精通protobuf原理系列之（一）

为什么要使用以及如何使用

精通protobuf原理之一：为什么要使用以及如何使用

津的技术专栏
后端架构师
关注他

目录

1 说在前面

因为看到网上都是一些零零散散的 protobuf 相关介绍，加上笔者最近因为项目的原因深入剖析了 protobuf，所以想做一个系统的《精通 protobuf 原理》系列的分享：

- 「精通 protobuf 原理之一：为什么要使用它以及如何使用」；
- 「精通 protobuf 原理之二：编码原理剖析」；
- 「精通 protobuf 原理之三：反射原理剖析」；
- 「精通 protobuf 原理之四： RPC 原理剖析」；
- 「精通 protobuf 原理之五： Arena 分配器原理剖析」。
- 后续的待定……

本文是系列文章的第一篇，阅读了本文，读者可以了解到：

1. 为什么要使用 protobuf ，而不使用 xml、json 等其他数据结构化标准；
2. 在 centos7 下怎么编译安装 protobuf 以及可能遇到哪些安装问题；
3. 如何将 proto IDL 文件生成 C++ 源代码；
4. protobuf 普通数据接口如何使用；
5. protobuf 的反射是什么？以及反射接口如何使用；
6. protobuf 的 RPC 接口有什么用处？以及如何使用。

阅读本文大概需要十分钟左右。建议读者先阅读目录，先大概了解有哪些内容，然后在选择全部阅读还是选择性阅读，以提高阅读效率。

2 为什么使用protobuf

为什么要使用 protobuf ？先说说 protobuf 问世的目的是解决什么问题。

protobuf (protocol buffer) 是谷歌内部的混合语言数据标准。通过将结构化的数据进行序列化，用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。其实就是和 xml、json 做的类似的事情。那么问题又来了，为什么不选择使用 xml、json，而要选择 protobuf 呢？先通过以下表格做一个比较：

特性 \ 类型	xml	json	protobuf
数据结构支持	简单结构	简单结构	复杂结构
数据保存方式	文本	文本	二进制

总结下来就是，使用 protobuf 能够多（数据结构支持、语言支持程度）、快（编解码效率）、好（数据保存方式）、省（数据保存大小）。

- [多]：业务场景中，不免可能有比较复杂的数据结构，对于扩展性没有后顾之忧；覆盖了主流的编程语言，在一定程度上减少了自研成本，开发者能够轻松上手；
- [快]：快是一个非常重要的系统性能指标；
- [好]：使用二进制对数字类型更节省空间、读取转换时间，因为数字转换成文件占用的字节数比较多，字符串和数字之间的转换也比较耗时；
- [省]：当海量数据都需要存储在 redis 内存中的时候，节省空间又多重要；当网络带宽有限的情况下，节省带宽有多重要。

3 编译环境

操作系统： `CentOS Linux release 7.9.2009 (Core)`

编译器版本： `gcc version 4.8.5 20150623 (Red Hat 4.8.5-44) (GCC)`

protobuf 版本： `3.17.3`

4 系统依赖包

```
$ yum install gcc-c++ make autoconf automake
```

5 下载源码

```
$ git clone https://github.com/protocolbuffers/protobuf.git
$ cd protobuf/
$ git checkout v3.17.3
```

6 编译安装

6.1 编译方法

```
# 生成 confiure 文件
$ ./autogen.sh
# 执行 confiure 文件，prefix 默认也是 /usr/local
$ ./configure CXXFLAGS="-fPIC -std=c++11" --prefix=/usr/local
# 执行 make
$ make -j 4
```

6.2 安装方法

```
$ make install
# bin安装目录
$ ls /usr/local/bin/
protoc
# lib安装目录
$ ls /usr/local/lib
libprotobuf-lite.a   libprotobuf-lite.so.28      libprotobuf.la     libprotobuf.so.28.
libprotobuf-lite.la  libprotobuf-lite.so.28.0.3  libprotobuf.so     libprotoc.a
libprotobuf-lite.so  libprotobuf.a               libprotobuf.so.28  libprotoc.la
```

6.3 submodule依赖

以上的编译安装，没有用到 submodule。但是如果需要执行单元测试和性能测试，就会用到（见 `tests.sh` ）。C++ 版本只需要执行如下命令：

```
$ ./tests.sh cpp
```

看看这个命令做了什么（见 `build_cpp` 函数）：

```
...
internal_build_cpp() {
  if [ -f src/protoc ]; then
    # Already built.
    return
  fi

  # Initialize any submodules.
  git submodule update --init --recursive

  ./autogen.sh
  ./configure CXXFLAGS="-fPIC -std=c++11"  # -fPIC is needed for python cpp test.
                                           # See python/setup.py for more details
  make -j$(nproc)
}

build_cpp() {
  internal_build_cpp
  make check -j$(nproc) || (cat src/test-suite.log; false)
  cd conformance && make test_cpp && cd ..

  # The benchmark code depends on cmake, so test if it is installed before
  # trying to do the build.
  if [[ $(type cmake 2>/dev/null) ]]; then
    # Verify benchmarking code can build successfully.
    cd benchmarks && make cpp-benchmark && cd ..
  else
    echo ""
    echo "WARNING: Skipping validation of the bench marking code, cmake isn't installe
    echo ""
  fi
}
...
```

> `build_cpp` 做了以下事情：

1. 通过 git submodule 命令下载第三方依赖；
2. 执行 autogen.sh 脚本生成 configure`；
3. 执行 configure 生成 Makefile`；
4. 根据 Makefile 执行 make 编译；
5. 编译和执行单元测试用例；
6. 编译 benchmarks。

通过 `.gitmodules` 文件可以看到 protobuf 依赖 benchmark（性能测试框架） 和googletest（单元测试框架）两个第三方模块。

```
    path = third_party/benchmark
    url = https://github.com/google/benchmark.git
[submodule "third_party/googletest"]
    path = third_party/googletest
    url = https://github.com/google/googletest.git
    ignore = dirty
```

6.4 常见编译问题

6.4.1 没有安装 autoconf 包

```
+ test -d third_party/googletest
+ mkdir -p third_party/googletest/m4
+ autoreconf -f -i -Wall,no-obsolete
autogen.sh: line 41: autoreconf: command not found
```

6.4.2 没有安装 automake包

```
+ test -d third_party/googletest
+ mkdir -p third_party/googletest/m4
+ autoreconf -f -i -Wall,no-obsolete
Can't exec "aclocal": No such file or directory at /usr/share/autoconf/Autom4te/FileUt
autoreconf: failed to run aclocal: No such file or directory
```

7 开发中使用

7.1 生成源代码

根据 proto IDL 文件生成 C++ 源代码。

```
$ tree proto/
proto/
├── Makefile
└── echo.proto
```

定义一个 proto IDL 文件：

```
//指定proto版本
syntax = "proto3";
//制定命名空间
package self;

//告诉proto编译器生成service接口
option cc_generic_services = true;

//枚举定义
enum QueryType {
        PRIMMARY = 0;
        SECONDARY = 1;
};

//message定义
message EchoRequest {
        QueryType querytype = 1;
```

```
        int32 code = 1;
        string msg = 2;
}

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

Makefile 源文件：

```
CC = g++
CXXFLAGS = -std=c++11
TARGET = libproto.a
SOURCE = $(wildcard *.cc)
OBJS = $(patsubst %.cc, %.o, $(SOURCE))
INCLUDE = -I./

$(TARGET): $(OBJS)
        ar rcv $(TARGET) $(OBJS)

%.o: %.c
        protoc -I=./ --cpp_out=./ ./echo.proto
        $(CC) $(CXXFLAGS) $(INCLUDE) -o $@ -c $^

.PHONY:clean
clean:
        rm *.o $(TARGET)
```

执行 make 生成 C++ 源代码

```
$ make -C proto/
$ tree proto/
proto/
├── Makefile
├── echo.pb.cc
├── echo.pb.h
└── echo.proto
```

7.2 使用源代码

```
$ tree test_echo
test_echo
├── Makefile
|── general.cpp
├── reflection.cpp
├── rpc.cpp
└── test_echo.cpp
```

test_echo.cpp 源文件

```
#include <iostream>
#include <string>
#include "../proto/echo.pb.h"

extern void test_general();
extern void test_relection();
extern void test_rpc();
```

```
        test_rpc();
        return 0;
}
```

Makefile 源文件

```
CC = g++
CXXFLAGS = -std=c++11
TARGET = test_echo
SOURCE = $(wildcard *.cpp)
OBJS = $(patsubst %.cpp, %.o, $(SOURCE))
INCLUDE = -I./
LIBS = -lproto -lprotobuf
LIBPATH = -L../proto

$(TARGET): $(OBJS)
        $(CC) $(CXXFLAGS) -o $@ $^ $(LIBPATH) $(LIBS)

%.o: %.c
        protoc -I=./ --cpp_out=./ ./echo.proto
        $(CC) $(CXXFLAGS) $(INCLUDE) -o $@ -c $^

.PHONY:clean
clean:
        rm -f *.o $(TARGET)
```

编译和执行

```
$ make
$ ./test_echo
=== START TEST GENERAL ===
req.querytype[1], req_rcv.querytype[1]
req.payload[this is a payload], req_rcv.payload[this is a payload]
=== END TEST GENERAL ===

=== START TEST REFLECTION ===
type_name: self.EchoRequest
ref_req_msg_payload:
ref_req_msg_payload: my payload
=== END TEST REFLECTION ===

=== END TEST RPC ===
  === START RPC SERVER ===
MyEchoServiceImpl::recieve request|I have received <querytype:{1}, payload:{rpc_server
MyEchoServiceImpl::OnCallbak: response|<code:0, msg:I have received <querytype:{1}, pa
  === END RPC SERVER ===
  === START RPC CLIENT ===
rpc_client::response<code:0,msg:I have sent <querytype:{1}, payload:{rpc_client::reque
  === END RPC CLIENT ===
=== END TEST RPC ===
```

可能遇到问题：

```
$ ./test_echo
./test_echo: error while loading shared libraries: libprotobuf.so.28: cannot open shar
```

因为 protobuf 安装目录为 `/usr/local/lib` ， 不在操作系统默认 `lib` 目录中（操作系统默

```
[root@af82601d9d63 test_echo]# cat /etc/ld.so.conf.d/usr_local_lib.conf
/usr/local/lib
```

还有一种方法是 `export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/lib` ，写在 `~/.bash_profile` 或者 `~/.bashrc` 配置文件中，每次登录用户即时生效。

7.3 普通接口

见 `general.cpp` 源文件。

```
extern void test_general() {
        std::cout << "=== START TEST GENERAL ===" << std::endl;
        self::EchoRequest req;
        req.set_querytype(self::SECONDARY);
        req.set_payload("this is a payload");

        std::string req_body;
        req.SerializeToString(&req_body);

        self::EchoRequest req_rcv;
        req_rcv.ParseFromString(req_body);
        std::cout << "req.querytype[" << req.querytype() << "], "
                        << "req_rcv.querytype[" << req_rcv.querytype() << "]" << std
                        << "req.payload[" << req.payload() << "], "
                        << "req_rcv.payload[" << req_rcv.payload() << "]"<< std::end
        std::cout << "=== END TEST GENERAL ===" << std::endl << std::endl;
}
```

开发中经常使用的是读写field（即get/set），序列化（SerializeToString）和反序列化（ParseFromString）。

7.4 反射接口

见 `reflection.cpp` 源文件。

```
#include "../proto/echo.pb.h"

void test_relection() {
        std::cout << "=== START TEST REFLECTION ===" << std::endl;
        std::string type_name = self::EchoRequest::descriptor()->full_name();
        std::cout << "type_name: " << type_name << std::endl;

        const google::protobuf::Descriptor* descriptor
                = google::protobuf::DescriptorPool::generated_pool()->FindMessageTypeB
        const google::protobuf::Message* prototype
                = google::protobuf::MessageFactory::generated_factory()->GetPrototype(
        google::protobuf::Message* req_msg = prototype->New();
        const google::protobuf::Reflection* req_msg_ref
                = req_msg->GetReflection();
        const google::protobuf::FieldDescriptor *req_msg_ref_field_payload
                = descriptor->FindFieldByName("payload");

        std::cout << "ref_req_msg_payload: "
                        << req_msg_ref->GetString(*req_msg, req_msg_ref_field_payloa
                        << std::endl;
        req_msg_ref->SetString(req_msg, req_msg_ref_field_payload, "my payload");
```

```
}
```

既然已经有了 get/set 的读写 API，为什么还需要反射呢?

使用场景比如：在推荐系统中，用户特征使用 protobuf 格式存储，每个用户有成百上千个特征（一个特征可以理解成一个字段），在建模的时候可以只需要这些特征的几个或者几十个特征即可。需求如下：

1. 选择这些特征；
2. 通过指定的函数对这些特征做数据转换，输出指定格式的结果。

如果使用 get/set 读写这些特征，那是不是每个模型都要写一遍实现代码（因为读写的特征字段不一样）。可以可以做到这样，给定一个配置文件，格式如下：

```
[第一列] [第二列]
特征字段  转换函数(即算子)
```

通过配置特征字段和其转换函数，程序自动进行字段、转换函数的选择，并执行转换和输出结果。这个时候 protobuf 的反射功能就派上用场了。

这里简单介绍了一下反射功能的使用背景，具体的实现原理将在后续系列文章中专门介绍。

7.5 RPC接口

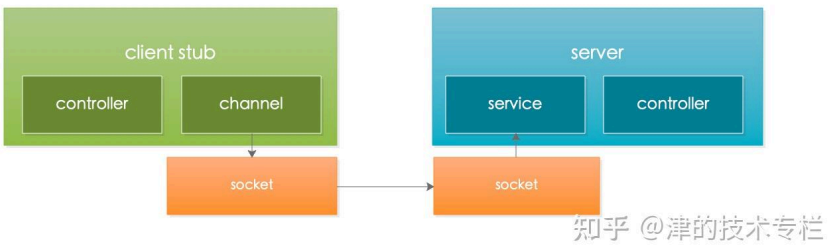
见 `rpc.cpp` 源文件。

protobuf 提供了一个 rpc 接口规范，可以使用它来定义接口格式，如下方式：

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

`EchoService` 是一个服务抽象， `Echo` 是该服务的方法，也可以理解成接口。

可不可以不使用 protobuf 的 `rpc` 接口规范? 当然可以。 protobuf 的 `rpc` 和 `message` 并不是强制绑定的，开发者可以选择使用只使用 `message` 或者使用 `rpc+message` 。这是谷歌内部沉淀的一个基于 protobuf 的 `rpc` 设计模式，笔者觉得这是一个很好的设计模式，建议使用。



RPC 交互图

7.5.1 服务端接口实现

```
static void rpc_server() {
  std::cout << "  === START RPC SERVER ===" << std::endl;
  MyEchoServiceImpl svc;
```

```
  request.set_querytype(self::SECONDARY);
  request.set_payload("rpc_server::request::payload");

  auto req_msg = dynamic_cast<google::protobuf::Message*>(&request);
  auto rsp_msg = dynamic_cast<google::protobuf::Message*>(&response);

  google::protobuf::Closure* done
        = google::protobuf::NewCallback(&svc,
            &MyEchoServiceImpl::OnCallbak, //指定回调函数，执行done->Run()的时候触发回调
            req_msg, //回调函数的第一个参数
            rsp_msg); //回调函数的第二个参数

  svc.Echo(&cntl, &request, &response, done);  //调用处理逻辑
  std::cout << "  === END RPC SERVER ===" << std::endl;
}
```

当服务端收到请求之后，会通过 `svc.Echo` 调用处理流程。是的， `svc` 实现的就是 `EchoService` 的 `Echo rpc` 接口，如下实现：

```
class MyEchoServiceImpl: public self::EchoService {
  public:
    virtual void Echo(google::protobuf::RpcController* cntl,
                      const self::EchoRequest* request,
                      self::EchoResponse* response,
                      google::protobuf::Closure* done) override {
      std::ostringstream oss;
      oss << "I have received <querytype:{" << request->querytype()
          << "}, payload:{" << request->payload() << "}";
      std::string rcv = oss.str();
      std::cout << "MyEchoServiceImpl::recieve request|" << rcv << std::endl;
      response->set_code(0);
      response->set_msg(rcv);
      done->Run(); //记得调用 Run 才能触发 OnCallback 操作。
  }
  void OnCallbak(google::protobuf::Message* request,
                 google::protobuf::Message* response) {
          std::cout << "MyEchoServiceImpl::OnCallbak: response|<code:"
                    << dynamic_cast<self::EchoResponse*>(response)->code() << ", msg:"
                        << dynamic_cast<self::EchoResponse*>(response)->msg() << ">"
                        << std::endl;
  }
};
```

7.5.2 客户端接口实现

```
static void rpc_client() {
  std::cout << "  === START RPC CLIENT ===" << std::endl;
  MyRpcControllerImpl cntl;
  self::EchoRequest request;
  self::EchoResponse response;

  request.set_querytype(self::SECONDARY);
  request.set_payload("rpc_client::request::payload");

  MyRpcChannelImpl channel;
  channel.init();
  self::EchoService_Stub stub(&channel);
  stub.Echo(&cntl, &request, &response, nullptr);
  std::cout << "rpc_client::response<code:" << response.code()
            << ",msg:" << response.msg() << ">" << std::endl;
```

stub 接收了 channel 参数，在执行 stub.Echo 的时候实际上是调用 channel 的 CallMethod 接口发送请求，如下实现：

```
class MyRpcChannelImpl: public google::protobuf::RpcChannel {
  public:
        void init() {}
    virtual void CallMethod(const google::protobuf::MethodDescriptor* method,
                            google::protobuf::RpcController* controller,
                            const google::protobuf::Message* request,
                            google::protobuf::Message* response,
                            google::protobuf::Closure* done) override {
      auto req = dynamic_cast<self::EchoRequest*>(
                             const_cast<google::protobuf::Message*>(request));
      auto rsp = dynamic_cast<self::EchoResponse*>(response);
      std::ostringstream oss;
      oss << "I have sent <querytype:{" << req->querytype()
          << "}, payload:{" << req->payload() << "}";
      std::string rcv = oss.str();
      rsp->set_code(0);
      rsp->set_msg(rcv);
    }
};
```

7.5.3 控制器 Controller

控制器提供了 Reset 、 Failed 、 ErrorText 、 StartCancel 、 SetFailed 、 IsCanceled 、 NotifyOnCancel 六个接口，主要是为了控制、操作、获取请求的状态。

```
class MyRpcControllerImpl: public google::protobuf::RpcController {
  public:
    virtual void Reset() override {
      std::cout << "MyRpcController::Reset" << std::endl;
    }
    virtual bool Failed() const override {
      std::cout << "MyRpcController::Failed" << std::endl;
      return false;
    }
    virtual std::string ErrorText() const override {
      std::cout << "MyRpcController::ErrorText" << std::endl;
      return "";
        }
    virtual void StartCancel() override {
      std::cout << "MyRpcController::StartCancel" << std::endl;
    }
    virtual void SetFailed(const std::string& reason) override {
      std::cout << "MyRpcController::SetFailed" << std::endl;
    }
    virtual bool IsCanceled() const override {
      std::cout << "MyRpcController::IsCanceled" << std::endl;
      return false;
    }
    virtual void NotifyOnCancel(google::protobuf::Closure* callback) override {
      std::cout << "MyRpcController::NotifyOnCancel" << std::endl;
    }
  //private:
    //bool concel_ = false;
    //std::string err_reason_;
};
```

9 说在最后

以上就是系列文章第一篇的所有内容。通过本文，读者应该已经了解在日常项目开发中如何使用 protobuf ，也对能够使用 protobuf 做什么有了一个初步的了解。从下一篇开始，将是对 protobuf 原理方面的一些介绍，如果说本文是告诉读者如何使用 protobuf ，那么后面的系列文章将会帮助读者怎么用好 protobuf 。

感谢阅读，如果还想了解更多的内容，请在评论区留言。

欢迎学习交流，也欢迎指正。

发布于 2022-12-11 16:58 · IP 属地广东

内容所属专栏