.NET 面试题解析(06)-GC 与内存管理

系列文章目录地址：

.NET 面试题解析(00)-开篇来谈谈面试 & 系列文章索引

GC 作为.NET 的重要核心基础，是必须要了解的。本文主要侧重于 GC 内存管理中的一些关键点，如要要全面深入了解其精髓，最好还是多看看书。

常见面试题目：

1. 简述一下一个引用对象的生命周期？
2. 创建下面对象实例，需要申请多少内存空间？

```
public class User
{
    public int Age { get; set; }
    public string Name { get; set; }

    public string _Name = "123" + "abc";
    public List<string> _Names;
}
```

3. 什么是垃圾？
4. GC 是什么，简述一下 GC 的工作方式？
5. GC 进行垃圾回收时的主要流程是？
6. GC 在哪些情况下回进行回收工作？
7. using() 语法是如何确保对象资源被释放的？如果内部出现异常依然会释放资源吗？
8. 解释一下 C#里的析构函数？为什么有些编程建议里不推荐使用析构函数呢？
9. Finalize() 和 Dispose() 之间的区别？
10. Dispose 和 Finalize 方法在何时被调用？
11. .NET 中的托管堆中是否可能出现内存泄露的现象？
12. 在托管堆上创建新对象有哪几种常见方式？

深入 GC 与内存管理

托管堆中存放引用类型对象，因此 GC 的内存管理的目标主要都是引用类型对象，本文中涉及的对象如无明确说明都指的是引用类型对象。

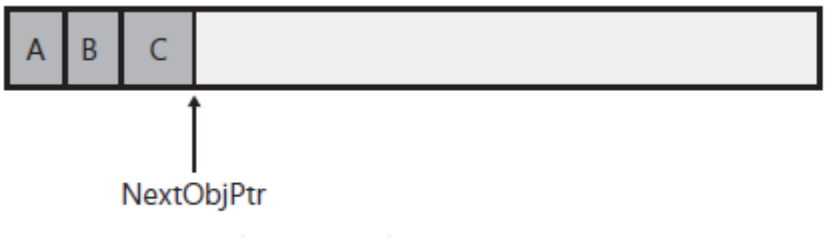
对象创建及生命周期

一个对象的生命周期简单概括就是：创建>使用>释放，在.NET 中一个对象的生命周期：

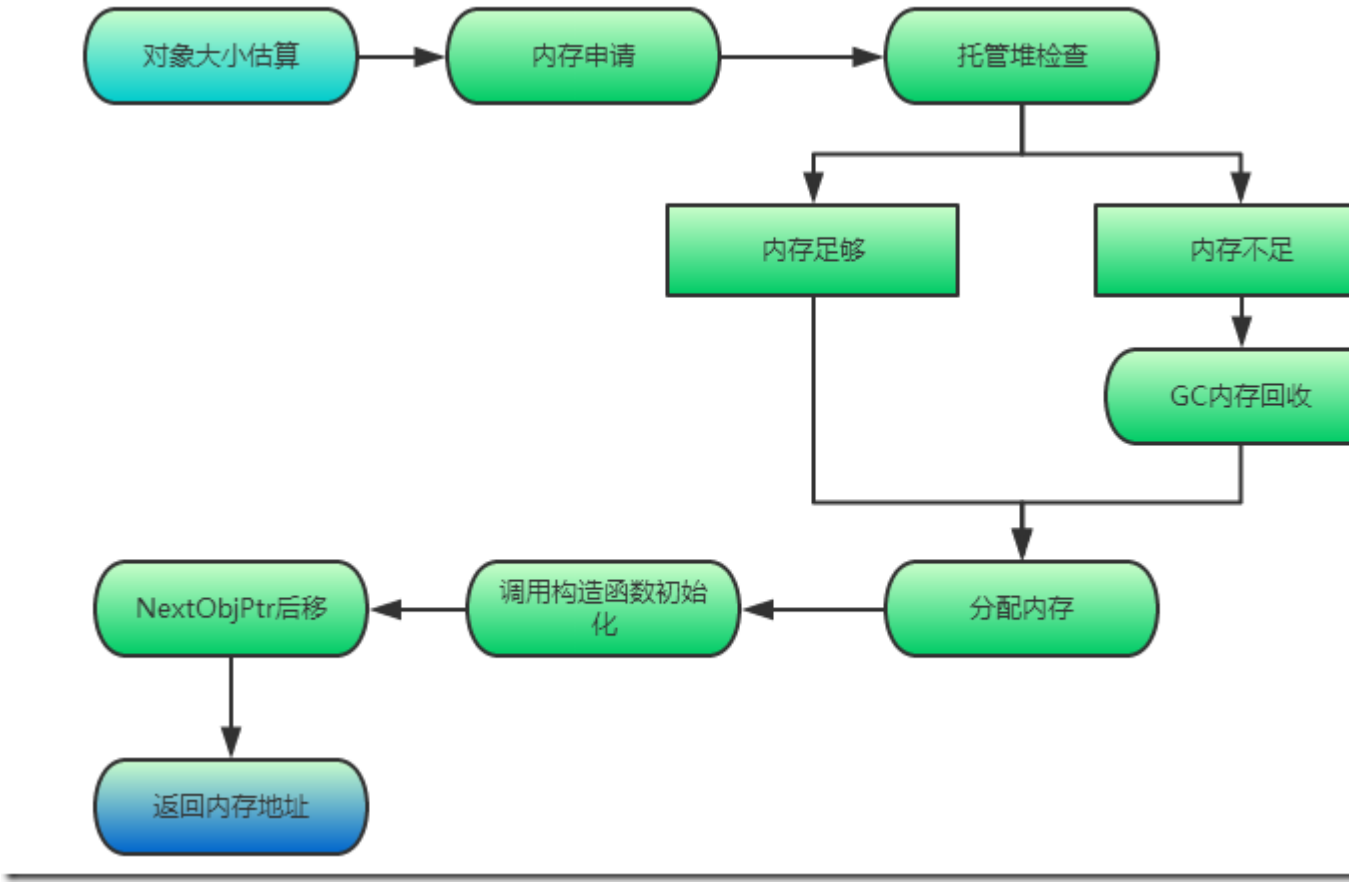
- new 创建对象并分配内存
- 对象初始化
- 对象操作、使用
- 资源清理（非托管资源）
- GC 垃圾回收

那其中重要的一个环节，就是对象的创建，大部分的对象创建都是开始于关键字 new。为什么说是大部分呢，因为有个别引用类型是由专门 IL 指令的，比如 string 有 ldstr 指令（参考前面的文章：.NET 面试题解析(03)-string 与字符串操作），0 基数组好像也有一个专门指令。

引用对象都是分配在托管堆上的， 先来看看托管堆的基本结构，如下图，托管堆中的对象是顺序存放的，托管堆维护着一个指针 NextObjPtr，它指向下一个对象在堆中的分配位置。



创建一个新对象的主要流程：



以题目 2 中的代码为例，模拟一个对象的创建过程：

```
public class User
{
    public int Age { get; set; }
    public string Name { get; set; }

    public string _Name = "123" + "abc";
    public List<string> _Names;
}
```

- **对象大小估算，共计 40 个字节：**
  - 属性 Age 值类型 Int，4 字节；
  - 属性 Name，引用类型，初始为 NULL，4 个字节，指向空地址；
  - 字段_Name 初始赋值了，由前面的文章（.NET 面试题解析(03)-string 与字符串操作）可知，代码会被编译器优化为_Name="123abc"。一个字符两个字节，字符串占用 2×6+8（附加成员：4 字节 TypeHandle 地址，4 字节同步索引块）=20 字节，总共内存大小=字符串对象 20 字节+_Name 指向字符串的内存地址 4 字节=24 字节；
  - 引用类型字段 List<string> _Names 初始默认为 NULL，4 个字节；
  - User 对象的初始附加成员（4 字节 TypeHandle 地址，4 字节同步索引块）8 个字节；
- **内存申请：** 申请 40 个字节的内存块，从指针 NextObjPtr 开始验证，空间是否足够，若不够则触发垃圾回收。
- **内存分配：** 从指针 NextObjPtr 处开始划分 40 个字节内存块。
- **对象初始化：** 首先初始化对象附加成员，再调用 User 对象的构造函数，对成员初始化，值类型默认初始为 0，引用类型默认初始化为 NULL；
- **托管堆指针后移：** 指针 NextObjPtr 后移 40 个字节。
- **返回内存地址：** 返回对象的内存地址给引用变量。

GC 垃圾回收

GC 是**垃圾回收（Garbage Collect）**的缩写，是.NET 核心机制的重要部分。她的基本工作原理就是遍历托管堆中的对象，标记哪些被使用对象（那些没人使用的就是所谓的垃圾），然后把可达对象转移到一个连续的地址空间（也叫压缩），其余的所有没用的对象内存被回收掉。

首先，需要再次强调一下托管堆内存的结构，如下图，很明确的表明了，只有 GC 堆才是 GC 的管辖区域，关于加载堆在前面文中有提到过（.NET 面试题解析(04)-类型、方法与继承）。GC 堆里面为了提高内存管理效率等因素，有分成多个部分，其中 两个主要部分：

- 0/1/2 代：代龄（Generation）在后面有专门说到；
- 大对象堆(Large Object Heap)，大于 85000 字节的大对象会分配到这个区域，这个区域的主要特点就是：不会轻易被回收；就是回收了也不会被压缩（因为对象太大，移动复制的成本太高了）；

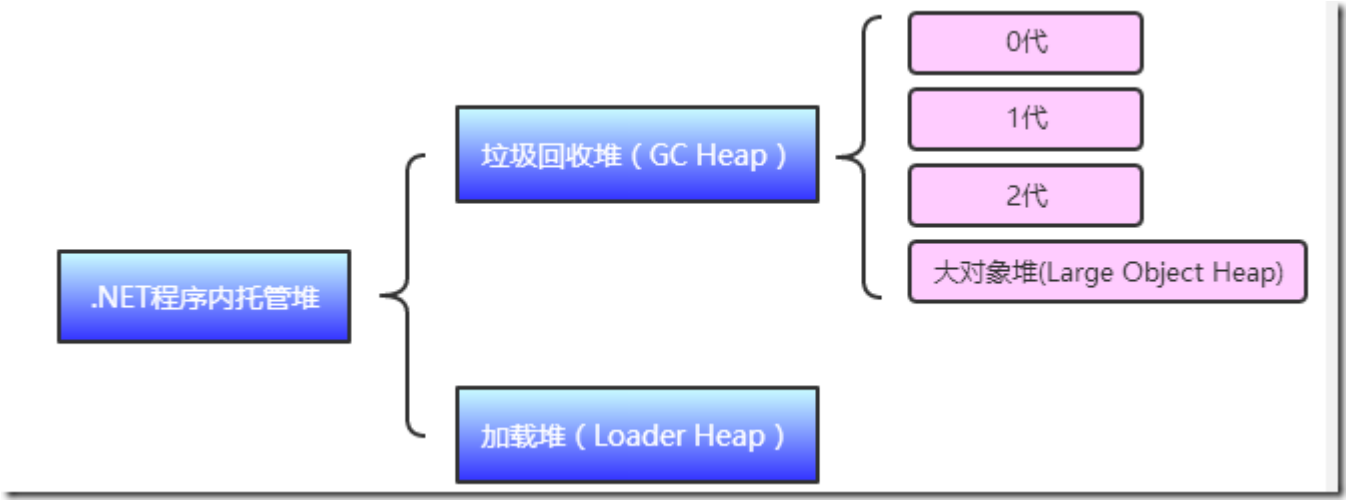


图 3（Figure-3）

什么是垃圾？简单理解就是没有被引用的对象。

垃圾回收的基本流程包含以下三个关键步骤：

① 标记

先假设所有对象都是垃圾，根据应用程序根指针 Root 遍历堆上的每一个引用对象，生成可达对象图，对于还在使用的对象（可达对象）进行标记（其实就是在对象同步索引块中开启一个标示位）。

> 其中 Root 根指针保存了当前所有需要使用的对象引用，他其实只是一个统称，意思就是这些对象当前还在使用，主要包含：静态对象/静态字段的引用；线程栈引用（局部变量、方法参数、栈帧）；任何引用对象的 CPU 寄存器；根引用对象中引用的对象；GC Handle table；Freachable 队列等。

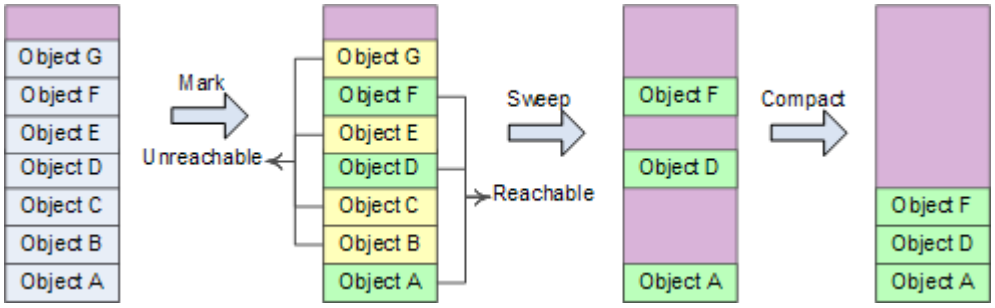
② 清除

针对所有不可达对象进行清除操作，针对普通对象直接回收内存，而对于实现了终结器的对象（实现了析构函数的对象）需要单独回收处理。清除之后，内存就会变得不连续了，就是步骤 3 的工作了。

③ 压缩

把剩下的对象转移到一个连续的内存，因为这些对象地址变了，还需要把那些 Root 跟指针的地址修改为移动后的新地址。

垃圾回收的过程示意图如下：



垃圾回收的过程是不是还挺辛苦的，因此建议不要随意手动调用垃圾回收 GC.Collect()，GC 会选择合适的时机、合适的方式进行内存回收的。

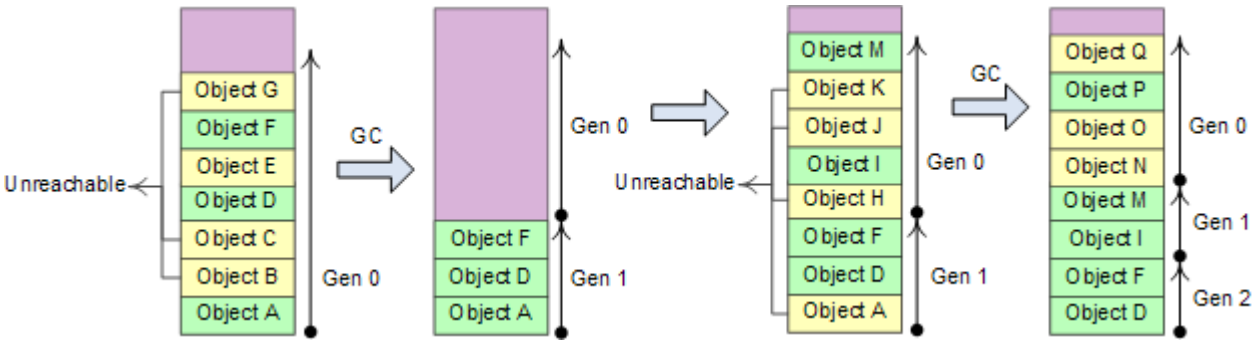
关于代龄（Generation）

当然，实际的垃圾回收过程可能比上面的要复杂，如果没次都扫描托管堆内的所有对象实例，这样做太耗费时间而且没有必要。**分代(Generation)算法是 CLR 垃圾回收器采用的一种机制，它唯一的目的就是提升应用程序的性能**。分代回收，速度显然快于回收整个堆。分代(Generation)算法的假设前提条件：

1、大量新创建的对象生命周期都比较短，而较老的对象生命周期会更长
2、对部分内存进行回收比基于全部内存的回收操作要快
3、新创建的对象之间关联程度通常较强。heap 分配的对象是连续的，关联度较强有利于提高 CPU cache 的命中率

如图 3，.NET 将托管堆分成 3 个代龄区域: Gen 0、Gen 1、Gen 2：

- 第 0 代，最新分配在堆上的对象，从来没有被垃圾收集过。任何一个新对象，当它第一次被分配在托管堆上时，就是第 0 代（大于 85000 的大对象除外）。
- 第 1 代，0 代满了会触发 0 代的垃圾回收，0 代垃圾回收后，剩下的对象会搬到 1 代。
- 第 2 代，当 0 代、1 代满了，会触发 0 代、1 代的垃圾回收，第 0 代升为第 1 代，第 1 代升为第 2 代。



大部分情况，GC 只需要回收 0 代即可，这样可以显著提高 GC 的效率，而且 GC 使用启发式内存优化算法，自动优化内存负载，自动调整各代的内存大小。

非托管资源回收

.NET 中提供释放非托管资源的方式主要是：**Finalize()** 和 **Dispose()**。

Dispose()：

常用的大多是 Dispose 模式，主要实现方式就是实现 IDisposable 接口，下面是一个简单的 IDisposable 接口实现方式。

```
public class SomeType : IDisposable
{
    public MemoryStream _MemoryStream;
    public void Dispose()
    {
        if (_MemoryStream != null) _MemoryStream.Dispose();
    }
}
```

Dispose 需要手动调用，在.NET 中有两中调用方式：

```
//方式 1：显示接口调用
SomeType st1=new SomeType();
//do sth
st1.Dispose();

//方式 2：using()语法调用，自动执行 Dispose 接口
using (var st2 = new SomeType())
{
    //do sth
}
```

第一种方式，显示调用，缺点显而易见，如果程序猿忘了调用接口，则会造成资源得不到释放。或者调用前出现异常，当然这一点可以使用 try…finally 避免。

一般都建议使用第二种实现方式，他可以保证无论如何 Dispose 接口都可以得到调用，原理其实很简单，using()的 IL 代码如下图，因为 using 只是一种语法形式，本质上还是 try…finally 的结构。

```
IL_000d:  nop
IL_000e:  newobj     instance void CLRTest.ConsoleTest.BlogTest/SomeType::.ctor()
IL_0013:  stloc.1
.try
{
  IL_0014:  nop
  IL_0015:  nop
  IL_0016:  leave.s    IL_0028
}  // end .try
finally
{
  IL_0018:  ldloc.1
  IL_0019:  ldnull
  IL_001a:  ceq
  IL_001c:  stloc.2
  IL_001d:  ldloc.2
  IL_001e:  brtrue.s   IL_0027
  IL_0020:  ldloc.1
  IL_0021:  callvirt   instance void [mscorlib]System.IDisposable::Dispose()
  IL_0026:  nop
  IL_0027:  endfinally
}  // end handler
```

Finalize() ：终结器（析构函数）

首先了解下 Finalize 方法的来源，她是来自 System.Object 中受保护的虚方法 Finalize，无法被子类显示重写，也无法显示调用，是不是有点怪？。她的作用就是用来释放非托管资源，由 GC 来执行回收，因此可以保证非托管资源可以被释放。

- 无法被子类显示重写：.NET 提供类似 C++析构函数的形式来实现重写，因此也有称之为析构函数，但其实她只是外表和 C++里的析构函数像而已。
- 无法显示调用：由 GC 来管理和执行释放，不需要手动执行了，再也不用担心猿们忘了调用 Dispose 了。

所有实现了终结器（析构函数）的对象，会被 GC 特殊照顾，GC 的终止化队列跟踪所有实现了 Finalize 方法（析构函数）的对象。

- 当 CLR 在托管堆上分配对象时，GC 检查该对象是否实现了自定义的 Finalize 方法（析构函数）。如果是，对象会被标记为可终结的，同时这个对象的指针被保存在名为终结队列的内部队列中。终结队列是一个由垃圾回收器维护的表，它指向每一个在从堆上删除之前必须被终结的对象。
- 当 GC 执行并且检测到一个不被使用的对象时，需要进一步检查“终结队列”来查询该对象类型是否含有 Finalize 方法，如果没有则将该对象视为垃圾，如果存在则将该对象的引用移动到另外一张 Freachable 列表，此时对象会被复活一次。
- CLR 将有一个单独的高优先级线程负责处理 Freachable 列表，就是依次调用其中每个对象的 Finalize 方法，然后删除引用，这时对象实例就被视为不再被使用，对象再次变成垃圾。
- 下一个 GC 执行时，将释放已经被调用 Finalize 方法的那些对象实例。

上面的过程是不是很复杂！是就对了，如果想彻底搞清楚，没有捷径，不要偷懒，还是去看书吧！

> **简单总结一下：**Finalize()可以确保非托管资源会被释放，但需要很多额外的工作（比如终结对象特殊管理），而且 GC 需要执行两次才会真正释放资源。听上去好像缺点很多，她唯一的优点就是不需要显示调用。

有些编程意见或程序猿不建议大家使用 Finalize，尽量使用 Dispose 代替，我觉得可能主要原因在于：第一是 Finalize 本身性能并不好；其次很多人搞不清楚 Finalize 的原理，可能会滥用，导致内存泄露。因此就干脆别用了，其实微软是推荐大家使用的，不过是和 Dispose 一起使用，同时实现 IDisposable 接口和 Finalize（析构函数），其实 FCL 中很多类库都是这样实现的，这样可以兼具两者的优点：

- 如果调用了 Dispose，则可以忽略对象的终结器，对象一次就回收了；
- 如果程序猿忘了调用 Dispose，则还有一层保障，GC 会负责对象资源的释放；

性能优化建议

尽量不要手动执行垃圾回收的方法：GC.Collect()

垃圾回收的运行成本较高（涉及到了对象块的移动、遍历找到不再被使用的对象、很多状态变量的设置以及 Finalize 方法的调用等等），对性能影响也较大，因此我们在编写程序时，应该避免不必要的内存分配，也尽量减少或避免使用 GC.Collect()来执行垃圾回收，一般 GC 会在最适合的时间进行垃圾回收。

而且还需要注意的一点，在执行垃圾回收的时候，所有线程都是要被挂起的（如果回收的时候，代码还在执行，那对象状态就不稳定了，也没办法回收了）。

推荐 Dispose 代替 Finalize

如果你了解 GC 内存管理以及 Finalize 的原理，可以同时使用 Dispose 和 Finalize 双保险，否则尽量使用 Dispose。

选择合适的垃圾回收机制：工作站模式、服务器模式

题目答案解析:

1. 简述一下一个引用对象的生命周期？

- new 创建对象并分配内存
- 对象初始化
- 对象操作、使用
- 资源清理（非托管资源）
- GC 垃圾回收

2. 创建下面对象实例，需要申请多少内存空间？

```
public class User
{
    public int Age { get; set; }
    public string Name { get; set; }

    public string _Name = "123" + "abc";
    public List<string> _Names;
}
```

40 字节内存空间，详细分析文章中给出了。

3. 什么是垃圾？

一个变量如果在其生存期内的某一时刻已经不再被引用，那么，这个对象就有可能成为垃圾

4. GC 是什么，简述一下 GC 的工作方式？

GC 是**垃圾回收（Garbage Collect）**的缩写，是.NET 核心机制的重要部分。她的基本工作原理就是遍历托管堆中的对象，标记哪些被使用对象（哪些没人使用的就是所谓的垃圾），

然后把可达对象转移到一个连续的地址空间（也叫压缩），其余的所有没用的对象内存被回收掉。

5. GC 进行垃圾回收时的主要流程是？

① **标记**：先假设所有对象都是垃圾，根据应用程序根 Root 遍历堆上的每一个引用对象，生成可达对象图，对于还在使用的对象（可达对象）进行标记（其实就是在对象同步索引块中开启一个标示位）。
② **清除**：针对所有不可达对象进行清除操作，针对普通对象直接回收内存，而对于实现了终结器的对象（实现了析构函数的对象）需要单独回收处理。清除之后，内存就会变得不连续了，就是步骤 3 的工作了。
③ **压缩**：把剩下的对象转移到一个连续的内存，因为这些对象地址变了，还需要把那些 Root 跟指针的地址修改为移动后的新地址。

6. GC 在哪些情况下回进行回收工作？

- 内存不足溢出时（0 代对象充满时）
- Windwos 报告内存不足时，CLR 会强制执行垃圾回收
- CLR 卸载 AppDomian，GC 回收所有
- 调用 GC.Collect
- 其他情况，如主机拒绝分配内存，物理内存不足，超出短期存活代的存段门限

7. using() 语法是如何确保对象资源被释放的？如果内部出现异常依然会释放资源吗？

using() 只是一种语法形式，其本质还是 try…finally 的结构，可以保证 Dispose 始终会被执行。

8. 解释一下 C#里的析构函数？为什么有些编程建议里不推荐使用析构函数呢？

C#里的析构函数其实就是终结器 Finalize，因为长得像 C++里的析构函数而已。

有些编程建议里不推荐使用析构函数要原因在于：第一是 Finalize 本身性能并不好；其次很多人搞不清楚 Finalize 的原理，可能会滥用，导致内存泄露，因此就干脆别用了

9. Finalize() 和 Dispose() 之间的区别？

Finalize() 和 Dispose()都是.NET 中提供释放非托管资源的方式，他们的主要区别在于执行者和执行时间不同：

- finalize 由垃圾回收器调用；dispose 由对象调用。
- finalize 无需担心因为没有调用 finalize 而使非托管资源得不到释放，而 dispose 必须手动调用。
- finalize 不能保证立即释放非托管资源，Finalizer 被执行的时间是在对象不再被引用后的某个不确定的时间；而 dispose 一调用便释放非托管资源。
- 只有 class 类型才能重写 finalize，而结构不能；类和结构都能实现 IDispose。

另外一个重点区别就是终结器会导致对象复活一次，也就说会被 GC 回收两次才最终完成回收工作，这也是有些人不建议开发人员使用终结器的主要原因。

10. Dispose 和 Finalize 方法在何时被调用？

- Dispose 一调用便释放非托管资源；
- Finalize 不能保证立即释放非托管资源，Finalizer 被执行的时间是在对象不再被引用后的某个不确定的时间；

11. .NET 中的托管堆中是否可能出现内存泄露的现象？

是的，可能会。比如：

- 不正确的使用静态字段，导致大量数据无法被 GC 释放；
- 没有正确执行 Dispose()，非托管资源没有得到释放；
- 不正确的使用终结器 Finalize()，导致无法正常释放资源；
- 其他不正确的引用，导致大量托管对象无法被 GC 释放；

12. 在托管堆上创建新对象有哪几种常见方式？

- new 一个对象；
- 字符串赋值，如 string s1=”abc”；
- 值类型装箱；