高性能Tars开发框架的实践之路

分享人：suziliu(刘豪)　时间：2018年12月22日

START

主要内容

- Tars整体介绍
- Tars开发框架在性能方面的技术实践
- Tars未来发展规划

1. TARS整体介绍

Tars微服务框架产生的背景

- **从2006年开始，腾讯接连推出了多个手机应用：手机QQ、超级QQ、手机腾讯网、手机QQ游戏大厅、手机QQ浏览器等，随着产品、用户规模的增长，后台服务开始面对各种各样的问题与挑战。**

业务逻辑

业务逻辑集中，耦合性强，开发维护成本高，服务模型多样化，业务协议不统一

基础组件

代码重复率高，性能、高可用性、可扩展性等方面能力参差不齐，难以适应业务海量访问发展趋势

运营管理

运维工具各异，部署管理混乱，规范性差，管理能力薄弱

监控体系

运营数据缺失，监控维度不立体，故障时分析和查找问题困难

Tars发展历程

- 面对业务海量访问，我们采用微服务的思想，设计和实现了一个通用的统一应用框架，给业务提供涉及到开发、运维的一整套解决方案，让开发和运维越来越简单高效。

Tars有什么优势

- **业界开源优秀的微服务框架现状**

无服务治理类

专注于网络通信，RPC或消息队列模式，部分框架支持多语言开发

Apache Thrift ™

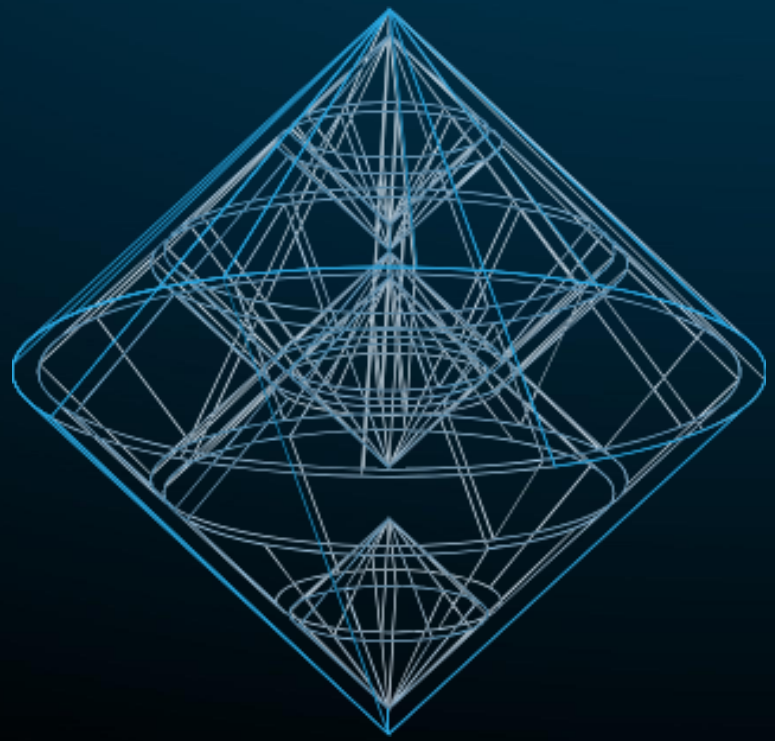
gRPC

单语言带服务治理类

在通信框架的基础上支持服务治理能力，单一编程语言实现，JAVA语言为主流

Spring Cloud

DUBBO



ServiceMesh

ServiceMesh体系，通过SideCar模式解决多技术栈问题，目前处于发展成熟期

LINKERD

envoy

多语言带服务治理类

在通信框架的基础上支持服务治理能力，多种编程语言实现

TARS

Tars最大优势是在于提供服务治理和解决多技术栈的同时，可以获得更好的性能

Tars整体结构

- Tars是一个支持多语言、内嵌服务治理功能，与Devops能很好协同的微服务框架

2. Tars开发框架在性能方面的技术实践

影响服务框架性能的主要因素有哪些

01

协议

02

IO模型

03

线程模型

04

编程模型

最初TARS开发框架的架构设计

- 基于IDL的通信协议
- 基于RPC的调用方式

Tars 文件

```
struct  UserInfo
{
        1  require   int                    age;
        2  require   char                   sex;
        3  optional string                  company;
        4  optional vector<string>          hobby;
};

interface User
{
        int  getUserInfo( int uid,  out UserInfo  info);
};
```

服务端

```
class UserServant
 {
        virtual int32 getUserInfo(int32 uid,
                                  UserInfo &info,
                                  taf::TarsCurrentPtr current) =
0;
};
```

客户端

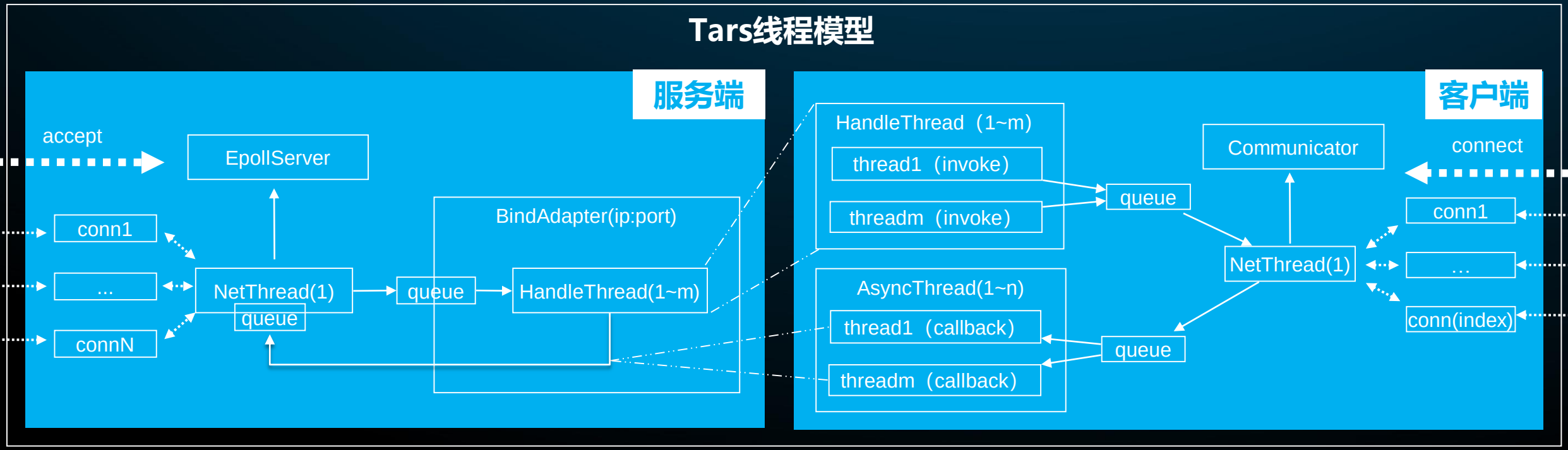
```
class UserProxy
 {
        int getUserInfo(int uid,UserInfo &info, ...);

        void async_getUserInfo(UserCallbackPtr cb, int uid, ...);
 };
```

✓ 高性能　　✓ 兼容性好

✓ 代码自动生成　✓ 多语言支持便利

最初Tars开发框架的架构设计

- 基于Epoll ET的单Reactor+threadpool线程模型



服务端：15w/s；服务端+客户端：4w/s

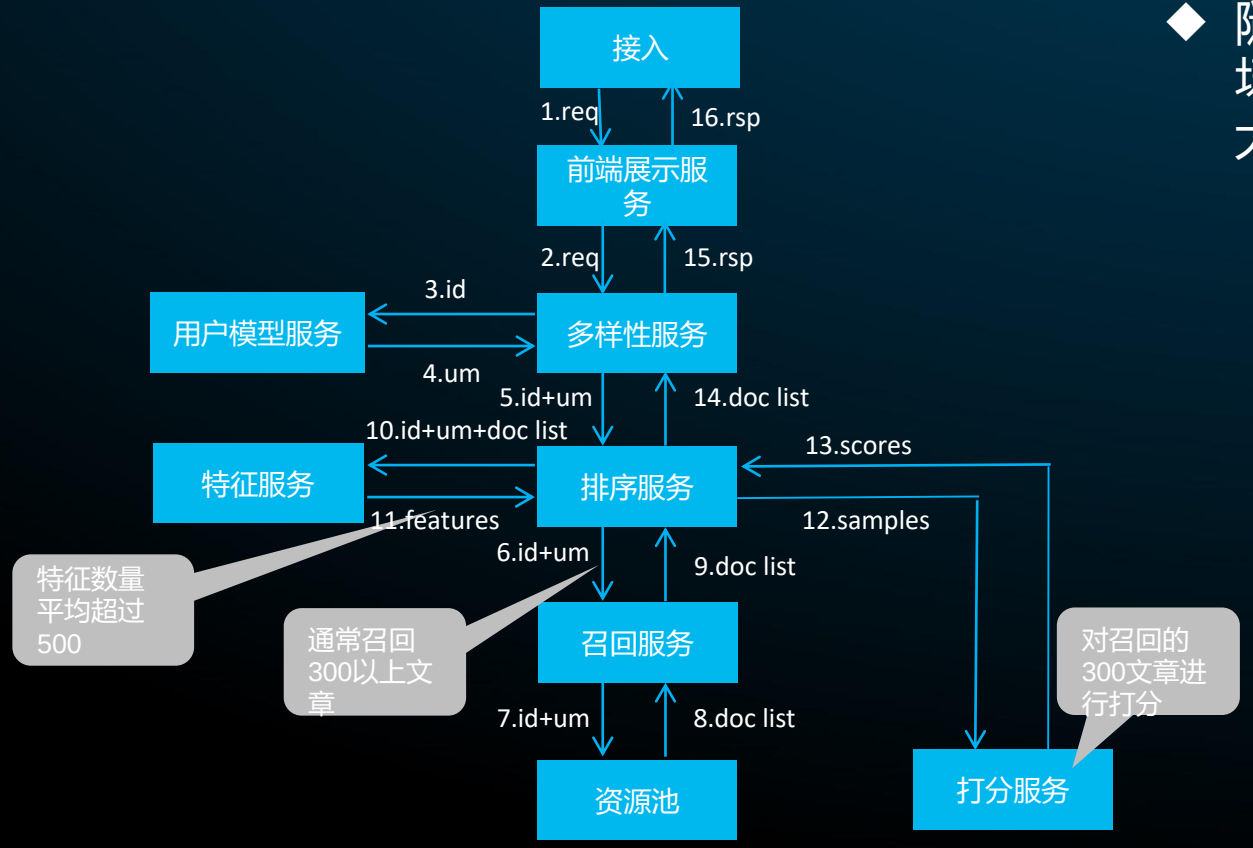
海量并发场景下面对的问题与挑战

- 业务侧
  - 服务模块数：
    100 -> 1000 -> 10000
  - 服务之间的直接调用关系数：
    (1-2) -> (10-20)
  - 。。。
- 硬件技术
  - Cpu核数：
    8 -> 16 -> 24 -> 48 -> 56 -> 80
  - 网卡：
    1Gb->10Gb->40Gb、单队列->多队列
  - 。。。

- 业务对框架的性能、延时、易用性要求越来越高，框架该如何优化，助力业务发展？
- 框架难以随着机器性能的增强而线性提升，框架该如何优化，提升性能同时节约成本？

具体的业务场景

- **推荐场景：用户请求量大、调用链条长、业务逻辑复杂、性能和延时要求高**



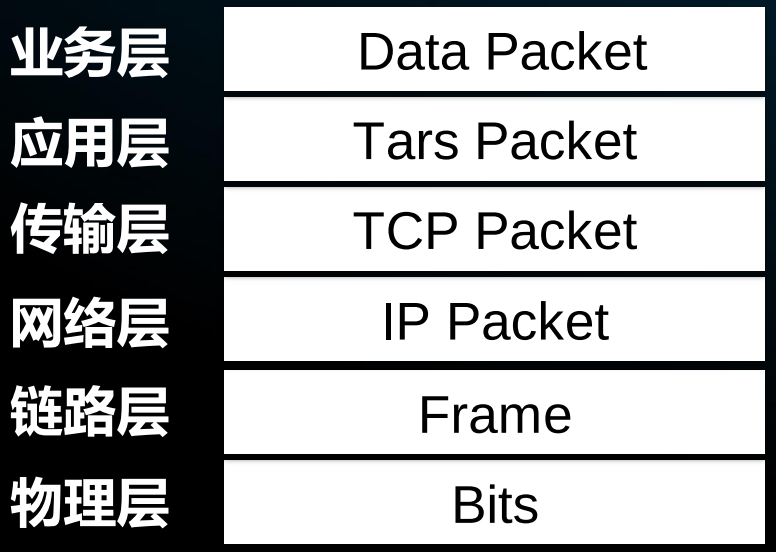
- 随着机器学习/AI的快速发展，类似的场景会越来越多，对框架的挑战越来越大，特别是在性能方面。

性能在理论上分析

- Tars开发框架的性能该如何提升，理论上性能能到多少?

以太网frame

千兆网卡，最小的以太网frame为84B，理论上每秒能发送148wframe。

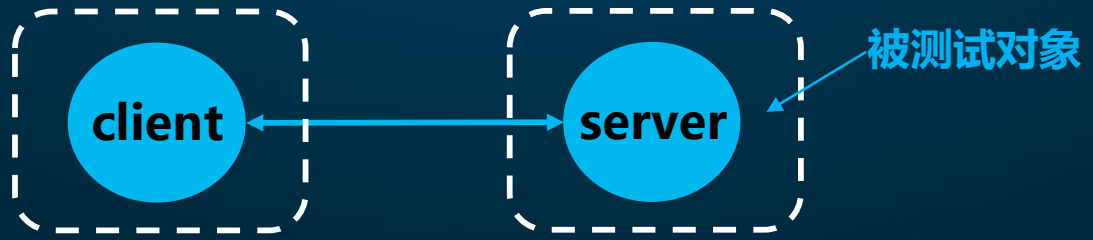


包含一个10B业务数据包的以太网frame大小为128B左右，

理论上Tars服务端的处理性能上限为97w/s(125MB/128)。

服务端模式性能问题分析

- 服务端性能测试：
  - 性能：15w/s



机器：8核cpu（HT），8G内存，千兆网卡

```
top - 10:07:53 up 53 days, 14:35,  3 users,  load average: 0.01, 0.11, 0.07
Tasks: 441 total,   7 running, 433 sleeping,   0 stopped,   1 zombie
Cpu(s): 61.6%us,  9.5%sy,  0.0%ni, 20.3%id,  0.0%wa,  0.0%hi,  8.7%si,  0.0%st
Mem:   8174040k total,  8108408k used,    65632k free,   595336k buffers
Swap:  2104504k total,        0k used,  2104504k free,  7245788k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
29056 mqq       20   0  593m  15m 2624 R   99  0.2   4:48.57 AAServer
29066 mqq       20   0  593m  15m 2624 S   75  0.2   3:31.66 AAServer
29068 mqq       20   0  593m  15m 2624 R   74  0.2   3:31.47 AAServer
29071 mqq       20   0  593m  15m 2624 R   73  0.2   3:32.42 AAServer
29065 mqq       20   0  593m  15m 2624 S   72  0.2   3:31.03 AAServer
29067 mqq       20   0  593m  15m 2624 R   72  0.2   3:31.66 AAServer
29069 mqq       20   0  593m  15m 2624 R   72  0.2   3:32.35 AAServer
29070 mqq       20   0  593m  15m 2624 R   72  0.2   3:32.54 AAServer
13772 root      20   0 9452 7368 7016 S    1  0.1   5:44.03 TsysAgent
13773 root      20   0 4236 2116 1484 S    1  0.0   5:33.45 TsysProxy
29337 mqq       20   0 8996 1464  864 R    1  0.0   0:00.95 top
31574 mqq       20   0  432m  20m 2524 S    1  0.3 327:57.30 demo
31576 mqq       20   0  432m  20m 2524 S    1  0.3 328:23.52 demo
31575 mqq       20   0  432m  20m 2524 S    1  0.3 330:32.18 demo
```

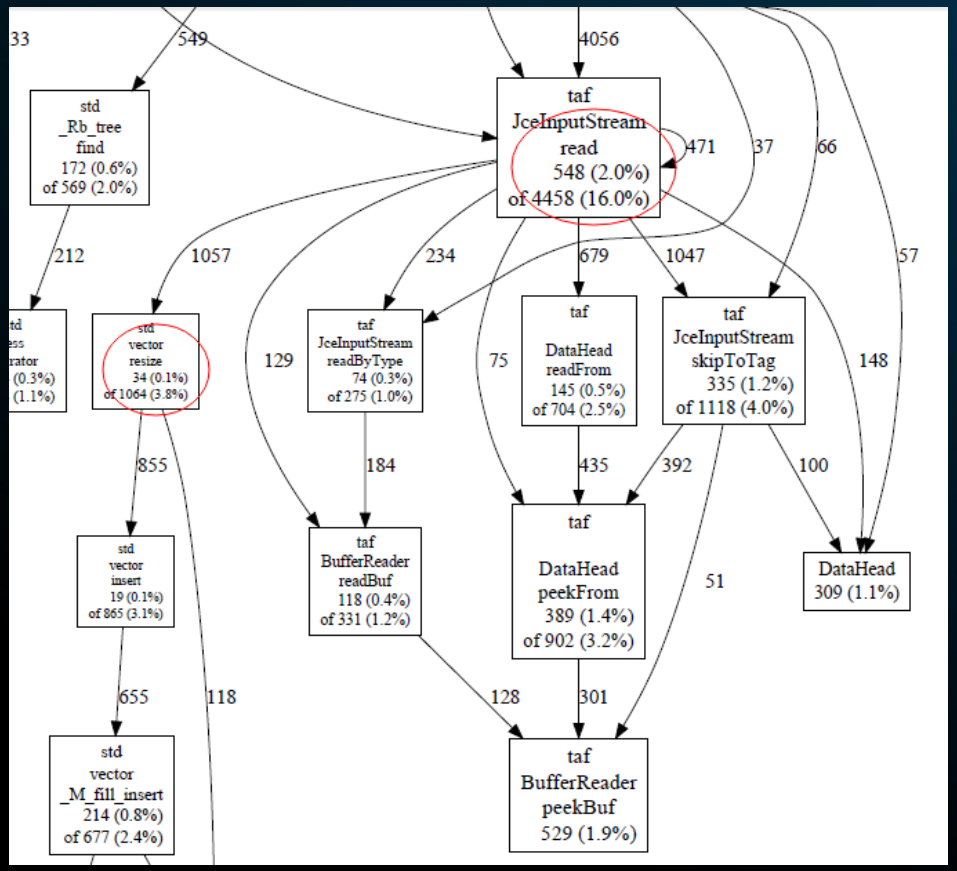
- CPU 20.3% 空闲，跑不满
- 网络线程忙，但网卡没跑满

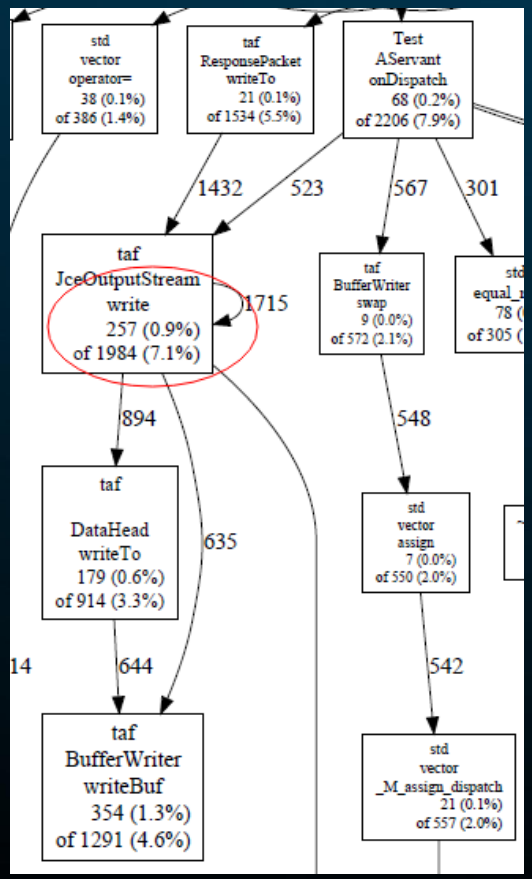
```
mqq@144_147:~> vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  0      0  66052 595304 7247204    0    0     0    13    1    1 21  8 71  0  0
 7  0      0  65308 595304 7247260    0    0     0     8 26231 69497 62 17 21  0  0
 8  0      0  65192 595304 7247320    0    0     0     8 26537 66210 60 20 21  0  0
 9  0      0  65192 595304 7247368    0    0     0     8 26693 73520 64 20 16  0  0
 6  0      0  64572 595304 7247424    0    0     0     8 25856 76107 64 20 16  0  0
 9  0      0  64564 595304 7247484    0    0     0  1824 26327 68746 61 19 20  0  0
 8  0      0  65228 595304 7247540    0    0     0     8 26161 70096 62 18 20  0  0
 8  0      0  65352 595304 7247596    0    0     0     8 26531 63778 58 18 24  0  0
 1  0      0  63864 595304 7247652    0    0     0     8 26741 65683 60 19 22  0  0
 4  0      0  64608 595304 7247708    0    0     0    72 27445 67046 59 18 23  0  0
 8  0      0  63920 595304 7247764    0    0     0    56 27708 68506 60 17 24  0  0
 8  0      0  64020 595312 7247820    0    0     0    48 27543 67265 60 18 22  0  0
10  0      0  63648 595312 7247872    0    0     0    60 27321 67256 62 16 22  0  0
 2  0      0  63896 595312 7247944    0    0     0     8 26853 70243 61 18 21  0  0
 2  0      0  64516 595312 7248032    0    0     0     8 27513 68309 61 17 22  0  0
 8  0      0  66128 595312 7248080    0    0     0    16 27208 65897 58 18 23  0  0
 9  0      0  68732 595312 7248152    0    0     0     8 26501 69006 61 18 21  0  0
 8  0      0  65632 595312 7248188    0    0     0     8 25001 72923 62 20 18  0  0
```

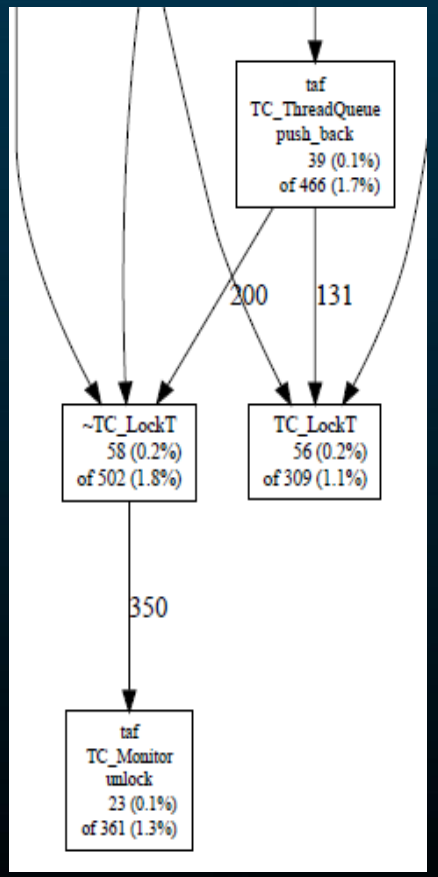
- 上下文切换次数多，3倍+于网络中断

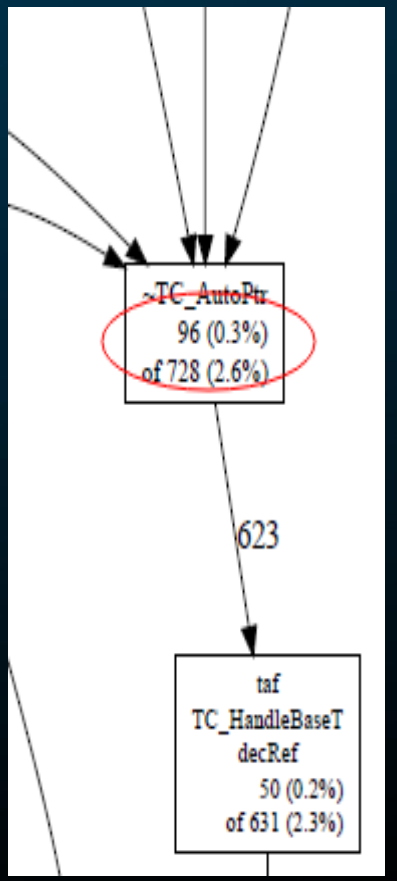
服务端模式性能问题分析

- 服务端的gperftools.cpu-profiler性能分析图：







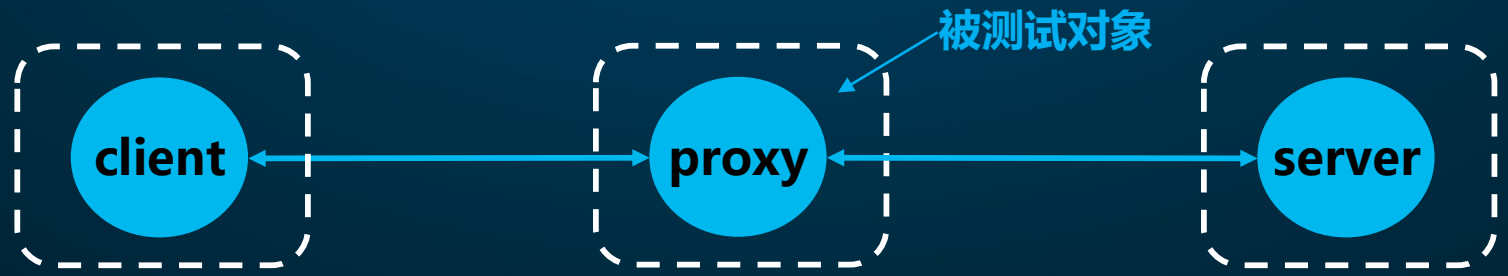


- 协议的序列化和反序列化占23.1%

- 加锁和解锁占3.9%
- 智能指针析构占2.6%

服务端+客户端模式性能问题分析

- 服务端+客户端性能测试：
  - 性能：4w/s



机器：8核cpu（HT），8G内存，千兆网卡

```
top - 10:51:58 up 53 days, 15:20,  3 users,  load average: 0.00, 0.00, 0.00
Tasks: 422 total,   4 running, 418 sleeping,   0 stopped,   0 zombie
Cpu(s): 48.2%us,  7.4%sy,  0.0%ni, 39.8%id,  0.0%wa,  0.0%hi,  4.6%si,  0.0%st
Mem:   8174040k total,  8072416k used,   101624k free,   596108k buffers
Swap:  2104504k total,        0k used,  2104504k free,  7213088k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5572 mqq       20   0  508m  19m 2640 R   94  0.2   3:10.23 BBServer
 5574 mqq       20   0  508m  19m 2640 R   80  0.2   2:44.10 BBServer
 5569 mqq       20   0  508m  19m 2640 R   55  0.2   1:50.92 BBServer
 5578 mqq       20   0  508m  19m 2640 S   46  0.2   1:29.50 BBServer
 5579 mqq       20   0  508m  19m 2640 S   45  0.2   1:29.45 BBServer
 5577 mqq       20   0  508m  19m 2640 S   45  0.2   1:29.40 BBServer
 5580 mqq       20   0  508m  19m 2640 S   45  0.2   1:28.49 BBServer
 5581 mqq       20   0  508m  19m 2640 S   45  0.2   1:29.45 BBServer
31576 mqq       20   0  432m  20m 2524 S    1  0.3 328:48.51 demo
31574 mqq       20   0  432m  20m 2524 S    1  0.3 328:22.19 demo
31575 mqq       20   0  432m  20m 2524 S    1  0.3 330:57.11 demo
```

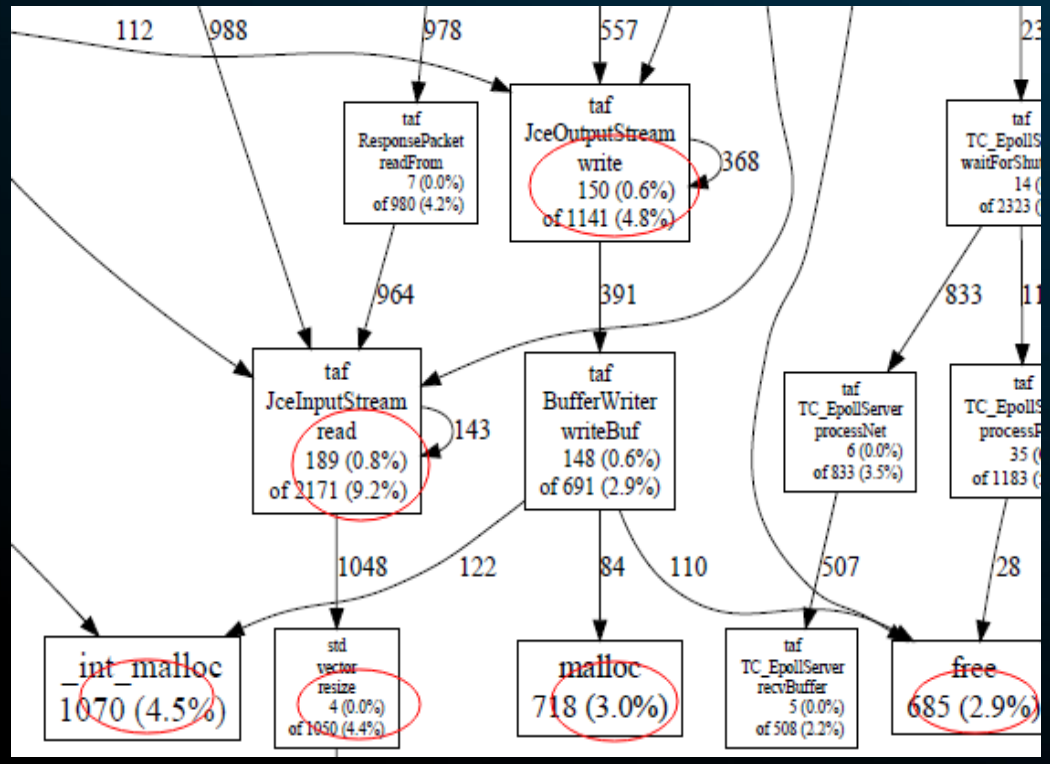
- CPU 39.8% 空闲，跑不满
- 网络线程忙，但网卡没跑满

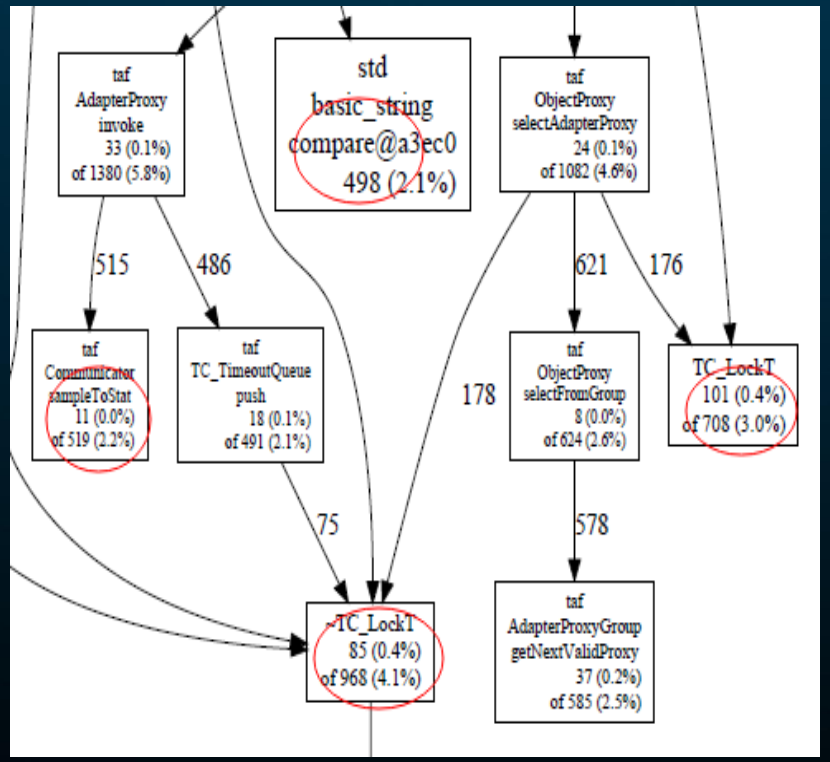
```
mqq@144_147:~/taf/app_log> vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
10  0      0 100052 596112 7215360    0    0     0    13    1    0 21  8 71  0  0
 8  0      0 100400 596112 7215416    0    0     0     8 18511 154012 45 11 44  0  0
 6  0      0  99904 596112 7215472    0    0     0    16 18932 152827 47 10 43  0  0
 3  0      0 100152 596112 7215524    0    0     0     8 18613 164356 51 11 37  0  0
 6  0      0  99392 596112 7215580    0    0     0     8 19013 159735 47 10 42  0  0
 3  0      0  98896 596112 7215636    0    0     0     8 18613 161170 48 11 41  0  0
 6  0      0  99640 596112 7215696    0    0     0     8 19124 159087 44 12 44  0  0
 3  0      0  99764 596112 7215756    0    0     0   176 19381 161623 44 12 45  0  0
 6  0      0  99020 596112 7215808    0    0     0    36 19310 160890 46 11 43  0  0
 4  0      0  98672 596116 7215856    0    0     0    68 18992 166303 50 13 37  0  0
 7  0      0  99416 596116 7215920    0    0     0     8 18607 162325 47 13 40  0  0
 7  0      0  98300 596116 7215976    0    0     0     8 18514 165355 49 11 40  0  0
 2  0      0  99168 596116 7216036    0    0     0    12 19058 163859 48 12 40  0  0
 4  0      0  98548 596116 7216088    0    0     0     8 18891 163610 47 10 42  0  0
 8  0      0  98300 596116 7216144    0    0     0     8 18878 163166 52  9 39  0  0
 3  0      0  98300 596116 7216200    0    0     0     8 19162 160879 50 11 39  0  0
 3  0      0  97680 596116 7216256    0    0     0     8 19117 164800 50 11 39  0  0
 4  0      0  98424 596116 7216308    0    0     0    68 18863 169092 50 11 39  0  0
 7  0      0  99044 596116 7216368    0    0     0     8 19035 162668 48 11 41  0  0
 8  0      0  98548 596116 7216416    0    0     0    12 18957 166149 48 13 39  0  0
```

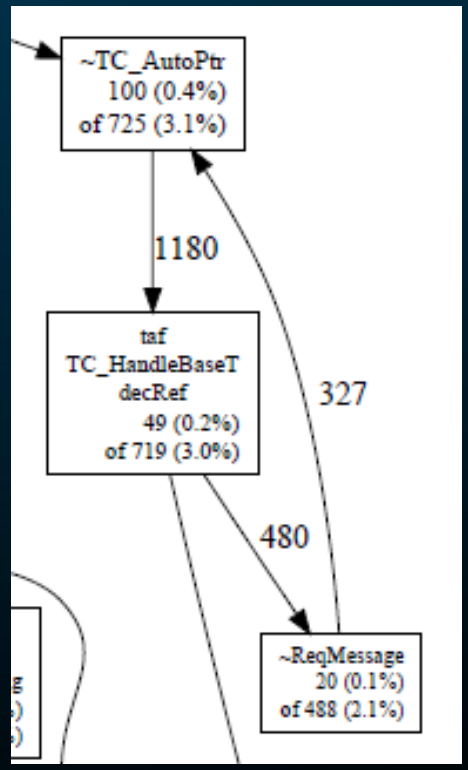
- 上下文切换次数多，8倍+于网络中断

服务端+客户端模式性能问题分析

- 服务端+客户端的gperftools.cpu-profiler性能分析图：







- 协议的序列化和反序列化占14%
- 内存的初始化、分配和释放占10.4%
- 加锁和解锁占7.1%
- 智能指针析构占3%

性能问题分析总结

CPU：idle

Network：idle

Disk I/O：idle

Memory：enough

+

网络线程很忙

太多的上下文切换

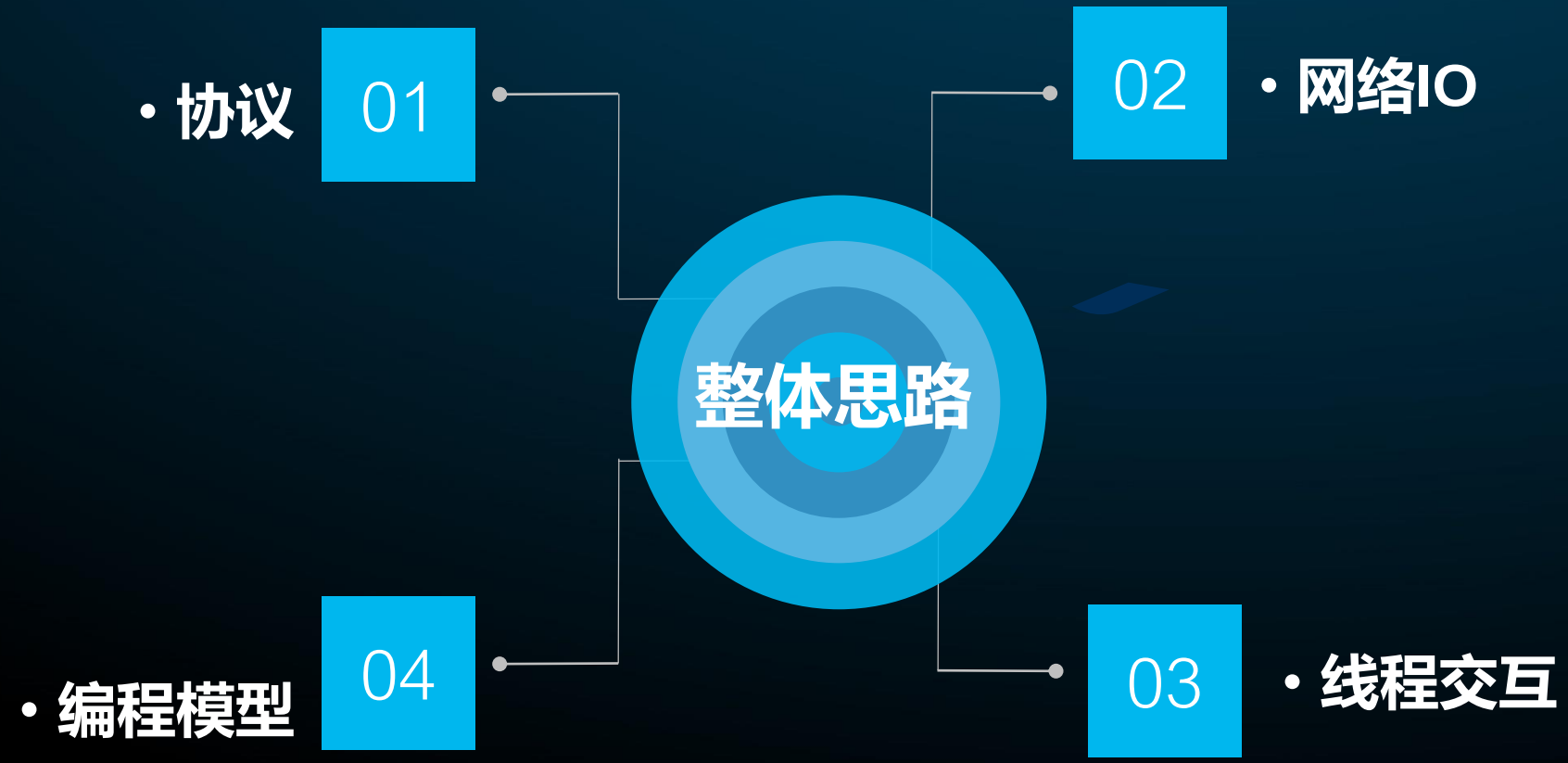
序列化和反序列化CPU消耗大

加锁和解锁操作频繁

内存的分配和释放频繁

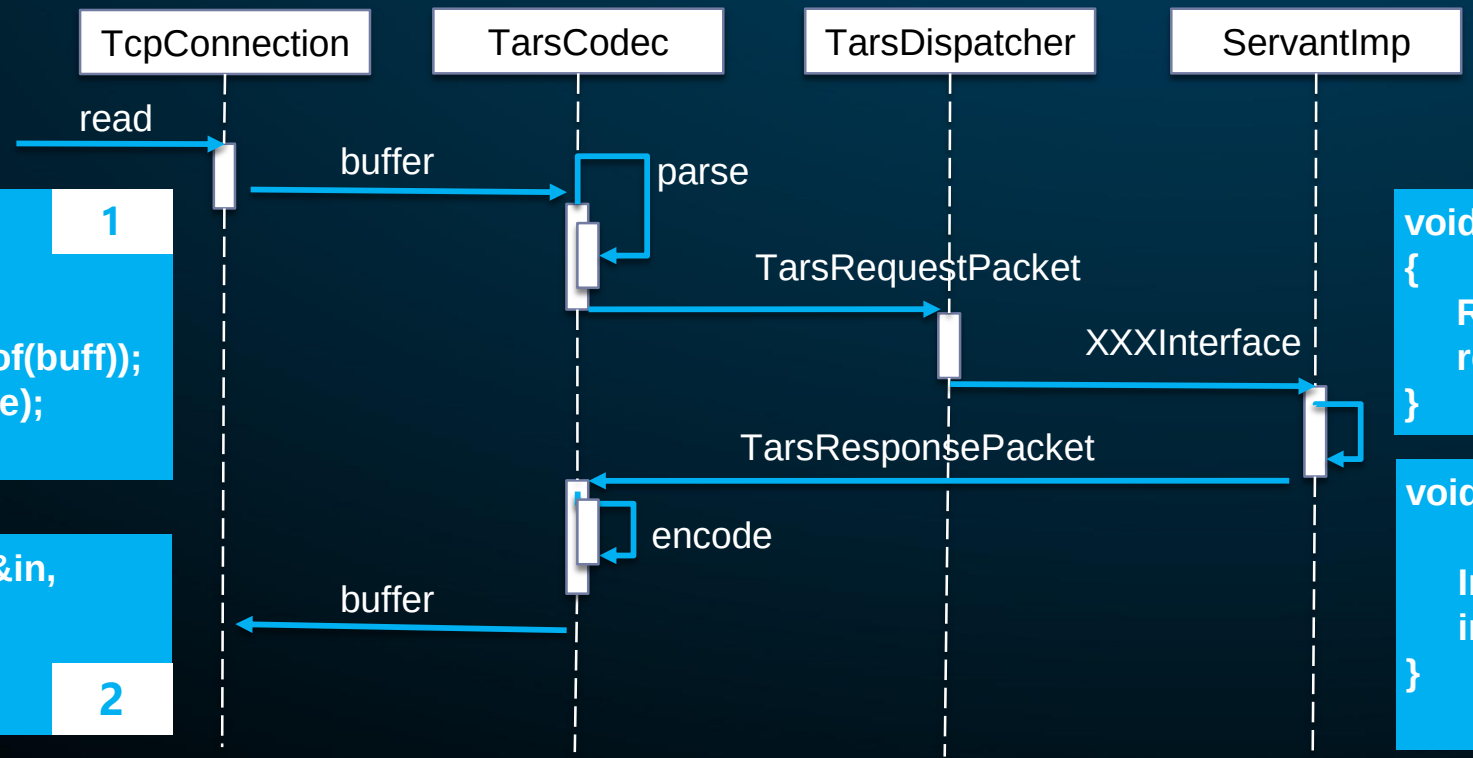
。。。

优化思路



协议之请求解析优化

- 优化前：



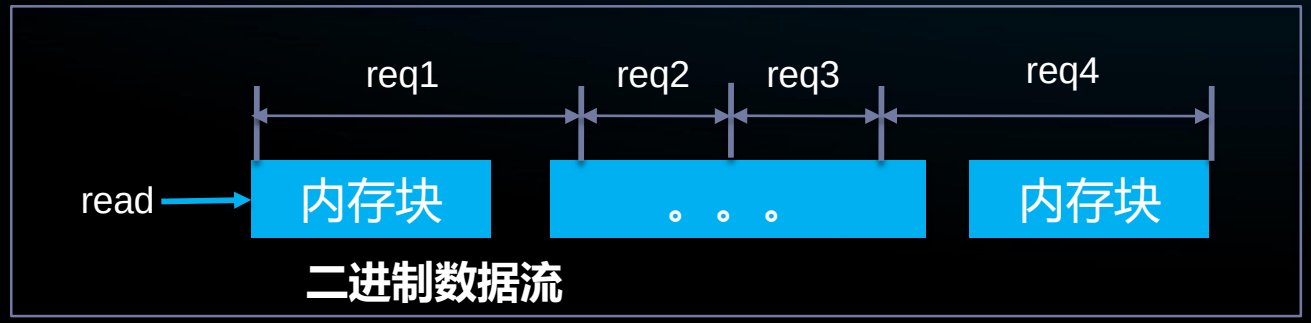

```
string  buffer;
while  (true)  {
   char  buff[64*1024];
   int size=read(buff,sizeof(buff));
   buffer.append(buff,,size);
}
```
1

```
void parse(const  string &in,
           string &out) {
   out = in.substr(4);
}
```
2

```
void  tars_decode(const  string &in)
{
   RequestPacket  req;
   req.readFrom(in);
}
```
3

```
void  interface_decode(
      const  RequestPacket &req) {
   Interface  interface;
   interface.readFrom(req.buffer);
}
```
4

- 优化后：



3次+的内存拷贝变成零拷贝

协议之编解码优化

```
struct TestInfo
{
    1  require  vector<string> vs;
    2  require  map<string, string> m;
    3  optional vector<map<string, string>> vm;
    4  optional map<vector<string>, vector<string>> mv;
    5  optional bool     b   = true;
    6  optional byte     by  = 0;
    7  optional short    si  = 0;
    8  optional int      ii  = 34;
    9  optional long     li  = 3456;
    10 optional float    f   = 45.34f;
    11 optional double   d   = 0;
    12 optional string   s   = "a\"bc";
    13 require  BTest    t;
    14 optional map<int, string> mi;
};

interface Query
{
    int queryTestInfo(string sQuery, out TestInfo testInfo, out string errStr);
};
```

结构体

tag

必选字段

字段类型

接口

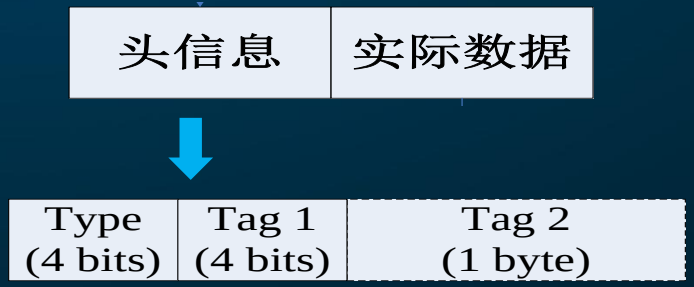
```
TarsOutputStream<BufferWriter> os;
ti.writeTo(os);
```

编码

```
TarsInputStream<BufferReader> is;
is.setBuffer(os.getBuffer(), os.getLength());
ti.readFrom(is);
```

解码

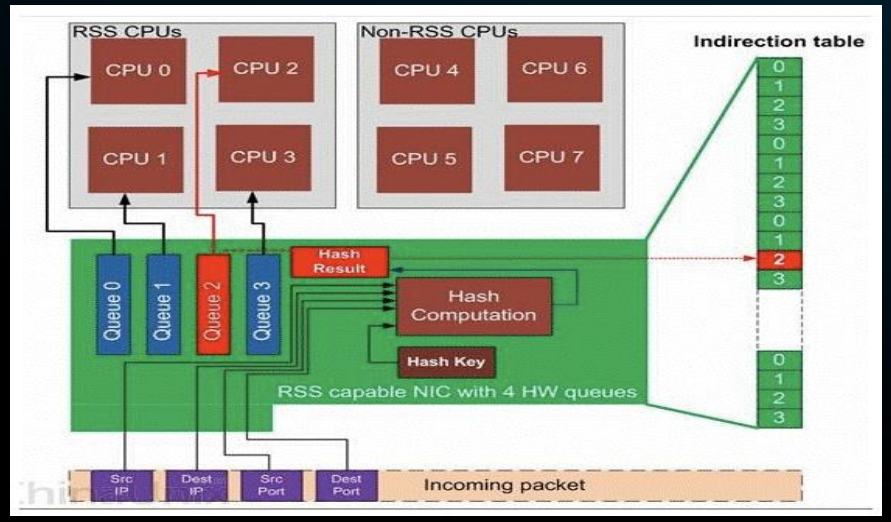
Tars协议编码:



- 问题:
  - 编码时，会不断申请释放内存
  - 解码时，元数据的读取和定位存在重复读
- 解决方案:
  - 减少函数调用的开销
  - 减少临时对象的定义，直接类型转换
  - 编码时，预先计算出所需要的内存空间
  - 解码时，只遍历数据一遍

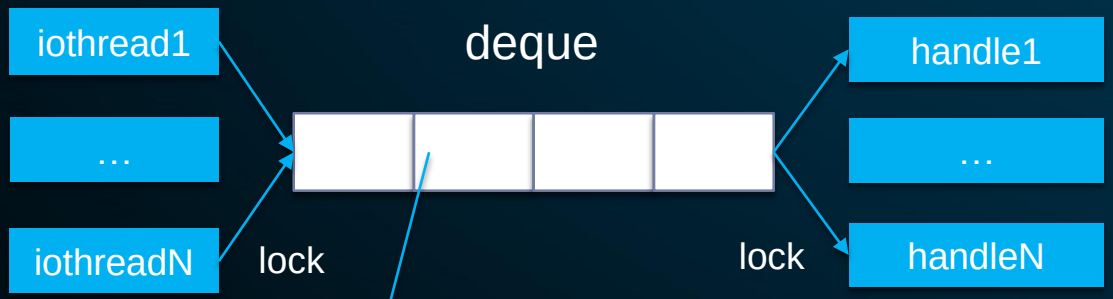
网络IO优化

- 问题：
  - 网络线程忙



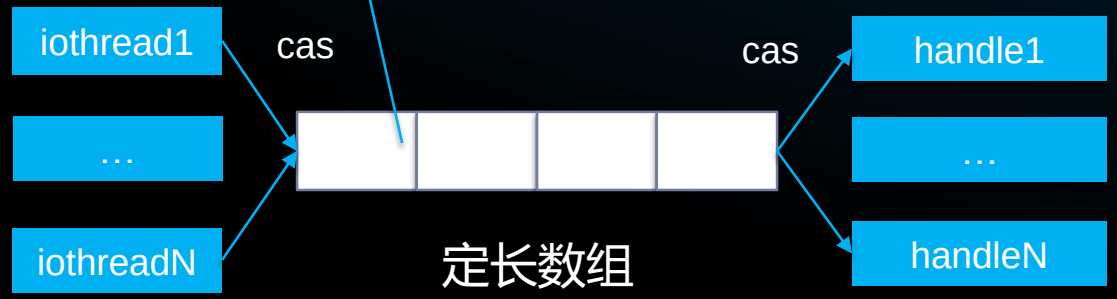
- 解决方案：
  - 使用多网络线程收发包（可配）
  - 减少内存拷贝(使用writev发送了多个不连续的内存数据块)
  - 减少系统调用(buff为空时，先write，再epoll；read或者write数据时，如果返回长度小于缓存大小，就可以退出循环了)

线程交互之队列优化




```
struct Item {
    …
    string data;
    …
}
```

```
Item *p = new Item();
p->data.swap(input);
//p->data = std::move(input);

enqueue(p);
```

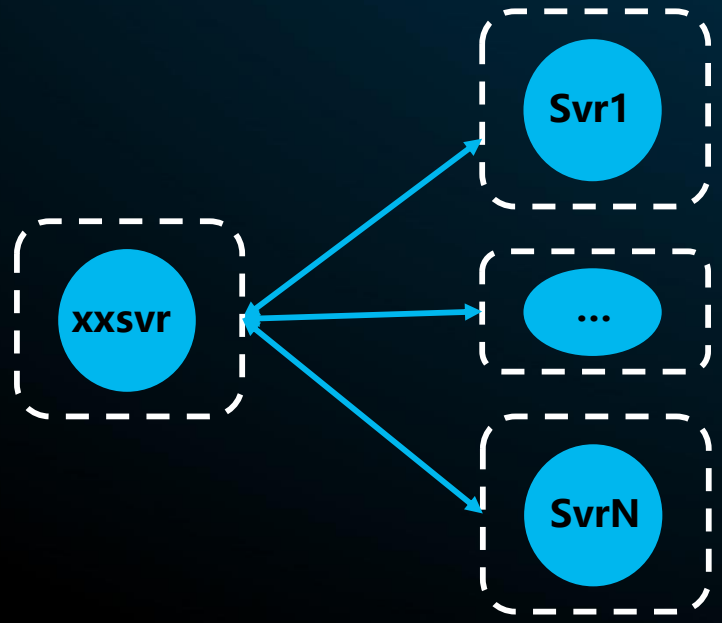


- 问题：
  - 上下文切换频繁
  - 出入队列请求信息存在内存拷贝

- 解决方案：
  - 无锁队列
  - 使用swap或者move

编程模型优化之异步编程

- Callback异步编程：



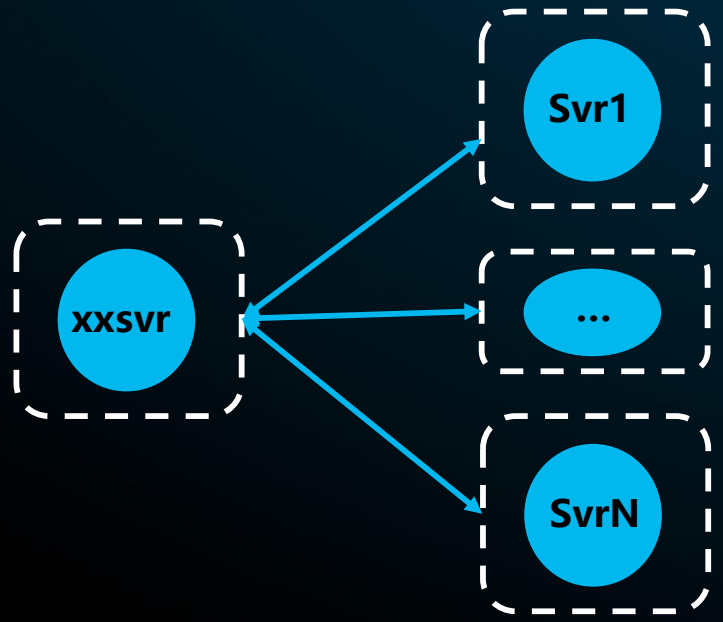

```
tars::Int32 AServantImp::testStr(const std::string& sIn, std::string &sOut,
                                 tars::TarsCurrentPtr current)
{
    current->setResponse(false);

    //业务逻辑处理...

    Test::BServantPrxCallbackPtr cb = new BServantCallback(current);
    _bPrx->async_testStr(cb,sIn);

    return 0;
}
```

```
virtual void callback_testStr(taf::Int32 ret,  const std::string& sOut)
{
    //业务逻辑处理...

    Test::CServantPrxCallbackPtr cb = new CServantCallback(_current);
    _cPrx->async_testStr(cb, sOut);
}
```

```
virtual void callback_testStr(taf::Int32 ret,  const std::string& sOut)
{
    //业务逻辑处理...

    CServant::async_response_testStr(_current, ret, sOut);
}
```

编程模型优化之异步编程

- future/promise编程：



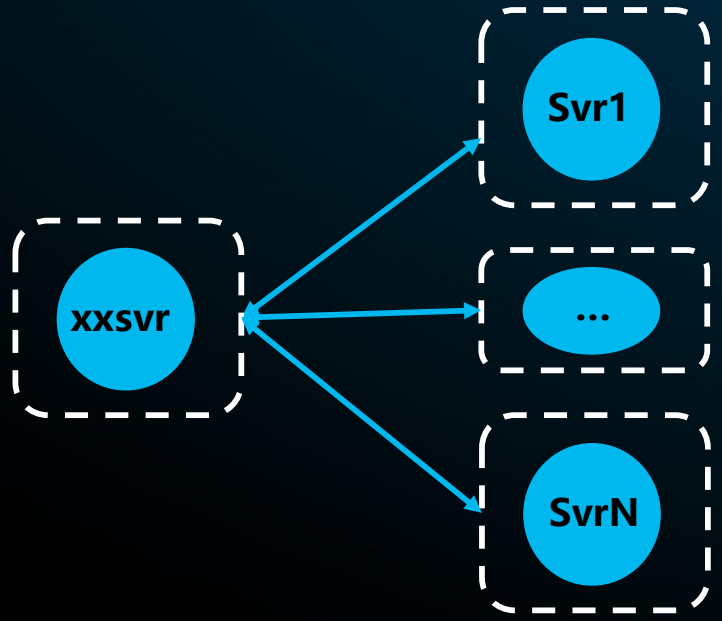

```
taf::Int32 AServantImp::testStr(const std::string& sIn, std::string &sOut,
                                taf::JceCurrentPtr current){
    current->setResponse(false);

    promise::Future<PromiseQueryResultPtr> f = _bPrx->promise_async_queryResult(sIn);

    f.then(promise::bind(&handleBRspAndSendCReq, _cPrx))
     .then(promise::bind(&handleCRspAndReturnClient, current));

    return 0;
}
```

```
promise::Future<PromiseQueryResultPtr> handleBRspAndSendCReq(
              CServantPrx prx, const promise::Future<PromiseQueryResultPtr>& future){
    PromiseQueryResultPtr result;

    result = future.get();

    return prx->promise_async_queryResult(result->sOut);
}
```

```
void handleCRspAndReturnClient(JceCurrentPtr current,
                               const promise::Future<PromiseQueryResultPtr>& future){
    CServantPrxCallbackPromise::PromisequeryResultPtr result;
    result = future.get();

    AServant::async_response_queryResultSerial(current, result->_ret, result->sOut);
}
```

编程模型优化之异步编程

- **协程编程：**

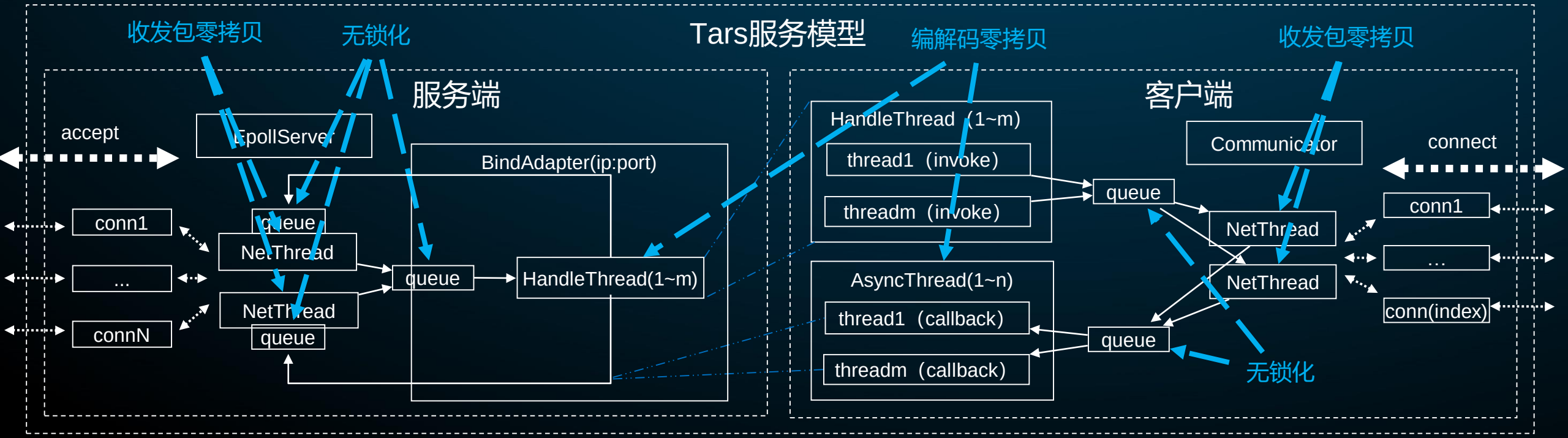



```
taf::Int32 AServantImp::testStr(const std::string& sIn, std::string &sOut,
                                taf::JceCurrentPtr current){
    int ret = 0;
    std::string sRsp;

    ret = _bPrx->queryResult(sIn,sRsp);

    ret = _cPrx->queryResult(sRsp,sOut);

    return 0;
}
```

其它细节优化

- 日志打印，先判断等级，再确定输出
- 日志异步批量落盘
- 编译时使用O2
- vector预先分配好空间
- 使用c++11后的转移语义
- 监控统计数据按接口名合并
- 使用snprintf替换Ostringstream
- strncasecmp替换strcmp(upper(a), upper(b))
- ...

优化后的服务模型结构图



服务端：15w/s -> 61w/s；

服务端+客户端：4w/s -> 29w/s

性能对比

- 与业界开源的微服务框架的性能测试数据：

框架	协议	TPS(10字节)	TPS(128字节)	TPS(256字节)
TARS(C++)	tars	617163	390686	280637
TARS(C++)	http	162761	160061	158412
Spring Cloud	http	160114	157010	156830
gRPC(C++)	http2+ protobuf	89351	86132	81630
gRPC(C++)+Envoy	http2+ protobuf	54512	53236	50617

业务效果

- 业务效果：

```
top - 15:58:41 up 502 days, 22:57,  1 user,  load average: 12.82, 12.09, 11.91
Tasks: 642 total,   1 running, 641 sleeping,   0 stopped,   0 zombie
Cpu(s): 19.5%us,  7.8%sy,  0.0%ni, 69.7%id,  0.0%wa,  0.0%hi,  3.0%si,  0.0%st
Mem:  131648444k total, 131257184k used,   391260k free,    73684k buffers
Swap:        0k total,        0k used,        0k free, 115785340k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
15256 mqq     20   0 15.7g 8.1g 4744 S 514.8  6.4  18289,55 MTTszProxyServe
45411 mqq     20   0 3737m 213m 3204 S 368.1  0.2 855462:29 BeaconProxyServ
28640 mqq     20   0 17.8g  12g  11g S 170.6  9.6 575956:39 MKCacheServer
 1078 mqq     20   0 3384m 431m 3460 S 155.0  0.3 191090:27 QQServiceProxyS
32991 mqq     20   0 3736m  54m 2380 S 72.4   0.0 139181:25 IPSProxyServer
27503 mqq     20   0 2766m 142m 1080 S 16.3   0.1  90955:44 MMGRCPPushHisto
35390 mqq     20   0 3911m 297m 3156 S 14.9   0.2  53802:06 TRomProxyServer
 3319 mqq     20   0 3742m  77m 2336 S 14.6   0.1  40115:24 HdfsLogWriter4S
```

```
top - 16:00:08 up 502 days, 22:58,  1 user,  load average: 11.91, 12.17, 11.97
Tasks: 641 total,   1 running, 640 sleeping,   0 stopped,   0 zombie
Cpu(s): 13.3%us,  6.3%sy,  0.0%ni, 77.7%id,  0.0%wa,  0.0%hi,  2.6%si,  0.0%st
Mem:  131648444k total, 123772188k used,  7876256k free,    73540k buffers
Swap:        0k total,        0k used,        0k free, 115779760k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
45411 mqq     20   0 3737m 213m 3204 S 322.7  0.2 855467:40 BeaconProxyServ
 1078 mqq     20   0 3384m 431m 3460 S 149.7  0.3 191092:37 QQServiceProxyS
44873 mqq     20   0 4100m 1.0g 5376 S 132.8  0.8   0:26.24 MTTszProxyServe
28640 mqq     20   0 17.8g  12g  11g S 117.9  9.6 575959:06 MKCacheServer
40808 mqq     20   0 9828m 5.1g 5.0g S 100.9  4.0  20753:36 MKCacheServer
32991 mqq     20   0 3736m  54m 2380 S 61.8   0.0 139182:26 IPSProxyServer
 3319 mqq     20   0 3742m  77m 2336 S 34.9   0.1  40115:38 HdfsLogWriter4S
```

CPU下降接近4倍

平均延时下降50%

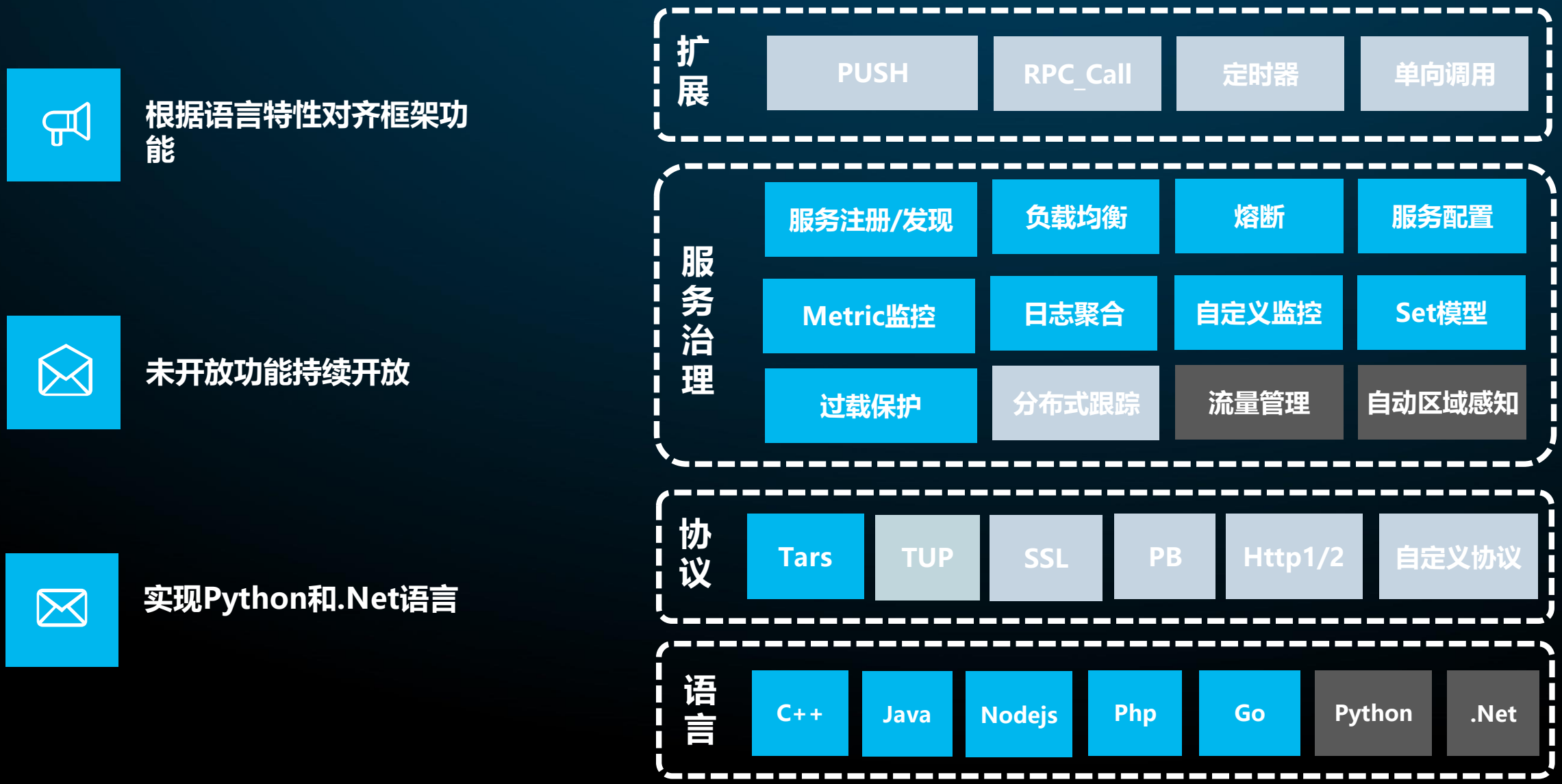
3. Tars未来发展规划

未来发展 – 语言能力对齐与功能开放

根据语言特性对齐框架功能

未开放功能持续开放

实现Python和.Net语言

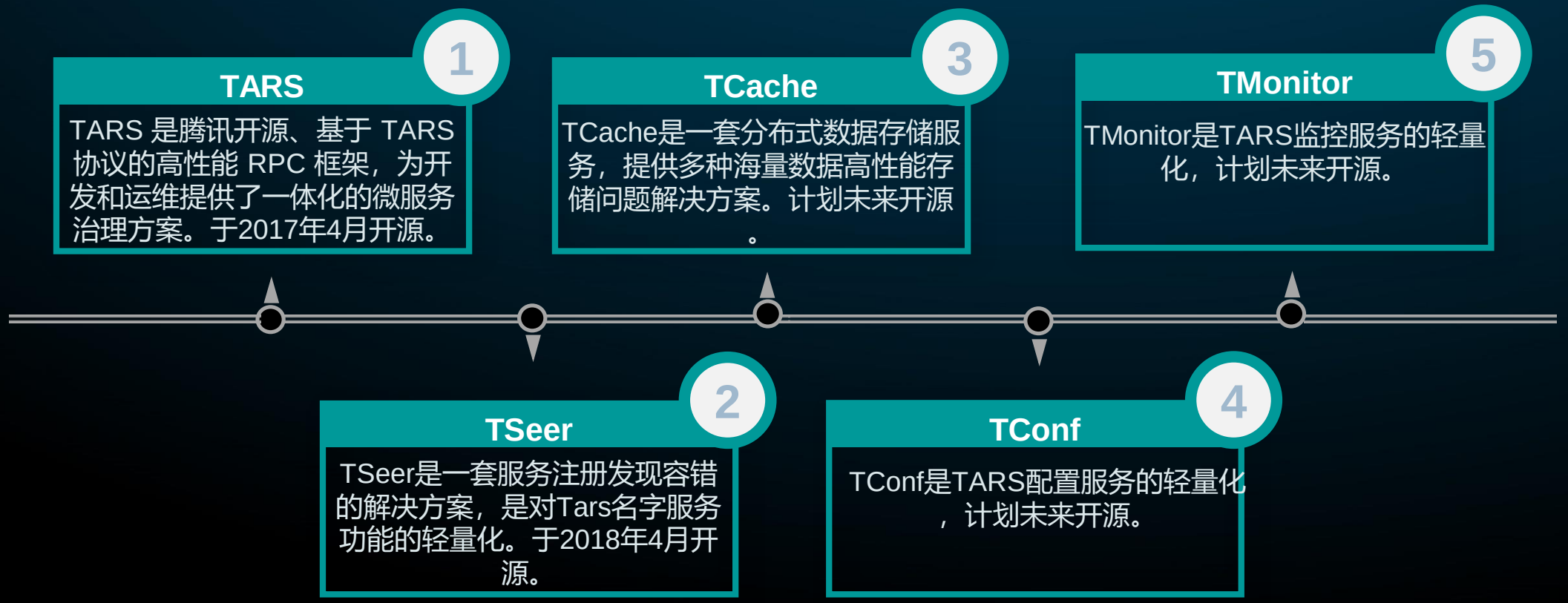


未来发展 – 开源生态结合

- 不封闭造轮子，与社区融合，通过社区来提升自己，并反哺社区，给用户更灵活的选择。

未来发展 – TARS开源生态建设

- **开源不同微服务周边项目，构建微服务生态体系，让用户更方便的打造其微服务架构。**



https://github.com/TarsCloud/Tars