

C、C#、C++

一. 数据类型

数据类型	32 位字节数	64 位字节数
bool	1	1
Char	1	1
Unsigned char	1	1
Short	2	2
Unsigned short	2	2
Int	4	4
Unsigned int	4	4
Long	4	8
Unsigned long	4	8
Long long	8	8
Float	4	4
Double	8	8
Long double	8	8
*	4	8

1. 数据类型在 C# 上的精度

C# 类型/关键字	范围	大小	
sbyte	-128 到 127	8 位带符号整数	
byte	0 到 255	无符号的 8 位整数	
short	-32,768 到 32,767	有符号 16 位整数	
ushort	0 到 65,535	无符号 16 位整数	
int	-2,147,483,648 到 2,147,483,647	带符号的 32 位整数	
uint	0 到 4,294,967,295	无符号的 32 位整数	
long	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	64 位带符号整数	
ulong	0 到 18,446,744,073,709,551,615	无符号 64 位整数	
nint	取决于（在运行时计算的）平台	带符号的 32 位或 64 位整数	
nuint	取决于（在运行时计算的）平台	无符号的 32 位或 64 位整数	
float	$\pm 1.5 \times 10^{-45}$ 至 $\pm 3.4 \times 10^{38}$	大约 6-9 位数字	4 个字节
double	$\pm 5.0 \times 10^{-324}$ 到 $\pm 1.7 \times 10^{308}$	大约 15-17 位数字	8 个字节
decimal	$\pm 1.0 \times 10^{-28}$ 至 $\pm 7.9228 \times 10^{28}$	28-29 位	16 个字节

二. 运算符优先级和结合性

优先级	运算符	结合方向	名称或含义
1	()	从左到右	圆括号
	[]		数组下标
	.		结构体成员
	->		用指针访问类或结构

			体成员
	::		类成员
2	+, -	从右到左	正号和负号
	++, --		自增和自减
	!		逻辑非
	~		按位取反
	(t)		类型转换
	*		由地址求内容
	&		求变量的地址
	Sizeof		求某类型变量的长度
3	*, /, %	从左到右	乘、除、求余
4	+, -		加和减
5	<<, >>		左移和右移
6	<, >, <=, >=		关系运算（比较）
7	==, !=		等于和不等
8	&		按位与
9	^		按位异或
10			按位或
11	&&		逻辑与
12			逻辑或
13	?::		条件运算
14	=		赋值运算
	+=, -=, *=		复合赋值运算
	/=, %=, >>=		
	<<=, &=, ^=		
	=		
15	,		顺序求值运算

三. ASCII

ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符
0	NUL	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	,	71	G	103	g
8	BS	40	(72	H	104	h
9	HT	41)	73	I	105	i

10	LF	42	*	74	J	106	g
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	/	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	`
31	US	63	?	95	_	127	DEL

四. 关键字

1. When

1)作为 switch 语句中的 case guard

case 模式可能表达功能不够，无法指定用于执行 switch 部分的条件。在这种情况下，可以使用 case guard。这是一个附加条件，必须与匹配模式同时满足。case guard 必须是布尔表达式。可以在模式后面的 when 关键字之后指定一个 case guard，如以下示例所示：

PS:

```
void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to {a}.");
            break;

        case (> 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement is
{b}.");
            break;
```

```

default:
    Console.WriteLine("One or both measurements are not valid.");
    break;
}
}

```

五. 浮点数的精度问题

很多朋友在平时的项目中会用 double 类型来代替 float 以提高数据精度，错误的认为 double 可以解决精度问题。其实浮点数计算在我们大多数项目中并没有使用到特别科学的计算部分，所以 float 基本都够用。其实 double 也同样有精度问题，无论怎么样都是无法避免精度导致的在逻辑中的不一致的问题。

我们不妨比较下 float 与 double 来看看它们有什么不同。

float 和 double 所占用的比特位数不同会导致精度不同，float 是 32 个位占用 4 字节而 double 是 64 个位占用了 8 个字节，因此它们在计算时也会引起计算效率的不同。

实际工作中我们很多时候想试图通过使用 double 替换 float 来解决精度问题，最后基本都会以失败告终。因此我们要认清精度这个问题的根源，才能真正解决问题，我们先来看下浮点数在内存中到底是如何存储的。

计算机只认识 0 和 1（以二进制存储），无论什么形式的数字都是一样的，无论是整数还是小数在计算机中都以二进制的方式储存在内存中的，那么浮点数是以怎样的方式来存储的呢？根据 IEEE 754 标准，任意一个二进制浮点数 F 均可表示为： $F = (-1)^s * (1.M) * (2^e)$ 。

从公式中可以看出，它被分为了 3 个部分，符号部分即 s 部分、尾数部分即 M 部分 和 阶码部分即 e 部分，s 为 sign 符号位 0 或 1，M 为尾数指具体的小数用二进制表示，它完整的表示为 1.xxx。其中 1 后面的尾数部分因为称它为尾数，e 是比例因子的指数是浮点数的指数。

(float1.png) (float2.png)

上面所示的是 32 位（左）和 64 位（右）的浮点数存储结构，即 float 和 double 的存储结构。不论是 32 位浮点数还是 64 位浮点数，它们都由 S、E、M 三部分组成并以相同的公式来计算得到最终值。

其中 E 采用隐含方式，即采用移码方法来表示正负指数。移码方法对两个指数大小的比较和对阶操作比较方便，因为阶码的值大时其指向的数值也是大的，这样更容易计算和辨认。移码（又叫增码）是符号位取反的补码，例如 float 的 8 位阶码，应将指数 e 加上一个固定的偏移值 127（01111111），即 e 加上 127 才是存储在二进制中的数据。

尾数 M 则更为简单，它只表示 1 后面的小数部分，而且是二进制直译的那种，然后再根据阶码来平移小数点，最后根据小数点的左右部分得出整数部分和小数部分的数据。

以 9.625 为例，转换为二进制为 1001.101 可以表达为 $1.001101 \times (2^3)$ ，因此尾数部分 M 为 0011010000000000000000 去掉了小数点左侧的 1，并用 0 在右侧补齐。

其中表示 9.625 时的 1001.101 的小数部分为什么是“101”，因为整数部分采用“除 2 取余法”来得到从 10 进制转换到 2 进制的数字，而小数部分则采用“乘 2 取整法”得到二进制。因此这里的小数部分 0.625 乘 2 为 1.25 取整得到 1，继续 0.25 乘以 2 为 0.5 取整得到 0，再继续 0.5 乘以 2 为 1 取整为 1，后面都是 0 不再计算，因此得到 0.101 这个小数点后面的二进制。

再以 198903.19 这个数字为例，先形象直接的转成二进制的数值为，整数部分采用“除 2 取余法”，小数部分采用“乘 2 取整法”，并把整数和小数部分用点号隔开得到 110000100011110111.0011000010100011（截取 16 位小数）。

以 1 为首平移小数点后得到 $1.100001000111101110011000010100011 * (2^{17})$ （平移 17 位）即 $198903.19(10) = -1^0 * 1.100001000111101110011000010100011 * (2^{17})$ 。

从结果可以看出，小数部分 0.19 转为二进制后，小数位数超过 16 位（已经手算到小数点后 32 位都还没算完，其实这个位数是无穷尽的），因此这里导致浮点数有诸多精度的问题，它很多时候无法准确的表示数字，甚至非常不准确。

浮点数的精度问题可不只是小数点的精度问题，随着数值越来越大，即使是整数也会有相同的问题，因为浮点数本身是一个 $1.M * (2^e)$ 公式形式得到的数字，当数字放大时，M 的尾数的存储位数没有变化，能表达的位数有限，自然越来越难以准确表达，特别是数字的末尾部分越来越难以准确表达。

1. 数值比较不相等

我们在写程序时经常会遇到特定阈值触发某逻辑的情景，比如某个变量，需要从 0 开始加，每次加某个小于 0.01 的数，加到刚好 0.23 时做某事，到 0.34 时做另外一件事，到 0.56 时再做另一件。

这种精确定位的问题，就会遇到麻烦。因为浮点数在加减乘数时无法完全精确定位到某个值，就会出现，要么比 0.23 小，要么比 0.23 大，永远不会刚刚与 0.23 相等的时候，这时我们不得不放弃 ‘==’ 这个等于号而选择 ‘>’ 大于号或者 ‘<’ 小于号来解决这种问题的出现。

如果一定要用等于来做比较，则需要有一个微小的浮动区间，即 $ABS(X-Y) < 0.00001$ 时认为 X 和 Y 是相等的。

2. 数值计算不确定

比如 $x = 1f$, $y = 2f$, $z = \frac{1f}{5555f} * 1110f$ ，如果 $\frac{x}{y} \leq 0.5f$ 时做某事，那么理论上说 $\frac{x}{z}$ 也能通过这个 if，因为在我们看来 z 就等于 2，和 y 是一样的数值。但实际上未必是这样的，浮点数在计算时由于位数的限制无法得到精确的数值而是一个被截断的数值，因此 z 的计算结果有可能是 0.4999999999991，当 $\frac{x}{z}$ 时，结果有可能得到大于 0.5。

这就让我们很头疼，在实际编码中，我们经常会遇到这样的情况，在外层的 if 判断成立，理论上同样的结果只是公式不同，它们在内层的 if 判断却可能不成立，使得程序就出现异常行为，因为看起来应该是得到同样的数值，但结果却不一样。

3. 不同设备计算结果不同

不同平台上的浮点数计算也有所偏差，由于不同设备上 CPU 的计算方式不同，导致相同的公式在不同的设备上计算出来的结果有略微的偏差。

面对这些精度上的问题我们该怎么办，下面就来看看我们有哪些解决方案。

2) 只计算一次

我们可以简单点，只计算一次，认定这个值为准确值，只用这个变量结果做判断，也省去了多次计算浪费的 CPU。

由于多次计算相同的结果可能造成精度问题，不如只计算一次，只用一次计算的结果把它当做唯一确定性结果，而不使用多次计算得到的结果。排除了多次结果不同导致的问题。

由于多次看似相等的计算其实得到的结果有可能不同，使得问题变得更复杂，比如上面所说的， $\frac{1f}{2f}$ 的结果，用 $\frac{1f}{\frac{1f}{5555f} * 1110f}$ 来表示得到的结果不一样，进而导致问题变得不可控。不如只使用一次计算，不再进行多次计算，认定这次的结果的数值为准确数值，只用这个浮点数值当做判断的标准。

当然这种方法使用范围比较小，可能不适用各位的实际需求，我们可以继续看看其他的解决方案。

3)用整型代替浮点型

我们可以改用 `int` 或 `long` 型来替代浮点数。

浮点数和整数的计算方式都是一样的，只是小数点部分不同而已，那么完全可以把浮点数乘以 10 的幂次得到更准确的精度部分的数字，把自己需要的精度提上来用整数表示。

比如保留 3 位精度，所有浮点数都乘以 1 万来存储(因为第四位不是很准确了)，1.5 变成了 15000 的整数，9.9 变成了 99000 整数存储。

这样整数 15000 乘以 99000 得到的结果，与，整数 30000 除以 2 再乘以 99000 得到的结果是完完全全相等的。

再复杂点 原来 $\frac{2.5}{3.1} * 5.1$ 与 $0.8064 * 5.1$ ，两者都约等于 4.1126，用整数替代， $\frac{2500}{31} * 51$ 与 $80 * 51$ ，等于 4080，把 4080 看作 4.08 虽然精度出现问题，但是前两者结果不一致，而后两者结果完全相同，使用整数来代替小数使得一致性得到了保证。

如果觉得用整数做计算精度问题比较大，我们可以再扩大数值上 10 的幂次，来看看扩大后如果是 $250000 / 31 * 51$ 就等于 411290，是不是发现精度提高了。但问题又来了，乘以 10 的幂次来提高精度时，当浮点数值比较大时就会超出了整数的最大上限 $2^{32} - 1$ 或者 $2^{64} - 1$ 的问题。

如果觉得精度可以接受，并且数值计算的范围肯定会被确定在 32 位或 64 位整数范围内，则可以用这种 `int` 和 `long` 的方式来代替浮点数。

4)用定点数保持一致性并缩小精度问题

浮点数在计算机中的表示方法是用 $V = F = (-1^S) * (1.M) * (2^E)$ 这样的公式表示的，也就是说浮点数的表达其实是模糊的，它用了另一个数乘以 2 的幂次来表示当前的数。

定点数则不同，它把整数部分和小数部分拆分开来，都用整数的形式表示，这样一来计算和表达都使用整数的方式。由于整数的计算是确定的，这样就不会存在误差，缺点是由于拆分了整数和小数，两个部分都要占用空间，所以受到存储位数的限制，占用字节多了通常使用 64 位的 `long` 型整数结构来存储定点数，计算的范围也会相对缩小些。

与浮点数不同的，用定点数来做计算就能保证在各设备上的计算结果一致性，C# 有种整数类型叫 `decimal`¹ 它并非基础类型，是基础类型的补充类型，是 C# 额外造出来的一种类型，可以认为是造了一个类作为数字实例并重载了操作符，它拥有更高的精度却比 `float` 范围还要小。它的内部实现就是定点数的实现方式，我们完全可以把它看作定点数来操作。

C# 的 `decimal` 类型数值有几个特点需要我们重点关注一下，它占用 128 位的存储空间即一个 `decimal` 变量占用 16 个字节，相当于 4 个 `int` 整数大小，或 2 个 `long` 型长整数大小，比 `double` 还要大 1 倍。它的数值范围在 $\pm 1.0 * 10^{28}$ 到 $\pm 7.9 * 10^{28}$ 之间，这么大的占用空间却比 `float` 的取值范围还小。`decimal` 精度比较大，精度范围在 28 个有效位，另外任何与它一起计算的数值都必须先转化为 `decimal` 类型否则就会编译报错，数值不会隐式的自动转换成 `decimal`。

看起来好用的 `decimal` 却不是大部分游戏开发者的首选选择。使用 C# 自带的 `decimal` 定点数在使用时存在诸多问题，最大的问题是无法和浮点数随意的互相转换，因此在计算上也会需要进行一定的封装，要么提前对 `float` 处理，要么在 `decimal` 基础上封装一层外壳(类)以适应所有数值的计算。精度过大导致 CPU 计算消耗量大，128 位的变量 28 位的精度范围，计算起来实在是比较大的负荷，如果大量用于程序内的逻辑计算则 CPU 就会不堪重负。内

¹ adj.十进位的，小数的 n. 小数；十进制

存也是同样的，大量使用时会使得堆栈内存直线飙升，这也间接的增大了 CPU 的消耗。因此它只适用于财务和金融领域的软件，对于游戏和其他普通应用来说实在不太合适，其根源是不需要这么高的精度浪费了诸多设备资源。

实际上大部分项目都是自己实现定点数的，具体实现如前面所说的那样：把整数和小数拆开来存储，用两个 int 整数分别表示整数部分和小数部分，或者用 long 长整型存储(前 32 位存储整数，后 32 位存储浮点数)，long 型存储会更好因为这样就确保定点数的内存就是连续的。这样无论整数还是小数部分都用整数表示，并封装在类中，继而我们需要重载(override)所有的基本计算和比较符号，包括+、-、*、/、==、!=、>、<、>=、<=、这些符号都需要重载，重载范围包括 float 浮点数、double 双精度、int 整数、long 长整数等。除了以上这些，为了能更好的融合定点数与外部数据的逻辑计算，我们还需要为此写一些额外的定点库，包括定点数坐标类，定点数 Quaternion 类等用于定点数的扩展。

看起来比较困难其实并不复杂，只要耐下心来写都会发现不是难事，把定点数与其他类型数字的加减乘除重写一下，如果涉及到更多的数学运算，则再建立一个定点数的数学库，存放一些数学运算的函数，再用写好的定点数类去写些用于扩展的逻辑类仅此而已，都是只要花点时间就能搞定的事，github 上也有很多定点数开源的代码，大可以下载下来参考着写，或者把它从头到尾看一遍，把它改成适合自己项目的。

5)最耗的办法，用字符串替代浮点数

如果想要精确度很高很高，那么就可以用字符串代替浮点数来计算结果。但它的缺点是 CPU 和内存的消耗特别大，如果只是少量使用高精度计算还是可以的。

我们把中小学算术的笔算方式写入到程序里去，把字符串转化为整数，并用整数计算当前位置，接着再用字符串形式存储数字，这样的计算方式就完全不需要担心越界问题，还能自由的控制精度。

缺点是很消耗 CPU 和内存，比如 $123456.78912345 * 456789.2345678$ ，这种类型的计算使用字符串代替浮点数，计算一次相当于计算好几万次的普通浮点数计算量。所以如果程序中对精度要求很高，且计算的次数不大，这种方式可以放在考虑范围内。

6)最差的办法，提高期望值

如果 $\frac{1.55f}{2f}$ 有可能等于 0.7749999 而无法达到 0.775 的目标值时，我们不妨在计算前多加个 0.01，使得 $1.56f / 2f$ ，这样就大概率保证超出 0.775 的结果目标了。

假设结果是模糊的，这个结果范围就有可能是 0.7751，也有可能是 0.7749。正常情况下，由计算机的寄存器和 CPU 决定到底会得到什么结果，但我们不想由它来决定。我们的期望是，宁愿高一点点，跨过这道门槛，也不要少一点点，被门槛拦在外面。

如果正常浮点数的精度不能给我们安全感，那么我们就自己给自己安全感。提高自己的数值一点点，从而降低门槛跨过去的门槛，差不多的就都当做是跨过门槛的。于是就有了 $(X + 1) / Y$ 或者 $(X + 0.001) * Y$ 的写法来度过“精度危机”。

六. 值类型与引用类型

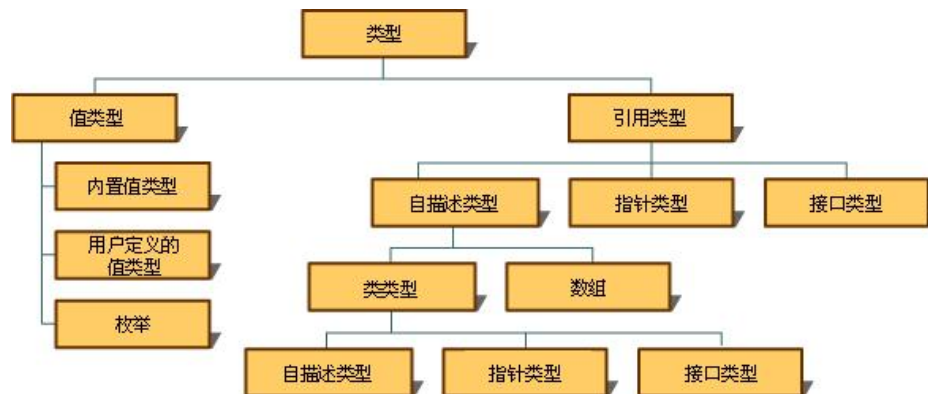


图 1. 值类型与引用类型

引用类型和值类型最显著的一个区别在于变量的赋值问题。**值类型**的变量将**直接获得一个真实的数据副本**，而对**引用类型**的赋值**仅仅是把对象的引用赋给变量**，这样就可能导致多个变量引用到一个实际对象实例上。

除了赋值的区别，引用类型和值类型在内存的分配位置上也有区别。**引用类型**的对象将会在**堆上分配内存**，而**值类型**的对象则会在**堆栈上分配内存**。**堆栈的空间相对有限，但运行效率却高的多。**

1. 引用类型

引用类型**存储对其数据的引用**，引用类型包括：**String**（字符串是一个特殊的引用类型）、**所有数组**（即使其元素为值类型）、**Class** 类型、**委托**。请注意，每个数组都是引用类型，即使其成员是值类型也是如此。

由于每个引用类型都表示一个基础 .NET Framework 类，因此必须在初始化该类型时使用 **New Operator** 关键字。

初始化一个数组：

PS:

```
Dim totals() As Single = New Single(8) {}
```

2. 值类型

如果数据类型**将数据保存在自己的内存分配中**，则该数据类型为“值类型”。值类型包括：**所有数值型数据类型**、**Boolean**、**Char**、**Date**、**所有的结构体**（即使其成员是引用类型）、**枚举**，因为它们的基础类型始终是 **SByte**、**Short**、**Integer**、**Long**、**Byte**、**UShort**、**UInteger** 或 **ULong**。

每个结构体都是值类型，即使其包含引用类型成员。因此，诸如 **Char** 和 **Integer** 之类的值类型由 .NET Framework 结构实现。

3. 不是类型的元素

下面的编程元素不限定为类型，因为不能将任何这些元素指定为已声明元素的数据类型。**命名空间**、**模块**、**事件**、**属性和过程**、**变量**、**常量和字段**。

4. Object 数据类型

可以将引用类型或值类型分配给 **Object** 数据类型的变量。**Object** 变量**始终保存对数据的引用**，而不是数据本身。但是，如果将值类型分配给 **Object** 变量，则其行为就像它保存自己的数据一样。

5. 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）

6. 值类型/引用类型为空时所占用的内存大小

1) 值类型

值类型相较于引用类型好理解多了，他直接分配到栈上，不需要 GC，不需要引用机制。所以它空间的计算也是最简单的。直接计算其中包含的字段空间即可（有一些特性会影响内存的编排，不在讨论范围内）。计算机中的内存通常以字节的形式组织，在对象地址位置可用的最小内存为 1 字节。所以空值类型的对象大小是 1 字节。

2) 引用类型

PS:

```
using System;
```

```
class SmallClass
```

```
{
    private byte[] _largeObj;

    public SmallClass(int size)
    {
        _largeObj = new byte[size];
        _largeObj[0] = 0xAA;
        _largeObj[1] = 0xBB;
        _largeObj[2] = 0xCC;
    }

    public byte[] LargeObj
    {
        {
            get { return this._largeObj; }
        }
    }
}
```

```
class SimpleProgram
```

```
{
    static void Main(string[] args)
    {
        SmallClass smallObj = SimpleProgram.Create(84930, 10, 15, 20, 25);
        return;
    }

    static SmallClass Create(int size1, int size2, int size3, int size4, int size5)
```

```

{
    int objSize = size1 + size2 + size3 + size4 + size5;
    SmallClass smallObj = new SmallClass(objSize);
    return smallObj;
}

```

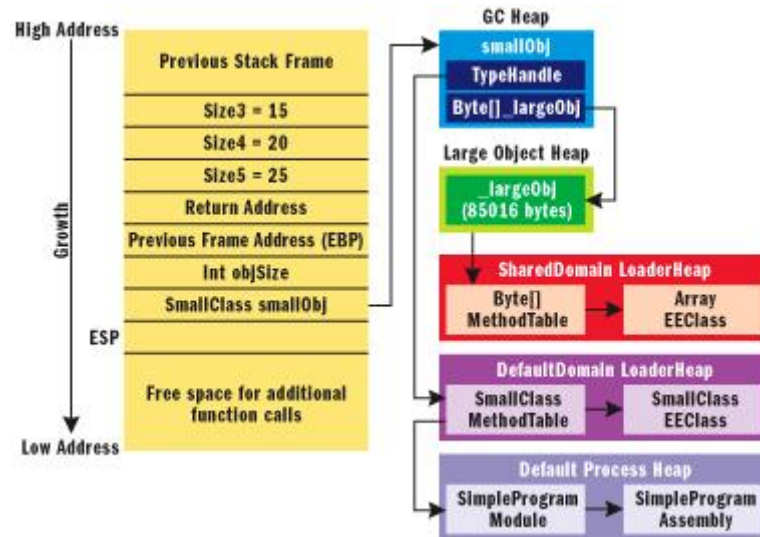


图 2. 运行时堆栈

引用类型变量（如 smallObj）以固定大小（4 字节）存储在栈上，并指向在托管堆上分配的对象实例的地址。smallObj 的实例包含指向相应类型的 MethodTable 的 TypeHandle（类型对象指针）和 syncblk index（同步块索引，用来做线程同步的，这里就不详细讲）。

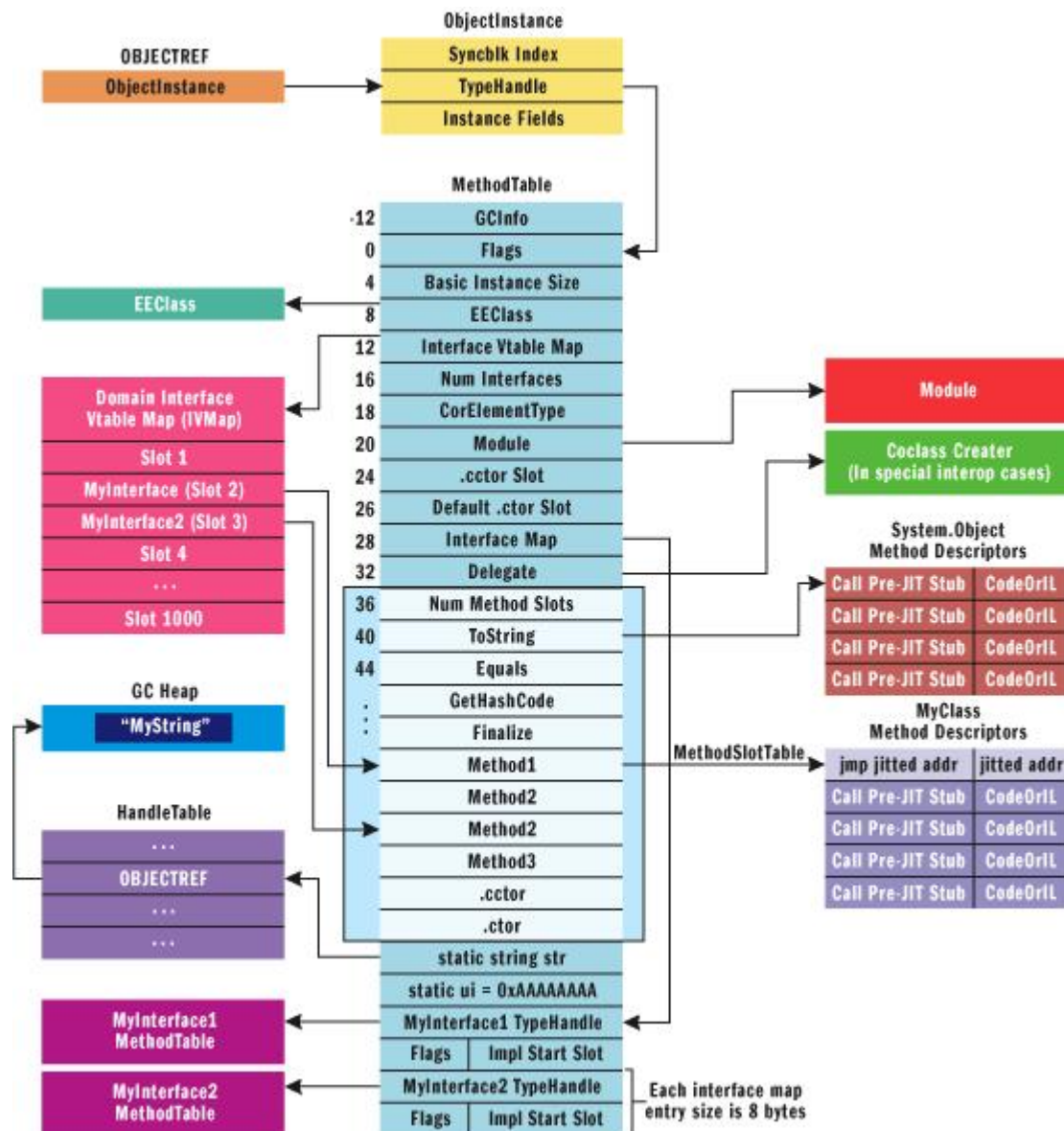


图 3. 引用类型 MethodTable

每种声明的类型都有一个 **MethodTable**，并且同一类型的所有对象实例都指向同一 **MethodTable**。其中包含大量信息，包含有关类型的信息（接口，抽象类，具体类，COM Wrapper 和代理），实现的接口数，用于方法调用的接口映射，方法表中的插槽数以及表的信息，指向实现的插槽数量（包括虚函数的实现也在这个 **MethodTable** 中）。当前的 GC 实现对于一个空类来说，需要至少 12 个字节的对象实例。如果一个类没有定义任何实例字段，它将产生 4 个字节的开销（用于分配到栈上来对他进行引用）。其余部分（8 个字节）的将由同步块索引和类型对象指针占用。所以对于一个空引用类型来说，他所占用的空间大小就是 12。

在 32 位系统下（一个指针的大小为 4 字节），一个空值类型占用内存为 1 字节。

一个空引用类型占用空间为 12 字节，其中包括 4 字节的栈内存分配，4 字节的类型对象指针，4 字节的同步块索引。

或许有些朋友要问，为什么引用类型分配在栈上，指针算在其所占内存，而值类型不算呢？这是因为我们引用类型在 C# 中需要进行 GC 托管，这就需要遍历每个引用类型的内存实例的引用，就算这个引用只在一次栈帧中存在，GC 的时候也会照例执行，所以栈帧中的引用是要算在引用类型所占内存中的。

注意！对于引用类型，使用 sizeof 操作得到的只是指向对象的指针，也就是说全是 4。

3)C++空类型的空间占用

我们都知道，C++当中并没有对位 C#的值类型和引用类型概念，只有一般数据类型、指针、引用，这几个概念。所以概括起来也比较容易一些。即**所有空数据类型所占用的内存大小都是 1 字节！**这是因为，对象需要有不同的地址。有了不同的地址，就可以比较指针和对象的身份。如果不为空的话，编译器会对其进行优化，会是正常的字段大小相加+内存对齐的结果。也就是说，如果 `class A {int i;};`。它的大小就是 4 字节。

4)C++非空类型的空间占用

PS:

非空类型就比较麻烦了，涉及内存对齐的问题。

```
#include <iostream>

#pragma pack(2)
struct S1
{
    S1() { f = 0; s = 0; i = 0; c = 0; }
    float f;
    short s;
    int i;
    char c;
};

#pragma pack(push)
#pragma pack(16)
struct S2
{
    S2() { d = 0; c = 0; i = 0; }
    double d;
    S1 s1;
    char c;
    int i;
};
#pragma pack(pop)

int main()
{
    std::cout << sizeof(S2) << std::endl;
}
```

对于 S1，float 占 4 个字节，short 占 2 个字节，int 占 4 个字节，char 占 1 个字节。

对于 S2，double 占 8 个字节，S1 展开另算，char 占 1 个，int 占 4 个。

结构体在内存中的存放按单元进行存放，每个单元的大小取决于结构体中最大基本类型的大小，为了优化性能会自动进行内存对齐，整个结构体的大小必须是最大成员变量类型所占字节数的整数倍 #pragma pack(16)定义补齐方式为 16 字节（在这里其实和 32 位默认的对齐方式也没差，用 S1 的 double 作为最宽长度即可）

所以，S2 就是 先算 S2 自带的：double(8)+char(1->4)+int(4)=16 再单独算 S1：

$\text{float}(4)+\text{short}(2 \times 4)+\text{int}(4)+\text{char}(1 \times 4)=16$ 。所以 S2 所占总空间为 32。

7. 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，此时它是引用类型的一部分；
- ③ 所有的引用类型肯定都是存放在托管堆上的。
 - ④ 还有一种情况，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。

七. C 和 C++的特点与区别

1. 特点

1) C

- ① 作为一种面向过程的结构化语言，易于调试和维护。
- ② 表现能力和处理能力极强，可以直接访问内存的物理地址。
- ③ C 语言实现了对硬件的编程操作，也适合于应用软件的开发。
- ④ C 语言还具有效率高，可移植性强等特点。

2) C++

- ① 在 C 语言的基础上进行扩充和完善，使 C++兼容了 C 语言的面向过程特点，又成为了一种面向对象的程序设计语言。
- ② 可以使用抽象数据类型进行基于对象的编程。
- ③ 可以使用多继承、多态进行面向对象的编程。
- ④ 可以担负起以模版为特征的泛型化编程。

3) C++与 C 语言的本质差别

在于 C++是面向对象的，而 C 语言是面向过程的。或者说 C++是在 C 语言的基础上增加了面向对象程序设计的新内容，是对 C 语言的一次更重要的改革，使得 C++成为软件开发的重要工具。

八. C 和 C++内存分配问题

1. C 语言编程中的内存基本构成

C 的内存基本上分为 4 部分：静态存储区、堆区、栈区以及常量区。他们的功能不同，对他们使用方式也就不同。

1) 静态存储区（全局区）

- ① 全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域（C++中已经不再这样划分），程序结束释放。
- ② 在所有函数体外定义的是全局量。
- ③ 加了 static 修饰符后不管在哪里都存放在全局区（静态区），在所有函数体外定义的 static 变量表示在该文件中有效，不能 extern 到别的文件用。
- ④ 在函数体内定义的 static 表示只在该函数体内有效。

2) 堆区

- ① 一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收。
- ② 用 malloc, calloc, realloc 等分配内存的函数分配得到的就是在堆上。

3) 栈区

- ① 由编译器自动分配释放。
- ② 函数体中定义的变量通常是在栈上。

4) 常量区

- ① 程序结束则释放。
- ② 函数中的"adgfd"这样的字符串存放在常量区。

2. C++编程中的内存基本构造

在C++中内存分成5个区，分别是堆、栈、全局/静态存储区、常量存储区和代码区。

1) 栈

就是那些由编译器在需要的时候分配，在不需要的时候自动清除的变量的存储区，里面的变量通常是局部变量、函数参数等。

2) 堆

就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

3) 全局/静态存储区

全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量又分为初始化的和未初始化的，在C++里面没有这个区分了，他们共同占用同一块内存区。

4) 常量存储区

这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改）。

5) 代码区（.text 段）

存放代码（如函数），不允许修改（类似常量存储区），但可以执行（不同于常量存储区）。

3. 内存模型组成部分

根据c/c++对象生命周期不同，c/c++的内存模型有三种不同的内存区域，即：自由存储区、动态区、静态区。

1) 自由存储区

局部非静态变量的存储区域，即平常所说的栈。

2) 动态区

用 new，malloc 分配的内存，即平常所说的堆。

3) 静态区

全局变量，静态变量，字符串常量存在的位置。

注意：代码虽然占内存，但不属于c/c++内存模型的一部分。

4. 正在运行着的C编译程序占用的内存

分为5个部分，即代码区、初始化数据区、未初始化数据区、堆区和栈区。

1) 代码区（text segment）

代码区指令根据程序设计流程依次执行，对于顺序指令，则只会执行一次（每个进程），如果反复，则需要使用跳转指令，如果进行递归，则需要借助栈来实现。注意：代码区的指令中包括操作码和要操作的对象（或对象地址引用）。如果是立即数（即具体的数值，如5），将直接包含在代码中。

2)全局初始化数据区/静态数据区（Data Segment）

只初始化一次。

3)未初始化数据区（BSS）

在运行时改变其值。

4)栈区（stack）

由编译器自动分配释放，存放函数的参数值、局部变量的值等，其操作方式类似于数据结构中的栈。

5)堆区（heap）

用于动态内存分配。

5. 分区优势

- ① 代码是根据流程依次执行的，一般只需要访问一次，而数据一般都需要访问多次，因此单独开辟空间以方便访问和节约空间。
- ② 未初始化数据区在运行时放入栈区中，生命周期短。
- ③ 全局数据和静态数据有可能在整个程序执行过程中都需要访问，因此单独存储管理。
- ④ 堆区由用户自由分配，以便管理。

九. C# 结构体和类-Struct and Class

1. Struct

在 C# 中，结构体是值类型数据结构。它使得一个单一变量可以存储各种数据类型的相关数据，struct 关键字用于创建结构体。

1) 定义结构体

为了定义一个结构体，我们必须使用 struct 语句。struct 语句为程序定义了一个带有多个成员的新的数据类型。

例如，我们可以按照如下的方式声明 Book 结构体。

```
struct Books
{
    public string title;
    public string author;
    public string subject;
    public int book_id;
};
```

2)C# 结构体的特点

我们已经有了一个名为 Books 的结构体。C# 中的结构体与传统的 C 或 C++ 中的结构体不同。C# 中的结构体有以下特点：

- ① 结构体可带有方法、字段、索引、属性、运算符方法和事件。
- ② 结构体可定义构造函数，但不能定义析构函数。但是，您不能为结构体定义无参构造函数。无参构造函数(默认)是自动定义的，且不能被改变。
- ③ 与类不同，结构体不能继承其他的结构体或类。
- ④ 结构体不能作为其他结构体或类的基础结构。
- ⑤ 结构体可实现一个或多个接口。
- ⑥ 结构体成员不能指定为 abstract、virtual 或 protected（成员不能添加访问修饰符）。
- ⑦ 当使用 New 操作符创建一个结构体对象时，会调用适当的构造函数来创建结构。与类不同，结构体可以不使用 New 操作符即可被实例化。

- ⑧ 如果不使用 `New` 操作符，只有在所有的字段都被初始化之后，字段才被赋值，对象才被使用。

3) ref struct (0 GC)

这种结构本质是 `struct`，结构存储在栈内存。但是与 `struct` 不同的是，**该结构不允许实现任何接口，并由编译器保证该结构永远不会被装箱**，因此不会给 GC 带来任何的压力。相对的，使用中就会有不能逃逸出栈的强制限制。

`Span<T>` 就是利用 `ref struct` 的产物，成功的封装出了安全且高性能的内存访问操作，且可在大多数情况下代替指针而不损失任何的性能。

PS:

```
ref struct MyStruct{
    public int Value { get; set; }
}
class RefStructGuide
{
    static void Test()
    {
        MyStruct x = new MyStruct();
        x.Value = 100;
        Foo(x); // ok
        Bar(x); // error, x cannot be boxed
    }
    static void Foo(MyStruct x) { }
    static void Bar(object x) { }
}
```

2. Class

当定义了一个类时，就是**定义了一个数据类型的蓝图**，Class 是**引用类型**。这实际上并没有定义任何的数据，但它定义了类的名称意味着什么，也就是说，类的对象由什么组成及在这个对象上可执行什么操作。**对象是类的实例**，构成类的方法和变量的称为类的成员。

1) 类的定义

类的定义是以关键字 `class` 开始，后跟类的名称。类的主体，包含在一对花括号内。下面是类定义的一般形式。

PS:

```
class class_name
{
    // member variables
    <access specifier> <data type> variable1;
    <access specifier> <data type> variable2;
    ...
    <access specifier> <data type> variableN;
    // member methods
    <access specifier> <return type> method1(parameter_list)
    {
        // method body
    }
}
```

```

    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
    <access specifier> <return type> methodN(parameter_list)
    {
        // method body
    }
}

```

访问标识符 <access specifier> 指定了对类及其成员的访问规则。如果没有指定，则使用默认的访问标识符。**类的默认访问标识符是 internal，成员的默认访问标识符是 private。**

数据类型 <data type> 指定了变量的类型，**返回类型** <return type> 指定了返回的方法返回的数据类型。

如果要访问类的成员，你要使用点（.）运算符。点运算符链接了对象的名称和成员的名称。

2)成员函数和封装

类的成员函数是一个在类定义中有它的定义或原型的函数，就像其他变量一样。作为类的一个成员，它能在类的任何对象上操作，且能访问该对象的类的所有成员。

成员变量是对象的属性（从设计角度），且它们保持私有来实现封装。这些变量只能使用公共成员函数来访问。

```

using System;
namespace BoxApplication
{
    class Box
    {
        private double length;    // 长度
        private double breadth;    // 宽度
        private double height;    // 高度
        public void setLength( double len )
        {
            length = len;
        }

        public void setBreadth( double bre )
        {
            breadth = bre;
        }

        public void setHeight( double hei )
        {
            height = hei;
        }

        public double getVolume()
    }
}

```

```

        {
            return length * breadth * height;
        }
    }
}
class Boxtester
{
    static void Main(string[] args)
    {
        Box Box1 = new Box();           // 声明 Box1，类型为 Box
        Box Box2 = new Box();           // 声明 Box2，类型为 Box
        double volume;                  // 体积

        // Box1 详述
        Box1.setLength(6.0);
        Box1.setBreadth(7.0);
        Box1.setHeight(5.0);

        // Box2 详述
        Box2.setLength(12.0);
        Box2.setBreadth(13.0);
        Box2.setHeight(10.0);

        // Box1 的体积
        volume = Box1.getVolume();
        Console.WriteLine("Box1 的体积: {0}", volume);

        // Box2 的体积
        volume = Box2.getVolume();
        Console.WriteLine("Box2 的体积: {0}", volume);

        Console.ReadKey();
    }
}
}

```

3)C# 类的静态成员

我们可以使用 **static** 关键字把类成员定义为静态的。当我们声明一个类成员为静态时，意味着无论有多少个类的对象被创建，只会有一个该静态成员的副本。

关键字 **static** 意味着类中只有一个该成员的实例。静态变量用于定义常量，因为它们的值可以通过直接调用类而不需要创建类的实例来获取。静态变量可在成员函数或类的定义外部进行初始化，也可以在类的定义内部初始化静态变量。

```

using System;
namespace StaticVarApplication
{

```

```
class StaticVar
{
    public static int num;
    public void count()
    {
        num++;
    }
    public int getNum()
    {
        return num;
    }
}

class StaticTester
{
    static void Main(string[] args)
    {
        StaticVar s1 = new StaticVar();
        StaticVar s2 = new StaticVar();
        s1.count();
        s1.count();
        s1.count();
        s2.count();
        s2.count();
        s2.count();
        Console.WriteLine("s1 的变量 num: {0}", s1.getNum());
        Console.WriteLine("s2 的变量 num: {0}", s2.getNum());
        Console.ReadKey();
    }
}
```

我们也可以把一个成员函数声明为 **static**。这样的函数只能访问静态变量。静态函数在对象被创建之前就已经存在。

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
        public static int num;
        public void count()
        {
            num++;
        }
        public static int getNum()
        {
```

```

        return num;
    }
}
class StaticTester
{
    static void Main(string[] args)
    {
        StaticVar s = new StaticVar();
        s.count();
        s.count();
        s.count();
        Console.WriteLine("变量 num: {0}", StaticVar.getNum());
        Console.ReadKey();
    }
}
}

```

将类成员函数声明为 **public static**，无需实例化即可调用类成员函数。反之，如果不声明为 **static**，即使和 **Main** 方法从属于同一个类，也必须经过实例化

3. 类与结构体的选择

首先明确，类的对象是存储在堆空间中，结构体存储在栈中。堆空间大，但访问速度较慢，栈空间小，访问速度相对更快。故而，当我们描述一个轻量级对象的时候，结构体可提高效率，成本更低。当然，这也得从需求出发，假如我们在传值的时候希望传递的是对象的引用地址而不是对象的拷贝，就应该使用类了。

结构体有性能优势，类有面向对象的扩展优势。

4. 类与结构体的区别

- ① 结构体中声明的字段无法赋予初值，但是类可以。
- ② 类可以有有参的构造函数，而结构体只能是有参的构造函数，而且在有参的构造函数中必须初始化所有成员。
- ③ 结构体的构造函数中，必须为结构体所有字段赋值，类的构造函数无此限制。
- ④ 结构体是值类型，它在栈中分配空间；而类是引用类型，它在堆中分配空间，栈中保存的只是引用。
- ⑤ 类允许继承和被继承；而结构体则不允许，它只能继承接口。
- ⑥ 类可以为抽象类，结构体不支持抽象。
- ⑦ 结构常用于数据存储，类 **class** 多用于行为。

结构体类型直接存储成员数据，而其他类的数据位于堆中，位于栈中的变量保存的是指向堆中数据对象的引用。由于结构体是值类型，并且直接存储数据，因此在一个对象的主要成员为数据且数据量不大的情况下，使用结构体会带来更好的性能。因为结构体是值类型，因此在为结构分配内存，或者当结构体超出了作用域被删除时，性能会非常好，因为他们将内联或者保存在堆栈中。当把一个结构体类型的变量赋值给另一个结构体时，对性能的影响取决于结构的大小，如果结构的数据成员非常多而且复杂，就会造成损失。

十. 构造函数

构造函数就像一个与类同名的方法，但它是**唯一的方法**。即使没有创建，编译器也会在创建类的对象时在内存中创建一个默认构造函数。构造函数用于使用一些默认值初始化对象。

默认构造函数、参数化构造函数、复制构造函数、静态构造函数和私有构造函数都是不同的构造函数类型。

- ① 调用构造函数，可以在一个类通过 **new** 关键字**实例化对象时**，对这个新实例化的对象进行**初始化**。
- ② 不带参数的构造函数称为“**默认构造函数**”。不管在什么时候，只要是使用 **new** 运算符去实例化对象，并且不为其提供任何形式的参数，系统就会去调用默认的构造函数。如果类不是 **static** 的，即这个类不是静态的，C# 编译器将会为不具备构造函数的类提供一个公共的默认构造函数，以便该类可以实例化。
- ③ 要想**阻止**一个类被**实例化**，可以通过将构造函数**设置为私有**构造函数来实现。
- ④ 结构体的构造函数与类的构造函数类似却又不完全相同，结构体 **struct** 并不能包含显式的默认构造函数，这是由于编译器将会自动的提供一个构造函数。编译器提供的构造函数会将结构体中的每个字段都初始化为默认值。只有当 **struct** 结构体用 **new** 关键字进行实例化时，才会调用编译器提供的这个默认构造函数。

十一. 虚函数

- ① 多态性是面向对象程序设计的重要特征之一。在 C++ 中，**编译时**的多态性主要是通过**函数重载**和**运算符重载**实现的。**运行时**的多态性主要是通过**虚函数**实现的。
- ② **虚函数是重载的另一种表现形式**，它提供了一种更为灵活的运行时的多态性机制。虚函数允许函数调用与函数体之间的联系在运行时才建立，也就是在运行时才决定进行何种操作，即**动态编译**。
- ③ 虚函数首先是基类中的成员函数，但这个成员函数前面缀上了**关键字 Virtual**，并在派生类中被重载。在基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。虚函数在派生类中**重新定义时**，其函数原型，包括其**返回值类型、函数名、参数类型的顺序**，都必须与基类中的原型完全一致。

1. 虚函数

虚函数，是指被 **virtual** 关键字修饰的成员函数。在某基类中声明为 **virtual** 并在一个或多个派生类中被重新定义的成员函数。

2. 纯虚函数

Virtual 函数类型 函数名(参数列表) = 0;

纯虚函数**没有函数体**，**不具有函数的功能**，**不能被调用**。

3. 抽象类

C#抽象类是特殊的类，只是不能被实例化；除此以外，具有类的其他特性；重要的是抽象类可以包括抽象方法，这是普通类所不能的。**抽象方法只能声明于抽象类中，且不包含任何实现**，派生类必须覆盖它们。另外，抽象类可以派生自一个抽象类，可以覆盖基类的抽象方法也可以不覆盖，如果不覆盖，则其派生类必须覆盖它们。

在 C++ 中，包含纯虚函数的类称为抽象类。

十二. 接口(Interface)和抽象类的区别

抽象类表示该类中可能已经有一些方法的具体定义，但接口就往往只能定义各个方法的界面，不能定义具体的实现代码在成员方法中。类是子类用来继承的，当父类已经有实际功能的方法时该方法在子类中可以不必实现，直接引用父类的方法，子类也可以重写该父类的方法。**实现接口的时候必须要实现接口中所有的方法，不能遗漏任何一个**。

1) 相同点

接口和抽象类都不能直接实例化。

2) 不同点

- ① 抽象类用 **abstract** 修饰、接口用 **interface** 修饰。
- ② 抽象类中的方法可以实现，也可以不实现，有抽象方法的类一定要用 **abstract** 修饰，接口中的方法不允许实现。
- ③ 抽象类只能单继承，接口支持多继承。
- ④ 抽象类可以有构造方法，接口不能有构造方法。
- ⑤ 接口只负责功能的定义，通过接口来规范类的，（有哪些功能），而抽象类即负责功能的定义又可以实现功能（实现了哪些功能）。

十三. C++的多态

C++的多态性用一句话概括，就是在基类的函数前加上 **virtual** 关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。

- ① 用 **virtual** 关键字声明的函数叫做虚函数，虚函数肯定是类的成员函数。
- ② 存在虚函数的类都有一个一维的虚函数表叫做虚表，类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。
- ③ 多态性是一个接口多种实现，是面向对象的核心，分为类的多态性和函数的多态性。
- ④ 多态用虚函数来实现，结合动态绑定。
- ⑤ 纯虚函数是虚函数再加上 **= 0**。
- ⑥ 抽象类是指包括至少一个纯虚函数的类。

包含纯虚函数——**virtual void fun()=0;**的类，即抽象类，必须在子类实现这个函数，即先有名称，没有内容，在派生类实现内容。

十四. C#中四种访问修饰符

1. 属性修饰符

Serializable 按值将对象封送到远程服务器。

STAThread 是单线程套间的意思，是一种线程模型。

MAThread 是多线程套间的意思，也是一种线程模型。

2. 存取修饰符

public: 存取不受限制。

private: 只有包含该成员的类可以存取。

internal: 只有当前命名空间可以存取。只能在包含该类的程序集中访问该类。

protected: 只有包含该成员的类以及派生类可以存取。

protected internal: **protected** + **internal**

3. 类修饰符

abstract: 抽象类。指示一个类只能作为其它类的基类。

sealed: 密封类。指示一个类不能被继承。理所当然，密封类不能同时又是抽象类，因为抽象总是希望被继承的。

4. 成员修饰符

abstract: 指示该方法或属性没有实现。

sealed: 密封方法。可以防止在派生类中对该方法的 **override**（重载）。不是类的每个成员方法都可以作为密封方法密封方法，必须对基类的虚方法进行重载，提供具体的实现方法。所以，在方法的声明中，**sealed** 修饰符总是和 **override** 修饰符同时使用。

delegate: 委托。用来定义一个函数指针。C#中的事件驱动是基于 `delegate + event` 的。

const: 指定该成员的值只读不允许修改。

event: 声明一个事件。

extern: 指示方法在外部实现。

override: 重写。对由基类继承成员的新实现。

readonly: 指示一个域只能在声明时以及相同类的内部被赋值。

static: 指示一个成员属于类型本身，而不是属于特定的对象。即在定义后可不经实例化，就可使用。

virtual: 指示一个方法或存取器的实现可以在继承类中被覆盖。

new: 在派生类中隐藏指定的基类成员，从而实现重写的功能。若要隐藏继承类的成员，请使用相同名称在派生类中声明该成员，并用 `new` 修饰符修饰它。

十五. In、ref 和 out 关键字

ref、**out** 都是引用传递，而引用传递是传递的这个参数的实际地址，因此在引用传递后的参数使用都会改变参数原有的数值。引用传递都需要显示传递到方法，即在方法的参数前加上相对应的关键字。

1. ref

ref 关键字有进有出，不仅能将参数传递进去还能将参数重新传递出来，通俗的说就是在函数里对变量的改变会在函数结束时改变变量的值，并将被改变的变量值重新传递出去，因此需要在传递进入函数前初始化。

2. out

而 **out** 关键字是只出不进的，将参数传递进去时值是无效的，**out** 关键字会要求把参数的值清空，然后在函数内部对其重新赋值，所以无法将一个值从 **out** 传递出去。

虽然使用 **ref** 和 **out** 都能返回多个值，但是必须确保函数方法的定义及调用都是显示使用的 **ref** 和 **out** 关键字。

3. in

in 关键字会按引用传递参数，但确保未修改参数。它让形参成为实参的别名，这必须是变量。换言之，对形参执行的任何操作都是对实参执行的。它类似于 **ref** 或 **out** 关键字，不同之处在于 **in** 参数无法通过调用的方法进行修改变。 **out** 参数必须由调用的方法进行修改变，这些修改变在调用上下文中是可观察的，而 **ref** 参数是可以修改变的。

PS:

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);    // 值依旧是 44
```

```
void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

这个示例说明调用函数处通常不需要 **in** 修饰符，仅在方法声明中需要它。

in 关键字还作为 `foreach` 语句的一部分，或作为 `LINQ` 查询中 `join` 子句的一部分，与泛型类型参数一起使用来指定该类型参数为逆变。

作为 **in** 参数传递的变量在方法调用之前必须进行初始化。但是，所调用的方法可能

不会分配值或修改参数。

尽管 `in`、`out` 和 `ref` 参数修饰符被视为签名的一部分，但在单个类型中声明的成员不能仅因 `in`、`ref` 和 `out` 而签名不同。因此，如果唯一的不同是一个方法采用 `ref` 或 `out` 参数，而另一个方法采用 `in` 参数，则无法重载这两个方法。

PS:

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

仅存在 `in` 时可以重载。

PS:

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

4. `ref` 和 `out` 的相同点

都能返回多个返回值。

若要使用 `ref` 和 `out` 参数，则方法定义和调用方法都必须显示使用 `ref` 和 `out` 关键字。

5. `ref` 和 `out` 的不同点

`ref` 指定的参数在函数调用时必须初始化，不能为空的引用。`Out` 指定的参数在函数调用时可以不初始化。

`Out` 指定的参数在进入函数时会清空自己，必须在函数内部对其赋值。`Ref` 指定的参数不需要内部赋值。

6. `In`、`ref` 和 `out` 参数的限制

不能将 `in`、`ref` 和 `out` 关键字用于以下几种方法。

- ① 异步方法，通过使用 `async` 修饰符定义。
- ② 迭代器方法，包括 `yield return` 或 `yield break` 语句。
- ③ 扩展方法的第一个参数不能有 `in` 修饰符，除非该参数是结构。
- ④ 扩展方法的第一个参数，其中该参数是泛型类型（即使该类型被约束为结构。）

十六. 方法参数-Params

使用 `params` 关键字可以指定采用数目可变的参数的方法参数。（参数类型必须是一维数组）

在方法声明中的 `params` 关键字之后不允许有任何其他参数，并且在方法声明中只允许有一个 `params` 关键字。

如果 `params` 参数的声明类型不是一维数组，则会发生编译器错误。

使用 <code>params</code> 参数调用方法时，可以传入的内容	数组元素类型的参数的逗号分隔列表。
	指定类型的参数的数组。
	无参数。如果未发送任何参数，则 <code>params</code> 列表的长度为零。

PS:

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // 我们可以发送以逗号分隔的指定类型的参数列表。
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // params 参数接受零个或多个参数
        // 下面的调用语句只显示一个空行
        UseParams2();

        // 可以传递数组参数，只要数组类型与被调用方法的参数类型匹配
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // 以下调用会导致编译器错误，因为无法将对象数组转换为整数数组。
        //UseParams(myObjArray);

        // 以下调用不会导致错误，但整个整数数组会成为 params 数组的第一个元素。
        UseParams2(myIntArray);
    }
}
```

```

}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/

```

十七. 分部类和方法

拆分一个类、一个结构、一个接口或一个方法的定义到两个或更多的文件中是可能的。每个源文件包含类型或方法定义的一部分，编译应用程序时将把所有部分组合起来。

1. 分部类

在以下几种情况下需要拆分类定义：

- ① 处理大型项目时，使一个类分布于多个独立文件中可以让多位程序员同时对该类进行处理。
- ② 当使用自动生成的源文件时，你可以添加代码而不需要重新创建源文件。Visual Studio 在创建 Windows 窗体、Web 服务包装器代码等时会使用这种方法。你可以创建使用这些类的代码，这样就不需要修改由 Visual Studio 生成的文件。
- ③ 使用源生成器在类中生成附加功能时。

若要拆分类定义，请使用 `partial` 关键字修饰符，如下所示：

```

public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}

```

partial 关键字指示可在命名空间中定义该类、结构或接口的其他部分。**所有部分都必须使用 `partial` 关键字**。在编译时，各个部分都必须用来形成最终的类型。**各个部分必须具有相同的可访问性**，如 `public`、`private` 等。

如果将**任意部分**声明为**抽象的**，则**整个**类型都被视为**抽象的**。如果将**任意部分**声明为**密封的**，则**整个**类型都被视为**密封的**。如果任意部分声明基类型，则整个类型都将继承该类。

指定基类的所有部分必须一致，但忽略基类的部分仍继承该基类型。各个部分可以指定不同的基接口，最终类型将实现所有分部声明所列出的全部接口。在某一分部定义中声明的任何类、结构或接口成员可供所有其他部分使用。最终类型是所有部分在编译时的组

合。

partial 修饰符不可用于委托或枚举声明中。

2. 分部方法

分部类或结构可以包含分部方法。类的一个部分包含方法的签名。可以在同一部分或另一部分中定义实现。如果未提供该实现，则会在编译时删除方法以及对方法的所有调用。根据方法签名，可能需要实现。在以下情况下，不需要使用分部方法即可实现：

- ① 没有任何可访问性修饰符（包括默认的 专用）。
- ② 返回 void。
- ③ 没有任何输出参数。
- ④ 没有以下任何修饰符：virtual、override、sealed、new 或 extern。

任何不符合所有这些限制的方法（例如 public virtual partial void 方法）都必须提供实现。此实现可以由源生成器提供。

十八. 面向对象的三大特点

面向对象是现实世界中万事万物的一种抽象概述，即万物皆对象，我们在现实世界中接触的万事万物就是一个面向对象的过程；例如电脑、汽车、房子等就是一个对象，我们使用这些对象也是一个面向对象的过程。

1. 继承

提高代码重用度，增强软件可维护性的重要手段，符合开闭原则。继承最主要的作用就是把子类的公共属性集合起来，便与共同管理，使用起来也更加方便。你既然使用了继承，那代表着你认同子类都有一些共同的特性，所以你把这共同的特性提取出来设置为父类。继承的传递性：传递机制 a->b; b->c; c 具有 a 的特性。

1) 继承的单根性

在 C# 中一个类只能继承一个类，不能有多个父类。

2. 封装

封装是将数据和行为相结合，通过行为约束代码修改数据的程度，增强数据的安全性，**属性是 C# 封装实现的最好体现**。就是将一些复杂的逻辑经过包装之后给别人使用就很方便，别人不需要了解里面是如何实现的，只要传入所需要的参数就可以得到想要的结果。**封装的意义**在于保护或者防止代码（数据）被我们无意中破坏。

1) 定义

封装就是把客观事物封装起来变成抽象的类，这个类可以把自己的属性和方法提供给可信的类或对象操作使用，并对不可行的类进行信息隐藏。

2) 理解

封装是对象和类的主要特性，简单的说，一个类就是一个封装了数据以及操作这些数据的代码逻辑实体，在一个封装的对象内部，某些代码或某些数据可以使私有的，不能被外界所访问。通过这种方式，对象内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

3. 多态性

多态性是指同名的方法在不同环境下，自适应的反应出不同得表现，是方法动态展示的重要手段。**多态就是一个对象多种状态，子类对象可以赋值给父类型的变量。**

1) 定义

多态就是指一个类实例（封装的对象）的相同方法在不同情形下被调用所展现的不同形式。

2)理解

多态机制使具有不同内部结构的对象可以共享相同的外部接口,这就意味着针对不同对象的具体操作不同,但通过共同的类,他们可以通过相同的方式予以调用,并展现出不同的结果形态;例如一辆汽车,用来比赛,展现出的用途就是赛车;用来上班,展现的用途就是上班代步;用来卖菜,展现的用途就是买菜车。

十九. 面向对象程序设计方法的主要优点

可提高程序的**重用性**。

可**控制**程序的**复杂性**。

可**改善**程序的**可维护性**。

能够**更好地**支持大型程序设计。

增强了计算机**处理信息的范围**。

二十. 模式匹配

有了 if-else、as 和强制类型转换,为什么要使用模式匹配呢?有三方面原因:**性能**、**鲁棒性**和**可读性**。

为什么说性能也是一个原因呢?因为 C# 编译器会根据我们的模式编译出最优的匹配路径。

1. 性能

PS:

```
int Match(int v)
{
    if (v > 3)
    {
        return 5;
    }
    if (v < 3)
    {
        if (v > 1)
        {
            return 6;
        }
        if (v > -5)
        {
            return 7;
        }
        else
        {
            return 8;
        }
    }
    return 9;
}
```

如果改用模式匹配,配合 switch 表达式写法则变成下面这样。

PS:

```

int Match(int v)
{
    return v switch
    {
        > 3 => 5,
        < 3 and > 1 => 6,
        < 3 and > -5 => 7,
        < 3 => 8,
        _ => 9
    };
}

```

以上代码会被编译器编译为

PS:

```

int Match(int v)
{
    if (v > 1)
    {
        if (v <= 3)
        {
            if (v < 3)
            {
                return 6;
            }
            return 9;
        }
        return 5;
    }
    if (v > -5)
    {
        return 7;
    }
    return 8;
}

```

使用模式匹配时，编译器选择了更优的比较方案，我们在编写的时候无需考虑如何组织判断语句，心智负担降低，并且代码的可读性和简洁程度显然更好，有哪些条件分支一目了然。

甚至遇到类似以下的情况时

PS:

```

int Match(int v)
{
    return v switch
    {
        1 => 5,
        2 => 6,

```

```

        3 => 7,
        4 => 8,
        _ => 9
    };
}

```

编译器会直接将代码从条件判断语句编译成 `switch` 语句：

PS:

```

int Match(int v)
{
    switch (v)
    {
        case 1:
            return 5;
        case 2:
            return 6;
        case 3:
            return 7;
        case 4:
            return 8;
        default:
            return 9;
    }
}

```

如此一来所有的判断都不需要比较（因为 `switch` 可根据 `HashCode` 直接跳转）。编译器非常智能地为我们选择了最佳的方案。

2. 鲁棒性

那鲁棒性从何谈起呢？假设我们漏掉了一个分支：

PS:

```

int v = 5;
var x = v switch
{
    > 3 => 1,
    < 3 => 2
};

```

此时编译的话，编译器就会警告我们漏掉了 `v` 可能为 `3` 的情况，帮助减少程序出错的可能性。

3. 可读性

假设我们现在有下面这样的东西：

PS:

```

abstract class Entry { }
class UserEntry : Entry
{
    public int UserId { get; set; }
}

```

```

class DataEntry : Entry
{
    public int DataId { get; set; }
}
class EventEntry : Entry
{
    public int EventId { get; set; }
    // 如果 CanRead 为 false 则查询的时候直接返回空字符串
    public bool CanRead { get; set; }
}

```

现在有接收类型为 Entry 的参数一个函数，该函数根据不同类型的 Entry 去数据库查询对应的 Content，那么只需要写：

PS:

```

string QueryMessage(Entry entry)
{
    return entry switch
    {
        UserEntry u => dbContext1.User.FirstOrDefault(i => i.Id == u.UserId).Content,
        DataEntry d => dbContext1.Data.FirstOrDefault(i => i.Id == d.DataId).Content,
        EventEntry { EventId: var eventId, CanRead: true } =>
dbContext1.Event.FirstOrDefault(i => i.Id == eventId).Content,
        EventEntry { CanRead: false } => "",
        _ => throw new ArgumentException("无效的参数")
    };
}

```

更进一步，假如 Entry.Id 分布在了数据库 1 和 2 中，如果在数据库 1 当中找不到则需要去数据库 2 进行查询，如果 2 也找不到才返回空字符串，由于 C# 的模式匹配支持递归模式，因此只需要这样写：

PS:

```

string QueryMessage(Entry entry)
{
    return entry switch
    {
        UserEntry u => dbContext1.User.FirstOrDefault(i => i.Id == u.UserId) switch
        {
            null => dbContext2.User.FirstOrDefault(i => i.Id == u.UserId)?.Content ?? "",
            var found => found.Content
        },
        DataEntry d => dbContext1.Data.FirstOrDefault(i => i.Id == d.DataId) switch
        {
            null => dbContext2.Data.FirstOrDefault(i => i.Id == d.DataId)?.Content ?? "",
            var found => found.Content
        },
        EventEntry { EventId: var eventId, CanRead: true } =>

```

```
dbContext1.Event.FirstOrDefault(i => i.Id == eventId) switch
{
    null => dbContext2.Event.FirstOrDefault(i => i.Id == eventId)?.Content ?? "",
    var found => found.Content
},
EventEntry { CanRead: false } => "",
_ => throw new ArgumentException("无效的参数")
};
}
```

就全部搞定了，代码非常简洁，而且数据的流向一眼就能看清楚，就算是没有接触过这部分代码的人看一下模式匹配的过程，也能一眼就立刻掌握各分支的情况，而不需要在一堆的 if-else 当中梳理这段代码到底干了什么。

二十一. 六大原则

1. 单一职责原则-Single Responsibility Principle

1) 定义

即一个类应该只负责一个职责。假设一个类叫做 T，它下面负责两个职责的方法分别是 x 和 y，假如 x 的职责发生改变是去修改了类 T，就有可能造成方法 y 发生故障。

2) 优点

- ① 降低类的复杂度，一个类只负责一项职责。
- ② 提高类的可读性和可维护性。
- ③ 降低变更引起的风险。

2. 里氏替换原则-Liskov Substitution Principle

1) 定义

如果对每个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型；简单的可以理解为所有引用基类的地方必须能透明的是用子类的对象，在子类中尽量不要重写或重载父类的方法。

3. 依赖倒转原则-Dependence Inversion Principle

1) 定义

高层模块不应该依赖于低层模块，二者都应该依赖其抽象；抽象不应该依赖于细节，细节应该依赖于抽象。因为相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建的架构要比以细节为基础的架构要稳定的多。依赖倒置的中心思想是面向接口编程。上层模块不应该依赖于下层模块，应该依赖于接口。从而使得下层模块依赖于上层的接口，降低耦合度，提高系统的弹性。

4. 接口隔离原则-InterfaceSegregation Principles

1) 定义

一个类不应该依赖他不需要的接口；一个类对另一个类的依赖应该建立在最小接口上。比如类 A 通过接口 E 依赖类 B，类 C 通过接口 E 依赖类 D，如果接口 E 对于类 A 和类 C 来说不是最小接口的话，则类 B 和类 D 必须去实现他们不需要的方法。这个时候我们将臃肿的接口拆分成独立的几个接口，类 A 和类 C 分别与他们需要的接口建立依赖关系。这就是接口隔离原则。

5. 迪米特原则-Law of Demeter (也称为最少知识原则 Least Knowledge Principle)

1) 定义

一个对象应该对其他对象保持最少的了解。**类与类之间的关系越密切，耦合度越大**。迪米特原则又叫最少知道原则，即一个类对自己依赖的类知道的越少越好。也就是说，**无论被依赖的类多么复杂，都尽量将逻辑封装在类的内部。对外只提供 public 方法，而不对外泄露任何信息**。迪米特原则还有个更简单的定义：只与直接的朋友通信。什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就称这两个对象之间是朋友关系。耦合的方式很多，依赖，关联，组合，聚合等。其中，我们称出现成员变量，方法参数，方法返回值中的类为直接的朋友，而出现在局部变量中的类不是直接的朋友。也就是说，陌生的类最好不要以局部变量的形式出现在类的内部。

6. 开闭原则-Open Close Principle

1) 定义

一个软件实体如类，模块和函数应该对扩展开放，对修改关闭。**用抽象构建框架，用实现扩展细节**。当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。当我们遵循前面介绍的五大原则，以及使用 23 种设计模式的目的就是遵循开闭原则。简单的理解就是构建框架的时候要保留足够的扩展性，通过扩展来实现修改代码而不是直接修改代码。

二十二. 运算符重载

二十三. C#属性-Property

属性 (Property) 是类 (class)、结构 (structure) 和接口 (interface) 的命名 (named) 成员，它提供灵活的机制来读取、写入或计算私有字段的值。**属性可用作公共数据成员，但它们是称为“访问器”的特殊方法**。属性使得我们可以轻松访问数据，还有助于**提高方法的安全性和灵活性**。

属性具有许多用途：它们可以先验证数据，再允许进行更改；可以在类上透明地公开数据，其中数据是从某个其他源（如数据库）检索到的；可以在数据发生更改时采取措施，例如引发事件或更改其他字段的值。

类或结构中的成员变量或方法称为字段 (Field)。**属性 (Property) 是字段 (Field) 的扩展**，且可使用相同的语法来访问。它们使用访问器 (accessors) 让私有字段的值可被读写或操作。

总而言之，无论什么时候，当我们想让类内部的数据被外界访问到时(无论是 public 还是 protected)，一定要用属性。

使用 Property，我们可以得到如下好处：

- ① 数据绑定的支持。
- ② 对于需求变化有更强的适应性，更方便的修改实现方法。

记住，现在多花 1 分钟使用 Property，会在修改程序以适应设计变化时，为我们节约 n 小时。

1. 概述

- ① 属性允许类公开获取和设置值的公共方法，而**隐藏实现**或验证代码。
- ② get 属性访问器用于返回属性值，而 set 属性访问器用于分配新值。在 C# 9 及更高版本中，**init 属性访问器仅用于在对象构造过程中分配新值**。这些访问器可以具有不同的访问级别。

- ③ **value** 关键字用于定义由 **set** 或 **init** 访问器分配的值。
- ④ 属性可以是读-写属性（既有 **get** 访问器又有 **set** 访问器）、只读属性（有 **get** 访问器，但没有 **set** 访问器）或只写访问器（有 **set** 访问器，但没有 **get** 访问器）。
只写属性很少出现，常用于限制对敏感数据的访问。
- ⑤ 不需要自定义访问器代码的简单属性可以作为表达式主体定义或自动实现的属性来实现。
- ⑥ 与字段不同，属性不会被归类为变量。因此，**不能将属性作为 ref 或 out 参数传递**。

2. 属性和字段的区别

1) 字段

字段（Field）是一种表示与对象或类关联的**变量的成员**，字段声明用于引入一个或多个给定类型的字段。**字段是类内部用的**，**private** 类型的变量(字段)，通常字段写法都是加个"**_**"符号，然后声明只读属性，字段用来储存数据。

2) 属性

属性（Property）是另一种类型的类成员，定义属性的目的是在于便于一些私有字段的访问。类提供给外部调用时可以设置或读取一个值，**属性则是对字段的封装**，将字段和访问自己字段的方法组合在一起，提供灵活的机制来读取、编写或计算私有字段的值。属性有自己的名称，并且包含 **get** 访问器和 **set** 访问器。

3) 适用情况

类型	适用情况	说明
公共字段	允许自由读写。	如果均满足上述条件，那么我们可以大胆地使用公共字段。
	取值范围只收数据类型约束而无其他任何特定限制。	
	值的变动不需要引发类中其它任何成员的相应变化。	
属性	要求字段只能读或只能写。	如果满足上述条件中的任何一个，就应该使用属性。
	需要限定字段的取值范围。	
	在改变一个字段的值的时候希望改变对象的其它一些状态。	

3. 访问器

属性（Property）的访问器（accessor）包含有助于获取（读取或计算）或设置（写入）属性的可执行语句。访问器（accessor）的声明**可包含一个 **get** 访问器、一个 **set** 访问器**，或者同时包含二者。

PS:

// 声明类型为 **string** 的 **Code** 属性

```
public string Code
{
    get
    {
        return code;
    }
}
```

```

    }
    set
    {
        code = value;
    }
}

```

1) Get 访问器

get 访问器的正文类似于方法，**它必须返回属性类型的值**。执行 get 访问器等效于读取字段的值，若在属性定义中省略了该访问器，则不能在其他类中获取私有类型的字段值，因此也称为**只读属性**。例如，在从 get 访问器返回私有变量且已启用优化时，对 get 访问器方法的调用由编译器内联，因此不存在方法调用开销。但是，无法内联虚拟 get 访问器方法，因为编译器在编译时不知道在运行时实际可调用哪些方法。以下示例显示一个 get 访问器，它返回私有字段 _name 的值。

PS:

```

class Employee
{
    private string _name; // 名字字段
    //public string Name => _name; //名字属性
    public string Name //成员属性(即对外公开的成员变量), the Name property
    {
        get
        {
            return name;
        }
    }
}

```

注意！ get 访问器必须以 return 或 throw 语句结尾，且控件不能超出访问器正文。

尤其注意！ 在使用 get 访问器更改对象的状态是一种错误的编程样式。例如，以下访问器在每次访问 number 字段时都产生更改对象状态的副作用。

PS:

```

public int Number
{
    get
    {
        return number++; //错误，在 get 访问器中不能修改 number 的值
    }
}

```

2) Set 访问器

set 访问器类似于返回类型为 void 的方法。它使用名为 **value** 的**隐式参数**，该参数的类型为属性的类型。在下面的示例中，将 set 访问器添加到 Name 属性。

PS:

```

class Student
{
    private string _name; // 私有类型的字段

```

```

//public string Name    //属性
//{
//    //get => _name;
//    //set => _name = value;
//}
public string Name    //成员属性(即对外公开的成员变量), the Name property
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}
}

```

注意！ 在 `set` 访问器中不能再声明或者定义一个名为 `value` 的局部变量，因为隐式参数名称已经使用了 `value` 这一标识名称。

3) init 访问器

用于创建 `init` 访问器的代码与用于创建 `set` 访问器的代码相同，只不过前者使用的关键字是 `init` 而不是 `set`。不同之处在于，`init` 访问器只能在构造函数中使用，或通过对象初始值设定项使用。

4. 属性访问修饰符

可以将属性标记为 `public`、`private`、`protected`、`internal`、`protected internal` 或 `private protected`。这些访问修饰符定义该类的用户访问该属性的方式。相同属性的 `get` 和 `set` 访问器可以具有不同的访问修饰符。例如，`get` 可能为 `public` 允许从类型外部进行只读访问；而 `set` 可能为 `private` 或 `protected`。

可以通过使用 `static` 关键字将属性声明为静态属性。静态属性可供调用方在任何时候使用，即使不存在类的任何实例。

可以通过使用 `virtual` 关键字将属性标记为虚拟属性。虚拟属性可使派生类使用 `override` 关键字重写属性行为。

重写虚拟属性的属性也可以是 `sealed`，指定对于派生类，它不再是虚拟的。最后，可以将属性声明为 `abstract`。抽象属性不定义类中的实现，派生类必须写入自己的实现。

注意！ 在 `static` 属性的访问器上使用 `virtual`、`abstract` 或 `override` 修饰符是错误的。

下面的示例演示实例、静态和只读属性。它接收通过键盘键入的员工姓名，按 1 递增 `NumberOfEmployees`，并显示员工姓名和编号。

PS:

```

public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;
}

```

```

// A read-write instance property:
public string Name
{
    get => _name;
    set => _name = value;
}

// A read-only static property:
public static int Counter => _counter;

// A Constructor:
public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's
number:
}

```

5. Hidden 属性

下面的示例演示如何访问基类中被派生类中具有同一名称的另一个属性隐藏的属性。

PS:

```

// property_hiding.cs
// Property hiding
using System;
public class BaseClass
{
    private string name;
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value; //value 为隐形参数
        }
    }
}

public class DerivedClass : BaseClass
{
    private string name;
    public new string Name // 派生类中定义一个与基类中同名的属性，注意使用了
new 修饰符
    {
        get
        {

```

```

        return name;
    }
    set
    {
        name = value;
    }
}

}

public class MainClass
{
    public static void Main()
    {
        DerivedClass d1 = new DerivedClass();
        d1.Name = "John"; // Derived class property
        Console.WriteLine("Name in the derived class is: {0}", d1.Name);
        ((BaseClass)d1).Name = "Mary"; // Base class property
        Console.WriteLine("Name in the base class is: {0}", ((BaseClass)d1).Name);
    }
}

```

以下是上例中显示的重点！

使用派生类中的属性 **Name** 去隐藏基类中的属性 **Name**。在这种情况下，派生类的该属性声明使用 **new** 修饰符。

PS:

```
public new string Name
(Employee) 转换用于访问基类中的隐藏属性。
```

PS:

```
((Employee)m1).Name = "Mary";
```

6. 抽象属性- Abstract Properties

抽象类可拥有抽象属性，这些属性应在派生类中被实现。

PS:

```
using System;
namespace runoob
{
    public abstract class Person
    {
        public abstract string Name
        {
            get;
            set;
        }
        public abstract int Age
        {
            get;

```

```
        set;
    }
}
class Student : Person
{

    private string code = "N.A";
    private string name = "N.A";
    private int age = 0;

    // 声明类型为 string 的 Code 属性
    public string Code
    {
        get
        {
            return code;
        }
        set
        {
            code = value;
        }
    }

    // 声明类型为 string 的 Name 属性
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    // 声明类型为 int 的 Age 属性
    public override int Age
    {
        get
        {
            return age;
        }
        set
```



```
        {
            age = value;
        }
    }
    public override string ToString()
    {
        return "Code = " + Code + ", Name = " + Name + ", Age = " + Age;
    }
}
class ExampleDemo
{
    public static void Main()
    {
        // 创建一个新的 Student 对象
        Student s = new Student();

        // 设置 student 的 code、name 和 age
        s.Code = "001";
        s.Name = "Zara";
        s.Age = 9;
        Console.WriteLine("Student Info:- {0}", s);
        // 增加年龄
        s.Age += 1;
        Console.WriteLine("Student Info:- {0}", s);
        Console.ReadKey();
    }
}
}
```

7. 泛型属性

我们可以用泛型+接口的属性类型。

PS:

```
public interface INameValuePair<T>
{
    T Name
    {
        get;
    }
    T Value
    {
        get;
        set;
    }
}
```

8. 属性可支持多线程

因为属性是用方法实现的，所以它拥有方法所拥有的一切语言特性。

比如，为属性增加多线程的支持是非常方便的。我们可以加强 `get` 和 `set` 访问器（accessors）的实现来提供数据访问的同步：

PS:

```
public class Customer
{
    private object syncHandle = new object();
    private string name;
    //使用属性进行封装
    public string Name
    {
        get
        {
            lock (syncHandle)
                return name;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                    "Name");
            lock (syncHandle)
                name = value;
        }
    }
    // More Elided.
}
```

二十四. C#索引器—Indexer

1. 概述

索引器允许类或结构的实例就像数组一样进行索引，无需显式指定类型或实例成员，即可设置或检索索引值。

索引器的行为的声明在某种程度上类似于属性（property）。就像属性（property），我们可以使用 `get` 和 `set` 访问器来定义索引器。但是，属性返回或设置一个特定的数据成员，而索引器返回或设置对象实例的一个特定值。换句话说，它把实例数据分为更小的部分，并索引每个部分，获取或设置每个部分。

定义一个属性（property）包括提供属性名称。索引器定义的时候不带有名称，但带有 `this` 关键字，它指向对象实例。索引器不必根据整数值进行索引；由我们决定如何定义特定的查找机制。

以下示例定义了一个泛型类，其中包含用于赋值和检索值的简单 `get` 和 `set` 访问器方法。Program 类创建了此类的一个实例，用于存储字符串。

PS:

```
using System;
```

```
class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer to allow client code to use [] notation.
    public T this[int i]
    {
        get { return arr[i]; }
        set { arr[i] = value; }
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection[0] = "Hello, World";
        Console.WriteLine(stringCollection[0]);
    }
}

// The example displays the following output:
//      Hello, World.
```

2. 一维索引器

PS:

```
element-type this[int index]
{
    // get 访问器
    get
    {
        // 返回 index 指定的值
    }

    // set 访问器
    set
    {
        // 设置 index 指定的值
    }
}
```

对于多维的索引器，我们应该尽可能的避免使用，索引器（即索引属性）应该使用一个索引。**多维索引器会大大降低库的可用性。** 如果设计需要多个索引，请重新考虑该类型是否表示逻辑数据存储。 否则，请使用方法。

3. 重载索引器

索引器（**Indexer**）可被重载。索引器声明的时候也可带有多个参数，且每个参数可以是不同的类型。没有必要让索引器必须是整型的，C# 允许索引器可以是其他类型，例如，字符串类型。

下面的实例演示了重载索引器。

PS:

```
using System;
namespace IndexerApplication
{
    class IndexedNames
    {
        private string[] namelist = new string[size];
        static public int size = 10;
        public IndexedNames()
        {
            for (int i = 0; i < size; i++)
            {
                namelist[i] = "N. A.";
            }
        }
        public string this[int index]
        {
            get
            {
                string tmp;

                if( index >= 0 && index <= size-1 )
                {
                    tmp = namelist[index];
                }
                else
                {
                    tmp = "";
                }

                return ( tmp );
            }
            set
            {
                if( index >= 0 && index <= size-1 )
                {
                    namelist[index] = value;
                }
            }
        }
    }
}
```

```

    }
    public int this[string name]
    {
        get
        {
            int index = 0;
            while(index < size)
            {
                if (namelist[index] == name)
                {
                    return index;
                }
                index++;
            }
            return index;
        }
    }

}

static void Main(string[] args)
{
    IndexedNames names = new IndexedNames();
    names[0] = "Zara";
    names[1] = "Riz";
    names[2] = "Nuha";
    names[3] = "Asif";
    names[4] = "Davinder";
    names[5] = "Sunil";
    names[6] = "Rubic";
    // 使用带有 int 参数的第一个索引器
    for (int i = 0; i < IndexedNames.size; i++)
    {
        Console.WriteLine(names[i]);
    }
    // 使用带有 string 参数的第二个索引器
    Console.WriteLine(names["Nuha"]);
    Console.ReadKey();
}
}

```

4. 创建多维索引器

为了和多维数组保持一致，我们可以创建多维索引器，在不同的维度上使用相同或不同类型：

PS:

```
class ComputeValueClass
{
    public int this[int x, int y]
    {
        get { return ComputeValue(x, y); }
    }

    public int this[int x, string name]
    {
        get { return ComputeValue(x, name); }
    }

    private int ComputeValue(int x, int y)
    {
        //...
    }

    private int ComputeValue(int x, string y)
    {
        //...
    }
}
```

值得注意的是**所有索引器必须使用 `this` 关键字声明**。在 C# 中我们不能自己命名索引器。所以一个类型的索引器必须有不同的参数列表来避免歧义。几乎所有的属性的功能都适用索引器。索引器可以是 `virtual` 或 `abstract`；索引器的 `setters` 和 `getters` 可以是不同的访问限制。不过，我们不能像创建隐式属性一样创建隐式索引器。

5. 属性和索引器之间的比较

索引器与属性相似。除下表所示的差别外，对属性访问器定义的所有规则也适用于索引器访问器。

Property	索引器
允许 以将方法视作公共数据成员的方式调用方法。	通过在对象自身上使用数组表示法， 允许访问对象内部集合的元素 。
通过简单名称访问。	通过索引访问。
可为静态成员或实例成员。	必须是实例成员 。
属性的 <code>get</code> 访问器没有任何参数。	索引器的 <code>get</code> 访问器具有与索引器相同的 形参列表 。
属性的 <code>set</code> 访问器 包含隐式 <code>value</code> 参数 。	索引器的 <code>set</code> 访问器具有与索引器相同的形参列表， <code>value</code> 参数也是如此。
通过自动实现的属性支持简短语法。	支持仅使用索引器的 <code>expression-bodied</code> 成员。

二十五. C#特性-Attribute

1. 特性定义

使用特性，可以有效地将元数据或声明性信息与代码（程序集、类型、方法、属性等）相关联。将特性与程序实体相关联后，可以在运行时使用反射这项技术查询特性。

特性，是用来给代码添加额外信息的一种手段，我们通常是将特性标记到方法，类或者属性上，在使用这些结构的时候，通过反射(reflection)这一非常高级的技术，获取它们通过特性标记的信息，从而进行某些特殊的处理。

.Net 框架提供了两种类型的特性，即预定义特性和自定义特性。比如 Serializable 标记一个可序列化的类，DebuggerStepThrough 设置方法在调试时为跳过的状态。

所有特性名称均以“Attribute”一词结尾，以便与 .NET 库中的其他项区分开来。不过，在代码中使用特性时，无需指定特性后缀。例如，[DllImport] 等同于 [DllImportAttribute]，但 DllImportAttribute 是此特性在 .NET 类库中的实际名称。

2. 特性与属性 (Attribut And Property)

我们使用静态的 Property 或者 Field 一样可以做到在不实例化的时候拿到一些信息，如果这样的话，Attribute 又有什么存在意义呢？

1) Property

Property 可以说是一个面向对象的概念，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。

PS:

```
public class Robot
{
    private string name = "";    //字段: Field
    public string Name          //属性: Property, 对 Field 进行封装。
    {
        get { return name; }
        set { name = value; }
    }
}
```

2)Attribut

Attribute 的目标是：为元素提供附加信息，它的作用更类似于注释。

可以说，Property 或 Field 和 Attribute 是两个完全不同的概念，虽然他们有些时候能做一些一样的事，但请记住，他们是从本质上就不同的两个东西。

3. 特性具有的属性

特性是向程序添加元数据（元数据是程序中定义的类型的相关信息）。

所有.NET 程序集都包含一组指定的元数据，用于描述程序集中定义的类型和类型成员。我们可以添加自定义特性来指定所需的其他任何信息。

可以将一个或多个特性应用于整个程序集、模块或较小的程序元素（如类和属性）。

特性可以像方法和属性一样接受自变量。

程序可使用反射来检查自己的元数据或其他程序中的元数据。

4. 使用特性

我们可以将特性附加到几乎任何的声明当中，尽管特定特性可能会限制可有效附加到的声明的类型。

特性的使用很简单，用方括号 ([]) 将特性名称括起来，并置于应用该特性的实体的声明上方以指定特性。

下方示例中，SerializableAttribute 特性用于将具体特征应用于类 SampleClass。

PS:

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

下方示例声明了一个具有特性 DllImportAttribute 的方法。

PS:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

如下方示例所示，我们可以将多个特性附加到声明中。

PS:

```
using System.Runtime.InteropServices;
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

对于给定实体，一些特性可以指定多次。ConditionalAttribute 便属于此类多用途特性。大多数特性只针对直接跟随在一个或多个特性片段后的结构。

PS:

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

单个结构也可以运用多个特性，使用时可以把独立的特性片段互相叠在一起或使用分成单个特性片段，特性之间用逗号分隔。

PS:

```
[Serializable]
[MyAttribute("firt","second","finally")]    //独立的特性片段
```

PS:

```
[MyAttribute("firt","second","finally"), Serializable]    //逗号分隔
```

5. 特性参数

许多属性都有参数，可以是位置参数、未命名参数或已命名参数。首先指定的是位置参数，我们必须以特定顺序指定任何位置参数，且不能省略。已命名参数是可选参数，可以通过任何顺序指定。例如，下面这三个特性是等同的。

PS:

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
```

```
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

第一个参数（DLL 名称）是位置参数，始终出现在第一个位置；其他是已命名参数。在此示例中，两个已命名参数的默认值均为 false，因此可以省略。[位置参数](#)与[特性构造函数](#)的参数相对应，[已命名](#)或可选参数与该特性的[属性](#)或[字段](#)相对应。

6. 特性目标

特性目标是指[应用特性的实体](#)。（特性可应用于类、特定方法或整个程序集）默认情况下，特性应用于紧跟在它后面的元素。不过，还可以[进行显式标识](#)。例如，可以标识为将特性应用于方法，还是应用于其参数或返回值。

PS:

```
[target : attribute-list]
```

target 目标值	适用对象
assembly	整个程序集
module	当前程序集模块
field	类或结构中的字段
event	事件
method	方法或 get 和 set 属性访问器
param	方法参数或 set 属性访问器参数
property	Property
return	方法、属性索引器或 get 属性访问器的返回值
type	结构、类、接口、枚举或委托

下面的示例展示了如何将特性应用于程序集和模块。

PS:

```
using System;
using System.Reflection;
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

以下示例演示如何将特性应用于 C# 中的方法、方法参数和方法返回值。

PS:

```
// 默认情况下是应用于方法上
[ValidatedContract]
int Method1() { return 0; }
```

PS:

```
// 特性应用于方法
[method: ValidatedContract]
int Method2() { return 0; }
```

PS:

```
// 特性应用于方法参数
int Method3([ValidatedContract] string contract) { return 0; }
```

PS:

```
// 特性应用于方法返回值
[return: ValidatedContract]
int Method4() { return 0; }
```

无论在哪个目标上将 `ValidatedContract` 定义为有效，都必须指定 `return` 目标，即使 `ValidatedContract` 定义为仅应用于返回值也是如此。换言之，编译器不会使用 `AttributeUsage` 信息来解析不明确特性目标。

7. 创建自定义特性

可通过定义特性类去创建我们自己的自定义特性，特性类是直接或间接派生自 `Attribute` 的类，可快速轻松地识别元数据中的特性定义。假设希望使用编写类型的程序员的姓名来标记该类型，可能就需要定义一个自定义的 `Author` 特性类。

PS:

```
[System.AttributeUsage(System.AttributeTargets.Class |
                        System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string name;
    public double version;

    public AuthorAttribute(string name)
    {
        this.name = name;
        version = 1.0;
    }
}
```

类名 `AuthorAttribute` 是该特性的名称，即 `Author` 加上 `Attribute` 后缀，使用的时候这个后缀可以省略。由于该类派生自 `System.Attribute`，因此它是一个自定义特性类。构造函数的参数是自定义特性的位置参数。在此示例中，`name` 是位置参数。所有公共读写字段或属性都是命名参数。在本例中，`version` 是唯一的命名参数。请注意，使用 `AttributeUsage` 特性可使 `Author` 特性仅对类和 `struct` 声明有效。

PS:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

8. AttributeUsage 特性

直接在一个元素上添加多个相同的特性，会报错提示特性重复。`AttributeUsage` 有一个命名参数 `AllowMultiple`，通过此命名参数可一次或多次使用自定义特性。我们在特性上面添加特性标记 `[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]`，这样就可以给同一个元素添加多个相同的特性了。

`AttributeUsage` 特性会影响编译器运行，指定修饰的对象、能否重复修饰、修饰的特性子类是否生效，建议是明确约束用在哪些对象上。

PS:

```
using System;
namespace MyAttribute
```

```

{
    [AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
    public class CustomAttribute : Attribute
    {
        public CustomAttribute()
        {
            Console.WriteLine($"{this.GetType().Name} 无参数构造函数执行");
        }
        public CustomAttribute(int id)
        {
            Console.WriteLine($"{this.GetType().Name} int 参数构造函数执行");
            this._Id = id;
        }
        public CustomAttribute(string name)
        {
            Console.WriteLine($"{this.GetType().Name} string 参数构造函数执行");
            this._Name = name;
        }
    }
}

```

PS:

```

using System;
namespace MyAttribute
{
    [Custom]
    [Custom()]
    [Custom(0)]
    public class Student
    {
        [Custom]
        public int Id { get; set; }
        public string Name { get; set; }
        [Custom]
        public void Study()
        {
            Console.WriteLine($"这里是{this.Name} ");
        }

        [Custom(0)]
        public string Answer([Custom]string name)
        {
            return $"This is {name}";
        }
    }
}

```

```
}
```

9. 使用反射访问特性

我们可以定义自定义特性并将其放入源代码中,但是在没有检索该信息并对其进行操作的方法的情况下将没有任何价值。通过使用反射,可以检索通过自定义特性定义的信息。主要方法是 `GetCustomAttributes`,它返回对象数组,这些对象在运行时等效于源代码特性。

PS:

```
[Author("P. Ackerman", version = 1.1)]
```

```
class SampleClass
```

上面的代码,在概念上等效于下面的。

PS:

```
Author anonymousAuthorObject = new Author("P. Ackerman");
```

```
anonymousAuthorObject.version = 1.1;
```

但是,在用特性查询 `SampleClass` 之前,代码将不会执行。对 `SampleClass` 调用 `GetCustomAttributes` 会导致按上述方式构造并初始化一个 `Author` 对象。如果该类具有其他特性,则将以类似方式构造其他特性对象。然后 `GetCustomAttributes` 会以数组形式返回 `Author` 对象和任何其他特性对象,之后我们可以循环访问此数组,根据每个数组元素的类型确定所应用的特性,并从特性对象中提取信息。

PS:

```
// Multiuse attribute.
```

```
[System.AttributeUsage(System.AttributeTargets.Class |
                        System.AttributeTargets.Struct,
                        AllowMultiple = true) // Multiuse attribute.
```

```
]
```

```
public class Author : System.Attribute
```

```
{
```

```
    string name;
```

```
    public double version;
```

```
    public Author(string name)
```

```
    {
```

```
        this.name = name;
```

```
        //默认值为 1.0
```

```
        version = 1.0;
```

```
    }
```

```
    public string GetName()
```

```
    {
```

```
        return name;
```

```
    }
```

```
}
```

```
//对类使用 Author 特性
```

```
[Author("P. Ackerman")]
```

```
public class FirstClass
{
    // ...
}

//这个类没有使用 Author 特性
public class SecondClass
{
    // ...
}

//这个类应用了多个特性
[Author("P. Ackerman"), Author("R. Koch", version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine("Author information for {0}", t);

        //使用反射
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t); //
Reflection.

        //显示输出内容
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("    {0}, version {1:f}", a.GetName(),
a.version);
            }
        }
    }
}
```

```

    }
}
}
/* Output:
    Author information for FirstClass
        P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
        R. Koch, version 2.00
        P. Ackerman, version 1.00
*/

```

二十六. C#反射-Reflection

反射机制是在**运行状态**中，对于任意一个类，都能够知道这个类的所有属性和方法，对于任意一个对象，都能够调用它的任意一个方法和属性，是一种**动态获取的信息以及动态调用对象的方法的功能**。

反射提供描述程序集、模块和类型的对象（Type 类型）。我们也可以使用反射来动态地创建类型的实例，然后将类型绑定到现有对象，或**从现有对象中获取类型**，然后**调用其方法**或访问器字段和属性。如果在代码中使用了特性，可以也可以利用反射来访问它们。

1. 优缺点

1) 优点

- ① 反射提高了程序的灵活性和扩展性。
- ② 降低耦合性，提高自适应能力。
- ③ 它允许程序创建和控制任何类的对象，无需提前硬编码目标类。

2) 缺点

- ① 使用反射会带来一定的性能问题，因为反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码。因此**反射机制主要应用在对灵活性和拓展性要求很高的系统框架上**，普通程序不建议使用。
- ② 使用反射会模糊程序内部逻辑；程序员希望在源代码中看到程序的逻辑，反射却绕过了源代码的技术，因而会**带来维护的问题**，反射代码比相应的直接代码更复杂。

2. 通过反射获取类型

1) 通过 typeof 获取

PS:

```

using UnityEngine;
using System;

public class Test : MonoBehaviour
{
    public int i = 42;
    public void Start()
    {
        Type type = typeof(Test);
        Debug.LogError(type);
    }
}

```



```
}
```

2)通过 GetType 获取

在一个变量上调用 `GetType()` 方法，而不是把类型的名称作为其参数。但要注意，返回的 `Type` 对象仍只与该数据类型相关，它不包含与该类型的实例相关的任何信息。如果引用了一个对象，但不能确保该对象实际上是哪个类的实例，这个方法就很有用。

PS:

```
using UnityEngine;
using System;
```

```
public class Test : MonoBehaviour
{
    public int i = 42;
    public void Start()
    {
        Type type = i.GetType();
        //输出结果为 System.Int32
        Debug.LogError(type);
    }
}
```

同时 `System.Type` 也有一个静态方法是 `GetType`，使用这个静态方法我们就可以直接使用字符串去获取类型。

PS:

```
Type type = Type.GetType("Test");
```

3. System.Type 类

前面我们已经使用 `typeof` 和 `GetType` 获取到了类型，而存储这个类型的就是 `Type` 类。

使用 `Type` 类只为了存储类型的引用，虽然我们把 `Type` 看作是一个类，但它实际上是一个抽象的基类。只要实例化了一个 `Type` 对象，实际上就实例化了 `Type` 的一个派生类。

`Type` 是许多反射功能的入口。注意，可用的属性都是只读的，我们可以使用 `Type` 确定数据的类型，但不能使用它修改数据的类型！

程序中用到的每一个类型，CLR 都会创建一个包含这个类型信息的 `Type` 类型的对象，程序中用到的每一个类型，都会关联到独立的 `Type` 类型的对象，不管创建的类型有多少个实例，只有一个 `Type` 对象会关联到所有这些实例。

下面的图像显示了一个运行的程序，它有两个 `MyClass` 对象和一个 `OtherClass` 对象。尽管有两个 `MyClass` 的实例，但是只会有一个 `Type` 对象来表示它。

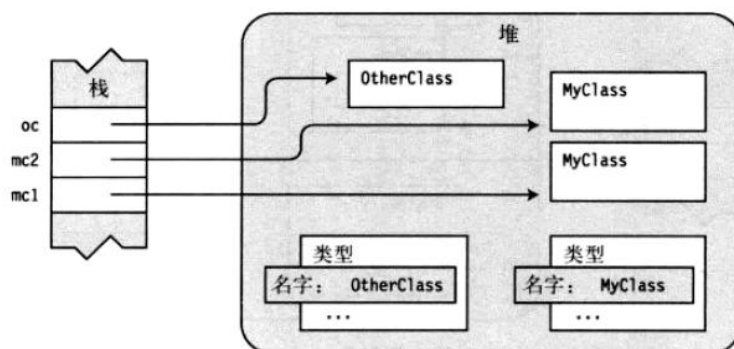


图 1. 对于程序中使用的每一个类型，CLR 都会实例化 `Type` 类型的对象

1) Type 的属性

属性	返回值
Name	数据类型名
FullName	数据类型的完全限定名(包括名称空间名)
Namespace	在其中定义数据类型的名称空间名

2) 方法

System.Type 的大多数方法都用于获取对应数据类型的成员信息，比如构造函数、属性、方法和事件等。它有许多方法，但它们都有相同的模式。例如，有两个方法可以获取数据类型的方法的细节信息：GetMethod()和 GetMethods()。GetMethod()方法返回 System.Reflection.MethodInfo 对象的一个引用，其中包含一个方法的细节信息。

GetMethods()返回这种引用的一个数组。其区别是 GetMethods()方法返回所有方法的细节信息；而 GetMethod()方法返回一个方法的细节信息，其中该方法包含特定的参数列表。这两个方法都有重载方法，重载方法有一个附加的参数，即 BindingFlags 枚举值，该值表示应返回哪些成员，例如，返回公有成员、实例成员和静态成员等。

例如，GetMethods()最简单的一个重载方法是不带参数的，其返回数据类型所有的公共方法信息。

PS:

```
using UnityEngine;
using System;
using System.Reflection;
```

```
public class Test : MonoBehaviour
{
    public void Start()
    {
        Type type = typeof(int);
        MethodInfo[] methods = type.GetMethods();
        //遍历 int 中的所有方法
        foreach (MethodInfo nextMethod in methods)
        {
            Debug.LogError(nextMethod);
        }
    }
}
```

方法名（复数形式表示返回数组）	功能描述	返回的对象类型
GetConstructor(string)	获取当前 Type 的特定构造函数。	ConstructorInfo
GetConstructors()	获取当前 Type 的构造函数，不加约束就是返回为当前 Type 定义的所有公共构造函数。	
GetEvent(string)	获取由当前 Type 声明或继	EventInfo

	承的 特定事件 。	
GetEvents()	获取由当前 Type 声明或继承的事件， 不加约束 就是返回由当前 Type 声明或继承的 所有公共事件 。	
GetField(string)	获取当前 Type 的 特定字段 。	FieldInfo
GetFields()	获取当前 Type 的字段， 不加约束 就是获取 所有公共字段 。	
GetMember(string)	获取当前 Type 的 指定成员 。	MemberInfo
GetMembers()	获取当前 Type 的成员， 不加约束 就是返回为当前 Type 的 所有公共成员 。	
GetDefaultMembers()	搜索当前 Type 设置了的 DefaultMemberAttribute 定义的成员。如果当前 MemberInfo 没有默认成员，则返回 Type 类型的空数组。	
GetMethod(string)	获取当前 Type 的 特定方法 。	MethodInfo
GetMethods()	获取当前 Type 的方法， 不加约束 就是返回当前 Type 的 所有公共方法 。	
GetProperty(string)	获取当前 Type 的 特定属性 。	PropertyInfo
GetProperties()	获取当前 Type 的属性， 不加约束 就是返回当前 Type 的 所有公共属性 。	

GetMember()和 GetMembers()方法返回数据类型的任何成员或所有成员的详细信息，不管这些成员是构造函数、属性和方法等。

4. 实际运用

我们首先定义一个用于反射的数据类，在这个数据类中我们定义了多个构造函数，以及一些公共方法。在后面的应用中，我们将使用反射来获取该类中的信息。

PS:

```
using UnityEngine;
```

```
public class ReflectionTest
{
    /// <summary>
    /// 排名
    /// </summary>
    public int ranking;
    /// <summary>
    /// 名字
    /// </summary>
    private string name;
    /// <summary>
```

```
/// 长度
/// </summary>
private string length;
/// <summary>
/// 形状
/// </summary>
private string shape;

#region 构造函数
public ReflectionTest(string name, string length)
{
    this.name = name;
    this.length = length;
}
public ReflectionTest(string name, string shape, string length)
{
    this.name = name;
    this.shape = shape;
    this.length = length;
}

public ReflectionTest(string shape)
{
    this.shape = shape;
}

public ReflectionTest()
{
}

#endregion

#region 属性
/// <summary>
/// 名字
/// </summary>
public string Name
{
    get => name;
    set => name = value;
}
```

```

    /// <summary>
    /// 长度
    /// </summary>
    public string Length
    {
        get => length;
        set => length = value;
    }

    /// <summary>
    /// 形状
    /// </summary>
    public string Shape
    {
        get => shape;
        set => shape = value;
    }

    #endregion

    /// <summary>
    /// 打印信息
    /// </summary>
    public void ShowContent()
    {
        Debug.Log($"姓名是: {name}, 它的形状为: {shape}, 形状的周长为: {length}");
    }

    /// <summary>
    /// 打印排名
    /// </summary>
    public void ShowRanking()
    {
        Debug.Log($"排名是: {ranking}");
    }

    /// <summary>
    /// 要开始跑动了!!
    /// </summary>
    public void StartMove()
    {
        Debug.Log($"{name}准备开始跑动了!! ");
    }
}

```

1) 获取类中的构造函数及其参数

在前面我们已经构造了一个用于反射测试的类, 接下来我们就将编写一个实际的测试脚

本。在这个脚本中，我们将使用[字符串内插](#)的方式打印构造函数的参数名称和类型。

由于前面我们已经说过，复数名称的反射方法是返回的一个数组，我们下面的代码中都是使用的复数形式。获取所有公共构造函数使用的是 [GetConstructors](#)，获取构造函数中的所有参数使用的是 [GetParameters](#)。

PS:

```
using UnityEngine;
using System;
using System.Reflection;
```

```
public class Test : MonoBehaviour
{
```

```
    private ReflectionTest reflectionTest = null;
```

```
    private void Awake()
```

```
    {
        //实例化
        reflectionTest = new ReflectionTest();
    }
```

```
    private void Start()
```

```
    {
        SeeConstructor();
    }
```

```
    /// <summary>
```

```
    /// 获取构造函数的所有参数及参数对应的数据类型
```

```
    /// </summary>
```

```
    private void SeeConstructor()
```

```
    {
        //获取 reflectionTest 的 Type
        Type type = reflectionTest.GetType();
        //通过 Type 获取这个类的所有构造函数信息
        ConstructorInfo[] constructorArray = type.GetConstructors();
        foreach (ConstructorInfo constructorInfo in constructorArray)
        {
            //获取获取每个构造函数所包含的所有参数
            ParameterInfo[] parameterArray = constructorInfo.GetParameters();
            foreach (ParameterInfo parameterInfo in parameterArray)
            {
```

```
                //打印的构造函数顺序是从上到下的，哪个构造函数在前面就先打印
```

谁的参数

```
                Debug.LogError($"参数的数据类型为:
```

```
<color=yellow>{parameterInfo.ParameterType}</color>\n 参数名字:
```

```
<color=green>{parameterInfo.Name}</color>");
```

```

    }
}
}
}

```



图 2. 打印出的构造函数中的参数类型和名称

2) 获取类型中的属性

同样的，除了可以运用反射获取构造函数及其参数，我们还可以使用反射获取到对应类型中的属性。**注意！属性是对字段的封装。**

PS:

```

#region SeeProperty
/// <summary>
/// 通过反射获取类型中的属性
/// </summary>
private void SeeProperty()
{
    //获取 reflectionTest 的 Type
    Type type = reflectionTest.GetType();
    //获取所有属性信息
    PropertyInfo[] propertyInfos = type.GetProperties();
    foreach (PropertyInfo propertyInfo in propertyInfos)
    {
        //打印出来属性的名字
        Debug.LogError($"属性名称:<color=green>{propertyInfo.Name}</color>");
    }
}
#endregion

```



图 3. 打印出的公共属性

3) 获取类型中的方法

在通过反射获取类型中的方法时，我们不光可以获取到自己编写的方法，还会获取到那些被隐藏起来的方法。

PS:

```
#region SeePublicMethod
/// <summary>
/// 查看类中的公共方法
/// </summary>
private void SeePublicMethod()
{
    Type type = reflectionTest.GetType();
    //通过 type 获取所有公共方法的信息
    MethodInfo[] methodInfos = type.GetMethods();
    foreach (MethodInfo methodInfo in methodInfos)
    {
        Debug.LogError($"公共方法的返回类型为:
<color=yellow>{methodInfo.ReturnType}</color>\n 方法的名称:
<color=green>{methodInfo.Name}</color>");
    }
}
#endregion
```



图 4. 不仅能获取到自己定义的，还可以获取到那些隐含的方法

4) 获取类型中的的字段

我们在 ReflectionTest 数据类中定义了四个字段，但是只有 ranking 字段是公共类型的访问修饰符。如果我们使用未加任何约束的 GetFields 方法，我们就是获取所有的公共类型字段，也即 ranking 字段。

PS:

```
#region SeePublicField
/// <summary>
/// 查看类型中的公共字段
/// </summary>
private void SeePublicField()
{
    Type type = reflectionTest.GetType();
    //通过 type 获取所有公共字段的信息，默认的不加约束的 GetFields 获取的就是公共字段
    FieldInfo[] fieldInfos = type.GetFields();
```

```

        foreach (FieldInfo fieldInfo in fieldInfos)
        {
            Debug.LogError($"公开的字段的名称:
<color=orange>{fieldInfo.Name}</color>");
        }
    }
#endregion

```



图 5. 未加约束的 GetFields，获取到的所有公共字段

5) 通过构造函数动态生成对象

我们可以通过 `GetConstructor` 获取到对应参数类型的构造函数，然后通过 `Invoke` 去生成对应的对象。

PS:

```

#region CreateObjectByConstruct
/// <summary>
/// 通过反射用构造函数动态生成对象
/// </summary>
private void CreateObjectByConstruct()
{
    Type type = reflectionTest.GetType();
    Type[] typeArray = { typeof(string), typeof(string), typeof(string) };
    //根据构造函数参数类型来获取构造函数
    ConstructorInfo constructorInfo = type.GetConstructor(typeArray);
    //要传入的参数
    object[] objectArray = { "黎明", "飘忽不定", "8cm" };
    //调用构造函数来生成对象
    object obj = constructorInfo.Invoke(objectArray);
    //调用打印方法，看是否有输出
    ((ReflectionTest)obj).ShowContent();
}
#endregion

```



图 6. 调用的对象中的方法

6) 用 Activator 静态生成对象

当我们需要动态的创建一个实例的时候，就可以使用 `Activator.CreateInstance(Type type)`；如果是明确的知道要创建哪个类型的实例，就可以直接用 `new` 关键字。其中，`Activator.CreateInstance` 默认调用类型的无参构造函数。

PS:

```

#region CreatObjByActivator
/// <summary>

```

```

/// 通过反射用 Activator 静态生成对象
/// </summary>
private void CreatObjByActivator()
{
    Type type = reflectionTest.GetType();
    //要传入的参数
    object[] objectArray = { "李明", "石头", "500cm" };
    //通过 Activator.CreateInstance 实例化对象
    object obj = Activator.CreateInstance(type, objectArray);
    //调用打印方法，看是否有输出
    ((ReflectionTest)obj).ShowContent();
}
#endregion

```



图 7. 对应参数的构造函数所生成对象中的方法

7) 创建对象并完整的使用对象

PS:

```

#region CreatAndUseObject
/// <summary>
/// 通过反射，创建对象，给字段，属性赋值，调用方法
/// </summary>
private void CreatAndUseObject()
{
    //获取 reflectionTest 的类型
    Type type = reflectionTest.GetType();
    //创建实例化对象
    object obj = Activator.CreateInstance(type);
    //通过名字获取字段，未加约束就是获取公共类型字段
    FieldInfo fieldInfo = type.GetField("ranking");
    /*//如果 ranking 是私有字段，我们就可以使用下面的约束来获取它
    FieldInfo rankInfo = type.GetField("ranking", BindingFlags.NonPublic |
BindingFlags.Instance);
    rankInfo.SetValue(obj, 1);
    Debug.LogError($"字段名称: {rankInfo.Name}.....{rankInfo}");*/
    //给字段赋值
    fieldInfo.SetValue(obj, 1);
    //获取所有属性的信息
    PropertyInfo[] propertyInfos = type.GetProperties();
    string[] strings = { "阿姆罗", "敢达", "150cm" };
    //依次给属性属性赋值
    for (int i = 0; i < propertyInfos.Length; i++)
    {
        Debug.LogError($"属性名称是

```

```

<color=orange>{propertyInfos[i].Name}</color>");
    propertyInfos[i].SetValue(obj, strings[i], null);
}
//根据名字找到方法的信息
MethodInfo methodInfo = type.GetMethod("StartMove");
//执行方法
methodInfo.Invoke(obj, null);
//根据名字找到方法的信息
MethodInfo methodInfo2 = type.GetMethod("ShowRanking");
//执行方法
methodInfo2.Invoke(obj, null);
}
#endregion

```



打印输出以及执行的方法体内容

二十七. C#的反射为什么慢以及怎么加快反射调用

1. 反射的设计初衷

- ① 在运行时非常快的访问我们所需要的代码的信息。
- ② 在编译时非常直接的访问生成代码所需的信息。
- ③ 垃圾回收器/计算堆栈能够在不对程序加锁/分配内存的情况下访问必要的信息。
- ④ 能极大减少一次性需要加载的类型数量。
- ⑤ 能极大减少给定类型加载时所加载的额外类型数目。
- ⑥ 类型系统数据结构必须在 NGEN 映像中是可存储的。

我们可以看到，上面只强调了最少依赖加载，并没有说我们可以直接从元数据获取所有 CLR 数据类型。也没有说所有的反射用法都是快的，只是说反射获取一些信息很快。MethodTable 的数据被分为“热”和“冷”两种数据结构来提升工作效率和缓存利用率，MethodTable 本身只存储那些在程序稳定状态下被需求的“热”数据。EEClass 储存那些只在类型加载时，JIT 编译时，反射时需要的“冷”数据。

2. 获取方法信息

在我们使用反射调用一个字段/属性/方法之前，我们不得不获取 FieldInfo/PropertyInfo/MethodInfo 来处理，就像下面这样。

PS:

```

Type t = typeof(Person);
FieldInfo m = t.GetField("Name");

```

使用反射获取信息是有代价的，因为相关的元数据必须被执行获取，解析等操作。非常

有趣的是，**运行时通过保留所有字段/属性/方法的内部缓存来帮助我们减少消耗**。这个缓存是由 `RuntimeTypeCache` 类实现的，其用法的一个例子是 `RuntimeMethodInfo` 类。通过运行上面的代码，我们可以看到缓存的运行情况，这些信息足够使用反射来检查运行时的内部信息！

在我们进行任何反射以获得 `FieldInfo` 之前，上面的代码将打印以下内容。

PS:

```
Type: ReflectionOverhead.Program
Reflection Type: System.RuntimeType (BaseType: System.Reflection.TypeInfo)
m_fieldInfoCache is null, cache has not been initialised yet
```

但是一旦我们获取过了哪怕一个字段，下面的内容就会被打印出来。

PS:

```
Type: ReflectionOverhead.Program
Reflection Type: System.RuntimeType (BaseType: System.Reflection.TypeInfo)
RuntimeTypeCache: System.RuntimeType+RuntimeTypeCache,
m_cacheComplete = True, 4 items in cache
[0] - Int32 TestField1 - Private
[1] - System.String TestField2 - Private
[2] - Int32 <TestProperty1>k__BackingField - Private
[3] - System.String TestField3 - Private, Static
```

PS:

```
class Program
{
    private int TestField1;
    private string TestField2;
    private static string TestField3;

    private int TestProperty1 { get; set; }
}
```

这意味着对 **GetField** 或 **GetFields** 的重复调用会比第一次调用消耗小很多，只需过滤已经创建的预先存在列表就可以了。这同样适用于 `GetMethod` 和 `GetProperty`，当您第一次调用 `MethodInfo` 或 `PropertyInfo` 时，缓存将会被构建出来。

3. 参数验证和错误处理

但是，一旦我们获得了 `MethodInfo`，当我们调用 `Invoke` 时，还有很多工作要做。假如我们写了一个下面的代码。

PS:

```
PropertyInfo stringLengthField = typeof(string).GetProperty("Length",
BindingFlags.Instance | BindingFlags.Public);
var length = stringLengthField.GetGetMethod().Invoke(new Uri(), new object[0]);
```

如果我们运行它，我们会得到以下异常。

PS:

System.Reflection.TargetException: Object does not match target type.

at System.Reflection.RuntimeMethodInfo.CheckConsistency(..)

at System.Reflection.RuntimeMethodInfo.InvokeArgumentsCheck(..)

at System.Reflection.RuntimeMethodInfo.Invoke(..)

at System.Reflection.RuntimePropertyInfo.GetValue(..)

这是因为我们获得了字符串类的 Length 属性的 PropertyInfo，但是把一个错误的 Uri 类对象当成它第一个参数（应该是一个字符串类才对），这显然是不对的！除此之外，还必须对传递给调用的方法的任何参数进行验证。为了使参数传递起作用，反射 api 使用一个 object[] 的参数，里面保存所需要的参数。

因此，如果我们使用反射来调用方法 Add(int x, int y)，您将调用 methodInfo.Invoke(..., new[] {5,6})。在运行时，需要对传入的值的数量和类型进行检查，在这种情况下，要确保有 2 个值，而且它们都是 int 型的。所有这些工作的一个缺点是，它经常涉及装箱操作，会有额外的开销。

4. 安全性检查

另一个主要任务是多重安全检查。例如，不允许使用反射来调用任何您想调用的方法。有一些受限制的或“危险的方法”，只能被 .net 框架代码调用。除了黑名单之外，还需要根据调用期间必须检查的当前代码访问安全权限进行动态安全检查。

5. 反射到底有多少性能消耗

现在，我们已经知道了反射在幕后做了什么，现在就可以看看它的消耗了。

请注意，这些基准测试是通过反射直接比较读取/写入属性。在 .net 中，属性实际上是编译器为我们生成的一对 Get/Set 方法，但是，当属性只有一个简单的支持字段时，出于性能原因，.net JIT 会将方法调用内联。这意味着使用反射来访问一个属性将会以更糟糕的方式显示反射，但还是选择它，因为它是最常见的用例，出现在 ORMs、Json 序列化/反序列化库和对象映射工具中。下面是由 BenchmarkDotNet 显示的原始结果，后面是在两个不同的表中显示的相同结果。

```
// * Summary *
```

BenchmarkDotNet-v0.10.0
OS-Microsoft Windows NT 6.1.7601 Service Pack 1
Processor-Intel(R) Core(TM) i7-4800MQ CPU 2.70GHz, ProcessorCount=8
Frequency=2530673 Hz, Resolution=300.1309 ns, Timer=TSC
Host Runtime=Clr 4.0.30319.42000, Arch=64-bit RELEASE [RyuJIT]
GC-Concurrent Workstation
jitModules=clrjit-v4.6.1590.0
Job Runtime(s):
Clr 4.0.30319.42000, Arch=64-bit RELEASE [RyuJIT]

Method	Mean	StdErr	StdDev	Median	Scaled	Scaled-StdDev	Gen 0	Bytes Allocated/Op
GetViaProperty	0.2159 ns	0.0047 ns	0.0183 ns	0.2143 ns	1.00	0.00	-	0.00
GetViaDelegate	1.8903 ns	0.0082 ns	0.0306 ns	1.8830 ns	8.82	0.74	-	0.00
GetViaILEmit	2.9236 ns	0.0067 ns	0.0261 ns	2.9189 ns	13.64	1.12	-	0.00
GetViaCompiledExpressionTrees	12.3623 ns	0.0200 ns	0.0776 ns	12.3536 ns	57.65	4.74	-	0.00
GetViaFastMember	35.9199 ns	0.0528 ns	0.1976 ns	35.8827 ns	167.52	13.77	-	0.00
GetViaReflectionWithCaching	125.3878 ns	0.2017 ns	0.6987 ns	125.4703 ns	584.78	48.06	-	0.00
GetViaReflection	197.9258 ns	0.2704 ns	1.0473 ns	197.6979 ns	923.08	75.85	-	0.01
GetViaDelegateDynamicInvoke	842.9131 ns	1.2649 ns	4.8991 ns	842.7109 ns	3,931.17	323.14	440.00	419.04
SetViaProperty	1.4043 ns	0.0200 ns	0.0776 ns	1.3699 ns	6.55	0.64	-	0.00
SetViaDelegate	2.8215 ns	0.0078 ns	0.0302 ns	2.8184 ns	13.16	1.09	-	0.00
SetViaILEmit	2.8226 ns	0.0061 ns	0.0227 ns	2.8216 ns	13.16	1.08	-	0.00
SetViaCompiledExpressionTrees	10.7329 ns	0.0221 ns	0.0827 ns	10.7111 ns	50.06	4.12	-	0.00
SetViaFastMember	36.6210 ns	0.0393 ns	0.1523 ns	36.6633 ns	170.79	14.02	-	0.00
SetViaReflectionWithCaching	214.4321 ns	0.3122 ns	1.2092 ns	214.3265 ns	1,000.07	82.19	104.00	98.49
SetViaReflection	287.1039 ns	0.3288 ns	1.1855 ns	287.2946 ns	1,338.99	109.94	122.20	115.63
SetViaDelegateDynamicInvoke	922.4618 ns	2.9192 ns	10.5253 ns	921.0714 ns	4,302.17	355.96	403.87	390.99

图 8. BenchmarkDotNet 显示的原始结果

因此，我们可以清楚地看到，常规反射代码(GetViaReflection 和 SetViaReflection)比直接访问属性(GetViaProperty 和 SetViaProperty)慢得多。但是其他的结果呢，让我们更详细地探讨一下。

6. 开始优化反射

首先，我们从一个像这样的 `TestClass` 开始。

PS:

```
public class TestClass
{
    public TestClass(String data)
    {
        Data = data;
    }

    private string data;
    private string Data
    {
        get { return data; }
        set { data = value; }
    }
}
```

通过以下筛选设置代码，所有代码都可以获取到。

PS:

```
// Setup code, done only once
TestClass testClass = new TestClass("A String");
Type @class = testClass.GetType();
BindingFlag bindingFlags = BindingFlags.Instance | BindingFlags.NonPublic
BindingFlags.Public;
```

1) 正常反射

首先，我们使用常规的测试代码，它作为起始点和“最坏情况”。

PS:

```
[Benchmark]
public string GetViaReflection()
{
    PropertyInfo property = @class.GetProperty("Data", bindingFlags);
    return (string)property.GetValue(testClass, null);
}
```

2) 优化 1-缓存 PropertyInfo

接下来，通过保持对 `PropertyInfo` 的引用，而不是每次都获取它，我们可以获得一个小的速度提升。但是我们仍然比直接访问属性慢得多，这表明在反射的“调用”部分有相当大的成本。

PS:

```
// Setup code, done only once
PropertyInfo cachedPropertyInfo = @class.GetProperty("Data", bindingFlags);

[Benchmark]
public string GetViaReflection()
{

```



```

        return (string)cachedPropertyInfo.GetValue(testClass, null);
    }

```

3) 优化 2-使用快速成员

在这里，我们利用 Marc Gravell 的优秀的[快速成员库](#)，你可以看到，它们使用起来非常方便!

PS:

```
// Setup code, done only once
```

```
TypeAccessor accessor = TypeAccessor.Create(@class, allowNonPublicAccessors: true);
```

```
[Benchmark]
```

```
public string GetViaFastMember()
```

```
{
```

```
    return (string)accessor[testClass, "Data"];

```

```
}
```

注意，它所做的事情与其他做法略有不同。它**创建了一个类型访问器 (TypeAccessor)**，**允许访问类型上的所有属性，而不仅仅是一个**。但有一个不好的地方，它需要更长的运行时间。这是因为在内部，在获取它的值之前，它首先必须获取您请求的属性的委托(在本例中是 'Data')。然而，这个开销非常小，FastMember 仍然比反射快得多，而且它非常容易使用，所以我建议需要的朋友先看看它。**此项和所有后续项都将反射代码转换为可直接调用的委托，而无需每次都进行反射开销，因此可以提高速度!**值得指出的是，创建委托是有成本的。简而言之，速度的提高是因为我们只做一次昂贵的工作(安全检查等)，并存储了一个强类型的委托，我们可以一次又一次地使用它，而开销很小。如果您只做一次反射，就应当使用这些技术。通过委托读取属性没有直接读取快的原因是.NET JIT 不会像访问属性那样内联委托方法调用。**对于委托，我们需要支付方法调用的成本，而直接访问不需要。**

4) 优化 3-创建一个委托

在这个选项中，我们使用 CreateDelegate 函数将 PropertyInfo 转换为一个常规委托。

PS:

```
// Setup code, done only once
```

```
PropertyInfo property = @class.GetProperty("Data", bindingFlags);
```

```
Func<TestClass, string> getDelegate = (Func<TestClass,
string>)Delegate.CreateDelegate(typeof(Func<TestClass, string>),
```

```
property.GetGetMethod(nonPublic: true));
```

```
[Benchmark]
```

```
public string GetViaDelegate()
```

```
{
```

```
    return getDelegate(testClass);

```

```
}
```

这种方法的**缺点是我们需要在编译时知道具体的类型**，即 Func<TestClass, string>部分。在上面的代码(不，你不能使用 Func<object, string>，如果你这样做会抛出一个异常!)在大多数情况下，当我们在做反射时，我们也没想着会这么爽，否则我们不会在一开始就使用反射，所以它不是一个完美的解决方案。

5) 优化 4-编译表达式树

这里我们生成了一个委托，但不同的是我们可以直接传递一个 object，所以我们绕过了‘[优化三-创建一个委托](#)’的限制（需要知道明确的对象类型）。我们利用 .NET 表达式树 API，允许动态代码生成。

PS:

```
// Setup code, done only once
PropertyInfo property = @class.GetProperty("Data", bindingFlags);
ParameterExpression = Expression.Parameter(typeof(object), "instance");
UnaryExpression instanceCast =
    !property.DeclaringType.IsValueType ?
        Expression.TypeAs(instance, property.DeclaringType) :
        Expression.Convert(instance, property.DeclaringType);
Func<object, object> GetDelegate =
    Expression.Lambda<Func<object, object>>(<
        Expression.TypeAs(
            Expression.Call(instanceCast, property.GetGetMethod(nonPublic: true)),
            typeof(object)),
        instance)
    .Compile();

[Benchmark]
public string GetViaCompiledExpressionTrees()
{
    return (string)GetDelegate(testClass);
}
```

基于表达式的方法的完整代码可以在[使用表达式树的更快的反射博文](#)中找到。

6) 优化 5-动态代码生成与 IL emit

最后，我们来到了最底层的方法，emit 原始 IL 代码，正所谓“能力越大，责任越大”。

PS:

```
// Setup code, done only once
PropertyInfo property = @class.GetProperty("Data", bindingFlags);
Sigil.Emit getterEmitter = Emit<Func<object, string>>
    .NewDynamicMethod("GetTestClassDataProperty")
    .LoadArgument(0)
    .CastClass(@class)
    .Call(property.GetGetMethod(nonPublic: true))
    .Return();
Func<object, string> getter = getterEmitter.CreateDelegate();

[Benchmark]
public string GetViaILEmit()
{
    return getter(testClass);
}
```

```
}
```

使用表达式树(如优化 4 所示)不会像直接 emit IL 代码那样给我们提供那么多的灵活性, 毕竟它确实可以防止我们发出无效代码! 正因为如此, 如果发现自己需要 emit IL, 我强烈建议使用优秀的 [Sigil 库](#), 因为当出错时它会给出更好的错误消息!

如果(且仅当)发现自己在使用反射时遇到性能问题, 有几种不同的方法可以使其更快。这些速度提升都是通过获得一个委托来实现的, 该委托允许直接访问属性/字段/方法, 而不需要每次都通过反射进行处理。

二十八. C#常用 String 方法

二十九. StringBuilder

在 C# 中, **String** 声明之后, 在内存中的大小是不可修改的, 而 **StringBuilder** 可以自由扩展大小, **String** 分配在**栈区**, **StringBuilder** 分配在**堆区**。在 java 中 **String** 类型它都是放在堆中的。而 C# 则不同, 微软对 **String** 类型进行优化并且在处理字符串的时候用到散列表: 简单的理解就是当我们再次创建字符串相同的 **string** 时, 编译器是不会再去开辟新的内存来存储的, 它会直接指向第一次创建的地址。

PS:

```
string greetingText = "Hello from all the guys at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";
for(int i = 'z'; i >= 'a'; i--)
{
    char Old1 = (char)i;
    char New1 = (char)(i+1);
    greetingText = greetingText.Replace(Old1, New1);
}
for(int i = 'Z'; i >= 'A'; i--)
{
    char Old1 = (char)i;
    char New1 = (char)(i+1);
    greetingText = greetingText.Replace(Old1, New1);
}
Console.WriteLine("Encoded:\n" + greetingText);
```

在上面的示例中, **Replace()** 方法以一种智能的方式工作, 在某种程度上, 它并没有创建一个新字符串, 除非它实际上要对旧字符串进行某些改变。原来的字符串包含 23 个不同的小写字母, 和 3 个不同的大写字母。所以 **Replace()** 就分配一个新字符串, 共 26 次, 每个新字符串都包含 103 个字符。

因此加密过程需要在堆上有一个总共能存储 2678 个字符的字符串对象, 最终再等待被垃圾回收! 显然, 如果使用字符串频繁进行文字处理, 应用程序就会遇到严重的性能问题。

为了解决这类问题, Microsoft 提供了 **System.Text.StringBuilder** 类。**StringBuilder** 类不像 **String** 类那样能够支持非常多的方法。在 **StringBuilder** 类上可以进行的处理仅限于替换和追加或删除字符串中的文本。但是, 它的工作方式非常高效。

在使用 **String** 类构造一个字符串时, 要给它分配足够的内存来保存字符串。然而, **StringBuilder** 类通常分配的内存会比 **String** 需要的更多。开发人员可以选择指定 **StringBuilder** 要分配多少内存, 但如果没有指定, 在默认情况下就根据初始化 **StringBuilder** 实例时的字符串长度来确定内存的大小。

StringBuilder 类有两个主要的属性, **Length** 指定字符串的实际长度; **Capacity** 指定字

字符串在分配的内存中的最大长度。

对字符串的修改就在赋予 `StringBuilder` 实例的内存块中进行，这就大大提高了追加子字符串和替换单个字符的效率。删除或插入子字符串仍然效率低下，因为这需要移动随后的字符串。只有执行扩展字符串容量的操作，才需要给字符串分配新内存，才可能移动包含的整个字符串。在添加额外的容量时，从经验来看，如果 `StringBuilder` 类检测到容量超出，且容量没有设置新值，就会使自己的容量翻倍。

为了使用 `StringBuilder` 类，需要在代码中引用 `System.Text` 类。

如果使用 `StringBuilder` 对象构造最初的欢迎字符串，就可以编写下面的代码。

PS:

```
StringBuilder greetingBuilder = new StringBuilder("Hello from all the guys at Wrox Press.", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we enjoyed writing it");
```

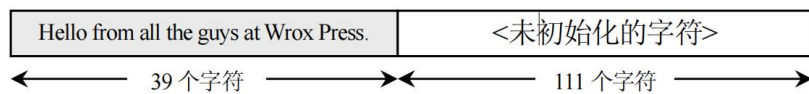


图 9. 为 `StringBuilder` 分配内存

在上面这段代码中，为 `StringBuilder` 类设置的初始容量是 150。最好把容量设置为字符串可能的最大长度，确保 `StringBuilder` 类不需要重新分配内存，因为其容量足够用了。

(容量默认设置为 16)。理论上，可以设置尽可能大的数字，足够给该容量传送一个 `int`，但如果实际上给字符串分配 20 亿个字符的空间(这是 `StringBuilder` 实例理论上允许拥有的最大空间)，系统就可能会没有足够的内存。

当我们需要增加字符数时，调用 `AppendFormat()` 方法，其他文本就放在空的空间中，不需要分配更多的内存。但是，多次替换文本才能获得 `StringBuilder` 类所带来的高效性能。

例如，如果要以前面的方式加密文本，就可以执行整个加密过程，无须分配更多的内存。

PS:

```
StringBuilder greetingBuilder =
    new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we " +
    "enjoyed writing it");
```

```
Console.WriteLine("Not Encoded:\n" + greetingBuilder);
```

```
for(int i = 'z'; i >= 'a'; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingBuilder = greetingBuilder.Replace(old1, new1);
}
```

```
for(int i = 'Z'; i >= 'A'; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
```

```
greetingBuilder = greetingBuilder.Replace(old1, new1);  
}
```

```
Console.WriteLine("Encoded:\n" + greetingBuilder);
```

这段代码使用了 `StringBuilder.Replace()` 方法，它的功能与 `String.Replace()` 一样，但**不需要在过程中复制字符串**。在上述代码中，为存储字符串而分配的总存储单元是 150 个字符，用于 `StringBuilder` 实例以及在最后一条 `Console.WriteLine()` 语句中执行字符串操作期间分配的内存。

一般，使用 `StringBuilder` 类执行对于字符串的操作，使用 `String` 类**存储字符串或显示最终结果**。

1. StringBuilder 成员

前面介绍了 `StringBuilder` 类的一个构造函数，它的参数是一个初始字符串及该字符串的容量。

`StringBuilder` 类还有几个其他的构造函数，例如，可以只提供一个字符串。

PS:

```
StringBuilder sb = new StringBuilder("Hello");  
或者用给定的容量创建一个空的 StringBuilder 类。
```

PS:

```
StringBuilder sb = new StringBuilder(20);
```

除了前面介绍的 `Length` 和 `Capacity` 属性外，还有一个只读属性 `MaxCapacity`，它表示对给定的 `StringBuilder` 实例的容量限制。在默认情况下，这由 `int.MaxValue` 给定(大约 20 亿，如前所述)。但在构造 `StringBuilder` 对象时，也可以把这个值设置为较低的值。

PS:

```
StringBuilder sb = new StringBuilder(100, 500);
```

还可以随时显式地设置容量，但如果把这个值设置为小于字符串的当前长度，或者超出了最大容量的某个值，就会抛出一个异常。

PS:

```
//超过最大容量就会抛出异常
```

```
StringBuilder sb = new StringBuilder("Hello");  
sb.Capacity = 100;
```

名 称	作 用
Append()	给当前字符串追加一个字符串
<code>AppendFormat()</code>	追加特定格式的字符串
Insert()	在当前字符串中插入一个子字符串
<code>Remove()</code>	从当前字符串中删除字符
<code>Replace()</code>	在当前字符串中，用某个字符全部替换另一个字符，或者用当前字符串中的一个子字符串全部替换另一个字符串
<code>ToString()</code>	返回当前强制转换为 <code>System.String</code> 对象的字符串(在 <code>System.Object</code> 中被重写)

不能把 `StringBuilder` 强制转换为 `String`(隐式转换和显式转换都不行)。如果要把 `stringBuilder` 的内容输出为 `String`，唯一的方式就是使用 `ToString()` 方法。

前面介绍了 `StringBuilder` 类，说明了使用它提高性能的一些方式。注意，这个类并不总能提高性能。`StringBuilder` 类基本上应在处理多个字符串时使用。但如果只是连接两个字

字符串，使用 `System.String` 类会比较好。

三十. C#元组与值元组 (Tuple And ValueTuple)

1. 元组

元组功能在 C# 7.0 及更高版本中可用，它提供了简洁的语法，用于将多个数据元素组成一个轻型数据结构。元组是一个对象，其在堆内存里分配空间。

注意！元组类型是值类型；元组元素是公共字段。（这使得元组为可变的值类型）

PS:

```
Tuple<int, string, string> person = new Tuple<int, string, string>(1, "Steve", "Jobs");
```

在上面的示例中，我们创建了一个 `Tuple`。我们为每个元素指定了一种类型，并将值传递给构造函数。但是，指定每个元素的类型很麻烦，于是 C# 引入了一个静态帮助器类 `Tuple`，该类返回 `Tuple<T>` 的实例，而不必指定每个元素的类型，如下所示。

PS:

```
var person = Tuple.Create(1, "Steve", "Jobs");
```

一个元组最多只能包含八个元素。当您尝试包含八个以上的元素时，它将产生编译器错误。

1) 元组字段名称

可以在元组初始化表达式中或元组类型的定义中显式指定元组字段的名称，如下面的示例所示：

PS:

```
var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");
```

```
(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

从 C# 7.1 开始，如果未指定字段名称，则可以根据元组初始化表达式中相应变量的名称推断出此名称，如下面的示例所示：

PS:

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

这称为元组投影初始值设定项。在以下情况下，变量名称不会被投影到元组字段名称中：

- ① 候选名称是元组类型的成员名称，例如 `Item3`、`ToString` 或 `Rest`。
- ② 候选名称是另一元组的显式或隐式字段名称的重复项。

在这些情况下，你可以显式指定字段的名称，或按字段的默认名称访问字段。

元组字段的默认名称为 `Item1`、`Item2`、`Item3` 等。始终可以使用字段的默认名称，即使字段名称是显式指定的或推断出的，如下面的示例所示：

PS:

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
```

```
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

元组赋值和元组相等比较不会考虑字段名称。

在编译时，编译器会将非默认字段名称替换为相应的默认名称。因此，显式指定或推断的字段名称在运行时不可用。

2) 元组赋值和析构

C# 支持满足以下两个条件的元组类型之间的赋值：

- ① 两个元组类型有相同数量的元素。
- ② 对于每个元组位置，右侧元组元素的类型与左侧相应的元组元素的类型相同或可以隐式转换为左侧相应的元组元素的类型。

元组元素是按照元组元素的顺序赋值的。元组字段的名称会被忽略且不会被赋值，如下面的示例所示：

PS:

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14
```

```
(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

还可以使用赋值运算符 `=` 在单独的变量中析构元组实例。为此，可以使用以下方式之一进行操作：

a. 在括号内显式声明每个变量的类型

PS:

```
var t = ("post office", 3.6);
(string destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

b. 在括号外使用 `var` 关键字来声明隐式类型化变量，并让编译器推断其类型

PS:

```
var t = ("post office", 3.6);
```

```
var (destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

c. 使用现有变量

PS:

```
var destination = string.Empty;
var distance = 0.0;

var t = ("post office", 3.6);
(destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

3) 访问元组元素

元组元素可以通过 `Item <elementnumber>` 属性访问，例如 `Item1`、`Item2`、`Item3` 等，最多可以访问 `Item7` 属性。`Item1` 属性返回第一个元素，`Item2` 返回第二个元素，依此类推。（这个顺序不能修改）最后一个元素(第 8 个元素)将使用 `Rest` 属性返回。

PS:

```
var person = Tuple.Create(1, "Steve", "Jobs");
person.Item1; // 返回 1
person.Item2; // 返回 "Steve"
person.Item3; // 返回 "Jobs"

var numbers = Tuple.Create("One", 2, 3, "Four", 5, "Six", 7, 8);
numbers.Item1; // 返回 "One"
numbers.Item2; // 返回 2
numbers.Item3; // 返回 3
numbers.Item4; // 返回 "Four"
numbers.Item5; // 返回 5
numbers.Item6; // 返回 "Six"
numbers.Item7; // 返回 7
numbers.Rest; // 返回 (8)
numbers.Rest.Item1; // 返回 8
```

通常，第 8 个位置用于嵌套元组，我们可以使用 `Rest` 属性访问该位置。

4) 嵌套元组

如果要在一个元组中包含八个以上的元素，则可以通过嵌套另一个元组对象作为第八个元素来实现。可以使用 `Rest` 属性访问最后一个嵌套元组。要访问嵌套元组的元素，请使用 `Rest.Item1.Item<elementNumber>` 属性。

PS:

```
var numbers = Tuple.Create(1, 2, 3, 4, 5, 6, 7, Tuple.Create(8, 9, 10, 11, 12, 13));
numbers.Item1; // 返回 1
numbers.Item7; // 返回 7
numbers.Rest.Item1; //返回(8,9,10,11,12,13)
numbers.Rest.Item1.Item1; //返回 8
numbers.Rest.Item1.Item2; //返回 9
```

我们可以在序列中的任何位置包括嵌套元组对象。但是，[建议将嵌套的元组放在序列的末尾，以便可以使用 Rest 属性访问它。](#)

PS:

```
var numbers = Tuple.Create(1, 2, Tuple.Create(3, 4, 5, 6, 7, 8), 9, 10, 11, 12, 13 );
numbers.Item1; // 返回 1
numbers.Item2; // 返回 2
numbers.Item3; // 返回 (3, 4, 5, 6, 7, 8)
numbers.Item3.Item1; // 返回 3
numbers.Item4; // 返回 9
numbers.Rest.Item1; //返回 13
```

5) 元组作为方法参数

PS:

```
static void Main(string[] args)
{
    var person = Tuple.Create(1, "Steve", "Jobs");
    DisplayTuple(person);
}

static void DisplayTuple(Tuple<int,string,string> person)
{
    Console.WriteLine($"Id = { person.Item1 }");
    Console.WriteLine($"First Name = { person.Item2 }");
    Console.WriteLine($"Last Name = { person.Item3 }");
}
```

6) 元组作为返回类型

PS:

```
static void Main(string[] args)
{
    var person = GetPerson();
}

static Tuple<int, string, string> GetPerson()
{
    return Tuple.Create(1, "Bill", "Gates");
}
```

7) 元组相等

从 C# 7.3 开始，元组类型支持 == 和 != 运算符。这些运算符[按照元组元素的顺序将左侧操作数的成员与相应的右侧操作数的成员进行比较。](#)

PS:

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right); // output: True
```



```
Console.WriteLine(left != right); // output: False
```

```
var t1 = (A: 5, B: 10);
```

```
var t2 = (B: 5, A: 10);
```

```
Console.WriteLine(t1 == t2); // output: True
```

```
Console.WriteLine(t1 != t2); // output: False
```

如前面的示例所示，`==` 和 `!=` 操作不会考虑元组字段名称。同时满足以下两个条件时，两个元组可比较：

两个元组具有相同数量的元素。例如，如果 `t1` 和 `t2` 具有不同数目的元素，`t1 != t2` 则不会进行编译。

对于每个元组位置，可以使用 `==` 和 `!=` 运算符对左右侧元组操作数中的相应元素进行比较。例如，`(1, (2, 3)) == ((1, 2), 3)` 不会进行编译，因为 `1` 不可与 `(1, 2)` 比较。

`==` 和 `!=` 运算符将以短路方式对元组进行比较。也就是说，**一旦遇见一对不相等的元素或达到元组的末尾，操作将立即停止。**但是，在进行任何比较之前，将对所有元组元素进行计算，如以下示例所示：

PS:

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));
```

```
int Display(int s)
```

```
{
```

```
    Console.WriteLine(s);
```

```
    return s;
```

```
}
```

```
// Output:
```

```
// 1
```

```
// 2
```

```
// 3
```

```
// 4
```

```
// False
```

8) 元组的用法

- ① 当我们想从一个方法中**返回多个值而不使用 `ref` 或 `out` 参数**时。
- ② 当我们想**通过单个参数将多个值传递给方法**时。
- ③ 当我们想暂时保存数据库记录或某些值而不创建单独的类时。

9) 元组缺点

- ① `Tuple` 是一个**引用类型**，而不是一个值类型。它在**堆上分配**，并可能导致 CPU 密集型操作。
- ② `Tuple` 被限制为包括八个元素。如果需要存储更多元素，则需要**使用嵌套元组**。但是，这**可能导致歧义**。
- ③ 可以使用名称模式 `Item <elementNumber>` 的属性访问 `Tuple` 元素，这是不太合理的。

C#7 引入了 `ValueTuple`（值元组）以克服 `Tuple` 的局限性，并使 `Tuple` 的工作更加轻松。

10) 值元组与元组的区别

`System.ValueTuple` 类型支持的 C# 元组不同于 `System.Tuple` 类型表示的元组。主要

区别如下：

- ① **System.ValueTuple** 类型是值类型，而 **System.Tuple** 类型是引用类型。
- ② **System.ValueTuple** 类型是可变的，而 **System.Tuple** 类型是不可变的。
- ③ **System.ValueTuple** 类型的数据成员是字段，而 **System.Tuple** 类型的数据成员是属性。

2. 值元组

C#7.0（.NET Framework 4.7）引入了 **ValueTuple** 结构，它是元组的值类型表示。

ValueTuple 仅在 .NET Framework 4.7 中可用。如果在项目中没有看到 **ValueTuple**，则需要安装 **ValueTuple**。（.NET Framework 4.7 或更高版本，或 .NET Standard Library 2.0 或更高版本已包含 **ValueTuple**。）

要安装 **ValueTuple** 软件包，请在解决方案资源管理器中的项目上单击鼠标右键，然后选择 **Manage NuGet Packages...**。这将打开 **NuGet 软件包管理器**。单击 **Browse** 选项卡，在搜索框中搜索 **ValueTuple**，然后选择 **System.ValueTuple** 包，如下所示。

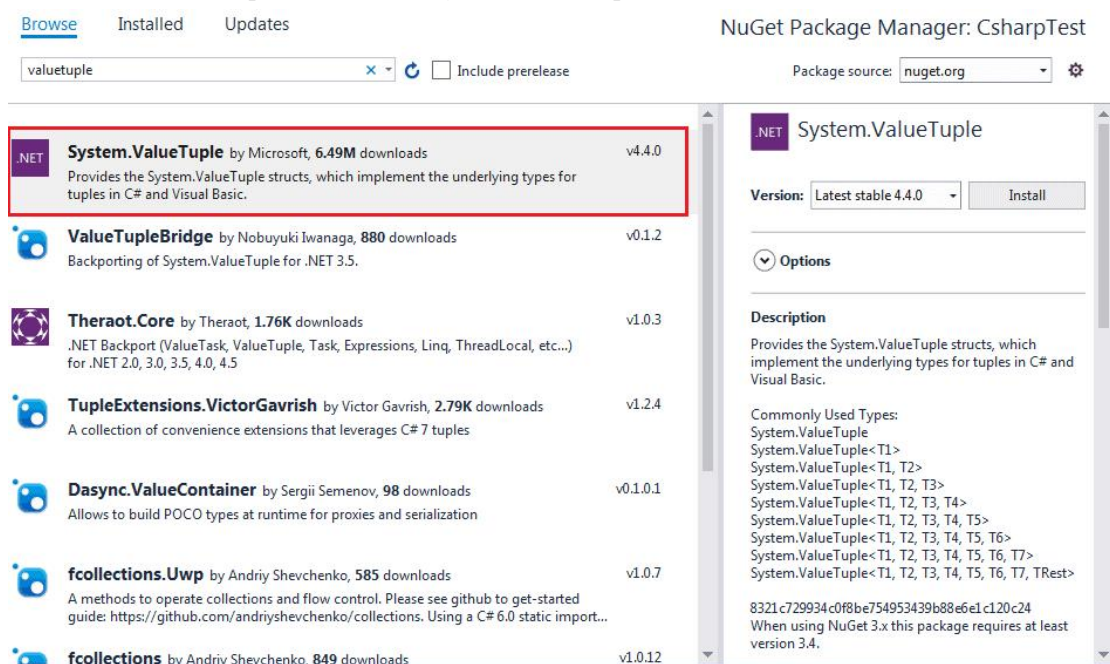


图 10. System.ValueTuple 包

1) ValueTuple 初始化

创建和初始化 **ValueTuple** 很容易。可以使用**使用括号()**并在其中指定值来创建和初始化它。

PS:

```
var person = (1, "Bill", "Gates");
//等效元组
//var person = ValueTuple.Create (1, " Bill", " Gates");"Bill", "Gates");
```

也可以通过**指定各元素的类型初始化** **ValueTuple**，如下所示。

PS:

```
ValueTuple<int, string, string> person = (1, "Bill", "Gates");
person.Item1; // 返回 1
person.Item2; // 返回 "Bill"
person.Item3; // 返回 "Gates"
```

以下是为每个成员声明类型的简写方法。

PS:

```
(int, string, string) person = (1, "Bill", "Gates");
person.Item1; // 返回 1
person.Item2; // 返回 "Bill"
person.Item3; // 返回 "Gates"
```

请注意，我们没有在上面的 tuple 初始化语句中使用 var；相反，我们在方括号中提供了每个成员值的类型。

元组至少需要两个值。以下不是元组的情况：

PS:

```
var number = (1); // int 类型，不是元组
var numbers = (1,2); // 有效元组
与 Tuple 不同，ValueTuple 可以包含八个以上的值。
```

PS:

```
var numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14);
```

2)命名成员

我们可以为 ValueTuple 属性分配名称，而不是使用默认属性名称（例如 Item1，Item2 等）。

PS:

```
(int Id, string FirstName, string LastName) person = (1, "Bill", "Gates");
person.Id; // 返回 1
person.FirstName; // 返回 "Bill"
person.LastName; // 返回 "Gates"
```

我们还可以在右侧为成员名称分配值，如下所示。

PS:

```
var person = (Id:1, FirstName:"Bill", LastName: "Gates");
```

请注意，我们可以在左侧或右侧提供成员名称，但不能在两侧提供成员名称。左侧优先于右侧。以下内容将忽略右侧的名称。

PS:

```
// PersonId, FName, LName 将被忽略。
```

```
(int Id, string FirstName, string LastName) person = (PersonId:1, FName:"Bill", LName:
"Gates");
```

//PersonId, FirstName, LastName 将被忽略。它将具有默认名称：Item1，Item2，Item3。

```
(string, string, int) person = (PersonId:1, FName:"Bill", LName: "Gates");
```

我们还可以将变量分配为成员值。

PS:

```
string firstName = "Bill", lastName = "Gates";
var per = (FirstName: firstName, LastName: lastName);
```

3)ValueTuple 作为返回类型

以下方法返回 ValueTuple。

PS:

```
static void Main(string[] args)
```

```

{
    DisplayTuple(1, "Bill", "Gates");
}

static void DisplayTuple((int, string, string) person)
{
    Console.WriteLine($"Id = { person.Item1 }");
    Console.WriteLine($"First Name = { person.Item2 }");
    Console.WriteLine($"Last Name = { person.Item3 }");
}

```

我们还可以为方法返回的 `ValueTuple` 指定不同的成员名称。

PS:

```

static void Main(string[] args)
{
    var person = GetPerson();
}

static (int, string, string) GetPerson()
{
    return (Id:1, FirstName: "Bill", LastName: "Gates");
}

```

4) 解构

可以通过解构 `ValueTuple` 来检索它的各个成员。解构声明语法将 `ValueTuple` 拆分为多个部分，并分别将这些部分分配给新变量。

PS:

```

static void Main(string[] args)
{
    // 更改属性名称
    (int PersonId, string FName, string LName) = GetPerson();
}

static (int, string, string) GetPerson()
{
    return (Id:1, FirstName: "Bill", LastName: "Gates");
}

```

我们还可以使用 `var` 代替显式数据类型名称。

PS:

```

static void Main(string[] args)
{
    // 使用 var 作为数据类型
    (var PersonId, var FName, var LName) person = GetPerson();
}

static (int, string, string) GetPerson()
{
    return (Id:1, FirstName: "Bill", LastName: "Gates");
}

```

ValueTuple 还允许对不打算使用的成员进行“丢弃”解构。

PS:

// 对未使用的成员 LName 使用 下划线 _ 丢弃

```
(var id, var FName, _) = GetPerson();
```

3. 值元组与元组示例

我们创建了元组后，其值不能进行修改；但是值元组创建后，我们是可以修改它的值的。

PS:

```
public class ValueTupleSample : MonoBehaviour
{
    public (int id, int value) sampleValueTuple;
    public (int , int ) sampleNoNameValueTuple;
    void Start()
    {
        //使用 new 关键字
        sampleValueTuple = new ValueTuple<int, int>(1, 1);
        //使用 Create 静态方法
        sampleValueTuple = ValueTuple.Create(2, 2);
        sampleValueTuple = (id: 3, value: 2);
        sampleValueTuple = sampleNoNameValueTuple;

        //未命名的值元组只能使用默认名称访问
        sampleNoNameValueTuple.Item1 = 3;
        sampleNoNameValueTuple.Item2 = 4;
    }
}
```

三十一. C#可空类型-Nullable

众所周知，**不能为值类型分配空值**。例如，`int i = null` 将抛出编译时错误。

c# 2.0 引入了可空类型，允许你将 `null` 赋值给值类型变量。您可以使用 `Nullable<T>` 来声明可空类型，其中 `T` 是一个类型。

1. 可空类型定义

PS:

```
Nullable<int> i = null;
```

可空类型可以表示其基础值类型的正确值范围，还外加一个额外的空值。例如，`Nullable<int>` 可以分配从 -2147483648 到 2147483647 的任何值，或者一个 `null` 值。

`Nullable` 类型是 `System.Nullable<t> struct` 的实例。

PS:

```
[Serializable]
public struct Nullable<T> where T : struct
{
    public bool HasValue { get; }

    public T Value { get; }
```

```
// 其他实现
}
```

Int 类型的 nullable 与普通 int 类型相同，加上一个标志，表示 int 是否有一个值(是否为 null)。其余所有都是编译器魔术，它将“null”视为有效值。

PS:

```
static void Main(string[] args)
{
    Nullable<int> i = null;

    if (i.HasValue)
        Console.WriteLine(i.Value); // 或 Console.WriteLine(i)
    else
        Console.WriteLine("Null");
}
```

如果对象已分配值，则 HasValue 返回 true；如果未分配任何值或已分配 null 值，则返回 false。

如果 NullableType.value 类型为 null 或未分配任何值，则使用 NullableType.value 访问该值将引发运行时异常。例如，如果 i 为 null，值将抛出一个异常：

```
static void Main(string[] args)
{
    Nullable<int> i = null;
    Console.WriteLine(i.Value);
}
```


 InvalidOperationException – run time

图 11. 访问为赋值的可空类型提示异常

如果不为 null，则使用 GetValueOrDefault() 方法获取实际值；如果为 null，则使用默认值。例如：

PS:

```
static void Main(string[] args)
{
    Nullable<int> i = null;

    Console.WriteLine(i.GetValueOrDefault());
}
```

2. 可空类型的简写语法

我们可以使用“?”运算符来简化语法，例如 int?, long? 而不是使用 Nullable<T>。

PS:

```
int? i = null;
double? D = null;
```

3. Null 合并运算符 - ??

使用 '??' 运算符，将可为空的类型分配给不可为空的类型。

PS:

```
int? i = null;
```

```
int j = i ?? 0;
```

```
Console.WriteLine(j);
```

在上面的示例中，i 是可为 null 的 int，如果将其分配给不可为 null 的 int j，如果 i 为 null，它将抛出运行时异常。因此，为了降低出现异常的风险，我们使用了“??”运算符，如果 i 为 null，则将 0 赋给 j。

PS:

```
using System;
```

```
namespace CalculatorApplication
```

```
{
```

```
    class NullablesAtShow
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            double? num1 = null;
```

```
            double? num2 = 3.14157;
```

```
            double num3;
```

```
            num3 = num1 ?? 5.34;        // num1 如果为空值则返回 5.34
```

```
            Console.WriteLine("num3 的值: {0}", num3);
```

```
            num3 = num2 ?? 5.34;
```

```
            Console.WriteLine("num3 的值: {0}", num3);
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

4. 赋值规则

可空类型的赋值规则与值类型的赋值规则相同。如果在函数中将可空类型声明为局部变量，则必须在使用它之前为其赋值。如果它是任何类的字段，那么默认情况下它将有一个空值。

例如，声明并使用以下 int 类型的 nullable 而不分配任何值。编译器将给出“使用未分配的局部变量‘i’”错误：

```
static void Main(string[] args)
{
    Nullable<int> i;
    Console.WriteLine(i);
}
```




图 12. 声明并使用 int 类型的 nullable 而不分配任何值

在下面的示例中，int 类型的 nullable 是该类的字段，因此不会产生任何错误。

PS:

```
class MyClass
{
    public Nullable<int> i;
}
class Program
{
    static void Main(string[] args)
    {
        MyClass mycls = new MyClass();

        if(mycls.i == null)
            Console.WriteLine("Null");
    }
}
```

5. Nullable 类比较方法

Null 被认为小于任何值，所以比较运算符不能用于 null。看以下示例，其中 i 既不小于 j，也不大于 j，也不等于 j:

PS:

```
static void Main(string[] args)
{
    int? i = null;
    int j = 10;

    if (i < j)
        Console.WriteLine("i < j");
    else if (i > 10)
        Console.WriteLine("i > j");
    else if (i == 10)
        Console.WriteLine("i == j");
    else
        Console.WriteLine("无法比较");
}
```

Nullable 静态类是 Nullable 类型的辅助类。它提供了比较可空类型的比较方法。它还具

有 `GetUnderlyingType` 方法，该方法返回可为 `null` 的类型的基础类型参数。

PS:

```
static void Main(string[] args)
{
    int? i = null;
    int j = 10;

    if (Nullable.Compare<int>(i, j) < 0)
        Console.WriteLine("i < j");
    else if (Nullable.Compare<int>(i, j) > 0)
        Console.WriteLine("i > j");
    else
        Console.WriteLine("i = j");
}
```

6. 可空类型的特性

- ① 可空类型只能与值类型一起使用。
- ② 如果 `Value` 为 `null`，则 `Value` 属性将抛出 `InvalidOperationException`；否则将返回
- 值。
- ③ 如果变量包含值，则 `HasValue` 属性返回 `true`；如果为 `null`，则返回 `false`。
- ④ 只能将 `==` 和 `!=` 运算符与可为空的类型一起使用。对于其他比较，请使用 `Nullable` 静态类。
- ⑤ 不允许使用嵌套的可空类型。`Nullable <Nullable <int >> i;` 将给出编译时错误。

7. 注意的要点

- ① `Nullable <T>` 类型允许将 `null` 分配给值类型。
- ② `?` 运算符是 `Nullable` 类型的简写语法。
- ③ 使用 `value` 属性获取可空类型的值。
- ④ 使用 `HasValue` 属性检查是否将值分配给可空类型。
- ⑤ 静态 `Nullable` 类是一个帮助器类，用于比较可空类型。

三十二. NULL 检查运算符(?.)

下面是一个常规的判 `null` 代码。

PS:

```
int? firstX = null;
if (points != null)
{
    var first = points.FirstOrDefault();
    if (first != null)
        firstX = first.X;
}
```

在 C# 6.0 中，引入了一个 `?.` 的运算符，前面的代码可以改成下面的形式。

PS:

```
int? firstX = points?.FirstOrDefault()?.X;
```

从这个例子中我们也可以看出它的基本用法：如果对象为 `NULL`，则不进行后面的获取成员的运算，否则直接返回 `NULL`。

需要注意的是，由于“?. “运算符返回的可以是 NULL，当返回的成员类型是 struct 类型的时候，”?. “和”. “运算符的返回值类型是不一样的。

三十三. C#异步

1. 什么是异步

同步和异步主要用于修饰方法。当一个方法被调用时，调用者需要等待该方法执行完毕并返回才能继续执行，我们称这个方法是同步方法；当一个方法被调用时立即返回，并获取一个线程执行该方法内部的业务，调用者不用等待该方法执行完毕，我们称这个方法为异步方法。

异步的好处在于非阻塞(调用线程不会暂停执行去等待子线程完成)，因此我们把一些不需要立即使用结果、较耗时的任务设为异步执行，可以提高程序的运行效率。net4.0 在 ThreadPool 的基础上推出了 Task 类，微软极力推荐使用 Task 来执行异步任务，现在 C# 类库中的异步方法基本都用到了 Task；net5.0 推出了 async/await，让异步编程更为方便

基于任务的异步编程模型 (TAP) 提供了异步代码的抽象化。我们只需像往常一样将代码编写为一连串语句即可。就如每条语句在下一句开始之前完成一样，我们可以流畅地阅读代码。编译器将执行许多转换，因为其中一些语句可能会开始运行并返回表示正在进行的工作的 Task。

这是此语法的目标：启用像语句序列一样读取的代码，但根据外部资源分配和任务完成时以更复杂的顺序执行。这与人们为包含异步任务的流程给予指令的方式类似。在本文中，我将使用有关早餐的说明示例来讲解 await 关键字和 async。

倒一杯咖啡。

加热平底锅，然后炸两个鸡蛋。

煎三片培根。

烤两片面包。

在烤面包上加黄油和果酱。

倒一杯橙汁。

如果你有烹饪经验，便可通过异步方式执行这些指令。你会先开始加热平底锅以备煎蛋，接着再从培根着手。你可将面包放进烤面包机，然后再煎鸡蛋。在此过程的每一步，你都可以先开始一项任务，然后将注意力转移到准备进行的其他任务上。

做早餐是非并行异步工作的一个好示例。单人(或单线程)即可处理所有这些任务。继续早餐类比，一个人可以通过在第一个任务完成之前启动下一个任务来异步做早餐。不管是否有人在看着，做早餐的过程都在进行。在开始加热平底锅准备煎蛋的同时就可以开始煎了培根。在开始煎培根后，你可以将面包放进烤面包机。

对于并行算法而言，你则需要多名厨师(或线程)。一名厨师煎鸡蛋，一名厨师煎培根，依次类推。每名厨师将仅专注于一项任务。每个厨师(或线程)将被同步阻止，等待熏肉准备好翻转，或烤面包弹出。



图 13. 同步方法做早餐的耗时

同步准备早餐大约需要 30 分钟，因为总计是每个任务的总和。

计算机不会按人类的方式来解释这些指令。计算机将阻塞每条语句，直到工作完成，然后再继续运行下一条语句。这将创造出令人不满意的早餐。在完成早期任务之前，以后的任务才会启动。这样做早餐花费的时间要长得多，有些食物在上桌之前就已经凉了。

如果你希望计算机异步执行上述指令，则必须编写异步代码。

这些问题对即将编写的程序而言至关重要。编写客户端程序时，你希望 UI 能够响应用户输入。从 Web 下载数据时，你的应用程序不应让手机出现卡顿。编写服务器程序时，你不希望线程受到阻塞。这些线程可以用于处理其他请求。存在异步替代项的情况下使用同步代码会增加你进行扩展的成本。你需要为这些受阻线程付费。

成功的现代应用程序需要异步代码。在没有语言支持的情况下，编写异步代码需要回调、完成事件，或其他掩盖代码原始意图的方法。同步代码的优点是，它的分步操作使其易于浏览和理解。传统的异步模型迫使我们侧重于代码的异步性质，而不是代码的基本操作。

2. Task

Task 是在 ThreadPool 的基础上推出的，我们简单了解下 ThreadPool。ThreadPool 中有若干数量的线程，如果有任务需要处理时，会从线程池中获取一个空闲的线程来执行任务，任务执行完毕后线程不会销毁，而是被线程池回收以供后续任务使用。当线程池中所有的线程都在忙碌时，又有新任务要处理时，线程池才会新建一个线程来处理该任务，如果线程数量达到设置的最大值，任务会排队，等待其他任务释放线程后再执行。线程池能减少线程的创建，节省开销。

PS:

```
static void Main(string[] args)
```

```

{
    for (int i = 1; i <= 10; i++)
    {
        //ThreadPool 执行任务
        ThreadPool.QueueUserWorkItem(new WaitCallback((obj) => {
            Console.WriteLine($"第{obj}个执行任务");
        }),i);
    }
    Console.ReadKey();
}

```

ThreadPool 相对于 **Thread** 来说可以减少线程的创建，有效减小系统开销；但是 **ThreadPool** 不能控制线程的执行顺序，我们也不能获取线程池内线程取消/异常/完成的通知，即我们不能有效监控和控制线程池中的线程。

3. 异步方法-async/await

异步编程的核心是 **Task** 和 **Task<T>** 对象，这两个对象对异步操作建模。它们受关键字 **async** 和 **await** 的支持。在大多数情况下模型十分简单：

对于 I/O 绑定代码，等待一个在 **async** 方法中返回 **Task** 或 **Task<T>** 的操作。

对于 CPU 绑定代码，等待一个使用 **Task.Run** 方法在后台线程启动的操作。

通过使用异步编程，可以避免性能瓶颈并增强应用程序的总体响应能力。但是，编写异步应用程序的传统技术可能比较复杂，使它们难以编写、调试和维护。C# 5.0 引入了一种简便方法，即异步编程。此方法利用了 .NET Framework 4.5 及更高版本、.NET Core 和 Windows 运行时中的异步支持。编译器可执行开发人员曾进行的高难度工作，且应用程序保留了一个类似于同步代码的逻辑结构。因此，只需做一小部分工作就可以获得异步编程的所有好处。

对于异步方法，我们需要使用 **Async** 修饰符修饰，且该方法的返回类型仅有三种：**void**, **Task**, **Task<T>**。在方法的内部使用 **await** 关键字标明开始执行异步代码，**await** 标志前的代码是同步执行，**await** 标志的方法是异步执行，**await** 标志的方法后面的代码相当于“回调函数”，在 **await** 标志的异步方法后面执行。使用 **async** 和 **await** 实现的异步代码，也可以使用 **Task** 对象的 **Wait()** 等方法替代。

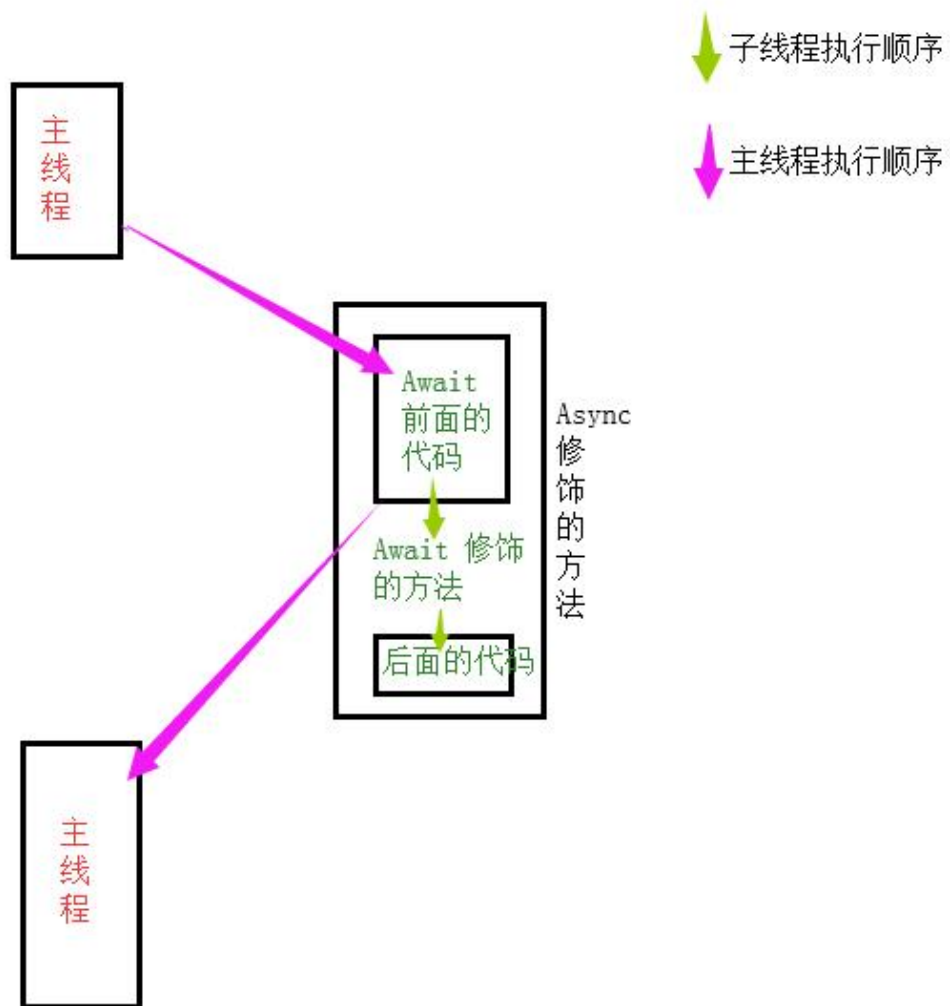


图 14. 使用 Async 和 Await 异步编程之后代码的执行顺序

PS:

```
static void Main(string[] args)
{
    string content = GetContentAsync(Environment.CurrentDirectory +
@"/test.txt").Result;
    Console.WriteLine(content);
    Console.ReadKey();
}
//异步读取文件内容
async static Task<string> GetContentAsync(string filename)
{
```

```

        FileStream fs = new FileStream(filename, FileMode.Open);
        var bytes = new byte[fs.Length];
        //ReadAsync 方法异步读取内容，不阻塞线程
        Console.WriteLine("开始读取文件");
        int len = await fs.ReadAsync(bytes, 0, bytes.Length);
        string result = Encoding.UTF8.GetString(bytes);
        return result;
    }

```

async/await 让异步编码变得更简单，我们可以像写同步代码一样去写异步代码。**注意**一个小问题：异步方法中方法返回值为 Task，代码中的返回值为 T。上述示例中 GetContentAsync 的签名返回值为 Task，而代码中返回值为 string。牢记这一细节对我们分析异步代码很有帮助。

异步方法签名的返回值有以下三种：

1) Task

如果调用方法想通过调用异步方法**获取一个 T 类型的返回值**，那么签名**必须**为 Task；

2) Task

如果调用方法**不想通过异步方法获取一个值**，仅仅想追踪异步方法的执行状态，那么我们可以设置异步方法签名的返回值为 Task；

3) void

如果调用方法**仅仅只是调用**一下异步方法，**不和异步方法做其他交互**，我们可以设置异步方法签名的返回值为 void，这种形式也叫做“调用并忘记”。

通过上面的示例。我们知道 **async/await 是基于 Task 的**，而 Task 是对 ThreadPool 的封装改进，主要是为了更有效的控制线程池中的线程(ThreadPool 中的线程，我们很难通过代码控制其执行顺序，任务延续和取消等等)；**ThreadPool 基于 Thread 的**，主要目的是**减少 Thread 创建数量和管理 Thread 的成本**。async/await Task 是 C# 中更先进的，也是微软大力推广的特性，我们在开发中可以尝试使用 Task 来替代 Thread/ThreadPool，处理本地 IO 和网络 IO 任务是尽量使用 async/await 来提高任务执行效率。

4. Unity 中使用 Async 和 Await

要在 Uni 中使用 Async 和 Await，我们需要引入一个命名空间，即 System.Threading.Tasks。

PS:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Threading.Tasks;

```

```

public class Mover
{
    public async Task Move(Transform transform)
    {
        await Task.Delay(System.TimeSpan.FromMilliseconds(0.5));
        transform.position += Vector3.right;
    }
}

```

```

}
public class AsyncTest : MonoBehaviour
{
    // Start is called before the first frame update:
    async void Start()
    {
        Mover move = new Mover();
        await move.Move(this.transform);
    }
}

```

5. 为什么 Thread < ThreadPool

Thread 默认创建的是**前台线程**，每次初始化一个 **Thread** 的实例，就会创建一个新的线程！

ThreadPool 默认创建的是**后台线程**，每次启用线程会从线程池寻找空闲的线程，如果找到了，就会调用这个空闲的线程，而不是直接创建一个新的线程。而每次创建一个线程，起码约消耗 1M 的内存。因此在**需要开启大量线程的情况下**，相比于 **Thread**，**ThreadPool** 具有**减少开启新线程消耗的资源,以及统一管理线程的优势！**

6. 为什么 ThreadPool < Task

ThreadPool 使用的是**线程池全局队列**，全局队列中的线程依旧会存在竞争共享资源的情况，从而影响性能。

Task 基于 **ThreadPool** 实现，相当于 **ThreadPool** 的优化版。它不再使用线程池的全局队列，而是**使用的本地队列**，使线程之间的资源竞争减少。同时 **Task** 提供了丰富的 API 来管理线程、控制。但是相对前面的两种耗内存，**Task** 依赖于 **CPU** 对于多核的 **CPU** 性能远超前两者，单核的 **CPU** 三者的性能没什么差别。

7. 为什么 Task < Async await

现在需要实现一个异步写数据库,写成功之后需要在页面上弹出一个小提示。

如果用 `task.result` 来实现的话，那么在获取到结果之前，`task` 会使你的项目进入一个阻塞状态。也就是说在小窗弹出来之前，你的程序会一直卡住！从而形成异步编程中的同步阻塞。

而 **Async await** 则不会发生这种情况。需要做的仅仅只是在 `await` 修饰的方法后面调一个弹出小窗的方法而已，也就是传说中的异步阻塞。

Async await 风格的编程会使代码看起来更像是同步代码，也就更像清新移动更通俗。

三十四. 类的执行顺序

父类->子类->静态块->静态字段->非静态块->非静态字段->构造器->方法。

三十五. 装箱和拆箱的区别

1. 装箱

值类型转换成引用类型。

2. 装箱操作

托管堆分配内存，值类型拷贝数据，object 地址指向托管堆对象。

3. 拆箱

引用类型转换成值类型。

4. 拆箱操作

根据 object 引用地址找到托管堆上的数据，栈上数据拷贝。

避免装箱操作，可以采用重载或泛型来解决。

5. C#中常规容器和泛性容器有什么区别？哪种效率更高？

常规容器有拆箱和装箱操作，速度慢，消耗性能；泛型容器效率更高，因为没有拆装箱的操作。

6. 为何需要装箱

值类型是在声明时就初始化了，因为它一旦声明就有了自己的空间因此它不可能为 null，也不能为 null。而引用类型在分配内存后，它其实只是一个空壳子，可以认为是指针，初始化后不指向任何空间，因此默认为 null。

值类型有，所有整数，浮点数，bool，以及 Struct 声明的结构，这里要注意 Struct 部分，它是经常我们犯错误的地方，常常很多人会把它当作类来使用是很错误的行为。因为它是值类型，在复制操作时是通过直接拷贝数据完成操作的，所以常常会有 a、b 同是结构的实例，a 赋值给了 b，当 b 更改了数据后发现 a 的数据却没有同步的疑问出现，事实上根本就是两个数据空间，当 a 赋值给 b 时其实并不是引用拷贝，而是整个数据空间拷贝，相当于有了 a、b 为两个不同西瓜，只是长得差不多而已。

引用类型包括，类，接口，委托(委托也是类)，数组以及内置的 object 与 string。前面说了 delegate 也是类，类都是引用类型，虽然有点问题也不妨碍它是一个比较好记的口号。虽然 int 等值类型也都是类，只不过它们是特殊的类，是值类型的类，因为在 C#里万物皆是类。

话锋一转这里稍微阐述下堆和栈内存。

栈是本着先进后出的数据结构(LIFO)原则的存储机制，它是一段连续的内存，所以对栈数据的定位比较快速，而堆则是随机分配的空间，处理的数据比较多，无论如何，至少要两次定位。堆内存的创建和删除节点的时间复杂度是 $O(\log n)$ 。栈创建和删除的时间复杂度则是 $O(1)$ ，栈速度更快。

那么既然栈速度这么快，全部用栈不就好了。这又涉及到生命周期问题，由于栈中的生命周期是必须确定的，销毁时必须按次序销毁，从最后分配的块部分开始销毁，创建后什么时候销毁必须是一个定量，所以在分配和销毁时不灵活，基本都用于函数调用和递归调用中，这些生命周期比较确定的地方。相反堆内存可以存放生命周期不确定的内存块，满足当需要删除时再删除的需求，所以堆内存相对于全局类型的内存块更适合，分配和销毁更灵活。

7. 装箱、拆箱对执行效率有哪些影响，如何优化

由于装箱、拆箱时生成的是全新的对象，不断的分配和销毁内存不但会大量消耗 CPU，也同时增加了内存碎片，降低了性能。

最需要我们做的就是减少装箱、拆箱的操作，在我们编程规范中要牢记这种比较浪费 CPU 的操作，在平时编程要特别注意。

整数、浮点数、布尔等数值型变量的变化手段很少，变不出什么花样来，主要靠加强规范减少装拆箱的情况来提高性能。Struct 有点不一样，它既是值类型，又可以像类一样继承，用途多转换的途径多可变的花样多，稍不留神花样就变成了麻烦，所以这里讲讲 Struct 变化后的优化方法。

1) Struct 通过重载函数来避免拆箱、装箱

比如常用的 ToString(), GetType()方法，如果 Struct 没有写重载 ToString()和 GetType()的方法，就会在 Struct 实例调用它们时先装箱再调用，导致内存块重新分配造成性能损耗，所以对于那些需要调用的引用方法，必须重载。

2)通过泛型来避免拆箱、装箱

不要忘了 Struct 也是可以继承的，在不同的、相似的、父子关系的 Struct 之间可以用泛型来传递参数，这样就不用装箱后再传递了。

比如 B, C 继承 A，就可以有这个泛型方法 `void Test(T t) where T:A`，以避免使用 object 引用类型形式传递参数。

3)通过继承统一的接口提前拆箱、装箱，避免多次重复拆箱、装箱

很多时候拆装箱不可避免，那么我们就让多种 Struct 继承某个统一的接口，不同的 Struct 就可以有相同的接口。把 Struct 传递到其他方法里去时就相当于提前进行了装箱操作，在方法中得到的是引用类型的值，并且有它需要的接口，避免了在方法中重复多次的拆装箱操作。

比如 Struct A 和 Struct B 都继承接口 I，我们调用的方法是 `void Test(I i)`。当调用 Test 方法时传进去的 Struct A 或 Struct B 的实例都相当于提前做了装箱操作，Test 里拿到的参数后就不用再担心内部再次装箱拆箱问题了。

最后依然要提醒大家，struct 值类型数据结构如果没有理解它的原理用起来可能会引起很多麻烦，切记盲目认为使用结构体会让性能提升，在没有完全彻底理解之前就冒然大量使用可能会对程序性能带来重创。

三十六. 深拷贝与浅拷贝

8. 浅拷贝

将对象中的所有字段复制到新的对象（副本）中。其中值类型字段的值被复制到副本之后，在副本中的修改不会影响到源对象的值；而引用类型的字段被复制到副本中是引用类型的引用，而不是引用的对象，在副本中对引用类型的字段值做修改会影响到源对象本身。就像一个人改名了一样，他还是这个人，只不过名字变了而已。

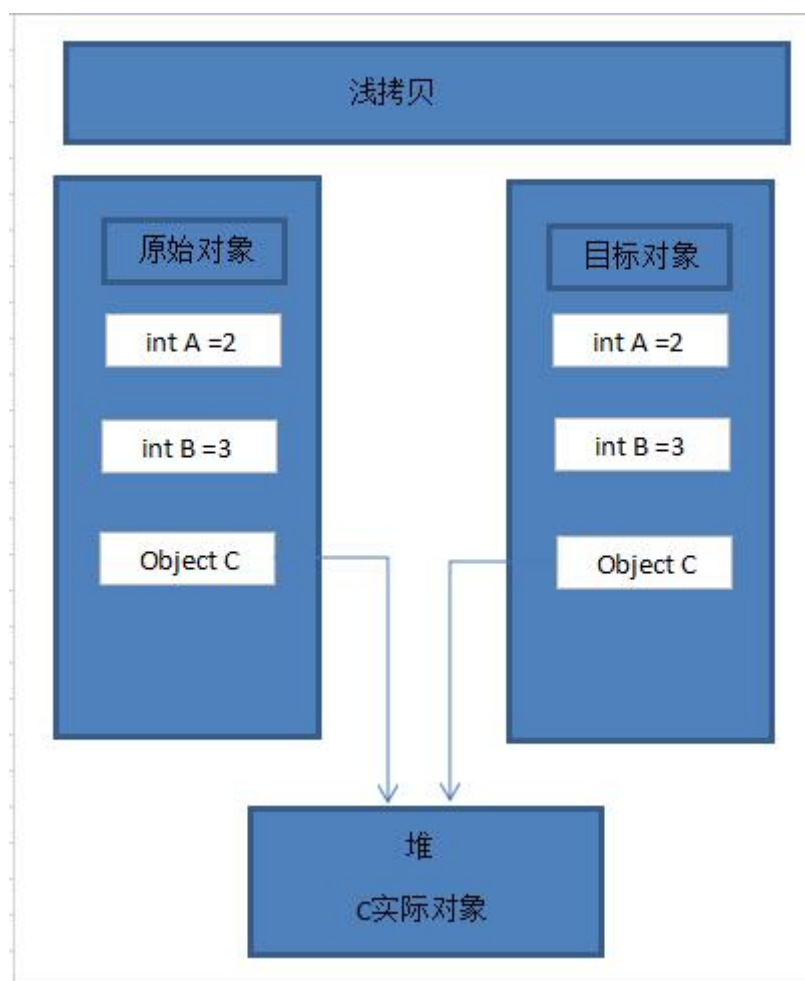


图 15. 浅拷贝

9. 深拷贝

深拷贝指的是拷贝一个对象时，不仅仅把对象的引用进行复制，还把该对象引用的值也一起拷贝。与浅拷贝不同的就是，深拷贝后的拷贝对象就和源对象相互独立，其中任何一个对象的改动都不会对另外一个对象造成影响。

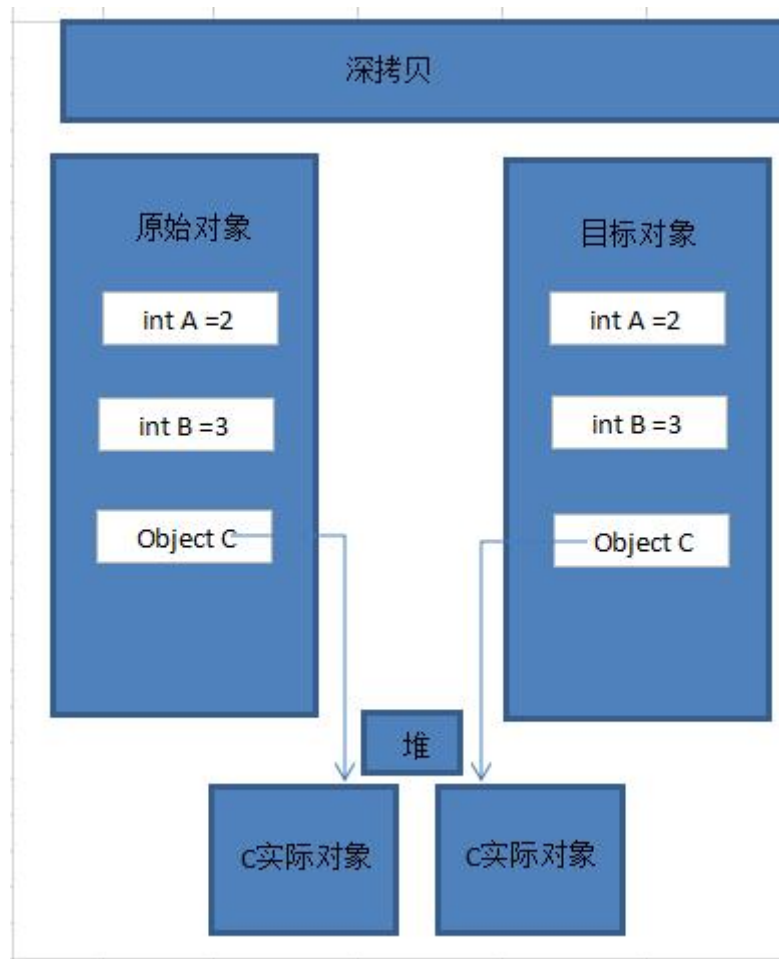


图 16. 深拷贝

10. ICloneable 接口

无论是浅拷贝还是深拷贝,微软都建议用 `Class` 继承 `ICloneable` 接口的方式告诉调用者:该类型可以被拷贝。当然, `ICloneable` 接口只提供了一个声明为 `Clone` 的方法,我们可以根据需求在 `Clone` 方法内实现浅拷贝或深拷贝。

在 Unity 中使用这个接口,需要引用一个命名空间,也即 `System`。

PS:

```
class Employee : ICloneable
{
    public string IDCode { get; set; }
    public int Age { get; set; }
    public Department Department { get; set; }

    #region ICloneable 成员

    public object Clone()
    {
        return this.MemberwiseClone();
    }

    #endregion
}
```

```

    }

    class Department
    {
        public string Name { get; set; }
        public override string ToString()
        {
            return this.Name;
        }
    }
}

```

调用 PS:

```

Employee mike = new Employee() { IDCode = "Mike", Age = 30, Department = new
Department() { Name = "Dep1" } };
Employee rose = mike.Clone() as Employee;
Console.WriteLine(rose.IDCode);
Console.WriteLine(rose.Age);
Console.WriteLine(rose.Department);
Console.WriteLine("开始改名 mike 的值: ");

```

```

mike.IDCode = "Allen";
mike.Age = 60;
mike.Department.Name = "Dep2";
Console.WriteLine(rose.IDCode);
Console.WriteLine(rose.Age);
Console.WriteLine(rose.Department);

```

注意 Employee 的 IDCode 属性是 string 类型。理论上 string 类型是引用类型，但是由于该引用类型的特殊性（无论是实现还是语义），Object.MemberwiseClone 方法仍旧为其创建了副本。也就是说，在浅拷贝过程，我们应该将字符串看成是值类型。

Employee 的 Department 属性是一个引用类型，所以，如果改变了源对象 mike 中的值，副本 rose 中的值也会随之一起变动。

Employee 的深拷贝有几种实现方法，最简单的方法是手动对字段逐个进行赋值。但这种方法容易出错，也就是说，如果类型的字段发生变化或有增减，那么该拷贝方法也要发生相应的变化。Employee 深拷贝的一个简单实现代码如下。

PS:

```

class Employee : ICloneable
{
    public string IDCode { get; set; }
    public int Age { get; set; }
    public Department Department { get; set; }

    #region ICloneable 成员

    public object Clone()
    {

```

```

        using(Stream objectStream = new MemoryStream())
        {
            IFormatter formatter = new BinaryFormatter();
            formatter.Serialize(objectStream, this);
            objectStream.Seek(0, SeekOrigin.Begin);
            return formatter.Deserialize(objectStream) as Employee;
        }
    }
}

```

```

#endregion

```

```

}

```

一旦进行了深度拷贝，我们修改 `mike` 的值时，就不会影响到副本 `rose` 的值。

由于接口 `ICloneable` 只有一个模棱两可的 `Clone` 方法，所以，如果要在一个类中同时实现深拷贝和浅拷贝，只能由我们自己实现两个额外的方法，声明为 `DeepClone()` 和 `ShallowClone()`。

PS:

```

[Serializable]

```

```

class Employee : ICloneable

```

```

{
    public string IDCode { get; set; }
    public int Age { get; set; }
    public Department Department { get; set; }
}

```

```

#region ICloneable 成员

```

```

public object Clone()
{
    return this.MemberwiseClone();
}

```

```

#endregion

```

```

public Employee DeepClone()
{
    using(Stream objectStream = new MemoryStream())
    {
        IFormatter formatter = new BinaryFormatter();
        formatter.Serialize(objectStream, this);
        objectStream.Seek(0, SeekOrigin.Begin);
        return formatter.Deserialize(objectStream) as Employee;
    }
}

```

```

public Employee ShallowClone()

```

```

    {
        return Clone() as Employee;
    }
}

```

11. 几种深拷贝方法探究及性能比较

1) 手写创建对象

简单对象创建，不考虑有构造函数的情况。

PS:

```
NewUserInfo newInfo = new NewUserInfo()
```

```

{
    Name = info.Name,
    Age = info.Age,
    UserId = info.UserId,
    Address = info.Address,
    UpdateTime = info.UpdateTime,
    CreateTime = info.CreateTime,
};

```

100 万次执行时间为 39.4073ms，位居第一。当然，在这种不考虑构造函数的情况下，手写创建肯定是最快的。但是同时，如果遇到复杂对象，代码量也是最多的。

2) C# 反射

PS:

```

private static TOut TransReflection<TIn, TOut>(TIn tIn)
{
    TOut tOut = Activator.CreateInstance<TOut>();
    var tInType = tIn.GetType();
    foreach (var itemOut in tOut.GetType().GetProperties())
    {
        var itemIn = tInType.GetProperty(itemOut.Name); ;
        if (itemIn != null)
        {
            itemOut.SetValue(tOut, itemIn.GetValue(tIn));
        }
    }
    return tOut;
}

```

调用：

```
NewUserInfo newInfo = TransReflection<UserInfo, NewUserInfo>(info);
```

3) Json 字符串序列化

使用 System.Text.Json 作为序列化和反序列化工具。100 万次执行时间为 2222.2078ms，比反射慢一点点。

PS:

```
UserInfo newInfo = JsonSerializer.Deserialize<UserInfo>(JsonSerializer.Serialize(info));
```

4)对象二进制序列化

首先不推荐使用这种方式，一是 `BinaryFormatter.Serialize` 微软已不推荐使用（据微软官网文档说是漏洞，具体有什么漏洞没细究），二是必须在要序列化的对象上面写上 `Serializable` 的关键字，三是速度并不理想。

PS:

```
private static TOut ObjectMemoryConvert<TIn, TOut>(TIn tIn)
{
    using (MemoryStream ms = new MemoryStream())
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(ms, tIn);
        ms.Position = 0;
        return (TOut)formatter.Deserialize(ms);
    }
}
```

100 万次执行时间为 8545.9835ms，讲道理应该是比 Json 序列化要更快的，但是实际上慢了许多。

5)AutoMapper

PS:

```
//循环外创建 MapperConfig
var config = new MapperConfiguration(cfg => cfg.CreateMap<UserInfo, UserInfo>());
var mapper = config.CreateMapper();
```

```
//循环内调用
```

```
UserInfo newInfo = mapper.Map<UserInfo>(info);
```

100 万次执行时间为 267.5073ms，位居第三。

6)表达式树

性能让人大吃一惊，其原理是反射和表达式树相结合，先用反射获取字段然后缓存起来，再用表达式树赋值。

PS:

```
public static class TransExp<TIn, TOut>
{
    private static readonly Func<TIn, TOut> cache = GetFunc();
    private static Func<TIn, TOut> GetFunc()
    {
        ParameterExpression parameterExpression = Expression.Parameter(typeof(TIn),
"p");
        List<MemberBinding> memberBindingList = new List<MemberBinding>();

        foreach (var item in typeof(TOut).GetProperties())
        {
            if (!item.CanWrite) continue;
            MemberExpression property = Expression.Property(parameterExpression,
typeof(TIn).GetProperty(item.Name));
```

```

        MemberBinding memberBinding = Expression.Bind(item, property);
        memberBindingList.Add(memberBinding);
    }

    MemberInitExpression memberInitExpression =
Expression.MemberInit(Expression.New(typeof(TOut)), memberBindingList.ToArray());
    Expression<Func<TIn, TOut>> lambda = Expression.Lambda<Func<TIn,
TOut>>(memberInitExpression, new ParameterExpression[] { parameterExpression });

    return lambda.Compile();
}

public static TOut Trans(TIn tIn)
{
    return cache(tIn);
}
}
调用

```

UserInfo newInfo = TransExp<UserInfo, UserInfo>.Trans(info);

100 万次执行时间为 77.3653ms，位居第二。仅比手写慢一点点。

三十七. C#堆和栈的区别

C#中**栈是编译期间就分配好的内存空间**，因此我们的代码中必须就栈的大小明确的定义；**堆是程序运行期间动态分配的内存空间**，我们可以根据程序的运行情况确定要分配的堆内存的大小。

堆和栈都是程序运行时的内存区域，我们把内存分为堆空间和栈空间。**栈空间比较小，但是读取速度快；而堆的空间比较大，但是读取速度慢。**

1. 内存分配和释放速度

栈内存：栈内存的分配和释放速度通常非常快，因为它使用一种称为 "栈指针增减" 的机制，仅需要移动栈指针来分配和释放内存。这种机制非常高效，适用于局部变量的生命周期。

堆内存：堆内存的分配和释放速度相对较慢，因为堆内存的分配需要在堆中搜索可用的内存块，而释放时需要进行垃圾回收或手动管理，以避免内存泄漏。这些额外的操作会导致分配和释放速度的降低。

2. 内存访问速度

栈内存：由于栈内存的局部性较高，变量通常**在内存中紧密存储**，所以对栈内存的访问速度更快，因为缓存利用率较高。

堆内存：堆内存中的数据**可能分散存储**，因此在访问时可能会导致缓存未命中，从而降低访问速度。

3. 内存管理复杂性

栈内存：栈内存的管理相对简单，**由编译器自动处理**。变量的分配和释放在变量的作用域结束时自动完成。

堆内存：堆内存的管理较复杂，**需要手动进行分配和释放**。如果不适当地进行内存管理，可能会导致内存泄漏或悬挂指针等问题。

三十八. C# 特殊字符

特殊字符是**预定义的上下文字符**，用于修改最前面插入了此类字符的程序元素（文本字符串、标识符或属性名称）。

@	逐字字符串标识符字符。
\$	内插的字符串字符。

1. \$ - 字符串内插

\$ 特殊字符**将字符串文本标识为内插字符串**。内插字符串是可能包含内插表达式的字符串文本。将内插字符串解析为结果字符串时，带有内插表达式的项会替换为表达式结果的字符串表示形式。

字符串内插为格式化字符串提供了一种可读性和便捷性更高的方式。它比字符串复合格式设置更容易阅读。比较一下下面的示例，它使用了这两种功能产生相同的输出。

PS:

```
string name = "Mark";
var date = DateTime.Now;

// 复合格式
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name,
date.DayOfWeek, date);
// 字符串内插
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm}
now.");
```

1) 内插字符串的结构

若要将字符串标识为内插字符串，可**在该字符串前面加上\$符号**。字符串字面量开头的\$ 和 " 之间不能有任何空格。若要连接多个内插字符串，请将 \$ 特殊字符添加到每个字符串字面量。

具备内插表达式的项的结构如下所示。

PS:

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

元素	描述
interpolationExpression	生成需要设置格式的结果的表达式。 null 的字符串表示形式为 String.Empty。
alignment	常数表达式，它的值定义表达式结果的字符串表示形式中的最小字符数。如果值为 正 ，则字符串表示形式为 右对齐 ；如果值为 负 ，则为 左对齐 。
formatString	受表达式结果类型支持的格式字符串。

PS:

```
Console.WriteLine($"{ "Left",-7} { "Right",7}");
```

```
const int FieldWidthRightAligned = 20;
```

```
Console.WriteLine($"{ Math.PI,FieldWidthRightAligned} - default formatting of the pi
```

```
number");
    Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal
digits of the pi number");
    // Expected output is:
    // |Left    | Right|
    // 3.14159265358979 - default formatting of the pi number
    //3.142 - display only three decimal digits of the pi number
```

可以在内插的原始字符串字面量中使用多个 \$ 字符，以在输出字符串中嵌入 { 和 } 字符，而无需对这些字符进行转义。

PS:

```
int X = 2;
int Y = 3;
```

```
var pointMessage = $$$"The point {{{X}}}, {{{Y}}} is {{Math.Sqrt(X * X + Y * Y)}} from
the origin$$$";
```

```
Console.WriteLine(pointMessage);
```

```
// output: The point {2, 3} is 3.605551275463989 from the origin
```

如果输出字符串应包含重复的 { 或 } 字符，可以添加更多的 \$ 来指定内插字符串。**任何比 \$ 数短的 { 或 } 序列将被嵌入到输出字符串中。**如前面的示例所示，比 \$ 字符的序列长的序列在输出中嵌入了额外的 { 或 } 字符。如果大括号字符序列等于或大于 \$ 字符序列长度的两倍，编译器将发出错误。

2) 特殊字符

要在内插字符串生成的文本中包含大括号 "{" 或 "}"，请使用两个大括号，即 "{{" 或 "}}"

因为冒号（“:”）**在内插表达式项中具有特殊含义**，为了在内插表达式中使用条件运算符，请**将表达式放在括号内**。

PS:

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{{}}});
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
//预计的输出是: He asked, "Is your name Horace?", but didn't wait for a reply :-{
//Horace is 34 years old
```

2. @ - 逐字字符串标识符

@ 特殊字符用作原义标识符，它具有一系列的用途。

1) @字符作为代码元素的前缀

使 C#关键字用作标识符。@ 符可作为代码元素的前缀，**编译器将把此代码元素解释为标识符而非 C#关键字**。下面的示例使用@字符定义其在 for 循环中使用的名为 for 的标识符。

PS:

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
    Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
```

```

}
// 这个实例显示的输出如下
//      Here is your gift, John!
//      Here is your gift, James!
//      Here is your gift, Joan!
//      Here is your gift, Jamie!

```

2)将原义解释为字符串

简单转义序列（如代表反斜杠的 `"\"`）、十六进制转义序列（如代表大写字母 A 的 `"\x0041"`）和 Unicode 转义序列（如代表大写字母 A 的 `"\u0041"`）都将按字面解释。只有引号转义序列（`"`）不会按字面解释；因为它生成一个双引号。此外，如果是**逐字内插字符串**，大括号转义序列（`{` 和 `}`）**不按字面解释**；它们**会生成单个大括号字符**。下面的示例分别使用常规字符串和原义字符串定义两个相同的文件路径，这是原义字符串的较常见用法之一。

PS:

```

string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

```

```

Console.WriteLine(filename1);
Console.WriteLine(filename2);
//这个示例的输出如下
//      c:\documents\files\u0066.txt
//      c:\documents\files\u0066.txt

```

3)使编译器在命名冲突的情况下区分两种属性

属性是派生自 `Attribute` 的类，其类型名称通常包含后缀 `Attribute`，但编译器不会强制进行此转换。随后可在代码中按其完整类型名称（例如 `[InfoAttribute]`）或短名称（例如 `[Info]`）引用此属性。但是，如果两个短名称相同，并且一个类型名称包含 `Attribute` 后缀而另一类型名称不包含，则会出现命名冲突。例如，由于编译器无法确定将 `Info` 还是 `InfoAttribute` 属性应用于 `Example` 类，因此下面的代码无法编译。

PS:

```

using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
    private string information;

    public Info(string info)
    {
        information = info;
    }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute

```

```
{
    private string information;

    public InfoAttribute(string info)
    {
        information = info;
    }
}
```

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.

// Prepend '@' to select 'Info' ([@Info("A simple executable.")]). Specify the full name 'InfoAttribute' to select it.

```
public class Example
{
    [InfoAttribute("The entry point.")]
    public static void Main()
    {
    }
}
```

三十九. C#预处理器指令

尽管编译器没有单独的预处理器,但这里所述指令的处理方式与有预处理器时一样。可使用这些指令来帮助条件编译。不同于 C 和 C++ 指令,我们不能使用这些指令来创建宏,预处理器指令必须是一行中唯一的说明。

1. 可为空上下文-#nullable

#nullable 预处理器指令将设置可为空注释上下文和可为空警告上下文 。 此指令控制是否可为空注释是否有效, 以及是否给出为 Null 性警告。 每个上下文要么处于已禁用状态, 要么处于已启用状态 。

可在项目级别 (C# 源代码之外) 指定这两个上下文。 #nullable 指令控制注释和警告上下文, 并优先于项目级设置。 指令会设置其控制的上下文, 直到另一个指令替代它, 或直到源文件结束为止。

#nullable disable	将可为空注释和警告上下文设置为“已禁用”。
#nullable enable	将可为空注释和警告上下文设置为“已启用”。
#nullable restore	将可为空注释和警告上下文还原为项目设置。
#nullable disable annotations	将可为空注释上下文设置为“已禁用”。
#nullable enable annotations	将可为空注释上下文设置为“已启用”。
#nullable restore annotations	将可为空注释上下文还原为项目设置。
#nullable disable warnings	将可为空警告上下文设置为“已启用”。
#nullable enable warnings	将可为空警告上下文设置为“已启用”。
#nullable restore warnings	将可为空警告上下文还原为项目设置。

2. 条件编译

使用四个预处理器指令来控制条件编译。

#if	打开条件编译, 其中仅在定义了指定的符号时
-----	-----------------------

	才会编译代码。
<code>#elif</code>	关闭前面的条件编译，并基于是否定义了指定的符号打开一个新的条件编译。
<code>#else</code>	关闭前面的条件编译，如果没有定义前面指定的符号，打开一个新的条件编译。
<code>#endif</code>	关闭前面的条件编译。

仅在定义指定的符号时或者在使用 **! not 运算符** 时未定义指定的符号时，C# 编译器才编译 `#if` 指令和 `#endif` 指令之间的代码。与 C 和 C++ 不同，不能将数字值分配给符号。C# 中的 `#if` 语句是布尔值，且仅测试是否已定义该符号。例如，定义 `DEBUG` 时将编译以下代码。

PS:

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
未定义 MYTEST 时，将编译以下代码。
```

PS:

```
#if !MYTEST
    Console.WriteLine("MYTEST is not defined");
#endif
```

可以使用运算符 `==`（相等）和 `!=`（不相等）来测试 `bool` 值是 `true` 还是 `false`。`true` 表示定义该符号。语句 `#if DEBUG` 具有与 `#if (DEBUG == true)` 相同的含义。可以使用 `&&` (and)、`||` (or) 和 `!` (not) 运算符来计算是否已定义多个符号。还可以用括号对符号和运算符进行分组。

`#if` 以及 `#else`、`#elif`、`#endif`、`#define` 和 `#undef` 指令，允许基于是否存在一个或多个符号包括或排除代码。条件编译在编译调试版本的代码或编译特定配置的代码时会很有用。

以 `#if` 指令开头的条件指令必须以 `#endif` 指令显式终止。`#define` 允许我们定义一个符号，通过将该符号用作传递给 `#if` 指令的表达式，该表达式的计算结果为 `true`。还可以通过 `DefineConstants` 编译器选项来定义符号，可以通过 `#undef` 取消定义符号。使用 `#define` 创建的符号的作用域是在其中定义它的文件。使用 `DefineConstants` 或 `#define` 定义的符号与具有相同名称的变量不冲突。也就是说，变量名称不应传递给预处理器指令，且符号仅能由预处理器指令评估。

`#elif` 可以创建复合条件指令。如果之前的 `#if` 和任何之前的可选 `#elif` 指令表达式的值都不为 `true`，则计算 `#elif` 表达式。如果 `#elif` 表达式计算结果为 `true`，编译器将计算 `#elif` 和下一条件指令间的所有代码。

PS:

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

`#else` 允许创建复合条件指令，因此，如果先前 `#if` 或（可选）`#elif` 指令中的任何表达式的计算结果都不是 `true`，则编译器将对介于 `#else` 和下一个 `#endif` 之间的所有代码进行

求值。**#endif(#endif)** 必须是 **#else** 之后的下一个预处理器指令。

#endif 指定条件指令的末尾，以 **#if** 指令开头。

其他预定义符号包括 **DEBUG** 和 **TRACE** 常数。 我们可以使用 **#define** 替代项目的值集。 例如，会根据生成配置属性（“调试”或者“发布”模式）自动设置 **DEBUG** 符号。

下例显示如何在文件上定义 **MYTEST** 符号，然后测试 **MYTEST** 和 **DEBUG** 符号的值。 此示例的输出取决于是在“调试”还是“发布”配置模式下生成项目。

PS:

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
        #if (DEBUG && !MYTEST)
            Console.WriteLine("DEBUG is defined");
        #elif (!DEBUG && MYTEST)
            Console.WriteLine("MYTEST is defined");
        #elif (DEBUG && MYTEST)
            Console.WriteLine("DEBUG and MYTEST are defined");
        #else
            Console.WriteLine("DEBUG and MYTEST are not defined");
        #endif
    }
}
```

3. 定义符号

使用以下两个预处理器指令来定义或取消定义条件编译的符号。

#define	定义符号。
#undef	取消定义符号。

使用 **#define** 来定义符号。 将符号用作传递给 **#if** 指令的表达式时，该表达式的计算结果为 **true**，如以下示例所示。

PS:

```
#define VERBOSE

#if VERBOSE
    Console.WriteLine("Verbose output version");
#endif
```

#define 指令不能用于声明常量值，这与 C 和 C++ 中的通常做法一样。 C# 中的常量最好定义为类或结构的静态成员。如果具有多个此类常量，请考虑创建一个单独的“常量”类来容纳它们。

符号可用于指定编译的条件。可通过 **#if** 或 **#elif** 测试符号。 还可以使用 **ConditionalAttribute** 来执行条件编译。**可以定义符号，但不能为符号分配值**。文件中必须先出现 **#define** 指令，才能使用并非同时也是预处理器指令的任何指示。还可以通过 **DefineConstants** 编译器选项来定义符号。可以通过 **#undef** 取消定义符号。

4. 定义区域

可以使用以下两个预处理器指令来定义可在大纲中折叠的代码区域。

<code>#region</code>	启动区域。
<code>#endregion</code>	结束区域。

利用 `#region`，可以指定在使用代码编辑器的大纲功能时可展开或折叠的代码块。在较长的代码文件中，折叠或隐藏一个或多个区域十分便利，这样，可将精力集中于当前处理的文件部分。下面的示例演示如何定义区域。

PS:

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

`#region` 块必须通过 `#endregion` 指令终止，同时 `#region` 块不能与 `#if` 块重叠。但是，可以将 `#region` 块嵌套在 `#if` 块内，或将 `#if` 块嵌套在 `#region` 块内。

5. 错误和警告信息

使用以下指令指示编译器生成用户定义的编译器错误和警告，并控制行信息。

<code>#error</code>	使用指定的消息生成编译器错误。
<code>#warning</code>	使用指定的消息生成编译器警告。
<code>#line</code>	更改用编译器消息输出的行号。

1) #error

`#error` 可从代码中的特定位置生成 CS1029 用户定义的错误。

PS:

```
#error Deprecated code in this method.
```

编译器以特殊的方式处理 `#error version` 并报告编译器错误 CS8304，消息中包含使用的编译器和语言版本。

2) #warning

`#warning` 允许我们从代码中的特定位置生成 CS1030 第一级编译器警告。

PS:

```
#warning Deprecated code in this method.
```

3) #line

借助 `#line`，可修改编译器的行号及（可选）用于错误和警告的文件名输出。

以下示例演示如何报告与行号相关联的两个警告。`#line 200` 指令将下一行的行号强制设为 200(尽管默认值为 #6);在执行下一个 `#line` 指令前，文件名都会报告为“特殊”。`#line default` 指令将行号恢复至默认行号，这会对上一指令重新编号的行进行计数。

PS:

```
class MainClass
{
    static void Main()
    {
```

```

#line 200 "Special"
    int i;
    int j;
#line default
    char c;
    float f;
#line hidden // numbering not affected
    string s;
    double d;
}
}

```

6. 杂注

1) #pragma

`#pragma` 为编译器给出特殊指令，以编译它所在的文件。这些指令必须受编译器支持！换句话说，**不能使用 `#pragma` 创建自定义的预处理指令。**

2) #pragma warning

`#pragma warning` 可以启用或禁用特定警告。

PS:

```
#pragma warning disable warning-list
```

```
#pragma warning restore warning-list
```

其中 `warning-list` 是以逗号分隔的警告编号的列表。“CS”前缀是可选的。未指定警告编号时，`disable` 会禁用所有警告，`restore` 会启用所有警告。

`disable` 从源文件的**下一行开始生效**。警告会在后面的 `restore` 行**上还原**。如果文件中**没有 `restore`**，则在同一编译中任何之后文件的第一行，警告将还原为其默认状态。

PS:

```
// pragma_warning.cs
```

```
using System;
```

```
#pragma warning disable 414, CS3021
```

```
[CLSCompliant(false)]
```

```
public class C
```

```
{
```

```
    int i = 1;
```

```
    static void Main()
```

```
    {
```

```
    }
```

```
}
```

```
#pragma warning restore CS3021
```

```
[CLSCompliant(false)] // CS3021
```

```
public class D
```

```
{
```

```
    int i = 1;
```

```
    public static void F()
```

```
    {
```



```

    }
}

```

四十. C#集合-Collection

对于许多应用程序，我们会想要创建和管理相关对象的组。这有两种方法可以对对象进行分组：**通过创建对象的数组，以及通过创建对象的集合。**

数组最适用于创建和使用**固定数量的强类型化对象**。

集合提供更灵活的方式来使用对象组。与数组不同，我们使用的对象组随着应用程序更改的需要动态地放大和缩小。对于某些集合，我们可以为放入集合中的任何对象分配一个密钥，这样便可以使用该密钥快速检索此对象。

集合是一个类，因此**必须在向该集合添加元素之前，声明类的实例**。

集合（Collection）类是**专门用于数据存储和检索的类**。这些类提供了对栈（stack）、队列（queue）、列表（list）和哈希表（hash table）的支持。大多数集合类实现了相同的接口。

集合（Collection）类服务于不同的目的，如为元素动态分配内存，基于索引访问列表项等等。这些类创建 Object 类的对象的集合。在 C# 中，**Object 类是所有数据类型的基类**。

如果**集合中只包含一种数据类型的元素**，则可以使用 `System.Collections.Generic` 命名空间中的一个类。泛型集合强制类型安全，因此无法向其添加任何其他数据类型。当我们从泛型集合检索元素时，我们无需确定其数据类型或对其进行转换。

1. 集合的类型

1) System.Collections.Generic 类

可以使用 `System.Collections.Generic` 命名空间中的某个类来**创建泛型集合**。当集合中的所有项都具有相同的数据类型时，泛型集合会非常有用。泛型集合通过仅允许添加所需的数据类型，强制实施强类型化。

下表列出了 `System.Collections.Generic` 命名空间中的一些常用类。

类	说明
<code>Dictionary<TKey,TValue></code>	表示基于键进行组织的键/值对的集合。
<code>List<T></code>	表示可按索引访问的对象的列表。提供用于对列表进行搜索、排序和修改的方法。
<code>Queue<T></code>	表示对象的先进先出 (FIFO) 集合，即队列。
<code>SortedList<TKey,TValue></code>	表示基于相关的 <code>IComparer<T></code> ，实现按“键”进行排序的键/值对的集合。
<code>Stack<T></code>	表示对象的后进先出 (LIFO) 集合，即栈。

2) System.Collections.Concurrent 类

在 .NET Framework 4 以及更新的版本中，`System.Collections.Concurrent` 命名空间中的集合可提供高效的线程安全操作，以便从多个线程访问集合项。

只要会有多个线程同时访问集合，就应使用 `System.Collections.Concurrent` 命名空间中的类，而不是 `System.Collections.Generic` 和 `System.Collections` 命名空间中的相应类型。

包含在 `System.Collections.Concurrent` 命名空间中的一些类为 `BlockingCollection<T>`、`ConcurrentDictionary<TKey,TValue>`、`ConcurrentQueue<T>` 和 `ConcurrentStack<T>`。

3) System.Collections 类

`System.Collections` 命名空间中的类不会将元素作为特别类型化的对象存储，而是**作为 Object 类型的对象存储**。

只要可能，则应使用 `System.Collections.Generic` 命名空间或 `System.Collections.Concurrent` 命名空间中的泛型集合，而不是 `System.Collections` 命名空间中的旧类型。

下表列出了 `System.Collections` 命名空间中的一些常用类：

类	说明
<code>ArrayList</code>	表示对象的数组，这些对象的大小会根据需要动态增加。
<code>Hashtable</code>	表示根据键的哈希代码进行组织的键/值对的集合。
<code>Queue</code>	表示对象的先进先出 (FIFO) 集合，即队列。
<code>Stack</code>	表示对象的后进先出 (LIFO) 集合，即栈。

`System.Collections.Specialized` 命名空间提供专门类型化以及强类型化的集合类，例如只包含字符串的集合以及链接列表和混合字典。

4) 如何选择集合类

请避免使用 `System.Collections` 命名空间中的类型。推荐使用泛型版本和并发版本的集合，因为它们的类型安全性很高，并且还经过了其他改进。

a. 是否需要顺序列表（其中通常在检索元素值后就将该元素丢弃）

在需要的情况下，如果需要先进先出 (FIFO) 行为，请考虑使用 `Queue` 类或 `Queue<T>` 泛型类。如果需要后进先出 (LIFO) 行为，请考虑使用 `Stack` 类或 `Stack<T>` 泛型类。若要从多个线程进行安全访问，请使用并发版本（`ConcurrentQueue<T>` 和 `ConcurrentStack<T>`）。如果要获得不可变性，请考虑不可变版本 `ImmutableQueue<T>` 和 `ImmutableStack<T>`。

如果不需要，请考虑使用其他集合。

b. 是否需要以特定顺序（如先进先出、后进先出或随机）访问元素

`Queue` 类以及 `Queue<T>`、`ConcurrentQueue<T>` 和 `ImmutableQueue<T>` 泛型类都提供 FIFO 访问权限。

`Stack` 类以及 `Stack<T>`、`ConcurrentStack<T>` 和 `ImmutableStack<T>` 泛型类都提供 LIFO 访问权限。

`LinkedList<T>` 泛型类允许从开头到末尾或从末尾到开头的顺序访问。

c. 是否需要按索引访问每个元素

`ArrayList` 和 `StringCollection` 类以及 `List<T>` 泛型类按从零开始的元素索引提供对其元素的访问。如果要获得不可变性，请考虑不可变泛型版本 `ImmutableArray<T>` 和 `ImmutableList<T>`。

`Hashtable`、`SortedList`、`ListDictionary` 和 `StringDictionary` 类以及 `Dictionary<TKey,TValue>` 和 `SortedDictionary<TKey,TValue>` 泛型类按元素的键提供对其元素的访问。此外，还有几个相应类型的不可变版本：`ImmutableHashSet<T>`、`ImmutableDictionary<TKey,TValue>`、`ImmutableSortedSet<T>` 和

² adj. 永恒的，不可改变的

ImmutableSortedDictionary<TKey,TValue>。

NameObjectCollectionBase 和 NameValueCollection 类以及 KeyedCollection<TKey,TItem> 和 SortedList<TKey,TValue> 泛型类按从零开始的元素索引或元素的键提供对其元素的访问。

d. 是否每个元素都包含一个值、一个键和一个值的组合或一个键和多个值的组合

i. 一个值

使用任何基于 IList 接口或 IList<T> 泛型接口的集合。要获得不可变选项，请考虑 ImmutableList<T> 泛型接口。

ii. 一个键和一个值

使用任何基于 IDictionary 接口或 IDictionary<TKey,TValue> 泛型接口的集合。要获得不可变选项，请考虑 ImmutableSet<T> 或 ImmutableDictionary<TKey,TValue> 泛型接口。

iii. 带有嵌入键的值

使用 KeyedCollection<TKey,TItem> 泛型类。

iv. 一个键和多个值

使用 NameValueCollection 类。

e. 是否需要以与输入方式不同的方式对元素进行排序

Hashtable 类按其哈希代码对其元素进行排序。

SortedList 类以及 SortedList<TKey,TValue> 和 SortedDictionary<TKey,TValue> 泛型类按键对元素进行排序。排序顺序的依据为，实现 SortedList 类的 IComparer 接口和实现 SortedList<TKey,TValue> 和 SortedDictionary<TKey,TValue> 泛型类的 IComparer<T> 泛型接口。在这两种泛型类型中，虽然 SortedDictionary<TKey,TValue> 的性能优于 SortedList<TKey,TValue>，但 SortedList<TKey,TValue> 占用的内存更少。

ArrayList 提供了一种 Sort 方法，此方法采用 IComparer 实现作为参数。其泛型对应项（List<T> 泛型类）提供一种 Sort 方法，此方法采用 IComparer<T> 泛型接口的实现作为参数。

f. 是否需要快速搜索和信息检索

对于小集合（10 项或更少），ListDictionary 速度比 Hashtable 快。Dictionary<TKey,TValue> 泛型类提供比 SortedDictionary<TKey,TValue> 泛型类更快的查找。

多线程的实现为 ConcurrentDictionary<TKey,TValue>。ConcurrentBag<T> 为无序数据

³ adj.并存的，同时发生的；同意的，一致的；（两个或两个以上徒刑判决）同时执行的；（三条或三条

提供快速的多线程插入。有关这两种多线程类型的详细信息，请参阅何时使用线程安全集合。

g. 是否需要只接受字符串的集合

StringCollection（基于 IList）和 StringDictionary（基于 IDictionary）位于 System.Collections.Specialized 命名空间。

此外，通过指定其泛型类参数的 String 类，可以使用 System.Collections.Generic 命名空间中的任何泛型集合类作为强类型字符串集合。例如，可以将变量声明为采用 List<String> 或 Dictionary<String,String> 类型。

2. 常用的集合类型

集合类型表示收集数据的不同方式，例如哈希表、队列、堆栈、包、字典和列表。

所有集合都直接或间接基于 ICollection<T> 或 ICollection 接口。IList 和 IDictionary 及其泛型对应项均派生自这两个接口。

在基于 IList 或直接基于 ICollection 的集合中，每个元素都只包含一个值。

这些类型包括：Array、ArrayList、List<T>、Queue、ConcurrentQueue<T>、Stack、ConcurrentStack<T>、LinkedList<T>。

在基于 IDictionary 接口的集合中，每个元素都只包含一个键和一个值。

这些类型包括：Hashtable、SortedList、SortedList<TKey,TValue>、Dictionary<TKey,TValue>、ConcurrentDictionary<TKey,TValue>。

KeyedCollection<TKey,TItem> 类是唯一的，因为它是值中嵌键的值的列表。因此，它的行为类似列表和字典。

需要高效的多线程集合访问时，请使用 System.Collections.Concurrent 命名空间中的泛型集合。

Queue 和 Queue<T> 类提供的是先进先出列表。

Stack 和 Stack<T> 类提供的是后进先出列表。

1) 强类型化

泛型集合都是强类型的最佳解决方案。

例如，将除 Int32 外的任何类型的元素添加到集合 List<Int32> 会导致编译时错误。但是，如果你的语言不支持泛型，那么 System.Collections 命名空间包含抽象基类，可扩展以创建强类型集合类。

这些基类包括：CollectionBase、ReadOnlyCollectionBase、DictionaryBase。

2) 集合之间存在哪些不同

集合在存储、排序和比较元素以及执行搜索的方式方面有所不同。

SortedList 类和 SortedList<TKey,TValue> 泛型类提供 Hashtable 类和 Dictionary<TKey,TValue> 泛型类的已排序版本。

所有集合中使用的索引都从零开始，Array 除外，它是允许不从零开始的数组。

可以通过键或元素的索引访问 SortedList 或 KeyedCollection<TKey,TItem> 的元素。

只能通过元素的键访问 Hashtable 或 Dictionary<TKey,TValue> 的元素。

3. ArrayList—动态数组

它代表了可被单独索引的对象的有序集合。

它基本上可以替代一个数组。但是，与数组不同的是，它可以使用索引在指定的位置添加和移除元素，动态数组会自动重新调整它的大小。它也允许在列表中进行动态内存分配、增加、搜索、排序各项。

ArrayList 继承自 Object。它实现了 ICollection、IEnumerable、IList、ICollection 等接口。

1) 示例

以下示例演示如何创建和初始化其 ArrayList 值以及如何显示其值。

PS:

```
using System;
using System.Collections;
public class SamplesArrayList {

    public static void Main() {

        // 创建一个新的 ArrayList 实例
        ArrayList myAL = new ArrayList();
        myAL.Add("Hello");
        myAL.Add("World");
        myAL.Add("!");

        // 显示 ArrayList 中的属性与其值
        Console.WriteLine( "myAL" );
        Console.WriteLine( "    Count:    {0}", myAL.Count );
        Console.WriteLine( "    Capacity: {0}", myAL.Capacity );
        Console.Write( "    Values:" );
        PrintValues( myAL );
    }

    public static void PrintValues( IEnumerable myList ) {
        foreach ( Object obj in myList )
            Console.Write( "    {0}", obj );
        Console.WriteLine();
    }
}
```

/*

This code produces output similar to the following:

```
myAL
Count:    3
Capacity: 4
Values:   Hello    World    !
```

*/

2) 注意事项

不建议将 `ArrayList` 用于新开发。相反，我们建议使用泛型 `List<T>` 类。`ArrayList` 类旨在保存对象的异类集合。但是，它并不总是提供最佳性能。相反，建议执行以下操作：

对于对象的同质集合，请使用 `List<T>` 类。

ArrayList 不能保证排序。在执行需要排序的操作（如需要排序的）之前，必须通过调用其 `Sort` 方法将 `ArrayList` 进行排序。`ArrayList` `BinarySearch` 若要维护在添加新元素时自动排序的集合，可以使用 `SortedSet<T>` 类。

`ArrayList` 的容量就是存储在 `ArrayList` 中的元素数量。如果某个元素被添加到 `ArrayList` 中，将通过重新分配自动增加其容量。我们可以通过调用 `TrimToSize` 或显式设置 `Capacity` 属性来减少容量。

仅.NET Framework：对于在 `ArrayList` 中存储非常大量的对象，可以通过将配置元素的属性 `<gcAllowVeryLargeObjects>` 设置为 `enabled`true` 运行时环境中，将 64 位系统上的最大容量增加到 20 亿个元素。

可以使用整数索引访问此集合中的元素。此集合中的索引从零开始。

集合 `ArrayList` 接受 `null` 为有效值。它还允许重复的元素。

不支持将多维数组用作集合中的 `ArrayList` 元素。

3) 构造函数

<code>ArrayList()</code>	初始化 <code>ArrayList</code> 类的新实例，该实例为空并且具有默认初始容量。
<code>ArrayList(ICollection)</code>	初始化 <code>ArrayList</code> 类的新实例，该类包含从指定集合复制的元素，并具有与复制的元素数相同的初始容量。
<code>ArrayList(Int32)</code>	初始化 <code>ArrayList</code> 类的新实例，该实例为空并且具有指定的初始容量。

4) 属性

<code>Capacity</code>	获取或设置 <code>ArrayList</code> 可包含的元素数。（即最大容量）
<code>Count</code>	获取 <code>ArrayList</code> 中实际包含的元素数。
<code>IsFixedSize</code>	获取一个值，该值指示 <code>ArrayList</code> 是否具有固定大小。
<code>IsReadOnly</code>	获取一个值，该值指示 <code>ArrayList</code> 是否为只读。
<code>IsSynchronized</code>	获取一个值，该值指示是否同步对 <code>ArrayList</code> 的访问（线程安全）。
<code>Item[Int32]</code>	获取或设置指定索引处的元素。
<code>SyncRoot</code>	

	获取可用于同步对 ArrayList 的访问的对象。
--	----------------------------

5)方法

Add(Object)	将对象添加到 ArrayList 的结尾处。
AddRange(ICollection)	将 ICollection 的元素添加到 ArrayList 的末尾。
Clear()	从 ArrayList 中移除所有元素。
Clone()	创建 ArrayList 的浅表副本。
Contains(Object)	确定某元素是否在 ArrayList 中。
CopyTo(Array)	从目标数组的开头开始, 将整个 ArrayList 复制到兼容的一维 Array。
Equals(Object)	确定指定对象是否等于当前对象。 (继承自 Object)

四十一. HashTable

1. 什么是哈希表

散列表 (Hash table, 也叫哈希表), 是根据关键码值(Key value)而直接进行访问的数据结构。也就是说, 它通过把关键码值映射到表中一个位置来访问记录, 以加快查找的速度。这个映射函数叫做散列函数, 存放记录的数组叫做散列表。

给定表 M, 存在函数 f(key), 对任意给定的关键字值 key, 代入函数后若能得到包含该关键字的记录在表中的地址, 则称表 M 为哈希(Hash) 表, 函数 f(key)为哈希(Hash) 函数。

- ① 哈希表是一种数据结构。
- ② 哈希表表示了关键码值和记录的映射关系。
- ③ 哈希表可以加快查找速度。
- ④ 任意哈希表, 都满足有哈希函数 f(key), 代入任意 key 值都可以获取包含该 key 值的记录在表中的地址。

2. 什么是 Hash

Hash, 一般翻译做“散列”, 也有直接音译为“哈希”的, 就是把任意长度的输入 (又叫做预映射), 通过散列算法, 变换成固定长度的输出, 该输出就是散列值。这种转换是一种压缩映射, 也就是, 散列值的空间通常远小于输入的空间, 不同的输入可能会散列成相同的输出, 而不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

3. 哈希表有什么优势

哈希表的优点呼之欲出: 通过关键值计算直接获取目标位置, 对于海量数据中的精确查

找有非常惊人的速度提升，理论上即使有无限的数据量，一个实现良好的哈希表依旧可以保持 $O(1)$ 的查找速度，而 $O(n)$ 的普通列表此时已经无法正常执行查找操作。

4. 哈希冲突

根据 key（键）即经过一个函数 $f(key)$ 得到的结果的作为地址去存放当前的 key value 键值对（这个是 hashmap 的存值方式），但是却发现算出来的地址上已经被占用了。这就是所谓的 hash 冲突。

5. 解决哈希冲突

常用的解决方案有散列法和拉链法。

需要注意的是，如果遭到恶意哈希碰撞攻击，拉链法会导致哈希表退化为链表，即所有元素都被存储在同一个节点的链表中，此时哈希表的查找速度=链表遍历查找速度= $O(n)$ 。

1) 开放定址法

该方法也叫做再散列法，其基本原理是：当关键字 key 的哈希地址 $p=H(key)$ 出现冲突时，以 p 为基础，产生另一个哈希地址 p1，如果 p1 仍然冲突，再以 p 为基础，产生另一个哈希地址 p2，…，直到找出一个不冲突的哈希地址 p_i 。

a. 线性探测

发生 hash 冲突时，顺序查找下一个位置，直到找到一个空位置（固定步长 1 探测）。

b. 平方探测

在发生 hash 冲突时，在表的左右位置进行按一定步长跳跃式探测（固定步长 n 探测）。

2) 再 Hash 法

这种方法就是同时构造多个不同的哈希函数： $H_i = RH1(key) \quad i=1, 2, \dots, k$ 。当哈希地址 $H_i = RH1(key)$ 发生冲突时，再计算 $H_i = RH2(key) \dots$ ，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

3) 链地址法

将所有哈希地址为 i 的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第 i 个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

4) 建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

四十二. List 和 ArrayList 的区别

说到这两个的差别我们想到的首先就应该是这两个集合的性质有相对较大的区别，List 就像是一个火车，ArrayList 就像是自己家的小轿车，List 对于装载的事物有限制，一开始必须定义好类型，就是拉货物的火车车厢中都几乎是某种货物。

1. ArrayList

而小汽车是自家的你想拉啥就拉啥，ArrayList 就充当小轿车的角色，但是其相对速度对比火车较差。ArrayList 是命名空间 System.Collections 下的一部分，在使用该类时必须进行引用，同时继承了 IList 接口，提供了数据存储和检索。ArrayList 对象的大小是按照其中存

储的数据来动态扩充与收缩的。所以，在声明 ArrayList 对象时并不需要指定它的长度。

在 ArrayList 中插入不同类型的数据是允许的，因为 ArrayList 会把所有插入其中的数据都当作为 object 类型来处理。这样，在我们使用 ArrayList 中的数据来处理问题的时候，很可能会报类型不匹配的错误，也就是说 ArrayList 不是类型安全的。既使我们保证在插入数据的时候都很小心，都有插入了同一类型的数据，但在使用的时候，我们也需要将它们转化为对应的原类型来处理。

由于 ArrayList 的每一次获得数据，都存在装箱与拆箱的操作。每一次的装箱拆箱都涉及 CPU 以及内存的分配，这会带来很大的性能损耗。

PS:

```
//1.首先创建对象
ArrayList arr=new ArrayList();
//使用 Add()方法添加元素,对元素类型没有限制
arr.Add(12);
arr.Add("1234");
arr.Add(12.7f);
//使用 /下标/ 来获取指定位置的元素
Console.WriteLine ("arr[0]="+arr[0]);
//获取当前数组的数量
int count=arr.Count;
//使用 insert()方法向指定下标位置插入元素
arr.Insert(1,"老张");
Console.WriteLine ("arr[1]="+arr[1]);
//使用 Remove()方法从数组中删除指定元素
arr.Remove("老张");
Console.WriteLine ("arr[1]="+arr[1]);
//使用 RemoveAt()方法,将指定下标位置的元素删除
arr.RemoveAt(0);
Log (arr);
//判断指定元素中是否存在当前数组中
bool b=arr.Contains("老王");
if (b)
{
    Console.WriteLine ("老王在数组中");
}
else
{
    Console.WriteLine ("老王不在数组中!!!!");
}
```

2. List

List 类是 ArrayList 类的泛型等效类。它的大部分用法都与 ArrayList 相似，因为 List 类也继承了 IList 接口。最关键的区别在于，在声明 List 集合时，我们必须同时为其声明 List 集合内数据的对象类型。

PS:

```
class Program
```

```

    {
        static void Main(string[] args)
        {
            List<string> list = new List<string>();
            //添加数据
            list.Add("书山有路勤为径");
            list.Add("学海无涯苦作舟");
            //list.Add(123);//如果向 list 中添加整型会报错,避免了类型安全问题与装箱
拆箱的性能问题了。
            Console.WriteLine($"{list[0]}, {list[1]}!");//输出书山有路勤为径,学海无涯
苦作舟!

            //修改数据
            list[0] = "乘风破浪会有时";
            list[1] = "直挂云帆济沧海";
            Console.WriteLine($"{list[0]}, {list[1]}!");//输出乘风破浪会有时,直挂云帆
济沧海!

            //删除数据
            list.RemoveAt(0);
            Console.WriteLine($"{list[0]}");//输出直挂云帆济沧海
            //插入数据
            list.Insert(0, "乘风破浪会有时");
            Console.WriteLine($"{list[0]}, {list[1]}!");//输出乘风破浪会有时,直挂云帆
济沧海!

            Console.ReadLine();
        }
    }
}

```

综上所述！在编程中，我们对于 `ArrayList` 尽量能不用就不用，由于他什么东西都放，所以往往会导致，我们无法用特定的类型进行接收，因为其取出时 `object` 类型需要强转为我们需要的类型。

3. Array、ArrayList 和 List 的区别总结

1) 相同点

`Array`、`ArrayList` 和 `List` 都是从 `ICollection` 派生出来的，它们都实现了 `IEnumerable` 接口。

2) 不同点

数组的容量是固定的，只能一次获取或设置一个元素的值，而 `ArrayList` 或 `List` 的容量可根据需要自动扩充、修改、删除或插入数据。

数组可以是一维数组、二维数组和多维数组，而 `ArrayList` 或 `List` 始终只具有一个维度。但是，可以轻松创建数组列表或列表的列表。特定类型（`Object` 除外）的数组的性能优于 `ArrayList` 的性能。这是因为 `ArrayList` 的元素属于 `Object` 类型；所以在存储或检索值类型时通常发生装箱和拆箱操作。不过，在不需要重新分配时（即最初的容量十分接近列表的最大容量），`List<T>` 的性能与同类型的数组十分相近。

在决定使用 `List` 类还是使用 `ArrayList` 类（两者具有类似的功能）时，记住 `List<T>` 类在大多数情况下执行得更好并且是类型安全的。如果对 `List` 类的类型使用引用类型，则两个类的行为是完全相同的。但是，如果类型 `T` 是值类型，则需要考虑实现装箱和拆箱问题。

List 属于泛型集合 ArrayList 属于非泛型集合。

四十三. C#LINQ

语言集成查询 (LINQ) 是一系列直接将查询功能集成到 C# 语言的技术统称。数据查询历来都表示为简单的字符串，没有编译时类型检查或 IntelliSense 支持。此外，需要针对每种类型的数据源了解不同的查询语言：SQL 数据库、XML 文档、各种 Web 服务等。借助 LINQ，查询成为了最高级的语言构造，就像类、方法和事件一样。可以使用语言关键字和熟悉的运算符针对强类型化对象集合编写查询。

1. LINQ 查询

所有 LINQ 查询操作都由以下三个不同的操作组成：

- ① 获取数据源
- ② 创建查询
- ③ 执行查询

下面的示例演示如何用源代码表示查询操作的三个部分。

PS：

```
class IntroToLINQ
{
    static void Main()
    {
        // 1. 获取数据源
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. 创建查询
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. 执行查询
        foreach (int num in numQuery)
        {
            Console.WriteLine("{0,1} ", num);
        }
    }
}
```

2. 查询关键字

查询表达式必须以 from 子句开头，以 select 或 group 子句结束。在这两个子句之间，可以使用 where、orderby、join、let 和其他 from 子句。

子句	说明
from	指定数据源和范围变量（类似于迭代变量）
where	基于由逻辑 AND 和 OR 运算符(&& 或) 分隔的一个或多个布尔表达式筛选源元素

select	指定执行查询时，所返回序列中元素的类型和形状
group	根据指定的密钥值对查询结果分组
into	提供可作为对 join、group 或 select 子句结果引用的标识符
orderby	根据元素类型的默认比较器对查询结果进行升序或降序排序
join	基于两个指定匹配条件间的相等比较而联接两个数据源
let	引入范围变量，在查询表达式中存储子表达式结果
in	join 子句中的上下文关键字
on	
equals	
by	group 子句中的上下文关键字
ascending	orderby 子句中的上下文关键字
descending	

1) From

查询表达式必须以 **from** 子句开头。此外，查询表达式可包含也以 **from** 子句开头的子查询。

from 子句指定下列各项：

- ① 将在其上运行查询或子查询的数据源。
- ② 表示源序列中每个元素的本地范围变量。

范围变量和数据源已强类型化。**from** 子句中引用的数据源必须具有 **IEnumerable**、**IEnumerable<T>** 类型之一，或 **IQueryable<T>** 等派生类型。

PS:

```
class LowNums
{
    static void Main()
    {
        // 一个简单的数据源
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // 创建查询
        // lowNums 是一个 IEnumerable<int>
        var lowNums = from num in numbers
                      where num < 5
                      select num;

        // 执行查询
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
```

```

    }
}
}
// Output: 4 1 3 2 0

```

a. 复合 from 子句

在某些情况下，源序列中的每个元素可能本身就是一个序列，或者包含一个序列。例如，数据源可能是 `IEnumerable<Student>`，其中序列中的每个学生对象都包含测试分数的列表。要访问每个 `Student` 元素的内部列表，可以使用复合 `from` 子句。这种方法类似于使用嵌套的 `foreach` 语句。可以向任一 `from` 子句添加 `where` 或 `orderby` 子句筛选结果。下面的示例演示 `Student` 对象的序列，其中每个对象都包含一个整数内部 `List`，表示测验分数。访问内部列表可使用复合 `from` 子句。如有必要，可以在这两个 `from` 子句间插入子句。

PS:

```

class CompoundFrom
{
    // 元素的类型为数据源
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores { get; set; }
    }

    static void Main()
    {
        // 使用集合初始值设定项创建数据源。请注意，列表中的每个元素都包含一个内部分数序列。
        List<Student> students = new List<Student>
        {
            new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81,
60}},
            new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91,
39}},
            new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65,
85}},
            new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
            new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // 使用复合 from 访问每个元素中的内部序列。
        // 请注意其与嵌套的 foreach 语句的相似性。
        var scoreQuery = from student in students
                        from score in student.Scores

```

```

where score > 90
select new { Last = student.LastName, score };

```

```

//执行查询
Console.WriteLine("scoreQuery:");

foreach (var student in scoreQuery)
{
    Console.WriteLine("{0} Score: {1}", student.Last, student.score);
}

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/

```

b. 使用多个 from 子句执行联接

复合 from 子句用于访问单个数据源中的内部集合。但是,查询也可以包含多个 from 子句,这些子句从独立的数据源生成补充查询。通过此方法,可以执行使用 join 子句无法实现的某类联接操作。

2)Where

where 子句用在查询表达式中,用于指定将在查询表达式中返回数据源中的哪些元素。它将一个布尔条件(谓词)应用于每个源元素(由范围变量引用),并返回满足指定条件的元素。一个查询表达式可以包含多个 where 子句,一个子句可以包含多个谓词子表达式。

a. 案例一

在下面的示例中,where 子句筛选出除小于五的数字外的所有数字。如果删除 where 子句,则会返回数据源中的所有数字。表达式 `num < 5` 是应用于每个元素的谓词。

PS:

```

class WhereSample
{
    static void Main()
    {

```

```
//简单数据源。数组支持 IEnumerable<T>.  
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
  
// 在 where 子句中包含一个谓词的简单查询  
var queryLowNums =  
    from num in numbers  
    where num < 5  
    select num;  
  
// 执行查询  
foreach (var s in queryLowNums)  
{  
    Console.Write(s.ToString() + " ");  
}  
}  
//Output: 4 1 3 2 0
```

b. 案例二

在单个 **where** 子句中，可使用 **&&** 和 **||** 运算符根据需要指定任意多个谓词。在下面的示例中，查询将指定两个谓词，以便只选择小于五的偶数。

PS:

```
class WhereSample2  
{  
    static void Main()  
    {  
        //数据源  
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
  
        // 在 where 子句中创建具有两个谓词的查询  
        var queryLowNums2 =  
            from num in numbers  
            where num < 5 && num % 2 == 0  
            select num;  
  
        // 执行查询  
        foreach (var s in queryLowNums2)  
        {  
            Console.Write(s.ToString() + " ");  
        }  
        Console.WriteLine();  
  
        // 使用两个 where 子句创建查询。
```

```
var queryLowNums3 =
    from num in numbers
    where num < 5
    where num % 2 == 0
    select num;

// 执行查询
foreach (var s in queryLowNums3)
{
    Console.Write(s.ToString() + " ");
}
}
// Output:
// 4 2 0
// 4 2 0
```

c. 案例三

一个 `where` 子句可以包含一个或多个返回布尔值的方法。在下面的示例中，`where` 子句使用一种方法来确定范围变量的当前值是偶数还是奇数。

PS:

```
class WhereSample3
{
    static void Main()
    {
        // 数据源
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

        // 执行查询
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // 方法可以是实例方法或静态方法。
    static bool IsEven(int i)
    {
```



```

        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0

```

where 子句是一种筛选机制。除了不能是第一个或最后一个子句外，它几乎可以放在查询表达式中的任何位置。**where** 子句可以出现在 **group** 子句的前面或后面，具体取决于时必须在对源元素进行分组之前还是之后来筛选源元素。

如果指定的谓词对于数据源中的元素无效，则会发生编译时错误。这是 LINQ 提供的强类型检查的一个优点。

在编译时，**where** 关键字将转换为对 **Where** 标准查询运算符方法的调用。

3)Select

在查询表达式中，**select** 子句指定在执行查询时产生的值的类型。根据计算所有以前的子句以及根据 **select** 子句本身的所有表达式得出结果。查询表达式必须以 **select** 子句或 **group** 子句结尾。

PS:

```

class SelectSample1
{
    static void Main()
    {
        //创建数据源
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // 创建查询
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        //执行查询
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81

```

select 子句生成的序列的类型确定查询变量 **queryHighScores** 的类型。在最简单的情况下，**select** 子句仅指定范围变量。这将导致返回的序列包含与数据源类型相同的元素。

4)Group

group 子句返回一个 **IGrouping<TKey,TElement>** 对象序列，这些对象包含零个或多个与该组的键值匹配的项。例如，可以按照每个字符串中的第一个字母对字符串序列进行分组。在这种情况下，第一个字母就是键，类型为 **char**，并且存储在每个 **IGrouping<TKey,TElement>** 对象的 **Key** 属性中。

a. 用 group 子句结束查询表达式

PS:

```
// 查询变量是一个 IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

b. 使用 into

如果要对每个组执行附加查询操作，可使用上下文关键字 `into` 指定一个临时标识符。使用 `into` 时，必须继续编写该查询，并最终使用一个 `select` 语句或另一个 `group` 子句结束该查询。

PS:

```
// 按姓氏的第一个字母对学生进行分组
// 查询变量是一个 IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

5) Orderby

在查询表达式中，`orderby` 子句可导致返回的序列或子序列（组）以升序或降序排序。若要执行一个或多个次级排序操作，可以指定多个键。元素类型的默认比较器执行排序。默认排序顺序为升序，还可以指定自定义比较器。但是，只适用于使用基于方法的语法。

在以下示例中，第一个查询按字母顺序从 A 开始对字词排序，而第二个查询则按降序对相同的字词排序。

PS:

```
class OrderbySample1
{
    static void Main()
    {
        // 创建美食的数据源。
        string[] fruits = { "cherry", "apple", "blueberry" };

        // 查询结果升序排列。
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        //查询结果降序排列
        IEnumerable<string> sortDescendingQuery =
```

```

        from w in fruits
        orderby w descending
        select w;

//执行查询
Console.WriteLine("Ascending:");
foreach (string s in sortAscendingQuery)
{
    Console.WriteLine(s);
}

//执行查询
Console.WriteLine(Environment.NewLine + "Descending:");
foreach (string s in sortDescendingQuery)
{
    Console.WriteLine(s);
}

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
    }
}

/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/

```

6)Join

join 子句可用于将来自不同源序列并且在对象模型中没有直接关系的元素相关联。唯一的要求是每个源中的元素需要共享某个可以进行比较以判断是否相等的值。例如，食品经销商可能拥有某种产品的供应商列表以及买主列表。例如，可以使用 join 子句创建该产品同一指定地区供应商和买主的列表。

join 子句将 2 个源序列作为输入。每个序列中的元素都必须是可以与其他序列中的相应属性进行比较的属性，或者包含一个这样的属性。join 子句使用特殊 equals 关键字比较指定的键是否相等。join 子句执行的所有联接都是同等联接，其输出形式取决于执行的联接的具体类型。

a. 内部联接

以下示例演示了一个简单的内部同等联接。此查询生成一个“产品名称/类别”对平面序列，同一类别字符串将出现在多个元素中。如果 `categories` 中的某个元素不具有匹配的 `products`，则该类别不会出现在结果中。

PS:

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat
sequence
```

b. 分组联接

含有 `into` 表达式的 `join` 子句称为分组联接。

PS:

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

分组联接会生成分层的结果序列，该序列将左侧源序列中的元素与右侧源序列中的一个或多个匹配元素相关联。**分组联接没有等效的关系术语；它本质上是一个对象数组序列。**

如果在右侧源序列中找不到与左侧源中的元素相匹配的元素，则 `join` 子句会为该组生成一个空数组。因此，分组联接基本上仍然是一种内部同等联接，区别在于**分组联接将结果序列组织为多个组。**

如果只选择分组联接的结果，则可访问各项，但无法识别结果所匹配的项。因此，通常更为有用的做法是：选择分组联接的结果并将其放入一个也包含该项名的新类型中，如上例所示。

当然，还可以将分组联接的结果用作其他子查询的生成器：

PS:

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

c. 左外部联接

在左外部联接中，将返回左侧源序列中的所有元素，即使右侧序列中没有其匹配元素也是如此。若要在 LINQ 中执行左外部联接，请结合使用 `DefaultIfEmpty` 方法与分组联接，指定要在某个左侧元素不具有匹配元素时生成的默认右侧元素。可以使用 `null` 作为任何引用类型的默认值，也可以指定用户定义的默认类型。以下示例演示了用户定义的默认类

型:

PS:

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty,
CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

四十四. Lambda 表达式

"Lambda 表达式"是一个**匿名函数**，是一种高效的类似于函数式编程的表达式。使用 Lambda 简化了匿名委托的使用，减少开发中需要编写的代码量。

使用 `lambda` **声明运算符**`=>` 从其主体中分离 `lambda` 参数列表。Lambda 表达式可采用以下任意一种形式。

若要创建 Lambda 表达式，需要在 Lambda 运算符**左侧指定输入参数**（如果有），然后在另一侧输入表达式或语句块。

任何 Lambda 表达式都可以转换为委托类型。Lambda 表达式可以转换的委托类型由其参数和返回值的类型定义。如果 `lambda` 表达式**不返回值**，则可以将其转换为 **Action 委托类型**之一；否则，可将其转换为 **Func** 委托类型之一。例如，有 2 个参数且不返回值的 Lambda 表达式可转换为 `Action<T1,T2>` 委托。有 1 个参数且不返回值的 Lambda 表达式可转换为 `Func<T,TResult>` 委托。下面的示例中，`lambda` 表达式 `x => x * x`（指定名为 `x` 的参数并返回 `x` 平方值）将分配给委托类型的变量。

PS:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:25
```

表达式 `lambda` 还可以转换为表达式树类型，如下面的示例所示。

PS:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output: x => (x * x)
```

可在需要委托类型或表达式树的实例的任何代码中使用 `lambda` 表达式。不过有两点需要**特别注意**！

- ① 如果要将“Lambda 表达式”转换为委托类型，那么“Lambda 表达式”的**参数数量必须和“委托”的参数数量相同**。
- ② 如果“委托”的参数中包括有 `ref` 或 `out` 修饰符，则“Lambda 表达式”的参数列中也**必须包括有修饰符**。

1. 表达式 Lambda

表达式位于 `=>` 运算符右侧的 `lambda` 表达式称为“表达式 `lambda`”。表达式 `lambda` 会返回表达式的结果，并采用以下基本形式。

PS:

```
//表达式为其主体
(input-parameters) => expression
```

2. 语句 Lambda

语句 lambda 与表达式 lambda 类似，只是语句括在大括号中。

PS:

//语句块作为其主体

```
(input-parameters) => { <sequence-of-statements> }
```

语句 lambda 的主体可以包含任意数量的语句；但是，实际上通常不会多于两个或三个。

PS:

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output: Hello World!
```

3. Lambda 表达式的输入参数

将 lambda 表达式的输入参数括在括号中，使用空括号指定零个输入参数。

PS:

```
Action line = () => Console.WriteLine();
```

如果 lambda 表达式只有一个输入参数，则括号是可选的。

PS:

```
Func<double, double> cube = x => x * x * x;
```

两个或更多输入参数使用逗号加以分隔。

PS:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

有时，编译器无法推断输入参数的类型。我们可以显式指定类型，如下面的示例所示。

PS:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

输入参数类型必须全部为显式或全部为隐式；否则，便会生成 CS0748 编译器错误。

从 C# 9.0 开始，可以使用弃元指定 lambda 表达式中不使用的两个或更多输入参数。

PS:

```
Func<int, int, int> constant = (_, _) => 42;
```

如果只有一个输入参数命名为 _，则在 lambda 表达式中，_ 将被视为该参数的名称。

4. 捕获 Lambda 表达式中的外部变量

lambda 可以引用外部变量，由 lambda 表达式引用的外部变量称为捕获变量。这些变量是在定义 lambda 表达式的方法中或包含 lambda 表达式的类型中的范围内变量。以这种方式捕获的变量将进行存储以备在 lambda 表达式中使用，即便在其他的情况下，这些变量将超出范围并可以进行垃圾回收。我们必须明确地分配外部变量，然后才能在 lambda 表达式中使用该变量。下面的示例演示这些规则。

lambda 表达式引用定义它的方法的局部变量和参数。

PS:

```
static void Main(){
    int factor = 2;
    Func<int, int> multiplier = n => n * factor;
    Console.WriteLine (multiplier (3)); // 6
}
```

Lambda 表达式可以更新捕获的变量。

PS:

```
int outerVariable = 0;
Func<int> myLambda = () => outerVariable++;
Console.WriteLine (myLambda()); // 0
Console.WriteLine (myLambda()); // 1
Console.WriteLine (outerVariable); // 2
```

PS:

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int>? updateCapturedLocalVariable;
        internal Func<int, bool>? isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{{j}} is greater than {{input}}: {{result}}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {{j}}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {{j}}");
        }
    }

    public static void Main()
```

```

{
    var game = new VariableCaptureGame();

    int gameInput = 5;
    game.Run(gameInput);

    int jTry = 10;
    bool result = game.isEqualToCapturedLocalVariable!(jTry);
    Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

    int anotherJ = 3;
    game.updateCapturedLocalVariable!(anotherJ);

    bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
    Console.WriteLine($"Another lambda observes a new value of captured variable:
{equalToAnother}");
}
// Output:
// Local variable before lambda invocation: 0
// 10 is greater than 5: True
// Local variable after lambda invocation: 10
// Captured local variable is equal to 10: True
// 3 is greater than 5: False
// Another lambda observes a new value of captured variable: True
}

```

5. Lambda 表达式中的变量范围

- ① Lambda 表达式捕获的变量将不会被作为垃圾回收，直至引用变量的委托超出范围为止。
- ② 在外部方法中看不到 Lambda 表达式内引入的变量。
- ③ Lambda 表达式无法从封闭方法中直接捕获 `ref` 或 `out` 参数。
- ④ Lambda 表达式中的返回语句不会导致封闭方法返回。
- ⑤ Lambda 表达式不能包含其目标位于所包含匿名函数主体外部或内部的 `goto` 语句、`break` 语句或 `continue` 语句。
- ⑥ Lambda 表达式的本质是“匿名方法”，即当编译我们的程序代码时，“编译器”会自动将“Lambda 表达式”转换为“匿名方法”。

四十五. 正则表达式

正则表达式作为小型技术领域的一部分在各种程序中都有着难以置信的作用，但开发人员几乎很少会去使用它。正则表达式可以看作一种有特定功能的小型编程语言：**在大的字符串表达式中定位一个子字符串**，它并不是一种新技术。

正则表达式提供了功能强大、灵活而又高效的方法来处理文本。正则表达式丰富的泛模式匹配表示法使我们可以**快速分析大量文本**，以便查找特定字符模式；验证文本以确保它匹配预定义模式（如电子邮件地址）；**提取、编辑、替换或删除**文本子字符串；将提取的字符串**添加到集合**中，以便生成报告。

对于处理字符串或分析大文本块的许多应用程序而言，正则表达式是不可缺少的工具。

1. 字符转义

正则表达式中的反斜线 (\)后接字符为特殊字符，如下面的表中所示。例如，\b 是定位标记，用于指示正则表达式的匹配应从单词边界开始，\t 表示制表符，而 \x020 表示空格。

本应解释为未转义语言构造的字符应按字面意思进行解释。 例如，大括号 ({} 开始定义限定符，而反斜杠后接大括号 (\{) 表示正则表达式引擎应匹配大括号。 同样，单个反斜杠标记转义的语言构造的开始，而两个反斜杠 (\\) 表示正则表达式引擎应匹配反斜杠。

字符或序列	说明
除以下字符外的所有字符： . \$ ^ { [() * + ? \ 	“字符或序列”列中未包含的字符在正则表达式中没有特殊含义；此类字符与自身匹配。 “字符或序列”列中包括的字符均为特殊的正则表达式语言元素。 若要在正则表达式中匹配这些字符，必须将其转义或纳入正字符组。 例如，正则表达式 \\$\d+ 或 [\$]\d+ 匹配 “\$1200”。
\a	匹配警报字符。
\b	在 [character_group] 字符类中，匹配退格。在字符类之外，\b 是匹配字边界的定位点。
\t	匹配制表符。
\r	匹配回车。 请注意，\r 不等同于换行符，\n。
\v	匹配垂直制表符。
\f	匹配换页。
\n	匹配换行。
\c	匹配转义。
\ nnn	匹配 ASCII 字符，其中 nnn 包含表示八进制字符代码的两位数或三位数。 例如，\040 表示空格字符。 如果此构造仅包含一个数字（如 \2）或者它对应捕获组的编号，则将它解释为向后引用。
\x nn	匹配 ASCII 字符，其中 nn 是两位数的十六进制字符代码。
\c X	匹配 ASCII 控制字符，其中 X 是控制字符的字母。 例如，\cC 为 CTRL-C。
\u nnnn	匹配的 UTF-16 代码单元，单元值是 nnnn 十六进制。
\	后接字符未识别为转义字符时，将匹配此字符。 例如，* 匹配星号 (*) 并等同于 \x2A。

2. 功能正则表达式案例

1) 火车车次

PS: /^[GCDZTSPKXLY1-9]\d{1,4}\$/

2)手机机身码——IMEI

PS: /^[^d]{15,17}\$/

3) 必须带端口号的网址(或 ip)

PS: `/^((ht|f)tps?:\W)?([\w-]+(\.[\w-]+)+:\d{1,5})\W?$`

4) 网址——URL

PS:

`/^(((ht|f)tps?:\W)?([^\!@#%&*?.\s-]([^\!@#%&*?.\s]{0,63}([^\!@#%&*?.\s])?.)+[a-z]{2,6})\W?)$`

5) 统一社会信用代码

`/^[0-9A-HJ-NPQRTUWXY]{2}\d{6}[0-9A-HJ-NPQRTUWXY]{10}$`

6) 统一社会信用代码(宽松匹配)(15 位/18 位/20 位数字/字母)

`/^([0-9A-Za-z]{15})|([0-9A-Za-z]{18})|([0-9A-Za-z]{20})$`

7) 迅雷链接

`/^thunderx?:\W[a-zA-Z\d]+=`

8) ed2k 链接(宽松匹配)

`/^ed2k:\W\file\.\+|\W$`

9) 磁力链接(宽松匹配)

`/^magnet:?\xt=urn:btih:[0-9a-fA-F]{40,}\.*$`

10) 子网掩码(不包含 0.0.0.0)

`/^(254|252|248|240|224|192|128)\.0\.0\.0|255\.(254|252|248|240|224|192|128|0)\.0\.0|255\.255\.(254|252|248|240|224|192|128|0)\.0|255\.255\.(255|254|252|248|240|224|192|128|0)$`

11) linux "隐藏文件" 路径

`/^V(?:[^\^]+V)*\.[^\^]*$`

12) linux 文件夹路径

`/^V(?:[^\^]+V)*$`

13) linux 文件路径

`/^V(?:[^\^]+V)*[^\^]+$`

14) window "文件夹" 路径

`/^[a-zA-Z]:\\(?:\w+\\?)*$`

15) window 下 "文件" 路径

`/^[a-zA-Z]:\\(?:\w+\\)*\w+\.\\w+$`

16) 股票代码(A 股)

`/^(s[hz]|S[HZ])(000[\d]{3}|002[\d]{3}|300[\d]{3}|600[\d]{3}|60[\d]{4})$`

17) 大于等于 0, 小于等于 150, 支持小数位出现 5, 如 145.5, 用于判断考卷分数

`/^150$|^(?:\d|[1-9]\d|1[0-4]\d)(?:\.\d{1,5})?$`

18) html 注释

`/<!--[\s\S]*?-->/g`

19) md5 格式(32 位)

`/^[a-fA-F0-9]{32}$`

- 20) GUID/UUID
`/^[a-fd]{4}?(?:[a-fd]{4}-){4}[a-fd]{12}$/i`
- 21) 版本号(version)格式必须为 X.Y.Z
`/^d+(?:\.\d+){2}$/`
- 22) 视频(video)链接地址（视频格式可按需增删）
`/^https?:\W(.+V)+.(swf|avi|flv|mpg|rm|mov|wav|asf|3gp|mkv|rmvb|mp4))$/i`
- 23) 图片(image)链接地址（图片格式可按需增删）
`/^https?:\W(.+V)+.(gif|png|jpg|jpeg|webp|svg|psd|bmp|tif))$/i`
- 24) 24 小时制时间（HH:mm:ss）
`/^(?:[01]\d|2[0-3]):[0-5]\d:[0-5]\d$/`
- 25) 12 小时制时间（hh:mm:ss）
`/^(?:1[0-2]|0?[1-9]):[0-5]\d:[0-5]\d$/`
- 26) base64 格式
`/^\s*data:(?:[a-z]+V[a-z0-9-+.]+(?:;[a-z-]+=a-z0-9-+)?(?:;base64)?,([a-z0-9!$&',()*+;\=\-\._~:@/?%\s]*?))\s*$/i`
- 27) 数字/货币金额（支持负数、千分位分隔符）
`/^-?\d+(\,\d{3})*(\.\d{1,2})?$/`
- 28) 数字/货币金额（只支持正数、不支持校验千分位分隔符）
`/^(?:^[1-9]([0-9]+)?(?:\.[0-9]{1,2})?)$|(?:^(?:0)$|(?^[0-9]\.[0-9](?:[0-9])?))$/`
- 29) 银行卡号（10 到 30 位）
`/^[1-9]\d{9,29}$/`
- 30) 中文姓名
`/^(?:[\u4e00-\u9fa5]{2,16})$/`
- 31) 英文姓名
`/^[a-zA-Z][a-zA-Z\s]{0,20}[a-zA-Z]$/`
- 32) 车牌号(新能源)
`/^[京津沪渝冀豫云辽黑湘皖鲁新苏浙赣鄂桂甘晋蒙陕吉闽贵粤青藏川宁琼使领][A-HJ-NP-Z](?:((\d{5}[A-HJK])|([A-HJK][A-HJ-NP-Z0-9]{0-9}{4})))|([A-HJ-NP-Z0-9]{4}[A-HJ-NP-Z0-9 挂学警港澳])$/`
- 33) 车牌号(非新能源)
`/^[京津沪渝冀豫云辽黑湘皖鲁新苏浙赣鄂桂甘晋蒙陕吉闽贵粤青藏川宁琼使领][A-HJ-NP-Z][A-HJ-NP-Z0-9]{4}[A-HJ-NP-Z0-9 挂学警港澳]$/`
- 34) 车牌号(新能源+非新能源)
`/^[京津沪渝冀豫云辽黑湘皖鲁新苏浙赣鄂桂甘晋蒙陕吉闽贵粤青藏川宁琼使领][A-HJ-NP-Z][A-HJ-NP-Z0-9]{4,5}[A-HJ-NP-Z0-9 挂学警港澳]$/`
- 35) 手机号(mobile phone)中国(严谨)，根据工信部 2019 年最新公布的手机号段
`/^(?:\+|00)86)?1(?:\d{3}|(\d{4}[5-79])|(\d{5}[0-35-9])|(\d{6}[5-7])|(\d{7}[0-8])|(\d{8}|\d)|(\d{9}[189])\d{8})$/`

DC00-\uDFFF][\uD869[\uDC00-\uDED6\uDF00-\uDFFF][\uD86D[\uDC00-\uDF34\uDF40-\uDF
FF][\uD86E[\uDC00-\uDC1D\uDC20-\uDFFF][\uD873[\uDC00-\uDEA1\uDEB0-\uDFFF][\uD87
A[\uDC00-\uDFE0)]+\$/

50) 小数

$$/\wedge\backslash d+\backslash.\backslash d+\$/$$

51) 只包含数字

$$/\wedge\backslash\mathbf{d}+\$/$$

52) html 标签(宽松匹配)

```
/<(\w+)[^>]*>(.*?<\1>)?/
```

53) 匹配中文汉字和中文标点

/[\u4e00-\u9fa5|\u3002|\uff1f|\uff01|\uff0c|\u3001|\uff1b|\uff1a|\u201c|\u201d|\u2018|\u2019|\uff08|\uff09|\u300a|\u300b|\u3008|\u3009|\u3010|\u3011|\u300e|\u300f|\u300c|\u300d|\ufe43|\ufe44|\u3014|\u3015|\u2026|\u2014|\uff5e|\ufe4f|\uffe5]/

54) qq 号格式正确

$$/^{[1-9][0-9]\{4,10\}}\$/$$

55) 数字和字母组成

`/^[A-Za-z0-9]+$`

56) 英文字母

`/^[a-zA-Z]+$`

57) 小写英文字母组成

`/^[a-z]+$`

58) 大写英文字母

`/^[A-Z]+$`

59) 密码强度校验，最少 6 位，包括至少 1 个大写字母，1 个小写字母，1 个数字，1 个特殊字符

$$\wedge S^*(?=S\{6,\})(?=S^*d)(?=S^*[A-Z])(?=S^*[a-z])(?=S^*![@\#\%^&*?])\S^*/$$

60) 用户名校验, 4 到 16 位 (字母, 数字, 下划线, 减号)

$$/^{\wedge}[a-zA-Z0-9 -]{4,16}\$/$$

61) ip-v4[:端口]

$$\wedge(\wedge[d[1-9]\backslash d|1\backslash d\backslash d2[0-4]\backslash d25[0-5])\backslash .\}\{(\wedge[d[1-9]\backslash d|1\backslash d\backslash d2[0-4]\backslash d25[0-5])(?:((?:[0-9][1-9][0-9]\{1,3\}[1-5][0-9]\{4\}|6[0-4][0-9]\{3\}|65[0-4][0-9]\{2\}|655[0-2][0-9]|6553[0-5]))? \$/$$

62) ip-v6[:端口]

$$\begin{aligned} & / \wedge (?: (?: [0-9A-Fa-f] \{1,4\} :) \{7\} [0-9A-Fa-f] \{1,4\}) (([0-9A-Fa-f] \{1,4\} :) \{6\} : [0-9A-Fa-f] \{1,4\}) \\ &) (([0-9A-Fa-f] \{1,4\} :) \{5\} : ([0-9A-Fa-f] \{1,4\} :) ? [0-9A-Fa-f] \{1,4\}) (([0-9A-Fa-f] \{1,4\} :) \{4\} : ([0-9A- \\ & Fa-f] \{1,4\} :) \{0,2\} [0-9A-Fa-f] \{1,4\}) (([0-9A-Fa-f] \{1,4\} :) \{3\} : ([0-9A-Fa-f] \{1,4\} :) \{0,3\} [0-9A-Fa-f] \\ & \{1,4\}) (([0-9A-Fa-f] \{1,4\} :) \{2\} : ([0-9A-Fa-f] \{1,4\} :) \{0,4\} [0-9A-Fa-f] \{1,4\}) (([0-9A-Fa-f] \{1,4\} :) \{6\} \\ & ((\backslash b ((25 [0-5]) (1 \backslash d \{2\}) (2 [0-4] \backslash d) (\backslash d \{1,2\})) \backslash .) \{3\} \backslash b ((25 [0-5]) (1 \backslash d \{2\}) (2 [0-4] \backslash d) (\backslash d \{1,2\})) \backslash .) \\ & (([0-9A-Fa-f] \{1,4\} :) \{0,5\} : (\backslash b ((25 [0-5]) (1 \backslash d \{2\}) (2 [0-4] \backslash d) (\backslash d \{1,2\})) \backslash .) \{3\} \backslash b ((25 [0-5]) (1 \backslash d \{2\}) \\ & (2 [0-4] \backslash d) (\backslash d \{1,2\})) \backslash .) (: ([0-9A-Fa-f] \{1,4\} :) \{0,5\} (\backslash b ((25 [0-5]) (1 \backslash d \{2\}) (2 [0-4] \backslash d) (\backslash d \{1,2\})) \backslash .) \\ & \{3\} \backslash b ((25 [0-5]) (1 \backslash d \{2\}) (2 [0-4] \backslash d) (\backslash d \{1,2\})) \backslash .) ([0-9A-Fa-f] \{1,4\} : : ([0-9A-Fa-f] \{1,4\} :) \{0,5\} [\end{aligned}$$

```
0-9A-Fa-f]{1,4})((?:[0-9A-Fa-f]{1,4}:){0,6}[0-9A-Fa-f]{1,4})((?:[0-9A-Fa-f]{1,4}:){1,7}:))$)(^\[
(?:?:?:[0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-f]{1,4})((?:[0-9A-Fa-f]{1,4}:){6}:[0-9A-Fa-f]{1,4})((?:[0-9
A-Fa-f]{1,4}:){5}:([0-9A-Fa-f]{1,4}:)?[0-9A-Fa-f]{1,4})((?:[0-9A-Fa-f]{1,4}:){4}:([0-9A-Fa-f]{1,
4}:){0,2}[0-9A-Fa-f]{1,4})((?:[0-9A-Fa-f]{1,4}:){3}:([0-9A-Fa-f]{1,4}:){0,3}[0-9A-Fa-f]{1,4})((
[0-9A-Fa-f]{1,4}:){2}:([0-9A-Fa-f]{1,4}:){0,4}[0-9A-Fa-f]{1,4})((?:[0-9A-Fa-f]{1,4}:){6}((b((2
5[0-5]))(1\d{2}))(2[0-4]\d)(\d{1,2}))\b\.\.){3}((b((25[0-5]))(1\d{2}))(2[0-4]\d)(\d{1,2}))\b\.\.))((?:[0-9
A-Fa-f]{1,4}:){0,5}((b((25[0-5]))(1\d{2}))(2[0-4]\d)(\d{1,2}))\b\.\.){3}((b((25[0-5]))(1\d{2}))(2[
0-4]\d)(\d{1,2}))\b\.\.))((?:[0-9A-Fa-f]{1,4}:){0,5}((b((25[0-5]))(1\d{2}))(2[0-4]\d)(\d{1,2}))\b\.\.){
3}((b((25[0-5]))(1\d{2}))(2[0-4]\d)(\d{1,2}))\b\.\.))((?:[0-9A-Fa-f]{1,4}:){0,5}([0-9A-Fa-f]{1,4}):
[0-9A-Fa-f]{1,4})((?:[0-9A-Fa-f]{1,4}:){0,6}[0-9A-Fa-f]{1,4})((?:[0-9A-Fa-f]{1,4}:){1,7}:)))(?:?:[0-9]
-9][1-9][0-9]{1,3}[1-5][0-9]{4}|6[0-4][0-9]{3}|65[0-4][0-9]{2}|655[0-2][0-9]|6553[0-5]))?)$/i
```

63) 16 进制颜色

```
/^#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3})$/
```

64) 微信号(wx)，6 至 20 位，以字母开头，字母，数字，减号，下划线

```
/^[a-zA-Z][-_a-zA-Z0-9]{5,19}$/
```

65) 邮政编码(中国)

```
/^0[1-7]1[0-356]2[0-7]3[0-6]4[0-7]5[1-7]6[1-7]7[0-5]8[013-6])\d{4}$/
```

66) 中文和数字

```
/^((?:[\u3400-\u4DB5\u4E00-\u9FEA\uFA0E\uFA0F\uFA11\uFA13\uFA14\uFA1F\uFA21\uF
A23\uFA24\uFA27-\uFA29][\uD840-\uD868\uD86A-\uD86C\uD86F-\uD872\uD874-\uD879][\u
DC00-\uDFFF]|\uD869[\uDC00-\uDED6\uDF00-\uDFFF]|\uD86D[\uDC00-\uDF34\uDF40-\uDF
FF]|\uD86E[\uDC00-\uDC1D\uDC20-\uDFFF]|\uD873[\uDC00-\uDEA1\uDEB0-\uDFFF]|\uD87
A[\uDC00-\uDFE0])|(d))+$/
```

67) 不能包含字母

```
/^[^A-Za-z]*$/
```

68) mac 地址

```
/^((([a-f0-9]{2}:){5})([a-f0-9]{2}-){5})[a-f0-9]{2}$/i
```

69) 匹配连续重复的字符

```
/(\.)1+ /
```

70) 数字和英文字母组成，并且同时含有数字和英文字母

```
/^(?=.*[a-zA-Z])(?=.*\d).+$/
```

71) 香港身份证

```
/^[a-zA-Z]\d{6}([\dA])$/
```

72) 澳门身份证

```
/^[1|5|7]\d{6}(\d)$/
```

73) 台湾身份证

```
/^[a-zA-Z][0-9]{9}$/
```

74) 大写字母，小写字母，数字，特殊符号 `@#%&*~()-+=` 中任意 3 项密码

```
/^(?!([a-zA-Z]+$)(?!([A-Z0-9]+$)(?!([A-Z\W_!@#%&*~()-+=]+$)(?!([a-z0-9]+$)(?!([a-z\W_!
@#%&*~()-+=]+$)(?!([0-9\W_!@#%&*~()-+=]+$)[a-zA-Z0-9\W_!@#%&*~()-+=])
```

- 75) ASCII 码表中的全部的特殊字符
`/[\x21-\x2F\x3A-\x40\x5B-\x60\x7B-\x7E]+/`
- 76) 正整数, 不包含 0
`/^[1-9]\d*$/`
- 77) 负整数, 不包含 0
`/^-[1-9]\d*$/`
- 78) 整数
`/^(?:0(?:-[1-9]\d*)?)$/`
- 79) 浮点数
`/^(-?[1-9]\d*\.\d+|-?0\.\d*[1-9]\d*|0\.\d+)$/`
- 80) 浮点数(严格)
`/^(-?[1-9]\d*\.\d+|-?0\.\d*[1-9]\d*)$/`
- 81) email(支持中文邮箱)
`/^[A-Za-z0-9\u4e00-\u9fa5]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$/`
- 82) 域名(非网址, 不包含协议)
`/^([0-9a-zA-Z]{1,}\.)([a-zA-Z]{2,})$/`

四十六. C#弃元

弃元是一种在应用程序代码中人为取消使用的占位符变量。弃元相当于未赋值的变量；它们没有值。弃元将意图传达给编译器和其他读取代码的文件：你打算忽略表达式的结果。

我们可能需要忽略表达式的结果、元组表达式的一个或多个成员、方法的 out 参数或模式匹配表达式的目标。

弃元使代码的意图更加明确，可指示代码永远不会使用变量。它们可以增强其可读性和可维护性。

通过将下划线 () 赋给一个变量作为其变量名，指示该变量为一个占位符变量。例如，以下方法调用返回一个元组，其中第一个值和第二个值为弃元。area 是以前声明的变量，设置为由 GetCityInformation 返回的第三个组件。

PS:

```
(_, _, area) = city.GetCityInformation(cityName);
```

从 C# 9.0 开始，可以使用弃元指定 Lambda 表达式中不使用的输入参数。

当 _ 是有效弃元时，尝试检索其值或在赋值操作中使用它时会生成编译器错误 CS0301：“当前上下文中不存在名称 “_””。出现此错误是因为 _ 未赋值，甚至可能未分配存储位置。如果它是一个实际变量，则不能像之前的示例那样对多个值使用弃元。

四十七. C#DateTime

1. 转换为字符串

1) 分类

DateTime 调用 ToString()传入的参数可分为制式和自定义两种。

a. 制式

符号	语法	示例(2016-05-09 13:09:55:2350)
y	DateTime.Now.ToString()	2016/5/9 13:09:55
d	DateTime.Now.ToString("d")	2016/5/9
D	DateTime.Now.ToString("D")	2016 年 5 月 9 日
f	DateTime.Now.ToString("f")	2016 年 5 月 9 日 13:09
F	DateTime.Now.ToString("F")	2016 年 5 月 9 日 13:09:55
g	DateTime.Now.ToString("g")	2016/5/9 13:09
G	DateTime.Now.ToString("G")	2016/5/9 13:09:55
t	DateTime.Now.ToString("t")	13:09
T	DateTime.Now.ToString("T")	13:09:55
u	DateTime.Now.ToString("u")	2016-05-09 13:09:55Z
U	DateTime.Now.ToString("U")	2016 年 5 月 9 日 5:09:55
m	DateTime.Now.ToString("m")	5 月 9 日
M	DateTime.Now.ToString("M")	5 月 9 日
r	DateTime.Now.ToString("r")	Mon, 09 May 2016 13:09:55 GMT
R	DateTime.Now.ToString("R")	Mon, 09 May 2016 13:09:55 GMT
y	DateTime.Now.ToString("y")	2016 年 5 月
Y	DateTime.Now.ToString("Y")	2016 年 5 月
o	DateTime.Now.ToString("o")	2016-05-09T13:09:55.2350000
O	DateTime.Now.ToString("O")	2016-05-09T13:09:55.2350000
s	DateTime.Now.ToString("s")	2016-05-09T13:09:55

b. 自定义

符号	说明	语法	示例(2016-05-09 13:09:55:2350)
yy	年份后两位	DateTime.Now.ToString("yy")	16
yyyy	4 位年份	DateTime.Now.ToString("yyyy")	2016
MM	两位月份；单数月份前面用 0 填充	DateTime.Now.ToString("MM")	05
dd	日数	DateTime.Now.ToString("dd")	09
ddd	周几	DateTime.Now.ToString("ddd")	周一
dddd	星期几	DateTime.Now.ToString("dddd")	星期一
hh	12 小时制的小时数	DateTime.Now.ToString("hh")	01
HH	24 小时制的小时数	DateTime.Now.ToString("HH")	13
mm	分钟数	DateTime.Now.ToString("mm")	09
ss	秒数	DateTime.Now.ToString("ss")	55
ff	毫秒数前 2 位	DateTime.Now.ToString("ff")	23
fff	毫秒数前 3 位	DateTime.Now.ToString("fff")	235

ffff	毫秒数前 4 位	DateTime.Now.ToString("ffff")	2350
分隔符	可使用分隔符来分隔年月 日时分秒。 包含的值可为：-、/、:等非 关键字符	DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:ffff");	2016-05-09 13:09:55:2350

2. 时间戳转换

C# DateTime 与时间戳的相互转换，包括 [JavaScript 时间戳](#) 和 [Unix 的时间戳](#)。

1) JavaScript 与 Unix 的时间戳的区别

[JavaScript 时间戳](#)：是指格林威治时间 1970 年 01 月 01 日 00 时 00 分 00 秒(北京时间 1970 年 01 月 01 日 08 时 00 分 00 秒)起至现在的**总毫秒数**。

[Unix 时间戳](#)：是指格林威治时间 1970 年 01 月 01 日 00 时 00 分 00 秒(北京时间 1970 年 01 月 01 日 08 时 00 分 00 秒)起至现在的**总秒数**。

比如同样是 2016/11/03 12:30:00，转换为 JavaScript 时间戳为 1478147400000；转换为 Unix 时间戳为 1478147400。

2) JavaScript 时间戳相互转换

a. C# DateTime 转换为 JavaScript 时间戳

PS:

```
System.DateTime startTime = TimeZone.CurrentTimeZone.ToLocalTime(new
System.DateTime(1970, 1, 1)); // 当地时间
long timeStamp = (long)(DateTime.Now - startTime).TotalMilliseconds; // 相差毫秒数
System.Console.WriteLine(timeStamp);
```

b. JavaScript 时间戳转换为 C# DateTime

PS:

```
long jsTimeStamp = 1478169023479;
System.DateTime startTime = TimeZone.CurrentTimeZone.ToLocalTime(new
System.DateTime(1970, 1, 1)); // 当地时间
DateTime dt = startTime.AddMilliseconds(jsTimeStamp);
System.Console.WriteLine(dt.ToString("yyyy/MM/dd HH:mm:ss:ffff"));
```

3) Unix 时间戳相互转换

a. C# DateTime 转换为 Unix 时间戳

PS:

```
System.DateTime startTime = TimeZone.CurrentTimeZone.ToLocalTime(new
System.DateTime(1970, 1, 1)); // 当地时间
long timeStamp = (long)(DateTime.Now - startTime).TotalSeconds; // 相差秒数
System.Console.WriteLine(timeStamp);
```

b. Unix 时间戳转换为 C# DateTime

PS:

```
long unixTimeStamp = 1478162177;
System.DateTime startTime = TimeZone.CurrentTimeZone.ToLocalTime(new
System.DateTime(1970, 1, 1)); // 当地时区
DateTime dt = startTime.AddSeconds(unixTimeStamp);
System.Console.WriteLine(dt.ToString("yyyy/MM/dd HH:mm:ss:ffff"));
```

四十八. C# Var 与 Dynamic

1. Var

Var 是 C# 3.0 中引入的，var 本身并不是一种类型，其实它仅仅只是一个语法糖，它要求编译器根据一个表达式推断具体的数据类型，变量实际的类型是编译时所赋值的类型。var 声明的变量在赋值的那一刻，就已经决定了它是什么类型，所以 Var 类型的变量在初始化时候，必须提供初始化的值。

1) 优缺点

a. 优点

- ① 它有利于更好地为本地变量命名。
- ② 它有利于设计更好的 API。
- ③ 它促使编程人员对变量进行强制初始化。
- ④ 它消除了代码的混乱。
- ⑤ 它不需要 using 指示符。

b. 缺点

- ① 可读性差：因为变量类型是隐式的，所以代码的可读性可能会降低。
- ② 难以调试：当遇到错误时，可能很难找到代码中隐含的类型。
- ③ 难以维护：因为变量类型是隐式的，如果要更改代码结构，则可能需要额外的工作。

2. dynamic

是 C# 4.0 中引入的新类型，它的特点是申明为 dynamic 类型的变量，不是在编译时候确定实际类型的，而是在运行时。用 dynamic 声明的变量是动态类型的。这个功能被添加到 CLR 中，以支持动态语言，比如 Ruby 和 Python。

这意味着 dynamic 声明是在运行时解析的，而 Var 声明是在编译时解析的。而 dynamic 被编译后，实际是一个 object 类型，只不过编译器会对 dynamic 类型进行特殊处理，让它在编译期间不进行任何的类型检查，而是将类型检查放到了运行期。

四十九. C# 弱引用

通常情况下我们声明一个引用类型的变量都是强引用的关系，如下：

PS:

```
Object obj = new Object();
```

此时引用计数会+1，在 GC 时该对象并不会被释放。此时若我们用一个新的引用指向它，

那么引用计数还会再次增加，如下：

PS：

```
Object otherObj = obj;
```

此时生成的 Object 对象引用计数为 2，如果我们执行 `obj = null;` Object 对象引用计数会变为 1，而 `otherObj` 依旧引用着，因此还是不会被 GC。

C#为我们提供了 `System.WeakReference` 类型来声明一个弱引用对象，如下：

PS：

```
System.WeakReference weakObj = new System.WeakReference(obj);
```

弱引用方式就不会导致引用计数增加，因此 Object 对象引用计数依旧为 1（没有 `otherObj` 那步操作）。所以如果我们执行 `obj = null;` Object 对象引用计数会变为 0，就会被 GC，可以有效避免内存泄漏问题。

可以利用 `.IsAlive` 属性来判断弱引用引用的对象是否还存在，使用 `.Target` 属性来获取弱引用引用的对象。

弱引用常用于一些游戏内的统计或者异步对象等。

五十. 序列化与反序列化

序列化和反序列化是将对象转换为可存储或传输的格式，以及将该格式转换回对象的过程。

序列化是将对象转换为字节流或类似的格式，以便它可以存储或传输。通常，序列化后的对象可以被存储到磁盘或通过网络发送到另一个计算机。

反序列化是将序列化后的格式转换回对象的过程。这通常涉及读取序列化的数据流，并将其转换为程序中的对象。反序列化后，可以使用对象作为程序的输入。

序列化和反序列化技术通常在分布式系统和网络通信中使用，以便在不同计算机之间传输数据。例如，当您在 Web 服务上调用 REST API 时，通常会将请求和响应序列化为 JSON 或 XML 格式以便在网络上传输，然后反序列化以在客户端上使用响应数据。

五十一. HashSet—哈希集

1. 概述

`HashSet<T>` 类主要被设计用来存储集合，做高性能集运算，例如两个集合求交集、并集、差集等。从名称可以看出，它是基于 Hash 的，可以简单理解为没有 Value 的 Dictionary。优势在于集合运算快，作为一种存放在内存的数据，可以很快的进行设置和取值的操作。

2. 特性介绍

- ① `HashSet<T>` 的容量指的是可以容纳的元素总数，增减元素时，容量会自动增加，但不会自动减少。
- ② `HashSet<T>` 不自带排序方法，如果需要排序的可以参考使用 `List` 集合配合 `Sort` 方法。
- ③ `HashSet<T>` 元素是唯一的，不可重复，同时区分大小写。
- ④ `HashSet<T>` 不能使用下标来访问元素。

3. HashSet<T> 的优势和与 List<T> 的比较

`HashSet<T>` 最大的优势是检索的性能，简单的说它的 `Contains` 方法的性能在大数据量时比 `List<T>` 好得多。

在内部算法实现上，`HashSet<T>` 的 `Contains` 方法复杂度是 $O(1)$ ，`List<T>` 的 `Contains` 方法复杂度是 $O(n)$ ，后者数据量越大速度越慢，而 `HashSet<T>` 不受数据量的影响。

这里的方法复杂度就是 时间复杂度 和 空间复杂度 的综合评定

所以在集合的目的是为了检索的情况下，我们应该使用 `HashSet<T>` 代替 `List<T>`。

比如一个存储关键字的集合，运行的时候通过其 `Contains` 方法检查输入字符串是否关键字。

4. HashSet<T> 类的方法和属性

1) 属性

属性	描述
获取集合中现有元素的总数	获取集合中现有元素的总数

2) 方法

方法	描述
<code>bool Add (T item);</code>	添加指定元素，返回 <code>bool</code> 值指示是否执行成功
<code>bool Remove (T item);</code>	移除指定元素，返回 <code>bool</code> 值表示是否执行成功
<code>void Clear ();</code>	移除所有元素
<code>bool Contains (T item);</code>	判断是否包含指定元素
<code>void CopyTo (T[] array);</code>	复制元素到数组中
<code>void ExceptWith (IEnumerable<T> other);</code>	移除当前集合中指定子集的元素
<code>void IntersectWith (IEnumerable<T> other);</code>	修改当前集合元素为当前集合与指定集合的交集
<code>void UnionWith (IEnumerable other);</code>	修改当前集合元素为当前集合与指定集合的并集
<code>bool IsProperSubsetOf (IEnumerable other);</code>	判断当前集合是否为指定集合的真子集
<code>bool IsProperSupersetOf (IEnumerable other);</code>	判断当前集合是否为指定集合的真超集
<code>bool IsSubsetOf (IEnumerable other);</code>	判断当前集合是否为指定集合的子集
<code>bool IsSupersetOf (IEnumerable other);</code>	判断当前集合是否为指定集合的超集
<code>bool Overlaps (IEnumerable other);</code>	判断当前集合是否与指定集合至少有一个公共元素
<code>bool SetEquals (IEnumerable other);</code>	判断当前集合是否与指定集合包含相同的元素
<code>bool TryGetValue (T equalValue, out T actualValue);</code>	搜索给定值，并返回所找到的相等值
<code>void TrimExcess ();</code>	将当前集合的容量设置为它包含的实际元素数
<code>bool Add (T item);</code>	添加指定元素，返回 <code>bool</code> 值指示是否执行成功
<code>bool Remove (T item);</code>	移除指定元素，返回 <code>bool</code> 值表示是否执行成功
<code>void Clear ();</code>	移除所有元素
<code>bool Contains (T item);</code>	判断是否包含指定元素
<code>void CopyTo (T[] array);</code>	复制元素到数组中
<code>void ExceptWith (IEnumerable<T> other);</code>	移除当前集合中指定子集的元素
<code>void IntersectWith (IEnumerable<T> other);</code>	修改当前集合元素为当前集合与指定集合的交集

<code>void UnionWith (IEnumerable other);</code>	修改当前集合元素为当前集合与指定集合的并集
--	-----------------------

五十二. 依赖注入

依赖注入是一种设计模式。我们不是直接在另一个类（依赖类）中创建一个类的对象，而是将对象作为参数传递给依赖类的构造函数。它有助于编写松散耦合的代码，并有助于使代码更加模块化和易于测试。实现依赖注入的三种方式：

- ① **构造函数注入**：这是最常用的注入类型。在构造函数注入中，我们可以将依赖项传递给构造函数。我们必须确保这里没有默认构造函数，唯一的应该是参数化构造函数。
- ② **属性注入**：在某些情况下，我们需要一个类的默认构造函数，那么在这种情况下，我们可以使用属性注入。
- ③ **方法注入**：在方法注入中，我们只需要在方法中传递依赖即可。当整个类不需要那个依赖时，就不需要实现构造函数注入。当我们对多个对象有依赖关系时，我们不会在构造函数中传递该依赖关系，而是在需要它的函数本身中传递该依赖关系。

五十三. Lazy

1. 功能简述

`Lazy<T>` 是一个类，用于实现懒加载（Lazy Initialization）。懒加载是指直到第一次被使用时，对象才进行创建。`Lazy<T>` 允许我们在第一次访问对象时进行初始化，这对于大型或资源密集型对象的性能优化非常有用。我们可以通过提供一个委托（Delegate）来延迟初始化对象，`Lazy<T>` 确保所有线程使用同一个懒加载对象的实例，并且丢弃未使用的实例，从而优化内存使用。

- ① **延迟初始化**（Lazy Initialization）：`Lazy<T>` 允许我们将对象的创建推迟到首次访问时。
- ② **线程安全**（Thread-Safe）：`Lazy<T>` 提供了线程安全的延迟初始化，确保在多线程环境中也能正确工作。
- ③ **自动丢弃未使用的实例**：如果对象未被使用，`Lazy<T>` 会自动丢弃初始化失败的实例，优化内存使用。
- ④ **支持复杂的初始化逻辑**：我们可以提供一个委托，允许我们在初始化对象时执行复杂的逻辑。
- ⑤ **Value 属性**：通过 `Lazy<T>.Value` 属性访问延迟初始化的对象。

1) Lazy 如何实现懒加载

懒加载（Lazy Loading）是一种设计模式，它允许我们将对象的创建推迟到实际需要的时候进行。在 C# 中，`Lazy<T>` 类实现了这个模式。当我们使用 `Lazy<T>` 时，它会推迟目标对象的创建直到我们第一次访问 `Lazy<T>` 的 `Value` 属性。这样，我们可以在程序运行时避免不必要的初始化和资源消耗，提高性能。

具体原理是，`Lazy<T>` 内部使用了一个委托，该委托负责创建目标对象。当我们第一次访问 `Lazy<T>` 的 `Value` 属性时，该委托会执行，实例化目标对象，并将其保存下来。随后的访问会直接返回已经创建好的对象，而不会再次执行委托。

PS:

```
Lazy<MyClass> lazyInstance = new Lazy<MyClass>(() => new MyClass());
MyClass myObject = lazyInstance.Value; // 对象在第一次访问时创建
```

2. Lazy 的缺点

- ① **性能开销**：在第一次访问 `Lazy<T>` 对象时，需要进行初始化操作，这可能会引入

一定的性能开销，特别是在初始化逻辑较复杂或耗时的情况下。

- ② **线程安全性**：默认情况下，`Lazy<T>`是线程安全的，但如果需要在多线程环境下共享实例，可能需要额外的线程同步措施，这会增加复杂性。
- ③ **内存占用**：虽然 `Lazy<T>`可以延迟对象的创建，但在对象创建后，它将一直占用内存，即使后续不再需要该对象。
- ④ **不适用于某些场景**：`Lazy<T>`适用于需要延迟初始化的场景，但并不适用于所有情况。在某些情况下，可能需要即时创建对象或使用其他设计模式。
- ⑤ **引入额外复杂性**：在某些情况下，使用 `Lazy<T>`可能会引入额外的复杂性，使代码变得难以理解和维护。

1) Lazy 是否存在性能问题？

`Lazy<T>` 类型是为了延迟初始化而设计的，**用于提高性能、避免资源浪费和减少内存需求**。`Lazy<T>` 在实例化时不会立即创建对象，只有在第一次访问时才会实际进行初始化。因此，它**不会导致性能问题**，相反，它通常用于优化性能，特别是在需要创建大量对象时，但不一定需要立即使用这些对象的情况下。

但需要注意的是，如果初始化逻辑本身非常耗时，那么在第一次访问时会有一定的性能开销。但这并不是 `Lazy<T>`的问题，而是**由初始化逻辑引起的**。

3. Lazy 的使用场景

- ① **延迟加载大对象**：当我们有一个大对象，希望在需要的时候再初始化，可以使用 `Lazy<T>`。这样可以避免在应用程序启动时就加载大对象，提高了启动速度。

```
Lazy<BigObject> lazyObject = new Lazy<BigObject>(() => new BigObject());
```

```
BigObject obj = lazyObject.Value; // 在需要时初始化
```

- ② **单例模式的延迟初始化**：`Lazy<T>` 可以用于实现线程安全的单例模式，**确保只有一个实例被创建**。

```
private static Lazy<SingletonClass> _instance = new Lazy<SingletonClass>(() => new SingletonClass());
```

```
public static SingletonClass Instance => _instance.Value;
```

- ③ **按需加载集合**：我们可以使用 `Lazy<T>` 来按需加载集合，以节省内存和提高性能。例如，加载大型数据集。

```
Lazy<List<DataItem>> lazyData = new
```

```
Lazy<List<DataItem>>(LoadDataFromDatabase);
```

```
List<DataItem> data = lazyData.Value; // 在需要时加载数据
```

- ④ **多线程应用中的线程安全延迟初始化**：`Lazy<T>` 提供了多种线程安全模式，可以**确保在多线程环境下仍然能够安全地进行延迟初始化**。

```
Lazy<ThreadSafeObject> lazyThreadSafeObject = new Lazy<ThreadSafeObject>(() => new ThreadSafeObject(), LazyThreadSafetyMode.ExecutionAndPublication);
```

五十四. Primary Constructors—主构造函数

1. 什么是主构造函数

把参数添加到 `class` 与 `record` 的类**声明中**就是**主构造函数**。例如

PS:

```
class Person(string name)
{
    private string _name = name;
}
```

上面的写法与以下代码写法一样。

PS:

```
class Person
{
    private string _name;
    public Person(string name)
    {
        _name = name;
    }
}
```

需要注意的是，类的所有其他构造函数都必须通过 `this()` 构造函数直接或间接调用主构造函数。

PS:

```
class Person(string name)
{
    public Person(int age,string name):this(name)
    {

    }

}
```

主构造函数参数的最常见用途包括但不限于以下几个方面：

- ① 初始化基类。
- ② 初始化成员字段或属性。
- ③ 简化依赖注入。

2. 初始化基类

可以从派生类的主构造函数调用基类的主构造函数。这是编写必须调用基类中主构造函数的派生类的最简单方法。

PS:

```
class Person(string name)
{
    private string _name = name;
}

class Man(string name):Person(name)
{

}
```

派生类如果没有主构造函数，可以在派生类中创建一个构造函数，用于调用基类的主构造函数。

PS:

```
class Person(string name)
{
    private string _name = name;
}
```



```
class Woman : Person
{
    public Woman(string name) : base(name)
    {

    }

}
}
```

3. 初始化成员字段或属性

PS:

```
class Person(string name)
{
    private string _name = name;
}
}
```

4. 简化依赖注入

主构造函数的另一个常见用途是指定依赖项注入的参数。下面的代码创建了一个简单的控制器，使用时需要有一个服务接口。

PS:

```
public interface IService
{
    Distance GetDistance();
}

public class ExampleController(IService service) : ControllerBase
{
    [HttpGet]
    public ActionResult<Distance> Get()
    {
        return service.GetDistance();
    }
}
}
```

五十五. Collection expressions

1. 集合表达式使用

可以使用集合表达式来创建常见的集合值。集合表达式是一种简洁的语法，在计算时，可以分配给许多不同的集合类型。集合表达式在 [和] 括号之间包含元素的序列。

PS:

```
static void Main(string[] args)
{
    List<string> names1 = ["one", "two"];
    List<string> names2 = ["three", "four"];
    List<List<string>> names3 = [["one", "two"], ["three", "four"]];
    List<List<string>> names4 = [names1, names2];
}
}
```


2. 集合表达式解构

在 C#12 中通过..即可将一个集合解构，并将其作为另一个集合的元素。

PS:

```
string[] vowels = ["a", "e", "i", "o", "u"];
string[] consonants = ["b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
    "n", "p", "q", "r", "s", "t", "v", "w", "x", "z"];
string[] alphabet = [.. vowels, .. consonants, "y"];
```

求值时，分布元素 `..vowels` 产生五个元素：“a”、“e”、“i”、“o”和“u”。分布元素 `..consonants` 产生 20 个元素，即 `consonants` 数组中的数字。分布元素中的变量必须使用 `foreach` 语句进行枚举。如前面的示例所示，可以将分布元素与集合表达式中的单个元素组合在一起。

五十六. 提高 Unity 的 C#代码质量

1. 尽可能地使用属性，而不是可直接访问的数据成员

属性(property)一直是 C#语言中比较有特点的存在。属性**允许将数据成员作为共有接口的一部分暴露出去，同时仍旧提供面向对象环境下所需的封装**。属性这个语言元素可以让你像访问数据成员一样使用，但其底层依旧是使用方法实现的。

使用属性，可以非常轻松的在 `get` 和 `set` 代码段中加入检查机制。

需要注意，正因为属性是用方法实现的，所以它**拥有方法所拥有的一切语言特性**。

- ① 属性增加多线程的支持是非常方便的。我们可以加强 `get` 和 `set` 访问器（`accessors`）的实现来提供数据访问的同步。
- ② 属性可以被定义为 **virtual**。
- ③ 可以把属性扩展为 **abstract**。
- ④ 可以使用泛型版本的属性类型。
- ⑤ 属性也**可以定义为接口**。
- ⑥ 因为实现属性访问的方法 `get` 与 `set` 是独立的两个方法，在 C# 2.0 之后，你可以给它们定义不同的访问权限，来更好的控制类成员的可见性。
- ⑦ 为了和多维数组保持一致，我们**可以创建多维索引器**，在不同的维度上使用相同或不同类型。

无论何时，**需要在类型的公有或保护接口中暴露数据，都应该使用属性**。如果可以也应该使用索引器来暴露序列或字典。现在多投入一点时间使用属性，换来的是今后维护时的更加游刃有余。

2. 偏向于使用运行时常量而不是编译时常量

对于常量，C#里有两个不同的版本，即**运行时常量（readonly）**和**编译时常量（const）**。

应该尽量使用运行时常量，而不是编译器常量。虽然编译器常量略快，但并没有运行时常量那么灵活。应**仅仅在那些性能异常敏感，且常量的值在各个版本之间绝对不会变化时，再使用编译时常量**。

编译时常量与运行时常量不同之处表现在于他们的访问方式不同，因为 `ReadOnly` 值是运行时解析的：

编译时常量（`const`）的值会被目标代码中的值直接取代。

运行时常量（`readonly`）的值是在运行时进行求值。引用运行时生成的 IL 将引用到 `readonly` 变量，而不是变量的值。

这个差别就带来了如下规则：

编译时常量（const）仅能用于数值和字符串。

运行时常量（readonly）可以为任意类型。运行时常量**必须在构造函数或初始化器中初始化**，因为在构造函数执行后不能再被修改。你可以让某个 `readonly` 值为一个 `DateTime` 结

构，而不能指定某个 `const` 为 `DataTime`。

可以用 `readonly` 值保存实例常量，为类的每个实例存放不同的值。而编译时常量就是静态的常量。

有时候你需要让某个值在编译时才确定，就最好是使用运行时常量（`readonly`）。

标记版本号的值就应该使用运行时常量，因为它的值会随着每个不同版本的发布而改变。

const 优于 readonly 的地方仅仅是性能，使用已知的常量值要比访问 `readonly` 值略高一点，不过这其中的效率提升，可以说是微乎其微的。

综上，在编译器必须得到确定数值时，一定要使用 `const`。例如特性（`attribute`）的参数和枚举的定义，还有那些在各个版本发布之间不会变化的值。除此之外的所有情况，都应尽量选择更加灵活的 `readonly` 常量。

3. 推荐使用 `is` 或 `as` 操作符而不是强制类型转换

C#中，`is` 和 `as` 操作符的用法概括如下。

1) `is`

检查一个对象是否兼容于其他指定的类型,并返回一个 `Bool` 值,永远不会抛出异常。

2) `as`

作用与强制类型转换是一样,但是永远不会抛出异常,即如果转换不成功,会返回 `null`。

尽可能的使用 `as` 操作符，因为相对于强制类型转换来说，`as` 更加安全，也更加高效。

`as` 在转换失败时会返回 `null`，在转换对象是 `null` 时也会返回 `null`，所以使用 `as` 进行转换时，只需检查返回的引用是否为 `null` 即可。

`as` 和 `is` 操作符都不会执行任何用户自定义的转换，它们仅当运行时类型符合目标类型时才能转换成功，也不会在转换时创建新的对象。

`as` 运算符对值类型是无效，此时可以使用 `is`，配合强制类型转换进行转换。

仅当不能使用 `as` 进行转换时，才应该使用 `is` 操作符。否则 `is` 就是多余的。

4. 推荐使用条件特性而不是 `#if` 条件编译

由于 `#if/#endif` 很容易被滥用，使得编写的代码难于理解且更难于调试。C#为此提供了——**条件特性**(`Conditional attribute`)。使用条件特性可以将函数拆分出来，让其只有在定义了某些环境变量或设置了某个值之后才能编译并成为类的一部分。`Conditional` 特性最常用的地方就是将一段代码变成调试语句。

`Conditional` 特性只可应用在整个方法上，另外，**任何一个使用 `Conditional` 特性的方法都只能返回 `void` 类型，且不能在方法内的代码块上应用 `Conditional` 特性**。也不可以在有返回值的方法上应用 `Conditional` 特性。但**应用了 `Conditional` 特性的方法可以接受任意数目的引用类型参数**。

使用 `Conditional` 特性生成的 IL 要比使用 `#if/#endif` 时更有效率。同时，将其限制在函数层面上可以更加清晰地将条件性的代码分离出来，以便进一步保证代码的良好结构。

5. 理解几个等同性判断之间的关系

C#中可以创建两种类型：**值类型和引用类型**。

如果**两个引用类型的变量指向的是同一个对象**，它们将被认为是“**引用相等**”。如果**两个值类型的变量类型相同，而且包含同样的内容**，它们被认为是“**值相等**”。这也是等同性判断需要如此多方法的原因。

当我们创建自己的类型时（无论是类还是 `struct`），应为类型定义“等同性”的含义。

C#提供了 4 种不同的函数来判断两个对象是否“相等”。

`public static bool ReferenceEquals(object left, object right)`;判断两个不同变量的对象标识（`object identity`）是否相等。无论比较的是引用类型还是值类型，该方法判断的依据都是对

象标识，而不是对象内容。

`public static bool Equals (object left, object right);` 用于判断两个变量的运行时类型是否相等。

`public virtual bool Equals(object right);` 用于重载

`public static bool operator ==(MyClass left, MyClass right);` 用于重载

不应该覆写 `Object.Equals()` 静态方法和 `Object.Equals()` 静态方法，因为它们已经完美的完成了所需要完成的工作，提供了正确的判断，并且该判断与运行时的具体类型无关。

对于值类型，我们**应该总是覆写 `Object.Equals()` 实例方法和 `operator==()`**，以便为其提供效率更高的等同性判断。对于引用类型，**仅当你认为相等的含义并非是对象标识相等时**，才需要覆写 `Object.Equals()` 实例方法。在覆写 `Equals()` 时也要实现 `IEquatable<T>`。

6. 了解 GetHashCode() 的一些坑

`GetHashCode()` 方法在使用时会有不少坑，要谨慎使用。

`GetHashCode()` 函数仅会在一个地方用到，即为基于散列(hash)的集合定义键的散列值时，此类集合包括 `HashSet<T>` 和 `Dictionary<K,V>` 容器等。对引用类型来讲，索然可以正常工作，但是效率很低。对值类型来讲，基类中的实现有时甚至不正确。而且，编写的自己 `GetHashCode()` 也不可能既有效率又正确。

在.NET 中，每个对象都有一个散列码，其值由 `System.Object.GetHashCode()` 决定。

实现自己的 `GetHashCode()` 时，要遵循下述三条原则：

- ① 如果**两个对象相等**（由 `operator==` 定义），那么他们**必须生成相同的散列码**。否则，这样的散列码将无法用来查找容器中的对象。
- ② 对于任何一个对象 A，**`A.GetHashCode()` 必须保持不变**。
- ③ 对于所有的输入，散列函数应该在所有整数中按随机分别生成散列码。这样散列容器才能得到足够的效率提升。

7. 理解短小方法的优势

将 C# 代码翻译成可执行的机器码需要两个步骤。C# 编译器将生成 IL，并放在程序集中。随后，JIT 将根据需要逐一为方法（或是一组方法，如果涉及内联）生成机器码。短小的方法让 JIT 编译器能够更好地分摊编译的代价，且短小的方法也更适合内联。

除了短小之外，简化控制流程也很重要。控制分支越少，JIT 编译器也会更容易地找到最适合放在寄存器中的变量。

所以，短小方法的优势，并不仅体现在代码的可读性上，还关系到程序运行时的效率。

8. 选择变量初始化而不是赋值语句

成员初始化器是保证类型中成员均被初始化的最简单的方法——无论调用的是哪一个构造函数。**初始化器将在所有构造函数执行之前执行**，使用这种语法也就保证了我们不会在添加新的构造函数时遗漏掉重要的初始化代码。

综上，**若是所有的构造函数都要将某个成员变量初始化成同一个值，那么应该使用初始化器**。

9. 正确地初始化静态成员变量

C# 提供了有静态初始化器和静态构造函数来专门用于静态成员变量的初始化。

静态构造函数是一个特殊的函数，将在其他所有方法执行之前以及变量或属性被第一次访问之前执行。可以用这个函数来初始化静态变量，实现单例模式或执行类可用之前必须进行的任何操作。

和实例初始化一样，也可以使用初始化器语法来替代静态的构造函数。若只是需要为某

个静态成员分配空间，那么不妨使用初始化器的语法。而若是要更复杂一些的逻辑来初始化静态成员变量，那么可以使用静态构造函数。

使用静态构造函数而不是静态初始化器最常见的理由就是处理异常。在使用静态初始化器时，我们无法自己捕获异常，而在静态构造函数中却可以做到。

10. 使用构造函数链（减少重复的初始化逻辑）

编写构造函数很多时候是个重复性的劳动，如果你发现多个构造函数包含相同的逻辑，可以将这个逻辑提取到一个通用的构造函数中。这样既可以避免代码重复，也可以利用构造函数初始化器来生成更高效的目标代码。

C#编译器将把构造函数初始化器看做是一种特殊的语法，并移除掉重复的变量初始化器以及重复的基类构造函数调用。这样使得最终的对象可以执行最少的代码来保证初始化的正确性。

构造函数初始化器允许一个构造函数去调用另一个构造函数。而 C# 4.0 添加了对默认参数的支持，这个功能也可以用来减少构造函数中的重复代码。你可以将某个类的所有构造函数统一成一个，并为所有的可选参数指定默认值。其他的几个构造函数调用某个构造函数，并提供不同的参数即可。

11. 实现标准的销毁模式

GC 可以高效地管理应用程序使用的内存，不过创建和销毁堆上的对象仍旧需要时间。若是在某个方法中创建了太多的引用对象，将会对程序的性能产生严重的影响。

这里有一些规则，可以帮我们尽量降低 GC 的工作量：

- ① 若某个引用类型（值类型无所谓）的局部变量用于被频繁调用的例程中，那么应该将其提升为成员变量。
- ② 为常用的类型实例提供静态对象。
- ③ 创建不可变类型的最终值。比如 `string` 类的 `+=` 操作符会创建一个新的字符串对象并返回，多次使用会产生大量垃圾，不推荐使用。对于简单的字符串操作，推荐使用 `string.Format`。对于复杂的字符串操作，推荐使用 `StringBuilder` 类。

12. 区分值类型和引用类型

C# 中，`class` 对应引用类型，`struct` 对应值类型。

C# 不是 C++，不能将所有类型定义成值类型并在需要时对其创建引用。C# 也不是 Java，不像 Java 中那样所有的东西都是引用类型。我们必须在创建时就决定类型的表现行为，这相当重要，因为稍后的更改可能带来很多灾难性的问题。

值类型无法实现多态，因此其最佳用途就是存放数据。引用类型支持多态，因此用来定义应用程序的行为。

一般情况下，我们习惯用 `class`，随意创建的大都是引用类型，若下面几点都是肯定回答，那么就应该创建 `struct` 值类型：

- ① 该类型主要职责是否在于数据存储？
- ② 该类型的公有接口都是由访问其数据成员的属性定义的吗？
- ③ 确定该类型绝不会有派生类型吗？
- ④ 确定该类型永远都不需要多态支持吗？

用值类型表示底层存储数据的类型，用引用类型来封装程序的行为。这样，我们就可以保证类暴露出的数据能以复制的形式安全提供，也能得到基于栈存储和使用内联方式存储带来的内存性能提升，更可以使用标准的面向对象技术来表达应用程序的逻辑。而倘若对类型未来的用图不确定，那么应该选择引用类型。

13. 保证 0 为值类型的有效状态

在创建自定义枚举值时，请确保 0 是一个有效的选项。若你定义的是标志(flag)，那么可以将 0 定义为没有选中任何状态的标志（比如 None）。即作为标记使用的枚举值（即添加了 Flags 特性）应该总是将 None 设置为 0。

14. 保证值类型的常量性和原子性

常量性的类型使得我们的代码更加易于维护，不要盲目地为类型中的每一个属性都创建 get 和 set 访问器。对于那些**目的是存储数据的类型**，应该**尽可能地保证其常量性和原子性**。

15. 限制类型的可见性

在**保证类型可以完成其工作的前提下**，我们应该**尽可能地给类型分配最小的可见性**。也就是，仅仅暴露那些需要暴露的，尽量使用较低可见性的类来实现公有接口。可见性越低，能访问你功能的代码越少，以后可能出现的修改也就越少。

16. 通过定义并实现接口替代继承

理解抽象基类（abstract class）和接口（interface）的区别。

接口是一种契约式的设计方式，一个实现某个接口的类型，必须实现接口中约定的方法。抽象基类则为的一组相关的类型提供了一个共同的抽象，也就是说**抽象基类描述了对象是什么，而接口描述了对象将如何表现其行为**。

接口不能包含实现，也不能包含任何具体的数据成员。而抽象基类可以为派生类提供一些具体的实现。

基类描述并实现了一组相关类型间共用的行为。接口则定义了一组具有原子性的功能，供其他不相关的具体类型来实现。

理解好两者之间的差别，我们便可以创造更富表现力、更能应对变化的设计。使用**类层次来定义相关的类型**，用**接口暴露功能**，并让不同的类型实现这些接口。

17. 理解接口方法和虚方法的区别

第一眼看来，实现接口和覆写虚方法似乎没有什么区别，实际上，实现接口和覆写虚方法之间的差别很大。

接口中声明的成员方法默认情况下并非虚方法，所以，派生类不能覆写基类中实现的非虚接口成员。若要覆写的话，将接口方法声明为 virtual 即可。

基类可以为接口中的方法提供默认的实现，随后，派生类也可以声明其实现了该接口，并从基类中继承该实现。

实现接口拥有的选择要比创建和覆写虚方法多。我们可以为类层次创建密封（sealed）的实现，虚实现或者抽象的契约。还可以创建密封的实现，并在实现接口的方法中提供虚方法进行调用。

18. 用委托实现回调

在 C# 中，回调是用**委托**来实现的，主要要点如下：

委托为我们提供了类型安全的回调定义。虽然大多数常见的委托应用都和事件有关，但这并不是 C# 委托应用的全部场合。当类之间有通信的需要，并且我们期望一种比接口所提供的更为松散的耦合机制时，委托便是最佳的选择。

委托允许我们在运行时配置目标并通知多个客户对象。委托对象中包含一个方法的应用，该方法可以是静态方法，也可以是实例方法。也就是说，使用委托，我们可以和一个或多个在运行时联系起来的客户对象进行通信。

由于回调和委托在 C# 中非常常用，以至于 C# 特地以 lambda 表达式的形式为其提供了精简语法。

由于一些历史原因，.NET 中的委托都是多播委托（multicast delegate）。多播委托调用

过程中，每个目标会被依次调用。委托对象本身不会捕捉任何异常。因此，任何目标抛出的异常都会结束委托链的调用。

19. 用事件模式实现通知

事件提供了一种标准的机制来通知监听者，而 **C#中的事件其实就是观察者模式的一个语法上的快捷实现**。

事件是一种内建的委托，用来为事件处理函数提供类型安全的方法签名。任意数量的客户对象都可以将自己的处理函数注册到事件上，然后处理这些事件，这些客户对象无需在编译器就给出，事件也不必非要有订阅者才能正常工作。

在 C#中使用事件可以降低发送者和可能的通知接受者之间的耦合，发送者可以完全独立于接受者进行开发。

20. 避免返回对内部类对象的引用

若将引用类型通过公有接口暴露给外界，那么对象的使用者即可绕过我们定义的方法和属性来更改对象的内部结构，这会导致常见的错误。

共有四种不同的策略可以防止类型内部的数据结构遭到有意或无意的修改：

- ① 值类型。当客户代码通过属性来访问值类型成员时，**实际返回的是值类型的对象副本**。
- ② 常量类型。如 `System.String`。
- ③ 定义接口。将客户对内部数据成员的访问限制在一部分功能中。
- ④ 包装器（wrapper）。提供一个包装器，仅暴露该包装器，从而限制对其中对象的访问。

21. 仅用 new 修饰符处理基类更新

使用 `new` 操作符修饰类成员可以重新定义继承自基类的非虚成员。

new 修饰符只是用来解决升级基类所造成的基类方法和派生类方法冲突的问题。

`new` 操作符必须小心使用。若随心所欲的滥用，会造成对象调用方法的二义性。

22. 尽量减少在公有 API 中使用动态对象

（这条准则在 Unity 中请忽略，因为 Unity 版本的 mono 并没有实现 .NET 4.0 中的 `dynamic language runtime`）

动态对象有一些“强制性”，即所有打交道的对象都变成动态的。若是某个操作的某个参数是动态的，那么其结果也会是动态的。若是每个方法返回了一个动态对象，那么所有使用该对象的地方也变成了动态对象。

所以，若你要在程序中使用动态特性，请尽量不要在公有接口中使用，这样可以将动态类型现在在一个单独的对象或类型中。

应该将动态对象限制在最需要的地方，然后立即将动态对象转换为静态类型。当我们的代码依赖于其他环境中创建的动态类型时，可以用专门的静态类型封装这些动态对象，并提供静态的公有接口。

五十七. 代码整洁之道

1. 整洁代码的命名法则

要点一：要名副其实。一个好的变量、函数或类的名称应该已经答复了所有的大问题。一个好名称可以大概告诉你这个名称所代表的内容，为什么会存在，做了什么事情，应该如何用等。

要点二：要避免误导。我们应该避免留下隐藏代码本意的错误线索，也应该**避免使用与本意相悖的词**。

要点三：尽量做有意义的区分。尽量避免使用数字系列命名（`a1`、`a2`……`aN`）和没有

意义的区分。

要点四：尽量使用读得出来的名称。如名称读不出来，讨论的时候会不方便且很尴尬。

要点五：尽量使用可搜索的名称。名称长短应与其作用域大小相对应，若变量或常量可能在代码中多处使用，应赋予其以便于搜索的名称。

要点六：取名不要绕弯子。取名要直白，要直截了当，明确就是王道。

要点七：类名尽量用名词。类名尽量用名词或名词短语，最好不要是动词。

要点八：方法名尽量用动词。方法名尽量用动词或动词短语。

要点九：每个概念对应一词，并一以贯之。对于那些会用到你代码的程序员，一以贯之的命名法简直就是天降福音。

要点十：通俗易懂。应尽力写出易于理解的变量名，要把代码写得让别人能一目了然，而不必让人去非常费力地去揣摩其含义。

要点十一：尽情使用解决方案领域专业术语。尽管去用那些计算机科学领域的专业术语、算法名、模式名、数学术语。

要点十二：要添加有意义的语境。需要用有良好命名的类，函数或名称空间来放置名称，给读者提供语境。若没能提供放置的地方，还可以给名称添加前缀。

2. 整洁代码的函数书写准则

第一原则：短小。若没有特殊情况，最好将单个函数控制在十行以内。

第二原则：单一职责。函数应该只做一件事情。只做一件事，做好这件事。

第三原则：命名合适且具描述性。长而具有描述性的名称，比短而令人费解的名称好。当然，如果短的名称已经足够说明问题，还是越短越好。

第四原则：参数尽可能少。最理想的函数参数形态是零参数，其次是单参数，再次是双参数，应尽量避免三参数及以上参数的函数，有足够的理由才能用三个以上参数。

第五原则：尽力避免重复。重复的代码会导致模块的臃肿，整个模块的可读性可能会应该重复的消除而得到提升。

3. 整洁代码的格式准则

第一原则：像报纸一样一目了然。优秀的源文件也要像报纸文章一样，名称应当简单并且一目了然，名称本身应该足够告诉我们是否在正确的模块中。源文件最顶部应该给出高层次概念和算法。细节应该往下渐次展开，直至找到源文件中最底层的函数和细节。

第二原则：恰如其分的注释。带有少量注释的整洁而有力的代码，比带有大量注释的零碎而复杂的代码更加优秀。

第三原则：合适的单文件行数。尽可能用几百行以内的单文件来构造出出色的系统，因为短文件通常比长文件更易于理解。

第四原则：合理地利用空白行。在每个命名空间、类、函数之间，都需用空白行隔开。

第五原则：让紧密相关的代码相互靠近。靠近的代码行暗示着他们之间的紧密联系。所以，紧密相关的代码应该相互靠近。

第六原则：基于关联的代码分布。

① 变量的声明应尽可能靠近其使用位置。

② 循环中的控制变量应该在循环语句中声明。

③ 短函数中的本地变量应当在函数的顶部声明。

④ 对于某些长函数，变量也可以在某代码块的顶部，或在循环之前声明。

⑤ 实体变量应当在类的顶部声明。

⑥ 若某个函数调用了另一个，就应该把它们放到一起，而且调用者应该尽量放到被调用者上面。

⑦ 概念相关的代码应该放到一起。相关性越强，则彼此之间的距离就该越短。

第七原则：团队遵从同一套代码规范。一个好的团队应当约定与遵从一套代码规范，并且每个成员都应当采用此风格。

4. 整洁类的书写准则

原则一：合理地分布类中的代码。类中代码的分布顺序大致是：

- ① 公有静态常量
- ② 私有静态变量
- ③ 公有普通变量
- ④ 私有普通变量
- ⑤ 公共函数
- ⑥ 私有函数

原则二：尽可能地保持类的封装。尽可能使函数或变量保持私有，不对外暴露太多细节。

原则三：类应该短小，尽量保持单一权责原则。类或模块应有且只有一条加以修改的理由。

原则四：合理提高类的内聚性。我们希望类的内聚性保持在较高的水平。内聚性高，表示类中方法和变量相互依赖，相互结合成一个逻辑整体。

原则五：有效地隔离修改。类应该依赖于抽象，而不是依赖于具体细节。尽量对设计解耦，做好系统中的元素的相互隔离，做到更加灵活与可复用。

五十八. C#语法糖

1. 空值条件运算符

PS:

```
// 传统空值检查
if (person != null && person.Address != null)
{
    // 访问地址属性
}

// 空值条件运算符
if (person?.Address != null)
{
    // 访问地址属性
}
```

2. 空合并运算符

PS:

```
int age = (person != null) ? person.Age : -1;
// 使用空合并运算符
int age = person?.Age ?? -1;
```

3. 使用语句处理 IDisposable 对象

PS:

```
// 不使用 using 语句
FileStream fs = new FileStream("file.txt", FileMode.Open);
// 使用文件流
fs.Close();
// 使用 using 语句
using (FileStream fs = new FileStream("file.txt", FileMode.Open))
```

```
{
    // 使用文件流
}
```

五十九. 程序逻辑优化技巧

1. 使用 List 和 Dictionary 时提高效率

我们知道 **List**、**Dictionary** 的实质都是数组，**Dictionary** 有两个数组，一个数组存索引一个数组存数据，我们遍历 List 和 Dictionary 都是在遍历数组。

了解底层的逻辑有助于我们更好的运用它们，比如当我们使用 List 插入时，我们知道其实就是向数组中写入元素并遍历其后面的数据依次向后移动的过程。了解了这些，每次当我们使用 List 的 Insert 时都会注意一些。还有 **Contains** 函数，它是一个以遍历形式来寻找结果的函数，每次使用它都会从头到尾遍历一下直到寻找到结果，Remove 也是一样，都是以遍历的形式存在。如果我们在代码中使用它们的频率比较高，就会带来很多不必要的性能消耗，这是我们很多人常常不注意的。我们常常因为不了解或图方便而使用了这些接口而并没有考虑它们带来的性能损耗。

Dictionary 也有诸多问题，首先它是一个使用哈希冲突解决关键字的字典组件，因此哈希值与容器中数组的映射很关键，**GetHashCode** 获取哈希值的函数也比较关键。哈希冲突与数组大小有很大关系，数组越大哈希冲突率就越小，因此我们在写程序时应该注意设置 Dictionary 的初始大小，尽量设置一个合理的大小，而不是什么都不做，任由它自己扩容，这不但让哈希冲突变得频繁，而且扩容时数组的回收也加重了 GC 的负担。除了之外，在 C# 中所有类都继承自 Object 类，Dictionary 使用 Object 的 GetHashCode 来获取类实例的哈希值，而 **GetHashCode** 是用算法将内存地址转化为哈希值的过程，因此我们可以认为它只是一个算法，并没有做任何值做缓存，每次调用它都会计算一次哈希值，这是比较隐形的性能损耗。如果频繁使用 GetHashCode 作为关键字来提取 Value，那么我们应该关注下 GetHashCode 的算力损耗，是否可以用唯一 ID 的方式来代替 GetHashCode 算法。

2. 巧用 Struct

Struct 和 Class 的区别常常被人遗忘，Struct 结构体是值类型，它与 Class 不同的是 **Struct 传递时并不是靠引用(指针)形式来传递而是靠复制**，我们可以通俗的认为它是通过内存拷贝来实现传递(真实的情况是通过字节对齐规则循环多次复制内存)，也就是说我们在传递 Struct 时，其实是在不断的克隆数据。

Struct 这样的值类型对我们做性能优化有什么好处呢？首先如果 Struct 被定义于函数中的局部变量，则 Struct 的值类型变量分配的内存是在栈上的，栈是连续内存，并且在函数调用结束后，栈的回收是非常快速和简单的，只要将尾指针置零就可以了(并非真正意义上的释放内存)，这样我们即不会产生内存碎片，也不需要内存垃圾回收，CPU 读取数据对连续内存也非常友好高效。

除了上述这些，Struct 数组也对我们提高内存访问速度有所帮助。我们要明白由于 Struct 是值类型，所以它的内存也与值类型一样是连续的，Class 数组则只是引用(指针)变量空间连续，这是大不同的。连续内存存在 CPU 读取数据时，CPU 的缓存可以帮助我们提高命中率，因为 CPU 在读取内存时会把一个大块内容放入缓存，当下次读取时先从高速缓存中找，如果命中则不需要再向内存读取数据(高速缓存(cache)比内存快 100 倍)，而非连续内存则在缓存使用时的命中率比较低，因此 CPU 缓存命中率的高低很影响 CPU 效率。

但是也不是所有 Struct 都能提高缓存命中率，如果 Struct 太大超过了缓存拷贝的数据块，则缓存就不再起作用了，因为拷贝进去的数据只有 1 个甚至半个 Struct。于是就有很多架构

抛弃了 Struct，彻底使用值类型连续空间的方式来提高 CPU 缓存命中率，把所有的数值都集合起来用数组的形式存放，而具体对象上则只存放一个索引值，当需要存取时都通过索引来操作数组。

3. 尽可能的使用对象池

说到对象，我们应该清楚的明白，内存分配和消耗对我们程序的影响，这也是提高程序效率的关键所在。Unity3D 使用的 C# 语言，它使用垃圾回收机制回收内存，即使 Unity3D 在发布后将 C# 转为了 C++ 也依然使用垃圾回收机制来执行分配和销毁内存。作为高级程序员，我们应该要感受到，在创建类实例时内存分配的 CPU 耗时，以及垃圾回收时的艰难。

垃圾回收有多艰难呢？我们来解释一下，我们在 C# 中随意的 new 创建类实例，又不管它们的死活任意的丢弃或置空引用变量，类实例不断的被引用和间接引用，又不断被抛弃，垃圾回收器就要负责仔仔细细的收拾我们的烂摊子。内存不可能永远被分配而不回收，于是垃圾回收只能在内存不够用的时候去到处问和检查(意思是遍历所有已分配的内存块)，看看哪个类实例完全被遗弃了就捡回来(意思是完全没有被人引用了)，将内存回收掉。因此当我们业务逻辑越庞大数据量越多时，垃圾回收需要检查的内容也越来越多，如果回收后依然内存不足，就得向系统请求分配更多内存。

垃圾回收过程如此艰难，它每次回收时都会占用大量 CPU 算力，因此我们应该尽可能的用对象池来重复利用已经创建的对象，这有助于减少内存配合时的耗时，也减少堆内存的内存块数量，最终减少了垃圾回收时带来的 CPU 损耗。

除了 new 某个类创建内存导致的 GC 耗时增加外，以我的经验来看，很容易被我们忽略的是 new List 这种类型的使用，我们在平时编程时会大量使用动态数组，并且随时将它抛弃，因为它太容易使用以及丢弃了。类似的 Dictionary<int,List> 也是众多被忽略的内存分配消耗之一，被装进 Dictionary 字典中的 List 常被随意的丢弃(Remove 掉)，没去注意它是否能被再次利用。

C# 中一个简单的通用对象池就能解决问题，但我们常常嫌弃它，并觉得麻烦。在我的编程经验中，图方便图好用的往往要付出性能损耗的代价，而性能高的代码通常都有点反人性，我们应该尽量找到一个平衡点，即有高的代码可读性，也尽量不要被人性所驱使了去做一些图方便的事，这在任何时候都是非常有用的价值的。

4. 字符串导致的性能问题

字符串性能在大部分语言中都是比较难解决的问题，C# 中尤其如此。在 C# 中 string 是引用类型，每次我们动态创建一个 string 时 C# 都会在堆内存中分配一个内存用于存放字符串。我们来看看它到底有多恐怖。

PS:

```
string strA = "test";
for(int i = 0 ; i<100 ; i++)
{
    string strB = strA + i.ToString();
    string[] strC = strB.Split('e');
    strB = strB + strC[0];
    string strD = string.Format("Hello {0}, this is {1} and {2}.",strB, strC[0], strC[1]);
}
```

上面是一段恐怖的程序，循环中每次都会将 strA 字符串和 i 整数字符串连接，strB 所得到的值是从内存中新分配的字符串，然后将 strB 切割成两半成为了 strC，这两半又新分配了两段新的内存，再将 strB 与 strC[0] 连接起来，这又申请了一段内存，这段内存装上了 strB 和 strC[0] 两者连接的内容赋值给了 strB，strB 原来的内容因为没有变量指向就找不到了，

最后用 `string.Format` 的形式将 4 个字符串串联起来，新分配的内存中装有 4 者的连接内容。

这里要注意一点，**字符串常量是不会被丢弃的**，比如这段程序中“test”和“Hello {0}, this is {1} and {2}.”这两个常量，**它们常驻于内存，即使下次没有变量指向它们，它们也不回被回收，下次使用时也不需要重新分配内存**。原因我们放到下面计算机执行原理里去说。

每次循环中，都向内存申请了 5 次内存，并且抛弃了 1 次 `strA + i.ToString()` 的字符串内容，因为没有变量指向了这个字符串。这还不是最恐怖的，最恐怖的是，每次循环结束都会将前面所有分配的内存内容都抛弃，重新分配一次，就这样向不断抛弃和申请，总共向内存申请了 500 次内存段，并全部抛弃，内存被浪费的很厉害。

为什么会这样呢？究其原因是 C# 语言对字符串并没有任何缓存机制，每次使用都需要重新分配 `string` 内容，据我所知很多语言都没有字符串的缓存机制，因此**我们的字符串连接、切割、组合等操作都会向内存申请新的内容，并且抛弃没有变量指向的字符串，等待 GC 回收**。我们知道 **GC 一次会消耗很多 CPU**，如果我们**不注意字符串的问题，不断浪费内存将导致程序不定时卡顿**，并且随着程序运行时间越来越长，各程序模块不良代码的运行积累，程序卡顿次数会逐步提高，运行效率也将越来越差。

解决字符串问题有两个方法，第一是自建缓存机制，可以用一些标志性的 `Key` 键值来一一对应字符串，比如游戏项目中常用 `ID` 来构造某个字符串。

5. 字符串的隐藏问题

`string` 的 `Length` 方法可不是省油的灯，由于 `string` 类中只存放了字符串 `char[]` 数据，没有其他变量，因此 **`string.Length` 需要通过遍历字符串来获取长度**。

PS:

```
int sum = 0;
for( ; data[sum] != '\0' ; ++sum);
return sum;
```

如果我们大量使用 `Length` 就会浪费很多 CPU，因为它每次都会遍历整个字符串，特别是在循环中使用时。

PS:

```
string str = "Hello world.";
for(int i = 0 ; i < str.Length ; ++i)
{
    // do some thing.
}
```

上述代码就相当于两层循环，每次判断时都会重新遍历一遍字符串以获得长度。

字符串比较也有隐藏问题，当两个字符串比较时，`string` 首先会比较两个字符串的指针是否是一致的，一致则返回 `true`，如果**指针不一致则会遍历两者判断是否每个字符都是相等的**。

PS:

```
public bool Equals(String value) {
    if (this == null) //this is necessary to guard against
reverse-pinvokes and
        throw new NullReferenceException(); //other callers who do not use the callvirt
instruction

    if (value == null)
        return false;
```

```

    if (Object.ReferenceEquals(this, value))
        return true;

    if (this.Length != value.Length)
        return false;

    return EqualsHelper(this, value); //遍历两者的字符
}

```

倘若我们操作后的两个字符串来自不同的内存段，那么它们在比较是否相等时就会遍历所有字符来判定是否相等。

六十. C#其他知识点

1. HashMap 和 Hashtable 的区别

HashMap 是 Hashtable 的轻量级实现（非线程安全的实现），他们都完成了 Map 接口，主要区别在于 HashMap 允许空 (null) 键值 (key)，由于非线程安全，效率上可能高于 Hashtable。

2. 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的实例所占的内存
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	可以。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？	可以，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）	可以
这个类型的变量什么时候消亡？	当越出定义的作用域时。	当托管堆被垃圾回收时。

1) for 循环和 foreach 差别

一般情况下，使用 foreach 循环的性能要高出普通的 for 循环 20% 左右。

foreach 只能用于遍历，不能更改循环目标，遍历速度快，执行效率高。for 循环可以用于任何形式的重复行为，在循环体中可以进行任何操作，遍历速度慢，执行效率低。

a. foreach 循环的优势

- ① foreach 语句简洁。
- ② 效率比 for 要高(C#是强类型检查, **for 循环对于数组访问的时候, 要对索引的有效值进行检查**)。
- ③ 不用关心数组的起始索引是几(因为有很多开发者是从其他语言转到 C#的, 有些语言的起始索引可能是 1 或者是 0)。
- ④ 处理多维数组(不包括锯齿数组)更加的方便。
- ⑤ 在类型转换方面 **foreach 不用显示地进行类型转换**。
- ⑥ 当集合元素如 List<T>等在使用 foreach 进行循环时, 每循环完一个元素, 就会释放对应的资源。

b. foreach 循环的劣势

- ① 上面说了 foreach 循环的时候会释放使用完的资源, 所以会造成额外的 gc 开销, 使用的时候请酌情考虑。
- ② **foreach 也称为只读循环**, 所以在循环数组/集合的时候, 无法对数组/集合进行修改。
- ③ 数组中的每一项必须与其他的项类型相等。

2)Foreach 遍历原理

任何集合类 (Array) 对象都有一个 GetEnumerator()方法, 该方法可以返回一个实现了 IEnumerator 接口的对象。

这个返回的 IEnumerator 对象既不是集合类对象, 也不是集合的元素类对象, 它是一个独立的类对象。

通过这个实现了 IEnumerator 接口对象 A, 可以遍历访问集合类对象中的每一个元素对象。对象 A 访问 MoveNext 方法, 方法为真, 就可以访问 Current 方法, 读取到集合的元素。

3)GameObject a=new GameObject() GameObject b=a **实例化出来了 A, 将 A 赋给 B, 现在将 B 删除, 问 A 还存在吗?**

存在; a 引用地址在线程栈中, 数据内容在托管堆中, b 引用地址在线程栈中, 数据内容指向 A 的托管堆中的内容, B 删除, 只是删除 b 的引用地址。

4)什么是虚函数? 什么是抽象函数?

虚函数: 没有实现的, 可由子类继承并重写的函数。

抽象函数: 规定其非虚子类必须实现的函数, 必须被重写。

5)HashMap 和 Hashtable 的区别

HashMap 是 Hashtable 的轻量级实现 (非线程安全的实现), 他们都完成了 Map 接口, 主要区别在于 **HashMap 允许空键值**, 由于非线程安全, 效率上可能高于 Hashtable。

6)在类的构造函数前加上 static 会报什么错?为什么?

在构造函数中, 如果有 public 修饰的静态构造函数时会报: “静态构造函数中不允许出现访问修饰符”, 如果什么修饰符都不加的话不会报错, 静态构造函数一般是起初始化作用。

7)请简述关键字 Sealed 用在类声明和函数声明时的作用

Sealed 访问修饰符**用于类**时, 该类是密封类, 可**防止其他类继承此类**。

在方法中使用则可防止派生类重写此方法。

8)C#的委托是什么？有何用处？

委托类似于一种安全的指针引用，在使用它时是当做类来看待而不是一个方法，相当于一组方法的列表的引用。用处：使用委托使程序员可以将方法引用封装在委托对象内。然后可以将该委托对象传递给可调用所引用方法的代码，而不必在编译时知道将调用哪个方法。与 C 或 C++ 中的函数指针不同，委托是面向对象，而且是类型安全的。

9)什么是偏类？

C# 语言中有一个特性是将单个类文件分成多个物理文件。为此，我们必须使用“partial”关键字。在编译时，它在逻辑上只是一个文件；我们不能在两个不同的分部类文件中拥有同名的方法或同名的变量。在这里，为了方便开发者将大类文件分解成多个小的物理文件，提供了这个功能。

10) HashMap 为什么是线程不安全的

HashMap 是线程不安全的主要原因有两个。

首先，HashMap 的实现是基于数组和链表（或红黑树）的数据结构。当多个线程同时对 HashMap 进行修改操作时，可能会导致数据结构的破坏，比如链表断裂、循环链表等，从而导致数据的丢失或错误。

其次，HashMap 的 put 和 get 操作都不是原子操作，它们涉及到多个步骤，比如计算 hash 值、定位数组索引、处理冲突等。当多个线程同时对 HashMap 进行 put 或 get 操作时，可能会导致数据的覆盖或读取错误。

为了保证线程安全，可以使用线程安全的 ConcurrentHashMap 或者在操作 HashMap 时使用同步控制（如使用 synchronized 关键字或者使用 Collections.synchronizedMap 方法包装 HashMap）。