WIKIPEDIA

RAII

维基百科，自由的百科全书

RAII全称为**R**esource **A**cquisition **I**s **I**nitialization，它是在一些面向对象语言中的一种惯用法。RAII源于C++，在Java，C#，D，Ada，Vala和Rust中也有应用。1984-1989年期间，比雅尼·斯特劳斯特鲁普和安德鲁·柯尼希在设计C++异常时，为解决资源管理时的异常安全性而使用了该用法[1]，后来比雅尼·斯特劳斯特鲁普将其称为RAII[2]。

RAII要求，资源的有效期与持有资源的对象的生命期严格绑定，即由对象的构造函数完成资源的分配(获取)，同时由析构函数完成资源的释放。在这种要求下，只要对象能正确地析构，就不会出现资源泄露问题。

目录

1 作用
2 典型用法
3 RRID
4 对比finally
5 参考资料

作用

RAII的主要作用是在不失代码简洁性[3]的同时，可以很好地保证代码的异常安全性。

下面的C++实例说明了如何用RAII访问文件和互斥量：

```cpp
#include <string>
#include <mutex>
#include <iostream>
#include <fstream>
#include <stdexcept>

void write_to_file(const std::string & message)
{
  // 创建关于文件的互斥锁
  static std::mutex mutex;

  // 在访问文件前进行加锁
  std::lock_guard<std::mutex> lock(mutex);

  // 尝试打开文件
  std::ofstream file("example.txt");
  if (!file.is_open())
     throw std::runtime_error("unable to open file");

  // 输出文件内容
  file << message << std::endl;

  // 当离开作用域时，文件句柄会被首先析构 (不管是否抛出了异常)
  // 互斥锁也会被析构 (同样地，不管是否抛出了异常)
}
```

C++保证了所有栈对象在生命周期结束时会被销毁(即调用析构函数)[4]，所以该代码是异常安全的。无论在write_to_file函数正常返回时，还是在途中抛出异常时，都会引发write_to_file函数的堆栈回退，而此时会自动调用lock和file对象的析构函数。

当一个函数需要通过多个局部变量来管理资源时，RAII就显得非常好用。因为只有被构造成功(构造函数没有抛出异常)的对象才会在返回时调用析构函数[4]，同时析构函数的调用顺序恰好是它们构造顺序的反序[5]，这样既可以保证多个资源(对象)的正确释放，又能满足多个资源之间的依赖关系。

由于RAII可以极大地简化资源管理，并有效地保证程序的正确和代码的简洁，所以通常会强烈建议在C++中使用它。

典型用法

RAII在C++中的应用非常广泛，如C++标准库中的lock_guard (http://en.cppreference.com/w/cpp/thread/lock_guard)便是用RAII方式来控制互斥量:

```
template <class Mutex> class lock_guard {
private:
    Mutex& mutex_;

public:
    lock_guard(Mutex& mutex) : mutex_(mutex) { mutex_.lock(); }
    ~lock_guard() { mutex_.unlock(); }

    lock_guard(lock_guard const&) = delete;
    lock_guard& operator=(lock_guard const&) = delete;
};
```

程序员可以非常方便地使用lock_guard，而不用担心异常安全问题

```
extern void unsafe_code();  // 可能抛出异常

using std::mutex;
using std::lock_guard;

mutex g_mutex;

void access_critical_section()
{
    lock_guard<mutex> lock(g_mutex);
    unsafe_code();
}
```

实际上，C++标准库的实现就广泛应用了RAII，典型的如容器、智能指针等。

RRID

RAII还有另外一种被称为RRID(Resource Release Is Destruction)的特殊用法[6]，即在构造时没有"*获取*"资源，但在析构时释放资源。ScopeGuard[7]和Boost.ScopeExit (http://www.boost.org/doc/libs/1_56_0/libs/scope_exit/doc/html/index.html)就是RRID的典型应用：

```
#include <functional>

class ScopeGuard {
private:
    typedef std::function<void()> destructor_type;

    destructor_type destructor_;
    bool dismissed_;

public:
    ScopeGuard(destructor_type destructor) : destructor_(destructor), dismissed_(false) {}

    ~ScopeGuard()
    {
        if (!dismissed_) {
            destructor_();
        }
    }

    void dismiss() { dismissed_ = true; }

    ScopeGuard(ScopeGuard const&) = delete;
    ScopeGuard& operator=(ScopeGuard const&) = delete;
};
```

ScopeGuard通常用于省去一些不必要的RAII封装，例如

```
void foo()
{
    auto fp = fopen("/path/to/file", "w");
    ScopeGuard fp_guard([&fp]() { fclose(fp); });

    write_to_file(fp);                  // 异常安全
}
```

在D语言中，scope关键字也是典型的RRID用法，例如

```
void access_critical_section()
{
  Mutex m = new Mutex;
  lock(m);
  scope(exit) unlock(m);

  unsafe_code();              // 异常安全
}

Resource create()
{
  Resource r = new Resource();
  scope(failure) close(f);

  preprocess(r);              // 抛出异常时会自动调用close(r)
  return r;
}
```

对比finally

虽然RAII和finally都能保证资源管理时的异常安全，但相对来说，使用RAII的代码相对更加简洁。正如比雅尼·斯特劳斯特鲁普所说，“在真实环境中，调用资源释放代码的次数远多于资源类型的个数，所以相对于使用用finally来说，使用RAII能减少代码量。”[8]

例如在Java中使用finally来管理Socket资源

```
void foo() {
  Socket socket;
  try {
    socket = new Socket();
    access(socket);
  } finally {
    socket.close();
  }
}
```

在采用RAII后，代码可以简化为

```
void foo() {
  try (Socket socket = new Socket()) {
    access(socket);
  }
}
```

特别是当大量使用Socket时，那些重复的finally就显得没有必要。

参考资料

- Stroustrup, Bjarne. The C++ Programming Language [C++程序设计语言]. Addison-Wesley. 2000. ISBN 0-201-70073-5（英语）.
- Stroustrup, Bjarne. The Design and Evolution of C++ [C++语言的设计和演化]. Addison-Wesley. 1994. ISBN 0-201-54330-3（英语）.
- Wilson, Matthew. Imperfect C++ [Imperfect C++中文版]. Addison-Wesley. 2004. ISBN 0321228774（英语）.

1. Exception Handling for C++ (http://www.stroustrup.com/except89.pdf), 5 Handling of Destructors
2. Stroustrup 1994，chpt. 16.5 Resource Management. I called this technique “resource acquisition is initialization.”
3. C++ FAQ, "I have too many try blocks; what can I do about it?" (http://www.parashift.com/c++-faq/too-many-trycatch-blocks.html)
4. Stroustrup 2000，chpt. 14.4.1 Using Constructors and Destructors.
5. C++ FAQ, "What's the order that local objects are destructed?" (http://www.parashift.com/c++-faq/order-dtors-for-locals.html)
6. Wilson 2004，chpt. 3.4 RRID.
7. Andrei Alexandrescu, Change the Way You Write Exception-Safe Code (http://www.drdobbs.com/cpp/generic-change-the-way-you-write-excepti/184403758)
8. Bjarne Stroustrup's C++ Style and Technique FAQ (http://www.stroustrup.com/bs_faq2.html). "Why doesn't C++ provide a "finally" construct?" (http://www.stroustrup.com/bs_faq2.html#finally)
