


jemalloc源码解析-核心架构

 brionas.github.io/2015/01/31/jemalloc源码解析-核心架构

发布者: [Renjian Qiu](#)

引言:

jemalloc是一种通用的内存管理方法， 着重于减少内存碎片和支持可伸缩的并发性。jemalloc首次在FreeBSD中引入， 后续增加了heap profiling, Valgrind integration, and extensive monitoring/tuning hooks等功能， 目前在多个大型项目中都有应用。近期我们项目中也引入了jemalloc, 下面记录一下我对jemalloc的理解。

jemalloc的设计目标:

- 快速分配/释放内存， 最小化内存使用
- 在最小化内存的前提下， 尽可能保证内存分配的连续性， 减少内存碎片
- 好的线程扩展性
- 支持堆性能分析

在实现malloc的时候， 其设计者参考了一些已被验证的好的设计思想:

- 划分不同size大小的小对象， 减少内存碎片
- 合理选择size class的数量， 基于内部碎片和外部碎片的折中考虑
- 严格限制分配器元数据的开销（低于2%， 不包括碎片）
- 最小化活跃页面集合（减少内存交换）
- 最小化锁竞争（arena and thread cache）
- 如果不通用， 说明不够好（If it isn't general purpose, it isn't good enough）

系统架构:

jemalloc基于申请内存的大小把内存分配分为三个等级: small, large, huge.

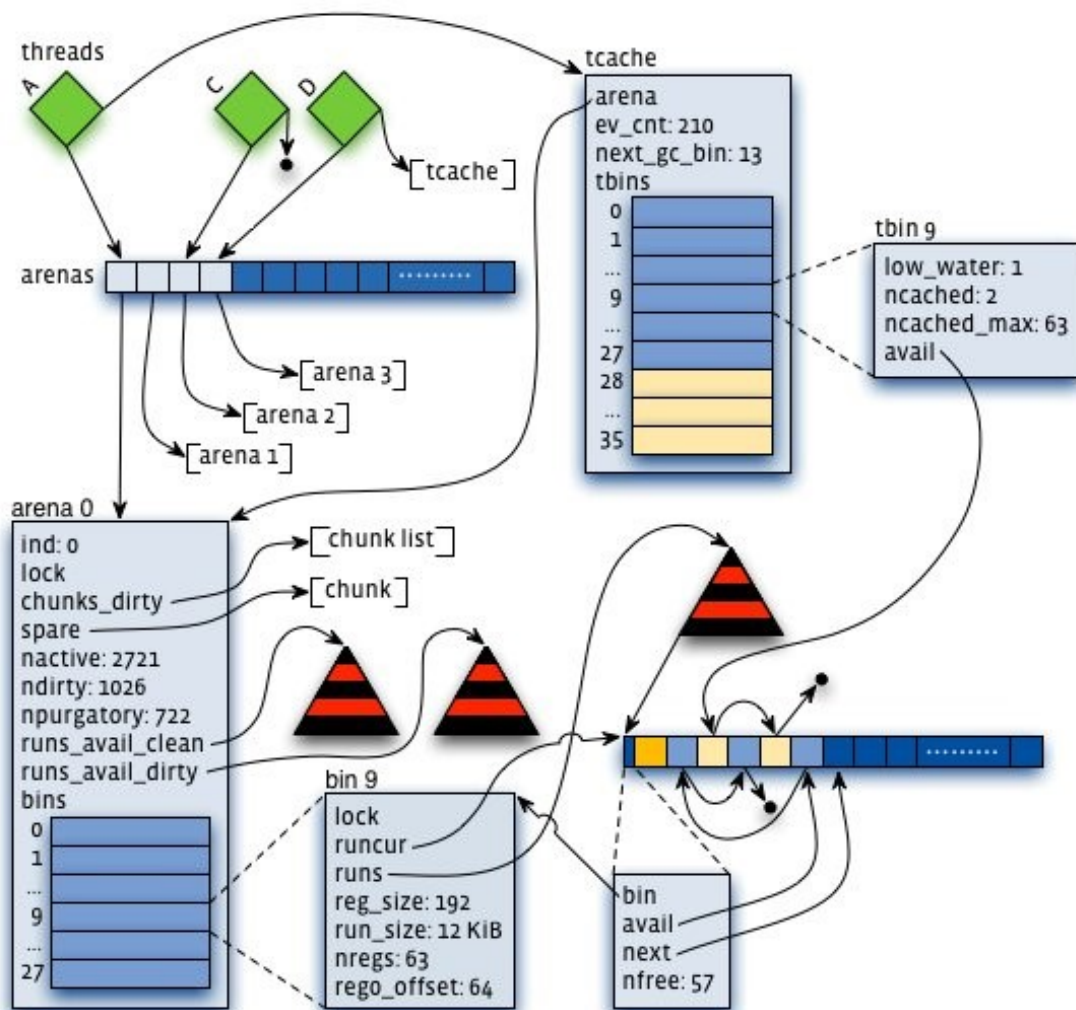
- Small objects的size以8字节, 16字节, 32字节等分隔开的, 小于页大小。
- Large objects的size以分页为单位, 等差间隔排列, 小于chunk的大小。
- Huge objects的大小是chunk大小的整数倍。

small objects和large objects由arena来管理, huge objects由线程间公用的红黑树管理。对于64位操作系统, 假设chunk大小为4M, 页大小为4K, 内存等级分配如下:

Category	Spacing	Size
Small	8	[8]
	16	[16, 32, 48, ..., 128]
	32	[160, 192, 224, 256]
	64	[320, 384, 448, 512]
	128	[640, 768, 896, 1024]
	256	[1280, 1536, 1792, 2048]

Category	Spacing	Size
	512	[2560, 3072, 3584]
Large	4 KiB	[4 KiB, 8 KiB, 12 KiB, ..., 4072 KiB]
Huge	4 MiB	[4 MiB, 8 MiB, 12 MiB, ...]

jemalloc的内存通过划分成等额大小的chunk进行管理， chunk的大小为2的k次方， 大于页大小。 Chunk的地址与chunk大小的整数倍对齐， 这样可以通过指针操作在常量时间内找到分配small/large objects的元数据， 在对数时间内定位到分配huge objects的元数据。 为了获得更好的线程扩展性， jemalloc采用多个arenas来管理内存， 减少了多个线程间的锁竞争。 每个线程独立管理自己的内存， 负责small和large的内存分配， 线程按第一次分配small或者large内存请求的顺序Round-Robin地选择arena。 并且从某个arena分配出去的内存块， 在释放的时候一定会回到该arena。 此外， jemalloc引入线程缓存来解决线程之间的同步问题， 通过对small和large对象的缓存， 实现通常情况下内存的快速申请和释放。 这里我们祭出jemalloc的架构图[2]， 虽然该图与当前版本的有一些出入， 但是整体框架保持不变， 图中架构主要涉及到几个主要的数据结构：arena/bin/run/chunk.， 接下来我们对其进行详细介绍。



核心结构：

Arena: 如上图所示，每个arena内都会包含对应的管理信息，记录该arena的分配情况。arena都有专属的chunks，每个chunk的头部都记录了chunk的分配信息。在使用某一个chunk的时候，会把它分割成很多个fun，并记录到bin中。不同size的class对应着不同的bin，在bin里都会有

个红黑树来维护空闲的run，并且在run里，使用了bitmap来记录了分配状态。此外，每个arena里面维护一组按地址排列的可获得的run的红黑树。

```

1. struct arena_s {
2.     ...
3.     /* 当前arena管理的dirty chunks */
4.     arena_chunk_tree_t  chunks_dirty;
5.     /* arena缓存的最近释放的chunk，每个arena一个spare chunk */
6.     arena_chunk_t      *spare;
7.     /* 当前arena中正在使用的page数。 */
8.     size_t              nactive;
9.     /*当前arena中未使用的dirty page数*/
10.    size_t              ndirty;
11.    /* 需要清理的page的大概数目 */
12.    size_t              npurgatory;
13.    /* 当前arena可获得的runs构成的红黑树， */
14.    /* 红黑树按大小/地址顺序进行排列。 分配run时采用first-best-fit策略*/
15.    arena_avail_tree_t  runs_avail;
16.    /* bins储存不同大小size的内存区域 */
17.    arena_bin_t        bins[NBINS];
18. };

```

Bin：前面已经提到，在arena中，不同bin管理不同size大小的run，在任意时刻，bin中会针对当前size保存一个run用于内存分配。

```

1. struct arena_bin_s {
2.     /* 作用域当前数据结构的锁*/
3.     malloc_mutex_t  lock;
4.     /* 当前正在使用的run */
5.     arena_run_t *runcur;
6.     /* 可用的run构成的红黑树， 主要用于runcur用完的时候。在查找可用run时，
7.      * 为保证对象紧凑分布，尽量从低地址开始查找，减少快要空闲的chunk的数量。
8.      */
9.     arena_run_tree_t runs;
10.    /* 用于bin统计 */
11.    malloc_bin_stats_t stats;
12. };

```

Run：每个bin在实际上是通过对它对应的正在运行的Run进行操作来进行分配的，一个run实际上就是chunk里的一块区域，大小是page的整数倍，在run的最开头会存储着这个run的信息，比如还有多少个块可供分配。下一块可分配区域的索引。

```

1. struct arena_run_s {
2.     /* 所属的bin */
3.     arena_bin_t *bin;
4.     /*下一块可分配区域的索引 */
5.     uint32_t    nextind;
6.     /* 当前run中空闲块数目。 */
7.     unsigned    nfree;
8. };

```

run中采用bitmap记录分配区域的状态，相比采用空闲列表的方式，采用bitmap具有以下优点：bitmap能够快速计算出第一块空闲区域，且能很好的保证已分配区域的紧凑型。分配数据与应用数据是隔离的，能够减少应用数据对分配数据的干扰。对很小的分配区域的支持更好。run的内存分配区域如下：

```

1. * Each run has the following layout:
2. *
3. *          /-----\
4. *          | arena_run_t header |
5. *          | ...                 |
6. * bitmap_offset | bitmap         |
7. *          | ...                 |
8. *   ctx0_offset | ctx map        |
9. *          | ...                 |
10. *          |-----|
11. *          | redzone             |
12. *   reg0_offset | region 0        |
13. *          | redzone             |
14. *          |-----| \
15. *          | redzone             | |
16. *          | region 1            | > reg_interval
17. *          | redzone             | /
18. *          |-----|
19. *          | ...                 |
20. *          | ...                 |
21. *          | ...                 |
22. *          |-----|
23. *          | redzone             |
24. *          | region nregs-1      |
25. *          | redzone             |
26. *          |-----|
27. *          | alignment pad?      |
28. *          \-----/

```

run的内存布局由arena_bin_info_s确定，与bin分开存储，与其他数据结构不同的是，arena_bin_info_s对于所有arena的不同size保持一份拷贝，而不是每个arena的不同size保存一份拷贝。这样既可以减少内存使用，又能避免缓存冲突。

Chunk: chunk是具体进行内存分配的区域，目前的默认大小是4M。chunk以page（默认为4K）为单位进行管理，每个chunk的前几个page（默认是6个）用于存储chunk的元数据，后面跟着一个或多个page的runs。后面的runs可以是未分配区域，多个小对象组合在一起组成run，其元数据放在run的头部。大对象构成的run，其元数据放在chunk的头部。

```

1. /* Arena chunk header. */
2. struct arena_chunk_s {
3.     /* 管理当前chunk的Arena */
4.     arena_t      *arena;
5.     /* 链接到所属arena的dirty chunks树的节点*/
6.     rb_node(arena_chunk_t)  dirty_link;
7.     /* 脏页数 */
8.     size_t       ndirty;
9.     /* 空闲run数 Number of available runs. */
10.    size_t       nruns_avail;
11.    /* 相邻的run数，清理的时候可以合并的run */
12.    size_t       nruns_adjac;
13.    /* 用来跟踪chunk使用状况的关于page的map，它的下标对应于run在chunk中的位置，通过加
        map_bias不跟踪chunk 头部的信息
14.     * 通过加map_bias不跟踪chunk 头部的信息
15.     */
16.    arena_chunk_map_t  map[1]; /* Dynamically sized. */
17. };

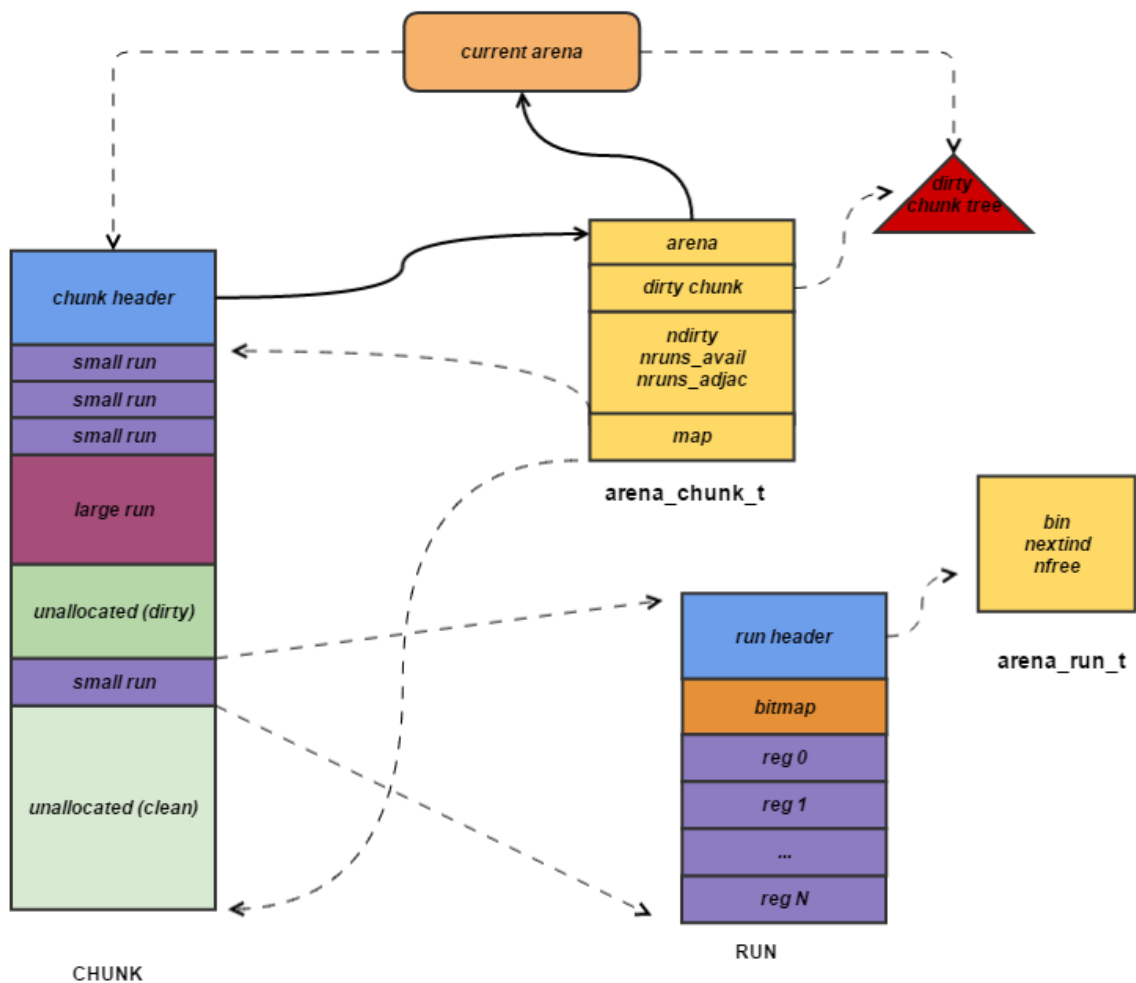
```

关于map[1]请参考如下code:

```
1. static arena_run_t *
2. arena_bin_runs_first(arena_bin_t *bin)
3. {
4.     /*通过bin的runs得到关于该run的描述信息mapelm*/
5.     arena_chunk_map_t *mapelm = arena_run_tree_first(&bin->runs);
6.     if (mapelm != NULL) {
7.         arena_chunk_t *chunk;
8.         size_t pageind;
9.         arena_run_t *run;
10.        /*通过红操作定位到对应的chunk*/
11.        chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(mapelm);
12.        /*通过地址计算得到run对应的pageid*/
13.        pageind = (((uintptr_t)mapelm - (uintptr_t)chunk->map) /
14.                   sizeof(arena_chunk_map_t)) + map_bias;
15.        /*通过地址运算得到对应的run*/
16.        run = (arena_run_t *)((uintptr_t)chunk + (uintptr_t)((pageind -
17.                   arena_mapbits_small_runind_get(chunk, pageind)) <<
18.                   LG_PAGE));
19.        return (run);
20.    }
21.    return (NULL);
22. }
```

Bin-Chunk-Run关系图:

可以看出, bin通过对run操作来实现对内存区域chunk的管理, 他们之间的关系如图所示:



tcache: tcache为线程对应的私有缓存空间， 用于减少线程分配内存时锁的争用， 提高内存分配的效率。如果使用tcache时, jemalloc分配内存时将优先从tcache中分配， 只有在tcache中找不到才进入正常的分配流程。

```

1. JEMALLOC_ALWAYS_INLINE void *
2. arena_malloc(arena_t *arena, size_t size, bool zero, bool try_tcache)
3. {
4.     tcache_t *tcache;
5.     assert(size != 0);
6.     assert(size <= arena_maxclass);
7.     /*分配small object*/
8.     if (size <= SMALL_MAXCLASS) {
9.         if (try_tcache && (tcache = tcache_get(true)) != NULL)
10.            return (tcache_alloc_small(tcache, size, zero));
11.         else {
12.             return (arena_malloc_small(choose_arena(arena), size,
13.                 zero));
14.         }
15.     } else {
16.         /*
17.          * Initialize tcache after checking size in order to avoid
18.          * infinite recursion during tcache initialization.
19.          */
20.         /*分配large object*/
21.         if (try_tcache && size <= tcache_maxclass && (tcache =
22.             tcache_get(true)) != NULL)
23.             return (tcache_alloc_large(tcache, size, zero));
24.         else {
25.             return (arena_malloc_large(choose_arena(arena), size,
26.                 zero));
27.         }
28.     }
29. }

```

每个tcache也有一个对应的arena, 这个arena内部也包含一个tbin数组来缓存不同大小的内存块, 与arena中的bin对应, 只是长度更大一些, 因为它需要缓存更大的内存块, tcache中对象的缓存, 没有对应的run的概念。

```

1. struct tcache_bin_s {
2.     ...
3.     unsigned    ncached;    /* 缓存对象数目. */
4.     void        **avail;    /*缓存对象栈. */
5. };
6. struct tcache_s {
7.     ...
8.     arena_t      *arena;    /* 线程对应arena. */
9.     unsigned     ev_cnt;    /* GC事件数 */
10.    unsigned     next_gc_bin; /* 下一个需要GC的bin. */
11.    tcache_bin_t  tbins[1];  /* 表示不同size的缓存 */
12. };

```

本文主要对jemalloc的核心架构进行介绍, 便于读者对jemalloc的内存框架有一个基本认识, 下文将对jemalloc内存管理进行详细介绍。