Boost 库的 pool 提供了一个内存池分配器，用于管理在一个独立的、大的分配空间里的动态内存分配。

Boost 库的 pool 主要适用于快速分配同样大小的内存块，尤其是反复分配和释放同样大小的内存块的情况。使用 pool 内存池主要有以下两个优点：

1．能够有效地管理许多小型对象的分配和释放工作，避免了自己去管理内存而产生的内存碎片和效率低下问题。

2． 告别程序内存泄漏的烦恼，pool 库会在内部对内存自动进行管理，避免了程序员一不小心而造成的内存泄漏问题。

pool 库主要提供了四种内存池接口，分别是

pool、object_pool、singleton_pool 和 pool_allocator/fast_pool_allocator

1． pool

pool 是最简单也最容易使用的内存池类，可以返回一个简单数据类型（POD） 的内存指针。它

pool 很容易使用，可以像 C 中的 malloc()一样分配内存，然后随意使用。除非有特殊要求，否则不必对分配的内存调用 free()释放，pool 会很好地管理内存。例如：

view plaincopy to clipboardprint?

```
#include <boost/pool/pool.hpp>

using namespace boost;

int main()

{
```

```
  pool<> pl(sizeof(int));          //一个可分配 int 的内存池
  int *p = (int *)pl.malloc();     //必须把 void*转换成需要的类型
  assert(pl.is_from(p));
  pl.free(p);                      //释放内存池分配的内存块
  for (int i = 0;i < 100; ++i)     //连续分配大量的内存
  {
    pl.ordered_malloc(10);
  }
}
```

2. object_pool

object_pool 是**用于类实例（对象）的内存池，它的功能与 pool 类似，但会在析构时对所有已经分配的内存块调用析构函数，从而正确地释放资源。**

malloc()和 free()函数分别分配和释放一块类型为 ElementType*的内存块，同样，可以用 is_from()来测试内存块的归属，只有是本内存池分配的内存才能被 free()释放。但它们被调用时并不调用类的构造函数和析构函数，也就是说操作的是一块原始内存块，里面的值是未定义的，因此我们应当尽量少使用 malloc()和 free()。

object_pool 的特殊之处是**construct()和 destroy()函数，**这两个函数是 object_ pool 的真正价值所在。construct()实际上是一组函数，有多个参数的重载形式（目前最多支持 3 个参数，但可以扩展），它先调用 malloc()分配内存，然后再在内存块上使用传入的参数调用类的构造函数，返回的是一个已经初始化的对象指针。destory()则先调用对象的析构函数，然后再用 free()释放内存块。

这些函数都不会抛出异常，**如果内存分配失败，将返回 0。**

object_pool 的用法也是很简单，我们既可以像 pool 那样分配原始内存块，也可以使用 construct()来直接在内存池中创建对象。当然，后一种使用方法是最方便的，也是本书所推荐的。

下面的代码示范了 object_pool 的用法：

```
#include <boost/pool/object_pool.hpp>

using namespace boost;

struct demo_class                    //一个示范用的类

{

public:

  int a,b,c;

  demo_class(int x = 1, int y = 2, int z = 3):a(x),b(y),c(z){}

};

int main()

{

  object_pool<demo_class> pl;          //对象内存池

  demo_class *p = pl.malloc();         //分配一个原始内存块

  assert(pl.is_from(p));     //p 指向的内存未经过初始化

  assert(p->a!=1 || p->b != 2 || p->c !=3);

  p = pl.construct(7, 8, 9);           //构造一个对象,可以传递参数

  assert(p->a == 7);

  object_pool<string> pls;             //定义一个分配 string 对象的内存池

  for (int i = 0; i < 10 ; ++i)        //连续分配大量 string 对象

  {

    string *ps = pls.construct("hello object_pool");
```

```
    cout << *ps << endl;
  }
}                                   //所有创建的对象在这里都被正确析构、释放内存
```

3. singleton_pool

singleton_pool 与 pool 的接口完全一致，可以分配简单数据类型（POD）的内存指针，但它是一个单件，**并提供线程安全。**

singleton_pool 主要有两个模板类型参数（其余的可以使用缺省值）。第一个 Tag 仅仅是用于标记不同的单件，可以是空类，甚至是声明。第二个参数 RequestedSize 等同于 pool 构造函数中的整数 requested_ size，指示 pool 分配内存块的大小。

singleton_pool 的接口与 pool 完全一致，但成员函数均是静态的，因此不需要声明 singleton_pool 的实例 ，直接用域操作符::来调用静态成员函数。因为 singleton_pool 是单件，所以它的生命周期与整个程序同样长，除非手动调用 release_memory()或 purge_memory()，否则 singleton_pool 不会自动释放所占用的内存。除了这两点，singleton_pool 的用法与 pool 完全相同。

下面的代码示范了 singleton_pool 的用法：

```
#include <boost/pool/singleton_pool.hpp>
using namespace boost;
struct pool_tag{};                          //仅仅用于标记的空类
typedef singleton_pool<pool_tag, sizeof(int)> spl;  //内存池定义
int main()
{
  int *p = (int *)spl::malloc();            //分配一个整数内存块
  assert(spl::is_from(p));
  spl::release_memory();                    //释放所有未被分配的内存
```

}　　//spl 的内存直到程序结束才完

singleton_pool 在使用时最好使用 typedef 来简化名称，否则会使得类型名过于冗长而难以使用。如代码中所示:

typedef singleton_pool<pool_tag, sizeof(int)> spl;

用于标记的类 pool_tag 可以再进行简化，直接在模板参数列表中声明 tag 类，这样可以在一条语句中完成对 singleton_pool 的类型定义，例如：

typedef singleton_pool<struct pool_tag, sizeof(int)> spl;

singleton_pool 为单例类，是对 pool 的加锁封装，适用于多线程环境，其中所有函数都是静态类型。它的模版参数有 5 个，

tag：　　标记而已，无意义；

RequestedSize：　block 的长度；

UserAllocator：分配子，默认还是 default_user_allocator_new_delete；

Mutex：　　锁机制，默认值最终依赖于系统环境，linux 下是 pthread_mutex，它是对 pthread_mutex_t 的封装；

NextSize：　内存不足的时候，申请的 block 数量，默认是 32。

最全面的使用 singleton_pool 类似这样：

typedef

boost::singleton_pool<singleton_pool_tag,sizeof(CStudent),default_user_allocator_new_delete,details::pool::default_mutex,200> global;

它暴露的函数和 pool 相同。

4）pool_allocator/fast_pool_allocator

stl::allocator 的替换方案。两者都是基于 singleton_pool 实现，实现了 stl::allocator 要求的接口规范。两者的使用相同，区别在于 pool_allocator 的内部实现调用了 ordered_malloc 和 ordered_free，可以满足对大量的连续内存块的分配请求。fast_pool_allocator 的内部实现调用了 malloc 和 free，比较适合于一次请求单个大内存块的情况，但也适用于通用分配，不过具有一些性能上的缺点。因此推荐使用后者。

```
#include <boost/pool/pool_alloc.hpp>
#include <vector>
typedef struct student_st
{
 char name[10];
 int age;
}CStudent;
int main()
{
  std::vector<CStudent *,boost::fast_pool_allocator<CStudent *> > v(8);
  CStudent *pObj=new CStudent();
  v[1]=pObj;
  boost::singleton_pool<boost::fast_pool_allocator_tag,sizeof(CStudent
*)>::purge_memory();
  return 0;
}
```

fast_pool_allocator 的模版参数有四个：类型，分配子，锁类型，内存不足时的申请的 block 数量，后三者都有默认值，不再说了。

它使用的 singleton_pool 的 tag 是 boost::fast_pool_allocator_tag。

总结：

boost::pool 小巧高效，多多使用，

boost::singleton_pool 多线程环境下使用，不要使用两者的 ordered_malloc/orderd_free 函数。

boost::object_pool 不建议使用，可以改造后使用。

pool_allocator/fast_pool_allocator 推荐使用后者。用于与 STL 关连。。