目录

3.回收技术

自动内存管理器有很多方法可以确定不再需要哪些内存。基本上，垃圾回收取决于确定任何程序变量未指向哪些块。下面简要描述了一些用于执行此操作的技术，但是存在许多潜在的陷阱，并且可能有许多改进。这些技术通常可以结合使用。

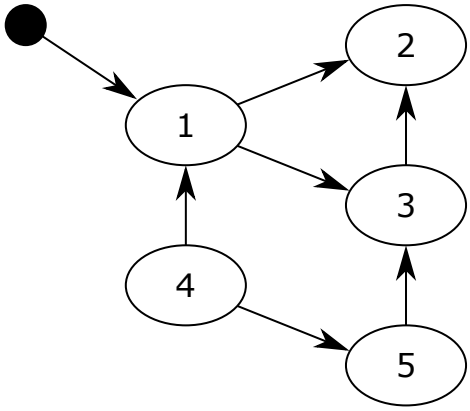
3.1。跟踪收藏家¶

跟随指针从程序变量（称为根集）确定可访问哪些内存块的自动内存管理器称为*跟踪*收集器。经典示例是标记扫描收集器。

3.1.1。标记扫描集合

在标记清除集合中，收集器首先检查程序变量；然后，检查程序变量。指向的所有内存块都将添加到要检查的块列表中。对于该列表上的每个块，它在该块上设置一个标志（标记），以表明它仍然是必需的，并且已经被处理。它还会将尚未标记的该块所指向的任何块添加到列表中。这样，将标记程序可以访问的所有块。

在第二阶段，收集器将*清除*所有分配的内存，以搜索尚未标记的块。如果找到任何内容，则将它们返回给分配器以供重用。



五个存储块，其中三个可通过程序变量访问。

在上图中，可以从程序变量直接访问块1，而可以间接访问块2和3。程序无法到达块4和5。第一步将标记块1，并记住块2和3供以后处理。第二步将标记块2。第三步将标记块3，但不会记住块2，因为它已被标记。清除阶段将忽略块1、2和3，因为它们已被标记，但将回收块4和5。

简单标记清除收集的两个缺点是：

- 它必须扫描正在使用的整个内存，然后才能释放任何内存；
- 它必须运行到完成，或者如果被中断，则必须重新开始。

如果系统需要实时或交互式响应，那么简单的标记清除收集可能不适合目前的情况，但是许多更复杂的垃圾收集算法都可以从该技术中获得。

3.1.2。复制收藏

在分配并回收了许多内存块之后，通常会出现两个问题：

- 使用中的内存广泛分散在内存中，导致大多数现代计算机的内存缓存或 虚拟内存系统性能不佳（称为参考位置差）；

- 分配大块变得困难，因为可用内存被分成小块，并由使用中的块分开（称为 外部碎片）。

解决这两个问题的一种技术是复制垃圾回收。复制垃圾收集器可能会在内存中移动分配的块，并调整对它们的任何引用以指向新位置。这是一项非常强大的技术，可以与许多其他类型的垃圾收集（例如标记清除收集）结合使用。

复制收集的缺点是：

- 很难与增量垃圾回收结合使用（见下文），因为必须调整所有引用以保持一致；
- 难以与保守的垃圾回收结合使用（请参阅下文），因为无法可靠地调整引用。
- 同时存在对象的新旧副本时，需要额外的存储空间；
- 复制数据需要额外的时间（与实时数据量成正比 ）。

3.1.3。增量收集

较旧的垃圾收集算法依赖于能够开始收集并继续工作直到收集完成而不会中断。这使得许多交互式系统在收集过程中暂停，并使垃圾收集的存在变得难以理解。

幸运的是，有现代技术（称为增量垃圾收集）可以使垃圾收集以一系列小的步骤执行，而程序永不停止。在这种情况下，使用和修改块的程序有时称为mutator。在收集器试图确定转换器可以访问哪些内存块的同时，转换器正在忙于分配新的块，修改旧的块以及更改其实际正在查看的块集。

增量收集通常是通过存储硬件或增变器的协作来实现的；这样可以确保每当访问关键位置的内存时，都会执行少量必要的簿记操作，以保持收集器的数据结构正确。

3.1.4。保守的垃圾收集

尽管垃圾回收最早是在1958年发明的，但在设计和实现许多语言时并未考虑到垃圾回收的可能性。通常很难在这种系统中添加普通垃圾收集，但是可以使用一种称为保守垃圾收集的技术 。

这种语言的常见问题是它不向收集器提供有关数据类型的信息，因此收集器无法确定什么是指针，什么不是指针。保守的收集器假设任何东西都*可能*是指针。它会将看起来像指向或指向已分配内存块的指针的任何数据值视为阻止该块的回收。

请注意，由于收集器不确定某些内存位置包含指针，因此无法轻易将其与复制垃圾收集结合使用。复制集合需要知道指针在哪里，以便在块移动时更新它们。

您可能会认为保守的垃圾收集很容易执行得很差，从而导致大量垃圾未被收集。实际上，它做得很好，并且有一些改进可以进一步改善问题。

3.2。参考计数

引用计数是多少的计数的引用（即，指针）有从其他块的特定存储块。它用作一些不依赖跟踪的自动回收技术的基础。

3.2.1。简单的引用计数

In a simple reference counting system, a reference count is kept for each object. This count is incremented for each new reference, and is decremented if a reference is overwritten, or if the referring object is recycled. If a reference count falls to zero, then the object is no longer required and can be recycled.

Reference counting is frequently chosen as an automatic memory management strategy because it seems simple to implement using manual memory management primitives. However, it is hard to implement efficiently because of the cost of updating the counts. It is also hard to implement reliably,

because the standard technique cannot reclaim objects connected in a loop. In many cases, it is an inappropriate solution, and it would be preferable to use tracing garbage collection instead.

Reference counting is most useful in situations where it can be guaranteed that there will be no loops and where modifications to the reference structure are comparatively infrequent. These circumstances can occur in some types of database structure and some file systems. Reference counting may also be useful if it is important that objects are recycled promptly, such as in systems with tight memory constraints.

3.2.2. Deferred reference counting

The performance of reference counting can be improved if not all references are taken into account. In one important technique, known as deferred reference counting, only references from other objects are counted, and references from program variables are ignored. Since most of the references to the object are likely to be from local variables, this can substantially reduce the overhead of keeping the counts up to date. An object cannot be reclaimed as soon as its count has dropped to zero, because there might still be a reference to it from a program variable. Instead, the program variables (including the control stack) are periodically scanned, and any objects which are not referenced from there and which have zero count are reclaimed.

Deferred reference counting cannot normally be used unless it is directly supported by the compiler. It's more common for modern compilers to support tracing garbage collectors instead, because they can reclaim loops. Deferred reference counting may still be useful for its promptness—but that is limited by the frequency of scanning the program variables.

3.2.3. One-bit reference counting

Another variation on reference counting, known as the one-bit reference count, uses a single bit flag to indicate whether each object has either "one" or "many" references. If a reference to an object with "one" reference is removed, then the object can be recycled. If an object has "many" references, then removing references does not change this, and that object will never be recycled. It is possible to store the flag as part of the *pointer* to the object, so no additional space is required in each object to store the count. One-bit reference counting is effective in practice because most actual objects have a reference count of one.

3.2.4. Weighted reference counting

Reference counting is often used for tracking inter-process references for distributed garbage collection. This fails to collect objects in separate processes if they have looped references, but tracing collectors are usually too inefficient as inter-process tracing entails much communication between processes. Within a process, tracing collectors are often used for local recycling of memory.

许多分布式收集器使用称为加权参考计数的技术，该技术甚至进一步降低了通信水平。每次复制参考时，在新副本和旧副本之间共享参考的权重。由于此操作不会更改所有引用的总权重，因此不需要与对象进行任何通信。仅当删除引用时才需要通信。