# Protoc工作原理分析

Posted May 19, 2017 by  - **31 min read**

分享:

---

在进行protoc插件开发之前，首先要了解protoc的工作原理。在protobuf的使用过程中，protoc作为proto文件的编译器，很多开发人员只会用但是不了解其工作原理，更不了解如何扩展其功能。protobuf作为目前常用的数据交换格式在协议开发中被广泛采用，此外，protoc对proto文件的强大解析能力使我们可以开发一些插件，通过插件快速生成特定于proto文件的工具类、配置文件等，从而提高开发效率。

本文首先会介绍一下protoc的整体工作机制，然后解释一下protoc对proto文件的解析过程，最后给出编写protoc插件扩展protoc功能的一个示例教程。

## 1. protoc源代码准备#

要想了解protoc的工作机制，查看其源代码了解其核心流程是最靠谱的方法。

获取程序源代码的方式如下：

```
git co https://github.com/google/protobuf
```

由于我们工程中常用的protoc版本是v2.5.0，所以这里检出对应版本的tag。

```
git ck v2.5.0
```

考虑到可能会进行测试、修改、注释等学习过程，这里最好创建一个新的分支来操作。

```
git branch ${new-branch-name}
git ck ${new-branch-name}
```

现在源代码准备好了，我比较喜欢使用vim、ctags、cscope来阅读源码，根据个人习惯吧，下面可以阅读protoc的源码梳理以下工作机制。

## 2. protoc源码分析#

上述git检出后的protobuf路径，记为${protobuf}，后面如果出现${protobuf}请知晓其含义。如果在描述源代码时没有提及起始路径${protobuf}，那么起始路径均为${protobuf}。

### 2.1. protoc程序入口#

src/google/protobuf/compiler/main.cc中的main函数，为protoc的入口函数。

file: src/google/protobuf/compiler/main.cc

```
// Author: kenton@google.com (Kenton Varda)

// 这个头文件定义了protoc的命令行接口
#include <google/protobuf/compiler/command_line_interface.h>

// protoc中内置了对cpp、python、java语言的支持，对其他语言的支持需要以plugin的方式来支持
#include <google/protobuf/compiler/cpp/cpp_generator.h>
#include <google/protobuf/compiler/python/python_generator.h>
#include <google/protobuf/compiler/java/java_generator.h>


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

  // 初始化protoc命令行接口并开启插件
  // - 插件只是普通的可执行程序，其文件名以AllowPlugins参数protoc-开头
  // - 假定protoc --foo_out，那么实际调用的插件是protoc-foo
  google::protobuf::compiler::CommandLineInterface cli;
  cli.AllowPlugins("protoc-");

  // Proto2 C++ (指定了--cpp_out将调用cpp::Generator)
  google::protobuf::compiler::cpp::CppGenerator cpp_generator;
  cli.RegisterGenerator("--cpp_out", "--cpp_opt", &cpp_generator,
                        "Generate C++ header and source.");

  // Proto2 Java (指定了--java_out将调用java::Generator)
  google::protobuf::compiler::java::JavaGenerator java_generator;
  cli.RegisterGenerator("--java_out", &java_generator,
                        "Generate Java source file.");

  // Proto2 Python (指定了python_out将调用python::Generator)
  google::protobuf::compiler::python::Generator py_generator;
  cli.RegisterGenerator("--python_out", &py_generator,
                        "Generate Python source file.");

  // 解析proto、生成源代码(借助内置的generator或者plugins)、创建源代码文件
  return cli.Run(argc, argv);
}
```

### 2.2. protoc命令行接口定义#

下面看一下protoc的命令行接口定义，以了解其对命令行
flags、options的解释过程以及对后续程序执行逻辑的影响。

file:
src/google/protobuf/compiler/command_line_interface.h

```
// Author: kenton@google.com (Kenton Varda)
//  Based on original Protocol Buffers design by
//  Sanjay Ghemawat, Jeff Dean, and others.
//
// Implements the Protocol Compiler front-end such that it may be reused by
// custom compilers written to support other languages.

#ifndef GOOGLE_PROTOBUF_COMPILER_COMMAND_LINE_INTERFACE_H__
#define GOOGLE_PROTOBUF_COMPILER_COMMAND_LINE_INTERFACE_H__

#include <google/protobuf/stubs/common.h>
#include <string>
#include <vector>
#include <map>
#include <set>
#include <utility>
```

```
namespace google {
namespace protobuf {

//proto文件中定义的数据类型可通过FileDescriptor来遍历、查看
class FileDescriptor;        // descriptor.h
class DescriptorPool;        // descriptor.h
class FileDescriptorProto;   // descriptor.pb.h
template<typename T> class RepeatedPtrField;  // repeated_field.h

namespace compiler {

class CodeGenerator;        // code_generator.h
class GeneratorContext;     // code_generator.h
class DiskSourceTree;       // importer.h

// 这个类实现了protoc的命令行接口，使得protoc很容易扩展。
// 例如我们想让protoc既支持cpp又支持另一种语言foo，那么我们可以定义一个实现了
// CodeGenerator接口的FooGenerator，然后在protoc的main方法中对这两种语言cpp、Foo
// 及其对应的CodeGenerator进行注册。
//
//   int main(int argc, char* argv[]) {
//     google::protobuf::compiler::CommandLineInterface cli;
//
//     // 支持cpp
//     google::protobuf::compiler::cpp::CppGenerator cpp_generator;
//     cli.RegisterGenerator("--cpp_out", &cpp_generator, "Generate C++ source and header.");
//
//     // 支持foo
//     FooGenerator foo_generator;
//     cli.RegisterGenerator("--foo_out", &foo_generator, "Generate Foo file.");
//
//     return cli.Run(argc, argv);
//   }
//
// The compiler is invoked with syntax like:
//   protoc --cpp_out=outdir --foo_out=outdir --proto_path=src src/foo.proto
//
// For a full description of the command-line syntax, invoke it with --help.
class LIBPROTOC_EXPORT CommandLineInterface {
 public:
  CommandLineInterface();
  ~CommandLineInterface();

  // 为某种编程语言注册一个对应的代码生成器（其实这里也不一定非得是语言）
  //
  // 命令行接口的参数:
  // @param flag_name 指定输出文件类型的命令，例如--cpp_out，参数名字必须以“-”开头，
                      如果名字大于两个字符，则必须以“--”开头。
  // @param generator 与flag_name对应的CodeGenerator接口实现
  // @param help_text 执行protoc --help的时候对这里的flag_name的说明性信息
  //
  // 某些代码生成器可接受额外参数，这些参数在输出路径之前给出，与输出路径之间用“:”分隔。
  //   protoc --foo_out=enable_bar:outdir
  // 这里的:outdir之前的enable_bar被作为参数传递给CodeGenerator::Generate()的参数。
  void RegisterGenerator(const string& flag_name,
                         CodeGenerator* generator,
                         const string& help_text);

  // 为某种编程语言注册一个对应的代码生成器
  // ...
  // @param option_flag_name 指定额外的选项
```

```
// ...
//
// 与前面一个函数RegisterGenerator所不同的是，这个重载函数多个参数
// option_flag_name，通过这个函数注册的语言和代码生成器可以接受额外的参数。例
// 如通过command_line_interface.RegisterGenerator("--foo_out", "--foo_opt", ...)
// 注册了foo以及对应代码生成器，那么我们可以在执行protoc 的时候指定额外的参数
// --foo_opt：protoc --foo_out=enable_bar:outdir --foo_opt=enable_baz，此时传
// 递给代码生成器的参数将会包括enable_bar和enable_baz。
void RegisterGenerator(const string& flag_name,
                       const string& option_flag_name,
                       CodeGenerator* generator,
                       const string& help_text);

// RegisterGenerator方法是在protoc的main方法中进行语言、代码生成器的注册，在
// 生产环境中不可能允许开发人员肆意修改公用程序库，这意味着我们如果要在稳定地
// protoc v2.5.0基础上进行源码的修改这条路是行不通的，那么如何自由地扩展其功
// 能呢？protoc提供了“plugin”机制，我们可以通过自定义插件来实现对其他语言
// （甚至不是语言）的支持。
// 开启protoc对插件的支持，这种模式下，如果一个命令行选项以_out结尾，例如
// --xxx_out，但是在protoc已经注册的语言支持中没有找到匹配的语言及代码生成器，
// 这个时候protoc就会去检查是否有匹配的插件支持这种语言，将这个插件来作为代
// 码生成器使用。这里的的protoc插件是一个$PATH中可搜索到的可执行程序，当然
// 这个可执行程序稍微有点特殊。
// 这里插件名称（可执行程序名称）是如何确定的呢？选项--xxx_out中，截取“xxx”，
// 然后根据${exe_name_prefix}以及xxx来拼接出一个插件的名字，假如${exe_name_prefix}
// 是protoc-，那么插件的名字就是protoc-xxx，protoc将尝试执行这个程序来完成代
// 码生成的工作。
// 假定插件的名字是plugin，protoc是这样调用这个插件的：
//   plugin [--out=OUTDIR] [--parameter=PARAMETER] PROTO_FILES < DESCRIPTORS
// 选项说明：
// --out：指明了插件代码生成时的输出目录（跟通过--foo_out传递给protoc的一样），
//        如果省略这个参数，输出目录就是当前目录。
// --parameter：指明了传递给代码生成器的参数。
// PROTO_FILES：指明了protoc调用时传递给protoc的待处理的.proto文件列表。
// DESCRIPTORS：编码后的FileDescriptorSet（这个在descriptor.proto中定义），
// 这里编码后的数据通过管道重定向到插件的标准输入，这里的FileDescriptorSet包括
// PROTO_FILES中列出的所有proto文件的descriptors，也包括这些PROTO_FILES中
// proto文件import进来的其他proto文件。插件不应该直接读取PROTO_FILES中的
// proto文件，而应该使用这里的DESCRIPTORS。
//
// 插件跟protoc main函数中注册的代码生成器一样，它也需要生成所有必须的文件。
// 插件会将所有要生成的文件的名字写到stdout，插件名字是相对于当前输出目录的。如
// 果插件工作过程中发生了错误，需要将错误信息写到stderr，如果发生了严重错误，
// 插件应该退出并返回一个非0的返回码。插件写出的数据会被protoc读取并执行后续
// 处理逻辑。
void AllowPlugins(const string& exe_name_prefix);

// 根据指定的命令行参数来执行protocol compiler，返回值将由main返回。
//
// Run()方法是非线程安全的，因为其中调用了strerror()，不要在多线程环境下使用。
int Run(int argc, const char* const argv[]);

// proto路径解析的控制说明，fixme
// Call SetInputsAreCwdRelative(true) if the input files given on the command
// line should be interpreted relative to the proto import path specified
// using --proto_path or -I flags.  Otherwise, input file names will be
// interpreted relative to the current working directory (or as absolute
// paths if they start with '/'), though they must still reside inside
// a directory given by --proto_path or the compiler will fail.  The latter
// mode is generally more intuitive and easier to use, especially e.g. when
// defining implicit rules in Makefiles.
```

```
  void SetInputsAreProtoPathRelative(bool enable) {
    inputs_are_proto_path_relative_ = enable;
  }

  // 设置执行protoc --version时打印的版本相关的信息，这行版本信息的下一行也会打印libprotoc的版本。
  void SetVersionInfo(const string& text) {
    version_info_ = text;
  }


 private:
  // ---------------------------------------------------------------

  // 这个类的后续部分代码，虽然也比较重要，但是即便在这里先不解释，也不会给我
  // 们的理解造成太多干扰，为了简化篇幅并且避免过早地陷入细节而偏离对整体的把
  // 握，这里我先把这个类的后续部分代码进行删减……只保留相对比较重要的。

  class GeneratorContextImpl;
  class MemoryOutputStream;

  // 清楚上次Run()运行时设置的状态
  void Clear();

  // 映射input_files_中的每个文件，使其变成相对于proto_path_中对应目录的相对路径
  // - 当inputs_are_proto_path_relative_为false的时候才会调用这个函数;
  // - 出错返回false，反之返回true;
  bool MakeInputsBeProtoPathRelative(DiskSourceTree* source_tree);

  // ParseArguments() & InterpretArgument()返回的状态
  enum ParseArgumentStatus {
    PARSE_ARGUMENT_DONE_AND_CONTINUE,
    PARSE_ARGUMENT_DONE_AND_EXIT,
    PARSE_ARGUMENT_FAIL
  };

  // 解析所有的命令行参数
  ParseArgumentStatus ParseArguments(int argc, const char* const argv[]);

  // 解析某个命令行参数
  // - 参数名放name，参数值放value
  // - 如果argv中的下一个参数应该被当做value则返回true，反之返回false
  bool ParseArgument(const char* arg, string* name, string* value);

  // 解析某个命令行参数的状态
  ParseArgumentStatus InterpretArgument(const string& name, const string& value);

  // 从输入的proto文件生成指定的源代码文件
  struct OutputDirective;

  // 对解析成功的每个proto文件，生成对应的源代码文件
  // @param parsed_files 解析成功的proto文件vector
  // @param output_directive 输出指示，包括了文件名、语言、代码生成器、输出目录
  // @param generator_context 代码生成器上下文，可记录待输出文件名、文件内容、尺寸等信息
  bool GenerateOutput(const vector<const FileDescriptor*>& parsed_files,
                      const OutputDirective& output_directive,
                      GeneratorContext* generator_context);

  // 对解析成功的每个proto文件，调用protoc插件生成对应的源代码
  // @param parsed_files 解析成功的proto文件vector
  // @param plugin_name 插件的名称，命名方式一般是protoc-gen-${lang}
  // @param parameter 传递给protoc插件的参数
```

```
// @param generator_context 代码生成器上下文，可记录待输出文件名、文件内容、尺寸等信息
// @param error 错误信息
bool GeneratePluginOutput(const vector<const FileDescriptor*>& parsed_files,
                          const string& plugin_name,
                          const string& parameter,
                          GeneratorContext* generator_context,
                          string* error);

// 编码、解码，实现命令行中的--encode和--decode选项
bool EncodeOrDecode(const DescriptorPool* pool);

// 实现命令行中的--descriptor_set_out选项
bool WriteDescriptorSet(const vector<const FileDescriptor*> parsed_files);

// 获取指定proto文件依赖的proto文件列表（列表中包括该proto文件本身）
// - proto文件通过FileDescriptorProto表示；
// - 这些依赖的proto文件列表会被重新排序，被依赖的proto会被排在依它的proto前面，
//   这样我们就可以调用DescriptorPool::BuildFile()来建立最终的源代码文件；
// - already_seen中已经列出的proto文件不会被重复添加，每一个被添加的proto文件都被加入到already_se
// - 如果include_source_code_info为true，则包括源代码信息到FileDescriptorProtos中；
static void GetTransitiveDependencies(const FileDescriptor* file,
                                      bool include_source_code_info,
                                      set<const FileDescriptor*>* already_seen,
                                      RepeatedPtrField<FileDescriptorProto>* output);

// -------------------------------------------------------------------

// 当前被调用的程序的名称，argv[0]
string executable_name_;

// 通过SetVersionInfo()设置的版本信息
string version_info_;

// 注册的代码生成器
struct GeneratorInfo {
  string flag_name;           // --foo_out
  string option_flag_name;    // --foo_opt
  CodeGenerator* generator;   // 对应的代码生成器
  string help_text;           // protoc --help时输出的--foo_out的帮助信息
};

typedef map<string, GeneratorInfo> GeneratorMap;
// flag_name、代码生成器map
GeneratorMap generators_by_flag_name_;
// option_name、代码生成器map
GeneratorMap generators_by_option_name_;

// flag_name、option map
// - 如果调用protoc --foo_out=outputdir --foo_opt=enable_bar ...,
//   map中将包括一个<--foo_out,enable_bar> entry.
map<string, string> generator_parameters_;

// protoc插件前缀，如果该变量为空，那么不允许使用插件
// @see AllowPlugins()
string plugin_prefix_;

// 将protoc插件名称映射为具体的插件可执行文件
// - 执行一个插件可执行程序时，首先搜索这个map，如果找到则直接执行；
// - 如果这个map中找不到匹配的插件可执行程序，则搜索PATH寻找可执行程序执行；
map<string, string> plugins_;
```

```
  // protoc命令行中指定的工作模式
  enum Mode {
    MODE_COMPILE,  // Normal mode:  parse .proto files and compile them.
    MODE_ENCODE,   // --encode:  read text from stdin, write binary to stdout.
    MODE_DECODE    // --decode:  read binary from stdin, write text to stdout.
  };

  Mode mode_;

  vector<pair<string, string> > proto_path_;  // Search path for proto files.
  vector<string> input_files_;                // Names of the input proto files.

  // protoc调用时每个--${lang}_out都对应着一个OutputDirective
  struct OutputDirective {
    string name;                // flag_name, E.g. "--foo_out"
    CodeGenerator* generator;   // 为NULL则表示使用protoc插件而非内置的代码生成器
    string parameter;           // 传递给代码生成器或者protoc插件的参数
    string output_location;     // 源代码文件输出目录
  };
  // 一次protoc调用可能会同时制定多个--${lang}_out选项
  vector<OutputDirective> output_directives_;

  // 当使用--encode或者--decode的时候，codec_type_指明了encode或者decode的类型
  // - 如果codec_type_为空则表示--decode_raw类型;
  string codec_type_;

  // 如果指定了--descriptor_set_out选项，FileDescriptorSet将被输出到指定的文件
  string descriptor_set_name_;

  // 如果指定了--include-imports那么所有的依赖proto都要写到DescriptorSet;
  // 如果未指定，则只把命令行中列出的proto文件写入;
  bool imports_in_descriptor_set_;

  // 如果指定--include_source_info为true，则不能从DescriptorSet中删除SourceCodeInfo
  bool source_info_in_descriptor_set_;

  // --disallow_services_这个选项有被使用吗?
  bool disallow_services_;

  // See SetInputsAreProtoPathRelative().
  bool inputs_are_proto_path_relative_;

  GOOGLE_DISALLOW_EVIL_CONSTRUCTORS(CommandLineInterface);
};

}  // namespace compiler
}  // namespace protobuf

}  // namespace google
#endif  // GOOGLE_PROTOBUF_COMPILER_COMMAND_LINE_INTERFACE_H__
```

### 2.3. protoc执行流程说明#

protoc执行流程的相关源码，主要包括如下两个部分。

#### *2.3.1. protoc*完成基本的初始化工作后调用*cli.Run(argc,argv)*开始生成代码#

这部分内容在前面已经有过较为详细的解释，这里不再详细展开，只给出相应的代码、注释。

file: src/google/protobuf/compiler/main.cc

```
// Author: kenton@google.com (Kenton Varda)

// 这个头文件定义了protoc的命令行接口
#include <google/protobuf/compiler/command_line_interface.h>

// protoc中内置了对cpp、python、java语言的支持，对其他语言的支持需要以plugin的方式来支持
#include <google/protobuf/compiler/cpp/cpp_generator.h>
#include <google/protobuf/compiler/python/python_generator.h>
#include <google/protobuf/compiler/java/java_generator.h>


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

  // 初始化protoc命令行接口并开启插件
  // - 插件只是普通的可执行程序，其文件名以AllowPlugins参数protoc-开头
  // - 假定protoc --foo_out，那么实际调用的插件是protoc-foo
  google::protobuf::compiler::CommandLineInterface cli;
  cli.AllowPlugins("protoc-");

  // Proto2 C++ (指定了--cpp_out将调用cpp::Generator)
  google::protobuf::compiler::cpp::CppGenerator cpp_generator;
  cli.RegisterGenerator("--cpp_out", "--cpp_opt", &cpp_generator,
                        "Generate C++ header and source.");

  // Proto2 Java (指定了--java_out将调用java::Generator)
  google::protobuf::compiler::java::JavaGenerator java_generator;
  cli.RegisterGenerator("--java_out", &java_generator,
                        "Generate Java source file.");

  // Proto2 Python (指定了python_out将调用python::Generator)
  google::protobuf::compiler::python::Generator py_generator;
  cli.RegisterGenerator("--python_out", &py_generator,
                        "Generate Python source file.");

  return cli.Run(argc, argv);
}
```

#### *2.3.2. protoc*命令接口*cli.Run(argc,argv)*详细处理流程#

这部分代码是protoc对proto文件进行读取、解析、通过注册的
代码生成器或者protoc插件生成源代码、保存源代码到源文件的
执行流程。

```
int CommandLineInterface::Run(int argc, const char* const argv[]) {
  Clear();
  switch (ParseArguments(argc, argv)) {
    case PARSE_ARGUMENT_DONE_AND_EXIT:     return 0;
    case PARSE_ARGUMENT_FAIL:              return 1;
    case PARSE_ARGUMENT_DONE_AND_CONTINUE: break;
  }

  // Set up the source tree.
  DiskSourceTree source_tree;
  for (int i = 0; i < proto_path_.size(); i++) {
    source_tree.MapPath(proto_path_[i].first, proto_path_[i].second);
  }

  // Map input files to virtual paths if necessary.
  if (!inputs_are_proto_path_relative_) {
    if (!MakeInputsBeProtoPathRelative(&source_tree)) {
      return 1;
    }
```

```
}

// Allocate the Importer.
ErrorPrinter error_collector(error_format_, &source_tree);
Importer importer(&source_tree, &error_collector);

vector<const FileDescriptor*> parsed_files;

// Parse each file.
for (int i = 0; i < input_files_.size(); i++) {
  // Import the file.
  const FileDescriptor* parsed_file = importer.Import(input_files_[i]);
  if (parsed_file == NULL) return 1;
  parsed_files.push_back(parsed_file);

  // Enforce --disallow_services.
  if (disallow_services_ && parsed_file->service_count() > 0) {
    cerr << parsed_file->name() << ": This file contains services, but "
            "--disallow_services was used." << endl;
    return 1;
  }
}

// We construct a separate GeneratorContext for each output location.  Note
// that two code generators may output to the same location, in which case
// they should share a single GeneratorContext so that OpenForInsert() works.
typedef hash_map<string, GeneratorContextImpl*> GeneratorContextMap;
GeneratorContextMap output_directories;

// Generate output.
if (mode_ == MODE_COMPILE) {
  for (int i = 0; i < output_directives_.size(); i++) {
    string output_location = output_directives_[i].output_location;
    if (!HasSuffixString(output_location, ".zip") &&
        !HasSuffixString(output_location, ".jar")) {
      AddTrailingSlash(&output_location);
    }
    GeneratorContextImpl** map_slot = &output_directories[output_location];

    if (*map_slot == NULL) {
      // First time we've seen this output location.
      *map_slot = new GeneratorContextImpl(parsed_files);
    }

    if (!GenerateOutput(parsed_files, output_directives_[i], *map_slot)) {
      STLDeleteValues(&output_directories);
      return 1;
    }
  }
}

// Write all output to disk.
for (GeneratorContextMap::iterator iter = output_directories.begin();
     iter != output_directories.end(); ++iter) {
  const string& location = iter->first;
  GeneratorContextImpl* directory = iter->second;
  if (HasSuffixString(location, "/")) {
    if (!directory->WriteAllToDisk(location)) {
      STLDeleteValues(&output_directories);
      return 1;
    }
```

```
    } else {
      if (HasSuffixString(location, ".jar")) {
        directory->AddJarManifest();
      }

      if (!directory->WriteAllToZip(location)) {
        STLDeleteValues(&output_directories);
        return 1;
      }
    }
  }

  STLDeleteValues(&output_directories);

  if (!descriptor_set_name_.empty()) {
    if (!WriteDescriptorSet(parsed_files)) {
      return 1;
    }
  }

  if (mode_ == MODE_ENCODE || mode_ == MODE_DECODE) {
    if (codec_type_.empty()) {
      // HACK:  Define an EmptyMessage type to use for decoding.
      DescriptorPool pool;
      FileDescriptorProto file;
      file.set_name("empty_message.proto");
      file.add_message_type()->set_name("EmptyMessage");
      GOOGLE_CHECK(pool.BuildFile(file) != NULL);
      codec_type_ = "EmptyMessage";
      if (!EncodeOrDecode(&pool)) {
        return 1;
      }
    } else {
      if (!EncodeOrDecode(importer.pool())) {
        return 1;
      }
    }
  }

  return 0;
}
```

*2.3.3. protoc*执行逻辑总结#

通过查看上面两部分代码及其他关键代码，可以对protoc的执行流程进行一个简单的总结了：

1. protoc main中初始化命令行参数接口cli；
2. protoc 中开启protoc-前缀的插件作为第三方代码生成器使用cli.AllowPlugins(“protoc-");
3. protoc main中注册编程语言cpp、java、python及对应的代码生成器，cli.RegisterGenerator(…)，原生支持cpp、java、python；
4. protoc main中cli.Run(argc, argv)，运行注册的代码生成器或者protoc插件；
    - Clear()清空所有的数据备用；
    - ParseArguments(argc,argv)解析参数，对于protoc的某些内置参数的检查，对某些插件相关的–${lang}_out参数的检查等，将–${lang}_out作为输出指令保存起来；

```
struct OutputDirective {
  string name;                // E.g. "--foo_out"
  CodeGenerator* generator;   // NULL for plugins
  string parameter;
  string output_location;
};
```

- 参数解析成功之后，继续执行处理，设置源代码树、映射输入文件到虚拟路径、分配importer；
- 针对每个输入的proto文件进行解析，const FileDescriptor* parsed_file = importer.Import(input_files_[i])，解析成功后的文件会被加入到vector<FileDescriptor*> parsed_files中记录，每一个proto文件解析后都可以用一个FileDescriptor结构体来表示；

备注：
这里解析proto文件的过程是这样的，首先将proto文件中的内容分割成一个个的token，将内容拆分成一个个词素并检查有效性，也就是词法分析。如果词法分析检查无误则进入后续的语法分析过程，parser对输入token串进行文法相关的分析检查是否可以构成一棵有效的语法分析树，如果可以则表示语法正确。

```
// Token定义如下
struct Token {
  TokenType type;
  string text;       // The exact text of the token as it appeared in
                     // the input.  e.g. tokens of TYPE_STRING will still
                     // be escaped and in quotes.

  // "line" and "column" specify the position of the first character of
  // the token within the input stream.  They are zero-based.
  int line;
  int column;
  int end_column;
};

// Token类型定义如下
enum TokenType {
  TYPE_START,       // Next() has not yet been called.
  TYPE_END,         // End of input reached.  "text" is empty.

  TYPE_IDENTIFIER,  // A sequence of letters, digits, and underscores, not
                    // starting with a digit.  It is an error for a number
                    // to be followed by an identifier with no space in
                    // between.
  TYPE_INTEGER,     // A sequence of digits representing an integer.  Normally
                    // the digits are decimal, but a prefix of "0x" indicates
                    // a hex number and a leading zero indicates octal, just
                    // like with C numeric literals.  A leading negative sign
                    // is NOT included in the token; it's up to the parser to
                    // interpret the unary minus operator on its own.
  TYPE_FLOAT,       // A floating point literal, with a fractional part and/or
                    // an exponent.  Always in decimal.  Again, never
                    // negative.
  TYPE_STRING,      // A quoted sequence of escaped characters.  Either single
                    // or double quotes can be used, but they must match.
                    // A string literal cannot cross a line break.
```

```
    TYPE_SYMBOL,      // Any other printable character, like '!' or '+'.
                      // Symbols are always a single character, so "!+$%" is
                      // four tokens.
  };
```

语法分析的过程这里就不解释了，感兴趣的可以看一下protobuf中grammar的定义，无非也就是些规约的事情，只要能够按照grammar将词法分析输出的token串构建出一棵完整的语法分析树proto文件就是合法的，否则就是不合法的，至于语法分析过程中伴随的语义分析过程，语义分析过程中执行哪些语义动作，不说也知道，肯定是生成某些“中间代码”之类的鬼东西。学过编译原理的这些处理过程应该都是比较清楚的，这里就不再展开了。语法分析成功之后就得到了proto对应的FileDescriptor对象，因为可能输入的是多个proto，所以多个FileDescriptor就用vector来存储了。

- 遍历之前记录下来的输出指令OutputDirective output_directives[]，output_directives[i].output_location指明了输出目录，针对输出目录创建GeneratorContextImpl，并记录到hash_map<string, GeneratorContextImpl*> output_directories这个map中，key为flag_out,如–foo_out，value为GeneratorContextImpl。由于可能多个–${lang}_out都指向相同的输出目录，所以同一个GeneratorContextImpl也存在复用的情况。每个GeneratorContextImpl记录了一个输出目录、所有该目录下的待创建的源代码文件的信息，待创建的源代码文件信息记录在map<string,string*> files_里面，key为源代码文件名，val为源代码文件的内容，另外还包括了一个vector<FileDescriptor*> parsed_files记录了所有解析成功的proto文件信息。遍历output_directives的同时，因为同一个output_directives[i]对应的输出目录下可能有多个源代码文件要输出，并且不管flag_name是什么，要处理的proto文件都是相同的，所以每个output_directives[i]都会对其调用**GenerateOutput**(parsed_files, output_directives[i], *map_slot)，output_directives[i].plugin指明了语言的代码生成器(为NULL则使用插件)，对所有的解析成功的proto文件parsed_files[i]生成源代码，源代码全部输出到output_directive[i].output_location下，源代码的文件名都记录在parsed_files[i].name()里面，而最终生成的源代码信息都存储在这里的CodeGeneratorImpl **map_slot中，也就相当于存储在了output_directories[]中。
- 最后遍历output_directories[]，将每个输出目录下要写的所有文件的数据全部写出

到磁盘，即output_directories[i]->WriteAllToDisk()。
- done!

了解从proto到源代码生成的关键之处就是这里的GenerateOutput是怎么实现的，接着看。

#### 2.3.3.1 protoc CommandLineInterface::GenerateOutput(…)实现#

下面看下GenerateOutput方法到底执行了哪些操作。

```
// 根据输出指示，为解析成功的proto文件调用代码生成器生成对应的代码并存储到generator_context
//@param parsed_files 所有解析成功的proto文件，每个解析成功的proto文件都用一个FileDescriptor来表示
//@param output_directive 输出指示，其指明了目标语言、语言对应的代码生成器、输出目录等
//@param generator_context 代码生成器上下文，可记录生成的代码
bool CommandLineInterface::GenerateOutput(const vector<const FileDescriptor*>& parsed_files,
                                          const OutputDirective& output_directive,
                                          GeneratorContext* generator_context) {
  // Call the generator.
  string error;

  // 如果输出指示中没有设置对应的代码生成器，表明没有在protoc main中注册语言对应的代码生成器，
  // 这种需要protoc通过插件机制，通过调用对应的插件来充当代码生成器的功能。
  if (output_directive.generator == NULL) {
    // This is a plugin.
    GOOGLE_CHECK(HasPrefixString(output_directive.name, "--") &&
          HasSuffixString(output_directive.name, "_out"))
        << "Bad name for plugin generator: " << output_directive.name;

    // 实际上protoc搜索插件对应的可执行程序的时候，搜索的名称是“protoc-gen-”+“语言”，
    // 如果我们调用的是protoc --xxx_out，那么实际搜索的就是protoc-gen-xxx。
    string plugin_name = plugin_prefix_ + "gen-" +
        output_directive.name.substr(2, output_directive.name.size() - 6);

    // 调用protoc插件来生成代码，这是我们要重点看的，我们就是要实现自己的protoc插件
    if (!GeneratePluginOutput(parsed_files, plugin_name, output_directive.parameter, generator
      cerr << output_directive.name << ": " << error << endl;
      return false;
    }
  } else {

    // 这种是protoc main函数中正常注册的语言和代码生成器
    // Regular generator.
    string parameters = output_directive.parameter;
    if (!generator_parameters_[output_directive.name].empty()) {
      if (!parameters.empty()) {
        parameters.append(",");
      }
      parameters.append(generator_parameters_[output_directive.name]);
    }

    // 为每个解析成功的proto文件生成代码
    for (int i = 0; i < parsed_files.size(); i++) {
      if (!output_directive.generator->Generate(parsed_files[i], parameters, generator_context
        // Generator returned an error.
        cerr << output_directive.name << ": " << parsed_files[i]->name() << ": " << error << e
        return false;
      }
    }
  }
```

```
  return true;
}
```

#### 2.3.3.2. protoc调用插件生成代码的执行逻辑#

下面再来看一下GeneratePluginOutput是如何工作的。

```
// 调用protoc插件为解析成功的proto文件生成代码
//
//@param parsed_files 解析成功的文件
//@param plugin_name protoc插件名称（这个是拼接出来的protoc-gen-${lang}）
//@param parameter 传给插件的参数
//@param generator_context 代码生成器上下文，可记录生成的代码
//@param error 代码生成过程中的错误信息
bool CommandLineInterface::GeneratePluginOutput(const vector<const FileDescriptor*>& parsed_fi
                                                const string& plugin_name,
                                                const string& parameter,
                                                GeneratorContext* generator_context,
                                                string* error) {
  // protoc生成一个代码生成请求，并发送给插件
  CodeGeneratorRequest request;

  // protoc插件根据接收到的代码生成请求生成代码，并发送响应给protoc
  CodeGeneratorResponse response;

  // Build the request.
  if (!parameter.empty()) {
    request.set_parameter(parameter);
  }

  set<const FileDescriptor*> already_seen;
  for (int i = 0; i < parsed_files.size(); i++) {
    request.add_file_to_generate(parsed_files[i]->name());
    GetTransitiveDependencies(parsed_files[i],
                              true,  // Include source code info.
                              &already_seen, request.mutable_proto_file());
  }

  // fork出一个子进程，子进程来执行插件完成代码生成工作，
  // 父子进程之间是通过管道通信完成请求、响应过程，如何控制子进程的stdin、stdout，
  // 这个可以通过dup2或者dup3来控制间fd 0、1分别设置到管道的读端、写端。
  // 事实上protobuf的开发人员也是这么来实现的。

  // Invoke the plugin.
  Subprocess subprocess;

  if (plugins_.count(plugin_name) > 0) {
    subprocess.Start(plugins_[plugin_name], Subprocess::EXACT_NAME);
  } else {
    subprocess.Start(plugin_name, Subprocess::SEARCH_PATH);
  }

  string communicate_error;

  // 请求插件生成代码
  if (!subprocess.Communicate(request, &response, &communicate_error)) {
    *error = strings::Substitute("$0: $1", plugin_name, communicate_error);
    return false;
  }

  // Write the files.  We do this even if there was a generator error in order
  // to match the behavior of a compiled-in generator.
  scoped_ptr<io::ZeroCopyOutputStream> current_output;
```

```
  for (int i = 0; i < response.file_size(); i++) {
    const CodeGeneratorResponse::File& output_file = response.file(i);

    if (!output_file.insertion_point().empty()) {

      // 首先关闭当前正在写入的文件数据（用CodeGeneratorResponse表示）
      // 打开待写入的文件数据，这个文件数据已经存在，定位到准确的插入点位置执行写入，然后关闭文件
      // - 这里的插入点如何定义，我们在后面再进行说明。具体可参考plugin.proto和plugin.pb.h。
      current_output.reset();

      // OpenForInsert返回一个输出流，以方便后面写入编码后数据
      current_output.reset(generator_context->OpenForInsert(output_file.name(), output_file.ir

    } else if (!output_file.name().empty()) {

      // 首先关闭当前正在写入的文件数据（用CodeGeneratorResponse表示）
      // 打开待写入的文件数据，这个文件数据不存在，不存在插入点信息，从开始处执行写入
      current_output.reset();

      // OpenForInsert返回一个输出流，以方便后面写入编码后数据
      current_output.reset(generator_context->Open(output_file.name()));

    } else if (current_output == NULL) {

      *error = strings::Substitute(
        "$0: First file chunk returned by plugin did not specify a file name.",
        plugin_name);
      return false;

    }

    // 从CodeGeneratorResponse中获取输出流，写出，这里输出流中的数据时存储在GeneratorContextImpl中
    // GenerateOutput调用成功之后后面会遍历每一个GenerateContextImpl完成WriteAllToDisk()的操作。

    // Use CodedOutputStream for convenience; otherwise we'd need to provide
    // our own buffer-copying loop.
    io::CodedOutputStream writer(current_output.get());
    writer.WriteString(output_file.content());
  }

  // Check for errors.
  if (!response.error().empty()) {
    // Generator returned an error.
    *error = response.error();
    return false;
  }

  return true;
}
```

#### 2.3.3.3. protoc & protoc插件数据交互的执行逻辑#

整体执行逻辑差不多理清楚了，然后这里我们需要看一下父进程给子进程发送的代码生成请求是什么，收到的代码生成的响应又是什么，以及父子进程通信的细节、子进程对请求的处理过程等。

先来看下plugin.proto的定义，protoc内置的支持语言里面并不包含go，我们后面需要用go来编写我们自己的插件，所以必须使用protoc的go插件来生成go对应的plugin.go代码，然后我们自己写一些业务类插件（非语言插件）的时候才能用上plugin.go。扯这么多是为了让大家明白这里为什么需要看下

plugin.proto，而不是误解为只是在堆砌内容。看了这里的plugin.proto之后才能理解到protoc中的插件机制的边界时什么，我们就可以明白利用protoc的插件机制，我们可以做到什么程度，哪些功能能实现，哪些实现不了，这个是很重要的。

file: src/google/protobuf/compiler/plugin.proto

```
// protoc (aka the Protocol Compiler) can be extended via plugins.  A plugin is
// just a program that reads a CodeGeneratorRequest from stdin and writes a
// CodeGeneratorResponse to stdout.
//
// Plugins written using C++ can use google/protobuf/compiler/plugin.h instead
// of dealing with the raw protocol defined here.
//
// A plugin executable needs only to be placed somewhere in the path.  The
// plugin should be named "protoc-gen-$NAME", and will then be used when the
// flag "--${NAME}_out" is passed to protoc.

package google.protobuf.compiler;
option java_package = "com.google.protobuf.compiler";
option java_outer_classname = "PluginProtos";

import "google/protobuf/descriptor.proto";

// 发送给插件的代码生成请求
// An encoded CodeGeneratorRequest is written to the plugin's stdin.
message CodeGeneratorRequest {
  // The .proto files that were explicitly listed on the command-line.  The
  // code generator should generate code only for these files.  Each file's
  // descriptor will be included in proto_file, below.

  //proto文件列表对应的要生成的文件的源代码文件的名字
  repeated string file_to_generate = 1;

  // The generator parameter passed on the command-line.
  //传递给插件代码生成器的参数
  optional string parameter = 2;

  // FileDescriptorProtos for all files in files_to_generate and everything
  // they import.  The files will appear in topological order, so each file
  // appears before any file that imports it.
  //
  // protoc guarantees that all proto_files will be written after
  // the fields above, even though this is not technically guaranteed by the
  // protobuf wire format.  This theoretically could allow a plugin to stream
  // in the FileDescriptorProtos and handle them one by one rather than read
  // the entire set into memory at once.  However, as of this writing, this
  // is not similarly optimized on protoc's end -- it will store all fields in
  // memory at once before sending them to the plugin.

  // 每一个正确解析的proto文件都用一个FileDescriptorProto来表示;
  // 这里的FileDescriptorProto与FileDescriptor其实是对应的，在请求插件进行代码
  // 生成的时候直接就有这样的代码FileDescriptor::CopyTo(FileDescriptorProto&)
  // 的用法。而在descriptor.h和descriptor.proto中查看二者的描述时，其注释清清
  // 楚楚地写着都是描述的一个完整的proto文件。
  repeated FileDescriptorProto proto_file = 15;
}

// 插件返回的代码生成响应
// The plugin writes an encoded CodeGeneratorResponse to stdout.
message CodeGeneratorResponse {
  // Error message.  If non-empty, code generation failed.  The plugin process
```

```
  // should exit with status code zero even if it reports an error in this way.
  //
  // This should be used to indicate errors in .proto files which prevent the
  // code generator from generating correct code.  Errors which indicate a
  // problem in protoc itself -- such as the input CodeGeneratorRequest being
  // unparseable -- should be reported by writing a message to stderr and
  // exiting with a non-zero status code.

  // 错误信息
  optional string error = 1;

  // Represents a single generated file.
  // 生成的源代码文件消息类型，注意这里是一个内部类型
  message File {
    // 待生成的源代码文件名（相对于输出目录），文件名中不能包括.或者..，路径是
    // 相对输出目录的路径，不能用绝对路径，另分隔符必须用/。
    // 如果name没有指定，那么输出的内容将追加到前一个输出的源代码文件中，这种
    // 方式使得代码生成器能够将一个大文件的生成分多次写入来完成，不用一次性将很
    // 大数据量的数据放在内存中。这里需要指出的是，protoc中并没有针对这种情况
    //  进行特殊的优化，它等待读取完整的CodeGeneratorResponse再写出到磁盘。
    optional string name = 1;

    // 如果insertion_point不空的话，name字段也不能为空，并且假定name字段指定的
    // 文件已经存在了。这里的内容将被插入到name指定的文件中的特定插入点（注解）的
    // 上一行。这有助于扩展代码生成器输出的内容。在一次protoc调用中，可能会同
    // 时指定多个protoc插件，前面的插件可能会在输出的内容中指定插入点，后面的
    // 插件可能会在这些指定的插入点的位置继续扩展代码内容。
    // 例如，前面的一个插件在输出的代码内容中增加了这样一行注解：
    //   @@protoc_insertion_point(NAME)
    // 这样就定义了一个插入点，插入点前面、后面可以包含任意的文本内容，即使在
    // 注释里面也是可以的。这里的插入点定义中的NAME应该可以唯一标识一个插入点
    // 才可以，类似于标识符，以供其他的插件使用，插件插入代码的时候将从插入点
    // 的上一行开始自行插入。如果包含多个插入点的话，插入点的内容将被插件依次
    // 扩展。
    //
    // 一开始创建这个源代码文件的代码生成器或者插件与后面的继续扩展源代码插入
    // 点位置内容的代码生成器或者插件，必须在protoc的同一次调用中，代码生成器
    // 或者插件按照protoc命令行调用过程中指定的顺序依次调用。
    optional string insertion_point = 2;

    // 待写入到源代码中的内容
    optional string content = 15;
  }

  // 一次要处理的 proto文件可能有多个，所以插件处理后这里的file是一个list
  repeated File file = 15;
}
```

下面看一下protoc与protoc插件这对父子进程之间是怎么通信的。

file: src/google/protobuf/compiler/subprocess.h

```
class LIBPROTOC_EXPORT Subprocess {
 public:
  Subprocess();
  ~Subprocess();

  enum SearchMode {
    SEARCH_PATH,   // Use PATH environment variable.
    EXACT_NAME     // Program is an exact file name; don't use the PATH.
  };
```

```
  // Start the subprocess.  Currently we don't provide a way to specify
  // arguments as protoc plugins don't have any.
  void Start(const string& program, SearchMode search_mode);

  // Serialize the input message and pipe it to the subprocess's stdin, then
  // close the pipe.  Meanwhile, read from the subprocess's stdout and parse
  // the data into *output.  All this is done carefully to avoid deadlocks.
  // Returns true if successful.  On any sort of error, returns false and sets
  // *error to a description of the problem.
  bool Communicate(const Message& input, Message* output, string* error);

  // win32 relevant ... neglect

 private:
#ifdef _WIN32
  // ...

#else  // !_WIN32
  pid_t child_pid_;

  // The file descriptors for our end of the child's pipes.  We close each and
  // set it to -1 when no longer needed.
  int child_stdin_;
  int child_stdout_;

#endif
};
```

下面是Linux平台下的子进程启动处理逻辑。

file: src/google/protobuf/compiler/subprocess.cc

```
void Subprocess::Start(const string& program, SearchMode search_mode) {
  // Note that we assume that there are no other threads, thus we don't have to
  // do crazy stuff like using socket pairs or avoiding libc locks.

  // [0] is read end, [1] is write end.
  int stdin_pipe[2];
  int stdout_pipe[2];

  GOOGLE_CHECK(pipe(stdin_pipe) != -1);
  GOOGLE_CHECK(pipe(stdout_pipe) != -1);

  char* argv[2] = { strdup(program.c_str()), NULL };

  child_pid_ = fork();
  if (child_pid_ == -1) {
    GOOGLE_LOG(FATAL) << "fork: " << strerror(errno);
  } else if (child_pid_ == 0) {
    // We are the child.
    // 将子进程的stdin重定向到stdin_pipe的读端
    dup2(stdin_pipe[0], STDIN_FILENO);
    // 将子进程的stdout重定向到stdout_pipe的写端
    dup2(stdout_pipe[1], STDOUT_FILENO);

    // 子进程通过0、1对管道进行操作就够了，释放多余的fd
    close(stdin_pipe[0]);
    close(stdin_pipe[1]);
    close(stdout_pipe[0]);
    close(stdout_pipe[1]);

    // 根据程序搜索模式调用exec族函数来调用插件执行，exec族函数通过替换当前进
    // 程的代码段、数据段等内存数据信息，然后调整寄存器信息，使得进程转而去执
```

```
    // 行插件的代码。插件代码执行之前进程就已经将fd 0、1重定向到父进程clone过
    // 来的管道了，因此插件程序的输出将直接被输出到父进程创建的管道中。
    // 正常情况下，exec一旦执行成功，那么久绝不对执行switch后续的代码了，只有
    // 出错才可能会执行到后续的代码。
    switch (search_mode) {
      case SEARCH_PATH:
        execvp(argv[0], argv);
        break;
      case EXACT_NAME:
        execv(argv[0], argv);
        break;
    }

    // 只有出错才可能会执行到这里的代码。
    // Write directly to STDERR_FILENO to avoid stdio code paths that may do
    // stuff that is unsafe here.
    int ignored;
    ignored = write(STDERR_FILENO, argv[0], strlen(argv[0]));
    const char* message = ": program not found or is not executable\n";
    ignored = write(STDERR_FILENO, message, strlen(message));
    (void) ignored;

    // Must use _exit() rather than exit() to avoid flushing output buffers
    // that will also be flushed by the parent.
    _exit(1);
  } else {
    free(argv[0]);

    // 父进程释放无用的fd
    close(stdin_pipe[0]);
    close(stdout_pipe[1]);

    // 子进程的stdin，对父进程来说也就是管道stdin_pipe的写端，CodeGeneratorRequest将通过这个fd写给
    child_stdin_ = stdin_pipe[1];
    // 子进程的stdout，对父进程来说也就是管道stdout_pipe的读端，CodeGeneratorResponse将通过这个fd从
    child_stdout_ = stdout_pipe[0];
  }
}
```

下面接着看父进程读取子进程返回的CodeGeneratorResponse的
执行逻辑。

```
bool Subprocess::Communicate(const Message& input, Message* output, string* error) {

  GOOGLE_CHECK_NE(child_stdin_, -1) << "Must call Start() first.";

  // The "sighandler_t" typedef is GNU-specific, so define our own.
  typedef void SignalHandler(int);

  // Make sure SIGPIPE is disabled so that if the child dies it doesn't kill us.
  SignalHandler* old_pipe_handler = signal(SIGPIPE, SIG_IGN);

  string input_data = input.SerializeAsString();
  string output_data;

  int input_pos = 0;
  int max_fd = max(child_stdin_, child_stdout_);

  // child_stdout==-1的时候表示子进程返回的数据已经读取完毕了，可以gg了
  while (child_stdout_ != -1) {
    fd_set read_fds;
    fd_set write_fds;
```

```
  FD_ZERO(&read_fds);
  FD_ZERO(&write_fds);
  if (child_stdout_ != -1) {
    FD_SET(child_stdout_, &read_fds);
  }
  if (child_stdin_ != -1) {
    FD_SET(child_stdin_, &write_fds);
  }

  // 这种情景下也用select，果然很google!
  if (select(max_fd + 1, &read_fds, &write_fds, NULL, NULL) < 0) {
    if (errno == EINTR) {
      // Interrupted by signal.  Try again.
      continue;
    } else {
      GOOGLE_LOG(FATAL) << "select: " << strerror(errno);
    }
  }

  // stdout_pipe写事件就绪，写请求CodeGeneratorRequest给子进程
  if (child_stdin_ != -1 && FD_ISSET(child_stdin_, &write_fds)) {
    int n = write(child_stdin_, input_data.data() + input_pos,
                                input_data.size() - input_pos);
    if (n < 0) {
      // Child closed pipe.  Presumably it will report an error later.
      // Pretend we're done for now.
      input_pos = input_data.size();
    } else {
      input_pos += n;
    }

    // 代码生成请求已经成功写给子进程了，关闭相关的fd
    if (input_pos == input_data.size()) {
      // We're done writing.  Close.
      close(child_stdin_);
      child_stdin_ = -1;
    }
  }

  // stdin_pipe读事件就绪，读取子进程返回的CodeGeneratorResponse
  if (child_stdout_ != -1 && FD_ISSET(child_stdout_, &read_fds)) {
    char buffer[4096];
    int n = read(child_stdout_, buffer, sizeof(buffer));

    if (n > 0) {
      output_data.append(buffer, n);
    } else {
      // 子进程返回的CodeGeneratorResponse已经读取完毕，关闭相关的fd
      close(child_stdout_);
      child_stdout_ = -1;
    }
  }
}

// 子进程还没有读取CodeGeneratorRequest完毕，就关闭了输出，这种情况下也不可
// 能读取到返回的CodeGeneratorResponse了，这种情况很可能是出现了异常。
if (child_stdin_ != -1) {
  // Child did not finish reading input before it closed the output.
  // Presumably it exited with an error.
  close(child_stdin_);
  child_stdin_ = -1;
```

```
  }

  // 等待子进程结束，子进程退出之后，需要父进程来清理子进程占用的部分资源。
  // 如果当前父进程不waitpid的话，子进程的父进程会变为init或者systemd进程，同样也会被清理的。
  int status;
  while (waitpid(child_pid_, &status, 0) == -1) {
    if (errno != EINTR) {
      GOOGLE_LOG(FATAL) << "waitpid: " << strerror(errno);
    }
  }

  // 刚才为了阻止SIGPIPE信号到达时导致进程终止，我们修改了SIGPIPE的信号处理函
  // 数，这里可以恢复之前的SIGPIPE的信号处理函数。
  signal(SIGPIPE, old_pipe_handler);

  // 根据子进程的退出状态执行后续的处理逻辑
  // - 异常处理
  if (WIFEXITED(status)) {
    if (WEXITSTATUS(status) != 0) {
      int error_code = WEXITSTATUS(status);
      *error = strings::Substitute(
          "Plugin failed with status code $0.", error_code);
      return false;
    }
  } else if (WIFSIGNALED(status)) {
    int signal = WTERMSIG(status);
    *error = strings::Substitute(
        "Plugin killed by signal $0.", signal);
    return false;
  } else {
    *error = "Neither WEXITSTATUS nor WTERMSIG is true?";
    return false;
  }

  // 将子进程返回的串行化之后的CodeGeneratorResponse数据进行反串行化，反串行化
  // 成Message对象，实际上这里的Message::ParseFromString(const string&)是个虚
  // 函数，是被CodeGeneratorResponse这个类重写了的，反串行化过程与具体的类密切
  // 相关，也必须在派生类中予以实现。
  if (!output->ParseFromString(output_data)) {
    *error = "Plugin output is unparseable.";
    return false;
  }

  return true;
}
```

到这里为止protoc进程的具体执行逻辑我们已经很清楚了，看到这里想必读者也看清楚了吧？下面再看下插件的执行逻辑。

插件的执行逻辑一定也是非常简单的，插件就只是从stdin读取串行化之后的CodeGeneratorRequest请求，然后执行反串行化得到一个完整的CodeGeneratorRequest对象，然后根据请求进行必要的代码生成逻辑，确定要生成的源代码信息，并将其设置到CodeGeneratorResponse中并串行化后写入到stdout，插件的执行逻辑就这么简单。

下面我们将进入插件的编写过程了。

## 2.4. protoc插件开发#

### *2.4.1. protoc*中的*descriptor*定义#

proto文件中的数据类型都是在descriptor.proto中定义好的，为了更好地帮助我们对proto文件中的数据类型进行解析，为了在插件开发过程中更加方便快速地获得与数据类型、变量、rpc等相关的这种内容、那种内容，我们都需要深入地理解descriptor.proto中的相关定义以及从它延伸出来的一些概念、算法等。

这部分的内容还不少，在不影响理解的大前提下，我还是稍微删减写些代码，避免对大家理解造成不必要的干扰。

file: src/google/protobuf/descriptor.proto

```
// Author: kenton@google.com (Kenton Varda)
// Based on original Protocol Buffers design by
// Sanjay Ghemawat, Jeff Dean, and others.

// descriptor.proto文件中的messages定义了proto文件中所能见到的所有的定义，一个有效的.proto文件在不报

package google.protobuf;
option java_package = "com.google.protobuf";
option java_outer_classname = "DescriptorProtos";

// descriptor.proto必须在速度方面优化，因为在启动过程中基于反射的算法不起作用
option optimize_for = SPEED;

// protoc可以将解析的proto文件中的descriptor添加到FileDescriptorSet并输出到文件
message FileDescriptorSet {
  repeated FileDescriptorProto file = 1;
}

// 下面的message FileDescriptorProto可以用于描述一个完整的proto文件
message FileDescriptorProto {

  optional string name = 1;              // proto文件名，file name，相对于源代码根目录
  optional string package = 2;           // proto包名，例如 "foo"、"foo.bar"
  repeated string dependency = 3;        // proto文件中import进来的其他proto文件列表
  repeated int32 public_dependency = 10; // 上面public import的proto文件在proto文件列表中的索引

  // Indexes of the weak imported files in the dependency list.
  repeated int32 weak_dependency = 11;   // 上面weak import的proto文件在proto文件列表中的索引
                                         // 不要使用，只用于google内部的迁移

  // proto文件中的所有顶层定义信息
  repeated DescriptorProto message_type = 4;    // 所有的消息(message)类型定义
  repeated EnumDescriptorProto enum_type = 5;   // 所有的枚举(enum)类型定义
  repeated ServiceDescriptorProto service = 6;  // 所有的服务(service)类型定义
  repeated FieldDescriptorProto extension = 7;  // 所有的扩展字段定义

  optional FileOptions options = 8;             // 文件选项

  // 这个字段包括了源代码的相关信息，这里的信息可以给开发工具使用，也仅应该提供给开发工具使用；
  // 可以选择将这个字段中的信息删除，在程序运行期间并不会造成破坏。
  optional SourceCodeInfo source_code_info = 9;
}

// 描述消息类型Message
message DescriptorProto {
  optional string name = 1;                     // Message的类型名称

  repeated FieldDescriptorProto field = 2;      // Message中包括的字段列表
  repeated FieldDescriptorProto extension = 6;  // Message中包括的扩展列表
```

```
  repeated DescriptorProto nested_type = 3;     // Message中嵌套的Message类型列表
  repeated EnumDescriptorProto enum_type = 4;   // Message中嵌套的枚举类型列表

  message ExtensionRange {
    optional int32 start = 1;
    optional int32 end = 2;
  }
  repeated ExtensionRange extension_range = 5;

  optional MessageOptions options = 7;
}

// 描述一个字段（字段可以是Message中的，也可以是某些扩展字段）
message FieldDescriptorProto {
  // 字段数据类型
  enum Type {
    // 0 is reserved for errors.
    // 由于历史方面的原因，这里的枚举值的顺序有点奇怪
    TYPE_DOUBLE         = 1;
    TYPE_FLOAT          = 2;
    // Not ZigZag encoded.  Negative numbers take 10 bytes.  Use TYPE_SINT64 if
    // negative values are likely.
    TYPE_INT64          = 3;
    TYPE_UINT64         = 4;
    // Not ZigZag encoded.  Negative numbers take 10 bytes.  Use TYPE_SINT32 if
    // negative values are likely.
    TYPE_INT32          = 5;
    TYPE_FIXED64        = 6;
    TYPE_FIXED32        = 7;
    TYPE_BOOL           = 8;
    TYPE_STRING         = 9;
    TYPE_GROUP          = 10;  // Tag-delimited aggregate.
    TYPE_MESSAGE        = 11;  // Length-delimited aggregate.

    // New in version 2.
    TYPE_BYTES          = 12;
    TYPE_UINT32         = 13;
    TYPE_ENUM           = 14;
    TYPE_SFIXED32       = 15;
    TYPE_SFIXED64       = 16;
    TYPE_SINT32         = 17;  // Uses ZigZag encoding.
    TYPE_SINT64         = 18;  // Uses ZigZag encoding.
  };

  // 字段修饰符optional、required、repeated
  enum Label {
    // 0 is reserved for errors
    LABEL_OPTIONAL      = 1;
    LABEL_REQUIRED      = 2;
    LABEL_REPEATED      = 3;
    // TODO(sanjay): Should we add LABEL_MAP?
  };

  optional string name = 1;             // 字段名称
  optional int32 number = 3;            // 字段tag编号
  optional Label label = 4;             // 字段修饰符

  // 如果type_name已设置，这个字段无须设置；
  // 如果这两个字段都设置了，这里的type字段必须是TYPE_ENUM类型或者TYPE_MESSAGE类型
  optional Type type = 5;
```

```
  // 对于TYPE_ENUM或者TYPE_MESSAGE类型，type_name就是type的名字。
  // 如果name以“.”开头那么它是完全保留的。对于C++来说，其作用域规则要求首先搜
  // 索当前Message类型的嵌套类型，然后才是parent namespace中的类型，一直到root
  // namespace。
  optional string type_name = 6;

  // 对于扩展，它就是被扩展的类型的名字，对它的解析与对type_name的解析时一样的
  optional string extendee = 2;

  // 对于数值类型，存储了数值的文本表示形式；
  // 对于布尔类型，存储字符串"true"或"false"；
  // 对于字符串类型，存储原始的文本内容（未转义的）
  // 对于字节，存储了c转义后的值（所有>=128的字节都会被转义）
  // TODO(kenton)，基于base64编码的?
  optional string default_value = 7;

  optional FieldOptions options = 8;    // 字段选项
}

// 描述一个枚举类型enum
message EnumDescriptorProto {
  optional string name = 1;                     // 枚举类型名称
  repeated EnumValueDescriptorProto value = 2;  // 枚举类型中包括的枚举值列表
  optional EnumOptions options = 3;             // 枚举类型选项
}

// 描述一个枚举类型中的一个枚举值
message EnumValueDescriptorProto {
  optional string name = 1;                     // 枚举值对应的name
  optional int32 number = 2;                    // 枚举值对应的number（默认为0，依次递增）
  optional EnumValueOptions options = 3;        // 枚举值选项
}

// 描述一个rpc service.
message ServiceDescriptorProto {
  optional string name = 1;                     // 服务名称
  repeated MethodDescriptorProto method = 2;    // 服务对应的方法列表
  optional ServiceOptions options = 3;          // 服务选项
}

// 描述一个服务的方法
message MethodDescriptorProto {
  optional string name = 1;                     // 方法名称
  optional string input_type = 2;               // 方法入参类型
  optional string output_type = 3;              // 方法出参类型
  optional MethodOptions options = 4;           // 方法选项
}

// ===================================================================
// Options

// 上面的每一个定义基本上都包括了选项option相关的字段，这些选项字段仅仅是一些
// 注解，这些注解会影响代码的生成，使得生成的代码稍有不同，注解也可能包含了操作
// message的代码的一些提示信息、说明信息。
//
// clients可能会定义一些自定义的选项来作为*Options message的extensions，这些
// extensions在parsing阶段可能还无法确定下来，所以parser不能存储他们的值，而是
// 将这些自定义的选项先存储到一个*Options message里面，称之为
// uinterpreted_option。这个字段的名字在所有的*Options message里面都必须保证是
// 相同的。之后在我们构建descriptor的时候，这个时候所有的proto文件也都解析完了、
// 所有的extensions也都知道了，这个时候我们再用这里的uinterpreted_option字段去
```

```
// 填充那些extensions。
//
// 用于自定义选项的extensions编号的选择一般遵循下面的方法：
// * 对于只在一个应用程序或者组织内使用的选项，或者用于实验目的的选项，使用字
//   段编号50000~99999范围内的。对于多个选项，用户需要确保不使用相同的编号。
// * 对于可能被多个互不依赖的实体所共同使用的选项，需要给
//   protobuf-global-extension-registry@google.com发邮件来申请预留扩展编号。需
//   要提供工程名称、工程站点，没必要解释为什么需要申请预留某个特定的编号。通
//   常只需要一个扩展编号，可以声明多个选项但是只使用这一个相同的扩展编号。如
//   果申请公共的扩展编号是个刚需，google可能会发布一个web service接口来自动分
//   配选项编号。

message FileOptions {

  // java包名，当前proto文件中生成的java类将位于这个package下
  optional string java_package = 1;

  // 指定一个外部类名称，当前proto文件中生成的所有的类将被封装在这个外部类当中
  optional string java_outer_classname = 8;

  // 如果设置为true，java代码生成器将为每个顶层message、enum、service定义生成
  // 单独的java文件，默认为false
  optional bool java_multiple_files = 10 [default=false];

  // 如果设置为true，java代码生成器将未每个message定义生成equals()、hashCode()
  // 方法，默认为false。本来AbstractMessage基类经包括了一个基于反射的equals()、
  // hashCode()方法实现，这里的这个设置项是一个性能方面的优化
  optional bool java_generate_equals_and_hash = 20 [default=false];

  // 优化类型，生成的类可以进行速度优化、代码尺寸优化
  enum OptimizeMode {
    SPEED = 1;        // Generate complete code for parsing, serialization,
                      // etc.
    CODE_SIZE = 2;    // Use ReflectionOps to implement these methods.
    LITE_RUNTIME = 3; // Generate code using MessageLite and the lite runtime.
  }
  optional OptimizeMode optimize_for = 9 [default=SPEED];

  // 设置go代码的包名
  optional string go_package = 11;

  // 是否应该针对每一门语言都生成generice services? generic服务并不特定于任何
  // 的rpc系统，它是由每个语言的注代码生成器来生成的，不借助于额外的插件。
  // generic services是早期protoo2这个版本说支持的唯一一种服务类型。
  //
  // 由于现在推崇使用plugins，plugins可以生成针对特定rpc系统的代码，generic
  // services现在可以看做是被废弃了。因此，以前proto2总的generice services的默
  // 认设置默认为false，早期的依赖于generic services的代码需要显示设置这些选项
  // 为true。
  optional bool cc_generic_services = 16 [default=false];
  optional bool java_generic_services = 17 [default=false];
  optional bool py_generic_services = 18 [default=false];

  // parser将不识别的选项存储在这里的uinterpreted_option
  repeated UninterpretedOption uninterpreted_option = 999;

  // 用户可以定义自定义选项来扩展当前Message
  extensions 1000 to max;
}

message MessageOptions {
```

```
  // 设为true则使用老的proto1 MessageSet wire format……兼容性目的，没必要使用
  optional bool message_set_wire_format = 1 [default=false];

  // 禁用标准的descriptor()方法的生成，因为如果有个字段名是descriptor的话会生
  // 成一个同名的函数，会冲突。这使得从proto1迁移到后续版本更简单，但是新版本
  // 中还是应该避免使用字段descriptor。
  optional bool no_standard_descriptor_accessor = 2 [default=false];

  // parser将不识别的选项存储在这个字段里
  repeated UninterpretedOption uninterpreted_option = 999;

  // 用户可以定义自定义选项来扩展当前Message
  extensions 1000 to max;
}

message FieldOptions {
  // 开启packed选项之后，对于repeated基本数据类型字段的表示会更加高效。不再针
  // 对repeated字段中的各个元素执行写tag、类型操作，而是将整个数组作为一个固定
  // 长度的blob来存储。
  optional bool packed = 2;

  // 当前字段是否需要lazy parsing? 只是建议，lazy为true，protoc不一定lazy parsing
  optional bool lazy = 5 [default=false];

  // 当前字段是否已经被废弃，跟目标平台相关，这个字段可以为生成的accessor方法
  // 生成Deprecated注解，如果目标平台不支持就会忽略这个选项。不管目标平台是否
  // 支持，proto里面要想废弃一个字段加deprecated选项还是非常正确的做法。
  optional bool deprecated = 3 [default=false];

  // map字段，目前还未完全实现，应避免使用
  optional string experimental_map_key = 9;

  // google内部迁移使用，因避免使用
  optional bool weak = 10 [default=false];

  // 用户可以定义自定义选项来扩展当前Message
  repeated UninterpretedOption uninterpreted_option = 999;

  // 用户自定义选项来扩展message
  extensions 1000 to max;
}

message EnumOptions {

  // 不允许将多个不同的tag names映射到一个相同的值
  // - 意思是说不允许多个字段的编号相同
  optional bool allow_alias = 2 [default=true];

  // 用户可以定义自定义选项来扩展当前Message
  repeated UninterpretedOption uninterpreted_option = 999;

  // 用户自定义选项来扩展message
  extensions 1000 to max;
}

message EnumValueOptions {
  // 用户可以定义自定义选项来扩展当前Message
  repeated UninterpretedOption uninterpreted_option = 999;

  // 用户自定义选项来扩展message
  extensions 1000 to max;
```

```
}

message ServiceOptions {
  // 用户可以定义自定义选项来扩展当前Message
  repeated UninterpretedOption uninterpreted_option = 999;

  // 用户自定义选项来扩展message
  extensions 1000 to max;
}

message MethodOptions {

  // 注意：字段编号1~32被保留给google内部rpc框架使用，google的解释是，在
  // protobuf被公开给外部使用之前内部就已经大量使用了，且1~32倍使用的很多，也
  // 是不得已的事情，总不能为了开源、推广一个内部组件就把自己的生意砸了吧。

  // 用户可以定义自定义选项来扩展当前Message
  repeated UninterpretedOption uninterpreted_option = 999;

  // 用户自定义选项来扩展message
  extensions 1000 to max;
}

// 描述一个parser不认识的option message
// - UninterpretedOption只会出现在compiler::Parser类创建的options protos中；
// - 构建Descriptor对象的时候DescriptorPool会解析UninterpretedOptions；
// 因此，descriptor对象中的options protos（通过Descriptor::options()返回，或者
// 通过Descriptor::CopyTo()生成）是不会包括UinterpretedOptions的。
message UninterpretedOption {
  // uinterpreted选项的名字，name中每个元素的name_part字段都表示name中的点分字
  // 符串的一段，如果name_part是一个扩展（通过在字符串两端用括号括起来表示），
  // is_extension字段为true。
  // 例如，{["foo", false], ["bar.baz",true], ["qux",false]}表示"foo.(bar.baz).qux"。
  message NamePart {
    required string name_part = 1;
    required bool is_extension = 2;
  }
  repeated NamePart name = 2;

  // uinterpreted选项的值，会设置下面字段中其中一个的值
  optional string identifier_value = 3;
  optional uint64 positive_int_value = 4;
  optional int64 negative_int_value = 5;
  optional double double_value = 6;
  optional bytes string_value = 7;
  optional string aggregate_value = 8;
}

// ===================================================================
// Optional source code info

// FileDescriptorProto是从之前的source file中生成的（source file指的是proto文
// 件），这里的SourceCodeInfo指的是proto中的“源代码”信息。
message SourceCodeInfo {
  // Location用于识别proto文件中的源代码片段，往往对应着一个特定的定义。这些
  // Location信息对于IDE、代码索引工具、文档生成工具等是非常重要的。
  //
  // 下面说明一下Location的概念和作用，以下面这个message为例：
  //   message Foo {
  //     optional string foo = 1;
  //   }
```

```
// 我们先只看上面这个message中的字段定义：
//   optional string foo = 1;
//   ^       ^^     ^^  ^  ^^^
//   a       bc     de  f  ghi
// 我们可以得到下面这几个Location：
//   span   path               represents
//   [a,i)  [ 4, 0, 2, 0 ]     The whole field definition.
//   [a,b)  [ 4, 0, 2, 0, 4 ]  The label (optional).
//   [c,d)  [ 4, 0, 2, 0, 5 ]  The type (string).
//   [e,f)  [ 4, 0, 2, 0, 1 ]  The name (foo).
//   [g,h)  [ 4, 0, 2, 0, 3 ]  The number (1).
//
// 每个proto文件解析之后用一个FileDescriptorProto来表示，所以Lcoation路径位
// 置从FileDescriptorProto开始。
// - 因为message Foo是一个message，proto中所有顶层message类型定义都在
//   FileDescriptorProto中message_type字段存储，这个字段的tag是4，所以Location为[4]；
// - 又因为message_type是repeated DescriptorProto类型，因为当前proto示例中
//   Foo为第一个message，所以其在message_type列表中的索引值为0，所以Location为[4,0]；
// - 因为我们现在看的“源代码”是“optional string foo = 1;”，我们需要定位到
//   message中的字段位置，message Foo中的所有字段都在DescriptorProto中的field字
//   段中记录，这个字段的tag=2，所以Location变为[4,0,2]；
// - 又因为这个DescriptorProto中的field为repeated FieldDescriptorProto field，
//   因为这个message中只有一个字段foo，所以foo在field列表中的索引值为0，Location变为[4,0,2,0];
// 上面解释了定位到完整的“optional string foo = 1”定义这个field的Location变
// 化过程，下面再说一下label、type、name、number的Location如何进一步确定。
// FieldDescriptorProto中label的tag位4，type的tag为5，name的tag为1，number的
// tag为3，Location对应的追加索引4、5、1、3。gg!
//
// proto文件中的源代码信息就是由一系列的Location来寻址的。
repeated Location location = 1;
message Location {
  // 前面已经描述了Location的确定过程，一个Location如[4,0,2,0]其中的数字要么
  // 是字段的tag编号要么是repeated列表中的索引值，这里的数字构成的数组保存在
  // path中。
  repeated int32 path = 1 [packed=true];

  // 该字段span总是包括3个或者4个元素，依次表示startline、startcolumn、endline、endcolumn
  repeated int32 span = 2 [packed=true];

  // 如果这个SourceCodeInfo代表一个完整的声明的话，可能在这个声明的前面或者
  // 后面可能有一些attached的注释。
  //
  // 连续的多个行注释看做是一个单独的注释。
  //
  // 这个字段只记录了注释内容，不包括注释内容开头的注释符号//。对于块注释，
  // 注释前面的空白字符、*这几种符号也会被清理掉。但是会包括换行符。
  //
  // Examples:
  //
  //   optional int32 foo = 1;  // Comment attached to foo.
  //   // Comment attached to bar.
  //   optional int32 bar = 2;
  //
  //   optional string baz = 3;
  //   // Comment attached to baz.
  //   // Another line attached to baz.
  //
  //   // Comment attached to qux.
  //   //
  //   // Another line attached to qux.
  //   optional double qux = 4;
```

```
    //
    //   optional string corge = 5;
    //   /* Block comment attached
    //    * to corge.  Leading asterisks
    //    * will be removed. */
    //   /* Block comment attached to
    //    * grault. */
    //   optional int32 grault = 6;

    // Location前面的注释信息
    optional string leading_comments = 3;

    // Location后面的注释信息
    optional string trailing_comments = 4;
  }
}
```

#### *2.4.2. 可以提取proto文件中的哪些信息 & 如何提取#*

前一节2.4.1中对descriptor.proto进行了详细地描述，可以说在proto文件中写的每一行内容都可以通过解析FileDescriptorProto来访问到。proto文件只是一种自描述的消息格式，基于这种格式生成面向特定编程语言的源代码文件时，我们想获取的信息不外乎如下几个：

1. 待生成的源文件的包名；
2. 待生成的源文件的wrapper class类名；
3. proto文件中定义的各个类型，包括枚举enum、消息message、服务service；
4. 对于枚举enum需要知道枚举类型名、列出的枚举值（包括字段、值、注释信息）、注释信息；
5. 对于消息message需要知道类型名、类成员（包括成员类型、成员名称、定义顺序、默认值、注释信息）、注释信息；
6. 对于服务service需要知道服务名称、服务rpc接口（rpc接口的请求参数、返回值类型、注释信息）、注释信息；
7. proto中可以添加注解吗？注解可以提取出来吗？

如何提取上述信息呢？可以肯定地是，只要能拿到当前proto文件对应的FileDescriptorProto，上述内容几乎都可以获取到。但是如何获取到对应的proto文件对应的这个FileDescriptorProto对象呢？下面我们先来看一个protoc插件的示例代码吧，看完之后，大家也就了解了如何获取proto对应的FileDescriptorProto以及如何从中提取想要的1~7上述信息，生成源代码文件也就简单了。

#### *2.4.3. protoc go语言插件protoc-gen-go#*

protoc-gen-go的源代码可以通过通过如下方式获取：

```
git co https://github.com/golang/protobuf
git branch -b ${new-branch}
```

##### *2.4.3.1. protoc-gen-go入口函数分析#*

file: protobuf/protoc-gen-go/main.go

```
package main

import (
    "io/ioutil"
    "os"

    "github.com/c4pt0r/proto"
    "github.com/c4pt0r/protoc-gen-go/generator"
```

```
)

func main() {
    // 首先创建一个代码生成器generator，CodeGeneratorRequest、CodeGeneratorResponse
    // 结构体都被保存在generator中，CodeGenerateResponse中保存着代码生成过程中
    // 的错误状态信息，因此我们可以通过这个结构体提取错误状态并进行错误处理
    g := generator.New()

    // 从标准输入中读取CodeGeneratorRequest信息（标准输入已经被重定向到了父进程
    // protoc进程创建的管道stdout_pipe的读端，父进程会从管道的写端写入该请求信息）
    data, err := ioutil.ReadAll(os.Stdin)
    if err != nil {
        g.Error(err, "reading input")
    }

    // 读取到的数据时串行化之后的CodeGeneratorRequest，将其反串行化成CodeGeneratorRequest
    if err := proto.Unmarshal(data, g.Request); err != nil {
        g.Error(err, "parsing input proto")
    }

    // 检查CodeGeneratorRequest中待生成的源代码文件数量，数量为0则无需生成
    if len(g.Request.FileToGenerate) == 0 {
        g.Fail("no files to generate")
    }

    // 将CodeGeneratorRequest中传递给代码生成器的参数设置到protoc插件的代码生成器中
    g.CommandLineParameters(g.Request.GetParameter())

    // 前面的proto.Unmarshal(...)操作将stdin中的请求反串行化成了CodeGeneratorRequest，
    // 这里的g.WrapTypes()将请求中的一些descriptors进行进一步封装，方便后面引用
    g.WrapTypes()

    g.SetPackageNames()
    g.BuildTypeNameMap()

    // 生成所有的源代码文件
    g.GenerateAllFiles()

    // 将CodeGeneratorResponse对象进行串行化处理
    data, err = proto.Marshal(g.Response)
    if err != nil {
        g.Error(err, "failed to marshal output proto")
    }
    // 将串行化之后的CodeGenerateResponse对象数据写入标准输出（标准输出已经被
    // 重定向到了父进程protoc进程创建的管道stdin_pipe的写端，父进程从管道的读
    // 端读取这里的响应）
    _, err = os.Stdout.Write(data)
    if err != nil {
        g.Error(err, "failed to write output proto")
    }
}
```

#### *2.4.3.2.* 回顾一下*CodeGeneratorRequest & CodeGeneratorResponse*的定义#

下面看下CodeGeneratorRequest和CodeGeneratorResponse的定义。

file:
${protobuf}/src/google/protobuf/compiler/plugin.go

```
// 串行化后的CodeGeneratorRequest信息会被写入到插件程序的stdin
message CodeGeneratorRequest {
```

```
  // protoc命令执行时，我们在命令行中列出了需要进行处理的.proto文件的名称，代
  // 码生成器应该只为这些.proto文件生成源代码文件。每一个.proto文件成功解析之
  // 后会生成一个FileDescriptorProto对象，这个对象会被加入到字段proto_file中
  repeated string file_to_generate = 1;

  // protoc命令行程序中传递给插件程序代码生成器的参数信息
  optional string parameter = 2;

  // protoc命令行中列出的所有的.proto文件被添加到了字段file_to_generate中，这
  // 些.proto文件中通过import引入进来的文件，这两部分文件解析成功后对应的
  // FileDescriptorProto对象都会被加入到这里的proto_file中，添加后的顺序是按照
  // 拓扑顺序排序的，怎么讲？就是被import的proto文件会出现在import它们的 proto
  // 文件前面。
  repeated FileDescriptorProto proto_file = 15;
}

// 串行化后的CodeGeneratorResponse信息会被写入到插件的stdout
message CodeGeneratorResponse {
  // 如果错误信息非空，表示代码生成失败。这种情况下尽管代码生成失败，插件进程
  // 仍然应该返回一个状态0。
  //
  // 这个字段用于指示.proto文件错误，.proto文件中的错误将使得代码生成器无法生
  // 成正确的代码。指示protoc本身的错误，例如CodeGeneratorRequest数据无法被正
  // 确地反串行化，这种情况应该被报告，错误信息应该写到stderr并且插件进程应该
  // 返回一个非0状态码
  optional string error = 1;

  // 描述一个待生成的源代码文件
  message File {
    // 待生成的源代码文件相对于输出目录的文件名
    optional string name = 1;

    // 写入到源代码文件中的插入点信息，方便后面的插件在插入点处进行扩展其他内容
    optional string insertion_point = 2;

    // 写入到文件或者文件插入点位置的内容
    optional string content = 15;
  }

  // 所有的待生成的源代码文件列表
  repeated File file = 15;
}
```

*2.4.3.3. generator实现分析#*

main.go中调用了generator的几个关键方法，我们先来看下这
几个方法都做了些什么，然 后再跟进一步看看generator的详细
实现过程。

**2.4.3.3.1. generator.New()#**

file: ${golang-protobuf}/protoc-gen-
go/generator/generator.go

```
// Generator类型的方法能够输出源代码，这些输出的源代码信息存储在Response成员中
type Generator struct {
    *bytes.Buffer

    Request  *plugin.CodeGeneratorRequest  // The input.
    Response *plugin.CodeGeneratorResponse // The output.

    Param             map[string]string // Command-line parameters.
    PackageImportPath string            // Go import path of the package we're generating code
    ImportPrefix      string            // String to prefix to imported package file names.
```

```
    ImportMap         map[string]string // Mapping from .proto file name to import path

    Pkg map[string]string // The names under which we import support packages

    packageName      string                     // What we're calling ourselves.
    allFiles         []*FileDescriptor          // All files in the tree
    allFilesByName   map[string]*FileDescriptor // All files by filename.
    genFiles         []*FileDescriptor          // Those files we will generate output for.
    file             *FileDescriptor            // The file we are compiling now.
    usedPackages     map[string]bool            // Names of packages used in current file.
    typeNameToObject map[string]Object          // Key is a fully-qualified name in input synt
    init             []string                   // Lines to emit in the init function.
    indent           string
    writeOutput      bool
}

// 创建一个新的代码生成器，并创建请求、响应对象
func New() *Generator {
    g := new(Generator)
    g.Buffer = new(bytes.Buffer)
    g.Request = new(plugin.CodeGeneratorRequest)
    g.Response = new(plugin.CodeGeneratorResponse)
    return g
}
```

**2.4.3.3.2. generator.CommandLineParameters(…)#**

这个函数是负责解析protoc传递过来的命令行参数信息的。

file: ${golang-protobuf}/protoc-gen-go/generator/generator.go

```
// 将都好分隔的key=value列表解析成<key,value> map
func (g *Generator) CommandLineParameters(parameter string) {
    g.Param = make(map[string]string)
    for _, p := range strings.Split(parameter, ",") {
        if i := strings.Index(p, "="); i < 0 {
            g.Param[p] = ""
        } else {
            g.Param[p[0:i]] = p[i+1:]
        }
    }

    g.ImportMap = make(map[string]string)
    pluginList := "none" // Default list of plugin names to enable (empty means all).
    for k, v := range g.Param {
        switch k {
        case "import_prefix":
            g.ImportPrefix = v
        case "import_path":
            g.PackageImportPath = v
        // --go_out=plugins=grpc:., 解析这里的参数plugins=grpc
        case "plugins":
            pluginList = v
        default:
            if len(k) > 0 && k[0] == 'M' {
                g.ImportMap[k[1:]] = v
            }
        }
    }

    // 在protoc-gen-go的某个地方已经将grpc插件注册到了当前generator（也就是添
    // 加到plugins []Plugin中），但是到底是在哪里注册的呢？只有注册并激活（参
```

```
    // 数中通过--go_out=plugins=grpc:.)grpc子插件，该子插件才能被使用于后续的
    // 代码生成过程中（生成rpc相关的go源代码）。
    //
    // 其实这里的grpc子插件注册是利用了link_grpc.go里面的import _操作来隐式地
    // 调用了grpc.init()方法，该初始化方法中负责完成向generator的注册操作，即
    // generator.RegisterPlugin(new(grpc))，这里的RegisterPlugin其实就是将指定
    // 的子插件加入到plugins []Plugin slice中。
    // 为了能够在protoc-gen-go中正确地将grpc link进去，在构建protoc-gen-go的时
    // 候需要执行命令：
    // cd protoc-gen-go & go build main.go link_grpc.go
    // go build的时候如果没有列出link_grpc.go，那么grpc是不会被link进
    // protoc-gen-go这个插件的，这样处理.proto文件中的service时插件是不会生成
    // service相关的go源代码的。

    // 根据--go_out=plugins=?+?+?:.，更新激活的插件列表
    if pluginList != "" {
        // Amend the set of plugins.
        enabled := make(map[string]bool)
        for _, name := range strings.Split(pluginList, "+") {
            enabled[name] = true
        }
        var nplugins []Plugin
        for _, p := range plugins {
            if enabled[p.Name()] {
                nplugins = append(nplugins, p)
            }
        }
        plugins = nplugins
    }
}
```

#### 2.4.3.3.3. generator.WrapTypes()#

file: ${golang-protobuf}/protoc-gen-go/generator/generator.go

```
// WrapTypes walks the incoming data, wrapping DescriptorProtos, EnumDescriptorProtos
// and FileDescriptorProtos into file-referenced objects within the Generator.
// It also creates the list of files to generate and so should be called before GenerateAllFi
func (g *Generator) WrapTypes() {
    g.allFiles = make([]*FileDescriptor, 0, len(g.Request.ProtoFile))
    g.allFilesByName = make(map[string]*FileDescriptor, len(g.allFiles))
    for _, f := range g.Request.ProtoFile {
        // We must wrap the descriptors before we wrap the enums
        descs := wrapDescriptors(f)
        g.buildNestedDescriptors(descs)
        enums := wrapEnumDescriptors(f, descs)
        g.buildNestedEnums(descs, enums)
        exts := wrapExtensions(f)
        fd := &FileDescriptor{
            FileDescriptorProto: f,
            desc:                descs,
            enum:                enums,
            ext:                 exts,
            exported:            make(map[Object][]symbol),
            proto3:              fileIsProto3(f),
        }
        extractComments(fd)
        g.allFiles = append(g.allFiles, fd)
        g.allFilesByName[f.GetName()] = fd
    }
    for _, fd := range g.allFiles {
        fd.imp = wrapImported(fd.FileDescriptorProto, g)
```

```
    }

    g.genFiles = make([]*FileDescriptor, 0, len(g.Request.FileToGenerate))
    for _, fileName := range g.Request.FileToGenerate {
        fd := g.allFilesByName[fileName]
        if fd == nil {
            g.Fail("could not find file named", fileName)
        }
        fd.index = len(g.genFiles)
        g.genFiles = append(g.genFiles, fd)
    }
}
```

#### 2.4.3.3.4. generator.GenerateAllFiles()#

调用generator针对所有解析成功的proto文件生成所有的go源代码

file: ${golang-protobuf}/protoc-gen-go/generator/generator.go

```
// 生成所有.proto文件对应的go源代码，这里只是将源代码内容存储到g.Response中，
// 并没有直接创建源代码文件，插件将Response传递给protoc进程后由protoc进程来负
// 责创建源代码文件
func (g *Generator) GenerateAllFiles() {
    // Initialize the plugins
    for _, p := range plugins {
        p.Init(g)
    }
    // Generate the output. The generator runs for every file, even the files
    // that we don't generate output for, so that we can collate the full list
    // of exported symbols to support public imports.
    genFileMap := make(map[*FileDescriptor]bool, len(g.genFiles))
    for _, file := range g.genFiles {
        genFileMap[file] = true
    }
    for _, file := range g.allFiles {
        g.Reset()
        g.writeOutput = genFileMap[file]

        // 调用generator的generate(...)方法来生成该proto文件的
        // FileDescriptorProto描述对应的go源代码
        g.generate(file)

        if !g.writeOutput {
            continue
        }
        g.Response.File = append(g.Response.File, &plugin.CodeGeneratorResponse_File{
            Name:    proto.String(file.goFileName()),
            Content: proto.String(g.String()),
        })
    }
}
```

再看下generator.generate(…)方法是如何实现的。

```
// 针对.proto文件（由FileDescriptor表示）生成对应的go源代码
func (g *Generator) generate(file *FileDescriptor) {
    g.file = g.FileOf(file.FileDescriptorProto)
    g.usedPackages = make(map[string]bool)

    // 要生成源代码的首个proto文件对应的go源代码，这部分代码顶部插入版权信息
    if g.file.index == 0 {
        // For one file in the package, assert version compatibility.
```

```
    g.P("// This is a compile-time assertion to ensure that this generated file")
    g.P("// is compatible with the proto package it is being compiled against.")
    g.P("// A compilation error at this line likely means your copy of the")
    g.P("// proto package needs to be updated.")
    g.P("const _ = ", g.Pkg["proto"], ".ProtoPackageIsVersion", generatedCodeVersion, " /
    g.P()
}

// 生成import语句
for _, td := range g.file.imp {
    g.generateImported(td)
}
// 生成enum类型定义语句
for _, enum := range g.file.enum {
    g.generateEnum(enum)
}
// 生成message类型定义语句
for _, desc := range g.file.desc {
    // Don't generate virtual messages for maps.
    if desc.GetOptions().GetMapEntry() {
        continue
    }
    g.generateMessage(desc)
}
// 生成extension类型定义语句
for _, ext := range g.file.ext {
    g.generateExtension(ext)
}
// 生成初始化函数语句
g.generateInitFunction()

// 前面生成enum、message、extension等的方式都基本类似，后面我们只给出一个
// 生成枚举类型方法的说明，生成message、extension的实现方法可以执行查看
// generator.go中的实现。
//
// 需要注意的是，前面的各个生成源代码的方法不能处理service服务定义的rpc接
// 口代码，这部分rpc代码的生成需要借助于grpc子插件来完成，即下面的g.runPlugins(...)
g.runPlugins(file)

g.generateFileDescriptor(file)

// 待输出的源代码需要知道哪些package是需要import的，哪些不需要，因此先运行
// 插件生成go代码中除import之外的其他部分代码，然后知道了哪些package需要
// import，再插入具体的import语句。
//
// 最后在go源代码中插入header、import
rem := g.Buffer
g.Buffer = new(bytes.Buffer)
g.generateHeader()
g.generateImports()
if !g.writeOutput {
    return
}
g.Write(rem.Bytes())

// 重新格式化生成的go源代码 (gofmt)
fset := token.NewFileSet()
raw := g.Bytes()
ast, err := parser.ParseFile(fset, "", g, parser.ParseComments)
if err != nil {
    // Print out the bad code with line numbers.
```

```
        // This should never happen in practice, but it can while changing generated code,
        // so consider this a debugging aid.
        var src bytes.Buffer
        s := bufio.NewScanner(bytes.NewReader(raw))
        for line := 1; s.Scan(); line++ {
            fmt.Fprintf(&src, "%5d\t%s\n", line, s.Bytes())
        }
        g.Fail("bad Go source code was generated:", err.Error(), "\n"+src.String())
    }
    g.Reset()
    err = (&printer.Config{Mode: printer.TabIndent | printer.UseSpaces, Tabwidth: 8}).Fprint(g
    if err != nil {
        g.Fail("generated Go source code could not be reformatted:", err.Error())
    }
}
```

上面generate.generate(…)方法中generateEnum()、generateMessage()方法与其他几个方法都是非常类似的，由于大家使用protobuf过程中使用enum、message比较多，并且generateEnum()、generateMessage()方法执行逻辑非常相似，考虑到篇幅方面generateEnum()比generateMessage()简短，这里我们就只以generateEnum()的源代码作为示例进行分析。相信如果看懂了generateEnum的实现思路，generateMessage的实现思路也很容易搞明白，读者也具备了自己实现子插件的能力。

```
// 生成指定enum类型的go源代码
func (g *Generator) generateEnum(enum *EnumDescriptor) {
    // enum类型的完整类型名
    typeName := enum.TypeName()
    // CamelCased之后的完整类型名
    ccTypeName := CamelCaseSlice(typeName)
    ccPrefix := enum.prefix()

    // 打印enum类型定义之前的leading comments
    // - 提取源代码信息SourceCodeInfo都是通过Location path来获取的;
    // - 提取注释信息也不例外，下面我们会介绍PrintComments(path)如何通过
    //   Location path来生成注释信息;
    g.PrintComments(enum.path)

    // 生成枚举类型的定义起始部分：type 枚举类型名 int32
    g.P("type ", ccTypeName, " int32")
    g.file.addExport(enum, enumSymbol{ccTypeName, enum.proto3()})

    // 枚举类型里面的各个枚举值都作为const int32常量来定义
    g.P("const (")

    // 枚举值定义之前缩进一下
    g.In()

    // 针对枚举类型里面的所有枚举值进行源代码生成
    for i, e := range enum.Value {
        // 生成枚举值前面的leading comments
        g.PrintComments(fmt.Sprintf("%s,%d,%d", enum.path, enumValuePath, i))
        // 生成枚举值的name = value形式的go源代码
        name := ccPrefix + *e.Name
        g.P(name, " ", ccTypeName, " = ", e.Number)
        g.file.addExport(enum, constOrVarSymbol{name, "const", ccTypeName})
    }

    // 枚举值定义完之后取消缩进
```

```
    g.Out()

    // 打印最后的结束信息
    g.P(")")

    // 生成枚举类型相关的两个map
    // - 其中一个是枚举值到枚举名的映射;
    // - 另一个是枚举名到枚举值的映射;
    g.P("var ", ccTypeName, "_name = map[int32]string{")
    g.In()
    // 第一个map
    generated := make(map[int32]bool) // avoid duplicate values
    for _, e := range enum.Value {
        duplicate := ""
        if _, present := generated[*e.Number]; present {
            duplicate = "// Duplicate value: "
        }
        g.P(duplicate, e.Number, ": ", strconv.Quote(*e.Name), ",")
        generated[*e.Number] = true
    }
    g.Out()
    g.P("}")
    // 第二个map
    g.P("var ", ccTypeName, "_value = map[string]int32{")
    g.In()
    for _, e := range enum.Value {
        g.P(strconv.Quote(*e.Name), ": ", e.Number, ",")
    }
    g.Out()
    g.P("}")

    // 其他处理动作，也会生成部分源代码，这里可以忽略不计了
    // ...
}
```

下面看一下PrintComments如何通过Location path来提取并打印关联的注释信息。

```
// 打印.proto文件中对该location path关联的leading comments注释信息
func (g *Generator) PrintComments(path string) bool {
    if !g.writeOutput {
        return false
    }

    // 在protoc进程解析.proto文件的时候就已经将各个类型、字段的comments信息维
    // 护起来了，k就是location的path，通过path就能获取到对应的location，每个
    // location中保存了这个位置的源代码的leading comments、trailing comments信
    // 息，这里只打印leading comments
    if loc, ok := g.file.comments[path]; ok {
        text := strings.TrimSuffix(loc.GetLeadingComments(), "\n")
        for _, line := range strings.Split(text, "\n") {
            g.P("// ", strings.TrimPrefix(line, " "))
        }
        return true
    }

    return false
}
```

看到这里我们对于基本的enum、message类型定义等都基本清楚了，下面我们需要看一下grpc子插件是如何生成service服务的rpc接口源代码的，这样的话，就得再来看一下g.runPlugins(file)是如何实现的。

```
// Run all the plugins associated with the file.
func (g *Generator) runPlugins(file *FileDescriptor) {
    // 在上述generator处理的基础上，继续运行generator中注册的插件，依次运行插件
    for _, p := range plugins {
        p.Generate(file)
    }
}
```

因为上述runPlugins(…)执行过程中，plugins这个slice内只有一个有效的、激活的子插件grpc，因此如果我们想了解service服务对应的rpc接口的实现方式，我们需要就只需要了解grpc这个插件的Generate(file)方法就可以了。

```
// 生成.proto文件中service定义的rpc接口的go源代码
func (g *grpc) Generate(file *generator.FileDescriptor) {

    // 如果没有定义service服务直接返回
    if len(file.FileDescriptorProto.Service) == 0 {
        return
    }

    // 相关变量定义
    g.P("// Reference imports to suppress errors if they are not otherwise used.")
    g.P("var _ ", contextPkg, ".Context")
    g.P("var _ ", grpcPkg, ".ClientConn")
    g.P()

    // 断言，检查版本兼容性
    g.P("// This is a compile-time assertion to ensure that this generated file")
    g.P("// is compatible with the grpc package it is being compiled against.")
    g.P("const _ = ", grpcPkg, ".SupportPackageIsVersion", generatedCodeVersion)
    g.P()

    // 针对所有的service定义生成相关的service的go源代码
    for i, service := range file.FileDescriptorProto.Service {
        g.generateService(file, service, i)
    }
}

// grpc中对generateService的实现，生成service相关的go源代码
// @param .proto解析后的各种DescriptorProto的wrapping类，通过它可以方便地访问.proto中定义的东西
// @param .proto中的某个service解析后对应的ServiceDescriptorProto
// @param .proto中可能定义了多个service，当前这个service对应的索引值
func (g *grpc) generateService(file *generator.FileDescriptor, service *pb.ServiceDescriptorPr
    // 构建当前service对应的path!
    path := fmt.Sprintf("6,%d", index) // 6 means service.

    // 获取service名称
    origServName := service.GetName()
    fullServName := origServName
    if pkg := file.GetPackage(); pkg != "" {
        fullServName = pkg + "." + fullServName
    }
    servName := generator.CamelCase(origServName)

    // 准备生成client相关的go源代码
    g.P()
    g.P("// Client API for ", servName, " service")
    g.P()

    // 服务用户端go源代码生成
    // - type 服务名+Client interface
```

```
    g.P("type ", servName, "Client interface {")
    // - 服务用户端定义的各个接口方法
    for i, method := range service.Method {

        // 打印接口的leading comments
        g.gen.PrintComments(fmt.Sprintf("%s,2,%d", path, i)) // 2 means method in a service.
        // 生成接口的签名
        g.P(g.generateClientSignature(servName, method))
    }
    g.P("}")
    g.P()

    // 服务的用户端struct，其中包括了一个cc *grpc.ClientConn，后面会在该struct
    // 上实现上述服务接口
    g.P("type ", unexport(servName), "Client struct {")
    g.P("cc *", grpcPkg, ".ClientConn")
    g.P("}")
    g.P()

    // NewClient工厂
    g.P("func New", servName, "Client (cc *", grpcPkg, ".ClientConn) ", servName, "Client {")
    g.P("return &", unexport(servName), "Client{cc}")
    g.P("}")
    g.P()

    var methodIndex, streamIndex int
    serviceDescVar := "_" + servName + "_serviceDesc"

    // 服务用户端的接口方法实现
    for _, method := range service.Method {
        var descExpr string
        if !method.GetServerStreaming() && !method.GetClientStreaming() {
            // Unary RPC method
            descExpr = fmt.Sprintf("&%s.Methods[%d]", serviceDescVar, methodIndex)
            methodIndex++
        } else {
            // Streaming RPC method
            descExpr = fmt.Sprintf("&%s.Streams[%d]", serviceDescVar, streamIndex)
            streamIndex++
        }
        g.generateClientMethod(servName, fullServName, serviceDescVar, method, descExpr)
    }

    g.P("// Server API for ", servName, " service")
    g.P()

    // 服务端接口go源代码生成
    ...
}
```

看完之后我们已经清楚了generator做了什么、grpc子插件又做了什么，以及各自的细节是什么样的。下面我们将在此学习、理解的基础上开发一个自己的protoc插件。