**Facebook 的 C++ 11 组件库 Folly Futures 【已翻译100%】**

英文原文：[Futures for C++11 at Facebook](https://code.facebook.com/posts/1661982097368498/futures-for-c-11-at-facebook/)

标签： [Folly](https://www.oschina.net/translate/tag/folly)

[oschina](https://my.oschina.net/osadmin) 推荐于 2年前 (共 10 段, 翻译完成于 09-23) [评论 18](https://www.oschina.net/translate/futures-for-c-11-at-facebook#comments)

分享

收藏

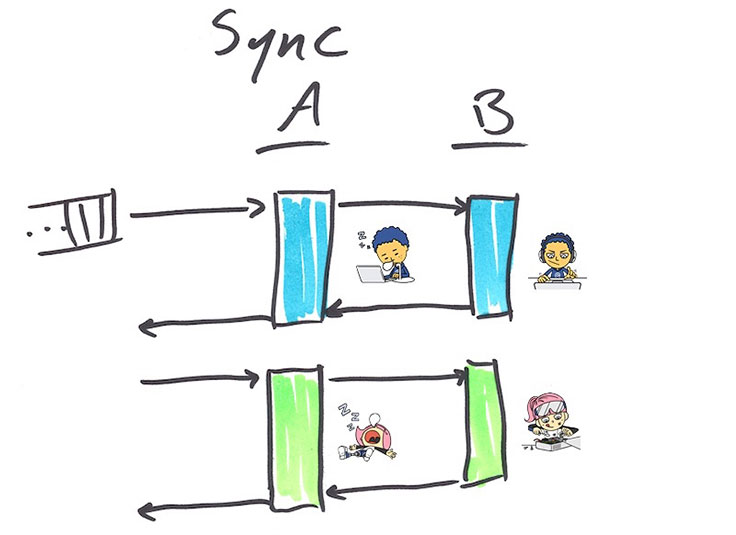
87

参与翻译 (1人) : [北风其凉](https://my.oschina.net/Tsybius2014)[仅中文](https://www.oschina.net/translate/futures-for-c-11-at-facebook) | [中英文对照](https://www.oschina.net/translate/futures-for-c-11-at-facebook?cmp) | [仅英文](https://www.oschina.net/translate/futures-for-c-11-at-facebook?lang=eng) | [打印此文章](https://www.oschina.net/translate/futures-for-c-11-at-facebook?print)

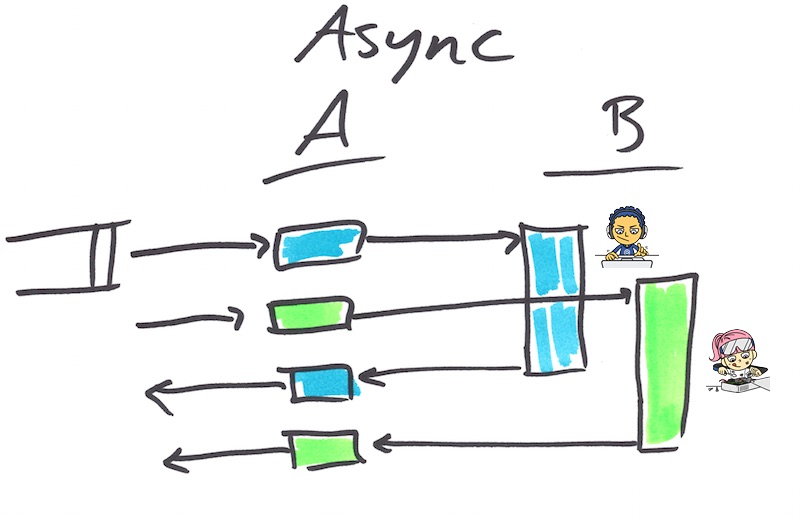
Futures 是一种通过自然的、可组合的方式表达异步计算的模式。这篇博文介绍了我们在 Facebook 中使用的一种适用于 C++11 的 futures 实现：[Folly Futures](https://github.com/facebook/folly/tree/master/folly/futures)。

**为什么要使用异步？**

想象一个服务 A 正在与服务 B 交互的场景。如果 A 被锁定到 B 回复后才能继续进行其他操作，则 A 是同步的。此时 A 所在的线程是空闲的，它不能为其他的请求提供服务。线程会变得非常笨重-切换线程是低效的，因为这需要耗费可观的内存，如果你进行了大量这样的操作，操作系统会因此陷入困境。这样做的结果就是白白浪费了资源，降低了生产力，增加了等待时间（因为请求都在队列中等待服务）。



如果将服务 A 做成异步的，会变得更有效率，这意味着当 B 在忙着运算的时候，A 可以转进去处理其他请求。当 B 计算完毕得出结果后，A 获取这个结果并结束请求。



[](https://my.oschina.net/Tsybius2014)

[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

1人顶

顶 翻译得不错哦！

**同步代码与异步代码的比较**

让我们考虑一个函数 fooSync，这个函数使用完全同步的方式完成基本计算 foo，同时用另一个函数 fooAsync 异步地在做同样的工作。fooAsync 需要提供一个输入和一个能在结果可用时调用的回调函数。

template <typename T> using Callback = std::function<void(T)>;

Output fooSync(Input);

void fooAsync(Input, Callback<Output>);

这是一种传统的异步计算表达方式。（老版本的 C/C++ 异步库会提供一个函数指针和一个 void\* 类型的上下文参数，但现在 C++11 支持隐蔽功能，已经不再需要显式提供上下文参数）

[](https://my.oschina.net/Tsybius2014)

[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

1人顶

顶 翻译得不错哦！

传统的异步代码比同步代码更为有效，但它的可读性不高。对比同一个函数的同步和异步版本，它们都实现了一个 multiFoo 运算，这个运算为输入向量（vector）中的每一个元素执行 foo 操作：

using std::vector;vector<Output> multiFooSync(vector<Input> inputs) {

  vector<Output> outputs;

  for (auto input : inputs) {

    outputs.push\_back(fooSync(input));

  }

  return outputs;}

void multiFooAsync(

  vector<Input> inputs,

  Callback<vector<Output>> callback){

  struct Context {

    vector<Output> outputs;

    std::mutex lock;

    size\_t remaining;

  };

  auto context = std::make\_shared<Context>();

  context->remaining = inputs.size();

  for (auto input : inputs) {

    fooAsync(

      input,

      [=](Output output) {

        std::lock\_guard<std::mutex> guard(context->lock);

        context->outputs->push\_back(output);

        if (--context->remaining == 0) {

          callback(context->outputs);

        }

      });

  }}

异步的版本要复杂得多。它需要关注很多方面，如设置一个共享的上下文对象、线程的安全性以及簿记工作，因此它必须要指定全部的计算在什么时候完成。更糟糕的是（尽管在这个例子中体现得并不明显）这使得代码执行的次序关系（computation graph）变得复杂，跟踪执行路径变得极为困难。程序员需要对整个服务的状态机和这个状态机接收不同输入时的不同行为建立一套思维模式，并且当代码中的某一处不能体现流程时要找到应该去检查的地方。这种状况也被亲切地称为“回调地狱”。

[](https://my.oschina.net/Tsybius2014)

[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

1人顶

顶 翻译得不错哦！

**Futures**

Future 是一个用来表示异步计算结果（未必可用）的对象。当计算完成，future 会持有一个值或是一个异常。例如：

#include <folly/futures/Future.h>

using folly::Future;

// Do foo asynchronously; immediately return a Future for the output

Future<Output> fooFuture(Input);

Future<Output> f = fooFuture(input);

// f may not have a value (or exception) yet. But eventually it will.

f.isReady();  // Maybe, maybe not.

f.wait();     // You can synchronously wait for futures to become ready.

f.isReady();  // Now this is guaranteed to be true.

Output o = f.value();  // If f holds an exception, this will throw that exception.

到目前为止，我们还没有做任何[std::future](http://l.facebook.com/l.php?u=http%3A%2F%2Fen.cppreference.com%2Fw%2Fcpp%2Fthread%2Ffuture&h=lAQEO_G7e&s=1)不能做的事情。但是 future 模式中的一个强有力的方面就是可以做到连锁回调，std::future 目前尚不支持此功能。我们通过方法 [Future::then](https://www.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md%23setting-callbacks-with-then-&h=yAQG1btiF&s=1) 来表达这个功能：

Future<double> f =

  fooFuture(input)

  .then([](Output o) {

    return o \* M\_PI;

  })

  .onError([](std::exception const& e) {

    cerr << "Oh bother, " << e.what()

      << ". Returning pi instead." << endl;

    return M\_PI;

  });// get() first waits, and then returns the valuecout << "Result: " << f.get() << endl;

在这里我们像使用 onError 一样使用连接起来的 then 去接住可能引发的任何异常。可以将 future 连接起来是一个重要的能力，它允许我们编写串行和并行的计算，并将它们表达在同一个地方，并为之提供明晰的错误处理。

[](https://my.oschina.net/Tsybius2014)

[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

1人顶

顶 翻译得不错哦！

**串行功能组成**

如果你想要按顺序异步计算 a、b、c 和 d，使用传统的回调方式编程就会陷入“回调地狱”- 或者，你使用的语言具备一流的匿名函数（如 C++11），结果可能是“回调金字塔”：

// the callback pyramid is syntactically annoying

void asyncA(Output, Callback<OutputA>);

void asyncB(OutputA, Callback<OutputB>);

void asyncC(OutputB, Callback<OutputC>);

void asyncD(OutputC, Callback<OutputD>);

auto result = std::make\_shared<double>();

fooAsync(input, [=](Output output) {

  // ...

  asyncA(output, [=](OutputA outputA) {

    // ...

    asyncB(outputA, [=](OutputB outputB) {

      // ...

      asyncC(outputB, [=](OutputC outputC) {

        // ...

        asyncD(outputC, [=](OutputD outputD) {

          \*result = outputD \* M\_PI;

        });

      });

    });

  });

});

// As an exercise for the masochistic reader, express the same thing without

// lambdas. The result is called callback hell.

有了 futures，顺序地使用then组合它们，代码就会变得干净整洁：

Future<OutputA> futureA(Output);

Future<OutputB> futureB(OutputA);

Future<OutputC> futureC(OutputB);

// then() automatically lifts values (and exceptions) into a Future.

OutputD d(OutputC) {

  if (somethingExceptional) throw anException;

  return OutputD();}Future<double> fut =

  fooFuture(input)

  .then(futureA)

  .then(futureB)

  .then(futureC)

  .then(d)

  .then([](OutputD outputD) { // lambdas are ok too

    return outputD \* M\_PI;

  });

**并行功能组成**

再回到我们那个 multiFoo 的例子。下面是它在 future 中的样子：

using folly::futures::collect;

Future<vector<Output>> multiFooFuture(vector<Input> inputs) {

  vector<Future<Output>> futures;

  for (auto input : inputs) {

    futures.push\_back(fooFuture(input));

  }

  return collect(futures);}

collect 是一种我们提供的构建块（compositional building block），它以 Future<T> 为输入并返回一个 Future<vector<T>>，这会在所有的 futures 完成后完成。（collect 的实现依赖于-你猜得到-then）有很多其他的[构建块](https://www.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md%23compositional-building-blocks&h=2AQF9GgoZ&s=1)，包括：collectAny、collectN、map 和 reduce。

请注意这个代码为什么会看上去与同步版本的 multiFooSync 非常相似，我们不需要担心上下文或线程安全的问题。这些问题都由框架解决，它们对我们而言是透明的。

[](https://my.oschina.net/Tsybius2014)

[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

2人顶

顶 翻译得不错哦！

**执行上下文**

其他一些语言里的 futures 框架提供了一个线程池用于执行回调函数，你除了要知道上下文在另外一个线程中执行，不需要关注任何多余的细节。但是 C++ 的开发者们倾向于编写 C++ 代码，因为他们需要控制底层细节来实现性能优化，Facebook 也不例外。因此我们使用简单的[Executor接口](https://www.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2FExecutor.h&h=vAQE6-D5K&s=1)提供了一个灵活的机制来明确控制回调上下文的执行：

struct Executor {

  using Func = std::function<void()>;

  virtual void add(Func) = 0;};

你可以向 then 函数传入一个 executor 来命令它的回调会通过 executor 执行。

a(input).then(executor, b);

在这段代码中，b 将会通过 executor 执行，b 可能是一个特定的线程、一个线程池、或是[一些更有趣的东西](https://www.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md%23executor-implementations&h=OAQGel9yi&s=1)。本方法的一个常见的用例是将 CPU 从 I/O 线程中解放出来，以避免队列中其他请求的排队时间。

[](https://my.oschina.net/Tsybius2014)

[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

1人顶

顶 翻译得不错哦！

**Futures 意味着你再也不用忘记说对不起**

传统的回调代码有一个普遍的问题，即不易对错误或异常情况的调用进行跟踪。程序员在检查错误和采取恰当措施上必须做到严于律己（即使是超人也要这样），更不要说当一场被意外抛出的情况了。Futures 使用包含一个值和一个异常的方式来解决这个问题，这些异常就像你希望的那样与 futures融合在了一起，除非它留在 future 单元里直到被 onErorr 接住，或是被同步地，例如，赋值或取值。这使得我们很难（但不是不可能）跟丢一个应该被接住的错误。

**使用 Promise**

我们已经大致看过了 futures 的使用方法，下面来说说我们该如何制作它们。如果你需要将一个值传入到 Future，使用 makeFuture：

using folly::makeFuture;

std::runtime\_error greatScott("Great Scott!");

Future<double> future = makeFuture(1.21e9);

Future<double> future = makeFuture<double>(greatScott);

但如果你要包装一个异步操作，你需要使用 Promise：

using folly::Promise;

Promise<double> promise;

Future<double> future = promise.getFuture();

当你准备好为 promise 赋值的时候，使用 setValue、setException 或是 setWith：

promise.setValue(1.21e9);

promise.setException(greatScott);

promise.setWith([]{

  if (year == 1955 || year == 1885) throw greatScott;

  return 1.21e9;

});

[](https://my.oschina.net/Tsybius2014)

[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

1人顶

顶 翻译得不错哦！

总之，我们通过生成另一个线程，将一个长期运行的同步操作转换为异步操作，如下面代码所示：

double getEnergySync(int year) {

  auto reactor = ReactorFactory::getReactor(year);

  if (!reactor) // It must be 1955 or 1885

    throw greatScott;

  return reactor->getGigawatts(1.21);

}

Future<double> getEnergy(int year) {

  auto promise = make\_shared<Promise<double>>();

  std::thread([=]{

    promise->setWith(std::bind(getEnergySync, year));

  }).detach();

  return promise->getFuture();

}

通常你不需要 promise，即使乍一看这像是你做的。举例来说，如果你的线程池中已经有了一个 executor 或是可以很轻易地获取它，那么这样做会更简单：

Future<double> future = folly::via(executor, std::bind(getEnergySync, year));

**用例学习**

我们提供了两个案例来解释如何在 Facebook 和 Instagram 中使用 future 来改善延迟、鲁棒性与代码的可读性。

Instagram 使用 futures 将他们推荐服务的基础结构由同步转换为异步，以此改善他们的系统。其结果是尾延迟（tail latency）得以显著下降，并仅用十分之一不到的服务器就实现了相同的吞吐量。他们把这些改动及相关改动带来的益处进行了记录，更多细节可以参考他们的[博客](http://l.facebook.com/l.php?u=http%3A%2F%2Finstagram-engineering.tumblr.com%2Fpost%2F121930298932%2Fc-futures-at-instagram&h=tAQEviG-6&s=1)。

[](https://my.oschina.net/Tsybius2014)

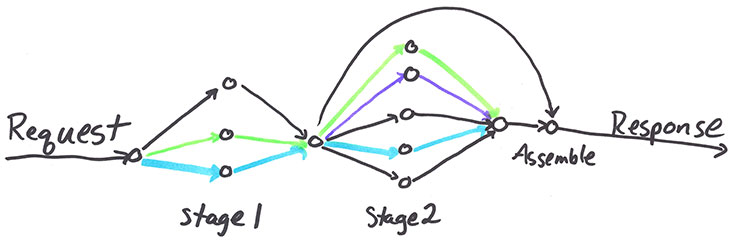
[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

1人顶

顶 翻译得不错哦！

下一个案例是一个真正的服务，它是 Facebook 新闻递送（News Feed）的一个组成部分。这个服务有一个两阶段的叶聚合模式（leaf-aggregate pattern），请求（request）会被分解成多个叶请求将碎片分配到不同的叶服务器，我们在做同样的事情，但根据第一次聚合的结果分配的碎片会变得不同。最终，我们获取两组结果集并将它们简化为一个单一的响应（response）。



下面是相关代码的简化版本：

Future<vector<LeafResponse>> fanout(

  const map<Leaf, LeafReq>& leafToReqMap,

  chrono::milliseconds timeout)

{

  vector<Future<LeafResponse>> leafFutures;

  for (const auto& kv : leafToReqMap) {

    const auto& leaf = kv.first;

    const auto& leafReq = kv.second;

    leafFutures.push\_back(

      // Get the client for this leaf and do the async RPC

      getClient(leaf)->futureLeafRPC(leafReq)

      // If the request times out, use an empty response and move on.

      .onTimeout(timeout, [=] { return LeafResponse(); })

      // If there's an error (e.g. RPC exception),

      // use an empty response and move on.

      .onError([=](const exception& e) { return LeafResponse(); }));

  }

  // Collect all the individual leaf requests into one Future

  return collect(leafFutures);

}

// Some sharding function; possibly dependent on previous responses.

map<Leaf, LeafReq> buildLeafToReqMap(

    const Request& request,

    const vector<LeafResponse>& responses);

// This function assembles our final response.

Response assembleResponse(

    const Request& request,

    const vector<LeafResponse>& firstFanoutResponses,

    const vector<LeafResponse>& secondFanoutResponses);

Future<Response> twoStageFanout(shared\_ptr<Request> request) {

  // Stage 1: first fanout

  return fanout(buildLeafToReqMap(\*request, {}),

                FIRST\_FANOUT\_TIMEOUT\_MS)

  // Stage 2: With the first fanout completed, initiate the second fanout.

  .then([=](vector<LeafResponse>& responses) {

    auto firstFanoutResponses =

      std::make\_shared<vector<LeafResponse>>(std::move(responses));

    // This time, sharding is dependent on the first fanout.

    return fanout(buildLeafToReqMap(\*request, \*firstFanoutResponses),

                  SECOND\_FANOUT\_TIMEOUT\_MS)

    // Stage 3: Assemble and return the final response.

    .then([=](const vector<LeafResponse>& secondFanoutResponses) {

      return assembleResponse(\*request, \*firstFanoutResponses, secondFanoutResponses);

    });

  });

}

该服务的历史版本中曾使用只允许整体超时的异步框架，同时使用了传统的“回调地狱”模式。是 Futures 让这个服务自然地表达了异步计算，并使用有粒度的超时以便在某些部分运行过慢时采取更积极的行动。其结果是，服务的平均延迟减少了三分之二，尾延迟减少到原来的十分之一，总体超时错误明显减少。代码变得更加易读和推理，作为结果，代码还变得更易维护。

[](https://my.oschina.net/Tsybius2014)

[北风其凉](https://my.oschina.net/Tsybius2014" \o "北风其凉)

翻译于 2年前

1人顶

顶 翻译得不错哦！

当开发人员拥有了帮助他们更好理解和表达异步操作的工具时，他们可以写出更易于维护的低延迟服务。

**结论**

Folly Futures 为 C++11 带来了健壮的、强大的、高性能的 futures。我们希望你会喜欢上它（就像我们一样）。如果你想了解更多信息，可以查阅相关[文档](https://www.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md&h=AAQF6oLih&s=1)、[文档块以及 GitHub 上的代码](https://github.com/facebook/folly/tree/master/folly/futures)。

**致谢**

Folly Futures 制作团队的成员包括 Hans Fugal，Dave Watson，James Sedgwick，Hannes Roth 和 Blake Mantheny，还有许多其他志同道合的贡献者。我们要感谢 Twitter，特别是 Marius，他在 Facebook 关于 Finagle 和 Futures 的技术讲座，激发了这个项目的创作灵感。