06讲异常：用还是不用，这是个问题

你好，我是吴咏炜。

到现在为止，我们已经有好多次都提到异常了。今天，我们就来彻底地聊一聊异常。

首先，开宗明义，如果你不知道到底该不该用异常的话，那答案就是该用。如果你需要避免使用异常，原因必须是你有明确的需要避免使用异常的理由。

下面我们就开始说说异常。

没有异常的世界

我们先来看看没有异常的世界是什么样子的。最典型的情况就是 C 了。

假设我们要做一些矩阵的操作，定义了下面这个矩阵的数据结构：

```
typedef struct {
  float* data;
  size_t nrows;
  size_t ncols;
} matrix;
```

我们至少需要有初始化和清理的代码：

```
enum matrix_err_code {
  MATRIX_SUCCESS,
  MATRIX_ERR_MEMORY_INSUFFICIENT,
  …
};

int matrix_alloc(matrix* ptr,
                 size_t nrows,
                 size_t ncols)
{
  size_t size =
    nrows * ncols * sizeof(float);
  float* data = malloc(size);
  if (data == NULL) {
    return MATRIX_ERR_MEMORY_INSUFFICIENT;
  }
  ptr->data = data;
  ptr->nrows = nrows;
  ptr->ncols = ncols;
}

void matrix_dealloc(matrix* ptr)
{
  if (ptr->data == NULL) {
    return;
  }
  free(ptr->data);
  ptr->data = NULL;
  ptr->nrows = 0;
  ptr->ncols = 0;
}
```

然后，我们做一下矩阵乘法吧。函数定义大概会是这个样子：

```
int matrix_multiply(matrix* result,
                    const matrix* lhs,
                    const matrix* rhs)
{
  int errcode;
  if (lhs->ncols != rhs->nrows) {
    return MATRIX_ERR_MISMATCHED_MATRIX_SIZE;
    // 呃，得把这个错误码添到 enum matrix_err_code 里
  }
  errcode = matrix_alloc(
    result, lhs->nrows, rhs->ncols);
  if (errcode != MATRIX_SUCCESS) {
    return errcode;
  }
  // 进行矩阵乘法运算
  return MATRIX_SUCCESS;
}
```

调用代码则大概是这个样子：

```
  matrix c;

  // 不清零的话，错误处理和资源清理会更复杂
  memset(c, 0, sizeof(matrix));

  errcode = matrix_multiply(c, a, b);
  if (errcode != MATRIX_SUCCESS) {
    goto error_exit;
  }
  // 使用乘法的结果做其他处理

error_exit:
  matrix_dealloc(&c);
  return errcode;
```

可以看到，我们有大量需要判断错误的代码，零散分布在代码各处。

可这是 C 啊。我们用 C++、不用异常可以吗?

当然可以，但你会发现结果好不了多少。毕竟，C++ 的构造函数是不能返回错误码的，所以你根本不能用构造函数来做可能出错的事情。你不得不定义一个只能清零的构造函数，再使用一个 `init` 函数来做真正的构造操作。C++ 虽然支持运算符重载，可你也不能使用，因为你没法返回一个新矩阵……

我上面还只展示了单层的函数调用。事实上，如果出错位置离处理错误的位置相差很远的话，每一层的函数调用里都得有判断

错误码的代码，这就既对写代码的人提出了严格要求，也对读代码的人造成了视觉上的干扰……

使用异常

如果使用异常的话，我们就可以在构造函数里做真正的初始化工作了。假设我们的矩阵类有下列的数据成员：

```
class matrix {
  …
private:
  float* data_;
  size_t nrows_;
  size_t ncols_;
}
```

构造函数我们可以这样写：

```
matrix::matrix(size_t nrows,
               size_t ncols)
{
  data_  = new float[nrows * ncols];
  nrows_ = nrows;
  ncols_ = ncols;
}
```

析构非常简单：

```
matrix::~matrix()
{
  delete[] data_;
}
```

乘法函数可以这样写：

```
class matrix {
  …
  friend matrix
  operator*(const matrix&,
            const matrix&);
};

matrix operator*(const matrix& lhs,
                 const matrix& rhs)
{
  if (lhs.ncols != rhs.nrows) {
    throw std::runtime_error(
      "matrix sizes mismatch");
  }
  matrix result(lhs.nrows, rhs.ncols);
  // 进行矩阵乘法运算
  return result;
}
```

使用乘法的代码则更是简单：

```
matrix c = a * b;
```

你可能已经非常疑惑了：错误处理在哪儿呢？只有一个 `throw`，跟前面的 C 代码能等价吗？

异常处理并不意味着需要写显式的 `try` 和 `catch`。**异常安全的代码，可以没有任何 `try` 和 `catch`。**

如果你不确定什么是“异常安全”，我们先来温习一下概念：异常安全是指当异常发生时，既不会发生资源泄漏，系统也不会处于一个不一致的状态。

我们看看可能会出现错误/异常的地方：

- 首先是内存分配。如果 `new` 出错，按照 C++ 的规则，一般会得到异常 `bad_alloc`，对象的构造也就失败了。这种情况下，在 `catch` 捕捉到这个异常之前，所有的栈上对象会全部被析构，资源全部被自动清理。
- 如果是矩阵的长宽不合适不能做乘法呢？我们同样会得到一个异常，这样，在使用乘法的地方，对象 `c` 根本不会被构造出来。
- 如果在乘法函数里内存分配失败呢？一样，`result` 对象根本没有构造出来，也就没有 `c` 对象了。还是一切正常。
- 如果 `a`、`b` 是本地变量，然后乘法失败了呢？析构函数会自动释放其空间，我们同样不会有任何资源泄漏。

总而言之，只要我们适当地组织好代码、利用好 RAII，实现矩阵的代码和使用矩阵的代码都可以更短、更清晰。我们可以统一在外层某个地方处理异常——通常会记日志、或在界面上向用户报告错误了。

避免异常的风格指南?

但大名鼎鼎的 Google 的 C++ 风格指南不是说要避免异常吗 [1]？这又是怎么回事呢？

答案实际已经在 Google 的文档里了：

> Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project. The conversion process would be slow and error-prone. We don't believe that the available alternatives to exceptions, such as error codes and assertions, introduce a significant burden.
>
> Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. Because we'd like to use our open-source projects at Google and it's difficult to do so if those projects use exceptions, we need to advise against exceptions in Google open-source projects as well. Things would probably be different if we had to do it all over again from scratch.

我来翻译一下（我的加重）：

> 鉴于 Google 的现有代码不能承受异常，**使用异常的代价要比在全新的项目中使用异常大一些**。转换[代码来使用异常的]过程会缓慢而容易出错。我们不认为可代替异常的方法，如错误码或断言，会带来明显的负担。
>
> 我们反对异常的建议并非出于哲学或道德的立场，而是出于实际考虑。因为我们希望在 Google 使用我们的开源项目，而如果这些项目使用异常的话就会对我们的使用带来困难，我们也需要反对在 Google 的开源项目中使用异常。**如果我们从头再来一次的话，事情可能就会不一样了。**

这个如果还比较官方、委婉的话，Reddit 上还能找到一个更个人化的表述 [2]：

> I use [sic] to work at Google, and Craig Silverstein, who wrote the first draft of the style guideline, said that he regretted the ban on exceptions, but he had no choice; when he wrote it, it wasn't only that the compiler they had at the time did a very bad job on exceptions, but that they already had a huge volume of non-exception-safe code.

我的翻译（同样，我的加重）：

> 我过去在 Google 工作，写了风格指南初稿的 Craig Silverstein 说过**他对禁用异常感到遗憾**，但他当时别无选择。在他写风格指南的时候，不仅**他们使用的编译器在异常上工作得很糟糕**，而且**他们已经有了一大堆异常不安全的代码了**。

当然，除了历史原因以外，也有出于性能等其他原因禁用异常的。美国国防部的联合攻击战斗机（JSF）项目的 C++ 编码规范就禁用异常，因为工具链不能保证抛出异常时的实时性能。不过在那种项目里，被禁用的 C++ 特性就多了，比如动态内存分配都不能使用。

一些游戏项目为了追求高性能，也禁用异常。这个实际上也有一定的历史原因，因为今天的主流 C++ 编译器，在异常关闭和开启时应该已经能够产生性能差不多的代码（在异常未抛出时）。代价是产生的二进制文件大小的增加，因为异常产生的位置决定了需要如何做栈展开，这些数据需要存储在表里。典型情况，使用异常和不使用异常比，二进制文件大小会有约百分之十到二十的上升。LLVM 项目的编码规范里就明确指出这是不使用 RTTI 和异常的原因 [3]：

> In an effort to reduce code and executable size, LLVM does not use RTTI (e.g.`dynamic_cast<>;`) or exceptions.

我默默地瞅了眼我机器上 88MB 大小的单个 clang-9 可执行文件，对 Chris Lattner 的决定至少表示理解。但如果想跟这种项目比，你得想想是否值得这么去做。你的项目对二进制文件的大小和性能有这么渴求吗？需要这么去拼吗？

异常的问题

异常当然不是一个完美的特性，否则也不会招来这些批评和禁用了。对它的批评主要有两条：

- 异常违反了“你不用就不需要付出代价”的 C++ 原则。只要开启了异常，即使不使用异常你编译出的二进制代码通常也会膨胀。
- 异常比较隐蔽，不容易看出来哪些地方会发生异常和发生什么异常。

对于第一条，开发者没有什么可做的。事实上，这也算是 C++ 实现的一个折中了。目前的主流异常实现中，都倾向于牺牲可执行文件大小、提高主流程（happy path）的性能。只要程序不抛异常，C++ 代码的性能比起完全不做错误检查的代码，都只有几个百分点的性能损失 [4]。除了非常有限的一些场景，可执行文件大小通常不会是个问题。

第二条可以算作是一个真正有效的批评。和 Java 不同，C++ 里不会对异常规约进行编译时的检查。从 C++17 开始，C++ 甚至完全禁止了以往的动态异常规约，你不再能在函数声明里写你可能会抛出某某异常。你唯一能声明的，就是某函数不会抛出异常——`noexcept`、`noexcept(true)` 或 `throw()`。这也是 C++ 的运行时唯一会检查的东西了。如果一个函数声明了不会抛出异常、结果却抛出了异常，C++ 运行时会调用 `std::terminate` 来终止应用程序。不管是程序员的声明，还是编译器的检查，都不会告诉你哪些函数会抛出哪些异常。

当然，不声明异常是有理由的。特别是在泛型编程的代码里，几乎不可能预知会发生些什么异常。我个人对避免异常带来的问题有几点建议：

1. 写异常安全的代码，尤其在模板里。可能的话，提供强异常安全保证 [5]，在任何第三方代码发生异常的情况下，不改变对象的内容，也不产生任何资源泄漏。
2. 如果你的代码可能抛出异常的话，在文档里明确声明可能发生的异常类型和发生条件。确保使用你的代码的人，能在不检查你的实现的情况，了解需要准备处理哪些异常。
3. 对于肯定不会抛出异常的代码，将其标为 `noexcept`。注意类的特殊成员（构造函数、析构函数、赋值函数等）会自动成为 `noexcept`，如果它们调用的代码都是 `noexcept` 的话。所以，像 `swap` 这样的成员函数应当尽可能标成 `noexcept`。

使用异常的理由

虽然后面我们会描述到一些不使用异常、也不使用错误返回码的错误处理方式，但异常是渗透在 C++ 中的标准错误处理方式。标准库的错误处理方式就是异常。其中不仅包括运行时错误，甚至包括一些逻辑错误。比如，在说容器的时候，有一个我没提的地方是，在能使用 `[]` 运算符的地方，C++ 的标准容器也提供了 `at` 成员函数，能够在下标不存在的时候抛出异常，作为一种额外的帮助调试的手段。

```
#include <iostream>   // std::cout/endl
#include <stdexcept>  // std::out_of_range
#include <vector>     // std::vector
using namespace std;
```

```
vector<int> v{1, 2, 3};
```

```
v[0]
```

```
1
```

```
v.at(0)
```

```
1
```

```
v[3]
```

```
-1342175236
```

```
try {
  v.at(3);
}
catch (const out_of_range& e) {
  cerr << e.what() << endl;
}
```

```
_M_range_check: __n (which is 3) >= this->size() (which is 3)
```

C++ 的标准容器在大部分情况下提供了强异常保证，即，一旦异常发生，现场会恢复到调用函数之前的状态，容器的内容不会发生改变，也没有任何资源泄漏。前面提到过，`vector` 会在元素类型没有提供保证不抛异常的移动构造函数的情况下，在移动元素时会使用拷贝构造函数。这是因为一旦某个操作发生了异常，被移动的元素已经被破坏，处于只能析构的状态，异常安全性就不能得到保证了。

只要你使用了标准容器，不管你自己用不用异常，你都得处理标准容器可能引发的异常——至少有 `bad_alloc`，除非你明确知道你的目标运行环境不会产生这个异常。这对普通配置的 Linux 环境而言，倒确实是对的……这也算是 Google 这么规定的一个底气吧。

虽然对于运行时错误，开发者并没有什么选择余地；但对于代码中的逻辑错误，开发者则是可以选择不同的处理方式的：你可以使用异常，也可以使用 `assert`，在调试环境中报告错误并中断程序运行。由于测试通常不能覆盖所有的代码和分支，`assert` 在发布模式下一般被禁用，两者并不是完全的替代关系。在允许异常的情况下，使用异常可以获得在调试和发布模式下都良好、一致的效果。

标准 C++ 可能会产生哪些异常，可以查看参考资料 [6]。

内容小结

今天我们讨论了使用异常的理由和不使用异常的理由。希望通过本讲，你能够充分理解为什么异常是 C++ 委员会和很多大拿推荐的错误处理方式，并在可以使用异常的地方正确地使用异常这一方便的错误处理机制。

如果你还想进一步深入了解异常的话，可以仔细阅读一下参考资料 [4]。

课后思考

你的 C++ 项目里使用异常吗？为什么？

欢迎留言和我交流你的看法。

参考资料

[1] Google, “Google C++ style guide”. https://google.github.io/styleguide/cppguide.html#Exceptions

[2] Reddit, Discussion on “Examples of C++ projects which embrace exceptions?”.

https://www.reddit.com/r/cpp/comments/4wkkge/examples_of_c_projects_which_embrace_exceptions/

[3] LLVM Project, “LLVM coding standards”. https://llvm.org/docs/CodingStandards.html#do-not-use-rtti-or-exceptions

[4] Standard C++ Foundation, “FAQ—exceptions and error handling”.https://isocpp.org/wiki/faq/exceptions

[5] cppreference.com, “Exceptions”. https://en.cppreference.com/w/cpp/language/exceptions

[5a] cppreference.com, “异常”. https://zh.cppreference.com/w/cpp/language/exceptions

[6] cppreference.com, “std::exception”. https://en.cppreference.com/w/cpp/error/exception

[6a] cppreference.com, “std::exception”. https://zh.cppreference.com/w/cpp/error/exception

精选留言

tt

文中下面的一句话：

“首先是内存分配。如果 new 出错，按照 C++ 的规则，一般会得到异常 bad_alloc，对象的构造也就失败了。这种情况下，在 catch 捕捉到这个异常之前，所有的栈上对象会全部被析构，资源全部被自动清理。”

谈的是new在分配内存时的错误，是堆上内存的错误，但自动被析构的却是栈上的对象。一开始我想是不是笔误了，但仔细想想，堆上的东西都是由栈上的变量所引用的，栈上对象析构的过程，堆上相应的资源自然就被释放了。而且被释放的对象的范围还被栈帧限定了。

2019-12-09 08:26

作者回复

对，这就是 RAII，非常重要。

学习速度飞快啊。

2019-12-09 09:36

tech2ipo

老师，你好。目前主流的开源项目中，有没有使用了异常的优秀的C++开源项目？可以用来作为参考案例。

2019-12-13 01:23

作者回复

我不觉得用异常有什么特别的地方，因而用异常的我个人没觉得有什么特别可参考的。

由于历史原因，有不少大名气的 C++ 程序没有使用异常，特别是 Google 的项目，比如 Chromium。不用异常，实际上是对用户友好（可执行文件略小，性能有可能有小提升），而对开发者更累。

我知道用到异常的一些项目：

- Boost
- C++ REST SDK
- pytorch
- pybind11
- Armadillo
- nlohmann/json
- cppcheck
- OpenCV

- LibreOffice

这篇文章也可以看一下：

https://cppdepend.com/blog/?p=311

2019-12-14 16:31

中年男子

用到异常的时候倒不是很多，但是异常千万别乱用，害人害己，
曾经同事离职，接手他项目的代码，把我坑的，几乎所有能引起crash的地方都用try catch 捕获异常，然而不处理异常，比如非法指针， 这种bug居然用try catch 来规避，坑了我两个月时间才把程序搞稳定了，现在想起他来，心里还有一句mmp想送给他。。。

2019-12-09 16:28

作者回复

任何东西用得不好都是坑。有朋友遇到小项目里用了一大堆（不必要的）设计模式，把代码硬生生弄得不可理解。不能说设计模式就是不好，是不?

MSVC 可以用 ... 捕获非法指针操作，这也是极易被误用的功能。以前也遇到过一次，一不小心用了这个功能，把明明在调试时可以发现的崩溃变成了程序的怪异行为。不过，严格来讲这不属于 C++ 的异常……这实际上是 Windows 的 SEH，纯 C 里都能做得到。

2019-12-09 20:49

Encoded
Star

一、使用异常
1.异常处理并不意味着需要写显示的try和catch。异常安全的代码，可以没有任何try和catch
2.适当组织好代码，利用好RAII，实现矩阵的代码和使用矩阵的代码都可以更短、更清晰，处理异常一般情况会记日志或者向外界用户报告错误。
二、使用异常的理由
1.vector C++标准容器中提供了at成员函数，能够在下标不存在的时候抛出异常(out_of_range)，作为一种额外的帮助调试手段
2.强异常保证，就是一旦异常发生，现场会恢复到调用异常之前的状态。(vector在元素类型没有提供保证不抛异常的移动构造函数的情况下，在移动元素时会使用拷贝构造函数，一旦某操作发生异常，就可以恢复原来的样子)
3.只要使用标准容器就都的处理可能引发的异常bad_alloc
4。可以使用异常，也可以使用assert
课后思考
你的C++项目里使用过异常吗？为什么？
答：按老师课里说的，只要使用了标准容器就得考虑使用处理异常(bad_alloc)，所以，大部分C++代码如果保证安全的情况下都的考虑这个异常。当然也在别的地方，之前在读取配置文件(json文件)字段的时候加过，如果读取失败，异常抛出

2019-12-22 10:51

作者回复

OK。很好！

2019-12-22 14:00

亮

看到老师说了部分开源的异常优秀的C++开源项目，老师能否推荐些现在流行的，能逐步深入的网络编程方面的C++开源项目看呢，从入门到深入的都推荐一些吧。谢谢老师

2019-12-17 16:10

作者回复

网络就看 Boost.Asio 吧。这个将是未来 C++ 网络标准库的基础。

2019-12-18 07:28

不谈

作为java程序员刚开始对于C++这种规则完全不能理解，看了文章之后又理解了，不使用异常是有道理的。现在理解了vector会在元素类型没有提供保证不抛异常的移动构造函数的情况下，在移动元素时会使用拷贝构造函数这句话的含义了。C++就是

这么与众不同

2019-12-17 23:13

作者回复

这种行为是和值语义密切关联的——和 Java 不同。而且，要在构造函数和运算符重载中表达错误，异常是唯一的方法。

2019-12-18 07:35

何敬

请问libbreakpad跟underflow_error有关联么？现在使用breakpad捕获异常，有很多underflow_error的信息

2019-12-13 16:17

作者回复

对不起，我对 libbreakpad 完全不了解。

2019-12-13 19:38

何敬

请问underflow_error 什么情况下会抛出?

2019-12-11 23:32

作者回复

https://zh.cppreference.com/w/cpp/error/underflow_error

定义作为异常抛出的类型。它可用于报告算术下溢错误（即计算结果是非正规浮点值的情形）。

标准库组建不抛此异常（数学函数按指定于 math_errhandling 的方式报告下溢错误）。然而第三方库使用它。例如，若启用了 boost::math::policies::throw_on_error （默认设置），则 boost.math 抛出 std::underflow_error 。

2019-12-12 07:33

花晨少年

C++ 虽然支持运算符重载，可你也不能使用，因为你没法返回一个新矩阵……

不太理解这句话的意思，是用运算符来替换构造函数吗，运算符为啥不能返回新矩阵?

2019-12-11 01:11

作者回复

因为返回值得留给错误码啊……既然不能用异常。

2019-12-11 08:09

neilyu

C++ 虽然支持运算符重载，可你也不能使用，因为你没法返回一个新矩阵……

//请问这句话怎么理解？为啥不能返回一个新矩阵呢?

2019-12-10 23:00

作者回复

因为返回值留给错误码了啊……

2019-12-11 08:09

李蔚韬

老师，对于异常的第一条批评我不太理解，什么叫“只要开启异常，即使不使用”，这里的开启是指什么呢?

2019-12-09 09:51

作者回复

GCC/Clang 下的 -fexceptions（缺省开启），MSVC 下的 /EHsc（我要求大家需要用的，Visual Studio 项目里也会自动用）。

我刚试了，用 GCC，加上 -fno-exceptions 命令行参数，对于下面这样的小程序，也能看到产生的可执行文件的大小的变化。

```
#include <vector>

int main()
{
std::vector<int> v{1, 2, 3, 4, 5};
v.push_back(20);
}
```

2019-12-09 13:51

禾桃

“异常处理并不意味着需要写显式的 try 和 catch。异常安全的代码，可以没有任何 try 和 catch。”

出现异常时，如果没有任何的try catch,只是让std::terminate, 即使没有资源泄漏之类的，感觉什么也做不了了，感觉还是应该要catch,做点啥，至少得记录一下哪抛出异常，什么异常，然后再主动的std::terminate。

2019-12-09 08:23

作者回复

外围（比如main里）当然是要写catch的。（我们一般也不会主动去调terminate；退出的话一般用exit。）但异常安全的代码本身可以没有任何try和catch。

学得真快。

2019-12-09 09:40

禾桃

happy path—-> hot path:)

2019-12-09 08:11

作者回复

不是，就是happy path。愉快的（乐观情况下的）执行路径，而不是说是否频繁。

2019-12-09 09:32