第 2 章

面试需要的基础知识

2.1 面试官谈基础知识

"C++的基础知识，如面向对象的特性、构造函数、析构函数、动态绑定等，能够反映出应聘者是否善于把握问题本质，有没有耐心深入一个问题。另外还有常用的设计模式、UML 图等，这些都能体现应聘者是否有软件工程方面的经验。"

——王海波（Autodesk，软件工程师）

"对基础知识的考查我特别重视 C++中对内存的使用管理。我觉得内存管理是 C++程序员特别要注意的，因为内存的使用和管理会影响程序的效率和稳定性。"

——蓝诚（Autodesk，软件工程师）

"基础知识反映了一个人的基本能力和基础素质，是以后工作中最核心的能力要求。我一般考查：（1）数据结构和算法；（2）编程能力；（3）部分数学知识，如概率；（4）问题的分析和推理能力。"

——张晓禹（百度，技术经理）

“我比较重视四块基础知识：（1）编程基本功（特别喜欢字符串处理这一类的问题）；（2）并发控制；（3）算法、复杂度；（4）语言的基本概念。”

——张珺（百度，高级软件工程师）

“我会考查编程基础、计算机系统基础知识、算法及设计能力。这些是成为一个软件工程师的最基本的要求，这些方面表现出色的人，我们一般认为是有发展潜力的。”

——韩伟东（盛大，高级研究员）

“（1）对 OS 的理解程度。这些知识对于工作中常遇到的内存管理、文件操作、程序性能、多线程、程序安全等有重要帮助。对于 OS 理解比较深入的人对于偏底层的工作上手一般比较快。（2）对于一门编程语言的掌握程度。一个热爱编程的人应该会对某种语言有比较深入的了解。通常这样的人对于新的编程语言上手也比较快，而且理解比较深入。（3）常用的算法和数据结构。不了解这些的程序员基本只能写写‘Hello World’。”

——陈黎明（微软，SDE II）

2.2 编程语言

程序员写代码总是基于某一种编程语言，因此技术面试的时候都会直接或者间接涉及至少一种编程语言。在面试过程中，面试官要么直接问语言的语法，要么让应聘者用一种编程语言写代码解决一个问题，通过写出的代码来判断应聘者对他使用的语言的掌握程度。现在流行的编程语言很多，不同公司开发用的语言也不尽相同。做底层开发比如经常写驱动的人更习惯用 C，Linux 下有很多程序员用 C++开发应用程序，基于 Windows 的 C#项目已经越来越多，跨平台开发的程序员则可能更喜欢 Java，随着苹果 iPad、iPhone 的热销已经有很多程序员投向了 Objective C 的阵营，同时还有很多人喜欢用脚本语言如 Perl、Python 开发短小精致的小应用软件。

因此，不同公司面试的时候对编程语言的要求也有所不同。每一种编程语言都可以写出一本大部头的书籍，本书限于篇幅不可能面面俱到。本书中所有代码都用 C/C++/C#实现，下面简要介绍一些 C++/C#常见的面试题。

2.2.1　C++

国内绝大部分高校都开设 C++的课程，因此绝大部分程序员都学过 C++，于是 C++成了各公司面试的首选编程语言。包括 Autodesk 在内的很多公司在面试的时候会有大量的 C++的语法题，其他公司虽然不直接面试 C++的语法，但面试题要求用 C++实现算法。因此，总的来说，应聘者不管去什么公司求职，都应该在一定程度上掌握 C++。

通常语言面试有 3 种类型。第一种题型是面试官直接询问应聘者对 C++概念的理解。这种类型的问题，面试官特别喜欢了解应聘者对 C++关键字的理解程度。例如，在 C++中，有哪 4 个与类型转换相关的关键字？这些关键字各有什么特点，应该在什么场合下使用？

在这种类型的题目中，sizeof 是经常被问到的一个概念。比如下面的面试片段，就反复出现在各公司的技术面试中。

面试官：定义一个空的类型，里面没有任何成员变量和成员函数。对该类型求 sizeof，得到的结果是多少？

应聘者：答案是 1。

面试官：为什么不是 0？

应聘者：空类型的实例中不包含任何信息，本来求 sizeof 应该是 0，但是当我们声明该类型的实例的时候，它必须在内存中占有一定的空间，否则无法使用这些实例。至于占用多少内存，由编译器决定。在 Visual Studio 中，每个空类型的实例占用 1 字节的空间。

面试官：如果在该类型中添加一个构造函数和析构函数，再对该类型求 sizeof，得到的结果又是多少？

应聘者：和前面一样，还是 1。调用构造函数和析构函数只需要知道函数的地址即可，而这些函数的地址只与类型相关，而与类型的实例无关，编译器也不会因为这两个函数而在实例内添加任何额外的信息。

面试官：那如果把析构函数标记为虚函数呢？

应聘者：C++的编译器一旦发现一个类型中有虚函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向虚函数表的指针。在 32 位的机器上，一个指针占 4 字节的空间，因此求 sizeof 得到 4；如果是 64 位的机器，则一个指针占 8 字节的空间，因此求 sizeof 得到 8。

第二种题型就是面试官拿出事先准备好的代码，让应聘者分析代码的运行结果。这种题型选择的代码通常包含比较复杂微妙的语言特性，这要求应聘者对 C++考点有着透彻的理解。即使应聘者对考点有一点点模糊，那么最终他得到的结果和实际运行的结果可能就会差距甚远。

比如，面试官递给应聘者一张有如下代码的 A4 打印纸要求他分析编译运行的结果，并提供 3 个选项：A. 编译错误；B. 编译成功，运行时程序崩溃；C. 编译运行正常，输出 10。

```
class A
{
private:
    int value;

public:
    A(int n) { value = n; }
    A(A other) { value = other.value; }

    void Print() { std::cout << value << std::endl; }
};

int _tmain(int argc, _TCHAR* argv[])
{
    A a = 10;
    A b = a;
    b.Print();

    return 0;
}
```

在上述代码中，复制构造函数 A(A other)传入的参数是 A 的一个实例。由于是传值参数，我们把形参复制到实参会调用复制构造函数。因此，如果允许复制构造函数传值，就会在复制构造函数内调用复制构造函数，就会形成永无休止的递归调用从而导致栈溢出。因此，C++的标准不允许复制构造函数传值参数，在 Visual Studio 和 GCC 中，都将编译出错。要解决这个问题，我们可以把构造函数修改为 A(const A& other)，也就是把传值参数改成常量引用。

第三种题型就是要求应聘者写代码定义一个类型或者实现类型中的成员函数。让应聘者写代码的难度自然比让应聘者分析代码要高不少，因

为能想明白的未必就能写得清楚。很多考查 C++语法的代码题重点考查构造函数、析构函数及运算符重载，比如面试题 1“赋值运算符函数”就是一个例子。

为了让大家能顺利地通过 C++面试，更重要的是能更好地学习掌握 C++这门编程语言，这里推荐几本 C++的书，大家可以根据自己的具体情况选择阅读的顺序。

- **《Effective C++》**：这本书很适合在面试之前突击 C++。这本书列举了使用 C++经常出现的问题及解决这些问题的技巧。该书中提到的问题也是面试官很喜欢问的问题。
- **《C++ Primer》**：读完这本书，就会对 C++的语法有全面的了解。
- **《深度探索 C++对象模型》**：这本书有助于我们深入了解 C++对象的内部。读懂这本书后，很多 C++难题，比如前面的 sizeof 的问题、虚函数的调用机制等，都会变得很容易。
- **《The C++ Programming Language》**：如果想全面深入掌握 C++，那么没有哪本书比这本书更适合的了。

面试题 1：赋值运算符函数

题目：如下为类型 CMyString 的声明，请为该类型添加赋值运算符函数。

```
class CMyString
{
public:
    CMyString(char* pData = nullptr);
    CMyString(const CMyString& str);
    ~CMyString(void);

private:
    char* m_pData;
};
```

当面试官要求应聘者定义一个赋值运算符函数时，他会在检查应聘者写出的代码时关注如下几点：

- 是否把返回值的类型声明为该类型的引用，并在函数结束前返回实例自身的引用（*this）。只有返回一个引用，才可以允许连续赋值。否则，如果函数的返回值是 void，则应用该赋值运算符将不能进行连续赋值。假设有 3 个 CMyString 的对象：str1、str2 和 str3，在程

序中语句 str1=str2=str3 将不能通过编译。

- 是否把传入的参数的类型声明为常量引用。如果传入的参数不是引用而是实例，那么从形参到实参会调用一次复制构造函数。把参数声明为引用可以避免这样的无谓消耗，能提高代码的效率。同时，我们在赋值运算符函数内不会改变传入的实例的状态，因此应该为传入的引用参数加上 const 关键字。
- 是否释放实例自身已有的内存。如果我们忘记在分配新内存之前释放自身已有的空间，则程序将出现内存泄漏。
- 判断传入的参数和当前的实例（*this）是不是同一个实例。如果是同一个，则不进行赋值操作，直接返回。如果事先不判断就进行赋值，那么在释放实例自身内存的时候就会导致严重的问题：当*this 和传入的参数是同一个实例时，一旦释放了自身的内存，传入的参数的内存也同时被释放了，因此再也找不到需要赋值的内容了。

❖ 经典的解法，适用于初级程序员

当我们完整地考虑了上述 4 个方面之后，可以写出如下的代码：

```
CMyString& CMyString::operator =(const CMyString &str)
{
    if(this == &str)
        return *this;

    delete []m_pData;
    m_pData = nullptr;

    m_pData = new char[strlen(str.m_pData) + 1];
    strcpy(m_pData, str.m_pData);

    return *this;
}
```

这是一般 C++教材上提供的参考代码。如果接受面试的是应届毕业生或者 C++初级程序员，能全面地考虑到前面 4 点并完整地写出代码，那么面试官可能会让他通过这轮面试。但如果面试的是 C++高级程序员，则面试官可能会提出更高的要求。

❖ 考虑异常安全性的解法，高级程序员必备

在前面的函数中，我们在分配内存之前先用 delete 释放了实例 m_pData

的内存。如果此时内存不足导致 new char 抛出异常，则 m_pData 将是一个空指针，这样非常容易导致程序崩溃。也就是说，一旦在赋值运算符函数内部抛出一个异常，CMyString 的实例不再保持有效的状态，这就违背了异常安全性（Exception Safety）原则。

要想在赋值运算符函数中实现异常安全性，我们有两种方法。一种简单的办法是我们先用 new 分配新内容，再用 delete 释放已有的内容。这样只在分配内容成功之后再释放原来的内容，也就是当分配内存失败时我们能确保 CMyString 的实例不会被修改。我们还有一种更好的办法，即先创建一个临时实例，再交换临时实例和原来的实例。下面是这种思路的参考代码：

```
CMyString& CMyString::operator =(const CMyString &str)
{
    if(this != &str)
    {
        CMyString strTemp(str);

        char* pTemp = strTemp.m_pData;
        strTemp.m_pData = m_pData;
        m_pData = pTemp;
    }

    return *this;
}
```

在这个函数中，我们先创建一个临时实例 strTemp，接着把 strTemp.m_pData 和实例自身的 m_pData 进行交换。由于 strTemp 是一个局部变量，但程序运行到 if 的外面时也就出了该变量的作用域，就会自动调用 strTemp 的析构函数，把 strTemp.m_pData 所指向的内存释放掉。由于 strTemp.m_pData 指向的内存就是实例之前 m_pData 的内存，这就相当于自动调用析构函数释放实例的内存。

在新的代码中，我们在 CMyString 的构造函数里用 new 分配内存。如果由于内存不足抛出诸如 bad_alloc 等异常，但我们还没有修改原来实例的状态，因此实例的状态还是有效的，这也就保证了异常安全性。

如果应聘者在面试的时候能够考虑到这个层面，面试官就会觉得他对代码的异常安全性有很深的理解，那么他自然也就能通过这轮面试了。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/01_AssignmentOperator

测试用例：

- 把一个 CMyString 的实例赋值给另外一个实例。
- 把一个 CMyString 的实例赋值给它自己。
- 连续赋值。

本题考点：

- 考查应聘者对 C++基础语法的理解，如运算符函数、常量引用等。
- 考查应聘者对内存泄漏的理解。
- 对于高级 C++程序员，面试官还将考查应聘者对代码异常安全性的理解。

2.2.2 C#

C#是微软在推出新的开发平台.NET 时同步推出的编程语言。由于 Windows 至今仍然是用户最多的操作系统，而.NET 又是微软近年来力推的开发平台，因此 C#无论是在桌面软件还是在网络应用的开发上都有着广泛的应用，所以我们也不难理解为什么现在很多基于 Windows 系统开发的公司都会要求应聘者掌握 C#。

C#可以看成一门以 C++为基础发展起来的托管语言，因此它的很多关键字甚至语法都和 C++类似。对于一个学习过 C++编程的程序员而言，他用不了多长时间的学习就能用 C#来开发软件。然而我们也要清醒地认识到，虽然学习 C#与 C++相同或者类似的部分很容易，但要掌握并区分两者不同的地方却不是一件很容易的事情。面试官总是喜欢深究我们模棱两可的地方以考查我们是不是真的理解了，因此我们要着重注意 C#与 C++不同的语法特点。下面的面试片段就是一个例子。

面试官： 在 C++中可以用 struct 和 class 来定义类型。这两种类型有什么区别？

应聘者：如果没有标明成员函数或者成员变量的访问权限级别，那么在 struct 中默认的是 public，而在 class 中默认的是 private。

面试官：那在 C#中呢？

应聘者：C#和 C++不一样。在 C#中如果没有标明成员函数或者成员变量的访问权限级别，则在 struct 和 class 中都是 private 的。struct 和 class 的区别是 struct 定义的是值类型，值类型的实例在栈上分配内存；而 class 定义的是引用类型，引用类型的实例在堆上分配内存。

和 C++一样，在 C#中，每个类型中都有构造函数。但和 C++不同的是，我们在 C#中可以为类型定义一个 Finalizer 和 Dispose 方法以释放资源。Finalizer 方法虽然写法与 C++的析构函数看起来一样，都是~后面跟类型名字，但与 C++析构函数的调用时机是确定的不同，C#的 Finalizer 是在运行时（CLR）进行垃圾回收时才会被调用的，它的调用时机是由运行时决定的，因此对程序员来说是不确定的。另外，在 C#中可以为类型定义一个特殊的构造函数：静态构造函数。这个函数的特点是在类型第一次被使用之前由运行时自动调用，而且保证只调用一次。关于静态构造函数，我们有很多有意思的面试题，比如运行下面的 C#代码，输出的结果是什么？

```
class A
{
    public A(string text)
    {
        Console.WriteLine(text);
    }
}

class B
{
    static A a1 = new A("a1");
    A a2 = new A("a2");

    static B()
    {
        a1 = new A("a3");
    }

    public B()
    {
        a2 = new A("a4");
    }
}

class Program
{
    static void Main(string[] args)
```

```
    {
        B b = new B();
    }
}
```

在调用类型 B 的代码之前先执行 B 的静态构造函数。静态构造函数先初始化类型的静态变量，再执行函数体内的语句。因此，先打印 a1 再打印 a3。接下来执行 B b = new B()，即调用 B 的普通构造函数。构造函数先初始化成员变量，再执行函数体内的语句，因此先后打印出 a2、a4。因此，运行上面的代码，得到的结果将是打印出 4 行，分别是 a1、a3、a2、a4。

我们除了要关注 C#和 C++不同的知识点，还要格外关注 C#一些特有的功能，比如反射、应用程序域（AppDomain）等。这些概念还相互关联，要花很多时间学习研究才能透彻地理解它们。下面就是一段关于反射和应用程序域的代码，运行它得到的结果是什么？

```
[Serializable]
internal class A : MarshalByRefObject
{
    public static int Number;

    public void SetNumber(int value)
    {
        Number = value;
    }
}

[Serializable]
internal class B
{
    public static int Number;

    public void SetNumber(int value)
    {
        Number = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        String assambly = Assembly.GetEntryAssembly().FullName;
        AppDomain domain = AppDomain.CreateDomain("NewDomain");

        A.Number = 10;
        String nameOfA = typeof(A).FullName;
        A a = domain.CreateInstanceAndUnwrap(assambly, nameOfA) as A;
        a.SetNumber(20);
        Console.WriteLine("Number in class A is {0}", A.Number);
```

```
            B.Number = 10;
            String nameOfB = typeof(B).FullName;
            B b = domain.CreateInstanceAndUnwrap(assambly, nameOfB) as B;
            b.SetNumber(20);
            Console.WriteLine("Number in class B is {0}", B.Number);
        }
}
```

上述 C#代码先创建一个名为 NewDomain 的应用程序域，并在该域中利用反射机制创建类型 A 的一个实例和类型 B 的一个实例。我们注意到类型 A 继承自 MarshalByRefObject，而 B 不是。虽然这两个类型的结构一样，但由于基类不同而导致在跨越应用程序域的边界时表现出的行为将大不相同。

先考虑 A 的情况。由于 A 继承自 MarshalByRefObject，那么 a 实际上只是在默认的域中的一个代理实例（Proxy），它指向位于 NewDomain 域中的 A 的一个实例。当调用 a 的方法 SetNumber 时，是在 NewDomain 域中调用该方法的，它将修改 NewDomain 域中静态变量 A.Number 的值并设为 20。由于静态变量在每个应用程序域中都有一份独立的拷贝，修改 NewDomain 域中的静态变量 A.Number 对默认域中的静态变量 A.Number 没有任何影响。由于 Console.WriteLine 是在默认的应用程序域中输出 A.Number，因此输出仍然是 10。

接着讨论 B。由于 B 只是从 Object 继承而来的类型，它的实例穿越应用程序域的边界时，将会完整地复制实例。因此，在上述代码中，我们尽管试图在 NewDomain 域中生成 B 的实例，但会把实例 b 复制到默认的应用程序域。此时调用方法 b.SetNumber 也是在默认的应用程序域上进行，它将修改默认的域上的 B.Number 并设为 20。再在默认的域上调用 Console.WriteLine 时，它将输出 20。

下面推荐两本与 C#相关的书籍，以方便大家应对 C#面试并学习好 C#。

- **《Professional C#》**：这本书最大的特点是在附录中有几章专门写给已经有其他语言（如 VB、C++和 Java）经验的程序员，它详细讲述了 C#和其他语言的区别，看了这几章之后就不会把 C#和之前掌握的语言相混淆。
- Jeffrey Richter 的**《CLR Via C#》**：该书不仅深入地介绍了 C#语言，

同时对 CLR 及.NET 进行了全面的剖析。如果能够读懂这本书，那么我们就能深入理解装箱卸箱、垃圾回收、反射等概念，知其然的同时也能知其所以然，通过与 C#相关的面试自然也就不难了。

面试题 2：实现 Singleton 模式

> 题目：设计一个类，我们只能生成该类的一个实例。

只能生成一个实例的类是实现了 Singleton（单例）模式的类型。由于设计模式在面向对象程序设计中起着举足轻重的作用，在面试过程中很多公司都喜欢问一些与设计模式相关的问题。在常用的模式中，Singleton 是唯一一个能够用短短几十行代码完整实现的模式。因此，写一个 Singleton 的类型是一个很常见的面试题。

❖ 不好的解法一：只适用于单线程环境

由于要求只能生成一个实例，因此我们必须把构造函数设为私有函数以禁止他人创建实例。我们可以定义一个静态的实例，在需要的时候创建该实例。下面定义类型 Singleton1 就是基于这个思路的实现：

```
public sealed class Singleton1
{
    private Singleton1()
    {
    }

    private static Singleton1 instance = null;
    public static Singleton1 Instance
    {
        get
        {
            if (instance == null)
                instance = new Singleton1();

            return instance;
        }
    }
}
```

上述代码在 Singleton1 的静态属性 Instance 中，只有在 instance 为 null 的时候才创建一个实例以避免重复创建。同时我们把构造函数定义为私有函数，这样就能确保只创建一个实例。

❖ 不好的解法二：虽然在多线程环境中能工作，但效率不高

解法一中的代码在单线程的时候工作正常，但在多线程的情况下就有问题了。设想如果两个线程同时运行到判断 instance 是否为 null 的 if 语句，并且 instance 的确没有创建时，那么两个线程都会创建一个实例，此时类型 Singleton1 就不再满足单例模式的要求了。为了保证在多线程环境下我们还是只能得到类型的一个实例，需要加上一个同步锁。把 Singleton1 稍作修改得到了如下代码：

```
public sealed class Singleton2
{
    private Singleton2()
    {
    }

    private static readonly object syncObj = new object();

    private static Singleton2 instance = null;
    public static Singleton2 Instance
    {
        get
        {
            lock (syncObj)
            {
                if (instance == null)
                    instance = new Singleton2();
            }

            return instance;
        }
    }
}
```

我们还是假设有两个线程同时想创建一个实例。由于在一个时刻只有一个线程能得到同步锁，当第一个线程加上锁时，第二个线程只能等待。当第一个线程发现实例还没有创建时，它创建出一个实例。接着第一个线程释放同步锁，此时第二个线程可以加上同步锁，并运行接下来的代码。这时候由于实例已经被第一个线程创建出来了，第二个线程就不会重复创建实例了，这样就保证了我们在多线程环境中也只能得到一个实例。

但是类型 Singleton2 还不是很完美。我们每次通过属性 Instance 得到 Singleton2 的实例，都会试图加上一个同步锁，而加锁是一个非常耗时的操作，在没有必要的时候我们应该尽量避免。

❖ 可行的解法：加同步锁前后两次判断实例是否已存在

我们只是在实例还没有创建之前需要加锁操作，以保证只有一个线程创建出实例。而当实例已经创建之后，我们已经不需要再执行加锁操作了。于是我们可以把解法二中的代码再做进一步的改进：

```
public sealed class Singleton3
{
    private Singleton3()
    {
    }

    private static object syncObj = new object();

    private static Singleton3 instance = null;
    public static Singleton3 Instance
    {
        get
        {
            if (instance == null)
            {
                lock (syncObj)
                {
                    if (instance == null)
                        instance = new Singleton3();
                }
            }

            return instance;
        }
    }
}
```

Singleton3 中只有当 instance 为 null 即没有创建时，需要加锁操作。当 instance 已经创建出来之后，则无须加锁。因为只在第一次的时候 instance 为 null，因此只在第一次试图创建实例的时候需要加锁。这样 Singleton3 的时间效率比 Singleton2 要好很多。

Singleton3 用加锁机制来确保在多线程环境下只创建一个实例，并且用两个 if 判断来提高效率。这样的代码实现起来比较复杂，容易出错，我们还有更加优秀的解法。

❖ 强烈推荐的解法一：利用静态构造函数

C#的语法中有一个函数能够确保只调用一次，那就是静态构造函数，我们可以利用 C#的这个特性实现单例模式。

```
public sealed class Singleton4
{
    private Singleton4()
    {
    }

    private static Singleton4 instance = new Singleton4();
    public static Singleton4 Instance
    {
        get
        {
            return instance;
        }
    }
}
```

Singleton4 的实现代码非常简洁。我们在初始化静态变量 instance 的时候创建一个实例。由于 C#是在调用静态构造函数时初始化静态变量，.NET 运行时能够确保只调用一次静态构造函数，这样我们就能够保证只初始化一次 instance。

C#中调用静态构造函数的时机不是由程序员掌控的，而是当.NET 运行时发现第一次使用一个类型的时候自动调用该类型的静态构造函数。因此在 Singleton4 中，实例 instance 并不是在第一次调用属性 Singleton4.Instance 的时候被创建的，而是在第一次用到 Singleton4 的时候就会被创建。假设我们在 Singleton4 中添加一个静态方法，调用该静态函数是不需要创建一个实例的，但如果按照 Singleton4 的方式实现单例模式，则仍然会过早地创建实例，从而降低内存的使用效率。

❖ 强烈推荐的解法二：实现按需创建实例

最后一个实现 Singleton5 则很好地解决了 Singleton4 中的实例创建时机过早的问题。

```
public sealed class Singleton5
{
    Singleton5()
    {
    }

    public static Singleton5 Instance
    {
        get
        {
            return Nested.instance;
        }
    }
```

```
    class Nested
    {
        static Nested()
        {
        }

        internal static readonly Singleton5 instance = new Singleton5();
    }
}
```

在上述 Singleton5 的代码中，我们在内部定义了一个私有类型 Nested。当第一次用到这个嵌套类型的时候，会调用静态构造函数创建 Singleton5 的实例 instance。类型 Nested 只在属性 Singleton5.Instance 中被用到，由于其私有属性，他人无法使用 Nested 类型。因此，当我们第一次试图通过属性 Singleton5.Instance 得到 Singleton5 的实例时，会自动调用 Nested 的静态构造函数创建实例 instance。如果我们不调用属性 Singleton5.Instance，就不会触发.NET 运行时调用 Nested，也不会创建实例，这样就真正做到了按需创建。

❖ 解法比较

在前面的 5 种实现单例模式的方法中，第一种方法在多线程环境中不能正常工作，第二种模式虽然能在多线程环境中正常工作，但时间效率很低，都不是面试官期待的解法。在第三种方法中，我们通过两次判断一次加锁确保在多线程环境中能高效率地工作。第四种方法利用 C#的静态构造函数的特性，确保只创建一个实例。第五种方法利用私有嵌套类型的特性，做到只在真正需要的时候才会创建实例，提高空间使用效率。如果在面试中给出第四种或者第五种解法，则毫无疑问会得到面试官的青睐。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/02_Singleton

本题考点：

- 考查应聘者对单例（Singleton）模式的理解。

- 考查应聘者对 C#基础语法的理解，如静态构造函数等。
- 考查应聘者对多线程编程的理解。

本题扩展：

在前面的代码中，5 种单例模式的实现把类型标记为 sealed，表示它们不能作为其他类型的基类。现在我们要求定义一个表示总统的类型 President，可以从该类型继承出 FrenchPresident 和 AmericanPresident 等类型。这些派生类型都只能产生一个实例。请问该如何设计实现这些类型？

2.3 数据结构

数据结构一直是技术面试的重点，大多数面试题都是围绕着数组、字符串、链表、树、栈及队列这几种常见的数据结构展开的，因此每一个应聘者都要熟练掌握这几种数据结构。

数组和字符串是两种最基本的数据结构，它们用连续内存分别存储数字和字符。链表和树是面试中出现频率最高的数据结构。由于操作链表和树需要操作大量的指针，应聘者在解决相关问题的时候一定要留意代码的鲁棒性，否则容易出现程序崩溃的问题。栈是一个与递归紧密相关的数据结构，同样队列也与广度优先遍历算法紧密相关，深刻理解这两种数据结构能帮助我们解决很多算法问题。

2.3.1 数组

数组可以说是最简单的一种数据结构，它占据一块连续的内存并按照顺序存储数据。创建数组时，我们需要首先指定数组的容量大小，然后根据大小分配内存。即使我们只在数组中存储一个数字，也需要为所有的数据预先分配内存。因此数组的空间效率不是很好，经常会有空闲的区域没有得到充分利用。

由于数组中的内存是连续的，于是可以根据下标在 $O(1)$时间读/写任何元素，因此它的时间效率是很高的。我们可以根据数组时间效率高的优点，用数组来实现简单的哈希表：把数组的下标设为哈希表的键值（Key），而

把数组中的每一个数字设为哈希表的值（Value），这样每一个下标及数组中该下标对应的数字就组成了一个“键值-值”的配对。有了这样的哈希表，我们就可以在 $O(1)$时间内实现查找，从而快速、高效地解决很多问题。面试题 50“第一个只出现一次的字符”就是一个很好的例子。

为了解决数组空间效率不高的问题，人们又设计实现了多种动态数组，比如 C++的 STL 中的 vector。为了避免浪费，我们先为数组开辟较小的空间，然后往数组中添加数据。当数据的数目超过数组的容量时，我们再重新分配一块更大的空间（STL 的 vector 每次扩充容量时，新的容量都是前一次的两倍），把之前的数据复制到新的数组中，再把之前的内存释放，这样就能减少内存的浪费。但我们也注意到每一次扩充数组容量时都有大量的额外操作，这对时间性能有负面影响，因此使用动态数组时要尽量减少改变数组容量大小的次数。

在 C/C++中，数组和指针是既相互关联又有区别的两个概念。当我们声明一个数组时，其数组的名字也是一个指针，该指针指向数组的第一个元素。我们可以用一个指针来访问数组。但值得注意的是，C/C++没有记录数组的大小，因此在用指针访问数组中的元素时，程序员要确保没有超出数组的边界。下面通过一个例子来了解数组和指针的区别。运行下面的代码，请问输出是什么？

```
int GetSize(int data[])
{
    return sizeof(data);
}

int _tmain(int argc, _TCHAR* argv[])
{
    int data1[] = {1, 2, 3, 4, 5};
    int size1 = sizeof(data1);

    int* data2 = data1;
    int size2 = sizeof(data2);

    int size3 = GetSize(data1);

    printf("%d, %d, %d", size1, size2, size3);
}
```

答案是输出“20, 4, 4”。data1 是一个数组，sizeof(data1)是求数组的大小。这个数组包含 5 个整数，每个整数占 4 字节，因此共占用 20 字节。data2 声明为指针，尽管它指向了数组 data1 的第一个数字，但它的本质仍然是一个指针。在 32 位系统上，对任意指针求 sizeof，得到的结果都是 4。在 C/C++

中，当数组作为函数的参数进行传递时，数组就自动退化为同类型的指针。因此，尽管函数 GetSize 的参数 data 被声明为数组，但它会退化为指针，size3 的结果仍然是 4。

面试题 3：数组中重复的数字

> 题目一：找出数组中重复的数字。
>
> 在一个长度为 n 的数组里的所有数字都在 $0 \sim n-1$ 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。例如，如果输入长度为 7 的数组{2, 3, 1, 0, 2, 5, 3}，那么对应的输出是重复的数字 2 或者 3。

解决这个问题的一个简单的方法是先把输入的数组排序。从排序的数组中找出重复的数字是一件很容易的事情，只需要从头到尾扫描排序后的数组就可以了。排序一个长度为 n 的数组需要 $O(n\log n)$的时间。

还可以利用哈希表来解决这个问题。从头到尾按顺序扫描数组的每个数字，每扫描到一个数字的时候，都可以用 $O(1)$的时间来判断哈希表里是否已经包含了该数字。如果哈希表里还没有这个数字，就把它加入哈希表。如果哈希表里已经存在该数字，就找到一个重复的数字。这个算法的时间复杂度是 $O(n)$，但它提高时间效率是以一个大小为 $O(n)$的哈希表为代价的。我们再看看有没有空间复杂度是 $O(1)$的算法。

我们注意到数组中的数字都在 $0 \sim n-1$ 的范围内。如果这个数组中没有重复的数字，那么当数组排序之后数字 i 将出现在下标为 i 的位置。由于数组中有重复的数字，有些位置可能存在多个数字，同时有些位置可能没有数字。

现在让我们重排这个数组。从头到尾依次扫描这个数组中的每个数字。当扫描到下标为 i 的数字时，首先比较这个数字（用 m 表示）是不是等于 i。如果是，则接着扫描下一个数字；如果不是，则再拿它和第 m 个数字进行比较。如果它和第 m 个数字相等，就找到了一个重复的数字（该数字在下标为 i 和 m 的位置都出现了）；如果它和第 m 个数字不相等，就把第 i 个数字和第 m 个数字交换，把 m 放到属于它的位置。接下来再重复这个比较、交换的过程，直到我们发现一个重复的数字。

以数组{2, 3, 1, 0, 2, 5, 3}为例来分析找到重复数字的步骤。数组的第 0

个数字（从 0 开始计数，和数组的下标保持一致）是 2，与它的下标不相等，于是把它和下标为 2 的数字 1 交换。交换之后的数组是{1, 3, 2, 0, 2, 5, 3}。此时第 0 个数字是 1，仍然与它的下标不相等，继续把它和下标为 1 的数字 3 交换，得到数组{3, 1, 2, 0, 2, 5, 3}。接下来继续交换第 0 个数字 3 和第 3 个数字 0，得到数组{0, 1, 2, 3, 2, 5, 3}。此时第 0 个数字的数值为 0，接着扫描下一个数字。在接下来的几个数字中，下标为 1、2、3 的 3 个数字分别为 1、2、3，它们的下标和数值都分别相等，因此不需要执行任何操作。接下来扫描到下标为 4 的数字 2。由于它的数值与它的下标不相等，再比较它和下标为 2 的数字。注意到此时数组中下标为 2 的数字也是 2，也就是数字 2 在下标为 2 和下标为 4 的两个位置都出现了，因此找到一个重复的数字。

上述思路可以用如下代码实现：

```
bool duplicate(int numbers[], int length, int* duplication)
{
    if(numbers == nullptr || length <= 0)
    {
        return false;
    }

    for(int i = 0; i < length; ++i)
    {
        if(numbers[i] < 0 || numbers[i] > length - 1)
            return false;
    }

    for(int i = 0; i < length; ++i)
    {
        while(numbers[i] != i)
        {
            if(numbers[i] == numbers[numbers[i]])
            {
                *duplication = numbers[i];
                return true;
            }

            // swap numbers[i] and numbers[numbers[i]]
            int temp = numbers[i];
            numbers[i] = numbers[temp];
            numbers[temp] = temp;
        }
    }

    return false;
}
```

在上述代码中，找到的重复数字通过参数 duplication 传给函数的调用

者，而函数的返回值表示数组中是否有重复的数字。当输入的数组中存在重复的数字时，返回 true；否则返回 false。

代码中尽管有一个两重循环，但每个数字最多只要交换两次就能找到属于它自己的位置，因此总的时间复杂度是 $O(n)$。另外，所有的操作步骤都是在输入数组上进行的，不需要额外分配内存，因此空间复杂度为 $O(1)$。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/03_01_DuplicationInArray

测试用例：

- 长度为 n 的数组里包含一个或多个重复的数字。
- 数组中不包含重复的数字。
- 无效输入测试用例（输入空指针；长度为 n 的数组中包含 0～n−1 之外的数字）。

本题考点：

- 考查应聘者对一维数组的理解及编程能力。一维数组在内存中占据连续的空间，因此我们可以根据下标定位对应的元素。
- 考查应聘者分析问题的能力。当应聘者发现问题比较复杂时，能不能通过具体的例子找出其中的规律，是能否解决这个问题的关键所在。

题目二：不修改数组找出重复的数字。

在一个长度为 n+1 的数组里的所有数字都在 1～n 的范围内，所以数组中至少有一个数字是重复的。请找出数组中任意一个重复的数字，但不能修改输入的数组。例如，如果输入长度为 8 的数组{2, 3, 5, 4, 3, 2, 6, 7}，那么对应的输出是重复的数字 2 或者 3。

这一题看起来和上面的面试题类似。由于题目要求不能修改输入的数组，我们可以创建一个长度为 $n+1$ 的辅助数组，然后逐一把原数组的每个数字复制到辅助数组。如果原数组中被复制的数字是 m，则把它复制到辅助数组中下标为 m 的位置。这样很容易就能发现哪个数字是重复的。由于需要创建一个数组，该方案需要 $O(n)$的辅助空间。

接下来我们尝试避免使用 $O(n)$的辅助空间。为什么数组中会有重复的数字？假如没有重复的数字，那么在从 1～n 的范围里只有 n 个数字。由于数组里包含超过 n 个数字，所以一定包含了重复的数字。看起来在某范围里数字的个数对解决这个问题很重要。

我们把从 1～n 的数字从中间的数字 m 分为两部分，前面一半为 1～m，后面一半为 m+1～n。如果 1～m 的数字的数目超过 m，那么这一半的区间里一定包含重复的数字；否则，另一半 m+1～n 的区间里一定包含重复的数字。我们可以继续把包含重复数字的区间一分为二，直到找到一个重复的数字。这个过程和二分查找算法很类似，只是多了一步统计区间里数字的数目。

我们以长度为 8 的数组{2, 3, 5, 4, 3, 2, 6, 7}为例分析查找的过程。根据题目要求，这个长度为 8 的所有数字都在 1～7 的范围内。中间的数字 4 把 1～7 的范围分为两段，一段是 1～4，另一段是 5～7。接下来我们统计 1～4 这 4 个数字在数组中出现的次数，它们一共出现了 5 次，因此这 4 个数字中一定有重复的数字。

接下来我们再把 1～4 的范围一分为二，一段是 1、2 两个数字，另一段是 3、4 两个数字。数字 1 或者 2 在数组中一共出现了两次。我们再统计数字 3 或者 4 在数组中出现的次数，它们一共出现了三次。这意味着 3、4 两个数字中一定有一个重复了。我们再分别统计这两个数字在数组中出现的次数。接着我们发现数字 3 出现了两次，是一个重复的数字。

上述思路可以用如下代码实现：

```
int getDuplication(const int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        return -1;

    int start = 1;
    int end = length - 1;
    while(end >= start)
    {
        int middle = ((end - start) >> 1) + start;
```

```
        int count = countRange(numbers, length, start, middle);
        if(end == start)
        {
            if(count > 1)
                return start;
            else
                break;
        }

        if(count > (middle - start + 1))
            end = middle;
        else
            start = middle + 1;
    }
    return -1;
}

int countRange(const int* numbers, int length, int start, int end)
{
    if(numbers == nullptr)
        return 0;

    int count = 0;
    for(int i = 0; i < length; i++)
        if(numbers[i] >= start && numbers[i] <= end)
            ++count;
    return count;
}
```

上述代码按照二分查找的思路，如果输入长度为 n 的数组，那么函数 countRange 将被调用 $O(\log n)$次，每次需要 $O(n)$的时间，因此总的时间复杂度是 $O(n\log n)$，空间复杂度为 $O(1)$。和最前面提到的需要 $O(n)$的辅助空间的算法相比，这种算法相当于以时间换空间。

需要指出的是，这种算法不能保证找出所有重复的数字。例如，该算法不能找出数组{2, 3, 5, 4, 3, 2, 6, 7}中重复的数字 2。这是因为在 1～2 的范围里有 1 和 2 两个数字，这个范围的数字也出现 2 次，此时我们用该算法不能确定是每个数字各出现一次还是某个数字出现了两次。

从上述分析中我们可以看出，如果面试官提出不同的功能要求（找出任意一个重复的数字、找出所有重复的数字）或者性能要求（时间效率优先、空间效率优先），那么我们最终选取的算法也将不同。这也说明在面试中和面试官交流的重要性，我们一定要在动手写代码之前弄清楚面试官的需求。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/03_02_DuplicationInArrayNoEdit

测试用例：

- 长度为 n 的数组里包含一个或多个重复的数字。
- 数组中不包含重复的数字。
- 无效输入测试用例（输入空指针）。

本题考点：

- 考查应聘者对一维数组的理解及编程能力。一维数组在内存中占据连续的空间，因此我们可以根据下标定位对应的元素。
- 考查应聘者对二分查找算法的理解，并能快速、正确地实现二分查找算法的代码。
- 考查应聘者的沟通能力。应聘者只有具备良好的沟通能力，才能充分了解面试官的需求，从而有针对性地选择算法解决问题。

面试题 4：二维数组中的查找

> 题目：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

例如下面的二维数组就是每行、每列都递增排序。如果在这个数组中查找数字 7，则返回 true；如果查找数字 5，由于数组不含有该数字，则返回 false。

```
1   2   8   9
2   4   9   12
4   7   10  13
6   8   11  15
```

在分析这个问题的时候，很多应聘者都会把二维数组画成矩形，然后从数组中选取一个数字，分 3 种情况来分析查找的过程。当数组中选取的数字刚好和要查找的数字相等时，就结束查找过程。如果选取的数字小于要查找的数字，那么根据数组排序的规则，要查找的数字应该在当前选取位置的右边或者下边，如图 2.1（a）所示。同样，如果选取的数字大于要查找的数字，那么要查找的数字应该在当前选取位置的上边或者左边，如图 2.1（b）所示。

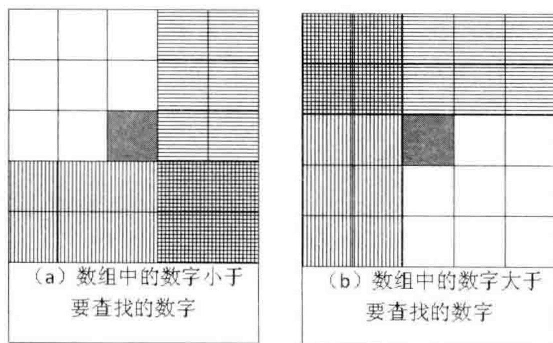

图 2.1 二维数组中的查找

注：在数组中间选择一个数（深色方格），根据它的大小判断要查找的数字可能出现的区域（阴影部分）。

在上面的分析中，由于要查找的数字相对于当前选取的位置有可能在两个区域中出现，而且这两个区域还有重叠，这问题看起来就复杂了，于是很多人就卡在这里束手无策了。

当我们需要解决一个复杂的问题时，一个很有效的办法就是从一个具体的问题入手，通过分析简单具体的例子，试图寻找普遍的规律。针对这个问题，我们不妨也从一个具体的例子入手。下面我们以在题目中给出的数组中查找数字 7 为例来一步步分析查找的过程。

前面我们之所以遇到难题，是因为我们在二维数组的中间选取一个数字来和要查找的数字进行比较，这就导致下一次要查找的是两个相互重叠的区域。如果我们从数组的一个角上选取数字来和要查找的数字进行比较，那么情况会不会变简单呢？

首先我们选取数组右上角的数字 9。由于 9 大于 7，并且 9 还是第 4 列的第一个（也是最小的）数字，因此 7 不可能出现在数字 9 所在的列。于

是我们把这一列从需要考虑的区域内剔除，之后只需要分析剩下的 3 列，如图 2.2（a）所示。在剩下的矩阵中，位于右上角的数字是 8。同样 8 大于 7，因此 8 所在的列我们也可以剔除。接下来我们只要分析剩下的两列即可，如图 2.2（b）所示。

在由剩余的两列组成的数组中，数字 2 位于数组的右上角。2 小于 7，那么要查找的 7 可能在 2 的右边，也可能在 2 的下边。在前面的步骤中，我们已发现 2 右边的列都已经被剔除了，也就是说 7 不可能出现在 2 的右边，因此 7 只有可能出现在 2 的下边。于是我们把数字 2 所在的行也剔除，只分析剩下的三行两列数字，如图 2.2（c）所示。在剩下的数字中，数字 4 位于右上角，和前面一样，我们把数字 4 所在的行也删除，最后剩下两行两列数字，如图 2.2（d）所示。

在剩下的两行两列 4 个数字中，位于右上角的刚好就是我们要查找的数字 7，于是查找过程就可以结束了。

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

（a）9 大于 7，下一次只需要在 9 的左边区域查找

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

（b）8 大于 7，下一次只需要在 8 的左边区域查找

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

（c）2 小于 7，下一次只需要在 2 的下边区域查找

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

（d）4 小于 7，下一次只需要在 4 的下边区域查找

图 2.2 在二维数组中查找 7 的步骤

注：矩阵中加阴影的区域是下一步查找的范围。

总结上述查找的过程，我们发现如下规律：首先选取数组中右上角的

数字。如果该数字等于要查找的数字，则查找过程结束；如果该数字大于要查找的数字，则剔除这个数字所在的列；如果该数字小于要查找的数字，则剔除这个数字所在的行。也就是说，如果要查找的数字不在数组的右上角，则每一次都在数组的查找范围中剔除一行或者一列，这样每一步都可以缩小查找的范围，直到找到要查找的数字，或者查找范围为空。

把整个查找过程分析清楚之后，我们再写代码就不是一件很难的事情了。下面是上述思路对应的参考代码：

```
bool Find(int* matrix, int rows, int columns, int number)
{
    bool found = false;

    if(matrix != nullptr && rows > 0 && columns > 0)
    {
        int row = 0;
        int column = columns - 1;
        while(row < rows && column >=0)
        {
            if(matrix[row * columns + column] == number)
            {
                found = true;
                break;
            }
            else if(matrix[row * columns + column] > number)
                -- column;
            else
                ++ row;
        }
    }

    return found;
}
```

在前面的分析中，我们每次都选取数组查找范围内的右上角数字。同样，我们也可以选取左下角的数字。感兴趣的读者不妨自己分析一下每次都选取左下角数字的查找过程。但我们不能选择左上角数字或者右下角数字。以左上角数字为例，最初数字 1 位于初始数组的左上角，由于 1 小于 7，那么 7 应该位于 1 的右边或者下边。此时我们既不能从查找范围内剔除 1 所在的行，也不能剔除 1 所在的列，这样我们就无法缩小查找的范围。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/04_FindInPartiallySortedMatrix

测试用例：

- 二维数组中包含查找的数字（查找的数字是数组中的最大值和最小值；查找的数字介于数组中的最大值和最小值之间）。
- 二维数组中没有查找的数字（查找的数字大于数组中的最大值；查找的数字小于数组中的最小值；查找的数字在数组的最大值和最小值之间但数组中没有这个数字）。
- 特殊输入测试（输入空指针）。

本题考点：

- 考查应聘者对二维数组的理解及编程能力。二维数组在内存中占据连续的空间。在内存中从上到下存储各行元素，在同一行中按照从左到右的顺序存储。因此我们可以根据行号和列号计算出相对于数组首地址的偏移量，从而找到对应的元素。
- 考查应聘者分析问题的能力。当应聘者发现问题比较复杂时，能不能通过具体的例子找出其中的规律，是能否解决这个问题的关键所在。这个题目只要从一个具体的二维数组的右上角开始分析，就能找到查找的规律，从而找到解决问题的突破口。

2.3.2 字符串

字符串是由若干字符组成的序列。由于字符串在编程时使用的频率非常高，为了优化，很多语言都对字符串做了特殊的规定。下面分别讨论 C/C++ 和 C#中字符串的特性。

C/C++中每个字符串都以字符'\0'作为结尾，这样我们就能很方便地找到字符串的最后尾部。但由于这个特点，每个字符串中都有一个额外字符的开销，稍不留神就会造成字符串的越界。比如下面的代码：

```
char str[10];
strcpy(str, "0123456789");
```

我们先声明一个长度为 10 的字符数组，然后把字符串"0123456789"复制到数组中。"0123456789"这个字符串看起来只有 10 个字符，但实际上它的末尾还有一个'\0'字符，因此它的实际长度为 11 字节。要正确地复制该字符串，至少需要一个长度为 11 字节的数组。

为了节省内存，C/C++把常量字符串放到单独的一个内存区域。当几个指针赋值给相同的常量字符串时，它们实际上会指向相同的内存地址。但用常量内存初始化数组，情况却有所不同。下面通过一个面试题来学习这一知识点。运行下面的代码，得到的结果是什么？

```
int _tmain(int argc, _TCHAR* argv[])
{
    char str1[] = "hello world";
    char str2[] = "hello world";

    char* str3 = "hello world";
    char* str4 = "hello world";

    if(str1 == str2)
        printf("str1 and str2 are same.\n");
    else
        printf("str1 and str2 are not same.\n");

    if(str3 == str4)
        printf("str3 and str4 are same.\n");
    else
        printf("str3 and str4 are not same.\n");

    return 0;
}
```

str1 和 str2 是两个字符串数组，我们会为它们分配两个长度为 12 字节的空间，并把"hello world"的内容分别复制到数组中去。这是两个初始地址不同的数组，因此 str1 和 str2 的值也不相同，所以输出的第一行是"str1 and str2 are not same"。

str3 和 str4 是两个指针，我们无须为它们分配内存以存储字符串的内容，而只需要把它们指向"hello world"在内存中的地址就可以了。由于"hello world"是常量字符串，它在内存中只有一个拷贝，因此 str3 和 str4 指向的是同一个地址。所以比较 str3 和 str4 的值得到的结果是相同的，输出的第二行是"str3 and str4 are same"。

在 C#中，封装字符串的类型 System.String 有一个非常特殊的性质：String 的内容是不能改变的。一旦试图改变 String 的内容，就会产生一个新的实例。请看下面的 C#代码：

```
String str = "hello";
str.ToUpper();
str.Insert(0, " WORLD");
```

虽然我们对 str 执行了 ToUpper 和 Insert 两个操作，但操作的结果都是生成一个新的 String 实例并在返回值中返回，str 本身的内容都不会发生改变，因此最终 str 的值仍然是"hello"。由此可见，如果试图改变 String 的内容，则改变之后的值只能通过返回值得到。用 String 进行连续多次修改，每一次修改都会产生一个临时对象，这样开销太大会影响效率。为此，C#定义了一个新的与字符串相关的类型 StringBuilder，它能容纳修改后的结果。因此，如果要连续多次修改字符串内容，用 StringBuilder 是更好的选择。

和修改 String 的内容类似，如果我们试图把一个常量字符串赋值给一个 String 实例，那么也不是把 String 的内容改成赋值的字符串，而是生成一个新的 String 实例。请看下面的代码：

```
class Program
{
    internal static void ValueOrReference(Type type)
    {
        String result = "The type " + type.Name;

        if (type.IsValueType)
            Console.WriteLine(result + " is a value type.");
        else
            Console.WriteLine(result + " is a reference type.");
    }

    internal static void ModifyString(String text)
    {
        text = "world";
    }

    static void Main(string[] args)
    {
        String text = "hello";

        ValueOrReference(text.GetType());
        ModifyString(text);

        Console.WriteLine(text);
    }
}
```

在上面的代码中，我们先判断 String 是值类型还是引用类型。类型 String 的定义是 public sealed class String {...}。既然是 class，那么 String 自然就是引用类型。接下来在方法 ModifyString 里，对 text 赋值一个新的字符串。

我们要记得 text 的内容是不能被修改的。此时会先生成一个新的内容是"world"的 String 实例，然后把 text 指向这个新的实例。由于参数 text 没有加 ref 或者 out，出了方法 ModifyString 之后，text 还是指向原来的字符串，因此输出仍然是"hello"。要想实现出了函数之后 text 变成"world"的效果，我们必须把参数 text 标记 ref 或者 out。

面试题 5：替换空格

题目：请实现一个函数，把字符串中的每个空格替换成"%20"。例如，输入"We are happy."，则输出"We%20are%20happy."。

在网络编程中，如果 URL 参数中含有特殊字符，如空格、'#'等，则可能导致服务器端无法获得正确的参数值。我们需要将这些特殊符号转换成服务器可以识别的字符。转换的规则是在'%'后面跟上 ASCII 码的两位十六进制的表示。比如空格的 ASCII 码是 32，即十六进制的 0x20，因此空格被替换成"%20"。再比如'#'的 ASCII 码为 35，即十六进制的 0x23，它在 URL 中被替换为"%23"。

看到这个题目，我们首先应该想到的是原来一个空格字符，替换之后变成'%'、'2'和'0'这 3 个字符，因此字符串会变长。如果是在原来的字符串上进行替换，就有可能覆盖修改在该字符串后面的内存。如果是创建新的字符串并在新的字符串上进行替换，那么我们可以自己分配足够多的内存。由于有两种不同的解决方案，我们应该向面试官问清楚，让他明确告诉我们他的需求。假设面试官让我们在原来的字符串上进行替换，并且保证输入的字符串后面有足够多的空余内存。

❖ 时间复杂度为 $O(n^2)$的解法，不足以拿到 Offer

现在我们考虑怎么执行替换操作。最直观的做法是从头到尾扫描字符串，每次碰到空格字符的时候进行替换。由于是把 1 个字符替换成 3 个字符，我们必须要把空格后面所有的字符都后移 2 字节，否则就有两个字符被覆盖了。

举个例子，我们从头到尾把"We are happy."中的每个空格替换成"%20"。为了形象起见，我们可以用一个表格来表示字符串，表格中的每个格子表示一个字符，如图 2.3（a）所示。

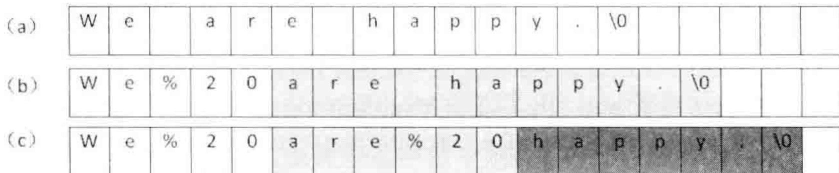

图 2.3 从前往后把字符串中的空格替换成'%20'的过程

注：（a）字符串"We are happy."。（b）把字符串中的第一个空格替换成'%20'。灰色背景表示需要移动的字符。（c）把字符串中的第二个空格替换成'%20'。浅灰色背景表示需要移动一次的字符，深灰色背景表示需要移动两次的字符。

我们替换第一个空格，这个字符串变成图 2.3（b）中的内容，表格中灰色背景的格子表示需要进行移动的区域。接着我们替换第二个空格，替换之后的内容如图 2.3（c）所示。同时，我们注意到用深灰色背景标注的"happy"部分被移动了两次。

假设字符串的长度是 n。对每个空格字符，需要移动后面 $O(n)$个字符，因此对于含有 $O(n)$个空格字符的字符串而言，总的时间效率是 $O(n^2)$。

当我们把这种思路阐述给面试官后，他不会就此满意，他将让我们寻找更快的方法。在前面的分析中，我们发现数组中很多字符都移动了很多次，能不能减少移动次数呢？答案是肯定的。我们换一种思路，把从前向后替换改成从后向前替换。

❖ 时间复杂度为 $O(n)$的解法，搞定 Offer 就靠它了

我们可以先遍历一次字符串，这样就能统计出字符串中空格的总数，并可以由此计算出替换之后的字符串的总长度。每替换一个空格，长度增加 2，因此替换以后字符串的长度等于原来的长度加上 2 乘以空格数目。我们还是以前面的字符串"We are happy."为例。"We are happy."这个字符串的长度是 14（包括结尾符号'\0'），里面有两个空格，因此替换之后字符串的长度是 18。

我们从字符串的后面开始复制和替换。首先准备两个指针：P_1 和 P_2。P_1 指向原始字符串的末尾，而 P_2 指向替换之后的字符串的末尾，如图 2.4（a）所示。接下来我们向前移动指针 P_1，逐个把它指向的字符复制到 P_2 指向的位置，直到碰到第一个空格为止。此时字符串如图 2.4（b）所示，灰

色背景的区域是进行了字符复制（移动）的区域。碰到第一个空格之后，把 P_1 向前移动 1 格，在 P_2 之前插入字符串"%20"。由于"%20"的长度为 3，同时也要把 P_2 向前移动 3 格，如图 2.4（c）所示。

我们接着向前复制，直到碰到第二个空格，如图 2.4（d）所示。和上一次一样，我们再把 P_1 向前移动 1 格，并把 P_2 向前移动 3 格插入"%20"，如图 2.4（e）所示。此时 P_1 和 P_2 指向同一位置，表明所有空格都已经替换完毕。

从上面的分析中我们可以看出，所有的字符都只复制（移动）一次，因此这个算法的时间效率是 $O(n)$，比第一个思路要快。

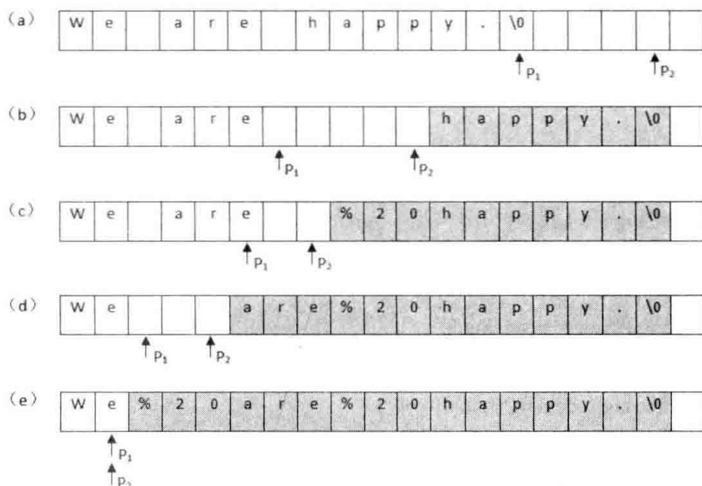

图 2.4　从后往前把字符串中的空格替换成"%20"的过程

注：图中带有阴影的区域表示被移动的字符。(a) 把第一个指针指向字符串的末尾，把第二个指针指向替换之后的字符串的末尾。(b) 依次复制字符串的内容，直至第一个指针碰到第一个空格。(c) 把第一个空格替换成"%20"，把第一个指针向前移动 1 格，把第二个指针向前移动 3 格。(d) 依次向前复制字符串中的字符，直至碰到空格。(e) 替换字符串中的倒数第二个空格，把第一个指针向前移动 1 格，把第二个指针向前移动 3 格。

在面试过程中，我们也可以和前面的分析一样画一两个示意图解释自己的思路，这样既能帮助我们厘清思路，也能使我们和面试官的交流变得更加高效。在面试官肯定我们的思路之后，就可以开始写代码了。下面是参考代码：

```
/*length 为字符数组 string 的总容量*/
void ReplaceBlank(char string[], int length)
{
    if(string == nullptr||length <= 0)
        return;

    /*originalLength 为字符串 string 的实际长度*/
    int originalLength = 0;
    int numberOfBlank = 0;
    int i = 0;
    while(string[i] != '\0')
    {
        ++ originalLength;

        if(string[i] == ' ')
            ++ numberOfBlank;

        ++ i;
    }

    /*newLength 为把空格替换成'%20'之后的长度*/
    int newLength = originalLength + numberOfBlank * 2;
    if(newLength > length)
        return;

    int indexOfOriginal = originalLength;
    int indexOfNew = newLength;
    while(indexOfOriginal >= 0 && indexOfNew > indexOfOriginal)
    {
        if(string[indexOfOriginal] == ' ')
        {
            string[indexOfNew --] = '0';
            string[indexOfNew --] = '2';
            string[indexOfNew --] = '%';
        }
        else
        {
            string[indexOfNew --] = string[indexOfOriginal];
        }

        -- indexOfOriginal;
    }
}
```

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/05_ReplaceSpaces

测试用例：

- 输入的字符串中包含空格（空格位于字符串的最前面；空格位于字符串的最后面；空格位于字符串的中间；字符串中有连续多个空格）。
- 输入的字符串中没有空格。
- 特殊输入测试（字符串是一个 nullptr 指针；字符串是一个空字符串；字符串只有一个空格字符；字符串中有连续多个空格）。

本题考点：

- 考查应聘者对字符串的编程能力。
- 考查应聘者分析时间效率的能力。我们要能清晰地分析出两种不同方法的时间效率各是多少。
- 考查应聘者对内存覆盖是否有高度的警惕。在分析得知字符串会变长之后，我们能够意识到潜在的问题，并主动和面试官沟通以寻找问题的解决方案。
- 考查应聘者的思维能力。在从前到后替换的思路被面试官否定之后，我们能迅速想到从后往前替换的方法，这是解决此题的关键。

相关题目：

有两个排序的数组 A1 和 A2，内存在 A1 的末尾有足够多的空余空间容纳 A2。请实现一个函数，把 A2 中的所有数字插入 A1 中，并且所有的数字是排序的。

和前面的例题一样，很多人首先想到的办法是在 A1 中从头到尾复制数字，但这样就会出现多次复制一个数字的情况。更好的办法是从尾到头比较 A1 和 A2 中的数字，并把较大的数字复制到 A1 中的合适位置。

举一反三：

在合并两个数组（包括字符串）时，如果从前往后复制每个数字（或字符）则需要重复移动数字（或字符）多次，那么我们可以考虑从后往前复制，这样就能减少移动的次数，从而提高效率。

2.3.3 链表

链表应该是面试时被提及最频繁的数据结构。链表的结构很简单，它由指针把若干个节点连接成链状结构。链表的创建、插入节点、删除节点等操作都只需要 20 行左右的代码就能实现，其代码量比较适合面试。而像哈希表、有向图等复杂数据结构，实现它们的一个操作需要的代码量都较大，很难在几十分钟的面试中完成。另外，由于链表是一种动态的数据结构，其需要对指针进行操作，因此应聘者需要有较好的编程功底才能写出完整的操作链表的代码。而且链表这种数据结构很灵活，面试官可以用链表来设计具有挑战性的面试题。基于上述几个原因，很多面试官都特别青睐与链表相关的题目。

我们说链表是一种动态数据结构，是因为在创建链表时，无须知道链表的长度。当插入一个节点时，我们只需要为新节点分配内存，然后调整指针的指向来确保新节点被链接到链表当中。内存分配不是在创建链表时一次性完成的，而是每添加一个节点分配一次内存。由于没有闲置的内存，链表的空间效率比数组高。如果单向链表的节点定义如下：

```
struct ListNode
{
    int         m_nValue;
    ListNode* m_pNext;
};
```

那么往该链表的末尾添加一个节点的 C++代码如下：

```
void AddToTail(ListNode** pHead, int value)
{
    ListNode* pNew = new ListNode();
    pNew->m_nValue = value;
    pNew->m_pNext = nullptr;

    if(*pHead == nullptr)
    {
        *pHead = pNew;
    }
    else
    {
        ListNode* pNode = *pHead;

        while(pNode->m_pNext != nullptr)
            pNode = pNode->m_pNext;

        pNode->m_pNext = pNew;
    }
}
```

在上面的代码中，我们要特别注意函数的第一个参数 pHead 是一个指向指针的指针。当我们往一个空链表中插入一个节点时，新插入的节点就是链表的头指针。由于此时会改动头指针，因此必须把 pHead 参数设为指向指针的指针，否则出了这个函数 pHead 仍然是一个空指针。

由于链表中的内存不是一次性分配的，因而我们无法保证链表的内存和数组一样是连续的。因此，如果想在链表中找到它的第 i 个节点，那么我们只能从头节点开始，沿着指向下一个节点的指针遍历链表，它的时间效率为 $O(n)$。而在数组中，我们可以根据下标在 $O(1)$时间内找到第 i 个元素。下面是在链表中找到第一个含有某值的节点并删除该节点的代码：

```
void RemoveNode(ListNode** pHead, int value)
{
    if(pHead == nullptr || *pHead == nullptr)
        return;

    ListNode* pToBeDeleted = nullptr;
    if((*pHead)->m_nValue == value)
    {
        pToBeDeleted = *pHead;
        *pHead = (*pHead)->m_pNext;
    }
    else
    {
        ListNode* pNode = *pHead;
        while(pNode->m_pNext != nullptr
            && pNode->m_pNext->m_nValue != value)
            pNode = pNode->m_pNext;

        if(pNode->m_pNext != nullptr && pNode->m_pNext->m_nValue == value)
        {
            pToBeDeleted = pNode->m_pNext;
            pNode->m_pNext = pNode->m_pNext->m_pNext;
        }
    }

    if(pToBeDeleted != nullptr)
    {
        delete pToBeDeleted;
        pToBeDeleted = nullptr;
    }
}
```

除了简单的单向链表经常被设计为面试题（详见面试题 6“从尾到头打印链表”、面试题 18“删除链表的节点”、面试题 22“链表中倒数第 k 个节点”、面试题 24“反转链表”、面试题 25“合并两个排序的链表”、面试题 52“两个链表的第一个公共节点”等），链表的其他形式同样也备受面试官的青睐。

- 把链表的末尾节点的指针指向头节点，从而形成一个环形链表（详见面试题 62“圆圈中最后剩下的数字”）。
- 链表中的节点中除了有指向下一个节点的指针，还有指向前一个节点的指针。这就是双向链表（详见面试题 36“二叉搜索树与双向链表”）。
- 链表中的节点中除了有指向下一个节点的指针，还有指向任意节点的指针。这就是复杂链表（详见面试题 35“复杂链表的复制”）。

面试题 6：从尾到头打印链表

> 题目：输入一个链表的头节点，从尾到头反过来打印出每个节点的值。链表节点定义如下：

```
struct ListNode
{
    int         m_nKey;
    ListNode* m_pNext;
};
```

看到这道题后，很多人的第一反应是从头到尾输出将会比较简单，于是我们很自然地想到把链表中链接节点的指针反转过来，改变链表的方向，然后就可以从头到尾输出了。但该方法会改变原来链表的结构。是否允许在打印链表的时候修改链表的结构？这取决于面试官的要求，因此在面试的时候我们要询问清楚面试官的要求。

面试小提示：

在面试中，如果我们打算修改输入的数据，则最好先问面试官是不是允许修改。

通常打印是一个只读操作，我们不希望打印时修改内容。假设面试官也要求这个题目不能改变链表的结构。

接下来我们想到解决这个问题肯定要遍历链表。遍历的顺序是从头到尾，可输出的顺序却是从尾到头。也就是说，第一个遍历到的节点最后一个输出，而最后一个遍历到的节点第一个输出。这就是典型的“后进先出”，我们可以用栈实现这种顺序。每经过一个节点的时候，把该节点放到一个栈中。当遍历完整个链表后，再从栈顶开始逐个输出节点的值，此时输出

的节点的顺序已经反转过来了。这种思路的实现代码如下：

```
void PrintListReversingly_Iteratively(ListNode* pHead)
{
    std::stack<ListNode*> nodes;

    ListNode* pNode = pHead;
    while(pNode != nullptr)
    {
        nodes.push(pNode);
        pNode = pNode->m_pNext;
    }

    while(!nodes.empty())
    {
        pNode = nodes.top();
        printf("%d\t", pNode->m_nValue);
        nodes.pop();
    }
}
```

既然想到了用栈来实现这个函数，而递归在本质上就是一个栈结构，于是很自然地又想到了用递归来实现。要实现反过来输出链表，我们每访问到一个节点的时候，先递归输出它后面的节点，再输出该节点自身，这样链表的输出结果就反过来了。

基于这样的思路，不难写出如下代码：

```
void PrintListReversingly_Recursively(ListNode* pHead)
{
    if(pHead != nullptr)
    {
        if (pHead->m_pNext != nullptr)
        {
            PrintListReversingly_Recursively(pHead->m_pNext);
        }

        printf("%d\t", pHead->m_nValue);
    }
}
```

上面的基于递归的代码看起来很简洁，但有一个问题：当链表非常长的时候，就会导致函数调用的层级很深，从而有可能导致函数调用栈溢出。显然用栈基于循环实现的代码的鲁棒性要好一些。更多关于循环和递归的讨论，详见本书的 2.4.1 节。

源代码：

本题完整的源代码：