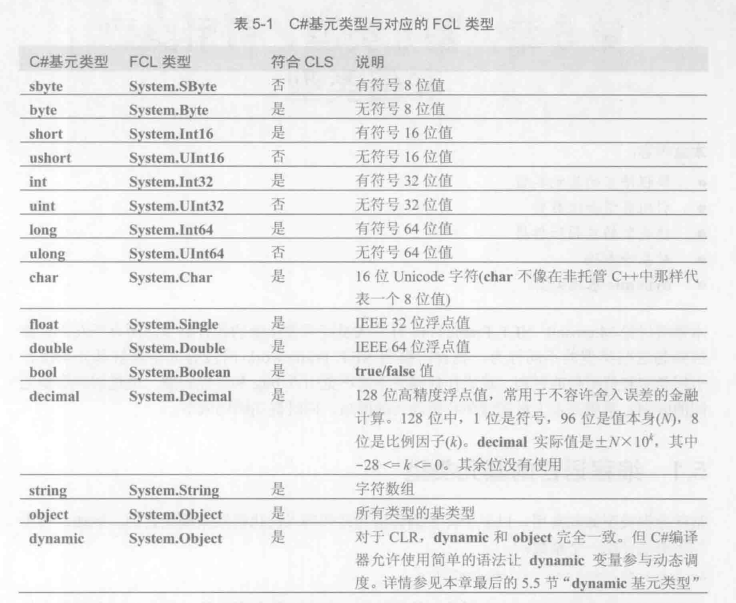
基元类型的定义：编译器直接支持的数据类型。

以下是C#基元类型和对应的FCL类型



使用基元类型的弊端：

报纸上的文字

描述已自动生成

对于基元类型，C#编译器在编译代码时应用了特殊的规则，因此基元类型间是可以隐式转换的，但仅限于低数量级转高数量级(比如Int32转Int64)，而高精度转低精度是被认为“不安全”的，因为可能丢失精度(比如Int64的第64位的数字会在转为Int32之后丢失)，因此如果需要高数量级转低数量级需要显示转换。

PS:如果进行显示转换让高精度转低精度，丢失的部分会被向下取整而不是四舍五入或者向上取整。

溢出检查：

CLR提供add指令和add.ovf指令，前者在两值相加时不执行溢出检查，后者提供溢出检查在溢出时抛出System.OverflowException异常。（还提供sub/sub.ovf、mul/mul.ovf、conv/conv.ovf适配其他运算）

C#编译器提供/checked+编译器开关决定是否进行溢出检查，若 打开后发现溢出抛出OverflowException异常。还可以使用checked和unchecked语句块来决定语句块内语句是否进行溢出检查。

Ps：

1.checked语句块中的方法不受影响，方法是否进行溢出检查取决于方法是否定义在checked语句块中

手机屏幕截图

描述已自动生成

2. 在计算哈希值和校验和时，算术运算的溢出是可以接受并且期望的，原因如下：

增加随机性：哈希值和校验和通常使用固定位数来表示结果。当算术运算发生溢出时，结果会被截断为指定的位数。这种截断可以增加结果的随机性和分布性，从而增强哈希函数的强度和校验和的可靠性。

减小碰撞的可能性：碰撞是指两个不同的输入产生相同的哈希值或校验和。通过允许溢出，可以增加输入的变化范围，从而减少碰撞的可能性。溢出可以增加结果的分散性，使得不同的输入更有可能产生不同的哈希值或校验和。

性能优势：在某些情况下，使用溢出可以避免执行昂贵的溢出检查或处理溢出的额外逻辑。这可以提高计算速度和效率，特别是对于大规模的数据处理和高性能的计算环境。

需要注意的是，溢出的处理方式可能因算法和应用而异。在某些情况下，溢出可能被简单地截断或忽略。在其他情况下，可能需要采取特殊的处理方法来确保结果的正确性和一致性。

引用类型和值类型的区别

存储方式：值类型的实例直接存储在栈上，而引用类型的实例存储在堆上，并通过引用（指针）来访问。

复制行为：当值类型被赋值给另一个变量或作为参数传递时，会进行值的复制。而引用类型的复制只会复制引用，两个引用指向同一个对象。

空值：值类型不能为null，它们总是有一个默认值。而引用类型可以为null，表示不引用任何对象。

默认初始化：值类型在声明时会自动进行默认初始化，即赋予一个默认值，默认初始化为0。引用类型在声明时默认为null，且引用类型实例会有同步块索引和类型对象指针，值类型并没有。

内存管理：值类型的内存管理由编译器自动处理，它们的生命周期与其所在的作用域相关。引用类型的内存管理由垃圾回收器（Garbage Collector）负责，它们的生命周期不仅与作用域相关，还受到垃圾回收器的影响。

比较行为：值类型的比较是按值进行的，即比较它们的实际值是否相等。引用类型的比较默认是比较引用是否指向同一个对象，除非重写了Equals方法。

继承和多态：引用类型支持继承和多态，可以通过基类引用来引用派生类的实例。值类型不支持继承和多态。

图片包含 图示

描述已自动生成

Ps：

1.C#中的一些类型（如字符串string）看起来像引用类型，但实际上是特殊的引用类型，称为不可变引用类型。它们的行为类似于值类型，但在内部仍然使用引用来管理对象的生命周期。

2.值类型从System.ValueType派生，引用类型从System.Object派生，但System.ValueType也是从System.Objcet派生。另外枚举作为值类型，但是从System.Enum抽象类型派生。

值类型不作为对象在托管堆中分配，不被垃圾回收，不通过指针进行引用。值类型的变量是存储在栈上的，它们的生命周期与其所在的作用域相对应。当变量超出作用域时，它们会自动被清理。

同一性和相等性：同一性判断地址是否相同，相等性判断数据是否相同。

值类型实例装箱时发生的事情：

1.在托管堆中分配内存。分配的内存量是值类型各字段所需的内存量，外加同步块索引和类型对象指针所需的内存量。

2.将值类型字段复制到新分配的堆内存。

3.返回对象的地址到栈上空间。

装箱过程需要注意的一些事情：

1.值类型调用ToString方法不会发生装箱，因为直接返回了一个字符串，可以用ToString避免装箱。

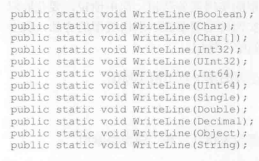
2.Console.WriteLine()方法定义了获取单个值类型的重载，所以不会发生装箱（P116）

3.避免循环装箱

4.调用非虚的、继承的方法时(比如GetType)，无论如何都要堆值类型进行装箱。

5.值类型的装箱使用的是栈上值类型实例的副本，并不会影响原本栈上的值类型实例

Console.WriteLine的不会造成装箱的重载



与已装箱的值类型实例拆箱相关的事情：

1.如果包含对“已装箱的值类型实例的引用”的变量为null，抛出NullReferenceException异常

2.如果引用的对象不是所需值类型的已装箱实例，抛出InvalidCastException异常。

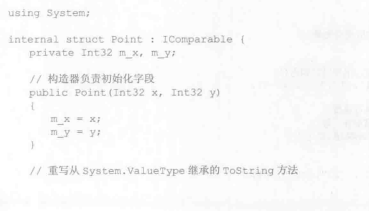
3.拆箱和隐式转换不可以同时进行，必须把这个过程拆开变成显示转换。

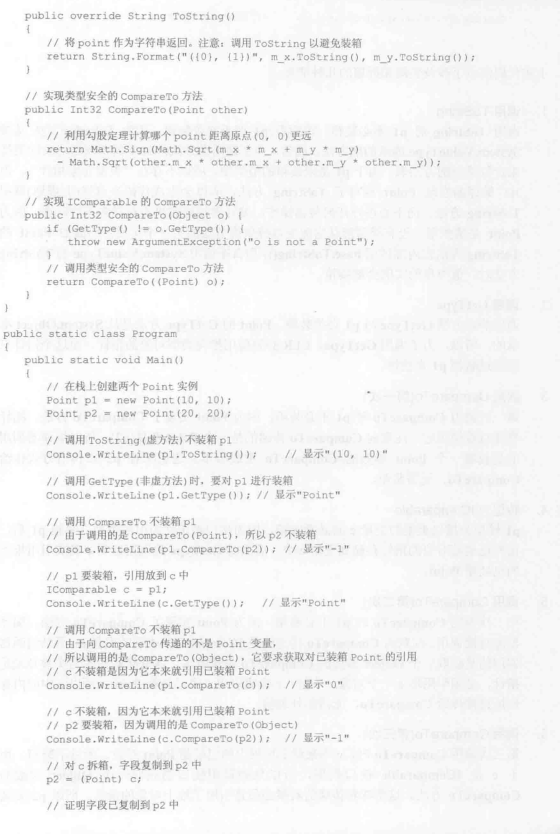
4.已装箱的值类型实例，拆箱时会伴随一次复制(把这些字段从堆复制到栈上的变量中去)，因此修改堆中已装箱的值类型必须先拆箱在栈上修改再装箱(见《CLR via C#》P113)

5.即使String作为特殊的引用类型会存储在栈上，但是值类型转化为String也需要装箱(见《CLR via C#》P114~115装箱3次代码)

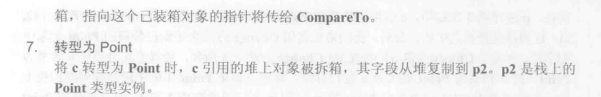
6. 拆箱只是把引用类型中的字段复制到栈上的一个临时变量中

涵盖装箱拆箱很多情况的例子：

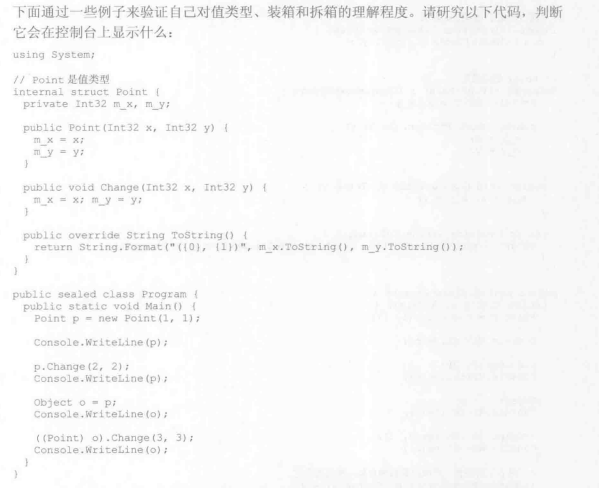








对装箱拆箱理解的测试：



1.先输出（1，1），此时调用了Console.WriteLine进行了第一次装箱，但装箱并不影响栈上原本的值类型实例，p仍是值类型

2.输出（2，2），调用Change方法改变栈上的值，此时p没有被装箱，p仍是值类型

3.输出（2，2），进行第二次装箱，此次是p装箱，引用类型在堆上的引用被返回到变量o那

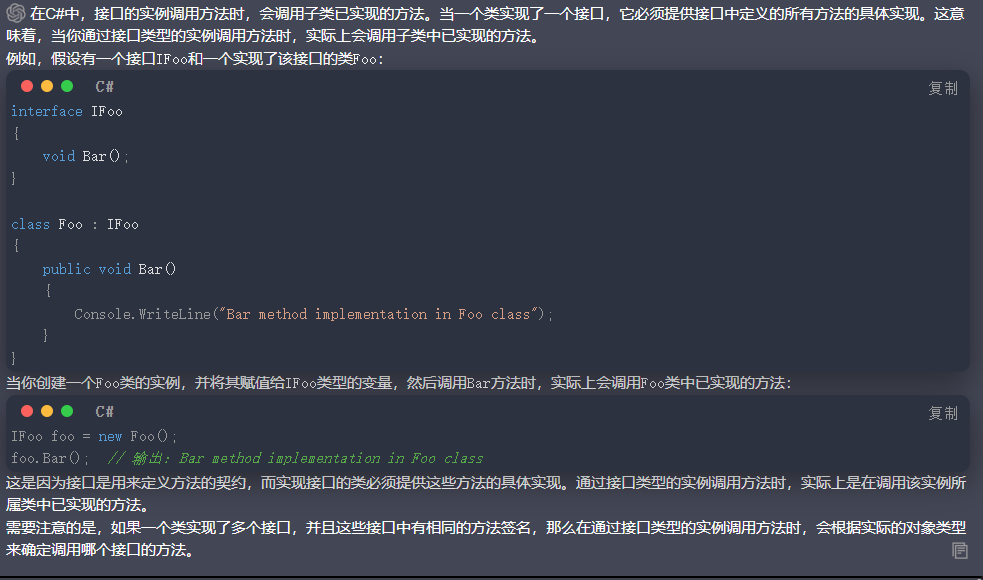
4输出（2，2），先拆箱，但拆箱只是把引用类型中的字段复制到栈上的一个临时变量中，修改为（3，3）再调用Console.WriteLine进行第三次装箱，但这次装箱修改的是临时变量的值，重新装箱后没有返回到栈上的变量o中，重新装箱后被垃圾回收

C#不允许直接修改已装箱的值类型中的字段，但仍然可以使用接口完成这一操作：

表格

中度可信度描述已自动生成

能这么做的原因：



C#的Object类默认实现的Equals方法实现的是同一性，即比较两字段引用的地址是否相同，若相同则一定相等返回true否则返回false。但这样实现的Equals非常不合理，因为存在两字段引用不同但值相同的情况。

理应如何实现Equals（具有相等性的Equals）：

1.若字段的引用为null则直接返回false

2.若两字段的引用相同则直接返回true

3.若两字段的类型不同则直接返回false

4.若两字段的类型不同但引用的值是相等的返回true

5.调用基类的Equals方法比较它定义的任何字段，若基类Equals方法返回的false则返回false，否则返回true （P124 这句话什么意思啊？？？？）

因此重写的Equals方法必须满足以下4个特征：

1.自反性。x.Equals(x)必然返回true。

2.对称性。x.Equals(y)和y.Equals(x)必然返回相同的值。

3.传递性。x.Equals(y)返回true，y.Equals(z)返回true，则x.Equals(z)一定返回true。 4.一致性。比较的两个值不变则结果必然不变。

这样实现相等性Equals方法还要注意两点细节：

1.继承System.IEquatable<T>接口，实现这个接口的Equals方法。这个接口保证了类型安全。

2.重载==和!=操作符

3.出于排序的目的进行的比较还应该继承System.ICompare<T>接口实现CompareTo方法

实现Equals()方法并继承System.IEquatable<T>接口可以提供更好的类型安全性和性能。

System.IEquatable<T>接口定义了一个泛型的Equals()方法，用于比较当前对象与另一个对象的值是否相等。通过实现该接口，您可以在类型安全的环境中进行相等性比较，并避免了需要进行类型转换的麻烦。

以下是一些原因说明为什么实现Equals()方法时继承System.IEquatable<T>接口是有益的：

类型安全性：通过继承System.IEquatable<T>接口，您可以在编译时捕获类型不匹配的错误。如果您尝试将一个对象与不兼容的类型进行比较，编译器将会报错。这有助于避免在运行时出现类型转换错误。

性能优化：实现IEquatable<T>接口可以提供更高的性能。在进行相等性比较时，如果两个对象的类型都实现了IEquatable<T>接口，那么将会调用泛型的Equals()方法，这比默认的Equals()方法更高效。这是因为泛型的Equals()方法避免了装箱和拆箱的开销。

一致性：通过继承IEquatable<T>接口，您可以确保Equals()方法的行为与==运算符的行为保持一致。这是因为在某些情况下，==运算符会调用Equals()方法来进行相等性比较。通过实现IEquatable<T>接口，您可以确保这两种方式的比较逻辑是一致的。

实现同一性比较的方法——ReferenceEquals (该方法的签名为static bool ReferenceEquals(object a,object b))

如果定义的类型重写了Equals方法，那么一定要重写GetHashCode方法！因为在一些类型，例如Dictionary和Hashtable类型还有另外一些System.Collections命名空间下的集合类型中，对象必须拥有相同的哈希码才被视为相同。因此在重写Equals方法时必须重写GetHashCode方法以保证相等性算法和哈希码算法一致。

ps疑问：P127最下方，意味着所有表达式都解析成类型的实例。

Dynamic基元类型

代码使用dynamic表达式/变量调用成员时，编译器生成特殊的IL代码描述所需的操作。这种特殊代码称为payload(有效载荷)。在运行时，payload代码根据表达式/变量引用的对象的实际类型来决定具体执行的操作。

Payload在运行时检查字段的真实类型，dynamic被编译器自动转化为Object。

但dynamic不等于Object：1.编译器不允许Object隐式转化为其他类型，但允许dynamic这么做。 2.可以定义对Object的扩展方法，而不可以定义对dynam进行扩展的扩展方法。 3.不能将匿名方法或者lambda表达式作为实参传递给dynam方法调用。

Dynamic和var的异同：（疑问：可以理解为C++的静态联编和动态联编吗？）

相同点：

类型推断：dynamic和var都是用于进行类型推断，即由编译器根据变量的初始化值来确定变量的类型。

运行时解析：dynamic和var都允许在运行时进行类型解析和动态绑定。它们可以在编译时不确定变量的具体类型，而是在运行时根据实际情况进行类型检查和调用。

不同点：

静态类型 vs 动态类型：var声明的变量是静态类型，即在编译时确定变量的类型，并且变量的类型在初始化时被确定。而dynamic声明的变量是动态类型，即在编译时不确定变量的类型，而是在运行时根据实际情况确定变量的类型。

编译时检查 vs 运行时检查：var声明的变量在编译时会进行类型检查，如果变量的初始化值与推断的类型不匹配，会导致编译错误。而dynamic声明的变量在编译时不进行类型检查，类型检查和绑定是在运行时进行的，因此可以接受任何类型的值。

静态绑定 vs 动态绑定：var声明的变量在编译时进行静态绑定，即变量的类型在编译时确定，绑定的方法和成员也在编译时确定。而dynamic声明的变量在运行时进行动态绑定，即变量的类型和绑定的方法和成员在运行时根据实际情况确定。（即dynamic是运行时绑定）

Ps：需要注意的是，使用dynamic会带来一些运行时的性能开销，因为类型检查和绑定是在运行时进行的。而var则没有这个开销，因为类型检查和绑定是在编译时确定的。因此只是一两个位置需要动态行为，调用反射方法更好。

Dynamic只能访问对象的实例成员，因为dynamic必须引用对象。但有时动态调用运行时才能确定一个类型的静态成员。