**cc**

**算法总结**

**算法参数**

|  |
| --- |
| C++ cc.explore.time //探索阶段持续时间 cc.fast.time //快速启动阶段时间 cc.fast.inc //速率单步增加量 cc.fast.range.min //快启动条件 cc.fast.range.max cc.fast.type //快启动方式 per-RTT or per-ACK  cc.seq cc.time cc.packet\_size cc.switch\_bandwidth  cc.pid.rtt\_target cc.pid.rtt\_min cc.pid.e cc.pid.eold cc.pid.eavg cc.pid.w cc.pid.range.min cc.pid.range.max cc.pid.target\_qlen  cc.lstm.rtt cc.lstm.rtt\_smooth cc.lstm.nn  cc.minrate cc.maxrate cc.rate |

**算法流程**

**TCP建立，算法参数初始化**

|  |
| --- |
| Plain Text cc.explore.time = 1 cc.minrate = 1Gbps cc.maxrate = 100Gbps cc.rate = cc.minrate  cc.fast.time= 25 cc.fast.inc =(cc.maxrate - cc.minrate)/cc.fast.time cc.fast.type = "per-ACK"  cc.pid.e = 0 cc.pid.eold = 0 cc.pid.eavg = 0 cc.pid.range.min = 0.1 cc.pid.range.max = 2 cc.pid.target\_qlen = 12  cc.seq = TCP.seq cc.time = 0  cc.packet\_size = 1000B cc.switch\_bandwidth = 100Gbps  cc.pid.w = conf.w cc.lstm.nn = conf.nn |

**接收到第一个数据包ACK，计算出RTT0,初始化参数**

|  |
| --- |
| Plain Text cc.lstm.rtt[0] = RTT0 cc.lstm.rtt[1] = RTT0 cc.lstm.rtt[2] = RTT0 cc.lstm.rtt\_smooth = RTT0 cc.time = 1  cc.pid.rtt\_min = RTT0 update\_target(RTT) cc.fast.range.min = cc.pid.rtt\_min cc.fast.range.max = cc.pid.rtt\_min + 0.05\*(cc.pid.rtt\_target - cc.pid.rtt\_min) |

**RTT目标更新**

|  |
| --- |
| Bash def update\_target(RTT):  if(RTT < cc.pid.rtt\_min)  cc.pid.rtt\_min = RTT  cc.pid.rtt\_target = cc.pid.rtt\_min + (cc.packet\_size\*cc.pid.target\_qlen)/cc.switch\_bandwidth |

**算法执行**

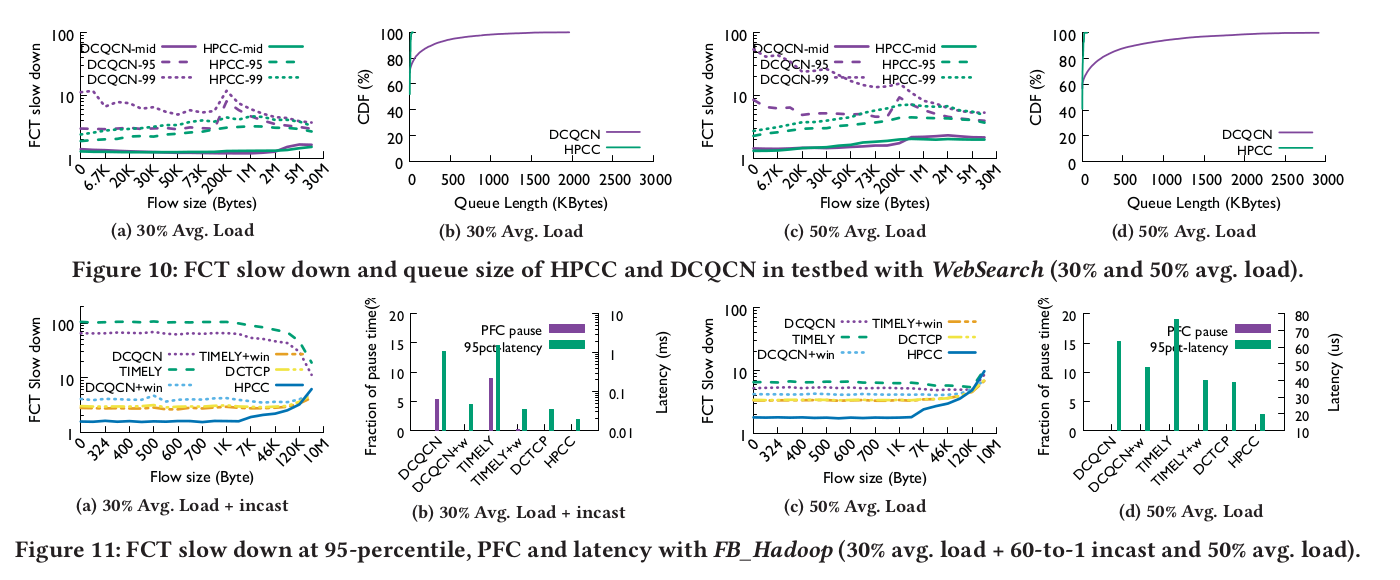
|  |
| --- |
| Plain Text //探索阶段 while cc.time < cc.explore.time  cc.rate = cc.minrate //启动阶段  while cc.time < cc.explore.time + cc.fast.time  if cc.fast.type == "per-RTT":  (RTT事件触发 && RTT>=cc.fast.range.min && RTT=<cc.fast.range.max）  cc.rate += cc.fast.inc  if cc.fast.type == "per-ACK":  (ACK事件触发 && RTT>=cc.fast.range.min && RTT=<cc.fast.range.max)  cc.rate += cc.fast.inc  (若不满足启动条件 && RTT事件触发)  cc.rate \*=LSTM\_PID()+1 //速率控制阶段  while cc.time > cc.explore.time + cc.fast.time  cc.rate \*=LSTM\_PID()+1 |

LSTM+PID模块：

|  |
| --- |
| Bash 计算RTT 更新 cc.lstm.rtt以及cc.lstm.rtt\_smooth 计算LSTM网络输入K 计算LSTM输出 def LSTM(K,cc.lstm.nn)  INIT(cc.lstm.nn.h,cc.lstm.nn.c)  CELL(it,cc.lstm.nn.wi[0:16],cc.lstm.nn.wh[0:16][0:16],cc.lstm.nn.bi[0:16],cc.lstm.nn.bh[0:16],K,cc.lstm.nn.h,cc.lstm.nn.sigmod)  CELL(fi,cc.lstm.nn.wi[16:32],cc.lstm.nn.wh[16:32][0:16],cc.lstm.nn.bi[16:32],cc.lstm.nn.bh[16:32],K,cc.lstm.nn.h,cc.lstm.nn.sigmod)  CELL(gt,cc.lstm.nn.wi[32:48],cc.lstm.nn.wh[32:48][0:16],cc.lstm.nn.bi[32:48],cc.lstm.nn.bh[32:48],K,cc.lstm.nn.h,cc.lstm.nn.tanh)  CELL(ot,cc.lstm.nn.wi[48:64],cc.lstm.nn.wh[48:64][0:16],cc.lstm.nn.bi[48:64],cc.lstm.nn.bh[48:64],K,cc.lstm.nn.h,cc.lstm.nn.sigmod)  //更新cc.lstm.nn.c以及cc.lstm.nn.h  cc.lstm.nn.c = fi\*cc.lstm.nn.c+it\*gt  cc.lstm.nn.h = ot\*cc.lstm.nn.tanh(cc.lstm.nn.c)  return cc.lstm.nn.h\*cc.lstm.nn.linearw+cc.lstm.nn.linearb 计算PID输出 def PID(cc.pid,rtt\_prediction)  c = cc.pid.w.p\*(rtt\_prediction-cc.pid.rtt\_target)\*cc.pid.w.i\*cc.pid.eavg+cc.pid.w.d\*(cc.pid.e-cc.pid.eold)  if cc.pid.range.min > c:  c = cc.pid.range.min  else if cc.pid.range.max < c:  c = cc.pid.range.max  return c |

**算法测试设计**

**NS3仿真测试**



**流量设计：**

**流分布数据：**

A.WebSearch

WebSearch\_distribution.txt

B.FB\_Hadoop

FbHdp\_distribution.txt

**流产生时长：**0.1s

使用流生成工具：

|  |
| --- |
| Plain Text python traffic\_gen.py -c WebSearch\_distribution.txt -n 320 -l 0.3 -b 100G -t 0.1 |

**效果统计：**

A FCT slow down和Queue Length(30%Avg.Load,50%Avg.Load)

B FCT slow down和Latency、PFC(30%Avg.Load + incast,50%Avg.Load)

C ECN标记概率

D 吞吐量

FCT slow down 统计：

|  |
| --- |
| Bash 修改平台analysis/fct\_analysis.py中CCs字段 python fct\_analysis -s 1 |

Queue Length统计：

|  |
| --- |
| Bash 设置 --enable\_tr 1观察文件跟踪结果,其中node>320为交换机，第四个参数为队列长度 ./trace\_reader <.tr file> 2000055540 n:338 4:3 100608 Enqu ecn:0 0b00d101 0b012301 10000 100 U 161000 0 3 1048(1000) |

PFC暂停时间统计：

|  |
| --- |
| Bash 仿真结束读取pfc.txt,统计其中的pfc暂停发送（只有TIMELY和DCQCN触发了PFC机制） Simulator::Now().GetTimeStep(), dev->GetNode()->GetId(), dev->GetNode()->GetNodeType(), dev->GetIfIndex(), type  时间戳，node id,设备node类型（1交换机，other终端），设备接口INDEX，PFC报文类型（1：拥赛停止发送，0：解除拥赛允许发送） |

Latency时延统计

|  |
| --- |
| Bash 设置ih.ts字段，打印每个报文的时间辍 |

**INCAST：**

1. 根据2%总体网络负载计算方法，得出INCAST发生次数

|  |
| --- |
| Bash 节点数：320 时间：0.1s INCAST场景：60个节点同时向一个目的地址发送500KB数据 节点带宽：100Gbps 发生次数：2%\*320\*100Gbps\*0.1/(60\*500KB) = 267次 前后两次平均间隔时间：T = 0.1s/267 = 374.5us |

1. 插入方法

使用泊松分布（或指数分布）生成随即INCAST，并插入flow.txt文件

flow.txt

**测试算法：**

A DCQCN PIDNN

B DCQCN DCTCP TIMELY HPCC PIDNN

|  |
| --- |
| Bash --cc 选择拥塞控制算法 --trace 选择流文件 --bw 节点NIC带宽 --topo拓扑文件 --enable\_tr 包跟踪 python run.py --cc hp --trace flow --bw 100 --topo fat --enable\_tr 1 |

**真实网络测试**

|  |
| --- |
| 交换机等转发设备使用OVS实现  拓扑问题：fat拓扑需要第一层交换机有400Gbps的转发速率，第二层100Gbps转发速率 |

方案1：使用f-stack，修改其freeBSD中的TCP协议栈，基于C++实现

https://github.com/F-Stack/f-stack

目前f-stack实现了BBR、DCTCP、CUBIC、vegas多种拥塞控制算法

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
| C++ struct tcp\_function\_set {  char function\_set\_name[TCP\_FUNCTION\_NAME\_LEN\_MAX];  uint32\_t pcbcnt; }; #define TCP\_FUNCTION\_BLK 8192 /\* Set the tcp function pointers to the specified stack \*/ struct dctcp {  uint32\_t bytes\_ecn; /\* # of marked bytes during a RTT \*/  uint32\_t bytes\_total; /\* # of acked bytes during a RTT \*/  int alpha; /\* the fraction of marked bytes \*/  int ce\_prev; /\* CE state of the last segment \*/  tcp\_seq save\_sndnxt; /\* end sequence number of the current window \*/  int ece\_curr; /\* ECE flag in this segment \*/  int ece\_prev; /\* ECE flag in the last segment \*/  uint32\_t num\_cong\_events; /\* # of congestion events \*/ }; |

|  |
| --- |
| 存在的问题：   1. 修改协议栈代码工作量大 2. 对于f-stack不熟悉，在部署上可能会遇到很多问题 |

**算法部署**

|  |
| --- |
| C++ 1.算法结构体定义 struct lstm\_pid { }; 2.使用FreeBSD操作系统中一个宏定义MALLOC\_DEFINE，用于定义新的内存分配器， 通过为特定的数据结构或算法指定专用的内存分配器，可以更好地管理内存使用情况，提高程序性能和可靠性。 static MALLOC\_DEFINE(M\_dctcp, "dctcp data",  "Per connection data required for the dctcp algorithm"); 3.复写cc\_algo结构体 struct cc\_algo {  char name[TCP\_CA\_NAME\_MAX];  /\* Init global module state on kldload. \*/  int (\*mod\_init)(void);  /\* Cleanup global module state on kldunload. \*/  int (\*mod\_destroy)(void);  /\* Init CC state for a new control block. \*/  int (\*cb\_init)(struct cc\_var \*ccv);  /\* Cleanup CC state for a terminating control block. \*/  void (\*cb\_destroy)(struct cc\_var \*ccv);  /\* Init variables for a newly established connection. \*/  void (\*conn\_init)(struct cc\_var \*ccv);  /\* Called on receipt of an ack. \*/  void (\*ack\_received)(struct cc\_var \*ccv, uint16\_t type);  /\* Called on detection of a congestion signal. \*/  void (\*cong\_signal)(struct cc\_var \*ccv, uint32\_t type);  /\* Called after exiting congestion recovery. \*/  void (\*post\_recovery)(struct cc\_var \*ccv);  /\* Called when data transfer resumes after an idle period. \*/  void (\*after\_idle)(struct cc\_var \*ccv);  /\* Called for an additional ECN processing apart from RFC3168. \*/  void (\*ecnpkt\_handler)(struct cc\_var \*ccv);  /\* Called for {get|set}sockopt() on a TCP socket with TCP\_CCALGOOPT. \*/  int (\*ctl\_output)(struct cc\_var \*, struct sockopt \*, void \*);  STAILQ\_ENTRY (cc\_algo) entries; }; struct cc\_algo lstm\_pid\_cc\_algo = {} 4.声明和注册 DECLARE\_CC\_MODULE(lstm\_pid, &dlstm\_pid\_cc\_algo); MODULE\_VERSION(lstm\_pid, 1); DECLARE\_CC\_MODULE()是一个宏定义，用于声明名为 lstm\_pid 的拥塞控制模块，并注册相应的 TCP 拥塞控制算法。 在代码中，&lstm\_pid\_cc\_algo 是指向 struct cc\_algo 结构体的指针，该结构体中包含了实现拥塞控制算法的相关函数。 通过将该指针作为参数传递给 DECLARE\_CC\_MODULE 宏定义，可以将拥塞控制算法与 lstm\_pid 模块关联起来 5.实现lstm\_pid\_cc\_algo内部各个函数，将算法结构体struct lstm\_pid储存到struct cc\_var \*ccv中cc\_data字段 6.f-stack/freebsd/netinet/tcp\_var.h文件内struct tcpcb结构体储存了有关TCP连接的信息 可以用CCV(ccv, 变量名)获取相应的变量数据 7.在fstack.conf文件中设置拥塞控制算法 net.inet.tcp.cc.algorithm=cubic |

**f-stack搭建**

|  |
| --- |
| C++ 从github下载f-stack源码 mkdir -p /data/f-stack git clone https://github.com/F-Stack/f-stack.git /data/f-stack 替换dpdk里igb\_uio模块 git clone http://dpdk.org/git/dpdk-kmods cp -r ./dpdk-kmods/linux/igb\_uio ./dpdk/kernel/linux/  配置环境 pip3 install pyelftools --upgrade apt-get install git gcc openssl libssl-dev linux-headers-$(uname -r) bc libnuma1 libnuma-dev libpcre3 libpcre3-dev zlib1g-dev python 编译dpdk(建议独自去dpdk官网编译) meson -Denable\_kmods=true build $ ninja -C build $ ninja -C build install 配置pkg-config get https://pkg-config.freedesktop.org/releases/pkg-config-0.29.2.tar.gz $ tar xzvf pkg-config-0.29.2.tar.gz $ cd pkg-config-0.29.2 $ ./configure --with-internal-glib $ make $ make install $ mv /usr/bin/pkg-config /usr/bin/pkg-config.bak $ ln -s /usr/local/bin/pkg-config /usr/bin/pkg-config 编译f-stack lib export FF\_PATH=/data/f-stack export PKG\_CONFIG\_PATH=/usr/lib64/pkgconfig:/usr/local/lib64/pkgconfig:/usr/lib/pkgconfig PKG\_CONFIG\_PATH可能报错 查找目录pkgconfig下的libdpdk.pc文件位置（/data/f-stack/dpdk/build/meson-private/libdpdk.pc） find / -name "libdpdk.pc" 将结果替换为PKG\_CONFIG\_PATH cd /data/f-stack/lib make 编译Nginx、Redis、f-stack tools cd ../app/nginx-1.16.1 ./configure --prefix=/usr/local/nginx\_fstack --with-ff\_module make make install https://www.cnblogs.com/first-semon/p/15190202.html root@zjp-System-Product-Name:sbin# ps a | grep nginx  6997 pts/0 S+ 0:00 grep --color=auto nginx  cd app/redis-6.2.6/deps/jemalloc ./autogen.sh cd ../redis-6.2.6 make  cd ../../tools make 运行测试样例 cd ../examples make |

f-stack.md

**f-stack测试样例**

|  |
| --- |
| Bash #设置大页 echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr\_hugepages echo 1024 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr\_hugepages #echo 1024 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr\_hugepages #mkdir /mnt/huge mount -t hugetlbfs nodev /mnt/huge #Close ASLR echo 0 > /proc/sys/kernel/randomize\_va\_space #加载igb\_uio模块 modprobe uio insmod /data/f-stack/dpdk/build/kernel/linux/igb\_uio/igb\_uio.ko insmod /data/f-stack/dpdk/build/kernel/linux/kni/rte\_kni.ko carrier=on #配置f-stack环境变量 export FF\_PATH=/data/f-stack export PKG\_CONFIG\_PATH="/usr/lib64/pkgconfig:/usr/local/lib64/pkgconfig:/usr/lib/pkgconfig:/usr/local/lib/x86\_64-linux-gnu/pkgconfig/" #dpdk绑定（dpdk-devbind.py） ifconfig ens37 down dpdk-devbind.py -b igb\_uio ens37 cd /data/f-stack test/server --conf config.ini --proc-type=primary --proc-id=0 example/tcp --conf config.ini --proc-type=primary --proc-id=0 example/tcp\_recv --conf config.ini --proc-type=primary --proc-id=0 ps aux | grep client dpdk-devbind.py -u 0000:00:03.0 dpdk-devbind.py -b igb\_uio 0000:00:03.0 |

linux查找并关闭某个端口进程

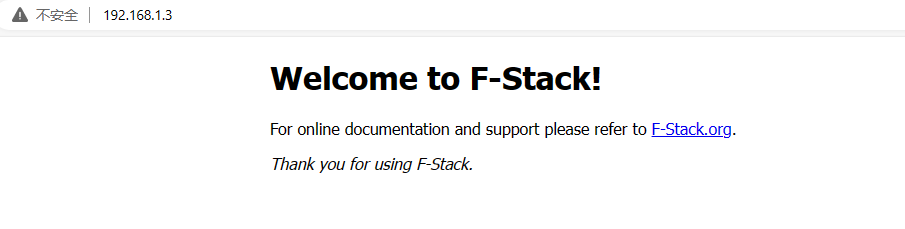
https://blog.csdn.net/qq\_63283137/article/details/128828119

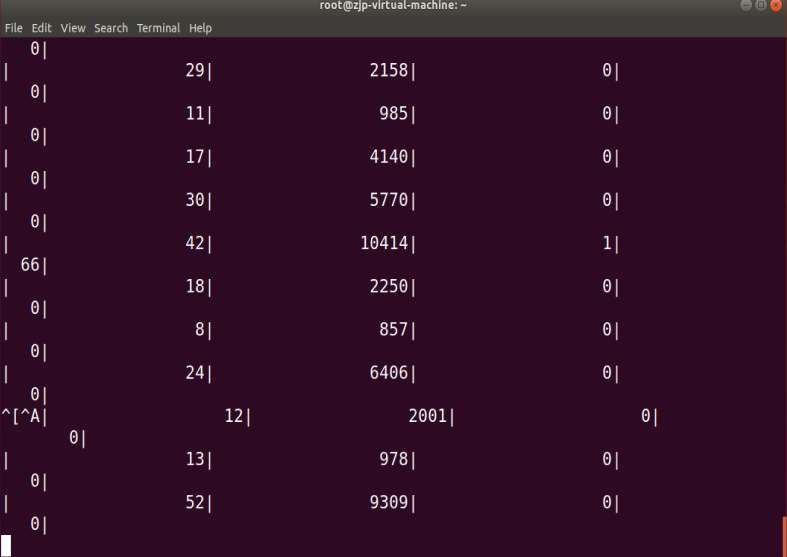
windows10添加静态arp

|  |
| --- |
| Plain Text 进入管理员模式检查网卡接口IDX  netsh i i show in  添加arp   netsh -c "i i" add neighbors IDX IP MAC |

**f-stack测试样例通过**

|  |
| --- |
| C++ [dpdk]: lcore\_mask=1 [dpdk]: channel=4 [dpdk]: promiscuous=1 [dpdk]: numa\_on=1 [dpdk]: tx\_csum\_offoad\_skip=0 [dpdk]: tso=0 [dpdk]: vlan\_strip=1 [dpdk]: idle\_sleep=0 [dpdk]: pkt\_tx\_delay=100 [dpdk]: symmetric\_rss=0 [dpdk]: port\_list=0 [dpdk]: nb\_vdev=0 [dpdk]: nb\_bond=0 [pcap]: enable=0 [pcap]: snaplen=96 [pcap]: savelen=16777216 [pcap]: savepath=. [port0]: addr=192.168.1.2 [port0]: netmask=255.255.255.0 [port0]: broadcast=192.168.1.255 [port0]: gateway=192.168.1.1 [freebsd.boot]: hz=100 [freebsd.boot]: fd\_reserve=1024 [freebsd.boot]: kern.ipc.maxsockets=262144 [freebsd.boot]: net.inet.tcp.syncache.hashsize=4096 [freebsd.boot]: net.inet.tcp.syncache.bucketlimit=100 [freebsd.boot]: net.inet.tcp.tcbhashsize=65536 [freebsd.boot]: kern.ncallout=262144 [freebsd.boot]: kern.features.inet6=1 [freebsd.sysctl]: kern.ipc.somaxconn=32768 [freebsd.sysctl]: kern.ipc.maxsockbuf=16777216 [freebsd.sysctl]: net.link.ether.inet.maxhold=5 [freebsd.sysctl]: net.inet.tcp.fast\_finwait2\_recycle=1 [freebsd.sysctl]: net.inet.tcp.sendspace=16384 [freebsd.sysctl]: net.inet.tcp.recvspace=8192 [freebsd.sysctl]: net.inet.tcp.cc.algorithm=cubic [freebsd.sysctl]: net.inet.tcp.sendbuf\_max=16777216 [freebsd.sysctl]: net.inet.tcp.recvbuf\_max=16777216 [freebsd.sysctl]: net.inet.tcp.sendbuf\_auto=1 [freebsd.sysctl]: net.inet.tcp.recvbuf\_auto=1 [freebsd.sysctl]: net.inet.tcp.sendbuf\_inc=16384 [freebsd.sysctl]: net.inet.tcp.sack.enable=1 [freebsd.sysctl]: net.inet.tcp.blackhole=1 [freebsd.sysctl]: net.inet.tcp.msl=2000 [freebsd.sysctl]: net.inet.tcp.delayed\_ack=1 [freebsd.sysctl]: net.inet.tcp.rfc1323=1 [freebsd.sysctl]: net.inet.udp.blackhole=1 [freebsd.sysctl]: net.inet.ip.redirect=0 [freebsd.sysctl]: net.inet.ip.forwarding=0 [freebsd.sysctl]: net.inet6.ip6.auto\_linklocal=1 [freebsd.sysctl]: net.inet6.ip6.accept\_rtadv=2 [freebsd.sysctl]: net.inet6.icmp6.rediraccept=1 [freebsd.sysctl]: net.inet6.ip6.forwarding=0 [freebsd.sysctl]: net.inet.tcp.functions\_default=freebsd [freebsd.sysctl]: net.inet.tcp.hpts.skip\_swi=1 [freebsd.sysctl]: net.inet.tcp.hpts.minsleep=250 [freebsd.sysctl]: net.inet.tcp.hpts.maxsleep=51200 f-stack -c1 -n4 --proc-type=primary  EAL: Detected CPU lcores: 20 EAL: Detected NUMA nodes: 1 EAL: Detected static linkage of DPDK EAL: Multi-process socket /var/run/dpdk/rte/mp\_socket EAL: Selected IOVA mode 'PA' EAL: No available 1048576 kB hugepages reported EAL: VFIO support initialized EAL: Probe PCI driver: net\_e1000\_igb (8086:1521) device: 0000:07:00.3 (socket 0) TELEMETRY: No legacy callbacks, legacy socket not created lcore: 0, port: 0, queue: 0 create mbuf pool on socket 0 create ring:dispatch\_ring\_p0\_q0 success, 2047 ring entries are now free! Port 0 MAC: a0 36 9f 05 bd bb Port 0 modified RSS hash function based on hardware support,requested:0x2003ffffc configured:0x38d34 RX checksum offload supported TX ip checksum offload supported TX TCP&UDP checksum offload supported TSO is disabled port[0]: rss table size: 128 set port 0 to promiscuous mode ok  Checking link status.................done Port 0 Link Up - speed 100 Mbps - full-duplex link\_elf\_lookup\_symbol: missing symbol hash table link\_elf\_lookup\_symbol: missing symbol hash table Timecounters tick every 10.000 msec WARNING: Adding ifaddrs to all fibs has been turned off by default. Consider tuning net.add\_addr\_allfibs if needed Attempting to load tcp\_bbr tcp\_bbr is now available TCP Hpts created 1 swi interrupt threads and bound 0 to cpus Timecounter "ff\_clock" frequency 100 Hz quality 1 TCP\_ratelimit: Is now initialized f-stack-0: No addr6 config found. f-stack-0: Ethernet address: a0:36:9f:05:bd:bb |



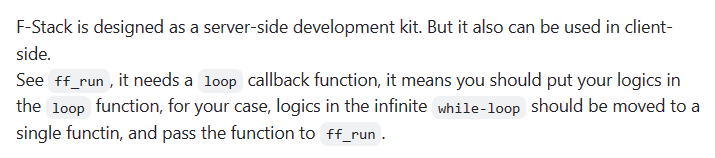


**f-stack测试**



|  |
| --- |
| C++ dpdk-learn https://blog.csdn.net/qq\_38051998/article/details/126667346?spm=1001.2014.3001.5502 F-Stack常用配置参数介绍 https://mp.weixin.qq.com/s/uVQjwPwf5vQeSgGMN5AZMw    https://asiocity.github.io/2020/09/20/f-stack-usage/  f-stack多进程 https://blog.csdn.net/shaoyunzhe/article/details/73498685 |

**f-stack**



**ff接口（API）**

|  |
| --- |
| C++ F-Stack（FF）是一种基于DPDK的高性能网络框架。 FF API提供标准的Kqueue/Epoll接口和微线程框架（SPP）。 为了方便各种服务更简单、更快地使用F-Stack，F-Stack已与Nginx和Redis集成  头文件ff\_api.h定义了api，在使用F-Stack时，应使用这些api替换系统调用。 |

**初始化**

|  |
| --- |
| C++ ff\_init int ff\_init(const char \*conf, int argc, char \* const argv[]); conf:Profile path argv：-c <coremask>,the coremask parameters can cover the coremask in configuration file 核心掩码参数可以覆盖配置文件中的核心掩码(初始化F-Stack，包括DPDK/FreeBSD网络堆栈等。) 运行ff\_run void ff\_run(loop\_func\_t loop, void \*arg); loop是一个回调函数，服务逻辑由用户实现，并由F-Stack的每次轮询调用。 |

**控制**

|  |
| --- |
| C++ 控制API ff\_fcntl fcntl（）对打开的文件描述符fd执行下面描述的操作之一。 ff\_sysctl ff\_sysctl用于在运行时修改内核参数。但是，只有在F-Stack启动之前才支持它。 ff\_ioctl ioctl（）函数操作特殊文件的底层设备参数 |

**网络**

|  |
| --- |
| C++ ff\_socket ff\_socket为通信创建一个端点，并返回一个引用该端点的文件描述符 ff\_setsockopt & ff\_getsockopt getsockopt（）和setstockopt（）操作由文件描述符sockfd表示的套接字的选项。 socketpair（）调用在指定类型的给定域中创建一对未命名的连接套接字，并使用可选的给定协议。 套介子操作函数 int ff\_listen(int s, int backlog); int ff\_bind(int s, const struct linux\_sockaddr \*addr, socklen\_t addrlen); int ff\_accept(int s, struct linux\_sockaddr \*addr, socklen\_t \*addrlen); int ff\_connect(int s, const struct linux\_sockaddr \*name, socklen\_t namelen); int ff\_close(int fd); int ff\_shutdown(int s, int how); ff\_getpeername（）返回连接到套接字sockfd的对等端的地址，位于addr指向的缓冲区中。 ff\_getsockname（）在addr指向的缓冲区中返回套接字sockfd绑定到的当前地址。 ff\_read() & ff\_readv() ff\_send() & ff\_sendto() & ff\_sendmsg() ff\_recv() & ff\_recvfrom() & ff\_recvmsg() ff\_select()允许程序监视多个文件描述符，等待一个或多个文件描述符为某类I/O操作“准备就绪”（例如，可能的输入）。 ff\_poll()等待文件描述符上的事件。  struct kevent {  \_\_uintptr\_t ident; /\* identifier for this event \*/  short filter; /\* filter for event \*/  unsigned short flags; /\* action flags for kqueue \*/  unsigned int fflags; /\* filter flag value \*/  \_\_int64\_t data; /\* filter data value \*/  void \*udata; /\* opaque user data identifier \*/  \_\_uint64\_t ext[4]; /\* extensions \*/ }; |

**队列**

|  |
| --- |
| C++ ff\_kqueue（）系统调用提供了一种通用方法，用于根据称为过滤器的少量内核代码的结果，在事件发生或条件成立时通知用户 |

**循环**

|  |
| --- |
| C++ ff\_epoll\_create()返回一个引用新epoll实例的文件描述符。 ff\_epoll\_ctl()这个系统调用对文件描述符epfd引用的epoll（7）实例执行控制操作 |

**微线程**

|  |
| --- |
| C++ UDP send/recv interface mt\_udpsendrcv() tcp send/recv interface mt\_tcpsendrcv() mt\_tcpsendrcv\_ex() mt\_tcpsendrcv() enum MT\_TCP\_CONN\_TYPE {  MT\_TCP\_SHORT = 1,  MT\_TCP\_LONG = 2,  MT\_TCP\_SHORT\_SNDONLY = 3,  MT\_TCP\_LONG\_SNDONLY = 4,  MT\_TCP\_BUTT }; |

**Dispatch API**

|  |
| --- |
| C++ ff\_regist\_packet\_dispatcher 注册数据包分发功能。 |

