第五部分

性能优化

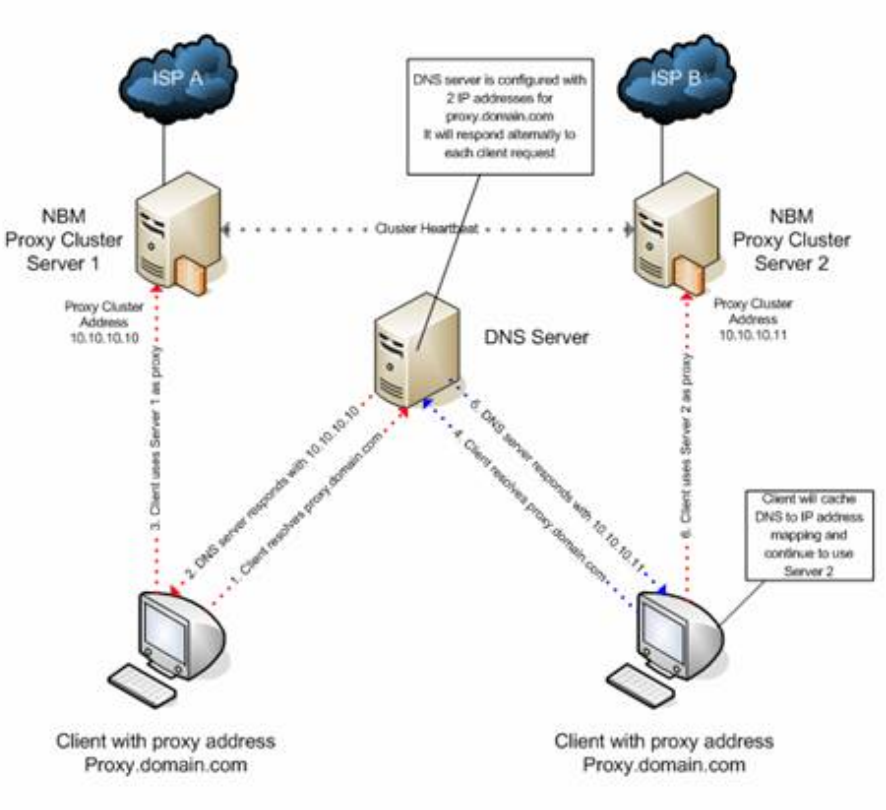
极客时间

扫码试看/订阅

《Nginx 核心知识100讲》

优化方法论

- **从软件层面提升硬件使用效率**
  - 增大CPU的利用率
  - 增大内存的利用率
  - 增大磁盘IO的利用率
  - 增大网络带宽的利用率
- **提升硬件规格**
  - 网卡：万兆网卡，例如10G、25G、40G等
  - 磁盘：固态硬盘，关注IOPS和BPS指标
  - CPU：更快的主频，更多的核心，更大的缓存，更优的架构
  - 内存：更快的访问速度
- **超出硬件性能上限后使用DNS**



如何增大Nginx使用CPU的有效时长？

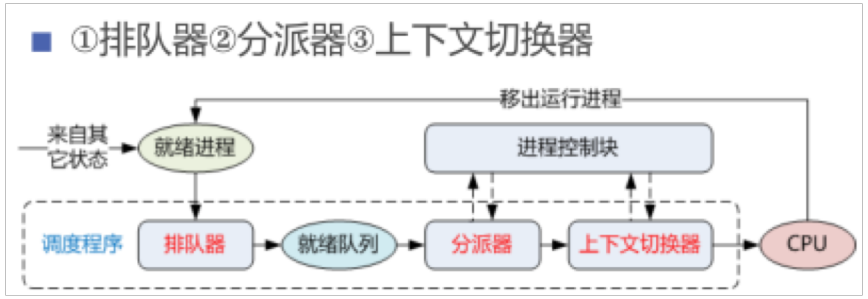
- **能够使用全部CPU资源**
  - master-worker多进程架构
  - worker进程数量应当大于等于CPU核数
- **Nginx进程间不做无用功浪费CPU资源**
  - worker进程不应在繁忙时，主动让出CPU
    - worker进程间不应由于争抢造成资源耗散
      - worker进程数量应当等于CPU核数
    - worker进程不应调用一些API导致主动让出CPU
      - 拒绝类似的第三方模块
- **不被其他进程争抢资源**
  - 提升优先级占用CPU更长的时间
  - 减少操作系统上耗资源的非Nginx进程

设置worker进程的数量

Syntax:	**worker_processes** number | auto;
Default:	worker_processes 1;
Context:	main

为何一个CPU就可以同时运行多个进程？

- **宏观上并行，微观上串行**
  - 把进程的运行时间分为一段段的时间片
  - OS调度系统依次选择每个进程，最多执行时间片指定的时长



- **阻塞API引发的时间片内主动让出CPU**
  - 速度不一致引发的阻塞API
    - 硬件执行速度不一致，例如CPU和磁盘
  - 业务场景产生的阻塞API
    - 例如同步读网络报文

确保进程在运行态！

- R运行：正在运行或在运行队列中等待。
- S中断：休眠中， 受阻， 在等待某个条件的形成或接受到信号。
- D不可中断：收到信号不唤醒和不可运行， 进程必须等待直到有中断发生。
- Z僵死：进程已终止， 但进程描述符存在， 直到父进程调用wait4()系统调用后释放。
- T停止：进程收到SIGSTOP， SIGSTP， SIGTIN， SIGTOU信号后停止运行。

减少进程间切换

- **Nginx worker尽可能的处于R状态**
  - R状态的进程数量大于cpu核心时，负载急速增高

```
)z ~]# uptime
  1 user,  load average: 0.00, 0.01, 0.05
```

```
top - 10:32:18 up 25 days, 23:43,  1 user,  load average: 0.00, 0.01, 0.05
Tasks: 131 total,   1 running, 130 sleeping,   0 stopped,   0 zombie
```

- **尽可能的减少进程间切换**
  - 何为进程间切换
    - 是指CPU 从一个进程或线程切换到另一个进程或线程
    - 类别
      - 主动切换
      - 被动切换：时间片耗尽
    - Cost: <5us
  - 减少主动切换
  - 减少被动切换
    - 增大进程优先级
- **绑定cpu**

延迟处理新连接

使用TCP_DEFER_ACCEPT 延迟处理新连接

Syntax:	**listen** address[:port] [deferred];
Default:	listen *:80 | *:8000;
Context:	server

如何查看上下文切换次数？

极客时间

- Vmstat

```
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 12773044 137824 1704168    0    0     0     4    0    0  1  1 98  0  0
 0  0      0 12772612 137824 1704172    0    0     0     0  974 1521  0  0 99  0  0
```

- Dstat

```
----total-cpu-usage---- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai hiq siq| read  writ| recv  send|  in   out | int   csw
  1   1  98   0   0   0|2158B   30k|   0     0 |   0     0 |2251  3411
```

- Pidstat -w
  - cswch/s：主动切换

```
11:05:56 AM   UID       PID   cswch/s nvcswch/s  Command
11:05:57 AM  1000      2874      2.00      0.00  nginx
```

什么决定CPU时间片的大小？

- Nice静态优先级：-20 -- 19
- Priority动态优先级：0-139

#define DEFAULT_PRIO (MAX_RT_PRIO + NICE_WIDTH /2)

```
top - 20:14:28 up 15 days,  9:26,  3 users,  load average: 0.00, 0.01, 0.0
Tasks: 122 total,   1 running, 121 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.0 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0
KiB Mem :  3881692 total,  1180072 free,   380488 used,  2321132 buff/cach
KiB Swap:        0 total,        0 free,        0 used.  3176148 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAN
  855 root      20   0  562388  18652   5888 S   0.3  0.5   2:10.33 tuned
 1081 root      20   0  129708  14244   9020 S   0.3  0.4  43:13.11 AliYun
15413 root      20   0  381096  24828   4144 S   0.3  0.6   1:19.69 docker
    1 root      20   0  190880   3844   2532 S   0.0  0.1   0:09.03 system
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kthrea
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.10 ksofti
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworke
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.10 migrat
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0   1:30.40 rcu_sc
   10 root      rt   0       0      0      0 S   0.0  0.0   0:05.89 watchd
   11 root      rt   0       0      0      0 S   0.0  0.0   0:05.05 watchd
```

O1调度算法（CFS）

- **优先级动态调整**
  - 幅度
    - +-5
  - 依据
    - CPU消耗型进程
    - IO消耗型进程
- **分时：时间片**
  - 5ms-800ms

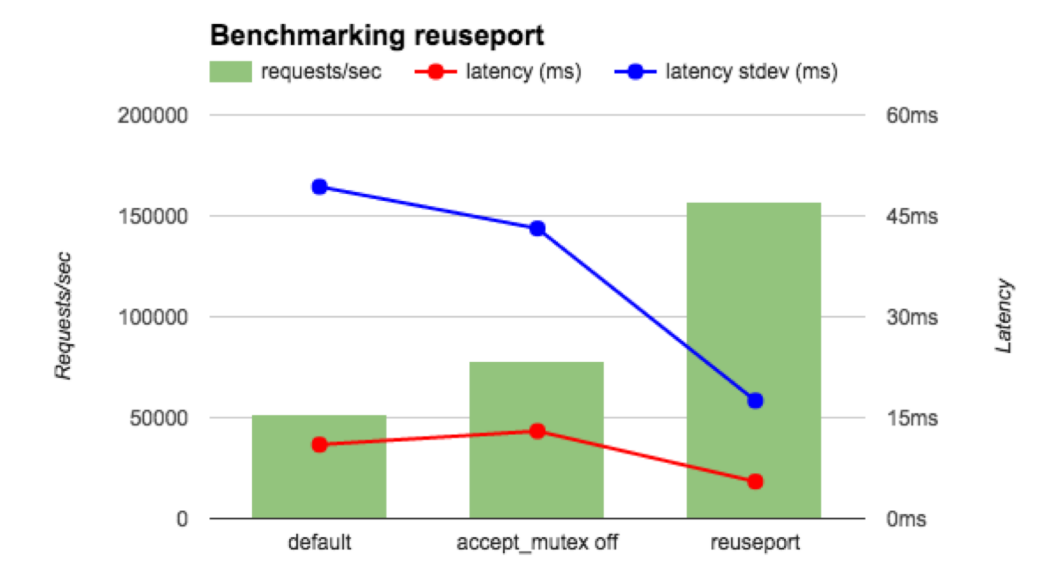
```
#define SCALE_PRIO(x, prio) \
    max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO/2), MIN_TIMESLICE)
static unsigned int task_timeslice(task_t *p)
{
    if (p->static_prio < NICE_TO_PRIO(0))
        return SCALE_PRIO(DEF_TIMESLICE*4, p->static_prio);
    else
        return SCALE_PRIO(DEF_TIMESLICE, p->static_prio);
}
```

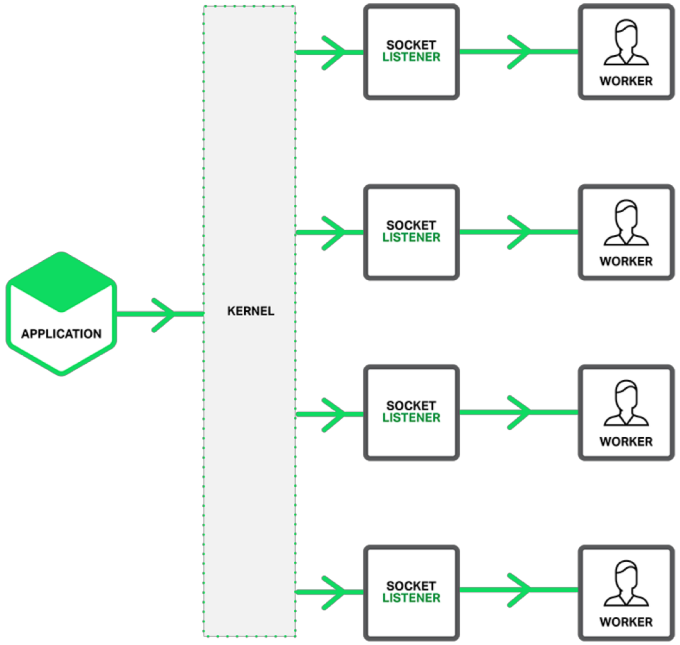
设置worker进程的静态优先级

极客时间

Syntax:	**worker_priority** number;
Default:	worker_priority 0;
Context:	main

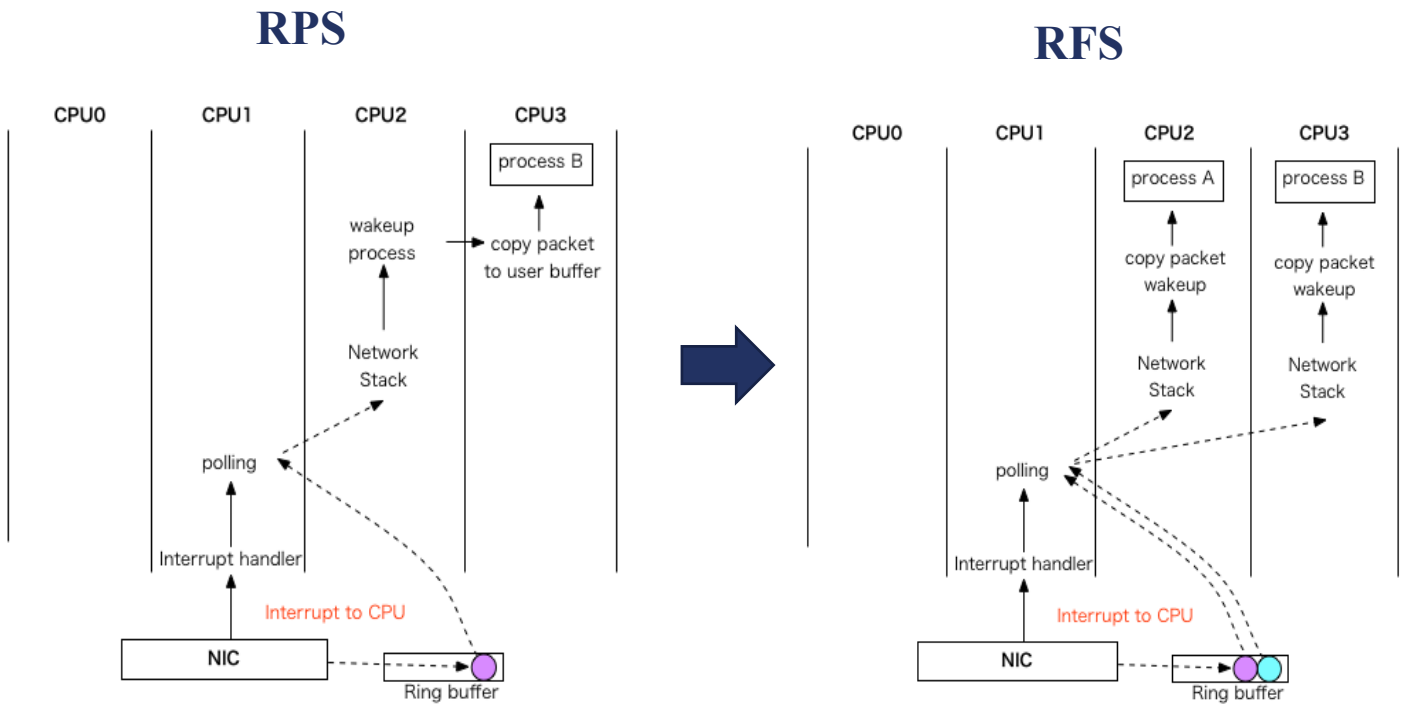
worker进程间负载均衡



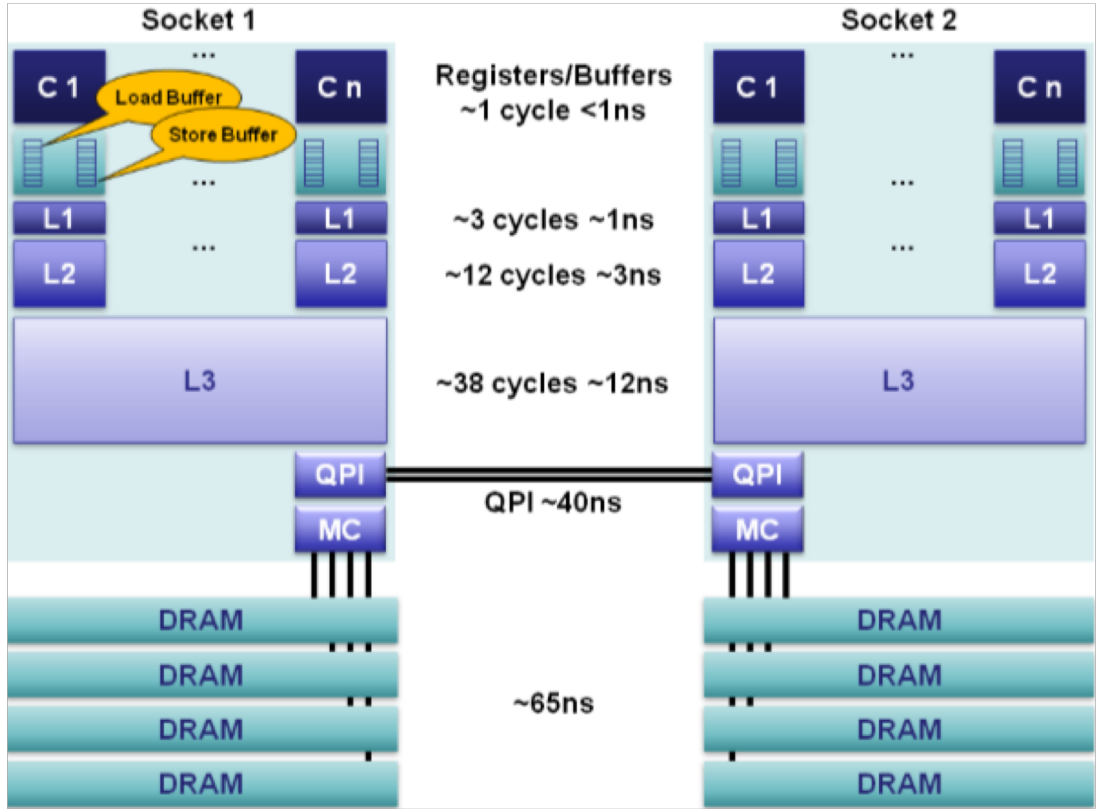


多队列网卡对多核CPU的优化

- RSS： Receive Side Scaling
  - 硬中断负载均衡
- RPS： Receive Packet Steering
  - 软中断负载均衡
- RFS： Receive Flow Steering



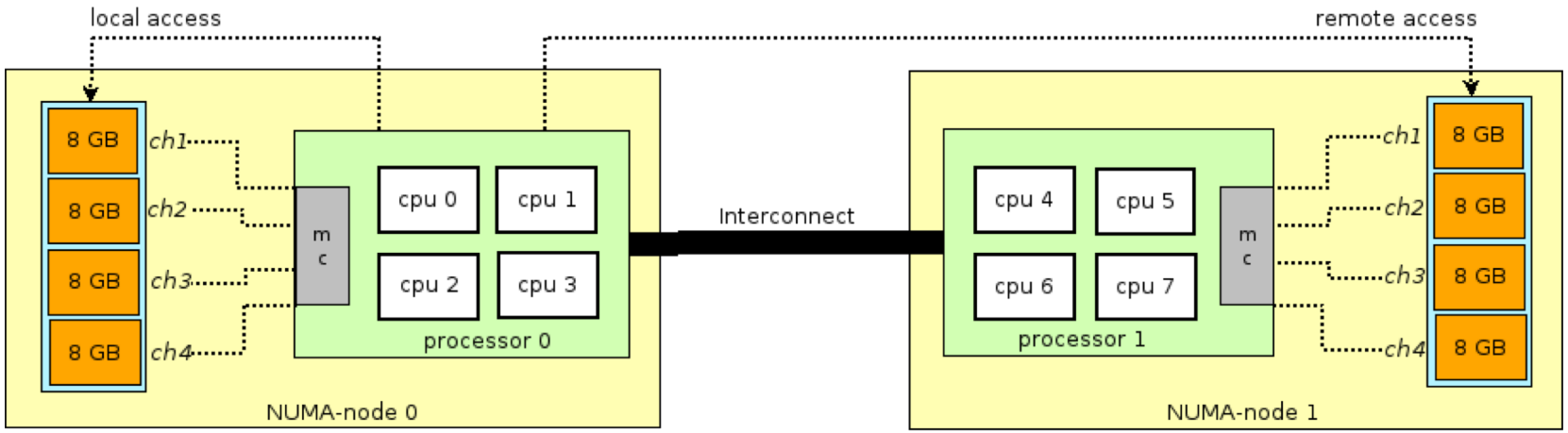
提升CPU缓存命中率：worker_cpu_affinity



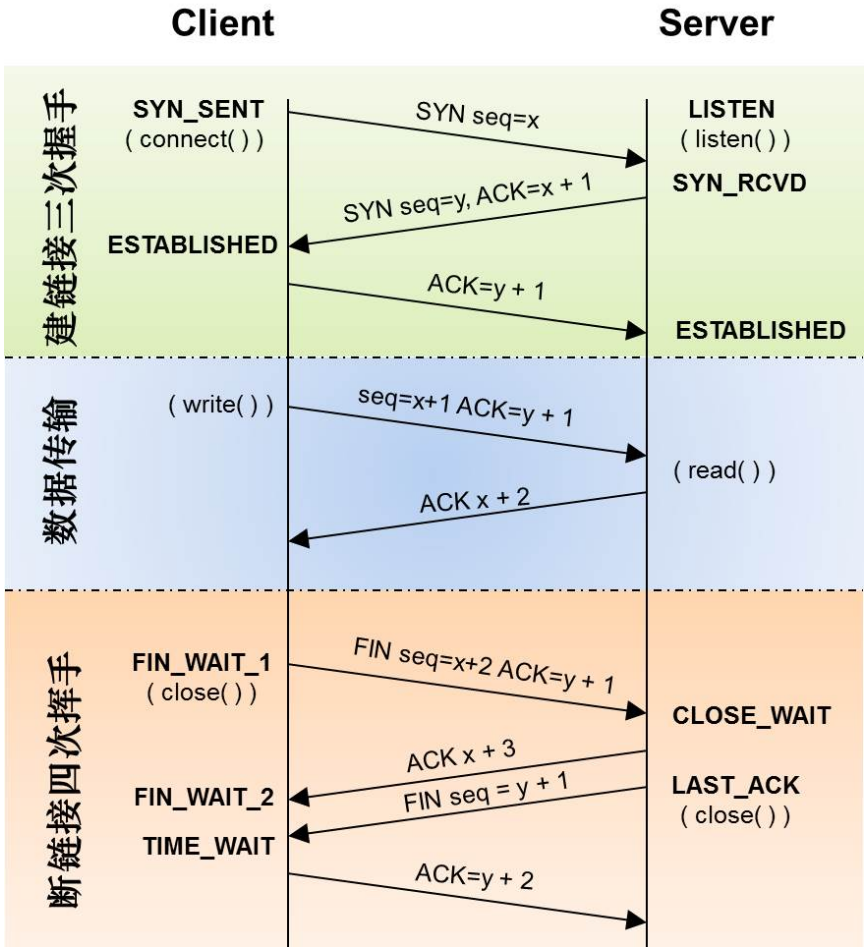
绑定worker到指定CPU

Syntax:	**worker_cpu_affinity** cpumask ...; **worker_cpu_affinity** auto [cpumask];
Default:	—
Context:	main

NUMA架构


local access
remote access
8 GB
8 GB
8 GB
8 GB
ch1
ch2
ch3
ch4
m
c
cpu 0
cpu 1
cpu 2
cpu 3
processor 0
NUMA-node 0
Interconnect
cpu 4
cpu 5
cpu 6
cpu 7
processor 1
NUMA-node 1

TCP连接


Client
Server
建链接三次握手
SYN_SENT
(connect( ))
SYN seq=x
LISTEN
(listen( ))
SYN_RCVD
SYN seq=y, ACK=x + 1
ESTABLISHED
ACK=y + 1
ESTABLISHED
数据传输
(write( ))
seq=x+1 ACK=y + 1
(read( ))
ACK x + 2
断链接四次挥手
FIN_WAIT_1
(close( ))
FIN seq=x+2 ACK=y + 1
CLOSE_WAIT
ACK x + 3
FIN_WAIT_2
FIN seq = y + 1
LAST_ACK
(close( ))
TIME_WAIT
ACK=y + 2

SYN_SENT状态

- **net.ipv4.tcp_syn_retries = 6**
  - 主动建立连接时，发SYN的重试次数

- **net.ipv4.ip_local_port_range = 32768　60999**
  - 建立连接时的本地端口可用范围

主动建立连接时应用层超时时间

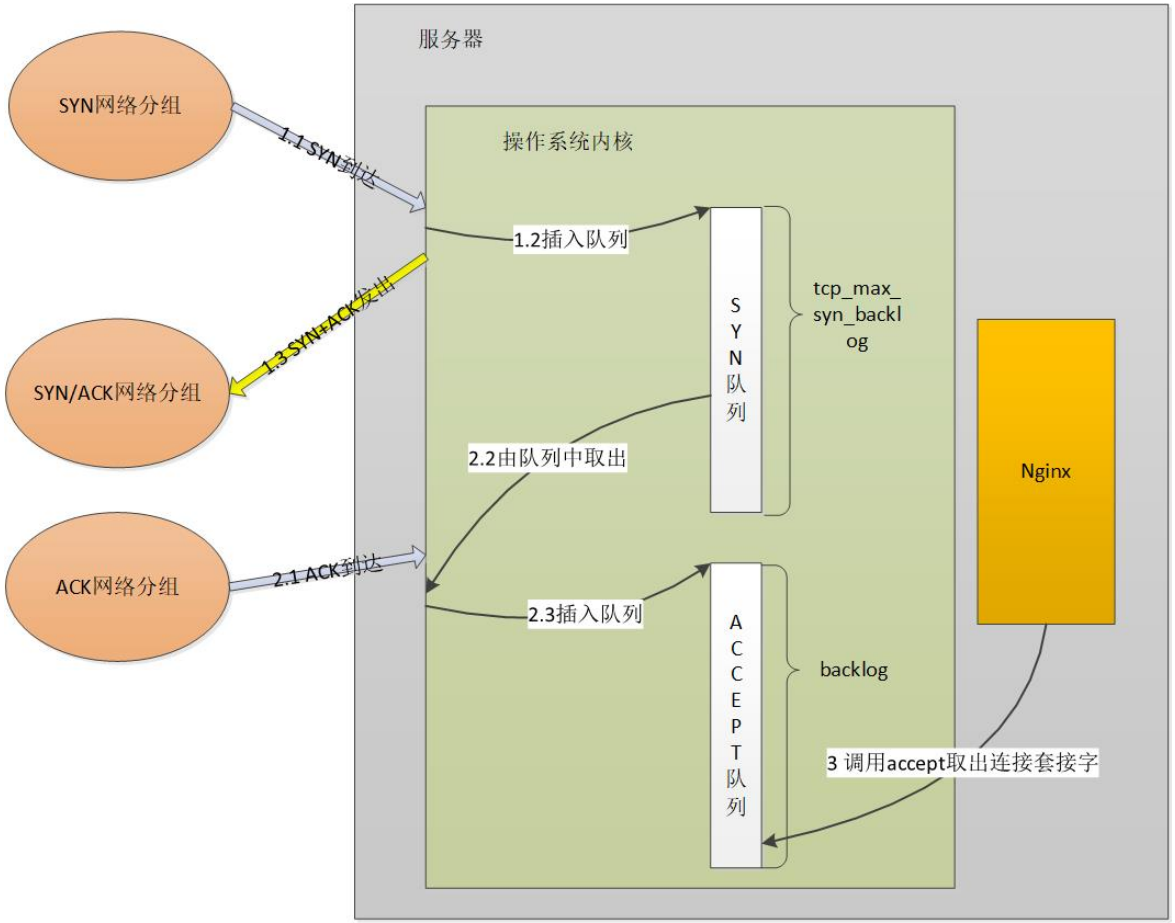
Syntax:	**proxy_connect_timeout** time;
Default:	proxy_connect_timeout 60s;
Context:	http, server, location

Syntax:	**proxy_connect_timeout** time;
Default:	proxy_connect_timeout 60s;
Context:	stream, server

SYN_RCVD状态

- **net.ipv4.tcp_max_syn_backlog**
  - SYN_RCVD状态连接的最大个数
- **net.ipv4.tcp_synack_retries**
  - 被动建立连接时，发SYN/ACK的重试次数

服务器端处理三次握手

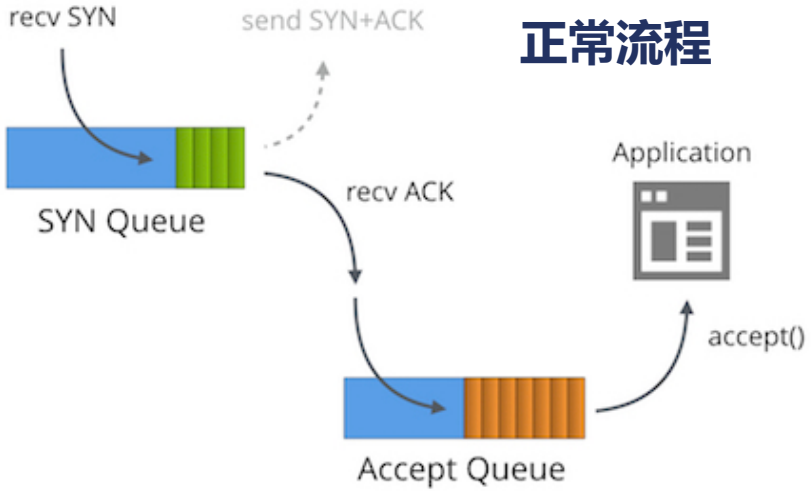

服务器
操作系统内核
SYN网络分组
SYN/ACK网络分组
ACK网络分组
1.1 SYN到达
1.2插入队列
1.3 SYN+ACK发出
2.1 ACK到达
2.2由队列中取出
2.3插入队列
SYN队列
tcp_max_syn_backlog
ACCEPT队列
backlog
Nginx
3 调用accept取出连接套接字

如何应对SYN攻击？

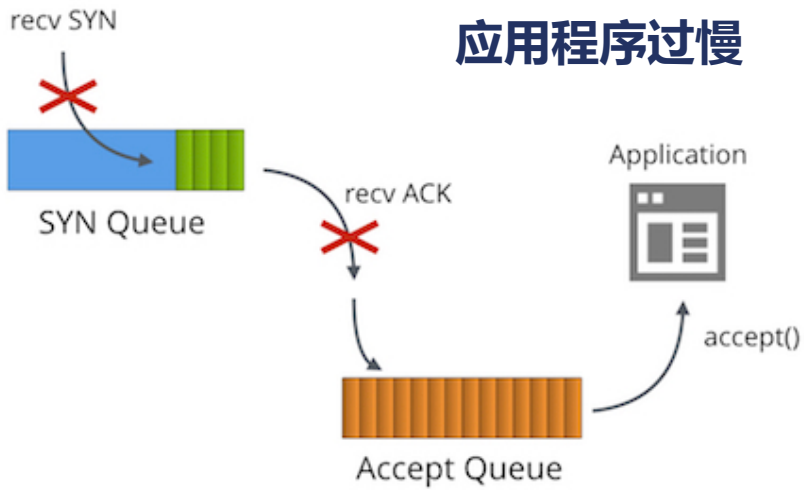
攻击者短时间伪造不同IP地址的SYN报文，快速占满backlog队列，使服务器不能为正常用户服务

- **net.core.netdev_max_backlog**
  - 接收自网卡、但未被内核协议栈处理的报文队列长度
- **net.ipv4.tcp_max_syn_backlog**
  - SYN_RCVD状态连接的最大个数
- **net.ipv4.tcp_abort_on_overflow**
  - 超出处理能力时，对新来的SYN直接回包RST，丢弃连接

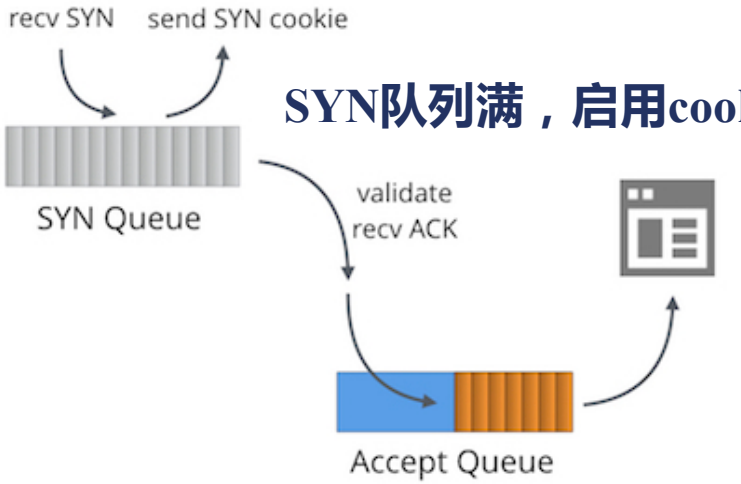
tcp_syncookies



正常流程



应用程序过慢



SYN队列满，启用cookie

- **net.ipv4.tcp_syncookies = 1**
  - 当SYN队列满后，新的SYN不进入队列，计算出cookie再以SYN+ACK中的序列号返回客户端，正常客户端发报文时，服务器根据报文中携带的cookie重新恢复连接
    - 由于cookie占用序列号空间，导致此时所有TCP可选功能失效，例如扩充窗口、时间戳等

一切皆文件：句柄数的上限

极客时间

- **操作系统全局**
  - fs.file-max
    - 操作系统可使用的最大句柄数
  - 使用fs.file-nr可以查看当前已分配、正使用、上限
    - fs.file-nr = 21632 0 40000500
- **限制用户**
  - /etc/security/limits.conf
    - root soft nofile 65535
    - root hard nofile 65535
- **限制进程**

Syntax: **worker_rlimit_nofile** number;
Default: —
Context: main

设置worker进程最大连接数量

极客时间

Syntax:	**worker_connections** number;
Default:	worker_connections 512;
Context:	events

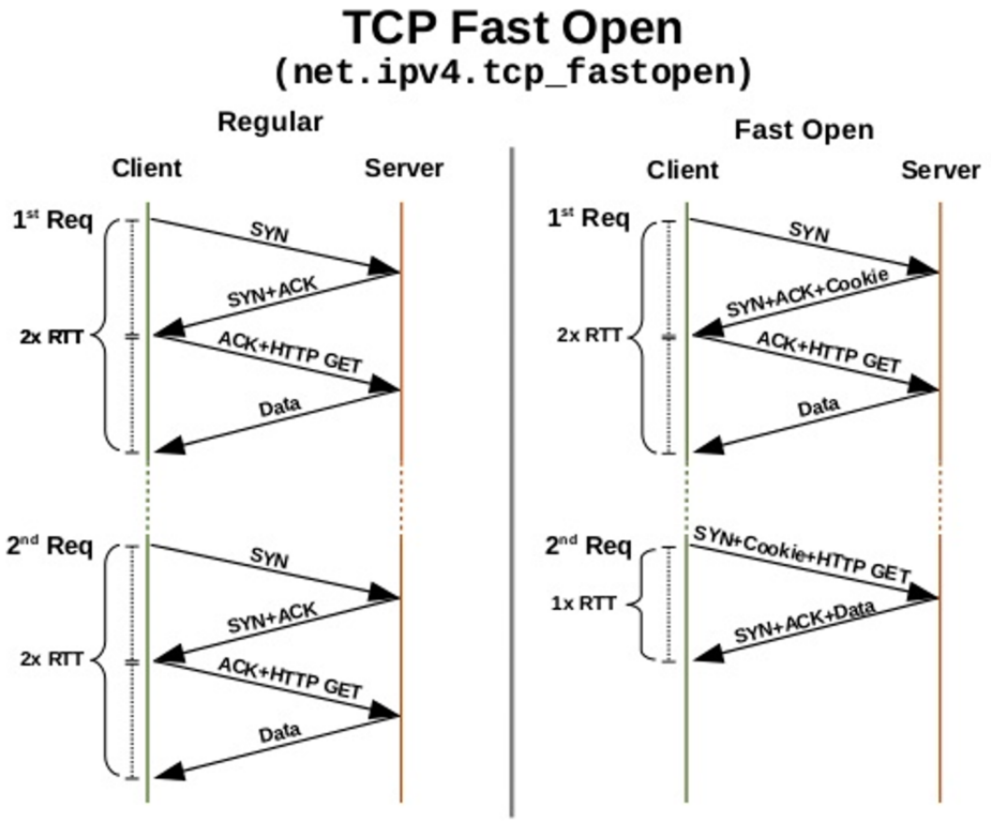
包括Nginx与上游、下游间的连接

两个队列的长度

- **SYN队列未完成握手**
  - net.ipv4.tcp_max_syn_backlog = 262144
- **ACCEPT队列已完成握手**
  - net.core.somaxconn
    - 系统级最大backlog队列长度

```
Syntax:     listen address[:port]  [backlog=number]
Default:    listen *:80 | *:8000;
Context:    server
```

Tcp Fast Open



TCP Fast Open

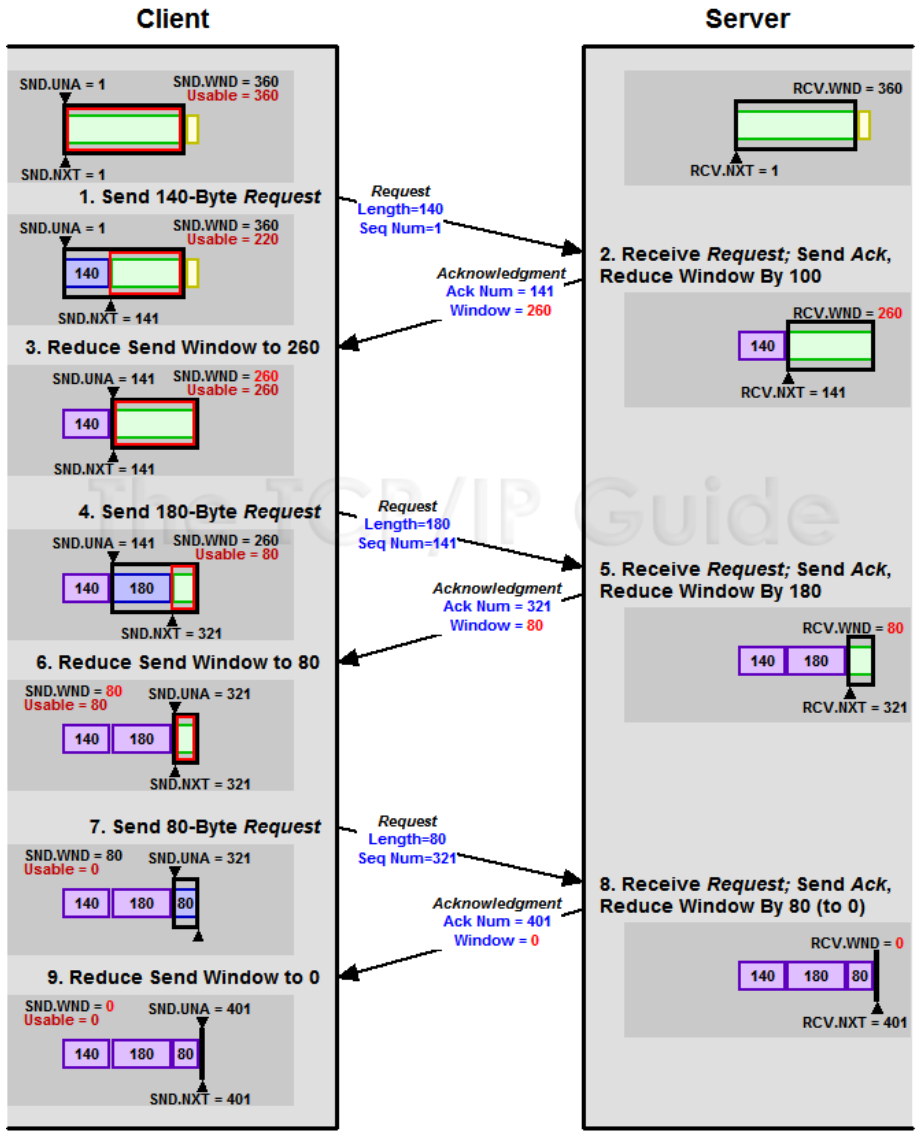
- **net.ipv4.tcp_fastopen：系统开启TFO功能**
  - 0：关闭
  - 1：作为客户端时可以使用TFO
  - 2：作为服务器时可以使用TFO
  - 3：无论作为客户端还是服务器，都可以使用TFO

Syntax:	listen address[:port] [fastopen=number];
Default:	listen *:80 | *:8000;
Context:	server

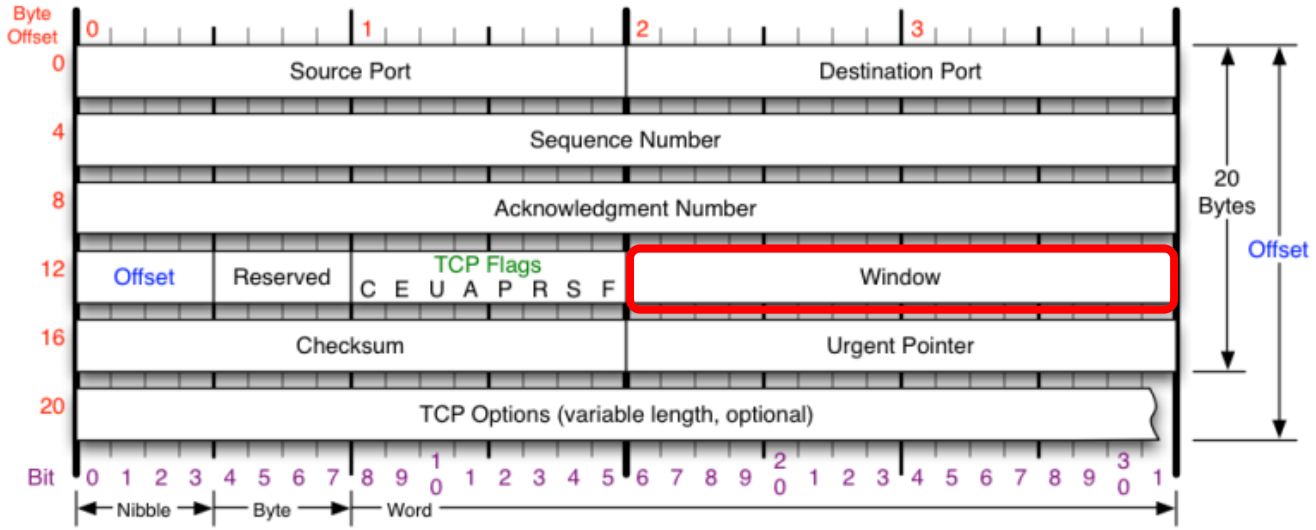
- **fastopen=number**
  - 为防止带数据的SYN攻击，限制最大长度，指定TFO连接队列的最大长度

滑动窗口

- **功能**
  - 用于限制连接的网速，解决报文乱序和可靠传输问题
  - Nginx中limit_rate等限速指令皆依赖它实现
  - 由操作系统内核实现
  - 连接两端各有发送窗口与接收窗口
- **发送窗口**
  - 用于发送内容
- **接收窗口**
  - 用于接收内容



通告窗口



TCP Flags

C E U A P R S F

Congestion Window
C 0x80 Reduced (CWR)
E 0x40 ECN Echo (ECE)
U 0x20 Urgent
A 0x10 Ack
P 0x08 Push
R 0x04 Reset
S 0x02 Syn
F 0x01 Fin

Congestion Notification

ECN (Explicit Congestion Notification). See RFC 3168 for full details, valid states below.

Packet State	DSB	ECN bits
Syn	0 0	1 1
Syn-Ack	0 0	0 1
Ack	0 1	0 0
No Congestion	0 1	0 0
No Congestion	1 0	0 0
Congestion	1 1	0 0
Receiver Response	1 1	0 1
Sender Response	1 1	1 1

TCP Options

0 End of Options List
1 No Operation (NOP, Pad)
2 Maximum segment size
3 Window Scale
4 Selective ACK ok
8 Timestamp

Checksum

Checksum of entire TCP segment and pseudo header (parts of IP header)

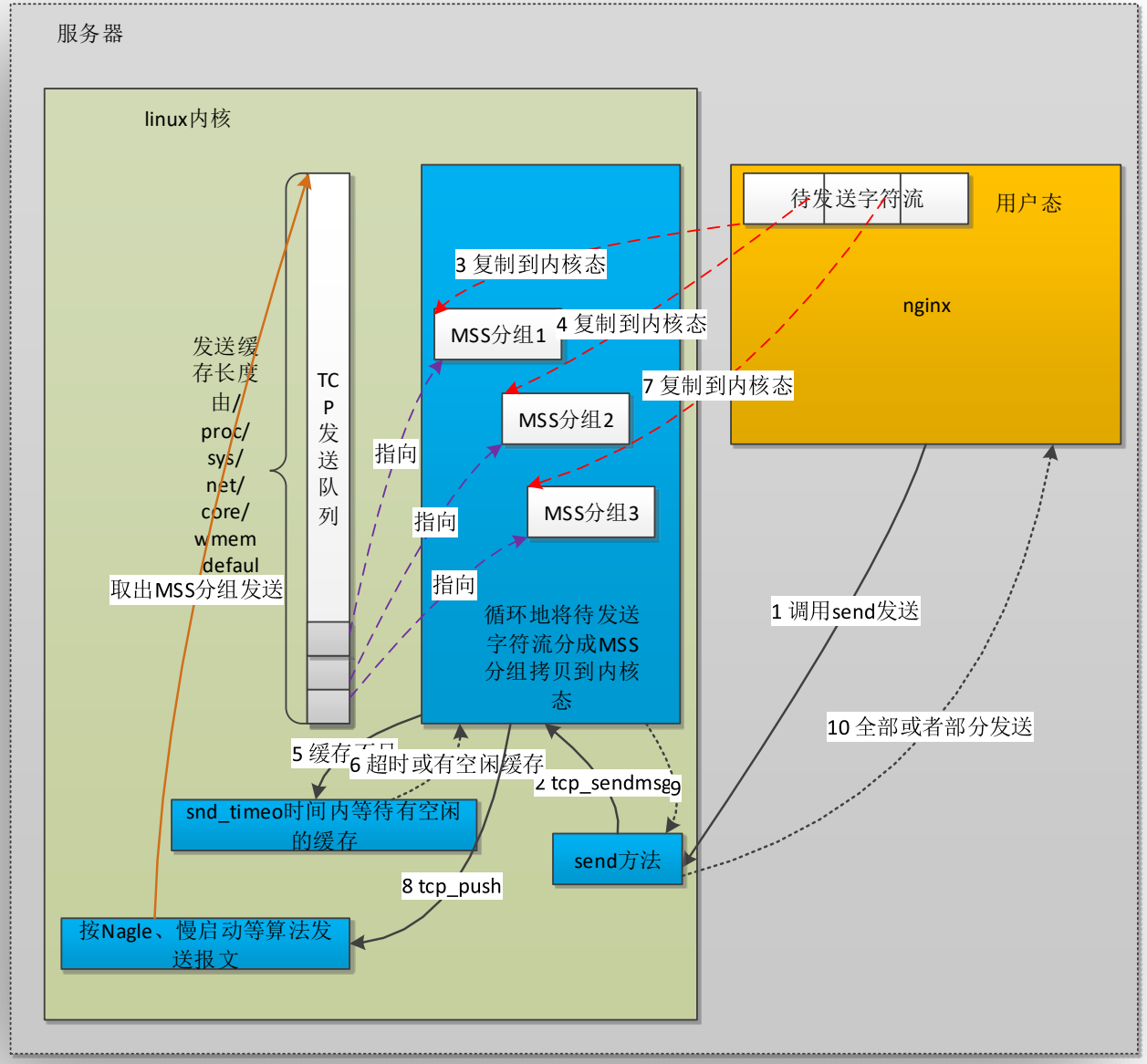
Offset

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

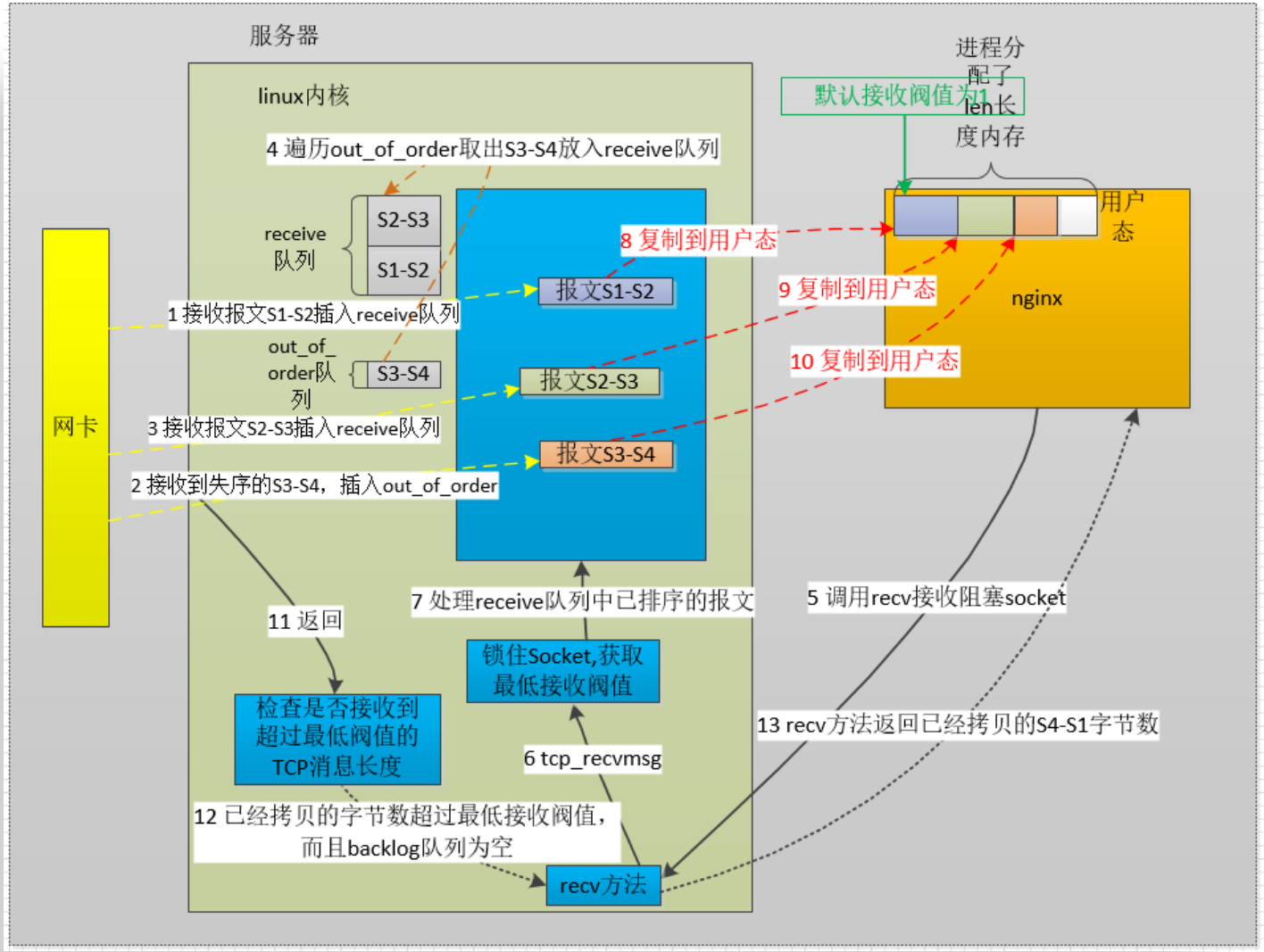
RFC 793

Please refer to RFC 793 for the complete Transmission Control Protocol (TCP) Specification.

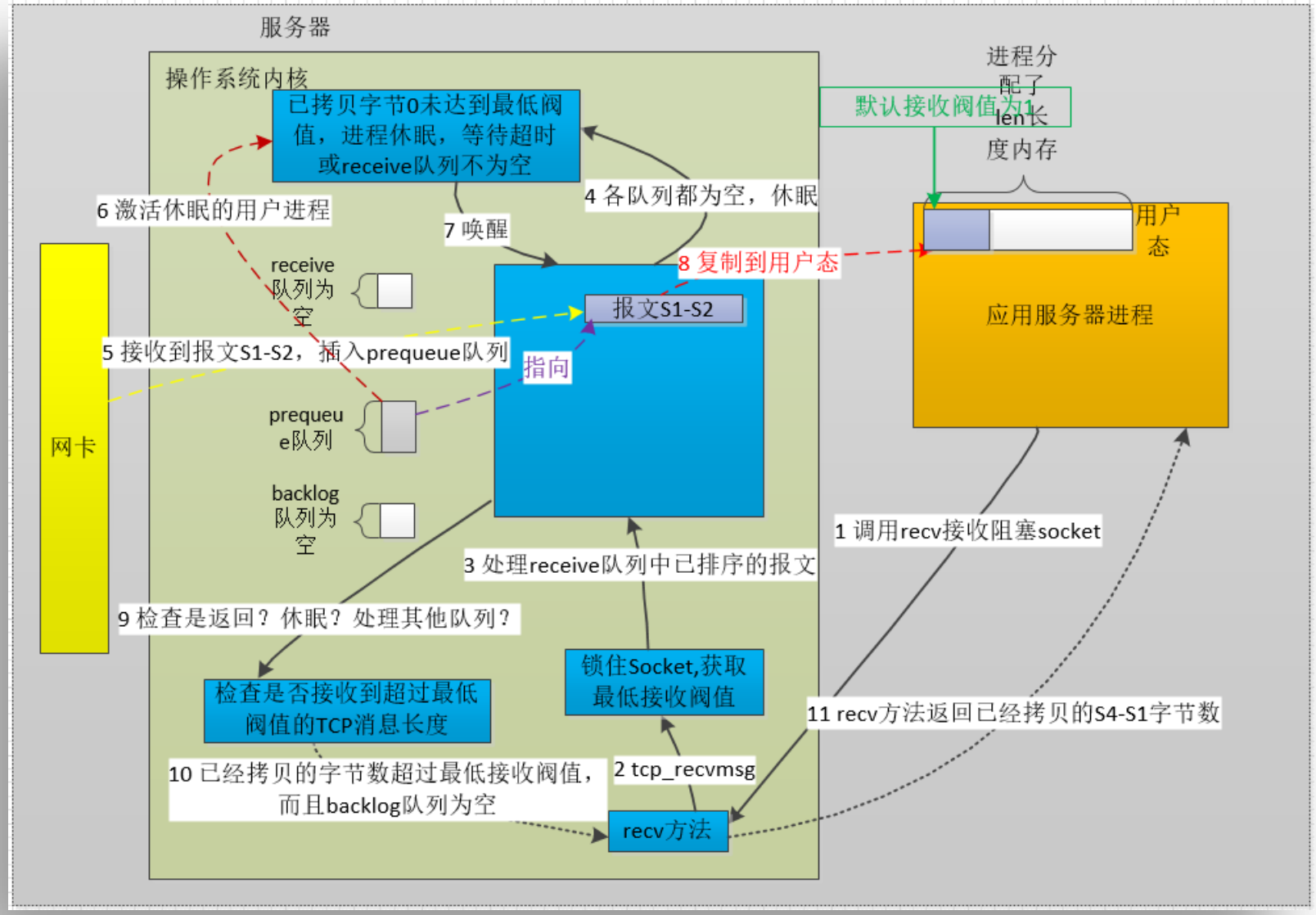
发送TCP消息


服务器
linux内核
待发送字符流
用户态
nginx
3 复制到内核态
4 复制到内核态
7 复制到内核态
MSS分组1
MSS分组2
MSS分组3
TCP发送队列
发送缓存长度由/proc/sys/net/core/wmem defaul
取出MSS分组发送
指向
指向
指向
循环地将待发送字符流分成MSS分组拷贝到内核态
1 调用send发送
10 全部或者部分发送
5 缓存不足
6 超时或有空闲缓存
9
snd_timeo时间内等待有空闲的缓存
send方法
8 tcp_push
按Nagle、慢启动等算法发送报文

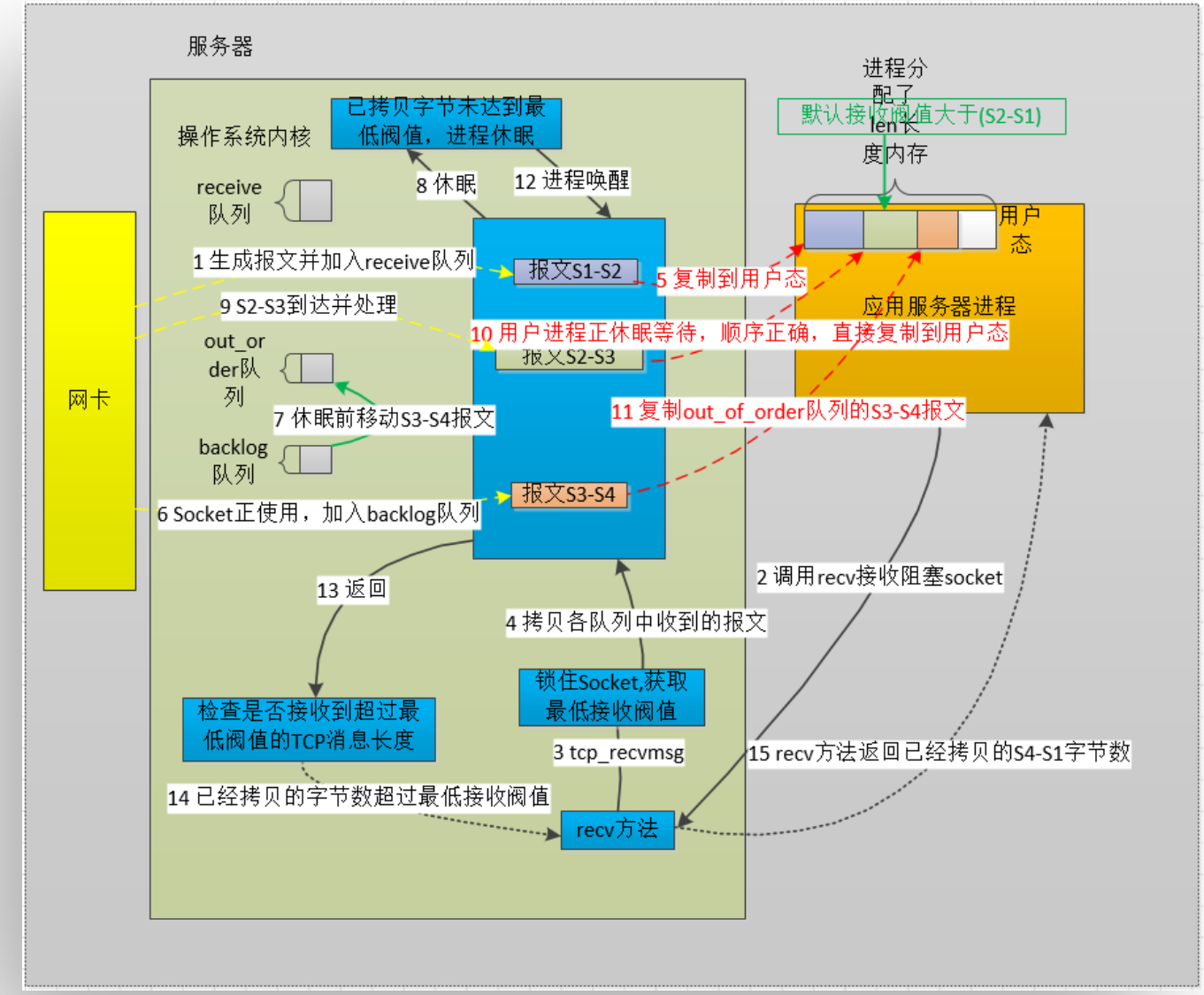
TCP消息接收


服务器
linux内核
进程分配了len长度内存
默认接收阀值为1
4 遍历out_of_order取出S3-S4放入receive队列
receive队列
S2-S3
S1-S2
out_of_order队列
S3-S4
网卡
1 接收报文S1-S2插入receive队列
3 接收报文S2-S3插入receive队列
2 接收到失序的S3-S4，插入out_of_order
报文S1-S2
报文S2-S3
报文S3-S4
8 复制到用户态
9 复制到用户态
10 复制到用户态
用户态
nginx
7 处理receive队列中已排序的报文
5 调用recv接收阻塞socket
11 返回
锁住Socket,获取最低接收阀值
检查是否接收到超过最低阀值的TCP消息长度
6 tcp_recvmsg
13 recv方法返回已经拷贝的S4-S1字节数
12 已经拷贝的字节数超过最低接收阀值，而且backlog队列为空
recv方法

TCP消息接收发生CS



TCP消息接收接收时新报文到达



nginx的超时指令与滑动窗口

极客时间

- **两次读操作间的超时**

Syntax:	**client_body_timeout** time;
Default:	client_body_timeout 60s;
Context:	http, server, location

- **两次写操作间的超时**

Syntax:	**send_timeout** time;
Default:	send_timeout 60s;
Context:	http, server, location

- **以上两者兼具**

Syntax:	**proxy_timeout** timeout;
Default:	proxy_timeout 10m;
Context:	stream, server

丢包重传

- **限制重传次数**
  - net.ipv4.tcp_retries1 = 3
    - 达到上限后，更新路由缓存
  - net.ipv4.tcp_retries2 = 15
    - 达到上限后，关闭TCP连接
  - 仅作近似理解，实际以超时时间为准，可能少于retries次数就认定达到上限

TCP缓冲区

- **net.ipv4.tcp_rmem = 4096　　87380　6291456**
  - 读缓存最小值、默认值、最大值，单位字节，覆盖net.core.rmem_max
- **net.ipv4.tcp_wmem = 4096　　16384　4194304**
  - 写缓存最小值、默认值、最大值，单位字节，覆盖net.core.wmem_max
- **net.ipv4.tcp_mem = 1541646　　2055528 3083292**
  - 系统无内存压力、启动压力模式阀值、最大值，单位为页的数量
- **net.ipv4.tcp_moderate_rcvbuf = 1**
  - 开启自动调整缓存模式

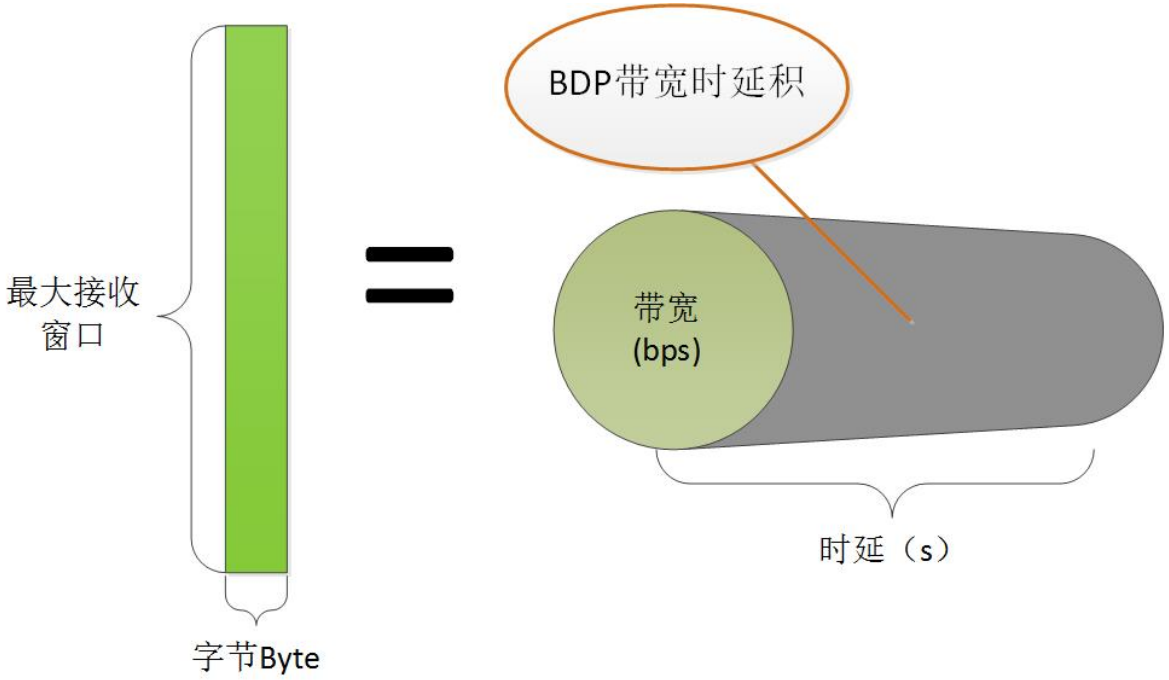
Syntax:	**listen** address[:port] [rcvbuf=size] [sndbuf=size];
Default:	listen *:80 | *:8000;
Context:	server

调整接收窗口与应用缓存

net.ipv4.tcp_adv_win_scale = 1

应用缓存 = buffer / (2^tcp_adv_win_scale)

BDP=带宽x时延，吞吐量=窗口/时延



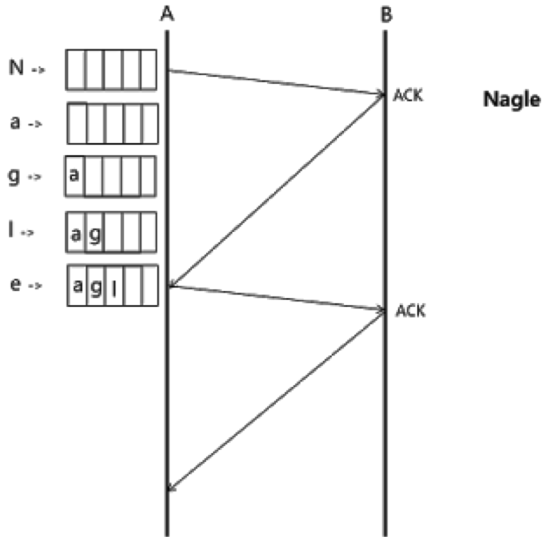
禁用Nagle算法？

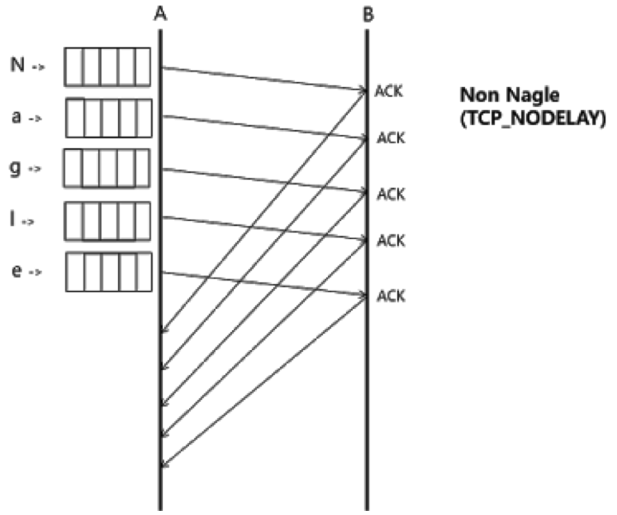
- **Nagle算法**
  - 避免一个连接上同时存在大量小报文
    - 最多只存在一个小报文
    - 合并多个小报文一起发送
  - 提高带宽利用率
- **吞吐量优先：启用Nagle算法，tcp_nodelay off**
- **低时延优先：禁用Nagle算法，tcp_nodelay on**

Syntax:	**tcp_nodelay** on | off;
Default:	tcp_nodelay on;
Context:	http, server, location

仅针对HTTP KeepAlive连接生效

Syntax:	**tcp_nodelay** on | off;
Default:	tcp_nodelay on;
Context:	stream, server





Nginx也可以避免发送小报文

Syntax:	**postpone_output** size;
Default:	postpone_output 1460;
Context:	http, server, location

启用CORK算法？

仅针对sendfile on开启时有效，完全禁止小报文的发送，提升网络效率

Syntax:	**tcp_nopush** on | off;
Default:	tcp_nopush off;
Context:	http, server, location

流量控制

- **拥塞窗口**
  - 发送方主动限制流量
- **通告窗口（对端接收窗口）**
  - 接收方限制流量
- **实际流量**
  - 拥塞窗口与通告窗口的最小值

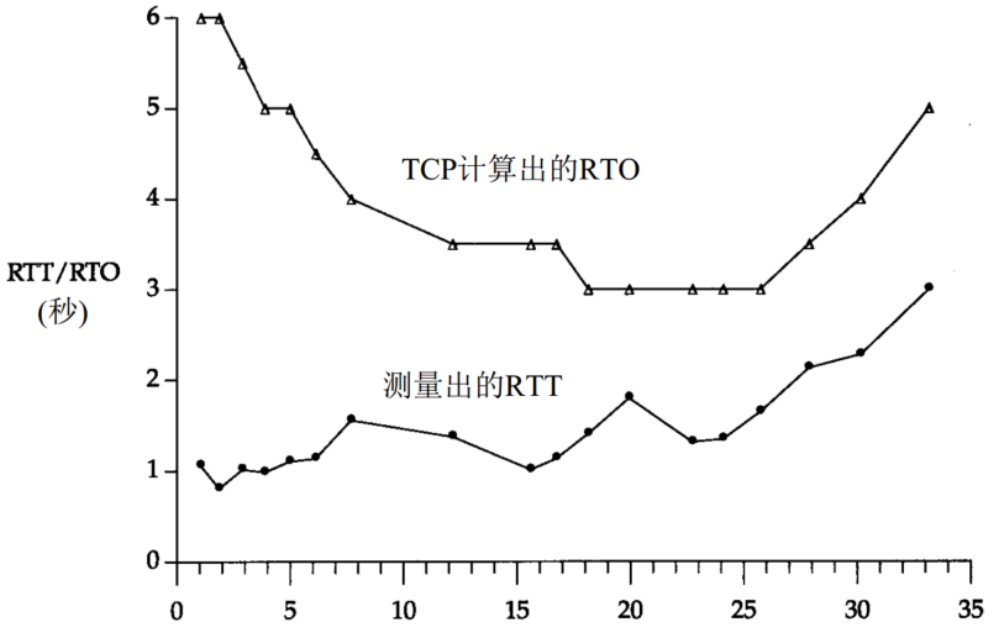
拥塞处理

- **慢启动**
  - 指数扩展拥塞窗口（cwnd为拥塞窗口大小）
    - 每收到1个ACK，cwnd = cwnd +1
    - 每过一个RTT，cwnd = cwnd * 2
- **拥塞避免：窗口大于threshold**
  - 线性扩展拥塞窗口
    - 每收到1个ACK，cwnd = cwnd + 1/cwnd
    - 每过一个RTT，窗口加1
- **拥塞发生**
  - 急速降低拥塞窗口
    - RTO超时，threshold = cwnd/2, cwnd =1
    - Fast Retransmit，收到3个duplicate ACK，cwnd = cwnd/2，threshold = cwnd
- **快速恢复**
  - 当Fast Retransmit出现时，cwnd调整为threshold + 3*MSS

慢启动

RTT与RTO

- **RTT：Round Trip Time**
  - 时刻变化
  - 组成
    - 物理链路传输时间
    - 末端处理时间
    - 路由器排队处理时间
  - 指导RTO
- **RTO： Retransmission TimeOut**
  - 正确的应对丢包



TCP的Keep-Alive功能

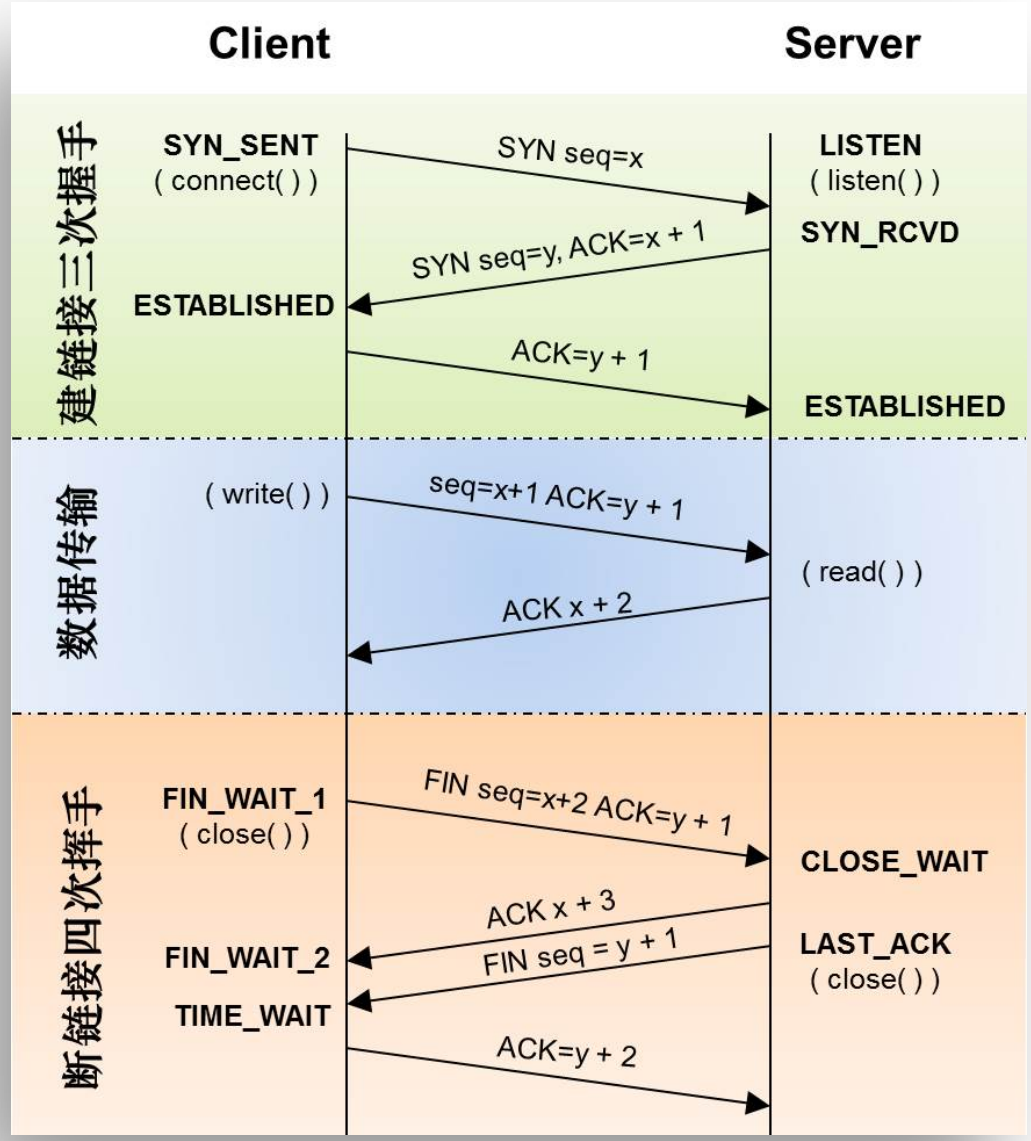
极客时间

- **应用场景**
  - 检测实际断掉的连接
  - 用于维持与客户端间的防火墙有活跃网络包

TCP的Keep-Alive功能

- **Nginx的Tcp keepalive**
  - so_keepalive=30m::10
  - keepidle, keepintvl, keepcnt
- **Linux的tcp keepalive**
  - 发送心跳周期
    - net.ipv4.tcp_keepalive_time = 7200
  - 探测包发送间隔
    - net.ipv4.tcp_keepalive_intvl = 75
  - 探测包重试次数
    - net.ipv4.tcp_keepalive_probes = 9

TCP连接


Client
Server
建链接三次握手
SYN_SENT
(connect( ))
SYN seq=x
LISTEN
(listen( ))
SYN seq=y, ACK=x + 1
SYN_RCVD
ESTABLISHED
ACK=y + 1
ESTABLISHED
数据传输
(write( ))
seq=x+1 ACK=y + 1
(read( ))
ACK x + 2
断链接四次挥手
FIN_WAIT_1
(close( ))
FIN seq=x+2 ACK=y + 1
CLOSE_WAIT
ACK x + 3
FIN_WAIT_2
FIN seq = y + 1
LAST_ACK
(close( ))
TIME_WAIT
ACK=y + 2

被动关闭连接端的状态

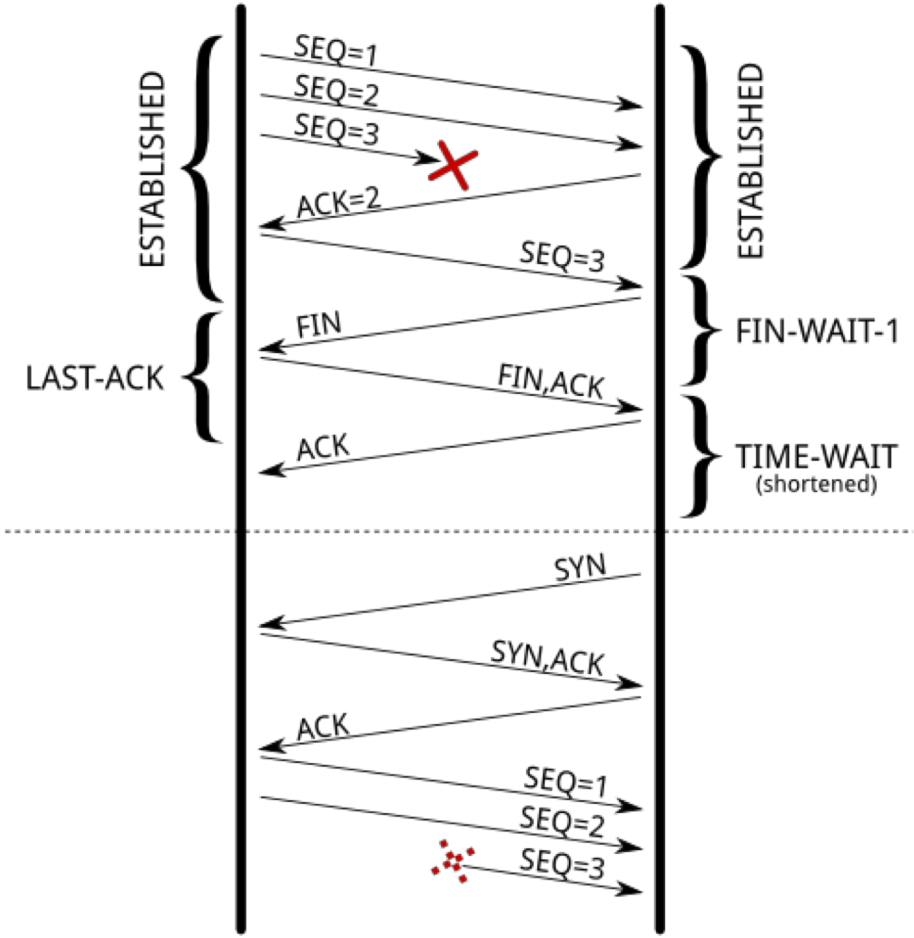
- **CLOSE_WAIT状态**
  - 应用进程没有及时响应对端关闭连接
- **LAST_ACK状态**
  - 等待接收主动关闭端操作系统发来的针对FIN的ACK报文

主动关闭连接端的状态

- **fin_wait1状态**
  - net.ipv4.tcp_orphan_retries = 0
    - 发送FIN报文的重试次数，0相当于8
- **fin_wait2状态**
  - net.ipv4.tcp_fin_timeout = 60
    - 保持在FIN_WAIT_2状态的时间
- **time_wait状态有什么作用？**

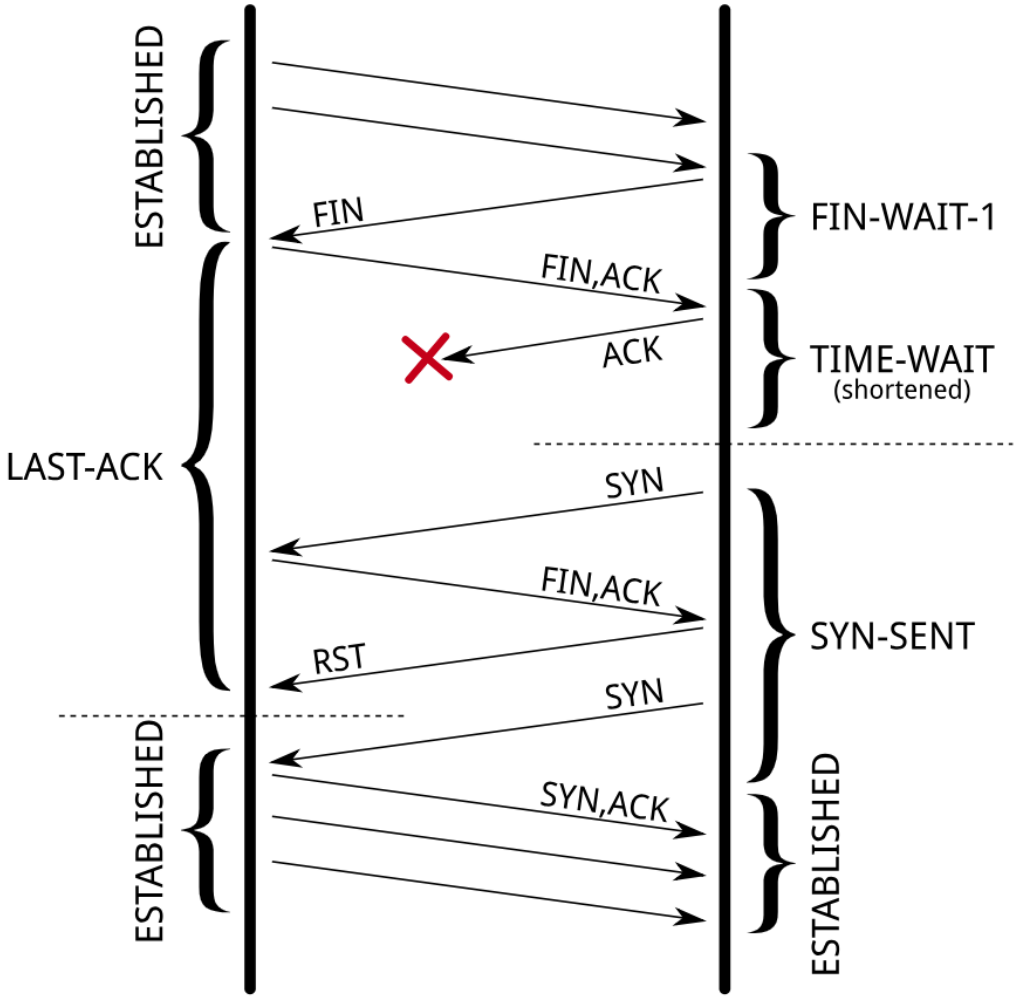
TIME-WAIT状态过短或者不存在会怎么样？

- **MSL(Maximum Segment Lifetime)**
  - 报文最大生存时间
- **维持2MSL时长的TIME-WAIT状态**
  - 保证至少一次报文的往返时间内端口是不可复用



TIME_WAIT优化：tcp_tw_reuse

- **net.ipv4.tcp_tw_reuse = 1**
  - 开启后，作为客户端时新连接可以使用仍然处于TIME-WAIT状态的端口
  - 由于timestamp的存在，操作系统可以拒绝迟到的报文
    - net.ipv4.tcp_timestamps = 1



TIME_WAIT优化

- **net.ipv4.tcp_tw_recycle = 0**
  - 开启后，同时作为客户端和服务器都可以使用TIME-WAIT状态的端口
  - 不安全，无法避免报文延迟、重复等给新连接造成混乱

- **net.ipv4.tcp_max_tw_buckets = 262144**
  - time_wait状态连接的最大数量
  - 超出后直接关闭连接

lingering_close延迟关闭的意义

当Nginx处理完成调用close关闭连接后，若接收缓冲区仍然收到客户端发来的内容，则服务器会向客户端发送RST包关闭连接，导致客户端由于收到RST而忽略了http response

lingering配置指令

Syntax:	**lingering_close** off | on | always;
Default:	lingering_close on;
Context:	http, server, location

- off：关闭功能
- on：由Nginx判断，当用户请求未接收完（根据chunk或者Content-Length头部等）时启用功能，否则及时关闭连接
- always：无条件启用功能

Syntax:	**lingering_time** time;
Default:	lingering_time 30s;
Context:	http, server, location

当功能启用时，最长的读取用户请求内容的时长，达到后立刻关闭连接

Syntax:	**lingering_timeout** time;
Default:	lingering_timeout 5s;
Context:	http, server, location

当功能启用时，检测客户端是否仍然请求内容到达，若超时后仍没有数据到达，则立刻关闭连接

以RST代替正常的四次握手关闭连接

当其他读、写超时指令生效引发连接关闭时，通过发送RST立刻释放端口、内存等资源来关闭连接

Syntax:	**reset_timedout_connection** on | off;
Default:	reset_timedout_connection off;
Context:	http, server, location

TLS/SSL优化握手性能

极客时间

Syntax:	**ssl_session_cache** off | none | [builtin[:size]] [shared:name:size];
Default:	ssl_session_cache none;
Context:	http, server

- **off**
  - 不使用Session缓存，且Nginx在协议中明确告诉客户端Session缓存不被使用
- **none**
  - 不使用Session缓存
- **builtin**
  - 使用Openssl的Session缓存，由于在内存中使用，所以仅当同一客户端的两次连接都命中到同一个worker进程时，Session缓存才会生效
- **shared:name:size**
  - 定义共享内存，为所有worker进程提供Session缓存服务。1MB大约可用于4000个Session

TLS/SSL中的会话票证tickets

Nginx将会话Session中的信息作为tickets加密发给客户端，当客户端下次发起TLS连接时带上tickets，由Nginx解密验证后复用会话Session。
会话票证虽然更易在Nginx集群中使用，但破坏了TLS/SSL的安全机制，有安全风险，必须频繁更换tickets密钥。
是否开启会话票证服务：

Syntax:	**ssl_session_tickets** on | off;
Default:	ssl_session_tickets on;
Context:	http, server

使用会话票证时加密tickets的密钥文件：

Syntax:	**ssl_session_ticket_key** file;
Default:	—
Context:	http, server

HTTP长连接

优点：

- 减少握手次数
- 通过减少并发连接数减少了服务器资源的消耗
- 降低TCP拥塞控制的影响

Syntax:	**keepalive_requests** number;
Default:	keepalive_requests 100;
Context:	http, server, location

Syntax:	**keepalive_requests** number;
Default:	keepalive_requests 100;
Context:	upstream

gzip压缩

功能：

通过实时压缩http包体，提升网络传输效率

模块：

ngx_http_gzip_module，通过--without-http_gzip_module禁用模块

Syntax:	**gzip** on | off;
Default:	gzip off;
Context:	http, server, location, if in location

压缩哪些请求的响应？

极客时间

Syntax:	**gzip_types** mime-type ...;
Default:	gzip_types text/html;
Context:	http, server, location

Syntax:	**gzip_min_length** length;
Default:	gzip_min_length 20;
Context:	http, server, location

Syntax:	**gzip_disable** regex ...;
Default:	—
Context:	http, server, location

Syntax:	**gzip_http_version** 1.0 | 1.1;
Default:	gzip_http_version 1.1;
Context:	http, server, location

是否压缩上游的响应

Syntax:	**gzip_proxied** off | expired | no-cache | no-store | private | no_last_modified | no_etag | auth | any ...;
Default:	gzip_proxied off;
Context:	http, server, location

- **off**
  - 不压缩来自上游的响应
- **expired**
  - 如果上游响应中含有Expires头部，且其值中的时间与系统时间比较后确定不会缓存，则压缩响应
- **no-cache**
  - 如果上游响应中含有“Cache-Control”头部，且其值含有”no-cache”值，则压缩响应
- **no-store**
  - 如果上游响应中含有“Cache-Control”头部，且其值含有”no-store”值，则压缩响应
- **private**
  - 如果上游响应中含有“Cache-Control”头部，且其值含有”private”值，则压缩响应
- **no_last_modified**
  - 如果上游响应中没有 “Last-Modified” 头部，则压缩响应
- **no_etag**
  - 如果上游响应中没有 “ETag” 头部，则压缩响应
- **auth**
  - 如果客户端请求中含有 “Authorization” 头部，则压缩响应
- **any**
  - 压缩所有来自上游的响应

其他压缩参数

Syntax:	**gzip_comp_level** level;
Default:	gzip_comp_level 1;
Context:	http, server, location

Syntax:	gzip_buffers number size;
Default:	gzip_buffers 32 4k|16 8k;
Context:	http, server, location

Syntax:	gzip_vary on | off;
Default:	gzip_vary off;
Context:	http, server, location

升级更高效的http2协议

向前兼容http/1.x协议

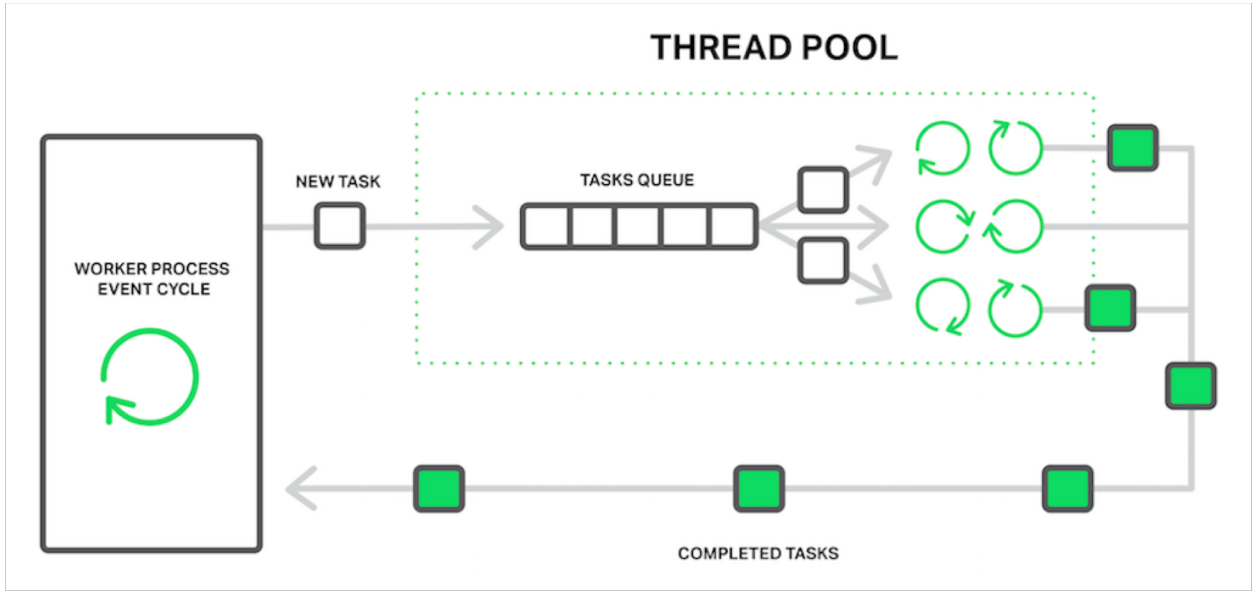
传输效率大幅度提升

磁盘IO的优化

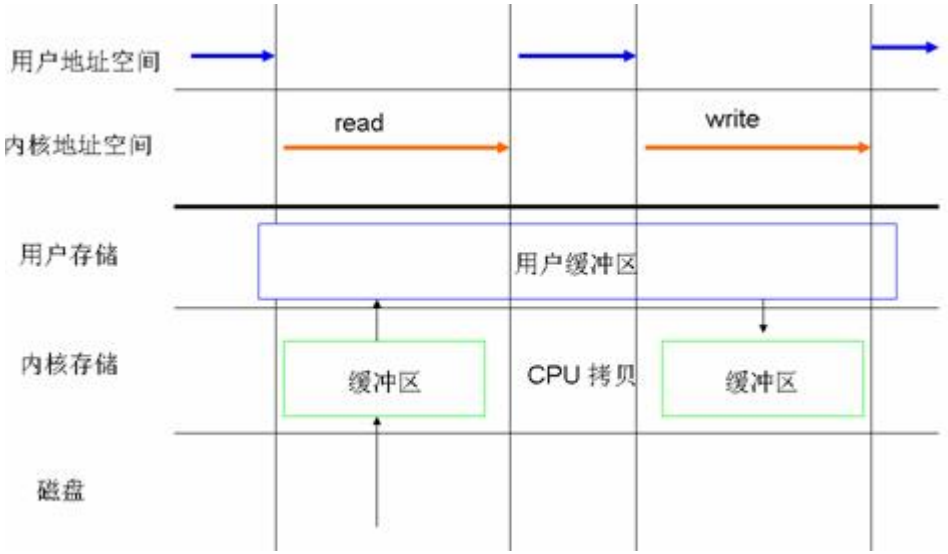
- 磁盘介质
  - 机械硬盘
    - 价格低
    - 存储量大
    - BPS较大：适用于顺序读写
    - IOPS较小
    - 寿命长
  - 固态硬盘
    - 价格高
    - 存储量小
    - BPS大
    - IOPS大：适用于随机读写
    - 写寿命短

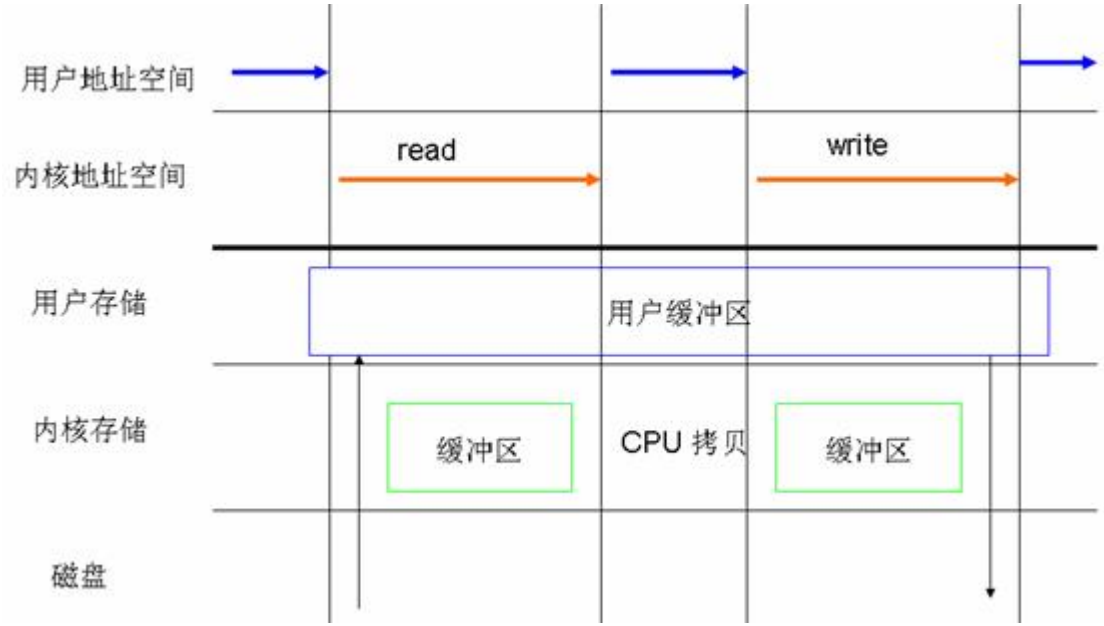
减少磁盘IO

- **优化读取**
  - Sendfile零拷贝
  - 内存盘、SSD盘
- **减少写入**
  - AIO
  - 增大error_log级别
  - 关闭access_log
  - 压缩access_log
  - 是否启用proxy buffering?
  - syslog替代本地IO
- **线程池thread pool**



直接IO绕开磁盘高速缓存





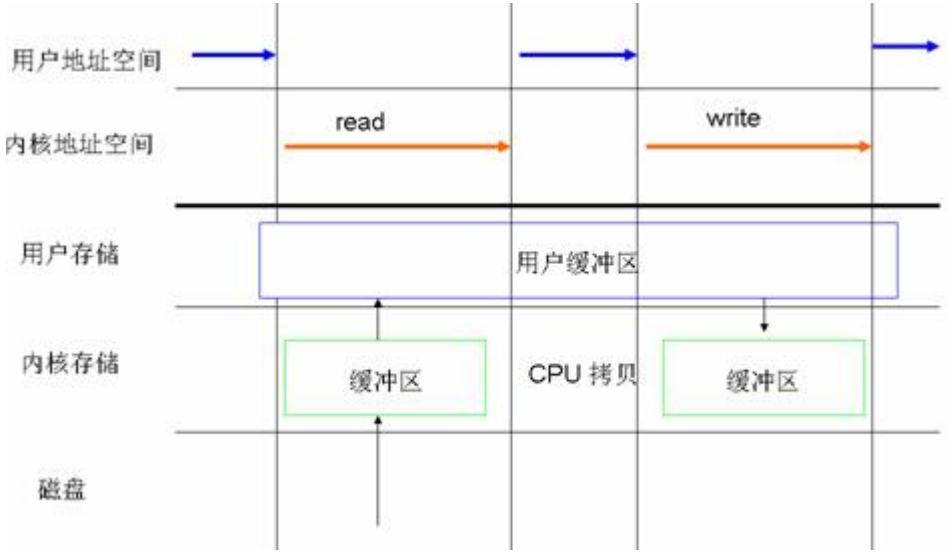
适用于大文件：直接IO

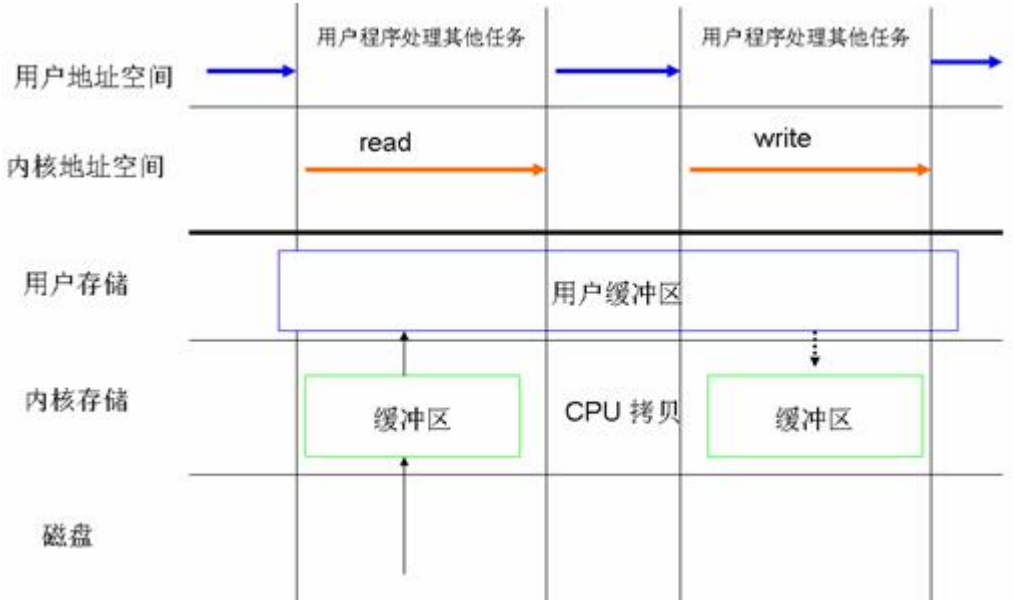
当磁盘上的文件大小超过size后，启用directIO功能，避免Buffered IO模式下磁盘页缓存中的拷贝消耗

Syntax:	**directio** size | off;
Default:	directio off;
Context:	http, server, location

Syntax:	**directio_alignment** size;
Default:	directio_alignment 512;
Context:	http, server, location

异步IO


用户地址空间
内核地址空间
read
write
用户存储
用户缓冲区
内核存储
缓冲区
CPU 拷贝
缓冲区
磁盘


用户程序处理其他任务
用户程序处理其他任务
用户地址空间
内核地址空间
read
write
用户存储
用户缓冲区
内核存储
缓冲区
CPU 拷贝
缓冲区
磁盘

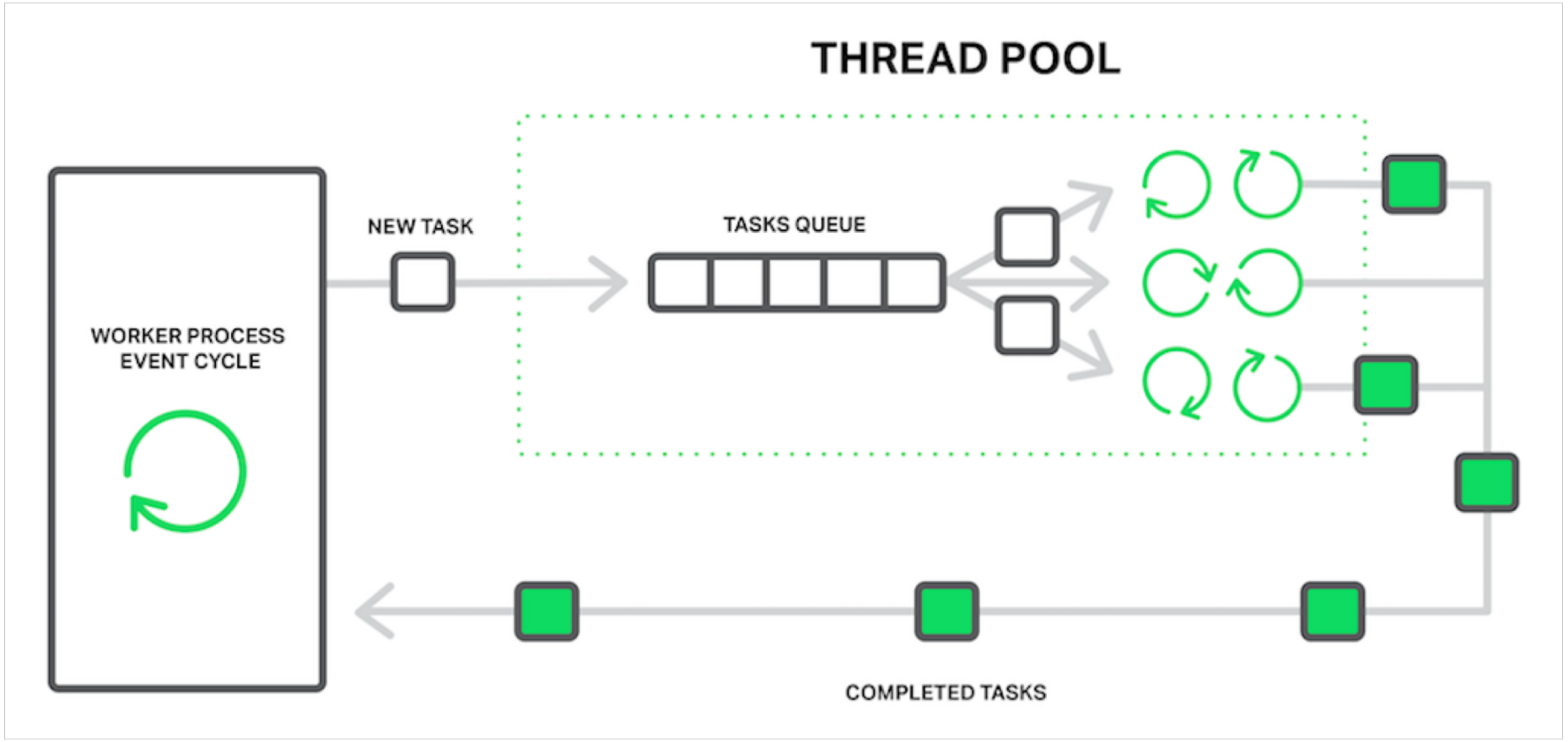
异步IO

Syntax:	**aio** on | off | threads[=pool];
Default:	aio off;
Context:	http, server, location

Syntax:	**aio_write** on | off;
Default:	aio_write off;
Context:	http, server, location

异步读IO线程池

编译时加--with-threads



定义线程池

极客时间

Syntax:	**thread_pool** name threads=number [max_queue=number];
Default:	thread_pool default threads=32 max_queue=65536;
Context:	main

异步IO中的缓存

将磁盘文件读入缓存中待处理，例如gzip模块会使用

Syntax:	**output_buffers** number size;
Default:	output_buffers 2 32k;
Context:	http, server, location

empty_gif 模块

- **模块：**
  - ngx_http_empty_gif_module模块，通过--without-http_empty_gif_module禁用模块
- **功能：**
  - 从前端页面做用户行为分析时，由于跨域等要求，前端打点的上报数据一般是GET请求，且考虑到浏览器解析DOM树的性能消耗，所以请求透明图片消耗最小，而1*1的gif图片体积最小（仅43字节），故通常请求gif图片，并在请求中把用户行为信息上报服务器。
  - Nginx可以在access日志中获取到请求参数，进而统计用户行为。但若在磁盘中读取1x1的文件则有磁盘IO消耗，empty_gif模块将图片放在内存中，加快了处理速度。

Syntax:	**empty_gif;**
Default:	—
Context:	location

access日志的压缩

Syntax:	**access_log** path [format **[buffer=size] [gzip[=level]] [flush=time]** [if=condition]];
Default:	access_log logs/access.log combined;
Context:	http, server, location, if in location, limit_except

buffer默认64KB

gzip默认级别为1

通过zcat解压查看

error.log日志输出内存

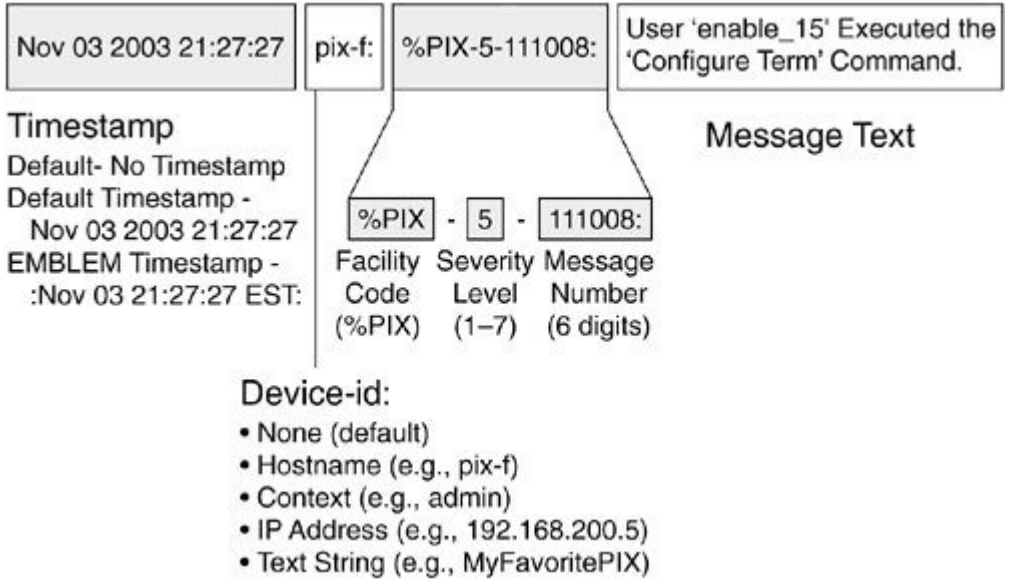
- **场景：**
  - 在开发环境下定位问题时，若需要打开debug级别日志，但对debug级别大量日志引发的性能问题不能容忍，可以将日志输出到内存中
- **配置语法：**
  - error_log memory:32m debug;
- **查看内存中日志的方法：**
  - gdb -p [worker进程id] -ex "source nginx.gdb" --batch
  - nginx.gdb脚本内容

```
set $log = ngx_cycle->log
while $log->writer != ngx_log_memory_writer
    set $log = $log->next
end
set $buf = (ngx_log_memory_buf_t *) $log->wdata
dump binary memory debug_log.txt $buf->start $buf->end
```

syslog协议

网络协议：

日志格式：



Nginx syslog

- **server**
  - 定义syslog服务器地址
- **facility**
  - 参见RFC3164，取值：“kern”，“user”，“mail”，“daemon”，“auth”，“intern”，“lpr”，“news”，“uucp”，“clock”，“authpriv”，“ftp”，“ntp”，“audit”，“alert”，“cron”，“local0”..“local7”
  - 默认localI7
- **severity**
  - 定义access.log日志的级别，默认info级别
- **tag**
  - 定义日志的tag，默认为”nginx”
- **nohostname**
  - 不向syslog中写入主机名hostname

rsyslog与nginx

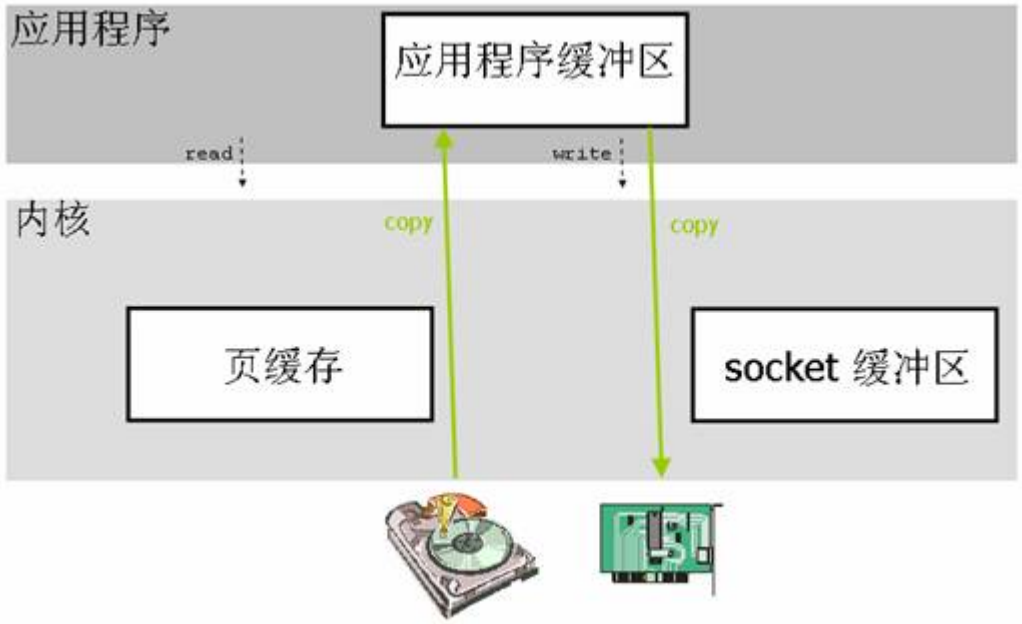
rsyslog配置：

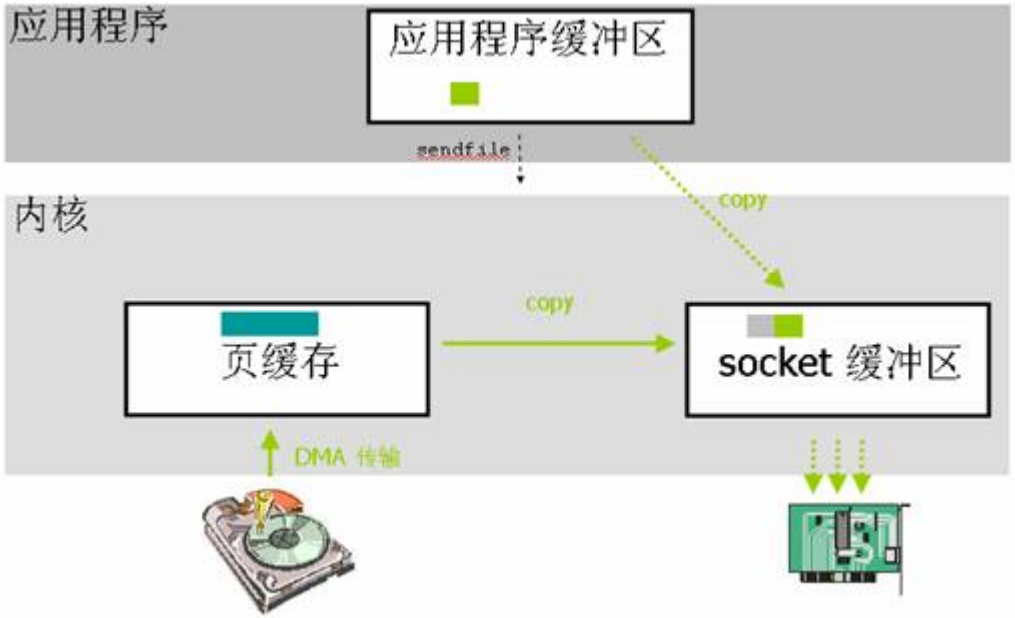
```
$ModLoad imudp
$UDPServerRun 514
local6.*                        /var/log/nginx.log
```

sendfile零拷贝提升性能

减少进程间切换

减少内存拷贝次数





sendfile、直接IO、异步IO同时开启？

```
location /video/ {
    sendfile on;
    aio on;
    directio 8m;
}
```

当文件大小超过8M时，启用AIO与directio

gzip_static模块

模块：

ngx_http_gzip_static_module，通过--with-http_gzip_static_module启用模块

功能：

检测到同名.gz文件时，response中以gzip相关header返回.gz文件的内容

Syntax:	**gzip_static** on | off | always;
Default:	gzip_static off;
Context:	http, server, location

gunzip模块

模块：

ngx_http_gunzip_module，通过--with-http_gunzip_module启用模块

功能：

当客户端不支持gzip时，且磁盘上仅有压缩文件，则实时解压缩并将其发送给客户端

Syntax:	**gunzip** on | off;
Default:	gunzip off;
Context:	http, server, location

Syntax:	**gunzip_buffers** number size;
Default:	gunzip_buffers 32 4k|16 8k;
Context:	http, server, location

tcmalloc

- **更快的内存分配器**
  - 并发能力强于glibc
    - 并发线程数越多，性能越好
  - 减少内存碎片
  - 擅长管理小块内存

http://goog-perftools.sourceforge.net/doc/tcmalloc.html

使用方式

- **编译google perf tools，得到tcmalloc库**
  - https://github.com/gperftools/gperftools/releases

- **编译nginx时，加入编译选项**
  - C语言生成可执行文件的两个步骤
    - 通过configure设定编译的额外参数
      - --with-cc-opt=OPTIONS　　set additional C compiler options
    - 通过configure设定链接的额外参数
      - --with-ld-opt=OPTIONS　　set additional linker options
  - --with-ld-opt=-ltcmalloc

使用gperftools定位nginx性能问题

- **gperftools库：**
  - https://github.com/gperftools/gperftools/releases
- **结果展示**
  - 文本展示
    - pprof --text
  - 图形展示：
    - pprof --pdf
    - 依赖graphviz
- **依赖：**
  - https://github.com/libunwind/libunwind/releases
- **模块：**
  - ngx_google_perftoos_module，通过--with-google_perftools_module启用

Syntax:	**google_perftools_profiles** file;
Default:	—
Context:	main

文本风格结果

2 0.9% 51.3% 166 73.5% ngx_epoll_process_events

1 0.4% 96.5% 2 0.9% ngx_open_and_stat_file

- **意义：每统计周期是10毫秒**

  第1列：当前函数的执行总共花的统计周期数

  第2列：当前函数执行时间的百分比

  第3列：当前函数及其之前的函数调用执行时间的百分比

  第4列：当前函数及其所调用函数消耗的统计周期数总和

  第5列：当前函数及其所调用函数执行时间总和的百分比

  第6列：函数名称

stub_status模块监控Nginx

模块：

ngx_http_stub_status_module，通过--with-http_stub_status_module启用模块

功能：

通过HTTP接口，实时监测nginx的连接状态。

统计数据存放于共享内存中，所以统计值包含所有worker进程，且执行reload不会导致数据清0，但热升级会导致数据清0

Syntax:	**stub_status;**
Default:	—
Context:	server, location

stub_status模块的监控项

Active connections: 1
server accepts handled requests
 2511 2511 4764
Reading: 0 Writing: 1 Waiting: 0

- **Active connections**
  - 当前客户端与Nginx间的TCP连接数，等于下面Reading、 Writing、 Waiting数量之和
- **accepts**
  - 自Nginx启动起，与客户端建立过的连接总数
- **handled**
  - 自Nginx启动起，处理过的客户端连接总数。如果没有超出worker_connections配置，该值与accepts相同
- **requests**
  - 自Nginx启动起，处理过的客户端请求总数。由于存在HTTP Keep-Alive请求，故requests值会大于handled值
- **Reading**
  - 正在读取HTTP请求头部的连接总数
- **Writing**
  - 正在向客户端发送响应的连接总数
- **Waiting**
  - 当前空闲的HTTP Keep-Alive连接总数

极客时间

扫码试看/订阅

《Nginx 核心知识100讲》