Memory Usage Performance Guidelines

内存使用编程指南

1 Introduction概述

Memory is an important system resource that all programs use. Programs must be loaded into memory before they can run and, while running, they allocate additional memory (both explicitly and implicitly) to store and manipulate program-level data. Making room in memory for a program’s code and data requires time and resources and therefore affect the overall performance of the system. Although you cannot avoid using memory altogether, there are ways to minimize the impact your memory usage has on the rest of the system.

内存是所有程序使用的重要系统资源。在应用运行之前程序必须被加载到内存中，并且在运行中它们可以（显式的或隐式的）分配额外的内存以存储和操作程序级别的数据。在内存中为程序的代码和数据创造空间，需要消耗时间和资源，因此会影响到整个系统的性能。虽然你不能完全避免使用内存，但有办法最小化你的内存使用对系统其他部分的影响。

This document provides background information about the memory systems of OS X and iOS and how you use them efficiently. You can use this information to tune your program’s memory usage by ensuring you are allocating the right amount of memory at the right time. This document also provides tips on how to detect memory-related performance issues in your program.

本文提供了关于OS X和iOS内存系统的背景信息以及如何有效的使用它们。你可以利用这些信息，通过确保在正确的时间分配正确的内存数，调整你的程序的内存使用。本文还提供了关于如何在程序中检测内存相关问题的建议。

## 1.1 Organization of This Document本文的结构

This programming topic includes the following articles:

本程序主题包括以下文章：

* [About the Virtual Memory System](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html#//apple_ref/doc/uid/20001880-BCICIHAB) introduces the terminology and provides a high-level overview of the virtual memory systems of OS X and iOS.
* 《关于虚拟内存》介绍了一些术语并提供了OS X和iOS的虚拟内存系统的高级概述。
* [Tips for Allocating Memory](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html#//apple_ref/doc/uid/20001881-CJBCFDGA) describes the best techniques for allocating, initializing, and copying memory. It also describes the proper ways to respond to low-memory notifications in iOS.
* 《分配内存建议》描述了分配、初始化和复制内存的最佳技术。并介绍了在iOS中响应低内存通知的适当方法。
* [Caching and Purgeable Memory](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/CachingandPurgeableMemory.html#//apple_ref/doc/uid/TP40013104-SW1) discusses the benefits of caching, and how to avoid some of the problems that can arise from implementing caches. It also details the advantages of implementing purgeable memory into a caching system and how to successfully implement this beneficial technology.
* 《缓存和可清除的内存》讨论了缓存的好处以及如何避免在实现缓存中可能遇到的问题。同时，还详述了在缓存系统中实现可清除内存的优点以及如何成功的实现这种有益的技术。
* [Tracking Memory Usage](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/FindingPatterns.html#//apple_ref/doc/uid/20001882-CJBJFIDD) describes the tools and techniques for analyzing your application’s memory usage.
* 《跟踪内存使用》讲述了分析应用内存使用的工具和技术。
* [Finding Memory Leaks](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/FindingLeaks.html#//apple_ref/doc/uid/20001883-CJBJFIDD) describes the tools and techniques for finding memory leaks in your application.
* 《找到内存泄露》讲述了在应用中找到内存泄露的工具和技术。
* [Enabling the Malloc Debugging Features](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/MallocDebug.html#//apple_ref/doc/uid/20001884-CJBJFIDD) describes the environment variables used to enable malloc history logging. You must set some of these variables before using some of the memory analysis tools.
* 《启用分配内存调试特性》讲述了用于启用分配内存历史日志的环境变量。你必须在使用内存分析工具之前设置其中的一些变量。
* [Viewing Virtual Memory Usage](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/VMPages.html#//apple_ref/doc/uid/20001985-CJBJFIDD) describes the tools and techniques for analyzing your application’s in-memory footprint.
* 《查看虚拟内存使用》讲述了分析应用程序内存占用的工具和技术。

# 2 About the Virtual Memory System关于虚拟内存系统

Efficient memory management is an important aspect of writing high performance code in both OS X and iOS. Minimizing memory usage not only decreases your application’s memory footprint, it can also reduce the amount of CPU time it consumes. In order to properly tune your code though, you need to understand something about how the underlying system manages memory.

在OS X和iOS中，有效的内存管理都是撰写高性能代码的重要方面。最小化内存使用不仅可以减少程序的内存占用，也可以减少CPU时间的消耗。为了正确的调整你的代码，你需要了解一些关于底层系统如何管理内存的知识。

Both OS X and iOS include a fully-integrated virtual memory system that you cannot turn off; it is always on. Both systems also provide up to 4 gigabytes of addressable space per 32-bit process. In addition, OS X provides approximately 18 exabytes of addressable space for 64-bit processes. Even for computers that have 4 or more gigabytes of RAM available, the system rarely dedicates this much RAM to a single process.

OS X和iOS都包含了完整集成且无法关闭的虚拟内存系统；该系统始终在运行着。OS X和iOS也都为每个32位进程提供了4GB的可寻址空间。另外，OS X还为64位进程提供了接近18EB的可寻址空间。即使是拥有4GB或更大可用RAM的计算机，操作系统也很少为某一个进程提供如此大的内存。

To give processes access to their entire 4 gigabyte or 18 exabyte address space, OS X uses the hard disk to hold data that is not currently in use. As memory gets full, sections of memory that are not being used are written to disk to make room for data that is needed now. The portion of the disk that stores the unused data is known as the backing store because it provides the backup storage for main memory.

为了让进程可以访问到他们的整个4GB或18EB的寻址空间，OS X使用硬盘持有当前不用的数据。当内存满了，不在使用的内存段会被写入硬盘，给现在需要的数据留出空间。存储未使用的数据的磁盘部分被称为后备存储器，因为它为主存储器提供备份存储。

Although OS X supports a backing store, iOS does not. In iPhone applications, read-only data that is already on the disk (such as code pages) is simply removed from memory and reloaded from disk as needed. Writable data is never removed from memory by the operating system. Instead, if the amount of free memory drops below a certain threshold, the system asks the running applications to free up memory voluntarily to make room for new data. Applications that fail to free up enough memory are terminated.

尽管OS X提供了备份存储，但是iOS没有。在iPhone应用中，磁盘中准备好的只读数据（跟代码页一样）只是简单的从内存中移除并在需要时重新从磁盘加载。可写的数据从不会被操作系统从内存中移除。取而代之的是，如果可用内存的数量低于某一阈值，系统就会让应用程序自主释放内存为新数据制造空间。没有释放足够的内存的应用程序将被终止。

**Note:** Unlike most UNIX-based operating systems, OS X does not use a preallocated disk partition for the backing store. Instead, it uses all of the available space on the machine’s boot partition.

注意：不像大部分基于UNIX的操作系统，OS X不使用预分配磁盘区域作为备份存储，而是使用机器引导分区的所有可用空间。

The following sections introduce terminology and provide a brief overview of the virtual memory system used in both OS X and iOS. For more detailed information on how the virtual memory system works, see Kernel Programming Guide.

以下各节介绍术语和提供用于OS X和iOS系统的虚拟内存的简要概述。关于虚拟内存系统如何工作的更多详细信息，参见《内核编程指南》。

## 2.1 About Virtual Memory 关于虚拟内存

Virtual memory allows an operating system to escape the limitations of physical RAM. The virtual memory manager creates a logical address space (or “virtual” address space) for each process and divides it up into uniformly-sized chunks of memory called pages. The processor and its memory management unit (MMU) maintain a page table to map pages in the program’s logical address space to hardware addresses in the computer’s RAM. When a program’s code accesses an address in memory, the MMU uses the page table to translate the specified logical address into the actual hardware memory address. This translation occurs automatically and is transparent to the running application.

虚拟内存允许操作系统避开物理RAM的限制。虚拟内存管理器为每个进程创建一个逻辑地址空间（或者“虚拟”地址空间）并将其分为大小一致的内存块，被称为页（page）。处理器和内存管理单元（MMU）维护一个页表，将程序的逻辑地址空间里的页与计算机RAM的硬件地址进行映射。当程序代码访问内存中的地址时，MMU使用页表将指定的逻辑地址翻译成实际的硬盘内存地址。该翻译自动发生并对运行的应用程序是透明的。

As far as a program is concerned, addresses in its logical address space are always available. However, if an application accesses an address on a memory page that is not currently in physical RAM, a page fault occurs. When that happens, the virtual memory system invokes a special page-fault handler to respond to the fault immediately. The page-fault handler stops the currently executing code, locates a free page of physical memory, loads the page containing the needed data from disk, updates the page table, and then returns control to the program’s code, which can then access the memory address normally. This process is known as paging.

就一个程序而言，它的逻辑地址空间中的地址总是可用的。然而，如果应用程序访问内存页上的地址当前不在物理RAM上，则会出现一个页面错误。当发生这种情况时，虚拟内存系统立即调用一个特殊的页面错误处理器响应这个错误。页面错误处理器会中断当前执行的代码，定位物理内存的空闲页，从磁盘中加载包含所需数据的页，更新页表，然后将控制权返回给程序代码，这样程序代码就能正常访问内存地址。这个过程称为分页（paging）。

If there are no free pages available in physical memory, the handler must first release an existing page to make room for the new page. How the system release pages depends on the platform. In OS X, the virtual memory system often writes pages to the backing store. The backing store is a disk-based repository containing a copy of the memory pages used by a given process. Moving data from physical memory to the backing store is called paging out (or “swapping out”); moving data from the backing store back in to physical memory is called paging in (or “swapping in”). In iOS, there is no backing store and so pages are are never paged out to disk, but read-only pages are still be paged in from disk as needed.

如果物理内存中没有可用的空闲页，处理器必须首先释放一个已存在的页为新的页创建空间。系统如何释放页取决于平台。在OS X中，虚拟内存系统通常将页写入备份存储中。备份存储是一个基于磁盘的库，该库中包含了一个被给定进程使用的内存页的拷贝。从物理内存移动数据到备份存储的过程被称为页出（paging out）或“换出”（swapping out）；从备份存储移动数据到物理内存的过程被称为页入（paging in）或“换入”（swapping in）。在iOS中，没有备份存储，因此页永远不会换出到磁盘，但是只读页仍然按需从磁盘换入。

In both OS X and iOS, the size of a page is 4 kilobytes. Thus, every time a page fault occurs, the system reads 4 kilobytes from disk. Disk thrashing can occur when the system spends a disproportionate amount of time handling page faults and reading and writing pages, rather than executing code for a program.

在OS X和iOS中，页的大小都是4KB。因此，每当页面错误发生，系统就从磁盘读取4KB。当系统花费相当多的时间处理页面错误、频繁的读写页（在内存/硬盘间频繁的换入换出），这个时间超过了程序执行代码的时间，就会发生磁盘抖动。

Paging of any kind, and disk thrashing in particular, affects performance negatively because it forces the system to spend a lot of time reading and writing to disk. Reading a page in from the backing store takes a significant amount of time and is much slower than reading directly from RAM. If the system has to write a page to disk before it can read another page from disk, the performance impact is even worse.

任何形式的分页，特别是磁盘抖动，都会给性能带来消极影响，因为它强迫系统花费大量的时间读和写磁盘。从备份存储中读取页会花费相当多的时间，这笔直接从RAM重读取慢很多。如果系统不得不在从磁盘中读取页之前将另一个页写入磁盘，对性能的影响就更加糟糕了。

## 2.2 Details of the Virtual Memory System虚拟内存系统详述

The logical address space of a process consists of mapped regions of memory. Each mapped memory region contains a known number of virtual memory pages. Each region has specific attributes controlling such things as inheritance (portions of the region may be mapped from “parent” regions), write-protection, and whether it is wired (that is, it cannot be paged out). Because regions contain a known number of pages, they are page-aligned, meaning the starting address of the region is also the starting address of a page and the ending address also defines the end of a page.

进程的逻辑地址空间由已映射的内存区域组成。每个已映射的内存区域包含已知数量的虚拟内存页。每个区域都有特定的属性控制继承性（该区域的一部分可能是从“父”区域映射过来）、写保护以及是否联动的（即，不能换出）。因为区域包含已知数量的页面，所以它们是页面对齐的，这意味着区域的起始地址也是一个页面的起始地址并且区域的结束地址也定义了一个页面的结束地址。

The kernel associates a VM object with each region of the logical address space. The kernel uses VM objects to track and manage the resident and nonresident pages of the associated regions. A region can map to part of the backing store or to a memory-mapped file in the file system. Each VM object contains a map that associates regions with either the default pager or the vnode pager. The default pager is a system manager that manages the nonresident virtual memory pages in the backing store and fetches those pages when requested. The vnode pager implements memory-mapped file access. The vnode pager uses the paging mechanism to provide a window directly into a file. This mechanism lets you read and write portions of the file as if they were located in memory.

内核（kernel）将VM对象与逻辑地址空间的每一个区域关联起来。内核使用VM对象来跟踪和管理所关联区域的常驻和非常驻页面。一个区域可以映射到备份存储的一部分或者映射到一个文件系统的内存映射文件。每个VM对象包含了一个关联区域与默认寻页器（default paper）或虚拟节点寻页器（vnode paper）的映射。默认寻页器是一个系统管理器，它管理了备份存储中的非常驻虚拟内存页，并在请求时获取这些页。虚拟节点寻页器实现了内存映射文件的访问。虚拟节点寻页器使用分页机制提供了一个窗口直接通到文件。该机制让你读写文件的某些部分，就仿佛它们就在内存中一样。

In addition to mapping regions to either the default or vnode pager, a VM object may also map regions to another VM object. The kernel uses this self referencing technique to implement copy-on-write regions. Copy-on-write regions allow different processes (or multiple blocks of code within a process) to share a page as long as none of them write to that page. When a process attempts to write to the page, a copy of the page is created in the logical address space of the process doing the writing. From that point forward, the writing process maintains its own separate copy of the page, which it can write to at any time. Copy-on-write regions let the system share large quantities of data efficiently in memory while still letting processes manipulate those pages directly (and safely) if needed. These types of regions are most commonly used for the data pages loaded from system frameworks.

除了将区域映射到默认或虚拟节点寻页器，VM对象还会将区域映射到另一个VM对象。内存使用自引用技术实现拷贝-写（copy-on-write）区域。拷贝-写区域允许不同的进程（或者一个进程中的多个代码块）共享一个页，只要它们都没有写入这个页。当一个进程试图往该页中写入时，会在该进程做写操作的逻辑地址空间创建一个页面拷贝。从这一点开始，正在写的京城操作它自己单独的页面的拷贝，它可以随时写入。拷贝-写区域让系统在内存中有效的分享大量数据，同时仍然可以在需要的时候让进程直接（且安全的）操作这些页。这些类型的区域通常用于从系统框架中加载属于页。

Each VM object contains several fields, as shown in Table 1.

每个VM对象包含若干字段，如表1所示。

**Table 1**  Fields of the VM object

|  |  |
| --- | --- |
| **Field** | **Description** |
| Resident pages | A list of the pages of this region that are currently resident in physical memory. |
| Size | The size of the region, in bytes. |
| Pager | The pager responsible for tracking and handling the pages of this region in backing store. |
| Shadow | Used for copy-on-write optimizations. |
| Copy | Used for copy-on-write optimizations. |
| Attributes | Flags indicating the state of various implementation details. |

If the VM object is involved in a copy-on-write (vm\_copy) operation, the shadow and copy fields may point to other VM objects. Otherwise both fields are usually NULL.

如果在拷贝-写（vm\_copy）操作中涉及VM对象，shadow和copy字段可能指向另一个VM对象。否则这两个字段通常是NULL。

## 2.3 Wired Memory联动内存

Wired memory (also called resident memory) stores kernel code and data structures that must never be paged out to disk. Applications, frameworks, and other user-level software cannot allocate wired memory. However, they can affect how much wired memory exists at any time. For example, an application that creates threads and ports implicitly allocates wired memory for the required kernel resources that are associated with them.

联动内存（也被称为常驻内存）存储了永远不能换出到磁盘的内核代码和数据结构。应用程序、框架和其他用户级别的软件不能分配联动内存。但是，它们可以随时影响存在多少联动内存。例如，创建线程和端口的线程的应用程序会隐式的为所需的内核资源分配联动内存

Table 2 lists some of the wired-memory costs for application-generated entities.

表2列举了应用生成实体时的联动内存花费。

**Table 2**  Wired memory generated by user-level software

|  |  |
| --- | --- |
| **Resource** | **Wired Memory Used by Kernel** |
| Process | 16 kilobytes |
| Thread | blocked in a continuation—5 kilobytes; blocked—21 kilobytes |
| Mach port | 116 bytes |
| Mapping | 32 bytes |
| Library | 2 kilobytes plus 200 bytes for each task that uses it |
| Memory region | 160 bytes |

**Note:** These measurements may change with each new release of the operating system. They are provided here to give you a rough estimate of the relative cost of system resource usage.

注意：这些值在新的操作系统版本中可能改变。它们在此仅提供给你一个系统资源使用相对成本的粗略的估计。

As you can see, every thread, process, and library contributes to the resident footprint of the system. In addition to your application using wired memory, however, the kernel itself requires wired memory for the following entities:

正如你所知的，每个线程、进程和库都会增加系统的常驻占用。然而，除了应用会使用联动内存，内核自己也会在下列实体中用到联动内存：

* VM objects
* the virtual memory buffer cache
* I/O buffer caches
* drivers

Wired data structures are also associated with the physical page and map tables used to store virtual-memory mapping information, Both of these entities scale with the amount of available physical memory. Consequently, when you add memory to a system, the amount of wired memory increases even if nothing else changes. When a computer is first booted into the Finder, with no other applications running, wired memory can consume approximately 14 megabytes of a 64 megabyte system and 17 megabytes of a 128 megabyte system.

联动数据结构与物理页和用于存储虚拟内存映射信息的映射表相关联。这些实体都与可用物理内存的数量成正比。因此，当你将内存添加到系统中，即使没有其他事情改变联动内存的数量也是增加的。当计算机第一次启动进入Finder，没有运行任何程序的时候，联动内存的消耗大约是64MB系统14MB，128M系统17MB。

Wired memory pages are not immediately moved back to the free list when they become invalid. Instead they are “garbage collected” when the free-page count falls below the threshold that triggers page out events.

当联动内存页不可用的时候，它们不会立即被移动到空闲列表。而是当空闲页的数量低于触发换出事件的阈值时被“垃圾回收（garbage collected）”。

## 2.4 Page Lists in the Kernel内核中的页表

The kernel maintains and queries three system-wide lists of physical memory pages:

内核维护和查询了三个全系统的物理内存页的列表：

* The active list contains pages that are currently mapped into memory and have been recently accessed.
* 活动表包含了当前映射到内存且最近访问过的页。
* The inactive list contains pages that are currently resident in physical memory but have not been accessed recently. These pages contain valid data but may be removed from memory at any time.
* 失活表包含当前常驻物理内存但最近未被访问过的页。这些页包含了有效的数据但是随时可能从内存中移除。
* The free list contains pages of physical memory that are not associated with any address space of VM object. These pages are available for immediate use by any process that needs them.
* 空闲表包含了没有关联到任何VM对象地址空间的物理内存页。在任何进程需要它们时，这些页都可以立即使用。

When the number of pages on the free list falls below a threshold (determined by the size of physical memory), the pager attempts to balance the queues. It does this by pulling pages from the inactive list. If a page has been accessed recently, it is reactivated and placed on the end of the active list. In OS X, if an inactive page contains data that has not been written to the backing store recently, its contents must be paged out to disk before it can be placed on the free list. (In iOS, modified but inactive pages must remain in memory and be cleaned up by the application that owns them.) If an inactive page has not been modified and is not permanently resident (wired), it is stolen (any current virtual mappings to it are destroyed) and added to the free list. Once the free list size exceeds the target threshold, the pager rests.

当空闲表里的页数低于一个阈值（由物理内存的大小决定）时，寻页器会尝试平衡队列。它通过从失活表中拉出页面来实现平衡。如果一个页面最近被访问过，它会重新被激活并且放到活动表的末尾。在OS X中，如果一个失活页包含未写入备份存储的数据，其内容必须先换出到磁盘才能放入空闲表。（在iOS中，修改了但又失活的页面必须留在内存中，并且会被拥有它们的应用程序清除。）如果一个失活页没有修改过并且不是永久常驻（联动）的，它就会被窃取（所有当前关联到它的虚拟映射都会被销毁）并且加入到空闲表。一旦空闲表的大小超过了目标阈值，寻页器就开始休息。

The kernel moves pages from the active list to the inactive list if they are not accessed; it moves pages from the inactive list to the active list on a soft fault (see Paging In Process). When virtual pages are swapped out, the associated physical pages are placed in the free list. Also, when processes explicitly free memory, the kernel moves the affected pages to the free list.

如果页面没有被访问，内核会将它们从活动表移到失活表；当软错误时内核会将页面从失活表移到活动表（参见《换入过程》）。当虚拟页被换出时，相关联的物理页被置入空闲表。当进程显式的释放内存，内核也会将影响到的页面移入空闲表。

## 2.5 Paging Out Process换出过程

In OS X, when the number of pages in the free list dips below a computed threshold, the kernel reclaims physical pages for the free list by swapping inactive pages out of memory. To do this, the kernel iterates all resident pages in the active and inactive lists, performing the following steps:

在OS X中，当空闲表的页数下降到低于算出的阈值，内核会通过将失活的页换出内存为空闲表获得物理页。为此，内存在所有活动表和失活表的常驻页中迭代，执行以下步骤：

* If a page in the active list is not recently touched, it is moved to the inactive list.
* 如果一个页面在活动表中，且最近没有被触及，它会被移到失活表。
* If a page in the inactive list is not recently touched, the kernel finds the page’s VM object.
* 如果一个页面在失活表中，且最近没有被触及，内核会找到这个页面的VM对象。
* If the VM object has never been paged before, the kernel calls an initialization routine that creates and assigns a default pager object.
* 如果这个VM对象之前从未被分页，内核会调用初始化程序创建并分配一个默认的寻页器对象。
* The VM object’s default pager attempts to write the page out to the backing store.
* 这个VM对象的默认寻页器会尝试将这个页写出到备份存储。
* If the pager succeeds, the kernel frees the physical memory occupied by the page and moves the page from the inactive to the free list.
* 如果寻页器成功，内核会释放这个页面占用的物理内存，并将这个页面从失活表移到空闲表。

**Note:** In iOS, the kernel does not write pages out to a backing store. When the amount of free memory dips below the computed threshold, the kernel flushes pages that are inactive and unmodified and may also ask the running application to free up memory directly. For more information on responding to these notifications, see [Responding to Low-Memory Warnings in iOS](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html#//apple_ref/doc/uid/20001881-SW1).

注意：在iOS中，内核不会将页面写出到备份内存。当空闲内存的数量下降至低于算出的阈值时，内存会刷新失活且未修改的页面，也可能直接让运行的应用释放内存。关于响应这些通知的信息，参见《在iOS中响应低内存警告》。

## 2.6 Paging In Process换入过程

The final phase of virtual memory management moves pages into physical memory, either from the backing store or from the file containing the page data. A memory access fault initiates the page-in process. A memory access fault occurs when code tries to access data at a virtual address that is not mapped to physical memory. There are two kinds of faults:

虚拟内存管理的最后阶段是从备份存储或者从包含页面数据的文件中将页面移入物理内存。内存访问错误会启动换入过程。当代码尝试访问一个虚拟地址的数据而这个地址又没有映射到物理内存是，就会发生内存访问错误。这会导致两种错误：

* A soft fault occurs when the page of the referenced address is resident in physical memory but is currently not mapped into the address space of this process.
* 软错误：当引用地址的页面常驻在物理内存中，但是当前没有映射到该进程的地址空间时，会发生软错误。
* A hard fault occurs when the page of the referenced address is not in physical memory but is swapped out to backing store (or is available from a mapped file). This is what is typically known as a page fault.
* 硬错误：当引用地址的页面不在物理内存中而是被换出到了备份存储中（或通过映射文件可用）时，会发生硬错误。这就是通常所说的页面错误。

When any type of fault occurs, the kernel locates the map entry and VM object for the accessed region. The kernel then goes through the VM object’s list of resident pages. If the desired page is in the list of resident pages, the kernel generates a soft fault. If the page is not in the list of resident pages, it generates a hard fault.

当任何一种类型的错误发生时，内核会为访问的区域定位映射输入和VM对象。然后内核遍历VM对象的常驻页面列表运行。如果所需的页面在常驻页面列表中，内核会生成一个软错误。如果所需页面不在常驻页面的列表中，内核会生成一个硬错误。

For soft faults, the kernel maps the physical memory containing the pages to the virtual address space of the process. The kernel then marks the specific page as active. If the fault involved a write operation, the page is also marked as modified so that it will be written to backing store if it needs to be freed later.

对于软错误，内核将包含页面的物理内存映射到进程的虚拟地址空间。然后内核会将这个特定的页标记为活动的。如果该错误涉及一个写操作，这个页面也会被标记为修改过的，在以后如果这个页面需要被释放它就会被写入备份存储。

For hard faults, the VM object’s pager finds the page in the backing store or from the file on disk, depending on the type of pager. After making the appropriate adjustments to the map information, the pager moves the page into physical memory and places the page on the active list. As with a soft fault, if the fault involved a write operation, the page is marked as modified.

对于硬错误，VM对象寻页器会在备份存储中或者从磁盘文件里找到这个页面，在哪里找取决于寻页器的类型。经过对映射信息做适当的调整之后，寻页器将该页移入物理内存并置入活动表。跟软错误一样，如果该错误涉及一个写操作，这个页面就会被标记为修改过的。

# 3 Tips for Allocating Memory 分配内存的小窍门

Memory is an important resource for your application so it’s important to think about how your application will use memory and what might be the most efficient allocation approaches. Most applications do not need to do anything special; they can simply allocate objects or memory blocks as needed and not see any performance degradation. For applications that use large amount of memory, however, carefully planning out your memory allocation strategy could make a big difference.

内存是应用程序中的重要资源，因此思考应用程序如何使用内存以及什么才是最有效的分配方法是非常重要的。大部分程序不需要特别做什么；它们可以简单的按需要分配对象或者内存块，而不用考虑任何性能退化。然而，对于使用大量内存的应用，小心的计划你的内存分配策略会带来很大的不同。

The following sections describe the basic options for allocating memory along with tips for doing so efficiently. To determine if your application has memory performance problems in the first place, you need to use the Xcode tools to look at your application’s allocation patterns while it is running. For information on how to do that, see [Tracking Memory Usage](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/FindingPatterns.html#//apple_ref/doc/uid/20001882-CJBJFIDD).

后续章节描述了分配内存的基本选项，以及一些行之有效的建议。首先，要确定你的应用程序是否有内存性能问题，你需要使用Xcode工具在你的应用程序运行时查看其分配模式。关于如何实现这项工作的信息，参见《跟踪内存使用》。

## 3.1 Tips for Improving Memory-Related Performance 提升内存相关性能的窍门

As you design your code, you should always be aware of how you are using memory. Because memory is an important resource, you want to be sure to use it efficiently and not be wasteful. Besides allocating the right amount of memory for a given operation, the following sections describe other ways to improve the efficiency of your program’s memory usage.

在设计代码时，你就应该总是想到如何使用内存。因为内存是重要的资源，你要确保有效的使用它而没有浪费。除了为一个给定的操作分配合适的内存，下面的章节将描述其他的方法来提高程序的内存使用效率。

### 3.1.1 Defer Your Memory Allocations延迟内存分配

Every memory allocation has a performance cost. That cost includes the time it takes to allocate the memory in your program’s logical address space and the time it takes to assign that address space to physical memory. If you do not plan to use a particular block of memory right away, deferring the allocation until the time when you actually need it is the best course of action. For example, to avoid the appearance of your app launching slowly, minimize the amount of memory you allocate at launch time. Instead, focus your initial memory allocations on the objects needed to display your user interface and respond to input from the user. Defer other allocations until the user issues starts interacting with your application and issuing commands. This lazy allocation of memory saves time right away and ensures that any memory that is allocated is actually used.

每次内存分配都有性能消耗。这个消耗包括在程序的逻辑地址空间分配内存的时间和将该地址空间关联到物理内存的时间。如果你不打算马上使用某一内存块，推迟到你真正需要它时再进行分配，这是最好的行为。例如，为了避免你的应用出现启动慢，在启动时最小化分配的内存数量。取而代之的是，关注那些需要用于显示用户界面和响应用户输入的对象的初始内存分配。推迟其他分配，直到用户开始与应用进行交互并发出命令。这种内存的懒分配当时节约了时间，并确保每个被分配的内存都是实际使用了的。

Once place where lazy initialization can be somewhat tricky is with global variables. Because they are global to your application, you need to make sure global variables are initialized properly before they are used by the rest of your code. The basic approach often taken with global variables is to define a static variable in one of your code modules and use a public accessor function to get and set the value, as shown in Listing 1.

对于懒初始化比较棘手的地方是使用全局变量。因为它们对于应用是全局的，你需要在它们被其余代码使用之前已经正确的初始化了。用在全局变量上的基本方法是在代码模型中定义一个静态变量，并使用一个公共的访问器方法来获取和设置它的值，如代码1所示。

**Listing 1**  Lazy allocation of memory through an accessor

代码1 通过访问器实现内存的懒分配

|  |
| --- |
| MyGlobalInfo\* GetGlobalBuffer() |
| { |
| static MyGlobalInfo\* sGlobalBuffer = NULL; |
| if ( sGlobalBuffer == NULL ) |
| { |
| sGlobalBuffer = malloc( sizeof( MyGlobalInfo ) ); |
| } |
| return sGlobalBuffer; |
| } |

The only time you have to be careful with code of this sort is when it might be called from multiple threads. In a multithreaded environment, you need to use locks to protect the if statement in your accessor method. The downside to that approach though is that acquiring the lock takes a nontrivial amount of time and must be done every time you access the global variable, which is a performance hit of a different kind. A simpler approach would be to initialize all global variables from your application’s main thread before it spawns any additional threads.

对于这种类型的代码你唯一需要小心的时候是它被多线程调用时。在多线程环境中，你需要使用锁保护访问器方法中的if语句。这个方法的缺陷是获取锁会消耗相当大量的时间，而每次访问这个全局变量时都需要获取锁，这对于性能将是另一种打击。一个简单的方法是在创建任何子线程之前在应用程序的主线程完成所有全局变量的初始化。

### 3.1.2 Initialize Memory Blocks Efficiently有效的初始化内存块

Small blocks of memory, allocated using the malloc function, are not guaranteed to be initialized with zeroes. Although you could use the memset function to initialize the memory, a better choice is to use the calloc routine to allocate the memory in the first place. The calloc function reserves the required virtual address space for the memory but waits until the memory is actually used before initializing it. This approach is much more efficient than using memset, which forces the virtual memory system to map the corresponding pages into physical memory in order to zero-initialize them. Another advantage of using the calloc function is that it lets the system initialize pages as they’re used, as opposed to all at once.

使用malloc方法分配的小的内存块不能保证被初始化为0。尽管你可以使用memset方法初始化这片内存，但更好的选择是首先是用calloc程序分配内存。Calloc方法在初始化之前就为内存预留所需的内存地址空间，直到内存被实际使用。这个方便比使用memset更加有效，它强迫虚拟内存系统将相应的页映射到物理内存，以将它们初始化为0。使用calloc方法的另一个好处是它让系统在使用页面时初始化，而不是一次初始化所有的页。

### 3.1.3 Reuse Temporary Memory Buffers重用临时内存缓冲区

If you have a highly-used function that creates a large temporary buffer for some calculations, you might want to consider reusing that buffer rather than reallocating it each time you call the function. Even if your function needs a variable buffer space, you can always grow the buffer as needed using the realloc function. For multithreaded applications, the best way to reuse buffers is to add them to your thread-local storage. Although you could store the buffer using a static variable in your function, doing so would prevent you from using that function on multiple threads at the same time.

如果你有一个常用的方法创建了大量的临时缓冲区用于某些计算，你应该要考虑重用这些缓冲区而不是每次调用该方法时都重新分配。即使你的方法需要一个可变的缓冲空间，你也可以使用realloc方法按需扩展缓冲区。对于多线程应用，重用缓冲区的最佳方法是将它们添加到线程本地存储。尽管你可以在方法中使用一个静态变量存储缓冲区，但这么做会阻碍你在多线程中同时使用这个方法。

Caching buffers eliminates much of the overhead for functions that regularly allocate and free large blocks of memory. However, this technique is only appropriate for functions that are called frequently. Also, you should be careful not to cache too many large buffers. Caching buffers does add to the memory footprint of your application and should only be used if testing indicates it would yield better performance.

缓存缓冲区可以消除经常分配和释放大量内存块的方法的大量开销。然而，这个技术只适用于频繁调用的方法。同时，你也要小心不要缓存太多的大缓冲区。缓存缓冲区确实增加了程序的内存占用，应该只在测试表明它能带来更好的性能时才使用它。

### 3.1.4 Free Unused Memory释放不用的内存

For memory allocated using the malloc library, it is important to free up memory as soon as you are done using it. Forgetting to free up memory can cause memory leaks, which reduces the amount of memory available to your application and impacts performance. Left unchecked, memory leaks can also put your application into a state where it cannot do anything because it cannot allocate the required memory.

使用malloc库进行内存分配，在用完之后立即释放内存是非常重要的。忘记释放内存会导致内存泄露，这将减少应用程序可用的内存数量并影响性能。即使留着不管，内存泄露也会让应用程序进入一种无法做任何事情的状态，因为已经不能分配需要的内存了。

**Note:** Applications built using the Automatic Reference Counting (ARC) compiler option do not need to release Objective-C objects explicitly. Instead, the app must store strong references to objects it wants to keep and remove references to objects it does not need. When an object does not have any strong references to it, the compiler automatically releases it. For more information about supporting ARC, see [Transitioning to ARC Release Notes](https://developer.apple.com/library/ios/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226).

注意：使用ARC编译设置的应用程序包不需要明确的释放Objective-C对象。取而代之的是，应用程序必须存储对对象的强引用，并在不需要时移除对对象的引用。当对象没有任何对它的强引用时，编译器会自动释放它。关于ARC的更多信息，参见《过渡到ARC释放注意事项》。

No matter which platform you are targeting, you should always eliminate memory leaks in your application. For code that uses malloc, remember that being lazy is fine for allocating memory but do not be lazy about freeing up that memory. To help track down memory leaks in your applications, use the Instruments app.

无论你在哪个平台，你都应该减少应用程序中的内存泄露。对于使用malloc的代码，在分配内存时懒惰是好的，但是在释放内存时不要懒惰。要帮助跟踪应用程序中的内存泄露，请使用Instruments应用。

## 3.2 Memory Allocation Techniques内存分配技术

Because memory is such a fundamental resource, OS X and iOS both provide several ways to allocate it. Which allocation techniques you use will depend mostly on your needs, but in the end all memory allocations eventually use the malloc library to create the memory. Even Cocoa objects are allocated using the malloc library eventually. The use of this single library makes it possible for the performance tools to report on all of the memory allocations in your application.

因为内存是这样一种基本的资源，OS X与iOS都提供了若干分配内存的方法。你要使用哪种分配技术主要取决于你的需求，但在最后所有分配最终都使用malloc库创建内存。即使是Cocoa对象最终也是使用malloc库分配的。只使用了这个库让性能工具报告你应用中的所有内存分配成为可能。

If you are writing a Cocoa application, you might allocate memory only in the form of objects using the [alloc](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/clm/NSObject/alloc) method of [NSObject](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject). Even so, there may be times when you need to go beyond the basic object-related memory blocks and use other memory allocation techniques. For example, you might allocate memory directly using malloc in order to pass it to a low-level function call.

如果你正在写一个Cocoa应用，你可能只是使用NSObject的alloc方法以对象的形式分配内存。尽管如此，仍然有一些时候你需要在基本对象相关的内存块之外使用其他内存分配技术。例如，你可能直接使用malloc方法分配内存，以将其传递传给一个低级的方法调用。

The following sections provide information about the malloc library and virtual memory system and how they perform allocations. The purpose of these sections is to help you identify the costs associated with each type of specialized allocation. You should use this information to optimize memory allocations in your code.

接下来的章节提供了关于malloc库和虚拟内存系统以及他们如何完成分配的信息。这些章节的目的是帮助你识别与每种特别的分配相关联的消耗。你应该使用这些信息你代码中的内存分配。

**Note:** These sections assume you are using the system supplied version of the malloc library to do your allocations. If you are using a custom malloc library, these techniques may not apply.

注意：这些章节假定你证使用系统提供的版本的malloc库完成分配。如果你使用的是自定义的malloc库，这些技术可能不管用了。

### 3.2.1 Allocating Objects分配对象

For Objective-C based applications, you allocate objects using one of two techniques. You can either use the [alloc](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/clm/NSObject/alloc) class method, followed by a call to a class initialization method, or you can use the [new](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/clm/NSObject/new) class method to allocate the object and call its default [init](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/instm/NSObject/init) method in one step.

对于基于Objective-C的应用，你使用两种技术之一分配对象。你可以使用alloc类方法，接着调用类初始化方法，或者你可以使用new类方法，一次性分配对象和调用其默认的init方法。

After creating an object, the compiler’s ARC feature determines the lifespan of an object and when it should be deleted. Every new object needs at least one strong reference to it to prevent it from being deallocated right away. Therefore, when you create a new object, you should always create at least one strong reference to it. After that, you may create additional strong or weak references depending on the needs of your code. When all strong references to an object are removed, the compiler automatically deallocates it.

For more information about ARC and how you manage the lifespan of objects, see [Transitioning to ARC Release Notes](https://developer.apple.com/library/ios/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226).

### 3.2.2 Allocating Small Memory Blocks Using Malloc使用Malloc分配小的内存块

For small memory allocations, where small is anything less than a few virtual memory pages, malloc sub-allocates the requested amount of memory from a list (or “pool”) of free blocks of increasing size. Any small blocks you deallocate using the free routine are added back to the pool and reused on a “best fit” basis. The memory pool itself is comprised of several virtual memory pages that are allocated using the vm\_allocate routine and managed for you by the system.

When allocating any small blocks of memory, remember that the granularity for blocks allocated by the malloc library is 16 bytes. Thus, the smallest block of memory you can allocate is 16 bytes and any blocks larger than that are a multiple of 16. For example, if you call malloc and ask for 4 bytes, it returns a block whose size is 16 bytes; if you request 24 bytes, it returns a block whose size is 32 bytes. Because of this granularity, you should design your data structures carefully and try to make them multiples of 16 bytes whenever possible.

**Note:** By their nature, allocations smaller than a single virtual memory page in size cannot be page aligned.

### 3.2.3 Allocating Large Memory Blocks using Malloc使用Malloc分配大的内存块

For large memory allocations, where large is anything more than a few virtual memory pages, malloc automatically uses the vm\_allocate routine to obtain the requested memory. The vm\_allocate routine assigns an address range to the new block in the logical address space of the current process, but it does not assign any physical memory to those pages right away. Instead, the kernel does the following:

* It maps a range of memory in the virtual address space of this process by creating a map entry; the map entry is a simple structure that defines the starting and ending addresses of the region.
* The range of memory is backed by the default pager.
* The kernel creates and initializes a VM object, associating it with the map entry.

At this point there are no pages resident in physical memory and no pages in the backing store. Everything is mapped virtually within the system. When your code accesses part of the memory block, by reading or writing to a specific address in it, a fault occurs because that address has not been mapped to physical memory. In OS X, the kernel also recognizes that the VM object has no backing store for the page on which this address occurs. The kernel then performs the following steps for each page fault:

1. It acquires a page from the free list and fills it with zeroes.
2. It inserts a reference to this page in the VM object’s list of resident pages.
3. It maps the virtual page to the physical page by filling in a data structure called the pmap. The pmap contains the page table used by the processor (or by a separate memory management unit) to map a given virtual address to the actual hardware address.

The granularity of large memory blocks is equal to the size of a virtual memory page, or 4096 bytes. In other words, any large memory allocations that are not a multiple of 4096 are rounded up to this multiple automatically. Thus, if you are allocating large memory buffers, you should make your buffer a multiple of this size to avoid wasting memory.

**Note:** Large memory allocations are guaranteed to be page-aligned.

For large allocations, you may also find that it makes sense to allocate virtual memory directly using vm\_allocate, rather than using malloc. The example in Listing 2 shows how to use the vm\_allocate function.

**Listing 2**  Allocating memory with vm\_allocate

|  |
| --- |
| void\* AllocateVirtualMemory(size\_t size) |
| { |
| char\* data; |
| kern\_return\_t err; |
|  |
| // In debug builds, check that we have |
| // correct VM page alignment |
| check(size != 0); |
| check((size % 4096) == 0); |
|  |
| // Allocate directly from VM |
| err = vm\_allocate( (vm\_map\_t) mach\_task\_self(), |
| (vm\_address\_t\*) &data, |
| size, |
| VM\_FLAGS\_ANYWHERE); |
|  |
| // Check errors |
| check(err == KERN\_SUCCESS); |
| if(err != KERN\_SUCCESS) |
| { |
| data = NULL; |
| } |
|  |
| return data; |
| } |

### 3.2.4 Allocating Memory in Batches在皮处理器中分配内存

If your code allocates multiple, identically-sized memory blocks, you can use the malloc\_zone\_batch\_malloc function to allocate those blocks all at once. This function offers better performance than a series of calls to malloc to allocate the same memory. Performance is best when the individual block size is relatively small—less than 4K in size. The function does its best to allocate all of the requested memory but may return less than was requested. When using this function, check the return values carefully to see how many blocks were actually allocated.

Batch allocation of memory blocks is supported in OS X version 10.3 and later and in iOS. For information, see the /usr/include/malloc/malloc.h header file.

### 3.2.5 Allocating Shared Memory分配共享内存

Shared memory is memory that can be written to or read from by two or more processes. Shared memory can be inherited from a parent process, created by a shared memory server, or explicitly created by an application for export to other applications. Uses for shared memory include the following:

* Sharing large resources such as icons or sounds
* Fast communication between one or more processes

Shared memory is fragile and is generally not recommended when other, more reliable alternatives are available. If one program corrupts a section of shared memory, any programs that also use that memory share the corrupted data. The functions used to create and manage shared memory regions are in the /usr/include/sys/shm.h header file.

### 3.2.6 Using Malloc Memory Zones使用分配内存区域

All memory blocks are allocated within a malloc zone (also referred to as a malloc heap). A zone is a variable-size range of virtual memory from which the memory system can allocate blocks. A zone has its own free list and pool of memory pages, and memory allocated within the zone remains on that set of pages. Zones are useful in situations where you need to create blocks of memory with similar access patterns or lifetimes. You can allocate many objects or blocks of memory in a zone and then destroy the zone to free them all, rather than releasing each block individually. In theory, using a zone in this way can minimize wasted space and reduce paging activity. In reality, the overhead of zones often eliminates the performance advantages associated with the zone.

**Note:** The term zone is synonymous with the terms heap, pool, and arena in terms of memory allocation using the malloc routines.

By default, allocations made using the malloc function occur within the default malloc zone, which is created when malloc is first called by your application. Although it is generally not recommended, you can create additional zones if measurements show there to be potential performance gains in your code. For example, if the effect of releasing a large number of temporary (and isolated) objects is slowing down your application, you could allocate them in a zone instead and simply deallocate the zone.

If you are create objects (or allocate memory blocks) in a custom malloc zone, you can simply free the entire zone when you are done with it, instead of releasing the zone-allocated objects or memory blocks individually. When doing so, be sure your application data structures do not hold references to the memory in the custom zone. Attempting to access memory in a deallocated zone will cause a memory fault and crash your application.

**Warning:** You should never deallocate the default zone for your application.

At the malloc library level, support for zones is defined in /usr/include/malloc/malloc.h. Use the malloc\_create\_zone function to create a custom malloc zone or the malloc\_default\_zone function to get the default zone for your application. To allocate memory in a particular zone, use the malloc\_zone\_malloc , malloc\_zone\_calloc , malloc\_zone\_valloc , or malloc\_zone\_realloc functions. To release the memory in a custom zone, call malloc\_destroy\_zone.

## 3.3 Copying Memory Using Malloc使用Malloc拷贝内存

There are two main approaches to copying memory in OS X: direct and delayed. For most situations, the direct approach offers the best overall performance. However, there are times when using a delayed-copy operation has its benefits. The goal of the following sections is to introduce you to the different approaches for copying memory and the situations when you might use those approaches.

### 3.3.1 Copying Memory Directly直接拷贝内存

The direct copying of memory involves using a routine such as memcpy or memmove to copy bytes from one block to another. Both the source and destination blocks must be resident in memory at the time of the copy. However, these routines are especially suited for the following situations:

* The size of the block you want to copy is small (under 16 kilobytes).
* You intend to use either the source or destination right away.
* The source or destination block is not page aligned.
* The source and destination blocks overlap.

If you do not plan to use the source or destination data for some time, performing a direct copy can decrease performance significantly for large memory blocks. Copying the memory directly increases the size of your application’s working set. Whenever you increase your application’s working set, you increase the chances of paging to disk. If you have two direct copies of a large memory block in your working set, you might end up paging them both to disk. When you later access either the source or destination, you would then need to load that data back from disk, which is much more expensive than using vm\_copy to perform a delayed copy operation.

**Note:** If the source and destination blocks overlap, you should prefer the use of memmove over memcpy. The implementation of memmove handles overlapping blocks correctly in OS X, but the implementation of memcpy is not guaranteed to do so.

### 3.3.2 Delaying Memory Copy Operations延迟内存拷贝操作

If you intend to copy many pages worth of memory, but don’t intend to use either the source or destination pages immediately, then you may want to use the vm\_copy function to do so. Unlike memmove or memcpy, vm\_copy does not touch any real memory. It modifies the virtual memory map to indicate that the destination address range is a copy-on-write version of the source address range.

The vm\_copy routine is more efficient than memcpy in very specific situations. Specifically, it is more efficient in cases where your code does not access either the source or destination memory for a fairly large period of time after the copy operation. The reason that vm\_copy is effective for delayed usage is the way the kernel handles the copy-on-write case. In order to perform the copy operation, the kernel must remove all references to the source pages from the virtual memory system. The next time a process accesses data on that source page, a soft fault occurs, and the kernel maps the page back into the process space as a copy-on-write page. The process of handling a single soft fault is almost as expensive as copying the data directly.

### 3.3.3 Copying Small Amounts of Data拷贝少量数据

If you need to copy a small blocks of non-overlapping data, you should prefer memcpy over any other routines. For small blocks of memory, the GCC compiler can optimize this routine by replacing it with inline instructions to copy the data by value. The compiler may not optimize out other routines such as memmove or BlockMoveData.

### 3.3.4 Copying Data to Video RAM将数据拷贝到视频RAM

When copying data into VRAM, use the BlockMoveDataUncachedfunction instead of functions such as bcopy. The bcopy function uses cache-manipulation instructions that may cause exception errors. The kernel must fix these errors in order to continue, which slows down performance tremendously.

## 3.4 Responding to Low-Memory Warnings in iOS 在iOS中响应低内存警告

The virtual memory system in iOS does not use a backing store and instead relies on the cooperation of applications to remove strong references to objects. When the number of free pages dips below the computed threshold, the system releases unmodified pages whenever possible but may also send the currently running application a low-memory notification. If your application receives this notification, heed the warning. Upon receiving it, your application must remove strong references to as many objects as possible. For example, you can use the warnings to clear out data caches that can be recreated later.

UIKit provides several ways to receive low-memory notifications, including the following:

* Implement the [applicationDidReceiveMemoryWarning:](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/occ/intfm/UIApplicationDelegate/applicationDidReceiveMemoryWarning:) method of your application delegate.
* Override the [didReceiveMemoryWarning](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/index.html#//apple_ref/occ/instm/UIViewController/didReceiveMemoryWarning) method in your custom UIViewController subclass.
* Register to receive the [UIApplicationDidReceiveMemoryWarningNotification](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html#//apple_ref/c/data/UIApplicationDidReceiveMemoryWarningNotification) notification.

Upon receiving any of these notifications, your handler method should respond by immediately removing strong references to objects. View controllers automatically remove references to views that are currently offscreen, but you should also override the [didReceiveMemoryWarning](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/index.html#//apple_ref/occ/instm/UIViewController/didReceiveMemoryWarning) method and use it to remove any additional references that your view controller does not need.

If you have only a few custom objects with known purgeable resources, you can have those objects register for the UIApplicationDidReceiveMemoryWarningNotification notification and remove references there. If you have many purgeable objects or want to selectively purge only some objects, however, you might want to use your application delegate to decide which objects to keep.

**Important:** Like the system applications, your applications should always handle low-memory warnings, even if they do not receive those warnings during your testing. System applications consume small amounts of memory while processing requests. When a low-memory condition is detected, the system delivers low-memory warnings to all running programs (including your application) and may terminate some background applications (if necessary) to ease memory pressure. If not enough memory is released—perhaps because your application is leaking or still consuming too much memory—the system may still terminate your application.

# 4 Caching and Purgeable Memory 缓存和可擦除内存

Caching and purgeable memory can be vital assets to developers who are dealing with large objects that require significant memory or computation time, or developers whose code is getting bogged down as the computer writes data to disk because the RAM is full.

## 4.1 Overview of Caching缓存概述

A cache is a collection of objects or data that can greatly increase the performance of applications.

### 4.1.1 Why Use Caching?为什么要使用缓存？

Developers use caches to store frequently accessed objects with transient data that can be expensive to compute. Reusing these objects can provide performance benefits, because their values do not have to be recalculated. However, the objects are not critical to the application and can be discarded if memory is tight. If discarded, their values will have to be recomputed again when needed.

### 4.1.2 Problems Caching Can Cause缓存会导致的问题

Although caching can provide enormous benefits in terms of performance, there are also some possible drawbacks that caching presents. Most importantly, caching can use very large amounts of memory. When caching many large data objects, it is possible to cache so many objects that there is no RAM left for other applications, and the computer will grind to a halt as it writes all of this data to disk in order to free up RAM.

### 4.1.3 Solutions 解决

Cocoa provides an [NSCache](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSCache_Class/index.html#//apple_ref/occ/cl/NSCache) object as a convenient storage container for items you want to cache, while at the same time addressing the memory management issues discussed above. The NSCache class is very similar to the [NSDictionary](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSDictionary_Class/index.html#//apple_ref/occ/cl/NSDictionary) class, in that they both hold key-value pairs. However, an NSCache object is a “reactive cache.” That is, when memory is available, it aggressively caches any data it is given. Yet, when memory is low, it will automatically discard some of its elements in order to free up memory for other applications. Later, if these discarded items are needed, their values will have to be recalculated.

NSCache provides two other useful "limit" features: limiting the number of cached elements and limiting the total cost of all elements in the cache. To limit the number of elements that the cache is allowed to have, call the method [setCountLimit:](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSCache_Class/index.html#//apple_ref/occ/instm/NSCache/setCountLimit:). For example, if you try to add 11 items to a cache whose [countLimit](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSCache_Class/index.html#//apple_ref/occ/instm/NSCache/countLimit) is set to 10, the cache could automatically discard one of the elements.

When adding items to a cache, you can specify a cost value to be associated with each key-value pair. Call the [setTotalCostLimit:](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSCache_Class/index.html#//apple_ref/occ/instm/NSCache/setTotalCostLimit:) method to set the maximum value for the sum of all the cached objects’ costs. Thus, when an object is added that pushes the totalCost above the totalCostLimit, the cache could automatically evict some of its objects in order to get back below the threshold. This eviction process is not guaranteed, so trying to manipulate the cost values to achieve specific behavior could be detrimental to the performance of the cache. Pass in 0 for the cost if you have nothing useful, or use the [setObject:forKey:](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSCache_Class/index.html#//apple_ref/occ/instm/NSCache/setObject:forKey:) method, which does not require a cost to be passed in.

**Note:** The count limit and the total-cost limit are not strictly enforced. That is, when the cache goes over one of its limits, some of its objects might get evicted immediately, later, or never, all depending on the implementation details of the cache.

## 4.2 Using Purgeable Memory使用可擦除内存

The Cocoa framework also provides the [NSPurgeableData](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSPurgeableData_Class/index.html#//apple_ref/occ/cl/NSPurgeableData) class to help ensure that your applications do not use up too much memory. The NSPurgeableData class adopts the [NSDiscardableContent](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSDiscardableContent_Protocol/index.html#//apple_ref/occ/intf/NSDiscardableContent) protocol, which any class can implement to allow memory to be discarded when clients of the class's instances are finished accessing those objects. You should implement NSDiscardableContent when creating objects that have disposable subcomponents. In addition, the NSPurgeableData class does not have to be used in conjunction with NSCache; you may use it independently to get purging behavior.

### 4.2.1 Advantages of Using Purgeable Memory使用可擦除内存的好处

By using purgeable memory, you allow the system to quickly recover memory if it needs to, thereby increasing performance. Memory that is marked as purgeable is not paged to disk when it is reclaimed by the virtual memory system because paging is a time-consuming process. Instead, the data is discarded, and if needed later, it will have to be recomputed.

A caveat when using purgeable memory is that the block of memory must be locked before being accessed. This locking mechanism is necessary to ensure that no auto-removal policies try to dispose of the data while you are accessing it. Similarly, the locking mechanism will ensure that the virtual memory system has not already discarded the data. The NSPurgeableData class implements a very simple locking mechanism to ensure that the data is safe while it is being read.

### 4.2.2 How to Implement Purgeable Memory 如何实现可擦除内存

The NSPurgeableData class is very simple to use, because the class simply implements the NSDiscardableContent protocol. Then notion of a “counter” variable is central to the life cycle of NSDiscardableContent objects. When the memory being used by this object is being read, or is still needed, its counter variable will be greater than or equal to 1. When it is not being used, and can be discarded, the counter variable is equal to 0.

When the counter is equal to 0, the block of memory may be discarded if memory is tight. To discard the content, call [discardContentIfPossible](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSDiscardableContent_Protocol/index.html#//apple_ref/occ/intfm/NSDiscardableContent/discardContentIfPossible) on the object, which frees the associated memory if the counter variable equals 0.

By default, when an NSPurgeableData object is initialized, it is created with the counter variable equal to 1 and can safely be accessed. To access purgeable memory, simply call the [beginContentAccess](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSDiscardableContent_Protocol/index.html#//apple_ref/occ/intfm/NSDiscardableContent/beginContentAccess) method. This method will first check to make sure the object’s data has not been discarded. If the data is still there, it will increment the counter variable in order to protect the memory while it is being read, and return [YES](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/doc/c_ref/YES). If the data has been discarded, this method will return [NO](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/doc/c_ref/NO). When you are done accessing the data, call [endContentAccess](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSDiscardableContent_Protocol/index.html#//apple_ref/occ/intfm/NSDiscardableContent/endContentAccess), which decrements the counter and allows the memory to be discarded if the system desires to do so. You must keep track of the counter variable’s state and access memory only if the beginContentAccess method returns [YES](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/doc/c_ref/YES).

The system or client objects call the [discardContentIfPossible](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSDiscardableContent_Protocol/index.html#//apple_ref/occ/intfm/NSDiscardableContent/discardContentIfPossible) method to discard the purgeable data if the system’s available memory is running low. This method will only discard the data if its counter variable is 0, and otherwise does nothing. Lastly, the [isContentDiscarded](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSDiscardableContent_Protocol/index.html#//apple_ref/occ/intfm/NSDiscardableContent/isContentDiscarded) method returns [YES](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/doc/c_ref/YES) if the memory has been discarded.

Below is an example of a life cycle for an NSPurgeableData object:

|  |
| --- |
| NSPurgeableData \* data = [[NSPurgeableData alloc] init]; |
| [data endContentAccess]; //Don't necessarily need data right now, so mark as discardable. |
| //Maybe put data into a cache if you anticipate you will need it later. |
|  |
| ... |
|  |
| if([data beginContentAccess]) { //YES if data has not been discarded and counter variable has been incremented |
| ...Some operation on the data... |
| [data endContentAccess] //done with the data, so mark as discardable |
| } else { |
| //data has been discarded, so recreate data and complete operation |
| data = ... |
| [data endContentAccess]; //done with data |
| } |
|  |
| //data is able to be discarded at this point if memory is tight |

### 4.2.3 Purgeable Memory and NSCache 可擦除内存和NSCache

When objects that implement the NSDiscardableContent protocol are put in NSCache objects, the cache keeps a strong reference to the object. However, if an object’s content has been discarded and the cache’s [evictsObjectsWithDiscardedContent](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSCache_Class/index.html#//apple_ref/occ/instm/NSCache/evictsObjectsWithDiscardedContent) value is set to [YES](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/doc/c_ref/YES), the object is automatically removed from the cache and is not found by a lookup call.

### 4.2.4 Some Warnings About Purgeable Memory 关于可擦除内存的一些警告

A caveat to using purgeable memory is that only large objects or chunks of memory can use it directly. The purgeable memory API acts on multi page virtual memory objects, which makes it hard to mark a single small cache element as purgeable. The caching API will do the required bookkeeping to allow small cache elements to use purgeable memory. Likewise, there will be cases where it is awkward to allocate the memory for cache elements through the API directly, such as when a convenience method is used to allocate an object or when an object is allocated in a different layer than the layer doing the caching. In such cases, it is not possible to use purgeable memory.

## 4.3 When to Use Purgeable Memory 什么时候使用可擦除内存

It makes sense to use purgeable memory when the expected cost of purging is less than the expected cost of paging — when the cost of paging is greater than the cost of recomputing the particular data value times the probability that that data item is reused. Many caches fall into this category because their speculative nature makes it likely that the items will not be used again. Similarly, cache items that are easily recomputed are prime candidates for purgeable memory, because the application will not take a huge performance hit if their values have to be recalculated.

# 5 Tracking Memory Usage 跟踪内存使用

If you suspect your code is not using memory as efficiently as it could, the first step in determining if there is a problem is to gather some baseline data. Monitoring your code using one of the Apple-provided performance tools can give you a picture of your code’s memory usage and may highlight potential problems or point to areas that need further examination. The following sections describe the tools most commonly used for memory analysis and when you might want to use them.

如果你怀疑你的代码没有最有效的使用内存，判断代码中是否有问题的第一步是收集一些基础数据。使用一个苹果提供的性能工具监测你的代码就可以得出你代码的内存使用图，并且可以高亮标出潜在的问题或者指出需要进一步检查的区域。接下来的章节介绍了最常用的内存分析工具，以及什么时候可以使用它们。

## 5.1 Tracking Allocations With Instruments 使用Instruments跟踪分配

The Instruments application is always a good starting point for doing any sort of performance analysis. Instruments is an integrated, data-gathering environment that uses special modules (called instruments) to gather all sorts of information about a process. Instruments can operate on your application’s shipping binary file—you do not need to compile special modules into your application to use it. The library of the Instruments application contains four modules that are designed specifically for gathering memory-related data. These modules are as follows:

对于任何种类的性能分析，Instruments应用程序都是一个不错的开始。Instruments是一个集成的数据收集环境，可以使用指定的模块（被称为instrument）收集一个进程的各种信息。Instruments可以在你的程序的运行着的二进制文件上操作——不需要将指定模块编译到你的程序中就可以使用。Instruments应用的库包含了4个专为收集内存相关数据而设计的模块。它们是：

* The ObjectAlloc instrument records and displays a history of all memory allocations since the launch of your application.
* ObjectAlloc仪器记录和显示了从你的应用启动开始的所有内存分配历史。
* The Leaks instrument looks for allocated memory that is no longer referenced by your program’s code; see [Finding Leaks Using Instruments](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/FindingLeaks.html#//apple_ref/doc/uid/20001883-SW2).
* Leaks仪器可以查找你的程序代码中已分配却不再被引用的内存；参见《使用Instruments找到内存泄露》。
* The Shared Memory instrument monitors the opening and unlinking of shared memory regions.
* Shared Memory仪器监测了共享内存区域的打开和分离。
* The Memory Monitor instrument measures and records the system’s overall memory usage.
* Memory Monitor仪器测量和记录了系统的全部内存使用。

You can add any or all of these instruments to a single trace document and gather data for each of them simultaneously. Being able to gather the data all at once lets you correlate information from one instrument with the others. For example, the Leaks instrument is often combined with the ObjectAlloc instrument so that you can both track the allocations and find out which ones were leaked.

你可以添加任意或者全部仪器到同一个跟踪文件中，并同时为每个仪器收集数据。一次收集所有数据的能力让你可以将一个仪器的信息与其他的相互关联。例如，Leaks仪器通常与ObjectAlloc仪器相关联，这样你就可以在跟踪内存分配的同时找出哪一个被泄露了。

After gathering data for your application, you need to analyze it. The following sections provide tips on how to analyze data using several of the memory-related instruments. For information on how to use the Leaks instrument, see [Finding Leaks Using Instruments](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/FindingLeaks.html#//apple_ref/doc/uid/20001883-SW2).

在为应用程序收集了数据之后，还需要分析它。以下章节提供了如何使用若干内存相关一起分析数据的建议。关于如何使用Leaks仪器的信息，参见《使用Instruments找到内存泄露》。

### 5.1.1 Analyzing Memory Allocations with the ObjectAlloc Instrument 使用ObjectAlloc 仪器分析内存分配

You use the ObjectAlloc instrument to track the memory allocation activity for your application. This instrument tracks memory allocations from the time your application is launched until you stop recording data. The instrument shows you the allocations in terms of the number of objects created and the size (or type) of the objects that were created. In the icon viewing mode, the instrument displays a real-time histogram that lets you see changes and trends directly as they occur. It also retains a history of allocations and deallocations, giving you an opportunity to go back and see where those objects were allocated.

你使用ObjectAlloc仪器来跟踪应用程序的内存分配活动。该仪器从应用开始时跟踪内存分配，直到你停止记录数据。该仪器按照创建的对象数量和大小（或类型）向你展示内存分配。在图标查看模式下，该仪器显示实时的直方图，让你在分配发生时直接看到变化和趋势。它还保留了分配和销毁的历史，给你一个机会回去看看这些对象是在哪儿分配的。

The information displayed by the ObjectAlloc instrument is recorded by an allocation statistics facility built into the Core Foundation framework. When this facility is active, every allocation and deallocation is recorded as it happens.

ObjectAlloc仪器显示的信息由编入Core Foundation框架的分配统计模块记录。当这个模块活跃时，每个分配和销毁发生时都会被记录。

### 5.1.2 Analyzing Shared Memory Usage 分析共享内存使用

For Mac apps, the Shared Memory instrument tracks calls to any shm\_open and shm\_unlink functions, which are used for opening and closing regions of shared memory in the system. You can use this information to find out where your application is getting references to shared memory regions and to examine how often those calls are being made. The detail pane displays the list of each function call along with information about the calling environment at the time of the call. Specifically, the pane lists the executable that initiated the call and the function parameters. Opening the Extended Detail pane shows the stack trace associated with the call.

对于Mac应用，Shared Memory仪器跟踪了所有shm\_open和shm\_unlink方法的调用，这两个方法用于打开和关闭系统中的共享内存区域。你可以使用该仪器来找出你的应用在哪儿引用了共享内存区域以及检查这些方法被调用的频率。详情窗格显示了每一个方法调用的列表以及调用时的调用环境信息。特别需要指出的是，该窗格还列出了发起调用的可执行程序和方法参数。打开扩展详情窗格可以展示关联到调用的栈轨迹。

### 5.1.3 Analyzing Data from the Memory Monitor Instrument 通过Memory Monitor仪器分析数据

For Mac apps, the Memory Monitor instrument displays assorted statistics about system memory usage. You can use this instrument to view the memory usage trends in your application or on the entire system. For example, you can see how much memory is currently active, inactive, wired, and free. You can also see the amount of memory that has been paged in or out. You might use this instrument in conjunction with other instruments to track the amount of memory your application uses with respect to specific operations.

对于Mac应用，Memory Monitor仪器显示了有关系统内存使用的各种统计数据。你可以使用该仪器查看你的应用或实体系统中的内存使用趋势。例如，你可以看到当前有多少内存是活动的、失活的、联动的和空闲的。你也能看到被换入或换出的内存数量。你可以使用该仪器与其他仪器协作，以跟踪某个特定操作时应用程序的内存使用量。

## 5.2 Tracking Memory Allocations With malloc\_history 使用malloc\_history跟踪内存分配

In OS X, the malloc\_history tool displays backtrace data showing exactly where your program made calls to the malloc and free functions. If you specify an address when calling malloc\_history, the tool tracks memory allocations occurring at that address only. If you specify the -all\_by\_size or -all\_by\_count options, the tool displays all allocations, grouping frequent allocations together.

Before using the malloc\_history tool on your program, you must first enable the malloc library logging features by setting the MallocStackLogging to 1. You may also want to set the MallocStackLoggingNoCompact environment variable to retain information about freed blocks. For more information on these variables, see [Enabling the Malloc Debugging Features](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/MallocDebug.html#//apple_ref/doc/uid/20001884-CJBJFIDD).

The malloc\_history tool is best used in situations where you need to find the previous owner of a block of memory. If you determine that a particular data is somehow becoming corrupted, you can put checks into your code to print the address of the block when the corruption occurs. You can then use malloc\_history to find out who owns the block and identify any stale pointers.

The malloc\_history tool is also suited for situations where Sampler or MallocDebug cannot be used. For example, you might use this tool from a remote computer or in situations where you want a minimal impact on the behavior of your program.

For more information on using the malloc\_history tool, see a target="\_self" malloc\_history/a man page.

## 5.3 Examining Memory With the heap Tool 使用heap工具检查内存

In OS X, the heap command-line tool displays a snapshot of the memory allocated by the malloc library and located in the address space of a specified process. For Cocoa applications, this tool identifies Objective-C objects by name. For both memory blocks and objects, the tool organizes the information by heap, showing all items in the same heap together.

The heap tool provides much of the same information as the ObjectAlloc instrument, but does so in a much less intrusive manner. You can use this tool from a remote session or in situations where the use of Instruments might slow the system down enough to affect the resulting output.

For more information about using the heap tool, see a target="\_self" heap(1)/a man page.

# 6 Finding Memory Leaks 找到内存泄露

Memory leaks are blocks of allocated memory that the program no longer references. Leaks waste space by filling up pages of memory with inaccessible data and waste time due to extra paging activity. Leaked memory eventually forces the system to allocate additional virtual memory pages for the application, the allocation of which could have been avoided by reclaiming the leaked memory.

For apps that use malloc, memory leaks are bugs and should always be fixed. For apps that use only Objective-C objects, the compiler’s ARC feature deallocates objects for you and generally avoids the problem of memory leaks. However, apps that mix the use of Objective-C objects and C-based structures must manage the ownership of objects more directly to ensure that the objects are not leaked.

The malloc library can only reclaim the memory you tell it to reclaim. If you call malloc or any routine that allocates memory, you must balance that call with a corresponding free. A typical memory leak occurs when you forget to deallocate memory for a pointer embedded in a data structure. If you allocate memory for embedded pointers in your code, make sure you release the memory for that pointer prior to deallocating the data structure itself.

Another typical memory leak example occurs when you allocate memory, assign it to a pointer, and then assign a different value to the pointer without freeing the first block of memory. In this example, overwriting the address in the pointer erases the reference to the original block of memory, making it impossible to release.

## 6.1 Finding Leaks Using Instruments 使用Instruments找到内存泄露

The Instruments application can be used to find leaks in both OS X and iPhone applications. To find leaks, create a new document template in Instruments and add the Leaks instrument to it. The Leaks instrument provides leak-detection capabilities identical to those in the leaks command-line tool. The Leaks instrument records all allocation events that occur in your application and then periodically searches the application’s writable memory, registers, and stack for references to any active memory blocks. If it does not find a reference to a block in one of these places, it deems the block a “leak” and displays the relevant information in the Detail pane.

In the Detail pane, you can view leaked memory blocks using Table and Outline modes. In Table mode, Instruments displays the complete list of leaked blocks, sorted by size. Selecting an entry in the table and clicking the arrow button next to the memory address shows the allocation history for the memory block at that address. Selecting an entry from this allocation history then shows the stack trace for that event in the Extended Detail pane of the document window. In Outline mode, the Leaks instrument displays leaks organized by call tree, which you can use to get information about the leaks in a particular branch of your code.

For more information about using the Instruments application, including more information about the information displayed by the Leaks instrument, see [Instruments User Guide](https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/index.html#//apple_ref/doc/uid/TP40004652).

## 6.2 Using the leaks Tool 使用内存泄露工具

In OS X, the leaks command-line tool searches the virtual memory space of a process for buffers that were allocated by malloc but are no longer referenced. For each leaked buffer it finds, leaks displays the following information:

* the address of the leaked memory
* the size of the leak (in bytes)
* the contents of the leaked buffer

If leaks can determine that the object is an instance of an Objective-C or Core Foundation object, it also displays the name of the object. If you do not want to view the contents of each leaked buffer, you can specify the -nocontext option when calling leaks. If the MallocStackLogging environment variable is set and you are running your application in gdb, leaks displays a stack trace describing where the buffer was allocated. For more information on malloc debugging options, see [Enabling the Malloc Debugging Features](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/MallocDebug.html#//apple_ref/doc/uid/20001884-CJBJFIDD).

**Note:** The leaks command-line tool is located in /usr/bin.

## 6.3 Tips for Improving Leak Detection 提升内存泄露侦测能力的窍门

The following guidelines can help you find memory leaks quickly in your program. Most of these guidelines are intended to be used with the leaks tool but some are also applicable for use with MallocDebug and general use.

1. Run leaks during unit testing. Because unit testing exercises all code paths in a repeatable manner, you are more likely to find leaks than you would be if you were testing your code in a production environment.
2. Use the -exclude option of leaks to filter out leaks in functions with known memory leaks. This option helps reduce the amount of extraneous information reported by leaks.
3. If leaks reports a leak intermittently, set up a loop around the target code path and run the code hundreds or thousands of times. This increases the likelihood of the leak reappearing more regularly.
4. Run your program against libgmalloc.dylib in gdb. This library is an aggressive debugging malloc library that can help track down insidious bugs in your code. For more information, see the libgmalloc man page.
5. For Cocoa and iPhone applications, if you fix a leak and your program starts crashing, your code is probably trying to use an already-freed object or memory buffer. Set the NSZombieEnabled environment variable to [YES](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/doc/c_ref/YES) to find messages to already freed objects.

Most unit testing code executes the desired code paths and exits. Although this is perfectly normal for unit testing, it creates a problem for the leaks tool, which needs time to analyze the process memory space. To fix this problem, you should make sure your unit-testing code does not exit immediately upon completing its tests. You can do this by putting the process to sleep indefinitely instead of exiting normally.

# 7 Enabling the Malloc Debugging Features 开启Malloc调试特性

Debugging memory-related bugs can be time consuming if you do not know where to start looking. This is usually compounded by the problem that many memory bugs occur well after the memory in question was manipulated by the code. Fortunately, Xcode includes options for identifying memory problems closer to when those problems actually happen.

## 7.1 Enabling Guard Malloc 启用Guard Malloc

Guard Malloc is a special version of the malloc library that replaces the standard library during debugging. Guard Malloc uses several techniques to try and crash your application at the specific point where a memory error occurs. For example, it places separate memory allocations on different virtual memory pages and then deletes the entire page when the memory is freed. Subsequent attempts to access the deallocated memory cause an immediate memory exception rather than a blind access into memory that might now hold other data. When the crash occurs, you can then go and inspect the point of failure in the debugger to identify the problem.

To enable debugging using Guard Malloc, configure your project to run with Guard Malloc in [Xcode’s scheme editor](https://developer.apple.com/library/ios/recipes/xcode_help-scheme_editor/Articles/SchemeDiagnostics.html#//apple_ref/doc/uid/TP40010402-CH9). You can use this option for Mac apps and iOS apps running in the simulator.

For more information about the types of memory problems that Guard Malloc can help you track down, see the a target="\_self" libgmalloc/a man page in a target="\_self" OS X Man Pages/a.

## 7.2 Configuring the Malloc Environment Variables 配置Malloc环境变量

The malloc library provides debugging features to help you track down memory smashing bugs, heap corruption, references to freed memory, and buffer overruns. You enable these debugging options through a set of environment variables. With the exception of MallocCheckHeapStart and MallocCheckHeapEach, the value for most of these environment variables is ignored. To disable a variable from Terminal, use the a target="\_self" unset/a command. Table 1 lists some of the key environment variables and describes their basic function. For a complete list of variables, see the [malloc](https://developer.apple.com/library/ios/documentation/System/Conceptual/ManPages_iPhoneOS/man3/malloc.3.html#//apple_ref/doc/man/3/malloc) man page.

**Table 1**  Malloc environment variables

|  |  |
| --- | --- |
| **Variable** | **Description** |
| MallocStackLogging | If set, malloc remembers the function call stack at the time of each allocation. |
| MallocStackLoggingNoCompact | This option is similar to MallocStackLogging but makes sure that all allocations are logged, no matter how small or how short lived the buffer may be. |
| MallocScribble | If set, free sets each byte of every released block to the value 0x55. |
| MallocPreScribble | If set, malloc sets each byte of a newly allocated block to the value 0xAA. This increases the likelihood that a program making assumptions about freshly allocated memory fails. |
| MallocGuardEdges | If set, malloc adds guard pages before and after large allocations. |
| MallocDoNotProtectPrelude | Fine-grain control over the behavior of MallocGuardEdges: If set, malloc does not place a guard page at the head of each large block allocation. |
| MallocDoNotProtectPostlude | Fine-grain control over the behavior of MallocGuardEdges: If set, malloc does not place a guard page at the tail of each large block allocation. |
| MallocCheckHeapStart | Set this variable to the number of allocations before malloc will begin validating the heap. If not set, malloc does not validate the heap. |
| MallocCheckHeapEach | Set this variable to the number of allocations before malloc should validate the heap. If not set, malloc does not validate the heap. |

The following example enables stack logging and heap checking in the current shell before running an application. The value for MallocCheckHeapStart is set to 1 but is irrelevant and can be set to any value you want. You could also set these variables from your shell’s startup file, although if you do be sure to a target="\_self" export/a each variable.

|  |
| --- |
| % MallocStackLogging=1 |
| % MallocCheckHeapStart=1000 |
| % MallocCheckHeapEach=100 |
| % ./my\_tool |

If you want to run your program in gdb, you can set environment variables from the Xcode debugging console using the command set env, as shown in the following example:

|  |
| --- |
| % gdb |
| (gdb) set env MallocStackLogging 1 |
| (gdb) run |

Some of the performance tools require these options to be set in order to gather their data. For example, the malloc\_history tool can identify the allocation site of specific blocks if the MallocStackLogging flag is set. This tool can also describe the blocks previously allocated at an address if the MallocStackLoggingNoCompact environment variable is set. The leaks command line tool will name the allocation site of a leaked buffer if MallocStackLogging is set. See the man pages for leaks and malloc\_history for more details.

### 7.2.1 Detecting Double Freed Memory 侦测二次释放的内存

The malloc library reports attempts to call free on a buffer that has already been freed. If you have enabled the MallocStackLoggingNoCompact option, you can use the logged stack information to find out where in your code the second free call was made. You can then use this information to set up an appropriate breakpoint in the debugger and track down the problem.

The malloc library reports information to stderr.

### 7.2.2 Detecting Heap Corruption 侦测堆腐化

To enable heap checking, assign values to the MallocCheckHeapStart and MallocCheckHeapEach environment variables. You must set both of these variables to enable heap checking. The MallocCheckHeapStart variable tells the malloc library how many malloc calls to process before initiating the first heap check. Set the second to the number of malloc calls to process between heap checks.

The MallocCheckHeapStart variable is useful when the heap corruption occurs at a predictable time. Once it hits the appropriate start point, the malloc library starts logging allocation messages to the Terminal window. You can watch the number of allocations and use that information to determine approximately where the heap is being corrupted. Adjust the values for MallocCheckHeapStart and MallocCheckHeapEach as necessary to narrow down the actual point of corruption.

### 7.2.3 Detecting Memory Smashing Bugs 侦测内存损毁的错误

To find memory smashing bugs, enable the MallocScribble variable. This variable writes invalid data to freed memory blocks, the execution of which causes an exception to occur. When using this variable, you should also set the MallocStackLogging and MallocStackLoggingNoCompact variables to log the location of the exception. When the exception occurs, you can then use the malloc\_history command to track down the code that allocated the memory block. You can then use this information to track through your code and look for any lingering pointers to this block.

# 8 Viewing Virtual Memory Usage 查看虚拟内存使用

If you need more detailed information about virtual memory usage, you can use the top, vm\_stat, pagestuff, and vmmap command-line tools for analyzing your Mac apps. The information returned by these tools ranges from summary information about all the system processes to detailed information about a specific process.

The following sections provide information on using the vm\_stat, pagestuff, and vmmap tools to gather detailed memory information. For more information on using Instruments to analyze memory, see [Instruments User Guide](https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/index.html#//apple_ref/doc/uid/TP40004652) and the other articles in this document. For information on how to use the top tool, see [Performance Overview](https://developer.apple.com/library/ios/documentation/Performance/Conceptual/PerformanceOverview/index.html).

## 8.1 Viewing Virtual Memory Statistics 查看虚拟内存统计

The vm\_stat tool displays high-level statistics about the current virtual memory usage of the system. By default, vm\_stat displays these statistics once, but you can specify an interval value (in seconds) to update these statistics continuously. For information on the usage of this tool, see the a target="\_self" vm\_stat/a man page.

Listing 1 shows an example of the output from vm\_stat.

**Listing 1**  Output of vm\_stat tool

|  |
| --- |
| Mach Virtual Memory Statistics: (page size of 4096 bytes) |
| Pages free: 3194. |
| Pages active: 34594. |
| Pages inactive: 17870. |
| Pages wired down: 9878. |
| "Translation faults": 6333197. |
| Pages copy-on-write: 81385. |
| Pages zero filled: 3180051. |
| Pages reactivated: 343961. |
| Pageins: 33043. |
| Pageouts: 78496. |
| Object cache: 66227 hits of 96952 lookups (68% hit rate) |

## 8.2 Viewing Mach-O Code Pages 查看Mach-O代码页

The pagestufftool displays information about the specified logical pages of a file conforming to the Mach-O executable format. For each specified page of code, symbols (function and static data structure names) are displayed. All pages in the \_\_TEXT, \_\_text section are displayed if no page numbers are given.

Listing 2 shows part of the output from pagestuff for the TextEdit application. This output is the result of running the tool with the -a option, which prints information about all of the executable’s code pages. It includes the virtual address locations of each page and the type of information on that page.

**Listing 2**  Partial output of pagestuff tool

|  |
| --- |
| File Page 0 contains Mach-O headers |
| File Page 1 contains Mach-O headers |
| File Page 2 contains contents of section (\_\_TEXT,\_\_text) |
| Symbols on file page 2 virtual address 0x3a08 to 0x4000 |
| File Page 3 contains contents of section (\_\_TEXT,\_\_text) |
| Symbols on file page 3 virtual address 0x4000 to 0x5000 |
| File Page 4 contains contents of section (\_\_TEXT,\_\_text) |
| Symbols on file page 4 virtual address 0x5000 to 0x6000 |
|  |
| ... |
|  |
| File Page 22 contains contents of section (\_\_TEXT,\_\_cstring) |
| File Page 22 contains contents of section (\_\_TEXT,\_\_literal4) |
| File Page 22 contains contents of section (\_\_TEXT,\_\_literal8) |
| File Page 22 contains contents of section (\_\_TEXT,\_\_const) |
| Symbols on file page 22 virtual address 0x17000 to 0x17ffc |
| File Page 23 contains contents of section (\_\_DATA,\_\_data) |
| File Page 23 contains contents of section (\_\_DATA,\_\_la\_symbol\_ptr) |
| File Page 23 contains contents of section (\_\_DATA,\_\_nl\_symbol\_ptr) |
| File Page 23 contains contents of section (\_\_DATA,\_\_dyld) |
| File Page 23 contains contents of section (\_\_DATA,\_\_cfstring) |
| File Page 23 contains contents of section (\_\_DATA,\_\_bss) |
| File Page 23 contains contents of section (\_\_DATA,\_\_common) |
| Symbols on file page 23 virtual address 0x18000 to 0x18d48 |
| 0x00018000 \_NXArgc |
| 0x00018004 \_NXArgv |
| 0x00018008 \_environ |
| 0x0001800c \_\_\_progname |
| ... |

In the preceding listing, if a page exports any symbols, those symbols are also displayed by the -a option. If you want to view the symbols for a single page, pass in the desired page number instead of the -a option. For more information about the pagestuff tool and its supported options, see the a target="\_self" pagestuff/a man page.

## 8.3 Viewing Virtual Memory Regions 查看虚拟内存区域

The vmmap and vmmap64 tools display the virtual memory regions allocated for a specified process. These tools provide access to the virtual memory of 32-bit and 64-bit applications, respectively. You can use them to understand the purpose of memory at a given address and how that memory is being used. For each virtual-memory region, these tools display the type of page, the starting address, region size (in kilobytes), read/write permissions, sharing mode, and the purpose of the pages in that region.

The following sections show you how to interpret the output from the vmmap tool. For more information about the vmmap and vmmap64 tools, see the a target="\_self" vmmap/a or a target="\_self" vmmap64/a man pages.

### 8.3.1 Sample Output From vmmap 从vmmap采样输出

Listing 3 shows some sample output from the vmmap tool. This example is not a full listing of the tool’s output but is an abbreviated version showing the primary sections.

**Listing 3**  Typical output of vmmap

|  |
| --- |
| ==== Non-writable regions for process 313 |
| \_\_PAGEZERO 0 [ 4K] ---/--- SM=NUL ...ts/MacOS/Clock |
| \_\_TEXT 1000 [ 40K] r-x/rwx SM=COW ...ts/MacOS/Clock |
| \_\_LINKEDIT e000 [ 4K] r--/rwx SM=COW ...ts/w/Clock |
| 90000 [ 4K] r--/r-- SM=SHM |
| 340000 [3228K] r--/rwx SM=COW 00000100 00320... |
| 789000 [3228K] r--/rwx SM=COW 00000100 00320... |
| Submap 90000000-9fffffff r--/r-- machine-wide submap |
| \_\_TEXT 90000000 [ 932K] r-x/r-x SM=COW /usr/lib/libSystem.B.dylib |
| \_\_LINKEDIT 900e9000 [ 260K] r--/r-- SM=COW /usr/lib/libSystem.B.dylib |
| \_\_TEXT 90130000 [ 740K] r-x/r-x SM=COW .../Versions/A/CoreFoundation |
| \_\_LINKEDIT 901e9000 [ 188K] r--/r-- SM=COW .../Versions/A/CoreFoundation |
| \_\_TEXT 90220000 [2144K] r-x/r-x SM=COW .../Versions/A/CarbonCore |
| \_\_LINKEDIT 90438000 [ 296K] r--/r-- SM=COW .../Versions/A/CarbonCore |
|  |
| [...data omitted...] |
|  |
| ==== Writable regions for process 606 |
| \_\_DATA 18000 [ 4K] rw-/rwx SM=PRV /Contents/MacOS/TextEdit |
| \_\_OBJC 19000 [ 8K] rw-/rwx SM=COW /Contents/MacOS/TextEdit |
| MALLOC\_OTHER 1d000 [ 256K] rw-/rwx SM=PRV |
| MALLOC\_USED(DefaultMallocZone\_0x5d2c0) 5d000 [ 256K] rw-/rwx SM=PRV |
| 9d000 [ 372K] rw-/rwx SM=COW 33320000 00000020 00000000 00001b84... |
| VALLOC\_USED(DefaultMallocZone\_0x5d2c0) ff000 [ 36K] rw-/rwx SM=PRV |
| MALLOC\_USED(CoreGraphicsDefaultZone\_0x10 108000 [ 256K] rw-/rwx SM=PRV |
| MALLOC\_USED(CoreGraphicsRegionZone\_0x148 148000 [ 256K] rw-/rwx SM=PRV |
|  |
| [...data omitted...] |
|  |
| Submap a000b000-a012ffff r--/r-- process-only submap |
| \_\_DATA a0130000 [ 28K] rw-/rw- SM=COW .../Versions/A/CoreFoundation |
| Submap a0137000-a021ffff r--/r-- process-only submap |
| \_\_DATA a0220000 [ 20K] rw-/rw- SM=COW .../Versions/A/CarbonCore |
| Submap a0225000-a048ffff r--/r-- process-only submap |
| \_\_DATA a0490000 [ 12K] rw-/rw- SM=COW .../IOKit.framework/Versions/A/IOKit |
| Submap a0493000-a050ffff r--/r-- process-only submap |
| \_\_DATA a0510000 [ 36K] rw-/rw- SM=COW .../Versions/A/OSServices |
| b959e000 [ 4K] rw-/rw- SM=SHM |
| b95a0000 [ 4K] rw-/rw- SM=SHM |
| b9630000 [ 164K] rw-/rw- SM=SHM |
| b965a000 [ 896K] rw-/rw- SM=SHM |
| bff80000 [ 504K] rw-/rwx SM=ZER |
| STACK[0] bfffe000 [ 4K] rw-/rwx SM=PRV |
| bffff000 [ 4K] rw-/rwx SM=PRV |
| \_\_DATA c000c000 [ 4K] rw-/rwx SM=PRV .../Versions/A/ApplicationEnhancer |
| STACK[1] f0001000 [ 512K] rw-/rwx SM=PRV |
| ff002000 [12272K] rw-/rw- SM=SHM |
|  |
| ==== Legend |
| SM=sharing mode: |
| COW=copy\_on\_write PRV=private NUL=empty ALI=aliased |
| SHM=shared ZER=zero\_filled S/A=shared\_alias |
|  |
| ==== Summary for process 313 |
| ReadOnly portion of Libraries: Total=27420KB resident=12416KB(45%) swapped\_out\_or\_unallocated=15004KB(55%) |
| Writable regions: Total=21632KB written=536KB(2%) resident=1916KB(9%) swapped\_out=0KB(0%) unallocated=19716KB(91%) |

If you specify the -d parameter (plus an interval in seconds), vmmap takes two snapshots of virtual-memory usage—one at the beginning of a specified interval and the other at the end—and displays the differences. It shows three sets of differences:

* individual differences
* regions in the first snapshot that are not in the second
* regions in the second snapshot that are not in the first

### 8.3.2 Interpreting vmmap’s Output 解释vmmap的输出

The columns of vmmap output have no headings. Instead you can interpret the type of data in each column by its format. Table 1 describes these columns.

**Table 1**  Column descriptions for vmmap

|  |  |  |
| --- | --- | --- |
| **Column Number** | **Example** | **Description** |
| 1 | \_\_TEXT, \_\_LINKEDIT, MALLOC\_USED, STACK, and so on | The purpose of the memory. This column can contain the name of a Mach-O segment or the memory allocation technique. |
| 2 | (DefaultMallocZone\_0x5d2c0) | If present, the zone used for allocation. |
| 3 | 4eee000 | The virtual memory address of the region. |
| 4 | [ 124K] | The size of the region, measured in kilobytes |
| 5 | rw-/rwx | Read, write and execution permissions for the region. The first set of flags specifies the current protection for the region. The second set of values specifies the maximum protection for the region. If an entry contains a dash (-), the process does not have the target permission. |
| 6 | SM=PRV | Sharing mode for the region, either COW (copy-on-write), PRV (private), NUL (empty), ALI (aliased), or SHM (shared). |
| 7 | ...ts/MacOS/Clock | The end of the pathname identifying the executable mapped into this region of virtual memory. If the region is stack or heap memory, nothing is displayed in this column. |

Column 1 identifies the purpose of the memory. A \_\_TEXT segment contains read-only code and data. A \_\_DATA segment contains data that may be both readable and writable. For allocated data, this column shows how the memory was allocated, such as on the stack, using malloc, and so on. For regions loaded from a library, the far right column shows the name of the library loaded into memory.

The size of the virtual memory region (column 4) represents the total size reserved for that region. This number may not reflect the actual number of memory pages allocated for the region. For example, calling vm\_allocate reserves a set of memory pages but does not allocate any physical memory until the pages are actually touched. Similarly, a memory-mapped file may reserve a set of pages, but the system does not load pages until a read or write event occurs on the file.

The protection mode (column 5) describes the access restrictions for the memory region. A memory region contains separate flags for read, write, and execution permissions. Each virtual memory region has a current permission, and a maximum permission. In the output from vmmap, the current permission appears first followed by the maximum permission. Thus, if the permissions are “r--/rwx“ the page is currently read-only but allows read, write, and execution access as its maximum allowed permissions. Typically, the current permissions do not permit writing to a region. However, these permissions may change under certain circumstances. For example, a debugger may request write access to a page in order to set a breakpoint.

**Note:** Pages representing part of a Mach-O executable are usually not writable. The first page (\_\_PAGEZERO, starting at address 0x00000000) has no permissions set. This ensures that any reference to a NULL pointer immediately causes an error. The page just before the stack is similarly protected so that stack overflows cause the application to crash immediately.

The sharing mode (SM= field) tells you whether pages are shared between processes and what happens when pages are modified. Private pages (PRV) are visible only to the process and are allocated as they are used. Private pages can also be paged out to disk. Copy-on-write (COW) pages are shared by multiple processes (or shared by a single process in multiple locations). When the page is modified, the writing process then receives its own copy of the page. Empty (NUL) sharing implies that the page does not really exist in physical memory. Aliased (ALI) and shared (SHM) memory are shared between processes.

The sharing mode typically describes the general mode controlling the region. For example, as copy-on-write pages are modified, they become private to the application. However, the region containing those private pages is still copy-on-write until all pages become private. Once all pages are private, the sharing mode changes to private.

Some lines in the output of vmmap describe submaps. A submap is a shared set of virtual memory page descriptions that the operating system can reuse between multiple processes. For example, the memory between 0x90000000 and 0xAFFFFFFF is a submap containing the most common dynamic libraries. Submaps minimize the operating system’s memory usage by representing the virtual memory regions only once. Submaps can either be shared by all processes (machine-wide) or be local to the process (process-only). If the contents of a machine-wide submap are changed—for example, the debugger makes a section of memory for a dynamic library writable so it can insert debugging traps—then the submap becomes local, and the kernel allocates memory to store the extra copy.