一、存储拓展

1、内存简介

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和IOS都包括一个完全集成的虚拟内存系统。你不能关闭它，它总是在。同时系统还提供高达4GB的寻址空间为32位的机器上。此外，OS X 64位进程提供约18EB寻址空间。即使是那些有4GB或更多的可用的RAM字节计算机系统很少把它全部分给某一个进程。

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.

程序可以访问他们的整个4GB或18EB的地址空间，OS X用的硬盘存储当前没有使用的数据。当内存变满时，将不使用的内存部分写到磁盘上，以便为现在需要的数据腾出空间。存储未使用数据的磁盘的一部分称为后备存储器，因为它为主存储器提供备份存储。

Exabyte往往亦可以指Exbibyte(EiB)，其换算公式是：

1EiB = 1,024 PiB

1EiB = 1,048,576 (1024)TiB

1EiB = 1,073,741,824 (1024)GiB

1EiB = 1,099,511,627,776 (1024)MiB

1EiB = 1,125,899,906,842,624 (1024)KiB

1EiB = 1,152,921,504,606,846,976 (1024)B

2、iPhone的内存机制

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.

iOS不支持后备存储，在iPhone的应用程序，只读数据已经是在磁盘（如代码页）是从内存中删除和重装需要从磁盘。操作系统永远不会从内存中删除可写数据。相反，如果空闲内存的数量降到一定的阈值以下，系统要求正在运行的应用程序自动释放内存以腾出空间容纳新数据。无法释放足够内存的应用程序被终止。

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.

虚拟内存让操作系统摆脱物理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“虚拟”地址空间），并将其划分为统一大小的内存块，称为页。处理器和内存管理单元（MMU）维护一个页表映射程

序的逻辑地址空间的页在计算机的RAM的硬件地址。当一个程序的代码访问一个内存地址，MMU通过页表翻译指定的逻辑地址转化为实际的硬件内存地址。这种转换是自动发生的，对运行的应用程序是透明的。

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.

在iOS中，没有后台存储，所以页面从不被分页到磁盘，但只读页面仍然需要从磁盘中分页。

In OS X and in earlier versions of iOS, the size of a page is 4 kilobytes. In later versions of iOS, A7- and A8-based systems expose 16-kilobyte pages to the 64-bit userspace backed by 4-kilobyte physical pages, while A9 systems expose 16-kilobyte pages backed by 16-kilobyte physical pages. These sizes determine how many kilobytes the system reads from disk when a page fault occurs. 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. In iOS, modified but inactive pages must remain in memory and be cleaned up by the application that owns them.

在OS X和iOS的早期版本中，一个页面的大小是4KB。在iOS版本的A7和A8，系统将16KB的页面映射到64位用户的支持4KB物理页面，而A9系统将16KB的虚拟地址空间映射到16KB的物理页面。这些尺寸，多少KB取决于系统从磁盘读取页时发生故障。当系统花费过多的时间处理页面错误和读写页面，而不是为程序执行代码时，可能会出现磁盘抖动。 在iOS中，修改但不活动的页面必须留在内存中，并由它们的应用程序清理。

3、改善应用内存使用的建议

3.1 延迟内存分配(Defer Your Memory Allocations)

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.

每个内存分配都有性能开销。该成本包括在程序的逻辑地址空间中分配内存所需的时间以及分配地址空间到物理内存所需的时间。如果您不打算立即使用特定的内存块，将分配推迟到您真正需要的时候才是最好的做法。

Listing 1 Lazy allocation of memory through an accessor

```
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.

唯一需要仔细处理此类代码的时间是从多线程调用它的时间。在多线程环境中，你需要使用锁来保护如果你访问器方法的声明。这种方法的缺点是获取锁需要花费大量的时间，每次访问全局变量时都必须这样做，这是不同类型的性能命中。一个简单的方法，将初始化的全局变量在应用程序的主线程在它产生任何额外的线程之前。

3.2 有效的初始化内存(Initialize Memory Blocks Efficiently)

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.

小内存块使用malloc函数分配，不保证被初始化为零。虽然你可以使用memset函数初始化内存，更好的选择是使用calloc首选分配内存。函数calloc申请所需的虚拟地址空间的内存但等到实际使用时才初始化。这种方法比使用memset的更高效，memset会使使系统虚拟内存映射到物理内存页并初始化为零。使用calloc函数

的另一个优点是，它可以让系统初始化页面。

3.3 复用临时存储(Reuse Temporary Memory Buffers)

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.

如果你有一个高频使用的功能，创造了一些计算的大型临时缓冲区，你可能要考虑重用缓冲区而不是每次调用时都重建缓冲区。即使你的功能需要一个可变的缓冲空间，你可以使用realloc函数实现一个可增长的缓冲区。对于多线程应用程序，重用缓冲区的最佳方法是将它们添加当前线程空间中。虽然可以在函数中使用静态变量存储缓冲区，但这样做会阻止您同时在多个线程上使用该函数。

3.4 释放不适用的内存(Free Unused Memory)

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，使用后尽快的释放内存是很重要的。忘记释放内存可能导致内存泄漏，这会减少应用程序可用内存的数量并影响性能。如果不加以控制，内存泄漏也会使应用程序处于无法执行任何操作的状态，因为它不能分配所需的内存。

4、 内存分配

4.1、 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.

在创建对象之后，编译器的ARC特性决定了对象的生命周期和对象何时被删除。每一个新的对象需要至少一个强引用它以防止它被释放。因此，当您创建一个新对象时，您应该至少创建一个强引用。之后您可以根据代码的需要创建额外的强引用或弱引用。当一个对象的所有强引用被移除，编译器自动释放它。

4.2、使用malloc分配小内存（Allocating Small Memory Blocks Using Malloc）

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.

当分配小内存块，记得用malloc库分块粒度为16个字节。因此，您可以分配的最小内存块是16字节，大于此的任何块都是16的倍数。例如，如果你调用malloc和要求4字节，它返回一个块的大小为16字节；如果你要求24个字节，它返回一个块的大小是32字节。由于这种粒度，您应该仔细设计数据结构，并尽可能使它们成为16字节的倍数。

4.3、用malloc分配大量内存(Allocating Large Memory Blocks using Malloc)

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

(1)、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. (2)、The range of memory is backed by the default pager. (3)、The kernel creates and initializes a VM object, associating it with the map entry.

大内存分配，任何更多的虚拟内存页是大，malloc自动使用vm*allocate常规获取请求的内存。常规的vm*allocate分配一个地址范围到新块到当前进程的逻辑地址空间，但它不分配任何物理内存的页面吧。相反内核执行以下操作 （1）它通过创建一个映射条目映射这个进程的虚拟地址空间中的一系列内存；map条目是一个简单的结构，它定义了该区域的起始地址和结束地址。 （2）内存的范围由默认的寻呼机支持。（3）内核创建并初始化一个VM对象，将它与map条目关联起来。

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;
}
```

4.4、分配一批内存

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.

如果你的代码分配多个大小相同的内存块，你可以使用malloc*zone*batch_malloc函数一次分配这些内存块。当分配相同的内存时，此功能提供了比malloc性能更好。当单个块大小小于4K大小时，性能最好。函数尽最大努力分配所有请求的内存，但可能返回小于请求的内存。使用此函数时，请仔细检查返回值，看看实际分配了多少块。

4.5、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.

共享内存是可以通过两个或多个进程写入或读取的内存。共享内存可以从父进程继承，由共享内存服务器创建，或者由应用程序显式创建，以便导出到其他应用程序。

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.

5、Using Malloc Memory Zones

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.

所有内存块malloc分配一个区域内（也被称为malloc堆）。区域是一个可变大小的虚拟内存范围，内存系统可以分配块。区域有自己的空闲列表和内存页池，分区中分配的内存仍保留在该页上。**在需要创建具有类似访问模式或生存期的内存块的情况下，区域是有用的。**您可以在一个区域中分配许多对象或内存块，然后销毁该区域以释放它们，而不是单独释放每个块。从理论上讲，用这种方式使用区域可以减少浪费的空间并减少分页活动。实际上，区域的开销常常消除了与区域相关的性能优势。

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.

6、内存拷贝

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.

6.1 直接拷贝内存(Copying Memory Directly)

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.

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.

6.2 延迟内存拷贝(Delaying Memory Copy Operations)

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.

6.3 拷贝少量数据(Copying Small Amounts of Data)

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

6.4 Copying Data to Video 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.

7、iOS低内存警告处理(Responding to Low-Memory Warnings in iOS)

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.

IOS中的虚拟内存系统不使用后备存储，而是依赖应用程序的协作来删除对对象的强引用。当空闲页面的数量低于所计算的阈值时，系统尽可能地释放未修改的页面，但也可能发送当前正在运行的应用程序的低内存通知。如果您的应用程序收到此通知，请注意警告。在接收到它时，应用程序必须尽可能地删除强引用到尽可能多的对象。例如，您可以使用警告清除稍后可以重新创建的数据缓存。

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

(1)、Implement the applicationDidReceiveMemoryWarning: method of your application delegate. (2)、Override the didReceiveMemoryWarning method in your custom UIViewController subclass. (3)、Register to receive the 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 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.

二、内存管理

1、常见的内存问题

There are two main kinds of problem that result from incorrect memory management:

1.1 Freeing or overwriting data that is still in use This causes memory corruption, and typically results in your application crashing, or worse, corrupted user data.

1.2 Not freeing data that is no longer in use causes memory leaks A memory leak is where allocated memory is not freed, even though it is never used again. Leaks cause your application to use ever-increasing amounts of memory, which in turn may result in poor system performance or your application being terminated.

2、Memory Management Policy

2.1 基本的内存管理准则(Basic Memory Management Rules) MRC

2.1.1 You own any object you create

You create an object using a method whose name begins with “alloc”, “new”, “copy”, or “mutableCopy” (for example, alloc, newObject, or mutableCopy).

2.1.2 You can take ownership of an object using retain

A received object is normally guaranteed to remain valid within the method it was received in, and that method may also safely return the object to its invoker. You use retain in two situations: (1) In the implementation of an accessor method or an init method, to take ownership of an object you want to store as a property value; and (2) To prevent an object from being invalidated as a side-effect of some other operation (as explained in Avoid Causing Deallocation of Objects You’re Using).

2.1.3 When you no longer need it, you must relinquish ownership of an object you own

You relinquish ownership of an object by sending it a release message or an autorelease message. In Cocoa terminology, relinquishing ownership of an object is therefore typically referred to as “releasing” an object.

2.1.4 You must not relinquish ownership of an object you do not own

This is just corollary of the previous policy rules, stated explicitly.

When an application terminates, objects may not be sent a dealloc message. Because the process’s memory is automatically cleared on exit, it is more efficient simply to allow the operating system to clean up resources than to invoke all the memory management methods.

2.1 Don’t Use Accessor Methods in Initializer Methods and dealloc

The only places you shouldn’t use accessor methods to set an instance variable are in initializer methods and dealloc. To initialize a counter object with a number object representing zero, you might implement an init method as follows:

2.2 Use Weak References to Avoid Retain Cycles delegate， block

2.3 Avoid Causing Deallocation of Objects You’re Using

2.4 Don’t Use dealloc to Manage Scarce Resources

Each thread in a Cocoa application maintains its own stack of autorelease pool blocks. If you are writing a Foundation-only program or if you detach a thread, you need to create your own autorelease pool block.

If your application or thread is long-lived and potentially generates a lot of autoreleased objects, you should use autorelease pool blocks (like AppKit and UIKit do on the main thread); otherwise, autoreleased objects accumulate and your memory footprint grows. If your detached thread does not make Cocoa calls, you do not need to use an autorelease pool block.