# C++11 实现模板化 (通用化)RAII 机制 (转)

作者：derek518
原文链接：https://ld246.com/article/1524705073004
来源网站：链滴

# 什么是RAII?

RAII（Resource Acquisition Is Initialization）,也称直译为“资源获取就是初始化”，是C++语言一种管理资源、避免泄漏的机制。

C++标准保证任何情况下，已构造的对象最终会销毁，即它的析构函数最终会被调用。

RAII 机制就是利用了C++的上述特性,在需要获取使用资源RES的时候，构造一个临时对象(T)，在其造T时获取资源，在T生命期控制对RES的访问使之始终保持有效，最后在T析构的时候释放资源。以到安全管理资源对象，避免资源泄漏的目的。

# 为什么要使用RAII?

那么我们经常所说的资源是如何定义的？说到资源,我们立刻能想到的就是内存啊，文件句柄等等啊，只有这些吗?

对于资源的概念不要想象的太狭隘，在计算机系统中，资源是个定义宽泛的概念，所有数量有限且对统正常运行具有一定作用的元素都是资源。比如：网络套接字、互斥锁、文件句柄、内存、数据库记等等，它们属于系统资源。由于系统的资源是有限的，就好比自然界的石油，铁矿一样，不是取之不，用之不竭的。

所以，我们在编程使用系统资源时，都必须遵循一个步骤：

1.申请资源；

2.使用资源；

3.释放资源。

第一步和第三步缺一不可，因为资源必须要申请才能使用的，使用完成以后，必须要释放，如果不释的话，就会造成资源泄漏。

# RAII的例子

# lock_guard

C++11中lock_guard对mutex互斥锁的管理就是典型的RAII机制，以下是C++11头文件``中的lock_guard的源代码，看代码注释就清楚了，这是典型的RAII风格。

```
/// @brief  Scoped lock idiom.
// Acquire the mutex here with a constructor call, then release with
// the destructor call in accordance with RAII style.
template
  class lock_guard
  {
  public:
    typedef _Mutex mutex_type;

    explicit lock_guard(mutex_type& __m) : _M_device(__m)
    { _M_device.lock(); }//作者注:构造对象时加锁(申请资源),构造函数结束，就可以正常使用资源了

    lock_guard(mutex_type& __m, adopt_lock_t) : _M_device(__m)
    { } // calling thread owns mutex
```

```
    ~lock_guard()
    { _M_device.unlock(); }//作者注:析构对象时解锁(释放资源)
    // 作者注:禁用拷贝构造函数
    lock_guard(const lock_guard&) = delete;
    // 作者注:禁用赋值操作符
    lock_guard& operator=(const lock_guard&) = delete;

  private:
    mutex_type&  _M_device;
  };
```

为了保证lock_guard对象不被错误使用，产生不可预知的后果，上面的代码中删除了lock_guard对的拷贝构造函数和赋值运算符,以确保lock_guard不会被复制，这是RAII机制的一个基本特征，后面有RAII实现都具备这个特性。

lock_guard的调用方式也很简单了，就借用cplusplus.com上的例程来说明吧

```
// lock_guard example
#include        // std::cout
#include          // std::thread
#include          // std::mutex, std::lock_guard
#include       // std::logic_error

std::mutex mtx;

void print_even (int x) {
  if (x%2==0) std::cout << x << " is even\n";
  else throw (std::logic_error("not even"));
}

void print_thread_id (int id) {
  try {
    // using a local lock_guard to lock mtx guarantees unlocking on destruction / exception:
    std::lock_guard lck (mtx);
    //作者注:定义一个变量lck就可以了。不用对lck作任何操作,lck在作用域结束的时候会自动释放mtx

    print_even(id);
  }
  catch (std::logic_error&) {
    std::cout << "[exception caught]\n";
  }
}

int main ()
{
  std::thread threads[10];
  // spawn 10 threads:
  for (int i=0; i<10; ++i)
    threads[i] = std::thread(print_thread_id,i+1);

  for (auto& th : threads) th.join();
```

```
  return 0;
}
```

# 对自定义锁的RAII实现

我在之前的文章无锁编程:c++11基于atomic实现共享读写锁(写优先)中提到过一个共享读写锁RWLock它实现了对资源的共享读取和独占写入。基本的接口如下:

```
class RWLock {
    int readLock();
    int readUnlock();
    int writeLock();
    int writeUnlock();
};
```

如果我要用RAII方式管理RWLock对象，我就要写一个针对RWLock的类，因为RWLock分为读取锁写入锁两种加锁方式，所以不能使用上节中现成的std::lock_guard来实现RAII管理，

那么，我就要分别针对两种类型写两个不同的类。下面是管理读取锁的RAII类代码：

```
class RWLockGuard_R{
    public:
        RWLockGuard_R(RWLock & lock):lock(lock){lock.readLock();}
        ~RWLockGuard_R(){lock.readUnlock();}
        RWLockGuard_R(const RWLockGuard_R&) = delete;
        RWLockGuard_R& operator=(const RWLockGuard_R&) = delete;
    private:
        RWLock & lock;
    };
```

当然，关于管理RWLock写入锁的RAII类写法也大同小异，无非是把readUnlock和readLock替换为writeLock和writeUnlock

调用方式也与前面的lock_guard相似

```
RWLock lock;
void raii_test(){
    RWLockGuard_R guard(lock);
    //do something...
}
int main(){
    raii_test();
}
```

# RAII模板化实现

按照上节的做法,如果你有很多个资源对象要用RAII方式管理，按这个办法就要为每个类写一个RAII类。

想到这里，我瞬间觉得好烦燥，都是类似的代码，却要一遍一遍的重复,就不能有个通用的方法让我少点代码嘛！！！

于是我利用C++11的新特性(类型推导、右值引用、移动语义、类型萃取、function/bind、lambda

达式等等)写了一个通用化的RAII机制,满足各种类型资源的管理需求。

```
#include
#include
namespace gyd{
/* 元模板，如果是const类型则去除const修饰符 */
template
struct no_const {
    using type=typename std::conditional::value,typename std::remove_const::type,T>::type;
};
/*
 * RAII方式管理申请和释放资源的类
 * 对象创建时,执行acquire(申请资源)动作(可以为空函数[]{})
 * 对象析构时,执行release(释放资源)动作
 * 禁止对象拷贝和赋值
 */
class raii{
public:
    using fun_type =std::function;
    /* release: 析构时执行的函数
     * acquire: 构造函数执行的函数
     * default_com:_commit,默认值,可以通过commit()函数重新设置
     */
    explicit raii(fun_type release, fun_type acquire = [] {}, bool default_com = true) noexcept:
            _commit(default_com), _release(release) {
        acquire();
    }
    /* 对象析构时根据_commit标志执行_release函数 */
    ~raii() noexcept{
        if (_commit)
            _release();
    }
    /* 移动构造函数 允许右值赋值 */
    raii(raii&& rv)noexcept:_commit(rv._commit),_release(rv._release){
        rv._commit=false;
    };
    /* 禁用拷贝构造函数 */
    raii(const raii&) = delete;
    /* 禁用赋值操作符 */
    raii& operator=(const raii&) = delete;

    /* 设置_commit标志 */
    raii& commit(bool c = true)noexcept { _commit = c; return *this; };
private:
    /* 为true时析构函数执行_release */
    bool _commit;
protected:
    /* 析构时执的行函数 */
    std::function _release;
}; /* raii */

/* 用于实体资源的raii管理类
 * T为资源类型
```

原文链接：C++11 实现模板化 (通用化)RAII 机制 (转)

```
 * acquire为申请资源动作，返回资源T
 * release为释放资源动作,释放资源T
 */
template
class raii_var{
public:
    using    _Self      = raii_var;
    using   acq_type    =std::function;
    using   rel_type    =std::function;
    explicit raii_var(acq_type acquire , rel_type release) noexcept:
            _resource(acquire()),_release(release) {
        //构造函数中执行申请资源的动作acquire()并初始化resource;
    }
    /* 移动构造函数 */
    raii_var(raii_var&& rv):
            _resource(std::move(rv._resource)),
            _release(std::move(rv._release))
            {
        rv._commit=false;//控制右值对象析构时不再执行_release
    }
    /* 对象析构时根据_commit标志执行_release函数 */
    ~raii_var() noexcept{
        if (_commit)
            _release(_resource);
    }
    /* 设置_commit标志 */
    _Self& commit(bool c = true)noexcept { _commit = c; return *this; };
    **重点内容**/* 获取资源引用 */
    T& get() noexcept{return _resource;}
    T& operator*() noexcept
    { return get();}

    /* 根据 T类型不同选择不同的->操作符模板 */
    template
    typename std::enable_if::value,_T>::type operator->() const noexcept
    { return _resource;}
    template
    typename std::enable_if::value,_T*>::type operator->() const noexcept
    { return std::addressof(_resource);}

private:
    /* 为true时析构函数执行release */
    bool    _commit=true;
    T    _resource;
    rel_type _release;
};
/* 创建 raii 对象,
 * 用std::bind将M_REL,M_ACQ封装成std::function创建raii对象
 * RES      资源类型
 * M_REL    释放资源的成员函数地址
 * M_ACQ    申请资源的成员函数地址
 */
template
raii make_raii(RES & res, M_REL rel, M_ACQ acq, bool default_com = true)noexcept {
```

```
    // 编译时检查参数类型
    // 静态断言中用到的is_class,is_member_function_pointer等是用于编译期的计算、查询、判断转换的type_traits类,
    // 有点类似于java的反射(reflect)提供的功能,不过只能用于编译期，不能用于运行时。
    // 关于type_traits的详细内容参见:http://www.cplusplus.com/reference/type_traits/
    static_assert(std::is_class::value, "RES is not a class or struct type.");
    static_assert(std::is_member_function_pointer::value, "M_REL is not a member function.");
    static_assert(std::is_member_function_pointer::value,"M_ACQ is not a member function.");
    assert(nullptr!=rel&&nullptr!=acq);
    auto p_res=std::addressof(const_cast::type&>(res));
    return raii(std::bind(rel, p_res), std::bind(acq, p_res), default_com);
}
/* 创建 raii 对象 无需M_ACQ的简化版本 */
template
raii make_raii(RES & res, M_REL rel, bool default_com = true)noexcept {
    static_assert(std::is_class::value, "RES is not a class or struct type.");
    static_assert(std::is_member_function_pointer::value, "M_REL is not a member function.");
    assert(nullptr!=rel);
    auto p_res=std::addressof(const_cast::type&>(res));
    return raii(std::bind(rel, p_res), []{}, default_com);
}
} /* namespace gyd*/
```

上面的代码已经在gcc5和vs2015下编译测试通过 gcc编译时需要加上 -std=c++11 编译选项

上面的代码中其实包括了两个类(raii,raii_var)和两个函数(make_raii参数重载),对应着代码提供的三种现通用RAII机制的方式:

- raii是基于可调用对象(Callable Object)来实现的通用RAII机制,直接以可调用对象定义申请资源和放资源的动作为类初始化参数构造raii对象。**适合任何类型(包括非对象资源)资源的RAII管理。**
- raii_var是实现对于实体资源(非互斥锁)的通用RAII机制模板类。**适合实体类(包括非对象资源)资源RAII管理。**
- 模板函数　make_raii在raii基础上做了进一步封装，对于资源对象(struct/class)指定资源对象成员数分别作为申请资源和释放资源的动作。**适用于class类型的资源对象(struct/class)**

下面分别对两种方式的用法举例：

# raii

还以前面的RWLock资源锁为例：

```
RWLock lock;
void raii_test(){
    raii guard_r([&](){lock.readUnlock();},[&](){lock.readLock();},true);
    //do something...
}
int main(){
    raii_test();
}
```

上面的例子中直接用lambda表达分别构造了释放/申请资源的可调用对象(Callable Object)做参数

再举一个文件操作的例子：

```
#include
 int main () {
     ofstream out("out.txt");
     raii guard_file([&out](){if (out.is_open())out.close();});
     //do something...
 }
```

上面的例子中，先打开一个文件,创建一个ofstream对象,因为在raii构造函数中不需要做任何事，所以raii的构造函数中后面两个参数acquire和default_com都省略使用默认值。只需要在raii对象析构的时执行关闭文件的动作。

# raii_var

上一节中文件操作的例子如果使用raii_var可以这样写：

```
#include
 int main () {
     raii_var out([]{return ofstream("out.txt");},[](ofstream &f){if (f.is_open())f.close();});
    auto& f=out.get();//获取ofstream对象;
    //do something....
 }
```

举一个jni方面的例子，下面的代码将一个java字节数组转为一个c++的数组结构face_code：

```
bool jni_utilits::jbytearraytoface_code(JNIEnv *env, jbyteArray bytes,face_code &code) {
    if(env->GetArrayLength(bytes)==sizeof(face_code)){
        //推导GetByteArrayElements返回的资源对象类型，定义为type
        using type=decltype(env->GetByteArrayElements(bytes, JNI_FALSE));
        raii_var byte_ptr(
                [&]()->type{return env->GetByteArrayElements(bytes, JNI_FALSE);},
                [&](type &r){env->ReleaseByteArrayElements(bytes, r, JNI_ABORT);}
                );
        code=*((face_code*)byte_ptr.get());//获取字节数组指针转为face_code结构
        return true;
    }
    return false;
}
```

我们知道，获取一个java基本类型数组，要先调用GetArrayElements获取数据的引用，用完后一定调用ReleaseArrayElements释放引用。否则会引起内存泄露。

上面的代码中把获取和释放的动作封装成了一个raii_var对象byte_ptr,对象构造的时候调用GetByteArrayElements，构造对象之后，调用byte_ptr.get()就可以获取GetByteArrayElements返回的字节数组针，就可以析构的时候调用ReleaseByteArrayElements

# make_raii

为了简化对象类(struct/class)资源的RAII管理机制，所以提供了模板化的make_raii函数。

下面以RWLock为资源对象说明make_raii的用法:

```
RWLock lock;
void make_raii_test(){
    auto guard_read=make_raii(lock,&RWLock::readUnlock,&RWLock::readLock);
    //在这里auto 是C++11中赋予了新含义的关键字，意思是guard_read的类型由编译器自动推算。
    //在VS2015 IDE下鼠标停在guard_read上就能看出，guard_read的实际类型就是
    //raii_class
    //跟上节例子中guard_read的类型定义完全一样
    //do something...
}
int main(){
    make_raii_test();
}
```

如果只有释放资源(M_REL)的动作，可以使用只有两个参数的版本

```
raii make_raii(RES & res, M_REL rel, bool default_com = true)
```

也正是因为这里要用make_raii构造raii对象再传递给自动变量guard_read,所以在raii中虽然禁止了拷构造函数和赋值操作符,却有移动构造函数,就是为了在这里make_raii生成的右值能传递给guard_read否则不能编译通过。

# 更进一步简化

如果你觉得上一节的调用方式还是不够简洁，我们可以修改RWLock，添加一个静态成员函数make_guard对make_raii进行便利化封装，进一步隐藏RAII细节。

```
class RWLock{
//other class defineition code....
static auto make_guard(RWLock &lock)->decltype(gdface::make_raii(lock,&RWLock::readUnlo
k,&RWLock::readLock,true)){
        return gdface::make_raii(lock,&RWLock::readUnlock,&RWLock::readLock,true);
    }
//这里auto xxx -> xxx 的句法使用用了C++11的"追踪返回类型"特性，将返回类型后置，
//使用decltype关键字推导出返回类型
}
```

然后我们就可以像这样调用:

```
RWLock lock;
void make_raii_test(){
    auto guard=RWLock::make_guard(lock);
    //do something...
}
int main(){
    make_raii_test();
}
```

# 总结

到这里，本文就算结束了，文中主要提出了一种模板化的RAII机制实现方法，给出了三种RAII使用方，你可以根据自己需要选择两种方法中的任意一种。

显然第一种直接构造raii对象的方法更通用,适合于任何类型资源，
第二种raii_var模板类适用于实体类资源比如打开关闭文件这种acquire动作有返回资源对象的，
第三种使用make_raii模板函数构造raii对象的方法对于对象型资源(struct/class)更方便。
如果是自己项目中的资源类对象，再对make_raii进一步封装，使用起来就更方便了。

# 参考资料

C++中的RAII机制
异常安全,RAII与C++11
Type support (basic types, RTTI, type traits)
支持 C++11/14/17 功能（现代 C++）