Thrift:可扩展的跨语言服务实现

Mark Slee, Aditya Agarwal and Marc Kwiatkowski
Facebook, 156 University Ave, Palo Alto, CA
{mcslee,aditya,marc}@facebook.com

翻译: 王浩
中国 成都
docongwh@gmail.com

摘要

Thrift是一个最初由Facebook公司开发的软件库和代码产生工具集，它加速了高效和可扩展后端服务的开发和实现。它的主要目标是使跨编程语言的高效、可靠通信成为可能，通过抽象每种语言的特定部分,满足由各种语言实现的通用库趋于最大化定制的需求。尤其是，Thrift允许开发者在一个语言中立性文档中定义数据结构和服务，并产生构建RPC客户端和服务器端的所有必需代码。

本文详细的说明了我们编写Thrift时的动机和设计选择，也包括一些更加有趣的实现细节。我们并非故意把其作为一项研究，而是展示我们所作与所思。

1. 引言

随着Facebook的流量和网络结构的扩展，站点上的很多操作（比如搜索，选择和分发，事件日志记录）的资源要求已经表现为技术需求，并远远的超出了LAMP架构能够处理的范围。在这些服务的实现中，我们选择了多种不同的编程语言来达到满意的性能、简单快速的开发、已存在库的重用，等等。大体上说，Facebook的工程师文化倾向于选择可用的最好的工具和实现手段，而不是选择标准化任何单一的编程语言并意阿Q般地接受这种语言带来的固有限制。

考虑到这样的设计选择，我们面临的挑战是设计透明、高性能的跨语言“桥梁”。我们发现大多数可用的方案要不是限制太多，就是没有提供足够自由的数据类型，或者性能不能满足需要。[1]

我们已经实现的方案包含了一个跨多种语言的语言中立软件堆栈，而且还有一个相应的代码产生引擎来把简单的接口和数据定义语言转化成客户端和服务器端远程过程调用库。选择静态代码产生方式能够让我们创建可以运行的有效代码，而不必满足动态系统任何高级的自省运行时检查需求。这也为开发人员设计的尽可能的简单，开发人员能够照例为一个复杂服务定义所有必需的数据结构和接口，而且只需要使用一个单独、精简的文件。

由于惊讶于对这样一些相关问题的健壮而开放的解决方案还没有存在，我们在初期便开放了Thrift的源代码。

[1] 参看附录A 对替代系统的讨论.

为了评估在网络环境中的跨语言交互挑战，一些关键的构件做如下标识：

类型(Types)，一个通用类型系统必须跨语言存在，不需要要求应用开发人员使用Thrift数据类型或者写专属的序列化代码。也就是说，一个C++程序员应该能够透明地用一个强定义STL map与一个Python dictionary进行数据交换。不需要强迫程序员为了使用这个系统而在应用层之下写任何其它代码。第2章详细描述了Thrift的类型系统。

传输(Transport)，每种语言必须有通用的接口来进行双向原始数据传输。一个给定传输的细节是如何实现的应该与服务开发人员无关。相同的应用代码对于TCP流套接字，内存中的原始数据，或者磁盘文件都应能良好的运行。第3章详细描述了Thrift的传输层。

协议(Protocol)，数据类型必须有一些方法来使用传输层进行编解码。同样，应用开发人员无需关心这一层。不管服务使用XML或者二进制协议对于应用层代码是无关紧要的。这一切意味着只要数据在一个持久，确定的形式中，就可以对其进行读写。第4章详细描述了Thrift的协议层。

版本管理(Versioning)，对于健壮的服务，其中的数据类型必须提供一种机制来对其进行版本管理。尤其是它应该可以在不中断服务(或者，更坏的情况，出现段错误)的前提下，添加或删除一个对象中的字段，或者改变一个函数的参数列表。第5章详细描述了Thrift的版本管理系统。

处理器(Processors)，最终，我们产出能够处理数据流并完成远程过程调用的代码。第六章详细描述了产生的这些代码和处理器的示例。

第7章讨论了实现的细节，第8章进行总结。

2. 类型

Thrift类型系统的目标是能够让程序员完全使用原生定义的类型，而不管他们使用何种编程语言。通过设计，Thrift类型系统没有引入任何特殊的动态类型或者包装对象。它也不要求开发人员为对象序列化或传输写任何代码。Thrift的IDL（Interface Definition Language，接口定义语言）文件是一种逻辑上的方法，以最小的额外代价使开发者注解他们的数据结构，以此来告诉代码产生器如何安全的跨语言传输对象。

2.1 基本类型

类型系统依赖少量的基本类型。在考虑何种类型应该被支持时，我们的目标是简洁和简单而不是大而全，关注所有程序语言中都可用的关键类型，忽略只有特定语言可用的任何包装类型。

被Thrift支持的基本类型是：

- `bool` 表示一个布尔值，取true或false
- `byte` 表示一个带符号字节
- `i16` 表示一个带符号16位整型
- `i32` 表示一个带符号32位整形
- `i64` 表示一个带符号64位整形
- `double` 表示一个带符号64为浮点数
- `string` 表示一个不可知编码的文本或二进制串

值得注意的是没有无符号整形。因为这样的类型无法翻译映射到很多语言的原始类型，所以无法体现他们提供的优点。更进一步，没有办法阻止应用开发人员在诸如Python这样的语言中，把一个负值赋给一个整形变量，而导致不可预测的情况。从设计的立场，我们观察到无符号整形是非常少的，如果有，也是因为算术计算的目的，但是在实际项目中更多的是用作一个key值或标识值。在这种情况下，符号是无关的，有符号整形能够用于相同的目的，并且能够在必要的时候，安全的映射他们无符号的部分（在C++中很常见）。

2.2 结构

一个Thrift结构定义了一个通用的对象以此来跨语言。在面向对象语言中，一个结构本质上就是一个类。一个结构有一系列强定义字段，每个字段都有一个唯一的标识。定义Thrift结构的基本语法看起来类似于C语言的结构定义。这些字段可以被一个整形字段标识注解（结构作用域所独有）也可选使用默认值，字段标识如果忽略将会被自动分配，但是因为版本管理的原因，还是强烈推荐使用标识，这一点我们会在后面讨论。

```
struct Example {
  1:i32 number=10,
  2:i64 bigNumber,
  3:double decimals,
  4:string name="thrifty"
}
```

2.3 容器

Thrift容器是强类型容器，能够与常用语言中使用最通用的容器相对应。使用C++模板（或Java范型）的风格对其进行标注。在Thrift中，有三种可供使用的容器：

- `list<type>` 一个有序元素列表。直接翻译为一个STL vector，Java ArrayList，或者脚本语言的原生数组。可以包含重复元素。
- `set<type>` 一个无序不重复元素集。翻译为STL set，Java HashSet，Python set，或者PHP/Ruby中的原生dictionary。
- `map<type1,type2>` 一个主键唯一键值映射表。翻译为STL map，Java HashMap，PHP associative array，或者Python/Ruby的dictionary。

虽然提供默认容器，但是类型映射并不是明确固定的。已经添加自定义代码产生器指令集，用来替换目标语言中的自定义类型。(比如，hash_map或者谷歌的sparse hash map能够被用到c++中)。唯一的要求是自定义类型支持所有必需的迭代原语。容器元素可以是任意合法的Thrift类型，甚至包括其它容器或者结构。

在目标语言中，每种定义产生一个类型和两个方法：read和write，用来实现序列化和通过使用Thrift Tprotocol对象传输这些对象。

2.4 异常

异常在语法和功能上等价于结构，唯一的不同就是异常使用exception关键字而不是struct关键字声明。

被产生的对象继承自各种目标编程语言中的一个恰当的异常基类，这样便可以无缝地与任意给定语言的原生异常整合。同样，设计的重点是产生对应用开发人员友好的代码。

2.5 服务

服务通过Thrift类型进行定义。一个服务的定义在语义上相当于面向对象编程中定义一个接口（或者一个纯虚抽象类）。Thrift编译器通过实行这个接口产生功能完整的客户端和服务器桩。服务被定义如下：

```
service <name> {
  <returntype> <name>(<arguments>)
    [throws (<exceptions>)]
  ...
}
```

一个例子：

```
service StringCache {
  void set(1:i32 key, 2:string value),
  string get(1:i32 key) throws (1:KeyNotFound knf),
  void delete(1:i32 key)
}
```

注意void类型是除了所有已经被定义的Thrift类型外的一个合法的函数返回类型。如果void前面加上async关键字修饰，那么将产生不需要等待服务器响应的代码。注意一个纯void类型函数将给客户端返回一个响应，以此来保证服务器端的操作已经完成。通过使用async方法调用，客户端将只是保证请求被成功的放到了传输层。（在很多传输场景下，这种固有的不可靠性是由于拜占庭将军问题导致的。因此，应用开发人员应该只在如下情况下谨慎使用async：1、可以接受丢失方法调用；2、传输已知是非常可靠的。）

还有个需要注意的事实是，参数列表和异常列表对于函数来说都是以Thrift structs的方式实现的。所有这三种构造无论在标记还是行为上都是相同的。

3. 传输

传输层被产生的代码用来便利的传送数据。

3.1 接口

Thrift实现的一个重要的设计选择就是：将传输层从代码产生层中分离出来。尽管Thrift是典型的通过流套接字被使用在TCP/IP协议栈上，并以此作为基础通信层，但是没有强制性的原因要求把这样的限制附加于系统中。使用抽象I/O层所带来的性能损失（简单的说就是对于每个操作需要一个虚拟的方法查找/函数调用），与直接使用实际的I/O操作的代价相比是无关紧要的（典型的就是系统调用）。

从根本上讲，产生的Thrift代码只需要知道如何读写数据，而与数据的源和目的是无关的；它可以是一个套接字，一段共享内存，或者一个本地磁盘上的文件。Thrift传输接口支持如下方法：

- `open` 打开一个传输
- `close` 关闭一个传输
- `isOpen` 指示传输是否已经打开
- `read` 从一个传输中读
- `write` 向一个传输中写
- `flush` 强制任何挂起的写

还有一些其他方法没有在这里列出，那些方法用来帮助进行批量读和选择性地从产生的代码中发生一个读（或写）操作已经完成的的信号。

除了上面的TTransport接口之外，还有一个TServerTransport接口被用来接受或者创建基本的传输对象。接口如下：

- `open` 打开一个传输
- `listen` 对一个连接开始侦听
- `accept` 返回一个新的客户端传输对象
- `close` 关闭传输

3.2 实现

传输接口被设计得可以在任何编程语言中简单实现。新的传输机制可以被应用开发人员按需方便的定义。

3.2.1 TSocket

TSocket类跨所有目标语言被实现。它为TCP/IP流套接字提供了一个通用的、简单的接口。

3.2.2 TFileTransport

TFileTransport是一个磁盘文件数据流的抽象。它被用来将收到的一系列Thrift请求写到磁盘文件中。磁盘数据可以从日志中重现，可用来后继处理或复制（模拟）过去的事件

3.2.3 工具程序

传输接口设计方便地支持通用面向对象技术的扩展，比如对象的组合。有一些简单的工具程序包含在TBufferedTransport中，用来缓存一个潜在传输上的读写，TFramedTransport传输的数据，通过帧头有帧大小信息来优化分块和无阻塞操作，TMemeoryBuffer允许直接从进程拥有的堆栈内存进行读写。

4. 协议

Thrift的第二个重要抽象是把数据结构从传输表达中分离出来。Thrift在传输数据时，强制使用某种消息结构，但是在使用中，这些消息结构对于协议编码是不可知的。也就是说，不管数据以何种形式编码（XML编码，还是人工可直接读到ASCII编码，或者高密度的二进制编码）只要数据支持的固定操作集允许它能够被产生的代码明确地读写。

4.1 接口

Thrift协议接口是非常易懂的。它基本上支持两条原则：1）双向顺序消息，和2）基本类型、容器和结构的编码。

```
writeMessageBegin(name, type, seq)
writeMessageEnd()
writeStructBegin(name)
writeStructEnd()
writeFieldBegin(name, type, id)
writeFieldEnd()
writeFieldStop()
writeMapBegin(ktype, vtype, size)
writeMapEnd()
writeListBegin(etype, size)
writeListEnd()
writeSetBegin(etype, size)
writeSetEnd()
writeBool(bool)
writeByte(byte)
writeI16(i16)
writeI32(i32)
writeI64(i64)
writeDouble(double)
writeString(string)

name, type, seq = readMessageBegin()
                  readMessageEnd()
name =            readStructBegin()
                  readStructEnd()
name, type, id =  readFieldBegin()
                  readFieldEnd()
k, v, size =      readMapBegin()
                  readMapEnd()
etype, size =     readListBegin()
                  readListEnd()
```

```
etype, size =     readSetBegin()
                  readSetEnd()
bool =            readBool()
byte =            readByte()
i16 =             readI16()
i32 =             readI32()
i64 =             readI64()
double =          readDouble()
string =          readString()
```

注意除了writeFieldStop()外，每个write函数都有一个恰好的read函数相对应。writeFieldStop()是一个特殊的方法，用来发送结构结束信号。读一结构的过程是从readFiledBegin()开始直到遇到停止字段，然后到readStructEnd()。产生的代码依赖于这个调用序列，并以此确保所有内容已被协议编码器(protocol encoder)写入，这些被写入的内容可以通过相匹配的协议解码器(protocol decoder)读出。需要进一步明确的是，这组函数设计得比实际需要地更健壮。比如，writeStructEnd不是严格需要的，因为一个结构的结尾可能会有一个暗示停止的字段。这个方法对于冗长的协议提供了便利，在这些协议中，更加干净的分割这些调用（比如，在XML中¡/struct¿闭合标记）。

4.2 结构

Thrift结构被设计得可以编码到一个协议流中。协议实现在编码这个结构之前永远不需要编帧或者计算一个结构的整体数据长度。在很多场景下，这一点对性能事关重要。考虑一个拥有相对大字符串的长list。如果协议接口要求读写一个list是原子操作，那么实现需要花费线性时间复杂度来对任何数据编码前的list做全扫描。然而，如果list能被迭代写，并且相应地进行并行读，那么理论上可以对端到端提供一个kN-C的时间复杂度，N是list的大小，k是序列化一个单一元素的相关代价因子，C是对数据正在被写和正在变为可读之间延迟代价的固定抵偿。

类似，结构不会事先编码它们的数据长度。与此相反，它们作为字段序列被编码，每个字段有一个类型标识和一个唯一的字段识别标识。注意，类型标识的包括，确保了协议能够在不需要任何产生代码或对原始IDL文件的访问的情况下，被安全地分析和解码。结构通过在一个字段头使用特殊的STOP类型被判断结束。因为所有的基本类型都确定能被读，所有的结构（甚至那些嵌套结构）也能确定地被读，因此Thrift是自适应划界的，不需要任何编帧也不考虑编码的格式。

在那些不需要流或者需要进行编帧的场景，通过使用TFramedTransport抽象是很容易把它添加到传输层的。

4.3 实现

Facebook已经实现和部署了一个有效利用空间的二进制协议，并被大多数后端服务使用。在本质上，它以普通二进制格式写所有数据。整形被转换成网络字节顺序，字符串在串头加入它们的字节长度，所有的消息和字段头都使用原始整形序列化构造写入。字段的字符串名被删除，当使用产生代码时，字段标识已经足够。

我们没有采用一些极端的存储优化方案（比如把小整形打包到ASCII码中，或者使用7为拓展格式），原因是为了编码的简单和简洁。如果当我们遇到需要进行这种优化的用例需求，这些改变很容易实现。

5. 版本管理

Thrift在版本管理和数据定义的改变方面是健壮的。这对于部署的服务逐步升级是非常关键的。系统必须能够支持从日志文件中读出旧数据，也要求支持旧客户端向新服务器端发送请求，反之亦然。

5.1 字段标识符

版本管理在Thrift中是通过字段标识符来实现的。对于每个被Thrift编码的结构的域头，都有一个唯一的字段标识符。这个字段标识符和它的类型说明符构成了对这个字段独一无二地识别。Thrift定义语言支持字段标识符的自动分配，但是好的程序实践中是明确的指出字段标识符。字段标识符使用如下方式指定：

```
struct Example {
  1:i32 number=10,
  2:i64 bigNumber,
  3:double decimals,
  4:string name="thrifty"
}
```

为了避免人工和自动分配的标识符冲突，忽略了标识符的字段被自动从-1递减分配字段标识符，并且Thrift定义语言只支持人工分配正的标识符。

当数据正在被反序列化的时候，产生的代码能够用这些字段标识符来恰当地识别字段，并判断这个标识符是否在它的定义文件中和一个字段对齐。如果一个字段标识符不被识别，产生的代码可以用类型说明符去跳过这个不可知的字段而不产生任何错误。同样，这个也可以归结为这样一个事实：所有的数据类型都是自划界的。

字段标识符也能够（应该）在函数参数列表中被指定。事实上，参数列表在后端不仅作为结构被呈现，而是在编译器前段分享了相同的代码。这是为了允许安全的修改方法的参数。

```
service StringCache {
  void set(1:i32 key, 2:string value),
  string get(1:i32 key) throws (1:KeyNotFound knf),
  void delete(1:i32 key)
}
```

对于每个结构，都选择指定字段标识符的语法。结构可以看成一个字典，标识符是主键，强类型名的字段是值。

字段标识符内部使用Thrift的i16数据类型，然而，要注意的是，TProtocol抽象可能会以任何格式对标识符编码。

5.2 Isset

当遇到一个未期望的字段，能够被安全的忽略和丢弃。当一个预期的字段未被找到，必须有一些方法来向告知开发者该字段未出现。这是通过内部结构isset来实现的，这个结构位于已定义对象内部。(Isset的函数特性通过一个空值隐含，如PHP中的null，Python中的none，Ruby中的nil)本质上说，每个Thrift结构内部的isser对象为每个字段包含了一个布尔值，以此指示这个字段是否出现在结构中。当一个阅读器（reader）接受一个结构，它应该在直接操作它之前检查这个是否置位。

```
class Example {
 public:
  Example() :
    number(10),
    bigNumber(0),
    decimals(0),
    name("thrifty") {}

  int32_t number;
  int64_t bigNumber;
  double decimals;
  std::string name;

  struct __isset {
    __isset() :
      number(false),
      bigNumber(false),
      decimals(false),
      name(false) {}
    bool number;
    bool bigNumber;
    bool decimals;
    bool name;
  } __isset;
...
}
```

5.3 案例分析

版本不匹配可能发生在如下四种情况中：

1. 已添加字段，旧客户端，新服务器。在这种情况下，旧客户端没有发送新字段。新服务器识别到那个新字段未置位，执行对于旧数据请求的默认操作。
2. 已删除字段，旧客户端，新服务器。在这种情况下，旧客户端发送已被删除的字段。新服务器简单地忽略这个字段。
3. 已添加字段，新客户端，旧服务器。新客户端发送了那个旧服务器不能识别的一个字段，旧服务器简单的忽略，并按正常请求处理。
4. 已删除字段，新客户端，旧服务器。这是最危险的情况，对于丢失的字段，旧服务器不大可能有默认的合适动作执行。对于这种情况，推荐在升级客户端之前升级服务器端。

5.4 协议/传输版本管理

Tprotocol抽象也是设计得给予协议实现充分的自由，他们选择任何他们认为合适的方法进行版本管理。特别地，任何协议实现对于在writeMessageBegin()调用中发送他们喜欢的任何东西是自由的。这完全取决于实现程序在协议层如何处理版本管理。关键点是协议编码的改变要安全地隔离于接口定义版本的改变。

注意对于TTransport接口，上述规则也是完全的一致。比如，如果我们想向TFileTransport中添加一些新的校验和和错误检测，我们可以简单的添加一个版本头到它写的文件的数据中，通过这样的方法，它仍然可以接受没有给定头的旧的日志文件。

6. RPC 实现

6.1 TProcessor

在Thrift设计中，最后一个核心的接口就是TProcessor，它也许是有简单的构造函数。接口如下：

```
interface TProcessor {
  bool process(TProtocol in, TProtocol out)
    throws TException
}
```

关键的设计思想是我们构建的复杂系统从根本上可以分为代理和服务，并在输入和输出上执行操作。通常，只有一个输入和输出(一个RPC客户端)需要处理。

6.2 产生代码

当一个服务被定义，我们通过一些协助程序（通常是指实现接口服务的那些helper类，译者注），产生一个能够对这个服务处理RPC请求的TProcessor实例。基本结构（以C++伪代码为例）如下：

```
Service.thrift
 => Service.cpp
     interface ServiceIf
     class ServiceClient : virtual ServiceIf
       TProtocol in
       TProtocol out
     class ServiceProcessor : TProcessor
       ServiceIf handler

ServiceHandler.cpp
```

```
class ServiceHandler : virtual ServiceIf
```

```
TServer.cpp
TServer(TProcessor processor,
        TServerTransport transport,
        TTransportFactory tfactory,
        TProtocolFactory pfactory)
serve()
```

从Thrift定义文件，我们产生特定的虚拟服务接口。一个客户端类被产生，它实现了这个接口并且使用两个TProtocol实例来执行I/O操作。
产生的processor实现了TProcessor接口。产生的代码通过调用process（），拥有处理RPC调用的所有逻辑，并且使用service接口的实例（实例由应用开发人员实现）作为参数。

使用者在分隔的，非产生代码中提供应用接口的实现。

6.3 TServer

最终，Thrift核心库提供一个TServer抽象。这个TServer对象通常以如下方式运行：

- 使用TServerTransport得到一个TTransport
- 使用TTransportFactory有选择的把基本传输（对象）变成合适的应用传输（对象）（TBufferedTransportFactory就是典型地被用于这种情况）
- 使用TProtocolFactory为TTransport创建一个输入和输出协议
- 调用TProcessor对象的process()方法

这些层次被恰当的分离，比如服务器端代码在运行过程中不需要知道任何传输、编码或应用的信息。当处理器处理RPC时，服务器在连接处理、线程等周围，包装了这样的逻辑。Thrift定义文件和接口实现是应用开发人员唯一需要写代码的地方。

Facebook已经部署了多种TServer实现，包括单线程的TSimpleServer，每个连接一个线程的TThreadedServer，和线程池TThreadPoolServer。

TProcessor接口设计得是非常通用的。没有必要，让每个TServer都配备一个产生的TProcessor对象。Thrift允许应用开发人员方便地在TProtocol对象中写服务器端的任何类型（比如，一个服务器端可以简单地流化某种类型的对象，而不需要任何实际的RPC方法调用）。

7. 实现细节

7.1 目标语言

Thrift目前支持5种目标语言：C++，Java，Python，Ruby，和PHP。在Facebook，我们已经部署的服务器主要使用C++，Java和Python。在PHP中实现的Thrift服务也已经嵌入到了Apache web服务器中，用来让后端通过THttpClient透明地访问我们很多前端结构，THttpClient实现了TTransport的接口。

尽管Thrift被明确地设计得比典型的web技术更高效和健壮，但是随着我们设计一个基于XML的REST web服务API，我们注意到Thrift能够容易地用来定义我们的服务接口。尽管我们目前没有使用SOAP封套元素（在作者们的眼中，已经有过多重复的Java软件做这一堆事情），但是我们能够很快地为我们的服务扩展Thrift，使其产生XML概要定义文件，Thrift就像是一个为我们web服务的不同实现进行版本管理的框架。尽管公共web服务诚然于Thrift的核心用例和设计无关，但是Thrift能够便利地快速迭代，并使我们有能力快速地将整个基于XML的web服务迁往更高性能的系统。

7.2 产生的结构

我们自觉的将产生的结构尽可能的透明。所以的字段都是公共可访问的；没有set()和get()方法。类似，isset对象的使用不是强制性的。我们没有包含任何FieldNotSetException这样的结构。开发人员可以选择这些字段写出更健壮的程序，但是系统对于开发人员完全忽略isset结构也是健壮的，并且将对所有情况提供一个恰当的默认行为。

这种选择来源于对简便应用开发的渴望。我们陈述设计目标不是为了让开发人员在他们选择的语言中学习一种新的丰富的库，而是产生代码允许他们在每种语言中使用他们最熟悉的结构。

我们也把产生的对象的read()和write()方法放置在公共作用域中，以便这个对象能够被RPC客户端和服务器端的外部上下文使用。仅就能够产生跨语言容易序列化的对象这一点来说，Thrift就是一个有用的工具。

7.3 RPC方法识别

在RPC中，方法调用是通过发送该方法名的字符串来实现的。一个关于这种方法的议题是：更长的方法名意味着需要更大的带宽。我们实验过使用固定长度的hash值去识别方法名，但是最后发现节省的带宽与引起的问题相比是不划算的。如果不使用元数据存储系统，要可靠地处理接口定义文件的跨版本冲突是不可能的。（比如，为了给一个文件的当前版本产生无冲突的hash值，我们可能必须知道这个文件先前已经存在的所有版本的冲突）。

我们想避免在方法调用过程中过多没有必要的字符串比较。为了解决这个问题，我们产生了从字符串到函数指针的映射，以便在通常情况下，调用可以通过固定时间的hash查找高效地完成。这就要求使用一个二元代码结构。因为Java没有函数指针，处理函数是所有实现一个通用接口的私有成员函数类?。

```
private class ping implements ProcessFunction {
  public void process(int seqid,
                      TProtocol iprot,
                      TProtocol oprot)
    throws TException
```

```
  { ...}
}

HashMap<String,ProcessFunction> processMap_ =
  new HashMap<String,ProcessFunction>();
```

在C++中，我们使用了一个相对孤僻的语言结构：成员函数指针。

```
std::map<std::string,
  void (ExampleServiceProcessor::*)(int32_t,
  facebook::thrift::protocol::TProtocol*,
  facebook::thrift::protocol::TProtocol*)>
 processMap_;
```

通过这些技术，处理字符串的代价被最小化了，因为知道了字符串方法名，所以我们还得到了这样的好处：可以容易地通过检查方法名，调试崩溃或错误的数据。

7.4 服务器端和多线程

Thrift服务要求基本的多线程来处理来自多个客户端的同时请求。对于实现Thrift服务器逻辑的的Python和Java代码，对应语言的标准线程库就已经提供了足够的支持。对于C++的实现，没有标准的多线程运行时库存在。特别是没有健壮的、轻量级的和易用的线程与定时类实现。我们考察了存在的一些实现，即：boost::thread，boost::threadpool，ACE_Thread_Manager和ACE_Timer。

虽然boost::threads提供了简洁、轻量级和健壮的多线程原语实现(互斥、条件变量、线程)，但是它没有提供线程管理或定时器实现。

boost::threadpool也是看起来很有希望，但是离我们想要的还差很远。我们想尽可能的减少对第三方库的依赖，因为boost::threadpool不是一个标准的模板库，它要求运行时库并且它还不是Boost官方发行版的一部分，所以我们感觉它还不能用到Thrift中。随着boost::threadpool的发展，尤其是如果它被添加到了Boost的发行版，我们也许会重新考虑是不是要用它。

ACE除了提供多线程原语外，还有有一个线程管理类，也有定时器类。但是ACE最大的问题是它本身。和Boost不同，ACE API的质量是不够的。ACE中的任何东西都对ACE中的其它任何东西有大量的依赖，因此强迫开发人员扔掉标准类，比如STL容器，而选择ACE的专属实现。另外，不同于Boost，ACE的实现显示了较弱的可理解性和C++编程的一些缺陷，没有利用现代模板技术来确保编译时安全并把一些编译错误信息合理化。因为这些原因，ACE不被选择。作为替代，我们选择实现我们自己的库，并在接下来的几节中描述。

7.5 线程原语

Thrift线程库在facebook::thrift::concurrency名字空间下实现，并由三部分构成：

- 原语
- 线程池管理器
- 定时器管理器

如上所述，我们对于引入任何外部依赖到Thrift是犹豫不决的。尽管如此，我们还是决定使用boost::shared_ptr，因为它对于多线程应用是如此的有用，它不要求链接时或运行时库（即，他是一个纯模板库）并且它将成为C++0x标准的一部分。

我们实现了标准的互斥和条件类，还有一个监控类。后者是对一个互斥和条件变量的简单组合，类似于对Java Object类提供的Monitor的实现。这有时也被看成是一个障碍。我们提供了一个同步守护类，以便允许类似Java一样的同步块。这只是语法上的一点好处，但是，像它的Java对应部分一样，清楚地界定代码的临界区。和它的Java对应部分不一样的是，我们仍然可以在编程中使用lock，unlock，block和signal monitors。

```
void run() {
 {Synchronized s(manager->monitor);
  if (manager->state == TimerManager::STARTING) {
    manager->state = TimerManager::STARTED;
    manager->monitor.notifyAll();
  }
 }
}
```

我们再次借鉴了Java中线程和一个runnable类的区别。一个线程实际是一个可调度对象。Runnalbe是线程运行中的逻辑。线程实现处理的是所有平台下的线程创建和销毁问题，而Runnalbe实现处理的是每个线程逻辑下的特定应用。这种方法的好处是，开发人员可以容易地编写Runnable类的子类，而不需要纠缠于特定平台的超类中。

7.6 线程、Runnable和shared_ptr

我们在ThreadManager和TimerManager实现的整改过程都使用了boost::shared_ptr，以此来保证被多线程访问的死对象已经被清理干净。对于Thread类的实现，boost::shared_ptr的用法要求特别注意确保Thread对象们在创建或关闭线程时没有泄漏也没有被过早引用。

线程创建要求在C库中调用（在我们的例子中是POSIX线程库、libpthread，但是对于WIN32线程也可能是一样的）。典型地，操作系统在调用ThreadMain（C语言线程入口函数）时，几乎无法做到（时间上的）保证。因此，我们在线程创建时调用ThreadFactory::newThread()，可能在系统调用之前就给调用者返回了正常。如果调用者在ThreadMain调用之前放弃了引用，为了确保返回的Thread对象不会提前被过早清除，Thread对象在它的start方法中，对它自己做了一个弱引用。

如果持有弱引用，那么ThreadMain函数可以在进入绑定到Thread的Runnalbe对象的Runnalbe::run方法前，尝试获取一个强引用。如果对于这个线程没有强引用能够在退

出Thread::start与进入ThreadMain之间被获得，那么弱引用返回null，函数立即退出。

Thread对自身建立一个弱引用的需求在API层面影响深远。因为引用是通过boost::shared_ptr模板被管理，Thread对象必须有一个关于它自己的引用，这个引用同样被一个boost::shared_ptr封装，并返回给调用者。这就促使了对工厂模式的使用。ThreadFactory创建了这个原始Thread对象和一个boost::shared_ptr包装器，并通过调用一个私有helper方法允许它通过boost::shared_ptr封装建立一个到它自身的弱引用，这个类实现了Thread接口(在本例中是PosixThread::weakRef)。

Thread和Runnalbe对象相互引用。一个Runnable对象可能需要知道那个它在其中执行的线程的信息。对于一个线程，显然地需要知道那个使它持有的Runnable对象。这种内部依赖更加复杂，因为每个对象的生命周期是独立于其它对象的。一个应用有可能创建了一个Runnable对象集，并将其重用到不同的线程中，或者，一旦一个线程已经为一个Runnalbe对象被创建并开始运行，但有可能再去创建，或者忘记创建那个已经被创建的Runnalbe对象。

当Runnalbe类有一个明确的线程方法允许对明确的持有线程绑定时，Thread类在它的构造函数中带了一个boost::shared_ptr引用到它持有的Runnable对象中。ThreadFactory::newThread负责了Thread对象到每个Runnalbe对象的绑定工作。

7.7 线程管理器

线程管理器创建了一个生产者线程池并且允许应用程序在空闲生产者线程可用时调度、执行任务。线程管理器没有实现动态线程池重新调节大小，但是提供了一些原语以便应用能够根据负载添加和删除线程。这样做是因为实现负载度量和线程池调整是应用相关的。比如，有些应用可能想根据通常运行情况下的任务到达率调整线程池大小，任务到达率通过轮询采样得出。另一些应用可能希望简单地通过工作队列的深度和高低水位立即作出反应。与其尽力创建足够处理这些不同方法的一个复杂的API抽象，不如我们简单地让特定应用决定如何调度，并提供特定的原语去实现想要的策略并抽样当前的状态。

7.8 定时器管理器

定时器管理器允许应用程序调度Runnable对象在未来的某个时刻执行。它的具体任务是允许应用程序以固定周期抽样ThreadManager的负载，并根据应用策略调整线程池的大小。当然它可以用来产生任意数量的定时器和事件告警。

定时器管理器的默认实现是使用一个单线程去执行到期的Runnalbe对象。因此，如果一个定期器操作需要做大量的工作，尤其是做阻塞性I/O操作的话，那就应该在另外的线程中来做。

7.9 非阻塞操作

尽管Thrift传输接口更多的是直接映射到阻塞I/O模型，但是我们在C++实现了一个基于libevent和TFramedTransport的高性能TNonBlockingServer。我们通过把I/O操作移动到一个紧凑的事件循环中实现了它，事件循环使用状态机表示。本质上，事件循环读TMemoryBuffer对象中编好帧的请求，一旦整个请求准备好他们就被分发给TProcessor对象，TProcessor对象可以直接从内存中读取数据。

7.10 编译器

Thrift编译器是通过使用lex/yacc进行词法分析和解析，并由C++实现的。尽管可以使用代码行数更少的其他语言实现（也就是Python Lex-Yacc（PLY）或者ocalmlyacc），但是我们使用C++强制对语言构造函数的显示定义。强定义的解析树元素（有争议地）使代码对于新开发者更友好。

代码产生分为两个过程，第一个过程只为头文件和类型定义的目的扫描。类型定义在这个阶段不会被检查，因为它们可能依赖头文件。在第一个过程，所有头文件都是被顺序地扫描。一旦头文件树已经被决定，那么对于所有文件的第二个过程就是将类型定义插入到解析树中，并且对任何未定义的类型报错。然后程序对照解析树被产生。

由于固有的复杂性和存在循环依赖的潜在危险，我们明确地禁止前向声明。两个Thrift结构不能相互包含彼此的实例。（因为我们在产生的C++代码中，不允许空结构实例，这实际是不可能的）

7.11 TFileTransport

TFileTransport根据输入数据的长度将其编帧并写到磁盘上，以此记录Thrift的请求/结构。通过一种磁盘上编帧的格式允许更好的错误检查，并帮助处理有限数量的离散事件。

当记录大量数据的时候，TFileTransport使用系统内存交换缓冲区确保良好的性能。一个Thrift日志文件被分割成若干特定大小的块；记录信息不允许跨越块的边界。一个可能引起跨越的消息将触发对该文件最后部分的填充，消息的第一个字节与下一个分块的开始对齐。对文件分块使从一个文件特定点开始读和解释数据成为可能。

8. Facebook Thrift 服务

在Facebook,Thrifty已经被部署到大量应用中，包括搜索、日志、移动，广告和开发者平台。下面将讨论两个特定应用。

8.1 搜索

Thrift被作为Facebook搜索服务的潜在协议和传输层。对于搜索来说，多语言的产生非常适合，因为他允许为应用开发使用一种高效的服务器端语言（C++）并且允许Facebook基于PHP的web应用通过使用了Thrift PHP库调用搜索服务。也有有大量搜索状态、部署和测试性功能建立在产生的Python代码上。除此之外，Thrift日志文件格式被用作

“重做日志”(redo log)来提供实时搜索索引的更新。Thrift已经允许搜索团队根据每种语言的特点来选择开发语言，并以此加快开发进度。

8.2 日志

Thrift TFileTransport的函数性是用来结构化日志的。每个服务函数与其参数的定义都能够被看作一个被函数名标识的结构化日志入口。这个日志能够在其后被广泛使用，包括在线和离线处理，状态聚集和作为“重做日志”(redo log)。

9. 结论

Thrift通过使工程师能够高效分治处理，而在Facebook建立起了可扩展的后端服务。应用开发人员可以关注于应用代码而不必担心套接字层。通过在同一个地方写缓存和I/O逻辑，我们避免了重复工作，这样比在应用程序中分散处理好。

Thrift已经在Facebook中的大量不同应用中部署，包括搜索、日志、移动广告和开发者平台。我们发现使用一个额外的软件抽象层所带来的微小花费与在开发效率和系统稳定性上的收获相比，微不足道。

A. 类似的系统

下面是一些与Thrift类似的软件系统。每个都非常简略的描述：

- *SOAP.* 基于XML，通过HTTP进行web服务，过多的XML解析消耗。
- *CORBA.* 使用相对广泛，由于过度设计和重量级受到争议。同样笨重的软件安装。
- *COM.* 主要在Windows客户端软件被选择，不是一个开放的解决方案。
- *Pillar.* 轻量级和高性能，但是没有版本管理和抽象。
- *Protocol Buffers.* Google拥有的闭源产品，在Swazall论文中被描述。

致谢

非常感谢Martin Smith, Karl Voskuil 和Yishan Wong对于Thrift的反馈。

Thrift是Pillar的继承者，这是个AdamD’Angelo开发的类似系统，开始在Caltech公司后来在Facebook公司继续。如果没有Adam的洞察力，也就绝不会有Thrift了。

References

[1] Kempf, William, “Boost.Threads”, http://www.boost.org/doc/html/threads.html

[2] Henkel, Philipp, “threadpool”, http://threadpool.sourceforge.net