# Thrift：可伸缩性、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

摘要

Thrift 是Facebook 开源的用于提高高效、可伸缩性后端服务的研发与实现的一组代码库与工具集的集合。它的主要目标是通过将每种语言中最需要定制的部分抽象到每种语言所实现的公共库中，实现跨编程语言的高效可靠的通信。具体的讲，Thrift 允许开发者通过定义数据类型与服务接口的方式来生成构建 RPC客户端与服务器所需要的所有代码。

本论文详述了我们在使用 Thrift 时的动机与设计选择，以及一些较为有趣的实现细节。它并不是用来作为研究的，而是一个关于我们做了什么和为什么做的解释。

1. 介绍

随着Facebook的流量和网络结构不断扩大，网站上许多操作（即搜索、广告选择和发布、事件日志记录）的资源需求已经大大超出了LAMP框架的范围。在很多服务的实现中，为了达到性能、开发的速度与便利性、已有库的可获得性，我们使用了大量的语言。大体上说，Facebook 的工程文化趋向于选择最好的工具与实现，而不是标准化任何一门语言并且不情愿的接收其固有的局限性。

考虑到了上述设计上的选择，我们面临一个在多语言中建立一个透明并且高新能的通信框架的挑战。我们发现常见的解决方案要么有太多限制、要么没有充足的自由的数据类型、要么性能很差。

我们所实现的解决方案有两部分，一部分为通过多种语言实现的语言中立的软件栈，另一部分是一个可以将简单的接口和数据类型的定义语言生成客户端和服务器远程过程调用代码库的代码生成工具。通过动态的生成静态代码允许我们创建带有隐式保证的验证代码，这些保证可以在不需要任何高级的内部运行时类型检查的情况下运行。对于开发人员来讲，它也是很方便的，通常可以通过定义一个很简短的文件来定义所有必须的数据结构跟接口，进而实现一个很复杂的服务。

令人惊讶的是，对于这些相对常见的问题，还没有强大的开源的解决方案，因此，在项目的早期，我们就致力于 Thrift 的开源。

在评估网络环境中跨语言交互的挑战时，定义了一些关键组件：

Types. 公共类型系统必须跨编程语言存在，而不需要应用程序开发人员使用自定义的 Thrift数据类型或编写自己的序列化代码。也就是说，C++程序员应该能够透明地交换一个动态Python字典的强类型STL映射。任何程序员都不应该被迫在应用层下面编写任何代码来实现这一点。第2节详述了 Thrift 数据类型系统。

Transport. 每种语言必须有双向数据通信的的通用接口。既定传输协议的实现细节应该与服务开发者无关。同样的应用程序应该能够通过 TCP 流，内存中的原始数据，磁盘上的文件运行。第3节详述了 Thrift 传输层协议。

Protocol. 协议层指数据类型为了使用 Transport 层所使用的一些编码和解码的一些方式。当然，应用程序开发者也无需关心这一层。服务本身使用是否使用 XML或者二进制协议进行编码解码对于应用代码是无关紧要的。重要的是，数据可以用一致的、确定性的事物来读写。第四节详述了 Thrift 协议层协议。

Versioning. 对于健壮的服务，所涉及的数据类型必须提供一种自我版本控制的机制。具体来说，应该可以在不中断服务的情况下（或者更糟的是，严重的段错误）添加或删除对象中的字段或更改函数的参数列表。第5节详细介绍了Thrift的版本控制系统。

Processors. 最终，我们生成能通过处理数据流来完成远程过程调用的代码。第6节详述了生成代码与TProcessor 范例。

第7节讨论了实现细节，第8节描述了我们的结论。

1. 类型

Thrift类型系统的目标是不管开发者使用什么数据类型，都可以完全本地化定义数据类型。处于设计上的考虑，Thrift不引进任何动态类型或者包裹对象，也不要求开发者为了对象序列化或传输写任何代码。Thrift 逻辑上是开发者为了告诉代码生成器如何使用各种语言传输对象，而使用最小的额外信息对他们的数据结构进行注释的一种方式。

* 1. 基础类型

Thrift 类型系统基于一些基本类型系统。在考虑支持哪种类型时，我们的目标是清晰和简单，而不是丰富，重点关注所有编程语言中可用的关键类型，并建议仅在特定语言中可用的任何具体类型。

Thrift支持的所有如下：

* bool，布尔值，true 或 false
* byte，有符号单字节值
* i16，有符号双字节值
* i32，有符号四字节值
* i64，有符号八字节值
* double，64位浮点数
* string，编码不可知文本或二进制串

特别要注意的是，没有无符号整数类型。因为这些类型在很多语言中没有原始的基础类型与之对应，所以他们提供的优势已经丧失了。更进一步的讲，没有办法保证像Python应用程序开发者将一个负数赋值给一个整数值，这将导致无法预测的行为。从设计的角度来看，我们观察到无符号整数很少（如果有的话）用于算术目的，但在实践中更常被用作键或标识符。在这种情况下，符号是不相关的。有符号整数提供了相同的目的，并且可以在绝对必要时安全地转换到它们的无符号的的对应值上（最常见的是C++）。

* 1. 容器

Thrift容器是强类型容器，并且可以映射到常用语言的常用容器中。他们使用C++ 模板（或Java 泛型）风格进行注释。Thrift有三种可获得的容器类型：

* list<type> 有序元素列表，被映射为C++ STL中的vector, Java 中的ArrayList，或脚本语言中的原始数组。可能包含副本。
* set<type> 无需唯一元素集合，被映射为C++ STL中的set ，Java 中的HashSet， 或PHP、Python、Ruby 中的原始字典。
* map<type1, type2> 严格唯一的key与value 的映射，被映射为C++ STL 中的map，Java中的HashMap，PHP中的联合数组，或Python、Ruby中的字典。

提供默认值时，类型映射没有显式固定。自定义代码生成器指令已被添加到目的语言中替换自定义类型（即，可以在C++中使用哈希映射或谷歌的稀疏哈希映射）。唯一的要求是自定义类型支持所有必要的迭代原语。容器元素可以是任何合法的Thrift类型，包括容器或结构体。

* 1. 结构体

一个Thrift 结构体定义了一个在各种语言中通用的对象。一个结构体大体上跟面向对象语言中一个类是等价的。结构体是强类型域的集合，每一个域都具有唯一的标识符。定义Thrift基础结构的语法跟定义C结构体的语法是相似的。每个域可以有一个整数域标识符注释（具有当前结构体域唯一性）和默认值。

1. **struct** Example {
2. 1:i32 number=10,
3. 2:i64 bigNumber,
4. 3:**double** decimals,
5. 4:string name="thrifty"
6. }

在生成目标代码中，每种定义有两个方法，read 和 write，分别用于用Thrift TProtocol 对象序列化和传输对应的对象。

* 1. 异常

异常句法上和功能上同结构是一致的，除了声明时使用的是exception关键字而不是struct关键字。生成的对象继承自每个目标编程语言中适当的异常基类，其目标是为开发人员在任何给定语言中提供与本机异常处理的无缝集成。同样，设计重点是让应用程序开发人员熟悉代码。

* 1. 服务

Service用于使用Thrift类型。定义一个Service语义上等同于在面向对象编程语言中定义一个纯虚接口。Thrift编译器生成完全功能上的实现接口的服务器与客户端桩。Service被定义如下：

1. service <name> {
2. <returntype> <name>(<arguments>)
3. [throws (<exceptions>)]
4. ...
5. }

例如：

1. service StringCache {
2. **void** set(1:i32 key, 2:string value),
3. string get(1:i32 key) throws (1:KeyNotFound knf),
4. **void** **delete**(1:i32 key)
5. }

注意除了Thrift类型之外，void 类型也可以作为函数返回值。除此之外，一个async 标识符也可以被添加到void 函数。它会使编译器生成不会等待服务器返回响应的代码。注意为了告诉客户端服务器端的方法已经执行完，即使是一个void方法，也会返回一个响应。使用async方法调用时，只能保证客户端在传输层上成功执行请求。（在许多运输场景中，由于拜占庭将军的问题，这本质上是不可靠的。因此，应用程序开发人员应该注意只在当前方法调用失败可接受或传输可靠的情况下使用async关键字优化。）

同样值得注意的是，函数的参数和异常列表是作为节俭结构实现的。它们在符号和行为上都是相同的。

1. 传输层

Transport层被生成代码使用来促使数据传输。

* 1. 接口

Thrift一个关键的设计上的选择是将传输层从代码生成层抽象出来。虽然Thrift通常用在TCP/IP栈的顶部，流套接字作为通信的基础层，但没有令人信服的理由将这种约束构建到系统中。与实际I/O操作（通常调用系统调用）的成本相比，抽象I/O层（每个操作大约一个虚拟方法查找/函数调用）所产生的性能折中无关紧要。

基本上，生成的Thrift代码仅需要知道如何读和写数据。数据的来源与去向是无关紧要的，它可能是socket，一段共享内存，或者本地磁盘上的文件。Thrift传输层接口支持下列接口：

* open() 打开传输层
* close() 关闭传输层
* isOpen() 判断传输层是否打开
* read() 从传输层读取数据
* write() 向传输层写入数据
* flash() 强制刷新写数据

这里还有一些没有记录的附加方法，用于帮助成批读取，并可选地通过生成的代码来指示读取或写入数据块的完成。

除了上述TTransport接口之外，还有一个TServerTransport接口用于接受或创建原始传输对象。接口如下：

* open() 打开传输
* listen() 开始监听等待连接
* accept() 返回一个新客户端传输
* close() 关闭传输
  1. 实现

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

* + 1. TSocket

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

* + 1. 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 曰志中重现，可用来后继处理或复制（模拟）过去的事件.

* + 1. 工具

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。

1. 协议层

**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

* 1. 接口

Thrift协议接口是十分易懂的，它基本上有两条原则：1、双向数据通信，2、对基本数据结构、容器、结构的解码，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(protocolencoder)写入，这些被写入的内容可以通过相匹配的协议解码器(protocoldecoder)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）。

* 1. 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。在很多场彔下，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彔，通过使用 TFrameTransport抽象是很容易把它添加到传输层的。

* 1. 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。

1. 版本控制

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曰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。

* 1. 域标识符

版本管理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 mmber=10 ,

2:i64 bigN-omber,

3:double decimals t 4:strmg nams = "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 StringCaclis {

void sst(l:i32 ~key t 2:string value) ^

string gst(l:i32 ksy) throws (1:KsyNotFound knf) t

void deleteC1: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il6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。

* 1. 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，PHP中的null, Python中的none, Ruby中的nil）本质上 说，每个Thrift结构内部的isset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査这个是否罝位。

1. **class** Example {
2. **public**:
3. Example() :
4. number(10),
5. bigNumber(0),
6. decimals(0),
7. name("thrifty") {}
8. int32\_t number;
9. int64\_t bigNumber;
10. **double** decimals;
11. std::string name;
12. **struct** \_\_isset {
13. \_\_isset() :
14. number(**false**),
15. bigNumber(**false**),
16. decimals(**false**),
17. name(**false**) {}
18. **bool** number;
19. **bool** bigNumber;
20. **bool** decimals;
21. **bool** name;
22. } \_\_isset;
23. ...
24. }
    1. Case 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罝位，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。

* 1. 协议层/传输层版本控制

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〇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。

1. RPC 实现
   1. TProcessor

在Thrift设计中，最后一个核心的接口就是TProcessor,，它有有简单的构造函数。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.

* 1. 生成代码

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handler类），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接口文件，我们产生虚拟服务接口，一个客户端被生产，它实现了接口，并使用两个TProtocol 接口IO。

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。

* 1. 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卖现，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。没有逊要，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).

1. 实现细节
   1. 目标语言

Thrift 目前支持5种目标语言：C++, Java, Python, Ruby, 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HttpClient透明地访问我们很 多前端结构，HttpClient实现了 TTransport的接口。

* 1. 结构生成

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〇和get〇方法，类似，isset对象的使用不是强制性的。我们没有包含任何FieldNotSetEsception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〇和write〇方法放罝在公共作用域中，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。

* 1. 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査找高效地完成。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.

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。

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 易地通过检査方法名，调试崩溃或错误的数据。

* 1. 服务器与多线程

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服务器端的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。

* 1. 线程原语

Thrift线程库在fackbook::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<T>, 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的卖现。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—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。

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Rmmalbe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。

* 1. Thread，Runnable和shared\_ptr

我们在ThreadManager和TimerManager实现的整个过程都使用了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至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 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。

* 1. ThreadManager

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。

* 1. TimeManager

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瞥。

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。

* 1. 非阻塞选项

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对象中编好倾的请求，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。

* 1. 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査，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)

* 1. 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査，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。

1. FackBook Thrift 服务

在Facebook，Thrift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：

* 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。

* 1. 日志

ThriftTFileTransport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”（redolog）。

1. 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。