うさぎさんでもわかる並列言語Chapel

2023年1月14日

無線部開発班 JG1VPP

book.nextzlog.dev

目次

第1章　はじめに

Chapel は、**型推論**と**テンプレート**と**区分化大域アドレス空間**を特徴とし、高度に抽象化された**高生産性並列言語**である。NUMA や分散メモリ環境を得意とし、通常の共有メモリ環境と同様の記述性ながら、高度な並列分散処理を実装できる。

```
$ VERSION=1.28.0
$ wget https://github.com/chapel-lang/chapel/releases/download/$VERSION/chapel-$VERSION.tar.gz
$ tar xzf chapel-$VERSION.tar.gz && cd chapel-$VERSION && ./configure && sudo make install
```

以下の hello.chpl を作成する。main 関数は省略できる。コメントの記法は、C 言語 (C99) と同様だが、ネストできる。

```
/*
multiline comments
 or block comments
*/
writeln("Hello, world!"); // 1-line comments
```

chpl でビルドして、hello を生成する。--fast を指定すると、最適化が有効になり、実行時の検査機能が無効化される。他にも、--savec を指定すると、C 言語に変換できる。hello には、--help を始め、便利な機能が自動的に付加される。

```
$ chpl --fast -o hello --savec savec hello.chpl
$ ./hello
Hello, world!
$ ./hello --help
```

Chapel では、module 宣言で**名前空間**を定義する。関数の外に記述された処理は、名前空間の初期化と同時に実行される。最も外側の処理は、拡張子.chpl を除外した名前の、名前空間を構成する。これが、main 関数を省略できた理由である。

```
module Foo {
  writeln("initilize Foo");
  proc main() {
    writeln("This is Foo");
  }
}
module Bar {
  writeln("initilize Bar");
  proc main() {
    writeln("This is Bar");
  }
}
import baz.Foo;
import baz.Bar;
proc main() {
  Foo.main();
  Bar.main();
}
```

複数の名前空間を定義した場合は、どの名前空間に実装された main 関数を起動するか、ビルド時に指定する必要がある。

```
$ chpl --main-module baz baz.chpl
$ ./baz
initilize Foo
initilize Bar
This is Foo
This is Bar
```

第2章　式

C 言語と同様に、演算子には、左右の**結合法則**と**優先順位**がある。優先順に掲載する。演算の順序は、括弧で調整できる。

関数適用	`. () []`	左結合
初期化	`new`	右結合
所有権	`owned shared borrowed unmanaged`	右結合
ヌル許容	`? !`	左結合
型変換	`:`	左結合
累乗	`**`	右結合
集約	`reduce scan dmapped`	左結合
否定	`! ~`	右結合
乗除	`* / %`	左結合
符号	`+ -`	右結合
シフト	`<< >>`	左結合
論理積	`&`	左結合
排他的論理和	`^`	左結合
論理和	`|`	左結合
加減	`+ -`	左結合
レンジ	`.. ..<`	左結合
順序比較	`<= => < >`	左結合
等値比較	`== !=`	左結合
短絡論理積	`&&`	左結合
短絡論理和	`||`	左結合
刻み幅	`by # align`	左結合
ループ変数	`in`	左結合
制御構造式	`if for forall sync single atomic`	左結合
区切り	`,`	左結合

2.1　型変換

Chapel は、**静的型付け言語**である。また、型変換を明示的に行う場合は、: 演算子を使う。以下に、型変換の例を示す。

```
writeln(1 + 2: real); // 3.0
writeln(int: string); // int(64)
```

2.2　集約演算

`reduce` 演算子は、右辺の値を集計する。必要に応じて、並列処理が行われる。`scan` 演算子は、集計の過程を順番に返す。集計の内容は、+や*や `min` や `max` を指定できる。`minloc` や `maxloc` の場合は、最小値または最大値の位置も取得できる。

```
writeln(+ scan(1..10)); // 1 3 6 10 15 21 28 36 45 55
writeln("value, index: ", minloc reduce zip([3, 2, 1], 0..2)); // value, index: (1, 2)
writeln("value, index: ", maxloc reduce zip([4, 5, 6], 0..2)); // value, index: (6, 2)
```

第3章　変数

var で宣言された識別子は、**変数**となる。宣言と同時に型と値を指定できる。自明な場合は、型または値を省略できる。変数の値は、**代入演算子**で変更できる。その場合の変数を**左辺値**と呼ぶ。また、**スワップ演算子**で両辺の値を交換できる。

```
var foo: int = 12;
var bar: int;
foo = 889464;
var a = "golden axe.";
var b = "silver axe.";
a <=> b;
writeln(a, b, foo);
```

3.1　定数

const で宣言された識別子は、**定数**となる。定数の値は変更できず、初期値で固定される。初期値は実行時に計算される。

```
const foo: int = 12;
const bar = 12;
```

param で宣言された識別子は、**静的定数**となる。定数と同様に初期値で固定され、初期値はコンパイル時に計算される。

```
param foo: int = 12;
param bar = 12;
type num = int;
```

type で宣言された識別子は、**型定数**となる。なお、静的定数には、以下に示す**基本型**または**列挙型**の値のみ設定できる。

```
param a: bool = true;
param b: uint = 1919;
param c: real = 8.10;
param d: imag = 364i;
param e: string = 'ABC' + "DEF";
param f: Gibier = Gibier.Rabbit;
enum Gibier {Deer, Boar, Rabbit};
```

3.2　設定

config を前置した変数や定数の値は、起動時に変更できる。同様に、静的定数や型定数も、コンパイル時に変更できる。

```
config const bar = 121;
config param foo = 121;
config type num = uint;
writeln(foo, bar, num: string);
```

例えば、定数 bar は、起動時に--bar で設定できる。静的定数 foo と型定数 num は、コンパイル時に--set で設定できる。

```
$ chpl config.chpl --set foo=12321 --set num=real
$ ./config --bar=12321
1232112321real(64)
```

第4章　構文

Chapel の構文には、Fortran の影響が見られる。**文**の区切りには、; を記す。基本的に、先頭の文から順番に実行される。複数の文を括弧で閉じた**複文**は、静的な**スコープ**を形成し、内側で宣言された変数や関数は、外側に対して秘匿される。

```
{
  var foo = 12;
  writeln(foo);
}
var foo = 13;
writeln(foo);
```

4.1　条件分岐

if 文は、**条件分岐**を行う。条件式は bool 型である。then 節が複文の場合は、then を省略できる。else 節も省略できる。

```
const age = 18;
if age < 18 then {
  writeln("Adults Only");
} else {
  writeln("Yeah, Right");
}
```

select 文は、**多分岐**を行う。条件式が合致した when 節か otherwise 節が実行される。両者とも固有のスコープを持つ。

```
select "cat" {
  when "cat" do writeln("meow");
  when "dog" do writeln("bowwow");
  otherwise writeln("gobblegobble");
}
```

4.2　反復処理

while do 文は、do 節を繰り返す。まず、条件式を評価して、true の場合は do 節を実行し、再び条件式の評価に戻る。do while 文も、do 節を繰り返す。まず、do 節を実行して、条件式を評価する。true の場合は、再び do 節を実行する。

```
var i = 1;
while i < 1024 do i *= 2;
do i >>= 1; while i >= 1;
writeln(i); // 0
```

for 文は、**イテレータ**が値を返す限り do 節を繰り返す。while do 文や for 文の do 節が複文の場合は、do を省略できる。

```
for i in 1..100 do writeln(i);
```

反復処理を離脱する場合は、C 言語と同様に、break 文や continue 文を使う。ラベルを指定すれば、**大域脱出**もできる。

```
while true do break;
for 1..100 do continue;
label outside for i in 1..10 do for j in 1..10 do break outside;
```

第5章　関数

Chapel の関数には、**手続き**と**イテレータ**と演算子の 3 種類が存在し、予約語の `proc` と `iter` と `operator` で定義できる。記事では、単に手続きを指して関数と呼ぶ。関数は、*first-class* で、**テンプレート**の機能を持ち、例外処理も可能である。

5.1　定義

以下の関数 `foo` は、`int` 型の引数 `x` と `y` を取り、`return` 文で `int` 型の値を返す。関数の呼び方は、C 言語と同様である。

```
proc foo(x: int, y: int): int {
  return x + y;
}
writeln(foo(1, 2));
```

引数や返り値の型は、省略できる。この場合の関数は、実質的に**テンプレート関数**であり、型は、実引数から推論される。

```
proc foo(x, y) return x + y;
writeln(foo(1, 2i)); // 1.0 + 2.0i
writeln(foo(2i, 3)); // 3.0 + 2.0i
```

引数が 0 個の場合は、引数の括弧を省略できる。また、内容が `return` 文だけの関数は、処理を囲む括弧 `{}` を省略できる。

```
proc foo return 100110;
writeln(foo);
```

関数に引数を渡す際に、引数の名前を指定できる。また、引数を省略した場合に渡されるデフォルトの値を設定できる。

```
proc foo(x: int = 0, y: int = 0): int return x + y;
writeln(foo(x = 1, y = 2)); // 3
writeln(foo(y = 2, x = 1)); // 3
writeln(foo(1)); // 1
```

関数は、可変長の引数を宣言できる。その実体は**タプル**である。また、タプルを渡す場合は、展開の演算子 `...` を使う。

```
proc sum(x: int ...): int return + reduce(x);
writeln(sum(1, 2, 3) + sum((...(100, 200)))); // 306
```

関数は、他の関数の内側にも定義できる。関数の外側から見ると、内側の関数は秘匿される。また、**高階関数**にもできる。

```
proc factorial(num: int): int {
  proc tc(n, accum: int): int {
    if n == 0 then return accum;
    return tc(n - 1, n * accum);
  }
  return tc(num, 1);
}
writeln(factorial(10)); // 3628800
```

例えば、関数を関数の引数や変数に代入できる。また、関数の名前が必要なければ、`lambda` で無名の関数を定義できる。

```
proc call(f, x: int, y: int): int return f(x, y);
const add = lambda(x: int, y: int) return x + y;;
writeln(call(add: func(int, int, int), 36, 514)); // 550
```

5.2 修飾子

inline で宣言された関数は、**インライン展開**される。export で宣言された関数は、**ライブラリ関数**として公開される。

```
inline proc foo(x: int): int return 2 * x;
export proc bar(x: int): int return 2 * x;
writeln(foo(100)); // 200
writeln(bar(300)); // 600
```

共有ライブラリで実装された関数を使う場合は、extern で関数を宣言する。以下に、CPU の番号を取得する例を示す。

```
require "sched.h";
extern proc sched_getcpu(): int;
writeln("CPU:", sched_getcpu());
```

引数や返り値にも、修飾子を指定できる。param や type で宣言された引数は、定数や型となる。返り値も同様にできる。

```
proc tuplet(param dim: int, type eltType) type return dim * eltType;
const septet: tuplet(7, int) = (114, 514, 1919, 810, 364, 889, 464);
```

inout や out で宣言された引数の値は、関数から戻る際に書き戻される。ただし、out の場合は、実引数が無視される。

```
proc intents(inout x: int, in y: int, out z: int, ref v: int): void {
  x += y;
  z += y;
  v += y;
}
var a: int = 1;
var b: int = 2;
var c: int = 3;
var d: int = 4;
intents(a, b, c, d);
writeln(a, b, c, d); // 3226
```

ref で宣言された引数は、**参照渡し**になる。また、返り値が ref の関数は、左辺値を返すので、代入の構文を定義できる。

```
var tuple = (0, 0, 0, 0, 0, 0, 0, 0, 0);
proc A(i: int) ref: int return tuple(i);
coforall i in tuple.indices do A(i) = i;
writeln(tuple);
```

5.3 条件分岐

Chapel の関数は、実質的に**テンプレート関数**である。関数が**インスタンス**になる条件は、where 節で詳細に指定できる。where 節は、コンパイル時に評価される。例えば、以下の whichType 関数は、引数 x の型 xt に応じて、内容が変化する。

```
proc whichType(x: ?xt): string where xt == real return "x is real";
proc whichType(x: ?xt): string where xt == imag return "x is imag";
proc whichType(x: ?xt): string return "x is neither real nor imag";
writeln(whichType(114.514));
writeln(whichType(364364i));
writeln(whichType(1919810));
```

where 節の条件分岐は、任意の定数式を扱えるので、**コンパイル時計算**に利用できる。以下は、階乗を計算する例である。

```
proc fact(param n: int) param: int where n >= 1 return n * fact(n-1);
proc fact(param n: int) param: int where n == 0 return 1;
if fact(8) != 5040 then compilerError("fact(7) == 5040");
```

5.4 型引数

任意の値を受け取る引数に、?を前置した**型引数**を宣言すると、型を取得できる。以下に、配列の型を取得する例を示す。ただし、型引数を宣言せずに、.type や.domain や.eltType などの**クエリ式**を活用し、型の詳細を取得する方法もある。

```
proc foo(x: [?dom] ?el) return (dom, el: string);
proc bar(x) return (x.domain, x.eltType: string);
writeln(foo([1, 2, 3, 4, 5, 6, 7, 8])); // ({0..7}, int(64))
writeln(bar([1, 2, 3, 4, 5, 6, 7, 8])); // ({0..7}, int(64))
writeln(foo(["one" => 1, "two" => 2])); // ({one, two}, int(64))
writeln(bar(["one" => 1, "two" => 2])); // ({one, two}, int(64))
```

5.5 例外処理

throw 文は、異常の発生を通知する。この異常を**例外**と呼ぶ。catch 文で捕捉されるまで、関数が順繰りに巻き戻される。例外が発生し得る関数は、throws 宣言が必要である。また、defer 文で予約した処理は、例外が発生しても実行される。

```
proc foo(message: string) throws {
  defer writeln("See you");
  throw new Error(message);
}
```

catch 文は、try 文の内側で例外が発生した場合には、その例外を捕捉し、回復処理を行う役割がある。以下に例を示す。

```
try {
  foo("Hello,");
  foo("world!");
} catch e {
  writeln(e);
}
```

5.6 演算子

operator で宣言された関数は、演算子の機能を再定義する。ただし、第 2 章に掲載した演算子に限る。以下に例を示す。

```
operator *(text: string, num: int) return + reduce (for 1..num do text);
writeln("Lorem ipsum dolor sit amet, consectetur adipiscing elit" * 10);
```

5.7 イテレータ

iter で宣言された関数は、**イテレータ**となる。yield 文は、処理を断ち、指定された値を返し、残りの処理を再開する。these を実装した構造体も、イテレータとして機能する。for 文に渡すと、暗黙的に these が呼ばれる。以下に例を示す。

```
iter iterator(): string {
  yield "EMURATED";
  yield "EMURATED";
  yield "EMURATED";
}
iter int.these() const ref: int {
  for i in 1..this do yield i;
}
var repetition: int = 10;
for i in iterator() do writeln(i);
for i in repetition do writeln(i);
```

第6章　構造体

Chapel では、構造体を定義して、構造体に変数や関数を定義できる。変数を**フィールド**と呼び、関数を**メソッド**と呼ぶ。構造体は、**クラス**と**レコード**と**共用体**に分類できる。クラスは**参照型**の性質を、レコードと共用体は**値型**の性質を持つ。

6.1　定義

以下に、クラスを定義して、new 演算子で**インスタンス**を生成する例を示す。参照型なので、変数には参照が格納される。

```
class Num {
  var r: real;
  var i: imag;
}
var num: Num = new Num();
num.r = 816.07;
num.i = 14.22i;
writeln(num.r);
```

以下に、レコードを定義する例を示す。クラスと似た機能だが、値型なので、インスタンスの複製が変数に格納される。

```
record Num {
  var r: real;
  var i: imag;
}
var num: Num;
num.r = 816.07;
num.i = 14.22i;
writeln(num.r);
```

共用体は、同じメモリ領域を複数のフィールドで共有する。最後に値を格納したフィールドのみ、意味のある値を持つ。

```
union Num {
  var r: real;
  var i: imag;
}
var num: Num;
num.r = 816.07;
num.i = 14.22i;
writeln(num.i);
```

6.2　総称型

総称型は、型引数や静的定数を引数に取る多相型である。型引数が異なると、異なる型として扱われる。以下に例を示す。

```
record Stack {
  type eltType;
  param size: int;
}
var a: Stack(eltType = uint, size = 12);
var b: Stack(eltType = real, size = 24);
writeln("a is ", a.type: string); // a is Stack(uint(64), 12)
writeln("b is ", b.type: string); // b is Stack(real(64), 24)
```

6.3 メソッド

構造体には、メソッドを定義できる。また、メソッドでインスタンスを参照する場合は、this を使う。以下に例を示す。

```
record User {
  var name: string;
  proc set(name) {
    this.name = name;
  }
}
var user: User;
user.set("Alicia");
writeln(user.name);
```

なお、値型の性質を持つレコードでも、this はインスタンスへの参照であり、意図通りにフィールドの値を変更できる。
既存の構造体にも、メソッドを追加できる。また、フィールドと同名で、参照と代入を隠蔽する、**アクセサ**を定義できる。

```
record User {
  var name: string;
}
proc User.name ref {
  writeln(".name");
  return this.name;
}
var user: User;
user.name = "Jane";
writeln(user.name);
```

this メソッドが定義された構造体は、関数と同様に振る舞う。引数を与えると、暗黙的に this メソッドが実行される。

```
class Add {
  proc this(a: int, b: int): int return a + b;
}
const add = new Add();
writeln(add(123, 45)); // 168
```

init メソッドが定義された構造体は、引数を与えてインスタンスを生成すると、暗黙的に init メソッドが実行される。
init メソッドは、フィールドを初期化できる。定数の場合は、init メソッドで初期化する必要がある。以下に例を示す。

```
class Add {
  const x: real;
  const y: real;
  proc init(x, y) {
    this.x = x;
    this.y = y;
  }
}
const add = new Add(x = 1, y = 2);
writeln("x, y: ", (add.x, add.y));
```

init メソッドは、省略しても自動的に定義される。また、deinit メソッドは、インスタンスを解放する際に実行される。

```
class Sub {
  const x: real = 0;
  const y: real = 0;
  proc deinit() {
    writeln("deleted");
  }
}
const sub = new Sub(x = 1, y = 2);
writeln("x, y: ", (sub.x, sub.y));
```

6.4 所有権

Chapel のクラスには、**所有権**の概念がある。所有権を持つ変数の参照が消滅したインスタンスは、自動的に解放される。所有権は、変数宣言で指定する。無修飾または owned で修飾した場合は、その変数がインスタンスの所有権を独占する。

```
class Hello {
  proc deinit() {
    writeln("See you");
  }
}
var hello: owned Hello = new owned Hello();
var hallo: borrowed Hello = hello.borrow();
```

borrowed で修飾された変数は、所有権を取得せず、他の owned や shared の変数が所有するインスタンスを参照できる。shared で修飾した場合は、変数の間でインスタンスが共有される。全ての参照が消滅するとインスタンスが解放される。

```
class Hello {
  proc deinit() {
    writeln("See you");
  }
}
var hello = new shared Hello();
var hallo = new unmanaged Hello();
delete hallo;
```

unmanaged で修飾した場合は、所有権による管理を受けず、delete 文で、明示的にインスタンスを解放する必要がある。変数に代入すると、所有権を移譲できる。owned の場合は、所有権が左辺値に移り、shared の場合は、単に共有される。

6.5 継承

クラスを定義する際に、**親クラス**を指定すると、その親クラスのフィールドやメソッドを継承できる。以下に例を示す。

```
class Foo {
  proc foo() return "foo!";
}
class Bar: Foo {
  override proc foo() return super.foo() + "bar!";
}
const foo = new Foo();
const bar = new Bar();
writeln(foo.foo()); // foo!
writeln(bar.foo()); // foo!bar!
```

継承したメソッドの内容を再定義する場合は、override 宣言を行う。再定義する前のメソッドは、super で参照できる。

6.6 構造的部分型

以下に示す voice 関数は、quack メソッドが定義された、任意の型の値を引数に取る。これを**ダックタイピング**と呼ぶ。

```
class Duck {
  proc quack() return "quack!";
}
class Kamo {
  proc quack() return "quack!";
}
proc voice(x) return x.quack();
writeln(voice(new Duck())); // quack!
writeln(voice(new Kamo())); // quack!
```

第7章　配列

Chapel の配列には、**矩形配列**と**連想配列**の 2 種類が存在する。また、類似の概念に、**タプル**と**レンジ**と**領域**が存在する。

7.1　タプル

タプルは、複数の要素をカンマ区切りで並べた値である。値型で、要素の型が揃う必要もなく、配列よりも軽量である。要素の型を揃えた場合は、要素の数と型の乗算により、タプル型を表現できる。また、these イテレータが利用できる。

```
const nums: 9 * int = (60, 45, 53, 163, 90, 53, 165, 75, 60);
const boys: (string, string, string) = ("Tom", "Ken", "Bob");
writeln(boys(0), boys(1), boys(2), for name in boys do name); // TomKenBobTom Ken Bob
```

タプルは、複数の変数を同時に宣言する場合や、値を同時に書き込む場合に利用できる。この機能を**アンパック**と呼ぶ。

```
var (a, b): (string, int) = ("student", 24);
writeln(a, b);
```

タプルのアンパック機能は、関数の宣言でも利用できる。タプルで宣言された引数には、タプルの値を渡す必要がある。

```
proc foo(x: int, (y, z): (int, int)): int return x * (y + z);
writeln(foo(2, (3, 4))); // 14
```

7.2　レンジ

レンジは、整数の有限区間や半無限区間や無限区間を表す値である。値型で、同様の機能を持つ領域よりも軽量である。

```
const from100To200: range(int, boundedType = BoundedRangeType.bounded) = 100..200;
const from100ToInf: range(int, boundedType = BoundedRangeType.boundedLow) = 100..;
const fromInfTo200: range(int, boundedType = BoundedRangeType.boundedHigh) = ..20;
```

by で刻み幅を指定できる。その場合は、align で指定された値を必ず含む。また、#でレンジの要素の個数を指定できる。

```
writeln(10..30 by -7);              // 30 23 16
writeln(10..30 by -7 align 13);     // 27 20 13
writeln(10..30 by -7 align 13 # 2); // 27 20
```

レンジに含まれる要素の個数は.size で、区間の下限は.low で、区間の上限は.high で、刻み幅は.stride で得られる。

```
writeln((100..200).size);      // 101
writeln((100..200).low);       // 100
writeln((100..200).high);      // 200
writeln((100..200).stride);    //   1
writeln((100..200).alignment); // 100
```

レンジでは、these イテレータが利用できる。for 文に限らず、並列化された forall 文や coforall 文でも利用できる。

```
for i in 1..100 do writeln(i);
forall i in 1..100 do writeln(i);
coforall i in 1..100 do writeln(i);
```

7.3 領域

領域は、配列の定義域や値の集合を表す。レンジのタプルと等価な**矩形領域**と、要素を格納した**連想領域**の 2 種類がある。領域の要素の型は.idxType で取得できる。特に、矩形領域は、.rank で階数を、.dims でレンジのタプルを取得できる。

```
const rectangular: domain = {0..10, -1..2};
const associative: domain = {"foo", "bar"};
writeln(rectangular.rank);
writeln(rectangular.dims());
writeln(rectangular.idxType: string); // int(64)
writeln(associative.idxType: string); // string
```

矩形領域も連想領域も、these イテレータが利用できる。特に、矩形領域は、多重ループを表す構文としても利用できる。

```
for xyz in {1..10, 1..10, 1..10} do writeln(xyz);
for boy in {"Tom", "Ken", "Bob"} do writeln(boy);
```

7.4 配列

配列は、定義域から値域への写像を表す。矩形領域に対応する**矩形配列**と、連想領域に対応する**連想配列**の 2 種類がある。

```
const rectangular = [1, 2, 3, 4, 5, 6, 7, 8];
const associative = [1 => "one", 2 => "two"];
writeln(rectangular, rectangular.domain); // 1 2 3 4 5 6 7 8{0..7}
writeln(associative, associative.domain); // one two{1, 2}
```

this メソッドが利用でき、要素の位置を引数に渡せば、その要素を参照できる。領域を渡せば、部分配列も参照できる。

```
var A: [{1..10, 1..10}] real;
var B: [{'foo', 'bar'}] real;
A[1, 2] = 1.2;
A(3, 4) = 3.4;
writeln(A(1..2, 1..3));
```

these イテレータも利用できる。左辺値を返すので、for 文でループ変数に値を書き込むと、その値が配列に反映される。

```
var boys = ['Tom', 'Ken', 'Bob'];
for boy in boys do boy += '-san';
writeln(boys); // Tom-san Ken-san Bob-san
```

祖な矩形領域を定義域に指定すると、粗な配列を宣言できる。これは、粗行列の実装として活用できる。以下に例を示す。

```
var D: sparse subdomain({1..16, 1..64});
var A: [D] real;
D += (8, 10);
D += (3, 64);
A[8, 10] = 114.514;
```

配列は値型であり、配列を他の配列に代入すると、値が複製される。ただし、関数に配列を渡す場合は、参照渡しになる。

```
proc update(arr: [] int) {
  arr = [2, 3, 4];
}
var A = [1, 2, 3];
var B = A;
update(A);
writeln(A); // 2 3 4
writeln(B); // 1 2 3
```

第8章　並列処理

Chapel は、qthreads の**軽量スレッド**を採用し、細粒度の並列処理が得意で、**タスク**の分岐と待機を高効率に実行できる。

8.1　タスク

まず、基本の構文を解説する。begin 文でタスクを非同期に実行し、sync 文でそのタスクを待機する。以下に例を示す。

```
sync {
  begin writeln("1st parallel task");
  begin writeln("2nd parallel task");
  begin writeln("3rd parallel task");
}
writeln("task 1, 2, 3 are finished");
```

begin 文は、入れ子にできる。糖衣構文として、cobegin 文でも同じ処理を記述できる。通常は cobegin 文で事足りる。

```
cobegin {
  writeln("1st parallel task");
  writeln("2nd parallel task");
  writeln("3rd parallel task");
}
writeln("task 1, 2, 3 are finished");
```

外側で宣言された変数に、begin 文や cobegin 文で値を書き戻す場合は、with 節の宣言が必要である。以下に例を示す。

```
var a: string;
var b: string;
sync {
  begin with(ref a) a = "1st parallel task";
  begin with(ref b) b = "2nd parallel task";
}
writeln(a);
writeln(b);
```

sync 文や cobegin 文による待機は、僅かな負荷を生じるので、効率を求める場合は、敢えて begin 文を使う場合もある。

```
inline proc string.shambles: void {
  proc traverse(a: int, b: int) {
    if b > a {
      const mid = (a + b) / 2;
      begin traverse(a, 0 + mid);
      begin traverse(1 + mid, b);
    } else writeln(this(a));
  }
  sync traverse(0, this.size - 1);
}
```

serial 文は、条件式が true の場合に、並列処理を逐次処理に切り替える。並列処理の最適化やデバッグに利用できる。

```
serial true {
  begin writeln("1st serial task");
  begin writeln("2nd serial task");
}
```

8.2 反復処理

forall 文と coforall 文は、並列化された for 文である。OpenMP の omp parallel for に相当する。以下に例を示す。

```
forall i in 1..100 do writeln(i);
coforall i in 1..100 do writeln(i);
```

coforall 文は、以下の糖衣構文である。反復回数と同じ個数のタスクを生成する。**タスク並列**を意識した機能と言える。

```
sync for i in 1..100 do begin writeln(i);
```

forall 文は、タスクを生成する *leader* と、末端の逐次処理を担当する *follower* の、2 個のイテレータで並列処理を行う。

```
iter foo(rng): int {
  for i in rng do yield i;
}
```

foo イテレータを多重に定義して、以下の派生型を実装する。これが *leader* で、タスクとデータを再帰的に分岐させる。

```
iter foo(param tag, rng): range where tag == iterKind.leader {
  if rng.size > 16 {
    const mid = (rng.high + rng.low) / 2;
    cobegin {
      for i in foo(tag, rng(..mid+0)) do yield i;
      for i in foo(tag, rng(mid+1..)) do yield i;
    }
  } else yield rng;
}
```

また、以下の派生型が *follower* で、並列処理の末端で、細粒度の逐次処理を担当する。データは followThis に渡される。

```
iter foo(param tag, rng, followThis): int where tag == iterKind.follower {
  for i in followThis do yield i;
}
```

処理の内容に応じて、最適なタスクとデータの分配方法を選べる点で、forall 文は**データ並列**を意識した機能と言える。

```
forall i in foo(1..100) do writeln(i);
```

8.3 ロケール

Chapel では、分散メモリ環境の構成単位を**ロケール**と呼ぶ。典型的には、1 個の共有メモリ環境がロケールに相当する。利用可能なロケールは Locales で得られる。また、GASNet を使う場合は、環境変数 CHPL_COMM を設定する必要がある。

```
coforall l in Locales do on l {
  writeln(here == l);
  writeln(here.name);
  writeln(here.numPUs());
}
```

特定のロケールを指定して、タスクを実行するには、on 文を使う。また、here を通じて、現在のロケールを参照できる。他のロケールに存在するデータを参照すると、**レイテンシ**が発生する。on 文は、参照の局所性を高める目的でも使える。

```
import BigInteger;
var x: BigInteger.bigint;
on Locales(0) do x = new BigInteger.bigint(12);
on x.locale do writeln(x, " is on ", x.locale);
```

8.4 分散配列

Chapel の配列のメモリ領域は、第 8.3 節のロケールに分散して配置できる。分配の方法は、dmapped 演算子で指定する。例えば、Block を選ぶと、配列は矩形の塊で分配される。また、localSubdomain で、そのロケールの領域を取得できる。

```
use BlockDist;
var A: [{1..10,1..10} dmapped Block(boundingBox={1..10,1..10})] real;
for l in Locales do on l do for (i, j) in A.localSubdomain() do writeln(A(i, j).locale);
```

Cyclic を選ぶと、周期的に分配される。dmapped で確保した配列は、必要に応じて on 文で参照の局所性を確保できる。

```
use CyclicDist;
var B: [{1..10,1..10} dmapped Cyclic(startIdx=(1,1))] real;
for l in Locales do on l do for (i, j) in B.localSubdomain() do writeln(B(i, j).locale);
```

8.5 アトミック変数

並列処理で、複数のタスクが共通の変数を読み書きする場合は、**アトミック演算**で、タスク間の競合を防ぐ必要がある。競合の例を以下に示す。変数 sum の値を取得し、新たな値を書き戻す間に、sum の値が変化すれば、誤った結果になる。

```
config const N = 80;
var sum: int;
do {
  coforall n in 1..N with(ref sum) do sum -= n * n;
  coforall n in 1..N with(ref sum) do sum += n * n;
} while sum == 0;
writeln(sum);
```

以下に、適切な実装を示す。add や sub の代わりに、fetchAdd や fetchSub を使えば、演算する直前の値も取得できる。

```
config const N = 80;
var sum: atomic int;
do {
  coforall n in 1..N do sum.sub(n * n);
  coforall n in 1..N do sum.add(n * n);
} while sum.read() == 0; // infinite loop
writeln(sum);
```

8.6 ロック付き変数

sync 型の変数を参照するには、readFE を使う。ただし、変数の状態が *empty* の場合は、*full* に遷移するまで待機となる。writeEF で値を書き込むと、状態が *full* に遷移する。これは**排他制御**の機能であり、タスク間の競合を防ぐ効果がある。

```
config const N = 80;
var sum: sync int = 0;
do {
  coforall n in 1..N do {
    const v = sum.readFE();
    sum.writeEF(v + n * n);
  }
  coforall n in 1..N do {
    const v = sum.readFE();
    sum.writeEF(v - n * n);
  }
} while sum.readFF() == 0; // infinite loop
writeln(sum.readFF());
```