C++14 Concurrency TS

# Future APIの改善

株式会社ロングゲート
高橋 晶(Akira Takahashi)
faithandbrave@longgate.co.jp

2013/12/14(土) C++14規格レビュー勉強会 #2

# はじめに

- この発表は、C++14後のConcurrency TSに予定されている、Future関連API改善のレビュー資料です。

- 提案文書：
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3784.pdf

# 概要

- std::futureクラスとstd::shared_futureクラスに、
  以下のメンバ関数を追加する。
  - then()
  - unwrap()
  - is_ready()
- それに加え、以下の非メンバ関数を追加する。
  - when_any()
  - when_any_swapped()
  - when_all()
  - make_ready_future()

# 提案元

- 提案者はMicrosoftの
  - Niklas Gustafsson
  - Artur Laksberg,
  - Herb Sutter,
  - Sana Mithani
- Visual C++のPPL(Parallel Pattern Library)、.NET FrameworkのTask Parallel Libraryから得られた経験を取り入れたものである。

# Futureのおさらい

- Futureとは、並行プログラミングのデザインパターンのひとつ。
- スレッド間のデータ(結果値)渡しとして使用する。
- C++11から標準ライブラリに<future>ヘッダが導入された。
- 基本的な使い方として、promiseと対にして使用する。

# Futureの基本的な使い方

```
void calc(std::promise<int> p) {
    int sum = 0;
    for (int i = 1; i <= 10; ++i) {
        sum += i;
    }
    p.set_value(sum);
}

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t(calc, std::move(p));
    t.join();

    int result = f.get();
    std::cout << result << std::endl; // 55
}
```

# Futureの基本的な使い方

```
void calc(std::promise<int> p) {
    int sum = 0;
    for (int i = 1; i <= 10; ++i) {
        sum += i;
    }
    p.set_value(sum);
}

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t(calc, std::move(p));
    t.join();

    int result = f.get();
    std::cout << result << std::endl; // 55
}
```

1. promiseとfutureの
   共有状態を作る

# Futureの基本的な使い方

```
void calc(std::promise<int> p) {
    int sum = 0;
    for (int i = 1; i <= 10; ++i) {
        sum += i;
    }
    p.set_value(sum);
}

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t(calc, std::move(p));
    t.join();

    int result = f.get();
    std::cout << result << std::endl; // 55
}
```

2. バックグラウンドスレッドに
   promiseの所有権を移譲する

# Futureの基本的な使い方

```
void calc(std::promise<int> p) {
    int sum = 0;
    for (int i = 1; i <= 10; ++i) {
        sum += i;
    }
    p.set_value(sum);
}
```

3. バックグラウンドスレッドの処理が終わったら、結果をpromiseに書き込む

```
int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t(calc, std::move(p));
    t.join();

    int result = f.get();
    std::cout << result << std::endl; // 55
}
```

# Futureの基本的な使い方

```
void calc(std::promise<int> p) {
    int sum = 0;
    for (int i = 1; i <= 10; ++i) {
        sum += i;
    }
    p.set_value(sum);
}

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t(calc, std::move(p));
    t.join();

    int result = f.get();
    std::cout << result << std::endl; // 55
}
```

**4. promiseに書き込まれるのを待って、結果をfutureが読み込む。**

# async関数

- std::async()関数は、futureを使用したこのパターンをシンプルに使えるようにしたラッパー関数。
- この関数に登録された処理が終わるのを待ってpromiseに書き込みを行い(*)、スレッドの起動まで行ってくれる。

```
int calc() {
    …
    return sum;
}

int main() {
    std::future<int> f = std::async(calc);

    int result = f.get();
    std::cout << result << std::endl;
}
```

* 実際は、promiseをラップしたpackaged_taskというクラスを使用する。

# おさらい終了！

では、C++14後のConcurrency TSで
予定されている機能の紹介を行っていきます。

# thenメンバ関数

- then()メンバ関数は、非同期処理の平坦化(flatten)を行う関数。

  コールバック地獄を解消し、連続的な非同期処理を、ネストではなく
  シーケンシャルに記述できるようにする。

```
int main() {
    future<int> f1 = async([] { return 123; });
    future<std::string> f2 = f1.then([](future<int> f) {
        return to_string(f.get()) + "hoge";
    });

    std::cout << f2.get() << std::endl;
}
```

- then()に登録した関数は、futureオブジェクトの値を取り出す準備ができたら呼ばれる。(then内のf.get()はブロッキングせずに呼べる)

- launchポリシーは引き継がれる。

# thenメンバ関数

- 実際的なシーケンス非同期処理のサンプル

```
int main() {
    future<std::string> result =
        async([] { return 123; })
        .then([](future<int> f) {
            return to_string(f.get()) + "hoge";
        })
        .then([](future<string> f) {
            return f.get() + "fuga";
        });

    std::cout << result.get() << std::endl; // 123hogefuga
}
```

# thenメンバ関数

- 最終的な結果もthen()で処理する例。
- std::futureのデストラクタはwait()するので、最後のwait()はなくてもいい。
- boost::future(1.55.0 V4)のデストラクタはwait()しない。

```
int main() {
    async(launch::async, [] { return 123; })
    .then([](future<int> f) {
        return to_string(f.get()) + "hoge";
    })
    .then([](future<string> f) {
        return f.get() + "fuga";
    })
    .then([](future<string> f) { // 何も返さないとfuture<void>
        std::cout << f.get() << std::endl;
    }).wait();
}
```

# thenはなぜfutureを受け取るのか

- futureは値だけでなく、例外(別スレッドで発生したエラー)も
  受け取れるから。

```
int main() {
    future<std::string> result =
        async([]() -> int {
            throw std::runtime_error("error"); // エラー発生
        })
        .then([](future<int> f) -> std::string {
            try {
                f.get(); // ここで例外が投げ直される
            }
            catch (std::runtime_error& e) {
                return "error";
            }
            return "success";
        });
    std::cout << result.get() << std::endl; // error
}
```

# thenメンバ関数

- シグニチャは以下のようになる。

```
template<typename F>
auto then(F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto then(executor& ex, F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto then(launch policy, F&& func) -> future<decltype(func(*this))>;
```

- executorは別提案(N3785)のスケジューラ。スレッドプールも含まれる。
- launchポリシーの個別指定も可能。

# unwrapメンバ関数

- unwrap()メンバ関数は、ネストしたfutureの内側を取り出す。

- get()と違い、外側のfutureが準備完了するまで待たない。

  外側のfutureを除去し、内側を指すプロキシオブジェクトを返す。

```
int main() {
    future<future<int>> outer = async([] {
        future<int> inner = async([] {
            return 123;
        });
        return inner;
    });

    future<int> inner = outer.unwrap();
    std::cout << inner.get() << std::endl; // 123
}
```

- コールバック関数の中で別の非同期処理を登録する、というようなコードをfutureにすると、こういう状況になる。

# 暗黙のunwrap

- then()メンバ関数は、一段階だけ暗黙にunwrapする。

```
int main() {
    future<future<int>> outer = async([] {
        future<int> inner = async([] {
            return 123;
        });
        return inner;
    });

   outer.then([](future<int> inner) {
       std::cout << inner.get() << std::endl;
   }).wait();
}
```

# is_readyメンバ関数

- is_ready()メンバ関数は、値を取り出す準備ができているかを調べる。

```
struct GameLoop {
    future<int> f_;
    void start() {
        f_ = std::async([]() -> int { return …; });
    }

    // 1/60秒ごとに呼び出される
    void update() {
        if (f_.is_ready()) { // 準備ができたら取り出す
            int result = f_.get();
        }
    }
};
```

- これによって、タスクが完了したかをたびたび問い合わせる、というポーリングの設計が許可される。これがない頃は、f.wait_for(seconds(0))してfutureの状態を問い合わせていた。

※この関数は、C++11にfutureが提案された初期にはあったが、
機能が多すぎるという理由で提案から削除されていた。

# when系非メンバ関数

- when_any()/when_any_swapped()/when_all()非メンバ関数は、複数のfutureを受け取り、どれか一つ、もしくは全てが完了するまで待機するfutureを返す。
- これはfutureの合成操作であり、futureのOR(any)とAND(all)をサポートする。
  元となったPPLでは||、&&演算子もサポートしていたが、この提案には含まれない。
- これらの関数は、futureのコンテナ、もしくはタプルどちらかを使用できる。
  たとえば、when_any()のシグニチャは以下のようになっている：

```
template <class InputIterator>
future<vector<(shared_)future<R>>>
  when_any(InputIterator first, InputIterator last);

template <typename... T>
future<tuple<T...>>
  when_any(T&&... futures);
```

# when_any非メンバ関数

- when_any()非メンバ関数は、複数のfutureを受け取り、どれか一つが完了するまで待機するfutureを作って返す。

- 非同期操作のキューとして使える。

```
future<int> futures[] = {async([]() { return intResult(125); }),
                         async([]() { return intResult(456); })};

future<vector<future<int>>> any_f = when_any(begin(futures), end(futures));

future<int> result = any_f.then([](future<vector<future<int>> f) {
    // 準備ができた最初のひとつだけを使用する
    for (future<int> i : f.get()) {
        if (i.is_ready())
            return i.get();
    }
});
```

# when_any_swapped非メンバ関数

- when_any_swapped()非メンバ関数は、複数のfutureを受け取り、どれか一つが完了するまで待機するfutureを作って返す、when_any()の亜種。
- 最初に見つかった準備完了のfutureを、最後尾のfutureとswapする。
  こうすることで、準備が完了したfutureを定数時間で抽出できる。
- 先頭でなく最後尾なのは、返すコンテナがvectorで、pop_backが速いから。

```
future<int> futures[] = {async([]() { return intResult(125); }),
                         async([]() { return intResult(456); })};

future<vector<future<int>>> any_f =
                            when_any_swapped(begin(futures), end(futures));

future<int> result = any_f.then([](future<vector<future<int>> f) {
    // 最後尾の要素は、必ず準備完了したfuture
    return f.get().back().get();
});
```

# when_all非メンバ関数

- when_all()非メンバ関数は、複数のfutureを受け取り、その全てが完了するまで待機するfutureを作って返す。

- 並行アルゴリズムの待ち合わせに使える。

```
shared_future<int> shared_future1 = async([] { return intResult(125); });
future<string> future2 = async([]() { return stringResult( “hi” ); });

future<tuple<shared_future<int>, future<string>>> all_f =
    when_all(shared_future1, future2);

future<int> result = all_f.then(
  [](future<tuple<shared_future<int>, future<string>>> f) {
      return doWork(f.get());
  });
```

# make_ready_future非メンバ関数

- make_ready_future()非メンバ関数は、値を指定して準備完了したfutureを作るヘルパ関数。

- make_ready_future<void>()でfuture<void>ができる。例外版はない。

```
future<int> compute(int x) {
    // パラメータの値によっては、計算する必要がない。
    if (x < 0)  return make_ready_future<int>(-1);
    if (x == 0) return make_ready_future<int>(0);

    future<int> f1 = async([]() { return do_work(x); });
    return f1;
}
```

# 所感

- 今回追加が予定されているものは、他言語(たとえばC#)やライブラリ(たとえばPPL)で、実際の設計経験があるもの。
  そのため、十分に練られた設計になっている。

- 文面に細かいtypoを発見したが、C++の標準ライブラリに導入するに十分な動機と設計が成されていると感じた。

# 細かいtypo 1

- then()の例。
  intは組み込み型なので、to_string()というメンバ関数はない。

```
#include <future>
using namespace std;
int main() {
    future<int> f1 = async([]() { return 123; });
    future<string> f2 = f1.then([](future<int> f) {
        return f.get().to_string(); // here .get() won’ t block
    });
}
```

- std::to_string()に修正すべきです。

```
        return to_string(f.get()); // here .get() won’ t block
```

# 細かいtypo 2

- unwrap()の例。futureにテンプレート引数がない。

```
#include <future>
using namespace std;
int main() {
    future<future<int>> outer_future = async([]{
        future<int> inner_future = async([] {
            return 1;
        });
        return inner_future;
    });

    future<int> inner_future = outer_future.unwrap();

    inner_future.then([](future f) {
        do_work(f);
    });
}
```

- future<int>に修正すべきです。

# 参考情報

- N3747 A Universal Model for Asynchronous Operations
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3747.pdf
  - Boost.Asio作者であるChristopher Kohlhoff氏が、非同期操作の統合モデルを考察している文書。
  - 非同期操作APIの使用法として、コールバック関数の登録、future、コルーチンといったものを選択的にできる設計を示している。
- N3722 Resumable Functions
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3722.pdf
  - `future::then()`の言語サポート。`await`と`resumable`。
  - C#から始まり、最近はScalaにも実装があるもの。VC++ 2013 CTPで`__await`/`__resumable`として実験的に実装されている。