

# 一、值类型与引用类型解析

常见面试题目：

1. 值类型和引用类型的区别？
2. 结构和类的区别？
3. delegate是引用类型还是值类型？enum、int[]和string呢？
4. 堆和栈的区别？
5. 什么情况下会在堆（栈）上分配数据？它们有性能上的区别吗？
6. “结构”对象可能分配在堆上吗？什么情况下会发生，有什么需要注意的吗？
7. 理解参数按值传递？以及按引用传递？
8. out 和 ref 的区别与相同点？
9. C#支持哪几个预定义的值类型？C#支持哪些预定义的引用类型？
10. 有几种方法可以判定值类型和引用类型？
11. 说说值类型和引用类型的生命周期？
12. 如果结构体中定义引用类型，对象在内存中是如何存储的？例如下面结构体中的class类 User对象是存储在栈上，还是堆上？

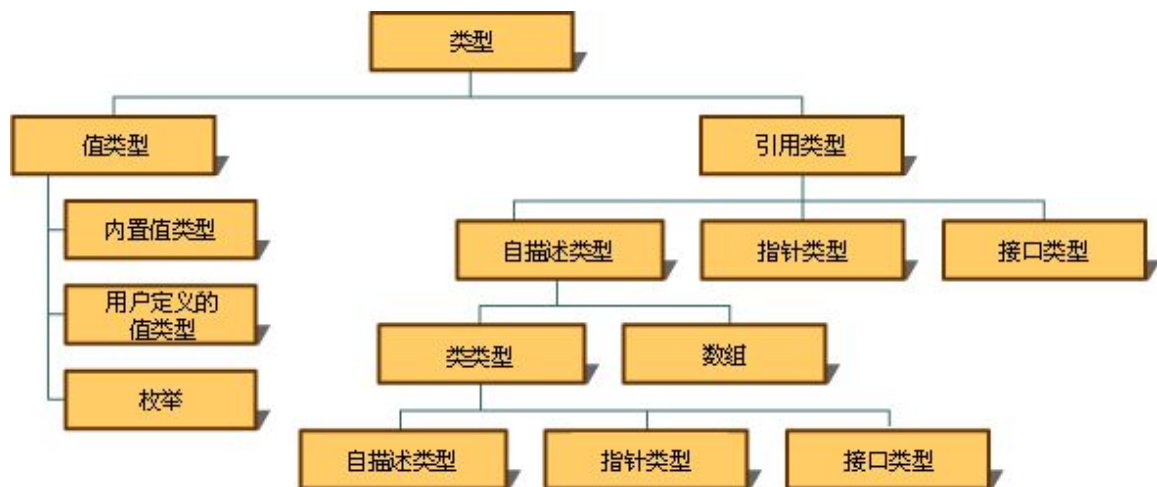
```
1 public struct MyStruct{  
2     public int Index;  
3     public User User;  
4 }
```

## 认识值类型与引用类型

先来认识值类型和引用类型，万变不离其宗，只要搞清楚值类型和引用类型的原理，上面所有题目就都迎刃而解了。

微笑 基本概念

CLR支持两只类型：引用类型和值类型。这是.NET语言的基础和关键，他们从类型定义、实例创建、参数传递，到内存分配都有所不同。虽然看上去简单，但真正理解其内涵的人却好像并不多。



下图清晰展示了.NET中类型分类，值类型主要是一些简单的、基础的数据类型，引用类型主要用于更丰富的、复杂的、复合的数据类型。

## 内存结构

值类型和引用类型最根源的区别就是其内存分配的差异，在这之前首先要理解CLR的内存中两个重要的概念：

**Stack 栈：**线程栈，由操作系统管理，存放值类型、引用类型变量（就是引用对象在托管堆上的地址）。栈是基于线程的，也就是说一个线程会包含一个线程栈，线程栈中的值类型在对象作用域结束后会被清理，效率很高。

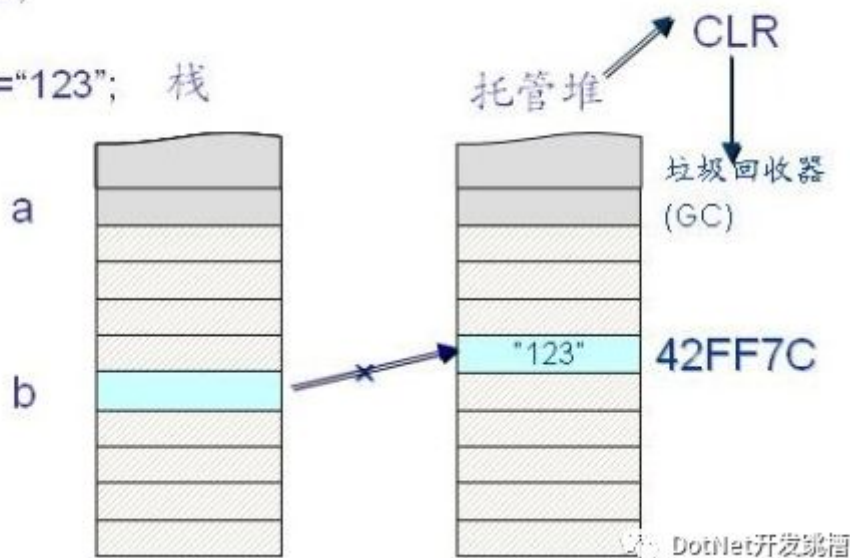
**GC Heap 托管堆：**进程初始化后在进程地址空间上划分的内存空间，存储.NET运行过程中的对象，所有的引用类型都分配在托管堆上，托管堆上分配的对象是由GC来管理和释放的。托管堆是基于进程的，当然托管堆内部还有其他更为复杂的结构，有兴趣的可以深入了解。

结合下图理解，变量a及其值3都是存储在栈上面。变量b在栈上存储，其值指向字符串“123”的托管堆对象地址(字符串是引用类型,字符串对象是存储在托管堆上面。字符串是一个特殊的引用类型，后面文章会专门探讨)。”

# 值类型变量与引用类型变量

Int a = 3;

string b="123"; 栈



值类型一直都存储在栈上面吗？所有的引用类型都存储在托管堆上面吗？

- 1.单独的值类型变量，如局部值类型变量都是存储在栈上面的；
- 2.当值类型是自定义class的一个字段、属性时，它随引用类型存储在托管堆上，此时她是引用类型的一部分；
- 4.所有的引用类型肯定都是存放在托管堆上的。
- 5.还有一种情况，同上面题目12，结构体（值类型）中定义引用类型字段，结构体是存储在栈上，其引用变量字段只存储内存地址，指向堆中的引用实例。

## 对象的传递

将值类型的变量赋值给另一个变量（或者作为参数传递），会执行一次值复制。将引用类型的变量赋值给另一个引用类型的变量，它复制的值是引用对象的内存地址，因此赋值后就会多个变量指向同一个引用对象实例。理解这一点非常重要，下面代码测试验证一下：

```
1 int v1 = 0;
2 int v2 = v1;
3 v2 = 100;
4 Console.WriteLine("v1=" + v1); //输出: v1=0
//输出: v2=100
5 User u1=new User();
6 u1.Age = 0;
7 User u2 = u1;
8 u2.Age = 100; Console.WriteLine("u1.Age=" + u1.Age); //输出: u1.Age=100
Console.WriteLine("u2.Age=" + u2.Age); //输出: u2.Age=100, 因为u1/u2指向同一个对象
```

当把对象作为参数传递的时候，效果同上面一样，他们都称为按值传递，但因为值类型和引用类型的区别，导致其产生的效果也不同。

### 参数-按值传递：

```
1 private void DoTest(int a) { a *= 2; } private void
  DoUserTest(User user) { user.Age *= 2; }
2 [NUnit.Framework.Test]
3 public void DoParaTest()
4 {
5     int a = 10;
6     DoTest(a);
7     Console.WriteLine("a=" + a); //输出: a=10
8     User user = new User();
9     user.Age = 10;
10    DoUserTest(user); Console.WriteLine("user.Age=" +
    user.Age); //输出: user.Age=20 }
```

上面的代码示例，两个方法的参数，都是按值传递

- 对于值类型(int a)：传递的是变量a的值拷贝副本，因此原本的a值并没有改变。
- 对于引用类型(User user)：传递的是变量user的引用地址（User对象实例的内存地址）拷贝副本，因此他们操作都是同一个User对象实例。

### 参数-按引用传递：

按引用传递的两个主要关键字：out 和 ref不管值类型还是引用类型，按引用传递的效果是一样的，都不传递值副本，而是引用的引用（类似c++的指针的指针）。out 和 ref告诉编译器方法传递额是参数地址，而不是参数本身，理解这一点很重要。

代码简单测试一下，如果换成out效果是相同的

```
1 private void DoTest( ref int a) { a *= 2; } private void
  DoUserTest(ref User user) { user.Age *= 2; }
2 [NUnit.Framework.Test] public void DoParaTest() { int a = 10;
  DoTest(ref a); Console.WriteLine("a=" + a); //输出: a=20 ,a的值改变
 了 User user = new User(); user.Age = 10;
  DoUserTest(ref user); Console.WriteLine("user.Age=" + user.Age); //输出:
  user.Age=20 }
```

### out 和 ref的主要异同：

- out 和 ref都指示编译器传递参数地址，在行为上是相同的；
- 他们的使用机制稍有不同，ref要求参数在使用之前要显式初始化，out要在方法内部初始化；
- out 和 ref不可以重载，就是不能定义Method(ref int a)和Method(out int a)这样的重载，从编译角度看，二者的实质是相同的，只是使用时有区别；

常见问题

| 问 题                               | 值 类 型                               | 引 用 类 型  |
|-----------------------------------|-------------------------------------|--|
| 这个类型分配在哪里？                        | 分配在栈上                               | 分配在托管堆上  |
| 变量是怎么表示的？                         | 值类型变量是局部复制                          | 引用类型变量指向被分配得实例所占的内存                            |
| 基类型是什么？                           | 必须继承自System.ValueType               | 可以继承自除了System.ValueType以外的任何类型，只要那个类型不是sealed的 |
| 这个类型能作为其他类型的基类吗？                  | 不能。值类型是密封的，不能被继承                    | 是的。如果这个类型不是密封的，它可以作为其他类型的基类                    |
| 默认的参数传递是什么？                       | 变量是按值传递的（也就是，一个变量的副本被传入被调用的函数）      | 变量是按引用传递（例如，变量的地址传入被调用的函数）                     |
| 这个类型能重写System.Object.Finalize()吗？ | 不能。值类型不好放在堆上，因此不需要被终结。              | 可以间接地重写  |
| 我可以为这个类型定义构造函数吗？                  | 是的，但是默认的构造函数被保留（也就是自定义构造函数必须全部带有参数） | 当然！  |
| 这个类型的变量什么时候消亡？                    | 当它们超出定义的作用域时。                       | 当托管堆被垃圾回收时。                                    |

题目答案解析:

1. 值类型和引用类型的区别？

值类型包括简单类型、结构体类型和枚举类型，引用类型包括自定义类、数组、接口、委托等。

- 1、赋值方式：将一个值类型变量赋给另一个值类型变量时，将复制包含的值。这与引用类型变量的赋值不同，引用类型变量的赋值只复制对象的引用（即内存地址，类似C++中的指针），而不复制对象本身。
- 2、继承：值类型不可能派生出新的类型，所有的值类型均隐式派生自 System.ValueType。但与引用类型相同的是，结构也可以实现接口。
- 3、null：与引用类型不同，值类型不可能包含 null 值。然而，可空类型允许将 null 赋给值类型（他其实只是一种语法形式，在clr底层做了特殊处理）。
- 4、每种值类型均有一个隐式的默认构造函数来初始化该类型的默认值，值类型初始会默认为0，引用类型默认为null。
- 5、值类型存储在栈中，引用类型存储在托管堆中。

2. 结构和类的区别？

结构体是值类型，类是引用类型，主要区别如题1。其他的区别：

- 结构不支持无参构造函数，不支持析构函数，并且不能有protected修饰；
- 结构常用于数据存储，类class多用于行为；
- class需要用new关键字实例化对象，struct可以不适用new关键字；
- class可以为抽象类，struct不支持抽象；



### 3. delegate是引用类型还是值类型？enum、int[]和string呢？

enum枚举是值类型，其他都是引用类型。

### 4. 堆和栈的区别？

1 线程堆栈：简称栈 Stack

2 托管堆：简称堆 Heap

- 值类型大多分配在栈上，引用类型都分配在堆上；
- 栈由操作系统管理，栈上的变量在其作用域完成后就被释放，效率较高，但空间有限。堆受CLR的GC控制；
- 栈是基于线程的，每个线程都有自己的线程栈，初始大小为1M。堆是基于进程的，一个进程分配一个堆，堆的大小由GC根据运行情况动态控制；

### 6. “结构”对象可能分配在堆上吗？什么情况下会发生，有什么需要注意的吗？

结构是值类型，有两种情况会分配在堆上面：

- 结构作为class的一个字段或属性，会随class一起分配在堆上面；
- 装箱后会在堆中存储，尽量避免值类型的装箱，值类型的拆箱和装箱都有性能损失，下一篇会重点关注；

### 7. 理解参数按值传递？以及按引用传递？

- 按值传递：对于值类型传递的是它的值拷贝副本，而引用类型传递的是引用变量的内存地址，他们还是指向的同一个对象。
- 按引用传递：通过关键字out和ref传递参数的内存地址，值类型和引用类型的效果是相同的。

### 8. out 和 ref 的区别与相同点？

- out 和 ref 都指示编译器传递参数地址，在行为上是相同的；
- 他们的使用机制稍有不同，ref要求参数在使用之前要显式初始化，out要在方法内部初始化；
- out 和 ref 不可以重载，就是不能定义Method(ref int a)和Method(out int a)这样的重载，从编译角度看，二者的实质是相同的，只是使用时有区别；

### 9. C#支持哪几个预定义的值类型？C#支持哪些预定义的引用类型？

值类型：整数、浮点数、字符、bool和decimal

引用类型：Object, String

### 10. 有几种方法可以判定值类型和引用类型？

简单来说，继承自System.ValueType的是值类型，反之是引用类型。

### 11. 说说值类型和引用类型的生命周期？

值类型在作用域结束后释放。

引用类型由GC垃圾回收期回收。这个答案可能太简单了，更详细的答案在后面的文章会说到。

## 12. 如果结构体中定义引用类型，对象在内存中是如何存储的？例如下面结构体中的class类 User对象是存储在栈上，还是堆上？

```
1 public struct MyStruct {  
2     public int Index;  
3     public User User; }  
4
```

MyStruct存储在栈中，其字段User的实例存储在堆中，MyStruct.User字段存储指向User对象的内存地址。

关注微信公众号：DotNet开发跳槽

## 二、拆箱与装箱

### 常见面试题目：

- 1.什么是拆箱和装箱？
- 2.什么是箱子？
- 3.箱子放在哪里？
- 4.装箱和拆箱有什么性能影响？
- 5.如何避免隐身装箱？
- 6.箱子的基本结构？
- 7.装箱的过程？
- 8.拆箱的过程？
- 9.下面这段代码输出什么？共发生多少次装箱？多少次拆箱？

```
1
```

```
1 int i = 5;object obj = i;  
2 IFormattable ftt = i;  
3 Console.WriteLine(System.Object.ReferenceEquals(i, obj));  
4 Console.WriteLine(System.Object.ReferenceEquals(i, ftt));  
5 Console.WriteLine(System.Object.ReferenceEquals(ftt, obj));  
6 Console.WriteLine(System.Object.ReferenceEquals(i, (int)obj));  
7 Console.WriteLine(System.Object.ReferenceEquals(i, (int)ftt));
```

有拆必有装，有装必有拆。

在上一文中我们提到，所有值类型都是继承自System.ValueType，而System.ValueType又是来自何方呢，不难发现System.ValueType继承自System.Object。因此**Object是.NET中的万物之源**，几乎所有类型都来自她，这是装箱与拆箱的基础。

特别注意的是，本文与上一文有直接关联，需要先了解上一文中值类型与引用类型的原理，才可以更好理解本文的内容。

## 基本概念

**拆箱与装箱就是值类型与引用类型的转换，她是值类型和引用类型之间的桥梁，他们可以相互转换的一个基本前提就是上面所说的：Object是.NET中的万物之源**

先看看一个小小的实例代码：

```
1      int x = 1023;
2      object o = x; //装箱
3      int y = (int) o; //拆箱
```

装箱：值类型转换为引用对象，一般是转换为System.Object类型或值类型实现的接口引用类型；

拆箱：引用类型转换为值类型，注意，这里的引用类型只能是被装箱的引用类型对象；

由于值类型和引用类型在内存分配的不同，从内存执行角度看，拆箱与装箱就势必存在内存的分配与数据的拷贝等操作，这也是装箱与拆箱性能影响的根源。

## 装箱的过程

```
1  int x = 1023;
2  object o = x; //装箱
```

装箱就是把值类型转换为引用类型，具体过程：

- 1.在堆中申请内存，内存大小为值类型的大小，再加上额外固定空间（引用类型的标配：TypeHandle和同步索引块）；
- 2.将值类型的字段值（x=1023）拷贝新分配的内存中；
- 3.返回新引用对象的地址（给引用变量object o）

如上图所示，装箱后内存有两个对象：一个是值类型变量x，另一个就是新引用对象o。装箱对应的IL指令为**box**，上面装箱的IL代码如下图：

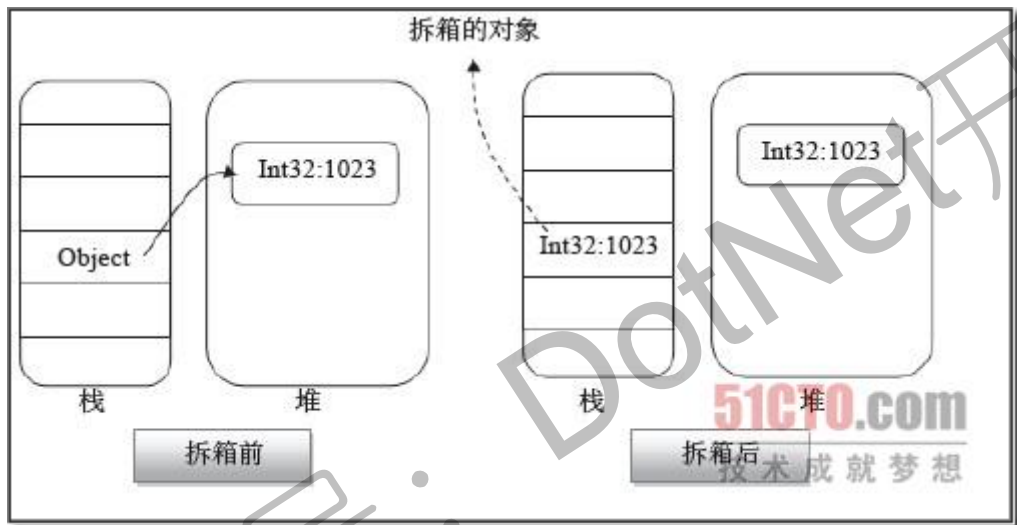


## 拆箱的过程

```
1 int x = 1023;
2 object o = x; //装箱
3 int y = (int) o; //拆箱
```

明白了装箱，拆箱就是装箱相反的过程，简单的说是把装箱后的引用类型转换为值类型。具体过程：

- 1.检查实例对象（object o）是否有效，如是否为null，其装箱的类型与拆箱的类型（int）是否一致，如检测不合法，抛出异常；
- 2.指针返回，就是获取装箱对象（object o）中值类型字段值的地址；
- 3.字段拷贝，把装箱对象（object o）中值类型字段值拷贝到栈上，意思就是创建一个新的值类型变量来存储拆箱后的值；



如上图所示，拆箱后，得到一个新的值类型变量y，拆箱对应的IL指令为unbox，拆箱的IL代码如下：

```
IL_000e: ldloc.1
IL_000f: unbox.any [mscorlib]System.Int32 int y = (int) o; //拆箱
IL_0014: stloc.2
```

## 装箱与拆箱总结及性能



### 装的的什么？拆的又是什么？什么是箱子？

通过上面深入了解了装箱与拆箱的原理，不难理解，只有值类型可以装箱，拆的就是装箱后的引用对象，箱子就是一个存放了值类型字段的引用对象实例，箱子存储在托管堆上。只有值类型才有装箱、拆箱两个状态，而引用类型一直都在箱子里。

## 关于性能

之所以关注装箱与拆箱，主要原因就是他们的性能问题，而且在日常编码中，经常有装箱与拆箱的操作，而且这些装箱与拆箱的操作往往是在不经意时发生。**一般来说，装箱的性能开销更大**，这不难理解，因为引用对象的分配更加复杂，成本也更高，值类型分配在栈上，分配和释放的效率都很高。装箱过程是需要创建一个新的引用类型对象实例，拆箱过程需要创建一个值类型字段，开销更低。为了尽量避免这种性能损失，尽量使用泛型，在代码编写中也尽量避免隐式装箱。

## 什么是隐式装箱？如何避免？

就是不经意的代码导致多次重复的装箱操作，看看代码就好理解了

```
1 int x = 100;
2 ArrayList arr = new ArrayList(3);
3 arr.Add(x);
4 arr.Add(x);
5 arr.Add(x);
```

这段代码共有多少次装箱呢？看看Add方法的定义：

```
public virtual int Add(object value);
```

再看看IL代码，可以准确的得到装箱的次数：

```
IL_0005: newobj instance void [mscorlib]System.Collections.ArrayList::.ctor(int32)
IL_000a: stloc.1
IL_000b: ldloc.1
IL_000c: ldloc.0
IL_000d: box [mscorlib]System.Int32 第一次装箱
IL_0012: callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
IL_0017: pop
IL_0018: ldloc.1
IL_0019: ldloc.0
IL_001a: box [mscorlib]System.Int32 第二次装箱
IL_001f: callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
IL_0024: pop
IL_0025: ldloc.1
IL_0026: ldloc.0
IL_0027: box [mscorlib]System.Int32 第三次装箱
IL_002c: callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
```

显示装箱可以避免隐式装箱，下面修改后的代码就只有一次装箱了。

```
1 int x = 100;
2 ArrayList arr = new ArrayList(3);
3 object o = x;
4 arr.Add(o);
5 arr.Add(o);
```

## 题目答案解析:

### 1.什么是拆箱和装箱?

装箱就是值类型转换为引用类型，拆箱就是引用类型（被装箱的对象）转换为值类型。

### 2.什么是箱子?

就是引用类型对象。

### 3.箱子放在哪里?

托管堆上。

### 4.装箱和拆箱有什么性能影响?

装箱和拆箱都涉及到内存的分配和对象的创建，有较大的性能影响。

### 5.如何避免隐身装箱?

编码中，多使用泛型、显示装箱。

### 6.箱子的基本结构?

上面说了，箱子就是一个引用类型对象，因此她的结构，主要包含两部分：

- 值类型字段值；
- 引用类型的标准配置，引用对象的额外空间：TypeHandle和同步索引块，关于这两个概念在本系列后面的文章会深入探讨。

### 7.装箱的过程?

- 1.在堆中申请内存，内存大小为值类型的大小，再加上额外固定空间（引用类型的标配：TypeHandle和同步索引块）；
- 2.将值类型的字段值（x=1023）拷贝新分配的内存中；
- 3.返回新引用对象的地址（给引用变量object o）

### 8.拆箱的过程?

- 1.检查实例对象（object o）是否有效，如是否为null，其装箱的类型与拆箱的类型（int）是否一致，如检测不合法，抛出异常；
- 2.指针返回，就是获取装箱对象（object o）中值类型字段值的地址；
- 3.字段拷贝，把装箱对象（object o）中值类型字段值拷贝到栈上，意思就是创建一个新的值类型变量来存储拆箱后的值；

## 9.下面这段代码输出什么？共发生多少次装箱？多少次拆箱？

```
1 int i = 5;
2 object obj = i;
3 IFormattable ftt = i;
4 Console.WriteLine(System.Object.ReferenceEquals(i, obj));
5 Console.WriteLine(System.Object.ReferenceEquals(i, ftt));
6 Console.WriteLine(System.Object.ReferenceEquals(ftt, obj));
7 Console.WriteLine(System.Object.ReferenceEquals(i, (int)obj));
8 Console.WriteLine(System.Object.ReferenceEquals(i, (int)ftt));
```

上面代码输出如下，至于发生多少次装箱多少次拆箱，你猜？

```
1 False
2 False
3 False
4 False
5 False
```

## 三、string与字符串操作含解析

### 常见面试题目：

- 1.字符串是引用类型还是值类型？
- 2.在字符串连接处理中，最好采用什么方式，理由是什么？
- 3.使用 StringBuilder时，需要注意些什么问题？
- 4.以下代码执行后内存中会存在多少个字符串？分别是什么？输出结果是什么？为什么呢？

```
1 string st1 = "123" + "abc";
2 string st2 = "123abc";
3 Console.WriteLine(st1 == st2);
4 Console.WriteLine(System.Object.ReferenceEquals(st1, st2));
```

- 5.以下代码执行后内存中会存在多少个字符串？分别是什么？输出结果是什么？为什么呢？

```
1 string s1 = "123";
2 string s2 = s1 + "abc";
3 string s3 = "123abc";
4 Console.WriteLine(s2 == s3);
```

```
5 Console.WriteLine(System.Object.ReferenceEquals(s2, s3));
```

6.使用C#实现字符串反转算法，例如：输入"12345", 输出"54321"。

7.下面的代码输出结果？为什么？

```
1 object a = "123";
2 object b = "123";
3 Console.WriteLine(System.Object.Equals(a,b));
4 Console.WriteLine(System.Object.ReferenceEquals(a,b));
5 string sa = "123";
6 Console.WriteLine(System.Object.Equals(a, sa));
7 Console.WriteLine(System.Object.ReferenceEquals(a, sa));
```

## 深入浅出字符串操作

string是一个特殊的引用类型，使用上有点像值类型。之所以特殊，也主要是因为string太常用了，为了提高性能及开发方便，对string做了特殊处理，给予了一些专用特性。为了弥补string在字符串连接操作上的一些性能不足，便有了StringBuilder。

## 认识string

首先需要明确的，string是一个引用类型，其对象值存储在托管堆中。string的内部是一个char集合，他的长度Length就是字符char数组的字符个数。string不允许使用new string()的方式创建实例，而是另一种更简单的语法，直接赋值（string aa= “000”这一点也类似值类型）。

认识string，先从一个简单的示例代码入手：

```
1 public void DoStringTest(){
2     var aa = "000";
3     SetStringValue(aa);
4     Console.WriteLine(aa);
5 }
6 private void SetStringValue(string aa)
7 {
8     aa += "111";
9 }
```

上面的输出结果为“000”。

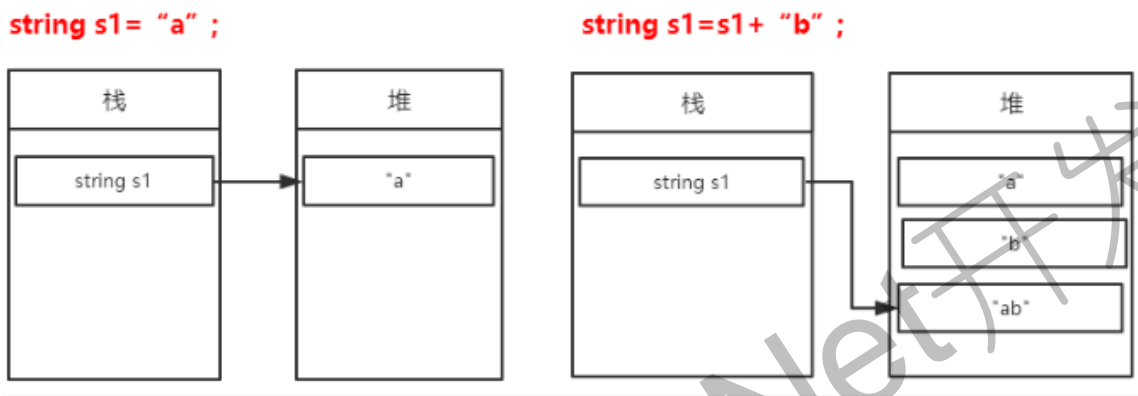
通过前面的值类型与引用类型的文章，我们知道string是一个引用类型，既然是一个引用类型，参数传递的是引用地址，那为什么不是输出“000111”呢？是不是很有值类型的特点呢！这一切的原因源于

string类型的两个重要的特性：**恒定性**与**驻留性**

## String的恒定性（不变性）

字符串是不可变的，字符串一经创建，就不会改变，任何改变都会产生新的字符串。比如下面的代码，堆上先创建了字符串s1=" a"，加上一个字符串 "b" 后，堆上会存在三个个字符串实例，如下图所示。

```
1 string s1 = "a";
2 string s2 = s1 + "b";
```



上文中的“**任何改变都会产生新的字符串**”，包括字符串的一些操作函数，如str1.ToLower, Trim(), Remove(int startIndex, int count), ToUpper()等，都会产生新的字符串，因此在很多编程实践中，对于字符串忽略大小的比较：

```
1 if (str1.ToLower() == str2.ToLower()) //这种方式会产生新的字符串，不推荐
2 if (string.Compare(str1, str2, true)) //这种方式性能更好
```

## String的驻留性



由于字符串的不变性，在大量使用字符串操作时，会导致创建大量的字符串对象，带来极大的性能损失。因此CLR又给string提供另外一个法宝，就是字符串驻留，先看看下面的代码，字符串s1、s2竟然是同一个对象！

```
1 var s1 = "123";
2 var s2 = "123";
3 Console.WriteLine(System.Object.Equals(s1, s2)); //输出 True
4 Console.WriteLine(System.Object.ReferenceEquals(s1, s2)); //输出 True
```

相同的字符串在内存（堆）中只分配一次，第二次申请字符串时，发现已经有该字符串，直接返回已有字符串的地址，这就是驻留的基本过程。



## 字符串驻留的基本原理：

- CLR初始化时会在内存中创建一个驻留池，内部其实是一个哈希表，存储被驻留的字符串和其内存地址。
- 驻留池是进程级别的，多个AppDomain共享。同时她不受GC控制，生命周期随进程，意思就是不会被GC回收（不回收！难道不会造成内存爆炸吗？不要急，且看下文）
- 当分配字符串时，首先会到驻留池中查找，如找到，则返回已有相同字符串的地址，不会创建新字符串对象。如果没有找到，则创建新的字符串，并把字符串添加到驻留池中。

如果大量的字符串都驻留到内存里，而得不到释放，不是很容易造成内存爆炸吗，当然不会了？因为不是任何字符串都会驻留，只有通过IL指令 `ldstr` 创建的字符串才会留用。

字符串创建的有多种方式，如下面的代码：

```
1 var s1 = "123";
2 var s2 = s1 + "abc";
3 var s3 = string.Concat(s1, s2);
4 var s4 = 123.ToString();
5 var s5 = s2.ToUpper();
```

其IL代码如下

```
.method public hidebysig instance void ValueTypeTest() cil managed
{
    .custom instance void [nunit.framework]NUnit.Framework.TestAttribute::ctor() = ( 01 00 00 00 )
    // 代码大小      48 (0x30)
    .maxstack 2
    .locals init ([0] string s1,
        [1] string s2,
        [2] string s3,
        [3] string s4,
        [4] string s5,
        [5] int32 CS$0$0000)
    IL_0000: nop
    IL_0001: ldstr      "123"
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: ldstr      "abc"
    IL_000d: call      string [mscorlib]System.String::Concat(string,
        string)
    IL_0012: stloc.1
    IL_0013: ldloc.0
    IL_0014: ldloc.1
    IL_0015: call      string [mscorlib]System.String::Concat(string,
        string)
    IL_001a: stloc.2
    IL_001b: ldc.i4.s   123
    IL_001d: stloc.s   CS$0$0000
    IL_001f: ldloc.s   CS$0$0000
    IL_0021: call      instance string [mscorlib]System.Int32::ToString()
    IL_0026: stloc.3
    IL_0027: ldloc.1
    IL_0028: callvirt   instance string [mscorlib]System.String::ToUpper()
    IL_002d: stloc.s   s5
    IL_002f: ret
} // end of method BlogTest::ValueTypeTest
```

在上面的代码中，出现两个字符串常量，“123”和“abc”，这个两个常量字符串在IL代码中都是通过IL指令ldstr创建的，只有该指令创建的字符串才会被驻留，其他方式产生新的字符串都不会被驻留，也就不会共享字符串了，会被GC正常回收。

那该如何来验证字符串是否驻留呢，string类提供两个静态方法：

- String.Intern(string str) 可以主动驻留一个字符串；
- String.IsInterned(string str);检测指定字符串是否驻留，如果驻留则返回字符串，否则返回NULL

```
...public string Intern(string str);  
...public static string Intern(string str);  
...public static string IsInterned(string str);
```

请看下面的示例代码

```
1 var s1 = "123";  
2 var s2 = s1 + "abc";  
3 Console.WriteLine(s2);  
4 //输出: 123abc Console.WriteLine(string.IsInterned(s2) ?? "NULL"); //输出: NULL。因为  
  "123abc"没有驻留 string.Intern(s2); //主动驻留字符串  
  Console.WriteLine(string.IsInterned(s2) ?? "NULL"); //输出: 123abc
```

## 认识StringBuilder



大量的编程实践和意见中，都说大量字符串连接操作，应该使用StringBuilder。相对于string的不可变，StringBuilder代表可变字符串，不会像字符串，在托管堆上频繁分配新对象，StringBuilder是个好同志。

首先StringBuilder内部同string一样，有一个char[]字符数组，负责维护字符串内容。因此，与char数组相关，就有两个很重要的属性：

- public int Capacity: StringBuilder的容量，其实就是字符数组的长度。
- public int Length: StringBuilder中实际字符的长度，>=0，<=容量Capacity。

StringBuilder之所以比string效率高，主要原因就是不会创建大量的新对象，StringBuilder在以下两种情况下会分配新对象：

- 追加字符串时，当字符总长度超过了当前设置的容量Capacity，这个时候，会重新创建一个更大的字符数组，此时会涉及到分配新对象。
- 调用StringBuilder.ToString()，创建新的字符串。

## 追加字符串的过程：

- StringBuilder的默认初始容量为16；
- 使用stringBuilder.Append()追加一个字符串时，当字符数大于16，StringBuilder会自动申请一个更大的字符数组，一般是倍增；
- 在新的字符数组分配完成后，将原字符数组中的字符复制到新字符数组中，原字符数组就被无情得抛弃了

(会被GC回收) ;

- 最后把需要追加的字符串追加到新字符数组中;

简单来说, 当StringBuilder的容量**Capacity**发生变化时, 就会引起托管对象申请、内存复制等操作, 带来不好的性能影响, 因此设置合适的初始容量是非常必要的, 尽量减少内存申请和对象创建。代码简单来验证一下:

```
1  StringBuilder sb1 = new StringBuilder();
2  Console.WriteLine("Capacity={0}; Length={1};", sb1.Capacity, sb1.Length);
3  //输出: Capacity=16; Length=0;
4  //初始容量为16
5  sb1.Append('a', 12);
6  //追加12个字符
7  Console.WriteLine("Capacity={0}; Length={1};", sb1.Capacity, sb1.Length);
8  //输出: Capacity=16; Length=12;
9  sb1.Append('a', 20);
10 //继续追加20个字符, 容量倍增了
11 Console.WriteLine("Capacity={0}; Length={1};", sb1.Capacity, sb1.Length);
12 //输出: Capacity=32; Length=32;
13 sb1.Append('a', 41);
14 //追加41个字符, 新容量=32+41=73
15 Console.WriteLine("Capacity={0}; Length={1};", sb1.Capacity, sb1.Length);
16 //输出: Capacity=73; Length=73;
17 StringBuilder sb2 = new StringBuilder(80);
18 //设置一个合适的初始容量
19 Console.WriteLine("Capacity={0}; Length={1};", sb2.Capacity, sb2.Length);
20 //输出: Capacity=80; Length=0; sb2.Append('a', 12);
21 Console.WriteLine("Capacity={0}; Length={1};", sb2.Capacity, sb2.Length);
22 //输出: Capacity=80; Length=12; sb2.Append('a', 20);
23 Console.WriteLine("Capacity={0}; Length={1};", sb2.Capacity, sb2.Length);
24 //输出: Capacity=80; Length=32; sb2.Append('a', 41);
25 Console.WriteLine("Capacity={0}; Length={1};", sb2.Capacity, sb2.Length);
26 //输出: Capacity=80; Length=73;
```

为什么少量字符串不推荐使用StringBuilder呢? 因为StringBuilder本身是有一定的开销的, 少量字符串就不推荐使用了, 使用String.Concat和String.Join更合适。

## 高效得使用字符串



- 在使用线程锁的时候, 不要锁定一个字符串对象, 因为字符串的驻留性, 可能会引发不可以预料的问题;

- 理解字符串的不变性，尽量避免产生额外字符串，如：

```
1 if (str1.ToLower()==str2.ToLower()) //这种方式会产生新的字符串，不推荐
2 if (string.Compare(str1,str2,true)) //这种方式性能更好
```

- 在处理大量字符串连接的时候，尽量使用StringBuilder，在使用StringBuilder时，尽量设置一个合适的长度初始值；
- 少量字符串连接建议使用String.Concat和String.Join代替。

## 题目答案解析:

### 1.字符串是引用类型类型还是值类型？

引用类型。

### 2.在字符串连加处理中，最好采用什么方式，理由是什么？

少量字符串连接，使用String.Concat，大量字符串使用StringBuilder，因为StringBuilder的性能更好，如果string的话会创建大量字符串对象。

### 3.使用 StringBuilder时，需要注意些什么问题？

- 少量字符串时，尽量不要用，StringBuilder本身是有一定性能开销的；
- 大量字符串连接使用StringBuilder时，应该设置一个合适的容量；

### 4.以下代码执行后内存中会存在多少个字符串？分别是什么？输出结果是什么？为什么呢？

```
1 string st1 = "123" + "abc";
2 string st2 = "123abc";
3 Console.WriteLine(st1 == st2);
4 Console.WriteLine(System.Object.ReferenceEquals(st1, st2));
```

输出结果：

```
1 True
2 True
```

内存中的字符串只有一个“123abc”，第一行代码（`string st1 = "123" + "abc";`）常量字符串相加会被编译器优化。由于字符串驻留机制，两个变量st1、st2都指向同一个对象。IL代码如下：

```

.maxstack 2
.locals init ([0] string st1,
              [1] string st2)
IL_0000: nop
IL_0001: ldstr      "123abc"
IL_0006: stloc.0
IL_0007: ldstr      "123abc"
IL_000c: stloc.1
IL_000d: ldloc.0
IL_000e: ldloc.1
IL_000f: call       bool [mscorlib]System.String::op_Equality(string,
                                                         string)
IL_0014: call       void [mscorlib]System.Console::WriteLine(bool)
IL_0019: nop
IL_001a: ldloc.0
IL_001b: ldloc.1
IL_001c: call       bool [mscorlib]System.Object::ReferenceEquals(object,
                                                         object)
IL_0021: call       void [mscorlib]System.Console::WriteLine(bool)
IL_0026: nop
IL_0027: ret

```

5.以下代码执行后内存中会存在多少个字符串？分别是什么？输出结果是什么？为什么呢？

```

1 string s1 = "123";
2 string s2 = s1 + "abc";
3 string s3 = "123abc";
4 Console.WriteLine(s2 == s3);
5 Console.WriteLine(System.Object.ReferenceEquals(s2, s3));

```

和第5题的结果肯定是不一样的，答案留给读者吧，文章太长了，写得好累！

## 6.使用C#实现字符串反转算法，例如：输入"12345"，输出"54321"

这是一道比较综合的考察字符串操作的题目，答案可以有很多种。通过不同的答题可以看出程序猿的基础水平。下面是网上比较认可的两种答案，效率上都是比较不错的。

```

1 public static string Reverse(string str){
2     if (string.IsNullOrEmpty(str))    {
3         throw new ArgumentException("参数不合法");    }
4     StringBuilder sb = new StringBuilder(str.Length);
5     //注意：设置合适的初始长度，可以显著提高效率（避免了多次内存申请）
6     for (int index = str.Length - 1; index >= 0; index--)
7         { sb.Append(str[index]);    }
8     return sb.ToString();}

```

```

1 public static string Reverse(string str){

```

```

2  if (string.IsNullOrEmpty(str))    {
3      throw new ArgumentException("参数不合法");    }
4      char[] chars = str.ToCharArray();
5      int begin = 0;
6      int end = chars.Length - 1;
7      char tempChar;
8      while (begin < end)    {
9          tempChar = chars[begin];
10         chars[begin] = chars[end];
11         chars[end] = tempChar;
12         begin++;        end--;    }
13         string strResult = new string(chars);
14         return strResult;

```

还有一个比较简单也挺有效的方法：

```

1  public static string Reverse(string str){
2      char[] arr = str.ToCharArray();
3      Array.Reverse(arr);
4      return new string(arr);}

```

## 7.下面的代码输出结果？为什么？

```

1  object a = "123";object b = "123";
2  Console.WriteLine(System.Object.Equals(a,b));
3  Console.WriteLine(System.Object.ReferenceEquals(a,b));
4  string sa = "123";
5  Console.WriteLine(System.Object.Equals(a, sa));
6  Console.WriteLine(System.Object.ReferenceEquals(a, sa));

```

输出结果全是True，因为他们都指向同一个字符串实例，使用object声明和string声明在这里并没有区别（string是引用类型）。

使用object声明和string声明到底有没有区别呢？，有点疑惑，一个朋友在面试时面试官有问过这个问题，那个面试官说sa、a是有区别的，且不相等。对于此疑问，欢迎交流。

## 四、类型、方法与继承

### 常见面试题目：

1. 所有类型都继承System.Object吗？



2. 解释virtual、sealed、override和abstract的区别

3. 接口和类有什么异同？

4. 抽象类和接口有什么区别？使用时有什么需要注意的吗？

5. 重载与覆盖的区别？

6. 在继承中new和override相同点和区别？看下面的代码，有一个基类A，B1和B2都继承自A，并且使用不同的方式改变了父类方法Print（）的行为。测试代码输出什么？为什么？

```
1 public void DoTest(){
2     B1 b1 = new B1();
3     B2 b2 = new B2();
4     b1.Print(); b2.Print();
5     //按预期应该输出 B1、B2
6     A ab1 = new B1();
7     A ab2 = new B2();
8     ab1.Print(); ab2.Print();
9     //这里应该输出什么呢？
10    public class A{
11        public virtual void Print()
12        { Console.WriteLine("A"); }
13        public class B1 : A{
14            public override void Print()
15            { Console.WriteLine("B1"); }
16            public class B2 : A{
17                public new void Print()
18                {
19                    Console.WriteLine("B2");
20                }
21            }
22        }
23    }
```

7. 下面代码中，变量a、b都是int类型，代码输出结果是什么？

```
1 int a = 123;
2 int b = 20;
3 var atype = a.GetType();
4 var btype = b.GetType();
5 Console.WriteLine(System.Object.Equals(atype,btype));
6 Console.WriteLine(System.Object.ReferenceEquals(atype,btype));
```

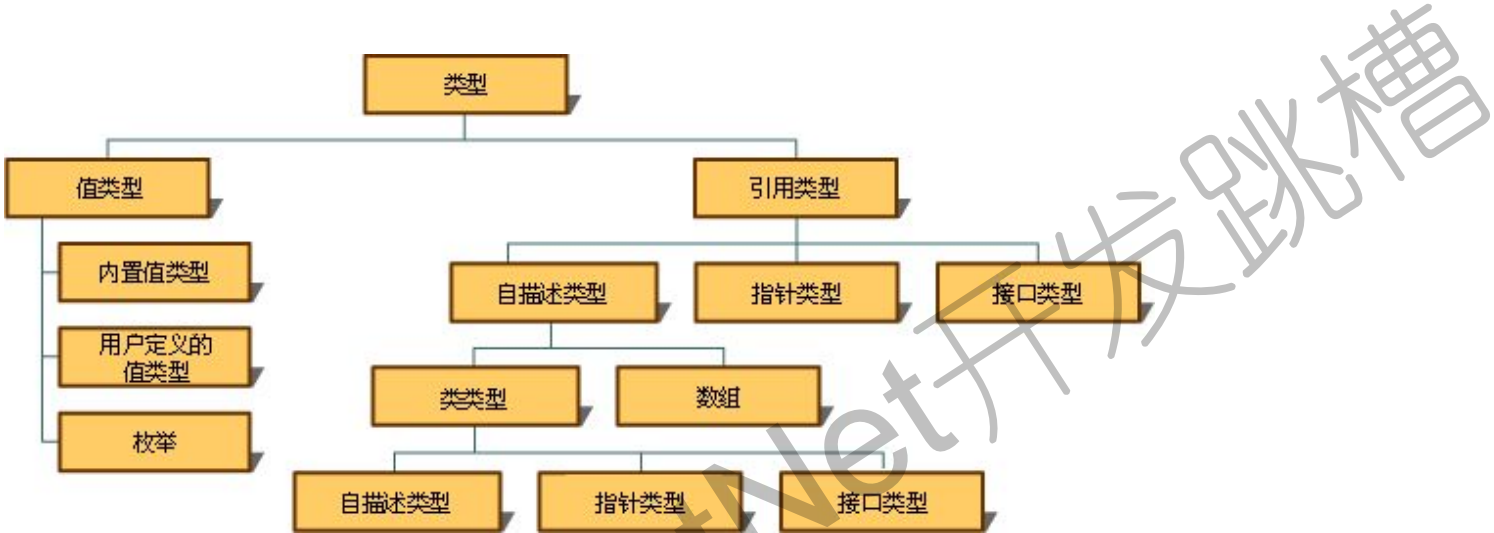
8.class中定义的静态字段是存储在内存中的哪个地方？为什么会说她不会被GC回收？

## 类型基础知识梳理

### 类型Type简述



通过本系列前面几篇文章，基本了解了值类型和引用类型，及其相互关系。如下图，.NET中主要的类型就是值类型和引用类型，所有类型的基类就是`System.Object`，也就是说我们使用FCL提供的各种类型的、自定义的所有类型都最终派生自`System.Object`，因此他们也都继承了`System.Object`提供的基本方法。



`System.Object`可以说是.NET中的万物之源，如果非要较真的话，好像只有接口不继承她了。接口是一个特殊的类型，可以理解为接口是普通类型的约束、规范，它不可以实例化。（实际编码中，接口可以用object表示，只是一种语法支持，此看法不知是否准确，欢迎交流）

在.NET代码中，我们可以很方便得创建各种类型，一个简单的数据模型、复杂的聚合对象类型、或是对客观世界实体的抽象。类 (**class**) 是最基础的 C# 类型（注意：本文主要探讨的就是引用类型，文中所述类型如没注明都为引用类型），支持继承与多态。一个c# 类Class主要包含两种基本成员：

- 状态（字段、常量、属性等）
- 操作（方法、事件、索引器、构造函数等）

利用创建的类型（或者系统提供的），可以很容易得创建对象的实例。使用 `new` 运算符创建，该运算符为新的实例分配内存，调用构造函数初始化该实例，并返回对该实例的引用，如下面的语法形式：

```
<类名> <实例名> = new <类名>([构造函数的参数])
```

创建后的实例对象，是一个存储在内存上（在线程栈或托管堆上）的一个对象，那可以创造实例的类型在内存中又是一个什么样的存在呢？它就是类型对象 (Type Object)。

### 类型对象 (Type Object)



看看下面的代码：

```
1 int a = 123;    // 创建int类型实例a
2 int b = 20;    // 创建int类型实例b
3 var atype = a.GetType(); // 获取对象实例a的类型Type
4 var btype = b.GetType(); // 获取对象实例b的类型Type
5 Console.WriteLine(System.Object.Equals(atype, btype));
6 //输出: True Console.WriteLine(System.Object.ReferenceEquals(atype, btype));
7 //输出: True
```

任何对象都有一个GetType()方法（基类System.Object提供的），该方法返回一个对象的类型，类型上面包含了对象内部的详细信息，如字段、属性、方法、基类、事件等等（通过反射可以获取）。在上面的代码中两个不同的int变量的类型（int.GetType()）是同一个Type，说明int在内存中有唯一的一个（类似静态的）System.Int32类型。

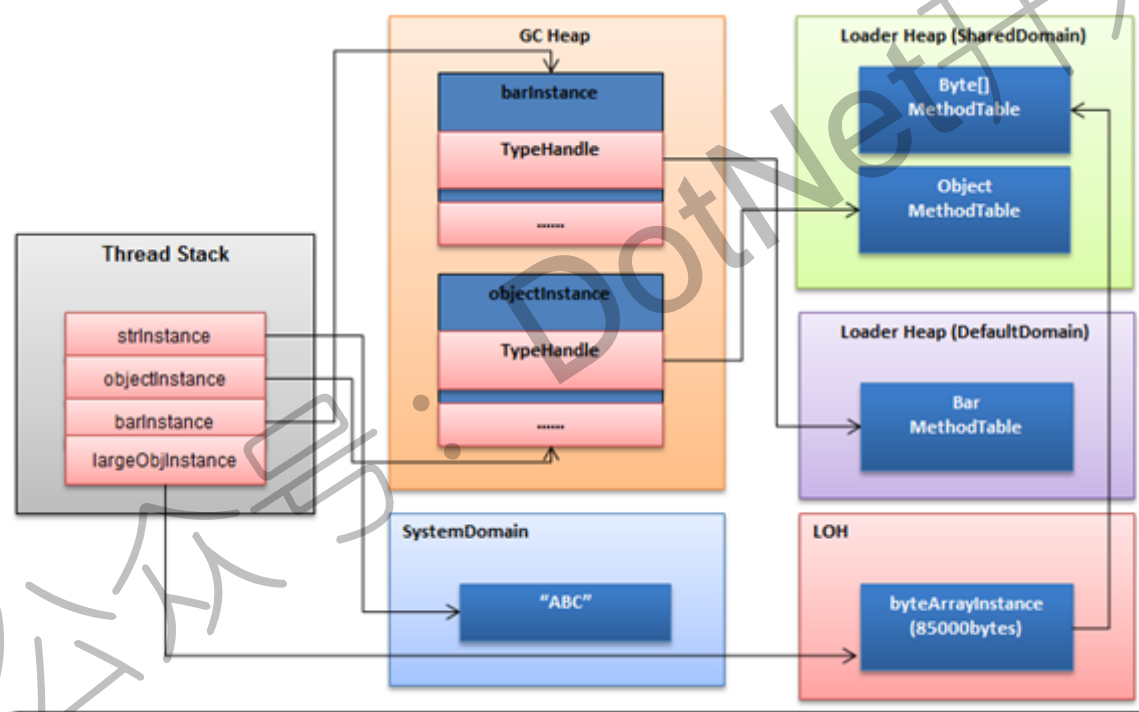
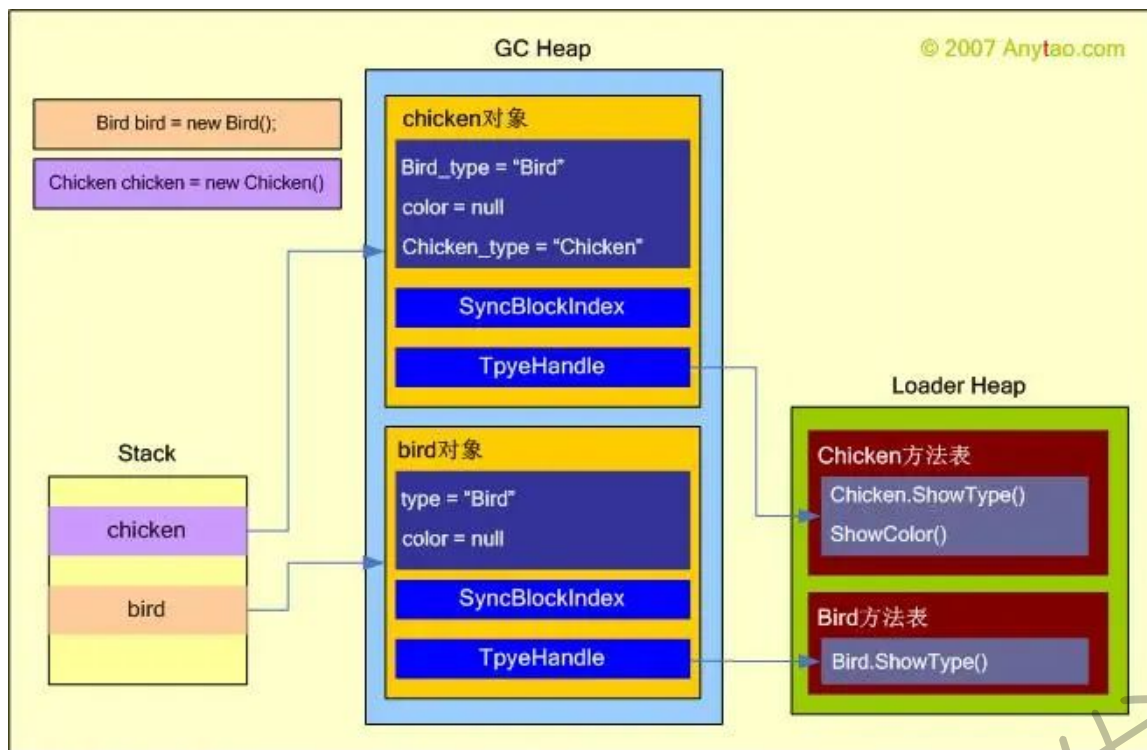
上面获取到的Type对象（System.Int32）就是一个类型对象，她同其他引用类型一样，也是一个引用对象，这个对象中存储了int32类型的所有信息（类型的所有元数据信息）。

### 关于类型对象（Object Type）：

>每一个类型（如System.Int32）在内存中都会有一个唯一的类型对象，通过（int）a.GetType()可以获取该对象；>类型对象（Object Type）存储在内存中一个独立的区域，叫加载堆（Load Heap），加载堆是在进程创建的时候创建的，不受GC垃圾回收管制，因此类型对象一经创建就不会被释放的，他的生命周期从AppDomain创建到结束；>前文说过，每个引用对象都包含两个附加成员：TypeHandle和同步索引块，其中TypeHandle就指向该对象对应的类型对象；>类型对象的加载由class loader负责，在第一次使用前加载；>类型中的静态字段就是存储在这里的（加载堆上的类型对象），所以说静态字段是全局的，而且不会释放；

可以参考下面的图，第一幅图描述了对象在内存中的一个关系，第二幅图更复杂，更准确、全面的描述了内存的结构分布。

图片来源



## 方法表



类型对象内部的主要的结构是怎么样的呢？其中最重要的就是**方法表**，包含了是类型内部的所有方法入口，关于具体的细节和原理这里不多赘述(太多了，可以参考文末给的参考资料)，本文只是初步介绍一下，主要目的是为了解决第6题。

```
1 public class A{
2     public virtual void Print()
3     {
```

```

4  Console.WriteLine("A");
5  }}
6  public class B1 : A{
7      public override void Print()
8      {
9          Console.WriteLine("B1");
10     }
11 }public class B2 : A{
12     public new void Print() {
13         Console.WriteLine("B2");
14     }}

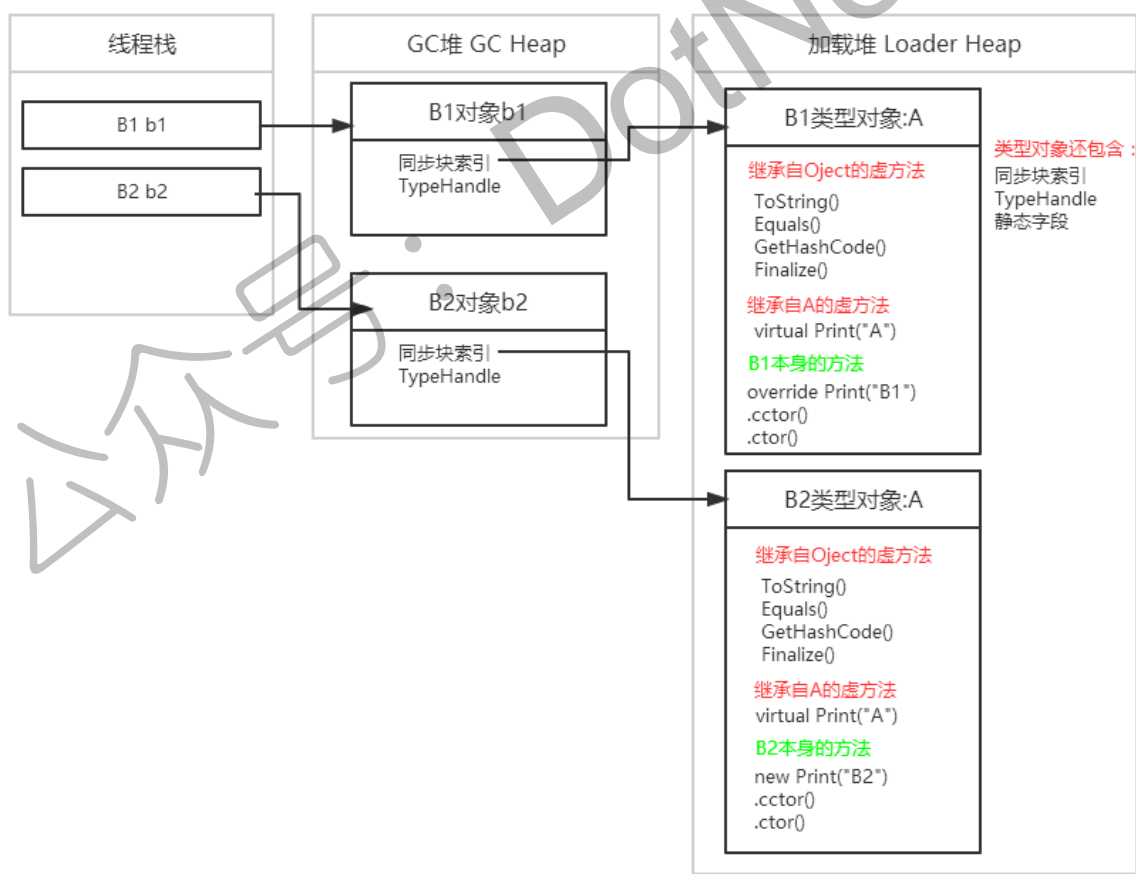
```

还是以第6题的代码为例，上面的代码中，定义两个简单的类，一个基类A，， B1和B2继承自A， 然后使用不同的方式改变了父类方法的行为。当定义了b1、b2两个变量后，内存结构示意图如下：

```

1  B1 b1 = new B1();
2  B2 b2 = new B2();

```



## 方法表的加载:

- 方法表的加载时父类在前子类在后的，首先加载的是固定的4个来自System.Object的虚方法：ToString, Equals, GetHashCode, and Finalize;

- 然后加载父类A的虚方法;
- 加载自己的方法;
- 最后是构造方法: 静态构造函数.cctor(), 对象构造函数.ctor();

方法表中的方法入口(方法表槽)还有很多其他的信息, 比如会关联方法的IL代码以及对应的本地机器码等。其实类型对象本身也是一个引用类型对象, 其内部同样也包含两个附件成员: 同步索引块和类型对象指针TypeHandle, 具体细节、原理有兴趣的可以自己深入了解。

方法的调用: 当执行代码**b1.Print()**时(此处只关注方法调用, 忽略方法的继承等因素), 通过b1的TypeHandle找到对应类型对象, 然后找到方法表槽, 然后是对应的IL代码, 第一次执行的时候, JIT编译器需要把IL代码编译为本地机器码, 第一次执行完成后机器码会保留, 下一次执行就不需要JIT编译了。这也是为什么说.NET程序启动需要预热的原因。

## .NET中的继承本质



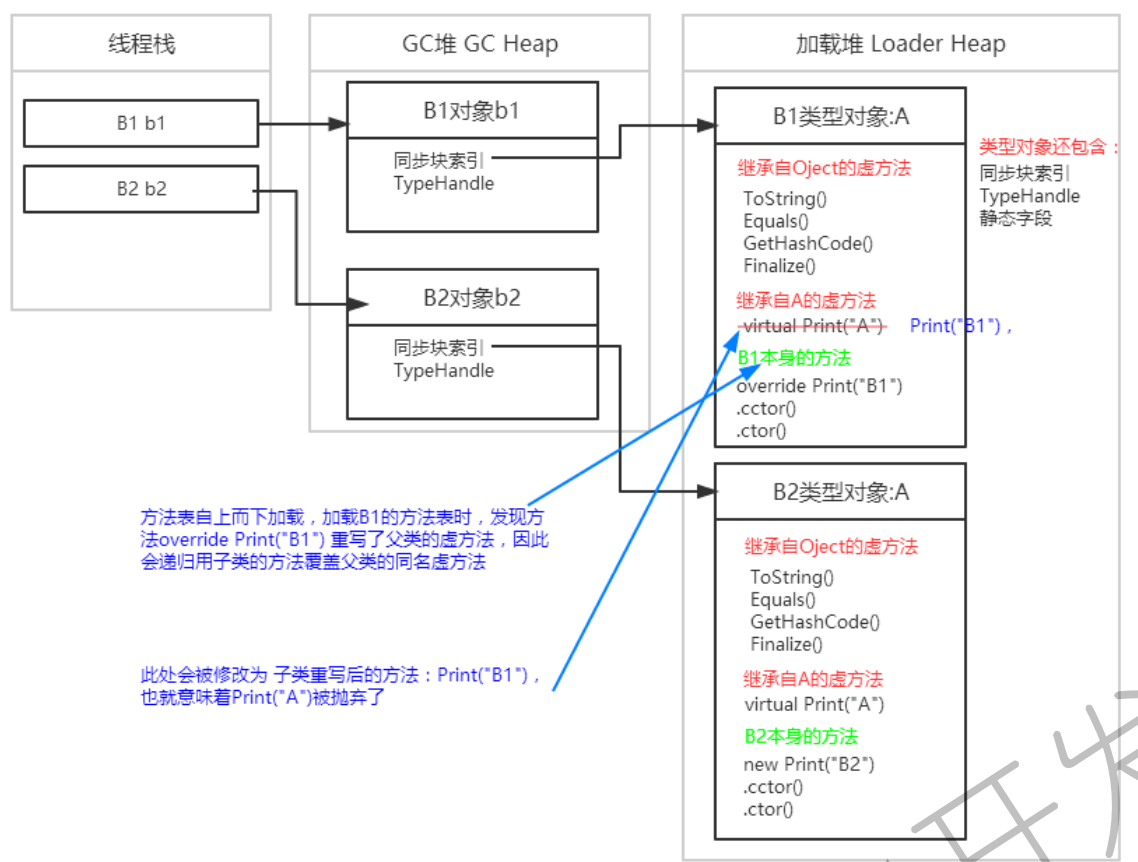
方法表的创建过程是从父类到子类自上而下的, 这是.NET中继承的很好体现, **当发现有覆写父类虚方法会覆盖同名的父方法**, 所有类型的加载都会递归到System.Object类。

- 继承是可传递的, 子类是对父类的扩展, 必须继承父类方法, 同时可以添加新方法。
- 子类可以调用父类方法和字段, 而父类不能调用子类方法和字段。
- 子类不光继承父类的公有成员, 也继承了私有成员, 只是不可直接访问。
- new关键字在虚方法继承中的阻断作用, 中断某一虚方法的继承传递。

因此类型B1、B2的类型对象进一步的结构示意图如下:

- 在加载B1类型对象时, 当加载override B1.Print("B1")时, 发现有覆写override的方法, 会覆盖父类的同名虚方法Print("A"), 就是下面的示意图, 简单来说就是在B1中Print只有一个实现版本;
- 加载B2类型对象时, new关键字表示要隐藏基类的虚方法, 此时B2中的Print("B2")就不是虚方法了, 她是B2中的新方法了, 简单来说就是在B2类型对象中Print有2个实现版本;





```

1 B1 b1 = new B1();
2 B2 b2 = new B2();
3 b1.Print();
4 b2.Print(); //按预期应该输出 B1、B2
5 A ab1 = new B1();
6 A ab2 = new B2();
7 ab1.Print();
8 ab2.Print(); //这里应该输出什么呢?

```

上面代码中红色高亮的两行代码, 用基类 (A) 和用本身B1声明到底有什么区别呢? 类似这种代码在实际编码中是很常见的, 简单的概括一下:

- 无论用什么做引用声明, 哪怕是object, 等号右边的[ = new 类型()]都是没有区别的, 也就说说对象的创建不受影响的, b1和ab1对象在内存结构上是一致的;
- 他们的差别就在引用指针的类型不同, 这种不同在编码中智能提示就直观的反应出来了, 在实际方法调用上也与引用指针类型有直接关系;
- 综合来说, 不同引用指针类型对于对象的创建 (new操作) 不影响; 但对于对象的使用 (如方法调用) 有影响, 这一点在上面代码的执行结果中体现出来了!

上面调用的IL代码:

```

IL_001b: newobj      instance void CLRTest.ConsoleTest.BlogTest/B1::.ctor() A ab1 = new B1();
IL_0020: stloc.2
IL_0021: newobj      instance void CLRTest.ConsoleTest.BlogTest/B2::.ctor() A ab2 = new B2();
IL_0026: stloc.3
IL_0027: ldloc.2
IL_0028: callvirt     instance void CLRTest.ConsoleTest.BlogTest/A::Print()| ab1.Print();
IL_002d: nop
IL_002e: ldloc.3
IL_002f: callvirt     instance void CLRTest.ConsoleTest.BlogTest/A::Print() ab2.Print();

```

对于虚方法的调用，在IL中都是使用指令callvirt，该指令主要意思就是具体的方法在运行时动态确定的：

callvirt使用虚拟调度，也就是根据引用类型的动态类型来调度方法，callvirt指令根据引用变量指向的对象类型来调用方法，在运行时动态绑定，主要用于调用虚方法。

不同的类型指针在虚拟方法表中有不同的附加信息作为标志来区别其访问的地址区域，称为**offset**。不同类型的指针只能在其特定地址区域内进行执行。编译器在方法调用时还有一个原则：

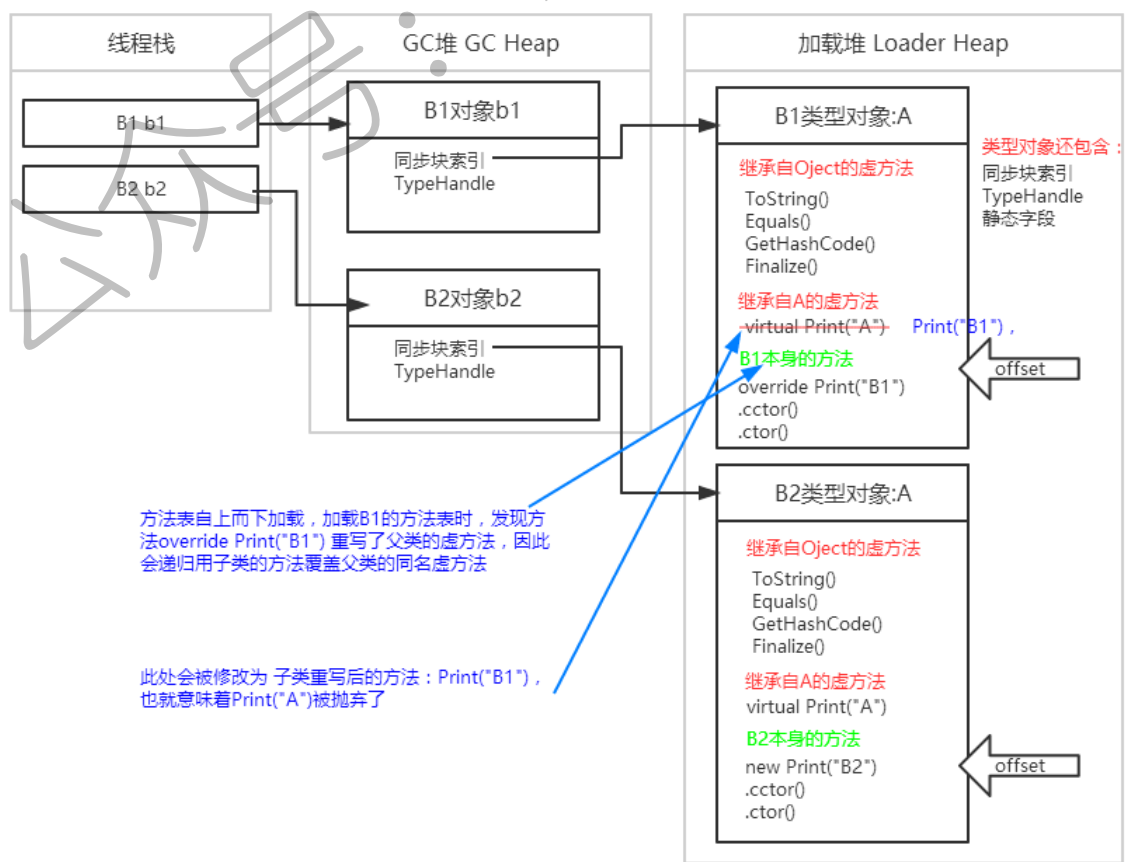
执行就近原则：对于同名字段或者方法，编译器是按照其顺序查找来引用的，也就是首先访问离它创建最近的字段或者方法。

因此执行以下代码时，引用指针类型的offset指向子类，如下图，按照就近查找执行原则，正常输出B1、B2

```

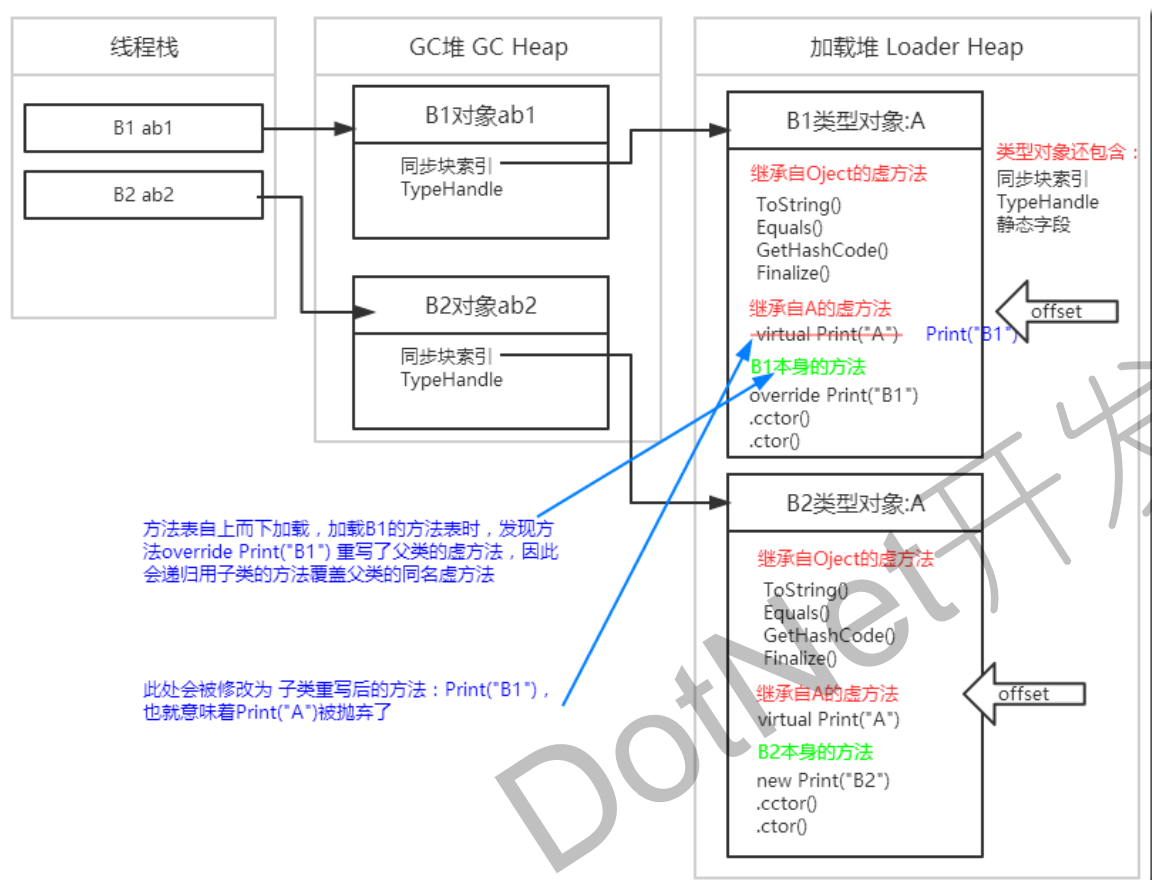
1  B1 b1 = new B1();
2  B2 b2 = new B2();
3  b1.Print(); b2.Print(); //按预期应该输出 B1、B2

```



而当执行以下代码时，引用指针类型都为父类A，引用指针类型的offset指向父类，如下图，按照就近查找执行原则，输出B1、A。

```
1 A ab1 = new B1();
2 A ab2 = new B2(); ab1.Print(); ab2.Print(); //这里应该输出什么呢？
```



## .NET中的继承

### 什么是抽象类



**抽象类提供多个派生类共享基类的公共定义，它既可以提供抽象方法，也可以提供非抽象方法。抽象类不能实例化，必须通过继承由派生类实现其抽象方法，因此对抽象类不能使用new关键字，也不能被密封。**

基本特点：

- 抽象类使用Abstract声明，抽象方法也是用Abstract标示；
- 抽象类不能被实例化；
- 抽象方法必须定义在抽象类中；
- 抽象类可以继承一个抽象类；
- 抽象类不能被密封（不能使用sealed）；
- 同类Class一样，只支持单继承；

一个简单的抽象类代码：

```
1 public abstract class AbstractUser
2 {
3     public int Age { get; set; }
4     public abstract void SetName(string name);
5 }
```

IL代码如下，类和方法都使用abstract修饰：

```
.class abstract auto ansi nested public beforefieldinit AbstractUser
    extends [mscorlib]System.Object
{
} // end of class AbstractUser

AbstractUser::SetName : void(string)
查找(F)  查找下一个(N)
.method public hidebysig newslot abstract virtual
    instance void SetName(string name) cil managed
{
} // end of method AbstractUser::SetName
```

## 什么是接口？



**接口简单理解就是一种规范、契约，使得实现接口的类或结构在形式上保持一致。实现接口的类或结构必须实现接口定义中所有接口成员，以及该接口从其他接口中继承的所有接口成员。**

基本特点：

- 接口使用interface声明；
- 接口类似于抽象基类，不能直接实例化接口；
- 接口中的方法都是抽象方法，不能有实现代码，实现接口的任何非抽象类型都必须实现接口的所有成员；
- 接口成员是自动公开的，且不能包含任何访问修饰符。
- 接口自身可从多个接口继承，类和结构可继承多个接口，但接口不能继承类。

下面一个简单的接口定义：

```
1 public interface IUser{
2     int Age { get; set; }
3     void SetName(string name);
4 }
```

下面是IUser接口定义的IL代码，看上去是不是和上面的抽象类AbstractUser的IL代码差不多！接口也是使用.Class ~ abstract标记，方法定义同抽象类中的方法一样使用abstract virtual标记。因此可以把

接口看做是一种特殊的抽象类，该类只提供定义，没有实现。

```
.class interface abstract auto ansi nested public IUser
{
} // end of class IUser

IUser::SetName : void(string)
查找(F) 查找下一个(N)

.method public hidebysig newslot abstract virtual
instance void SetName(string name) cil managed
{
} // end of method IUser::SetName

void SetName(string name);
```

另外一个小细节，上面说到接口是一个特殊的类型，不继承`System.Object`，通过IL代码其实可以证实这一点。无论是自定义的任何类型还是抽象类，都会隐式继承`System.Object`，`AbstractUser`的IL代码中就有“`extends [mscorlib]System.Object`”，而接口的IL代码并没有这一段代码。

## 关于继承



关于继承，太概念性了，就不细说了，主要还是在平时的搬砖过程中多思考、多总结、多体会。

在.NET中继承的主要两种方式就是**类继承和接口继承**，两者的主要思想是不一样的：

- 类继承强调父子关系，是一个“IS A”的关系，因此只能单继承（就像一个人只能有一个Father）；
- 接口继承强调的是一种规范、约束，是一个“CAN DO”的关系，支持多继承，是实现多态一种重要方式。

更准确的说，类可以叫继承，接口叫“实现”更合适。更多的概念和区别，可以直接看后面的答案，更多的还是要自己理解。

## 题目答案解析:

### 1. 所有类型都继承System.Object吗？

基本上是的，所有值类型和引用类型都继承自`System.Object`，接口是一个特殊的类型，不继承自`System.Object`。

### 2. 解释virtual、sealed、override和abstract的区别

- virtual申明虚方法的关键字，说明该方法可以被重写
- sealed说明该类不可被继承
- override重写基类的方法
- abstract申明抽象类和抽象方法的关键字，抽象方法不提供实现，由子类实现，抽象类不可实例化。

### 3. 接口和类有什么异同？

不同点：

- 1、接口不能直接实例化。
- 2、接口只包含方法或属性的**声明**，不包含方法的实现。
- 3、接口可以**多继承**，类只能单继承。
- 4、类有**分部类**的概念，定义可在不同的源文件之间进行拆分，而接口没有。（这个地方确实不对，接口也可以分部，谢谢@xclin163的指正）
- 5、表达的含义不同，接口主要定义一种**规范**，统一调用方法，也就是规范类，约束类，类是方法功能的实现和集合

#### 相同点：

- 1、接口、类和结构都可以从多个接口继承。
- 2、接口类似于抽象基类：继承接口的任何非抽象类型都必须实现接口的所有成员。
- 3、接口和类都可以包含事件、索引器、方法和属性。

## 4. 抽象类和接口有什么区别？

- 1、继承：接口支持多继承；抽象类不能实现多继承。
- 2、表达的概念：接口用于规范，更强调契约，抽象类用于共性，强调父子。抽象类是一类事物的高度聚合，那么对于继承抽象类的子类来说，对于抽象类来说，属于"Is A"的关系；而接口是定义行为规范，强调"Can Do"的关系，因此对于实现接口的子类来说，相对于接口来说，是"行为需要按照接口来完成"。
- 3、方法实现：对抽象类中的方法，即可以给出实现部分，也可以不给出；而接口的方法（抽象规则）都不能给出实现部分，接口中方法不能加修饰符。
- 4、子类重写：继承类对于两者所涉及方法的实现是不同的。继承类对于抽象类所定义的抽象方法，可以不用重写，也就是说，可以沿用抽象类的方法；而对于接口类所定义的方法或者属性来说，在继承类中必须重写，给出相应的方法和属性实现。
- 5、新增方法的影响：在抽象类中，新增一个方法的话，继承类中可以不用作任何处理；而对于接口来说，则需要修改继承类，提供新定义的方法。
- 6、接口可以作用于值类型（枚举可以实现接口）和引用类型；抽象类只能作用于引用类型。
- 7、接口不能包含字段和已实现的方法，接口只包含方法、属性、索引器、事件的签名；抽象类可以定义字段、属性、包含有实现的方法。

## 5. 重载与覆盖的区别？

重载：当类包含两个名称相同但签名不同(方法名相同,参数列表不相同)的方法时发生方法重载。用方法重载来提供在语义上完成相同而功能不同的方法。

覆写：在类的继承中使用，通过覆写子类方法可以改变父类虚方法的实现。

#### 主要区别：

- 1、方法的覆盖是子类和父类之间的关系，是垂直关系；方法的重载是同一个类中方法之间的关系，是水平关系。
- 2、覆盖只能由一个方法，或只能由一对方法产生关系；方法的重载是多个方法之间的关系。



3、覆盖要求参数列表相同；重载要求参数列表不同。

4、覆盖关系中，调用那个方法体，是根据对象的类型来决定；重载关系，是根据调用时的实参表与形参表来选择方法体的。

**6. 在继承中new和override相同点和区别？看下面的代码，有一个基类A，B1和B2都继承自A，并且使用不同的方式改变了父类方法Print () 的行为。测试代码输出什么？为什么？**

```
1 public void DoTest(){
2     B1 b1 = new B1();
3     B2 b2 = new B2();
4     b1.Print();
5     b2.Print();
6     //按预期应该输出 B1、B2
7     A ab1 = new B1();
8     A ab2 = new B2();
9     ab1.Print(); ab2.Print();
10    //这里应该输出什么呢？输出B1、A
11    }
12    public class A{
13        public virtual void Print()
14        { Console.WriteLine("A"); }
15        public class B1 : A{
16            public override void Print()
17            { Console.WriteLine("B1"); }}
18        public class B2 : A{
19            public new void Print() {
20                Console.WriteLine("B2"); }}
```

**7. 下面代码中，变量a、b都是int类型，代码输出结果是什么？**

```
1 int a = 123;
2 int b = 20;
3 var atype = a.GetType();
4 var btype = b.GetType();
5 Console.WriteLine(System.Object.Equals(atype,btype));
6 //输出True
```

```
7 Console.WriteLine(System.Object.ReferenceEquals(atype,btype));
8 //输出True
```

## 8.class中定义的静态字段是存储在内存中的哪个地方？为什么会说她不会被GC回收？

随类型对象存储在内存的加载堆上，因为加载堆不受GC管理，其生命周期随AppDomain，不会被GC回收。

关注微信公众号：DotNet开发跳槽

## 五、常量、字段、属性、特性与委托

### 常见面试题目：

1. const和readonly有什么区别？
2. 哪些类型可以定义为常量？常量const有什么风险？
3. 字段与属性有什么异同？
4. 静态成员和非静态成员的区别？
5. 自动属性有什么风险？
6. 特性是什么？如何使用？
7. 下面的代码输出什么结果？为什么？

```
1
2 List<Action> acs = new List<Action>(5);
3 for (int i = 0; i < 5; i++)
4 {
5     acs.Add(() => { Console.WriteLine(i); });
6 }
7 acs.ForEach(ac => ac());
```

8. C#中的委托是什么？事件是不是一种委托？

## 字段与属性的恩怨

### 常量



常量的基本概念就不细说了，关于常量的几个特点总结一下：

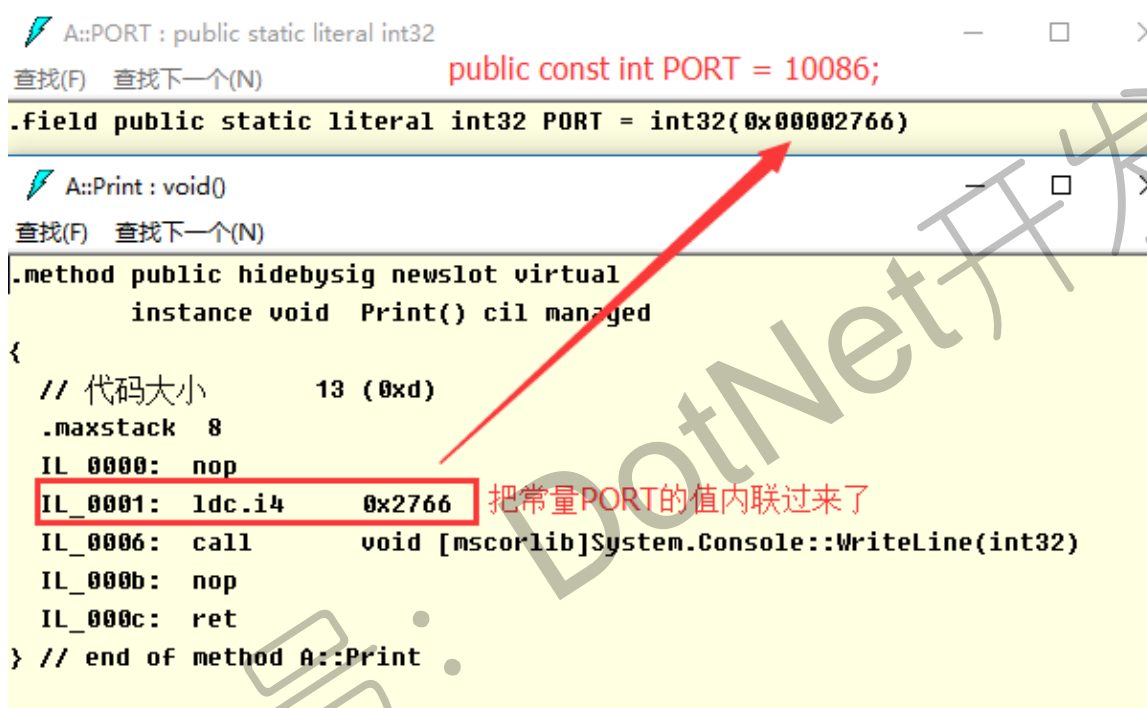
- 常量的值必须在编译时确定，简单说就是在定义时设置值，以后都不会被改变了，她是编译常量。
- 常量只能用于简单的类型，因为常量值是要被编译然后保存到程序集的元数据中，只支持基元类型，如 int、char、string、bool、double等。

- 常量在使用时，是把常量的值内联到IL代码中的，常量类似一个占位符，在编译时被替换掉了。正是这个特点导致常量的一个风险，就是不支持跨程序集版本更新；

关于常量不支持跨程序集版本更新，举个简单的例子来说明：

上面一段非常简单代码，其生产的IL代码如下，在使用常量变量的地方，把她的值拷过来了（把常量的值内联到使用的地方），与常量变量A.PORT没有关系了。假如A引用了B程序集（B.dll文件）中的一个常量，如果后面单独修改B程序集中的常量值，只是重新编译了B，而没有编译程序集A，就会出问题了，就是上面所说的不支持跨程序集版本更新。常量值更新后，所有使用该常量的代码都必须重新编译，这是我们在使用常量时必须要注意的一个问题。

- 不要随意使用常量，特别是有可能变化的数据；
- 不要随便修改已定义好的常量值；



```
A::PORT : public static literal int32
public const int PORT = 10086;
.field public static literal int32 PORT = int32(0x00002766)

A::Print : void()
.method public hidebysig newslot virtual
    instance void Print() cil managed
{
    // 代码大小      13 (0xd)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldc.i4      0x2766 把常量PORT的值内联过来了
    IL_0006: call        void [mscorlib]System.Console::WriteLine(int32)
    IL_000b: nop
    IL_000c: ret
} // end of method A::Print
```

## 补充一下枚举的本质

😁 接着上面的const说，其实枚举enum也有类似的问题，其根源和const一样，看看代码你就明白了。下面的是一个简单的枚举定义，她的IL代码定义和const定义是一样一样的啊！枚举的成员定义和常量定义一样，因此枚举其实本质上就相当是一个常量集合。

```
1 public enum EnumType : int{    None=0,    Int=1,    String=2,}
```

|  |  |
|--|--|
| Field public static literal              | valuetype CLRTest.ConsoleTest.BlogTest/EnumType None = int32(0x00000000)   |
| EnumType::Int : public static literal    | valuetype CLRTest.ConsoleTest.BlogTest/EnumType                            |
| Field public static literal              | valuetype CLRTest.ConsoleTest.BlogTest/EnumType Int = int32(0x00000001)    |
| EnumType::String : public static literal | valuetype CLRTest.ConsoleTest.BlogTest/EnumType                            |
| Field public static literal              | valuetype CLRTest.ConsoleTest.BlogTest/EnumType String = int32(0x00000002) |

## 关于字段



字段本身没什么好说的，这里说一个字段的内联初始化问题吧，可能容易被忽视的一个小问题（不过好像也没什么影响），先看看一个简单的例子：

```
1 public class SomeType{
2     private int Age = 0;
3     private DateTime StartTime = DateTime.Now;
4     private string Name = "三体";}
```

定义字段并初始化值，是一种很常见的代码编写习惯。但注意了，看看IL代码结构，一行代码（定义字段+赋值）被拆成了两块，最终的赋值都在构造函数里执行的。

|  |  |
|--|--|
| .field private valuetype [mscorlib]System.DateTime StartTime                         | int32(0x00000001)  |
| 定义字段：private DateTime StartTime  |  |
| SomeType::ctor : void()  |  |
| .method public hidebysig specialname rtspecialname instance void .ctor() cil managed |  |
| {  |  |
| // 代码大小 37 (0x25)  |  |
| .maxstack 8  |  |
| IL_0000: ldarg.0   | 在构造函数里设置字段值：StartTime=Datetime.Now;  |
| IL_0001: ldc.i4.0  |  |
| IL_0002: stfld   | int32 CLRTest.ConsoleTest.BlogTest/SomeType::Age                                     |
| IL_0007: ldarg.0   |  |
| IL_0008: call  | valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()             |
| IL_000d: stfld   | valuetype [mscorlib]System.DateTime CLRTest.ConsoleTest.BlogTest/SomeType::StartTime |
| IL_0012: ldarg.0   |  |
| IL_0013: ldstr   | bytearray {09 4E 53 4F } // .NS0   |
| IL_0018: stfld   | string CLRTest.ConsoleTest.BlogTest/SomeType::Name                                   |
| IL_001d: ldarg.0   |  |
| IL_001e: call  | instance void [mscorlib]System.Object::ctor()  |
| IL_0023: nop   |  |
| IL_0024: ret   |  |
| } // end of method SomeType::ctor  |  |

那么问题来了，如果有多个构造函数，就像下面这样，**有多半个构造函数，会造成在两个构造函数.ctor中重复产生对字段赋值的IL代码，这就造成了不必要的代码膨胀。**这个其实也很好解决，在非默认构造函数后加一个“**.this()**”就OK了，或者显示得在构造函数里初始化字段。

```
1 public class SomeType{
2     private DateTime StartTime = DateTime.Now;
```

```

3     public SomeType() { }
4     public SomeType(string name)
5     {
6
    }
}

```

## 属性的本质

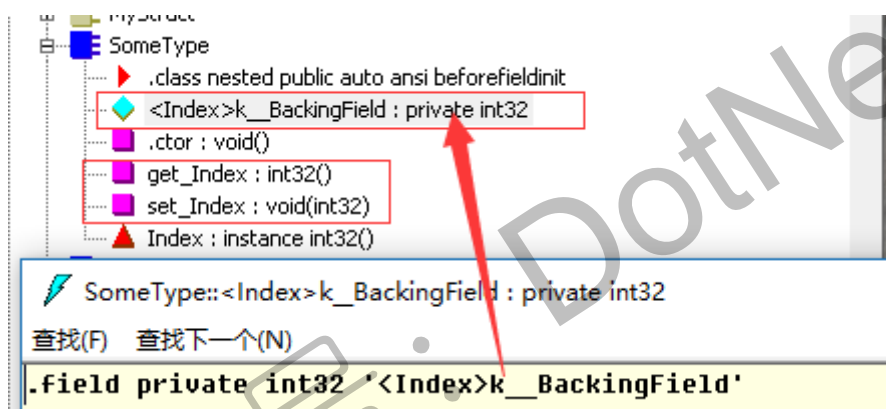


属性是面向对象编程的基本概念，提供了对私有字段的访问封装，在C#中以get和set访问器方法实现对可读可写属性的操作，提供了安全和灵活的数据访问封装。我们看看属性的本质，主要手段还是IL代码：

```

1 public class SomeType{
2     public int Index { get; set; }
3     public SomeType() {
4
    }
}

```



上面定义的属性Index被分成了三个部分：

- 自动生成的私有字段“<Index>k\_\_BackingField”
- 方法：get\_Index(), 获取字段值；
- 方法：set\_Index(int32 'value'), 设置字段值；

因此可以说属性的本质还是方法，使用面向对象的思想把字段封装了一下。在定义属性时，我们可以自定义一个私有字段，也可以使用**自动属性** “{ get; set; }” 的简化语法形式。

使用自动属性时需要注意一点的是，私有字段是由编译器自动命名的，是不受开发人员控制的。正因为这个问题，曾经在项目开发中遇到一个因此而产生的Bug：

这个Bug是关于序列化的，有一个类，定义很多个（自动）属性，这个类的信息需要持久化到本地文件，当时使用了.NET自带的二进制序列化组件。后来因为一个需求变更，把其中一个字段修改了一下，需要把自动属性改为自己命名的私有字段的属性，就像下面实例这样。测试序列化到本地没有问题，反序列化也没问题，但最终bug还是被测试出来了，问题在反序列化以前（修改代码之前）的本地文件时，Index属性的值丢失了！！

```

1 private int _Index;

```

```

2 public int Index{
3     get { return _Index; }
4     set { _Index = value; }
5 }

```

因为属性的本质是方法+字段，真正的值是存储在字段上的，字段的名称变了，反序列化以前的文件时找不到对应字段了，导致值的丢失！这也就是使用自动属性可能存在的风险。

## 委托与事件

什么是委托？简单来说，委托类似于 C 或 C++ 中的函数指针，允许将方法作为参数进行传递。

- C# 中的委托都继承自 `System.Delegate` 类型；
- 委托类型的声明与方法签名类似，有返回值和参数；
- 委托是一种可以封装命名（或匿名）方法的引用类型，把方法当做指针传递，但委托是面向对象、类型安全的；

## 委托的本质——是一个类

.NET 中没有函数指针，方法也不可能传递，委托之所可以像一个普通引用类型一样传递，那是因为她本质上就是一个类。下面代码是一个非常简单的自定义委托：

```

1 public delegate void ShowMessageHandler(string mes);

```

看看她生产的 IL 代码



我们一行定义一个委托的代码，编译器自动生成了一堆代码：

- 编译器自动帮我们创建了一个类 `ShowMessageHandler`，继承自 `System.MulticastDelegate`（她又继承自 `System.Delegate`），这是一个多播委托；
- 委托类 `ShowMessageHandler` 中包含几个方法，其中最重要的就是 `Invoke` 方法，签名和定义的方法签名一致；
- 其他两个版本 `BeginInvoke` 和 `EndInvoke` 是异步执行版本；

因此，也就不难猜测，当我们调用委托的时候，其实就是调用委托对象的 `Invoke` 方法，可以验证一下，下面的调用代码会被编译为对委托对象的 `Invoke` 方法调用：



```
1 private ShowMessageHandler ShowMessage;  
2 //调用  
3 this.ShowMessage("123");
```

```
IL_000e: ldstr      "123"  
IL_0013: callvirt instance void CLRTest.ConsoleTest.BlogTest/SomeType/ShowMessageHandler::Invoke(string)  
IL_0018: nop
```

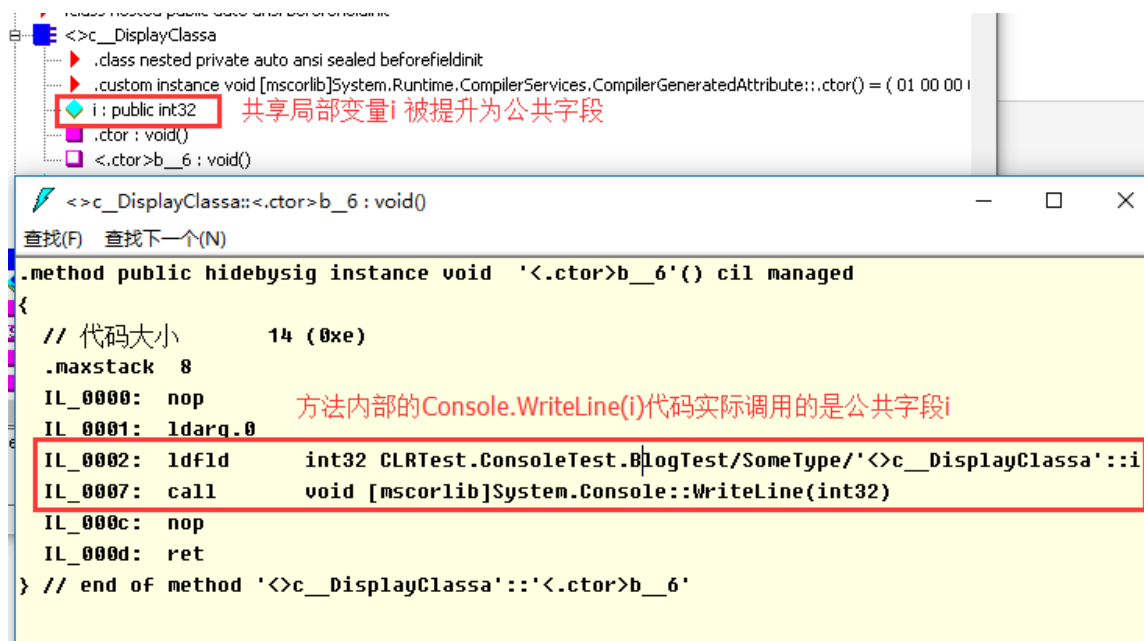
## .NET的闭包



闭包提供了一种类似脚本语言函数式编程的便捷、可以共享数据，但也存在一些隐患。题目列表中的第7题，就是一个.NET的闭包的问题。

```
1  
2  
3  
4 List<Action> acs = new List<Action>(5);  
5 for (int i = 0; i < 5; i++)  
6 {  
7     acs.Add(() => { Console.WriteLine(i); });  
8 }  
9 acs.ForEach(ac => ac());  
10 // 输出了 5 5 5 5 5，全是5？这一定不是你想要的吧！这是为什么呢？
```

上面的代码中的Action就是.NET为我们定义好的一个无参数无返回值的委托，从上一节我们知道委托实质是一个类，理解这一点是解决本题的关键。在这个地方委托方法共享使用了一个局部变量i，那生成的类会是什么样的呢？看看IL代码：



共享的局部变量被提升为委托类的一个字段了：

- 变量i的生命周期延长了；
- for循环结束后字段的值是5了；
- 后面再次调用委托方法，肯定就是输出5了；

那该如何修正呢？很简单，委托方法使用一个临时局部变量就OK了，不共享数据：

```
1
2 List<Action> acss = new List<Action>(5);
3 for (int i = 0; i < 5; i++)
4 {
5     int m = i;
6     acss.Add(() => { Console.WriteLine(m); });
7 }
8 acss.ForEach(ac => ac()); // 输出了 0 1 2 3 4
```

至于原理，可以自己探索了！

## 题目答案解析：

### 1. const和readonly有什么区别？

const关键字用来声明编译时常量，readonly用来声明运行时常量。都可以标识一个常量，主要有以下区别：1、初始化位置不同。const必须在声明的同时赋值；readonly即可以在声明处赋值，也可以在构造方法里赋值。2、修饰对象不同。const即可以修饰类的字段，也可以修饰局部变量；readonly只能修饰类的字段。

3、const是编译时常量，在编译时确定该值，且值在编译时被内联到代码中；readonly是运行时常量，在运行时确定该值。

4、const默认是静态的；而readonly如果设置成静态需要显示声明。

5、支持的类型是不同，const只能修饰基元类型或值为null的其他引用类型；readonly可以是任何类型。

## 2. 哪些类型可以定义为常量？常量const有什么风险？

基元类型或值为null的其他引用类型，常量的风险就是不支持跨程序集版本更新，常量值更新后，所有使用该常量的代码都必须重新编译。

## 3. 字段与属性有什么异同？

- 属性提供了更为强大的，灵活的功能来操作字段
- 出于面向对象的封装性，字段一般不设计为Public
- 属性允许在set和get中编写代码
- 属性允许控制set和get的可访问性，从而提供只读或者可读写的功能（逻辑上只写是没有意义的）
- 属性可以使用override 和 new

## 4. 静态成员和非静态成员的区别？

- 静态变量使用 static 修饰符进行声明，静态成员在加类的时候就被加载（上一篇中提到过，静态字段是随类型对象存放在Load Heap上的），通过类进行访问。
- 不带有static 修饰符声明的变量称做非静态变量，在对象被实例化时创建，通过对象进行访问。
- 一个类的所有实例的同一静态变量都是同一个值，同一个类的不同实例的同一非静态变量可以是不同的值。
- 静态函数的实现里不能使用非静态成员，如非静态变量、非静态函数等。

## 5. 自动属性有什么风险？

因为自动属性的私有字段是由编译器命名的，后期不宜随意修改，比如在序列化中会导致字段值丢失。

## 6. 特性是什么？如何使用？

特性与属性是完全不相同的两个概念，只是在名称上比较相近。Attribute特性就是关联了一个目标对象的一段配置信息，本质上是一个类，其为目标元素提供关联附加信息，这段附加信息存储在dll内的元数据，它本身没什么意义。运行期以反射的方式来获取附加信息。使用方法可以参考：<http://www.cnblogs.com/anding/p/5129178.html>

## 7. 下面的代码输出什么结果？为什么？

```
1 List<Action> acs = new List<Action>(5);
2 for (int i = 0; i < 5; i++)
3 {
```

```
4     acs.Add(() => { Console.WriteLine(i); });  
5 }  
6 acs.ForEach(ac => ac());
```

输出了 5 5 5 5 5，全是5！因为闭包中的共享变量*i*会被提升为委托对象的公共字段，生命周期延长了

## 8. C#中的委托是什么？事件是不是一种委托？

什么是委托？简单来说，委托类似于 C或 C++中的函数指针，允许将方法作为参数进行传递。

- C#中的委托都继承自System.Delegate类型；
  - 委托类型的声明与方法签名类似，有返回值和参数；
  - 委托是一种可以封装命名（或匿名）方法的引用类型，把方法当做指针传递，但委托是面向对象、类型安全的；
- 事件可以理解为一种特殊的委托，事件内部是基于委托来实现的。

## 六、GC与内存管理

### 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？

```
1 public class User{  
2     public int Age { get; set; }  
3     public string Name { get; set; }  
4     public string _Name = "123" + "abc";  
5     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在哪些情况下会进行回收工作？
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漏的现象？
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

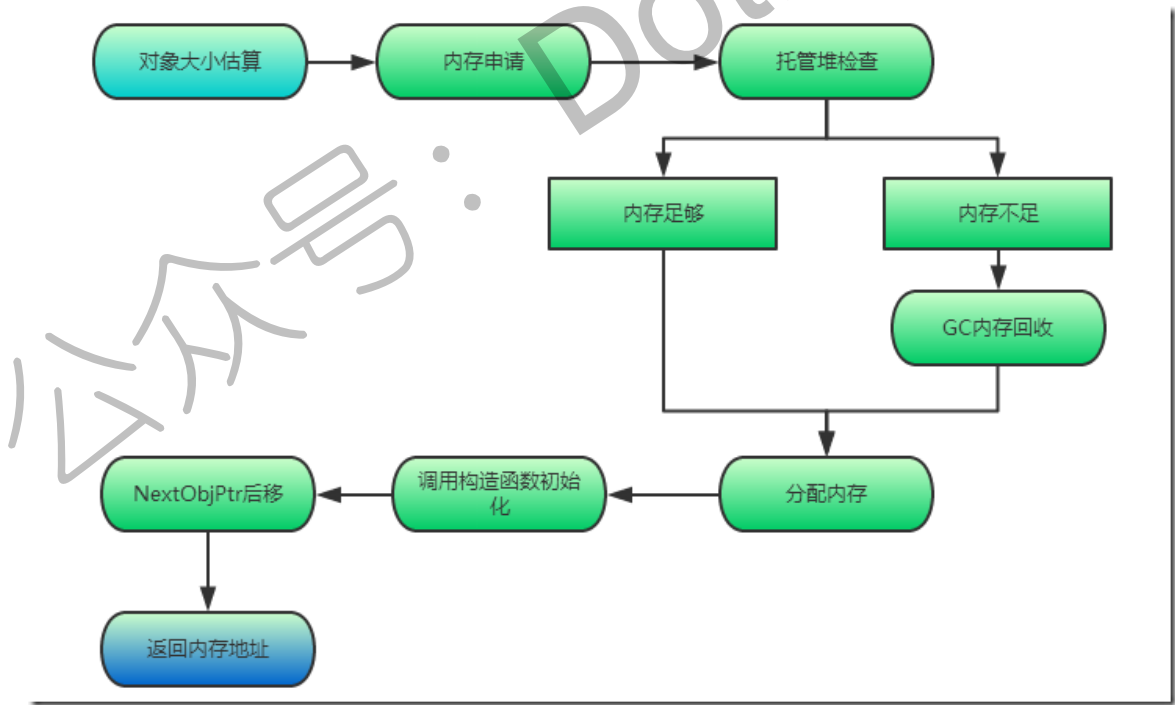


一个对象的生命周期简单概括就是：创建>使用>释放，在.NET中一个对象的生命周期：

- 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专题面试题：string与字符串操作含解析（精）](#)），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：

```
1 public class User{
2     public int Age { get; set; }
3     public string Name { get; set; }
4     public string _Name = "123" + "abc";
```

```
5     public List<string> _Names;
6 }
```

- **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专题面试题：string与字符串操作含解析（精）](#)）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44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44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44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）；



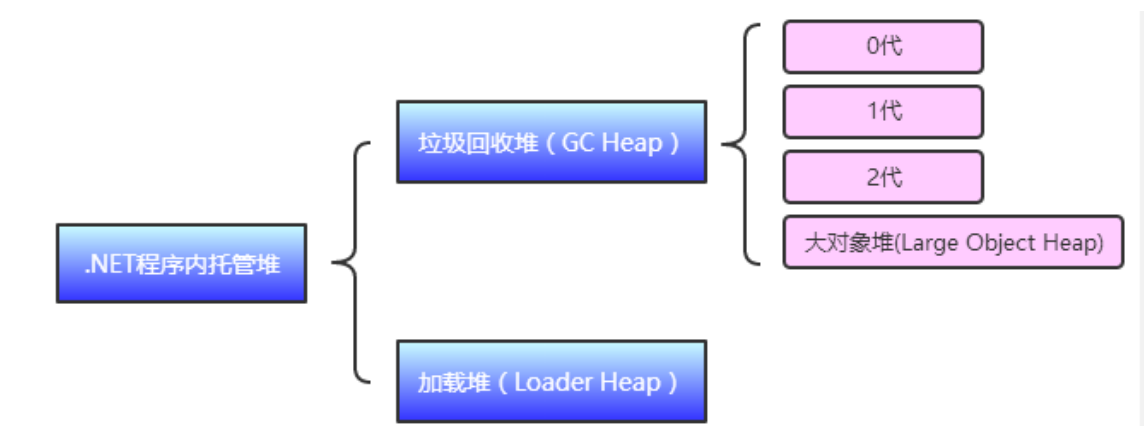


图3 (Figure-3)

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；Foreachable队列等。

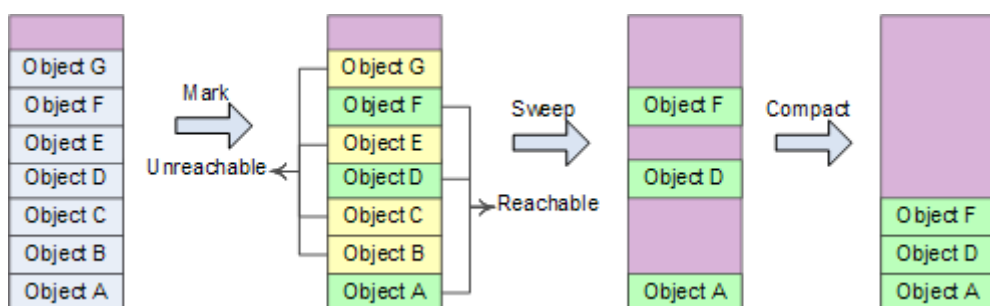
### ② 清除

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)

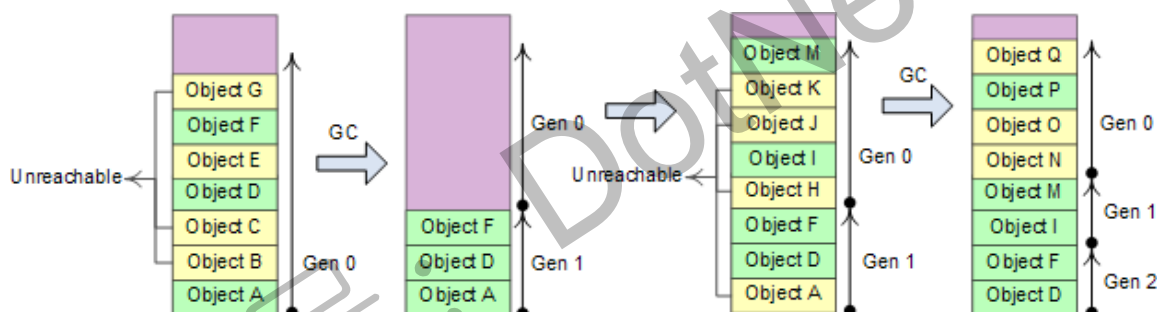


当然，实际的垃圾回收过程可能比上面的要复杂，如果每次都扫描托管堆内的所有对象实例，这样做太耗费时间而且没有必要。**分代(Generation)算法是CLR垃圾回收器采用的一种机制，它唯一的目的是提升应用程序的性能。**分代回收，速度显然快于回收整个堆。分代(Generation)算法的假设前提条件：

- 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。

```
1 public class SomeType : IDisposable{
2     public MemoryStream _MemoryStream;
3     public void Dispose() {
```

```
4     if (_MemoryStream != null) _MemoryStream.Dispose();
5 }
```

Dispose需要手动调用，在.NET中有两种调用方式：

```
1
2 //方式1: 显示接口调用
3 SomeType st1=new SomeType();
4 //do sth
5 st1.Dispose();
6
7 //方式2: using()语法调用，自动执行Dispose接口
8 using (var st2 = new SomeType())
9 {
10     //do sth
11 }
```

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Foreachable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Foreachable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？

```
1 public class User{
2     public int Age { get; set; }
3     public string Name { get; set; }
4     public string _Name = "123" + "abc";
5     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）
- Windows报告内存不足时，CLR会强制执行垃圾回收
- CLR卸载AppDomain，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IDisposable。

另外一个重点区别就是终结器会导致对象复活一次，也就是说会被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：

- 不正确地使用静态字段，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;

## 七、多线程编程与线程同步

### 常见面试题:

1. 描述线程与进程的区别?
2. 为什么GUI不支持跨线程访问控件? 一般如何解决这个问题?
3. 简述后台线程和前台线程的区别?
4. 说说常用的锁, lock是一种什么样的锁?
5. lock为什么要锁定一个参数, 可不可以锁定一个值类型? 这个参数有什么要求?
6. 多线程和异步有什么关系和区别?
7. 线程池的优点有哪些? 又有哪些不足?
8. Mutex和lock有何不同? 一般用哪一个作为锁使用更好?
9. 下面的代码, 调用方法DeadLockTest (20) , 是否会引起死锁? 并说明理由。

```
1 public void DeadLockTest(int i){
2     lock (this)    //或者lock一个静态object变量
3     {
4         if (i > 10)
5         {
6             Console.WriteLine(i--);
7             DeadLockTest(i);
8         }
9     }
}
```

10. 用双检锁实现一个单例模式Singleton。
11. 下面代码输出结果是什么? 为什么? 如何改进她?

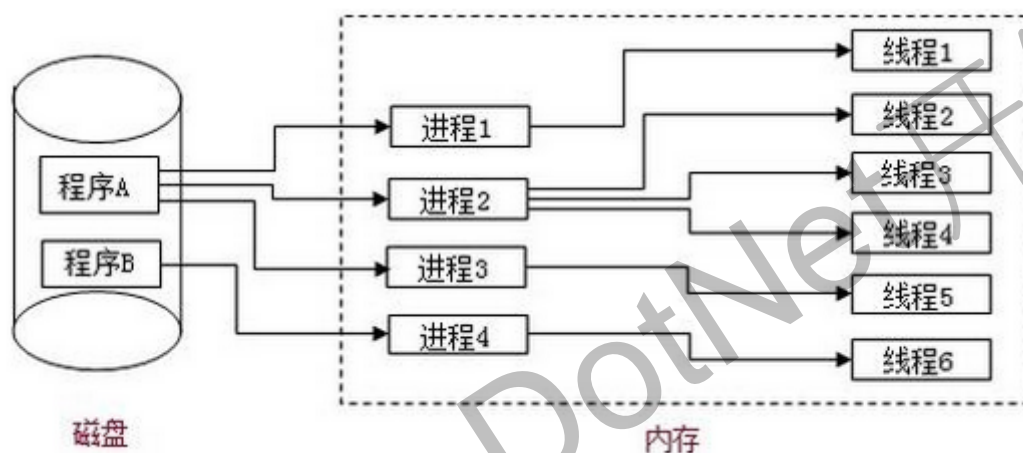
```
1
2 int a = 0;
```

```
3 System.Threading.Tasks.Parallel.For(0, 100000, (i) =>
4 {
5     a++;
6 });
7 Console.Write(a);
```

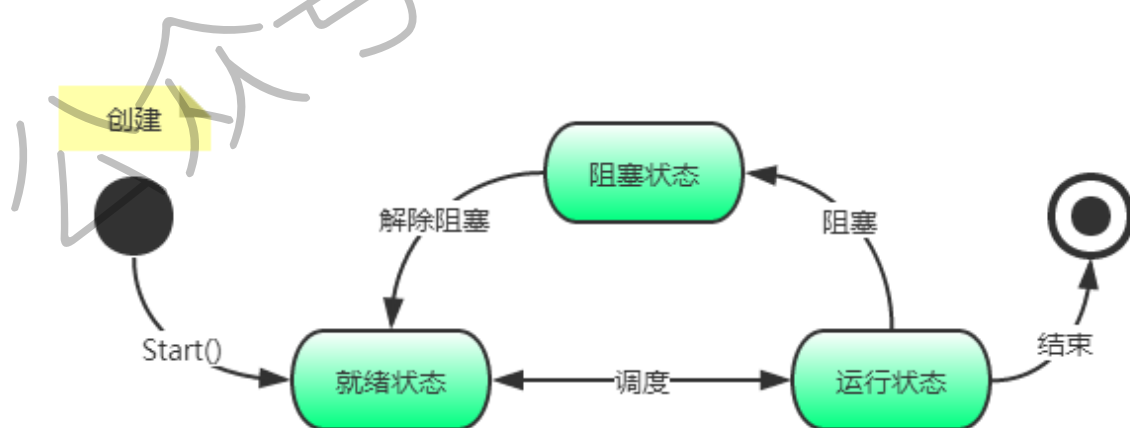
## 线程基础

### 进程与线程

我们运行一个exe，就是一个进程实例，系统中有很多个进程。每一个进程都有自己的内存地址空间，每个进程相当于一个独立的边界，有自己的独占的资源，进程之间不能共享代码和数据空间。



每一个进程有一个或多个线程，进程内多个线程可以共享所属进程的资源 and 数据，**线程是操作系统调度的基本单元**。线程是由操作系统来调度和执行的，她的基本状态如下图。



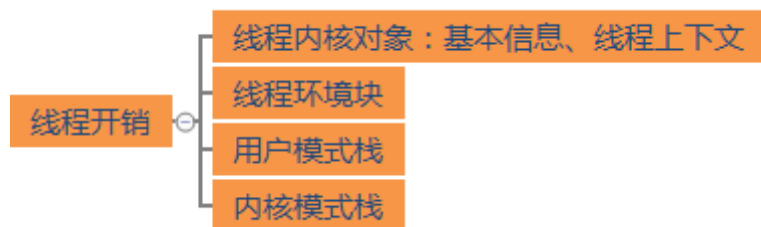
### 线程的开销及调度



当我们创建了一个线程后，线程里面到底有些什么东西呢？主要包括线程内核对象、线程环境块、1M大小的用户模式栈、内核模式栈。其中用户模式栈对于普通的系统线程那1M是预留的，在需要的时候

才会分配，但是对于CLR线程，那1M是一开始就分类了内存空间的。

补充一句，CLR线程是直接对应于一个Windows线程的。



还记得以前学校里学习计算机课程里讲到，计算机的核心计算资源就是CPU核心和CPU寄存器，这也就是线程运行的主要战场。操作系统中那么多线程（一般都有上千个线程，大部分都处于休眠状态），对于单核CPU，一次只能有一个线程被调度执行，那么多线程怎么分配的呢？Windows系统采用时间轮询机制，CPU计算资源以时间片(大约30ms)的形式分配给执行线程。

计算鸡资源（CPU核心和CPU寄存器）一次只能调度一个线程，具体的调度流程：

- 把CPU寄存器内的数据保存到当前线程内部（线程上下文等地方），给下一个线程腾地方；
- 线程调度：在线程集合里取出一个需要执行的线程；
- 加载新线程的上下文数据到CPU寄存器；
- 新线程执行，享受她自己的CPU时间片（大约30ms），完了之后继续回到第一步，继续轮回；

上面线程调度的过程，就是一次线程切换，一次切换就涉及到线程上下文等数据的搬入搬出，性能开销是很大的。因此线程不可滥用，线程的创建和消费也是很昂贵的，这也是为什么建议尽量使用线程池的一个主要原因。

对于Thread的使用太简单了，这里就不重复了，总结一下线程的主要几点性能影响：

- 线程的创建、销毁都是很昂贵的；
- 线程上下文切换有极大的性能开销，当然假如需要调度的新线程与当前是同一线程的话，就不需要线程上下文切换了，效率要快很多；
- 这一点需要注意，GC执行回收时，首先要（安全的）挂起所有线程，遍历所有线程栈（根），GC回收后更新所有线程的根地址，再恢复线程调用，线程越多，GC要干的活就越多；

当然现在硬件的发展，CPU的核心越来越多，多线程技术可以极大提高应用程序的效率。但这也必须在合理利用多线程技术的前提下，了线程的基本原理，然后根据实际需求，还要注意相关资源环境，如磁盘IO、网络等情况综合考虑。

## 多线程

单线程的使用这里就略过了，那太easy了。上面总结了线程的诸多不足，因此微软提供了可供多线程编程的各种技术，如线程池、任务、并行等等。

## 线程池ThreadPool

线程池的使用是非常简单的，如下面的代码，把需要执行的代码提交到线程池，线程池内部会安排一个空闲的线程来执行你的代码，完全不用管理内部是如何进行线程调度的。

```
1 ThreadPool.QueueUserWorkItem(t => Console.WriteLine("Hello thread pool"));
```

每个CLR都有一个线程池，线程池在CLR内可以多个AppDomain共享，线程池是CLR内部管理的一个线程集合，初始是没有线程的，在需要的时候才会创建。线程池的主要结构图如下图所示，基本流程如下：

- 线程池内部维护一个请求队列，用于缓存用户请求需要执行的代码任务，就是ThreadPool.QueueUserWorkItem提交的请求；
- 有新任务后，线程池使用空闲线程或新线程来执行队列请求；
- 任务执行完后线程不会销毁，留着重复使用；
- 线程池自己负责维护线程的创建和销毁，当线程池中有大量闲置的线程时，线程池会自动结束一部分多余的线程来释放资源；

线程池是有一个容量的，因为他是一个池子嘛，可以设置线程池的最大活跃线程数，调用方法ThreadPool.SetMaxThreads可以设置相关参数。但很多编程实践里都不建议程序员们自己去设置这些参数，其实微软为了提高线程池性能，做了大量的优化，线程池可以很智能的确定是否要创建或是消费线程，大多数情况都可以满足需求了。

线程池使得线程可以充分有效地被利用，减少了任务启动的延迟，也不用大量的去创建线程，避免了大量线程的创建和销毁对性能的极大影响。

上面了解了线程的基本原理和诸多优点后，如果你是一个爱思考的猿类，应该会很容易发现很多疑问，比如把任务添加到线程池队列后，怎么取消或挂起呢？如何知道她执行完了呢？下面来总结一下线程池的不足：

- 线程池内的线程不支持线程的挂起、取消等操作，如想要取消线程里的任务，.NET支持一种协作式方式取消，使用起来也不少很方便，而且有些场景并不满足需求；
- 线程内的任务没有返回值，也不知道何时执行完成；
- 不支持设置线程的优先级，还包括其他类似需要对线程有更多的控制的需求都不支持；

因此微软为我们提供了另外一个东西叫做Task来补充线程池的某些不足。

## 任务Task与并行Parallel



任务Task与并行Parallel本质上内部都是使用的线程池，提供了更丰富的并行编程的方式。任务Task基于线程池，可支持返回值，支持比较强大的任务执行计划定制等功能，下面是一个简单的示例。Task提供了很多方法和属性，通过这些方法和属性能够对Task的执行进行控制，并且能够获得其状态信息。Task的创建和执行都是独立的，因此可以对关联操作的执行拥有完全的控制权。

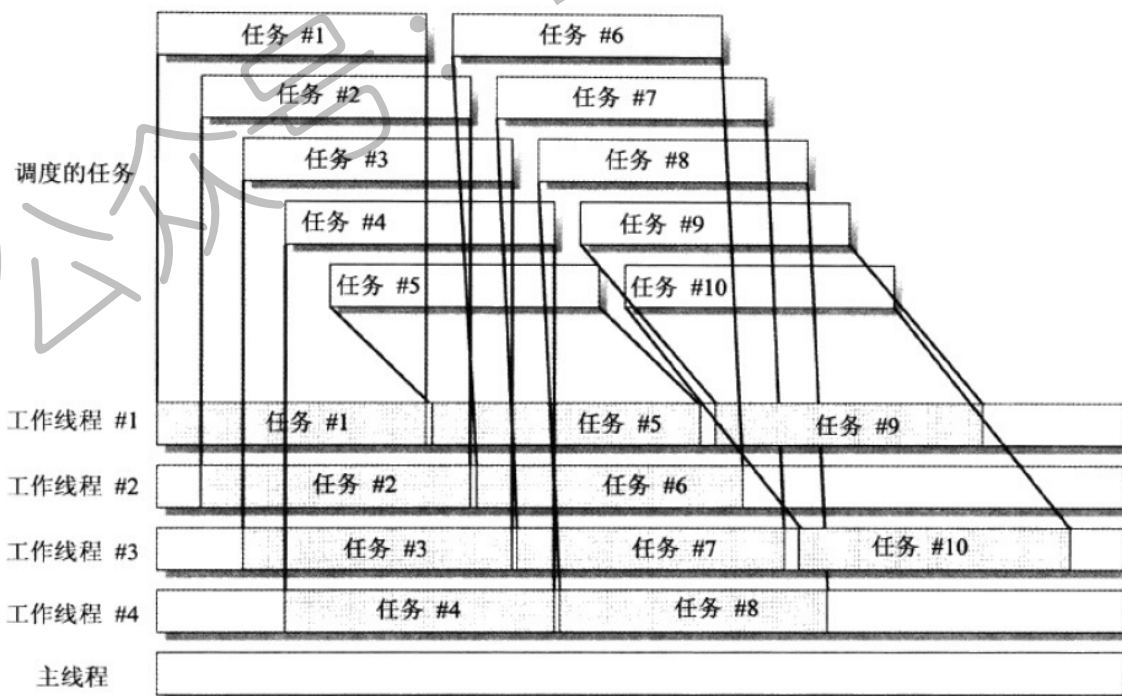
```
1
2 //创建一个任务
3 Task<int> t1 = new Task<int>(n =>
```

```

4 {
5     System.Threading.Thread.Sleep(1000);
6     return (int)n;
7 }, 1000);
8 //定制一个延续任务计划
9 t1.ContinueWith(task =>
10 {
11     Console.WriteLine("end" + t1.Result);
12 }, TaskContinuationOptions.AttachedToParent);
13 t1.Start();
14 //使用Task.Factory创建并启动一个任务
15 var t2 = System.Threading.Tasks.Task.Factory.StartNew(() =>
16 {
17     Console.WriteLine("t1:" + t1.Status);
18 });
19 Task.WaitAll();
20 Console.WriteLine(t1.Result);

```

并行Parallel内部其实使用的是Task对象（TPL会在内部创建System.Threading.Tasks.Task的实例），所有并行任务完成后才会返回。少量短时间任务建议就不要使用并行Parallel了，并行Parallel本身也是有性能开销的，而且还要进行并行任务调度、创建调用方法的委托等等。



## GUI线程处理模型





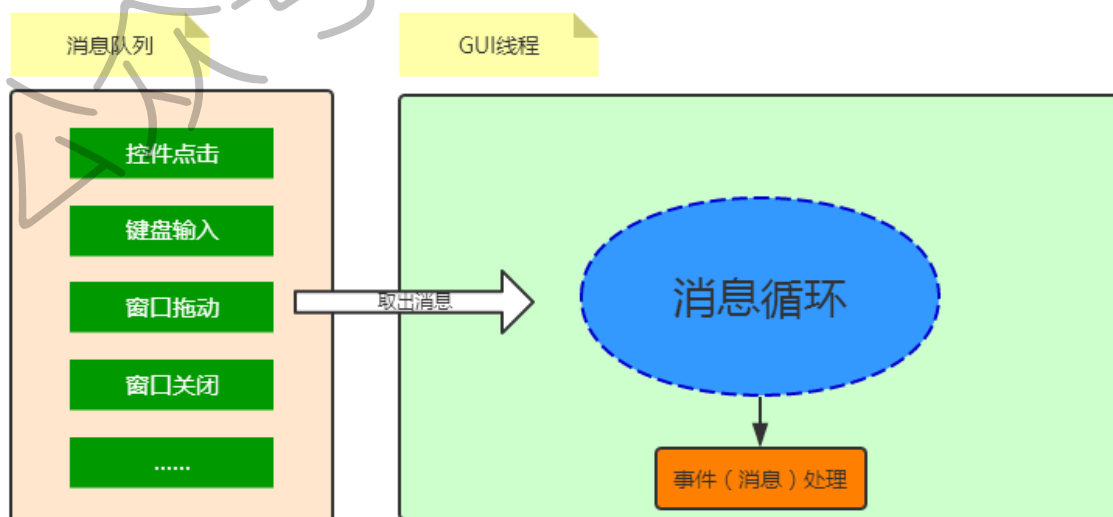
这是很多开发C/S客户端应用程序会遇到的问题，GUI程序的界面控件不允许跨线程访问，如果在其他线程中访问了界面控件，运行时就会抛出一个异常，就像下面的图示，是不是很熟悉！这其中的罪魁祸首就是，就是“GUI的线程处理模型”。



.NET支持多种不同应用程序模型，大多数的线程都是可以做任何事情（他们可能没有引入线程模型），但GUI应用程序（主要是Winform、WPF）引入了一个特殊线程处理模型，UI控件元素只能由创建它的线程访问或修改，微软这样处理是为了保证UI控件的线程安全。

为什么在UI线程中执行一个耗时的计算操作，会导致UI假死呢？这个问题要追溯到Windows的消息机制了。

因为Windows是基于消息机制的，我们在UI上所有的键盘、鼠标操作都是以消息的形式发送给各个应用程序的。GUI线程内部就有一个消息队列，GUI线程不断的循环处理这些消息，并根据消息更新UI的呈现。如果这个时候，你让GUI线程去处理一个耗时的操作（比如花10秒去下载一个文件），那GUI线程就没办法处理消息队列了，UI界面就处于假死的状态。



那我们该怎么办呢？不难想到使用线程，那在线程里处理事件完成后，需要更新UI控件的状态，又该怎么办呢？常用几种方式：



## ① 使用GUI控件提供的方法，Winform是控件的Invoke方法，WPF中是控件的Dispatcher.Invoke方法

```
1 //1.Winform: Invoke方法和BeginInvoke
2 this.label.Invoke(method, null);
3
4 //2.WPF: Dispatcher.Invoke
5 this.label.Dispatcher.Invoke(method, null);
```

## ② 使用.NET中提供的BackgroundWorker执行耗时计算操作，在其任务完成事件RunWorkerCompleted 中更新UI控件

```
1 using (BackgroundWorker bw = new BackgroundWorker())
2 {
3     bw.RunWorkerCompleted += new RunWorkerCompletedEventHandler((obj, arg) =>
4     {
5         this.label.Text = "anidng";
6     });
7     bw.RunWorkerAsync();
8 }
```

## ③ 看上去很高大上的方法：使用GUI线程处理模型的同步上下文来送封UI控件修改操作，这样可以不需要调用UI控件元素

.NET中提供一个用于同步上下文的类SynchronizationContext，利用它可以把应用程序模型链接到他的线程处理模型，其实它的本质还是调用的第一步①中的方法。

实现代码分为三步，第一步定义一个静态类，用于GUI线程的UI元素访问封装：

```
1 public static class GUIThreadHelper
2 {
3     public static System.Threading.SynchronizationContext GUISyncContext
4     {
5         get { return _GUISyncContext; }
6         set { _GUISyncContext = value; }
7     }
8
9     private static System.Threading.SynchronizationContext _GUISyncContext =
10         System.Threading.SynchronizationContext.Current;
11
12     /// <summary>
13     /// 主要用于GUI线程的同步回调
14     /// </summary>
```

```

15     /// <param name="callback"></param>
16     public static void SyncContextCallback(Action callback)
17     {
18         if (callback == null) return;
19         if (UISyncContext == null)
20         {
21             callback();
22             return;
23         }
24         UISyncContext.Post(result => callback(), null);
25     }
26
27     /// <summary>
28     /// 支持APM异步编程模型的GUI线程的同步回调
29     /// </summary>
30     public static AsyncCallback SyncContextCallback(AsyncCallback callback)
31     {
32         if (callback == null) return callback;
33         if (UISyncContext == null) return callback;
34         return asyncresult => UISyncContext.Post(result => callback(result as
35             IAsyncResult), asyncresult);
36     }

```

第二步，在主窗口注册当前SynchronizationContext:

```

1
2 public partial class MainWindow : Window
3 {
4     public MainWindow()
5     {
6         InitializeComponent();
7         CLRTest.ConsoleTest.GUIThreadHelper.UISyncContext =
8         System.Threading.SynchronizationContext.Current;
9     }

```

第三步，就是使用了，可以在任何地方使用

```

1
2 GUIThreadHelper.SyncContextCallback(() =>

```

```
3 {  
4     this.txtMessage.Text = res.ToString();  
5     this.btnTest.Content = "DoTest";  
6     this.btnTest.IsEnabled = true;  
7 }));
```

## 线程同步构造

多线程编程中很常用、也很重要的一点就是线程同步问题，掌握线程同步对临界资源正确使用、线程性能有至关重要的作用！基本思路是很简单的，就是加锁嘛，在临界资源的门口加一把锁，来控制多个线程对临界资源的访问。但在实际开发中，根据资源类型不同、线程访问方式的不同，有多种锁的方式或控制机制（基元用户模式构造和基元内核模式构造）。.NET提供了两种线程同步的构造模式，需要理解其基本原理和使用方式。

基元线程同步构造分为：基元用户模式构造和基元内核模式构造，两种同步构造方式各有优缺点，而混合构造（如lock）就是综合两种构造模式的优点。

## 用户模式构造



基元用户模式比基元内核模式速度要快，她使用特殊的cpu指令来协调线程，在硬件中发生，速度很快。但也因此Windows操作系统永远检测不到一个线程在一个用户模式构造上阻塞了。举个例子来模拟一下用户模式构造的同步方式：

- 线程1请求了临界资源，并在资源门口使用了用户模式构造的锁；
- 线程2请求临界资源时，发现有锁，因此就在门口等待，并不停的去询问资源是否可用；
- 线程1如果使用资源时间较长，则线程2会一直运行，并且占用CPU时间。占用CPU干什么呢？她会不停的轮询锁的状态，直到资源可用，这就是所谓的活锁；

缺点有没有发现？线程2会一直使用CPU时间（假如当前系统只有这两个线程在运行），**也就意味着不仅浪费了CPU时间，而且还会有频繁的线程上下文切换，对性能影响是很严重的。**

当然她的优点是效率高，适合哪种对资源占用时间很短的线程同步。.NET中为我们提供了两种原子性操作，利用原子操作可以实现一些简单的用户模式锁（如自旋锁）。

**System.Threading.Interlocked**：易失构造，它在包含一个简单数据类型的变量上执行原子性的读或写操作。

**Thread.VolatileRead 和 Thread.VolatileWrite**：互锁构造，它在包含一个简单数据类型的变量上执行原子性的读和写操作。

以上两种原子性操作的具体内涵这里就细说了（有兴趣可以去研究文末给出的参考书籍或资料），针对题目11，来看一下题目代码：

```
1 int a = 0;
```

```
2 System.Threading.Tasks.Parallel.For(0, 100000, (i) =>
3 {
4     a++;
5 });
6 Console.Write(a);
```

上面代码是通过并行（多线程）来更新共享变量a的值，结果肯定是小于等于100000的，具体多少是不稳定的。解决方法，可以使用我们常用的Lock，还有更有效的就是使用

**System.Threading.Interlocked**提供的原子性操作，保证对a的值操作每一次都是原子性的：

```
1 System.Threading.Interlocked.Add(ref a, 1); //正确
```

下面的图是一个简单的性能验证测试，分别使用Interlocked、不用锁、使用lock锁三种方式来测试。不用锁的结果是95，这答案肯定不是你想要的，另外两种结果都是对的，性能差别却很大。

```
✓ DoThreadTest_Interlocked [0:00.527] Success
✓ DoThreadTest_NoLock      [0:00.526] Success
✓ DoThreadTest_WithLock    [0:02.095] Success
```

为了模拟耗时操作，对代码稍作了修改，如下，所有的循环里面加了代码Thread.Sleep(20);。如果没有Thread.Sleep(20);他们的执行时间是差不多的。

```
1 System.Threading.Tasks.Parallel.For(0, 100, (i) =>{
2     lock (_obj) {
3         a++; //不正确
4         Thread.Sleep(20); } });
```

## 内核模式构造



这是针对用户模式的一个补充，先模拟一个内核模式构造的同步流程来理解她的工作方式：

- 线程1请求了临界资源，并在资源门口使用了内核模式构造的锁；
- 线程2请求临界资源时，发现有锁，就会被系统要求睡眠（阻塞），线程2就不会被执行了，也就不会浪费CPU和线程上下文切换了；
- 等待线程1使用完资源后，解锁后会发送一个通知，然后操作系统会把线程2唤醒。假如有多个线程在临界资源门口等待，则会挑选一个唤醒；

看上去是不是非常棒！彻底解决了用户模式构造的缺点，但内核模式也有缺点的：将线程从用户模式切换到内核模式（或相反）导致巨大性能损失。调用线程将从托管代码转换为内核代码，再转回来，会浪费大量CPU时间，同时还伴随着线程上下文切换，因此尽量不要让线程从用户模式转到内核模式。

她的优点就是阻塞线程，不浪费CPU时间，适合那种需要长时间占用资源的线程同步。

内核模式构造的主要有两种方式，以及基于这两种方式的常见的锁：

- **基于事件**：如AutoResetEvent、ManualResetEvent
- **基于信号量**：如Semaphore

## 混合线程同步



既然内核模式和用户模式都有优缺点，混合构造就是把两者结合，充分利用两者的优点，把性能损失降到最低。大概的思路很好理解，就是如果是在没有资源竞争，或线程使用资源的时间很短，就是用用户模式构造同步，否则就升级到内核模式构造同步，其中最典型的代表就是Lock了。

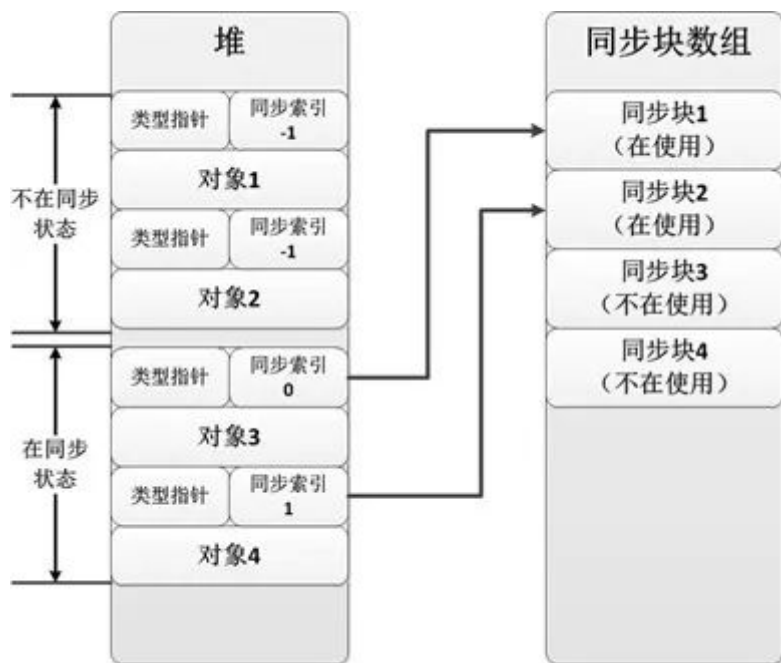
常用的混合锁还不少呢！如SemaphoreSlim、ManualResetEventSlim、Monitor、ReadWriteLockSlim，这些锁各有特点和锁使用的场景。这里主要就使用最多的lock来详细了解下。lock的本质就是使用的Monitor，lock只是一种简化的语法形式，实质的语法形式如下：

```
1 bool lockTaken = false;try{
2     Monitor.Enter(obj, ref lockTaken);
3     //...
4 }finally{
5     if (lockTaken) Monitor.Exit(obj);}
```

那lock或Monitor需要锁定的那个对象是什么呢？注意这个对象才是锁的关键，在此之前，需要先回顾一下引用对象的同步索引块（AsynBlockIndex），这是前面文章中提到过的引用对象的标准配置之一（还有一个是类型对象指针TypeHandle），它的作用就在这里了。

同步索引块是.NET中解决对象同步问题的基本机制，该机制为每个堆内的对象（即引用类型对象实例）分配一个同步索引，她其实是一个地址指针，初始值为-1不指向任何地址。

- 创建一个锁对象Object obj，obj的同步索引块（地址）为-1，不指向任何地址；
- Monitor.Enter（obj），创建或使用一个空闲的同步索引块（如下图中的同步块1），（图片来源），这个才是真正的同步索引块，其内部结构就是一个混合锁的结构，包含线程ID、递归计数、等待线程统计、内核对象等，类似一个混合锁AnotherHybridLock。obj对象（同步索引块AsynBlockIndex）指向该同步块1；
- Exit时，重置为-1，那个同步索引块1可以被重复利用；



因此，锁对象要求必须为一个引用对象（在堆上）。

## 多线程使用及线程同步总结



首先还是尽量避免线程同步，不管使用什么方式都有不小的性能损失。一般情况下，大多使用Lock，这个锁是比较综合的，适应大部分场景。在性能要求高的地方，或者根据不同的使用场景，可以选择更符合要求的锁。

在使用Lock时，关键点就是锁对象了，需要注意以下几个方面：

- 这个对象肯定要是引用类型，值类型可不可以呢？值类型可以装箱啊！你觉得可不可以？但也不要值类型，因为值类型多次装箱后的对象是不同的，会导致无法锁定；
- 不要锁定this，尽量使用一个没有意义的Object对象来锁；
- 不要锁定一个类型对象，因类型对象是全局的；
- 不要锁定一个字符串，因为字符串可能被驻留，不同字符串对象可能指向同一个字符串；
- 不要使用[System.Runtime.CompilerServices.MethodImpl(MethodImplOptions.Synchronized)]，这个可以使用在方法上面，保证方法同一时刻只能被一个线程调用。她实质上是使用lock的，如果是实例方法，会锁定this，如果是静态方法，则会锁定类型对象；

## 题目答案解析:

### 1. 描述线程与进程的区别？

- 一个应用程序实例是一个进程，一个进程内包含一个或多个线程，线程是进程的一部分；
- 进程之间是相互独立的，他们有各自的私有内存空间和资源，进程内的线程可以共享其所属进程的所有资源；

### 2. 为什么GUI不支持跨线程访问控件？一般如何解决这个问题？



因为GUI应用程序引入了一个特殊的线程处理模型，为了保证UI控件的线程安全，这个线程处理模型不允许其他子线程跨线程访问UI元素。解决方法还是比较多的，如：

- 利用UI控件提供的方法，Winform是控件的Invoke方法，WPF中是控件的Dispatcher.Invoke方法；
- 使用BackgroundWorker；
- 使用GUI线程处理模型的同步上下文SynchronizationContext来提交UI更新操作

上面几个方式在文中已详细给出。

### 3. 简述后台线程和前台线程的区别？

应用程序必须运行完所有的前台线程才可以退出，或者主动结束前台线程，不管后台线程是否还在运行，应用程序都会结束；而对于后台线程，应用程序则可以不考虑其是否已经运行完毕而直接退出，所有的后台线程在应用程序退出时都会自动结束。

通过将 Thread.IsBackground 设置为 true，就可以将线程指定为后台线程，主线程就是一个前台线程。

### 4. 说说常用的锁，lock是一种什么样的锁？

常用的如SemaphoreSlim、ManualResetEventSlim、Monitor、ReadWriteLockSlim，lock是一个混合锁，其实质是Monitor['mɒnɪtə]。

### 5. lock为什么要锁定一个参数，可不可锁定一个值类型？这个参数有什么要求？

lock的锁对象要求为一个引用类型。她可以锁定值类型，但值类型会被装箱，每次装箱后的对象都不一样，会导致锁定无效。

对于lock锁，锁定的这个对象参数才是关键，这个参数的同步索引块指针会指向一个真正的锁（同步块），这个锁（同步块）会被复用。

### 6. 多线程和异步有什么关系和区别？

多线程是实现异步的主要方式之一，异步并不等同于多线程。实现异步的方式还有很多，比如利用硬件的特性、使用进程或线程等。在.NET中就有很多的异步编程支持，比如很多地方都有Begin\*\*\*、End\*\*\*的方法，就是一种异步编程支持，她内部有些是利用多线程，有些是利用硬件的特性来实现的异步编程。

### 7. 线程池的优点有哪些？又有哪些不足？

优点：减小线程创建和销毁的开销，可以复用线程；也从而减少了线程上下文切换的性能损失；在GC回收时，较少的线程更有利于GC的回收效率。

缺点：线程池无法对一个线程有更多的精确的控制，如了解其运行状态等；不能设置线程的优先级；加入到线程池的任务（方法）不能有返回值；对于需要长期运行的任务就不适合线程池。

### 8. Mutex和lock有何不同？一般用哪一个作为锁使用更好？

Mutex是一个基于内核模式的互斥锁，支持锁的递归调用，而Lock是一个混合锁，一般建议使用Lock更好，因为lock的性能更好。

## 9. 下面的代码，调用方法DeadLockTest (20) ，是否会引起死锁？并说明理由。

```
1 public void DeadLockTest(int i){
2     lock (this)    //或者lock一个静态object变量
3     {
4         if (i > 10)
5         {
6             Console.WriteLine(i--);
7             DeadLockTest(i);
8         }
9     }
10 }
```

不会的，因为lock是一个混合锁，支持锁的递归调用，如果你使用一个ManualResetEvent或AutoResetEvent可能就会发生死锁。

## 10. 用双检锁实现一个单例模式Singleton。

```
1 public static class Singleton<T> where T : class, new()
2 {
3     private static T _Instance;
4     private static object _lockObj = new object();
5
6     /// <summary>
7     /// 获取单例对象的实例
8     /// </summary>
9     public static T GetInstance()
10 {
11     if (_Instance != null) return _Instance;
12     lock (_lockObj)
13     {
14         if (_Instance == null)
15         {
16             var temp = Activator.CreateInstance<T>();
17             System.Threading.Interlocked.Exchange(ref _Instance, temp);
18         }
19     }
20     return _Instance;
21 }
```

```
21     }  
22 }
```

## 11.下面代码输出结果是什么？为什么？如何改进她？

```
1  int a = 0;  
2  System.Threading.Tasks.Parallel.For(0, 100000, (i) =>  
3  {  
4      a++;  
5  });  
6  Console.Write(a)
```

输出结果不稳定，小于等于100000。因为多线程访问，没有使用锁机制，会导致有更新丢失。具体原因和改进在文中已经详细的给出了。

关注微信公众号：DotNet开发跳槽

## 八、SQL语言基础及数据库基本原理

### 常见面试题目：

1. 索引的作用？她的优点缺点是什么？
2. 介绍存储过程基本概念和 她的优缺点？
3. 使用索引有哪些需要注意的地方？
4. 索引碎片是如何产生的？有什么危害？又该如何处理？
5. 锁的目的是什么？
6. 锁的粒度有哪些？
7. 什么是事务？什么是锁？
8. 视图的作用，视图可以更改么？
9. 什么是触发器(trigger)？触发器有什么作用？
10. SQL里面IN比较快还是EXISTS比较快？
11. 维护数据库的完整性和一致性，你喜欢用触发器还是自写业务逻辑？为什么？

### 基础SQL语法

以下SQL所使用的实例数据库为Sqlite（因为相当轻量），数据库文件（下载链接,test.db,6KB），SQLite数据库管理工具推荐SQLite Expert Personal。

## 0. 创建表

定义如下表结构，后面的题目都以此表结构为依据。

- 1 **Student**(ID, Name, Age, Sex) 学生表
- 2 **Course**(ID, Name, TeacherID) 课程表
- 3 **Score**(StudentID, CourseID, Score) 成绩表
- 4 **Teacher**(ID, Name) 教师表

创建表的语法很简单，SQL语句：

```
1 CREATE TABLE [Student] (  
2     [ID] INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
3     [Name] NVARCHAR(20),  
4     [Age] INT,  
5     [Sex] INT);  
6  
7 CREATE TABLE [Course] (  
8     [ID] INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
9     [Name] NVARCHAR(20),  
10    [TeacherID] INT)  
11  
12 CREATE TABLE [Score] (  
13    [Score] double,  
14    [StudentID] INT,  
15    [CourseID] INT)  
16  
17 CREATE TABLE [Teacher] (  
18    [ID] INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
19    [Name] NVARCHAR(20))
```

## 1. 查询语文“1”比数学“2”课程成绩高的所有学生的姓名

这是一个嵌套查询的题目，考察对子查询的使用，子查询结果作为一个集合可以当做一个独立的表来看待，子查询必须用括号括起来：

```
1 select st.[Name], c1.Score, c2.Score from  
2 (select sc.[Score], sc.StudentID from Score sc where sc.[CourseID]=1)c1,  
3 (select sc.[Score], sc.StudentID from Score sc where sc.[CourseID]=2)c2  
4 join Student st on st.[ID]= c1.[StudentID]  
5 where c1.[Score]>c2.[Score] and c1.[StudentID]==c2.[StudentID]
```

## 2. 查询平均成绩大于60分的同学的学号和平均成绩

GROUP BY 语句用于结合合计函数，根据一个或多个列对结果集进行分组。GROUP BY子句在SELECT语句的WHERE子句之后并ORDER BY子句之前。WHERE 关键字无法与合计函数一起使用，**GROUP BY后面不能接WHERE条件，使用HAVING代替。**

```
1 select sc.[StudentID],avg(sc.Score) from Score sc
2 group by sc.[CourseID] having avg(sc.Score)>60
```

## 3. 查询所有同学的学号、姓名、选课数、总成绩；

```
1 select st.[ID],st.Name,count(sc.CourseID),sum(sc.Score) from Student st
2 left outer join Score sc on sc.[StudentID]=st.[ID]
3 group by st.[ID]
```

外连接的三种形式如下表，其中outer可以省略。与外连接对应的就是内连接inner join，要两个表同时满足指定条件。

## 4. 查询姓“张”的老师的个数；

```
1 select count(t.ID) from Teacher t where t.Name like '张%'
```

SQL LIKE子句使用通配符运算符比较相似的值。符合LIKE操作符配合使用2个通配符：

- 百分号 (%)：百分号代表零个，一个或多个字符
- 下划线 (\_)：下划线表示单个数字或字符

## 5. 找出教师表中姓名重复的数据，然后删除多余重复的记录，只留ID小的那个。

```
1 select t.Name,count(t.Name) from Teacher t group by t.[Name] having count(t.Name)>1
```

删除多余的记录，写这种稍微复杂一点的sql的时候，要学会拆解，此题可以拆解为三个部分（删除+重复数据+重复数据中ID最小的数据），先分别把3个部分的sql写了，然后再一步步合并，这样就轻松多了。

```
1 delete from Teacher where Name in
2     (select t2.Name from Teacher t2 group by t2.[Name] having count(t2.Name)>1)
3 and ID not in
4     (select min(t3.ID) from Teacher t3 group by t3.Name having count(t3.Name)>1)
```

## 6. 按照成绩分段标示 (<60不及格, 60-80良, >80优), 输出所有学生姓名、课程名、成绩、成绩分段标示。

```
1 select st.Name,c.Name,sc.Score,(case
2     when sc.Score > 80 then '优'
3     when sc.Score < 60 then '不及格'
4     else '良' end) as 'Remark'
5 from Score sc
6 inner join Student st on st.ID=sc.[StudentID]
7 inner join Course c on c.ID=sc.[CourseID]
```

|   |    |    |    |     |
|---|----|----|----|-----|
| 1 | 张三 | 语文 | 50 | 不及格 |
| 2 | 张三 | 数学 | 75 | 良   |
| 3 | 李四 | 语文 | 66 | 良   |
| 4 | 李四 | 数学 | 90 | 优   |
| 5 | 小美 | 语文 | 88 | 优   |
| 6 | 小美 | 数学 | 55 | 不及格 |

## 7. 查询所有课程成绩小于60分的同学的学号、姓名信息

这个比较简单, 下面给出了两种方法, 使用join链接和子查询:

```
1 select distinct st.* from Student st
2 left join Score sc on sc.[StudentID]=st.ID
3 where sc.[Score]<60
4 --
5 select * FROM Student st WHERE st.ID in
6 (SELECT s1.ID FROM Student s1,Score s2 WHERE s1.ID=s2.[StudentID] AND
7  s2.Score<60)
```

## 8. 查询各科成绩最高和最低的分: 以如下形式显示: 课程名称, 最高分, 最低分

```
1 select c.Name as 课程,max(sc.Score) as 最高分,min(sc.Score) as 最低分 from Score sc
2 left join Course c on c.ID=sc.[CourseID]
3 group by sc.[CourseID]
```

## 9. 查询不同老师所教不同课程平均分从高到低显示

```
1 select t.Name,c.Name, avg(sc.Score) from Score sc,Teacher t,Course c
```



```
2 where sc.[CourseID]=c.ID and c.TeacherID=t.ID
3 group by sc.[CourseID] order by avg(sc.Score) desc
```

| Name                          | Name_1 | avg(sc.Score)     |
|-------------------------------|--------|-------------------|
| Click here to define a filter |        |                   |
| 王老师                           | 数学     | 73.33333333333333 |
| 张老师                           | 语文     | 68                |

## 10. 查询和“1”号的同学学习的课程完全相同的其他同学学号和姓名

```
1 select s1.StudentID from Score s1 where s1.CourseID in(select s2.CourseID from Score s2
where s2.StudentID=1)
2 group by s1.StudentID having count(*)=(select count(*) from Score s3 where
s3.StudentID=3)
```

## 11. 查询选修“张老师”老师所授课程的学生中，成绩最高的学生姓名及其成绩

```
1 select t.Name,c.Name,s1.Score from Score s1,Teacher t,Course c
2 where t.ID=c.[TeacherID] and s1.CourseID=c.ID and t.Name='张老师'
3 and s1.Score=(select max(Score) from Score where CourseID=c.ID)
```

| Name                          | Name_1 | Score | R |
|-------------------------------|--------|-------|---|
| Click here to define a filter |        |       |   |
| 张老师                           | 语文     | 99    |   |
| 张老师                           | 体育     | 61    |   |

**注意：**多表连接查询有多种写法，比如本题目中多表连接可以有以下两种方式：

```
1 select t.Name,c.Name,s1.Score from Score s1,Teacher t,Course c where t.ID=c.[TeacherID]
and s1.CourseID=c.ID
2 -- 下面的写法和上面的效果是一样的! --
3 select t.Name,c.Name,s1.Score from Score s1
4 join Teacher t on t.ID=c.[TeacherID]
5 join Course c on s1.CourseID=c.ID
```

FROM TABLE1,TABLE2...效果等效于FROM TABLE1 join TABLE2..., 都是内连接inner操作。详细的SQL表连接操作可以参考：深入理解SQL的四种连接-左外连接、右外连接、内连接、全连接

## 12. 查询所有成绩第二名到第四名的成绩

```
1 select s.[StudentID],s.Score from Score s order by s.Score desc limit 2 offset 2
```

```

2  -- SQL 2005/2008中的分页函数是ROW_NUMBER() Over (Order by 列...)--
3  select t.[StudentID],t.Score from(
4      select s2.[StudentID],s2.Score,ROW_NUMBER() OVER (ORDER BY s2.[Score]) AS rn
      from Score s2) t
5  where t.rn>=2 and t.rn<=4

```

这是一个分页的题目，上面这是Sqlite提供的内置方法limite进行分页，不同数据库的分页方式又有些差别，但都大同小异。[基本的过程都是先根据条件查询所需数据（加上行号），然后再此基础上返回指定行区间段的数据。其实SQLServer也很简单，不同的版本也有些不同，可以参考：SQL Server 常用分页SQL](#)

## 12. 查询各科成绩前2名的记录:(不考虑成绩并列情况)

```

1  select * from Score s1 where s1.Score
2      in(select s2.Score from Score s2 where s1.[CourseID]=s2.[CourseID] order by
3      s2.Score desc limit 2 offset 0)
4  order by s1.[CourseID],s1.[Score] desc
5  -- 上面是sqlite中的语法，sqlite中没有top，使用limit代替，效果是一样的 --
6  select * from Score s1 where s1.Score
7      in(select Top 2 s2.Score from Score s2 where s1.[CourseID]=s2.[CourseID] order by
8      s2.Score desc)
9  order by s1.[CourseID],s1.[Score] desc

```

| Score | StudentID | CourseID |
|-------|-----------|----------|
| 99    | 4         | 1        |
| 88    | 3         | 1        |
| 90    | 2         | 2        |
| 75    | 1         | 2        |
| 61    | 4         | 3        |

## 数据库基本存储原理

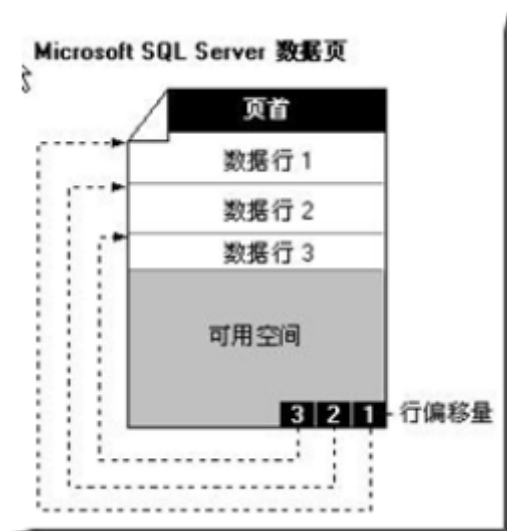
### 基本存储单元——页



数据库文件存储是以页为存储单元的，一个页是8K（8192Byte），一个页就可以存放N行数据。我们表里的数据都是存放在页上的，这种叫数据页。还有一种页存放索引数据的，叫索引页。

同时，页也是IO读取的最小单元（物理IO上不是按行读取），也是所有权的最小单位。如果一页中包含了表A的一行数据，这页就只能存储表A的行数据了。或是一页中包含了索引B的条目，那这页也仅

仅只能存储索引B的条目了。每页中除去存储数据之外，还存储一些页头信息以及行偏移以便SQL Server知道具体每一行在页中的存储位置。

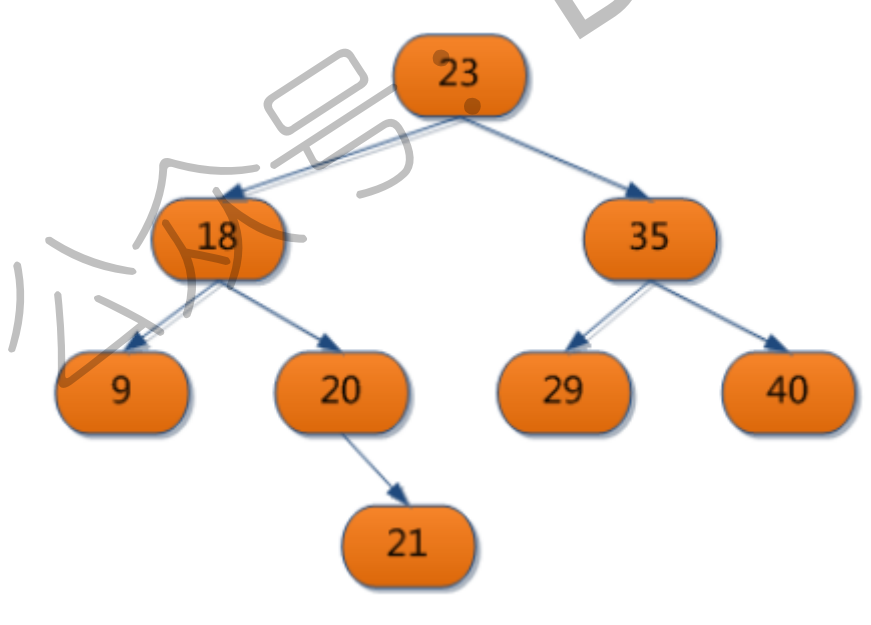


数据库的基本物理存储单元是页，一个表由很多个页组成，那这些页又是如何组织的呢？我们一般都会对表创建索引，这些索引又是如何存储的呢？不要走开，请看下文。

## 表/索引的存储结构



如下图，是一个B树（二叉搜索树）的示例，都是小的元素放左边，大的元素放右边，依次构造的，比如要查找元素9，从根节点开始，只要比较三次就找到他了，查询效率是非常高的。



B+树和B-树都是B树的变种，或者说是更加高级的复杂版本（关于B树的资料，有兴趣可以自己去学习，这里只是抛砖引玉）。B+树和B-树是数据库中广泛应用的索引存储结构，它可以极大的提高数据查找的效率。前面说了数据库存储的基本单元是页，因此，索引树上的节点就是页了。因此不难看出，索引的主要优点和目的就是为了提高查询效率。

为了保证数据的查询效率，当新增、修改、删除数据的时候，都需要维护这颗索引树，就可能会出现分裂、合并节点（页）的情况（这是树的结构所决定的，想要更好理解这一点，可以尝试自己代码实现一下B-树B+树）。好！重点来了！

### 索引的缺点：

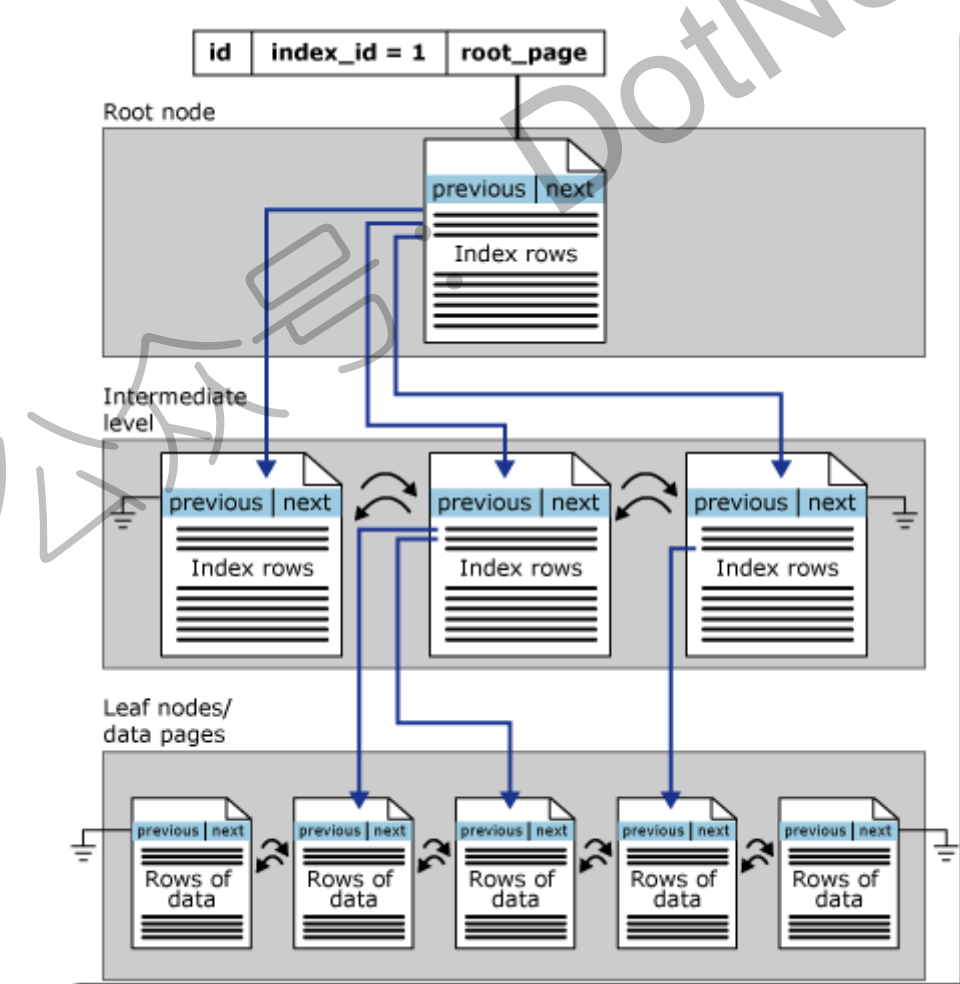
- 当新增、修改、删除数据的时候，需要维护索引树，有一定的性能影响；
- 同上面，在频繁的树维护过程中，B树的页拆分、合并会造成大量的索引碎片，又会极大的影响查询效率，因此索引还需要维护；
- 非聚集索引需要额外的存储空间，不过这个一般问题都不是很大，但是需要注意的一个问题；

## 聚集索引



**聚集索引**决定了表数据的物理存储顺序，也就是说表的物理存储是根据聚集索引结构进行顺序存储的，因此一个表只能有一个聚集索引。如下图，就是一个聚集索引的树结构：

- 所有数据都在叶子节点的页上，在叶子节点（数据页）之间有一个链指针，这是B+树的特点；
- 非叶子节点都是索引页，存储的就是聚集索引字段的值；
- 表的物理存储就是依据聚集索引的结构，一个表只能有一个聚集索引；

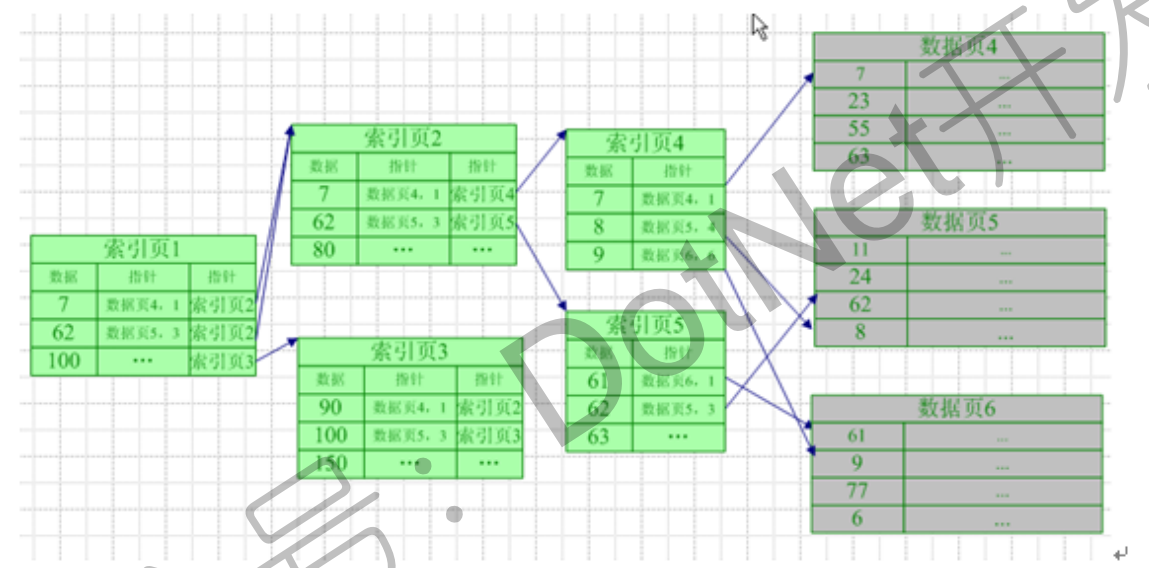


聚集索引的所有的数据都存储在叶子节点上，数据查询的复杂度都是一样的（树的深度），按照聚集索引列查找数据效率是非常高的。上面说了，聚集索引决定了表的物理存储结构，那如果没有创建聚集索引，会如何呢？——**表内的所有页都无序存放，是一个无序的堆结构。**堆数据的查询就会造成表扫描，性能是非常低的。

**因此聚集索引的重要性不言而喻，一般来说，大多会对主键建立聚集索引，大多数普通情况这么做也可以。但实际应用应该遵从一个原则就是“频繁使用的、排序的字段上创建聚集索引”**

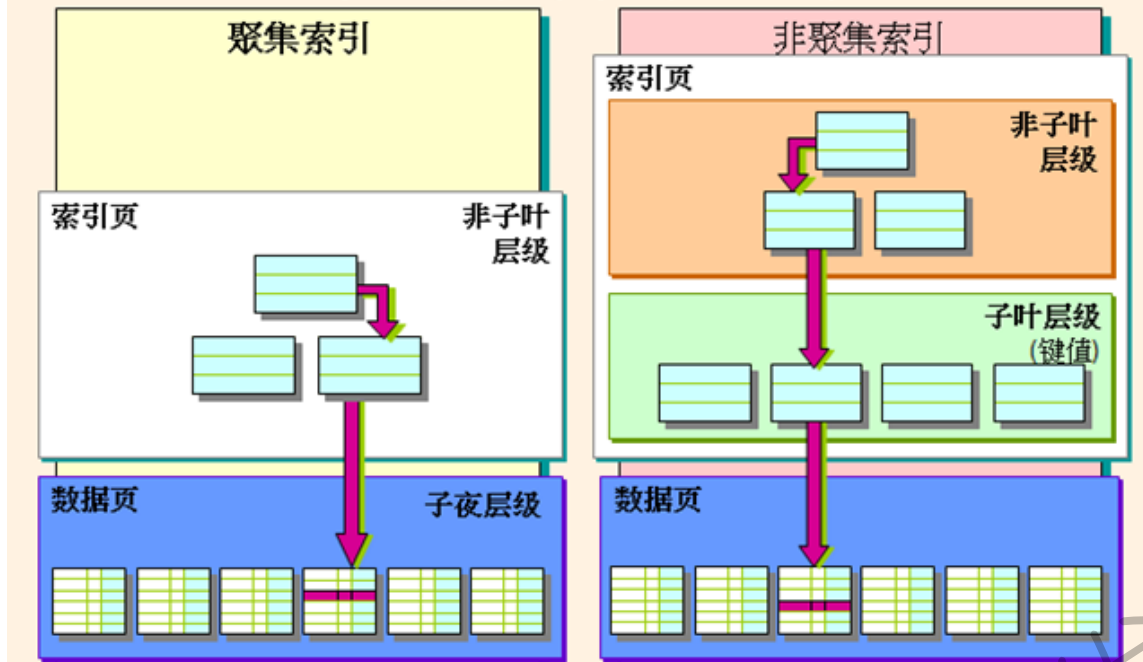
## 非聚集索引

除了聚集索引以外的其他索引，都称之为非聚集索引，非聚集索引一般都是为了优化特定的查询效率而创建的。非聚集索引也是B树（B+树和B-树）的结构，与非聚集索引的存储结构唯一不一样的，就是非聚集索引中不存储真正的数据行，因为在聚集索引中已经存放了所有数据，非聚集索引只包含一个指向数据行的指针即可。



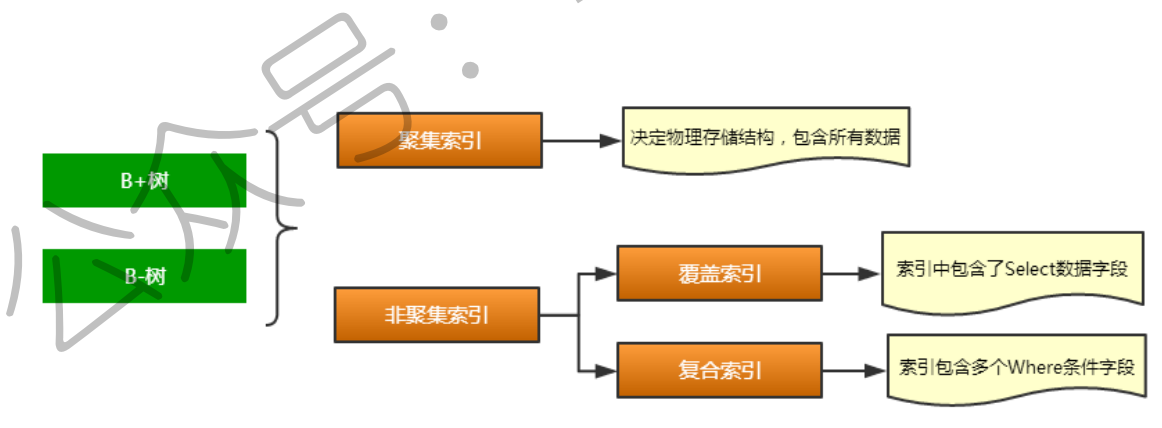
- 非聚集索引的创建会单独创建索引文件来存储索引结构，会占用一定存储空间，就是用空间换时间；
- 非聚集索引的目的很单纯：提高特定条件的查询效率，一个表有可能根据多种查询需求创建多个非聚集索引；

# 索引架构与查询



数据查询SQL简单来看，分为两个部分：**SELECT\*\*\*\*** 和 **Where \*\*\*\***，因此索引的创建也是根据这两部分来决定的。根据这两点，有两种主要的索引形式：**复合索引**和**覆盖索引**，在实际使用中，根据具体情况可能都会用到，只要能提高查询效率就是好索引。

**覆盖索引：**就是在索引中包含的数据列（非索引列，SELECT需要的列），这样在使用该索引查询数据时就不会再进行**键查找（也叫书签查找）**了。**复合索引：**主要针对Where中有多个条件的情况，索引包含多个数据列。在使用复合索引时，应注意多个索引键的顺序问题，这个是会影响查询效率的，一般的原则是**唯一性高的放前面**，还有就是SQL语句中Where条件的顺序应该和索引顺序一致。



## 索引碎片



前面说过了，索引在使用一段时间后（主要是新增、修改、删除数据，如果该页已经存储满了，就要进行页的拆分，频繁的拆分，会产生较多的索引碎片）会产生索引碎片，这就造成了索引页在磁盘上存储的不连续。会造成磁盘的访问使用的是随机的i/o，而不是顺序的i/o读取，这样访问索引页会变得



更慢。如果碎片过多，数据库是可能会不使用该索引的（她嫌弃你太慢了，数据库会选择一个更优的执行计划）。

解决这个问题主要是两种方法：

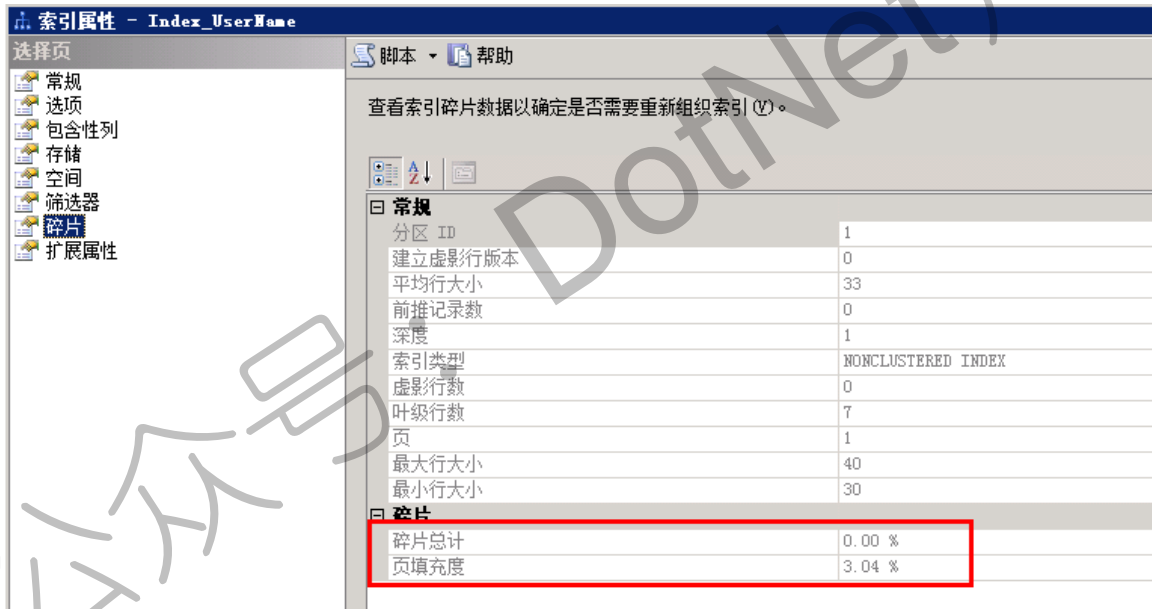
**第一种是预防：**设置页的**填充因子**

意思就是在页上设置一段空白区域，在新增数据的时候，可以使用这段空白区域，可以一定的避免页的拆分，从而减少索引碎片的产生。

填充因子就是用来描述这种页中填充数据的一个比例，一般默认是100%填充的。如果我们修改填充因子为80%，那么页在存储数据时，就会剩余20%的剩余空间，这样在下次插入的时候就不会拆分页了。那么是不是我们可以把填充因子设置低一点，留更多的剩余空间，不是很好嘛？当然也不好，填充因子设置的低，会需要分配更多的存储空间，叶子节点的深度会增加，这样是会影响查询效率的，因此，这是要根据实际情况而定的。

那么一般我们是怎么设置填充因子的呢，主要根据表的读写比例而定的。如果读的多，填充因子可以设置高一点，如100%，读写各一半，可以80~90%；修改多可以设置50~70%。

**第二种是索引修复：**定期对索引进行检查、维护，写一段SQL检查索引的碎片比例，如果碎片过多，进行碎片修复或重建，定期执行即可。具体可以参考本文末尾的相关参考资料。



|                            |                    |
|----------------------------|--------------------|
| 索引属性 - Index_UserName      |                    |
| 选择页                        |                    |
| 脚本 帮助                      |                    |
| 查看索引碎片数据以确定是否需要重新组织索引 (V)。 |                    |
| 常规                         |                    |
| 分区 ID                      | 1                  |
| 建立虚影行版本                    | 0                  |
| 平均行大小                      | 33                 |
| 前推记录数                      | 0                  |
| 深度                         | 1                  |
| 索引类型                       | NONCLUSTERED INDEX |
| 虚影行数                       | 0                  |
| 叶级行数                       | 7                  |
| 页                          | 1                  |
| 最大行大小                      | 40                 |
| 最小行大小                      | 30                 |
| 碎片                         |                    |
| 碎片总计                       | 0.00 %             |
| 页填充度                       | 3.04 %             |

**索引使用总结**



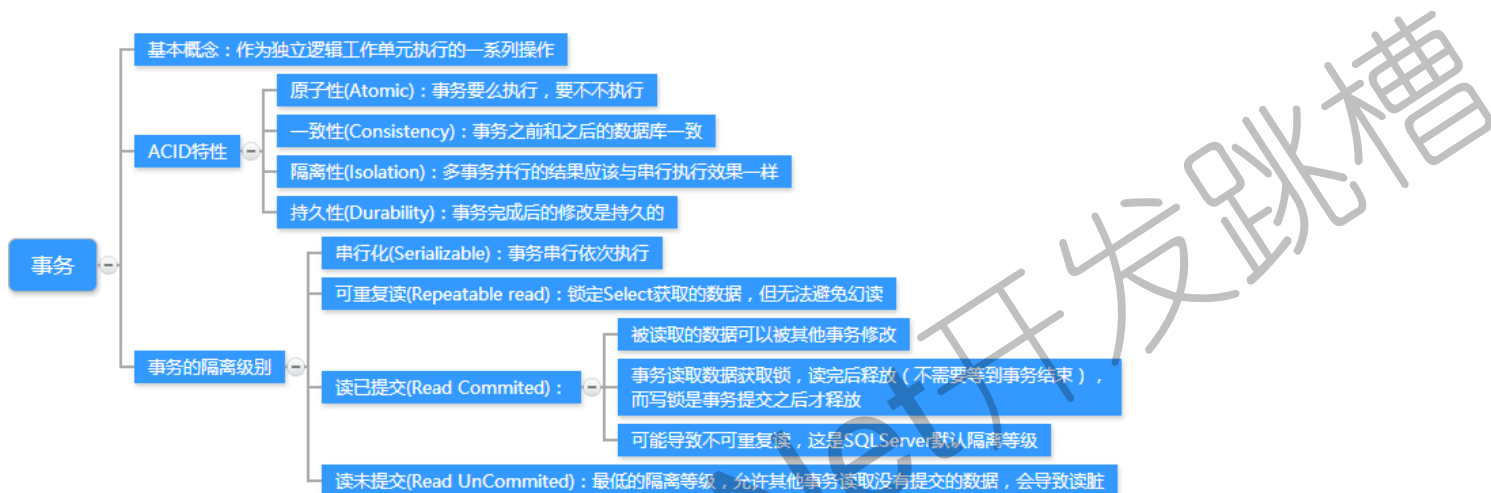
- 创建索引的的字段尽量小，最好是数值，比如整形int等；
- 对于频繁修改的字段，尽量不要创建索引，维护索引的成本很高，而且更容易产生索引碎片；
- 定期的索引维护，如索引碎片的修复等；
- 不要建立或维护不必要的重复索引，会增加修改数据（新增、修改、删除数据）的成本；
- 使用唯一性高的字段创建索引，切不可在性别这样的低唯一性的字段上创建索引；
- 在SQL语句中，尽量不要在Where条件中使用函数、运算符或表达式计算，会造成索引无法正常使用；
- 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描；

- 应尽量避免在 where 子句中使用!=或<>操作符，否则将导致引擎放弃使用索引而进行全表扫描；

## 事务与锁

事务就是作为一个逻辑工作单元的SQL语句，如果任何一个语句操作失败那么整个操作就被失败，以后操作就会回滚到操作前状态，或者是上个节点。为了确保要么执行，要么不执行，就可以使用事务。而锁是实现事务的关键，锁可以保证事务的完整性和并发性。

这部分的理论性太强了，不如看看下文章末尾的参考资料更好（博客园的高质量文章还是相当多的），简单总结一下就好了！



和在.NET中的锁用途类似，数据库中的锁也是为了解决在并发访问时出现各种冲突的一种机制。



## 题目答案解析:

## 1. 索引的作用？和它的优缺点是什么？

索引就是一种特殊的查询表，数据库的搜索引擎可以利用它加速对数据的检索。索引很类似与现实生活中书的目录，不需要查询整本书内容就可以找到想要的数据库。缺点是它减慢了数据录入的速度，同时也增加了数据库的尺寸大小。

## 2. 介绍存储过程基本概念和 她的优缺点

存储过程是一个预编译的SQL语句，他的优点是允许模块化的设计，也就是说只需创建一次，在该程序中就可以调用多次。例如某次操作需要执行多次SQL，就可以把这个SQL做一个存储过程，因为存储过程是预编译的，所以使用存储过程比单纯SQL语句执行要快。缺点是可移植性差，交互性差。

## 3. 使用索引有哪些需要注意的地方？

- 创建索引的的字段尽量小，最好是数值，比如整形int等；
- 对于频繁修改的字段，尽量不要创建索引，维护索引的成本很高，而且更容易产生索引碎片；
- 定期的索引维护，如索引碎片的修复等；
- 不要建立或维护不必要的重复索引，会增加修改数据（新增、修改、删除数据）的成本；
- 使用唯一性高的字段创建索引，切不可在性别这样的低唯一性的字段上创建索引；
- 在SQL语句中，尽量不要在Where条件中使用函数、运算符或表达式计算，会造成索引无法正常使用；
- 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描；
- 应尽量避免在 where 子句中使用!=或<>操作符，否则将导致引擎放弃使用索引而进行全表扫描；

## 4. 索引碎片是如何产生的？有什么危害？ 又该如何处理？

索引在使用一段时间后（主要是新增、修改、删除数据，如果该页已经存储满了，就要进行页的拆分，频繁的拆分，会产生较多的索引碎片）会产生索引碎片。

索引碎片会严重影响数据的查询效率，如果碎片太多，索引可能不会被使用。

碎片的处理方式主要有两种：

**第一种是预防：**设置页的填充因子

意思就是在页上设置一段空白区域，在新增数据的时候，可以使用这段空白区域，可以一定的避免页的拆分，从而减少索引碎片的产生。

填充因子就是用来描述这种页中填充数据的一个比例，一般默认是100%填充的。如果我们修改填充因子为80%，那么页在存储数据时，就会剩余20%的剩余空间，这样在下次插入的时候就不会拆分页了。那么是不是我们可以把填充因子设置低一点，留更多的剩余空间，不是很好嘛？当然也不好，填充因子设置的低，会需要分配更多的存储空间，叶子节点的深度会增加，这样是会影响查询效率的，因此，这是要根据实际情况而定的。

那么一般我们是怎么设置填充因子的呢，主要根据表的读写比例而定的。如果读的多，填充因子可以设置高一点，如100%，读写各一半，可以80~90%；修改多可以设置50~70%。

**第二种是索引修复：**定期对索引进行检查、维护，写一段SQL检查索引的碎片比例，如果碎片过多，进行碎片修复或重建，定期执行即可。具体可以参考本文末尾的相关参考资料。

## 5. 锁的目的是什么？

主要解决多个用户同时对数据库的并发操作时会带来以下数据不一致的问题：

- 丢失更新，同时修改一条数据
- 读脏，A修改了数据后，B读取后A又取消了修改，B读脏
- 不可重复读，A用户读取数据,随后B用户读取该数据并修改,此时A用户再读取数据时发现前后两次的值不一致
- 还有一种是幻读，这个情况好像不多。

并发控制的主要方法是封锁,锁就是在一段时间内禁止用户做某些操作以避免产生数据不一致

## 6. 锁的粒度有哪些？

- 数据库锁：锁定整个数据库，这通常发生在整个数据库模式改变的时候。
- 表锁：锁定整个表，这包含了与该表相关联的所有数据相关的对象，包括实际的数据行(每一行)以及与该表相关联的所有索引中的键。
- 区段锁：锁定整个区段，因为一个区段由8页组成，所以区段锁定是指锁定控制了区段、控制了该区段内8个数据或索引页以及这8页中的所有数据行。
- 页锁：锁定该页中的所有数据或索引键。
- 行或行标识符：虽然从技术上将，锁是放在行标识符上的，但是本质上，它锁定了整个数据行。

## 7. 什么是事务？什么是锁？

事务就是被绑定在一起作为一个逻辑工作单元的SQL语句分组，如果任何一个语句操作失败那么整个操作就被失败，以后操作就会回滚到操作前状态，或者是上个节点。为了确保要么执行，要么不执行，就可以使用事务。要将所有组语句作为事务考虑，就需要通过ACID测试，即原子性，一致性，隔离性和持久性。

锁是实现事务的关键，锁可以保证事务的完整性和并发性。

## 8. 视图的作用，视图可以更改么？

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询；不包含任何列或数据。使用视图可以简化复杂的sql操作，隐藏具体的细节，保护数据；视图创建后，可以使用与表相同的方式利用它们。

视图的目的在于简化检索，保护数据，并不用于更新。

## 9. 什么是触发器(trigger)？触发器有什么作用？

触发器是数据库中由一定时间触发的特殊的存储过程，他不是由程序调用也不是手工启动的。触发器的执行可以由对一个表的insert,delete, update等操作来触发，触发器经常用于加强数据的完整性约束和业务规则等等。

## 10. SQL里面IN比较快还是EXISTS比较快?

这个题不能一概而论，要根据具体情况来看。IN适合于外表大而内表小的情况；EXISTS适合于外表小而内表大的情况。

如果查询语句使用了not in，那么对内外表都进行全表扫描，没有用到索引；而not exists的子查询依然能用到表上的索引。所以无论哪个表大，用not exists都比not in 要快。参考资料：<http://www.cnblogs.com/seasons1987/archive/2013/07/03/3169356.html>

## 11. 维护数据库的完整性和一致性，你喜欢用触发器还是自写业务逻辑?为什么?

尽可能使用约束，如check、主键、外键、非空字段等来约束。这样做效率最高，也最方便。其次是使用触发器，这种方法可以保证，无论什么业务系统访问数据库都可以保证数据的完整性和一致性。最后考虑的是自写业务逻辑，但这样做麻烦，编程复杂，效率低下。

更多初中高面试题可以扫码关注公众号↓↓↓↓↓



北京群 群1: 219690210, 群2: 377501688, 群3: 262827065, 成都群: 209844460  
上海群: 376029918 杭州群: 376029918 广州群: 344744167 深圳群: 542733289  
西安群: 542733289 高级群: 165150112  
微信公众号: DotNet开发跳槽