垃圾回收的基本知识

2018/03/08 • 作者

本文内容

在公共语言运行时 (CLR) 中，垃圾回收器用作自动内存管理器。 它提供如下优点：

- 使你可以在开发应用程序时不必释放内存。
- 有效分配托管堆上的对象。
- 回收不再使用的对象，清除它们的内存，并保留内存以用于将来分配。 托管对象会自动获取干净的内容来开始，因此，它们的构造函数不必对每个数据字段进行初始化。
- 通过确保对象不能使用另一个对象的内容来提供内存安全。

本主题介绍垃圾回收的核心概念。

内存基础知识

下面的列表总结了重要的 CLR 内存概念。

- 每个进程都有其自己单独的虚拟地址空间。 同一台计算机上的所有进程共享相同的物理内存，如果有页文件，则也共享页文件。
- 默认情况下，32 位计算机上的每个进程都具有 2 GB 的用户模式虚拟地址空间。
- 作为一名应用程序开发人员，你只能使用虚拟地址空间，请勿直接操控物理内存。 垃圾回收器为你分配和释放托管堆上的虚拟内存。

如果你编写的是本机代码，请使用 Win32 函数处理虚拟地址空间。 这些函数为你分配和释放本机堆上的虚拟内存。

- 虚拟内存有三种状态：
  - 可用。 该内存块没有引用关系，可用于分配。
  - 保留。 内存块可供你使用，并且不能用于任何其他分配请求。 但是，在该内存块提交之前，你无法将数据存储到其中。
  - 提交。 内存块已指派给物理存储。
- 可能会存在虚拟地址空间碎片。 就是说地址空间中存在一些被称为孔的可用块。 当请求虚拟内存分配时，虚拟内存管理器必须找到满足该分配请求的足够大的单个可用块。 即使有 2GB 可用空间，2GB 分配请求也会失败，除非所有这些可用空间都位于一个地址块中。
- 如果用完保留的虚拟地址空间或提交的物理空间，则可能会用尽内存。

即使在物理内存压力（即物理内存的需求）较低的情况下也会使用页文件。 首次出现物理内存压力较高的情况时，操作系统必须在物理内存中腾出空间来存储数据，并将物理内存中的部分数据备份到页文件中。 该数据只会在需要时进行分页，所以在物理内存压力非常低的情况下也可能会进行分页。

返回页首

垃圾回收的条件

当满足以下条件之一时将发生垃圾回收：

- 系统具有低的物理内存。 这是通过 OS 的内存不足通知或主机指示的内存不足检测出来。
- 由托管堆上已分配的对象使用的内存超出了可接受的阈值。 随着进程的运行，此阈值会不断地进行调整。
- 调用 GC.Collect 方法。 几乎在所有情况下，你都不必调用此方法，因为垃圾回收器会持续运行。 此方法主要用于特殊情况和测试。

返回页首

托管堆

在垃圾回收器由 CLR 初始化之后，它会分配一段内存用于存储和管理对象。 此内存称为托管堆（与操作系统中的本机堆相对）。

每个托管进程都有一个托管堆。 进程中的所有线程都在同一堆上分配对象记忆。

若要保留内存，垃圾回收器将调用 Win32 VirtualAlloc 函数，并且每次会为托管应用程序保留一个内存段。 垃圾回收器还会根据需要保留段，并通过调用 Win32 VirtualFree 函数将段释放回操作系统（在清除所有对象的段之后）。

> ⓘ **重要**
>
> 垃圾回收器分配的段大小特定于实现，并且随时可能更改（包括定期更新）。 应用程序不应假设特定段的大小或依赖于此大小，也不应尝试配置段分配可用的内存量。

堆上分配的对象越少，垃圾回收器必须执行的工作就越少。 分配对象时，请勿使用超出你需求的舍入值，例如在仅需要 15 个字节的情况下分配了 32 个字节的数组。

当触发垃圾回收时，垃圾回收器将回收由死对象占用的内存。 回收进程会对活动对象进行压缩，以便将它们一起移动，并移除死空间，从而使堆更小一些。 这将确保一起分配的对象全都位于托管堆上，从而保留它们的局部性。

垃圾回收的侵入性（频率和持续时间）是由分配的数量和托管堆上保留的内存数量决定的。

此堆可视为两个堆的累计：大对象堆和小对象堆。

大对象堆包含大小为 85,000 个字节和更多字节的大型对象。 大对象堆上的对象通常是数组。 非常大的实例对象是很少见的。

返回页首

代数

堆按代进行组织，因此它可以处理长生存期的对象和短生存期的对象。 垃圾回收主要在回收通常只占用一小部分堆的短生存期对象时发生。 堆上的对象有三代：

- **第 0 代。** 这是最年轻的代，其中包含短生存期对象。 短生存期对象的一个示例是临时变量。 垃圾回收最常发生在此代中。

  新分配的对象构成新一代的对象并且为隐式的第 0 代回收，除非它们是大对象，在这种情况下，它们将进入第 2 代回收中的大对象堆。

  大多数对象通过第 0 代中的垃圾回收进行回收，不会保留到下一代。

- **第 1 代。** 这一代包含短生存期对象并用作短生存期对象和长生存期对象之间的缓冲区。

- **第 2 代。** 这一代包含长生存期对象。 长生存期对象的一个示例是服务器应用程序中的一个包含在进程期间处于活动状态的静态数据的对象。

当条件得到满足时，垃圾回收将在特定代上发生。 回收某个代意味着回收此代中的对象及其所有更年轻的代。 第 2 代垃圾回收也称为完整垃圾回收，因为它回收所有代上的所有对象（即，托管堆中的所有对象）。

幸存和提升

垃圾回收中未回收的对象也称为幸存者，并会被提升到下一代。 在第 0 代垃圾回收中幸存的对象将被提升到第 1 代；在第 1 代垃圾回收中幸存的对象将被提升到第 2 代；而在第 2 代垃圾回收中幸存的对象将仍为第 2 代。

当垃圾回收器检测到某个代中的幸存率很高时，它会增加该代的分配阈值，因此下一次回收将会获取一个非常大的回收内存。 CLR 会在以下两个优先级别之前进行平衡：不允许应用程序的工作集获取太大内存以及不允许垃圾回收花费太多时间。

暂时代和暂时段

因为第 0 代和第 1 代中的对象的生存期较短，因此，这些代被称为暂时代。

暂时代必须在称为暂时段的内存段中进行分配。 垃圾回收器获取的每个新段将成为新的暂时段，并包含在第 0 代垃圾回收中幸存的对象。 旧的暂时段将成为新的第 2 代段。

根据系统为 32 位还是 64 位以及它正在哪种类型的垃圾回收器上运行，暂时段的大小发生相应变化。 下表列出了默认值。

	32 位	64 位
工作站 GC	16 MB	256 MB
服务器 GC	64 MB	4 GB
服务器 GC（具有 4 个以上的逻辑 CPU）	32 MB	2 GB
服务器 GC（具有 8 个以上的逻辑 CPU）	16 MB	1 GB

暂时段可以包含第 2 代对象。 第 2 代对象可使用多个段（在内存允许的情况下进程所需的任意数量）。

从暂时垃圾回收中释放的内存量限制为暂时段的大小。 释放的内存量与死对象占用的空间成比例。

返回页首

垃圾回收过程中发生的情况

垃圾回收分为以下几个阶段：

- 标记阶段，找到并创建所有活动对象的列表。
- 重定位阶段，用于更新对将要压缩的对象的引用。
- 压缩阶段，用于回收由死对象占用的空间，并压缩幸存的对象。 压缩阶段将垃圾回收中幸存下来的对象移至段中时间较早的一端。

  因为第 2 代回收可以占用多个段，所以可以将已提升到第 2 代中的对象移动到时间较早的段中。 可以将第 1 代幸存者和第 2 代幸存者都移动到不同的段，因为它们已被提升到第 2 代。

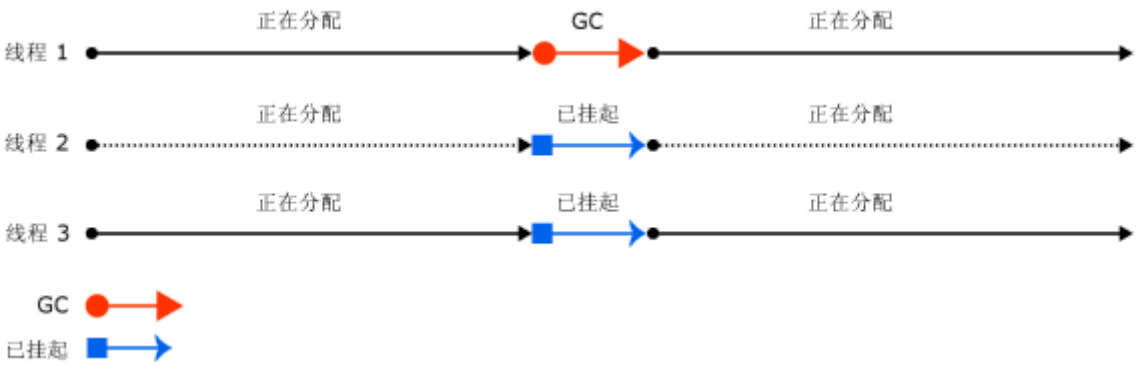
  通常，由于复制大型对象会造成性能代偿，因此不会压缩大型对象堆。 但是，从 .NET Framework 4.5.1开始，你可以使用 GCSettings.LargeObjectHeapCompactionMode 属性按需压缩大对象堆。

垃圾回收器使用以下信息来确定对象是否为活动对象：

- **堆栈根。** 由实时 (JIT) 编译器和堆栈查看器提供的堆栈变量。 请注意，JIT 优化可以延长或缩短报告给垃圾回收器的堆栈变量内的代码的区域。
- **垃圾回收句柄。** 指向托管对象且可由用户代码或公共语言运行时分配的句柄。
- **静态数据。** 应用程序域中可能引用其他对象的静态对象。 每个应用程序域都会跟踪其静态对象。

在垃圾回收启动之前，除了触发垃圾回收的线程以外的所有托管线程均会挂起。

下图演示了触发垃圾回收并导致其他线程挂起的线程。



触发垃圾回收的线程

返回页首

操作非托管资源

如果你的托管对象使用非托管对象的本机文件句柄来引用非托管对象，则必须显式释放非托管对象，因为垃圾回收器仅跟踪托管堆上的内存。

托管对象的用户可能不会释放由该对象使用的本机资源。 为了执行清理，可以使托管对象成为可终结的。 终结由不再使用对象时执行的清理操作组成。 当托管对象不活动时，它将执行在其终结器方法中指定的清理操作。

当发现某个可终结对象处于不活动状态时，则会将其终结器放入队列中，以便执行其清理操作，但要将该对象自身提升到下一代。 因此，你必须等待该代上发生下一次垃圾回收（并不一定是下一次垃圾回收），以确定对象是否已收回。

返回页首

工作站和服务器垃圾回收

垃圾回收器可自行优化并且适用于多种方案。 你可使用配置文件设置来基于工作负荷的特征设置垃圾回收的类型。 CLR 提供了以下类型的垃圾回收：

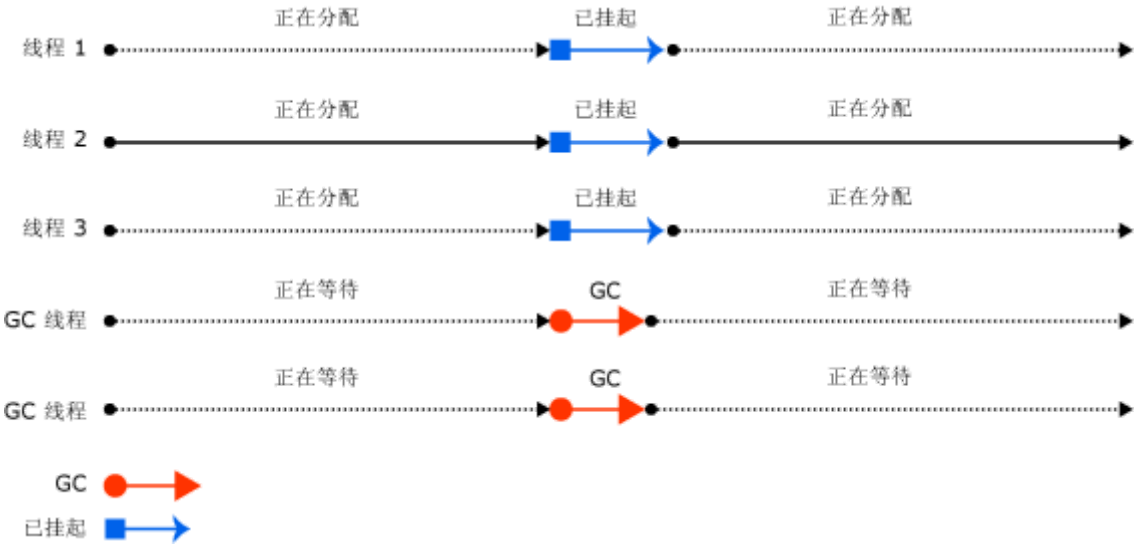
- 工作站垃圾回收，用于所有客户端工作站和独立 PC。 这是运行时配置架构中 <gcServer> 元素的默认设置。

  工作站垃圾回收既可以是并发的，也可以是非并发的。 并发垃圾回收使托管线程能够在垃圾回收期间继续操作。

  从 .NET Framework 4开始，后台垃圾回收取代了并发垃圾回收。

- 服务器垃圾回收，用于需要高吞吐量和可伸缩性的服务器应用程序。 服务器垃圾回收既可以是非并发也可以是背景。

下图演示了服务器上执行垃圾回收的专用线程。



服务器垃圾回收

配置垃圾回收

可以使用运行时配置架构的 <gcServer> 元素，指定要 CLR 执行的垃圾回收类型。 在将此元素的 `enabled` 特性设置为 `false` （默认值）时，CLR 将执行工作站垃圾回收。 在将 `enabled` 特性设置为 `true` 时，CLR 将执行服务器垃圾回收。

并发垃圾回收是使用运行时配置架构的 <gcConcurrent> 元素进行指定。 默认设置为 `enabled`。 此设置可控制并发和后台垃圾回收。

还可以使用非托管承载接口来指定服务器垃圾回收。 请注意，如果你的应用程序承载在这些环境之一中，则 ASP.NET 和 SQL Server 将自动启用服务器垃圾回收。

工作站和服务器垃圾回收比较

以下是工作站垃圾回收的线程处理和性能注意事项：

- 回收发生在触发垃圾回收的用户线程上，并保留相同优先级。 因为用户线程通常以普通优先级运行，所以垃圾回收器（在普通优先级线程上运行）必须与其他线程竞争 CPU 时间。

  不会挂起运行本机代码的线程。

- 工作站垃圾回收始终用于只有一个处理器的计算机，无论 <gcServer> 设置如何。 如果你指定服务器垃圾回收，则 CLR 会使用工作站垃圾回收，并禁用并发。

以下是服务器垃圾回收的线程处理和性能注意事项：

- 回收发生在以 `THREAD_PRIORITY_HIGHEST` 优先级运行的多个专用线程上。
- 为每个 CPU 提供一个用于执行垃圾回收的一个堆和专用线程，并将同时回收这些堆。 每个堆都包含一个小对象堆和一个大对象堆，并且所有的堆都可由用户代码访问。 不同堆上的对象可以相互引用。
- 因为多个垃圾回收线程一起工作，所以对于相同大小的堆，服务器垃圾回收比工作站垃圾回收更快一些。
- 服务器垃圾回收通常具有更大的段。 但是请注意，这是通常情况：段大小特定于实现且可能更改。 调整应用程序时，不应假设垃圾回收器分配的段大小。
- 服务器垃圾回收会占用大量资源。 例如，如果在一台具有 4 个处理器的计算机上运行了 12 个进程，则在它们都使用服务器垃圾回收的情况下，将有 48 个专用垃圾回收线程。 在高内存加载的情况下，如果所有进程开始执行垃圾回收，则垃圾回收器将要计划 48 个线程。

如果运行应用程序的数百个实例，请考虑使用工作站垃圾回收并禁用并发垃圾回收。 这可以减少上下文切换，从而提高性能。

返回页首

并行垃圾回收

在工作站或服务器垃圾回收中，你可以启用并发垃圾回收，以便在大多数回收期间，让各线程与执行垃圾回收的专用线程并发运行。 此选项只影响第 2 代中的垃圾回收；第 0 代和第 1 代中的垃圾回收始终是非并发的，因为它们完成的速度非常快。

并发垃圾回收通过最大程度地减少因回收引起的暂停，使交互应用程序能够更快地响应。 在运行并发垃圾回收线程的大多数时间，托管线程可以继续运行。 这可以使得在发生垃圾回收时的暂停时间更短。

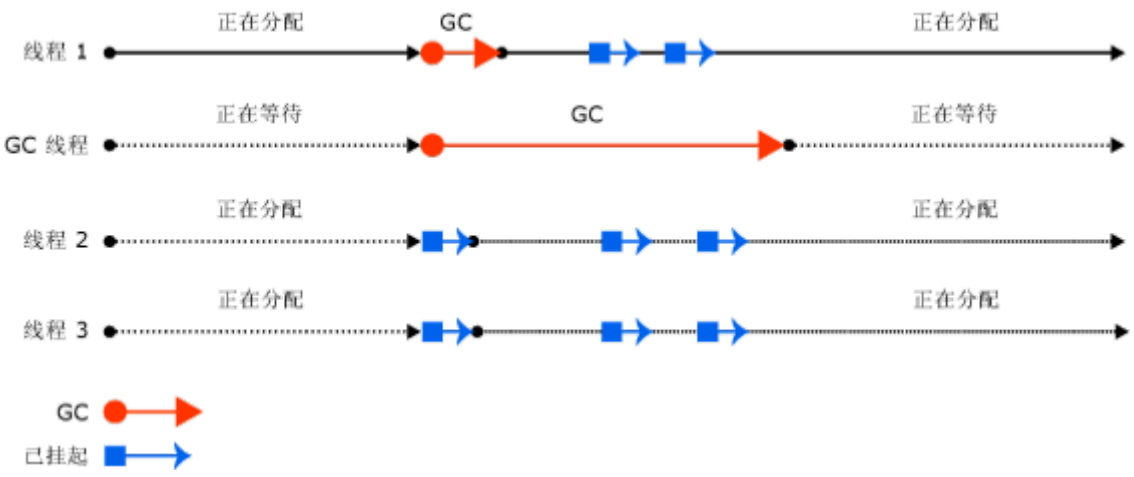
若要在运行多个进程时提高性能，请禁用并发垃圾回收。 为此，可以将 <gcConcurrent> 元素添加到应用的配置文件，并将其 `enabled` 属性的值设置为 `"false"`。

并发垃圾回收在一个专用线程上执行。 默认情况下，CLR 将运行工作站垃圾回收并启用并发垃圾回收。 对于单处理器计算机和多处理器计算机都是如此。

你在并发垃圾回收期间在堆上为小对象分配空间的能力将受到在并发垃圾回收启动时暂时段上保留的对象的限制。 一旦到达暂时段的末尾，将必须等待并发垃圾回收完成，同时将挂起需要执行小对象分配的托管线程。

并发垃圾回收具有一个稍微大点的工作集（与非并发垃圾回收相比），这是因为你可以在并发回收期间分配对象。 但是，这会影响性能，原因是分配的对象将会成为你的工作集的一部分。 实质上，并发垃圾回收会牺牲一些 CPU 和内存来换取更短的暂停。

下图演示了在单独的专用线程上执行的并发垃圾回收。



并行垃圾回收

返回页首

后台工作站垃圾回收

在后台垃圾回收中，在进行第 2 代回收的过程中，将会根据需要收集暂时代（第 0 代和第 1 代）。 后台垃圾回收无法设置；它会自动运行并启用并发垃圾回收。 后台垃圾回收是对并发垃圾回收的替代。 与并发垃圾回收一样，后台垃圾回收是在一个专用线程上执行的并且只适用于第 2 代回收。

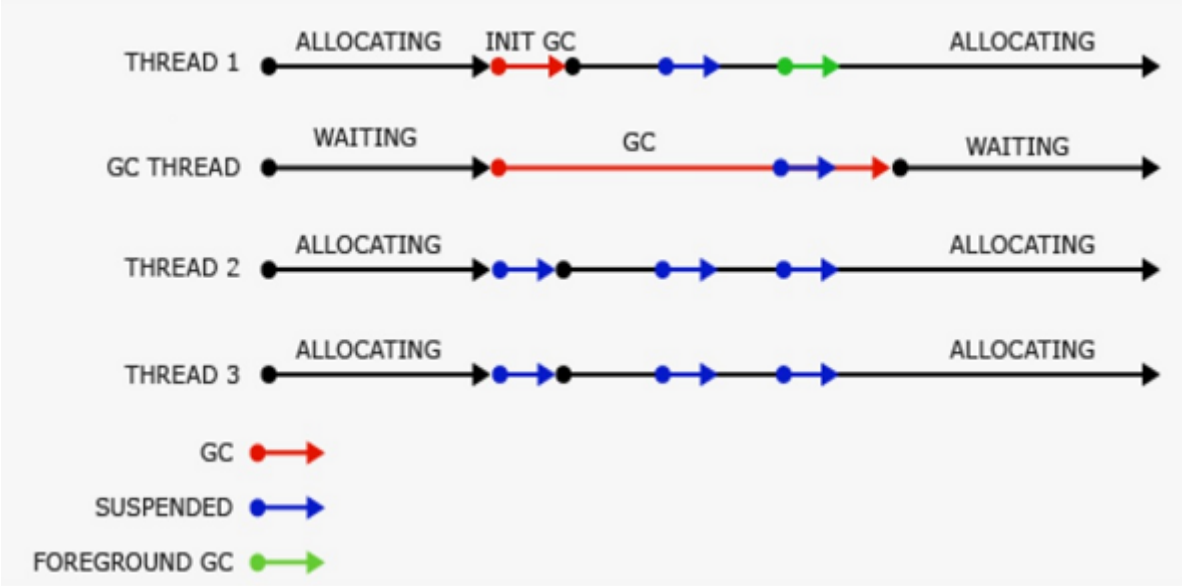
> **备注**
>
> 后台垃圾回收只在 .NET Framework 4 及更高版本中可用。 在 .NET Framework 4 中，仅支持工作站垃圾回收。 从 .NET Framework 4.5 开始，后台垃圾回收可用于工作站和服务器垃圾回收。

后台垃圾回收期间对暂时代的回收称为前台垃圾回收。 发生前台垃圾回收时，所有托管线程都将被挂起。

当后台垃圾回收正在进行并且你已在第 0 代中分配了足够的对象时，CLR 将执行第 0 代或第 1 代前台垃圾回收。 专用的后台垃圾回收线程将在常见的安全点上进行检查以确定是否存在对前台垃圾回收的请求。 如果存在，则后台回收将挂起自身以便前台垃圾回收可以发生。 在前台垃圾回收完成之后，专用的后台垃圾回收线程和用户线程将继续。

后台垃圾回收可以消除并发垃圾回收所带来的分配限制，因为在后台垃圾回收期间，可发生暂时垃圾回收。 这意味着，后台垃圾回收可以移除暂时代中的死对象，而且还可以在第 1 代垃圾回收期间根据需要展开堆。

下图显示对工作站上的独立专用线程执行的后台垃圾回收：

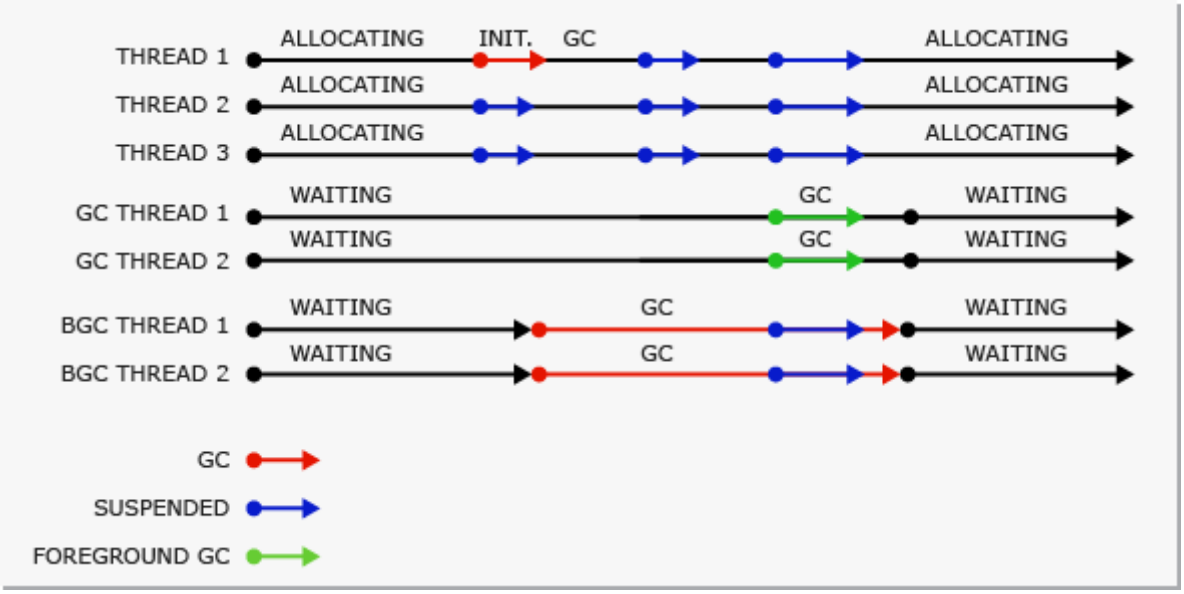


返回页首

后台服务器垃圾回收

从 .NET Framework 4.5 开始，后台服务器垃圾回收是服务器垃圾回收的默认模式。 若要选择此模式，请在运行时配置架构中将 <gcServer> 元素的 `enabled` 属性设置为 `true`。 此模式与后台工作站垃圾回收（如上一章节所描述）具有类似功能，但有一些不同之处。 后台工作区域垃圾回收使用一个专用的后台垃圾回收线程，而后台服务器垃圾回收使用多个线程，通常一个专用的线程用于一台逻辑处理器。 不同于工作站后台垃圾回收线程，这些线程不会超时。

下图显示对服务器上的独立专用线程执行的后台垃圾回收：



请参阅

- 垃圾回收