
# ビットとビット文字列の操作

Version 2024.1
2024-06-03

InterSystems Corporation　1 Memorial Drive　Cambridge MA 02142　www.intersystems.com

ビットとビット文字列の操作
InterSystems IRIS Data Platform　Version 2024.1　2024-06-03


インターシステムズ社の製品に関するサポートやご質問は、以下にお問い合わせください：

InterSystems Worldwide Response Center (WRC)
Tel:　+1-617-621-0700
Tel:　+44 (0) 844 854 2917
Email:　support@InterSystems.com

# 目次



テーブル一覧

# ビットとビット文字列の操作

InterSystems IRIS データ・プラットフォーム上に構築したアプリケーションに、関連するブーリアン値のシーケンスを格納することが必要な場合があります。このような場合、多数のブーリアン変数を作成するか、そのブーリアン値を 1 つの配列またはリストに格納することができます。または、"ビット文字列" と呼ばれる概念を使用することもできます。ビット文字列はビットのシーケンスとして定義でき、最下位ビットが最初に表示されます。ビット文字列を使用すると、ストレージ・スペースと処理速度の両面できわめて効率的な方法で、このようなデータを格納することができます。

ビット文字列を格納する方法は 2 つあり、圧縮された文字列または整数として格納できます。"ビット文字列" という用語がコンテキストなしで使われている場合は、ビットのシーケンスが圧縮された文字列として格納されていることを意味します。このページでは、両方の種類のビット文字列を紹介した後、ビット文字列の操作に使用できる手法をいくつか説明します。

## 1 ビット文字列としてのビットのシーケンスの格納

ビットのシーケンスの格納方法として最も一般的なのはビット文字列として格納する方法です。ビット文字列は、特殊な種類の圧縮された文字列です。ストレージ・スペースを節約できることに加え、ビット文字列は、ObjectScript システム関数を使用して効率的に操作できます。

このようなシステム関数の 1 つが $factor で、これは整数をビット文字列に変換します。次の文を実行すると、整数 11744 をビット文字列に変換できます。

```
set bitstring = $factor(11744)
```

ビット文字列の内容の表現を確認するには、zwrite コマンドを使用します。

```
zwrite bitstring
bitstring=$zwc(128,4)_$c(224,45,0,0)/*$bit(6..9,11,12,14)*/
```

最初は暗号のように見えますが、出力の最後になると、実際に設定されているビットのリスト 6、7、8、9、11、12、14 がコメントに表示されていることがわかります。ビット文字列のビット 1 は $2^0$ を表し、ビット 2 は $2^1$ を表すというようになっています。すべてのビットを加算すると、$2^5 + 2^6 + 2^7 + 2^8 + 2^{10} + 2^{11} + 2^{13} = 11744$ となります。

もっと見やすい表現にするには、別のシステム関数 $bit を使用できます。

```
for i=1:1:14 {write $bit(bitstring, i)}
00000111101101
```

この例では、`$bit(bitstring, i)` は、ビット文字列のビット `i` の値を返します。

注釈　ビットのシーケンスが内部的にどのように格納されるかの詳細は、zwrite コマンドの出力を細かく見ると確認できます。

```
bitstring=$zwc(128,4)_$c(224,45,0,0)/*$bit(6..9,11,12,14)*/
```

このビット文字列は、それぞれが 8 ビットの 4 つのチャンクに格納されます。各チャンクを個別に確認すると、224、45、0、0 であることがわかります。

ビット文字列を文字列と考えたほうがわかりやすい場合は、各チャンクを 8 ビットの文字と考えることができます。

一般的にビット文字列は、ビットマップ・インデックスのストレージで適用されます。ビットマップ・インデックスは特殊なタイプのインデックスで、特定のプロパティの指定した値に対応する一連のオブジェクトを表す一連のビット文字列を使用します。ビットマップ内の各ビットは、クラス内のオブジェクトを表します。

例えば、動物とその特徴に関するデータベースを作成する場合、次のようにクラスを定義できます。

```
Class User.Animal Extends %Persistent
{
Property Name As %String [Required];
Property Classification As %String;
Property Diet As %String;
Property Swims As %Boolean;

Index ClassificationIDX On Classification [Type = bitmap];
Index DietIDX On Diet [Type = bitmap];
Index SwimsIDX On Swims [Type = bitmap];
}
```

データベースに動物をいくつか入力すると、データベースは次のようになります。

テーブル 1: Animal データベースの例

| ID | Name (名前) | Classification (分類) | Diet (食性) | Swims (泳ぐかどうか) |
|---|---|---|---|---|
| 1 | Penguin (ペンギン) | Bird (鳥類) | Carnivore (肉食) | 1 |
| 2 | Giraffe (キリン) | Mammal (哺乳類) | Herbivore (草食) | 0 |
| 3 | Cheetah (チーター) | Mammal (哺乳類) | Carnivore (肉食) | 0 |
| 4 | Bullfrog (ウシガエル) | Amphibian (両生類) | Carnivore (肉食) | 1 |
| 5 | Shark (サメ) | Fish (魚類) | Carnivore (肉食) | 1 |
| 6 | Fruit Bat (オオコウモリ) | Mammal (哺乳類) | Herbivore (草食) | 0 |
| 7 | Snapping Turtle (カミツキガメ) | Reptile (爬虫類) | Carnivore (肉食) | 1 |
| 8 | Manatee (マナティ) | Mammal (哺乳類) | Herbivore (草食) | 1 |
| 9 | Ant (アリ) | Insect (昆虫類) | Omnivore (雑食) | 0 |
| 10 | Rattlesnake (ガラガラヘビ) | Reptile (爬虫類) | Carnivore (肉食) | 0 |

ビットマップ・インデックス `DietIDX` は、Diet プロパティが特定の値である動物を追跡します。インデックスの各行に、最大 64,000 の動物を表すチャンクを格納できます。

テーブル 2: Diet プロパティのビットマップ・インデックス

| Diet | チャンク | ビットマップ |
|---|---|---|
| Carnivore | 1 | 1011101001 |
| Herbivore | 1 | 0100010100 |
| Omnivore | 1 | 0000000010 |

内部的に、このビットマップ・インデックスはグローバル `^User.AnimalI` の次のノードに格納されます。

```
^User.AnimalI("DietIDX"," CARNIVORE",1)
^User.AnimalI("DietIDX"," HERBIVORE",1)
^User.AnimalI("DietIDX"," OMNIVORE",1)
```

最初の添え字はインデックスの名前 (DietIDX) で、2 つ目の添え字はインデックスが付けられたプロパティの値 (例えば、CARNIVORE) で、3 つ目の添え字はチャンク番号 (この例では 1) です。

同様に、ビットマップ・インデックス SwimsIDX は、Swims プロパティが特定の値である動物を追跡します。

テーブル 3: Swims プロパティのビットマップ・インデックス

| Swims | チャンク | ビットマップ |
|---|---|---|
| True | 1 | 1001101100 |
| False | 1 | 0110010011 |

このビットマップ・インデックスは ^User.AnimalI の次のノードに格納されます。

```
^User.AnimalI("SwimsIDX",1,1)
^User.AnimalI("SwimsIDX",0,1)
```

ビット文字列の機能がどのようなものかを示すために、データベース内の肉食動物の数を数えてみます。これは、実際のデータを確認する必要なく、ビットマップ内の 1 の数を数えるだけで簡単にわかります。システム関数 $bitcount を使用するだけです。

```
set c = ^User.AnimalI("DietIDX"," CARNIVORE",1)

write $bitcount(c,1)
6
```

同様に、泳げる動物の数を数えることができます。

```
set s = ^User.AnimalI("SwimsIDX",1,1)

write $bitcount(s,1)
5
```

泳げる肉食動物の数を数えるには、$bitlogic 関数を使用して、2 つのセットの共通集合を見つけます。

```
set cs = $bitlogic(c&s)

write $bitcount(cs,1)
4
```

注釈　もう一度 zwrite を使用して、肉食動物のビットマップが内部的にどのように格納されるのかを確認してみます。

```
zwrite ^User.AnimalI("DietIDX"," CARNIVORE",1)
^User.AnimalI("DietIDX"," CARNIVORE",1)=$zwc(413,2,0,2,6,8,9)/*$bit(2,4..6,8,11)*/
```

ここでは、Diet プロパティが CARNIVORE である動物すべてに対応するビットを確認できます。ご覧のように、ビットマップ・インデックスが 64,000 ビットのチャンクに分割されています。指定の ID を持つ動物について格納されているビットは、チャンク (ID¥64000) + 1、位置 (ID#64000) + 1 に格納されます。したがって、ID が 1 の動物を表すビットは、チャンク 1、位置 2 に格納されます。このため、このビット文字列では、ビット 2 は、Giraffe ではなく Penguin を表します。

SQL エンジンにはビットマップ・インデックスを活用できる特別な最適化が多数含まれているため、SQL クエリを記述する際に常にそのメリットを得ることができます。

ビット文字列を使用する方法のその他の例は、"ビット文字列の操作" を参照してください。

# 2 整数としてのビットのシーケンスの格納

ブーリアン引数のシーケンスをメソッドに渡す場合、一般的な方法の 1 つは、単一の整数にエンコードされたビットのシーケンスとして渡す方法です。

例えば、メソッド Security.System.ExportAll() は、InterSystems IRIS インスタンスからのセキュリティ設定のエクスポートに使用されます。このメソッドのクラス・リファレンスを見てみると、このメソッドは次のように定義されていることがわかります。

```
classmethod ExportAll(FileName As %String = "SecurityExport.xml",
ByRef NumExported As %String, Flags As %Integer = -1) as %Status
```

3 つ目の引数の `Flags` は整数で、その各ビットはエクスポートできるセキュリティ・レコードの種類を表します。

```
Flags - What type of records to export to the file, -1 = ALL
Bit 0 - System
Bit 1 - Events
Bit 2 - Services
Bit 4 - Resources
Bit 5 - Roles
Bit 6 - Users
Bit 7 - Applications
Bit 8 - SSL Configs
Bit 9 - PhoneProvider
Bit 10 - X509Credential
Bit 11 - OpenAMIdentityService
Bit 12 - SQL privileges
Bit 13 - X509Users
Bit 14 - DocDBs
Bit 15 - LDAPConfig
Bit 16 - KMIPServer
```

整数として格納されるビット文字列のビット 0 は $2^0$ を表し、ビット 1 は $2^1$ を表すというようになっています。ビット 5、6、7、8、10、11、13 に対応する種類のセキュリティ・レコードをエクスポートするには、Flags を $2^5 + 2^6 + 2^7 + 2^8 + 2^{10} + 2^{11} + 2^{13} = 11744$ に設定します。

ObjectScript では、以下のようになります。

```
set flags = (2**5) + (2**6) + (2**7) + (2**8) + (2**10) + (2**11) + (2**13)
set status = ##class(Security.System).ExportAll("SecurityExport.xml", .numExported, flags)
```

一部の InterSystems API には、コードを読みやすくするためにマクロが定義されています。その 1 つが DataMove ユーティリティです。このユーティリティでは、メソッド DataMove.Data.CreateFromMapEdits() を使用して DataMove オブジェクトを作成します。ここでは詳しくは説明しませんが、このメソッドはクラス・リファレンスでは次のように定義されています。

```
classmethod CreateFromMapEdits(Name As %String, ByRef Properties As %String,
ByRef Warnings As %String, ByRef Errors As %String) as %Status
```

引数は以下のとおりです。

```
Parameters:
Name - A name for the DataMove object to create.
Properties - Array of properties used to create the object. The following properties may
optionally be specified. See the class definition for more information on each property.
Properties("Description") - Description of the Data Move operation, default ="".
Properties("Flags") - Flags describing the operation, default = 0.
Properties("LogFile") - Directory and filename of the logfile, default = \iris\mgr\DataMovename.log.
```

`Properties("Flags")` を定義しやすくするために、以下のマクロが使用可能になっています。

```
Bit Flags to control Data Move.
$$$BitNoSrcJournal - Allow source database to be non-journaled
$$$BitNoWorkerJobs - Don't use 'worker' jobs during the copying of data
$$$BitBatchMode - Run Copy jobs in 'batch' mode
$$$BitCheckActivate - Call $$CheckActivate^ZDATAMOVE() during Activate()
```

これらのマクロは、特定のビットの計算値として定義されているため、どのビットがどのフラグを表すかを覚えておかなくても、適切なビットを設定できます。

```
#;Definitions for DataMove Flags properties
#define BitNoSrcJournal 1
#define BitNoWorkerJobs 512
#define BitBatchMode 2048
#define BitCheckActivate 4096
```

コードで次のコード・スニペットを使用して、フラグの設定と DataMove オブジェクトの作成を行うことができます。

```
// Set properties("Flags") to 6657
set properties("Flags") = $$$BitNoSrcJournal + $$$BitNoWorkerJobs + $$$BitBatchMode + $$$BitCheckActivate
set status = ##class(DataMove.Data).CreateFromMapEdits("dm", .properties, .warnings, .errors)
```

ビット文字列を整数として使用する方法のその他の例は、"整数として実装されるビット文字列の操作"を参照してください。

# 3 ビット文字列の操作

## 3.1 ビットの設定

ビット文字列を新規作成するには、$bit 関数を使用して、目的のビットを 1 に設定します。

```
kill bitstring

set $bit(bitstring, 3) = 1

set $bit(bitstring, 6) = 1

set $bit(bitstring, 11) = 1
```

$bit を使用して、既存のビット文字列内の特定のビットを 1 に設定する場合 :

```
set $bit(bitstring, 5) = 1
```

$bit を使用して、既存のビット文字列内の特定のビットを 0 に設定する場合 :

```
set $bit(bitstring, 5) = 0
```

ビット文字列の最初のビットはビット 1 であるため、ビット 0 を設定しようとすると、エラーが返されます。

```
set $bit(bitstring, 0) = 1

SET $BIT(bitstring, 0) = 1
^
<VALUE OUT OF RANGE>
```

## 3.2 ビットが設定されているかどうかのテスト

既存のビット文字列内の特定のビットが設定されているかどうかをテストする場合も、$bit 関数を使用します。

```
write $bit(bitstring, 6)
1
write $bit(bitstring, 5)
0
```

明示的に設定されていないビットをテストした場合、$bit は 0 を返します。

```
write $bit(bitstring, 4)
0
write $bit(bitstring, 55)
0
```

$bit 関数の詳細は、"$BIT" を参照してください。

## 3.3 ビットの表示

ビット文字列内のビットを表示するには、$bitcount 関数を使用して、ビット文字列内のビットの数を取得してから、ビットをループします。

```
for i=1:1:$bitcount(bitstring) {write $bit(bitstring, i)}
00100100001
```

$bitcount は、ビット文字列内の 1 または 0 の数の取得にも使用できます。

```
write $bitcount(bitstring, 1)
3
write $bitcount(bitstring, 0)
8
```

$bitcount 関数の詳細は、"$BITCOUNT" を参照してください。

## 3.4 設定ビットの検索

ビット文字列内で設定されているビットを検索するには、$bitfind 関数を使用します。この関数は、ビット文字列内の指定の位置から始めて、指定した値の次のビットの位置を返します。

```
Class User.BitStr
{

ClassMethod FindSetBits(bitstring As %String)
{
   set bit = 0
   for {
      set bit = $bitfind(bitstring, 1, bit)
      quit:'bit
      write bit, " "
      set bit = bit + 1
   }
}

}
```

このメソッドは、文字列内を検索して、$bitfind が 0 を返したら終了します。0 はこれ以上一致が見つからなかったことを示します。

```
do ##class(User.BitStr).FindSetBits(bitstring)
3 6 11
```

ビット文字列が等価かどうかをテストする場合は十分に注意してください。

例えば、ビット・セットが同じ 2 つのビット文字列 `b1` と `b2` があるとします。

```
do ##class(User.BitStr).FindSetBits(b1)
3 6 11
do ##class(User.BitStr).FindSetBits(b2)
3 6 11
```

しかし、これらを比較すると、実際には等価でないことがわかります。

```
write b1 = b2
0
```

zwrite を使用すると、これら 2 つのビット文字列の内部表現が異なることがわかります。

```
zwrite b1
b1=$zwc(405,2,2,5,10)/*$bit(3,6,11)*/

zwrite b2
b2=$zwc(404,2,2,5,10)/*$bit(3,6,11)*/
```

この場合、b2 のビット 12 が 0 に設定されています。

```
for i=1:1:$bitcount(b1) {write $bit(b1, i)}
00100100001
USER>for i=1:1:$bitcount(b2) {write $bit(b2, i)}
001001000010
```

さらに、他の内部表現が存在する場合もあります。例えば、整数をビット文字列に変換する $factor で作成される内部表現です。

```
set b3 = $factor(1060)

zwrite b3
b3=$zwc(128,4)_$c(36,4,0,0)/*$bit(3,6,11)*/

for i=1:1:$bitcount(b3) {write $bit(b3, i)}
00100100001000000000000000000000
```

内部表現が異なる可能性がある 2 つのビット文字列を比較する場合、$bitlogic 関数を使用して、設定ビットを直接比較することができます。これについては、"ビット単位算術演算の実行" で説明しています。

$bitfind 関数の詳細は、"$BITFIND" を参照してください。

## 3.5 ビット単位算術演算の実行

ビット文字列にビット単位論理演算を実行するには、$bitlogic を使用します。

次の例では、a と b の 2 つのビット文字列を使用していることを想定しています。

```
for i=1:1:$bitcount(a) {write $bit(a, i)}
100110111
for i=1:1:$bitcount(b) {write $bit(b, i)}
001000101
```

$bitlogic 関数を使用して、これらのビットに対して論理 OR を実行します。

```
set c = $bitlogic(a|b)

for i=1:1:$bitcount(c) {write $bit(c, i)}
101110111
```

$bitlogic 関数を使用して、これらのビットに対して論理 AND を実行します。

```
set d = $bitlogic(a&b)

for i=1:1:$bitcount(d) {write $bit(d, i)}
000000101
```

次の例は、$bitlogic 関数を使用して論理 XOR を実行し、b1 と b3 の 2 つのビット文字列の設定ビットが内部表現にかかわらず同じかどうかをテストする方法を示しています。

```
zwrite b1
b1=$zwc(405,2,2,5,10)/*$bit(3,6,11)*/

zwrite b3
b3=$zwc(128,4)_$c(36,4,0,0)/*$bit(3,6,11)*/

write $bitcount($bitlogic(b1^b3),1)
0
```

論理 XOR により、これら 2 つのビット文字列内の設定ビットに違いがないことをすばやく判断できます。

ビット文字列に対するビット単位論理演算の詳細は、"BITLOGIC" を参照してください。

## 3.6 整数としてのビット文字列への変換

標準のビット文字列を、整数として格納されるビット文字列に変換するには、$bitfind 関数を使用して設定ビットを検索し、それらの 2 の累乗を合計します。結果を 2 で割って、ビットを左にシフトさせることを忘れないでください。標準のビット文字列のビット 1 は、整数としてのビット文字列のビット 0 に対応するからです。

```
ClassMethod BitstringToInt(bitstring As %String)
{
   set bitint = 0
   set bit = 0
   for {
      set bit = $bitfind(bitstring, 1, bit)
      quit:'bit
      set bitint = bitint + (2**bit)
      set bit = bit + 1
   }
   return bitint/2
}
```

`bitstring` を整数としてのビット文字列に変換します。

```
for i=1:1:$bitcount(bitstring) {write $bit(bitstring, i)}
00100100001
set bitint = ##class(User.BitStr).BitstringToInt(bitstring)

write bitint
1060
```

# 4 整数として実装されるビット文字列の操作

## 4.1 ビットの設定

整数として格納されるビット文字列を新規作成するには、各ビットの 2 の累乗を合計します。

```
set bitint = (2**2) + (2**5) + (2**10)

write bitint
1060
```

既存の、整数としてのビット文字列内にある特定のビットを 1 に設定するには、$zboolean 関数のオプション 7 (arg1 ! arg2) (論理 OR) を使用します。

```
set bitint = $zboolean(bitint, 2**4, 7)

write bitint
1076
```

既存の、整数としてのビット文字列内にある特定のビットを 0 に設定するには、$zboolean 関数のオプション 2 (arg1 & ~arg2) を使用します。

```
set bitint = $zboolean(bitint, 2**4, 2)

write bitint
1060
```

既存の、整数としてのビット文字列内にある特定のビットを切り替えるには、$zboolean 関数のオプション 6 (arg1 ^ arg2) (論理 XOR) を使用します。

```
set bitint = $zboolean(bitint, 2**4, 6)

write bitint
1076
set bitint = $zboolean(bitint, 2**4, 6)

write bitint
1060
```

## 4.2 ビットが設定されているかどうかのテスト

既存の、整数としてのビット文字列内にある特定のビットが設定されているかどうかをテストするには、$zboolean 関数のオプション 1 (arg1 & arg2) (論理 AND) を使用します。

```
write $zboolean(bitint, 2**5, 1)
32
write $zboolean(bitint, 2**4, 1)
0
```

ビット 5 は設定されているので、$zboolean はそのビットの値を返します。

ビット 4 は設定されていないので、$zboolean は 0 を返します。

## 4.3 ビットの表示

整数としてのビット文字列内のビットを表示するには、以下のようなメソッドを使用します。このメソッドは、ビットをループして $zboolean 関数を使用します。

```
Class User.BitInt{
{
ClassMethod LogicalToDisplay(bitint as %Integer)
{
   for i = 0:1 {
      quit:((2**i) > bitint)
      if $zboolean(bitint, 2**i, 1) {write 1} else {write 0}
   }
}
}
```

このメソッドを `bitint` に対して実行すると、結果は次のようになります。

```
do ##class(User.BitInt).LogicalToDisplay(bitint)
00100100001
```

## 4.4 設定ビットの検索

次のメソッドは、$zlog 関数を使用して、整数としてのビット文字列内で設定されているビットを検索します。この関数は、常用対数値を返します。このメソッドは、ビット文字列内のチャンクを大きいものから順に、残っているチャンクがなくなるまで取り出していきます。

```
ClassMethod FindSetBits(bitint as %Integer)
{
   set bits = ""
   while (bitint '= 0) {
      set bit = $zlog(bitint) \ $zlog(2)
      set bits = bit _ " " _ bits
      set bitint = bitint - (2**bit)
   }
   write bits
}
```

このメソッドを bitint に対して実行すると、結果は次のようになります。

```
do ##class(User.BitInt).FindSetBits(bitint)
2 5 10
```

## 4.5 ビット単位算術演算の実行

整数として格納されているビット文字列にビット単位論理演算を実行するには、$zboolean を使用します。

次の例では、整数として格納されている 2 つのビット文字列 a および b があり、"ビットの表示" で説明した LogicalToDisplay() メソッドを使用してこれらのビットを表示することを想定しています。

```
do ##class(User.BitInt).LogicalToDisplay(a)
100110111
do ##class(User.BitInt).LogicalToDisplay(b)
001000101
```

$zboolean 関数のオプション 7 を使用して、これらのビットに対して論理 OR を実行します。

```
set c = $zboolean(a, b, 7)

do ##class(User.BitInt).LogicalToDisplay(c)
101110111
```

$zboolean 関数のオプション 1 を使用して、これらのビットに対して論理 AND を実行します。

```
set d = $zboolean(a, b, 1)

do ##class(User.BitInt).LogicalToDisplay(d)
000000101
```

ビット単位論理演算の詳細は、"$ZBOOLEAN" を参照してください。

## 4.6 標準のビット文字列への変換

整数として格納されているビット文字列を標準のビット文字列に変換するには、$factor 関数を使用します。この例では、整数としてのビット文字列 bitint があり、"設定ビットの検索" で説明した FindSetBits() メソッドを使用して、設定されているビットを表示することを想定しています。

```
do ##class(User.BitInt).FindSetBits(bitint)
2 5 10
set bitstring = $factor(bitint)

zwrite bitstring
bitstring=$zwc(128,4)_$c(36,4,0,0)/*$bit(3,6,11)*/
```

標準のビット文字列にはビット 0 がないため、ビットは右に 1 つシフトして表示されることに注意してください。ビット文字列内の最初のビットはビット 1 です。