深入理解 Linux 内核--jemalloc 引起的 TLB shootdown 及优化

原创 基础架构团队 字节跳动技术团队 2020-03-09 09:41

收录于话题
#基础架构 18 #Kernel 4 #Linux 4

> 本文选自“字节跳动基础架构实践”系列文章。
>
> “字节跳动基础架构实践”系列文章是由字节跳动基础架构部门各技术团队及专家倾力打造的技术干货内容，和大家分享团队在基础架构发展和演进过程中的实践经验与教训，与各位技术同学一起交流成长。
>
> “延迟突刺”、“性能抖动”等问题通常会受到多方因素影响不便排查，本文以线上问题为例，详解 TLB shootdown，最终使得 CPU 的消耗降低 2% 左右，并消除了抖动突刺，变得更加稳定。

问题背景

在互联网业务运行的过程中，难免遇到“延迟突刺”、“性能抖动”等问题，而通常这类问题会受到多种软件环境甚至硬件环境的影响，原因较为隐晦，解决起来相对棘手。

本文以一个线上问题为例子，深入 x86 体系结构，结合内核内存管理的知识，辅以多种 Linux 平台上的 Debug 工具，详解 TLB shootdown 问题，最终解决掉该问题，提升了业务性能。

名词约定

Kernel： 本文中特指 Linux-4.14。

KVM： Kernel-based Virtual Machine。现在主流的虚拟机技术之一。

Host： 指虚拟化场景下的宿主机。

Guest： 指虚拟化场景下的虚拟机。

APIC： Advanced Programmable Interrupt Controller 。Intel CPU 使用的中断控制器。

LAPIC： Local Advanced Programmable Interrupt Controller 。

IPI： Inter-Processor Interrupt。CPU 之间相互通知使用。

MMU： Memory Management Unit。Kernel 用来管理虚拟地址和物理地址映射的硬件。

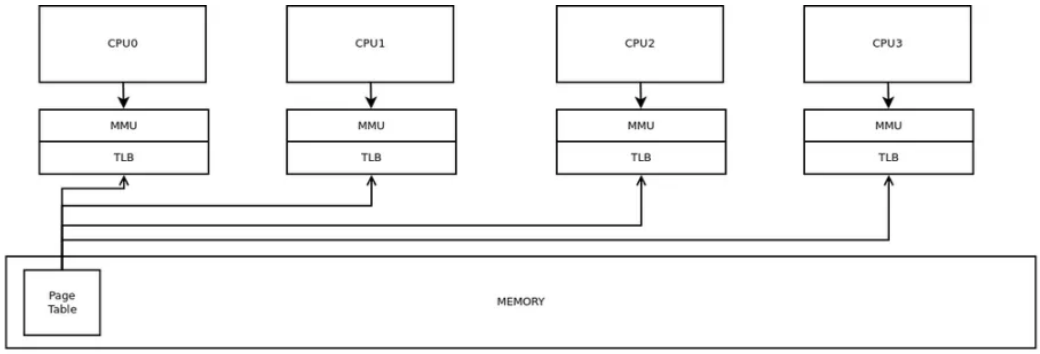
TLB： Translation Lookaside Buffer。MMU 为了加速查找页表，使用的 cache。用来加速 MMU 的转化速度。

PTE： Page Table Entry 。管理页表使用的页表项。

jemalloc： 一个用户态内存管理的库，在多线程并发的场景下，malloc/free 的性能好于 glibc 默认的实现。

TLB shootdown

TLB shootdown 是如何发生的

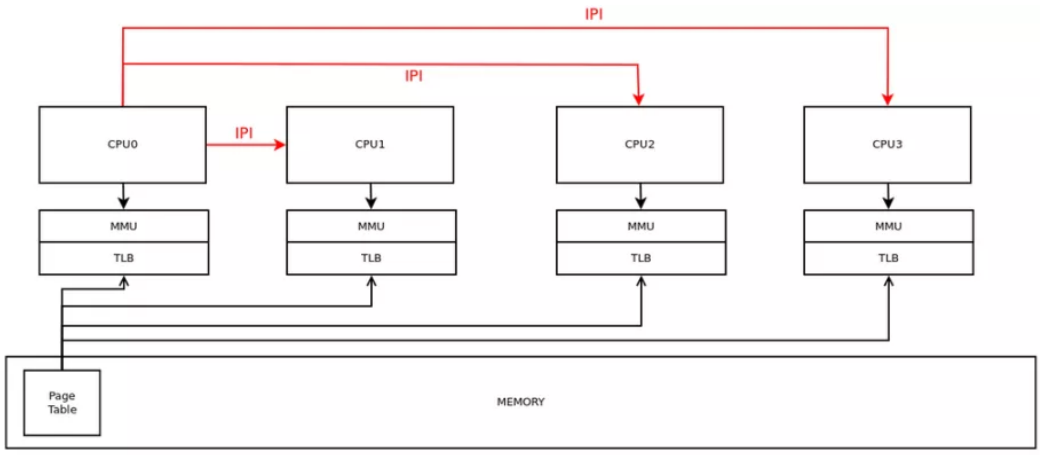


如上图所示，一个进程有 **4** 个 **thread**并行执行。由于 4 个 thread 共享同一个进程的页表，在执行的过程中，通过把 pgd 加载到 cr3 的方式，每个 **CPU** 的 **TLB** 中加载了相同的 **page table**。

如果 CPU0 上，想要修改 page table，尤其是想要释放一些内存，那么需要修改 page table，同时修改自己的 **TLB**（或者重新加载 TLB）。

然而，这还不够。例如，CPU0 上释放了 page A，并且 page A 被 kernel 回收，很有可能被其他的进程使用。但是，**CPU1**、**CPU2** 以及 **CPU3** 的 **TLB** 中还是缓存了对应的 **PTE** 表项，依然可以访问到 **page A**。

为了防止这个事情发生，**CPU0** 需要通知 **CPU1**、**CPU2** 和 **CPU3,**也需要在 **TLB** 中禁用掉对应的 **PTE**。通知的方式就是使用**IPI** （Inter-Processor Interrupt）。



在虚拟化的场景下，IPI 的成本比较高。如果 Guest 中有大量的 IPI，就会看到 Guest 的 CPU sys 暴涨。同时，在 Host 上可以发现虚拟机发生 vmexit 突增，其中主要是 wrmsr 的 ICR Request 产生。（熟悉 x86 的同学知道，x2apic 模式下，x86 上 IPI 的实现即通过 wrmsr 指令请求 ICR）

如何确认是 **TLB shootdown** 引起的问题

1. 在 Guest 中执行：

```
#watch -d -n 1 "cat /proc/interrupts | grep TLB"
```

如果看到数据上涨比较厉害，那么基本就可以看到问题了。

2. 在 Guest 中执行：

```
#perf top
```

如果看到 **smp_call_function_many**，那么很不幸，就是在批量发送 IPI。

好消息是这个场景并不常见，比较特定的情况下才会发生。典型的就是用户态进程中调用了系统调用：

```
int madvise(void *addr, size_t length, MADV_DONTNEED);
```

3. 如何检查进程使用了 jemalloc，jemalloc 会调用 madvise，见下文：

```
# ls /proc/*/maps | xargs grep jemalloc
```

```
root@n19-096-012:/proc# ls /proc/*/maps | xargs grep libjemalloc
grep: /proc/143646/maps: No such file or directory
/proc/1510/maps:7f2563607000-7f2563663000 r-xp 00000000 08:10 131227                     /data00/tiger/databus_deploy/databus2/lib/libjemalloc.so.2
/proc/1510/maps:7f2563663000-7f2563863000 ---p 0005c000 08:10 131227                     /data00/tiger/databus_deploy/databus2/lib/libjemalloc.so.2
/proc/1510/maps:7f2563863000-7f2563866000 rw-p 0005c000 08:10 131227                     /data00/tiger/databus_deploy/databus2/lib/libjemalloc.so.2
/proc/1510/maps:7f2563867000-7f2563868000 rw-p 0027c000 08:10 131227                     /data00/tiger/databus_deploy/databus2/lib/libjemalloc.so.2
/proc/1514/maps:7fc9486b9000-7fc948715000 r-xp 00000000 08:10 1966633                    /data00/tiger/metricserver2/lib/libjemalloc.so.2
/proc/1514/maps:7fc948715000-7fc948915000 ---p 0005c000 08:10 1966633                    /data00/tiger/metricserver2/lib/libjemalloc.so.2
/proc/1514/maps:7fc948915000-7fc948918000 rw-p 0005c000 08:10 1966633                    /data00/tiger/metricserver2/lib/libjemalloc.so.2
/proc/1514/maps:7fc948919000-7fc94891a000 rw-p 0027c000 08:10 1966633                    /data00/tiger/metricserver2/lib/libjemalloc.so.2
```

4. 确实对应的进程是否在执行 madvise 的方法, MADV_DONTNEED 会释放页表项，进而引起 tlb shootdown。所以 MADV_DONTNEED 是判断 tlb 问题的重要线索：

```
# strace -f -p 1510 2>&1 | grep madvise
```

```
root@n19-096-012:/proc# strace -f -p 1510 2>&1 | grep madvise
[pid 2353208] madvise(0x7f2267aae000, 69632, MADV_DONTNEED) = 0
[pid 2353208] madvise(0x7f2267ae5000, 745472, MADV_DONTNEED <unfinished ...>
[pid 2353208] <... madvise resumed> )   = 0
[pid 1358201] madvise(0x7f2267e0d000, 1732608, MADV_DONTNEED <unfinished ...>
[pid 1358201] <... madvise resumed> )   = 0
[pid 1358201] madvise(0x7f2267a0d000, 659456, MADV_DONTNEED <unfinished ...>
[pid 1358201] <... madvise resumed> )   = 0
[pid  2440] madvise(0x7f24d7297000, 45056, MADV_DONTNEED <unfinished ...>
[pid  2440] <... madvise resumed> )     = 0
[pid  2440] madvise(0x7f24d7a0d000, 86016, MADV_DONTNEED) = 0
[pid  2440] madvise(0x7f24d72ac000, 1314816, MADV_DONTNEED) = 0
[pid  2440] madvise(0x7f2210a00000, 4194304, MADV_DONTNEED <unfinished ...>
[pid  2440] <... madvise resumed> )     = 0
[pid  2433] madvise(0x7f22d5800000, 6291456, MADV_DONTNEED) = 0
[pid 1358138] madvise(0x7f221180d000, 1974272, MADV_DONTNEED <unfinished ...>
[pid 1358138] <... madvise resumed> )   = 0
[pid 1358138] madvise(0x7f251e2c7000, 45056, MADV_DONTNEED) = 0
[pid 1358138] madvise(0x7f239a4a7000, 86016, MADV_DONTNEED) = 0
[pid 1358137] madvise(0x7f225bc7d000, 331776, MADV_DONTNEED <unfinished ...>
[pid 1358137] <... madvise resumed> )   = 0
[pid 1358442] madvise(0x7f225bcea000, 659456, MADV_DONTNEED <unfinished ...>
[pid 1358442] <... madvise resumed> )   = 0
[pid  2440] madvise(0x7f24d7297000, 45056, MADV_DONTNEED <unfinished ...>
[pid  2440] <... madvise resumed> )     = 0
[pid  2440] madvise(0x7f24d7a0d000, 86016, MADV_DONTNEED <unfinished ...>
[pid  2440] <... madvise resumed> )     = 0
[pid  2440] madvise(0x7f24d72ac000, 1314816, MADV_DONTNEED <unfinished ...>
[pid  2440] <... madvise resumed> )     = 0
[pid  2440] madvise(0x7f2210a00000, 4194304, MADV_DONTNEED) = 0
[pid  2433] madvise(0x7f22d5800000, 6291456, MADV_DONTNEED <unfinished ...>
[pid  2433] <... madvise resumed> )     = 0
[pid 1358137] madvise(0x7f239a4a7000, 69632, MADV_DONTNEED <unfinished ...>
[pid 1358137] <... madvise resumed> )   = 0
[pid 1358137] madvise(0x7f251e2c7000, 45056, MADV_DONTNEED <unfinished ...>
[pid 1358137] <... madvise resumed> )   = 0
[pid 1358137] madvise(0x7f225bc7d000, 331776, MADV_DONTNEED <unfinished ...>
[pid 1358137] <... madvise resumed> )   = 0
[pid 2353208] madvise(0x7f2267bb1000, 69632, MADV_DONTNEED <unfinished ...>
[pid 2353208] <... madvise resumed> )   = 0
```

madvise MADV_DONTNEED 和 munmap

确认上述的 TLB shootdown 问题之后，我们再来回顾一下，系统调用 madvise 到底起了什么作用呢？

```
int madvise(void *addr, size_t length, MADV_DONTNEED);
```

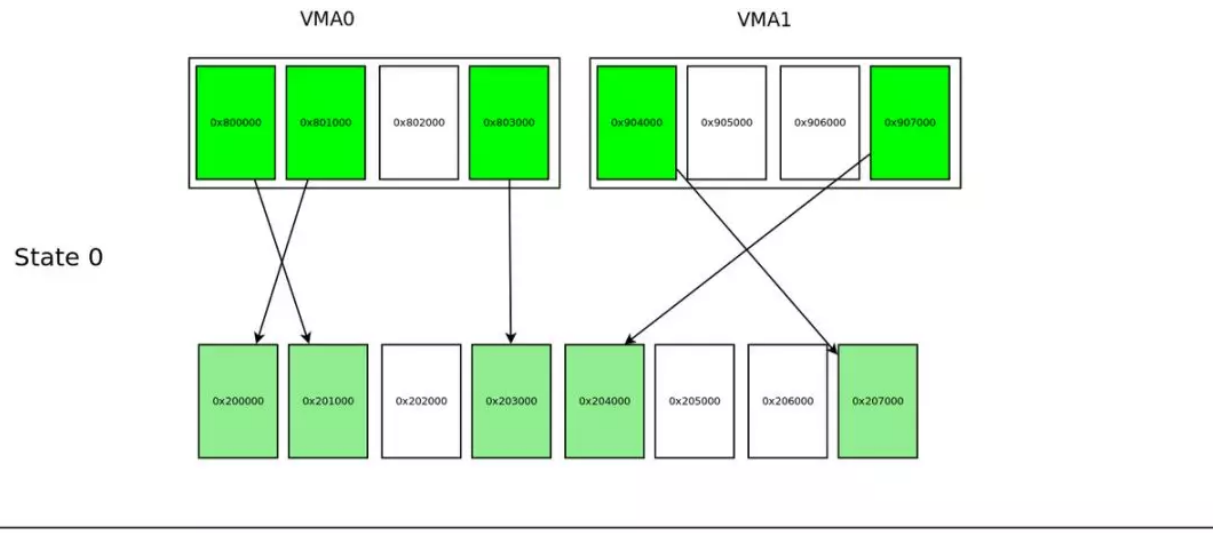
内存分配的一般过程

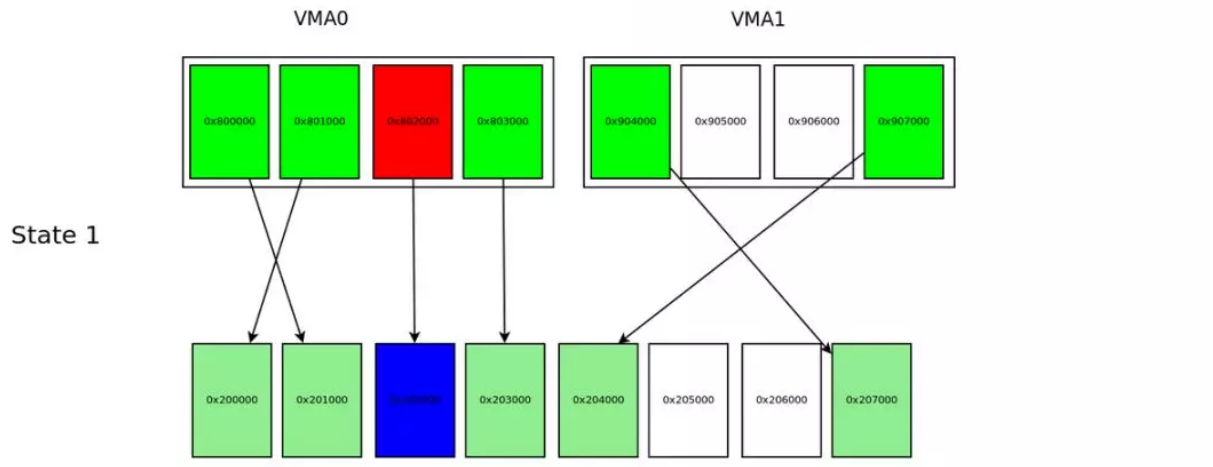
1. 使用 mmap 分配一段虚拟地址空间；
2. 第一次访问到某一个 4k 内的地址的时候，MMU 发现没有对应的 PTE，触发 page fault；
3. kernel 分配对应的 page。

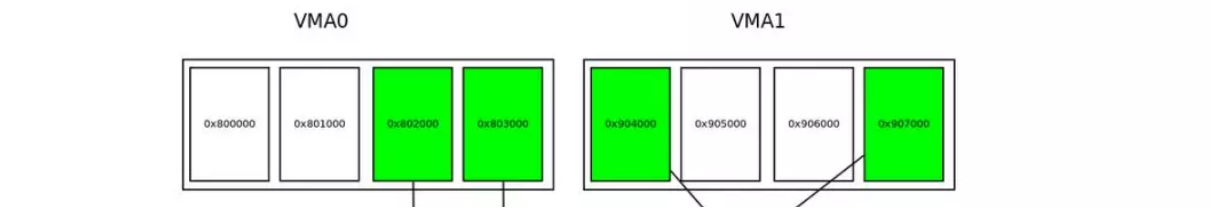
如果使用了 DONTNEED，就会释放对应的 page。如果下一次再访问到，就会重复上述的 2 和 3。

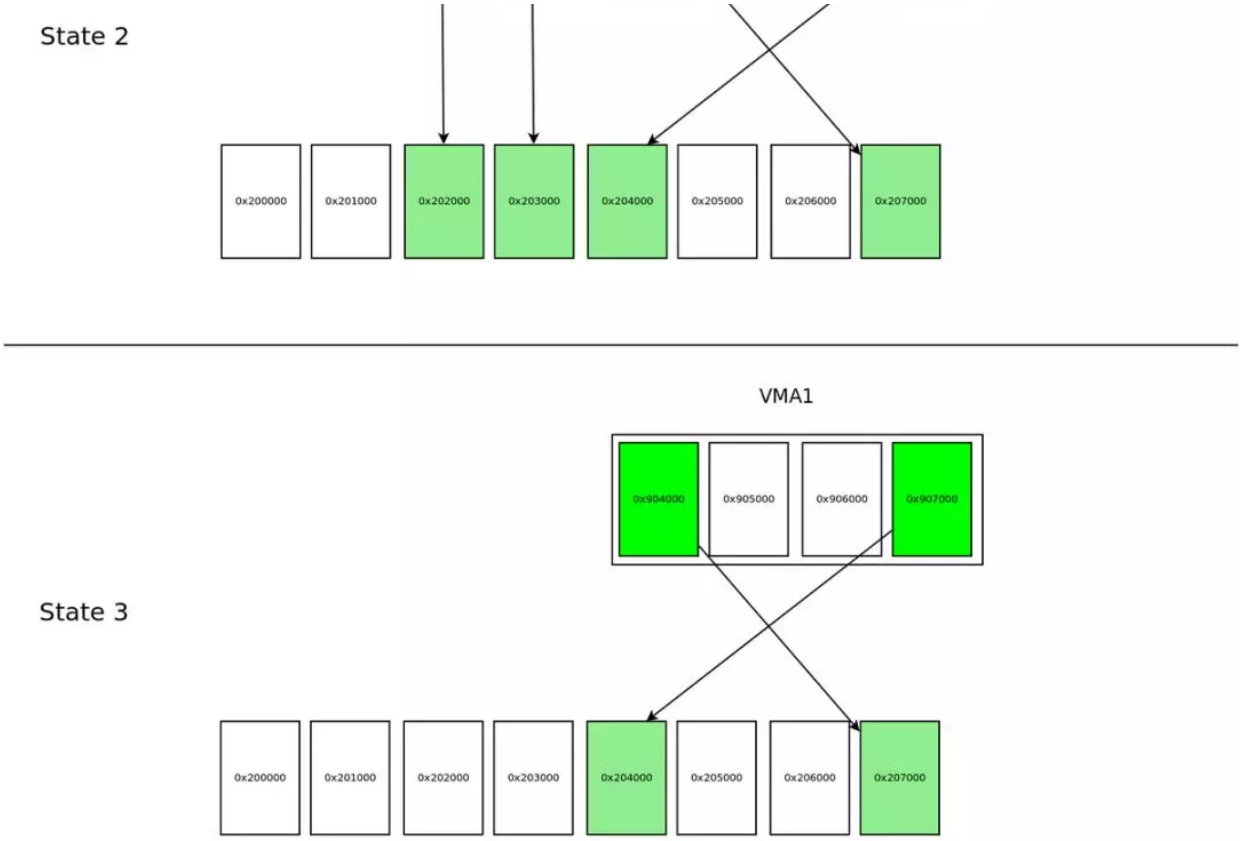
效果就是短暂的 page 归还 kernel 之后，下次访问重新分配。

madvise DONTNEED 和 munmap 的区别









例如 state 0 所示，用户态进程分配了 VMA0 和 VMA1 两个虚拟机地址空间。有的地址上已经分配了物理页面（例如 0x800000），有的还没有分配（例如 0x802000）。

如 state 1 所示，用户态进程第一次访问到了例如 0x802000 地址的时候，触发了 **page fault**，内核为用户态进程的 0x802000 分配了物理页面（地址是 0x202000）。

如 state 2 所示，执行了：

```
madvise(0x800000, 8192, MADV_DONTNEED)
```

之后，内核释放了对应的物理页面。那么下一次访问到 0x800000 ～ 0x801fff 的时候，就会触发 page fault。处理过程类似 state 1。

如 state 3 所示，执行了：

```
munmap(0x800000, 16384);
```

就把对应的 VMA 释放了。那么下次访问到 0x800000 ～ 0x803fff 的时候，就会触发 **segment fault**。因为地址已经释放，属于非法地址，内核会给进程发送 **signal 11**。大部分情况下，会杀掉进程。

使用 **jemalloc ENV** 解决 **TLB shootdown**

问题产生自 jemalloc，所以尝试从 jemalloc 本身入手解决问题。

尝试去社区，问 jemalloc 的 maintainer，是否有办法解决 TLB shootdown 引起的问题，maintainer 建议通过 jemalloc 环境变量（MALLOC_CONF）动态控制 jemalloc 是否启动 madvise。问题和答复见：

```
https://github.com/jemalloc/jemalloc/issues/1422
```

在本地写测试代码，实际测试 jemalloc（比较靠近 upstream 的 **5.0**版本）和 maintainer 给出来的建议，在进程启动前导入环境变量：

```
MALLOC_CONF=dirty_decay_ms:-1,muzzy_decay_ms:-1
```

可以验证可以成功避免问题。该环境变量可以解决 tlb 问题，详细参数作用请参看手册：

```
http://jemalloc.net/jemalloc.3.html#opt.dirty_decay_ms
```

某业务同样使用了 jemalloc，但是测试没有效果。

对业务实际使用的 so 动态链接库进行：

```
#strings libjemalloc.so.2 | grep -i version
```

可以发现实际使用的版本是：

```
JEMALLOC_VERSION "4.2.0-0-gf70a254d44c8d30af2cd5d30531fb18fdabaae6d"
```

通过阅读 jemalloc 的源代码发现，在 4.2 版本的时候，还不支持 maintainer 给出来的变量参数。但是可以通过如下变量来达到类似的效果：

```
MALLOC_CONF=purge:decay,decay_time:-1
```

设置了 jemalloc 的参数之后，业务的表现得到了明显的提升。如下图所示，最后一个零点和前一个零点进行对比，CPU 的抖动情况得到了很大的改善，从之前的 6% 左右抖动到低于 4% 的稳定运行，且 CPU 的消耗曲线更加稳定平滑。

与此同时，业务上的延迟也更加稳定，PCT99 也降低了延迟突刺情况。

写在最后

在解决问题的过程中，也并非如文章所写的一般有序进行。期间也多次使用 perf 观察热点函数的变化；使用 atop 对比前后的业务表现和系统指标；也观察虚拟化的监控数据（wrmsr 的数量）等等手段，一步一步排除干扰，锁定问题。

随着当代操作系统的复杂度的提高，问题的难度也在提高。在解决问题的过程中，我们也在进步！

最后，欢迎加入字节跳动基础架构团队，一起探讨、解决问题，一起变强！

更多分享

字节跳动自研万亿级图数据库 & 图计算实践

字节跳动 EB 级 HDFS 实践

字节跳动基础架构团队

字节跳动基础架构团队是支撑字节跳动旗下包括抖音、今日头条、西瓜视频、火山小视频在内的多款亿级规模用户产品平稳运行的重要团队，为字节跳动及旗下业务的快速稳定发展提供了保证和推动力。

公司内，基础架构团队主要负责字节跳动私有云建设，管理数以万计服务器规模的集群，负责数万台计算/存储混合部署和在线/离线混合部署，支持若干 EB 海量数据的稳定存储。

文化上，团队积极拥抱开源和创新的软硬件架构。我们长期招聘基础架构方向的同学，具体可参见 **job.bytedance.com** （点击左下角“阅读原文”直达官网），感兴趣可以联系邮箱 **arch-graph@bytedance.com** 。

欢迎关注「字节跳动技术团队」

点击阅读原文，快来加入我们吧！

收录于话题 #基础架构 18

上一篇
字节跳动自研万亿级图数据库 & 图计算实践

下一篇
字节跳动在 RocksDB 存储引擎上的改进实践

阅读原文

喜欢此内容的人还喜欢

不完整收录：过去一年字节开源的10个项目丨字节技术年货
字节跳动技术团队