OneFlow中的错误处理：Maybe

撰文 | 李新奇、twice、姚迟

1、C++ 中错误处理的困境

C++ 编程中错误情况处理的机制大概分为两种：

- 异常
- 函数返回错误码

异常

考虑以下的代码中get_cute_cat的实现：让img分别经过crop_to_cat、add_bow_tie、make_eyes_sparkle、make_smaller、add_rainbow处理，返回一张“可爱的猫”图片。

```
image get_cute_cat (const image& img) {
    return add_rainbow(
             make_smaller(
               make_eyes_sparkle(
                 add_bow_tie(
                   crop_to_cat(img)))));
}
```

它完全没有错误处理，因此当crop_to_cat的img里没有猫，或者其它意外情况时，程序的行为是不确定的。此时，可以加上异常处理：

```
image get_cute_cat (const image& img) {
  try {
    return add_rainbow(
             make_smaller(
               make_eyes_sparkle(
                 add_bow_tie(
                   crop_to_cat(img)))));
  catch (...) {
    return nullptr;
  }
}
```

但是，在 C++ 中的异常处理有它本身的弊端，比如效率的损失：在正常的流程上（即不触发异常），使用try...catch不会带来效率的损失；但是在触发异常的情况下，因为要做异常展开（unwind），会比较影响效率，尤其是当异常情况发生比较频繁时（比如以上例子中50%的图片都没有猫的话，程序效率会大大降低，当然这时候有读者会建议需要考虑把异常处理的逻辑提取到正常程序流程才更合理，这是后话了）。

除了效率问题，C++ 程序想要保证异常安全是一件非常难的事情，如果一个库声称自己是异常安全的，那这个库的作者应该花了数倍于功能实现的时间来保持“异常安全”这个称号。

在这方面Google C++ 风格指南列举了异常的详细优缺点，并给出了 Google 的决定：出于实践的考虑，Google 的大部分项目是不使用异常的（虽然如果一切从零开始可能会不一样），而是使用错误码检查或者 assertion。像 LLVM 这样的大型 C++ 项目，也设计了专门的Error Handling来避免使用异常。

函数返回错误码

使用函数返回错误码做错误处理的第一个问题是预期结果与表示意外情况的错误码容易混淆。

以简单的整数除法举例，除数是不能为零的：

```
int div(int x, int y) {
  if(y == 0){
    return 错误码;
  }
  return x / y;
}
```

以上的错误码无论选择是多少，都会“挤占”掉一个正常结果，比如，如果将-1作为错误码，那该如何表示调用div(5,-5)的结果呢？为了继续解决这类问题，就不得不使用传出参数或其它办法。

此外，对错误码的检查if对程序的正常逻辑有干扰，破坏了程序逻辑流畅性。以上文中的get_cute_cat为例，如果加上错误检查：

```
image get_cute_cat (const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
      return nullptr;
    }
    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
      return nullptr;
    }
    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
      return nullptr;
    }
    return add_rainbow(make_smaller(*with_sparkles));
}
```

程序的“正确路径”的逻辑，就被if(error)打碎，显得不够流畅。

TensorFlow 中使用的是基于错误码的错误处理，通过设计了通用类Status以及一组TF_CHECK_OK相关的宏来处理错误。

其核心实现如下：

```
string* TfCheckOpHelperOutOfLine(const ::tensorflow::Status& v,
                                 const char* msg) {
  string r("Non-OK-status: ");
  r += msg;
  r += " status: ";
  r += v.ToString();
  // Leaks string but this is only to be used in a fatal error message
  return new string(r);
}
```

```
inline tensorflow::string* TfCheckOpHelper(::tensorflow::Status v,
                                           const char* msg) {
  if (v.ok()) return nullptr;
  return TfCheckOpHelperOutOfLine(v, msg);
}

#define TF_DO_CHECK_OK(val, level)                                 \
  while (auto _result = ::tensorflow::TfCheckOpHelper(val, #val)) \
  LOG(level) << *(_result)

#define TF_CHECK_OK(val) TF_DO_CHECK_OK(val, FATAL)
#define TF_QCHECK_OK(val) TF_DO_CHECK_OK(val, QFATAL)
```

TensorFlow 的开发者，使用TF_CHECK_OK宏或者对调用函数的返回状态做检查，如果不ok的话，会触发FATAL或者QFATAL级别的事件，终止程序。

```
    Status s = allocate_tensor(tensor.dtype(), tensor.shape(), new_tensor.get(),
                               output_alloc_attr(index));
    TF_CHECK_OK(s);
```

这一定程度上减缓了if(error)打碎正确流程逻辑的问题。但是TF_CHECK_OK的限制也比较明显：

- 能够被 TF_CHECK_OK 做检查的方法，只能返回 Status 类
- TF_CHECK_OK 包裹后的方法调用，只能被当作语句使用，如 TF_CHECK_OK(Foo(…));，而不能作为表达式继续参与运算（如 const auto data = TF_CHECK_OK(Foo(…)) 是错误的）。这其实是上一个限制的衍生问题。

2、Haskell 中的优雅处理方式：Just Return

函数式语言 Haskell，其实早就提供了非常优雅一致的错误处理方式：Maybe 类型。看我们在 haskell 中如何实现一个带错误处理的除法（考虑熟悉命令式编程的读者数目更多，这里的代码写法尽量接近命令式风格）：

```
safediv x y =
  if y /= 0          -- 如果 y 不等于 0
  then Just(div x y) -- 直接返回 x/y 的结果
  else Nothing       -- 返回 Nothiing
```

这样的函数的返回类型是Maybe类型：

```
ghci> :t safediv
safediv :: Integral a ⇒ a → a → Maybe a
```

Maybe类型的特点是，如果是期待的结果，那么Maybe里面的值为被Just包裹的真实值（haskell 中的 "Just" 有 "just return it" 的含义）。

```
ghci> safediv 10 2
Just 5
```

而如果是错误情况，则Maybe类型里的值是Nothing：

```
ghci> safediv 10 0
Nothing
```

被Just包裹的真实值是无法直接运算的，可以调用fromJust取出（其实>>=更重要，限于篇幅我们就不在本文介绍）：

```
ghci> fromJust (safediv 10 2)
5
```

如果值为Nothing，则fromJust时会触发异常：

```
ghci> fromJust(safediv 10 0)
*** Exception: Maybe.fromJust: Nothing
CallStack (from HasCallStack):
  error, called at libraries\base\Data\Maybe.hs:148:21 in base:Data.Maybe
  ...
```

Maybe的显著好处是，在做了错误处理的同时，函数调用的结果可以继续**当作表达式**使用：

```
a = 1 + fromJust(safediv 10 2)
```

再上升一个层面说，Maybe其实是 Haskell 中**Monad**的代表之一，所谓 Monad，可以看作为不同的数据类型和控制结构，都提供一致的接口。

更抽象地说，Maybe是一种可以同时装载正常数据和错误信息的 ADT，即Union<T, Error>。

由于 HaskellMaybe的成功，其它许多语言中，都提供了类似的数据类型：

但是 C++ 中的std::optional在 C++17 中才引入，并且让std::optional支持monadic operations的提案要在 C++23 才引入。OneFlow 用自己的方式实现了Maybe。

3、OneFlow 中的 Maybe

了解了上文 Haskell 中 Maybe 的用法，就很容易理解 OneFlow 中Maybe类如何配合JUST宏使用。我们在此先展示如何用 OneFlow 使用它们，再介绍其背后的实现原理。

使用 OneFlow Maybe

同样用整数除法的例子，用 OneFlowMaybe实现safediv代码如下：

```
Maybe<int> safediv(int x, int y){
    CHECK_NE_OR_RETURN(y, 0) << "y cannot be zero";
    return x / y;
}
```

其中用到的CHECK_NE_OR_RETURN及其它宏（如等于的检查CHECK_EQ_OR_RETURN、不为空的检查 CHECK_NOTNULL_OR_RETURN等），都定义在maybe.h中，用于表达式检查。

凡是返回类型为Maybe类型的函数、方法，都需要使用宏JUST包裹，整个JUST的结果，**是表达式**，而不是语句，因此可以直接参与后续的表达式求值：

```
int a = 1 + JUST(safediv(10, 2));
```

实现原理

OneFlow 中的Maybe<T>类定义在maybe.h文件中，如上文所说，它其实是Union<T, Error>。在maybe.h文件中其实是为各类数据类型做了模板特化，都实现了相同的接口，使得Maybe<T>容器可以装载它们（以及错误），

这些类型包括：

- Maybe<void>：它对应了原 void 返回类型，等同于 TensorFlow 的 Status 类。
- Maybe<ClassType>：用户自定义数据类型（类/结构体）。当数据从 Maybe 中取出时，类型为 shared_ptr。
- Maybe<ScalarType>：C++ 中的标量数据类型，当数据从 Maybe 中取出时，类型为 ScalarType 本身。
- Maybe<ReferenceType>：C++ 中的引用数据类型，当数据从 Maybe 中取出时，类型为引用本身。

Maybe<T>提供的主要接口有：

- IsOk()：Maybe 中是否有正常数据
- error()：获取错误信息
- Data_YouAreNotAllowedToCallThisFuncOutsideThisFile()：获取正常流程的数据，之所以起这么复杂的名字，就是要疏远用户，防止用户直接调用

从Maybe取数据的JUST其实是一个宏，它定义在just.h文件中：

```
#define JUST(...)                                                                  \
  ::oneflow::private_details::RemoveRValConst(({                                   \
    auto&& value_to_check_ = __JustStackCheckWrapper__(__VA_ARGS__);               \
    if (!::oneflow::private_details::JustIsOk(value_to_check_)) {                  \
      return ::oneflow::private_details::JustErrorAddStackFrame(                   \
          ::oneflow::private_details::JustGetError(value_to_check_), __FILE__, __LINE__, \
          __FUNCTION__, OF_PP_STRINGIZE(__VA_ARGS__));                             \
    }                                                                              \
    std::forward<decltype(value_to_check_)>(value_to_check_);                      \
  })).Data_YouAreNotAllowedToCallThisFuncOutsideThisFile()
```

抛开以上的为了类型安全的操作（RemoveRValConst），预留的中间层（__JustStackCheckWrapper__）和为了兼容其它数据类型做的重载函数（JustIsOk），JUST宏的核心逻辑其实如下：

```
#define JUST(...)                                                  \
  ({                                                               \
    auto&& value_to_check_ = __VA_ARGS__;                          \
    if (!value_to_check_.IsOk()) {                                 \
      auto* stack_frame = value_to_check_.error()->add_stack_frame(); \
      stack_frame->set_file(__FILE__);                             \
      stack_frame->set_line(__LINE__);                             \
      stack_frame->set_function(__FUNCTION__);                     \
      stack_frame->set_error_msg(OF_PP_STRINGIZE(__VA_ARGS__));    \
      return value_to_check_.error();                              \
    }                                                              \
    value_to_check_;                                               \
  }).Data_YouAreNotAllowedToCallThisFuncOutsideThisFile()
```

可以看到，被 JUST 包括的函数调用，其返回结果（Maybe<T> 类型），会先存为 value_to_check，然后对其是否发生错误做检查 if(!value_to_check_IsOK())，如果发生了错误，则记录出错栈的信息，并直接返回错误。

如果一切正常，则利用statement expression语法，让value_to_check作为整个语句块（从({到})）的值，并且调用Data_YouAreNotAllowedToCallThisFuncOutsideThisFile从中取出正常数据。

说句题外话，此处用到的 statement expression 是 GNU 的扩展语法，GCC 和 Clang 均支持，不过 MSVC 还不支持。在不支持 statement expression 的编译环境中，JUST应该可以退化成使用异常实现，感兴趣的读者欢迎给OneFlow-Inc/oneflow仓库提 issue 或 PR。

以上的做法，保证了把JUST包裹后，既可以当作表达式使用，又可以在发生错误时使当前函数提前返回。比如考虑以下代码：

```
Maybe<float> rsqrt(float x1) {
  float x2 = JUST(sqrt(x1));
  float x3 = JUST(div(1, x2));
  return x3;
}
```

如果float x2 = JUST(sqrt(x1));出错，rsqrt函数的执行会直接终止，不会再执行sqrt后面的div。OneFlow 的Maybe<T>与JUST配合使用，除了在错误处理的同时可以直接当作表达式这个基本好处外，还有其一些其它的优势。

JUST 链构建的错误栈

在 OneFlow 中使用Maybe<T>作为函数返回值时，我们做出了以下约定：

1. 所有函数不得以 Maybe 作为输入参数；
2. 对于所有以 Maybe 为返回值的函数，其调用都必须被 JUST （或者 CHECK_JUST 等 OneFlow 错误检查机制提供的宏）包裹。

这样，一旦发生错误，最内层JUST函数的错误返回结果，会逐层向上传播，构建一个错误栈。比如尝试在 Python 端错误地使用flow.gather：

```
import oneflow as flow
input = flow.randn(2,2)
index = flow.randn(2,2) # 类型错误
flow.gather(input, 0, index)
```

得到的报错信息从触发错误的 C++ 代码位置，一直记录到 Python 层，报错信息友好。

```
~/oneflow/python/oneflow/nn/modules/gather.py in gather_op(input, dim, index, sparse_grad)
     64                 index.shape[i] <= input.shape[i]
     65             ), "index.size(d) <= input.size(d) for all dimensions d != dim"
---> 66     return flow._C.dim_gather(input, index, dim=dim)
     67
     68

CheckFailedException:
  File "oneflow/oneflow/core/framework/op_interpreter/op_interpreter_util.cpp", line 139, in Dispatch<on
    Dispatch<TensorTuple>(op_expr, inputs, ctx)
  File "oneflow/oneflow/core/framework/op_interpreter/op_interpreter_util.cpp", line 131, in Dispatch<on
    Dispatch(op_expr, inputs, outputs.get(), ctx)
  File "oneflow/oneflow/core/framework/op_interpreter/op_interpreter.cpp", line 94, in Apply
    internal_->Apply(op_expr, inputs, outputs, ctx)
  File "oneflow/oneflow/core/framework/op_interpreter/eager_mirrored_op_interpreter.cpp", line 139, in N
    user_op_expr.InferPhysicalShapeAndDType( attrs, device_tag ... TensorMeta* { return output_tensor_me
  File "oneflow/oneflow/core/framework/op_expr.cpp", line 436, in InferPhysicalShapeAndDType
    dtype_infer_fn_(&infer_ctx)
  File "oneflow/oneflow/user/ops/dim_gather_op.cpp", line 50, in operator()
    Check failed: IsIndexDataType(index.data_type())
```

不过有读者可能会考虑到：万一有开发者实现了返回Maybe<T>的函数，但是在调用时忘记使用JUST包裹，那么不就导致错误栈断裂吗？

实际上，如果只是“告诫”开发者使用JUST，有点类似于告诫使用异常的程序员要记得 RAII，听起来很合理，执行起来却容易发生遗漏。

不过，OneFlow 围绕Maybe<T>和JUST的约定，已经基于 LLVM 开发了一套静态分析工具（https://github.com/llvm/llvm-project/compare/main…Oneflow-Inc:maybe ），确保开发者的代码是遵循JUST有关约定的。关于其中的实现细节，我们可以在后续的文章中做详细介绍。

4、总结

开源分布式深度学习框架，汲取了函数式编程语言 Haskell 中的思想，构建了Maybe<T>这种 ADT，并围绕它构建了一系列的基础设施，借助这套基础设置，开发者可以不花费额外精力于错误处理，就写出高质量、高容错的代码。

OneFlow 还基于 LLVM 构建了静态分析工具，用于确保开发者按照约定正确使用 OneFlowMaybe<T>。

参考资料

其他人都在看