13.智能指针的使用及其原理

1. 智能指针的使用场景分析

下面程序中我们可以看到，new了以后，我们也delete了，但是因为抛异常导，后面的delete没有得到执行，所以就内存泄漏了，所以我们需要new以后捕获异常，捕获到异常后delete内存，再把异常抛出，但是因为new本身也可能抛异常，连续的两个new和下面的Divide都可能会抛异常，让我们处理起来很麻烦。智能指针放到这样的场景里面就让问题简单多了。

```
double Divide(int a, int b)
{
    // 当b == 0时抛出异常
    if (b == 0)
    {
            throw "Divide by zero condition!";
    }
    else
    {
        return (double)a / (double)b;
    }
}

void Func()
{
    // 这里可以看到如果发生除0错误抛出异常，另外下面的array和array2没有得到释放。
    // 所以这里捕获异常后并不处理异常，异常还是交给外面处理，这里捕获了再重新抛出去。
    // 但是如果array2new的时候抛异常呢，就还需要套一层捕获释放逻辑，这里更好解决方案
    // 是智能指针，否则代码太戳了
    int* array1 = new int[10];
    int* array2 = new int[10];   // 抛异常呢

    try
    {
            int len, time;
            cin >> len >> time;
            cout << Divide(len, time) << endl;
    }
    catch (...)
    {
            cout << "delete []" << array1 << endl;
            cout << "delete []" << array2 << endl;

```

```
34              delete[] array1;
35              delete[] array2;
36
37              throw; // 异常重新抛出，捕获到什么抛出什么
38      }
39
40      // ...
41
42      cout << "delete []" << array1 << endl;
43      delete[] array1;
44
45      cout << "delete []" << array2 << endl;
46      delete[] array2;
47  }
48
49  int main()
50  {
51      try
52      {
53              Func();
54      }
55      catch (const char* errmsg)
56      {
57              cout << errmsg << endl;
58      }
59      catch (const exception& e)
60      {
61              cout << e.what() << endl;
62      }
63      catch (...)
64      {
65              cout << "未知异常" << endl;
66      }
67
68      return 0;
69  }
```


2. RAII和智能指针的设计思路

- RAII是Resource Acquisition Is Initialization的缩写，他是一种管理资源的类的设计思想，本质是一种利用对象生命周期来管理获取到的动态资源，避免资源泄漏，这里的资源可以是内存、文件指针、网络连接、互斥锁等等。RAII在获取资源时把资源委托给一个对象，接着控制对资源的访问，资源在对象的生命周期内始终保持有效，最后在对象析构的时候释放资源，这样保障了资源的正常释放，避免资源泄漏问题。

- 智能指针类除了满足RAII的设计思路，还要方便资源的访问，所以智能指针类还会想迭代器类一样，重载`operator*/operator->/operator[]`等运算符，方便访问资源。

```cpp
template<class T>
class SmartPtr
{
public:
    // RAII
    SmartPtr(T* ptr)
            :_ptr(ptr)
    {}

    ~SmartPtr()
    {
            cout << "delete[] " << _ptr << endl;
            delete[] _ptr;
    }

    // 重载运算符，模拟指针的行为，方便访问资源
    T& operator*()
    {
            return *_ptr;
    }

    T* operator->()
    {
            return _ptr;
    }

    T& operator[](size_t i)
    {
            return _ptr[i];
    }
private:
    T* _ptr;
};

double Divide(int a, int b)
{
    // 当b == 0时抛出异常
    if (b == 0)
    {
        throw "Divide by zero condition!";
    }
    else
    {
```

```
44          return (double)a / (double)b;
45      }
46  }
47
48  void Func()
49  {
50      // 这里使用RAII的智能指针类管理new出来的数组以后，程序简单多了
51      SmartPtr<int> sp1 = new int[10];
52      SmartPtr<int> sp2 = new int[10];
53
54      for (size_t i = 0; i < 10; i++)
55      {
56          sp1[i] = sp2[i] = i;
57      }
58
59      int len, time;
60      cin >> len >> time;
61      cout << Divide(len, time) << endl;
62  }
63
64  int main()
65  {
66      try
67      {
68              Func();
69      }
70      catch (const char* errmsg)
71      {
72              cout << errmsg << endl;
73      }
74      catch (const exception& e)
75      {
76              cout << e.what() << endl;
77      }
78      catch (...)
79      {
80              cout << "未知异常" << endl;
81      }
82
83      return 0;
84  }
```


3. C++标准库智能指针的使用

- C++标准库中的智能指针都在<memory>这个头文件下面，我们包含<memory>就可以是使用了，智能指针有好几种，除了weak_ptr他们都符合RAII和像指针一样访问的行为，原理上而言主要是解决智能指针拷贝时的思路不同。
- auto_ptr是C++98时设计出来的智能指针，他的特点是拷贝时把被拷贝对象的资源的管理权转移给拷贝对象，这是一个非常糟糕的设计，因为他会到被拷贝对象悬空，访问报错的问题，C++11设计出新的智能指针后，强烈建议不要使用auto_ptr。其他C++11出来之前很多公司也是明令禁止使用这个智能指针的。
- unique_ptr是C++11设计出来的智能指针，他的名字翻译出来是唯一指针，他的特点的不支持拷贝，只支持移动。如果不需要拷贝的场景就非常建议使用他。
- shared_ptr是C++11设计出来的智能指针，他的名字翻译出来是共享指针，他的特点是支持拷贝，也支持移动。如果需要拷贝的场景就需要使用他了。底层是用引用计数的方式实现的。
- weak_ptr是C++11设计出来的智能指针，他的名字翻译出来是弱指针，他完全不同于上面的智能指针，他不支持RAII，也就意味着不能用它直接管理资源，weak_ptr的产生本质是要解决shared_ptr的一个循环引用导致内存泄漏的问题。具体细节下面我们再细讲。
- 智能指针析构时默认是进行delete释放资源，这也就意味着如果不是new出来的资源，交给智能指针管理，析构时就会崩溃。智能指针支持在构造时给一个删除器，所谓删除器本质就是一个可调用对象，这个可调用对象中实现你想要的释放资源的方式，当构造智能指针时，给了定制的删除器，在智能指针析构时就会调用删除器去释放资源。因为new[]经常使用，所以为了简洁一点，unique_ptr和shared_ptr都特化了一份[]的版本，使用时 `unique_ptr<Date[]> up1(new Date[5]);shared_ptr<Date[]> sp1(new Date[5]);` 就可以管理new []的资源。
- `template <class T, class... Args> shared_ptr<T> make_shared (Args&&... args);`
- `shared_ptr` 除了支持用指向资源的指针构造，还支持 `make_shared` 用初始化资源对象的值直接构造。
- `shared_ptr` 和 `unique_ptr` 都支持了operator bool的类型转换，如果智能指针对象是一个空对象没有管理资源，则返回false，否则返回true，意味着我们可以直接把智能指针对象给if判断是否为空。
- `shared_ptr` 和 `unique_ptr` 都得构造函数都使用explicit 修饰，防止普通指针隐式类型转换成智能指针对象。

```
1 struct Date
2 {
3     int _year;
4     int _month;
5     int _day;
6
7     Date(int year = 1, int month = 1, int day = 1)
8             :_year(year)
```

```
            ,_month(month)
            ,_day(day)
    {}

    ~Date()
    {
            cout << "~Date()" << endl;
    }
};

int main()
{
    auto_ptr<Date> ap1(new Date);
    // 拷贝时，管理权限转移，被拷贝对象ap1悬空
    auto_ptr<Date> ap2(ap1);

    // 空指针访问，ap1对象已经悬空
    //ap1->_year++;

    unique_ptr<Date> up1(new Date);
    // 不支持拷贝
    //unique_ptr<Date> up2(up1);
    // 支持移动，但是移动后up1也悬空，所以使用移动要谨慎
    unique_ptr<Date> up3(move(up1));

    shared_ptr<Date> sp1(new Date);
    // 支持拷贝
    shared_ptr<Date> sp2(sp1);
    shared_ptr<Date> sp3(sp2);
    cout << sp1.use_count() << endl;
    sp1->_year++;
    cout << sp1->_year << endl;
    cout << sp2->_year << endl;
    cout << sp3->_year << endl;

    // 支持移动，但是移动后sp1也悬空，所以使用移动要谨慎
    shared_ptr<Date> sp4(move(sp1));

    return 0;
}
```

```
template<class T>
void DeleteArrayFunc(T* ptr)
{
    delete[] ptr;
```

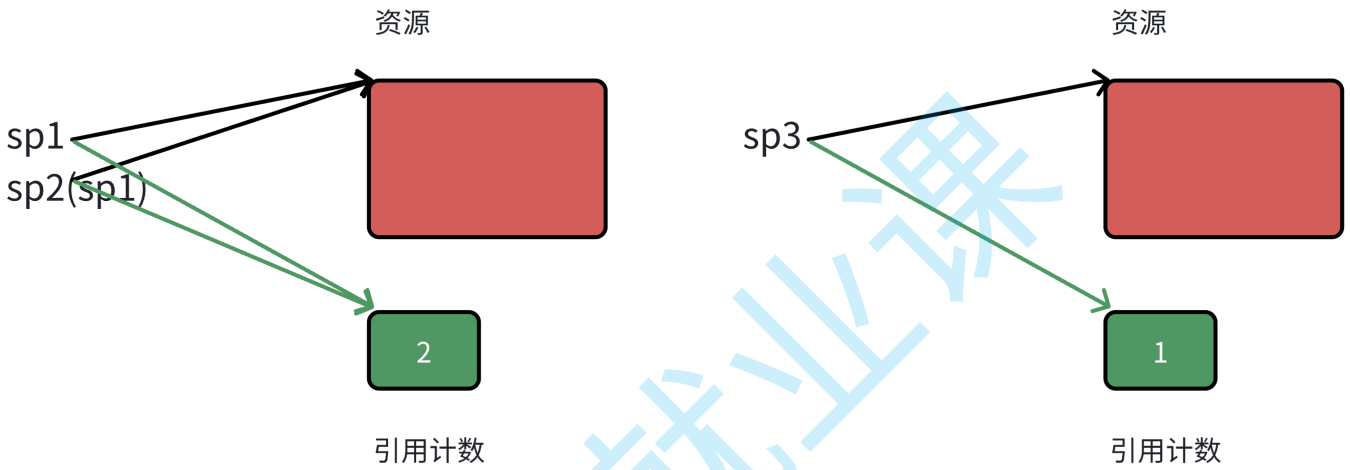
```cpp
}

template<class T>
class DeleteArray
{
public:
    void operator()(T* ptr)
    {
            delete[] ptr;
    }
};

class Fclose
{
public:
    void operator()(FILE* ptr)
    {
            cout << "fclose:" << ptr << endl;
            fclose(ptr);
    }
};

int main()
{
    // 这样实现程序会崩溃
    // unique_ptr<Date> up1(new Date[10]);
    // shared_ptr<Date> sp1(new Date[10]);

    // 解决方案1
    // 因为new[]经常使用，所以unique_ptr和shared_ptr
    // 实现了一个特化版本，这个特化版本析构时用的delete[]
    unique_ptr<Date[]> up1(new Date[5]);
    shared_ptr<Date[]> sp1(new Date[5]);

    // 解决方案2

    // 仿函数对象做删除器
    //unique_ptr<Date, DeleteArray<Date>> up2(new Date[5], DeleteArray<Date>());
    // unique_ptr和shared_ptr支持删除器的方式有所不同
    // unique_ptr是在类模板参数支持的，shared_ptr是构造函数参数支持的
    // 这里没有使用相同的方式还是挺坑的
    // 使用仿函数unique_ptr可以不在构造函数传递，因为仿函数类型构造的对象直接就可以调用
    // 但是下面的函数指针和lambda的类型不可以
    unique_ptr<Date, DeleteArray<Date>> up2(new Date[5]);
    shared_ptr<Date> sp2(new Date[5], DeleteArray<Date>());

```

```
    // 函数指针做删除器
    unique_ptr<Date, void(*)(Date*)> up3(new Date[5], DeleteArrayFunc<Date>);
    shared_ptr<Date> sp3(new Date[5], DeleteArrayFunc<Date>);

    // lambda表达式做删除器
    auto delArrOBJ = [](Date* ptr) {delete[] ptr; };
    unique_ptr<Date, decltype(delArrOBJ)> up4(new Date[5], delArrOBJ);
    shared_ptr<Date> sp4(new Date[5], delArrOBJ);

    // 实现其他资源管理的删除器
    shared_ptr<FILE> sp5(fopen("Test.cpp", "r"), Fclose());
    shared_ptr<FILE> sp6(fopen("Test.cpp", "r"), [](FILE* ptr) {
            cout << "fclose:" << ptr << endl;
            fclose(ptr);
    });

    return 0;
}
```

```
int main()
{
    shared_ptr<Date> sp1(new Date(2024, 9, 11));
    shared_ptr<Date> sp2 = make_shared<Date>(2024, 9, 11);
    auto sp3 = make_shared<Date>(2024, 9, 11);
    shared_ptr<Date> sp4;

    // if (sp1.operator bool())
    if (sp1)
            cout << "sp1 is not nullptr" << endl;

    if (!sp4)
            cout << "sp1 is nullptr" << endl;

    // 报错
    shared_ptr<Date> sp5 = new Date(2024, 9, 11);
    unique_ptr<Date> sp6 = new Date(2024, 9, 11);

    return 0;
}
```


4. 智能指针的原理

- 下面我们模拟实现了auto_ptr和unique_ptr的核心功能，这两个智能指针的实现比较简单，大家了解一下原理即可。auto_ptr的思路是拷贝时转移资源管理权给被拷贝对象，这种思路是不被认可的，也不建议使用。unique_ptr的思路是不支持拷贝。
- 大家重点要看看shared_ptr是如何设计的，尤其是引用计数的设计，主要这里一份资源就需要一个引用计数，所以引用计数才用静态成员的方式是无法实现的，要使用堆上动态开辟的方式，构造智能指针对象时来一份资源，就要new一个引用计数出来。多个shared_ptr指向资源时就++引用计数，shared_ptr对象析构时就--引用计数，引用计数减到0时代表当前析构的shared_ptr是最后一个管理资源的对象，则析构资源。




```cpp
namespace bit
{
    template<class T>
    class auto_ptr
    {
    public:
        auto_ptr(T* ptr)
                :_ptr(ptr)
        {}

        auto_ptr(auto_ptr<T>& sp)
                :_ptr(sp._ptr)
        {
                // 管理权转移
                sp._ptr = nullptr;
        }

        auto_ptr<T>& operator=(auto_ptr<T>& ap)
        {
```

```
20                // 检测是否为自己给自己赋值
21                if (this != &ap)
22                {
23                        // 释放当前对象中资源
24                        if (_ptr)
25                                delete _ptr;
26
27                        // 转移ap中资源到当前对象中
28                        _ptr = ap._ptr;
29                        ap._ptr = NULL;
30                }
31
32                return *this;
33        }
34
35        ~auto_ptr()
36        {
37                if (_ptr)
38                {
39                        cout << "delete:" << _ptr << endl;
40                        delete _ptr;
41                }
42        }
43
44        // 像指针一样使用
45        T& operator*()
46        {
47                return *_ptr;
48        }
49
50        T* operator->()
51        {
52                return _ptr;
53        }
54    private:
55            T* _ptr;
56    };
57
58    template<class T>
59    class unique_ptr
60    {
61    public:
62        explicit unique_ptr(T* ptr)
63                :_ptr(ptr)
64        {}
65
66        ~unique_ptr()
```

```
67          {
68                  if (_ptr)
69                  {
70                          cout << "delete:" << _ptr << endl;
71                          delete _ptr;
72                  }
73          }
74
75          // 像指针一样使用
76          T& operator*()
77          {
78                  return *_ptr;
79          }
80
81          T* operator->()
82          {
83                  return _ptr;
84          }
85
86          unique_ptr(const unique_ptr<T>& sp) = delete;
87          unique_ptr<T>& operator=(const unique_ptr<T>& sp) = delete;
88          unique_ptr(unique_ptr<T>&& sp)
89                  :_ptr(sp._ptr)
90          {
91                  sp._ptr = nullptr;
92          }
93
94          unique_ptr<T>& operator=(unique_ptr<T>&& sp)
95          {
96                 delete _ptr;
97                  _ptr = sp._ptr;
98                  sp._ptr = nullptr;
99          }
100     private:
101             T* _ptr;
102     };
103
104     template<class T>
105     class shared_ptr
106     {
107     public:
108         explicit shared_ptr(T* ptr = nullptr)
109                 : _ptr(ptr)
110                 , _pcount(new int(1))
111         {}
112
113         template<class D>
```

```
114         shared_ptr(T* ptr, D del)
115                 : _ptr(ptr)
116                 , _pcount(new int(1))
117                 , _del(del)
118         {}
119
120         shared_ptr(const shared_ptr<T>& sp)
121                 :_ptr(sp._ptr)
122                 , _pcount(sp._pcount)
123                 ,_del(sp._del)
124         {
125                 ++(*_pcount);
126         }
127
128         void release()
129         {
130                 if (--(*_pcount) == 0)
131                 {
132                         // 最后一个管理的对象，释放资源
133                         _del(_ptr);
134                         delete _pcount;
135                         _ptr = nullptr;
136                         _pcount = nullptr;
137                 }
138         }
139
140         shared_ptr<T>& operator=(const shared_ptr<T>& sp)
141         {
142                 if (_ptr != sp._ptr)
143                 {
144                         release();
145
146                         _ptr = sp._ptr;
147                         _pcount = sp._pcount;
148                         ++(*_pcount);
149                         _del = sp._del;
150                 }
151
152                 return *this;
153         }
154
155         ~shared_ptr()
156         {
157                 release();
158         }
159
160         T* get() const
```

```
161         {
162                 return _ptr;
163         }
164
165         int use_count() const
166         {
167                 return *_pcount;
168         }
169
170         T& operator*()
171         {
172                 return *_ptr;
173         }
174
175         T* operator->()
176         {
177                 return _ptr;
178         }
179     private:
180         T* _ptr;
181         int* _pcount;
182         //atomic<int>* _pcount;
183
184         function<void(T*)> _del = [](T* ptr) {delete ptr; };
185     };
186
187     // 需要注意的是我们这里实现的shared_ptr和weak_ptr都是以最简洁的方式实现的，
188     // 只能满足基本的功能，这里的weak_ptr lock等功能是无法实现的，想要实现就要
189     // 把shared_ptr和weak_ptr一起改了，把引用计数拿出来放到一个单独类型，shared_ptr
190     // 和weak_ptr都要存储指向这个类的对象才能实现，有兴趣可以去翻翻源代码
191     template<class T>
192     class weak_ptr
193     {
194     public:
195         weak_ptr()
196         {}
197
198         weak_ptr(const shared_ptr<T>& sp)
199                 :_ptr(sp.get())
200         {}
201
202         weak_ptr<T>& operator=(const shared_ptr<T>& sp)
203         {
204                 _ptr = sp.get();
205
206                 return *this;
207         }
```

```
    private:
        T* _ptr = nullptr;
    };
}

int main()
{
    bit::auto_ptr<Date> ap1(new Date);
    // 拷贝时，管理权限转移，被拷贝对象ap1悬空
    bit::auto_ptr<Date> ap2(ap1);

    // 空指针访问，ap1对象已经悬空
    //ap1->_year++;

    bit::unique_ptr<Date> up1(new Date);
    // 不支持拷贝
    //unique_ptr<Date> up2(up1);
    // 支持移动，但是移动后up1也悬空，所以使用移动要谨慎
    bit::unique_ptr<Date> up3(move(up1));


    bit::shared_ptr<Date> sp1(new Date);
    // 支持拷贝
    bit::shared_ptr<Date> sp2(sp1);
    bit::shared_ptr<Date> sp3(sp2);
    cout << sp1.use_count() << endl;
    sp1->_year++;
    cout << sp1->_year << endl;
    cout << sp2->_year << endl;
    cout << sp3->_year << endl;

    return 0;
}
```


5. shared_ptr和weak_ptr

5.1 shared_ptr循环引用问题

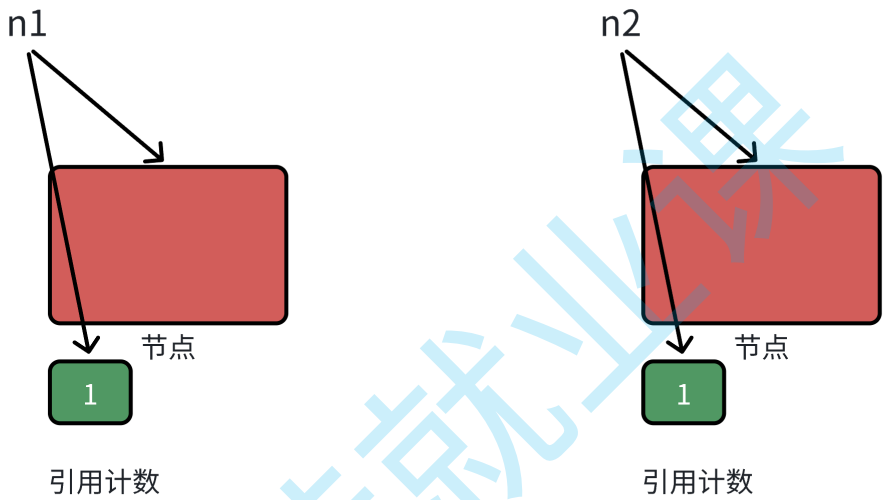
- shared_ptr大多数情况下管理资源非常合适，支持RAII，也支持拷贝。但是在循环引用的场景下会导致资源没得到释放内存泄漏，所以我们要认识循环引用的场景和资源没释放的原因，并且学会使用weak_ptr解决这种问题。
- 如下图所述场景，n1和n2析构后，管理两个节点的引用计数减到1

1. 右边的节点什么时候释放呢，左边节点中的_next管着呢，_next析构后，右边的节点就释放了。
2. _next什么时候析构呢，_next是左边节点的的成员，左边节点释放，_next就析构了。

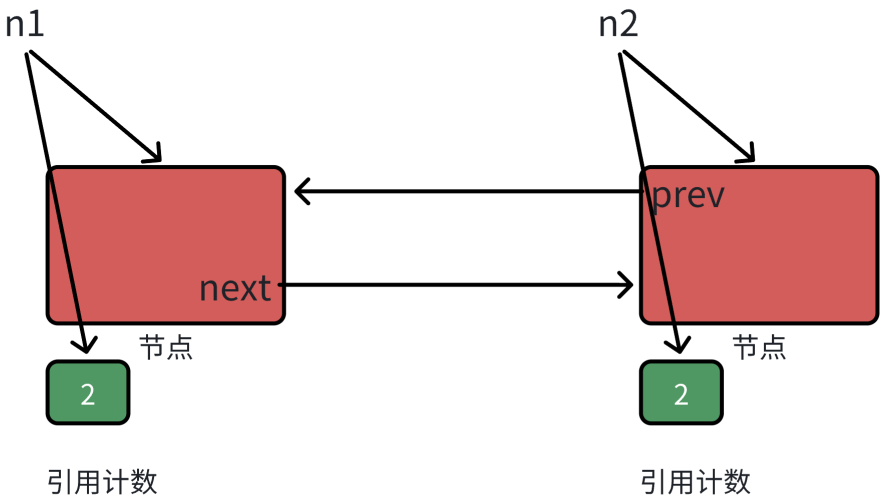
3. 左边节点什么时候释放呢，左边节点由右边节点中的_prev管着呢，_prev析构后，左边的节点就释放了。
4. _prev什么时候析构呢，_prev是右边节点的成员，右边节点释放，_prev就析构了。

- 至此逻辑上成功形成回旋镖似的循环引用，谁都不会释放就形成了循环引用，导致内存泄漏
- 把ListNode结构体中的_next和_prev改成weak_ptr，weak_ptr绑定到shared_ptr时不会增加它的引用计数，_next和_prev不参与资源释放管理逻辑，就成功打破了循环引用，解决了这里的问题

```
std::shared_ptr<ListNode> n1(new ListNode);
std::shared_ptr<ListNode> n2(new ListNode);
```




```
n1->_next = n2;
n2->_prev = n1;
```



n1和n2析构

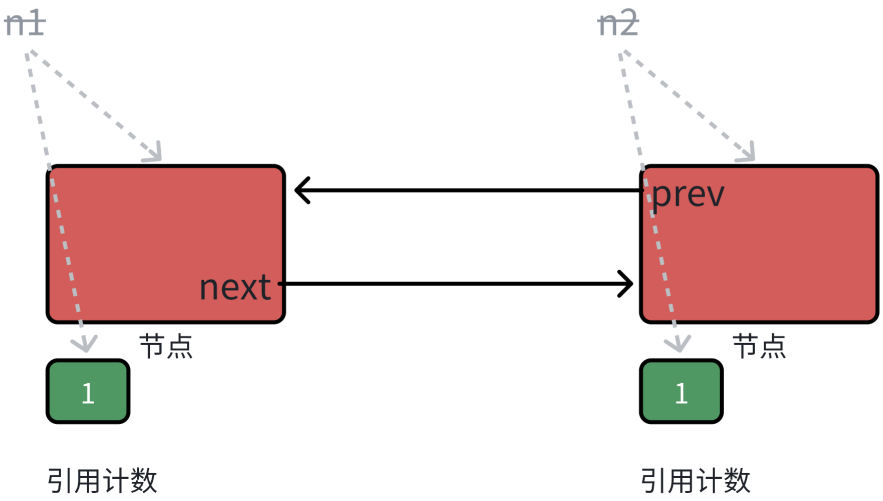


图1

```cpp
struct ListNode
{
    int _data;

    std::shared_ptr<ListNode> _next;
    std::shared_ptr<ListNode> _prev;

    // 这里改成weak_ptr，当n1->_next = n2;绑定shared_ptr时
    // 不增加n2的引用计数，不参与资源释放的管理，就不会形成循环引用了
    /*std::weak_ptr<ListNode> _next;
    std::weak_ptr<ListNode> _prev;*/

    ~ListNode()
    {
            cout << "~ListNode()" << endl;
    }
};

int main()
{
    // 循环引用 -- 内存泄露
    std::shared_ptr<ListNode> n1(new ListNode);
    std::shared_ptr<ListNode> n2(new ListNode);

    cout << n1.use_count() << endl;
    cout << n2.use_count() << endl;

    n1->_next = n2;
    n2->_prev = n1;

    cout << n1.use_count() << endl;
```

```
    cout << n2.use_count() << endl;

    // weak_ptr不支持管理资源，不支持RAII
    // weak_ptr是专门绑定shared_ptr，不增加他的引用计数，作为一些场景的辅助管理
    //std::weak_ptr<ListNode> wp(new ListNode);

    return 0;
}
```

5.2 weak_ptr

- weak_ptr不支持RAII，也不支持访问资源，所以我们看文档发现weak_ptr构造时不支持绑定到资源，只支持绑定到shared_ptr，绑定到shared_ptr时，不增加shared_ptr的引用计数，那么就可以解决上述的循环引用问题。
- weak_ptr也没有重载operator*和operator->等，因为他不参与资源管理，那么如果他绑定的shared_ptr已经释放了资源，那么他去访问资源就是很危险的。weak_ptr支持expired检查指向的资源是否过期，use_count也可获取shared_ptr的引用计数，weak_ptr想访问资源时，可以调用lock返回一个管理资源的shared_ptr，如果资源已经被释放，返回的shared_ptr是一个空对象，如果资源没有释放，则通过返回的shared_ptr访问资源是安全的。

```
int main()
{
    std::shared_ptr<string> sp1(new string("111111"));
    std::shared_ptr<string> sp2(sp1);
    std::weak_ptr<string> wp = sp1;
    cout << wp.expired() << endl;
    cout << wp.use_count() << endl;

    // sp1和sp2都指向了其他资源，则weak_ptr就过期了
    sp1 = make_shared<string>("222222");
    cout << wp.expired() << endl;
    cout << wp.use_count() << endl;

    sp2 = make_shared<string>("333333");
    cout << wp.expired() << endl;
    cout << wp.use_count() << endl;

    wp = sp1;
    //std::shared_ptr<string> sp3 = wp.lock();
    auto sp3 = wp.lock();
    cout << wp.expired() << endl;
    cout << wp.use_count() << endl;

    *sp3 += "###";
```

```
    cout << *sp1 << endl;


    return 0;
}
```

6. shared_ptr的线程安全问题

- shared_ptr的引用计数对象在堆上，如果多个shared_ptr对象在多个线程中，进行shared_ptr的拷贝析构时会访问修改引用计数，就会存在线程安全问题，所以shared_ptr引用计数是需要加锁或者原子操作保证线程安全的。
- shared_ptr指向的对象也是有线程安全的问题的，但是这个对象的线程安全问题不归shared_ptr管，它也管不了，应该有外层使用shared_ptr的人进行线程安全的控制。
- 下面的程序会崩溃或者A资源没释放，bit::shared_ptr引用计数从int*改成atomic<int>*就可以保证引用计数的线程安全问题，或者使用互斥锁加锁也可以。

```
struct AA
{
    int _a1 = 0;
    int _a2 = 0;

    ~AA()
    {
        cout << "~AA()" << endl;
    }
};

int main()
{
    bit::shared_ptr<AA> p(new AA);
    const size_t n = 100000;

    mutex mtx;
    auto func = [&]()
    {
        for (size_t i = 0; i < n; ++i)
        {
            // 这里智能指针拷贝会++计数
            bit::shared_ptr<AA> copy(p);
            {
                unique_lock<mutex> lk(mtx);
                copy->_a1++;
                copy->_a2++;
```

```
                    }
                }
        };

        thread t1(func);
        thread t2(func);

        t1.join();
        t2.join();

        cout << p->_a1 << endl;
        cout << p->_a2 << endl;

        cout << p.use_count() << endl;

        return 0;
}
```

7. C++11和boost中智能指针的关系

- Boost库是为C++语言标准库提供扩展的一些C++程序库的总称，Boost社区建立的初衷之一就是为C++的标准化工作提供可供参考的实现，Boost社区的发起人Dawes本人就是C++标准委员会的成员之一。在Boost库的开发中，Boost社区也在这个方向上取得了丰硕的成果，C++11及之后的新语法和库有很多都是从Boost中来的。
- C++ 98 中产生了第一个智能指针auto_ptr。
- C++ boost给出了更实用的scoped_ptr/scoped_array和shared_ptr/shared_array和weak_ptr等.
- C++ TR1，引入了shared_ptr等，不过注意的是TR1并不是标准版。
- C++ 11，引入了unique_ptr和shared_ptr和weak_ptr。需要注意的是unique_ptr对应boost的scoped_ptr。并且这些智能指针的实现原理是参考boost中的实现的。

8. 内存泄漏

8.1 什么是内存泄漏，内存泄漏的危害

什么是内存泄漏：内存泄漏指因为疏忽或错误造成程序未能释放已经不再使用的内存，一般是忘记释放或者发生异常释放程序未能执行导致的。内存泄漏并不是指内存在物理上的消失，而是应用程序分配某段内存后，因为设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的危害：普通程序运行一会就结束了出现内存泄漏问题也不大，进程正常结束，页表的映射关系解除，物理内存也可以释放。长期运行的程序出现内存泄漏，影响很大，如操作系统、后台服务、长时间运行的客户端等等，不断出现内存泄漏会导致可用内存不断变少，各种功能响应越来越慢，最终卡死。

```cpp
int main()
{
    // 申请一个1G未释放，这个程序多次运行也没啥危害
    // 因为程序马上就结束，进程结束各种资源也就回收了
    char* ptr = new char[1024 * 1024 * 1024];
    cout << (void*)ptr << endl;

    return 0;
}
```

8.2 如何检测内存泄漏（了解）

- linux下内存泄漏检测：linux下几款内存泄漏检测工具
- windows下使用第三方工具：windows下的内存泄露检测工具VLD使用_windows内存泄漏检测工具-CSDN博客

8.3 如何避免内存泄漏

- 工程前期良好的设计规范，养成良好的编码规范，申请的内存空间记着匹配的去释放。ps：这个理想状态。但是如果碰上异常时，就算注意释放了，还是可能会出问题。需要下一条智能指针来管理才有保证。
- 尽量使用智能指针来管理资源，如果自己场景比较特殊，采用RAII思想自己造个轮子管理。
- 定期使用内存泄漏工具检测，尤其是每次项目快上线前，不过有些工具不够靠谱，或者是收费。
- 总结一下：内存泄漏非常常见，解决方案分为两种：1、事前预防型。如智能指针等。2、事后查错型。如泄漏检测工具。