Roberto Gomes Rosmaninho Neto

arXiv:2301.10353v1 [cs.PL] 24 Jan 2023

将 Swift 错误处理模型桥接到 C++

贝洛奥里藏特，米纳斯吉拉斯

2022

Roberto Gomes Rosmaninho Neto

将 Swift 错误处理模型桥接到 C++

提交给米纳斯吉拉斯联邦大学计算机科学学士学位的计算机专业毕业项目报告

米纳斯吉拉斯联邦大学
精确科学研究所
计算机科学系

指导教师 Fernando Magno Quintão Pereira

贝洛奥里藏特，米纳斯吉拉斯
2022

原创声明

本人谨此声明，所提交的学位论文为本人独立完成。所有直接或间接使用的资料均已列入参考文献。本人进一步声明，未曾将本学位论文提交至其他任何机构以获取学位。

献给我的朋友和家人，一切的意义所在。

致谢

我谨向导师 FernandoPereira 致以谢意，感谢他为我提供了人生中最好的机会；感谢同事 Angelica Moreira 和 Breno Campos 一路以来对我的支持；也感谢父母 Roberto 和 Rogeria Rosmaninho，帮助我走到今天！

摘要

错误处理是指程序响应错误状况并从中恢复的过程。在 Swift 中，错误由符合 Error 协议的类型的值表示。抛出错误可以表明发生了意外情况，导致程序无法继续按正常流程执行。throw 语句用于抛出错误。可选返回值用于表示值的缺失；但操作失败时，了解失败原因往往很有用，这样代码便能做出相应处理。因此，我提议将 Swift 错误处理模型桥接到 C++ ，以提升这些编程语言之间的互操作性。核心思路是能够抛出一个 C++ 异常，其中保存了一个抛出的 Swift 错误，而该错误必须由一个 C++ 类来表示。此外，为了支持不使用异常的 C++ 程序，我提议为抛出函数增加一种互操作模式。禁用 C++ 异常时，C++ 函数应返回一个结果值，其中包含函数返回的值，或 Swift 错误值：Swift::Exception<T> 类。

关键词：编译器、C++、Swift、std::expected、错误处理、异常、编程语言

程序代码列表

1 C++ 中异常处理的示例.11 2 简单 throw 语句的示例.11 3 可抛出错误的函数声明示例。此程序与 1 几乎等效.12 4 调用可抛出错误的函数的正确实现示例.12 5 Swift 对应函数抛出错误时，可抛出 NaiveException 的函数示例.16 6 可抛出 Error，但没有 throw 语句的 Swift函数 . 17 7 用于替换函数类型和返回值、以适配异常使用方式的宏定义.19 8 用户定义的 Error 以及可抛出该错误的 Swift函数示例 . . 20 9 创建 Swift 程序桥接头文件的命令.20 10 在 C++ 中使用可抛出且可捕获 Error 的 Swift函数示例.21 11 在 C++ 中使用并处理可抛出错误的 Swift函数示例.22 12 构建、链接并执行调用 Swift函数且可抛出异常的 C++ 程序的命令. 23 13 使用基于 std::expected 提案的 expected 类型处理 Swift 错误.24 14 构建、链接并执行调用 Swift函数且不会抛出异常，而是使用 Swift::Expected<T> 类的 C++ 程序的命令.25

目录

1 引言.81.1 总体目标 .81.2 具体目标 .92

理论参考.10
2.1 历史 .102.2 异常.10

2.3 Swift 错误处理模型.11 2.4 std::expected<T,E> .13

3 方法论.15
3.1 C++ 与 Swift 互操作性现状 . . . . . 15 3.2 初始实现 .15 3.3 最终实现 .17

4 结果与示例.20

参考文献.26

1 引言

Swift 是一种通用编程语言，采用现代方法构建，注重安全性、性能和软件设计模式。2015年12月3日，Swift语言及其支持库、调试器和包管理器以 Apache 2.0许可证并附带运行时库例外的方式发布。源代码托管在 GitHub 上，任何人都可以轻松获取代码、自行构建，甚至创建拉取请求，为项目贡献代码。（Swift.org，2022c）改进和优化 Swift 编译器一直是 Swift社区的主要目标之一，社区主要在 Swift Online Fóru m 上讨论相关主题的构想、目标、实现，以及具体实现。

越来越多的 API 和项目使用 Swift 编程语言，这不仅引起了 Swift 社区的关注，也吸引了 C++社区的关注。因此，如果 C++ 社区中的任何人想要使用以 Swift 开发的某个 API，或者想逐步将其应用程序的代码库从 C++ 迁移到 Swift，就应该能够依靠这两种语言之间的互操作性来实现。这正是 C++ 与 Swift 互操作如此重要的原因。

为了在 C++ 代码中使用 Swift API 或逐步采用 Swift，开发者必须确保应用程序不会因 C++ 代码或 Swift 代码中的错误而崩溃。在 C++ 代码中处理来自 C++ 的错误很简单。另一方面，如果通过桥接头文件在 C++ 代码库中调用的 Swift 函数抛出错误，目前就无法从 C++ 端得知发生了什么。因此，程序无法处理来自 Swift 的错误，并且可能在没有任何有意义的错误信息的情况下崩溃。

本项目旨在开发并实现 Swift 编译器基础设施，通过桥接头文件将 Swift 错误处理扩展到 C++。

1.1 总体目标

本项目旨在扩展 Swift 编译器，使 Swift 函数能够将错误传递到 C++ 代码中。第一项修改是调整当前实现，使其能够将“Swift::Error” C++ 类转换为用户定义的错误。第二项修改是提供另一种 C++ 类，它同样封装 Swift 错误，但会将其封装在一个 *std::expected*(cplusplus.com，2021) 风格的实现中，供禁用 C++ 异常的用户使用，以便他们自行处理错误，而不是抛出错误。

1.2 具体目标

本项目的目标可以概括为：

- 创建一个新的 C++ 类，例如 *std::exception*，并存储一个 Swift 错误值。
- 扩展这种 C++ Swift 错误表示，使其能够转换为用户定义的错误。
- 生成一个 C++ 类，类似于*std::expected*(cplusplus.com，2021)，为不使用 C 的客户端提供错误处理++ 异常。
- 根据 cpp_exceptions 标志值自动切换错误处理模型。

2 理论参考

2.1 历史

C语言的历史++ 始于 1979年，当时 Bjarne Stroustrup 正在为他的博士论文开展研究。他开始着手开发“C with Classes”，顾名思义，它旨在成为 C语言的超集。他的目标是将面向对象编程引入 C语言。除了 C语言的所有特性外，他的语言还包含类、基本继承、内联、默认函数参数和强类型检查。（Albatross，2021）

第一个 C with Classes 编译器名为 Cfront，它源自一个名为 CPre 的 C编译器。它是一个用于将 C with Classes 代码转换为普通 C 代码的程序。后来，Cfront 于 1993 年被弃用，因为在其中集成新功能变得困难，尤其是 C++ 异常。（Albatross，2021）

2.2 异常

异常提供了一种在程序中应对异常情况（例如运行时错误）的方式：将控制权转交给称为处理程序的特殊函数。

要捕获异常，需要将一部分代码置于异常检查之下。具体做法是将这部分代码放在 try 块 中。当该代码块内出现异常情况时，会抛出异常，并将控制权转交给异常处理程序。如果没有抛出异常，代码就会正常继续执行，所有处理程序都会被忽略。（cplusplus.com，2021）

在 ***try*** 块中使用 throw关键字抛出异常。异常处理程序使用关键字 ***catch*** 声明，且必须紧接在 try 块之后：

```cpp
// exceptions
#include <iostream>
using namespace std;

int main () {
  try {
    throw 20;
  } catch (int e) {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

代码清单 1 – C++ 中异常处理示例。

需要进行异常处理的代码放在 try 块中。在此示例中，这段代码只是将一个整数作为异常抛出：

```cpp
throw 20;
```

代码清单 2 – 简单 throw 语句示例。

2.3 Swift 错误处理模型

在 Swift 中，错误由符合 Error 协议的类型的值表示。抛出错误意味着发生了意料之外的情况，正常的执行流程无法继续。

当函数遇到错误情况时，它会抛出一个错误。该函数的调用方随后可以捕获该错误并作出适当响应。（Swift.org，2022a）

```
enum numError: Error {
    case twenty
}

func main () throws -> Int {
    do {
        throw numError.twenty
    } catch {
        print("An exception occurred. Exception Nr. \(error)");
    }
    return 0;
}
```

列表 3 – 可抛出错误的函数声明示例。这几乎等同于程序 1。

函数在声明中包含 ***throws***（抛出异常）关键字，以表明它可能抛出错误。关键字 ***throw*** 在 C++ 中也曾使用，至少在 C++17 标准之前是如此。（ cppreference.com, 2022）然而，与 Swift 不同，在 C++中，该关键字用作动态异常规范，后面跟着一个类型，表示带有此规范的函数可能抛出该类型或其派生类型的异常。另一方面，如果函数后跟 ***throw()***，编译器便知道它不会抛出异常。

调用可能抛出错误的函数时，需要在表达式前加上 try 关键字。在 Swift 中，这种结构称为 ***do-try-catch***。在选择使用异常的代码库中，这种结构非常常见。

```
do {
    try canThrowAnError()
    // no error was thrown
} catch {
    // an error was thrown
}
```

清单 4 – 调用可能抛出错误的函数的正确实现示例。

如果一个函数 throws（抛出异常），Swift 会自动将错误从当前作用域向外传播，直到某个 catch 子句处理该错误，即使该函数是从 C++程序中调用的也是如此。这正是促使我们开展这项工作的原因之一：通过提出两种在 C++中处理 Swift 错误的方案，改善 Swift 与 C++ 之间的互操作。

2.4 std::expected<T,E>

提案 p0323r3（Botet，2017b）提出了类模板 ***expected<T,E>*** ，其中包含以下两者之一：

- 类型为 T 的值，即预期值类型；或者
- 类型为 E 的值，即在出现意外结果时使用的错误类型。

该接口可以理解为：底层值是预期值（类型为 T）还是意外值（类型为 E）。这个想法最初来自 Andrei Alexandrescu 的 C++ and Beyond 2012:Systematic Error Handling in C++ Alexan-drescu(Alexandrescu, 2012)。该接口及其设计理由基于 ***std::optional***(Cacciola, 2013)。提案作者认为 ***expected<T, E>*** 是对***optional<T>*** 的补充，用来说明对象中为何不包含预期值。

需要重申的是，C++的两种主要错误处理机制是异常和返回码。良好的错误处理机制应具备以下特性(Botet, 2017b)：

1. 错误可见性：失败情况应贯穿代码审查；如果错误被隐藏，调试可能会很痛苦。2. 错误信息：错误应携带有关其来源、原因以及可能的解决方法的信息。3. 整洁代码：错误处理应置于单独的代码层中，并尽可能不可见。读者无需停下来，也能注意到特殊情况的存在。4. 非侵入式错误：错误不应独占专用于正常代码流程的通信通道，而应尽可能不显眼。例如，函数的返回值是一种通道，不应专门用于返回错误。

可以对***expected<T, E>*** 类进行同样的分析，并观察其相对于经典错误报告系统的优势。（Botet，2017b）

1. 错误可见性：它兼具异常和错误代码的优点。由于返回类型是 ***expected<T, E>***, 错误情况是可见的；如果用户想要获取其中包含的值，就不能忽略错误情况。

2. 信息：内容可任意丰富。

3. 整洁代码：Expected 的单子接口提供了一种框架，可将错误处理委托给代码的另一层。请注意，***expected<T, E>*** 也可以充当面向异常的代码与无异常环境之间的桥梁。

4. 非侵入式 使用返回信道，而不将其独占。

3 方法论

3.1 C++ 与 Swift 互操作性的现状

Swift 与 C++ 编程语言之间的互操作通过桥接头文件实现。该头文件是一个 C++ 文件，其中包含用户希望从 Swift 源代码导出到 C++ 程序的所有函数签名、类型和类声明。要生成此头文件，我们必须修改 Swift 编译器，使其能够正确识别并使用 C++ 类型、宏、函数及其他有用资源来表示我们希望导出的信息。

3.2 初始实现

这项工作的主要目标是继续将 Swift 错误处理转换为 C++ 错误处理。

为了实现这一目标，我联系了 Alex Lorenz，他是 Apple 员工，负责与 Swift 互操作工作组中的学生合作（Swift.org，2022b）。该小组由社区成员和来自 Clang 团队的 Apple 员工组成。我们每周都会召开同步会议，询问项目中遇到的阻碍，彼此了解并熟悉各自的项目，同时听取 Apple 方面的指导。此外，工作组还有一个公开的 Slack 频道，我继续在那里与 Alex 保持联系，询问简单的实现问题，并讨论 GitHub CI 测试和代码审查。

第一步是分别从用户的角度研究 Swift 和 C++ 错误处理模型，编写示例1 ，使用这两种编程语言，并分析各自的语法，如理论参考部分所示。随后，我开始研究 Swift 编译器，尤其是 PrintAs- Clang(Swift，2022) 目录中的代码生成部分。

第二步，也是我的首次正式贡献(Swift#59217，2022)，按照测试驱动开发的方式，创建了包含抛出错误的函数的测试。其思路是实现一个有两个分支和两个函数的简单错误枚举：第一个函数标记了“throws”标志，但从不抛出任何错误；第二个函数也标记了“throws”，但会抛出先前创建的错误枚举中的错误。

第三步，也是我第二次正式贡献（Swift#59787，2022），是修改编译器，删除所有将每个函数归类为“noexcept（不抛出异常）”函数的标志。在之前的实现中，从 C++ 的角度来看，没有函数能够

抛出错误。解决这种硬编码行为的方法，是在生成头文件代码时，检查 Swift 中声明的函数是否带有“throws（抛出异常）”标志。此修改使（Swift#59217，2022）中的测试如预期般通过，因为这些函数在 Swift 中可以抛出异常，但在 C++ 桥接头文件中却被标记为“noexcept（不抛出异常）”。

遗憾的是，最后这一步并不像预期的那样顺利。“GitHub CI”指出，整个测试套件中只有一个测试在“i386-apple-watchos2.0-simulator”架构上失败了。经过一番调试，我们发现函数签名中缺少两个默认参数，第一个叫“self”，第二个叫“error”。前者对后续步骤至关重要，因为它包含函数执行期间若抛出异常时的错误信息。

```
1  inline void throwFunction() {
2    void* opaqueError = nullptr;
3    void* self = nullptr;
4    _impl::$s9Functions13throwFunctionyyKF(self, &opaqueError);
5    if (opaqueError != nullptr)
6      throw (swift::_impl::NaiveException("Exception"));
7  }
```

列表 5——如果对应的 Swift函数 throws（抛出异常）一个错误，则会抛出 NaiveException 的函数示例。

以上示例展示了本项目的第三项贡献（Swift#60079，2022）：

- “if condition”用于检查“opaqueError”是否在 Swift函数执行期间发生了变化。
- NaiveException C++类。

第一项修改会测试 Swift函数是否 throws（抛出异常）。在这种情况下，“opaqueError”参数会保存识别所抛出错误所需的信息，而不是“nullptr”。第二项修改实现了一个 C++ class，用于构造一个包含消息的简单对象，并将其抛给 C++ caller 函数。此外，还可以通过“*getMessage()*”方法访问此消息。

3.3 最终实现

本项目的第一部分旨在为 C++提供 Swift 错误处理模型的最简版本，以便 C++ function caller 能够知道被调用的 Swift函数 throws（抛出异常）。本项目的第二也是最后一部分旨在通过提供通用的“***Swift::Error***”表示形式、用于识别函数所抛出的错误情况的基础设施，以及一个用于返回所抛出错误的新 C++ class ***Swift::Expected***”来专门化这方面的知识。

本项目的第四项贡献（Swift#60858，2022）是提出前述通用的“***Swift::Error***”表示形式，用它替换生成的桥接头文件中 C++ 函数表示里的“throw”语句中的“NaiveException”。该类使用“***swift__errorRetain***”和“***swift__errorRelease***”函数实现其构造函数和析构函数，以正确处理 Swift 错误，避免意外行为和内存泄漏。该类的主要构想是提供一种 Swift 错误类型的表示形式，使其能够在 C++ 端进行检查。这种表示形式在 C++中表现为 Swift 协议类型，并且能够表示所有 Swift 错误值。

该类的主要构想是取代对“std::exception”库的依赖；该库目前依赖于无法导入生成的桥接头文件的内容，并让该类充当 C++ 中所有 Swift 错误的“超类”。

```
1  class TestDestroyed {
2    deinit {
3      print("Test destroyed")
4    }
5  }
6
7  public struct DestroyedError : Error {
8    let t = TestDestroyed()
9  }
10
11 public func testCast(_ e: Error) {
12     let _ = e as? DestroyedError
13 }
```

列表 6 – 可以抛出错误但不包含 throw 语句的 Swift函数 t.

第五项贡献（Swift#61626，2022）是这个项目中最具挑战性的部分。它要求使用 lldb（调试器，2007年）跟踪和断点，在汇编级别调试示例 6，以找出复现第 12 行类型转换所需的信息，并使用我们的 C++ 基础设施。

在与 Apple 的 Alex 进行一对一调试会议时，我们发现，要动态转换错误，需要一些元数据信息、一种符号类型以及外部全局变量。其中一些必需信息可以硬编码，其他的则可以通过将它们声明为“external "C""函数和变量来获取，因为 Swift 编译器会使用它们构建 LLVM IR。

最后，在实现了所有辅助函数，以便将“***Swift::Error***”动态类型转换为表示所抛出错误情形的具体 Swift 类型后，我将“***as<T>()***”函数实现为“***Swift::Error***”类的一个方法，让用户能够以自然的方式在 C++ 中处理错误。

在第六项贡献（Swift#62197，2022）中，我改进了前面提到的动态类型转换：如果转换成功，就在值中返回“***Swift::Optional<T>***”类型；如果失败，则返回“***Swift::Optional<T>::none()***”。此外，我删除了不再需要的“***NaiveException***”，并将“***Swift:Error***”的实现移至定义“***Swift::Optional<T>***”的同一文件中，以便修改其返回类型。

最后，在第七项也是最后一项贡献（Swift#61823，2022）中，我开发并实现了自己版本的“***std::Expected<T>***”（Botet，2017a）提案，将其命名为“***Swift::Expected<T>***”，并为桥接头文件实现了支持，使其能够根据 C用户是否选择使用异常，通过向编译器传递“*-fno-exceptions*”标志，采用恰当的方式处理 Swift 抛出的错误。由于需要构建一个将被社区广泛使用的优质 API，本项目的第一部分极具挑战性。此外，这还是一项在社区中讨论了多年的提案的实现，而敢于实现自己版本的“***std::Expected<T>***”的人寥寥无几。

此外，Apple 提出的两项重要要求是：仅使用“***Swift:Error***”作为可返回的错误类型，并将错误或 T 类型的值存储在同一缓冲区中，以优化此类对象的内存使用。这项贡献的最后一部分也并非易事：为了避免在编译器和桥接头文件中重复代码，我使用了一个别名（*ThrowingResult*）和一个宏*(SWIFT_RETURN_THUNK)*，用于指定 C++实现中应采用哪种实现，以便根据 C++ 用户选择的错误建模方式，正确表示 Swift 函数，如下所示：

```
#ifdef __cpp_exceptions
 template<class T>
 using ThrowingResult = T;
 #define SWIFT_RETURN_THUNK(T, v) v
 #else
 template<class T>
 using ThrowingResult = Swift::Expected<T>;
 #define SWIFT_RETURN_THUNK(T, v) Swift::Expected<T>(v)
 #endif
```

代码清单7 – 用于替换函数类型及其返回方式，以适配异常使用情况的宏定义。

4 结果与示例

上述贡献表明，我们成功实现了主要目标。最后，通过两种方法，C++ 用户可以处理 Swift 函数抛出的错误。下面这个简单示例展示了一个可以抛出错误以避免意外行为的 Swift 函数。

```
@_expose(Cxx)
public enum DivByZero : Error {
    case divisorIsZero
    case bothAreZero

    // Function to print the case thrown
    public func getMessage() {
        print(self)
    }
}

@_expose(Cxx)
public func division(_ a: Int, _ b: Int) throws -> Float {
    if a == 0 && b == 0 {
        throw DivByZero.bothAreZero
    } else if b == 0 {
        throw DivByZero.divisorIsZero
    } else {
        return Float(a / b)
    }
}
```

List图 8 – 用户定义的 Error 以及可抛出错误的 Swift 函数示例 t.

此函数的客户端，即 C++ 用户，必须先编译定义此函数的程序，以创建桥接头文件，将 Swift 实现导出到 C++。以下命令可用于生成此头文件；该命令摘自 Swift 测试套件，并尽可能做了简化。

```
$ swift-frontend Example.swift -typecheck -module-name Functions \
    -enable-experimental-cxx-interop -emit-clang-header-path functions.h
```

清单 9 – 创建 Swift 程序桥接头文件的命令。

此命令以“Example.swift”文件作为输入，并输出“functi- ons.h”文件，其中包含 Swift 函数在“Functions”命名空间下的 C++ 表示形式。

```
inline Swift::ThrowingResult<float> division(swift::Int a, swift::Int b) {
  void* opaqueError = nullptr;
  void* _ctx = nullptr;
  auto returnValue = _impl::$s9Functions8divisionySfSi_SitKF(a,b,_ctx,&opaqueError);
  if (opaqueError != nullptr)
    #ifdef __cpp_exceptions
    throw (Swift::Error(opaqueError));
    #else
    return SWIFT_RETURN_THUNK(float, Swift::Error(opaqueError));
    #endif
  return SWIFT_RETURN_THUNK(float, returnValue);
  }
```

清单 10 – 在 C++ 中调用一个可能抛出错误、且可被捕获的 Swift 函数的示例。

上述由编译器生成并展示的 C++ 函数，可分为四个部分进行说明：

- **函数签名**：该函数的类型为“*ThrowingResult*<浮点数>”，而不是 Swift 中声明的“浮点数”类型。该类型的实现见清单 7，通过以不同类型两次声明该函数来避免代码重复，从而满足两种错误处理方式的要求。

- **Swift 函数调用**：这里有两个我们之前定义的额外参数；第一个保存上下文，第二个保存错误信息。它们必须与 Swift ABI 匹配，而最后一个参数则包含核心信息，用于指明 Swift 函数是否抛出了错误，以及具体是什么错误，或者是否按预期返回。

- **opaqueError 是 nullptr 吗?** ：此 if 条件用于检查函数执行期间，“*opaqueError*”参数是否被修改为保存 Swift 错误信息。此外，在该分支中，根据 C++ user 是否启用异常，我们有两种不同的实现方式；如果启用了异常，我们会抛出一个以“*opaqueError*”参数初始化的“*Swift::Error*”对象。如果未启用，我们会**返回**一个宏，该宏将被替换为：“*Swift::Expected*<浮点数> *(Swift::Error(opaqueError))*”。

- **返回表达式**：此处返回的宏会根据 C++ 异常的使用方式在编译时替换。它可以只返回

“*returnValue*”，也可以返回包含我们所期望的值而非错误的“*Swift::Expected<浮点数>(returnValue)*”，与它在此示例中的先前用法一样。

C++ 程序员通过将生成的 Swift 桥接头文件导入程序中，使用这种函数表示。下面展示了使用 C++ 异常调用此函数的两种方式：

```cpp
#include <cassert>
#include <cstdio>
#include "functions.h"

int main() {

    // This example catches an exception
    try {
        auto result = Functions::division(0,0);
        printf("result = %f\n", result);
    } catch (Swift::Error& e) {
        auto errorOpt = e.as<Functions::DivByZero>();
        assert(errorOpt.isSome());

        auto errorVal = errorOpt.get();
        assert(errorVal == Functions::DivByZero::bothAreZero);
        errorVal.getMessage();
    }

    // This example gets the correct value returned by the function
    try {
        float result = Functions::division(4,2);
        printf("result = %f\n", result);
    } catch (Swift::Error& e) {
        auto errorOpt = e.as<Functions::DivByZero>();
        assert(errorOpt.isSome());

        auto errorVal = errorOpt.get();
        errorVal.getMessage();
    }
    return 0;
}
```

清单11——在 C++ 中调用可能抛出错误的 Swift 函数并处理错误的示例。

在这个示例中，我们可以看到如何捕获“*Swift::Error*”，并将其动态转换为用户定义的错误。转换结果是一个 Optional类型，用于指示类型转换是否

成功，然后检查它是否抛出了我们预期的情况，接着使用“*getMessage()*”函数告知用户其名称。如果函数没有抛出错误，我们就会得到“浮点数”结果，并将其打印到标准输出。

以下命令展示了如何编译和测试此程序。首先，我们需要编译并生成其目标文件，将其与包含函数和错误定义的 Swift 文件一起编译，然后生成可执行程序。

```
$ clang++ -I ${SWIFT_BUILD_DIR}/swift-macosx-arm64/./lib/swift \
        -c Example.cpp -I /. -o example.o

$ swiftc -Xfrontend -enable-experimental-cxx-interop Example.swift -o example \
        -Xlinker example.o -module-name Functions -Xfrontend \
        -entry-point-function-name -Xfrontend swiftMain

$ ./example

```

清单12 – 用于构建、链接和执行调用 Swift函数且能够抛出异常的 C++ 程序的命令

在 C 中处理 Swift 错误的第二种方法++ 受到 C++提案“p0323r4 std::expected”（Botet，2017a）的启发。此方法需要一个新的 C++ 参数化类，名为“Swift::Expected<T>”，它可以初始化为持有 Swift::Error 或 T 类型的值，但不能同时持有两者。该类提供了辅助函数，以改善用户接口，并简化测试以及获取错误或值的操作。对于禁用了编译时异常的 C++ 用户，可抛出错误的 Swift 函数的 C++ 表示形式会返回此类的一个实例。

下面的示例使用相同的 Swift函数及其 C++ 表示形式，但采用了这种新的错误处理模型：

```cpp
#include <cassert>
#include <cstdio>
#include "functions.h"

int main() {
    auto errorResult = Functions::division(1,0);
    if (errorResult.has_value()) {
        printf("result = %f\n", errorResult.value());
    } else {
        auto optionalError = errorResult.error().as<Functions::DivByZero>();
        assert(optionalError.isSome());

        auto errorValue = optionalError.get();
        assert(errorValue == Functions::DivByZero::divisorIsZero);
        errorValue.getMessage();
    }

    auto goodResult = Functions::division(4,2);
    if (goodResult.has_value()) {
        printf("result = %f\n", goodResult.value());
    } else {
        auto optionalError = goodResult.error().as<Functions::DivByZero>();
        assert(optionalError.isSome());

        auto errorValue = optionalError.get();
        errorValue.getMessage();
    }
    return 0;
}
```

清单13 – 使用基于 std::expected 提案的 expected 类型处理 Swift 错误。

这个示例的主要区别在于，它没有使用“try-catch”块，而是检查函数的返回值是否包含值：如果包含，就打印该值；如果不包含，则进入另一种情况。后一种情况最值得关注，因为错误已作为返回值返回，可以通过“error()”函数访问，然后像之前一样进行处理。

下面的命令展示了如何编译和测试此程序。首先，我们需要使用“-fno-exceptions”标志编译并生成其目标文件，再将其与包含函数和错误定义的 Swift 文件一起编译，然后生成可执行程序。

```
$ clang++ -I ${SWIFT_BUILD_DIR}/swift-macosx-arm64/./lib/swift \
        -c Expected_Example.cpp -fno-exceptions -I /. -o expected_example.o

$ swiftc -Xfrontend -enable-experimental-cxx-interop Example.swift \
        -o expected_example -Xlinker expected_example.o -module-name Functions \
        -Xfrontend -entry-point-function-name -Xfrontend swiftMain

$ ./expected_example
```

清单 14 – 用于构建、链接和执行 C++ 程序的命令：该程序调用 Swift函数且无法抛出异常，而是使用 Swift::Expected<T>类。

参考文献

Albatross。《C 语言的历史》++。2021。最后访问于2022年5月6日。

Alexandrescu, A. *C 语言中的系统化错误处理++*。2012。最后访问于2022年12月19日。可在此处获取：<https://isocpp.org/blog/2012/12/systematic-error-handling-in-c-andrei-alexandrescu>。

Botet, J. B. V. *p0323r4 std::expected*。2017。最后访问于2022年5月5日。可在此处获取：<http://wg21.link/P0323r4>。

Botet, J. B. V. 用于表示期望对象的实用工具类。2017。最后访问于2022年12月19日。可在此处获取：<https://wg21.link/p0323r3>。

Cacciola, A. K. F. 一项提案：添加用于表示可选对象的实用工具类。2013。最后访问于2022年12月19日。可在此处获取：<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3793.html>。

cplusplus.com. 异常 - *C++* 教程。2021年。最后访问于2022年5月6日。网址：<https://www.cplusplus.com/doc/tutorial/exceptions/>。

cppreference.com. 动态异常规范（截至 *C++17*）。2022。最后访问于2022年7月20日。网址：<https://en.cppreference.com/w/cpp/language/except_规范>。

Debugger, T. L. *LLDB*调试器。2007年。最后访问于2022年12月18日。网址：<https://lldb.llvm.org>。

Swift。 *Swift*编程语言。2022。最后访问于2022年7月18日。网址：<https://github.com/apple/swift/tree/main/lib/PrintAsClang>。

Swift#59217。 *[互操作] [SwiftToCxx]* 新的抛出错误测试 *#59217*。2022。最后访问于2022年7月18日。网址： <https://github.com/apple/swift/pull/59217>。

Swift#59787. *[互操作] [SwiftToCxx]* 新建抛出错误测试 *#59217*。2022。最后访问于2022年7月18日。可访问：<https://github.com/apple/swift/pull/59787>。

Swift#60079. *[互操作] [SwiftToCxx]*在 *C* 中实现一种简单的异常抛出机制*++* 如果在 *Swift* 中抛出错误 *#60079*。2022。最后访问于 2022年7月18日。可访问：<https://github.com/apple/swift/pull/60079>。

Swift#60858。 *[SwiftToCxx]* 包含 *Swift* 错误的 *Cxx* 表示形式 *#60858*。2022。最后访问于 2022年12月18日。可访问：<https://github.com/apple/swift/pull/60858>。

Swift#61626。 *[互操作] [SwiftToCxx]* 为 *swift::Error* 引入动态类型转换 *#61626*。2022。最后访问于 2022年12月18日。可访问：<https://github.com/apple/swift/pull/61626>。

Swift#61823。 *[互操作][SwiftToCxx]* 引入 *swift::Expected #61823*。2022。最后访问于2022年12月19日。可在以下网址获取：<https://github.com/apple/swift/pull/61823>。

Swift#62197。 *[互操作] [SwiftToCxx] 修改 swift::Error 动态类型转换，使其返回 Swift::Optional #62197*。2022。最后访问于2022年12月18日。可在以下网址获取：<https://github.com/apple/swift/pull/62197>。

Swift.org。 基础知识 - *Swift* 编程语言（*Swift 5.6*）。2022。最后访问日期：2022年5月6日。可访问： <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID335>。

Swift.org。 *C++* 互操作工作组。2022。最后访问日期：2022年7月11日。可访问：<https://www.swift.org/cxx-interop-workgroup/>。

Swift.org。 *Swift.org* - 欢迎访问 *Swift.org*。2022。最后访问日期：2022年5月5日。可访问： <https://www.swift.org>。