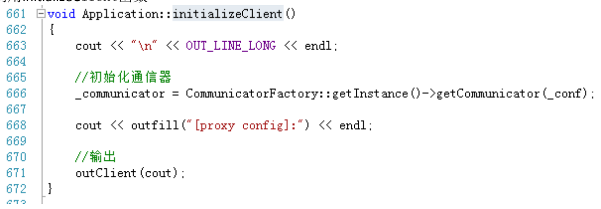
Taf Client

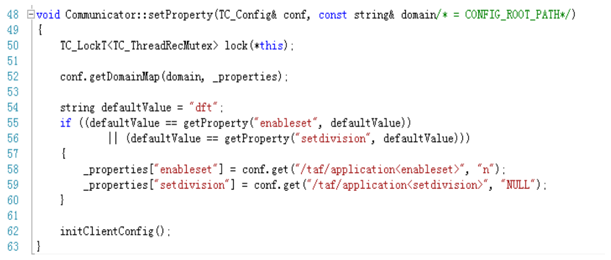
2015年12月20日　　17:01

1. 在程序启动的时候，Application程序的main函数
   a. 调用initializeClient函数



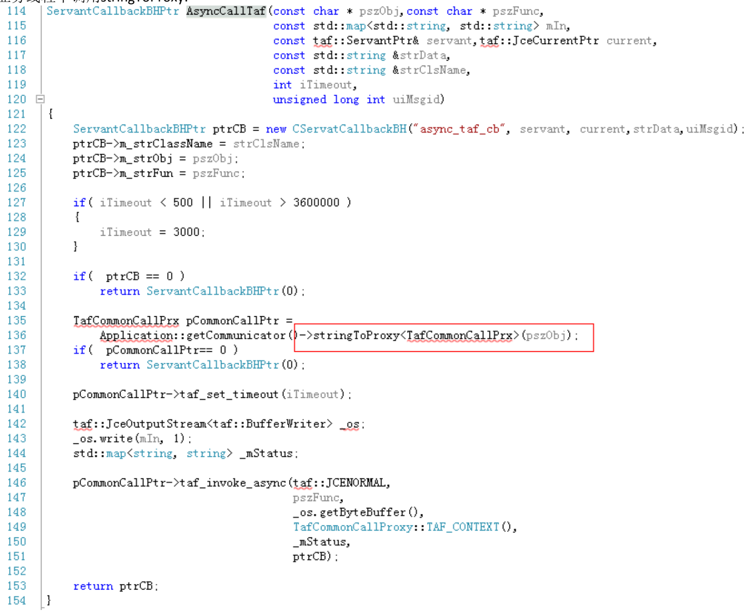
```cpp
void Application::initializeClient()
{
    cout << "\n" << OUT_LINE_LONG << endl;

    //初始化通信器
    _communicator = CommunicatorFactory::getInstance()->getCommunicator(_conf);

    cout << outfill("[proxy config]:") << endl;

    //输出
    outClient(cout);
}
```

   b. 这里会初始化一个通信器，用配置中的内容来初始化通信器的属性（分组属性）



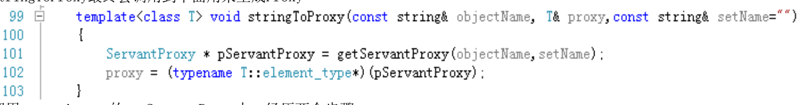
```cpp
void Communicator::setProperty(TC_Config& conf, const string& domain/* = CONFIG_ROOT_PATH*/)
{
    TC_LockT<TC_ThreadRecMutex> lock(*this);

    conf.getDomainMap(domain, _properties);

    string defaultValue = "dft";
    if ((defaultValue == getProperty("enableset", defaultValue))
            || (defaultValue == getProperty("setdivision", defaultValue)))
    {
        _properties["enableset"] = conf.get("/taf/application<enableset>", "n");
        _properties["setdivision"] = conf.get("/taf/application<setdivision>", "NULL");
    }

    initClientConfig();
}
```

2. 业务线程中调用stringToProxy:



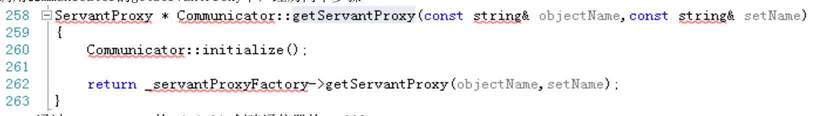
```cpp
ServantCallbackBHPtr AsyncCallTaf(const char * pszObj,const char * pszFunc,
                                  const std::map<std::string, std::string> mIn,
                                  const taf::ServantPtr& servant,taf::JceCurrentPtr current,
                                  const std::string &strData,
                                  const std::string &strClsName,
                                  int iTimeout,
                                  unsigned long int uiMsgid)
{
    ServantCallbackBHPtr ptrCB = new CServatCallbackBH("async_taf_cb", servant, current,strData,uiMsgid);
    ptrCB->m_strClassName = strClsName;
    ptrCB->m_strObj = pszObj;
    ptrCB->m_strFun = pszFunc;

    if( iTimeout < 500 || iTimeout > 3600000 )
    {
        iTimeout = 3000;
    }

    if(  ptrCB == 0 )
        return ServantCallbackBHPtr(0);

    TafCommonCallPrx pCommonCallPtr =
        Application::getCommunicator()->stringToProxy<TafCommonCallPrx>(pszObj);
    if(  pCommonCallPtr== 0 )
        return ServantCallbackBHPtr(0);

    pCommonCallPtr->taf_set_timeout(iTimeout);

    taf::JceOutputStream<taf::BufferWriter> _os;
    _os.write(mIn, 1);
    std::map<string, string> _mStatus;

    pCommonCallPtr->taf_invoke_async(taf::JCENORMAL,
                                     pszFunc,
                                     _os.getByteBuffer(),
                                     TafCommonCallProxy::TAF_CONTEXT(),
                                     _mStatus,
                                     ptrCB);

    return ptrCB;
}
```

   a. stringToProxy最终会调用到下面用来生成Proxy



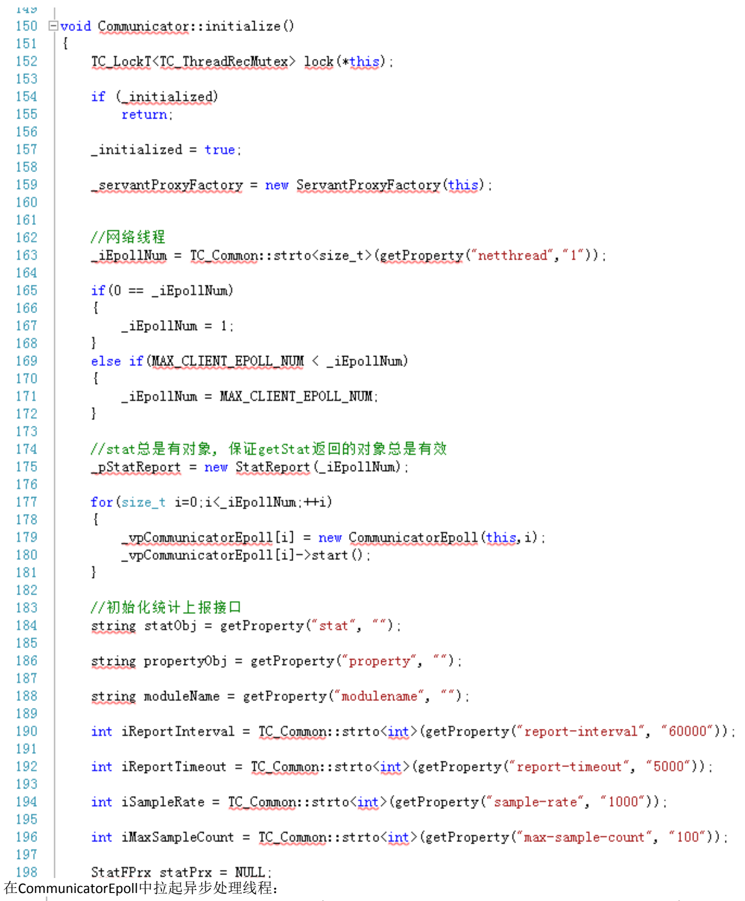
```cpp
    template<class T> void stringToProxy(const string& objectName, T& proxy,const string& setName="")
    {
        ServantProxy * pServantProxy = getServantProxy(objectName,setName);
        proxy = (typename T::element_type*)(pServantProxy);
    }
```

   b. 调用communicator的getServantProxy中，经历两个步骤



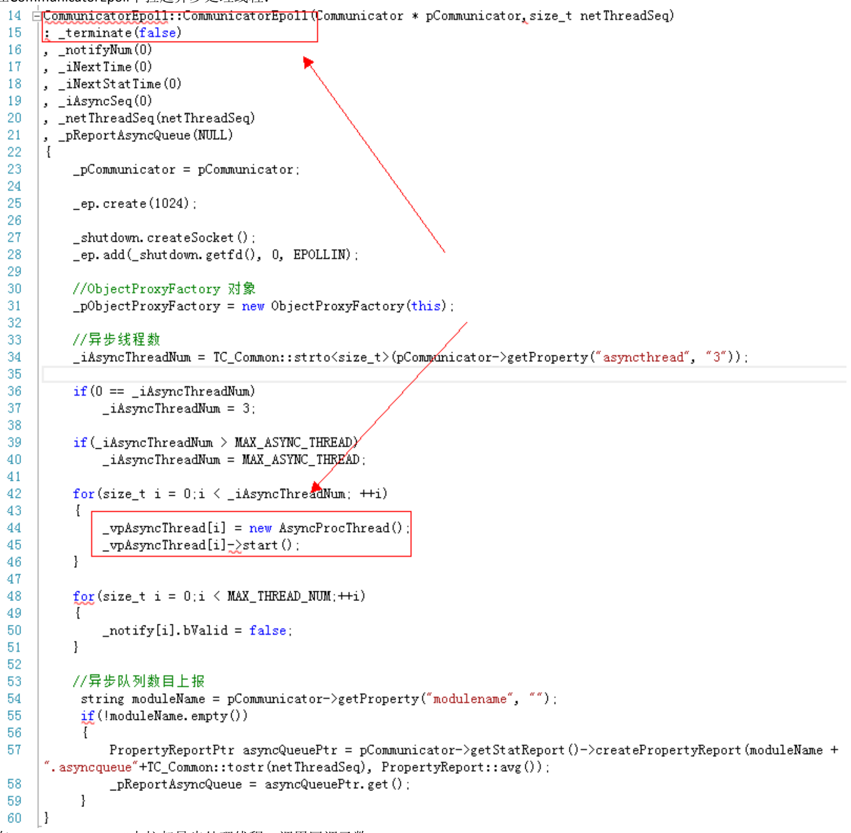
```cpp
ServantProxy * Communicator::getServantProxy(const string& objectName,const string& setName)
{
    Communicator::initialize();

    return _servantProxyFactory->getServantProxy(objectName,setName);
}
```

      i. 通过communicator的initializ创建通信器的epollServer



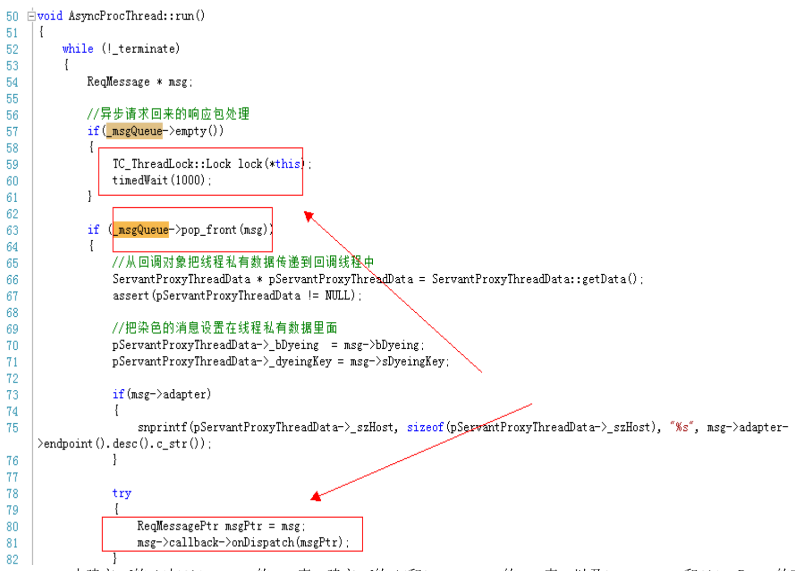
```
void Communicator::initialize()
{
    TC_LockT<TC_ThreadRecMutex> lock(*this);

    if (_initialized)
        return;

    _initialized = true;

    _servantProxyFactory = new ServantProxyFactory(this);


    //网络线程
    _iEpollNum = TC_Common::strto<size_t>(getProperty("netthread","1"));

    if(0 == _iEpollNum)
    {
        _iEpollNum = 1;
    }
    else if(MAX_CLIENT_EPOLL_NUM < _iEpollNum)
    {
        _iEpollNum = MAX_CLIENT_EPOLL_NUM;
    }

    //stat总是有对象, 保证getStat返回的对象总是有效
    _pStatReport = new StatReport(_iEpollNum);

    for(size_t i=0;i<_iEpollNum;++i)
    {
        _vpCommunicatorEpoll[i] = new CommunicatorEpoll(this,i);
        _vpCommunicatorEpoll[i]->start();
    }

    //初始化统计上报接口
    string statObj = getProperty("stat", "");

    string propertyObj = getProperty("property", "");

    string moduleName = getProperty("modulename", "");

    int iReportInterval = TC_Common::strto<int>(getProperty("report-interval", "60000"));

    int iReportTimeout = TC_Common::strto<int>(getProperty("report-timeout", "5000"));

    int iSampleRate = TC_Common::strto<int>(getProperty("sample-rate", "1000"));

    int iMaxSampleCount = TC_Common::strto<int>(getProperty("max-sample-count", "100"));

    StatFPrx statPrx = NULL;
```

ii. 在CommunicatorEpoll中拉起异步处理线程:

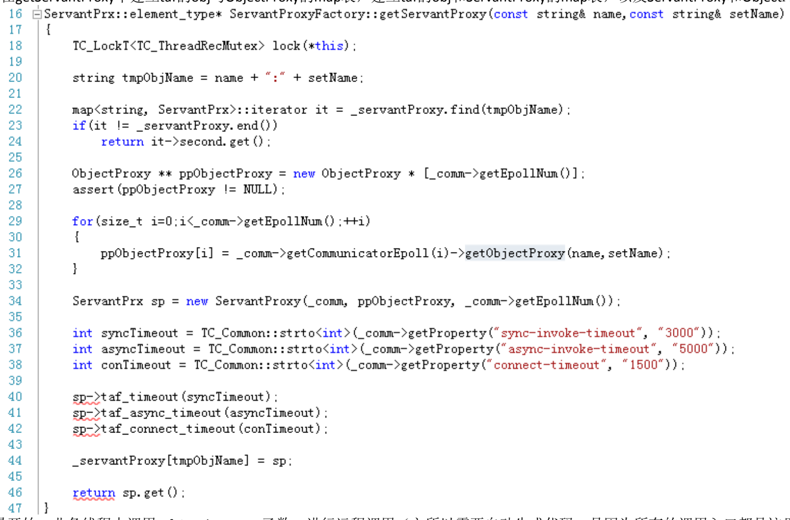


```
CommunicatorEpoll::CommunicatorEpoll(Communicator * pCommunicator,size_t netThreadSeq)
: _terminate(false)
, _notifyNum(0)
, _iNextTime(0)
, _iNextStatTime(0)
, _iAsyncSeq(0)
, _netThreadSeq(netThreadSeq)
, _pReportAsyncQueue(NULL)
{
    _pCommunicator = pCommunicator;

    _ep.create(1024);

    _shutdown.createSocket();
    _ep.add(_shutdown.getfd(), 0, EPOLLIN);

    //ObjectProxyFactory 对象
    _pObjectProxyFactory = new ObjectProxyFactory(this);

    //异步线程数
    _iAsyncThreadNum = TC_Common::strto<size_t>(pCommunicator->getProperty("asyncthread", "3"));

    if(0 == _iAsyncThreadNum)
        _iAsyncThreadNum = 3;

    if(_iAsyncThreadNum > MAX_ASYNC_THREAD)
        _iAsyncThreadNum = MAX_ASYNC_THREAD;

    for(size_t i = 0;i < _iAsyncThreadNum; ++i)
    {
        _vpAsyncThread[i] = new AsyncProcThread();
        _vpAsyncThread[i]->start();
    }

    for(size_t i = 0;i < MAX_THREAD_NUM;++i)
    {
        _notify[i].bValid = false;
    }

    //异步队列数目上报
     string moduleName = pCommunicator->getProperty("modulename", "");
     if(!moduleName.empty())
     {
         PropertyReportPtr asyncQueuePtr = pCommunicator->getStatReport()->createPropertyReport(moduleName +
".asyncqueue"+TC_Common::tostr(netThreadSeq), PropertyReport::avg());
         _pReportAsyncQueue = asyncQueuePtr.get();
     }
}
```

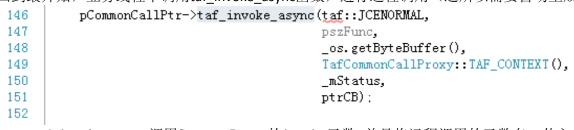
iii. 在CommunicatorEpoll中拉起异步处理线程, 调用回调函数


```
void AsyncProcThread::run()
{
    while (!_terminate)
    {
        ReqMessage * msg;

        //异步请求回来的响应包处理
        if(_msgQueue->empty())
        {
            TC_ThreadLock::Lock lock(*this);
            timedWait(1000);
        }

        if (_msgQueue->pop_front(msg))
        {
            //从回调对象把线程私有数据传递到回调线程中
            ServantProxyThreadData * pServantProxyThreadData = ServantProxyThreadData::getData();
            assert(pServantProxyThreadData != NULL);

            //把染色的消息设置在线程私有数据里面
            pServantProxyThreadData->_bDyeing  = msg->bDyeing;
            pServantProxyThreadData->_dyeingKey = msg->sDyeingKey;

            if(msg->adapter)
            {
                snprintf(pServantProxyThreadData->_szHost, sizeof(pServantProxyThreadData->_szHost), "%s", msg->adapter->endpoint().desc().c_str());
            }

            try
            {
                ReqMessagePtr msgPtr = msg;
                msg->callback->onDispatch(msgPtr);
            }
```

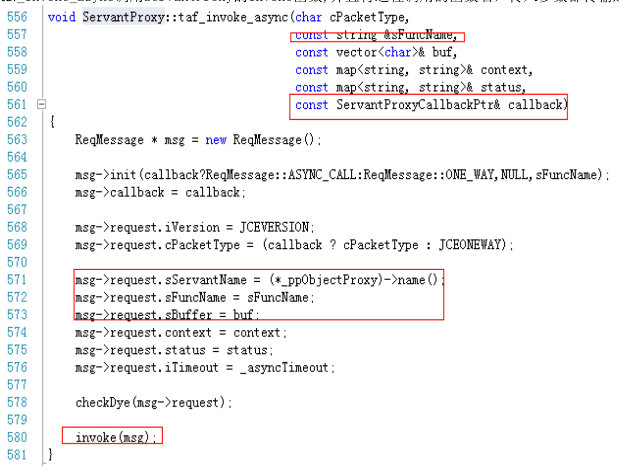
c. 在getServantProxy中建立taf的obj与ObjectProxy的map表，建立taf的obj和ServantProxy的map表，以及ServantProxy和ObjectProxy的对应关系；需要注意的是ObjectProxy的个数是和Communicator的网络线程数目一致的；


```
ServantPrx::element_type* ServantProxyFactory::getServantProxy(const string& name,const string& setName)
{
    TC_LockT<TC_ThreadRecMutex> lock(*this);

    string tmpObjName = name + ":" + setName;

    map<string, ServantPrx>::iterator it = _servantProxy.find(tmpObjName);
    if(it != _servantProxy.end())
        return it->second.get();

    ObjectProxy ** ppObjectProxy = new ObjectProxy * [_comm->getEpollNum()];
    assert(ppObjectProxy != NULL);

    for(size_t i=0;i<_comm->getEpollNum();++i)
    {
        ppObjectProxy[i] = _comm->getCommunicatorEpoll(i)->getObjectProxy(name,setName);
    }

    ServantPrx sp = new ServantProxy(_comm, ppObjectProxy, _comm->getEpollNum());

    int syncTimeout = TC_Common::strto<int>(_comm->getProperty("sync-invoke-timeout", "3000"));
    int asyncTimeout = TC_Common::strto<int>(_comm->getProperty("async-invoke-timeout", "5000"));
    int conTimeout = TC_Common::strto<int>(_comm->getProperty("connect-timeout", "1500"));

    sp->taf_timeout(syncTimeout);
    sp->taf_async_timeout(asyncTimeout);
    sp->taf_connect_timeout(conTimeout);

    _servantProxy[tmpObjName] = sp;

    return sp.get();
}
```

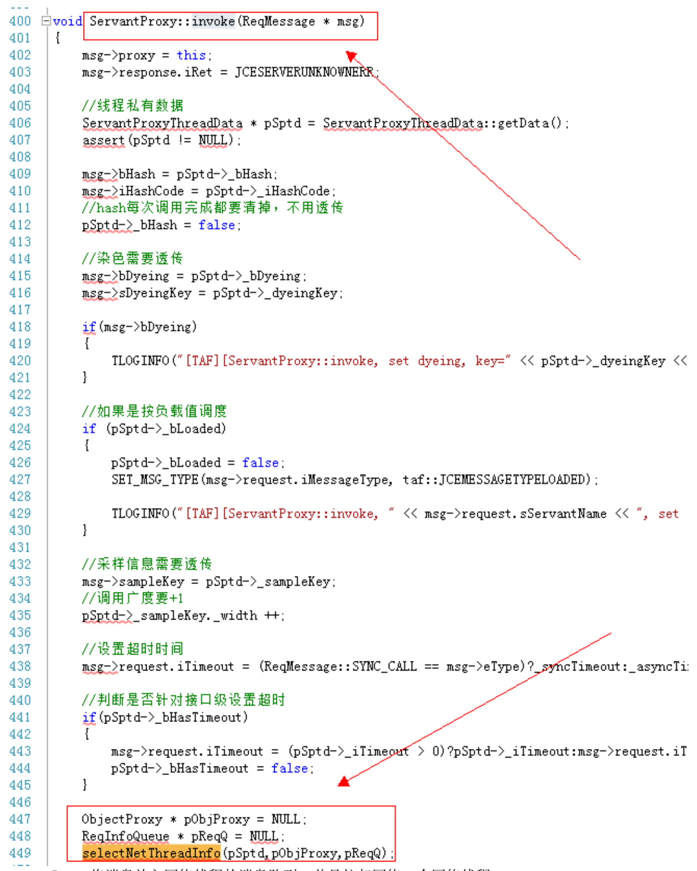
3. 回到最开始，业务线程中调用taf_invoke_async函数，进行远程调用（之所以需要自动生成代码，是因为所有的调用入口都是这里）


```
    pCommonCallPtr->taf_invoke_async(taf::JCENORMAL,
                                     pszFunc,
                                     _os.getByteBuffer(),
                                     TafCommonCallProxy::TAF_CONTEXT(),
                                     _mStatus,
                                     ptrCB);
```

a. taf_invoke_async调用ServantProxy的invoke函数,并且将远程调用的函数名，传入参数都传输过去了；（ppObjcetProxy->name应该是tafObj的名字）

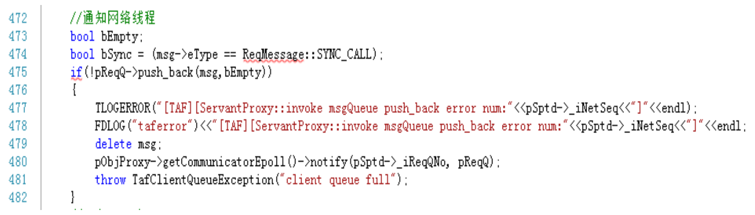

```
void ServantProxy::taf_invoke_async(char cPacketType,
                                    const string &sFuncName,
                                    const vector<char>& buf,
                                    const map<string, string>& context,
                                    const map<string, string>& status,
                                    const ServantProxyCallbackPtr& callback)
{
    ReqMessage * msg = new ReqMessage();

    msg->init(callback?ReqMessage::ASYNC_CALL:ReqMessage::ONE_WAY,NULL,sFuncName);
    msg->callback = callback;

    msg->request.iVersion = JCEVERSION;
    msg->request.cPacketType = (callback ? cPacketType : JCEONEWAY);

    msg->request.sServantName = (*_ppObjectProxy)->name();
    msg->request.sFuncName = sFuncName;
    msg->request.sBuffer = buf;
    msg->request.context = context;
    msg->request.status = status;
    msg->request.iTimeout = _asyncTimeout;

    checkDye(msg->request);

    invoke(msg);
}
```

b. ServantProxy::invoke会轮训获取一个网络ObjectProxy



```
void ServantProxy::invoke(ReqMessage * msg)
{
    msg->proxy = this;
    msg->response.iRet = JCESERVERUNKNOWNERR;

    //线程私有数据
    ServantProxyThreadData * pSptd = ServantProxyThreadData::getData();
    assert(pSptd != NULL);

    msg->bHash = pSptd->_bHash;
    msg->iHashCode = pSptd->_iHashCode;
    //hash每次调用完成都要清掉，不用透传
    pSptd->_bHash = false;

    //染色需要透传
    msg->bDyeing = pSptd->_bDyeing;
    msg->sDyeingKey = pSptd->_dyeingKey;

    if(msg->bDyeing)
    {
        TLOGINFO("[TAF][ServantProxy::invoke, set dyeing, key=" << pSptd->_dyeingKey <<
    }

    //如果是按负载值调度
    if (pSptd->_bLoaded)
    {
        pSptd->_bLoaded = false;
        SET_MSG_TYPE(msg->request.iMessageType, taf::JCEMESSAGETYPELOADED);

        TLOGINFO("[TAF][ServantProxy::invoke, " << msg->request.sServantName << ", set
    }

    //采样信息需要透传
    msg->sampleKey = pSptd->_sampleKey;
    //调用广度要+1
    pSptd->_sampleKey._width ++;

    //设置超时时间
    msg->request.iTimeout = (ReqMessage::SYNC_CALL == msg->eType)?_syncTimeout:_asyncTi

    //判断是否针对接口级设置超时
    if(pSptd->_bHasTimeout)
    {
        msg->request.iTimeout = (pSptd->_iTimeout > 0)?pSptd->_iTimeout:msg->request.iT
        pSptd->_bHasTimeout = false;
    }

    ObjectProxy * pObjProxy = NULL;
    ReqInfoQueue * pReqQ = NULL;
    selectNetThreadInfo(pSptd,pObjProxy,pReqQ);
```

c. ServantProxy将消息放入网络线程的消息队列，并且拉起网络一个网络线程



```
    //通知网络线程
    bool bEmpty;
    bool bSync = (msg->eType == ReqMessage::SYNC_CALL);
    if(!pReqQ->push_back(msg,bEmpty))
    {
        TLOGERROR("[TAF][ServantProxy::invoke msgQueue push_back error num:"<<pSptd->_iNetSeq<<"]"<<endl);
        FDLOG("taferror")<<"[TAF][ServantProxy::invoke msgQueue push_back error num:"<<pSptd->_iNetSeq<<"]"<<endl;
        delete msg;
        pObjProxy->getCommunicatorEpoll()->notify(pSptd->_iReqQNo, pReqQ);
        throw TafClientQueueException("client queue full");
    }
```