**一、何为RAII**

RAII(Resource Acquisition is Initialization)——“资源取得时即初始化”。这是一种资源管理的观念，一般通过资源管理类来实现。其精髓在于，在构造函数中获得资源，在析构函数中释放资源。这里的资源包括内存、文件句柄、网络连接、互斥量等等。

**二、为什么要用RAII**

请看如下[代码](http://www.xuebuyuan.com/" \o "代码" \t "_blank)：

void function()

{

Resource\* pRc = createResource();

...

delete pRc;

}

pRc 是指向某资源的指针。我们本意是希望在function中初始化一个资源，利用该资源执行某些运算，最后释放资源。然而"delete pRc;"并不是总能执行到，如果在操作时发生了什么异常，或者别的什么事，比如很愚蠢的使用了goto语句，或者也许这个资源在某个循环内，而这个循环在某种情况下执行了continue。总之只要发生能让"delete pRc"不执行的情况，这个资源就会得不到释放，这绝不是我们所乐于见到的。

当然也许有人会考虑用下面的方法解决异常的问题。

void function()

{

try

{

Resource\* pRc = createResource();

...

delete pRc;

}catch(...){ delete pRc; }

}

这也许不失为一种方法，然后考虑到日后维护时，维护[程序](http://www.xuebuyuan.com/" \o "程序" \t "_blank)员也许会在不完全理解这个函数对资源进行管理的用意下加入某些代码。如果这些代码引起了不在catch内声明的异常，那么资源仍旧会遇到没有释放的情况。为此我们引入用对象管理资源的思想。

**三、资源管理类**

资源管理类的基本构造并不复杂。一个private的原始资源字段，一个能够初始化原始资源的构造函数以及一个释放资源用的析构函数。基本思想上文已经提及，在构造函数中获得资源，在析构函数中释放资源。还是考虑上面的代码，这次我们有一个资源管理类Manager。

class Manager

{

private:Resource\* pRc ;

public:Manager(Resource\* pRc1);

~Manager();

...

}

public Manager();

Manager::Manager(Resource\* pRc1)

{

pRc = pRc1;

}

Manager::~Manager()

{

try

{

delete pRc ;

}catch(...)

{

delete pRc ;

}

}

void function()

{

Manager(createResource());

...

}

利用Manager管理Resource的好处在于不论function的生命周期是如何结束的，Manager的析构函数总是会被调用，资源会被释放。当然如果遇到异常就不太好办了，关于这点会在下面提及如何解决。

**四、常用的RAII实现**

对于指针的管理，实际上C++的库已经为我们提供了不少智能智能。例如std::auto\_ptr。

上面的function我们可以这么改写：

void function()

{

std::auto\_ptr<Resource> pRc(createResource());

...

}

当function结束的时候，我们借由std::auto\_ptr的析构函数将资源Resource释放掉。这也是我们写自己的资源管理类时依仗的思想——利用自动调用的析构函数来释放资源。

然而std::auto\_ptr的使用需要注意以下几点：

1、别用多个std::auto\_ptr指向同一个资源，这会导致同一资源“多次”被 delete，会发生什么你懂的。

2、不要对std::auto\_ptr使用赋值或者copy函数，否则原来的那个就会被置为null。这时候如果你再使用原来的std::auto\_ptr，那么恭喜你，中奖了。

（C++ 11中对auto\_ptr做了一些修复）

为此推荐使用另一款智能指针——RCSP(referene-counting smart pointer)，引用计数型智能指针。该指针的行为有点像Java对于reference的管理(与C++的reference不同，行为接近这里提到的智能指针)，它持续追踪统计公有多少个对象在管理某笔资源，并在无人指向时，释放该资源。学过Java或者.net托管的童鞋对于这种近似垃圾回收器(gargbage collection)的行为一定不陌生。

不过C++的世界从来就没有最优方案，只有最适用方案。RCSP同样有其弊病，当遇到环状引用(两个实际没有被使用却彼此互相指着)时它就束手无策了。

tr1:shared\_ptr就是个RCSP。(注：TR1 是std的拓展，全称为std::tr1，如果标准库不能满足你的话，推荐去boost上找找别人写的库，也许会有惊喜)。这里我们再次改写function：

void function()

{

tr1:shared\_ptr<Resource> pRc(createResource());

...

}

**五、RAII简单分类**

RAII根据资源获取方式可以分为：外部初始化的RAII与内部初始化的RAII两种。其中后者实现起来较为简单，std::string就是一个内部初始化的RAII，它把底层的char数组封装了，资源对于外部程序来说是不可见的。上文中提到的std::auto\_ptr与tr1:shared\_ptr都是外部初始化RAII，我们可以通过给其构造函数传参的方式给予被管理资源。

**六、使用RAII的注意事项**

注意copy行为。Copy行为的发生通常是隐晦的，很难引起注意。这种行为可以通过默认提供的copy构造函数、copy assignment实现。然而对于一个用于实现RAII的资源管理类来说，copy行为往往是值得注意的。把一个资源管理类复制一份，意味着同一资源被多个资源管理类管理，甚至原始资源本身也被复制了一份。如果这并不是你想要的设计，那么你必须防止这种行为的发生。以下列举了四种常见的可能性：

1. 不希望资源管理类发生复制时，请将copy构造函数显示的声明为private。

2. 允许资源管理类发生复制却不希望资源被复制时，应该采用“浅拷贝”并引入“引用计数法”，确保在最后一个引用销毁时，释放资源。

3. 允许资源管理类发生复制的同时复制资源时应该采用“深拷贝”。

4. 允许资源管理类发生复制却不允许指向同一原始资源时，应该在复制的同时销毁原资源管理类对象管理着的原始资源。

**提供访问原始资源的途径。这么做的好处有二：**

1. 如果你调用第三方的库，比如某某API，那么API函数所要的参数往往是原始资源，而非你自定义的资源管理类。因此提供访问原始资源的途径就显得相当必要了。

2. 关于析构函数释放资源的问题。释放资源时如果发生了异常那么恭喜你，资源管理类往往没有好的方法。例如下面的代码：

class Manager

{

private:Resource\* pRc ;

public:Manager(Resource\* pRc1);

~Manager();

...

}

public Manager();

Manager::Manager()

{

pRc = new Resource;//姑且用内部初始化的RAII形式

}

Manager::~Manager()

{

try

{

delete pRc ;

}catch(...)

{

delete pRc ;

}

}

上面展示的这种try-catch的用法的思想事实上在电商系统中也经常用到。例如在海航的项目中，退票模块对于void操作主机抛出异常的情况就会再次进行void，出票模块对于第一次打票失败抛出异常的情况会再次进行打票。然后日前退票也好出票也好都出现了失败的情况。因为这种设计方式只能防止一次失败，却无法防止第二次失败。设计是建立在连续两次操作失败是小概率事件的基础之上的。然而事实上，在现实中，真正异常所导致的往往是大概率的二次操作失败，因为两次操作间隔很小，此时异常往往尚未解除。上面的设计只能用来防止打票机偶尔被占用的情况。为了弥补这种设计，电商网站会引入自动进程。而此处我们所需要做的只是向客户提供原始资源的访问，使客户代码可以直接释放资源，这样，当发生异常时，可由客户代码决定如何处理（如再次释放、记录日志、通知客户等等）。

本文到此便结束了，然而要写出完全可靠的RAII对象几乎是不可能完成的任务，因为当资源释放发生异常时，也许最终在代码级别we can do nothing。这也正式C++世界永远都必须面对的问题（其实任何[编程语言](http://www.xuebuyuan.com/)都是如此，总有代码够不到的地方）。因此，只要合适，就请带着RAII的思想设计你的资源管理类。