

MySQL内存问题分析利器--Jemalloc

原创 宋利兵 MySQL代码研究 2025年05月06日 20:55 北京

内存泄漏、内存占用高是MySQL中较常见的问题，这些问题的排查非常依赖完善的内存监控信息。MySQL的Performance Schema中提供了内存的监控信息，PFS的内存统计信息粒度比较粗，很难精准的定位到问题代码。

```
mysql> select * from memory_global_by_current_bytes limit 2;
+-----+-----+-----+-----+-----+-----+-----+
| event_name | current_count | current_alloc | current_avg_alloc | high_count | high_alloc | high_avg_alloc |
+-----+-----+-----+-----+-----+-----+-----+
| memory/sql/String::value | 7281667 | 27.54 GiB | 4.01 KiB | 8156441 | 27.61 GiB | 3.55 KiB |
| memory/sql/thd::main_mem_root | 1798791 | 19.73 GiB | 11.50 KiB | 1900199 | 19.77 GiB | 3.40 KiB |
+-----+-----+-----+-----+-----+-----+-----+
```

上图是PFS中监控到的内存信息，虽然从内存监控中我们看到 `String::value` 和 `thd::main_mem_root` 上分配了很多的内存，但是这两个对象是MySQL中的基本对象，在非常多的地方使用。我们无法从这些信息推断出到底是什么操作导致了内存的占用。此外，由于PFS本身的内存占用比较高并且对性能有一定的影响，因此很多使用者不会在实例上开启PFS。

Jemalloc[1]是一个高效的内存分配器，通过进程和线程级的内存缓存机制，提升内存的分配、回收性能，并减少内存的碎片。此外Jemalloc提供了一套内存监控和分析机制，有以下几个特点：

- 通过采样的方式记录内存的分配和释放，来减少对进程运行的影响。
- 采样时记录了内存分配时的函数调用栈，通过函数栈可以准确的定位内存分配行为。
- 可以将采样信息dump到文件中。
- 通过jeprof工具可以将采样的信息用图形直观的展示。
- 可以将两个时间点的采样信息Diff后进行展示，这样可以聚焦到一段时间内分配的内存，让问题的分析更容易。

Jemalloc非常适合用来诊断MySQL中的内存使用问题，以及研究MySQL中的内存使用情况。下面我们通过两个案例来体验一下Jemalloc的强大内存分析能力。

案例 1 - Clone_persist_gtid内存泄漏

Bug#107991[2]是AliSQL的同学利用Jemalloc定位到的一个内存泄漏问题。`Clone_persist_gtid` 是MySQL-8.0引入的一个线程用来将事务的Gtid持久化到 `mysql.gtid_executed` 表中。如果实例的写事务非常频繁，就会遇到这个内存泄漏的问题。每次泄漏的内存不多，但是经过几天、十几天后泄漏的内存就会非常多。下图是该bug的函数调用栈的一部分，完整的快照文件在Bug#107991的页面里，需要的可以自取。

由于泄漏的内存占总内存的很小比例，这里的图是用两个时间点的采样信息 `Diff` 后产生的。通过这个函数调用栈，我们可以清楚地看到内存是Clone_persist_gtid线程打开系统表时分配的。通过这个函数调用栈很快就可以定位到，是因为该线程 `THD::mem_root` 一直没有清理导致的内存泄漏（详情请参考Bug#107991）。修复也很简单，只有一行代码。我们很快在AliSQL中做了修复，并且提交了Patch给官方。官方在最新的版本MySQL-8.0.42修复了该问题。

案例 2 - AHI内存浪费

除了分析内存泄漏，在研究MySQL的内存使用上也非常有用。比如拉起一个实例后，立即生成一个内存分配的快照，来分析实例启动时的内存分配情况。下图是一个64GB Buffer Pool实例拉起后的内存快照的一部分。

我们可以看到Buffer Pool初始化的过程中，分配的内存包括以下两部分：

- `btr_search_sys_create` 创建自适应哈希索引结构分配了1280MB内存。
- `buf_block_init` 为buffer pool中的每个page的rw lock和mutex创建的os_event结构，总共占用了1918.8MB内存。

由于AHI存在着严重的稳定性问题，我们默认都是关闭AHI的，官方也从MySQL-8.4开始将默认值改成了OFF。然而在AHI关闭的情况下，AHI仍然占用了1GB的内存。查阅代码可知这部分内存是buffer pool大小的 `1/64`（详细分析见Bug#112223[3]）。浪费的内存有点多，所以AliSQL修复了这个问题，在AHI关闭的情况下不会分配这部分的内存。

为什么图中没有Buffer Pool占用的64G内存呢？因为Buffer Pool的内存分配是用mmap不是malloc，所以不会被jemalloc监控到。

MySQL使用jemalloc

系统中安装libjemalloc后，做如下配置：

```
● ● ●
# 设置jemalloc库到LD_PRELOAD
export LD_PRELOAD=<路径/jemalloc.so.2>
# 启动 mysqld
```

如果使用mysqld_safe启动MySQL，则可以在MySQL的配置文件中配置，如下：



```
[mysqld_safe]  
malloc-lib = </路径/jemalloc.so.2>
```

可以通过如下方法检查mysqld是否使用了jemalloc

```
● ● ●  
lsof -p <pid> |grep jemalloc
```

开启jemalloc profiling

在启动mysqld前，需要提前设置以下环境变量开启jemalloc的profiling

```
● ● ●  
export MALLOC_CONF="prof:true,prof_active:true,prof_prefix:/tmp/mysqld.jedump"
```

- `prof`：设置开启profiling，只能在启动mysqld前设置。
- `prof_active`:设置profiling 的状态为active，如果设置为false，则不会记录内存分配的信息。
- `prof_prefix`: 设置profiling 快照的存储位置 and 文件前缀。

Profiling功能需要在编译时开启，如果jemalloc库不支持profiling，则会报如下的错误:

这时需要重新编译一个支持profiling的版本，编译时需要加上参数 `--enable-prof`。

对性能的影响

通过sysbench压测，大概可以得出以下的结论：

- Jemalloc-5.2 `prof:true,prof_active:false` 时, 对性能基本没有影响， `prof:true, prof_active:true` 时，高并发下大约有4%的性能下降。
- Jemalloc-5.3 在两种情况下对性能都没有明显的影响。

自动生成内存快照

jemalloc提供了两种自动生成快照的方式：

- `lg_prof_interval` 每分配多少内存后，dump一次。值为2的N次方，比如设置为30，则为2的30次方即1GB，意味着每1GBdump一次。
- `prof_gdump` 每次当总内存数创新高时dump 一次。

```
● ● ●
```

```
export MALLOC_CONF="prof:true,prof_active:true,lg_prof_interval:30,prof_prefix:/tmp/mysqld.jedump"
export MALLOC_CONF="prof:true,prof_active:true,prof_gdump:true,prof_prefix:/tmp/mysqld.jedump."
```

手动生成内存快照

自动生成快照虽然简单，但是对于MySQL这种会频繁分配释放内存的系统，我们更倾向于自己控制生成快照。手动产生快照需要在进程内调用相关的函数来生成快照，比如Percona[4]就在内核中集成了相关的命令来产生快照。

如果使用的是社区版的MySQL，该如何生成快照呢？这里提供了两种方式来手动生成快照，一种是通过gdb来生成，适合于开发环境，或者临时使用场景。另一种是通过UDF的方式来生成快照，这种方法更适用于在生产环境使用。

gdb 产生内存快照

这种方法是通过gdb调用mallctl函数来产生快照。这时就不需要设置log_prof_interval。

```
define jeprof_dump
  p mallctl("prof.dump", 0, 0, 0, 0)
end

jeprof_dump
```

将以上内容拷贝一个文件中(jemalloc.gdb)，然后调用gdb执行以上文件中的脚本。

```
gdb -p <pid> -x jemalloc.gdb -batch
```

此外，可以动态控制prof_active状态，查看active的状态，脚本如下：

```
define jeprof_status
  set $backup_opt_help = opt_help
  set $backup_opt_tc_log_size = opt_tc_log_size
  set opt_tc_log_size = sizeof(opt_help)

  call mallctl("opt.prof", &opt_help, &opt_tc_log_size, 0, 0)
  printf "opt.prof is %d\n", opt_help

  call mallctl("prof.active", &opt_help, &opt_tc_log_size, 0, 0)
  printf "prof.active is %d\n", opt_help

  set opt_help = $backup_opt_help
  set opt_tc_log_size = $backup_opt_tc_log_size
end

jeprof_status
```

```

define jeprof_off
    p mallctl("prof.active", 0, 0, &opt_help, sizeof(bool))
end
jeprof_off

```



```

define jeprof_on
    set $backup_opt_help = opt_help
    set opt_help = 1

    p mallctl("prof.active", 0, 0, &opt_help, sizeof(bool))

    set opt_help = $backup_opt_help
end
jeprof_on

```

UDF 产生内存快照

gdb的方式比较hack, 不适合自动化。此外gdb会中断进程的运行, 会导致实例的抖动。MySQL提供了一套可加载函数的机制, 这套机制通常也成为UDF[5]。通过这个机制, 我们可以用C语言, 将上述功能实现到一个动态库中, 然后动态的加载到正在运行的实例中。代码如下:



```

#include <jemalloc/jemalloc.h>
#include <string.h>
#include <stdbool.h>
/* The following is for user defined functions */

#define PLUGIN_EXPORT
#define longlong long
#define my_bool bool

enumItem_result {STRING_RESULT=0, REAL_RESULT, INT_RESULT, ROW_RESULT,
                  DECIMAL_RESULT};

typedefstructst_udf_args
{
    unsignedint arg_count;          /* Number of arguments */
    enumItem_result *arg_type;      /* Pointer to item_results */
    char **args;                   /* Pointer to argument */
    unsignedlong *lengths;          /* Length of string arguments */
    char *maybe_null;             /* Set to 1 for all maybe_null args */
    char **attributes;             /* Pointer to attribute name */
    unsignedlong *attribute_lengths; /* Length of attribute arguments */
    void *extension;
} UDF_ARGS;

/* This holds information about the result */

typedefstructst_udf_init
{
    my_bool maybe_null;            /* 1 if function can return NULL */
    unsignedint decimals;          /* for real functions */
    unsignedlong max_length;       /* For string functions */
}

```

```

char *ptr;                /* free pointer for function data */
my_bool const_item;       /* 1 if function always returns the same value */
void *extension;
} UDF_INIT;

// 查看prof是否开启 (opt.prof), 成功return 0
PLUGIN_EXPORT my_bool
jeprof_prof_status_init(UDF_INIT *initid, UDF_ARGS *args, char *message)
{
    if (args->arg_count != 0) {
        strcpy(message, "Usage: prof_opt_status()");
        return 1;
    }
    return 0;
}

PLUGIN_EXPORT longlong
jeprof_prof_status(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
{
    bool enabled;
    size_t len = sizeof(enabled);

    if (mallctl("opt.prof", &enabled, &len, NULL, 0)) {
        *error = 1;
        return -1;
    }
    return enabled ? 1 : 0;
}

// 查看prof.active状态 (prof.active), 成功return 0
PLUGIN_EXPORT my_bool
jeprof_active_status_init(UDF_INIT *initid, UDF_ARGS *args, char *message)
{
    if (args->arg_count != 0) {
        strcpy(message, "Usage: prof_active_status()");
        return 1;
    }
    return 0;
}

PLUGIN_EXPORT longlong
jeprof_active_status(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
{
    bool active;
    size_t len = sizeof(active);

    if (mallctl("prof.active", &active, &len, NULL, 0)) {
        *error = 1;
        return -1;
    }
    return active ? 1 : 0;
}

// 开启 profiling, 设置prof.active为true, 成功return 0
PLUGIN_EXPORT my_bool
jeprof_enable_init(UDF_INIT *initid, UDF_ARGS *args, char *message)
{
    if (args->arg_count != 0) {
        strcpy(message, "Usage: prof_enable()");
        return 1;
    }
}

```

```
        return 0;
    }

    PLUGIN_EXPORT longlong
    jeprof_enable(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
    {
        bool enable = true;

        if (mallctl("prof.active", NULL, NULL, &enable, sizeof(enable))) {
            *error = 1;
            return -1;
        }
        return 0;
    }

    // 关闭 profiling, 设置prof.active为false, 成功 return 0
    PLUGIN_EXPORT my_bool
    jeprof_disable_init(UDF_INIT *initid, UDF_ARGS *args, char *message)
    {
        if (args->arg_count != 0) {
            strcpy(message, "Usage: prof_disable()");
            return 1;
        }
        return 0;
    }

    PLUGIN_EXPORT longlong
    jeprof_disable(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
    {
        bool enable = false;
        if (mallctl("prof.active", NULL, NULL, &enable, sizeof(enable))) {
            *error = 1;
            return -1;
        }
        return 0;
    }

    // Dump 内存堆栈信息, 成功return 0
    PLUGIN_EXPORT my_bool
    jeprof_dump_init(UDF_INIT *initid, UDF_ARGS *args, char *message)
    {
        if (args->arg_count != 0) {
            strcpy(message, "Usage: prof_dump()");
            return 1;
        }
        return 0;
    }

    PLUGIN_EXPORT longlong
    jeprof_dump(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
    {
        int ret = mallctl("prof.dump", NULL, NULL, NULL, 0);
        if (ret) {
            *error = 1;
            return ret;
        }
        return 0;
    }
}
```

编译以上代码依赖jemalloc的头文件，因此需要先安装jemalloc的头文件。yum下安装方法如下：

```
yum install jemalloc-devel
```

然后通过以下命令进行编译，编译后将jemalloc_udf.so拷贝到实例的plugin_dir[6]

```
gcc -shared -fPIC -o jemalloc_udf.so jeprof_udf.c
```

在使用这些udf前，需要先加载这些UDF，SQL如下：

```
CREATE FUNCTION jeprof_dump RETURNS INTEGER SONAME 'jemalloc_udf.so';
CREATE FUNCTION jeprof_enable RETURNS INTEGER SONAME 'jemalloc_udf.so';
CREATE FUNCTION jeprof_disable RETURNS INTEGER SONAME 'jemalloc_udf.so';
CREATE FUNCTION jeprof_prof_status RETURNS INTEGER SONAME 'jemalloc_udf.so';
CREATE FUNCTION jeprof_active_status RETURNS INTEGER SONAME 'jemalloc_udf.so';
```

然后就可以通过SELECT jeprof_xxx()来调用。如图所示：

- `jeprof_prof_status` 显示 `prof` 参数的状态，返回1代表开启，0代表关闭。
- `jeprof_active_status` 显示 `prof.active` 的状态，返回1代表开启，0代表关闭。
- `jeprof_enable` 设置 `prof.active` 为 `true`，成功返回0，否则返回1。
- `jeprof_disable` 设置 `prof.active` 为 `false`，成功返回0，否则返回1。
- `jeprof_dump` 生成一个内存快照文件，成功返回0，否则返回1。

生成Profiling 调用关系图


Jemalloc中有一个 `jeprof` 命令行工具，用来直观的显示内存快照中的信息，一个常用的就是生成调用关系图。快照信息可以用svg,pdf等方式呈现，如图所示

生图命令如下：



```
jeprof ./sql/mysqld mysqld.jedump.2 -svg > jedump.svg
```

产生两个快照的diff命令如下:



```
jeprof ./sql/mysqld --base mysqld.jedump.1 mysqld.jedump.2 -svg > jedump_diff.svg
```

总结

Jemalloc提供了强大内存分析功能，在记录分配的内存信息时采集了函数栈信息，这些信息可以帮助我们快速准确地定位内存问题。通过MySQL的UDF机制，可以很方便的将jemalloc的内存快照能力集成到MySQL中，并且在线的开启和使用。Jemalloc通过采样的来统计内存信息，在默认配置下开销很小，可以放心开启。

引用链接

- [1] Jemalloc:<https://github.com/jemalloc/jemalloc>
- [2] Bug#107991:<https://bugs.mysql.com/bug.php?id=107991>
- [3] Bug#112223:<https://bugs.mysql.com/bug.php?id=112223>
- [4] Percona:<https://docs.percona.com/percona-server/8.0/jemalloc-profiling.html#use-percona-server-for-mysql-with-jemalloc-with-profiling-enabled>
- [5] UDF:<https://dev.mysql.com/doc/extending-mysql/8.4/en/adding-loadable-function.html>
- [6] plugin_dir:https://dev.mysql.com/doc/refman/8.4/en/server-system-variables.html#sysvar_plugin_dir

