# 背景

无论从使用、研发还是运维的角度，内存监控一直是MySQL的重点之一。完善的内存监控手段有很多作用，包括但不限于：

1、发现内存泄漏，避免MySQL实例内存耗尽

2、对实例的运行状态进行定量分析

3、资源管控和优化

但内存监控想要“完善”并不是那么简单的事。

## PFS内存监控

在PFS（Performance\_schema）中，一共有五张内存相关的监控表，每张表会从不同维度收集和聚合内存事件。

memory\_summary\_by\_account\_by\_event\_name: 从用户和连接host的角度统计内存信息。

memory\_summary\_by\_host\_by\_event\_name: 从host角度统计内存信息。

memory\_summary\_by\_thread\_by\_event\_name: 从线程角度统计内存信息。

memory\_summary\_by\_user\_by\_event\_name: 从用户角度统计内存信息。

memory\_summary\_global\_by\_event\_name: 从Memory Event（内存事件）本身，统计全局的内存信息。

每张表内，内存相关的列如下：

COUNT\_ALLOC, COUNT\_FREE: 调用内存分配器进行内存分配和释放的次数。

SUM\_NUMBER\_OF\_BYTES\_ALLOC, SUM\_NUMBER\_OF\_BYTES\_FREE: 总共分配和释放内存的字节数。

CURRENT\_COUNT\_USED: COUNT\_ALLOC − COUNT\_FREE.

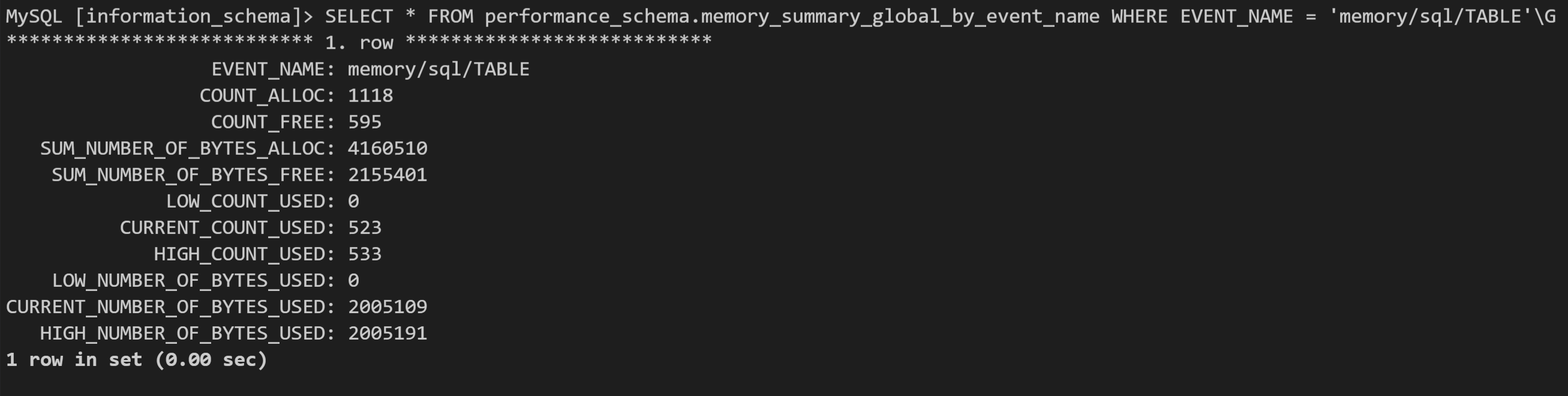
CURRENT\_NUMBER\_OF\_BYTES\_USED: 目前正在使用的内存字节数。它等于 SUM\_NUMBER\_OF\_BYTES\_ALLOC − SUM\_NUMBER\_OF\_BYTES\_FREE.

LOW\_COUNT\_USED, HIGH\_COUNT\_USED: 内存block的使用范围（最小-最大）。

LOW\_NUMBER\_OF\_BYTES\_USED, HIGH\_NUMBER\_OF\_BYTES\_USED: 内存字节数的使用范围（最小-最大）。

如下：

SELECT \* FROM performance\_schema.memory\_summary\_global\_by\_event\_name WHERE EVENT\_NAME = 'memory/sql/TABLE'\G



## 8.0.28以前的InnoDB内存监控

最简单的内存监控，就是把malloc()和free()包装一下，在里面做其他的事情：

void \*traced\_malloc(size\_t size, const char \*user) {

void \*ptr = malloc(size); // record the allocation in some ways

// trace(size,user)

return ptr;

}

void traced\_free(void \*ptr) {

// obtain the allocation information in some ways

// information = get\_trace(ptr)

free(ptr);

}

上面代码的意思是，在执行真正的内存分配/释放操作之前，通过某些手段记录这次“内存事件”，随后再执行真正的分配/释放，从而能够统计内存的使用情况。

因为我们在讨论C++，所以也可以把new/delete包一层，做同样的事情。

具体到InnoDB的代码上，InnoDB通过allocate\_trace和deallocate\_trace来做这件事：

/\*\* Trace a memory allocation.

@param[in] size number of bytes that were allocated

@param[in] key Performance Schema key

@param[out] pfx placeholder to store the info which will be needed when freeing the memory \*/

void allocate\_trace(size\_t size, PSI\_memory\_key key, ut\_new\_pfx\_t \*pfx)

{

if (m\_key != PSI\_NOT\_INSTRUMENTED) {

key = m\_key;

}

pfx->m\_key = PSI\_MEMORY\_CALL(memory\_alloc)(key, size, &pfx->m\_owner);

pfx->m\_size = size;

}

/\*\* Trace a memory deallocation.

@param[in] pfx info for the deallocation \*/

void deallocate\_trace(const ut\_new\_pfx\_t \*pfx)

{

PSI\_MEMORY\_CALL(memory\_free)(pfx->m\_key, pfx->m\_size, pfx->m\_owner);

}

但是，这个内存监控已经很老了，有一些显而易见的缺点：

1、对于STL容器内的Allocator没有实现，如std::vector<>内的元素无法统计到

2、对于新的语法（如C++17引入的std::align\_val\_t等）无法支持统计

3、对于智能指针的支持不到位（如make\_unique(), make\_shared()）

4、强耦合PFS，扩展性不高

在8.0.28，MySQL官方把内存监控彻底重构，解决了上述问题。

# 概述

参考：<https://cloud.tencent.com/developer/article/2135517?areaSource=&traceId=>

内存管理MEM\_ROOT：<https://codeantenna.com/a/8SjxogbP6Y>

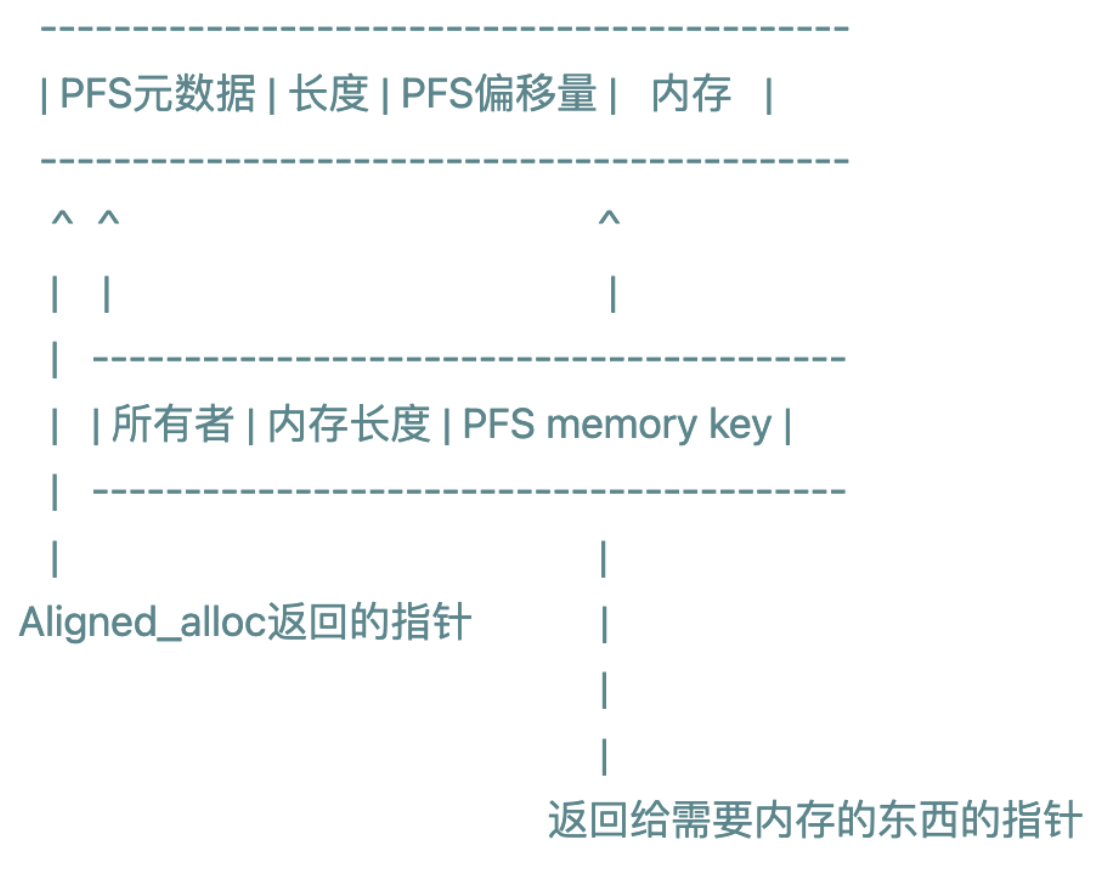
MySQL内存分配与管理：<https://ost.51cto.com/posts/21359>

<http://mysql.taobao.org/monthly/2022/11/02/>

MySQL源码解析动态数组：<https://www.cnblogs.com/jkin/p/14961910.html>

InnoDB引入了一个新的内存区段，叫做PFS元数据。所有通过performance\_schema追踪内存使用的allocator都会使用该统一的元数据结构。

结构大概长这样：



该PFS元数据由内部分配器分配额外的长度储存，并将用户申请的真实内存指针贴在后面。也就是这个实现细节是对上层应用隐藏的，在分配/释放的时候，通过指针计算，获取该元数据的偏移量来统计内存事件。

一个内存元数据由三部分组成：

申请的线程（所有者）

申请的内存长度

PFS Memory Key，用于分类别统计内存

来看一个具体实现，以operator new的allocate()函数为例：

static inline void \*alloc(std::size\_t size, pfs\_metadata::pfs\_memory\_key\_t key)

{

const auto total\_len = size + Alloc\_pfs::metadata\_len;

auto mem = Alloc\_fn::alloc<Zero\_initialized>(total\_len);

if (unlikely(!mem)) return nullptr;

// The point of this allocator variant is to trace the memory allocations

// through PFS (PSI) so do it.

pfs\_metadata::pfs\_owning\_thread\_t owner;

key = PSI\_MEMORY\_CALL(memory\_alloc)(key, total\_len, &owner);

// To be able to do the opposite action of tracing when we are releasing the // memory, we need right about the same data we passed to the tracing

// memory\_alloc function. Let's encode this it into our allocator so we

// don't have to carry and keep this data around. pfs\_metadata::pfs\_owning\_thread(mem, owner); //所有者 pfs\_metadata::pfs\_datalen(mem, total\_len); //内存长度 pfs\_metadata::pfs\_key(mem, key); //PFS Memory Key pfs\_metadata::pfs\_metaoffset(mem, Alloc\_pfs::metadata\_len); //PFS偏移量 return static\_cast<uint8\_t \*>(mem) + Alloc\_pfs::metadata\_len;

}

在申请内存之前，MySQL首先通过metadata\_len计算出额外所需的内存大小，然后根据总和申请内存。

申请内存后，根据元数据结构的定义，依次将内存所有者，内存长度，PFS Key，偏移量写入额外的内存空间。

最后，通过指针计算出返回值的内存偏移，将真实的内存返回给上层（隐藏了额外的内容）。

同样，在释放内存时，根据上层传入的指针，逆向计算出整块内存的起始地址，并取出元数据后，再释放所有内存。

实现内存分配器后，InnoDB在头文件中使用using语法对常用的容器进行了重定向，这样即使开发者忘记指定内存分配器，也不会影响内存统计。

template <typename T>using vector = std::vector<T, ut::allocator<T>>;

/\*\* Specialization of list which uses ut\_allocator. \*/

template <typename T>using list = std::list<T, ut::allocator<T>>;

/\*\* Specialization of set which uses ut\_allocator. \*/

template <typename Key, typename Compare = std::less<Key>>using set = std::set<Key, Compare, ut::allocator<Key>>;

template <typename Key>using unordered\_set = std::unordered\_set<Key, std::hash<Key>, std::equal\_to<Key>, ut::allocator<Key>>;

/\*\* Specialization of map which uses ut\_allocator. \*/

template <typename Key, typename Value, typename Compare = std::less<Key>>using map = std::map<Key, Value, Compare, ut::allocator<std::pair<const Key, Value>>>;

同时，还有对智能指针的实现：

template <typename T, typename Deleter = detail::Array\_deleter<std::remove\_extent\_t<T>>>std::enable\_if\_t<detail::is\_bounded\_array\_v<T>, std::shared\_ptr<T>> make\_shared(PSI\_memory\_key\_t key)

{

return std::shared\_ptr<T>( ut::new\_arr\_withkey<std::remove\_extent\_t<T>>( key, ut::Count{detail::bounded\_array\_size\_v<T>}), Deleter{});

}

那扩展性如何解决呢？上述函数所在的类叫做

Alloc\_pfs : public allocator\_traits<true>

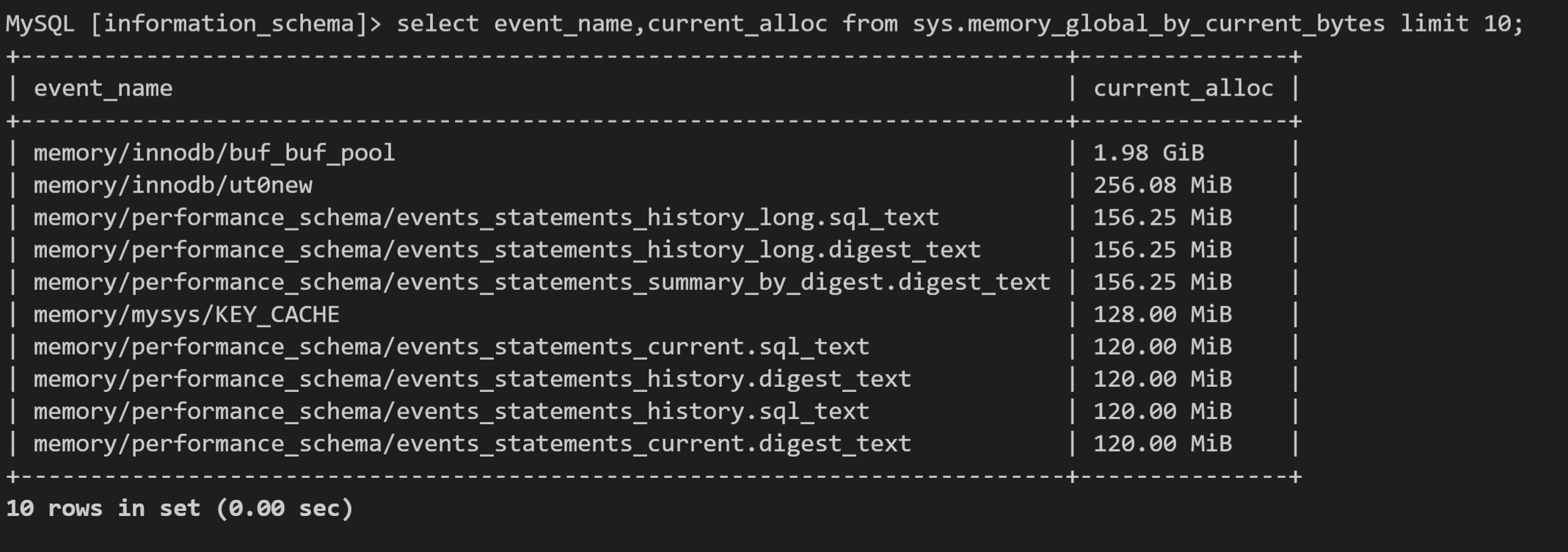
继承了一个统一的基类allocator\_traits。如果以后有需要，还可以扩展出使用其他统计方式的内存分配器，不需要更改上层逻辑，只需要更改内存分配策略即可。

# 原理

首先，简单举例一下PFS内存监控的使用方法。

打开performance\_schema后，可以通过如下SQL语句获取全局的内存使用情况：

select event\_name,current\_alloc from ys.memory\_global\_by\_current\_bytes;



这句话的意思是，获取整个实例的前10内存消耗量的元素。可以看到，排第一的是InnoDB Buffer Pool。

接下来，我们来了解一个线上用户的实际案例。

某线上用户实例频繁OOM。通过PFS观察该用户的内存使用情况如下：

select \* from memory\_by\_thread\_by\_current\_bytes ;+-----------+--------------------------------------+--------------------+-------------------+-------------------+-------------------+-----------------+| thread\_id | user | current\_count\_used | current\_allocated | current\_avg\_alloc | current\_max\_alloc | total\_allocated |+-----------+--------------------------------------+--------------------+-------------------+-------------------+-------------------+-----------------+| 55 | root@localhost | 364315 | 1.76 GiB | 5.06 KiB | 1.75 GiB | 8.33 GiB |

mysql> select event\_name,current\_alloc from sys.memory\_global\_by\_current\_bytes limit 10;+-----------------------------------------------------------------------------+---------------+| event\_name | current\_alloc |+-----------------------------------------------------------------------------+---------------+| memory/sql/user\_var\_entry::value | 1.92 GiB || memory/innodb/buf\_buf\_pool | 1.05 GiB || memory/performance\_schema/events\_statements\_summary\_by\_digest | 40.28 MiB || memory/innodb/ut0link\_buf | 24.00 MiB || memory/innodb/log\_buffer\_memory | 16.00 MiB || memory/performance\_schema/events\_statements\_history\_long | 14.19 MiB || memory/performance\_schema/events\_errors\_summary\_by\_thread\_by\_error | 12.70 MiB || memory/performance\_schema/events\_statements\_summary\_by\_thread\_by\_event\_name | 11.04 MiB || memory/performance\_schema/events\_statements\_summary\_by\_digest.digest\_text | 9.77 MiB || memory/performance\_schema/events\_statements\_history\_long.sql\_text | 9.77 MiB |+-----------------------------------------------------------------------------+---------------+

可以看到，thread\_id为55的用户占用内存较多（这里只截取了部分），且全局内存使用中有一项memory/sql/user\_var\_entry::value 异常增大。

通过PSI Memory Key定位到代码，发现该用户的一个存储过程存在死循环，并且在循环中频繁更改一个变量的值。由于用户开启了Binlog，所有的变量修改都会记录一份“历史记录”，在生成Binlog Event事件时一并写入。但因为存储过程死循环，此时并没有DML执行，因此“历史记录”在内存中堆积，堆积过多就引发了OOM现象。

排查清楚后，联系用户修改了存储过程代码，后来没有再复现。

# 源码