2.4 Run (arena_run_t)

如同在2.1节所述, 在jemalloc中**run才是真正负责分配的主体**(前提是对small region来说). run的大小对齐到page size上, 并且在内部划分成大小相同的region. 当有外部分配请求时, run就会从内部挑选一个free region返回. 可以认为run就是small region仓库.

> **注：**run的大小就是其内部region的大小。假设run大小为4KB，则可以把run看成是4KB的region的仓库，真正分配给user memory的是其内部的region。

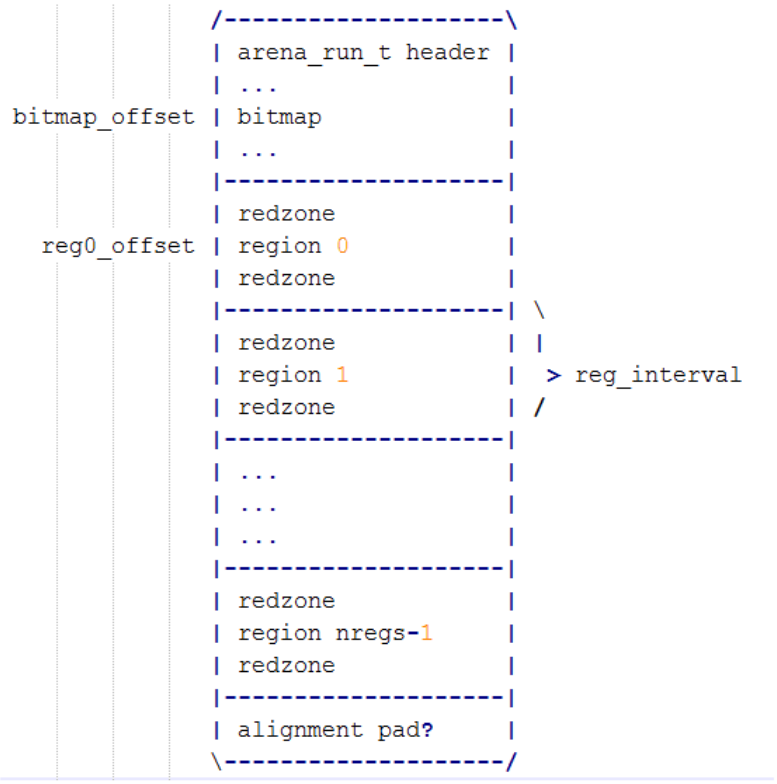
2.4.1 Run结构

```
struct arena_run_s {
    arena_bin_t    *bin;
    uint32_t        nextind;
    unsigned        nfree;
};
```

run的结构非常简单, 但同chunk类似, 所谓的arena_run_t不过是整个run的header部分.

- **bin:** 与该run相关联的bin. 每个run都有其所属的bin, 详细内容在之后介绍.
- **nextind:** 记录下一个clean region的索引.
- **nfree:** 记录当前空闲region数量.

除了header部分之外, run的真实layout如下,



正如chunk通过chunk map记录内部所有page状态一样, run通过在header后挂载bitmap来记录其内部的region状态. bitmap之后是regions区域. 内部region大小相等, 且在前后都有redzone保护(需要在设置里打开redzone选项).

简单来说, r**un就是通过查询bitmap来找到可用的region. 而传统分配器由于使用boundary tag, 空闲region一般是被双向链表管理的**. 相比之下, 传统方式查找速度更快, 也更简单. 缺点之前也提到过, 安全和稳定性都存在缺陷. 从这一点可以看到, jemalloc在设计思路上将bookkeeping和user memory分离是贯穿始终的原则,更甚于对性能的影响(事实上这点影响在并发条件下被大大赚回来了).

2.4.2 size classes

内存分配器对内部管理的region往往按照某种特殊规律来分配. 比如jemalloc将内存划分成small和large两种类型. small类型从8字节开始每8个字节为一个分割直至256字节,而large类型则从256字节开始, 挂载到dst上. 这种划分方式有助于分配器对内存进行有效的管理和控制.

让已分配的内存更加紧实(tightly packed), 以降低外部碎片率.

jemalloc进一步优化了分配效率. 采用了类似于"二分伙伴(Binary Buddy)算法"的分配方式. **在jemalloc中将不同大小的类型称为size class**

在jemalloc源码的size_classes.h文件中, 定义了不同体系架构下的region size. 该文件实际是通过名为size_classes.sh的shell script自动生成的. script按照四种不同量纲定义来区分各个体系平台的区别, 然后将它们做排列组合, 就可以兼容各个体系.这四种量纲分别是,

- **LG_SIZEOF_PTR:** 代表指针长度, ILP32下是2, LP64则是3.
- **LG_QUANTUM:** 量子, binary buddy分得的最小单位. 除了tiny size, 其他的size classes都是quantum的整数倍大小.
- **LG_TINY_MIN:** 是比quantum更小的size class, 且必须对齐到2的指数倍上. 它是jemalloc可分配的最小的size class.
- **LG_PAGE:** 就是page大小

根据binary buddy算法, jemalloc会将内存不断的二平分, 每一份称作一个group. 同一个group内又做四等分. 例如, 一个典型的ILP32, tiny等于8byte, quantum为16byte,page为4096byte的系统, 其size classes划分是这样的,

```
#if (LG_SIZEOF_PTR == 2 && LG_TINY_MIN == 3 && LG_QUANTUM == 4 && LG_PAGE == 12)
#define    SIZE_CLASSES \
      index, lg_grp, lg_delta, ndelta,  bin, lg_delta_lookup  \
    SC(  0,      3,        3,      0,   yes,        3) \
                                                      \
    SC(  1,      3,        3,      1,   yes,        3) \
    SC(  2,      4,        4,      1,   yes,        4) \
    SC(  3,      4,        4,      2,   yes,        4) \
    SC(  4,      4,        4,      3,   yes,        4) \
                                                      \
    SC(  5,      6,        4,      1,   yes,        4) \
    SC(  6,      6,        4,      2,   yes,        4) \
    SC(  7,      6,        4,      3,   yes,        4) \
    SC(  8,      6,        4,      4,   yes,        4) \

```

宏SIZE_CLASSES主要功能就是可以生成几种类型的table. 而SC则根据不同的情况被定义成不同的含义. SC传入的6个参数的含义如下,

- **index:** 在table中的位置。
- **lg_grp:** 所在group的指数。
- **lg_delta:** group内偏移量指数。
- **ndelta:** group内偏移数。
- **bin:** 是否由bin记录. small region是记录在bins中。
- **lg_delta_lookup:** 在lookup table中的调用S2B_#的尾数后缀。

因此得到reg_size的计算公式, reg_size = 1 << lg_grp + ndelta << lg_delta根据该公式, 可以得到region的范围,

Category	Subcategory	Spacing	Size
Smail	Tiny	lg	[8]
	Quantum	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]

		512	[2560, 3072, 3584]
	Page	1KiB	[1 KiB, 2 KiB]
Large		4KiB	[4 KiB, 8 KiB, 12 KiB, ..., 4072 KiB]
Huge		4MiB	[4 MiB, 8 MiB, 12 MiB, ...]

除此之外, 在size_classes.h中还定义了一些常量,

tiny bins的数量

```
#define    NTBINS            1
```

可以通过lookup table查询的bins数量

```
#define    NLBINS            29
```

small bins的数量

```
#define    NBINS             28
```

最大tiny size class的指数

```
#define    LG_TINY_MAXCLASS    3
```

最大lookup size class, 也就是NLBINS - 1个bins

```
#define    LOOKUP_MAXCLASS    ((((size_t)1) << 11) + (((size_t)4) << 9))
```

最大small size class, 也就是NBINS - 1个bins

```
#define    SMALL_MAXCLASS    ((((size_t)1) << 11) + (((size_t)3) << 9))
```

2.4.3 size2bin/bin2size

通过SIZE_CLASSES建立的table就是为了在O(1)的时间复杂度内快速进行size2bin或者bin2size切换. 同样的技术在dlmalloc中有所体现, 来看jemalloc是如何实现的.

size2bin切换提供了两种方式, 较快的是通过查询lookup table, 较慢的是计算得到.从原理上, 只有small size class需要查找bins, 但可通过lookup查询的size class数量要小于整个small size class数量. 因此, 部分size class只能计算得到. 在原始jemalloc中统一只采用查表法, 但在android版本中可能是考虑减小lookup table size, 而增加了直接计算法.

```
JEMALLOC_ALWAYS_INLINE size_t
small_size2bin(size_t size)
{
    ......
    if (size <= LOOKUP_MAXCLASS)
        return (small_size2bin_lookup(size));
    else
        return (small_size2bin_compute(size));
}
```

小于LOOKUP_MAXCLASS的请求通过small_size2bin_lookup直接查表. 查询的算法是这样的,

```
size_t ret = ((size_t)(small_size2bin_tab[(size-1) >> LG_TINY_MIN]));
```

也就是说, jemalloc通过一个

```
f(x) = (x - 1) / 2^LG_TINY_MIN
```

的变换将size映射到lookup table的相应区域. 这个table在gdb中可能是这样的,

```
(gdb) p  /d small_size2bin
$6 = {0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 9, 9, 10, 10, 10, 10,
      11, 11, 11, 11, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14,
      14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16, 16, 16, 16,
      16, 16, 17 <repeats 16 times>, 18 <repeats 16 times>, 19 <repeats 16 times>,
      20 <repeats 16 times>, 21 <repeats 32 times>, 22 <repeats 32 times>,
      23 <repeats 32 times>, 24 <repeats 32 times>, 25 <repeats 64 times>,
      26 <repeats 64 times>, 27 <repeats 64 times>}
```

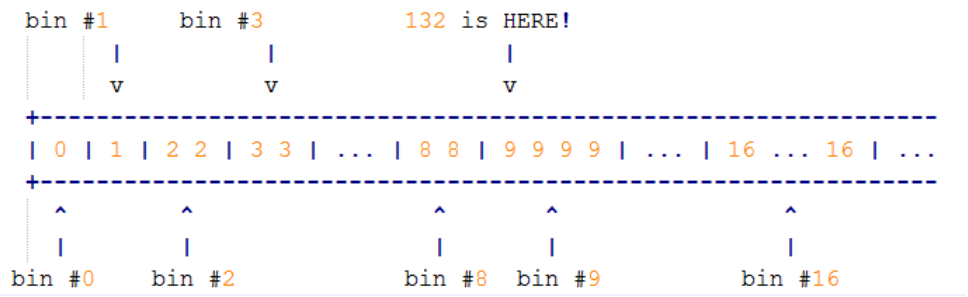
该数组的含义与binary buddy算法是一致的. 对应的bin index越高, 其在数组中占用的字节数就越多. 除了0号bin之外, 相邻的4个bin属于同一group, 两个group之间相差二倍,因此在数组中占用的字节数也就相差2倍. 所以, 上面数组的group划分如下,

```
{0}, {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}, ...
```

以bin#9为例, 其所管辖的范围(128, 160], 由于其位于更高一级group, 因此相比bin#8 在lookup table中多一倍的字节数, 假设我们需要查询132, 经过映射,

```
(132 - 1) >> 3 = 16
```

这样可以快速得到其所在的bin #9. 如图,



jemalloc巧妙的通过前面介绍CLASS_SIZE宏生成了这个lookup table, 代码如下,

```
JEMALLOC_ALIGNED(CACHELINE)
const uint8_t    small_size2bin_tab[] = {
#define    S2B_3(i)    i,
#define    S2B_4(i)    S2B_3(i) S2B_3(i)
#define    S2B_5(i)    S2B_4(i) S2B_4(i)
#define    S2B_6(i)    S2B_5(i) S2B_5(i)
#define    S2B_7(i)    S2B_6(i) S2B_6(i)
#define    S2B_8(i)    S2B_7(i) S2B_7(i)
#define    S2B_9(i)    S2B_8(i) S2B_8(i)
#define    S2B_no(i)
#define    SC(index, lg_grp, lg_delta, ndelta, bin, lg_delta_lookup) \
    S2B_##lg_delta_lookup(index)
    SIZE_CLASSES
#undef S2B_3
#undef S2B_4
```

这里的S2B_xx是一系列宏的嵌套展开, 最终对应的就是不同group在lookup table中占据的字节数以及bin索引. 相信看懂了前面的介绍就不难理解.

另一方面, 大于LOOKUP_MAXCLASS但小于SMALL_MAXCLASS的size class不能查表获得, 需要进行计算. 简言之, 一个bin number是三部分组成的,

```
bin_number = NTBIN + group_number << LG_SIZE_CLASS_GROUP + mod
```

即tiny bin数量加上其所在group再加上group中的偏移(0-2). 源码如下,

```
JEMALLOC_INLINE size_t
```

```
small_size2bin_compute(size_t size)
{
    ......
    {
        // xf: lg_floor相当于ffs
        size_t x = lg_floor((size<<1)-1);

        // xf: 计算size class所在group number
        size_t shift = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM) ? 0 :
            x - (LG_SIZE_CLASS_GROUP + LG_QUANTUM);
        size_t grp = shift << LG_SIZE_CLASS_GROUP;

        size_t lg_delta = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM + 1)
            ? LG_QUANTUM : x - LG_SIZE_CLASS_GROUP -
```

其中LG_SIZE_CLASS_GROUP是size_classes.h中的定值, 代表一个group中包含的bin数量, 根据binary buddy算法, 该值通常情况下是2

而要找到size class所在的group, 与其最高有效位相关. jemalloc通过类似于ffs的函数首先获得size的最高有效位x,

```
size_t x = lg_floor((size<<1)-1);
```

至于group number, 则与quantum size有关. 因为除了tiny class, quantum size位于group #0的第一个. 因此不难推出,

```
group_number = 2^x / quantum_size / 2^LG_SIZE_CLASS_GROUP
```

对应代码就是,

```
size_t shift = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM) ? 0 :
            x - (LG_SIZE_CLASS_GROUP + LG_QUANTUM);
```

而对应group起始位置就是,

```
size_t grp = shift << LG_SIZE_CLASS_GROUP;
```

至于mod部分, 与之相关的是最高有效位之后的两个bit.

jemalloc在这里经过了复杂的位变换,

```
size_t lg_delta = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM + 1)
    ? LG_QUANTUM : x - LG_SIZE_CLASS_GROUP - 1;
size_t delta_inverse_mask = ZI(-1) << lg_delta;
size_t mod = ((((size-1) & delta_inverse_mask) >> lg_delta)) &
    ((ZU(1) << LG_SIZE_CLASS_GROUP) - 1);
```

上面代码直白的翻译, 实际上就是要求得如下两个bit,

```
                    1 0000
                   10 0000
                   11 0000
group #0          100 0000
-----------------------------------------------
                                      +--+
                  101 0000 - 1 = 1|00| 1111
                  110 0000 - 1 = 1|01| 1111
                  111 0000 - 1 = 1|10| 1111
group #1         1000 0000 - 1 = 1|11| 1111
                                      +--+
-----------------------------------------------
                                      +--+
                 1010 0000 - 1 = 1|00|1 1111
                 1100 0000 - 1 = 1|01|1 1111
                 1110 0000 - 1 = 1|10|1 1111
group #2        10000 0000 - 1 = 1|11|1 1111
```

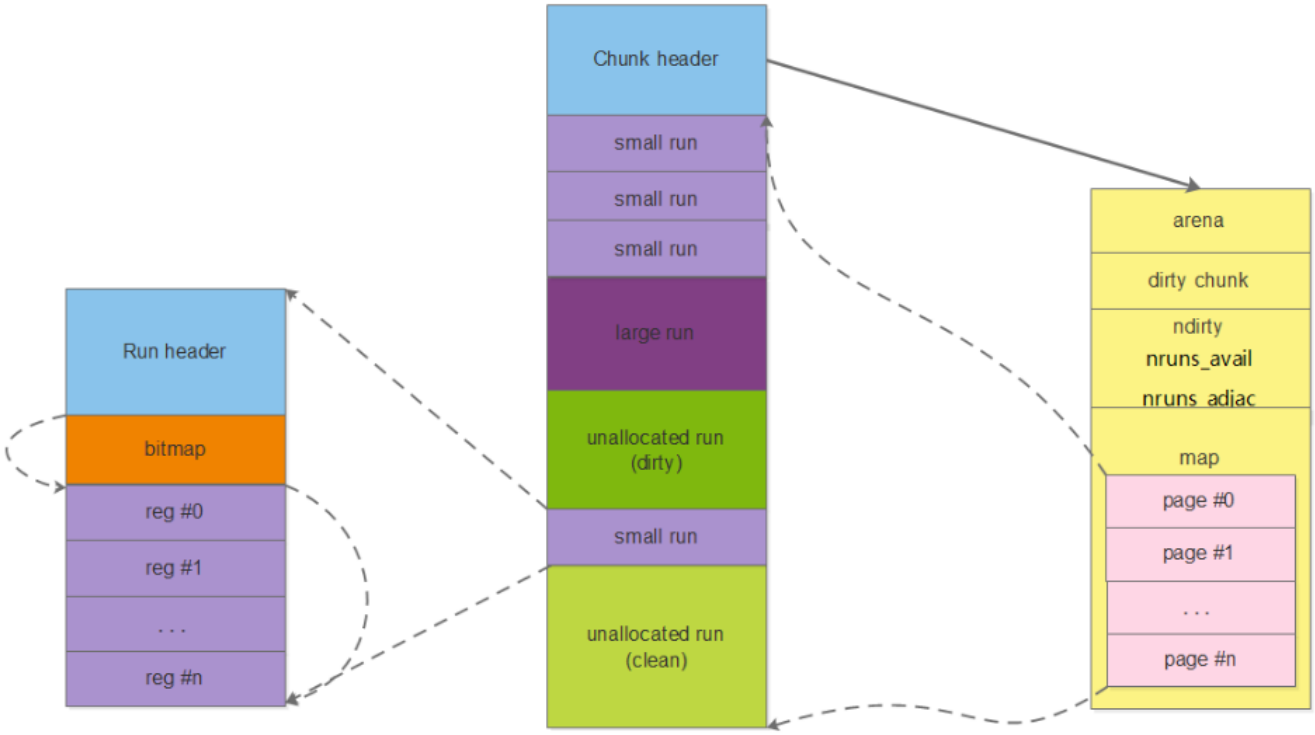
```
                                                                +--+
-------------------------------------------------------------
```

根据这个图示再去看jemalloc的代码就不难理解了. mod的计算结果就是从0-3的数值.
而最终的结果是前面三部分的组合即,

```
1  size_t bin = NTBINS + grp + mod;
```

而bin2size查询就简单得多. 上一节介绍SIZE_CLASSES时提到过small region的计算公式, 只需要根据该公式提前计算出所有bin对应的region size, 直接查表即可.这里不再赘述.

2.5 Chunk和run关系图



2.6 bins (arena_bin_t)

run是分配的执行者, 而分配的调度者是bin. 这个概念同dlmalloc中的bin是类似的, 但jemalloc中bin要更复杂一些. 直白地说, 可以把bin看作non-full run的仓库, **bin负责记录当前arena中某一个size class范围内所有non-full run的使用情况**. 当有分配请求时, arena查找相应size class的bin, 找出可用于分配的run, 再由run分配region. 当然, 因为只有small region分配需要run, 所以**bin也只对应small size class.**

与bin相关的数据结构主要有两个, 分别是arena_bin_t和arena_bin_info_t.在2.1.3中提到arena_t内部保存了一个bin数组, 其中的成员就是arena_bin_t.

其结构如下,

```
1  struct arena_bin_s {
2      malloc_mutex_t          lock;
3      arena_run_t             *runcur;
4      arena_run_tree_t        runs;
5      malloc_bin_stats_t      stats;
6  };
```

- **lock:** 该lock同arena内部的lock不同, 主要负责保护current run. 而对于run本身的分配和释放还是需要依赖arena lock. 通常情况下, 获得bin lock的前提是获得arena lock, 但反之不成立.
- **runcur:** 当前可用于分配的run, 一般情况下指向地址最低的non-full run, 同一时间一个bin只有一个current run用于分配.
- **runs:** rb tree, 记录当前arena中该bin对应size class的所有non-full runs. 因为分配是通过current run完成的, 所以也相当于current run的仓库.

- **stats:** 统计信息.

另一个与bin相关的结构是arena_bin_info_t. 与前者不同, bin_info保存的是arena_bin_t的静态信息, 包括相对应size class run的bitmap offset, region size, region number, bitmap info等等(此类信息只要class size决定, 就固定下来). 所有上述信息在jemalloc中由全局数组arena_bin_info记录. 因此与arena无关, 全局仅保留一份.

arena_bin_info_t的定义如下,

```
struct arena_bin_info_s {
    size_t             reg_size;
    size_t             redzone_size;
    size_t             reg_interval;
    size_t             run_size;
    uint32_t           nregs;
    uint32_t           bitmap_offset;
    bitmap_info_t     bitmap_info;
    uint32_t           reg0_offset;
};
```

- **reg_size:** 与当前bin的size class相关联的region size.
- **reg_interval:** reg_size+redzone_size.
- **run_size:** 当前bin的size class相关联的run size.
- **nregs:** 当前bin的size class相关联的run中region数量.
- **bitmap_offset:** 当前bin的size class相关联的run中bitmap偏移.
- **bitmap_info:** 记录当前bin的size class相关联的run中bitmap信息.
- **reg0_offset:** index为0的region在run中的偏移量.

以上记录的静态信息中尤为重要的是bitmap_info和bitmap_offset.

其中bitmap_info_t定义如下,

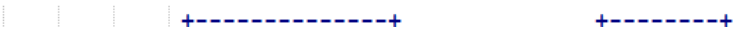
```
struct bitmap_info_s {
    size_t nbits;
    unsigned nlevels;
    bitmap_level_t levels[BITMAP_MAX_LEVELS+1];
};
```

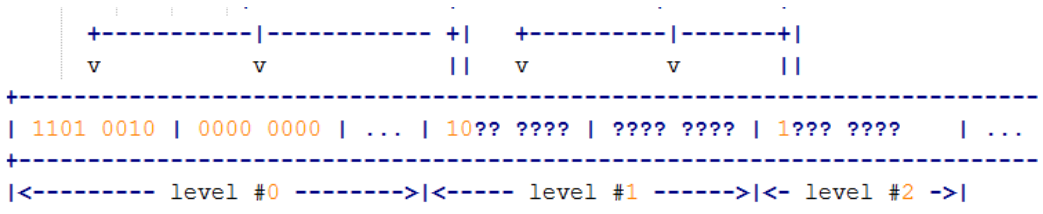
- **nbits:** bitmap中逻辑bit位数量(特指level#0的bit数).
- **nlevels:** bitmap的level数量.
- **levels:** level偏移量数组, 每一项记录该级level在bitmap中的起始index.

```
struct bitmap_level_s {
    size_t group_offset;
};
```

在2.3.1节中介绍arena_run_t时曾提到jemalloc通过外挂bitmap将bookkeeping和user memory分离. 但bitmap查询速度要慢于boundary tag. 为了弥补这个缺陷, jemalloc对此做了改进, 通过多级level缓冲以替代线性查找.

jemalloc为bitmap增加了多级level, bottom level同普通bitmap一致, 每1bit代表一个region.而高一级level中1bit代表前一级level中一个byte. 譬如说, 若我们在当前run中存在128个region, 则在ILP32系统上, 需要128/32 = 4byte来表示这128个region. jemalloc将这4个byte看作level #0. 为了进一步表示这4个字节是否被占用, 又额外需要1byte以缓存这4byte的内容(仅使用了4bit), 此为level#1. 即整个bitmap, 一共有2级level, 共5byte来描述.




| 1101 0010 | 0000 0000 | ... | 10?? ???? | ???? ???? | 1??? ????
|<--------- level #0 -------->|<----- level #1 ------>|<- level #2 ->|