PHP 内存泄漏分析定位

目录

前言

本文开始撰写时我负责的项目需要用php开发一个通过 Socket 与服务端建立长连接后持续实时上报数据的常驻进程程序，在程序业务功能开发联调完毕后实际运行发送大量数据后发现内存增长非常迅速，在很短的时间内达到了 php 默认可用内存上限 128M ，并报错：

```
Fatal error: Allowed memory size of X bytes exhausted (tried to allocate Y bytes)
```

我第一反应是内存泄露了，但是不知道在哪。第二反应是无用的变量应该用完就 unset 掉，修改完毕后问题依旧。经过了几番周折终于解决了问题。就决定好好把类似情况整理一下，遂有此文，与诸君共勉。

观察 PHP 程序内存使用情况

php提提供了两个方法来获取当前程序的内存使用情况。

- memory_get_usage()，这个函数的作用是获取目前PHP脚本所用的内存大小。
- memory_get_peak_usage()，这个函数的作用返回当前脚本到目前位置所占用的内存峰值，这样就可能获取到目前的脚本的内存需求情况。

```
int memory_get_usage ([ bool $real_usage = false ] )
int memory_get_peak_usage ([ bool $real_usage = false ] )
```

函数默认得到的是调用 `emalloc()` 占用的内存，如果设置参数为 `TRUE` ，则得到的是实际程序向系统申请的内存。因为 PHP 有自己的内存管理机制，所以有时候尽管内部已经释放了内存但并没有还给系统。

linux 系统文件 /proc/{$pid}/status 会记录某个进程的运行状态，里面的 VmRSS 字段记录了该进程使用的常驻物理内存(Residence)，这个就是该进程实际占用的物理内存了，用这个数据比较靠谱，在程序里面提取这个值也很容易

场景一：程序操作数据过大

情景还原：一次性读取超过php可用内存上限的数据导致内存耗尽

```
<?php
ini_set('memory_limit', '128M');
$string = str_pad('1', 128 * 1024 * 1024);
```

```
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 134217729 bytes) in
/Users/zouyi/php-oom/bigfile.php on line 3
```

这是告诉我们程序运行时试图分配新内存时由于达到了PHP允许分配的内存上限而抛出致命错误，无法继续执行了，在 java 开发中一般称之为 OOM (Out Of Memory) 。

PHP 配置内存上限是在 `php.ini` 中设置 `memory_limit` ，PHP 5.2 以前这个默认值是 `8M` ，PHP 5.2 的默认值是 `16M` ，在这之后的版本默认值都是 `128M` 。

问题现象：特定数据处理时可复现，做任何 IO 操作都有可能遇到此类问题，比如：一次 mysql 查询返回大量数据、一次把大文件读取进程序等。

解决方法：

1. 能用钱解决的问题都不是问题，如果程序要读大文件的机会不是很多，且上限可预期，那么通过 `ini_set('memory_limit', '1G');` 来设置一个更大的值或者 `memory_limit=-1` 。内存管够的话让程序一直跑也可以。
2. 如果程序需要考虑在小内存机器上也能正常使用，那就需要优化程序了。如下，代码复杂了很多。

```
<?php
//php7 以下版本通过 composer 引入 paragonie/random_compat ，为了方便来生成一个随机名称的临时文件
require "vendor/autoload.php";

ini_set('memory_limit', '128M');
//生成临时文件存放大字符串
$fileName = 'tmp'.bin2hex(random_bytes(5)).'.txt';
touch($fileName);
for ( $i = 0; $i < 128; $i++ ) {
    $string = str_pad('1', 1 * 1024 * 1024);
    file_put_contents($fileName, $string, FILE_APPEND);
}
$handle = fopen($fileName, "r");
for ( $i = 0; $i <= filesize($fileName) / 1 * 1024 * 1024; $i++ )
{
   //do something
   $string = fread($handle, 1 * 1024 * 1024);
}

fclose($handle);
unlink($fileName);
```


场景二：程序操作大数据时产生拷贝

情景还原：执行过程中对大变量进行了复制，导致内存不够用。

```
<?php
ini_set("memory_limit",'1M');

$string = str_pad('1', 1* 750 *1024);
$string2 = $string;
$string2 .= '1';
```

```
Fatal error: Allowed memory size of 1048576 bytes exhausted (tried to allocate 768001 bytes) in
/Users/zouyi/php-oom/unset.php on line 8

Call Stack:
    0.0004     235440   1. {main}() /Users/zouyi/php-oom/unset.php:0

zend_mm_heap corrupted
```

问题现象：局部代码执行过程中占用内存翻倍。

问题分析：

php 是写时复制（Copy On Write），也就是说，当新变量被赋值时内存不发生变化，直到新变量的内容被操作时才会产生复制。

解决方法：

及早释放无用变量，或者以引用的形式操作原始数据。

```
<?php
ini_set("memory_limit",'1M');
```

```php
$string = str_pad('1', 1* 750 *1024);
$string2 = $string;
unset($string);
$string2 .= '1';
```

```php
<?php
ini_set("memory_limit",'1M');

$string = str_pad('1', 1* 750 *1024);
$string2 = &$string;
$string2 .= '1';

unset($string2, $string);
```

场景三：配置不合理系统资源耗尽

情景还原：因配置不合理导致内存不够用，2G 内存机器上设置最大可以启动 100 个 php-fpm 子进程，但实际启动了 50 个 php-fpm 子进程后无法再启动更多进程

问题现象：线上业务请求量小的时候不出现问题，请求量一旦很大后部分请求就会执行失败

问题分析：

一般为了安全方面考虑，php 限制表单请求的最大可提交的数量及大小等参数，`post_max_size`、`max_file_uploads`、`upload_max_filesize`、`max_input_vars`、`max_input_nesting_level`。
假设带宽足够，用户频繁的提交 `post_max_size = 8M` 数据到服务端，nginx 转发给 php-fpm 处理，那么每个 php-fpm 子进程除了自身占用的内存外，即使什么都不做也有可能多占用 8M 内存。

解决方法：

合理设置 `post_max_size`、`max_file_uploads`、`upload_max_filesize`、`max_input_vars`、`max_input_nesting_level` 等参数并调优 php-fpm 相关参数。

php.ini

```
$ php -i |grep memory
memory_limit => 1024M => 1024M //php脚本执行最大可使用内存
$php -i |grep max
max_execution_time => 0 => 0 //最大执行时间，脚本默认为0不限制，web请求默认30s
max_file_uploads => 20 => 20 //一个表单里最大上传文件数量
max_input_nesting_level => 64 => 64 //一个表单里数据最大数组深度层数
max_input_time => -1 => -1 //php从接收请求开始处理数据后的超时时间
max_input_vars => 1000 => 1000 //一个表单（包括get、post、cookie的所有数据）最多提交1000个字段
post_max_size => 8M => 8M //一次post请求最多提交8M数据
upload_max_filesize => 2M => 2M //一个可上传的文件最大不超过2M
```

如果上传设置不合理那么出现大量内存被占用的情况也不奇怪，比如有些内网场景下需要 post 超大字符串 `post_max_size=200M`，那么当从表单提交了 200M 数据到服务端，php 就会分配 200M 内存给这条数据，直到请求处理完毕释放内存。

php-fpm.conf

```
pm = dynamic //仅dynamic模式下以下参数生效
pm.max_children = 10 //最大子进程数
pm.start_servers = 3 //启动时启动子进程数
pm.min_spare_servers = 2 //最小空闲进程数，不够了启动更多进程
pm.max_spare_servers = 5 //最大空闲进程数，超过了结束一些进程
pm.max_requests = 500 //最大请求数，注意这个参数是一个php-fpm如果处理了500个请求后会自己重启一下，可以避免一些三方扩展的内存泄露问题
```

一个 php-fpm 进程按 30MB 内存算，50 个 php-fpm 进程就需要 1500MB 内存，这里需要简单估算一下在负载最重的情况下所有 php-fpm 进程都启动后是否会把系统内存耗尽。

ulimit

```
$ulimit -a
-t: cpu time (seconds)              unlimited
-f: file size (blocks)              unlimited
-d: data seg size (kbytes)          unlimited
-s: stack size (kbytes)             8192
-c: core file size (blocks)         0
-v: address space (kbytes)          unlimited
-l: locked-in-memory size (kbytes)  unlimited
-u: processes                       1024
-n: file descriptors                1024
```

这是我本地mac os的配置，文件描述符的设置是比较小的，一般生产环境配置要大得多。

场景四：无用的数据未及时释放

情景还原：这种问题从程序逻辑上不是问题，但是无用的数据大量占用内存导致资源不够用，应该有针对性的做代码优化。

Laravel开发中用于监听数据库操作时有如下代码:

```
DB::listen(function ($query) {
    // $query->sql
    // $query->bindings
    // $query->time
});
```

启用数据库监听后，每当有 SQL 执行时会 new 一个 QueryExecuted 对象并传入匿名函数以便后续操作，对于执行完毕就结束进程释放资源的php程序来说没有什么问题，而如果是一个常驻进程的程序，程序每执行一条 SQL 内存中就会增加一个 QueryExecuted 对象，程序不结束内存就会始终增长。

问题现象：程序运行期间内存逐渐增长，程序结束后内存正常释放。

问题分析：

此类问题不易察觉，定位困难，尤其是有些框架封装好的方法，要明确其适用场景。

解决方法：

本例中要通过 `DB::listen` 方法获取所有执行的 SQL 语句记录并写入日志，但此方法存在内存泄露问题，在开发环境下无所谓，在生产环境下则应停用，改用其他途径获取执行的 SQL 语句并写日志。

深入了解

1. 名词解释

- 内存泄漏（Memory Leak）：是程序在管理内存分配过程中未能正确的释放不再使用的内存导致资源被大量占用的一种问题。在面向对象编程时，造成内存泄露的原因常常是对象在内存中存储但是运行中的代码却无法访问他。由于产生类似问题的情况很多，所以只能从源码上入手分析定位并解决。
- 垃圾回收（Garbage Collection，简称GC）：是一种自动内存管理的形式，GC程序检查并处理程序中那些已经分配出去但却不再被对象使用的内存。最早的GC是1959年前后John McCarthy发明的，用来简化在Lisp中手动控制内存管理。
  PHP的内核中已自带内存管理的功能，一般应用场景下，不易出现内存泄露。
- 追踪法（Tracing）：从某个根对象开始追踪，检查哪些对象可访问，那么其他的（不可访问）就是垃圾。
- 引用计数法（reference count）：每个对象都一个数字用来标示被引用的次数。引用次数为0的可以回收。当对一个对象的引用创建时他的引用计数就会增加，引用销毁时计数减少。引用计数法可以保证对象一旦不被引用时第一时间销毁。但是引用计数有一些缺陷：1.循环引用，2.引用计数需要申请更多内存，3.对速度有影响，4.需要保证原子性，5.不是实时的

2. php 内存管理

> 在 PHP 5.2 以前, PHP 使用引用计数(Reference count)来做资源管理, 当一个 zval 的引用计数为 0 的时候, 它就会被释放. 虽然存在循环引用(Cycle reference), 但这样的设计对于开发 Web 脚本来说, 没什么问题, 因为 Web 脚本的特点和它追求的目标就是执行时间短,

> 不会长期运行. 对于循环引用造成的资源泄露, 会在请求结束时释放掉. 也就是说, 请求结束时释放资源, 是一种部补救措施(backup).
>
> 然而, 随着 PHP 被越来越多的人使用, 就有很多人在一些后台脚本使用 PHP , 这些脚本的特点是长期运行, 如果存在循环引用, 导致引用计数无法及时释放不用的资源, 则这个脚本最终会内存耗尽退出.
>
> 所以在 PHP 5.3 以后, 我们引入了 GC .

—— 摘自鸟哥博客文章《请手动释放你的资源》

在 PHP 5.3 以后引入了同步周期回收算法(Concurrent Cycle Collection)来处理内存泄露问题，代价是对性能有一定影响，不过一般 web 脚本应用程序影响很小。 PHP的垃圾回收机制是默认打开的， php.ini 可以设置 `zend.enable_gc=0` 来关闭。 也能通过分别调用 gc_enable() 和 gc_disable()函数来打开和关闭垃圾回收机制。

虽然垃圾回收让php开发者在内存管理上无需担心了，但也有极端的反例：php界著名的包管理工具composer曾因加入一行 `gc_disable()`; 性能得到极大提升。传送门

引用计数基本知识

回收周期(Collecting Cycles)

上面两个链接是php官方手册中的内存管理、GC相关知识讲解，图文并茂，这里不再赘述。

3. php-fpm 内存泄露问题

在一台常见的 nginx + php-fpm 的服务器上：

1. nginx 服务器 fork 出 n 个子进程（ worker ）， php-fpm 管理器 fork 出 n 个子进程。
2. 当有用户请求， nginx 的一个 worker 接收请求，并将请求抛到 socket 中。
3. php-fpm 空闲的子进程监听到 socket 中有请求，接收并处理请求。

一个 php-fpm 的生命周期大致是这样的：

模块初始化（ MINIT ）-> 请求初始化（ RINIT ）-> 请求处理 -> 请求结束（ RSHUTDOWN ）-> 请求初始化（ RINIT ）-> 请求处理 -> 请求结束（ RSHUTDOWN ）……. 请求初始化（ RINIT ）-> 请求处理 -> 请求结束（ RSHUTDOWN ）-> 模块关闭（ MSHUTDOWN ）。

在 请求初始化（RINIT）-> 请求处理 -> 请求结束（RSHUTDOWN） 这个“请求处理”过程是： php 读取相应的 php 文件，对其进行词法分析，生成 opcode ， zend 虚拟机执行 opcode 。

php 在每次请求结束后自动释放内存，有效避免了常见场景下内存泄露的问题，然而实际环境中因某些扩展的内存管理没有做好或者 php 代码中出现循环引用导致未能正常释放不用的资源。

在 php-fpm 配置文件中，将 `pm.max_requests` 这个参数设置小一点。这个参数的含义是：一个 php-fpm 子进程最多处理 `pm.max_requests` 个用户请求后，就会被销毁。当一个 php-fpm 进程被销毁后，它所占用的所有内存都会被回收。

4. 常驻进程内存泄露问题

Valgrind 包括如下一些工具：

- Memcheck。这是 valgrind 应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。
- Callgrind。它主要用来检查程序中函数调用过程中出现的问题。
- Cachegrind。它主要用来检查程序中缓存使用出现的问题。
- Helgrind。它主要用来检查多线程程序中出现的竞争问题。
- Massif。它主要用来检查程序中堆栈使用中出现的问题。
- Extension。可以利用core提供的功能，自己编写特定的内存调试工具。

Memcheck 对调试 C/C++ 程序的内存泄露很有帮助，它的机制是在系统 alloc/free 等函数调用上加计数。 php 程序的内存泄露，是由于一些循环引用，或者 gc 的逻辑错误, valgrind 无法探测，因此需要在检测时需要关闭 php 自带的内存管理。

```
$ export USE_ZEND_ALLOC=0    # 设置环境变量关闭内存管理
$ valgrind --tool=memcheck --num-callers=30 --log-file=php.log /Users/zouyi/Downloads/php-5.6.31/sapi/cli/php  leak.php
```

通过命令行执行 valgrind 分析可能有内存泄露的文件

```
==12075== Memcheck, a memory error detector
==12075== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12075== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==12075== Command: /Users/zouyi/Downloads/php-5.6.31/sapi/cli/php leak.php
==12075== Parent PID: 42043
==12075==
==12075== Syscall param msg->desc.port.name points to uninitialised byte(s)
==12075==    at 0x10121F34A: mach_msg_trap (in /usr/lib/system/libsystem_kernel.dylib)
==12075==    by 0x10121E796: mach_msg (in /usr/lib/system/libsystem_kernel.dylib)
==12075==    by 0x101218485: task_set_special_port (in /usr/lib/system/libsystem_kernel.dylib)
==12075==    by 0x1013B410E: _os_trace_create_debug_control_port (in
/usr/lib/system/libsystem_trace.dylib)
==12075==    by 0x1013B4458: _libtrace_init (in /usr/lib/system/libsystem_trace.dylib)
==12075==    by 0x100DF09DF: libSystem_initializer (in /usr/lib/libSystem.B.dylib)
==12075==    by 0x100C37A1A: ImageLoaderMachO::doModInitFunctions(ImageLoader::LinkContext const&) (in
/usr/lib/dyld)
==12075==    by 0x100C37C1D: ImageLoaderMachO::doInitialization(ImageLoader::LinkContext const&) (in
/usr/lib/dyld)
==12075==    by 0x100C334A9: ImageLoader::recursiveInitialization(ImageLoader::LinkContext const&,
unsigned int, char const*, ImageLoader::InitializerTimingList&, ImageLoader::UninitedUpwards&) (in
/usr/lib/dyld)
==12075==    by 0x100C33440: ImageLoader::recursiveInitialization(ImageLoader::LinkContext const&,
unsigned int, char const*, ImageLoader::InitializerTimingList&, ImageLoader::UninitedUpwards&) (in
/usr/lib/dyld)
==12075==    by 0x100C32523: ImageLoader::processInitializers(ImageLoader::LinkContext const&,
unsigned int, ImageLoader::InitializerTimingList&, ImageLoader::UninitedUpwards&) (in /usr/lib/dyld)
==12075==    by 0x100C325B8: ImageLoader::runInitializers(ImageLoader::LinkContext const&,
ImageLoader::InitializerTimingList&) (in /usr/lib/dyld)
==12075==    by 0x100C24433: dyld::initializeMainExecutable() (in /usr/lib/dyld)
==12075==    by 0x100C288C5: dyld::_main(macho_header const*, unsigned long, int, char const**, char
const**, char const**, unsigned long*) (in /usr/lib/dyld)
==12075==    by 0x100C23248: dyldbootstrap::start(macho_header const*, int, char const**, long,
macho_header const*, unsigned long*) (in /usr/lib/dyld)
==12075==    by 0x100C23035: _dyld_start (in /usr/lib/dyld)
==12075==    by 0x1: ???
==12075==    by 0x1054AC862: ???
==12075==    by 0x1054AC891: ???
==12075==  Address 0x1054aa98c is on thread 1's stack
==12075==  in frame #2, created by task_set_special_port (???:)
==12075==
--12075-- UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option
--12075-- UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option (repeated 2 times)
--12075-- UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option (repeated 4 times)
==12075==
==12075== HEAP SUMMARY:
==12075==     in use at exit: 125,805 bytes in 185 blocks
==12075==   total heap usage: 14,686 allocs, 14,501 frees, 3,261,322 bytes allocated
==12075==
==12075== LEAK SUMMARY:
==12075==    definitely lost: 3 bytes in 1 blocks
==12075==    indirectly lost: 0 bytes in 0 blocks
==12075==      possibly lost: 72 bytes in 3 blocks
==12075==    still reachable: 107,582 bytes in 23 blocks
==12075==         suppressed: 18,148 bytes in 158 blocks
==12075== Rerun with --leak-check=full to see details of leaked memory
==12075==
==12075== For counts of detected and suppressed errors, rerun with: -v
==12075== Use --track-origins=yes to see where uninitialised values come from
==12075== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 1 from 1)
```



```
definitely lost: 肯定内存泄露
indirectly lost: 非直接内存泄露
possibly lost: 可能发生内存泄露
still reachable: 仍然可访问的内存
suppressed: 外部造成的内存泄露
```

Callgrind 配合 php 扩展 xdebug 输出的 profile 分析日志文件可以分析程序运行期间各个函数调用时占用的内存、 CPU 占用情况。

总结

遇到了内存泄露时先观察是程序本身内存不足还是外部资源导致，然后搞清楚程序运行中用到了哪些资源：写入磁盘日志、连接数据库 SQL 查询、发送 Curl 请求、 Socket 通信等， I/O 操作必然会用到内存，如果这些地方都没有发生明显的内存泄露，检查哪里处理大量数据没有及时释放资源，如果是 php 5.3 以下版本还需考虑循环引用的问题。多了解一些 Linux 下的分析辅助工具，解决问题时可以事半功倍。

最后宣传一下穿云团队今年最新开源的应用透明链路追踪工具 Molten：https://github.com/chuan-yun/Molten。安装好php扩展后就能帮你实时收集程序的 curl,pdo,mysqli,redis,mongodb,memcached 等请求的数据，可以很方便的与 zipkin 集成。

参考资料

- http://php.net/manual/zh/features.gc.php
- http://www.php-internals.com/book/?p=chapt06/06-07-memory-leaks
- http://www.programering.com/a/MDN5UjMwATk.html
- https://stackoverflow.com/questions/20458136/using-valgrind-to-debug-a-php-cli-segmentation-fault
- http://www.laruence.com/2013/08/14/2899.html
- https://mengkang.net/873.html