C++11 shared_ptr智能指针（超级详细）

在实际的 C++ 开发中，我们经常会遇到诸如程序运行中突然崩溃、程序运行所用内存越来越多最终不得不重启等问题，这些问题往往都是内存资源管理不当造成的。比如：

- 有些内存资源已经被释放，但指向它的指针并没有改变指向（成为了野指针），并且后续还在使用；
- 有些内存资源已经被释放，后期又试图再释放一次（重复释放同一块内存会导致程序运行崩溃）；
- 没有及时释放不再使用的内存资源，造成内存泄漏，程序占用的内存资源越来越多。

针对以上这些情况，很多程序员认为 C++ 语言应该提供更友好的内存管理机制，这样就可以将精力集中于开发项目的各个功能上。

事实上，显示内存管理的替代方案很早就有了，早在 1959 年前后，就有人提出了“垃圾自动回收”机制。所谓垃圾，指的是那些不再使用或者没有任何指针指向的内存空间，而“回收”则指的是将这些“垃圾”收集起来以便再次利用。

如今，垃圾回收机制已经大行其道，得到了诸多编程语言的支持，例如 Java、Python、C#、PHP 等。而 C++ 虽然从来没有公开得支持过垃圾回收机制，但 C++98/03 标准中，支持使用 auto_ptr 智能指针来实现堆内存的自动回收；C++11 新标准在废弃 auto_ptr 的同时，增添了 unique_ptr、shared_ptr 以及 weak_ptr 这 3 个智能指针来实现堆内存的自动回收。

所谓智能指针，可以从字面上理解为“智能”的指针。具体来讲，智能指针和普通指针的用法是相似的，不同之处在于，智能指针可以在适当时机自动释放分配的内存。也就是说，使用智能指针可以很好地避免“忘记释放内存而导致内存泄漏”问题出现。由此可见，C++ 也逐渐开始支持垃圾回收机制了，尽管目前支持程度还有限。

> C++ 智能指针底层是采用引用计数的方式实现的。简单的理解，智能指针在申请堆内存空间的同时，会为其配备一个整形值（初始值为 1），每当有新对象使用此堆内存时，该整形值 +1；反之，每当使用此堆内存的对象被释放时，该整形值减 1。当堆空间对应的整形值为 0 时，即表明不再有对象使用它，该堆空间就会被释放掉。

接下来，我们将分别对 shared_ptr、unique_ptr 以及 weak_ptr 这 3 个智能指针的特性和用法做详细的讲解，本节先介绍 shared_ptr 智能指针。

C++11 shared_ptr智能指针

实际上，每种智能指针都是以类模板的方式实现的，shared_ptr 也不例外。shared_ptr<T>（其中 T 表示指针指向的具体数据类型）的定义位于 <memory> 头文件，并位于 std 命名空间中，因此在使用该类型指针时，程序中应包含如下 2 行代码：

```
01.  #include <memory>
```

```
02.  using namespace std;
```

> 注意，第 2 行代码并不是必须的，也可以不添加，则后续在使用 shared_ptr 智能指针时，就需要明确指明 std:: 。

值得一提的是，和 unique_ptr、weak_ptr 不同之处在于，多个 shared_ptr 智能指针可以共同使用同一块堆内存。并且，由于该类型智能指针在实现上采用的是引用计数机制，即便有一个 shared_ptr 指针放弃了堆内存的“使用权”（引用计数减 1），也不会影响其他指向同一堆内存的 shared_ptr 指针（只有引用计数为 0 时，堆内存才会被自动释放）。

1、shared_ptr智能指针的创建

shared_ptr<T> 类模板中，提供了多种实用的构造函数，这里给读者列举了几个常用的构造函数（以构建指向 int 类型数据的智能指针为例）。

1）通过如下 2 种方式，可以构造出 shared_ptr<T> 类型的空智能指针：

```
01.  std::shared_ptr<int> p1;            //不传入任何实参
02.  std::shared_ptr<int> p2(nullptr);   //传入空指针 nullptr
```

注意，空的 shared_ptr 指针，其初始引用计数为 0，而不是 1。

2) 在构建 shared_ptr 智能指针，也可以明确其指向。例如：

```
01.  std::shared_ptr<int> p3(new int(10));
```

由此，我们就成功构建了一个 shared_ptr 智能指针，其指向一块存有 10 这个 int 类型数据的堆内存空间。

同时，C++11 标准中还提供了 std::make_shared<T> 模板函数，其可以用于初始化 shared_ptr 智能指针，例如：

```
01.  std::shared_ptr<int> p3 = std::make_shared<int>(10);
```

以上 2 种方式创建的 p3 是完全相同。

3) 除此之外，shared_ptr<T> 模板还提供有相应的拷贝构造函数和移动构造函数，例如：

```
01.  //调用拷贝构造函数
02.  std::shared_ptr<int> p4(p3);//或者 std::shared_ptr<int> p4 = p3;
03.  //调用移动构造函数
04.  std::shared_ptr<int> p5(std::move(p4)); //或者 std::shared_ptr<int> p5 = std::move(p4);
```

> 有关拷贝构造函数，读者可阅读《C++拷贝构造函数》一节做系统了解；有关移动构造函数，读者可阅读《C++移动构造函数》做详细了解；有关 move() 函数的功能和用法，读者可阅读《C++11 move()》一节。

如上所示，p3 和 p4 都是 shared_ptr 类型的智能指针，因此可以用 p3 来初始化 p4，由于 p3 是左值，因此会调用拷贝构造函数。需要注意的是，如果 p3 为空智能指针，则 p4 也为空智能指针，其引用计数初始值为 0；反之，则表明 p4 和 p3 指向同一块堆内存，同时该堆空间的引用计数会加 1。

而对于 std::move(p4) 来说，该函数会强制将 p4 转换成对应的右值，因此初始化 p5 调用的是移动构造函数。另外和调用拷贝构造函数不同，用 std::move(p4) 初始化 p5，会使得 p5 拥有了 p4 的堆内存，而 p4 则变成了空智能指针。

注意，同一普通指针不能同时为多个 shared_ptr 对象赋值，否则会导致程序发生异常。例如：

```
int* ptr = new int;
std::shared_ptr<int> p1(ptr);
std::shared_ptr<int> p2(ptr);//错误
```

4) 在初始化 shared_ptr 智能指针时，还可以自定义所指堆内存的释放规则，这样当堆内存的引用计数为 0 时，会优先调用我们自定义的释放规则。

在某些场景中，自定义释放规则是很有必要的。比如，对于申请的动态数组来说，shared_ptr 指针默认的释放规则是不支持释放数组的，只能自定义对应的释放规则，才能正确地释放申请的堆内存。

对于申请的动态数组，释放规则可以使用 C++11 标准中提供的 default_delete<T> 模板类，我们也可以自定义释放规则：

```
//指定 default_delete 作为释放规则
std::shared_ptr<int> p6(new int[10], std::default_delete<int[]>());

//自定义释放规则
void deleteInt(int*p) {
    delete []p;
}
//初始化智能指针，并自定义释放规则
std::shared_ptr<int> p7(new int[10], deleteInt);
```

实际上借助 lambda 表达式，我们还可以像如下这样初始化 p7，它们是完全相同的：

```
std::shared_ptr<int> p7(new int[10], [](int* p) {delete[]p; });
```

> shared_ptr<T> 模板类还提供有其它一些初始化智能指针的方法，感兴趣的读者可前往讲解 shared_ptr 的官网做系统了解。

2、shared_ptr<T>模板类提供的成员方法

为了方便用户使用 shared_ptr 智能指针，shared_ptr<T> 模板类还提供有一些实用的成员方法，它们各自的功能如表 1 所示。

表 1 shared_ptr<T>模板类常用成员方法

成员方法名	功 能
operator=()	重载赋值号，使得同一类型的 shared_ptr 智能指针可以相互赋值。
operator*()	重载 * 号，获取当前 shared_ptr 智能指针对象指向的数据。
operator->()	重载 -> 号，当智能指针指向的数据类型为自定义的结构体时，通过 -> 运算符可以获取其内部的指定成员。
swap()	交换 2 个相同类型 shared_ptr 智能指针的内容。
reset()	当函数没有实参时，该函数会使当前 shared_ptr 所指堆内存的引用计数减 1，同时将当前对象重置为一个空指针；当为函数传递一个新申请的堆内存时，则调用该函数的 shared_ptr 对象会获得该存储空间的所有权，并且引用计数的初始值为 1。
get()	获得 shared_ptr 对象内部包含的普通指针。
use_count()	返回同当前 shared_ptr 对象（包括它）指向相同的所有 shared_ptr 对象的数量。
unique()	判断当前 shared_ptr 对象指向的堆内存，是否不再有其它 shared_ptr 对象再指向它。
operator bool()	判断当前 shared_ptr 对象是否为空智能指针，如果是空指针，返回 false；反之，返回 true。

> 除此之外，C++11 标准还支持同一类型的 shared_ptr 对象，或者 shared_ptr 和 nullptr 之间，进行 ==，!=，<，<=，>，>= 运算。

下面程序给大家演示了 shared_ptr 智能指针的基本用法，以及该模板类提供了一些成员方法的用法：

```
01.  #include <iostream>
02.  #include <memory>
03.  using namespace std;
```

```
04.
05.  int main()
06.  {
07.      //构建 2 个智能指针
08.      std::shared_ptr<int> p1(new int(10));
09.      std::shared_ptr<int> p2(p1);
10.      //输出 p2 指向的数据
11.      cout << *p2 << endl;
12.      p1.reset();//引用计数减 1,p1为空指针
13.      if (p1) {
14.          cout << "p1 不为空" << endl;
15.      }
16.      else {
17.          cout << "p1 为空" << endl;
18.      }
19.      //以上操作，并不会影响 p2
20.      cout << *p2 << endl;
21.      //判断当前和 p2 同指向的智能指针有多少个
22.      cout << p2.use_count() << endl;
23.      return 0;
24.  }
```

程序执行结果为：

```
10
p1 为空
10
1
```