Codifica e operazioni bitwise

Codifica dei caratteri e operazioni bit a bit su valori interi

Tutorial di supporto alla realizzazione del progetto “Controllo errori”

SISTEMI 5^a

Anno 2013/2014

Ambiente: .NET 2.0+/C# 2.0+

Indice generale

1 Visualizzare i tipi integrali in notazione binaria

La classe **Convert** definisce molti metodi di conversione tra tipi di dati, tra i quali il metodo **ToString()**, che converte in stringa. Esiste in varie versioni, una delle quali consente di utilizzare una base numerica a scelta.

> **Convert.ToString(valore, base)**
>
> La conversione in una base a scelta è applicabile soltanto a un sottoinsieme de tipi integrali. Sono esclusi: **bool**, **char**, **byte**, **ushort**, **uint**, **ulong**.

Il seguente codice converte e visualizza un byte e un intero.

```
byte b = 10;
int i = 127;
string sb = Convert.ToString(b, 2);
string si = Convert.ToString(i, 2);
Console.WriteLine("BYTE {0,3}   Bin: {1}",b, sb);
Console.WriteLine("INT  {0,3}   Bin: {1}", i, si);
```

Output

```
BYTE  10   Bin: 1010
INT  127   Bin: 1111111
```

1.1 Ottenere il numero di cifre desiderato: PadLeft()

ToString() produce una stringa contenente il numero di cifre necessarie per rappresentare il valore convertito. Per ottenere un numero predefinito di cifre è possibile aggiungere gli zeri mancanti mediante il metodo **PadLeft()** della classe string.

PadLeft() consente di "riempire a sinistra" una stringa con un carattere a scelta, fino a raggiungere il numero massimo di caratteri desiderato.

Ad esempio, il seguente codice produce la stringa "00001111" a partire dalla stringa "1111":

```
string si = "1111";
string so = si.PadLeft(8, '0');   // aggiunge '0' fino a ottenere 8 caratteri in totale
Console.WriteLine(so);
```

Nota bene: **PadLeft()** si applica alla variabile stringa originale e produce una nuova stringa.

Usando **PadLeft()** si può produrre la rappresentazione binaria effettiva di variabili int, byte, etc.

```
byte b = 100;
int i = -1;
uint ui = 32767;
char c = '0';
```

```
string sb = Convert.ToString(b, 2).PadLeft(8, '0');
string si = Convert.ToString(i, 2).PadLeft(32, '0');
string sui = Convert.ToString(ui, 2).PadLeft(32, '0');
string sc = Convert.ToString(c, 2).PadLeft(16, '0');
Console.WriteLine("BYTE  {0,5}   Bin: {1}", b, sb);
Console.WriteLine("INT   {0,5}   Bin: {1}", i, si);
Console.WriteLine("UINT  {0,5}   Bin: {1}", ui, sui);
Console.WriteLine("CHAR  {0,5}   Bin: {1}", c, sc);
```

Output

```
BYTE    100   Bin: 01100100
INT      -1   Bin: 11111111111111111111111111111111
UINT  32767   Bin: 00000000000000000111111111111111
CHAR      0   Bin: 0000000000110000
```

Nota bene, il codice: contiene istruzioni come la seguente:

```
string sb = Convert.ToString(b, 2).PadLeft(8, '0');
```

contenente l'esecuzione concatenata di due metodi. In sostanza, il precedente codice può essere riscritto come:

```
string tmp = Convert.ToString(b, 2)
string sb = tmp.PadLeft(8, '0');
```

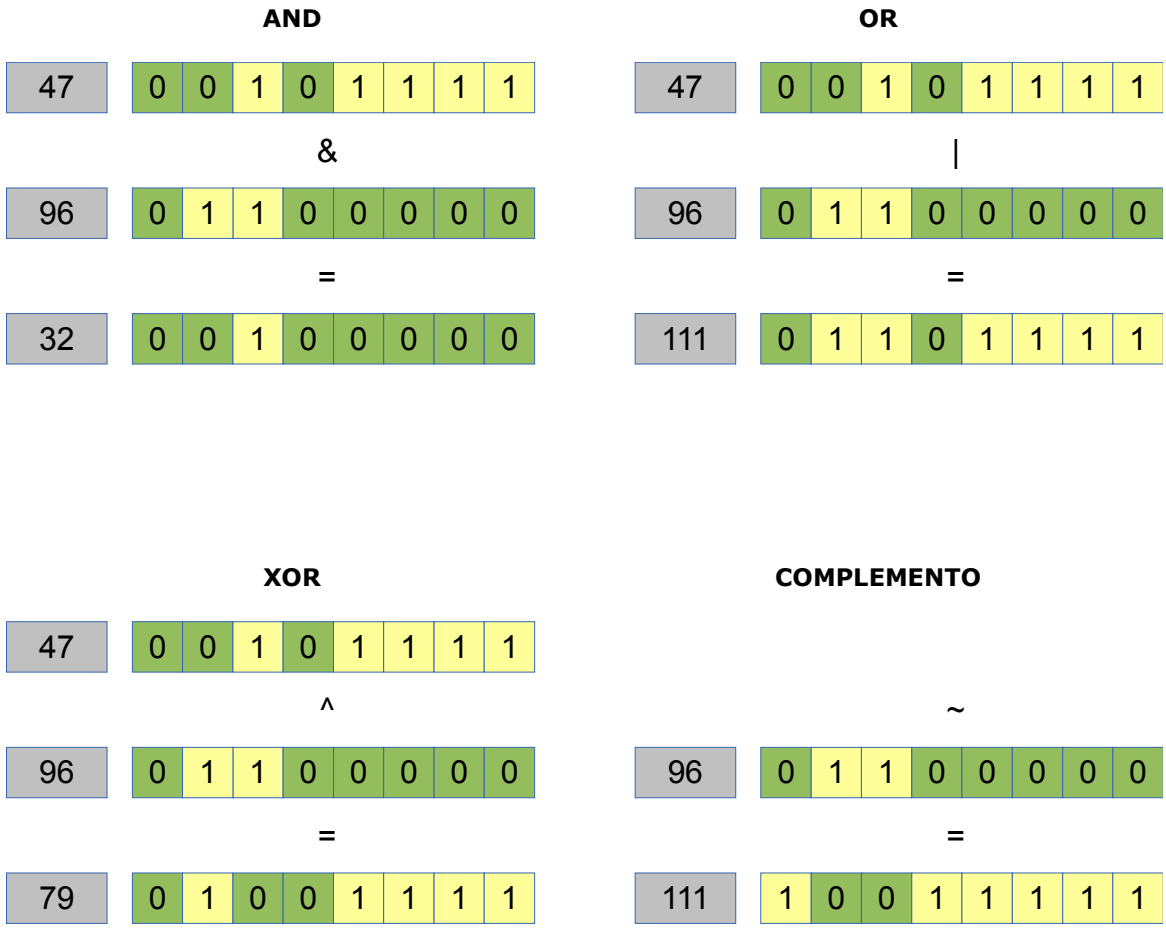
2 Operazioni "*bitwise*"

Gli operatori "*bit a bit*" considerano gli operandi delle semplici sequenze di bit e sono applicabili soltanto ai tipi integrali[1].

Operatore	Sintassi	Descrizione
&	val1 & val2	Esegue l'AND sui bit degli operandi.
|	val1 | val2	Esegue l'OR sui bit degli operandi.
^	val1 ^ val2	Esegue lo XOR sui bit degli operandi.
~	~ val	Esegue il complemento dell'operando

&, | e ^ agiscono sui bit corrispondenti dei due operandi, producendo un terzo bit come risultato. L'operatore ~ nega i bit dell'operando e produce dunque il complemento a 1.

2.1 Esempi di operazioni bitwise



1 &, | e ^ sono applicabili anche a operandi bool, ma in questo caso agiscono come operatori logici.

2.2 Mascherare i bit

Gli operatori *bitwise*, soprattutto &, sono particolarmente utili nelle ***operazioni di mascheramento***. Queste consentono di analizzare un sottoinsieme dei bit di una sequenza allo scopo di conoscerne il valore.

2.2.1 Ottenere il valore di un bit specifico

Data un byte si supponga di voler conosce il valore del secondo bit di peso maggiore.

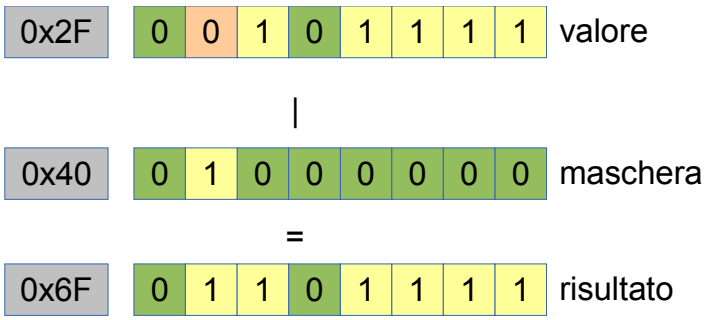
Per farlo si utilizza una "maschera" e cioè un valore intero contenente tutti zero e il bit corrispondente a 1. Quindi si esegue un AND tra il valore e la maschera. Il risultato è uguale alla maschera se il bit da verificare è 1, è zero altrimenti.

Il codice seguente dimostra l'operazione descritta:

```
byte b = 0x6F;      // 01101111
byte mask = 0x40;   // 01000000
if ((b & mask) == mask)
    Console.WriteLine("Settimo bit = 1");
else
    Console.WriteLine("Settimo bit = 0");
```

2.2.2 Settare un bit specifico

Si supponga di voler impostare a 1 il settimo bit di una variabile. Per farlo si utilizza una "maschera" contenente tutti zero e il bit corrispondente a 1. Quindi si esegue un OR tra la variabile e maschera, assegnando il risultato nuovamente alla variabile.



Il codice seguente dimostra l'operazione descritta:

```
byte b = 0x2F;          // 00101111
byte mask = 0x40;       // 01000000
b = (byte) (b | mask);  // 01101111
```

3 Operazioni di shift

Gli operatori di ***shift*** consentono di spostare verso sinistra o verso destra i bit di un valore intero.

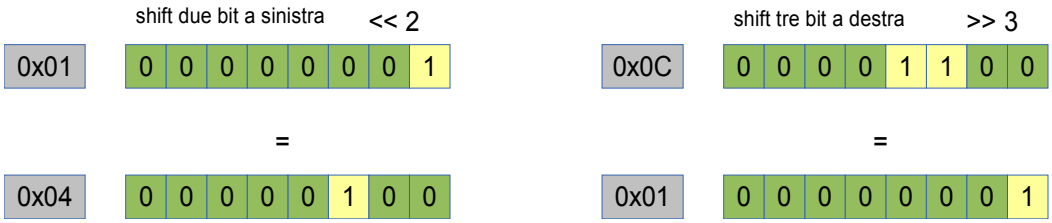
Da un punto di vista matematico, lo *shift* a sinistra equivale a moltiplicare il valore per due per ogni posizione "shiftata"; lo *shift* a destra equivale a dividere il valore per due per ogni posizione "shiftata".

3.1 Operatori di shift

Gli operatori di *shift* sono applicabili a operandi di tipo int, uint, long, ulong e producono un risultato del tipo corrispondente.

Operatore	Sintassi	Descrizione
<<	val << n	Sposta i bit di "val" a sinistra di "n" posizioni.
>>	val >> n	Sposta i bit di "val" a destra di "n" posizioni.

Segue l'esempio di operazioni di shift a sinistra e a destra.



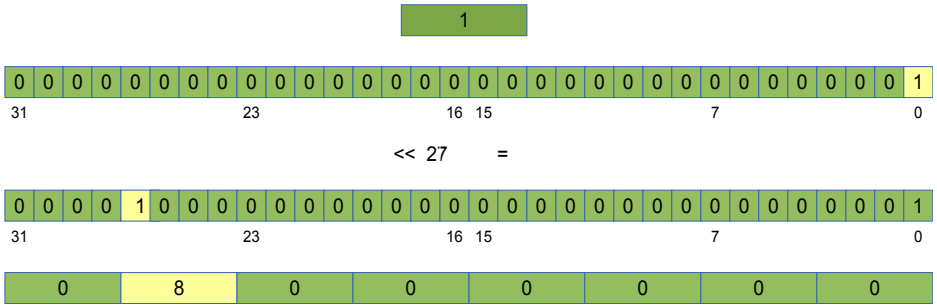
Nota bene: un operazione di *shift* non produce mai un overflow; i bit che eccedono il valore (sia a sinistra che a destra) vengono semplicemente scartati.

3.2 Creazione di maschere con gli operatori di *shift*

Le operazioni di *shift* sono comode per creare delle maschere di bit.

Si ipotizzi di voler creare una maschera (intera) che abbia il bit 27 a uno e tutti gli altri a zero. Specificare la maschera come costante esadecimale richiede qualche calcolo, mentre usando l'operatore `<<` diventa tutto estremamente semplice:

```
int mask = 1 << 27;                    // genera 0x08000000
```



4 Codifica dei caratteri

Nel linguaggio C# i caratteri esistono sia come valori a se stanti (tipo char) che come elementi di una stringa. Le seguenti istruzioni sono assolutamente legittime:

```
string s = "Ciao!";
char ch = s[4];                          // assegna '!' a ch
char[] cars = { 'C', 'i', 'a', 'o', '!' };
cars[0] = ch;                            // assegna '!' al primo elemento di cars
```

In tutti i casi, ogni singolo carattere viene rappresentato in memoria mediante una valore numerico di uno o più byte, in base al tipo di codifica utilizzata. In C# sono disponibili i seguenti tipi di codifica:

Codifica	Descrizione
ASCII	Codifica i caratteri con un valore a 7 bit ("ASCII esteso" usa 8 bit). Supporta soltamente i caratteri che hanno un valore tra 0 e 127.
UTF7	Codifica i caratteri mediante una sequenza di uno o più byte, per ognuno dei quali vengono usati 7 bit.
UTF8	Codifica i caratteri mediante una sequenza di uno o più byte. Per i caratteri inclusi nell'insieme ASCII utilizza un solo byte.
Unicode[2] (LittleEndianUnicode)	Codifica i caratteri mediante due byte. I caratteri dell'insieme ASCII hanno il primo byte sempre a zero. Tale codifica viene denominata anche UTF16.
Unicode (BigEndianUnicode)	E' analoga alla precedente: codifica i caratteri mediante due byte. Rispetto alla LittleEndian, i byte sono memorizzati in ordine inverso. (Dunque, i caratteri dell'insieme ASCII hanno il secondo byte sempre a zero.

Come impostazione predefinita, il linguaggio C# memorizza i valori di tipo char e string usando la codifica Unicode (*LittleEndian* o *BigEndian* in base al sistema operativo installato.)

4.1 Convertire i caratteri in byte: metodo GetBytes()

.NET mette a disposizione degli *encoder* per convertire i caratteri in byte e viceversa. Ogni tipo di codifica (ASCII, UTF8, etc) ha il proprio *encoder,* accessibile mediante la proprietà omonima del tipo **Encoding**.

Il codice che segue converte in byte la stringa "ABC", utilizzando la codifica UTF8.

```
string s = "ABC";
byte[] bytes = Encoding.UTF8.GetBytes(s);          // converte "ABC" in byte
Console.Write("{0} -> ", s);
for (int i = 0; i < bytes.Length; i++)
{
```

2 In realtà 16 bit sono sufficienti per i caratteri più comuni, inclusi nella *Basic Multilingual Plane*. Altrimenti, servono 32 bit.

```
    Console.Write("{0:X2} ", bytes[i]);
}
```

Il risultato è un vettore di 3 byte:

Output

```
ABC -> 41 42 43
```

4.2 Convertire da byte a caratteri: metodo GetString()

Ogni *encoder* mette a disposizione il metodo **GetString()**, il quale riceve una sequenza di byte e ritorna l'elenco dei caratteri corrispondenti.

Invocando **GetString()** su un vettore prodotto da **GetBytes()** si ottiene la sequenza di caratteri originale, *ma soltanto se viene usato lo stesso encoder in entrambe le operazioni.*

Il seguente codice converte il vettore di byte {0x41, 0x42, 0x43} in una stringa usando l'*encoder* UTF8:

```
byte[] bytes = { 0x41, 0x42, 0x43 };
string utf8 = Encoding.UTF8.GetString(bytes);
Console.WriteLine(utf8); // -> "ABC"
```