Vol. 40 No. SIG 7 (PRO 4) 情報処理学会論文誌 Aug. 1999

場所の概念を備えたJava言語とその処理系

渡 辺 昌 寛† 伊 藤 貴 康†

インターネット環境において，計算機は分散化されたサイトに位置している．分散計算環境下のオブジェクトのマイグレーションやリモートオペレーションはオブジェクトの位置する場所を用いて自然に記述することが可能である．本論文では，ロケーション（location）とリージョン（region）から成る場所の概念と付随する分散計算機能を導入し，Java言語を拡張することを提案する．ロケーションはオブジェクトの属する所在位置であり，リージョンはリージョンマネージャにより許可を受けたものがアクセス可能となるロケーションの集合である．マイグレーション，リモートオペレーション，failure機能のようなモバイルオペレーションはロケーションとリージョンを用いて明示的に記述され，オブジェクトのモバイルオペレーションはリージョンマネージャによる許可が与えられた場合にのみ許容される．従って，ユーザはリージョンおよびリージョンマネージャを用いてシステムのセキュリティを向上することができる．Javaのサブセットにロケーションとリージョンを備えたmJava/LRを設計し，その処理系を試作した．mJava/LRによってモバイルオペレーションとそのセキュリティ設定がロケーションとリージョンおよびリージョンマネージャを用いて実現できることを示す．

Design and Implementation of Java with Locations and Regions

MASAHIRO WATANABE† and TAKAYASU ITO†

In an internet environment a computer is located at a distributed site. In distributed computing environments migration of objects and remote operation on objects can be naturally described using places where objects are located. We extend Java, introducing locations and regions and their related operations. A location is a place where objects are located, and a region is a set of locations accessible by objects with permissions by the region manager. Mobile operations like migrations, remote operation and failure mechanism can be explicitly described by means of locations and regions. The region is designed in such a way that mobile operations of objects would become effective only when they are approved by the region manager, so that a user can use regions and their region managers to specify and increase the system's security. A subset of Java with locations and regions, called "mJava/LR", is designed and implemented to show how mobile operations and their security can be realized with use of locations and regions.

1. はじめに

Java言語[1]はインターネット環境で広く用いられているコンパクトなオブジェクト指向言語である．Java Virtual Machine (JVM)[2]の採用によって，プラットフォーム独立やネットワーク安全など分散処理向言語として優れた機能が実現されている．電子商取引のようなJavaのアプリケーションでは，ネットワーク安全に加えて高信頼性やセキュリティの向上が要求される．例えば，ネットワーク上のあるサイトにおけるシステムの故障を回避するプログラムを記述することや，不法なユーザから自分のサイトやWebページを攻撃された場合にサイトやプログラムを保護するための方法が，ユーザに必要とされる．通常，システムのセキュリティ設定法として用いられているのはパスワードや電子署名である．しかし，パスワードはしばしば盗用され，電子署名はインターネットの平均的なユーザにとって一般的ではなく，さらにこれらが盗用されてしまった場合，ユーザはネットワーク管理者の助けを借りて対処する必要がある．

本論文では，ロケーション（location）とリージョン（region）からなる場所の概念と付随する分散機能を導入し，ユーザによるセキュリティ設定とエラー回避が可能なJava言語の拡張を提案する．ロケーションはオブジェクトが属する所在位置である．リージョンは，リージョンマネージャの管理を受けるロケーショ

† 東北大学大学院情報科学研究科
Department of Computer and Mathematical Sciences, Graduate School of Information Sciences, Tohoku University

ンの集合である．インターネット環境におけるオブジェクトの移動や遠隔操作は，リージョンからリージョンにまたがるとき，リージョンマネージャによって許可されたロケーションのみをアクセスできる．オブジェクトのマイグレーションやリモートオペレーション，failure 検出などのオペレーションはロケーションおよびリージョンを用いて明示的に記述される．リージョン内へ侵入またはリージョンの外へ退出するようなモバイルオペレーションはリージョンマネージャの許可が必要である．ユーザはリージョンマネージャを用いて，特定のリージョンに対するセキュリティ設定を行うことができる☆．またロケーションとリージョンは，それらを用いて指定されたネットワーク上の場所での故障を回避し部分的に復帰するために用いることもできる．

ロケーションを用いた分散計算用の言語としては Telescript 等[3)4)5)]が存在し，ロケーションを用いた failure 機能として分散 Join Calculus[6)] が提案されている．リージョンとリージョンマネージャを用いたユーザによるセキュリティ設定機能を備えた Java の実現が本研究の特徴である．なお，リージョンマネージャによるセキュリティ設定は、電子署名と直交する概念であり，両者を併用することによってより高いセキュリティシステムを構築することも可能である．

本論文の構成は次の通りである．2 章でロケーションおよびリージョンを備えた Java 言語の基本機能について説明する．3 章では failure 機構とそれを用いたネットワーク故障からの回避について述べる．4 章では，ロケーションおよびリージョンを定義した後，ユーザによるセキュリティ設定を実現するリージョンマネージャのとその実現法について述べる．5 章ではロケーション及びリージョンを用いた Java の拡張言語構文について述べ，それらを用いたセキュリティ設定の例を与える．6 章で mJava/LR とその処理系の概要について述べ，7 章の結言においては，関連研究や今後の課題についても述べる．

2. ロケーションとリージョンを用いた Java 言語の基本機能

この章ではロケーション及びリージョンを備えた Java 言語の基本機能を Java の知識を前提として説明する．これらの機能を実現する Java 言語の拡張構文は 5 章に与えた．

☆ 本論文で扱うセキュリティとはシステムへの不正なアクセスや破壊行為，情報の盗用を防ぐため，オブジェクトの不正な侵入・退出や処理を防止あるいは禁止するものである．

Java にロケーションおよびリージョンを導入することによりオブジェクトの概念が拡張される．先ず，これについて述べる．

[a] オブジェクトの機能の拡張

Java にロケーションとリージョンおよびそれらに関連するオペレーションを加えることにより，次のような拡張が行われている．

1) **オブジェクトとその所在位置** 全てのオブジェクトはいずれかのロケーションに属し，オブジェクト名 id を持つ obj_{id} は次のように定義される．

$$obj_{id} = [(id, loc), \{o_1, \ldots, o_n\}]$$

ここに，loc は obj_{id} の所在するロケーションであり，$o_1, \ldots, o_n$ は obj_{id} に属するサブオブジェクトである．ロケーションの詳細は 4 章で述べる．

2) **オブジェクトのマイグレーション** オブジェクトはマイグレーション機能により，あるロケーションから他のロケーションへと移動することが可能である．オブジェクト `obj` のロケーション `loc` へのマイグレーションは `obj.go(loc)` と記述される．リージョンマネージャによって管理されるロケーションの集合をリージョンという．あるリージョンに存在するオブジェクトが他のリージョン中のロケーションへマイグレーションする場合には，関連するリージョンのリージョンマネージャから許可が与えられたときにのみ，そのマイグレーションは可能となる．

3) **リモートオブジェクトの扱い** リモートオブジェクトに対する動作の記述はローカルオブジェクトと区別なく行うことができる．例えば，ローカルオブジェクト `arg` が `loc1` に位置し，リモートロケーション `loc2` に位置するオブジェクト `obj` がメソッド `meth` を持つと仮定する．この時，リモートオブジェクト `obj` のメソッド `meth` の呼び出しは `obj.meth(arg)` と表される．ローカルオブジェクト `arg` の参照がリモートオブジェクト `obj` が位置する `loc2` へ送られ `obj.meth(arg)` が `loc2` で実行された後，実行の制御と残りの計算は `obj.meth(arg)` の計算結果と共に元のロケーション `loc1` へ戻る．

ここで，`loc1` と `loc2` が異なるリージョンに属している場合には，リモートオブジェクトに対する上述のオペレーションは，後述するようにリージョンマネージャによって管理されることになる．

[b] リモートオペレーション構文

構文 S を実行するリモートオペレーション構文は次のように記述される．

do $\{S\}$ **at** (loc)

ここで，ロケーション `loc` はリモートロケーション

であると仮定する．**do** $\{S\}$ **at** (loc) 構文は，ロケーション loc で処理 S を実行し，処理 S が終了した後，実行の制御と残りの計算がこの構文を実行する元のロケーションに戻る．このとき，**do** $\{S\}$ **at** (loc)が実行されるロケーション loc0 が属しているリージョン reg0 が，ロケーション loc が属しているリージョン reg と異なるとき，リモートオペレーションは次のように処理される．

1) reg0 のリージョンマネージャが reg0 からの退出の許可を与え，reg のリージョンマネージャが reg への侵入許可を与え，
2) reg に侵入後，処理 S がロケーション loc で実行され，
3) S の実行後，reg のリージョンマネージャから退出の許可を得て，reg0 のリージョンマネージャから reg0 への侵入の許可が与えられる．

従って，リージョンマネージャによって不正なリモートオペレーションを防ぐことが可能となる．

また，次の構文はリモートリージョンにおける処理を定義する．

do $\{S\}$ **in** (reg)

この構文の意味は，リモートリージョン reg 中の利用可能なロケーションにおいて処理 S を実行するものである．利用可能なロケーションはリージョン reg のリージョンマネージャによって決定される．

Obliq[7] ではリモートオペレーションを実現するために，IP アドレスと処理コードおよび実行環境を送らなければならない．これに比べると，**do** $\{S\}$ **at** (loc) 構文によるリモートオペレーションの記述はより単純化，抽象化されている☆．

[c] オブジェクトのマイグレーション

オブジェクト obj のロケーション loc へのマイグレーションは，go 構文を用いて

```
obj.go(loc)
```

という形で実現される．obj.go(loc) は，obj がロケーション loc へマイグレーション可能であるとき，オブジェクト obj の実体（具体的にはオブジェクトの持つインスタンス変数の値）をロケーション loc へ移動しその後，残りの計算の実行制御が元のロケーションへ戻ることを意味している．

また，オブジェクト obj のリージョン reg へのマイグレーションも次のように記される．

```
obj.go(reg)
```

☆ **do** $\{S\}$ **at** (loc) によるリモートオペレーションは go 文を利用して実現することができるが，処理 S に注目したプログラム作成の容易さと明解さの点から導入した構文である．

この構文は，オブジェクト obj がリージョン reg のリージョンマネージャにより決定されるロケーションにマイグレーションするという意味を持つ．従ってリージョンマネージャはリージョンへ侵入するオブジェクトにロケーションを割り当てる機能を持つ必要がある．

異なるリージョンのロケーションへマイグレーションする場合，現在位置するリージョン及び移動先のリージョンのリージョンマネージャからマイグレーションの許可を得たとき初めてマイグレーションが許される．

[d] 非同期メソッド呼び出し

非同期メソッド呼び出しは，メソッド呼び出しを行った後にその処理の終了を待たずに次の処理へ制御が移る機能である．非同期メソッド呼び出しにより，分散環境下で効率的な並行処理を記述できる．

HORB[8] を参考に以下のような機能を導入する．

名前が *method* である Java メソッドを非同期メソッドとして定義する場合，

*method*_Async

というように，メソッド *method* 名の最後に "_Async" を追加する．非同期メソッド呼び出しを行うときには

*method*_Request

というように，非同期メソッド名の最後を "_Request" に変更し呼び出す．要求した非同期メソッド呼び出しの結果の受信は，

*method*_Receive

として非同期メソッド名の最後を "_Receive" に変更し呼び出すことにより行われる．

[e] リージョンとリージョンマネージャ

リージョン R_{id} はリージョン名 id，リージョン内のロケーションの集合およびリージョンマネージャ RM から構成され以下のように定義される．

$$R_{id} = [id, \{l_1, \ldots, l_m\}]_{RM}$$

リージョンマネージャ RM は，オブジェクトのマイグレーション時のリージョンへの侵入やリージョンからの退出の管理，リモートオペレーションの管理などを行う．リージョンとリージョンマネージャはユーザが定義でき，それらを用いてユーザがセキュリティ設定を行うことができる．4 章ではリージョン及びリージョンマネージャの詳細について述べ，5 章でこれらを用いた例を示す．

[f] failure 機能を用いたエラー回避機能

failure 機能は次の二つの機能から成るものである．

(1) ロケーションにおける計算の停止の検出
(2) 指定したロケーションにおける計算の強制的停止

failure 機能を用いて，ユーザは計算の停止を検出し，

エラーを回避しシステムの信頼性を向上させるプログラムを記述することが可能となる．failure 機能については次の 3 章で詳述する．

3. failure 機能を用いたエラー検出と回避

ネットワーク環境において分散処理を実行する際には，一つのサイトの計算機でトラブルが発生したり通信に障害が発生すると全ての計算が無駄になる可能性がある．サイトにおける failure を検出する failure 検出機能と，failure サイトでの計算を停止させる機能を導入し，システムのエラーを回避させるのが failure 機能である．failure 検出とそのエラー回避は，獲得可能なエラーメッセージと failure のモデル化の方法に依存する．本論文では，分散 Join Calculus[6] の方法を参考に，指定したロケーションでの failure 検出及び計算停止機能を導入する．さらにこの機能をリージョンの failure 検出ならびに計算停止に拡張している．

[A] failure 検出機能

failure 検出機能は，ロケーションの failure 検出を行うものと，リージョンの failure 検出を行うものが存在し，failure 検出がされた場合に他のメソッドが実行されるよう設計されている．

1) ロケーションの failure 検出

ロケーションの failure 検出は次の構文により記述される．

try{ S_1 }**fail** (*loc*){ S_2 }

この構文は，S_1 の実行中にロケーション *loc* の計算が停止した場合には，S_1 の計算を打ち切り S_2 の計算を直ちに開始するという意味を持つ．そうでない場合には S_1 の処理が終了後 **try-fail** 構文に続く処理を行う．

2) リージョンの failure 検出

リージョンの failure 検出は次の構文により記述される．

try{S} **fail**(*reg*) **catch**(*Vector vec*)
cond(E_1){S_1}...**cond**(E_n){S_n}

この構文は次のような意味を持つ．

a) リージョン *reg* 中での計算が停止せずに S の処理が終了した場合，この構文に続く計算を行う．

b) S の実行中にリージョン *reg* に属するロケーションで計算停止した場合は，`java.util.Vector` 型のベクトル変数 *vec* に，リージョン *reg* に属している計算が停止していないロケーションを要素とするベクトルオブジェクトが **catch** 構文により束縛される．`java.util.Vector` クラスには指定したオブジェクトがその要素であるかを調べる contains メソッドやベクトルの要素数を調べる size メソッドなどが定義されている．

c) 次に条件式 E_1 を評価する．この値が `true` であれば S の処理を打ち切り S_1 の処理を直ちに開始する．`false` であれば E_2 を評価し以下同様に E_n と S_n まで評価され続ける．

d) もし E_1 から E_n までの評価値が全て `false` であった場合にはそのまま S が実行される．続行された処理中に再度リージョン *reg* に属する他のロケーションの計算が停止した場合には，*vec* に新しいベクトルオブジェクトが束縛され，以降同様の処理が繰り返される．S の処理が終了した場合は，この構文に続く処理を行う．

次のプログラムは，リージョンの failure 検出構文を用いることにより，リージョンでのエラー検出とその後の処理を記述する例である．

```
try { obj.method(); }
fail(region) catch(Vector vec)
  cond(!vec.contains(importantLoc)) {
    reconstructNetwork();
} cond(vec.size() < Const.THRESHOLD) {
  for (int i=0; i<vec.size(); i++) {
    ((Location)vec.elementAt(i)).go(loc);
  }
}
```

リージョン `region` での failure 検出が行われたとき，まず重要な機能を司るロケーション `importantLoc` で計算停止したか否かの判断を行い，計算停止の時には `reconstructNetwork()` によって新たにネットワークを再構築する．そうでない場合 `region` 中に属している計算が停止していないロケーションの数を `vec.size()` により調べる．`vec.size()` がある閾値 `Const.THRESHOLD` を下回った場合には，生き残っているロケーションも近いうちに計算停止することが考えられるため，他のロケーション `loc` へ移動する．どちらにも当てはまらない場合には，そのまま処理 `obj.method` を実行し続ける．

[B] 計算停止機能

ロケーション `loc` での計算停止は次のように表される．

`loc.halt()`

`halt` メソッドは指定したロケーション `loc` の計算を停止するものである．計算が停止したロケーションからの計算結果は永久に返ってこない．リージョンの計算停止はリージョン中に属しているロケーション全てに対して halt メソッド呼び出しを行うことによって実現される．なお，Region クラスで定義されてい

る getContents メソッドを用いて，リージョン中の全てのロケーションの列挙を得ることができる．

このような計算停止機能は信頼性が高く効率のよいプログラムの記述に役立つ．例えば，あるロケーションで計算を強制終了させ別のロケーションで計算を行なう，というようなプログラムを書くことができる．なお，**halt** メソッドは指定したロケーションの計算を強制停止し，他のロケーションに計算を依頼するためのものであるから，タイムアウトに基づくメソッドと併用することも可能である．

[C] failure 機能を用いたエラー回避

failure 検出機能と計算停止機能により，エラー回避のプログラムを書くことができる．failure 検出構文により，指定したロケーションでの計算の停止が検出可能である．failure が検出されたロケーションから別のロケーションへプログラムの実行を移すことによりエラーを回避することが可能である．

次の例では，処理が行われる Agent 自体をロケーションとして生成し，ロケーション loc を引数として取り，そのロケーションへ go(loc) によってマイグレーションした後に，calc を行いその結果を持って元のロケーションに戻る casa メソッドが定義されている．

```
class Agent extends Location
                    implements Migratable {
  void casa(Location loc) {
    Location returnLoc = this.getLocation();
    this.go(loc);
    this.calc();
    this.go(returnLoc);
  }
  static boolean isDone(Agent agent,Location l){
    try { agent.casa(l); }
    fail(agent) { return false; }
    return true;
  }
  ....
}
```

メソッド isDone は casa メソッドを実行した結果が正常終了したか否かを **try-fail** 構文を用いて調べるメソッドである．次のプログラムは上で定義した Agent を用いた例である．

```
  do {
    Agent agent = new Agent();
    Location loc = locationList.nextLoc();
  } while ( !Agent.isDone(agent, loc) )
```

locationList は信頼性の高い順に並んでいるサーバロケーションのリストであるとする．このプログラムは agent オブジェクトを作り，locationList の中の次のロケーションで casa メソッドが実行される．新しく生成された agent は **try-fail** 構文を用いて実現された isDone メソッドによって正常終了したかどうかチェックを受け，処理が繰り返される．

4. ロケーションとリージョン

ロケーションはオブジェクトの所在位置であり，リージョンはリージョンマネージャによって管理されるロケーションの集合である．これらは，既述のようにリモートオペレーション，オブジェクトのマイグレーション，failure 機能およびユーザ定義のセキュリティ設定にとって重要であり，有用なものである．

4.1 ロケーション

インスタンスメソッドは，メソッドが呼び出されたロケーションで実行される．クラスメソッドは，メソッドが存在するロケーションの Java サーバで実行され，これらのロケーションにオブジェクトは生成される．

あらゆるオブジェクトはあるロケーションに属しており，ロケーションは Location クラスもしくはそのサブクラスのインスタンスとして具体化される．ロケーションはネットワーク接続されているため，ロケーションの集合はグラフ構造を成し，また，ルートロケーションから到達可能な木構造を成すとみなすことができる．

$\mathcal{O}$を全てのオブジェクトの集合とし，$\mathcal{L}$を全てのロケーションの集合とする．この時，$\mathcal{L}$は $\mathcal{O}$のサブセットである．すなわち，$\mathcal{L} \subset \mathcal{O}$．ロケーション $loc_{id} \in \mathcal{L}$ を以下のように識別子 id とそこに属するオブジェクトの集合 $\{o_1, \ldots, o_n\}$ として表す．

$$loc_{id} = [id, \{o_1, \ldots, o_n\}] \ (0 \leq n)$$

ここで，$o_1, \ldots, o_n \in \mathcal{O}$ であり，$n = 0$ の場合はロケーションにオブジェクトが存在しないことを表す．また，o_i はロケーションオブジェクトの可能性もある．次の二つ関数を定義する．

$$obj[loc_{id}] = \{o_1, \ldots, o_n\}$$

$$locname[loc_{id}] = id$$

ネットワークがサーバの集合 $\{nameServer, client, server1, \ldots, serverN\}$ から成っていると考え，それぞれが $loc_{nameServer}, loc_{client}, loc_{server1}, \ldots, loc_{serverN}$ に位置するとした時，ロケーション構造 $locStrct$ は次のように表される．

$$locStrct = \{loc_{nameServer}, loc_{client}, loc_{server1}, \ldots, loc_{serverN}\} \ (0 \leq N)$$

ロケーションはネットワーク接続されており，ロケーションはグラフ構造を成す．したがってロケーション loc_i からロケーション loc_j 間に道が存在するときに loc_i から loc_j に到達可能となる．

新たなロケーション生成は `new Location()` で，またオブジェクトが現在属しているロケーションは `getLocation` メソッドにより得ることができる．また，`getContents` メソッドは現在のロケーションに属するオブジェクトを得るものであり，返り値は `java.util.Enumeration` 型のインスタンスである．ロケーションに関するメソッドを5章に与えた．

4.2 リージョンとリージョンマネージャ

リージョンとリージョンマネージャの基礎的な定義を述べた後，リージョンマネージャの実装法について述べる．

[1] リージョンとリージョンマネージャ

リージョンはリージョンマネージャによって管理されるロケーションの集合である．リージョンは Region クラスもしくはそのサブクラスのインスタンスとして具体化される．全てのリージョンの集合を$\mathcal{R}$とし，リージョン $R_{id} \in \mathcal{R}$ を名前 id とロケーションの集合 $\{loc_1, \ldots, loc_m\}$で定義する．

$$R_{id} = [id, \{loc_1, \ldots, loc_m\}]\ (0 \leq m)$$

ここで，$l_1, \ldots, l_m \in \mathcal{L}$ であり，$m = 0$ の場合は空リージョンであるとする．次の2つの関数を定義する．

$$r\text{-}loc[R_{id}] = \{loc_1, \ldots, loc_m\}$$

$$regname[R_{id}] = id$$

また関数 obj を次のように拡張する．

$$obj[R] = obj[loc_1] \cup \cdots \cup obj[loc_m]$$

リージョン R のリージョンマネージャRM_Rは，侵入許可関数および退出許可関数により定義される．

- 侵入許可関数
  $enterOK_R : \mathcal{O}[R^*] \rightarrow \{\text{true, false}\}$
- 退出許可関数
  $exitOK_R : obj[R] \rightarrow \{\text{true, false}\}$

ここに $\mathcal{O}[R^*]$ はリージョン R の外にあるオブジェクトの集合であり，$obj[R]$ は R を構成するロケーション $l_1, \ldots, l_m$ に属するオブジェクトの集合である．他のリージョン中のオブジェクトへのモバイルオペレーションは次の制約を受ける．

a) リージョン R の外に位置するオブジェクト o^* は，$enterOK_R[o^*]$ が true の場合にはリージョン R への侵入が許可され，そうでない場合には侵入は許可されない．

b) $exitOK_R[o]$ が true であった場合リージョン R 中のオブジェクト o はリージョン R からの退出が許可され，そうでない場合には退出は許可されない．

2つのリージョン R_i と R_j について，

(1) $r\text{-}loc[R_i] \supset r\text{-}loc[R_j]$ のとき，R_j は R_i のサブリージョンという．また，$r\text{-}loc[R_i] \cap r\text{-}loc[R_j] = \emptyset$ のとき，R_i と R_j は分離している，そうでないとき，R_i と R_j は共通部分を持つという．

(2) $loc_i \in r\text{-}loc[R_i]$ と $loc_j \in r\text{-}loc[R_j]$ の間に少くとも一つの道が存在するとき，R_i と R_j は連結しているという．R_i と R_j が連結しているときには，R_i に属するロケーションと R_j に属するロケーションがネットワーク接続されていることになる．

リージョンの集合 $regStruct$ をリージョン構造と呼ぶ．すなわち，

$$regStruct = \{R_{id_1}, \ldots, R_{id_n}\}$$

ネットワーク通信を考えたとき，ロケーション間の道の存在やリージョン間の連結は重要である．

リージョンに関するメソッドを5章に与えた．例えば，リージョンを新たに生成するメソッド，指定したロケーションを自分に属させるメソッド，自分の子リージョンを生成するメソッドなどが Region クラスで定義されている．

[2] リージョンマネージャ

ユーザは各リージョンに対して，次のようなインターフェイスとして定義されるリージョンマネージャを設定することが可能である．

```
1:  public interface RegionManager {
2:    boolean enterOK(Object obj);
3:    boolean exitOK(Object obj);
4:    void enterObj(Object obj);
5:    void exitObj(Object obj);
6:  }
```

ここで，`enterOK` メソッドおよび `exitOK` メソッドはそれぞれ侵入許可関数と退出許可関数に対応している．また，`enterObj` と `exitObj` はそれぞれ，リージョン内へ侵入してきたオブジェクトとリージョン外へ退出するオブジェクトについての動作を記述するためのメソッドである．リージョンマネージャはリージョン中のロケーションに位置するオブジェクトの振る舞いを管理する．すなわち，リージョンを介するオブジェクトのマイグレーションおよびリモートオペレーションはリージョンマネージャによって管理される．

リージョンマネージャはユーザによって上記のように定義できるが，ユーザが特に指定しなかった場合のリージョンマネージャとして RegionManegerDefault

クラスが実装されている．5章で例を与える．

(a) オブジェクトのマイグレーション

ロケーション $l_{id} = [id, \{o_1, \ldots, o_n\}]$ に属するオブジェクト $o_i (i \leq n)$ がロケーション $l_{dst} = [dst, \{o'_1, \ldots, o'_m\}]$ へマイグレーションしたとき，そのロケーション l_{id} と l_{dst} は次のように変更される．

$$l_{id} = [id, \{o_1, \ldots, o_n\} - \{o_i\}],$$
$$l_{dst} = [dst, \{o'_1, \ldots, o'_m, o_i\}]$$

通常，同一リージョン内のロケーション間では，オブジェクトは，自由にマイグレーションすることが許される．リージョン *RegA* 中のオブジェクト *obj* が別のリージョン *RegB* へマイグレーションする時，

(1) $exitOK_{RM_A}(obj)$ と $enterOK_{RM_B}(obj)$ は共に true となり，その後，

(2) リージョンマネージャRM_A と RM_B の管理の下で *obj* のマイグレーションが実行される．

ここで，RM_A と RM_B はそれぞれ，リージョン *RegA* と *RegB* のリージョンマネージャである．

(b) リモートオペレーションの管理

リージョン *RegA* 中のロケーション *loc*1 で実行される次の処理について考える．

do $\{S\}$ **at** (*loc2*)

ここで，ロケーション *loc2* はリージョン *RegB* 中のロケーションであるとする．

まず，処理 S がロケーション *loc2* で実行されるためには，リージョン *RegA* のリージョンマネージャ RM_A から退出許可が与えられ，その後ロケーション *loc2* が存在するリージョン *RegB* のリージョンマネージャRM_B によるへの侵入許可が与えられなくてはならない．ロケーション *loc2* で S が実行終了した後は，RM_B と RM_A から許可を得た後，リージョン *RegA* 中のロケーション *loc*1 で残りの計算を再開する．

(c) リージョンマネージャの設計について

ユーザは個々に設けたリージョンに対してリージョンマネージャを定義し，リージョンマネージャの侵入許可関数と退出許可関数によってリージョンのセキュリティを設定することができる．上述のインターフェイスをユーザが実装することによりリージョンマネージャを定義することができるが，`setRegionManager` メソッドを用いてこれを変更することも可能である．

リージョンとそのリージョンマネージャを階層構造や入れ子構造として構築することによって，重要な情報をより深いレベルにおくことが可能である．従来のセキュリティ設定では侵入時にのみチェックを行うことが一般的であるのに対して，リージョン機構では退出許可関数 $exitOK_{RM}$ によるチェックも可能である．リージョンマネージャによるこのセキュリティバリアはリージョンマネージャがJavaのプログラムとして書かれるため，ハッカーに侵入される可能性がある．しかし，各々のリージョンのアプリケーションプログラムをリージョンマネージャによって隔離することができるから，ハッカーがあるリージョンに侵入しても，必ずしも他のリージョンに侵入することができるとは限らない．なお，mJava/LRの処理系では，ロケーションとリージョンは処理系のロケーション表とリージョン表によって管理されるから，処理系を全てJavaで実現するより安全なものとなっている．

5. ロケーションおよびリージョンによる拡張構文

本研究の目的は，これまでに述べてきたロケーションとリージョンおよびそれらに付随する機能を用いて拡張したJava言語の処理系を作成し，実用に供する点にある。しかし，現時点では，Javaのサブセットであるm Javaに対してロケーションとリージョンを導入したmJava/LRを設計し，その処理系を作成している段階である．この章では，mJava言語の定義を与えた後，mJava/LRに導入されたロケーションとリージョンに関する拡張構文を与える．

5.1 mJava言語

mJava (mini Javaの略) はロケーションとリージョンの機能をJavaに導入する研究のために設計されたJavaのサブセットである．mJava言語は，オブジェクト指向言語として最小限の機能を持つJavaのサブセットであり，他の構文や演算子で実現可能なものは省略するという方針で設計された言語である．mJavaの単純化したシンタクスを図1に示す．ここで，$\overline{A}$ は系列 $A_1, \ldots, A_n$ を表す表記であるとする。

mJava言語で利用可能な基本型は`int`型と`boolean`型のみであり，これらを扱う演算が定義されている．構文はreturn文，if文，for文，while文，局所変数宣言文といったJavaの主要な構文をサポートしている．また，オブジェクト指向として重要なオーバーロード，オーバーライド，インターフェイス，アクセス制御機能は実現している．従って，mJavaはオブジェクト指向言語としての必要最小限の機能を持ったJavaのサブセットである．しかし，システム実現の簡単化のために浮動小数点数，配列，マルチスレッド機能，パッケージ機能，例外機構などは備えていない．

5.2 ロケーションとリージョンによる拡張構文

ロケーションとリージョンに関する構文としては，リモートオペレーションおよびfailure検出などに関

$$
\begin{array}{ll}
P ::= & \overline{KI} \\
KI ::= & K \mid I \\
 & \\
K ::= & \textbf{class } c \textbf{ extends } \mathrm{C} \textbf{ implements } \overline{A} \; \{ \; \overline{KM} \; \} \\
I ::= & \textbf{interface } c \textbf{ extends } \overline{C} \; \{ \; \overline{IM} \; \} \\
KM ::= & \overline{mdf} \; A \; x = E; \mid \overline{mdf} \; A \;\; x(\overline{B} \;\; \overline{y})\{S\} \\
IM ::= & \overline{mdf} \; A \; x = E; \mid \overline{mdf} \;\; A \;\; x(\overline{B} \;\; \overline{y}); \\
 & \\
S ::= & \textbf{return } E; \mid S_1 S_2 \mid E; \mid A \; x = E; \mid \; ; \\
 & \mid \textbf{if } (E) \; S_1 \textbf{ else } S_2 \mid \textbf{while } (E) \; \{ S \} \\
 & \mid \textbf{for } (E_1; E_2; E_3) \; \{ S \} \\
 & \\
E ::= & number \mid \textbf{true} \mid \textbf{false} \mid x \mid c.x \mid E.x \\
 & \mid E_1 = E_2 \mid E_1 \; op \; E_2 \mid E(\overline{E'}) \\
 & \mid \textbf{new } c(\overline{E}) \mid (c)E \\
 & \\
mdf ::= & \textbf{public} \mid \textbf{private} \mid \textbf{protected} \mid \textbf{static} \\
 & \mid \textbf{abstract} \\
op ::= & + \mid - \mid * \mid / \mid \textbf{instanceof} \mid \&\& \mid || \mid ! \mid == \\
 & \mid != \mid < \mid > \mid <= \mid >= \\
A, B ::= & \textbf{int} \mid \textbf{boolean} \mid c, C, \ldots
\end{array}
$$

図 1 mJava のシンタクス
Fig. 1 Syntax of mJava.

する構文と，ロケーションおよびリージョンに関するクラス定義の構文がある．

[1] リモートオペレーションおよび failure 検出に関する構文 mJava/LR では，既述のようにリモートオペレーションや failure 検出の構文が導入されており，図 1 で示した mJava のシンタクスが次のように拡張される．

$$
\begin{array}{l}
S ::= \ldots. \\
\mid \textbf{do} \; \{ \; S \; \} \textbf{ at } (E) \\
\mid \textbf{do} \; \{ \; S \; \} \textbf{ in } (E) \\
\mid \textbf{try} \; \{ \; S \; \} \textbf{ fail } (E) \; \{ \; S \; \} \\
\mid \textbf{try} \; \{ \; S \; \} \textbf{ fail } (E) \\
\qquad \textbf{catch } (Vector \; x) \; \overline{Cond} \\
Cond ::= \textbf{cond } (E) \; \{ \; S \; \}
\end{array}
$$

[2] ロケーションおよびリージョンに関するクラス定義構文ロケーションおよびリージョンに関するクラスと付随するメソッドとして次のようなものが mJava/LR において定義されている．

Object クラス Object クラスで新たに定義されるメソッドは以下のものである．

- **void go(Location loc)**
  オブジェクトが指定されたロケーション loc へマイグレーション可能な場合には，そのオブジェクトの実体をロケーション loc へ移動する．
- **void go(Region reg)**
  オブジェクトが指定されたリージョン reg へマイグレーション可能な場合には，そのオブジェクトの実体をリージョン reg 中のロケーションへ移動する．移動先のロケーションは reg のリージョンマネージャにより決定される．
- **Location getLocation()**
  オブジェクトが現在属しているロケーションオブジェクトを返す．
- **void changeName(Name name)** オブジェクト名を指定された名前 name に変更する．

Location クラス Location クラスで定義されているメソッドは以下のものである．

- **Enumeration getContents()**
  ロケーションに属するオブジェクトの列挙を返す．
- **Enumeration getRegion()** ロケーションが属するリージョンの列挙を返す．
- **Region getDefaultRegion()** ロケーション生成時に自動的に生成されるリージョンを返す．
- **boolean isContain(Object obj)** obj がロケーションに属している場合には `true`，そうでない場合には `false` を返す．
- **void halt()** ロケーションとそれに属しているロケーションの計算を停止させる．

Region クラス Region クラスで定義されているメソッドは以下のものである．

- **static Region makeRegion(Name name, Enumeration locEnum, RegionManager regManager)**
  名前が name でロケーションの集合が locEnum，リージョンマネージャが regManager から成るリージョンを新たに生成し返り値とする．
- **Region makeSubRegion(Name name, Enumeration locEnum, RegionManager regManager)** 名前が name でロケーションの集合が locEnum，リージョンマネージャが regManager により定義される自分のサブリージョンを生成し返り値とする．
- **void addLoc(Location loc)** 指定したロケーション loc を自分に属するロケーションとする．
- **void delLoc(Location loc)** 自分に属しているロケーション loc を取り除く．
- **void setRegionManager(RegionManager regionManager)** 自分のリージョンマネージャを指定された regionManager へ変更する．
- **void addRegion(Region reg)** 自分のリージョンに指定されたリージョン reg をサブリージョンとして追加する．この時 reg のリージョンマネージャはそのまま保持される．
- **Region delRegion(Region reg)** 自分のサブリージョン reg を自分のリージョンから削除し，分離した新しいリージョン reg を返り値とする．

reg のリージョンマネージャは自分のリージョンマネージャのコピーとなる．

- **Enumeration getContents()** 自分のリージョンに属するロケーションの列挙を返す．
- **Enumeration getSubRegion()** 自分の子であるサブリージョンの列挙を返す．

NameServer クラス ネームサーバはサイト間でオブジェクトの参照を共有するのに用いられる．これを定義しているクラスが NameServer クラスであり，定義されているメソッドは以下のものである．

- **void register(String name, Object obj)** ネームサーバに name という名前で obj を登録する．
- **Object search(String name)** ネームサーバに name という名前で登録されているオブジェクトを返り値とする．

System クラス System クラスに新たに加えられたクラス変数は，以下のものである．

- **static Location topLoc** そのサーバのトップロケーション
- **static NameServer nameServer** そのサーバのネームサーバ．ネームサーバサイトのネームサーバオブジェクトの参照を各サーバで共有している．

5.3 リージョンマネージャの例

4.2 節でも述べたようにリージョンマネージャにはリージョン **reg** を指定したマイグレーション **go(reg)** やリモートオペレーション構文**do** $\{S\}$ **in** (reg)が実行されたとき，それらが実行されるロケーションをリージョン **reg** 中から決定・割り当てる機能が要求される．この機構は処理系依存とし，mJava/LR では次のような優先度によりこれを決定する．

(1) 計算停止しているロケーションは候補から除去する．

(2) リージョンがネットワークを介して構成されている場合には，計算機の故障が発生していない計算機中のロケーションを優先する．また，割り当てられているスレッド数が少ない計算機中のロケーションを優先する．

リージョンマネージャはインターフェイスとして定義されており，ユーザがそれを実装するクラスを記述することにより，容易に実現可能である．現在の mJava/LR システムには RegionManager インターフェイスとして RegionManagerDefault クラスが実装されており，ユーザが特に指定しなかった場合のリージョンマネージャとして用いられる．オブジェクトの管理は，

```
public class RegionManagerDatabase
                    implements RegionManager {
  int state;
  Database database;

  public boolean enterOK(Object obj) {
    if  (obj instanceof Man) {
      Man man = (Man)obj;
      int id = man.getId();
      if (this.state == 1) {
        return this.database.
         recognition(id, man.getBirthDay());
      } else if (this.state == 2) {
        return this.database.
         recognition(id, man.getTelNumber());
      } else if (this.state == 3) {
        return this.database.recognition
            (id, man.getDrivingLicenceNo());
      } else  {
        return true;
      }
    } else {
      return false;
    }
  }
  .....
}
```

図 2 RegionManagerDatabase クラスの一部
Fig. 2 RegionManagerDatabase class.

(1) 指定したリージョンの中に属しているロケーションからのマイグレーションならば許可する．

(2) 指定したオブジェクトならばそのマイグレーションを許可する．

という 2 種類を併用することにより行われる．それぞれ侵入・退出毎に指定可能である．`enterOK` メソッドと `exitOK` メソッドではこれらの条件を満たしていれば `true` を，そうでない場合は `false` を返す．また，オブジェクトが侵入，退出するときに実行される `enter` メソッドと `exit` メソッドはその中身を空文として定義している．

図 2 に示す RegionManagerDatabase クラスは RegionManagerDefault クラスを拡張した例である．このリージョンマネージャは侵入してくるオブジェクトに対してまず名前（Id）を要求する．その後自分の状態変数 `state` の値により，オブジェクトの個人情報 Birthday, TelNumber, DrivingLicenceNo を動的に取得する．得られた個人情報は，電算化された個人情報を扱う `database` により名前をキーに認証問い合わせされる．正しく認証されないオブジェクトのマイグレーションは禁止される．この方法は侵入して来る個人を特定することが可能であり，さらに侵入してきたオブジェクトを記憶することも `enter` メソッドを用いて容易に記述可能である．さらに，職業や資格などの個人情報を積極的に用いることにより，例えば特定地域に住む特定の職業に就いている人や資格を有してい

```
class Server extends Location {
 void policy1() {
  ((RegionManagerDafault)defaultRegion
   .getRegionManager()).setExit(defaultRegion);
  ((RegionManagerDafault)defaultRegion
   .getRegionManager()).setEnter(System
   .getTopLoc().getDefaultRegion());
 }
 void policy2() {
  ((RegionManagerDafault)defaultRegion
   .getRegionManager()).setExit(null);
  ((RegionManagerDafault)defaultRegion
   .getRegionManager()).setEnter(defaultRegion);
 }
 void policy3() {
  this.defaultRegion.setRegionManager
   (new RegionManagerDatabase());
 }

 void service(Client client) {
  if (this.isContain(client)) {
   this.trulyService(client);
  } else { }
 }

 private void trulyService(Client client) {
  //     do some service here
 }
}
```

図 3 Server クラス
Fig. 3 Server class.

```
class Client extends Man implements Migratable {
  int input;
  int output;

  void service(Server server,int input) {
    this.input = input;
    this.go(server);
    server.service(this);
  }
}
```

図 4 Client クラス
Fig. 4 Client class.

る人のみにその情報を公開するといったようなきめ細かなセキュリティ設定も簡単に実現可能である.

mJava/LR のエージェント自信は Telescript のように実世界の個人に対応するようなオーソリティを持たないが, RegionManagerDatabase クラスはこのような問題に対処できる管理方法を実現する例である.

5.4 リージョンマネージャを用いたセキュリティ設定の例

4.2 節で述べた方法を用いて, クライアント, サーバ方式にリージョンを導入しそのセキュリティ設定を行うサーバクラス, クライアントクラスの例をそれぞれ図 **3**, 図 **4** に示す.

Server クラスは Location クラスを継承しており, 各ロケーションが属性として持つインスタンス変数 **defaultRegion** によって自動的に生成されるリージョンマネージャを用いてオブジェクト侵入の管理を行う. サーバはオブジェクトに対していくつかのポリシーを持つ. 全てのオブジェクトの侵入を許す **policy1** メソッド, 逆に全てのオブジェクトの侵入を禁止する **policy2** メソッドは RegionManagerDefault クラスのメソッドを用いて容易に実現可能である. また, **policy3** メソッドは図 2 に示した RegionManagerDatabase を用いて個人認証を動的に行う設定である. サービスを提供する **service** メソッドではクライアントがサーバ中に存在するか否かを調べ, 存在した場合にのみ実際のサービス **truelyService** を提供する.

Client クラスは RegionManagerDatabase による個人情報の要求に対する返信メソッドが定義されている Man クラスを継承する. サーバからサービスを受けるために **service** メソッドは指定されたサーバへとマイグレーションしてからサービスを受ける記述を行う.

これを用いて次のプログラムをあるクラスで実行することを考える.

```
1: public static void main(String arg[]) {
2:    Server s = new Server();
3:    Client c = new Client();
4:    s.policy1();    c.service(s, 1);
5:    s.policy2();    c.service(s, 2);
6:    s.policy3();    c.service(s, 3);
7: }
```

サーバが全てのマイグレーションを許すポリシーの時には, クライアントはサーバへ侵入が許されるのでサービスを受けることが可能である (4 行目). 逆にマイグレーションを禁止した場合にはサービスを受けることができない (5 行目). データベースを用いた個人情報の認証の際には 3 行目で生成されたクライアントが正しい個人情報を有しておりデータベースに認証されたときに初めてサービスが提供される (6 行目).

このようにリージョンマネージャを用いることにより, ユーザが意図するセキュリティ設定を容易に実現することが可能である.

また, 引数としてセキュリティに関わるリージョンオブジェクトの参照等を渡すことが可能である. 従って **setRegionManager** 等の重要なメソッドに関しては呼び出し元を確認するというような動作をオーバーライドによって記述するなど, ユーザ側で対処する必要がある.

6. mJava/LR の概要

5 章で述べた mJava/LR 言語の処理系を実現するアプローチとして, mJava/LR プログラムを Java 言語のプログラムへ変換するトランスレータを作成する方

```
 1: public class Fib {
 2:   int n;
 3:   Fib(int n) {
 4:     this.n = n;
 5:   }
 6:   int fib() {
 7:     if (this.n < 2)
 8:       return n;
 9:     else
10:       return new Fib(n-1).fib() +
11:         new Fib(n-2).fib();
12:   }
13:   static int fib_static(int n) {
14:     if (n < 2)
15:       return n;
16:     else return fib_static(n-1) +
17:            fib_static(n-2);
18:   }
19: }
```

図5 フィボナッチ数を求めるベンチマーク
Fig. 5 Program of computing the Fibonacci sequence.

法が考えられる．しかし，将来的に計算機の failure 検出を実現するため OS レベルでの処理を導入して failure 検出機能をより実用的なものとすること，ロケーションおよびリージョンを処理系内で直接扱うことによりリージョンマネージャによる安全性の向上や処理系の効率化が可能であるといった観点から，mJava の処理系を作成し，それを基に mJava/LR の処理系を作成する方針を取った．

6.1 mJava の処理系の概要

mJava の処理系は Java の処理系に準拠する形で実現されている．コンパイラでは mJava のプログラムを Java Virtual Machine(JVM)[2] で用いられるバイトコードのサブセットに変換する．実行系 mJVM（mini JVM の略）は JVM の仕様に準拠している．mJVM はスタックマシンであり，スレッド領域，実行時のデータ領域であるヒープ，プログラム情報を格納するメソッドエリアを備えている．試作した処理系を図**5**に示したプログラムを用いて評価した実験結果を表**1**に示した．この例では，フィボナッチ数を求めるメソッドを，Garbage Collection(GC) を含む全体の処理を行う `fib` メソッドと，GC を行わない計算処理のみを行う `fib_static` メソッドとして表している．

試作したコンパイラではこのようなシンプルなプログラムならば Java Development Kit(JDK) のコンパイラと同程度の最適化が行われている．図5のフィボナッチ数を求めるプログラムについて $n = 30$ として評価を行った場合，mJVM の実行速度は JDK1.1.7 の JVM での実行と比較して，表1に見られるように，2.3 ～ 2.7 倍遅くなっている．この速度差は，主に実行系を実現する言語による差と考えられる．JDK1.1.7 の JVM はアセンブリ言語で実現されているのに対し，mJVM は C 言語で実現されている．なお，mJVM

表1 ベンチマーク評価結果
Table 1 Execution of Fibonacci sequence program.

メソッド名	実行系	コンパイラ	実行時間 [s]
fib_static(30)	mJVM	mJava	5.1
	mJVM	JDK	5.1
	JDK1.1.7	mJava	1.9
	JDK1.1.7	JDK	1.9
	JDK1.0.2	mJava	3.0
	JDK1.0.2	JDK	3.0
fib(30)	mJVM	mJava	11.6
	mJVM	JDK	11.6
	JDK1.1.7	mJava	5.0
	JDK1.1.7	JDK	5.0
	JDK1.0.2	mJava	12.3
	JDK1.0.2	JDK	12.3

と同様に C 言語で実現された JDK1.0.2 の場合には，mJVM と同程度の速度となっている．

6.2 mJava/LR の処理系の概要

mJava/LR の処理系は mJava の処理系にロケーションとリージョンおよび付随する諸機能を備えたシステムである．mJava/LR の処理系は mJava/LR のプログラムをバイトコードへ変換するコンパイラと，バイトコードを解釈実行する実行系から成る．まず，オブジェクトの内部表現について説明してから処理系について説明する．

[a] オブジェクトの内部表現

mJava/LR のオブジェクトの内部表現はオブジェクトの識別子，カレントロケーション *c-loc*，ロケーションヒストリ *lochist* およびサブオブジェクト・リスト *sobj-list* のリストからなる．

$$obj_{id} = (id, c\text{-}loc, lochist, sobj\text{-}list)$$

ここで，ロケーションヒストリ $lochist = (loc_n : id_n, \ldots, loc_1 : id_1)$ はそのオブジェクトが属していたロケーションのリストである．また，ロケーションヒストリが *lochist* であるオブジェクト obj_{id} がロケーション loc' へマイグレーションした時ロケーションヒストリは $[loc' : id] + lochist$ へ変化する．ロケーションヒストリによりオブジェクトは過去に属していたロケーションでの名前を知ることができる．

[b] 名前の管理

全てのオブジェクトは名前を有する．ロケーションオブジェクト，リージョンオブジェクトがもつ名前をそれぞれロケーション名，リージョン名と呼ぶ．名前は Name クラスのインスタンスとして表されるため，名前を用いた参照の受け渡しなども可能である．それぞれの名前は次のように管理される．

1) オブジェクト名の管理

オブジェクトの名前変更や，オブジェクトの移動時

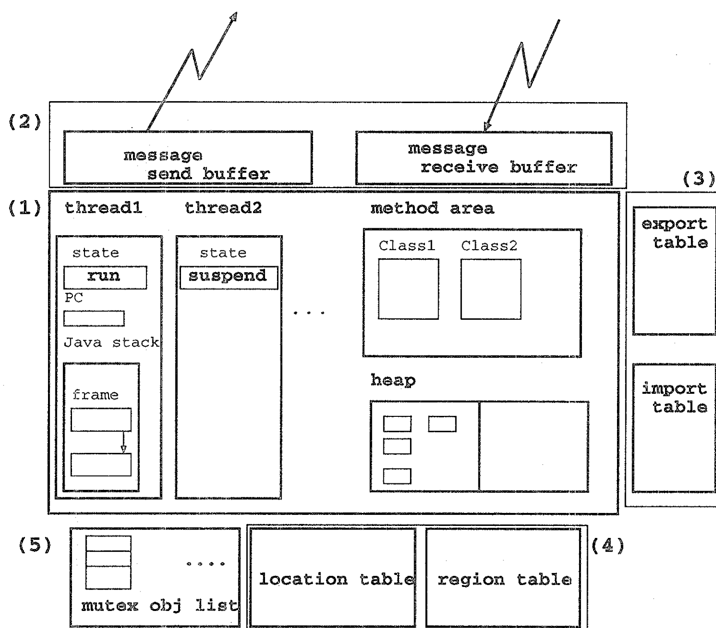


図 6 mJava/LR VM の構成
Fig. 6 mJava/LR VM system.

にロケーション内で同一のオブジェクト名を持つ可能性が存在する．名前の変更の不許可，許可する場合に名前衝突の回避の管理を行う．

2） ロケーション名の管理

リージョンに対してロケーションを追加する場合やロケーションオブジェクトがマイグレーションする時に，リージョン内で同一のロケーション名を持つ可能性が存在する．このような場合には，追加される方のロケーション名が名前衝突を起こす場合には適当な名前へと変更する．

3） リージョン名の管理

リージョン名については，一意に定められるように，リージョン生成時に指定されたリージョン名に対してシステム内部で自動的にプログラムが実行されている URL およびサーバ番号の追加を行う．ロケーションが複数のサーバに位置している場合には，全てのサーバで同一のリージョン名になるようエイリアスをかける．

このようなリージョン・ロケーションを単位とする名前管理を用いることによりアプリケーションの記述に有用なディレクトリサービスは比較的容易に実現できると考えられる．

[c] mJava/LR コンパイラ

mJava/LR コンパイラは mJava のコンパイラに分散処理の立場からの拡張を行った上で，failure 検出構文，リモートオペレーション構文などについて，拡張したVMコードを割り当てることにより実現されている．拡張したVMコードは下記のものである．

- **do_at** リモートオペレーションの開始および終了を表す． 開始および終了はオペランドにより区別される．
- **go_obj** ロケーションオブジェクト以外のオブジェクトのマイグレーションを行う．
- **go_loc** ロケーションオブジェクトのマイグレーションを行う．
- **invokevirtual_async_req** 非同期メソッド呼び出し要求を行う．
- **invokevirtual_async_recv** 非同期メソッド呼び出しの結果を受信する．
- **failure_detect** ロケーションを fail detect list へ登録する．
- **halt** ロケーションの計算を停止させる．

[d] mJava/LR 実行系 mJava/LR の実行系は mJVM に対して分散処理や場所概念に付随する機能を実現するための拡張がなされており，その構成は図 6 のようになっている．

(1) mJVM

上述した mJVM に対し，新たに拡張された VM コードの解釈実行部が加わっている．また，既存の VM コードに関しても分散処理実現のためにその実行を，JVM の仕様を拡張する形で行っている．例えばオブジェクトのインスタンス変数値を取り出す

getfield オペレーションの場合，対象となるオブジェクトがリモートに存在する場合にはそのロケーションへ要求を送り，送られてきた結果をスタックに積んでいる．

(2) メッセージ送受信部

他のサーバからのメッセージは message receive buffer に蓄えられ，必要に応じてそれらが取り出され処理がなされる．他のサーバへメッセージを送る際には message send buffer にメッセージをパックした後に目的のサーバへと送られる．メッセージのパックや，メッセージの送受信は PVM(Parallel Virtual Machine)[9] のライブラリを用いて実現されている．

(3) **Export Table, Import Table**

輸出表（Export Table）はサーバ中のオブジェクトの参照を他のサーバへ送信する際に利用されるものである．ネットワーク参照はサーバ番号とオブジェクトが輸出表に登録されている番号の組で表される．輸入表（Import Table）は過去に他のサーバから送信されたネットワーク参照を登録するものである．これらを用いてネットワーク参照を実現している．

(4) **Location Table, Region Table**

ロケーションおよびリージョンはそれぞれロケーション表（Location Table）およびリージョン表（Region Table）によって管理される．ロケーション表は，属しているオブジェクトの集合，ロケーションの親ロケーション名，failure 検出フラグと計算停止フラグが登録されている．リージョン表には，属しているロケーションの集合，リージョンマネージャ表への参照，親リージョン，サブリージョンと failure 検出フラグならびに計算停止フラグが登録されている．これにより，ロケーションやリージョンに関する処理が処理系内部で実行可能となるため，それらの実行効率を上げることが可能である．リージョンマネージャによる管理もリージョン表を通して処理系内部で行われるため，ハッカーが一つのリージョンに侵入しても他のリージョンへも容易に侵入することはできない．

(5) **Mutex object list**

オブジェクトへ他のサーバからのアクセスの依頼時に，そのオブジェクトがロックされている場合がある．この時，ロックの解除待ち状態となる要求のリストが mutex object list である．

オブジェクトのマイグレーションは拡張 VM コード go_obj，go_loc で行われるが，このオペレーションでは，シリアライズしたオブジェクトをデシリアライズし，元のオブジェクトの参照を移動先のオブジェクトへとエイリアスしている．

mJava/LR の VM の処理能力をフィボナッチ数列を求めるプログラムを用いて測定すると mJVM での実行に比べ 1.41 ～ 1.45 倍程度実行速度が低下することが知られている．これは，mJava/LR では mJava に存在しない排他処理，failure 機能，分散オブジェクトなどを実現しているコストが加わっているためである．

7. おわりに

本論では，Java のサブセット mJava にロケーションとリージョンに基づく構文を導入した mJava/LR を提案し，これを用いてモバイルオペレーションの記述とユーザによるセキュリティ設定が行えることを示した．このまた，mJava と mJava/LR の試作処理系の概要についても報告した．リージョンとリージョンマネージャを用いて，ユーザ定義のセキュリティ設定が行える点が mJava/LR の特徴である．これに加えて，ロケーションとリージョンにおける failure 検出と計算停止の機能も取り入れ，ネットワーク障害を回避する機能も備えた言語として mJava/LR を設計した．

ロケーションとリージョンは，Java 言語にのみ対応した固有のものでなく，他のインターネット言語にも利用可能なものである．本研究のロケーションとリージョンおよびリージョンマネージャは，筆者の1人(伊藤)が行っている時間と場所の概念を備えたインターネット言語のモデルの研究の一部を Java 言語を対象として具体化したものである．筆者等の研究室において研究を始めている関数型インターネット言語のセキュリティ設定やエラー回避機能の実現への利用も可能である．セキュリティ設定の観点からは，電子署名などの既存の手法とは直交する概念であり，幅広い応用が可能であると考える．

mJava/LR のために設計されたロケーションとリージョンの拡張構文は，Java をフルセット化した場合に現われるマルチスレッド機構，パッケージ機能，例外処理機能との関連からは，mJava/LR の実現法は不十分で諸種の問題が発生する可能性がある．これらについては今後検討の必要がある．ロケーションの移動とそれに伴うロケーション名の consistent な書き換えについて論じる必要があるが，その詳細は紙数の都合もあり省略したので，これについては別の機会に報告したい．

ロケーションを導入した言語としてよく知られたも

のとして Telescript があるが，ロケーションをグループ化しリージョンとして把えセキュリティ設定に用いることは筆者等が知る限りでは行われていない．本研究のリージョンとリージョンマネジャの考え方は Telescript のセキュリティの向上にも用いることができる方法であると考える．また，Java に基づく様々なモバイル言語[10]にも本研究のようなリージョン機構は見受けられない．

failure 機能のための構文は，Join Calculus の考え方をそのまま利用したと言えるものである．簡潔であるから一般ユーザが利用しやすい反面，各種のシステム障害がシステム固有のものであることが多いことを考えると実用的にはかえって利用しにくい場合も予想される．これについては，現在の failure 機能を具体的なシステムの障害に対応できるように実現した上で改良や拡張を検討したい．

現在の mJava/LR は，mJava という Java の小さなサブセットを基にした実験言語であり，ロケーションとリージョンに関する構文を導入した本格的な Java/LR のためのプロトタイプである．ロケーションとリージョンの導入に伴って拡張されたバイトコード検証器の実装なども含めて mJava/LR の処理系を完成度の高いものとし，実際的なインターネット環境においてリージョンマネージャを用いたセキュリティ設定や failure 機能によるシステム障害の回避の実験を行える実用的なシステムにすることを当面の目標としている．これを目指して，言語機能の拡張，処理系の実行速度やサイト間の通信応答の改善，failure 検出のためシステム障害信号を扱う機能の実現などを取り入れた mJava/LR 処理系の改良が筆者の1人 (渡辺) によって進められている．

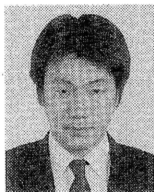
実用的 Java/LR の処理系の作成，ロケーションとリージョンおよび付随する機能を他のインターネット言語に適用すること，Java/LR に時間に関する構文を取り入れ実時間対応の言語と処理系にすること，電子署名とリージョンマネージャによるセキュリティ設定を組み合わせた高セキュリティシステムの実験など多くの課題が今後に残されている．

謝辞 本論文は，1999 年 3 月 24 日に開催された情報処理学会プログラミング研究会において配布の資料に，発表時における討論と研究会論文誌の査読報告を基に改訂を加えたものである．研究会における討論に参加頂き有益な意見やコメントを頂いた方々に謝意を表します．また，研究会論文誌の査読報告では，関連文献を含め，懇切なコメントや指摘を頂いた．査読者の方々に深謝の意を表します．

参考文献

1) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley Publishing (1996).
2) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley Publishing (1996).
3) General-Magic: Telescript 言語入門，アスキー出版局 (1996).
4) Thorn, T.: Programming Languages for Mobile Code, *ACM Computing Surveys*, Vol. 29, No. 3, pp. 21–30 (1997).
5) Sekigchi, T. and Yonezawa, A.: A Calculus with Code Mobility, *Proceeding of Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, Chapman & Hall, pp. 21–36 (1997).
6) Fourene, C., Gonthier, G., Lévy, J. J., Maranget, L. and Rémy, D.: A calculus of mobile agents, *Proceedings of the 7th International Conference on Concurrency Theory* (Montanari, U. and Sassone, V.(eds.)), LNCS, No. 1119, Springer-Verlag (1996).
7) Cardelli, L.: A language with distributed scope, *Computing Systems*, Vol. 8, No. 1, pp. 27–59 (1995).
8) Hirano, S.: HORB: Distributed Execution of Java Programs, *Worldwide Computing and Its Applications*, LNCS, No.1274, Springer-Verlag, pp. 29–42 (1997).
9) Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.: *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press (1994).
10) Kiniry, J. and Zimmerman, D.: A Hands-on Look at Java Mobile Agents, *IEEE Internet Computing*, Vol. 1, No. 4, pp. 21–30 (1997).
11) Stamos, J. W. and Gifford, D. K.: Remote Evaluation, *ACM Transactions on Programming Languages and Systems*, Vol. 12, pp. 537–565 (1990).

(平成 11 年 3 月 10 日受付)
(平成 11 年 4 月 28 日採録)



渡辺 昌寛（学生会員）

1974年生．1999年東北大学情報科学研究科情報基礎科学専攻博士前期課程終了．同年東北大学情報科学研究科情報基礎科学専攻博士後期課程進学．

伊藤 貴康（正会員）

1940年生．1962年京都大学工学部電気工学科卒業．スタンフォード大学コンピュータサイエンス学科および人工知能プロジェクト研究助手，三菱電機中研を経て1978年から東北大学．現職，東北大学情報科学研究科教授．工学博士．本会理事，東北支部長などを歴任．現在，Information and ComputationのEditor，Higher-Order and Symbolic ComputationのAssociate Editor，IFIP TC1 memberなど．専攻分野，ソフトウェア基礎科学．日本ソフトウェア科学会，電子情報通信学会，人工知能学会，ACM各会員．