Boost.Contextで継続

高橋 晶(Akira Takahashi)
id:faith_and_brave/@cpp_akira

2012/04/08(土) Boost.Contextオンリーイベント

Boost.Context

- Oliver Kowalkeが作ったBoost.Fiberというライブラリの、コンテキストスイッチ部分を切り出したライブラリ
- Boost 1.49.0時点でtrunkに入ってる
- 継続(continuation)、ファイバー(fiber)、コルーチン(coroutine)、マイクロスレッド(microthread)などなど、いろんな呼び名のある技術を実現することができる。

資料などのあるところ

- Boost.Context について調べた – melpon日記
  http://d.hatena.ne.jp/melpon/20111213/1323704464
- ドキュメント
  http://ok73.ok.funpic.de/boost/libs/context/doc/html/
- ソースコード
  https://github.com/ryppl/boost-svn
  boost/context以下、およびlibs/context以下にソースコードおよびサンプル、テストがある。

継続は何をするのか

- スレッドがやってることを自前でやる
- ユニプロセッサを考える。
  スレッドは一つのプロセッサで複数の処理を同時に実行するために、
  「少し実行してコンテキスト内のスタックを一旦保存して別なコンテキストに切り替えて…」
  ということをおこなっている。
- 継続と呼ばれる概念では、このコンテキストスイッチを自前で行うことにより、いくつかのケースのプログラムを自然な流れで書くことができる。

基本的な使い方

- boost::contexts::contextというクラスのコンストラクタで以下を設定する:
  - resume()時に呼ばれる関数
  - スタックサイズ
  - 最後にデストラクタを呼ぶかどうか
  - 呼び元に必ず戻るかどうか
- context::suspend()メンバ関数
  - コンテキストを中断
- context::resume()メンバ関数
  - コンテキストを再開
- context::start()メンバ関数
  - コンテキストを開始(2回目以降はresume()を呼ぶ)

継続クラス(contextのラッパー)

```
#include <boost/context/all.hpp>
#include <boost/function.hpp>
#include <boost/utility/value_init.hpp>

class continuation {
  boost::contexts::context ctx_;
  boost::function<void(continuation&)> fn_;
  boost::initialized<bool> started_;
  boost::initialized<bool> is_break_;

  void trampoline_() { fn_(*this); }

public:
  continuation() {}
```

継続クラス(contextのラッパー)

```
continuation(boost::function<void(continuation&)> const& fn)
{
  ctx_ = boost::contexts::context(
              &continuation::trampoline_,
              this,
              boost::contexts::default_stacksize(),
              boost::contexts::stack_unwind,
              boost::contexts::return_to_caller);
   fn_ = fn;
}
```

継続クラス(contextのラッパー)

```
continuation& operator=(continuation&& other)
{
  // thisに依存してるとmoveできないので作り直し
  ctx_ = boost::contexts::context(
              &continuation::trampoline_,
              this,
              boost::contexts::default_stacksize(),
              boost::contexts::stack_unwind,
              boost::contexts::return_to_caller);

  fn_ = boost::move(other.fn_);
  started_ = boost::move(other.started_);
  is_break_ = boost::move(other.is_break_);
  return *this;
}
```

継続クラス(contextのラッパー)

```
int resume()
{
  if (!started_.data()) {
    started_.data() = true;
    is_break_.data() = false;
    return ctx_.start();
  }
  else { return ctx_.resume(); }
}

int restart()
{
  started_.data() = false;
  is_break_.data() = false;
  ctx_ = boost::contexts::context(
              &continuation::trampoline_,
              this,
              boost::contexts::default_stacksize(),
              boost::contexts::stack_unwind,
              boost::contexts::return_to_caller);
  return resume();
}
```

継続クラス(contextのラッパー)

```
  int suspend(int vp = 0)
  { return ctx_.suspend(vp); }

  int suspend_break(int vp = 0)
  {
    is_break_.data() = true;
    return ctx_.suspend(vp);
  }

  bool is_complete() const
  { return is_break_.data() || ctx_.is_complete(); }

  void unwind_stack() { ctx_.unwind_stack(); }
};
```

考えられる基本的なユースケース

- 非同期処理の逐次化
  - コールバック地獄の解消
- (ゲームループなど)継続的に実行する処理の逐次化
  - ファイルの読み込み、シューティングゲームの弾道など
- リスト処理の遅延評価のようなこと
  - C#のyield return/IEnumerableみたいなこと
  - イテレータ／ジェネレータ

非同期処理の逐次化

通常の非同期処理

```
void foo()
{
  async_xxx(…, bar);
}

void bar()
{
  async_xxx(…, hoge);
}

void hoge();
```

コールバック関数をあちこちに書く

継続バージョン

```
async_xxx(…, complete);
suspend();

async_xxx(…, complete);
suspend();

…

void complete()
{
  resume();
}
```

非同期処理を実行
→ 中断
→ コールバック関数内で再開を指示

非同期処理の逐次化

```
class Client {
  io_service& io_service_;
  socket socket_;

  continuation cont_;
  error_code ec_;

public:
  Client(io_service& io_service)
    : io_service_(io_service), socket_(io_service)
  {
    cont_ = continuation(bind(&Client::do_, this));
  }

  void start()
  {
    ec_ = error_code();
    cont_.resume();
  }
  …
```

非同期処理の逐次化

```
private:
  void do_()
  {
    while (!ec_) {
      const string req = "ping¥n";
      async_write(socket_, buffer(req), bind(…, completion_handler));
      cont_.suspend();

      if (ec_) break;

      streambuf rep;
      async_read_until(socket_, rep, '¥n', bind(…, completion_handler));
      cont_.suspend();

      if (ec_) break;

       cout << buffer_cast<const char*>(rep.data());
    }

    cout << ec_.message() << endl;
  }
```

非同期処理の逐次化

```
  void completion_handler(const error_code& ec)
  {
    ec_ = ec;
    cont_.resume();
  }
};


asio::io_service io_service;
Client client(io_service);

client.start();

io_service.run();
```

http://d.hatena.ne.jp/faith_and_brave/20120329/1333008572

(ゲームループなど)継続的に実行する処理の逐次化

通常の定期実行処理

```
// 定期的に呼ばれる関数
void updatePosition()
{
  switch (count_) {
   case 0: x = 0; y = 0;
   case 1: x = 1; y = 0;
   case 2: x = 1; y = 1;
   case 3: x = 0; y = 1;
   case 4: x = 0; y = 0;
  }
  count_++;
}
```

どこまで実行したかを自分で覚えておき、次回呼ばれたときに状態に合わせた地点から実行

継続バージョン

```
// 定期的に呼ばれる関数
void updatePosition()
{
  x = 0; y = 0; suspend();
  x = 1; y = 0; suspend();
  x = 1; y = 1; suspend();
  x = 0; y = 1; suspend();
  x = 0; y = 0; suspend();
}
```

次回呼ばれたときに次の行を実行
(自然な流れ)

(ゲームループなど)継続的に実行する処理の逐次化

```
class Game {
  vector<string> data;
  continuation cont;
public:
  Game()
    : cont(bind(&Game::load_file, this)) {}

  void update()
  {
    if (!cont.is_complete()) {
        cont.resume();
        cout << data.back() << endl;
    }
  }

  void draw() {}
```

(ゲームループなど)継続的に実行する処理の逐次化

```
private:
  void load_file()
  {
    ifstream file("a.txt");
    string line;
    while (std::getline(file, line)) {
      data.push_back(line);
      if (file.peek() != EOF) {
        cont.suspend();
      }
    }
  }
};
```

(ゲームループなど)継続的に実行する処理の逐次化

```
const milliseconds timer_duration(static_cast<int>(1.0 / 60.0 * 1000));

void on_timer(Game& game, steady_timer& timer)
{
  game.update();
  game.draw();

  timer.expires_from_now(timer_duration);
  timer.async_wait(
    bind(&on_timer, ref(game), ref(timer)));
}

Game game;
io_service io_service;
steady_timer timer(io_service);

timer.expires_from_now(timer_duration);
timer.async_wait(
         bind(&on_timer, ref(game), ref(timer)));

io_service.run();
```

http://d.hatena.ne.jp/faith_and_brave/20120312/1331537869

リスト処理の遅延評価

```
// nから始まる無限リスト
void enum_from(continuation& cont, int n)
{
  for (int i = n;; ++i) {
    cont.suspend(i);
  }
}

int main()
{
  continuation cont(bind(enum_from, _1, 3));

  // 3から始まる無限リストから先頭10個を取り出す
  for (int i = 0; i < 10; ++i) {
    int n = cont.resume();
    cout << n << endl;
  }
}
```

これを抽象化したBoost.Enumeratorというのが開発されている
https://github.com/jamboree/boost.enumerator

中断可能なアルゴリズム

- 中断可能なアルゴリズムを後付で作れないかと試した。
  Boost.Graphでの最短経路の計算を外部から少しずつ実行する。
- Boost.GraphにはEvent Visitorというのがあり、最短経路計算において「点が交差したときに指定された関数を呼ぶ」というようなことができる。
  ここに継続を置いてみた。

```
class path_history_visitor {
  continuation& cont;
public:
  typedef boost::on_discover_vertex event_filter;

  path_history_visitor(continuation& cont) : cont(cont) {}

  template<typename Vertex, typename Graph>
  void operator()(Vertex v, const Graph&)
  {
    std::cout << Names[v] << ' ';
    cont.suspend();
  }
};
```

```
int main()
{
  const Graph g = make_graph();
  const Vertex from = S;

  continuation cont = [&](continuation& cont) {
    path_history_visitor vis(cont);

    boost::dijkstra_shortest_paths(g, from,
           boost::visitor(boost::make_dijkstra_visitor(vis)));
  };

  while (!cont.is_complete()) {
    cont.resume();
  }
}
```

ポイントはラムダ式とEvent Visitor。
Event Visitorは、アルゴリズムの中断ポイントを定義するのに使用できる。
ラムダは、既存のプログラムをBoost.Contextで使用できるコンテキストに変換するのに使用できる。

その他

- ambを実装した
  https://github.com/faithandbrave/Keizoku/tree/master/amb

- Boost.CoroutineがBoost.Contextで書き直されるらしい
  「Boost.Coroutineを用いてステートマシンを解決する – C++Now! 2012」
  https://sites.google.com/site/boostjp/cppnow/2012#coroutine

- Scala設計者のMartin Odersky先生が「Observerパターンやめて継続使おうぜ」という論文を書いてる
  「Deprecating the Observer Pattern」
  http://lampwww.epfl.ch/~imaier/pub/DeprecatingObserversTR2010.pdf