数组是引用类型，在托管堆上分配，在变量或字段中，包含的是对数组的引用。CLR支持一维、多维、交错数组。

Ps：数组是一个引用类型，即使是返回一个长度为0的数组，返回的也是一个引用而不是null。但是数组

如何声明数组

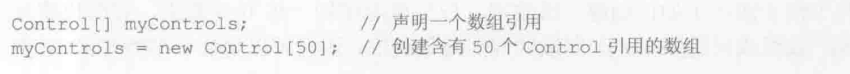
Ps：CLS要求所有数组必须为“0基数组”（最小索引为0），但是也可以创建“非0基数组”，并且实际上只有0基一维数组和非0基一维或多维数组，实际上内存中0基多维数组也是按照一维数组的方式存储的

1.一维数组

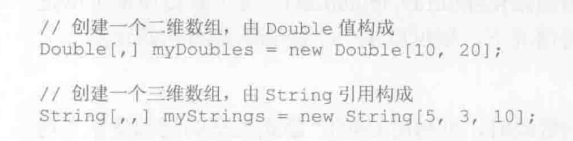
值类型数组：



也可以声明引用类型数组：



2.多维数组



3.交错数组（数组构成的数组）

文本

描述已自动生成

值类型数组和引用类型数组在托管堆中的不同

图示, 表格

描述已自动生成

Ps：无论是值类型还是引用类型数组都包含了一些额外开销，即上图中的“Overhead”（开销类型），这包含了数组的秩（维数），数组的下限（通常为0），每一维的长度、数组元素类型的信息。

对于值类型数组，例如上图int[] 数组，数组本身作为一个对象存储在托管堆中，而它的元素也存储在托管堆中。这与值类型通常存储在栈上的情况不同。这意味着，即使数组的元素是值类型，它们也不会存储在栈上，而是与数组本身一起存储在托管堆中。

对于引用类型数组，例如上图Control[] 数组，与值类型数组类似。数组本身作为一个对象存储在托管堆中，而它的元素也存储在托管堆中，数组中存储的是对堆其他上引用类型的引用。

Ps：尽量使用一维0基数组，这种数组称为“向量”，这样的数组性能是最佳的，因为可以使用一些特殊的IL指令（如newarr、Idelem、Idelema、Idlen和stelem）来处理。

为什么一维0基数组性能是最佳的？

Ps：0基一维数组和0基一维交错数组性能一样好，不过交错数组的元素意味着至少进行2次访问。

一维0基数组在内存中是连续存储（一维0基、多维、非0基数组在内存中都一定是连续存储的，而交错数组可能在内存中不一定是连续存储的），当访问数组中的元素时，计算机可以快速定位元素的位置，并且可以利用缓存来提高性能。多维数组通常比 “向量”有较低的性能，因为多维数组需要额外的计算来确定元素的位置，非0基数组性能低于向量也是同样的原因。

初始化数组元素的方法：

1.使用数组初始化器（大括号中的以逗号分隔的数据项称为“数据初始化器”）



隐式类型也可以用于数组：



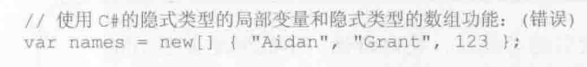
Ps：每个数据项都可以是一个任意复杂度的表达式，在多维数组中可以是一个嵌套的初始化器

比较隐式初始化一组报错与不报错的情况

不报错：



报错：



不报错的情况是因为null可隐式转换为任意引用类型。报错是因为123为Int32类型而其他的元素为是string类型，如果需要创建一个数组那么必须是Object类型数组，但是堆数组中每一个元素都装箱代价太大了，所以C#编译器并不会创建一个Object数组然后装箱，而是直接抛出异常。

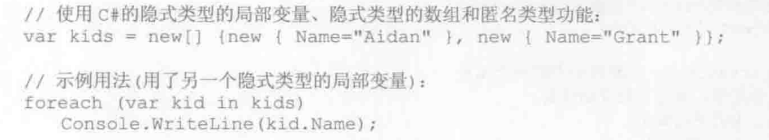
2.简化语法初始化（因为string为基元类型）

文本

描述已自动生成

Ps：这种语法不可以使用隐式类型局部变量声明。

3.同时使用匿名类型和简化语法初始化：



输出

文本

描述已自动生成

数组转型的规则：

1.数组维数相同

2.必须存在从元素源类型到目标类型的显式或隐式转换

3.值类型数组无法转型为其他任何类型的数组

下面是书上给出的数组转型的例子：

显式转型和隐式转型的方式参考变量间的转型

文本

描述已自动生成

文本

描述已自动生成

Array.Copy方法的作用：将元素从一个数组复制到另一个，Copy方法还能处理内存重叠区域

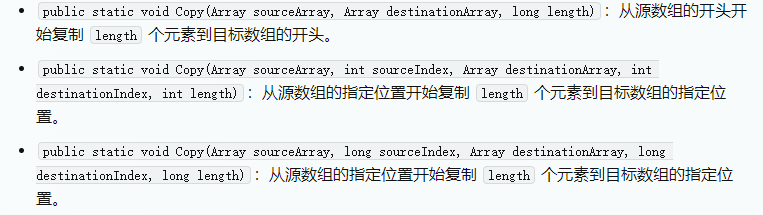
Ps：内存重叠区域指两个或多个内存块共享相同的物理内存地址。

Array.Copy如何处理内存重叠区域？

当源数组和目标数组重叠时，Array.Copy 函数会自动检测这种情况，并采取适当的措施来确保数据的完整性，它会根据源数组和目标数组的位置关系，选择从前往后或从后往前复制元素，以避免数据被覆盖。

下面是例子：

函数以及其重载函数的签名：



图片包含 图表

描述已自动生成

这个数组对自己使用copy方法，因此内存重叠。它成功地将前三个元素赋值到了后三个位置上

Copy函数的其他用法：

文本

描述已自动生成

但是Copy类型的转型会带来性能损失，因为它会检查数组的实际类型从而确定是否可以转型，若不能转型则抛出异常。

所有数组类型都隐式继承了System.Array类型

比如书上提供的例子



FileStrean[]隐式继承了System.Array类

那么所有的数组类型都继承了System.Array的属性和方法。

System.Array包含了哪些方法与属性？



表格

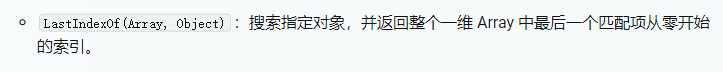
描述已自动生成

文本

描述已自动生成

文本

中度可信度描述已自动生成



文本

描述已自动生成

表格

描述已自动生成

所有数组类型除了继承了System.Array类型以外，都实现了IEnumerable、ICollection和IList接口

Ps：System.Array也实现了这三个接口

实现这三个泛型接口，这三个接口把所有数组都视为Object类型

仅在创建一维0基数组时，一维0基数组的对应的类型实现了IEnumerable<T>、ICollection<T>和IList<T>接口

Ps：特例，DateTime[]类只会实现IEnumerable< DateTime >、ICollection< DateTime >和IList< DateTime >接口，而不是实现它的泛型版本，这是因为DateTime为值类型，值类型在内存中的排布与引用类型不同。（所以为什么不实现泛型版本？）

如何创建一个非0基数组？

使用CreateInstance方法可以动态创建数组，将参数信息保存到overhead块，然后返回数组的引用（如果是一个多维数组，则返回一个一个一维数组，这个一维数组中每个元素是每一维的引用）

CreateInstance方法的所有重载方法的签名：

图形用户界面, 文本, 应用程序

描述已自动生成

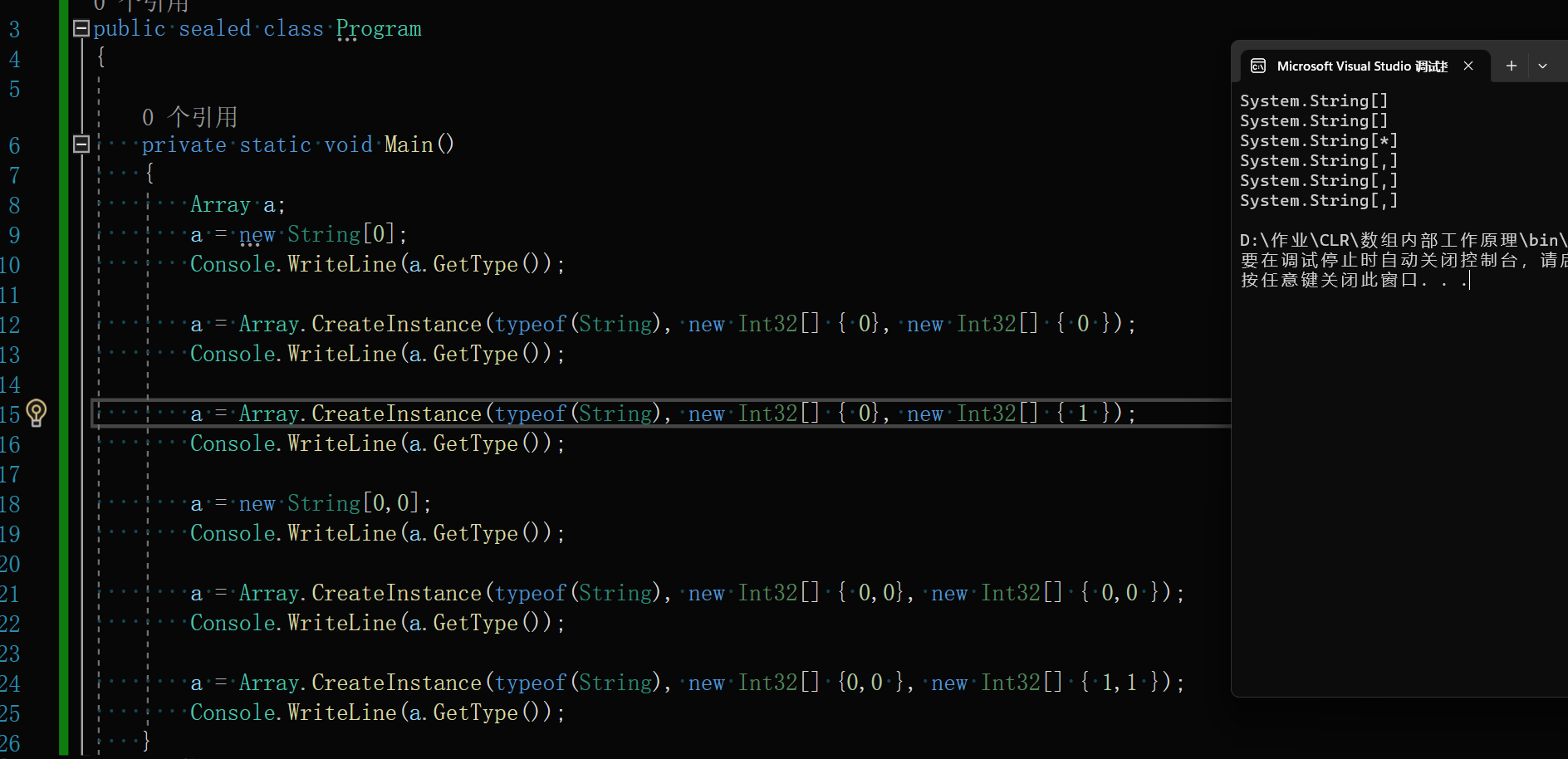
使用CreateInstance动态创建数组：



（ 如何将非0基数组的上下限硬编码到代码中？）

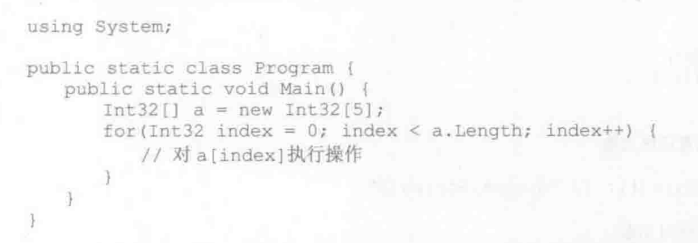
数组内部的工作原理

下面是书上的例子（P341）：



一维非0基数组显示的类型很奇怪，C#中不允许声明这种类型，因此这样的数组不能使用C#语法访问。并且C#将所有多维数组都视为非0基数组，C#编译器会为一维0基数组生成特殊优化的IL代码。

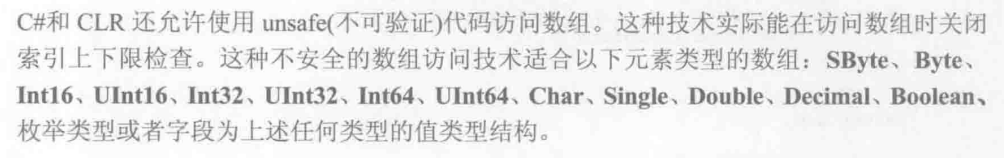
比如书上的例子：



此时针对0基一维数组的优化是，不进行越界检查。并且只执行了一次查询数组长度的操作，查询完之后将它存入了一个临时变量，之后的循环条件检查的代码中都使用这个临时变量。

Ps：多维数组和非0基数组性能低于向量的原因是查找时JIT编译器会在每次循环时进行越界检查。

使用unsafe代码访问数组也可以关闭越界检查



书上二维数组的三种访问方式（安全、不安全、交错）

文本

描述已自动生成

文本, 信件

描述已自动生成

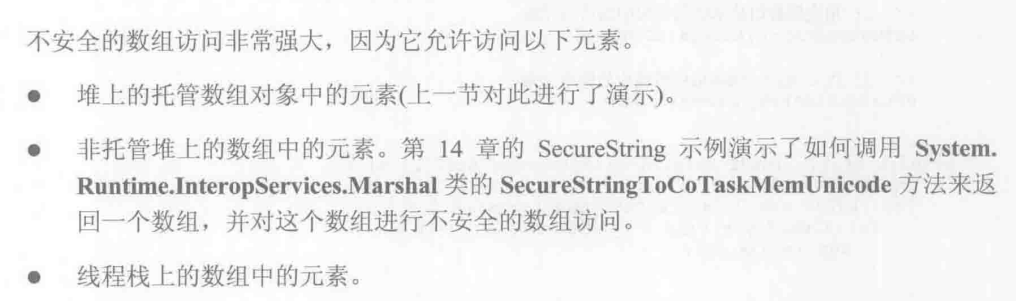
使用unsafe代码访问数组的缺陷：

1.相较于其他技术，处理数组元素的代码更加复杂，不利于读和写，因为要使用C#fix语句要执行内存地址计算（因为是非托管代码）。

2.计算过程中出错可能访问到不属于数组的内存。

3.CLR禁止在降低了安全级别的安全环境中运行不安全代码（？？？？怎么理解降低了安全级别的安全环境）

不安全数组的访问



线程栈上的数组中的元素是？？？？？？

如何在线程栈上分配数组？

使用stackalloc表达式，这个函数只能创建0基一维数组、由值类型元素构成的数组。只能是局部数组，在方法返回时销毁。并且不可以显式释放stackalloc分配的内存。

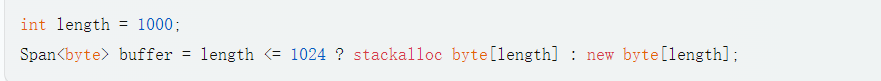
例子：

图形用户界面, 文本, 应用程序

描述已自动生成

Ps：将栈分配的内存块分配给 Span<T> 或 ReadOnlySpan<T> 变量时，不必使用 unsafe 上下文。当使用这些类型时可以在条件或赋值表达式中使用 stackalloc 表达式。

例子：



为什么尽可能使用 Span<T> 或 ReadOnlySpan<T> 类型来处理栈分配的内存？

1. Span<T> 和 ReadOnlySpan<T> 是轻量级的内存缓冲区，它们包装对托管或非托管内存的引用。由于这些类型只能存储在栈上，因此它们不适用于诸如异步方法调用之类的场景。

2. Span<T> 比 Memory<T> 更通用，可以表示更多种类的连续内存缓冲区。Span<T> 还提供了更多的优化，因此在可能的情况下，应该在同步 API 中使用 Span<T> 而不是 Memory<T> 作为参数。

3. Span<T> 和 ReadOnlySpan<T> 是结构体，因此每个 span 实际上只是当前线程栈上添加的几个字节。栈分配非常快速，且不会影响 GC。

在结构struct中嵌入数组需要满足的条件：

