解 説

# マルチコア時代のプログラミング言語「X10」

**河内谷 清久仁**
日本アイ・ビー・エム（株）東京基礎研究所

**最新のプロセッサでは複数の実行コアを備えた「マルチコア」が一般的になっている．しかし，多数のマルチコアCPU や GPGPU などが相互接続された並列分散環境を活用するためのプログラミングは容易ではない．このような並列分散プログラミングの生産性を向上させることを目標とした，新しいプログラミング言語が X10（エックステン）である．**

**X10 はいわゆる PGAS（Partitioned Global Address Space）を採用している言語族の一員であり，複数の「プレース」にまたがったデータ構造を扱える．さらに，軽量な並列タスクを生成するための「`async`」や分散処理のための「`at`」，同期のための「`atomic`」や「`clocked`」などの構文を提供している．本稿では，X10 の設計思想やプログラミングモデルに加え，これらの特徴的な構文についても使用例を交えつつ解説する．**

## マルチコア時代のプログラミング

コンピュータの性能向上は長らく，クロックの高速化やパイプラインの改善によって達成されてきた．しかし，シングルプロセッサの性能向上は頭打ちとなり，古きよき時代は終わりを告げた．最新のプロセッサは，複数の実行コアを備えたマルチコア構成をとるものが普通である．また，グラフィクス処理用チップの多数の演算ユニットを汎用計算処理に用いる GPGPU[☆1] も身近になってきている．近い将来には，これらを複数搭載したマルチプロセッサマシンが高速ネットワークで相互接続された大規模並列分散環境が一般的なものになっているだろう．

しかし，そのような環境では実行コアの総数は数千〜数十万個にもなるため，それらを活用できるソフトウェアを開発することは容易ではない．そのためには，ハードウェア環境の変革に合わせ，プログラミング環境も「マルチコア時代」へと向けた変革を遂げることが必要となる．具体的な要件としては，以下のものが挙げられる．

1. 大量の非同期タスクを軽量に生成・消滅できること．また，デッドロック等に悩まされずにタスク間の同期が行えること．
2. 共有メモリ環境と分散メモリ環境を単一のプログラムで扱えること．また，異なるアーキテクチャの実行環境をサポートできること．
3. 複数マシン間の複雑な通信処理を記述しなくてよいこと．しかし，並列処理や分散処理をある程度明示的に制御できること．
4. 開発をサポートするツール群が用意されていること．

これらの要件を満たすべく IBM Research が開発中

☆1 General-Purpose computing on Graphics Processing Units.

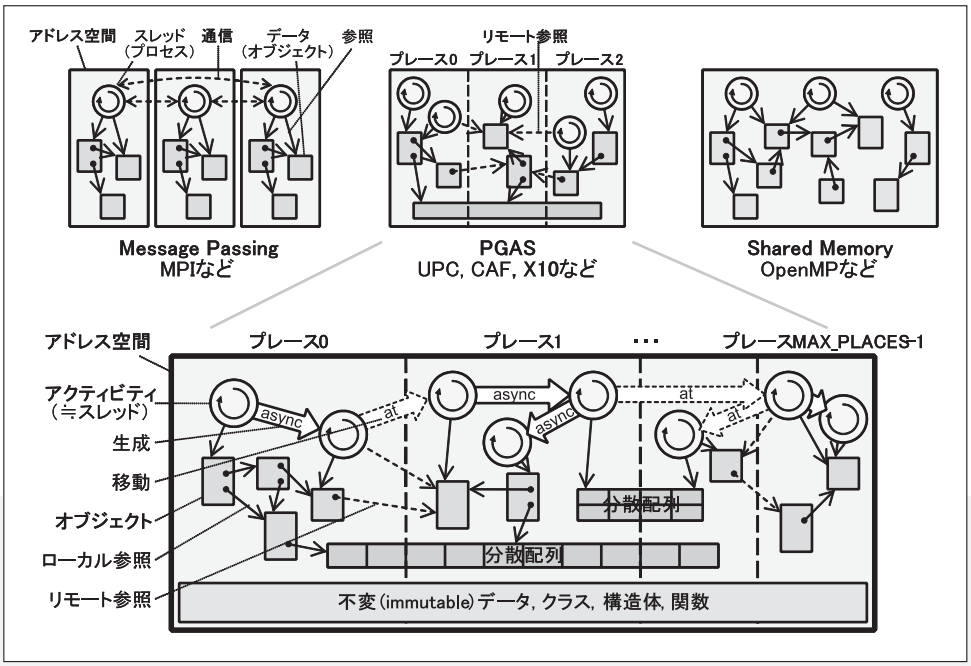


図 -1　X10 の実行モデル

の新しいプログラミング言語が，X10（エックステン）である．本稿では，X10 の設計思想やプログラミングモデル，データ型や文法について，X10 2.1.2 の仕様をベースに解説を行う．掲載したプログラムはすべて単体でコンパイル・実行が可能な形となっているので，X10 アプリケーション開発時のサンプルとして活用していただければ幸いである．

## X10 の概要

X10 は，IBM Research が開発している新しい並列分散プログラミング言語で，米 DARPA[☆2] の HPCS[☆3] プログラムに基づく IBM の PERCS[☆4] プロジェクトの一部である．PERCS プロジェクトは，先進的なチップ技術とアーキテクチャ，OS，プログラミング言語などを統合し，ハードウェアとソフトウェアを総合的にデザインすることで，並列アプリケーションの生産性を向上させることを目標としている[☆5]．そのために，X10 は新しいプログラミングモデルと開発ツール群を提供する．

X10 の開発は 2004 年に開始され，オープンソースプロジェクトとして行われている．ライセンス形態は Eclipse Public License である．成果は http://x10-lang.org/[1] からアクセス可能で，ここから最新の実行環境とそのソースコードに加え，言語仕様[2] やチュートリアル等も取得できる．メーリングリストなどによるサポートも行われている．

### ❑ PGAS プログラミングモデル

さて，多数の実行コアを備えた並列分散環境におけるプログラミングでは，その並列性やメモリ構成をプログラマにどのように見せるかの選択が重要になる（**図 -1** 上）．並列性を隠蔽し処理系が暗黙的に並列化を行うモデル（右）では，プログラミングが容易になる反面，ハードウェアを活かしきった高性能な処理が難しくなる．一方，データの送受信まで明示的に記述する MPI のような並列化モデル（左）は，高性能だがハードウェア構成を意識したプログラミングが必要になってしまう．

X10 は，この 2 つの中間の「PGAS（Partitioned Global Address Space）プログラミングモデル」を採用している．このモデルでは，グローバルなアドレ

☆2 Defense Advanced Research Projects Agency.
☆3 High Productivity Computing Systems.
☆4 Productive Easy-to-use Reliable Computer Systems.
☆5 X10 という名前は，「（並列アプリケーションの）生産性を 10 倍向上させる」という目標からきている．

ス空間が複数の「プレース（Places）」に分割されている．プレースはメモリの局所性を抽象化したもので，典型的には 1 つのマシンに対応すると考えればよい．データはどれかのプレースに所属し，それをまたいで移動することはない．データは他のプレースからも参照することができるが，中身にアクセスできるのは同じプレースからのみである．

X10 は PGAS を採用し，並列分散環境を隠蔽するのではなく「抽象化して見せる」ことで，生産性を保ちつつ高性能を達成することを目指している．なお，PGAS 自体は X10 特有の概念ではなく，UPC[☆6] や CAF[☆7] などの言語でも採用されている．ただし，X10 では以下に述べるように非同期実行のサポートなどを強化しているため，区別して APGAS（Asynchronous PGAS）モデルと呼ぶこともある．

### ❑ X10 の実行モデル

X10 の実行モデルについて詳細に示したものが**図 -1** 下である．アドレス空間を分割した各プレースの中には，それぞれ複数の「アクティビティ」と「オブジェクト」が存在できる．

アクティビティは，プレース内で逐次動作する非同期な実行主体で，軽量なスレッドだと考えればよい．後述する `async` 文により，同じプレース内に動的に生成可能である．また，`at` 文により他のプレースに移動することができる．同じプレース内のアクティビティ間では同期・排他制御を行うことができるが，他のプレースのアクティビティに対しては，処理完了を待ち合わせる以外の同期は行えない．

オブジェクトは特定のプレースに所属する変更可能なデータ構造で，中身にアクセスできるのは同じプレースのアクティビティに限定される．ただし，後述する `GlobalRef` という仕組みにより，他のプレースからオブジェクトを「リモート参照」することができる．リモート参照には，参照しているオブジェクトがどのプレースに存在するかの情報が入っており，アクティビティがそのプレースに移動することでオブジェクトの中身にアクセス可能となる．

```
1 class MyHello {
2   public static def main(Array[String]) {
3     finish for (pl in Place.places()) {
4       async at (pl) Console.OUT.println(
5                     "Hello World from place " + here.id);
6 } } }
```

```
$ x10c MyHello.x10            # コンパイル
$ X10_NPLACES=4 x10 MyHello   # 実行
Hello World from place 3
Hello World from place 0
Hello World from place 1
Hello World from place 2
```

**図 -2　X10 による Hello World と，その実行例**

X10 ではさらに，複数のプレースにまたがった特殊なデータ構造として「分散配列（`DistArray`）」を作ることができる．分散配列の各要素は特定のプレースに所属しており，そのプレース内のアクティビティによって操作される．

X10 はデータのアクセスに関して，「変更可能（mutable）なデータは特定のプレースに所属しており，同じプレースのアクティビティからしかアクセスできない」という基本ポリシーで設計されている．一方，生成後は変更できない変更不能（immutable）データは，どのプレースからでもアクセス（読み出し）することができる．これには，クラスや，後述する構造体，関数などがある．

### ❑ Hello World

**図 -2** は，X10 で書いた Hello World プログラムの例である．4 行目の `async` 文により，新しいアクティビティが生成され，続く `at` 文により指定されたプレースへと移動する．そして，各プレースで `println` 文が実行され，プレースの ID（`here.id`）が出力されている．

図 -2 の下部は，このプログラムをコンパイル・実行した例である．X10 プログラムは現在，Java，C++，CUDA[☆8] などの環境に向けてコンパイル可能で，それぞれ使用するコマンドが異なっている

☆6 Unified Parallel C.
☆7 Co-Array Fortran.
☆8 Compute Unified Device Architecture.

が，ここでは取り扱いが簡単な Java 版を使ってコンパイル・実行する例を示した．なお，プレース数は構成によって変化するがこの実行例では 4 つである．アクティビティは並列に動作するため，出力の順序は一定でないことに注意してほしい．

次章以降では，X10 のデータ型や文法について解説していく．

## X10 の基本文法

X10 は，静的に型付けされたオブジェクト指向言語で，逐次処理部分の記法は Java に近いものになっている．本章では，Java との違いを中心に X10 の基本文法を説明する．並列・分散処理については次章で述べる．

### ❑ X10 プログラミング入門

**図 -3** は，X10 の記法を概観するためのサンプルプログラムである．X10 プログラムの実行は Java と同様，指定したクラスの `main` メソッドから開始される（7 行目）．ただし，引数の型は `Array[String](1)` となる．ここで，「`[String]`」は総称型クラス `Array` の型パラメータを示し，「`(1)`」は配列の次元が 1 という制約を表している．なお，配列のアクセスは「`( )`」を用いて行われる（9 行目）．

コメントは Java と同様に「`/*〜*/`」か「`//〜`」の形式で記述できる．また，パッケージ機構も Java と同様のものが利用可能で，ライブラリを `import` して使用することができる（1 行目）．なお，`x10.lang.*` と `x10.array.*` は自動的に `import` されるので指定しなくてよい．

Java と異なり，メソッドは `def` というキーワードで宣言する（4 〜 7 行目）．返り値の型はコロンの後に指定するが，推論できる場合は省略可能である．また，メソッド本体を「`=`」の右辺として記述することもできる（5 行目）．コンストラクタの宣言には型名でなく `this` を用いる（4 行目）．

```
 1 import x10.util.Pair; // struct to represent a pair
 2 public class MySample[T] {
 3   val data:T;
 4   def this(d:T) { data = d; } // constructor
 5   def get() = data;
 6
 7   public static def main(args:Array[String](1)):void {
 8     /* Add from 1 to arg */
 9     val end = (args.size > 0) ? Int.parse(args(0)) : 10;
10     var sum:Int = 0;
11     for (var i:Int = 1; i <= end; i++) sum += i;
12     Console.OUT.printf("Sum of 1-%d: %d\n", end, sum);
13
14     /* Object creation with generics */
15     val obj = new MySample[Double](1.2);
16     Console.OUT.println(obj.get()); // -> 1.2
17
18     /* Various data types */
19     val pair = Pair[Int,Int](3,4); // struct
20     val func = (i:Int,j:Int)=>i*j; // function
21     Console.OUT.println(func(pair.first,pair.second)); // -> 12
22   }
23 }
```

**図 -3　X10 の記法を概観するための例**

変数やフィールドの宣言は，`val` もしくは `var` というキーワードを用い，型名は変数名の後にコロンで指定する（10 行目など）．`val` は変更不能な変数の宣言に用いるが，初期値から型が推論できる場合は型の指定を省略できる[☆9]（9 行目など）．フィールドやメソッドには，Java と同様，`public` や `static` などの修飾子が指定可能である．

X10 の特徴として，クラスやインタフェースのほかに，構造体や関数（クロージャ）を型として使用可能なことが挙げられる．構造体は，値渡しされる変更不能なデータのまとまりで，コンストラクタの呼び出しは `new` なしで指定する（19 行目）．関数は「`=>`」を用いたリテラル表現（20 行目）や，メソッドセレクタ記法により生成され，「`( )`」により引数を指定して実行される（21 行目）．

クラス，インタフェース，構造体は，総称型として宣言することもできる（2 行目）．型パラメータは「`[ ]`」で指定し，実体化の際には必ず指定しなければならない（15 行目など）．

制御構文や演算子には，Java と同様のものが使用できる（11 行目など）．さらに，図 -2 で示した `async` や `at` のような並列・分散処理のための構文

☆9 変数の型は，コンパイル時に推論（infer）される．型指定を省略しても，実行時の変数の型が動的になるわけではない．

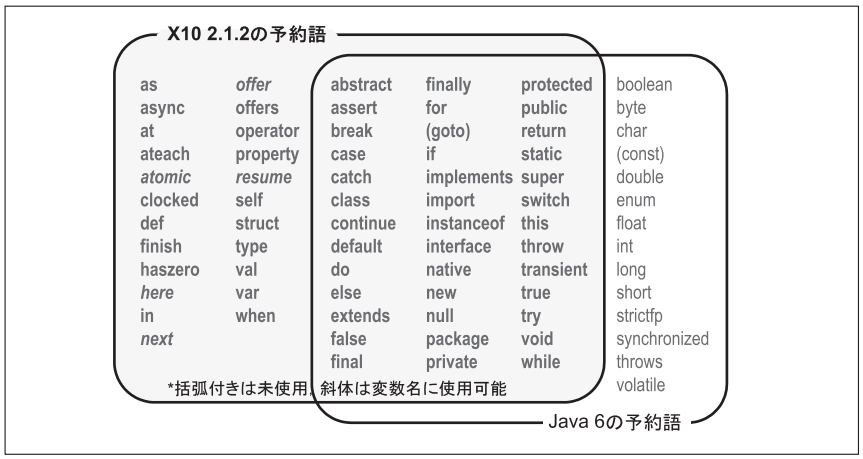


**図 -4　X10 と Java の予約語**

が追加されている．

## ❑ 基本型と制御構文

**図 -4** は X10 の予約語一覧である．比較のため，Java の予約語も掲載している☆10．X10 の予約語は Java とかなり重なっているが，並列・分散実行のためのキーワード等が追加されている．

### 基本型とリテラル

X10 では基本型は予約語ではない．`Boolean`，`Char`，`Byte`，`Short`，`Int`，`Long`，`Float`，`Double`，`UByte`，`UShort`，`UInt`，`ULong` が用意されているが，これらは標準ライブラリ内で構造体として定義されている．符号なしの数値型が用意されている点も，Java と異なる点である．X10 の基本型は，大文字で始まる名前（たとえば `Int`）が正式であるが，標準で読み込まれる `x10.lang._` クラス内で小文字で始まる名前が `type` 宣言されているため，そちら（たとえば `int`）を使うこともできる．

基本型にはそれぞれリテラル表現が存在している．記法は Java とほぼ同じであるが，`Short` 型，`Byte` 型のリテラルも，それぞれ末尾に `s`，`y` を付けることで表現可能である．また，符号なし型のリテラルを表すためには末尾に `u` を付ける．たとえば `0xBBUY` は `UByte` 型のリテラル（値は 187）である．

そのほかに，`"Hello\n"` は `String` 型，`[1,3,7]` は `Array[Int]` 型，`1..10` は `Region` 型のリテラルを表す記法である．また，「`(i:Int,j:Int):Int=>i*j`」のように書くことで「`(Int,Int)=>Int`」型の関数リテラルを表現することができる．

### 制御構文と演算子

X10 では，`if`，`else`，`switch`，`case`，`default`，`for`，`while`，`do`，`break`，`continue`，`throw`，`try`，`catch`，`finally`，`return` などの制御構文を Java と同様のイメージで使うことができる．ただし，`for` による `Iterable` 要素のイテレーションは「`for (i in iterableData) ～`」のように `in` を使って記述する．ほかに X10 特有の制御構文として，並列・分散処理を記述するためのものがあるが，それらについては次章で説明する．

式や演算子についても Java と同様のものが使えるが，キャストを「`v as TypeName`」の形で記述する点が異なる．また，X10 特有の演算子として，関数リテラルや関数型の定義に使う「`=>`」，`Region` 型のリテラル生成に使う「`..`」，特定プレースへの分散（`Dist`）を作る「`->`」，分散や分散配列の要素制限を行う「`|`」，型の包含関係を調査または指定するのに使う「`<:`」「`:>`」などがある．**表 -1** は，演算子の

☆10 厳密には，`true`，`false`，`null` は Java のキーワードではなくリテラルだが，ここでは便宜上予約語としている．

| 演算子（上のグループほど高優先度） | 結合規則 |
|---|---|
| ( ) [ ]（Array リテラル） new<br>' '（文字リテラル） " "（文字列リテラル）<br>.（フィールドアクセス，メソッドセレクタ）<br>( )（メソッド呼び出し，配列アクセス） | 左から右 |
| ++ --（後置） as（キャスト） | 左から右 |
| ++ --（前置） + -（単項） ~ ! | 右から左 |
| ..（Region リテラル） | なし |
| * / % | 左から右 |
| + - | 左から右 |
| << >> >>> ->（Dist リテラル） | 左から右 |
| < > <= >= <: :> instanceof in | 左から右 |
| == != | 左から右 |
| & | 左から右 |
| ^ | 左から右 |
| \| | 左から右 |
| && | 左から右 |
| \|\| | 左から右 |
| ? : =>（関数リテラル） | 右から左 |
| = *= /= %= += -= <<= >>= >>>= &= ^= \|= | 右から左 |
| , | 左から右 |

表 -1　演算子の優先度と結合規則

```
1 public class MyPoly(a:Int,b:Int,c:Int) { // denotes a*x^2 + b*x + c
2   public def toString() = ""+a+"*x^2 + " + b+"*x + " + c;
3   operator this(n:Int) = a*n*n + b*n + c;
4
5   operator this+(p:MyPoly) = new MyPoly(a+p.a, b+p.b, c+p.c);
6   static operator (p:MyPoly)*(q:MyPoly) {
7     if (p.a!=0 || q.a!=0) Console.OUT.println("unsupported");
8     return new MyPoly(p.b*q.b, p.b*q.c+p.c*q.b, p.c*q.c);
9   }
10  operator -this = new MyPoly(-a,-b,-c);
11  static operator (n:Int):MyPoly = new MyPoly(0,0,n); // coercion
12
13  public static def main(Array[String]) {
14    val X = new MyPoly(0,1,0);      // == 0*x^2 + 1*x + 0
15    val t = -X + 3;                 // -X+3
16    val u = 2*X + 1;                // 2X+1
17    val v = t * u;                  // (-X+3)(2X+1)
18    Console.OUT.println(v);         // -> -2*x^2 + 5*x + 3
19    Console.OUT.println(v(2));      // -> 5
20    val func = MyPoly.+; // operator function
21    Console.OUT.println(func(t,u)); // -> 0*x^2 + 1*x + 4
22  }
23 }
```

図 -5　演算子のユーザ定義例

一覧と優先順位を示したものである．

一部の演算子は，ユーザが定義することも可能である．**図 -5** は，二次多項式を表す MyPoly というクラスに「+」などの演算子をユーザ定義した例である．def の代わりに operator キーワードを用い，定義したい演算子が出てくる部分にそれを書いて宣言すればよい（5 行目）．静的メソッドとして定義（6 行目）したり，暗黙の型変換を定義（11 行目）することもできる．

## ❑ データ型

X10 では，オブジェクト（class），構造体（struct），関数（function）の 3 種のデータ型を扱える．本節では，これらの概要と関係について，**図 -6** のサンプルプログラムを参照しつつ説明する．

### オブジェクト（クラス）

オブジェクトは，Java のオブジェクトとほぼ同じで，クラス名を指定して new により生成され，「参照」を通じて使用される．オブジェクトは生成されたプレースから移動できないが，アクティビティが他のプレースに移動する際に自動的に複製される☆11．

オブジェクトの元となるクラスは，Java と同じく class 文によって定義され，extends による単一継承により，x10.lang.Object からの木構造を成す．抽象（abstract）クラスや拡張不可（final）なクラス，インタフェース（interface）も，Java と同様に定義可能である．ただし，X10 ではクラスの静的フィールドには，変更不能なもの（static val フィールド）しか定義できない．これは，どのプレースからでもアクセス可能なデータは変更不能というポリシーによる．

### 構造体

構造体は，「値渡し」されるひとまとまりのデータを表すための型で，class 文の代わりに struct 文で定義される（図 -6 の 1 行目）．この例では後述するプロパティ機能を使って構造体 MyPair のフィールド first と second を定義している．

X10 の構造体データは，内部を部分的に変更することはできない．つまり，val フィールドしか持てない．そのかわり，プレースをまたいで自由に読み出し可能である．構造体は extends で拡張

☆11 この挙動は X10 2.1 からである，詳しくは「プレース」の節で述べる．

```
 1 public struct MyPair[T,U](first:T, second:U)
 2                 implements (String)=>String {
 3   public operator this(s:String) = s+"("+first+","+second+")";
 4   static def curry[S](f:(S,S)=>S, b:S) = (x:S)=>f(x,b);
 5
 6   public static def main(Array[String]) {
 7     /* Struct example */
 8     val p = MyPair[Int,Double](1,2.3);
 9     Console.OUT.println(p("Data is ")); // -> Data is (1,2.3)
10     val x<:Int = 4; // exact type is Int{self==4}
11     val q = MyPair[(Int)=>Int, Int]((i:Int)=>i*x, 5);
12     Console.OUT.println(q.first(q.second)); // -> 20
13     var a:Any = p;
14     Console.OUT.println(a.typeName());
15                 // -> MyPair[x10.lang.Int,x10.lang.Double]
16
17     /* Function example */
18     val power = Math.pow.(Double,Double); // method selector
19     Console.OUT.println(power(2,5)); // -> 32.0
20     val square = curry[Double](power, 2);
21     Console.OUT.println(square(5));  // -> 25.0
22
23     val sq = (n:Int)=>{
24                 var s:Int=0; val abs_n = n<0 ? -n : n;
25                 for ([i] in 1..abs_n) s+=abs_n;
26                 s // return value of the closure
27               };
28     Console.OUT.println(sq(5));      // -> 25
29   }
30 }
```

**図 -6　構造体や関数の使用例**

することはできないが，メソッドや演算子を定義する（3 行目）ことや，インタフェースや関数を実装（implements）する（2 行目）ことは可能である．構造体データの生成は，new なしでコンストラクタを呼び出す☆12 ことで行われる（8 行目，11 行目）．

構造体は参照をとることができないので，オブジェクトの null に相当する値はない．オブジェクト内に構造体のフィールドを定義した場合，構造体はそのオブジェクト中にインラインされる．また，構造体データの同一性（==）は内容が同じかどうかで判断される．構造体は，長いビット数のスカラー値だと考えると分かりやすいだろう．X10 では，Int や Double などの基本型も，構造体として定義されている．

構造体データにはメタ情報が不要なので，オブジェクトよりもメモリ消費が少ないという利点がある☆13．一方，継承ができない，参照がとれず常に値渡しである，などの制約があるので，それほど多くない量のデータをまとめて扱いたい場合に用いるのがよいだろう．

## 関数

関数は，他の言語ではクロージャやラムダ式とも呼ばれているデータである．X10 の関数はオブジェクトや構造体と同じく一級データなので，変数に代入したり，引数として受け渡しも可能である（20 行目など）．構造体と同じく中身の変更はできないが，プレースをまたいで使用可能である．関数の呼び出しは，メソッドと同様に名前の後ろに「( )」で引数を指定して行う（21 行目など）．

関数のリテラルや型は，「$(a_1, a_2, \cdots) => b$」の形式で記述することができる（括弧は省略できない）．関数リテラルでは，左辺の $a$ の部分に仮引数の名前と型，右辺の $b$ の部分に処理内容を書く☆14．関数の型を記述する場合は，左辺に引数の型名，右辺に返り値の型名を書く．たとえば 11 行目の「(i:Int)=>i*x」は，引数として渡された Int 値を外部環境の x（関数定義中で束縛されていない変数，10 行目で宣言）と掛けて返す関数で，その型は「(Int)=>Int」（Int 値を 1 つ受け取り Int 値を返す関数）である．右辺に「{ }」ブロックを用いて，変数や制御構文を使った長い関数リテラルを記述することもできる（23 ～ 27 行目）．

関数データはほかに，メソッド名の後ろに「.」と引数の型を指定する「メソッドセレクタ記法」や型名の後ろに「.」と二項演算子を指定する「オペレータ関数記法」によって作ることもできる．18 行目の「Math.pow.(Double,Double)」や，図 -5 の 20 行目の「MyPoly.+」はその例である．

X10 の関数型は無名クラスのような扱いで，型を定義する function のようなキーワードはない．そのため，型パラメータを使用したり new で関数データを生成することはできない．一方，関数型はクラスや構造体で実装（implements）することができる（2 行目）．そのようなデータを関数として使用する（9 行目）と，引数の型に対応する「this(～)」演算子（3 行目）が呼び出される☆15．

☆12 X10 2.1.1 からは，new を付けても呼べるようになった．
☆13 ただし，Java 版の X10 では今のところ，基本型以外の構造体は Java オブジェクトにマップされているので，メモリ消費上のメリットはない．
☆14 関数リテラルの返り値の型は右辺の処理内容から推論されるが，明示したい場合は左辺の「)」の後にコロンをつけて指定することもできる．
例：「(i:Int,j:Int):Int=>i*j」．

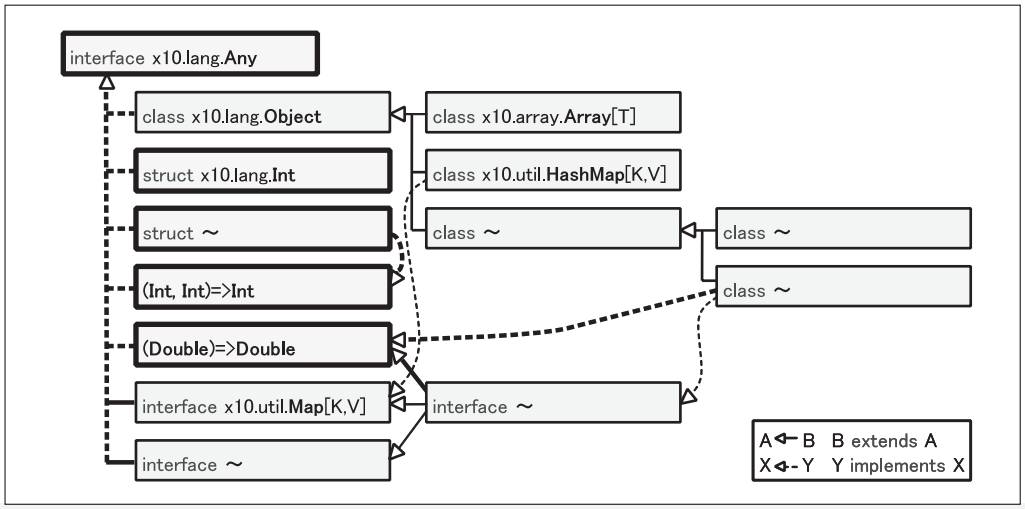


図 -7　型の階層構造

## 型の階層構造

**図 -7** は，クラス，構造体，関数，およびインタフェースの間の階層構造を示したものである．実線の矢印は継承（extends），破線の矢印は実装（implements）を表している．クラス部分については，Java と同様である．クラスは単一継承による階層構造をとり，すべて x10.lang.Object のサブクラスである．

図中，太線で示されているのが，X10 特有の部分である．X10 では Object の上に x10.lang.Any という型（インタフェース）がある．構造体や関数型は，すべて Any の直下にあり，それらの間の継承関係はない．ただし，クラスや構造体は複数のインタフェースや関数を実装することができる．

すべての型は，Any インタフェースを暗黙的に実装している．そのため，X10 のあらゆるデータは Any 型の変数に代入可能である（13 行目）．Any には，toString，typeName，equals，hashCode の 4 つのメソッドが宣言されており，どのデータに対してもこれらのメソッドを呼び出すことができる（14 行目）．

## 総称型

X10 では総称型（ジェネリクス）も利用可能で，クラスや構造体，インタフェースの定義では型パラメータを指定することができる（1 行目）．たとえば 8 行目の MyPair[Int,Double] は，Int 値と Double 値のペアを表す型となる．型パラメータには，関数を含むあらゆる型を使用することができる（11 行目など）．また，メソッド定義に型パラメータを指定することも可能である（4 行目）．

Java の総称型との主な違いとしては，型パラメータの指定法が「< >」ではなく「[ ]」であること，実装が型消去ではなく各データが型パラメータの実際の値を保持していること（15 行目），そのため使用時に型パラメータを省略することはできないこと，などが挙げられる．

## プロパティと制約

クラスや構造体，インタフェースの定義では，型名の後ろに「( )」を付加して「プロパティ」を宣言することができる．たとえば図 -6 の 1 行目では，MyPair 構造体に対して first と second というプロパティ（型はそれぞれ T と U）を宣言している．プロパティは変更不能なインスタンスフィールドで，コンストラクタ内で property[☆16] 文でまとめて設定されるが，ここで示した例のようにコンストラクタ定義ごと省略することも可能で，その場合はプロ

☆15 実はこの演算子は implements を宣言していなくても呼べる（「配列」の節を参照）が，宣言した場合は対応する this（～）演算子を実装していないとコンパイルエラーになる．
☆16 property キーワードは，「プロパティメソッド」の定義にも使用されるが，本稿では省略する．

```
 1 public class MyAssoc[T] { // simple associative array
 2   val map = new x10.util.HashMap[String,T]();
 3   operator this(key:String)=(value:T) = map.put(key, value);
 4   operator this(key:String) = map.getOrThrow(key);
 5
 6   public static def main(Array[String]) {
 7     /* Array-like access */
 8     val aa = new MyAssoc[Int]();
 9     aa("one") = 1;             // ==operator this("one")=1
10     aa("two") = 1 + aa("one"); // ==operator this("one")
11     Console.OUT.println(aa("two")); // -> 2
12
13     /* Basic array example */
14     val arr = new Array[Int](1000);
15     for ([i] in 1..arr.size) arr(i-1) = i;
16     var s:Int = 0;
17     for (pt in arr) s += arr(pt)*arr(pt);
18     Console.OUT.println(s); // -> 333833500
19
20     /* Point */
21     val pt1 = [1,3,7] as Point;      // rank==3
22     Console.OUT.println(pt1(1));     // -> 3
23     val [a,b,c] = pt1;               // a==1, b==3, c==7
24     val pt2[x,y] = Point.make(1,3);  // x==1, y==3
25     Console.OUT.println(pt2);        // -> [1,3]
26
27     /* Region */
28     val R1 = 3..5;       // rank==1
29     val R2 = (-1..1)*R1; // rank==2
30     Console.OUT.println(pt2 in R2);  // -> true
31
32     /* Array */
33     val arr1 = new Array[Int](R2, ([i,j]:Point)=>i*j);
34     Console.OUT.println(arr1(pt2));  // -> 3
35     Console.OUT.println(arr1(-1,4)); // -> -4
36   }
37 }
```

**図 -8　配列の使用例**

パティ値を引数にとるコンストラクタが自動的に用意される．

X10 では，型やメソッドに「{ }」を付加して「制約」をつけることが可能で，プロパティはこの制約の記述に用いることができる．たとえば，`MyPair[Int,Int]{first!=0}` は，第一要素の値を 0 以外に制限した `MyPair` 型である．また，値「4」の最も厳密な型は `Int{self==4}` である．ここで，`self` は制約記述の中で用いる自分自身を表すキーワードである．

制約が満たされているかどうかは，基本的にコンパイル時に型推論によりチェックされる☆17．X10 の型推論は非常に強力で，制約も含めた推論が行われる．そのため，`val` 変数の定義では型を明示的に書かず初期値から推論させた方が，より厳密な型が設定されて都合がよいことが多い☆18．

X10 プログラミングで最もよく使われるプロパティは，配列などの次元を表す「`rank`」であろう．たとえば，「`Array[String]{rank==1}`」は，1 次元の文字列配列を示す型である．これは，図 -3 の 7 行目のように「`Array[String](1)`」と書くこともできる．これは，標準で読み込まれる `x10.lang._` クラス内で，`type` という構文を使って，

```
public static type Array[T](r:Int)
                = Array[T]{self.rank==r};
```

と別名定義されているからである．

## ❑ 配列

X10 には，Java のような言語コンストラクトとしての配列は存在しない．しかし，「`this(〜)=〜`」および「`this(〜)`」という演算子により，配列アクセス風の処理を一般化した形で実現している☆19．前者が配列要素の設定，後者が配列要素の読み出しに対応する処理を行う．**図 -8** は，これらの演算子をユーザ定義することで連想配列を実装した例である．3 行目で演算子「`this(String)=T`」，4 行目で演算子「`this(String)`」を定義し，9 〜 10 行目で配列アクセス風の呼び出しを行っている．

X10 の標準クラスである `x10.array.Array` は，同様の仕組みを使って配列を提供している．図の 14 行目からは，その使用例である．14 〜 15 行目で，要素数 1000 の 1 次元整数配列を作成し，1 〜 1000 の値を設定している．ここで内部的には「`this(〜)=〜`」演算子が呼ばれる．16 〜 17 行目では，この配列の各要素を二乗して合計する処理を行っている．ここでは「`this(〜)`」演算子が呼ばれる．

X10 の `Array` は多次元配列もサポートしている．また，添え字は 0 から始まらなくてもよい．これらは，以下で述べる `x10.array.Point` と `x10.array.Region` というクラスを用いて実現されている．

☆17 静的に判定できない場合は，実行時にチェックするコードが (警告つきで) 生成される．オプション指定でコンパイルエラーとすることもできる．
☆18 どうしても型を明示したい場合は，図 -6 の 10 行目のように「`<:`」を用いて「上限」を指定することもできる．
☆19 この機能は，X10 2.1.1 までは `set` および `apply` というメソッドによって実現されていた．

### Point クラス

Point は $n$ 次元整数格子上の 1 点を表すクラスで，$n$ 次元配列の「添え字」として使用できる．たとえば，図中 21 行目の pt1 は 3 次元上の点 (1, 3, 7) を表している．Point 型のデータは，整数配列からキャストしたり（21 行目），Point.make メソッド (24 行目) で作成できる．各次元の要素は Int 型に限定されており，負の値も可能である．pt1(1) のようにして読み出せる（22 行目）が，個別に変更することはできない．なお，次元数 $n$ は rank プロパティとして保持されており，制約の指定にも使用できる．

Point では，「分解型」のアクセス法 (destructuring syntax) がサポートされている．これは，変数宣言時に「[ ]」を使って指定される．23 ～ 24 行目がその例で，変数 a，b，c，x，y に Point の各次元の値が設定される．実は，「for ([i] in 1..10) ～」のような記法 (15 行目など) はこれを利用したものである☆20．

### Region クラス

Region は次元（rank）が同じ Point の集合を表すクラスで，配列の「有効な添え字（定義域)」を示すのに使用できる．28 行目の「3..5」は，1 次元の Point (3) (4) (5) の 3 点からなる，Region{rank==1} 型のリテラルである．Region クラスではさまざまな演算が定義されているが，よく使うのは「*」による直積である．29 行目の R2 は，(−1, 3) (−1, 4) (−1, 5) (0, 3) (0, 4) (0, 5) (1, 3) (1, 4) (1, 5) の 9 点からなる Region{rank==2} となる．

なお，Region には必ずしも実際の Point が入っているわけではなく，集合の「形」が論理的に記録されている．ある Point が Region に含まれるかどうかは，in 演算子や contains メソッドで調べることができる (30 行目)．

### Array クラス

Array[T] は要素の型が T の配列を表すクラスで，その定義域は Region で指定される．Region を用いることで，X10 では多次元配列や添え字が負の配列，要素が疎な配列，三角行列なども柔軟に表現できるようになっている．

たとえば 33 行目の arr1 は，29 行目で定義した 2 次元 Region R2 を定義域とする整数配列である．Array にはさまざまなコンストラクタがあるが，ここでは第 2 引数に (Point)=>T 型の関数をとるものを用い，生成時に初期値を計算・設定するようにしている．

配列の各要素には，定義域に含まれる Point や，各次元の Int 値を直接指定することでアクセスできる（34 ～ 35 行目）．Region 外のインデックスを使用すると，x10.lang.ArrayIndexOutOfBoundsException となる．

なお，Array のサイズ（要素数）は Java と同じく変更不能で，size プロパティに入っている (15 行目)．また，Array や Region は Iterable[Point] を実装しているので，含まれる Point を for 文で順に取り出すことができる (17 行目)．

## 並列・分散プログラミング

前章では，X10 の基本的な文法とデータ型について，Java と比較しながら解説したが，使用した例はいずれも，1 プレース・1 アクティビティで逐次実行されるものであった．しかし，X10 の本領は，複数アクティビティを用いた並列処理や複数プレースにまたがった分散処理を容易に記述できる点にある．

本章では，並列・分散処理のための機能と，それらを利用するための特徴的な構文やデータ構造について解説する．

☆20 この記法の「[ ]」を取ると，Int でなく Point に対するイテレーションになる．

```
 1 public class MyFib {
 2   var r:Int; // in-out parameter
 3   def this(i:Int) { r = i; }
 4   def run() {
 5     if (r < 2) return r; // MyFib(0)==0, MyFib(1)==1
 6     val f1 = new MyFib(r-1), f2 = new MyFib(r-2);
 7     finish {
 8       async f1.run(); // compute MyFib(r-1) in parallel
 9             f2.run(); // compute MyFib(r-2) by myself
10     }
11     return (r = f1.r + f2.r); // MyFib(r-1) + MyFib(r-2)
12   }
13
14   public static def main(args:Array[String](1)) {
15     /* Compute a Fibonacci number in parallel */
16     val n = (args.size>0)? Int.parse(args(0)) : 10;
17     val f = new MyFib(n); f.run();
18     Console.OUT.println("Fib("+n+")="+f.r); // -> Fib(10)=55
19
20     /* Place shift */
21     Console.OUT.println(Place.MAX_PLACES); // -> 4 (etc.)
22     val r = at (here.next()) new MyFib(10).run();
23     Console.OUT.println(r);   // -> 55
24
25     /* Local variable access */
26     var m:Int = 0;            // mutable
27     val i = 1;                // immutable
28     finish async { m = i; }   // ok
29     Console.OUT.println(m);   // -> 1
30     val h = here;
31     at (here.next()) {
32       Console.OUT.println(i); // -> 1
33       //     m = 2;           // compile error
34       at (h) m = 2;           // ok
35     }
36     Console.OUT.println(m);   // -> 2
37
38     /* Object and GlobalRef */
39     val o = new MyFib(0);
40     at (here.next()) o.r = 1; // copy of o is modified
41     Console.OUT.println(o.r); // -> 0
42     val g = GlobalRef[MyFib](o);
43     at (here.next()) { at (g.home) g().r = 2; }
44     Console.OUT.println(o.r); // -> 2
45   }
46 }
```

**図 -9　アクティビティとプレースの使用例**

### ❑ アクティビティ

「実行モデル」の節で述べたように，X10 の実行環境は複数の「プレース」から構成され，プレース内では複数の「アクティビティ」を実行できる．アクティビティの生成は，async 文で行われる．**図 -9** の 1 ～ 18 行目は，複数のアクティビティを生成してフィボナッチ数を並列計算するプログラムの例である．8 行目の async 文で 1 つ前のフィボナッチ数を並列計算している間に，9 行目で 2 つ前のフィボナッチ数を計算している．7 行目の finish 文は，そのブロック自身と内部で生成したアクティビティ（孫も含む）がすべて終了するまで待つ文である．アクティビティは再帰的に多数生成されるが，すべてが終了するとこの finish 文を抜け，11 行目で結果が合計される．

finish 文ではその内部のすべてのアクティビティの終了を確定できるので，これを try～catch で囲むことで，非同期処理中に起きた例外もキャッチすることができる．これを「Rooted Exception Model」と呼ぶ．複数のアクティビティで例外が起きた場合は，x10.lang.MultipleExceptions が送られる．

X10 の async 文は，ハンドルのようなものは返さない．そのため，アクティビティの終了を待ち合わせることができるのは，親アクティビティかその先祖だけであり，待ち合わせがループすることはない．これにより，意図せぬデッドロックを言語仕様レベルで抑制している．アクティビティは，main を実行開始する「初期アクティビティ」をおおもととする木構造を成し，それらが全部終了すると X10 プログラムが終了する．main の呼び出しの外側に仮想的な finish 文があると考えればよい．

async ブロック内では，その外側で定義されたローカル変数を読み書きすることができる（28 行目）．ただし，複数のアクティビティから同時に書き込まないように注意が必要である．そのための排他制御については，節をあらためて述べる．

### ❑ プレース

プレースはメモリを共有する 1 つの実行環境を抽象化したもので，典型的には個別のマシンに対応すると考えればよい．アクティビティは，at 文で，別のプレースに移動することができる．22 行目は，隣のプレース（here.next()）でフィボナッチ数を計算する例である．ここで，here は「現在のプレース」を示す x10.lang.Place 型構造体の変数である．プレースには 0 から Place.MAX_PLACES−1 の続き番号が振られており，id プロパティとしてアクセスできる（図 -2 の 5 行目）．

at 文では，新しいアクティビティが生成される

わけではなく，意味的にはそのアクティビティ自体が指定プレースに「移動」して処理が実行される．内部の処理が終了すると元のプレースに処理が戻ってくるが，この際に値を返すこともできる（22 行目）．at 内で生成したアクティビティの処理は終了していなくてもよいので，「at（place）async ～」（もしくは「async at（place）～」）で，指定プレースでアクティビティを分散動作させることが可能である．最初に示した図 -2 はその例である．

概要でも述べたように，「変更可能なデータは特定のプレースに所属しており，同じプレースからしかアクセスできない」というのが X10 の基本ポリシーである．そのため at ブロック内では，その外側で定義された val 変数を読むことはできる（32 行目）が，var 変数にアクセスするには，それが宣言されたプレースに戻る必要がある（33 ～ 34 行目）．

構造体や関数，クラスは変更不能なデータであり，どのプレースからも自由にアクセスできる．一方，オブジェクトは特定のプレースに所属しており，生成されたプレースからしかアクセスできない．at ブロック内からオブジェクトにアクセスすると，暗黙のうちに複製が作られる☆21 ため，取り扱いには注意が必要である（40 ～ 41 行目）．

オブジェクトを x10.lang.GlobalRef[T] という構造体に入れることで，暗黙の複製を抑制し，同一オブジェクトへのグローバルな参照を持つことができる．ただし，中身にアクセスするには，オブジェクトの存在するプレース（GlobalRef の home プロパティに入っている）に移動する必要がある．42 ～ 43 行目がその使用例である．

```
 1 public class MyBuf {
 2   var datum:Int = 0, filled:Boolean = false;
 3   def send(v:Int) {
 4     when (!filled) { filled = true;  datum = v; }
 5   }
 6   def receive():Int {
 7     when ( filled) { filled = false; return datum; }
 8   }
 9   static atomic def say(s:String) = Console.OUT.println(s);
10
11   public static def main(Array[String]) {
12     /* Atomic block */
13     var s:Int = 0;
14     finish for ([i] in 1..1000) async {
15       val tmp = i*i; atomic { s += tmp; }
16     }
17     Console.OUT.println(s); // -> 333833500
18
19     /* Conditional atomic blocks */
20     val buf = new MyBuf();
21     finish {
22       for ([i] in 0..9) async buf.send(i); // senders
23       for ([i] in 0..9) Console.OUT.print(buf.receive());
24       Console.OUT.println();    // -> 0217635489 (etc.)
25     }
26
27     /* Clocked execution */
28     clocked finish {
29       clocked async { // clocked activity A
30         say("A-1    "); next;
31         say("    A-2"); next;
32         say("A-3    ");
33       }
34       clocked async { // clocked activity B
35         System.sleep(1000); // sleep 1 sec
36         say("B-1    "); next;
37         say("    B-2"); next;
38         say("B-3    ");
39       }
40       clocked async { // clocked activity C
41         for ([i] in 1..3) {
42           val spc = (i%2==0) ? "    " : "";
43           say(spc + "C-" + i); next;
44     } } }
45 } }
```

実行例

```
A-1
C-1
B-1
    C-2
    B-2
    A-2
B-3
C-3
A-3
```

図 -10　排他・同期制御の例

## ❑ 排他・同期制御

同一プレースで動作するアクティビティの間では，さまざまな同期制御を行える．**図 -10** はその例である．

atomic 文により，プレース内で不可分に実行されるブロックを記述することができる．13 ～ 16 行目では，1000 個のアクティビティを生成し $1^2+2^2+...+1000^2$ を計算しているが，s に値を足し込む部分を atomic 文により排他制御している（15 行目）ので，正しい結果が得られる．メソッド定義に atomic をつけて，メソッド全体を atomic ブロックとして扱うこともできる（9 行目）．不可分処理のためにはほかに，x10.util.concurrent パッケージのクラス（AtomicInteger など）を利用することもできる．

「when（condition）～」は，指定された条件が成立するまでブロックする構文である．条件のチェックと本体の実行は，不可分（atomic）に行われる．図 -10 の MyBuf は 1 つの Int 値を保持できる

☆21 この挙動は X10 2.1 からのもので，将来的に Java オブジェクトを X10 オブジェクトと区別せず扱えるようにするための準備として導入された．オブジェクトのフィールドを transient と宣言することで，値の複製を抑制することもできる．

バッファで，when 文でバッファの状態 filled をチェックして送り手と受け手の同期制御を行っている(2 ～ 8 行目)．22 ～ 23 行目が使用例で，10 個のアクティビティから値を送り，元のアクティビティでそれらを受け取っている．

アクティビティ間の同期のためにはほかに，「クロック」という仕組みが用意されている．これは，全タスクが特定の処理を終えるまで先に進むことを抑止する「バリア同期」の概念を拡張して，複数のフェーズに対して繰り返し使えるようにしたものである．使用するにはまず，クロックに複数のアクティビティを「登録」する．各アクティビティは，1 フェーズ分の処理を終えると next を実行する．すると，登録されている全アクティビティが next を呼び出すまで，処理がブロックされる．

28 ～ 44 行目が，クロックの使用例である．28 行目の「clocked finish」により，クロックが生成される．その内部の「clocked async」は，アクティビティを生成し，そのクロックに登録する構文である．ここでは，3 つのアクティビティ A，B，C を生成・登録している．各アクティビティでは，順に 1，2，3 を出力しているが，それぞれの間で next を呼び，バリア同期を行っている．そのため，図の右下に示した実行例のように，A，B，C の 3 つ分の出力が揃わないと次の出力へは進まない．

クロックへの登録は，アクティビティが終了すると解除される．また，通常の finish 文と同様，内部のアクティビティがすべて終了すると，clocked finish 自体が完了し，クロックも解放される．

複雑なバリア同期のためには，resume 文でフェーズ終了を「早めに伝える」ことや，x10.lang.Clock オブジェクトを明示的に生成したり複数用いることも可能であるが，本稿では省略する．

並列プログラミングの難しさの 1 つは，デッドロックの回避である．X10 では，デッドロックが起きにくいように言語仕様レベルで工夫がなされている．具体的には，以下の条件を満たす X10 プログラムは，デッドロックしない(できない)．

1. at，async，finish，atomic は自由に使ってよいが，when は使わない．
2. クロックは使ってよいが，finish ブロックの外側で生成したクロックを内側で使わない．

2 番目の条件は分かりにくいが，図 -10 で示したように clocked finish と clocked async をセットで使用し，途中に余計な finish をはさまなければ問題は生じない．

### ❑ 分散配列

複数のプレースにまたがった分散処理のためのデータ構造として，X10 では，要素が各プレースに散らばった「分散配列（x10.array.DistArray)」を作成可能である．分散配列の各要素がどのプレースに存在するかは，分散（x10.array.Dist）データによって示される．**図 -11** は，これらのクラスの使用例である．

#### Dist クラス

Dist は，Region 内の各 Point から Place へのマッピングを表すクラスで，分散配列の各要素がどのプレースに存在するかを示すのに使われる．

最も一般的な Dist の作成方法は，Dist.makeBlock(region) である．これにより，指定された Region 内の点をプレース間で均等に分割した Dist が生成される．たとえばプレース数が 4 の場合，Dist.makeBlock(0..9) は， 点 (0) (1) (2) をプレース 0，(3) (4) (5) をプレース 1，(6) (7) をプレース 2，(8) (9) をプレース 3 にマップする Dist になる(4 行目)．

特定の Point がどのプレースにマップされているかは，Dist 変数を関数として呼び出すことで調べられる（6 行目）．これは，Dist クラスに this(Point):Place や this(Int):Place という演算子が定義されているからである．

「|」演算子で，既存の Dist から要素を制限した

新しい Dist を作成することができる（7 ～ 8 行目）．また，「->」演算子で，Region 内のすべての点を特定のプレースへとマッピングすることができる（9 行目）．ほかに，Dist.makeUnique() では，存在するプレース分の単純なマッピングが生成される（10 行目）．

### DistArray クラス

DistArray[T] は要素の型が T の分散配列を表すクラスである．定義域と要素の存在場所は作成時に Dist で指定される（後から変更することはできない）．Dist を用いることで，X10 では分散配列の各要素とプレース（≒実マシン）との対応を柔軟に制御することが可能になっている．

DistArray は，new ではなく DistArray.make[T] というファクトリーメソッドで，Dist を指定して生成する．第 2 引数に関数を渡して，生成時に初期値を設定することもできる（15 行目）．分散配列の各要素にアクセスするには，その要素が存在するプレースに移動する必要がある（17 行目）．異なるプレースからアクセスすると，x10.lang.BadPlaceException となる．

20 ～ 33 行目は，DistArray を用いて $1^2+2^2+\ldots+1000^2$ の計算を分散して行うプログラム例である．まず，1 ～ 1000 の値が入った DistArray da0 を生成する（20 ～ 21 行目）．そして，各プレースでの計算結果を受け取る配列 tmp を用意し（22 ～ 23 行目），分散して計算を行う（24 行目）．各プレースではまず，「|」演算子で da0 の自プレース分の部分配列を作り（26 行目），つづく for 文で各要素を二乗して合計している（28 行目）．全プレースの計算終了を finish（24 行目）で待った後，各プレースの計算結果を合計して結果を得る（32 行目）．

実は，DistArray には同様の処理を簡単に行うためのメソッドが用意されている．各要素に対し特定の処理を行う map メソッドと要素を特定の処理方式で集約する reduce メソッドを組み合わせ，適切な関数を渡すことで，1 行で記述することができる（36 行目）．

```
 1 public class MyDist {
 2   public static def main(Array[String]) {
 3     /* Dist */
 4     val D1 = Dist.makeBlock(0..9);
 5                // (0..2)->0 (3..5)->1 (6..7)->2 (8..9)->3
 6     Console.OUT.println(D1(3)); // -> (Place 1)
 7     val D2 = D1 | here;  // (0)->0 (1)->0 (2)->0
 8     val D3 = D1 | 2..4;  // (2)->0 (3)->1 (4)->1
 9     val D4 = 0..9->here; // (0..9)->0
10     val D5 = Dist.makeUnique(); // (0)->0 (1)->1 (2)->2 (3)->3
11
12     /* DistArray */
13     val R = (1..4)*(5..6);
14     val D = Dist.makeBlock(R); Console.OUT.println(D);
15     val da = DistArray.make[Int](D, ([i,j]:Point)=>i*j);
16     for (pt in da)
17       Console.OUT.println(at (da.dist(pt)) da(pt));
18
19     /* Advanced DistArray example */
20     val D0 = Dist.makeBlock(1..1000);
21     val da0 = DistArray.make[Int](D0, ([i]:Point)=>i);
22     val places = da0.dist.places(); // Sequence[Place]
23     val tmp = new Array[Int](places.size);
24     finish for ([i] in 0..(places.size-1)) async {
25       tmp(i) = at (places(i)) {
26                  val a = da0 | here; // restriction
27                  var s:Int = 0;
28                  for (pt in a) s += a(pt)*a(pt);
29                  s // return value of at
30                };
31     }
32     var s0:Int = 0; for (pt in tmp) s0 += tmp(pt);
33     Console.OUT.println(s0); // -> 333833500
34
35     /* DistArray.map and reduce */
36     val s1 = da0.map((i:Int)=>i*i).reduce(Int.+, 0);
37     Console.OUT.println(s1); // -> 333833500
38   }
39 }
```

図 -11　分散配列の使用例

## マルチコア時代と X10

以上，本稿では X10 の設計思想，基本的なデータ構造と文法，および並列・分散処理の概要について解説した．より詳しい内容としては，http://x10-lang.org/[1] から言語仕様[2] やチュートリアル資料を入手できる．ソースコードは同サイトのリンクから SVN で取得可能で，サンプルプログラムも多数含まれている．日本語の資料としては，この解説のほかセミナー資料[3] などがある．本稿ではクラスライブラリについてはあまり触れられなかったが，HashMap など基本的なものについては，Java と同

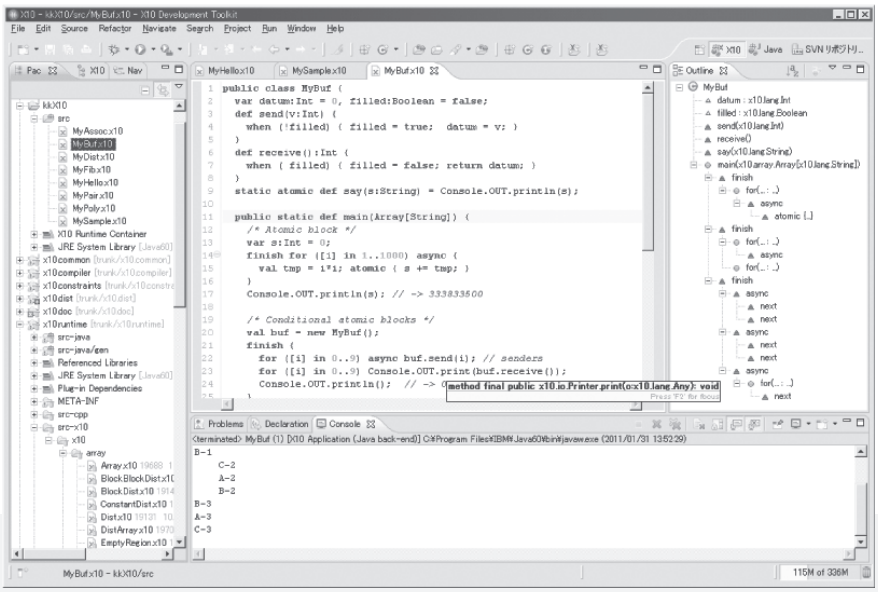

**図 -12　X10DT の使用イメージ**

様に利用可能である．これも上記サイトから仕様を確認できる．

X10 は，C＋＋か Java 環境が用意されていれば動作可能で，MPI，GPGPU や Blue Gene なども含めたさまざまな環境で使用できるが，まずは試してみたいという場合は，開発ツールである X10DT（X10 Development Toolkit）を使うのがよいだろう．これは Eclipse 用のプラグインで，個別インストールすることもできるが，Eclipse 実行環境も含んだものが配布されているので，それを利用するのが簡単である．**図 -12** は，X10DT の使用イメージである．X10DT 上で，X10 プログラムの開発と実行を行うことができる．本稿に掲載したプログラムは，それぞれ単体でコンパイル・実行が可能なので試してもらいたい．

さて，本稿の冒頭で，マルチコア時代のプログラミング環境に求められる要件を挙げた．それらについての X10 の対応は以下のとおりである．まず，非同期タスクは，`async` 文により容易に生成可能で，デッドロックが起きないような構文上の工夫が行われている．また，実行モデルとして「APGAS」を採用し，言語レベルで「プレース」という概念を導入することで，共有メモリ環境と分散メモリ環境を単一のプログラムで扱えるようになっている．プレース間の通信は，`at` 文や `DistArray` によって抽象化されており，プレースごとにアーキテクチャが異なっていてもかまわない．そして，開発をサポートするツールとして，X10DT が提供されている．

マルチコアの一般化により，専門家だけが並列・分散プログラミングを行う時代ではなくなりつつある．そのため，新しいプログラミングモデルと，言語の基本コンストラクトとして並列・分散実行を指定できる機能は必須となっていくだろう．それらを備えた X10 のような新言語が，マルチコア時代のソフトウェアの生産性と性能向上に役立つことを期待してやまない．

**参考文献**


1）X10 Home, http://x10-lang.org/
2）Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. and Grove, D.：X10 Language Specification, http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf
3）河内谷清久仁：X10 セミナー：発表資料とサンプルプログラム，http://www.trl.ibm.com/people/kawatiya/X10seminar.htm


（平成 23 年 1 月 5 日受付）


**河内谷清久仁（正会員）**　kawatiya@jp.ibm.com
1987 年東京大学大学院情報科学修士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所にてオペレーティング・システムやプログラミング言語処理系の研究に従事．博士（政策・メディア）．