Libhaisqlmalloc

高性能内存分配库

V2.91

用

户

手

册

# 引言

## 1.编写目的

为了帮助用户更好地了解和使用该软件,提高用户与软件的亲和度。该用户手册讲述怎样安装和使用高性能内存分配库，该系统使用过程中的配置，以及应注意的一些问题。

## 2.背景

a．该软件系统的名称：高性能内存分配库

b．该软件项目开发：乌鲁木齐云山云海信息技术有限责任公司

云奈（北京）科技发展有限责任公司深圳分公司

c．该软件的用户（或首批用户）：企业用户

## 3.参考资料

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 资料名称[标识符] | 出 版 单 位 | 作 者 | | 日 期 |
| tcmalloc谷歌内存分配库 | 谷歌 | |  |  |
| Jemalloc Facebook内存分配库 | Facebook | |  |  |
| Linux内存分配库 | Linux | |  |  |

# 软件概述

## 1.描述

Libhaisqlmalloc高性能内存分配库，简称“高性能内存分配库”，是由乌鲁木齐云山云海信息技术有限责任公司和云奈（北京）科技发展有限责任公司深圳分公司联合开发的内存分配库，是一种新型高性能内存分配库，管理堆内存，主要接口是malloc和free，用于减慢频繁分配、释放内存造成的性能损耗，并且有效地控制内存浪费和内存碎片，同时兼具高性能和高稳定性，更适合大型应用程序的配套使用。

Libhaisqlmalloc高性能内存分配库开发语言是C++,编译链工具是gcc，软件源码总行数大约5万行。

第三章 软硬件环境和使用方法

## 1.运行硬件环境

（1）、内存：512M及以上（建议1G以上）

（2）、CPU：主频1 GMHz以上（建议Intel Core2 以上）

## 2.运行软件环境

建议使用：Linux系统4.0 以上64bit版本 x86\_64 GNU/Linux

对于低于Linux系统3.X 64bit版本, 可能会存在Hugepage无法使用的问题。

## 3.使用方法：

Libhaisqlmalloc 内存分配库的使用有两种方法：

1. 设置LD\_PRELOAD=带路径的libhaisqlmalloc.so，

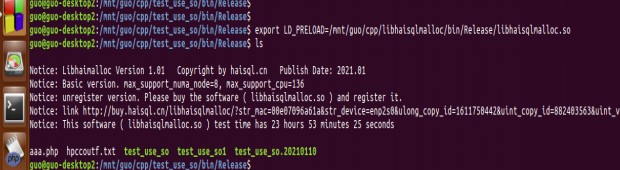
例如执行：export LD\_PRELOAD=/mnt/guo/libhaisqlmalloc.so，这种方法很简单不需重新编译程序，并且可以随时切换回系统默认的内存库， 或者使用其他内存库。如果需要默认自动配置，可以在配置默认环境变量配置终端启动时自动执行一些指令, 可以在 ~/.bashrc, ~/.bash\_profile, ~/.profile, 或者 ~/.zshrc 文件添加这些指令。Ubuntu系统修改的~/.bashrc，增加一行export LD\_PRELOAD=带路径的libhaisqlmalloc.so。

2）将libhaisqlmalloc库链接到程序中，注意应该将haisqlmalloc库后链接到程序中；

在gcc 编译器的链接中使用参数 -L路径（libhaisqlmalloc.so所在的路径） -lhaisqlmalloc

建议使用前一种方法，这种方式简单并且可以根据需要随时切换内存分配库。

下面是使用Libhaisqlmalloc库的一个范例：



以上可以看到，使用export LD\_PRELOAD配置后，就已经自动使用Libhaisqlmalloc库了。

使用Libhaisqlmalloc库将会显示5行Notice：

第1行：提供了版权信息，

第2行：提供了版本支持信息，基本版本，支持的NUMA node数量，默认为8,一般的机器只有一个NUMA node，中高端服务器一般也只有8个以下的NUMA node。一般的机器均在64个CPU以下，只有高端服务器才会有64个以上的CPU, 此时就需要联系软件开发商索要ADVANCE版本, 以便支持更多的CPU。

第3行：表示这是一个未注册的测试版本，需要购买。

第4行：提供了一个购买合法版本的链接，购买后将提供一个libhaisqlmalloc.conf的配置文件，将这个配置文件拷贝到/etc/libhaisqlmalloc.conf下或者拷贝到当前libhaisqlmalloc.so文件所在的目录下均可以实现软件许可证License的正确配置，变为合法用户。

第5行：提供了libhaisqlmalloc.so文件还能被测试多久，剩余的测试时间。默认的测试时间是100天。

注册成功后，不会显示以上信息。可以通过调用haisql\_malloc\_version()函数，显示版权信息。

# 第四章 配置文件和配置选项

外部配置文件：

默认的配置文件1是： 当前目录libhaisqlmalloc.conf

默认的配置文件2是： /etc/libhaisqlmalloc.conf

默认的配置文件3是： 当前libhaisqlmalloc.so文件所在的目录下的libhaisqlmalloc.conf文件

在Libhaisqlmalloc 库的使用过程中，将会依次使用配置文件1和配置文件2,获取配置信息。如果配置文件1存在，将不读取配置文件2。如果两个配置文件均不存在，那么就按照默认值执行。

每个配置参数即可以通过conf文件来修改，也可以通过调用函数mallopt( int int\_key, int int\_value ) 来修改定义。其中int\_value是设置的值。

Libhaisqlmalloc 库的key值范围是10000 ---10058。

uint\_mpmc\_giantpage\_2m\_min=100

uint\_mpmc\_giantpage\_2m\_max=128

uint\_mpmc\_giantpage\_2m\_force\_update\_count=16

bool\_only\_bind\_pcore=0

uint\_bind\_cpu\_begin=99999

uint\_bind\_cpu\_pcore\_ecore\_multi=2

uint\_thread\_stack\_default\_size=2097152

bool\_enable\_mmap\_populate=1

bool\_show\_numa\_map\_on\_exit=0

uint\_main\_thread\_bind\_cpu=99999

bool\_show\_config\_on\_exit=1

bool\_force\_enable\_giantpage=0

uint\_thread\_background\_check\_release\_thread\_millisecond=200

bool\_hook\_pthread\_lock\_check\_sync=0

bool\_show\_thread\_cpu\_bind=1

uint\_giantpage\_malloc\_fail\_wait\_microsecond=100000

uint\_hugepage\_malloc\_fail\_wait\_microsecond=10000

uint\_fork\_thread\_bind\_cpu=99999

bool\_enable\_hugepage\_bind\_numa=0

bool\_show\_hugepage\_bind\_numa=0

bool\_use\_hugepage\_other\_numa=1

bool\_enable\_giantpage\_bind\_numa=0

bool\_show\_giantpage\_bind\_numa=0

bool\_use\_giantpage\_other\_numa=1

uint\_string\_size\_div8\_multi=10

uint\_vector\_size\_div8\_multi=12

uint\_giantpage\_delay\_unmap\_millisecond=4000

uint\_sleep\_microsecond\_on\_exit=20000

bool\_force\_main\_thread\_auto\_bind\_cpu=0

bool\_force\_fork\_thread\_auto\_bind\_cpu=0

uint\_create\_thread\_sleep\_microsecond=0

uint\_stack\_block\_delay\_free\_microsecond=256000

str\_hex\_bind\_cpu\_mask=FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

配置参数范例：

uint\_microsecond\_thread\_background\_sleep=100

uint\_wait\_microsecond\_between\_release\_2m=4000

bool\_enable\_auto\_bind\_cpu=true

下列扩展函数，可以用于当前显示配置参数的设置情况。

void malloc\_show\_config( void ) noexcept;

配置参数介绍如下：

uint\_microsecond\_thread\_background\_sleep

类型：无符号整数

默认值：1000

意思：默认值是1000, 表示后台线程每1000微秒（1毫秒）会主动唤醒一次，执行后台任务。实际后台线程在工作中，每次探测到提前准备2M大块的需求，将下次唤醒时间周期减少一半直到50微秒，否则恢复使用配置值。

范围：大于等于50， 低于50的配置将自动修改为50

mallopt int\_key=10000

uint\_wait\_microsecond\_between\_release\_2m

类型：无符号整数

默认值：4000

意思：默认值是4000, 表示后台线程每4000微秒可以释放1个2M块，这样每秒允许释放512M内存， 这个是一个快速释放的配置参数，类似的uint\_microsecond\_thread\_background\_force\_release\_one\_2m是一个慢速释放的配置参数。

这个配置参数定义了后台线程的内存的释放速度。由于延迟释放，提高了重复使用待释放内存的概率，因此提高了性能。

mallopt int\_key=10001

bool\_enable\_thread\_background\_clear\_cpu\_cache

类型：Boolean

意思：此参数已经作废。见参数uint\_max\_wait\_nanosecond\_before\_clear\_cpu\_cache的说明。

bool\_enable\_hugepage

类型：Boolean

默认值：true

意思：是否启用 hugepage2m， 这里是hugepage2m的总开关。在下列场景下，将试图启用hugepage2m，第1个场景是应用程序进入初始化的最早阶段(内存库so自身初始化完毕前)，内存分配总量就已经超过 uint\_use\_sbrk\_byte\_threshold\_force\_page\_prefault，例如 geany / codeblocks 这样的大型程序启动时会触发此条件，说明这个程序是一个大型应用软件，可以从一开始就尝试立即使用hugepage，这样如果配置有足够的hugepage，那么就可以在后续流程中100%使用hugepage，会有更高的应用程序性能。 第2个场景是库自身初始化完成后，将尝试启用hugepage2m，此前内存库已经申请的2个2M内存块（其中一个用于meta内部管理），是常规的4KB页面的内存，并且没有预先分配物理页面，这样对于一些使用内存比较少的场景下，可以节约物理内存的使用总量。

mallopt int\_key=10003

uint\_size\_use\_thread\_background\_threshold

类型：无符号32bit整数

默认值：8192000

意思：申请内存总量超过配置值（默认是8M） 将会尝试启用后台线程

mallopt int\_key=10004

bool\_enable\_thread\_background

类型：Boolean

默认值：true

意思：是否启用 后台线程， 这里是后台线程总开关。有两种情况下，将尝试启用后台线程，一种是申请内存总量超过配置值(见uint\_size\_use\_thread\_background\_threshold的说明)，另外一种是探测到创建了额外工作线程， 也会自动尝试启用后台线程。

mallopt int\_key=10005

bool\_enable\_auto\_bind\_cpu

类型：Boolean

默认值：true

意思：是否启用＂工作线程自动绑定CPU＂．此功能启用后，将会在线程申请内存时自动检查是否已经绑定了CPU，如果没有绑定，将自动执行线程自动绑定CPU：

首先探测CPU 是否支持超线程，如果是支持超线程的CPU, 将自动将工作线程绑定到一个物理CPU上的2个逻辑CPU上（这种在同一个物理CPU上的在逻辑CPU上的线程切换不会造成Cache颠簸，具备良好的线程切换性能）。

如果CPU不支持超线程，将自动将工作线程绑定到同一NUMA node下的1--2个相邻的物理CPU上。（见后面配置参数uint\_auto\_bind\_cpu\_count）。

uint\_bind\_cpu\_begin 用于设置绑定CPU的起始值。

uint\_bind\_cpu\_pcore\_ecore\_multi 用于线程设置绑定CPU大核小核的使用比例。

uint\_main\_thread\_bind\_cpu 用于设置主线程绑定CPU。

bool\_only\_bind\_pcore 用于设置是否线程只绑定大核CPU。

mallopt int\_key=10006

bool\_enable\_numa\_memory\_bind

类型：Boolean

默认值：false

意思：是否启用NUMA node内存绑定，目前只有中高端服务器才支持NUMA, 传统PC和低端服务器（单一CPU槽位的服务器）不支持NUMA。目前默认是关闭的，打开此选项，将会在支持NUMA的服务器上启用按工作线程的所在NUMA node优先的内存分配策略，带来性能增加。

mallopt int\_key=10007

uint\_vt\_thread\_local\_wait\_reuse\_limit\_size

类型：无符号整数

默认值：8

意思： 限制用于保存备用thread\_local的池子大小的限制大小．每个线程都有一个thread local用于分配当前线程的内存申请，即当线程退出时，如果备用thread\_local池的空间超过限制大小, 并且该thread\_local相关的所有malloc指针都已经释放，那么这个thread\_local数据结构将执行析构。否则，这个数据结构将被缓存下来用于重用，在下次新的线程创建后，将优先复用这些备用的thread\_local数据，

这种pool池减少了大量线程反复创建退出，一堆thread\_local内部相关对象反复创建和析构带来执行开销浪费。

mallopt int\_key=10008

bool\_enable\_page\_prefault

类型：Boolean

默认值：false

意思：申请内存时是否实际分配4KiB page页面的物理地址，这个参数是一个动态变量，即程序会随着程序的进展自动修改此参数，也允许用户配置修改此变量，在下列场景下，此参数将会被自动修改为true：场景1是应用程序进入初始化的最早阶段(内存库so自身初始化完毕前)，内存分配总量就已经超过 uint\_use\_sbrk\_byte\_threshold\_force\_page\_prefault，说明这个程序是一个大型应用软件，此参数将被自动修改为true。场景2是本内存库自身初始化完成后，此参数将被自动修改为true，这样除了最初申请的12M内存外（sbrk 8M+meta 2M + main thread 2M）外，其它2M内存块都会提前分配物理内存，实现更快的性能和更高的稳定性（减少缺页中断）。

这个参数的设计目标是：对于小型程序（malloc总量在2M以下）按需分配物理内存，这样可以节约物理内存，对于大型应用程序或者使用内存较多的程序，在mmap系统调用后将整个大块2M都申请为物理内存，可以有更高的性能。

此功能点启用后，会略微浪费空间，会增加用户程序的性能。因为集中产生实际物理页面的分配，会提升一些程序的稳定性（例如：避免Facebook fstring早期版本末尾‘\0’推迟实现Bug）。由于很多时候是在后台线程中分配的物理内存，节约了应用程序缺页中断的开销，提升了性能，提升了稳定性，解决了其它内存库存在的缺页中断瞬间闪退的小概率问题。

注意：申请内存总量在2M以上的场景，参数 bool\_enable\_malloc\_over\_2m\_force\_page\_prefault 决定malloc是否分配物理内存（默认是分配物理内存）。

注意：对于申请内存总量在2M以上的场景，bool\_enable\_realloc\_over\_2m\_force\_page\_prefault 决定realloc是否分配物理内存（默认是分配物理内存）。

mallopt int\_key=10009

bool\_enable\_giantpage

类型：Boolean

默认值：true

注意：是否启用 giantpage， 这里是giantpage的总开关。

mallopt int\_key=10010

bool\_force\_enable\_hugepage

类型：Boolean

默认值：true

注意：是否强制提前启用 hugepage。默认开启，考虑大内存的服务器和PC越来越普遍，提前使用巨页可以实现接近全部内存都是巨页，进一步减少TLB miss来提升性能。

mallopt int\_key=10011

bool\_force\_thread\_background

类型：Boolean

默认值：false

注意：是否强制提前启用 后台线程

mallopt int\_key=10012

除了修改配置参数的方法以外，如何在代码中提前启用hugepage和后台线程，增加性能表现？

mallopt int\_key=10011,是一个功能提前启用点，表示提前启用hugepage，而不考虑其他配置参数

１）如何在代码中提前启用hugepage，增加性能表现？

方法：mallopt( 10011, 1 );

Int\_key=编码10011内部定义为立即尝试启用hugepage。

注意：必须要保证机器内部配置有Hugepage2M的页面，才会真正启动hugepage2M。具体的hugepage2M的配置方法见《第五章产品介绍和说明 第11节关于HugePage的一些建议》。

mallopt int\_key=10012,是一个功能提前启用点，表示提前启用后台线程，　而不考虑其他配置参数

2）如何在代码中提前启用后台线程，增加性能表现？

方法：mallopt( 10012, 1 );

Int\_key=编码10012内部定义为立即尝试启用后台线程。

uint\_thread\_background\_div\_number\_calc\_prepare\_2m

类型：无符号整数

默认值：8

意思：后台线程计算需要提前准备的2M大块时的除数，默认值是8, 就是说内存使用2M块总量/除数8=需要准备的2M大块的数量。

范围：大于等于2， 低于2的配置将自动修改为2

mallopt int\_key=10013

uint\_thread\_background\_need\_prepare\_2m\_count\_limit

类型：无符号整数

默认值：32

意思：后台线程需要提前准备的2M大块的数量限制。默认值是32,表示限制需要准备的2M大块的数量为不超过32, 即限制最多只能提前准备32个2M大块。（计算方法见上面一条）对于需要更高性能的场景，可以配置更高的参数值。

mallopt int\_key=10014

uint\_microsecond\_thread\_background\_force\_release\_one\_2m

类型：无符号整数

默认值：100000

意思：默认值100000, 表示后台线程在长期没有内存申请的时候，每100000微秒（100毫秒）强制释放掉一个2M大块。这个功能是一个后台线程慢速释放的过程，是一个低优先级任务，前面还有一个后台线程快速释放的参数uint\_microsecond\_thread\_background\_force\_release\_one\_2m，可以在空闲时自动释放掉提前准备的多余2M大块内存，实现更少内存占用，副作用是降低了一些场景下的性能。与这个参数有关的 uint\_thread\_background\_keep\_2m\_count 表示慢速的一个释放过程中，将强制释放到最少需要保留的2M大块数量为止。 如果这个参数配置为零，表示彻底关闭掉这个功能，会浪费内存，会有更好的性能。

mallopt int\_key=10015

uint\_auto\_bind\_phy\_cpu\_count

类型：无符号整数

默认值：2

意思：默认值2, 表示启用＂工作线程自动绑定CPU＂后（见前面bool\_enable\_auto\_bind\_cpu

的配置说明），如果CPU不支持超线程，将自动将工作线程绑定到同一NUMA node下的2个相邻的物理CPU上。如果CPU支持超线程， 此参数无效。

范围：1或者2,超出范围的配置将自动修改为2。

相关参数：

bool\_enable\_auto\_bind\_cpu 用于设置启用线程绑定CPU功能的总开关。

uint\_bind\_cpu\_begin 用于设置绑定CPU的起始值。

uint\_bind\_cpu\_pcore\_ecore\_multi 用于线程设置绑定CPU大核小核的使用比例。

uint\_main\_thread\_bind\_cpu 用于设置主线程绑定CPU。

bool\_only\_bind\_pcore 用于设置是否线程只绑定大核CPU。

mallopt int\_key=10016

uint\_max\_wait\_nanosecond\_before\_clear\_cpu\_cache

类型：无符号整数

默认值：0

意思：默认值是0, 表示关闭此功能。当非零值时，表示如果等待释放的2M块指针已经在等待释放的队列中等待了超过此配置时间，将不执行这个2M块的清理Cache的任务，否则会通知所有CPU这个2M块所对应的L1/L2/L3 Cache已经失效了，以便腾出更多可用的空闲Cache空间，因此提高了应用程序的综合性能。

当此参数为零，表示关闭此功能。

mallopt int\_key=10017

bool\_enable\_thread\_background\_page\_release

类型：Boolean

默认值：true

意思：后台线程将会定期自动释放给多余的空闲常规page内存， 默认是允许。

mallopt int\_key=10018

bool\_enable\_thread\_background\_hugepage\_release

类型：Boolean

默认值：true

意思：后台线程将会定期自动释放多余的空闲hugepage2M内存， 默认是允许。

mallopt int\_key=10019

bool\_malloc\_size\_plus

类型：Boolean

默认值：false

意思：是否启用申请内存的时候1----2000KiB字节多申请8字节内存，主要是用于防止用户应用程序内部的一些越界bug， 默认是没有启用。当这个选项启用时，将会在申请1字节---2000KiB字节时至少保证多分配8字节的内存（类似于ptmalloc每次申请都会多分配8字节的设计，可以解决一些存在越界8字节以内的bug的程序稳定运行，可以提供更好的兼容性和稳定性）。对于2M字节以上的情况，不会多分配8字节的内存，还是按照2M字节进行对齐。

mallopt int\_key=10020

bool\_enable\_leak\_check

类型：Boolean

默认值：false

意思：是否启用内存泄露检查，当探测到内存泄露的时候，程序退出前会输出相关信息。Linux系统启用进程/线程时会有少量几个指针不释放，是正常现象，大约是进程有320字节和1024字节各一个指针，每创建一个线程有320字节的一个指针，另外版权探测代码有4096字节一个指针，由于系统设计的原因这些都不会自动释放，是正常现象。默认是没有启用内存泄露检查。

mallopt int\_key=10021

uint\_wait\_microsecond\_can\_release\_2m\_after\_malloc\_2m

类型：无符号整数

默认值：32000

意思：默认值是32000, 表示从系统申请2M块后（表示没有足够的备用2M块），后台线程将在之后的32000微秒（32毫秒）内暂停执行释放2M块的动作。这个配置参数减慢了内存的释放速度，提高了重复使用待释放内存的概率，因此提高了性能。

mallopt int\_key=10022

uint\_size\_use\_giantpage\_threshold

类型：无符号整数

默认值：1000000000

意思：默认值是1000000000，Giantpage页的使用阈值，大约是1G

mallopt int\_key=10023

bool\_use\_real\_realloc

类型：Boolean

默认值：true

意思：默认值是true, 表示使用真正的realloc，当设置为false后实际是使用malloc new+memcpy old+free old的方式实现的，减慢了一些场景下的性能，但是有更好的兼容性。

对于大于等于2MiB的内存申请，是尽量使用了mremap 系统调用来扩容，这种场景下可以实现无内存拷贝的高速扩容，具体参见操作系统的说明，实际还是使用了真正的realloc,无论是否返回原指针，Linux系统内部都有一些页面迁移优化，都可以尽量避免页面拷贝，提供更高的性能。

mallopt int\_key=10024

uint\_use\_sbrk\_byte\_threshold\_force\_page\_prefault

类型：无符号整数

默认值：163840

意思：默认值是163840，即160KiB，当程序使用的紧急内存超过160KiB后，将强制启用提前分配所有内存的功能，修改bool\_enable\_page\_prefault为true, 具体见参数bool\_enable\_page\_prefault的说明。应用程序进入初始化的最早阶段(内存库so自身初始化完毕前)，内存分配总量超过本参数的阈值，说明这个程序是一个大型应用软件。大型软件最早的初始化过程使用的内存更高，因此可以提前识别出来。

（紧急内存的解释：是通过sbrk方式申请的共享的一块内存（默认大小为8MiB），这块内存是紧急内存块，是so文件初始化尚未完成前就开始执行malloc申请的内存，主要用于内存分配库自身so文件还没有初始化完毕的场景下分配内存，因此也叫紧急内存)。

这个功能点的目的是对于一些大型程序通过启用分配物理内存和尝试提前使用 hugepage，增加大型应用程序的性能和稳定性。

mallopt int\_key=10025

bool\_show\_malloc\_status\_on\_enter

类型：Boolean

默认值：false

意思：在进入libhaisqlmalloc库后，自动显示内存使用情况，主要是以sbrk相关数据结构的内存使用情况。

mallopt int\_key=10026

bool\_show\_malloc\_status\_on\_exit

类型：Boolean

默认值：false

意思：在退出libhaisqlmalloc库前，自动显示内存使用情况。主要是用于调试显示各应用程序的在退出时的内存使用情况。

mallopt int\_key=10027

uint\_thread\_stack\_run\_page\_prefault\_size

类型：无符号整数

默认值：8192

意思：所有线程工作时，至少实际分配8KiB的物理栈空间。此参数在启用hugepage时无效，具体见配置参数 uint\_thread\_stack\_default\_size 的说明

mallopt int\_key=10028

bool\_enable\_thread\_background\_giantpage\_release

类型：Boolean

默认值：true

注意：后台线程允许执行自动释放giantpage内存。默认开启

mallopt int\_key=10029

uint\_cpu\_l1\_write\_cacheline\_clock

注意：此参数已经废弃，不再使用。

bool\_enable\_malloc\_over\_2m\_force\_page\_prefault

类型：Boolean

默认值：true

意思：默认值是true，表示对于大于2M的malloc内存申请，malloc申请内存返回前，将向系统申请实际分配物理内存。启用后将会提升程序稳定性，会浪费内存。本参数只管理单次申请大于2MiB内存。

mallopt int\_key=10031

bool\_enable\_realloc\_over\_over\_2m\_force\_page\_prefault

类型：Boolean

默认值：true

意思：默认值是true，表示对于大于2M的realloc内存申请，realloc申请内存返回前，将向系统申请实际分配物理内存。启用后将会提升程序稳定性，会浪费内存。本参数只管理单次申请大于2MiB内存。

mallopt int\_key=10032

uint\_vt\_bitmap\_preload\_size

类型：无符号整数

默认值：2

意思：默认值是2，表示提前加载2个bitmap数据块（每个bitmap大约占用128Byte）。这个功能可以略微提升性能，增大此参数会增加碎片出现的概率。不建议修改该值。

mallopt int\_key=10033

uint\_memset\_write\_through\_threshold

类型：无符号整数

默认值：1536

意思：默认值是1536，表示对于memset函数如果操作长度超过阈值1536字节，那么将使用直写内存模式，这种模式不写CPU Cache，减少了CPU Cache颠簸，可以提供更高的memset性能，但是由于CPU Cache中没有数据，应用程序使用这部分数据时需要读一次内存，因此，这个参数设置为多少没有标准答案。由于使用avx/sse指令集重写了memset，该参数目前只对X86系列CPU有用.

mallopt int\_key=10034

uint\_memcpy\_write\_through\_threshold

类型：无符号整数

默认值：1536

意思：默认值是1536，表示对于memcpy / memmove函数如果操作长度超过阈值1536字节，那么将使用直写内存模式，这种模式不写CPU Cache，减少了CPU Cache颠簸，可以提供更高的memcpy / memmove性能，但是由于CPU Cache中没有数据，应用程序使用这部分数据时需要读一次内存，因此，这个参数设置为多少没有标准答案。由于使用avx/sse指令集重写了memcpy / memmove，该参数目前只对X86系列CPU有用.

mallopt int\_key=10035

uint\_free\_delay\_nanosecond

类型：无符号整数

默认值：5000

意思：默认值是5000，表示在多线程下，将启用延迟释放，对于已经使用过的释放内存，在5微秒后才可以重复使用。这个功能点可以解决或者缓解一部分use after free(UAF)问题，也可以解决或者缓解一部分Lock Free ABA问题，可以提升一些应用程序的多线程稳定性。

mallopt int\_key=10036

uint\_thread\_background\_keep\_2m\_count

类型：无符号整数

默认值：2

意思：默认值是2，表示强制释放最少需要保留的2M大块数量，默认值是2，表示强制释放到最后只剩下2个2M大块为止。后台线程在长期没有内存申请的时候，定期强制释放掉一个2M大块。这个功能是一个后台线程慢速释放的过程，是一个低优先级任务，具体可以参考uint\_microsecond\_thread\_background\_force\_release\_one\_2m，

前面还有一个后台线程快速释放的参数uint\_microsecond\_thread\_background\_force\_release\_one\_2m，可以在空闲时自动释放掉提前准备的多余2M大块内存，实现更低的内存占用，但是副作用就是降低了一些场景下的性能。

mallopt int\_key=10037

bool\_preload\_wait\_free\_queue1

类型：Boolean

默认值：false

注意：当这个参数设置为true时，将提前预先加载一个无等待队列1，大约会提前申请占用36KB的内存空间，会减少第一次使用无等待队列的动态创建时间。启用此参数可提升多线程性能。无等待队列1用于内存的跨线程释放，该队列最大支持8个工作线程，可以在同一时钟周期同时无等待并发8个工作线程。这个无等待队列1可以在数ns内实现push/pop功能。当并发超过8个线程时，将创建和使用无等待队列2（见配置参数bool\_preload\_wait\_free\_queue2），这个无等待队列2支持无限个CPU，最多8个CPU在同一时钟周期内同时无等待并发，支持无限数量CPU和无限长度队列空间，支持队列长度自增长，无等待队列2的效率低于无等待队列1，两种类型的无等待队列共同处理跨线程释放内存，具备极高的并发性能。

mallopt int\_key=10038

bool\_preload\_wait\_free\_queue2

类型：Boolean

默认值：false

注意：当这个参数设置为true时，将提前预先加载一个无等待队列2，大约会提前申请占用36KB的内存空间，超过8个线程以上的场景下，减少了无等待队列2的动态创建时间。启用此参数可提升多线程性能。无等待队列2用于内存的跨线程释放，支持无限个CPU，最多8个CPU在同一时钟周期内同时无等待并发，支持无限数量CPU和无限长度队列空间，支持队列长度自增长，无等待队列2的工作效率略低于无等待队列1，远超任何锁或者Lock Free Queue的性能，均有数量级的性能提升，两种类型的无等待队列共同处理跨线程释放内存，具备极高的并发性能。

mallopt int\_key=10039

uint\_cpu\_hyper\_thread\_tsc\_multi

意思：此参数已经废弃，不再使用。

uint\_sbrk\_prefault\_byte\_size

类型：无符号整数

默认值：16384

意思：默认值是16384，表示将会在唯一的一次sbrk申请（用于紧急场景的内存申请）后，提前分配多少字节的物理内存，这样后续使用这块内存的过程中就不用分配物理内存了。此功能点的作用是减少Linux内核的物理分配内存的并发冲突，可以提升一些应用程序的性能和稳定性。当so文件初始化尚未完成前就开始执行malloc申请的内存（即紧急场景的内存申请），这部分有一个专门的小模块单独处理，使用sbrk申请一大块内存，然后切成小片分配给用户。

mallopt int\_key=10041

uint\_thread\_create\_wait\_random\_microsecond\_mask

类型：无符号整数

默认值：0

意思：默认值是0，表示不启用。启用此功能后，新创建的线程随机sleep小段时间，以便错开各线程的操作时间，减少并发冲突。这个值要求是2的N次方减一，是随机值的上限。

mallopt int\_key=10042

uint\_thread\_create\_wait\_random\_tsc\_mask

类型：无符号整数

默认值：0

意思：默认值是0。

意思：默认值是0，表示不启用。启用此功能后，新创建的线程随机mm\_pause小段cpu\_tsc时间，以便错开各线程的操作时间，减少并发冲突。这个值要求是2的N次方减一，是随机值的上限。

mallopt int\_key=10043

uint\_mpmc\_giantpage\_2m\_min

类型：无符号整数

默认值：100

意思：默认值是100。mpmc giantpage队列中，2M块低于阈值时，后台线程会进行提前分配2M块到队列中，保证队列中的2M块始终在阈值以上。

mallopt int\_key=10044

uint\_mpmc\_giantpage\_2m\_max

类型：无符号整数

默认值：128

意思：默认值是128。mpmc giantpage队列中2M块使用阈值，超过该值后台线程会自动进行回收。

mallopt int\_key=10045

uint\_mpmc\_giantpage\_2m\_force\_update\_count

类型：无符号整数

默认值：16

意思：默认值是16。mpmc giantpage队列中，强制使用16个新的2M块替换原来的旧2M块，这样依次进行替换，可以置换出早期释放的2M块，以便及时回收完全空闲的1G页面，减少碎片。

mallopt int\_key=10046

bool\_only\_bind\_pcore

类型：Boolean

默认值：false

意思：默认值是false。是否启用线程只绑定大核CPU。

相关参数：

bool\_enable\_auto\_bind\_cpu 用于设置启用线程绑定CPU功能的总开关。

uint\_bind\_cpu\_begin 用于设置绑定CPU的起始值。

uint\_bind\_cpu\_pcore\_ecore\_multi 用于线程设置绑定CPU大核小核的使用比例。

uint\_main\_thread\_bind\_cpu 用于设置主线程绑定CPU。

mallopt int\_key=10047

uint\_bind\_cpu\_begin

类型：无符号整数

默认值：9999

意思：默认值是9999，默认值表示不启用。当此值大于最大CPU编号时，此功能无效。此配置值用于设置线程绑定cpu起始序号。当此配置参数无效时，默认从当前主线程所在CPU的下一个CPU开始。

相关参数：

bool\_enable\_auto\_bind\_cpu 用于设置启用线程绑定CPU功能的总开关。

uint\_bind\_cpu\_begin 用于设置绑定CPU的起始值。

uint\_bind\_cpu\_pcore\_ecore\_multi 用于线程设置绑定CPU大核小核的使用比例。

uint\_main\_thread\_bind\_cpu 用于设置主线程绑定CPU。

mallopt int\_key=10048

uint\_bind\_cpu\_pcore\_ecore\_multi

类型：无符号整数

默认值：2

意思：默认值是2。大核和小核的负载倍数，被调度顺序先大核，后小核。 用于线程设置绑定CPU大核小核的使用比例。

相关参数：

bool\_enable\_auto\_bind\_cpu 用于设置启用线程绑定CPU功能的总开关。

uint\_bind\_cpu\_begin 用于设置绑定CPU的起始值。

uint\_bind\_cpu\_pcore\_ecore\_multi 用于线程设置绑定CPU大核小核的使用比例。

uint\_main\_thread\_bind\_cpu 用于设置主线程绑定CPU。

mallopt int\_key=10049

uint\_thread\_stack\_default\_size

类型：无符号整数

默认值：2097152

意思：默认值是2M。线程栈内存空间大小。如果有可用的巨页（Hugepage，Giantpage），该参数才会生效。此配置值为零，表示关闭此功能。此功能点启用后，线程的stack栈空间将自动分配和使用1个huagepage巨页，这样可以减少TLB miss，提升多线程的性能。例外情况如下：当用户调用pthread\_create的第2个参数pthread\_attr非空的场景下，表示用户进程对pthread\_attr属性有读写可能对栈内存大小类型等的前后一致性有特殊要求，Libhaisqlmalloc 库将不会尝试替换栈空间为巨页，以免出现问题，另外对于进程的主线程也不会尝试替换栈空间为巨页。

mallopt int\_key=10050

bool\_enable\_mmap\_populate

类型：Boolean

默认值：true

意思：默认值是true。是否启用mmap函数调用时，携带MAP\_POPULATE参数，将导致从操作系统申请的匿名内存块，提前分配共享的一个全零的readonly的空白页，会提升读性能，因为不再有读缺页中断。

mallopt int\_key=10051

uint\_cpu\_tsc\_bias\_per\_phy\_cpu

注意：此参数已经废弃，不再使用

bool\_show\_numa\_map\_on\_exit

类型：Boolean

默认值：false

意思：默认值是false。默认值表示不启用。启用此功能点后，在程序退出时，显示numa map详细信息。可以观察到page giantpage hugepage 的占用情况和NUMA的编号。

mallopt int\_key=10053

uint\_main\_thread\_bind\_cpu

类型：无符号整数

默认值：99999

意思：默认值是99999。默认值表示不启用。当此值大于最大CPU编号时，此功能无效。此配置值用于设置主线程绑定cpu。此参数对于一些单线程程序，例如Redis也可以实现绑定CPU。默认参数下，尝试绑定CPU到到一个物理CPU下的2个逻辑CPU，当此参数对应的逻辑CPU不支持超线程时，将按照参数uint\_auto\_bind\_phy\_cpu\_count所定义的值依次绑定一个或者2个逻辑CPU。

mallopt int\_key=10054

bool\_show\_config\_on\_exit

类型：Boolean

默认值：false

意思：默认值是false。在程序退出时，显示malloc\_config的各个参数的配置信息。

mallopt int\_key=10055

bool\_force\_enable\_giantpage

类型：Boolean

默认值：false

意思：默认值是false。是否强制启用giantpage巨页。当启用此功能后会立即尝试启用1G页面。

mallopt int\_key=10056

uint\_thread\_background\_check\_release\_thread\_millisecond

类型：无符号整数

默认值：200

意思：默认值是200。后台线程回收已退出线程相关资源的保护时长，默认200毫秒。实际后台运行该回收机制会大于该值。

mallopt int\_key=10057

uint\_cpu\_mesi\_keep\_tick

注意：此参数已经废弃，不再使用。

bool\_hook\_pthread\_lock\_check\_sync

类型：Boolean

默认值：true

意思：默认值是true，表示默认启用 pthread\_lock 的同步校验功能。此功能启用后，内存库将hook pthread\_lock / unlock函数调用，执行锁保护的数据的同步校验，pthread\_unlock增加大约100ns开销（存储unlock cpu和当前unlock utc microsecond）,pthread\_lock当检测到上次 pthread\_unlock 的CPU与当前CPU一致时，或者上次pthread\_unlock 的utc时间与当前utc时间差>1微秒时，这两种情况占比很高，这两种情况可以保证数据100%同步可以走短流程，避免此处长流程的锁保护数据的同步校验大约200ns的额外开销。此功能点提升了 lock / unlock锁的可靠性，保证了锁保护数据的同步，解决了一处小概率的数据同步问题（现代CPU非常复杂，写入指令提交到写入队列后即认为写入完成，指令返回，此时保证严守顺序性承诺，但是现代CPU Cache由于超大的容量，每一级Cache之间都有非常深的写入队列，CPU设计中成批写入的burst模式在L3和内存的写入过程中被大量使用，用于以减少写入过程的片选信号周期，对可能连续写入的数据在L3/内存写入队列内进行重排序，按照地址大小顺序写入以减少片选信号时延，在L3层面或者内存层面上内存写入顺序被重排序，就失去了指令层面上的顺序性承诺。其它CPU在L3层面或者内存层面有概率会读到 “与sfence承诺不一样的内容”，即有可能先看到sfence后写入的内容，后看到sfence前写入的内容。此问题是小概率问题，只有个别特殊场景下才会出现。)

mallopt int\_key=10059

bool\_show\_thread\_cpu\_bind

类型：Boolean

默认值：false

意思：默认值是false。是否显示线程与CPU的自动绑定方案。默认场景下，线程将优先绑定到一个大核即一个物理CPU下的2个逻辑CPU中，支持Intel CPU的大小核的设计，在使用小核时将默认绑定2个物理小核。

mallopt int\_key=10060

uint\_giantpage\_malloc\_fail\_wait\_microsecond

类型：无符号整数

默认值：100000

意思：默认值是100000。表示0.1秒，当后台线程尝试向操作系统申请1GB的页面失败时，将记录下此时间，在接下来的0.1秒内不会继续重试，由于此时操作系统没有空闲的1GB页面，过于频繁重试没有意义，因此利用此参数暂停1GB页面的操作系统调用，降低了总开销，提升了综合性能。

mallopt int\_key=10061

uint\_hugepage\_malloc\_fail\_wait\_microsecond

类型：无符号整数

默认值：10000

意思：默认值是10000。表示10毫秒，当后台线程尝试向操作系统申请2MB的页面失败时，将记录下此时间，在接下来的10毫秒秒内不会继续重试，由于此时操作系统没有空闲的2MB页面，过于繁重试没有意义，因此利用此参数暂停2MB页面的操作系统调用，降低了总开销，提升了综合性能。

mallopt int\_key=10062

uint\_fork\_thread\_bind\_cpu

类型：无符号整数

默认值：99999

意思：默认值是99999。默认值表示不启用。当此值大于最大CPU编号时，此功能无效。此配置值用于设置fork之后的进程绑定cpu。此参数对于部分少量使用了fork函数的程序有用，例如Redis使用fork来实现数据备份，进程绑定CPU可以避免写盘IO等待过程出现多个CPU上的切换颠簸。默认参数下，尝试绑定CPU到到一个物理CPU下的2个逻辑CPU，当此参数对应的逻辑CPU不支持超线程时，将按照参数uint\_auto\_bind\_phy\_cpu\_count所定义的值依次绑定一个或者2个逻辑CPU。

mallopt int\_key=10063

bool\_enable\_hugepage\_bind\_numa

类型：Boolean

默认值：false

意思：默认值是false。是否启用hugepage（2M页面）绑定到NUMA，目前由于Linux内核不支持直接Hugepage绑定NUMA的函数调用，因此使用特殊方式实现此功能，在一些版本的LInux下存在问题。

mallopt int\_key=10064

bool\_memory\_show\_hugepage\_bind\_numa

类型：Boolean

默认值：false

意思：默认值是false。是否启用显示hugepage绑定到NUMA的情况，这个参数用于上个参数的调试，显示具体的hugepage 与 NUMA node的绑定关系。

mallopt int\_key=10065

bool\_memory\_use\_hugepage\_other\_numa

类型：Boolean

默认值：true

意思：默认值是true。当前线程所在的NUMA node的hugepaage空间不足时，是否能够使用其它NUMA node 的 hugepage，推荐是允许，这样在多个NUMA的内存分配不平衡时，也会有比较稳定的性能表现。

mallopt int\_key=10066

bool\_enable\_giantpage\_bind\_numa

类型：Boolean

默认值：false

意思：默认值是false。是否启用giantpage（1G页面）绑定到NUMA，目前由于Linux内核不支持直接giantpage绑定NUMA的函数调用，因此使用特殊方式实现此功能，在一些版本的LInux下存在问题。

mallopt int\_key=10067

bool\_memory\_show\_giantpage\_bind\_numa

类型：Boolean

默认值：false

意思：默认值是false。是否启用显示giantpage绑定到NUMA的情况，这个参数用于上个参数的调试，显示具体的giantpage 与 NUMA node的绑定关系。

mallopt int\_key=10068

bool\_memory\_use\_giantpage\_other\_numa

类型：Boolean

默认值：true

意思：默认值是true。当前线程所在的NUMA node的giantpaage空间不足时，是否能够使用其它NUMA node 的 giantpage，推荐是允许，这样在多个NUMA的内存分配不平衡时，也会有比较稳定的性能表现。

mallopt int\_key=10069

uint\_string\_size\_div8\_multi

类型：无符号整数

默认值：10

意思：默认值是10。表示计算string size 时先除以8再乘以10，那么实际就会分配1.25倍的内存空间，这个参数配置对于多数场景下节约了内存，但是降低了string扩展时的性能。对比gcc std::string默认是按照2倍空间进行分配的，浪费了空间，提升了string扩展时的性能。

mallopt int\_key=10070

uint\_vector\_size\_div8\_multi

类型：无符号整数

默认值：12

意思：默认值是12。表示计算vector size 时先除以8再乘以12，那么实际就会分配1.5倍的内存空间，这个参数配置对于多数场景下节约了内存，但是降低了vector扩展时的性能。对比gcc std::vector默认是按照2倍空间进行分配的，浪费了空间，提升了vector扩展时的性能。

mallopt int\_key=10071

uint\_giantpage\_delay\_unmap\_millisecond

类型：无符号整数

默认值：4000

意思：默认值是4000。表示后台线程释放giantpage（1G页面）给操作系统时，将延迟4000毫秒再释放，此期间如果需要 giantpage,那么就可以复用此giantpage而避免了与操作系统的交互，节约了开销，提升了综合性能，避免刚释放的giantpage之后马上又需要从操作系统重新申请giantpage。

mallopt int\_key=10072

uint\_sleep\_microsecond\_on\_exit

类型：无符号整数

默认值：20000

意思：默认值是20000。表示内存库在退出前usleep20000微秒。由于内存库的各种函数性能过快，此功能用于限制退出速度，提升类似于 cat xxx.txt | grep aaa这样需要跨进程传送数据的场景下的稳定性。当此参数小于40微秒时，上述场景有小概率崩溃的风险，因此不建议此参数配置过小。

mallopt int\_key=10073

bool\_force\_main\_thread\_auto\_bind\_cpu

类型：Boolean

默认值：false

意思：默认值是false。是否启用强制主线程自动绑定CPU，具体见uint\_main\_thread\_bind\_cpu的说明，当uint\_main\_thread\_bind\_cpu时一个非法值时，将自动使用当前主线程所在的CPU 作为默认绑定CPU。

mallopt int\_key=10074

bool\_force\_fork\_thread\_auto\_bind\_cpu

类型：Boolean

默认值：false

意思：默认值是false。是否启用强制主线程自动绑定CPU，具体见uint\_fork\_thread\_bind\_cpu的说明，当uint\_fork\_thread\_bind\_cpu时一个非法值时，将使用当前默认的下一个线程绑定CPU 作为默认绑定CPU。

mallopt int\_key=10075

uint\_create\_thread\_sleep\_microsecond

类型：无符号整数

默认值：0

意思：默认值是0。表示创建线程后usleep 一小段时间。默认是不启用。此参数用于在批量创建线程时，避免创建线程速度快导致大量线程同时竞争少量锁等带来一些小概率的稳定性问题，由于默认启用了参数 bool\_hook\_pthread\_lock\_check\_sync 的关系，此功能点是可以配置关闭的。

mallopt int\_key=10076

uint\_stack\_block\_delay\_free\_microsecond

类型：无符号整数

默认值：256000

意思：默认值是256000。表示用于线程栈内存的2M块在线程退出后，延迟一小段时间（256毫秒）再释放。此功能用于解决线程线程栈内存的2M块指针用于操作系统内部thread id和C++ std::thread\_id的区分，销毁线程后立即复用此内存块如果此2M块也被用于另外一个新线程的栈内存，那么就存在小概率问题（前后两个线程无法区分导致内核逻辑混乱），因此，这里延迟一小段时间后再复用，前一个正在退出的线程内核数据随着线程退出内部数据全部销毁后复用就没有冲突了。

mallopt int\_key=10077

g\_bool\_memory\_use\_thread此是内部值无法直接配置，但是可以通过mallopt 修改

类型：Boolean

默认值：false

意思：默认值是false。用于实现多线程的全部线程退出后，通过mallopt(10078,0) 修改此参数为false, 实现更快速的free，提升退出速度

mallopt int\_key=10078

reload\_init\_data 此是内部函数调用无法直接配置，但是可以通过mallopt 间接调用，实现默认后台线程的默认参数的重加载。

用于实现多线程的全部线程退出后，通过mallopt(10079,0) 调用执行reload\_init\_data函数，实现后台线程的默认参数的重加载, 实现更快速的free，提升后台线程的对操作系统的内存释放速度。

mallopt int\_key=10079

str\_hex\_bind\_cpu\_mask

类型：字符型

默认值：FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

意思：表示线程绑定CPU的掩码。通过掩码的方式，我们可以隔离出一些CPU不让线程使用，以便这些CPU隔离用于特殊用途。参考linux的cpu mask规范，即零号CPU对应字符串最右端的最后一位。

mallopt int\_key=无编号

# 第五章 产品介绍和说明

## 1.产品介绍

(1)libhaisqlmalloc.so

这个是库文件，适合在Linux 系统下的内存分配库。

（2）test\_malloc\_use\_so：

这个是linux 下的已经编译好的一个测试文件。

性能测试软件的用法：

1. 测试用法1： ./test\_malloc\_use\_so

2）测试用法2： ./test\_malloc\_use\_so --help

Usage: test\_malloc\_use\_so test\_count loop\_count thread\_count

Default: test\_count 1000000

Default: loop\_count 2

Default: thread\_count 1

1. 测试用法3： ./test\_malloc\_use\_so 10000 2 8

（3）test\_malloc\_use\_so.cpp文件：

这个文件是上面所述的性能测试文件的C++源代码，用于进行性能测试比对。这个测试软件是自研的开源代码，为了方便大家使用和修改，Apache2开源协议，提供了全部源码。

为了方便软件工程师和系统管理员使用内存分配库，需要对Libhaisqlmalloc内存分配库的内部实现有细致的理解，因此在后面详细介绍了内存库的各种参数的设计和实现方案。

## 2.性能高的主要原因

Libhaisqlmalloc高性能内存分配库，用于减少频繁分配、释放内存造成的性能损耗，并且有效地控制内存碎片。Libhaisqlmalloc采用了新的软件技术，具备极高的性能表现。在Intel CPU 9400F(2.9G默认频率)下测试100万次malloc4KiB平均一次4KiB malloc动作, tcmalloc需要近一千ns, 而Libhaisqlmalloc只要5ns。

Libhaisqlmalloc对多线程做了很多优化，所有分配几乎不存在锁竞争。分配给线程的内存全部是按需分配，提高了内存利用率，不会浪费内存，也减少了内存碎片。

Libhaisqlmalloc为每个线程分配了一个线程局部的Thread Local Cache，线程申请的2000KiB以下内存都是在其Thread Local中的对象负责分配，由于是Thread Local的，所以是无锁的。同时，Libhaisqlmalloc维护了进程级别的Cache，所有的2M大小的块都在这个进程级别的Cache中分配，这个Cache使用了一种多生产者多消费者MPMC的队列，开销小，性能高。

Libhaisqlmalloc内存分配库性能高的主要原因是：

1. 后台线程是高性能的关键。消耗时间大的系统调用，直接用后台线程处理，提高前端的性能，后台线程处理比较耗时的那些动作全部处理了。例如，一次 mmap /ummap 2M系统调用，大约开销会有几百微秒，如果用后台线程去处理，提前准备好2M 块，以及推迟释放2M，大约只需要几十个ns就可以获得/释放2M块，这块提高了“获得/释放2M块”这个功能点大约1万倍性能，分摊到每次8byte申请上的时间前端表现就是申请8byte 与申请4K byte 消耗的时间几乎一致，都是几个ns 的时间。
2. Bitmap算法。现代CPU受内存墙影响性能，Bitmap算法占用内存小，占用释放一个基本空间只有1个bit修改, 而传统算法是放一堆指针，修改一个指针就是64bit的修改量，实际要修改很多地方。如果申请和释放大量内存，从数据修改总量上看，Bitmap具备更低的数据修改总量。
3. 连续性好的数据结构。传统算法中的链表数据结构由于cache不友好性能很低 。Libhaisqlmalloc 库采用连续性好的数据结构（例如vector）保存数据，性能很高。
4. 无锁或者锁冲突比较少的数据结构，增加多线程性能。Libhaisqlmalloc 库内部实现了广义上的无锁，大量使用RCU/Wait\_free/Per CPU/ Thread Local的无锁高性能组件，没有并发瓶颈，具备非常高的多线程性能。
5. Libhaisqlmalloc 库重写了部分常用的std库，提供了更好的性能。

## 3.内存块种类group说明

Libhaisqlmalloc 库的管理内存范围是：

1字节到2000KiB字节使用Bitmap方式分配。

2000KiB---2MiB（含2MiB）之间的内存是由系统mmap方式直接分配或者由后台线程间接从系统mmap方式申请得到。

2MiB以上的内存实际是由系统mmap方式直接分配。

1)1-3408字节以下是小内存，使用17种不同大小的块，17种group进行分配。

8字节以下单独使用一组group,按照8Byte对齐，其余是16Byte对齐，使用其他16组group。

17种group的总体块大小依次是4K, 8K, 8K×2, 8K×3, 8K×4, 8K×5, 8K×6, 8K×7, 8K×8, 8K×9, 8K×10, 8K×11, 8K×12, 8K×13, 8K×16, 8K×32, 8K×64。

A)头9个类型即128字节以内的是小分组，每个group块的内部都有一个512bit(64Byte)的Bitmap数据结构用于申请和释放内存，只允许单bit（1bit）申请。每个group都是Cache Line 64Byte对齐的，group数据结构自身总体大小约是128字节，对应512次内存申请，平均每次内存申请的cookie数据结构平均管理内存占用开销大约只有2bit。这种高度压缩的bit数据结构比传统的内存分配库的cookie数据结构小N倍，性能极高。这种内存的处理流程通过Thread Local方式实现无锁。

B)其他8个类型是第二个分组，用于分配128字节以上的情况，每个group块的内部都有一个512bit(64Byte)的Bitmap数据结构用于申请和释放内存，允许多bit申请，所以另外还有一组256字节压缩bit数据结构用于保存每个已经申请内存的指针对应的内存大小等cookie参数。每个group都是Cache Line 64Byte对齐的，group数据结构自身总体大小约是400字节，对应512次内存申请，平均每次内存申请的cookie数据结构平均管理内存占用开销大约只有6bit，这种高度压缩的bit数据结构比传统的内存分配库的cookie数据结构小N倍，性能极高。这种内存的处理流程通过Thread Local方式实现无锁。

2)3409字节----10KiB是小内存（N\*1KiB)或者中内存(N\*4KiB)，使用2种不同大小的块，每个group子块是1024或者4096，根据情况分别选择不同大小的group进行处理。这种内存的处理流程通过Thread Local方式实现无锁。

3)10KiB----2MiB字节是中内存，使用4KiB对齐的分配策略，由总大小是2MiB的group进行分配。其中(10KiB+1)----2000KiB字节的申请会在4KiB对齐的group内分配；group的数据结构是Cache Line对齐的，group自身的总体大小约是128字节，额外使用了4KiB的管理空间，对应512次内存申请，每指针的管理内存占用开销大约只有66bit,这种压缩的bit数据结构性能极高。这种内存的处理流程通过Thread Local方式实现无锁。

4)(2000KiB+1)----2048KiB之间实际分配2048KiB，在group外分配，这里会优先使用进程级别的Cache----多线程MPMC无锁队列中的2MiB块，所以，性能表现会比group内的略小于2000KiB的分配性能更高。这种内存的处理流程是MPMC Per CPU 方式。

5)2M以上内存，是大内存，由于默认自动支持Hugepage2M, 2MiB以上均是按照2MiB对齐方式分配的, 使用2MiB对齐的策略，使用系统调用mmap进行分配。在存在Hugepage2M的情况下，优先使用Hugepage2M。大内存的cookie数据结构是一种优化的unordered\_map数据结构（开放地址法+线性探测+SSE4.2 crc32c hash算法），用于指针ptr\_void与申请内存大小size等cookie数据的关联关系查询，大内存在实际应用中情况比较少，处理并发的代码很简单，直接使用了自研Spin Lock锁，来处理并发。自研Spin Lock锁支持偏向锁（类似于Java），具备更高的稳定性。

## 4.内存地址对齐和默认支持SSE指令集

Libhaisqlmalloc库有良好的分配内存地址对齐，默认支持SSE（Single Instruction Multiple Data）单指令多数据技术指令集，方便开发高性能计算密集的高性能用户代码。对于N\*16字节( N=任意数）,默认分配的内存地址都是16字节对齐的, 默认支持SSE指令集。（C++委员会建议尽量16字节对齐，以便默认支持SSE指令集）

下列字节的申请大小保证是彻底按照申请大小对齐的：8,16,32,64,128,256,512,1024,4096,2M。

支持C++17 带std::align\_val\_t 参数的new等, 具体见下一章关于对外函数接口的说明。 因此， 对于C++17以后可以通过支持std::align\_val\_t，进而支持更多SIMD指令集 。

对于N\*4KiB字节,分配的内存地址都是4KiB字节对齐的。

对于N\*2MiB字节,分配的内存地址都是2MiB字节对齐的。目前的版本更高支持到2MiB对齐。

## 5.内部碎片

Libhaisqlmalloc库碎片表现：Libhaisqlmalloc库内部碎片平均浪费小于10%。Libhaisqlmalloc库内部碎片设计要求是32Byte----10KiB的内存分配，内部碎片平均浪费小于10%，碎片浪费率小于20%。在满足这个内部碎片率的前提下才会考虑性能，从多种group方案中选取折中的技术方案。因此，低碎片率是基本要求，其次才是性能。

申请不同大小的内存，因为内部对齐而产生碎片，这种碎片称为内部碎片， 存在三种内部碎片。

内部碎片１：

1----8字节首先按照 8字节对齐，内部碎片１范围是1---7字节;

9字节----4KiB按照16字节对齐，内部碎片１范围是1---15字节;

(4KiB+1)字节----10KiByte按照1KiB对齐，内部碎片１范围是1---1023字节;

(10KiB+1)字节----2000KiByte按照4KiB对齐，内部碎片１范围是1---4095字节;

C++委员会建议内存分配16字节以上的场景下，尽量采用16字节对齐，以便默认支持SSE指令要求的对齐方式, 提高软件的性能表现。Libhaisqlmalloc库对于9字节以上的内存申请均是16字节对齐。

内部碎片２：

对于16字节----3408字节的内存，Libhaisqlmalloc库有第2次对齐的过程，因此，同时存在内部碎片１和内部碎片２。由于group内部块对齐而产生碎片，这种碎片称为内部碎片2。

内部碎片2是可以调整的碎片，换不同大小的group块，获得不同的性能表现和不同大小的内部碎片2。group类型过多会带来更多外部碎片，因为每种group都会浪费更多的内存, 所以，我们目前的设计方案中group类型只有18种，就覆盖了1Byte----2MB字节的所有分配方案．

Libhaisqlmalloc库的Bitmap算法搜索连续bit数越多性能越低，低bit数的内存分配方案会有更好的性能，因此总体的碎片设计规则是：在满足内部碎片率低于20%的前提下，尽量选择bit数低的group方案(在18个group中选择）。malloc的调用从多种group方案中选取一个折中的技术方案，适当的内部碎片，尽可能高的性能。

具体的设计方案:

16字节----3408字节 参见后面附录一内部碎片的设计方案。内部碎片2碎片率是16.7%以下，平均碎片率小于8%，这部分同时存在内部碎片1和内部碎片2。内部碎片1是按16字节对齐而产生的，这个碎片很小，是不可以优化的碎片。内部碎片2是由于存在多种group类型适配造成的碎片，是可以调整可以优化的内部碎片。

3409字节----4KiB按照4KiB分配，碎片范围1---687Byte，碎片率低于16.77%，平均碎片率8.4%，只有一次对齐，只有内部碎片１。

(4KiB+1)----2000KiB的malloc的内部碎片浪费很小，只有一次对齐，只有内部碎片１。

(4KiB+1)----10KiB是按照1KiB对齐分配，碎片范围1---1023字节.。碎片率低于20%，平均碎片率低于10%。只有一次对齐，只有内部碎片１。

(10KiB+1)----2000KiB是按照4KiB对齐分配，浪费范围1---4095字节。平均碎片率远低于10%。只有一次对齐，只有内部碎片１。

内部碎片３：

对于2000KiB以上的内存，Libhaisqlmalloc库有１次额外的对齐过程，这种对齐是按照2MiB字节对齐，因此，存在内部碎片３。2MiB以上内存，是大内存，由于支持Hugepage2M, 为了提供一致的表现，2000KiB以上均是按照2MiB对齐方式分配的, 使用了2MiB对齐的策略，浪费范围1---2097151(2MiB-1)字节。这种场景下，如果申请内存量不是2MiB的倍数，那么就会有内部碎片３带来的浪费（好在这种情况在应用中很少出现）。如果申请内存量是2MiB的倍数，那么就不会有碎片浪费。

测试代码的开头有测试内存分配库内部碎片的情况, 但是实际情况要复杂的多，Linux malloc库虽然内部碎片很低，但是有外部碎片，总体的碎片浪费很严重。

## 6.外部碎片：

Libhaisqlmalloc库有很低的外部碎片。Libhaisqlmalloc 库设计思路是优先考虑低外部碎片和低内存浪费的方式，宁可在一些场景下以牺牲性能表现为代价，来换取低外部碎片和低内存浪费。

外部碎片的定义：由于执行过程中反复的malloc/free不同大小的内存块，会造成连续的内存中间一些块被占用，缺乏连续空间用于大块和连续块的分配，这种碎片是外部碎片，会造成的空间浪费，会占用更多的空间。

在减少外部碎片方面，Libhaisqlmalloc库有更好的解决方案，一个设计思路就是线程内部不做空间缓冲的方案： 内部的内存申请原则上采用按需分配，在需要时会立即申请，释放时都是立即释放不做缓存。这种方式减慢了性能，增加了内存的利用率，有更小的内存消耗．更少的外部碎片。

1）小内存无缓存，不会预分配备用。每一次free释放都是真实的释放，不会缓冲起来下次再用，这样保证只要一大块内存内的多个连续块释放后，就可以获得彻底连续的空间，不会有传统算法缓存带来额外碎片的问题。例如：有些传统内存分配库为了性能将一些指针缓冲起来下次再用，而有可能刚好就缺这部分块就可以空间连续。

2）所有大小的group都是动态分配，不存在预分配，不会有空闲单元保留，一定会释放掉全部空闲单元，宁可大幅度减慢性能，也不会保存起来等待下次再用。这种精细的内存管理设计就减少了预留内存块的浪费，不会因为预分配的group块不符合实际需求造成浪费，简单的就实现了低浪费低外部碎片。

3）低碎片率的内存分配的策略，首先是采用了18种group（远远低于很多库的设计数量）, 这样不同大小的内存分配走不同的group, 尤其是常用的256Byte以下，都是单bit分配，没有碎片浪费，高于256Byte采用了2倍增长的group类型，有足够的区分度，是小bit数分配，碎片浪费也比较少。

4）搜索算法上使用了一种低内存浪费的搜索解决方案，First Fit+Next Fit同时使用。当free()调用后，将自动根据释放的内存地址重新计算下次Next Fit的地址，等效于使用First Fit从第一个可用地址开始搜索，这样保证释放掉的内存可以立即被重复使用．扩大了搜索范围，大幅度减慢了性能，提高了碎片利用率，第二次malloc同样大小的数据块，使用Next Fit算法，从上次分配内存的地方继续向后搜索和分配，加快内存分配速度．这样既保证了性能，又保证了刚释放的空间立即复用不浪费，减少了碎片，大大节约了内存。

5）一种低碎片率的古老的原始的Bitmap算法，这种算法不同于传统算法（产生大量碎片的）buddy伙伴分配算法，算法的优点是自身不会产生额外碎片，具备很低的碎片。缺点是需要检索更多的数据，执行更多的指令，理论上会减慢性能表现。

Bitmap架构算法，就是一次在一个Cache Line中（64Byte，512bit内存中）对多个连续N个全零bit搜索，找到了连续N个零bit就可以在这里映射bit对应的地方分配内存了，并且free时会自动实现大小块自动合并，连续块自动合并，没有其他(非Bitmap)模型的连续块合并的开销和大小块合并的开销，因此，一方面碎片极小，一方面合并数据块的性能极高．比其他(非Bitmap)算法有更小的开销．每个group子块空间512Bit(即64Byte)刚好是Intel CPU的一个Cache Line的长度， 保证了连续申请和释放内存都在同一个Cache Line中动作，性能极高，同时这个长度也足够小，也可以尽快及时将全空的group子块彻底释放。这个长度设计在性能和内存浪费之间做了权衡。单个零bit的搜索性能很高，多个连续零bit的搜索开销要大很多，bit数越高开销也越大，因为要比较更多次数，以及更多的管理空间占用，更多的bit数修改和更新，设计最大支持连续500个零bit的搜索。因此，设计时如果有多个方案可选要尽量实现低bit数的设计，才会实现更高的性能。

使用原始的bitmap算法+Cacheline对齐+现代的Intrinsic函数可以在现代CPU上实现极高的性能。Bitmap算法主要是C++实现的，不需要bit对齐（不同于buddy伙伴分配算法），可以从任意bit点开始分配连续N bit内存，因此算法自身不会产生额外的碎片。由于现代CPU的多发射特性，相邻的多条指令处理同一个CacheLine的数据是高度并发的，可以在一个时钟周期内执行多个相邻指令（现代CPU 8并发即同一时钟周期并发8条指令），CPU内部的工作情况比几十年前的情况有变化，同一个Cacheline中连续多个数据的处理几乎不消耗CPU时间，大量CPU时间其实是消耗在等待内存到CPU L1的存取时间开销上，现代CPU受内存墙影响性能，Bitmap算法具备超低的管理内存占用（尤其是1bit的算法），因此在现代CPU上使用CacheLine对齐的bitmap算法，可以实现极高的性能。

总之Libhaisqlmalloc库设计时用古老的Bitmap原始算法，实现了低碎片率，但是需要执行更多的指令，理论上会减慢性能表现，设计上采用了理论上会减慢性能表现的技术方案来换取更低的碎片率和减少内存浪费。现代C++新技术采用多发射友好的数据结构和新的算法设计，在一个时钟周期内CPU可以并发执行更多的指令，实现极高的性能。

6）按bit数分配的外部碎片浪费极小。低bit数的总体设计方案，保证了外部碎片很小。具体如下：

128字节以下大小的内存分配全部是Bitmap 1bit的分配方式，这种方式保证了常用的小字节下，不存在外部碎片，每一个bit空间都可以得到分配，并且性能很高。

129字节-----3408字节其中一部分采用多Bit的分配方式，这种分配方式会产生少量外部碎片，多Bit的bit数（参见附录一）范围1---9bit,因此浪费碎片bit数的上限是8bit,相对总量512bit，这种浪费是很少的，单个非连续空间浪费低于1.56%。

3409字节----4KiB大小的内存分配全部是Bitmap 1bit的4KiB分配方式，这种方式不存在外部碎片，每一个bit空间都可以得到分配，并且性能极高。

(4KiB+1)----10KiB默认按照1K对齐（如果不是8KiB的情况），这种分配方式会产生少量外部碎片，bit数上限是10bit,因此浪费碎片bit数上限是9bit,相对总量512bit，这种浪费也是很少的，单个非连续空间浪费低于1.76%。

(10KiB+1)----2000KiB按照4KiB对齐，采用多Bit的分配方式，会产生外部碎片，并且这里是设计方案中容易出现外部碎片的地方，主要集中于大Bit数的分配数，因为检索空间的范围是从1bit到500bit个连续零，这个范围是很大的一个范围，但是由于malloc检索空闲空间是任意连续N bit的顺序全搜索模式，没有bit对齐的假设，并且常用bit数均是小bit数，超过32KiB(即8bit)的场景很少见，因此，实际使用中一般也很少出现外部碎片问题。

7）工作线程内部不会缓存2MiB大块内存，也不会预先分配给线程2MiB大块内存备用。

空闲的2MiB大块内存是所有线程共同管理的，因此，不存在线程内部浪费。每次4K连续块group对应的2MiB空闲块都会立即释放到一个多生产者多消费者MPMC队列中，不会在线程内部的数据结构中保存等待下次再用。不存在传统内存分配库线程内部的2MiB块浪费，也没有预分配给线程2M块数量不符合线程的实际需求造成浪费，不存在传统内存分配库中一个线程已经空闲的大块内存无法被其他线程利用的情况，极大的节约了内存，减少了碎片。对于短期内不会复用的多生产者多消费者MPMC队列中的空闲2MiB块，由后台线程智能的释放给系统。

8）Libhaisqlmalloc库的低缓存的设计，以牺牲性能表现为代价来换取低外部碎片和低内存浪费。Libhaisqlmalloc库在各级处理流程中都是尽快释放，不做额外缓存。每级动作流程中如果有缓存的话，那么就可以走短流程，实现更高性能，如果没有缓存，就必须要走长流程，向上级申请或者释放内存，减慢了性能，有个别时候会有多级长流程动作，导致性能下滑。另外一方面，由于Libhaisqlmalloc库的性能很高，每一级的开销也只有大约数十纳秒，即使是多级长流程总体开销也不大。

## 2MiB大块内存的统一进程缓存：

Libhaisqlmalloc库使用了现代C++无锁编程技术解决空闲2MiB大块数据管理问题。

工作线程的2MiB块空闲后会立即释放到按NUMA CPU node分组的容量512的两种（page和Hugepage2M)多生产者多消费者MPMC队列中，这种释放开销只有几十ns（二次atomic存取开销）。

这里是标准的多生产者多消费者场景，所有的线程既是生产者又是消费者，所以，这里采用了一种多生产者多消费者的MPMC队列，释放掉的2MiB块数据对所有线程共享，提前给队列预分配了3组(分别对应page和hugepage/ giantPage的情况)每组4096个数据存储结构，没有ABA问题（ 队列大小固定不会申请释放内存），是一种连续的Per CPU数据结构，入队/出队动作主要是atomic存取开销，性能很高。

释放掉的2MiB块存储在无锁队列中可以立即被同一NUMA CPU node的多个工作线程复用，重用开销也只有几十ns（atomic存取开销）。

常规page内存和hugepage / giantpage这样3组队列，每个都是多生产者多消费者MPMC队列，容量是4096个指针空间，对应8G内存的缓存，可以应对多数内存申请总量的抖动，存储的2MiB大块内存数据对所有线程透明共享， 一般情况下此容量已经足够满足需求，当出现超过队列长度的情况出现时，将会直接释放给系统。

MPMC队列设计上使用了per cpu 的分组技术，目前版本默认是32个分组，默认都是在当前cpu所在的分组中执行cas，只有少数场景下资源不够时，才找邻居分组借用一下使用cas，实现起来在工程上的表现就是高并发下几乎没有 cache coherence 的同步开销，接近完美，所以性能表现和稳定性很好，几乎没有cpu的并发冲突，简单的来说就是以多组无冲突的cas来替换一组有冲突的cas实现更高的性能和稳定性。工程上CAS循环封装了多层提升稳定性的代码，具体的CAS指令前后都有before和after的封装，循环的失败分支有超过 intel 指导意见数量的 mm pause(更慢更稳定)，也有检测 intel cpu CAS 内部错误的代码。总之，高并发组件的设计思路是: 愿意以降低性能为代价，也要换取可靠性。使用了大量高稳定措施，MPMC队列的性能依旧具备极高的多线程性能，设计的同一时钟周期下的多CPU的最大并发度是64（push/pop可同时并发），远好于传统的设计方案。

## 8.Thread Local算法：

目前C++ std / boost库以及 Linux pthread库均提供了Thread Local数据结构，这些库接口的Thread Local算法上存在优化空间，并且存在库依赖，因此重新实现了一个高速的没有库依赖条件的Thread Local专用数据结构，有更高的性能（数倍），以及良好的适应性(没有外部依赖)，避免了一些场景下的多个so库初始化顺序的兼容性问题，提高了内存库的兼容性。

Libhaisqlmalloc库的 Thread Local 算法采用了无锁RCU技术开发，每次查询只有1ns的开销，远高于C++ std / boost库以及 Linux pthread库的性能。

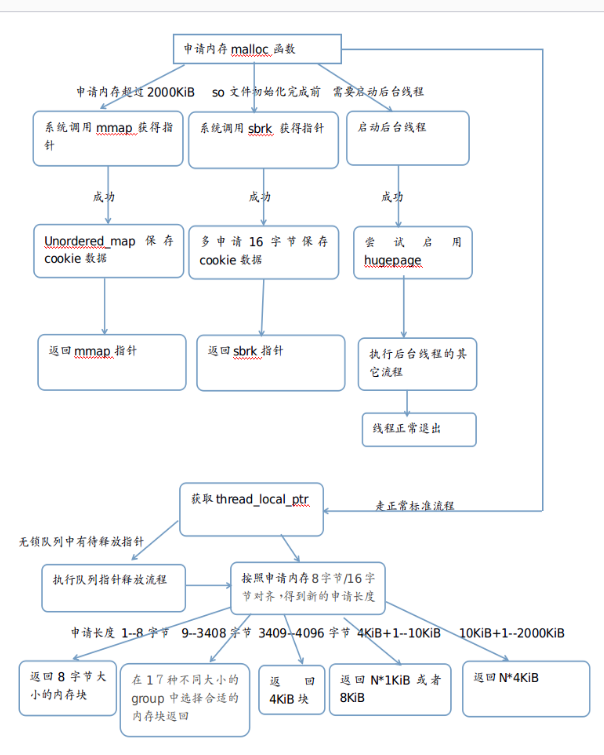
## 9. Wait Free Queue算法：

目前C++ Wait Frees算法已经流行，Libhaisqlmalloc库内部的Wait Free Queue非常复杂，使用了大量前沿技术和高稳定技术，主要的技术方案是：Per CPU和Per Thread的数据结构解决多生产者的并发问题和高稳定问题，Ring Buffer解决生产者和消费者的隔离问题，同步设计没有使用CAS指令，而是采用软件分布式的解决方案，将多CPU当做分布式场景，对CPU的同步不做严格要求，适配CPU之间的数据最终一致性场景，不要求atomic指令和fence指令对外的内存序的严格时序，即使由于某种原因（例如CPU内部网络拥塞）各CPU的数据出现短期不一致，但是只要能够在微秒级前达成最终一致（不超时），要求非常宽松，多数时候ns就可以完全同步，这部分Wait Free可以自动适配最终同步的结果（适应同步等待1ns到1微秒之间数据最终一致），适应最乱序的CPU设计，也适应云端共享CPU被其它用户影响或者广播流量拥塞的极端场景。Libhaisqlmalloc库使用了类似于传统数字通讯原理的设计思路，有做两层独立数据同步检测解决同步适配问题，pop 读数据之前做了两层CPU之间的同步检测，push写数据之前也做了两层CPU之间的同步检测，两次同步检测内存空间地址分别位于有效数据地址前后，都成功了才会继续，两层独立同步检测更慢更可靠。总之，类似于传统数据通讯的思路，保证通讯可靠和数据的同步，开销很大，相比不校验的方式降低了大约一倍的性能，但是提供了完善的可靠性保证。多生产者 Wait Free 队列有适配cpu之间的分布式最终一致性的保护代码，提供了非常好的数据稳定性。内存库的测试源码中最后有测试 Wait Free 实现跨线程释放,主要开销是用于通讯同步的开销。对于待释放指针这种数据，生产者实际写入了三倍的数据量，同步头+数据+同步尾，每个都是64bit的开销，同样生产者也需要读入三倍的数据量并做校验，降低了性能表现，但是提供了更高的稳定性，解决了极端场景下的小概率问题。

Libhaisqlmalloc库内部使用了两组无等待Wait Free Queue队列，用于内存的free跨线程释放，队列1最大支持8个工作线程，可以在同一时钟周期同时无等待并发8个生产者工作线程，性能非常好，主要瓶颈在消费者，因为只有一个消费者，push/pop都是数个ns。由于生产者过多过快导致队列全满或者累计的生产者线程超过8个线程时，将自动创建和使用无等待Wait\_free队列2，Wait\_free队列2支持无限个CPU有等待并发，可以在同一时钟周期内支持最多8个CPU同时无等待并发，支持无限数量CPU和无限长度队列空间，支持队列长度自增长，无等待队列2的效率略低于无等待队列1，两种类型的无等待队列共同组合起来处理跨线程释放内存，具备极高的并发性能和极强的灵活性，适应各种场景。两组无等待队列默认是动态创建，即需要时才依次创建和使用，以便节约内存空间，同时也支持预先创建和加载，具体见参数bool\_preload\_wait\_free\_queue1和bool\_preload\_wait\_free\_queue2的说明。

由于近年来软硬件技术的高速发展，目前Wait Free无等待技术已经非常成熟稳定可靠，有多种细分的软件技术方案解决每一个细分领域的技术难题，只需要灵活组合就好，Wait Free的性能比传统锁或者基于CAS指令的Lock Free有数量级的性能优势。

主要流程和开销情况：



以malloc动作为例, 平均开销是数个纳秒, 主要流程和开销如下:

1. so动态链接接口的开销。动态链接有比较大的开销。这部分是固有开销，无法被优化掉。
2. 申请大小为2000KiB以上，走系统mmap调用, 保存长度等cookie数据到一个传统锁保护的全局hash\_map的数据结构中，返回mmap指针。
3. 探测是否是so文件初始化尚未完成前就开始执行malloc申请的内存（即紧急场景的内存申请），这部分有一个专门的小模块单独处理，使用sbrk申请一大块内存，然后切成小片分配给用户。
4. 探测是否需要启动后台线程，如果需要，则启动后台线程。后台线程探测是否需要启用hugepage2M。如果需要，则探测系统是否支持hugepage2M，设置允许使用 hugepage2M 标志。
5. thread local ptr.get()的查询开销。
6. 探测跨线程释放无锁队列中是否有待释放指针，如果有则立即执行释放。
7. 按照数据块大小选择不同大小group的开销，有几个判断：

首先是按照申请内存的大小进行8字节/16字节对齐

1---8字节走8字节;

9字节---3408字节16字节对齐走一个跳表查询，会在1７种不同大小的group中选择合适的group，可选的有N个方案，优先少bit数，同时限制浪费低于２０％，平均浪费小于10%. (参见附录中的详细说明)；

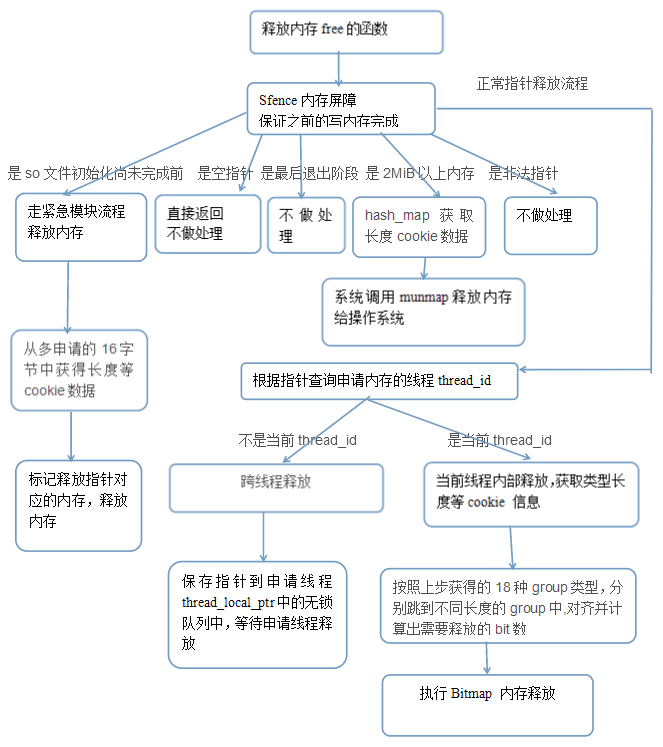
3409字节---4KiB的会申请4K；

（4KiB＋１）---10KiB的会在1KiB／４KiB的group中选择合适的group对齐，8KiB选择2\*4KiB, 其他选择N\*1KiB，例如4097字节选择５个1KiB的连续块，这样的浪费比较少；

10KiB---2000KiB的全部是4KiB的N倍数, 2000KiB就是连续500个4K块。

使用了预处理软件定义好方案，自动生成类似于跳表数据结构代码，实际代码就是几个简单的判断＋类似跳表的C++对象, 性能极高。

1. 各种大小的内存的内部分配流程就是按照待申请的bit数，执行Bitmap申请内存流程。Bitmap代码开销只占总体开销的一小部分。
2. 其它同步原语的开销。在一些数据结构的创建过程中有大量的同步原语的使用，例如用于与后台线程的数据交换等，另外Bitmap对于所有线程thread\_id相关cookie数据均进行了严格的数据写同步，具体参见后面第16节关于数据同步的说明。 总之，Libhaisqlmalloc库对指针的跨线程使用和释放是thread safe的。



以free动作为例，平均开销是数个纳秒, 主要流程和开销如下：

1. so动态链接接口的开销。动态链接有比较大的开销。这部分是固有开销，无法被优化掉。
2. 写同步的开销，在free内部执行写同步（使用sfence内存屏障），保证释放前与这个指针相关的写动作都已经完成。这部分是增加多线程稳定性的代码，用于同步CPU的内存数据。
3. 探测是否是so文件初始化尚未完成前就开始执行malloc申请的内存（紧急模块的）内存申请，探测方法是一个简单的地址范围判断（sbrk申请的地址段的范围），获得长度等cookie数据，执行紧急模块的释放流程。
4. 探测是否是空指针（空指针立即返回）。
5. 探测是否是已经处于进程退出的析构so内部全局数据结构的退出阶段，此阶段不作处理。
6. 探测是否是直接从系统mmap方式申请的2MiB以上大内存块，从一个传统锁保护的全局hash\_map的数据结构中获取长度等cookie数据，释放这种内存块直接走munmap 系统调用, 从一个全局hash\_map数据结构中获得长度等cookie数据的数据结构。
7. 探测是否是空指针或者非法指针，这两种情况均立即返回，不做处理。
8. 探测是否是跨线程指针释放(探测指针对应的group类型的数据结构中的thread\_id标志与当前线程的thread\_id标志是否一致)，对于A线程申请, B线程释放的指针，需要执行跨线程释放流程，通过指针数据cookie找到申请内存的线程A内部结构，直接保存指针到申请线程A内部的一个无锁队列中，队列中的指针由当时申请内存的线程A择机释放，实现了高性能的跨线程释放。
9. 探测指针数据cookie对应的group块大小，分别走不同块大小的group流程, 跳转到多种group中的一种类型， 错位1---(N-1)（N是group类型单bit对应的长度）个字节的非对齐的指针会自动对齐，支持少量错位容错，检索指针对应的待释放的bit数，将group内部的此bit数清零，实现避免重复释放。如果bit数为零，表示可能是重复释放的指针或非法指针，不做处理。
10. 按照待释放的bit数，执行Bitmap释放内存流程。Bitmap代码开销只占总体开销的一小部分。
11. 同步原语的开销。在一些数据结构的析构过程中有大量的同步原语的使用。

从以上的malloc/free的内部主要流程和开销情况可以看出，malloc / free的流程很复杂，考虑的情况很多，有大量的复杂流程，但是执行起来，平均开销这么小，所有流程的代码一定是经过高度优化的，并且一定采用了很多不同寻常的技术方案，解决了很多CPU的执行瓶颈问题，代码才能跑得如此之快。下面将简单介绍几个主要技术方案（黑科技）。

## 10.堆内存和栈内存使用HugePage和GiantPage：

Libhaisqlmalloc 库提供了对Hugepage的支持，可以大幅度提升应用程序的性能。Hugepage可以大幅度减少CPU TLB miss，增加应用程序的读写内存的速度。目前很多传统内存库不包含此功能点。

默认启用Hugepage2M，该配置可以关闭。只有在后台线程启用以后，并且系统提前配置了Hugepage页面，才会开始使用Hugepage2M, 这种设计方案主要是考虑Hugepage2M的内存申请都会实际分配物理内存，开销比较大，启用后台线程才能避免这种时延抖动。相关配置参数：见 /etc/libhaisqlmalloc.conf　中bool\_enable\_hugepage的说明。

在很多场景下配置Hugepgae2M可以增加内存库的性能表现， 以及用户程序的性能表现。因此， 建议用户配置一部分内存为Hugepage2M，根据不同的场景，可以采取不同的分配策略。建议在Linux4.X或者以上版本并且可用内存比较多的场景下，分配一半内存配置为Hugepage2M。由于目前Linux系统下Hugepage2M内存不参与swap，会导致虚拟内存更加紧张，在内存比较少比较紧张的场景下，减少分配Hugepage2M的总量，以便提供更好的swap性能。

提供Hugepage2M的支持方法:

方法一：修改/etc/sysctl.conf配置文件增加一行

vm.nr\_hugepages=2048

这里2048表示使用2048个Hugepage2M页面，即分配4G内存为Hugepage2M。

修改完配置文件后，需要重新启动生效，或者执行下列指令生效。

sudo sysctl -p

方法二：修改/etc/default/grub启动配置项，修改默认值

GRUB\_CMDLINE\_LINUX=""

为

GRUB\_CMDLINE\_LINUX="transparent\_hugepage=never default\_hugepagesz=2M hugepagesz=2M hugepages=2048"

这里2048表示使用2048个Hugepage2M页面，即分配4G内存为Hugepage2M。

修改完配置文件后，需要更新一次grub,然后下次启动才能生效。

sudo update-grub

下列指令可以显示Hugepage2M的使用情况

cat /proc/meminfo | grep Huge

AnonHugePages: 0 kB

HugePages\_Total: 3200

HugePages\_Free: 954

HugePages\_Rsvd: 30

HugePages\_Surp: 0

Hugepagesize: 2048 kB

Libhaisqlmalloc 库提供了对Giantpage的稳定支持，1GB页面的推广，可以大幅度提升大型应用程序的性能。giantpage可以大幅度减少CPU TLB miss，提升应用程序的读写内存的速度。目前多数传统内存库不包含此功能点。

默认启用GiantPgage，该配置可以关闭。默认只有使用的内存总量超过1GB，并且操作系统提前配置了Giantpage页面，才会开始使用GiantPage, 这种设计方案主要是考虑GiantPgage一次申请就是1GB内存，申请都会实际分配物理内存，开销比较大，因此实际上都是后台线程来与操作系统交互实现GiantPage内存的申请和释放，1GB的内存块一部分将会切成2M的小块，以方便其它线程使用，其它线程通过MPMC队列来获取和释放这部分2M块。相关配置参数：见 /etc/libhaisqlmalloc.conf　中bool\_enable\_giantpage的说明。

提供GiantPage的支持方法: 修改/etc/default/grub启动配置项，修改为

GRUB\_CMDLINE\_LINUX="hugepagesz=1g hugepages=4 default\_hugepagesz=2m hugepagesz=2m hugepages=2048 "

表示启用1G页面，占用4G(4个）, 默认页面是2M, 占用4G(2048个）

执行 sudo update-grub，然后重启动机器后才能生效。跑内存库测试代码，可以看到已经启用了1G页面。或者建立当前目录下的 libhaisqlmalloc.conf 增加2行

bool\_show\_malloc\_status\_on\_exit=true

表示程序正常退出前，显示内存库状态。默认是不显示

bool\_force\_enable\_giantpage=true

表示强制使用1G巨页。默认是占用内存总量超过1G以后，才使用1G巨页。

bool\_show\_numa\_map\_on\_exit=true

表示程序正常退出前，查看当前进程的操作系统提供的内存分配列表，可以看到4K/2M/1G页面 numa node等。默认是不显示。1GB页面显示范例：

7fe340000000 default file=/anon\_hugepage\040(deleted) huge anon=1 dirty=1 N0=1 kernelpagesize\_kB=1048576

2MB页面显示范例：

7fe38ce00000 default file=/anon\_hugepage\040(deleted) huge anon=1 dirty=1 N0=1 kernelpagesize\_kB=2048

Libhaisqlmalloc 库支持线程thread的栈内存默认自动使用hugePage页面或者Giantpage页面切下的内存块，默认thread栈内存内存块的大小是2MiB。当栈内存使用hugePage或者Giantpage后，进一步减少了CPU TLB miss ，提升了综合性能。具体见参数uint\_thread\_stack\_default\_size的说明，默认配置下，将尝试替换操作系统默认配置的8M page 栈空间。例外情况如下：当用户调用pthread\_create的第2个参数pthread\_attr非空的场景下，表示用户进程对pthread\_attr属性有读写可能对栈内存大小类型等的前后一致性有特殊要求，Libhaisqlmalloc 库将不会尝试替换栈空间为巨页，以免出现问题。对于用户进程的主线程也不会尝试替换栈空间为巨页。

## 11.支持物理内存提前分配：

Libhaisqlmalloc 库中的内存分配的物理地址分配，是默认配置为提前分配的。这样对于一些对性能有要求的场合可以提供更低的推迟，例如量化交易等，实现更高的性能表现。常规内存库的物理内存只有在内存使用中才能利用缺页中断分配物理内存，其实也是有不小的性能开销的。目前很多传统内存库不包含此功能点。

申请内存时是否实际分配4KiB page页面的物理地址，这个参数是一个动态变量，即程序会随着程序的进展自动修改此参数，也允许用户配置修改此变量，在下列场景下，此参数将会被自动修改为true：场景一在启用后台线程的过程中此参数将被自动修改为true（默认使用超过8M将启用后台线程，使用多线程也将自动启用后台线程），场景二是应用程序进入初始化的最早阶段(内存库so自身初始化完毕前)，内存分配总量就已经超过 uint\_use\_sbrk\_byte\_threshold\_force\_page\_prefault，说明这个程序是一个大型应用软件，此参数将被自动修改为true。

这个参数的设计目标是：对于小型程序按需分配物理内存，这样可以节约物理内存，对于大型应用程序或者使用内存较多的程序，在mmap系统调用后将整个大块2M都申请为物理内存，可以有更高的性能。

此功能点启用后，将会略微浪费空间，将会增加用户程序的性能。因为集中产生实际物理页面的分配，会增加一些程序的稳定性（例如：避免Facebook fstring早期版本末尾‘\0’推迟实现Bug）。由于很多时候是在后台线程中分配的物理内存，节约了应用程序缺页中断的开销，大大提升了性能，并且大幅度提升了稳定性，解决了其它内存库存在的缺页中断瞬间闪退的小概率问题。

注意：本内存库所有申请内存总量在2M一下的malloc返回的指针，均会在返回前实际分配物理内存，目的是为了彻底解决其它内存库存在的缺页中断瞬间闪退的小概率问题。

注意：对于申请内存总量在2M以上的场景，malloc受参数 bool\_enable\_malloc\_over\_2m\_force\_page\_prefault 控制，决定是否分配物理内存（默认是分配物理内存）。

注意：对于申请内存总量在2M以上的场景，realloc受参数 bool\_enable\_realloc\_over\_2m\_force\_page\_prefault 控制，决定是否分配物理内存（默认是分配物理内存）。

每次申请hugepage2M的页面都会实际分配物理地址，Hugepage2M没有修改配置的地方。一般情况下均是后台线程处理这种物理内存的分配，当后台线程忙不过来或者未被系统调度时，前端用户线程也会处理这种物理内存的分配。

## 12.支持释放内存自动清理CPU Cache：

Libhaisqlmalloc 库提供了释放内存自动清理CPU Cache的支持。目前很多传统内存库不包含此功能点。

Libhaisqlmalloc 库可以将释放内存从前端工作CPU的L1/L2/L3级Cache中彻底淘汰，提供了更多空闲CPU Cache给内存库和应用程序使用，增加了内存库和应用程序的性能。

一般情况下，彻底释放给系统的内存还会占用宝贵的CPU缓存空间。Libhaisqlmalloc 库的后台线程在彻底释放2MiB大块内存给系统前，主动通知CPU ,将这些缓存从CPU的一二三级缓存中彻底淘汰，这样就腾出了大量缓存空间给其它真正需要的内存，这项动作一般会提升内存分配库自身的性能10%，并且会大幅提升应用程序的性能。

后台线程目前只对释放的单个2MiB大块内存使用此功能，对其他大小的内存释放不执行此动作。由于后台线程动作时不能影响其它线程和CPU的正常工作，因此，不能使用直接清空全部cache的做法，只能使用一种从后台线程所在的CPU发起对前端多个工作CPU的cache shootdown的方法，逐个对每个CPU的L1/L2/L3 cache进行清理，通知全部CPU逐条清理cache，执行追踪检索释放CPU L1L2L3 cache的过程会消耗很多的后台线程CPU资源，增加后台线程的工作负荷。大约一个使用的2MiB大块对应的cache全部清理掉大约会消耗掉550微秒(Intel CPU 9400F 2.9G主频下)的后台线程工作时间。

后台线程默认启用此功能，具体参见配置uint\_max\_wait\_nanosecond\_before\_clear\_cpu\_cache的说明。

## 13.支持线程与CPU自动绑定：

Libhaisqlmalloc 库提供了对线程与CPU的自动绑定的支持。目前很多传统内存库不包含此功能点。

现代CPU的数量很高，因此，高性能的工作场景一般要求绑定线程到少数几个（一般1---2个）CPU上，以便减少无效CPU切换，减少CPU Cache因为在多个CPU之间的颠簸．但是，在编程的实践中很多程序并没有绑定CPU,使用此功能后大约可以增加多线程程序性能，因此，我们采用了下列措施，实现了一种工作线程与CPU的自动绑定算法，相关配置参数：见 /etc/libhaisqlmalloc.conf　中bool\_enable\_auto\_bind\_cpu的说明。

工作线程自动绑定CPU．此功能启用后，将会在线程申请内存时自动检查是否已经指定了运行的CPU范围，如果没有指定，将自动执行线程自动绑定CPU：首先探测CPU 是否支持超线程，如果是支持超线程的CPU, 将自动将工作线程绑定到一个物理CPU上．如果不支持超线程，将自动将工作线程绑定到同一NUMA node下的1---2个相邻的物理CPU上。NUMA 的内存绑定都是动态获取NUMA node参数的，不会预设thread与NUMA node之间存在关联关系。

## 14.支持NUMA node内存绑定:

Libhaisqlmalloc 库提供了对NUMA内存绑定node的支持。目前很多传统内存库不包含此功能点。

NUMA是一种多槽位CPU计算机系统的工作方式，每个槽位的CPU有独立的内存总线，多CPU槽位通过总线连接起来的SMP系统。由于每个槽位的CPU有独立的内存总线，因此可以获得更高的内存带宽，有更好的性能．目前高端服务器一般都是支持NUMA方式。

Libhaisqlmalloc 对于NUMA的管理模式提供了一种新的工作模式。默认关闭NUMA内存绑定．原因是NUMA 这种工作模式的机器总量比较少（对硬件有很高的要求），并且低版本的Linux系统于NUMA的支持不完善，默认启用NUMA可能会导致一些版本下崩溃, 因此需要用户修改配置选项后才能启用NUMA相关功能．相关配置参数：见 /etc/libhaisqlmalloc.conf　中bool\_enable\_numa\_memory\_bind的说明。

尝试启用NUMA前建议首先请检查numactl -s的显示结果，下面是不支持NUMA node的情况，只有一个NUMA node, 这种就不需要打开NUMA 内存绑定功能。

numactl -s

policy: default

preferred node: current

physcpubind: 0 1 2 3 4 5

cpubind: 0

nodebind: 0

membind: 0

只有多个NUMA node节点存在的情况下，才有启用NUMA绑定的价值。对于支持NUMA的情况，如果bool\_enable\_numa\_memory\_bind 配置为允许NUMA内存绑定，那么就会执行下列动作：当某个线程malloc()时，线程优先在当前申请内存的CPU所在的NUMA node（如果当前NUMA node没有内存条，将选相邻NUMA node）上分配内存；如果探测到当前优先NUMA node上没有足够的可用内存，将不使用线程优先分配策略，此时，由系统配置的默认NUMA分配策略分配内存．这种NUMA node自动绑定方式，在很多场景下会提供比Linux系统自带的libnuma库有更灵活更高性能的内存分配策略．因为这种分配策略是基于工作线程所在的NUMA node优先的分配策略，而传统的libnuma库是基于进程的NUMA分配策略．在内存足够并且用量均衡的理想情况下，每个CPU上跑的线程都是在当前NUMA node下申请的内存，不会出现线程在node A上跑，使用的却是node B上的内存的情况。

在目前常用的Linux5.X版本下，NUMA的相关系统调用还不能稳定支持Hugepage2M，因此，Libhaisqlmalloc库关闭了Hugepage2M 的NUMA内存绑定．就是说当前的分配策略是：当申请的2M块是Hugepage的时候，将不执行NUMA node内存绑定．只有申请的是常规Page内存，并且配置启用NUMA内存绑定功能，才会优先执行基于线程的NUMA node内存绑定，失败将执行系统默认NUMA策略。

## 15. 低碎片的系统调用:

系统调用：Libhaisqlmalloc 库内部在申请内存和释放内存的时候，只使用mmap / munmap，没有使用madvice函数（脏牛漏洞就是利用madvise(map, XXX, MADV\_DONTNEED)实现入侵，Facebook jemalloc 等库都有调用），so文件初始化成功后没有sbrk和brk申请释放内存（so文件初始化成功前也只存在一次2MiB的sbrk调用），有利于系统进行页面归集整理，减少系统的内部碎片。

超低的系统碎片：Libhaisqlmalloc 库管理内存范围是1Byte---2MiB, (传统内存分配库一般只负责管理128KiB以下的大小)，所有的内存申请都是将2MiB的大块内存切成小内存实现内存分配。Libhaisqlmalloc 库对系统的内存申请除了管理内存申请有偶然个别少量64KiB的申请外，其他内存申请都是2MiB或者N\*2MiB，对系统的内存管理很友好，系统的碎片率很低，提高了系统的整体工作效率。

## 16.默认提供thread safe性和数据同步保证:

Libhaisqlmalloc 库内部默认提供了完整的thread safe性保证，对应用程序没有同步或者thread safe性的要求。目前很多传统内存库不包含此功能点。Libhaisqlmalloc 库设计思路是以牺牲性能表现为代价，以CPU所有顺序性保证均不存在的严苛的数据存取条件，以严格的多线程编程规范，换取完整彻底的thread safe性和数据同步保证，具备CPU移植性。Libhaisqlmalloc 库在多线程下很稳定，代价就是在一些场景下减慢了性能。

由于Libhaisqlmalloc 库使用的C++库基础算法很快，具备极高的性能，容易暴露出多线程冲突和数据同步问题，因此Libhaisqlmalloc 库对应用程序没有同步要求，对库自身有严格的thread safe和数据同步要求，代码设计时假定Intel X86系列CPU的所有顺序性的保证均不存在，按照苛刻的CPU顺序性进行代码设计（好处是方便今后移植到其它型号的CPU上），malloc / free内部过程大量频繁的使用同步原语(内存屏障sfence / mfence / lfence / atomic / mutex等)，更频繁的同步，以及大量的探测代码，减慢了性能表现，但是分布在各个CPU的cache数据也更同步，也顺便增加了使用Libhaisqlmalloc 库的应用程序的整体多线程稳定性，提供thread safe保证。

Libhaisqlmalloc 库在多线程下很稳定，实现了很多额外（远远超过多线程编程规范要求）的thread safe和数据同步保证。C++多线程编程一直是程序员容易犯错误的难点。Libhaisqlmalloc 库为了避免多线程BUG，设计代码时采用了一种防御性编程规范，以解决此难题。宁可在一些场景下减慢性能表现，也要预防多线程BUG。

1. 在设计多线程代码时，按照苛刻的CPU顺序性进行代码设计。假定Intel X86系列CPU的所有顺序性的保证均不存在，方便移植到各种CPU下。
2. 大量的探测。假定所有的线程同步措施都有漏洞，封装探测代码，只要能够探测的都设法进行探测（例如std::mutex和Spin\_lock锁也封装了一层复杂的同步探测代码和错误检测修复代码），处处探测，步步设防。
3. 有疑点的数据存取，均使用同步原语，宁可多用重复，也不可放过，更频繁的使用内存屏障。
4. spin\_lock锁内部设计超过一定次数会自动执行os yield，因此支持实时系统。
5. 使用高并发度的wait\_free的无锁无等待架构，基于cas指令的lock\_free无锁架构只允许用于并发冲突较小的场合（减少cas指令的广播风暴）。
6. 所有需要锁保护的变量或者对象，全部在名称前面标记locking\_，表示提醒程序员这个变量或对象需要锁的额外保护。
7. Wait free 算法中，push/pop之前都做了两层同步检测，只有两次同步检测都成功了，才会继续。例如：push实际写入的数据是：同步头+数据+同步尾，对于跨线程释放的指针，实际push写入了3倍的数据，pop也读入了3倍的数据进行分析，对于已经处理完毕的数据，pop提供反向同步功能，会写入同步头和同步尾，push实际写入前会再此检测同步，只有反向的两次同步检测都成功了，才会继续执行push。这种复杂的设计方案类似于传统通讯原理，大幅度降低了性能，但是更可靠。
8. CAS循环中封装了多层提升稳定性的代码，具体的CAS指令前后都有before和after的封装，循环的失败分支有超过 intel 指导意见数量的 mm pause(更慢更稳定)，也有检测 intel cpu CAS 内部错误的代码，cas循环函数内有调用多种不同的其它函数来解决多种小概率的稳定性问题，总之，通过更慢更复杂的适配 intel cpu 的设计解决了CAS循环函数的工程上的一些坑，以降低性能为代价换取更高的可靠性，减少小概率事件的发生。

总之，Libhaisqlmalloc 库设计思路是在多线程领域的代码以thread safe为首要问题，宁可浪费各种开销，宁可在这些地方慢一点，冗余次数多一点，也要确保thread safe。在其它不涉及并发的代码上，在这些地方不留余地的进行各种优化，实现高性能。这样兼顾的总体设计方案，既保证了thread safe，又实现了高性能。

## 17.使用了多种无锁算法，减少并发冲突，提升效率

1. 使用thread\_local, 每个线程有自己独立的操作对象，这种存取方式非常高效的，没有锁。使用thread\_local，每个线程内部有一个小内存池子，存放到thread\_local中，多数情况下，可以直接从这个线程内部的小池子分配内存， 没有并发冲突。
2. per CPU的方式，就是说数据对象按CPU分组，这种方式的锁冲突也是非常小（例如执行中线程被切换），虽然代码中使用了锁，但是实质上几乎等同于无锁，因为几乎没有锁冲突，效率也非常高，接近无锁的性能，设计代码也非常简单，获得当前CPU编号，作为vector的offset, 获取按CPU分组的对象 。用于thread\_local的管理内存，从per cpu的池子中分配，减少了并发冲突。
3. lock\_free技术，这种主要是cas指令来实现。一个MPMC多生产者多消费者mpmc队列，用于待释放的2M块的复用和释放。mpmc 设计上主要使用了per cpu 的分组技术，就是默认都是在当前cpu所在的分组中执行cas，只有少数极端场景下资源不够时，才找周围的邻居分组借用一下使用cas，实现起来在工程上的表现就是高并发下几乎没有 cache coherence 的同步开销，接近完美，所以性能表现和稳定性也非常好，几乎没有cpu的并发冲突，简单的来说就是以多组无冲突的cas来替换一组有冲突的cas以便实现更高的性能和稳定性。CAS获取得到的数据在这里是一个指针，指针所指向的2M内存，也有复杂的同步保护代码提供额外的同步。
4. wait\_free技术，这种是目前性能最高的并发算法，主要是内存屏障fence指令和atomic等来实现，使用了较常见的ring buffer架构。主要用于跨线程释放的队列数据结构，A线程申请的内存，生成class X, 例如:shared\_ptr, 最后是B线程释放这个对象，free时B线程检测指针对应的cookie 数据获取thread id, 判断是跨线程的释放，就传给A线程的wait free queue，是以生产者的方式push进队列。 A线程作为消费者可以择机pop出队列（下次malloc/free执行时）释放指针。这种方式性能很高， 并发开销很小。

## 18.默认分配物理内存，增加内存连续读写性能

Libhaisqlmalloc 库为了增加用户程序的内存连续读写的性能和程序的稳定性，默认启用了实际分配物理内存，因此malloc/realloc申请到的内存默认都是已经分配物理内存的内存空间，这样，读写动作不会触发缺页中断，因此memset之类的连续内存动作，由于不会被缺页中断所打断，因此有较大幅度的性能增加。此功能点默认启用，也可以通过设置参数关闭，具体的参数，见 bool\_enable\_malloc\_force\_page\_prefault / bool\_enable\_realloc\_force\_page\_prefault / bool\_enable\_malloc\_over\_2m\_force\_page\_prefault / bool\_enable\_realloc\_over\_over\_2m\_force\_page\_prefault 这些参数的配置说明。其中，对于2M以上的内存申请，可以单独关闭分配物理内存，见bool\_enable\_malloc\_over\_2m\_force\_page\_prefault / bool\_enable\_realloc\_over\_over\_2m\_force\_page\_prefault的说明。

## 19.现代C++是高性能的关键技术：

Libhaisqlmalloc库使用了一些高性能组件，代码模块是用C++写的，经过反复测试和调整，具备较高的性能表现。软件工程的实践中，很多的性能增加，来自于现代C++技术。计算机从1970年只有50年的发展时间，近一二十年来计算机的硬件技术发生了进步，软件追随硬件而变，新一代C++软件发挥硬件新特性，发挥硬件的各种潜力（例如CPU多发射各种新指令集各种多线程高并发友好的新的数据结构与算法），时代变了，因此，以前写的古老的C/C++库用现代C++技术重写一遍，很多都可以获得大幅度的性能增加，这是时代的红利，只是需要投入人力资源投入时间去做而已，just do it。

下面举2个现代C++库的例子：

范例1：跨线程内存释放是一种常见的场景，传统算法是使用传统锁保护下的队列。跨线程内存释放是一个很典型的多生产者单消费者MPSC的工作场景, Libhaisqlmalloc库使用了一种多生产者单消费者MPSC无锁队列C++模板库，库代码大约数千行代码，也是thread safe的，应用层代码几行就实现了高性能的解决方案，Libhaisqlmalloc库使用的是一种自研的经过多线程防御性编程加固过的C++模板库，比传统方案快数倍，使用常见的Ring Buffer技术方案扩展实现，代码总体很复杂，不仅考虑了大量corner case的情况，数千行源码，还有冗余的同步措施。

测试代码后面有一段代码测试跨线程释放，可以测试看到线程数N=1---16之间，性能是逐步增加的，因为没有并发冲突并且是无等待架构，超过16个线程并发后性能才略有下降。现代C++的无锁无等待队列可以提供更好的性能表现，更低的推迟时间。即使代码设计中有大量的冗余同步和探测措施，Libhaisqlmalloc库的跨线程内存释放也比传统旧方案快N倍。测试代码中16线程并发跨线程释放测试1000万次8字节内存申请和释放，平均每组malloc + 指针传给其它线程 + 其它线程push进并发队列 + 主线程pop出并发队列 + 主线程free这五个动作累计大约27ns, 平均每个动作只有5ns的开销，远超传统方案。

范例2：重写了一部分常用std库组件，提供了更好的性能。以vector为例说明如下：

下面这种是内存库中的一个class中的一个vector对象， 把C++的vector , unique ptr, allocator 等组件都全部重写了，实现更多种类的组件，实现std库扩展，提升了性能，提供了各种无锁或锁冲突少的组件，实现高性能。一行代码定义一个vector容器，如下所示：

vector\_allocator<unique\_ptr\_include\_alloc<Memory\_Bitmap\_page\_piece, Memory\_pool\_without\_lock\_128byte>, Memory\_allocator<Malloc\_page\_part\_cpu, Realloc\_page\_part\_cpu, Free\_page\_part\_cpu> > d\_vt\_up\_normal;

这是一个装unique\_ptr（unique\_ptr\_include\_alloc）的vector容器，其中：Memory\_allocator< Malloc\_page\_part\_cpu, Realloc\_page\_part\_cpu, Free\_page\_part\_cpu>　是自定义实现的一组按CPU编号的多组页面对齐的内存分配组件，由于是per CPU的，按CPU分区，所以内部几乎是无锁的，这里作为vector（vector\_allocator）的内存分配器，Malloc\_page\_part\_cpu／Realloc\_page\_part\_cpu／Free\_page\_part\_cpu这些都是函数对象，可以随时定义，随时使用，灵活组合和定义，比std::allocator的定义灵活的多。预先定义了一些常用的Memory\_allocator，都是基于函数对象的实现，很灵活，函数对象调用起来没有额外开销。Memory\_pool\_without\_lock\_128byte 是一个线程内部的内存分配池，彻底无锁，只能分配128Byte, 这里作为unique\_ptr（unique\_ptr\_include\_alloc） 的内存分配器，性能极高。

另外由于使用了比较新的C++算法，重写vector后一些常用函数的性能比std库有大幅增加，例如删除 / push\_back等，另外提供了一些更高性能的扩展函数，例如乱序删除等。

## 20. Libhaisqlmalloc 库的性能测试：

Libhaisqlmalloc 库是一个新的内存分配库，具备很高的稳定性，可以增加应用程序的性能，减少内存浪费和碎片，更有效使用内存。Libhaisqlmalloc 库使用了大量类似于Intel dpdk / mpdk / spdk 的高性能C++技术（这些库对比同领域的旧传统方案都有较大性能改进，网上有大量的公开资料和说明），因此，Libhaisqlmalloc 库在 “所有大小的内存申请和释放” 的性能也比以前设计的旧内存库Linux malloc库 / Google tcmalloc库 / Facebook jemalloc库快。使用现代C++软件技术重新写了一个可测试可下载可运行的新型的内存分配库。

Libhaisqlmalloc 库不是一个单纯以高性能为目标的内存分配库，代码的健壮性，内存的低浪费和低碎片，thread safe，数据的同步等因素，都比性能更重要，程序设计时有大量的设计目标都是以大幅度牺牲性能为代价，以放弃性能表现来换取更重要的其它方面的技术指标。即使这样，全新设计的Libhaisqlmalloc 库的性能也比传统内存分配库有较大的性能改进。

Libhaisqlmalloc 库的架构思路，设计语言，碎片设计等更接近Google tcmalloc库的理念，都是C++,都是thread\_local cache的，都是低碎片设计，因此主要与Google tcmalloc库进行对比。Libhaisqlmalloc 库加权平均性能比传统库Google tcmalloc库快很多（参见附录的测试结果）, 其中4KiB以上内存申请的性能较好，测试方案是一个动态链接的外部接口的性能测试（so接口有开销），测试代码本身还有额外的开销（代码中有测试，但是没有扣除这块的额外开销），真实的性能还要略高一些。

具体的测试性能的细节可以参考附录二给出的测试情况，可以使用附录三给出的方法编译测试代码，调整测试代码的行参数进行更多的详细的对比测试。（测试软件的用法请见第五章第1节的说明。）

配套的性能测试源码也是我们开发的，要求编译器支持C++11，开源协议是Apache2协议，商业友好，已经在开源中国的网站上开放源码和二进制执行程序，具体的源代码二进制so文件和用户手册可以从下列地址下载：[https://gitee.com/wlmqgzm/](https://gitee.com/wlmqgzm/test_malloc_use_so)

Libhaisqlmalloc 库自身是一个商业软件，使用需要支付费用，不是开源软件。只对用户提供二进制的动态链接库so文件。目前只支持Linux系统，今后有足够需求的话会移植到其它系统，也会支持更多的CPU类型。

# 第六章 内存库的对外函数接口

Libhaisqlmalloc 库目前接管了下列内存相关的函数接口实现内存分配，能够满足多数场景下的内存分配的需求，也可以兼容多数Linux应用。

内存库的所有对外接口：

对外的C函数接口支持如下：

void\* malloc( size\_t sizet\_size\_in ) \_\_THROW

void free( void\* ptr\_void\_in ) \_\_THROW

void\* realloc( void\* ptr\_void\_in, size\_t sizet\_size\_in ) \_\_THROW

void \*reallocarray( void \*ptr\_void\_in, size\_t sizet\_nmemb\_in, size\_t sizet\_size\_in ) \_\_THROW

void\* calloc( size\_t n, size\_t sizet\_size\_in ) \_\_THROW

void cfree( void\* ptr\_void\_in ) \_\_THROW

void\* valloc( size\_t sizet\_size\_in ) \_\_THROW

void\* pvalloc( size\_t sizet\_size\_in ) \_\_THROW

void\* aligned\_alloc( size\_t sizet\_alignment\_in, size\_t sizet\_size\_in ) \_\_THROW

void\* memalign( size\_t sizet\_alignment\_in, size\_t sizet\_size\_in ) \_\_THROW

int posix\_memalign( void\*\* ptr\_ptr\_void\_in, size\_t sizet\_alignment\_in, size\_t sizet\_size\_in ) \_\_THROW

void malloc\_stats( void ) \_\_THROW

int mallopt( int int\_key, int int\_value ) \_\_THROW

size\_t malloc\_size( void\* p ) \_\_THROW

int malloc\_trim( size\_t sizet\_pad\_in ) \_\_THROW

size\_t malloc\_usable\_size( void\* ptr\_void\_in ) \_\_THROW

相关主要函数简单介绍：

函数原型：void free(void\* p)函数功能：释放指针p指向的内存块p必须是由malloc、calloc或realloc返回的指针

calloc：申请一段内存空间，并将这段内存空间初始化为0

realloc：改变之前申请的一段内存块的大小

malloc函数接口默认与Linux的接口兼容，但是也有区别，说明如下：

头文件： #include <stdlib.h>

函数原型：void\* malloc(size\_t size)

返回值：成功：返回一个指针，该指针指向一块有size个字节的内存块。如果size是0，返回8字节的指针。

失败：将会执行std::terminate()，原因在于目前的系统下都是虚拟内存，可以申请很多虚拟内存，几乎没有可能在申请内存时出现内存不足的情况，就是说几乎不可能出现内存申请失败的情况，实际内存不足的出错都是在缺页中断时。

这种模式可以从源头上增加应用程序的性能。原因在于：应用程序在每次malloc/new/realloc后，应用程序的代码中可以不用检查返回的指针是否为空，所有返回值一定不是nullptr，这样就大幅度简化了内存申请/对象创建的流程，简化了代码设计，提升了性能。由于也不需要用catch thow 捕捉内存不足的情况，所有的函数都可以设置为noexecpt（可以增加应用函数性能）。这种高性能内存申请模式，适合从头开发的高性能应用程序的开发，因为只有在代码中大幅度简化流程，才可以真正增加性能。

库增加了下列额外的扩展函数：

void malloc\_show\_config( void ) noexcept;

size\_t malloc\_all\_size( void ) noexcept;

void malloc\_set\_limit\_callback( const size\_t sizet\_limit\_size\_in, void ( \*ptr\_callback\_function\_in )( size\_t ) ) noexcept;

void malloc\_version( void ) noexcept;

unsigned int malloc\_page\_type( void\* ptr\_void\_in ) noexcept;

void malloc\_multi( const unsigned int uint\_size\_in, const unsigned int uint\_count\_in, void\*\* ptr\_ptr\_void\_out ) noexcept;

void free\_multi( void\*\* ptr\_ptr\_void\_in, const unsigned int uint\_count\_in ) noexcept;

void memzero( void\* ptr\_void\_in, const size\_t sizet\_size\_in ) noexcept;

void\* try\_realloc( void\* ptr\_void\_in, size\_t sizet\_new\_size\_in ) noexcept;

void\* malloc\_less\_2000k( const unsigned int uint\_size\_in ) noexcept;

void\* malloc\_less\_4k( const unsigned int uint\_size\_in ) noexcept

void\* malloc\_less\_3280( const unsigned int uint\_size\_in ) noexcept

void\* malloc\_less\_128( const unsigned int uint\_size\_in ) noexcept

void free\_this\_thread\_less\_2000k( void\* ptr\_void\_in ) noexcept;

void free\_this\_thread\_less\_4k( void\* ptr\_void\_in ) noexcept;

void free\_this\_thread\_less\_3280( void\* ptr\_void\_in ) noexcept;

void free\_this\_thread\_less\_128( void\* ptr\_void\_in ) noexcept;

void free\_less\_2000k( void\* ptr\_void\_in ) noexcept;

void\* thread\_local\_get( void\* ptr\_void\_key\_in ) noexcept;

void thread\_local\_set( void\* ptr\_void\_key\_in, void\* ptr\_void\_value\_in, void(\*destructor\_in)(void\*) ) noexcept;

void thread\_local\_free( void\* ptr\_void\_key\_in ) noexcept;

1）void malloc\_version( void ) noexcept;

用于显示软件的版权信息。

1. size\_t malloc\_all\_size( void ) noexcept;

用于获取目前已经申请的全部内存大小。

1. Void malloc\_set\_limit\_callback( const size\_t sizet\_limit\_size\_in, void ( \*ptr\_callback\_function\_in )( size\_t ) ) noexcept;

用于设置一个使用内存总量的限制值，当到达这个限制值时，将会调用函数指针ptr\_callback\_function\_in所指的用户函数，callback函数的参数size\_t表示超出阈值的字节数。这个函数实现通知用户线程已经超过内存使用量上限。Callback()在执行时间是内部锁的保护下，因此不会出现并发冲突，任意时刻只会有一个callback()在执行，是thread safe的，callback()的执行时机是在一次malloc()函数的调用中，此时没有返回申请的内存，因此要求callback函数的内部定义比较简单，不能占用较长时间，callback()内部不能有malloc()内存申请的行为，适合简单的设置一个标志后，由定时器执行用户自己的内存回收或者收缩功能。

1. void malloc\_show\_config( void ) noexcept;

用于当前显示配置参数的设置情况。配置文件是libhaisqlmalloc.conf， 具体的配置参数可以通过这个函数显示出来，方便进行参数配置调试。

1. void malloc\_multi( const unsigned int uint\_size\_in, const unsigned int uint\_count\_in, void\*\* ptr\_ptr\_void\_out ) noexcept;

用于批量申请内存块，一次可以申请uint\_count\_in个uint\_size\_in大小的内存块(uint\_size\_in的大小限制小于等于2000KiB),申请得到的内存块保存到ptr\_ptr\_void\_out[0]开始的连续数据结构中。这些内存块可以随时释放，可以单独free释放，这样就比传统内存池有更大的灵活性, 与普通malloc申请的内存块一致，可以混用。由于是批处理申请同样大小的内存，只需要一次判断就跳到相应大小的内部函数中连续进行内存块生成，因此这个函数的性能很高，远快于uint\_count\_in次调用malloc的性能，并且多数情况下这些申请的内存块是连续的，可以提高生成对象的扫描性能，可以用于生成内存池和批量生成对象的场合等。

1. void free\_multi( void\*\* ptr\_ptr\_void\_in, const unsigned int uint\_count\_in ) noexcept;

1.27及以后版本开始支持此功能点。

用于批量释放内存块，一次可以释放uint\_count\_in个同样大小的内存块（单个内存块大小限制小于等于2000KiB），这些内存块的大小必须相同（否则会有不可预料的后果），等待释放的内存块在ptr\_ptr\_void\_in[0]开始的连续数据结构中。主要是用于malloc\_multi申请的内存块，也可以用于同样大小的内存池的集中释放，支持跨线程的内存释放, 支持跨线程和本地线程内存的混用。由于是批处理释放同样大小的内存，只需要一次判断就跳到相应大小的内部函数中连续进行内存块的释放，因此这个函数的性能很高，远快于uint\_count\_in次调用free的性能，可以用于内存池释放和批量释放对象的场合等。

内存库的稳定性的选择方面需要做出抉择：是否为了兼容越界Bug多分配几个字节。

Linux早期系统中一些应用代码存在越界bug, 判断是否要重新申请更多的内存的判断出现了错误，为了避免这种低级bug, 保证有越界bug的程序也可以稳定运行，工程上的内存库的临时解决办法就是malloc多分配几个字节，不用改一行代码，就可以解决很多越界bug, 这个是工程上妥协的手段。现在Linux的各种程序的代码质量已经很高了，很少有越界bug。因此，Libhaisqlmalloc 库默认按实际内存申请量分配内存，减少了内存的浪费，同时也支持修改配置参数bool\_malloc\_size\_plus实现多分配8字节的功能，具体参见配置参数bool\_malloc\_size\_plus的说明。当这个选项启用时，将会在申请1字节--(2M-8)字节时多申请8字节的内存。

下面是Linux 系统默认的ptmalloc的malloc()后执行malloc\_usable\_size()的返回结果，可以看到每个内存申请都多分配了几个字节

Log: check\_malloc\_usable\_size() malloc() malloc\_size=8, malloc\_usable\_size=24

Log: check\_malloc\_usable\_size() malloc() malloc\_size=16, malloc\_usable\_size=24

Log: check\_malloc\_usable\_size() malloc() malloc\_size=24, malloc\_usable\_size=24

Log: check\_malloc\_usable\_size() malloc() malloc\_size=16, malloc\_usable\_size=24

Log: check\_malloc\_usable\_size() malloc() malloc\_size=32, malloc\_usable\_size=40

Log: check\_malloc\_usable\_size() malloc() malloc\_size=48, malloc\_usable\_size=56

Log: check\_malloc\_usable\_size() malloc() malloc\_size=64, malloc\_usable\_size=72

Log: check\_malloc\_usable\_size() malloc() malloc\_size=80, malloc\_usable\_size=88

Log: check\_malloc\_usable\_size() malloc() malloc\_size=96, malloc\_usable\_size=104

Log: check\_malloc\_usable\_size() malloc() malloc\_size=112, malloc\_usable\_size=120

Log: check\_malloc\_usable\_size() malloc() malloc\_size=128, malloc\_usable\_size=136

Log: check\_malloc\_usable\_size() malloc() malloc\_size=144, malloc\_usable\_size=152

Log: check\_malloc\_usable\_size() malloc() malloc\_size=160, malloc\_usable\_size=168

Log: check\_malloc\_usable\_size() malloc() malloc\_size=176, malloc\_usable\_size=184

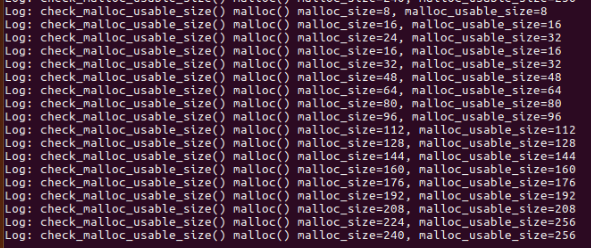
Log: check\_malloc\_usable\_size() malloc() malloc\_size=192, malloc\_usable\_size=200

Log: check\_malloc\_usable\_size() malloc() malloc\_size=208, malloc\_usable\_size=216

Log: check\_malloc\_usable\_size() malloc() malloc\_size=224, malloc\_usable\_size=232

Log: check\_malloc\_usable\_size() malloc() malloc\_size=240, malloc\_usable\_size=248

下面是Libhaisqlmalloc 库的malloc()后执行malloc\_usable\_size()的默认情况下的返回结果，可以看到每个内存申请都刚好适配。



类似的Google tcmalloc 也是按实际内存申请量分配内存，每个内存申请都刚好适配。

Libhaisqlmalloc 库支持int malloc\_trim( size\_t sizet\_pad\_in )

使用时参数sizet\_pad\_in在函数内部其实并没有使用，这个限制值不会起作用。Libhaisqlmalloc 库中该函数执行后将尽可能释放所有可以释放的内存,。

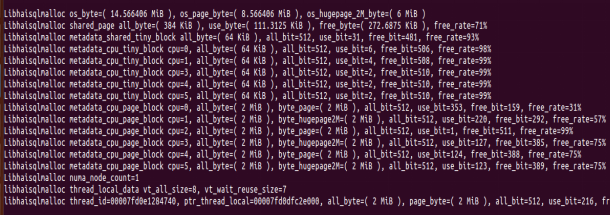
Libhaisqlmalloc 库提供了显示内部分配情况的接口malloc\_stats，与其他内存库的调用方式兼容。

用法见下面的定义:

#include <malloc.h>

void malloc\_stats(void);

Libhaisqlmalloc 库中的内存分配是很复杂的一个体系，这个体系首先区分了两大类内存，一种是自身消耗的内存，用于管理数据，这种内存我们称为Metadata, 第二种是给用户分配的内存，这种内存以NUMA node和Thread Local为核心，从系统申请的块大小是2M块，再切成小内存分配给用户。下面是一个malloc\_stats()调用后的显示案例：



Libhaisqlmalloc os\_byte=( 14.566406 MiB ), os\_page\_byte=( 8.566406 MiB ), os\_hugepage\_2M\_byte=( 6 MiB )

Libhaisqlmalloc sbrk\_shared\_page all\_byte=( 384 KiB ), use\_byte=( 111.3125 KiB ), free\_byte=( 272.6875 KiB ), free\_rate=71%

Libhaisqlmalloc metadata\_shared\_tiny\_block all\_byte=( 64 KiB ), all\_bit=512, use\_bit=31, free\_bit=481, free\_rate=93%

Libhaisqlmalloc metadata\_cpu\_tiny\_block cpu=0, all\_byte=( 64 KiB ), all\_bit=512, use\_bit=6, free\_bit=506, free\_rate=98%

Libhaisqlmalloc metadata\_cpu\_tiny\_block cpu=1, all\_byte=( 64 KiB ), all\_bit=512, use\_bit=4, free\_bit=508, free\_rate=99%

Libhaisqlmalloc metadata\_cpu\_tiny\_block cpu=3, all\_byte=( 64 KiB ), all\_bit=512, use\_bit=2, free\_bit=510, free\_rate=99%

Libhaisqlmalloc metadata\_cpu\_tiny\_block cpu=4, all\_byte=( 64 KiB ), all\_bit=512, use\_bit=2, free\_bit=510, free\_rate=99%

Libhaisqlmalloc metadata\_cpu\_tiny\_block cpu=5, all\_byte=( 64 KiB ), all\_bit=512, use\_bit=2, free\_bit=510, free\_rate=99%

Libhaisqlmalloc metadata\_cpu\_page\_block cpu=0, all\_byte=( 2 MiB ), byte\_page=( 2 MiB ), all\_bit=512, use\_bit=353, free\_bit=159, free\_rate=31%

Libhaisqlmalloc metadata\_cpu\_page\_block cpu=1, all\_byte=( 2 MiB ), byte\_hugepage2M=( 2 MiB ), all\_bit=512, use\_bit=220, free\_bit=292, free\_rate=57%

Libhaisqlmalloc metadata\_cpu\_page\_block cpu=2, all\_byte=( 2 MiB ), byte\_page=( 2 MiB ), all\_bit=512, use\_bit=1, free\_bit=511, free\_rate=99%

Libhaisqlmalloc metadata\_cpu\_page\_block cpu=3, all\_byte=( 2 MiB ), byte\_hugepage2M=( 2 MiB ), all\_bit=512, use\_bit=127, free\_bit=385, free\_rate=75%

Libhaisqlmalloc metadata\_cpu\_page\_block cpu=4, all\_byte=( 2 MiB ), byte\_page=( 2 MiB ), all\_bit=512, use\_bit=124, free\_bit=388, free\_rate=75%

Libhaisqlmalloc metadata\_cpu\_page\_block cpu=5, all\_byte=( 2 MiB ), byte\_hugepage2M=( 2 MiB ), all\_bit=512, use\_bit=123, free\_bit=389, free\_rate=75%

Libhaisqlmalloc numa\_node\_count=1

libhaisqlmalloc thread\_local\_data vt\_all\_size=8, vt\_wait\_reuse\_size=7

libhaisqlmalloc thread\_id=00007fd0e1284740, ptr\_thread\_local=00007fd0dfc2e000, all\_byte=( 2 MiB ), page\_byte=( 2 MiB ), all\_bit=512, use\_bit=216, free\_bit=296, free\_rate=57%

在上面的例子中，os\_byte表示向系统(OS)申请的内存总量，os\_page\_byte表示向系统(OS)申请的4K Page的内存总量，os\_hugepage\_2M\_byte表示向系统(OS)申请的Hugepage2M的内存总量。

sbrk\_shared\_page表示通过sbrk方式申请的共享的一块内存（默认大小为4MiB），这块内存是紧急内存块，是so文件初始化尚未完成前就开始执行malloc申请的内存，主要用于so文件还没有初始化完毕，以及so文件已经开始析构的场景下，也叫紧急分配内存块，这部分内存很特殊。这部分内存是使用sbrk方式一次性分配的内存，这部分内存在常规场景下只需要128KiB就足够满足需求，在X Windows的应用下有个别大型程序消耗高达MiB(例如：Codeblocks)，因此，统一配置为4MiB。

all\_byte表示全部申请内存， use\_byte表示已经使用的内存，free\_byte表示空闲的内存，free\_rate表示空闲百分比。

Metadata用于Libhaisqlmalloc 库的内部管理数据，有三种类型：

类型一metadata\_shared\_tiny\_block是多CPU共享的用来分配128字节大小的管理空间。此数据结构自身的128字节来自于从shared\_page中的分配。

类型二metadata\_cpu\_tiny\_block是每CPU独立的用来分配128字节大小的管理空间，此数据结构自身的128字节来自于类型一。

类型三metadata\_cpu\_page\_block是每CPU独立的用来分配按页面对齐的管理空间，此数据结构自身的128字节来自于类型二。

numa\_node\_count 表示 NUMA node的数量

numa\_node\_2m\_block\_queue　表示进程单个NUMA node下所有CPU共享的２M块的MPMC队列

vt\_all\_size 表示thread\_local容器（含备用数据）的数据量是多少(Vt是vector的缩写)

vt\_wait\_reuse\_size 表示thread\_local备用容器的的数据量是多少(备用数据的相关配置参数：见 /etc/libhaisqlmalloc.conf　中uint\_vt\_thread\_local\_wait\_reuse\_limit\_size的说明)。

thread\_id 是c/c++获得的thread id值，是一个与线程相关的地址值，等效于std::thread::thread\_id,当thread\_id显示为零时表示该数据结构对应的线程已经退出，但是有内存没有释放（并不表示出现了内存泄露，有可能一些指针所指的对象正由其他工作线程持有），对应的也有一些少量的2M块没有释放掉。这些thread\_local数据结构是可以被反复复用的，因此，有一个thread\_local的池子用于复用。

ptr\_thread\_local表示指向thread\_local的指针。

all\_byte 表示所有字节数。 page\_byte 表示常规页面的字节数。

all\_bit 表示所有的Bitmap的bit数。

use\_bit 表示已经使用的bit数。

free\_bit 表示空闲的bit数。

free\_rate 表示空闲的百分比。

# 内存库的购买和备份恢复

Libhaisqlmalloc内存库是一个自研的非开源的内存分配库，花费了商业公司的大量开发时间进行开发和各种优化，使用了大量前沿现代C++技术，研发了大量新的C++模板库，使用了大量的新算法，自有知识产权的源码总行数高达4.2万行，代码量很大，代码量超过很多常见内存分配库的规模，性能极高，技术研发周期很长，研发时间数年，研发成本很高。

考虑软件市场的实际情况，该软件定价也很低，希望所有用户能够支持一下中国基础软件的研发。获取的收入将继续投入到其它重要基础软件的研发。

下面是非注册版运行时的显示内容

Notice: Libhaimalloc Version 1.01 Copyright by haisql.cn Publish Date: 2021.01

Notice: Basic version. max\_support\_numa\_node=8, max\_support\_cpu=136

Notice: unregister version. Please buy the software ( libhaisqlmalloc.so ) and register it.

Notice: link http://buy.haisql.cn/libhaisqlmalloc/?str\_mac=00e07096a61a&str\_device=enp2s0&ulong\_copy\_id=1610526900&uint\_copy\_id=599826717&uint\_version=101&ulong\_create\_id=1101081971070163&ulong\_verify\_id=0&str\_run=ls

Notice: This software ( libhaisqlmalloc.so ) test time has 23 hours 15 minutes 11 seconds

购买链接是第4行的那个http链接。购买后将获取一个libhaisqlmalloc.conf 的配置文件，这个配置文件放置到 /etc/libhaisqlmalloc.conf 或者 当前libhaisqlmalloc.so文件所在的目录下即可。用户可以修改此libhaisqlmalloc.conf文件中的其他内容，以便优化性能，见第四章配置文件和配置选项的说明。

配置文件的一个范例：

str\_mac=00e07096a61a

str\_device=enp2s0

ulong\_copy\_id=1605876329

uint\_copy\_id=156155068

uint\_version=101

ulong\_create\_id=1101081971070163

ulong\_verify\_id=12739876336654934

这些配置内容主要是依据用户机器的网卡的MAC地址，so文件自身参数生成的一系列编码, 标识了每一个so文件的拷贝，用于防盗版，每一份拷贝都有单独的验证码。

如果用户重新装了系统，从备份或者其他机器或者其他压缩格式中恢复的（例如同一台机器的同样ext4文件系统格式的），可能是从其他网站中下载的，那么就需要重新注册和认证。此备份将会被认为是一个新的拷贝，将自动获得100\*24小时的测试时间，作为测试版本运行（测试版本将会在每次使用Libhaisqlmalloc库时向stderr输出NOTICE版本信息），测试超过100\*24小时将自动abort退出，会影响用户程序的正常使用，因此，从其他机器或者其他压缩格式中恢复备份后，请及时重新注册和认证。

已经付费的用户，在同样的机器上（通过网卡的MAC地址识别），默认自动获得10次重新注册的机会，可以直接获取新的验证码和新的libhaisqlmalloc.conf文件，直接下载下来覆盖掉旧配置文件即可。非注册版默认获得100\*24小时的测试时间，超过100\*24小时，运行将自动退出。

关于内存库的其他限制条件：

目前用户直接获取的Libhaisqlmalloc库都是基本版。

该版本默认支持8个NUMA node(一般等于CPU槽位数，PC和低端服务器只有一个默认的NUMA node)。

该版本默认支持64个CPU, 一般中低端服务器均低于64个CPU。

对于超过8个NUMA node或超过64个CPU的需求，这些需要ADVANCE VERSION, 请联系：[www.haisql.cn](http://www.haisql.cn)

微信联系方式：微信wlmqgzm。

邮件联系方式：haisql@sina.com.cn

对于大批量购买或者定制需求，也欢迎联系，谢谢您的支持。

# 第八章 附录

**附录一：内部碎片2的设计**

申请不同大小的内存，因为内部块对齐而产生碎片，这种碎片称为内部碎片。

16-3408Byte内部碎片2的设计方案如下所示。其中size表示申请的内存大小，align表示对齐的字节数，type表示group的类型号，bit\_size表示需要申请的bit\_size数，bit\_size\*align大于等于申请的内存大小size, waste\_byte表示按照align对齐后内部碎片2浪费的字节数，waste\_rate表示内部碎片2浪费的字节数占获取内存的百分比。

size=16, align=16, type=0, bit\_size=1, waste\_byte=0, waste\_rate=0

size=32, align=32, type=1, bit\_size=1, waste\_byte=0, waste\_rate=0

size=48, align=48, type=2, bit\_size=1, waste\_byte=0, waste\_rate=0

size=64, align=64, type=3, bit\_size=1, waste\_byte=0, waste\_rate=0

size=80, align=80, type=4, bit\_size=1, waste\_byte=0, waste\_rate=0

size=96, align=96, type=5, bit\_size=1, waste\_byte=0, waste\_rate=0

size=112, align=112, type=6, bit\_size=1, waste\_byte=0, waste\_rate=0

size=128, align=128, type=7, bit\_size=1, waste\_byte=0, waste\_rate=0

size=144, align=144, type=8, bit\_size=1, waste\_byte=0, waste\_rate=0

size=160, align=160, type=9, bit\_size=1, waste\_byte=0, waste\_rate=0

size=176, align=176, type=10, bit\_size=1, waste\_byte=0, waste\_rate=0

size=192, align=192, type=11, bit\_size=1, waste\_byte=0, waste\_rate=0

size=208, align=208, type=12, bit\_size=1, waste\_byte=0, waste\_rate=0

size=224, align=256, type=13, bit\_size=1, waste\_byte=32, waste\_rate=12.5

size=240, align=256, type=13, bit\_size=1, waste\_byte=16, waste\_rate=6.25

size=256, align=256, type=13, bit\_size=1, waste\_byte=0, waste\_rate=0

size=272, align=144, type=8, bit\_size=2, waste\_byte=16, waste\_rate=5.55556

size=288, align=144, type=8, bit\_size=2, waste\_byte=0, waste\_rate=0

size=304, align=160, type=9, bit\_size=2, waste\_byte=16, waste\_rate=5

size=320, align=160, type=9, bit\_size=2, waste\_byte=0, waste\_rate=0

size=336, align=176, type=10, bit\_size=2, waste\_byte=16, waste\_rate=4.54545

size=352, align=176, type=10, bit\_size=2, waste\_byte=0, waste\_rate=0

size=368, align=192, type=11, bit\_size=2, waste\_byte=16, waste\_rate=4.16667

size=384, align=192, type=11, bit\_size=2, waste\_byte=0, waste\_rate=0

size=400, align=208, type=12, bit\_size=2, waste\_byte=16, waste\_rate=3.84615

size=416, align=208, type=12, bit\_size=2, waste\_byte=0, waste\_rate=0

size=432, align=512, type=14, bit\_size=1, waste\_byte=80, waste\_rate=15.625

size=448, align=512, type=14, bit\_size=1, waste\_byte=64, waste\_rate=12.5

size=464, align=512, type=14, bit\_size=1, waste\_byte=48, waste\_rate=9.375

size=480, align=512, type=14, bit\_size=1, waste\_byte=32, waste\_rate=6.25

size=496, align=512, type=14, bit\_size=1, waste\_byte=16, waste\_rate=3.125

size=512, align=512, type=14, bit\_size=1, waste\_byte=0, waste\_rate=0

size=528, align=176, type=10, bit\_size=3, waste\_byte=0, waste\_rate=0

size=544, align=192, type=11, bit\_size=3, waste\_byte=32, waste\_rate=5.55556

size=560, align=192, type=11, bit\_size=3, waste\_byte=16, waste\_rate=2.77778

size=576, align=192, type=11, bit\_size=3, waste\_byte=0, waste\_rate=0

size=592, align=208, type=12, bit\_size=3, waste\_byte=32, waste\_rate=5.12821

size=608, align=208, type=12, bit\_size=3, waste\_byte=16, waste\_rate=2.5641

size=624, align=208, type=12, bit\_size=3, waste\_byte=0, waste\_rate=0

size=640, align=256, type=13, bit\_size=3, waste\_byte=128, waste\_rate=16.6667

size=656, align=256, type=13, bit\_size=3, waste\_byte=112, waste\_rate=14.5833

size=672, align=256, type=13, bit\_size=3, waste\_byte=96, waste\_rate=12.5

size=688, align=256, type=13, bit\_size=3, waste\_byte=80, waste\_rate=10.4167

size=704, align=256, type=13, bit\_size=3, waste\_byte=64, waste\_rate=8.33333

size=720, align=256, type=13, bit\_size=3, waste\_byte=48, waste\_rate=6.25

size=736, align=256, type=13, bit\_size=3, waste\_byte=32, waste\_rate=4.16667

size=752, align=256, type=13, bit\_size=3, waste\_byte=16, waste\_rate=2.08333

size=768, align=256, type=13, bit\_size=3, waste\_byte=0, waste\_rate=0

size=784, align=208, type=12, bit\_size=4, waste\_byte=48, waste\_rate=5.76923

size=800, align=208, type=12, bit\_size=4, waste\_byte=32, waste\_rate=3.84615

size=816, align=208, type=12, bit\_size=4, waste\_byte=16, waste\_rate=1.92308

size=832, align=208, type=12, bit\_size=4, waste\_byte=0, waste\_rate=0

size=848, align=176, type=10, bit\_size=5, waste\_byte=32, waste\_rate=3.63636

size=864, align=1024, type=15, bit\_size=1, waste\_byte=160, waste\_rate=15.625

size=880, align=1024, type=15, bit\_size=1, waste\_byte=144, waste\_rate=14.0625

size=896, align=1024, type=15, bit\_size=1, waste\_byte=128, waste\_rate=12.5

size=912, align=1024, type=15, bit\_size=1, waste\_byte=112, waste\_rate=10.9375

size=928, align=1024, type=15, bit\_size=1, waste\_byte=96, waste\_rate=9.375

size=944, align=1024, type=15, bit\_size=1, waste\_byte=80, waste\_rate=7.8125

size=960, align=1024, type=15, bit\_size=1, waste\_byte=64, waste\_rate=6.25

size=976, align=1024, type=15, bit\_size=1, waste\_byte=48, waste\_rate=4.6875

size=992, align=1024, type=15, bit\_size=1, waste\_byte=32, waste\_rate=3.125

size=1008, align=1024, type=15, bit\_size=1, waste\_byte=16, waste\_rate=1.5625

size=1024, align=1024, type=15, bit\_size=1, waste\_byte=0, waste\_rate=0

size=1040, align=208, type=12, bit\_size=5, waste\_byte=0, waste\_rate=0

size=1056, align=176, type=10, bit\_size=6, waste\_byte=0, waste\_rate=0

size=1072, align=256, type=13, bit\_size=5, waste\_byte=208, waste\_rate=16.25

size=1088, align=256, type=13, bit\_size=5, waste\_byte=192, waste\_rate=15

size=1104, align=256, type=13, bit\_size=5, waste\_byte=176, waste\_rate=13.75

size=1120, align=256, type=13, bit\_size=5, waste\_byte=160, waste\_rate=12.5

size=1136, align=256, type=13, bit\_size=5, waste\_byte=144, waste\_rate=11.25

size=1152, align=256, type=13, bit\_size=5, waste\_byte=128, waste\_rate=10

size=1168, align=256, type=13, bit\_size=5, waste\_byte=112, waste\_rate=8.75

size=1184, align=256, type=13, bit\_size=5, waste\_byte=96, waste\_rate=7.5

size=1200, align=256, type=13, bit\_size=5, waste\_byte=80, waste\_rate=6.25

size=1216, align=256, type=13, bit\_size=5, waste\_byte=64, waste\_rate=5

size=1232, align=256, type=13, bit\_size=5, waste\_byte=48, waste\_rate=3.75

size=1248, align=256, type=13, bit\_size=5, waste\_byte=32, waste\_rate=2.5

size=1264, align=256, type=13, bit\_size=5, waste\_byte=16, waste\_rate=1.25

size=1280, align=512, type=14, bit\_size=3, waste\_byte=256, waste\_rate=16.6667

size=1296, align=512, type=14, bit\_size=3, waste\_byte=240, waste\_rate=15.625

size=1312, align=512, type=14, bit\_size=3, waste\_byte=224, waste\_rate=14.5833

size=1328, align=512, type=14, bit\_size=3, waste\_byte=208, waste\_rate=13.5417

size=1344, align=512, type=14, bit\_size=3, waste\_byte=192, waste\_rate=12.5

size=1360, align=512, type=14, bit\_size=3, waste\_byte=176, waste\_rate=11.4583

size=1376, align=512, type=14, bit\_size=3, waste\_byte=160, waste\_rate=10.4167

size=1392, align=512, type=14, bit\_size=3, waste\_byte=144, waste\_rate=9.375

size=1408, align=512, type=14, bit\_size=3, waste\_byte=128, waste\_rate=8.33333

size=1424, align=512, type=14, bit\_size=3, waste\_byte=112, waste\_rate=7.29167

size=1440, align=512, type=14, bit\_size=3, waste\_byte=96, waste\_rate=6.25

size=1456, align=512, type=14, bit\_size=3, waste\_byte=80, waste\_rate=5.20833

size=1472, align=512, type=14, bit\_size=3, waste\_byte=64, waste\_rate=4.16667

size=1488, align=512, type=14, bit\_size=3, waste\_byte=48, waste\_rate=3.125

size=1504, align=512, type=14, bit\_size=3, waste\_byte=32, waste\_rate=2.08333

size=1520, align=512, type=14, bit\_size=3, waste\_byte=16, waste\_rate=1.04167

size=1536, align=512, type=14, bit\_size=3, waste\_byte=0, waste\_rate=0

size=1552, align=256, type=13, bit\_size=7, waste\_byte=240, waste\_rate=13.3929

size=1568, align=256, type=13, bit\_size=7, waste\_byte=224, waste\_rate=12.5

size=1584, align=256, type=13, bit\_size=7, waste\_byte=208, waste\_rate=11.6071

size=1600, align=256, type=13, bit\_size=7, waste\_byte=192, waste\_rate=10.7143

size=1616, align=256, type=13, bit\_size=7, waste\_byte=176, waste\_rate=9.82143

size=1632, align=256, type=13, bit\_size=7, waste\_byte=160, waste\_rate=8.92857

size=1648, align=256, type=13, bit\_size=7, waste\_byte=144, waste\_rate=8.03571

size=1664, align=256, type=13, bit\_size=7, waste\_byte=128, waste\_rate=7.14286

size=1680, align=256, type=13, bit\_size=7, waste\_byte=112, waste\_rate=6.25

size=1696, align=256, type=13, bit\_size=7, waste\_byte=96, waste\_rate=5.35714

size=1712, align=1024, type=15, bit\_size=2, waste\_byte=336, waste\_rate=16.4062

size=1728, align=1024, type=15, bit\_size=2, waste\_byte=320, waste\_rate=15.625

size=1744, align=1024, type=15, bit\_size=2, waste\_byte=304, waste\_rate=14.8438

size=1760, align=1024, type=15, bit\_size=2, waste\_byte=288, waste\_rate=14.0625

size=1776, align=1024, type=15, bit\_size=2, waste\_byte=272, waste\_rate=13.2812

size=1792, align=1024, type=15, bit\_size=2, waste\_byte=256, waste\_rate=12.5

size=1808, align=1024, type=15, bit\_size=2, waste\_byte=240, waste\_rate=11.7188

size=1824, align=1024, type=15, bit\_size=2, waste\_byte=224, waste\_rate=10.9375

size=1840, align=1024, type=15, bit\_size=2, waste\_byte=208, waste\_rate=10.1562

size=1856, align=1024, type=15, bit\_size=2, waste\_byte=192, waste\_rate=9.375

size=1872, align=1024, type=15, bit\_size=2, waste\_byte=176, waste\_rate=8.59375

size=1888, align=1024, type=15, bit\_size=2, waste\_byte=160, waste\_rate=7.8125

size=1904, align=1024, type=15, bit\_size=2, waste\_byte=144, waste\_rate=7.03125

size=1920, align=1024, type=15, bit\_size=2, waste\_byte=128, waste\_rate=6.25

size=1936, align=1024, type=15, bit\_size=2, waste\_byte=112, waste\_rate=5.46875

size=1952, align=1024, type=15, bit\_size=2, waste\_byte=96, waste\_rate=4.6875

size=1968, align=1024, type=15, bit\_size=2, waste\_byte=80, waste\_rate=3.90625

size=1984, align=1024, type=15, bit\_size=2, waste\_byte=64, waste\_rate=3.125

size=2000, align=1024, type=15, bit\_size=2, waste\_byte=48, waste\_rate=2.34375

size=2016, align=1024, type=15, bit\_size=2, waste\_byte=32, waste\_rate=1.5625

size=2032, align=1024, type=15, bit\_size=2, waste\_byte=16, waste\_rate=0.78125

size=2048, align=1024, type=15, bit\_size=2, waste\_byte=0, waste\_rate=0

size=2064, align=256, type=13, bit\_size=9, waste\_byte=240, waste\_rate=10.4167

size=2080, align=256, type=13, bit\_size=9, waste\_byte=224, waste\_rate=9.72222

size=2096, align=256, type=13, bit\_size=9, waste\_byte=208, waste\_rate=9.02778

size=2112, align=256, type=13, bit\_size=9, waste\_byte=192, waste\_rate=8.33333

size=2128, align=256, type=13, bit\_size=9, waste\_byte=176, waste\_rate=7.63889

size=2144, align=512, type=14, bit\_size=5, waste\_byte=416, waste\_rate=16.25

size=2160, align=512, type=14, bit\_size=5, waste\_byte=400, waste\_rate=15.625

size=2176, align=512, type=14, bit\_size=5, waste\_byte=384, waste\_rate=15

size=2192, align=512, type=14, bit\_size=5, waste\_byte=368, waste\_rate=14.375

size=2208, align=512, type=14, bit\_size=5, waste\_byte=352, waste\_rate=13.75

size=2224, align=512, type=14, bit\_size=5, waste\_byte=336, waste\_rate=13.125

size=2240, align=512, type=14, bit\_size=5, waste\_byte=320, waste\_rate=12.5

size=2256, align=512, type=14, bit\_size=5, waste\_byte=304, waste\_rate=11.875

size=2272, align=512, type=14, bit\_size=5, waste\_byte=288, waste\_rate=11.25

size=2288, align=512, type=14, bit\_size=5, waste\_byte=272, waste\_rate=10.625

size=2304, align=512, type=14, bit\_size=5, waste\_byte=256, waste\_rate=10

size=2320, align=512, type=14, bit\_size=5, waste\_byte=240, waste\_rate=9.375

size=2336, align=512, type=14, bit\_size=5, waste\_byte=224, waste\_rate=8.75

size=2352, align=512, type=14, bit\_size=5, waste\_byte=208, waste\_rate=8.125

size=2368, align=512, type=14, bit\_size=5, waste\_byte=192, waste\_rate=7.5

size=2384, align=512, type=14, bit\_size=5, waste\_byte=176, waste\_rate=6.875

size=2400, align=512, type=14, bit\_size=5, waste\_byte=160, waste\_rate=6.25

size=2416, align=512, type=14, bit\_size=5, waste\_byte=144, waste\_rate=5.625

size=2432, align=512, type=14, bit\_size=5, waste\_byte=128, waste\_rate=5

size=2448, align=512, type=14, bit\_size=5, waste\_byte=112, waste\_rate=4.375

size=2464, align=512, type=14, bit\_size=5, waste\_byte=96, waste\_rate=3.75

size=2480, align=512, type=14, bit\_size=5, waste\_byte=80, waste\_rate=3.125

size=2496, align=512, type=14, bit\_size=5, waste\_byte=64, waste\_rate=2.5

size=2512, align=512, type=14, bit\_size=5, waste\_byte=48, waste\_rate=1.875

size=2528, align=512, type=14, bit\_size=5, waste\_byte=32, waste\_rate=1.25

size=2544, align=512, type=14, bit\_size=5, waste\_byte=16, waste\_rate=0.625

size=2560, align=1024, type=15, bit\_size=3, waste\_byte=512, waste\_rate=16.6667

size=2576, align=1024, type=15, bit\_size=3, waste\_byte=496, waste\_rate=16.1458

size=2592, align=1024, type=15, bit\_size=3, waste\_byte=480, waste\_rate=15.625

size=2608, align=1024, type=15, bit\_size=3, waste\_byte=464, waste\_rate=15.1042

size=2624, align=1024, type=15, bit\_size=3, waste\_byte=448, waste\_rate=14.5833

size=2640, align=1024, type=15, bit\_size=3, waste\_byte=432, waste\_rate=14.0625

size=2656, align=1024, type=15, bit\_size=3, waste\_byte=416, waste\_rate=13.5417

size=2672, align=1024, type=15, bit\_size=3, waste\_byte=400, waste\_rate=13.0208

size=2688, align=1024, type=15, bit\_size=3, waste\_byte=384, waste\_rate=12.5

size=2704, align=1024, type=15, bit\_size=3, waste\_byte=368, waste\_rate=11.9792

size=2720, align=1024, type=15, bit\_size=3, waste\_byte=352, waste\_rate=11.4583

size=2736, align=1024, type=15, bit\_size=3, waste\_byte=336, waste\_rate=10.9375

size=2752, align=1024, type=15, bit\_size=3, waste\_byte=320, waste\_rate=10.4167

size=2768, align=1024, type=15, bit\_size=3, waste\_byte=304, waste\_rate=9.89583

size=2784, align=1024, type=15, bit\_size=3, waste\_byte=288, waste\_rate=9.375

size=2800, align=1024, type=15, bit\_size=3, waste\_byte=272, waste\_rate=8.85417

size=2816, align=1024, type=15, bit\_size=3, waste\_byte=256, waste\_rate=8.33333

size=2832, align=1024, type=15, bit\_size=3, waste\_byte=240, waste\_rate=7.8125

size=2848, align=1024, type=15, bit\_size=3, waste\_byte=224, waste\_rate=7.29167

size=2864, align=1024, type=15, bit\_size=3, waste\_byte=208, waste\_rate=6.77083

size=2880, align=1024, type=15, bit\_size=3, waste\_byte=192, waste\_rate=6.25

size=2896, align=1024, type=15, bit\_size=3, waste\_byte=176, waste\_rate=5.72917

size=2912, align=1024, type=15, bit\_size=3, waste\_byte=160, waste\_rate=5.20833

size=2928, align=1024, type=15, bit\_size=3, waste\_byte=144, waste\_rate=4.6875

size=2944, align=1024, type=15, bit\_size=3, waste\_byte=128, waste\_rate=4.16667

size=2960, align=1024, type=15, bit\_size=3, waste\_byte=112, waste\_rate=3.64583

size=2976, align=1024, type=15, bit\_size=3, waste\_byte=96, waste\_rate=3.125

size=2992, align=1024, type=15, bit\_size=3, waste\_byte=80, waste\_rate=2.60417

size=3008, align=1024, type=15, bit\_size=3, waste\_byte=64, waste\_rate=2.08333

size=3024, align=1024, type=15, bit\_size=3, waste\_byte=48, waste\_rate=1.5625

size=3040, align=1024, type=15, bit\_size=3, waste\_byte=32, waste\_rate=1.04167

size=3056, align=1024, type=15, bit\_size=3, waste\_byte=16, waste\_rate=0.520833

size=3072, align=1024, type=15, bit\_size=3, waste\_byte=0, waste\_rate=0

size=3088, align=512, type=14, bit\_size=7, waste\_byte=496, waste\_rate=13.8393

size=3104, align=512, type=14, bit\_size=7, waste\_byte=480, waste\_rate=13.3929

size=3120, align=512, type=14, bit\_size=7, waste\_byte=464, waste\_rate=12.9464

size=3136, align=512, type=14, bit\_size=7, waste\_byte=448, waste\_rate=12.5

size=3152, align=512, type=14, bit\_size=7, waste\_byte=432, waste\_rate=12.0536

size=3168, align=512, type=14, bit\_size=7, waste\_byte=416, waste\_rate=11.6071

size=3184, align=512, type=14, bit\_size=7, waste\_byte=400, waste\_rate=11.1607

size=3200, align=512, type=14, bit\_size=7, waste\_byte=384, waste\_rate=10.7143

size=3216, align=512, type=14, bit\_size=7, waste\_byte=368, waste\_rate=10.2679

size=3232, align=512, type=14, bit\_size=7, waste\_byte=352, waste\_rate=9.82143

size=3248, align=512, type=14, bit\_size=7, waste\_byte=336, waste\_rate=9.375

size=3264, align=512, type=14, bit\_size=7, waste\_byte=320, waste\_rate=8.92857

size=3280, align=512, type=14, bit\_size=7, waste\_byte=304, waste\_rate=8.48214

size=3296, align=512, type=14, bit\_size=7, waste\_byte=288, waste\_rate=8.03571

size=3312, align=512, type=14, bit\_size=7, waste\_byte=272, waste\_rate=7.58929

size=3328, align=512, type=14, bit\_size=7, waste\_byte=256, waste\_rate=7.14286

size=3344, align=512, type=14, bit\_size=7, waste\_byte=240, waste\_rate=6.69643

size=3360, align=512, type=14, bit\_size=7, waste\_byte=224, waste\_rate=6.25

size=3376, align=512, type=14, bit\_size=7, waste\_byte=208, waste\_rate=5.80357

size=3392, align=512, type=14, bit\_size=7, waste\_byte=192, waste\_rate=5.35714

size=3408, align=512, type=14, bit\_size=7, waste\_byte=176, waste\_rate=4.91071

附录二：性能测试建议

方法用例：export LD\_PRELOAD=/mnt/guo/cpp/libhaisqlmalloc/bin/Release/libhaisqlmalloc.so

建议：建议配置一半内存为hugepage2M,见前面第26页的 sysctl.conf 的配置说明。

附录三：测试代码main.cpp的编译方法

g++ -Wall -fexceptions -march=core2 -O2 -std=c++11 -c ./main.cpp -o ./main.o

g++ -o ./test\_malloc\_use\_so ./main.o -s -lpthread

附录四：man的兼容性解决办法

在一些比较低版本的man下，例如：man 2.8.3（man -V可以显示版本号）。使用本内存分配库后，man 命令查看任务的函数或命令时，显示崩溃以及信息如下。man: command exited with status 159。原因在于man 2.8.3版本与linux 的SECCOMP 机制和本内存库内部使用了更多的扩展系统调用有关。需要设置环境变量，关闭掉man的SECCOMP功能，方案是：export MAN\_DISABLE\_SECCOMP=1

man 2.9 以后的版本不会有这个问题