自动内存管理是公共语言运行时在托管执行过程中提供的服务之一。 公共语言运行时的垃圾回收器为应用程序管理内存的分配和释放。 对开发人员而言，这就意味着在开发托管应用程序时不必编写执行内存管理任务的代码。 自动内存管理可解决常见问题，例如，忘记释放对象并导致内存泄漏，或尝试访问已释放对象的内存。 本节描述垃圾回收器如何分配和释放内存。

分配内存

初始化新进程时，运行时会为进程保留一个连续的地址空间区域。 这个保留的地址空间被称为托管堆（即是初始化新进程的时候会有一个托管堆和其对应）。 托管堆维护着一个指针，用它指向将在堆中分配的下一个对象的地址。 最初，该指针设置为指向托管堆的基址。 托管堆上包含了所有引用类型。 应用程序创建第一个引用类型时，将为托管堆的基址中的类型分配内存。 应用程序创建下一个对象时，垃圾回收器在紧接第一个对象后面的地址空间内为它分配内存。 只要地址空间可用，垃圾回收器就会继续以这种方式为新对象分配空间。

从托管堆中分配内存要比非托管内存分配速度快。 由于运行时通过为指针添加值来为对象分配内存，所以这几乎和从堆栈中分配内存一样快。 另外，由于连续分配的新对象在托管堆中是连续存储，所以应用程序可以快速访问这些对象。

释放内存

垃圾回收器的优化引擎根据所执行的分配决定执行回收的最佳时间。 垃圾回收器在执行回收时，会释放应用程序不再使用的对象的内存。 它通过检查应用程序的根来确定不再使用的对象。 每个应用程序都有一组根。 每个根或者引用托管堆中的对象，或者设置为空。 应用程序的根包含线程堆栈上的静态字段、局部变量和参数以及 CPU 寄存器。 垃圾回收器可以访问由实时 (JIT) 编译器和运行时维护的活动根的列表。 垃圾回收器对照此列表检查应用程序的根，并在此过程中创建一个图表，在其中包含所有可从这些根中访问的对象。

不在该图表中的对象将无法从应用程序的根中访问。 **垃圾回收器会考虑无法访问的对象垃圾，并释放为它们分配的内存。** 在回收中，垃圾回收器检查托管堆，查找无法访问对象所占据的地址空间块。 发现无法访问的对象时，它就使用内存复制功能来压缩内存中可以访问的对象，释放分配给不可访问对象的地址空间块。 在压缩了可访问对象的内存后，垃圾回收器就会做出必要的指针更正，以便应用程序的根指向新地址中的对象。 它还将托管堆指针定位至最后一个可访问对象之后。 请注意，**只有在回收发现大量的无法访问的对象时，才会压缩内存。** 如果托管堆中的所有对象均未被回收，则不需要压缩内存。

为了改进性能，运行时为单独堆中的大型对象分配内存。 垃圾回收器会自动释放大型对象的内存。 但是，为了避免移动内存中的大型对象，不会压缩此内存。

级别和性能

为优化垃圾回收器的性能，将托管堆分为三代：第 0 代、第 1 代和第 2 代。 运行时的垃圾回收算法基于以下几个普遍原理，这些垃圾回收方案的原理已在计算机软件业通过实验得到了证实。 首先，压缩托管堆的一部分内存要比压缩整个托管堆速度快。 其次，较新的对象生存期较短，而较旧的对象生存期则较长。 最后，较新的对象趋向于相互关联，并且大致同时由应用程序访问。

运行时的垃圾回收器将新对象存储在第 0 级中。 在应用程序生存期的早期创建的对象如果未被回收，则被升级并存储在第 1 级和第 2 级中。 本主题中稍后介绍了对象升级过程。 因为压缩托管堆的一部分

要比压缩整个托管堆速度快，所以此方案允许垃圾回收器在每次执行回收时释放特定级别的内存，而不是整个托管堆的内存。

实际上，垃圾回收器在第 0 级托管堆已满时执行回收。 如果应用程序在第 0 级托管堆已满时尝试新建对象，垃圾回收器将会发现第 0 级托管堆中没有可分配给该对象的剩余地址空间。 垃圾回收器执行回收，尝试为对象释放第 0 级托管堆中的地址空间。 垃圾回收器从检查第 0 级托管堆中的对象（而不是托管堆中的所有对象）开始执行回收。 这是最有效的途径，因为新对象的生存期往往较短，并且期望在执行回收时，应用程序不再使用第 0 级托管堆中的许多对象。 另外，单独回收第 0 级托管堆通常可以回收足够的内存，这样，应用程序便可以继续创建新对象。

垃圾回收器执行第 0 级托管堆的回收后，会压缩可访问对象的内存，如本主题前面的释放内存中所述。 然后，垃圾回收器升级这些对象，并考虑第 1 级托管堆的这一部分。 因为未被回收的对象往往具有较长的生存期，所以将它们升级至更高的级别很有意义。 因此，垃圾回收器在每次执行第 0 级托管堆的回收时，不必重新检查第 1 级和第 2 级托管堆中的对象。

在执行第 0 级托管堆的首次回收并把可访问的对象升级至第 1 级托管堆后，垃圾回收器将考虑第 0 级托管堆的其余部分。 它将继续为第 0 级托管堆中的新对象分配内存，直至第 0 级托管堆已满并需执行另一回收为止。 这时，垃圾回收器的优化引擎会决定是否需要检查较旧的级别中的对象。 例如，如果第 0 级托管堆的回收没有回收足够的内存，不能使应用程序成功完成创建新对象的尝试，垃圾回收器就会先执行第 1 级托管堆的回收，然后再执行第 2 级托管堆的回收。 如果这样仍不能回收足够的内存，垃圾回收器将执行第 2、1 和 0 级托管堆的回收。 每次回收后，垃圾回收器都会压缩第 0 级托管堆中的可访问对象并将它们升级至第 1 级托管堆。 第 1 级托管堆中未被回收的对象将会升级至第 2 级托管堆。 由于垃圾回收器只支持三个级别，因此第 2 级托管堆中未被回收的对象会继续保留在第 2 级托管堆中，直到在将来的回收中确定它们为无法访问为止。

为非托管资源释放内存

对于应用程序创建的大多数对象，可以依赖垃圾回收器自动执行必要的内存管理任务。 但是，**非托管资源需要显式清除。 最常用的非托管资源类型是包装操作系统资源的对象，例如，文件句柄、窗口句柄或网络连接。** 虽然垃圾回收器可以跟踪封装非托管资源的托管对象的生存期，但却无法具体了解如何清理资源。 **创建封装非托管资源的对象时，建议在公共 Dispose 方法中提供必要的代码以清理非托管资源。 通过提供 Dispose 方法，对象的用户可以在使用完对象后显式释放其内存。** 使用封装非托管资源的对象时，应该了解 Dispose 并在必要时调用它。 有关清理非托管资源的详细信息和实现 Dispose 的设计模式示例，请参见 垃圾回收