22 | 处理数据类型变化和错误：optional、variant、expected和Herbception

2020-01-15 吴咏炜

现代C++实战30讲

进入课程 >

讲述：吴咏炜
时长 15:53　大小 14.55M

你好，我是吴咏炜。

我们之前已经讨论了异常是推荐的 C++ 错误处理方式。不过，C++ 里有另外一些结构也很适合进行错误处理，今天我们就来讨论一下。

optional

在面向对象（引用语义）的语言里，我们有时候会使用空值 null 表示没有找到需要的对象。也有人推荐使用一个特殊的空对象，来避免空值带来的一些问题 [1]。可不管是空值，还是空对象，对于一个返回普通对象（值语义）的 C++ 函数都是不适用的——空值和空对

象只能用在返回引用 / 指针的场合，一般情况下需要堆内存分配，在 C++ 里会引致额外的开销。

C++17 引入的 `optional` 模板 [2] 可以（部分）解决这个问题。语义上来说，`optional` 代表一个“也许有效”“可选”的对象。语法上来说，一个 `optional` 对象有点像一个指针，但它所管理的对象是直接放在 `optional` 里的，没有额外的内存分配。

构造一个 `optional<T>` 对象有以下几种方法：

1. 不传递任何参数，或者使用特殊参数 `std::nullopt`（可以和 `nullptr` 类比），可以构造一个“空”的 `optional` 对象，里面不包含有效值。
2. 第一个参数是 `std::in_place`，后面跟构造 `T` 所需的参数，可以在 `optional` 对象上直接构造出 `T` 的有效值。
3. 如果 `T` 类型支持拷贝构造或者移动构造的话，那在构造 `optional<T>` 时也可以传递一个 `T` 的左值或右值来将 `T` 对象拷贝或移动到 `optional` 中。

对于上面的第 1 种情况，`optional` 对象里是没有值的，在布尔值上下文里，会得到 `false`（类似于空指针的行为）。对于上面的第 2、3 两种情况，`optional` 对象里是有值的，在布尔值上下文里，会得到 `true`（类似于有效指针的行为）。类似的，在 `optional` 对象有值的情况下，你可以用 `*` 和 `->` 运算符去解引用（没值的情况下，结果是未定义行为）。

虽然 `optional` 是 C++17 才标准化的，但实际上这个用法更早就通行了。因为 `optional` 的实现不算复杂，有些库里就自己实现了一个版本。比如 cpptoml [3] 就给出了下面这样的示例（进行了翻译和重排版），用法跟标准的 `optional` 完全吻合：

复制代码

```
auto val = config->
  get_as<int64_t>("my-int");
// val  是  cpptoml::option<int64_t>

if (val) {
  // *val  是  "my-int"  键下的整数值
} else {
  // "my-int"  不存在或不是整数
```

```
9 }
```

cpptoml 里只是个缩微版的 `optional`，实现只有几十行，也不支持我们上面说的所有构造方式。标准库的 `optional` 为了方便程序员使用，除了我目前描述的功能，还支持下面的操作：

安全的析构行为

显式的 `has_value` 成员函数，判断 `optional` 是否有值

`value` 成员函数，行为类似于 `*`，但在 `optional` 对象无值时会抛出异常 `std::bad_optional_access`

`value_or` 成员函数，在 `optional` 对象无值时返回传入的参数

`swap` 成员函数，和另外一个 `optional` 对象进行交换

`reset` 成员函数，清除 `optional` 对象包含的值

`emplace` 成员函数，在 `optional` 对象上构造一个新的值（不管成功与否，原值会被丢弃）

`make_optional` 全局函数，产生一个 `optional` 对象（类似 `make_pair`、`make_unique` 等）

全局比较操作

等等

如果我们认为无值就是数据无效，应当跳过剩下的处理，我们可以写出下面这样的高阶函数：

复制代码

```
1 template <typename T>
2 constexpr bool has_value(
3   const optional<T>& x) noexcept
4 {
5     return x.has_value();
6 }
7
8 template <typename T,
9           typename... Args>
```

```
constexpr bool has_value(
  const optional<T>& first,
  const optional<
    Args>&... other) noexcept
{
  return first.has_value() &&
         has_value(other...);
}

template <typename F>
auto lift_optional(F&& f)
{
  return [f = forward<F>(f)](
           auto&&... args) {
    typedef decay_t<decltype(f(
      forward<decltype(args)>(args)
        .value())...)>
      result_type;
    if (has_value(args...)) {
      return optional<result_type>(
        f(forward<decltype(args)>(
            args)
            .value()...));
    } else {
      return optional<
        result_type>();
    }
  };
}
```

has_value 比较简单，它可以有一个或多个 optional 参数，并在所有参数都有值时返回真，否则返回假。lift_optional 稍复杂些，它接受一个函数，返回另外一个函数。在返回的函数里，参数是一个或多个 optional 类型，result_type 是用参数的值（value()）去调用原先函数时的返回值类型，最后返回的则是 result_type 的 optional 封装。函数内部会检查所有的参数是否都有值（通过调用 has_value）：有值时会去拿参数的值去调用原先的函数，否则返回一个空的 optional 对象。

这个函数能把一个原本要求参数全部有效的函数抬升（lift）成一个接受和返回 optional 参数的函数，并且，只在参数全部有效时去调用原来的函数。这是一种非常函数式的编程方式。使用上面函数的示例代码如下：

复制代码

```
#include <iostream>
```

```
#include <functional>
#include <optional>
#include <type_traits>
#include <utility>

using namespace std;

//  需包含  lift_optional  的定义

constexpr int increase(int n)
{
    return n + 1;
}

//  标准库没有提供  optional  的输出
ostream&
operator<<(ostream& os,
           optional<int>(x))
{
  if (x) {
    os << '(' << *x << ')';
  } else {
    os << "(Nothing)";
  }
  return os;
}

int main()
{
  auto inc_opt =
    lift_optional(increase);
  auto plus_opt =
    lift_optional(plus<int>());
  cout << inc_opt(optional<int>())
       << endl;
  cout << inc_opt(make_optional(41))
       << endl;
  cout << plus_opt(
            make_optional(41),
            optional<int>())
       << endl;
  cout << plus_opt(
            make_optional(41),
            make_optional(1))
       << endl;
}
```

输出结果是：

```
(Nothing)
(42)
(Nothing)
(42)
```

variant

`optional` 是一个非常简单而又好用的模板，很多情况下，使用它就足够解决问题了。在某种意义上，可以把它看作是允许有两种数值的对象：要么是你想放进去的对象，要么是 `nullopt`（再次提醒，联想 `nullptr`）。如果我们希望除了我们想放进去的对象，还可以是 `nullopt` 之外的对象怎么办呢（比如，某种出错的状态）？又比如，如果我希望有三种或更多不同的类型呢？这种情况下，`variant` [4] 可能就是一个合适的解决方案。

在没有 `variant` 类型之前，你要达到类似的目的，恐怕会使用一种叫做带标签的联合（tagged union）的数据结构。比如，下面就是一个可能的数据结构定义：

复制代码

```
struct FloatIntChar {
  enum {
    Float,
    Int,
    Char
  } type;
  union {
    float float_value;
    int int_value;
    char char_value;
  };
};
```

这个数据结构的最大问题，就是它实际上有很多复杂情况需要特殊处理。对于我们上面例子里的 POD 类型，这么写就可以了（但我们仍需小心保证我们设置的 `type` 和实际使用的类型一致）。如果我们把其中一个类型换成非 POD 类型，就会有复杂问题出现。比如，下面的代码是不能工作的：

复制代码

```
struct StringIntChar {
  enum {
```

```
    String,
    Int,
    Char
  } type;
  union {
    string string_value;
    int int_value;
    char char_value;
  };
};
```

编译器会很合理地看到在 union 里使用 `string` 类型会带来构造和析构上的问题，所以会拒绝工作。要让这个代码工作，我们得手工加上析构函数，并且，在析构函数里得小心地判断存储的是什么数值，来决定是否应该析构（否则，默认不调用任何 union 里的析构函数，从而可能导致资源泄漏）：

复制代码

```
  ~StringIntChar()
  {
    if (type == String) {
      string_value.~string();
    }
  }
```

这样，我们才能安全地使用它（还是很麻烦）：

复制代码

```
StringIntChar obj{
  .type = StringIntChar::String,
  .string_value = "Hello world"};
cout << obj.string_value << endl;
```

这里用到了按成员初始化的语法，把类型设置成了字符串，同时设置了字符串的值。不用说，这是件麻烦、容易出错的事情。同时，细查之后我发现，这个语法虽然在 C99 里有，但在 C++ 里要在 C++20 才会被标准化，因此实际是有兼容性问题的——老版本的 MSVC，或最新版本的 MSVC 在没有开启 C++20 支持时，就不支持这个语法。

所以，目前的主流建议是，应该避免使用“裸” union 了。替换方式，就是这一节要说的 variant。上面的例子，如果用 variant 的话，会非常的干净利落：

```
variant<string, int, char> obj{
  "Hello world"};
cout << get<string>(obj) << endl;
```

可以注意到我上面构造时使用的是 const char*，但构造函数仍然能够正确地选择 string 类型，这是因为标准要求实现在没有一个完全匹配的类型的情况下，会选择成员类型中能够以传入的类型来构造的那个类型进行初始化（有且只有一个时）。string 类存在形式为 string(const char*) 的构造函数（不精确地说），所以上面的构造能够正确进行。

跟 tuple 相似，variant 上可以使用 get 函数模板，其模板参数可以是代表序号的数字，也可以是类型。如果编译时可以确定序号或类型不合法，我们在编译时就会出错。如果序号或类型合法，但运行时发现 variant 里存储的并不是该类对象，我们则会得到一个异常 bad_variant_access。

variant 上还有一个重要的成员函数是 index，通过它我们能获得当前的数值的序号。就我们上面的例子而言，obj.index() 即为 1。正常情况下，variant 里总有一个有效的数值（缺省为第一个类型的默认构造结果），但如果 emplace 等修改操作中发生了异常，variant 里也可能没有任何有效数值，此时 index() 将会得到 variant_npos。

从基本概念来讲，variant 就是一个安全的 union，相当简单，我就不多做其他介绍了。你可以自己看文档来了解进一步的信息。其中比较有趣的一个非成员函数是 visit [5]，文档里展示了一个非常简洁的、可根据当前包含的变量类型进行函数分发的方法。

平台细节：在老于 Mojave 的 macOS 上编译含有 optional 或 variant 的代码，需要在文件开头加上：

```
#if defined(__clang__) && defined(__APPLE__)
#include <__config>
```

```
#undef _LIBCPP_AVAILABILITY_BAD_OPTIONAL_ACCESS
#undef _LIBCPP_AVAILABILITY_BAD_VARIANT_ACCESS
#define _LIBCPP_AVAILABILITY_BAD_OPTIONAL_ACCESS
#define _LIBCPP_AVAILABILITY_BAD_VARIANT_ACCESS
#endif
```

原因是苹果在头文件里把 `optional` 和 `variant` 在早期版本的 macOS 上禁掉了，而上面的代码去掉了这几个宏里对使用 `bad_optional_access` 和 `bad_variant_access` 的平台限制。我真看不出使用这两个头文件跟 macOS 的版本有啥关系。😒

expected

和前面介绍的两个模板不同，`expected` 不是 C++ 标准里的类型。但概念上这三者有相关性，因此我们也放在一起讲一下。

我前面已经提到，`optional` 可以作为一种代替异常的方式：在原本该抛异常的地方，我们可以改而返回一个空的 `optional` 对象。当然，此时我们就只知道没有返回一个合法的对象，而不知道为什么没有返回合法对象了。我们可以考虑改用一个 `variant`，但我们此时需要给错误类型一个独特的类型才行，因为这是 `variant` 模板的要求。比如：

复制代码

```
enum class error_code {
  success,
  operation_failure,
  object_not_found,
  …
};

variant<Obj, error_code>
  get_object(…);
```

这当然是一种可行的错误处理方式：我们可以判断返回值的 `index()`，来决定是否发生了错误。但这种方式不那么直截了当，也要求实现对允许的错误类型作出规定。Andrei Alexandrescu 在 2012 年首先提出的 Expected 模板 [6]，提供了另外一种错误处理方式。他的方法的要点在于，把完整的异常信息放在返回值，并在必要的时候，可以“重放”出来，或者手工检查是不是某种类型的异常。

他的概念并没有被广泛推广，最主要的原因可能是性能。异常最被人诟病的地方是性能，而他的方式对性能完全没有帮助。不过，后面的类似模板都汲取了他的部分思想，至少会用一种显式的方式来明确说明当前是异常情况还是正常情况。在目前的 expected 的标准提案 [7] 里，用法有点是 `optional` 和 `variant` 的某种混合：模板的声明形式像 `variant`，使用正常返回值像 `optional`。

下面的代码展示了一个 expected 实现 [8] 的基本用法。

复制代码

```
#include <cstdint>
#include <iostream>
#include <string>
#include <tl/expected.hpp>

using namespace std;
using tl::expected;
using tl::unexpected;

//  返回  expected  的安全除法
expected<int, string>
safe_divide(int i, int j)
{
  if (j == 0)
    return unexpected(
      "divide by zero"s);
  if (i == INT_MIN && j == -1)
    return unexpected(
      "integer divide overflows"s);
  if (i % j != 0)
    return unexpected(
      "not integer division"s);
  else
    return i / j;
}

//  一个测试函数
expected<int, string>
caller(int i, int j, int k)
{
  auto q = safe_divide(j, k);
  if (q)
    return i + *q;
  else
    return q;
}

//  支持  expected  的输出函数
```

```
39 template <typename T, typename E>
40 ostream& operator<<(
41   ostream& os,
42   const expected<T, E>& exp)
43 {
44   if (exp) {
45     os << exp.value();
46   } else {
47     os << "unexpected: "
48        << exp.error();
49   }
50   return os;
51 }
52
53 //  调试使用的检查宏
54 #define CHECK(expr)                 \
55   {                                 \
56     auto result = (expr);           \
57     cout << result;                 \
58     if (result ==                   \
59         unexpected(                 \
60           "divide by zero"s)) {     \
61       cout                          \
62         << ": Are you serious?";    \
63     } else if (result == 42) {      \
64       cout << ": Ha, I got you!";   \
65     }                               \
66     cout << endl;                   \
67   }
68
69 int main()
70 {
71   CHECK(caller(2, 1, 0));
72   CHECK(caller(37, 20, 7));
73   CHECK(caller(39, 21, 7));
74 }
```

输出是:

```
unexpected: divide by zero: Are you serious?
unexpected: not integer division
42: Ha, I got you!
```

一个 `expected<T, E>` 差不多可以看作是 `T` 和 `unexpected<E>` 的 `variant`。在学过上面的 `variant` 之后，我们应该很容易看明白上面的程序了。下面是几个需要注意一下的

地方：

> 如果一个函数要正常返回数据，代码无需任何特殊写法；如果它要表示出现了异常，则可以返回一个 `unexpected` 对象。
>
> 这个返回值可以用来和一个正常值或 unexpected 对象比较，可以在布尔值上下文里检查是否有正常值，也可以用 `*` 运算符来取得其中的正常值——与 `optional` 类似，在没有正常值的情况下使用 `*` 是未定义行为。
>
> 可以用 `value` 成员函数来取得其中的正常值，或使用 `error` 成员函数来取得其中的错误值——与 `variant` 类似，在 `expected` 中没有对应的值时产生异常 `bad_expected_access`。
>
> 返回错误跟抛出异常比较相似，但检查是否发生错误的代码还是要比异常处理啰嗦。

Herbception

上面的用法初看还行，但真正用起来，你会发现仍然没有使用异常方便。这只是为了解决异常在错误处理性能问题上的无奈之举。大部分试图替换 C++ 异常的方法都是牺牲编程方便性，来换取性能。只有 Herb Sutter 提出了一个基本兼容当前 C++ 异常处理方式的错误处理方式 [9]，被戏称为 Herbception。

上面使用 expected 的示例代码，如果改用 Herbception 的话，可以大致如下改造（示意，尚无法编译）：

复制代码

```
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;
}


```

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

#define CHECK(expr)                \
  try {                            \
    int result = (expr);           \
    cout << result;                \
    if (result == 42) {            \
      cout << ": Ha, I got you!"; \
    }                              \
  }                                \
  catch (error e) {                \
    if (e == arithmetic_errc::     \
                divide_by_zero) {  \
      cout                         \
        << "Are you serious? ";    \
    }                              \
    cout << "An error occurred";   \
  }                                \
  cout << endl

int main()
{
  CHECK(caller(2, 1, 0));
  CHECK(caller(37, 20, 7));
  CHECK(caller(39, 21, 7));
}
```

我们可以看到，上面的代码和普通使用异常的代码非常相似，区别有以下几点：

函数需要使用 `throws`（注意不是 `throw`）进行声明。

抛出异常的语法和一般异常语法相同，但抛出的是一个 `std::error` 值 [10]。

捕捉异常时不需要使用引用（因为 `std::error` 是个“小”对象），且使用一般的比较操作来检查异常“类型”，不再使用开销大的 RTTI。

虽然语法上基本是使用异常的样子，但 Herb 的方案却没有异常的不确定开销，性能和使用 expected 相仿。他牺牲了异常类型的丰富，但从实际编程经验来看，越是体现出异常优越性的地方——异常处理点和异常发生点距离较远的时候——越不需要异常有丰富的类型。因此，总体上看，这是一个非常吸引人的方案。不过，由于提案时间较晚，争议颇多，这个方案要进入标准至少要 C++23 了。我们目前稍稍了解一下就行。

更多技术细节，请查看参考资料。

内容小结

本讲我们讨论了两个 C++ 标准库的模板 `optional` 和 `variant`，然后讨论了两个标准提案 expected 和 Herbception。这些结构都可以使用在错误处理过程中——前三者当前可用，但和异常相比有不同的取舍；Herbception 当前还不可用，但有希望在错误处理上达到最佳的权衡点。

课后思考

错误处理是一个非常复杂的问题，在 C++ 诞生之后这么多年仍然没有该如何处理的定论。如何对易用性和性能进行取舍，一直是一个有矛盾的老大难问题。你的实际项目中是如何选择的？你觉得应该如何选择？

欢迎留言和我分享你的看法。

参考资料

[1] Wikipedia, “Null object pattern”.
https://en.wikipedia.org/wiki/Null_object_pattern

[2] cppreference.com, “std::optional”.
https://en.cppreference.com/w/cpp/utility/optional

[2a] cppreference.com, “std::optional”.
https://zh.cppreference.com/w/cpp/utility/optional

[3] Chase Geigle, cpptoml. https://github.com/skystrife/cpptoml

[4] cppreference.com, “std::optional”.
https://en.cppreference.com/w/cpp/utility/variant

[4a] cppreference.com, “std::optional”.
https://zh.cppreference.com/w/cpp/utility/variant

[5] cppreference.com, “std::visit”. https://en.cppreference.com/w/cpp/utility/variant/visit

[5a] cppreference.com, “std::visit”. https://zh.cppreference.com/w/cpp/utility/variant/visit

[6] Andrei Alexandrescu, “Systematic error handling in C++”. https://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C

[7] Vicente J. Botet Escribá and JF Bastien, “Utility class to represent expected object”. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0323r3.pdf

[8] Simon Brand, expected. https://github.com/TartanLlama/expected

[9] Herb Sutter, “P0709R0: Zero-overhead deterministic exceptions: Throwing values”. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0709r0.pdf

[10] Niall Douglas, “P1028R0: SG14 `status_code` and standard `error object` for P0709 Zero-overhead deterministic exceptions”. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1028r0.pdf

上一篇　21 | 工具漫谈：编译、格式化、代码检查、排错各显身手

下一篇　23 | 数字计算：介绍线性代数和数值计算库

精选留言 (2)

写留言

廖熊猫
2020-01-15

有的语言里面没有try catch，统一使用类似optional的结局方案，比如Rust里面的Err，Haskell中对应的应该是Either类型，这些都是处理可以恢复的错误，不可恢复的直接就让程序崩了。
lift_optional让我想起来被Haskell支配的恐惧 (Just (+)) <*> Just 41 <*> Just 1
不知道老师后面会不会讲到monad😂

展开

作者回复: optional 对应的是 Maybe 吧。expected 比较像 Either。

不会讲 monad。这个即使专门讲函数式编程也要比较后面呢。

tt

2020-01-15

老师，听了您的课后，觉得现在C++标准提案有很多都是利用C++的语义和语法来写提升编程便利性的模板，是这样么？

还有，一直不知道C++的异常是怎么实现的，还有这里说的异常处理的性能问题，有推荐的比较好阅读的参考文献么？

展开

作者回复: 是的。提高开发的友好程度，尤其是对新手的友好程度，一直是 C++ 委员会的目标。你应该发现虽然语言越来越复杂，但很多东西缺越来越好写了。

异常的实现非常复杂。想了解的话需要花点力气。不过我倒是查到有篇还不错的中文文章，推荐一看：

http://baiy.cn/doc/cpp/inside_exception.htm