20180403_weak_ptr 的作用

```
1- 可以解决环的问题
2- 可以判断shared_ptr 是否为空，即使为空,weak_ptr 还是能够使用
3- 对象被析构了，weakptr会自动等于nullptr
4- weakptr可以还原成sharedptr而不会让引用计数错乱
5- weakptr 不会增加shared_ptr 的引用计数
6- 一般和shared_ptr 一起使用
```

补充1：为什么不能检测`shared_ptr`是否为NULL？

因为这就是错的。`shared_ptr`和`nullptr`相等，并不代表所引用的对象不存在了。

`shared_ptr`允许有一个“empty”状态，代表`shared_ptr<T>`对象的实例本身存在，但并不指向一个有效的`T`。`shared_ptr`重载了`==`和`=`来表示以下意义：

- `shared_ptr`和`nullptr`判等，代表判断是否处在empty状态；
- `shared_ptr`被赋值为`nullptr`，不代表`shared_ptr`实例本身没了，而是把这个`shared_ptr`实例的状态改为empty。
- 一个已经处于empty状态的`shared_ptr`仍可以被继续赋值成为一个有效、有值的`shared_ptr`。

如果有两个`shared_ptr`同时指向同一个`T`在内存中的实例，那么任意一个`shared_ptr`和`nullptr`判等成功，都不能说明指向的对象已经不存在（被销毁）了！

任何一个具体用例中，都必然有多个`shared_ptr`指向同一个实例（如果没有多个你shared干嘛）。所以通过判空某 1 个`shared_ptr`去确定被指向的对象是否已被销毁，这个在语义和实效上都绝无任何正确（哪怕是“蒙对了”）的可能！

补充2：`shared_ptr`到底是什么？

首先`shared_ptr`虽然名为“智能指针”，但其实他不是指针。一个`shared_ptr<T>`定义出来的东西，只是一个单纯的变量，存储了一个实例化出来的`class`。其本质上等同于以下代码：

```
struct MyClass
{
public:
    int MyValue;
    MyClass(int my_value) {
        MyValue = my_value;
    }
}

void main()
{
    auto a = new MyClass(1);
    // a 是一个单纯的变量，不是指针

    // a 天然与 main() 函数拥有等同的生命周期
```

```
}
```

为什么`shared_ptr`能够像指针一样使用`*`、`&`、`->`等运算符，是因为`shared_ptr`类把这些运算符重载了，从而让程序编写者"看起来"像是在用指针而已。

一个非指针的局部变量，如果不靠运算符重载，你是不能判`nullptr`的。如同你不能对一个`int`判`nullptr`一样。

`shared_ptr`判`nullptr`由于运算符重载，其本质意义都变了，必须具体分析，而不能想当然！

嗯……非读者注意：`StrBlob`仅仅是那本书里的一个范例类，其内部用`shared_ptr`维护其管理的`vector<string>`集合本身的生存期。与C++本身无关。

阻止用户访问一个不再存在的`vector`，这个用`shared_ptr`没问题，能做到。事实上很多情况下，`weak_ptr`全改`shared_ptr`并不会立刻就炸。

但是`shared_ptr`就意味着你的引用和原对象是一个**强联系**。你的引用不解开，原对象就不能销毁。滥用强联系，这在一个运行时间长、规模比较大，或者是资源较为紧缺的系统中，极易造成隐性的内存泄漏，这会成为一个灾难性的问题。

更糟的是，滥用强联系可能造成**循环引用**的灾难。即：`B`持有指向`A`内成员的一个`shared_ptr`，`A`也持有指向`B`内成员的一个`shared_ptr`，此时`A`和`B`的生命周期互相由对方决定，事实上都无法从内存中销毁。——这还仅仅是一个简单的情况。如果存在间接的强引用，或者是多于两个实例之间的强引用，这个相关的bug解起来将是灾难性的。

`shared_ptr`是C++内存管理机制的一种放松。但放松绝不意味着可以滥用，否则最后的结局恐怕不会比裸指针到处申请了不释放更好。

必须明确：从语义上来说，`shared_ptr`代表了一种**对生命周期的自动推断**。其本质的意义是：`A`持有`B`的`shared_ptr`，代表`B`的生命周期反而完全覆盖了`A`。以树形结构的层级来理解，**指针持有者是下级，指针指向的目标反而才是上级**——下级短命，上级长存；上级不存，下级焉附。

从这个意义上，有些关系你用`shared_ptr`就表示不了了：

- 隶属关系中，祖先到子孙的关系。比如一个对象容器，具体对象找其隶属的容器可以用`shared_ptr`，但是容器去找具体的对象则不行，因为容器不能要求对象生存的比自己更久。
- 生命周期没有本质关联的两个无关对象。例如一个全局事件管理器（订阅者模型），事件订阅者构造时把自己注册进来，析构时把自己解注册掉。管理器要维护到达这个对象的指针（从而发送消息），但绝对不允许染指对象的生命周期，对象的析构需要无视订阅者的存在，只由其他业务所必须的强引用来控制。

这种时候就是`weak_ptr`的用处。`weak_ptr`提供一个（1）能够确定对方生存与否（2）互相之间生命周期无干扰（3）可以临时借用一个强引用（在你需要引用对方的短时间内保证对方存活）的智能指针。

而`weak_ptr`要求程序员在运行时确定生存并加锁，这也是逻辑上必须的本征复杂度——如果别人活的比你短，你当然要：（1）先确定别人的死活（2）如果还活着，就给他续个命续到你用完了为止。

事实上**弱引用**和**强引用**这个概念，在有GC的语言中也是一个需要注意的问题。以C#为例：

```
public class EventDisposer
{
    private Dictionary<int, WeakReference<IEventListener>> pendingEvents;

    private void Dispose(int event_id)
```

```
    {
        var reference = pendingEvents[event_id];
        IEventListener context;
        if (reference.TryGetTarget(out context))
        {
            // context is valid from here to the end of function
            context.Trigger();
        }
        else
        {
            // context has already been destroyed...
        }
    }
}
```

这个程序使用`WeakReference<IEventListener>`替代强引用的`IEventListener`，从而使接受事件分发的对象，在生存周期上可以获得彻底销毁的自由，而不和事件分发者产生什么必然的关联。

有两个好处 1、对象被析构了，weakptr会自动等于nullptr 2、weakptr可以还原成sharedptr而不会让引用计数错乱

这两者普通指针都是做不到的。

---

weak_ptr的用法楼上几位都讲的很清楚了。俺补充一下实现原理吧。 weak_ptr 只能由shared_ptr或者其它的weak_ptr构造。参见 std::weak_ptr::weak_ptr 关于shared_ptr的实现，可以参见 make a shared_ptr from scratch . weak_ptr和shared_ptr共享一个引用计数对象，在引用计数对象上增加一个weak_count, 但不增加ref_count.引用计数对象当ref_count减至zero时会销毁其管理的资源，weak_ptr可以通过ref_count是否为0来判断指向的资源是否可用。当ref_count和weak_count都为0时引用计数对象会销毁其自身。作者：QAMichaelPeng 链接：https://www.zhihu.com/question/26851369/answer/34392105来源：知乎著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

首先，不要把智能指针和裸指针的区别看得那么大，它们都是指针。因此，我们可以把智能指针和裸指针都统称为指针，它们共同的目标是通过地址去代表资源。既然指针能代表资源，那么不可避免地会涉及资源的**所有权**问题。在选择具体指针类型的时候，通过问以下几个问题就能知道使用哪种指针了。

- 指针是否需要拥有资源的所有权？

如果指针变量需要绑定资源的所有权，那么会选择unique_ptr或shared_ptr。它们可以通过RAII完成对资源生命期的自动管理。如果不需要拥有资源的所有权，那么会选择weak_ptr和raw pointer，这两种指针变量在离开作用域时不会对其所指向的资源产生任何影响。

- 如果指针拥有资源的所有权(owning pointer)，那么该指针是否需要独占所有权？

独占则使用unique_ptr(人无我有，人有我丢)，否则使用shared_ptr(你有我有全都有)。这一点很好理解。

- 如果不拥有资源的所有权(non-owning pointer)，那么指针变量是否需要在适当的时候感知到资源的有效性？

```
auto p = make_shared<int>(1);
auto result = f(p.get());
```

这样会衍生出另外一个问题，为何unique_ptr不能和weak_ptr配合？ 这是因为unique_ptr是独占所有权，也就是说资源的生命期等于指针变量的生命期，那么程序员可以很容易通过指针变量的生命期来判断资源是否有效，这样weak_ptr就不再有必要了。而相对来说，shared_ptr则不好判断，特别是多线程环境下。

另外，很多人说weak_ptr的作用是可以破除循环引用，这个说法是对的，但没有抓住本质(裸指针也可以破除，那为何要用weak_ptr?)。写出循环引用的原因是因为程序员自己没有理清楚资源的所有权问题。

你要销毁一个对象，但是有其他引用指向这个对象，怎么给其他指针一个机会判断这个对象是否存在，而又不因为使用野指针产生错误呢？ 比如要能调用这样的代码： if(p.alive())//dosth

如果p是原始指针基本不行。

如果没有weak_ptr，对应的做法大概是这样：

```
template<class T>
class Holder{
  unsigned refcount,holdcount;
  byte buf[sizeof(T)];
  void release(){
    --holdcount;
    if(!--refcount)
    reinterpret_cast<T*>(buf)->~T();
  }
};
```

在所有引用释放这个对象之前，你有sizeof(T)的空间释放不了, 而且这么做还不兼容重载opeator new。

weak_ptr和share_ptr通过这样以及间接结构指向对象：

```
ref_impl{
  T* val;
  unsigned refcount,holdcount;
};

class weak_ptr{
  ref_impl* p_;
};
```

那么删除指向的对象时依然可以保留ref_impl。通过ref_impl的值判断对象是否释放。

弱引用指针的引入目的是为了在更细的粒度上区分不同类型的指针,以实现更好的自动内存分配策略.

弱引用指针，是为了 解决引用环问题。引用环是 引用计数 技术的重大缺陷。

---

shared_ptr就是gcd，指针对象就是房子，shared_ptr有所有权和使用权，开心了还可以把资源分享给屌丝(weak_ptr)。weak_ptr虽然也能使用资源，但最多也就是七十年大产权。lock相当于打开房门 当屌丝打开房门发现屋里什么都没了(nullptr),也属于正常情况 因为shared_ptr强拆了(reset)作者：麦穗链接：https://www.zhihu.com/question/26851369/answer/64209984来源：知乎著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

解决引用计数型智能指针的循环引用问题。常见的使用场景：你并不关心对象的生命周期，还想调用这个对象。
