Item 20: 当std::shard_ptr可能悬空时使用std::weak_ptr

自相矛盾的是，如果有一个像`std::shared_ptr`的指针但是不参与资源所有权共享的指针是很方便的。换句话说，是一个类似`std::shared_ptr`但不影响对象引用计数的指针。这种类型的智能指针必须要解决一个`std::shared_ptr`不存在的问题：可能指向已经销毁的对象。一个真正的智能指针应该跟踪所值对象，在悬空时知晓，悬空(*dangle*)就是指针指向的对象不再存在。这就是对`std::weak_ptr`最精确的描述。

你可能想知道什么时候该用`std::weak_ptr`。你可能想知道关于`std::weak_ptr`API的更多。它什么都好除了不太智能。`std::weak_ptr`不能解引用，也不能测试是否为空值。因为`std::weak_ptr`不是一个独立的智能指针。它是`std::shared_ptr`的增强。

这种关系在它创建之时就建立了。`std::weak_ptr`通常从`std::shared_ptr`上创建。当从`std::shared_ptr`上创建`std::weak_ptr`时两者指向相同的对象，但是`std::weak_ptr`不会影响所指对象的引用计数：

```
auto spw =                    // after spw is constructed
std::make_shared<Widget>(); // the pointed-to Widget's
                              // ref count(RC) is 1
                              // See Item 21 for in on std::make_shared
…
std::weak_ptr<Widget> wpw(spw); // wpw points to same Widget as spw. RC remains 1
…
spw = nullptr;                    // RC goes to 0, and the
                                  // Widget is destroyed.
                                  // wpw now dangles
```

`std::weak_ptr`用**expired**来表示已经dangle。你可以用它直接做测试：

```
if (wpw.expired()) … // if wpw doesn't point to an object
```

但是通常你期望的是检查`std::weak_ptr`是否已经失效，如果没有失效则访问其指向的对象。这做起来比较容易。因为缺少解引用操作，没有办法写这样的代码。即使有，将检查和解引用分开会引入竞态条件：在调用**expired**和解引用操作之间，另一个线程可能对指向的对象重新赋值或者析构，并由此造成对象已析构。这种情况下，你的解引用将会产生未定义行为。

你需要的是一个原子操作实现检查是否过期，如果没有过期就访问所指对象。这可以通过从`std::weak_ptr`创建`std::shared_ptr`来实现，具体有两种形式可以从`std::weak_ptr`上创建`std::shared_ptr`，具体用哪种取决于`std::weak_ptr`过期时你希望`std::shared_ptr`表现出什么行为。一种形式是`std::weak_ptr::lock`，它返回一个`std::shared_ptr`，如果`std::weak_ptr`过期这个`std::shared_ptr`为空：

```
std::shared_ptr<Widget> spw1 = wpw.lock();  // if wpw's expired, spw1 is null

auto spw2 = wpw.lock();                     // same as above, but uses auto
```

另一种形式是以`std::weak_ptr`为实参构造`std::shared_ptr`。这种情况中，如果`std::weak_ptr`过期，会抛出一个异常：

```
std::shared_ptr<Widget> spw3(wpw);          // if wpw's expired, throw
std::bad_weak_ptr
```

但是你可能还想知道为什么`std::weak_ptr`就有用了。考虑一个工厂函数，它基于一个UID从只读对象上产出智能指针。根据Item18的描述，工厂函数会返回一个该对象类型的`std::unique_ptr`：

```
    std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

如果调用`loadWidget`是一个昂贵的操作（比如它操作文件或者数据库I/O）并且对于ID来重复使用很常见，一个合理的优化是再写一个函数除了完成`loadWidget`做的事情之外再缓存它的结果。当请求获取一个Widget时阻塞在缓存操作上这本身也会导致性能问题，所以另一个合理的优化可以是当Widget不再使用的时候销毁它的缓存。

对于可缓存的工厂函数，返回`std::unique_ptr`不是好的选择。调用者应该接收缓存对象的智能指针，调用者也应该确定这些对象的生命周期，但是缓存本身也需要一个指针指向它所缓的对象。缓存对象的指针需要知道它是否已经悬空，因为当工厂客户端使用完工厂产生的对象后，对象将被销毁，关联的缓存条目会悬空。所以缓存应该使用`std::weak_ptr`，这可以知道是否已经悬空。这意味着工厂函数返回值类型应该是`std::shared_ptr`，因为只有当对象的生命周期由`std::shared_ptr`管理时，`std::weak_ptr`才能检测到悬空。

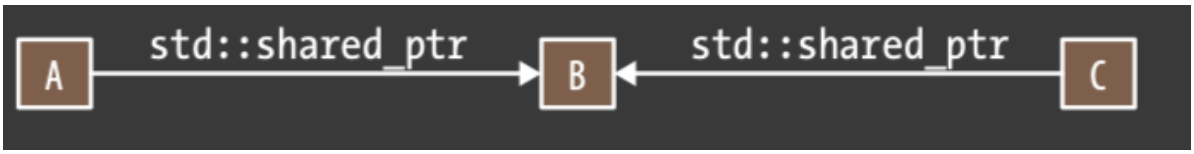
下面是一个临时凑合的`loadWidget`的缓存版本的实现：

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
    static std::unordered_map<WidgetID,
                              std::weak_ptr<const Widget>> cache; // 译者注：这里是
高亮
    auto objPtr = cache[id].lock();     // objPtr is std::shared_ptr
                                        // to cached object
                                        // (or null if object's not in cache)
    if (!objPtr) {                      // if not in cache
        objPtr = loadWidget(id);        // load it
        cache[id] = objPtr;             // cache it
    }
    return objPtr;
}
```

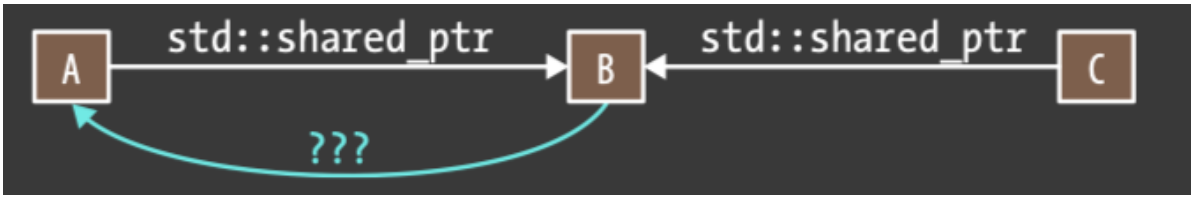
这个实现使用了C++11的hash表容器`std::unordered_map`，但是需要的`WidgetID`哈希和相等性比较函数在这里没有展示。

`fastLoadWidget`的实现忽略了以下事实：cache可能会累积过期的`std::weak_ptr`(对应已经销毁的`Widget`)。可以改进实现方式，但不要花时间在不会引起对`std :: weak_ptr`的深入了解的问题上，让我们考虑第二个用例：观察者设计模式。此模式的主要组件是subjects（状态可能会更改的对象）和observers（状态发生更改时要通知的对象）。在大多数实现中，每个subject都包含一个数据成员，该成员持有指向其observer的指针。这使subject很容易发布状态更改通知。subject对控制observers的生命周期（例如，当它们被销毁时）没有兴趣，但是subject对确保observers被销毁时，不会访问它具有极大的兴趣 。一个合理的设计是每个subject持有其observers的`std::weak_ptr`，因此可以在使用前检查是否已经悬空。

作为最后一个使用`std::weak_ptr`的例子，考虑一个持有三个对象A、B、C的数据结构，A和C共享B的所有权，因此持有`std::shared_ptr`：



假定从B指向A的指针也很有用。应该使用哪种指针？



有三种选择：

- **原始指针**。使用这种方法，如果A被销毁，但是C继续指向B，B就会有一个指向A的悬空指针。而且B不知道指针已经悬空，所以B可能会继续访问，就会导致未定义行为。
- **`std::shared_ptr`**。这种设计，A和B都互相持有对方的`std::shared_ptr`，导致`std::shared_ptr`在销毁时出现循环。即使A和B无法从其他数据结构被访问（比如，C不再指向B），每个的引用计数都是1。如果发生了这种情况，A和B都被泄露：程序无法访问它们，但是资源并没有被回收。
- **`std::weak_ptr`**。这避免了上述两个问题。如果A被销毁，B还是有悬空指针，但是B可以检查。尤其是尽管A和B互相指向，B的指针不会影响A的引用计数，因此不会导致无法销毁。

使用`std::weak_ptr`显然是这些选择中最好的。但是，需要注意使用`std::weak_ptr`打破`std::shared_ptr`循环并不常见。在严格分层的数据结构比如树，子节点只被父节点持有。当父节点被销毁时，子节点就被销毁。从父到子的链接关系可以使用`std::unique_ptr`很好的表征。从子到父的反向连接可以使用原始指针安全实现，因此子节点的生命周期肯定短于父节点。因此子节点解引用一个悬垂的父节点指针是没有问题的。

当然，不是所有的使用指针的数据结构都是严格分层的，所以当发生这种情况时，比如上面所述cache和观察者情况，知道`std::weak_ptr`随时待命也是不错的。

从效率角度来看，`std::weak_ptr`与`std::shared_ptr`基本相同。两者的大小是相同的，使用相同的控制块（参见Item 19），构造、析构、赋值操作涉及引用计数的原子操作。这可能让你感到惊讶，因为本Item开篇就提到`std::weak_ptr`不影响引用计数。我写的是`std::weak_ptr`不参与对象的*共享所有权*，因此不影响指向对象的引用计数。实际上在控制块中还是有第二个引用计数，`std::weak_ptr`操作的是第二个引用计数。想了解细节的话，继续看Item 21吧。

记住

- 像`std::shared_ptr`使用`std::weak_ptr`可能会悬空。
- `std::weak_ptr`的潜在使用场景包括：caching、observer lists、打破`std::shared_ptr`指向循环。