讲师介绍--专业来自专注和实力

Darren老师

曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

重点内容

1 rpc-client
2 rpc-server

1 rpc-client概要设计

tar-rpc-client主要由4个组件构成：ServantProxy，ObjectProxy，CommunicatorEpoll，AsyncProcThread，其中：

- ServantProxy：直接与使用者交互，提供简便易用接口
- ObjectProxy：封装网络层收发细节
- CommunicatorEpoll：提供收发调度功能
- AsyncProcThread：提供异步调用功能

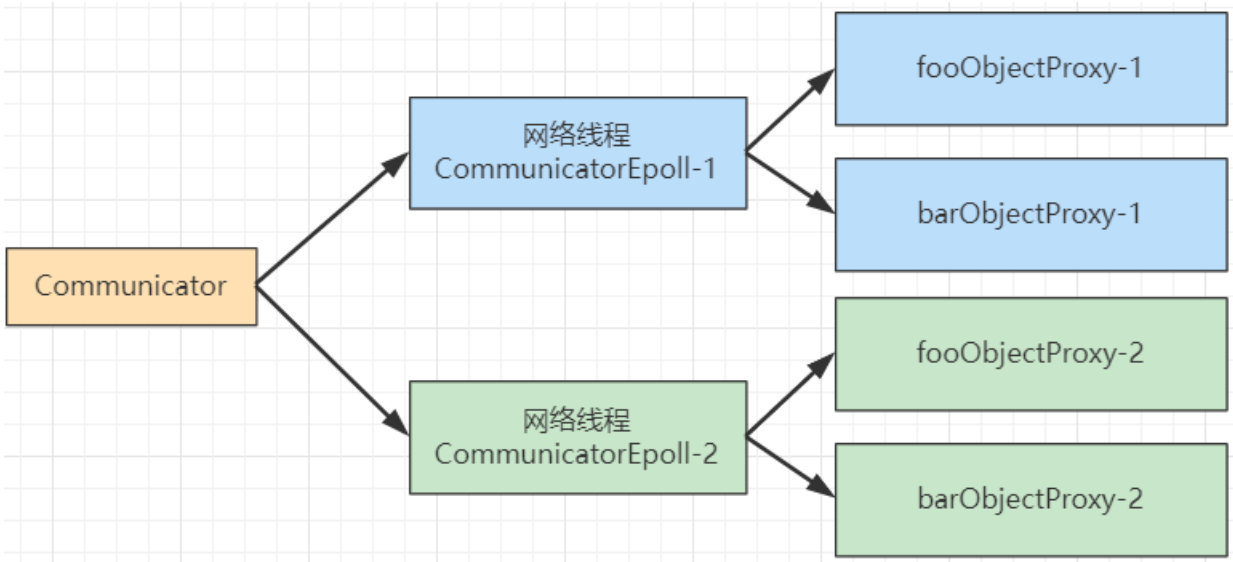
1.1 Communicator

TARS的客户端最重要的类是Communicator，一个客户端只能声明出一个Communicator类实例。

Communicator类聚合了两个重要的类：

1. CommunicatorEpoll，负责网络线程的建立与通过ObjectProxyFactory生成ObjectProxy；
2. ServantProxyFactory，生成不同的RPC服务句柄，即ServantProxy，用户通过ServantProxy调用RPC服务。

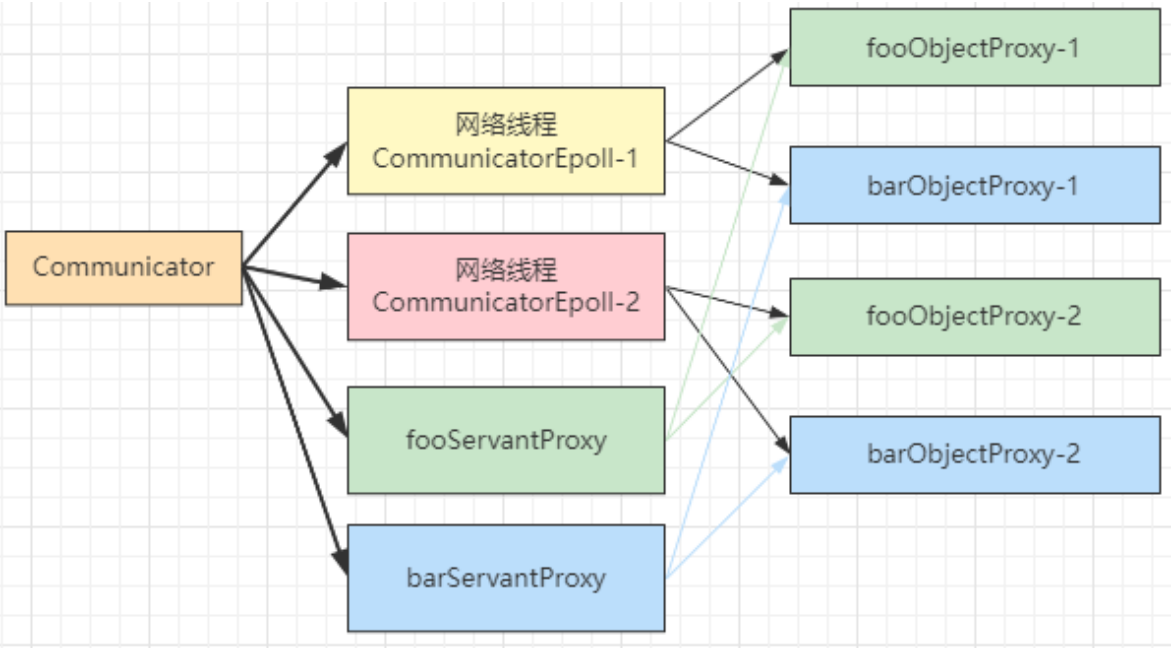
1.2 CommunicatorEpoll和ServantProxyFactory



Communicator拥有n个网络线程，即n个**CommunicatorEpoll**。每个CommunicatorEpoll拥有一个ObjectProxyFactory类，每个ObjectProxyFactory可以生成一系列的不同服务的实体对象ObjectProxy，因此，假如Communicator拥有两个CommunicatorEpoll，**并有foo与bar这两类不同的服务实体对象。**

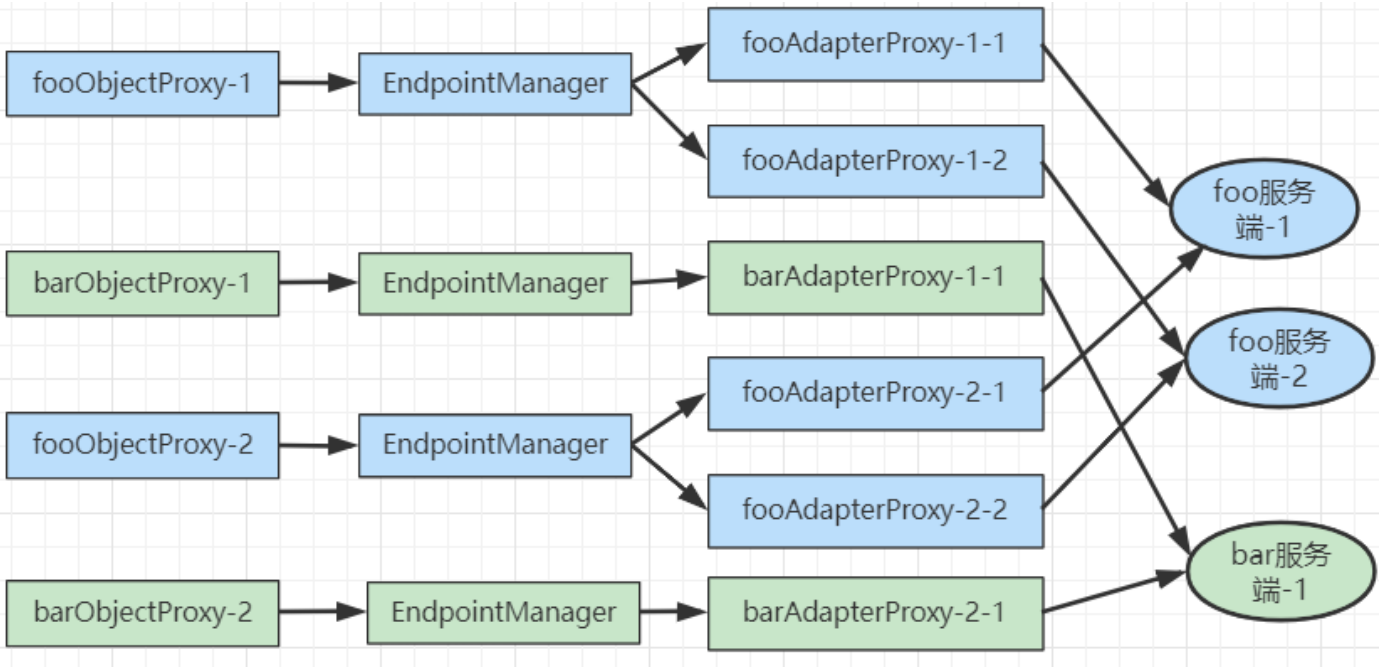
每个CommunicatorEpoll可以通过 ObjectProxyFactory创建两类ObjectProxy，这是TARS客户端的第一层负载均衡，每个线程都可以分担所有服务的RPC请求，因此，一个服务的阻塞可能会影响其他服务，因为网络线程是多个服务实体ObjectProxy所共享的。

1.3 EndpointManager管理



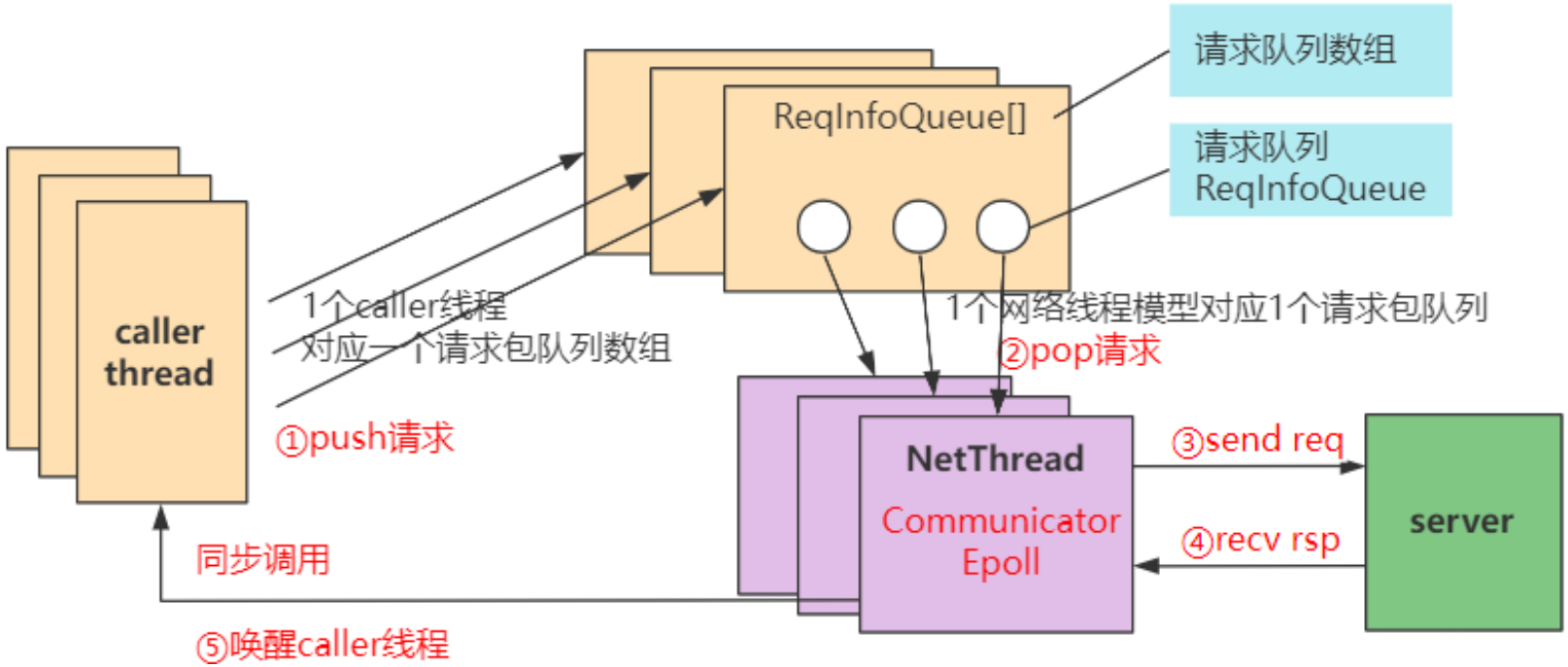
Communicator实例通过ServantProxyFactory成员变量的getServantProxy()接口在构造fooServantProxy句柄的时候，会获取Communicator实例下的所有CommunicatorEpoll（即CommunicatorEpoll-1与CommunicatorEpoll-2）中的fooObjectProxy（即fooObjectProxy-1与fooObjectProxy-2)，并作为构造fooServantProxy的参数。Communicator通过ServantProxyFactory能够获取foo与bar这两类ServantProxy，ServantProxy与相应的ObjectProxy存在相应的聚合关系。

1.3 EndpointManager管理



每个ObjectProxy都拥有一个EndpointManager，例如，fooObjectProxy 的EndpointManager管理fooObjectProxy 下面的所有fooAdapterProxy，每个AdapterProxy连接到一个提供相应foo服务的服务端物理机socket上。通过EndpointManager还可以以不同的负载均衡方式获取连接AdapterProxy。对于同一RPC服务，选取不同的ObjectProxy（或可认为选取不同的网络线程CommunicatorEpoll）是第一层的负载均衡，而对于同一个被选中的ObjectProxy，通过EndpointManager选择不同的socket连接AdapterProxy是第二层的负载均衡。

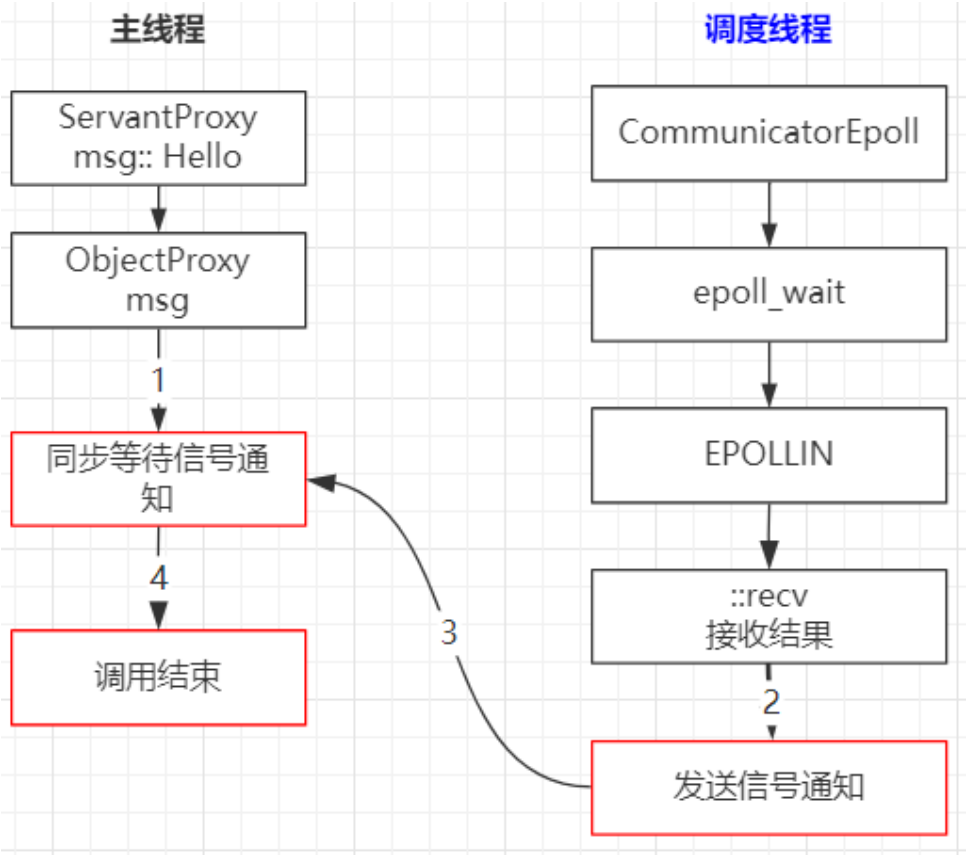
1.4 客户端同步调用逻辑



每条caller线程与每条客户端网络线程CommunicatorEpoll进行信息交互的桥梁——通**信队列ReqInfoQueue数组**，数组中的每个ReqInfoQueue元素负责与一条网络线程进行交互。

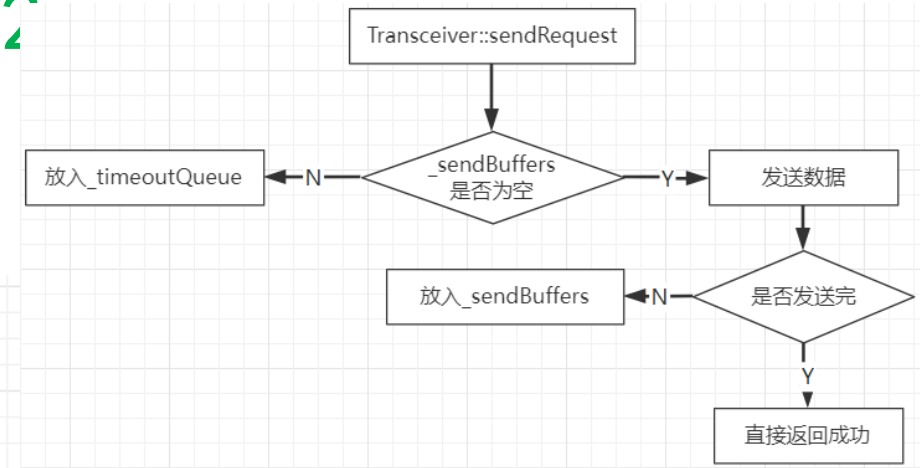
生产者Caller线程向自己的线程私有数据**ReqInfoQueue[]**中的**第N个元素**ReqInfoQueue[N] push请求包，消费者客户端**第N个网络线程**就会从这个队列中pop请求包。

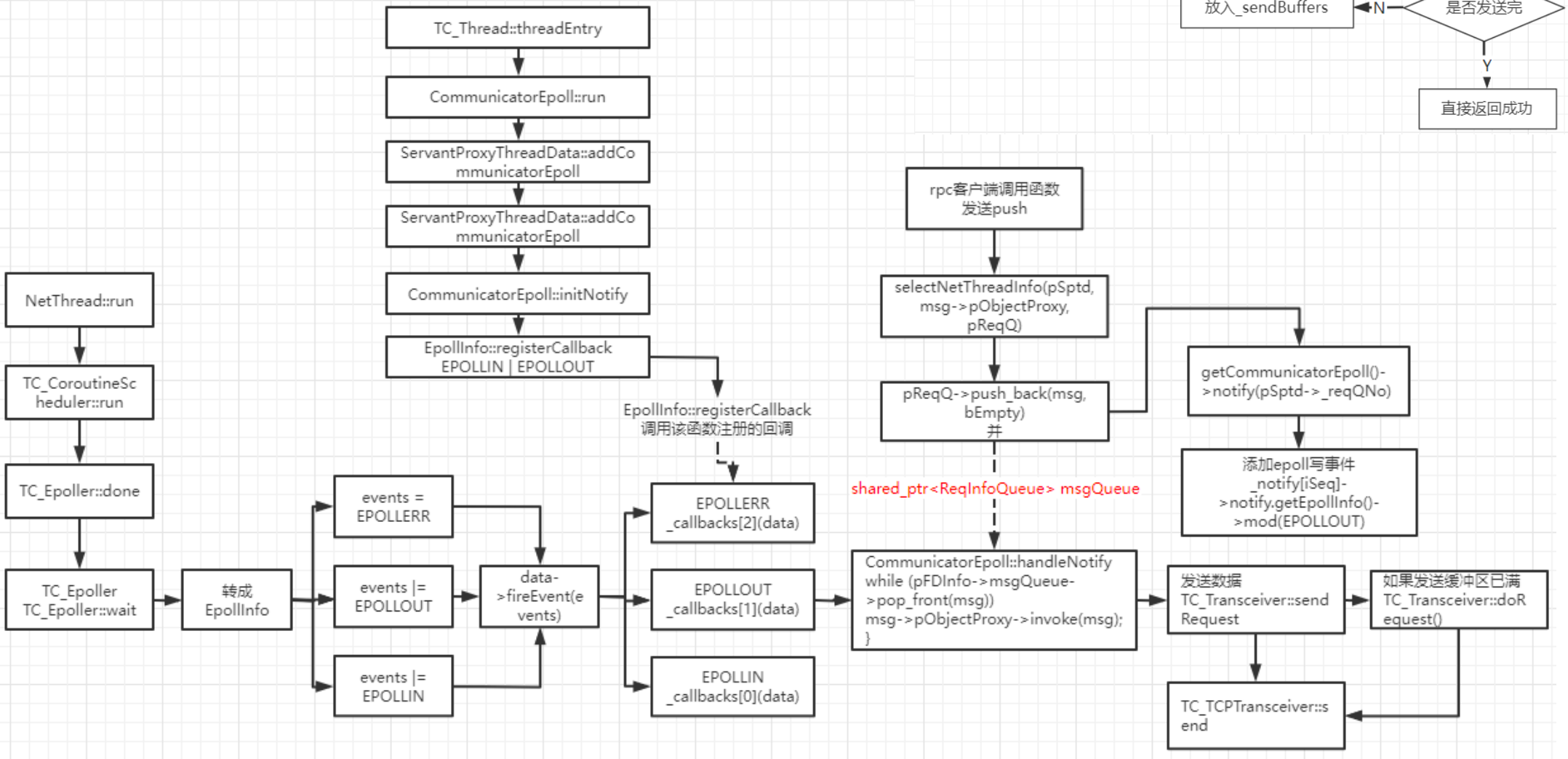
1.5 客户端同步发送详细逻辑1



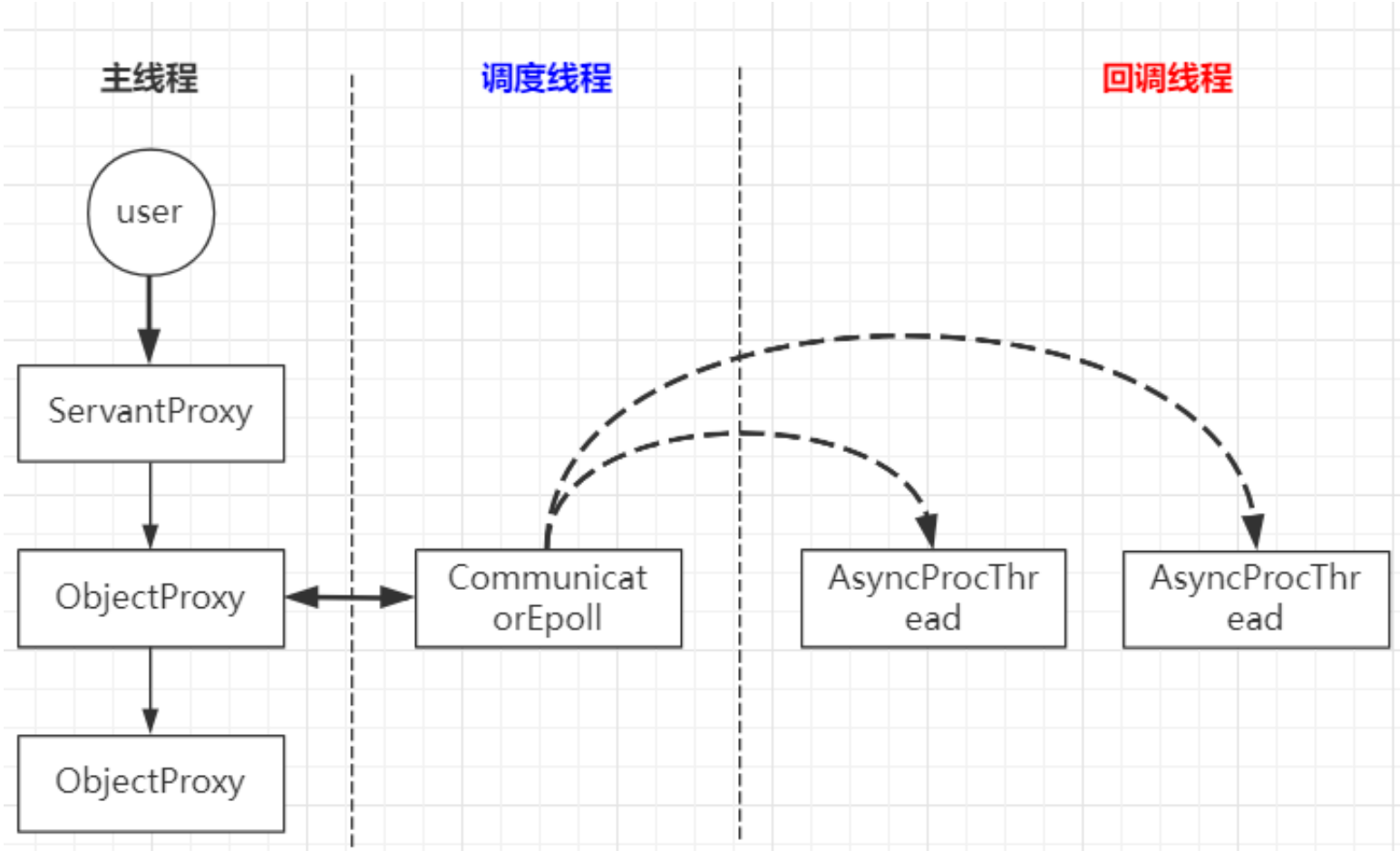
1.主线程调用函数后阻塞等待调度线程的信号通知，
2.调度线程收到结果后，主备发送信号通知
3.发送信号通知
4.主线程接收到信号后，本次调用结束

1.5 客户端同步发送详细逻辑2

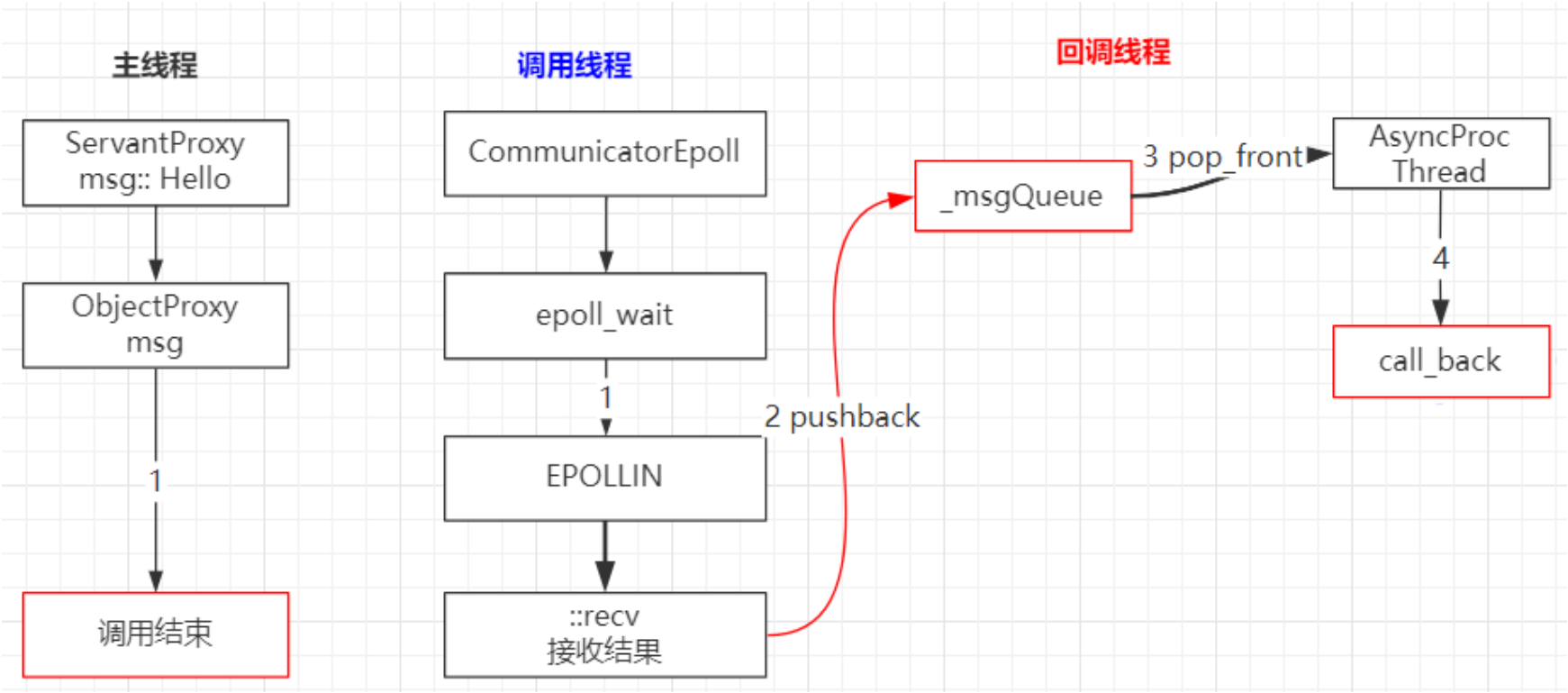




1.6 异步回调处理线程1



1.6 异步回调处理线程2



1.主线程调用完方法后直接结束
2.调度线程接收到结果后，放入回调线程的队列_msgQueue中
3.回调线程循环等待_msgQueue中的msg，当有msg进入时，会使用pop_front取出
4.调用回调函数处理msg

2 RPC server概要设计1

Application 代表一个应用
TC_EpollServer 管理服务
NetThread 网络线程
BindAdapter服务端口管理,监听socket信息，绑定对应的业务
Handle 处理线程的handle封装

RecvContext接收包的上下文
SendContext发送包的上下文
DataBuffer数据队列包装
Connection服务连接管理
ConnectionList带有时间链表的map支持自动删除超时的连接

EpollInfo 用来管理fd的收发等事件
NotifyInfo 通知epoll从wait中醒过来
TC_Epoller Epoll封装

每个类的大致内容：https://www.processon.com/view/link/624e8deb0791290727b50675

2 RPC server概要设计2

- Application：一个服务端就是一个Application，Application帮助用户读取配置文件，根据配置文件初始化代理（假如这个服务端需要调用其他服务，那么就需要初始化代理了）与服务，新建以及启动网络线程与业务线程。
- TC_EpollServer：是真正的服务端，如果把Application比作风扇，那么TC_EpollServer就是那个马达。TC_EpollServer掌管两大模块——网络模块NetThread与业务模块Handle 。
- NetThread：代表着网络模块，内含TC_Epoller作为IO复用，TC_Socket建立socket连接，ConnectionList记录众多对客户端的socket连接。任何与网络相关的数据收发都与NetThread有关。
- HandleGroup与Handle：代表着业务模块，Handle是执行PRC服务的一个线程，而众多Handle组成的HandleGroup就是同一个RPC服务的一组业务线程了。
- BindAdapter：BindAdapter本身可以认为是一个服务的实例，能建立真实存在的监听socket并对外服务，与网络模块NetThread以及业务模块HandleGroup都有关联

threads=5　处理HelloObj的线程池，是5个线

2.1 Servant和Adapter

```
-p 22785
```

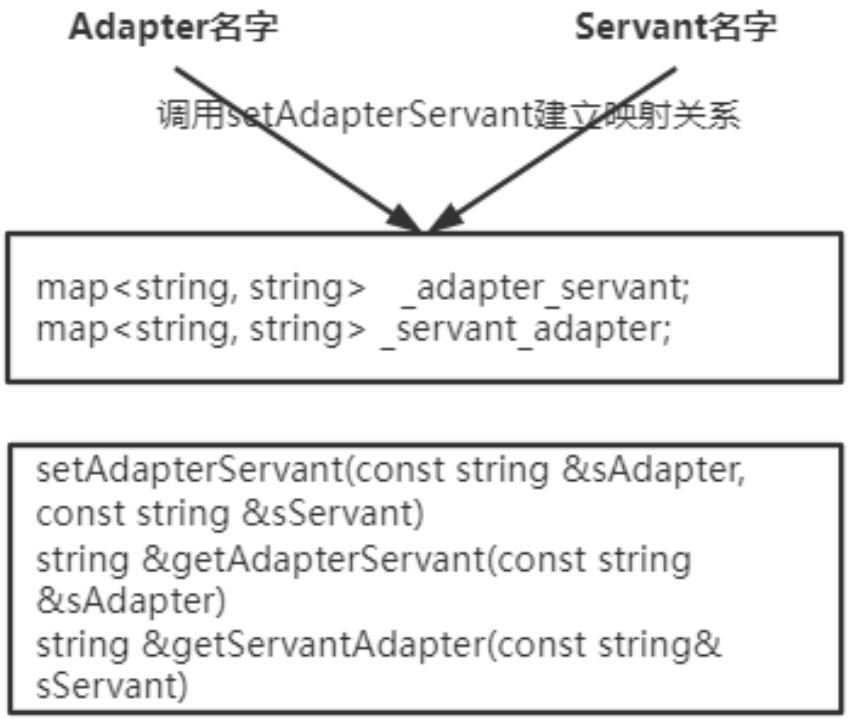
- tars节点的名字由三级组成:App.Server.Servant，在web上部署的时候也是一个Servant对应一个ip:port;
- 但实际上Servant并没有和ip:port直接绑定，而是由Adapter来管理ip:port，Servant再和Adapter进行一一映射。 HelloObj

```
./tars/tarsnode/data/TestApp.HelloServer/conf/TestApp.HelloServer.config.conf:27:      <TestApp.HelloServer.HelloObjAdapter>
```

```
<TestApp servant=TestApp.HelloServer.HelloObj
  allow
  endpoint=tcp -h 192.168.0.143 -p 22785 -t 60000
  maxconns=100000
  protocol=tars
  queuecap=50000
  queuetimeout=20000
  servant=TestApp.HelloServer.HelloObj
  threads=5
</TestApp.HelloServer.HelloObjAdapter>
```

2.1 Servant和Adapter

每个Servant和Adapter都有自己的名字,在框架进行初始化时，会调用setAdapterServant()把Servant和Adapter的名字映射起来



类ServantHelperManager

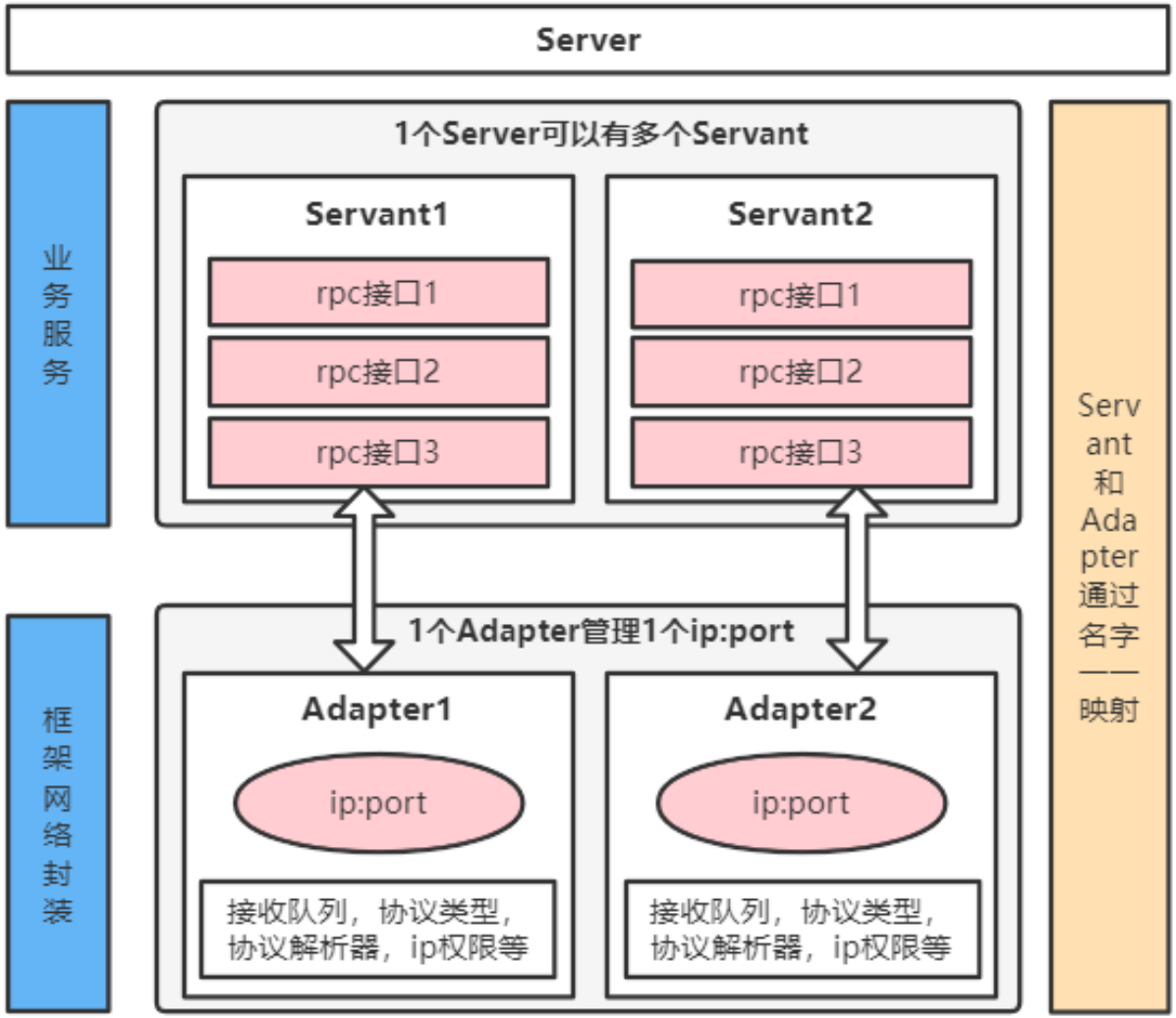
```
/* servant for server */
class Hello : public tars::Servant
```

- Adapter和Servant的映射关系保存在全局单列类ServantHelperManager中
- 调用setAdapterServant把两者的名字存在map中
- 用Adapter的名字调用getAdapterServant可以获取到对应的Servant名字
- 用Servant的名字调用getServantAdapter可以获取到对应的Adapter名字

2.1 Servant和Adapter

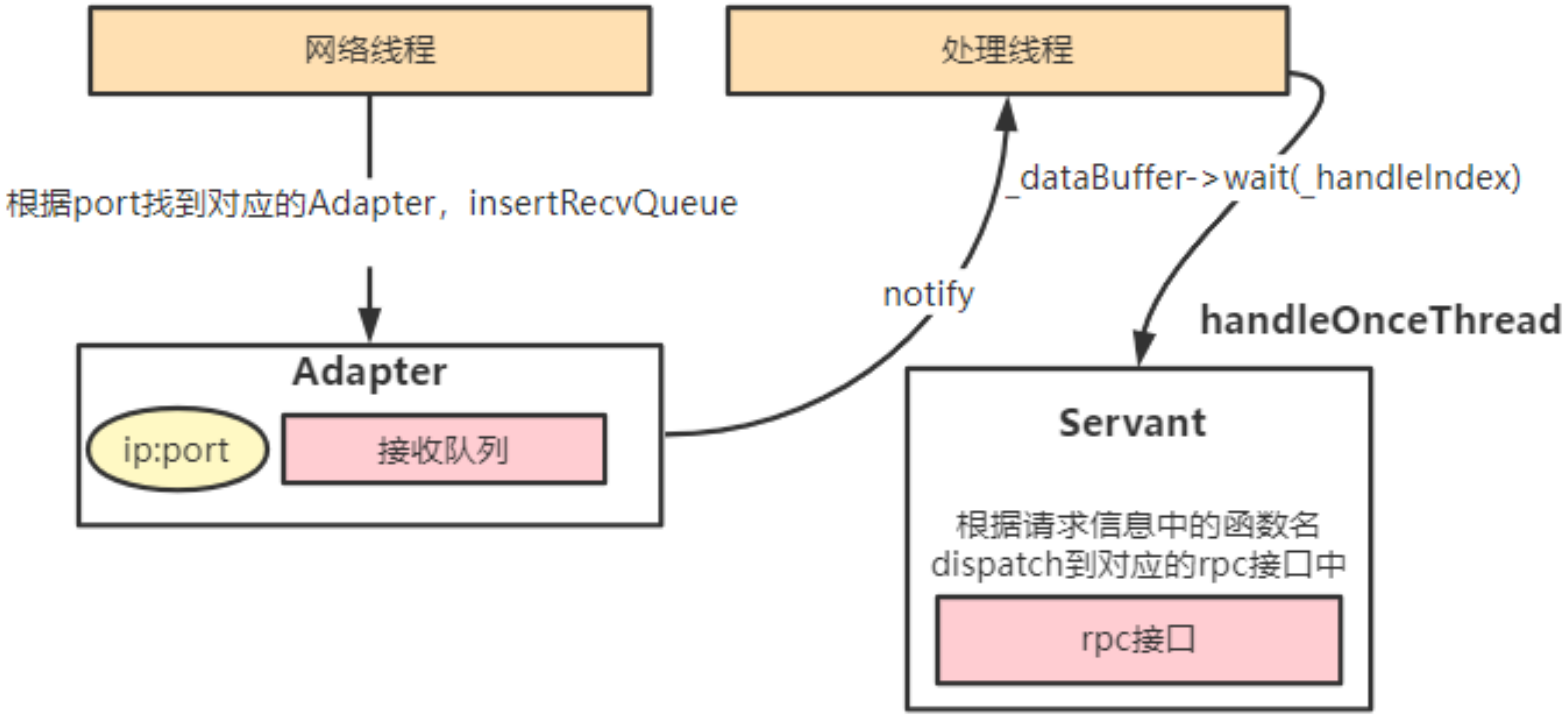
每个Adapter管理一个端口，同时网络线程和业务线程都是直接与Adapter进行交互：

- Adapter是端口在框架网络层的封装；
- Servant是该端口提供的业务逻辑的集合：



2.1 Servant和Adapter

每个adapter有属于自己的接收队列，在网络线程收到请求后，找到对应的Adapter，把包push到Adapter的接收队列中，然后业务线程再把包从adapter的接收队列中取出来分发到servant中进行处理：




```
in tars::TC_EpollServer::Handle::handleOnceThread (this=0xc57db0)
rsFramework-v3.0.5/tarscpp/util/src/tc_epoll_server.cpp:356
in tars::TC_EpollServer::Handle::handleLoopThread (this=0xc57db0)
rsFramework-v3.0.5/tarscpp/util/src/tc_epoll_server.cpp:438
in std::__invoke_impl<void, void (tars::TC_EpollServer::Handle::*
```

2.2 服务端线程划分

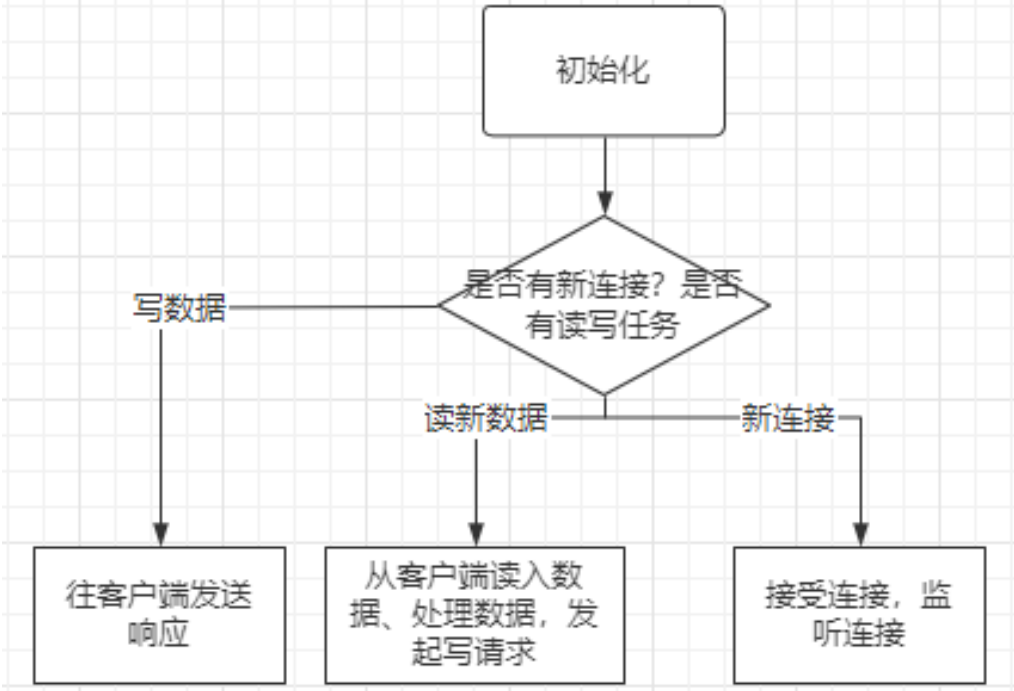
服务端的工作线程分为两类->网络线程与业务线程:
1. 网络线程负责接受客户端的连接与收发数据
2. 业务线程只关注执行用户所定义的PRC方法

两种线程在初始化的时候执行start()启动。

大部分服务器都是按照accept()->read()->write()->close()的流程执行的

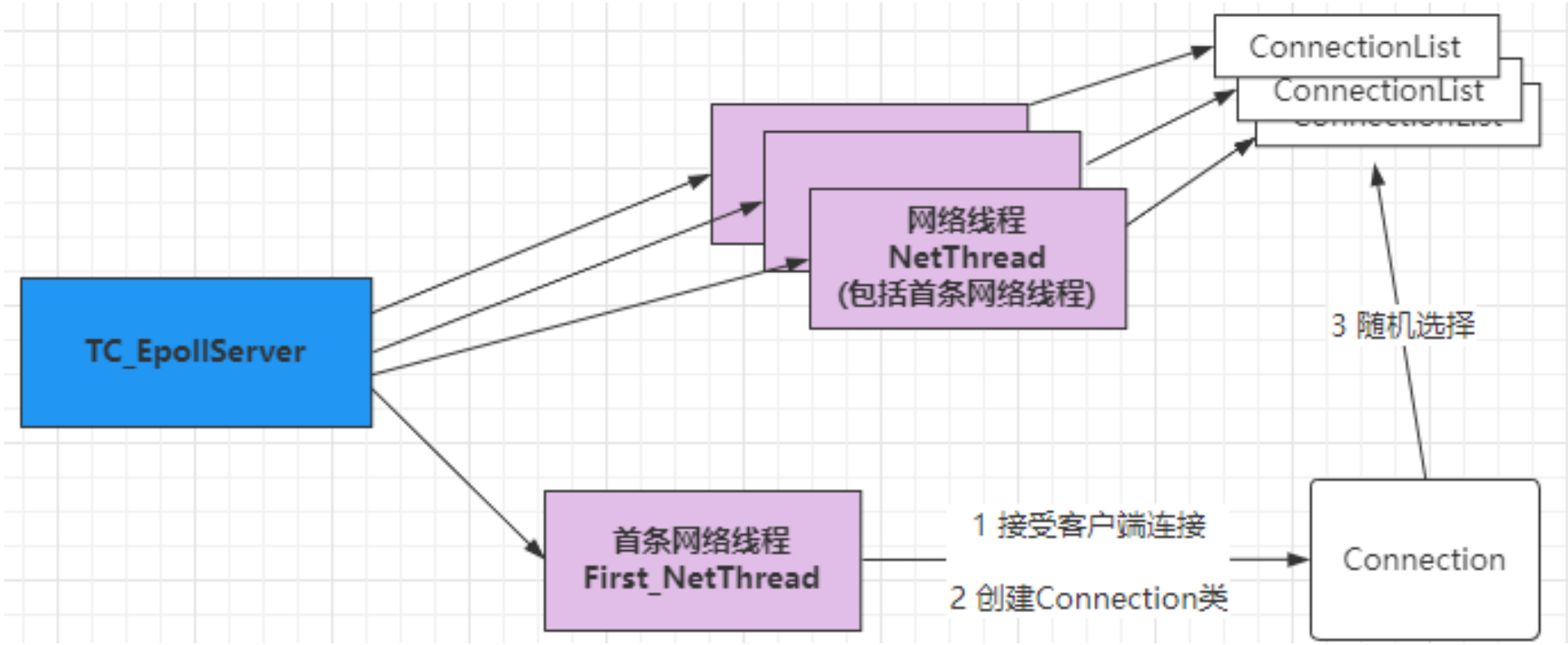
2.3 服务器收发大体流程

大部分服务器都是按照accept()->read()->write()->close()的流程执行的

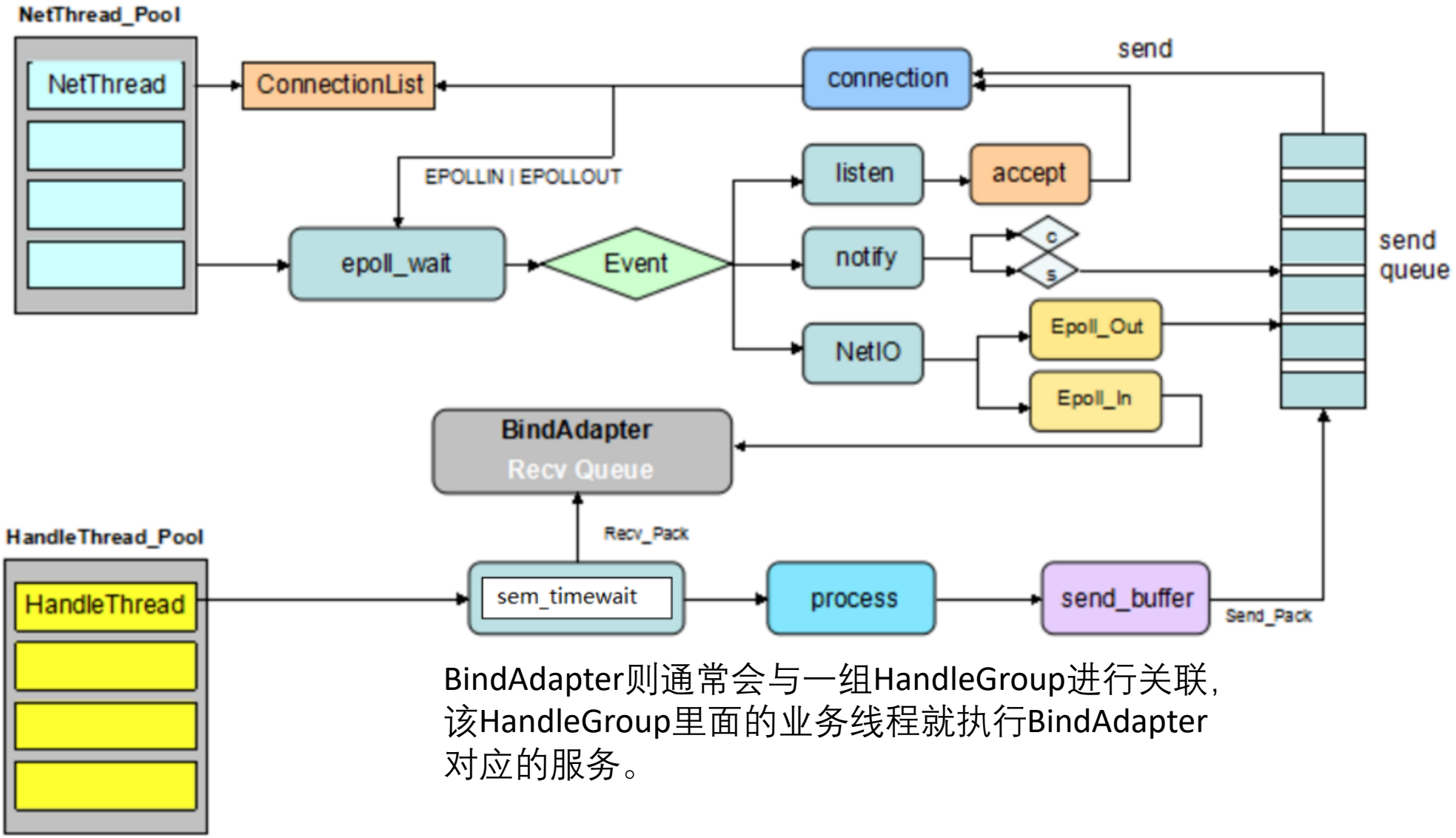


- 判定逻辑采用Epoll IO复用模型实现，每一条网络线程NetThread都有一个TC_Epoller来做事件的收集、侦听、分发。
- **只有第一条网络线程会执行连接的监听工作**，接受新的连接之后，就会构造一个Connection实例，并选择处理这个连接的网络线程。
- 请求被读入后，将暂存在接收队列中，并通知业务线程进行处理，业务处理完后，将结果放到发送队列。
- 发送队列有数据，需要通知网络线程进行发送，接收到发送通知的网络线程会将响应发往客户端。

2. 4服务端接受一个客户端连接

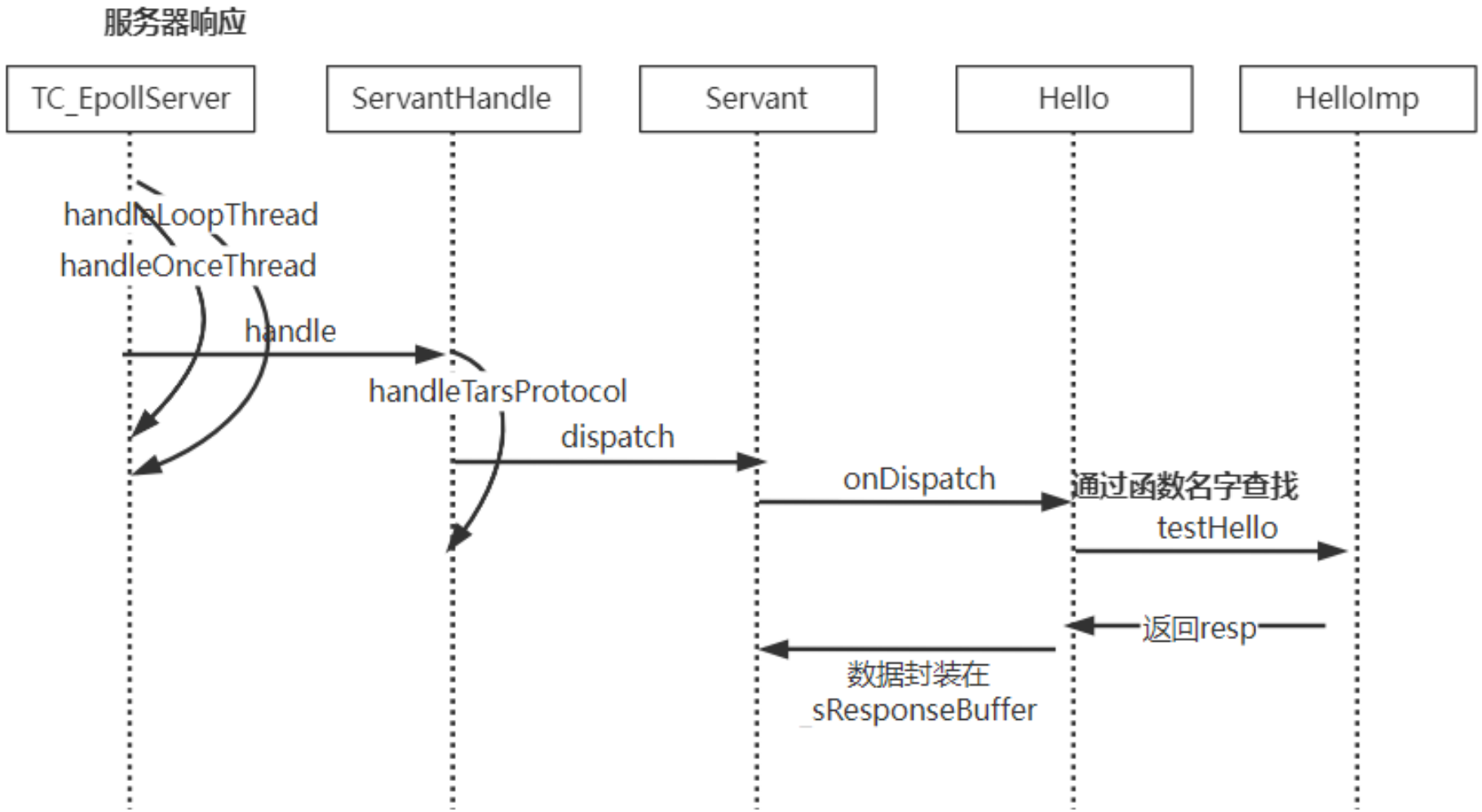


2.5 数据收发处理逻辑



BindAdapter则通常会与一组HandleGroup进行关联，该HandleGroup里面的业务线程就执行BindAdapter对应的服务。

2.6 以Hello为例-服务端-响应



TestApp::Hello::onDispatch(tars::TC_AutoPtr<tars::Current>, std::vector<char, std::allocator<char> >&)

3 参考

主要来自于Tars官方PPT、文章和范例