C++14 Concurrency TS

ExecutorとScheduler

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、ExecutorとSchedulerのレビュー資料です。
- 提案文書：
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3731.pdf
- いろいろと設計が不十分なようなので、その点を重視してレビューする。
  - 大きな懸念点はすでに、鈴木一生さんを通して提案者にフィードバック済み。改善案を検討してもらっている。

提案の概要

- 標準ライブラリに、executorとschedulerのクラス群を導入する。
- executorとは、複数タスクの実行を管理するクラス。
  - スレッドプールを含む。
- schedulerとは、executorにタスクの時間指定実行が付いたもの。
- これらのクラスは、Googleが社内で使っていたexecutorと、Microsoftのschedulerを合わせたもの。
- 規格にそのまま入れるようなProposal Wordingの形にはまだない。(たとえば、名前空間が記載されていない)

追加が予定されているヘッダ

<executor>	executorの抽象クラス。 class executor; class scheduled_executor : public executor;
<thread_pool>	スレッドプールのクラス。 class thread_pool : public scheduled_executor;
<serial_executor>	他のexecutorをFIFO実行するアダプタ。 class serial_executor : public executor;
<loop_executor>	シングルスレッドで使うexecutor。 class loop_executor : public executor;
<inline_executor>	キューイングせずにスレッドで実行するexecutor。 class inline_executor : public executor;

- ヘッダファイル多すぎじゃないですかね。
- 本来このような分割は望ましいですが、標準ライブラリではそうなっていない。1ライブラリ1ヘッダにするのがよい。

executorクラスのインタフェース

```
class executor {
public:
  virtual ~executor();
  virtual void add(function<void()> closure) = 0;
  virtual size_t num_pending_closures() const = 0;
};
```

- このクラス自体は、何もしない単なるインタフェースクラス。
- add()メンバ関数はタスクの追加。
- num_pending_closures()は実行待機中のタスク数を返す。

scheduled_executorクラスの インタフェース

```
class scheduled_executor : public executor {
public:
  …
  virtual void add_at(const chrono::system_clock::time_point& abs_time,
                      function<void()> closure) = 0;
  virtual void add_after(const chrono::system_clock::duration& rel_time,
                         function<void()> closure) = 0;
};
```

- このクラス自体も同様に、何もしない単なるインタフェースクラス。
- add_at()メンバ関数は、特定の日時に実行予約するタスクの追加。
- add_after()メンバ関数は、特定の時間経過したら実行するタスクの追加。

thread_poolクラス

```
class thread_pool : public scheduled_executor {
public:
  explicit thread_pool(int num_threads);
  ~thread_pool();
  …
};
```

- スレッドプール。指定された数のスレッドを使いまわして、
  タスクを実行する。
- コンストラクタでスレッド数を指定してプール。(size_tにすべし)
- デストラクタで全てのタスクが完了するまで待機。

serial_executorクラス

```
class serial_executor : public executor {
public:
  explicit serial_executor(executor* underlying_executor);
  virtual ~serial_executor();
  executor* underlying_executor();
  …
};
```

- 元となる他のexecutorを、FIFO順に処理することを保証するアダプタ。

loop_executorクラス

```
class loop_executor : public executor {
public:
  loop_executor();
  virtual ~loop_executor();
  void loop();
  void run_queued_closures();
  bool try_run_one_closure();
  void make_loop_exit();
  …
};
```

- シングルスレッドでFIFO実行するexecutor。
  タスクの追加や実行中断は他のスレッドからも行える。
- `loop()`は`make_loop_exit()`がほかのスレッドから呼ばれるまでひたすらタスクを実行＆ブロッキング。
- `run_queued_closures()`はすでにキューが入ってるものが空になるまで実行。`try_run_one_closure()`の方は1タスクだけ実行。

inline_executorクラスのインタフェース

```
class inline_executor : public executor {
public:
  explicit inline_executor();
  …
};
```

- キューイングしないシンプルなexecutor。
- add()メンバ関数でタスクを追加するとその場で実行される。
- ダミー実装として使う。

default executor

```
shared_ptr<executor> default_executor();
void set_default_executor(shared_ptr<executor> executor);
```

- つまりは切り替え可能でグローバルなexecutor。
- `std::async(launch::async, f);`で使用されるexecutor。

この設計の問題点 1/2

- メモリアロケーションが考慮されていない。
  - 内部コンテナのアロケータ(型／オブジェクト)を指定できない。
  - function<void()>を受け取る設計。このクラスは内部でnewする。
- メモリアロケーションを考慮すると、設計が根本的に変わりうる。
  - たとえば、function<void()>の代わりに

    template <class F> void add(F f);

    とする場合、このメンバ関数は仮想関数にできない。
  - アロケータ型をクラスのテンプレートパラメータで指定する必要があるかもしれない。(現在は非テンプレート)

この設計の問題点 2/2

- 仮想関数が必要なのは、default_executorとserial_executorのため。
  これらの機能は本当に必要だろうか？async()は今まで通り、呼ばれるたびにexecutorの関係なしにスレッドを作ればいいのではないか。
  これらのために、多くの設計選択を犠牲にしすぎているのではないか。

- いずれにしても、メモリアロケーションに対する方針が必要。
  考慮不足の機能は、寿命が短くなる(deprecatedになりやすい)。
  ライブラリの標準化には、十分な設計・使用経験が必要だと考える。

考えられる設計

- default executorさえなくなれば設計の幅がかなり向上するので、
  たとあえばアロケータ指定や実装のバリエーションを考慮し、
  以下のようにPolicy-based Designにするのはどうだろう？

```
using thread_pool = basic_executor<
                      thread_pool_model,
                      allocator<function<void()>> // default arg
                    >;
```

- デフォルトのアロケータを使いたくなければ自分でエイリアスを
  定義し直すだけでいいようにする。

- 基礎となるexecutorに共通インタフェースと実装を持たせ、
  可変部は各ポリシー(thread_pool, serial, inline)としてパラメータ化する。