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. Tars线程网络服务框架
2. Tars servant分析
3. Tars 常用组件分析

0 文档

官方文档：
https://newdoc.tarsyun.com/#/markdown/TarsCloud/TarsDocs/README.md

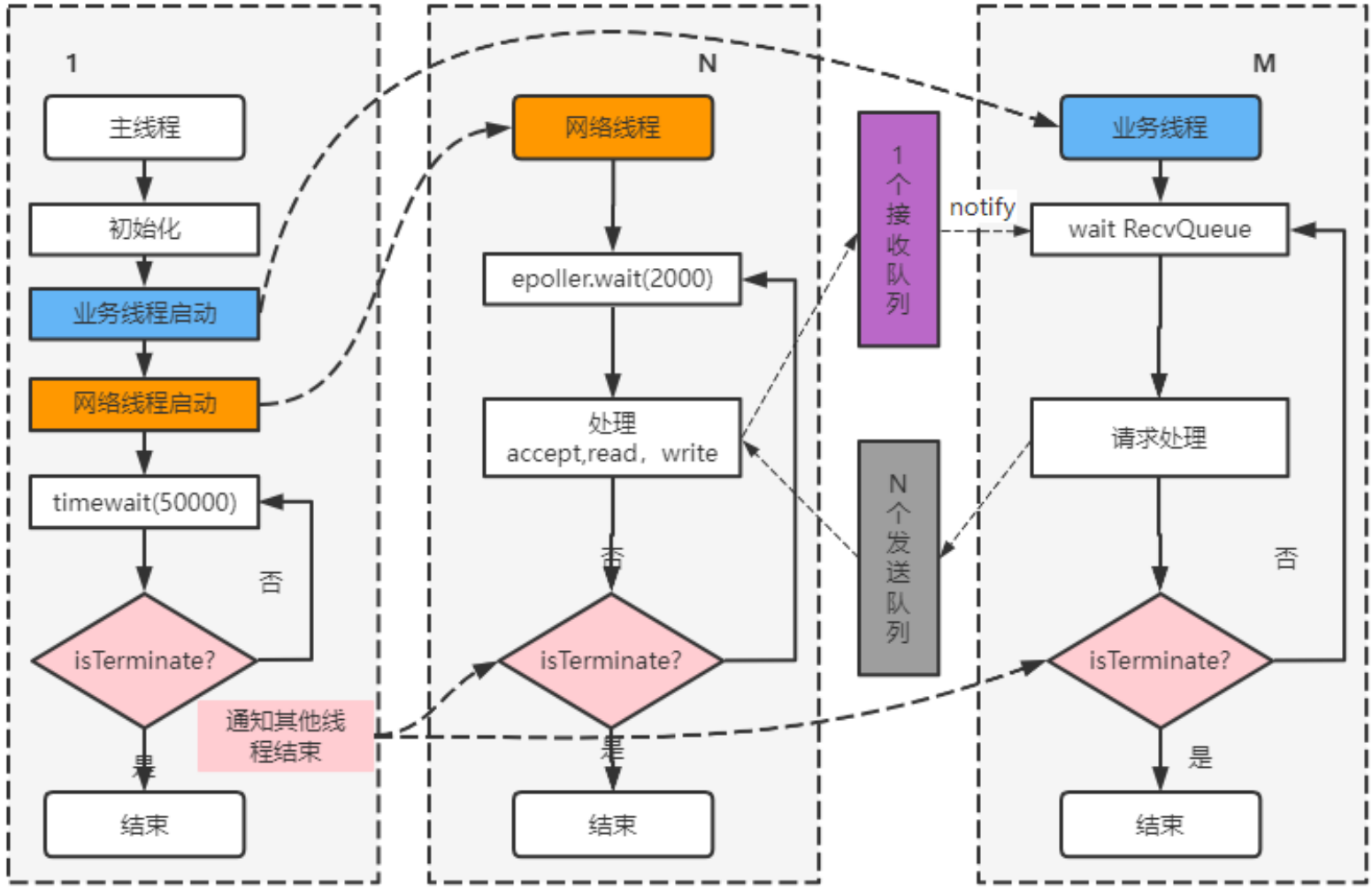
Github文档：https://tarscloud.github.io/TarsDocs/SUMMARY.html

Tars基金会文章： https://segmentfault.com/u/tarsfoundation

1 框架封装分析

- 线程模型创建：主线程，网络线程，业务线程
- 网络io模型创建
- 业务模型创建:Adapter,Servant的管理
- 通信器创建
- 其他的：~~异常信息，统计信息上报，ssl，协程，日志，链路跟踪，消息染色等~~

1.1 线程模



- 1个主线程：框架的初始化,业务层初始化，servant和adapter的管理，epoll模型的创建,业务线程创建,网络线程创建。
- N个网络线程：调度epoll处理网络事件，socket的accept，read，write，请求包push到接收队列让业务线程处理。
- M个业务线程：解析请求包，分发到业务层进行处理。
- 其他的线程，例如通信器的线程

1.1 线程模型

- 蓝色部分，主线程启动了业务线程。业务线程的数量可以针对不同的servant设置不同的线程数。之后业务线程在一个循环逻辑中，每次从队列中取出请求进行处理。
- 黄色部分，网络线程启动。框架限制了网络线程的数量最大为15。网络线程同样把收到的请求push到队列中，供业务线程处理。
- 红色部分，当进程被设置为terminate的时候，主线程结束，同时网络线程和业务线程也把terminate设置为true，线程结束。紫色的接收队列，网络线程收到请求后push到队列中，业务线程通过notify -> wait 取出请求进行处理。接收队列只有1个，属于adapter的。
- 灰色的N个发送队列，**每个网络线程都有1个发送队列**，当业务线程向客户端回包的时候，会push到发送队列中，网络线程把发送包取出来发送到对应的客户端。
- 另外还有一点需要注意的，不管是网络线程还是业务线程，都是基于一个循环去处理事件和消息，所以一般不能出现耗时比较久的操作(例如长时间的阻塞)。

1.2 Servant和Adapter

```
-p 22785
```

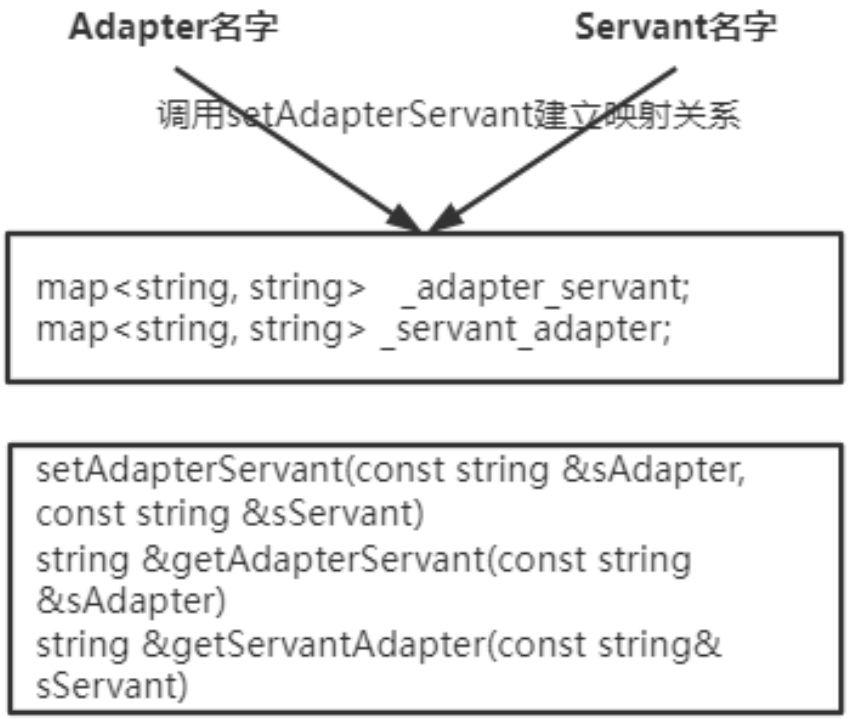
- tars节点的名字由三级组成:App.Server.Servant，在web上部署的时候也是一个Servant对应一个ip:port;
- 但实际上Servant并没有和ip:port直接绑定，而是由Adapter来管理ip:port，Servant再和Adapter进行一一映射。 HelloObj

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

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

1.2 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



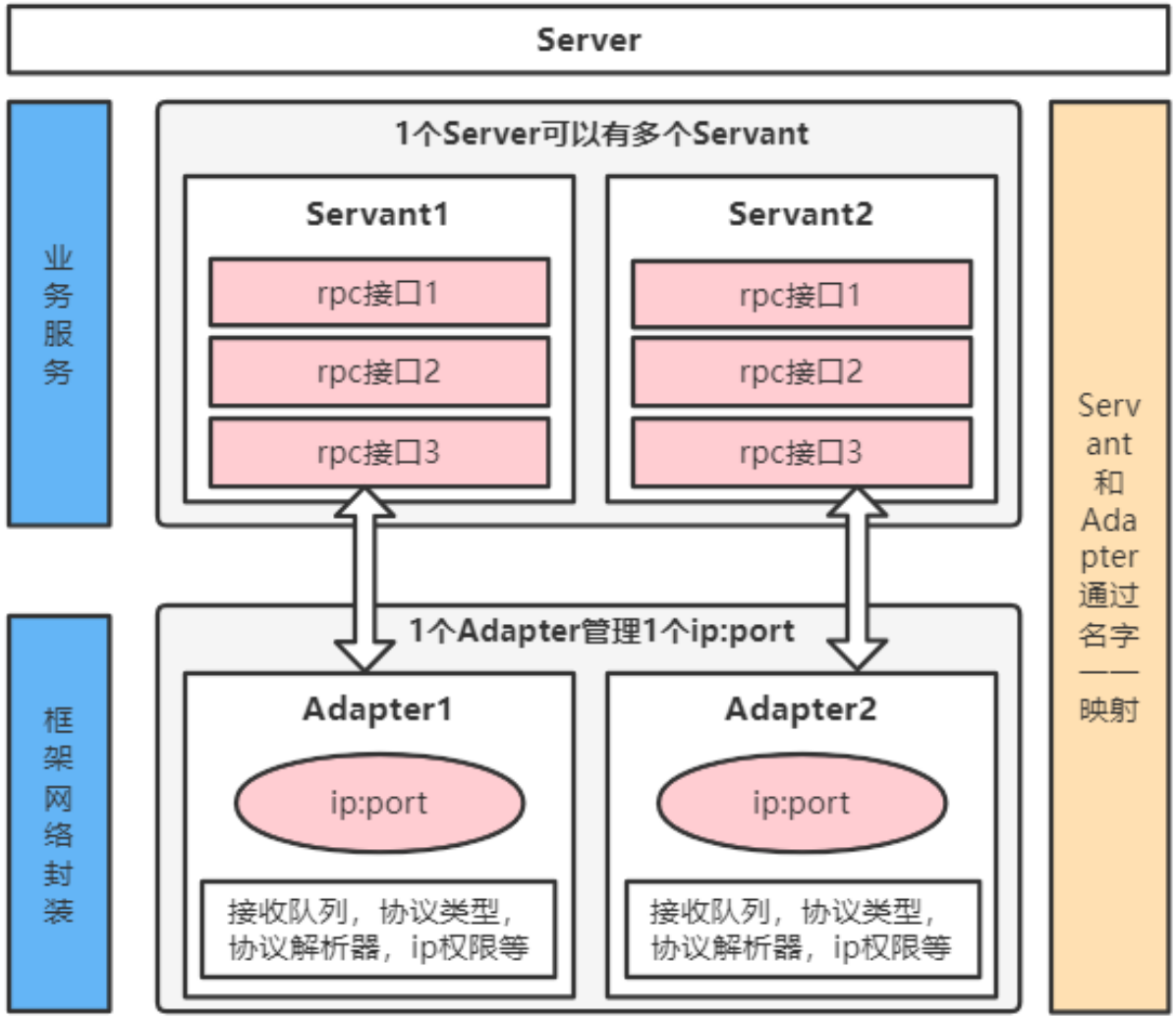
类ServantHelperManager

- 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

1.2 Servant和Adapter

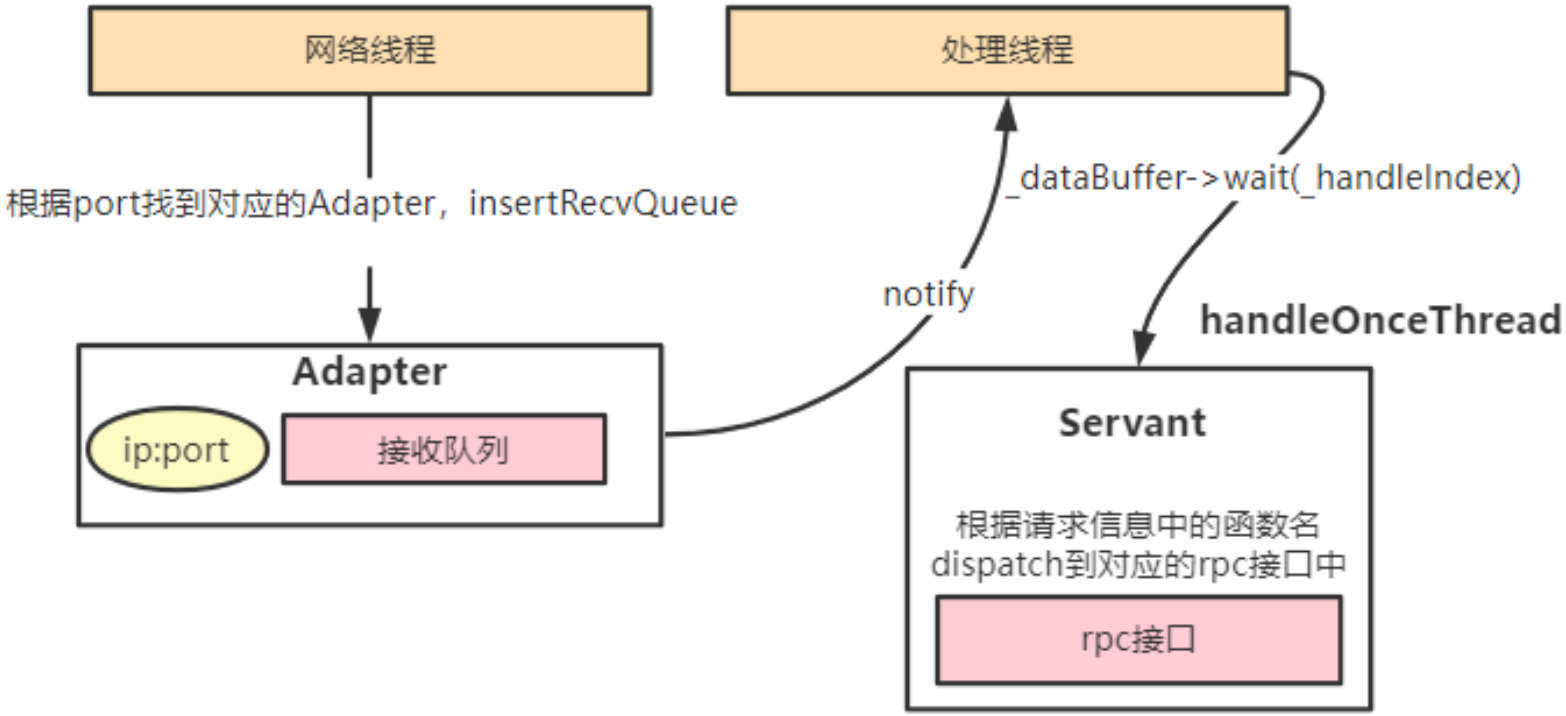
每个Adapter管理一个端口，同时网络线程和业务线程都是直接与Adapter进行交互：

- 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：



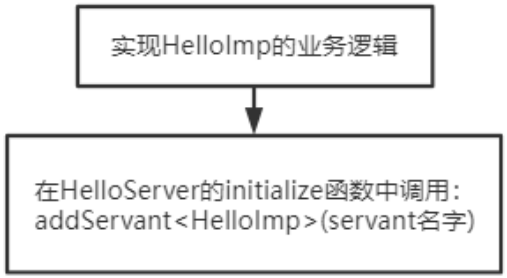
1.2 Servant和Adapter

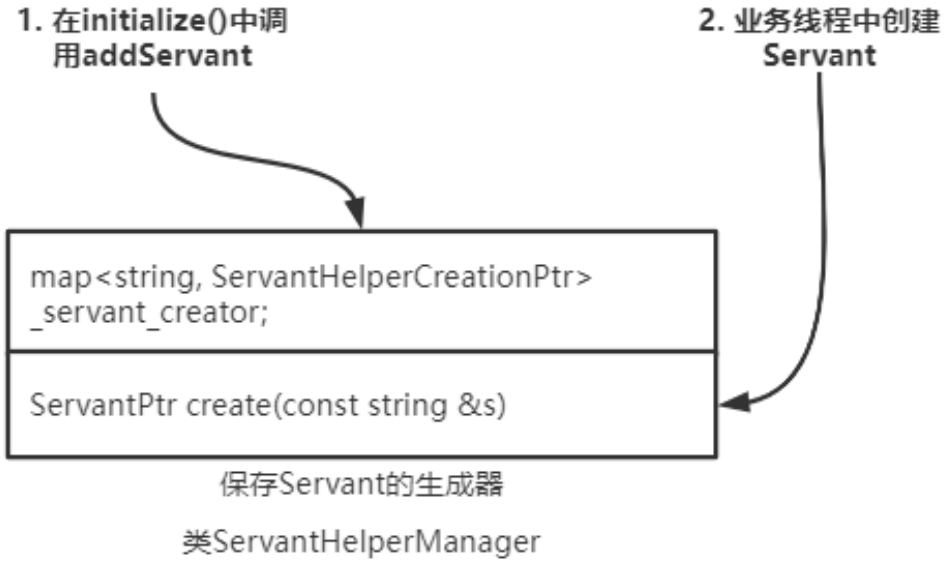
每个adapter有属于自己的接收队列，在网络线程收到请求后，找到对应的Adapter，把包push到Adapter的接收队列中，然后业务线程再把包从adapter的接收队列中取出来分发到servant中进行处理：

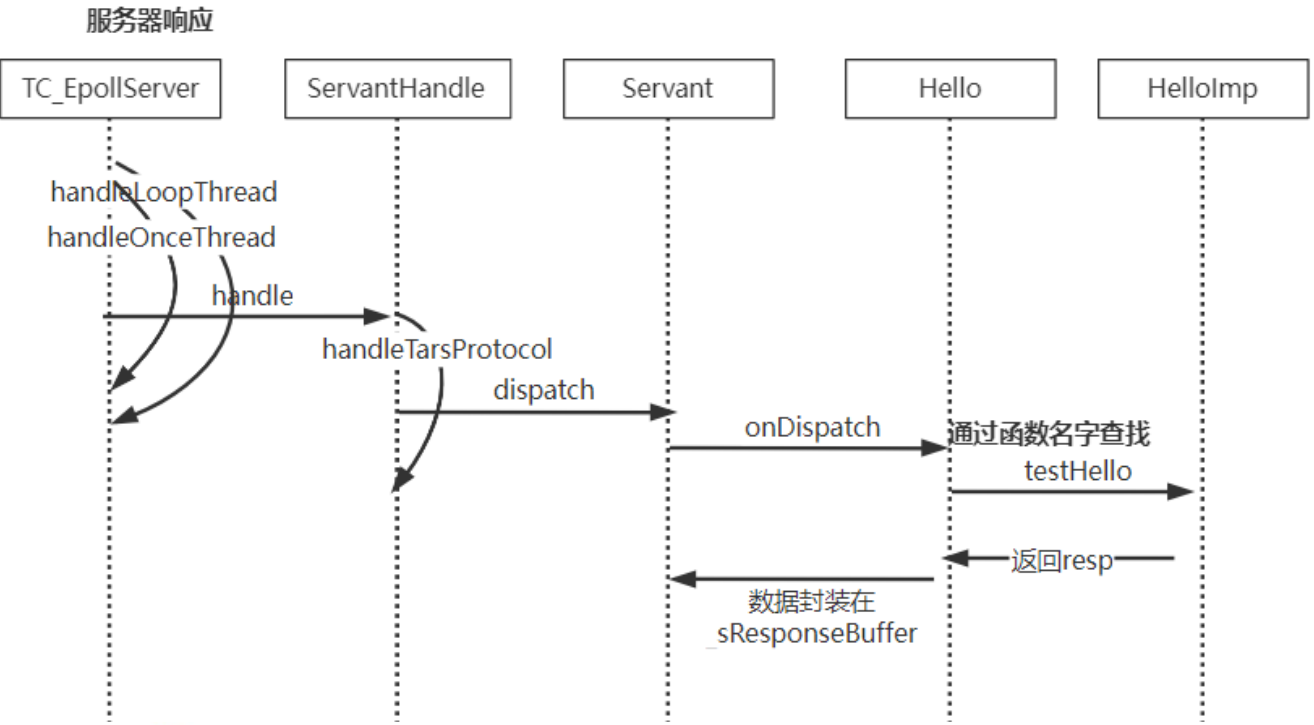



```
in tars::TC_EpollServer::Handle::handleOnceThread (this=0xc57db0)
rsFramework-v3.0.5/tarscpp/util/src/tc_epoll_server.cpp:356
in tars::TC_EpollServer::Handle::handleLoopThread (this=0xc57db0)
rsFramework-v3.0.5/tarscpp/util/src/tc_epoll_server.cpp:438
in std::__invoke_impl<void, void (tars::TC_EpollServer::Handle::*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

1.3 基于HelloServer服务的框架分析





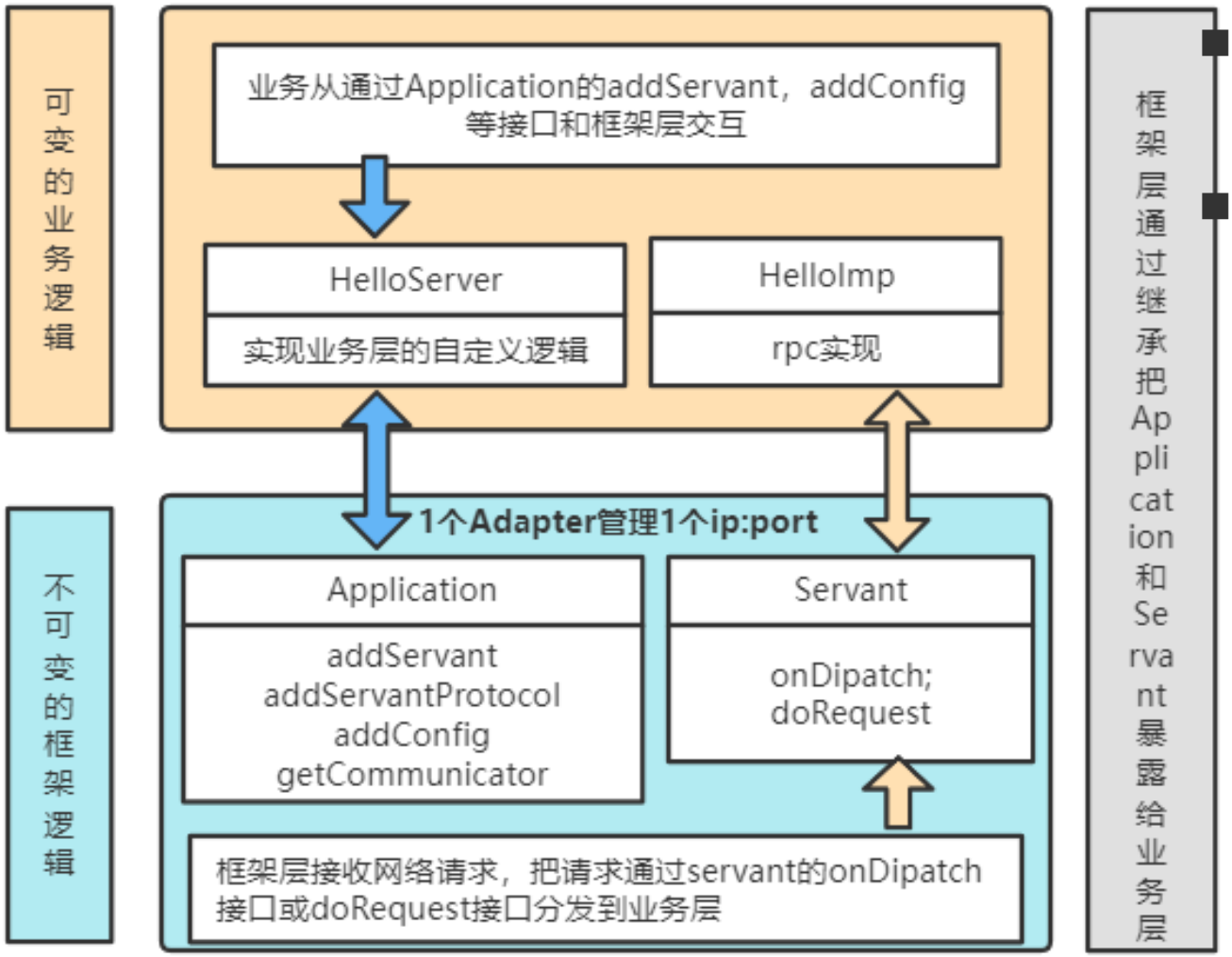


1.3 servant的可拔插机制

框架借助ServantHelperManager这个全局单列类实现了servant的可插拔注册，又在运行时动态创建servant实例，通过servant管理所有可变的业务逻辑。总结起来，框架为我们封装了线程模型，epoll模型,网络通信细节和协议解析等不变的部分；而变化的部分通过继承把两个类暴露给业务层:

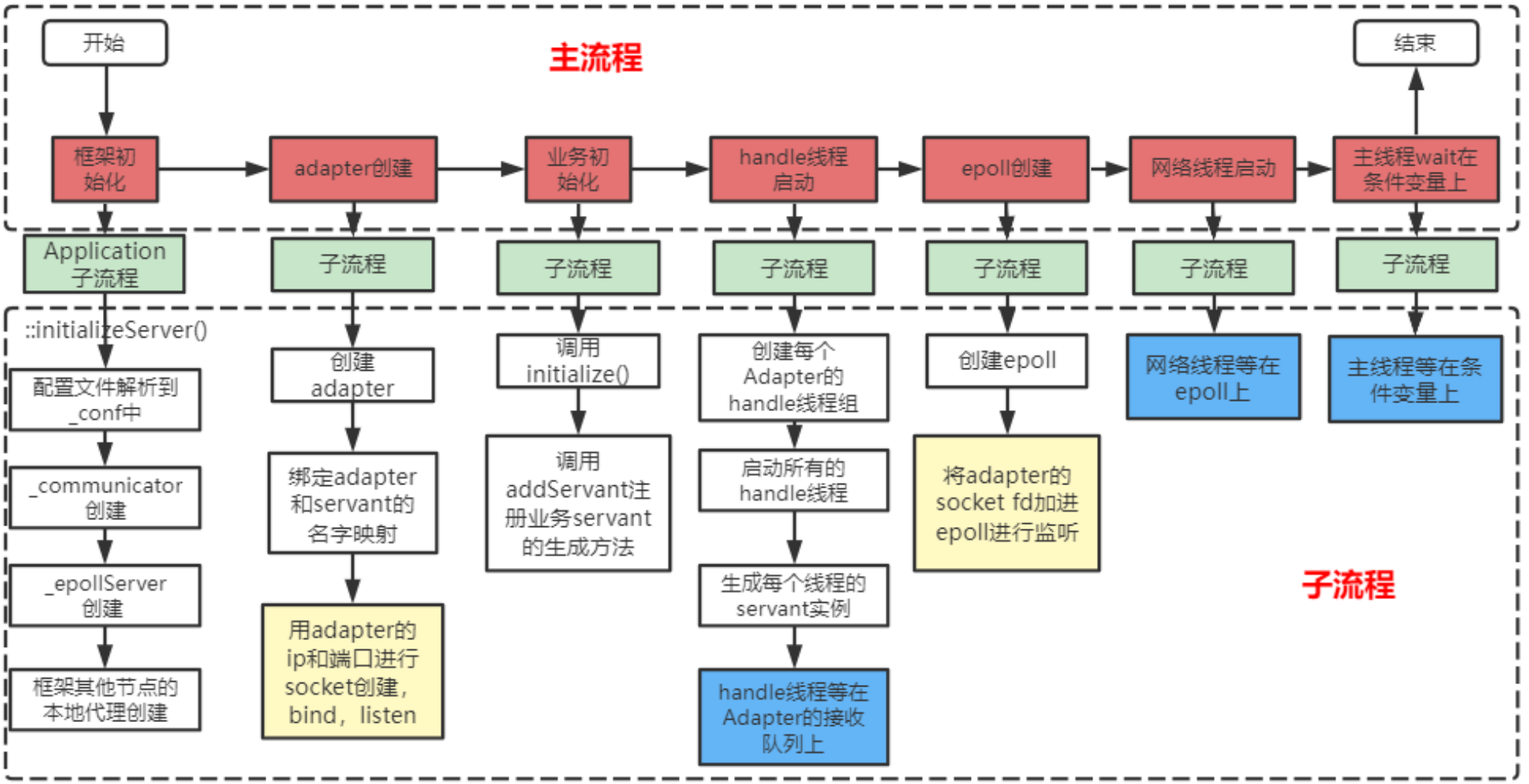
1. 业务逻辑是变化的。一个server可以同时监听多个端口，框架把每个端口提供的服务抽象成servant，业务层通过继承servant，可以实现自己的业务逻辑，然后把实现好的servant注册进框架中。
2. 业务层和框架层的交互，例如业务层需要把servant注册进框架，把协议注册进框架，注册命令到框架，获取框架的默认通信器，拉配置文件等，这些操作都是通过Application暴露给业务层的。每个业务server继承Application，还可以重载Application的一些接口，实现业务的自定义。

1.3 servant的变和不变



- 蓝色箭头方向，HelloServer通过几个接口调用了框架提供的功能：servant的注册，配置文件拉取，协议注册等；
- 黄色箭头方向，框架在收到客户端的请求后，找到该端口对应的servant，如果是tars协议，则调用servant的**onDispatch**接口把请求分发到HelloImp对应的rpc接口；如果不是tars接口，则调用servant的doRequest接口，而HelloImp会重载这个接口去处理请求。

2 主线程逻辑



主要逻辑分布在application.main()和application.watiForShutdown()。

2.1 框架初始化

对应源码的函数

```
void Application::main(const string &config)
```

2.2 adapter创建

对应函数: void Application::bindAdapter(vector<TC_EpollServer::BindAdapterPtr>& adapters)

2.3 业务初始化

```
virtual void Servant::initialize() = 0;
```

实际调用的是子类函数

```
void HelloImp::initialize()
```

```
784     try
785     {
786         //业务应用的初始化
787         initialize();
788
789         {
790             std::unique_lock<std::mutex> lock(mtx);
791             initing = false;
792             cond.notify_all();
793         }
794
795         keepActiving.join();
796     }
```

2.4 handle线程启动

1. 创建每个adapter的handle线程组
2. 启动所有的handle线程。
3. 在启动handle线程后，每个handle线程会根据该handle所处理的adapter名字获取servant的生成器，创建对应的servant对象来处理网络请求消息（所以，每一个handle线程都会有自己的一个独立的servant对象)。

```
ServantHandle::initialize()
ServantHandle::handle
        handleTarsProtocol(current);
        handleNoTarsProtocol(current);
```

2.5 epoll创建

```
TC_EpollServer::waitForShutdown()
    initHandle();

    createEpoll();

    startHandle();
```

(1)每个网络线程对象分别创建epoll
(2)将adapter的socket fd加到第一个网络线程对象的epoll中。另外每个网络线程还有两个socket fd:_shutdown,_notify。_shutdown是进程结束时，用来唤醒网络线程的epoll.wait()用的；_notify是handle线程往发送队列push消息后，通知网络线程有发送消息用的。

2.6 网络线程启动

(1)网络线程等待在epoll上，开始处理网络请求
(2)如果检查到terminated，则网络线程结束。

TC_EpollServer::NetThread::run()

```
(gdb) bt
#0  tars::TC_EpollServer::NetThread::NetThread (this=0xc59bc0, threadIndex=0, epollServer=0xc3fbf0)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_epoll_server.cpp:1587
#1  0x0000000000089a5f6 in tars::TC_EpollServer::initHandle (this=0xc3fbf0) at /home/lqf/tars/TarsFrame
#2  0x0000000000899c1f in tars::TC_EpollServer::waitForShutdown (this=0xc3fbf0) at /home/lqf/tars/Tars
#3  0x000000000074de68 in tars::Application::waitForShutdown (this=0xc227a0 <g_app>) at /home/lqf/tars
```

TC_EpollServer::Connection::onParserCallback

TC_Transceiver::initializeClient

2.7主线程等待在_epollServer的条件变量上

当服务被设置为terminate(例如ctrl+c或者web上关掉该服务),则主线程准备结束，同时通知其他线程服务状态已经为terminate，可以结束了。
之后主线程调用destroyApp(),让业务层在进程结束前做清理工作。到这里，waitForShutdown()结束，返回:main()

3 网络IO

TC_Thread::coroutineEntry
    TC_CoroutineScheduler::run
        TC_Epoller::done
            TC_Epoller::wait -> epoll_wait

3.1 tc_epoller, tc_epoll_server1

TC_Epoller：继承自TC_TimerBase定时器类，内部类EpollInfo（重要属性_cookie和回调函数）和NotifyInfo相关信息类（包含TC_Socket）通过内部指向TC_Epoller的指针操纵对象属性。TC_Epoller又拥有指向NotifyInfo的指针_notify，还可以通过_notify操控EpollInfo。
创建TC_Epoller通过create会创建NotifyInfo，NotifyInfo->init->createEpollInfo创建EpollInfo对象。也就是说每一个连接拥有一个NotifyInfo和EpollInfo。

```
TC_Epoller::EpollInfo::registerCallback 注册回调处理事件
```

- notify：这里使用了一个技巧，默认et边缘模式，通过epoll_ctl一个fd为EPOLLOUT，会马上唤醒epoll。
- syncCallback：同步回调，一般是其他线程调用，epoll监听的线程处理回调。创建一个udp套接字，将套接字包装到NotifyInfo中然后将NotifyInfo指向这个TC_Epoller，接着包装一个执行同步函数和用来通知事件的回调匿名函数，将匿名函数注册到epoll中，然后条件变量等待。此时这个线程就在等待事件的发生。

3.1 tc_epoller, tc_epoll_server2

服务模型，四种模式：

1. NET_THREAD_QUEUE_HANDLES_THREAD: 独立网路线程 + 独立handle线程: 网络线程负责收发包, 通过队列唤醒handle线程中处理
2. NET_THREAD_QUEUE_HANDLES_CO: 独立网路线程组 + 独立handle线程: 网络线程负责收发包, 通过队列唤醒handle线程中处理, handle线程中启动协程处理
3. NET_THREAD_MERGE_HANDLES_THREAD: 合并网路线程 + handle线程(线程个数以处理线程配置为准, 网络线程配置无效): 连接分配到不同线程中处理(如果是UDP, 则网络线程竞争接收包), 这种模式下延时最小, 相当于每个包的收发以及业务处理都在一个线程中
4. NET_THREAD_MERGE_HANDLES_CO: 合并网路线程 + handle线程(线程个数以处理线程配置为准, 网络线程配置无效): 连接分配到不同线程中处理(如果是UDP, 则网络线程竞争接收包), 每个包会启动协程来处理

更多参考： tarscpp\util\include\util\tc_epoll_server.h的注释

3.2 TC_EpollServer

TC_EpollServer：继承自TC_HandleBase智能指针基类和LogInterface日志接口。包含内部类RecvContext接收包上下文类和SendContext发送包上下文类。

细节：

- RecvContext继承了std::enable_shared_from_this，拥有了将自己包装成智能指针的成员函数。
- initHandle：初始化handle线程。
- createEpoll：创建资源。
- startHandle：启动handle线程。
- waitForReady：等待线程都初始化好。
- waitForShutdown：先执行new TC_Epoller()，创建TC_Epoller对象，组合顺序执行上面四个操作。accept阻塞，有连接过来就会创建EpollInfo，并注册这个连接fd的回调函数。然后执行loop循环直到状态变成停止。_bindAdapters需要bind监听操作。manualListen手动监听。只能一个线程epoll_wait，将事件分发给其他线程。

TC_EpollServer维护整个服务对象，核心成员包括：

- std::vector<NetThread *> _netThreads,
- TC_Epoller *_epoller = NULL;
- vector _bindAdapters;。
- unordered_map<int, BindAdapterPtr> _listeners;

3.2 NetThread

NetThread维护网络线程对象，核心成员包括：

- TC_Epoller* _epoller = NULL;
- shared_ptr _list;
- send_queue _sbuffer;
- std::function<void()> _handle;
- Epoller在NetThread运行

BindAdapter维护服务端口的信息，核心成员包括：

- vector _handles;
- TC_Socket _s;
- shared_ptr _dataBuffer;

4. 腾讯tars组件

1. TC_ThreadLock普通线程锁
    1. TC_ThreadMutex
    2. TC_ThreadCond
2. TC_Thread线程基类
3. 线程安全队列TC_ThreadQueue
4. TC_CasQueue 无锁队列,spinlock
5. mysql操作类：TC_Mysql
6. 网络组件
    1. TC_Socket 封装了socket的基本方法
        - 提供socket的操作类;
        - 支持tcp/udp socket;
    2. TC_Epoller ： 提供网络epoll的操作类
        - 提供add、mod、del、wait等基础操作
    3. TC_ClientSocket，用于客户端，提供init(const string &slp, int iPort, int iTimeout); 传入ip 端口 和 超时时间
7. 命令解析、配置文件
    1. TC_Config 2. TC_Option
8. 异常处理 TC_Exception

tc_cas_queue

无锁队列，其实还是用了自旋锁，有尝试次数，次数到了还是会睡眠

用途：TC_LoggerRoll

异步回调后的处理线程：AsyncProcThread

Servant：TC_CasQueue<ReqMessagePtr> _asyncResponseQueue;

TC_CasQueue<ReqMessagePtr> _asyncResponseQueue;

协程

协程上下文切换声明在tc_fcontext中的make_fcontext和jump_fcontext函数，汇编实现。协程切换原理是：保存上下文相关的寄存器到参数指定的地址，然后将传入的参数的值传给相应的寄存器，然后jum跳转到准备好栈的新地址。（eax、ebx、ecx、edx、esi、edi通用寄存器，esp栈指针寄存器，ebp基地址指针寄存器，eip下一条指令指针寄存器）。

- TC_CoroutineQueue：协程队列，类似线程池。
- TC_CoroutineInfo：协程信息类。调度对象和协程对象可以快速查询协程信息。要执行的协程会通过CoroutineDel移除协程，然后设置可执行状态加入到队列尾部。
- TC_CoroutineScheduler：协程调度类，每个线程独立拥有，很多地方可以避免加锁，因为这些对象只有一个线程拥有。g_scheduler放在thread_local中，静态的表示初始化的时候就分配好内存。构造函数会创建一个TC_Epoller的对象_epoller，用来处理调度。调度对象由每个协程运行函数run创建。
- TC_CoroutineScheduler::run()调度器核心执行函数，wakeup唤醒需要激活的协程，wakeupbytimeout唤醒sleep协程，wakeupbyself唤醒yield协程。如果没有协程可以运行了就退出调度循环。
- TC_Coroutine：协程类，继承自TC_Thread线程基类，每个线程内部都有一个指向协程调度器的智能指针_coroSched。run()->handleCoro()真正运行协程初始化回调和加入调度执行。由于协程数量一般比较多，需要暂定协程需要执行_coroSched的terminate，最终调用epoll_ctl系统调用。
- 协程总结：协程当线程用，自动调度（当_activeCoroQueue，_avail，_active队列没有协程时，等待epoll事件），调度器发现没有协程可运行自动退出。