从 C++ 的错误处理说起

2020-04-16

错误处理是一个非常重要的软件工程问题。对软件中出现的非致命错误的不当处理，是几乎所有的灾难性系统故障的诱因。[1] 编程语言往往需要提供一些用于错误处理的语言设施，这些设施反过来会影响项目中错误处理的方式。不同的语言错误处理方式不同。例如 Java 采用基于 try-throw-catch 语法的异常机制，而 Go 语言则选择手动检测函数返回的 `error` 对象。一个令人惊讶的事实是，C++ 到现在还没有一个被广泛接受的错误处理方式。

尽管 C++ 当中引入了基于 try-throw-catch 的异常（Exception）处理机制，而且将其应用到了标准库当中，但这种处理方式遭到了广泛的批评。Linus Torvalds 曾说 C++ 的异常处理「从根本上坏掉了[2]」。一些重要的 C++ 风格指南，如《谷歌 C++ 风格指南[3]》、《联合打击战斗机飞行器 C++ 代码规范（JSF++）[4]》，都禁止使用 C++ 异常。其中制定 JSF++ 的 C++ 专家还包括了 C++ 创始人 Bjarne Stroustrup。除此之外，有调查显示 52% 的 C++ 开发者都至少有一部分项目完全禁用异常。

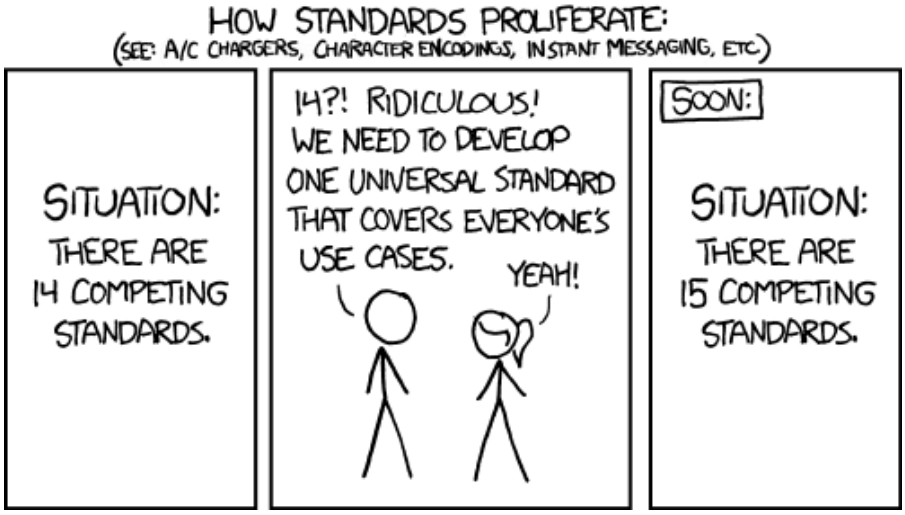
这导致了非常严重的问题。由于异常在 C++ 标准库中广泛的使用，禁用异常的 C++ 已经不是标准的 ISO C++ 而成为了一种方言。[5]这与标准委员会的使命，当然也与广大程序员的切身利益相违背。基于此，微软的 C++ 专家 Herb Sutter，同时也是 C++ 标准委员会的召集人，在他撰写的 C++ 标准提案 P0709 当中提出了一种解决方案。这份长达 65 页的标准提案并不晦涩难懂，相反，它深入浅出地探讨了异常处理的方方面面，并针对存在的问题给出了令人信服的解决方案。即使这个提案距离真正进入标准还遥遥无期，但阅读这份文档仍可令人受益匪浅。本文将以这份文档为蓝本，讨论一些错误处理方面的重要问题。虽然 Sutter 是在 C++ 的语境下讨论这些问题的，但本文有意弱化了 C++ 的语境，并加入了与其他语言的对比。

C++ 的错误处理出了什么问题?

为何在其他语言（Java、Python 等）中广泛应用的异常机制在 C++ 之中如此臭名昭著？因为现有的异常实际上和 C++ 的**核心价值**不符。主要分两点原因：

- 现有的异常不符合**零开销抽象**。异常不仅会引入多余的运行时开销，而且还会使编译产生的二进制文件急剧增大。不同的环境下增大的比例不同，Sutter 收集到的资料报告了 +15%，+16%，+38% 甚至 +52% 的文件体积增大。
- 现有的异常处理是**非确定性的**。由于异常对象储存在堆上，而且用到了 RTTI 特性，无论是运行的时间，还是运行的内存占用，都不能得到保证。

以上的原因使得异常处理在高实时性，低内存的应用场景下变得不可接受。为了解决这个问题，标准库中引入了「`error_code` [6]」作为一种替代方案。但这个方案并没有解决所有问题，反而把问题搞复杂了。新加入标准库的组件如 `<filesystem>` 陷入了需要同时支持两种错误处理方案的尴尬境地。标准库的两个方案都有缺陷，人们开始寻求标准之外基于库的解决方案，Expected 和 Outcome 就是这样的尝试。但它们最后达成的效果，只能用 xkcd 927 来描述：



重构「错误处理」

前面提到的已有的对于 C++ 错误处理的尝试并非徒劳无功，它们为后来的设计者提供了经验和教训。在充分理解现有方案的优点和弊端之后，Herb Sutter 准备提供一个新的大一统方案，解决之前提出的**所有**问题。这当然有陷入 xkcd 927 的「N+1 种标准」危险，但解决方案也是有的：只要新的方案能够至少比一种现有的方案绝对更优，亦即，在完全涵盖某种方案的优点的前提下，克服其一部分缺点，就可以完全取而代之，至少不会使现有的方案增加。

当然仅仅避免「N+1」的问题当然还不够。为了提出一个真正能满足各种需求的方案，Sutter 对错误处理做了认真的分析。在 P0709 中包含了许多重要的见解。

> 注意在本文中「错误」和「异常」的用法区别。「错误（error）」是一个通用的软件工程概念，泛指程序编写时和运行时出现的各种异常情况，而「异常（exception）」是一个特定的 C++ 概念。

逻辑错误和内存错误应特殊处理

逻辑错误也就是 bug，是程序员由于考虑不周产生的错误。广受诟病的 `ArgumentNullException` 以及对负数开平方根，数组下标溢出错误等，都是属于这一类型。Sutter 指出，这些错误总是会导致状态损坏，这种编写时的错误绝不可能在运行时得到恢复，所以程序无需向上层代码报告错误，而是应该特殊专门处理（比如直接崩溃掉）。这一部分问题有望在尚未合并的 C++ 特性 Contracts 中得到解决。

而对于内存分配错误（allocation failure），基于以下的原因，Sutter 认为也不应该抛出异常。

- **根本没有人能真正正确的处理内存异常。** Sutter 与 Clow 在 2019 年编写了一套测试框架 babb，对 Visual C++ 标准库的 STL 部分，Microsoft PowerPoint 的 File>Open 功能和 Microsoft Word[7] 做了测试。这些代码都宣称在编写时仔细考虑了内存错误，但是都没有通过测试。
- **很少有人花心思编写代码来正确的处理内存异常。** 尽管 VC++ STL 存在不能正确抛出 `bad_alloc` 的问题，但 2015-2019 年间总共只收到了 6 次有关的错误报告。由 `bad_alloc` 派生出的 `bad_array_new_length` 根本就没人使用。
- 由于各个平台的内存管理实现不同，**根本不可能在所有平台上正确实现堆分配错误的抛出。**

令人惊讶的是，将逻辑错误和内存错误移除出正常的异常处理机制之后，90% 的函数都可以变成 `noexcept` 的[8]。这将带来巨大影响。在一次内部会议上，微软的软件架构师 Pavel Curtis 曾提到，如果 STL 中引入了 contracts，并且 `bad_alloc` 能够立刻停止程序（fail fast），那么整个 Windows 代码都可以直接使用标准的 STL 库。

应当抛出相同类型、不同值的错误对象，而不是让错误类型属于某个继承树

历史确实是在曲折中前进的。在错误处理方面，工业界走过的一个巨大弯路便和异常（Exception）的继承层次结构有关。Java 和 C++ 都选择了把所有的异常都挂在一个以 Exception 类（C++ 中为 `std::exception`）为根的一个继承树上面。这样捕获异常时，便可以在层次结构中选择一个合适的基类进行捕获。更进一步地，语言还提供了「异常规范」，把一个函数可能产生的异常作为函数的签名标注出来，以期实现更严谨的错误处理。

然而这一切在被加入标准之后，终于逐渐被证明是弊大于利的。现在在网上存在大量关于 Java 异常规范的批评，ISO C++ 的第二个大版本 C++11 也赶紧弃用了异常规范。而且在 C++ 这种动态性很低的语言中，不仅是异常规范，基于继承的异常层次结构也有很大问题。注意到异常捕获通常有两类场景：

- **不关心异常类型的捕获** 这种被称为「distant catch」的捕获往往在最终的调用端代码中执行，被 Sutter 认为是「最有价值的（this most valuable）」。由于函数的封装性，在调用树中较远处的异常捕获并不知道也不关心[9]发生异常的具体类型。Sutter 注意到，由于抛出的异常类型无法在编译时确定，大量的高层代码最终只好编写 `catch(...)` 来捕获所有类型的异常。
- **关心异常类型的捕获** 这种捕获常发生在库代码中，这些代码需要了解底层逻辑，往往伴随着「重新抛出」等操作。对于这样的情况，基于继承的异常层次结构也不如使用 `error_code` 的方案。因为异常只能在继承结构中向上转换，难以实现 `error_code` 那样对错误的组合。另外，`error_code` 在传播，转换方面也比 `exception` 做得好。

错误总是应该得到处理

Sutter 指出，任何允许错误被悄无声息地忽略掉的错误处理方案，都会导致程序健壮性和安全性长期的损失，同时影响语言的声誉。但错误应该得到处理，并不意味着程序员总是应该编写大量的处理代码（像 Go 那样）。错误可以被隐式的处理，例如通过某种机制自动向上级调用者传播。

通向解决之道

基于以上的见解，Herb Sutter 给出了最终的方案。`throw` 关键字被用来抛出 `std::error` 类型，为了和传统的 `throw exception` 的抛出方式做区分，`throw error` 的函数需要加上 `throws` 标记。`std::error` 作为值类型被使用，在 `catch` 语句中直接按值捕获。下面是提案中的示例。

```
int safe_divide(int i, int j) throws {
  if (j == 0)
    throw arithmetic_errc::divide_by_zero;
  if (i == INT_MIN && j == -1)
    throw arithmetic_errc::integer_divide_overflows;
  if (i % j != 0)
    throw arithmetic_errc::not_integer_division;
  else return i / j;
}

double caller(double i, double j, double k) throws {
  return i + safe_divide(j, k);
}

int caller2(int i, int j) noexcept {
```

```
  try {
    return safe_divide(i, j);
  } catch(error e) {
    if (e == arithmetic_errc::divide_by_zero)
      return 0;
    if (e == arithmetic_errc::not_integer_division)
      return i / j; // ignore
    if (e == arithmetic_errc::integer_divide_overflows)
      return INT_MIN;
    // Adding a new enum value “can” cause a compiler
    // warning here, forcing an update of the code (see Note).
  }
}
```

同归却殊途

尽管 P0709 的方案足够有说服力，在 C++ 委员会全球会议中的投票中，这份提案的许多部分得到了没有反对票的支持。但将这样一个不小的语言改动合并到 C++ 标准之中仍旧是一件困难的事情。自 2018 年 5 月 2 日 P0709R0 提交之后，这份提案迄今已经修订了 5 个版本，然而这份提案仍然存在未解决问题。它不仅无法进入 C++20 的标准之中，甚至 C++23 都不一定能包含全新的错误处理机制。

然而正如前面所反复提到的那样，错误处理是一个非常通用的概念，经验和教训在各个语言之中都是通用的。现代语言如 Haskell，Rust 和 Swift 在错误处理方面都得出了一致的结论。借助 Fehler 库，上面的 C++ 示例代码在 Rust 中几乎具有相同的实现方式和语义：

```
#[throws(ArithmeticError)]
fn safe_divide(i: i32, j: i32) -> i32 {
    if j == 0 {
        throw!(ArithmeticError::DivideByZero);
    }
    if i == i32::MIN && j == -1 {
        throw!(ArithmeticError::IntegerDivideOverflows);
    }
    if i % j != 0 {
        throw!(ArithmeticError::NotIntegerDivision);
    }
    i / j
}

#[throws(ArithmeticError)]
fn caller(i: i32, j: i32, k: i32) -> i32 {
    i + safe_divide(j, k)
}

fn caller2(i: i32, j: i32) -> i32 {
    safe_divide(i, j).unwrap_or_else(|e| match e {
        ArithmeticError::DivideByZero => 0,
```

```
            ArithmeticError::NotIntegerDivision => i / j,
            ArithmeticError::IntegerDivideOverflows => i32::MIN,
        })
    }
```

虽然在设计思路上趋于相同，但 Rust 与 C++ 在语言层面的落实程度，却是大相径庭的。Rust 的错误处理机制自设计之初就选择了返回值类型这一正确的方向。在接下来的语言演进当中，新的设计想法几乎总是以库的形式首先出现，在经过了大量真实项目的实验之后进入语言本身。C++ 的 boost 库和 TS 库也部分承担了这样的功能，但 Rust 的强大包管理器无疑大幅度降低了试错成本，而依靠 Rust 强大的宏系统，第三方库具有对语法本身做扩展的能力。诚然，Rust 也会犯错，最初的 `std::error::Error` 提供的两个接口现在都已经分别在 1.33 和 1.42 被弃用，但这取代这两个的新接口已经被大量实践证明确实更优。比起 C++ 从 C++98 开始几十年的毫无进展，Rust 的错误处理的确在切实的进步。

C++ 作为一个有这悠久历史的语言，大量的基础代码库是其坚强生命力的重要来源。但大量的旧有代码也给语言本身戴上了沉重的枷锁。虽然从 C++11 开始，C++ 也成为了「现代」的 C++（Modern C++），但新兴的现代语言如 Rust，从发展的眼光来看，已经在许多方面超过了 C++。新项目在进行技术选型时，也应当首先对这些语言加以考虑。

结语

为了达成足够的可靠性，系统不可避免地会变复杂。[10] 软件工程的本质也许就是以人类的智慧去掌控人类无法真正理解的复杂系统。现代语言提供了有力的工具来简化错误处理，但程序员仍然需要学习大量的知识以高效地应用它们。这大概也是软件复杂度的体现吧。

---

1. “almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software”, Simple Testing Can Prevent Most Critical Failures —— http://www.eecg.toronto.edu/~yuan/papers/failure_analysis_osdi14.pdf
2. “the whole C++ exception handling thing is fundamentally broken.” —— https://lkml.org/lkml/2004/1/20/20
3. Google C++ Style Guide
4. The Joint Strike Fighter Air Vehicle C++ Coding Standards
5. 这两种方言的鸿沟可能比想象中的更大。例如，没有办法写出泛型代码，同时支持「启用异常」和「禁用异常」。
6. `std::error_code` 可以算作是错误码的面向对象实现。具体的用法在 Andrzej's C++ Blog 的文章 Your own error code 中有介绍。
7. 对 Word 的测试是使用另外的测试框架进行的。
8. Duffy. "Safe native code". Joe Duffy’s Blog, 2015-12-19 —— http://joeduffyblog.com/2015/12/19/safe-native-code/
9. 这里的「不关心」，是指不用编程的方式理解具体的错误内容，而是将其以统一的方式提供给人类，如输出日志。
10. C. A. R. Hoare. "The emperor's old clothes." ACM Turing award lectures. 2007. 1980.

#programming #cpp #rust

‹ \(\LaTeX\) 公式 C++ 每三年才解决一点点问题 ›

0 Comments - *powered by utteranc.es*

Write Preview

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub

从 C++ 的错误处理说起 由 Peng Guanwen 采用 知识共享 署名-非商业性使用-相同方式共享 4.0 国际 许可协议进行许可。