TSSI基盤技術研修コース
— プログラミング言語 — ＃3

久野　靖*

2004.5.7

1　はじめに

1.1　前回のアンケートを拝見して…

□ オブジェクト指向についてかなり時間を掛けてやったので、それなりに役立ったという人が多くてよかったです。

□ 例題を提示したときもっと時間を取ってくれないと理解し切らないという意見がありました。どうしても時間が限られていますので…

1.2　今回の内容

□ 前回までに前提としてきた言語…←(Java)

- 「強い型の」「クラス方式」オブジェクト指向言語
- 単独でアプリケーションを構成する

□ それと違う言語も取り上げておきたい…←(JavaScript)

- 「弱い型の」「プロトタイプ方式」オブジェクト指向言語
- 組み込み型処理系←ブラウザに組み込まれてブラウザを制御

□ オブジェクト指向だけで十分かどうか?　→ AOP のイントロ

- ここは今年入れた内容なのでこなれてないと思いますが…

□ 前回までに取り上げられなかった言語関係の話題

- オブジェクト指向言語の利用技術
- 並列と分散

□ どのみち時間が足りないと思うのでできる範囲でやります。

*筑波大学大学院経営システム科学専攻

2　JavaScript - 弱い型のOOPL

□ スクリプト言語

□ 弱い型のオブジェクト指向言語

□ プロトタイプ方式

□ 組み込み型処理系

ftp://ftp.ascii.co.jp/pub/my-unix/javascript/

2.1　スクリプト言語

□ って、聞いたことありますか?　何を意味すると思う?

□ たとえば…

- (答1) コマンドを並べたもの←シェルスクリプト?
- (答2) いい加減な/適当な言語←そんなの目標にするか?
- (答3) インタプリタ方式の言語← Lisp なんかもそうなの?
- (答4) ささっと書いて動かせる言語←コンパクト、インタプリタ
- (答5) くっつける (glue) 言語←組み込み型

□ スクリプト言語世代説

- 第1世代→シェルスクリプト、Awk →コンパクト、適当に書ける
- 第2世代→ Perl →それ1つで結構何でも書ける (汎用)、自立した言語。but キタナイ
- 第3世代→ Python、Ruby →かなり美しい、オブジェクト指向
- JavaScript も第3世代でイイと思うが…

2.2　JavaScript の由来

□ Netscape Navigator 2 →当時の「最先端の」ブラウザ

- Netscape 社はブラウザにスクリプト言語を組み込んで様々なことができるようにしようとした

- 言語は当初「LiveScript」という名前で開発→おりからの Java ブーム→ Sun にお金を払って「JavaScript」にした
- 文法はできるだけ Java そっくりにしたが、言語としてはまったく別の言語。よく JavaScript と Java を混同している人がいるので迷惑
- ブラウザ組み込みのスクリプト言語として「事実上の標準」
- MSIE もこれと互換の「JScript」を搭載

□ 標準化の必要性→ECMA(欧州情報通信システム標準化機構) による標準化→ ECMA-262 規格として成立

- その後、ブラウザ組み込みだけでなく Flash のスクリプト言語 (AcitonScript)、MS ASP(Active Server Page) のスクリプト言語としても採用
- ECMA によるコア部分は共通だが組み込まれている追加機能はそれぞれの環境で異なる

□ 以下では (当然) ブラウザ組み込みの JavaScript を扱う (どこにでもあって簡単に試せる)

2.3 JavaScript 入門

□ HTML の説明はしませんので…

□ JavaScript コードを書ける場所…HTML ファイル中の次の 3 箇所

□ その 1: script 要素 (埋め込み、別ファイル)

```
<script type="text/javascript">
コード...
</script>

<script type="text/javascript" src="ファイル">
</script>
```

- この場合、コード中で「document.write(...)」を実行して出力したものは HTML 中のその場所 (script 要素のある場所) に埋め込まれる

```
<pre><script type="text/javascript">
document.writeln(' 乱数: ' + Math.random());
</script></pre>
```

□ その 2: HTML タグの「onmouseover="..."」(マウスが上に乗ったら〜) などの指定 (イベントハンドラ) の中

- 長いものは書きづらいのでよそに関数を定義して呼ぶことが多い

```
<p onmouseover="window.alert('　乗った ね')">ここ</p>
```

□ その 3: URI として「javascript: コード」という形で書く

- コードが返した値を文字列に変換し、それを URI として利用
- リンク等に使う場合、返した値が undefined なら動作なしになる

```
<p><a href="javascript: window.alert(' 選択')">
ここ</a></p>
```

□ まとめると…

- 常に実行するもの (関数定義、ページ中への埋め込み) → script 要素
- ユーザの操作に対応するもの→イベントハンドラ、javascript URI

```
<html>
<head>
<title>文書のタイトル…</title>
<script type="text/javascript">
  常に実行するコード (関数定義等)
  ...
</script>
</head>
<body>
ページ内容…
<script type="text/javascript">
  document.write() を使うようなコード…
</script>
... onclick="..." ...
</html>
```

2.4 JavaScript の特徴

□ 制御構造の構文は Java にソックリ

- while、for、do-while、if、try-catch
- 文字列との連結の「+」も同じ

```
var x1 = 1, x2 = 1;
while(x1 < 50) {
  document.writeln(x1);
  var z = x1+x2; x1 = x2; x2 = z;
}
```

□ オブジェクト指向言語

```
o = new Object();
document.writeln("...");
```

□ 弱い型→型宣言はない、変数に型がない

```
var x = "ABC", y = 1;
```

- 文字列/文字の区別はなく「'x'」でも「"x"」でも同じこと

□ 「値」と「オブジェクト」の 2 種類がある (Java と同じ)

□ 「値」は数値型 (1 つだけ)、論理値型 (Java と同じ)、文字列型 (文字型も兼ねる。Java では文字型は値、文字列はオブジェクト)

□ 3 種類の「値」それぞれに対応するオブジェクト種別もある (Java と同じ)

□ 値に対してメソッドを呼ぼうとすると自動的にオブジェクトに

```
var n = 3.141592;
document.writeln(n.toExponential(4));
```

□ 関数 (function) がある、関数もオブジェクト
□ 関数リテラルがある。下の 2 つは同じこと

```
function add(x, y) { return x+y; }
var sub = function(x, y) { return x-y; }
document.writeln(add(3, 5) + ', ' + sub(3, 5));
```

□ オブジェクトはすなわち連想配列 (任意の型の値を添字として取れる配列)
□ オブジェクトのプロパティ(フィールド) は連想配列の要素と同じ
□ 一般に「x.y」と「x['y']」は同じものを指す

```
var x = new Object();
x['a'] = 10; x['bc'] = 11; x[100] = 12;
x.b = 13; x.de = 14;
for(var i in x) document.writeln('x['+i+'] == '+x[i]);
```

2.5 JavaScript のオブジェクト指向機能

□ オブジェクトのプロパティとして関数を入れる→メソッド
□ メソッドの中ではオブジェクトを「this」で参照できる

```
var o = new Object();
o.count = 100;
o.show = function(msg) {
  document.writeln(msg + this.count);
}
o.show('My number is: ');
```

□ コンストラクタも実はただの関数だが、「new 関数名 (...)」の形で呼び出すと中で this が使える

```
function Counter(n) {
  this.count = n;
  this.add = function(n) { this.count += n; }
  this.getCount = function() { return this.count; }
}
var c1 = new Counter(3), c2 = new Counter(5);
c1.add(2); c2.add(2);
document.writeln(c1.getCount() + ', ' +
                 c2.getCount());
```

□ これで Java と遜色なくオブジェクト指向…だと思う?
□ ちゃんとあるもの

- オブジェクト、インスタンス変数 (プロパティ)、メソッド
- 同種のオブジェクト、コンストラクタ←かなり ad hoc
- メッセージ送信記法
- 動的分配←型がないので簡単

□ 足りないと思うもの…

- クラス→クラスは必要なのか? (単なる構文単位?)
- カプセル化→確かに保護は「全然」ない→スクリプトだから? (ブラウザと組み合わせた場合、ブラウザ側での保護はアリ)
- 継承 (のようなもの) →以下で説明

2.6 プロトタイプ方式オブジェクト指向言語

□ オブジェクトの種類 (形) を表す (定義する) やり方→おもに 2 通り

- 「こういう種類のオブジェクトはこういう形」という定義 (==クラス) を書く→クラス方式
- 必要なプロパティ、メソッドを持つオブジェクトをまず作り、あとはそれをコピーする (概念的には) →プロトタイプ方式
- 実際には全部コピーすると領域が無駄なので、プロトタイプ (親) へのポインタを持っておき、「自分が持っていないものは親が持っているものを使う」ようにする

□ JavaScript ではどうなっているか…

- 「new 関数 (...)」によってオブジェクトが作成されるときに、
- その「関数」に prototype というプロパティが定義されていると、
- その値が作成されたオブジェクトの「親」としてセットされる
- (実際には関数を作ると prototype プロパティには「new Object()」の結果が自動的にセットされる→そこに色々設定すればよい)

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
Counter.prototype.getCount =
  function() { return this.count; }
var c1 = new Counter(3), c2 = new Counter(5);
c1.add(2); c2.add(2);
document.writeln(c1.getCount() + ', ' +
                 c2.getCount());
```

□ プロトタイプ方式の特徴…

- いつでも親にメソッドを追加したりしていじることができる (単なるオブジェクトだから)
- 場合によっては親を指しているポインタをとり替えることで、オブジェクトをまったく別物に「変身」させられる (JavaScript では不許可、ただし一部の実装ではこれを許していた)

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
```

```
Counter.prototype.getCount =
  function() { return this.count; }
var c1 = new Counter(3); c1.add(4);
Counter.prototype.sub =
  function(n) { this.count -= n; }
c1.sub(2);
document.writeln(c1.getCount());
```

□ 継承みたいなのはどうするの?

- 親 (プロトタイプ) を探して見つからない場合はさらにその親を探しに行く。
- 例: Counter オブジェクトを継承してメソッド sub を追加するには…

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
Counter.prototype.getCount =
  function() { return this.count; }
function ExCounter(n) { this.count = n; }
ExCounter.prototype = new Counter(0);
ExCounter.prototype.sub =
  function(n) { this.count -= n; }
var c1 = new ExCounter(7);
c1.add(4); c1.sub(2);
document.writeln(c1.getCount());
```

□ プロトタイプ方式の特徴をまとめると…

- 実行系のデザインはわりとシンプルにできる
- コンパイラよりインタプリタ向き。動的
- その分、何をやっているのか分かりにくい
- (きちっと構文を作って行くとクラス方式みたいに…)

2.7 組み込み型スクリプト

□ スクリプト言語の 1 つの目的→組み込み型処理系

- 大きなアプリケーションがあったとして、それをカスタマイズすることを考える→どうしますか?

□ 沢山パラメタがあって、それを設定して調整すればいいか?

□ パラメタは静的

- 「こういう場合はこう」というのが書きにくい
- 「動き」はつけにくい

□ →パラメタを設定する代わりに「プログラム」を設定する

- →そのプログラムが動いてさまざまな動作をすればよい
- →組み込み型スクリプト
- (例は既に出て来た…「マウスが乗ったら〜をする」)

□ 組み込み型スクリプトに使われる言語…

- tcl →もともとこのような用途のために作られた。しかし言語仕様があまり美しくないので普及はいまいち (tcl/tk としてだけ有名)
- scheme →もともとは Lisp 系の汎用言語だが、コンパクトな処理系が作れるので組み込みスクリプト用としても使われるように。gwm(ウィンドウマネージャ)、gimp(画像加工ソフト)
- ECMA-262(JavaScript) →元はブラウザ用の組み込みスクリプト言語だが、他の用途にも進出 (Flash ActionScript など)

□ JavaScript の場合、オブジェクト指向ならではの利点→具体的には?

□ アプリケーション自体および、アプリケーションが扱うデータ→オブジェクトであると考えることができる

- オブジェクトを自由に操作するには、オブジェクト指向言語がよい (アタリマエ)
- これまでと違うところ→オブジェクトは既に大量に (アプリやそのデータとして) 存在している→既存のオブジェクトを操るのが主目的
- 従来のプログラムでは自前でオブジェクトを設計したり生成したりしてからそれを使っていた→大きな変化 (どんな?)

□ できあいのオブジェクトを操作する場合の考慮点

- 自分が設計したデータ構造でないのでよく分からない
- 向こう側はチェックして例外を投げる側、こちらはだましだまし使う側
- きちんと型を書くのが煩雑 (しかし型検査はあった方がよいと思うが…)

□ 具体例は次の DOM で

2.8 Document Object Model (DOM)

□ DOM →ブラウザ内のページをオブジェクトの組み合わせで表現

- W3C による標準化→ http://www.w3.org/DOM/
- 現在のブラウザでは DOM level 2(DOM2) が中心
- DOM2 の標準は Core、Views、Events、Style、Traversal、HTML と分かれている
- 以下では例題とともに簡単に説明

□ DOM2 では文書全体は Node オブジェクトの木構造として表される

- Node オブジェクトのプロパティ/メソッドを用いて木構造を自由に変更できる

```
<script type="text/javascript">
function rotate() {
  var n = document.body.firstChild;
  document.body.removeChild(n);
  document.body.appendChild(n);
}
</script>
<button onclick="rotate()">Rotate</button>
```

□ HTML 要素に対応するノードのスタイル→その Node オブジェクトの style プロパティに値をつけることで色々操作できる

- 位置の変更→ CSS の位置指定機能で実現できる

```
<script type="text/javascript">
var xpos = 500, ypos = 400;
function change(dx, dy, col) {
  var n = document.getElementById('x1');
  n.style.backgroundColor = col;
  n.style.left = (xpos += dx) + 'px';
  n.style.top = (ypos += dy) + 'px';
}
</script>
<button onclick="change(-10,0,'yellow')">B1</button>
<button onclick="change(0,-10,'purple')">B2</button>
```

□ その他できること…

- テキストノードの中の文字列を操作できる (挿入、削除、…)
- マウスイベント、キーイベントを受け止めて処理できる
- フォーム部品の内容 (値など) を操作できる (わりと昔からある機能)

□ 結局、DOM で何ができるかというと…

- ブラウザが表示している内容→内部のデータ構造が対応
- データ構造 (オブジェクト) を任意に操作→自由に内容が変更できる
- →究極のカスタマイズ (?)

2.9 本節のまとめ

□ スクリプト言語→ささっと書ける言語、glue 言語

□ JavaScript →ブラウザ内蔵むけスクリプト言語が発端

- 弱い型の言語
- プロトタイプ方式のオブジェクト指向言語

□ 組み込み型スクリプト

- ブラウザ組み込み→ブラウザの制御、DOM によるページ内容の制御

3 AOP — Aspect-Oriented Programming

□ オブジェクト指向 (OO、Object-Orientation) は確かに問題を「人間に分かりやすく構造化」する手段を提供してくれた (と思う)。

□ しかし、どのようにうまく「構造化」しても、その構造化から「洩れている」側面が必ず存在する→横断的側面 (Croscutting Concern、 XC)

□ OO が定着した現在では、この XC をうまく取り扱う方法が言語研究者の関心時となっている→ Aspect-Oriented Programming(AOP)

- AOP によって OO だけでは扱えない部分をうまくカバーした言語が作れる (はずだし、実際そういう言語を提案している) → AOP 派の主張

□ 警告! 以下で紹介するのは AOP 派の主張であり、それに「説得」される必要は必ずしもありません。

- ある言語 X が 99%のニーズをうまく満たしているなら残り 1%をカバーするため言語を 2 倍複雑にすることは引き合わない。
- しかしこれが「80%満たしている」のと「1.2 倍複雑」だったら引き合うのかも知れない。
- なお、いつまでたっても「100%カバー」はあり得ない。
- 実は OO が従来の手続き型に対してやったことも似たようなもの→ということは、OO による進歩のあとで「残っていること」がどれくらい大きいのかは疑問な気もする。
- また現状の AOP 言語による「カバーのしかた」がいいのかどうかも分かっていない (研究段階)

□ これだけ警告したので、では遠慮なく AOP 言語について売り込んで見ることにします :-)

3.1 なぜ OO だけでは不足なのか?

□ OO がもたらしたもの→プログラムの扱う世界を「もの」の集まりとして位置付け、「もの」ごとに「性質」(インスタンス変数など) と「動作」(メソッドなど) を記述するようにした

- なおかつ「もの (の種類)」どうしの関係についても取り扱えるようにした (インタフェース/動的分配による総称性、継承による階層化、実装の引き継ぎ)。

□ じゃあ「それで十分じゃん」と思いますか? 何か「こういうことは欠けているのではないか」というのはありますか?

□ 例: 図形を扱うアプリケーションで「図形が変化したことを記録する」

- いろいろな図形がいろいろな風に変化する
- これまでの方法→変化させるメソッドに記録のコードを挿入
- ×本体の動作と記録の動作がごちゃまぜになる
- ×記録コードそのものは類似したものをそこら中にばらまくことに

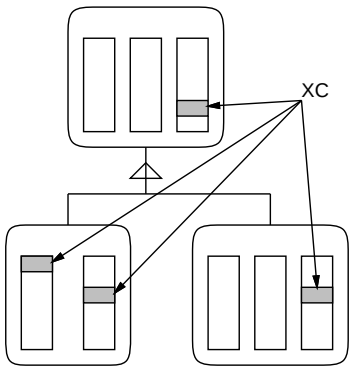
□ この他にもさまざまな「あちこちに同じもの挿入」の例がある

- データの永続化
- 表示の更新
- トレース、アンドゥ
- などなど…

3.2 横断的側面

□ 結局、これらの問題は「オブジェクトの階層構造とは直交した横断的な機能や動作」をプログラムに組み込もうとするところに起因する。

- このようなものを「横断的側面 (Croscutting Concern)」と呼ぶ



□ 横断的側面はオブジェクト指向による構造化と直交しているので、無理矢理オブジェクト指向で扱うよりも、新しい言語機構を導入して扱う方がよい。

□ …という演説でしたが、どう思いますか?

- ○確かに、その通りだと思う
- ×いや、言語をこれ以上いじくるより今のOO言語でやって行くべし

3.3 Aspect Oriented Programming (AOP)

□ AOPの基本的な考え方

- プログラムはいくつもの「側面」を持っている→それらをそれぞれ「分けて」取り扱うのがよい (separation of concerns)
- そのような「側面」のうちには、さまざまな部分にバラバラに現れて来るものがある→横断的側面 (croscutting concern)
- それらを分けて扱うため、メソッドやクラス階層とは「別の」メカニズム (=アスペクト) を導入して、これを通じて横断的側面を記述可能にする→Aspect Oriented Programming
- AOP言語→そのような機構を採り入れたプログラミング言語
- 用語: join point → 複数の側面を「接合させる」点のこと

□ 代表的なAOP言語の流派 (?)

- Adaptive Methods --- メソッドを状況に応じて変化 (適応) させることでさまざまな場合に使えるように → Demeter/Java (DJ) ライブラリ
- MultiDimensional Separation Of Concerns (MSDOC) アプローチ--- 複数の「側面」をそれぞれ記述し合成して1つのシステムに →Hyper/J言語
- Composition Filters --- 複数の側面を組み合わせるフィルタを用意してこれらを集めてシステムを組み立てる
- Aspect/J --- Aspectと呼ばれる記述を別に用意して既存のクラス構造を修正/拡張 ← AOP言語の「代表」の地位を占めつつある

□ 以下ではAspect/Jを例にどんな感じか見ていただく

3.4 Aspect/J

□ Gregor Kiczalesらが作成したJavaを土台としたAOP言語

- http://www.eclipse.org/aspectj/
- 現在の最新版 (AspectJ1.1rc1) はここから取れます
- JDK 1.4.xを前もって入れておく必要あり

□ 特徴…

- Javaの上位互換 (ふつーのJavaプログラムはAspectJのプログラムでもある)
- 既存のJavaプログラムにアスペクトを追加して行ける (既存のクラスを後づけで変化)

□ 以下では最低限感じが分かる程度の例題のみ説明

- 網羅的な解説とかは上記サイトで

3.5 Pointcut

□ Pointcut --- プログラム実行時の「ここ」という点。

- call(返値の型 クラス名.メソッド名(引数の型…)) --- これこれのクラスのこれこれのメソッドが呼ばれるところ
- call(クラス名.new(引数の型…)) --- これこれのクラスのコンストラクタが呼ばれるところ
- initialization(クラス名.new(引数の型…)) --- 同様だが、初期設定が行われるところ
- cflow(call(...)) --- 指定した呼び出し以下で行われる実行
- cflowbelow(call(...)) --- 同様だが、その呼び出し自体は含まない

□ 上記の基本的な pointcut を組み合わせて名前をつけることができる

- pointcut 名前(...) : 定義すみ pointcut の組合せ ;

3.6 Advice

□ Advice --- 「既存のプログラムのここをこういう風に直せ」という指定。

- before() : PC { コード } --- pointcut PC 実行直前にこれこれを行え。
- after() : PC { コード } --- pointcut PC 実行直後にこれこれを行え。
- after() returning : PC { コード } --- 同上ただし正常リターン時のみ。
- after() throwing : PC { コード } --- 同上ただし例外で戻った場合のみ。
- 返値の型 around() : PC { コード } --- pointcut PC の代わりに実行するコードを指定。コード中で proceed(...) というものを書いておくとそこで本来の実行コードが呼ばれる。

3.7 下敷きとなる Java の例題

□ 前回の例題の類似品

- アイコンをマウスでドラグして移動できるだけ

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

// 窓を開くアプリケーション
public class Sample41 extends Frame {
  DrawSet set = new DrawSet(); // アイコンの集合
  public Sample41() {
    set.addObj(new DraggableIcon("A", 100, 100));
    set.addObj(new DraggableIcon("B", 120, 120));
    set.addObj(new DraggableIcon("C", 140, 140));
    setSize(400, 400);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        set.mousePressed(evt.getX(), evt.getY());
        repaint(); // マウスダウン時の処理
      }
      public void mouseReleased(MouseEvent evt) {
        set.mouseReleased(evt.getX(), evt.getY());
        repaint(); // マウスアップ時の処理
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        set.mouseDragged(evt.getX(), evt.getY());
        repaint(); // ドラグ時の処理
      }
    });
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) {
        System.exit(0); // 窓が閉じる→終了
      }
    });
  }
  public void paint(Graphics g) { set.draw(g); }
  public static void main(String[] a) {
    (new Sample41()).setVisible(true);
  } // ここで窓を開く
}

// 画面上に現れるものを表すインタフェース
interface DrawObj {
  public void draw(Graphics g);
  public boolean hit(double x, double y);
  public void mouseDown();
  public void mouseUp();
  public void mouseDrag(double x, double y);
}

// DrawObj の集まりを保持しマウスで選択可能に
class DrawSet {
  DrawObj hit = null; // 選択中のもの
  DrawObj[] a = new DrawObj[200];
  int count = 0;
  public void addObj(DrawObj o) {
    if(count < a.length) a[count++] = o;
  }
  public void draw(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }
  public void mousePressed(double x, double y) {
    hit = null;
    for(int i = count-1; i >= 0; --i) {
      if(a[i].hit(x, y)) { hit = a[i]; break; }
    } // マウスの座標に当たった最初のものを選択
    if(hit != null) hit.mouseDown();
  }
  public void mouseReleased(double x, double y) {
    if(hit != null) hit.mouseUp();
    hit = null;
  }
  public void mouseDragged(double x, double y) {
    if(hit != null) hit.mouseDrag(x, y);
```

```
  }
}

// ドラグできるアイコン
class DraggableIcon implements DrawObj {
  Font fn = new Font("Serif", Font.BOLD, 24);
  String label;
  double xpos, ypos, rad = 25.0;
  Color col = Color.yellow;
  public DraggableIcon(String s, double x, double y) {
    label = s; xpos = x; ypos = y;
  }
  public void draw(Graphics g) {
    int r2 = (int)(rad*2);
    g.setColor(col);
    g.fillOval((int)(xpos-rad), (int)(ypos-rad), r2, r2);
    g.setColor(Color.black); g.setFont(fn);
    g.drawString(label, (int)(xpos-12), (int)(ypos+8));
  }
  public boolean hit(double x, double y) {
    return (xpos-x)*(xpos-x)+(ypos-y)*(ypos-y) < rad*rad;
  }
  public void moveTo(double x, double y) { xpos = x; ypos = y; }
  public void setColor(Color c) { col = c; }
  public double getX() { return xpos; }
  public double getY() { return ypos; }
  public void mouseDrag(double x, double y) { moveTo(x, y); }
  public void mouseDown() { setColor(Color.pink); }
  public void mouseUp() { setColor(Color.yellow); }
}
```

3.8 例: 画面の変化を記録するとしたら…

□ 案 1: DraggableIcon クラスのメソッドに記録用のコードを挿入してまわる

- もっといろいろなものが画面に現れるようになると大変そう
- 「記録する」のはどこか 1 箇所で何とかしたい

□ 案 2: アスペクトとして別途分離して記述

□ AspectJ で記述した変化表示アスペクト

```
import java.awt.*;

aspect DisplayChange {
  int count = 0;

  pointcut displayChange() :
    call(void DraggableIcon.moveTo(double, double)) ||
    call(void DraggableIcon.setColor(Color));

  after() : displayChange() {
    System.out.println("changed: " + (++count));
  }
}
```

□ 読み方

- DraggableIcon のメソッド moveTo() または setColor() が呼ばれたところ→「表示変化」という pointcut にする。
- 「表示変化」という pointcut の直後に「いくつ目の変更」というメッセージ表示を挿入する。

□ 動かし方

```
% ajc Sample41.java DisplayChange.java
% java Sample41
```

- ajc --- AspectJ Compiler
- 実行はこれまでと同じ ← 生成されるコードは普通の Java バイトコードだからそのまま客先に配布できる
- ただし! 1 つだけ追加のライブラリファイル (.jar ファイル) がある→それを CLASSPATH に追加 (上記 AspectJ 配布キットにドキュメント)

□ 驚くべきこと

- もとの Sample41.java には何ら手を加えていない
- 「ここをこう直したい」ということはアスペクト側だけで指定
- これによって複数のアスペクトを混合しても対処可能に
- またアスペクトの書き方を工夫すれば再利用可能に

3.9 例: 画面の変化を記録し後戻り可能にする

□ たとえば上のプログラムをさらに次のようにする

- とりあえず「動き」だけについて記録
- 記録の各レコードを表すクラスを用意する
- 位置変化ごとにそのインスタンスを配列に追加する
- 後戻りボタンを画面に貼り付けておく
- ボタンが押されたら配列から最後の位置記録を取り出してそこへ戻す

□ こういうのを実現するとしたらどういう機能がさらに必要?

□ さっきのと何が違うか? → 挿入したコードの中からアイコン等のオブジェクトを参照しなければできない

- 実は pointcut の冒頭で「変数宣言」できる。
- さらにその宣言した変数が「カレントオブジェクトである」「メソッド呼び出し対象オブジェクトである」等の指定を書くことであてはめが行われ、変数に値が取り出せる。
- その pointcut を利用する側でこのパラメタを参照できる。

□ さっきより少し長いが全部まとめて示すと…

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

aspect RecordChange {
  History[] record = new History[5000];
  int count = 0;
  Button b1 = new Button("Back");

  pointcut createApp(Sample41 app) :
    this(app) &&
    initialization(Sample41.new(..));
  pointcut positionChange(DraggableIcon i) :
    target(i) &&
    call(void DraggableIcon.moveTo(double, double))&&
    !cflowbelow(call(void History.apply()));
  pointcut colorChange(DraggableIcon i) :
    target(i) &&
    call(void DraggableIcon.setColor(Color));

  after(DraggableIcon i) : positionChange(i) {
    System.out.println("moved: " + (++count) + " : " +
      i.getX() + "," + i.getY());
  }
  before(DraggableIcon i) : positionChange(i) {
    record[count] = new History(i, i.getX(), i.getY());
  }
  after(Sample41 a) : createApp(a) {
    final Sample41 app = a;
    app.setLayout(null); app.add(b1);
    b1.setBounds(20, 20, 80, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        if(count > 0) {
          record[--count].apply(); app.repaint();
        }
      }
    });
  }
  static class History {
    DraggableIcon icon; double xpos, ypos;
    public History(DraggableIcon i, double x, double y) {
     icon = i; xpos = x; ypos = y;
    }
    public void apply() { icon.moveTo(xpos, ypos); }
  }
}
```

□ まず最後に記録用の入れ子クラスがつけてある。これのインスタンスにアイコンとその位置を記録する

□ 冒頭の配列 record にインスタンスを順次入れていく

□ pointcut は 3 つに増えている。いずれもパラメタを取るようにしている

- Sample41 のインスタンスを作るところ
- アイコンが動くところ、ただし History からの呼び出しは除く
- アイコンの色が変わるところ

□ 位置変化の「後」の動作はさっきと同じ

□ 位置変化の「前」に配列にに位置を記録する

□ Sample41 が初期化された「後」にボタンを貼り付ける

- ボタンの動作の中で後戻りを実現
- ボタンの動作中から app を参照するには変数 app は final である必要がある

3.10 AspectJ と AOP のまとめ

□ AspectJ にはほかにもいろいろ機能があるけど…

- 要するに「外から」既存のクラスの動作前後に介入できることがポイント
- SOC (Separation of Concern) とはこういう感じ
- 他の AOP 言語もそれぞれ違った側面からこのような「分離」に挑戦している

□ それで…

- ○このような言語はすばらしい! と思いましたか? または
- ×こんなに複雑にしてもしょうがないじゃん! と思いましたか?

4 オブジェクト指向の利用技術

□ オブジェクト指向言語→これまでの言語にないさまざまな「道具だて」(例: サブクラス、動的分配、継承、委譲、インタフェース、…)

- それをどのように使うか→なかなか難しい問題

□ どのような方向での利用があるか?

4.1 オブジェクト指向にあった設計

□ 美しいプログラム構造 (何が美しい???) ←オブジェクト指向言語の特性に合ったプログラム設計← OOSE(オブジェクト指向分析、オブジェクト指向設計、UML、統一メソッド): 本講の範囲外

4.2 再利用

□ 再利用←書くよりは書かないで済ませた方が生産効率はよいに決まっている

- なぜオブジェクト指向で再利用? → クラス、オブジェクトといった単位は従来の「サブルーチン」より固まりが大きく、再利用に向いている
- 再利用するものは何? コード? 設計知識? → さまざまなレベルがある

4.3 クラスライブラリ

□ Smalltalk →充実したクラスライブラリが付属→クラスライブラリを熟知すれば生産性が高まる、というブームに

- 実際にやってみると、よいクラスライブラリの開発/クラスライブラリに熟知した人材の育成ともに簡単ではない

□ Smalltalk クラスライブラリでは差分プログラミング (子クラスで親クラスの機能を少しずつ拡張していく) を多用→これもよい手法だと一時思われていた

- しかしやってみると、よい差分プログラミングは難しい (クラス間の依存関係が大きくなりぐちゃぐちゃになりやすい)

□ 現在では、クラスライブラリはもちろん必要だが、整った機能を一式、分かりやすいインタフェースで提供するという当たり前の結論に

4.4 アプリケーションフレームワーク

□ 抽象メソッド: 親クラスで「自分自身へのメソッド呼び出し」を用いたメソッドを定義→子クラスでそれらのメソッドを具体的なものに差し替え

□ これをさらに発展させて、汎用的なアプリケーション全体の構造を予め定義しておく→その中のいくつかのクラスをサブクラス化してそこに各アプリケーション固有の部分を記述することでアプリケーションを完成させる

- 例: MFC、ET++、Choices、…
- うまく当てはまれば生産性は高まるが、何をどうサブクラス化するか、サブクラスはどのような規約に従う必要があるか、といったことを学ぶのが大変

□ たとえば、アプレットもアプリケーションフレームワークの 1 つ

- アプレットはクラス Applet のサブクラスとして作り、必要なメソッドをオーバライドする
- init() → 初期設定する
- paint() → 描画する
- start()、stop() → ページの表示開始/終了
- アプレットの場合は難しいカスタマイズはしないという感じ

4.5 インタフェースを用いたフレームワーク

□ サブクラスによるオーバライド→前回述べた問題 (?)

□ (つまり、親クラスを熟知しないと難しい、カプセル化が破れるなど)

□ インタフェースを使えばこれらの問題がない

- モデルと UI の分離

□ 例題: ボタン、入力欄、表示欄が 1 つずつある UI

```
import java.awt.*;
import java.awt.event.*;

public class Sample31 extends Frame {
  MyApp app;
  Label l0 = new Label();
  Button b0 = new Button();
  TextField t0 = new TextField();
  public Sample31(MyApp a) {
    app = a;
    l0.setText(app.getMessage());
    b0.setLabel(app.getName());
    setLayout(null); setSize(200, 250);
    add(l0); l0.setBounds(10, 40, 180, 40);
    add(t0); t0.setBounds(10, 100, 180, 40);
    add(b0); b0.setBounds(10, 160, 90, 40);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        app.setData(t0.getText()); t0.setText("");
        l0.setText(app.getMessage());
      }
    });
  }
  public static void main(String[] args)
      throws Exception {
    Frame f = new Sample31(new App1());
//  Frame f = new Sample31(new App2());
//  Frame f = new Sample31((MyApp)
//    (Class.forName(args[0])).newInstance());
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
  }
}

interface MyApp {
  public String getName();
  public String getMessage();
  public void setData(String s);
}

class App1 implements MyApp {
  String mesg = "摂氏華氏変換です";
  public String getName() { return "華氏→摂氏"; }
  public String getMessage() { return mesg; }
  public void setData(String s) {
    try {
      float x = (new Float(s)).floatValue();
      float y = (5f/9f)*(x - 32f);
      mesg = "摂氏の温度: " + y;
    } catch(Exception e) { mesg = e.toString(); }
```

```
  }
}

class App2 implements MyApp {
  double sum = 0.0;
  String mesg = "合計を求めます";
  public String getName() { return "合計"; }
  public String getMessage() { return mesg; }
  public void setData(String s) {
    try {
      sum += (new Double(s)).doubleValue();
      mesg = "合計: " + sum;
    } catch(Exception e) { mesg = e.toString(); }
  }
}
```

□ アプリケーションはインタフェース MyApp にさえ従っていればよい

□ UI 側はアプリケーションが何であるか知らなくてよい

□ アプリケーション側もインタフェースについて知らなくてよい

□ たとえばコマンド行インタフェースにもできる

```
import java.io.*;

public class Sample32 {
  public static void main(String[] args)
      throws Exception {
//  MyApp app = new App1());
//  MyApp app = new App2());
    MyApp app = (MyApp)
      (Class.forName(args[0])).newInstance();
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    while(true) {
      System.out.println(app.getMessage());
      System.out.print("> ");
      String line = in.readLine();
      if(line.equals("")) break;
      app.setData(line);
    }
  }
}
(以下まったく同じにつき略)
```

4.6 コンポーネント

□ ここまでの再利用技術→基本的にクラス (群) が対象→プログラマ向け (クラスベースとも言う)

□ プログラミングをしない人に使える再利用技術→インスタンスベースの再利用

- インスタンスを生成し、そのプロパティ(属性、要するにインスタンス変数の値) をカスタマイズする
- カスタマイズしたインスタンス群をディスク等に保存しておき、それを取り出してそのまま動かす
- そのようなインスタンスを「コンポーネント」と呼んでいる。ソフトウェア開発のためのコンポーネント群→「コンポーネントウェア」、コンポーネントウェアに基づくソフトウェア開発→「部品組み立てプログラミング」

□ 代表的な成功例→ VisualBasic (部品: VBX、OCX… COM、DCOM の部品)

- ほかにも国産の IntelligentPad、Java ベースの JavaBeans などいろいろある
- しかし、部品とその配線だけでできるプログラムで十分なの?

4.7 デザインパターン

□ パターン: 「繰り返し現われるようなカタチ」

□ (ソフトウェアにおける) デザインパターン: オブジェクト指向ソフトウェア開発において、有効に使えるようなオブジェクト群の構成のパターン

- 1990 ころから、Peter Cord 他がはじめた。日本では「ガンマ本」が有名になっている。ガンマ本はよく使うパターンを集めた「パターンカタログ」になっている。
- なぜデザインパターン? → オブジェクトの接続関係のノウハウはかなり難しい (ちょっと思いつかないようなものもある) →それを蓄積しておいて流通させると、うまくはまったときに役立つ

□ 例: Command パターン

- メニューの選択、画面上のボタンなどはどれも「何らかの動作」を起動する→「動作をするオブジェクト」を用意して、それをメニューやボタンに結びつけていけばよい。

□ 例: Adapter パターン

- ボタンが押されたときに呼び出されるメソッドはある名前に決まっている。しかし実際に起きて欲しいことを実行するメソッドは別のメソッドである→「仲介するオブジェクト」を用意して、それが橋渡しをすればよい。

□ 例: Visitor パターン

- オブジェクトの階層構造で構造化グラフィクスとか複合文書のようなものを作ったとする。「印刷する」「表示する」「ファイルに保存する」「スペルチェックする」等それぞれの場合について、各クラスにそれ用のメソッドを作るのは面倒である → どうする???

□ Visitor パターンとは:

- 各オブジェクト側には「accept」というメソッドを 1 つだけ用意しておく。その引数として、「印刷用の Visitor」「表示用の Vsitor」などさまざまな Viritor オブジェクトをそのつど渡せばよい

□ 例: AbstractFactory パターン

- Windows でも Mac でも X11/Unix でも同じに動作する GUI アプリケーションを開発するには???

□ AbstractFactory パターンとは:

- Window、Button、Dialog などの汎用的なクラスを用意する
- そのサブクラスとして MacWindow、X11Window などそれぞれ用意する
- WinFactory という抽象クラスを用意し、メソッドとして makeWindow、makeButton 等を用意する
- MacWinFactory、X11WinFactory などの具象クラスでこれらをそれぞれ実装する
- アプリケーションの実行開始時に MacWinFactory などのインスタンスを作って WinFactory 型の変数に格納し、以後それを利用する

4.8 本節のまとめ

□ オブジェクト指向にはさまざまな「道具」が含まれている→その「道具」をどう使うか、についていろいろな工夫がある

□ しかしこれらもまだほんの一部? →これからより多くの「よりよい利用方法」が現われる (はず)

□ それらの利用方法を個別に「新しい」と思って受け入れるだけでは、流行に追われるだけ→必要なこと:

- その「新しい」技術がこれまで行われてきたさまざまなことの中にどのように位置づけられるのかを考える
- 結局、ソフトウェアの生産において「自分が必要とすることは何か」をまず考え、それに照らして必要なものを取捨選択する

5 並列システム

□ 並列 (parallel) システムとは

- 複数の動作が「同時に」「並行的に」行われるようなシステム
- 動作が「順番に」「逐次的に」行われるよりも速い

□ cf. 並行 (concurrent) →「論理的に並列」(例: 1CPU でのマルチタスク)

- 並行であっても並列でないと速くはならない。システムが並列であってもどこかに並行性がないと速くならない。

5.1 システムを速くする方法

□ 方法 1: CPU の動作を速くする

- ソフトウェアを変更しなくて済む→大きな利点
- 素子の高速化は限界に近付きつつある (といいつつ…)

□ 方法 2: CPU をたくさん搭載する (マルチプロセサ)

- 過去においてはハードウェア量が N 倍になるので不利だとされていた
- 現在は CPU が安く/小さくなり、実装技術も進歩
- 結果的に「動作を速くする」よりコストパフォーマンスが良くなった。例: スーパーコンピュータ
- しかし、ソフトウェアは同じでは済まない「かも」

5.2 細粒度並列性

□ 命令レベル並列性ともいう

- 1 つの機械命令の中でも並列に動作可能な部分が含まれる
- 隣接する命令群の中でも同様
- これを活かすには→命令スケジューリング (ハード、ソフト) →＃ 1 で解説した通り

□ 命令レベル並列性の限界→ループ等は別として、ふつうのコード部分では並列度が 4 くらいで頭打ちと言われている

□ 命令レベル並列性の利点→言語は同じままでコンパイラだけ交換

5.3 粗粒度並列性

□ 命令レベルよりもっと上のレベルでの並列性→マルチプロセサ

- 例: 配列のスカラー倍→各要素独立に計算可能
- 例: 返値を使わないサブルーチン呼び出し→本体と並列に実行可能

□ 並列度の上限はいくらでも大きくできる→以下ではこちらを対象

□ 粗粒度並列性の利用には 2 つ半 (?) のアプローチ

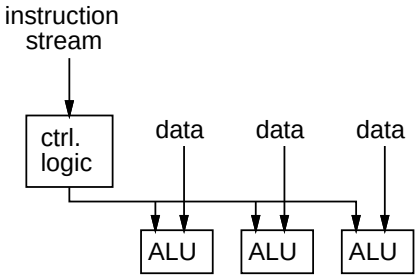
- 言語を変更しない→自動並列化コンパイラ

- 言語を変更しないが、プログラマが多少サポート→自動並列化+プラグマ
- プログラマが明示的に並列を意識してコーディング→ライブラリまたは並列言語

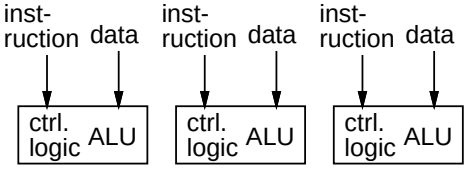
□ アムダールの法則： 並列化できる部分が 90%→並列化で最大 10 倍までしか速くできない

5.4 並列システムのモデル

□ SIMD (single instruction stream、multiple data) →プログラムの実行の流れは 1 つで、データは多数→データ並列とも呼ぶ→行列/ ベクトル計算などに適したモデル→現在は純粋な SIMD マシンはない。ベクトルプロセサがややこれに近い



□ MIMD (multiple instruction stream、multiple data) → CPU が複数あり、それぞれが別の実行の流れを持つ→現在のマルチプロセサの素直なモデル、主流

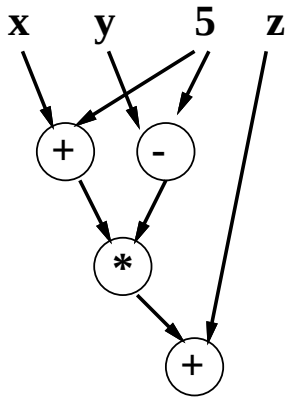


□ SPMD (single program、multiple data) →複数の CPU で同一のプログラムを同期を取りながら実行→MIMD マシンで SIMD のまねをする→データ並列の計算に使われる手法

5.5 データフローマシン

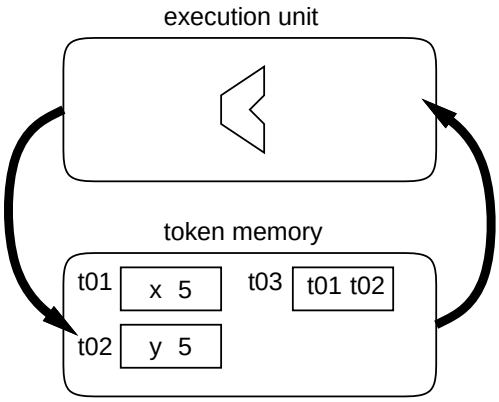
□ データの依存関のみに基づいて実行順序を制御するような専用マシン

□ プログラムはデータフローグラフとして表される



- 読みづらいので普通の言語ふうのものも

□ フローグラフの「ノード」をプールして「準備ができた」ものから「発火」

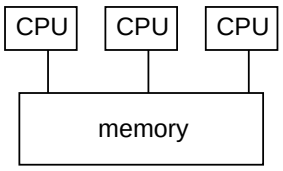


- 並列単位が小さいので同期処理のオーバヘッドが大きく性能が出なかった
- 処理単位をもっと大きくしたものを電総研 (現産総研) などで開発 (EM-4、EM-X など)
- しかしあまり主流とは言えない

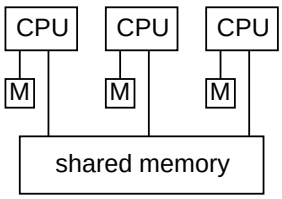
5.6 マルチプロセサの分類

□ マルチプロセサのモデル分類

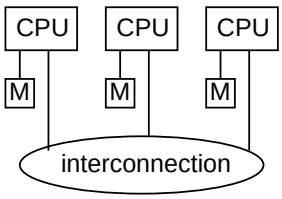
- UMA(uniform memory access) アーキテクチャ→各 CPU は 1 つの共有メモリをアクセスし、CPU ごとにアクセス時間の変動はない



- NUMA(non-uniform memory access) アーキテクチャ→各 CPU は手元のメモリとそうでないメモリにともにアクセスできる→場所によってアクセス時間が異なる



- NORMA(no remote memory access) アーキテクチャ→各 CPU は手元のメモリだけにアクセスできる (リモートアクセスはできない)



□ UMA → NUMA → NORMA の順で実装は面倒、使いやすさは低くなるが、その代わりスケーラビリティが増す

- これまでは普通の OS でそのまま利用できる UMA が多く使われて来た
- これからは 1000 とか 10000 とかの CPU を搭載したシステムも→ NUMA、NORMA が中心に

□ 現在ではネットワークも極めて速くなってきた→複数マシンをネットワーク接続したものを「並列マシン」として使うことも→ NORMA の延長線上

- コスト的には極めて有利
- 性能が出るかどうかは、並列の粒度とネットワーク遅延の比率次第

5.7 相互結合網

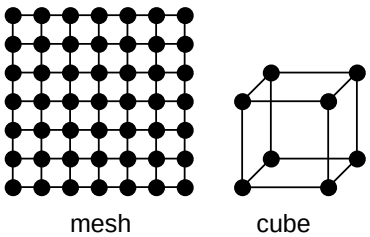
□ マルチプロセサ内部のデータのやりとり→相互結合網

□ UMA →スケーラビリティはあまりない (30 くらいまで?)

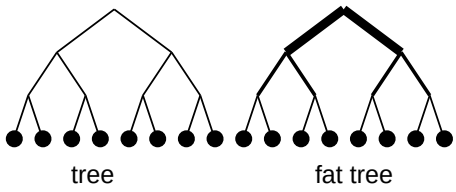
- クロスバスイッチで N 個の CPU と M 個のメモリを結合
- 共有バスとキャッシュ(後述) による結合

□ NUMA、NORMA →大規模マルチプロセッサ

- CPU+ローカルメモリ+キャッシュ→ 1 モジュール
- モジュール間の相互結合方式: ネットワークのようなもの
- メッシュ、トーラス、ハイパーキューブ、…



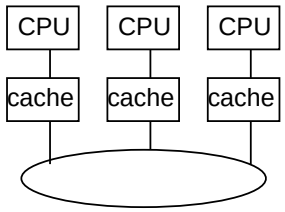
- ツリー、fat tree



5.8 キャッシュと整合性

□ キャッシュ→ CPU に隣接した高速記憶→メモリの内容を一時的に保持

- UMA、NUMA ではキャッシュを利用することで「遅い共有メモリを見なくて済む」ように



□ ただし共有メモリが書き換えられたときそのことを知らないでキャッシュを利用し続けると困る→整合性 (コヒーレンス) の問題

- スヌープキャッシュ→バスにキャッシュ+CPU がつながっている場合→バス上の通信を監視することで、各キャッシュが整合性を取れる→バス方式ではスケーラビリティはあまりない
- ディレクトリ方式→階層構造のシステムで使用→「どことどこにキャッシュされているか」を記憶しておき個別に通知

5.9 この節のまとめ

□ 並列とはどういう意味か、なぜ必要か

□ 並列システムの分類

□ 並列システムの構造

6 並列プログラミングのモデル

□ 並列プログラミング→「並列性を明示的に扱うような」プログラミング

□ 〃のモデル→「どのようなシステム」を抽象化したもの

□ 実際のシステムの構成と一致しない場合も

6.1 自動並列化

□ 通常のプログラムを自動的に (コンパイラが) 変換

□ 自動ベクトル化、MIMD 化、SPMD 化ともある

- データの依存解析が必要→コンパイラにとって難しい課題
- プログラマがプラグマでコンパイラに教えてやることも一般的

□ NUMA、NORMA の場合は配列などのデータ配置も重要→これも自動的に行う場合、プログラマがアドバイスする場合ともにある

□ 以下では明示的に並列性をプログラムするものとする

6.2 通信モデル

□ 複数の実行主体がどのように情報を交換するかのモデル

- 共有メモリモデル→ UMA のモデル化
- メッセージモデル→ NORMA のモデル化
- 実際にハードが UMA や NORMA だとは限らない

6.3 共有メモリモデル

□ 複数の実行主体がメモリ領域を (少なくとも部分的に) 共有しているというモデル

□ 利点:

- 通信が高速 (送ろうと思った瞬間にはもう相手にアクセスできる状態になっている
- 通信の単位が任意 (1 語ずつでも読み書きできる)
- ポインタが渡せる

□ 弱点:

- 競合/副作用の問題→排他制御 (後述)

6.4 メッセージモデル

□ 実行主体どうしでメモリは共有せず、メッセージ送信によって互いにデータをやりとりする

□ 利点:

- やりとりの機構に同期が組み込まれている (後述)
- 競合や副作用が起こりにくい

□ 弱点:

- 遅い、ポインタが渡せない、通信のオーバヘッド

6.5 モデルとシステム構成の独立性

□ 共有メモリシステムでメッセージ←別に難しくない (キューやバッファをメッセージチャネルと思えばよい)

□ NORMA や分散システムで共有メモリモデル←分散共有メモリ (ページフォルトが起きたときにページの内容を転送し、これまでページを持っていたところはページをはがす)

6.6 並列性のモデル

□ データ並列→プログラムの実行箇所は 1 つ (SPMD かも) →理解しやすい。並列性は行列、ベクトルなどのデータによる

- 数値計算ものではその利用は確立している
- 行列、ベクトルではメモリ配置が重要になる

□ 制御並列→複数の実行主体があるようなモデル

- 分配収集 (scatter/gather) モデル→ある程度規則的→実装はしやすい
- 並列オブジェクト指向→もっとばらばら

6.7 並列プログラミングの枠組み

□ 専用言語か旧来の言語か

- 新しい言語を作って普及させるのはとても大変
- そのため、研究レベルでは多くの言語が作られても、世の中に普及することは極めて少ない
- Java は極めて特異な例 (ただし Java は並列言語かどうか?)

6.8 並列言語

□ 並列言語→並列実行をモデルとして持つようなプログラミング言語

□ 実際にはまだ「広く普及した」例はない

□ データ並列型→それなりに実用。C* とか HPF (High Performance Fortran) など。基本的には多数のデータに対する並列動作をいかに輻輳を起こさず (マシンの特性を活かし切るように) 制御するか

□ それ以外→複数の「実行の流れ」(MIMD に対応) →さまざまな流儀→以下で取り上げて行く

6.9 直列言語による並列プログラミング

□ 通常の言語+ライブラリ呼び出しで並列実行を実現

□ たとえばプロセスを複数生成→複数の CPU で実行可能

□ 複数プロセス間でネットワーク通信→メッセージ通信による並列プログラミング

- PVM、MPI、Linda 等→分散システムとしての利用が多い

□ 複数プロセス間でメモリを共有→共有メモリモデルによる並列プログラミング

- 複数プロセスを利用した場合、それらの切り替わりには必ず OS が介在→性能上は不利→スレッドの利用に

6.10 スレッド

□ スレッドとは「実行の流れ」程度の意味

- 1 つのプロセスの中で複数の実行の流れ→切り替わりのオーバヘッドがない→性能上有利
- 実現→複数のスタックを置く
- スタック以外のコード、データは共有されている

□ カーネルスレッド→カーネルがそのスケジューリングを管理→複数の CPU で並列に動かすためにはこれが必須

□ ユーザレベルスレッド→システムコールなしで、自前で (ライブラリの内部で) 切り替わりを実現→オーバヘッドが小さい

□ 実際には両者を併用→論理的なスレッド数>>CPU 数なので、CPU 数+ α 程度のカーネルスレッドを用意し、カーネルスレッドが多数のユーザレベルスレッドを切替えながら動かして行く

6.11 Java のスレッド機能

□ スレッドはクラス Thread で実現

□ 方法 1: Thread のサブクラスを作り run() をオーバライド

```
public class Sample35 {
  public static void main(String args[]) {
    Thread t1 = new Sample35Thread("A", 10, 2000);
    Thread t2 = new Sample35Thread("B", 15, 1500);
    t1.start(); t2.start();
    try {
      t1.join(); t2.join();
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

```
class Sample35Thread extends Thread {
  String mesg = "???";
  int count = 10, time = 1000;
  public Sample35Thread(String m, int c, int t) {
    mesg = m; count = c; time = t;
  }
  public void run() {
    try {
      for(int i = 0; i < count; ++i) {
        Thread.sleep(time);
        System.out.println(mesg);
      }
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

□ 方法 1 では、既に別のクラスのサブクラスになっているものを Thread のサブクラスにできないので不便

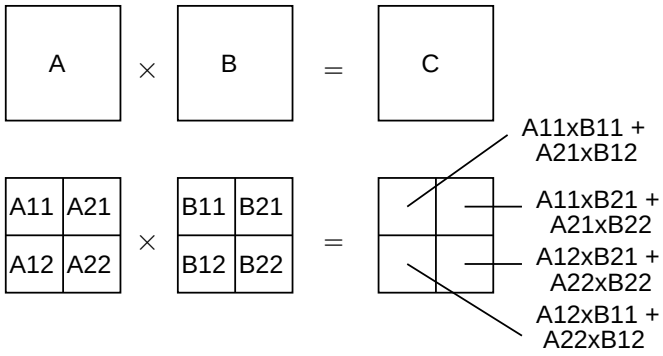
□ 方法 2: Runnable インタフェースを実装したオブジェクトを用意し Thread のコンストラクタで引数として渡す

```
public class Sample36 {
  public static void main(String args[]) {
    Runnable r1 = new Sample36Run("A", 10, 2000);
    Runnable r2 = new Sample36Run("B", 15, 1500);
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    t1.start(); t2.start();
    try {
      t1.join(); t2.join();
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

```
class Sample36Run implements Runnable {
  String mesg = "???";
  int count = 10, time = 1000;
  public Sample36Run(String m, int c, int t) {
    mesg = m; count = c; time = t;
  }
  public void run() {
    try {
      for(int i = 0; i < count; ++i) {
        Thread.sleep(time);
        System.out.println(mesg);
      }
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

□ これだけでは何が嬉しいのかよく分からないかも知れないが…

□ たとえば「行列の乗算」を考えてみる…次のように分割すると、4 個の部分行列の計算は並行して行なえる



□ 乗算ができる行列のクラス

```
class S37Matrix {
  int n;
  double[][] a;
  public S37Matrix(int n0) {
    n = n0; a = new double[n][];
    for(int i = 0; i < n; ++i) {
      a[i] = new double[n];
      for(int j = 0; j < n; ++j) a[i][j] = 0;
    }
  }
  public void fillRandom() {
    for(int i = 0; i < n; ++i)
      for(int j = 0; j < n; ++j)
        a[i][j] = Math.random();
  }
  public int getdim() { return n;  }
  public double get(int i, int j) {
    return a[i][j];
  }
  public void set(int i, int j, double d) {
    a[i][j] = d;
  }
  public void copy(int i0, int j0, int n0,
                   S37Matrix m, int i1, int j1) {
    for(int i = 0; i < n0; ++i)
      for(int j = 0; j < n0; ++j)
        a[i0+i][j0+j] = m.get(i1+i, j1+j);
  }
  public void mult(S37Matrix b, S37Matrix c) {
    if(b.getdim() != n || c.getdim() != n)
      throw new RuntimeException("matrix size");
    for(int i = 0; i < n; ++i)
      for(int j = 0; j < n; ++j) {
        double d = 0.0;
        for(int k = 0; k < n; ++k)
          d += b.get(i, k) * c.get(k, j);
        a[i][j] += d;
      }
  }
}
```

□ ランダムに行列を生成して掛け算してみる

```
public class Sample37 {
  public static void main(String[] args) {
    int w = 200;
    int h = w / 2;
    S37Matrix a = new S37Matrix(w);
    S37Matrix b = new S37Matrix(w);
    S37Matrix c = new S37Matrix(w);
    a.fillRandom(); b.fillRandom();
    System.out.println("start.");
    double l0 = System.currentTimeMillis();
    c.mult(a, b);
    double l1 = System.currentTimeMillis();
    System.out.println(l1-l0);
    final S37Matrix a11 = new S37Matrix(h);
    final S37Matrix a12 = new S37Matrix(h);
    final S37Matrix a21 = new S37Matrix(h);
    final S37Matrix a22 = new S37Matrix(h);
    final S37Matrix b11 = new S37Matrix(h);
    final S37Matrix b12 = new S37Matrix(h);
    final S37Matrix b21 = new S37Matrix(h);
    final S37Matrix b22 = new S37Matrix(h);
    final S37Matrix c11 = new S37Matrix(h);
    final S37Matrix c12 = new S37Matrix(h);
    final S37Matrix c21 = new S37Matrix(h);
    final S37Matrix c22 = new S37Matrix(h);
    a11.copy(0, 0, h, a, 0, 0);
    a12.copy(0, 0, h, a, h, 0);
    a21.copy(0, 0, h, a, 0, h);
    a22.copy(0, 0, h, a, h, h);
    b11.copy(0, 0, h, b, 0, 0);
    b12.copy(0, 0, h, b, h, 0);
    b21.copy(0, 0, h, b, 0, h);
    b22.copy(0, 0, h, b, h, h);
    System.out.println("start.");
    double l2 = System.currentTimeMillis();
    c11.mult(a11, b11); c11.mult(a12, b21); /**/
    c12.mult(a12, b12); c12.mult(a12, b22); /**/
    c21.mult(a21, b11); c21.mult(a22, b21); /**/
    c22.mult(a21, b12); c22.mult(a22, b22); /**/
    double l3 = System.currentTimeMillis();
    System.out.println(l3-l2);
    S37Matrix d = new S37Matrix(w);
    d.copy(0, 0, h, c11, 0, 0);
    d.copy(0, h, h, c12, 0, 0);
    d.copy(h, 0, h, c21, 0, 0);
    d.copy(h, h, h, c22, 0, 0);
  }
}
```

□ 「final」は「この変数は最初に初期設定したら後は書き換えない」という指定。スレッド内と外とで共有されるのでそういう風に指定しないといけなくしてある (Java の方針)。

□ では、4 つの計算をスレッドに分けて 2CPU のマシンで動かしてみる。

- 先の 4 行を次のように直した

```
Thread t1 = new Thread() {
  public void run() {
    c11.mult(a11, b11); c11.mult(a12, b21);
  }
}; t1.start();
Thread t2 = new Thread() {
  public void run() {
    c12.mult(a12, b12); c12.mult(a12, b22);
  }
}; t2.start();
Thread t3 = new Thread() {
  public void run() {
    c21.mult(a21, b11); c21.mult(a22, b21);
  }
```

```
  }; t3.start();
  Thread t4 = new Thread() {
    public void run() {
      c22.mult(a21, b12); c22.mult(a22, b22);
    }
  }; t4.start();
  try {
    t1.join(); t2.join(); t3.join(); t4.join();
  } catch(Exception ex) { }
```

```
% java Sample37
start.
199.0
start.
95.0
%
```

□ 確かに倍くらい (もっと?!) 速くなっている。

- CPU 数 N 倍で計算が N 倍速いのを linear speedup と言い「理想状態」の基準として用いられる。
- N 倍よりもっと速いのを superlinear speedup と言い「まゆつば」を表すものとしている (本当はそれなりの訳がある)。

□ 注意! この例題の計算手順そのものにはバグがある (こういうのが苦手でして…)

6.12 この節のまとめ

□ 通信のモデル→共有メモリ vs メッセージ

□ 実行のモデル→データ並列 vs 制御並列

□ 言語の分類→並列言語 vs 直列言語+ライブラリ

7 並行制御と並列言語

□ 並行制御とは→複数の実行主体間で「実行の順序」を制御すること

□ なぜ必要?

- 例: 複数の下請けが仕事を請け負って並列実行→全部完了したらまとめて完成→「全部完了した」後でなければまとめられない
- 例: 複数の作業者が 1 つのデータベースを更新→同時に更新を行うと矛盾が生じる。たとえば次のは NG

```
Process_A:          Process_B:
  //金額 x を計算      //金額 x を計算
  bal += x;           bal += x;
  //完了              //完了
```

- 上のようなコードは機械語レベルでは…

```
Process_A:          Process_B:
  load r1,x           load r1,x
* load r2,bal         load r2,bal
* add r2,r1           add r2,r1
* store r2,bal        store r2,bal
```

- だから何らかの並行制御により、「*」の 3 命令を同時には実行しないようにする必要がある。

□ 並行制御のための「機能」としてさまざまなものが提案されている→以下で見ていく→共有メモリを前提としたものが多い (歴史的事情)

7.1 排他領域

□ 排他領域とは→一時点には 1 つの実行主体しか実行が入れないようなプログラムコードの範囲

- 1CPU の場合「割り込み禁止」にすれば実現できる。マルチプロセッサでは別の方法が必要
- ある実行主体が入っているとき、別の実行主体が入ろうとすると待たされる
- 入っている主体が出て行くと別の実行主体が入れるようになる

7.2 排他領域の実現手法

□ 「専用命令」???

□ 共有メモリ→メモリ上の特定番地は 1 つの値しか保持しない (CPU ごとに別の値に見えることはない) →これを利用して実現。

- ダメな例:

```
Process_A:              Process_B:
 while(lock != 0) ;      while(lock != 0) ;
 lock = 1;               lock = 1;
 // critical             // ciritcal
 lock = 0;               lock = 0;
```

- 「同時に」実行しても大丈夫にするのはかなり面倒

7.3 Dekker のアルゴリズム

□ 最初に「正しい解」を考えた人が Dekker。

```
Process_A:            Process_B:
 in[A] = 1;            in[B} = 1;
 while(in[B]==1) {     while(in[A]==1) {
   while(pri!=A)         while(pri!=B)
     in[A] = 0;            in[B] = 0;
   in[A] = 1; }          in[B] = 1; }
 // critical           // critical
 pri = B;              pri = A;
 in[A] = 0;            in[B] = 0;
```

7.4 Peterson のアルゴリズム

□ Dekker の解よりも簡単なのを発見したのが Peterson。

```
Process_A:      Process_B:
 in[A] = 1;      in[B] = 1;
 turn = A;       turn = B;
 while(turn==A   while(turn==B
  && in[B]==1) ;  && in[A]==1) ;
 // critical     // critical
 in[A] = 0;      in[B] = 0;
```

□ 結局、これを N 個のプロセスでプログラムするのは大変

7.5 スピンロック

□ メモリ番地を「不可分にテストして変更」する命令 (Test and Set) があれば話はずっと簡単→現在の CPU にはほどんと備わっている。

- 具体的な動作→「番地 x が 0 でなければ false を返し、0 なら 1 に変更して true を返す」(CPU ごとに仕様はやや違う)

```
Process_A:      Process_B:
 while(TS(x)) ;  while(TS(x)) ;
 // critical     // critical
 x = 0;          x = 0;
```

- これならプロセスが何個あっても大丈夫
- 待っているプロセスはそこで「ぐるぐる回る」→「スピンロック」
- CPU を消費しながら待つ→「busy wait」→短時間の待ちにのみ使用する (排他領域は極めて小さいものであるべき)
- ハード的には→ TS のループはバストラフィックを作らない (キャッシュに入っているから) →システムの負担にならない (CPU 以外は)

7.6 セマフォ

□ セマフォ→特別なカウンタ。P と V という 2 つの操作が可能。

- P: カウンタを 1 つ減らす。減らした結果が 0 以下ならそのプロセスは「寝る」
- V: カウンタを 1 つ増やす。増やした結果が 1 なら 1 つプロセスを「起こす」

□ N 個の資源を共同利用するには初期値 N のセマフォを用いる

□ 初期値 0 のセマフォ→バイナリセマフォ→排他領域の実現に使える

- スピンロックと何が違うか→待っている間は CPU を消費しない。ただしオーバヘッドは大きい
- 実際には「セマフォの操作」を「スピンロックで排他制御」する

```
Sem_P:            Sem_V:
  lock(sem);        lock(sem);
  if(--sem<0) {     if(++sem==1) {
    unlock(sem);      //wakeup someone }
    //sleep self } unlock(sem);
  else
    unlock(sem);
```

7.7 条件変数

□ セマフォは「カウントが 1 未満の間寝て待つ」

□ より一般的に「これこれの条件を寝て待つ」→条件変数

- wait(c) : 条件変数 c で寝て待つ
- signal(c) : c で寝て待っている人を一人起こす

7.8 モニタ

□ モニタ→ C. A. R. Hoare などが提案した言語機構

□ セマフォやスピンロックは排他領域の「入口」と「出口」でペアになって使わないといけない→失敗すると大変

□ 実際に「保護したい対象」も含めて言語の構文として用意した方がよい

□ モニタ: モジュールのカプセル化機能に排他制御を追加したもの

- モニタの操作 (手続き) を呼び出すとロックが掛かる
- 手続きが終って出るとロックが開放される
- 待ち合わせは条件変数を指定して wait(c) →待ちに入るとロックは開放
- 起こすには別の人がモニタに入って signal(c) → 1 人だけ起きる
- 起きたプロセスはロックを獲得した状態で動作を続行

□ 例: Hoare ふうのモニタ

```
type buf = monitor
  var val:integer;
      isin:boolean := false;
      full,empty: condition;
  procedure put(i:integer);
  begin
    if isin then wait(full);
    isin := true; val := i;
    signal(empty)
  end;
  function get():integer;
  begin
    if not isin then wait(empty);
    isin := false; get := val;
    signal(full)
  end
end;
```

7.9 Java のモニタ機構

□ Hoare のモニタを Java むけに手直し

- モニタ→ 1 つのオブジェクトに 1 つのモニタ (ロック) が対応
- 「synchronized(オブジェクト) {…}」が排他領域
- メソッドの修飾子として synchronized をつけてもよい (そのインスタンスまたは static メソッドならクラスオブジェクトのモニタが使われる)

□ 簡単な例 (金額の更新を排他制御)

```
class Account {
  int bal;
  public calcadjust(...) {
    // amt に調整値を計算
    synchronized(this) { bal += amt; }
  }
  public synchronized adjust(int v) {
    bal += v; // synchronized(this)... と同じ
  }
  int getvalue() {
    return bal; // 読み出すので synchronized 不要
  }
}
```

- もし値が long だとこれは保証されない (Java では long と double は JVM 内部で 2 つ以上の命令に分かれて実行されることを想定)

□ 条件変数→クラス Object でメソッド wait()、notify() を用意→ということは、モニタ 1 つに条件変数が 1 つしかない。結構不便

```
public class Sample38 {
  public static void main(String args[]) {
    BoundedBuf b = new BoundedBuf();
    Thread t1 = new Putter(b, 100, 20);
    Thread t2 = new Putter(b, 200, 30);
    Thread t3 = new Getter(b, 50);
    t1.start(); t2.start(); t3.start();
    try {
      t1.join(); t2.join(); t3.join();
    } catch(Exception e) { e.printStackTrace(); }
  }
}

class Putter extends Thread {
  BoundedBuf buf;
  int start, count;
  public Putter(BoundedBuf b, int s, int c) {
    buf = b; start = s; count = c;
  }
  public void run() {
    try {
      for(int i = 0; i < count; ++i) {
        System.out.println("put "+(start+i));
        buf.put(start+i);
      }
    } catch(Exception e) { e.printStackTrace(); }
  }
}

class Getter extends Thread {
  BoundedBuf buf;
  int count;
  public Getter(BoundedBuf b, int c) {
    buf = b; count = c;
  }
  public void run() {
    try {
      for(int i = 0; i < count; ++i) {
        System.out.println("got "+buf.get());
      }
    } catch(Exception e) { e.printStackTrace(); }
  }
}

class BoundedBuf {
  int[] a = new int[10];
  int ipt = 0, opt = 0;
  public synchronized void put(int v)
      throws InterruptedException {
    while((ipt+1)%10 == opt) wait();
    a[ipt] = v; ipt = (ipt+1)%10; notifyAll();
  }
  public synchronized int get()
      throws InterruptedException {
    while(opt == ipt) wait();
    int x = a[opt]; opt = (opt+1)%10;
    notifyAll(); return x;
  }
}
```

7.10 デッドロックの検出と回避

□ デッドロック (dead lock) とは「死んでしまった (開かない) 錠前」のこと。

□ 簡単に言えば「ロック (ないし一般の資源) の待ち合い」

```
Process_1:  Process_2: Process_3:
  lock a;     lock b;    lock c;
  lock b;     lock c;    lock a;
  //...       //...      //...
  unlock b;   unlock c;  unlock a;
  unlock a;   unlock b;  unlock c;
```

- より定式化すると「待ちに環状の依存関係ができてしまうこと」

□ 検出→待ちの依存関係を調べれば可能

- しかし実際にはこれは結構大変
- 実用的には「ぱたっと止まってしまった」→時間切れ→エラー (→再試行)

□ デッドロックを回避するには???

- その 0: 検出してやり直す→やり直せるとは限らない?
- その 1: 資源を「横取り可能」にする→横取り可能にできるかどうか?

- その 2: 資源を確保する時常に「この先必ず最後まで実行できる経路が存在する」ことを確認→確認するのはかなり大変
- その 3: すべての資源を最初にまとめて確保させる→資源の無駄の可能性
- その 4: 資源に「順番」をつけ、その順番でのみ確保→同上だがややマシ

7.11 楽観的な並行制御

□ 悲観的な並行制御とは→何がどう悪く行っても大丈夫なように考えて行動→ロックを掛けるというのはその代表例

□ 楽観的な並行制御とは→おおむねうまく行くんじゃないかと考えて行動→うまく行かなかったらご破算にしてやり直す (検出機構が必要)

□ トランザクション: 楽観的並行制御の代表

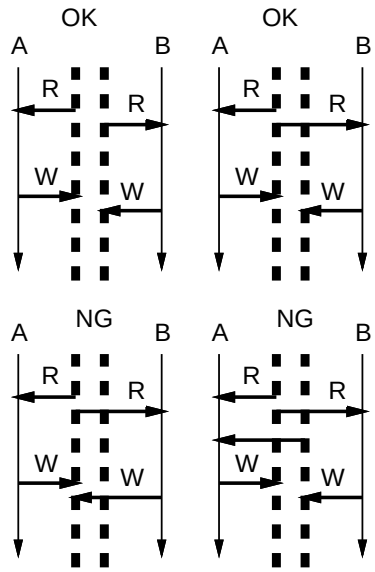
- 重要なデータを参照/更新する作業は必ず「トランザクション開始」と「トランザクション終了」で囲む
- 「トランザクション終了」は commit (成功) と abort (中止) の 2 種類ある
- すべての変更は commit したときはじめて恒久的なものとなる。abort したときは何ら状態に変化は起きない (システムダウン等のときも)

□ トランザクションの実装

- 影のページ方式→変更するデータをメモリ上で用意するが、commit まで書き込まないでおく。abort したらメモリ上のデータを捨てる
- ログ方式→変更する時は前の状態をすべてログに書いておき、abort したらログを見て元に戻す

□ どうやって並行制御?

□ 複数の実行主体による操作が入り混じっていても、「順番にやったのと同等の結果である」(=直列可能) ならばそれは正しい



- 各実行主体が何をどうアクセスしたかを記録しておき検査
- データベース機構としては当然存在するが、プログラミング言語でもトランザクション機能を取り入れた言語はある (例: Argus)。ただし少数派
- 並列/分散用の API/ライブラリにトランザクション機能が含まれている、というのはこれからは主流になると思われる (例: Jini)

7.12 メッセージ送信

□ メッセージ送信→「送った人が送った時点よりも、受け取った人が受け取る時点が必ず後になる」(アタリマエ)→同期機構として利用可能。

- 例: 分配収集型の実行→各分担者がマスターに「終った」というメッセージを送る→マスターは全員ぶんのメッセージが集まったら完了とみなす (実際には結果データもメッセージに入れて送るのが普通)

□ メッセージ送信に基づく言語: さまざまなバリエーションがある

□ メッセージの分類: 同期/非同期 (+未来)

7.13 同期メッセージ

□ 同期メッセージ→送り側が「送り終った」時点で、受け取り側も「受け取り終った」ことが保証されている

- 言語によってはさらに、受け取り側が処理を行い、返値を返す→送り側が返値を受け取れる、というものも
- 分散でない並列言語にとっては、1 つの自然な選択 (手続き呼び出しに類似しているため)
- 代表的なもの: CSP、Ada など

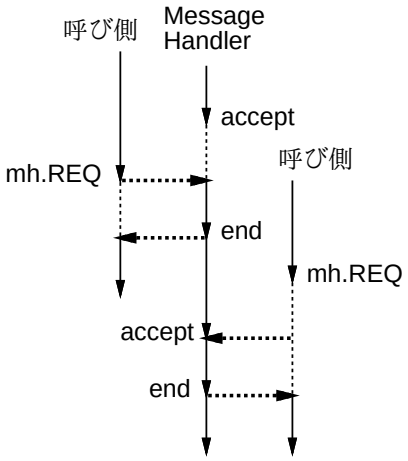
7.14 Adaのタスク機能

□ タスク→並列実行の単位(スレッドのようなもの)

□ エントリ→タスク中のメッセージを受け取るところ(言語的には手続き呼び出しのように見える)

```
task body MessageHandler is
begin
  loop
    accept REQ(M: in STRING) do
      -- メッセージの処理
    end REQ
    --- その他の処理
   end loop
end MessageHandler
----
mh.REQ(...); ←呼び側
```

- 呼び出し側と受け取り側のタイミングが一致しないと、早い側が待たされる
- accept --- end が終るまで呼び側は先に進まない



□ さらに、複数の accept を並行して行える

```
loop
  select
    accept E1(...) do
      ...
    end REQ
    ...
  or
    accept E2(...) do
      ...
    end
    ...
  or
    ...
  or
    delay 0.5*SECONDS;
  else  ↑時間待ち文
    ... ←どの accept も
  end select  呼ばれず時間切れ
end loop
```

□ Ada では「呼び側」も「呼ばれ側」と同様に選択的に呼び出したり時間待ちできる→機能的にはとても充実

- もともとリアルタイム制御のための言語という側面があったため

7.15 非同期メッセージ

□ 非同期メッセージ→送り側が「送り終った」としても受信側はまだまったくそのことを知らないかも知れない。

- いわゆる「メッセージ」のイメージによく合う
- 確認や返事などの機能が必要なら自前で実現する必要
- 面倒なので普通の並列言語ではあまり採用されない

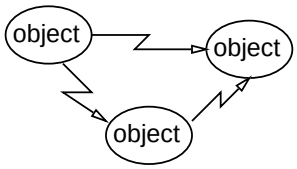
□ しかし、分散環境では非同期が当り前になるわけで…

7.16 並列オブジェクト指向言語

□ Smalltalk-80 のメソッド呼び出し→「メッセージ」

- Smalltalk-80 のメソッド呼び出しは逐次的だったが、これを「並列メッセージ」と解釈することも自然→並列オブジェクト指向言語

□ 並列オブジェクト指向言語とは→各オブジェクトが自律的に並行動作し、メッセージを交換し合いながら計算が進んで行くような言語



- 並列計算のモデルとして Actor、また実際の言語として Plasma、ABCL/1 などがある→実際の言語としてはあくまで研究向け
- ABCL/1 などは同期メッセージもサポート(便利さを重視)

7.17 並列オブジェクト指向言語と継続

□ オブジェクトはあくまでも先頭でのみメッセージを受け取る→ある仕事をしてその返事、というのは結構むずかしい

□ 返事を受け取る手段としての継続(continuetion)→返事を元のオブジェクトに返すのではなく、あるメッセージを送って、受け側の処理が終ったら、その結果を継続オブジェクトに送ってもらう

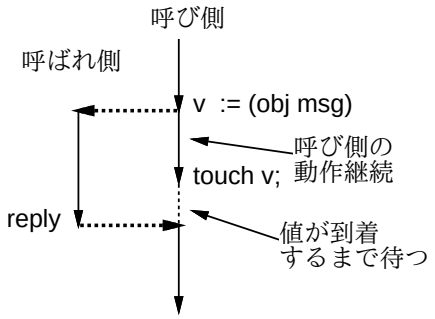
```
(send query: name to OBJ
   reply-to: ((reply: v) => value := value + v))
```

- Smalltalk-80 のブロックのような感じといえばいえる

7.18 未来 (future) メッセージ

□ 便利な形で返事を受け取るためのもう 1 つの方法

□ 同期メッセージと同様に返事を変数に受け取る→実際には返事はすぐにはかえってこない→その変数の値は実際に返事が帰って来るまでは利用できない



- 返事を使うところで touch を実行→返事が到着してなければ待つ
- 返事を使う箇所までは呼び側と呼ばれ側が並行動作可能

7.19 条件同期

□ 同期を取るときには、「このような条件が成立するまで待ち合わせる」ということがしたい

□ モニタでは条件ごとに条件変数を用意して、そこで寝ることで実現できる

- Java のモニタはモニタ 1 つに条件変数が 1 つしかないから面倒なことに…「全員が起きて、条件を調べて、成立してなければ寝る」

□ メッセージによる同期の場合は?→条件に応じて受理するメッセージを変えたい

- 例: 有限バッファなら「満杯のとき」は get()、「空っぽのとき」は put() のみを受理したい

7.20 ガード

□ メッセージの受理部分に書く条件式→ガード。ガードが真であるようなメッセージだけが受理される

□ 例: Ada では select 文の一部としてガード機能が使える

```
select
  when COUNT > 0 =>
    accept GET(V: out ITEM) do
      ...
    end GET;
or
  when COUNT <= MAX =>
    accept PUT(V: in ITEM) do
      ...
    end PUT;
end select
```

- ガードの実装は結構面倒 (なぜだと思いますか?)

7.21 受理集合

□ 受理集合→受理可能なメッセージの集合。受理集合を切替えることで「満杯ならば get() のみを受理」といった制御が行える

- 概念的には単純だが→あらゆるメッセージの on/off を自前で制御する→いわば goto 文のようなもの

7.22 継承異常問題

□ オブジェクト指向→継承機能を使いたい

□ しかし、同期条件を継承するのは難しい

□ 機能を拡張すると、同期条件がすべて変更になり、全部書き直しになりがち

- 例: Ada 流の accept 文→機能が増えると accept 文を全部書き直し
- 例: ガード→機能が増えるとガードの条件を書き直し
- 例: 受理集合→どれかの状態を分割したときに書き直し

□ 書き直しにならないための方法を多くの研究者が提案しているが、まだ決定版はない。

7.23 抽象状態同期 (久野の研究 :-)

□ オブジェクトの状態 == インスタンス変数群の値

□ 抽象状態 == 並行的ふるまいの観点から区別すべき「抽象的な」状態→ 例: 有限バッファなら { full, mid, empty }

□ メソッド入口: オブジェクトや引数の持つべき抽象状態を指定→ 条件に合わないメソッド起動は遅延される

□ メソッド本体: 起動されると抽象状態は「不定」 → 処理結果に応じてメソッド内で抽象状態を新たに設定

```
get = method({full,mid}b:bbuf{mid,empty}, e:element)
  ...
  if b.size > 0 then b!{mid} else b!{empty} end
end get
```

□ 得失

- △状態を明示的に設定するのは煩わしい? → 「抽象状態」を明示的に持つことは利点とも言える

- ○ガードより効率よい実装が可能
- ○継承異常への対応 (集合の細分化、集合の直積など集合演算の利用)

7.24 Java は並列言語か?

□ 広く普及した言語で並列機能をもとから持っているもの → Ada、Java

- ただし、Java では「オブジェクト」と「スレッド」は直交している→ C 言語にスレッドを入れて書くのとたいして変わらない
- より「並列」をきちんと (言語仕様として) 取り込んだ言語が必要では?
- 個人的にはそろそろ「並列オブジェクト指向言語」が実用になって欲しい (しかし現実にはなかなか…)

7.25 この節のまとめ

□ 並列プログラミングにはさまざまな機能や流儀

□ 進歩→次第に言語機構と統合

□ 今後の並列の利用増大→よりよい言語はまだ模索中 (永遠に?)

8 分散プログラミング

□ 並行・並列と分散はどう違うか?

□ 分散だとどういうことが問題になるのか?

□ その問題にはどう対処するのか?

□ 分散とは「複数の実行主体が通信しながら協調動作」

- 従って「嫌でも並列・並行である」
- しかしそれだけではない

□ これまでのプログラミングになかった点…

- 「よその人と協調しないといけない」
- 「どこに何があるか調べないとわからない」
- 「部分的に壊れることがある」

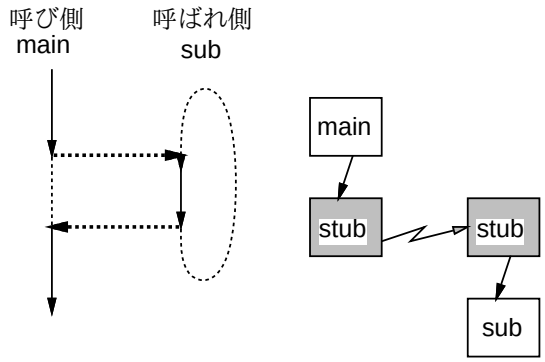
□ ではどうするか?

- 並列プログラミングの蓄積の活用 (しかしそのままでは済まないかも)
- 分散透明← local でも remote でも同じに見える (しかし完全な透明は困難)
- 分散トランザクションの活用

8.1 RPC

□ RPC(remote procedure call) とは…手続きの呼びと戻りをメッセージに置き換えるもの

□ 呼び側→クライアント、呼ばれ側→サーバ



- 呼ばれ側は Ada のタスクで言えば

```
loop
  select
    accept ... end
  or
    accept ... end
  ...
end loop
```

  に相当する無限ループ。
- パラメタや返値はネットワークメッセージへの詰め込み (marshalling) と取り出し (demarshalling) を行う。
- クライアントスタブ→呼び側で「本ものの代わりに呼ばれる」もの
- サーバスタブ (プロキシ) →呼ばれ側で「クライアントに代わって本ものの呼ぶ」もの

□ C 言語と RPC に基づく分散システム→多く使われている (例: NFS)

□ なぜ RPC か?→使い慣れた「手続き」そのままで済む

- その代わり Ada に見られるような細かい制御はない

8.2 分散オブジェクト技術

□ RPC では「平たい」手続きの集まり→構造化しにくい

□ 「オブジェクト指向で」RPC を使うには… → 「遠隔オブジェクト」に対する「メソッド呼び出し」が実は RPC である、というふうにする → 分散オブジェクト技術 (言語はふつうの言語)

□ 代表例: CORBA、Java RMI、HORB

8.3 CORBA

□ 「Common Object Request Broker Architecture」 ← OMG(object management group) という業界団体が管理している規格

□ ORB とは…遠隔オブジェクトの同定や呼び出しを仲介してくれる機能

- オブジェクトを名前で指定 → オブジェクト ID を返す → オブジェクト ID を指定してメソッド呼び出し → PPC による遠隔呼び出し

□ CORBA では呼び側と呼ばれ側が別個の言語であってよい

- C++ binding、Java binding、Smalltalk binding …
- 呼び出しのインタフェースは合わせる必要 → IDL (interface description language) による記述

□ CORBA 製品 → IDL コンパイラ、スタブジェネレータ等の集合

- さまざまな言語が使える… → 本質的にごちゃごちゃ
- OMG ではいろいろな機能を追加している… → さらにごちゃごちゃ

8.4 Java RMI

□ Java に限定した遠隔オブジェクト機能。Sun が設計/実装し、標準で装備されている (JDK 1.1 以降)。

- CORBA より簡略。レジストリサーバが名前解決を行う。
- サーバ側はレジストリサーバにオブジェクト名を登録。
- クライアント側はホスト、ポートを指定してレジストリサーバに接続。
- インタフェースは Java のインタフェースを使う。

□ 例: 簡単な問い合わせサーバのインタフェース

```
import java.rmi.*;

public interface Sample39Interface extends Remote {
  public void accum(String name, int val)
   throws RemoteException;
  public int query(String name) throws RemoteException;
}
```

□ Remote インタフェースを拡張していることで、遠隔オブジェクト用インタフェースであることを表している)。

□ 例: 問い合わせサーバの実装部分

```
import java.rmi.*;
import java.rmi.server.*;

public class Sample39Server
    extends UnicastRemoteObject
    implements Sample39Interface {
  String[] names; int[] vals; int count;
  public Sample39Server() throws RemoteException {
    names = new String[10]; vals = new int[10];
    count = 0;
  }
  public void accum(String name, int val) {
    for(int i = 0; i < count; ++i)
      if(names[i].equals(name)) {
        vals[i] += val; return;
     }
    if(count >= names.length)
      throw new RuntimeException("overflow.");
    names[count] = name; vals[count] = val; ++count;
  }
  public int query(String name) {
    for(int i = 0; i < count; ++i)
      if(names[i].equals(name)) return vals[i];
    return 0;
  }
  public void printall() {
    for(int i = 0; i < count; ++i)
      System.out.println("Name=" + names[i] +
                         " val="+vals[i]);
  }
}
```

□ メソッドの中で先のインタフェースにないもの→ローカルにだけ呼べるメソッドということになる。

□ スタブとプロキシが必要→ rmic コマンドで生成する

```
% javac Sample39Server.java
% rmic Sample39Server
```

□ 例: 問い合わせサーバを起動するメインプログラム

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.*;

public class Sample39Main {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    try {
      Sample39Server s = new Sample39Server();
      Registry r = LocateRegistry.createRegistry(2020);
      r.bind("DB", s);
      System.out.print("command> ");
      while(!in.readLine().equals("quit")) {
        s.printall(); System.out.print("command> ");
      }
      System.exit(0);
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

□ レジストリサーバを立ち上げ、サーバオブジェクトを登録する

□ 立ち上がったら quit と言われるまで状態表示

□ 例: 問い合わせサーバを呼び出す簡単なインタフェース

```
import java.rmi.*;
import java.awt.*;
import java.awt.event.*;

public class Sample39 extends Frame {
  Sample39Interface server;
  TextField t0 = new TextField();
  TextField t1 = new TextField();
  Button b0 = new Button("Query");
  Button b1 = new Button("Accumlate");
  Button b2 = new Button("Connect");
  TextField l0 = new TextField();
  public Sample39() {
    setLayout(null);
    add(t0); t0.setBounds(10, 40, 100, 30);
    add(t1); t1.setBounds(110, 40, 60, 30);
    add(b0); b0.setBounds(10, 80, 60, 40);
    add(b1); b1.setBounds(80, 80, 60, 40);
    add(b2); b2.setBounds(150, 80, 60, 40);
    add(l0); l0.setBounds(10, 120, 380, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          t1.setText(""+server.query(t0.getText()));
          t0.setText("");
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          server.accum(t0.getText(),
              (new Integer(t1.getText())).intValue());
           t0.setText(""); t1.setText("");
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
       try {
          server = (Sample39Interface)
             Naming.lookup("rmi://ホ                ス
ト:2020/DB"); /**/
          l0.setText("connected.");
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
  }
  public static void main(String[] args) {
    Frame f = new Sample39(); f.setSize(400, 400);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) {
        System.exit(0);
      }
    });
  }
}
```

□ RMI の嬉しいところ→普通のオブジェクトとメソッドに見える

□ この例は単独プログラムだったが、アプレットでもよい。アプレットを利用すると「インストールなし」でネットワークアプリケーションが作成できる。

□ アプレットはアプレットをダウンロードしてきたホストとしかネットワーク接続が張れないことに注意。

8.5 オブジェクトの直列化

□ RMI などで渡す引数はどうなっているか?

- 整数や文字列など→パケットへの詰め込みと取り出し
- オブジェクトも渡したい→オブジェクトをバイト列に変換 (直列化、serialize) して転送し、向こう側で復元 (deserialize)
- Java では「implements Serializable」と指定したオブジェクトは標準の直列化機能が使われて転送できるようになる
- インスタンス変数の内容は?→それも再帰的に直列化 (直列化できないオブジェクトがあればエラー)
- 直列化の時に持って行かなくてもよい変数→transient 指定

□ オブジェクトにはメソッドがついているが、それはどうするのか?

- 旧来の言語→向こう側にも同じクラスのコードがないと駄目
- Java では→クラスのコードもネットワーク経由で転送可能
- ただしアプレット等と同様にセキュリティ制約を課される

□ 「オブジェクトを渡す」→つまり「コピーが渡る」→これは嬉しいことか? ふつうはオブジェクトを渡すというのは「参照を渡す」こと

□ Java では「リモートオブジェクトを渡す」場合はその Proxy が渡されるので、その後 RMI で呼び出されるのは「元のオブジェクト」だから参照が渡されることになる。

□ 分散オブジェクトではこのような区別 (コピー渡し vs オブジェクト渡し) が必ず問題になる

8.6 Jini

□ RMI では「ここにあるこういうものを取得してくる」ところからはじまる←「ディレクトリサービス」「ネーミングサービス」

□ 他の分散オブジェクトでも「持って来るものを URI やパス名で指定する」ところからはじまる

- 問題点→「名前で指定したもの」が思い通りのものであるかどうかは分からない。ダウンキャストしようとしたら失敗するかも。

□ せっかく言語に型があるのだから「こういう型/インタフェースのものを欲しい」と言えるようにすれば? → Jini の考え方

□ Jini ではさまざまなサービスを Java の型/インタフェースで表す。

□ 特別なサービス「Lookup サービス」がある (ブロードキャストにより探索)

□ 各サービスは Lookup サービスに対してオブジェクト (Proxy) を登録

□ 登録時に「リース期間」を指定し、期間内に更新し続けないと自動的に削除→古い「ごみ」が残っている心配がない

□ サービス利用側は Lookup サービスに対して「こういう型に合うサービスが欲しい」と言って Proxy を取り出す

- Printer ⊃ ColorPrinter、Printer ⊃ FAX

□ Proxy を取り寄せる← Proxy オブジェクトのメソッドも一緒に取り寄せられて来る

- Proxy は外部と通信しながら自分の役割りを果たす (サービスを提供する)

□ 結局、何が嬉しいか?

- ○サービスの種別をオブジェクト指向言語のクラス階層で表せる
- ○サービスを行なうコードが手元にやってきて仕事してくれる

□ Jini の発表直後には非常に盛り上がったのだが…

- △ Jini の仁義に従う道具だて一式を用意するのが結構大変
- △組み込み機器には JVM を用意するのが結構つらい
- →「これまで通り簡単なプロトコル決め打ちが楽なんでは?」との説も

8.7 Linda と JavaSpaces

□ Linda →ここまでの「相手を決めて通信する」という考えを 180 度覆したパラダイム。1980 年ころからあるが今でも新鮮。

□ 基本的なアイデア→「タプルスペース (TS)」がすべての機器の間を「エーテルのように」満たしている

□ TS にタプル (情報) を出すと、どこからでもそのタプルが取り出せる→ 3 つのオペレーション

- in(...) →タプルを TS に投入する
- out(...) →指定したパターンにマッチするタプルを取り出す (取り除く)
- read(...) → out と同様だが、読むだけで取り除かない

□ 例: RPC の真似をするには?

```
Client:            Server:
  out('calc', 1, 2)  in('calc', ?a, ?b)
  in('result', ?x)   計算…
                     out('result', 結果)
```

□ 例: 時刻サービス

```
Server:              Client:
  while(true) {        read('time', ?t)
    out('time', 時刻)
    次の時刻まで待つ
    in('time', ?x)
  }
```

□ 例: 行列乗算の CPU 数が任意のもの

```
Main:                Client:
  forall i, j {        while(true) {
    out('a', i, j, v);   in('task', ?i, ?j)
    out('b', i, j, v);   d = 0.0;
    out('task', i, j);   forall k {
  }                        read('a', i, k, ?x)
  forall i, j {            read('b', k, j, ?y)
    in('c', i, j, ?v);     d += x*y;
    c[i][j] = v;         }
  }                      out('c', i, j, d);
                       }
```

□ TS を介して「いつやるか」「どこでやるか」を隠蔽

□ 非常に面白い…と言われつつ 20 年…

□ JavaSpaces →それの Java 版 (タプルの代りに Jini のパターンマッチを使う) →これも面白いと言われつつ数年

8.8 分散オブジェクト指向言語

□ ここまでの分散オブジェクトでは「全部オブジェクト」といいながら「ノードに固定したオブジェクト」と「ノード間を渡り歩くオブジェクト」の混合でシステムができていた

□ すべてのオブジェクトは平等でありノード群全体にちらばっている、というモデルの方が美しいのでは?

□ その場合、従来の言語と違ってオブジェクトへの「参照」はポインタではない (メモリアドレスはノードを移ったら無意味)。他のノードにあるものも手元のものも同様に扱える工夫が必要。

□ さらに、分散ごみ集めも必要。

□ 色々研究されているが、実用化はまだまだ…

8.9 分散透明 vs 分散不透明

□ 分散透明→極めて魅力的な考え方

- local/remote に関わらず同じようにプログラムすればよい→プログラム構築が楽
- オブジェクトの置き場所は後でチューニングすればよい→開発中は考えなくても事後に調整できる
- しかし実際には「部分的に失敗 (partial failure)」が起きる→それをどうするのか? (「なかったことにする」ではお仕事のシステムにはならない

□ 説 1 → local と remote はやはり区別するべきである

- 区別するのは嫌だけど実用的には安心

□ 説 2 →至るところで「部分的に失敗」が許されるように作るべきである

- たとえばトランザクションなどを用いて失敗をカバー?

□ この辺はまだ結論の出ていないところだと考える

8.10 本節のまとめ

□ 分散は並列・並行を包含するがそれだけではない

- 巨大な遅延時間、部分的な失敗
- 使えそうな道具だてはいろいろ研究されているが、
- どのように扱うかはまだ確定したものはない

9 第 3 回課題

□ 以下の課題から 1 つ以上を選択してプログラム作成ないし実験を行ない、作成したプログラムが課題を達成していることを論ぜよ。課題中に指定した考察をきちんと書くこと。

□ JavaScript を用いて、自分の Web ページに動きや機能をつけるような「ライブラリ」(つまり自分だけにしか使えないのではなく、他人にあげたら使ってもらえるような形のもの) を作ってみよ。その経験に基づいて、JavaScript によるプログラミングと他の言語 (Java 等) によるプログラミングの違いについて考察せよ。「動きや機能」の例としてはたとえば次のものが考えられる (ただしこれらに限定せず自由にアイデアを出してよい)。

- HTML 要素を動かしたり現われさせたり消したりさせる
- ポップアップ/プルダウンメニュー機能 (サブメニューも出せるとなおよい)
- ページ内でクリックすることでクリックした個所を強調するなどページ内容が加工できるようにする

□ AspectJ の処理系を入手し (入手先は本文中に記載)、本文の例題を動かしてみよ。納得したら次のような拡張を行ってみよ。

- アイコンの色変化も記録/再現できるようにしてみよ。
- もっと他の図形や他の動きも入れたプログラムにして、それを記録/再生できるようにしてみよ。
- その他、AspectJ の機能を活用した例題を新たに考えて作成してみよ。

□ 「インタフェースを用いたフレームワーク」の例題をもとに、複数のボタンを持つことができるように拡張したフレームワークを設計し実装せよ (ボタンの個数は固定でも可変でもよい)。それを用いた複数のアプリケーションも作成すること。

- ヒント: メソッド setData() に、何番のボタンが押されたかを示すパラメタを追加する。getName() も、何番のボタンのラベルを取得するのか指定できるようにパラメタを追加する。
- ヒント: ボタンの数を可変にする場合は、インタフェースにボタンの数を返すメソッドを増やす。また、内部クラス側から外側の局所変数を参照するためには、その変数を final 指定しないといけないことにも注意 (ループ内側でも final 局所変数は定義できる)。

□ (複数 CPU のマシンが使える人むけ) 行列乗算プログラムを動かし、実際に複数 CPU を利用すると速くなることを確認せよ。スレッド起動/終了のオーバヘッドを別に計測し、並列実行する部分がどれくらいあれば複数 CPU を使うことでトータルとして儲かるかの判定基準を検討せよ。

□ 有限バッファの例題プログラムを動かし、どのようなタイミングでスレッドが切り替わっているか検討せよ。複数 CPU の場合と単一 CPU の場合で比較できるとなおよい。また、同期を行なう場合にどれくらい時間が掛かっているか計測する方法を考え、試してみよ。

□ Java でセマフォクラスを作成せよ (各セマフォオブジェクトは p() と v() というメソッドを持つ)。これが実際にセマフォとして動作していることをデモプログラムを作って確認すること。次に作成したセマフォと Java 組み込みのモニタの動作を比較するデモプログラムを作成し、両者の使いやすさや性能について考察せよ。

□ Java で汎用の (どのようなオブジェクトを持って来ても使えるような) ガード機能を持つ排他制御クラスを作成せよ。この問題はやや難しいので考察は自由でよい。

- ヒント： 排他制御を行うクラスは内部に排他制御オブジェクトのインスタンスを保持し、排他制御の必要な区間をメソッド enter() と leave() で囲む。ガードということは、enter() の引数として「自身の状態を検査して OK かどうかを調べる」ようなオブジェクト (内部クラスとして記述できると見やすいだろう) を渡すのでしょうね。

□ RMI の例題をそのまま動かして確認せよ。また、例題ではパラメタは文字列や整数だったが、独自のクラスを渡すのにも挑戦してみよ。やってみて分かったことを整理して示せ。

□ RMI を利用して「行列乗算プログラム」の計算を他のマシンに依頼するようなバージョンを作成してみよ。他のマシンに依頼しても引き合うのは行列の大きさがどれくらいになった時か計測し、引き合うポイントを見積もる手法を考えてみよ。

□ 「Linda 版 CPU 数任意の行列乗算プログラム」を Java で実装してみよ。並列版 (複数 CPU マシンで動かす) でも分散版 (複数マシンで動かす) でもよい。

- ヒント： 結構難しいので好きなように作ってよい。TS オブジェクトは汎用の in や out を作るのは大変なので、このプログラムに出て来る in や out や read のパターンに特化したメソッドだけ持つようにするのがよいと思う。

10 さいごに

□ この講座では、さまざまなプログラミング言語に現れる概念を、その設計思想、用途、実装、特徴、問題点などの観点から整理して見てきたつもりです。

□ 構成としては、1 回目→通常の言語、2 回目→オブジェクト指向、3 回目→オブジェクト指向つづき+並列・分散、のように構成しました。

□ 例年内容が多くて大変なので、今年は重要だと思うことをゆっくり喋ってあとはスキップするようにしてみました。どうでしょうか。

□ ともあれ、おつかれ様でした。