Jemalloc内存分配与优化实践

Team: Data-SYS-STE
Author: 郦泽坤
Last Update: 2023-05-23
Status: Normal

前言

C++ 语言中提供了大量的类库和编程接口，虽然可以帮助开发者提升研发效率，但在特定场景下，其性能表现仍存在优化空间。开发者往往追求极致的代码性能逻辑，一点点的优化改变就可以帮助业务获得良好的性能收益。在字节降本提效的过程中，STE 团队在算力监控系统中发现 Jemalloc 是业务的前五大 CPU 热点基础库，具有很高的潜在性能优化空间。因此，从 2019 年开始对 Jemalloc 进行深度优化，并在字节内部进行了大范围的优化落地，帮助业务团队取得了较好的收益。本文将主要介绍 Jemalloc 的基本原理以及一些简单易用的优化方法，帮助开发者在 Jemalloc 的实际应用中，获得更好的性能表现。

内存相关概念简介

Linux内存分配与分配器

当代 Linux 系统中可以同时运行多种多样的进程，并且进程之间可以做到内存互相隔离，这得益于 Linux 的进程地址空间管理。

一个进程的地址空间中，包含了静态内存、以及动态内存(常说的堆栈)，栈的动态分配和释放由编译器完成，对于堆上内存，Linux 提供了 brk、sbrk、mmap、munmap 等系统调用来进行内存分配和释放，但是这些函数的直接使用会带来不小的理解门槛和使用复杂性，如 brk 需要指定堆的上界地址，容易出现内存错误；mmap 直接申请 pagesize 为单位的内存，对于小于此内存的分配会造成极大的内存浪费。因此需要有内存分配器来辅助管理堆的动态申请和释放。

通常内存分配器如 ptmalloc提供了 malloc 等函数来进行内存的分配，free 进行内存释放，这些函数在底层调用了 brk、mmap 等函数申请内存，并以地址的形式返回给用户，用户在 malloc、free 匹配的情况下不必担心在分配和释放时出现内存错误或者内存浪费。

内存碎片

虽然内存分配器在一定程度上保证了内存的利用率，但是不可避免地会出现内存碎片，包括了内存页内碎片和内存页间碎片，碎片的产生会导致部分内存不可用，内存碎片的大小也是评估一个内存分配器好坏的重要指标。

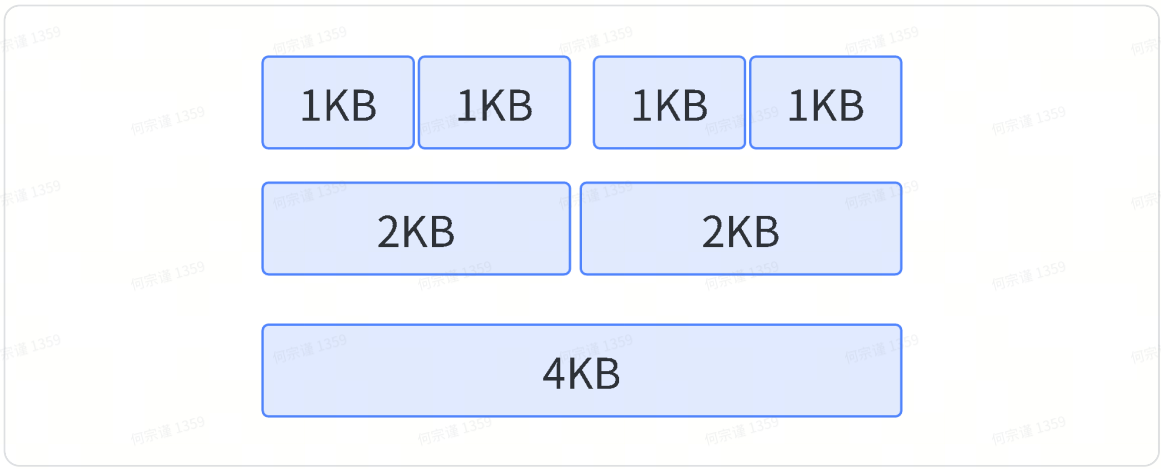
常见的内存分配管理算法

堆上内存以链式形式存在，最简单的动态内存分配算法有：

- First fit：寻找找第一个满足请求 size 的内存块做分配
- Next fit：从当前分配的地址开始，寻找下一个满足请求 size 的空闲块
- best fist：对空闲块进行排序，然后找第一个满足要求的空闲块

另外还有 Buddy 算法和 Slab 算法，也是 jemalloc 中用到的核心算法：

Buddy allocation



Buddy 算法简单来说如上图，一般 2 的 n 次幂大小来管理内存，当申请的内存 size 较小，且当前空闲内存块均大于 size 的两倍，那么会将较大的块分裂，直到分裂出大于 size，并小于 size * 2的块为止；当内存 size 较大时则相反，会将空闲块不断合并。

Buddy 算法没有块间内存碎片，但是块内内存碎片较大，可以看到当申请 2KB+1B 的 size 时，需要用 4KB 的内存块，内存碎片最坏情况可达 50%。

Slab allocation

调用 Linux 系统调用进行内存的分配和释放会让程序陷入内核态，带来不小的性能开销，slab 算法应运而生。每个 slab 都是一块连续内存，并将其划分成大小相同的 slots，用 bitmap 来记录这些 slots 的空闲情况，内存分配时，返回一个 bitmap 中记录的空闲块，释放时，将其记录忙碌即可，而 slab size 和 slot size 是内存碎片大小的关键。

Jemalloc简介

Jemalloc 是 malloc(3) 的实现，在现代多线程、高并发的互联网应用中，有良好的性能表现，并提供了优秀的内存分析功能。

Jemalloc 主要有以下几个特点：

- 高效地分配和释放内存，可以有效提升程序的运行速度，并且节省 CPU 资源
- 尽量少的内存碎片，一个长稳运行地程序如果不控制内存碎片的产生，那么可以预见地这个程序会在某一时刻崩溃，限制了程序的运行生命周期
- 支持堆的 profiling，可以有效地用来分析内存问题
- 支持多样化的参数，可以针对自身地程序特点来定制运行时 Jemalloc 各个模块大小，以获得最优的性能和资源占用比

下文主要介绍了 Jemalloc 的内存分配算法、数据结构，以及一些针对具体程序的优化实践和建议。

Jemalloc核心算法与数据结构

Jemalloc 整体的算法和数据结构基于高效和低内存碎片的原则进行设计，主要体现在：

- 隔离了大 Size 和小 Size 的内存分配(区分默认阈值为 3.5 个 Pagesize)，可以有效地减少内存碎片
- 在内存重用时默认使用低地址，并将内存控制在尽量少的内存页上
- 制定 size class 和 slab class，以便减少内存碎片
- 严格限制 Jemalloc 自身的元数据大小
- 用一定数量的 arena 来管理内存的单元，每个 arena 管理相当数量的线程，arena 之间独立，尽量减少多线程时锁竞争

我们来看下 Jemalloc 是如何来实现这些特性的。

Extent

Jemalloc 的内存管理结合了 buddy 算法和 slab 算法。Jemalloc 引入 extent 的概念，extent 是 arena 管理的内存对象，在 large size 的 allocation 中充当 buddy 算法中的 chunk，small size allocation 中，充当 slab。

每个 extent 的大小是 Pagesize 的整数倍，不同 size 的 extent 会用 buddy 算法来进行拆分和合并，大内存的分配会直接使用一整个的 extent 来存储。小内存的分配使用的是 slab 算法，slab size 的计算规则为 size 和 pagesize 的最小公倍数，因此每个extent总是可以存储整数倍个对应 size。

extent 本身设置 bitmap，来记录内存占用情况，以及自身的各种属性，同类型的 extents 用 paring heap 存储，

此外，arena 将 extent 分为多种类型，有当前正在使用未被填满的extent，有一段时间未使用的dirty extent，还有长时间未使用的muzzy extent，以及开启retained功能后的 retained extent，extent分类的作用相当于多级缓存，当线程内存分配压力较小时，空余的extent会被缓存，以备压力增大时使用，可以避免与操作系统的交互。

Small size align and Slab size

为了减少页内内存碎片，Jemalloc 对 small size 进行了对齐，对于每一个 size，以二进制的视角来看，将其分为

两个数：group、mod。group 表示 size 的二进制最高位，如果 size 正好为 2 的幂次，则将其分在上一个 group 中；mod 表示最高位的后两位，有 0、1、2、3 共 4 种可能。这样构成的 align 后的 size 在同一个 group 中步长（即两个相邻 mod 计算得到的 size 之间的差值）相同，group 越大，步长会呈 2 的倍数增长。

如下图，框中的 4个是同一个 group 中 4 种 mod 在 align 后的 size，其中 160 表示包含了 129B 到 160B 在对齐后的大小：

1024	512	256	128	64	32	16	mod	
				1	0	0		reg_size = 64
				1	0	1	0	reg_size = 80
				1	1	0	1	reg_size = 96
				1	1	1	2	reg_size = 112
			1	0	0		3	reg_size = 128
			1	0	1		0	reg_size = 160
			1	1	0		1	reg_size = 192
			1	1	1		2	reg_size = 224
		1	0	0			3	reg_size = 256
		1	0	1				reg_size = 320
		1	1	0				reg_size = 384
		1	1	1				reg_size = 448
	1	0	0					reg_size = 512
	1	0	1					reg_size = 640
	1	1	0					reg_size = 768
	1	1	1					reg_size = 896
1	0	0						reg_size = 1024
1	0	1						reg_size = 1280
1	1	0						reg_size = 1536
1	1	1						reg_size = 1792
								reg_size = 2048
								reg_size = 2560
								reg_size = 3072
								reg_size = 3584
								reg_size = 4096
								reg_size = 5120
								reg_size = 6144
								reg_size = 7168
								reg_size = 8192
								reg_size = 10240
								reg_size = 12288
								reg_size = 14336

计算出 aligned size 后，就需要计算 slab size，每个 slab size 为 pagesize 和 aligned size 的最小公倍数，以防止跨 slab 的 size 或者 slab 无法被填满的情况出现。以 4K page 为例，128B 的 slab size 即 4K，160B 的 slab size 为 20K。

buddy 算法和 slab 算法

张青山 2023年5月5日

感觉可以在这个前面介绍 buddy 和 slab 的基本思路，不要放在前面介绍，逻辑上有点突兀。

郦泽坤 2023年5月5日

buddy 和 slab 算是背景知识，在 os 上用到的会比较多，所以放在背景中和其他的分配算法一起讲

xtent 本身设置 bitmap

张青山 2023年5月5日

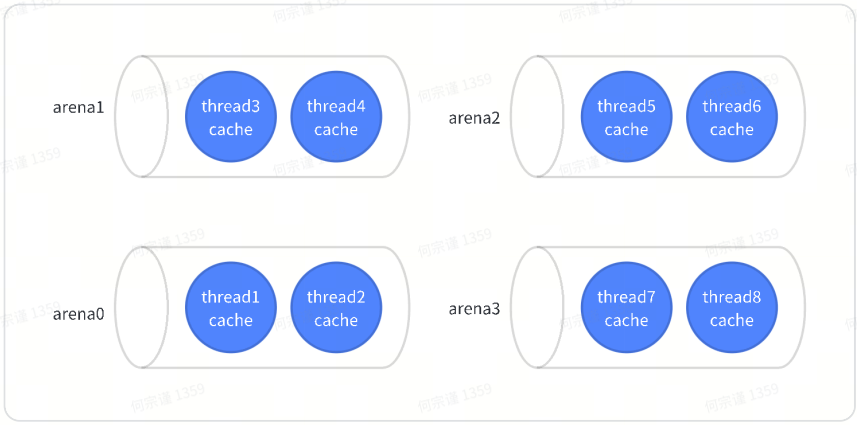
extend 的逻辑意义我们这边好画一个图吗，可以和 slab 的图做类比，看的更清楚一点。

郦泽坤 2023年5月5日

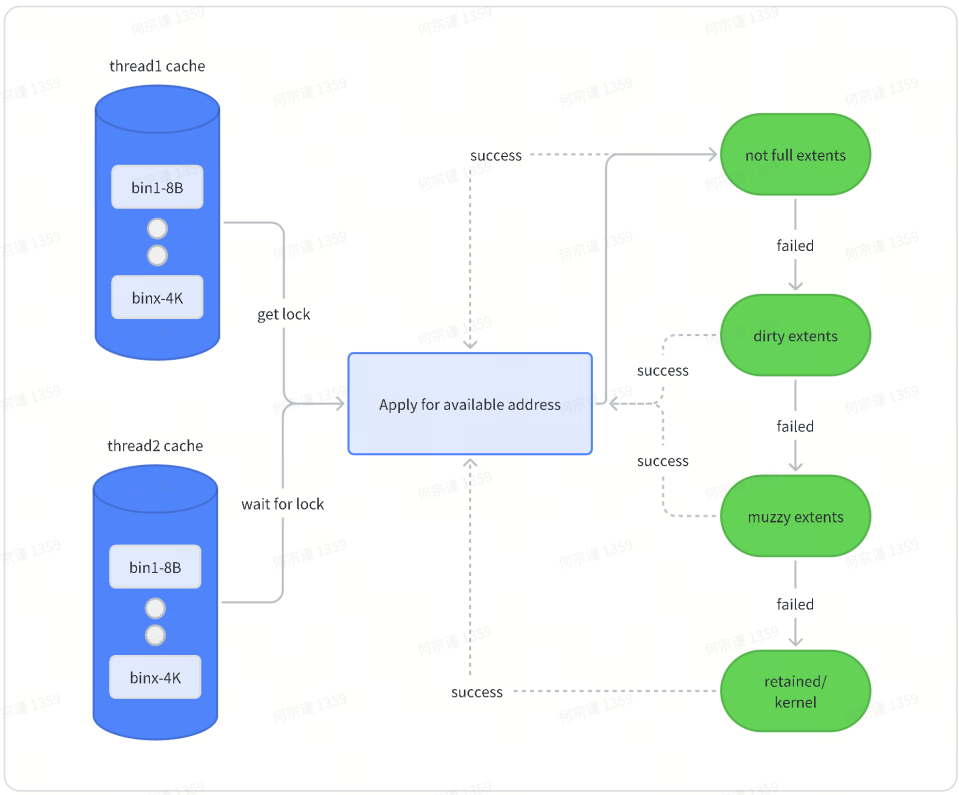
extent 用作 slab 的时候和 slab 是一样的，没有直观比较的差异

Tcache and arena

为了减少多线程下锁的竞争，Jemalloc 参考 lkmalloc 和 tcmalloc，实现了一套由多个 arena 独立管理内存加 thread cache 的机制，形成 tcache 有空余空间时不需要加锁分配，没有空余空间时将锁控制在线程所属 arena 管理的几个线程之间的模式。



tcache 中每一个 size 对应一个 bin，当 tcache 需要填充时，在 arena 中发生的如下图：



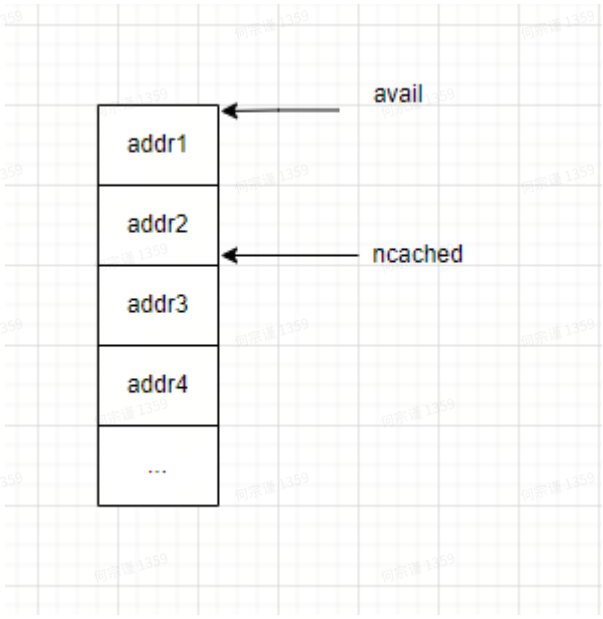
allocation/dallocation in tcache

tcache 以 thread local storage对象的形式存储，主要服务于 small size 和一小部分 large size。

当 tcache 中有空闲时，一次 malloc 的过程很简单：

1. 对 size 做 align 得到 usize
2. 查找 usize 对应的 bin，bin 为 tcache 中针对不同 size 设置的 slots
3. bin 有空闲地址则直接返回，没有空闲地址则会向 arena 请求填充

每个 bin 的结构如下图，avail 指向 bin 的起始地址，ncached 初始为 bin 的最大值 ncached_max (与 slab size 相关，最小为 20 最大为 200)，每次申请内存会返回 ncached 指向的地址并自减1，直到小于限制值。



释放的时候相反，当 tcache 不为空，即 ncached 不等于 bin 的 ncached_max 时，ncached 自加1，并且将 free 的地址填入 bin 中。

Tcache fill

上面的 allocation 过程是 tcache 中有足够的空闲块供分配，当 tcache 中已经没有空闲块时，会向其所属的 arena 申请 fill，此时 arena 中会加锁去分级 extent 取空闲块，并把当前使用的 extent 移入full extent。

Tcache flush

当 dallocate 触发 tcache 中又没有分配任何内存，即 ncached_max 等于 ncached_max 时，tcache 会触发 flush，flush 会将 tcache 中一半的内存释放回原 extent，即将 tache 的可用空间压缩到原来的一半，这个过程中也会加对应 extent 的锁以保证同步。

Jemalloc优化思路

从上一章节可以看到，jemalloc 对于内存用的是多级缓存的思路，tcache 的代价最小，无须加锁可以直接返回；其次是 arena 的 bin->extent，锁的粒度在对应的 bin 上，会是 bin 对应的 size 在这个 arena 中无法再做 fill 或 flush；然后是 dirty extent、muzzy extent，这部分是 arena 全局加锁，会锁住其他线程的 fill 或者 purge，那么在多线程下，我们可以用几个思路来优化锁的竞争。

arena优化

从上一章节可知，jemalloc 将锁的范围都控制在 arena 中，每个 arena 会管理一系列线程，线程在 arena 中是平均分配的，arena 默认数量是 CPU 个数 * 4。因此，当我们在一台 8 核的机器上运行 256 个线程时，意味着每个arena 需要管理 8 个线程，这些线程在内存任务繁重时会产生严重的锁竞争，从而影响性能。此时可选择使用 malloc_conf:narenas:128，增加 arena 数量到 128 个，每个 arena 只需管理 2 个线程，线程之间产生锁竞争的概率就会大大减小。

此外还可以选择用 mallocx 隔离线程，让内存分配任务较重的线程独占 arena。

Slab size优化

Slab size 的大小如上所述，为 usize 大小和 pagesize 的最小公倍数，这一机制可以保证减少内存碎片，但是tcache 的 fill 与 flush 都与 slab size 相关，一个和业务内存模型匹配的 slab class 才可以得到最好的性能效果。

下面是一张 jemalloc 和 ptmalloc 的对比图，可以看到在 1024 以下的性能 jemalloc 都优于 ptmalloc，但是jemalloc 自身的性能明显存在波动，几个波动出现在 128B、

将 tcache 中一半的内存释放回原 extent

谭锦彪 2023年7月12日

这里释放的是 no full extent 吗

Jemalloc优化思路

谭锦彪 2023年7月12日

也介绍一下 bin_shards 优化?

arena优化

徐林 2023年10月11日

多进程模型适用么，比如 Nginx

郦泽坤 2023年10月11日

不共享内存的话不适用

管理一系列线程

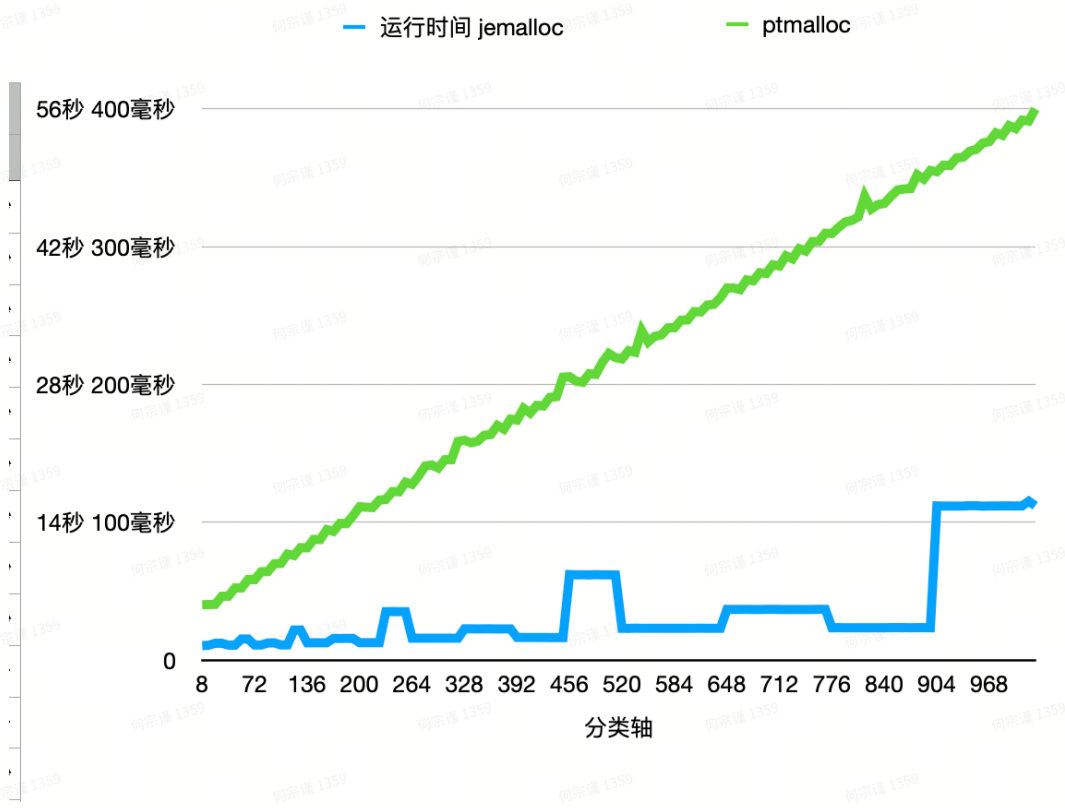
张明明 2024年10月18日

小白求问下这里管理的是分配或者释放内存的线程是吗？线程修改内存不在这个管理或者分配的范围内。

郦泽坤 2024年10月18日

对的，jemalloc 只管分配和释放

256B、512B 以及 1024B 周围，因为这些 size 本身就是 pagesize 的因子或者公因子较多，所以 slab size 占用的 page 数也相对较少，fill 和 flush 所需要的slab数也越多。



dirty decay & muzzy decay

尽管我们希望将所有的 malloc、free 内存都可以放在 tcache 中或者 bin 中，这样可以最大化执行效率，但是实际的程序中这很难做到，因为每个线程都需要增加内存，会造成不小的内存压力，而且内存的申请释放往往会有波峰，dirty extents 和 muzzy extents 就可以来应对这些内存申请的波峰，而避免需要转入内核态来重新申请内存页。

dirty_decay_ms 和 muzzy_decay_ms 是 jemalloc 中用来控制长时间空闲内存衰变的时间参数，适当地扩大 dirty decay 的时间可以有效地解决性能劣化的尖刺。

Tcache ncached_max

tcache 中每一个 bin 的 slots 数量由 ncached_max 决定，当 tcache 中 ncached_max 耗尽时会触发 arena 的 fill tcache 而产生锁，而 ncached_max 的大小默认为 2 * slab size，最小为 20，最大为 200，适当地扩大 ncached_max 值可以在一些线程上形成更优的 allocation/deallocation 循环（5.3版本已支持用malloc_conf进行更改）。

优化方法：调优三板斧

结合以上优化思路，通过以下步骤对应用进行调优：

Dump stats

在 long exist 的程序中可调用 jemalloc 的 malloc_stats_print 函数，dump 出应用当前内存分配信息：

```
// reference: https://jemalloc.net/jemalloc.3.html
void malloc_stats_print(void (*write_cb)(void *, const char *),
// 回调函数，可以写入文件
                        void *cbopaque, // 回调函数参数
                        const char *opts); // stats的一些选项，
如"J"是导出json格式
```

或通过设置 malloc_conf，在程序运行结束后自动 dump stats：

适当地扩大 dirty decay 的时间可以有效…

张彪 2024年9月16日

请问有最佳实践吗？一般设置几秒合适呀？

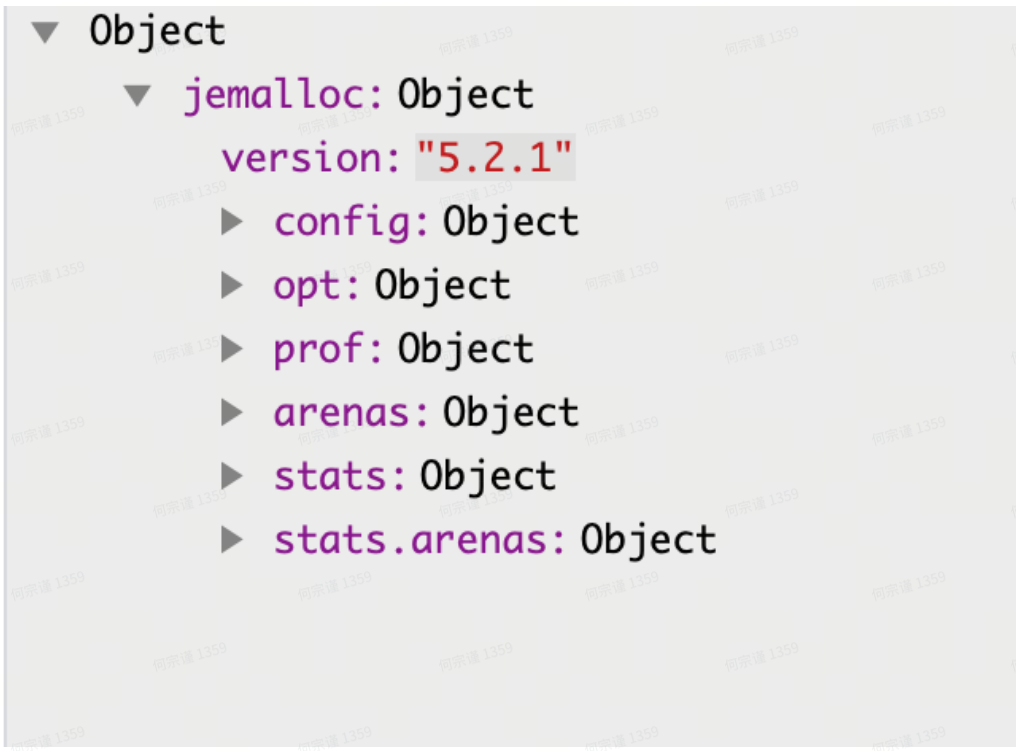
郦泽坤 2024年9月18日

这个不同的 workload 有不同的需求，对于性能来说肯定是越大越好，但是会占用大量内存，所以是一个 performance & memory 的 tradeoff

```
export MALLOC_CONF=stats_print:true
```

stats 分析

用 Json 格式 dump stats 后，可以得到如下图所示结构的 json 文件：



各字段含义可参考：https://jemalloc.net/jemalloc.3.html。

按上一章节思路，可主要关注以下几点：

- arena 数量与 threads 数量比例
  - arena 数：jemalloc->arenas->narenas
  - threads 数：jemalloc->stats.arenas->merged->nthreads
    - **分析：threads : arenas 比例代表了单个 arena 中管理的线程数，将 malloc、free 较多，并且有可能产生竞争的线程尽量独占 arena**
- 各个 extent 中 mutex 开销
  - jemalloc->stats.arenas->merged->mutexes
    - **分析：该节点中的 mutex 操作次数、等锁时间可以反映出该类型 extent 的锁竞争程度，若 extent_retained 锁竞争严重，可适当调大 muzzy_decay_ms；同理，当 extents_muzzy 锁竞争严重，可适当调大 diry_decay_ms；extents_dirty 锁竞争严重，可适当调大 ncached_max，让malloc 尽量可以在 tcache 中完成**
- arena 中各个 bin 的 malloc、free 次数
  - jemalloc->stats.arenas->merged->bins
    - **分析：bin 中的 nfills 可以反映该 slab 填充的次数，针对 regions 本身较少，nfill 次数又多的 size，如 521B、1024B、2048B、4096B 等，可适当调大 slab size 来减小开销**

添加MALLOC_CONF参数或修改代码

MALLOC_CONF 是 jemalloc 中用来动态设置参数的途径，无须重新编译二进制，可以通过 MALLOC_CONF 环境变量或者 /etc/malloc_conf 软链接形式设置，参数之间用 ',' 分割。除需要使用线程独占的 arena 外，以上其他优化均可通过 MALLOC_CONF 配置来完成。

arena优化方法

narenas 设置：

```
export MALLOC_CONF=narenas:xxx  # xxx最大为4096
```

或

```
ln -s "narenas:xxx" /etc/malloc_conf
```

设置线程独占的 arena：

```
unsigned thread_set_je_exclusive_arena() {
  unsigned arena_old, arena_new;
  size_t sz = sizeof(unsigned);

  /* Bind to a manual arena. */
  if (mallctl("arenas.create", &arena_new, &sz, NULL, 0)) {
    std::cout << "Jemalloc arena create error\n";
    return 0;
  }
  if (mallctl("thread.arena", &arena_old, &sz, &arena_new,
sizeof(arena_new))) {
    std::cout << "Thread bind to jemalloc arena error\n";
    return 0;
  }
  return arena_new;
}

```

各类大小优化方法

dirty extents：

```
export MALLOC_CONF=dirty_decay_ms:xxx  # -1为不释放dirty extents，易
发生OOM
```

muzzy extents：

```
export MALLOC_CONF=muzzy_decay_ms:xxx  # -1为不释放muzzy extents，易
发生OOM
```

tcache ncached_max 调整，ncached_max 与 slab size 相关，计算方式为

```
(slab_size / region_size) << lg_tcache_nslots_mul (默认值1)
```

最大限值为 tcache_nslots_small_max(默认200)，最小限值为 tcache_nslots_small_min(默认20)。

如调整 32B 的 ncached_max，当前系统 page size 为 4K，计算默认的 ncached_max 的方法：

```
(slab_size / region_size) << lg_tcache_nslots_mul = (4096 / 32) << 1 = 256
```

超过了 tcache_nslots_small_max，所以 32B 的 ncache_max 默认即为 200。

调整 ncached_max 默认值相关参数：

```
export MALLOC_CONF=tcache_nslots_small_min:xxx,tcache_nslots_small_max:xxx,lg_tcache_nslots_mul:xxx
```

Slab size 设置方法：

```
export MALLOC_CONF="slab_sizes:1-4096:17|100-200:1|128-128:2" # -左右表示size范围，:后设置page数，|分割各个不同的size范围
```

字节业务优化案例

Jemalloc 的 stats dump 已经集成到监控系统中，经过分析发现，字节内部的应用普遍线程数量较多，在 arena 中的锁竞争比较激烈，并且 allocte/deallocte 集中在某些线程中，因此可以通过让核心线程独占 arena 来完成优化。

以其中一个业务在平台上的 stats 数据为例：

Jemalloc&Arenas 配置信息

统计详情

Allocate内存统计	allocated	18445838991400376000	metadata_thp	7490	resident	142782496768
	active	111543451648	mapped	145091198976	mutexes	点击查看
	metadata	15367000680	retained	1209706217472	background_thread	点击查看

arenas_id	nthreads	mapped	metadata_thp	retained	base	tcache_bytes	resident	mute
merged	1776	145091198976	7490	1209706217472	15367000680	1243806248	142782496768	点击查
256	0	4957667328	0	97679048704	129576	0	4955701248	点击查
255	7	1236729856	395	66162589696	826636128	6898496	1224540160	点击查
254	7	196935680	2	317915136	2716944	4176984	193396736	点击查
253	7	263127040	12	2239823872	23377992	4492064	255066112	点击查

可以看到进程总线程数为 1776 个，arenas 数量为 256 个，平均每个 arena 中的内存需要有 7-8 个线程共享，再查看 mutexs 的 stats：

字节业务优化案例

桑春雷 2023年5月25日

这个目录建议提一级

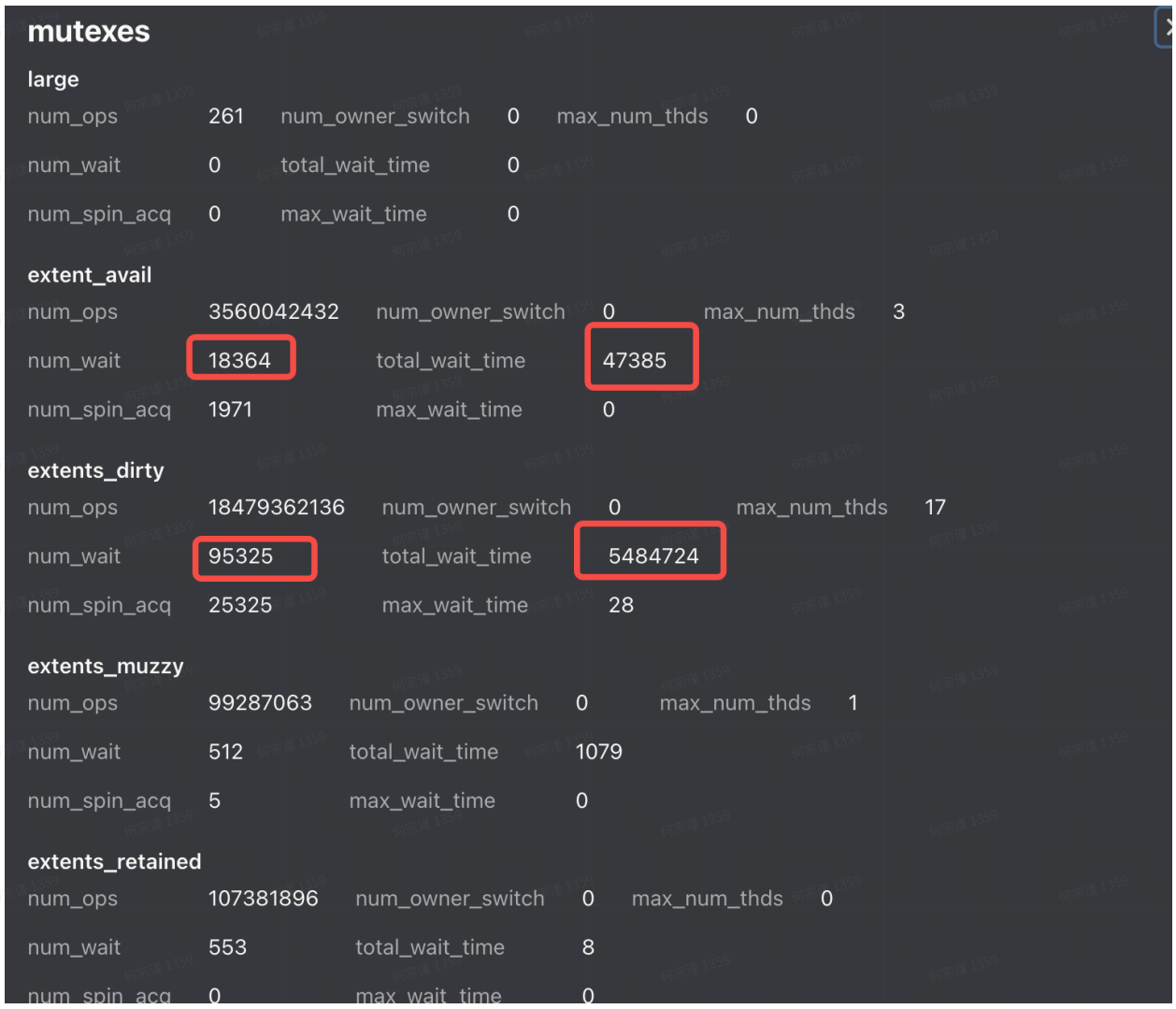
[图片]

王莹 2023年5月17日 （编辑过）

辛苦再次 check 图片是否已完成脱敏

郦泽坤 2023年5月17日

这里没啥业务相关的名称，最多有一些应用内存相关统计数据，我理解不是敏感数据



可以看到在 extents 中产生锁的开销并不小，首先选择扩大 arenas 的数量，从 256 扩大到 1024 个，发现 CPU 相对下降了 4.5%，但是相对的内存上涨了 10%，在分析代码后，发现 allocate/deallocate 较多的线程总数量只有80+，针对这些线程通过 mallctl 单独创建了 arena，并绑定 tcache，并调小其他线程的 muzzy_decay_ms，最终为该业务节省了 4% 的 CPU 收益，内存基本持平。

总结

最好的基础库总是通用的，最适合的基础库总是最个性化的。对于 jemalloc 在业务上的优化与实践，STE团队进行不断探索，采集内存信息并进行平台化展示，以便业务及时发现自身程序在 jemalloc 上的性能瓶颈，并做出针对性地调优，目前已在 10+ 业务上进行参数优化，平均帮助各业务团队节省了 3% 的 CPU（jemalloc 在部分业务中平均占用 10% ~ 15% 的CPU）。未来 STE 团队将继续深入 jemalloc 性能优化，探索定制化的业务内存分配和管理方案，以获得更好的优化成果。