

QCore/Library 实现文档

李文超

前言

QCore/Library 是一套类 STL 的类库，它在标准库的范围内删去了不常用的 heap、deque 等结构（至少我是不常用的）。并为一些容器提供了一些特殊的接口，比如 vector 中的 push_back_unique、add 和 add_unique 等。

Library 主要分为六部分，内存调试相关、容器、算法、正则、IO 和 Graphic，每个模块都有各自的分工，他们之间的耦合度极低，几乎每个模块都可以拆出来独立使用，下面来分别介绍各个模块。

内存调试

我们知道，在 C/C++ 中内存相关的东西是极难控制的，在使用不当时可能造成各种错误，轻则内存泄漏，重则程序崩溃。所以，在生产环境中我们必须通过一个有效的手段来管理好内存。当然，在小块内存频繁 new、delete 的过程中也会产生大量的内存碎片，从而导致可用内存数量越来越少。应此我们设计了一个内存池来控制小块内存的频繁 new、delete，以及做到对内存泄漏的检测。

在内存池的设计之初，我只是简单的设计出了可以使用的 MemoryPool 的最简版本，它包含一个大块内存的 free_list 和每个小块内存的 chunk_list，当时足以应付大部分的需求，而且最初用的是 [Visual Leak Detector](#) 来检测内存泄漏。但随着时间的推移，想要自己检测内存泄漏的欲望越来越强烈，之后便有了一个 use_list 来保存内存块的释放情况。当时完成了这个 patch 之后，兴奋的跑了一下 TestCase，然后的结果我想大家应该知道了，一路的飘红，到处是内存泄漏。

经过一天的调试，实在无法容忍的情况下，我翻阅了 MSDN，查到了 dbghelp.dll 中可以通过许多函数来获取调用堆栈，于是在此之下便生产出了 CallStack 模块。有了它之后你就可以在任意地方保存当前的调用堆栈了，真是十分方便。当然直到现在，它还只支持在 Windows 下调用堆栈的获取（稍后我会翻阅资料，实现一个 like unix 的版本，如果可能的话）。

这里不过多的描述实现的细节，具体可以看 <http://www.cppblog.com/lwch/archive/2012/07/14/183420.html> 和 <http://www.cppblog.com/lwch/archive/2013/01/19/197415.html> 两篇文章。

最后来看 allocator，这里只是简单的为其包装了一层。

```
template <typename T>
class allocator
```

```

{
public:
    allocator()
    {
    }

    allocator(const allocator<T>&)
    {
    }

    static T* allocate()
    {
        MemoryPool* pool = getPool();
        return reinterpret_cast<T*>(pool->allocate(sizeof(T), free_handler));
    }

    static T* allocate(size_t n)
    {
        MemoryPool* pool = getPool();
        return reinterpret_cast<T*>(pool->allocate(n * sizeof(T), free_handler));
    }

    static void deallocate(T* p)
    {
        MemoryPool* pool = getPool();
        pool->deallocate(p, sizeof(T));
    }

    static void deallocate(T* p, size_t n)
    {
        MemoryPool* pool = getPool();
        pool->deallocate(p, n * sizeof(T));
    }

    static void deallocateWithSize(T* p, size_t n)
    {
        MemoryPool* pool = getPool();
        pool->deallocate(p, n);
    }

    static T* reallocate(T* p, size_t old_size, size_t n)
    {
        MemoryPool* pool = getPool();
        return pool->reallocate(p, old_size, n * sizeof(T), free_handler);
    }

```

```

    }
public:
    static void(*free_handler)(size_t);

    static void set_handler(void(*h)(size_t))
    {
        free_handler = h;
    }
protected:
    static MemoryPool* getPool()
    {
        static MemoryPool pool;
        return &pool;
    }
};

template <typename T>
void (*allocator<T>::free_handler)(size_t) = 0;

```

容器

容器占了 Library 的大部分，容器的作用是用来存储对象的，容器分为线性和非线性两种。线性的容器有 `vector`、`list`、`string` 以及用它们作为容器实现的 `queue`、`stack` 四种，非线性的则有 `rbtree`、`hashtable` 以及用它们作为容器实现的 `set`、`map`、`hashset`、`hashmap` 六种。对于每种容器，都必须定义出它的 `value_type`、`pointer`、`reference`、`const_reference`、`size_type`、`distance_type`、`const_iterator`、`const_reverse_iterator`、`iterator`、`reverse_iterator` 的类型。

所有容器必须包含以下几个接口：`size`（获取容器内元素个数）、`clear`（清空容器）、`begin`（获取`[first,last)`区间中的 `first` 迭代器）、`end`（获取`[first,last)`区间中的 `last` 迭代器）、`rbegin`（获取反向的 `first` 迭代器）、`rend`（获取反向的 `last` 迭代器）。

traits

`traits` 是一种萃取技术，通过它你可以获取某种类型的一些特性，比如是否含有默认构造函数、拷贝构造函数等。

__type_traits

`__type_traits` 用于萃取出某种类型的一些特性，它的原型如下

```

template <typename T>

```

```

struct __type_traits
{
    typedef __true_type    has_default_construct;
    typedef __true_type    has_copy_construct;
    typedef __true_type    has_assign_operator;
    typedef __true_type    has_destruct;
    typedef __false_type   is_POD;
};

```

通过特例化，可以定义出所有类型的这些属性，比如 char

```

template <>
struct __type_traits<char>
{
    typedef __true_type    has_default_construct;
    typedef __true_type    has_copy_construct;
    typedef __true_type    has_assign_operator;
    typedef __false_type   has_destruct;
    typedef __true_type    is_POD;
};

```

__container_traits

__container_traits 用于萃取出容器的特性，如上文所说的 value_type 等特性，它的代码很简单

```

template <typename T>
struct __container_traits
{
    typedef typename T::value_type      value_type;
    typedef typename T::pointer         pointer;
    typedef typename T::reference        reference;
    typedef typename T::const_reference const_reference;
    typedef typename T::size_type        size_type;
    typedef typename T::distance_type    distance_type;
    typedef typename T::const_iterator   const_iterator;
    typedef typename T::const_reverse_iterator const_reverse_iterator;
    typedef typename T::iterator         iterator;
    typedef typename T::reverse_iterator reverse_iterator;
};

```

__char_traits

__char_traits 定义了一些对于 Char 的操作，包括 assign（赋值）、eq（相等）、lt（小于）、

compare（比较两个字符串的大小）、length（获取字符串的长度）、move（移动）、copy（拷贝）、assign（字符串赋值）、eof（结束符），它的[代码](#)比较简洁，这里不做说明。

type_compare

type_compare 用于对两种类型做运行时的匹配，判断所给定的两种类型是否相同。同样通过特例化技术可以很轻松的实现它的[代码](#)。

迭代器

迭代器类是一种类似于 smart pointer 的东西，一般的它都会支持前置和后置的++和--操作，有一些特殊的迭代器同样支持+=和-=操作。当然作为一种 smart pointer 少不了的是->和*操作，而对于比较操作，则比较的是迭代器所保存的值。

迭代器分为 bidirectional_iterator 和 random_access_iterator 两种类型，前者只支持++和--操作而后者支持+和-运算符，之所以会定义出这两种类型是为了提高算法的速度。对于一个迭代器来说同样需要定义 value_type、distance_type、pointer、reference、const_pointer、const_reference 的类型。

反向迭代器

反向迭代器与正向的正好相反，应此我们可以类似的定义它的++为正向迭代器的--等运算符

iterator_traits

iterator_traits 用于萃取出迭代器的所有特性，应此它比较简单

```
template <typename Iterator>
struct iterator_traits
{
    typedef typename Iterator::value_type      value_type;
    typedef typename Iterator::distance_type   distance_type;
    typedef typename Iterator::pointer         pointer;
    typedef typename Iterator::reference       reference;
    typedef typename Iterator::const_pointer   const_pointer;
    typedef typename Iterator::const_reference const_reference;
};
```

vector

vector 是一种比较常用的容器，它的内部是一个连续的内存块，因此它有两个接口分别用于获取内存块已使用的大小和容器的大小，它们是 **size** 和 **capacity**，同样它也有一个 **reserve** 接口来调整它的容量。由于 **vector** 的内部是连续的，因此它只允许从后面插入元素，所以它有 **push_back**、**push_back_unique** 和 **pop_back** 方法。当然为了作为 **queue** 的容器，我还为其增加了 **pop_front** 方法用于删除前端的元素。**insert** 和 **erase** 用于在某个地方插入和删除元素，**add** 和 **add_unique** 用于插入另一个 **vector** 里的内容，而 **unique** 则会将容器内重复的元素删除。

当然 **vector** 可以像数组一样的使用，它支持方括号的运算符与 **at** 接口来获取某个位置上的元素值。

vector 容器就先介绍到这里，它的代码你可以在 [QCore/Library/vector.h](#) 中找到。

list

list 的内部则是一个双向的链表，因此它在删除前端的元素时会比 **vector** 快很多，它的接口基本跟 **vector** 相同，这里就不做过多的介绍了，它的代码你可以在 [QCore/Library/list.h](#) 中找到。

queue 和 stack

queue 是一种 FIFO（先进先出）的结构，因此我们建议使用 **list** 作为它的容器，通过 **list** 的 **push_back** 和 **pop_front** 可以使代码变的高效。它拥有 **front** 和 **back** 方法来获取队列中队头和队尾的元素值，同样在插入队列时，你可以不加选择的直接插入或是插入一个不重复的值。

stack 是一种 FILO（先进后出）的结构，同样它拥有 **push**、**push_unique** 和 **pop** 方法以及 **top** 和 **bottom** 方法用于获取栈顶端和底部的元素值。

对于这两种结构的代码，你可以在 [QCore/Library/queue.h](#) 和 [QCore/Library/stack.h](#) 中找到。

rbtree

rbtree（红黑树）是一棵自平衡的二叉查找树，因此它拥有较高的查找效率，红黑树有以下 5 条性质

- 性质 1. 节点是红色或黑色。
- 性质 2. 根节点是黑色。

性质 3 每个叶节点是黑色的。

性质 4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

性质 5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

以上摘自百度百科

对于红黑树的每个节点，它都拥有它的 `key` 和 `value`，当然我们是按照节点的 `key` 来排序的（废话）。红黑树拥有 `insert_equal` 和 `insert_unique` 方法分别用于插入一个节点，前者允许待插入节点的 `key` 已存在于这棵树中，而后者在插入时并不允许这个节点的 `key` 已存在于这棵树中，应此它的返回值则是一个二元的。`erase` 方法则提供了对于树中某个节点的删除操作，可以通过迭代器或某个 `key` 值来删除其中的节点。

由于红黑树是一棵二叉查找树，应此它应该具备 `find` 方法，用于在树中查找某个节点，它返回的则是指向这个节点的一个迭代器。由于红黑树是有序的，应此可以通过 `maximum` 和 `minimum` 方法得到其中的最大和最小值。通过 `lower_bound` 和 `upper_bound` 可以得到属于某个 `key` 的 `[first,last]` 区间，`equal_range` 就是干这个活的，`count` 方法可以得到某个 `key` 所对应的节点数。

`rbtree` 就先介绍到这里，稍后我会在博客中继续更新提供更完整的实现方法，它的代码你可以在 [QCore/Library/rbtree.h](#) 中找到。

set 和 map

`set` 是一个集合，应此在集合中是不允许有重复的元素的，`set` 是以 `rbtree` 作为容器的。应此它的 `insert` 方法对应于 `rbtree` 的 `insert_unique` 方法，同样 `rbtree` 所具备的接口 `set` 也同样拥有，`set` 的 `key_type` 与 `value_type` 相同，都是给定的类型。

`map` 则是一种 `key-value` 的 `directory` 结构，应此它的 `key` 是不允许重复的，`map` 同样是以 `rbtree` 作为容器的。应此它的 `insert` 方法同样对应于 `rbtree` 的 `insert_unique` 方法，在 `map` 中除了 `rbtree` 的 `maximum`、`minimum`、`lower_bound`、`upper_bound`、`equal_range` 和 `count` 没有之外其他接口基本全都拥有，`map` 的 `key_type` 是给定的类型，而 `value_type` 则是一个以 `Key` 和 `T` 组成的二元组。

对于这两种结构的代码，你可以在 [QCore/Library/set.h](#) 和 [QCore/Library/map.h](#) 中找到。

结束

修改记录

2013.4.23 第一次编写

2013.4.24 添加容器的说明