expectedによる
エラーハンドリング

高橋 晶(Akira Takahashi)
faithandbrave@gmail.com
2015/12/05(土) Boost.勉強会 #19 東京

自己紹介

- 高橋 晶(Akira Takahashi)
- Boost.勉強会 東京の主催者
- boostjp、cpprefjpサイトのコアメンバ
- 株式会社ロングゲート所属
- システム開発、ゲーム開発、教育のお仕事などを
  しています。
- 書籍：『C++テンプレートテクニック』『C++ポケット
  リファレンス』『プログラミングの魔導書シリーズ』など

expectedとは

- 正常値とエラー値のどちらかが入る、汎用的な型
- 作者 Vicente J. Botet Escriba
- アイディア元は、C++ and Beyond 2012での
  Andrei Alexandrescuのプレゼンテーション
  「Systematic Error Handling in C++」
- リポジトリはここ：https://github.com/ptal/expected
- 標準への提案文書もGitHubにある。仕様の議論は主に、
  std-proposalsメーリングリストで行われている。

話すこと

- これまでのエラーハンドリングと問題点
- expectedの概要
- expectedの仕様

お断り

- expectedは現在策定中の機能であり、正式に導入されるまでに、仕様が変更される可能性があります。
- この発表では、変更される可能性の高そうなところは、説明を薄くします。
- たとえば、「エラー型の扱い」「例外の扱い」といったところは、変更される可能性が高いと考えています。

Chapter 1

これまでのエラーハンドリングと問題点

expectedの解説に入るまえに

- これまでのエラーハンドリングを振り返っていきます

boolによるエラーハンドリング 1/2

- bool値によって、処理が成功したか失敗したかを
  判定してきました

```
bool validateUserInput(string input)
{
  ...
  return true; // 成功したらtrue、失敗したらfalseを返す
}

if (validateUserInput("hoge"))
  // 正しいユーザー入力
else
  // 不正なユーザー入力
```

boolによるエラーハンドリング 2/2

- 成功時に戻り値を付加したい場合は、
  戻り値の型を`pair<bool, T>`としたり、

```
template <class T>
pair<bool, T> f() {
  if (…成功…) return make_pair(true, …T型の戻り値…);
  return make_pair(false, T());
}
```

- 非const参照のパラメータで返したりしていました。

```
bool f(T& result)
{
  if (…成功…) result = …T型の戻り値…;
}
```

有効値と無効値によるハンドリング

- 成功したときに有効な値を返し、失敗したときに無効な値を返す、という方法も、昔から行われていました。

```
int f() {
  return -1; // 成功したら0以上、失敗したら0未満の値を返す
}
```

```
T* f() {
  return nullptr; // 成功したら有効なポインタ、
                  // 失敗したらヌルポインタを返す
}
```

```
string f() {
  return string(); // 成功したら非空文字列、
                   // 失敗したら空文字列を返す
}
```

optionalによるエラーハンドリング

- 有効値と無効値の統一的な表現として、
  `boost::optional<T>`という型が導入されました。

- 整数、ポインタ、文字列といったもので、システムごと、または関数ごとに無効値が異なる仕様になるのを防いでくれます。

```
optional<int> f() {
  if (…成功…)
    return 42;   // 成功したらT型の値を返す
  else
    return none; // 失敗したらnoneという特殊な値を返す
}
```

error_codeクラス

- OSのエラーコードを扱うことを主目的として、error_codeというクラスが導入された(Boost, C++11)

- こちらはoptionalとは逆に、エラー値か成功かのどちらかを状態として持つ

```
error_code error = make_error_code(errc::invalid_argument);
if (error) // エラー
  cout << error.message() << endl;
else
  cout << "成功" << endl;
```

futureクラス

- 非同期操作の結果を読むためのクラスfutureでは、
  成功値か例外のどちらかを状態として持つ。

```
int calc() {
  if (…成功…) return 42;
  throw runtime_error("failed");
}

future<int> f = async(calc);
try {
  int result = f.get(); // 成功したら値が返され、
                        // 失敗したら例外が送出される
}
catch (runtime_error& e) { … }
```

これまでのエラーハンドリングの問題点

- optionalやerror_codeでは、正常値かエラー値の
  どちらかしかとれなかった

- エラーになる原因が複数あるような場合(I/O関係でよくある)
  に、「なにが原因でエラーになったか」が知りたい

- エラーになりうる操作を連続で呼び出す場合に、
  「どこでどういった原因で操作が中断されたか」が知りたい
  (たとえば、HaskellのMaybeモナド、SwiftやC# 6のOption)

- optionalやfutureの経験を活かして、正常値とエラー値の
  両方をとれるようにしよう！ => そしてexpectedへ

他言語の例

HaskellのMaybeモナド

```
#  失敗する可能性のある操作を連続して記述する
#  途中で失敗したら、それ以降の関数は呼ばれない
f >>= g >>= h
```

SwiftのOptional Chaining (C# 6のnull条件演算子も、?.構文)

```
// residenceとaddress、両方あったら住所の処理をする
// いずれかがなかったら、else節が呼ばれる
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
```

Chapter 2

expectedの概要と仕様

expectedの概要

- expectedは、正常値かエラー値の、どちらかが入る型

```
template <class T, class E>
class expected;
```

- 処理が正常に終了したときには戻り値を取得でき、
エラーになったときには、その原因を取得できる。

- 最近のoptionalのインタフェース(emplace系)に合わせて
設計されている。futureの設計経験も反映させている。

- HaskellのMonadErrorやEitherの考え方を取り入れ、失敗する
可能性のある操作のシーケンスを、エラー理由付きで
書きやすくしている。

使い方1 - エラーが起きる可能性のある処理側

```
// 0割りをハンドリングする除算関数
expected<double,error_condition> safe_divide(double i, double j)
{
  if (j == 0)
    return make_unexpected(arithmetic_errc::divide_by_zero);
  else
    return i / j;
}
```

- 正常の場合は、T型(ここではdouble)の値を代入する

- エラーの場合は、make_unexpected()関数に
  E型(ここではerror_condition)の値を渡す

使い方2 - エラーをハンドリングする側

```
expected<double, error_condition>
  f1(double i, double j, double k)
{
  auto q = safe_divide(j, k)
  if(q) return i + *q;
  else return q;
}
```

- ポインタのインタフェースで、正常かどうかの判定、および正常値の取得ができる。

- それぞれの値の取得は、value()メンバ関数とerror()メンバ関数を使用してもよい。

使い方3 - エラーハンドリングに より高級なインタフェースを使用する

```
expected<double, error_condition>
  f1(double i, double j, double k)
{
  return safe_divide(j, k).map([&](double q){
    return i + q;
  });
}
```

- map()メンバ関数に登録した関数は、expectedが正常値を持っている場合に、正常値を引数として呼び出される (expectedではなく、その中身である正常値が渡される)

- エラー値を持っている場合には呼び出されない。 (この場合は、エラー値がそのまま返される)

使い方4 - エラーの場合

```
expected<double, error_condition>
  f1(double i, double j, double k)
{
  return safe_divide(j, k).
                  catch_error([&](const error_condition& e) {
    if(e.value() == arithmetic_errc::divide_by_zero)
      return 0;
    return make_unexpected(e);
  });
}
```

- エラーをハンドリングする場合は、catch_error()メンバ関数を使用する。expectedがエラー値を持っている場合に、登録した関数が呼び出される。

使い方5 - 処理を連続で行う

```
expected<int, string> f() { return 3; }

f().map([](int x) {
  return x + 1;
}).map([](int y) {
  return to_string(y + 1);
}).map([](const string& z) {
  cout << "success : " << stoi(z) + 1 << endl;
}).catch_error([](const string& e) {
  cout << "error : " << e << endl;
  return make_unexpected(e);
});
```

- エラーになる可能性のある操作を列挙し、
  エラーを一括で処理する

正常値のハンドリング3種

- 正常値のハンドリングをするための関数は、map()のほかにbind()とthen()もある。

- Haskell風に書くと、以下のような型の違いがある：

```
map :: (T -> U) -> [U]
map :: (T -> [U]) -> [[U]]

bind :: (T -> U) -> [U]
bind :: (T -> [U]) -> [U]

then :: ([T] -> U) -> [U]
then :: ([T] -> [U]) -> [U]
```

(T -> U)で、T型を受け取ってU型を返す関数。

角カッコになっているものは、expectedでラップされた型

正常値のハンドリング3種

- map()は、expectedを返したときにexpected<expected<T, U>, U>となる。
- bind()は、expectedを返したときにexpected<T, U>となる。
- then()は、ハンドラ内でTではなくexpected<T, U>を扱う。(future::then()由来)

optional由来のインタフェース 1/2

- emplace()メンバ関数と、そのコンストラクタ版
  - 型Tのコンストラクタ引数から、
    正常値を持つexpectedオブジェクトを構築する

```
expected<T, E> e {in_place, T_args...};
e.emplace(T_args...);
```

- エラー値を持つexpectedオブジェクトを構築するコンストラクタ

```
expected<T, E> e {unexpect, E_args...};
```

- なお、デフォルト構築ではE()の値を持つ。

optional由来のインタフェース 2/2

- `template <class U> T value_or(U&&) &&;`
  - 正常値を持っている場合はそれを返し、
    エラーの場合はパラメータで指定された値を返す。

- 開発リポジトリでは、エラーの場合にテンプレートパラメータで指定された例外を送出する、`value_or_throw<Exception>()`メンバ関数もある。

expected<void, E>

- expectedは、正常値の型をvoidにすることを許可している。
- このような型のオブジェクトは、map()やbind()に指定した関数が何も返さないと作られる。
- expected<void, E>は、error_codeの代わりとしても使用できる。

```
// 正常値を生成する方法
expected<void, E> e {in_place};
e.emplace();
```

連想コンテナのキーとして使用する

- expectedは、連想コンテナのキーとして使用できるようにするために、以下をサポートしている：
  - operator<
  - operator==
  - std::hashの特殊化

expectedの現在の状況

- expectedは、標準C++に現在あるエラー報告の仕組みである例外を置き換えるものである。
- そのためexpectedは、現在のエラー報告の仕組みでできる全ての要件を満たさなければならない。
- 仕様は慎重に作る必要があるため、時間がかかる見込み。
- expectedの議論は、std-proposalsメーリングリスト、およびGitHub開発リポジトリのIssuesで行われているので、興味がある人は参加しよう。

まとめ

- `expected`は、正常値かエラー値の、いずれかが入る型である
- `optional`と違って、エラーとなった理由を保持できる
- 従来のif(false) -> return -> if(false) -> returnのような流れを、map/bindの高級インタフェースを使用することで、綺麗に書ける
- GitHubの参照実装を使用したり、`boost::variant`をラップして自分で作ってみたりして、試してみてください。