RPC浅析

Vincentyao

2015-05-18

Protobuf

- https://github.com/google/protobuf
- 优点：
  - 用来序列化结构化数据，类似于xml，但是smaller, faster, and simpler，适合网络传输
  - 支持跨平台多语言(e.g. Python, Java, Go, C++, Ruby, JavaNano)
  - 消息格式升级，有较好的兼容性(想想以前用struct定义网络传输协议,解除version的痛楚)
- 缺点：
  - 可读性差(not human-readable or human-editable)
  - 不具有自描述性(self-describing)

Protobuf version3

- 支持map类型

```
message Foo {
  map<string, string> values = 1;
}
```

- 去掉required fields, 去掉default values
- Arena allocation support。通过内存分配，性能提升20%~50%

```
{
  google::protobuf::Arena arena;
  // Allocate a protobuf message in the arena.
  MyMessage* message = Arena::CreateMessage<MyMessage>(&arena);
  // All submessages will be allocated in the same arena.
  if (!message->ParseFromString(data)) {
    // Deal with malformed input data.
  }
  // Must not delete the message here. It will be deleted automatically
  // when the arena is destroyed.
}
```

Protobuf的反射机制

- Reflection: 常用于pb与xml,json等其他格式的转换。

```
  const Reflection* reflection = message.GetReflection();

  vector<const FieldDescriptor*> fields;
  reflection->ListFields(message, &fields);

  for (size_t i = 0; i < fields.size(); i++) {
    const FieldDescriptor* field = fields[i];
    switch (field->cpp_type()) {
#define CASE_FIELD_TYPE(cpptype, method, jsontype) \
      case FieldDescriptor::CPPTYPE_##cpptype: { \
        if (field->is_repeated()) { \
          int field_size = reflection->FieldSize(message, field); \
          for (int index = 0; index < field_size; index++) { \
            (*json_value)[field->name()].append( \
                                static_cast<jsontype>( \
                                    reflection->GetRepeated##method( \
                                        message, field, index))); \
          } \
        }  else { \
          (*json_value)[field->name()] = static_cast<jsontype>( \
                                reflection->Get##method( \
                                    message, field)); \
        } \
        break; \
      }
```

Self-describing Messages

- 生产者：产生消息，填充内容，并序列化保存
- 消费者：读取数据，反序列化得到消息，使用消息
- 目的：解除这种耦合，让消费者能动态的适应消息格式的变换。
- 生产者把定义消息格式的.proto文件和消息作为一个完整的消息序列化保存，完整保存的消息我称之为Wrapper message，原来的消息称之为payload message。
- 消费者把wrapper message反序列化，先得到payload message的消息类型，然后根据类型信息得到payload message，最后通过反射机制来使用该消息。

```
message SelfDescribingMessage {
  // Set of .proto files which define the type.
  required FileDescriptorSet proto_files = 1;

  // Name of the message type.  Must be defined by one of the files in
  // proto_files.
  required string type_name = 2;

  // The message data.
  required bytes message_data = 3;
}
```

Self-describing Messages 生产者

- 使用 protoc生成代码时加上参数–descriptor_set_out，输出类型信息(即SelfDescribingMessage的第一个字段内容)到一个文件，这里假设文件名为desc.set,
  protoc –cpp_out=. –descriptor_set_out=desc.set addressbook.proto
- payload message使用方式不需要修改
  tutorial::AddressBook address_book;
  PromptForAddress(address_book.add_person());
- 在保存时使用文件desc.set内容填充SelfDescribingMessage的第一个字段，使用AddressBook
  AddressBook的full name填充SelfDescribingMessage的第二个字段，
  AddressBook序列化后的数据填充第三个字段。最后序列化SelfDescribingMessage保存到文件中。

Self-describing Messages 消费者

- 消费者编译时需要知道SelfDescribingMessage，不需要知道AddressBook，运行时可以正常操作AddressBook消息。

1. 首先反序列化SelfDescribingMessage

```
tutorial::SelfDescribingMessage sdmessage;
fstream input(argv[1], ios::in | ios::binary);
sdmessage.ParseFromIstream(&input));
```

2. 通过第一个字段得到FileDescriptorSet，通过第二个字段取得消息的类型名，使用DescriptorPool得到payload message的类型信息Descriptor

```
SimpleDescriptorDatabase db;
for(int i=0;i<sdmessage.proto_files().file_size();i++)
{    db.Add(sdmessage.proto_files().file(i));  }
DescriptorPool pool(&db);
const Descriptor *descriptor = pool.FindMessageTypeByName(sdmessage.type_name());
```

3. 使用DynamicMessage new出这个类型的一个空对象，从第三个字段反序列化得到原来的message对象

```
DynamicMessageFactory factory(&pool);
Message *msg = factory.GetPrototype(descriptor)->New();
msg->ParseFromString(sdmessage.message_data());
```

4. 通过Message的reflection接口操作message的各个字段

Protobuf实践

- 一般对日志数据只加不删不改，所以其字段设计要极慎重。
- 千万不要随便修改tag number。
- 不要随便添加或者删除required field。
- Clear并不会清除message memory（clear操作适合于清理那些数据量变化不大的数据，对于大小变化较大的数据是不适合的，需要定期（或每次）进行delete操作。建议swap或者delete）
- repeated message域，size不要太大。
- 如果一个数据太大，不要使用protobuf。

Socket编程

Client:

```
int fd = socket(PF_INET, SOCK_STREAM, 0);
PCHECK(fd >= 0);

sockaddr_in server_sin;
memset(&server_sin, 0, sizeof(server_sin));
server_sin.sin_family = AF_INET;
inet_pton(AF_INET, FLAGS_ip.c_str(), &(server_sin.sin_addr));
server_sin.sin_port = htons(FLAGS_port);

PCHECK(connect(fd, reinterpret_cast<sockaddr*>(&server_sin),
    sizeof(server_sin)) >= 0);
char buffer[kBufferLen];
printf("Please enter the message: ");
memset(buffer, 0, sizeof(buffer[0]) * kBufferLen);
fgets(buffer, kBufferLen - 1, stdin);
int n = write(fd, buffer, strlen(buffer));
PCHECK(n > 0);
LOG(INFO) << "Write message:" << buffer;
memset(buffer, 0, sizeof(buffer[0]) * kBufferLen);
n = read(fd, buffer, kBufferLen - 1);
PCHECK(n > 0);
LOG(INFO) << "Receive message:" << buffer;
```

Server:

```
int accept_sock = socket(PF_INET, SOCK_STREAM, 0);
PCHECK(accept_sock >= 0);

sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(FLAGS_port);
PCHECK(bind(accept_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == 0);
PCHECK(listen(accept_sock, 128) == 0);

int n = 0;
char buffer[kBufferLen];
while (true) {
  sockaddr_in client_sin;
  socklen_t addrlen = sizeof(client_sin);
  int new_fd = accept(accept_sock, reinterpret_cast<sockaddr*>(&client_sin), &addrlen);
  PCHECK(new_fd >= 0);
  memset(buffer, 0, sizeof(buffer[0]) * kBufferLen);
  n = read(new_fd, buffer, kBufferLen - 1);
  LOG(INFO) << "Read Message:" << buffer;
  int send_len = snprintf(buffer, kBufferLen - 1,
      "Hello, port:%d, get your message", client_sin.sin_port);
  n = write(new_fd, buffer, send_len);
  LOG(INFO) << "send_len:" << send_len << ",have sent:" << n;
}
```

Socket option

- 非阻塞IO: O_NONBLOCK
- CloseOnExec: FD_CLOEXEC (该句柄在fork子进程后执行exec时就关闭)
- SO_SNDBUF
- SO_RCVBUF
- SO_LINGER 设置套接口关闭后的行为
- TCP_NODELAY：禁用Nagle‘s Algorithm(积累数据量到TCP Segment Size后发送)
- SO_REUSEADDR：让端口释放后立即可以被再次使用

基于protobuf的rpc

```
option cc_generic_services = true;

message EchoRequest {
  optional string request = 1;
  optional int32 response_length = 2;
}

message EchoResponse {
  optional string response = 1;
}

service EchoService {
  rpc Echo(EchoRequest) returns (EchoResponse);
}
```

```
class EchoService_Stub : public EchoService {
 public:
  EchoService_Stub(::google::protobuf::RpcChannel* channel);
  EchoService_Stub(::google::protobuf::RpcChannel* channel,
                   ::google::protobuf::Service::ChannelOwnership ownership);
  ~EchoService_Stub();

  inline ::google::protobuf::RpcChannel* channel() { return channel_; }

  // implements EchoService ------------------------------------------

  void Echo(::google::protobuf::RpcController* controller,
                    const ::gdt::srpc::EchoRequest* request,
                    ::gdt::srpc::EchoResponse* response,
                    ::google::protobuf::Closure* done);
 private:
  ::google::protobuf::RpcChannel* channel_;
  bool owns_channel_;
  GOOGLE_DISALLOW_EVIL_CONSTRUCTORS(EchoService_Stub);
};
```

```
class EchoService : public ::google::protobuf::Service {
 protected:
  // This class should be treated as an abstract interface.
  inline EchoService() {};
 public:
  virtual ~EchoService();

  typedef EchoService_Stub Stub;

  static const ::google::protobuf::ServiceDescriptor* descriptor();

  virtual void Echo(::google::protobuf::RpcController* controller,
                    const ::gdt::srpc::EchoRequest* request,
                    ::gdt::srpc::EchoResponse* response,
                    ::google::protobuf::Closure* done);

  // implements Service ----------------------------------------------

  const ::google::protobuf::ServiceDescriptor* GetDescriptor();
  void CallMethod(const ::google::protobuf::MethodDescriptor* method,
                  ::google::protobuf::RpcController* controller,
                  const ::google::protobuf::Message* request,
                  ::google::protobuf::Message* response,
                  ::google::protobuf::Closure* done);
  const ::google::protobuf::Message& GetRequestPrototype(
    const ::google::protobuf::MethodDescriptor* method) const;
  const ::google::protobuf::Message& GetResponsePrototype(
    const ::google::protobuf::MethodDescriptor* method) const;

 private:
  GOOGLE_DISALLOW_EVIL_CONSTRUCTORS(EchoService);
};
```

SRpc: Simple rpc

- 代码走读
  - Svn link: http://tc-svn.tencent.com/isd/isd_qzonebrand_rep/qzone_adsdev_proj/trunk/app/qzap/service/dynamic_creative/srpc
- 怎么改进？
  - Epoll
  - 多线程
  - 异步&回调
  - 适配更多协议(http,qzone,ckv)
  - 增加更多功能(压缩, built-in service)

Epoll

- 常见的多路复用有：PPC(Process Per Connection)，TPC(Thread PerConnection)，这些模型的缺陷是： resource usage and context-switching time influence the ability to handle many clients at a time。
- select的缺点：
  - 最大并发数限制。一个进程所打开的FD（文件描述符）是有限制的，由FD_SETSIZE设置。
  - 效率问题，select每次调用都会线性扫描全部的FD集合。O(n)复杂度。
  - 内核/用户空间的内存拷贝问题。通过内存拷贝让内核把FD消息通知给用户空间。
- poll解决了第一个缺点，但第二，三个缺点依然存在。
- epoll是一个相对完美的解决方案。(1)最大FD个数很大(由/proc/sys/fs/file-max给出)；(2)epoll不仅会告诉应用程序有I/0事件到来，还会告诉应用程序相关的信息，不用遍历；(3)内核与用户态传递消息使用共享内存；
- level triggered vs edge triggered：edge-trigger模式中，epoll_wait仅当状态发生变化的时候才获得通知(即便缓冲区中还有未处理的数据)；而level-triggered模式下，epoll_wait只要有数据，将不断被触发。

Epoll与回调

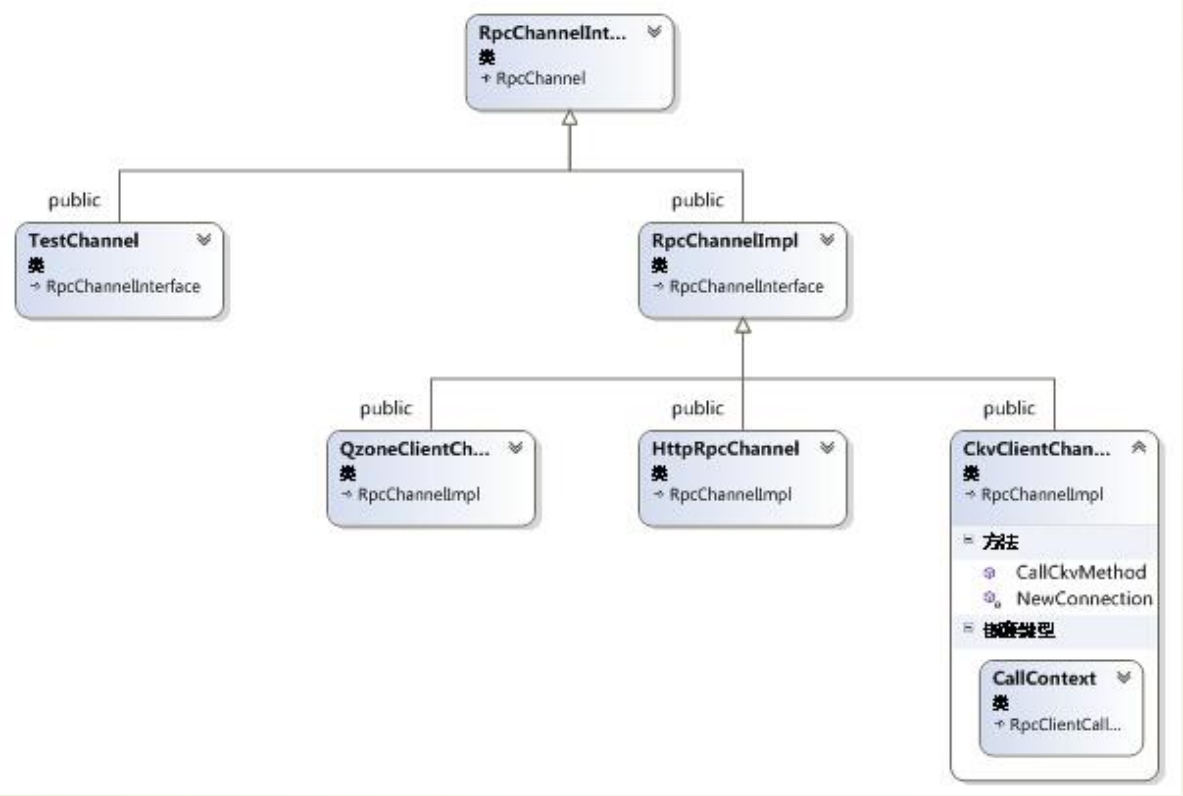
- 代码走读

```
int nfds = epoll_wait(epoll_fd_, events, kEpollLen, 100);  //
for (int i=0; i < nfds; ++i) {
  if (events[i].data.fd == listen_fd_) {
    int new_fd = accept(listen_fd_,
      reinterpret_cast<sockaddr*>(&client_sin), &addrlen);
    if (new_fd < 0) { continue; }
    struct epoll_event ev;
    ev.data.fd = new_fd;
    ev.events = EPOLLIN|EPOLLET;
    epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, new_fd, &ev);
  } else if (events[i].events & EPOLLIN) {
    int new_fd = events[i].data.fd;
    if (SyncCallMethod(new_fd) <= 0) {
      VLOG(1) << "client close fd:" << new_fd;
      close(new_fd);
    }
  } else if (events[i].events & EPOLLOUT) {
    LOG(ERROR) << "not recognized, fd:" << events[i].data.fd;
    continue;
  }
```

Gdt rpc

- RpcClient:
  - 负责所有RpcChannel对象的管理和对服务器端应答的处理
- RpcChannel:
  - 代表通讯通道，每个服务器地址对应于一个RpcChannel对象，客户端通过它向服务器端发送方法调用请求并接收结果。
- RpcController:
  - 存储一次rpc方法调用的上下文，包括对应的连接标识，方法执行结果等。
- RpcServer:
  - 服务器端的具体业务服务对象的容器，负责监听和接收客户端的请求，分发并调用实际的服务对象方法。

怎么处理qzone,http,ckv



怎么处理qzone,http,ckv

- Qzone协议：
  - Header: qzone_adsdev_proj/trunk/base_class_old/include/qzone_protocol.h
  - 利用qzone_protocol_version， qzone_protocol_cmd查找对应方法实现
- Http协议：
  - Header: 包含method full name，例如chorus.DbRpcService.GetQueryImp
  - Body: protobuf二进制化 or 文本化 or json
- CKV协议：
  - rpc_channel_impl封装了socket send和read的方法，只需要使用方提供buffer数据。那ckv这里需要做的是，兼容现有接口，实现EncodeRequest，OnPacketReceived方法即可。

http-rpc协议格式

Request

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = http
:path = /google.pubsub.v2.PublisherService/CreateTopic
:authority = pubsub.googleapis.com
grpc-timeout = 1S
content-type = application/grpc+proto
grpc-encoding = gzip
authorization = Bearer y235.wef315yfh138vh31hv93hv8h3v

DATA (flags = END_STREAM)
<Delimited Message>
```

Response

```
HEADERS (flags = END_HEADERS)
:status = 200
grpc-encoding = gzip

DATA
<Delimited Message>

HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0 # OK
trace-proto-bin = jher831yy13JHy3hc
```

怎么处理jce

- Jce：
  - client先转成proto，接收响应后再转成jce。

```
bool JceStructToPbMessage(const QZAP_NEW::wup_qzap_search_display_req* jce,
                          MixerRequest* request);
bool PbMessageToJceStruct(const MixerResponse* response,
                          QZAP_NEW::wup_qzap_search_display_rsp* jce);

GDT_RPC_DEFINE_JCE_PROTO_SERVICE(0, JceMixerService, MixerService,
  GDT_RPC_JCE_METHOD(0, SearchAd, QZAP_NEW::wup_qzap_search_display_req,
                     QZAP_NEW::wup_qzap_search_display_rsp)
```

利用rpc实现state,action

```
class RpcStateRunner {
 public:
  static RpcStateRunner* Create() {
    return new RpcStateRunner;
  }
  ~RpcStateRunner() {}
  void RunState(RpcContext* context,
                RpcState* state,
                std::string* next_state_name,
                Closure* done);

 private:
  RpcStateRunner()
    : context_(NULL), state_(NULL),
      next_state_name_(NULL), done_(NULL) {}
  void HandleStateDone();

 private:
  RpcContext* context_;
  RpcState* state_;
  std::string* next_state_name_;
  Closure* done_;
  std::vector<shared_ptr<RpcAction> > state_actions_;
};
```

```
class RpcActionRunner {
 public:
  static RpcActionRunner* Create() {
    return new RpcActionRunner();
  }
  ~RpcActionRunner() {}
  void RunAction(RpcContext* context,
                 RpcAction* action,
                 Closure* done);

 private:
  RpcActionRunner() : context_(NULL), action_(NULL), done_(NULL) {}
  void HandleActionDone();

 private:
  RpcContext* context_;
  RpcAction* action_;
  Closure* done_;
};
```

Gdt common库介绍

- 简要介绍 & 代码走读
- Base: 单例，智能指针，类注册，回调，字符串处理等
- Cache，Collection(prefix_map)
- Net: uri，http，hdfs
- Spp: channel, proxy
- System: 线程，锁，socket，内存池，时间等
- Rpc

grpc

- https://github.com/google/protobuf/wiki/Third-Party-Add-ons
- http://www.grpc.io/
- 代码走读

Thanks