Fantacity

Stand Alone Complex

C++中RAII技巧(转载)

2016-06-14 (/2016/06/14/cpp-raii/) yosef gao Program (/categories/Program/)

最近在学习c++多线程编程的时候，偶然看到了RAII的概念，有种这么多年c++白学了的感觉，路漫漫其修远兮啊。下面是我在查找RAII资料时候看到的一篇非常好的博客，因为觉得自己实在写不出比这篇更好的对于RAII的总结的博客了，所以就把博客摘过来了。
首先先给出博客的作者：Andrew，作者博客主页：http://www.cnblogs.com/hsinwang/ (http://www.cnblogs.com/hsinwang/)，以及博文的链接：RAII惯用法：C++资源管理的利器 (http://www.cnblogs.com/hsinwang/articles/214663.html)。下面是摘过来的博文内容：

RAII惯用法：C++资源管理的利器

RAII是指C++语言中的一个惯用法(idiom)，它是“Resource Acquisition Is Initialization”的首字母缩写。中文可将其翻译为“资源获取就是初始化”。虽然从某种程度上说这个名称并没有体现出该惯性法的本质精神，但是作为标准C++资源管理的关键技术，RAII早已在C++社群中深入人心。

我记得第一次学到RAII惯用法是在Bjarne Stroustrup的《C++程序设计语言(第3版)》一书中。当讲述C++资源管理时，Bjarne这样写道：

使用局部对象管理资源的技术通常称为“资源获取就是初始化”。这种通用技术依赖于构造函数和析构函数的性质以及它们与异常处理的交互作用。

Bjarne这段话是什么意思呢？

首先让我们来明确资源的概念，在计算机系统中，资源是数量有限且对系统正常运转具有一定作用的元素。比如，内存，文件句柄，网络套接字(network sockets)，互斥锁(mutex locks)等等，它们都属于系统资源。由于资源的数量不是无限的，有的资源甚至在整个系统中仅有一份，因此我们在使用资源时必须严格遵循的步骤是：

1. 获取资源
2. 使用资源
3. 释放资源

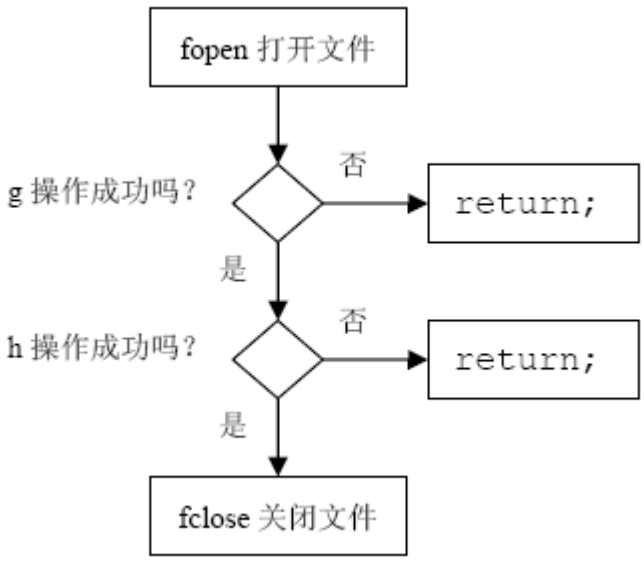
例如在下面的UseFile函数中：

```
void UseFile(char const* fn)
{
    FILE* f = fopen(fn, "r");        // 获取资源
    // 在此处使用文件句柄f...         // 使用资源
    fclose(f);                       // 释放资源
}
```

调用fopen()打开文件就是获取文件句柄资源，操作完成之后，调用fclose()关闭文件就是释放该资源。资源的释放工作至关重要，如果只获取而不释放，那么资源最终会被耗尽。上面的代码是否能够保证在任何情况下都调用fclose函数呢？请考虑如下情况：

```
void UseFile(char const* fn)
{
    FILE* f = fopen(fn, "r");        // 获取资源
    // 使用资源
    if (!g()) return;                // 如果操作g失败!
    // ...
    if (!h()) return;                // 如果操作h失败!
    // ...
    fclose(f);                       // 释放资源
}
```

在使用文件f的过程中，因某些操作失败而造成函数提前返回的现象经常出现。这时函数UseFile的执行流程将变为：



(https://yosef-gao.github.io/2016/06/14/cpp-raii/raii-flowchart.png)

执行流程

很明显，这里忘记了一个重要的步骤：在操作g或h失败之后，UseFile函数必须首先调用fclose()关闭文件，然后才能返回其调用者，否则会造成资源泄漏。因此，需将UseFile函数修改为：

```
void UseFile(char const* fn)
{
    FILE* f = fopen(fn, "r");        // 获取资源
    // 使用资源
    if (!g()) { fclose(f); return; }
    // ...
    if (!h()) { fclose(f); return; }
    // ...
    fclose(f);                       // 释放资源
}
```

现在的问题是：用于释放资源的代码fclose(f)需要在不同的位置重复书写多次。如果再加入异常处理，情况会变得更加复杂。例如，在文件f的使用过程中，程序可能会抛出异常：

```
void UseFile(char const* fn)
{
    FILE* f = fopen(fn, "r");        // 获取资源
    // 使用资源
    try {
        if (!g()) { fclose(f); return; }
        // ...
        if (!h()) { fclose(f); return; }
        // ...
    }
    catch (...) {
        fclose(f);                   // 释放资源
        throw;
    }
    fclose(f);                       // 释放资源
}
```

我们必须依靠catch(…)来捕获所有的异常，关闭文件f，并重新抛出该异常。随着控制流程复杂度的增加，需要添加资源释放代码的位置会越来越多。如果资源的数量还不止一个，那么程序员就更加难于招架了。可以想象这种做法的后果是：代码臃肿，效率下降，更重要的是，程序的可理解性和可维护性明显降低。是否存在一种方法可以实现资源管理的自动化呢？答案是肯定的。假设UseResources函数要用到n个资源，则进行资源管理的一般模式为：

```
void UseResources()
{
    // 获取资源1
    // ...
    // 获取资源n

    // 使用这些资源

    // 释放资源n
    // ...
    // 释放资源1
}
```

不难看出资源管理技术的关键在于：要保证资源的释放顺序与获取顺序严格相反。这自然使我们联想到局部对象的创建和销毁过程。在C++中，定义在栈空间上的局部对象称为自动存储(automatic memory)对象。管理局部对象的任务非常简单，因为它们的创建和销毁工作是由系统自动完成的。我们只需在某个作用域(scope)中定义局部对象(这时系统自动调用构造函数以创建对象)，然后就可以放心大胆地使用之，而不必担心有关善后工作；当控制流程超出这个作用域的范围时，系统会自动调用析构函数，从而销毁该对象。

读者可能会说：如果系统中的资源也具有如同局部对象一样的特性，自动获取，自动释放，那该有多么美妙啊！。事实上，您的想法已经与RAII不谋而合了。既然类是C++中的主要抽象工具，那么就将资源抽象为类，用局部对象来表示资源，把管理资源的任务转化为管理局部对象的任务。这就是RAII惯用法的真谛！可以毫不夸张地说，RAII有效地实现了C++资源管理的自动化。例如，我们可以将文件句柄FILE抽象为FileHandle类：

```
class FileHandle {
public:
    FileHandle(char const* n, char const* a) { p = fopen(n, a); }
    ~FileHandle() { fclose(p); }
private:
    // 禁止拷贝操作
    FileHandle(FileHandle const&);
    FileHandle& operator= (FileHandle const&);
    FILE *p;
};
```

FileHandle类的构造函数调用fopen()获取资源；FileHandle类的析构函数调用fclose()释放资源。请注意，考虑到FileHandle对象代表一种资源，它并不具有拷贝语义，因此我们将拷贝构造函数和赋值运算符声明为私有成员。如果利用FileHandle类的局部对象表示文件句柄资源，那么前面的UseFile函数便可简化为：

```
void UseFile(char const* fn)
{
    FileHandle file(fn, "r");
    // 在此处使用文件句柄f...
    // 超出此作用域时，系统会自动调用file的析构函数，从而释放资源
}
```

现在我们就不必担心隐藏在代码之中的return语句了；不管函数是正常结束，还是提前返回，系统都必须“乖乖地”调用f的析构函数，资源一定能被释放。Bjarne所谓“使用局部对象管理资源的技术……依赖于构造函数和析构函数的性质”，说的正是这种情形。

且慢！如若使用文件file的代码中有异常抛出，难道析构函数还会被调用吗？此时RAII还能如此奏效吗？问得好。事实上，当一个异常抛出之后，系统沿着函数调用栈，向上寻找catch子句的过程，称为栈辗转开解(stack unwinding)。C++标准规定，在辗转开解函数调用栈的过程中，系统必须确保调用所有已创建起来的局部对象的析构函数。例如：

```
void Foo()
{
    FileHandle file1("n1.txt", "r");
    FileHandle file2("n2.txt", "w");
    Bar();       // 可能抛出异常
    FileHandle file3("n3.txt", "rw")
}
```

当Foo()调用Bar()时，局部对象file1和file2已经在Foo的函数调用栈中创建完毕，而file3却尚未创建。如果Bar()抛出异常，那么file2和file1的析构函数会被先后调用(注意：析构函数的调用顺序与构造函数相反)；由于此时栈中尚不存在file3对象，因此它的析构函数不会被调用。只有当一个对象的构造函数执行完毕之后，我们才认为该对象的创建工作已经完成。栈辗转开解过程仅调用那些业已创建的对象的析构函数。

RAII惯用法同样适用于需要管理多个资源的复杂对象。例如，Widget类的构造函数要获取两个资源：文件myFile和互斥锁myLock。每个资源的获取都有可能失败并且抛出异常。为了正常使用Widget对象，这里我们必须维护一个不变式(invariant)：当调用构造函数时，要么两个资源全都获得，对象创建成功；要么两个资源都没得到，对象创建失败。获取了文件而没有得到互斥锁的情况永远不能出现，也就是说，不允许建立Widget对象的“半成品”。如果将RAII惯用法应用于成员对象，那么我们就可以实现这个不变式：

```
class Widget {
public:
    Widget(char const* myFile, char const* myLock)
    : file_(myFile),     // 获取文件myFile
      lock_(myLock)      // 获取互斥锁myLock
    {}
    // ...
private:
    FileHandle file_;
    LockHandle lock_;
};
```

FileHandle和LockHandle类的对象作为Widget类的数据成员，分别表示需要获取的文件和互斥锁。资源的获取过程就是两个成员对象的初始化过程。在此系统会自动地为我们进行资源管理，程序员不必显式地添加任何异常处理代码。例如，当已经创建完file_，但尚未创建完lock_时，有一个异常被抛出，则系统会调用file_的析构函数，而不会调用lock_的析构函数。Bjarne所谓构造函数和析构函数“与异常处理的交互作用”，说的就是这种情形。

综上所述，RAII的本质内容是用对象代表资源，把管理资源的任务转化为管理对象的任务，将资源的获取和释放与对象的构造和析构对应起来，从而确保在对象的生存期内资源始终有效，对象销毁时资源必被释放。换句话说，拥有对象就等于拥有资源，对象存在则资源必定存在。由此可见，RAII惯用法是进行资源管理的有力武器。C++程序员依靠RAII写出的代码不仅简洁优雅，而且做到了异常安全。难怪微软的MSDN杂志在最近的一篇文章中承认：“若论资源管理，谁也比不过标准C++”。

最后再次感谢原文的作者！

2016/6/17修改

关于RAII与fork()

在C++程序中，我们可以用对象来包装资源，把资源管理与对象生命期管理统一起来(RAII)。但是，加入程序会fork()，这一假设就会被破坏了。考虑下面这个例子，Foo对象构造了一次，但是析构了两次。

```
int main()
{
    Foo foo;         // 调用构造函数
    fork();          // fork为两个进程
    foo.doit();      // 在父子进程中都使用foo
    // 析构函数会被调用两次，父进程和子进程各一次
}
```

如果Foo class封装了某种资源，而这个资源没有被子进程继承，那么Foo::doit()的功能在子进程中是错乱的。
通常我们会用RAII手法来管理资源(加锁解锁、创建销毁定时器等等)，但是fork()出来的子进程中不一定正常工作，因为资源在fork()时已经被释放了。
因此，我们在变成服务端程序的时候，“是否允许fork()”是在一开始就应该慎重考虑的问题，在一个没有为fork()做好准备的程序中使用fork()，会遇到难以预料的问题。

---

c/c++ (/tags/c-c/)

Share

Linux shell program(3) (/2016/06/13/linux-shell-program/)

Mutex Lock Guard (/2016/06/16/mutex-lock-guard/)

About

Hi, I am yosef gao.