SYBASE®
An SAP® Company

SPLASH プログラマーズ・リファレンス

Sybase Event Stream Processor 5.0

ドキュメント ID： DC01744-01-0500-01
改訂： 2011 年 12 月

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

目次

第 1 章：SPLASH プログラミング言語 1

変数宣言と型宣言 1
関数 2
Flex 演算子による CCL への SPLASH の統合 3

第 2 章：文 7

式の文 7
ブロックの文 7
条件文 8
output 文 8
while 文 9
for ループ 9
制御文 10
switch 文 11

第 3 章：データ構造 13

レコード・イベント 13
XML 値 15
ベクトル 16
辞書 19
辞書のオペレーション 20
ストリーム 21
ストリーム反復子 22
イベント・キャッシュ 23
手動による挿入 24
バケットの変更 25

バケット・サイズの管理 25
レコードの保持 26
順序付け 26
イベント・キャッシュのオペレーション 27

索引 29

第 1 章 SPLASH プログラミング言語

この章では、Flex 演算子とグローバルとローカルの宣言ブロック内で使用する SPLASH (Streaming Platform LAnguage SHell) プログラミング言語について説明します。

SPLASH の構文は、式言語と、文のブロックに対する C のような構文を組み合わせたものです。C と同様に、ブロック内に変数宣言があり、変数への代入を行うための文、条件、ループがあります。 SPLASH 内では、スカラ型のほかに、レコードの型、レコードのコレクション、コレクション内のレコードに対する反復子など、その他のデータ型も使用できます。 コメントは、`/*-*/` のペアで囲まれたテキストのブロックとして、または `//` 付きの行コメントとして表示できます。

変数宣言と型宣言

SPLASH の変数宣言は C と似ています。変数名の前にデータ型を指定し、宣言の最後にセミコロンを付けます。 変数に初期値を代入することもできます。

次は、SPLASH の宣言例です。

```
integer a, r;
float b := 9.9;
string c, d := 'dd';
[ integer key1; string key2; | string data; ] record;
```

最初の 3 つの宣言は、データ型 `integer`、`float`、`string` のスカラ変数の宣言です。 最初の宣言には変数が 2 つあります。 2 つ目の宣言では、変数 “b” は 9.9 に初期設定されます。 3 つ目の宣言では、変数 “c” は初期設定されませんが、“d” は初期設定されます。 4 つ目の宣言は、カラムが 3 つあるレコードの宣言です。 キー・カラム “key1” と “key2” が最初にリストされ、その後に `|` 文字が入ります。 残りのカラム “data” は非キー・カラムです。 新しいレコードを作成する場合の構文は、この構文タイプと同様です。

`typeof` 演算子は、変数を宣言する便利な方法として使用できます。 たとえば、`rec1` が式で、型が `[ integer key1; string key2; | string data; ]` の場合は、次のようになります。

```
typeof(rec1) rec2;
```

上記の宣言は、次の宣言と同じです。

```
[ integer key1; string key2; | string data; ] rec2;
```

SPLASH の型宣言も C と似ています。typedef 演算子は、型の式のシノニムを定義する方法として使用できます。

```
typedef float newFloatType;
typedef [ integer key1; string key2; | string dataField; ] rec_t;
```

これらの宣言では、それぞれ浮動小数点数型と所定のレコード型のシノニムとして新たに newFloatType と rec_t が作成されます。 これらの名前を後続の変数宣言で使用すれば、読みやすさを向上させ、宣言のサイズを抑えることができます。

```
newFloatType var1;
rec_t var2;
```

関数

SPLASH では独自の関数を作成できます。これらは、グローバルなブロック (任意のストリームで使用する場合) でもローカルなブロックでも宣言できます。 関数は、内部で他の関数を呼び出すことも、自身を再帰的に呼び出すこともできます。

SPLASH 関数の構文は C に似ています。通常、関数は次のように記述します。

```
type functionName(type1 arg1, ..., typen argn) { ... }
```

各“関数型”は SPLASH のデータ型で、各 arg は引数の名前です。 {...} の内側には、任意の SPLASH 文を指定できます。 関数から返される値は、内側の return 文から返される値です。

例を示します。

```
int32 factorial(integer x) {
   if (x <= 0) {
     return 1;
   } else {
     return factorial(x-1) * x;
   }
 }
string odd(integer x) {
   if (x = 1) {
      return 'odd';
   } else {
      return even(x-1);
   }
 }
```

```
string even(integer x) {
   if (x = 0) {
     return 'even';
   } else {
    return odd(x-1);
   }
}
int32 sum(integer x, integer y) { return x+y; }
string getField([ integer k; | string data;] rec) { return rec.data;}
```

1 つ目の関数は再帰的です。2 つ目と 3 つ目は相互に再帰的です。C と異なり、“odd” 関数を宣言するために “even” 関数のプロトタイプは必要ありません。最後の 2 つの関数は、複数の引数とレコード入力を示します。

SPLASH 関数の実際の使用目的は、計算を 1 回定義して、デバッグすることです。 たとえば、現在の価格、満期までの日数、将来の値上げの予測に基づいて、債券の価値を計算する方法があるとします。 この関数を次のように記述して、プロジェクト内の多くの場所で使用することができます。

```
float bondValue(float currentPrice,
                integer daysToMature,
                float inflation)
{
  ...
}
```

Flex 演算子による CCL への SPLASH の統合

SPLASH で記述されたプロシージャは、CCL の Flex 演算子を使用してプロジェクトに統合します。

SPLASH で記述されたプロシージャは、スタンドアロン・プログラムではありません。 これらは、主に CCL で記述された Sybase® Event Stream Processor プロジェクトで使用するためのものです。次の例は、CCL の Flex 演算子を使用して SPLASH コードを組み込む完全なプロジェクトを示します。

以下のプロジェクトは、各銘柄記号の上位 3 つの価格を示します。

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
```

```
    Price float,
    Shares integer
)
;

/* ******************************************************
 * Create a Nasdaq Trades Input Window
 */
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;

/* ******************************************************
 * Use Case a:
 *         Keep records corresponding to only the top three
 * distinct values. Delete records that falls of the top
 * three values.
 *
 * Here the trades corresponding to the top three prices
 * per Symbol is maintained. It uses
 * - eventcaches
 * - local UDF
 */
CREATE FLEX Top3TradesFlex
    IN QTrades
    OUT OUTPUT WINDOW Top3Trades SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
    BEGIN
        DECLARE
            eventCache(QTrades[Symbol], manual, Price asc)
tradesCache;
            /*
             * Inserts record into cache if in top 3 prices and
returns
             * the record to delete or just the current record if it
was
             * inserted into cache with no corresponding delete.
             */
            typeof(QTrades) insertIntoCache( typeof(QTrades)
qTrades )
            {
               // keep only the top 3 distinct prices per symbol in
the
                // event cache
                integer counter := 0;
                typeof (QTrades) rec;
                long cacheSz := cacheSize(tradesCache);
                while (counter < cacheSz) {
                    rec := getCache( tradesCache, counter );
                 if( round(rec.Price,2) = round(qTrades.Price,2) ) {
                        // if the price is the same update
                        // the record.
                        deleteCache(tradesCache, counter);
                        insertCache( tradesCache, qTrades );
                        return rec;
                        break;
```

```
                    } else if( qTrades.Price < rec.Price) {
                        break;
                    }
                    counter++;
                }

                //Less than 3 distinct prices
                if(cacheSz < 3) {
                    insertCache(tradesCache, qTrades);
                    return qTrades;
                } else {   //Current price is > lowest price
                    //delete lowest price record.
                    rec := getCache(tradesCache, 0);
                    deleteCache(tradesCache, 0);
                    insertCache(tradesCache, qTrades);
                    return rec;
                }

                return null;
            }
        END;

        ON QTrades {
            keyCache( tradesCache, [Symbol=QTrades.Symbol;|] );
            typeof(QTrades) rec := insertIntoCache( QTrades );
            if(rec.Id) {
                //When id does not match current id it is a
                //record to delete
                if(rec.Id <> QTrades.Id) {
                    output setOpcode(rec, delete);
                }
                output setOpcode(QTrades, upsert);
            }
        };
    END;
```

以下のプロジェクトでは、30 秒間データを収集して、目的の出力値を計算します。

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;

/* ***************************************************
 * Create a Nasdaq Trades Input Window
 */
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
```

```
;

/* ****************************************************
 * Use Case b:
 * Perform a computation every N seconds for records
 * arrived in the last N seconds.
 *
 * Here the Nasdaq trades data is collected for 30 seconds
 * before being released for further computation.
 */
CREATE FLEX PeriodicOutputFlex
    IN QTrades
    OUT OUTPUT WINDOW QTradesPeriodicOutput SCHEMA TradesSchema
PRIMARY KEY(Symbol,Price)
    BEGIN
        DECLARE
            dictionary(typeof(QTrades), integer) cache;
                END;
        ON QTrades {
                //Whenever a record arrives just insert into
dictionary.
              //The key of the dictionary is the key to the record.
            cache[QTrades] := 0;
        };
        EVERY 30 SECONDS {
                //Cycle through event cache and output all the rows
                //and delete the rows.
                for (rec in cache) {
                        output setOpcode(rec, upsert);
                }
                clear(cache);
        };
    END;

/**
 * Perform a computation from the periodic output.
 */
CREATE OUTPUT WINDOW QTradesSymbolStats
PRIMARY KEY DEDUCED
AS SELECT
    q.Symbol,
    MIN(q.Price)        Minprice,
    MAX(q.Price)        MaxPrice,
    sum(q.Shares * q.Price)/sum(q.Shares) Vwap,
    count(*) TotalTrades,
    sum(q.Shares) TotalVolume
FROM
    QTradesPeriodicOutput q
GROUP BY
    q.Symbol
;
```

第 2 章 文

SPLASH には、式とブロックに対する文形式のほかに、条件、output、“break” と “continue”、“while” と “for” ループの文形式があり、文のブロックもあります。

式の文

式の最後にセミコロンを付けることで、式を文にすることができます。

例を示します。

```
setOpcode(input, 3);
```

代入は式なので、代入も同じように文にすることができます。 たとえば、次の文は変数 “address” に文字列を代入します。

```
address := '550 Broad Street';
```

ブロックの文

複数の文は中カッコで囲み、必要に応じて変数宣言を挿入して、文のシーケンスにすることができます。

例を示します。

```
{
  float d := 9.99;
  record.b := d;
}
```

文の間に変数宣言を挿入できます。

```
{
  float pi := 3.14;
  print (string(pi));
  float e := 2.71;
  print (string(e));
}
```

条件文

条件文を使用して、特定の条件が true か false かに基づくアクションを指定します。SPLASH の条件文は C の条件文と同じ構文を使用します。

例を示します。

```
if (record.a = 9)
  record.b := 9.99;
```

条件には、オプションの “else” 文がある場合もあります。

```
if (record.a = 9)
  record.b := 9.99;
else {
  float d := 10.9;
  record.b := d;
}
```

output 文

`output` 文では、イベントを下流のストリームに送信するスケジュールを設定します。また、イベントを関連ストアに入れるスケジュールも設定します。

次に例を示します。

```
output [k = 10; | d = 20;];
```

Flex 演算子がステートレスなストアに出力を送信する場合、非挿入を `output` する試みはすべて拒否されます。

注意： 複数の output 文を使用して 1 つのイベントを処理できます。この場合、出力はトランザクション・ブロックとして収集されます。 同様に、Flex 演算子がトランザクション・ブロックを受け取ると、トランザクション・ブロック全体が処理されて、出力はすべて別のトランザクション・ブロックに収集されます。 つまり、下流のストリームと、ストリームに格納されたレコード・データは、イベン

ト全体 (1 つのイベントまたはトランザクション・ブロック) が処理されるまで変更されません。

while 文

while 文は条件処理の形式です。while 文を使用して、特定の条件が満たされた場合に実行するアクションを指定します。while 文は C の while 文と同じ構文を使用し、ループとして処理されます。

例を示します。

```
while (not(isnull(record))) {
  record.b := record.a + record.b;
  record := getNext(record_iterator);
}
```

for ループ

通常、ループは “for” ループを使用してコーディングします。これは、入力ストリーム内の一部またはすべてのレコードや、ベクトルまたは辞書内のすべてのデータをループ処理する便利な手段です。

入力ストリーム “input_stream” 内のすべてのレコードをループ処理するには、次のように指定します。

```
for (record in input_stream) {
  ...
}
```

変数 `record` は新しい変数です。ここでは、任意の名前を使用できます。 スコープはループ内のステートメントまたはステートメントのブロックです。ループの外側では意味を持ちません。 また、フィールドの特定の値でレコードを検索する場合に等価条件を設定することもできます。 次に例を示します。

```
for (record in input_stream where c=10, d=11) {
  ...
}
```

この文では、ループ処理の動作は同じですが、ループ処理の対象が c フィールドが 10 で、d フィールドが 11 のレコードに制限されます。 キー・フィールドで検索する場合、ループの実行回数は 1 回ですが、ストリームの基本となるインデックスを使用するため、実行は著しく高速になります。

ベクトル “vec1” 内の値をループ処理するには、次のように指定します。ここで、`val` は任意の新しい変数です。

```
for (val in vec1) {
 ...
}
```

ベクトルの終わりに達するか、ベクトルの値が null であると、ループは停止します。

辞書 “dict1” 内の値をループ処理するには、次のように指定します。ここで、`key` は任意の新しい変数です。

```
for (key in dict1) {
 ...
}
```

一般的には、ループの内側で、式 `dict1[key]` を使用して、辞書に保持されているその特定のキーの値を取得します。

制御文

制御文を使用して、`while` ループと `for` ループの両方を終了または再開します。

`break` 文は最も内側のループを終了します。`continue` 文は最も内側のループを再開します。

`return` 文は処理を停止し、値を返します。 これは、SPLASH の関数で非常に役立ちます。

`exit` 文は処理を停止します。

switch 文

switch 文は条件の特殊な形式です。

たとえば、次のように記述できます。

```
switch(intvar*2) {
  case 0: print('case0'); break;
  case 1+1: print('case2'); break;
  default: print('default'); break;
}
```

この文は、intvar*2 の値が 0 の場合は “case0” を出力し、intvar*2 の値が 2 の場合は “case2” を出力し、それ以外の場合は “default” を出力します。 default はオプションです。 switch(...) のカッコ内の式は基本型にし、case キーワードに続く式は同じ基本型にします。

C と Java の場合と同様に、終わりまでスキップするには break が必要です。 たとえば、1 つ目の case の後の break を省略すると、この文は intvar*2 が 0 の場合に “case0” と “case2” の両方を出力します。

```
switch(intvar*2) {
  case 0: print('case0');
  case 1+1: print('case2'); break;
  default: print('default'); break;
}
```

第 3 章 データ構造

SPLASH では、特定のデータ操作機能に対応することを目的としたさまざまなデータ構造セットにデータを格納して編成することがあります。

レコード・イベント

SPLASH はレコード・イベントを直接作成します。レコード・イベントとは、“insert” のような関連するオペレーションがあるレコードです。 このマニュアルでは、“レコード・イベント”と同じ意味を持つ用語として“レコード”を使用します。

レコード・イベントの詳細
レコード型の構文では、フィールドの名前と型と、キー構造を指定します。

```
[ integer key1; string key2; | string dataField; ]
```

たとえば、上記の型は、それぞれ整数型と文字列型のキー・フィールド key1 と key2 と、文字列型の非キー・フィールド dataField を持つレコードを表します。キー・フィールドの後に “|” 記号が入ります。

レコード値の構文はレコード型の構文によく似ています。次に、前述の型のレコードを示します。

```
[ key1 = 9; key2 = 'USD'; | string data = 'US Currency'; ]
```

レコード値の構文は非常に柔軟性があります。同じレコードを次のように作成できます。

```
[ key1 = 9; key2 = 'USD' | string data = 'US Currency' ]
[ key1 = 9; key2 = 'USD' | string data = 'US Currency'; ]
```

`field = value` の間のセミコロンを除くすべてのセミコロンが削除されています。

新しいレコード値のオペレーションは insert です。これを変更するには、次のように、`setOpcode` 関数を使用します。

```
setOpcode([ key1 = 9 | string data = 'US Currency' ], update)
```

フィールドが必要な数より多いレコードも使用できます。追加フィールドは強制的に削除されます。 逆に、フィールドが必要な数より少ないレコードも使用できます。不足フィールドは null と見なされます。 たとえば、`var` が次の型の変数の場合、

```
[ integer key1; | string dataField; float otherData]
```

次のように設定できます。

```
var := [key1 = 1; dataField = 'newdata'];
```

レコード値は適切な型に暗黙的にキャストされ、key1 がキー・フィールドになり、otherData フィールドは null に設定されます。

レコードのオペレーションは、次のとおりです。

- **フィールドを取得する** – 構文： `record.field`

  型： 返される値の型はフィールドの型です。

  例： `rec.data1`
- **フィールドを割り当てる** – レコードにフィールドを割り当てます。

  構文： `record.field := value`

  型：value ではレコードのフィールドの型と一致する値を指定します。 この式はレコードを返します。

  例： `rec.data1 := 10`
- **`getOpcode`** – レコードに関連付けられたオペレーションを取得します。オペレーションは整数型です。それぞれの意味は次のとおりです。
  - 1 は “insert” の意味です。
  - 3 は “update” の意味です。

- 5 は “delete” の意味です。
- 7 は “upsert” の意味です (存在しない場合は挿入、それ以外の場合は更新)。
- 13 は “safe delete” の意味です (存在する場合は削除、それ以外の場合は無視)。

構文： `getOpcode(record)`

型： 引数ではイベントを指定します。 この関数は整数を返します。

例： `getOpcode(input)`

- **`setOpcode`** – レコードに関連付けるオペレーションを設定します。有効な opCodeNumber オペレーションについては、前述の `getOpcode` の説明を参照してください。

  構文： `setOpcode(record,opCodenumber)`

  型：1 つ目の引数ではレコードを指定し、2 つ目の引数では整数を指定します。 この関数は変更後のレコードを返します。

  例： `setOpcode(input,insert)`

XML 値

XML 値は XML 要素と属性で構成される値です。XML 要素は他の XML 要素やテキストで構成できます。 XML 値は、直接作成するか、文字列値を解析して作成することができます。 XML 値はレコードには格納できませんが、文字列表現に変換してその形式で格納できます。

XML 値のオペレーション

次のように、`xml` 型の変数を宣言して、その変数に XML 値を代入できます。

```
xml xmlVar;
```

XML 値に使用する変数の宣言のほかに、次のオペレーションも実行できます。

- **`xmlagg`** – 複数の XML 値を単一の値に集約します。これは、集約ストリームまたはイベント・キャッシュでのみ使用できます (以下を参照してください)。

  構文： `xmlagg(xml value)`

  型： 引数では XML 値を指定します。 この関数は XML 値を返します。

  例： `xmlagg(xmlparse(stringCol))`

- **`xmlconcat`** – 複数の XML 値を単一の値に連結します。

  構文： `xmlconcat(xml value ..., xml value)`

型：引数では XML 値を指定します。 この関数は XML 値を返します。

例：`xmlconcat(xmlparse(stringCol), xmlparse('<t/>'))`

- **`xmlelement`** – 属性と XML 式で構成される新しい XML データ要素を作成します。

  構文：`xmlelement(name xmlattributes(string AS name ..., string AS name) , xml value,...,xml value)`

  型： 命名規則は次のとおりです。

  - 名前は、英字、数字、アンダースコア文字のシーケンスか、二重引用符で囲まれた任意の文字のシーケンスです。
  - 名前を二重引用符で囲まない場合は、名前の先頭を英字またはアンダースコア文字にしてください。
  - スペースは二重引用符で囲まない限り、名前には使用できません。
  - 予約語は二重引用符で囲まない限り、名前として使用できません。予約語は大文字と小文字の区別がないので、"AND"、"and"、"AnD" などの名前は使用できません。
  - カラムに "rowid" と "rowtime" の名前を付けることはできません。

  この関数は XML 値を返します。

  例：`xmlelement(top, xmlattributes('data' as attr1), xmlparse('<t/>'))`
- **`xmlparse`** – 文字列を XML 値に変換します。

  構文：`xmlparse(string value)`

  型：引数では文字列値を指定します。 この関数は XML 値を返します。

  例：`xmlparse('<tag/>')`
- **`xmlserialize`** – XML 値を文字列に変換します。

  構文：`xmlserialize(xml value)`

  型：引数では XML 値を指定します。 この関数は文字列を返します。

  例：`xmlserialize(xmlparse('<t/>'))`

ベクトル

ベクトルとは値のシーケンスです。値はすべて同じ型にします。整数のインデックスでシーケンスの要素にアクセスできます。 ベクトルにはサイズがあります。サイズの範囲は最小 0 個から最大 20 億個のエントリです。

セマンティックとオペレーション
ベクトルでは、C から継承されたセマンティックを使用します。 インデックスで要素にアクセスした場合、インデックス 0 はベクトルの最初の位置で、インデックス 1 は 2 番目の位置、というようになります。

グローバルまたはローカルのブロックでベクトルを宣言する場合の構文は、次のとおりです。

```
vector(valueType) variable;
```

たとえば、32 ビットの整数を保持するベクトルを宣言するには、次のように指定します。

```
vector(integer) pos;
```

ベクトルのオペレーションは、次のとおりです。

- **作成する** – 新しい空のベクトルを作成します。

  構文： `new vector(type)`

  型： 宣言された型のベクトルが返されます。

  例： `pos := new vector(integer);`
- **インデックスで値を取得する** – ベクトルから値を取得します。インデックスが 0 より小さいか、ベクトルのサイズ以上であると、null が返されます。

  構文： `vector[index]`

  型： index の型は整数型とします。 返される値の型は、ベクトルに保持されている値の型です。

  例： `pos[10]`
- **値を割り当てる** – ベクトルにセルを割り当てます。

  構文： `vector[index] := value`

  型： index の型は整数型とします。value はベクトルの値の型と一致させます。返される値は更新後のベクトルです。

  例： `pos[5] := 3`
- **`size`** – ベクトル内の要素の数を返します。

  構文： `size(vector)`

  型： 引数ではベクトルを指定します。 返される値は整数型です。

  例： `size(pos)`
- **`push_back`** – ベクトルの終わりに要素を挿入し、変更後のベクトルを返します。

構文： `push_back(vector value)`

型： 2 つ目の引数では、ベクトルの値の型を持つ値を指定します。 返される値はベクトルの型です。

例： `push_back(pos, 3)`

- **`resize`** – ベクトルのサイズを変更します。ベクトルを縮小する場合は要素を削除し、ベクトルを拡大する場合は null 要素を追加します。

  構文： `resize(vector newsize)`

  型： 2 つ目の引数の型は整数型とします。 返される値はベクトルの型です。

  例： `resize(vec1, 2)`

また、“for” ループを使用して、ベクトル内のすべての要素 (最初の null 要素まで) を繰り返し処理することもできます。

辞書とベクトルのデータ構造はグローバルに定義できますが、グローバルに使用できるのは読み取りの場合のみです。辞書またはベクトルのデータ構造に書き込みを実行できるのは 1 つのストリームのみです。また、そのストリームが書き込みを実行している間は、他のストリームはそのデータ構造に対して書き込みも読み取りも実行できません。グローバルな辞書またはベクトルのデータ構造の管理に使用する基本となるオブジェクトはスレッドセーフではありません。ストリームが書き込みを実行する場合は、グローバルな辞書またはベクトルのデータ構造に対する排他的アクセスが必要です。あるストリームが書き込みを実行しているときに他のストリームがこれらのデータ構造にアクセスすることを許可すると、サーバ障害が発生することがあります。

これらのデータ構造の使用は、複数のストリームによって読み取られるが処理中に更新する必要はない比較的静的なデータ (国コードなど) に限定してください。辞書またはベクトルへのデータの書き込みは、ストリームが読み取る前に完了してください。

次の例に示すように、グローバルな辞書またはベクトルを読み取るすべてのオペレーションで、isnull チェックを実行してください。

```
>typeof(streamname) rec := dict[symbol];
if( not (isnull(rec)) {
// use rec
}
```

辞書

辞書とは、キーを値に関連付けるデータ構造です。 C++ と Java ではマップ、AWK では配列、LISP では連想リストと呼ばれ、これらは共通のデータ構造です。

グローバルまたはローカルのブロックで辞書を宣言する場合は、次の構文を使用します。

```
dictionary(keyType, valueType) variable;
```

たとえば、入力ストリーム "input_stream" がある場合は、個々のレコードに対する整数を次のように格納できます。

```
dictionary(typeof(input_stream), integer) counter;
```

キーごとに 1 つの値のみが格納されます。したがって、キーが等しいことが何を意味しているかを理解することが重要です。 単純なデータ型では、この等しさは通常の等しさ (たとえば、整数が等しいことや文字列値が等しいこと) を意味します。 レコード型の場合は、等しさはキーが一致することを意味します (データ・フィールドとオペレーションは無視されます)。

辞書とベクトルのデータ構造はグローバルに定義できますが、グローバルに使用できるのは読み取りの場合のみです。辞書またはベクトルのデータ構造に書き込みを実行できるのは 1 つのストリームのみです。また、そのストリームが書き込みを実行している間は、他のストリームはそのデータ構造に対して書き込みも読み取りも実行できません。グローバルな辞書またはベクトルのデータ構造の管理に使用する基本となるオブジェクトはスレッドセーフではありません。ストリームが書き込みを実行する場合は、グローバルな辞書またはベクトルのデータ構造に対する排他的アクセスが必要です。あるストリームが書き込みを実行しているときに他のストリームがこれらのデータ構造にアクセスすることを許可すると、サーバ障害が発生することがあります。

これらのデータ構造の使用は、複数のストリームによって読み取られるが処理中に更新する必要はない比較的静的なデータ (国コードなど) に限定してください。辞書またはベクトルへのデータの書き込みは、ストリームが読み取る前に完了してください。

次の例に示すように、グローバルな辞書またはベクトルを読み取るすべてのオペレーションで、isnull チェックを実行してください。

```
>typeof(streamname) rec := dict[symbol];
if( not (isnull(rec)) {
// use rec
}
```

辞書のオペレーション

辞書とは、キーを値に関連付けるデータ構造です。 辞書では特定のオペレーションを実行できます。

オペレーション
辞書では次のオペレーションを実行できます。

- **作成する –** 新しい空の辞書を作成します。

  構文： `new dictionary(type, type)`

  型： 宣言された型の辞書が返されます。

  例： `d := new dictionary(integer, string);`
- **キーで値を取得する –** キーで辞書から値を取得します。このようなキーが辞書にない場合は、null が返されます。

  構文： `dictionary[key]`

  型： key の型は辞書のキーの型とします。 この関数は、辞書に保持されている値の型の値を返します。

  例： `counter[input]`
- **キーで値を割り当てる –** 値を辞書内のキーに関連付けます。

  構文： `dictionary[key] := value`

  型：key は辞書のキーの型と一致させ、value は辞書の値の型と一致させます。この関数は更新後の辞書を返します。

  例： `counter[input] := 3`
- **キー／値のペアを削除する –** キーとそのキーに関連付けられた値を辞書から削除します。

  構文： `remove(dictionary, key)`

  型： key は辞書のキーの型と一致させます。 この関数は整数を返します。 キーが存在しなかった場合は 0、それ以外の場合は 1 です。

  例： `remove(counter, input)`
- **辞書をクリアする –** すべてのキー／値のペアを辞書から削除します。

  構文： `clear(dictionary)`

  この関数はクリアされた辞書を返します。

  例： `clear(counter)`

- **空かどうかをテストする** – 辞書が空かどうかをテストします。

  構文： `empty(dictionary)`

  この関数は整数を返します。辞書が空の場合は 1、空でない場合は 0 です。

  例： `empty(counter)`

また、“for” ループを使用して、辞書内のすべてのキー／値のペアを繰り返し処理することもできます。

ストリーム

辞書に似た手段を使用して、入力ストリーム内のレコードにアクセスする方法はありますが、入力ストリーム内のレコードは変更できません。

ストリームのオペレーション

- **キーで値を取得する** – キーでストリームから値を取得します。このようなキーがストリームにない場合は、null が返されます。

  構文： `streamValue[ recordValue ]`

  型： キーの型はストリームのレコード型とします。 このオペレーションは、ストリームのレコード型の値を返します。

  例： `input_stream[ [k = 3; | ] ]`

  注意： 引数に非キー・フィールドがあっても問題ありません。 キー・フィールドを持つレコードが存在すれば、このオペレーションは非キー・フィールドの現在の値を持つレコードを返します。

  キー・フィールドが引数にないか、キー・フィールドが null の場合、このオペレーションは常に null を返します。 ストリーム内のキー・フィールドを null と比較しても意味がありません。null は (null を含め) どのような値とも等しくありません。
- **一致で値を取得する** – 所定のレコードと一致するレコードをストリームから取得します。 キーによる値の取得と異なり、一致するレコードが複数あることがあります。 一致するレコードが複数ある場合は、一致するレコードのいずれか 1 つが返されます。 ストリームにこのような一致がない場合は、null が返されます。

  構文： `streamValue{ recordValue }`

  型： レコードは、ストリームのレコード型と一致させます。 このオペレーションは、ストリームのレコード型の値を返します。

例：`input_stream{ [ | d = 5 ] }`

レコードではキー・フィールドと非キー・フィールドを使用できます。

“for” ループを使用して、ストリーム内のすべてのレコードを繰り返し処理することもできます。

ストリーム反復子

ストリーム反復子は、ストリームに格納されているすべてのレコードを明示的に繰り返し処理する手段です。 通常は、`for` ループ・メカニズムを使用する方が便利で安全ですが、ストリーム反復子では非常に柔軟な対応が可能です。

ストリーム反復子の関数
コードの各ブロックに、ストリームとストリーム反復子の暗黙的な変数があります。 入力ストリームの名前が `Stream1` の場合は、`Stream1_stream` と `Stream1_iterator` の変数があります。

これらの変数は、次の関数と一緒に使用できます。

- **`deleteIterator`** – 反復子に関連付けられたリソースを解放します。

  構文：`deleteIterator(iterator)`

  型：引数では反復子の式を指定します。 この関数は null 値を返します。

  例：`deleteIterator(input_iterator)`

  注意： ストリーム反復子は暗黙的に削除されません。明示的に削除しないと、ストリームに対する後続の更新がすべてブロックされる可能性があります。

- **`getIterator`** – ストリームの反復子を取得します。

  構文：`getIterator(stream)`

  型：引数ではストリームの式を指定します。 この関数は反復子を返します。

  例：`getIterator(input_stream)`

- **`getNext`** – 反復子の次のレコードを返します。レコードがなくなった場合は、null を返します。

  構文：`getNext(iterator)`

  型：1 つ目の引数では反復子の式を指定します。 この関数はレコードを返します。反復子にデータがなくなった場合は、“null” を返します。

  例：`getNext(input_iterator)`

- **resetIterator** – 反復子を開始点にリセットします。

  構文： `resetIterator(iterator)`

  型： 引数では反復子の式を指定します。 この関数は反復子を返します。

  例： `resetIterator(input_iterator)`
- **setRange** – 検索対象のカラムの範囲を設定します。後続の getNext 呼び出しでは、所定の値と一致するカラムを持つレコードのみが返されます。

  構文： `setRange(iterator fieldName... expr...)`

  型：1 つ目の引数では反復子の式を指定します。次の引数ではレコード内のフィールドの名前を指定します。最後の引数では式を指定します。 この関数は反復子を返します。

  例： `setRange(input_iterator,Currency,Rate,'EUR',9.888)`
- **setSearch** – 検索対象のカラムの値を設定します。後続の getNext 呼び出しでは、所定の値と一致するカラムを持つレコードのみが返されます。

  構文： `setSearch( iterator number... expr...)`

  型： 1 つ目の引数では反復子の式を指定します。次の引数ではレコード内のカラム番号を指定します (0 から始まります)。最後の引数では式を指定します。この関数は反復子を返します。

  例： `setSearch(input_iterator,0,2,'EUR',9.888)`

  注意： `setSearch` 関数は、フィールドの特定のレイアウトが必要なため、廃止される予定です。 既存のプロジェクトとの下位互換性のために残されています。 新しいプロジェクトを作成する場合は、代わりに `setRange` 関数を使用してください。

イベント・キャッシュ

イベント・キャッシュには、派生ストリームへの 1 つまたは複数の入力ストリームに対する過去の数多くのイベントが保持されます。 これらのイベントは、レコード内のフィールドの値に基づいてバケットに編成され、ベクトルや辞書が完全には同じデータ構造でない場合によく使用されます。

イベント・キャッシュはローカル・ブロックで定義できます。 次に、単純なイベント・キャッシュ宣言を示します。

```
eventCache(input_stream) e0;
```

このイベント・キャッシュには、入力ストリーム “input_stream” に対するすべてのイベントが保持されます。 入力ストリームのデフォルトのキー構造によってバケット・ポリシーが定義されます。 つまり、このストリームのバケットは入力ストリームのキーに対応します。

この場合の入力ストリームに、キー・フィールド `k` とデータ・フィールド `d` の 2 つのフィールドがあるとします。 次のようなイベントがあったとします。

```
<input_stream ESP_OPS="i" k="1" d="10"/>
<input_stream ESP_OPS="u" k="1" d="11"/>
<input_stream ESP_OPS="i" k="2" d="21"/>
```

これらのイベントが入ると、バケットの数は 2 つになります。 最初の 2 つのイベントのキーが同じであるため、1 つ目のバケットには最初の 2 つのイベントが入ります。2 つ目のバケットには最後のイベントが入ります。

イベント・キャッシュではイベントの集約を実行できます。つまり、集約ストリームで使用できる通常の集約オペレーションをイベント・キャッシュでも同じように使用できます。 集約で選択される“グループ”は、現在のイベントに関連付けられたグループです。

```
<input_stream ESP_OPS="u" k="1" d="12"/>
```

たとえば、このストリームに上記のイベントが入ると、式 `sum(e0.d)` は 10+11+12=33 を返します。 `avg`、`count`、`max`、`min` など、許容される集約関数のすべてを使用できます。

手動による挿入

デフォルトでは、イベント・キャッシュが設定されているストリームに入るイベントはすべてイベント・キャッシュに挿入されます。

このデフォルトの動作は、次のように `auto` オプションを使用して明示的に指定できます。

```
eventCache(instream, auto) e0;
```

また、`manual` でマークされているイベントもイベント・キャッシュに挿入できます。

```
eventCache(instream, manual) e0;
```

これを行うには、関数 `insertCache` を使用します。

バケットの変更

イベント・キャッシュで、イベントはバケットに編成されます。デフォルトでは、バケットは入力ストリームのキーから決定されます。 デフォルトの動作を別のキーに変更するには、ストリームの名前の後に他のフィールドを角カッコで囲んで指定します。

次のように指定すると、バケットは `d0` フィールドと `d1` フィールドのそれぞれの値で編成されます。

```
eventCache(instream[d0,d1]) e0;
```

すべてのイベントを 1 つの大きなバケットに保持するには、次のように記述します。

```
eventCache(instream[]) e0;
```

バケット・サイズの管理

イベント・キャッシュでは、バケットのサイズを管理できます。 これは、メモリの使用を制御する上で重要になることがあります。

バケットのサイズは、最新のイベントに制限したり、秒数または時間で制限したりできます。

```
eventCache(instream, 3 events) e0;
eventCache(instream, 3 seconds) e1;
```

また、`jump` オプションを指定して、サイズを超えるか期限切れになったときにバケットを完全にクリアするかどうかも指定できます。

```
eventCache(instream, 3 seconds, jump);
```

```
```

デフォルトは nojump です。

これらのオプションはすべて一緒に使用できます。たとえば、次は、イベント数が 10 個に達したとき (11 個目のイベントが入ったとき) または 3 秒が経過したときに、バケットをクリアする例です。

```
eventCache(instream, 10 events, 3 seconds, jump);
```

レコードの保持

イベント・キャッシュには、挿入、更新、削除の個々のイベントではなく、coalesce オプションを指定してレコードを保持することもできます。

次に例を示します。

```
eventCache(instream, coalesce) e0;
```

一般に、このオプションは順序付けオプションと一緒に使用されます。

順序付け

通常、バケット内のイベントは着信順に保持されます。 イベントのフィールドを基準とする別の順序付けを指定できます。

たとえば、イベントをバケットに保持する場合にフィールド d を基準として降順で順序付するには、次のように指定します。

```
eventCache(instream, d desc) e0;
```

複数のフィールドを基準として順序付けすることもできます。次は、フィールド d0 を基準として降順で順序付けし、d0 フィールドが等しい場合はフィールド d1 を基準として昇順で順序付けする例です。

```
eventCache(instream, d0 desc, d1 asc) e0;
```

イベント・キャッシュのオペレーション

イベント・キャッシュには、派生ストリームへの 1 つまたは複数の入力ストリームに対する過去の数多くのイベントが保持されます。

サポートされるイベント・キャッシュのオペレーション

- **expireCache** – 現在のバケットから、一定の秒数よりも前のイベントを削除します。

  構文： `expireCache(events, seconds)`

  型： 1 つ目の引数ではイベント・キャッシュ変数を指定します。 2 つ目の引数では整数を指定します。 この関数はイベント・キャッシュを返します。

  例： `expireCache(events, 50)`
- **insertCache** – イベント・キャッシュにレコード値を挿入します。

  構文： `insertCache(events, record)`

  型： 1 つ目の引数ではイベント・キャッシュ変数を指定します。 2 つ目の引数ではレコード型を指定します。 この関数は挿入されたレコードを返します。

  例： `insertCache(events, inputStream)`
- **keyCache** – イベント・キャッシュの現在のバケットを選択します。通常、現在の入力レコードはアクティブなバケットを選択します。 場合によって、現在のアクティブなバケットを変更できます。 たとえば、デバッグの式を評価するときは現在の入力バケットはないので、デフォルトではバケットは設定されません。 その場合にバケットを設定する唯一の方法は、この関数を使用して手動で設定することです。

  構文： `keyCache(events, event)`

  型： 1 つ目の引数ではイベント・キャッシュ変数を指定します。 2 つ目の引数ではレコード型を指定します。 この関数は同じレコードを返します。

  例： `keyCache(ec1, rec)`
- **getCache** – イベント・キャッシュの現在のバケットから、所定のインデックスのローを返します。 このインデックスは 0 から始まります。関数は、引数として整数を受け取り、ローを返します。無効なインデックス・パラメータを指定すると、不正なレコードが生成されます。

  構文： `getCache(cache, index)`

  型： 1 つ目の引数ではイベント・キャッシュ変数を指定します。 2 つ目の引数では、取得するローを指定する整数を指定します。 この関数はキャッシュの指定されたローを返します。

例： `getCache(tradesCache, 3)`

- **`deleteCache`** – イベント・キャッシュの現在のバケットから、所定のインデックスのローを返します。 このインデックスは 0 から始まります。関数は、引数として整数を受け取り、ローを返します。無効なインデックス・パラメータを指定すると、不正なレコードが生成されます。

  構文： `deleteCache(cache, index)`

  型： 1 つ目の引数ではイベント・キャッシュ変数を指定します。 2 つ目の引数では、削除するローを指定する整数を指定します。 この関数は指定されたローを削除します。出力は返されません。

  例： `deleteCache(tradesCache, 0)`
- **`cacheSize`** – イベント・キャッシュの現在のバケットのサイズを返します。

  構文： `cacheSize()`

  型： この関数は引数を取りません。整数が返されます。

  例： `cacheSize()`

索引

F

Flex 演算子
    SPLASH の使用 3
for ループ 9

O

output 文 8

S

SPLASH プログラミングの基本 1
switch 文 11

W

while 文 9

X

XML 値 15

い

イベント・キャッシュ 23
    イベント・バケットの順序付け 26
    オペレーション 27
    バケット・サイズの管理 25
    バケットの変更 25
    レコードの保持 26
    手動による挿入 24

す

ストリーム 21
    反復子の使用 22

て

データ構造 13
    XML 値 15
    イベント・キャッシュ 23
    ストリーム 21
    ストリーム反復子 22
    ベクトル 16
    レコード・イベント 13
    辞書 19

ふ

ブロックの文 7

へ

ベクトル
    宣言 16

れ

レコード・イベント 13