基于C++20 协程的高性能rpc 库coro_rpc

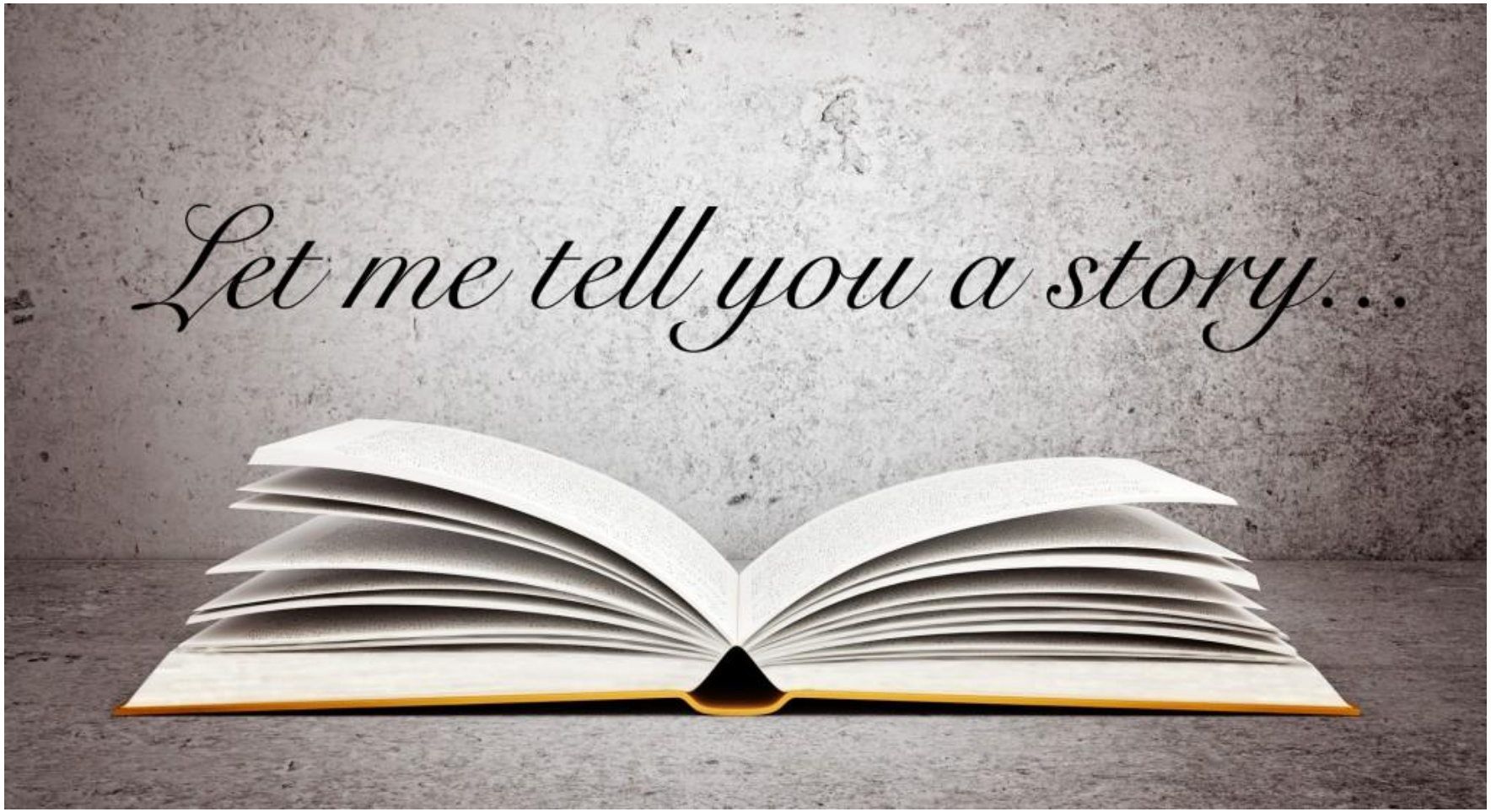
祁宇

purecpp@163.com

主要内容

- coro_rpc 的故事
- coro_rpc example
- coro_rpc benchmark
- coro_rpc 性能优化实践
- 踩过的坑

coro_rpc的故事



为什么一些广泛使用的c++ rpc库都那么难用？！

- Grpc hello world（more than 100 lines code）

  https://github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_server.cc

  https://github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_callback_client.cc

python的rpc

```python
from xmlrpc.server import SimpleXMLRPCServer
def add(x, y):
    return x + y
if __name__ == '__main__':
    s = SimpleXMLRPCServer(('127.0.0.1', 8088))
    s.register_introspection_functions()
    s.register_function(add)
    s.register_function(pow)
    s.serve_forever()
```

```python
from xmlrpc.client import ServerProxy
if __name__ == '__main__':
    s = ServerProxy("http://127.0.0.1:8080")
    print(s.system.listMethods())
    print(s.add(99,98))
    print(s.pow(4,9))
```

注册rpc，调用rpc，简单直接！

Rpc的本质

- RPC(Remote function call)

  远程函数调用！

  像调用本地函数一样调用远程函数，不用关心底层细节(序列化/反序列化、网络IO、rpc路由和调用……)

C++的rpc库可以做得更好用吗？

- 2015年提出了rest_rpc想法
  - https://www.cnblogs.com/qicosmos/p/5019339.html

REST RPC变体原型：

```
string call(string service_name, Args… args);
```

- 客户端调用：

```
call("handler2", " TiMax", 20, "zhuhai");
```

- 对应的服务端处理程序

```
handler2(string name, int age, string city);
```

从rest_rpc开始

- 2016 年实现了rest_rpc的第一个版本 (https://github.com/qicosmos/rest_rpc)

```
std::string echo(rpc_conn conn, const std::string &src)
{
  g_qps.increase();
  return src;
}

rpc_server server(9000,                    // 注册
std::thread::hardware_concurrency());      rpc
server.register_handler("echo", echo);
rpc_client client("127.0.0.1", 9000);
client.connect();
auto result = client.call<std::string>("echo",     // 调用
"test");                                           rpc
std::cout << result << std::endl;
```

简单直接，不到10行代码

从rest_rpc开始

- rest_rpc优点：非常易用
- rest_rpc缺点：
  - rpc调用缺少参数匹配的检查
  - rpc服务名是字符串，可能会写错

如何增强api调用的安全性？

从rest_rpc 到 Ray C++ Worker

- 2020年参与开发了Ray C++ Worker(https://github.com/ray-project/ray/blob/master/cpp/example/example.cc)

```
int Plus(int x, int y) { return x + y; }
RAY_REMOTE(Plus);

int main() {
  auto task_object = ray::Task(Plus).Remote(1, 2);
  auto result = task_object.Get();
}
```

rpc调用更安全（rpc函数类型检查、参数检查）

缺点：
为了rpc调用的安全性，client需要依赖rpc的实现(动态库)，依赖比较重，rpc函数的实现事实上也暴露给client了

coro_rpc

https://github.com/alibaba/yalantinglibs.git

- 2022年开发了coro_rpc 第一个版本

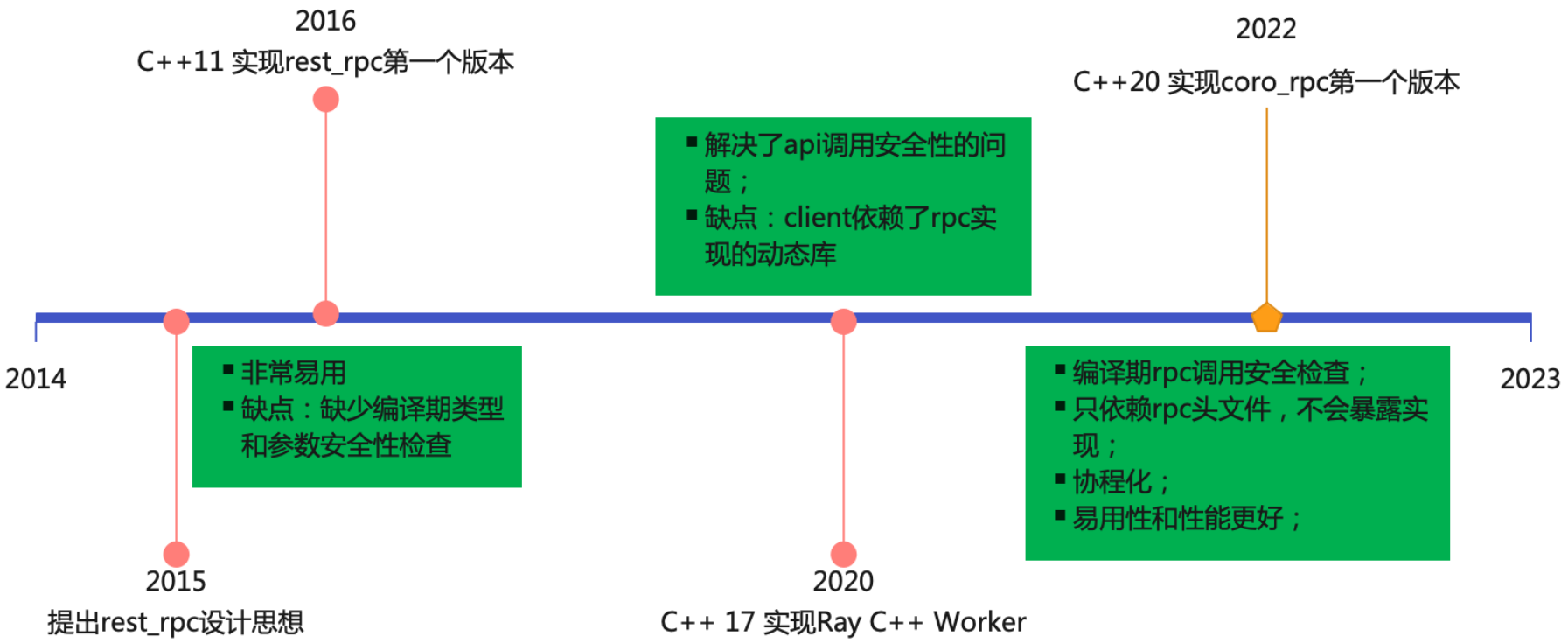
```
// rpc_service.h
std::string echo(std::string str);

// rpc_service.cpp
std::string echo(std::string str) { return str; }

#include <coro_rpc/coro_rpc_server.hpp>
#include "rpc_service.h"
int main() {
  register_handler<echo>(); // register rpc function

  coro_rpc_server server(/*thread_num =*/10, 8801);
  server.start();
}
```

coro_rpc

```
# include "rpc_service.h"
# include <coro_rpc/coro_rpc_client.hpp>

Lazy<void> test_client() {
  coro_rpc_client client;
  co_await client.connect("localhost", /*port =*/9000);

  auto r = co_await client.call<echo>("hello coro_rpc"); //调用rpc函数
  std::cout << r.result.value() << "\n"; //will print "hello coro_rpc"
}

int main() {
  syncAwait(test_client());
}
```

api调用很安全，有类型和参数检查；
Client只依赖rpc函数的头文件，rpc的实现不会暴露给client;

coro_rpc的故事


2016
C++11 实现rest_rpc第一个版本
2022
C++20 实现coro_rpc第一个版本
解决了api调用安全性的问题；
缺点：client依赖了rpc实现的动态库
2014
非常易用
缺点：缺少编译期类型和参数安全性检查
2023
编译期rpc调用安全检查；
只依赖rpc头文件，不会暴露实现；
协程化；
易用性和性能更好；
2015
提出rest_rpc设计思想
2020
C++ 17 实现Ray C++ Worker

coro_rpc的特色

- 易用
  - 就像定义普通函数一样定义rpc函数
  - 就像调用本地函数一样调用远程rpc函数
  - 几行代码就可以完成一个rpc服务
  - 用户只需要关注业务逻辑，不用关心底层细节(序列化、路由由框架完成，用户无感知)
  - 免安装，包含头文件就可以用
- 安全
  - client调用rpc，会在编译期检查做类型和参数的检查
  - client只依赖rpc函数的头文件，rpc的实现不会暴露给client
- 高性能(详见benchmark部分)

coro_rpc example

支持任意参数

```
int get_value(int a, int b){return a + b;}
person get_person(person p, int id) { return p;}

struct dummy {
  std::string echo(std::string str) { return str; }
};
```

参数的序列化由struct_pack自动完成，用户无感知

```
int main() {
  register_handler<get_value, get_person>();//注册任意参数类型的普通函数

  dummy d{};
  register_handler<&dummy::echo>(&d); //注册成员函数

  coro_rpc_server server(/*thread_num =*/10, /*port =*/9000);
  server.start();
}
```

支持协程函数

```
Lazy<int> get_coro_value(int val) { co_return val; }

int main() {
  register_handler<get_coro_value>();//注册协程函数
  coro_rpc_server server(/*thread_num =*/hardware_concurrency(), /*port =*/9000);
  server.start();
}
```

```
auto client = create_client();
auto ret = client.call<get_coro_value>(42);
CHECK(ret.value() == 42);
```

支持delay处理

```
void hello_with_delay(connection<std::string> conn) {
  // 在线程或者线程池中处理rpc请求
  std::thread([conn]() mutable {
    // 处理完成后response结果
    conn.response_msg("hello coro_rpc");
  }).detach();
}

int main() {
  register_handler<hello_with_delay>();
  coro_rpc_server server(2, 8801);
  server.start();
}
```

支持异步回调

```
# include <coro_rpc/async_rpc_server.hpp>
inline std::string hello() { return "hello coro_rpc"; }

int main() {
  register_handler<hello>();
  async_rpc_server server(/*thread_num =*/10, /*port =*/9000);
  server.start();
}

int main() {
  register_handler<hello>();
  coro_rpc_server server(/*thread_num =*/10, /*port =*/9000);
  server.start();
}
```

经典的异步回调的server

异步协程的server

coro_rpc benchmark

coro_rpc benchmark

- 环境

  Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz 96核

  OS: Linux version 4.9.151-015.ali3000.alios7.x86_64

  编译器：Alibaba Clang13 C++20

  编译选项：Release –O3

Alibaba Cloud Compiler(LLVM)

Alibaba Cloud Compiler(ACC) LLVM编译器，相比GCC，或其他Clang/LLVM版本在编译、构建速度上有很大的提升；

利用ACCThinLTO、AutoFDO和Bolt等技术可以在不同程度上优化程序性能。

通过编译器切换升级到ACC，可以在不用大幅修改代码的情况下获得性能提升和编译速度大幅提升。

ACC与C++基础库(async_simple、struct_pack和coro_rpc等库)，为龙蜥社区开发者提供了C++开发的一站式解决方案，快速构建高性能的C++应用。

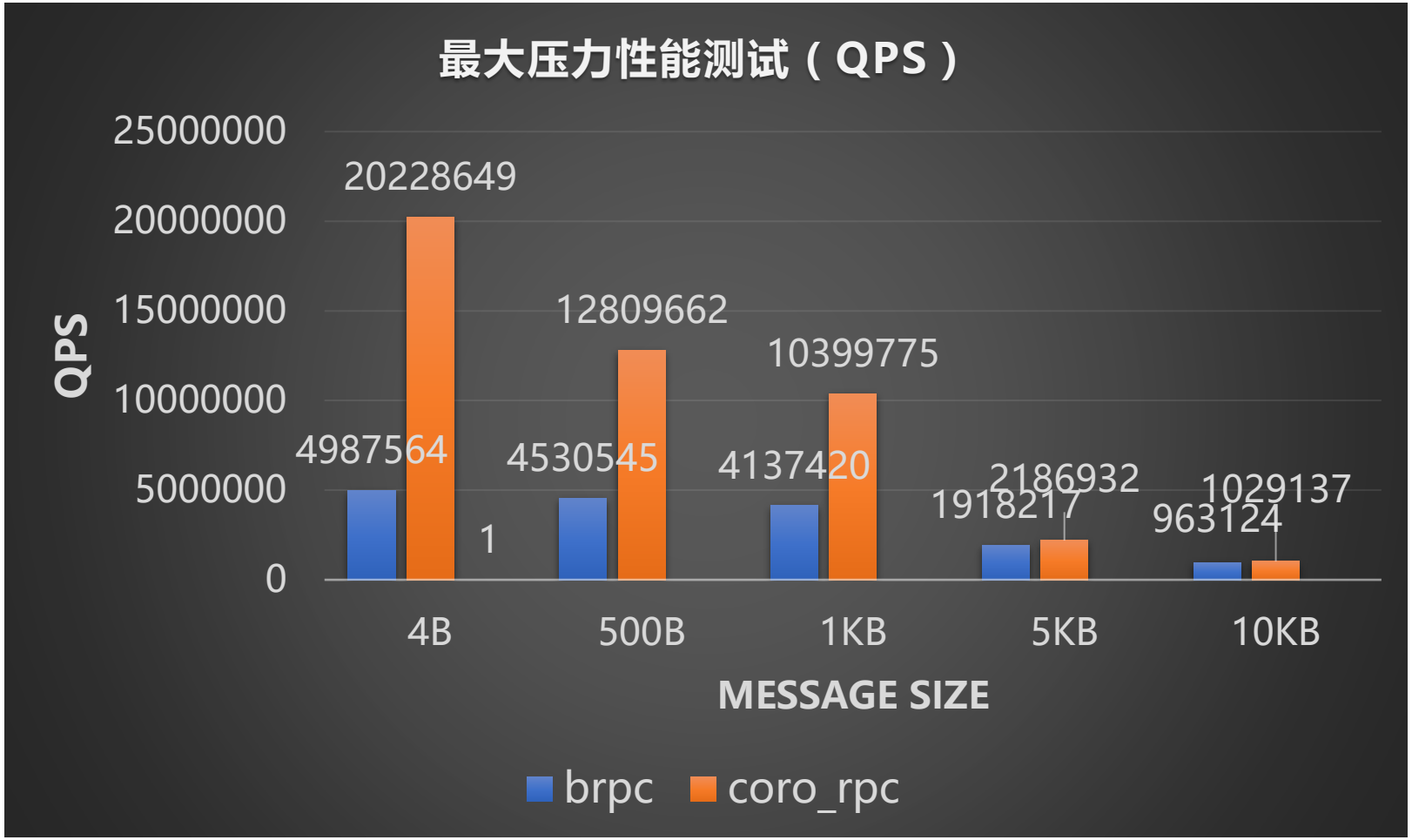
ACC Clang13 还提供了C++ 20 协程调试功能(GCC还没有)和std::module，对C++20 的支持较好。

Benchmark 说明

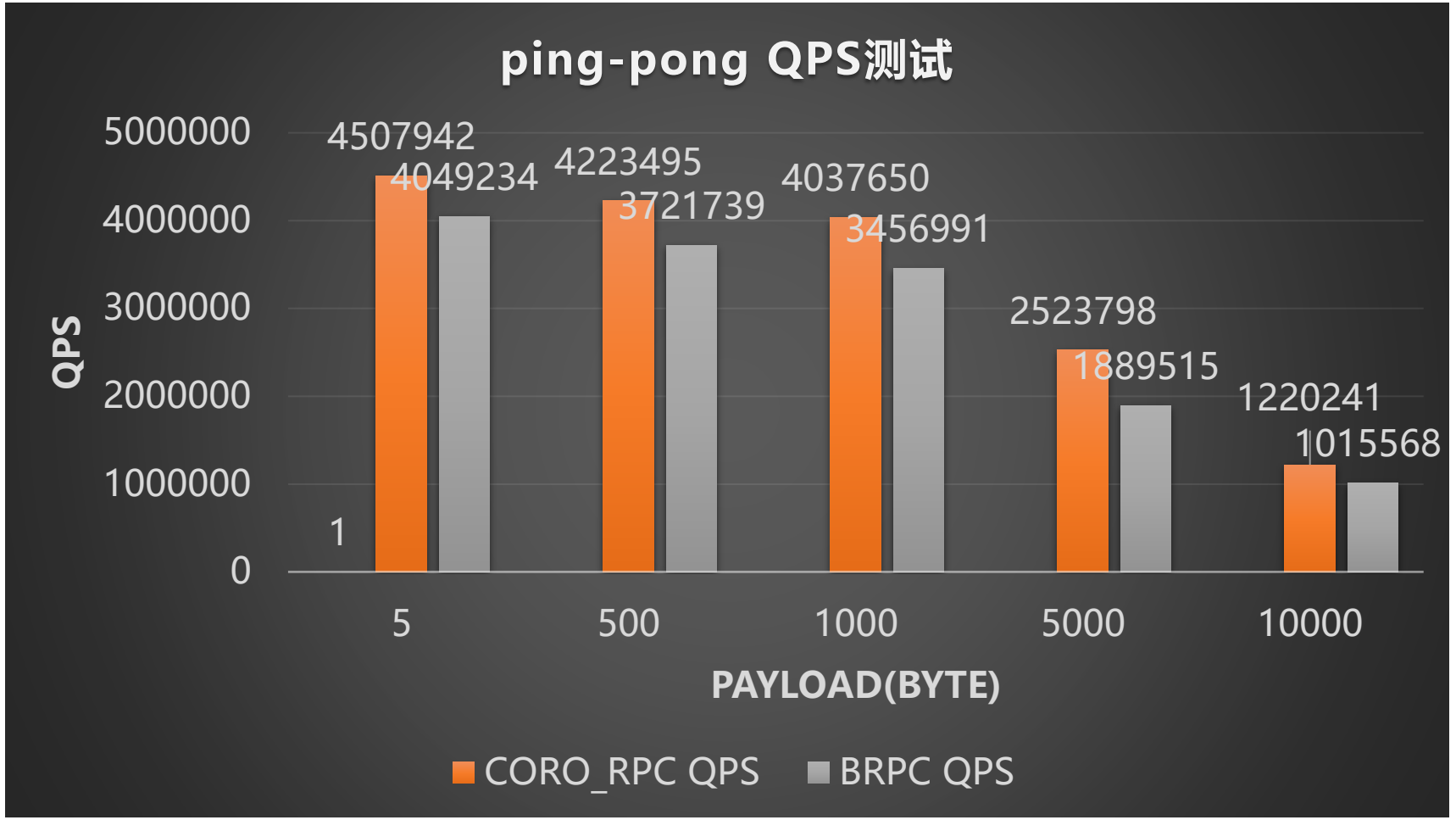
- 测试case
  - 客户端和服务端都在同一台机器上，使用不同连接数发送请求做echo测试
  - Pipeline 模式下极限qps 测试
  - Ping-pong模式下的qps和时延测试

- 测试备注
  - brpc由于采用了连接复用，实际上的socket连接数并没有那么多(实际连接数为96)，coro_rpc的连接数是实际的连接数。
  - 测试客户端统一用coro_rpc的压测客户端测试，压测效果较好，基本能把cpu打满。没有使用brpc测试客户端原因是：如果用brpc客户端压测，brpc qps会降低一倍；
  - grpc的qps始终不会超过10万，故没有放进来做性能比较；
  - 目前测试场景仅仅是echo测试，没有测试更多更复杂的场景，测试数据仅供参考；

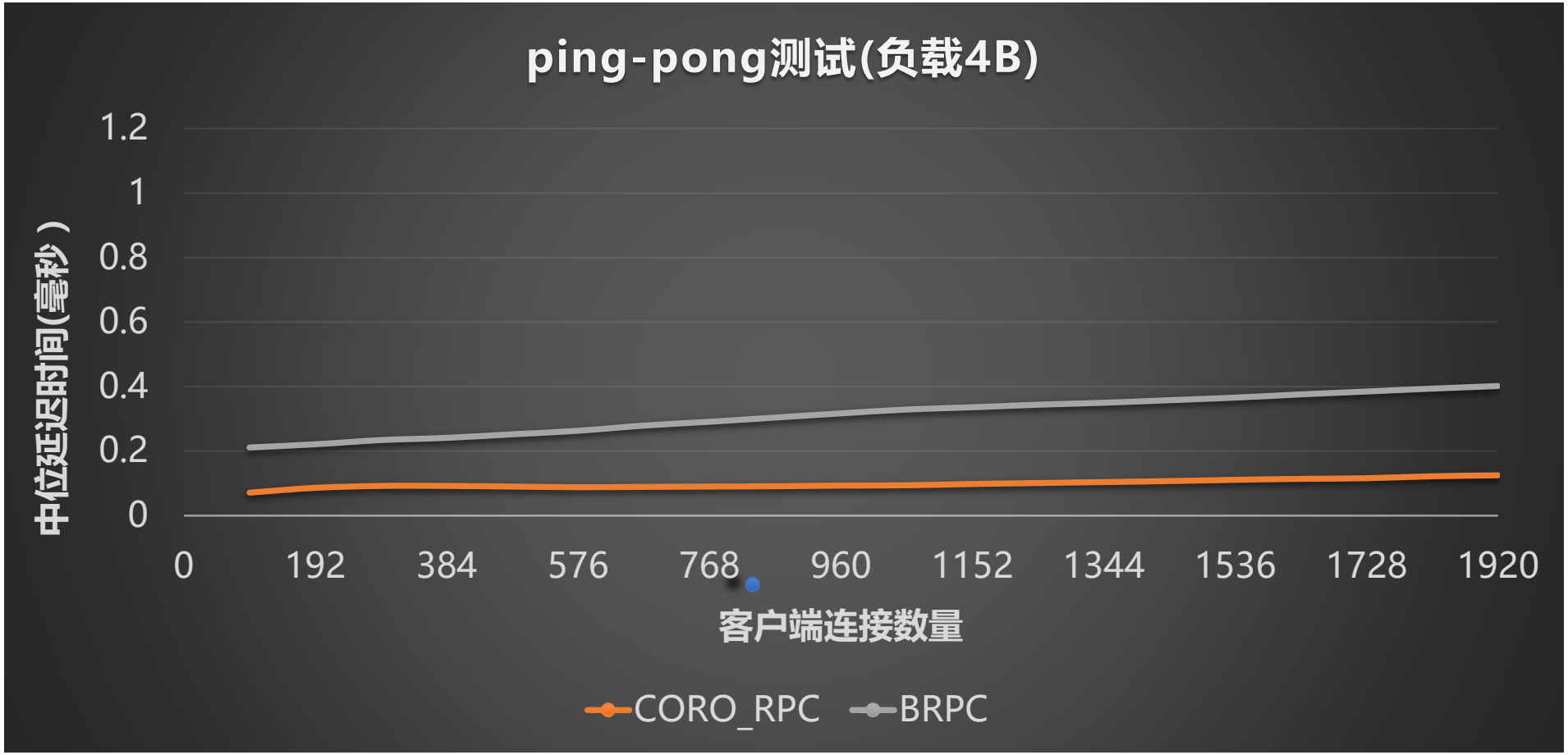
极限qps


最大压力性能测试（QPS）
QPS
25000000
20000000
15000000
10000000
5000000
0
20228649
12809662
10399775
4987564
4530545
4137420
2186932
1918217
1029137
963124
1
4B
500B
1KB
5KB
10KB
MESSAGE SIZE
brpc
coro_rpc

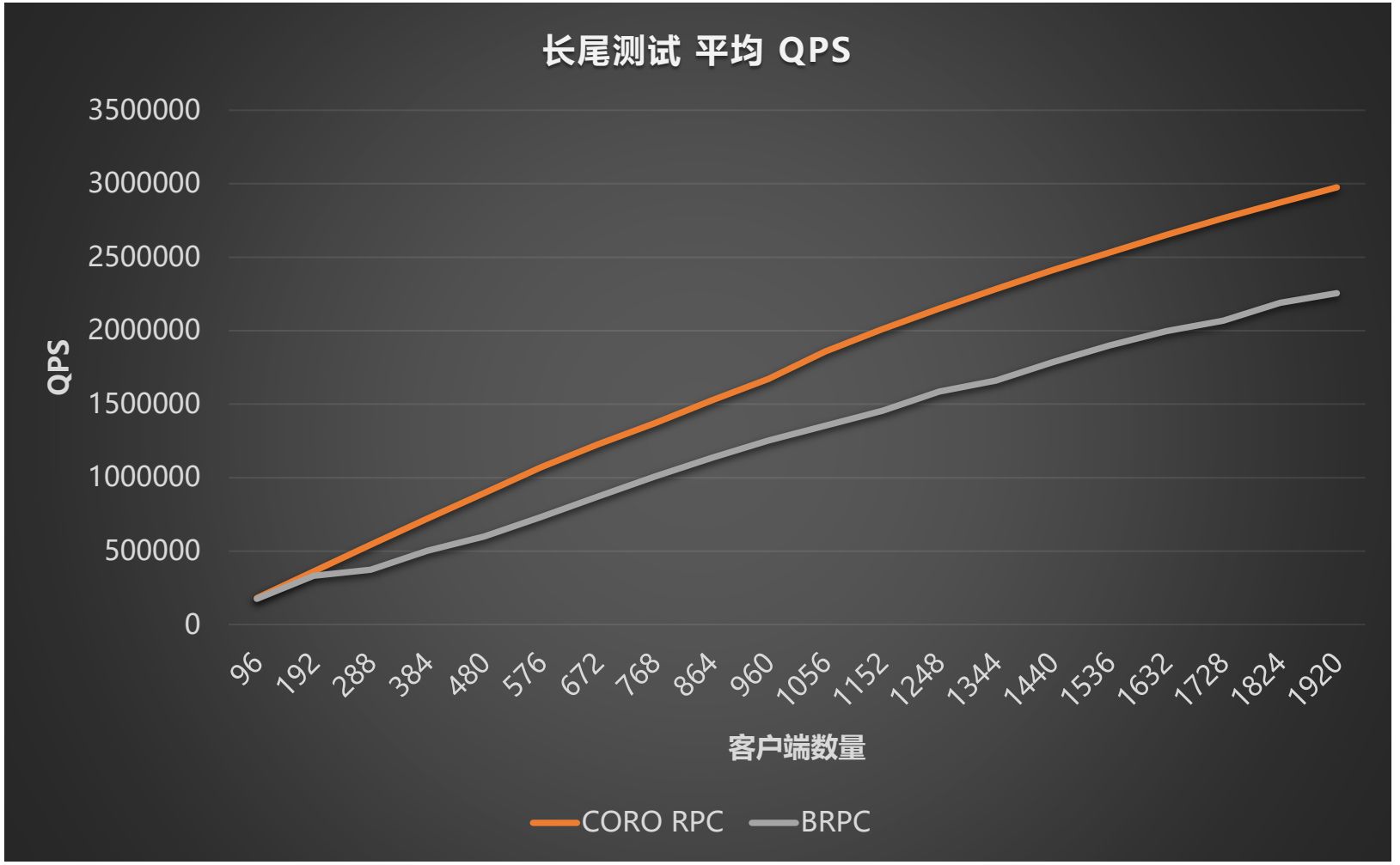
ping-pong qps



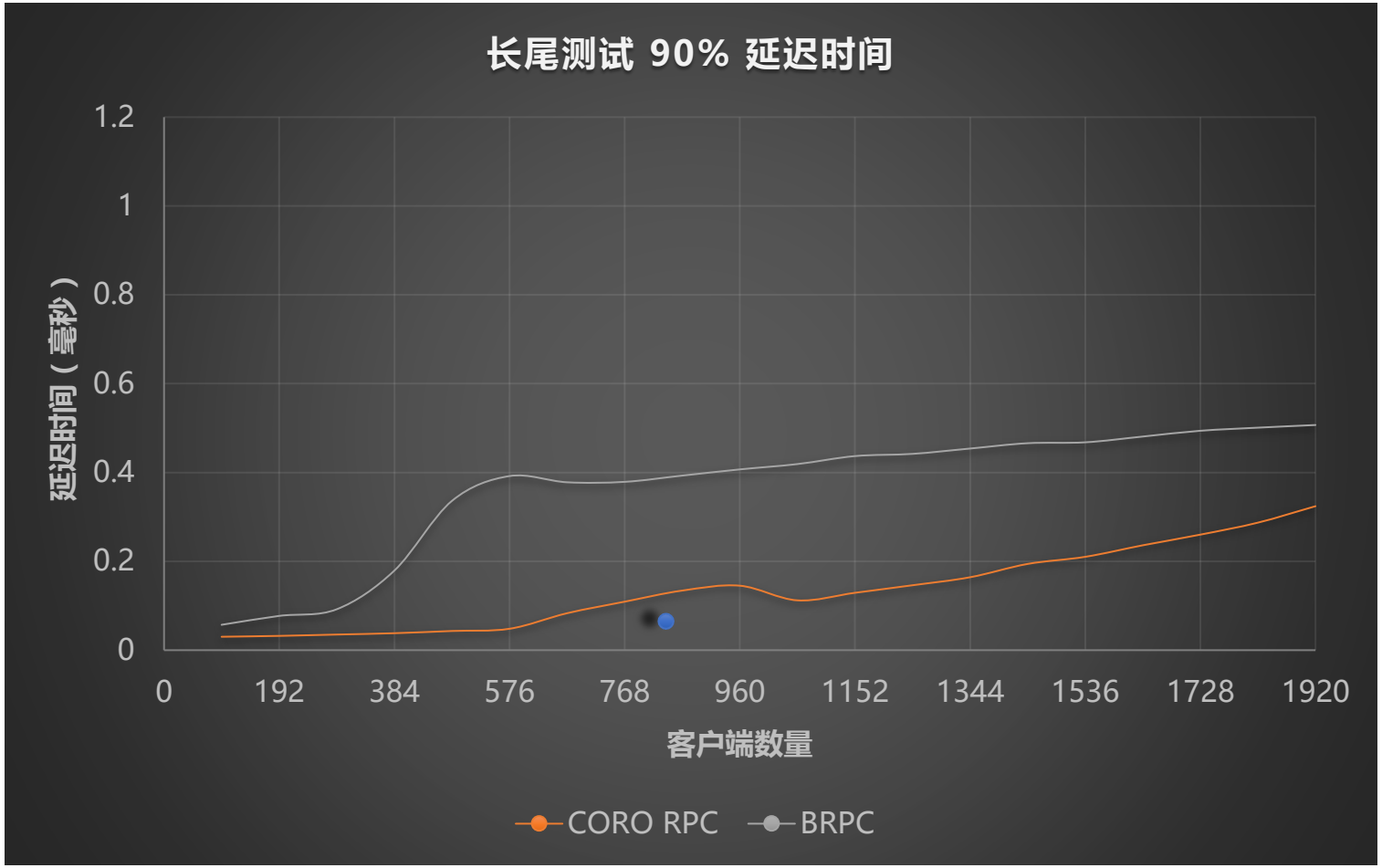
ping-pong 时延


ping-pong测试(负载4B)
中位延迟时间(毫秒)
1.2
1
0.8
0.6
0.4
0.2
0
0
192
384
576
768
960
1152
1344
1536
1728
1920
客户端连接数量
CORO_RPC
BRPC

长尾测试 qps

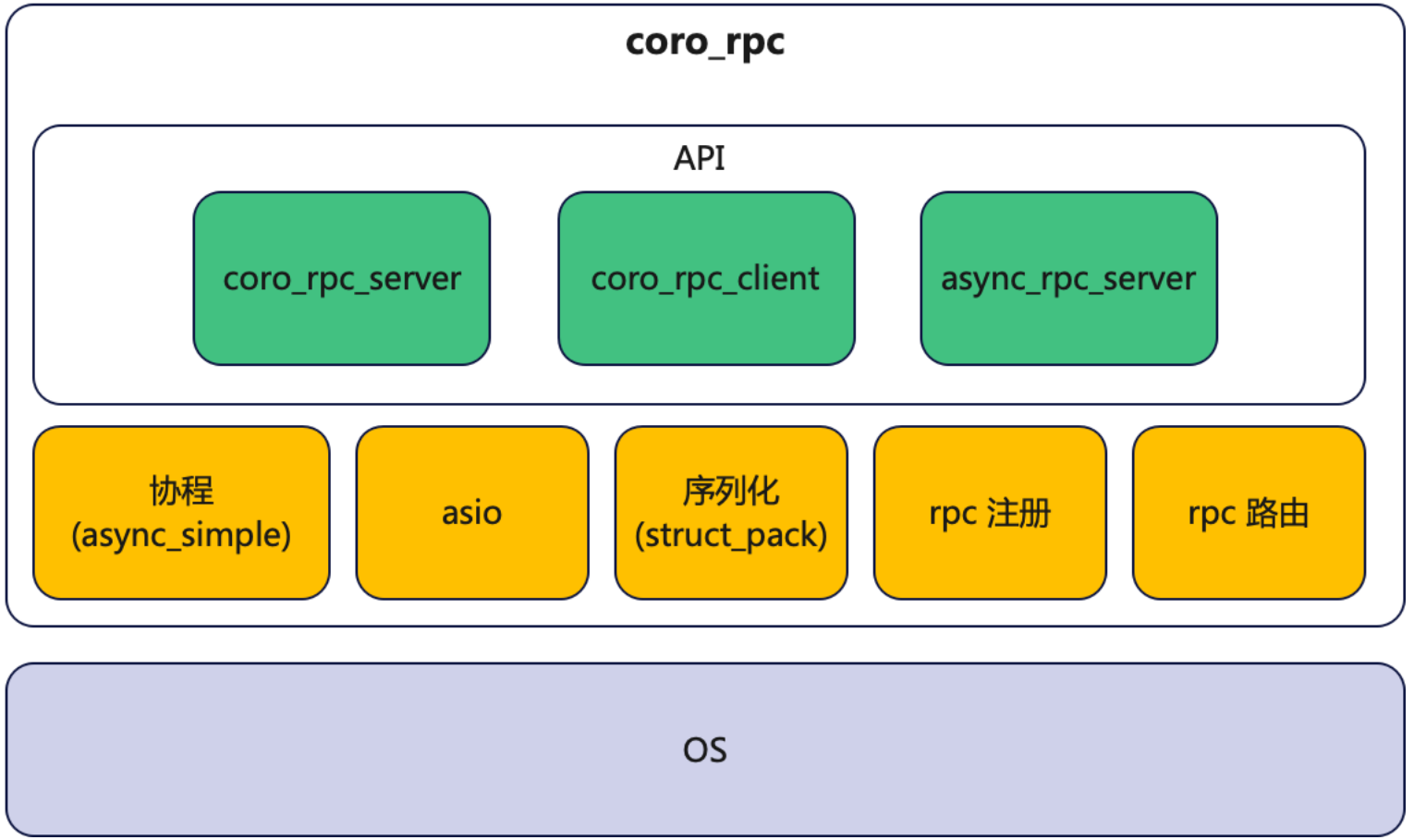

长尾测试 平均 QPS
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
QPS
96
192
288
384
480
576
672
768
864
960
1056
1152
1248
1344
1440
1536
1632
1728
1824
1920
客户端数量
CORO RPC
BRPC

长尾测试时延


长尾测试 90% 延迟时间
延迟时间（毫秒）
1.2
1
0.8
0.6
0.4
0.2
0
0
192
384
576
768
960
1152
1344
1536
1728
1920
客户端数量
CORO RPC
BRPC

coro_rpc 性能优化实践

coro_rpc性能优化实践



coro_rpc性能优化实践

- 从底层开始优化，开发了高性能序列化库struct_pack(性能是protobuf的10-40倍)
- 利用asio优秀的异步网络IO模型(proactor) + 多核提升性能
- 避免内存移动
- 避免锁
- 编译期计算

利用asio多核的优势

```
// 利用asio io_context pool，每次连接从pool中拿一个io_context去处理，并行化
void do_accept() {
  acceptor_.async_accept(
    [this](asio::error_code ec, asio::ip::tcp::socket socket) mutable {
      auto conn = std::make_shared<async_connection>(
            pool_.get_io_context_ptr(), conn_timeout_duration_,
            std::move(socket));
      conn->start();

      do_accept();
  });
}
```

qps 20w

利用asio多核的优势

```
void do_accept() {
  auto conn =   std::make_shared<async_connection>(pool_.get_io_context(),
conn_timeout_duration_);

  acceptor_.async_accept(
    conn->socket(), [this, conn](asio::error_code ec) mutable {
      conn->start();

      do_accept();
  });
}
```

qps 2000w

避免内存移动

- rpc响应的时候需要一个head和body
  先创建head再创建body，响应时合并成一个buffer发送；
  三次创建buffer，四次内存拷贝；

  优化：序列化的时候把head部分预留出来；
  一次创建buffer，两次内存拷贝；

```
template <typename R>
void response_msg(const R &ret) {
  auto buf = struct_pack::serialize_with_offset(RESPONSE_HEADER_LEN, ret);
  *((uint32_t *)buf.data()) = buf.size() - RESPONSE_HEADER_LEN;
  write(std::move(buf));
}
```

使用std::string_view去操作内存而不是std::string，避免内存拷贝

避免锁

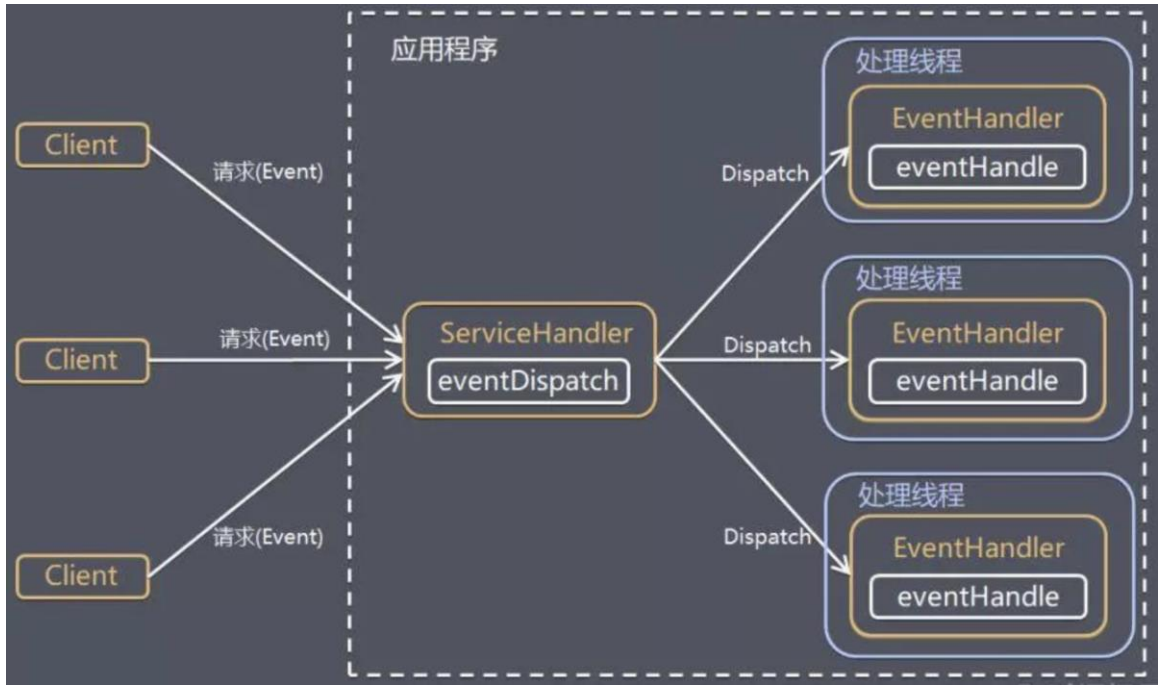
- 一个io_context处理一个connection
- connection内部无锁，性能最优
- 通过io_context.post来保证线程安全

编译期计算

- 编译期从函数映射到请求id

```
// 编译期生成function_id，消除了运行期开销
template <auto func>
consteval auto func_id() {
  constexpr auto name = get_func_name<func>();
  constexpr auto id = MD5Hash32Constexpr(name.data(), name.length());
  return id;
}
```

Reactor模式和Proactor模式



Proactor模式感知的是已完成的读写事件，而不需要像 Reactor 感知到事件后，还需要调用 read 来从内核中获取数据，性能更优。

https://www.zhihu.com/question/26943938

asio采用的是proactor模式

踩过的坑

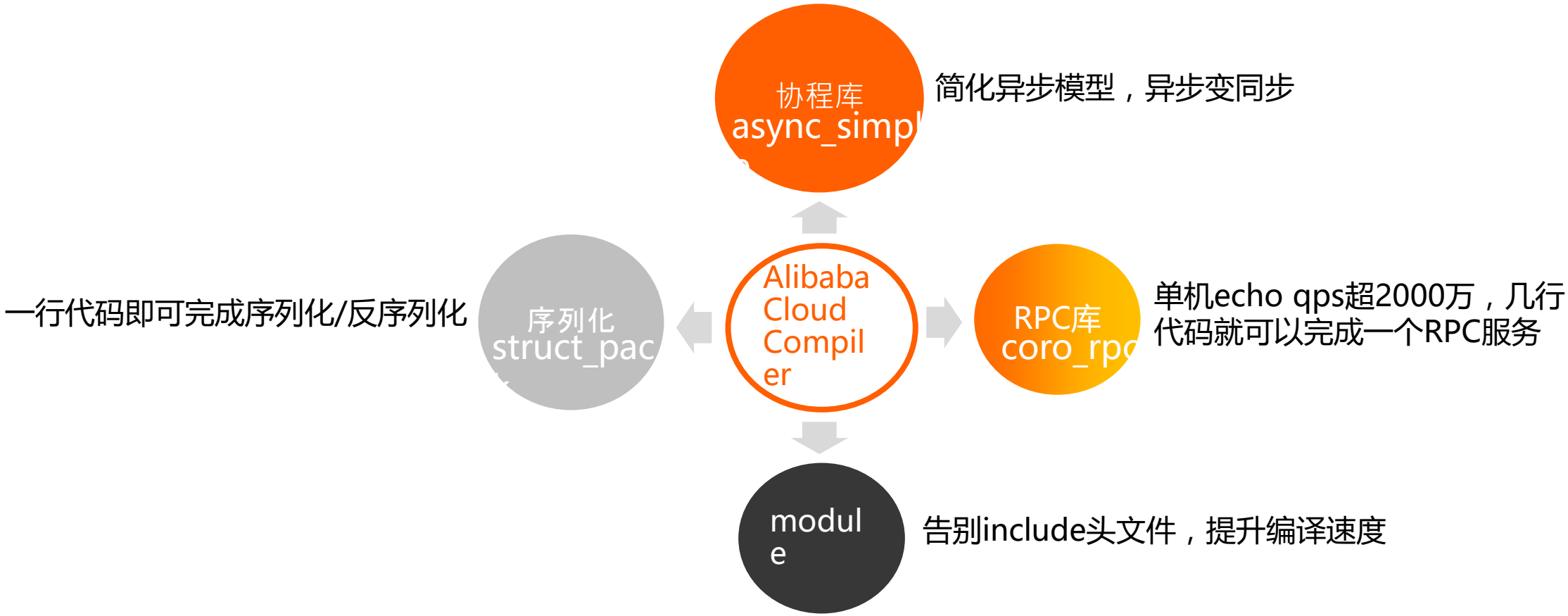
踩过的坑

- **使用协程需要注意的问题（一）**
- http://www.purecpp.cn/detail?id=2313
- **使用协程需要注意的问题（二）**
  http://www.purecpp.cn/detail?id=2315
- **使用协程需要注意的问题（三）**
  http://www.purecpp.cn/detail?id=2316
- **使用asio需要注意的问题**
  http://www.purecpp.cn/detail?id=2308

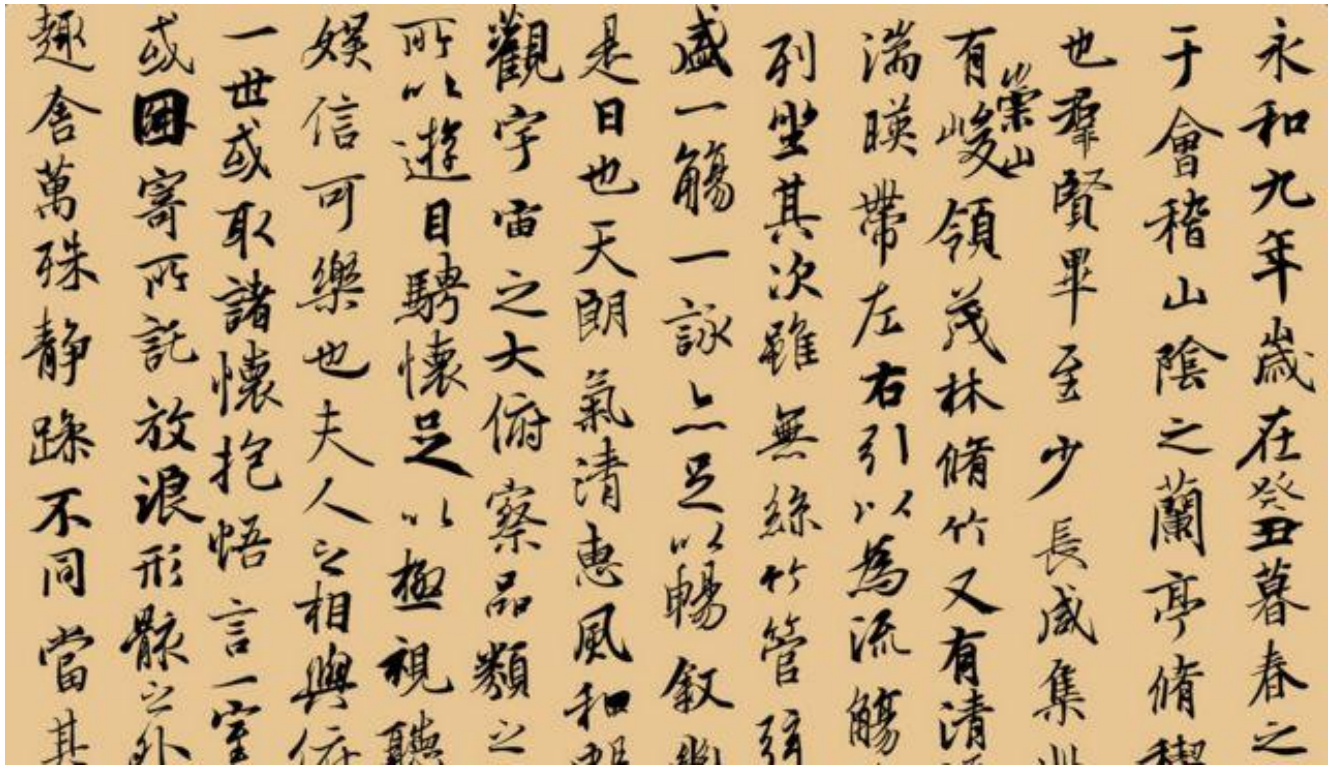
协程没有退出导致内存泄漏
协程生命周期的问题
Asio异步的多线程操作socket和io的问题

C++ 20 兰亭集库 yaLanTingLibs
https://github.com/alibaba/yalantinglibs.git

yaLanTingLibs 是基于C++最新标准C++20开发的库合集;
yaLanTingLibs和Alibaba Cloud Compiler(LLVM)为龙蜥社区开发者提供了C++开发的一站式解决方案，帮助用户快速构建高性能的C++应用



C++ 20 兰亭集库 yalantinglibs
https://github.com/alibaba/yalantinglibs.git



本次purecpp大会的问卷调查

purecpp公众号

purecpp