10讲到底应不应该返回对象

你好，我是吴咏炜。

前几讲里我们已经约略地提到了返回对象的问题，本讲里我们进一步展开这个话题，把返回对象这个问题讲深讲透。

F.20

《C++ 核心指南》的 F.20 这一条款是这么说的[1]：

> F.20: For “out” output values, prefer return values to output parameters

翻译一下：

> 在函数输出数值时，尽量使用返回值而非输出参数

这条可能会让一些 C++ 老手感到惊讶——在 C++11 之前的实践里，我们完全是采用相反的做法的啊！

在解释 F.20 之前，我们先来看看我们之前的做法。

调用者负责管理内存，接口负责生成

一种常见的做法是，接口的调用者负责分配一个对象所需的内存并负责其生命周期，接口负责生成或修改该对象。这种做法意味着对象可以默认构造（甚至只是一个结构），代码一般使用错误码而非异常。

示例代码如下：

```
MyObj obj;
ec = initialize(&obj);
…
```

这种做法和 C 是兼容的，很多程序员出于惯性也沿用了 C 的这种做法。一种略为 C++ 点的做法是使用引用代替指针，这样在上面的示例中就不需要使用 `&` 运算符了；但这样只是语法略有区别，本质完全相同。如果对象有合理的析构函数的话，那这种做法的主要问题是啰嗦、难于组合。你需要写更多的代码行，使用更多的中间变量，也就更容易犯错误。

假如我们已有矩阵变量 $\mathbf{A}$、$\mathbf{B}$ 和 $\mathbf{C}$，要执行一个操作

$$
\mathbf{R} = \mathbf{A} \times \mathbf{B} + \mathbf{C}
$$

那在这种做法下代码大概会写成：

```
error_code_t add(
  matrix* result,
  const matrix& lhs,
  const matrix& rhs);
error_code_t multiply(
  matrix* result,
  const matrix& lhs,
  const matrix& rhs);
…
  error_code_t ec;
  …
  matrix temp;
  ec = multiply(&temp, a, b);
  if (ec != SUCCESS) {
    goto end;
  }
  matrix r;
  ec = add(&r, temp, c);
  if (ec != SUCCESS) {
    goto end;
  }
  …
end:
  // 返回 ec 或类似错误处理
```

理论上该方法可以有一个变体，不使用返回值，而使用异常来表示错误。实践中，我从来没在实际系统中看到过这样的代码。

接口负责对象的堆上生成和内存管理

另外一种可能的做法是接口提供生成和销毁对象的函数，对象在堆上维护。`fopen` 和 `fclose` 就是这样的接口的实例。注意使用这种方法一般不推荐由接口生成对象，然后由调用者通过调用 `delete` 来释放。在某些环境里，比如 Windows 上使用不同的运行时库时，这样做会引发问题。

同样以上面的矩阵运算为例，代码大概就会写成这个样子：

```
matrix* add(
  const matrix* lhs,
  const matrix* rhs,
  error_code_t* ec);
matrix* multiply(
  const matrix* lhs,
  const matrix* rhs,
  error_code_t* ec);
void deinitialize(matrix** mat);
…
  error_code_t ec;
  …
  matrix* temp = nullptr;
  matrix* r = nullptr;
  temp = multiply(a, b, &ec);
  if (!temp) {
    goto end;
  }
  r = add(temp, c, &ec);
  if (!r) {
    goto end;
  }
  …
end:
  if (temp) {
    deinitialize(&temp);
  }
  // 返回 ec 或类似错误处理
```

可以注意到，虽然代码看似稍微自然了一点，但啰嗦程度却增加了，原因是正确的处理需要考虑到各种不同错误路径下的资源释放问题。这儿也没有使用异常，因为异常在这种表达下会产生内存泄漏，除非用上一堆 `try` 和 `catch`，但那样异常在表达简洁性上的优势就没有了，没有实际的好处。

不过，如果我们同时使用智能指针和异常的话，就可以得到一个还不错的变体。如果接口接受和返回的都是 `shared_ptr<matrix>`，那调用代码就简单了：

```
shared_ptr<matrix> add(
  const shared_ptr<matrix>& lhs,
  const shared_ptr<matrix>& rhs);
shared_ptr<matrix> multiply(
  const shared_ptr<matrix>& lhs,
  const shared_ptr<matrix>& rhs);
…
  auto r = add(multiply(a, b), c);
```

调用这些接口必须要使用 `shared_ptr`，这不能不说是一个限制。另外，对象永远是在堆上分配的，在很多场合，也会有一定的性能影响。

接口直接返回对象

最直接了当的代码，当然就是直接返回对象了。这回我们看实际可编译、运行的代码：

```
#include <armadillo>
#include <iostream>

using arma::imat22;
using std::cout;

int main()
{
  imat22 a{{1, 1}, {2, 2}};
  imat22 b{{1, 0}, {0, 1}};
  imat22 c{{2, 2}, {1, 1}};
  auto r = a * b + c;
  cout << r;
}
```

这段代码使用了 Armadillo，一个利用现代 C++ 特性的开源线性代数库 [2]。你可以看到代码非常简洁，完全表意（`imat22` 是元素类型为整数的大小固定为 2 x 2 的矩阵）。它有以下优点：

- 代码直观、容易理解。
- 乘法和加法可以组合在一行里写出来，无需中间变量。
- 性能也没有问题。实际执行中，没有复制发生，计算结果直接存放到了变量 `r` 上。更妙的是，因为矩阵大小是已知的，这儿不需要任何动态内存，所有对象及其数据全部存放在栈上。

Armadillo 是个比较复杂的库，我们就不以 Armadillo 的代码为例来进一步讲解了。我们可以用一个假想的 `matrix` 类来看看返回对象的代码是怎样编写的。

如何返回一个对象?

一个用来返回的对象，通常应当是可移动构造/赋值的，一般也同时是可拷贝构造/赋值的。如果这样一个对象同时又可以默认构造，我们就称其为一个半正则（semiregular）的对象。如果可能的话，我们应当尽量让我们的类满足半正则这个要求。

半正则意味着我们的 matrix 类提供下面的成员函数：

```
class matrix {
public:
  // 普通构造
  matrix(size_t rows, size_t cols);
  // 半正则要求的构造
  matrix();
  matrix(const matrix&);
  matrix(matrix&&);
  // 半正则要求的赋值
  matrix& operator=(const matrix&);
  matrix& operator=(matrix&&);
};
```

我们先看一下在没有返回值优化的情况下 C++ 是怎样返回对象的。以矩阵乘法为例，代码应该像下面这样：

```
matrix operator*(const matrix& lhs,
                 const matrix& rhs)
{
  if (lhs.cols() != rhs.rows()) {
    throw runtime_error(
      "sizes mismatch");
  }
  matrix result(lhs.rows(),
                rhs.cols());
  // 具体计算过程
  return result;
}
```

注意对于一个本地变量，我们永远不应该返回其引用（或指针），不管是作为左值还是右值。从标准的角度，这会导致未定义行为（undefined behavior），从实际的角度，这样的对象一般放在栈上可以被调用者正常覆盖使用的部分，随便一个函数调用或变量定义就可能覆盖这个对象占据的内存。这还是这个对象的析构不做事情的情况：如果析构函数会释放内存或破坏数据的话，那你访问到的对象即使内存没有被覆盖，也早就不是有合法数据的对象了……

回到正题。我们需要回想起，在 [第 3 讲] 里说过的，返回非引用类型的表达式结果是个纯右值（prvalue）。在执行 `auto r = …` 的时候，编译器会认为我们实际是在构造 `matrix r(…)`，而“…”部分是一个纯右值。因此编译器会首先试图匹配 `matrix(matrix&&)`，在没有时则试图匹配 `matrix(const matrix&)`；也就是说，有移动支持时使用移动，没有移动支持时则拷贝。

返回值优化（拷贝消除）

我们再来看一个能显示生命期过程的对象的例子：

```
#include <iostream>

using namespace std;

// Can copy and move
class A {
public:
  A() { cout << "Create A\n"; }
  ~A() { cout << "Destroy A\n"; }
  A(const A&) { cout << "Copy A\n"; }
  A(A&&) { cout << "Move A\n"; }
};

A getA_unnamed()
{
  return A();
}

int main()
{
  auto a = getA_unnamed();
}
```

如果你认为执行结果里应当有一行“Copy A”或“Move A”的话，你就忽视了返回值优化的威力了。即使完全关闭优化，三种主流编译器（GCC、Clang 和 MSVC）都只输出两行：

```
Create A
Destroy A
```

我们把代码稍稍改一下：

```
A getA_named()
{
  A a;
  return a;
}

int main()
{
  auto a = getA_named();
}
```

这回结果有了一点点小变化。虽然 GCC 和 Clang 的结果完全不变，但 MSVC 在非优化编译的情况下产生了不同的输出（优化编译——使用命令行参数 `/O1`、`/O2` 或 `/Ox`——则不变）：

```
Create A
Move A
Destroy A
Destroy A
```

也就是说，返回内容被移动构造了。

我们继续变形一下：

```
#include <stdlib.h>

A getA_duang()
{
  A a1;
  A a2;
  if (rand() > 42) {
    return a1;
  } else {
    return a2;
  }
}

int main()
{
  auto a = getA_duang();
}
```

这回所有的编译器都被难倒了，输出是：

```
Create A
Create A
Move A
Destroy A
Destroy A
Destroy A
```

关于返回值优化的实验我们就做到这里。下一步，我们试验一下把移动构造函数删除：

```
  A(A&&) = delete;
```

我们可以立即看到“Copy A”出现在了结果输出中，说明目前结果变成拷贝构造了。

如果再进一步，把拷贝构造函数也删除呢？是不是上面的 `getA_unnamed`、`getA_named` 和 `getA_duang` 都不能工作了？

在 C++14 及之前确实是这样的。但从 C++17 开始，对于类似于 `getA_unnamed` 这样的情况，即使对象不可拷贝、不可移

动，这个对象仍然是可以被返回的！C++17 要求对于这种情况，对象必须被直接构造在目标位置上，不经过任何拷贝或移动的步骤 [3]。

回到 F.20

理解了 C++ 里的对返回值的处理和返回值优化之后，我们再回过头看一下 F.20 里陈述的理由的话，应该就显得很自然了：

> A return value is self-documenting, whereas a `&` could be either in-out or out-only and is liable to be misused.
>
> 返回值是可以自我描述的；而 `&` 参数既可能是输入输出，也可能是仅输出，且很容易被误用。

我想我对返回对象的可读性，已经给出了充足的例子。对于其是否有性能影响这一问题，也给出了充分的说明。

我们最后看一下 F.20 里描述的例外情况：

- “对于非值类型，比如返回值可能是子对象的情况，使用 `unique_ptr` 或 `shared_ptr` 来返回对象。”也就是面向对象、工厂方法这样的情况，像 [第 1 讲] 里给出的 `create_shape` 应该这样改造。
- “对于移动代价很高的对象，考虑将其分配在堆上，然后返回一个句柄（如 `unique_ptr`），或传递一个非 const 的目标对象的引用来填充（用作输出参数）。”也就是说不方便移动的，那就只能使用一个 RAII 对象来管理生命周期，或者老办法输出参数了。
- “要在一个内层循环里在多次函数调用中重用一个自带容量的对象：将其当作输入/输出参数并将其按引用传递。”这也是个需要继续使用老办法的情况。

内容小结

C++ 里已经对返回对象做了大量的优化，目前在函数里直接返回对象可以得到更可读、可组合的代码，同时在大部分情况下我们可以利用移动和返回值优化消除性能问题。

课后思考

请你考虑一下：

1. 你的项目使用了返回对象了吗？如果没有的话，本讲内容有没有说服你？
2. 这讲里我们没有深入讨论赋值；请你思考一下，如果例子里改成赋值，会有什么样的变化？

欢迎留言和我交流你的想法。

参考资料

[1] Bjarne Stroustrup and Herb Sutter (editors), “C++ core guidelines”, item F.20. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-out (非官方中文版可参见 https://github.com/lynnboy/CppCoreGuidelines-zh-CN)

[2] Conrad Sanderson and Ryan Curtin, Armadillo. http://arma.sourceforge.net/

[3] cppreference.com, “Copy elision”. https://en.cppreference.com/w/cpp/language/copy_elision

[3a] cppreference.com, “复制消除”. https://zh.cppreference.com/w/cpp/language/copy_elision

精选留言

小一日一

我认为老师应该讲一下NRVO/RVO与std::move()的区别，这个问题曾经困扰过我，从stackoverflow的问题来看，学习c++11时大多数人都思考过这个问题：https://stackoverflow.com/questions/4986673/c11-rvalues-and-move-semantics-confusion-return

-statement

2019-12-18 12:09

作者回复

简单来说，在对本地变量进行返回时，不用 std::move。实际上，我在第 3 讲就写了：

“有一种常见的 C++ 编程错误，是在函数里返回一个本地对象的引用。由于在函数结束时本地对象即被销毁，返回一个指向本地对象的引用属于未定义行为。理论上来说，程序出任何奇怪的行为都是正常的。

“在 C++11 之前，返回一个本地对象意味着这个对象会被拷贝，除非编译器发现可以做返回值优化（named return value optimization，或 NRVO），能把对象直接构造到调用者的栈上。从 C++11 开始，返回值优化仍可以发生，但在没有返回值优化的情况下，编译器将试图把本地对象移动出去，而不是拷贝出去。这一行为不需要程序员手工用 `std::move` 进行干预——使用`std::move` 对于移动行为没有帮助，反而会影响返回值优化。”

2019-12-18 13:19

木瓜777

项目中一直使用您说的老方法，目前看编译器有优化的话，后面会逐步考虑采用返回对象的方法！ 有个问题问下，如果要返回空对象，该如何做？ 是直接采用空的构造函数?

2019-12-18 12:30

作者回复

用默认构造函数代表空，或者用 optional<对象> （不构造）代表空，或者抛异常代表不正常（视是否不正常而定）。

optional 会在第 22 讲里讨论。

2019-12-18 13:33

nelson

文稿中的代码片段

```
ec = multiply(&temp, a, b);
if (result != SUCCESS)
{
goto end;
}
```

result 应该是 ec吧

2019-12-19 00:15

作者回复

多谢。已修正。

2019-12-19 14:13

hello world

请问老师这个C++20什么时候发布编译器之类的啊？还是说已经有了？

2019-12-18 07:52

作者回复

看这个页面吧：

https://en.cppreference.com/w/cpp/compiler_support

目前 GCC 领先一些（可以用 -std=c++2a 启用 20 的功能），但还没有哪家完整支持 C++20。

2019-12-18 09:37

petit_kayak

一直使用共享指针，非常喜欢这些新的优化，能简化非常多代码，但现实是需要考虑很多无法升级的旧环境，不能随便使用c++11及以后的写法

2019-12-24 09:38

作者回复

先试试升级环境、充分测试看看有没有问题。也许没问题呢?

2019-12-24 18:01

光城~兴

加入了move assignment后，默认是调用move assignment而不是copy assignment。

2019-12-22 22:28

光城~兴

您好，老师，我想问一下c++xx与gcc版本的对应关系，还有这节提到的返回值优化在c++17中的结果与c++14及之前的结果(禁用返回值优化，编译后的结果)是不一样的，像这种有没有参考资料呢?

2019-12-22 18:00

作者回复

对于功能和版本的关系，这个页面比较全：

https://en.cppreference.com/w/cpp/compiler_support

2019-12-23 09:43

花晨少年

我们继续变形一下：

```
#include <stdlib.h>
A getA_duang()
{
A a1;
A a2;
if (rand() > 42) {
return a1;
} else {
return a2;
}
}
int main()
{
auto a = getA_duang();
}
```

这回所有的编译器都被难倒了，输出是：

```
Create A
Create A
Move A
Destroy A
Destroy A
Destroy A
```

————————

老师这个结果应该还是会有优化在的吧？如果完全没有优化应该是两个移动才对，a1或者a2移动给返回值是一次，返回值移动给a又是一次，如果真是这样，哪次被优化掉了？第二次吗

2019-12-21 22:37

作者回复

C++编译器哪会做这么不必要的事……就是一次移动。如果有返回值优化的话，一次移动都不会有。

2019-12-22 10:57

花晨少年

关于返回值优化的实验我们就做到这里。下一步，我们试验一下把移动构造函数删除：

```
A(A&&) = delete;
```

我们可以立即看到“Copy A”出现在了结果输出中，说明目前结果变成拷贝构造了

————————————————————————

请问这种情况说的是针对getA_duang()函数吧？不包括 getA_named() 等函数吧

2019-12-21 22:29

作者回复

对，是调用移动构造变成调用拷贝构造，如果原来就直接返回值优化掉了，那不会变化。

2019-12-22 13:58

西钾钾

以下的代码中，无论是将拷贝构造函数还是移动构造函数置为delete，都不能正常编译（vs2017）。为啥只是使用一次构造函数，老师能简单讲下这个原理么？

```
#include <iostream>

using namespace std;

// Can copy and move
class A {
public:
A() { cout << "Create A\n"; }
~A() { cout << "Destroy A\n"; }
A(const A&) { cout << "Copy A\n"; }
A(A&&) { cout << "Move A\n"; }
};

A getA_unnamed()
{
return A();
}

int main()
{
auto a = getA_unnamed();
}
```

2019-12-20 09:41

作者回复

VS 2017 下也能过的。你没有按环境要求里说的加上 /std:c++17。了解细节，可以看参考资料 [3]。

2019-12-20 13:15