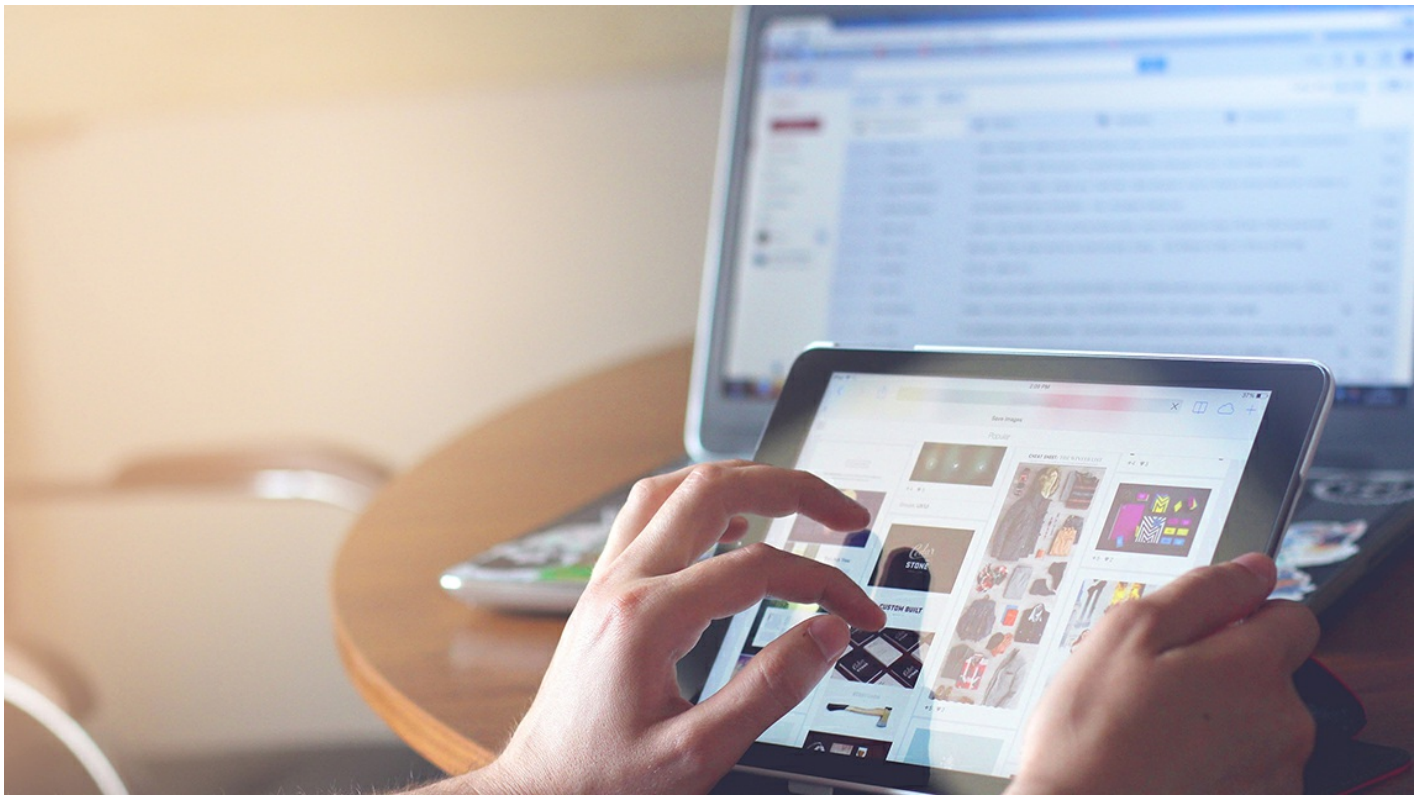
# 27讲C++RESTSDK：使用现代C++开发网络应用



你好，我是吴咏炜。

在实战篇，我们最后要讲解的一个库是 C++ REST SDK（也写作 cpprestsdk）[1]，一个支持 HTTP 协议 [2]、主要用于 RESTful [3] 接口开发的 C++ 库。

## 初识 C++ REST SDK

向你提一个问题，你认为用多少行代码可以写出一个类似于 curl [4] 的 HTTP 客户端？

使用 C++ REST SDK 的话，答案是，只需要五十多行有效代码（即使是适配到我们目前的窄小的手机屏幕上）。请看：

```
#include <iostream>
#ifdef _WIN32
#include <fcntl.h>
#include <io.h>
#endif
#include <cpprest/http_client.h>

using namespace utility;
using namespace web::http;
using namespace web::http::client;
using std::cerr;
using std::endl;

#ifdef _WIN32
#define tcout std::wcout
```

```
#else
#define tcout std::cout
#endif

auto get_headers(http_response resp)
{
  auto headers = resp.to_string();
  auto end =
    headers.find(U("\r\n\r\n"));
  if (end != string_t::npos) {
    headers.resize(end + 4);
  };
  return headers;
}

auto get_request(string_t uri)
{
  http_client client{uri};
  // 用 GET 方式发起一个客户端请求
  auto request =
    client.request(methods::GET)
      .then([](http_response resp) {
        if (resp.status_code() !=
            status_codes::OK) {
          // 不 OK, 显示当前响应信息
          auto headers =
            get_headers(resp);
          tcout << headers;
        }
        // 进一步取出完整响应
        return resp
          .extract_string();
      })
      .then([](string_t str) {
        // 输出到终端
        tcout << str;
      });
  return request;
}

#ifdef _WIN32
int wmain(int argc, wchar_t* argv[])
#else
int main(int argc, char* argv[])
```

```
int main(int argc, char* argv[])
#endif
{
#ifdef _WIN32
  _setmode(_fileno(stdout),
           _O_WTEXT);
#endif

  if (argc != 2) {
    cerr << "A URL is needed\n";
    return 1;
  }

  // 等待请求及其关联处理全部完成
  try {
    auto request =
      get_request(argv[1]);
    request.wait();
  }
  // 处理请求过程中产生的异常
  catch (const std::exception& e) {
    cerr << "Error exception: "
         << e.what() << endl;
    return 1;
  }
}
```

这个代码有点复杂，需要讲解一下：

- 第 14–18 行，我们根据平台来定义 `tcout`，确保多语言的文字能够正确输出。
- 第 20–29 行，我们定义了 `get_headers`，来从 `http_response` 中取出头部的字符串表示。
- 第 36 行，构造了一个客户端请求，并使用 `then` 方法串联了两个下一步的动作。`http_client::request` 的返回值是 `pplx::task<http_response>`。`then` 是 `pplx::task` 类模板的成员函数，参数是能接受其类型参数对象的函数对象。除了最后一个 `then` 块，其他每个 `then` 里都应该返回一个 `pplx::task`，而 `task` 的内部类型就是下一个 `then` 块里函数对象接受的参数的类型。
- 第 37 行开始，是第一段异步处理代码。参数类型是 `http_response`——因为`http_client::request` 的返回值是 `pplx::task<http_response>`。代码中判断如果响应的 HTTP 状态码不是 200 OK，就会显示响应头来帮助调试。然后，进一步取出所有的响应内容（可能需要进一步的异步处理，等待后续的 HTTP 响应到达）。
- 第 49 行开始，是第二段异步处理代码。参数类型是 `string_t`——因为上一段 `then` 块的返回值是 `pplx::task<string_t>`。代码中就是简单地把需要输出的内容输出到终端。
- 第 56–60 行，我们根据平台来定义合适的程序入口，确保命令行参数的正确处理。
- 第 62–65 行，在 Windows 上我们把标准输出设置成宽字符模式，来确保宽字符（串）能正确输出（参考 [第 11 讲]）。注意 `string_t` 在 Windows 上是 `wstring`，在其他平台上是 `string`。
- 第 72–83 行，如注释所言，产生 HTTP 请求、等待 HTTP 请求完成，并处理相关的异常。

整体而言，这个代码还是很简单的，虽然这种代码风格，对于之前没有接触过这种函数式编程风格的人来讲会有点奇怪——这被称作持续传递风格（continuation-passing style），显式地把上一段处理的结果传递到下一个函数中。这个代码已经处理了 Windows 环境和 Unix 环境的差异，底下是相当复杂的。

另外提醒一下，在 Windows 上如果你把源代码存成 UTF-8 的话，需要确保文件以 BOM 字符打头。Windows 的编辑器通常缺省就会做到；在 Vim 里，可以通过 `set bomb` 命令做到这一点。

### 安装和编译

上面的代码本身虽然简单，但要把它编译成可执行文件比我们之前讲的代码都要复杂——C++ REST SDK 有外部依赖，在 Windows 上和 Unix 上还不太一样。它的编译和安装也略复杂，如果你没有这方面的经验的话，建议尽量使用平台推荐的二进制包的安装方式。

由于其依赖较多，使用它的编译命令行也较为复杂。正式项目中绝对是需要使用项目管理软件的（如 cmake）。此处，我给出手工编译的典型命令行，仅供你尝试编译上面的例子作参考。

Windows MSVC：

```
cl /EHsc /std:c++17 test.cpp cpprest.lib zlib.lib libeay32.lib ssleay32.lib winhttp.lib
httpapi.lib bcrypt.lib crypt32.lib advapi32.lib gdi32.lib user32.lib
```

Linux GCC：

```
g++ -std=c++17 -pthread test.cpp -lcpprest -lcrypto -lssl -lboost_thread -lboost_chrono -
lboost_system
```

macOS Clang：

```
clang++ -std=c++17 test.cpp -lcpprest -lcrypto -lssl -lboost_thread-mt -lboost_chrono-mt
```

### 概述

有了初步印象之后，现在我们可以回过头看看 C++ REST SDK 到底是什么了。它是一套用来开发 HTTP 客户端和服务器的现代异步 C++ 代码库，支持以下特性（随平台不同会有所区别）：

- HTTP 客户端
- HTTP 服务器
- 任务
- JSON
- URI
- 异步流
- WebSocket 客户端
- OAuth 客户端

上面的例子里用到了 HTTP 客户端、任务和 URI（实际上是由 `string_t` 隐式构造了 `uri`），我们下面再介绍一下异步流、JSON 和 HTTP 服务器。

### 异步流

C++ REST SDK 里实现了一套异步流，能够实现对文件的异步读写。下面的例子展示了我们如何把网络请求的响应异步地存储到文件 results.html 中：

```
#include <iostream>
#include <utility>
#ifdef _WIN32
#include <fcntl.h>
#include <io.h>
#endif
#include <stddef.h>
#include <cpprest/http_client.h>
#include <cpprest/filestream.h>

using namespace utility;
using namespace web::http;
using namespace web::http::client;
using namespace concurrency::streams;
using std::cerr;
using std::endl;

#ifdef _WIN32
#define tcout std::wcout
#else
#define tcout std::cout
#endif

auto get_headers(http_response resp)
{
  auto headers = resp.to_string();
  auto end =
    headers.find(U("\r\n\r\n"));
  if (end != string_t::npos) {
    headers.resize(end + 4);
  };
  return headers;
}

auto get_request(string_t uri)
{
  http_client client{uri};
  // 用 GET 方式发起一个客户端请求
  auto request =
    client.request(methods::GET)
      .then([](http_response resp) {
        if (resp.status_code() ==
            status_codes::OK) {
```

```
          // 正常的话
          tcout << U("Saving...\n");
          ostream fs;
          fstream::open_ostream(
            U("results.html"),
            std::ios_base::out |
              std::ios_base::trunc)
            .then(
              [&fs,
               resp](ostream os) {
                fs = os;
                // 读取网页内容到流
                return resp.body()
                  .read_to_end(
                    fs.streambuf());
              })
            .then(
              [&fs](size_t size) {
                // 然后关闭流
                fs.close();
                tcout
                  << size
                  << U(" bytes "
                       "saved\n");
              })
            .wait();
        } else {
          // 否则显示当前响应信息
          auto headers =
            get_headers(resp);
          tcout << headers;
          tcout
            << resp.extract_string()
                 .get();
        }
      });
  return request;
}

#ifdef _WIN32
int wmain(int argc, wchar_t* argv[])
#else
int main(int argc, char* argv[])
#endif
```

```
{
#ifdef _WIN32
  _setmode(_fileno(stdout),
           _O_WTEXT);
#endif

  if (argc != 2) {
    cerr << "A URL is needed\n";
    return 1;
  }

  // 等待请求及其关联处理全部完成
  try {
    auto request =
      get_request(argv[1]);
    request.wait();
  }
  // 处理请求过程中产生的异常
  catch (const std::exception& e) {
    cerr << "Error exception: "
         << e.what() << endl;
  }
}
```

跟上一个例子比，我们去掉了原先的第二段处理统一输出的异步处理代码，但加入了一段嵌套的异步代码。有几个地方需要注意一下：

- C++ REST SDK 的对象基本都是基于 `shared_ptr` 用引用计数实现的，因而可以轻松大胆地进行复制。
- 虽然 `string_t` 在 Windows 上是 `wstring`，但文件流无论在哪个平台上都是以 UTF-8 的方式写入，符合目前的主流处理方式（`wofstream` 的行为跟平台和环境相关）。
- `extract_string` 的结果这次没有传递到下一段，而是直接用 `get` 获得了最终结果（类似于 [第 19 讲] 中的 `future`）。

这个例子的代码是基于 cpprestsdk 官方的例子改编的。但我做的下面这些更动值得提一下：

- 去除了不必要的 `shared_ptr` 的使用。
- `fstream::open_ostream` 缺省的文件打开方式是 `std::ios_base::out`，官方例子没有用 `std::ios_base::trunc`，导致不能清除文件中的原有内容。此处 C++ REST SDK 的 `file_stream` 行为跟标准 C++ 的 `ofstream` 是不一样的：后者缺省打开方式也是 `std::ios_base::out`，但此时文件内容**会**被自动清除。
- 沿用我的前一个例子，先进行请求再打开文件流，而不是先打开文件流再发送网络请求，符合实际流程。
- 这样做的一个结果就是 `then` 不完全是顺序的了，有嵌套，增加了复杂度，但展示了实际可能的情况。

### JSON 支持

在基于网页的开发中，JSON [5] 早已取代 XML 成了最主流的数据交换方式。REST 接口本身就是基于 JSON 的，自然，C++ REST SDK 需要对 JSON 有很好的支持。

JSON 本身可以在网上找到很多介绍的文章，我这儿就不多讲了。有几个 C++ 相关的关键点需要提一下：

- JSON 的基本类型是空值类型、布尔类型、数字类型和字符串类型。其中空值类型和数字类型在 C++ 里是没有直接对应物的。数字类型在 C++ 里可能映射到 `double`，也可能是 `int32_t` 或 `int64_t`。
- JSON 的复合类型是数组（array）和对象（object）。JSON 数组像 C++ 的 `vector`，但每个成员的类型可以是任意 JSON 类型，而不像 `vector` 通常是同质的——所有成员属于同一类型。JSON 对象像 C++ 的 `map`，键类型为 JSON 字符串，值类型则为任意 JSON 类型。JSON 标准不要求对象的各项之间有顺序，不过，从实际项目的角度，我个人觉得保持顺序还是非常有用的。

如果你去搜索“c++ json”的话，还是可以找到一些不同的 JSON 实现的。功能最完整、名声最响的目前似乎是 nlohmann/json [6]，而腾讯释出的 RapidJSON [7] 则以性能闻名 [8]。需要注意一下各个实现之间的区别：

- nlohmann/json 不支持对 JSON 的对象（object）保持赋值顺序；RapidJSON 保持赋值顺序；C++ REST SDK 可选保持赋值顺序（通过 `web::json::keep_object_element_order` 和 `web::json::value::object` 的参数）。
- nlohmann/json 支持最友好的初始化语法，可以使用初始化列表和 JSON 字面量；C++ REST SDK 只能逐项初始化，并且一般应显式调用 `web::json::value` 的构造函数（接受布尔类型和字符串类型的构造函数有 `explicit` 标注）；RapidJSON 介于中间，不支持初始化列表和字面量，但赋值可以直接进行。
- nlohmann/json 和 C++ REST SDK 支持直接在用方括号 `[]` 访问不存在的 JSON 数组（array）成员时改变数组的大小；RapidJSON 的接口不支持这种用法，要向 JSON 数组里添加成员要麻烦得多。
- 作为性能的代价，RapidJSON 里在初始化字符串值时，只会传递指针值；用户需要保证字符串在 JSON 值使用过程中的有效性。要复制字符串的话，接口要麻烦得多。
- RapidJSON 的 JSON 对象没有 `begin` 和 `end` 方法，因而无法使用标准的基于范围的 for 循环。总体而言，RapidJSON 的接口显得最特别、不通用。

如果你使用 C++ REST SDK 的其他功能，你当然也没有什么选择；否则，你可以考虑一下其他的 JSON 实现。下面，我们就只讨论 C++ REST SDK 里的 JSON 了。

在 C++ REST SDK 里，核心的类型是 `web::json::value`，这就对应到我前面说的“任意 JSON 类型”了。还是拿例子说话（改编自 RapidJSON 的例子）：

```
#include <iostream>
#include <string>
#include <utility>
#include <assert.h>
#ifdef _WIN32
#include <fcntl.h>
#include <io.h>
#endif
#include <cpprest/json.h>

using namespace std;
using namespace utility;
using namespace web;

#ifdef _WIN32
#define tcout std::wcout
```

```
#else
#define tcout std::cout
#endif

int main()
{
#ifdef _WIN32
  _setmode(_fileno(stdout),
           _O_WTEXT);
#endif

  // 测试的 JSON 字符串
  string_t json_str = U(R"(
    {
      "s": "你好，世界",
      "t": true,
      "f": false,
      "n": null,
      "i": 123,
      "d": 3.1416,
      "a": [1, 2, 3]
    })");
  tcout << "Original JSON:"
        << json_str << endl;

  // 保持元素顺序并分析 JSON 字符串
  json::keep_object_element_order(
    true);
  auto document =
    json::value::parse(json_str);

  // 遍历对象成员并输出类型
  static const char* type_names[] =
    {
      "Number", "Boolean", "String",
      "Object", "Array",   "Null",
    };
  for (auto&& value :
       document.as_object()) {
    tcout << "Type of member "
          << value.first << " is "
          << type_names[value.second
                        .type()]
          << endl;
```

```
        << endl;
}

// 检查 document 是对象
assert(document.is_object());

// 检查 document["s"] 是字符串
assert(document.has_field(U("s")));
assert(
  document[U("s")].is_string());
tcout << "s = "
      << document[U("s")] << endl;

// 检查 document["t"] 是字符串
assert(
  document[U("t")].is_boolean());
tcout
  << "t = "
  << (document[U("t")].as_bool()
        ? "true"
        : "false")
  << endl;

// 检查 document["f"] 是字符串
assert(
  document[U("f")].is_boolean());
tcout
  << "f = "
  << (document[U("f")].as_bool()
        ? "true"
        : "false")
  << endl;

// 检查 document["f"] 是空值
tcout
  << "n = "
  << (document[U("n")].is_null()
        ? "null"
        : "?")
  << endl;

// 检查 document["i"] 是整数
assert(
  document[U("i")].is_number());
```

```
assert(
  document[U("i")].is_integer());
tcout << "i = "
      << document[U("i")] << endl;

// 检查 document["d"] 是浮点数
assert(
  document[U("d")].is_number());
assert(
  document[U("d")].is_double());
tcout << "d = "
      << document[U("d")] << endl;

{
  // 检查 document["a"] 是数组
  auto& a = document[U("a")];
  assert(a.is_array());

  // 测试读取数组元素并转换成整数
  int y = a[0].as_integer();
  (void)y;

  // 遍历数组成员并输出
  tcout << "a = ";
  for (auto&& value :
       a.as_array()) {
    tcout << value << ' ';
  }
  tcout << endl;
}

// 修改 document["i"] 为长整数
{
  uint64_t bignum = 65000;
  bignum *= bignum;
  bignum *= bignum;
  document[U("i")] = bignum;

  assert(!document[U("i")]
            .as_number()
            .is_int32());
  assert(document[U("i")]
           .as_number()
           .to_uint64() ==
```

```
            bignum);
    tcout << "i is changed to "
          << document[U("i")]
          << endl;
  }

  // 在数组里添加数值
  {
    auto& a = document[U("a")];
    a[3] = 4;
    a[4] = 5;
    tcout << "a is changed to "
          << document[U("a")]
          << endl;
  }

  // 在 JSON 文档里添加布尔值：等号
  // 右侧 json::value 不能省
  document[U("b")] =
    json::value(true);

  // 构造新对象，保持多个值的顺序
  auto temp =
    json::value::object(true);
  // 在新对象里添加字符串：等号右侧
  // json::value 不能省
  temp[U("from")] =
    json::value(U("rapidjson"));
  temp[U("changed for")] =
    json::value(U("geekbang"));

  // 把对象赋到文档里；json::value
  // 内部使用 unique_ptr，因而使用
  // move 可以减少拷贝
  document[U("adapted")] =
    std::move(temp);

  // 完整输出目前的 JSON 对象
  tcout << document << endl;
}
```

例子里我加了不少注释，应当可以帮助你看清 JSON 对象的基本用法了。唯一遗憾的是宏 `U`（类似于 [第 11 讲] 里提到过的 `_T`）的使用有点碍眼：要确保代码在 Windows 下和 Unix 下都能工作，目前这还是必要的。

建议你测试一下这个例子。查看一下结果。

C++ REST SDK 里的 `http_request` 和 `http_response` 都对 JSON 有原生支持，如可以使用 `extract_json` 成员函数来异步提取 HTTP 请求或响应体中的 JSON 内容。

### HTTP 服务器

前面我们提到了如何使用 C++ REST SDK 来快速搭建一个 HTTP 客户端。同样，我们也可以使用 C++ REST SDK 来快速搭建一个 HTTP 服务器。在三种主流的操作系统上，C++ REST SDK 的 `http_listener` 会通过调用 Boost.Asio [9] 和操作系统的底层接口（IOCP、epoll 或 kqueue）来完成功能，向使用者隐藏这些细节、提供一个简单的编程接口。

我们将搭建一个最小的 REST 服务器，只能处理一个 sayHi 请求。客户端应当向服务器发送一个 HTTP 请求，URI 是：

> `/sayHi?name=…`

“…”部分代表一个名字，而服务器应当返回一个 JSON 的回复，形如：

```
{"msg": "Hi, …!"}
```

这个服务器的有效代码行同样只有六十多行，如下所示：

```
#include <exception>
#include <iostream>
#include <map>
#include <string>
#ifdef _WIN32
#include <fcntl.h>
#include <io.h>
#endif
#include <cpprest/http_listener.h>
#include <cpprest/json.h>

using namespace std;
using namespace utility;
using namespace web;
using namespace web::http;
using namespace web::http::
  experimental::listener;

#ifdef _WIN32
#define tcout std::wcout
#else
#define tcout std::cout
#endif

void handle_get(http_request req)
```

```
{
  auto& uri = req.request_uri();

  if (uri.path() != U("/sayHi")) {
    req.reply(
      status_codes::NotFound);
    return;
  }

  tcout << uri::decode(uri.query())
        << endl;

  auto query =
    uri::split_query(uri.query());
  auto it = query.find(U("name"));
  if (it == query.end()) {
    req.reply(
      status_codes::BadRequest,
      U("Missing query info"));
    return;
  }

  auto answer =
    json::value::object(true);
  answer[U("msg")] = json::value(
    string_t(U("Hi, ")) +
    uri::decode(it->second) +
    U("!"));

  req.reply(status_codes::OK,
            answer);
}

int main()
{
#ifdef _WIN32
  _setmode(_fileno(stdout),
           _O_WTEXT);
#endif

  http_listener listener(
    U("http://127.0.0.1:8008/"));
  listener.support(methods::GET,
                   handle_get);
```

```
  try {
    listener.open().wait();

    tcout << "Listening. Press "
             "ENTER to exit.\n";
    string line;
    getline(cin, line);

    listener.close().wait();
  }
  catch (const exception& e) {
    cerr << e.what() << endl;
    return 1;
  }
}
```

如果你熟悉 HTTP 协议的话，上面的代码应当是相当直白的。只有少数几个细节我需要说明一下：

- 我们调用 `http_request::reply` 的第二个参数是 `json::value` 类型，这会让 HTTP 的内容类型（Content-Type）自动置成“application/json”。
- `http_request::request_uri` 函数返回的是 `uri` 的引用，因此我用 `auto&` 来接收。`uri::split_query` 函数返回的是一个普通的 `std::map`，因此我用 `auto` 来接收。
- `http_listener::open` 和 `http_listener::close` 返回的是 `pplx::task<void>`；当这个任务完成时（`wait` 调用返回），表示 HTTP 监听器上的对应操作（打开或关闭）真正完成了。

运行程序，然后在另外一个终端里使用我们的第一个例子生成的可执行文件（或 curl）：

```
curl "http://127.0.0.1:8008/sayHi?name=Peter"
```

我们就应该会得到正确的结果：

```
{"msg":"Hi, Peter!"}
```

你也可以尝试把路径和参数写错，查看一下程序对出错的处理。

### 关于线程的细节

C++ REST SDK 使用异步的编程模式，使得写不阻塞的代码变得相当容易。不过，底层它是使用一个线程池来实现的——在 C++20 的协程能被使用之前，并没有什么更理想的跨平台方式可用。

C++ REST SDK 缺省会开启 40 个线程。在目前的实现里，如果这些线程全部被用完了，会导致系统整体阻塞。反过来，如果你只是用 C++ REST SDK 的 HTTP 客户端，你就不需要这么多线程。这个线程数量目前在代码里是可以控制的。比如，下面的代码会把线程池的大小设为 10：

```
#include <pplx/threadpool.h>
…
crossplat::threadpool::
  initialize_with_threads(10);
```

如果你使用 C++ REST SDK 开发一个服务器，则不仅应当增加线程池的大小，还应当对并发数量进行统计，在并发数接近线程数时主动拒绝新的连接——一般可返回 `status_codes::ServiceUnavailable`——以免造成整个系统的阻塞。

### 内容小结

今天我们对 C++ REST SDK 的主要功能作了一下概要的讲解和演示，让你了解了它的主要功能和这种异步的编程方式。还有很多功能没有讲，但你应该可以通过查文档了解如何使用了。

这只能算是我们旅程中的一站——因为随着 C++20 的到来，我相信一定会有更多好用的网络开发库出现的。

### 课后思考

作为实战篇的最后一讲，内容还是略有点复杂的。如果你一下子消化不了，可以复习前面的相关内容。

如果对这讲的内容本身没有问题，则可以考虑一下，你觉得 C++ REST SDK 的接口好用吗？如果好用，原因是什么？如果不好用，你有什么样的改进意见？

**参考资料**


[1] Microsoft, cpprestsdk. https://github.com/microsoft/cpprestsdk

[2] Wikipedia, “Hypertext Transfer Protocol”. https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[2a] 维基百科, “超文本传输协议”. https://zh.m.wikipedia.org/zh-hans/超文本传输协议

[3] RESTful. https://restfulapi.net/

[4] curl. https://curl.haxx.se/

[5] JSON. https://www.json.org/

[6] Niels Lohmann, json. https://github.com/nlohmann/json

[7] Tencent, rapidjson. https://github.com/Tencent/rapidjson

[8] Milo Yip, nativejson-benchmark. https://github.com/miloyip/nativejson-benchmark

[9] Christopher Kohlhoff, Boost.Asio. https://www.boost.org/doc/libs/release/doc/html/boost_asio.html


精选留言



幻境之桥

吴老师要是可以把所有的内容都使用 cmake 来构建，以后参考起来可以更方便啊

2020-02-03 20:33

作者回复

这要花时间的……用 CMake 的来顶这个评论，人多我就考虑一下。

更新：我听到大家的声音了。CMake 用户的福利在这里：

https://github.com/adah1972/geek_time_cpp

有空时我会继续更新。

2020-02-04 10:29

心情难以平静

coroutine现在有好多轮子。希望标准实现快点到来。据说里面的坑较多，希望有人能先帮忙踩一踩。

2020-02-03 14:56

作者回复

（C++20）协程是第 30 讲的内容。在 MSVC 和 Clang 已经基本可用了。坑，总得用了之后才知道有多少的……

2020-02-03 19:09

申学晋

只用过cpprest开发客户端，在windows下字符串处理还是有点麻烦。最近想开发WebSocket服务器，不知道是否能用?

2020-02-05 10:29

作者回复

还行吧，坑我也标出来了。全部用 wcout（tcout）一般就可以用。

WebSocket支持我没用过，给不了建议。

2020-02-05 11:46

莫珣

cpprest这个库在linux下编译起来真是太麻烦了，今天折腾了大半天竟然没有编译出来。github上给出的编译步骤过于简单。

2020-02-04 19:25

作者回复

遇到特定困难可以网上搜一下。我印象里，依赖装好之后，Linux上编译还是不麻烦的。你不是因为 GCC 版本太低吧?

实际上，这虽然是微软出的，我觉得还是在 Windows 上编译更麻烦呢……

2020-02-05 11:49

hb

支持https吗

2020-02-03 23:25

作者回复

支持。客户端直接用。服务器端麻烦点，根据平台不同有不同的配置方法。

2020-02-04 10:25

tt

我觉得纯从网路编程上来说，比起直接用EPOLL，然后加上一堆线程、队列、锁、条件变量啥的方便多了，隐藏了事件和循环，还是方便多了。

和JAVASCRIPT中的PROMISE已经非常像了。

2020-02-03 22:31

作者回复

那是肯定的，否则我介绍它干嘛……

但话说回来，如果你的服务器性能要求非常高，这个方案就不一定适合了。一个并发连接目前还是需要一个线程的。

2020-02-04 10:36