精通protobuf原理系列之（三）

一文彻底搞懂反射原理

精通 protobuf 原理之三：一文彻底搞懂反射原理

津的技术专栏
后端架构师
关注他

1 说在前面

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 otobuf，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 原理之三：一文彻底搞懂反射原理」；
- 「精通 protobuf 原理之四：反射实践，和json的相互转换」；
- 「精通 protobuf 原理之五：一文彻底搞懂 RPC 原理」；
- 「精通 protobuf 原理之六：自己动手写一个 RPC 框架」；
- 「精通 protobuf 原理之七：一文彻底搞懂 Arena 分配器原理剖析」。
- 后续的待定……

本文是系列文章的第三篇，主讲 protobuf 反射原理。本文适合 protobuf 入门、进阶的开发者阅读，是一篇讲原理的文章，主要是深入介绍了 protobuf 反射的底层原理。通过阅读本文，开发者能够对 protobuf 反射原理有深入的理解，对如何更好的运用 protobuf 反射特性提供很大的参考价值。

如果你还在为protobuf 反射原理存在很多问号?

如果让你自己实现一个反射组件，你还不知道怎么实现?

那么通过这篇文章，可以帮你解决这些问题。

文章内容有点长，可能需要阅读 5~10分钟。

2 什么是反射

这里所说的“反射”，指是程序在运行时能够动态的获取到一个类型的元信息的一种操作。而知道了该类型的元信息，就可以利用元信息构造出该类型的实例，并对该实例进行读写操作。和明确地指定一个变量的类型的区别是，后者是在编译阶段就已经生成了该类型的实例，而前者（反射）的过程是在运行时完成，或者说是在运行时推算出该实例的类型。

3 先上一个示例

先上 echo.proto 源码：

```
syntax = "proto3";
package self;

option cc_generic_services = true;

enum QueryType {
  PRIMMARY = 0;
  SECONDARY = 1;
};

message EchoRequest {
  QueryType querytype = 1;
  string payload = 2;
}

message EchoResponse {
  int32 code = 1;
  string msg = 2;
}

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

测试源代码这么写（test_reflection.cc）：

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

void test_relection() {
  const std::string type_name = "self.EchoRequest";
  /*
   * ① 在 DescriptorPool 中检索 self.EchoRequest
   *    Message 类型的 discriptor 元数据
   */
  const google::protobuf::Descriptor* descriptor
    = google::protobuf::DescriptorPool::generated_pool()
      ->FindMessageTypeByName(type_name);
  if (descriptor == nullptr) {
    std::cout << "[ERROR] Cannot found " << type_name
          << " in DescriptorPool" << std::endl;
    return;
  }
  /*
   * ② 通过 discriptor 元信息在 MessageFactory 检索类型工厂，
   *    用于创建该类型的实例。
   */
  const google::protobuf::Message* prototype
    = google::protobuf::MessageFactory::generated_factory()
      ->GetPrototype(descriptor);
  /*
   * ③ 创建 self.EchoRequest 类型的 Message 实例。
   *    google::protobuf::Message 是所有 Message
   *    类型的基类。
   */
  google::protobuf::Message* req_msg = prototype->New();
  /*
   * ④ 因为只知道基类的实例指针，需要 Reflection 信息协助判断
   *    具体类型。
   */
```

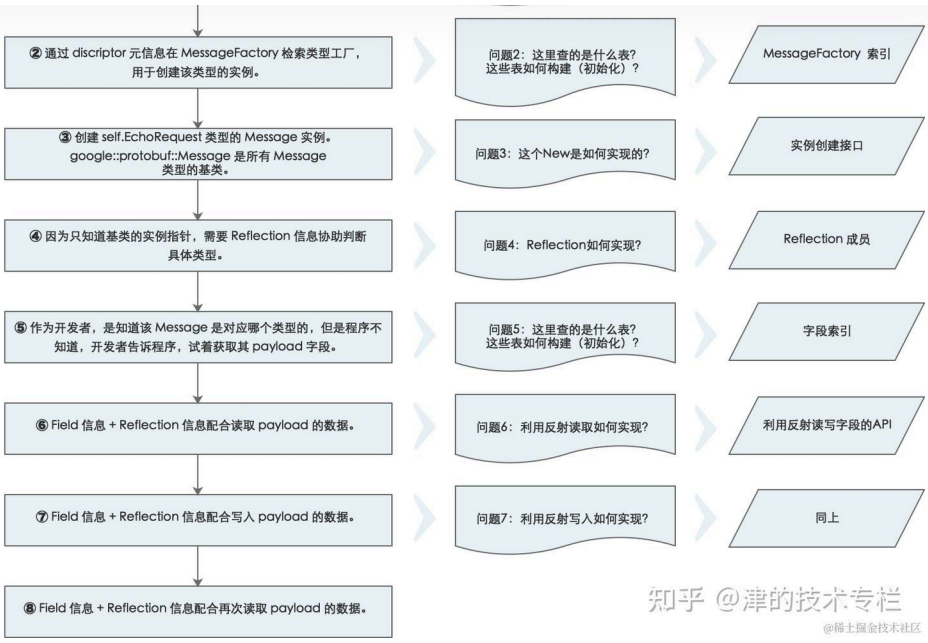
```
     *    开发者告诉程序，试看获取其 payload 字段。
     */
    const google::protobuf::FieldDescriptor *req_msg_ref_field_payload
      = descriptor->FindFieldByName("payload");

    /*
     * ⑥ Field 信息 + Reflection 信息配合读取 payload 的数据。
     */
    std::cout << "before set, ref_req_msg_payload: "
              << req_msg_ref->GetString(*req_msg, req_msg_ref_field_payload)
              << std::endl;
    /*
     * ⑦ Field 信息 + Reflection 信息配合写入 payload 的数据。
     */
    req_msg_ref->SetString(req_msg, req_msg_ref_field_payload, "my payload");
    /*
     * ⑧ Field 信息 + Reflection 信息配合再次读取 payload 的数据。
     */
    std::cout << "after set, ref_req_msg_payload: "
              << req_msg_ref->GetString(*req_msg, req_msg_ref_field_payload)
              << std::endl;
}

int main() {
    test_relection();
    return 0;
}
```

看似写了很多源代码，但是其实就做了一个事情，定义个 self::EchoRequest 变量，然后对其进行读和写。编译执行得到结果：

```
$ ./test_reflection
before set, ref_req_msg_payload:
after set, ref_req_msg_payload: my payload
```

> PS：需要注意的是这里其实有一个坑，笔者猜测可能是编译器优化的原因造成的。现象是会输出 “[ERROR] Cannot found in DescriptorPool”。因为main函数中没有使用到 echo.proto 中的任何类型，编译器认为没有使用到echo.proto 的代码，所以不让程序执行以下变量的初始化，从而导致索引没有初始化：
> PROTOBUF_ATTRIBUTE_INIT_PRIORITY static ::PROTOBUF_NAMESPACE_ID::internal::AddDescriptorsRunner dynamic_init_dummy_echo_2eproto(&descriptor_table_echo_2eproto);
> 其原理会在后续的章节中涉及到。
> 解决办法：在 main 函数中加上一行 self::EchoRequest req 的代码即可，让编译器以为使用到了echo.proto 中的类型。

那么接下来笔者的思路就是通过分析 ① ~ ⑧ 各个步骤的实现原理来了解反射是如何工作的。



4 原理分析

4.1 DescriptorPool 索引

4.1.1 google::protobuf::Descriptor

前面示例中有使用到 DescriptorPool 的 FindMessageTypeByName 接口函数（如下代码），这里的目的是获取该Message 的元信息。这里获取元信息的过程是一个查表的过程，本节主要了解一下此表的索引是什么原理，以及如何构建的。

```
const google::protobuf::Descriptor* descriptor
  = google::protobuf::DescriptorPool::generated_pool()
    ->FindMessageTypeByName(type_name);
```

4.1.2 DescriptorPool 索引的构建时机

这里会用到 google::protobuf::internal::AddDescriptorsRunner（下简称 AddDescriptorsRunner），它的实现比较简单，如下代码，先看看它的原型。

```
struct PROTOBUF_EXPORT AddDescriptorsRunner {
  explicit AddDescriptorsRunner(const DescriptorTable* table);
};
AddDescriptorsRunner::AddDescriptorsRunner(const DescriptorTable* table) {
  AddDescriptors(table);
}
```

从如上代码中，可以看出执行构造函数的时候会触发 Descriptor 表的构建。那什么情况下会执行构造函数呢？我们在 echo.pb.cc 源代码文件中找到这样一行代码（如下），这行代码的作用是定义一个静态类型的 AddDescriptorsRunner 类型的变量，因为是静态类型的，所以在程序启动是生成，在程序退出时销毁。而在定义该变量时回触发构造函数的调用，所以我们不难理解，DescriptorPool 索引的构建时机是程序启动的时候，销毁时机是在程序退出的时候。

```
PROTOBUF_ATTRIBUTE_INIT_PRIORITY static ::PROTOBUF_NAMESPACE_ID::internal::AddDescript
```

4.1.3 DescriptorPool索引的构建原理

我们先从 AddDescriptors 函数开始分析。

```
void AddDescriptors(const DescriptorTable* table) {
  if (table->is_initialized) return;
  table->is_initialized = true;
  AddDescriptorsImpl(table);
}

void AddDescriptorsImpl(const DescriptorTable* table) {
  // Reflection refers to the default fields so make sure they are initialized.
  internal::InitProtobufDefaults();

  // Ensure all dependent descriptors are registered to the generated descriptor
  // pool and message factory.
  int num_deps = table->num_deps;
  for (int i = 0; i < num_deps; i++) {
    // In case of weak fields deps[i] could be null.
    if (table->deps[i]) AddDescriptors(table->deps[i]);
  }

  // Register the descriptor of this file.
  DescriptorPool::InternalAddGeneratedFile(table->descriptor, table->size);
  MessageFactory::InternalRegisterGeneratedFile(table);
}
```

可以看出，总共三个步骤：

1. 初始化变量：反射需要使用的变量，先确保其已经初始化 了；
2. 解析依赖：如果有import 其他proto 源文件，那么先解析其他proto源文件，存在 deps 中。
3. 注册 Descriptor：

- 构建 DescriptorPool 索引（DescriptorPool::InternalAddGeneratedFile）；
- 构建MessageFactory 索引（MessageFactory::InternalRegisterGeneratedFile）。

本节我们主要分析构建 DescriptorPool 索引的实现原理，至于MessageFactory 索引的实现原理我们在下一节中再详细分析。

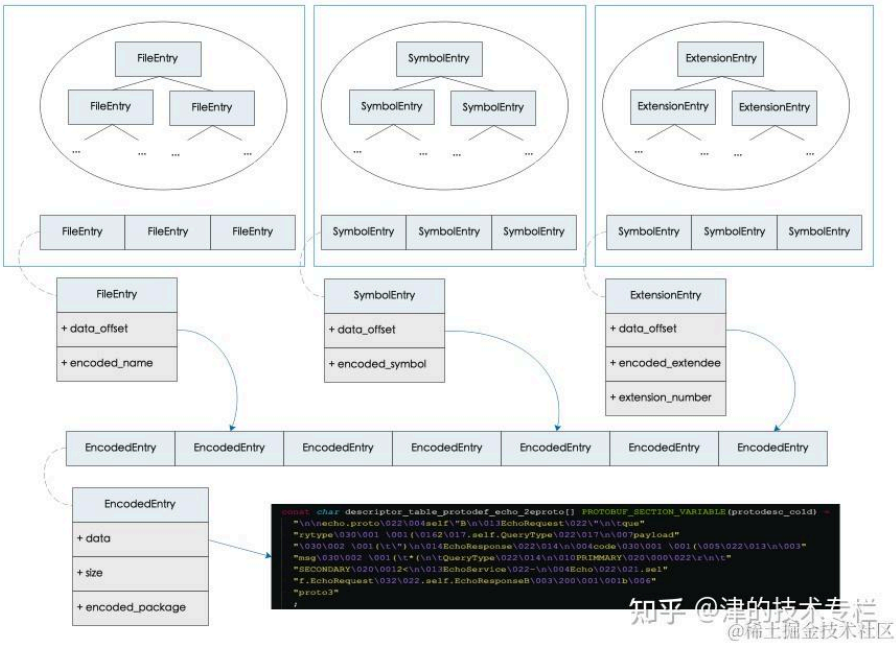
database 是一个比较抽象的名称，database 底层实现其实就是索引 index_，这里会先把文件定义信息先解析，主要是确定 encoded_file_descriptor 信息是正确的。

```
void DescriptorPool::InternalAddGeneratedFile(
    const void* encoded_file_descriptor, int size) {
  GOOGLE_CHECK(GeneratedDatabase()->Add(encoded_file_descriptor, size));
}

bool EncodedDescriptorDatabase::Add(const void* encoded_file_descriptor, int size) {
  FileDescriptorProto file;
  if (file.ParseFromArray(encoded_file_descriptor, size)) {
    return index_->AddFile(file, std::make_pair(encoded_file_descriptor, size));
  } else {
    GOOGLE_LOG(ERROR) << "Invalid file descriptor data passed to "
                  "EncodedDescriptorDatabase::Add().";
    return false;
  }
}
```

DescriptorIndex 的索引如下：

```
/*数据表，存的是最终数据，包括：
/* - 文件元数据
 * - 标签元数据
 * - 扩展元数据 */
std::vector<EncodedEntry> all_values_;
/* 文件元数据索引，指向 all_values_ 的位置（即下标） */
std::set<FileEntry, FileCompare> by_name{FileCompare{*this}};
std::vector<FileEntry> by_name_flat;
/* 标签元数据索引，包括：
 * - message
 * - enum
 * - externion
 * - service */
std::set<SymbolEntry, SymbolCompare> by_symbol{SymbolCompare{*this}};
std::vector<SymbolEntry> by_symbol_flat;
/* 扩展元数据索引 */
std::set<ExtensionEntry, ExtensionCompare> by_extension{ExtensionCompare{*this}};
std::vector<ExtensionEntry> by_extension_flat;
```

关系图如下图所示：



AddFile 实现如下：

```
template <typename FileProto>
bool EncodedDescriptorDatabase::DescriptorIndex::AddFile(const FileProto& file,
                                                         Value value) {
  // We push `value` into the array first. This is important because the AddXXX
  // functions below will expect it to be there.

  //############## 数据表 ##############
  all_values_.push_back({value.first, value.second, {}});

  if (!ValidateSymbolName(file.package())) {
    GOOGLE_LOG(ERROR) << "Invalid package name: " << file.package();
    return false;
```

```
  // 1. 文件元信息索引
  if (!InsertIfNotPresent(
          &by_name_, FileEntry{static_cast<int>(all_values_.size() - 1),
                               EncodeString(file.name())}) ||
      std::binary_search(by_name_flat_.begin(), by_name_flat_.end(),
                         file.name(), by_name_.key_comp())) {
    GOOGLE_LOG(ERROR) << "File already exists in database: " << file.name();
    return false;
  }

  // 2. 类型索引
  // - 所有类型都会进 symbol 表
  // - extension 会进入 extension 表
  for (const auto& message_type : file.message_type()) {
    if (!AddSymbol(message_type.name())) return false;
    if (!AddNestedExtensions(file.name(), message_type)) return false;
  }
  for (const auto& enum_type : file.enum_type()) {
    if (!AddSymbol(enum_type.name())) return false;
  }
  for (const auto& extension : file.extension()) {
    if (!AddSymbol(extension.name())) return false;
    if (!AddExtension(file.name(), extension)) return false;
  }
  for (const auto& service : file.service()) {
    if (!AddSymbol(service.name())) return false;
  }

  return true;
}
```

讲到这里，构建索引的实现原理就告了一段落，但是你以为构建索引已经结束了吗？当然没有！在下一节中分析。

4.1.4 DescriptorPool索引的查询过程

我们还是从一行代码开始（DescriptorPool 的 FindMessageTypeByName 接口函数）。

```
const google::protobuf::Descriptor* descriptor
  = google::protobuf::DescriptorPool::generated_pool()
  ->FindMessageTypeByName("self.EchoRequest");
```

FindMessageTypeByName 函数实际上调用了 tables_ 成员的 FindByNameHelper 成员函数。

```
const Descriptor* DescriptorPool::FindMessageTypeByName(
    ConstStringParam name) const {
  Symbol result = tables_->FindByNameHelper(this, name);
  return (result.type == Symbol::MESSAGE) ? result.descriptor : nullptr;
}
```

FindByNameHelper 函数也并不复杂，首先查表，如果miss，就会调用 TryFindSymbolInFallbackDatabase 进行索引构建（这里会用到之前讲过的 DescriptorIndex 的信息）。源代码如下：

> 这里刻意略过 underlay，underlay 这个特性笔者猜测是为了效率实现的多层cache，underlay 也就是下层的意思，逻辑都是一样的，这里我们没有涉及 underlay，就先不展开分析。

```
Symbol DescriptorPool::Tables::FindByNameHelper(const DescriptorPool* pool,
                                                StringPiece name) {
  if (pool->mutex_ != nullptr) {
```

```
      if (!result.IsNull()) return result;
    }
  }
  MutexLockMaybe lock(pool->mutex_);
  if (pool->fallback_database_ != nullptr) {
    known_bad_symbols_.clear();
    known_bad_files_.clear();
  }
  Symbol result = FindSymbol(name);

  if (result.IsNull() && pool->underlay_ != nullptr) {
    // Symbol not found; check the underlay.
    result = pool->underlay_->tables_->FindByNameHelper(pool->underlay_, name);
  }

  if (result.IsNull()) {
    // Symbol still not found, so check fallback database.
    if (pool->TryFindSymbolInFallbackDatabase(name)) {
      result = FindSymbol(name);
    }
  }

  return result;
}
```

分析 FindSymbol，发现其查的是 symbols_by_name_ 这个索引表（其定义如下），但是这个表我们还没有构建啊。是的，之前没有构建过，但是为什么需要等待这个时候才构建呢？笔者认为有两个原因：一个是内存占用原因，如果没有改proto文件没有被使用到，就不需要前置构建，占用内存；另一个是启动效率原因，没有必要为了没有被使用到的proto文件做无用功，而且就算后续使用到了再构建，也只是第一个使用者会牺牲一些效率（如果读者有认为是其他什么原因导致这样设计，欢迎来交流和探讨）。

```
typedef HASH_MAP<StringPiece, Symbol, HASH_FXN<StringPiece>> SymbolsByNameMap;
class DescriptorPool::Tables {
...
  SymbolsByNameMap symbols_by_name_;
}
```

最终会使用 DescriptorBuilder来进行 symbol 索引的构建：TryFindSymbolInFallbackDatabase -> BuildFileFromDatabase -> DescriptorBuilder().BuildFile(proto)，BuildFile -> BuildFileImpl，BuildFileImpl 如下：

```
FileDescriptor* BuildFileImpl(const FileDescriptorProto& proto) {
  ...
  BUILD_ARRAY(proto, result, message_type, BuildMessage, nullptr);
  BUILD_ARRAY(proto, result, enum_type, BuildEnum, nullptr);
  BUILD_ARRAY(proto, result, service, BuildService, nullptr);
  BUILD_ARRAY(proto, result, extension, BuildExtension, nullptr);
  ...
}
```

BuildFileImpl 会针对每个 message_type、enum_type、service、extension 构建索引，举一个 BuildMessage 的例子。

```
void BuildMessage(const DescriptorProto& proto,
                  const Descriptor* parent,
                  Descriptor* result) {
  ...
  BUILD_ARRAY(proto, result, oneof_decl, BuildOneof, result);
  BUILD_ARRAY(proto, result, field, BuildField, result);
```

```
  BUILD_ARRAY(proto, result, reserved_range, BuildReservedRange, result);
  ...
  AddSymbol(...);
)
```

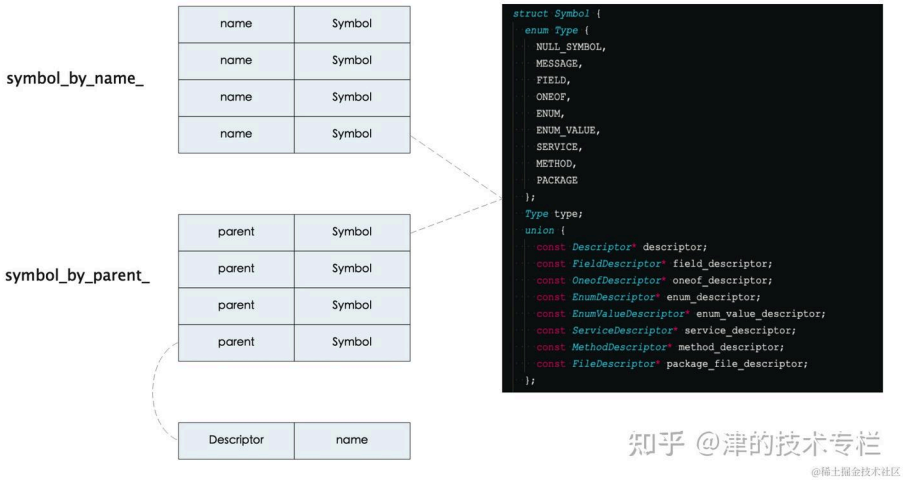
再看 AddSymbol 函数，从其代码可以看出会写两个表：一个是 symbols_by_name_ ，另一个是 symbols_by_parernt_ ，前者是通过命名来查找，后者是通过父类型调用触发的查找。

```
bool DescriptorBuilder::AddSymbol(const std::string& full_name,
                                  const void* parent, const std::string& name,
                                  const Message& proto, Symbol symbol) {
  ...
  if (tables_->AddSymbol(full_name, symbol)) {
    if (!file_tables_->AddAliasUnderParent(parent, name, symbol)) {
  ..
}
```

这两个表分布在不同的地方，symbols_by_name_ 是 DescriptorPool::Tables 类中，一般是全局搜索某个类型需要调用到，而symbols_by_parent_ 是在 FileDescriptorTables 类中，一般我们用来查询当前类型的某个字段（即field）用到比较多。

```
typedef HASH_MAP<StringPiece, Symbol, HASH_FXN<StringPiece>> SymbolsByNameMap;
class DescriptorPool::Tables {
...
  SymbolsByNameMap symbols_by_name_;
}

class FileDescriptorTables {
...
  SymbolsByParentMap symbols_by_parent_;
}
```

Symbol 是个抽象的类型，可以表示proto文件中的所有类型。所以，symbols_by_name_ 和 symbols_by_parent_ 这两个表也用来存储所有的类型。



4.2 MessageFactory 索引

4.2.1 google::protobuf::Message

这是所有Message 类型的基类，所以用他来表示索引的类型。

和DescriptorPool索引的构建时机相同，程序启动的时候构建了一部分索引，而在使用（也就是查询的时候）还会触发构建完整的Message索引数据。

4.2.3 MessageFactory 索引的构建原理

还是从AddDescriptorsImpl函数接口开始。这个函数是在程序启动的时候触发执行的。

```
void AddDescriptorsImpl(const DescriptorTable* table) {
  ...
  MessageFactory::InternalRegisterGeneratedFile(table);
}
void MessageFactory::InternalRegisterGeneratedFile(
    const google::protobuf::internal::DescriptorTable* table) {
  GeneratedMessageFactory::singleton()->RegisterFile(table);
}
```

GeneratedMessageFactory 类的定义如下，我们主要关注两个成员 file_map_ 和 type_map_ ，但实际上最有用的是type_map_ ， file_map_ 只是辅助作用。那为什么这里只是构建了type_map_ 呢？笔者认为和 DescriptorPool 索引的原因是一样的，一个是内存占用原因，另一个是启动效率原因（如果有认为是其他什么原因导致这样设计，欢迎探讨）。

```
class GeneratedMessageFactory final : public MessageFactory {
 public:
  //构建 file_map_
  void RegisterFile(const google::protobuf::internal::DescriptorTable* table);
  //构建 type_map_
  void RegisterType(const Descriptor* descriptor, const Message* prototype);
  const Message* GetPrototype(const Descriptor* type) override;

 private:
  // Only written at static init time, so does not require locking.
  HASH_MAP<StringPiece, const google::protobuf::internal::DescriptorTable*,
           STR_HASH_FXN> file_map_;
  ...
  std::unordered_map<const Descriptor*, const Message*> type_map_;
};
```

4.2.4 MessageFactory 索引的查询过程

从开发者怎么使用说起吧。开发者一般是调用 GetPrototype 函数来获取Messgae 实例。

```
  const google::protobuf::Message* prototype
    = google::protobuf::MessageFactory::generated_factory()
      ->GetPrototype(descriptor);
```

GetPrototype 函数的逻辑也很简单，先查 type_map_ 表，如果查不到，再根据 file_map_ 中的信息构建 type_map_ 表索引（见 internal::RegisterFileLevelMetadata 函数）。

```
const Message* GeneratedMessageFactory::GetPrototype(const Descriptor* type) {
  {
    /* 如果是第一次查询，那这里的查询结果是 Miss */
    ReaderMutexLock lock(&mutex_);
    const Message* result = FindPtrOrNull(type_map_, type);
    if (result != NULL) return result;
  }
  ...
  // Apparently the file hasn't been registered yet.  Let's do that now.
  const internal::DescriptorTable* registration_data =
```

```
  /* 如果查询结果为 Miss
   * 那么 需要调用 internal::RegisterFileLevelMetadata 构建 type_map_ 索引 */

  // Check if another thread preempted us.
  const Message* result = FindPtrOrNull(type_map_, type);
  if (result == NULL) {
    // Nope.  OK, register everything.
    internal::RegisterFileLevelMetadata(registration_data);
    // Should be here now.
    result = FindPtrOrNull(type_map_, type);
  }
  ...
  return result;
}
```

RegisterFileLevelMetadata 函数的一系列实现如下：

```
void RegisterFileLevelMetadata(const DescriptorTable* table) {
  AssignDescriptors(table);
  RegisterAllTypesInternal(table->file_level_metadata, table->num_messages);
}
void RegisterAllTypesInternal(const Metadata* file_level_metadata, int size) {
  for (int i = 0; i < size; i++) {
    const Reflection* reflection = file_level_metadata[i].reflection;
    MessageFactory::InternalRegisterGeneratedMessage(
        file_level_metadata[i].descriptor,
        reflection->schema_.default_instance_);
  }
}
void MessageFactory::InternalRegisterGeneratedMessage(
    const Descriptor* descriptor, const Message* prototype) {
  GeneratedMessageFactory::singleton()->RegisterType(descriptor, prototype);
}
void GeneratedMessageFactory::RegisterType(const Descriptor* descriptor,
                                           const Message* prototype) {
  ...
  if (!InsertIfNotPresent(&type_map_, descriptor, prototype)) {
    GOOGLE_LOG(DFATAL) << "Type is already registered: " << descriptor->full_name();
  }
}
```

最终是把 prototype（也就是 reflection->schema.*default_instance* ）插入 type_map_ 表中，我们回到 echo.pb.cc 源代码文件，见以下源代码：

```
struct EchoRequestDefaultTypeInternal {
  constexpr EchoRequestDefaultTypeInternal()
    : _instance(::PROTOBUF_NAMESPACE_ID::internal::ConstantInitialized{}) {}
  ~EchoRequestDefaultTypeInternal() {}
  union {
    EchoRequest _instance;
  };
};
PROTOBUF_ATTRIBUTE_NO_DESTROY PROTOBUF_CONSTINIT EchoRequestDefaultTypeInternal _EchoR
```

default_instance_ 指向的就是 *instance。因为Message 都实现了 New 函数，可以通过 default_instance*->New()创建出 Message 实例，即使不知道其真实类型是 EchoRequest。

```
inline EchoRequest* New() const final {
  return new EchoRequest();
}
```

通过 New 函数接口实现，实际上调用的 EchoRequest 的 New 函数，返回值为 EchoRequest *，而 EchoRequest 继承了 google::protobuf::Message 类。

```
google::protobuf::Message* req_msg = prototype->New();
```

4.4 Reflection 成员

还是以 EchoRequest 为例子。

```
message EchoRequest {
  QueryType querytype = 1;
  string payload = 2;
}
```

如果需要对 payload 字段读写，那我们直接使用 set_payload 和 get_payload 这两个函数接口就可以了。但是如果是使用 google::protobuf::Message 基类指针类型来操作，它是没有 set_payload 和 get_payload 这两个接口函数的。这个时候 Reflection （即 google::protobuf::Reflection）出现了，它类似一个代理人的角色，可以帮忙做一些读写的操作。如下 SetString、GetString 函数。Reflection 类过于庞大，这里就不详细分析，感兴趣的读者可以自行阅读源代码。

```
class PROTOBUF_EXPORT Reflection final {
 public:
    ...
    void SetString(Message* message, const FieldDescriptor* field,
                   std::string value) const;
    std::string GetString(const Message& message,
                          const FieldDescriptor* field) const;
    ...
 };
```

4.5 字段索引（Field）

前面「4.1.4 DescriptorPool索引的查询过程」章节中介绍了构建symbol 索引的过程，字段（即 field）索引也是在那个时候解析并构建的。如下使用到了 BuildField 函数进行字段索引构建。

```
void BuildField(const FieldDescriptorProto& proto, Descriptor* parent,
                FieldDescriptor* result) {
  BuildFieldOrExtension(proto, parent, result, false);
}

void BuildFieldOrExtension(const FieldDescriptorProto& proto,
                           Descriptor* parent, FieldDescriptor* result,
                           bool is_extension);
  ...
   AddSymbol(result->full_name(), parent, result->name(), proto, Symbol(result));
}
```

protobuf 中使用FieldDescriptor来描述字段（field），如下所示：

```
class PROTOBUF_EXPORT FieldDescriptor {
 public:
  typedef FieldDescriptorProto Proto;

  // Identifies a field type.  0 is reserved for errors.  The order is weird
  // for historical reasons.  Types 12 and up are new in proto2.
```

```
                        // take 10 bytes.  Use TYPE_SINT64 if negative
                        // values are likely.
  TYPE_UINT64 = 4,    // uint64, varint on the wire.
  TYPE_INT32 = 5,     // int32, varint on the wire.  Negative numbers
                        // take 10 bytes.  Use TYPE_SINT32 if negative
                        // values are likely.
  TYPE_FIXED64 = 6,   // uint64, exactly eight bytes on the wire.
  TYPE_FIXED32 = 7,   // uint32, exactly four bytes on the wire.
  TYPE_BOOL = 8,      // bool, varint on the wire.
  TYPE_STRING = 9,    // UTF-8 text.
  TYPE_GROUP = 10,    // Tag-delimited message.  Deprecated.
  TYPE_MESSAGE = 11,  // Length-delimited message.

  TYPE_BYTES = 12,     // Arbitrary byte array.
  TYPE_UINT32 = 13,    // uint32, varint on the wire
  TYPE_ENUM = 14,      // Enum, varint on the wire
  TYPE_SFIXED32 = 15,  // int32, exactly four bytes on the wire
  TYPE_SFIXED64 = 16,  // int64, exactly eight bytes on the wire
  TYPE_SINT32 = 17,    // int32, ZigZag-encoded varint on the wire
  TYPE_SINT64 = 18,    // int64, ZigZag-encoded varint on the wire

  MAX_TYPE = 18,  // Constant useful for defining lookup tables
                  // indexed by Type.
};

// Specifies the C++ data type used to represent the field.  There is a
// fixed mapping from Type to CppType where each Type maps to exactly one
// CppType.  0 is reserved for errors.
enum CppType {
  CPPTYPE_INT32 = 1,     // TYPE_INT32, TYPE_SINT32, TYPE_SFIXED32
  CPPTYPE_INT64 = 2,     // TYPE_INT64, TYPE_SINT64, TYPE_SFIXED64
  CPPTYPE_UINT32 = 3,    // TYPE_UINT32, TYPE_FIXED32
  CPPTYPE_UINT64 = 4,    // TYPE_UINT64, TYPE_FIXED64
  CPPTYPE_DOUBLE = 5,    // TYPE_DOUBLE
  CPPTYPE_FLOAT = 6,     // TYPE_FLOAT
  CPPTYPE_BOOL = 7,      // TYPE_BOOL
  CPPTYPE_ENUM = 8,      // TYPE_ENUM
  CPPTYPE_STRING = 9,    // TYPE_STRING, TYPE_BYTES
  CPPTYPE_MESSAGE = 10,  // TYPE_MESSAGE, TYPE_GROUP

  MAX_CPPTYPE = 10,  // Constant useful for defining lookup tables
                     // indexed by CppType.
};

// Identifies whether the field is optional, required, or repeated.  0 is
// reserved for errors.
enum Label {
  LABEL_OPTIONAL = 1,  // optional
  LABEL_REQUIRED = 2,  // required
  LABEL_REPEATED = 3,  // repeated

  MAX_LABEL = 3,  // Constant useful for defining lookup tables
                  // indexed by Label.
};
...
//因为一个field 只有一个类型，
//所以使用内联结构，节省内存，
union {
  int32 default_value_int32_;
  int64 default_value_int64_;
  uint32 default_value_uint32_;
  uint64 default_value_uint64_;
  float default_value_float_;
  double default_value_double_;
```

```
    mutable std::atomic<const Message*> default_generated_instance_;
  };
  ...
};
```

4.6 利用反射访问字段的API

结合 Reflection 来分析一下 field 的使用。看 field->default_value_string() 这一行，其实是返回了上述 union 中的 default_value_string_ 成员。

4.6.1 Reflection::GetString

```
std::string Reflection::GetString(const Message& message,
                                  const FieldDescriptor* field) const {
  USAGE_CHECK_ALL(GetString, SINGULAR, STRING);
  if (field->is_extension()) {
    return GetExtensionSet(message).GetString(field->number(),
                                              field->default_value_string());
  } else {
    if (schema_.InRealOneof(field) && !HasOneofField(message, field)) {
      return field->default_value_string();
    }
    switch (field->options().ctype()) {
      default:  // TODO(kenton):  Support other string reps.
      case FieldOptions::STRING: {
        if (auto* value =
                GetField<ArenaStringPtr>(message, field).GetPointer()) {
          return *value;
        }
        return field->default_value_string();
      }
    }
  }
}
```

4.6.2 Reflection::SetString

```
void Reflection::SetString(Message* message, const FieldDescriptor* field,
                           std::string value) const {
  USAGE_CHECK_ALL(SetString, SINGULAR, STRING);
  if (field->is_extension()) {
    return MutableExtensionSet(message)->SetString(
        field->number(), field->type(), std::move(value), field);
  } else {
    switch (field->options().ctype()) {
      default:  // TODO(kenton):  Support other string reps.
      case FieldOptions::STRING: {
        // Oneof string fields are never set as a default instance.
        // We just need to pass some arbitrary default string to make it work.
        // This allows us to not have the real default accessible from
        // reflection.
        const std::string* default_ptr =
            schema_.InRealOneof(field)
                ? nullptr
                : DefaultRaw<ArenaStringPtr>(field).GetPointer();
        if (schema_.InRealOneof(field) && !HasOneofField(*message, field)) {
          ClearOneof(message, field->containing_oneof());
          MutableField<ArenaStringPtr>(message, field)
              ->UnsafeSetDefault(default_ptr);
        }
```

```
                break;
            }
        }
    }
}
```

5 小结一下

- 程序启动时会先初始化一部分索引，这是轻量级的，只是一些基础的数据，为下一步构建全量索引做准备。
  - DescriptorTool 的 EncodedEntry、FileEntry、SymbolEntry、ExtensionEntry；
  - MessageFactory的 name->DescriptorTable。
- 当使用到某个类型的时候，会触发对该proto文件的全量索引的构建；
  - DescriptorTool 的 name-> Symbol、parent -> Symbol；
  - MessageFactory 的 Descriptor->Message。
- Reflection 作为 Message 的一个代理人，结合 Descriptor 和 Message的接口，对 Message 进行读写操作。
- 反射的一般使用场景：
  - 和其他数据结构比如 json、xml 等的相互转换；
  - 推荐系统中的特征抽取（平台化，所以需要对数据类型进行可配置化）。

编辑于 2023-11-05 03:24 · IP 属地广东

内容所属专栏

protobuf 反射（编程语言）

推荐阅读

ABAQUS笔记分享——接触分析中收敛问题的解决方法

Relax... 发表于有限元分析...

利用DFT结果拟合紧束缚模型参数并计算e-p coulping（1）

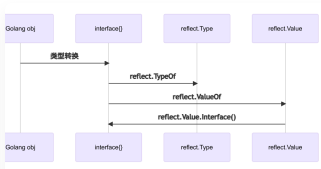
最近在学习 如何以DFT第一性原理计算出的高对称路径上的能量本征值为数据源，进行晶体TB模型参数的拟合，为有效学习，记录之。 本文以Marzari发表在RMP上的文章： Maximally localized Wann...

科研泡泡水

一文读懂运放共模抑制比（下）

之前说过，这个输出其实有两个分量组成，即共模电压输出和差模电压输出。我们这里只关心共模电压输出这部分。 经过变换，我们得到第一个公式。 1)第一个公式很明显，就是输出经过差分增益返...

54工程师



Golang反射原理详解

xcros... 发表于Go语言学...