

C#内存管理变化

议题

- C++ 的手动回收机制
- .net的自动回收机制（GC技术）
- 使用C++实现自动回收算法
- 如何规划合理内存
- GC线程与内存管理的惰性

Win32 内存

- 在WINDOWS系统中，32位处理器上的每个进程最多可以使用4GB的内存。这4GB内存称为虚拟地址空间，或虚拟内存。

C++ 的手动回收机制

- 在C++中，在堆上分配的内存必须手动回收。即new和delete操作符是要成对使用的。这种方式给程序员提供了强大而灵活的控制能力，程序员可以控制何时销毁对象，回收内存。这种控制能力，令C++程序员感到自己对程序具有很强的驾驭能力，因此对此功能津津乐道。
- 但是，这种手动回收机制，非常的难以控制，即使经验丰富的C++程序员，也难免疏漏，导致程序中发生内存泄露，轻则导致系统运行越来越慢，重则最终崩溃。这也是手动回收机制所最为人所诟病的地方。

栈与托管堆

- 值类型存放在堆栈中，引用类型存放在托管堆上。堆栈上的内存总是向下填充的，即由高地址空间向低地址空间填充；托管堆上的内存总是向上填充的，即由低地址空间向高地址空间分配。
- C#的托管堆与旧未托管堆（如从C++的堆）的区别在于压缩操作，即只要垃圾收集器释放了托管堆上的对象，就会压缩其它对象，把释放的内存块移动到堆的端部，再次形成一个连续的块，就会使为新对象分配内存空间时的速度非常快，这就是C#中创建新对象比C++中创建新对象快的原因。（C++的堆中就没有这种操作，内存释放后，已释放的内存会和未释放的内存混合在一起，导致连续的未被使用的内存块都非常小，这样的话当创建一个新的对象时，运行库必须搜索堆，才能找到足够大的内存块来存放新对象）

垃圾回收器的基本思想

- 寻找不再使用的对象，将他们从内存中删除，并压实托管堆以释放不在使用的对象所占用的内存。在堆被压实之后，所有的对象引用都将被调整为指向对象新的存储位置。
- 垃圾回收机制的功能：用来管理托管资源和非托管资源所占用的内存分配和释放。
- 垃圾回收器的基本假定：
 - 1. 最近被分配的内存空间的对象最有可能需要被释放。在方法执行时，通常需要为该方法所使用到的对象分配内存空间，搜索最近被分配的对象集合有助于花费最少的工作来释放尽可能多的空闲内存空间。
 - 2. 生命期最长的对象需要释放的可能性最小。在通过几轮的垃圾回收后仍然存在的对象不大可能是那种能够在下一轮回收中被释放的临时对象，搜索这些内存块往往要进行大量的工作，却只能释放很小一部分的内存空间。
 - 3. 同时分配内存的对象通常也会同时使用，将同时分配内存的对象的存储位置彼此相连有助于提高缓存性能，在垃圾回收时也往往是同一批处理或者是遵循后分配，先释放原则。

触发回收的条件

- 当垃圾回收器的指针指向托管堆以外的内存空间时，就需要回收内存中的垃圾了。
- 在这个过程中，垃圾回收器首先假设在托管堆中任何的对象都需要被回收。然后他在托管堆中寻找被根对象引用的对象（根对象就是全局，静态或处于活动中的局部变量连同寄存器指向的对象），找到后将他们加入一个有效对象的列表中，并在已搜索过的对象中寻找是否有对象被新加入的有效对象引用。直到垃圾回收器检查完任何的对象后，就有一份根对象和根对象直接或间接引用了的对象的列表，而其他没有在表中的对象就被从内存中回收。

自动垃圾回收机制

- 自动垃圾回收无需程序员的干预，自动回收废弃对象，释放内存。它有效的避免了内存泄露问题，但是，相应的也带来了新的问题：
- 1、废弃对象的回收时间是不确定的，即你无法确定系统在何时会执行垃圾回收。每个垃圾收集器都会按照一定的算法执行垃圾收集，但它何时执行，每次执行回收哪些对象是不确定的。
- 2、它使程序员丧失了部分的控制能力，相应的在系统运行过程中，在性能上也会有一定的损失。

.net的自动回收机制

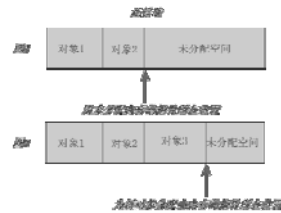
- C#中对象的销毁和回收，在实现方式上比较灵活。它既允许立刻销毁对象，回收内存，也允许通过垃圾回收器自动回收。能够做到这点，是因为C#提供了两种对象的销毁方式：
- 1、类支持IDisposable接口，在每次使用完对象后，立即通过手工Dispose（）方法，或者利用Using块隐式的强制调用Dispose（）方法，实现在对象的立即销毁。在Dispose（）方法中，将调用Dispose（）方法的带一个布尔参数的重载，在该重载函数中，手工编写代码释放对象所打开的所有托管和非托管资源。
- 2、通过垃圾回收器自动回收，在垃圾回收器销毁对象的时候，调用类的析构函数，释放对象打开的非托管资源，销毁对象，回收内存。由于析构函数仅在使用垃圾回收器回收的时候调用，因此析构函数内仅需要释放非托管资源即可，对象所打开占用的托管资源对象，会被垃圾回收器自动销毁。析构函数对非托管资源的释放，也是通过调用Dispose（）方法的带一个布尔参数的重载来实现。

.NET 框架的垃圾回收器

- .NET 框架的垃圾回收器被称为分代的垃圾回收器(Generational Garbage Collector)，也就是说被分配的内存分为三个类别（生存期等级），或者说分为三代，即0，1，2三代，对应的托管堆的初始化大小分别是256K，2M和10M。
- 当垃圾回收器在发现改变托管堆的大小能够提高性能的话，会改变托管堆的大小。例如：当应用程序实例化了一些占用内存较少的对象时，而且这些对象所占用的资源能被很快回收的话，0代托管堆的大小会变为128K。相反，如果垃圾回收器发现所占用的资源不能被很快回收的话，会增加托管堆的大小，这时就是512k了。（注意：前提是：当垃圾回收器在发现改变托管堆的大小能够提高性能的话，才会改变托管堆的大小。）
- 最近被分配的内存空间的对象被放置在第0代，一般存储于二级缓存中，所以能对第0代中的对象实现快速存取。经过一轮垃圾回收后，仍然保留在第0代中的对象则被移进第1代中，再经过一轮垃圾回收后，仍然保留再第1代中的对象则被移进第2代中，第2代中包含了生存期较长的对象，这些对象至少经过了两轮回收。
- 现在来考虑一个问题，为什么要将本轮中没有被回收掉的对象移到下一代中呢？就是为了把存取速度较快的空间给置空，分配给最近实例化的对象，为了提高效率用来实现对象的快速读取的。

托管堆上的内存分配

- 当程序请求为某对象分配空间时，托管堆上的指针会指向下一个最近的可用的内存空间，如下图所示：



- 由此图我们可以清晰的知道托管堆的内存分配是线性分配的，
- 当然，这种分配方式的效率也是最高的。而普通的堆对于内存分配是基于内存块大小的，两个同时声明的对象在普通堆上被分配的位置可能相隔较远，于是降低了缓存的性能。所以，在一个不需要太多垃圾回收的应用程序中，托管堆的表现会优于传统的堆。

托管堆连续分配空间

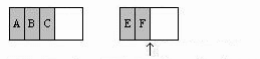
- 在应用程序初始化的之前，所有等级的托管堆都是空的。当对象被初始化的时候，他们会按照初始化的先后顺序被放入等级为0的托管堆中。在托管堆中对象的存放是连续的，这样使得托管堆存取对象的速度很快，因为托管对不必对内存进行搜索。垃圾回收器中保存了一个指针指向托管堆中最后一个对象之后的内存空间。图一中显示了一个包含四个对象的0等级的托管堆。



图一 包含四个对象的托管堆

带间转移

- 当0等级托管堆被对象填满后，例如候程序初始化了新的对象，使0等级托管堆的大小超过了256K，垃圾回收器会检查托管堆中的所有对象，看是否有对象可以回收。当开始回收操作时，如前面提到的，垃圾回收器会找出根节点和根节点直接或间接引用了的对象，然后将这些对象转移到1等级托管堆中，并将0等级托管堆的指针移到最开始的位置以清除所有的对象。同时垃圾回收器会压缩1等级托管堆以保证所有对象之间没有内存空隙。当1等级托管堆满了之后，会将对象转移到2等级的托管堆。
- 例如在图一之后，垃圾回收器开始回收对象，假定D对象将被回收，同时程序创建了E和F对象。这时候托管堆中的对象如图二所示。



图二 回收对象后的0等级和1等级托管堆

大对象区

- 然后程序创建了新的对象G和H，再一次触发了垃圾回收器。对象E将被回收。这时候托管堆中的对象如图三所示。



- 生存期垃圾回收器的原则也有例外的情况。当对象的大小超过84K时，对象会被放入"大对象区"。大对象区中的对象不会被垃圾回收器回收，也不会被压缩。这样做是为了强制垃圾回收器只能回收小对象以提高程序的性能。

对象内存管理函数

- C#程序为一个对象分配内存时，托管堆几乎可以立即返回新对象所需的内存，托管堆之所以能有这样高效的内存分配性能是由于托管堆较为简单的数据结构。托管堆类似于简单的字节数组，有一个指向第一个可用内存空间的指针。
- 在某块被某对象所请求时，上述指针值就会返回给调用函数，而指针会重新调整至指向下一个可用的内存空间。分配一个托管内存块只比递增一个指针的值稍微复杂一点。这也是托管堆所优化的性能之一。在一个不需太多垃圾回收的应用程序中，托管堆的表现会优于传统的堆。

回收机制

- 由于这个线性的内存分配方法的存在，在C#应用程序中同时分配的对象在托管堆上通常会被分配成彼此相邻。此安排和传统的堆内存分配完全不同，传统的堆内存分配是基于内存块大小的。例如，两个同时分配的对象在堆上的位置可能相距很远，从而降低了缓存的性能。因此虽然内存分配很快，但在一些比较重要的程序中，第0代中的可用内存很有可能会彻底被消耗光。
- 记住，第0代小到可以装进L2缓冲区，并且没有被使用的内存不会被自动释放。当第0代中没有可以分配的有效内存时，就会在第0代中触发一轮垃圾回收，在这轮垃圾回收中将删除所有不再被引用的对象，并将当前正在使用中的对象移至第1代。针对第0代的垃圾回收是最常见的回收类型，而且速度很快。
- 在第0代的垃圾内存回收不能有效的请求到充足的内存时，就启动第1代的垃圾内存回收。第2代的垃圾内存回收要作为最后一种手段而使用，当且仅当第1代和第0代的垃圾内存回收不能被提供足够内存时进行。如果各代都进行了垃圾回收后仍没有可用的内存，就会引发一个OutOfMemoryException异常。

垃圾回收器使用finalize()方法

- 当对象被加入到托管堆中时，假如他实现了finalize（）方法，垃圾回收器会在他的终结列表（finalization list）中加入一个指向该对象的指针。当该对象被回收时，垃圾回收器会检查终结列表，看是否需要调用对象的finalize（）方法。假如有的话，垃圾回收器将指向该对象的指针加入一个完成器队列中，该完成器队列保存了那些准备调用finalize（）方法的对象。到了这一步对象还不是真正的垃圾对象。因此垃圾回收器还没有把他们从托管堆中回收。
- 当对象准备被终结时，另一个垃圾回收器线程会调用在完成器队列中每个对象的finalize（）方法。当调用完成后，线程将指针从完成器队列中移出，这样垃圾回收器就知道在下次回收对象时能够清除被终结的对象了。从上面能够看到垃圾回收机制带来的很大一部分额外工作就是调用finalize（）方法，因此在实际编程中研发人员应该避免在类中实现finalize（）方法。
- 对于finalize（）方法的另一个问题是研发人员不知道什么时候他将被调用。他不像c++中的析构函数在删除一个对象时被调用。为了解决这个问题，在.net中提供了一个接口idisposable。

终结器[Finalize() 方法]

- `Protected void Finalize() { Base.Finalize(); }`
- 类提供一个终结器以在对象被销毁时执行，值得注意的是：当一个对象实现了Finalize方法，垃圾回收器会在它的终结列表（Finalization List）中加入一个指向该对象的指针（终结列表中包含的是等待终结的对象）。
- 等到此轮垃圾回收开始时，垃圾回收器会将该对象的引用置入终结列表，标识此对象为等待终结的对象，虽然此对象现在仍然存在，但是在应用程序中已经引用不到该对象了。垃圾回收器在此轮使用一个专用的线程，使终结列表中的对象执行终结器，执行完终结器的对象会被此对象标记为不再需要终结的对象，并从终结列表中除去。
- 终结完成后，对象所占用的资源将在下一轮垃圾回收中被回收。
- 终结顺序是不确定的，但是当某个对象被终结时，由此对象产生都应该已经被终结，大家肯定在想这是为什么吧？

终结器[Finalize() 方法]

- 另外值得一提的是finalize () 方法应该在较短的时间内完成，这是因为垃圾回收器给finalize () 方法限定了一个时间，如果finalize () 方法在规定时间内还没有完成，垃圾回收器会终止运行finalize () 方法的线程。在下面这些情况下程序会调用对象的finalize () 方法：
 - o等级垃圾回收器已满
 - 程序调用了执行垃圾回收的方法
 - 公共语言运行库正在卸载一个应用程序域
 - 公共语言运行库正在被卸载

对比C++与C#析构函数

- 与C++的析构函数相比，C#析构函数的问题是它的不确定性。在删除C++对象时，其析构函数会立即执行。但由于垃圾收集器的工作方式，无法确定C#中的析构函数何时执行。所以不能在C#的析构函数中放置需要在某一时刻运行的代码，也不应使用能以任意顺序对不同类实例调用的析构函数。
- 另一个问题是C#析构函数的执行会延迟对象最终从内存中删除的时间。没有析构函数的对象会在垃圾收集器的一次处理中从内存中删除，但有析构函数的对象需要两次处理才能删除：第一次调用析构函数时，没有删除对象，第二次调用才真正删除对象。另外，运行库使用一个线程来执行所有对象的Finalize()方法。如果频繁使用析构函数，而且使用它们执行长时间的清理任务，对性能的影响就会非常显著。

控制垃圾回收器

- 在.Net框架中提供了很多方法使开发人员能够直接控制垃圾回收器的行为。通过使用GC.Collect () 或GC.Collect (int GenerationNumber) 开发人员可以强制垃圾回收器对所有等级的托管堆进行回收操作。
- 在大多数的情况下开发人员不需要干涉垃圾回收器的行为, 但是有些情况下, 例如当程序进行了非常复杂的操作后希望确认内存中的垃圾对象已经被回收, 就可以使用上面的方法。
- 另一个方法是GC.WaitForPendingFinalizers () , 它可以挂起当前线程, 直到处理完成器队列的线程清空该队列为止。

.net手动释放机制

- 使用垃圾回收器最好的方法就是跟踪程序中定义的对象, 在程序不需要它们的时候手动释放它们。例如程序中的一个对象中有一个字符串属性, 该属性会占用一定的内存空间。
- 当该属性不再被使用时, 开发人员可以在程序中将其设定为null, 这样垃圾回收器就可以回收该字符串占用的空间。另外, 如果开发人员确定不再使用某个对象时, 需要同时确定没有其它对象引用该对象, 否则垃圾回收器不会回收该对象。

使用C++实现自动回收算法

- **引用计数 (Reference Counting)**

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为0的对象。此算法最致命的是无法处理循环引用的问题。

- **标记-清除 (Mark-Sweep)**

此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

- **复制 (Copying)**

此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不过出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

如何规划合理内存

- 采用动静结合的规划内存的方法
- 宏观内存设计
- 微观内存设计