深入 ProtoBuf - 反射原理解析

抽奖

在介绍了 ProtoBuf 序列化原理之后，本文介绍 ProtoBuf 的反射技术原理。

反射技术简介

对于反射大家应该不会陌生，如果你接触过一些框架（如 ORM、IOC、OSGi 等） 的内部实现，应该更能体会反射技术的应用可谓无处不在。

笔者读研期间所在实验室的核心技术并与反射息息相关，在反射基础之上衍生出了许多非常有趣的应用。关于反射可讲的东西很多，后续想单独写一篇文章介绍，这里先只做一个简介。

反射概念最早出现于人工智能领域，20 世纪 70 年代末被引入到程序语言设计中。1982 年 MIT 的 Smith, Brian Cantwell 在他的博士论文中最早提出了程序反射的概念：

> 既然我们可以构造“有关某个外部世界表示”的计算过程，并通过它来对那个外部世界进行推理；那么我们也可以构造能够对自身表示和计算进行推理的计算过程 ，它包含负责管理有关自身的操作和结构表示的内部过程。
>
> — 1982年 Smith, Brian Cantwell 博士论文首次提出

从某种角度来看，所谓编程实际上就是在构造 “关于外部世界” 的计算过程。如果用 F 表示这个构造过程，用 X 表示外部世界，那么编写一个计算系统可表示为 F(X)。

那么非常有趣的点就在于：**我们完全可以构造对上述过程本身进行描述和推理的计算过程。即将 F(X) 视为新的 “世界” 和 研究对象，构造 F(F(X))。**

我们平时编写的计算系统是面向特定领域的（通常是面向现实建模），系统中包含 用来描述领域中的实体 和 实体间关系 的数据结构以及处理这些数据结构的规则。那么反射系统面向领域便是这个系统本身。

从上层概念往下走，很容易就能理解反射将为我们提供这样的能力：

> 计算机程序在运行时可以访问、检测和修改它本身状态或行为
> — 反射（计算机科学）Wikipedia

很多编程语言可对程序本身进行构建（如何构建见下文），从而为程序员提供反射能力，即可在运行时访问和修改自身状态和行为：

```
// 运行时动态创建对象
Class<?> class1 = Class.forName("cn::lcy::dog");
Dog dogObj = (Dog) class1.newInstance();
// 运行时动态访问状态
Field dogNameField = class1.getDeclaredField("name");
// 运行时动态修改状态
dogNameField.set(dogObj, "鸡你太美");
```

```
// 运行时动态修改行为
Method method = class1.getMethod("signJumpRapAndBall");
method.invoke(dogObj);
```

例子中写的是字符串常量形式如 `"cn::lcy::dog"` ，但实际上 `"cn::lcy::dog"` 等可以是一个动态变化的变量，它可以由程序计算生成，可以从本地文件中读取，也可以通过网络从远程获取，这便是运行时动态的含义。

站在使用者的角度，通过上面例子应该可以对反射有一定的直观感受。

而站在实现者的角度，编程语言是如何实现程序反射的呢？

如果用一句话来总结反射实现的关键，笔者会概括为：**获取系统元信息**

> 元信息：即系统自描述信息，用于描述系统本身。举例来讲，即系统有哪些类？类中有哪些字段、哪些方法？字段属于什么类型、方法又有怎样的参数和返回值？....

对于上述例子的 Java 语言而言，其能够提供反射能力的关键是在编译阶段将程序的元信息编译进了 `.class` 文件，在程序运行时 JVM 将会把 `.class` 文件加载到 JVM 内存模型中的方法区。此后程序运行时将有能力获取关于自身的元信息。除了 Java 语言之外，JS、Python、GO、PHP 等各种语言也都在语言层面实现了程序反射。

而由于 C++ 在编译时并不会将类的元信息写进结果中，最终编译结果中只会包含变量、函数地址偏移、函数关系等，所以 C++ 自身无法获取元信息，那么自然无法提供反射能力。但这并不意味着使用 C++ 就无法应用反射技术，程序可以自己设计与实现程序元信息以及元信息与地址之间的映射。例如本文所要介绍的 ProtoBuf 反射实现。

ProtoBuf 反射技术简介

与其它语言提供的反射类似，ProtoBuf 能够为使用者提供了如下的反射能力：

```
/* 反射创建实例 */
auto descriptor = google::protobuf::DescriptorPool::generated_pool()->FindMessageTypeByName("[
auto prototype = google::protobuf::MessageFactory::generated_factory()->GetPrototype(descriptc
auto instance = prototype->New();

/* 反射相关接口 */
auto reflecter = instance.GetReflection();
auto field = descriptor->FindFieldByName("name");
reflecter->SetString(&instance, field, "鸡你太美") ;

// 获取属性的值.
std::cout<<reflecter->GetString(instance , field)<< std::endl ;
return 0 ;
```

利用上述 ProtoBuf 的反射能力，我们将能够实现许多强大的功能。例如各种 pb2json 库底层多是利用 ProtoBuf 的反射能力，实际上ProtoBuf 自身对编码结果反序列化并构建内存对象的过程用的也正是反射。

正如上一节提到的，反射的核心要点是：**获取程序元信息。**

ProtoBuf 自然也不会例外，那么 ProtoBuf 反射所需的元信息在哪？答案便是使用 ProtoBuf 的第一步就会接触到的：**.proto 文件。**

ProtoBuf 反射原理概述

我们在 深入 ProtoBuf - 简介 一文中介绍过使用 ProtoBuf 的第一步便是创建 .proto 文件，定义我们所需的数据结构。但很多人没有意识到，这个过程同时也是为 ProtoBuf 提供我们数据元信息的过程，这些元信息包括数据由哪些字段构成，字段又属于什么类型以及字段之间的组合关系等。

当然元信息也并非一定由 .proto 文件提供，它也可来自于网络或其它可能的输入，只要它满足 ProtoBuf Message 的定义语法即可。那么元信息的可能来源和处理就有：

- .proto 文件
    - 使用 ProtoBuf 内置的工具 protoc 编译器编译，protoc 将 .proto 文件内容编码并写入生成的代码中（.pb.cc 文件）
    - 使用 ProtoBuf 提供的编译 API 在运行时手动（指编码）解析 .proto 文件内容。实际上 protoc 底层调用的也正是这个编译 API。
- 非 .proto 文件
    - 从远程读取，如将数据与数据元信息一同进行 protobuf 编码并传输：
      ```
      message Req {
        optional string proto_file = 1;
        optional string data = 2;
      }
      ```
    - 从 Json 或其它格式数据中转换而来
    - ......

无论 .proto 文件来源于何处，我们都需要对其做进一步的处理，将其解析成内存对象，并构建其与实例的映射，同时也要计算每个字段的内存偏移。可总结出如下步骤：

1. 提供 .proto （范指 ProtoBuf Message 语法描述的元信息）
2. 解析 .proto 构建 FileDescriptor、FieldDescriptor 等，即 .proto 对应的内存模型（对象）
3. 之后每创建一个实例，就将其存到相应的实例池中
4. 将 Descriptor 和 instance 的映射维护到表中备查
5. 通过 Descriptor 可查到相应的 instance，又由于了解 instance 中字段类型（FieldDescriptor），所以知道字段的内存偏移，那么就可以访问或修改字段的值

ProtoBuf 反射源码解析

反射源码有许多阅读路径，在上一节中列出了元信息的不同来源，不同来源元信息就会导致处理上有所不同。我们没有必要解读源码中所有的实现路径，这里只选择最直观也是我们接触最多的一种路径。

重看 ProtoBuf 反射技术简介 中的例子，可对这个例子一步步深入从而理解反射实现的原理（代码 4-1）：

```
/* 反射创建实例 */
auto descriptor = google::protobuf::DescriptorPool::generated_pool()->FindMessageTypeByName("[
auto prototype = google::protobuf::MessageFactory::generated_factory()->GetPrototype(descriptc
auto instance = prototype->New();

/* 反射相关接口 */
auto reflecter = instance.GetReflection();
auto field = descriptor->FindFieldByName("name");
```

```
reflecter->SetString(&instance, field, "鸡你太美") ;

// 获取属性的值.
std::cout<<reflecter->GetString(instance , field)<< std::endl ;
return 0 ;
```

一、

代码 4-1 第一步我们通过 DescriptorPool 的 FindMessageTypeByName 获得了元信息 Descriptor。

DescriptorPool 为元信息池，对外提供了诸如 FindServiceByName、FindMessageTypeByName 等各类接口以便外部查询所需的元信息。当 DescriptorPool 不存在时需要查询的元信息时，将进一步到 DescriptorDatabase 中去查找。

DescriptorDatabase 可从硬编码或磁盘中查询对应名称的 .proto 文件内容，解析后返回查询需要的元信息。

DescriptorPool 相当于缓存了文件的 Descriptor（底层使用 Map），查询时将先到缓存中查询，如果未能找到再进一步到 DB 中（即 DescriptorDatabase）查询，此时可能需要从磁盘中读取文件内容，然后再解析成 Descriptor 返回，这里需要消耗一定的时间。

从上面的描述不难看出，DescriptorPool 和 DescriptorDatabase 通过缓存机制提高了反射运行效率，但这只是反射工程实现上的一种优化，我们更感兴趣的应该是 Descriptor 的来源。

DescriptorDatabase 从磁盘中读取 .proto 内容并解析成 Descriptor 这一来源很容易理解，但我们大多数时候并不会采用这种方式，反射时也不会去读取 .proto 文件。那么我们的 .proto 内容在哪?

实际上我们在使用 protoc 生成 xxx.pb.cc 和 xxx.pb.h 文件时，其中不仅仅包含了读写数据的接口，还包含了 .proto 文件内容。阅读任意一个 xxx.pb.cc 的内容，你可以看到如下类似代码（代码 4-2）：

```
static void AddDescriptorsImpl() {
  InitDefaults();

  // .proto 内容
  static const char descriptor[] GOOGLE_PROTOBUF_ATTRIBUTE_SECTION_VARIABLE(protodesc_cold) =
      "\n\022single_int32.proto\"\035\n\010Example1\022\021\n\010int3"
      "2Val\030\232\005 \001(\005\" \n\010Example2\022\024\n\010int32Val\030\377\377\377\377"
      "\001 \003(\005b\006proto3"
  };

  // 注册 descriptor
  ::google::protobuf::DescriptorPool::InternalAddGeneratedFile(
      descriptor, 93);

  // 注册 instance
  ::google::protobuf::MessageFactory::InternalRegisterGeneratedFile(
    "single_int32.proto", &protobuf_RegisterTypes);
}
```

其中 descriptor 数组存储的便是 .proto 内容。这里当然不是简单的存储原始文本字符串，而是经过了 **SerializeToString** 序列化处理，而后将结果以硬编码的形式保存在 xxx.pb.cc 中，真是充分利用了自己的高效编码能力。

硬编码的 .proto 元信息内容将以懒加载的方式（被调用时才触发）被 DescriptorDatabase 加载、解析，并缓存到

DescriptorPool 中。

二、

代码 4-1 例子中的第二步是根据 MessageFactory 获得了一个实例。

MessageFactory 是实例工厂，对外提供了根据元信息 descriptor 获取相应实例的能力。

其实在代码 4-2 中已经涉及到该工厂，即：

```
// 注册对应 descriptor 的 instance 到 MessageFactory
// InternalRegisterGeneratedFile 函数内部，会将创建一个实例并做好 descriptor 与 instance 的映射
::google::protobuf::MessageFactory::InternalRegisterGeneratedFile(
    "single_int32.proto", &protobuf_RegisterTypes);
```

每次构建实例后，都将 descriptor 和 instance 维护到一个 _table 中，即映射表以便获取。后续所谓通过反射获得某个类的某个实例子，实际就是查表的过程。

三、

代码 4-1 例子中的第三步，就是对 instance 实例对象的属性进行读写。

实例对象的 reflection 里面存储了对象属性的偏移地址，而这些信息其实与 .proto 内容信息一样，在 protoc 编译时通过解析 proto 文件内容获得且记录在 xxx.pb.cc 中，阅读 xxx.pb.cc 代码，可看到如下类似代码：

```
const ::google::protobuf::uint32 TableStruct::offsets[] GOOGLE_PROTOBUF_ATTRIBUTE_SECTION_VARI
  ~0u,  // no _has_bits_
  // 将会计算实例与属性的内存偏移
  GOOGLE_PROTOBUF_GENERATED_MESSAGE_FIELD_OFFSET(::Example1, _internal_metadata_),
  ~0u,  // no _extensions_
  ~0u,  // no _oneof_case_
  ~0u,  // no _weak_field_map_
  GOOGLE_PROTOBUF_GENERATED_MESSAGE_FIELD_OFFSET(::Example1, int32val_),
  ~0u,  // no _has_bits_
  GOOGLE_PROTOBUF_GENERATED_MESSAGE_FIELD_OFFSET(::Example2, _internal_metadata_),
  ~0u,  // no _extensions_
  ~0u,  // no _oneof_case_
  ~0u,  // no _weak_field_map_
  GOOGLE_PROTOBUF_GENERATED_MESSAGE_FIELD_OFFSET(::Example2, int32val_),
};
```

有了属性的内存偏移，自然可以对属性进行读写操作，以例子中出现的 SetString 为例，其内部实现位于 generated_message_reflection.cc 中，核心代码如下：

```
// 获取属性内存地址指针，内部根据 __
const std::string* default_ptr =
    &DefaultRaw<ArenaStringPtr>(field).Get();

// DefaultRaw 底层调用：
// reinterpret_cast<const uint8*>
// (default_instance_) +
//     OffsetValue(offsets_[field->index()], field->type());

// .......

// assign 赋值
MutableField<ArenaStringPtr>(message, field)
    ->Mutable(default_ptr, GetArena(message))
    ->assign(std::move(value));
```

其他 SetInt32、SetInt64、SetBool 等等接口原理类似。

ProtoBuf 的源码十分庞大，本节只是解析了其中一条路径，但已经足够呈现 ProtoBuf 的反射原理。如果想了解更多工程细节可进一步阅读 ProtoBuf 源码。

参考文献

Procedural reflection in programming languages
Google Developers. Protocol Buffers
一种自动反射消息类型的 Google Protobuf 网络传输方案
google protobuf 反射机制学习笔记

以上

汪

汪