Нестандартный future/promise

Сергей Видюк

Кратко обо мне

- Работаю в команде 3D карты мобильной версии 2ГИС
- Написал свою реализацию `future/promise`

`std::future` FAQ

Q: Когда мы хотим запускать асинхронные задачи?
A: Кода нам нельзя блокировать текущий поток.
Q: Что мы можем сделать с `std::future` на результат асинхронной задачи?
A: Заблокироваться чтобы получить результат.

Альтернативы

- portable_concurrency
- Folly
- HPX
- Kokkos
- Seastar

Давайте напишем прогу, которая...

- Рисует карту на базе web-тайлов
- Рисует много маркеров POI разных приоритетов
- Не использует mutex...
- Не зависает и не падает на выходе
- Лежит на github:
  `https://github.com/VestniK/mapex`

План работ

- В начале у нас есть синхронный прототип рисующий тайлы лежащие локально на диске.
- Распаковка изображений происходит синхронно прямо в UI потоке.

План работ

- Унесём распаковку в отдельные потоки.
- Добавим загрузку изображений по сети.
- Реализуем загрузку и кэширование POI.
- Распараллелим генерализацию POI разных приоритетов.

На всякий случай

В угоду лаконичности и удобству чтения слайдов код примеров в презентации упрощён, не следует правилам хорошего тона и может даже не компилироваться.

`future` в UI потоке

- Всё взаимодействие с интерфейсом должно происходить в одном потоке.
- Обработка пользовательких действий происходит в нём же.
- Его нельзя блокировать на долго.

future в UI потоке

Есть задачи по загрузке тайлов и загруженные тайлы.

```
struct tile_id {
  int tx;
  int ty;
  int z_level;
};

map<tile_id, future<QImage>> tasks;
map<tile_id, QImage> tiles;
```

future в UI потоке

Перед отрисовкой хотим забирать готовые тайлы без SMS и блокировок.

```
map<tile_id, future<QImage>> tasks;
map<tile_id, QImage> tiles;

for (auto [id, task]: tasks) {
  if (task.is_ready())
    tiles[id] = task.get();
}
```

future в UI потоке

Или в std::future исполнении

```
map<tile_id, std::future<QImage>> tasks;
map<tile_id, QImage> tiles;

for (auto [id, task]: tasks) {
  if (
    task.wait_for(0ms) == std::future_status::ready)
    tiles[id] = task.get();
}
```

Распаковываем тайлы `async`'ронно

- Самый простой способ создать асинхронную задачу `async`.
- Принимает функцию и аргументы.
- Возвращает `future` на результат.

Распаковываем тайлы async'ронно

Асинхронная загрузка в исполнении std::async

```
QImage load_tile(tile_id id);
map<tile_id, std::future<QImage>> tasks;

tasks[id] = std::async(
  launch::async,
  [id] {return load_tile(id);}
);
```

Распаковываем тайлы `async`'ронно

Проблемы:

- Удаление задач только через ожидание их завершения
- На выходе ждём завершения всех текущих задач

Распаковываем тайлы async'ронно

Перейдём на pc::async

```
QImage load_tile(tile_id id);
map<tile_id, future<QImage>> tasks;

tasks[id] = async(
  QThreadPool::globalInstance(),
  [id] {return load_tile(id);}
);
```

Executors

Откуда `pc::async` знает про `QThreadPool`?

- portable_concurrency не зависит от Qt
- не предоставляет своих рекомендуемых к использованию executor’ов

Executors

```
namespace portable_concurrency {
template <>
struct is_executor<QThreadPool*>: std::true_type {};
} // namespace portable_concurrency

// ADL
template<typename F>
// requires Callable<F, void()> && MoveConstructable<F>
void post(
  QThreadPool* pool,
  F&& task
);
```

Executors

```
namespace portable_concurrency {
template <>
struct is_executor<QThreadPool*>: std::true_type {};
} // namespace portable_concurrency

// ADL
void post(
  QThreadPool* pool,
  unique_function<void()> task
);
```

Executors

```
namespace portable_concurrency {
template <>
struct is_executor<QObject*>: std::true_type {};
} // namespace portable_concurrency

// ADL
void post(
  QObject* pool,
  unique_function<void()> task
);
```

Executors

Требования на Executor:

- Дешёвый для копирования.
- Хранение его копии в задаче отсылаемой через `post` не приводит к циклическим ссылкам и утечкам.

Executors

Разобравшись в нюансах мы можем наказать `QThreadPool` *за длинное имя и в последующих слайдах принебрежительно писать* `pool` *вместо* `QThreadPool::globalInstance()`.

Сетевые запросы и `promise`

- Мы хотим работать с Qt сетью через `future`.
- Но она асинхронно работает через сигналы.
- На помощь приходит `promise` самый низкоуровневый и сложный в обращении способ взаимодействовать с `future`.

Сетевые запросы и promise

```
class promised_reply: QObject {
  promise<QNetworkReply*> p_;
slots:
  void on_reply_finished() {
    p_.set_value(sender());
  }
  void on_reply_error() {
    p_.set_error(sender()->error());
  }
};
```

Сетевые запросы и promise

```
future<QNetworkReply*> send_request(
  QNetworkAccessManager nm, QUrl url
) {
  promised_reply* listener = new promised_reply;
  QNetworkReply* reply = nm.get(url);
  QMetaObject::connectSlotsByName(this);
  return listener.p_.get_future();
}
```

future<future<future<T>>>

- Вызывать `send_request` можно только в сетевом потоке.
- Она возвращает `future`.
- Функция `async` добавляет к в озвращаемому значению `future`.

future<future<future<T>>>

```
QNetworkAccessManager nm;

auto f = async(
  &nm, [&nm] {return send_request(nm, url);}
);
```

Какой тип вернёт нам async?

future<future<future<T>>>

```
QNetworkAccessManager nm;

future<QNetworkReply*> f = async(
  &nm, [&nm] {return send_request(nm, url);}
);
```

Удобный в использовании, а не тот, что в заголовке.

Цепочки задач

Получение `QImage` пригодного для отрисовки состоит из двух этапов.

- Загрузки по сети.
- Распаковки полученного изображения.

Второй шаг необходимо уметь запускать по факту завершения первого.

Цепочки задач

```
map<tile_id, std::future<QImage>> tasks;
QUrl get_tile_url(tile_id);
QImage parse_image(QIODevice*);

tasks[id] = async(&nm, [&nm, id] {
  return send_reques(nm, get_tile_url(id));
});
```

Цепочки задач

```
map<tile_id, std::future<QImage>> tasks;
QUrl get_tile_url(tile_id);
QImage parse_image(QIODevice*);

tasks[id] = async(&nm, [&nm, id] {
  return send_reques(nm, get_tile_url(id));
}).next(pool, [](QNetworkReply* reply) {
  return parse_image(reply);
});
```

Цепочки задач

```
map<tile_id, std::future<QImage>> tasks;
QUrl get_tile_url(tile_id);
QImage parse_image(QIODevice*);

tasks[id] = async(&nm, [&nm, id] {
  return send_reques(nm, get_tile_url(id));
}).next(pool, [](QNetworkReply* reply) {
  return parse_image(reply);
}).then(&widget, [&widget](future<QImage> f) {
  widget.update();
  return f;
});
```

Отмена задач специфично для portable_concurrency

Q: Будет ли выполняться `parse_image` для уже невидимых тайлов?
A: Если он ещё не стартовал, а `future` на его результат уже разрушен, то нет.

Отмена задач специфично для portable_concurrency

Q: А может нужно прервать саму загрузку тайлов?
A: Обработчик отмены назавершённой задачи можно передать в конструктор `promise`.

Отмена задач специфично для portable_concurrency

Q: А может нужно прервать саму загрузку тайлов?

A: Обработчик отмены назавершённой задачи можно передать в конструктор `promise`.

Осторожно, поток вызова обработчика неопределён.

Отмена задач специфично для portable_concurrency

```
QNetworkReply* reply = nm.get(url);
promise<QNetworkReply*> p{
  canceller_arg, [reply] {reply->abort();}
};
```

Отмена задач специфично для portable_concurrency

Q: А что делать с очень долго выполняющимся синхронным кодом?
A: Есть ещё одна форма `then` и `promise::is_awaiten`.

Отмена задач специфично для portable_concurrency

```
future<T> f;
future<R> res = f.then(
  [](promise<R> p, future<T> f) {
    not_too_long_op1();
    if (!p.is_awaiten()) return;
    not_too_long_op2();
  }
);
```

Загрузка POI

- База POI загружается по сети и кэшируется локально.
- Включение отображения POI должно происходить быстро.
- Одновременно идём в сеть и в кэш и смотрим кто быстрей.

Загрузка POI

```
future<points> load_poi();
points fetch_poi_cache();
using poi_future = ???;

poi_future f = when_any(
  load_poi(),
  async(pool, fetch_poi_cache())
);
```

Тип `poi_future` не влез на этот слайд

Загрузка POI

```
template<typename Sequence>
struct when_any_result {
  size_t index;
  Sequence futures;
};

using poi_future = future<
  when_any_result<
    tuple<future<points>, future<points>>
  >
>;
```

Загрузка POI

```
future<points> load_poi();
points fetch_poi_cache();
using poi_future = when_any_result<
  vector<future<points>>
>;

future<points> futures[2] = {
  load_poi(),
  async(pool, fetch_poi_cache())
};
auto f = when_any(begin(futures), end(futures));
```

Быстрый кэш и отмена запроса

- Кэш всегда отвечает первым
- `future` на результат загрузки по сети разрушается
- Хочется при этом не отменять загрузку
- Для этого есть метод `future::detach`

Быстрый кэш и отмена запроса

```
template<typename T>
future<T> future<T>::detach();

future<points> futures[2] = {
  load_poi().detach(),
  async(pool, fetch_poi_cache())
};
```

Быстрый кэш и отмена запроса

Отмену можно нарезать фигурно

```
auto f = async(pool, foo)
  .next(bar).detach()
  .next(baz);
```

Не рисуем пустой кэш

- Когда кэш пуст отдавать его содержимое на отрисовку бессмысленно.
- В этом случае хочется продолжить дожидаться ответа от сети.

Не рисуем пустой кэш

```
future<points> f =
  when_any(begin(futures), end(futures))
  .next([](auto seq) {
    points poi = seq.futures[seq.index].get();
    if (poi.empty() && seq.index == 1)
      return seq[0];
    return make_ready_future(poi);
  });
```

Генерализация POI

- Есть два приоритета POI «рекламные/обычные».
- Генерализуем разные приоритеты независимо.
- А затем объединяем результат.

Генерализация POI

```
using points = vector<geo_point>;
points generalize(points, int z_level);

points ads;
points poi;
auto f_ads = async(pool,
  [=] {return generalize(ads, z_level);});
auto f_poi = async(pool,
  [=] {return generalize(poi, z_level);});
```

Генерализация POI

```
auto f = when_all(f_ads, f_poi)
  .next([](
    tuple<future<points>, future<points>> ready
  ) {
    points g_ads = get<0>(ready).get();
    points g_poi = get<1>(ready).get();
    // ...
  });
```

В итоге мы имеем

- Начав с синхронного прототипа мы легко сделали код асинхронным.

В итоге мы имеем

- Начав с синхронного прототипа мы легко сделали код асинхронным.
- Код управления потоками отделён от описания асинхронных задач.

В итоге мы имеем

- Начав с синхронного прототипа мы легко сделали код асинхронным.
- Код управления потоками отделён от описания асинхронных задач.
- Порядок исполнения строго определён зависимостью задач друг от друга.

В итоге мы имеем

- Начав с синхронного прототипа мы легко сделали код асинхронным.
- Код управления потоками отделён от описания асинхронных задач.
- Порядок исполнения строго определён зависимостью задач друг от друга.
- Механизм описания зависимостей не позволяет создавать циклов.

Спасибо за внимание

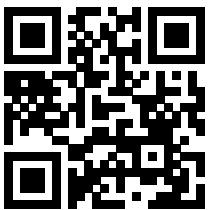
Мои контакты:

sir.vestnik@gmail.com

https://github.com/VestniK

Ссылки:

Демо проект:



portable_concurrency: