

16 | 网络通信：我不想写原生Socket

2020-06-11 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 15:26 大小 14.15M



你好，我是 Chrono。

在上一节课，我讲了 JSON、MessagePack 和 ProtoBuffer 这三种数据交换格式。现在，我们手里有了这些跨语言、跨平台的通用数据，该怎么与外部通信交换呢？

你肯定首先想到的就是 Socket 网络编程，使用 TCP/IP 协议栈收发数据，这样不仅可以在本地的进程间通信，也可以在主机、机房之间异地通信。

大方向上这是没错的，但你也肯定知道，原生的 Socket API 非常底层，要考虑很多细节，比如 TIME_WAIT、CLOSE_WAIT、REUSEADDR 等，如果再加上异步就更复杂了。



虽然你可能看过、学过不少这方面的资料，对如何处理这些问题“胸有成竹”，但无论如何，像 Socket 建连 / 断连、协议格式解析、网络参数调整等，都要自己动手做，想要“凭空”写出一个健壮可靠的网络应用程序还是相当麻烦的。

所以，今天我就来谈谈 C++ 里的几个好用的网络通信库：libcurl、cpr 和 ZMQ，让你摆脱使用原生 Socket 编程的烦恼。

libcurl：高可移植、功能丰富的通信库

第一个要说的库是 libcurl，它来源于著名的 [curl 项目](#)，也是 curl 的底层核心。

libcurl 经过了多年的开发和实际项目的验证，非常稳定可靠，拥有上百万的用户，其中不乏 Apple、Facebook、Google、Netflix 等大公司。

它最早只支持 HTTP 协议，但现在已经扩展到支持所有的应用层协议，比如 HTTPS、FTP、LDAP、SMTP 等，功能强大。

libcurl 使用纯 C 语言开发，兼容性、可移植性非常好，基于 C 接口可以很容易写出各种语言的封装，所以 Python、PHP 等语言都有 libcurl 相关的库。

因为 C++ 兼容 C，所以我們也可以在 C++ 程序里直接调用 libcurl 来收发数据。

在使用 libcurl 之前，你需要用 apt-get 或者 yum 等工具安装开发库：

```
1 apt-get install libcurl4-openssl-dev
```

 复制代码

虽然 libcurl 支持很多协议，但最常用的还是 HTTP。所以接下来，我也主要介绍 libcurl 的 HTTP 使用方法，这样对其他的协议你也可以做到“触类旁通”。

libcurl 的接口可以粗略地分成两大类：easy 系列和 multi 系列。其中，easy 系列是同步调用，比较简单；multi 系列是异步的多线程调用，比较复杂。通常情况下，我们用 easy 系列就足够了。

使用 libcurl 收发 HTTP 数据的基本步骤有 4 个：

1. 使用 `curl_easy_init()` 创建一个句柄，类型是 `CURL*`。但我们完全没有必要关心句柄的类型，直接用 `auto` 推导就行。
2. 使用 `curl_easy_setopt()` 设置请求的各种参数，比如请求方法、URL、header/body 数据、超时、回调函数等。这是最关键的操作。
3. 使用 `curl_easy_perform()` 发送数据，返回的数据会由回调函数处理。
4. 使用 `curl_easy_cleanup()` 清理句柄相关的资源，结束会话。

下面我用个简短的例子来示范一下这 4 步：

 复制代码

```
1 #include <curl/curl.h>           // 包含头文件
2
3 auto curl = curl_easy_init();     // 创建CURL句柄
4 assert(curl);
5
6 curl_easy_setopt(curl, CURLOPT_URL, "http://nginx.org"); // 设置请求URI
7
8 auto res = curl_easy_perform(curl); // 发送数据
9 if (res != CURLE_OK) {           // 检查是否执行成功
10     cout << curl_easy_strerror(res) << endl;
11 }
12
13 curl_easy_cleanup(curl);         // 清理句柄相关的资源
```

这段代码非常简单，重点是调用 `curl_easy_setopt()` 设置了 URL，请求 Nginx 官网的首页，其他的都使用默认值即可。


由于没有设置你自己的回调函数，所以 libcurl 会使用内部的默认回调，把得到的 HTTP 响应数据输出到标准流，也就是直接打印到屏幕上。

这个处理结果显然不是我们所期待的，所以如果想要自己处理返回的 HTTP 报文，就得写一个回调函数，在里面实现业务逻辑。

因为 libcurl 是 C 语言实现的，所以回调函数必须是函数指针。不过，C++11 允许你写 lambda 表达式，这利用了一个特别规定：**无捕获的 lambda 表达式可以显式转换成一个**

函数指针。注意一定要是“无捕获”，也就是说 lambda 引出符“[]”必须是空的，不能捕获任何外部变量。

所以，只要多做一个简单的转型动作，你就可以用 lambda 表达式直接写 libcurl 的回调，还是熟悉的函数式编程风格：

 复制代码

```
1 // 回调函数的原型
2 size_t write_callback(char* , size_t , size_t , void* );
3
4 curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION,          // 设置回调函数
5   (decltype(&write_callback))          // decltype获取函数指针类型，显式转换
6   [](char *ptr, size_t size, size_t nmemb, void *userdata)// lambda
7   {
8       cout << "size = " << size * nmemb << endl;    // 简单的处理
9       return size * nmemb;                          // 返回接收的字节数
10  }
11 );
```


libcurl 的用法大概就是这个样子了，开头的准备和结尾的清理工作都很简单，关键的就是 curl_easy_setopt() 这一步的参数设置。我们必须通过查文档知道该用哪些标志宏，写一些单调重复的代码。

你可能想到了，可以自己用 C++ 包装出一个类，就能够少敲点键盘。但不要着急，因为我们有一个更好的选择，就是 cpr。

cpr：更现代、更易用的通信库


cpr 是对 libcurl 的一个 C++11 封装，使用了很多现代 C++ 的高级特性，对外的接口模仿了 Python 的 requests 库，非常简单易用。

你可以从 [GitHub](#) 上获取 cpr 的源码，再用 cmake 编译安装：

 复制代码

```
1 git clone git@github.com:whoshuu/cpr.git
2 cmake . -DUSE_SYSTEM_CURL=ON -DBUILD_CPR_TESTS=OFF
3 make && make install
```

和 libcurl 相比，cpr 用起来真的是太轻松了，不需要考虑什么初始化、设置参数、清理等杂事，一句话就能发送 HTTP 请求：

 复制代码

```
1 #include <cpr/cpr.h>                                // 包含头文件
2
3 auto res = cpr::Get(                                  // GET请求
4     cpr::Url{"http://openresty.org"}                  // 传递URL
5 );
```

你也不用写回调函数，HTTP 响应就是函数的返回值，用成员变量 url、header、status_code、text 就能够得到报文的各个组成部分：

 复制代码

```
1 cout << res.elapsed << endl;                        // 请求耗费的时间
2
3 cout << res.url << endl;                              // 请求的URL
4 cout << res.status_code << endl;                      // 响应的状态码
5 cout << res.text.length() << endl;                    // 响应的body数据
6
7 for(auto& x : res.header) {                            // 响应的头字段
8     cout << x.first << "=>"                          // 类似map的结构
9         << x.second << endl;
10 }
```

在 cpr 里，HTTP 协议的概念都被实现为相应的函数或者类，内部再转化为 libcurl 操作，主要的有：

GET/HEAD/POST 等请求方法，使用同名的 Get/Head/Post 函数；

URL 使用 Url 类，它其实是 string 的别名；

URL 参数使用 Parameters 类，KV 结构，近似 map；


请求头字段使用 Header 类，它其实是 map 的别名，使用定制的函数实现了大小写无关比较；

Cookie 使用 Cookies 类，也是 KV 结构，近似 map；

请求体使用 Body 类；

超时设置使用 Timeout 类。


这些函数和类的用法都非常自然、符合思维习惯，而且因为可以使用 C++11 的花括号 “{}” 初始化语法，如果你以前用过 Python requests 库的话一定会感到很亲切：

 复制代码

```
1  const auto url = "http://openresty.org"s; // 访问的URL
2
3  auto res1 = cpr::Head(                      // 发送HEAD请求
4              cpr::Url{url}                   // 传递URL
5  );
6
7  auto res2 = cpr::Get(                      // 发送GET请求
8              cpr::Url{url},                  // 传递URL
9              cpr::Parameters{                // 传递URL参数
10                 {"a", "1"}, {"b", "2"}
11            };
12
13  auto res3 = cpr::Post(                     // 发送POST请求
14              cpr::Url{url},                 // 传递URL
15              cpr::Header{                   // 定制请求头字段
16                 {"x", "xxx"}, {"expect", ""}},
17              cpr::Body{"post data"},        // 传递body数据
18              cpr::Timeout{200ms}           // 超时时间
19  );
```

cpr 也支持异步处理，但它内部没有使用 libcurl 的 multi 接口，而是使用了标准库里的 future 和 async（参见[第 14 讲](#)），和 libcurl 的实现相比，既简单又好理解。

异步接口与同步接口的调用方式基本一样，只是名字多了个 “Async” 的后缀，返回的是一个 future 对象。你可以调用 wait() 或者 get() 来获取响应结果：

 复制代码

```
1  auto f = cpr::GetAsync(                    // 异步发送GET请求
2              cpr::Url{"http://openresty.org"}
3  );
4
5  auto res = f.get();                        // 等待响应结果
6  cout << res.elapsed << endl;             // 请求耗费的时间
```

看了上面这些介绍，你是不是有些心动了。说实话，我原来在 C++ 里也是一直用 libcurl，也写过自己的包装类，直到发现了 cpr 这个 “大杀器”，就立即 “弃暗投明” 了。

相信有了 cpr，你今后在 C++ 里写 HTTP 应用就不再是痛苦，而是一种享受了。

ZMQ：高效、快速、多功能的通信库

libcurl 和 cpr 处理的都是 HTTP 协议，虽然用起来很方便，但协议自身也有一些限制，比如必须要一来一回，必须点对点直连，在超大数据量通信的时候就不是太合适。

还有一点，libcurl 和 cpr 只能充当 HTTP 的客户端，如果你想写服务器端程序，这两个工具就完全派不上用场。

所以，我们就需要一个更底层、更灵活的网络通信工具，它应该能够弥补 libcurl 和 cpr 的不足，不仅快速高效，还能同时支持客户端和服务端编程。

这就是我要说的第三个库：[🔗ZMQ](#)。

其实，ZMQ 不仅是一个单纯的网络通信库，更像是一个高级的异步并发框架。

从名字上就可以看出来，Zero Message Queue——零延迟的消息队列，意味着它除了可以收发数据外，还可以用作消息中间件，解耦多个应用服务之间的强依赖关系，搭建高效、有弹性的分布式系统，从而超越原生的 Socket。

作为消息队列，ZMQ 的另一大特点是零配置零维护零成本，不需要搭建额外的代理服务，只要安装了开发库就能够直接使用，相当于把消息队列功能直接嵌入到你的应用程序里：

```
1 apt-get install libzmq3-dev
```

 复制代码

ZMQ 是用 C++ 开发的，但出于兼容的考虑，对外提供的是纯 C 接口。不过它也有很多 C++ 封装，这里我选择的是自带的 [🔗cppzmq](#)，虽然比较简单，但也基本够用了。

由于 ZMQ 把自身定位于更高层次的“异步消息队列”，所以它的用法就不像 Socket、HTTP 那么简单直白，而是定义了 5 种不同的工作模式，来适应实际中常见的网络通信场景。

我来大概说一下这 5 种模式：

原生模式（RAW），没有消息队列功能，相当于底层 Socket 的简单封装；

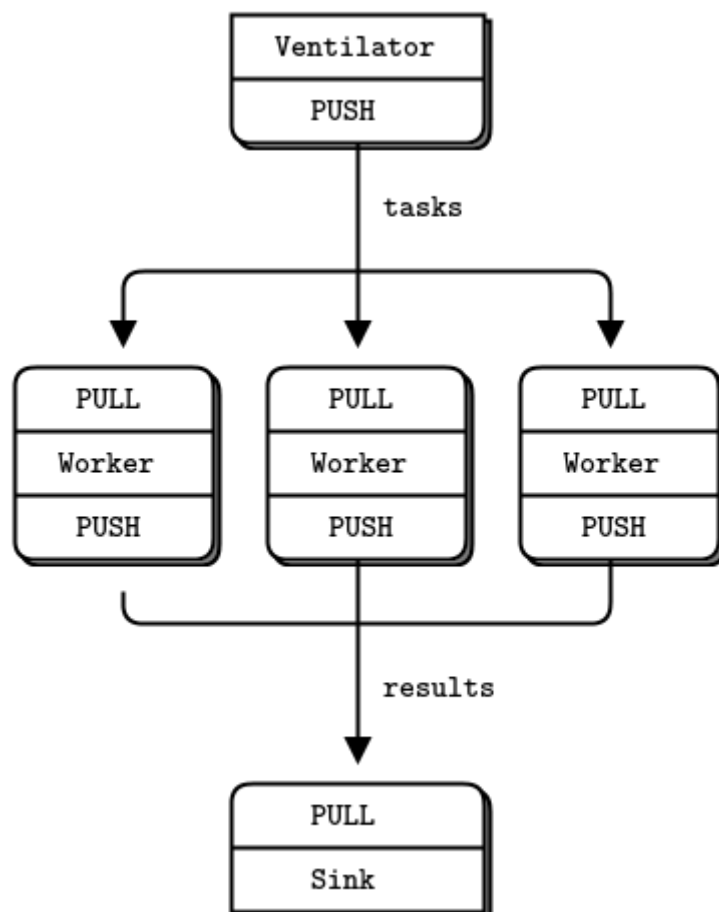
结对模式（PAIR），两个端点一对一通信；

请求响应模式（REQ-REP），也是两个端点一对一通信，但请求必须有响应；

发布订阅模式（PUB-SUB），一对多通信，一个端点发布消息，多个端点接收处理；

管道模式（PUSH-PULL），或者叫流水线，可以一对多，也可以多对一。

前四种模式类似 HTTP 协议、Client-Server 架构，很简单，就不多说了。我拿我在工作中比较常用的管道模式来给你示范一下 ZMQ 的用法，它非常适合进程间无阻塞传送海量数据，也有点 map-reduce 的意思。




在 ZMQ 里有两个基本的类。

第一个是 `context_t`，它是 ZMQ 的运行环境。使用 ZMQ 的任何功能前，必须要先创建它。

第二个是 `socket_t`，表示 ZMQ 的套接字，需要指定刚才说的那 5 种工作模式。注意它与原生 `Socket` 没有任何关系，只是借用了名字来方便理解。

下面的代码声明了一个全局的 ZMQ 环境变量，并定义了一个 lambda 表达式，生产 ZMQ 套接字：

 复制代码

```
1  const auto thread_num = 1;           // 并发线程数
2
3  zmq::context_t context(thread_num);    // ZMQ环境变量
4
5  auto make_sock = [&](auto mode)       // 定义一个lambda表达式
6  {
7      return zmq::socket_t(context, mode); // 创建ZMQ套接字
8  };
9
```


和原生 `Socket` 一样，ZMQ 套接字也必须关联到一个确定的地址才能收发数据，但它不仅支持 TCP/IP，还支持进程内和进程间通信，这在本机交换数据时会更高效：

TCP 通信地址的形式是 “tcp://...” ，指定 IP 地址和端口号；

进程内通信地址的形式是 “inproc://...” ，指定一个本地可访问的路径；

进程间通信地址的形式是 “ipc://...” ，也是一个本地可访问的路径。

用 `bind()/connect()` 这两个函数把 ZMQ 套接字连接起来之后，就可以用 `send()/recv()` 来收发数据了，看一下示例代码吧：

 复制代码

```
1  const auto addr = "ipc:///dev/shm/zmq.sock"s; // 通信地址
2
3  auto receiver = [=]()                       // lambda表达式接收数据
4  {
5      auto sock = make_sock(ZMQ_PULL);        // 创建ZMQ套接字，拉数据
6
7      sock.bind(addr);                        // 绑定套接字
8      assert(sock.connected());
9
10     zmq::message_t msg;
11     sock.recv(&msg);                         // 接收消息
12
```

```

13     string s = {msg.data<char>(), msg.size()};
14     cout << s << endl;
15 };
16
17 auto sender = [=]()                // lambda表达式发送数据
18 {
19     auto sock = make_sock(ZMQ_PUSH);    // 创建ZMQ套接字，推数据
20
21     sock.connect(addr);                // 连接到对端
22     assert(sock.connected());
23
24     string s = "hello zmq";
25     sock.send(s.data(), s.size());    // 发送消息
26 }

```

这段代码实现了两个最基本的客户端和服务端，看起来好像没什么特别的。但你应该注意到，使用 ZMQ 完全不需要考虑底层的 TCP/IP 通信细节，它会保证消息异步、安全、完整地到达服务器，让你关注网络通信之上更有价值的业务逻辑。

ZMQ 的用法就是这么简单，但想要进一步发掘它的潜力，处理大流量的数据还是要去看 [它的文档](#)，选择合适的工作模式，再仔细调节各种参数。

接下来，我再给你分享两个实际工作中会比较有用的细节吧。

一个是 **ZMQ 环境的线程数**。它的默认值是 1，太小了，适当增大一些就可以提高 ZMQ 的并发处理能力。我一般用的是 4~6，具体设置为多少最好还是通过性能测试来验证下。

另一个是**收发消息时的本地缓存数量**，ZMQ 的术语叫 High Water Mark。如果收发的数据过多，数量超过 HWM，ZMQ 要么阻塞，要么丢弃消息。

HWM 需要调用套接字的成员函数 `setsockopt()` 来设置，注意收发使用的是两个不同的标志：

```

1 sock.setsockopt(ZMQ_RCVHWM, 1000);    // 接收消息最多缓存1000条
2 sock.setsockopt(ZMQ_SNDHWM, 100);    // 发送消息最多缓存100条

```

 复制代码

我们把 HWM 设置成多大都可以，比如我就曾经在一个高并发系统里用过 100 万以上的值，不用担心，ZMQ 会把一切都处理得很好。

关于 ZMQ 就暂时说到这里，它还有很多强大的功能，你可以阅读 [官网](#) 上的教程和指南，里面非常详细地讨论了 ZMQ 的各种模式和要点。

小结

好了，我来给今天的内容做一个小结：

1. libcurl 是一个功能完善、稳定可靠的应用层通信库，最常用的就是 HTTP 协议；
2. cpr 是对 libcurl 的 C++ 封装，接口简单易用；
3. libcurl 和 cpr 都只能作为客户端来使用，不能编写服务器端应用；
4. ZMQ 是一个高级的网络通信库，支持多种通信模式，可以把消息队列功能直接嵌入应用程序，搭建出高效、灵活、免管理的分布式系统。

最后，再说说即将到来的 C++20，原本预计会加入期待已久的 networking 库，但现在已经被推迟到了下一个版本（C++23）。

networking 库基于已有多年实践的 boost.asio，采用前摄器模式（Proactor）统一封装了操作系统的各种异步机制（epoll、kqueue、IOCP），而且支持协程。有了它，我们的网络通信工作就会更加轻松。

课下作业

最后是课下作业时间，给你留两个思考题：

1. 你在网络编程的时候都遇到过哪些“坑”，今天说的这几个库能否解决你的问题？
2. 你觉得 ZMQ 能够在多大程度上代替原生 Socket？

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友。我们下节课见。

课外小贴士

1. libcurl附带了一个非常有用的工具：curl-config，可以用它来查看libcurl的基本信息，比如版本号、编译选项、链接选项等等。我建议在正式开发前最好先用它确认一下相关的信息。
2. 个人认为cpr的Body类设计得不是很好，其他的类都是把标准容器作为成员，以组合的方式使用，而它却直接从string继承。
3. Beast是Boost程序库提供的一个高级网络通信库。它基于asio，简单易用，完全异步无阻塞，支持HTTP和WebSocket。
4. ZMQ的原作者后来又开发出了一个新的网络通信库nanomq，改进了一些ZMQ设计的不足，但不知道为什么，应用得不是很广。

更多课程推荐

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



涨价倒计时 🕒

今日秒杀 **¥79**，6月13日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 序列化：简单通用的数据交换格式有哪些？

下一篇 17 | 脚本语言：搭建高性能的混合系统

精选留言 (16)

写留言



赖阿甘

2020-06-11

说实在在看文章之前，我还不知道ZMQ这个网络通信库，哈哈。老师怎么没提到libevent、libev、asio、muduo等网络通信库，是否这些库的接口比较原始，不好用，还望解答

作者回复: 你说的很对，这些库用起来比较复杂，一时半会说不清楚，光里面的epoll、多线程机制就要解释半天，要用起来更是要很多步骤。

我的想法还是让大家尽量快速上手，能用起来最重要。



6



robonix

2020-06-11

疑惑同楼上 ~ ASIO 据说要加到c++ 20了, 应该介绍呀?

作者回复: asio推迟到了C++23, 而且从它的前身boost.asio来看, 用起来也是很复杂的, 里面的概念很多, 不好一下子说清楚, 以后如果有机会可以专门写一篇。



👍 2



编程国学

2020-06-11

军工行业, 用户强调实时, 一般采用udp, 目前我们采用qt 自己的库, 有没有好的建议

作者回复: qt我没用过, 不好评价, 我觉得可以试试zmq, 它的性能也很不错。



👍 2



屈肖东

2020-06-15

老师什么时候可以推荐几个值得读的比较通用的开源代码, 因为很多开源代码虽然很好, 但是太过复杂庞大, 很难阅读。或者写一篇针对如何更好的阅读源码的文章。毕竟读代码应该是学习写代码最好的方式

作者回复: C++开源库很多都比较大、比较复杂, 找合适学习的还真不是件容易的事情。

我觉得比较好的开始方式是看课程里推荐的这几个库, 都不是特别大, 功能比较单一, 比较容易聚焦学习点。

看的时候可以先从代码风格看起, 再熟悉C++关键字的用法, 再到整体架构、接口设计。不能心急, 不要想着几天或者一个月就能看完。而且也没必要完全看懂, 只要能从中学到一两个点就可以说是值得了。



👍 1



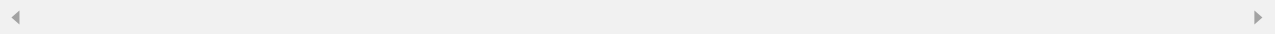
黄骏

2020-06-11

之前用过zmq, 他的创始人也非常传奇

展开 ▾

作者回复: zmq算是老牌的消息中间件了, 用的比较广泛, 不过确实有点老。



1



lckfa李钊

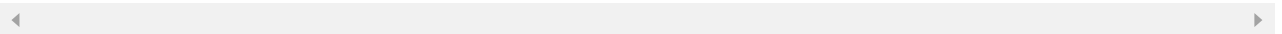
2020-06-11

涨知识了, 之前http通信一直使用的原生 libcurl, 看起来现在可以转cpr了, 不是不知道这个库的存在, 只是没人推荐不敢换而已。另外ZMQ, 这个库看起来可以解决手写TCP/IP通信的问题, 后面会试试。

网络编程最大的坑其实是不懂网络协议, 以及联调的问题, 感觉不是库能解决的。但是库能解决底层封装, 避免重复造轮子的痛苦。

展开 ∨

作者回复: 对, 这些通讯库可以减轻底层的一部分工作, 把时间和精力集中在上层应用上。



1



范闲

2020-06-15

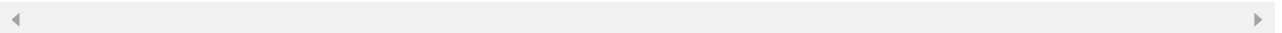
zmq是可以做成服务端的是吧? 那这个除了易用性, 和其他网络库相比还有啥优势呢? 另外cpp的rpc框架其实也都提供了这些功能, 是否用rpc更合理?

展开 ∨

作者回复:

1.zmq是消息中间件, 可以保证消息不丢失, 准确送达。

2.rpc适用于开发请求-响应的场景, 是远程调用, 而zmq是更底层一些的网络传输库, 所以应用范围就更广。



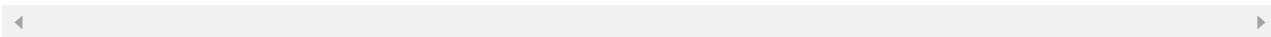
朱熙

2020-06-13

老师您好, 有一个caf的框架, 希望可以在之后的课程里有机会能分析下优劣, 是一个多线程的框架, 包括线程间的通信

展开 ∨

作者回复: 大概看了一下, 挺有意思的, 可惜要求gcc>7, 这在很多老系统上满足不了要求。

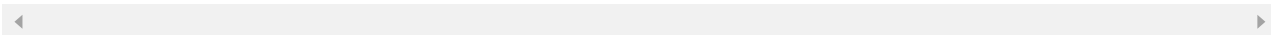


王珏大大

2020-06-12

请问老师，“ZMQ 环境的线程数” 这是什么设置吗，如何设置？

作者回复: 这个是zmq内部的工作线程数量，开多个线程可以增加zmq的处理能力。



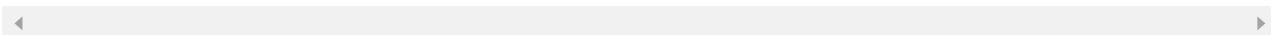
java2c++

2020-06-12

```
cpr::Url{"http://openresty.org"} // 传递URL
);
URL 使用 Url 类，它其实是 string 的别名；
```

这个地方的语法没有看懂，Url后面跟个大括号 {} 表示什么意思，是Url的匿名子类吗
展开 ▾

作者回复: 花括号是C++11的新初始化语法，和圆括号的构造函数类似，用这种形式看起来更清楚一些，也和Python比较像。

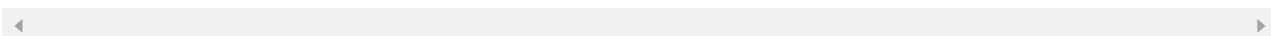


幻境之桥

2020-06-12

这几个库都很 nice 🍷
cpp 有类似 JAVA Netty 这样的框架吗？
或者上面这些有可以方便实现自定义协议的吗？
展开 ▾

作者回复: 很遗憾，目前C++还没有非常完善易用的高级框架，rpc框架倒是有一些。



土土人

2020-06-12

ASIO使用起来极其别扭

展开 ▾

作者回复: 可能是因为它用的是proactor吧, 但设计架构还是挺好的, 作为底层比较合适, 如果有高级封装库就好了。



木瓜777

2020-06-12

zmp是否支持websocket?

展开 ▾

作者回复: 不支持, 它用的是自己的协议, 不走标准。



TC128

2020-06-11

HP-Socket有人用吗?

展开 ▾

作者回复: 我没用过, 简单搜了一下, 感觉还不错, 可以一试。



Confidant.

2020-06-11

老师, zmq的代码有一个lamda用&捕获的错误, 你的都可以编过吗?

作者回复: GitHub上的代码都用g++5.4编译通过的, 注意用std=C++14, 可以看一下源码开头的注释。

如果还不行, 可以在GitHub上提issue。



Jason

前段时间遇到过一个问题，通过http发送图片数据老是发送不全，最终定位到有两个原因：一是因为Linux 系统的TCP 缓冲区太小，通过/etc/sysctl.conf设置解决；二是因为代码中socket的SO_SNDBUF参数不够大，通过setsockopt函数重新设置解决。如果用老师说的这三个库来开发的话，不知道会不会遇到上面的问题？

展开 ∨

作者回复：

1.系统的问题，libcurl肯定解决不了，得外部调整。

2.可以用函数Curl_sndbufset()调整，源码在lib/connect.c。不过我看它只在Windows上生效，Linux下是空的，所以在linux上应该不需要用这个。可以用libcurl试验一下，我还没有遇到过你这样的问题。

3.zmq不用担心这个问题，它属于mq，会自动处理大数据。

