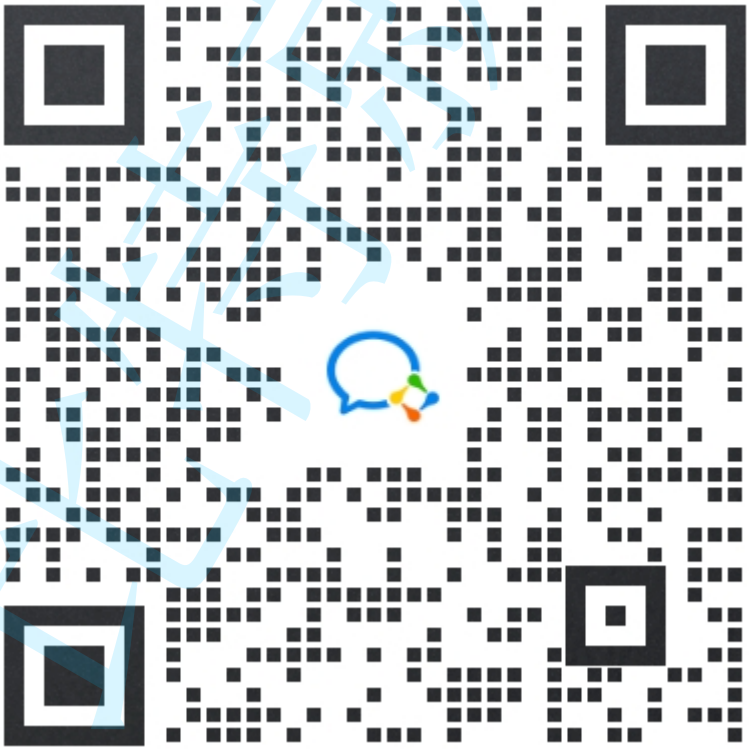
brpc 安装及使用

版权说明

对比特项目感兴趣，可以联系这个微信。



代码 & 板书链接

https://gitee.com/bitedu-tech/cpp-chatsystem

介绍

brpc 是用 c++语言编写的工业级 RPC 框架，常用于搜索、存储、机器学习、广告、推荐等高性能系统。

你可以使用它：

- 搭建能在**一个端口**支持多协议的服务，或访问各种服务
  - restful http/https, h2/gRPC。使用 brpc 的 http 实现比 libcurl 方便多了。从其他语言通过 HTTP/h2+json 访问基于 protobuf 的协议.
  - redis 和 memcached，线程安全，比官方 client 更方便。
  - rtmp/flv/hls，可用于搭建流媒体服务.
  - 支持 thrift，线程安全，比官方 client 更方便
  - 各种百度内使用的协议：baidu_std, streaming_rpc, hulu_pbrpc, sofa_pbrpc, nova_pbrpc, public_pbrpc, ubrpc 和使用 nshead 的各种协议.
  - 基于工业级的 RAFT 算法实现搭建高可用分布式系统，已在 braft 开源。
- Server 能同步或异步处理请求。
- Client 支持同步、异步、半同步，或使用组合 channels 简化复杂的分库或并发访问。
- 通过 http 界面调试服务，使用 cpu，heap，contention profilers.
- 获得更好的延时和吞吐.
- 把你组织中使用的协议快速地加入 brpc，或定制各类组件，包括命名服务 (dns, zk, etcd)，负载均衡 (rr, random, consistent hashing)

对比其他 rpc 框架

RPC框架	开发者	语言支持	序列化	http协议	文档	编译问题 (ubuntu 22.04)	服务端
grpc	google	C++、Java、Python、Go	pb	支持	文档较为完善，没有中文文档	主要是下载submodule的问题，没有vpn	不支持
brpc	baidu	C++	pb	支持	文档较为完善，有中文文档	编译基本没有问题	支持，的不好一次长用实现

点击图片可查看完整电子表格

安装

先安装依赖

```
C++
dev@dev-host:~/workspace$ sudo apt-get install -y git g++ make
libssl-dev libprotobuf-dev libprotoc-dev protobuf-compiler
libleveldb-dev
```

安装 brpc

```
C++
dev@dev-host:~/workspace$ git clone
https://github.com/apache/brpc.git
dev@dev-host:~/workspace$ cd brpc/
dev@dev-host:~/workspace/brpc$ mkdir build && cd build
dev@dev-host:~/workspace/brpc/build$ cmake -
DCMAKE_INSTALL_PREFIX=/usr .. && cmake --build . -j6
dev@dev-host:~/workspace/brpc/build$ make && sudo make install
```

类与接口介绍

日志输出类与接口

包含头文件：`#include <butil/logging.h>`

日志输出这里，本质上我们其实用不着 brpc 的日志输出，因此在这里主要介绍如何关闭日志输出。

```C++
namespace logging {
enum LoggingDestination {
    LOG_TO_NONE             = 0
};
struct BUTIL_EXPORT LoggingSettings {
    LoggingSettings();
    LoggingDestination logging_dest;
};
bool InitLogging(const LoggingSettings& settings);
}
```

protobuf 类与接口

```C++
namespace google {
namespace protobuf {
    class PROTOBUF_EXPORT Closure {
    public:
        Closure() {}
        virtual ~Closure();
        virtual void Run() = 0;
};
inline Closure* NewCallback(void (*function)());
class PROTOBUF_EXPORT RpcController {
    bool Failed();
    std::string ErrorText() ;
}
}
}
```

服务端类与接口

这里只介绍主要用到的成员与接口。

```C++
namespace brpc {
struct ServerOptions {
    //无数据传输，则指定时间后关闭连接
    int idle_timeout_sec; // Default: -1 (disabled)
    int num_threads; // Default: #cpu-cores
    //....
}
```

```
enum ServiceOwnership {
    //添加服务失败时，服务器将负责删除服务对象
    SERVER_OWNS_SERVICE,
    //添加服务失败时，服务器也不会删除服务对象
    SERVER_DOESNT_OWN_SERVICE
};
class Server {
    int AddService(google::protobuf::Service* service,
                   ServiceOwnership ownership);
    int Start(int port, const ServerOptions* opt);
    int Stop(int closewait_ms/*not used anymore*/);
    int Join();
    //休眠直到 ctrl+c 按下，或者 stop 和 join 服务器
    void RunUntilAskedToQuit();
}
class ClosureGuard {
    explicit ClosureGuard(google::protobuf::Closure* done);
    ~ClosureGuard() { if (_done) _done->Run(); }
}
class HttpHeader {
    void set_content_type(const std::string& type)
    const std::string* GetHeader(const std::string& key)
    void SetHeader(const std::string& key,
        const std::string& value);
    const URI& uri() const { return _uri; }
    HttpMethod method() const { return _method; }
    void set_method(const HttpMethod method)
    int status_code()
    void set_status_code(int status_code);

}
class Controller : public google::protobuf::RpcController {
    void set_timeout_ms(int64_t timeout_ms);
    void set_max_retry(int max_retry);
    google::protobuf::Message* response();
    HttpHeader& http_response();
    HttpHeader& http_request();
    bool Failed();
    std::string ErrorText();

    using AfterRpcRespFnType = std::function<
        void(Controller* cntl,
        const google::protobuf::Message* req,
        const google::protobuf::Message* res)>;
```

```
    void set_after_rpc_resp_fn(AfterRpcRespFnType&& fn)
}
```

客户端类与接口：

```C++
namespace brpc {
struct ChannelOptions {
    //请求连接超时时间
    int32_t connect_timeout_ms;// Default: 200 (milliseconds)
    //rpc 请求超时时间
    int32_t timeout_ms;// Default: 500 (milliseconds)
    //最大重试次数
    int max_retry;// Default: 3
    //序列化协议类型  options.protocol = "baidu_std";
    AdaptiveProtocolType protocol;
    //....
}
class Channel : public ChannelBase {
    //初始化接口，成功返回 0；
    int Init(const char* server_addr_and_port,
        const ChannelOptions* options);
```

使用

同步调用

同步调用是指客户端会阻塞收到 server 端的响应或发生错误。

下面我们以 Echo（输出 hello world）方法为例, 来讲解基础的同步 RPC 请求是如何实现的。

- 创建 proto 文件 - main.proto

```ProtoBuf
syntax="proto3";
package example;

option cc_generic_services = true;

// 定义 Echo 方法请求参数结构
message EchoRequest {
      string message = 1;
```

```
};

// 定义 Echo 方法响应参数结构
message EchoResponse {
      string message = 1;
};

// 定义 RPC 远端方法
service EchoService {
      rpc Echo(EchoRequest) returns (EchoResponse);
};
```

- 创建服务端源码 - brpc_server.cpp

```
C++
#include <gflags/gflags.h>
#include <butil/logging.h>
#include <brpc/server.h>
#include <json2pb/pb_to_json.h>
#include "main.pb.h"

// 使用 gflags 定义一些命令行参数
DEFINE_int32(listen_port, 8000, "服务器地址信息");
DEFINE_int32(idle_timeout_s, -1, "空闲连接超时关闭时间：默认-1 表示不关闭");
DEFINE_int32(thread_count, 3, "服务器启动线程数量");

namespace example {
class EchoServiceImpl : public EchoService {
public:
    EchoServiceImpl() {}
    virtual ~EchoServiceImpl() {}
    // cntl_base：包含除了 request 和 response 之外的参数集合
    // request: 请求，只读的，来自 client 端的数据包
    // response: 回复。需要用户填充，如果存在 required 字段没有被设置，该次调用会失败。
    // done: done 由框架创建，递给服务回调，包含了调用服务回调后的后续动作，包括检查 response 正确性，序列化，打包，发送等逻辑。不管成功失败，done->Run()必须在请求处理完成后被用户调用一次。
    virtual void Echo(google::protobuf::RpcController* cntl_base,
                      const EchoRequest* request,
                      EchoResponse* response,
                      google::protobuf::Closure* done) {
```

```
        // 类型于守卫锁，以 ARII 方式自动释放 done 对象
        brpc::ClosureGuard done_guard(done);

        brpc::Controller* cntl =
            static_cast<brpc::Controller*>(cntl_base);

        // 可选项： 本质是设置一个 hook 函数，在发送响应后及在
cntl_base、request、response 释放之前调用
        cntl-
>set_after_rpc_resp_fn(std::bind(&EchoServiceImpl::CallAfterRpc,
            std::placeholders::_1, std::placeholders::_2,
std::placeholders::_3));

        // 打印一些相关的参数日志信息
        std::cout << "请求内容：" << request->message() <<
std::endl;

        // 填充响应，客户端发送什么数据，服务器就回复什么数据
        response->set_message(request->message() + " Hello");
    }

    // 可选项： 回调函数， 此时响应已经发回给客户端,但是相关结构还没释放
    static void CallAfterRpc(brpc::Controller* cntl,
                        const google::protobuf::Message* req,
                        const google::protobuf::Message* res) {
        std::string req_str;
        std::string res_str;
        json2pb::ProtoMessageToJson(*req, &req_str, NULL);
        json2pb::ProtoMessageToJson(*res, &res_str, NULL);
        std::cout << "req:" << req_str << std::endl;
        std::cout << "res:" << res_str << std::endl;
    }
};
}  // namespace example

int main(int argc, char* argv[]) {

    logging::LoggingSettings log_setting;
    log_setting.logging_dest =
logging::LoggingDestination::LOG_TO_NONE;
    logging::InitLogging(log_setting);
    // 解析命令行参数
    google::ParseCommandLineFlags(&argc, &argv, true);
```

```
    // 定义服务器
    brpc::Server server;

    // 创建服务对象.
    example::EchoServiceImpl echo_service_impl;

    // 将服务添加到服务器中
    if (server.AddService(&echo_service_impl,
        brpc::SERVER_DOESNT_OWN_SERVICE) != 0) {
        std::cout << "add service failed!\n";
        return -1;
    }
    // 开始运行服务器
    brpc::ServerOptions options;
    options.idle_timeout_sec = FLAGS_idle_timeout_s;
    options.num_threads = FLAGS_thread_count;
    if (server.Start(FLAGS_listen_port, &options) != 0) {
        std::cout << "Fail to start EchoServer";
        return -1;
    }

    // 阻塞等待服务端运行
    server.RunUntilAskedToQuit();
    return 0;
}
```

- 创建客户端源码 - client.cpp

```
C++
#include <gflags/gflags.h>
#include <butil/logging.h>
#include <butil/time.h>
#include <brpc/channel.h>
#include "main.pb.h"

DEFINE_string(protocol, "baidu_std", "通信协议类型，默认使用 brpc 自定制协议");
DEFINE_string(server_host, "127.0.0.1:8000", "服务器地址信息");
DEFINE_int32(timeout_ms, 500, "Rpc 请求超时时间-毫秒");
DEFINE_int32(max_retry, 3, "请求重试次数");

int main(int argc, char* argv[]) {
    // 解析命令行参数
```

```
    google::ParseCommandLineFlags(&argc, &argv, true);

    // 创建通道， 可以理解为客户端到服务器的一条通信线路
    brpc::Channel channel;

    // 初始化通道，NULL 表示使用默认选项
    brpc::ChannelOptions options;
    options.protocol = FLAGS_protocol;
    options.timeout_ms = FLAGS_timeout_ms;
    options.max_retry = FLAGS_max_retry;
    if (channel.Init(FLAGS_server_host.c_str(), &options) != 0) {
        LOG(ERROR) << "Fail to initialize channel";
        return -1;
    }
    //通常，我们不应直接调用通道，而是包装它的 stub 服务,通过 stub 进行
rpc 调用
    example::EchoService_Stub stub(&channel);

    // 创建请求、响应、控制对象
    example::EchoRequest request;
    example::EchoResponse response;
    brpc::Controller cntl;
    // 构造请求响应
    request.set_message("hello world");

    //由于“done”（最后一个参数）为 NULL，表示阻塞等待响应
    stub.Echo(&cntl, &request, &response, NULL);
    if (cntl.Failed()) {
        std::cout << "请求失败: " << cntl.ErrorText() << std::endl;
        return -1;
    }
    std::cout << "响应: " << response.message() << std::endl;
    return 0;
}
```

- 编写 Makefile

参考 example 的例子，修改一下 BRPC_PATH 即可。

```
Shell
all: brpc_server brpc_client
brpc_server: brpc_server.cc main.pb.cc
    g++ -std=c++17 $^ -o $@ -lbrpc -lleveldb -lgflags -lssl -
```

```
lcrypto -lprotobuf
brpc_client: brpc_client.cc main.pb.cc
    g++ -std=c++17 $^ -o $@ -lbrpc -lleveldb -lgflags -lssl -
lcrypto -lprotobuf
%.pb.cc : %.proto
    protoc --cpp_out ./ $<
```

异步调用

异步调用是指客户端注册一个响应处理回调函数， 当调用一个 RPC 接口时立即返回，不会阻塞等待响应， 当 server 端返回响应时会调用传入的回调函数处理响应。

> 具体的做法：给 CallMethod 传递一个额外的回调对象 done，CallMethod 在发出 request 后就结束了，而不是在 RPC 结束后。当 server 端返回 response 或发生错误（包括超时）时，done->Run()会被调用。对 RPC 的后续处理应该写在 done->Run()里，而不是 CallMethod 后。由于 CallMethod 结束不意味着 RPC 结束，response/controller 仍可能被框架及 done->Run()使用，它们一般得创建在堆上，并在 done->Run()中删除。如果提前删除了它们，那当 done->Run()被调用时，将访问到无效内存。

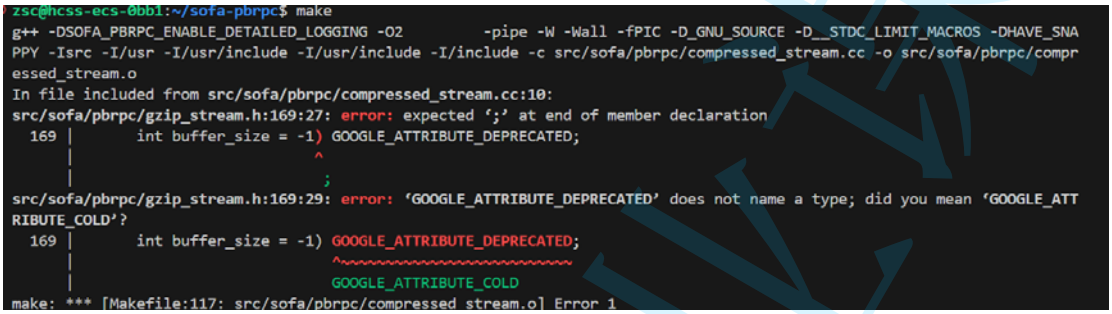
下面是异步调用的伪代码:

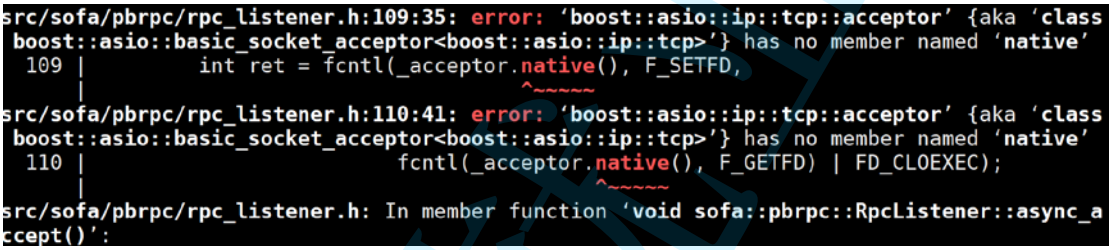
```C++
static void OnRPCDone(MyResponse* response, brpc::Controller*
cntl) {
    // unique_ptr 会帮助我们在 return 时自动删掉 response/cntl，防止忘记。gcc 3.4 下的 unique_ptr 是模拟版本。
    std::unique_ptr<MyResponse> response_guard(response);
    std::unique_ptr<brpc::Controller> cntl_guard(cntl);
    if (cntl->Failed()) {
        // RPC 失败了. response 里的值是未定义的，勿用。
    } else {
        // RPC 成功了，response 里有我们想要的数据。开始 RPC 的后续处理。
    }
    // NewCallback 产生的 Closure 会在 Run 结束后删除自己，不用我们做。
}

MyResponse* response = new MyResponse;
brpc::Controller* cntl = new
brpc::Controller;
MyService_Stub stub(&channel);
```

```
MyRequest request;  // 不用new request
request.set_foo(...);cntl->set_timeout_ms(...);
stub.some_method(cntl, &request, response,
brpc::NewCallback(OnRPCDone, response, cntl));
```

sofa-brpc

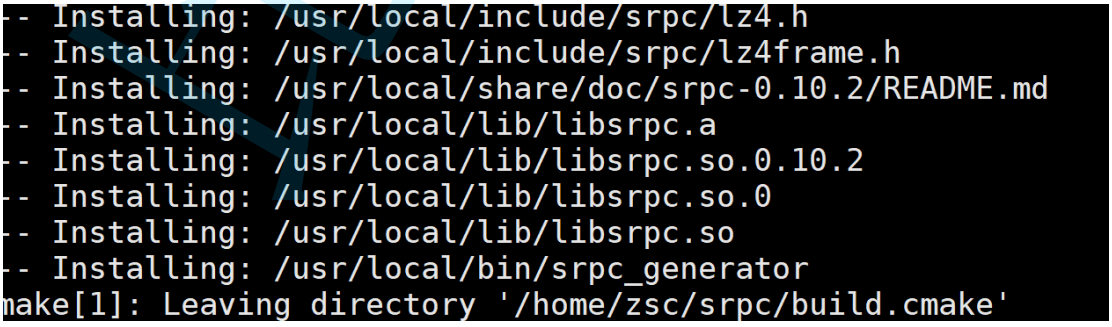




srpc

```Shell
git clone --recursive https://github.com/sogou/srpc.git
cd srpc
make
make install

# 编译示例
cd tutorial
make
```



封装思想

rpc 调用这里的封装，因为不同的服务调用使用的是不同的 Stub，这个封装起来的意义不大，因此我们只需要封装通信所需的 Channel 管理即可，这样当需要进行什么样的服务调用的时候，只需要通过服务名称获取对应的 channel，然后实例化 Stub 进行调用即可。

- 封装 Channel 的管理，每个不同的服务可能都会有多个主机提供服务，因此一个服务可能会对应多个 Channel，需要将其管理起来，并提供获取指定服务 channel 的接口
    - 进行 rpc 调用时，获取 channel，目前以 RR 轮转的策略选择 channel
- 提供进行服务声明的接口，因为在整个系统中，提供的服务有很多，但是当前可能并不一定会用到所有的服务，因此通过声明来告诉模块哪些服务是自己关心的，需要建立连接管理起来，没有添加声明的服务即使上线也不需要进行连接的建立。
- 提供服务上线时的处理接口，也就是新增一个指定服务的 channel
- 提供服务下线时的处理接口，也就是删除指定服务下的指定 channel