Cgroup - Linux的网络资源隔离

Hi，我是Zorro。这是我的微博地址，如果你有兴趣，可以来关注我呦。

这是我的博客地址，我会不定期在这里更新文章，如有谬误，欢迎随时指正。

另外，我的其他联系方式：

Email: mini.jerry@gmail.com

QQ: 30007147

由于本文不会涉及一些网络基础知识的讲解以及iproute2相关命令的使用的讲解，建议如果想要更好理解本文，之前应该对网络知识、tc命令和LARTC的文档有一定了解。如果本文中有什么知识点让不够清楚，可以结合LARTC文档一起服用。

想要直接上手配置cgroup的网络资源隔离的人，可以直接看本文倒数第二部分：**使用cgroup限制网络流量**。

今天我们来谈谈：

Linux的网络资源隔离

如果说Linux内核的cgroup算是个新技术的话，那么它的网络资源隔离部分的实现算是个不折不扣的老技术了。实际上是先有的网络资源的隔离技术，才有的cgroup。或者说是先有的网络资源的隔离才有的2.4、2.6版本的Linux内核，而现在的最主流的内核版本应该是3.10了（考虑到android手机的出货量，你公司那几千几万台服务器真的算是个零头对吧？）。好吧，Linux早在内核2.2版本就已经引入了网络QoS的机制，并且网络资源的隔离功能只是其所实现功能的一部分而已。无论如何，cgroup并没有再重新搞一套网络资源隔离的实现，而是直接使用了Linux的iproute2的traffic control（tc）功能。实际上网络资源隔离的文档真的不用我再多写什么了，我最亲爱的前同事＋朋友＋导师－－johnbull同志早已经在2003年的非典期间就因为无聊而完成了非常高质量的相关技术文档翻译工作，将这方面最权威的LARTC（Linux Advanced Routing & Traffic Control）文档翻译成了中文版。

英文版链接

中文版链接

曾经chinaunix的资深版主johnbull同志现在在新浪微博工作，所以经常在微博出没，如果对以上文档有兴趣和疑问的人可以直接去找他对质，传送门在此。

其实原则上说，本技术文章已经讲完了，但是为了不让大家有种上当受骗的感觉，我觉得我还是有必要从cgroup的角度再来讲讲tc，也算是对TC近几年发展做一个补充。

什么是队列规则

tc命令引入了一系列概念，其中我们最需要先理解的就是**队列规则**。它的英文名字叫做queueing discipline，在tc命令中也叫qdisc，或者直接简写为qd。我们先来看看它，有个感性的认识：

在我的虚拟机的centos7中，它是这样的：

```
[root@localhost Desktop]# tc qd ls
qdisc pfifo_fast 0: dev eno16777736 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
```

在我的台式机上装的archlinux（更新到了当前最新版的4.3.3内核）以及fedora 23上是这样的：

```
[root@zorrozou-pc0 zorro]# tc qd ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev enp2s0 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5.0ms interval 100.0ms ecn
```

在公司的服务器上是这样的：

```
[root@tencent64 /data/home/zorrozou]# tc qd ls
qdisc mq 0: dev eth1 root
qdisc pfifo_fast 0: dev tun0 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_121_54 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_135_194 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_123_25 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_121_112 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_123_207 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_123_82 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_117_111 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
```

从以上输出大家应该可以判断出来，这个所谓的qdisc是针对网卡的，每有一个网卡就会有一个qdisc。而且如果你用过ip命令并且比较细心的话，应该早就注意到ip ad sh的时候也会出现相关的信息：

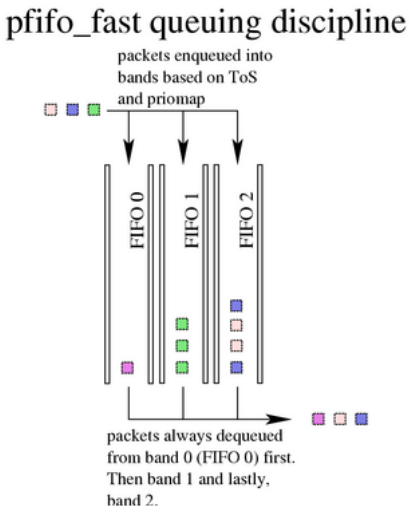
```
[zorro@zorrozou-pc0 ~]$ ip ad sh
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 34:64:a9:15:a2:17 brd ff:ff:ff:ff:ff:ff
    inet 10.18.73.69/24 brd 10.18.73.255 scope global dynamic enp2s0
       valid_lft 28283sec preferred_lft 28283sec
    inet6 fe80::3664:a9ff:fe15:a217/64 scope link
       valid_lft forever preferred_lft forever
```

虽然看上去有些高深莫测，但是qdisc其实是个挺简单的概念，它就是它字面的意思：队列规则，或者叫做排队规则。我们都知道，网络数据都是被封装成一个一个的数据包进行传输的。如果网卡相当于数据包要出发的大门的话，那么qdisc无非就是规定了这些包在出发前如果需要排队的话该怎么排。我们先拿这个叫做pfifo_fast的队列规则来举例子描述一下吧，这个qdisc实现了一个以数据包（package）为单位的fifo队列，实际上可以认为是实现了三个队列（叫做bands），给每个队列定了一个优先级，以实现带优先级的排队规则。我们举个现实中的例子再来说明一下，大家都应该有去公交车站排队的经验吧？（神马？作为中国人你从来不排队？）无论怎样，我们假定你是排队的。每来一次公交车，就相当于网卡处理一次队列中的数据包，而每个人就是一个数据包。那么我们一般人到了公交站，如果发现前面已经排了一队人，此时根据fifo（first in first out）的规则，我们会排在队列尾部。如果来车了，就从队列头的人先上车，车满就走，没上完的人继续等待。但是我们也知道，如果此时来了个孕妇或者大爷大娘等一些按照我们社会美德要求应该让他们优先的乘客的话，这些人应该有权利优先上车。那么怎么办呢？我们公交站台的解决办法一般是直接让他们去队列头插队就好，但是如果空间允许的话，我们可以考虑多建立一个队列。让这些可以优先上车的人排一个队，正常人排一个队，车来了先上优先级比较高的那个队列中的人，他们都上完了再让一般队列中上人车。这样就实现了一个简单的队列规则，大家根据自己的情况去选择排队就好了。

pfifo_fast实现了一个类似上述描述的队列规则，区别是它实现了3个优先级的队列（bands），每个数据包来了都根据自己的情况选择一个band进行排队，每个band都是fifo方式处理数据包。它总是先处理优先级最高的band，直到没有数据包了再处理下一个优先级的band，直到三个都处理完，或者本次处理不完，继续等着下次处理。那么数据包按什么规则进行选择自己该进入哪个band呢？这就是后面显示的**priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1**的含义，这个字段描述了一个priomap，可以理解为优先级位图，后面的16个不同的位，表示相关制如果为真时的数据包应该进入哪个队列，一共有0、1、2三个队列。而这个16位的位图标记，针对的就是我们IP报头中的TOS字段。根据IP协议的定义我们知道，TOS字段8位中的4位分别是用来标示最小延时、最大吞吐量、最大可靠性和最小消费四种数据包类型的，IP协议原则上会根据这些标示的不同以不同的QOS对上层交付不同的服务质量。这些不同的搭配理论上一躬有16种，这就是priomap所映射的优先级的概念。

> *如果你对TOS的概念还不熟悉，请自行补充网络相关基础知识。推荐的教材是《TCP／IP详解卷1》。*

pfifo_fast队列处理过程如图所示：



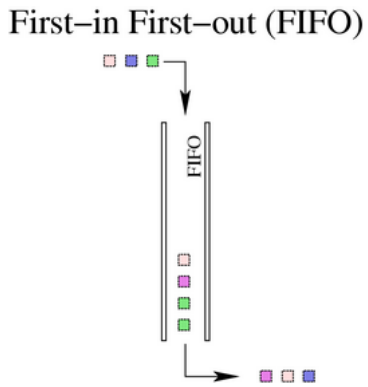
pfifo_fast一般情况下是内核对网卡默认选择的qdisc，它虽然提供了简单的优先级分类的支持，但是并没有提供可供修改的参数，就是说默认的优先级分类设置不能更改，也没有提供相关限速的功能。这个队列规则在一般情况下工作的都很稳定，但是最近Linux已经开始放弃使用这个qd作为默认的队列规则而改用一种叫做fq_codel的qdisc了。主要原因是，由于移动互联网的广泛应用，一种叫做Bufferbloat的现象影响越来越大了。

Bufferbloat

Bufferbloat现象最初是用来形容在一个分组交换网络上，路由器为防止丢包，往往buffer缓冲区都会实现的很大，但是这种过大的fifo缓冲区可能导致数据包buffer中等待时间过长而导致很多问题（后面会有分析）。再加上网络上TCP的拥塞控制算法的影响，以及很多商业操作系统甚至并不实现拥塞控制，

导致数据传输质量抖动很大（全局同步），甚至于达到服务不可用的状态。

后来我们发现，Bufferbloat这种现象比较广泛的存在在各种系统中，只要系统中使用了类似队列、缓存等机制的时候，就在某些极端状态下出现这种类似雪崩的现象。我们简要描述一下这个状态。我们先简单构建一个试用buffer的场景，如图所示：



根据图的描述，我们假定这个简单的fifo就是我们要的buffer系统，它在两个处理过程之间充当缓冲区的作用。每个请求从队列的上面进入排队，然后依次被下面的处理程序处理。大家应该知道buffer的作用：一个缓冲器的作用主要是弥补两个处理系统之间的速度差异，能够在一定程度的请求速度抖动的时候缓解处理速度慢而导致的请求失败。假设，后段处理请求的速度为1000个/s，每个请求平均长度为100byte，队列长队为1Mbyte，此时，如果请求突然增加到了2000个/s，那么这个压力直接压给后端是处理不过来的，每秒钟就要丢弃1000个包。所以我们使用一个buffer，可以让这一秒钟来的请求先处理1000个，然后有1000个排在队列中，下一秒处理。只要来的请求的抖动范围还算正常，我们的系统将会工作的良好，没有失败的请求。

对于一般的系统，我们发送的请求都是有延时要求的，鉴于我们的系统每秒钟可以处理1000个请求，所以每个请求的处理时间平均为1ms。而我们的系统基于目前的处理时间，对外提供了100ms的延时SLA，就是说，后端系统保证每个请求的处理时间是100ms以内，这已经很大了，是正常情况的100倍。于是前端的请求方，会根据后端给出的SLA在程序中设定一个超时时间，在这个例子中就应该是100ms，这可能意味着，程度调用后端系统，如果等待100ms还没有结果，那么将重试一次或者几次不等，之后应该会返回失败。场景就是这样一个场景，那么我们来看看究竟什么是bufferbloat?

假定现在因为业务问题，比如上线了一个秒杀的抢购活动，导致从前端发来的请求一瞬间远远大于后端的处理能力。比如，一秒钟内产生了10000次请求，这一万次请求都会立即进入队列中等待后端处理。因为后端的处理速度是1000次每秒，所以可以想像，当前在队列中的最后一个数据包至少要等待9秒钟才能处理到。实际上根本处理不到这最后一个请求，由于我们设置了100ms的超时时间，那么调用方将很快因为发现100ms中没有返回而重试一次，于是又来了将近10000个请求。这些请求都积压在了队列中，还没交给后端进行处理，如果交给了后端处理，后端肯定会因为压力变大处理变慢，而导致处理事件超过100ms的SLA，会在超时之后告诉前端本次请求失败（如果是这样实现的话），而现在由于队列的存在，并大量的积压请求，导致调用方不能明确的得知失败。所以一般都是等待至少一次超时重试一次再失败，当然也有很多情况会重试个4，5次也说不定。

无论如何，这突发的10000个请求的流量来了之后，如果平均每个请求100字节，这1M的缓冲区就已经满了，后续再有任何请求来，都会排在队列末尾，一直等到前面的请求处理完再处理这个请求，而此时因为整体处理时间很慢，要将此队列中的全部请求处理完需要9秒钟，无论如何，这个请求都已经超时失败了。这个时候后端服务一直满载的处理队列中的请求，而前端还不断有新请求源源不断的放进队列，但是由于超时，前端所有请求都是返回失败，后端所处理的请求也都是等待时间超过100ms的无效的请求，即使成功返回结果给前端，前端也不会要了。效果就是后端很忙，而整体服务却是不可用的。此时哪怕请求平均速度恢复到1000个每秒，服务也无法恢复。这就是一个典型的bufferbloat场景。

于是我们可以考虑一下这个场景会发生在什么地方？比如buffer比较大的路由器，由于tcp的流量控制和重试机制导致网络质量的抖动；比如一个后端的数据库系统为了能够承载更大的吞吐量而添加了队列系统；比如io调度；比如网卡调度；只要是大buffer的场景都会可能产生类似的问题。那么该如何解决这个问题呢？于是主动队列管理算法应运而出了。

AQM

AQM就是主动队列管理（Active Queue Management）的英文缩写，其思路就是对buffer缓冲的队列管理采取有效的主动管理手段，并不等待队列满之后才被动丢弃请求，而是在某个条件触发的情况下主动对请求进行丢弃，以防止类似Bufferbloat现象的发生。最简单的AQM思路就是监控队列长度，当队列长度一直维持在最大长度的时候，开始对新入队的数据包进行丢弃，直到使拥塞恢复（根据上面的例子可以想像，不断减少队列长度，就可以让新来的请求等待时间变短，直到可以正常服务）。这种做法虽然可以最终使拥塞恢复，但是整个过程并不十分理想，bufferbloat现象仍然存在。由于是对新入队数据包进行丢弃，所以容易在类似TCP拥塞控制的使用场景下引发**全局同步**现象，在很多场景下还会有死锁。所以我们需要更先进的队列管理算法。

RED算法

RED算法主要是为了解决全局同步现象而产生的算法，其基本思路是，通过监控平均队列长度来探测是否有拥塞，一旦发现开始拥塞，就以某一个概率从队列中（而不是队列尾）开始丢弃请求（在网络上也可以通过ecn通知连接有拥塞）。

对于RED来说，关键的可配置参数有这样几个：

min:最小队列长度。

max:最大队列长度。

probability:可能性，取值范围为0.00 - 1，一般可以理解为百分比，比如0.01为1%。

有了以上几个关键参数，RED算法就可以工作了，其工作的原理大概是这样的。首先，RED会对目前队列状态计算一个平均队列长度（算法采用的是指数加权平均算法计算的，在此不做更细节的说明），然后检查当前队列的平均长度是否：

1. 低于min：此时不做任何处理，队列压力较小，可以直接正常处理。
2. 在min和max之间：此时界定为队列感受到阻塞压力，开始按照某一几率P从队列中丢包，几率计算公式为：P = probability * (平均队列长度 − min)/(max − min)。
3. 高于max：此时新入队的请求也将丢弃。

所以probability可以理解为当队列感受到阻塞压力的时候，最大的丢包几率是多少。知道了这几个参数，我们就可以了解一下如何在Linux上进行RED的配置了，其实很简单，使用以下命令即可：

```
[root@zorrozou-pc0 zorro]# tc qd ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev enp2s0 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5.0ms interval 100.0ms ecn
[root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 root red limit 200000 min 20000 max 40000 avpkt 1000 burst 30 ecn adaptive
bandwidth 5Mbit
[root@zorrozou-pc0 zorro]# tc qd ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc red 8001: dev enp2s0 root refcnt 2 limit 200000b min 20000b max 40000b ecn adaptive
```

这样我们就将默认的qdisc规则改为了RED，解释一下相关参数：

首先是命令前部分：

```
tc qd add dev enp2s0 root
```

这部分没什么可解释的，唯一需要说明的是root参数，这个参数表示**根节点**，修改了这个参数描述的队列一般表示我们整个这个网卡所发出的数据包都用的是指定的规则，暂时我们还用不到其他节点，所以就只是root就可以了。另外请注意，目前所学习的队列规则只对发出的数据包有效。

之后是red参数，在这里描述使用什么队列规则。在之后丢失red这个队列规则所要使用的参数描述，具体可以通过man tc-red找到帮助。我们简单解释一下：

limit:此队列的字节数硬限制。配置的长度应该比max大。但是需要注意的是max和min的单位是数据包个数而不是字节数。

avpkt:平均包长度。这个参数是用根burst参数一起来计算平均队列长度的参数，所以选择一个合适的值对整体效果的影响较大。一般的推荐值为1000。

burst:字面含义是队列可以容纳的爆发流量。但是我们知道，爆发流量的承载是根据队列容量上限（limit）决定的，当一个大于当前网络带宽处理能力的爆发流量来临时，不能及时发出的数据包将缓存在队列中，队列满了就会丢包。所以实际影响爆发流量承载能力的是limit参数。当然我们建议的limit长度应该是不少于max+burst的长度，这样才能有实际意义。但是这个参数将对平均队列长度的变化速度产生影响，可以想像，如果我们想要支持队列能处理尽可能大的爆发流量的话，当队列突然变长的时候，应该让平均队列长度的计算结果变化没那么敏感，这样爆发流量来的时候丢包的可能性会减小。所以，这个值设置的越高，那么平均队列长度的计算敏感度就约小，变化速度将会变慢，反之变快。

bandwidth:用于在网络空闲的时候计算平均队列长度的参数，应该配置成你的网络的实际带宽大小。并不是说RED有限速作用。

ecn:ecn实际上是TCP/IP协议用来通知网络拥塞所实现的一个数据报字段。添加这个参数标示意味着，当RED检测到拥塞都是通过标记数据包的ecn字段来通知数据源端减少数据发送量，并且在实际队列长度达到limit限制之前丢不会丢弃数据包。

adaptive:算是一种更智能的probability参数的选择，添加了这个参数之后就可以不用人为指定一个固定的probability了，当平均队列长度超过(max-min)/2时，RED会动态的根据情况让probability的值在1%到50%之间变化。具体描述参见这里。

以上就是RED队列规则的配置方法和意义，其作用主要是缓解全局同步的问题。但是我们在实际使用的时候发现，RED的min、max、probability这些参数的选择在实际场景中可能会根据情况变化而改变才是最优的，但是RED的配置不能自适应这些变化。并且实际上在很多特定的网络负载下依然会导致TCP的全局同步。这些缺陷促使我们寻找更优秀的方式来解决这些问题。

> *内核还实现了另一个队列规则叫做choke，其所有配置参数跟RED完全一样，区别是，RED是通过字节为单位进行队列控制，而choke是以数据包为单位。更多帮助请：man tc-choke*

CoDel算法

CoDel算法是另一种AQM算法，其全称是Controlled Delay算法。是由Van Jacobson和Kathleen Nichols在2012年实现的。具体描述参见Controlling Queue Delay。CoDel采用了另外一种角度来观察队列满载的问题，其出发点并不是对队列长度进行控制，而是对队列中的数据包的驻留时间进行控制。事实上如果我们将管理方式由队列长度控制变成等待时间控制的时候，bufferbloat就可以彻底解决了，这也是更先进的AQM算法所用的方式。我们仔细观察bufferbloat问题，会发现，引起这个问题的重要原因就是数据包在队列中的驻留时间过长，超过了有效的处理时间（SLA定义的时间或者重试时间），导致处理到的数据包都已经超时。

首先我们根据我们的业务设计，确定出请求在队列中正常情况应该驻留多久。我们还是假定这样一种场景，根上面bufferbloat中描述的例子差不多：后端处理速度是1000次每秒，就是1ms可以处理一个请求，而队列平均长度一般为5，就是说一个新请求进入队列之后，发现前面还有5个请求在等待，那么这个新请求的处理时间大约为6ms（在队列中等待5ms）。那么请求在队列中的驻留时间正常情况下基本为5ms。而我们服务的SLA确定的时间是100ms（由诸如服务超时时间或者所在网络的最大RTT时间等条件确定），就是说，服务应确保在100ms内给出反馈，这个时间叫做interval time，如果超过这个时间应该返回失败。针对这样的情况，我们可以根据请求驻留时间的情况来描述一个动态长度的队列，当一个请求入队之后，对其驻留时间（sojourn time）进行追踪，以正常的情况作为其目标驻留时间（target time），在这个例子中是5ms，就是说一般情况下，我们期望请求在队列中的驻留时间不高于5ms。由于业务的超时时间或者说我们提供的SLA处理时间是100ms，所以，在这个队列中驻留超过100ms的请求都应该丢弃（从队列头开始），因为即使处理完成它们也没有意义了。丢弃将持续到队列中的请求等待时间回到理想的target time为止，并且队列长度整体不大于队列容量上限。这样就根据驻留时间维持了一个动态长度的队列，这个队列中的所有请求理论上都应该等待100ms以内，要么被正常处理掉，要么被丢弃。这就是CoDel算法的基本思路。

为了有助于大家理解，我们再详细一点描述一下这个算法的处理过程：

CoDel算法对队列状态维护一个状态机，进行队列dequeue处理的时候，先判检查队列头请求的驻留时间（sojourn time）是否大于target time，如果不大于target time，就直接dequeue；如果大于（target time）的请求维持了interval time这么长的时间，则队列应该进入dropping状态开始丢包。这种丢包状态将可能维持一段时间，这段时间的长度将根据情况而定（驻留时间一直处在target以上，并且下一个包丢弃的时间采用逆平方根运算（inverse-square-root）,公式为：

```
t（第一次取now，以后取上次的值） + interval / sqrt(count)）
```

count的取值为丢弃包的个数，如果count大于2则count＝count－2，其他情况count取值为1。直到驻留时间小于target time，就退出dropping状态。

算法的伪代码描述参见这里。

> *我们之所以要如此详细的描述bufferbloat问题以及其解决方案，尤其是CoDel算法，原因是其不仅仅被用在网络的分组交换和路由的处理上。除了TC的队列规则外，CoDel当前还被用在了内核TCP协议栈的拥塞控制中，并且rabbitmq也已经把这个算法应用于消息队列的延时控制了,参见。这个算法在数据中心的应用场景下，是一个非常好的解决队列阻塞的方案。*

了解了以上知识之后，我们来看一下再Linux上如何配置一个CoDel的队列规则，我们刚才已经将队列规则改为RED了，此时如果要将其改为CoDel，需要先删除RED的队列规则，再添加新的队列规则：

```
[root@zorrozou-pc0 zorro]# tc qd del dev enp2s0 root
[root@zorrozou-pc0 zorro]# tc qdisc add dev enp2s0 root codel limit 100 target 4ms interval 30ms ecn
[root@zorrozou-pc0 zorro]# tc qd ls dev enp2s0
qdisc codel 8002: root refcnt 2 limit 100p target 4.0ms interval 30.0ms ecn
[root@zorrozou-pc0 zorro]# tc -s qd ls dev enp2s0
qdisc codel 8002: root refcnt 2 limit 100p target 4.0ms interval 30.0ms ecn
 Sent 5546 bytes 39 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
  count 0 lastcount 0 ldelay 0us drop_next 0us
  maxpacket 0 ecn_mark 0 drop_overlimit 0
```

tc的-s参数相信你已经明白什么意思了。来说一下codel队列规则的相关参数：

limit:队列长度上限，如果超过这个长度，新来的数据包将被直接丢弃。单位为字节数，默认值为1000.

target && **interval**:这两个参数相信大家已经明白是什么意思了，根据自己的场景进行配置就好了。

ecn && **noecn**:这个参数的含义根RED中的一样，默认是开启的ecn方式通知源端，不丢包。

大家也可以直接使用codel规则的默认参数，就是其他参数都省略即可。我们来看看什么效果：

```
[root@zorrozou-pc0 zorro]# tc qd del dev enp2s0 root
[root@zorrozou-pc0 zorro]# tc qdisc add dev enp2s0 root codel
[root@zorrozou-pc0 zorro]# tc -s qd ls dev enp2s0
qdisc codel 8003: root refcnt 2 limit 1000p target 5.0ms interval 100.0ms
 Sent 8613 bytes 33 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
  count 0 lastcount 0 ldelay 0us drop_next 0us
  maxpacket 0 ecn_mark 0 drop_overlimit 0
```

fq-codel队列规则

在比较老版本的Linux内核上，由于当时还没实现基于CoDel算法的队列规则，所以一直使用的是pfifo_fast作为默认队列规则。作为一个简单的队列规则，pfifo_fast越来越不能适应Linux的发展需要。这个发展主要指的是Linux作为android系统的操作系统内核被广泛用在了手机等移动互联设备上。在移动互联网的场景下，网络延时问题变的更普遍，而导致网络上的bufferbloat问题变成了急需解决的问题。于是，CoDel的算法引入变的非常必要。CoDel算法虽然比较高质量的解决了bufferbloat问题，但是并没有解决多链接处理的公平性问题。这个公平性问题其实也比较好理解，因为网络有不同的传输要求，某些传输数据量很大，但是延时要求不大，某些则是数据量很小，但是延时要求很高（IP协议TOS字段所描述的情况）。如果各种链接占用同一个队列，那么数据量大的的连接势必数据包就更多，那么从概率上讲，这样的连接挤占队列的能力就更强。而主动队列管理一般都是以ecn或者丢包为手段的，如果丢弃的是那些延时要求较高的连接的数据包，又会对用户的服务质量感受造成很大的影响。所以，最好的办法就是实现一个针对每一个数据流(flow)公平的CoDel队列规则，就是fq-codel。

fq-codel叫做flow queue codel调度，因为其特点也被叫做fair queue codel（完全公平）。fq-codel为每个需要使用网络的flow创建一个单独的队列（实际上是默认实现1024个队列，使用五元组hash给相关flow选择一个队列），队列之间使用针对字节的DRR(Deficit Round Robin)调度算法进行调度。它的工作方式是跟踪每个队列的当前差额（*deficit*）的字节个数。这个差额的初始值可以用quantum参数指定。每当一个队列获得发送数据（出队）的机会时就开始发送数据包，并根据发送的数据包的字节数减少deficit的值，直到这个值变为负值的时候，对其增加一个quantum的大小，并且本队列发送结束，调度下一个队列。

这意味着，如果目前有两个队列，一个队列中的数据包长度都是quantum/3这么大，而另一个队列中的数据包长度每个都是一个quantum长度的话，调度器处理第一个队列的时候，每次处理3个数据包，而第二个队列就只能处理1个数据包。这意味着DRR算法对每个队列发送数据的时候是针对字节数计数，不会因为数据包数的大小而有差别。

quantum取值的大小决定了调度周期的粒度，所以也就决定了调度器的调度开销。当网络带宽比较小的时候，推荐的设置是从默认的MTU的值来取quantum的值，并可以考虑适当减小这个值。

不同于标准DRR调度的地方是，我们的调度器将所有flow队列分成了两个sets。实际上可以认为所有队列有两个分类，一类里面都是new flow，针对新建的网络连接；而另一类是old flow，针对原来机已经建立的网络连接。

Interval

这个值的意义根CoDel算法中的语义完全一样，是用来确定最小延时时间的取值不至于导致数据包长时间在队列里堆积。最小延时的取值必须根据上一个

周期interval检查的经验而得来，应该被设置为，数据包通过网络瓶颈点发给对端之后，能够接收到对端返回的确认的最差RTT时间。

默认间隔时间值为100ms。

Target

这个值的意义根CoDel算法中的语义完全一样，是用来设定在FQ－CoDel的每个队列中数据包的最小延时时间（可以等待的最长时间）的。最小延时时间是通过追踪本地最小队列延时的经验得来的。

默认的Target值为5ms，但是这个值应该根据本地的网络情况得来，最少应配制成本地网络的mtu长度的数据包在相应的带宽环境下发送的时间。（如：本地网卡mtu为1500，带宽为1Mbps的情况下，应配置为15ms。）

下面简述一下fq-codel的处理过程：

FQ-CoDel的入队（enqueue）：

入队由三个步骤组成：根据flow特点进行分类选择一个队列，记录数据包入队时间并记账（bookkeeping），另外如果队列满了还会丢弃数据包。

分类的时候会根据数据包的源、目的ip；源、目的端口和使用的协议（五元组）并参杂一个随机数，用这个值对队列个数取模运算，得出把这个flow放到哪个队列中。

FQ-CoDel的出队（dequeue）：

队列规则的绝大多数工作都是在出队的时候做的。分三个步骤：选择从那个队列发送数据包；dequeue数据包（在所选队列中处理CoDel算法）；记账（bookkeeping）；

在第一部分处理的过程中：调度器先查找new list队列，对这个list中的每个队列进行处理处理，如果队列有负的赤字（negative deficit）说明起已经被出队了至少一个quantum的字节数，那么就说明这个队列已经不再是new队列了，则追加到old list中，并且给其增加一个quantum的字节数的deficit，然后处理new list中的下一个队列。

如果选择的队列不是上述情况，就说明这是一个new队列，则对其dequeue。如果new列表为空，则开始处理old列表，处理过程根上述过程类似。

选择好处理哪个queue之后，CoDel算法就会作用于这个队列。这个算法可能在返回需要dequque的数据包之前，先删除队列中的一个或者多个数据包，数据包的删除是从队列头开始的。

最后，如果CoDel没有返回需要dequeue的数据包，或者队列为空，调度器将根据情况做这两件事的其中一个：如果队列是new列表中的队列，则将其移动到old列表的最后一个。如果队列是old列表中的队列，那么这个队列讲从old列表中删除，直到下次这个队列中有数据包需要处理的时候，就再把它加到new列表中。如果所有队列中都没有需要dequeue的数据包之后，就对所有队列重来一次上述调度过程。

如果调度算法返回了一个需要dequeue的数据包，处理过程将会先去处理deficit数字，然后对数据包进行相关dequeue处理。

检查new列表并把符合条件的队列移动到old列表这个过程会因为可能存在的无限循环而导致饥饿。则是因为当某一个数据流符合一个速率进行小包发送的时候，这个队列会在new列表中重现，而导致调度器一直无法处理old列表。预防这种饥饿的方法是，在第一次讲队列移动到old列表的时候，强制跳过不再检查。

以上过程更详细的描述参见。我们再来看看如何配置一个fq-codel队列规则。跟刚才步骤类似：

```
[root@zorrozou-pc0 zorro]# tc qd del dev enp2s0 root
[root@zorrozou-pc0 zorro]# tc -s qd ls dev enp2s0
qdisc fq_codel 0: root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5.0ms interval 100.0ms ecn
 Sent 7645 bytes 45 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
  maxpacket 0 drop_overlimit 0 new_flow_count 0 ecn_mark 0
  new_flows_len 0 old_flows_len 0
```

其实我们会发现，作为默认的队列规则，删除了原来配置的队列规则之后，显示的就是fq-codel了，默认参数就是显示的这样了。这个队列规则包含的参数包括：

limit

flows

target

interval

quantum

ecn | noecn

帮助可以参见man tc-fq_codel。唯一需要再稍作解释的就是flows，这个参数决定了有少个队列，默认1024。

另外，内核还提供了一个fq队列，实际上就是fq-codel不带codel的一个基于DRR算法的公平队列。这里没有更多参考，你可以直接使用这个队列。

> *本节涉及到了一个负载均衡算法，DRR- 基于赤字的轮训算法。实际上内核也实现了一个专门的DRR调度队列，大家可以参考man tc-drr。关于这个算法本身的描述请自行查找资料。*

SFQ随机公平队列

首先我要引用LARTC中文版中对SFQ队列的讲解，毕竟这已经足够权威了：

> *SFQ(Stochastic Fairness Queueing，随机公平队列)是公平队列算法家族中的一个简单实现。它的精确性不如其它的方法，但是它在实现高度公平的同时，需要的 计算量却很少。SFQ 的关键词是"会话"(或称作"流")，主要针对一个 TCP 会话或者 UDP流。流量被分成相当多数量的 FIFO 队列中，每个队列对应一个会话。数据按照 简单轮转的方式发送，每个会话都按顺序得到发送机会。这种方式非常公平，保证了每一个会话都不会没其它会话所淹没。SFQ 之所以被称为"随机"，是因为它并不是真的为每一个会话创建一个队列，而是使用一个散列算法，把所有的会话映射到有限的几个队列中去。因为使用了散列，所以可能多个会话分配在同一个队列里，从而需要共享发包的机会，也就是共享带宽。为了不让这种效应太明显，SFQ 会频繁地改变散列算法，以便把这种效应控制在几秒钟之内。有很重要的一点需要声明：只有当你的出口网卡确实已经挤满了的时候，SFQ 才会起作用！否则在你的 Linux 机器中根本就不会有队列，SFQ 也就不会起作用。稍后我们会描述如何把 SFQ 与其它的队列规定结合在一起，以保证两种情况下都比较好的结果。特别地，在你使用 DSL modem 或者 cable modem 的以太网卡上设置 SFQ 而不进行任何进一步地流量整形是无谋的！*
> *SFQ 基本上不需要手工调整：* ***perturb****:多少秒后重新配置一次散列算法。如果取消设置，散列算法将永远不会重新配置（不建议这样做）。10 秒应该是一个合适的值。* ***quantum****:一个流至少要传输多少字节后才切换到下一个队列。却省设置为一个最大包的长度(MTU 的大小)。不要设置这个数值低于 MTU！*
> *如果你有一个网卡，它的链路速度与实际可用速率一致——比如一个电话MODEM——如下配置可以提高公平性：*

```
# tc qdisc add dev ppp0 root sfq perturb 10
# tc -s -d qdisc ls

qdisc sfq 800c: dev ppp0 quantum 1514b limit 128p flows 128/1024 perturb 10sec
Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

> *"800c:"这个号码是系统自动分配的一个句柄号，"limit"意思是这个队列中可以有 128 个数据包排队等待。一共可以有 1024 个散列目标可以用于速率审计，而其中 128 个可以同时激活。(no more packets fit in the queue!)每隔 10 秒种散列算法更换一次。*

以上是对SFQ队列的权威解释，但是毕竟时过境迁，目前的实现稍有不同。现在的SFQ在原有队列的基础上实现了RED模式，就是针对每一个SFQ队列，都可以用RED算法来防止bufferbloat问题。目前的RED跟SFQ队列规则的关系有点像codel跟fq_codel队列规则之间的关系，它们一个是基础版算法的队列实现，另一个是其多队列版。

新版中需要解释的参数:

redflowlimit:用来限制在RED模式下的SFQ的每个队列的字节数上限。

perturb:默认值为0，表示不重新配置hash算法。原来为10，单位是秒。

depth:限制每一个队列的深度（长度），默认值127，只能减少，单位包个数。

如果需要配置一个RED模式的SFQ，操作方式如下:

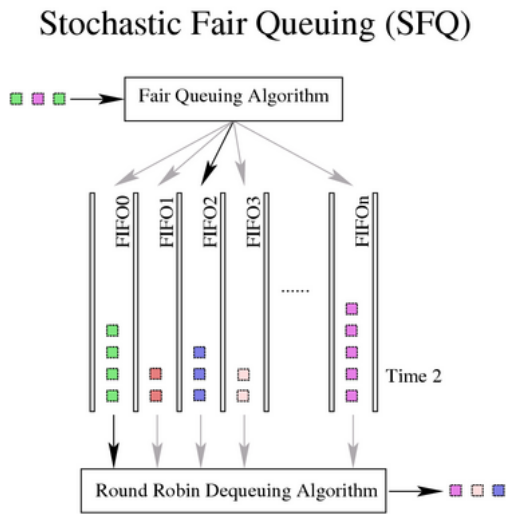
```
tc qdisc add dev eth0 parent 1:1 handle 10: sfq limit 3000 flows 512 divisor 16384 redflowlimit 100000 min 8000 max 60000
probability 0.20 ecn headdrop
```

更多的帮助情参阅:

```
man tc-sfq
```

> *内核还给我们提供了一个名叫sfb的随机公平队列，相对sfq来说，sfb的意思就是采用的blue算法对每个队列进程处理。什么是blue算法？这是相对于red来说的（有红的算法，也要有蓝的）。我们不对BLUE算法做更详细的解释了，大家有兴趣可以自行查找资料。*

SFQ的结构如下:



PIE比例积分控制队列

PIE是Proportional Integral controller Enhanced的简写，其中文名称是加强的比例积分控制。比例积分控制是非常有名的一种工控算法。想要详细了解这个方法的，可以自行查阅相关资料。而在tc的队列规则中，pie是内核帮我们实现的另一个用来解决bufferbloat问题的AQM机制。其控制思路跟CoDel一样，都是针对请求的延时进行控制而不是队列长度，但是其对超时请求处理方法跟RED一样，都是随机对数据包进行丢弃。

PIE是根据队列中请求的延时情况而对不同级别的拥塞做出相关的相应动作的（比如丢包），严格来说，是根据队列中请求延时时间的变化率（就是当前延时时间与目标延时时间的差值与时间的积分）来判断。这就能做到影响算法参数值选择是根据稳态感受的变化而变化的，目的就是可以让算法本身在各种网络阻塞的情况下都能自动调节以优化性能表现。

PIE包括三个简单的必需组件：1.入队时的随机丢弃；2.周期的更新丢弃可能性比率（probability）；3.对延时（latency）进行计算。当一个请求到达队列时（入队之前），会被评估这个请求是否会被随机丢弃。丢弃的几率会根据目前的延时状态和目标延时（target）的差距（比例控制）以及队列的延时是否变长或者变短（积分控制）的状态，每隔一定时间周期（tupdate）进行更新。队列的延时是通过直接测量请求的等待时间或计算队列长度和出队速率获得的。

跟其他最先进的AQM算法一样，当一个数据包到达时PIE会根据一个随机丢弃的可能性p来丢弃数据包，p的计算方式如下：

1. 首先根据以下公式估计当前队列延时： est_del = qlen/depart_rate;
2. 计算丢弃可能性几率p： p = p + alpha * (est_del - target_del) + beta * (est_del - est_del_old); est_del_old = est_del.

以上计算过程会按一定时间周期进行估算，周期的时间由tupdate参数指定，est_del是当前周期的队列延时，est_del_old是上一个周期的队列延时，target_del是目标延时。qlen表示当前队列长度。

alpha是用来确定当前的延迟与目标延时的偏差将如何影响丢弃概率。beta值会对整个p的估算起到另一个校准作用，这个作用通过目前的延时是在上升还是在下降进行估算的。请注意p的运算是一个逐渐达到的过程（积分过程），并不是一步达到的。在运算p的时候，为了避免校准过程中比较大的波动，我们一般是对p做小的增量调整。假设p在1%的范围内，那么我们希望单步校准的幅度也比较小，比如0.1%，那么alpha和beta也都要足够小，。但是如果p的值更高了，比如说达到了10%，在这种情况下，我们的单步教准的幅度也希望更大，比如达到1%。所以我们在p取值的每一个量级范围内，都可能需要一个单步的调教幅度的取值范围，在必要的情况下p可能会精确到0.0001%。这个单步调校的范围可以通过类似这样一个方式实现：

```
if (drop_prob_ < 0.000001) {
    drop_prob_ /= 2048;
} else if (drop_prob_ < 0.00001) {
    drop_prob_ /= 512;
} else if (drop_prob_ < 0.0001) {
    drop_prob_ /= 128;
} else if (drop_prob_ < 0.001) {
    drop_prob_ /= 32;
} else if (drop_prob_ < 0.01) {
    drop_prob_ /= 8;
} else if (drop_prob_ < 0.1) {
    drop_prob_ /= 2;
} else {
    drop_prob_ = drop_prob_;
}
```

对p进行调校的目标是让p稳定下来，稳定的条件就是当队列的当前延时等于目标延时，并且延时状态已经稳定的情况（就是说est_del等于est_del_old）。alpha和beta的取值实际上就是一个权重值，如果alpha较大则丢弃几率对延时偏移（latency offset即相对于目标延时的差距）更敏感，如果beta较大则丢弃几率p对延时抖动（latency jitter即相对于上周期延时的差距）更敏感。

计算周期tupdate参数也是一个让整个校准过程能够稳定发挥效果的重要参数，当我们配置更快的tupdate周期，并且alpha和beta的值相同时，则周期增益效果更明显。请注意alpha和beta的配置单位是hz，由于在上面的计算公式中表示的不明显，所以这可能会成为配置出错的地方。

请注意，丢弃可能性p的计算不仅与当前队列延时的估算有关，还与延时变化的方向有关，就是说，延时变大或者变小都会影响计算。延时变化的方向可以从当前队列延时和之前一个周期的队列延时进行比较来确定。这就是采用标准的比例积分控制算法对队列的延时进行控制。

队列的出队速率可能会经常波动，造成这种情况的原因是我们可能与其它队列共享同一个连接设备，活着链路的容量波动。在无线网络的情况下，链路的波动尤其常见。因此，我们通过以下方法直接测量出队速率：

当队列中有足够的数据时，才进入测量周期：

```
qlen > dq_threshold
```

进入测量周期之后，在数据包出队时： dq_count = dq_count + deque_pkt_size;

然后判断dq_coune是不是高于采样阈值： if dq_count > dq_threshold then

```
depart_rate = dq_count/(now-start);

dq_count = 0;

start = now;
```

我们只在队列中存在足够的数据的时候才计算出队速率，就是当队列长度超过deq_threshold这个阈值的时候。这是因为时不时出现的短的和非持久性的爆发数据流量进入空队列时会使测量不准确。参数dq_count表示从上次测量之后离开的字节数，一旦这个值超过了deq_threshold阈值，我们就得到一次有效的测量采样。在数据包长度在1k到1.5k长度的时候，我们建议dq_count的值为16k，这样的设置既可以让我们有足够长的时间周期来对出队速率做平均，也能够足够快的反馈出出队速率的突然变化。这个阈值并不影响系统的稳定性。

除了上面的基本算法描述以外，PIE算法该提供了一些其它增强功能来提升算法的性能：

网络流量往往都会有一定的自然波动，当队列的延时因为这样的波动而出现临时性的“虚假”上涨的时候，我们不希望在这样的情况下引起不必要的丢包。所以，PIE算法实现了一个自动开启和关闭算法的机制，当队列长度不足缓冲区长度的1/3时，算法是不会生效的，此时处于关闭状态，当队列中的数据量超过了1/3这个阈值的时候，算法自动打开，开始对队列中的数据进行处理。当阻塞情况完全恢复的时候，就是说丢弃概率、队列长度和队列延时都为0的时候，PIE的作用关闭。

虽然PIE采用随机丢弃的策略来处理入队的数据包，但是仍然可能会有几率因为丢弃的数据包很连续或者很稀疏而导致丢弃效果偏离丢弃几率p。这就好比抛硬币问题，虽然概率上出现正面或者反面的几率都是50%，但是当你真的去抛硬币的时候，仍然可能碰见连续多次的出现正面或者反面的情况。所以，我们引入了一种“去随机”的丢弃机制来防止这样的事情发生。我们引入了一个参数prob，当发生丢弃的时候，这个参数被重置为0，当数据包到达进行丢弃判断的时候，prob参数也会进行累加，累加的值是每次计算丢弃概率得到p这个值的总量。prob会有一个阈值下限和一个上限，当累计的prob低于阈值下线的时候，我们不丢包，直接入队，当高于阈值上限的时候，我们无论几率如何，强制丢包。只有当prob在阈值下限和上限之间时，我们才按照p的几率丢弃数据包。这样就能保证，如果几率导致连续没丢包，积累到一定程度后一定会丢包，另一方面，如果丢包，则prob一定在下限以下，则下一个包一定会入队，以防止问题的发生。

关于PIE更多的资料，可以参考这里。

TBF队列

以上的算法主要都是解决bufferbloat问题的。我们可以看到Linux内核为了适应移动互联网的环境做了很多努力。而接下来我们要介绍的TBF（令牌痛过滤器）是我们遇到的第一个可以对流量进行整形（就是限速）的算法。自它诞生到现在，基本功能没有什么太大变化，毕竟token bucket filter算法已经是一个非常经典的限速算法了。所以我们只需要引用LARTC中的讲解即可：

令牌桶过滤器(TBF)是一个简单的队列规定：只允许以不超过事先设定的速率到来的数据包通过，但可能允许短暂突发流量朝过设定值。TBF 很精确,对于网络和处理器的影响都很小。所以如果您想对一个网卡限速，它应该成为您的第一选择！TBF 的实现在于一个缓冲器(桶)，不断地被一些叫做“令牌”的虚拟数据以特定速率填充着。 (token rate)。桶最重要的参数就是它的大小，也就是它能够存储令牌的数量。每个到来的令牌从数据队列中收集一个数据包，然后从桶中被删除。这个算法关联到两个流上——令牌流和数据流，于是我们得到 3 种情景：• 数据流以等于令牌流的速率到达 TBF。这种情况下，每个到来的数据包都能对应一个令牌，然后无延迟地通过队列。

• 数据流以小于令牌流的速度到达 TBF。通过队列的数据包只消耗了一部分令牌，剩下的令牌会在桶里积累下来，直到桶被装满。剩下的令牌可以在需要以高于令牌流速率发送数据流的时候消耗掉，这种情况下会发生突发传输。

• 数据流以大于令牌流的速率到达 TBF。这意味着桶里的令牌很快就会被耗尽。导致 TBF 中断一段时间，称为“越限”。如果数据包持续到来，将发生丢包。

最后一种情景非常重要，因为它可以用来对数据通过过滤器的速率进行整形。令牌的积累可以导致越限的数据进行短时间的突发传输而不必丢包，但是持续越限的话会导致传输延迟直至丢包。

请注意，实际的实现是针对数据的字节数进行的，而不是针对数据包进行的。

即使如此，你还是可能需要进行修改，TBF 提供了一些可调控的参数。第一个参数永远可用：

limit/latency

limit 确定最多有多少数据（字节数）在队列中等待可用令牌。你也可以通过设置 latency 参数来指定这个参数，latency 参数确定了一个包在 TBF中等待传输的最长等待时间。后者计算决定桶的大小、速率和峰值速率。

burst/buffer/maxburst

桶的大小，以字节计。这个参数指定了最多可以有多少个令牌能够即刻被使用。通常，管理的带宽越大，需要的缓冲器就越大。在 Intel 体系上，10 兆 bit/s 的速率需要至少 10k 字节的缓冲区才能达到期望的速率。如果你的缓冲区太小，就会导致到达的令牌没有地方放（桶满了），这会导致潜在的丢包。

mpu

一个零长度的包并不是不耗费带宽。比如以太网，数据帧不会小于 64 字节。Mpu(Minimum Packet Unit，最小分组单位)决定了令牌的最低消耗。

rate

速度操纵杆。参见上面的 limits！如果桶里存在令牌而且允许没有令牌，相当于不限制速率(缺省情况)。If the bucket contains tokens and is allowed to empty, by default it does so at infinite speed. 如果不希望这样，可以调整入下参数：

peakrate

如果有可用的令牌，数据包一旦到来就会立刻被发送出去，就象光速一样。那可能并不是你希望的，特别是你有一个比较大的桶的时候。峰值速率可以用来指定令牌以多块的速度被删除。用书面语言来说，就是：释放一个数据包，但后等待足够的时间后再释放下一个。我们通过计算等待时间来控制峰值速率然而，由于 UNIX 定时器的分辨率是10 毫秒，如果平均包长 10k bit，我们的峰值速率被限制在了 1Mbps。

mtu/minburst

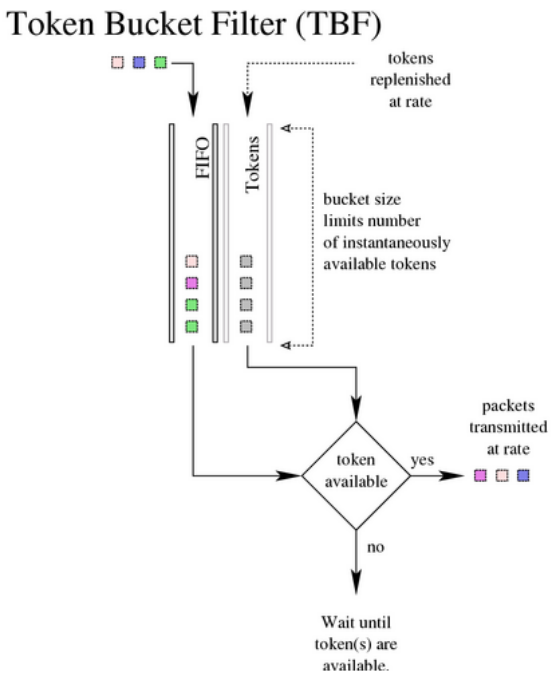
但是如果你的常规速率比较高，1Mbps 的峰值速率对我们就没有什么价值。要实现更高的峰值速率，可以在一个时钟周期内发送多个数据包。最有效的办法就是：再创建一个令牌桶！这第二个令牌桶缺省情况下为一个单个的数据包，并非一个真正的桶。要计算峰值速率，用 mtu 乘以 100 就行了。 (应该说是乘以 HZ 数，Intel体系上是 100，Alpha 体系上是 1024)

这是一个非常简单而实用的例子：

```
# tc qdisc add dev ppp0 root tbf rate 220kbit latency 50ms burst 1540
```

为什么它很实用呢？ 如果你有一个队列较长的网络设备，比如 DSL modem 或者cable modem 什么的，并通过一个快速设备(如以太网卡)与之相连，你会发现上载数据绝对会破坏交互性。这是因为上载数据会充满 modem 的队列，而这个队列为了改善上载数据的吞吐量而设置的特别大。但这并不是你需要的，你可能为了提高交互性而需要一个不太大的队列。也就是说你希望在发送数据的时候干点别的事情。上面的一行命令并非直接影响了 modem 中的队列，而是通过控制 Linux 中的队列而放慢了发送数据的速度。把 220kbit 修改为你实际的上载速度再减去几个百分点。如果你的 modem 确实很快，就把“burst”值提高一点。

以上为引用原文内容，请原谅我的懒惰。TBF结构图如下：



分类(class)、过滤器(filter)以及HTB

基于目前我们已经知道的这些内容，我们已经可以在一个运行着比较复杂的网络服务的系统环境中按照网络的数据流为调度对象，建立一个比较公平的队列环境了，并且还能避免bufferbloat现象。比如fq-codel、sfq等队列规则都能做到。这也是内核目前选择fq-codel作为默认队列规则的初衷。实际上这已经可以适应绝大多数场景了。

但是在一些QoS要求更高的场景中，我们可能需要对网络流量的服务做更细节的分类，来实现更多的功能。比如说我们有这样一个场景：我们的服务器上运行了一个web服务，对外服务端口是tcp的80，还运行了一个邮件服务，对外服务协议是smtp的tcp的25端口，可能还会开一个sshd以便管理员可以远程控制，其端口为22。我们的对外带宽一共为10Mbps。我们想要做到这样一种效果，当所有服务都很繁忙的需要占用带宽时，我们希望80端口上限不超过6Mbps，25端口上限不超过3Mbps，而22端口1Mbps足够了。当其它端口不忙的时候，某个端口可以突破自己的上限带宽设置能达到10Mbps的带宽。这种网络资源分配策略跟cgroup的cpushare方式的分配概念类似。

当我们的需求负载到类似这样的程度时，我们会发现以上的各种队列规则都不能满足需求，而能满足需求的队列规则都起吗必需实现一个功能，就是对数据包的分类（class）功能，并且这个分类要能够人为指定分类策略（实际上pfifo_fast本身对数据包进行了分类，但是并不能人为改变分类策略，所以我们仍然把它当成不可分类的队列规则）。比如针对当前的例子，我们就至少需要三个分类（可以认为就是三个队列），然后把从80端口发出的数据包都排进分类1里，从25端口发出的数据包排进分类2里，再将22端口发出的数据包放到分类3里。当然如果你的服务器还有别的服务也要用网络，可能还要额外配置一个分类或者共用以上某一个分类。

在这个描述中，我们会发现，当需求确定了，分类也就可以确定了，并且如何进行分类（过滤方法）也就可以确定了。如果我们还是把数据包比作去公交

车站排队的人的话，那么可分类的队列规则就相当于公交站有个管理人员，这个管理人员可以根据情况自行确定目前乘客可以排几个队、哪个队排什么样特征的人。自然，我们需要在每个乘客来排队之前，根据确定好的策略对乘客进行过滤，让相关特征的乘客去相应的队伍。这个决定乘客所属分类的人就是过滤器（filter）。

以上是我对这两个概念的描述，希望能够帮助大家理解。相关概念的官方定义在此。

由于相关知识的细节说明在LARTC中已经有了更细节的说明，我们再次不在废话。我们直接来看使用HTB（分层令牌桶）队列规则如何实现上述功能，其实无非就是以下系列命令：

首先，我们需要先讲当前网卡的队列规则换成HTB，保险起见，可以先删除当前队列规则再添加：

```
[root@zorrozou-pc0 zorro]# tc qd del dev enp2s0 root    [root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 root handle 1: htb
default 30
```

default参数的含义就是，默认数据包都走标记为30的类（class）。然后我们开始建立分类，并对各种分类进行限速：

```
[root@zorrozou-pc0 zorro]# tc cl add dev enp2s0 parent 1: classid 1:1 htb rate 10mbit burst 20k
[root@zorrozou-pc0 zorro]# tc cl add dev enp2s0 parent 1:1 classid 1:10 htb rate 6mbit ceil 10mbit burst 20k
[root@zorrozou-pc0 zorro]# tc cl add dev enp2s0 parent 1:1 classid 1:20 htb rate 3mbit ceil 10mbit burst 20k
[root@zorrozou-pc0 zorro]# tc cl add dev enp2s0 parent 1:1 classid 1:30 htb rate 1mbit ceil 10mbit burst 20k
```

这样我们建立好了一个root分类，id为1:1，速率上限为10mbit。然后在这个分类下建立了三个子分类，id分别为1:10、1:20、1:30，这个10、20、30的编号就是针对上面default的参数，你想让默认数据流走哪个分类，就在default参数后面加上它相应的id即可。我们建立了分类并且给分类做了速度限制，并且使用ceil参数指定每个分类都可以在其它分类空闲的时候借用带宽资源最高可以达到10mbit。

之后是给每个分类下再添加相应的过滤器，我们这里分别给三个分类使用了不同的过滤器，以实现不同的Qos保障。当然，每个子分类下还可以继续添加htb过滤器，让整个htb的分层树形结构变的更大，分类更细。一般情况下，两层的结构足以应付绝大多数场景了。

```
[root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 parent 1:10 handle 10: fq_codel
[root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 parent 1:20 handle 20: sfq
[root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 parent 1:30 handle 30: pie
```

最后，我们使用u32过滤器，对数据包进行过滤，这两条命令分别将源端口为80的数据包放到分类1:10里，源端扣为25的数据包放到分类1:20里。默认其它数据包（包括22），根据default规则走分类1:30。

```
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1 u32 match ip dport 80 0xffff flowid 1:10
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1 u32 match ip dport 25 0xffff flowid 1:20
```

至此，htb以及u32过滤器的简单使用介绍完毕。

内核除了实现了u32过滤器来帮我们过滤数据包以外，还有一个常用的过滤器叫fw，就是实用防火墙标记作为数据包分类的区分方法（firewall mark）。我们可以先使用iptable的mangle表对数据包先做mark标记，然后在tc中使用fw过滤器去识别相应的数据包，并进行分类。还是用以上的例子进行说明，此时我们使用fw过滤器的话，最后两条命令将变成这样：

```
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1 handle 1 fw flowid 1:10
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1 handle 2 fw flowid 1:20
```

这两条命令说明，凡是被fwmark标记为1的数据包都走分类1:10，标记为2的走分类1:20。之后，别忘了在iptable里面添加对数据包的标记：

```
[root@zorrozou-pc0 zorro]# iptables -t mangle -A OUTPUT -p tcp --sport 80 -j MARK --set-mark 1
[root@zorrozou-pc0 zorro]# iptables -t mangle -A OUTPUT -p tcp --sport 25 -j MARK --set-mark 2
```

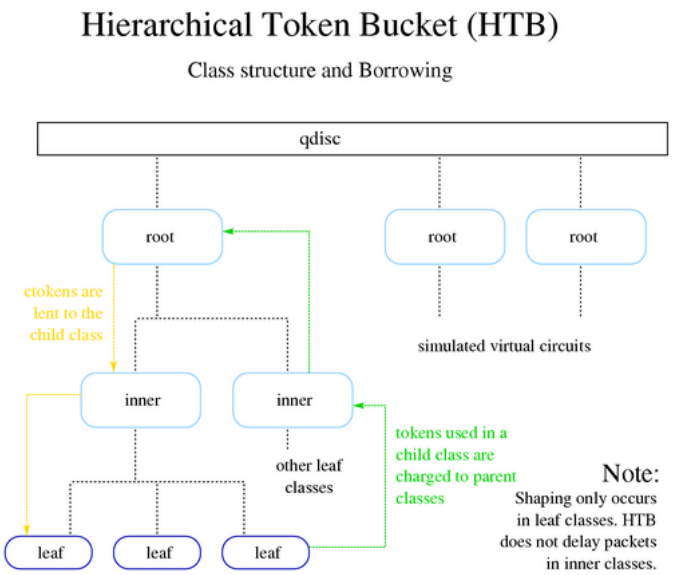
如果你不想学习u32过滤器哪些复杂的语法，那么fwmark是一种很好的替代方式。当然前提是你对iptables和tcp／ip协议有一定了解。

> *思考题：添加完iptables规则后，我们可以通过以下命令查看目前mangle表的内容：*
>
> ```
> [root@zorrozou-pc0 zorro]# iptables -t mangle -L
> Chain PREROUTING (policy ACCEPT)
> target     prot opt source               destination
> Chain INPUT (policy ACCEPT)
> target     prot opt source               destination
> Chain FORWARD (policy ACCEPT)
> target     prot opt source               destination
> Chain OUTPUT (policy ACCEPT)
> target     prot opt source               destination
> MARK       tcp  --  anywhere             anywhere             tcp spt:http MARK set 0x1
> MARK       tcp  --  anywhere             anywhere             tcp spt:smtp MARK set 0x2
> Chain POSTROUTING (policy ACCEPT)
> target     prot opt source               destination
> ```

在本例中，我们使用了其中的OUTPUT链添加了规则。那么问题是：使用不同的链（Chain）的区别是什么？

因为一些原因，我们推荐使用HTB的方式对比较复杂的网络数据包进行分类并流量整形。当然，可分类的队列规则中，除了HTB还有PRIO以及非常著名的CBQ。其中CBQ尤其在网络设备的限速方面有着最广泛的使用，但是如果从软件实现的角度来说，令牌桶方式(htb就是分层令牌桶)的流量限制在性能和稳定性上都更具有优势。PRIO由于分类过于简单，并不适合更复杂的场景。

关于这些知识的介绍，大家依然可以在LARTC上找到更详细的讲解。根据上面的命令，我们再参照结构图来理解一下HTB：



使用cgroup限制网络流量

如果你是从头开始看到这里的，那么真的很佩服你的耐心。我们前面似乎讲了一堆跟cgroup做网络资源隔离没有关系的知识，但是无疑每一个知识点的理解对于我们规划网络的资源隔离都有很重要的作用。毕竟，我么要规划一个架构，必需了解清楚其相关实现以及要解决的问题。但是很不幸，我们依然没有能够讲完目前所有的qdisc实现，比如还有HFSC、ATM、MULTIQ、TEQL、GRED、DSMARK、MQPRIO、QFQ、HHF等，这些还是留着给大家自己去解密吧。相信大家如果真正理解了队列规则要解决的问题和其基础知识，理解这些东西并不难。

最后，我们要来看看如何在cgroup的场景下对网络资源进行隔离了。实际上跟我们上面讲的HTB的例子类似，区别是，上面的例子是通过端口分类，而现在需要通过cgroup进行分类。我们还是通过一个例子来说明一下场景，并实现其功能：我们假定现在有两个cgroup，一个叫jerry，另一个叫zorro。我们现在需要给jerry组中运行的网络程序限制带宽为10mbit，zorro组的网路资源占用为20mbit，总带宽为100mbit，并且不允许借用（ceil）网络资源。那么配置思路是这样：

我们的配置环境是一台centos7的虚拟机，首先，我们在这个服务器上运行一个apache的http服务，并发布了一个1G的数据文件作为测试文件，并在不限速的情况下对齐进行下载速度测试，结果为100MBps，注意这里的速度是byte而不是bit：

```
zorrozou-nb:~ zorro$ curl -O http://192.168.139.136/file
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1024M  100 1024M    0     0   101M      0  0:00:10  0:00:10 --:--:--  100M
```

之后我们在centos7(192.168.139.136)上实现三个分类，一个带宽限制10m给jerry，另一个20m给zorro，还有一个为30m用作default，总带宽100m，剩余的资源给以后可能新加入的cgroup来分配，于是先建立相关的规则和分类：

```
[root@localhost Desktop]# tc qd add dev eno16777736 root handle 1: htb default 100
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1: classid 1:1 htb rate 100mbit burst 20k
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1:1 classid 1:10 htb rate 10mbit burst 20k
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1:1 classid 1:20 htb rate 20mbit burst 20k
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1:1 classid 1:100 htb rate 30mbit burst 20k

[root@localhost Desktop]# tc qd add dev eno16777736 parent 1:10 handle 10: fq_codel
[root@localhost Desktop]# tc qd add dev eno16777736 parent 1:20 handle 20: fq_codel
[root@localhost Desktop]# tc qd add dev eno16777736 parent 1:100 handle 100: fq_codel
```

建立完分类之后，由于默认情况都要走1:100的分类，所以限速应该是30mbit，验证一下：

```
zorrozou-nb:~ zorro$ curl -O http://192.168.139.136/file
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0 1024M    0 3484k    0     0  3452k      0  0:05:03  0:00:01  0:05:02 3452k
```

当前速度为3452kB左右，大概为30mbit，符合预期。之后将我们的http服务放到zorro组中看看效果，当然是首先建立相关cgroup以及相关配置：

```
[root@localhost Desktop]# ls /sys/fs/cgroup/net_cls/
cgroup.clone_children  cgroup.event_control  cgroup.procs  cgroup.sane_behavior  net_cls.classid  notify_on_release
release_agent  tasks
[root@localhost Desktop]# mkdir /sys/fs/cgroup/net_cls/zorro
[root@localhost Desktop]# mkdir /sys/fs/cgroup/net_cls/jerry
[root@localhost Desktop]# ls /sys/fs/cgroup/net_cls/{zorro,jerry}
/sys/fs/cgroup/net_cls/jerry:
cgroup.clone_children  cgroup.event_control  cgroup.procs  net_cls.classid  notify_on_release  tasks

/sys/fs/cgroup/net_cls/zorro:
cgroup.clone_children  cgroup.event_control  cgroup.procs  net_cls.classid  notify_on_release  tasks
```

建立完毕之后分别配置相关的cgroup，将对应cgroup产生的数据包对应到相应的分类中，配置方法：

```
[root@localhost Desktop]# echo 0x00010100 > /sys/fs/cgroup/net_cls/net_cls.classid
[root@localhost Desktop]# echo 0x00010010 > /sys/fs/cgroup/net_cls / jerry/net_cls.classid
[root@localhost Desktop]# echo 0x00010020 > /sys/fs/cgroup/net_cls / zorro/net_cls.classid
[root@localhost Desktop]# tc fi add dev eno16777736 parent 1: protocol ip prio 1 handle 1: cgroup
```

这里的tc命令是对filter进行操作，这里我们使用了cgroup过滤器，来实现将cgroup的数据包送到1:0分类中，细节不再解释。对于net_cls.classid文件，我们一般echo的是一个0xAAAABBBB的值，AAAA对应class中:前面的数字，而BBBB对应后面的数字，如：0x00010100就表示这个组的数据包将被分类到1:100中，限速为30mbit，以此类推。之后我们把http服务放倒jerry组中看看效果：

```
[root@localhost Desktop]# for i in `ps ax|grep httpd|awk '{ print $1}'`;do echo $i > /sys/fs/cgroup/net_cls/jerry/tasks;done
bash: echo: write error: No such process
[root@localhost Desktop]# cat /sys/fs/cgroup/net_cls/jerry/tasks
75733
75734
75735
75736
75737
75738
75777
75778
75779
```

测试效果：

```
zorrozou-nb:~ zorro$ curl -O http://192.168.139.136/file
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0 1024M    0 5118k    0     0  1162k      0  0:15:01  0:00:04  0:14:57 1162k
```

确实限速在了10mbitps。成功达到效果，再来看看放倒zorro组下：

```
[root@localhost Desktop]# for i in `ps ax|grep httpd|awk '{ print $1}'`;do echo $i > /sys/fs/cgroup/net_cls/zorro/tasks;done
bash: echo: write error: No such process
[root@localhost Desktop]# cat /sys/fs/cgroup/net_cls/zorro/tasks
75733
75734
75735
75736
75737
75738
75777
75778
75779
```

再次测试效果：

```
zorrozou-nb:~ zorro$ curl -O http://192.168.139.136/file
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0 1024M    0 5586k    0     0  2334k      0  0:07:29  0:00:02  0:07:27 2334k
```

限速20mbps成功。如果想要修改对于一个分类的限速，使用如下命令即可：

```
tc cl change dev eno16777736 parent 1: classid 1:100 htb rate 100mbit
```

关于命令参数的详细解释，这里不做过多说明了。大家可以自行找帮助。

最后

终于，我的Cgroup系列四部曲算是告一段落了。实际上Linux的Cgroup除了CPU、内存、IO和网络的资源管理以外，还有一些其它的配置，比如针对设备文件的访问控制和freezer机制等功能，但是这些功能都相对比较简单，个人认为没必要过多介绍了，大家要用的时候自己找帮助即可。 最后的最后，还是奉送一张Linux网络相关的数据包处理流程图，从这张图上大家可以清晰的看到qdisc的作用位置和其根iptables的作用关系。[原图链接在此]。