

## 2.7 Thread caches (tcache\_t)

TLS/TSD是另一种针对多线程优化使用的分配技术, jemalloc中称为tcache. tcache解决的是同一cpu core下不同线程对heap的竞争. 通过为每个线程指定专属分配区域,来减小线程间的干扰. 但显然这种方法会增大整体内存消耗量. 为了减小副作用, jemalloc将tcache设计成一个bookkeeping结构, 在tcache中保存的仅仅是指向外部region的指针, region对象仍然位于各个run当中. 换句话说, 如果一个region被tcache记录了, 那么从run的角度看, 它就已经被分配了.

tcache的内容如下,

```
1 struct tcache_s {
2     ql_elm(tcache_t) link;
3     uint64_t         prof_accumbytes;
4     arena_t          *arena;
5     unsigned         ev_cnt;
6     unsigned         next_gc_bin;
7     tcache_bin_t     tbins[1];
8 };
```

- **link**: 链接节点, 用于将同一个arena下的所有tcache链接起来.
- **prof\_accumbytes**: memory profile相关.
- **arena**: 该tcache所属的arena指针.
- **ev\_cnt**: 是tcache内部的一个周期计数器. 每当tcache执行一次分配或释放时, ev\_cnt会记录一次. 直到周期到来, jemalloc会执行一次 incremental gc. 这里的gc会清理tcache中多余的region, 将它们释放掉. 尽管这不意味着系统内存会获得释放, 但可以解放更多的region 交给其他更饥饿的线程以分配.
- **next\_gc\_bin**: 指向下一次gc的binidx. tcache gc按照一周期清理一个bin执行.
- **tbins**: tcache bin数组. 同样外挂在tcache后面.

不是大小限制运行gc

同arena bin类似, tcache同样有tcache\_bin\_t和tcache\_bin\_info\_t. tcache\_bin\_t作用类似于arena bin, 但其结构要比后者更简单. 准确的说, **tcache bin并没有分配调度的功能, 而仅起到记录作用**. 其内部通过一个stack记录指向外部arena run中的region指针. 而一旦region被cache到tbins内, 就不能再被其他任何线程所使用, 尽管它可能甚至与其他线程tcache中记录的region位于同一个arena run中.

tcache bin结构如下,

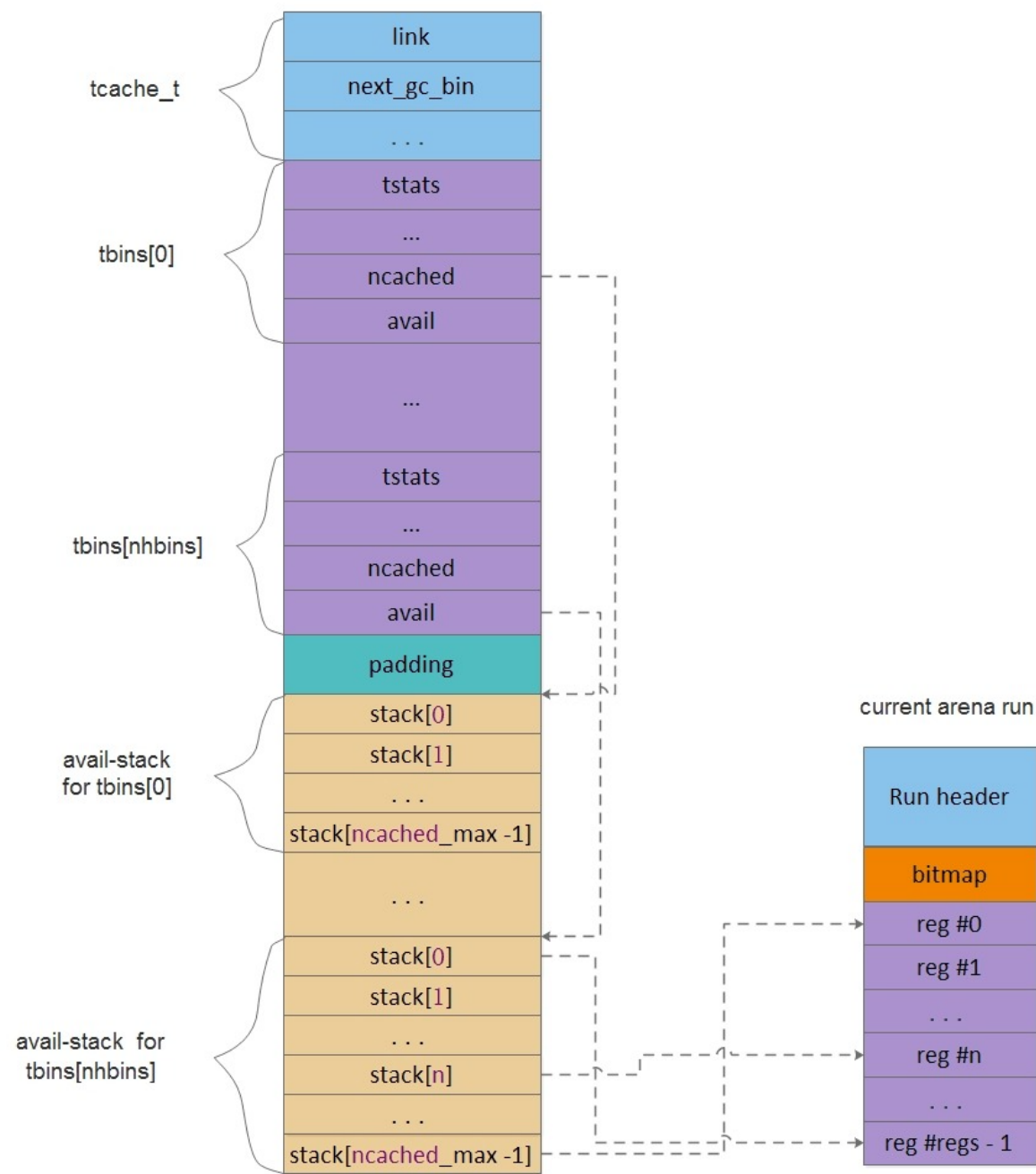
```
1 struct tcache_bin_s {
2     tcache_bin_stats_t tstats;
3     int                low_water;
4     unsigned           lg_fill_div;
5     unsigned           ncached;
6     void               **avail;
7 };
```

- **tstats**: tcache bin内部统计.
- **low\_water**: 记录两次gc间tcache内部使用的最低水线. 该数值与下一次gc时尝试释放的region数量有关. 释放量相当于low water数值的3/4.
- **lg\_fill\_div**: 用作tcache refill时作为除数. 当tcache耗尽时, 会请求arena run进行refill. 但refill不会一次性灌满tcache, 而是依照其最大容量缩小 $2^{\lg\_fill\_div}$ 的倍数. 该数值同low\_water一样是动态的, 两者互相配合确保tcache处于一个合理的充满度.
- **ncached**: 指当前缓存的region数量, 同时也代表栈顶index.
- **avail**: 保存region指针的stack, 称为avail-stack.

tcache\_bin\_info\_t保存tcache bin的静态信息. 其本身只保存了tcache max容量. 该数值是在tcache boot时根据相对应的arena bin的nregs决定的. 通常等于nregs的二倍, 但不得超过TCACHE\_NSLOTS\_SMALL\_MAX. 该数值默认为200, 但在android中大大提升了该限制, small bins不得超过8, large bins则为16.

```
1 struct tcache_bin_info_s {
2     unsigned    ncached_max;
3 };
```

tcache layout如下,



2.8 Extent Node (extent\_node\_t)

extent node代表huge region, 即大于chunk大小的内存单元. 同arena run不同, extent node并非是一个header构造, 而是外挂的. 因此其本身仍属small region. 只不过并不通过bin分配, 而由base\_nodes直接动态创建.

jemalloc中对所有huge region都是通过rb tree管理. 因此extent node同时也是tree node. 一个node节点被同时挂载到两棵rb tree上. 一棵采用szad的查询方式, 另一棵则采用纯ad的方式. 作用是当执行chunk recycle时查询到可用region, 后面会详述.

```
1 struct extent_node_s {
2     rb_node(extent_node_t)    link_szad;
3     rb_node(extent_node_t)    link_ad;
4     prof_ctx_t                *prof_ctx;
5     void                       *addr;
6     size_t                     size;
7     arena_t                    *arena;
8     bool                       zeroed;
9 };
```

- **link\_szad**: szad tree的link节点.
- **link\_ad**: ad tree的link节点.
- **prof\_ctx**: 用于memory profile.
- **addr**: 指向huge region的指针.
- **size**: region size.
- **arena**: huge region所属arena.
- **zeroed**: 代表是否zero-filled, chunk recycle时会用到.

## 2.9 Base

base并不是数据类型, 而是一块特殊区域, 主要服务于内部meta data(例如arena\_t, tcache\_t, extent\_node\_t等等)的分配. base区域管理与如下变量相关,

```
1 static malloc_mutex_t    base_mtx;
2 static void              *base_pages;
3 static void              *base_next_addr;
4 static void              *base_past_addr;
5 static extent_node_t     *base_nodes;
```

- **base\_mtx**: base lock
- **base\_pages**: base page指针, 代表整个区域的起始位置.
- **base\_next\_addr**: 当前base指针, 类似于brk指针.
- **base\_past\_addr**: base page的上限指针.
- **base\_nodes**: 指向extent\_node\_t链表的外挂头指针.

base page源于arena中的空闲chunk, 通常情况下, 大小相当于chunk. 当base耗尽时,会以chunk alloc的名义重新申请新的base pages.

为了保证内部meta data的快速分配和访问, Je将内部请求大小都对齐到cache-line上,以避免在SMP下的false sharing. 而分配方式上, 采用了快速移动base\_next\_addr指针进行高速开采的方法.

```
1 void *base_alloc(size_t size)
2 {
3     .....
4     // xf: 将内部分配请求对齐的cache-line上, 阻止false sharing
5     csize = CACHELINE_CEILING(size);
6
7     malloc_mutex_lock(&base_mtx);
8     // xf: 如果base耗尽, 则重新分配base_pages. 默认大小为chunksize.
9     if ((uintptr_t)base_next_addr + csize > (uintptr_t)base_past_addr) {
10         if (base_pages_alloc(csize)) {
11             malloc_mutex_unlock(&base_mtx);
12             return (NULL);
13         }
14     }
15     // xf: 快速向前开采
16     ret = base_next_addr;
17     base_next_addr = (void *)((uintptr_t)base_next_addr + csize);
18     malloc_mutex_unlock(&base_mtx);
19
20     return (ret);
21 }
```

与通常的base alloc有所不同, 分配extent\_node\_t会优先从一个node链表中获取节点, 而base\_nodes则为该链表的外挂头指针. 只有当其



耗尽时, 才使用前面的分配方式. 这里区别对待extent\_node\_t与其他类型, 主要与chunk recycle机制有关, 后面会做详细说明.

有意思的是, 该链表实际上借用了extent node内部rb tree node的左子树节点指针作为其link指针. 如2.7节所述, extent\_node\_t结构的起始位置存放一个rb node. 但在这里, 当base\_nodes赋值给ret后, 会强制将ret转型成(extent\_node\_t\*\*), 实际上就是指向extent\_node\_t结构体的第一个field的指针, 并将其指向的node指针记录到base\_nodes里, 成为新的header节点. 这里需要仔细体会这个强制类型转换的巧妙之处.

```

1  ret = base_nodes
2      |
3      v  +---- (extent_node_t**)ret
4      +---+-----+
5      |  |          extent_node  |
6      | +-+-----+
7      | | v          rb_node      |
8      | | +-----+
9      | | | rbn_Left | rbn_right | | ... |
10     | | +-----+
11     | +-----+
12     +-----+
13         v
14 base_nodes----> +-----+
15                 | extent_node |
16                 +-----+

```

```

1  extent_node_t *
2  base_node_alloc(void)
3  {
4      extent_node_t *ret;
5
6      malloc_mutex_lock(&base_mtx);
7      if (base_nodes != NULL) {
8          ret = base_nodes;
9          // xf: 这里也可以理解为, base_nodes = (extent_node_t*)(*ret);
10         base_nodes = *(extent_node_t **)ret;
11         malloc_mutex_unlock(&base_mtx);
12     } else {
13         malloc_mutex_unlock(&base_mtx);
14         ret = (extent_node_t *)base_alloc(sizeof(extent_node_t));
15     }
16
17     return (ret);
18 }

```