# 1.radix tree：

参考1：<https://github.com/kennycaiguo/libart>

参考2，<https://blog.csdn.net/lindorx/article/details/103431530>

#### 参考源码

|  |
| --- |
| #pragma once  #include<stdlib.h>  #include<stdio.h>  #define MEMPAGE 4096//内存页的大小，一般为4kb  #define INIT\_POOL\_SIZE (MEMPAGE\*32) //初始内存池大小  #define INIT\_FREE\_SIZE (INIT\_POOL\_SIZE/2) //初始备用节点长度  #define INIT\_NODE\_NUM (16\*32)    #define RADIX\_INSERT\_VALUE\_OCCUPY -1 //该节点已被占用  #define RADIX\_INSERT\_VALUE\_SAME -2 //插入了相同的值  #define RADIX\_DELETE\_ERROR -3 //删除错误  typedef unsigned int ptr\_t;  typedef unsigned int uint32;    #define BITS 2  const int radix\_tree\_height = sizeof(ptr\_t) \* 8 / BITS;//树的高度  //返回key中由pos指定的位的值，位数由BITS指定  #define CHECK\_BITS(key,pos) ((((unsigned int)(key))<<sizeof(int)\*8-((pos)+1)\*BITS)>>(sizeof(int)\*8-BITS))  //基数树节点  typedef struct radix\_node\_t radix\_node\_t;  struct radix\_node\_t {  radix\_node\_t\* child[4];  radix\_node\_t\* parent;  ptr\_t value;//节点储存的值  };  //使用内存池是为减少建立节点时重新申请内存的时间  //内存池描述结构，放在内存池的前段  typedef struct radix\_pool {  struct radix\_pool\* next;//内存池是双向循环链表的一个节点  struct radix\_pool\* prev;  //已分配内存中还未使用的内存首地址  char\* start;  //已分配内存中还未使用的内存长度  size\_t size;  }radix\_pool, \* pool\_t;  //基数树管理结构  typedef struct radix\_tree\_t {  //根节点  radix\_node\_t\* root;  //内存池指针  pool\_t pool;  //储存已分配但不在树中的节点（双向链表，这里储存其中的一个节点）  radix\_node\_t\* free;  }radix\_tree\_t;  //内存池扩大函数，num：新内存池的大小，=-1使用默认值,单位：页  pool\_t get\_new\_pool(radix\_tree\_t\* t, size\_t num)  {  if (num == -1)num = INIT\_POOL\_SIZE;  pool\_t pool = (pool\_t)malloc(num \* MEMPAGE);  if (pool == NULL)return NULL;  pool->start = (char\*)pool + sizeof(radix\_pool);  pool->size = num \* MEMPAGE - sizeof(radix\_pool);  pool->next = t->pool->next;  pool->prev = t->pool;  t->pool->next->prev = pool;  t->pool->next = pool;  t->pool = pool;  return pool;  }    //创建一个节点，从内存池中取出可以使用的节点  radix\_node\_t\* radix\_node\_alloc(radix\_tree\_t\* t)  {  radix\_node\_t\* node;  if (t->free != NULL) {//从free中提取节点  node = t->free;  t->free = node->parent;  }  else {//在内存池中寻找可以使用的内存  if (t->pool->size < sizeof(radix\_node\_t)) {//如果剩余空间不够分配，则重新分配  get\_new\_pool(t, -1);  }  node = (radix\_node\_t\*)t->pool->start;  t->pool->start += sizeof(radix\_node\_t);  t->pool->size -= sizeof(radix\_node\_t);  }  node->child[0] = NULL;  node->child[1] = NULL;  node->child[2] = NULL;  node->child[3] = NULL;  node->parent = NULL;  node->value = NULL;  return node;  }  //创建管理结构  radix\_tree\_t\* radix\_tree\_create()  {  int i;  radix\_tree\_t\* tree = (radix\_tree\_t\*)malloc(sizeof(radix\_tree\_t));  if (tree == NULL)return NULL;  char\* p = (char\*)malloc(INIT\_POOL\_SIZE);  radix\_node\_t\* ns;  if (!p) {  free(tree); return NULL;  }  //为内存池结构分配空间  ((pool\_t)p)->next = (pool\_t)p;  ((pool\_t)p)->prev = (pool\_t)p;  ns = (radix\_node\_t\*)((char\*)p + sizeof(radix\_pool));    //在内存中创建链表  for (i = 1; i < INIT\_NODE\_NUM - 2; ++i) {  ns[i].parent = &ns[i + 1];  }  ns[i].parent = NULL;  ns[0].child[0] = NULL;  ns[0].child[1] = NULL;  ns[0].child[2] = NULL;  ns[0].child[3] = NULL;  ns[0].parent = NULL;  ns[0].value = NULL;  tree->pool = (pool\_t)p;  tree->root = ns;  tree->free = &ns[1];  ((pool\_t)p)->start = (char\*)ns + sizeof(radix\_node\_t) \* INIT\_NODE\_NUM;  ((pool\_t)p)->size = INIT\_POOL\_SIZE - sizeof(radix\_pool) - sizeof(radix\_node\_t) \* INIT\_NODE\_NUM;  return tree;  }    //插入  int radix\_tree\_insert(radix\_tree\_t\* t, uint32 key, ptr\_t value)  {  int i, temp;  radix\_node\_t\* node, \* child;  node = t->root;  for (i = 0; i < radix\_tree\_height; i++) {  temp = CHECK\_BITS(key, i);  if (!node->child[temp]) {  child = radix\_node\_alloc(t);  if (!child)return NULL;  child->parent = node;  node->child[temp] = child;  node = node->child[temp];  }  else {  node = node->child[temp];  }  }  if (node->value == value)return RADIX\_INSERT\_VALUE\_SAME;  if (node->value != NULL)return RADIX\_INSERT\_VALUE\_OCCUPY;  node->value = value;  return 0;  }    //由于插入时会创建很多节点，为了提高删除速度这里只会删除最底层的指定节点  int radix\_tree\_delete(radix\_tree\_t\* t, uint32 key)  {  radix\_node\_t\* node = t->root, \* par;  int i = 0, temp = 0;  if (node == NULL)return RADIX\_DELETE\_ERROR;  do {  temp = CHECK\_BITS(key, i++);  node = node->child[temp];  } while (node && i < radix\_tree\_height);  //node为储存value的节点，在父节点中将此节点的链接置空，  //然后清空value后将此节点加入free中  if (node == NULL)return RADIX\_DELETE\_ERROR;  par = node->parent;  par->child[temp] = NULL;  node->value = NULL;  node->child[0] = NULL;  node->child[1] = NULL;  node->child[2] = NULL;  node->child[3] = NULL;  node->parent = t->free->parent;  t->free->parent = node;  return 0;  }  //打印函数，会打印出所有叶节点储存的值  void radix\_print(radix\_node\_t\* node)  {  if (node == NULL)return;  if (node->value != NULL)  printf("%x\n", node->value);  radix\_print(node->child[0]);  radix\_print(node->child[1]);  radix\_print(node->child[2]);  radix\_print(node->child[3]);  }  //节点查找函数  //key为索引，返回叶节点被查找到的值  ptr\_t radix\_tree\_find(radix\_tree\_t\* t, uint32 key)  {  int i = 0, temp;  radix\_node\_t\* node;  node = t->root;  while (node && i < radix\_tree\_height) {  temp = CHECK\_BITS(key, i++);  node = node->child[temp];  }  if (node == NULL)return NULL;  return node->value;  }  上面仍有值得改进的地方，比如节点结构，每一个节点不论是内部节点还是叶节点都有储存数据的value有点浪费空间。删除操作还可以优化，多删一些没用的节点，可以节省空间，虽然这样会增加删除操作占用的时间，等等。 |

参考3：GitHub：<https://github.com/ytakano/radix_tree>

# 2. C++ 实现的并行 K-Means 算法，

参考：<https://github.com/StopPointTeam/K-Means>

参考2：<https://blog.csdn.net/github_39533414/article/details/84895166>

# 3.实现STL算法：

参考1，GitHub：<https://github.com/Maxmengt/MicroSTL>

参考2：<https://github.com/SilverMaple/STLSourceCodeNote>

# 4、实现内存泄漏检测工具，

#### 参考1：

|  |
| --- |
| **【内存】C/C++ 自己实现简单的内存泄漏检测工具（重载new/free)**  **目录**  ​[​C++语言实现​](https://blog.51cto.com/liangchaoxi/4051751#C%2B%2B%E8%AF%AD%E8%A8%80%E5%AE%9E%E7%8E%B0)​  ​[​C语言实现​](https://blog.51cto.com/liangchaoxi/4051751#C%E8%AF%AD%E8%A8%80%E5%AE%9E%E7%8E%B0)​  学习练习用，真正检测内存泄漏最好还是 借助专门的工具。  **C++语言实现**  思路：  1.重载operator new/new[ ] 与 operator delete/delete[ ], 并借助双向链表结构(带头节点)管理内存，new的时候将 内存信息存入链表，delete的时候将内存信息踢出链表，程序结束后，查看链表剩余多少节点判断内存泄漏。  ​    new操作符是由C++语言内建的, 就像sizeof那样, 不能改变意义, 总是做相同的事情:   1. 调用operator new (sizeof(A)) 2. 调用A:A() 3. 返回指针   第一: 它分配足够的内存, 用来放置某类型的对象.  第二: 它调用一个构造函数, 为刚才分配的内存中的那个对象设定初始值。  第三: 对象被分配了空间并构造完成, 返回一个指向该对象的指针  第二、第三，new operator(即 new 操作符)总是做这两件事，无论如何你是不能改变其行为。  能够改变的是用来容纳对象的那块内存的分配行为, new operator(new)调用某个函数, 执行必要的内存分配动作, 你可以重写或者重载那个函数, 改变其行为. 这个函数名称就叫operator new 。  函数 operator new 通常声明如下:  void \* operator new (size\_t size);   * 1.   其返回类型void\*. 即返回一个指针, 指向一块原始的, 未设置初始值的内存。  函数中的size\_t参数表示需要分配多少内存, 你可以将operator new 重载, 加上额外的参数, 但第一个参数类型必须总是size\_t.  可以这样理解: new int -> new(sizeof(int)) -> operator new(sizeof(int)/\*即size\_t size\*/)->重载.  不能改变关键字new的行为 但我们能重载operator new( size\_t size )  注意: operator new( size\_t size )中的参数size 是new 计算的. 不用我们自己计算. 我们重载时, 只需要开辟 size个字节的内存大小即可    LeakDetector.h:  // 注意, 我们的头文件是要被包含进被测试的.cpp 的, 所以头文件中不要出现"多余的"代码及库文件, 以免影响被测文件 #ifndef LEAK\_DETECTOR\_H\_ #define LEAK\_DETECTOR\_H\_ // 有个小技巧: C/C++库中标准的头文件宏定义是这种形式: \_STDIO\_H( 标准规定保留下划线作前缀 ) // 所以平时我们为了避免自己定义的宏意外地与标准头文件定义的宏发生冲突, 我们使用下划线作后缀, 并且不用下划线作前缀    // 重载版本: operator new/new[]( ), operator delete/delete[]( ) 的声明 void\* operator new(size\_t size, const char\* file, size\_t line); void\* operator new[](size\_t size, const char\* file, size\_t line); // 注意到, 上面我们重载的函数中, 第一个参数和第三个参数的类型是size\_t // 其中第一个参数size为 sizeof的返回值, 所以为size\_t类型 // 第三个参数的含义为 行号, 是我们重载 operator new/new[]( )后自己加的参数, 此处也可以用 //unsigned int. 但最好用 size\_t. 原因是size\_t的可移植性好. 理由见上面链接 void operator delete(void\* ptr); void operator delete[](void\* ptr);   // 这个宏在LeakDetector.cpp中定义, 使得编译时 源码中的new被替换为  //new( \_\_FILE\_\_, \_\_LINE\_\_ ),源码就变成了使用我们自己的重载版本 //operator new/new[]( size\_t size, const char\* file, size\_t line )  #ifndef NEW\_OVERLOAD\_IMPLEMENTATION\_ #define new new( \_\_FILE\_\_, \_\_LINE\_\_ ) // 预定义宏:  // \_\_FILE\_\_(两个下划线): 代表当前源代码文件名 // \_\_LINE\_\_(两个下划线): 代表当前源代码文件中的行号 #endif   class LeakDetector { public: // LeakDetector.cpp和被测试的.cpp都会包 LeakDetector.h头文件 // 因此两个源文件中会创建两个静态LeakDetector类对象 exitCounter (两个静态类对象名相同， //但是它们的链接属性均为内链接(只在当前源文件有效), 因此不会重定义), 如果此时两个对析 //构, 会调用两次析构函数, 调用两次内存泄漏检测函数. 而我们的预期是只调用一次内存泄漏 //检测函数. 所以我们声明一个所有类对象共享的静态变量来实现我们的目的  static size\_t \_callCount;  LeakDetector() { ++\_callCount; } ~LeakDetector() { if (0 == --\_callCount) \_LeakDetector(); }  private: void \_LeakDetector(); }; //为什么要设计 callCount? callCount 保证了我们的 LeakDetector 只调用了一次。 // 静态对象 static LeakDetector exitCounter;    #endif  **LeakDetector.cpp:**  // 这个宏保证 LeakDetector.cpp 中的new 不会被LeakDetector.h中的 宏替换 替换掉 #define NEW\_OVERLOAD\_IMPLEMENTATION\_   #include <iostream> //cout  #include <cstring> //strlen 和 strcpy #include "LeakDetector.h" #pragma warning(disable:4996)  // 初始化 LeakDetector类中定义的静态变量 size\_t LeakDetector::\_callCount = 0;   // 我们使用带头节点的双向链表来手动管理内存申请与释放, 头节点的\_prev指向最后一个结点, \_next指向第一个结点 // 双向链表结构 typedef struct MemoryList { struct MemoryList\* \_prev; struct MemoryList\* \_next; size\_t \_size; // operator new( )申请的内存大小 bool \_isArray; // 是否为申请数组(即使用operator new[]( ) 而不是 operator new( )) char\* \_file; // 如果有, 存储存在内存泄漏文件的文件信息 size\_t \_line; // 存储存在内存泄漏位置的行号 } MemoryList;  // 创建一个头结点, 它的前后指针均初始化为指向自己(插入、删除双向链表中结点 和 \_LeakDetector( )函数中遍历双向链表时, 这样初始化的作用就体现出来了)。使用静态变量使其只在本文件内有效 // 我们只使用这个头节点的 \_prev 和 \_next 成员 static MemoryList memoryListHead = { &memoryListHead, &memoryListHead, 0, false, NULL, 0 };   // 保存未释放的内存大小 static size\_t memoryAllocated = 0;   // 对双向链表采用头插法分配内存 void\* AllocateMemory(size\_t size, bool array, const char\* file, size\_t line) { // 我们需要为我们管理内存分配的 MemoryList结点 也申请内存 // 计算新的大小 size\_t newSize = size + sizeof(MemoryList);  // 把接收到的地址强转为 MemoryList\*, 以便我们后续操作 // 由于重载了new, 所以我们使用 malloc 来申请内存 MemoryList\* newElem = (MemoryList\*)malloc(newSize);  // 更新MemoryList结构成员的值 newElem->\_prev = &memoryListHead; newElem->\_next = memoryListHead.\_next; newElem->\_size = size; // 注意, 此处为size而不是newSize. 因为我们管理记录的是 new申请的内存, 验证它是否未释放, 存在内存泄漏问题. 申请 newSize的内存(为 MemoryList结点多申请出的内存), 只是为了实现手动管理内存所必须, 这个内存我们一定会释放, 不需关注. 所以保存 时用size而不是newSize newElem->\_isArray = array;  // 如果有文件信息, 则保存下来 if (NULL != file) { newElem->\_file = (char\*)malloc(strlen(file) + 1); strcpy(newElem->\_file, file);  } else newElem->\_file = NULL;  // 保存行号 newElem->\_line = line;  // 更新双向链表结构 memoryListHead.\_next->\_prev = newElem; memoryListHead.\_next = newElem;  // 更新未释放的内存数 // 我们管理的只是 new申请的内存. 为memoryListHead结点多申请的内存,和为保存文件信息多申请内存无关, 这些内存我们一定会释放, 所以这里只记录size memoryAllocated += size;  // 返回new 申请的内存地址 // 将newElem强转为char\* 类型(保证指针+1时每次加的字节数为1) + memoryListHead所占用字节数( 总共申请的newSize字节数 减去memoryListHead结点占用的字节数, 即为new申请的字节数 ) return (char\*)newElem + sizeof(memoryListHead); }  // 对双向链表采用头删法手动管理释放内存 // 注意: delete/delete[]时 我们并不知道它操作的是双向链表中的哪一个结点 void DeleteMemory(void\* ptr, bool array) { // 注意, 堆的空间自底向上增长. 所以此处为减 MemoryList\* curElem = (MemoryList\*)((char\*)ptr - sizeof(MemoryList));  // 如果 new/new[] 和 delete/delete[] 不匹配使用. 直接返回 if (curElem->\_isArray != array) return;  // 更新链表结构 curElem->\_next->\_prev = curElem->\_prev; curElem->\_prev->\_next = curElem->\_next;  // 更新memoryAllocated值 memoryAllocated -= curElem->\_size;  // 如果curElem->\_file不为NULL, 释放保存文件信息时申请的内存 if (NULL != curElem->\_file) free(curElem->\_file);  // 释放内存 free(curElem); }   // 重载new/new[]运算符 void\* operator new(size\_t size, const char\* file, size\_t line) { return AllocateMemory(size, false, file, line); }  void\* operator new[](size\_t size, const char\* file, size\_t line) { return AllocateMemory(size, true, file, line); }  // 重载delete/delete[]运算符 void operator delete(void\* ptr) { DeleteMemory(ptr, false); }  void operator delete[](void\* ptr) { DeleteMemory(ptr, true); }   // 我们定义的最后一个静态对象析构时调用此函数, 判断是否有内存泄漏, 若有, 则打印出内存泄漏信息 void LeakDetector::\_LeakDetector() { if (0 == memoryAllocated) { std::cout << "恭喜, 您的代码不存在内存泄漏!" << std::endl; return;  }  // 存在内存泄漏 // 记录内存泄漏次数 size\_t count = 0;  // 若不存在内存泄漏, 则双向链表中应该只剩下一个头节点 // 若存在内存泄漏, 则双向链表中除头节点之外的结点都已泄露，个数即内存泄漏次数 MemoryList\* ptr = memoryListHead.\_next; while ((NULL != ptr) && (&memoryListHead != ptr)) { if (true == ptr->\_isArray) std::cout << "new[] 空间未释放, "; else std::cout << "new 空间未释放, ";  std::cout << "指针: " << ptr << " 大小: " << ptr->\_size;  if (NULL != ptr->\_file) std::cout << "kkkk"; //std::cout << " 位于 " << ptr->\_file << " 第 " << ptr->\_line << " 行"; else std::cout << " (无文件信息)";  std::cout << std::endl;  ptr = ptr->\_next; ++count;  }  std::cout << "存在" << count << "处内存泄露, 共包括 " << memoryAllocated << " byte." << std::endl; return; }  **test.cpp:**  #include "LeakDetector.h"   int main() {  // 忘记释放指针 b 申请的内存, 从而导致内存泄露 int \*a = new int; int \*b = new int[12];  delete a;  return 0;  }    备注：按原文写代码报错  所以将void\* operator new( size\_t size, char\* file, size\_t line ); 改成void\* operator new( size\_t size, const char\* file, size\_t line );    【内存】C/C++  自己实现简单的内存泄漏检测工具（重载new/free)_头文件    **C语言实现**  原理：申请一个链表，malloc 一个内存时，将申请的信息 插入到链表中，free 内存时，将内存信息从链表中移除，最后推出程序的时候，打印内存信息。    这个内存泄漏检测工具很简单，只能检测同一个模块，同一个线程中发送的内存泄漏，对于在编写代码过程中的代码调试有一定的帮助。如果要在集成测试或功能测试中检测内存泄漏，还需借助专门的工具。    源码：  memcheck.c  2. 定义保存内存信息的单向链表  /\*\* \* 定义链表节点，表示一个内存泄漏信息 \*/ typedef struct \_mem\_node { void \*ptr; // 泄漏内存地址 size\_t block; // 泄漏内存大小 size\_t line; // 泄露发生的代码行 char \*filename; // 泄漏发生的文件名 struct \_mem\_node \*next; // 下一个节点指针 } mem\_node;  // 定义指向头节点的指针 mem\_node \*head = NULL;  3. 用于将节点加入单项链表的函数  /\*\* \* 产生一个节点并加入链表 \* @param ptr 分配的内存地址 \* @param block 分配的内存单元大小 \* @param line 代码行号 \* @param filename 文件名称 \*/ static void mem\_node\_add(void \*ptr, size\_t block, size\_t line, char \*filename) { // 产生节点 mem\_node \*node = malloc(sizeof(mem\_node)); node->ptr = ptr; node->block = block; node->line = line; node->filename = filename; node->next = NULL;  // 加入链表头节点 if (head)  { node->next = head; head = node;  } else head = node; }  4. 从单项链表中删除节点的函数  /\*\* \* 从链表中删除一个节点 \* @param ptr 分配的内存地址 \*/ static void mem\_node\_remove(void \*ptr) { // 判断头节点是否存在 if (head)  { // 处理头节点 if (head->ptr == ptr)  { // 获取头节点的下一个节点 mem\_node \*pn = head->next; // 删除头节点 free(head); // 令头节点指针指向下一个节点 head = pn;  } else // 判断链表是否为空  { // 指向节点的指针 mem\_node \*pn = head->next; // 指向前一个节点的指针 mem\_node \*pc = head; // 遍历所有节点 while (pn)  { // 获取指向下一个节点的指针 mem\_node \*pnext = pn->next; if (pn->ptr == ptr)  { pc->next = pnext; // 删除当前节点 free(pn);  } else pc = pc->next; pn = pnext;  }  }  } }  5. 显示内存泄露信息报告  /\*\* \* 显示内存泄漏信息 \*/ void show\_block() { if (head)  { // 保存总内存泄漏数量 size\_t total = 0; // 指向头节点的指针 mem\_node \*pn = head;  // 输出标题 puts("\n\n-------------------------------内存泄漏报告------------------------------------\n");  // 遍历链表 while (pn)  { mem\_node \*pnext = pn->next; // 处理文件名 char \*pfile = pn->filename, \*plast = pn->filename; while (\*pfile)  { // 找到\字符 if (\*pfile == '\\') plast = pfile + 1; // 获取\字符的位置 pfile++;  } // 输出内存泄漏信息 printf("位置:%s(%d), 地址:%p(%dbyte)\n", plast, pn->line, pn->ptr, pn->block); // 累加内存泄漏总量 total += pn->block; // 删除链表节点 free(pn); // 指向下一个节点 pn = pnext;  } printf("总计内存泄漏:%dbyte\n", total);  } }  6. 定义调试用malloc函数  /\*\* \* 用于调试的malloc函数 \* @param elem\_size 分配内存大小 \* @param filename 文件名称 \* @param line 代码行号 \*/ void \*dbg\_malloc(size\_t elem\_size, char \*filename, size\_t line) { void \*ptr = malloc(elem\_size); // 将分配内存的地址加入链表 mem\_node\_add(ptr, elem\_size, line, filename); return ptr; }  7. 定义调试用的calloc函数  /\*\* \* 用于调试的calloc函数 \* @param count 分配内存单元数量 \* @param elem\_size 每单元内存大小 \* @param filename 文件名称 \* @param line 代码行号 \*/ void \*dbg\_calloc(size\_t count, size\_t elem\_size, char \*filename, size\_t line) { void \*ptr = calloc(count, elem\_size); // 将分配内存的地址加入链表 mem\_node\_add(ptr, elem\_size \* count, line, filename); return ptr; }  8. 定义调试用的free函数  /\*\* \* 用于调试的free函数 \* @param ptr 要释放的内存地址 \*/ void dbg\_free(void \*ptr) { free(ptr); // 从链表中删除节点 mem\_node\_remove(ptr); }     上述代码应包含在一个C文件中（例如memcheck.c），完成上述步骤，就可以利用这一组函数来检测内存泄露了，需要定义如下头文件memcheck.h，  1. 先取消malloc，free和calloc这几个标识符的定义  2.然后#define 重定义 malloc、free和calloc 为我们前面定义的函数  然后工程内其他.c代码文件包含该头文件即可。  memcheck.h  #ifndef \_MEM\_CHECK\_H #define \_MEM\_CHECK\_H  #include <stdlib.h>    // 取消malloc, calloc, free的宏定义 #undef malloc #undef calloc #undef free    // instead of malloc #define malloc(s) dbg\_malloc(s, \_\_FILE\_\_, \_\_LINE\_\_)  // instead of calloc #define calloc(c, s) dbg\_calloc(c, s, \_\_FILE\_\_, \_\_LINE\_\_)  // instead of free #define free(p) dbg\_free(p)  /\*\* \* allocation memory \*/ void \*dbg\_malloc(size\_t elem\_size, char \*filename, size\_t line);  /\*\* \* allocation and zero memory \*/ void \*dbg\_calloc(size\_t count, size\_t elem\_size, char \*filename, size\_t line);  /\*\* \* deallocate memory \*/ void dbg\_free(void \*ptr);  /\*\* \* show memory leake report \*/ void show\_block();  #endif // \_MEM\_CHECK\_H  使用的时候只需要包含上述头文件（例如命名为memcheck.h），并将上述C文件引入到项目中即可。测试代码如下：  #ifdef DEBUG #include "memcheck.h" #endif  int main() { int\* p;  #ifdef DEBUG atexit(show\_block); // 在程序结束后显示内存泄漏报告 #endif // DEBUG // 分配内存并不回收，显示内存泄漏报告 p = (int\*)malloc(1000);  return 0; } |

#### 参考2：

|  |  |
| --- | --- |
| |  | | --- | | **一个小项目 --- C++实现内存泄漏检查器**    https://img-blog.csdn.net/20170817003512726?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvc3NvcHAyNA==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  先贴出代码:  .h:  *// 注意, 我们的头文件是要被包含进被测试的.cpp 的, 所以头文件中不要出现"多余的"代码及库文件, 以免影响被测文件*  #ifndef LEAK\_DETECTOR\_H\_  #define LEAK\_DETECTOR\_H\_  *// 有个小技巧: C/C++库中标准的头文件宏定义是这种形式: \_STDIO\_H( 标准规定保留下划线作前缀 )*  *// 所以平时我们为了避免自己定义的宏意外地与标准头文件定义的宏发生冲突, 我们使用下划线作后缀, 并且不用下划线作前缀*        *// 重载版本: operator new/new[]( ), operator delete/delete[]( ) 的声明*  void\* operator new( size\_t size, char\* file, size\_t line );  void\* operator new[]( size\_t size, char\* file, size\_t line );  *// 注意到, 上面我们重载的函数中, 第一个参数和第三个参数的类型是size\_t*  *// 其中第一个参数size为 sizeof的返回值, 所以为size\_t类型*  *// 第三个参数的含义为 行号, 是我们重载 operator new/new[]( )后自己加的参数, 此处也可以用unsigned int. 但最好用 size\_t. 原因是size\_t的可移植性好. 理由见上面链接*  void operator delete( void\* ptr );  void operator delete[]( void\* ptr );      *// 这个宏在LeakDetector.cpp中定义, 以防止LeakDetector.cpp中, 我们自己重载的 operator new/new[]( ) 被宏替换. 而这个宏在被测试文件中未定义(我们除过在被测试文件包含LeakDetector.h头文件外, 不改变被测试文件的代码), 所以 替换被测试文件new运算符, 传进两个参数 文件名 和 行号 使用我们自己的重载版本operator new/new[]( size\_t size, char\* file, size\_t line )*  #ifndef NEW\_OVERLOAD\_IMPLEMENTATION\_  #define new new( \_\_FILE\_\_, \_\_LINE\_\_ )  *// 预定义宏:*  *// \_\_FILE\_\_(两个下划线): 代表当前源代码文件名的字符串文字(我们用这个宏获得存在内存泄漏文件的文件名)*  *// \_\_LINE\_\_(两个下划线): 代表当前源代码文件中的行号的整数常量(我们用这个宏获得存在内存泄漏文件内存泄漏的行号)*  #endif      class LeakDetector{  public:  *// LeakDetector.cpp和被测试的.cpp都会包 LeakDetector.h头文件*  *// 因此两个源文件中会创建两个静态LeakDetector类对象 exitCounter (两个静态类对象同名, 但是它们的链接属性均为内链接(只在当前源文件有效), 因此不会重定义), 如果此时两个对象析构, 会调用两次析构函数, 调用两次内存泄漏检测函数. 而我们的预期是只调用一次内存泄漏检测函数. 所以我们声明一个所有类对象共享的静态变量来实现我们的目的*  static size\_t \_callCount;    LeakDetector( ){ ++\_callCount; }  ~LeakDetector( ){ if(0 == --\_callCount) \_LeakDetector( ); }    private:  void \_LeakDetector( );  };    *// 静态对象*  static LeakDetector exitCounter;        #endif    .cpp:  *// 这个宏保证 LeakDetector.cpp 中的new 不会被LeakDetector.h中的 宏替换 替换掉*  #define NEW\_OVERLOAD\_IMPLEMENTATION\_      #include <iostream> *//cout*  #include <cstring> *//strlen 和 strcpy*  #include "LeakDetector.h"      *// 初始化 LeakDetector类中定义的静态变量*  size\_t LeakDetector::\_callCount = 0;      *// 我们使用带头节点的双向链表来手动管理内存申请与释放, 头节点的\_prev指向最后一个结点, \_next指向第一个结点*  *// 双向链表结构*  typedef struct MemoryList{  struct MemoryList\* \_prev;  struct MemoryList\* \_next;  size\_t \_size; *// operator new( )申请的内存大小*  bool \_isArray; *// 是否为申请数组(即使用operator new[]( ) 而不是 operator new( ))*  char\* \_file; *// 如果有, 存储存在内存泄漏文件的文件信息*  size\_t \_line; *// 存储存在内存泄漏位置的行号*  } MemoryList;    *// 创建一个头结点, 它的前后指针均初始化为指向自己(插入、删除双向链表中结点 和 \_LeakDetector( )函数中遍历双向链表时, 这样初始化的作用就体现出来了)。使用静态变量使其只在本文件内有效*  *// 我们只使用这个头节点的 \_prev 和 \_next 成员*  static MemoryList memoryListHead = { &memoryListHead, &memoryListHead, 0, false, NULL, 0 };      *// 保存未释放的内存大小*  static size\_t memoryAllocated = 0;      *// 对双向链表采用头插法分配内存*  void\* AllocateMemory( size\_t size, bool array, char\* file, size\_t line){  *// 我们需要为我们管理内存分配的 MemoryList结点 也申请内存*  *// 计算新的大小*  size\_t newSize = size + sizeof( MemoryList );    *// 把接收到的地址强转为 MemoryList\*, 以便我们后续操作*  *// 由于重载了new, 所以我们使用 malloc 来申请内存*  MemoryList\* newElem = (MemoryList\*)malloc(newSize);    *// 更新MemoryList结构成员的值*  newElem->\_prev = &memoryListHead;  newElem->\_next = memoryListHead.\_next;  newElem->\_size = size; *// 注意, 此处为size而不是newSize. 因为我们管理记录的是 new申请的内存, 验证它是否未释放, 存在内存泄漏问题. 申请 newSize的内存(为 MemoryList结点多申请出的内存), 只是为了实现手动管理内存所必须, 这个内存我们一定会释放, 不需关注. 所以保存 时用size而不是newSize*  newElem->\_isArray = array;    *// 如果有文件信息, 则保存下来*  if ( NULL != file ){  newElem->\_file = (char\*)malloc(strlen(file) + 1);  strcpy( newElem->\_file, file );  }  else  newElem->\_file = NULL;    *// 保存行号*  newElem->\_line = line;    *// 更新双向链表结构*  memoryListHead.\_next->\_prev = newElem;  memoryListHead.\_next = newElem;    *// 更新未释放的内存数*  *// 我们管理的只是 new申请的内存. 为memoryListHead结点多申请的内存,和为保存文件信息多申请内存无关, 这些内存我们一定会释放, 所以这里只记录size*  memoryAllocated += size;    *// 返回new 申请的内存地址*  *// 将newElem强转为char\* 类型(保证指针+1时每次加的字节数为1) + memoryListHead所占用字节数( 总共申请的newSize字节数 减去memoryListHead结点占用的字节数, 即为new申请的字节数 )*  return (char\*)newElem + sizeof(memoryListHead);  }    *// 对双向链表采用头删法手动管理释放内存*  *// 注意: delete/delete[]时 我们并不知道它操作的是双向链表中的哪一个结点*  void DeleteMemory( void\* ptr, bool array ){  *// 注意, 堆的空间自底向上增长. 所以此处为减*  MemoryList\* curElem = (MemoryList\*)( (char\*)ptr - sizeof(MemoryList) );    *// 如果 new/new[] 和 delete/delete[] 不匹配使用. 直接返回*  if ( curElem->\_isArray != array )  return;    *// 更新链表结构*  curElem->\_next->\_prev = curElem->\_prev;  curElem->\_prev->\_next = curElem->\_next;    *// 更新memoryAllocated值*  memoryAllocated -= curElem->\_size;    *// 如果curElem->\_file不为NULL, 释放保存文件信息时申请的内存*  if ( NULL != curElem->\_file )  free( curElem->\_file );    *// 释放内存*  free( curElem );  }      *// 重载new/new[]运算符*  void\* operator new( size\_t size, char\* file, size\_t line ){  return AllocateMemory( size, false, file, line );  }    void\* operator new[]( size\_t size, char\* file, size\_t line ){  return AllocateMemory( size, true, file, line );  }    *// 重载delete/delete[]运算符*  void operator delete( void\* ptr ){  DeleteMemory( ptr, false );  }    void operator delete[]( void\* ptr ){  DeleteMemory( ptr, true );  }      *// 我们定义的最后一个静态对象析构时调用此函数, 判断是否有内存泄漏, 若有, 则打印出内存泄漏信息*  void LeakDetector::\_LeakDetector( ){  if ( 0 == memoryAllocated ){  std::cout << "恭喜, 您的代码不存在内存泄漏!" << std::endl;  return;  }    *// 存在内存泄漏*  *// 记录内存泄漏次数*  size\_t count = 0;    *// 若不存在内存泄漏, 则双向链表中应该只剩下一个头节点*  *// 若存在内存泄漏, 则双向链表中除头节点之外的结点都已泄露，个数即内存泄漏次数*  MemoryList\* ptr = memoryListHead.\_next;  while ( (NULL != ptr) && (&memoryListHead != ptr) ){  if ( true == ptr->\_isArray )  std::cout << "new[] 空间未释放, ";  else  std::cout << "new 空间未释放, ";    std::cout << "指针: " << ptr << " 大小: " << ptr->\_size;    if ( NULL != ptr->\_file )  std::cout << " 位于 " << ptr->\_file << " 第 " << ptr->\_line << " 行";  else  std::cout << " (无文件信息)";    std::cout << std::endl;    ptr = ptr->\_next;  ++count;  }    std::cout << "存在" << count << "处内存泄露, 共包括 "<< memoryAllocated << " byte." << std::endl;  return;  }    test.cpp:  #include "LeakDetector.h"      int main() {    *// 忘记释放指针 b 申请的内存, 从而导致内存泄露*  int \*a = new int;  int \*b = new int[12];    delete a;    return 0;    }    思路:  1.内存泄露产生于 new/new[] 操作进行后没有执行 delete/delete[]  2.最先被创建的对象, 其析构函数是最后执行的  解决方法:  1.重载operator new/new[ ] 与 operator delete/delete[ ], 并借助双向链表结构(带头节点)我们自己手动管理内存  2.创建一个静态对象, 在程序退出时才调用这个静态对象的析构函数( 在析构函数中, 我们调用内存泄漏检测函数 ), 这就得保证, 我们的静态对象必须先于被测文件的静态对象创建(如果有), 这样我们的静态对象才会最后一个析构(必须保证最后一个析构, 以免发生问题( 如: 假设被测文件也有一个静态对象, 且静态对象申请了空间,  被测文件的静态对象在我们的静态对象析构后析构, 这样内存泄漏检测就会不准确 )), 所以被测文件必须第一个包含我们的 LeakDetector.h 头文件保证我们的静态对象第一个创建.  这样两个步骤的好处在于: 无需修改原始代码的情况下, 就能进行内存检查. 这同时也是我们希望看到的    既然我们已经重载了new/new[], delete/delete[]操作符, 那么我们很自然就能想到通过手动管理内存申请和释放, 如果我们delete/delete[]时没有将申请的内存全部释放完毕, 那么一定发生了内存泄露. 接下来一个问题就是，使用什么结构来实现手动管理内存:  不妨使用双向链表来实现内存泄露检查. 原因在于, 对于内存检查器来说, 并不知道实际代码在什么时候会需要申请内存空间, 所以使用线性表并不够合理, 一个动态的结构（链表）是非常便捷的. 而我们在删除内存检查器中的对象时, 需要更新整个结构, 对于单向链表来说, 也是不够便利的  我们的内存泄漏检测函数会在静态对象析构时被调用, 这时候其他所有申请的对象都已经完成析构, 这时, 如果我们的双向链表除头节点外仍有结点, 那么一定是泄露且尚未释放的内存, 所以我们只需要遍历双向链表即可得到我们需要的结果    new操作符是由C++语言内建的, 就像sizeof那样, 不能改变意义, 总是做相同的事情:   1. 调用operator new (sizeof(A)) 2. 调用A:A() 3. 返回指针   第一: 它分配足够的内存, 用来放置某类型的对象.  第二: 它调用一个构造函数, 为刚才分配的内存中的那个对象设定初始值。  第三: 对象被分配了空间并构造完成, 返回一个指向该对象的指针  new operator(即 new 操作符)总是做这两件事，无论如何你是不能改变其行为。  能够改变的是用来容纳对象的那块内存的分配行为, new operator(new)调用某个函数, 执行必要的内存分配动作, 你可以重写或者重载那个函数, 改变其行为. 这个函数名称就叫operator new 。  函数 operator new 通常声明如下:  void \* operator new (size\_t size);  其返回类型void\*. 即返回一个指针, 指向一块原始的, 未设置初始值的内存。  函数中的size\_t参数表示需要分配多少内存, 你可以将operator new 重载, 加上额外的参数, 但第一个参数类型必须总是size\_t.    可以这样理解: new int -> new(sizeof(int)) -> operator new(sizeof(int)/\*即size\_t size\*/)->重载.  不能改变关键字new的行为 但我们能重载operator new( size\_t size )  注意: operator new( size\_t size )中的参数size 是new 计算的. 不用我们自己计算. 我们重载时, 只需要开辟 size个字节的内存大小即可    同样, delete关键字做两件事:  第一: 调用对象析构函数  第二: 调用operator delete( )释放对象所占用的内存  string\* pte = new string("KobeBryant");  delete ptr -> ptr->~string -> operator delete( ptr )  void operator delete(void\* memoryToBeDeallocated);  同样, 我们能做的也只有重载operator delete( void\* ) 来自己手动管理内存释放.    接下来我们来说下new [] 和 delete []  :  AA为4字节: new AA[10] -> 并不是只开40个字节大小, 而要在对象数组的大小上加上一个额外数据，用于编译器区分对象数组大小:  sizeof(size\_t) + 4 \* sizeof(AA)  这个多出来的内存用来存申请数目, 由编译器在[operator](https://so.csdn.net/so/search?q=operator&spm=1001.2101.3001.7020) new之后, new expression前设置, 你最终拿到的地址实际上和operator new分到的地址并不一样  编译器通过这个数目(对象数组中元素个数), 来确定调用几次对象的构造函数.  当然,这时, 调用几次构造函数已经和operator new[]无关了. operator new[]根本无法判断到底分配了几个对象，它只知道一共有多少字节  所以, 我们重载时不用关注这个步骤.  这是编译器要做的.  在实现层，两者就是一样的，通常new[] 直接调用new实现  而operator new 和 operator new[] 的参数 size也是由编译器传给我们的, 我们不用自己去计算要多开的 sizeof(size\_t)大小.    总结下: 我们要重载的 operator new 和·operator new[] 只需要做一件事: 调用malloc 开编译器传给我们的 size 个字节即可.  同样delete[ ]:  1. 因为之前保存过对象个数, 它调用对象个数次析构函数  2. 调用 operator delete[ ] 对内存进行释放(释放对象占用的内存和多开的保存对象个数的内存)    operator delete[]( void\* ptr ) 不管对象个数的事, 它也不知道这些. 它只需要做一件事: 释放ptr指向内存即可. 通常调用 operator delete()实现  我们重载的operator delete[] 也实现同样功能.    总体总结下:  new做了两件事: 1. 调用operator new分配空间。 2. 调用构造函数初始化对象。 delete也做了两件事: 1. 调用析构函数清理对象 2. 调用operator delete释放空间 new[N]: 1. 调用operator new分配空间。 2. 调用N次构造函数分别初始化每个对象。 delete[]: 1. 调用N次析构函数清理对象。 2. 调用operator delete释放空间。  1. operator new/operator delete operator new[]/operator delete[] 和 malloc/free用法一样。 2. 他们只负责分配空间/释放空间，不会调用对象构造函数/析构函数来初始化/清理对象。 3. 实际operator new和operator delete只是malloc和free的一层封装。    对了, 并不是所有的类型new []都会多开sizeof(size\_t)个字节保存对象个数:  为什么要保存对象个数呢?  因为编译器得知道它要调用几次构造函数和析构函数:  问题来了, 内置类型(int, char等)需要调用析构函数吗:  答案是不一定, 参考这篇文章: http://blog.csdn.net/mind\_v/article/details/70740354  所以, 自定义类型, 一定会多开. 内置类型: 不一定.      https://img-blog.csdn.net/20170817002305547?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvc3NvcHAyNA==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  :http://www.cnblogs.com/fly1988happy/archive/2012/04/26/2471099.html  :http://blog.csdn.net/wudaijun/article/details/9273339#t7  :https://www.zhihu.com/question/25497587      通过这个项目我获得的知识点(有些知识点是之前学过但遗忘了进行复习):  0.更深入了解了 C++ new/new[]和delete/delete[]背后所做的事  1.LeakDetector.cpp 和 被测试文件.cpp 都包含了 LeakDetector.h. 会造成我们这个类重定义吗?  不会:  在一个给定的源文件中，一个类只能被定义一次。如果在多个文件中定义一个类，那么每个文件中的定义必须是完全相同的  因为这遵守“单一定义规则”(One-Definition Rule, ODR)。根据此规则， 如果对同一个类的两个定义完全相同且出现在不同编译单位，会被当作同一个定义。  这里头文件分别被两个不同的编译单位（LeakDetector.cpp, test.cpp）包含，满足ODR规则，会被当作同一个定义。 所以不会有冲突。  此外，模板和inline函数也适用此规则。http://blog.csdn.net/baoxiaofeicsdn/article/details/48338515  2.\_\_FILE\_\_, \_\_LINE\_\_两个预定义宏  3.带头节点双向链表结构(头插, 和删除任意节点)  4.变量存储持续性，作用域， 链接性  5.堆栈生长规律  6.头文件中通常包含的内容: 函数原型, 使用#define/const 定义的符号常量, 结构声明, 类声明, 模板声明, 内联函数  7.复习了条件编译  8.类中静态数据成员与静态类方法:  静态数据成员: 属于所有对象而不是特定对象, 为了实现共享数据  静态类方法: 通过类名::FunName( )调用, 不能通过对象调用.  它们不能通过对象调用是因为它们没有隐含的this指针  http://www.cnblogs.com/ppgeneve/p/5091794.html  http://blog.csdn.net/kerry0071/article/details/25741425/  9.复习了多个.cpp源文件 如何编译链接成 一个可执行程序过程  10.typedef struct结构体时一些规则. 还有typedef 和 #define 区别  11.unsigned int / unsigned long 和 size\_t 故事:  size\_t平台移植性更好:  http://blog.csdn.net/lemoncyb/article/details/12012987  http://jeremybai.github.io/blog/2014/09/10/size-t  12.静态对象:可以调用它的所有成员, 包括非静态成员. 但静态函数智能调用静态成员.  静态对象何时创建, 何时销毁:http://blog.csdn.net/shltsh/article/details/45959493 | |

# 5.c++实现高性能内存池

#### 参考：

复制自GitHub：<https://github.com/ForrestPi/Blog/blob/master/Code/C%2B%2B%20%E5%AE%9E%E7%8E%B0%E9%AB%98%E6%80%A7%E8%83%BD%E5%86%85%E5%AD%98%E6%B1%A0.md>

|  |
| --- |
| **C++ 实现高性能内存池**  在 C/C++ 中，内存管理是一个非常棘手的问题，我们在编写一个程序的时候几乎不可避免的要遇到内存的分配逻辑，这时候随之而来的有这样一些问题：是否有足够的内存可供分配? 分配失败了怎么办? 如何管理自身的内存使用情况? 等等一系列问题。在一个高可用的软件中，如果我们仅仅单纯的向操作系统去申请内存，当出现内存不足时就退出软件，是明显不合理的。正确的思路应该是在内存不足的时，考虑如何管理并优化自身已经使用的内存，这样才能使得软件变得更加可用。本次项目我们将实现一个内存池，并使用一个栈结构来测试我们的内存池提供的分配性能。最终，我们要实现的内存池在栈结构中的性能，要远高于使用 std::allocator 和 std::vector   * C++ 中的内存分配器 std::allocator * 内存池技术 * 手动实现模板链式栈 * 链式栈和列表栈的性能比较   **内存池简介**  内存池是池化技术中的一种形式。通常我们在编写程序的时候回使用 new delete 这些关键字来向操作系统申请内存，而这样造成的后果就是每次申请内存和释放内存的时候，都需要和操作系统的系统调用打交道，从堆中分配所需的内存。如果这样的操作太过频繁，就会找成大量的内存碎片进而降低内存的分配性能，甚至出现内存分配失败的情况。  而内存池就是为了解决这个问题而产生的一种技术。从内存分配的概念上看，内存申请无非就是向内存分配方索要一个指针，当向操作系统申请内存时，操作系统需要进行复杂的内存管理调度之后，才能正确的分配出一个相应的指针。而这个分配的过程中，我们还面临着分配失败的风险。  所以，每一次进行内存分配，就会消耗一次分配内存的时间，设这个时间为 T，那么进行 n 次分配总共消耗的时间就是 nT；如果我们一开始就确定好我们可能需要多少内存，那么在最初的时候就分配好这样的一块内存区域，当我们需要内存的时候，直接从这块已经分配好的内存中使用即可，那么总共需要的分配时间仅仅只有 T。当 n 越大时，节约的时间就越多。  **二、主函数设计**  我们要设计实现一个高性能的内存池，那么自然避免不了需要对比已有的内存，而比较内存池对内存的分配性能，就需要实现一个需要对内存进行动态分配的结构（比如：链表栈），为此，可以写出如下的代码：  #include <iostream> // std::cout, std::endl  #include <cassert> // assert()  #include <ctime> // clock()  #include <vector> // std::vector  #include "MemoryPool.hpp" // MemoryPool<T>  #include "StackAlloc.hpp" // StackAlloc<T, Alloc>  // 插入元素个数  #define ELEMS 10000000  // 重复次数  #define REPS 100  int main()  {  clock\_t start;  // 使用 STL 默认分配器  StackAlloc<int, std::allocator<int> > stackDefault;  start = clock();  for (int j = 0; j < REPS; j++) {  assert(stackDefault.empty());  for (int i = 0; i < ELEMS; i++)  stackDefault.push(i);  for (int i = 0; i < ELEMS; i++)  stackDefault.pop();  }  std::cout << "Default Allocator Time: ";  std::cout << (((double)clock() - start) / CLOCKS\_PER\_SEC) << "\n\n";  // 使用内存池  StackAlloc<int, MemoryPool<int> > stackPool;  start = clock();  for (int j = 0; j < REPS; j++) {  assert(stackPool.empty());  for (int i = 0; i < ELEMS; i++)  stackPool.push(i);  for (int i = 0; i < ELEMS; i++)  stackPool.pop();  }  std::cout << "MemoryPool Allocator Time: ";  std::cout << (((double)clock() - start) / CLOCKS\_PER\_SEC) << "\n\n";  return 0;  }  在上面的两段代码中，StackAlloc 是一个链表栈，接受两个模板参数，第一个参数是栈中的元素类型，第二个参数就是栈使用的内存分配器。  因此，这个内存分配器的模板参数就是整个比较过程中唯一的变量，使用默认分配器的模板参数为 std::allocator，而使用内存池的模板参数为 MemoryPool。  std::allocator 是 C++标准库中提供的默认分配器，他的特点就在于我们在 使用 new 来申请内存构造新对象的时候，势必要调用类对象的默认构造函数，而使用 std::allocator 则可以将内存分配和对象的构造这两部分逻辑给分离开来，使得分配的内存是原始、未构造的。  下面我们来实现这个链表栈。  **三、模板链表栈**  栈的结构非常的简单，没有什么复杂的逻辑操作，其成员函数只需要考虑两个基本的操作：入栈、出栈。为了操作上的方便，我们可能还需要这样一些方法：判断栈是否空、清空栈、获得栈顶元素。  #include <memory>  template <typename T>  struct StackNode\_  {  T data;  StackNode\_\* prev;  };  // T 为存储的对象类型, Alloc 为使用的分配器, 并默认使用 std::allocator 作为对象的分配器  template <typename T, typename Alloc = std::allocator<T> >  class StackAlloc  {  public:  // 使用 typedef 简化类型名  typedef StackNode\_<T> Node;  typedef typename Alloc::template rebind<Node>::other allocator;  // 默认构造  StackAlloc() { head\_ = 0; }  // 默认析构  ~StackAlloc() { clear(); }  // 当栈中元素为空时返回 true  bool empty() {return (head\_ == 0);}  // 释放栈中元素的所有内存  void clear();  // 压栈  void push(T element);  // 出栈  T pop();  // 返回栈顶元素  T top() { return (head\_->data); }  private:  //  allocator allocator\_;  // 栈顶  Node\* head\_;  };  简单的逻辑诸如构造、析构、判断栈是否空、返回栈顶元素的逻辑都非常简单，直接在上面的定义中实现了，下面我们来实现 clear(), push() 和 pop() 这三个重要的逻辑：  // 释放栈中元素的所有内存  void clear() {  Node\* curr = head\_;  // 依次出栈  while (curr != 0)  {  Node\* tmp = curr->prev;  // 先析构, 再回收内存  allocator\_.destroy(curr);  allocator\_.deallocate(curr, 1);  curr = tmp;  }  head\_ = 0;  }  // 入栈  void push(T element) {  // 为一个节点分配内存  Node\* newNode = allocator\_.allocate(1);  // 调用节点的构造函数  allocator\_.construct(newNode, Node());  // 入栈操作  newNode->data = element;  newNode->prev = head\_;  head\_ = newNode;  }  // 出栈  T pop() {  // 出栈操作 返回出栈元素  T result = head\_->data;  Node\* tmp = head\_->prev;  allocator\_.destroy(head\_);  allocator\_.deallocate(head\_, 1);  head\_ = tmp;  return result;  }  **总结**  本节我们实现了一个用于测试性能比较的模板链表栈，目前的代码如下。在下一节中，我们开始详细实现我们的高性能内存池。  // StackAlloc.hpp  #ifndef STACK\_ALLOC\_H  #define STACK\_ALLOC\_H  #include <memory>  template <typename T>  struct StackNode\_  {  T data;  StackNode\_\* prev;  };  // T 为存储的对象类型, Alloc 为使用的分配器,  // 并默认使用 std::allocator 作为对象的分配器  template <class T, class Alloc = std::allocator<T> >  class StackAlloc  {  public:  // 使用 typedef 简化类型名  typedef StackNode\_<T> Node;  typedef typename Alloc::template rebind<Node>::other allocator;  // 默认构造  StackAlloc() { head\_ = 0; }  // 默认析构  ~StackAlloc() { clear(); }  // 当栈中元素为空时返回 true  bool empty() {return (head\_ == 0);}  // 释放栈中元素的所有内存  void clear() {  Node\* curr = head\_;  while (curr != 0)  {  Node\* tmp = curr->prev;  allocator\_.destroy(curr);  allocator\_.deallocate(curr, 1);  curr = tmp;  }  head\_ = 0;  }  // 入栈  void push(T element) {  // 为一个节点分配内存  Node\* newNode = allocator\_.allocate(1);  // 调用节点的构造函数  allocator\_.construct(newNode, Node());  // 入栈操作  newNode->data = element;  newNode->prev = head\_;  head\_ = newNode;  }  // 出栈  T pop() {  // 出栈操作 返回出栈结果  T result = head\_->data;  Node\* tmp = head\_->prev;  allocator\_.destroy(head\_);  allocator\_.deallocate(head\_, 1);  head\_ = tmp;  return result;  }  // 返回栈顶元素  T top() { return (head\_->data); }  private:  allocator allocator\_;  Node\* head\_;  };  #endif // STACK\_ALLOC\_H  // main.cpp  #include <iostream>  #include <cassert>  #include <ctime>  #include <vector>  // #include "MemoryPool.hpp"  #include "StackAlloc.hpp"  // 根据电脑性能调整这些值  // 插入元素个数  #define ELEMS 25000000  // 重复次数  #define REPS 50  int main()  {  clock\_t start;  // 使用默认分配器  StackAlloc<int, std::allocator<int> > stackDefault;  start = clock();  for (int j = 0; j < REPS; j++) {  assert(stackDefault.empty());  for (int i = 0; i < ELEMS; i++)  stackDefault.push(i);  for (int i = 0; i < ELEMS; i++)  stackDefault.pop();  }  std::cout << "Default Allocator Time: ";  std::cout << (((double)clock() - start) / CLOCKS\_PER\_SEC) << "\n\n";  // 使用内存池  // StackAlloc<int, MemoryPool<int> > stackPool;  // start = clock();  // for (int j = 0; j < REPS; j++) {  // assert(stackPool.empty());  // for (int i = 0; i < ELEMS; i++)  // stackPool.push(i);  // for (int i = 0; i < ELEMS; i++)  // stackPool.pop();  // }  // std::cout << "MemoryPool Allocator Time: ";  // std::cout << (((double)clock() - start) / CLOCKS\_PER\_SEC) << "\n\n";  return 0;  }  **二、设计内存池**  在节中，我们在模板链表栈中使用了默认构造器来管理栈操作中的元素内存，一共涉及到了 rebind::other, allocate(), dealocate(), construct(), destroy()这些关键性的接口。所以为了让代码直接可用，我们同样应该在内存池中设计同样的接口：  #ifndef MEMORY\_POOL\_HPP  #define MEMORY\_POOL\_HPP  #include <climits>  #include <cstddef>  template <typename T, size\_t BlockSize = 4096>  class MemoryPool  {  public:  // 使用 typedef 简化类型书写  typedef T\* pointer;  // 定义 rebind<U>::other 接口  template <typename U> struct rebind {  typedef MemoryPool<U> other;  };  // 默认构造, 初始化所有的槽指针  // C++11 使用了 noexcept 来显式的声明此函数不会抛出异常  MemoryPool() noexcept {  currentBlock\_ = nullptr;  currentSlot\_ = nullptr;  lastSlot\_ = nullptr;  freeSlots\_ = nullptr;  }  // 销毁一个现有的内存池  ~MemoryPool() noexcept;  // 同一时间只能分配一个对象, n 和 hint 会被忽略  pointer allocate(size\_t n = 1, const T\* hint = 0);  // 销毁指针 p 指向的内存区块  void deallocate(pointer p, size\_t n = 1);  // 调用构造函数  template <typename U, typename... Args>  void construct(U\* p, Args&&... args);  // 销毁内存池中的对象, 即调用对象的析构函数  template <typename U>  void destroy(U\* p) {  p->~U();  }  private:  // 用于存储内存池中的对象槽,  // 要么被实例化为一个存放对象的槽,  // 要么被实例化为一个指向存放对象槽的槽指针  union Slot\_ {  T element;  Slot\_\* next;  };  // 数据指针  typedef char\* data\_pointer\_;  // 对象槽  typedef Slot\_ slot\_type\_;  // 对象槽指针  typedef Slot\_\* slot\_pointer\_;  // 指向当前内存区块  slot\_pointer\_ currentBlock\_;  // 指向当前内存区块的一个对象槽  slot\_pointer\_ currentSlot\_;  // 指向当前内存区块的最后一个对象槽  slot\_pointer\_ lastSlot\_;  // 指向当前内存区块中的空闲对象槽  slot\_pointer\_ freeSlots\_;  // 检查定义的内存池大小是否过小  static\_assert(BlockSize >= 2 \* sizeof(slot\_type\_), "BlockSize too small.");  };  #endif // MEMORY\_POOL\_HPP  在上面的类设计中可以看到，在这个内存池中，其实是使用链表来管理整个内存池的内存区块的。内存池首先会定义固定大小的基本内存区块(Block)，然后在其中定义了一个可以实例化为存放对象内存槽的对象槽（Slot\_）和对象槽指针的一个联合。然后在区块中，定义了四个关键性质的指针，它们的作用分别是：   1. currentBlock\_: 指向当前内存区块的指针 2. currentSlot\_: 指向当前内存区块中的对象槽 3. lastSlot\_: 指向当前内存区块中的最后一个对象槽 4. freeSlots\_: 指向当前内存区块中所有空闲的对象槽   梳理好整个内存池的设计结构之后，我们就可以开始实现关键性的逻辑了。  **三、实现**  **MemoryPool::construct() 实现**  MemoryPool::construct() 的逻辑是最简单的，我们需要实现的，仅仅只是调用信件对象的构造函数即可，因此：  // 调用构造函数, 使用 std::forward 转发变参模板  template <typename U, typename... Args>  void construct(U\* p, Args&&... args) {  new (p) U (std::forward<Args>(args)...);  }  **MemoryPool::deallocate() 实现**  MemoryPool::deallocate() 是在对象槽中的对象被析构后才会被调用的，主要目的是销毁内存槽。其逻辑也不复杂：  // 销毁指针 p 指向的内存区块  void deallocate(pointer p, size\_t n = 1) {  if (p != nullptr) {  // reinterpret\_cast 是强制类型转换符  // 要访问 next 必须强制将 p 转成 slot\_pointer\_  reinterpret\_cast<slot\_pointer\_>(p)->next = freeSlots\_;  freeSlots\_ = reinterpret\_cast<slot\_pointer\_>(p);  }  }  **MemoryPool::~MemoryPool() 实现**  析构函数负责销毁整个内存池，因此我们需要逐个删除掉最初向操作系统申请的内存块：  // 销毁一个现有的内存池  ~MemoryPool() noexcept {  // 循环销毁内存池中分配的内存区块  slot\_pointer\_ curr = currentBlock\_;  while (curr != nullptr) {  slot\_pointer\_ prev = curr->next;  operator delete(reinterpret\_cast<void\*>(curr));  curr = prev;  }  }  **MemoryPool::allocate() 实现**  MemoryPool::allocate() 毫无疑问是整个内存池的关键所在，但实际上理清了整个内存池的设计之后，其实现并不复杂。具体实现如下：  // 同一时间只能分配一个对象, n 和 hint 会被忽略  pointer allocate(size\_t n = 1, const T\* hint = 0) {  // 如果有空闲的对象槽，那么直接将空闲区域交付出去  if (freeSlots\_ != nullptr) {  pointer result = reinterpret\_cast<pointer>(freeSlots\_);  freeSlots\_ = freeSlots\_->next;  return result;  } else {  // 如果对象槽不够用了，则分配一个新的内存区块  if (currentSlot\_ >= lastSlot\_) {  // 分配一个新的内存区块，并指向前一个内存区块  data\_pointer\_ newBlock = reinterpret\_cast<data\_pointer\_>(operator new(BlockSize));  reinterpret\_cast<slot\_pointer\_>(newBlock)->next = currentBlock\_;  currentBlock\_ = reinterpret\_cast<slot\_pointer\_>(newBlock);  // 填补整个区块来满足元素内存区域的对齐要求  data\_pointer\_ body = newBlock + sizeof(slot\_pointer\_);  uintptr\_t result = reinterpret\_cast<uintptr\_t>(body);  size\_t bodyPadding = (alignof(slot\_type\_) - result) % alignof(slot\_type\_);  currentSlot\_ = reinterpret\_cast<slot\_pointer\_>(body + bodyPadding);  lastSlot\_ = reinterpret\_cast<slot\_pointer\_>(newBlock + BlockSize - sizeof(slot\_type\_) + 1);  }  return reinterpret\_cast<pointer>(currentSlot\_++);  }  }  **四、与 std::vector 的性能对比**  我们知道，对于栈来说，链栈其实并不是最好的实现方式，因为这种结构的栈不可避免的会涉及到指针相关的操作，同时，还会消耗一定量的空间来存放节点之间的指针。事实上，我们可以使用 std::vector 中的 push\_back() 和 pop\_back() 这两个操作来模拟一个栈，我们不妨来对比一下这个 std::vector 与我们所实现的内存池在性能上谁高谁低，我们在 主函数中加入如下代码：  // 比较内存池和 std::vector 之间的性能  std::vector<int> stackVector;  start = clock();  for (int j = 0; j < REPS; j++) {  assert(stackVector.empty());  for (int i = 0; i < ELEMS; i++)  stackVector.push\_back(i);  for (int i = 0; i < ELEMS; i++)  stackVector.pop\_back();  }  std::cout << "Vector Time: ";  std::cout << (((double)clock() - start) / CLOCKS\_PER\_SEC) << "\n\n"; |

# 6.C++ 实现高性能 RTTI 库

#### 参考1：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C++ RTTI库设计 RTTI 是 Runtime type information 的缩写, 是指在运行时保存对象的类型信息, 一般是用来描述C++的一种特性. C++本身提供了 dynamic\_cast 和 typeid 运算符来实现运行时类型识别, 但是 dynamic\_cast的 效率往往较低, 所以实现了一个较高性能的 RTTI 库.  dynamic\_cast的“运行时类型的转换匹配”，是通过维护一棵由type\_info类型对象作为节点的类型继承关系的树，遍历这棵继承树来确定一个待转换的对象的类型和目标类型之间是否存在 “is-a” 关系.  于是我们的思路就是参考dynamic\_cast的思路, 通过保存在继承链的类型信息, 然后线性查找来判断是否存在 “is-a” 关系.  根据 C++ 继承初始化顺序的特性, 我们可以通过让继承链上的每一个类都继承我们的协议类, 在协议类中保存每个类型的信息, 那么用什么来表示一类类型呢? 模版类 + 静态成员, 模版类实例化后每个特化的类都会持有同一个针对该类型特化的类的静态成员变量, 我们可以用该静态成员变量的地址来表示该类型.  接着还有一个问题, 我们的思路在多继承时会产生菱形继承问题, 这个比较简单, 用虚继承就可以, 接着会发现, 我们通过虚继承巧妙的将协议类中的成员实现了持久使用, 并且继承链上的每个类都保存了该继承类的类型信息.  为了简单起见, 可以使用使用数组来保存类型信息以及对应类型的对象地址. 因为虚继承的特性, 我们设计一个基类作为底层, 然后功能类虚继承基类, 通过继承的增长来实现下标的偏移将所需信息保存在数组中.  reinterpret\_cast 是为了将 id 地址的位模式重新解释, 具体使用方式见: [reinterpret\_cast 转换](https://zh.cppreference.com/w/cpp/language/reinterpret_cast)  实现如下:   |  |  | | --- | --- | |  | class rtti { protected:  std::array<std::intptr\_t, RTTI\_CHAIN\_MAX\_SIZE> inherChainID;  std::array<void\*, RTTI\_CHAIN\_MAX\_SIZE> inherChainPtr;   int inherChainCounter;  rtti(): inherChainCounter(-1) {  for (auto& id : inherChainID) id = -1;  for (auto& ptr : inherChainPtr) ptr = nullptr;  } public:   virtual ~rtti() {}  inline void\* getPtrKindOf(std::intptr\_t type) {  for (int i = 0; i <= inherChainCounter; ++i) {  if (inherChainID[i] == type) {  return inherChainPtr[i];  }  }  return nullptr;  }  inline bool isKindOf(std::intptr\_t type) {  return getPtrKindOf(type) ? true : false;  } };  template <typename T> class rttiport: public virtual rtti { private:  static std::intptr\_t id; public:  static std::intptr\_t type(); protected:  rttiport() {  ++inherChainCounter;  inherChainID[inherChainCounter] = type();  inherChainPtr[inherChainCounter] = static\_cast<T\*>(this);  }  virtual ~rttiport() {} }; template <typename T> std::intptr\_t rttiport<T>::id(0);  template <typename T> std::intptr\_t rttiport<T>::type() {  return reinterpret\_cast<std::intptr\_t>(&id); } |   为了使用方便, 还需要实现两个小函数, 来方便使用.   |  |  | | --- | --- | |  | template <typename T> static T\* cast(rtti\* to) {  if (to == nullptr) {  return nullptr;  }  return static\_cast<T\*> (to->getPtrKindOf(rttiport<T>::type())); }  template <typename T> static bool is(rtti\* obj) {  if (obj == nullptr) {  return false;  }  return obj->getPtrKindOf(rttiport<T>::type()) != nullptr; } |   cast 函数是将 to 指向的对象转换为 T 类型.  is 函数是为了更方便的进行类型识别, 判断 obj指向对象的类型和 T 是否相同. |

#### 参考2：

|  |
| --- |
| 运行时类型识别(run-time type identification, RTTI)的功能由两个[运算符](https://so.csdn.net/so/search?q=%E8%BF%90%E7%AE%97%E7%AC%A6&spm=1001.2101.3001.7020)实现：   * typeid运算符，用于返回表达式的类型。 * dynamic\_cast<>运算符，用于将基类的指针或引用安全的转换成派生类的指针或引用。   C++中没有JAVA中的[反射机制](https://so.csdn.net/so/search?q=%E5%8F%8D%E5%B0%84%E6%9C%BA%E5%88%B6&spm=1001.2101.3001.7020)，在运行时获得任何对象的类型信息，通常类型信息还非常丰富。C++只针对内置类型和多态的自定义类型实现了RTTI。C++引入了运行时类型信息，实际上将对象分为了两部分。   * 一种包含了类型信息，可以在运行时获取类型信息 * 一种没有包含类型信息   We cannot impose the burden of being able to identify an object at run-time on built-in types such as int and double without unacceptable costs in run-time. —[[C++之父Stroustrup]](https://docs.google.com/viewer?url=http%3A%2F%2Fwww.open-std.org%2Fjtc1%2Fsc22%2Fwg21%2Fdocs%2Fpapers%2F1992%2FWG21%25201992%2FX3J16_92-0068%2520WG21_N0145.pdf)  按照C++之父的说法他没有为所有对象引入运行时机制的原因是，有些对象根本不需要，并且负担不起运行时获取类型信息的开销。为了不使语言设计复杂化，并且在对多态类型引入RTTI机制已经足够的情况下，只为多态类型引入了RTTI。其中通常放在virtual table中的第一个槽(slot)内，GCC会将type\_info存放在虚表的第一个槽，而Visual Studio会将type\_info存放在虚表的最后，其实实现方式大同小异。如下图所示： 这里写图片描述  如果通过对象来获取类型信息，则可能的实现方式就是：  ((type\_info\*)(ptr->vptr[0]))->\_type\_descriptor;   * 1   获取到type\_info的object后，再调用runtime library函数与class type进行比对，这种方式显然会带来很大的运行时开销。  另外需要注意的是，当dynamic\_cast<>用于引用时需要注意：  const Derived& d = dynamic\_cast<const Derived&>(b);   * 1   因为不存在所谓的空引用，如果转换不成功的话程序会抛出名为std::bad\_cast的异常。  另一种常用的RTTI使用方式就是typeid()，typeid()会返回对象的类型信息，虽然这种方式很方便，但是我们也不能滥用这种方式来代替虚函数，这种方式扩展性不是很好，总之尽量不要使用typeid来代替虚函数的功能。  当然这只是一个示意图，真正编译器实现的时候要比这个复杂的多，有兴趣的可以看一看MSVC在RTTI上的实现方式[《Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI》](http://www.openrce.org/articles/full_view/23)  参考下面的代码： |
| To illustrate the following material, let's consider this simple example:  **class A**  **{**  **int a1;**  **public:**  **virtual int A\_virt1();**  **virtual int A\_virt2();**  **static void A\_static1();**  **void A\_simple1();**  **};**  **class B**  **{**  **int b1;**  **int b2;**  **public:**  **virtual int B\_virt1();**  **virtual int B\_virt2();**  **};**  **class C: public A, public B**  **{**  **int c1;**  **public:**  **virtual int A\_virt2();**  **virtual int B\_virt2();**  };  In most cases MSVC lays out classes in the following order:  **1. Pointer to virtual functions table (\_vtable\_ or \_vftable\_), added only when the class has virtual methods and no suitable table from a base class can be reused.**  **2. Base classes**  **3. Class members**  Virtual function tables consist of addresses of virtual methods in the order of their first appearance. Addresses of overloaded functions replace addresses of functions from base classes.  Thus, the layouts for our three classes will look like following:  **class A size(8):**  **+---**  **0 | {vfptr}**  **4 | a1**  **+---**  **A's vftable:**  **0 | &A::A\_virt1**  **4 | &A::A\_virt2**  **class B size(12):**  **+---**  **0 | {vfptr}**  **4 | b1**  **8 | b2**  **+---**  **B's vftable:**  **0 | &B::B\_virt1**  **4 | &B::B\_virt2**  **class C size(24):**  **+---**  **| +--- (base class A)**  **0 | | {vfptr}**  **4 | | a1**  **| +---**  **| +--- (base class B)**  **8 | | {vfptr}**  **12 | | b1**  **16 | | b2**  **| +---**  **20 | c1**  **+---**  **C's vftable for A:**  **0 | &A::A\_virt1**  **4 | &C::A\_virt2**  **C's vftable for B:**  **0 | &B::B\_virt1**  **4 | &C::B\_virt2**  The above diagram was produced by the VC8 compiler using an undocumented switch. To see the class layouts produced by the compiler, use: -d1reportSingleClassLayout to see the layout of a single class -d1reportAllClassLayout to see the layouts of all classes (including internal CRT classes) The layouts are dumped to stdout.  As you can see, C has two vftables, since it has inherited two classes which both already had virtual functions. Address of C::A\_virt2 replaces address of A::A\_virt2 in C's vftable for A, and C::B\_virt2 replaces B::B\_virt2 in the other table.  **Calling Conventions and Class Methods**  All class methods in MSVC by default use \_thiscall\_ convention. Class instance address (\_this\_ pointer) is passed as a hidden parameter in the ecx register. In the method body the compiler usually tucks it away immediately in some other register (e.g. esi or edi) and/or stack variable. All further adressing of the class members is done through that register and/or variable. However, when implementing COM classes, \_stdcall\_ convention is used. The following is an overview of the various class method types.  1) **Static Methods** Static methods do not need a class instance, so they work the same way as common functions. No \_this\_ pointer is passed to them. Thus it's not possible to reliably distinguish static methods from simple functions. Example:  A::A\_static1();  call A::A\_static1  2) **Simple Methods** Simple methods need a class instance, so \_this\_ pointer is passed to them as a hidden first parameter, usually using \_thiscall\_ convention, i.e. in \_ecx\_ register. When the base object is not situated at the beginning of the derived class, \_this\_ pointer needs to be adjusted to point to the actual beginning of the base subobject before calling the function. Example:  ;pC->A\_simple1(1);  ;esi = pC  push 1  mov ecx, esi  call A::A\_simple1  ;pC->B\_simple1(2,3);  ;esi = pC  lea edi, [esi+8] ;adjust this  push 3  push 2  mov ecx, edi  call B::B\_simple1  As you see, \_this\_ pointer is adjusted to point to the B subobject before calling B's method.  3) **Virtual Methods** To call a virtual method the compiler first needs to fetch the function address from the \_vftable\_ and then call the function at that address same way as a simple method (i.e. passing \_this\_ pointer as an implicit parameter). Example:  ;pC->A\_virt2()  ;esi = pC  mov eax, [esi] ;fetch virtual table pointer  mov ecx, esi  call [eax+4] ;call second virtual method    ;pC->B\_virt1()  ;edi = pC  lea edi, [esi+8] ;adjust this pointer  mov eax, [edi] ;fetch virtual table pointer  mov ecx, edi  call [eax] ;call first virtual method  4) **Constructors and Destructors** Constructors and destructors work similar to a simple method: they get an implicit \_this\_ pointer as the first parameter (e.g. ecx in case of \_thiscall\_ convention). Constructor returns the \_this\_ pointer in eax, even though formally it has no return value.  **RTTI Implementation**  RTTI (Run-Time Type Identification) is special compiler-generated information which is used to support C++ operators like dynamic\_cast<> and typeid(), and also for C++ exceptions. Due to its nature, RTTI is only required (and generated) for polymorphic classes, i.e. classes with virtual functions.  MSVC compiler puts a pointer to the structure called "Complete Object Locator" just before the vftable. The structure is called so because it allows compiler to find the location of the complete object from a specific vftable pointer (since a class can have several of them). COL looks like following:  struct RTTICompleteObjectLocator  {  DWORD signature; //always zero ?  DWORD offset; //offset of this vtable in the complete class  DWORD cdOffset; //constructor displacement offset  struct TypeDescriptor\* pTypeDescriptor; //TypeDescriptor of the complete class  struct RTTIClassHierarchyDescriptor\* pClassDescriptor; //describes inheritance hierarchy  };  Class Hierarchy Descriptor describes the inheritance hierarchy of the class. It is shared by all COLs for a class.  struct RTTIClassHierarchyDescriptor  {  DWORD signature; //always zero?  DWORD attributes; //bit 0 set = multiple inheritance, bit 1 set = virtual inheritance  DWORD numBaseClasses; //number of classes in pBaseClassArray  struct RTTIBaseClassArray\* pBaseClassArray;  };  Base Class Array describes all base classes together with information which allows compiler to cast the derived class to any of them during execution of the \_dynamic\_cast\_ operator. Each entry (Base Class Descriptor) has the following structure:  struct RTTIBaseClassDescriptor  {  struct TypeDescriptor\* pTypeDescriptor; //type descriptor of the class  DWORD numContainedBases; //number of nested classes following in the Base Class Array  struct PMD where; //pointer-to-member displacement info  DWORD attributes; //flags, usually 0  };  struct PMD  {  int mdisp; //member displacement  int pdisp; //vbtable displacement  int vdisp; //displacement inside vbtable  };  The PMD structure describes how a base class is placed inside the complete class. In the case of simple inheritance it is situated at a fixed offset from the start of object, and that value is the \_mdisp\_ field. If it's a virtual base, an additional offset needs to be fetched from the vbtable. Pseudo-code for adjusting \_this\_ pointer from derived class to a base class looks like the following:  //char\* pThis; struct PMD pmd;  pThis+=pmd.mdisp;  if (pmd.pdisp!=-1)  {  char \*vbtable = pThis+pmd.pdisp;  pThis += \*(int\*)(vbtable+pmd.vdisp);  }  For example, the RTTI hierarchy for our three classes looks like this:  [http://www.openrce.org/articles/img/igor2_rtti1_thumb.gif](http://www.openrce.org/articles/img/igor2_rtti1.gif) RTTI hierarchy for our example classes  **Extracting Information**  1) **RTTI** If present, RTTI is a valuable source of information for reversing. From RTTI it's possible to recover class names, inheritance hierarchy, and in some cases parts of the class layout. My RTTI scanner script shows most of that information. (see Appendix I)  2) **Static and Global Initializers** Global and static objects need to be initialized before the main program starts. MSVC implements that by generating initializer funclets and putting their addresses in a table, which is processed during CRT startup by the \_cinit function. The table usually resides in the beginning of .data section. A typical initializer looks like following:  \_init\_gA1:  mov ecx, offset \_gA1  call A::A()  push offset \_term\_gA1  call \_atexit  pop ecx  retn  \_term\_gA1:  mov ecx, offset \_gA1  call A::~A()  retn  Thus, from this table way we can find out:   * Global/static objects addresses * Their constructors * Their destructors   See also MSVC \_#pragma\_ directive \_init\_seg\_ [5].  3) **Unwind Funclets** If any automatic objects are created in a function, VC++ compiler automatically generates exception handling structures which ensure deletion of those objects in case an exception happens. See [Part I](http://www.openrce.org/articles/full_view/21) for a detailed description of C++ exception implementation. A typical unwind funclet destructs an object on the stack:  unwind\_1tobase: ; state 1 -> -1  lea ecx, [ebp+a1]  jmp A::~A()  By finding the opposite state change inside the function body or just the first access to the same stack variable, we can also find the constructor:    lea ecx, [ebp+a1]  call A::A()  mov [ebp+\_\_$EHRec$.state], 1  For the objects constructed using new() operator, the unwind funclet ensures deletion of allocated memory in case the constructor fails:  unwind\_0tobase: ; state 0 -> -1  mov eax, [ebp+pA1]  push eax  call operator delete(void \*)  pop ecx  retn  In the function body:  ;A\* pA1 = new A();  push  call operator new(uint)  add esp, 4  mov [ebp+pA1], eax  test eax, eax  mov [ebp+\_\_$EHRec$.state], 0; state 0: memory allocated but object is not yet constructed  jz short @@new\_failed  mov ecx, eax  call A::A()  mov esi, eax  jmp short @@constructed\_ok  @@new\_failed:  xor esi, esi  @@constructed\_ok:  mov [esp+14h+\_\_$EHRec$.state], -1  ;state -1: either object was constructed successfully or memory allocation failed  ;in both cases further memory management is done by the programmer  Another type of unwind funclets is used in constructors and destructors. It ensures destruction of the class members in case of exception. In this case the funclets use the \_this\_ pointer, which is kept in a stack variable:  unwind\_2to1:  mov ecx, [ebp+\_this] ; state 2 -> 1  add ecx, 4Ch  jmp B1::~B1  Here the funclet destructs a class member of type B1 at the offset 4Ch. Thus, from unwind funclets we can find out:   * Stack variables representing C++ objects or pointers to objects allocated with \_operator new\_. * Their destructors * Their constructors * in case of new'ed objects, their size   4) **Constructors / Destructors Recursion** This rule is simple: constructors call other constructors (of base classes and member variables) and destructors call other destructors. A typical constructor does the following:   * Call constructors of the base classes. * Call constructors of complex class members. * Initialize vfptr(s) if the class has virtual functions * Execute the constructor body written by the programmer.   Typical destructor works almost in the reverse order:   * Initialize vfptr if the class has virtual functions * Execute the destructor body written by the programmer. * Call destructors of complex class members * Call destructors of base classes   Another distinctive feature of destructors generated by MSVC is that their \_state\_ variable is usually initialized with the highest value and then gets decremented with each destructed subobject, which make their identification easier. Be aware that simple constructors/destructors are often inlined by MSVC. That's why you can often see the vftable pointer repeatedly reloaded with different pointers in the same function.  5) **Array Construction Destruction** The MSVC compiler uses a helper function to construct and destroy an array of objects. Consider the following code:  A\* pA = new A[n];    delete [] pA;  It is translated into the following pseudocode:  array = new char(sizeof(A)\*n+sizeof(int))  if (array)  {  \*(int\*)array=n; //store array size in the beginning  'eh vector constructor iterator'(array+sizeof(int),sizeof(A),count,&A::A,&A::~A);  }  pA = array;    'eh vector destructor iterator'(pA,sizeof(A),count,&A::~A);  If A has a vftable, a 'vector deleting destructor' is invoked instead when deleting the array:  ;pA->'vector deleting destructor'(3);  mov ecx, pA  push 3 ; flags: 0x2=deleting an array, 0x1=free the memory  call A::'vector deleting destructor'  If A's destructor is virtual, it's invoked virtually:  mov ecx, pA  push 3  mov eax, [ecx] ;fetch vtable pointer  call [eax] ;call deleting destructor  Consequently, from the vector constructor/destructor iterator calls we can determine:   * addresses of arrays of objects * their constructors * their destructors * class sizes   6) **Deleting Destructors** When class has a virtual destructor, compiler generates a helper function - deleting destructor. Its purpose is to make sure that a proper \_operator delete\_ gets called when destructing a class. Pseudo-code for a deleting destructor looks like following:  virtual void \* A::'scalar deleting destructor'(uint flags)  {  this->~A();  if (flags&1) A::operator delete(this);  };  The address of this function is placed into the vftable instead of the destructor's address. This way, if another class overrides the virtual destructor, \_operator delete\_ of that class will be called. Though in real code \_operator delete\_ gets overriden quite rarely, so usually you see a call to the default delete(). Sometimes compiler can also generate a vector deleting destructor. Its code looks like this:  virtual void \* A::'vector deleting destructor'(uint flags)  {  if (flags&2) //destructing a vector  {  array = ((int\*)this)-1; //array size is stored just before the this pointer  count = array[0];  'eh vector destructor iterator'(this,sizeof(A),count,A::~A);  if (flags&1) A::operator delete(array);  }  else {  this->~A();  if (flags&1) A::operator delete(this);  }  };  I skipped most of the details on implementation of classes with virtual bases since they complicate things quite a bit and are rather rare in the real world. Please refer to the article by Jan Gray[1]. It's very detailed, if a bit heavy on Hungarian notation. The article [2] describes an example of the virtual inheritance implementation in MSVC. See also some of the MS patents [3] for more details.  **Appendix I: ms\_rtti4.idc**  This is a script I wrote for parsing RTTI and vftables. You can download the scripts associated with both this article and the previous article from [Microsoft VC++ Reversing Helpers](http://www.openrce.org/downloads/details/196). The script features:   * Parses RTTI structures and renames vftables to use the corresponding class names. * For some simple cases, identifies and renames constructors and destructors. * Outputs a file with the list of all vftables with referencing functions and class hierarchy.    **Usage**: after the initial analysis finishes, load ms\_rtti4.idc. It will ask if you want to scan the exe for the vtables. Be aware that it can be a lengthy process. Even if you skip the scanning, you can still parse vtables manually. If you do choose to scan, the script will try to identify all vtables with RTII, rename them, and identify and rename constructors and destructors. In some cases it will fail, especially with virtual inheritance. After scanning, it will open the text file with results.  After the script is loaded, you can use the following hotkeys to parse some of the MSVC structures manually: * Alt-F8 - parse a vtable. The cursor should be at the beginning of the vtable. If there is RTTI, the script will use the class name from it. If there is none, you can enter the class name manually and the script will rename the vtable. If there is a virtual destructor which it can identify, the script will rename it too. * Alt-F7 - parse FuncInfo. FuncInfo is the structure present in functions which have objects allocated on the stack or use exception handling. Its address is passed to \_CxxFrameHandler in the function's exception handler: * mov eax, offset FuncInfo1 * jmp \_CxxFrameHandler   In most cases it is identified and parsed automatically by IDA, but my script provides more information. You can also use ms\_ehseh.idc from the first part of this article to parse all FuncInfos in the file.  Use the hotkey with cursor placed on the start of the FuncInfo structure.   * Alt-F9 - parse throw info. Throw info is a helper structure used by \_CxxThrowException to implement the \_throw\_ operator. Its address is the second argument to \_CxxThrowException: * lea ecx, [ebp+e] * call E::E() * push offset ThrowInfo\_E * lea eax, [ebp+e] * push eax * call \_CxxThrowException   Use the hotkey with the cursor placed on the start of the throw info structure. The script will parse the structure and add a repeatable comment with the name of the thrown class. It will also identify and rename the exception's destructor and copy constructor.  **Appendix II: Practical Recovery of a Class Structure**  Our subject will be MSN Messenger 7.5 (msnmsgr.exe version 7.5.324.0, size 7094272). It makes heavy use of C++ and has plenty of RTTI for our purposes. Let's consider two vftables, at .0040EFD8 and .0040EFE0. The complete RTTI structures hierarchy for them looks like following:  [http://www.openrce.org/articles/img/igor2_rtti2_thumb.gif](http://www.openrce.org/articles/img/igor2_rtti2.gif) RTTI hierarchy for MSN Messenger 7.5  So, these two vftables both belong to one class - CContentMenuItem. By checking its Base Class Descriptors we can see that:   * CContentMenuItem contains three bases that follow it in the array - i.e. CDownloader, CNativeEventSink and CNativeEventSource. * CDownloader contains one base - CNativeEventSink. * Hence, CContentMenuItem inherits directly from CDownloader and CNativeEventSource, and CDownloader in turn inherits from CNativeEventSink. * CDownloader is situated in the beginning of the complete object, and CNativeEventSource is at the offset 0x24.   http://www.openrce.org/articles/img/igor2_rtti3.gif  So we can conclude that the first vftable lists methods of CNativeEventSource and the second one of either CDownloader or CNativeEventSink (if neither of them had virtual methods, CContentMenuItem would reuse the vftable of CNativeEventSource). Now let's check what refers to these tables. They both are referred by two functions, at .052B5E0 and .052B547. (That reinforces the fact that they both belong to one class.) Moreover, if we look at the beginning of the function at .052B547, we see the \_state\_ variable initialized with 6, which means that that function is the destructor. As a class can have only one destructor, we can conclude that .052B5E0 is its constructor. Let's looks closer at it:  CContentMenuItem::CContentMenuItem proc near  this = esi  push this  push edi  mov this, ecx  call sub\_4CA77A  lea edi, [this+24h]  mov ecx, edi  call sub\_4CBFDB  or dword ptr [this+48h], 0FFFFFFFFh  lea ecx, [this+4Ch]  mov dword ptr [this], offset const CContentMenuItem::'vftable'{for 'CContentMenuItem'}  mov dword ptr [edi], offset const CContentMenuItem::'vftable'{for 'CNativeEventSource'}  call sub\_4D8000  lea ecx, [this+50h]  call sub\_4D8000  lea ecx, [this+54h]  call sub\_4D8000  lea ecx, [this+58h]  call sub\_4D8000  lea ecx, [this+5Ch]  call sub\_4D8000  xor eax, eax  mov [this+64h], eax  mov [this+68h], eax  mov [this+6Ch], eax  pop edi  mov dword ptr [this+60h], offset const CEventSinkList::'vftable'  mov eax, this  pop this  retn  sub\_52B5E0 endp  The first thing compiler does after prolog is copying \_this\_ pointer from ecx to esi, so all further addressing is done based on esi. Before initializing vfptrs it calls two other functions; those must be constructors of the base classes - in our case CDownloader and CNativeEventSource. We can confirm that by going inside each of the functions - first one initializes its vfptr field with CDownloader::'vftable' and the second with CNativeEventSource::'vftable'. We can also investigate CDownloader's constructor further - it calls constructor of its base class, CNativeEventSink.  Also, the \_this\_ pointer passed to the second function is taken from edi, which points to this+24h. According to our class structure diagram it's the location of the CNativeEventSource subobject. This is another confirmation that the second function being called is the constructor of CNativeEventSource.  After calling base constructors, the vfptrs of the base objects are overwritten with CContentMenuItem's implementations - which means that CContentMenuItem overrides some of the virtual methods of the base classes (or adds its own). (If needed, we can compare the tables and check which pointers have been changed or added - those will be new implementations by CContentMenuItem.)  Next we see several function calls to .04D8000 with \_ecx\_ set to this+4Ch to this+5Ch - apparently some member variables are initialized. How can we know whether that function is a compiler-generated constructor call or an initializer function written by the programmer? There are several hints that it's a constructor.   * The function uses \_thiscall\_ convention and it is the first time these fields are accessed. * The fields are initialized in the order of increasing addresses.   To be sure we can also check the unwind funclets in the destructor - there we can see the compiler-generated destructor calls for these member variables.  This new class doesn't have virtual methods and thus no RTTI, so we don't know its real name. Let's name it RefCountedPtr. As we have already determined, 4D8000 is its constructor. The destructor we can find out from the CContentMenuItem destructor's unwind funclets - it's at 63CCB4.  Going back to the CContentMenuItem constructor, we see three fields initialized with 0 and one with a vftable pointer. This looks like an inlined constructor for a member variable (not a base class, since a base class would be present in the inheritance tree). From the used vftable's RTTI we can see that it's an instance of CEventSinkList template.  Now we can write a possible declaration for our class.  class CContentMenuItem: public CDownloader, public CNativeEventSource  {  /\* 00 CDownloader \*/  /\* 24 CNativeEventSource \*/  /\* 48 \*/ DWORD m\_unknown48;  /\* 4C \*/ RefCountedPtr m\_ptr4C;  /\* 50 \*/ RefCountedPtr m\_ptr50;  /\* 54 \*/ RefCountedPtr m\_ptr54;  /\* 58 \*/ RefCountedPtr m\_ptr58;  /\* 5C \*/ RefCountedPtr m\_ptr5C;  /\* 60 \*/ CEventSinkList m\_EventSinkList;  /\* size = 70? \*/  };  We can't know for sure that the field at offset 48 is not a part of CNativeEventSource; but since it wasn't accessed in CNativeEventSource constructor, it is most probably a part of CContentMenuItem. The constructor listing with renamed methods and class structure applied:  public: \_\_thiscall CContentMenuItem::CContentMenuItem(void) proc near  push this  push edi  mov this, ecx  call CDownloader::CDownloader(void)  lea edi, [this+CContentMenuItem.\_CNativeEventSource]  mov ecx, edi  call CNativeEventSource::CNativeEventSource(void)  or [this+CContentMenuItem.m\_unknown48], -1  lea ecx, [this+CContentMenuItem.m\_ptr4C]  mov [this+CContentMenuItem.\_CDownloader.\_vfptr], offset const CContentMenuItem::'vftable'{for 'CContentMenuItem'}  mov [edi+CNativeEventSource.\_vfptr], offset const CContentMenuItem::'vftable'{for 'CNativeEventSource'}  call RefCountedPtr::RefCountedPtr(void)  lea ecx, [this+CContentMenuItem.m\_ptr50]  call RefCountedPtr::RefCountedPtr(void)  lea ecx, [this+CContentMenuItem.m\_ptr54]  call RefCountedPtr::RefCountedPtr(void)  lea ecx, [this+CContentMenuItem.m\_ptr58]  call RefCountedPtr::RefCountedPtr(void)  lea ecx, [this+CContentMenuItem.m\_ptr5C]  call RefCountedPtr::RefCountedPtr(void)  xor eax, eax  mov [this+CContentMenuItem.m\_EventSinkList.field\_4], eax  mov [this+CContentMenuItem.m\_EventSinkList.field\_8], eax  mov [this+CContentMenuItem.m\_EventSinkList.field\_C], eax  pop edi  mov [this+CContentMenuItem.m\_EventSinkList.\_vfptr], offset const CEventSinkList::'vftable'  mov eax, this  pop this  retn  public: \_\_thiscall CContentMenuItem::CContentMenuItem(void) endp |

#### 参考3：

|  |
| --- |
| **C++对象模型之RTTI的实现原理**  RTTI是Runtime Type Identification的缩写，意思是运行时类型识别。C++引入这个机制是为了让程序在运行时能根据基类的指针或引用来获得该指针或引用所指的对象的实际类型。但是现在RTTI的类型识别已经不限于此了，它还能通过typeid操作符识别出所有的基本类型（int，指针等）的变量对应的类型。  C++通过以下的两个操作提供RTTI：  （1）typeid运算符，该运算符返回其表达式或类型名的实际类型。  （2）dynamic\_cast运算符，该运算符将基类的指针或引用安全地转换为派生类类型的指针或引用。  下面分别详细地说明这两个操作的实现方式。  注所有的测试代码的测试环境均为：32位Ubuntu 14.04 g++ 4.8.2，若在不同的环境中进行测试，结果可能有不同。  1、typeid运算符  typeid运算符，后接一个类型名或一个表达式，该运算符返回一个类型为std::tpeinf的对象的const引用。type\_info是std中的一个类，它用于记录与类型相关的信息。类type\_info的定义大概如下：  class type\_info {  public:  virtual ~type\_info();  bool operator==(const type\_info&)const;  bool operator!=(const type\_info&)const;  bool before(const type\_info&)const;  const char\* name()const;  private:  type\_info(const type\_info&);  type\_info& operator=(const type\_info&);   // data members };  至于data members部分，不同的编译器会有所不同，但是都必须提供最小量的信息是class的真实名称和在type\_info对象之间的某些排序算法（通过before()成员函数提供），以及某些形式的描述器，用来表示显式的类的类型和该类的任何子类型。  从上面的定义也可以看到，type\_info提供了两个对象的相等比较操作，但是用户并不能自己定义一个type\_info的对象，而只能通过typeid运算符返回一个对象的const引用来使用type\_info的对象。因为其只声明了一个构造函数（复制构造函数）且为private，所以编译器不会合成任何的构造函数，而且赋值操作运行符也为private。这两个操作就完全禁止了用户对type\_info对象的定义和复制操作，用户只能通过指向type\_info的对象的指针或引用来使用该类。  下面说说，typeid对静态类型的表达式和动态类型的表达式的处理和实现。  1）typeid识别静态类型  当typeid中的操作数是如下情况之一时，typeid运算符指出操作数的静态类型，即编译时的类型。  （1）类型名  （2）一个基本类型的变量  （3）一个具体的对象  （4）一个指向不含有virtual函数的类对象的指针的解引用  （5）一个指向不含有virtual函数的类对象的引用  静态类型在程序的运行过程中并不会改变，所以并不需要在程序运行时计算类型，在编译时就能根据操作数的静态类型，推导出其类型信息。例如如下的代码片断，typeid中的操作数均为静态类型：  class X { ...... // 具有virtual函数 };  class XX : public X { ...... // 具有virtual函数};  class Y { ...... // 没有virtual函数};   int main() {  int n = 0;  XX xx;  Y y;  Y \*py = &y;   // int和XX都是类型名  cout << typeid(int).name() << endl;  cout << typeid(XX).name() << endl;  // n为基本变量  cout << typeid(n).name() << endl;  // xx所属的类虽然存在virtual，但是xx为一个具体的对象  cout << typeid(xx).name() << endl;  // py为一个指针，属于基本类型  cout << typeid(py).name() << endl;  // py指向的Y的对象，但是类Y不存在virtual函数  cout << typeid(\*py).name() << endl;  return 0; }  2）typeid识别多态类型  当typeid中的操作数是如下情况之一时，typeid运算符需要在程序运行时计算类型，因为其其操作数的类型在编译时期是不能被确定的。  （1）一个指向不含有virtual函数的类对象的指针的解引用  （2）一个指向不含有virtual函数的类对象的引用  多态的类型是可以在运行过程中被改变的，例如，一个基类的指针，在程序运行的过程中，它可以指向一个基类对象，也可以指向该基类的派生类的对象，而typeid运算符需要在运行过程中识别出该基类指针所指向的对象的实际类型，这就需要typeid运算符在运行过程中计算其指向的对象的实际类型。例如对于以下的类定义：  class X {  public:  X()  {  mX = 101;  }  virtual void vfunc()  {  cout << "X::vfunc()" << endl;  }  private:  int mX; }; class XX : public X {  public:  XX():  X()  {  mXX = 1001;  }  virtual void vfunc()  {  cout << "XX::vfunc()" << endl;  }  private:  int mXX; };  使用如下的代码进行测试：  void printTypeInfo(const X \*px) {  cout << "typeid(px) -> " << typeid(px).name() << endl;  cout << "typeid(\*px) -> " << typeid(\*px).name() << endl; } int main() {  X x;  XX xx;  printTypeInfo(&x);  printTypeInfo(&xx);  return 0; }  其输出如下：  C++对象模型之RTTI的实现原理_运算符_02  从输出的结果可以看出，无论printTypeInfo函数中指针px指向的对象是基类X的对象，还是指向派生类XX的对象，typeid运行返回的px的类型信息都是相同的，因为px为一个静态类型，其类型名均为PX1X。但是typeid运算符却能正确地计算出了px指向的对象的实际类型。（注：由于C++为了保证每一个类在程序中都有一个独一无二的类名，所以会对类名通过一定的规则进行改写，所以在这里显示的类名跟我们定义的有一些不一样，如类XX的类名，被改写成了2XX。）  那么问题来了，typeid是如何计算这个类型信息的呢？下面将重点说明这个问题。  多态类型是通过在类中声明一个或多个virtual函数来区分的。因为在C++中，一个具备多态性质的类，正是内含直接声明或继承而来的virtual函数。多态类的对象的类型信息保存在虚函数表的索引的-1的项中，该项是一个type\_info对象的地址，该type\_info对象保存着该对象对应的类型信息，每个类都对应着一个type\_info对象。下面就对这一说法进行验证。  使用如以的代码，对上述的类X和类XX的对象的内存布局进行测试：  typedef void (\*FuncPtr)(); int main() {  XX xx;  FuncPtr func;  char \*p = (char\*)&xx;  // 获得虚函数表的地址  int \*\*vtbl = (int\*\*)\*(int\*\*)p;  // 输出虚函数表的地址，即vptr的值  cout << vtbl << endl;  // 获得type\_info对象的指针，并调用其name成员函数  cout << "\t[-1]: " << (vtbl[-1]) << " -> "  << ((type\_info\*)(vtbl[-1]))->name() << endl;  // 调用第一个virtual函数  cout << "\t[0]: " << vtbl[0] << " -> ";  func = (FuncPtr)vtbl[0];  func();  // 输出基类的成员变量的值  p += sizeof(int\*\*);  cout << \*(int\*)p << endl;  // 输出派生类的成员变量的值  p += sizeof(int);  cout << \*(int\*)p << endl;  return 0; }  测试代码，对类XX的对象的内存布局进行测试，其输出结果如下：  C++对象模型之RTTI的实现原理_c++_03  从运行结果可以看到，利用虚函数表的-1的项的地址转换成一个type\_info的指针类型，并调用name成员函数的输出为2XX，其输出与前面的测试代码中利用typeid的输出一致。从而可以知道，关于多态类型的计算是通过基类指针或引用指向的对象（子对象）的虚函数表获得的。  从运行的结果可以知道，类XX的对象的内存布局如下：  C++对象模型之RTTI的实现原理_运算符_04  对于以下的代码片断：  typeid(\*px).name()  可能被转换成如下的C++伪代码，用于计算实际对象的类型：  (\*(type\_info\*)px->vptr[-1]).name();  在多重继承和虚拟继承的情况下，一个类有n（n>1）个虚函数表，该类的对象也有n个vptr，分别指向这些虚函数表，但是一个类的所有的虚函数表的索引为-1的项的值（type\_info对象的地址）都是相等的，即它们都指向同一个type\_info对象，这样就实现了无论使用了哪一个基类的指针或引用指向其派生类的对象，都能通过相应的虚函数表获取到相同的type\_info对象，从而得到相同的类型信息。  3）typeid的识别错误的情况  从第2）节可以看到，typeid对于多态类型是通过虚函数表来计算的，若一个基类的指针指向了一个派生类，而该派生类并不存在virtual函数会出现什么情况呢？  例如，把第2）节中的X和XX类中的virtual函数全部去掉，改成以下的代码：  class X {  public:  X()  {  mX = 101;  }  private:  int mX; };  class XX : public X {  public:  XX():  X()  {  mXX = 1001;  }  private:  int mXX; };  测试代码不变，如下：  void printTypeInfo(const X \*px) {  cout << "typeid(px) -> " << typeid(px).name() << endl;  cout << "typeid(\*px) -> " << typeid(\*px).name() << endl; } int main() {  X x;  XX xx;   printTypeInfo(&x);  printTypeInfo(&xx); // 注释1   return 0; }  其输出如下：  C++对象模型之RTTI的实现原理_向下转型_05  从输出的结果可以看到，对于注释1的函数调用，虽然函数中基类（X）的指针px指向一个派生类对象（XX类的对象xx），但是typeid却并不没有像第2）节那样能正确地通过指针px计算出其所指对象的实际类型。  其原因在于类XX和类X都没有一个virtual函数，所以类XX和类X并不表现出多态类的性质。所以对类的指针的解引用符合第1）节中所说的静态类型，所以其类型信息是在编译时就已经确定的，并不需要在程序运行的过程中运行计算，所以其输出的类型均为1X而没有输出1XX。更进一步说，是因为类X和类XX都不存在virtual函数，所以类X和XX都不存在虚函数表，所以也就没有空间存储跟类X和XX类型有关的type\_info对象的地址。  然而在C++中即使一个类不具有多态的性质，仍然允许把一个派生类的指针赋值给一个基类的指针，所以这个错误比较隐晦。  2、dynamic\_cast运算符  把一个基类类型的指针或引用转换至继承架构的末端某一个派生类类型的指针或引用被称为向下转型（downcast）。dynamic\_cast运算符的作用是安全而有效地进行向下转型。  把一个派生类的指针或引用转换成其基类的指针或引用总是安全的，因为通过分析对象的内存布局可以知道，派生类的对象中必然存在基类的子对象，所以通过基类的指针或引用对派生类对象进行的所有基类的操作都是合法和安全的。而向下转型有潜在的危险性，因为基类的指针可以指向基类对象或其任何派生类的对象，而该对象并不一定是向下转型的类型的对象。所以向下转型遏制了类型系统的作用，转换后对指针或引用的使用可能会引发错误的解释或腐蚀程序内存等错误。  例如对于以下的类定义：  登录后复制  class X {  public:  X()  {  mX = 101;  }  virtual ~X()  {  }  private:  int mX; };  class XX : public X {  public:  XX():  X()  {  mXX = 1001;  }  virtual ~XX()  {  }  private:  int mXX; };  class YX : public X {  public:  YX()  {  mYX = 1002;  }  virtual ~YX()  {  }  private:  int mYX; };  使用如下的测试代码，其中的类型转换均为向下转型：int main(){ X x; XX xx; YX yx; X \*px = &xx; cout << px << endl; XX \*pxx = dynamic\_cast<XX\*>(px); // 转换1 cout << pxx << endl; YX \*pyx = dynamic\_cast<YX\*>(px); // 转换2 cout << pyx << endl; pyx = (YX\*)px; // 转换3 cout << pyx << endl; pyx = static\_cast<YX\*>(px); // 转换4 cout << pyx << endl; return 0;}  其运行结果如下：  C++对象模型之RTTI的实现原理_派生类_06  运行结果分析  px是一个基类（X）的指针，但是它指向了派生类XX的一个对象。在转换1中，转换成功，因为px指向的对象确实为XX的对象。在转换2中，转换失败，因为px指向的对象并不是一个YX对象，此时dymanic\_cast返回NULL。转换3为C风格的类型转换而转换4使用的是C++中的静态类型转换，它们均能成功转换，但是这个对象实际上并不是一个YX的对象，所以在转换3和转换4中，若继续通过指针使用该对象必然会导致错误，所以这个转换是不安全的。  从上述的结果可以看出在向下转型中，只有dynamic\_case才能实现安全的向下转型。那么dynamic\_case是如何实现的呢？有了上面typeid和虚函数表的知识后，这个问题并不难解释了，以转换1为例。  1）计算指针或引用变量所指的对象的虚函数表的type\_info信息，如下：  \*(type\_info\*)px->vptr[-1]  2）静态推导向下转型的目标类型的type\_info信息，即获取类XX的type\_info信息  3）比较1）和2）中获取到的type\_info信息，若2）中的类型信息与1）中的类型信息相等或是其基类类型，则返回相应的对象或子对象的地址，否则返回NULL。  引用的情况与指针稍有不同，失败时并不是返回NULL，而是抛出一个bad\_cast异常，因为引用不能参考NULL。 |

# 7.C++ 实现智能指针

#### 参考1：

|  |
| --- |
| 何为智能指针，即不需要用户再自动手动New和Delete掉，由内部类进行new和delete，这样可以防止用户忘记释放掉指针而造成的内存泄漏。  这里简单的实现unique\_ptr智能指针的方式。  头文件  **// 自制智能指针类头文件**  **template<typename T>**  **class SmartPointer{**  **public:**  **SmartPointer(T\* ptr\_);**  **~SmartPointer();**  **T\* operator->() const; //重载->符号**  **T& operator\*() const; //重载\*符号**  **SmartPointer<T>& operator=(SmartPointer<T>&ptr\_); //重写=符号::转移所有权**  **void Release(); //释放智能指针**  **T\* GetObjPtr(); //得到指针地址**  **T \* m\_ptr; //存放智能指针**  **private:**  **int m\_count;//指针计数:为0时可释放**  **};**   * 实现文件   **template<typename T>**  **SmartPointer<T>::SmartPointer(T\* ptr\_)**  **{**  **// 判断是否为空**  **if (ptr\_)**  **{**  **m\_ptr = ptr\_;**  **m\_count = 1;**  **}**  **else{**  **m\_ptr = nullptr;**  **cout << "指针为空，无法实现智能指针" << endl;**  **m\_count = 0;**  **}**  **}**  **template<typename T>**  **SmartPointer<T>::~SmartPointer()**  **{**  **if (m\_ptr&&(m\_count <= 1))**  **{**  **cout << "---------析构-------" << endl;**  **delete m\_ptr;**  **m\_ptr = NULL; //置为空**  **}**  **else**  **{**  **cout << "智能指针被转移:不进行析构" << endl;**  **}**  **}**  **template<typename T>**  **T\* SmartPointer<T>::operator->() const**  **{**  **if (m\_ptr)**  **{**  **return m\_ptr;**  **}**  **else**  **{**  **cout << "智能指针为空，出错了！" << endl;**  **return nullptr;**  **}**  **}**  **template<typename T>**  **T& SmartPointer<T>::operator\*()const**  **{**  **if (m\_ptr)**  **{**  **return \*m\_ptr;**  **}**  **else{**  **cout << "智能指针为空，出错了！" << endl;**  **return NULL;**  **}**  **}**  **template<typename T>**  **SmartPointer<T>& SmartPointer<T>::operator=(SmartPointer<T>& ob\_)**  **{**  **if (this != &ob\_)**  **{**  **delete m\_ptr;**  **m\_ptr = ob\_.m\_ptr;**  **ob\_.m\_ptr = NULL;**  **}**  **return \*this;**  **}**  **template<typename T>**  **void SmartPointer<T>::Release()**  **{**  **if(m\_ptr)**  **{**  **delete m\_ptr; //提前释放**  **m\_ptr = NULL; //需要置为空**  **m\_count = 0;**  **}**  **}**  **//得到指针对象**  **template<typename T>**  **T\* SmartPointer<T>::GetObjPtr()**  **{**  **if (m\_ptr)**  **{**  **m\_count ++; //计数增加**  **return m\_ptr;**  **}**  **return nullptr;**  **}**  \*使用方法如下：  **//测试智能指针方法**  **void TestSmartPointer()**  **{**  **SmartPointer<A> MM(new A());**  **SmartPointer<A> FF(new A());**  **}**  **int \_tmain(int argc, \_TCHAR\* argv[])**  **{**  **TestSmartPointer();**  **system("pause");**  **return 0;**  **}**  运行结果： 在这里插入图片描述 其中注意点： 1.类模板的声明与实现要在同一个文件中（原因是因为：c++标准中规定的模板类的在不用到它的时候不会被具现出来，编译后的链接时会出现问题。）解答源自：https://blog.csdn.net/qq\_37623612/article/details/79473627 2.假如调用GetObjPtr()需要手动释放掉，这种的原因在于，用于特殊情况，需要自己手动删掉指针，为这种情况下提供接口（一般不建议使用） |

#### 参考2：

|  |
| --- |
| **C++智能指针简单实现**  参考: [C++ 引用计数技术及智能指针的简单实现 - melonstreet - 博客园](https://www.cnblogs.com/QG-whz/p/4777312.html) c++ primer 5th 12.1  **文章目录**   * + [智能指针是什么](https://blog.csdn.net/qq_32523711/article/details/109163762#_6)   + [普通指针存在的问题](https://blog.csdn.net/qq_32523711/article/details/109163762#_9)   + [什么是引用计数?](https://blog.csdn.net/qq_32523711/article/details/109163762#_38)   + [智能指针实现](https://blog.csdn.net/qq_32523711/article/details/109163762#_44)     - [基础对象类](https://blog.csdn.net/qq_32523711/article/details/109163762#_47)     - [辅助类](https://blog.csdn.net/qq_32523711/article/details/109163762#_61)     - [为基础对象类实现智能指针类](https://blog.csdn.net/qq_32523711/article/details/109163762#_75)     - [智能指针类的使用与测试](https://blog.csdn.net/qq_32523711/article/details/109163762#_103)     - [智能指针类的改进一](https://blog.csdn.net/qq_32523711/article/details/109163762#_171)     - [智能指针改进二](https://blog.csdn.net/qq_32523711/article/details/109163762#_206)   [智能指针](https://so.csdn.net/so/search?q=%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88&spm=1001.2101.3001.7020" \t "_blank)**是什么**  简单来说，智能指针是一个管理内存的类，它对普通指针进行封装，使智能指针[类对象](https://so.csdn.net/so/search?q=%E7%B1%BB%E5%AF%B9%E8%B1%A1&spm=1001.2101.3001.7020)具有普通指针类型一样的操作。具体而言，拷贝对象时，副本和原对象都指向同一存储区域，如果通过一个副本改变其所指的值，则通过另一对象访问的值也会改变。所不同的是，智能指针能够对内存进行进行自动管理，避免出现悬垂指针等情况。  **普通指针存在的问题**  多个指针指向同一个基础对象时，如果某个指针delete了该基础对象，对这个指针来说它是明确了它所指的对象被释放掉了，所以它不会再对所指对象进行操作，但是对于剩下的其他指针来说呢？它们还傻傻地指向已经被删除的基础对象并随时准备对它进行操作。于是悬垂指针就形成了，程序崩溃也“指日可待”。我们通过代码+图来来探求悬垂指针的解决方法。  #include <iostream>  using namespace std;  int main(){  int \*ptr1 = new int(1);  int \*ptr2 = ptr1;  int \*ptr3 = ptr2;  cout << \*ptr1 << endl;//1  cout << \*ptr2 << endl;//1  cout << \*ptr3 << endl;//1  delete ptr1;  cout << \*ptr2 << endl;//0  return 0;  }  运行结果是输出ptr2时并不是期待的1，因为1已经被删除了。这个过程是这样的： 在这里插入图片描述 在这里插入图片描述 在这里插入图片描述 从图可以看出，错误的产生来自于ptr1的”无知“：它并不知道还有其他指针共享着它指向的对象。如果有个办法让ptr1知道，除了它自己外还有两个指针指向基础对象，而它不应该删除基础对象，那么悬垂指针的问题就得以解决了。如下图： 在这里插入图片描述 在这里插入图片描述 那么何时才可以删除基础对象呢？当然是只有一个指针指向基础对象的时候，这时通过该指针就可以大大方方地把基础对象删除了。  **什么是引用计数?**  如何来让指针知道还有其他指针的存在呢？这个时候我们该引入引用计数的概念了。引用计数是这样一个技巧，它**允许有多个相同值的对象共享这个值的实现**。引用计数的使用常有两个目的：   * 简化跟踪堆中（也即C++中new出来的）的对象的过程。一旦一个对象通过调用new被分配出来，记录谁拥有这个对象是很重要的，因为其所有者要负责对它进行delete。但是对象所有者可以有多个，且所有权能够被传递，这就使得内存跟踪变得困难。引用计数可以跟踪对象所有权，并能够自动销毁对象。可以说引用计数是个简单的垃圾回收体系。 * 节省内存，提高程序运行效率。如何很多对象有相同的值，为这多个相同的值存储多个副本是很浪费空间的，所以最好做法是让左右对象都共享同一个值的实现。**C++标准库中string类采取一种称为”写时复制“的技术，使得只有当字符串被修改的时候才创建各自的拷贝，否则可能（标准库允许使用但没强制要求）采用引用计数技术来管理共享对象的多个对象**。这不是本文的讨论范围。   **智能指针实现**  智能指针的实现策略有两种：辅助类与句柄类。这里介绍辅助类的实现方法。  **基础对象类**  首先，我们来定义一个基础对象类Point类，为了方便后面我们验证智能指针是否有效，我们为Point类创建如下接口：  class Point{  public:  Point(int xVal = 0, int yVal = 0) :x(xVal), y(yVal) { }  int getX() const { return x; }  int getY() const { return y; }  void setX(int xVal) { x = xVal; }  void setY(int yVal) { y = yVal; }  private:  int x, y;  };  **辅助类**  在创建智能指针类之前，我们先创建一个辅助类。这个类的所有成员**皆为私有类型，因为它不被普通用户所使用**。为了只为智能指针使用，还需要**把智能指针类声明为辅助类的友元**。这个辅助类含有两个数据成员：计数count与基础对象指针。也即**辅助类用以封装使用计数与基础对象指针**。  class U\_Ptr{  private:  friend class SmartPtr;  U\_Ptr(Point \*ptr) : p(ptr), count(1) { }//创建的时候count初始化为1  ~U\_Ptr() { delete p; }  Point \*p;  int count;  };  **为基础对象类实现智能指针类**  引用计数是实现智能指针的一种通用方法。**智能指针将一个计数器与类指向的对象相关联，引用计数跟踪共有多少个类对象共享同一指针**。它的具体做法如下：   * 当创建类的新对象时，初始化指针，并将引用计数设置为1 * 当对象作为另一个对象的副本时，拷贝构造函数拷贝副本指针，并增加与指针相应的引用计数（加1） * 使用赋值操作符对一个对象进行赋值时，处理复杂一点：先使左操作数的指针的引用计数减1（为何减1：因为指针已经指向别的地方），如果减1后引用计数为0，则释放指针所指对象内存。然后增加右操作数所指对象的引用计数（为何增加：因为此时做操作数指向对象即右操作数指向对象）。 * 析构函数：调用析构函数时，析构函数先使引用计数减1，如果减至0则delete对象。   做好前面的准备后，我们可以来为基础对象类Point书写一个智能指针类了。根据引用计数实现关键点，我们可以写出我们的智能指针类如下：  class SmartPtr{  public:  SmartPtr(Point \*ptr) : rp(new U\_Ptr(ptr)) {}//构造函数  SmartPtr(const SmartPtr &sp) : rp(sp.rp) { ++rp->count;}//复制构造函数  SmartPtr& operator=(const SmartPtr& rhs){//重载赋值操作符  ++rhs.rp->count;//首先将右操作数引用计数加1，  if(--rp->count == 0) delete rp;//然后将引用计数减1，可以应对自赋值  rp = rhs.rp;  return \*this;  }  ~SmartPtr(){//析构函数  if(--rp->count == 0) delete rp;//当引用计数减为0时，删除辅助类对象指针，从而删除基础对象  else cout << "还有 " << rp->count << " 个指针指向基础对象" << endl;  }  private:  U\_Ptr \*rp;//辅助类对象指针  };  **智能指针类的使用与测试**  #include <iostream>  using namespace std;  class Point{  public:  Point(int xVal = 0, int yVal = 0) :x(xVal), y(yVal) { }  int getX() const { return x; }  int getY() const { return y; }  void setX(int xVal) { x = xVal; }  void setY(int yVal) { y = yVal; }  private:  int x, y;  };  class U\_Ptr{  private:  friend class SmartPtr;  U\_Ptr(Point \*ptr) : p(ptr), count(1) { }  ~U\_Ptr() { delete p; }  Point \*p;  int count;  };  class SmartPtr{  public:  SmartPtr(Point \*ptr) : rp(new U\_Ptr(ptr)) {}//构造函数  SmartPtr(const SmartPtr &sp) : rp(sp.rp) { ++rp->count;}//复制构造函数  SmartPtr& operator=(const SmartPtr& rhs){//重载赋值操作符  ++rhs.rp->count;//首先将右操作数引用计数加1，  if(--rp->count == 0) delete rp;//然后将引用计数减1，可以应对自赋值  rp = rhs.rp;  return \*this;  }  ~SmartPtr(){//析构函数  if(--rp->count == 0) delete rp;//当引用计数减为0时，删除辅助类对象指针，从而删除基础对象  else cout << "还有 " << rp->count << " 个指针指向基础对象" << endl;  }  private:  U\_Ptr \*rp;//辅助类对象指针  };  int main(){  Point \*pa = new Point(10, 20);定义一个基础对象类指针  //定义三个智能指针类对象，对象都指向基础类对象pa  //使用花括号控制三个指针指针的生命期，观察计数的变化  {  SmartPtr sptr1(pa);//此时计数count=1  {  SmartPtr sptr2(sptr1);//调用拷贝构造函数，此时计数为count=2  {  SmartPtr sptr3 = sptr1;//调用拷贝赋值运算符，此时计数为conut=3  }//此时count=2  }//此时count=1  }//此时count=0；pa对象被delete掉  cout << pa->getX() << endl;//未定义的值  return 0;  }  输出： 在这里插入图片描述 在离开大括号后，共享基础对象的指针从3->2->1->0变换，最后计数为0时，pa对象被delete，此时使用getX（）已经获取不到原来的值。  **智能指针类的改进一**  虽然我们的SmartPtr类称为智能指针，但它目前并不能像真正的指针那样有->、\*等操作符，为了使它看起来更像一个指针，我们来为它重载这些操作符。代码如下所示：  class SmartPtr{  public:  SmartPtr(Point \*ptr) : rp(new U\_Ptr(ptr)) {}//构造函数  SmartPtr(const SmartPtr &sp) : rp(sp.rp) { ++rp->count;}//复制构造函数  SmartPtr& operator=(const SmartPtr& rhs){//重载赋值操作符  ++rhs.rp->count;//首先将右操作数引用计数加1，  if(--rp->count == 0) delete rp;//然后将引用计数减1，可以应对自赋值  rp = rhs.rp;  return \*this;  }  ~SmartPtr(){//析构函数  if(--rp->count == 0) delete rp;//当引用计数减为0时，删除辅助类对象指针，从而删除基础对象  else cout << "还有 " << rp->count << " 个指针指向基础对象" << endl;  }  Point& operator\*() const {//重载\*操作符  return \*(rp->p);  }  Point\* operator->() const {//重载->操作符  return rp->p;  }  private:  U\_Ptr \*rp;//辅助类对象指针  };  那么就可以实现：  SmartPtr sptr1(pa);//此时计数count=1  cout << sptr1->getX() << endl;  cout << (\*sptr1).getX() << endl;  **智能指针改进二**  目前这个智能指针智能用于管理Point类的基础对象，如果此时定义了个矩阵的基础对象类，那不是还得重新写一个属于矩阵类的智能指针类吗？但是矩阵类的智能指针类设计思想和Point类一样啊，就不能借用吗？答案当然是能，那就是使用模板技术。为了使我们的智能指针适用于更多的基础对象类，我们有必要把智能指针类通过模板来实现。这里贴上上面的智能指针类的模板版：  #include <iostream>  using namespace std;  //模板类作为友元时要先有声明  template <class T>  class SmartPtr;  template <class T>  class U\_Ptr{  private://该类成员访问权限全部为private，因为不想让用户直接使用该类  friend class SmartPtr<T>;//定义智能指针类为友元，因为智能指针类需要直接操纵辅助类  U\_Ptr(T \*ptr) : p(ptr), count(1) { }  ~U\_Ptr() { delete p; }  T \*p;//基础对象指针  int count;//引用计数  };  template <class T>  class SmartPtr{//智能指针类  public:  SmartPtr(T \*ptr) : rp(new U\_Ptr<T>(ptr)) {}//构造函数  SmartPtr(const SmartPtr &sp) : rp(sp.rp) { ++rp->count;}//复制构造函数  SmartPtr& operator=(const SmartPtr& rhs){//重载赋值操作符  ++rhs.rp->count;//首先将右操作数引用计数加1，  if(--rp->count == 0) delete rp;//然后将引用计数减1，可以应对自赋值  rp = rhs.rp;  return \*this;  }  T& operator\*() const {//重载\*操作符  return \*(rp->p);  }  T\* operator->() const {//重载->操作符  return rp->p;  }  ~SmartPtr(){//析构函数  if(--rp->count == 0) delete rp;//当引用计数减为0时，删除辅助类对象指针，从而删除基础对象  else cout << "还有 " << rp->count << " 个指针指向基础对象" << endl;  }  private:  U\_Ptr<T> \*rp;//辅助类对象指针  };  int main(){  int \*i = new int(2);  //定义三个智能指针类对象，对象都指向基础类对象pa  //使用花括号控制三个指针指针的生命期，观察计数的变化  {  SmartPtr<int> ptr1(i);//此时计数count=1  {  SmartPtr<int> ptr2(ptr1);//调用拷贝构造函数，此时计数为count=2  {  SmartPtr<int> ptr3 = ptr1;//调用拷贝赋值运算符，此时计数为conut=3  cout << \*ptr1 << endl;  \*ptr1 = 20;  cout << \*ptr2 << endl;  }//此时count=2  }//此时count=1  }//此时count=0  return 0;  }  输出：  在这里插入图片描述 |

#### 参考3：

|  |
| --- |
| **【C++】智能指针的原理和实现**  **不要用auto\_ptr**  回顾：  C++11 4种智能指针种，C++STL  auto\_ptr 不提倡用的原因是：  auto\_ptr<TYPE> A,B;  A = B;  第二行执行完毕后，B的地址为0；  这就是智能指针auto\_ptr是转移语义造成的，容易让人用错的地方。    **智能指针的原理和实现**    **智能指针的实现主要依赖于两个技术**    **C++中智能指针的实现主要依赖于两个技术概念：**  **1、析构函数**，对象被销毁时会被调用的一个函数，对于基于栈的对象而言，如果对象离开其作用域则对象会被自动销毁，而此时析构函数也自动会被调用。  **2、引用计数技术**，维护一个计数器用于追踪资源(如内存)的被引用数，当资源被引用时，计数器值加1，当资源被解引用时，计算器值减1。  **3、操作符重载。**  智能指针的大致实现原理就是在析构函数中，检查所引用对象的引用计数，如果引用计数为0，则真正释放该对象内存。    我们是怎么创建原生指针的：  // 创建一个不知所指的指针 char \*punknown ; // 创建空指针 char \*pnull = NULL; // 创建字符串指针 const char \*pstr = "Hello world!"; // 创建指向具体类的指针 SomeClass \*pclass = new SomeClass(); // 从同类型指针创建指针 SomeClass \*pclassother = pclass;  求我们的智能指针类，也能满足对应的用法：  // 创建一个不知所指的智能指针 SmartPointer spunknown; // 创建空智能指针 SmartPointer spnull = NULL; // 创建字符串的智能指针 SmartPointer spstr = "Hello world!"; // 创建指向具体类的智能指针 SmartPointer spclass = new SomeClass();  **实现智能指针版本1**  /\* \* file name : smartpointer.h \* desp : 智能指针版本v1 \*/ #ifndef \_\_SMARTPOINTER\_H\_\_ #define \_\_SMARTPOINTER\_H\_\_  template <typename T> // 将智能指针类定义成模板类 class SmartPointer { public: // 默认构造函数 SmartPointer(): mPointer(NULL)  { std::cout << "create unknown smart pointer." << std::endl;   } // 接收不同指针类型的构造函数 SmartPointer(T \*p): mPointer(p)  { std::cout << "create smart pointer at " << static\_cast<const void \*>(p) << std::endl;  } // 析构函数 ~SmartPointer()  { std::cout << "release smart pointer at " << static\_cast<const void \*>(mPointer) << std::endl; // 实现内存资源自动销毁机制 if (mPointer) delete mPointer;  } private: T \*mPointer; // 指向智能指针实际对应的内存资源，根据参数自动推导规则，定义内部资源指针类型 }; #endif // \_\_SMARTPOINTER\_H\_\_    上面的智能指针没有实现赋值和拷贝，  默认情况下C++编译器会为我们定义的类生成拷贝构造函数和赋值操作符的实现，但是对于我们的智能指针而言，使用系统默认为我们生成的赋值操作符的实现，是会有问题的。    **拷贝构造函数和赋值构造函数的区别：**  1）会调用拷贝构造函数的例子：  CExample A;   CExample B=A; //把B初始化为A的副本  2）会调用赋值构造函数的例子：  CExample A;   CExample C;   C = A; //现在需要一个对象赋值操作,被赋值对象的原内容被清除，并用右边对象的内容填充。  1）中“ CExample B=A;  **”语句**中的 "=" 在对象声明语句中，表示初始化。更多时候,这种初始化也可用括号表示。  例CExample B(A);  2）中，"=" 表示赋值操作。将对象 A 的内容复制到对象C;，这其中涉及到对象C 原有内容的丢弃，新内容的复制。    智能指针类中，有我们自己申请的内存，所以需要自己定义**拷贝构造函数和赋值构造函数（完成深拷贝）**  **2.2 定义构造函数和赋值操作符的函数原型**    **实现智能指针版本2**  /\*  \* file name : smartpointer.h \* desp : 智能指针版本v2 \*/ #ifndef \_\_SMARTPOINTER\_H\_\_ #define \_\_SMARTPOINTER\_H\_\_  template <typename T> // 将智能指针类定义成模板类 class SmartPointer { public: // 默认构造函数 SmartPointer():mPointer(NULL) {std::cout <<"Create null smart pointer."<< std::endl;}  // 接收不同指针类型的构造函数 SmartPointer(T \*p):mPointer(p) {std::cout <<"Create smart pointer at "<<static\_cast<const void\*>(p)<<std::endl;}  // 析构函数 ~SmartPointer(){ std::cout << "Release smart pointer at "<<static\_cast<const void\*>(mPointer)<<std::endl; // 实现内存资源自动销毁机制 if (mPointer) delete mPointer;  } // 拷贝构造函数 SmartPointer(const SmartPointer &other):mPointer(other.mPointer) { std::cout <<"Copy smart pointer at "<<static\_cast<const void\*>(other.mPointer)<<std::endl;  }  // 赋值操作符  SmartPointer &operator = (const SmartPointer &other) { // 处理自我赋值的情况 if (this == &other) return \*this; // 处理底层指针的释放问题 if (mPointer) delete mPointer; mPointer = other.mPointer;  std::cout <<"Assign smart pointer at "<<static\_cast<const void\*>(other.mPointer)<<std::endl; return \*this;  }   private: T \*mPointer; // 指向智能指针实际对应的内存资源，根据参数自动推导规则，定义内部资源指针类型 }; #endif // \_\_SMARTPOINTER\_H\_\_  还是存在一些问题：就是当有多个智能指针执行同一块底层资源，在释放时，每个指针都会去释放一次底层资源，这就造成了最后的 double free 错误  【C++】智能指针的原理和实现_运算符  【C++】智能指针的原理和实现_赋值_02  此时智能指针SmartPointer2将会变成野指针！  所以要引入计数：  引入引用计数的智能指针，其实现主要需要更新以下几点：  **1接收不同对象类型的构造函数**  这个构造函数实现，比较简单，直接将引用计数加1 即可。  **2析构函数**  析构函数的实现，不能再直接做delete操作，而是需要先对引用计数减1，当引用计数为0时，才做delete操作。  **3拷贝构造函数**  拷贝构造函数的实现，底层指针共享，然后将引用计数加1 即可。  **4赋值操作符**  赋值操作的实现，稍微复杂一些，涉及到将新指向对象的引用计数加1，将原指向对象的引用计数减1，如果有需要还要销毁原指向对象。这里有一点值得注意的地方，我们新的赋值操作的实现，不再需要if (this == &other) return \*this;语句处理自我赋值的情况，读者可自行分析一下我们新的赋值操作的实现为何不需要通过if语句去处理自我赋值的情况。    为了能够使用引用计数技术，我们的智能指针不能再像原生指针那样能用可以指向任意资源对象，我们的智能指针只能指向实现了存在方法incRefCount和方法decRefCount的资源类了。  **引用计数技术的智能指针v3版本：**  /\*  \* file name : smartpointer.h \* desp : 智能指针版本v3 \*/ #ifndef \_\_SMARTPOINTER\_H\_\_ #define \_\_SMARTPOINTER\_H\_\_  template <typename T> // 将智能指针类定义成模板类 class SmartPointer { public: // 默认构造函数 SmartPointer():mPointer(NULL) {std::cout <<"Create null smart pointer."<< std::endl;}  // 接收不同对象类型的构造函数 SmartPointer(T \*p):mPointer(p) { std::cout <<"Create smart pointer at "<<static\_cast<const void\*>(p)<<std::endl; /\*智能指针指向类T，引用计数加1\*/ if (mPointer) mPointer->incRefCount();  }  // 析构函数 ~SmartPointer(){ std::cout << "Release smart pointer at "<<static\_cast<const void\*>(mPointer)<<std::endl; // 实现内存资源自动销毁机制 if (mPointer && mPointer->decRefCount()==0) delete mPointer;  } // 拷贝构造函数 SmartPointer(const SmartPointer &other):mPointer(other.mPointer) { std::cout <<"Copy smart pointer at "<<static\_cast<const void\*>(other.mPointer)<<std::endl; // 引用计数加1 if(mPointer) mPointer->incRefCount();  }  // 赋值操作符  SmartPointer &operator = (const SmartPointer &other) { T \*temp(other.mPointer); // 新指向对象，引用计数值加1 if (temp) temp->incRefCount(); // 原指向对象，引用计数值减1，如果减1后引用计数为0 销毁原资源对象 if (mPointer && mPointer->decRefCount()==0) delete mPointer; // 智能指针指向新资源对象 mPointer = temp;  return \*this;  }   private: T \*mPointer; // 指向智能指针实际对应的内存资源，根据参数自动推导规则，定义内部资源指针类型 };  /\*引用计数基类\*/ class RefBase  {  public:  RefBase() : mCount(0){ }  void incRefCount(){  mCount++;   }  int decRefCount(){  return --mCount;  }  // 调试接口，返回对象当前引用计数  int getRefCount(){  return mCount;   }  virtual ~RefBase(){ } private:  int mCount;  };  #endif // \_\_SMARTPOINTER\_H\_\_   测试代码(sptestcase3.cpp)：   /\*  \* file name : sptestcase3.cpp \* desp : 智能指针测试代码 case3 测试智能指针的引用计数功能 \*/  #include <iostream> #include "smartpointer.h"  /\*继承于引用计数基类的SomeClass类\*/ class SomeClass: public RefBase{ public: SomeClass(){std::cout << "SomeClass default constructor !"<<std::endl;} ~SomeClass(){std::cout << "SomeClass deconstructor !"<<std::endl;} };  void testcase3(void) { SomeClass \*pSomeClass = new SomeClass(); //1 SmartPointer<SomeClass> spOuter = pSomeClass; std::cout << "SomeClass Ref Count (" << pSomeClass->getRefCount() << ") outer 1."<< std::endl;  { // inner 语句块 SmartPointer<SomeClass> spInner = spOuter; std::cout << "SomeClass Ref Count (" << pSomeClass->getRefCount() << ") inner."<< std::endl;  } std::cout << "SomeClass Ref Count (" << pSomeClass->getRefCount() << ") outer 2."<< std::endl; // delete pSomeClass ; 不需要也不能执行delete操作!  std::cout << "new another SomeClass class for spOuter."<< std::endl; SmartPointer<SomeClass> spOuter2 = new SomeClass(); spOuter = spOuter2;// 1处new出来的SomeClass将会被自动释放  }  int main(void) { testcase3(); return 0; }  还不够完美，我们使用指针指代资源对象，最终还是需要去访问资源对象的内容，那么智能指针如何满足这个需求呢？  2.1 解引用  我们使用指针最终还是需要通过指针去访问其所指向的资源对象的数据及方法，这个过程称为指针的​​解引用​​过程。指针的​​解引用​​可以使用​​\*​​运算符和​​->​​运算符；  我们需要重载​​\*​​运算符和​​->​​运算符：  template <typename T>  class SmartPointer { public: /\*重载运算符\* \*/ T& operator\* () const; /\*重载运算符-> \*/ T\* operator-> () const; };  在我们的智能指针类重载了运算符后，当我们这样使用智能指针时：  (\*SmartPointer).func(); SmartPointer->func();  其实等价于：  (\*(SmartPointer.operator\*())).func(); (SmartPointer.operator->())->func();  也就是说，我们在重载​​\*​​运算符时其返回的应该是智能指针所指的**资源对象实体**, 而​​->​​运算符则应该返回的是智能指针指针所指的**资源对象的内存地址**，既是我们的实现是这样的：  // 重载\*和-> 运算符  /\*重载运算符\* \*/  T& operator\* () const {return \*mPointer;};  /\*重载运算符-> \*/  T\* operator-> () const {return mPointer;}; |

#### 参考4：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 技术: C++ 智能指针实现  [技术](https://wizardmerlin.github.io/categories/%E6%8A%80%E6%9C%AF/)   * [cpp](https://wizardmerlin.github.io/tags/cpp/) * [智能指针](https://wizardmerlin.github.io/tags/%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88/)   说完了Boost和std中的智能指针, 也尝试实现看看。  引用计数, 标记擦除, 智能指针还有其他实现方式么?  本篇主要涉及:   * 尝试自己实现 引用计数类型的智能指针 * 仔细说说 auto\_ptr 和 unique\_ptr   **引子**  如果明白了RAII手法, 引用计数原理, 标记擦除原理, 那么实现一个scoped\_ptr, shared\_ptr, 或者unique\_ptr都不是难事儿. 难得的是实现的时候, 对于下面两个函数的处理:   * 赋值函数 * 拷贝构造函数   当然也包括右值问题.  **正文**  **智能指针实现机制**  按照其实现机制, 大致可以把智能指针分为如下几类:   * 引用计数机制 解释: 引用计数主要是使用系统记录对象被引用的次数。当对象被引用的次数变为0时，该对象即可被视为“垃圾”，从而可以被回收。使用引用计数做垃圾回收的算法的一个优点是实现很简单，与其它垃圾回收算法相比，该方法不会造成程序暂停，因为计数的增减与对象的使用时紧密结合的。 * 跟踪处理的垃圾回收机制 根据不同算法又可以重新划分   + 标记-清除(Mark-Sweep) 首先该算法将程序中正在使用的对象视为“根对象”，从根对象开始查找它们所引用的堆空间，并在这些堆空间上做标记。当标记结束后，所有被标记的对象就是可达对象或活对象，而没有被标记的对象就被认为是垃圾，在第二步的清扫阶段会被回收掉; 这种方法的特点是活的对象不会被移动，但是其存在会出现大量的内存碎片问题。   + 标记-整理(Mark-Compact) 这个算法标记的方法和标记-清除方法一样，但是标记完之后，不再遍历所有对象清扫垃圾了，而是将活跃的对象向“左”靠齐，这就解决了内存碎片的问题。 标记-整理算法有个特点就是移动活的对象，因此相应的，程序中所有对堆内存的引用都要更新。   + 标记-拷贝(Mark-Copy) 这种算法的一大特点就是将堆空间分为两部分：From，To。开始的时候我们只在From里分配，当From分配满的时候出发垃圾收集，这个时候会找出From空间里所有的存活对象，然后将这些存活的对象拷贝到To空间里。这样From空间里剩下的就都全是垃圾，而且对象拷贝到To里，在To里是紧凑排列的。这个事儿做完了之后From和To的角色就转变了一下。原来的From变成了To，原来的To变成了现在的From。现在又可以在这个完全是空的From里分配了。这个算法实现起来也很简单，高效（Sun JVM的新生代的垃圾回收就使用了这种算法）。不过这个算法有一个问题，堆的利用率只有一半了，这对那些内存占用率比较低的对象还算好，如果随着应用的内存占用率的增高，问题就出现了，第一个要拷贝的对象太多了，还有可能无法回收内存了。程序失败了。 * RAII机制(资源和对象绑定, 局部对象自动销毁) 这种机制把资源的声明周期和对象的声明周期绑定, 存在的问题就是资源的当前使用者是唯一的, 出现赋值则情况需要特殊处理.   **引用计数简单实现**  如果把引用计数器也当做一种(共享)资源, 那么很容易可以给出一种实现:  **template<class \_T>  class SmartPtr  {  private:   \_T \*key;   size\_t \*counter;   void decrease() {   if (--\*counter == 0)   delete key;   }   public:   SmartPtr (\_T \*t = NULL) : key(t), counter(new size\_t(1)) {   }    SmartPtr(const SmartPtr & sp) {   key = sp.key;   counter = sp.counter;   (\*counter)++;   }    SmartPtr & operator = (const SmartPtr & sp) {   decrease();   key = sp.key;   counter = sp.counter;   (\*counter)++;   }    virtual ~SmartPtr() {   decrease();   }  };** ```  核心实现代码非常简单: \* 拷贝构造是共同拥有, 指向统一资源(因为拷贝构造时, 原来是没有指向的);  \* 赋值是转移所有权(赋值是原来指向的内容要判断一下是否减少引用计数, 然后再接收之后的指向).    ## auto\_ptr  说先要说这个智能指针当前已经废弃了, 平常大概也就会这么用: ```c++ auto\_ptr<T> pt( new T );//只有new, 没有delete调用  代替原来的:   |  |  | | --- | --- | | 1 2 3 | T\* pt( new T );  //... delete pt; |   当然和其他职能指针一样, 也包含其他方法: reset()函数   |  |  | | --- | --- | | 1 2 3 | <memeory> auto\_ptr<T> pt( new T(1) );  pt.reset( new T(2) ); // 删除由"new T(1)"分配出来的第一个T, 改变指向 |   以及成员函数:   |  |  | | --- | --- | | 1 2 | X\* get() const throw(); //返回保存的指针, 原指针中仍保留指向空间 X\* release() const throw(); //返回保存的指针, 原指针已经不再指向原来空间 |   这个智能指针实现的时候, 在唯一所有权的问题上没有处理好, 或者说赋值函数没有处理好. 例如下面一段代码:  **void TestFunc(auto\_ptr<A> Obj) {  Obj->SetA(20);  cout << Obj->GetA() << endl; }  int main() {  auto\_ptr<A> pAObj(new A(10));  TestFunc(pAObj); //注意这里发生值拷贝  cout << pAObj->GetA() << endl; //pAObj 智能指针此时已经不再指向先关数据了. }**  然后最后一句报错了, 也就是说, auto\_ptr不能共享内存, 在同一时间, 只有一个auto\_ptr指向一个指定的内存(托管转移). 为什么不能? 因为他的实现机制是, 一个对象绑定它的成员资源.   * 利用特点”栈上对象在离开作用范围时会自动析构” * 对于动态分配的内存，其作用范围是程序员手动控制的，这给程序员带来了方便但也不可避免疏忽造成的内存泄漏，毕竟只有编译器是最可靠的 * auto\_ptr通过在栈上构建一个对象a，对象a中wrap了动态分配内存的指针p，所有对指针p的操作都转为对对象a的操作   或者你从他的源码也可以看到:   |  |  | | --- | --- | |  | **// 赋值构造函数 auto\_ptr& operator=(auto\_ptr& \_\_a) \_\_STL\_NOTHROW {  if (&\_\_a != this) {  delete \_M\_ptr;  \_M\_ptr = \_\_a.release();//放弃\_\_a保留的原指针, 转让给当前接受者  }  return \*this; }** |   既然不能共享所有权, 那么凡是调用 operator=() 的操作都是危险的, 比如吧auto\_ptr放入容器内, 即   |  |  | | --- | --- | |  | vector<auto\_ptr<MyClass>>m\_example; |   (当然你可以使用std:ref智能引用进行包装, 不过一拷贝, 原对象就失效, 总的来说还是不能放入容器).  正是auto\_ptr实现机制上的问题(原件一拷贝就失效, 你受得了?), 所以后来引入标准库的 std::unique\_ptr用静态检验的方式去确认用户是否需要在赋值后转移所有权, 并且采用引用计数的shared\_ptr可以共享所有权(除了一个相互引用比较烦人以外). 唯一可惜的是, shared\_ptr(配合weaked\_ptr使用)效率会有微小损失, 而auto\_ptr则没有. 哦, scoped\_ptr 这又是一个对所有权的几种做法: 不允许转移, 也就是说保证资源从一而终(此时也保障了唯一所有者).  补充: auto\_ptr还不能存储数组, 即auto\_ptr 原因可能是因为析构的时候, 没有调用delete[]处理, 而只有 delete. unique\_ptr std::unique\_ptr 它也是通过指针的方式来管理对象资源, 并且在 unique\_ptr 的生命期结束后释放该资源. unique\_ptr 持有对对象的独有权, 即两个 unique\_ptr 不能指向一个对象, 不能进行复制操作**只能进行移动操作**(某种意义上禁止了拷贝语义, 只保留move语义, 查看函数原型就能发现), 即右值(临时变量)转移给某个具名值(左值)或者函数返回. 那么它为什么就比 auto\_ptr好呢?  即unique\_ptr赋值拷贝构造有多个重载, 但是都不具备拷贝语义, 而只有移动语义. (从其原型看出, 它其实是强制你在需要转义的时候使用move语句, 因为根本找不到T&, 只有T&&)  转移仅仅发生在:   * unique\_ptr 所要接收的对象即将销毁(右值) * unique\_ptr 所要接收的对象显示的放弃了所有权, 比如release(), 即左值或者非const引用电泳move语句.   通俗的说: auto\_ptr是可以说你随便赋值,但赋值完了之后原来的对象就不知不觉的报废.搞得你莫名其妙.而unique就干脆不让你可以随便去复制,赋值.如果实在想传个值就哪里,显式的说明内存转移std:move一下.然后这样传值完了之后,之前的对象也同样报废了.只不过整个move你让明显的知道这样操作后会导致之前的unique\_ptr对象失效.  显示的说明你要转移所有权, 而不再是 auto\_ptr 莫名其妙地隐式转移权限; 直接禁止拷贝语义, 保留转移语义(只有右值引用的版本允许调用)  对比一下就知道:  **auto\_ptr<int> ap(new int(88)); auto\_ptr<int> one (ap) ; // ok auto\_ptr<int> two = one; //ok  //但unique\_ptr不支持上述操作 unique\_ptr<int> ap(new int(88)); unique\_ptr<int> one (ap) ; // 会出错 unique\_ptr<int> two = one; //会出错  //uniuqe\_ptr 可以进行移动操作 //例如作为函数的返回值(直接右值) unique\_ptr<int> GetVal() {  unique\_ptr<int> up(new int(88));  return up; }   unique\_ptr<int> uPtr = GetVal(); //ok  //或者直接move unique\_ptr<int> up(new int(88) ); //这里是显式的所有权转移, 把up所指的内存转给uPtr2了 unique\_ptr<int> uPtr2 = std:move(up) ;**  cppreference上的例子:  **#include <iostream> #include <memory>  struct Foo { // object to manage  Foo() { std::cout << "Foo ctor\n"; }  Foo(const Foo&) { std::cout << "Foo copy ctor\n"; }//真正拷贝  Foo(Foo&&) { std::cout << "Foo move ctor\n"; } //真正转移  ~Foo() { std::cout << "~Foo dtor\n"; } };    struct D { // deleter  D() {};  D(const D&) { std::cout << "D copy ctor\n"; }  D(D&) { std::cout << "D non-const copy ctor\n";}  D(D&&) { std::cout << "D move ctor \n"; }  void operator()(Foo\* p) const {  std::cout << "D is deleting a Foo\n";  delete p;  }; };  int main() {  std::cout << "Example constructor(1)...\n";  std::unique\_ptr<Foo> up1; // up1 is empty  std::unique\_ptr<Foo> up1b(nullptr); // up1b is empty   std::cout << "\nExample constructor(2)...\n";  {  std::unique\_ptr<Foo> up2(new Foo); //up2 now owns a Foo  } // Foo deleted   std::cout << "\nExample constructor(3)...\n";  D d;  { // deleter type is not a reference  std::unique\_ptr<Foo, D> up3(new Foo, d); // deleter copied  }  { // deleter type is a reference   std::unique\_ptr<Foo, D&> up3b(new Foo, d); // up3b holds a reference to d  }   std::cout << "\nExample constructor(4)...\n";  { // deleter is not a reference   std::unique\_ptr<Foo, D> up4(new Foo, D()); // deleter moved  }   std::cout << "\nExample constructor(5)...\n";  {  std::unique\_ptr<Foo> up5a(new Foo);  std::unique\_ptr<Foo> up5b(std::move(up5a)); // ownership transfer  }   std::cout << "\nExample constructor(6)...\n";  {  std::unique\_ptr<Foo, D> up6a(new Foo, d); // D is copied  std::unique\_ptr<Foo, D> up6b(std::move(up6a)); // D is moved   std::unique\_ptr<Foo, D&> up6c(new Foo, d); // D is a reference  std::unique\_ptr<Foo, D> up6d(std::move(up6c)); // D is copied  }   std::cout << "\nExample constructor(7)...\n";  {  std::auto\_ptr<Foo> up7a(new Foo);  std::unique\_ptr<Foo> up7b(std::move(up7a)); // ownership transfer  } }**  可以看到凡是涉及到左值的, 全部都用了move()调用了右值引用的重载版本.  个人觉得, unique支持容器, 并且支持数组类型, unique\_ptr<T[]>, 这才是重点.  最初我也怀疑, unique不支持拷贝, 怎么能放入容器: auto\_ptr不可做为容器元素, 而unique\_ptr也同样不能直接做为容器元素.  实际上, 发现用右值, 或者移动语义是可以实现的:  **unique\_ptr<int> sp(new int(88) ); vector<unique\_ptr<int> > vec; vec.push\_back(std::move(sp)); //显示地说明你要放入 //vec.push\_back( sp ); 这样不行,会报错的. //cout<<\*sp<<endl;但这个也同样出错,说明sp添加到容器中之后,它自身报废了.** |

# 8.C++ 实现即时通信软件

#### 参考1：<https://github.com/adroitwolf/IM>

#### 参考2：<https://github.com/KangLin/RabbitIm>

#### 参考3：<https://gitee.com/jeffreylee/QQrobot>

#### 参考4：<https://github.com/balloonwj/flamingo>

# 9.C++实现课程管理系统

参考1.GitHub：<https://github.com/kennycaiguo/CourseManageSystem-linux-sys>

参考2.本地文件：<C++实现课程管理系统.docx>

# 10.C++ 实现银行排队服务模拟

#### 参考1：

|  |
| --- |
| **C++ 实现银行排队服务模拟**  教程简介：使用 C++对银行排队服务进行模拟，以事件驱动为核心思想，手动实现模板链式队列、随机数产生器等内容，进而学习概率编程等知识。作为可选进阶，这个模型同时还能稍加修改的应用到 CPU 资源争夺模型中。  本教程由[欧龙崎](https://link.zhihu.com/?target=https%3A//www.shiyanlou.com/teacher/29879)发布在[实验楼](https://link.zhihu.com/?target=https%3A//www.shiyanlou.com/courses/557)，完整教程及在线练习地址：[C++ 实现银行排队服务模拟](https://link.zhihu.com/?target=https%3A//www.shiyanlou.com/courses/557)  如果您也想通过实验楼课程的形式分享自己的实践经验，欢迎投稿给我们：[项目投稿通道](https://link.zhihu.com/?target=https%3A//www.shiyanlou.com/contribute)  **一、概述**  **实验所需的前置知识**   * C++ 基本语法知识   **实验所巩固并运用的知识**   * OOP 编程思想 * std::rand() 函数原理 * 概率编程 * 排队理论 * 链式队列数据结构及其模板实现 * 事件驱动的设计 * 蒙特卡洛方法 * C++ 动态内存管理和设计理念 * CPU 资源争夺模型 * 时间片轮转调度   **要解决的问题**  蒙特卡洛方法这个名字听起来很高大上，但它的本质其实是使用计算机的方法对问题进行模拟和复现。本次实验将使用蒙特卡洛方法来模拟银行排队这个问题：  端午节当天，某个银行从早上八点开始服务并只服务到中午十二点就停止营业。假设当天银行只提供了 w 个服务窗口进行服务，问：   1. 平均每分钟有多少个顾客抵达银行? 2. 平均每个顾客占用服务窗口的时间是多少？   我们先来分析一下这个业务的逻辑：  首先我们要分析银行提供服务的逻辑。在银行服务中，所有顾客都是通过取号排队的方式等待服务的，这和火车站买票有所不同，在火车站买票时，顾客必须在某一个窗口所排的队列下进行排队，且无法变更自己所属的窗口，否则只能从队尾重新排队。换句话说，对于银行提供的服务来说，所有用户都是位于同一个队列上的，当某个服务窗口可用时，才会从排队队列的队首取出一个新的用户来办理银行业务。即代码实现过程中，服务窗口可以创建 w 个，但只需要实现一个顾客队列即可。  其次，对于顾客而言，有两个属性是能够被抽象出来的：   1. 到达银行的时间； 2. 需要服务的时间。   并且，这两个属性是随机的。到此，我们整个的排队模型就变成了：  https://pic2.zhimg.com/80/72209348c918d96afb72b0b06b84258d_720w.png下面我们来详细对这个问题的实现逻辑进行分析，让我们的程序能够给出类似下面的结果：  https://pic1.zhimg.com/80/ade7bc8f1b3a9189861e0aa1642b9524_720w.png  **第一部分、银行排队系统的设计与建模**  **一、计算机中的随机**  **std::rand() 函数的原理**  C++ 中的 std::rand() 函数产生的随机数并不是真正意义上的随机数，它并不服从数学上的均匀分布。为了使我们的模拟系统变得更加真实，我们需要知道 std::rand() 函数的原理。  std::rand() 生成的是一个随机的二进制序列（在硬件底层更好实现），这个序列的每一位0或者1的概率都是相等的。而对于 std::rand()%n 这个运算，会在 [0, n-1] 之间生成随机数，所以，如果 n-1 的二进制表示的值不都是由 1 组成，那么这里面的数是不会从均匀分布了（因为某些位可能不能为 1）。  所以，当且仅当 [0, n-1] 中的随机数可以用这个序列的子序列表示时，才能满足均匀分布。换句话说，仅当 n-1 的二进制数全为1 时，0，1出现的概率才是均等的。  我们先来实现随机这个类：  //  // Random.hpp  // QueueSystem  //  #ifndef Random\_hpp  #define Random\_hpp  #include <cstdlib>  #include <cmath>  class Random {  public:  // [0, 1) 之间的服从均匀分布的随机值  static double uniform(double max = 1) {  return ((double)std::rand() / (RAND\_MAX))\*max;  }  };  #endif /\* Random\_hpp \*/  这样的话，当我们调用 Random::uniform() 时，便能获得真正的服从均匀分布的随机数了。当指定参数后，便能够生成 [0, max) 之间的随机值了。  **二、主函数逻辑设计**  对于一个银行而言，对外界来说只需要提供两个参数：   1. 总共的服务时间 2. 服务窗口的数量   所以我们希望实现这样的代码：  //  // main.cpp  // QueueSystem  //  #include "QueueSystem.hpp"  #include <iostream>  #include <cstdlib>  int main() {  std::srand((unsigned)std::time(0)); // 使用当前时间作为随机数种子  int total\_service\_time = 240; // 按分钟计算  int window\_num = 4;  int simulate\_num = 100000; // 模拟次数  QueueSystem system(total\_service\_time, window\_num);  system.simulate(simulate\_num);  std::cout << "The average time of customer stay in bank: "  << system.getAvgStayTime() << std::endl;  std::cout << "The number of customer arrive bank per minute: "  << system.getAvgCustomers() << std::endl;  return 0;  }  **三、对象及逻辑设计**  总结一下，现在我们需要实现的东西有：   1. 服务窗口类(会被创建 w 个) 2. 顾客队列类(只会被创建一个) 3. 顾客结构(包含两个随机属性: 到达时间, 服务时间)   为了更好练习 C++，我们会弃用诸如 vector 这些快捷编码的标准库来进行『过度编码』，自行编写模板类。  根据前面的问题描述，我们可以初步确定这样一些类的设计需求：   1. QueueSystem 类: 负责整个队列系统的模拟； 2. ServiceWindow 类: 队列系统的服务窗口对象，每当一个银行创建时，服务窗口会被创建，为了让整个问题更加灵活，我们假设需要创建 window\_num 个窗口； 3. Queue 类: 银行队列系统的顾客排队的队列； 4. Random 类: 在第二节中已经讨论过。   然而，在设计 ServiceWindow 之前，我们要考虑 ServiceWindow 类到底要放置什么成员，首先，对于一个服务窗口，会有一个顾客属性，用于存放顾客。另一方面，一个窗口只会有两种状态：要么正在服务（被占用），要么空闲。因此 ServiceWindow 中首先会有下面的枚举：  //  // ServiceWindow.hpp  // QueueSystem  //  enum WindowStatus {  SERVICE,  IDLE,  };  既然我们要在 ServiceWindow 中存放顾客，由于顾客本身并不需要提供什么方法，因此可以直接将顾客设计为一个结构体 Customer，同时，顾客也会成为等待队列中的一员。所以，Customer 也可以被称之为队列的一个 Node，此外，每个顾客说需要的服务时间是随机的，但是到达时间并不应该由顾客自身确定（我们在下一节再讨论为什么），所以Customer结构的默认构造应该被设计出来：  //  // Node.h  // QueueSystem  //  #ifndef Node\_h  #define Node\_h  #include "Random.hpp"  #define RANDOM\_PARAMETER 100  struct Node {  int arrive\_time;  int duration;  struct Node \*next;  // 默认到达事件为0，需要服务的事件是随机的  Node(int arrive\_time = 0,  int duration = Random::uniform(RANDOM\_PARAMETER)):  arrive\_time(arrive\_time),  duration(duration),  next(nullptr) {}  };  typedef struct Node Node;  typedef struct Node Customer;  #endif /\* Node\_h \*/  那么，结合前面的 WindowStatus枚举和 Customer结构，我们的 ServiceWindow 类可以这样设计，因为窗口本身涉及的操作还算是比较简单，比如设置窗口状态是否繁忙，获取当前服务顾客的到达时间来方便后续计算等等，因此我们直接将其设计成类内的 inline 函数：  //  // ServiceWindow.hpp  // QueueSystem  //  #ifndef ServiceWindow\_hpp  #define ServiceWindow\_hpp  #include "Node.hpp"  enum WindowStatus {  SERVICE,  IDLE,  };  class ServiceWindow {  public:  inline ServiceWindow() {  window\_status = IDLE;  };  inline bool isIdle() const {  if (window\_status == IDLE) {  return true;  } else {  return false;  }  }  inline void serveCustomer(Customer &customer) {  this->customer = customer;  }  inline void setBusy() {  window\_status = SERVICE;  }  inline void setIdle() {  window\_status = IDLE;  }  inline int getCustomerArriveTime() const {  return customer.arrive\_time;  }  inline int getCustomerDuration() const {  return customer.duration;  }  private:  Customer customer;  WindowStatus window\_status;  };  #endif /\* ServiceWindow\_hpp \*/  **三、事件驱动的设计**  有了上面的这些设计，似乎我们只要编写好用户排队队列，就已经足够描述整个排队的系统了，然而，在上面的设计中，还有一个很大的问题，那就是：整个系统还处于静止状态。当顾客位于等待队列时，窗口什么时候服务下一个顾客，如何处理这里面的逻辑，到目前为止，我们都没有思考过。  为了让整个系统『运行』起来，我们还要考虑整个系统的运行时间线。这里我们给出一种**事件驱动**的设计。  在前面的分析中，我们知道整个系统中，无非出现两种事件：   1. 有顾客到达 2. 有顾客离开   其中，第二种顾客离开的事件，同时还包含了窗口服务等待队列中的下一个顾客这个事件。所以，我们如果能够维护一个事件列表，那么就能够驱动整个队列系统的运行了。因为，当事件发生时，我们通知这个队列系统更新他自身的状态即可。  综上所述，我们可以先设计事件表中的事件结构：  //  // Event.hpp  // QueueSystem  //  #ifndef Event\_hpp  #define Event\_hpp  #include "Random.hpp"  #define RANDOM\_PARAMETER 100  struct Event {  int occur\_time;  // 使用 -1 表示到达事件, >=0 表示离开事件, 同时数值表示所离开的服务窗口  int event\_type;  Event\* next;  // 默认为到达事件，发生事件随机  Event(int occur\_time = Random::uniform(RANDOM\_PARAMETER),  int event\_type = -1):  occur\_time(occur\_time),  event\_type(event\_type),  next(nullptr) {}  };  #endif /\* Event\_hpp \*/  这里我们使用了一个小小的 trick，那就是用整数来表示事件的类型，而不是简单的使用枚举。  这是因为，对于 ServiceWindow 来说，我们可以使用数组来管理多个 ServiceWindow，那么对应的事件类型如果涉及为整数，事件类型就可以同时作为 ServiceWindow 的索引下标了，当 event\_type 大于等于 0 时，数值还表示离开的服务窗口。  又因为事件列表、顾客队列，本质上可以归类为同一个结构，那就是队列：只不过他们的入队方式有所差异，对于事件列表而言，入队方式必须按发生事件的时间顺序入队，而对于顾客，则是直接添加到队尾。考虑到了这一点，我们便能很容易的利用模板来设计队列的基本需求了：  //  // Queue.hpp  // QueueSystem  //  #ifndef Queue\_hpp  #define Queue\_hpp  #include <iostream>  #include <cstdlib>  #include "Event.hpp"  // 带头结点的队列  template <typename T>  class Queue  {  public:  Queue();  ~Queue();  void clearQueue(); // 清空队列  T\* enqueue(T &node);  T\* dequeue();  T\* orderEnqueue(Event &event); // 只适用于事件入队  int length();  private:  T \*front; // 头结点  T \*rear; // 队尾  };  #endif /\* Queue\_hpp \*/  **四、QueueSystem**  经过前面的讨论，我们已经完成了对所有基本结构的设计，根据这些设计，我们能够初步确定我们要实现的队列系统的基本结构。  首先，根据对主函数的设计，初始化整个队列系统我们需要两个参数：   1. 银行的总服务时间(分钟) int total\_service\_time 2. 银行开放的服务窗口数 int window\_num   其次，我们需要 QueueSystem 发开放至少三个接口：   1. 模拟 simulate() 2. 获得顾客平均逗留时间 getAvgStayTime() 3. 获得平均每分钟顾客数 getAvgCustomers()   第三，内部需要实现的内容包括：   1. 系统运行前的初始化 init() 2. 让系统运行的 run() 3. 系统结束一次运行的清理工作 end()   第四，整个系统需要管理的核心成员有：   1. 可供服务的窗口 ServiceWindow\* windows 2. 顾客等待队列 Queue<Customer> customer\_list 3. 事件列表 Queue<Event> event\_list 4. 当前的系统事件 Event\* current\_event   第五，处理事件的方法：   1. 处理顾客到达事件 void customerArrived() 2. 处理顾客离开事件 void customerDeparture()   最后，我们所希望的平均顾客逗留时间和平均每分钟的顾客数涉及的四个变量：   1. 顾客的总逗留时间 int total\_customer\_stay\_time 2. 一次运行中系统服务的中顾客数量 int total\_customer\_num 3. 每分钟平均顾客数 double avg\_customers 4. 顾客平均逗留时间 double avg\_stay\_time   事实上，可以预见的是，在处理顾客服务逻辑的时候，我们还需要一个方法getIdleServiceWindow 来获取当前服务窗口的状态，从而增加代码的复用度。  所以，整个 QueueSystem 类的代码设计为：  //  // QueueSystem.hpp  // QueueSystem  //  #ifndef QueueSystem\_hpp  #define QueueSystem\_hpp  #include "Event.hpp"  #include "Queue.hpp"  #include "ServiceWindow.hpp"  class QueueSystem {  public:  // 初始化队列系统  QueueSystem(int total\_service\_time, int window\_num);  // 销毁  ~QueueSystem();  // 启动模拟  void simulate(int simulate\_num);  inline double getAvgStayTime() {  return avg\_stay\_time;  }  inline double getAvgCustomers() {  return avg\_customers;  }  private:  // 让队列系统运行一次  double run();  // 初始化各种参数  void init();  // 清空各种参数  void end();  // 获得空闲窗口索引  int getIdleServiceWindow();  // 处理顾客到达事件  void customerArrived();  // 处理顾客离开事件  void customerDeparture();  // 服务窗口的总数  int window\_num;  // 总的营业时间  int total\_service\_time;  // 顾客的逗留总时间  int customer\_stay\_time;  // 总顾客数  int total\_customer\_num;  // 核心成员  ServiceWindow\* windows;  Queue<Customer> customer\_list;  Queue<Event> event\_list;  Event\* current\_event;  // 给外部调用的结果  double avg\_customers;  double avg\_stay\_time;  };  #endif /\* QueueSystem\_hpp \*/  **本节回顾**  在这一节中，我们设计了整个银行排队系统的基本逻辑，并借鉴了事件驱动的思想设计了驱动队列系统的事件类。本节中我们一共创建了：   1. Event.hpp 2. Node.hpp 3. Queue.hpp 4. Random.hpp 5. ServiceWindow.hpp 6. QueueSystem.hpp 7. main.cpp   现在我们的代码还不能够直接运行，本节我们先关注理清我们的业务逻辑。在下一节中，我们将实现这些代码的详细逻辑，这包括：   1. Queue.hpp 中模板链式队列的具体实现 2. QueueSystem.cpp 中的详细服务逻辑 3. Random.hpp 中更复杂的随机概率分布   在这些实现中，我们将进一步巩固下面的知识的运用：   1. C++ 类模板 2. 链式队列的数据结构 3. 概率编程 4. C++ 中内存管理和设计理念 |

#### 参考2：

gitee：<https://gitee.com/arrti/CPP_Exercises>

本地源码：<CPP_Exercises-master>

# 11.C++操作 redis 实现异步订阅和发布

#### 参考1：

<https://blog.csdn.net/chenmeng729970897/article/details/115068630>

#### 参考2：

GitHub：<https://github.com/zjbhuxian/pubsub>

已经下载的本地资源：<pubsub-redis-op-master>

# 12.C++ 使用 openGL 实现吃豆人游戏

参考1：<https://blog.csdn.net/qq_40086382/article/details/118539227>

参考2：<https://codeantenna.com/a/okUSjj3kfu>

已经下载的项目：<Pacman-吃豆人>

# 13.C++ 解决迷宫问题

#### 参考1：

|  |
| --- |
| **#include<iostream.h>**  **#include<iomanip.h>**  **#define M 11**  **#define N 10**  **//默认迷宫结构**  **char maze\_array[M][N]=**  **{**  **{1,1,1,1,1,1,1,1,1,1},**  **{0,0,0,1,0,0,0,1,0,1},**  **{1,0,0,1,0,0,0,1,0,1},**  **{1,0,0,1,0,1,1,0,1,1},**  **{1,0,1,1,1,0,0,1,0,1},**  **{1,0,0,0,1,0,0,0,0,1},**  **{1,0,1,0,0,0,1,0,1,1},**  **{1,0,1,1,1,1,0,0,1,1},**  **{1,1,1,0,0,0,1,0,1,1},**  **{1,1,1,0,0,0,0,0,0,0},**  **{1,1,1,1,1,1,1,1,1,1}**  **};**  **//记录迷宫结点的路过次数**  **int maze\_node\_passby\_array[M][N]=**  **{**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0},**  **{0,0,0,0,0,0,0,0,0,0}**  **};**    **//输出当前的迷宫局势**  **void print\_maze();**  **//输出记录路径结点出现次数的数组**  **void print\_maze\_node\_passby\_array();**      **//路径结点类**  **class Maze\_Path\_Node**  **{**  **private:**  **int i;**  **int j;**  **int path\_direction;**  **int number;**  **public:**  **Maze\_Path\_Node(int m=0,int n=0,int direct=0,int c=1)//构造函数**  **{**  **i=m;**  **j=n;**  **path\_direction=direct;**  **number=c;**  **}**  **Maze\_Path\_Node(Maze\_Path\_Node& t)//拷贝构造函数**  **{**  **i=t.get\_i();**  **j=t.get\_j();**  **path\_direction=t.get\_path\_direction();**  **number=t.get\_number();**  **}**  **~Maze\_Path\_Node() //析构函数**  **{**  **}**    **bool operator!=(Maze\_Path\_Node& t)**  **{**  **if(i==t.get\_i()&&j==t.get\_j())//&&path\_direction==t.get\_path\_direction()&&maze\_node\_passby\_array[(\*this).get\_i()][(\*this).get\_j()]==maze\_node\_passby\_array[t.get\_i()][t.get\_j()]**  **return 0;**  **else**  **return 1;**  **}**  **Maze\_Path\_Node& operator=(Maze\_Path\_Node& t)**  **{**  **if(this!=&t)**  **{**  **i=t.get\_i();**  **j=t.get\_j();**  **path\_direction=t.get\_path\_direction();**  **number=t.get\_number();**  **}**  **return \*this;**  **}**    **void print\_maze\_path\_node()**  **{**  **cout<<"("<<get\_i()<<","**  **<<get\_j()<<","**  **<<get\_path\_direction()**  **<<","**  **<<maze\_node\_passby\_array[get\_i()][get\_j()]<<")"<<endl<<endl;**  **}**    **int get\_i(){return i;}**  **int get\_j(){return j;}**  **int get\_number(){return number;}**  **int get\_path\_direction(){return path\_direction;}**    **};**    **//入口和出口坐标**  **Maze\_Path\_Node maze\_entrance(1,0,0);**  **Maze\_Path\_Node maze\_exit(9,9,0);**    **//路径类**  **class Maze\_Path**  **{**  **private:**    **Maze\_Path\_Node maze\_path[M\*N]; //路径**    **int m,n; //路径结点的行列坐标**  **int direction; //默认路径的前进方向**  **int maze\_path\_length; //路径长度**    **public:**  **Maze\_Path()//**  **{**  **m=maze\_entrance.get\_i();**  **n=maze\_entrance.get\_j();**  **direction=get\_next\_direction();**  **maze\_path\_length=0;**  **//maze\_node\_passby\_array[maze\_entrance.get\_i()][maze\_entrance.get\_j()]=1;**  **}**  **~Maze\_Path(){}**      **//返回路径长度**  **int get\_maze\_path\_length(){return maze\_path\_length;}**    **//打印路径**  **void print\_maze\_path();**    **//把路径节点存入一维数组中！**  **void push\_maze\_path\_node(Maze\_Path\_Node m);**    **//判断路径是否有回路**  **int is\_back();**    **//找到下一位置，并且设置为2，重置全局变量direction！**  **void one\_more\_step();**    **//计算得到路径**  **void get\_maze\_path();**    **//简化路径**  **void sim\_maze\_path();**    **//计算出在此结点基础上，下一步移动方向**  **int get\_next\_direction();**      **//删除重复路径**  **void delete\_useless\_path(int i,int m)**  **{**  **int count\_i,count\_m;**  **for(count\_i=i,count\_m=m;count\_m<=maze\_path\_length;count\_i++,count\_m++)**  **{**  **maze\_path[count\_i]=maze\_path[count\_m];**  **maze\_node\_passby\_array[maze\_path[count\_i].get\_i()][maze\_path[count\_i].get\_j()]--;**  **if(maze\_node\_passby\_array[maze\_path[count\_i].get\_i()][maze\_path[count\_i].get\_j()]==0)**  **maze\_array[maze\_path[count\_i].get\_i()][maze\_path[count\_i].get\_j()]=0;**  **}**    **maze\_path\_length=maze\_path\_length-(m-i);**  **cout<<endl<<"abcdefghijklmnopqrstuvwxyz"<<endl;**  **print\_maze\_path();**  **print\_maze();**  **print\_maze\_node\_passby\_array();**    **}**      **};**    **void main()**  **{**  **/\***  **Maze\_Path\_Node node(10,10,-1);**  **node.print\_maze\_path\_node();**  **Maze\_Path\_Node nodex=node;**  **nodex.print\_maze\_path\_node();**  **\*/**  **Maze\_Path maze\_path\_curr;**    **print\_maze();**  **maze\_path\_curr.get\_maze\_path();**  **maze\_path\_curr.print\_maze\_path();**    **print\_maze();**  **print\_maze\_node\_passby\_array();**    **maze\_path\_curr.sim\_maze\_path();**  **print\_maze();**  **print\_maze\_node\_passby\_array();**    **}**    **//输出当前的迷宫局势**  **void print\_maze()**  **{**  **int x,y;**  **cout<<"输出迷宫局势："<<endl;**  **for(x=0;x<=M-1;x++)**  **{**  **for(y=0;y<=N-1;y++)**  **cout<<" "**  **<<maze\_array[x][y];**  **cout<<endl;**  **}**  **cout<<endl**  **<<endl**  **<<endl**  **<<endl;**  **}**    **//输出记录路径结点出现次数的数组**  **void print\_maze\_node\_passby\_array()**  **{**  **int x,y;**  **cout<<"输出记录路径结点出现次数的数组："<<endl;**  **for(x=0;x<=M-1;x++)**  **{**  **for(y=0;y<=N-1;y++)**  **cout<<" "**  **<<maze\_node\_passby\_array[x][y];**  **cout<<endl;**  **}**  **cout<<endl**  **<<endl**  **<<endl**  **<<endl;**  **}**    **//打印路径**  **void Maze\_Path::print\_maze\_path()**  **{**  **cout<<"迷宫路径输出："<<endl;**  **for(int g=0;g<maze\_path\_length;g++)**  **{**  **cout<<"第 "<<setw(2)<<g+1<<" 个路径结点为："<<endl;**  **maze\_path[g].print\_maze\_path\_node();**  **}**  **cout<<endl<<"共有 "<<get\_maze\_path\_length()<<" 步，走出迷宫！"<<endl<<endl;**  **}**    **//计算出在此结点基础上，下一步移动方向**  **int Maze\_Path::get\_next\_direction()**  **{**    **//向右**  **if(direction==0)**  **{**    **if(m>=1&&maze\_array[m-1][n]!=1)**  **{**  **return 3;**  **}**  **else**  **if(n<=N-2&&maze\_array[m][n+1]!=1)**  **{**  **return 0;**  **}**  **else**  **if(m<=M-2&&maze\_array[m+1][n]!=1)**  **{**  **return 1;**  **}**  **else**  **if(n>=1&&maze\_array[m][n-1]!=1)**  **{**  **return 2;**  **}**    **}**  **//向下**  **else**  **if(direction==1)**  **{**  **if(n<=N-2&&maze\_array[m][n+1]!=1)**  **{**  **return 0;**  **}**  **else**  **if(m<=M-2&&maze\_array[m+1][n]!=1)**  **{**  **return 1;**  **}**  **else**  **if(n>=1&&maze\_array[m][n-1]!=1)**  **{**  **return 2;**  **}**  **else**  **if(m>=1&&maze\_array[m-1][n]!=1)**  **{**  **return 3;**  **}**  **}**  **//向左**  **else**  **if(direction==2)**  **{**  **if(m<=M-2&&maze\_array[m+1][n]!=1)**  **{**  **return 1;**  **}**  **else**  **if(n>=1&&maze\_array[m][n-1]!=1)**  **{**  **return 2;**  **}**  **else**  **if(m>=1&&maze\_array[m-1][n]!=1)**  **{**  **return 3;**    **}**  **else**  **if(n<=N-2&&maze\_array[m][n+1]!=1)**  **{**  **return 0;**    **}**  **}**  **//向上**  **else**  **if(direction==3)**  **{**    **if(n>=1&&maze\_array[m][n-1]!=1)**  **{**  **return 2;**  **}**  **else**  **if(m>=1&&maze\_array[m-1][n]!=1)**  **{**  **return 3;**  **}**  **else**  **if(n<=N-2&&maze\_array[m][n+1]!=1)**  **{**  **return 0;**  **}**  **else**  **if(m<=M-2&&maze\_array[m+1][n]!=1)**  **{**  **return 1;**  **}**    **}**    **}**      **//把路径节点存入一维数组中！**  **void Maze\_Path::push\_maze\_path\_node(Maze\_Path\_Node ma)**  **{**  **maze\_path[maze\_path\_length]=ma;**  **maze\_path\_length++;**  **}**      **//判断路径是否有回路**  **int Maze\_Path::is\_back()**  **{**  **for(int i=0;i<M;i++)**  **for(int j=0;j<N;j++)**  **{**  **if(maze\_node\_passby\_array[i][j]>1)**  **return 1;**  **}**  **}**    **//简化路径**  **void Maze\_Path::sim\_maze\_path()**  **{**  **for(;is\_back();)**  **{**  **for(int i=0;i<maze\_path\_length;i++)**  **{**  **if(maze\_node\_passby\_array[maze\_path[i].get\_i()][maze\_path[i].get\_j()]==2)**  **{**  **int m=i+1;**  **for(**  **Maze\_Path\_Node curr=maze\_path[m];**  **!(curr.get\_i()==maze\_path[i].get\_i()&&curr.get\_j()==maze\_path[i].get\_j());**  **)**  **{**  **curr=maze\_path[m++];**  **}**  **cout<<endl<<i<<","<<m<<endl;**  **delete\_useless\_path(i,m);**  **}**  **}**  **}**  **}**      **//计算得到路径,更新迷宫局势**  **void Maze\_Path::get\_maze\_path()**  **{**  **//入口结点，作为第一个路径结点，存入路径**  **maze\_array[maze\_entrance.get\_i()][maze\_entrance.get\_j()]=2; //迷宫中入口点，设置为经过**  **maze\_node\_passby\_array[m][n]++; //记录次数的数组**  **push\_maze\_path\_node(Maze\_Path\_Node(m,n,direction,maze\_node\_passby\_array[m][n]));//路径结点，存入**      **//没有走出出口，也没有走出入口**  **for(;!(m==maze\_exit.get\_i()&&n==maze\_exit.get\_j()&&direction==maze\_exit.get\_path\_direction())**  **&&!(m==maze\_entrance.get\_i()&&n==maze\_entrance.get\_j()&&maze\_entrance.get\_path\_direction()==2);)**  **{**  **one\_more\_step();**  **}**  **}**    **//找到下一位置，并且设置为2，重置全局变量direction，寻求下一步路径结点！**  **void Maze\_Path::one\_more\_step()**  **{**    **//向右**  **if(direction==0)**  **{**    **if(m>=1&&maze\_array[m-1][n]!=1)**  **{**  **m=m-1;**  **direction=3;**  **}**  **else**  **if(n<=N-2&&maze\_array[m][n+1]!=1)**  **{**  **n=n+1;**  **direction=0;**  **}**  **else**  **if(m<=M-2&&maze\_array[m+1][n]!=1)**  **{**  **m=m+1;**  **direction=1;**  **}**  **else**  **if(n>=1&&maze\_array[m][n-1]!=1)**  **{**  **n=n-1;**  **direction=2;**  **}**    **}**  **//向下**  **else**  **if(direction==1)**  **{**  **if(n<=N-2&&maze\_array[m][n+1]!=1)**  **{**  **n=n+1;**  **direction=0;**  **}**  **else**  **if(m<=M-2&&maze\_array[m+1][n]!=1)**  **{**  **m=m+1;**  **direction=1;**  **}**  **else**  **if(n>=1&&maze\_array[m][n-1]!=1)**  **{**  **n=n-1;**  **direction=2;**  **}**  **else**  **if(m>=1&&maze\_array[m-1][n]!=1)**  **{**  **m=m-1;**  **direction=3;**  **}**  **}**  **//向左**  **else**  **if(direction==2)**  **{**  **if(m<=M-2&&maze\_array[m+1][n]!=1)**  **{**  **m=m+1;**  **direction=1;**  **}**  **else**  **if(n>=1&&maze\_array[m][n-1]!=1)**  **{**  **n=n-1;**  **direction=2;**  **}**  **else**  **if(m>=1&&maze\_array[m-1][n]!=1)**  **{**  **m=m-1;**  **direction=3;**    **}**  **else**  **if(n<=N-2&&maze\_array[m][n+1]!=1)**  **{**  **n=n+1;**  **direction=0;**    **}**  **}**  **//向上**  **else**  **if(direction==3)**  **{**    **if(n>=1&&maze\_array[m][n-1]!=1)**  **{**  **n=n-1;**  **direction=2;**  **}**  **else**  **if(m>=1&&maze\_array[m-1][n]!=1)**  **{**  **m=m-1;**  **direction=3;**  **}**  **else**  **if(n<=N-2&&maze\_array[m][n+1]!=1)**  **{**  **n=n+1;**  **direction=0;**  **}**  **else**  **if(m<=M-2&&maze\_array[m+1][n]!=1)**  **{**  **m=m+1;**  **direction=1;**  **}**    **}**    **maze\_node\_passby\_array[m][n]++;//路过次数加1**  **maze\_array[m][n]=2;**  **push\_maze\_path\_node(Maze\_Path\_Node(m,n,direction,maze\_node\_passby\_array[m][n]));//添加路径结点**  **}** |

#### 参考2：

|  |
| --- |
| **C++ 用回溯法求解迷宫问题（递归）**  一、定义：  [回溯法](https://so.csdn.net/so/search?q=%E5%9B%9E%E6%BA%AF%E6%B3%95&spm=1001.2101.3001.7020)（探索与回溯法）是一种选优搜索法，又称为试探法，按选优条件向前搜索，以达到目标。      但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就      退回再走的技术为回溯法，而满足回溯[条件](https://baike.baidu.com/item/%E6%9D%A1%E4%BB%B6/1783021)的某个[状态](https://baike.baidu.com/item/%E7%8A%B6%E6%80%81/33204)的点称为“回溯点”。      用回溯法求解问题时常使用[递归方法](https://so.csdn.net/so/search?q=%E9%80%92%E5%BD%92%E6%96%B9%E6%B3%95&spm=1001.2101.3001.7020)进行试探，或使用栈帮助向前试探和回溯。    二、思路：     1.用一个二维数组表示迷宫：               数组中的值如果是1，则表示该位置是墙壁，不能通行；               如果等于0，表示该位置是通路；        用一个标记数组，标记该位置是否被访问过；     2.迷宫前进的方向：要定义一个结构体存储方向              可能有8个方向：东，东南，南，西南，西，西北，北，东北     3.给一个入口和出口的坐标，从入口入手：        循环入口的8个方向，如果位置通，则递归该位置，一直递归下去，直到到达出口或者所有方向的路不通回溯； |
| 三、实现程序：  **/\***  **测试数据：**  **1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**  **0 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1**  **1 1 0 0 0 1 1 0 1 1 1 0 0 1 1 1 1**  **1 0 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1**  **1 1 1 0 1 1 1 1 0 1 1 0 1 1 0 0 1**  **1 1 1 0 1 0 0 1 0 1 1 1 1 1 1 1 1**  **1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 1 1**  **1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 1 1**  **1 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1**  **1 0 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1**  **1 1 1 0 0 0 1 1 0 1 1 0 0 0 0 0 1**  **1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 1**  **1 0 1 0 0 1 1 1 1 1 0 1 1 1 1 0 0**  **1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**    **入口：1 0**  **出口：12 16**  **\*/**  **#include <iostream>**  **using namespace std;**    **const int XLEN = 14, YLEN=17; // 默认迷宫的行数和列数**  **int m, p; // 出口坐标**  **int maze[XLEN+2][YLEN+2]; // 迷宫数组**  **int mark[XLEN+2][YLEN+2]; // 访问标记数组**    **struct Offsets {**  **int a, b; // a和b是x,y方向的偏移**  **// char \*dir; // dir是方向 ISO C++11是不允许字符指针指向（存储）字符串**  **};**    **// 因为我不需要输出方向，所以把方向去掉**  **Offsets direction[8] = {**  **{-1, 0}, // 北：{-1, 0, "N"}**  **{-1, 1}, // 东北：{-1, 1, "NE"}**  **{0, 1}, // 东：{0, 1, "E"}**  **{1, 1}, // 东南：{1, 1, "SE"}**  **{1, 0}, // 南：{1, 0, "S"}**  **{1, -1}, // 西南：{1, -1, "SW"}**  **{0, -1}, // 西：{0, -1, "W"}**  **{-1, -1} // 西北：{-1, -1, "NW"}**  **};**    **// 解决迷宫问题的递归算法**  **bool seekPath(int x, int y);**    **int main(int argc, const char \* argv[]) {**  **int i, j, x, y;**    **// 输入迷宫数据和初始化标记数组**  **cout << "请输入" << XLEN << "行和" << YLEN << "列的迷宫数据:" << endl;**  **for(i = 0; i < XLEN; i++)**  **for(j = 0; j < YLEN; j++) {**  **cin >> maze[i][j];**  **mark[i][j] = 0;**  **}**  **cout << "请输入入口位置坐标：";**  **cin >> x >> y;**  **cout << "请输入出口位置坐标:";**  **cin >> m >> p;**  **mark[x][y] = 1; // 从入口开始**  **// 调用求解迷宫的递归算法**  **if(seekPath(x, y)) // 成功**  **cout << "(" << x << ", " << y << ")" << endl;**  **else**  **cout << "找不到从入口到出口的路径！" << endl;**  **return 0;**  **}**    **// 解决魔攻问题的递归算法**  **bool seekPath(int x, int y) {**  **// 从迷宫某一位置[i][j]开始，寻找出口[m][p]的一条路径,如果找到，则函数返回true**  **int i, g, h; // g, h记录位置信息**  **// char \*dir; // dir记录方向**    **if(x == m && y == p) // m和p是出口坐标**  **return true;**  **// 循环遍历(x, y)的8个方向**  **for(i = 0; i < 8; i++) {**  **g = x + direction[i].a;**  **h = y + direction[i].b;**  **// dir = direction[i].dir;**    **// 找下一位置寻找通向出口的路径**  **if(maze[g][h] == 0 && mark[g][h] == 0) { // 如果通且未被访问过**  **mark[g][h] = 1; // 标记为已访问过**  **if(seekPath(g, h)) { // 递归试探**  **cout << "(" << g << ", " << h << ")，"; // 逆向输出路径坐标**  **return true;**  **}**  **}**  **// 回溯，换一个方向再试探通向出口的路径**  **}**  **return false; // 找不到通向出口的路径**  **}**   |  | | --- | | 测试结果：  https://img-blog.csdnimg.cn/201901091856258.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2NodWFuemhvdXhpYW8=,size_16,color_FFFFFF,t_70 | |

#### 参考3：

|  |
| --- |
| [迷宫问题求解——C++](https://www.cnblogs.com/Solomon-xm/p/9363952.html)  **迷宫问题思路**  根据昨天的博客，有如下几种解决方案   1. 克鲁斯卡尔 ，为避免死循环，需要设定优化路径的次数。 2. Prim，为避免死循环，需要设定优化路径的次数，暂定200次。 3. BFS ， 实现简单，无死循环。 4. DFS , 实现简单，无死循环，复杂度较低。 5. 动态规划，实时根据权重调整方向，目前看来最合适的解决方案。需要以数据证明。   综上，本次优先选择BFS，首先不存在死循环的风险，其次算法复杂度较低，容易理解且实现。适合初步练手。  **一. 思路及程序算法**  首先建立迷宫，将其看作点位矩阵，先把墙堆起来，剩下的就是路。  而每个点都有四个方向（上、下、左、有），每个方向上都对应一个点。在这四个点中，有一个点是当前站立点的“上一个点”，另外三个点是当前站立点的“下一个点”，如图1.1所示，红色框为当前站立点，蓝色框为“上一个点”，黄色框为三个“下一个点”。  https://images2018.cnblogs.com/blog/910321/201807/910321-20180725092249747-351910127.png 图1.1站立点的四个方向  当前站立点有三种状态，分别是“终点”、“通路”和“死路”。如果当前站立点是“终点”，则停止搜索；如果当前站立点是“通路”，则向下继续走；如果当前站立点是“死路”，则要向回走，然后搜索其它路径。其流程如图1.2所示。  https://images2018.cnblogs.com/blog/910321/201807/910321-20180725092532050-833439003.png 图1.2判断迷宫路径流程图 |
| 二. 实现实现效果如图2.1所示： https://images2018.cnblogs.com/blog/910321/201807/910321-20180725092710269-913548441.png 图2.1 BFS迷宫求解实现图 |
| BFS迷宫求解实现代码如下：  #include <bits/stdc++.h>  using namespace std;  #define coordi(x,y) ( m\*(x-1)+y )  const int maxn = 30;  const int dx[] = {0,0,1,-1};  const int dy[] = {1,-1,0,0};  int mp[maxn+10][maxn+10];  int nxtx[maxn+10][maxn+10];  int nxty[maxn+10][maxn+10];  bool vis[maxn+10][maxn+10];  int fa[(maxn+10)\*(maxn+10)];  int n , m;  int stx , sty , edx , edy;  inline int check( int x , int y )  {  return 1<=x && x<=n && 1<=y && y<=m;  }  inline void print\_map()  {  puts("\n==============================================");  for( int i = 1; i <= n; i++ )  {  for( int j = 1; j <= m; j++ )  printf("%c",mp[i][j]);  putchar('\n');  }  puts("==============================================");  }  // 并查集  int getfa( int x )  {  return x==fa[x]?x:fa[x] = getfa(fa[x]);  }  void unio( int a , int b )  {  int fx = getfa(a) , fy = getfa(b);  if ( fx != fy ) fa[fx] = fy;  }  // 并查集  void connect()  {  int t = n\*m/3\*2;  for( int i = 1; i <= n\*m; i++ ) fa[i] = i;  int fs = getfa(coordi(stx,sty)) , ft = getfa(coordi(edx,edy));  while( fs != ft || t > 0 )  {  t--;  int px = rand()%n+1 , py = rand()%m+1;  if ( mp[px][py] == 'X' )  {  mp[px][py] = '.';  for( int k = 0 ; k< 4; k++ )  {  int xx = px + dx[k] , yy = py + dy[k];  if ( check(xx,yy) && mp[xx][yy] != 'X' ) unio( coordi(px,py) , coordi(xx,yy) );  }  }  fs = getfa(coordi(stx,sty)) , ft = getfa(coordi(edx,edy));  }  }  void init()  {  srand(time(0));  n = rand()%maxn+10;  m = rand()%maxn+10;  cout<<"map size : "<<n<<" \* "<<m<<endl;  for( int i = 1; i <= n; i++ )  for( int j = 1; j <= m; j++ ) mp[i][j] = 'X';  stx = rand()%n+1 , sty = rand()%m+1;  edx = rand()%n+1 , edy = rand()%m+1;  while( abs(edx-stx) + abs(edy-sty) <= 1 ) edx = rand()%n+1 , edy = rand()%m+1;  mp[stx][sty] = 'S' , mp[edx][edy] = 'T';  cout<<"start:("<<stx<<","<<sty<<")"<<endl;  cout<<"end:("<<edx<<","<<edy<<")"<<endl;  connect();  print\_map();  }  void print\_path() // path = '\*' st = S , ed = T , road = . , wall = X  {  int x = edx , y = edy;  while( !( x == stx && y == sty ) )  {  mp[x][y] = '\*';  int tx = nxtx[x][y];  y = nxty[x][y];  x = tx;  }  mp[edx][edy] = 'T';  print\_map();  }  void bfs()  {  queue< pair<int,int> > q;  q.push( make\_pair(stx,sty) );  memset(vis,0,sizeof(vis));  vis[stx][sty] = true;  while( !q.empty() )  {  pair<int,int> temp = q.front();  q.pop();  if ( temp.first == edx && temp.second == edy )  {  print\_path();  return;  }  for( int k = 0; k < 4; k++ )  {  int xx = temp.first + dx[k] , yy = temp.second + dy[k];  if ( !check(xx,yy) || vis[xx][yy] || mp[xx][yy] == 'X' ) continue;  vis[xx][yy] = 1 , nxtx[xx][yy] = temp.first , nxty[xx][yy] = temp.second;  q.push( make\_pair(xx,yy) );  }  }  }  int main()  {  init();  bfs();  return 0;  } |

#### 参考4：

|  |
| --- |
| 经典迷宫问题（C++）简单DFS算法c++ \****给出一个起点S 输入地图 ‘.’ 代表可以走 ‘*’代表墙壁不能走‘T’代表终点** **question**：**判断是否能够能从到达终点** （不要求最短路径） **thinking and methods** **：单独考虑一个点，从上下左右四个方向进行探索，如果满足条件（不是墙壁&&未超界&&未被探索过），向这个方向探索（递归），直到满足条件或者无路可走return 进行回溯 好好理解dfs（x，y）回溯之后的下一步：vis[tx][ty]=0 “还原现场”；** https://img-blog.csdnimg.cn/20210112150724737.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L20wXzQ4MzAwNjg0,size_16,color_FFFFFF,t_70 **(4,3)为单点讲解**  代码： |
| #include <stdio.h>  #include <iostream>  using namespace std;  char Map[50][50];  int n,m,p,q,vis[50][50],flag=0;  const int N=1000;  int dir[4][2]={  {  -1,0},{  1,0},{  0,-1},{  0,1}};//上下左右走位  void dfs(int x,int y)//经典DFS模板↓↓↓  {    if(Map[x][y]=='S')//走到终点  {    flag=1;  return;  }  for(int i=0;i<4;i++)  {    int tx=x+dir[i][0];  int ty=y+dir[i][1];  if(tx>=0&&tx<=m&&ty>=0&&ty<=n&&vis[tx][ty]==0&&Map[tx][ty]!='\*')//下一步未出界且下一步位置未被访问过(DFS算法的vis[][])  {    vis[tx][ty]=1;  dfs(tx,ty);  vis[tx][ty]=0;//回溯  }  }  }  int main()  {    while(cin>>n,n)  {    cin>>m>>n;  memset(vis,0,sizeof(vis));  for(int i=0;i<m;i++)  for(int j=0;j<n;j++)  {    cin>>Map[i][j];  if(Map[i][j]=='T') p=i,q=j;  }  dfs(p,q);  if(flag==1) cout<<"yes"<<endl;  else cout<<"No"<<endl;  }  } |

# 14.C++ 基于 OpenCV 实现实时监控和运动检测记录

#### 参考1：<https://blog.csdn.net/oxuzhenyi/article/details/54982554>

#### 参考2：

GitHub：<https://github.com/Forec/monitor-recorder>

已经下载的代码：<monitor-recorder-win-linux>

# 15.C++ 实现简易 Docker 容器

#### 参考1：<https://blog.csdn.net/BJUT_bluecat/article/details/92443153>

|  |
| --- |
| **c++ 实现docker容器(namespace隔离)**  参考  <https://www.cnblogs.com/luosongchao/p/3680312.html>  <https://blog.csdn.net/yangkuanqaz85988/article/details/52403726> 进程栈  <https://blog.csdn.net/hty46565/article/details/79934205> namespace  参考实验楼  利用namespace技术模拟[docker](https://so.csdn.net/so/search?q=docker&spm=1001.2101.3001.7020)容器。  自己实现  **//dockr.hpp**  **//用到的头文件**  **#include <sys/wait.h> // waitpid**  **#include <sys/mount.h> // mount**  **#include <fcntl.h> // open**  **#include <unistd.h> // execv, sethostname, chroot, fchdir**  **#include <sched.h> // clone**  **#include <iostream>**  **#include <string>**  **// C 标准库**  **#include <cstring>**  **#define STACK\_SIZE (1024 \* 1024) // 定义子进程空间大小 1M**  **namespace bluecat**  **{**  **typedef struct docker\_info//一些容器信息**  **{**  **std::string hostname;//主机名字**  **std::string root\_dir;//根目录名字**  **}docker\_info;**  **typedef int process\_pid;//增强**  **class bluecat\_docker**  **{**  **private:**  **docker\_info info;//容器信息**  **char child\_stack[STACK\_SIZE];//子进程栈空间大小**  **//设置容器主机名字**  **void set\_hostname()**  **{**  **sethostname(this->info.hostname.c\_str(), this->info.hostname.size());**  **//std::cout<<"设置容器主机名字完毕"<<std::endl;**  **}**  **//设置容器根目录**  **void set\_rootdir()**  **{**  **// chdir 系统调用, 切换到某个目录下**  **chdir(this->info.root\_dir.c\_str());**  **// chroot 系统调用, 设置根目录, 因为刚才已经切换过当前目录**  **// // 直接使用当前目录作为根目录就可以了**  **chroot(".");**  **}**  **// 挂载 proc 文件系统**  **void set\_procsys()**  **{**  **mount("none", "/proc", "proc", 0, nullptr);**  **mount("none", "/sys", "sysfs", 0, nullptr);**    **}**  **void start\_bash()**  **{**  **// 将 C++ std::string 安全的转换为 C 风格的字符串 char \***  **std::string bash="/bin/bash";**  **char \*c\_bash=new char[bash.size()+1];//多一个字节存储\0**    **strcpy(c\_bash, bash.c\_str());**  **char\* const child\_args[] = { c\_bash, NULL };**  **execv(child\_args[0], child\_args); // 在子进程中执行 /bin/bash**  **delete []c\_bash;**  **c\_bash=nullptr;**  **}**  **public:**  **bluecat\_docker(docker\_info &d\_info):info(d\_info)//构造函数**  **{**    **}**  **static int set\_info(void \*args)**  **{**  **auto bc\_docker=static\_cast<bluecat\_docker\*>(args);**  **bc\_docker->set\_hostname();**  **bc\_docker->set\_rootdir();**  **bc\_docker->set\_procsys();**  **bc\_docker->start\_bash();**    **return 1;**  **}**  **void start()**  **{**  **std::cout<<"welcome to bluecat's docker"<<std::endl;**  **process\_pid child\_process=clone(set\_info,child\_stack+STACK\_SIZE,//每个容器是一个进程**  **CLONE\_NEWUTS| //uts 域名隔离**  **CLONE\_NEWNS| //Mount namespace**  **CLONE\_NEWIPC|//进程通信**  **CLONE\_NEWPID|**  **SIGCHLD, // 子进程退出时会发出信号给父进程 避免僵尸进程**  **this);//把this传给静态函数**  **waitpid(child\_process, nullptr, 0); // 等待子进程的退出 避免僵尸进程**  **printf("已经退出\n");**  **}**  **};**  **};**    **#include "docker.hpp"**  **#include <iostream>**    **int main(int argc, char\*\* argv)**  **{**  **std::cout << "start docker..." << std::endl;**  **bluecat::docker\_info doc\_info;**  **doc\_info.hostname = "蓝猫的docker";**  **doc\_info.root\_dir="./bluecat";**  **bluecat::bluecat\_docker mydocker(doc\_info);**  **mydocker.start(); // 启动容器**  **std::cout << "stop docker..." << std::endl;**  **return 0;**  **}**    **bluecat@bluecat:~/docker\_cpp/my\_docker2$ g++ main.cpp -o docker -std=c++11**  **bluecat@bluecat:~/docker\_cpp/my\_docker2$ sudo ./docker**  **[sudo] password for bluecat:**  **start docker...**  **welcome to bluecat's docker**  **groups: cannot find name for group ID 125**  **root@蓝猫的docker:/# ps**  **PID TTY TIME CMD**  **1 ? 00:00:00 bash**  **4 ? 00:00:00 ps**  **root@蓝猫的docker:/# ls**  **bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var**  **root@蓝猫的docker:/# exit**  **exit**  **已经退出**  **stop docker...**  **bluecat@bluecat:~/docker\_cpp/my\_docker2$ ps**  **PID TTY TIME CMD**  **3537 pts/4 00:00:00 bash**  **4912 pts/4 00:00:00 ps**  **bluecat@bluecat:~/docker\_cpp/my\_docker2$** |

# 16.C++ 开发 Web 服务框架

#### 参考1：<https://github.com/artyom-beilis/cppcms>

#### 参考2：<https://segmentfault.com/a/1190000040889524>

#### 参考3：

|  |
| --- |
| C++ 开发 Web 服务[框架](https://so.csdn.net/so/search?q=%E6%A1%86%E6%9E%B6&spm=1001.2101.3001.7020) - 基础知识：C++11 与 Boost Asio一、概述项目介绍 服务器开发中 Web 服务是一个基本的代码单元，将服务端的请求和响应部分的逻辑抽象出来形成框架，能够做到最高级别的框架级代码复用。本次项目将综合使用 C++11 及 [Boost](https://so.csdn.net/so/search?q=Boost&spm=1001.2101.3001.7020) 中的 Asio 实现 HTTP 和 HTTPS 的服务器框架。 项目涉及的知识点  * C++基本知识   + 面向对象   + 模板   + 命名空间   + 常用 IO 库 * C++11 相关   + lambda expression   + std::shared\_ptr   + std::make\_shared   + std::unordered\_map   + std::regex   + std::smatch   + std::regex\_match   + std::function   + std::thread * Boost Asio 相关   + boost::asio::io\_service   + boost::asio::ip::tcp::socket   + boost::asio::ip::tcp::v4()   + boost::asio::ip::tcp::endpoint   + boost::asio::ip::tcp::acceptor   + boost::asio::streambuf   + boost::asio::async\_read   + boost::asio::async\_read\_until   + boost::asio::async\_write   + boost::asio::transfer\_exactly   + boost::asio::ssl::stream   + boost::asio::ssl::stream\_base::server   + boost::asio::ssl::context   + boost::asio::ssl::context::sslv23   + boost::asio::ssl::context::pem   + boost::system::error\_code  编译环境提示 在 g++ 4.9 之前，regex 库并不支持 ECMAScript 的正则语法，换句话说，在 g++4.9 之前，g++ 对 C++11 标准库的支持并不完善，为保证本次项目的顺利进行，请确保将 g++ 版本升级至 4.9 以上。  **// 下面的这段代码可以测试你的编译器对正则表达式的支持情况**  **#include <iostream>**  **#include <regex>**    **int main()**  **{**  **std::regex r1("S");**  **printf("S works.\n");**  **std::regex r2(".");**  **printf(". works.\n");**  **std::regex r3(".+");**  **printf(".+ works.\n");**  **std::regex r4("[0-9]");**  **printf("[0-9] works.\n");**  **return 0;**  **}**  如果你的运行结果遇到了下图所示的错误，说明你确实需要升级你的 g++ 了：  使用 g++ -v 可以查看到当前编译器版本：  如果你最后一行中的 gcc version 显示的是 4.8.x，那么你需要手动将编译器版本升级至 4.9 以上，方法如下：  **# 安装 add-apt-repository 工具**  **sudo apt-get install software-properties-common**  **# 增加源**  **sudo add-apt-repository ppa:ubuntu-toolchain-r/test**  **# 更新源**  **sudo apt-get update**  **# 更新安装**  **sudo apt-get upgrade**  **# 安装 gcc/g++ 4.9**  **sudo apt-get install gcc-4.9 g++-4.9**  **# 更新链接**  **sudo updatedb**  **sudo ldconfig**  **sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 48 \**  **--slave /usr/bin/g++ g++ /usr/bin/g++-4.8 \**  **--slave /usr/bin/gcc-ar gcc-ar /usr/bin/gcc-ar-4.8 \**  **--slave /usr/bin/gcc-nm gcc-nm /usr/bin/gcc-nm-4.8 \**  **--slave /usr/bin/gcc-ranlib gcc-ranlib /usr/bin/gcc-ranlib-4.8**  **sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.9 49 \**  **--slave /usr/bin/g++ g++ /usr/bin/g++-4.9 \**  **--slave /usr/bin/gcc-ar gcc-ar /usr/bin/gcc-ar-4.9 \**  **--slave /usr/bin/gcc-nm gcc-nm /usr/bin/gcc-nm-4.9 \**  **--slave /usr/bin/gcc-ranlib gcc-ranlib /usr/bin/gcc-ranlib-4.9**  此外，本次项目依赖了 Boost 和 OpenSSL 这两个库，不过好在实验楼的环境已经提供了这两个非常基本的库，你不需要再操心他们的安装了。 一、C++ 基础 面向对象和模板是 C++进阶知识的基础，这里不做过多介绍，本次项目我们将开发一个 Web 框架，我们在这里先回顾一下命名空间、和 sstream 字符串 IO 流的相关知识。如果对这部分比较熟悉，可以直接跳过本小节。 命名空间 在开发库时，库通常会有定义大量的全局名称，这时候当我们使用的库越来越多时，就不可避免的发生名称冲突的情况，这也就是我们常说的命名空间污染。  在命名空间诞生以前，通常使用的办法就是把一个函数、类、甚至变量名等名字取得足够长，在每一个名字的前面都增加相应的前缀，例如，当我们只想要定义一个 port 的变量时候：  **// 原本的样子**  **int port;**  **// 实际的样子**  **int shiyanlou\_web\_server\_port;**  命名空间的定义非常简单，通过关键字 namespace 加上命名空间的名字，再使用花括号包裹需要的定义和声明即可完成相关的定义，例如：  **namespace shiyanlou\_web\_server {**  **int port = 0;**  **}**  这时，这个 port 就被限制在了命名空间 shiyanlou\_web\_server 当中，如果不通过命名空间的指定，就不会被访问到。  参考下面的例子：  **//**  **// main.cpp**  **//**  **#include <iostream>**  **#include "web.hpp"**  **#include "web2.hpp"**  **int main() {**  **std::cout << "hello world!" << std::endl;**  **std::cout << "shiyanlou\_web\_server, port=" << shiyanlou\_web\_server::port << std::endl;**  **std::cout << "shiyanlou\_web2\_server, port=" << shiyanlou\_web2\_server::port << std::endl;**  **return 0;**  **}**  **//**  **// web.hpp**  **//**  **namespace shiyanlou\_web\_server{**  **int port = 0;**  **}**  **//**  **// web2.hpp**  **//**  **namespace shiyanlou\_web2\_server{**  **int port = 2;**  **}**  最后的输出结果为：  hello world!  shiyanlou\_web\_server, port=0  shiyanlou\_web2\_server, port=2 常用 IO 库 我们常说的 C++ IO 库一般指 iostream, fstream, sstream。   * iostream 包含了 istream(从流读)/ostream(从流写)/iostream(读写流) * fstream 包含了 ifstream(从文件读)/ofstream(condition 文件写)/fstream(读写文件) * sstream 包含了 istringstream(从 string 读)/ostringstream(向 string 写)/stringstream(读写 string)   其实标准库还有宽字符版本，但我们这里不讨论，有兴趣的话可以参考参考链接。  iostream 和 fstream 是两个比较常用的IO 库，我们这里不再回顾，这里简单回顾一下 sstream。  如果你熟悉 C 语言，就知道将 int 转换为 string 类型其实是一件很麻烦的事情，虽然标准库中提供了 itoa() 这种函数，但是依然需要对转换后的 C 风格字符串(char \*)通过 std::string 的构造函数构造为 std::string。 如果使用流操作，那么这将变得异常的简单：  **#include <string>**  **#include <sstream>**  **#include <iostrea>**    **int main() {**  **// std::stringstream 支持读写**  **std::stringstream stream;**  **std::string result;**  **int number = 12345;**  **stream << number; // 将 number 输入到 stream**  **stream >> results; // 从 stream 读取到 result**  **std::cout < result << std::endl; // 将输出为字符串"12345"**  **}**  如果希望让sstream 和 C 风格的字符串打交道，同样也可以：  **#include <sstream>**  **#include <iostream>**    **int main()**  **{**  **std::stringstream stream;**  **char result[6];**  **stream << 12345;**  **stream >> result;**  **std::cout << result << std::endl;**  **}**  需要注意的一点就是，在进行多次IO 操作时，如果希望结果彼此不影响，需要对 stream 对象进行一次 clear() 操作：  stream.clear() 二、C++11 相关 C++11 几乎重新定义了 C++ 的一切，C++11 的出现伴随着大量的有用的新特性和标准库，这些特性和标准使得 C++ 变得更加现代，甚至在编码范式上都与传统 C++ 有着本质上的差异，本节我们将回顾一下这些特性：   * lambda expression * std::shared\_ptr * std::make\_shared * std::unordered\_map * std::regex * std::smatch * std::regex\_match * std::function * std::thread   如果对这些特性比较熟悉，可以直接跳过本节。 lambda 表达式 Lambda 表达式是 C++11中最重要的新特性之一，而 Lambda 表达式，实际上就是提供了一个类似匿名函数的特性，而匿名函数则是在需要一个函数，但是又不想费力去命名一个函数的情况下去使用的。这样的场景其实有很多很多，所以匿名函数几乎是现代编程语言的标配。  Lambda 表达式的基本语法如下：  **[捕获列表](参数列表) mutable(可选) 异常属性 -> 返回类型 {**  **// 函数体**  **}**  上面的语法规则除了 [捕获列表] 内的东西外，其他部分都很好理解，只是一般函数的函数名被略去，返回值使用了一个 ->的形式进行。  所谓捕获列表，其实可以理解为参数的一种类型，lambda 表达式内部函数体在默认情况下是不能够使用函数体外部的变量的，这时候捕获列表可以起到传递外部数据的作用。根据传递的行为，捕获列表也分为以下几种：  **1. 值捕获**  与参数传值类似，值捕获的前期是变量可以拷贝，不同之处则在于，被捕获的变量在 lambda 表达式被创建时拷贝，而非调用时才拷贝：  **void learn\_lambda\_func\_1() {**  **int value\_1 = 1;**  **auto copy\_value\_1 = [value\_1] {**  **return value\_1;**  **};**  **value\_1 = 100;**  **auto stored\_value\_1 = copy\_value\_1();**  **// 这时, stored\_value\_1 == 1, 而 value\_1 == 100.**  **// 因为 copy\_value\_1 在创建时就保存了一份 value\_1 的拷贝**  **}**  **2. 引用捕获**  与引用传参类似，引用捕获保存的是引用，值会发生变化。  **void learn\_lambda\_func\_2() {**  **int value\_2 = 1;**  **auto copy\_value\_2 = [&value\_2] {**  **return value\_2;**  **};**  **value\_2 = 100;**  **auto stored\_value\_2 = copy\_value\_2();**  **// 这时, stored\_value\_2 == 100, value\_1 == 100.**  **// 因为 copy\_value\_2 保存的是引用**  **}**  **3. 隐式捕获**  手动书写捕获列表有时候是非常复杂的，这种机械性的工作可以交给编译器来处理，这时候可以在捕获列表中写一个 & 或= 向编译器声明采用 引用捕获或者值捕获.  总结一下，捕获提供了lambda 表达式对外部值进行使用的功能，捕获列表的最常用的四种形式可以是：   * [] 空捕获列表 * [name1, name2, ...] 捕获一系列变量 * [&] 引用捕获, 让编译器自行推导捕获列表 * [=] 值捕获, 让编译器执行推导应用列表  std::shared\_ptr, std::make\_shared C++11 在内存管理上同样做了很多改进，std::make\_shared 就是其中之一。它是和 std::shared\_ptr 共同出现的，std::shared\_ptr 是一种智能指针，它能够记录多少个 shared\_ptr 共同指向一个对象（熟悉 Objective-C 的可能知道，这种特性叫做引用计数），能够消除显示的调用 delete，当引用计数变为0的时候就会将对象自动删除。  但还不够，因为使用 std::shared\_ptr 仍然需要使用 new来调用，这使得代码出现了某种程度上的不对称。因此就需要另一种手段(工厂模式)来解决这个问题。  std::make\_shared 就能够用来消除显示的使用 new，所以std::make\_shared 会分配创建传入参数中的对象，并返回这个对象类型的std::shared\_ptr指针。例如：  **#include <iostream>**  **#include <memory>**    **void foo(std::shared\_ptr<int> i)**  **{**  **(\*i)++;**  **}**  **int main()**  **{**  **// 构造了一个 std::shared\_ptr**  **auto pointer = std::make\_shared<int>(10);**  **foo(pointer);**  **std::cout << \*pointer << std::endl;**  **}** 无序容器 std::unordered\_map 在传统的 C++中，我们已经熟知了 std::map 关联容器，std::map 容器在插入元素时，会根据 < 操作符比较元素大小并判断元素是否相同，并选择合适的位置插入到容器中。当对这个容器中的元素进行遍历时，输出结果会按照 < 操作符的顺序来逐个遍历。  而 C++11 终于推出了无序容器。 std::unordered\_map 就是无序容器其中之一，这个容器会计算元素的 Hash 值，并根据 Hash 值来判断元素是否相同。由于无序容器没有定义元素之间的顺序，仅靠 Hash 值来判断元素是否已经存在于容器中，所以遍历 std::unordered\_map 时，结果是无序的。  来看一个例子：  **#include <iostream>**  **#include <string>**  **#include <unordered\_map>**  **#include <map>**    **int main() {**  **// 两组结构按同样的顺序初始化**  **std::unordered\_map<int, std::string> u = {**  **{1, "1"},**  **{3, "3"},**  **{2, "2"}**  **};**  **std::map<int, std::string> v = {**  **{1, "1"},**  **{3, "3"},**  **{2, "2"}**  **};**    **// 分别对两组结构进行遍历**  **std::cout << "std::unordered\_map" << std::endl;**  **for( const auto & n : u)**  **std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";**    **std::cout << std::endl;**  **std::cout << "std::map" << std::endl;**  **for( const auto & n : v)**  **std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";**  **}**  最终的输出结果为：  **std::unordered\_map**  **Key:[2] Value:[2]**  **Key:[3] Value:[3]**  **Key:[1] Value:[1]**    **std::map**  **Key:[1] Value:[1]**  **Key:[2] Value:[2]**  **Key:[3] Value:[3]**  可以看到 std::map 的遍历结果是有序的，而 std::unordered\_map 的遍历结果是无序的。  事实上，std::unordered\_map 在单个元素访问时，总是能够获得更高的性能。 std::regex/std::regex\_match/std::smatch 正则表达式是一个独立于 C++ 语言本身的另一个很大的话题，我们这里不详细讨论它的行为。  作为学习 std::regex 的一些介绍性内容，我们这里说明一下接下来会用到的一些正则表达式：   * [a-z]+\.txt: 在这个正则表达式中, [a-z] 表示匹配一个小写字母, + 可以使前面的表达式匹配多次，因此 [a-z]+ 能够匹配一个小写字母组成的字符串。在正则表达式中一个 . 表示匹配任意字符，而 \. 则表示匹配字符 .，最后的 txt 表示严格匹配 txt 则三个字母。因此这个正则表达式的所要匹配的内容就是由纯小写字母组成的文本文件。   std::regex\_match 用于匹配字符串和正则表达式，有很多不同的重载形式。最简单的一个形式就是传入std::string 以及一个 std::regex 进行匹配，当匹配成功时，会返回 true，否则返回 false。例如：  **#include <iostream>**  **#include <string>**  **#include <regex>**    **int main() {**  **std::string fnames[] = {"foo.txt", "bar.txt", "test", "a0.txt", "AAA.txt"};**  **// 在 C++ 中 `\` 会被作为字符串内的转义符，为使 `\.` 作为正则表达式传递进去生效，需要对 `\` 进行二次转义，从而有 `\\.`**  **std::regex txt\_regex("[a-z]+\\.txt");**  **for (const auto &fname: fnames)**  **std::cout << fname << ": " << std::regex\_match(fname, txt\_regex) << std::endl;**  **}**  另一种常用的形式就是依次传入 std::string/std::smatch/std::regex 三个参数，其中 std::smatch 的本质其实是 std::match\_results，在标准库中， std::smatch 被定义为了std::match\_results<std::string::const\_iterator>，也就是一个子串迭代器类型的 match\_results。使用 std::smatch可以方便的对匹配的结果进行获取，例如：  **std::regex base\_regex("([a-z]+)\\.txt");**  **std::smatch base\_match;**  **for(const auto &fname: fnames) {**  **if (std::regex\_match(fname, base\_match, base\_regex)) {**  **// sub\_match 的第一个元素匹配整个字符串**  **// sub\_match 的第二个元素匹配了第一个括号表达式**  **if (base\_match.size() == 2) {**  **std::string base = base\_match[1].str();**  **std::cout << "sub-match[0]: " << base\_match[0].str() << std::endl;**  **std::cout << fname << " sub-match[1]: " << base << std::endl;**  **}**  **}**  **}**  以上两个代码段的输出结果为：  **foo.txt: 1**  **bar.txt: 1**  **test: 0**  **a0.txt: 0**  **AAA.txt: 0**  **sub-match[0]: foo.txt**  **foo.txt sub-match[1]: foo**  **sub-match[0]: bar.txt**  **bar.txt sub-match[1]: bar** std::function std::function 是一种通用、多态的函数封装，它的实例可以对任何可以调用的目标实体进行存储、复制和调用操作，它也是对 C++中现有的可调用实体的一种类型安全的包裹（相对来说，函数指针的调用不是类型安全的），简而言之，std::function 就是函数的容器。  在前面的 Lambda 表达式中，我们已经介绍过使用 auto 关键字来接受一个 lambda 表达式。但有时候我们可能希望明确的指明这个 lambda 表达式的类型，这时就可以使用 std::function 来进行书写，例如：  **#include <functional>**  **#include <iostream>**    **int foo(int para) {**  **return para;**  **}**    **int main() {**  **// std::function 包装了一个返回值为 int, 参数为 int 的函数**  **std::function<int(int)> func = foo;**  **std::cout << func(10) << std::endl;**  **}** std::thread std::thread 用于创建一个执行的线程实例，所以它是一切并发编程的基础，使用时需要包含<thread>头文件，它提供了很多基本的线程操作，例如get\_id()来获取所创建线程的线程 ID，例如使用 join() 来等待线程等等，例如：  **#include <iostream>**  **#include <thread>**  **void foo() {**  **std::cout << "hello world" << std::endl;**  **}**  **int main() {**  **std::thread t(foo);**  **t.join();**  **return 0;**  **}** 三、Boost Asio 相关 Boost 是一个 C++的可移植库，是对标准库的后备扩展，也是 C++标准化进程的开发引擎之一。Boost 库是由 C++标准委员会的成员发起的，里面发展的内容很有可能会成为 C++标准库的内容之一。因此 Boost 也是 C++社区中影响力最大的 『准』标准库。  Boost Asio，就是 Boost 库中的一个部分，Asio 的全称为 Asynchronous input and output （异步输入输出）的缩写。顾名思义，结合 Boost 的特点，Asio 提供了一套平台无关的异步数据处理能力（当然它也支持同步数据处理）。  本节我们将熟悉一下下面这些知识点的用法，如果你对这些内容比较熟悉，可以直接跳过本节。   * boost::asio::io\_service * boost::asio::ip::tcp::socket * boost::asio::ip::tcp::v4() * boost::asio::ip::tcp::endpoint * boost::asio::ip::tcp::acceptor * boost::asio::streambuf * boost::asio::async\_read * boost::asio::async\_read\_until * boost::asio::async\_write * boost::asio::transfer\_exactly * boost::asio::ssl::stream * boost::asio::ssl::stream\_base::server * boost::asio::ssl::context * boost::asio::ssl::context::sslv23 * boost::asio::ssl::context::pem * boost::system::error\_code   使用 Asio 只需要引入一个头文件即可：  #include <boost/asio.hpp>  对于所有使用 Asio 的程序，都必须要包含至少一个 io\_service 对象。对于 Asio 这个 Boost 库而言，它抽象了诸如网络、串口通信等等这些概念，并将其统一规为 IO 操作，所以 io\_service 这个类提供了访问 I/O 的功能。因此，使用 Asio 时，必须定义：  boost::asio::io\_service io; HTTP 连接 既然网络的相关概念已经被抽象为 IO，我们就只需要关心从这个 IO 流中获取消息，因此，我们本质上还是在进行 IO 操作，只不过这个操作需要具备一些基本的网络概念。  我们知道，HTTP 和 HTTPS 的底层实际上是使用的 TCP 可靠连接，通过 Socket 技术进行通信，而一个 Socket 由 IP 地址及端口构成。无例外地，Asio 同样也需要建立一个和 socket 有关的对象，那就是 boost::asio::ip::tcp::socket。可想而知，Socket 既然是网络通信的基础，那么自然的我们要进行的 IO 操作也就必须在这里完成，因此，我们定义的 boost::asio::ip::tcp::socket 对象，必须由 io\_service 来进行构造，即：  boost::asio::ip::tcp::socket socket(io);  有了 socket 对象是不够的。在网络通信中，网络 IO 就入口串口一样，是以流的方式进行的。所以这个 socket 对象只能用来做我们日后进行 IO 操作时的一个必要属性。  不难看出，一个普通的 boost::asio::ip::tcp::socket 对象，实际上就是一个 HTTP 的 Socket 连接，因此我们在日后进行代码编写时，甚至于可以使用 typedef 将这个类型直接定义为 HTTP:  typedef boost::asio::ip::tcp::socket HTTP;  然而，作为服务端，我们可能构建很多很多的连接从而响应并发，所以当我们需要建立连接时候，就需要使用一个叫做 acceptor 的对象。  而 boost::asio::ip::tcp::acceptor 从名字上就可以看出，这个对象应该被用于建立连接。在 Boost 中，我们需要初始化一个 acceptor 对象时，必须提供一个 io\_service 对象和一个 endpoint 对象。  那么 endpoint 又是什么？事实上，socket 是一个端到端的连接，所谓 endpoint 就是 socket 位于服务端的一个端点，我们知道，socket 是由 IP 地址和端口号组成的，那么当我们需要为其建立一个 IPv4 的网络，首先可以建立一个 boost::asio::ip::tcp::endpoint 对象：  **unsigned short port = 8080;**  **boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::tcp::v4(), port);**  其中 boost::asio::ip::tcp::v4() 用于初始化一个 IPv4 的网络。最后在使用这个 endpoint 对象来初始化 acceptor:  boost::asio::ip::tcp::acceptor acceptor(io, endpoint);  至此，我们讨论了如何使用 Asio 建立一个普通的网络操作对象acceptor，以及在进行普通 HTTP 网络操作时需要的 socket对象。 HTTPS 连接 讨论完了 Asio 里 HTTP 连接，我们再来看看 Asio 中的 HTTPS 是如何建立连接的。Asio 是一个开源的库，所以它也不可避免的在处理不擅长的逻辑时需要添加对别的框架的依赖。Asio 的 HTTPS 相关的 SSL 操作，就依赖了 OpenSSL 库。  要使用 SSL 相关的操作，还需要额外引入一个头文件：  #include <boost/asio/ssl.hpp>  我们在上一小节里讨论了 boost::asio::ip::tcp::socket 产生的 Socket 对象实际上就是普通的 HTTP 对象。对于 HTTPS 而言，实际上就是对这个 socket 所产生的通道进行一个一层封装和加密。在 Boost Asio 中，加密 socket 的方式就是使用 boost::asio::ssl::stream，并将 boost::asio::ip::tcp::socket 作为模板参数传入给这个对象，即：  typedef boost::asio::ssl::stream<boost::asio::ip::tcp::socket> HTTPS;  而当我们要构造一个 HTTPS 的 socket 对象时，Boost Asio 要求必须为这个 socket 建立一个 boost::asio::ssl::context对象。而一个 context 可以有很多不同的类型，最常用的，就是boost::asio::ssl::context::sslv23。构造好了 context对象之后这还不够，因为一个 https 的服务器需要提供证书文件和秘钥文件，所以还需要使用 use\_certificate\_chain\_file() 和 use\_private\_key\_file() 这两个方法来进行进一步的配置：  **context.use\_certificate\_chain\_file(cert\_file);**  **context.use\_private\_key\_file(private\_key\_file, boost::asio::ssl::context::pem);**  其中的 boost::asio::ssl::context::pem 是指定的证书类型。因此相较于 HTTP 而言，HTTPS 的建立其实就是增加了对证书的配置、和 socket 加密的环节，对比如下：  **// http**  **boost::asio::ip::tcp::socket http\_socket(io);**  **// https**  **boost::asio::ssl::context context(boost::asio::ssl::context::sslv23);**  **context.use\_certificate\_chain\_file(cert\_file);**  **context.use\_private\_key\_file(private\_key\_file, boost::asio::ssl::context::pem);**  **boost::asio::ssl::stream<boost::asio::ip::tcp::socket> https\_socket(io, context);** IO 操作 上面我们讨论了如何建立连接，现在我们再来看看如何进行 IO 操作。  当我们有了 socket 对象之后，就可以从里面读取网络流数据了。读取数据时，我们需要定义一个流缓冲 boost::asio::streambuf 对象，用于逐行读取 socket 中的数据：  boost::asio::streambuf read\_buffer;  另外，很多网络协议其实都是基于行实现的，也就是说这些协议元素是由 \r\n 符号进行界定，HTTP 也不例外，所以在 Boost Asio 中，读取使用分隔符的协议，可以使用 async\_read\_untile() 方法：  boost::asio::async\_read\_until(socket, readbuffer, "\r\n\r\n", read\_handler);  其中 socket 就是我们的 socket 连接，而 readbuffer 就是根据界定符读取到的一行数据，"\r\n\r\n" 就是分隔符，而对于read\_handler 我们还需要再进一步讨论。  read\_handler 是一个无返回类型的函数对象，它接受两个参数，一个是 boost::system::error\_code，另一个是 size\_t（bytes\_transferred）：  **void read\_handler(**  **const boost::system::error\_code& ec,**  **std::size\_t bytes\_transferred)**  **{**  **...**  **}**  boost::system::error\_code 用来描述操作是否成功，而 size\_t bytes\_transferred 则是用来确定接受到的字节数，通常情况下，我们可以用 std::bind 来将参数绑定到我们的某个函数传入，但实际上我们还有更好的做法，那就是 lambda 表达式，因为 Lambda 表达式还具有另外的一个功能，那就是进行值捕获，对于这一点，我们在后面实现框架的时候再详细讨论。  在这个 read\_handler 中，我们实际上是在不断的读取 socket 里面的内容，因此我们还需要使用 boost::asio::async\_read对后面的内容进行进一步的读取，而它的用法 和 boost::asio::async\_read\_until 几乎一样，唯一的区别就是在 read\_handler 这个参数之前，需要指定 读取的长度，通常我们可以使用 boost::asio::transfer\_exactly 进行指定，故这里不再详细赘述，我们在后面实现框架的时候，再详细讨论。  最后，我们完成了读取的操作，就只剩下最后一步了，那就是服务器响应请求，回写请求的资源供给客户端，这时候我们就需要使用另一个方法：boost::asio::async\_write。从名字上可以看出，这个方法和 boost::asio::async\_read 属于同一个方法家族，可想而知用法也完全类似，我们还是留到后面的实际代码中再进行讨论。 总结 本节中我们回顾了 C++11 的相关知识，并对 Boost 中的 Asio 做了一些介绍。在下一节中，我们将综合运用上面提到的全部知识，开始实现我们的框架。 参考资料  1. [C++ 11/14/17 标准在各编译器下支持情况](http://en.cppreference.com/w/cpp/compiler_support) 2. [C++ IO 库](http://en.cppreference.com/w/cpp/io) 3. [C++ 正则表达式库](http://en.cppreference.com/w/cpp/regex) 4. [Boost Asio 库](http://www.boost.org/doc/libs/1_61_0/doc/html/boost_asio.html) |

#### 参考4：

|  |
| --- |
| C++ 开发 Web 服务框架 - HTTP Web 框架的设计与实现一、概述项目介绍 服务器开发中 Web 服务是一个基本的代码单元，将服务端的请求和响应部分的逻辑抽象出来形成框架，能够做到最高级别的框架级代码复用。本次项目将综合使用 C++11 及 Boost 中的 Asio 实现 HTTP 和 HTTPS 的服务器框架。 项目涉及的知识点  * C++基本知识   + 面向对象   + 模板   + 命名空间   + 常用 IO 库 * C++11 相关   + lambda expression   + std::shared\_ptr   + std::make\_shared   + std::unordered\_map   + std::regex   + std::smatch   + std::regex\_match   + std::function   + std::thread * Boost Asio 相关   + boost::asio::io\_service   + boost::asio::ip::tcp::socket   + boost::asio::ip::tcp::v4()   + boost::asio::ip::tcp::endpoint   + boost::asio::ip::tcp::acceptor   + boost::asio::streambuf   + boost::asio::async\_read   + boost::asio::async\_read\_until   + boost::asio::async\_write   + boost::asio::transfer\_exactly   + boost::asio::ssl::stream   + boost::asio::ssl::stream\_base::server   + boost::asio::ssl::context   + boost::asio::ssl::context::sslv23   + boost::asio::ssl::context::pem   + boost::system::error\_code  编译环境提示 在 g++ 4.9 之前，regex 库并不支持 ECMAScript 的正则语法，换句话说，在 g++4.9 之前，g++ 对 C++11 标准库的支持并不完善，为保证本次项目的顺利进行，请确保将 g++ 版本升级至 4.9 以上。  **// 下面的这段代码可以测试你的编译器对正则表达式的支持情况**  **#include <iostream>**  **#include <regex>**    **int main()**  **{**  **std::regex r1("S");**  **printf("S works.\n");**  **std::regex r2(".");**  **printf(". works.\n");**  **std::regex r3(".+");**  **printf(".+ works.\n");**  **std::regex r4("[0-9]");**  **printf("[0-9] works.\n");**  **return 0;**  **}**  如果你的运行结果遇到了下图所示的错误，说明你确实需要升级你的 g++ 了：  使用 g++ -v 可以查看到当前编译器版本：  如果你最后一行中的 gcc version 显示的是 4.8.x，那么你需要手动将编译器版本升级至 4.9 以上，方法如下：  **# 安装 add-apt-repository 工具**  **sudo apt-get install software-properties-common**  **# 增加源**  **sudo add-apt-repository ppa:ubuntu-toolchain-r/test**  **# 更新源**  **sudo apt-get update**  **# 更新安装**  **sudo apt-get upgrade**  **# 安装 gcc/g++ 4.9**  **sudo apt-get install gcc-4.9 g++-4.9**  **# 更新链接**  **sudo updatedb**  **sudo ldconfig**  **sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 48 \**  **--slave /usr/bin/g++ g++ /usr/bin/g++-4.8 \**  **--slave /usr/bin/gcc-ar gcc-ar /usr/bin/gcc-ar-4.8 \**  **--slave /usr/bin/gcc-nm gcc-nm /usr/bin/gcc-nm-4.8 \**  **--slave /usr/bin/gcc-ranlib gcc-ranlib /usr/bin/gcc-ranlib-4.8**  **sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.9 49 \**  **--slave /usr/bin/g++ g++ /usr/bin/g++-4.9 \**  **--slave /usr/bin/gcc-ar gcc-ar /usr/bin/gcc-ar-4.9 \**  **--slave /usr/bin/gcc-nm gcc-nm /usr/bin/gcc-nm-4.9 \**  **--slave /usr/bin/gcc-ranlib gcc-ranlib /usr/bin/gcc-ranlib-4.9** 设计前的思考 我们知道，HTTP 和 HTTPS 都是应用层的一种协议，他们的底层实际上是通过 TCP 进行传输的。因此，要实现一个 Web 框架，就必须要对浏览器访问 Web 服务器的过程做一个了解。  首先，服务端已经运行起了服务，因此在服务器启动后便开始通过 Socket 监听端口上的请求。这时，客户端浏览器想要访问服务器资源时，就会发送相应的 HTTP 或者 HTTPS 请求。当服务端收到请求后，就会处理这部分请求，返回客户端所需的资源。  仔细思考后不难发现，HTTP 和 HTTPS 两种方式的服务器之间在处理请求、返回请求唯一区别就在于他们究竟如何处理与客户端建立连接的方式上，也就是常说的 accept() 方法。  因此，我们在设计基类的时候，就可以将 accept() 方法设计成一个虚函数，留给具体的子类来实现这个方法，而真正对于服务器内部的相关逻辑，全部由基类 ServerBase 来实现。  对此，我们先确定好基类的基本设计，在ServerBase模板类中，我们只需预留一个接口start()用于启动服务器给外部调用。  由于子类需要实现虚函数 accept() 的具体方式，因此需要保留为 protected，由于我们需要对请求进行解析和应答，而这部分逻辑其实是与协议类型无关的，因此我们可以将这部分逻辑直接放到 ServerBase 中进行实现，到目前为止，我们有：  **#ifndef SERVER\_BASE\_HPP**  **#define SERVER\_BASE\_HPP**    **#include <unordered\_map>**  **#include <thread>**    **namespace ShiyanlouWeb {**    **// socket\_type 为 HTTP or HTTPS**  **template <typename socket\_type>**  **class ServerBase {**  **public:**  **// 启动服务器**  **void start();**  **protected:**  **// 需要不同类型的服务器实现这个方法**  **virtual void accept() {}**  **// 处理请求和应答**  **void process\_request\_and\_respond(std::shared\_ptr<socket\_type> socket) const;**  **};**    **template<typename socket\_type>**  **class Server : public ServerBase<socket\_type> {};**  **}**  **#endif /\* SERVER\_BASE\_HPP \*/** 二、设计 ServerBase 实现一个 Web 服务器，最重要的就是对来自客户端的请求信息进行解析，为此，我们需要在 ShiyanlouWeb 命名空间中定义一个 Request 结构体：  **struct Request {**  **// 请求方法, POST, GET; 请求路径; HTTP 版本**  **std::string method, path, http\_version;**  **// 对 content 使用智能指针进行引用计数**  **std::shared\_ptr<std::istream> content;**  **// 哈希容器, key-value 字典**  **std::unordered\_map<std::string, std::string> header;**  **// 用正则表达式处理路径匹配**  **std::smatch path\_match;**  **};**  这个结构体用于解析请求，如请求方法，请求路径，HTTP 版本等信息。同时，并定义一个 std::istream 指针来保存请求体中包含的内容，由于我们并不关心 header 中信息的顺序，所以可以考虑使用 std::unordered\_map 来指定一个无序容器，保存 header。此外，由于还需要对请求路径进行解析，我们不妨用正则表达式来处理这部分的解析。  在定义好请求后，我们还需要考虑服务器资源的类型。这个资源类型，决定了我们如何让别人使用我们的库，首先我们定义资源类型：  **typedef std::map<std::string, std::unordered\_map<std::string,**  **std::function<void(std::ostream&, Request&)>>> resource\_type;**  并在ServerBase中定义好资源成员：  **template <typename socket\_type>**  **class ServerBase {**  **public:**  **// 用于服务器访问资源处理方式**  **resource\_type resource;**  **// 用于保存默认资源的处理方式**  **resource\_type default\_resource;**  **protected:**  **// 用于内部实现对所有资源的处理**  **std::vector<resource\_type::iterator> all\_resources;**  **……**  首先，resource\_type 是一个 std::map，其键为一个字符串，值则为一个无序容器std::unordered\_map，这个无序容器的键依然是一个字符串，其值接受一个返回类型为空、参数为 ostream 和 Request 的函数。  因此，我们在使用这套框架的时候，当我们有了一个 Server 的对象，定义资源可以是如下的形式：  // 处理访问 /info 的 GET 请求, 返回请求的信息  server.resource["^/info/?$"]["GET"] = [](ostream& response, Request& request) {  // 处理请求及资源  // ...  };  其中，std::map 用于存储请求路径的正则表达式，而 std::unordered\_map 用于存储请求方法，而最后通过一个匿名 Lambda 函数来保存处理方法。有了资源类型，我们仅仅只是定义了当他人使用这套框架时的接口。  为此，我们还需要考虑实现其他接口需要的成员及方法。  首先，Boost Asio 库要求每一个应用都具有一个 io\_service 对象的调度器，而实现 TCP socket 连接，就需要一个 acceptor 对象，而初始化一个 acceptor 还需要有一个 endpoint 对象，因此，我们需要在 ServerBase 类中的 protected 作用域内定义：  **boost::asio::io\_service m\_io\_service;**  **boost::asio::ip::tcp::endpoint endpoint;**  **boost::asio::ip::tcp::acceptor acceptor;**  单个线程的服务器注定是鸡肋的，所以我们不妨在内部实现一个线程池，所以我们可以继续定义：  **size\_t num\_threads;**  **std::vector<std::thread> threads;**  整个 ServerBase 应该在被构造时完成一些关键成员的初始化，对于 endpoint 我们可以将其通过tcp::v4()及申明的端口号进行初始化，完成后，在将 io\_service 对象和 endpoint 共同交给 acceptor 进行初始化，因此我们有构造函数：  **ServerBase(unsigned short port, size\_t num\_threads=1) :**  **endpoint(boost::asio::ip::tcp::v4(), port),**  **acceptor(m\_io\_service, endpoint),**  **num\_threads(num\_threads) {}**  至此，我们整个 ServerBase 中的设计就变成了：  **//**  **// server\_base.hpp**  **// web\_server**  **// created by changkun at shiyanlou.com**  **//**    **#ifndef SERVER\_BASE\_HPP**  **#define SERVER\_BASE\_HPP**    **#include <boost/asio.hpp>**    **#include <regex>**  **#include <unordered\_map>**  **#include <thread>**    **namespace ShiyanlouWeb {**  **struct Request {**  **// 请求方法, POST, GET; 请求路径; HTTP 版本**  **std::string method, path, http\_version;**  **// 对 content 使用智能指针进行引用计数**  **std::shared\_ptr<std::istream> content;**  **// 哈希容器, key-value 字典**  **std::unordered\_map<std::string, std::string> header;**  **// 用正则表达式处理路径匹配**  **std::smatch path\_match;**  **};**    **// 使用 typedef 简化资源类型的表示方式**  **typedef std::map<std::string, std::unordered\_map<std::string,**  **std::function<void(std::ostream&, Request&)>>> resource\_type;**    **// socket\_type 为 HTTP or HTTPS**  **template <typename socket\_type>**  **class ServerBase {**  **public:**  **resource\_type resource;**  **resource\_type default\_resource;**    **// 构造服务器, 初始化端口, 默认使用一个线程**  **ServerBase(unsigned short port, size\_t num\_threads=1) :**  **endpoint(boost::asio::ip::tcp::v4(), port),**  **acceptor(m\_io\_service, endpoint),**  **num\_threads(num\_threads) {}**    **void start();**  **protected:**  **// asio 库中的 io\_service 是调度器，所有的异步 IO 事件都要通过它来分发处理**  **// 换句话说, 需要 IO 的对象的构造函数，都需要传入一个 io\_service 对象**  **boost::asio::io\_service m\_io\_service;**  **// IP 地址、端口号、协议版本构成一个 endpoint，并通过这个 endpoint 在服务端生成**  **// tcp::acceptor 对象，并在指定端口上等待连接**  **boost::asio::ip::tcp::endpoint endpoint;**  **// 所以，一个 acceptor 对象的构造都需要 io\_service 和 endpoint 两个参数**  **boost::asio::ip::tcp::acceptor acceptor;**    **// 服务器线程**  **size\_t num\_threads;**  **std::vector<std::thread> threads;**    **// 所有的资源及默认资源都会在 vector 尾部添加, 并在 start() 中创建**  **std::vector<resource\_type::iterator> all\_resources;**    **// 需要不同类型的服务器实现这个方法**  **virtual void accept() {}**    **// 处理请求和应答**  **void process\_request\_and\_respond(std::shared\_ptr<socket\_type> socket) const;**  **};**    **template<typename socket\_type>**  **class Server : public ServerBase<socket\_type> {};**  **}**  **#endif /\* SERVER\_BASE\_HPP \*/** 三、实现 ServerBase 真正要实现的只有两个方法：   1. void ServerBase::start() 2. void ServerBase::process\_request\_and\_respond()   首先来实现 start()。实现 start() 时，我们要将考虑下面几个问题：   1. 默认资源应该在最后被添加到 all\_resources 中，当我们处理请求路径时，应该先处理好所有的非特殊路径，当找不到匹配请求路径时，再使用默认的请求资源。 2. 当指定启用多个线程时，线程启动后需要让线程各自等待，直到整个请求应答过程结束   **void start() {**  **// 默认资源放在 vector 的末尾, 用作默认应答**  **// 默认的请求会在找不到匹配请求路径时，进行访问，故在最后添加**  **for(auto it=resource.begin(); it!=resource.end();it++) {**  **all\_resources.push\_back(it);**  **}**  **for(auto it=default\_resource.begin(); it!=default\_resource.end();it++) {**  **all\_resources.push\_back(it);**  **}**    **// 调用 socket 的连接方式，还需要子类来实现 accept() 逻辑**  **accept();**    **// 如果 num\_threads>1, 那么 m\_io\_service.run()**  **// 将运行 (num\_threads-1) 线程成为线程池**  **for(size\_t c=1;c<num\_threads;c++) {**  **threads.emplace\_back([this](){**  **m\_io\_service.run();**  **});**  **}**    **// 主线程**  **m\_io\_service.run();**    **// 等待其他线程，如果有的话, 就等待这些线程的结束**  **for(auto& t: threads)**  **t.join();**  **}**  再来实现 process\_request\_and\_respond()：  **// 处理请求和应答**  **void process\_request\_and\_respond(std::shared\_ptr<socket\_type> socket) const {**  **// 为 async\_read\_untile() 创建新的读缓存**  **// shared\_ptr 用于传递临时对象给匿名函数**  **// 会被推导为 std::shared\_ptr<boost::asio::streambuf>**  **auto read\_buffer = std::make\_shared<boost::asio::streambuf>();**    **boost::asio::async\_read\_until(\*socket, \*read\_buffer, "\r\n\r\n",**  **[this, socket, read\_buffer](const boost::system::error\_code& ec, size\_t bytes\_transferred) {**  **if(!ec) {**  **// 注意：read\_buffer->size() 的大小并一定和 bytes\_transferred 相等， Boost 的文档中指出：**  **// 在 async\_read\_until 操作成功后, streambuf 在界定符之外可能包含一些额外的的数据**  **// 所以较好的做法是直接从流中提取并解析当前 read\_buffer 左边的报头, 再拼接 async\_read 后面的内容**  **size\_t total = read\_buffer->size();**    **// 转换到 istream**  **std::istream stream(read\_buffer.get());**    **// 被推导为 std::shared\_ptr<Request> 类型**  **auto request = std::make\_shared<Request>();**    **// 接下来要将 stream 中的请求信息进行解析，然后保存到 request 对象中**  **……**  **});**  **}**  当我们通过 read\_buffer 拿到 istream 对象后，就需要对这个这些信息进行解析，然后保存到 request 中，为此，我们不妨增加一个 parse\_request() 方法：  **// 解析请求**  **Request parse\_request(std::istream& stream) const {**  **Request request;**    **// 使用正则表达式对请求报头进行解析，通过下面的正则表达式**  **// 可以解析出请求方法(GET/POST)、请求路径以及 HTTP 版本**  **std::regex e("^([^ ]\*) ([^ ]\*) HTTP/([^ ]\*)$");**    **std::smatch sub\_match;**    **//从第一行中解析请求方法、路径和 HTTP 版本**  **std::string line;**  **getline(stream, line);**  **line.pop\_back();**  **if(std::regex\_match(line, sub\_match, e)) {**  **request.method = sub\_match[1];**  **request.path = sub\_match[2];**  **request.http\_version = sub\_match[3];**    **// 解析头部的其他信息**  **bool matched;**  **e="^([^:]\*): ?(.\*)$";**  **do {**  **getline(stream, line);**  **line.pop\_back();**  **matched=std::regex\_match(line, sub\_match, e);**  **if(matched) {**  **request.header[sub\_match[1]] = sub\_match[2];**  **}**  **} while(matched==true);**  **}**  **return request;**  **}**  然后我们来继续实现 process\_request\_and\_respond() 方法：  **// 处理请求和应答**  **void process\_request\_and\_respond(std::shared\_ptr<socket\_type> socket) const {**  **auto read\_buffer = std::make\_shared<boost::asio::streambuf>();**  **boost::asio::async\_read\_until(\*socket, \*read\_buffer, "\r\n\r\n",**  **[this, socket, read\_buffer](const boost::system::error\_code& ec, size\_t bytes\_transferred) {**  **if(!ec) {**  **……**    **// 被推导为 std::shared\_ptr<Request> 类型**  **auto request = std::make\_shared<Request>();**  **\*request = parse\_request(stream);**    **size\_t num\_additional\_bytes = total-bytes\_transferred;**    **// 如果满足，同样读取**  **if(request->header.count("Content-Length")>0) {**  **boost::asio::async\_read(\*socket, \*read\_buffer,**  **boost::asio::transfer\_exactly(stoull(request->header["Content-Length"]) - num\_additional\_bytes),**  **[this, socket, read\_buffer, request](const boost::system::error\_code& ec, size\_t bytes\_transferred) {**  **if(!ec) {**  **// 将指针作为 istream 对象存储到 read\_buffer 中**  **request->content = std::shared\_ptr<std::istream>(new std::istream(read\_buffer.get()));**  **respond(socket, request);**  **}**  **});**  **} else {**  **respond(socket, request);**  **}**  **}**  **});**  **}**  最后，在代码的最后，我们需要将请求的内容和 socket 一同传递给 respond() 来处理应答，因此，还需要增加一个 respond() 方法：  **// 应答**  **void respond(std::shared\_ptr<socket\_type> socket, std::shared\_ptr<Request> request) const {**  **// 对请求路径和方法进行匹配查找，并生成响应**  **for(auto res\_it: all\_resources) {**  **std::regex e(res\_it->first);**  **std::smatch sm\_res;**  **if(std::regex\_match(request->path, sm\_res, e)) {**  **if(res\_it->second.count(request->method)>0) {**  **request->path\_match = move(sm\_res);**    **// 会被推导为 std::shared\_ptr<boost::asio::streambuf>**  **auto write\_buffer = std::make\_shared<boost::asio::streambuf>();**  **std::ostream response(write\_buffer.get());**  **res\_it->second[request->method](response, \*request);**    **// 在 lambda 中捕获 write\_buffer 使其不会再 async\_write 完成前被销毁**  **boost::asio::async\_write(\*socket, \*write\_buffer,**  **[this, socket, request, write\_buffer](const boost::system::error\_code& ec, size\_t bytes\_transferred) {**  **// HTTP 持久连接(HTTP 1.1), 递归调用**  **if(!ec && stof(request->http\_version)>1.05)**  **process\_request\_and\_respond(socket);**  **});**  **return;**  **}**  **}**  **}**  **}** 四、实现 HTTP Server 当我们实现完报头解析、请求应答这两个重要的逻辑之后，剩下的，就是对针对不同类型的服务器实现不同的 accept() 方法了。  在 Boost 中，HTTP 类型就是普通的 socket 类型（boost::asio::ip::tcp::socket）。为此，我们可以通过以下不到四十行代码简单实现 HTTP 服务器：  **//**  **// server\_http.hpp**  **// web\_server**  **// created by changkun at shiyanlou.com**  **//**    **#ifndef SERVER\_HTTP\_HPP**  **#define SERVER\_HTTP\_HPP**    **#include "server\_base.hpp"**    **namespace ShiyanlouWeb {**  **typedef boost::asio::ip::tcp::socket HTTP;**  **template<>**  **class Server<HTTP> : public ServerBase<HTTP> {**  **public:**  **// 通过端口号、线程数来构造 Web 服务器, HTTP 服务器比较简单，不需要做相关配置文件的初始化**  **Server(unsigned short port, size\_t num\_threads=1) :**  **ServerBase<HTTP>::ServerBase(port, num\_threads) {};**  **private:**  **// 实现 accept() 方法**  **void accept() {**  **// 为当前连接创建一个新的 socket**  **// Shared\_ptr 用于传递临时对象给匿名函数**  **// socket 会被推导为 std::shared\_ptr<HTTP> 类型**  **auto socket = std::make\_shared<HTTP>(m\_io\_service);**    **acceptor.async\_accept(\*socket, [this, socket](const boost::system::error\_code& ec) {**  **// 立即启动并接受一个连接**  **accept();**  **// 如果出现错误**  **if(!ec) process\_request\_and\_respond(socket);**  **});**  **}**  **};**  **}**  **#endif /\* SERVER\_HTTP\_HPP \*/** 五、编写测试 现在我们可以来正式使用我们的框架了。到目前为止，我们一共创建了下面的这些文件：  **src**  **├── server\_base.hpp**  **└── server\_http.hpp**  我们的 HTTP Web 框架就只有这两个核心文件。  下面我们可以基于我们的 Web 框架开发一个 Web 服务器了：  首先，我们创建好 main 逻辑：  **//**  **// main\_http.cpp**  **// web\_server**  **// created by changkun at shiyanlou.com**  **//**    **#include "server\_http.hpp"**  **#include "handler.hpp"**    **using namespace ShiyanlouWeb;**    **int main() {**  **// HTTP 服务运行在 12345 端口，并启用四个线程**  **Server<HTTP> server(12345, 4);**  **start\_server<Server<HTTP>>(server);**  **return 0;**  **}**  在这个逻辑中，有 start\_server<T>() 这个方法，传递了一个 Server<HTTP> 对象。 而 handler.hpp 则负责实现我们整个 HTTP Web 服务器实例的逻辑。  在开发这个处理逻辑的时候，我们之前提到的框架资源类型定义了我们向外提供的接口，使用形式如下所示：  **// 处理访问 /info 的 GET 请求, 返回请求的信息**  **server.resource["^/info/?$"]["GET"] = [](ostream& response, Request& request) {**  **// 处理请求及资源**  **// ...**  **};**  为了测试 GET 请求和 POST 请求，我们先创建一个 www 目录来存放我们的 Web 资源，随便编写一些 HTML 代码：  **<!-- www/index.html -->**  **<html>**  **<head>**  **<title>Shiyanlou Web Server Test</title>**  **</head>**  **<body>**  **Hello world in index.html.**  **</body>**  **</html>**  我们可以编写下面的服务器测试代码：  **//**  **// handler.hpp**  **// web\_server**  **// created by changkun at shiyanlou.com**  **//**    **#include "server\_base.hpp"**  **#include <fstream>**    **using namespace std;**  **using namespace ShiyanlouWeb;**    **template<typename SERVER\_TYPE>**  **void start\_server(SERVER\_TYPE &server) {**  **// 向服务器增加请求资源的处理方法**    **// 处理访问 /string 的 POST 请求，返回 POST 的字符串**  **server.resource["^/string/?$"]["POST"] = [](ostream& response, Request& request) {**  **// 从 istream 中获取字符串 (\*request.content)**  **stringstream ss;**  **\*request.content >> ss.rdbuf(); // 将请求内容读取到 stringstream**  **string content=ss.str();**    **// 直接返回请求结果**  **response << "HTTP/1.1 200 OK\r\nContent-Length: " << content.length() << "\r\n\r\n" << content;**  **};**    **// 处理访问 /info 的 GET 请求, 返回请求的信息**  **server.resource["^/info/?$"]["GET"] = [](ostream& response, Request& request) {**  **stringstream content\_stream;**  **content\_stream << "<h1>Request:</h1>";**  **content\_stream << request.method << " " << request.path << " HTTP/" << request.http\_version << "<br>";**  **for(auto& header: request.header) {**  **content\_stream << header.first << ": " << header.second << "<br>";**  **}**    **// 获得 content\_stream 的长度(使用 content.tellp() 获得)**  **content\_stream.seekp(0, ios::end);**    **response << "HTTP/1.1 200 OK\r\nContent-Length: " << content\_stream.tellp() << "\r\n\r\n" << content\_stream.rdbuf();**  **};**    **// 处理访问 /match/[字母+数字组成的字符串] 的 GET 请求, 例如执行请求 GET /match/abc123, 将返回 abc123**  **server.resource["^/match/([0-9a-zA-Z]+)/?$"]["GET"] = [](ostream& response, Request& request) {**  **string number=request.path\_match[1];**  **response << "HTTP/1.1 200 OK\r\nContent-Length: " << number.length() << "\r\n\r\n" << number;**  **};**    **// 处理默认 GET 请求, 如果没有其他匹配成功，则这个匿名函数会被调用**  **// 将应答 web/ 目录及其子目录中的文件**  **// 默认文件: index.html**  **server.default\_resource["^/?(.\*)$"]["GET"] = [](ostream& response, Request& request) {**  **string filename = "web/";**    **string path = request.path\_match[1];**    **// 防止使用 `..` 来访问 web/ 目录外的内容**  **size\_t last\_pos = path.rfind(".");**  **size\_t current\_pos = 0;**  **size\_t pos;**  **while((pos=path.find('.', current\_pos)) != string::npos && pos != last\_pos) {**  **current\_pos = pos;**  **path.erase(pos, 1);**  **last\_pos--;**  **}**    **filename += path;**  **ifstream ifs;**  **// 简单的平台无关的文件或目录检查**  **if(filename.find('.') == string::npos) {**  **if(filename[filename.length()-1]!='/')**  **filename+='/';**  **filename += "index.html";**  **}**  **ifs.open(filename, ifstream::in);**    **if(ifs) {**  **ifs.seekg(0, ios::end);**  **size\_t length=ifs.tellg();**    **ifs.seekg(0, ios::beg);**    **// 文件内容拷贝到 response-stream 中，不应该用于大型文件**  **response << "HTTP/1.1 200 OK\r\nContent-Length: " << length << "\r\n\r\n" << ifs.rdbuf();**    **ifs.close();**  **} else {**  **// 文件不存在时，返回无法打开文件**  **string content="Could not open file "+filename;**  **response << "HTTP/1.1 400 Bad Request\r\nContent-Length: " << content.length() << "\r\n\r\n" << content;**  **}**  **};**    **// 运行 HTTP 服务器**  **server.start();**  **}** 六、结果测试 到目前为止，我们整个目录树应该是这个样子：  **├── handler.hpp**  **├── main\_http.cpp**  **├── server\_base.hpp**  **├── server\_http.hpp**  **└── web**  **└── index.html**  由于我们使用了 boost 库，因此需要在编译时告诉编译器去索引 boost 的位置，如果直接使用编译命令编译，会出现指令过长的情况，我们不妨编写一个 Makefile：  **#**  **# Makefile**  **# web\_server**  **#**  **# created by changkun at shiyanlou.com**  **#**    **CXX = g++**  **EXEC\_HTTP = server\_http**    **SOURCE\_HTTP = main\_http.cpp**    **OBJECTS\_HTTP = main\_http.o**    **# 开启编译器 O3 优化, pthread 启用多线程支持**  **LDFLAGS\_COMMON = -std=c++11 -O3 -pthread -lboost\_system**  **LDFLAGS\_HTTP =**    **LPATH\_COMMON = -I/usr/include/boost**  **LPATH\_HTTP =**    **LLIB\_COMMON = -L/usr/lib**  **LLIB\_HTTP =**    **http:**  **$(CXX) $(SOURCE\_HTTP) $(LDFLAGS\_COMMON) $(LDFLAGS\_HTTP) $(LPATH\_COMMON) $(LPATH\_HTTP) $(LLIB\_COMMON) $(LLIB\_HTTP) -o $(EXEC\_HTTP)**    **clean:**  **rm -f $(EXEC\_HTTP) \*.o**  最终，我们能够使用 make http 来编译我们的代码，并通过 ./server\_http 来运行我们的服务器，并在浏览器中测试我们的服务器运行情况：  对于 GET 请求，我们可以直接在浏览器中访问：  localhost:12345/ # 会访问到 index.html  localhost:12345/match/123abc # 会获得到一个 123abc 的字符串  localhost:12345/info/ # 会获得整个请求体的信息  而对于 POST 请求，我们可以使用 curl 命令进行测试：  curl -d "test string" "http://localhost:12345/string/"  这时候能看到服务器返回测试结果，就是我们 POST 发送的字符串。 总结 本节实验我们实现了服务器除开建立 TCP 连接具体实现外的 ServerBase 基类，并从此类继承出了 Server<HTTP> 子类，实现了 HTTP 服务器框架，同时，基于我们编写的框架，我们开发出了一个简易的 HTTP Web 服务器。在整个过程中，我们用到了大量 C++11 和 Boost Asio 的相关知识。在下一节实验中，我们将据此进一步实现 HTTPS 服务器框架，并编写启用 HTTPS 服务器。  原文地址：https://www.shiyanlou.com/courses/568/labs/1984/document |

#### 参考5：

|  |
| --- |
| CppCMS  <http://cppcms.com/wikipp/en/page/main>  以高性能著称，MVC，内容较全，包括数据库、会话等管理，内置http server  TreeFrog  <http://www.treefrogframework.org/>  以Qt为基础开发的，MVC，内容较全，包括数据库、会话生管理，好像是个日本人写的。内置http server  wt  <https://www.webtookkit.eu/wt>  号称是 个Web GUI TookKit，即将HTML组件做成C++类组件，开发方式和Qt十分相似（就像以前的C#的WebForm），内置一个httpd，也可以与FastCGI/ISAPI方式运行  Tofao  <http://vinipsmaker.github.io/tufao/>  也是基于Qt开发的，还有QtCreator插件用了辅助开发，内置http server  Drogon  <https://github.com/an-tao/drogon>  C++14/17 based http web application framework，跨平台的  Oat++  <https://oatpp.io>  Light, zero-dependency, performance oriented Web Framework，pure C++  CWF  <https://github.com/HerikLyma/CPPWebFramework>  这也是个基于Qt的Web开发框架，内阁制http server  Cutelyst  <https://cutelyst.org>  一个Qt Web Framework，这么直白.  Civetweb  <https://github.com/civetweb/civetweb>  一个嵌入式Web服务器  QDjango  <https://github.com/jlaine/qdjango>  一个基于Qt的Web框架，和django的API类似  cinatra  <https://github.com/qicosmos/cinatra>  C++17的，只包含头文件的http framework，简单、高效、跨平台、支持面向切面编程，这个有点意思。这个是purecpp社区开发的（中国的），内置http server  crow  <https://github.com/ipkn/crow>  C++ micro web framework，使用起来只包含个头文件就行了（类似Python Flask)，内置http server  Silicon  <http://siliconframework.org>  一个用于构建C++ Web API的高性能的网络库 ,使用也比较简单看样子，但第一句，就说要求编译器是clang++。。。。。。，而且它只是个库，需要安装个http server配合  fastcgi++  这个就不用说了吧，就是一个C++库，用于开发fastcgi程序  <https://github.com/eddic/fastcgipp> |

# 17.C++ 编写 json 生成器

#### 参考1：

|  |
| --- |
| /\*  json-writer.h  \*/  #ifndef \_JSONWRITER\_H\_  #define \_JSONWRITER\_H\_  #include<iostream>  #include<string>  #include<stack>  #include<cstdint>  using namespace std;  class JsonWriter  {  public:  enum ContainerType  {  CONTAINER\_TYPE\_ARRAY,  CONTAINER\_TYPE\_OBJECT  };  enum ContainerLayout  {  CONTAINER\_LAYOUT\_INHERIT,  CONTAINER\_LAYOUT\_MULTI\_LINE,  CONTAINER\_LAYOUT\_SINGLE\_LINE  };  explicit JsonWriter():  writer(NULL),  initialIndentDepth(0),  indent(" "),  containerPadding(" "),  keyPaddingLeft(""),  keyPaddingRight(" "),  defaultContainerLayout(CONTAINER\_LAYOUT\_MULTI\_LINE),  forceDefaultContainerLayout(false)  {  }    void ConfigureCompressedOutput()  {  SetIndent ("");  SetContainerPadding("");  SetKeyPaddingLeft("");  SetKeyPaddingRight("");  SetDefaultContainerLayout(CONTAINER\_LAYOUT\_MULTI\_LINE);  SetForceDefaultContainerLayout(true);  }  ostream \*GetWriter() {return writer; }  void SetWriter (ostream \*writer)  {  this->writer = writer;  }  int GetInitialIndentDepth()  {  return initialIndentDepth;  }  void SetInitialIndentDepth(int depth)  {  initialIndentDepth = depth;  }  const char \*GetIndent()  {  return indent;  }  void SetIndent(const char \*indent)  {  this->indent = indent;  }  const char \*GetContainerPadding()  {  return containerPadding;  }  void SetContainerPadding(const char \*padding)  {  keyPaddingRight = padding;  }  const char \*GetKeyPaddingLeft()  {  return keyPaddingLeft;  }  void SetKeyPaddingLeft(const char \*padding)  {  keyPaddingLeft = padding;  }  const char \*GetKeyPaddingRight()  {  return keyPaddingRight;  }  void SetKeyPaddingRight(const char \*padding)  {  keyPaddingRight = padding;  }  ContainerLayout GetDefaultContainerLayout()  {  return defaultContainerLayout;  }  void SetDefaultContainerLayout(ContainerLayout layout)  {  defaultContainerLayout = layout;  }  bool GetForceDeafultContainerLayout()  {  return forceDefaultContainerLayout;  }  void SetForceDefaultContainerLayout(bool force)  {  forceDefaultContainerLayout = force;  }  std::ostream& Write()  {  if (writer == NULL)  {  return std::cout;  }  return \*writer;  }  void WriteEscapedChar (char c);  void WriteString(const char \*str);  void StartChild (bool isKey);  void StartChild()  {  StartChild (false);  }  void StartContainer(ContainerType type, ContainerLayout layout);  void EndContainer();  void StartArray()  {  StartContainer(CONTAINER\_TYPE\_ARRAY, CONTAINER\_LAYOUT\_INHERIT);  }  void StartArray(ContainerLayout layout)  {  StartContainer(CONTAINER\_TYPE\_ARRAY, layout);  }  void StartShortArray()  {  StartContainer(CONTAINER\_TYPE\_ARRAY, CONTAINER\_LAYOUT\_SINGLE\_LINE);  }  void EndArray()  {  EndContainer();  }  void StartObject()  {  StartContainer(CONTAINER\_TYPE\_OBJECT, CONTAINER\_LAYOUT\_INHERIT);  }  void StartObject(ContainerLayout layout)  {  StartContainer(CONTAINER\_TYPE\_OBJECT, layout);  }  void StartShortObject()  {  StartContainer (CONTAINER\_TYPE\_OBJECT, CONTAINER\_LAYOUT\_SINGLE\_LINE);  }  void EndObjct()  {  EndContainer();  }  void Key(const char \*key);  void NullValue();  void Value(const char \*value);  void Value(std::string value);  void Value(bool value);  #define VALUE\_DEF(t) void Value (t value) {StartChild(); Write() << value; }  #define KEYVALUE\_DEF(t) void KeyValue (const char \*key, t value) { Key (key); Value (value); }  VALUE\_DEF(int8\_t)  VALUE\_DEF(uint8\_t)  VALUE\_DEF(int16\_t)  VALUE\_DEF(uint16\_t)  VALUE\_DEF(int32\_t)  VALUE\_DEF(uint32\_t)  VALUE\_DEF(int64\_t)  VALUE\_DEF(uint64\_t)  VALUE\_DEF(float)  VALUE\_DEF(double)  void KeyNullValue( const char \*key)  {  Key (key);  NullValue();  }  KEYVALUE\_DEF(const char \*)  KEYVALUE\_DEF(std::string)  KEYVALUE\_DEF(bool)  KEYVALUE\_DEF(int8\_t)  KEYVALUE\_DEF(uint8\_t)  KEYVALUE\_DEF(int16\_t)  KEYVALUE\_DEF(uint16\_t)  KEYVALUE\_DEF(int32\_t)  KEYVALUE\_DEF(uint32\_t)  KEYVALUE\_DEF(int64\_t)  KEYVALUE\_DEF(uint64\_t)  KEYVALUE\_DEF(float)  KEYVALUE\_DEF(double)  private:  ostream \*writer;  int initialIndentDepth;  const char \*indent;  const char \*containerPadding;  const char \*keyPaddingLeft;  const char \*keyPaddingRight;  ContainerLayout defaultContainerLayout;  bool forceDefaultContainerLayout;  struct Container  {  ContainerType type;  ContainerLayout layout;  bool isKey;  int childCount;  Container (ContainerType type , ContainerLayout layout):  type (type),  layout(layout),  isKey(false),  childCount(0)  {  }  };  std::stack<Container \*> depth;  void Indent();  };  #endif |
| /\*  json-writer.cpp  \*/  #include"json-writer.h"  using namespace std;  void JsonWriter::Indent()  {  for (int i = 0, s = initialIndentDepth + depth.size(); i < s; i++)  {  Write() << indent;  }  }  void JsonWriter::StartChild(bool isKey)  {  if (0 == depth.size())  {  if (initialIndentDepth > 0)  {  Indent();  }  return;  }  Container \*container = depth.top();  if (container->childCount > 0 &&  (container->type == CONTAINER\_TYPE\_ARRAY ||  (container->type == CONTAINER\_TYPE\_OBJECT && !container->isKey)))  {  Write() << ",";  if (container->layout == CONTAINER\_LAYOUT\_SINGLE\_LINE)  {  Write() << containerPadding;  }  else  {  Write() << endl;  Indent();  }  }  else if (container->childCount == 0)  {  Write() << containerPadding;  if (container->layout == CONTAINER\_LAYOUT\_MULTI\_LINE)  {  Write() << endl;  Indent();  }  }    container->isKey = isKey;  container->childCount++;  }  void JsonWriter::Key (const char \*key)  {  StartChild (true);  WriteString(key);  Write() << keyPaddingLeft << ":" << keyPaddingRight;  }  void JsonWriter::NullValue()  {  StartChild();  Write() << "null";  }  void JsonWriter::Value(const char \* value)  {  StartChild();  WriteString(value);  }  void JsonWriter::Value(string value)  {  StartChild();  WriteString(value.c\_str());  }  void JsonWriter::Value (bool value)  {  StartChild();  Write() << (value ? "true" : "false");  }    void JsonWriter::StartContainer(ContainerType type, ContainerLayout layout)  {  if (forceDefaultContainerLayout)  {  layout = defaultContainerLayout;  }  else if (layout == CONTAINER\_LAYOUT\_INHERIT)  {  if (depth.size() > 0)  {  layout = depth.top()->layout;  }  else  {  layout = defaultContainerLayout;  }  }  StartChild();  depth.push(new Container(type, layout));  Write() << (type == CONTAINER\_TYPE\_OBJECT ? '{' : '[');  }  void JsonWriter::EndContainer()  {  Container \*container = depth.top();  depth.pop();  if (container->childCount > 0)  {  if (container->layout == CONTAINER\_LAYOUT\_MULTI\_LINE)  {  Write() << endl;  Indent();  }  else  {  Write() << containerPadding;  }  }  Write() << (container->type == CONTAINER\_TYPE\_OBJECT ? '}' : ']');  delete container;  }  void JsonWriter::WriteEscapedChar(char c)  {  switch(c)  {  case '"': Write() << "\\\""; break;  case '\\': Write() << "\\\\"; break;  case '\b': Write() << "\\b"; break;  case '\f': Write() << "\\f"; break;  case '\n': Write() << "\\n"; break;  case '\r': Write() << "\\r"; break;  case '\t': Write() << "\\t"; break;  default: Write() << c; break;  }  }  void JsonWriter::WriteString(const char \*str)  {  Write() << "\"";  for (int i = 0; str[i] != 0; i++)  {  WriteEscapedChar(str[i]);  }  Write() << "\"";  } |
| 测试：  #include<iostream>  #include<cmath>  #include"json-writer.h"  int main()  {  auto writer = new JsonWriter;  writer->StartArray();  writer->StartShortObject();  writer->KeyValue("name", "shiyanlou");  writer->KeyValue("age", "3");  writer->EndObjct();  writer->StartObject();  writer->KeyValue("skills", "c++");  writer->KeyValue("skills", "python");  writer->KeyValue("skills", "php");  writer->KeyValue("skills", "java");  writer->KeyValue("uri", "http://shiyanlou.com");  writer->Key("path");  writer->StartArray();  writer->Value("web");  writer->Value("algorithm");  writer->Value("linux");  writer->EndArray();  writer->Key("short-array");  writer->StartShortArray();  writer->Value(1);  writer->Value((uint64\_t)0xabcdef123456);  writer->Value(M\_PI);  writer->EndContainer();  writer->EndObjct();  writer->Value(false);  writer->EndArray();  }  [复制代码](javascript:void(0);)    多种格式测试代码：  [复制代码](javascript:void(0);)  #include "json-writer.h"  #define BEGIN\_TEST(name) \  { \  JsonWriter \*w = new JsonWriter; \  w->SetInitialIndentDepth(2); \  if (compress) \  { \  w->ConfigureCompressedOutput(); \  } \  cout << #name << ":" << endl << endl;  #define END\_TEST \  delete w; \  cout << endl << endl; \  }  int main()  {  bool compress;  for (int i = 0; i < 2; compress = i == 0, i++)  {  BEGIN\_TEST(null)  w->NullValue();  END\_TEST  BEGIN\_TEST(bool-false)  w->Value(false);  END\_TEST  BEGIN\_TEST(bool-true)  w->Value(true);  END\_TEST  BEGIN\_TEST(int)  w->Value(30000);  END\_TEST  BEGIN\_TEST(double)  w->Value(0.131415926);  END\_TEST  BEGIN\_TEST(empty-string)  w->Value("");  END\_TEST  BEGIN\_TEST(simple-string)  w->Value("Hello");  END\_TEST  BEGIN\_TEST(escaped-string)  w->Value("\"newline\ntab\t\"");  END\_TEST  BEGIN\_TEST(empty-object)  w->StartObject();  w->EndObjct();  END\_TEST  BEGIN\_TEST(empty-array)  w->StartArray();  w->EndArray();  END\_TEST  BEGIN\_TEST(short-object)  w->StartShortObject();  w->KeyValue("name", "Aaron");  w->EndObjct();  END\_TEST  BEGIN\_TEST(short-array)  w->StartShortArray();  for(int i = 0; i < 10; i++)  {  w->Value(i);  }  w->EndArray();  END\_TEST  BEGIN\_TEST(array-with-object)  w->StartArray();  w->StartShortObject();  w->KeyValue("name", "Aaron");  w->KeyValue("age", 7);  w->EndObjct();  w->StartObject();  w->KeyValue("animal", "cat");  w->KeyValue("life-expectancy", "forever");  w->KeyValue("url", "http://catoverflow.com");  w->Key("catch-phrases");  w->StartArray();  w->Value("meow");  w->Value("hiss");  w->Value("purr");  w->EndArray();  w->EndObjct();  w->Value(false);  w->EndArray();  END\_TEST  BEGIN\_TEST(nested-objects)  w->StartObject();  w->Key("a");  w->StartObject();  w->Key("b");  w->StartObject();  w->Key("c");  w->StartObject();  w->EndObjct();  w->EndObjct();  w->EndObjct();  w->EndArray();  END\_TEST  BEGIN\_TEST(nested-arrays)  w->StartArray();  w->StartArray();  w->StartArray();  w->StartArray();  w->EndArray();  w->EndArray();  w->EndArray();  w->EndArray();  END\_TEST  }  return 0;  }  [复制代码](javascript:void(0);)    makefile文件：   |  | | --- | | CXXFLAGS = -Wall -g -std=c++11  all: test json-writer-test  clean:  rm -rf \*.o test json-writer-test  test: json-writer.h json-writer.cpp test.cpp  $(CXX) $(CXXFLAGS) json-writer.h json-writer.cpp test.cpp -o test  json-writer-test: json-writer-test.cpp json-writer.h json-writer.cpp  $(CXX) $(CXXFLAGS) json-writer.h json-writer.cpp json-writer-test.cpp -o json-writer-test | |

#### 参考2：

GitHub：<https://github.com/LiDing0611/JsonGenerator>

已经下载的资源：<JsonGenerator-master>

#### 参考3：

<https://blog.csdn.net/qq_35865125/article/details/101047672>

#### 参考4：

<https://blog.csdn.net/piaopiaopiaopiaopiao/article/details/104513216>

<https://github.com/open-source-parsers/jsoncpp>

#### 参考5：

RapidJson，GitHub：<https://github.com/Tencent/rapidjson>

# 18.C++ 利用 QT 实现 PDF 阅读器

#### 参考1：

<https://blog.csdn.net/chy555chy/article/details/51613545>

|  |
| --- |
| **Qt 5.5 使用Poppler实现pdf阅读器**  Qt实现pdf阅读器和[MFC](https://so.csdn.net/so/search?q=MFC&spm=1001.2101.3001.7020)实现pdf阅读器，其实原理都是差不多的。  需要用到Poppler开源库，下载地址如下 <https://poppler.freedesktop.org/>  如果只是要在window的gcc下运行的话，可以下载已经编译好的库 <https://sourceforge.net/projects/poppler-win32/>  https://img-blog.csdn.net/20161128183907971?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  注意：这个是MinGW版本的Qt，也就是运行在GCC环境下的库，里面只包含 \*.dll 和 \*.a 。如果是Vistual Studio版本的Qt ，那么很不幸里面没有 \*.lib文件。  1、新建项目，在项目的[根目录](https://so.csdn.net/so/search?q=%E6%A0%B9%E7%9B%AE%E5%BD%95&spm=1001.2101.3001.7020)新建一个“poppler”文件夹，将poppler中qt5目录下的文件都丢进去（\*.h头文件，另外再将编译好的2个\*.a文件和2个\*.dll丢进去，我这里多丢了实现的\*.cc文件，因为\*.cc已经被编译成动态库了，所以可以不用包含在代码中）  https://img-blog.csdn.net/20160608154252483?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  2、在项目的pro配置文件中添加以下内容，引用poppler的[头文件](https://so.csdn.net/so/search?q=%E5%A4%B4%E6%96%87%E4%BB%B6&spm=1001.2101.3001.7020)和库文件（注意：我这里是win32，所以前面加了win32前缀）  INCLUDEPATH += $$PWD/poppler  win32: LIBS += -L$$PWD/poppler -llibpoppler  win32: LIBS += -L$$PWD/poppler -llibpoppler-qt5  3、创建pdf工具类（该类负责与poppler库做对接，主要负责获取pdf的总页数，和每页的图像）  （1）pdfutils.h  **#ifndef PDFUTILS\_H**  **#define PDFUTILS\_H**  **#include <QObject>**  **#include <QImage>**  **#include <QSize>**  **#include <QDebug>**  **#include "poppler-qt5.h"**  **class PdfUtils**  **{**  **public:**  **explicit PdfUtils(QString filePath);**  **~PdfUtils();**  **//获取指定页pdf图像(页码从0开始)**  **QImage getPdfImage(int pageNumber);**  **//获取pdf总页码**  **int getNumPages();**  **//获取pdf页面大小**  **QSize getPageSize();**  **private:**  **QString filePath;**  **int numPages;**  **QSize pageSize;**  **void getPdfInfo();**  **};**  **#endif // PDFUTILS\_H**  **（2）pdfutils.cpp**  **#include "pdfutils.h"**  **PdfUtils::PdfUtils(QString filePath) {**  **this->filePath = filePath;**  **getPdfInfo();**  **}**  **PdfUtils::~PdfUtils() {**  **}**  **QImage PdfUtils::getPdfImage(int pageNumber) {**  **QImage image;**  **Poppler::Document\* document = Poppler::Document::load(filePath);**  **if (!document || document->isLocked()) {**  **// ... error message ....**  **delete document;**  **return image;**  **}**  **// Document starts at page 0**  **Poppler::Page\* pdfPage = document->page(pageNumber);**  **if (pdfPage == 0) {**  **// ... error message ...**  **return image;**  **}**  **// Generate a QImage of the rendered page**  **image = pdfPage->renderToImage(72, 72, -1, -1, -1, -1);**  **if (image.isNull()) {**  **// ... error message ...**  **return image;**  **}**  **// after the usage, the page must be deleted**  **delete pdfPage;**  **delete document;**  **return image;**  **}**  **int PdfUtils::getNumPages() {**  **return numPages;**  **}**  **QSize PdfUtils::getPageSize() {**  **return pageSize;**  **}**  **void PdfUtils::getPdfInfo() {**  **numPages = 0;**  **Poppler::Document\* document = Poppler::Document::load(filePath);**  **if (!document || document->isLocked()) {**  **// ... error message ....**  **delete document;**  **return;**  **}**  **numPages = document->numPages();**  **Poppler::Page\* pdfPage = document->page(0);**  **pageSize = pdfPage->pageSize();**  **qDebug()<<pageSize;**  **delete pdfPage;**  **delete document;**  **}**  4、pdf显示类（pdf的右侧显示滚动条，①拖动滚动条翻页 ②鼠标拖动pdf到最上或最底时翻页）  注意：本文省略了页面缓存，如果是真实的项目的话，本着严谨的态度，请务必缓存页面  （1）mypdfcanvas.h（继承父类的resizeEvent是为了 ①当pdf只有1页时不显示滚动条 ②当用户拖动缩放窗口时动态改变pdf显示尺寸）  **#ifndef MYPDFCANVAS\_H**  **#define MYPDFCANVAS\_H**  **#include <QWidget>**  **#include <QVector>**  **#include <QMouseEvent>**  **#include <QPaintEvent>**  **#include <QPainter>**  **#include <QPaintEvent>**  **#include <QMap>**  **#include <QPalette>**  **#include <QResizeEvent>**  **#include "pdfutils.h"**  **class MyPdfCanvas : public QWidget**  **{**  **Q\_OBJECT**  **public:**  **explicit MyPdfCanvas(QWidget \*parent = 0);**  **~MyPdfCanvas();**  **void resizeEvent(QResizeEvent\* e);**  **void paintEvent(QPaintEvent \*e);**  **void mousePressEvent(QMouseEvent \*e);**  **void mouseReleaseEvent(QMouseEvent \*e);**  **void mouseMoveEvent(QMouseEvent \*e);**  **void setMaxCachedNum(int maxCachedNum);**  **//如果不能解析pdf返回false**  **bool setPath(QString pdfPath);**  **//页码从0开始**  **bool setPage(int pageNumber);**  **//获取页数**  **int getNumPages();**  **float getScaledRatio();**  **//显示裁剪后的图片**  **bool showClipImage(int pageNumber, int x, int y, int w, int h);**  **//取消显示裁剪图片,恢复正常显示**  **void cancelClip();**  **//实际获取的pdf宽高度**  **QSize pdfActualSize;**  **signals:**  **void pageChanged(int currentPage);**  **private:**  **PdfUtils\* pdfUtils;**  **QString pdfPath;**  **//最大缓存图片数量**  **int maxCachedNum;**  **//用来缓存pdf的每一个页面的图像(从0开始)**  **QMap<int, QImage> cachedImageMap;**  **//用来存储已缓存的pdf页面序号(从0开始)**  **// QQueue<int> cachedPageQueue;**  **//当前页码(从0开始)**  **int currentPage;**  **//总页码(从0开始)**  **int numPages;**  **bool isMouseDown;**  **int lastMouseY;**  **//当前pdf页面的图像**  **QImage image;**  **int imageX;**  **int imageY;**  **int imageMinY;**  **//是否是剪裁状态**  **bool isClip;**  **//获取指定页的图片**  **bool getPdfImage(int pageNumber);**  **void reachTop();**  **void reachBottom();**  **//判断是否需要发送重定位签名框的信号**  **void needLocateSignArea();**  **};**  **#endif // MYPDFCANVAS\_H**  （2）pdfcanvas.cpp  **#include "mypdfcanvas.h"**  **MyPdfCanvas::MyPdfCanvas(QWidget \*parent) : QWidget(parent) {**  **pdfUtils = NULL;**  **imageX = 0;**  **imageY = 0;**  **isClip = false;**  **setAutoFillBackground(true);**  **}**  **MyPdfCanvas::~MyPdfCanvas() {**  **if(pdfUtils != NULL) delete pdfUtils;**  **}**  **void MyPdfCanvas::resizeEvent(QResizeEvent \*e) {**  **image = this->cachedImageMap[currentPage];**  **if(!image.isNull()) {**  **float radio = (float)e->size().width()/(float)e->oldSize().width();**  **int imageHeight = image.height()\* e->size().width()/image.width();**  **image = image.scaled(e->size().width(), imageHeight);**  **if(imageHeight < this->height()) {**  **imageY = (this->height()-imageHeight)/2;**  **//如果图片高度小于控件高度，则图片居中**  **// imageMinY = imageY;**  **imageMinY = 0;**  **imageY = imageMinY;**  **} else {**  **if(radio>0) {**  **imageY = (int)(imageY\*radio);**  **if(imageY > 0) {**  **imageY = 0;**  **}**  **} else {**  **imageY = 0;**  **}**  **}**  **}**  **}**  **void MyPdfCanvas::paintEvent(QPaintEvent \*e) {**  **QPainter\* painter = new QPainter(this);**  **if(image.isNull()) {**  **painter->fillRect(this->rect(), Qt::transparent);**  **return;**  **}**  **painter->drawImage(0, imageY, image);**  **delete painter;**  **}**  **void MyPdfCanvas::mousePressEvent(QMouseEvent \*e) {**  **isMouseDown = true;**  **lastMouseY = e->y();**  **}**  **void MyPdfCanvas::mouseReleaseEvent(QMouseEvent \*e){**  **isMouseDown = false;**  **}**  **void MyPdfCanvas::mouseMoveEvent(QMouseEvent \*e){**  **if(!isMouseDown || image.isNull()) {**  **return;**  **}**  **int distance = e->y() - lastMouseY;**  **lastMouseY = e->y();**  **imageY += distance;**  **if(imageY > 0) {**  **imageY = 0;**  **reachTop();**  **return;**  **} else if(imageY < imageMinY) {**  **imageY = imageMinY;**  **reachBottom();**  **return;**  **}**  **update();**  **}**  **void MyPdfCanvas::setMaxCachedNum(int maxCachedNum) {**  **this->maxCachedNum = maxCachedNum;**  **}**  **bool MyPdfCanvas::setPath(QString pdfPath) {**  **this->pdfPath = pdfPath;**  **if(pdfUtils != NULL) delete pdfUtils;**  **pdfUtils = new PdfUtils(pdfPath);**  **numPages = pdfUtils->getNumPages();**  **if(numPages > 0) {**  **isClip = false;**  **pdfActualSize = pdfUtils->getPageSize();**  **}**  **cachedImageMap.clear();**  **currentPage = 0;**  **imageY = 0;**  **lastMouseY = 0;**  **return numPages > 0;**  **}**  **bool MyPdfCanvas::setPage(int pageNumber) {**  **if(!getPdfImage(pageNumber)) {**  **return false;**  **}**  **isClip = false;**  **isMouseDown = false;**  **image = image.scaledToWidth(this->width());**  **imageMinY = this->height() - image.height();**  **if(image.height() < this->height()) {**  **//如果图片高度小于控件高度，则图片居中**  **// imageMinY /= 2;**  **imageMinY = 0;**  **imageY = imageMinY;**  **} else {**  **imageY = 0;**  **}**  **update();**  **return true;**  **}**  **int MyPdfCanvas::getNumPages() {**  **return numPages;**  **}**  **float MyPdfCanvas::getScaledRatio() {**  **int pdfWidth = pdfUtils->getPageSize().width();**  **return (float)this->width()/(float)pdfWidth;**  **}**  **bool MyPdfCanvas::showClipImage(int pageNumber, int x, int y, int w, int h) {**  **if(!getPdfImage(pageNumber)) {**  **return false;**  **}**  **isClip = true;**  **imageY = 0;**  **image = image.copy(x, y, w, h).scaled(this->size());**  **update();**  **}**  **void MyPdfCanvas::cancelClip() {**  **isClip = false;**  **setPage(currentPage);**  **}**  **bool MyPdfCanvas::getPdfImage(int pageNumber) {**  **if(pageNumber<0 || pageNumber >= numPages) {**  **return false;**  **}**  **if(cachedImageMap.contains(pageNumber)) {**  **image = cachedImageMap.value(pageNumber);**  **} else {**  **image = pdfUtils->getPdfImage(pageNumber);**  **if(!image.isNull()) {**  **cachedImageMap[pageNumber] = image;**  **pdfActualSize = image.size();**  **}**  **}**  **if(image.isNull()) {**  **return false;**  **}**  **currentPage = pageNumber;**  **return true;**  **}**  **void MyPdfCanvas::reachTop() {**  **if(currentPage > 0) {**  **emit pageChanged(currentPage-1);**  **}**  **}**  **void MyPdfCanvas::reachBottom() {**  **if(currentPage < numPages-1) {**  **emit pageChanged(currentPage+1);**  **}**  **}**  5、pdf及右侧滑块的装载容器  （1）mainwindow.h  **#ifndef MAINWINDOW\_H**  **#define MAINWINDOW\_H**  **#include <QMainWindow>**  **#include <QScrollBar>**  **#include "mypdfcanvas.h"**  **#define SCROLLBAR\_WIDTH 30**  **class MainWindow : public QMainWindow**  **{**  **Q\_OBJECT**  **public:**  **explicit MainWindow(QWidget \*parent = 0);**  **~MainWindow();**  **void resizeEvent(QResizeEvent\* e);**  **bool setPdfPath(QString path);**  **//重新调整pdf界面大小**  **void resizeCanvas();**  **void setWidgetVisible(bool pdfCanvasVisible, bool scrollbarVisible);**  **public slots:**  **//当拖动pdf上滑到顶（或下滑到底）时触发该方法**  **onPageChange(int currentPage);**  **//当滑动条的滑块被滑动时，会调用该方法**  **onScrollBarValueChange();**  **private:**  **MyPdfCanvas \*pdfCanvas;**  **QScrollBar \*scrollbar;**  **};**  **#endif // MAINWINDOW\_H**  （2）mainwindow.cpp  **#include "mainwindow.h"**  **MainWindow::MainWindow(QWidget \*parent) : QMainWindow(parent) {**  **pdfCanvas = new MyPdfCanvas(this);**  **scrollbar = new QScrollBar(Qt::Vertical, this);**  **setWidgetVisible(false, false);**  **connect(pdfCanvas, SIGNAL(pageChanged(int)), this, SLOT(onPageChange(int)));**  **connect(scrollbar, SIGNAL(valueChanged(int)), this, SLOT(onScrollBarValueChange()));**  **}**  **MainWindow::~MainWindow() {**  **}**  **void MainWindow::resizeEvent(QResizeEvent \*e) {**  **resizeCanvas();**  **}**  **bool MainWindow::setPdfPath(QString path) {**  **bool result = pdfCanvas->setPath(path);**  **if(result) {**  **int numPages = pdfCanvas->getNumPages();**  **if(numPages>1) {**  **scrollbar->setMaximum(numPages-1);**  **scrollbar->setValue(0);**  **}**  **pdfCanvas->setPage(0);**  **}**  **resizeCanvas();**  **return result;**  **}**  **void MainWindow::resizeCanvas() {**  **qDebug()<<"resize "<<this->rect()<<", "<<pdfCanvas->rect();**  **int numPages = pdfCanvas->getNumPages();**  **if(numPages == 1) {**  **pdfCanvas->setGeometry(this->rect());**  **setWidgetVisible(true, false);**  **} else if(numPages > 1) {**  **pdfCanvas->setGeometry(0, 0, this->width()-SCROLLBAR\_WIDTH, this->height());**  **scrollbar->setGeometry(this->width()-SCROLLBAR\_WIDTH, 0, this->width()-SCROLLBAR\_WIDTH, this->height());**  **setWidgetVisible(true, true);**  **} else {**  **//numPages <= 0**  **setWidgetVisible(false, false);**  **}**  **}**  **void MainWindow::setWidgetVisible(bool pdfCanvasVisible, bool scrollbarVisible) {**  **pdfCanvas->setVisible(pdfCanvasVisible);**  **scrollbar->setVisible(scrollbarVisible);**  **}**  **MainWindow::onPageChange(int currentPage) {**  **pdfCanvas->setPage(currentPage);**  **}**  **MainWindow::onScrollBarValueChange() {**  **pdfCanvas->setPage(scrollbar->value());**  **}**  6、调用方式  （1）main.cpp  **#include "mainwindow.h"**  **#include <QApplication>**  **int main(int argc, char \*argv[])**  **{**  **QApplication a(argc, argv);**  **MainWindow w;**  **w.resize(500, 500);**  **w.show();**  **QString path = "D://test.pdf";**  **w.setPdfPath(path);**  **w.setWindowTitle(path);**  **return a.exec();**  **}**  7、实际效果图  https://img-blog.csdn.net/20160608234904750?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  **更新于2016-08-03**  8、项目下载地址（使用当前最新的库poppler-0.45.0、poppler-0.39.0-win32）  <http://download.csdn.net/detail/chy555chy/9593364>  该项目在win7（Qt5.1）、win10（Qt5.7）下测试过了，均可正常运行。  下图为项目目录中的poppler文件夹（已经删去所有.cc文件），因为只用库和头文件，Qt便可隐式调用dll中的函数了。  https://img-blog.csdn.net/20160803095953734  更新于2016-08-22  你们评论中遇到的加载库的时候就奔溃现象我还真没遇到过。  下面是测试情况：  （1）当PDF文件未找到的情况，会输出错误日志，但是并不会崩溃。  https://img-blog.csdn.net/20160822103349779  （2）当路径中包含”中文“，且包含"空格"的情况，poppler是可以正常打开的。  https://img-blog.csdn.net/20160822103408067 |

#### 参考2：

<https://www.jqhtml.com/38123.html>

muPdf 源码下载 <https://github.com/ArtifexSoftware/mupdf/tags?after=1.15.0>

|  |  |  |
| --- | --- | --- |
| 基于MuPDF和Qt的PDF阅读器的开发一、引言 设备上要渲染显示PDF文档，类似Adobe Reader、福昕阅读器等软件。如何去实现？[Handling PDF](http://wiki.qt.io/Handling_PDF)这篇文档讨论介绍了在Qt应用程序中各种操作PDF的资料。从资料中看出，我们是要阅读渲染PDF文档到我们的设备中，关键还要能交叉编译移植到我们的设备上。在渲染显示方面整体给出三种方案：   * QtPDF（基于PDFium），不太好交叉编译。 * poppler-qt，与我们设备操作系统版本匹配的版本bug太多。 * muPDF，非常成熟，很多产品上都已使用，格式支持丰富，方便交叉编译。   最终选muPDF库来实现。 二、muPDF下载和编译 [muPDF官网](https://www.mupdf.com/)，[muPDF-1.12.0源码下载](https://www.mupdf.com/downloads/)。 1、交叉编译iMX6-ARM版本 修改Makerules   |  | | --- | | ...  157:  158: ifeq "$(OS)" "imx6-linux"  159: CCPATH = /opt/linaro-14.04/gcc-linaro-arm-linux-gnueabihf-4.8/bin  160: CC = $(CCPATH)/arm-linux-gnueabihf-gcc  161: LD = $(CCPATH)/arm-linux-gnueabihf-gcc  162: AR = $(CCPATH)/arm-linux-gnueabihf-ar  163: CROSSCOMPILE=yes  164: HAVE\_GLUT := no  165: HAVE\_PTHREAD := yes  166: SYS\_PTHREAD\_CFLAGS :=  167: SYS\_PTHREAD\_LIBS := -lpthread  168: HAVE\_X11 := yes  169: SYS\_X11\_CFLAGS += -I/opt/linaro-14.04/usr/include  170: SYS\_X11\_LIBS += -L/opt/linaro-14.04/usr/lib/arm-linux-gnueabihf -lX11 -lXext -lxcb -lXau -lXdmcp -lpthread  171: endif  172:  ... |   执行命令：make generate && make OS=imx6-linux HAVE\_GLUT=no -j32 最后源码目录的build/release/下生成四个库文件和多个二进制程序工具。libcurl.a libmupdf.a libmupdfthird.a libmuthreads.a mjsgen mujstest mupdf-x11 mupdf-x11-curl muraster mutool  使用的时候把四个库放到Qt工程里面使用，curl又依赖libcrypto.a库，所以去交叉编译工具链下找到libcrypto.a放进去。 2、编译win版本 用VS打开源码目录下的platform/win32/mupdf.sln,把所有工程中的属性--配置属性--C/C++--代码生成--运行库改为多线程调试DLL（/MDd） 编译mupdf生成三个库libmupdf.lib、libthirdparty.lib，编译libresources工程，生成libresources.lib，目前项目需要这三个库就够了。 三、使用muPDF库 其实重点在于如何使用muPDF库，这个库的API实在是难用，也没有帮助文档，只能看它提供的例子或这小工具的源码来学习。 先看工程目录结构：   |  | | --- | | main.cpp  mainwindow.h  mainwindow.cpp  mupdfdocument.h  mupdfdocument\_p.h  mupdfdocument.cpp  mupdfpage.h  mupdfpage\_p.h  mupdfpage.cpp  pagerender.h  pagerender.cpp  sequentialpagewidget.h  sequentialpagewidget.cpp |   **mupdfdocument**：主要实现加载文档，提供文档信息功能。对mupdf接口封装主要实现在mupdfdocument\_p里面； **mupdfpage和mupdfpage\_p**：主要实现渲染单个页面，输出QImage。主要难度在reanderImage函数； **pagerender**：主要是借助QThread线程来渲染单个PDF页面； **sequentialpagewidget**：实现对PDF文件显示的封装，如何做到高效显示，而又不占用太多资源？如何实现跳页？放大？缩小？滚动页面？非常小的屏幕下实现页码的正确显示？这里面都有所体现。  整个工程，都是短时间完成的，并且没有时间对muPDF做深入研究，所以肯定还有很多不足之处，还请见谅。整个工程代码没什么大的难度，难点在如何封装使用muPDF的API，以及如何使用Qt的控件更有效率的显示出来。  这是基于VS2013的PDFReader的工程。 资源工程下载：<https://github.com/KikyoShaw/MuPDF-Qt>  PDF Reader  原文地址：https://www.jianshu.com/p/d91b26a9ae05 |

已经下载的资源：<MuPDF-Qt-master>

# 19.C++17 自制脚本语言

#### 参考1：<两周自制脚本语言.pdf>

#### 参考2：[自制编程语言基于C语言 (郑钢) (z-lib.org).pdf](自制编程语言基于C语言%20(郑钢)%20(z-lib.org).pdf)

# 20.C++ 实现太阳系行星系统

#### 参考1：

GitHub：<https://github.com/ub1990228/solarsystem>

#### 参考2：

|  |
| --- |
| C++ 实现太阳系行星系统（OpenGL）基本[框架](https://so.csdn.net/so/search?q=%E6%A1%86%E6%9E%B6&spm=1001.2101.3001.7020)设计一、介绍 本次实验将使用 [OpenGL](https://so.csdn.net/so/search?q=OpenGL&spm=1001.2101.3001.7020) GLUT 编写一个简单的太阳系运行系统。  实验涉及的知识点   * C++ 语言基础 * 基本的 Makefile * 基本的 OOP 编程思想 * OpenGL GLUT 的结构基本使用   实验效果图  运行效果如图所示：  1-1-1   二、基础知识 认识 OpenGL 和 GLUT  OpenGL 包含了很多渲染函数，但是他们的设计目的是独立于任何窗口系统或操作系统的。因此，它自身并没有包含创建打开窗口或者从键盘或鼠标读取时间的函数，甚至连最基本的显示窗口的功能都没有，所以单纯只使用 OpenGL 是完全不可能创建一个完整的图形程序的。并且绝大多数程序都需要与用户进行交互（响应键盘鼠标等操作）。GLUT 则提供了这一便利。  GLUT 其实是 OpenGL Utility Toolkit 的缩写，它是一个处理 OpenGL 程序的工具库，主要负责处理与底层操作系统的调用及 I/O操作。使用 GLUT 可以屏蔽掉底层操作系统 GUI 实现上的一些细节，仅使用 GLUT 的 API 即可跨平台的创建应用程序窗口、处理鼠标键盘事件等等。  我们先在实验楼环境中安装 GLUT：  sudo apt-get update && sudo apt-get install freeglut3 freeglut3-dev  一个标准的 GLUT 程序结构如下代码所示：    **// 使用 GLUT 的基本头文件**  **#include <GL/glut.h>**    **// 创建图形窗口的基本宏**  **#define WINDOW\_X\_POS 50**  **#define WINDOW\_Y\_POS 50**  **#define WIDTH 700**  **#define HEIGHT 700**    **// 用于注册 GLUT 的回调**  **void onDisplay(void);**  **void onUpdate(void);**  **void onKeyboard(unsigned char key, int x, int y);**    **int main(int argc, char\* argv[]) {**    **// 对 GLUT 进行初始化，并处理所有的命令行参数**  **glutInit(&argc, argv);**  **// 这个函数指定了使用 RGBA 模式韩式颜色索引模式。另外还可以**  **// 指定是使用单缓冲还是双缓冲窗口。这里我们使用 RGBA 和 双缓冲窗口**  **glutInitDisplayMode(GLUT\_RGBA | GLUT\_DOUBLE);**  **// 设置窗口被创建时左上角位于屏幕上的位置**  **glutInitWindowPosition(WINDOW\_X\_POS, WINDOW\_Y\_POS);**  **// 设置窗口被创建时的宽高, 为了简便起见**  **glutInitWindowSize(WIDTH, HEIGHT);**  **// 创建一个窗口，输入的字符串为窗口的标题**  **glutCreateWindow("SolarSystem at Shiyanlou");**    **// glutDisplayFunc 的函数原型为 glutDisplayFunc(void (\*func)(void))**  **// 这是一个回调函数，每当 GLUT 确定一个窗口的内容需要更新显示的时候,**  **// glutDisplayFunc 注册的回调函数就会被执行.**  **//**  **// glutIdleFunc(void (\*func)(void)) 将指定一个函数，用于处理当事件循环**  **// 处于空闲的时候，就执行这个函数。这个回调函数接受一个函数指针作为它的唯一参数**  **//**  **// glutKeyboardFunc(void (\*func)(unsigned char key, int x, int y))**  **// 会将键盘上的键与一个函数关联，当这个键被按下或者释放时，函数就会调用**  **//**  **// 因此下面的三行实际上是在向 GLUT 注册关键的三个回调函数**  **glutDisplayFunc(onDisplay);**  **glutIdleFunc(onUpdate);**  **glutKeyboardFunc(onKeyboard);**    **glutMainLoop();**  **return 0;**    **}**  在 /home/shiyanlou/ 目录下新建 main.cpp 文件，并向其中写入如下代码：  **//**  **// main.cpp**  **// solarsystem**  **//**  **#include <GL/glut.h>**  **#include "solarsystem.hpp"**    **#define WINDOW\_X\_POS 50**  **#define WINDOW\_Y\_POS 50**  **#define WIDTH 700**  **#define HEIGHT 700**    **SolarSystem solarsystem;**    **void onDisplay(void) {**  **solarsystem.onDisplay();**  **}**  **void onUpdate(void) {**  **solarsystem.onUpdate();**  **}**  **void onKeyboard(unsigned char key, int x, int y) {**  **solarsystem.onKeyboard(key, x, y);**  **}**    **int main(int argc, char\* argv[]) {**  **glutInit(&argc, argv);**  **glutInitDisplayMode(GLUT\_RGBA | GLUT\_DOUBLE);**  **glutInitWindowPosition(WINDOW\_X\_POS, WINDOW\_Y\_POS);**  **glutCreateWindow("SolarSystem at Shiyanlou");**  **glutDisplayFunc(onDisplay);**  **glutIdleFunc(onUpdate);**  **glutKeyboardFunc(onKeyboard);**  **glutMainLoop();**  **return 0;**  **}**  **提示**  单缓冲，是将所有的绘图指令在窗口上执行，就是直接在窗口上绘图，这样的绘图效率是比较慢的，如果使用单缓冲，而电脑处理性能不够，屏幕会出现闪烁状。  双缓冲，会将绘图指令是在一个缓冲区完成，这里的绘图非常的快，在绘图指令完成之后，再通过交换指令把完成的图形立即显示在屏幕上，进而避免出现绘图的不完整，效率很高。  双缓冲则主要分为前台缓冲和后台缓冲，前台缓冲即我们说看到的屏幕，后台缓冲则维护内存中，对用户不可见。使用双缓冲时所有绘图操作都会在后台进行，当完成绘制后，才会将结果复制到屏幕上。  这么做的好处是，如果我们让绘制操作实时与显卡进行操作，当绘制任务复杂时，IO 操作同样会变得复杂，造成性能较低；而双缓冲只会在交换缓冲区时将绘制完成后的结果直接发送给显卡进行渲染，IO 显著降低。  在 OpenGL 中，推荐使用 GLfloat 来表示浮点数。 三、类设计 OOP 编程中首先要梳理清楚我们要处理的对象是什么。显然整个天体系统中，他们都是一颗星球(Star)，区别行星和恒星只需要通过它们是否具有父节点即可；其次，对于不同的星球而言，它们通常具有自身的材质，并且不同的材质会表现出自身是否发光，由此我们有了初步的对象模型。故我们将星球分为：普通能够自转并绕某个点公转的星球(Star), 具有特殊材质的星球(Planet), 能够发光的星球(LightPlanet)  此外，为了编程实现上的方便，我们要对现实世界的实际编程模型做一些前提假设：   1. 星球的运行轨道为圆形； 2. 自转速度保持相同； 3. 每次刷新绘制的时候假设时间经过了一天。   首先，我们可以考虑按下面的思路整个实现逻辑：   1. 初始化星球对象; 2. 初始化 OpenGL 引擎, 实现 onDraw 和 onUpdate; 3. 星球应该自己负责处理自己的属性、绕行关系、变换相关绘制，因此在设计星球的类时应该提供一个绘制 draw() 方法; 4. 星球也应该自己处理自己自转公转等更新显示的绘制，因此在设计星球类时候也应该提供一个更新方法 update(); 5. 在 onDraw() 中应调 用星球的 draw() 方法; 6. 在 onUpdate() 中调用星球的 update() 方法; 7. 在 onKeyboard() 键盘调整整个太阳系的显示.   进一步，对于每个星球而言，都具有如下的属性：   1. 颜色 color 2. 公转半径 radius 3. 自转速度 selfSpeed 4. 公转速度 speed 5. 距离太阳中心的距离 distance 6. 绕行的星球 parentStar 7. 当前的自转的角度 alphaSelf 8. 当前的公转角度 alpha   在 /home/shiyanlou/ 目录下新建 stars.hpp 文件，根据前面的分析，我们可以设计如下的类代码：  **class Star {**  **public:**  **// 星球的运行半径**  **GLfloat radius;**  **// 星球的公转、自传速度**  **GLfloat speed, selfSpeed;**  **// 星球的中心与父节点星球中心的距离**  **GLfloat distance;**  **// 星球的颜色**  **GLfloat rgbaColor[4];**    **// 父节点星球**  **Star\* parentStar;**    **// 构造函数，构造一颗星球时必须提供**  **// 旋转半径、旋转速度、自转速度、绕行(父节点)星球**  **Star(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parentStar);**  **// 对一般的星球的移动、旋转等活动进行绘制**  **void drawStar();**  **// 提供默认实现，负责调用 drawStar()**  **virtual void draw() { drawStar(); }**  **// 参数为每次刷新画面时的时间跨度**  **virtual void update(long timeSpan);**  **protected:**  **GLfloat alphaSelf, alpha;**  **};**  **class Planet : public Star {**  **public:**  **// 构造函数**  **Planet(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parentStar, GLfloat rgbColor[3]);**  **// 增加对具备自身材质的行星绘制材质**  **void drawPlanet();**  **// 继续向其子类开放重写功能**  **virtual void draw() { drawPlanet(); drawStar(); }**  **};**  **class LightPlanet : public Planet {**  **public:**  **LightPlanet(GLfloat Radius, GLfloat Distance,**  **GLfloat Speed, GLfloat SelfSpeed,**  **Star\* ParentStar, GLfloat rgbColor[]);**  **// 增加对提供光源的恒星绘制光照**  **void drawLight();**  **virtual void draw() { drawLight(); drawPlanet(); drawStar(); }**  **};**  **`**  此外，我们还需要考虑太阳系类的设计。在太阳系中，太阳系显然是由各个行星组成的；并且，对于太阳系而言，太阳系中行星运动后的视图刷新应该由太阳系来完成。据此太阳系成员变量应为包含行星的变量，成员函数应作为处理太阳系内的视图刷新及键盘响应等事件，所以，在 /home/shiyanlou/ 目录下新建 solarsystem.hpp 文件,在其中我们可以设计 SolarSystem 类：  **class SolarSystem {**    **public:**    **SolarSystem();**  **~SolarSystem();**    **void onDisplay();**  **void onUpdate();**  **void onKeyboard(unsigned char key, int x, int y);**    **private:**  **Star \*stars[STARS\_NUM];**    **// 定义观察视角的参数**  **GLdouble viewX, viewY, viewZ;**  **GLdouble centerX, centerY, centerZ;**  **GLdouble upX, upY, upZ;**  **};**  **提示**   1. 这里使用传统形式的数组来管理所有星球，而不是使用 C++中的 vector，是因为传统形式的数组就足够了 2. 在 OpenGL 中定义观察视角是一个较为复杂的概念，需要一定篇幅进行解释，我们先在此记下定义观察视角至少需要九个参数，我们将在下一节中具体实现时再详细讲解它们的作用。   最后我们还需要考虑一下基本的参数和变量设置。  在 SolarSystem 中，包括太阳在内一共有九颗星球（不包括冥王星），但是在我们所设计的 Star 类中，每一个 Star 对象都具有 Star 的属性，因此我们还可以额外实现这些星球的卫星，比如围绕地球运行的月球，据此我们一共考虑实现十个星球。于是我们可以设置如下枚举，用于索引一个数组中的星球：  **#define STARS\_NUM 10**  **enum STARS {**  **Sun, // 太阳**  **Mercury, // 水星**  **Venus, // 金星**  **Earth, // 地球**  **Moon, // 月球**  **Mars, // 火星**  **Jupiter, // 木星**  **Saturn, // 土星**  **Uranus, // 天王星**  **Neptune // 海王星**  **};**  **Star \* stars[STARS\_NUM];**  我们还假设了自转速度相同，使用一个宏来设置其速度，：  #define TIMEPAST 1  #define SELFROTATE 3  至此，将未实现的成员函数创建到对应的 .cpp 文件中，我们便完成了本节实验。 四、总结本节中的代码 我们来总结一下本节实验中需要完成的代码：  首先我们在 main.cpp 中创建了一个 SolarSystm，然后将显示刷新、空闲刷新 及键盘事件的处理交给了 glut。  **//**  **// main.cpp**  **// solarsystem**  **//**  **#include <GL/glut.h>**  **#include "solarsystem.hpp"**    **#define WINDOW\_X\_POS 50**  **#define WINDOW\_Y\_POS 50**  **#define WIDTH 700**  **#define HEIGHT 700**    **SolarSystem solarsystem;**    **void onDisplay(void) {**  **solarsystem.onDisplay();**  **}**  **void onUpdate(void) {**  **solarsystem.onUpdate();**  **}**  **void onKeyboard(unsigned char key, int x, int y) {**  **solarsystem.onKeyboard(key, x, y);**  **}**    **int main(int argc, char\* argv[]) {**  **glutInit(&argc, argv);**  **glutInitDisplayMode(GLUT\_RGBA | GLUT\_DOUBLE);**  **glutInitWindowPosition(WINDOW\_X\_POS, WINDOW\_Y\_POS);**  **glutCreateWindow("SolarSystem at Shiyanlou");**  **glutDisplayFunc(onDisplay);**  **glutIdleFunc(onUpdate);**  **glutKeyboardFunc(onKeyboard);**  **glutMainLoop();**  **return 0;**  **}**  其次，我们在 stars.hpp 中分别创建了 Star/Planet/LightPlanet 类。  //  **// stars.hpp**  **// solarsystem**  **//**  **//**  **#ifndef stars\_hpp**  **#define stars\_hpp**    **#include <GL/glut.h>**    **class Star {**  **public:**  **GLfloat radius;**  **GLfloat speed, selfSpeed;**  **GLfloat distance;**  **GLfloat rgbaColor[4];**    **Star\* parentStar;**    **Star(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parentStar);**  **void drawStar();**  **virtual void draw() { drawStar(); }**  **virtual void update(long timeSpan);**  **protected:**  **GLfloat alphaSelf, alpha;**  **};**    **class Planet : public Star {**  **public:**  **Planet(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parentStar, GLfloat rgbColor[3]);**  **void drawPlanet();**  **virtual void draw() { drawPlanet(); drawStar(); }**  **};**    **class LightPlanet : public Planet {**  **public:**  **LightPlanet(GLfloat Radius, GLfloat Distance,**  **GLfloat Speed, GLfloat SelfSpeed,**  **Star\* parentStar, GLfloat rgbColor[]);**  **void drawLight();**  **virtual void draw() { drawLight(); drawPlanet(); drawStar(); }**  **};**    **#endif /\* star\_hpp \*/**  在 /home/shiyanlou/ 目录下新建 stars.cpp 文件，在其中填写stars.hpp 中对应类的成员函数实现:  **//**  **// stars.cpp**  **// solarsystem**  **//**  **#include "stars.hpp"**    **#define PI 3.1415926535**    **Star::Star(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parentStar) {**  **// TODO:**  **}**    **void Star::drawStar() {**  **// TODO:**  **}**    **void Star::update(long timeSpan) {**  **// TODO:**  **}**      **Planet::Planet(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parentStar, GLfloat rgbColor[3]) :**  **Star(radius, distance, speed, selfSpeed, parentStar) {**  **// TODO:**  **}**    **void Planet::drawPlanet() {**  **// TODO:**  **}**    **LightPlanet::LightPlanet(GLfloat radius, GLfloat distance, GLfloat speed,**  **GLfloat selfSpeed, Star\* parentStar, GLfloat rgbColor[3]) :**  **Planet(radius, distance, speed, selfSpeed, parentStar, rgbColor) {**  **// TODO:**  **}**    **void LightPlanet::drawLight() {**  **// TODO:**  **}**  在 solarsystem.hpp 中设计了 SolarSystem 类：  **//**  **// solarsystem.hpp**  **// solarsystem**  **//**  **#include <GL/glut.h>**    **#include "stars.hpp"**    **#define STARS\_NUM 10**    **class SolarSystem {**    **public:**    **SolarSystem();**  **~SolarSystem();**    **void onDisplay();**  **void onUpdate();**  **void onKeyboard(unsigned char key, int x, int y);**    **private:**  **Star \*stars[STARS\_NUM];**    **// 定义观察视角的参数**  **GLdouble viewX, viewY, viewZ;**  **GLdouble centerX, centerY, centerZ;**  **GLdouble upX, upY, upZ;**  **};**  在 /home/shiyanlou/ 目录下新建 solarsystem.cpp 文件，并在其中实现对应的 solarsystem.hpp 中的成员函数:  **//**  **//**  **// solarsystem**  **//**  **#include "solarsystem.hpp"**    **#define TIMEPAST 1**  **#define SELFROTATE 3**    **enum STARS {Sun, Mercury, Venus, Earth, Moon,**  **Mars, Jupiter, Saturn, Uranus, Neptune};**    **void SolarSystem::onDisplay() {**  **// TODO:**  **}**  **void SolarSystem::onUpdate() {**  **// TODO:**  **}**  **void SolarSystem::onKeyboard(unsigned char key, int x, int y) {**  **// TODO:**  **}**  **SolarSystem::SolarSystem() {**  **// TODO:**    **}**  **SolarSystem::~SolarSystem() {**  **// TODO:**  **}**  在 /home/shiyanlou/ 目录下新建Makefile 文件，并向其中添加如下代码：  CXX = g++  EXEC = solarsystem  SOURCES = main.cpp stars.cpp solarsystem.cpp  OBJECTS = main.o stars.o solarsystem.o  LDFLAGS = -lglut -lGL -lGLU    all :  $(CXX) $(SOURCES) $(LDFLAGS) -o $(EXEC)    clean:  rm -f $(EXEC) \*.gdb \*.o  **提示**  书写编译命令时注意 -lglut -lGLU -lGL 的位置  这是因为 g++ 编译器中 -l 选项用法有点特殊： 例如： foo1.cpp -lz foo2.cpp 如果目标文件 foo2.cpp 应用了库 z 中的函数，那么 这些函数不会被直接加载。而如果 foo1.o 使用了 z 库中的函数则不会出现任何编译错误。  换句话说，整个链接的过程是自左向右的。当 foo1.cpp 中遇到无法解析的函数符号时，会查找右边的链接库，当 发现选项 z 时，在 z 中查找然后发现函数，进而顺利完 成链接。  所以 -l 选项的库应该存在于所有编译文件的右边。  更多编译器细节请学习：g++/gdb 使用技巧  最后，在终端中运行：  make && ./solarsystem  运行结果如图所示：  1-4-1    可以看到，窗口已经创建出来了，但是之所以什么都没有(显示的是窗口背后的内容)，是因为我们还没有实现窗口中图形的刷新机制，在下一实验中我们再继续实现未完成的代码，让整个太阳系系统运行起来。 五、拓展阅读  1. D Shreiner. OpenGL 编程指南. 人民邮电出版社, 2005.   这本书详细介绍了 OpenGL 编程相关的方方面面，被誉为 『OpenGL 红宝书』。 编码实现一、实验介绍 本节实验我们将对上一节实验中所设计的基本框架进行详细的实现。  涉及的知识点   * OpenGL 的矩阵模式 * OpenGL 常用的图像绘制接口 * OpenGL 中的视角 * OpenGL 的光照实现  二、基础知识 这一小节主要讲解与本节实验相关的基础知识。 2.1 OpenGL 中的矩阵概念 在线性代数中，我们知道了矩阵的概念，但实际上我们对矩阵的具体作用和认识并没有体会到多少。矩阵到底是什么？  首先我们看这样一个式子：  x = Ab  其中 A 为矩阵，x,b 为向量。  **观点一：**  x 和 b 都是 我们三维空间中的一个向量，那么 A 做了什么事情？———— A 把 b 变成了（变换到了） x。在这样的观点下，矩阵 A 可以被理解为一种变换。  再来看另一个式子：  Ax = By  其中 A,B 为矩阵, x,y为向量。  **观点二：**  对于两个不同的向量x,y来说，它们本质上是同一个东西，因为它们只需要乘以矩阵A,B就可以相等。在这样的观点下，矩阵 A 可以被理解为一种坐标系。换句话说，向量本身是独一无二的，但是我们因为要描述它，所以定义了一个坐标系。由于坐标系取得不同，向量的坐标也会发生变化，即对于同一个向量来说，在不同的坐标系下有着不同的坐标。 矩阵 A 恰好描述了一个坐标系，而 矩阵B 也描述了一个坐标系，这两个坐标系作用到 x,y上之后，得到了相同的结果，也即 x,y本质是同一个向量，只不过他们有着不同的坐标系。  综合上述两个观点，所以矩阵的本质是：**描述运动**。  于是在 OpenGL 内部有一个负责绘制变换的矩阵，这就是 OpenGL 中的矩阵模式。  正如前面所说，矩阵既能够描述运动的变换，也能够描述某个物体所处的坐标系，因此在处理不同的操作时，我们要给 OpenGL 设置不同的矩阵模式，这就需要用到   1. glMatrixMode()   这个函数接受三个不同的模式：GL\_PROJECTION 投影, GL\_MODELVIEW 模型视图, GL\_TEXTURE 纹理。  GL\_PROJECTION 会向 OpenGL 声明将进行投影操作，会把物体头投影到一个平面上。开启这个模式后要使用glLoadIdentity()把矩阵设置为单位矩阵，而后的操作如可以通过 gluPerspective 设置视景（在下面介绍完 OpenGL 中视角的概念后我们再详细说明这个函数的功能）。  GL\_MODELVIEW 会向 OpenGL 声明接下来的语句将描绘一个以模型为基础的操作，比如设置摄像机视角，开启后同样需要设置 OpenGL 的矩阵模式为单位矩阵。  GL\_TEXTURE 则是进行纹理相关的操作，我们暂时用不到。  如果对矩阵的概念不熟悉，可以直接将 glMatrixMode 理解为针对 OpenGL 申明接下来要做的事情。我们在绘制、旋转对象之前，一定要通过 glPushMatrix 保存当前的矩阵环境，否则会出现莫名其妙的绘制错误。 2.2 常用的 OpenGL 图像绘制 API OpenGL 中提供了很多常用的与图形绘制相关的 API，这里我们挑几个常用的进行简单介绍，并在接下来的代码中进行使用：   * **glEnable(GLenum cap)**: 这个函数用于激活 OpenGL 中提供的各种功能，传入的参数 cap 是OpenGL 内部的宏，提供诸如光源、雾化、抖动等效果； * **glPushMatrix()** 和 **glPopMatrix()**: 将当前矩阵保存到堆栈栈顶（保存当前矩阵）； * **glRotatef(alpha, x, y, z)**: 表示当前图形沿 (x,y,z) 逆时针旋转 alpha 度; * **glTranslatef(distance, x, y)**: 表示当前图形沿 (x,y) 方向平移 distance 距离; * **glutSolidSphere(GLdouble radius , GLint slices , GLint stacks)**: 绘制球体， radius 为半径，slices 为经线条数, stacks 为纬线条数; * **glBegin()** 和 **glEnd()**：当我们要进行图形绘制时，需要在开始绘制前和绘制完成后调用这两个函数，glBegin()指定了绘制图形的类型，例如 GL\_POINTS 表示绘制点、 GL\_LINES 表示绘制依次画出的点及他们之间的连线、GL\_TRIANGLES 则是在每绘制的三个点中完成一个三角形、GL\_POLYGON 则是绘制一个从第一个点到第 n 个点的多边形，等等。例如当我们需要绘制一个圆时可以边很多的多边形来模拟：   // r 是半径，n 是边数  glBegin(GL\_POLYGON);  for(i=0; i<n; ++i)  glVertex2f(r\*cos(2\*PI/n\*i), r\*sin(2\*PI/n\*i));  glEnd(); 2.3 OpenGL 里的视角坐标 上一节中，我们在 SolarSystem类中定义了九个成员变量  GLdouble viewX, viewY, viewZ;  GLdouble centerX, centerY, centerZ;  GLdouble upX, upY, upZ;  为了理解这九个变量，我们首先需要树立起 OpenGL 三维编程中摄像机视角的概念。  想象平时我们观看电影的画面其实都是由摄像机所在的视角拍摄完成的，因此 OpenGL 中也有类似的概念。如果我们把摄像机想象成我们自己的头，那么：   1. viewX, viewY, viewZ 就相当于头（摄像机）在 OpenGL 世界坐标中的坐标位置; 2. centerX, centerY, centerZ 则相当于头所看（摄像机所拍）物体的坐标位置; 3. upX, upY, upZ 则相当于头顶（摄像机顶部）朝上的方向向量（因为我们可以歪着头观察一个物体）。   至此，你便有了 OpenGL 中坐标系的概念。  我们约定本次实验的初始视角在 (x, -x, x) 处，则即有：  **#define REST 700**  **#define REST\_Y (-REST)**  **#define REST\_Z (REST)**  所观察物体（太阳）的位置在 (0,0,0)，则在SolarSystem类中的构造函数将视角初始化为：  **viewX = 0;**  **viewY = REST\_Y;**  **viewZ = REST\_Z;**  **centerX = centerY = centerZ = 0;**  **upX = upY = 0;**  **upZ = 1;**  则可以通过 gluLookAt 函数来设置视角的九个参数：  gluLookAt(viewX, viewY, viewZ, centerX, centerY, centerZ, upX, upY, upZ);  然后我们再来看 gluPerspective(GLdouble fovy,GLdouble aspect,GLdouble zNear,GLdouble zFar)。  这个函数会创建一个对称的透视型视景体，在使用这个函数前需要将 OpenGL 的矩阵模式设置为 GL\_PROJECTION。  如下图所示:    2-2.3-1  在窗口中的画面是通过摄像机来捕捉的，捕捉的实际内容就是远平面上的内容，显示的内容则是近平面上的内容，所以，这个函数需要四个参数：  1.第一个参数为视角的大小 2.第二个参数为实际窗口的横纵比，如图中 aspect=w/h 3.第三个参数为近平面距离 4.第四个参数则为远平面距离 2.4 OpenGL 里的光照效果 1. 基本概念  OpenGL 在处理光照时将光照系统分为了三个部分：光源、材质、光照环境。  顾名思义，光源就是光的来源，比如太阳； 材质这是指接受光照的各种物体的表面，比如太阳系中除太阳以外的行星和卫星都是这里所指的材质； 光照环境则是一些额外的参数，他们讲应将最终得到的光照面积，比如光线通常会经过多次反射，这时制定一个『环境亮度』的参数，可以使得最后的画面接近真实情况。  物理学里，平行光射入光滑平面上后所得的反射光依然是平行光，这种反射被叫做『镜面反射』；而对于不光滑的平面造成的反射，便就是所谓的『漫反射』。  2-2.4-1    2. 光源  在 OpenGL 里要实现光照系统，我们首先需要做的就是设置光源。值得一提的是，OpenGL 内只支持有限数量的光源（八个）分别使用 GL\_LIGHT0 至 GL\_LIGHT7 这八个宏来表示。通过 glEnable 函数来启用， glDisable 函数来禁用。例如：glEnable(GL\_LIGHT0);  设置光源位置则需要使用 glMaterialfv进行设置，例如：  **GLfloat light\_position[] = {0.0f, 0.0f, 0.0f, 1.0f};**  **glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position); // 指定零号光源的位置**  这里的位置由四个值来表示，(x,y,z,w) 其中当 w为0时，表示该光源位于无限远，而 x,y,z 便指定了这个无限远光源的方向； 当 w 不为0时，表示 位置性光源，其位置为 (x/w, y/w, z/w) 。  3. 材质  设置一个物体的材质一般有五个属性需要设置：   1. 多次反射后追踪在环境中遗留的光照强度; 2. 漫反射后的光照强度; 3. 镜面反射后的光照强度; 4. OpenGL 中不发光物体发出的微弱且不影像其他物体的光照强度; 5. 镜面指数，指越小，表示材质越粗糙，点光源发射的光线照射后，产生较大的亮点；相反，值越大，材质越像镜面，产生较小的亮点。   设置材质 OpenGL 提供了两个版本的函数：  **void glMaterialf(GLenum face, GLenum pname, TYPE param);**  **void glMaterialfv(GLenum face, GLenum pname, TYPE \*param);**  其差异在于，镜面指数只需要设置一个数值，这时只需要使用 glMaterialf；而其他的材质设置都需要设置多个值，这是需要使用数组进行设置，使用带指针向量参数的版本 glMaterialfv，例如：    **GLfloat mat\_ambient[] = {0.0f, 0.0f, 0.5f, 1.0f};**  **GLfloat mat\_diffuse[] = {0.0f, 0.0f, 0.5f, 1.0f};**  **GLfloat mat\_specular[] = {0.0f, 0.0f, 1.0f, 1.0f};**  **GLfloat mat\_emission[] = {0.5f, 0.5f, 0.5f, 0.5f};**  **GLfloat mat\_shininess = 90.0f;**  **glMaterialfv(GL\_FRONT, GL\_AMBIENT, mat\_ambient);**  **glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse);**  **glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular);**  **glMaterialfv(GL\_FRONT, GL\_EMISSION, mat\_emission);**  **glMaterialf (GL\_FRONT, GL\_SHININESS, mat\_shininess);**  4. 光照环境  OpenGL 是默认关闭光照处理的，要打开光照处理功能，需要使用 GL\_LIGHTING 宏来激活，即 glEnable(GL\_LIGHTING);。 三、实现 本节将通过实践操作，带领大家实现太阳系行星系统。 3.1 行星的绘制 行星在绘制时，首先要考虑自身的公转角度和自转角度，因此我们可以先在 /home/shiyanlou/stars.cpp 文件中实现星球类的Star::update(long timeSpan) 成员函数：  **void Star::update(long timeSpan) {**  **alpha += timeSpan \* speed; // 更新角度**  **alphaSelf += selfSpeed; // 更新自转角度**  **}**  在完成公转和自转角度的更新后，我们便可以根据参数绘制具体的星球了：  **void Star::drawStar() {**    **glEnable(GL\_LINE\_SMOOTH);**  **glEnable(GL\_BLEND);**    **int n = 1440;**    **// 保存 OpenGL 当前的矩阵环境**  **glPushMatrix();**  **{**  **// 公转**    **// 如果是行星，且距离不为0，那么 且向原点平移一个半径**  **// 这部分用于处理卫星**  **if (parentStar != 0 && parentStar->distance > 0) {**  **//将绘制的图形沿 z 轴旋转 alpha**  **glRotatef(parentStar->alpha, 0, 0, 1);**  **// x 轴方向上平移 distance , y,z 方向不变**  **glTranslatef(parentStar->distance, 0.0, 0.0);**  **}**  **// 绘制运行轨道**  **glBegin(GL\_LINES);**  **for(int i=0; i<n; ++i)**  **glVertex2f(distance \* cos(2 \* PI \* i / n),**  **distance \* sin(2 \* PI \* i / n));**  **glEnd();**  **// 绕 z 轴旋转 alpha**  **glRotatef(alpha, 0, 0, 1);**  **// x 轴方向平移 distance, y,z 方向不变**  **glTranslatef(distance, 0.0, 0.0);**    **// 自转**  **glRotatef(alphaSelf, 0, 0, 1);**    **// 绘制行星颜色**  **glColor3f(rgbaColor[0], rgbaColor[1], rgbaColor[2]);**  **glutSolidSphere(radius, 40, 32);**  **}**  **// 恢复绘制前的矩阵环境**  **glPopMatrix();**    **}**  这里用到了 sin() 和 cos() 函数，需要引入 #include<cmath> 3.2 光照的绘制 对于 Planet 类而言，属于不发光的星球，我们要绘制它的光照效果，在 /home/shiyanlou/stars.cpp 文件中添加如下代码：  **void Planet::drawPlanet() {**  **GLfloat mat\_ambient[] = {0.0f, 0.0f, 0.5f, 1.0f};**  **GLfloat mat\_diffuse[] = {0.0f, 0.0f, 0.5f, 1.0f};**  **GLfloat mat\_specular[] = {0.0f, 0.0f, 1.0f, 1.0f};**  **GLfloat mat\_emission[] = {rgbaColor[0], rgbaColor[1], rgbaColor[2], rgbaColor[3]};**  **GLfloat mat\_shininess = 90.0f;**    **glMaterialfv(GL\_FRONT, GL\_AMBIENT, mat\_ambient);**  **glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse);**  **glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular);**  **glMaterialfv(GL\_FRONT, GL\_EMISSION, mat\_emission);**  **glMaterialf (GL\_FRONT, GL\_SHININESS, mat\_shininess);**  **}**  而对于 LightPlanet 类来说，属于发光的星球，所以我们不但要设置其光照材质，还要设置其光源位置：  **void LightPlanet::drawLight() {**    **GLfloat light\_position[] = {0.0f, 0.0f, 0.0f, 1.0f};**  **GLfloat light\_ambient[] = {0.0f, 0.0f, 0.0f, 1.0f};**  **GLfloat light\_diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};**  **GLfloat light\_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};**  **glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position); // 指定零号光源的位置**  **glLightfv(GL\_LIGHT0, GL\_AMBIENT, light\_ambient); // 表示各种光线照射到该材质上，经过很多次反射后追踪遗留在环境中的光线强度**  **glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse); // 漫反射后的光照强度**  **glLightfv(GL\_LIGHT0, GL\_SPECULAR, light\_specular); // 镜面反射后的光照强度**    **}** 3.3 实现窗口的绘制 在上一节实验中我们提到过 glutDisplayFunc 、glutIdleFunc 这两个处理图像显示的最重要的函数，glutDisplayFunc 会在 GLUT 确定窗口内容需要更新的时候将回调函数执行，glutIdleFunc则是处理当事件循环空闲时的回调。  我们要实现整个太阳系运动起来，就应该考虑在什么时候更新行星的位置，在什么时候刷新视图。  显然，glutDisplayFunc 应该专注于负责刷新视图显示，当事件空闲时，我们便可以开始更新星球的位置，当位置更新完毕后，再调用视图刷新函数进行刷新。  因此，我们可以先实现 glutDisplayFunc中调用的 SolarSystem 类中的成员函数 SolarSystem::onUpdate()：  **#define TIMEPAST 1 // 假设每次更新都经过了一天**  **void SolarSystem::onUpdate() {**    **for (int i=0; i<STARS\_NUM; i++)**  **stars[i]->update(TIMEPAST); // 更新星球的位置**    **this->onDisplay(); // 刷新显示**  **}**  其次，对于显示视图的刷新则是实现SolarSystem::onDisplay()：  **void SolarSystem::onDisplay() {**    **// 清除 viewport 缓冲区**  **glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);**  **// 清空并设置颜色缓存**  **glClearColor(.7f, .7f, .7f, .1f);**  **// 指定当前矩阵为投影矩阵**  **glMatrixMode(GL\_PROJECTION);**  **// 将指定的矩阵指定为单位矩阵**  **glLoadIdentity();**  **// 指定当前的观察视景体**  **gluPerspective(75.0f, 1.0f, 1.0f, 40000000);**  **// 指定当前矩阵为视景矩阵堆栈应用术后的矩阵操作**  **glMatrixMode(GL\_MODELVIEW);**  **// 指定当前的矩阵为单位矩阵**  **glLoadIdentity();**  **// 定义视图矩阵，并与当前矩阵相乘**  **gluLookAt(viewX, viewY, viewZ, centerX, centerY, centerZ, upX, upY, upZ);**    **// 设置第一个光源(0号光源)**  **glEnable(GL\_LIGHT0);**  **// 启用光源**  **glEnable(GL\_LIGHTING);**  **// 启用深度测试，根据坐标的远近自动隐藏被遮住的图形**  **glEnable(GL\_DEPTH\_TEST);**    **// 绘制星球**  **for (int i=0; i<STARS\_NUM; i++)**  **stars[i]->draw();**    **// 我们在 main 函数中初始化显示模式时使用了 GLUT\_DOUBLE**  **// 需要使用 glutSwapBuffers 在绘制结束后实现双缓冲的缓冲区交换**  **glutSwapBuffers();**  **}** 3.4 类的构造函数和析构函数 stars.hpp 中所定义类的构造函数需要对类中的成员变量进行初始化，相对较为简单，甚至可以使用默认析构，因此这部分请读者自行实现这些构造函数：  **Star::Star(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parent);**  **Planet::Planet(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parent, GLfloat rgbColor[3]);**  **LightPlanet::LightPlanet(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parent, GLfloat rgbColor[3]);**  **提示**：注意在初始化速度变量时将其转化为角速度  其转换公式为：alpha\_speed = 360/speed  对于solarsystem.cpp的构造函数，我们需要对所有的星球进行初始化，这里为了方便起见我们先给出适当的行星之间的参数：  **// 公转半径**  **#define SUN\_RADIUS 48.74**  **#define MER\_RADIUS 7.32**  **#define VEN\_RADIUS 18.15**  **#define EAR\_RADIUS 19.13**  **#define MOO\_RADIUS 6.15**  **#define MAR\_RADIUS 10.19**  **#define JUP\_RADIUS 42.90**  **#define SAT\_RADIUS 36.16**  **#define URA\_RADIUS 25.56**  **#define NEP\_RADIUS 24.78**    **// 距太阳的距离**  **#define MER\_DIS 62.06**  **#define VEN\_DIS 115.56**  **#define EAR\_DIS 168.00**  **#define MOO\_DIS 26.01**  **#define MAR\_DIS 228.00**  **#define JUP\_DIS 333.40**  **#define SAT\_DIS 428.10**  **#define URA\_DIS 848.00**  **#define NEP\_DIS 949.10**    **// 运动速度**  **#define MER\_SPEED 87.0**  **#define VEN\_SPEED 225.0**  **#define EAR\_SPEED 365.0**  **#define MOO\_SPEED 30.0**  **#define MAR\_SPEED 687.0**  **#define JUP\_SPEED 1298.4**  **#define SAT\_SPEED 3225.6**  **#define URA\_SPEED 3066.4**  **#define NEP\_SPEED 6014.8**    **// 自转速度**  **#define SELFROTATE 3**    **// 为了方便操作数组，定义一个设置多为数组的宏**  **#define SET\_VALUE\_3(name, value0, value1, value2) \**  **((name)[0])=(value0), ((name)[1])=(value1), ((name)[2])=(value2)**    **// 在上一节实验中我们定义了星球的枚举**  **enum STARS {Sun, Mercury, Venus, Earth, Moon,**  **Mars, Jupiter, Saturn, Uranus, Neptune};**  **提示**  我们在这里定义了一个 SET\_VALUE\_3 的宏，读者或许会认为我们可以编写一个函数来达到快速设置的目的  事实上，宏会在编译过程就完成整体的替换工作，而定义函数  这需要在调用期间进行函数的堆栈操作，性能远不及编译过程就完成的宏处理  因此，使用宏会变得更加高效  但是值得注意的是，虽然宏能够变得更加高效，但过分的滥用则会造成代码的丑陋及弱可读性，而适当的使用宏则是可以提倡的  因此我们可以实现SolarSystem类的构造函数，其中星球的颜色是随机选取的，读者可以自行更改星球的颜色：  **SolarSystem::SolarSystem() {**    **// 定义视角，在前面我们已经讨论过视角的初始化了**  **viewX = 0;**  **viewY = REST\_Y;**  **viewZ = REST\_Z;**  **centerX = centerY = centerZ = 0;**  **upX = upY = 0;**  **upZ = 1;**    **// 太阳**  **GLfloat rgbColor[3] = {1, 0, 0};**  **stars[Sun] = new LightPlanet(SUN\_RADIUS, 0, 0, SELFROTATE, 0, rgbColor);**  **// 水星**  **SET\_VALUE\_3(rgbColor, .2, .2, .5);**  **stars[Mercury] = new Planet(MER\_RADIUS, MER\_DIS, MER\_SPEED, SELFROTATE, stars[Sun], rgbColor);**  **// 金星**  **SET\_VALUE\_3(rgbColor, 1, .7, 0);**  **stars[Venus] = new Planet(VEN\_RADIUS, VEN\_DIS, VEN\_SPEED, SELFROTATE, stars[Sun], rgbColor);**  **// 地球**  **SET\_VALUE\_3(rgbColor, 0, 1, 0);**  **stars[Earth] = new Planet(EAR\_RADIUS, EAR\_DIS, EAR\_SPEED, SELFROTATE, stars[Sun], rgbColor);**  **// 月亮**  **SET\_VALUE\_3(rgbColor, 1, 1, 0);**  **stars[Moon] = new Planet(MOO\_RADIUS, MOO\_DIS, MOO\_SPEED, SELFROTATE, stars[Earth], rgbColor);**  **// 火星**  **SET\_VALUE\_3(rgbColor, 1, .5, .5);**  **stars[Mars] = new Planet(MAR\_RADIUS, MAR\_DIS, MAR\_SPEED, SELFROTATE, stars[Sun], rgbColor);**  **// 木星**  **SET\_VALUE\_3(rgbColor, 1, 1, .5);**  **stars[Jupiter] = new Planet(JUP\_RADIUS, JUP\_DIS, JUP\_SPEED, SELFROTATE, stars[Sun], rgbColor);**  **// 土星**  **SET\_VALUE\_3(rgbColor, .5, 1, .5);**  **stars[Saturn] = new Planet(SAT\_RADIUS, SAT\_DIS, SAT\_SPEED, SELFROTATE, stars[Sun], rgbColor);**  **// 天王星**  **SET\_VALUE\_3(rgbColor, .4, .4, .4);**  **stars[Uranus] = new Planet(URA\_RADIUS, URA\_DIS, URA\_SPEED, SELFROTATE, stars[Sun], rgbColor);**  **// 海王星**  **SET\_VALUE\_3(rgbColor, .5, .5, 1);**  **stars[Neptune] = new Planet(NEP\_RADIUS, NEP\_DIS, NEP\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **}**  此外，不要忘了在析构函数中释放申请的内存：  **SolarSystem::~SolarSystem() {**  **for(int i = 0; i<STARS\_NUM; i++)**  **delete stars[i];**  **}** 3.5 键盘按键变换视角的实现 我们不妨用键盘上的 w,a,s,d,x 五个键来控制视角的变换，并使用 r 键来复位视角。首先我们要确定一次按键后视角的变化大小，这里我们先定义一个宏 OFFSET。然后通过传入的 key 来判断用户的按键行为  **#define OFFSET 20**  **void SolarSystem::onKeyboard(unsigned char key, int x, int y) {**    **switch (key) {**  **case 'w': viewY += OFFSET; break; // 摄像机Y 轴位置增加 OFFSET**  **case 's': viewZ += OFFSET; break;**  **case 'S': viewZ -= OFFSET; break;**  **case 'a': viewX -= OFFSET; break;**  **case 'd': viewX += OFFSET; break;**  **case 'x': viewY -= OFFSET; break;**  **case 'r':**  **viewX = 0; viewY = REST\_Y; viewZ = REST\_Z;**  **centerX = centerY = centerZ = 0;**  **upX = upY = 0; upZ = 1;**  **break;**  **case 27: exit(0); break;**  **default: break;**  **}**    **}** 四、总结本节中的代码(供参考) 本节实验中我们主要实现了 stars.cpp 和 solarsystem.cpp 这两个文件中的代码。  stars.cpp 的代码如下：  **//**  **// star.cpp**  **// solarsystem**  **//**    **#include "stars.hpp"**  **#include <cmath>**    **#define PI 3.1415926535**    **Star::Star(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parent) {**  **this->radius = radius;**  **this->selfSpeed = selfSpeed;**  **this->alphaSelf = this->alpha = 0;**  **this->distance = distance;**    **for (int i = 0; i < 4; i++)**  **this->rgbaColor[i] = 1.0f;**    **this->parentStar = parent;**  **if (speed > 0)**  **this->speed = 360.0f / speed;**  **else**  **this->speed = 0.0f;**  **}**    **void Star::drawStar() {**    **glEnable(GL\_LINE\_SMOOTH);**  **glEnable(GL\_BLEND);**    **int n = 1440;**    **glPushMatrix();**  **{**  **if (parentStar != 0 && parentStar->distance > 0) {**  **glRotatef(parentStar->alpha, 0, 0, 1);**  **glTranslatef(parentStar->distance, 0.0, 0.0);**  **}**  **glBegin(GL\_LINES);**  **for(int i=0; i<n; ++i)**  **glVertex2f(distance \* cos(2 \* PI \* i / n),**  **distance \* sin(2 \* PI \* i / n));**  **glEnd();**  **glRotatef(alpha, 0, 0, 1);**  **glTranslatef(distance, 0.0, 0.0);**    **glRotatef(alphaSelf, 0, 0, 1);**    **glColor3f(rgbaColor[0], rgbaColor[1], rgbaColor[2]);**  **glutSolidSphere(radius, 40, 32);**  **}**  **glPopMatrix();**    **}**    **void Star::update(long timeSpan) {**  **alpha += timeSpan \* speed;**  **alphaSelf += selfSpeed;**  **}**      **Planet::Planet(GLfloat radius, GLfloat distance,**  **GLfloat speed, GLfloat selfSpeed,**  **Star\* parent, GLfloat rgbColor[3]) :**  **Star(radius, distance, speed, selfSpeed, parent) {**  **rgbaColor[0] = rgbColor[0];**  **rgbaColor[1] = rgbColor[1];**  **rgbaColor[2] = rgbColor[2];**  **rgbaColor[3] = 1.0f;**  **}**    **void Planet::drawPlanet() {**  **GLfloat mat\_ambient[] = {0.0f, 0.0f, 0.5f, 1.0f};**  **GLfloat mat\_diffuse[] = {0.0f, 0.0f, 0.5f, 1.0f};**  **GLfloat mat\_specular[] = {0.0f, 0.0f, 1.0f, 1.0f};**  **GLfloat mat\_emission[] = {rgbaColor[0], rgbaColor[1], rgbaColor[2], rgbaColor[3]};**  **GLfloat mat\_shininess = 90.0f;**    **glMaterialfv(GL\_FRONT, GL\_AMBIENT, mat\_ambient);**  **glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse);**  **glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular);**  **glMaterialfv(GL\_FRONT, GL\_EMISSION, mat\_emission);**  **glMaterialf (GL\_FRONT, GL\_SHININESS, mat\_shininess);**  **}**    **LightPlanet::LightPlanet(GLfloat radius, GLfloat distance, GLfloat speed,**  **GLfloat selfSpeed, Star\* parent, GLfloat rgbColor[3]) :**  **Planet(radius, distance, speed, selfSpeed, parent, rgbColor) {**  **;**  **}**    **void LightPlanet::drawLight() {**    **GLfloat light\_position[] = {0.0f, 0.0f, 0.0f, 1.0f};**  **GLfloat light\_ambient[] = {0.0f, 0.0f, 0.0f, 1.0f};**  **GLfloat light\_diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};**  **GLfloat light\_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};**  **glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);**  **glLightfv(GL\_LIGHT0, GL\_AMBIENT, light\_ambient);**  **glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse);**  **glLightfv(GL\_LIGHT0, GL\_SPECULAR, light\_specular);**    **}**    solarsystem.cpp 的代码如下：  **//**  **// solarsystem.cpp**  **// solarsystem**  **//**    **#include "solarsystem.hpp"**    **#define REST 700**  **#define REST\_Z (REST)**  **#define REST\_Y (-REST)**    **void SolarSystem::onDisplay() {**    **glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);**  **glClearColor(.7f, .7f, .7f, .1f);**  **glMatrixMode(GL\_PROJECTION);**  **glLoadIdentity();**  **gluPerspective(75.0f, 1.0f, 1.0f, 40000000);**  **glMatrixMode(GL\_MODELVIEW);**  **glLoadIdentity();**  **gluLookAt(viewX, viewY, viewZ, centerX, centerY, centerZ, upX, upY, upZ);**    **glEnable(GL\_LIGHT0);**  **glEnable(GL\_LIGHTING);**  **glEnable(GL\_DEPTH\_TEST);**    **for (int i=0; i<STARS\_NUM; i++)**  **stars[i]->draw();**    **glutSwapBuffers();**  **}**    **#define TIMEPAST 1**  **void SolarSystem::onUpdate() {**    **for (int i=0; i<STARS\_NUM; i++)**  **stars[i]->update(TIMEPAST);**    **this->onDisplay();**  **}**    **#define OFFSET 20**  **void SolarSystem::onKeyboard(unsigned char key, int x, int y) {**    **switch (key) {**  **case 'w': viewY += OFFSET; break;**  **case 's': viewZ += OFFSET; break;**  **case 'S': viewZ -= OFFSET; break;**  **case 'a': viewX -= OFFSET; break;**  **case 'd': viewX += OFFSET; break;**  **case 'x': viewY -= OFFSET; break;**  **case 'r':**  **viewX = 0; viewY = REST\_Y; viewZ = REST\_Z;**  **centerX = centerY = centerZ = 0;**  **upX = upY = 0; upZ = 1;**  **break;**  **case 27: exit(0); break;**  **default: break;**  **}**    **}**    **#define SUN\_RADIUS 48.74**  **#define MER\_RADIUS 7.32**  **#define VEN\_RADIUS 18.15**  **#define EAR\_RADIUS 19.13**  **#define MOO\_RADIUS 6.15**  **#define MAR\_RADIUS 10.19**  **#define JUP\_RADIUS 42.90**  **#define SAT\_RADIUS 36.16**  **#define URA\_RADIUS 25.56**  **#define NEP\_RADIUS 24.78**    **#define MER\_DIS 62.06**  **#define VEN\_DIS 115.56**  **#define EAR\_DIS 168.00**  **#define MOO\_DIS 26.01**  **#define MAR\_DIS 228.00**  **#define JUP\_DIS 333.40**  **#define SAT\_DIS 428.10**  **#define URA\_DIS 848.00**  **#define NEP\_DIS 949.10**    **#define MER\_SPEED 87.0**  **#define VEN\_SPEED 225.0**  **#define EAR\_SPEED 365.0**  **#define MOO\_SPEED 30.0**  **#define MAR\_SPEED 687.0**  **#define JUP\_SPEED 1298.4**  **#define SAT\_SPEED 3225.6**  **#define URA\_SPEED 3066.4**  **#define NEP\_SPEED 6014.8**    **#define SELFROTATE 3**    **enum STARS {Sun, Mercury, Venus, Earth, Moon,**  **Mars, Jupiter, Saturn, Uranus, Neptune};**    **#define SET\_VALUE\_3(name, value0, value1, value2) \**  **((name)[0])=(value0), ((name)[1])=(value1), ((name)[2])=(value2)**    **SolarSystem::SolarSystem() {**    **viewX = 0;**  **viewY = REST\_Y;**  **viewZ = REST\_Z;**  **centerX = centerY = centerZ = 0;**  **upX = upY = 0;**  **upZ = 1;**    **GLfloat rgbColor[3] = {1, 0, 0};**  **stars[Sun] = new LightPlanet(SUN\_RADIUS, 0, 0, SELFROTATE, 0, rgbColor);**    **SET\_VALUE\_3(rgbColor, .2, .2, .5);**  **stars[Mercury] = new Planet(MER\_RADIUS, MER\_DIS, MER\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **SET\_VALUE\_3(rgbColor, 1, .7, 0);**  **stars[Venus] = new Planet(VEN\_RADIUS, VEN\_DIS, VEN\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **SET\_VALUE\_3(rgbColor, 0, 1, 0);**  **stars[Earth] = new Planet(EAR\_RADIUS, EAR\_DIS, EAR\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **SET\_VALUE\_3(rgbColor, 1, 1, 0);**  **stars[Moon] = new Planet(MOO\_RADIUS, MOO\_DIS, MOO\_SPEED, SELFROTATE, stars[Earth], rgbColor);**    **SET\_VALUE\_3(rgbColor, 1, .5, .5);**  **stars[Mars] = new Planet(MAR\_RADIUS, MAR\_DIS, MAR\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **SET\_VALUE\_3(rgbColor, 1, 1, .5);**  **stars[Jupiter] = new Planet(JUP\_RADIUS, JUP\_DIS, JUP\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **SET\_VALUE\_3(rgbColor, .5, 1, .5);**  **stars[Saturn] = new Planet(SAT\_RADIUS, SAT\_DIS, SAT\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **SET\_VALUE\_3(rgbColor, .4, .4, .4);**  **stars[Uranus] = new Planet(URA\_RADIUS, URA\_DIS, URA\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **SET\_VALUE\_3(rgbColor, .5, .5, 1);**  **stars[Neptune] = new Planet(NEP\_RADIUS, NEP\_DIS, NEP\_SPEED, SELFROTATE, stars[Sun], rgbColor);**    **}**  **SolarSystem::~SolarSystem() {**  **for(int i = 0; i<STARS\_NUM; i++)**  **delete stars[i];**  **}**  在终端中运行：  make && ./solarsystem  运行结果如图所示：    由于星球的颜色是单一的，光照效果不够明显但依然能够看到，诸如黄色的木星右侧可以看到有泛白，  此外，我们还可以通过键盘来调整太阳系的查看视角：    本课程相关代码：  wget http://labfile.oss.aliyuncs.com/courses/558/solarsystem.zip 五、拓展阅读  1. D Shreiner. OpenGL 编程指南. 人民邮电出版社, 2005. 2.   这本书详细介绍了 OpenGL 编程相关的方方面面，被誉为 『OpenGL 红宝书』  <https://www.shiyanlou.com/courses/558/labs/1884/document> |

下载的代码：<solarsystem>

# 21.C++11 从零实现 NoSQL 数据库

#### 参考1：

<https://github.com/ideawu/ssdb>

下载的文件：<ssdb-master>

#### 参考2：

mongodb源码：<https://github.com/mongodb/mongo/tags?after=r5.0.7>

已经下载好的mongodb源码：[G:\常见开源项目和工具源代码集合\part5\mongo-r5.0.8](file:///G:\常见开源项目和工具源代码集合\part5\mongo-r5.0.8)

#### 参考3：<https://github.com/KvrocksLabs/kvrocks/tags>

下载好的kvrocks项目：<kvrocks-2.0.4>

# 22.C++ 实现 XML 解析

#### 参考1：

GitHub：tinyXml：<https://github.com/icebreaker/TinyXML>

下载好的tinyxml项目：<TinyXML-master>

用法：

|  |
| --- |
| **C++解析XML文件**   * [TinyXML安装下载](https://blog.csdn.net/u014311125/article/details/89199435#TinyXML_3) * [解析XML文件](https://blog.csdn.net/u014311125/article/details/89199435#XML_6)   最近在使用C++处理GF-3和RADARSAT-2雷达影像，需要通过解析XML文件获取影像时间，我选用了 [TinyXML](http://www.grinninglizard.com/tinyxml/)库来解析。参考了网上的一些教程，这个文档个人觉得是解释的很清晰的。(https://www.cnblogs.com/hgwang/p/5833638.html)，第一次使用记录一下学习过程，供以后参考。  **TinyXML安装下载**  进入官网下载，解压缩，将其中的tinystr.h、tinystr.cpp、tinyxml.h、tinyxml.cpp、tinyxmlerror.cpp、tinyxmlparser.cpp 6个文件放在你的项目目录下，就是你自己写的.h和.cpp文件所在的目录。  **解析XML文件**  先看一下XML文件内容。下面这个是RADARSAT-2影像的product.xml文件  在XML文档中，每个<></>标签都是一个节点或元素。解析时需要逐层搜索（好像是不能全文检索），直到找到我们要的那个标签。同一层中，可以根据元素名使用IterateChildren方法来查找匹配的节点。  我的需求是获取rawDataStartTime节点中的时间。直接上代码。  **#include"tinyxml.h"**  **#include"tinystr.h"**  **TiXmlDocument doc;**  **//加载文档**  **if (!doc.LoadFile(xml\_file))**  **{**  **std::cerr << doc.ErrorDesc() << std::endl;**  **exit(1);**  **}**  **//获取根节点，也是xml文档的声明部分**  **TiXmlElement\* root = doc.FirstChildElement();**  **if (root == NULL)**  **{**  **std::cerr << "Failed to load file: No root element." << std::endl;**  **doc.Clear();**  **exit(1);**  **}**  **std::string xml\_time;**  **TiXmlNode \*source\_node = NULL;**  **//查找sourceAttributes节点**  **source\_node = root->IterateChildren("sourceAttributes", source\_node);**  **TiXmlElement \*source\_elem = NULL;**  **source\_elem = source\_node->ToElement();**  **TiXmlNode \*rawdata\_node = NULL;**  **//查找rawDataStarTime节点**  **rawdata\_node = source\_elem->IterateChildren("rawDataStartTime", rawdata\_node);**  **//获取该节点的文本信息。**  **xml\_time = rawdata\_node->ToElement()->GetText();**  **return xml\_time;**   * 5 |

#### 参考2：

tinyXml2：<https://github.com/leethomason/tinyxml2>

下载好的项目：<tinyxml2-master>

使用方法

|  |
| --- |
| C++那些事之优雅的解析XML 2020-05-28阅读 1.1K0  [本文被 2 个清单收录，推荐清单](https://cloud.tencent.com/developer/inventory/853)  **[C++那些事之实战问题篇](https://cloud.tencent.com/developer/inventory/853)**  最近使用开源的C++ XML解析器-TinyXML-2，今天主要分享该开源项目的基本情况及使用。  1.TinyXML介绍  TinyXML是一个比较优秀的c++ xml解析器，有两个版本，分别是TinyXML与TinyXML-2。最新开源版本TinyXML-2相对于旧版本的 TinyXml 使用更少的内存 , 更快 , 并且使用更少的内存分配 , 因此被广泛应用于现在开发中。  TinyXml2不需要STL ，自然降低了所有的 STL 支持 , 所有字符串操作使用 const char \*。两者相同点：   * 基于DOM解析 * API简单 * 支持UTF-8   不同点：  (1) TinyXML   * 支持部分STL操作：string、流   (2) TinyXML-2   * 适合现代C++开发 * 内存分配少，需要更少的内存 * 无需STL   TinyXML：  https://sourceforge.net/projects/tinyxml/  TinyXML-2:  https://github.com/leethomason/tinyxml2 ****2. TinyXML-2使用**** 使用很简单，下载上述github代码，copy其中的tinyxml2.h与.cpp到你的当前目录下，直接引用.h文件，编译即可。  例如：现有test.xml，parse\_test.cpp来解析test.xml中的节点。  test.xml  <category version="1.0">  <book>茅草屋</book>  <number>1</number>  <animal>猫</animal>  </category>  复制  parse\_test.cpp  #include "tinyxml2.h"  #include <iostream>  using namespace std;  using namespace tinyxml2;  int main()  {  XMLDocument xml;  xml.LoadFile("test.xml");  XMLElement \*category = xml.RootElement();  XMLElement \*book = category->FirstChildElement("book");  XMLElement \*number = category->FirstChildElement("number");  XMLElement \*animal = category->FirstChildElement("animal");  const char\* book\_name = book->GetText();  cout << book\_name << endl;  int num = atoi(number->GetText());  cout << num << endl;  const char\* animal\_name = animal->GetText();  cout << animal\_name << endl;  return 0;  }  复制  编译：  g++ -o parse parse\_test.cpp tinyxml2.cpp  复制  输出：  茅草屋  1  猫 |

#### 参考3:

Xerces C++: <https://archive.apache.org/dist/xerces/c/3/sources/>

已经下载好的项目：<xerces-c-3.1.2>

用法：

|  |
| --- |
| 编译：  **Xerces的下载，编译和在VS 2019中的配置**  **1.下载** 笔者选用的是Xerces 3.1.2版本：<https://archive.apache.org/dist/xerces/c/3/sources/>  注：下载该版本的源文件，编译生成库文件时很方便。因为该源文件中有.sln文件，可以便于使用VS打开，然后进行编译。  **2.编译** .编译xerces，获得dll文件和lib文件 具体方法：下载[xerces-c-3.1.2.zip](https://archive.apache.org/dist/xerces/c/3/sources/xerces-c-3.1.2.zip)后进行解压。 加压后，打开E:\xerces-c-3.1.2\projects\Win32\VC12\xerces-all\xerces-all.sln文件。  注：projects目录节后，如图：    VC9，VC10等分别对应不同的VS版本。因为笔者用的VS是2019，这里可供选择的最高版本VC12。因此笔者选择打开的是VC12下的.sln文件。  用VS打开项目文件后，编译。注意用Win32平台（X64平台笔者没有尝试，不知能不能用）。    编译后在Debug目录下生成一些文件，找到xerces-c\_3\_1D.dll和xerces-c\_3D.lib文件。 把它们复制到我们要使用得的项目目录下。不复制也行，但在VS中配置时，记得它们的路径就好。  注：笔者的Debug路径为： E:\xerces-c-3.1.2\Build\Win32\VC12\Debug  **3.配置**  项目上右键，点击属性： 1.C/C++ ->常规->附加包含目录：xerces-c-3.1.2\src  注：附加包含目录为xerces的源码目录，笔者的为：E:\xerces-c-3.1.2\src 当然，也可以把src拷贝到项目目录下，此时的“附加包含目录”做对应改变即可。  2.链接器->常规->附加库目录 ：xerces-c-3.1.1\Build\Win32\VC12\Debug  注：附加库目录为我们在编译xerces源码时，生成的一些库文件目录。笔者的为： E:\xerces-c-3.1.2\Build\Win32\VC12\Debug  3.链接器->输入->附加依赖项：加上一项 xerces-c\_3D.lib      4.加上头文件和命名空间  **#include <xercesc/util/PlatformUtils.hpp>**  **#include <xercesc/dom/DOM.hpp>**  **#include <xercesc/sax/HandlerBase.hpp>**  **#include <xercesc/parsers/XercesDOMParser.hpp>**  **XERCES\_CPP\_NAMESPACE\_USE**  具体使用，自行查阅文档：<http://xerces.apache.org/xerces-c/ApacheDOMC++Binding.html> Xerces实现xml解析支持DOM, SAX, or SAX2的方式：<http://xerces.apache.org/xerces-c/program-3.html>  参照官网，大家会有更多收获！ |
| 使用：  **Xerces库使用说明（DOM解析）**    **3.Xerces库使用说明（DOM解析）**  红色的叫标签，蓝色的是标签的属性。  1.根据某个标签，获取整个标签中的内容。  **XercesDOMParser\* parser = new XercesDOMParser();**  **const char\* xmlFile = "C:\\Users\\tianyanbin\\source\\repos\\MyXml\\MyXml\\test.xml";//绝对路径**  **parser->parse(xmlFile);**  **DOMDocument\* doc = parser->getDocument();**  **DOMElement\* m\_pRoot = doc->getDocumentElement(); //获取XML根节点**  **DOMNodeList\* pVersionEleList = m\_pRoot->getElementsByTagName(XMLString::transcode("Sector"));//获取Sector节点**  2.获取便签中某个属性的值  **DOMNode\* child001 = XXXX; //chilid001是某个结点，对应一个标签对象**  **DOMNamedNodeMap\* pAttributes = child001->getAttributes(); //----------------取属性表**  **const XMLCh\* attrDef = XMLString::transcode("ProcessParameter"); //----------------把属性名字符串转变一下存起来；**  **DOMNode\* node = pAttributes->getNamedItem(attrDef); //---------------- - 根据属性名取出属性**  **XMLString::release((XMLCh\*\*)(&attrDef)); //----------------记得和XMLString::transcode配对使用，否则内存泄露**  **char\* pc = XMLString::transcode(node->getNodeValue()); // ----------------把属性转变为char\***  **cout << "ProcessParameter:" << pc << endl;**  **XMLString::release(&pc);//realse和**  完成项目代码和xml文件：  test.xml：  <Group>  <Id>0</Id>  <Detail>  <SectorsNum>100</SectorsNum>  <Sector>  <Id>0</Id>  <Gun>  <Id>0</Id>  <DataBlock ProcessParameter="9" ReginId="2" Type="NormalPoly">  <PolyLine Count="8">193.784 27.234 0.000 193.731 26.621 -0.001 193.414 21.899 -0.011 193.189 17.245 -0.021 193.052 12.596 -0.031 193.004 7.983 -0.041 193.046 3.055 -0.052 193.131 0.000 -0.058</PolyLine>  <PolyLine Count="5">195.168 4.852 -0.058 196.265 4.867 -0.061 196.252 8.144 -0.068 195.153 8.148 -0.070 195.168 4.852 -0.077</PolyLine>  <PolyLine Count="5">196.253 8.544 -0.077 196.288 11.821 -0.084 195.189 11.837 -0.087 195.154 8.548 -0.094 196.253 8.544 -0.096</PolyLine>  <PolyLine Count="5">195.236 1.160 -0.096 196.335 1.188 -0.098 196.271 4.467 -0.105 195.172 4.452 -0.108 195.236 1.160 -0.115</PolyLine>  <PolyLine Count="5">196.297 12.220 -0.115 196.378 15.495 -0.122 195.278 15.532 -0.124 195.197 12.236 -0.131 196.297 12.220 -0.134</PolyLine>  <PolyLine Count="4">196.372 0.000 -0.134 196.345 0.788 -0.135 195.246 0.760 -0.138 195.272 0.000 -0.139</PolyLine>  <PolyLine Count="5">196.393 15.894 -0.139 196.524 19.171 -0.146 195.426 19.221 -0.149 195.293 15.931 -0.156 196.393 15.894 -0.158</PolyLine>  <PolyLine Count="5">196.545 19.568 -0.158 196.725 22.840 -0.165 195.627 22.906 -0.168 195.445 19.620 -0.175 196.545 19.568 -0.177</PolyLine>  <PolyLine Count="5">196.750 23.237 -0.177 196.978 26.504 -0.184 195.883 26.588 -0.186 195.653 23.305 -0.193 196.750 23.237 -0.196</PolyLine>  <PolyLine Count="3">195.962 27.540 -0.196 195.915 26.985 -0.197 197.011 26.902 -0.199</PolyLine>  <PolyLine Count="2">197.079 27.697 -0.199 197.011 26.902 -0.201</PolyLine>  <PolyLine Count="7">199.269 28.005 -0.201 199.003 24.686 -0.208 198.709 19.796 -0.219 198.513 14.951 -0.229 198.403 8.246 -0.243 198.437 3.560 -0.253 198.533 0.000 -0.261</PolyLine>  </DataBlock>  <DataBlock ProcessParameter="11" ReginId="2" Type="Hatch">  <PolyLine Count="8">193.784 27.234 0.000 193.731 26.621 -0.001 193.414 21.899 -0.011 193.189 17.245 -0.021 193.052 12.596 -0.031 193.004 7.983 -0.041 193.046 3.055 -0.052 193.131 0.000 -0.058</PolyLine>  </DataBlock>  </Gun>  <Gun>  <Id>1</Id>  <DataBlock ProcessParameter="9" ReginId="2" Type="NormalPoly">  <PolyLine Count="8">193.784 27.234 0.000 193.731 26.621 -0.001 193.414 21.899 -0.011 193.189 17.245 -0.021 193.052 12.596 -0.031 193.004 7.983 -0.041 193.046 3.055 -0.052 193.131 0.000 -0.058</PolyLine>  <PolyLine Count="5">195.168 4.852 -0.058 196.265 4.867 -0.061 196.252 8.144 -0.068 195.153 8.148 -0.070 195.168 4.852 -0.077</PolyLine>  <PolyLine Count="5">196.253 8.544 -0.077 196.288 11.821 -0.084 195.189 11.837 -0.087 195.154 8.548 -0.094 196.253 8.544 -0.096</PolyLine>  <PolyLine Count="5">195.236 1.160 -0.096 196.335 1.188 -0.098 196.271 4.467 -0.105 195.172 4.452 -0.108 195.236 1.160 -0.115</PolyLine>  <PolyLine Count="5">196.297 12.220 -0.115 196.378 15.495 -0.122 195.278 15.532 -0.124 195.197 12.236 -0.131 196.297 12.220 -0.134</PolyLine>  <PolyLine Count="4">196.372 0.000 -0.134 196.345 0.788 -0.135 195.246 0.760 -0.138 195.272 0.000 -0.139</PolyLine>  <PolyLine Count="5">196.393 15.894 -0.139 196.524 19.171 -0.146 195.426 19.221 -0.149 195.293 15.931 -0.156 196.393 15.894 -0.158</PolyLine>  <PolyLine Count="5">196.545 19.568 -0.158 196.725 22.840 -0.165 195.627 22.906 -0.168 195.445 19.620 -0.175 196.545 19.568 -0.177</PolyLine>  <PolyLine Count="5">196.750 23.237 -0.177 196.978 26.504 -0.184 195.883 26.588 -0.186 195.653 23.305 -0.193 196.750 23.237 -0.196</PolyLine>  <PolyLine Count="3">195.962 27.540 -0.196 195.915 26.985 -0.197 197.011 26.902 -0.199</PolyLine>  <PolyLine Count="2">197.079 27.697 -0.199 197.011 26.902 -0.201</PolyLine>  <PolyLine Count="7">199.269 28.005 -0.201 199.003 24.686 -0.208 198.709 19.796 -0.219 198.513 14.951 -0.229 198.403 8.246 -0.243 198.437 3.560 -0.253 198.533 0.000 -0.261</PolyLine>  </DataBlock>  <DataBlock ProcessParameter="11" ReginId="2" Type="Hatch">  <PolyLine Count="8">193.784 27.234 0.000 193.731 26.621 -0.001 193.414 21.899 -0.011 193.189 17.245 -0.021 193.052 12.596 -0.031 193.004 7.983 -0.041 193.046 3.055 -0.052 193.131 0.000 -0.059</PolyLine>  </DataBlock>  </Gun>  </Sector>  </Detail>  </Group>    代码：MyXml.cpp  **#include <xercesc/util/PlatformUtils.hpp>**  **#include <xercesc/parsers/XercesDOMParser.hpp>**  **#include <xercesc/dom/DOM.hpp>**  **#include <xercesc/sax/HandlerBase.hpp>**  **#include <xercesc/util/XMLString.hpp>**  **#include <iostream>**  **#include<fstream>**  **#include<string>**  **#include <vector>**  **#include<sstream> //istringstream 必须包含这个头文件**  **// Other include files, declarations, and non-Xerces-C++ initializations.**  **#pragma comment (lib,"xerces-c\_3D.lib")**  **using namespace xercesc;**  **using namespace std;**  **string trim(string s) {**  **if (s.empty())**  **return s;**  **s.erase(0, s.find\_first\_not\_of(" "));**  **s.erase(s.find\_last\_not\_of(" ") + 1);**  **return s;**  **}**  **typedef struct DataBlock {**  **int SectorID;**  **int GunID;**  **int ProcessParameter;**  **//vector里面存储的是一个完整DataBlock**  **vector<vector<float>> DataBlock;**  **}datablock;//结构体中含有string要使用new分配内存，不能使用malloc**  **int main()**  **{**  **try {**  **XMLPlatformUtils::Initialize();**  **}**  **catch (const XMLException& toCatch) {**  **// Do your failure processing here**  **cout << "Initialize Error!" << endl;**  **return 1;**  **}**  **// Do your actual work with Xerces-C++ here.**  **XercesDOMParser\* parser = new XercesDOMParser();**  **parser->setValidationScheme(XercesDOMParser::Val\_Always);**  **parser->setDoNamespaces(true); // optional**  **ErrorHandler\* errHandler = (ErrorHandler\*) new HandlerBase();**  **parser->setErrorHandler(errHandler);**  **//此处请用绝对路径 C:\\Users\\tianyanbin\\source\\repos\\MyXml\\MyXml\\book.xml personal.xml 0.xml job.xml**  **//相对路径/book.xml**  **const char\* xmlFile = "C:\\Users\\tianyanbin\\source\\repos\\MyXml\\MyXml\\test.xml";**  **try {**  **parser->parse(xmlFile);**  **}**  **catch (const XMLException& toCatch) {**  **char\* message = XMLString::transcode(toCatch.getMessage());**  **cout << "Exception message is: \n"**  **<< message << "\n";**  **XMLString::release(&message);**  **return -1;**  **}**  **catch (const DOMException& toCatch) {**  **char\* message = XMLString::transcode(toCatch.msg);**  **cout << "Exception message is: \n"**  **<< message << "\n";**  **XMLString::release(&message);**  **return -2;**  **}**  **catch (const SAXException& toCatch) {**  **char\* message = XMLString::transcode(toCatch.getMessage());**  **cout << "Exception message is: \n"**  **<< message << "\n";**  **XMLString::release(&message);**  **return -3;**  **}**  **catch (...) {**  **cout << "Unexpected Exception \n";**  **return -4;**  **}**  **//读取**  **DOMDocument\* doc = parser->getDocument();**  **DOMNodeList\* nodelist = doc->getChildNodes();**  **DOMElement\* m\_pRoot = doc->getDocumentElement(); //获取XML根节点**  **DOMNode\* root = (DOMNode\*)doc->getDocumentElement();**  **DOMNodeList\* childlist;**  **DOMNodeList\* childlistProcessParameterSets;**  **DOMNode\* childProcessParameterSets;**  **DOMNode\* childParameterSet;**  **DOMNode\* Parameter;**  **DOMNode\* Parameter1;**  **DOMNodeList\* childParameter1;**  **DOMNode\* Parameter11;**  **DOMNode\* Parameter2;**  **DOMNodeList\* childParameter2;**  **DOMNode\* Parameter22;**  **DOMNodeList\* Parameters;**  **//头文件#include<fstream>**  **ofstream out("./data.txt", ios::app);//app表示每次操作前均定位到文件末尾**  **if (out.fail()) {**  **cout << "error\n";**  **}**  **//out << nodename << endl;**  **vector<datablock> db;**    **if (m\_pRoot)**  **{**  **DOMNodeList\* pVersionEleList = m\_pRoot->getElementsByTagName(XMLString::transcode("Sector"));//获取Sector节点**  **//**  **//cout << XMLString::transcode(m\_pRoot->getTextContent());**  **//out << XMLString::transcode(m\_pRoot->getTextContent()) << endl;**  **for (int i = 0; i < pVersionEleList->getLength(); ++i)**  **{**  **//每个i 对应一个Sector,一个node1就是一个Sector节点**  **DOMNode\* node1 = pVersionEleList->item(i);**  **DOMNode\* child = 0;**  **datablock p;**  **int m = 0;//记录那个枪，0，1，2，3,循环往复**  **for (child = node1->getFirstChild(); child != 0; child = child->getNextSibling())**  **{**  **//Node node = childNodes.item(i);**  **//int nodeType = node.getNodeType(); nodeType为3时，为#text**  **//if (XMLString::transcode(child->getNodeName()) != "#text")//必须先对XMLString::transcode(child->getNodeName())去空格**    **if (child->getNodeType() != 3)**  **{**  **if(trim(XMLString::transcode(child->getNodeName()) )=="Id")//Sector-ID**  **{**  **p.SectorID = stoi(XMLString::transcode(child->getTextContent()));**  **//m++;**  **cout << XMLString::transcode(child->getNodeName()) << ":" << XMLString::transcode(child->getTextContent());//注：这里使用child->getNodeValue()报错**  **}**  **if (trim(XMLString::transcode(child->getNodeName())) == "Gun")//有四个枪，每解析完四个枪，写一次vector,下次就是下一个Sector扇区**  **{**  **if (m == 0)//第一个电子枪**  **{**  **DOMNode\* child001 = 0;**  **for (child001 = child->getFirstChild(); child001 != 0; child001 = child001->getNextSibling())**  **{**  **if (trim(XMLString::transcode(child001->getNodeName())) == "Id")//Gun-ID**  **{**  **p.GunID = stoi(XMLString::transcode(child->getTextContent()));**  **cout << XMLString::transcode(child001->getNodeName()) << ":" << XMLString::transcode(child001->getTextContent());//注：这里使用child->getNodeValue()报错**    **}**  **//解析DataBlock，每个枪里面不知道有多少个DataBlock，来一个加一个**  **if (trim(XMLString::transcode(child001->getNodeName())) == "DataBlock")**  **{**  **//获取DataBlock标签的属性值attribute**  **DOMNamedNodeMap\* pAttributes = child001->getAttributes(); //----------------取属性表**  **const XMLCh\* attrDef = XMLString::transcode("ProcessParameter"); //----------------把属性名字符串转变一下存起来；**  **DOMNode\* node = pAttributes->getNamedItem(attrDef); //---------------- - 根据属性名取出属性**  **XMLString::release((XMLCh\*\*)(&attrDef)); //----------------记得和XMLString::transcode配对使用，否则内存泄露**  **char\* pc = XMLString::transcode(node->getNodeValue()); // ----------------把属性转变为char\***  **p.ProcessParameter = stoi(XMLString::transcode(node->getNodeValue()));**  **cout << "ProcessParameter:" << pc << endl;**  **XMLString::release(&pc);**  **DOMNode\* child002 = 0;**  **//解析PolyLine，每个DataBlock里面不知道有多少个PolyLine**  **for (child002 = child001->getFirstChild(); child002 != 0; child002 = child002->getNextSibling())**  **{**  **vector<float> v;**  **if (trim(XMLString::transcode(child002->getNodeName())) == "PolyLine")**  **{**  **//这里必须调用个函数：其功能是把数字字符串转化为数字接收**  **string s = "05 15 25";**  **istringstream ss(XMLString::transcode(child002->getTextContent()));**  **//vector<int> v;**  **float i;**  **while (ss >> i)**  **{**  **v.push\_back(i);**  **}**  **p.DataBlock.push\_back(v);**  **cout << "打印vector内容" << endl;**  **copy(v.begin(), v.end(), ostream\_iterator<float>(cout, " "));**  **//p.ID = stoi(XMLString::transcode(child->getTextContent()));**  **cout << XMLString::transcode(child002->getNodeName()) << ":" << XMLString::transcode(child002->getTextContent());//注：这里使用child->getNodeValue()报错**  **}**  **}**  **}**  **}**    **}**  **}**    **}**  **}**  **}**  **}**  **out.close();//关闭文件**  **int stop;**  **cin >> stop;**  **delete parser;**  **delete errHandler;**  **XMLPlatformUtils::Terminate();**  **// Other terminations and cleanup.**  **return 0;**  **}**  代码功能：  把test.xml文件中的信息解析出来，存到vector里面。 同时，在控制台打印出信息。 也可以把解析出的信息存入txt文件中，这部分代码里也有，但没写完整。  Refferences: 1.[xml解析库对比总结(解析方式，第三方库，TinyXml)](https://blog.csdn.net/shyjhyp11/article/details/89392383) 2.[xerces dom xml解析](https://blog.csdn.net/an_zhenwei/article/details/9376919) |

# 参考4：

# ticpp: <https://github.com/wxFormBuilder/ticpp>

下载好的项目：<ticpp-master>