元素。相反，它将它们的实际地址（它们是固定的而不是连续的）重新映射到始终连续的映射地址。如果插入不强制改变容量，则原始起始地址（即从**data()**返回的地址 ）是相同的。如果插入迫使容量发生变化，则基本内存也会发生变化，项目将移动到新位置。

擦除（），删除（）和弹出

C++ 复制代码

```
void remove(const T& val);
template <class Predicate>
        void remove_if(Predicate pred);
listvectoriterator<T> erase(listvectoriterator<T> position);
listvectoriterator<T> erase(listvectoriterator<T> first,listvectoriterator<T> last);
void pop_back();
void pop_front();
```

类似于 STL。擦除不会改变容量。

更多内部结构

在这里，您将看到有关容器的一些有趣的细节。

迭代器的增减不是简单的指针类型本身的+=，因为我们需要对齐。因此，让我们看一个示例：

复制代码

```
listvectoriterator<T>& operator++()
    {
```

```
    size_t pp = (size_t)p;
    if (R)
        pp -= sz;
    else
        pp += sz;
    p = (T*)pp;
    return *this;
    }
```

所以首先我们将它转换为**size_t**，然后添加元素大小（这是对齐的乘法），然后返回到指针。

MapTheList () 遍历元素，然后使用 **MapViewOfFileEx** 强制基地址。

复制代码

```
for (size_t i = 0; i < elements.size(); i++)
    {
    size_t bb = b;
    bb += i*MemPerElement();
    ULARGE_INTEGER mp = {0};
    mp.QuadPart = (unsigned long long)elements[i].ActualPtr;
    mp.QuadPart -= (unsigned long long)FullMap;
    elements[i].Map = (T*)MapViewOfFileEx(hF,FILE_MAP_ALL_ACCESS,mp.HighPart,mp.LowPart,grd,
(LPVOID)bb);
    if (!elements[i].Map)
        throw std::bad_alloc();
    }
```

因为我们之前有 tester 文件映射可以映射整个容量，所以我们知道调用会成功（当然除非其他人已经映射到该内存！）。

emplace函数使用placement new 来调用构造函数：

复制代码

```
template <class... Args>
listvectoriterator<T> emplace(const listvectoriterator<T> position,Args&&... args)
    {
    MAP& m = CreateMemoryBeforePosition(position.idx());
    new (m.ActualPtr) T(std::forward<T>(args...));
    listvectoriterator<T> r((size_t)MinAddress,(size_t)m.Map,MemPerElement());
    return r;
    }
```

请注意可变参数模板的使用，以便将所有参数“就地”转发给构造函数。

该 **排序** 功能只需使用std ::排序，然后重新映射的结果：

复制代码

```
void sort()
    {
    UnmapTheList();
    std::sort(elements.begin(),elements.end());
    MapTheList();
    }
```

由于**元素** 数组包含指针，因此不会重新分配或重建任何内容。

其他 STL 助手

还有以下功能：

- 容量（）；
- 清除（）；
- 数据（）；
- 尺寸（）；
- 最大尺寸（）；
- 预订（）；
- 调整大小（）；
- 逆转（）；
- 缩小到适合（）；
- sort() 和模板排序与比较类。
- 交换（）；
- unique() 和模板在 BinaryPredicate 类中是唯一的

缺少其他一些，例如 **assign** () 和 **merge** ()。

测试

在该文件中有一些测试功能将向您展示容器是如何工作的。我还创建了一个带有复制/移动构造函数等的类 F，以验证是否调用了正确的构造函数。

结论

当然，并不是所有的功能都经过任何种类的 contains-ee 测试。做贡献并向我报告错误。

玩得开心！

历史

- 2015 年 13 月 9 日：首次发布

执照

分享

关于作者

迈克尔·乔达基斯
软件开发人员
希腊
手表该会员

我正在使用 C++、PHP、Java、Windows、iOS、Android 和 Web（HTML/Javascript/CSS）。

我拥有数字信号处理和人工智能博士学位，专攻专业音频和人工智能应用。

我的主页：https://www.turbo-play.com

评论和讨论

添加评论或问题 | 电子邮件提醒 | Search Comments

第一 页上一页 下一页

我的5票
hooodaticus 18-Nov-16 4:18

回复：我的5票
Michael Chourdakis 19-Dec-16 21:02

用法示例?
Marius Bancila 13-Nov-15 4:43

Re: Example of usage?
Michael Chourdakis 13-Nov-15 5:40

My vote of 5
Ștefan-Mihai MOGA 18-Oct-15 10:01

Re: My vote of 5
Michael Chourdakis 18-Oct-15 16:19

My vote of 5
Farhad Reza 9-Oct-15 12:12

[My vote of 2] What are the advantages of using your listvector instead of something simpler and portable?
PedroMC 17-Sep-15 20:57

Re: [My vote of 2] What are the advantages of using your listvector instead of something simpler and portable?
Michael Chourdakis 18-Sep-15 14:01

Re: [My vote of 2] What are the advantages of using your listvector instead of something simpler and portable?
PedroMC 18-Sep-15 18:24

Re: [My vote of 2] What are the advantages of using your listvector instead of something simpler and portable?
Michael Chourdakis 18-Sep-15 20:20

Interesting ...
koothkeeper **16-Sep-15 6:14**

Reinventing the wheel
Member 11986335 **16-Sep-15 1:01**

Re: Reinventing the wheel
Michael Chourdakis 16-Sep-15 1:25

My vote of 5
sprice86 **16-Sep-15 0:12**

You have lost me
_kb_ **14-Sep-15 21:26**

Re: You have lost me
Michael Chourdakis 15-Sep-15 0:12

Interesting trick, but...
ZhenyOK 16-Sep-15 5:26

Re: Interesting trick, but...
Michael Chourdakis 17-Sep-15 5:03

Re: Interesting trick, but...
ZhenyOK 17-Sep-15 6:18

Re: Interesting trick, but...
Michael Chourdakis 18-Sep-15 14:02

Re: You have lost me
_kb_ 16-Sep-15 17:37

Re: You have lost me
Michael Chourdakis 17-Sep-15 5:05

回复：你失去了我
Paulo Zemek 26-Sep-15 8:08

回复：你失去了我
Michael Chourdakis 26-Sep-15 15:22

刷新 1 2 下一个 ▷

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。