接口、引用计数及智能指针

万里红中央研究院

2022 年 2 月 17 日

目录

1 前言

本文主要回顾一些大家耳熟能详内容：

- **什么是接口**

为什么要有接口；

接口是用来解决什么问题的；

- **跨模块接口的设计**

跨模块接口有什么特殊之处；

在设计的时候有哪些需要特别注意的地方；

跨模块接口的扩展问题；

- **引用计数**

引用计数是用来解决什么问题的；

- **智能指针**

智能指针的原理；

为什么要使用智能指针；

以上内容其实大家平时经常听到，可能也一直在用，但从我工作这么多年的观察来看，很多程序员并没有真正理解，因此本文先把一些看上去像是常识的东西重新梳理一下，我们不但要会使用，还要理解这些东西为什么会是这样，之所以这样是为了解决什么问题，很多东西只有理解了“为什么”，才能用的得心应手。

2 什么是接口

这里简单介绍一下什么是接口，以及接口用来解决什么问题，但本文并不讲述如何设计接口的方法，这部分内容后续会有专门的文章详细介绍。

用比较通俗易懂的文字来描述，假设 A 程序员做了一个功能，B 程序员也想用，为此 A 程序员把这个功能做成一个模块，对外提供一个或者一组函数，供 B 程序员使用，这就是接口。

为什么 A 程序员应该提供接口，而不是直接把代码拷贝给 B 程序员呢？一般来说基于以下考虑：

- **B 程序员只想直接用，不关心内部是如何实现；**
- **有利于提高效率和提升产品质量；**
- **有些细节不方便公开，例如给客户的 SDK，肯定不能直接提供代码；**

如果用拷贝代码的方式，万一 A 程序员写的代码有 Bug，那 B 程序员这里必然也有同样的 Bug，后果就是大家都要修复这个 Bug，并且分别都得测试和发布，同样的事情要做多次。而如果 A 程序员提供的是一个动态库，那 A 程序员修改之后，B 程序员只需要把修改后的动态库拷贝过去即可，B 程序员自己的代码不需要做任何修改（前提是接口不变），既然 B 程序员自己的代码没有修改过，测试的工作量也会随之降低很多。

还有更多的好处这里不再一一细说，后续会有专门的文章讲这些。

3 跨二进制模块的接口的问题和注意事项

理论上任何一个函数都可以被认为是一个接口，任何接口存在的目的都是为了降低代码重复度，提高效率，最终提高产品质量。

二进制模块一般指的是动态链接库和静态链接库。在接口设计上二者有类似之处，但动态链接库要求更高，在接口的二进制兼容问题上需要考虑更多的因素。为简单起见，本文说的二进制模块，默认指的是动态链接库，在本文最后会补充说明一下动态库和静态库在接口设计上的区别，以及为什么建议接口统一以动态库的方式提供。

相对于模块内部的接口来说，跨二进制模块的接口在设计上要更加难一些，而且其中有更多需要注意的事项，跨二进制模块接口最大的问题就是二进制兼容问题，很多在模块内习以为常的做法，在跨模块的场合下是万万不能这么做的，列举如下（为了方便，本文中跨二进制模块，简写为**跨模块**）：

3.1 常用的接口导出方式

模块内部的接口，如果某一天需要增加几个函数，或者修改几个函数的声明，一般来说没什么限制，直接改了并且重新编译就行，只要能编译通过，基本上不会有问题，但同样的做法在跨模块接口上，就会有很大问题。

从一个动态库中导出的接口，一般有以下几种方式：

- **导出一系列纯 C 函数**

这是一种经典的动态库接口导出方式，除了使用起来麻烦一点，不是面向对象之外，本身没有太明显的缺点，而且这种方式导出来的接口，还能很容易的被各种其它语言调用。事实上目前很多经典的开源库都是以这种方式来导出接口的。

导出一系列纯 C 函数，最大的好处是增加函数时不会产生兼容性问题，用纯 C 函数导出接口的方式，属于不太友好，使用起来麻烦，但是风险低的接口方式。

- **导出一系列类（不建议）**

在 C++里，其实是可以直接把整个类导出去的，例如 MFC 的运行库中，大量的接口都是直接导出类，这种方式用起来很方便，但是使用中问题很多，会有各种兼容性问题，有时候有些问题还很难排查，总而言之，这种方式不建议使用，用了后患无穷，本文中不做详细描述，如果有兴趣以后专门讨论。

- **导出一个纯 C 的函数，通过这个函数创建所有接口**

这种方式其实就是微软 COM 的做法，虽然 COM 这么多年来没少被人吐槽，但客观的说，COM 的做法是目前以面向对象方式从动态链接库中导出接口的最好方式。至于网上吐槽 COM 的各种说法，大多数问题不是 COM 本身的问题，而是因为使用时候不规范导致。而之所以会使用不规范，主要还是因为很多程序员并没有真正理解 COM 导致，另外而且 COM 的有些设计确实也过于复杂。

但这些并不是没有解决方案，COM 之所以复杂，是因为 COM 的目标是要解决所有问题，而我们只需要解决自己的问题，不需要解决所有问题，因此我们只需要借用 COM 最基础的一些思想就可以，这就大大的降低了复杂度，也更便于使用。例如我们完全没必要去考虑我们导出的接口是否能被 Java 或者 Python 调用，如果有一天真的有别的语言需要调用我们的模块，大可在现有模块基础上再按需封装少量的纯 C 接口即可（一般这种情况很少出现）。

如果对 COM 不熟悉的话， 推荐阅读**潘爱民**翻译的**《COM 本质论》**，认真看一下**前 4 章**，第四章之后的暂时用不上，简单浏览一下即可（也还是值得学习一下的）。

3.2 接口修改和扩展问题

只要是跨模块的接口，无论使用什么方式导出，都必须保证一个原则：

接口发布之后，不能做任何修改！

这个修改包含：

- **不能增加或者减少函数的参数，以及修改参数类型**

这个很容易理解，假如一个 DLL 导出的函数增加了参数，并且通过升级的方式发布出去了，使用了这个 DLL 的其他模块并不知道这个 DLL 已经变了，还是按照以往的参数数量调用，必然会出现压栈和出栈不匹配的问题，导致程序崩溃。

- **基于纯虚函数的接口，不允许增加、减少虚函数，以及调整虚函数的顺序**

不允许减少和调整虚函数顺序很容易理解，一旦做了两件事，意味着虚函数表发生了变化，会导致已经发布的程序调用了不符合预期的函数，必然会引发问题。

在一个接口现有虚函数的最后增加虚函数，编译出来的动态库发布出去后，碰上老的使用者，虽然不会出问题，但这种做法也是属于不专业和危险的，在某些特殊情况下也是存在很大隐患的，不建议这样使用。在理想状况下，提供接口的模块和使用接口的模块，对于新老版本应该是可以随意组合的，最多可能某些功能不生效，但是不应该导致程序不稳定。和纯 C 函数的导出接口不同的是，如果我们直接导出一个纯 C 函数，使用者是有机会通过判断某个模块是否导出了某个函数，而决定是否做某件事的。但是如果是虚函数的话，使用者没有机会判断一个接口中是否存在某个虚函数，这会对使用者造成很大的麻烦，可能会调用了其实不存在的虚函数。所以对于虚函数组成的接口，我们应该遵循一个约定，**就是这个接口发布之后，不得做任何修改，包括在最后增加虚函数。**

那如果现有的接口不足以满足需求，确实需要在模块中增加更多的接口，以便扩展更多的功能，这种情况应该如何解决呢？

COM 推荐的做法是新写一个接口，然后在一个老接口里通过 QueryInterface 的方式去查询这个新接口，使用新接口之前先判断 QueryInterface 返回的新接口是否为空，如果不为空，就可以正常使用了。这种方法非常安全，但是看上去有点麻烦，不过有了智能指针的话，这个麻烦就不存在了，**所以我们一定要熟练使用智能指针，并且习惯使用智能指针。**

说句题外话，设计一个好的接口，其实是一件比较有挑战的事，并没有想象中这么简单，在符合 COM 规范的前提下，时间一长我们经常会看到某个接口有很多版本，因为是接口提供的函数不够用，需要扩展，但因为接口已经发布了不能改，所以只能新增加一个版本为 2 的接口并且发布出去，但很快发现新接口又不够用了，只好再写一个版本为 3 的接口，时间一长接口的版本甚至能超过 10，这不但看上去非常丑陋，使用的人也会非常郁闷，而且这样的接口用起来很没有安全感。

所以在设计接口的时候，需要做好以下几件事：

- **要把握好接口的粒度，不要把不相干的功能放在同一个接口中；**
- **要从使用者的角度出发来设计接口，预判可能出现的需求，提前实现；**
- **对于一些未来可能会用到，但现在暂时用不到的接口，目前很难提供全部的功能，但也应该提前预留扩展接口（本文后面的章节中会介绍）；**

关于接口设计的思想和方法，后续会有专门的文章和组织一些培训。

3.3 函数调用和参数使用问题

一个动态链接库的接口，对于提供者和使用者双方，其实都是一个黑盒，对于某些问题，双方其实都是不知道对方的环境和使用场景的，这里举一些例子：

- **函数调用问题**

有一个被很多程序员忽略的问题是，无论是 C 还是 C++程序，在函数调用的时候，是有很多调用方式的，但学校里的教科书一般不会讲到，平时看书或者阅读代码的时候可能也没注意，也没遇到过由于此问题引发的后果，但这问题是客观存在的。

在 Windows 上，如果我们注意观察的话会发现，Windows 上所有的 API 声明里都会带上一个 WINAPI 的宏，例如 CloseHandle 函数，函数声明为：

```
BOOL WINAPI CloseHandle(HANDLE hObject);
```

WINAPI 这个宏是这么定义的：

```
#define     WINAPI     __stdcall
```

这涉及到一个 C 和 C++的基础知识，就是函数是有多种不同的调用方式的。

下表列举一些常见的调用方式：

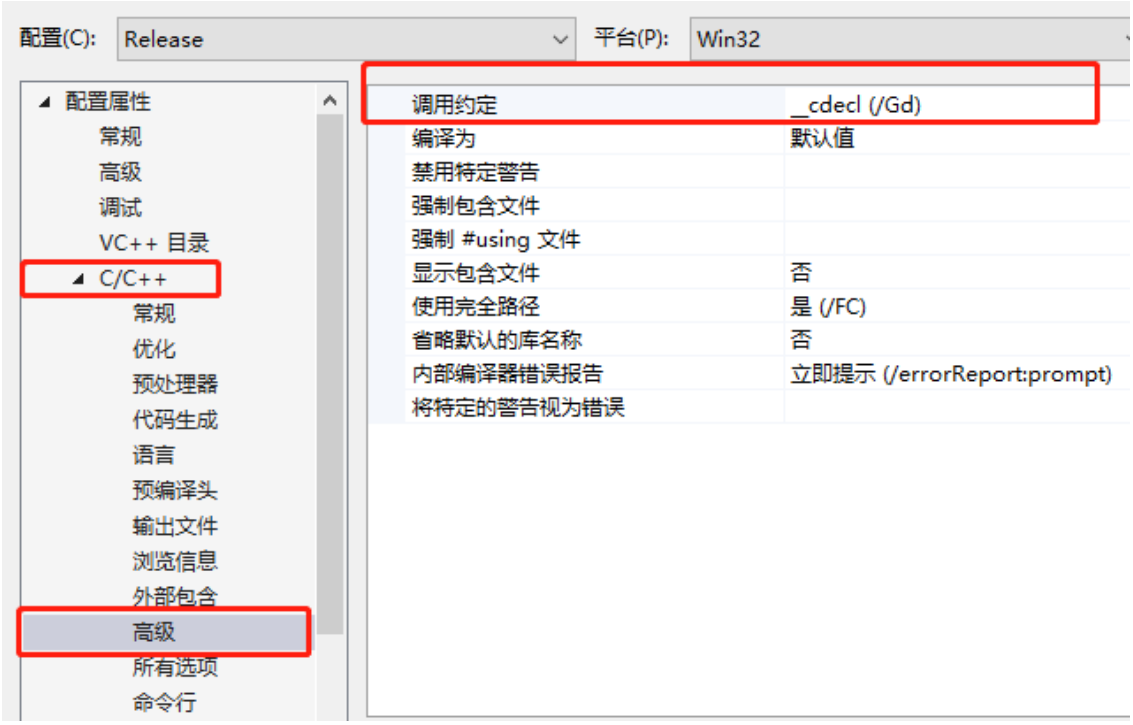
调用方式	说明
__cdecl	所有参数从右到左依次入栈，这些参数由调用者清除，称为手动清栈。被调用函数不会要求调用者传递多少参数，调用者传递过多或者过少的参数，甚至完全不同的参数都不会产生编译阶段的错误。
__stdcall	1.参数从右向左压入堆栈 2.函数被调用者修改堆栈

	3.函数名(在编译器这个层次)自动加前导的下划线,后面紧跟一个@符号，其后紧跟着参数的尺寸
__fastcall	规定将前两个参数由寄存器 ecx 和 edx 来传递，其余参数还是通过堆栈传递（从右到左)。 不同编译器编译的程序规定的寄存器不同。在 Intel 386 平台上，使用 ECX 和 EDX 寄存器。 使用__fastcall 方式无法用作跨编译器的接口。

我们可以不用熟记每种调用方式的具体细节,只需要知道,如果调用方式没用对，是会出问题的，简单的说，如果一个接口，使用者以为是__cdecl 的，但这个接口在实现的时候，其实是 __stdcall 的，那很显然当这个接口被调用的时候，一定会出问题。(**有兴趣可以自行上网搜索为什么会有这么多的调用方式**)

默认情况下，C 和 C++的函数，是以 __cdecl 方式调用的。

可能有人会问，既然大家都是用 C 或者 C++，那默认大家都用的是__cdecl，那何必多此一举要加上 WINAPI，强行把函数调用方式设置成__stdcall 呢？这是因为虽然大家都是用 C 或者 C++，但并不能保证双方默认都是__cdecl，因为事实上是可以通过设置改变编译选项，让默认的调用方式改成不是__cdecl 的（虽然我也不知道为什么有人会去改这个设置，但事实上这种事情确实发生过)，如图:



所以一个接口不能假设使用者一定会按照预想的调用方式来使用，必须明确告诉编译器每个函数的调用方式，这样才能生成正确的调用代码。

- **参数使用问题**

跨模块接口的参数，只能是以下类型：

- **标准类型**

标准类型指的是 char、int、long 以及这些类型的指针等，但需要注意的是，不同编译器对于 long 的长度是不一样的，如果使用者和模块提供者用的编译器不一样，有时候是会出问题的。为此从 C++11 开始，在 C++标准中提供了 int8_t、int16_t、int32_t、int64_t 等一系列类型，C++标准中要求在任何编译器下，对于上述这些类型的 size 都是跟名称对应的，例如 int32_t 在任何 C++编译器里，都是 32 位的，保证了不会由于编译器版本而产生任何的歧义

因此我们需要注意的是，今后代码里不要再出现直接使用 int、long 的情况，而应该使用 in32_t 这样的类型。尤其是如果我们的目标是跨平台的话，一定要使用这

些 C++标准的标准类型。

- **接口的裸指针，或者指向指针的指针**

我们看 COM 的接口就能发现，虽然 COM 提倡使用智能指针，但所有 COM 接口的函数中，如果涉及到对象的传递，参数都是形如 IUnknown *、IDispatch * 这样的裸指针，从来没有某个 COM 接口的函数中使用 IUnknownPtr 这样的智能指针作为参数，原因很简单，就是没有任何人可以保证双方使用的智能指针都是相同的，即使大家都是使用了同样的一个名为 _smart_ptr_t 的智能指针，但是不同版本的开发工具中对于这个 _smart_ptr_t 的实现未必都相同，因此智能指针本身作为参数在跨二进制的模块中传递是不合适也不安全的。

需要注意的是，这些接口都必须是稳定的，也就是说发布之后的接口，是永远不能再改的，一旦修改必然引起二进制兼容问题。

- **由标准类型或接口裸指针组成的结构体**

这个比较容易理解，不多做解释；

注意：以下类型但不仅限于这些类型，都不能作为跨模块的参数来使用：

- **任何 STL 的容器、迭代器等，以及 STL 对象的指针**
  - 这个原因和智能指针一样，没有任何人能保证双方使用的 STL 是同一个版本，以及双方模块的编译选项都是相同的；
- **任何自己写的类及类的指针**
  - 自己写的类很难保证未来永远不修改，所以直接跨模块传递类是不安全的，但可以把类抽象成一个稳定的接口是可以的，这个接口**经过 review** 之后可以合并到公共库的接口里去，公共库除了产品无关的基础库之外，也会有产

品相关模块的接口；

3.4 接口的创建和销毁问题

对于自己模块内部创建出来的对象或者内存，在模块内部使用，相对来说生命周期比较容易控制（其实如果内部模块比较复杂的话，也不容易控制），但如果在 A 模块中创建了某个对象给 B 模块使用，同时可能 C 模块也在用，这个对象的生命周期比较难处理了，到底应该让谁来销毁这个对象呢？另外一个问题是，在销毁对象的时候，应该用什么方式去销毁呢？能直接 delete 掉这个对象吗？

我们首先来看，B 模块能不能直接 delete 这个对象的问题。

从结果来看， B 模块直接 delete 别人模块里创建出来的对象，是不对的。很明显的一个问题是，B 模块怎么知道 A 模块是用什么方法创建出这个对象的呢？万一对象不是通过 new 的方式创建出来的呢？或者万一这个对象是个单实例，结果被直接 delete 了，别人还怎么用呢？就算这个对象也是 new 出来的，但我们知道，new 是 C++的一个操作符，而 C++的操作符是可以重载的，如何保证 A 模块里的 new，跟 B 模块里的 delete 是匹配的呢？显然这些都无法保证。

基于这个前提，我们必须要求一个对象的创建和销毁，最终执行的代码都在同一个二进制模块里完成，几乎所有可靠的接口都这么做的，甚至即使是纯 C 语言的接口也会这么做，一般纯 C 语言的接口，如果提供了一个 create 的函数，一定会同时提供一个 destroy 的函数用来销毁内存，保证这个内存的分配和释放都在同一个二进制模块中。

4 跨二进制模块接口问题的解决

这里主要讨论如何解决跨二进制模块接口的生存周期问题和接口扩展的问题，解决问题方法是借鉴 COM 的思想，为此我们需要先了解一下 COM 是怎么做的。

为了方便，本章中默认接口都是从**动态链接库**中导出的。

4.1 接口的创建、销毁和引用计数

4.1.1 创建和销毁

前面说了，接口的创建和销毁的原则是，创建函数和销毁函数，必须在同一个模块中，保证内存分配和释放的函数是匹配的。

创建这事很容易解决，这里主要说一下销毁。为了能让销毁接口的代码一定是在二进制模块内部执行，一般比较容易想到的方法是：**二进制模块提供一个销毁函数**，例如动态链接库在导出一个诸如 CreateObject 的用于创建接口的函数的同时，再导出一个 DestroyObject 的函数，专门用于销毁接口。这个方法看似没什么问题，仔细一想其实是不靠谱的，原因在于 C++是支持多态的，一个接口可以指向任何一个派生类，那么问题就来了，对于一个形如 void DestroyObject(IObject * pObject) 这样的函数，当我传入一个 IObject 对象，希望这个函数对这个对象进行销毁的时候，DestroyObject 怎么知道 IObject 这个接口到底指向的是哪个对象呢？所以导出一个销毁函数的方式在 C++里是不可取的。

要想以最可靠的方式把一个对象销毁掉，正确的做法是让这个对象具有自我销毁的能力，只有这个对象本身才最了解自己应该怎么销毁，这就是 COM 里所有对象

的基类 IUnknown 的 Release 函数的由来，我们可以通过调试发现，Release 执行到最后，真正销毁对象的语句，就是 delete this;

4.1.2 引用计数

解决了对象的创建和销毁必须在同一个动态库中，并且能可靠的销毁对象的问题后，另一个比较麻烦的问题就是，如果有很多地方同时在使用一个对象，那如何保证这个对象既不会被提前释放掉，导致程序崩溃，也不会因为最终没有被释放掉，导致内存泄露呢？为此几乎所有现代的语言都使用了引用计数的方式来解决问题，C++因为出现的太早，所以语言原生不支持引用计数，只能借用一些技术手段来实现。

引用计数是一个相当巧妙的设计，任何地方如果要使用某个对象，先增加引用计数，保证对象不会别人释放掉，使用完之后再减少引用计数，表示我已经用完了，对象内部会判断，当引用计数已经减少到 0 的时候，会真正释放对象。

引用计数这个设计很好的解决了多个地方使用同一个对象的时候，对象生命周期管理的问题，现代的面向对象编程语言，都引入了引用计数这个设计，虽然大多数时候我们并没有直接看到引用计数。

虽然引用计数解决了对象的生命周期问题，但在真实的使用场景里，由于使用者不理解引用计数，或者由于编写代码时候一些不好的习惯，经常会用出各种问题来，很可能使用引用计数带来的问题比带来的好处还要多。解决这一问题的方法是：

一定要使用智能指针!!!

注意，这里不是建议使用智能指针，而是一定要使用智能指针。

在智能指针章节中会详细介绍智能指针，以及如何正确使用智能指针。

4.2 接口扩展

前面说过，为了保持接口稳定可靠，正确的做法是，**发布之后的接口永远不做任何修改。**但现实存在的问题是，就算接口设计的时候再完善，也难免会遇到目前已经有的接口不能满足所有需求，需要做一些扩展的情况，因此必然涉及到接口扩展。

对于纯 C 接口的扩展不需要在此讨论，直接增加一个新的导出函数即可，这里只讨论面向对象的接口。

4.2.1 QueryInterface

QueryInterface 方式，是行业内流行的做法，安全可靠，除了相对麻烦一点之外，基本上没什么副作用。

QueryInterface 方式的做法是，当某个接口需要扩展一些功能，但是又不方便在原有接口上修改的时候，新增加一个从原有接口继承的新接口，然后让实现这个接口的类从新的接口继承，并实现新增加的纯虚函数。QueryInterface 函数中通过判断传入的接口 ID 来返回不同的接口，外部可以通过判断 QueryInterface 的返回值，以及判断 QueryInterface 得到的新接口是否为空，来判断当前使用的模块是否已经支持了新的接口，如果发现接口版本不支持，可以采取一些措施（例如提示用户升级、或者某些功能禁用），而不是程序直接崩溃。

QueryInterface 使用的时候，有一些需要注意的地方，这里做一些说明：

- **引用计数**

标准的 QueryInterface 过程，是在原有接口的基础上，通过查询得到一个新的接口，显然此时 QueryInterface 内部应该会增加引用计数，因此当查询得到的新接

口不再使用的时候，需要调用 Release 函数减少引用计数，否则就会导致这个对象最终无法被释放，引发内存泄露问题。

- **QueryInterface 之后一定要判断返回值**

如果传入的接口 ID 不正确，或者当前使用的模块版本不够新，新的接口在当前使用的模块中没有实现等各种原因，都会导致查询接口失败，所以千万不能假设 QueryInterface 一定会成功，查询完成之后一定要判断返回值，否则真碰上查询不成功的时候，因为实际接口为空，直接使用的话程序就会崩溃。

和引用计数类似，QueryInterface 虽然能解决接口扩展问题，但使用起来确实也挺麻烦，万一用完之后忘记 Release，也会造成很多问题，解决方法还是：

一定要使用智能指针!!!

- **避免不必要的接口扩展**

虽然说接口扩展是刚需，不能完全避免，但一个接口如果扩展的次数很频繁，大多因为接口设计者不够专业，设计接口时考虑不周，才导致了经常发现接口不够用，不得不扩展的情况。有时候甚至能看到一个接口在半年内，版本已经到了 5 甚至更高，这样的接口会让人缺乏安全感，使用者看到这样一个频繁扩展，短时间内版本号如此高的接口，使用的时候心里会很不踏实，很难信任这个接口。所以设计接口的时候，有一个原则是：一定要从使用者的角度来设计接口，而不是从设计者本人的角度出发设计接口。一个好的接口，必须要让使用者用的方便，用的放心。使用者现在就需要的，一定要及时满足；使用者暂时没想到的，接口设计者应该提前想到，提前实现好，以便未来使用者需要的时候随时可以用。如果由于时间关系，暂时没有精力提前实现好所有接口，那也应该留有扩展余地，以便在不改变接口的前提下扩展功能。

4.2.2 轻量级的扩展方式

前面提到接口扩展不可避免，应该提前预料需求，尽可能保证接口稳定。

但对于某些类别的接口，无论设计者考虑有多全面，也难免未来会有层出不穷的新需求出现。例如现在有个接口，是用来获取系统各种信息的，接口名称为：ISystemInfo，可以通过此接口获取 CPU 型号、内存大小、硬盘空间、显卡类型等，就算在设计的时候，已经把目前能想到的所有信息全部都封装成了接口，未来也难免会再出现某个现有的接口中没有支持的新需求。例如未来有人提出，还想获取硬盘的序列号，难道仅仅因为这个需求，我们就需要新增加一个 ISystemInfo2 的接口？如果增加之后，过几天又有人想获取当前使用的鼠标的类型，难道再增加一个 ISystemInfo3 的接口？如果是这样的话，这个接口就没法用了。

我们在设计接口的时候，其实是可以先做一个判断，某个接口未来可能要扩展的概率大致会有多大，对于那些大概率会扩展的接口，就不能使用增加接口版本的方法，而应该使用别的更好的方法。

这里介绍一下在我们现在使用的公共库中是怎么做的：

```
namespace WLHFoundation
{
    // {816F5C4E-65AD-40FB-8879-6A5281D8024F}
    COMMON_EXTERN_C const GUID IID_IBaseInfoInterface =
        { 0x816f5c4e, 0x65ad, 0x40fb, { 0x88, 0x79, 0x6a, 0x52, 0x81, 0xd8, 0x2, 0x4f } };

    // 使用者不需要接触这个接口，使用 IBaseInfo 即可
    struct IBaseInfoInterface : IBaseObject
    {
        // 获取整数类型的信息
        virtual uint64_t STDCALLTYPE GetIntInfo(int32_t nId, const void * param = nullptr) = 0;

        // 获取字符串类型的信息（拿到之后千万不要释放）
        virtual const char_type* STDCALLTYPE GetStringInfo(int32_t nId, const void* param = nullptr) = 0;

        // 通用获取方式
        virtual const void* STDCALLTYPE GetInfo(int32_t nId, const void* param = nullptr) = 0;
    };

    _BASE_SMARTPTR_TYPEDEF_RAW(IBaseInfoInterface, IBaseInfoInterfacePtr);
};
```

在这个接口中，只有几个最基本的函数，通过接口参数中的 Id 来获取不同的信息，这样即使未来出现再多的获取某个未知基础信息的需求，也不需要修改这个接口，只需要新增加一个 Id，并且增加一个和这个 Id 配套的参数即可完成接口的扩展。

但是这带来的一个问题是，这种做法虽然解决了接口的稳定性问题，但这个接口使用起来会非常不方便。可以想象的到，没有任何程序员会愿意使用这样一个接口。那有什么方法能同时满足接口的稳定性和接口使用的方便性呢？

在公共库中，我们在 IBaseInfoInterface 这个接口之上，提供了另外一个 IBaseInfo 的接口，在这个接口中对 IBaseInfoInterface 接口做了一个二次封装，并且 IBaseInfo 接口中所有的函数，都是内联函数，如图：

```
// 一些获取基础信息的接口
// 本接口 【 +++++ 多线程安全 +++++ 】
// 在进程启动的时候就会初始化，在进程运行过程中，内部数据不可更改
namespace WLHFoundation
{
    // {6A912CD5-331C-4B8D-9BF4-D84C4BFF62C1}
    COMMON_EXTERN_C const GUID IID_IBaseInfo =
        { 0x6a912cd5, 0x331c, 0x4b8d, { 0x9b, 0xf4, 0xd8, 0x4c, 0x4b, 0xff, 0x62, 0xc1 } };

    // 本接口中所有函数，全部是内联函数，可以随时扩展
    struct IBaseInfo : public IBaseInfoInterface
    {
        // 获取当前进程的位数
        __inline int32_t GetCurrentProcessBits(void);

        // 判断当前进程是否为64位进程
        __inline bool IsCurrentProcess64Bits(void);

        // 判断当前进程是否为32位进程
        __inline bool IsCurrentProcess32Bits(void);

        // 当前进程Id
        __inline uint64_t GetCurrentProcessId(void);

        // 当前线程Id
        __inline uint64_t GetCurrentThreadId(void);

        // 最好随取随用，不要保留指针，保留下来也没啥意义，万一程序运行了一天，指针指向的内容会发生变化
        // (内部用数组来保存这个字符串，因此这种情况下指针仍然有效，只是内容会变化)
        __inline const char_type* GetTodayString(void);

        // 获取可执行程序的绝对路径
        __inline const char_type* GetAppPathFile(void);
```

在 IBaseInfo 中，由于所有函数都是内联函数，没有虚函数及纯虚函数，因此在

IBaseInfo 里可以随时添加新的内联函数，而不会导致二进制兼容问题的【为什么？】，以后如果有新的获取某个信息的需求，只需要在 IBaseInfo 里增加一个内联函数就可以，不需要增加 IBaseInfo2 这样的高版本接口。

我们可以看一下，IBaseInfo 接口中的这些内联函数是如何实现的：

```
// 当前进程Id
__inline uint64_t WLHFoundation::IBaseInfo::GetCurrentProcessId(void)
{
    return this->GetIntInfo(BaseInfo::TypeInt::ID_CURRENT_PROCESS_ID);
}

// 当前线程Id
__inline uint64_t WLHFoundation::IBaseInfo::GetCurrentThreadId(void)
{
    return this->GetIntInfo(BaseInfo::TypeInt::ID_CURRENT_THREAD_ID);
}

// 获取可执行程序的绝对路径
__inline const char_type* WLHFoundation::IBaseInfo::GetAppPathFile(void)
{
    return this->GetStringInfo(BaseInfo::TypeString::ID_GET_APP_PATH_FILE);
}

// 获取当前可执行程序所在目录
__inline const char_type* WLHFoundation::IBaseInfo::GetAppDir(void)
{
    return this->GetStringInfo(BaseInfo::TypeString::ID_GET_APP_DIR);
}

// 获取可执行程序的名称
__inline const char_type* WLHFoundation::IBaseInfo::GetAppName(void)
{
    return this->GetStringInfo(BaseInfo::TypeString::ID_GET_APP_NAME);
}
```

从这里能看出，内联函数里只是调用了基类的获取某种类型的函数，真正干活的事情，还是在最终的动态库中完成。

这就实现了既能让接口保持稳定，又不给使用者带来麻烦的接口扩展方式。

4.3 智能指针

重要的事情说三遍：**一定要使用智能指针!!!**

使用 COM 思想的接口，在解决了二进制兼容问题的同时，带来了很多不方便的地方，直接使用裸指针是非常容易出错的（即使是老程序员也很容易阴沟里翻船），因此使用这类接口的时候，有一个一定要铭记在心的原则是，除非是传递参数，否则**不要在任何地方使用接口的裸指针，一定要使用智能指针。**

4.3.1 智能指针的实现原理

智能指针原理很简单，是利用 C++对象构造函数和析构函数会被自动执行的机制，帮助开发人员在合适的时机自动完成了引用计数的增加和减少，以避免出现引用计数忘记增加或者忘记释放的情况。

具体做法就是，在各种拷贝构造函数、operator = 这类操作符等处，内部会调用 AddRef，而在智能指针的析构函数中，会调用 Release，这就避免了我们编写代码的时候在执行过程中提前 return 的时候忘记减小引用计数的情况发生。

说句题外话，虽然这里说的是 COM 中使用的智能指针，但几乎所有的智能指针，（无论是 C++标准库中的，还是 boost 里的，以及某些第三方的不知名智能指针），都是基于 C++的 class 的析构函数机制去完成对于某个对象生命周期的管理，而且这也是 C++里鼓励的做法，最好在打开和关闭文件的时候也使用智能指针，这样能保证打开的文件在不再被使用之后被及时释放，以前如果大家习惯于用 C 语言的风格编写代码的话，最好能改变习惯，使用 C++的方式来编写代码。

有些多年的老程序员可能会认为，我编写的代码习惯非常好，在我手上绝不会出

现提前 return 导致内存忘记释放，或者引用计数忘记减少、文件忘记关闭的事情发生，我不需要使用 C++的手段来保证对象释放问题。但我们需要注意的一点是，这些代码未必永远是你在维护，万一未来有个经验并不是这么丰富的同事修改了这些代码，很可能就会发生各种问题，此时会耗费大量的资源去排查、修改和重新发布，并且还会对客户造成麻烦，因此我们一定要在问题还没有发生的时候就提前做些什么，来避免未来可能发生的问题。

4.3.2 接口的创建

前面曾经提到，跨二进制传递接口的使用，使用的是裸指针，不能使用类以及智能指针，也就是说，动态库不会导出一个参数是智能指针的创建函数，例如 COM 的 DLL 中导出的最终创建接口的函数声明为：

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID * ppv);
```

这个接口中，最后的那个 ppv 的参数，就是最终创建出来的对象的指针。

而在我们的公共库中，创建接口的函数也类似，参数使用的是裸指针。

但是，虽然创建接口的函数最终传递的是裸指针，但我们在创建的时候，**仍然应该使用智能指针**，使用方法为：

```
//智能指针一般的命名方式是 接口名称 + Ptr，例如 ITestObject 的智能指针，
// 默认为 ITestObjectPtr
// 智能指针变量命名方式，一般以小写的 sp 作为开头，例如 spObject；
ITestObjectPtr    spObject;
CreateObject(xxxx,  (void **) &spObject);
```

这种方式是使用 COM 的标准做法。

我们的公共库借用了 COM 的思想，创建接口的方式和 COM 的类似，但为了便于使用，并且避免可能产生的错误，对创建接口做了一些封装，此处做一些介绍。

以 **IBaseInfo** 接口为例，对应的智能指针是 **IBaseInfoPtr**，这个接口在 **WLHFoundation.dll** 中，为了跨平台方便，新的公共库的设计理念是不在代码中出现任何硬编码的文件，为此给每个动态库分配了一个 ID，都是通过 ID 来实现从某个动态库中创建接口，模块 ID 目前在配置文件中配置为 **“module.foundation”**，这个字符串常量已经在公共库的某个头文件中做了如下定义：

```
namespace Base
{
    namespace ModuleId
    {
        // 对应WLHBase模块，如果Windows下，就是WLHBase.dll，别的平台类似
        static const char_type* MODULE_BASE = _STD_TEXT("module.base");

        // 对应WLHFoundation模块，如果Windows下，就是WLHFoundation.dll，别的平台类似
        static const char_type* MODULE_FOUNDATION = _STD_TEXT("module.foundation");

        // 对应WLHNetwork模块，如果Windows下，就是WLHNetwork.dll，别的平台类似
        static const char_type* MODULE_NETWORK = _STD_TEXT("module.network");
    };
};
```

- **标准的创建方式**

```
CObjectCreator base(MODULE_FOUNDATION);

IBaseInfoPtr spBaseInfo = base.CreateObject<IBaseInfoPtr>();

// 如果是C++11及以上，可以这么创建：
// auto spBaseInfo = base.CreateObject<IBaseInfoPtr>();

if (spBaseInfo == nullptr) return;  // 注意，创建之后应该判断接口是否为空

// 调用IBaseInfo接口中的某个函数
// auto strAppName = spBaseInfo->GetAppName();
```

- **推荐的创建方式**

目前大多数接口都是在 WLHFoundation 这个模块中，为此公共库封装了一个

更为友好和简便的创建接口方式，仍然以 IBaseInfo 作为例子，也可以这么创建：

auto spBaseInfo = *WLHFoundation* :: *CreateObject*< *IBaseInfoPtr*> ();

未来如果有接口在别的动态库中实现，公共库中也会提供类似的创建接口的封装好的友好接口，并且未来每个产品会有自己产品相关的公共接口，建议也做一下此类封装，便于别的产品的同事调用。

4.3.3 接口的查询

我们知道，查询接口标准的做法是 QueryInterface，但这毕竟还是挺麻烦的，但如果使用智能指针的话，这个过程就变的非常简单了，在两个不同的智能指针之间直接赋值就可以完成接口的查询。这是因为智能指针内部实现了 operator = 这个操作符，看似直接赋值，其实内部是调用了 QueryInterface 函数。例如我们需要在 IBaseInfo 接口中查询 IProductInfo 接口，可以这么做：

```
auto spBaseInfo = WLHFoundation::CreateObject<IBaseInfoPtr>();

IProductInfoPtr spProduct = spBaseInfo;      // 此处实际上完成了QueryInterface
if (spProduct != nullptr)                    // 注意，一定要判断接口是否为空
{

}
```

再次强调，**一定要使用智能指针!!**

4.3.4 接口参数的传递

前面多次提到，跨模块传递参数的时候，不能传递智能指针，只能传递裸指针，但这不代表类似的场景下智能指针就不能使用，例如有这样一个函数：

void Test(IBaseInfo * pInfo)

我们现在有一个智能指针的变量：

IBaseInfoPtr spBaseInfo;

在调用 Test 的时候，直接把 spBaseInfo 作为参数传给 Test 函数即可，例如：

Test(spBaseInfo)

不需要通过别的方式去获取 IBaseObject 接口的裸指针，这是因为在智能指针内部，实现了这样一个 operator：

```
// Return the interface. This value may be nullptr.
//
operator Interface* () const throw()
{
    return m_pInterface;
}
```

4.3.5 智能指针使用的注意事项

早期的 STL 里，vector 容器在中间插入 Item 的时候，由于需要把插入位置后面的节点往后移动，内部实现的时候会通过&，去获取每个对象的指针，然后进行移动，然而 COM 的智能指针里正好实现了一个 & 的操作符，这个操作符是这么实现的：

```
// Returns the address of the interface pointer contained in this
// class. This is useful when using the COM/OLE interfaces to create
// this interface.
//
Interface** operator&() throw()
{
    _Release();
    m_pInterface = nullptr;
    return &m_pInterface;
}
```

函数内部第一行，是一个_Release，也就是说一进来就会先把当前智能指针给释放了，然后返回裸指针的地址，这个操作符一般用在创建接口的时候，把指向指针的指针传给创建函数，如果之前的智能指针里已经有东西了，需要先释放掉。所以说如

果一个智能指针直接放到 vector 里，在某些意想不到的情况下，这个指针可能突然就没了，这显然不是我们希望的。

现在的 STL 可能已经不存在这问题了，但建议把智能指针放到 vector 里的时候，最好还是做一些特殊处理，以避免出现预期之外的状况，毕竟 STL 不是我们自己写的，谁也不能保证当前编译器对应的 STL 版本一定没有这问题。

如果要把一个智能指针放到 vector 里去，这里有一种简单的方法可以使用，就是定义一个结构体，然后把这个只能指针的变量作为结构体的成员变量，用法如下；

这个用法是**不**安全的 ： std::vector<IBaseInfoPtr> vec;

这个用法是安全的：

```
struct BaseInfoItem
{
    IBaseInfoPtr m_spBaseInfo;
};
std::vector<BaseInfoItem>  vec;
```

除此之外，使用智能指针基本上没有别的问题了。

本节最后分享一下个人的一些习惯，我本人在使用一个东西的时候，都会花点时间去仔细看一下这东西内部是如何实现的，例如 STL、智能指针等，一边学习一边还会思考，为什么作者要这么设计，这么设计有什么好处和坏处等问题，只有真正了解一个库，才能最大程度的把这个库用好，很多时候我们觉得一个东西复杂，其实并不是这东西真的多复杂，而仅仅只是我们觉得它复杂，当你真正去尝试理解它的时候，其实很多东西并没有这么复杂，有时候只是自己被吓住了。

5 为什么是动态库

一般来说我们要把自己实现的功能提供给别人使用，有以下几种方法：

- **直接把源代码拷贝给别人 – 不推荐**
- **做成静态库 – 不推荐**
- **做成动态库**

本章主要介绍一下这几种方式的区别和特点，以及说明为什么公共库是以动态库的方式提供，并且也强烈建议产品中各模块都做成动态库，然后封装成接口，以便各模块之间能互相调用。

5.1 拷贝代码方式和静态库方式的缺陷

拷贝代码的方式和静态库的方式，本质上没什么区别，归根到底这两种方式都不是推荐的方式，都有很大的不足，列举如下：

- **不利于提高产品质量和员工能力成长**

无论是拷贝代码还是静态库，万一代码里有 Bug，都会连 Bug 一起拷贝到别人那里去，修复 Bug 的时候，所有用到了这些代码或者静态库的地方都得跟着改，这些代码的任何有价值的改动，使用者都必须做一些额外的工作才能从这些改动中受益，所以一般情况下，如果使用了拷贝代码或者静态库的方式，大部分程序员会很很自然的形成这种观点：既然这东西目前没啥问题，那就别去动他了。久而久之产品就会落后于时代，质量也无法提升，并且这种做法会误导刚参加工作的新同学，如果他们长年累月看到的都是这样的使用方式，他们会认为这种做法是对的，这不利于新同学的成长。

- **静态库的特有缺陷**

使用静态库，基本上没有任何好处，有时候甚至还不如直接拷贝代码。如果一个公共模块以静态库的方式提供，对这个静态库的维护者来说，是一件工作量非常大的事情。作为一个静态库的提供者，并不知道使用者使用的是什么版本的编译器（即使都是使用 Visual Studio，也有很多版本），因此很大可能需要使用不同编译器分别编译出不同版本的静态库，同时每种编译器编译的时候，还有 Debug 和 Release 之分，每配置里还有“多线程”和“多线程 DLL”之分，同时还有 x86 和 x64 之分，这几个东西是一个乘法的关系，如果要提供某一个平台下所有门类的静态库，很大可能要提供多达几十个不同的 LIB 文件，这是一件非常麻烦的事情，甚至提交到 Git 上都会占用大量的磁盘空间，使用者下载到本地，又会占用大量的本地磁盘空间。

万一某一天这个静态库被发现了一个 Bug，而此时这个静态库已经在几十个产品中使用了，那么当静态库的维护者修复了 Bug 之后，这几十个产品必须全部重新链接、测试和发布，这开销非常大，从这一点来说，提供静态库还不如直接拷贝代码。

为此我们在项目中，要避免使用静态库，当然也不要拷贝代码，这两种方法都是不专业的做法。

5.2 动态库的优势

动态库基本上没有前面说的这些问题，如果动态库中有 Bug，修复之后各产品不用做任何修改，直接把新的动态库替换到安装包里，简单测试一下即可发布，各动态库也完全可以独立迭代，直接通过单个文件升级的方式发布出去，如果有别的产品需要使用到一些功能，直接把动态库拷贝走就可以。

使用动态库，还有利于提高开发效率，例如我们要发布新产品，如果某个动态库

在本次发布中根本就没有修改过，那有什么理由要去重新编译这个动态库呢？仅从编译时间都能节省很多。而且使用动态库，大家约定好接口后就可以各自独立进行开发，各自开发好之后东西放到一起就可以进行测试，测试中间发现了 Bug，谁的动态库有 Bug 自己修改了，然后就可以立刻进行测试，不会影响到别的人，如果使用静态库，别的拿到新的静态库，还得重新编译、提测，效率大幅降低。

使用动态库对接口设计的要求会更高一些，难度会更大一些，但是带来的好处也非常明显，我们不能因为这事情有点难度就不去做，总和考虑整体成本，使用动态库必然是性价比最高的方案。

6 总结及建议

程序员是一个比较忙碌的职业，日常工作量也不小，但有时候我们回头过来看，是不是所有时间都花的有价值？其实有些时间是被浪费的，例如由于编码的时候不仔细、考虑不周，产品发布出去之后在客户那里发现问题，反馈回来，然后定位、修复、测试和发布，整个周期会比较长，一通折腾下来花了大量的时间和精力，如果当初编码的时候能多注意一点，低级错误少一些，最终能节约大量的资源，开发效率更快，产品质量更高，客户更满意，我们自己也能更加轻松。

还有一个问题就是，我们一定要有意识的避免在工作中的重复劳动，拷贝代码也是属于重复劳动，拷贝代码虽然一时看来很快，但万一出现问题后患无穷。在编码阶段，如果某个功能可能会有很多地方使用，千万不要到处编写相同的代码，至少也应该封装成一个函数，然后所有地方都去调用这个函数；最好的做法是把这个代码放到某个动态库里，所有地方都使用这个动态库中的函数，这么做短期看可能有点麻烦，

长期看会带来巨大的好处，这里举一个例子：

多年前在某公司工作的时候，为了帮助改善产品，一般产品都会有一个所谓的“用户体验改善计划”，说白了就是用户做某些操作的时候，往服务器上打个点，便于后续统计产品使用上的各种问题，一般做法都是在客户端带着某些参数访问一个 URL，然后服务器根据这个 URL 的不同参数进行统计。当时几乎所有人都是直接在代码里硬编码的去访问这个 URL，因此在产品中随处可见类似于这样的代码：

URLDownloadToFile(“http://stat.aaaa.com?func=xxx&action=xxxx”，xxx)

这个看上去也没啥问题，代码也很简单。

但如果我们多想几步，如果某一天发生这些事情，那会怎么办？

- **URL 需要修改**
- **用户反馈要求提供关闭统计功能**

如果真的要这样，那意味着所有使用了这功能的地方，全部都要修改，而一个产品中需要统计的地方非常多，并且分布在不同的模块中（不同动态库中），因此会有大量的模块需要修改、编译、测试和发布，非常耗时耗力，浪费了大量的人力。然而这件事情有什么技术含量？做起来有什么意义？能给我们带来什么成就感？

在当时，几乎所有团队的同事都是用硬编码的方式来实现这种打点统计，只有我们团队，从一开始就觉得这东西未来有可能会发生变化，因此从一开始就是在一个 DLL 里导出了一个类似于这样的函数：

void __stdcall SendStat(const char * strFuncId，const char *strAction)

在我们团队负责的所有需要统计的地方，都调用这个函数来实现统计。当某一天更换 URL，或者要关闭统计的事情真的发生之后，全公司所有客户端的团队都在热

火朝天的加班修改代码，只有我们团队在这个 DLL 里加了几行代码，问题就全部解决了。类似的还有第三方库，曾经大家计算 MD5，或者做加解密的时候，都是直接把 OpenSSL 的静态库链接到自己的工程中，我们团队是把这些功能封装在某个 DLL 里，并导出一个通用的接口供产品使用。某一天客户通过一些工具扫描，发现产品里很多模块使用的第三方库太老，存在很多多年前的漏洞，要求必须都换成最新的第三方库，涉及到的模块非常多，又是大家热火朝天的修复，只有我们负责的产品，把那个最终用到第三方库的 DLL 升级了一下，问题就全部解决了。

归根到底，如果我们想让产品质量高一点，自己也轻松愉快一点的话，就应该在设计和编码阶段尽量多考虑各种情况，多考虑一些问题并不会花掉我们很多时间，有可能就只是多想一步的事，但是带来的回报是非常大的，就以上面那个例子为例，多想一步，提前把这功能放在动态库中，能有多大工作量？就是一件顺手就完成的事。

总而言之，一次性把事情做正确了，才是效率最高的工作方式！