原创 guangcheng0312q 于 2019-10-22 00:00:00 发布 304 收藏 1 版权

STL 设计之 EBO(空基类优化)

0.导语

EBO 简称 Empty Base Optimization。

本节从空类开始，到 STL 内部，到测试，再到我们自己实现一个 EBO，对比性能，最后再测试，总结。

1.空类

定义一个空类：没有成员变量，没有继承，没有数据元素的类。

```cpp
class Empty{
public:
    void print() {
        std::cout<<"I am Empty class"<<std::endl;
    }
};
```

由于它是空的，所以这个 sizeof 是多少呢?

```cpp
std::cout<<sizeof(Empty)<<std::endl; //1
```

结果是 1，**它是空的怎么不是 0 呢?**

因为空类同样可以被实例化，每个实例在内存中都有一个独一无二的地址，为了达到这个目的，编译器往往会给一个空类隐含的加一个字节，这样空类在实例化后在内存得到了独一无二的地址．所以上述大小为 1.

根据上面的回答，估计大家或引出另一个问题：**为什么两个不同对象的地址应该不同?**

现在有下面这个例子：

```cpp
class Empty{
public:
    void print() {
        std::cout<<"I am Empty class"<<std::endl;
    }
};
template < typename T >
bool isSame( T const & t1, T const & t2 )
{
    return &t1 == &t2;
}
```

我们来测试一下：

```cpp
int main() {
    Empty a,b;
    assert(!isSame(a,b));  // 编译通过，a与b的地址不同

    Empty *p=new Empty;
    Empty *q=new Empty;
    assert(!isSame(p,q)); // 编译通过，a与b的地址不同
    return 0;
}
```

上面测试了，两个不同对象地址是不同的，考虑下面场景：

```cpp
Empty a,b;
// 在同一地址具有两个对象将意味着在使用指针引用它们时将无法区分这两个对象。
Empty *p1=&a;
p1->print();
```

此时会发现，如果 a 的地址与 b 的地址一样，那么在同一地址具有两个对象将意味着在使用指针引用它们时将无法区分这两个对象。因此两个不同对象的地址不同。

2.空基类优化

现在对比一下下面两个用法，第一种，一个类中包含了另一个类作为成员，然后通过这个来获得被包含类的功能。

```cpp
class notEbo  {
    int i;
    Empty e;
    // do other things
};
```

另一种直接采用继承的方式来获得基类的成员函数及其他功能等等。

```cpp
class ebo:public Empty {
    int i;
    // do other things
};
```

接下来做个测试：

```cpp
std::cout<<sizeof(notEbo)<<std::endl;
std::cout<<sizeof(ebo)<<std::endl;
```

输出：

```
8
4
```

第一种，会因为字节对齐，将其原来只占 1 字节，进行扩充到 4 的倍数，最后就是 8 字节。

对比这两个发现，第二种通过继承方式来获得基类的功能，**并没有产生额外大小的优化称之为 EBO(空基类优化)**。

接下来，我们回到 STL 源码中，看看其中的使用！

3.STL 中的 EBO 世界

不管是 deque、rb_tree、list 等容器，都离不开内存管理，在这几个容器中都包含了相应的内存管理，并通过`_xx_impl`来继承下面这几个类：

```cpp
std::allocator<_Tp>
__gnu_cxx::bitmap_allocator<_Tp>
__gnu_cxx::bitmap_allocator<_Tp>
__gnu_cxx::__mt_alloc<_Tp>
__gnu_cxx::__pool_alloc<_Tp>
__gnu_cxx::malloc_allocator<_Tp>
```

那这和我们的 EBO 有啥关系呢?

实际上，上面所列出继承的基类都是内存管理的 EBO(空基类)。

在每个容器中的使用都是调用每个内存管理的`rebind<_Tp>::other`。

例如红黑树源码结构：

```cpp
typedef typename __gnu_cxx::__alloc_traits<_Alloc>::template
        rebind<_Rb_tree_node<_Val> >::other _Node_allocator;
struct _Rb_tree_impl : public _Node_allocator
{
// do somethings
};
```

接下来我们看上面列出的内存管理类里面的源码结构：这里拿`allocator`举例：

```cpp
template<typename _Tp>
class allocator: public __allocator_base<_Tp>
{
     template<typename _Tp1>
     struct rebind { typedef allocator<_Tp1> other; };
};
```

看到了没，通过`rebind<_Tp>::other`来获得传递进来的内存分配器，也就是前面提到的这些。

```cpp
std::allocator<_Tp>
__gnu_cxx::bitmap_allocator<_Tp>
__gnu_cxx::bitmap_allocator<_Tp>
__gnu_cxx::__mt_alloc<_Tp>
__gnu_cxx::__pool_alloc<_Tp>
__gnu_cxx::malloc_allocator<_Tp>
```

搞懂了这些，来测试一波：

```cpp
void print() {
    cout<<sizeof(std::allocator<int>)<<" "<<sizeof(std::allocator<int>::rebind<int>::other)<<endl;
    cout<<sizeof(__gnu_cxx::bitmap_allocator<int>)<<" "<<sizeof(__gnu_cxx::bitmap_allocator<int>::rebind<in
    cout<<sizeof(__gnu_cxx::new_allocator<int>)<<" "<<sizeof(__gnu_cxx::new_allocator<int>::rebind<int>::ot
    cout<<sizeof(__gnu_cxx::__mt_alloc<int>)<<" "<<sizeof(__gnu_cxx::__mt_alloc<int>::rebind<int>::other)<<
    cout<<sizeof(__gnu_cxx::__pool_alloc<int>)<<" "<<sizeof(__gnu_cxx::__pool_alloc<int>::rebind<int>::othe
    cout<<sizeof(__gnu_cxx::malloc_allocator<int>)<<" "<<sizeof(__gnu_cxx::malloc_allocator<int>::rebind<in
}
```

我们来测试这些 sizeof 是不是 1，经过测试输出如下：

```
1 1
1 1
1 1
1 1
1 1
1 1
```

说明内存管理的实现就是通过采用继承的方式，使用空基类优化，来达到尽量降低容器所占的大小。

4.利用 EBO,手动实现一个简单的内存分配与释放

首先定义一个 sizeof(class)=1 的类，同 STL 一样，里面使用 allocate 与 deallocate 来进行内存管理。

```cpp
class MyAllocator {
public:
    void *allocate(std::size_t size) {
        return std::malloc(size);
    }

    void deallocate(void *ptr) {
        std::free(ptr);
    }
};
```

第一种方式的内存管理：嵌入一个内存管理类

```cpp
template<class T, class Allocator>
class MyContainerNotEBO {
    T *data_ = nullptr;
    std::size_t capacity_;
    Allocator allocator_;   // 嵌入一个MyAllocator
public:
    MyContainerNotEBO(std::size_t capacity)
            : capacity_(capacity), allocator_(), data_(nullptr) {
        std::cout << "alloc malloc" << std::endl;
        data_ = reinterpret_cast<T *>(allocator_.allocate(capacity * sizeof(T))); // 分配内存
    }

    ~MyContainerNotEBO() {
        std::cout << "MyContainerNotEBO free malloc" << std::endl;
        allocator_.deallocate(data_);
    }
};
```

第二种方式：采用空基类优化，继承来获得内存管理功能

```cpp
template<class T, class Allocator>
class MyContainerEBO
        : public Allocator {    // 继承一个EBO
    T *data_ = nullptr;
    std::size_t capacity_;
public:
    MyContainerEBO(std::size_t capacity)
            : capacity_(capacity), data_(nullptr) {
        std::cout << "alloc malloc" << std::endl;
        data_ = reinterpret_cast<T *>(this->allocate(capacity * sizeof(T)));
    }

    ~MyContainerEBO() {
        std::cout << "MyContainerEBO free malloc" << std::endl;
        this->deallocate(data_);
    }
};
```

开始测试：

```cpp
int main() {
    MyContainerNotEBO<int, MyAllocator> notEbo = MyContainerNotEBO<int, MyAllocator>(0);
    std::cout << "Using Not EBO Test sizeof is " << sizeof(notEbo) << std::endl;
    MyContainerEBO<int, MyAllocator> ebo = MyContainerEBO<int, MyAllocator>(0);
    std::cout << "Using EBO Test sizeof is " << sizeof(ebo) << std::endl;

    return 0;
}
```

测试结果：

```
alloc malloc
Using Not EBO Test sizeof is 24
alloc malloc
Using EBO Test sizeof is 16
MyContainerEBO free malloc
MyContainerNotEBO free malloc
```

我们发现采用 EBO 的设计确实比嵌入设计好很多。至此，本节学习完毕。