Apache Thrift 框架结构：自底向上

概览

介绍

Apache Thrift 是开源，跨语言的**序列化**和 **RPC** 框架。

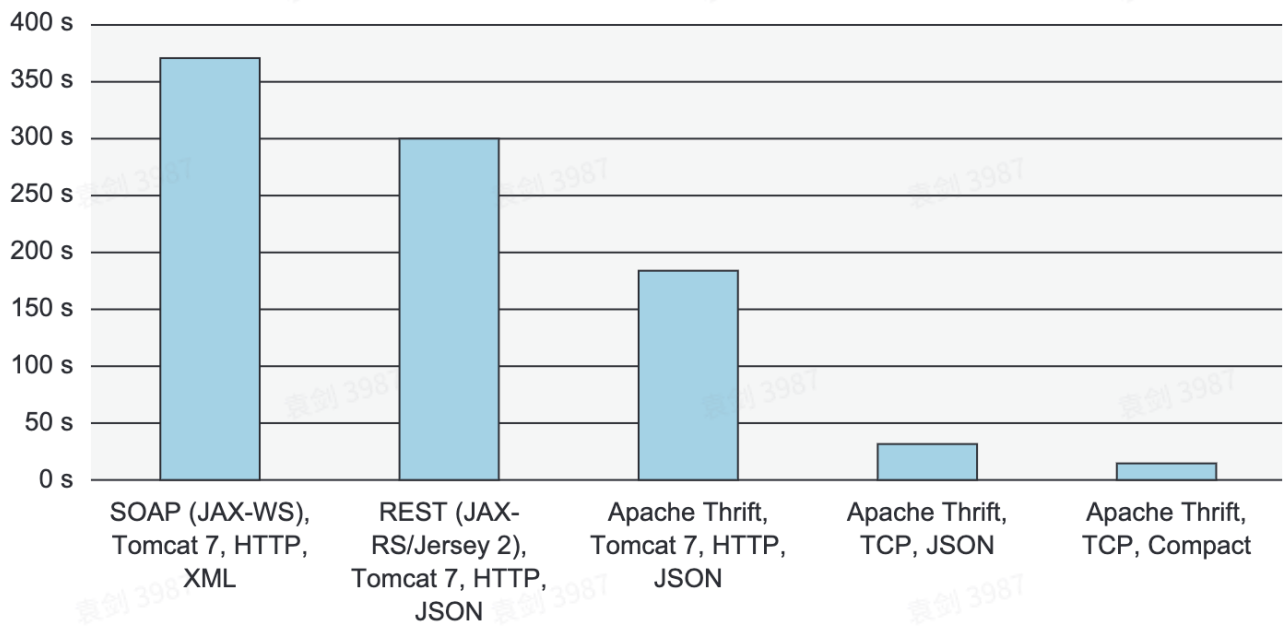
Thrift 是由 **Thrift 编译器**（compiler）和**特定语言库**（library）构成的。

其中编译器负责生成 RPC 框架所需要的代码，特定语言库提供所使用的编程语言的传输、序列化等功能。Thrift 的特定语言库是可以抛开编译器单独使用的，比如使用 Thrift 库中的序列化、网络传输、文件读写等功能；但是编译器生成的代码是必须要依赖相应语言的 Thrift 库。

一些主流通讯框架比较

- **REST**：基于 HTTP，相比较二进制的 Thrift 会慢许多。
- **Protocol Buffers**：Protocol Buffers 是一个序列化框架，其本身并不支持 RPC。Protocol Buffers 序列化性能远优于 Thrift 的序列化模块，但是 CPU 开销更大。文档友好程度吊打 Thrift。
- **gRPC**：基于 Protocol Buffers 和 HTTP/2 的 RPC 框架，虽然序列化性能表现要比 Thrift 好很多，但是在服务请求中网络 I/O 作为最大的性能瓶颈，基于 TCP 的 Thrift 的性能通常优于基于 HTTP 的 gRPC。同时 Thrift 支持的语言更丰富。

当你需要 **RPC 框架**、**序列化**、**模块化特性**、**性能**以及**广泛的语言支持**全都在**一个工具包**中的时候，你可以选择 Thrift，其他情况下，gRPC 或者 REST 可能是更好的选择。



性能比较：不同 Java 服务器的一百万次服务请求耗时

框架结构

总体结构

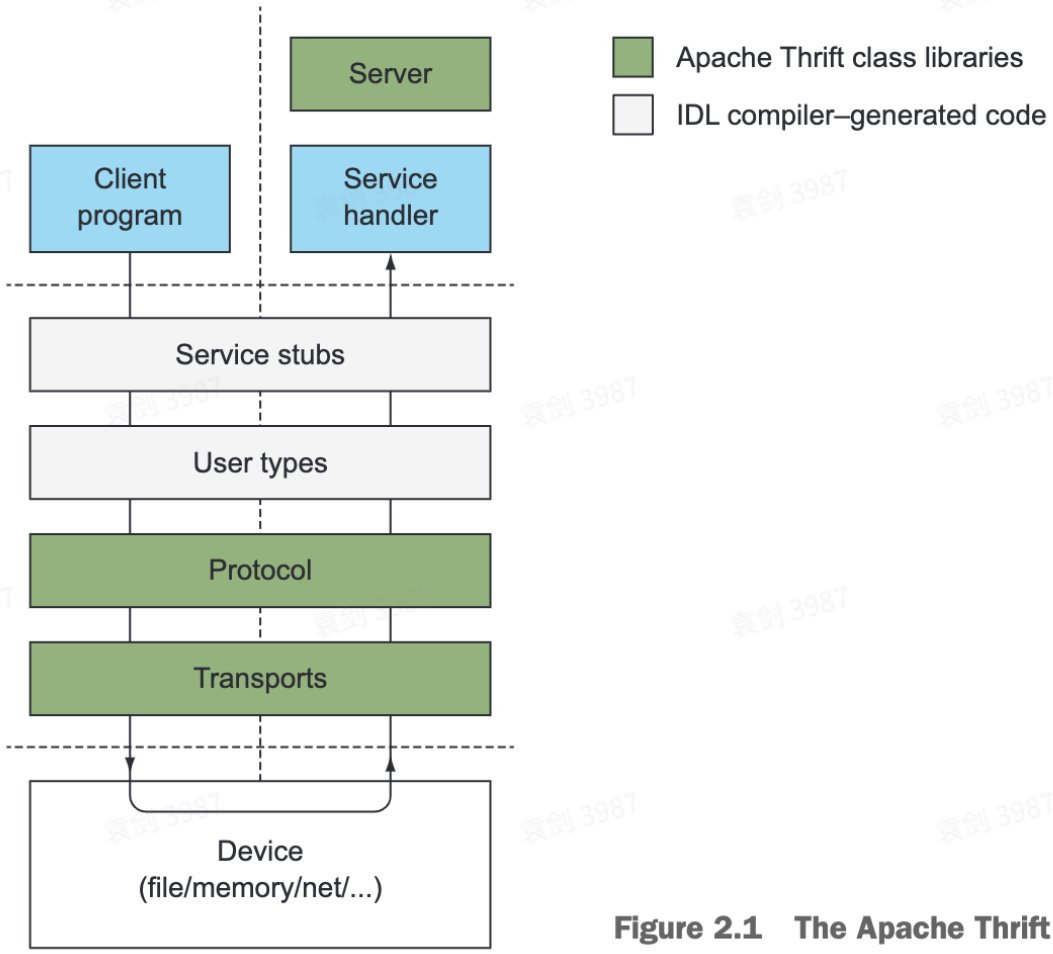


Figure 2.1 The Apache Thrift framework

Apache Thrift 框架可以被分为五层：

- **The RPC Server library**：服务器功能代码，如连接、线程管理（来自特定语言库的支持）
- **Service Stubs**：桩代码，实现远程调用（来自 Thrift 编译器生成）
- **User types**：用户自定义类型，即各种 struct 的实现（来自 Thrift 编译器生成）
- **Protocol**：序列化协议（来自特定语言库的支持）
- **Transports**：传输层（来自特定语言库的支持）

tips：Thrift 库中所有的类都以 T 开头。如 `TTransport`、`TProtocol`。

Transports

Transports 层在 Thrift 框架中处于**最底层**，承担**读写设备、网络（传输）**的任务。同时 Transports 层内部具有分层设计。

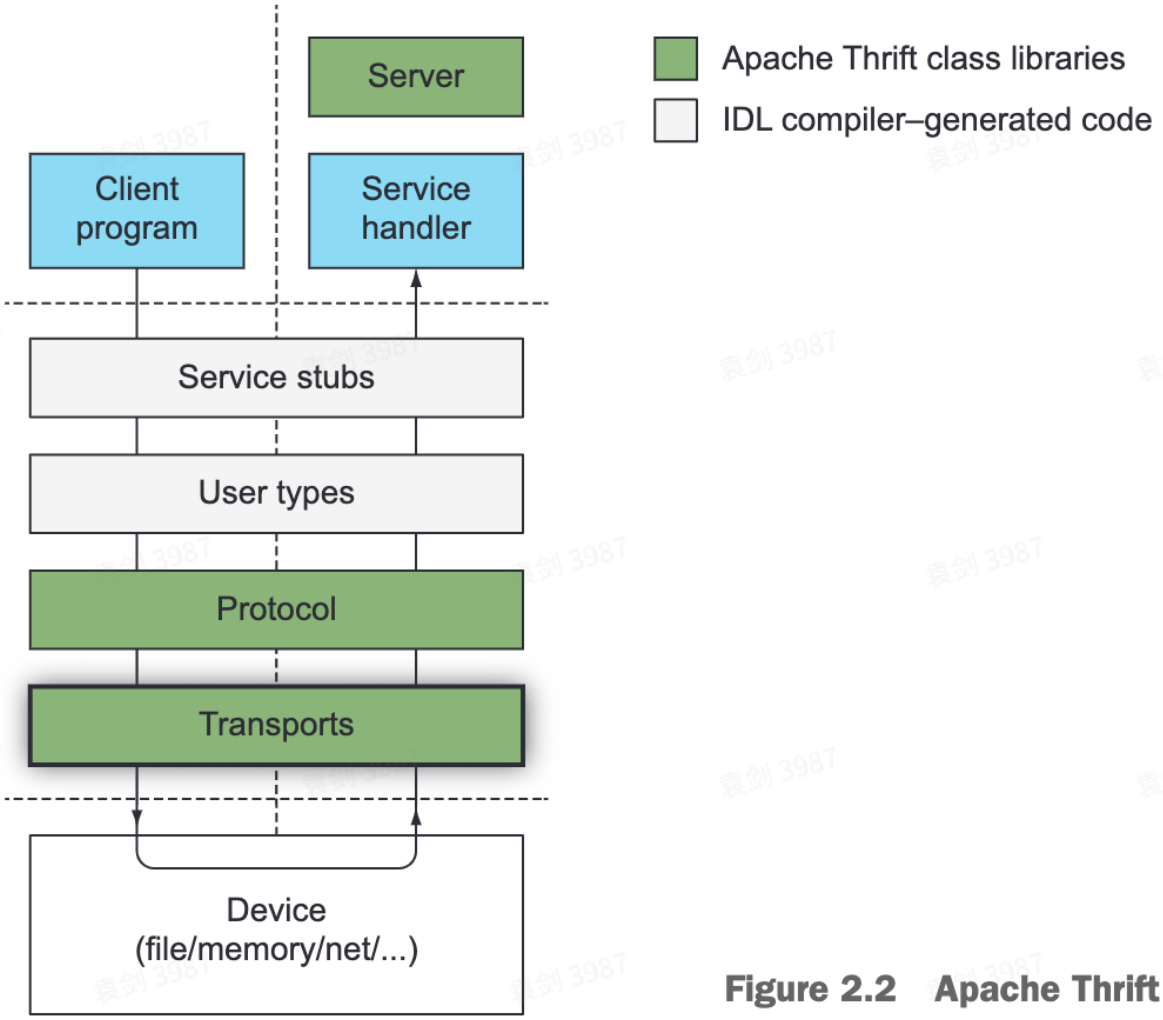


Figure 2.2 Apache Thrift transports

`TTransport` 是一个具有**抽象接口**的类。它定义了基本的**设备无关**的**字节级**读写操作。

基本的 transport 操作：

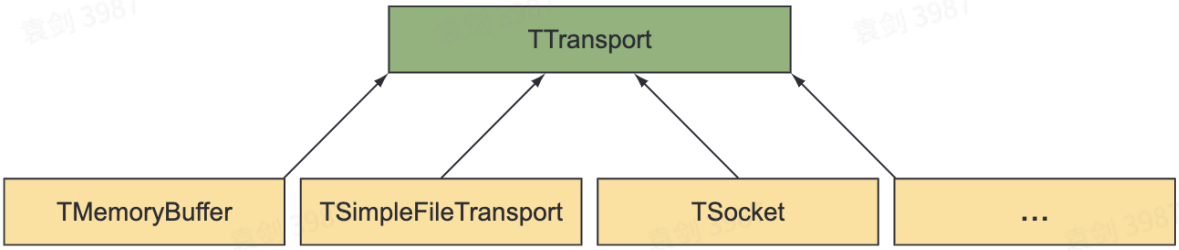
- `open()` ——开启 transport
- `write()` ——将字节流写入 transport
- `read()` ——从 transport 读取字节流
- `flush()` ——将缓冲区的字节流写入到终端设备
- `close()` ——flush 然后关闭 transport

在绝大多数语言中，各种不同功能的 Transport 都实现了上述接口。

在一般的 Thrift 编程中，只需要我们自行处理 `open()` 和 `close()`，但是 read 和 write 操作会被序列化模块 protocols 执行，但是 protocols 不知道什么时候读写了一个完整的 RPC message，所以在更高层级中，由 Thrift 编译器生成的 RPC stub 代码来调用 flush。

Endpoint transports

Endpoint tranports 实现了和底层的通讯，如物理设备（内存、磁盘文件）或者网络（socket），你甚至可以自定义终端通讯协议，只需要实现 `TTransport` 的接口。



Thrift 库中 Endpoint transports 继承关系

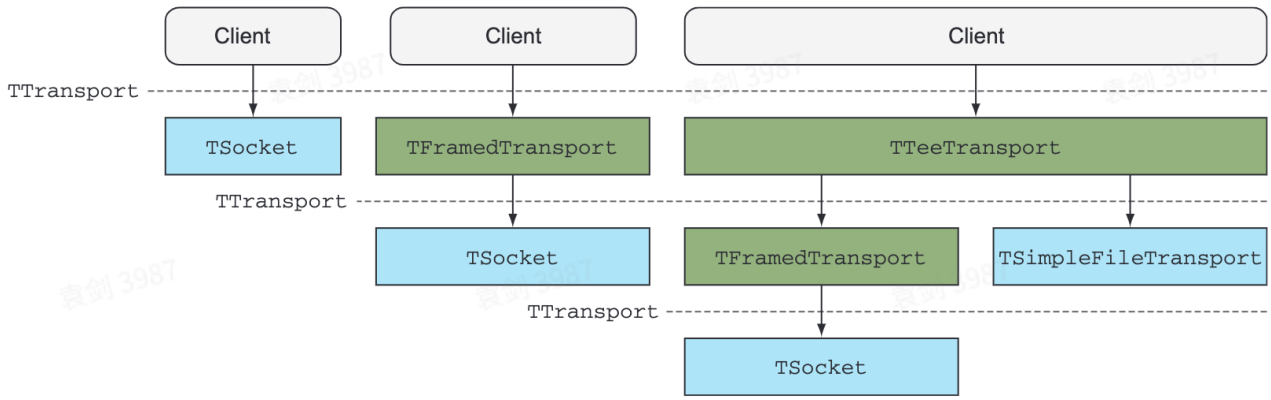
大多数的 Thrift 语言库都提供了三种最重要的 transport：

- Memory transports——读/写内存块，可以用作缓冲和缓存
- File transports——读/写磁盘文件, 可以用作日志和对象持久化
- Network transports——读/写网络设备, 可以用作远程调用

Layered transport

在 endpoint transport 之上可以叠加任意层数的 layered transport，这些所有的 transport 共同构成一个 transport stack。

每个 layered transport 都实现并暴露标准的 `TTransport` 接口，所以增加 layered transport 对于上层的调用者来说是完全透明的。可以在不修改其他代码的情况下增减 layered transport。



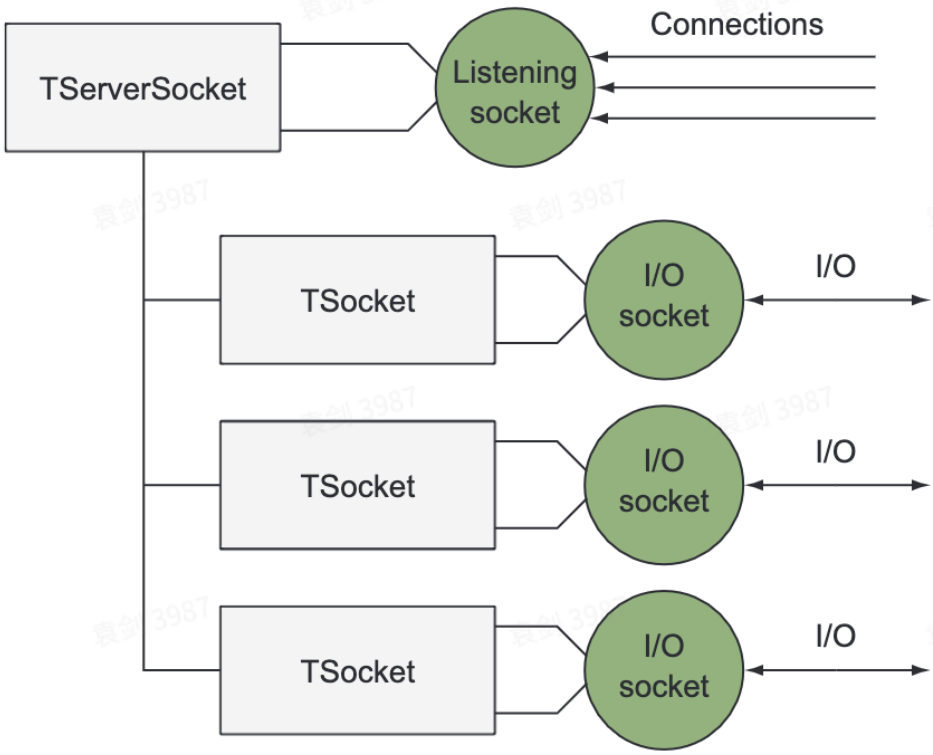
layered transport

两个重要的 layered transport：

- TBufferTransports——使用 buffer，减少 IO 开销。
- TFramedTransport——在了个 4 字节的 Header 来表示消息体的大小，以此来实现非阻塞 IO 和异步处理。

Server transports

服务器上的 transport 主要是监听端口、建立连接。服务器上的 `TServerSocket` 并非一个真正的 transport，`TServerSocket` 采用工厂模式，对客户端的连接请求创建上述的 endpoint transport，通常是 `TSocket`，从而实现通讯。



`TServerSocket` 监听端口、创建 `TSocket` 建立连接

小结：

- Transports 在 Thrift 框架中位于最底层。
- `TTransport` 定义了 transports 所要实现的抽象接口。
- Endpoint transports 实现了 `TTransport` 并且完成针对终端设备的读写操作。
- Server transports 使用工厂模式来创建 endpoint transports。
- Layered transports 在 endpoint transports 之上透明地提供其他功能特性。

Protocols

在 Thrift 语境中，protocol 指的是**序列化方式**，位于 Transports 层上方。

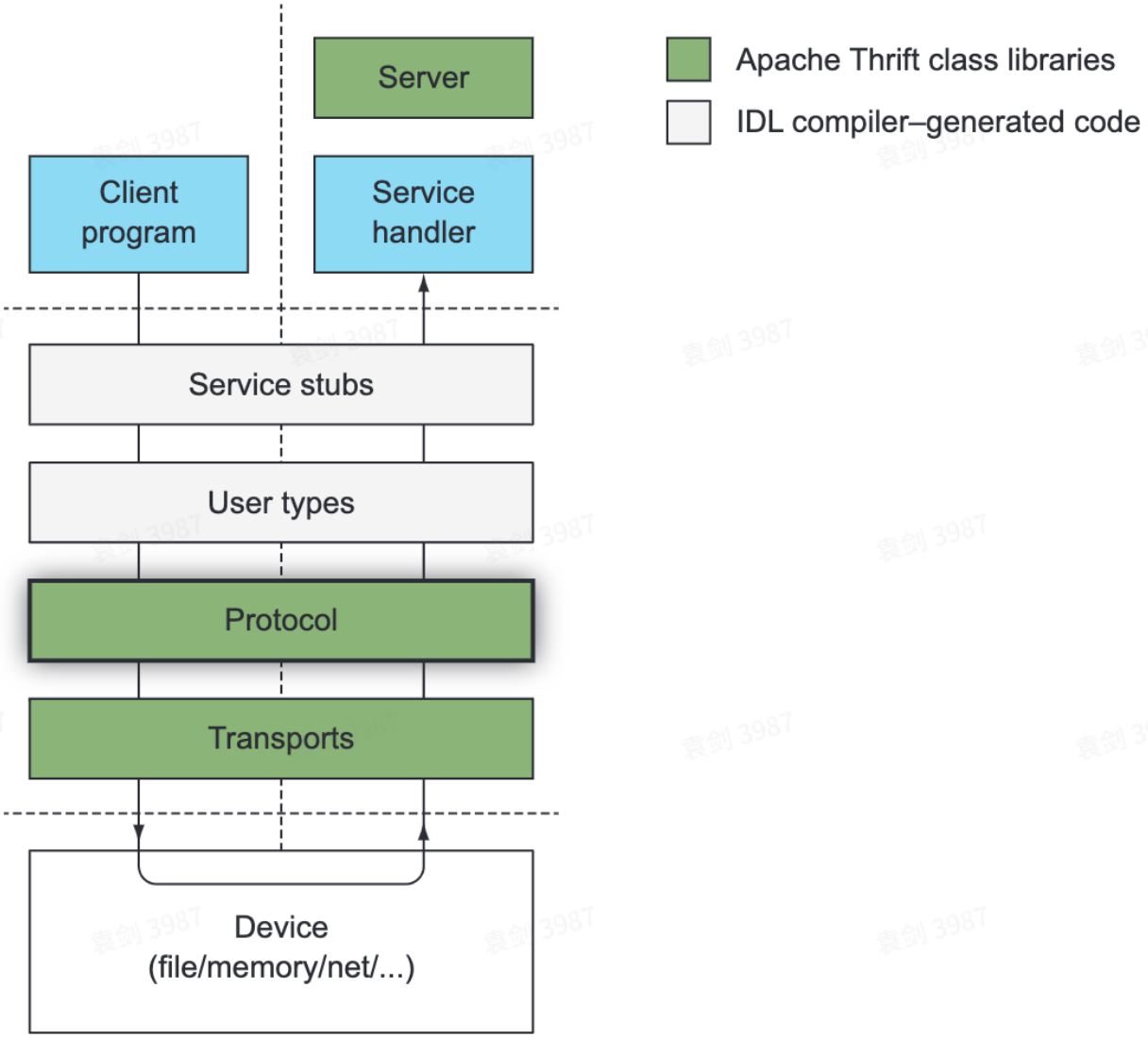


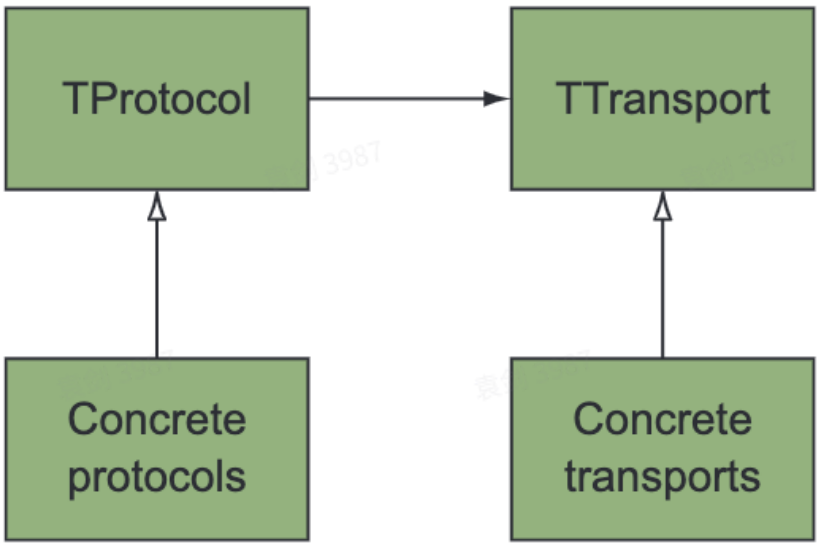
Figure 5.1 The Apache Thrift protocol layer

Thrift 提供了一个抽象接口叫做 `Tprotocol`。提供了一系列针对 Thirft 类型系统的**序列化**和**反序列化**方法。

例如：`writeI32()`、`readI32()`、`writeListBegin()`、`writeListEnd()` 等。

因此一个 protocol 可以在不修改其他代码的情况下被另个一替换。

`TProtocol` 的实现会持有一个 `TTransport` 的引用来进行读写操作。`TTransport` 的引用一般在 protocol 被构造的时候传入，绑定在一起作为整个 I/O stack 的一部分。任何 protocol 可以使用任何 transport，因为 protocol 仅仅依赖于抽象的 `TTransport` 接口，它并不关心底层的实现。



Transport/protocol 依赖关系

Apache Thrift 提供的三种序列化方式：

- The Binary Protocol——简单而快速（默认）
- The Compact Protocol——更小的体积，减少 IO 开销（压缩算法增加了 CPU 的额外开销）
- The JSON Protocol——基于统一标准，可读，广泛运用

Protocols 的作用是标准化不同平台和语言的序列化方式，实现跨平台和跨语言的通讯。

protocols 所支持的序列化对象：

- RPC message——携带完整的调用、返回、异常信息
- 结构体——struct
- 容器——list, set, map
- 基本类型——int, double, string...

容器序列化

Thrift 序列化系统为了避免增加不必要的开销，对于 set 中元素的唯一性和 map 中 key 的唯一性都没有做检查。

一个有趣的例子：

Python

```python
#IDL:
service HelloSvc { string hello_func(1: set<i32> s) }

#Client:
client = HelloSvc.Client(protocol)
s = set()
[s.add(x)for x in[1,2,3]]
client.hello_func(s) #用 set 作为参数去调用
print("Python Client: calling with "+ str(s))

l = list()
[l.append(x)for x in[1,1,1]]
client.hello_func(l) #用 list 作为参数去调用
print("Python Client: calling with "+ str(l))
```

Python

```python
class HelloHandler:
    def hello_func(self, s):
        print("Python Server: handling client request: "+ str(s))
        return "Hello thrift, from the python server"
```

客户端结果：

Bash

```bash
$ python hello_client.py
Python Client: calling with set([1, 2, 3])
Python Client: calling with [1, 1, 1]
```

服务器结果：

VBScript

```
$ python hello_server.py
Python Server: started
Python Server: handling client request: set([1, 2, 3])
Python Server: handling client request: set([1])
```

支持鸭子类型的语言，如 Python 可能会出现上述现象。因为 Python 的 Thrift 序列化中对容器的访问使用的是 `for in` 方式而并没有对容器类型做检查，但是反序列化创建 set 的过程中，重复的元素不再被添加到 set 当中，所以 Python 中任何支持迭代的容器都能适用于序列化。

Struct 序列化

Thrift 的 struct 序列化中使用元信息的都是字段 id，而非字段名。

以 binary protocol 为例，字段被序列化的过程：

`（1 byte 的数据类型）+（2 byte 的字段 id）+(数据)` 。

	A	B	C	D
1			简单数据类型	
2	数据类型	类型标识(8位)	编号(16位)	值
3	bool	2		一个字节的值（true：1，fa
4	byte	3		一个字节值
5	double	4		八个字节值
6	i16	6		两个字节值
7	i32	8		四个字节值
8	i64	10		八个字节值
9			复合数据类型	
10	数据类型	类型标识（8位）	编号（16位）	值(长度+值)
11	string	11		四个字节数据长度+数据
12	struct	12		嵌套数据+一个字节停止符
13	map	13		一个字节的key类型+一个字节的val类 据长度+数据的值（key
14	set	14		一个字节的val类型+四个字节的数据
15	list	15		一个字节的val类型+四个字节的数据

接收方在反序列化的时候字段名称一致只能通过代码一致来保证。同时反序列化过程会丢弃无法识别 id 的字段。

小结：

- Protocols 提供跨语言的序列化方式。
- `TProtocol` 定义了 protocols 需要实现的抽象接口。
- Protocols 依赖于 Transport 来读写序列化的字节流。
- Thrift 提供三种序列化方式：binary、compact、json。

IDL

IDL 所描述的接口包含了两个主要的部分：

- 用户自定义类型 UDT（user-defined-type，即 struct）——作为系统之间数据交换的载体。
- 服务 services——暴露出来的方法合集。

本质上, IDL 文件里的声明的任何元素都是接口约定的一部分。

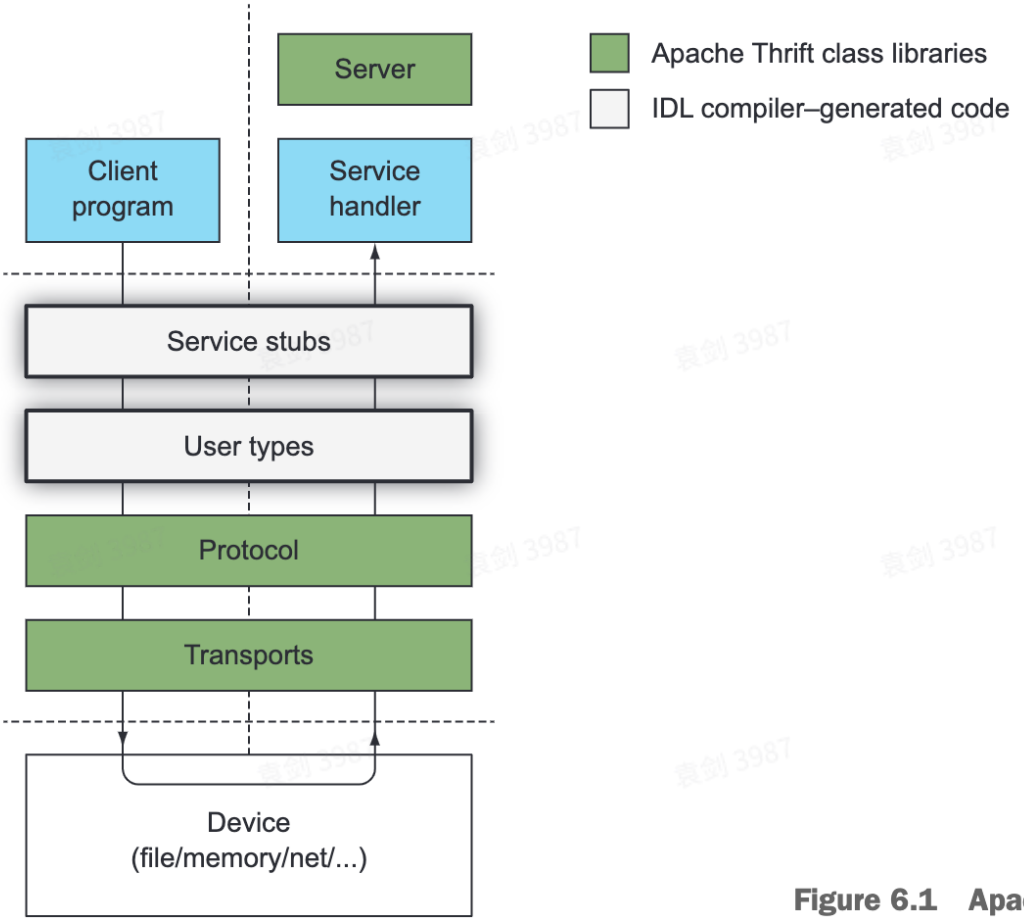


Figure 6.1 Apache Thrift IDL

UDT（User-defined type）in IDL

用户自定义类型就是 IDL 当中的 **struct 结构体**。

Thrift 编译器将 UDT 转换为各个特定语言当中的数据结构，比如 class in C++。这些生成的类型有**内建**的序列化能力。编译器为所有的 UDT 创建 `read(TProtocol protocol)` 和 `write(TProtocol protocol)` 方法来自动化序列化过程。同时参数列表在 Thrift 内部也被当做一个 args struct，使得参数在 RPC 中能够像普通的 struct 一样方便地被序列化。

Field requiredness	Write behavior	Read behavior
`Required`	Always written	Must be read or error
`<default>`	Always written	Read if present
`Optional`	Written if set	Read if present
`<undefined>`	N/A	Ignored

Struct 结构体字段要求度

UDT 的实现：

- 字段数据库——存储每个字段的元数据
- 默认的构造函数——将字段的值初始化为默认值
- `read(TProtocol protocol)` 方法——通过传入的 protocol 参数进行反序列化
- `write(TProtocol protocol)` 方法——通过传入的 protocol 参数进行序列化

UDT 的读取时丢弃字段的实现：

read() 方法会丢弃无法识别 id 的字段，并调用一个 skip() 方法来计算应该丢弃字节流当中的多少数据，以保证后续的反序列化正常进行。

UDT 迭代（即 struct 迭代）：

- 更改字段名——由于序列化过程中没有使用字段名，所以字段名可以随时更改而对整个程序没有任何影响，但是永远不要改变字段的语义。
- 增加字段——添加 default 字段，或者 optional 字段。
- 删除字段——删除字段时，注释掉 idl 中的字段，以保留其先前存在的记录。永远不要重用已删除的 id。不要删除 required 字段。
- 修改字段类型——更改字段的类型会导致接收方忽略该字段，因为 id 和 type 没有同时匹配。一种更好的方式是在可预见未来该字段类型有可能变动的情况下，使用 union。Thrift 中 union 的本质也是 struct。
- 修改字段的要求度——不应该修改 `required` 字段的要求度。
- 修改字段的默认值——在程序双方 IDL 版本不一致时，默认值的不一致可能会对程序产生影响。因为默认发送的情况下发送方和接收方对默认值认知不一致。

Service

Thrift 的 service 由一系列函数组成，Thrift 通过对 service 的调用，发送 message 来完成远程调用。

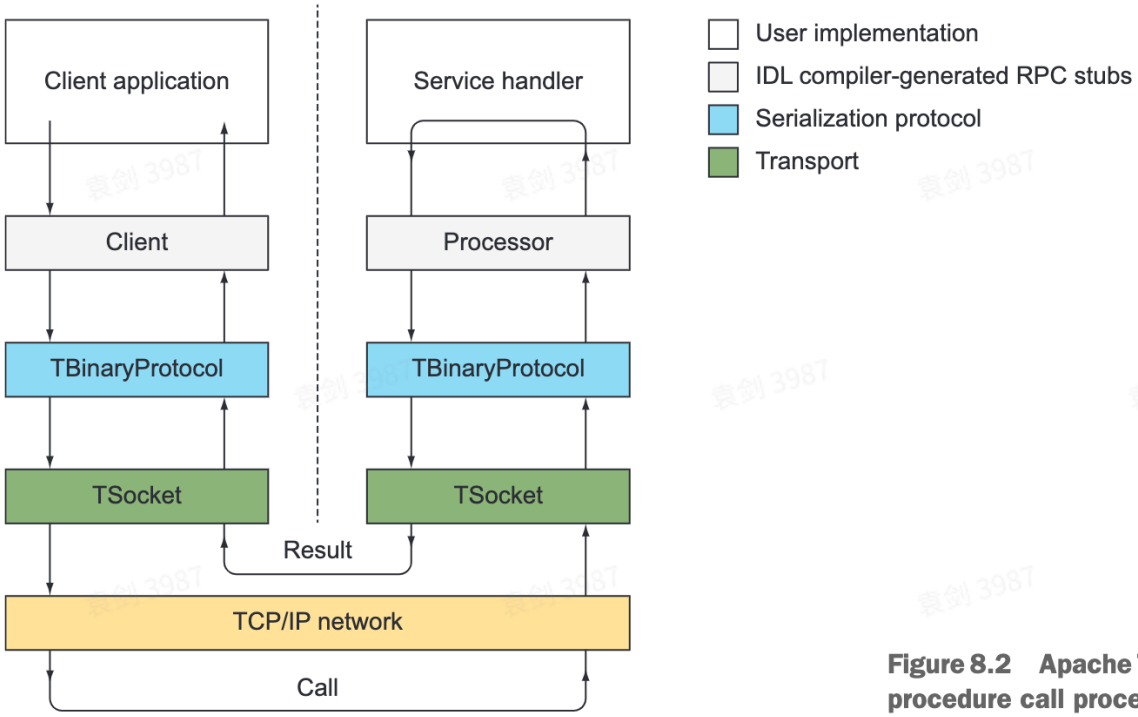


Figure 8.2 Apache Thrift remote procedure call processing

Service 的实现：

- Iface——接口定义。
- IfFactory——工厂类，用来生成接口的实现（handlers）。
- Client——客户端代理，用来调用服务请求。
- Processor——进行调度，将正确的方法 handler 分配给对应的调用
- ProcessorFactory——工厂类，用来生成 processor 实例。A factory designed to manufacture instances of the processor.
- *_args struct——参数结构体，拥有一个方法的完整参数列表。用于参数的序列化和反序列化。
- *_result struct——返回值结构体，用于结果的序列化和反序列化。

接口迭代：

- 接口增加参数
  - 如果新增参数提供了默认值，则旧客户端可以调用更新后的服务。
  - 更新后的客户端可以调用旧的服务，因为旧的服务会直接无视新增参数。
- 接口移除参数
  - 旧的客户端可以调用新的服务，新的服务会直接无视掉已经被移除的参数。
  - 更新后的客户端不能安全地调用旧的服务，除非被移除的参数具有默认值。
- 增加接口
  - 旧的客户端可以调用新的服务。
  - 更新后的客户端不能调用旧的服务，接口已经被移除了。

- 移除接口
  - 旧的客户端不能调用更新后的服务，接口已经被移除了。
  - 更新后的客户端可以调用旧的服务。

值得注意的点：

- void 类型在 IDL 中只能被用作函数的返回类型，void 返回类型的函数在调用完成后仍然会生成 reply message。
- 在一般的语言实现中，每个字段会有一个 `isset` 的标志位来表示字段是否已经被赋值过。值得注意的是：C++ 和很多语言为了减小内存开销对于 `required` 字段没有增加额外的 `isset` 标志位，总是隐式地表示字段已经被设置过。
- Python 中通过字段值是否为 `None` 来判断是否被 set，如果一个字段是 `None`，那么这个字段将不会被序列化。
- 默认值：对于 `default/required` 字段的发送方来说，默认值会被赋值给字段并传输。
- 如果 `default/required` 字段发送发和接收方的默认值不同，则发送方的默认值会覆盖接收方默认值。因为只有当接收方没有接收到该字段才会使用默认值。
- `optional` 字段则情况相反，如果字段没有值发送方不会发送默认值，而接收方会使用默认值进行初始化。因此当双方的默认值不一致的时候，接收方会覆盖发送方。
- service 不支持多继承。method 的实现不能被继承，每个 service 都必须为所有的 method（无论是定义的还是继承的）提供实现。
- 接口的参数无论是 required 还是 default 属性，无论是否有默认值，client 都必须显示地传入参数。参数是不支持 optional 的，如果需要支持 optional，可以使用 struct 来包装参数。
- 默认值对 required 的接口参数没有任何意义，client 必须显示传入 required 参数并且 server 端必须接收到 required 参数，否则报错。
- default 参数的默认值：client 端的参数会无视默认值。default 参数应该总是被 client 传值。参数的默认值唯一作用是在客户端（使用旧 IDL 的客户端，可能并不包含某个新增的参数）缺失某个参数的情况下，server 端使用该参数的默认值以保证服务的正常，这样可以提升 server 的兼容性，在升级 server 的情况下可以同时对新老 client 提供服务。

小结：

- IDL 主要用于描述用户定义的数据类型和服务接口。
- IDL 中字段和参数的不同属性会对数据结构和接口的迭代造成不同的影响，选择合适的要求度属性可以增强程序的兼容性。

Server

Server 在 Thrift 框架中位于服务端的最上层，处理并发管理、可伸缩性和跨线程通信等问题。Server 的特性和语言强相关，主要是不同的语言的并发模型造成的。

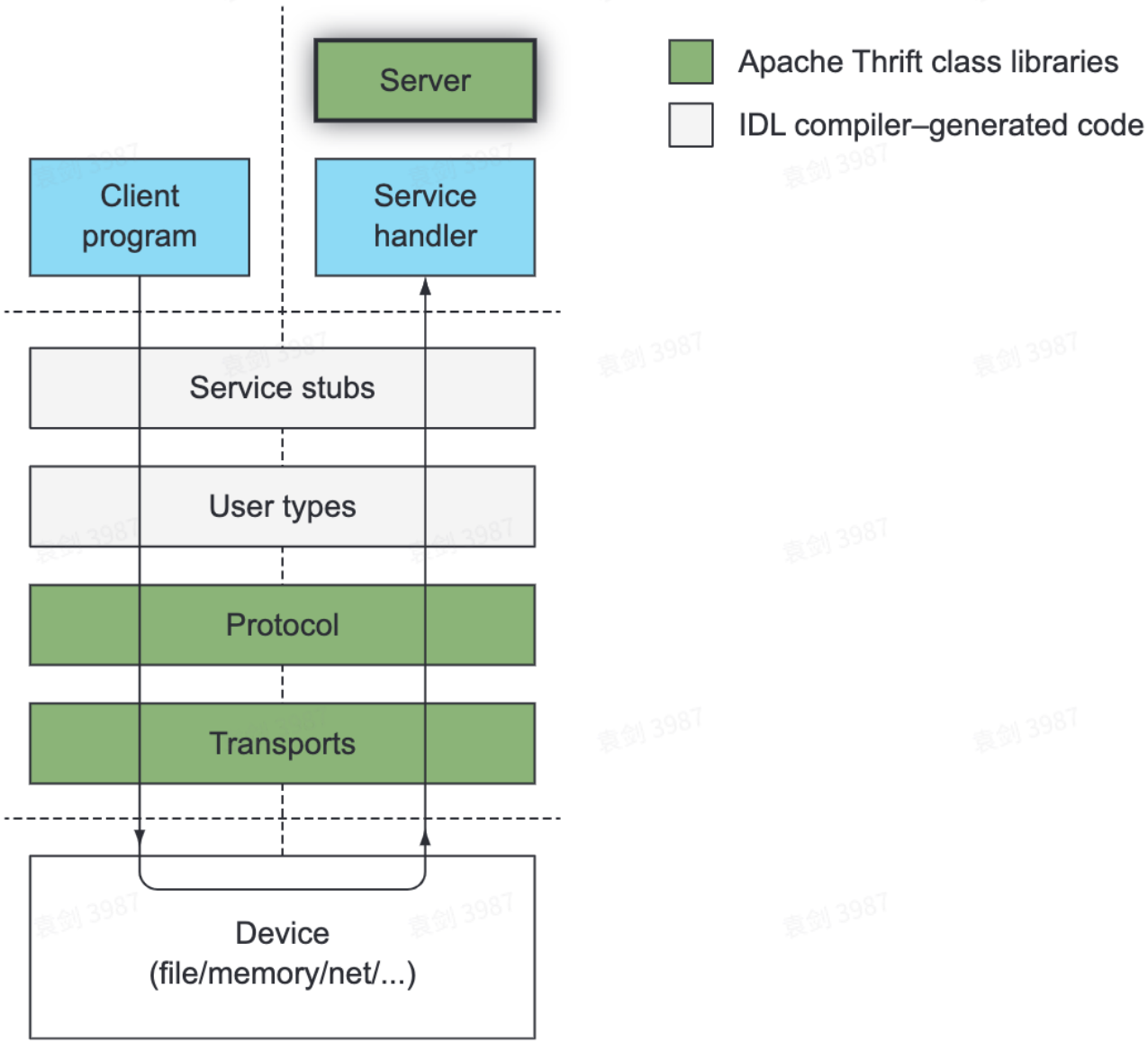
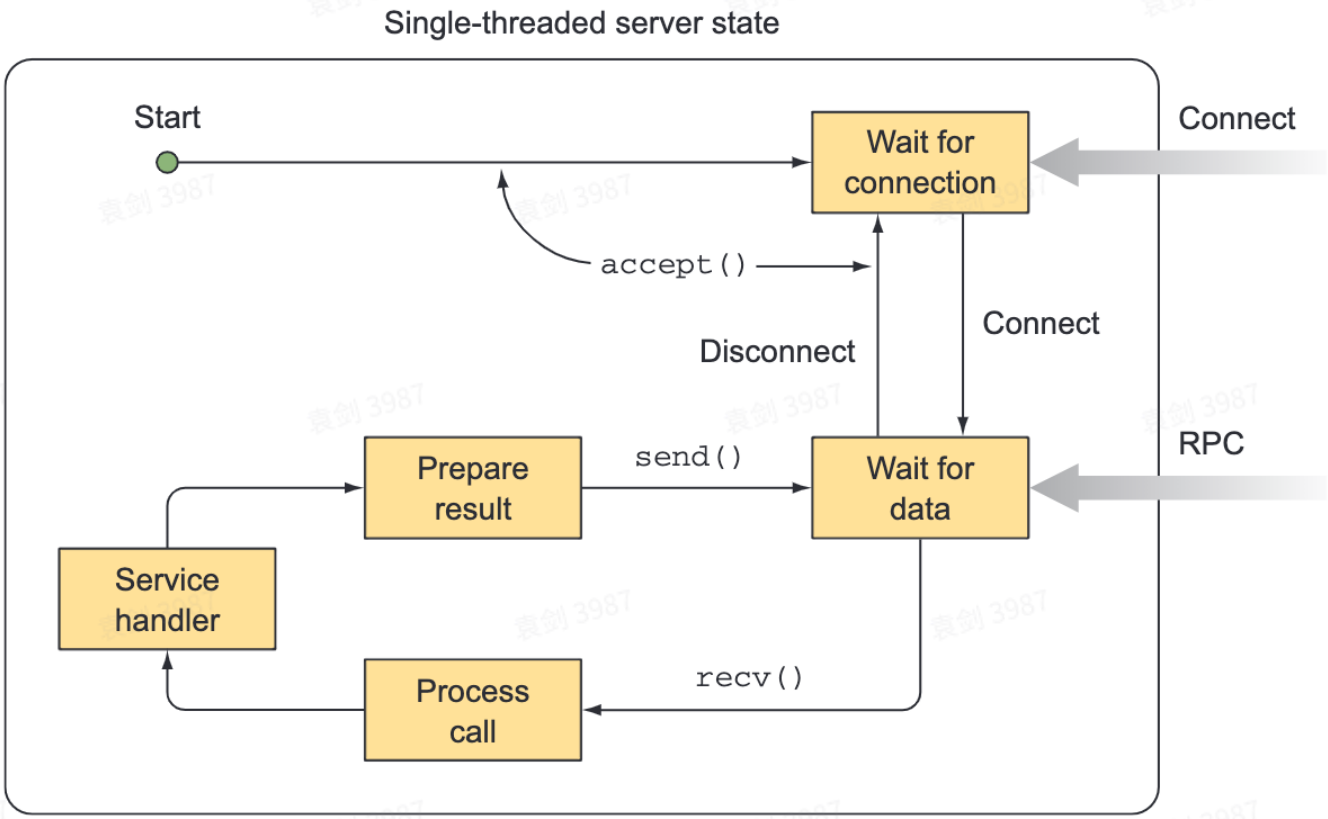


Figure 10.1 The Apache Thrift framework server library

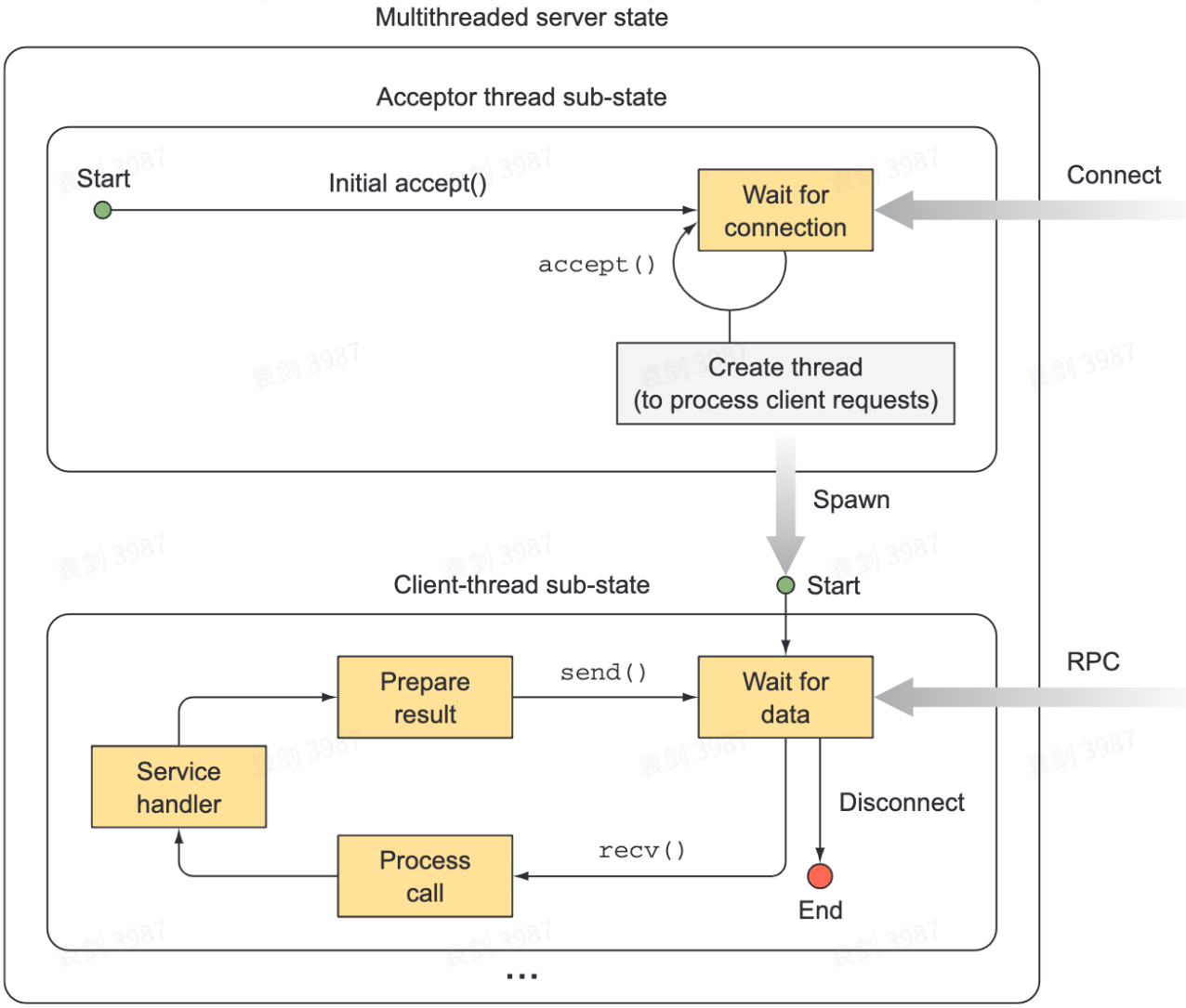
最简单的 server 模型：单进程单线程。



基于连接的，单线程 server 状态图

这种情况下不是在等待消息就是在等待连接。

基于连接的多线程模型：



基于连接的，多线程 server 状态图

Server 并发模型

基于连接：

模型简单，但是当有大量的连接被建立时，线程的开销是巨大的，同时有大量的线程会被空闲占用大量的内存，可能导致数据抖动（磁盘和内存频繁进行数据交换）。

基于任务：

模型更复杂，资源利用率更高，但是无法保证多个调用结果的返回顺序，需要增加额外的机制。但是在具体实现中，难以处理多个线程从同一个连接去等待读取的情况。

混合模型：

I / O 线程负责建立连接，task 线程负责处理调用。

TFramedTransport 和 基于任务的并发模型的联动：

Frame 可以让 I / O 线程在无需反序列化的情况下得到帧的大小，从而确定一条 RPC message，然后分配给 task thread 去执行调用。好处在于 I/O 线程能够快速、完整地处理入站请求，并且独立于所涉

及的服务。

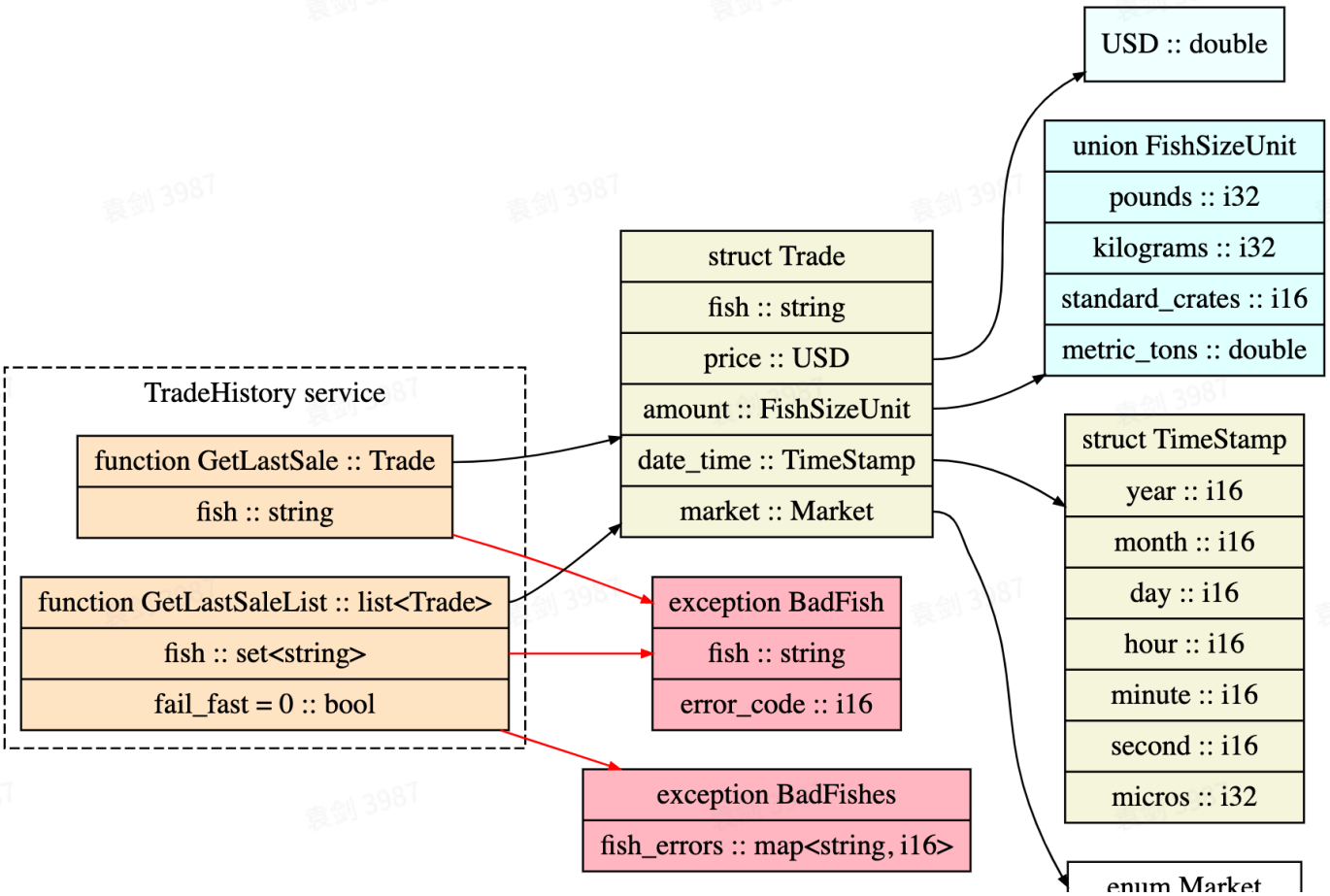
IDL 可视化工具

HTML

```Bash
thrift -gen html fish_trade.thrift
```

Graphviz

```Bash
thrift -gen gv:exceptions fish_trade.thrift
```


USD :: double
union FishSizeUnit
pounds :: i32
kilograms :: i32
standard_crates :: i16
metric_tons :: double
struct Trade
fish :: string
price :: USD
amount :: FishSizeUnit
date_time :: TimeStamp
market :: Market
struct TimeStamp
year :: i16
month :: i16
day :: i16
hour :: i16
minute :: i16
second :: i16
micros :: i32
enum Market
TradeHistory service
function GetLastSale :: Trade
fish :: string
function GetLastSaleList :: list<Trade>
fish :: set<string>
fail_fast = 0 :: bool
exception BadFish
fish :: string
error_code :: i16
exception BadFishes
fish_errors :: map<string, i16>