Native SDK for Node.js の使用法

Version 2023.1
2024-01-02

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

Native SDK for Node.js の使用法
InterSystems IRIS Data Platform　Version 2023.1　2024-01-02

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

1

Native SDK for Node.js の概要

このドキュメントで取り上げる内容の詳細なリストは、"目次" を参照してください。Native SDK クラスおよびメソッドの簡単な説明は、"Native SDK for Node.js のクイック・リファレンス" を参照してください。

InterSystems Native SDK for Node.js は、以前は ObjectScript を介してのみ使用可能であった、強力な InterSystems IRIS® リソースへの軽量インタフェースです。

- ObjectScript メソッドおよび関数の呼び出し ー JavaScript のネイティブ・メソッドを呼び出すのと同じくらい簡単に、JavaScript アプリケーションから任意の埋め込み言語のクラスメソッドを呼び出します。
- グローバル配列の操作 ー グローバル (InterSystems 多次元ストレージ・モデルの実装に使用されるツリーベースのスパース配列) に直接アクセスします。
- インターシステムズのトランザクションとロックの使用 ー 埋め込み言語のトランザクションおよびロック・メソッドの Native SDK 実装を使用して、インターシステムズのデータベースを操作します。

重要　Native SDK for Node.js を使用するには、"接続ツール" で説明されている Node.js 接続パッケージをダウンロードする必要があります。

注釈　**Node.js リレーショナル・アクセスのサポート**

InterSystems では、Node.js リレーショナル・アクセスもサポートしています。node-odbc オープン・ソース Node.js モジュールは、Windows および UNIX® ベースのシステムでサポートされています。詳細は、"InterSystems ODBC ドライバの使用法" の "Node.js リレーショナル・アクセスのサポート" を参照してください。

その他の言語用の Native SDK

Native SDK のバージョンは Java、.NET、および Python でも使用できます。

- Native SDK for Java の使用法
- Native SDK for .NET の使用法
- Native SDK for Python の使用法

グローバルに関する詳細情報

グローバルを自由自在に使いこなしたい開発者には、以下のドキュメントを強くお勧めします。

- グローバルの使用法 ー ObjectScript でグローバルを使用する方法、およびサーバに多次元ストレージを実装する方法の詳細を示します。

2

ObjectScript メソッドおよび関数の呼び出し

この章では、ObjectScript クラス・メソッドおよび関数を JavaScript アプリケーションから直接呼び出すことを可能にするクラス iris のメソッドを説明します。詳細および例は、以下のセクションを参照してください。

- クラス・メソッドの呼び出し ー ObjectScript クラス・メソッドを呼び出す方法を示します。
- 関数の呼び出し ー ObjectScript 関数およびプロシージャを呼び出す方法を示します。

2.1 クラス・メソッドの呼び出し

以下の Native SDK メソッドは、指定された ObjectScript クラス・メソッドを呼び出します。これらは、className および methodName の文字列引数に加え、0 個以上のメソッド引数を含む配列を取ります。

- Iris.ClassMethodValue() ー クラス・メソッドを呼び出して、返り値を取得します。
- Iris.ClassMethodVoid() ー クラス・メソッドを呼び出して、返り値を破棄します。

引数リストでは、末尾の引数を省略できます。すべての引数の数よりも少ない数の引数が渡されるか、末尾の引数に対して null が渡されることでそれらの引数には既定の値が使用されるようになります。非 null 引数が null 引数の右側に渡されると、例外がスローされます。

この例のコードでは、User.NativeTest という ObjectScript テスト・クラスからいくつかのデータ型のクラス・メソッドを呼び出します (この例の直後に表示されています)。

JavaScript からの ObjectScript クラス・メソッドの呼び出し

この例では、変数 irisjs はクラス Iris の以前に定義されたインスタンスで、現在サーバに接続されていると想定します。

```
const className = 'User.NativeTest';
let cmValue = "";
let comment = "";

comment = ".cmBoolean() tests whether arguments 2 and 3 are equal: "
cmValue = irisjs.classMethodValue(className,'cmBoolean',2,3);
console.log(className + comment + cmValue);

comment = ".cmBytes() returns integer arguments 72,105,33 as a byte array (string value 'Hi!'):
"
cmValue = irisjs.classMethodValue(className,'cmBytes',72,105,33); //ASCII 'Hi!'
console.log(className + comment + cmValue);

comment = ".cmString() concatenates 'Hello' with argument string 'World': "
cmValue = irisjs.classMethodValue(className,'cmString','World');
console.log(className + comment + cmValue);
```

```
comment = ".cmLong() returns the sum of arguments 7+8: "
cmValue = irisjs.classMethodValue(className,'cmLong',7,8);
console.log(className + comment + cmValue);

comment = ".cmDouble() multiplies argument 4.5 by 1.5: "
cmValue = irisjs.classMethodValue(className,'cmDouble',4.5);
console.log(className + comment + cmValue);

comment = ".cmVoid() assigns argument value 75 to global node ^cmGlobal: "
try {
  irisjs.kill('cmGlobal') // delete ^cmGlobal if it exists
  irisjs.classMethodVoid(className,'cmVoid',75);
  cmValue = irisjs.get("cmGlobal");  //get current value of ^cmGlobal
}
catch {cmValue = 'method failed.'}
console.log(className + comment + cmValue);
```

ObjectScript クラス User.NativeTest

Class Definition

```
Class User.NativeTest Extends %Persistent
  {

  ClassMethod cmBoolean(cm1 As %Integer, cm2 As %Integer) As %Boolean
  {
     Quit (cm1=cm2)
  }

  ClassMethod cmBytes(cm1 As %Integer, cm2 As %Integer, cm3 As %Integer) As %Binary
  {
     Quit $CHAR(cm1,cm2,cm3)
  }

  ClassMethod cmString(cm1 As %String) As %String
  {
     Quit "Hello "_cm1
  }

  ClassMethod cmLong(cm1 As %Integer, cm2 As %Integer) As %Integer
  {
     Quit cm1+cm2
  }

  ClassMethod cmDouble(cm1 As %Double) As %Double
  {
     Quit cm1 * 1.5
  }

  ClassMethod cmVoid(cm1 As %Integer)
  {
     Set ^cmGlobal=cm1
     Quit
  }
}
```

ターミナルからこれらのメソッドを呼び出すことで、それらをテストできます。例を以下に示します。

```
USER>write ##class(User.NativeTest).cmString("World")
Hello World
```

2.2 関数の呼び出し

以下の Native SDK メソッドは、ユーザ定義の ObjectScript 関数またはプロシージャを呼び出します (″ObjectScript の使用法″ の “呼び出し可能なユーザ定義コード・モジュール”″ を参照)。これらは、className および methodName の文字列引数に加え、0 個以上のメソッド引数を含む配列を取ります。

- Iris.function() ー ユーザ定義関数を呼び出して、返り値を取得します。
- Iris.procedure() ー ユーザ定義プロシージャを呼び出して、返り値を破棄します。

引数リストでは、末尾の引数を省略できます。すべての引数の数よりも少ない数の引数が渡されるか、末尾の引数に対して null が渡されることでそれらの引数には既定の値が使用されるようになります。非 null 引数が null 引数の右側に渡されると、例外がスローされます。

この例のコードでは、NativeRoutine という ObjectScript テスト・ルーチンからいくつかのデータ型の関数を呼び出します (この例の直後に表示されているファイル NativeRoutine.mac)。

JavaScript からの ObjectScript ルーチンの呼び出し

この例のコードでは、ObjectScript ルーチン NativeRoutine からサポートされている各データ型の関数を呼び出します。irispy はクラス Iris の既存のインスタンスで、現在サーバに接続されていると想定します。

```
const routineName = 'NativeRoutine';
let fnValue = "";
let comment = "";

comment = ".fnBoolean() tests whether arguments 2 and 3 are equal: "
fnValue = irisjs.function(routineName,'fnBoolean',2,3);
console.log(routineName + comment + fnValue);

comment = ".fnBytes() returns integer arguments 72,105,33 as a byte array (string value 'Hi!'):
"
fnValue = irisjs.function(routineName,'fnBytes',72,105,33);  // ASCII 'Hi!'
console.log(routineName + comment + fnValue);

comment = ".fnString() concatenates 'Hello' with argument string 'World': "
fnValue = irisjs.function(routineName,"fnString","World");
console.log(routineName + comment + fnValue);

comment = ".fnLong() returns the sum of arguments 7+8: "
fnValue = irisjs.function(routineName,'fnLong',7,8);
console.log(routineName + comment + fnValue);

comment = ".fnDouble() multiplies argument 4.5 by 1.5: "
fnValue = irisjs.function(routineName,'fnDouble',4.5);
console.log(routineName + comment + fnValue);

comment = ".fnProcedure() assigns argument value 66 to global node ^fnGlobal: "
try {
  irisjs.kill('fnGlobal') // delete ^fnGlobal if it exists
  irisjs.procedure(routineName,"fnProcedure",66);
  fnValue = irisjs.get("fnGlobal"); //  get current value of node ^fnGlobal
}
catch {fnValue = 'procedure failed.'}
console.log(routineName + comment + fnValue);
```

ObjectScript ルーチン NativeRoutine.mac

前の例を実行するには、この ObjectScript ルーチンがコンパイルされ、サーバで使用可能である必要があります。

```
fnBoolean(fn1,fn2) public {
   quit (fn1=fn2)
}
fnBytes(fn1,fn2,fn3) public {
    quit $CHAR(fn1,fn2,fn3)
}
fnString(fn1) public {
    quit "Hello "_fn1
}
fnLong(fn1,fn2) public {
    quit fn1+fn2
}
fnDouble(fn1) public {
    quit fn1 * 1.5
}
fnProcedure(fn1) public {
    set ^fnGlobal=fn1
    quit
}
```

ターミナルからこれらの関数を呼び出すことで、それらをテストできます。例を以下に示します。

```
USER>write $$fnString^NativeRoutine("World")
Hello World
```

3
グローバル配列の操作

Node.js Native SDK は、InterSystems IRIS のインスタンスからグローバル配列を操作するメカニズムを提供します。この章では、以下の項目について説明します。

- グローバル配列の概要 – グローバル配列の概念を示し、Native SDK がどのように使用されるかを簡単に説明します。
- ノードの作成、更新、および削除 – グローバル配列を作成し、ノード値を設定または変更する方法について説明します。
- グローバル配列のノードの検索 – グローバル配列のノードへの高速アクセスを可能にする反復メソッドについて説明します。

注釈 **Node.js でのデータベース接続の作成**

この章の例では、irisjs という名前の Iris オブジェクトが既に存在し、サーバに接続されていると想定します。これは、以下のコードを使用して作成し、接続されています。

```
const IRISNative = require('intersystems-iris-native')
let connectionInfo = {host:'127.0.0.1', port:51773, ns:'USER', user:'_SYSTEM', pwd:'SYS'};
const conn = IRISNative.createConnection(connectionInfo);
const irisjs = conn.createIris();
```

詳細は、createConnection() および createIris() のクイック・リファレンスのエントリを参照してください。

3.1 グローバル配列の概要

グローバル配列は、すべてのスパース配列のように、ツリー構造です (連続的に番号付けされたリストではありません)。グローバル配列の背後にある基本概念は、ファイル構造に例えて示すことができます。ツリーの各ディレクトリは、ルート・ディレクトリ識別子とそれに続く一連のサブディレクトリ識別子で構成されるパスによって一意に識別され、ディレクトリにはデータが含まれることも、含まれないこともあります。

グローバル配列の仕組みも同じです。ツリーの各ノードは、グローバル名識別子と一連の添え字識別子で構成されるノード・アドレスによって一意に識別され、ノードには値が含まれることも、含まれないこともあります。例えば、以下は 5 つのノードで構成されるグローバル配列で、そのうち 2 つのノードには値が含まれます。

```
root -->|--> foo --> SubFoo='A'
        |--> bar --> lowbar --> UnderBar=123
```

値はその他のノード・アドレス (root または root->bar など) に格納できますが、それらのノード・アドレスが値なしの場合、リソースは無駄になりません。InterSystems ObjectScript グローバルの表記では、値を持つ 2 つのノードは次のようになります。

```
root('foo','SubFoo')
root('bar','lowbar','UnderBar')
```

グローバル名 (root) の後に、括弧で囲まれたコンマ区切り添え字リストが続きます。この両者で、ノードのパス全体を指定します。

このグローバル配列は、Native SDK set() メソッドへの 2 つの呼び出しで作成されます。

```
irisjs.set('A', 'root', 'foo', 'SubFoo');
irisjs.set(123, 'root', 'bar', 'lowbar', 'UnderBar');
```

グローバル配列 root は、最初の呼び出しが値 'A' をノード root('foo','SubFoo') に割り当てるときに作成されます。ノードは任意の順序で、任意の添え字セットを使用して作成できます。これら 2 つの呼び出しの順序を逆にした場合でも、同じグローバル配列が作成されます。値なしノードは自動的に作成され、不要になると自動的に削除されます。詳細は、このドキュメントで後述する "ノードの作成、更新、および削除" を参照してください。

この配列を作成する Native SDK コードを、以下に例示します。Connection オブジェクトをサーバに接続して、クラス Iris のインスタンスを作成します。Iris メソッドを使用してグローバル配列を作成し、生成された永続値をデータベースから読み取った後、グローバル配列を削除します。

NativeDemo プログラム

```
// Create a reference to the Native SDK module
  const IRISNative = require('intersystems-iris-native')

// Open a connection to the server and create an Iris object
  let connectionInfo = {
    host: '127.0.0.1',
    port: 51773,
    ns: 'USER',
    user: '_SYSTEM',
    pwd: 'SYS'
  };
  const conn = IRISNative.createConnection(connectionInfo);
  const irisjs = conn.createIris();

// Create a global array in the USER namespace on the server
  irisjs.set('A', 'root', 'foo', 'SubFoo');
  irisjs.set(123, 'root', 'bar', 'lowbar', 'UnderBar');

// Read the values from the database and print them
  let subfoo = irisjs.get('root', 'foo', 'SubFoo')
  let underbar = irisjs.get('root', 'bar', 'lowbar', 'UnderBar')
  console.log('Created two values: ');
  console.log('   root("foo","SubFoo")=' + subfoo);
  console.log('   root("bar","lowbar","UnderBar")=' + underbar);
```

NativeDemo は、以下の行を出力します。

```
Created two values:
   root("foo","SubFoo")=A
   root("bar","lowbar","UnderBar")=123
```

この例では、IRISNative は Native SDK のインスタンスです。静的メソッド createConnection() は、USER ネームスペースに関連付けられているデータベースへの接続を提供する、conn という名前の Connection オブジェクトを定義します。Native SDK メソッドは以下のアクションを実行します。

- Connection.createIRIS() は、サーバ接続 conn を介してデータベースにアクセスする、irisjs という名前の Iris の新しいインスタンスを作成します。
- Iris.set() は、データベース・ネームスペース USER に新しい永続ノードを作成します。
- Iris.get() は、データベースに対してクエリを実行して、指定されたノードの値を返します。

- Iris.kill() は、指定されたルート・ノードとそのサブノードすべてをデータベースから削除します。

次の章では、これらのすべてのメソッドについて詳細に説明し、例を示します。

3.1.1 Native SDK 用語の用語集

ここに示す概念の概要については、前のセクションを参照してください。この用語集の例は、以下に示すグローバル配列構造を指しています。Legs グローバル配列には、10 個のノードと 3 つのノード・レベルがあります。10 個のノードのうち 7 つには、値が含まれます。

```
Legs                        // root node, valueless, 3 child nodes
  fish = 0                  // level 1 node, value=0
  mammal                    // level 1 node, valueless
    human = 2               // level 2 node, value=2
    dog = 4                 // level 2 node, value=4
  bug                       // level 1 node, valueless, 3 child nodes
    insect = 6              // level 2 node, value=6
    spider = 8              // level 2 node, value=8
    millipede = Diplopoda   // level 2 node, value='Diplopoda', 1 child node
      centipede = 100       // level 3 node, value=100
```

子ノード

指定された親ノードの直下にあるノードです。子ノードのアドレスは、親添え字リストの末尾に 1 つの添え字を追加して指定します。例えば、親ノード Legs('mammal') には子ノード Legs('mammal','human') および Legs('mammal','dog') があります。

グローバル名

ルート・ノードの識別子は、グローバル配列全体の名前でもあります。例えば、ルート・ノード識別子 `Legs` は、グローバル配列 Legs のグローバル名です。

ノード

グローバル配列の要素で、グローバル名と任意の数の添え字識別子で構成されるネームスペースによって一意に識別されます。ノードは、データを含むか、子ノードを持つか、またはこれらの両方を持つ必要があります。

ノード・レベル

ノード・アドレス内の添え字の数。'レベル 2 ノード' は、'2 つの添え字を持つノード' のもう 1 つの表現方法です。例えば、Legs('mammal','dog') は、レベル 2 ノードです。ルート・ノード Legs の 2 レベル下で、Legs('mammal') の 1 レベル下です。

ノード・アドレス

グローバル名とすべての添え字を含む、ノードの完全なネームスペースです。例えば、ノード・アドレス Legs("fish") は、ルート・ノード識別子 `'Legs'` と 1 つの添え字 `'fish'` を含むリストで構成されます。コンテキストに応じて、Legs (添え字リストなし) はルート・ノード・アドレスまたはグローバル配列全体を参照することができます。

ルート・ノード

グローバル配列ツリーの基点にある添え字なしノードです。ルート・ノードの識別子は、その添え字なしのグローバル名です。

サブノード

特定のノードのすべての下位ノードは、そのノードのサブノードと呼ばれます。例えば、ノード Legs('bug') には 2 つのレベルの 4 つの異なるサブノードがあります。9 つの添え字付きノードはすべて、ルート・ノード Legs のサブノードです。

添え字/添え字リスト

ルート・ノードの下にあるノードはすべて、グローバル名および 1 つまたは複数の添え字識別子のリストを指定して処理されます(グローバル名に添え字リストを加えたものが、ノード・アドレスです)。

ターゲット・アドレス

多くの Native SDK メソッドは有効なノード・アドレスを指定する必要がありますが、ノード・アドレスは必ずしも既存のノードを指す必要はありません。例えば、set() メソッドは value 引数とターゲット・アドレスを取り、そのアドレスに値を格納します。ターゲット・アドレスにノードが存在しない場合は、新しいノードが作成されます。

値

ノードには、サポートされている任意の型の値を含めることができます。子ノードを持たないノードには値を含める必要があり、子ノードを持つノードは値なしにすることができます。

値なしノード

ノードは、データを含むか、子ノードを持つか、またはこれらの両方を持つ必要があります。子ノードを持っているがデータを含まないノードは、値なしノードと呼ばれます。値なしノードは、単に下位レベルのノードへのポインタです。

3.1.2 グローバル命名規則

グローバル名と添え字は、次の規則に従います。

- ノード・アドレスの長さ (グローバル名とすべての添え字の長さの合計) は最大 511 文字です(入力した一部の文字は、この制限のために、複数のエンコードされた文字としてカウントされることがあります。詳細は、"グローバルの使用法" の "グローバル参照の最大長" を参照してください)。
- グローバル名には文字、数字、およびピリオド ('.') を使用でき、最大 31 文字の有効文字を使用できます。文字で始まる必要があり、ピリオドで終了することはできません。
- 添え字は文字列、整数、または数値にすることができます。文字列の添え字は大文字と小文字が区別され、すべてのタイプの文字を使用できます。長さの制限は、ノード・アドレス全体の最大長が 511 文字という制限のみです。

3.2 ノードの作成、更新、および削除

ここでは、ノードの作成、更新、および削除に使用する Native SDK について説明します。set()、increment()、および kill() は、グローバル配列を作成したり、そのコンテンツを変更したりできる唯一のメソッドです。以下の例で、これらの各メソッドを使用する方法を示します。

ノード値の設定および変更

Iris.set() は、value 引数を取り、指定されたアドレスにその値を格納します。

そのアドレスにノードが存在しない場合は、新しいノードが作成されます。

set() メソッドは、サポートされているすべてのデータ型の値を割り当てることができます。以下の例では、set() の最初の呼び出しによって、新しいノードがサブノード・アドレス myGlobal('A') に作成され、このノードの値が文字列 `'first'` に設定されます。2 回目の呼び出しによって、サブノードの値が変更され、整数 `1` に置き換えられます。

```
irisjs.set('first', 'myGlobal', 'A');     // create node myGlobal('A') = 'first'
irisjs.set(1, 'myGlobal', 'A');   // change value of myGlobal('A') to 1.
```

set() は、この例に示すように、サポートされている任意のデータ型の値を作成および変更できるポリモーフィック・アクセサです。

ノード値のインクリメント

Iris.increment() は、整数の number 引数を取り、その数だけノードの値をインクリメントし、そのインクリメントした値を返します。ターゲット・ノードの初期値は、サポートされている任意の数値型ですが、インクリメントした値は整数になります。ターゲット・アドレスにノードがない場合、このメソッドはノードを作成し、値として number 引数を割り当てます。このメソッドはスレッドセーフなアトミック処理を使用して、ノードの値を変更します。したがって、ノードは決してロックされません。

以下の例では、increment() の最初の呼び出しによって、新しいサブノード myGlobal('B') が値 -2 で作成され、返り値が total に割り当てられます。次の 2 回の呼び出しによって、それぞれ -2 だけインクリメントされ、新しい値が total に割り当てられ、ノード値が -6 になるとループが終了します。

```
do {
  var total = irisjs.increment(-2,'myGlobal', 'B');
} while (total > -6)
console.log('total = ' + total + ' and myGlobal('B') = ' + irisjs.get('myGlobal', 'B'))

// Prints: total = -6 and myGlobal('B') = -6
```

ノードまたはノード・グループの削除

Iris.kill() – 指定したノードとそのサブノードすべてを削除します。ルート・ノードが削除された場合、または値を持つすべてのノードが削除された場合、グローバル配列全体が削除されます。

グローバル配列 myGlobal には、最初に以下のノードが含まれます。

```
myGlobal = <valueless node>
  myGlobal('A') = 0
    myGlobal('A',1) = 0
    myGlobal('A',2) = 0
  myGlobal('B') = <valueless node>
    myGlobal('B',1) = 0
```

この例では、その 2 つのサブノードで kill() を呼び出すことで、グローバル配列を削除します。最初の呼び出しによって、ノード myGlobal('A') とその両方のサブノードが削除されます。

```
irisjs.kill('myGlobal', 'A');
// also kills child nodes myGlobal('A',1) and myGlobal('A',2)
```

2 番目の呼び出しによって、値を持つ最後のサブノードが削除され、グローバル配列全体が削除されます。

```
irisjs.kill('myGlobal','B',1);
```

- 親ノード myGlobal('B') は、値がなく、サブノードもなくなったため、削除されます。
- ルート・ノード myGlobal は値がなく、サブノードもなくなったため、グローバル配列全体がデータベースから削除されます。

ルート・ノードで kill() を呼び出すことで、グローバル配列全体を即座に削除できます。

```
irisjs.kill('myGlobal');
```

3.3 グローバル配列のノードの検索

Native SDK は、1 つのグローバル配列の一部または全部に対して反復処理する方法を提供します。以下のトピックでは、さまざまな反復メソッドについて説明します。

- 一連の子ノードにわたる反復 – 指定した親ノードの下のすべての子ノードにわたって反復する方法について説明します。
- next() を使用した反復処理 – 反復処理を詳細に制御するメソッドについて説明します。
- 子ノードおよびノード値のテスト – ノード・レベルに関係なくすべてのサブノードを検索して、値を持つノードを識別する方法について説明します。

3.3.1 一連の子ノードにわたる反復

子ノードは、同じ親ノードの直下にある一連のノードです。親の添え字リストに 1 つの添え字を追加することで、子ノード・アドレスを定義できます。例えば、以下のグローバル配列には、親ノード heroes('dogs') の下に 4 つの子ノードがあります。

heroes グローバル配列

このグローバル配列では、いくつかのヒーロー犬 (および無謀な少年と先導する羊) の名前を添え字として使用します。値は生まれた年です。

```
heroes                                          // root node,    valueless, 2 child nodes
   heroes('dogs')                               // level 1 node, valueless, 4 child nodes
      heroes('dogs','Balto') = 1919             // level 2 node, value=1919
      heroes('dogs','Hachiko') = 1923           // level 2 node, value=1923
      heroes('dogs','Lassie') = 1940            // level 2 node, value=1940, 1 child node
         heroes('dogs','Lassie','Timmy') = 1954 // level 3 node, value=1954
      heroes('dogs','Whitefang') = 1906         // level 2 node, value=1906
   heroes('sheep')                              // level 2 node, valueless, 1 child node
      heroes('sheep','Dolly') = 1996            // level 2 node, value=1996
```

以下のメソッドを使用して反復子を作成し、反復処理の方向を定義し、検索の開始位置を設定します。

- Iris.iterator() は、指定されたターゲット・ノードの子ノードの Iterator のインスタンスを返します。
- Iterator.reversed() – 順方向と逆方向の照合順序の間で反復処理を切り替えます。
- Iterator..startFrom() は、反復子の開始位置を指定の添え字に設定します。添え字は任意の開始ポイントであり、既存のノードを処理する必要はありません。

逆順序での子ノード値の読み取り

以下のコードは、heroes('dogs') の子ノードを逆方向の照合順序で反復処理します (添え字 V から開始します)。

```
// Iterate in reverse, seeking nodes lower than heroes('dogs','V') in collation order
  let iterDogs = irisjs.iterator('heroes','dogs').reversed().startFrom('V');

  let output = '\nDog birth years: ';
  for ([key,value] of iterDogs) {
    output += key + ':' + value + '  ';
  };
  console.log(output);
```

このコードによって、以下のような出力が生成されます。

```
Dog birth years: Lassie:1940  Hachiko:1923  Balto:1919
```

この例では、heroes('dogs') の以下の 2 つのサブノードは無視されます。

- 子ノード heroes('dogs','Whitefang') は検索範囲外にあるため、見つかりません (`Whitefang` は照合順序が `V` より上位です)。
- レベル 3 ノード heroes('dogs','Lassie','Timmy') は、`dogs` ではなく `Lassie` の子であるため、見つかりません。

複数のノード・レベルにわたって反復処理する方法については、この章の最後のセクション（“子ノードおよびノード値のテスト”）を参照してください。

注釈 **照合順序**

ノードが取得される順序は、添え字の照合順序によって決まります。ノードが作成されると、ストレージ定義で指定された照合順に、自動的にノードが格納されます。この例では、heroes('dogs') の子ノードは、作成された順序に関係なく、表示されている順序（`Balto`、`Hachiko`、`Lassie`、`Whitefang`）で格納されます。詳細は、″グローバルの使用法″ の “グローバル・ノードの照合” を参照してください。

3.3.2 next() を使用した反復処理

Native SDK は、標準の next() および以下の返りタイプの反復子のメソッドもサポートしています。

- Iterator.next() ー 次の子ノードに反復子を配置し（次の子ノードがある場合）、プロパティ `done` と `value` を含む配列を返します。これ以上ノードがない場合、`done` プロパティは `false` になります。反復子が作成されると、既定で entries() 返りタイプになります。
- Iterator.entries() ー 返りタイプを、キー（子ノードの最上位の添え字）とノード値の両方を含む配列に設定します。例えば、ノード heroes(,'dogs','Balto') の返り値は `['Balto',1919]` になります。
- Iterator.keys() ー キー（最上位の添え字）のみを返すように返りタイプを設定します。
- Iterator.values() ー ノード値のみを返すように返りタイプを設定します。

以下の例では、next() メソッドを呼び出すたびに、反復子の `done` プロパティと `value` プロパティの現在の値を含む配列に変数 iter が設定されます。keys() メソッドは反復子の作成時に呼び出されたため、`value` プロパティには heroes('dogs') の現在の子ノードのキー（最上位の添え字）のみが含まれます。

next() を使用したノード heroes('dogs') の下の添え字の一覧表示

```
// Get a list of child subscripts under node heroes('dogs')
  let iterDogs = irisjs.iterator('heroes','dogs').keys();
  let output = "\nSubscripts under node heroes('dogs'): ";

  let iter = iterDogs.next();
  while (!iter.done) {
    output += iter.value + ' ';
    iter = iterDogs.next();
  }
  console.log(output);
```

このコードによって、以下のような出力が生成されます。

```
  Subscripts under node heroes('dogs'): Balto Hachiko Lassie Whitefang
```

3.3.3 子ノードおよびノード値のテスト

前の例では、検索の範囲は heroes('dogs') の子ノードに制限されています。グローバル配列 heroes の以下の 2 つの値は異なる親の下にあるため、反復子はこれらの値を見つけられません。

- レベル 3 ノード heroes('dogs','Lassie','Timmy') は、`dogs` ではなく `Lassie` の子であるため、見つかりません。
- レベル 2 ノード heroes('sheep','Dolly') は、`dogs` ではなく `sheep` の子であるため、見つかりません。

グローバル配列全体を検索するには、子ノードを持つすべてのノードを見つけて、子ノードのセットごとに反復子を作成する必要があります。isDefined() メソッドは、必要な情報を提供します。

- Iris.isDefined() – これを使用して、ノードに値、サブノード、またはその両方があるかどうかを確認できます。以下の値を返します。
  - `0` – 指定したノードは存在しません。
  - `1` – ノードは存在し、値があります。
  - `10` – ノードに値はありませんが、子ノードがあります。
  - `11` – ノードには値と子ノードの両方があります。

  返り値を使用して、いくつかの有用なブーリアン値を確認できます。

```
let exists = (irisjs.isDefined(root,subscripts) > 0);      // returns 1, 10, or 11
let hasValue = (irisjs.isDefined(root,subscripts)%10 > 0); // returns 1 or 11
let hasChild = (irisjs.isDefined(root,subscripts) > 9);    // returns 10 or 11
```

以下の例は、2 つのメソッドで構成されます。

- findAllHeroes() は、現在のノードの子ノードを反復処理して、ノードごとに testNode() を呼び出します。testNode() によって現在のノードに子ノードがあると示された場合、findAllHeroes() は次のレベルの子ノードの新しい反復子を作成します。
- testNode() は、heroes グローバル配列のノードごとに呼び出されます。現在のノードで isDefined() を呼び出して、ノードに子ノードがあるかどうかを示すブーリアン値を返します。各ノードのノード情報も出力します。

メソッド findAllHeroes()

この例では、既知の構造を処理して、入れ子になった単純な呼び出しを使用してさまざまなレベルを検索します。構造に任意の数のレベルがあるようなあまり一般的でない場合には、再帰アルゴリズムを使用できます。

```
function findAllHeroes() {
  const root = 'heroes';
  console.log('List all subnodes of root node '+root+':\n'+root)
  let iterRoot = irisjs.iterator(root);
  let hasChild = false;

  // Iterate over children of root node heroes
  for ([sub1,value] of iterRoot) {
    hasChild = testNode(value,root,sub1);

    // Iterate over children of heroes(sub1)
    if (hasChild) {
      let iterOne = irisjs.iterator(root,sub1);
      for ([sub2,value] of iterOne) {
        hasChild = testNode(value,root,sub1,sub2);

        // Iterate over children of heroes(sub1,sub2)
        if (hasChild) {
          let iterTwo = irisjs.iterator(root,sub1,sub2);
          for ([sub3,value] of iterTwo) {
            testNode(value,root,sub1,sub2,sub3); //no child nodes below level 3
          }
        } //end level 2
      }
    } //end level 1
  } // end main loop
} // end findAllHeroes()
```

メソッド testNode()

```
function testNode(value, root, ...subs) {

// Test for values and child nodes
  let state = irisjs.isDefined(root,...subs);
  let hasValue = (state%10 > 0); // has value if state is 1 or 11
  let hasChild = (state > 9);    // has child if state is 10 or 11

// format the node address output string
  let subList = Array.from(subs);
  let level = subList.length-1;
```

```
    let indent = '  ' + String('      ').slice(0,(level*2));
    let address = indent + root+'(' + subList.join() + ')';

  // Add node value to string and note special cases
    if (hasValue) { // ignore valueless nodes
      address += ' = ' + value;
      for (name of ['Timmy','Dolly']) {
        if (name == subList[level]) {
          address += ' (not a dog!)'
        }
      }
    }
    console.log(address);
    return hasChild;
  }
}
```

このメソッドは、以下の行を記述します。

```
List all subnodes of root node heroes:
heroes
  heroes(dogs)
    heroes(dogs,Balto) = 1919
    heroes(dogs,Hachiko) = 1923
    heroes(dogs,Lassie) = 1940
      heroes(dogs,Lassie,Timmy) = 1954  (not a dog!)
    heroes(dogs,Whitefang) = 1906
  heroes(sheep)
    heroes(sheep,Dolly) = 1996  (not a dog!)
```

testNodes() の出力には、heroes('dogs') の子ノードではないために、前の例では見つからなかったノードがいくつか含まれます。

- heroes('dogs','Lassie','Timmy') は、`dogs` ではなく `Lassie` の子です。
- heroes('sheep','Dolly') は、`dogs` ではなく `sheep` の子です。

4

トランザクションとロック

このセクションで説明する項目は以下のとおりです。

- トランザクションの制御 ー トランザクションを処理するためのメソッドについて説明します。
- ロックの取得および解放 ー さまざまなロック・メソッドの使用法について説明します。
- トランザクションでのロックの使用 ー トランザクション内のロックの例を提供します。

4.1 トランザクションの制御

Native SDK には、トランザクションを制御するために以下のメソッドが用意されています。

- Iris.tCommit() ー 1 レベルのトランザクションをコミットします。
- Iris.tStart() ー トランザクションを開始します (このトランザクションは入れ子になっている場合あり)。
- Iris.getotalevel() ー 現在のトランザクション・レベル (トランザクション内でない場合は 0) を返します。
- Iris..tRollback() ー セッション内の開いているトランザクションすべてをロールバックします。
- Iris..tRollbackOne() ー 現在のレベルのトランザクションのみをロールバックします。入れ子になったトランザクションである場合、それより上のレベルのトランザクションはロールバックされません。

以下の例では、3 レベルの入れ子のトランザクションを開始し、各トランザクション・レベルに異なるノードの値を設定します。3 つのノードはすべて出力され、それらに値があることが証明されます。その後、この例では、2 番目と 3 番目のレベルがロールバックされ、最初のレベルがコミットされます。3 つのノードはすべて再び出力され、依然として値があるのは最初のノードのみであることが証明されます。

トランザクションの制御 : 3 レベルの入れ子トランザクションの使用法

```
  const node = 'myGlobal';
  console.log('Set three values in three different transaction levels:');
  for (let i=1; i<4; i++) {
    irisjs.tStart();
    let lvl = irisjs.getTLevel()
    irisjs.set(('Value'+lvl), node, lvl);
    let val = '<valueless>'
    if (irisjs.isDefined(node,lvl)%10 > 0) val = irisjs.get(node,lvl);
    console.log('   ' + node + '(' + i + ') = ' + val + ' (tLevel is ' + lvl + ')');
  }
// Prints: Set three values in three different transaction levels:
//         myGlobal(1) = Value1 (tLevel is 1)
//         myGlobal(2) = Value2 (tLevel is 2)
//         myGlobal(3) = Value3 (tLevel is 3)
```

```
  console.log('Roll back two levels and commit the level 1 transaction:');
  let act = ['  tRollbackOne','  tRollbackOne','       tCommit'];
  for (let i=3; i>0; i--) {
    if (i>1) {irisjs.tRollbackOne();} else {irisjs.tCommit();}
    let val = '<valueless>'
    if (irisjs.isDefined(node,i)%10 > 0) val = irisjs.getString(node,i);
    console.log(act[3-i]+' (tLevel='+irisjs.getTLevel()+'): '+node+'('+i+') = '+val);
  }

// Prints: Roll back two levels and commit the level 1 transaction:
//           tRollbackOne (tLevel=2): myGlobal(3) = <valueless>
//           tRollbackOne (tLevel=1): myGlobal(2) = <valueless>
//                tCommit (tLevel=0): myGlobal(1) = Value1
```

4.2 ロックの取得および解放

クラス Iris の以下のメソッドを使用して、ロックを取得および解放します。どちらのメソッドも、ロックが共有であるか排他であるかを指定する lockMode 引数を取ります。

- Iris.lock() ー lockMode、timeout、lockReference、および subscripts 引数を取り、ノードをロックします。lockMode 引数は、前に保持されたロックを解放するかどうかを指定します。このメソッドは、ロックを取得できない場合、事前に定義した間隔が経過するとタイムアウトします。
- Iris.unlock() ー lockMode、lockReference、および subscripts 引数を取り、ノードのロックを解放します。

以下の引数値を使用できます。

- lockMode ー 共有ロックを表す S という文字、エスカレート・ロックを表す E という文字、または共有およびエスカレート・ロックを表す SE という文字の組み合わせ。既定値は空の文字列 (排他の非エスカレート) です。
- lockReference ー 曲折アクセント記号 (^) で始まり、その後にグローバル名が続く文字列 (例えば、単なる `myGlobal` ではなく `^myGlobal`)。ほとんどのメソッドで使用される `globalName` パラメータとは異なり、`lockReference` パラメータには先頭に曲折アクセント記号を付ける必要があります。`globalName` ではなく `lockReference` を使用するのは、lock() および unlock() のみです。
- timeout ー ロックの取得を待機する秒数。この秒数を経過するとタイムアウトになります。

注釈　管理ポータルを使用して、ロックを検証できます。System Operation > Locks に移動して、システムでロックされている項目のリストを表示します。

4.3 トランザクションでのロックの使用

このセクションでは、前に説明したメソッドを使用して、トランザクション内での増分ロックについて説明します (“トランザクションの制御” および “ロックの取得および解放” を参照してください)。管理ポータルを開き、System Operation > Locks.に移動することによって、システムのロックされた項目のリストを表示できます。以下のコードの alert() の呼び出しによって、実行が一時停止され、リストが変更されるたびにリストを調べることができます。

現在保持されているロックをすべて解放するには、2 つの方法があります。

- Iris.releaseAllLocks() ー この接続で現在保持されているロックをすべて解放します。
- 接続オブジェクトの close() メソッドが呼び出されると、すべてのロックおよびその他の接続リソースが解放されます。

以下の例に、さまざまなロック・メソッドおよび解放メソッドを示します。

トランザクションでの増分ロックの使用

```
  irisjs.set('exclusive node','nodeOne');
  irisjs.set('shared node','nodeTwo');

// unlike global names, lock references *must* start with circumflex
  const nodeOneRef = '^nodeOne';
  const nodeTwoRef = '^nodeTwo';

  try {
    irisjs.tStart();
    irisjs.lock('E',10,nodeOneRef,''); // lock nodeOne exclusively
    irisjs.lock('S',10,nodeTwoRef,''); // lock nodeTwo shared
    console.log('Exclusive lock on nodeOne and shared lock on nodeTwo');

    alert('Press return to release locks individually');
    irisjs.unlock('D',nodeOneRef,''); // release nodeOne after transaction
    irisjs.unlock('I',nodeTwoRef,''); // release nodeTwo immediately

    alert('Press return to commit transaction');
    irisjs.tCommit();
  }
  catch { console.log('error'); }
```

トランザクションでの非増分ロックの使用

```
// lock nodeOne non-incremental, nodeTwo shared non-incremental
  irisjs.lock('',10,nodeOneRef,'');

  alert('Exclusive lock on nodeOne, return to lock nodeOne non-incrementally');
  irisjs.lock('S',10,nodeTwoRef,'');

  alert('Verify that only nodeTwo is now locked, then press return');
```

トランザクションで releaseAllLocks() を使用して、すべての増分ロックを解放

```
// lock nodeOne shared incremental, nodeTwo exclusive incremental
  irisjs.lock('SE',10,nodeOneRef,'');
  irisjs.lock('E',10,nodeTwoRef,'');

  alert('Two locks are held (one with lock count 2), return to release both locks');
  irisjs.releaseAllLocks();

  alert('Verify both locks have been released, then press return');
```

5

Native SDK for Node.js のクイック・リファレンス

この章は、external:"intersystems-iris-native" のモジュールのメンバである、以下のクラスのクイック・リファレンスです

- クラス Connection は、サーバへの接続を確立します。
- クラス Iris は、主な機能を提供します。
- クラス Iterator は、グローバル配列を操作するためのメソッドを提供します。

注釈　この章は、このドキュメントの読者の利便性を目的としたものであり、Native SDK の最終的なリファレンスではありません。これらのクラスに関する最も包括的な最新情報については、SDK オンライン・ドキュメントを参照してください。

5.1 用途別のメソッドのリスト

クラス Connection

Connection クラスは、サーバへの接続をカプセル化します。external:"intersystems-iris-native" のメソッド createConnection() によって、Connection のインスタンスが作成され、サーバに接続されます。

- close() ー 接続を閉じます。
- createIris() ー この接続のために Iris のインスタンスを作成します。
- isClosed() ー 接続が閉じられている場合、`true` を返します。
- isUsingSharedMemory() ー 接続で共有メモリが使用されている場合、`true` を返します。

クラス Iris

Iris クラスは、Native SDK のほとんどの機能を提供します。Iris のインスタンスは、Connection.createIris() によって作成されます。メソッドを用途別にまとめて以下にリストします。

クラス Iris : グローバルの反復処理と管理

- getAPIVersion() ー このバージョンの Native SDK のバージョン文字列を返します。
- getServerVersion() ー 現在接続されているサーバのバージョン文字列を返します。
- increment() ー グローバル・ノードの値を、指定された数だけインクリメントします。
- isDefined() ー グローバル・ノードが存在するかどうか、およびそれにデータが含まれているかどうかを確認します。

- iterator() ー Iterator のインスタンスを返します。
- kill() ー 下位ノードを含め、グローバル・ノードを削除します。

クラス Iris : ノード値アクセサ

- get() ー 指定されたノード値を返します。
- getBoolean() ー ノード値をブーリアンとして返します。
- getBytes() ー ノード値を ArrayBuffer として返します。
- getNumber() ー ノード値を数値として返します。
- getString() ー ノード値を文字列として返します。

クラス Iris : トランザクションとロック

- lock() ー グローバルで増分ロックを実行し、成功すると `true` を返します。
- unlock() ー グローバルで即時または遅延アンロックを実行します。
- releaseAllLocks() ー セッションに関連付けられたすべてのロックを解放します。
- getTLevel() ー 現在のトランザクション・レベル (トランザクション内でない場合は `0`) を返します。
- tCommit() ー 現在のトランザクションをコミットして、トランザクション・レベルをデクリメントします。
- tRollback() ー セッション内の開いているトランザクションすべてをロールバックします。
- tRollbackOne() ー 現在のレベルのトランザクションのみをロールバックします。
- tStart() ー トランザクションを開始して、トランザクション・レベルをインクリメントします。

クラス Iris : クラス・メソッドおよび関数の呼び出し

- classMethodValue() ー ユーザ定義の ObjectScript メソッドを呼び出して、返り値を取得します。
- classMethodVoid() ー ユーザ定義の ObjectScript メソッドを呼び出して、返り値を無視します。
- function() ー ユーザ定義の ObjectScript ルーチンを呼び出して、返り値を取得します。
- procedure() ー ユーザ定義の ObjectScript ルーチンのプロシージャを呼び出します。

クラス Iterator

Iterator クラスは、一連のノードを反復処理するメソッドを提供します。Iterator のインスタンスは、Iris.iterator() によって作成されます。

- next() ー 次の兄弟ノードに反復子を配置します。
- startFrom() ー 開始位置を指定の添え字に設定します。
- reversed() ー 順方向と逆方向の照合順序の間で反復処理を切り替えます。
- entries() ー 添え字とノード値を含む配列に返りタイプを設定します。
- keys() ー ノードの添え字 (キー) のみを返すように返りタイプを設定します。
- values() ー ノード値のみを返すように返りタイプを設定します。

5.2 クラス Connection

クラス Connection は external:"intersystems-iris-native" のメンバです。Connection のインスタンスは、intersystems-iris-native メソッド createConnection() によって作成されます。

createConnection()

intersystems-iris-native createConnection() は、サーバへの接続を確立します。

```
(static) createConnection(connectionInfo)   {external:"intersystems-iris-native".Connection}
```

パラメータ：

- `connectionInfo` — 接続プロパティを含むオブジェクト。connectionInfo の有効なプロパティは以下のとおりです。
  - `host` — ホスト・マシンのアドレスを含む必須の文字列。
  - `port` — ポート番号を指定する必須の整数。
  - `ns` — データベース・ネームスペースを含む必須の文字列。
  - `user` — ユーザ名を含む文字列。
  - `pwd` — パスワードを含む文字列。
  - `sharedmemory` — 使用可能な場合、共有メモリを使用するかどうかを示すブーリアン値 (既定値は `true`)。
  - `timeout` — 接続試行がタイムアウトするまでに待機するミリ秒数を指定する整数 (既定値は `10000`)。
  - `logfile` — この接続のログ・ファイルのフル・パスと名前を指定する文字列。指定しない場合、ログ・ファイルは書き込まれません。

  例えば、以下のコードの断片は connectionInfo のすべてのプロパティを定義し、conn という名前の Connection のインスタンスを作成します。

```
const IRISNative = require('intersystems-iris-native')

let connectionInfo = {
  host: '127.0.0.1',
  port: 51773,
  ns: 'USER',
  user: '_SYSTEM',
  pwd: 'SYS',
  sharedmemory: true
  timeout: 5,
  logfile: 'C:\temp\mylogfile.log'
};

const conn = IRISNative.createConnection(connectionInfo);
```

close()

Connection.close() は、サーバへの接続を閉じます。

```
close()
```

createIris()

Connection.createIris() は、この Connection によってサーバに接続されているクラス Iris のインスタンスを返します。

```
createIris()    {external:"intersystems-iris-native".Iris}
```

isClosed()

Connection.isClosed() は、接続が現在閉じられている場合は `true`、開いている場合は `false` を返します。

```
isClosed()   {boolean}
```

isUsingSharedMemory()

Connection.isUsingSharedMemory() は、接続で共有メモリが使用されている場合は `true`、使用されていない場合は `false` を返します。

```
isUsingSharedMemory()   {boolean}
```

5.3 クラス Iris

クラス Iris は external:"intersystems-iris-native" のメンバです。Iris のインスタンスは、Connection.createIris() を呼び出すことによって作成されます。

クラス Iris は、グローバル関数、関数とプロシージャの呼び出し、ロック、およびトランザクションを実装する主要な Native SDK モジュールです。

classMethodValue()

Iris.classMethodValue() は、クラス・メソッドを呼び出して、0 個以上の引数を渡し、呼び出されたメソッドの返り値を返します。

```
classMethodValue(className, methodName, ...args)   {any}
```

パラメータ：

- `className` — 呼び出されるメソッドが属するクラスの完全修飾名。
- `methodName` — クラス・メソッドの名前。
- `args` — サポートされている型の 0 個以上のメソッド引数（"ObjectScript メソッドおよび関数の呼び出し" を参照してください）。末尾の引数は省略できます。

classMethodVoid()

Iris.classMethodVoid() は、クラス・メソッドを呼び出して、0 個以上の引数を渡し、何も返しません。

```
classMethodVoid(className, methodName, ...args)
```

パラメータ：

- `className` — 呼び出されるメソッドが属するクラスの完全修飾名。
- `methodName` — クラス・メソッドの名前。
- `args` — サポートされている型の 0 個以上のメソッド引数（"ObjectScript メソッドおよび関数の呼び出し" を参照してください）。末尾の引数は省略できます。

function()

Iris.function() は、関数を呼び出して、0 個以上の引数を渡し、呼び出された関数の返り値を返します。

```
function(routineName, functionName, ...args)   {any}
```

パラメータ：

- `routineName` — 関数を含むルーチンの名前。
- `functionName` — 呼び出す関数の名前。
- `args` — サポートされている型の 0 個以上の関数引数 (“ObjectScript メソッドおよび関数の呼び出し” を参照してください)。末尾の引数は省略できます。

get()

Iris.get() は、グローバル・ノードの値を返します。ノード値が空の文字列の場合は `false` を返し、指定されたノード・アドレスに値がない場合は `null` を返します。

```
get(globalName, ...subscripts)   {any}
```

パラメータ：

- `globalName` — グローバル名
- `subscripts` — ターゲット・ノードを指定する添え字の配列

getAPIVersion()

Iris.getAPIVersion() は、Native SDK バージョン文字列を返します。

```
getAPIVersion()   {string}
```

getBoolean()

Iris.getBoolean() は、ノード値を boolean として返します。ノード値が空の文字列の場合は `false` を返し、指定されたノード・アドレスに値がない場合は `null` を返します。

```
getBoolean(globalName, ...subscripts)   {boolean}
```

パラメータ：

- `globalName` — グローバル名
- `subscripts` — ターゲット・ノードを指定する添え字の配列

getBytes()

Iris.getBytes() は、ノード値を ArrayBuffer として返します。ノード値が空の文字列の場合は `false` を返し、指定されたノード・アドレスに値がない場合は `null` を返します。

```
getBytes(globalName, ...subscripts)   {ArrayBuffer}
```

パラメータ：

- `globalName` — グローバル名
- `subscripts` — ターゲット・ノードを指定する添え字の配列

getNumber()

Iris.getNumber() は、ノード値を number として返します。ノード値が空の文字列の場合は false を返し、指定されたノード・アドレスに値がない場合は null を返します。

```
getNumber(globalName, ...subscripts)   {number}
```

パラメータ：

- globalName ― グローバル名
- subscripts ― ターゲット・ノードを指定する添え字の配列

getServerVersion()

Iris.getServerVersion() は、現在接続されているサーバのバージョン文字列を返します。

```
getServerVersion()   {string}
```

getString()

Iris.getString() は、ノード値を string として返します。ノード値が空の文字列の場合は false を返し、指定されたノード・アドレスに値がない場合は null を返します。

```
getString(globalName, subscripts)   {string}
```

パラメータ：

- globalName ― グローバル名
- subscripts ― ターゲット・ノードを指定する添え字の配列

getTLevel()

Iris.getTLevel() は、入れ子になった現在のトランザクションのレベルを返します。開いているトランザクションが 1 つのみである場合は 1 を返します。開いているトランザクションがない場合は 0 を返します。これは、$TLEVEL 特殊変数の値のフェッチと同じです。詳細と例は、“トランザクションとロック”を参照してください。

```
getTLevel()   {number}
```

increment()

Iris.increment() は、指定されたノードを incrementBy の整数値だけインクリメントします。指定されたノード・アドレスに値がない場合、ノードが作成され、その値が incrementBy の値に設定されます。ノードの新しい整数値を返します。

```
increment(incrementBy, globalName, ...subscript)   {number}
```

パラメータ：

- incrementBy ― このノードを設定する数値 (null 値はグローバルを 0 に設定し、小数値は整数に切り捨てられます)。
- globalName ― グローバル名
- subscripts ― ターゲット・ノードを指定する添え字の配列

isDefined()

Iris.isDefined() は、グローバルが存在してデータがあるかどうかを確認します（"$DATA" を参照してください）。ノードが存在しない場合は 0 を返し、グローバル・ノードが存在し、それにデータがある場合は 1 を返します。ノードに値はないが、下位ノードがある場合は 10 を返します。データも下位ノードもある場合は 11 を返します。詳細と例は、“子ノードおよびノード値のテスト” を参照してください。

```
isDefined(globalName, ...subscript)   {number}
```

パラメータ：

- `globalName` — グローバル名
- `subscripts` — このノードの添え字の配列

iterator()

Iris.iterator() は、指定されたノードの Iterator オブジェクト（“クラス Iterator” を参照）を返します。

```
iterator(globalName, ...subscript)   {external:"intersystems-iris-native".Iterator}
```

パラメータ：

- `globalName` — グローバル名
- `subscripts` — ターゲット・ノードを指定する添え字の配列

kill()

Iris.kill() は、下位ノードを含め、グローバル・ノードを削除します。

```
kill(globalName, ...subscript)
```

パラメータ：

- `globalName` — グローバル名
- `subscripts` — このノードの添え字の配列

lock()

Iris.lock() は、グローバルをロックし、成功すると `true` を返します。このメソッドは増分ロックを実行し、ObjectScript で提供されるロック前の暗黙的なアンロック機能は実行しません。

```
lock(lockMode, timeout, lockReference, ...subscript)   {boolean}
```

パラメータ：

- `lockMode` — 次のいずれかの文字列：共有ロックの場合は `"S"`、エスカレート・ロックの場合は `"E"`、両方の場合は `"SE"`、どちらでもない場合は `""`。空の文字列が既定モードです（非共有または非エスカレート）。
- `timeout` — ロックの取得を待機する秒数
- `lockReference` — 曲折アクセント記号 (^) で始まり、その後にグローバル名が続く文字列（例えば、単なる `myGlobal` ではなく `^myGlobal`）。

注意：ほとんどのメソッドで使用される globalName パラメータとは異なり、lockReference パラメータには先頭に曲折アクセント記号を付ける必要があります。globalName ではなく lockReference を使用するのは、lock() および unlock() のみです。

- subscripts ― このノードの添え字の配列。

procedure()

Iris.procedure() はプロシージャを呼び出し、0 個以上の引数を渡します。

```
procedure(routineName, procedureName, ...args)
```

パラメータ：

- routineName ― プロシージャを含むルーチンの名前。
- procedureName ― 呼び出すプロシージャの名前。
- args ― サポートされている型の 0 個以上のプロシージャ引数（“ObjectScript メソッドおよび関数の呼び出し”を参照してください）。末尾の引数は省略できます。

releaseAllLocks()

Iris..releaseAllLocks() は、セッションに関連付けられたすべてのロックを解放します。

```
releaseAllLocks()
```

set()

Iris.set() は、指定されたノードをサポートされている任意のデータ型の値に設定します（値が null の場合は ""）。

```
set(value, globalName, ...subscript)
```

パラメータ：

- value ― サポートされている任意のデータ型の値（null 値の場合、グローバルは "" に設定されます）。
- globalName ― グローバル名
- subscripts ― このノードの添え字の配列

tCommit()

Iris.tCommit() は、現在のトランザクションをコミットして、トランザクション・レベルをデクリメントします。詳細と例は、“トランザクションとロック”を参照してください。

```
tCommit()
```

tRollback()

Iris..tRollback() は、セッション内の開いているトランザクションすべてをロールバックします。詳細と例は、“トランザクションとロック”を参照してください。

```
tRollback()
```

tRollbackOne()

Iris..tRollbackOne() は、現在のレベルのトランザクションのみをロールバックします。入れ子になったトランザクションである場合、それより上のレベルのトランザクションはロールバックされません。詳細と例は、“トランザクションとロック”を参照してください。

```
tRollbackOne()
```

tStart()

Iris.tStart() は、トランザクションを開始して、トランザクション・レベルをインクリメントします。詳細と例は、“トランザクションとロック”を参照してください。

```
tStart()
```

unlock()

Iris..unlock() は、グローバルをアンロックします。このメソッドは増分アンロックを実行し、ObjectScript でも提供されるロック前の暗黙的なアンロック機能は実行しません。

```
unlock(lockMode, lockReference, ...subscript)
```

パラメータ：

- `lockMode` — 次のいずれかの文字列：共有ロックの場合は `"S"`、エスカレート・ロックの場合は `"E"`、両方の場合は `"SE"`、どちらでもない場合は `""`。空の文字列が既定モードです（非共有または非エスカレート）。
- `lockReference` — 曲折アクセント記号 (^) で始まり、その後にグローバル名が続く文字列（例えば、単なる `myGlobal` ではなく `^myGlobal`）。

  注意：ほとんどのメソッドで使用される `globalName` パラメータとは異なり、`lockReference` パラメータには先頭に曲折アクセント記号を付ける必要があります。`globalName` ではなく `lockReference` を使用するのは、lock() および unlock() のみです。
- `subscripts` — このノードの添え字の配列。

5.4 クラス Iterator

クラス Iterator は external:"intersystems-iris-native" のメンバです。Iterator のインスタンスは、Iris.iterator() を呼び出すことによって作成されます。詳細と例は、“グローバル配列のノードの検索”を参照してください。

next()

Iterator.next() は、照合順で次の兄弟ノードに反復子を配置し、done プロパティと value プロパティを含むオブジェクトを返します。反復子が最後まで到達している場合、done は `true` になり、最後まで到達していない場合は `false` になります。

```
next()   {any}
```

done が `false` の場合、以下のいずれかのメソッドによって value プロパティの返りタイプを設定できます。

- entries() は、value が配列として返されるようにします。配列では、value(0) は添え字で、value(1) はノード値です（反復子が作成される際には、この配列が既定になります）。
- keys() は、value が現在の添え字のみを返すようにします。

- values() は、value が現在のノードの値のみを返すようにします。

startFrom()

Iterator..startFrom() は、反復子の開始位置を指定の添え字に設定します。開始位置は、有効なサブノードでなくてもかまいません。連鎖の場合は `this` を返します。

```
startFrom(subscript)   {external:"intersystems-iris-native".Iterator}
```

パラメータ：

- `subscript` – 開始ポイントとして使用する添え字。既存のノードを指定する必要はありません。

next() を呼び出して、照合順で次の既存のノードに進めるまで、反復子はノードを指しません。

reversed()

Iterator.reversed() は、順方向と逆方向の照合順序の間で反復処理を切り替えます。既定で、反復子は定義の際に順方向の反復処理に設定されます。連鎖の場合は `this` を返します。

```
reversed()   {external:"intersystems-iris-native".Iterator}
```

entries()

Iterator.entries() は、next().value がノードの添え字 (value(0)) とノード値 (value(1)) を含む配列でなければならないと指定します。連鎖の場合は `this` を返します。

```
entries()   {external:"intersystems-iris-native".Iterator}
```

keys()

Iterator.keys() は、next().value にノードの添え字 (キー) のみを含める必要があると指定します。連鎖の場合は `this` を返します。

```
keys()   {external:"intersystems-iris-native".Iterator}
```

values()

Iterator.values() は、next().value にはノードの値のみを含める必要があると指定します。連鎖の場合は `this` を返します。

```
values()   {external:"intersystems-iris-native".Iterator}
```