

前言

随着携程的应用大规模在生产上用容器部署，各种上规模的问题都慢慢浮现，其中比较难定位和解决的就是偶发性超时问题，下面将分析目前为止我们遇到的几种偶发性超时问题以及排查定位过程和解决方法，希望能给遇到同样问题的小伙伴们以启发。

问题描述

某一天接到用户报障说，Redis集群有超时现象发生，比较频繁，而访问的QPS也比较低。紧接着，陆续有其他用户也报障Redis访问超时。在这些报障容器所在的宿主机里面，我们猛然发现有之前因为时钟漂移问题升级过内核到4.14.26的宿主机ServerA，心里突然有一丝不详的预感。

初步分析

因为现代软件部署结构的复杂性以及网络的不可靠性，我们很难快速定位“connect timeout”或“connectreset by peer”之类问题的根因。

历史经验告诉我们，一般比较大范围的超时问题要么和交换机路由器之类的网络设备有关，要么就是底层系统不稳定导致的故障。从报障的情况来看，4.10和4.14的内核都有，而宿主机的机型也涉及到多个批次不同厂家，看上去没头绪，既然没什么好办法，那就抓个包看看吧。

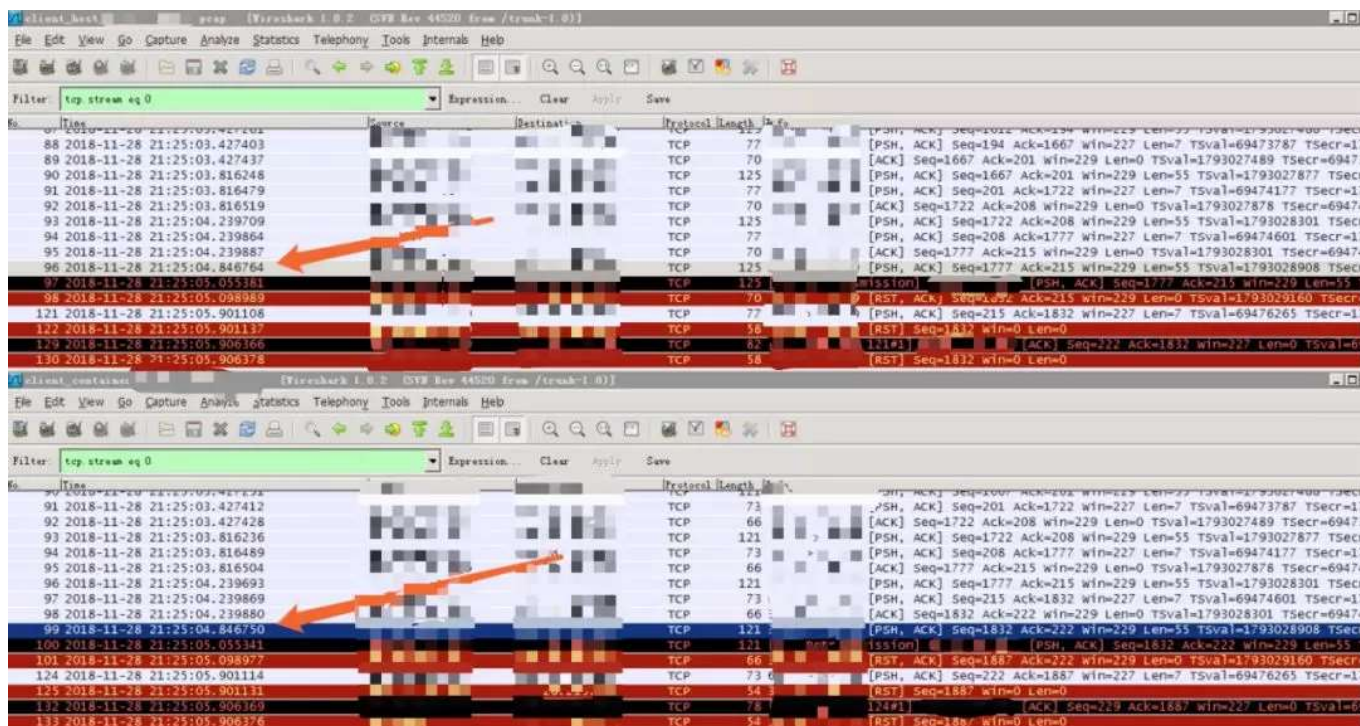


图1

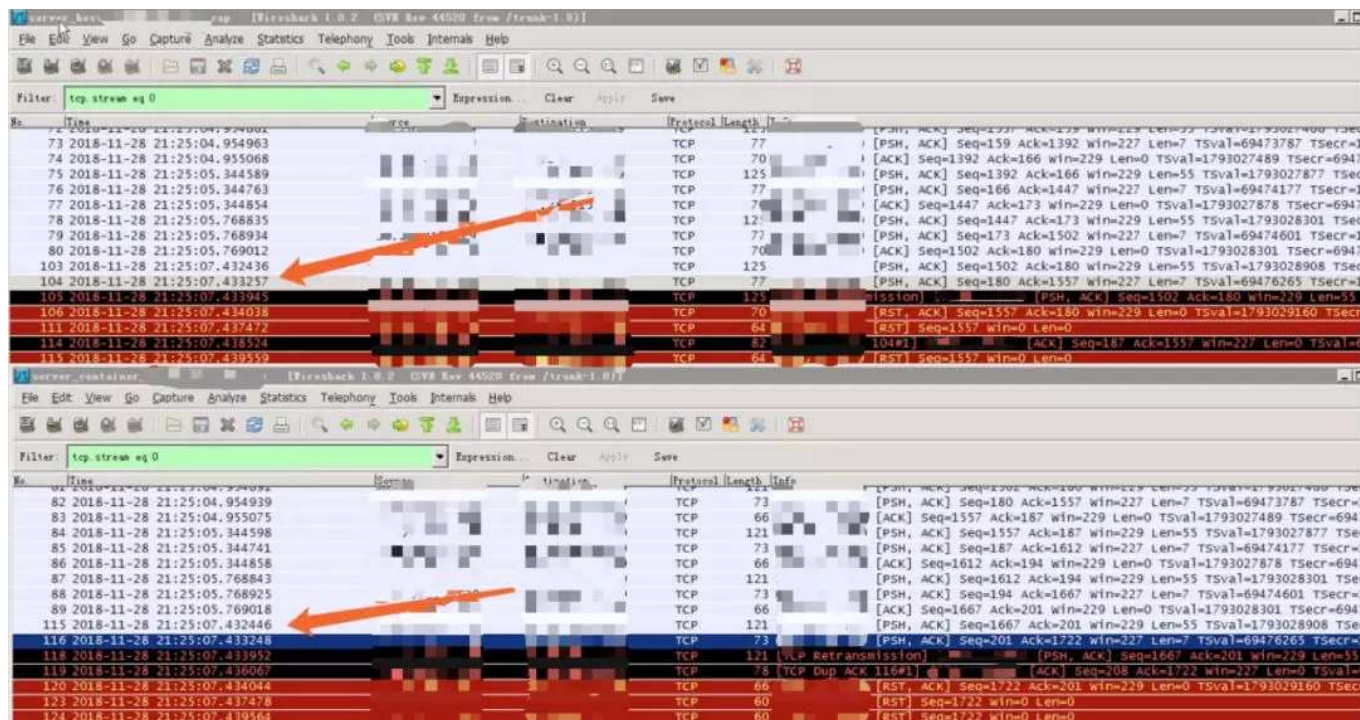


图2

图1是App端容器和宿主机的抓包，图2是Redis端容器和宿主机的抓包。因为APP和Redis都部署在容器里面（图3），所以一个完整请求的包是B->A->C->D，而Redis的回包是D->C->A->B。

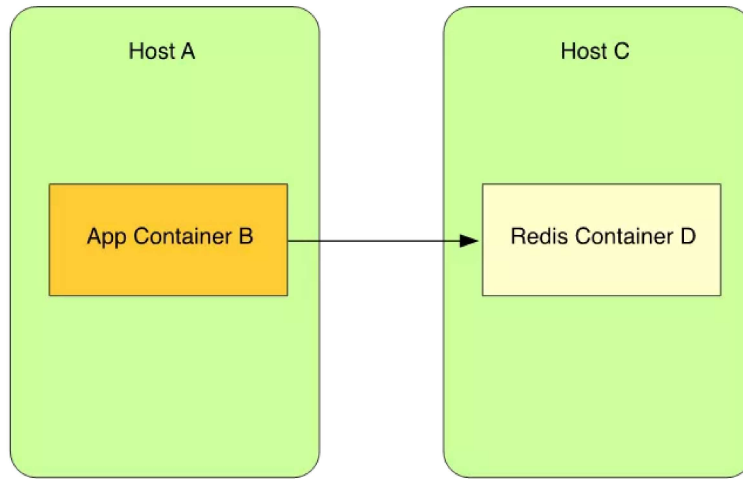


图3

上面抓包的某一条请求如下：

- 1) B(图1第二个)的请求时间是21:25:04.846750 #99
- 2) 到达A(图1第一个)的时间是21:25:04.846764 #96
- 3) 到达C(图2第一个)的时间是21:25:07.432436 #103
- 4) 到达D(图2第二个)的时间是21:25:07.432446 #115

该请求从D回复如下：

- 1) D的回复时间是21:25:07.433248 #116
- 2) 到达C的时间是21:25:07.433257 #104
- 3) 到达A点时间是21:25:05:901108 #121
- 4) 到达B的时间是21:25:05:901114 #124

从这一条请求的访问链路我们可以发现，B在200ms超时后等不到回包。在21:25:05.055341重传了该包#100，并且可以从C收到重传包的时间#105看出，几乎与#103的请求包同时到达，也就是说该第一次的请求包在网络上被延迟传输了。大致的示意如下图4所示：

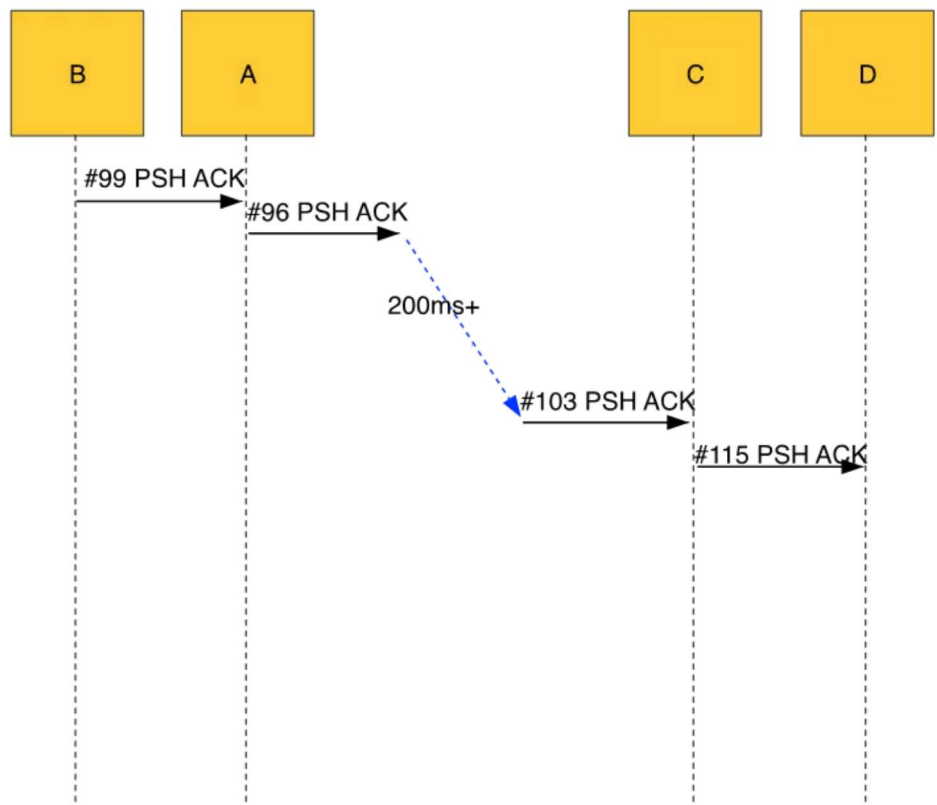


图4

从抓包分析来看，宿主机上好像并没有什么问题，故障在网络上。而我们同时在两边宿主机，容器里以及连接宿主机的交换机抓包，就变成了下面图5所示，假设连接A的交换机为E，也就是说A->E这段的网络有问题。

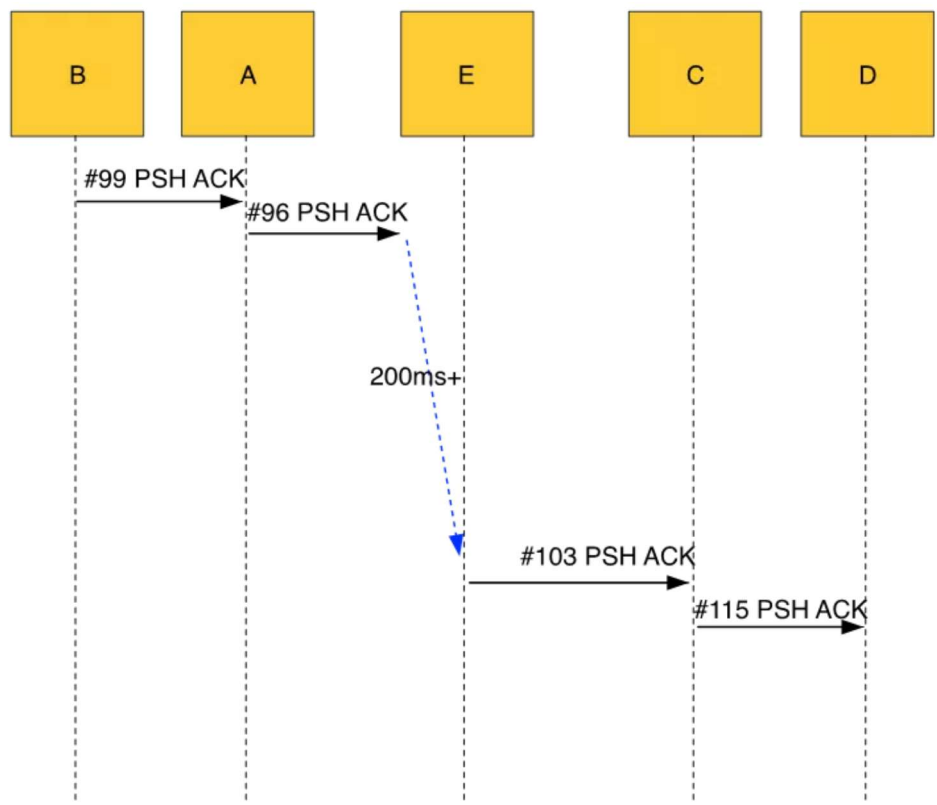


图5

陷入僵局

尽管发现A->E这段有问题，排查却也就此陷入了僵局，因为影响的宿主机遍布多个IDC，这也让我们排除了网络设备的问题。我们怀疑是否跟宿主机的某些TCP参数有关，比如TSO/GSO，一番测试后发现开启关闭TSO/GSO和修改内核参数对解决问题无效，但同时我们也观察到，从相同IDC里任选一台宿主机Ping有问题的宿主机，百分之几的概率看到很高的响应值，如下图6所示：

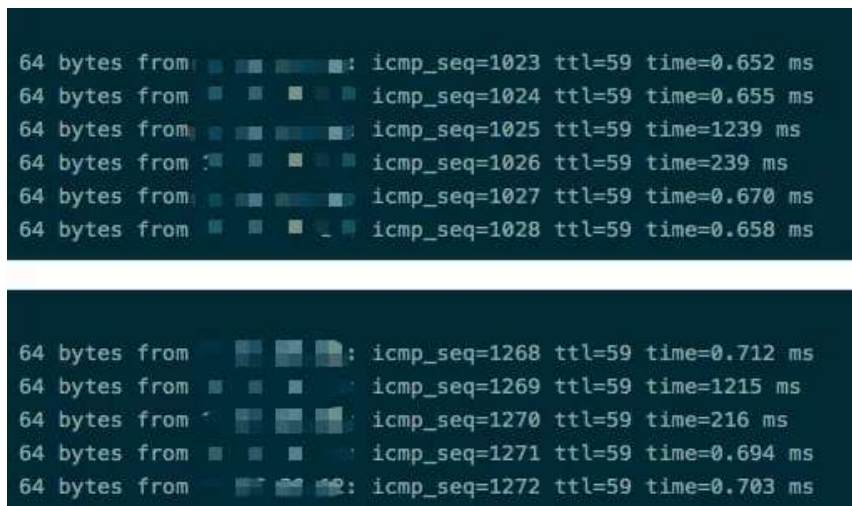


图6

同一个IDC内如此高的Ping响应延迟，很不正常。而这时DBA告诉我们，他们的某台物理机ServerB也有类似的问题，Ping延迟很大，SSH上去后明显感觉到有卡顿，这无疑给我们解决问题带来了希望，但又更加迷惑：

- 1) 延迟好像跟内核版本没有关系，3.10，4.10，4.14的三个版本内核看上去都有这种问题。
- 2) 延迟和容器无关，因为延迟都在宿主机上到连接宿主机的交换机上发现的。
- 3) ServerB跟ServerA虽然表现一样，但细节上看有区别，我们的宿主机在重启后基本上都能恢复一段时间后再复现延迟，但ServerB重启也无效。

由此我们判断ServerA和ServerB的症状并不是同一个问题，并让ServerB先升级固件看看。在升级固件后ServerB恢复了正常，那么我们的宿主机是不是也可以靠升级固件修复呢？答案是并没有。升级固件后没过几天，延迟的问题又出现了。

意外发现

回过头来看之前为了排查Skylake时钟漂移问题的ServerA，上面一直有个简单的程序在运行，来统计时间漂移的值，将时间差记到文件中。当时这个程序是为了验证时钟漂移问

题是否修复，如图7：

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>

struct timespec diff(struct timespec start, struct timespec end)
{
    struct timespec temp;
    if ((end.tv_nsec-start.tv_nsec)<0) {
        temp.tv_sec = end.tv_sec-start.tv_sec-1;
        temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
    } else {
        temp.tv_sec = end.tv_sec-start.tv_sec;
        temp.tv_nsec = end.tv_nsec-start.tv_nsec;
    }
    return temp;
}

int main()
{
    struct timespec time1, time2;
    int temp;
    FILE *ff = fopen("logout_clock", "wt");
    while (1){
        clock_gettime(CLOCK_MONOTONIC_RAW, &time1);
        usleep(0);
        clock_gettime(CLOCK_MONOTONIC_RAW, &time2);
        usleep(1000000);
        fprintf(ff, "time1:%lld,time2:%lld,seconds:%d,nano seconds:%ld\n", time1.tv_sec, time2.tv_sec, diff(time1, time2).tv_sec, diff(time1, time2).tv_nsec);
        fflush(ff);
    }
    return 0;
}
```

图7

这个程序跑在宿主机上，每个机器各有差异，但正常的时间差应该是100us以内，但1个多月后，时间差异越来越大，如图8，最大能达到几百毫秒以上。这告诉我们可能通过这无意中的log来找到根因，而且验证了上面3的这个猜想，宿主机是运行一段时间后逐渐出问题，表现为第一次打点到第二次打点之间，调度会自动delay第二次打点。

```
time1:8239869,time2:8239869,seconds:0,nano seconds:62127
time1:8239870,time2:8239870,seconds:0,nano seconds:55051
time1:8239870,time2:8239870,seconds:0,nano seconds:55734
time1:8239870,time2:8239870,seconds:0,nano seconds:55623
time1:8239870,time2:8239870,seconds:0,nano seconds:54874
time1:8239870,time2:8239870,seconds:0,nano seconds:54834
time1:8239870,time2:8239870,seconds:0,nano seconds:56361
time1:8239870,time2:8239870,seconds:0,nano seconds:3583498
time1:8239870,time2:8239870,seconds:0,nano seconds:54789
time1:8239870,time2:8239870,seconds:0,nano seconds:55415
time1:8239870,time2:8239870,seconds:0,nano seconds:55526
time1:8239871,time2:8239871,seconds:0,nano seconds:54904
time1:8239871,time2:8239871,seconds:0,nano seconds:55798
time1:8239871,time2:8239871,seconds:0,nano seconds:54906
time1:8239871,time2:8239871,seconds:0,nano seconds:55301
time1:8239871,time2:8239871,seconds:0,nano seconds:54596
time1:8239871,time2:8239871,seconds:0,nano seconds:54955
```

图8

Turbostat是intel开发的，用来查看CPU状态以及睿频的工具，同样可以用来查看TSC的频率。而关于TSC，之前的文章《携程一次Redis迁移容器后Slowlog“异常”分析》中有过相关介绍，这里就不再展开。

在有问题的宿主机上，TSC并不是恒定的，如图9所示，这个跟相关资料有出入，当然我们分析更可能的原因是，turbostat两次去取TSC的时候，被内核调度delay了，如果第一次取时被delay导致取的结果比实际TSC的值要小，而如果第二次取时被delay会导致取的结果比实际值要大。

Avg_MHz	Busy%	Bzy_MHz	TSC_MHz	IRQ	S
168	10.07	1539	1844	217282	0
253	12.78	1533	2196	29565	0
266	13.32	1543	2196	34804	0
270	16.41	1540	1815	33539	0
245	14.77	1553	1815	34009	0
208	13.04	1495	1815	34055	0
216	13.51	1502	1812	30473	0
72	4.50	1508	1812	3163	0
63	3.75	1570	1812	2318	0
140	8.58	1572	1764	4632	0
169	10.74	1581	1696	5712	0
58	3.65	1592	1696	2263	0
56	3.64	1540	1696	2749	0

图9

Perf 是内置于Linux上的基于采样的性能分析工具，一般随着内核一起编译出来，具体的用法可以搜索相关资料，这里也不展开。用perf sched record -a sleep 60和perf sched latency -s max来查看linux的调度延迟，发现最大能录得超过1s的延迟，如图10和图11所示。用户态的进程有时因为CPU隔离和代码问题导致比较大的延迟还好理解，但这些进程都是内核态的。尽管linux的CFS调度并非实时的调度，但在负载很低的情况下超过1s的调度延迟也是匪夷所思的。

Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
migration/37:233	0.000 ms	10	avg: 109.346 ms	max: 1092.834 ms	max at: 3492961.4
ksoftirqd/37:234	3.218 ms	107	avg: 9.974 ms	max: 1056.658 ms	max at: 3492961.4
kworker/21:1:64709	0.368 ms	35	avg: 30.132 ms	max: 1053.845 ms	max at: 3492966.4
ksoftirqd/21:138	0.538 ms	16	avg: 65.884 ms	max: 1053.031 ms	max at: 3492966.4
migration/21:137	0.000 ms	106	avg: 9.900 ms	max: 1049.017 ms	max at: 3492966.4
kworker/20:1:87977	0.449 ms	38	avg: 13.578 ms	max: 515.803 ms	max at: 3492961.4
migration/12:83	0.000 ms	2	avg: 75.992 ms	max: 151.982 ms	max at: 3492962.2
ksoftirqd/12:84	0.962 ms	64	avg: 2.249 ms	max: 142.684 ms	max at: 3492962.2
migration/17:113	0.000 ms	4	avg: 26.521 ms	max: 106.079 ms	max at: 3492962.4
ksoftirqd/17:114	0.939 ms	72	avg: 1.568 ms	max: 105.647 ms	max at: 3492962.4

图10

Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
kworker/8:2:22841	3.841 ms	185	avg: 7.247 ms	max: 1332.951 ms	max at: 9595693.23
ksoftirqd/8:60	2.430 ms	173	avg: 7.654 ms	max: 1322.959 ms	max at: 9595693.23
kworker/11:2:1028	0.749 ms	45	avg: 45.537 ms	max: 1284.833 ms	max at: 9595708.51
migration/8:59	0.000 ms	9	avg: 134.107 ms	max: 1206.944 ms	max at: 9595693.23
migration/11:77	0.000 ms	2	avg: 968.031 ms	max: 1192.832 ms	max at: 9595708.51
kworker/11:1H:772	0.394 ms	8	avg: 47.417 ms	max: 379.286 ms	max at: 9595710.93
watchdog/11:76	0.000 ms	8	avg: 23.915 ms	max: 191.294 ms	max at: 9595710.93
handler29:1174	49.499 ms	888	avg: 0.147 ms	max: 92.127 ms	max at: 9595692.00
watchdog/8:58	0.000 ms	8	avg: 8.326 ms	max: 66.576 ms	max at: 9595690.80
kubelet:(8)	6073.132 ms	14910	avg: 0.035 ms	max: 19.201 ms	max at: 9595709.67
udevadm:(5)	9.779 ms	47	avg: 0.476 ms	max: 19.042 ms	max at: 9595706.63
dockerd:(7)	151.344 ms	1977	avg: 0.077 ms	max: 15.467 ms	max at: 9595698.91

图11

根据之前的打点信息和turbostat以及perf的数据，我们非常有理由怀疑是内核的调度有问题，这样我们就用基于rdtscp指令更精准地来获取TSC值来检测CPU是否卡顿。rdtscp指令不仅可以获得当前TSC的值，并且可以得到对应的CPU ID。如图12所示：

```
#include <stdio.h>

unsigned long tacc_rdtscp(int *chip, int *core)
{
    unsigned long int x;
    unsigned a, d, c;
    __asm__ volatile("rdtscp" : "=a" (a), "=d" (d), "=c" (c));
    *chip = (c & 0xFFF000) >> 12;
    *core = c & 0xFFF;
    return (((unsigned long)a) | (((unsigned long)d) << 32));
}

int main()
{
    unsigned long tsc_start, tsc_end;
    int start_chip=0, start_core=0, end_chip=0, end_core=0;
    for (;;)
    {
        tsc_start = tacc_rdtscp(&start_chip, &start_core);
        usleep(100000);
        tsc_end = tacc_rdtscp(&end_chip, &end_core);
        if (tsc_end-tsc_start > 170686502){
            printf("In 0.1 seconds, the TSC advanced by %lu, start chip core:%d-%d, end chip core:%d-%d\n", tsc_end-tsc_start, start_chip, start_core, end_chip, end_core);
        }
    }
    return 0;
}
```

图12

上面的程序编译后，放在宿主机上依次绑核执行，我们发现在问题的宿主机上可以打印出比较大的TSC的值。每间隔100ms去取TSC的值，将获得的相减，在正常的宿主机上它的值应该跟CPU的TSC紧密相关，比如我们的宿主机上TSC是1.7GHZ的频率，那么0.1s它的累加值应该是170000000，正常获得的值应该是比170000000多一点，图13的第五条的值本身就说明了该宿主机的调度延迟在2s以上。


```

In 0.1 seconds, the TSC advanced by 248916960, start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 223017784, start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 241442826, start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 486876966, start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 4708212362, start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 505758500, start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 803742164, start chip core:1-61, end chip core:1-61

```

图13

真相大白

通过上面的检测手段，可以比较轻松地定位问题所在，但还是找不到根本原因。这时我们突然想起来，线上Redis大规模使用宿主机的内核4.14.67并没有类似的延迟，因此我们怀疑这种延迟问题是在4.14.26到4.14.67之间的bugfix修复掉了。

查看commit记录，先二分查找大版本，再将怀疑的点单独拎出来patch测试，终于发现了这个4.14.36-4.14.37之间的（图14）commit:

<https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v4.14.37&id=b8d4055372b58aad4a51b67e176eabdcc238fde3>

x86/acpi: Prevent X2APIC id 0xffffffff from being accounted

commit 10daf10ab154e31237a8c07242be3063fb6a9bf4 upstream.

RongQing reported that there are some X2APIC id 0xffffffff in his machine's ACPI MADT table, which makes the number of possible CPU inaccurate.

The reason is that the ACPI X2APIC parser has no sanity check for APIC ID 0xffffffff, which is an invalid id in all APIC types. See "Intel® 64 Architecture x2APIC Specification", Chapter 2.4.1.

Add a sanity check to acpi_parse_x2apic() which ignores the invalid id.

Reported-by: Li RongQing <lirongqing@baidu.com>

Signed-off-by: Dou Liyang <douly.fnst@cn.fujitsu.com>

Signed-off-by: Thomas Gleixner <tglx@linutronix.de>

Cc: stable@vger.kernel.org

Cc: len.brown@intel.com

Cc: rjw@rjwsocki.net

Cc: hpa@zytor.com

Link: <https://lkml.kernel.org/r/20180412014052.25186-1-douly.fnst@cn.fujitsu.com>

Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>

图14

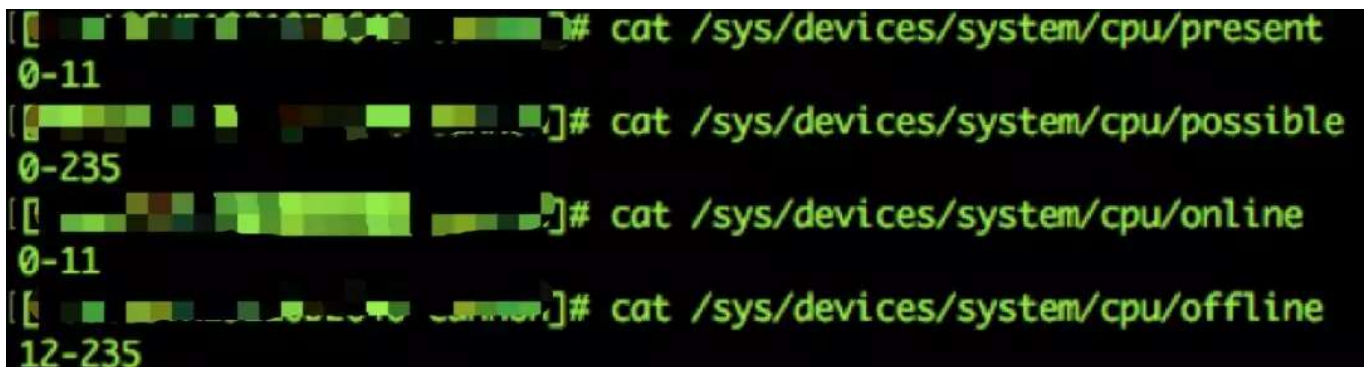
从该commit的内容来看，修复了无效的apic id会导致possible CPU个数不正确的情况，那么什么是x2apic呢？什么又是possible CPU？怎么就导致了调度的延迟呢？

说到x2apic，就必须先知道apic，翻查网上的资料就发现，apic全称Local Advanced Programmable Interrupt Controller，是一种负责接收和发送中断的芯片，集成在CPU内部，每个CPU有一个属于自己的local apic。它们通过apic id进行区分。而x2apic是apic的增强版本，将apic id扩充到32位，理论上支持 $2^{32}-1$ 个CPU。简单的说，操作系统通过apic id来确定CPU的个数。

而possible CPU则是内核为了支持CPU热插拔特性，在开机时一次载入，相应的就有online，offline CPU等，通过修改/sys/devices/system/cpu/cpu9/online可以动态关闭或打开一个CPU，但所有的CPU个数就是possible CPU，后续不再修改。

该commit指出，因为没有忽略apic id为0xffffffff的值，导致possible CPU不正确。此commit看上去跟我们的延迟问题找不到关联，我们也曾向该issue的提交者请教调度延迟的问题，对方也不清楚，只是表示在自己环境只能复现possible CPU增加4倍，vmstat的运行时间增加16倍。

这时我们查看有问题的宿主机CPU信息，奇怪的事情发生了，如图15所示，12核的机器上possible CPU居然是235个，而其中12-235个是offline状态，也就是说真正工作的只有12个，这么说好像还是跟延迟没有关系。



```
[root@k8s-master ~]# cat /sys/devices/system/cpu/present
0-11
[root@k8s-master ~]# cat /sys/devices/system/cpu/possible
0-235
[root@k8s-master ~]# cat /sys/devices/system/cpu/online
0-11
[root@k8s-master ~]# cat /sys/devices/system/cpu/offline
12-235
```

图15

继续深入研究possible CPU，我们发现了一些端倪。从内核代码来看，引用for_each_possible_cpu()这个函数的有600多处，遍布各个内核子模块，其中关键的核心模块比如vmstat，shed，以及loadavg等都有对它的大量调用。而这个函数的逻辑也很简单，就是遍历所有的possible CPU，那么对于12核的机器，它的执行时间是正常宿主机执行时间的将近20倍！该commit的作者也指出太多的CPU会浪费向量空间并导致BUG (<https://lkml.org/lkml/2018/5/2/115>)，而BUG就是调度系统的缓慢延迟。

以下图16，图17是对相同机型相同厂商的两台空负载宿主机的kubelet的perf数据(perf stat -p \$pid sleep 60)，图16是uptime 2天的，而图17是uptime 89天的。

```

Performance counter stats for process id '17564':

      4911.119674      task-clock (msec)      #    0.082 CPUs utilized
        33,814        context-switches      #    0.007 M/sec
         3,388        cpu-migrations        #    0.690 K/sec
        31,001        page-faults           #    0.006 M/sec
    8,235,208,977      cycles                #    1.677 GHz
    11,217,407,462     instructions          #    1.36   insn per cycle
    2,374,349,738      branches              #   483.464 M/sec
      47,377,239      branch-misses          #    2.00% of all branches

      60.001901173 seconds time elapsed

```

图16

```

Performance counter stats for process id '29620':

      1359.985833      task-clock (msec)      #    0.023 CPUs utilized
        34,253        context-switches      #    0.025 M/sec
         2,168        cpu-migrations        #    0.002 M/sec
         1,842        page-faults           #    0.001 M/sec
    2,210,146,665      cycles                #    1.625 GHz
    2,475,897,721     instructions          #    1.12   insn per cycle
    503,869,447        branches              #   370.496 M/sec
      9,120,867        branch-misses          #    1.81% of all branches

      60.002397596 seconds time elapsed

```

图17

我们一眼就看出图16的宿主机不正常，因为无论是CPU的消耗，上下文的切换速度，指令周期，都远劣于图17的宿主机，并且还在持续恶化，这就是宿主机延迟的根本原因。而图17宿主机恰恰只是图16的宿主机打上图14的patch后的内核，可以看到，possible CPU恢复正常(图18)，至此超时问题的排查告一段落。

```

[... ]# cat /sys/devices/system/cpu/present
0-11
[... ]# cat /sys/devices/system/cpu/possible
0-11
[... ]# cat /sys/devices/system/cpu/online
0-11
[... ]# cat /sys/devices/system/cpu/offline
[... ]#

```

图18

总结

我们排查发现，不是所有的宿主机，所有的内核都在此BUG的影响范围内，具体来说4.10（之前的有可能会有影响，但我们没有类似的内核，无法测试）-4.14.37（因为是stable分支，所以master分支可能更靠后）的内核，CPU为skylake及以后型号的某些厂商的机型会触发这个BUG。

确认是否受影响也比较简单，查看possible CPU是否是真实CPU即可。