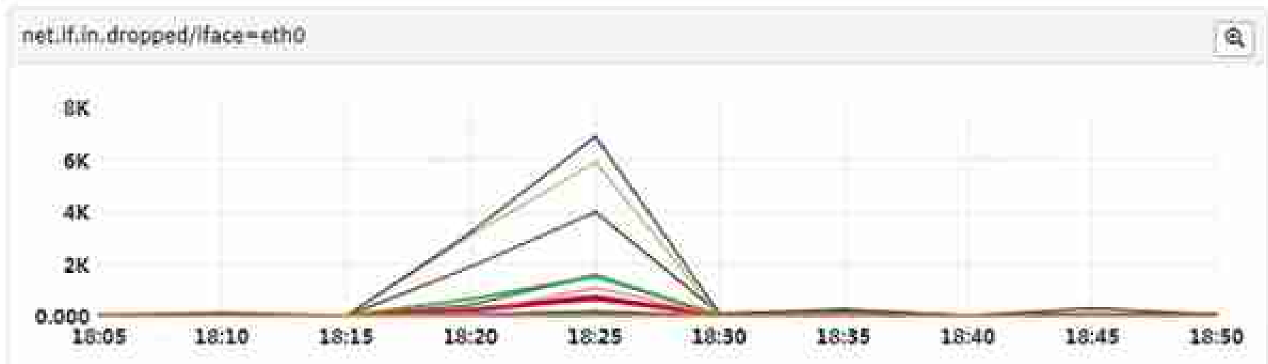


这个指标的数据源，是读取/proc/net/dev中的数据，监控Agent做简单的处理之后上报。以下为/proc/net/dev 的一个示例，可以看到第一行Receive代表in，Transmit代表out，第二行即各个表头字段，再往后每一行代表一个网卡设备具体的值。

字段	解释
bytes	The total number of bytes of data transmitted or received by the interface.
packets	The total number of packets of data transmitted or received by the interface.
errs	The total number of transmit or receive errors detected by the device driver.
drop	The total number of packets dropped by the device driver.
fifo	The number of FIFO buffer errors.
frame	The number of packet framing errors.
colls	The number of collisions detected on the interface.
compressed	The number of compressed packets transmitted or received by the device driver. (This appears to be unused in the 2.2.15 kernel.)
carrier	The number of carrier losses detected by the device driver.
multicast	The number of multicast frames transmitted or received by the device driver.

/proc/net/dev的数据来源，根据源码文件net/core/net-procfs.c，可以知道上述指标是通过其中的dev_seq_show()函数和dev_seq_printf_stats()函数输出的：通过上述字段解释，我们可以了解丢包发生在网卡设备驱动层面；但是想要了解真正的原因，需要继续深入源码。

```
static int dev_seq_show(struct seq_file *seq, void *v){ if (
v == SEQ_START_TOKEN) /* ??/proc/net/dev???? */ seq_puts(seq
, "Inter-| Receive " " | Transmit\n" " face |bytes packets e
rrs drop fifo frame " "compressed multicast|bytes packets er
rs " "drop fifo colls carrier compressed\n"); else /* ??/pro
c/net/dev???? */ dev_seq_printf_stats(seq, v); return 0;}sta
tic void dev_seq_printf_stats(struct seq_file *seq, struct n
et_device *dev){ struct rtnl_link_stats64 temp; /* ?????????
??? */ const struct rtnl_link_stats64 *stats = dev_get_stats
(dev, &temp); /* /proc/net/dev ????????? */ seq_printf(seq,
"%6s: %7llu %7llu %4llu %4llu %4llu %5llu llu %9llu " "%8llu
 %7llu %4llu %4llu %4llu %5llu %7llu llu\n", dev->name, stat
s->rx_bytes, stats->rx_packets, stats->rx_errors, stats->rx_
dropped + stats->rx_missed_errors, stats->rx_fifo_errors, st
ats->rx_length_errors + stats->rx_over_errors + stats->rx_cr
c_errors + stats->rx_frame_errors, stats->rx_compressed, sta
ts->multicast, stats->tx_bytes, stats->tx_packets, stats->tx
_errors, stats->tx_dropped, stats->tx_fifo_errors, stats->co
llisions, stats->tx_carrier_errors + stats->tx_aborted_error
s + stats->tx_window_errors + stats->tx_heartbeat_errors, st
ats->tx_compressed);}
```

dev_seq_printf_stats()函数里，对应drop输出的部分，能看到由两块组成：stats->rx_dropped+stats->rx_missed_errors。

继续查找dev_get_stats函数可知，rx_dropped和rx_missed_errors
都是从设备获取的，并且需要设备驱动实现。

```
/*** dev_get_stats - get network device statistics* @dev: de
vice to get statistics from* @storage: place to store stats*
* Get network statistics from device. Return @storage.* The
device driver may provide its own method by setting* dev->ne
tdev_ops->get_stats64 or dev->netdev_ops->get_stats;* otherw
ise the internal statistics structure is used.*/struct rtnl_
```

```
link_stats64 *dev_get_stats(struct net_device *dev, struct r
tnl_link_stats64 *storage){ const struct net_device_ops *ops
 = dev->netdev_ops; if (ops->ndo_get_stats64) { memset(stora
ge, 0, sizeof(*storage)); ops->ndo_get_stats64(dev, storage)
; } else if (ops->ndo_get_stats) { netdev_stats_to_stats64(s
torage, ops->ndo_get_stats(dev)); } else { netdev_stats_to_s
tats64(storage, &dev->stats); } storage->rx_dropped += (unsi
gned long)atomic_long_read(&dev->rx_dropped); storage->tx_dr
opped += (unsigned long)atomic_long_read(&dev->tx_dropped);
storage->rx_nohandler += (unsigned long)atomic_long_read(&de
v->rx_nohandler); return storage;}
```

结构体 rtnl_link_stats64 的定义在 /usr/include/linux/if_link.h 中：

```
/* The main device statistics structure */struct rtnl_link_s
tats64 { __u64 rx_packets; /* total packets received */ __u6
4 tx_packets; /* total packets transmitted */ __u64 rx_bytes
; /* total bytes received */ __u64 tx_bytes; /* total bytes
transmitted */ __u64 rx_errors; /* bad packets received */ _
_u64 tx_errors; /* packet transmit problems */ __u64 rx_drop
ped; /* no space in linux buffers */ __u64 tx_dropped; /* no
 space available in linux */ __u64 multicast; /* multicast p
ackets received */ __u64 collisions; /* detailed rx_errors:
*/ __u64 rx_length_errors; __u64 rx_over_errors; /* receiver
 ring buff overflow */ __u64 rx_crc_errors; /* recved pkt wi
th crc error */ __u64 rx_frame_errors; /* recv'd frame align
ment error */ __u64 rx_fifo_errors; /* recv'r fifo overrun *
/ __u64 rx_missed_errors; /* receiver missed packet */ /* de
tailed tx_errors */ __u64 tx_aborted_errors; __u64 tx_carrie
r_errors; __u64 tx_fifo_errors; __u64 tx_heartbeat_errors; _
_u64 tx_window_errors; /* for cslip etc */ __u64 rx_compress
ed; __u64 tx_compressed;};
```

至此，我们知道rx_dropped是Linux中的缓冲区空间不足导致的丢包，而rx_missed_errors则在注释中写的比较笼统。有资料指出，rx_missed_errors是fifo队列（即rx ring buffer）满而丢弃的数量，但这样的话也就和rx_fifo_errors等同了。后来公司内网络内核研发大牛王伟给了我们点拨：不同网卡自己实现不一样，比如Intel的igb网卡rx_fifo_errors在missed的基础上，还加上了RQDPC计数，而ixgbe就没这个统计。RQDPC计数是描述符不够的计数，missed是fifo满的计数。所以对于ix

gbe来说，rx_fifo_errors和rx_missed_errors确实是等同的。

通过命令ethtool -S eth0可以查看网卡一些统计信息，其中就包含了上文提到的几个重要指标rx_dropped、rx_missed_errors、rx_fifo_errors等。但实际测试后，我发现不同网卡型号给出的指标略有不同，比如Intel
ixgbe就能取到，而Broadcom bnx2/tg3则只能取到rx_discards（对应rx_fifo_errors）、rx_fw_discards（对应rx_dropped）。这表明，各家网卡厂商设备内部对这些丢包的计数器、指标的定义略有不同，但通过驱动向内核提供的统计数据都封装成了struct rtnl_link_stats64定义的格式。

在对丢包服务器进行检查后，发现rx_missed_errors为0，丢包全部来自rx_dropped。说明丢包发生在Linux内核的缓冲区中。接下来，我们要继续探索到底是什么缓冲区引起了丢包问题，这就需要完整地了解服务器接收数据包的过程。

了解接收数据包的流程

接收数据包是一个复杂的过程，涉及很多底层的技术细节，但大致需要以下几个步骤：

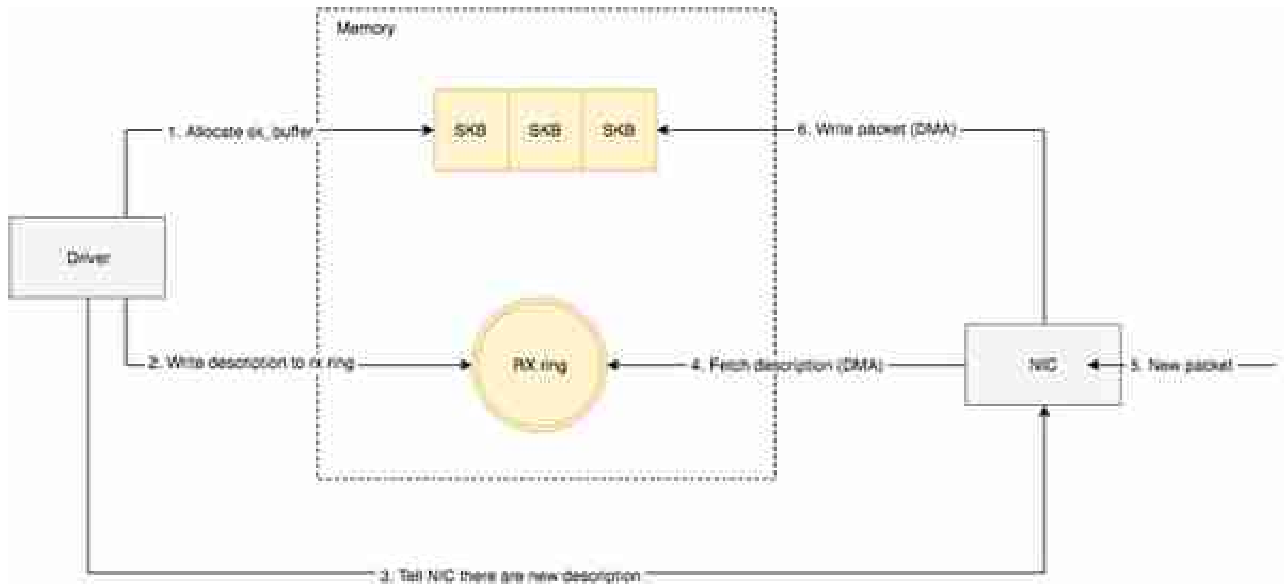
1. 网卡收到数据包。

2. 将数据包从网卡硬件缓存转移到服务器内存中。

3. 通知内核处理。

4. 经过TCP/IP协议逐层处理。

5. 应用程序通过read()从socket buffer读取数据。



当驱动处理速度跟不上网卡收包速度时，驱动来不及分配缓冲区，NIC接收到的数据包无法及时写到sk_buffer，就会产生堆积，当NIC内部缓冲区写满后，就会丢弃部分数据，引起丢包。这部分丢包为rx_fifo_errors，在/proc/net/dev中体现为fifo字段增长，在ifconfig中体现为overruns指标增长。

通知系统内核处理（驱动与Linux内核交互）

这个时候，数据包已经被转移到了sk_buffer中。前文提到，这是驱动程序在内存中分配的一片缓冲区，并且是通过DMA写入的，这种方式不依赖CPU直接将数据写到了内存中，意味着对内核来说，其实并不知道已经有新数据到了内存中。那么如何让内核知道有新数据进来了呢？答案就是中断，通过中断告诉内核有新数据进来了，并需要进行后续处理。

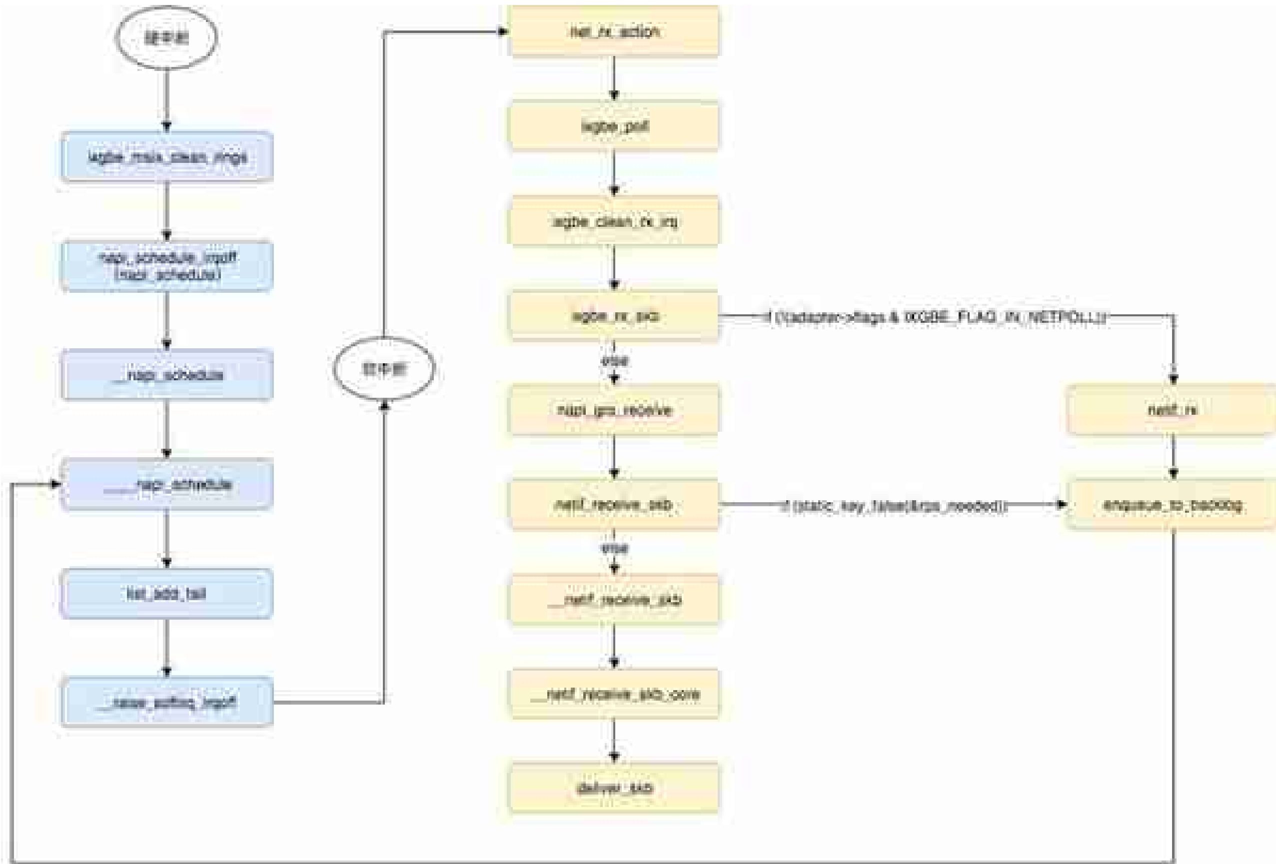
提到中断，就涉及到硬中断和软中断，首先需要简单了解一下它们的区别：

- 硬中断： 由硬件自己生成，具有随机性，硬中断被CPU接收后，触发执行中断处理程序。中断处理程序只会处理关键性的、短时间内可以处理完的工作，剩余耗时较长工作，会放到中断之后，由软中断来完成。硬中断也被称为上半部分。

- 软中断： 由硬中断对应的中断处理程序生成，往往是预先在代码里实现好的，不具有随机性。（除此之外，也有应用程序触发的软中断，与本文讨论的网卡收包无关。）也被称为下半部分。

当NIC把数据包通过DMA复制到内核缓冲区sk_buffer后，NIC立即发起一个硬件中

断。CPU接收后，首先进入上半部分，网卡中断对应的中断处理程序是网卡驱动程序的一部分，之后由它发起软中断，进入下半部分，开始消费sk_buffer中的数据，交给内核协议栈处理。



注意到，enqueue_to_backlog函数中，会对CPU的softnet_data 实例中的接收队列（input_pkt_queue）进行判断，如果队列中的数据长度超过netdev_max_backlog ，那么数据包将直接丢弃，这就产生了丢包。netdev_max_backlog是由系统参数net.core.netdev_max_backlog指定的，默认大小是 1000。

```
 /** enqueue_to_backlog is called to queue an skb to a per CPU backlog* queue (may be a remote CPU queue).*/static int enqueue_to_backlog(struct sk_buff *skb, int cpu, unsigned int *qtail){ struct softnet_data *sd; unsigned long flags; sd = &per_cpu(softnet_data, cpu); local_irq_save(flags); rps_lock(sd); /* ??????????????? netdev_max_backlog */ if (skb_queue_len(&sd->input_pkt_queue) <= netdev_max_backlog) { if (skb_queue_len(&sd->input_pkt_queue)) {enqueue: /* ?????????????????? */ __skb_queue_tail(&sd->input_pkt_queue, skb); input_queue_tail_incr_save(sd, qtail); rps_unlock(sd); local_irq_r
```

estore(flags); return NET_RX_SUCCESS; } /* Schedule NAPI for backlog device * We can use non atomic operation since we own the queue lock */ /* ????????? ____napi_schedule??poll_list????????????? */ if (!__test_and_set_bit(NAPI_STATE_SCHED, &sd->backlog.state)) { if (!rps_ipi_queued(sd)) ____napi_schedule(sd, &sd->backlog); } goto enqueue; } /* ?????????????? +1 */ sd->dropped++; rps_unlock(sd); local_irq_restore(flags); atomic_long_inc(&skb->dev->rx_dropped); kfree_skb(skb); return NET_RX_DROP;}

内核会为每个CPU Core都实例化一个softnet_data对象，这个对象中的input_pkt_queue用于管理接收的数据包。假如所有的中断都由一个CPU Core来处理的话，那么所有数据包只能经由这个CPU的input_pkt_queue，如果接收的数据包数量非常大，超过中断处理速度，那么input_pkt_queue中的数据包就会堆积，直至超过netdev_max_backlog，引起丢包。这部分丢包可以在cat /proc/net/softnet_stat的输出结果中进行确认：

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7		
135:	1	0	0	0	0	0	0	0	IR-PCI-MSI-edge	eth0
136:	527938126	0	315864811	0	331494359	0	166362734	0	IR-PCI-MSI-edge	eth0-TxRx-0
137:	3920620869	636228352	624047238	0	306026265	0	100392484	0	IR-PCI-MSI-edge	eth0-TxRx-1
138:	1308525100	0	578643538	0	580156872	0	1618476592	0	IR-PCI-MSI-edge	eth0-TxRx-2
139:	3158683235	0	356503336	578376147	319069355	0	506989713	0	IR-PCI-MSI-edge	eth0-TxRx-3
140:	4133597158	0	327907259	0	717001407	0	179516622	0	IR-PCI-MSI-edge	eth0-TxRx-4
141:	4221862708	0	325846602	0	338278055	409075354	177630767	0	IR-PCI-MSI-edge	eth0-TxRx-5
142:	327432506	0	138961	0	0	0	2748897340	0	IR-PCI-MSI-edge	eth0-TxRx-6
143:	1657413323	0	328991638	0	338213329	0	204525401	492307994	IR-PCI-MSI-edge	eth0-TxRx-7
144:	1	0	0	0	0	0	0	0	IR-PCI-MSI-edge	eth1
145:	718868368	0	306939492	0	337766750	0	183415711	0	IR-PCI-MSI-edge	eth1-TxRx-0
146:	3570866221	0	681075539	0	339390926	0	185842906	0 3	IR-PCI-MSI-edge	eth1-TxRx-1
147:	2258777018	0	341129658	0	718866281	0	2252739909	0	IR-PCI-MSI-edge	eth1-TxRx-2
148:	3026116830	0	341811416	0	1379935197	0	532457615	0	IR-PCI-MSI-edge	eth1-TxRx-3
149:	428831714	0	337604357	0	312685854	0	226823171	0	IR-PCI-MSI-edge	eth1-TxRx-4
150:	1754126180	0	325384817	0	330443034	0	192058999	0	IR-PCI-MSI-edge	eth1-TxRx-5
151:	249233769	0	0	0	81195	0	0	0	IR-PCI-MSI-edge	eth1-TxRx-6
152:	638608879	0	121078355	0	356335776	0	192316458	0	IR-PCI-MSI-edge	eth1-TxRx-7

中断的亲缘性设置可以在 cat /proc/irq/${中断号}/smp_affinity 或 cat /proc/irq/${中断号}/smp_affinity_list
中确认，前者是16进制掩码形式，后者是以CPU Core序号形式。例如下图中，将16进制的400转换成2进制后，为10000000000，“1”在第10位上，表示亲缘性是第10个CPU Core。

Redis进程亲缘性设置

如果某个CPU Core正在处理Redis的调用，执行到一半时产生了中断，那么CPU不得不停止当前的工作转而处理中断请求，中断期间Redis也无法转交给其他core继续运行，必须等处理完中断后才能继续运行。Redis本身定位就是高速缓存，线上的平均端到端响应时间小于1ms，如果频繁被中断，那么响应时间必然受到极大影响。容易想到，由最初的CPU 0单核处理中断，改进到多核处理中断，Redis进程被中断影响的几率增大了，因此我们需要对Redis进程也设置CPU亲缘性，使其与处理中断的Core互相错开，避免受到影响。

使用命令taskset可以为进程设置CPU亲缘性，操作十分简单，一句taskset -cp cpu-list pid即可完成绑定。经过一番压测，我们发现使用8个core处理中断时，流量直至打满双万兆网卡也不会出现丢包，因此决定将中断的亲缘性设置为物理机上前8个core，Redis进程的亲缘性设置为剩下的所有core。调整后，确实有明显的效果，慢查询数量大幅优化，但对比初始情况，仍然还是高了一些些，还有没有优化空间呢？

```
PID    COMMAND                      PSR
4972   /usr/local/bin/redis-server 0
8897   /usr/local/bin/redis-server 14
46790  /usr/local/bin/redis-server 0
61575  /usr/local/bin/redis-server 0
86927  /usr/local/bin/redis-server 10
90249  /usr/local/bin/redis-server 8
94854  /usr/local/bin/redis-server 18
98208  /usr/local/bin/redis-server 10
102625 /usr/local/bin/redis-server 18
116323 /usr/local/bin/redis-server 12
124845 /usr/local/bin/redis-server 16
135070 /usr/local/bin/redis-server 0
143051 /usr/local/bin/redis-server 10
155179 /usr/local/bin/redis-server 12
168789 /usr/local/bin/redis-server 6
182278 /usr/local/bin/redis-server 4
193082 /usr/local/bin/redis-server 18
194616 /usr/local/bin/redis-server 10
```

NUMA

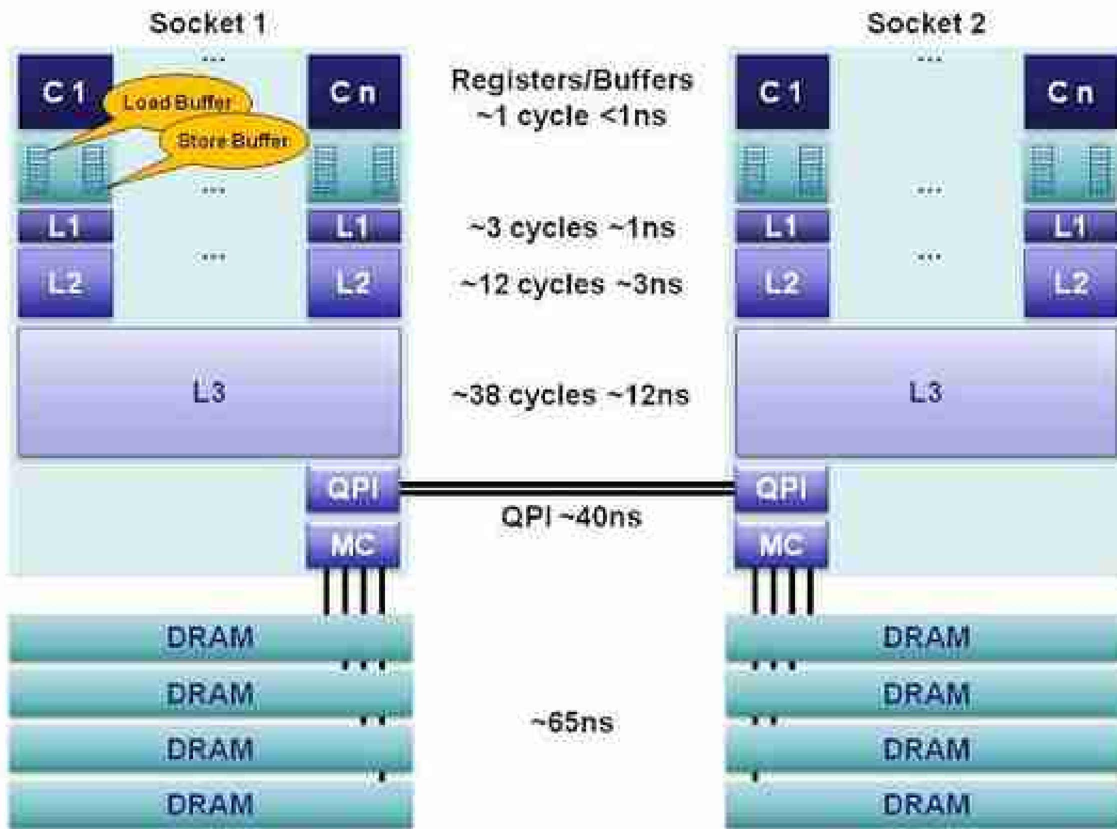
SMP 架构

随着单核CPU的频率在制造工艺上的瓶颈，CPU制造商的发展方向也由纵向变为横向：从CPU频率转为每瓦性能。CPU也就从单核频率时代过渡到多核性能协调。

SMP(对称多处理结构)：即CPU共享所有资源，例如总线、内存、IO等。

SMP 结构：一个物理CPU可以有多个物理Core，每个Core又可以有多个硬件线程。即：每个HT有一个独立的L1 cache，同一个Core下的HT共享L2 cache，同一个物理CPU下的多个core共享L3 cache。

下图(摘自内核月谈)中，一个x86 CPU有4个物理Core，每个Core有两个HT(Hyper Thread)。



NUMA 架构下的中断优化

这时我们再回归到中断的问题上，当两个NUMA节点处理中断时，CPU实例化的softnet_data以及驱动分配的sk_buffer都可能是跨node的，数据接收后对上层应用Redis来说，跨node访问的几率也大大提高，并且无法充分利用L2、L3 cache，增加了延时。

同时，由于Linux wake affinity 特性，如果两个进程频繁互动，调度系统会觉得它们很有可能共享同样的数据，把它们放到同一CPU核心或NUMA Node有助于提高缓存和内存的访问性能，所以当一个进程唤醒另一个的时候，被唤醒的进程可能会被放到相同的CPU core或者相同的NUMA节点上。此特性对中断唤醒进程时也起作用，在上一节所述的现象中，所有的网络中断都分配给CPU 0去处理，当中断处理完成时，由于wakeup affinity特性的作用，所唤醒的用户进程也被安排给CPU 0或其所在的numa节点上其他core。而当两个NUMA node处理中断时，这种调度特性有可能导致Redis进程在CPU core之间频繁迁移，造成性能损失。

综合上述，将中断都分配在同一NUMA Node中，中断处理函数和应用程序充分利

用同NUMA下的L2、L3缓存、以及同node下的内存，结合调度系统的wake affinity特性，能够更进一步降低延迟。