特集

1．Java言語仕様設計上の考慮点

Javaは，高セキュリティ，高能力のフルスケールのプログラミング言語として，ネットワークプログラムに使われてきた．その進展と共に，たくさんの，面白い，普通とは違ったアプリケーションが出現してきた．これらのアプリケーションは，言語それ自身とAPIの2種類のサポートに支えられている．今後も広がるさまざまなアプリケーション領域をサポートするように，Java言語自身に要求されるものは，何なのだろうか？

Cや，C++と比較した時，Javaの良い点として，よく見られるプログラミングエラーをかなり早い段階で，たとえばコンパイル時に，キャッチできるということがある．また，陽なチェックを埋め込むことで，実行時にエラーをキャッチし，そして，プログラムの動作を中断させることなく障害をリポートすることができることがある．

Javaは，コンパイル時に厳密な型分析をして，プログラミングエラーをキャッチできる．ある型を持つとコンパイル時に宣言されている変数に対して，コンパイラはその変数の値がその型に本当に属するものであることを保証する．Cや，C++でのように，プログラマは，キャスト演算子を挿入することで，ある型から別の型へ，式を変換させる指示をさせることができるが，もしJavaコンパイラが，そのキャストが常に正しいものだと検証できなければ，実行時に陽にテストするコードを生成して，型の正しさが保持されるようにする．

また，Cプログラムは割り付けられた記憶領域をフリーにしたのちに，（誤って）そのメモリの内容をそれがまだ持っているポインタを介して変更すると，トラブルに会う．Javaは，自動ガベジコレクションを用いているので，これが起きない．記憶域を，陽にフリーにする方法はない．また，記憶域の内容はすでにそれを参照することがなくなった後でのみ回収する．

ここまでのところ，Java言語の設計のこうした側面は大変うまくいっている．けれども，この言語は大きくいって，3つの領域での拡張の要求を受けている．サブセット，型システム，数値計算である．なぜ現在のJava言語ではこうした目的には不十分だと感じている人がいるのか，なぜ，変更プロポーザルがあるのか，これらのプロポーザルはJavaの全体の設計哲学にどうやってうまくフィットさせるのかなどを見ていく．

Guy L. Steele Jr.
Sun Microsystems Laboratories

訳・編集：井田昌之
青山学院大学

言語設計への3つの拡張要求

1）サブセッティング

■アリゾナ効果？

ある言語仕様を，それ1つだけであらゆるアプリケーションに対して，あらゆるプログラミング上の問題を解決するように設計するのは，困難なことである．Javaは，すべての目的に対してよい言語になるように，と試みてはいない．

アプリケーション全般に対応しようとする言語は，普通私がアリゾナ効果と呼んでいるもので悩むことになる．これは，アメリカ合衆国の初期の頃の話に由来する（この話は歴史的事実，というよりも比喩のために用いている）．その時代，アレルギーとか喘息あるいはその他の呼吸器関係の疾患を持っている人たちには，アリゾナへの引っ越しが有効な治療手段だと考えられていた．砂漠性気候で，また，花粉をまくような植物が大変少なかったからだ．そして，合衆国の他の地域，特に東海岸からたくさんの人たちが，アリゾナに移住した．しばらくそこに住むと，その人たちはその砂漠性気候は無味乾燥なものに思え，今まで住んでいた所を恋しく思うようになった．そこで，花や他の植物を移植して，新しい家もまた，以前の家の懐かしい雰囲気を持つようにしようとした．まもなくその人たちは，昔持っていた問題をまた持つようになる．そして，以前と同じように不幸になるというのである．

■Switch文はよくない？

Javaの設計者達は，古いプログラミング言語の難しい機能を組み込まないよう

にしてきた．もっともそれらは，必ずしも成功してはいない．私自身の好みでいうと，switch文は，パスカルでのcase 文に置き換えられるべき難しい機能だと思っている．けれども，多分全体としてのトレードオフがあり，このことは非常に取るに足らない部分だったのだろう．おそらく，この基本的な文／式を，C言語から大変違ったものにすると，Javaをプログラマが最初に受け入れるのを難しくしたからだろう．Javaは，なるべく小さく，そして，すべての処理系製作者がこの言語に完全に対応させるのを容易にさせるように考えられている．これは，大変重要なことである．つまり，移植性とプラットフォーム独立という2つが，Javaの大変重要なゴールなのである．そしてすべての処理系製作者が言語全体を完全にサポートする時にだけ，そのゴールは達成できる．

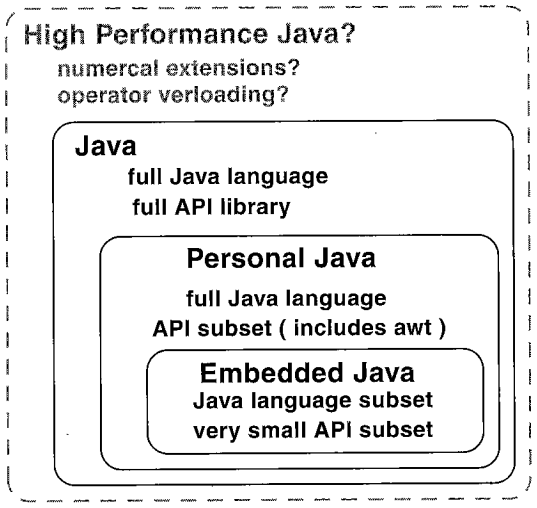


図-1　Java言語仕様の階層

■Javaは大きすぎる？

しかし，小さいはずのJavaでさえもいくつかのアプリケーションに対しては，大きすぎる言語である．たとえば，スマートカードのメモリは小さく，ディスプレイスクリーンも持っていない．この2つの理由で，JavaのAWTウィンドウソフトウェアを，スマートカードが完全にサポートすることを求めるのは，よいことではない．

これは，決して新しい話ではない．私が，X3J11（C言語）の標準化委員会のメンバであった1983年から1984年に，そこでは「I/O関数は，その標準の必須の部分にあるべきかどうか」，という議論をしきりにしていた．それは大変よく使われる機能であり，ほとんどすべてのCプログラマが使っていることが認められるが，同時にI/Oサポートはたくさんのメモリを使う大変複雑な部分であるので，別の代表は常に次のように聞くのである．「エレベータコントローラの場合はどうだい？」

別の言い方でいうと，C言語の重要なアプリケーション領域は組込みシステムであり，そうしたシステムはオフィシャルな標準言語で書くべきだが，その組込み環境では意味を持たない機能をサポートするように要求されるべきではないというのである．

■サブセッティングと仕様の階層

サブセッティングは，大変注意深く扱わなければならない．言語の定義は，移植性に問題を起こすフラグメント化をさけるようにするべきだからである．現在，Java Softでは，Embedded Java（http://java.sun.com/products/embeddedjava/）や，Personal Java（http://java.sun.com/products/personaljava/）を定義している．それらは，組込み環境および，個人用の装置環境に対して，Java言語の機能のサブセットを提供するものである．個人用の装置環境とは，ネットワークベースのモバイル消費者用電子機器などである．これらのものは，すでにJava言語全体を利用しているブラウザや，コマーシャルアプリケーションや，システムプログラムなどに対して混乱を与えない，別のアプリケーション領域としてよく定義されているようである（**図-1**）．

にもかかわらず，注意をする必要がある．それは，2つあるいは3つの別のセットというのは，管理できるが，たとえば，10とか15のセットというのは多すぎる．そのようにプログラミング言語を定義するというのは誤りだと私は信じている．

2）型システム

現在のJavaの型システムはほとんどの部分で成功したといえる．十分に強力であり，プログラムの振舞いに便利な制約を与え，また，多くのプログラムの操作上の誤りを正しくさせ，また，多くのよく見られるプログラミングエラーを実行時にではなく，コンパイル時に検出し，報告することができるようにしている．これによって，プログラマはアプリケーションユーザに見せるのではなく，自分でエラーメッセージを見ることができる．この型システムは，アプリケーションプログラマが毎日効率よく使うように，十分簡単にできている．

■Vectorはあらゆるオブジェクトを要素にできる

プログラミングの際に，Javaの型システムがうまくいっていない部分にしばしば出会うことがある．集合データ構造がその例である．一番簡単な例は，Vectorクラスである．Vectorは，配列を拡張したようなものである．Vectorは，あらゆるJavaオブジェクトへの参

照を中に持つことができる．

けれども実際にはプログラマは，Vectorはある特定の型に対する参照だけを持つようにすることが多い．たとえば，あるVectorは，文字列に対する参照だけを，あるいは別のものは，スレッドに対する参照だけを持つようにする．そのプログラマは，おそらく文字列のVectorの中にスレッド参照を格納しようとするのは，エラーであると考えて，型システムが，そうした問題を解決して欲しいと思っているだろう．また，もしJavaコンパイラが文字列のVectorから要素を取り出す時はいつでも，それは文字列への参照だと理解していてくれると便利だと考えられる．

■Vector要素の型のチェックは？

残念ながら，Javaの現在の型システムは，これらのどちらの仕事もしてはくれない．その結果この2番目の問題の方を解決するには，キャストを用いてプログラムするように強いられる．そして，最初の問題を避けるには，キャストを用いるかあるいは，自分で十分大きな注意を払うことになる．

文字列のVectorの要素を取り出す時には，

```
String s=
myVectorOfStrings.elementAt (3)；
```

の代わりに，プログラマは次のように書かなければならない．

```
String s=（String）
myVectorOfStrings.elementAt (3)；
```

ある要素を文字列のVectorに格納しようとする時，プログラマは，次のようなキャストを書くか，

```
myVectorOfStrings.setElementAt
((String) aThing, 3) ；
```

あるいは，大変注意深くなって，aThingの値が本当に文字列への参照であるかを，別の手段によって確認する必要がある．

キャストを用いる場合の問題は，実行時のオーバーヘッドではない．それは，心配の一番小さな部分である．問題は，コードの中の表現力，そして，コンパイル時に単純なエラーを見つけることに失敗する可能性，ということである．

■型システムの拡張提案

その結果，Javaに対して型システムを拡張する方向でのプロポーザルがいくつか存在している（たとえば，1997POPL，1997PLDI，1997OOPSLAなどのACMの会議の報告集を見よ）．これらは，大別して2つの種類に分けられる．1つは仮想型もう1つはパラメタ型である．パラメタ型のポイントを**表-1**に示す．

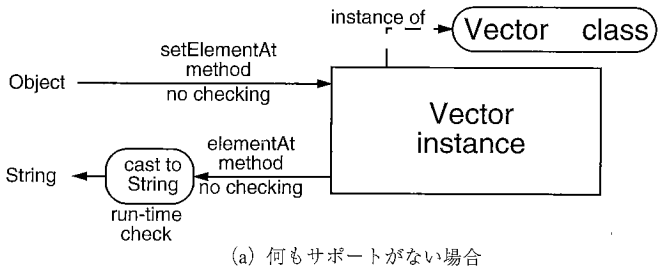
仮想型は大変一般的なものであって，熟練者の手にかかれば，大変柔軟性の高いものである．Betaプログラミング言語で，それらをかなり使ってきた経験がある．けれども，それらは一般的な利用のためには，ちょっと難解である．

パラメタ型は，おそらくもう少しなじみやすいものだろう．たとえば，C++へのテンプレート，あるいは，Adaのgenerics facilityは，この範疇に入る．

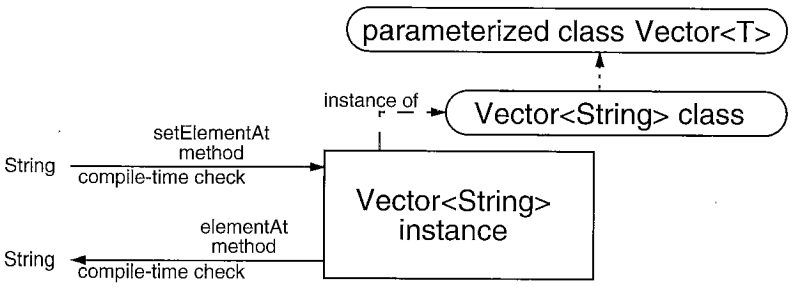
ここまでの議論に関しては**図-2**に示した対比を参照してほしい．パラメタ型の導入により，コンパイル時のチェックが可能となり，同時にcastも不要になる．

表-1 Parameterized Types

- Better compile-time type checking
- Support generic "collection" libraries
- Reduce use of casts
- Similar to C++ templates
- Also support generic numerical types (complex, interval, vector and matrix)



(a) 何もサポートがない場合



(b) パラメタ型を導入する一例

図-2 パラメタ型の導入

おすすめURL集　http://www.javasoft.com/などとは一味違ったさまざまな内容を持っているURLのいくつかを拾い出してみた．その内容についての保証はしないが，少なくともどのようなことが今議論されているかは把握することができる．

(1) Java FAQ What's New（日本語） http://www.webcity.co.jp/info/andoh/java/javanew.html Javaとその周辺の話題についての最新情報へのポインタがほぼ毎日掲載される．新製品，フリーソフトウェアの情報からマスメディアのJava関連情報まで．
(2) Java House Mailing List Homepage（日本語） http://java-house.center.nitech.ac.jp/ml/ Javaに関する話題を対象としたメイリングリストJava Houseのアーカイブ．「トピックス」のページには，Javaプログラミングについてのよくある質問と回答が生の議論へのポインタの形で数百件整理されている．全文検索機能も付いている．
(3) お薦めJava本（日本語）http://www.webcity.co.jp/info/andoh/java/javabooklist.html 日本人読者によって投票されたJava関連書籍一覧．投票によるお薦め度付き．
(4) JavaWorld Book List http://www.javaworld.com/javaworld/books/jw-books-index.html 分野別に整理されたJava関連洋書の一覧．
(6) Java Programing Information（日本語） http://www.ingrid.org/java/ Javaプログラミングに関する総合リンク集．スレッド，国際化対応プログラミングや，Servlet，分散オブジェクトなどの話題についてまとめられている．
(5) JavaWorld - Java Jumps http://www.javaworld.com/javaworld/common/jw-jumps.html 厳選されたJava関連Webページへのリンク集．
(7) じゃばじゃば（日本語） http://www.asahi-net.or.jp/~DP8T-ASM/java/ Javaの基本的なプログラミングのうちつまづきやすいところが重点的に解説されている．オブジェクト指向設計の基礎，コーディングレベルでの最適化，デザインレベルでの最適化などの解説もある．
(8) マルチメディア・プログラミング（日本語） http://www.wakhok.ac.jp/~tatsuo/kougi97/ 稚内北星学園短期大学の植田教授による講義のページ．

高木浩光（名工大）編

■うまく仕様を拡張するには

1997年の12月の時点では，Javaにパラメタ型を採用するかどうか，あるいは，もっと全体的にどんな設計を考慮するか，ということにはなおも，合意がない．設計上の手ごわい考慮点をクリアする必要があるのもやっかいである．次のようなものである．

①それは単純である必要がある．たとえば，文字列へのベクタを参照している変数を簡単に宣言できなければならない．理想的にはその宣言は，単語vector，単語stringそして区切り記号1，2文字くらいですませるべきである．

②それは次の問題を解決する必要がある．たとえば，以下のような宣言が与えられたとしよう．

Vector<String> v = new Vector<String>（20）；

その時にコンパイラは次の型

v.elementAt（3）

が，stringであると判断できるようにするべきである．同様に，そのコンパイラはv.setElementAt（aThing,3）において，aThingの型はString型にアサインできると，判断できるようになっていてほしい．

③それはJava言語の残りの部分にきれいにフィットする必要がある．たとえば，現在のJava言語ではStackクラスはVectorのサブクラスである．Stack<String>をVector<String>のクラスだと期待できると便利だろう．

④その処理系は，メモリ空間や処理時間をやたらに多く必要とするべきでない．

⑤それはコード互換のアップグレードパスがあるべきだ．Vectorクラスを使っている既存のコードがそのまま使えるのが理想だ．

⑥それはJVM互換のアップグレードパスがあるべきだ．パラメタ型を用いているコードが，現在存在するブラウザ上で，実行できるような処理系戦略が存在するはずである（これは，改良されたJVMの処理系はさらによりよいエラーチェックをしたり，よりよい性能を与えるという可能性を禁止していない）．

あるただ1つの設計がこれらすべての制約に対して満足を与えるかどうかは，まだはっきりしていない．もし1つ見つかったとしても次の問題が残る．それを採用すべきかどうかである．私たちは「アリゾナ効果」が我々をおびやかさないようにしなければならない．

表-2　Numerical Computing

· Performance	Compilers
· Functionality	More IEEE support?
· Extensibility	Parameterized types
· Notation	Operator overloading

3）数値計算

■さまざまな希望

Javaを拡張して，浮動小数点演算を多用する科学アプリケーションにも対応させる提案がされてきている．もともとJavaの設計は速度ではなく，互換性と単純性を強調していた．その結果，正しいJavaの処理系は，すべてのハードウェア上で定められた浮動小数点演算結果を与えることができるが，ほとんどすべてのハードウェアには，ある種のちょっと奇妙な部分がある．それが，処理系作成者があらゆる場合に効率よく正しい結果を出すのを困難にしている．

この問題とは別に，Javaでの表現性を高くしようという提案もある．それによって，数値演算コードを書きやすくしたり，保守しやすくするためである．

どんな拡張が必要かどうかはまだよく分からないが，High Performance Fortran（Fortran 95の拡張）の設計がモデルとして役立つかもしれない．

■4つの問題点

数値計算に対して，現在のJavaの設計には次の4つの問題点がある（表-2）．

1. 性能．移植性の点から，Javaはすべての浮動小数点演算に対して，ある特定の再現可能なマシン独立な振舞いをするように求めている．この動作はIEEE754浮動小数点標準に合致しなければならない．さらに実際にはもっと具体的で制限のあるものにしている．というのは，IEEE754はあるいくつかの点では意図的に明確な定義をしていないからだ（たとえば不当な演算に対して生成されるNaNの値，式の中で中間の値が計算される時の正確な精度の規定など）．IEEE754だけでは，多数のハードウェアアーキテクチャの間で99%の互換性があるが，100%の互換性ではない．残りの1%がなおも大きな問題を作り出す．そこで，Javaの現在の設計はある種の妥協である．それは単純で実現性があり，理解もしやすいものだが，Javaの仕様のいくつかの部分は，現在のポピュラーなハードウェアアーキテクチャのすべてでは直接サポートされていないので，したがって，トリッキーなソフトウェアエミュレーションが必要となり，その結果，性能を低下させる．

2. 機能．IEEE754では定義されているが，Javaでは現在サポートされていない機能がある．フラグ情報へのアクセスがその例である．オーバーフロー，アンダーフロー，ゼロ除算などである．

3. 拡張性．数値プログラムでのデータ操作は単純な浮動小数点データだけではない．複素数やベクタ，行列，複素数の行列，有理数，インターバル，などさまざまな数値オブジェクトがある．

4. 表記法．普通のオブジェクト指向でのメソッド記法で十分用を足りているし，多くの場合快適なのだが，同時に数値演算では＋，－，＊，／，不等号など間置記法が一般的である．Javaは単純な整数や浮動小数点演算に対して間置記法を組み込んでる．そこで疑問が生じる．それを他の数値および数学的なオブジェクトに拡張すべきかどうか．これらの点をどのようにして考えるべきかについて，順に見ていく．

・*Javaは将来への道筋を示す．*
・*地球上にはすでに何百万ものコンピュータがある．*
・*Leave Java alone !*

■性能とは？

まず最初に性能について．インタプリタの使用は性能上のボトルネックではないと暗黙のうちに仮定してきた．現在存在する多くのJava処理系はバイトコードインタプリタを用いているが，そうでないものもある．JIT（Just In Timeつまりロード時）のコンパイル技法を用いていたり，伝統的なスタティックなコンパイラもある．1998年の終わりまでにこれらのコンパイラは確かに性能において，最もよいC++コンパイラなどとたちうちできるようなよいコードを生成するだろう．問題というのはJavaの定義と現在のハードウェアアーキテクチャの間のミスマッチをどう扱うかということだ．考え方は次のようなものがあり，議論をよんでいる．

1. ポジションA.

『Javaはいつも将来のための道筋である．』

移植性が最も重要である．そして正確性がまずなければならない．したがって，我々は，Javaを定義されたその通りに実現するべきだ．そして将来のハードウェアの設計はJavaを正しく，そして効率よくサポートするように変化することを含むべきである．

2. ポジションB.

『地球上にはすでに何百万ものコンピュータが使われている．』

今日もまた，毎日も，多数のコンピュータが使われている．それらはJavaの現在の定義をサポートするのは困難である．仕様をちょっとゆるやかにすることは，ほとんどのプログラムにとってはおそらく害がないだろう．そしてそれらをより早く実行させることで，インストールされたハードウェアをもっと効率よく使えるだろう．そのハードウェアはすでに多額の金額の投資がなされているものである．我々は何年も待つべきではない．ハードウェアのベースが置き換えられるまで待つべきではない．そして今，よい性能を持つべきだ．

3. ポジションC.

『Javaにそこまで要求するのか？　大きなお世話だ．ほっておいて．』

すべての処理系が浮動小数点数の複雑な扱いをサポートする必要はない．高性能を必要とするならFortranを使いなさい．

■IEEE754は絶対か

次に，機能．Javaが現在省いているIEEE754の機能なしで書くのは難しいプログラムが存在する．同時に一方で，IEEE754は低レベルの単純な逐次型プロセッサ，アセンブリ言語のレベルを提示していて，隠された状態（副作用）に依存したり，制御構造（そして高水準プログラミング言語の残りの部分との関係）が指定されていないトラップ機構に依存したりしている．副作用依存性は，並列ハードウェアアーキテクチャや，パイプラインアーキテクチャなどの現在の高性能のためのゴールに対して障害となる．オーバーフローなどを検出できるのは望ましいことである一方，IEEE754プログラミングモデルを絶対視するべきだとはいえない．

■再びアリゾナ効果

そこで次に拡張性．アリゾナ！新しい組込みデータ

型をJavaに何十も加えるというのは，確かに望ましいことではない．それは処理系をもっと機能あふれるものにするだろうが，同時に言語のサブセットとして現在の形を保とうという強い要求が出てくるだろう．それは入れようとしている数値計算機能を落とすということである．

複合型のデータ型，たとえばベクタ，行列，複素数，ユーザ定義のインターバルなどをJavaのクラスとして許すようにするとか，また，よいJavaコンパイラがそうしたデータ構造上のオープンコードを可能にしたり，おそらくは多くの場合にヒープの割り付けを避けるようなそうした機能やキーワードを与えるというようなプロポーザルはたくさん研究されている．コア言語をどちらかといえば単純なものに保つ一方で，現在のJavaの処理系と互換であり，サードパーティの数値計算サポートライブラリ市場を開くものとなるようなアプローチも考えられる．パラメタ型はそうした数値ライブラリを開発するのにふさわしいかもしれない．

■オペレータオーバーローディングは却下された

最後に表記法．これは技術的にはすべての中で最も単純なものである．それはすべてコンパイル時に扱うことができる．したがって，特定の実行時のサポートを必要としない．また，技術的には些細なことである．Javaの設計者達はもともとオペレータオーバーローディングを考えていた．そしてそれを却下した．彼らのよく考えた結果の意見というのは次のようなものである．

オペレータオーバーローディングはまったく数値計算コードをクリアなものにするが，同時に非数値演算コードをより読みにくくする．そしてそれは多くのC++プログラマによってひどく悪い方法で乱用されてきた．Javaに対して意図された応用領域では，あまり，数値計算コードを考えてはいなかった．そしてオペレータオーバーローディングをやるにはためらいがあったようだ．

技術的な微妙な点として取り上げたいのは以下のようなことである．ほとんどすべての人は，

a+b

は，次のことを意味すると合意できるだろう．

.a.plus(b)

つまり，オーバーロードされたオペレータは単にメソッド呼び出しの簡略化された形式なのである．これはaの型が参照型である場合に，意味を持つ．また，aもbも参照型でないが共にプリミティブ型であるとき，Javaに現在組み込まれたルールがそうしたケースをカバーする．けれどももし，aがプリミティブ型であって，bが参照型であった場合はどうだろうか？

提案1：そうしたケースは禁止する．これは大変単純で，説明が容易である．不幸なことに，それはよく使われる数学的な表記方法を禁止することになる．和は定数を最後に書くことになる．(a+3)．一方で，積は定数を最初に書くことになる．(3＊v)．古い習慣はなかなか死なないのだ．

提案2：aが参照である場合にa+bはa.plus(b)を意味するようにする．一方，bが参照である場合にはaはbの前に評価される点を除いてb.reversePlus(a)を意味するようにする．提案1より多少よりよいものになる．けれども解決できない別の問題がある．数学的なベクタクラスが定義されていて，その時に行列クラスを定義したい人がいたとする．これは行列をベクタでかけるメソッドを含むことになってくる．m＊vはm.multiply(v)のような何かを意味することになる．けれども，ベクタを行列でかけるというのはどうだろう．この乗算は可換ではない．したがって，我々は常にm＊vと書くことを強要できず，v＊mは何か別のことを意味するようになる．けれども，ベクタクラスは行列クラスを書くプログラマのコントロールの元にはない．行列クラスがv＊mを扱う方法が存在する必要がある．けれども，ベクタクラスがv＊mを本当に処理しようとしたらどうなるだろう？（それは，ベクタを任意のオブジェクトで乗算するようにできるということになっていくのだろうか）．

提案3：これが現在の私の好みである．もしaが参照であり，bがそうでない場合にはa+bはa.plus(b)を意味する．bが参照であり，aがそうでなければ，a+bはb.reversePlus(a)を意味する．もし，a，b共に参照であればa+bは，a.plus(b)とb.reversePlus(a)の両方を意味する．つまり，これら2つのメソッドの1つだけが最も特定的に適用可能なメソッドを持つべきであり，もし両方ともが持っていれば，a+bはあいまいとなり，したがってコンパイル時のエラーとなる．

さらに，これらの提案が議論していないいくつかの点がある．たとえば，ある種の機械では，拡張（80ビット）あるいは4倍（128ビット）精度の浮動小数点型というものをサポートしている．けれども，そうした型をJavaに加え，それを必須の型にすると，それをサポートしていないマシン上では貧弱な性能となり，処理系の頭痛の種となるだろう．そうした型をオプションとして加えることは移植性を破壊させる．どうしたらいいのだろう？

処理系戦略

Java言語が変更されるか，あるいは拡張されるか，にかかわらず，なおも処理系の技術や戦略の改良について，たくさんの部分が残ってる．

言語のサブセッティングに関する論点とは別に，性能やコスト特性が異なる処理系がさまざまに存在する必要がある．たとえば，ソフトウェアあるいはハードウェアに対するリアルタイム応答性など特殊な要求も

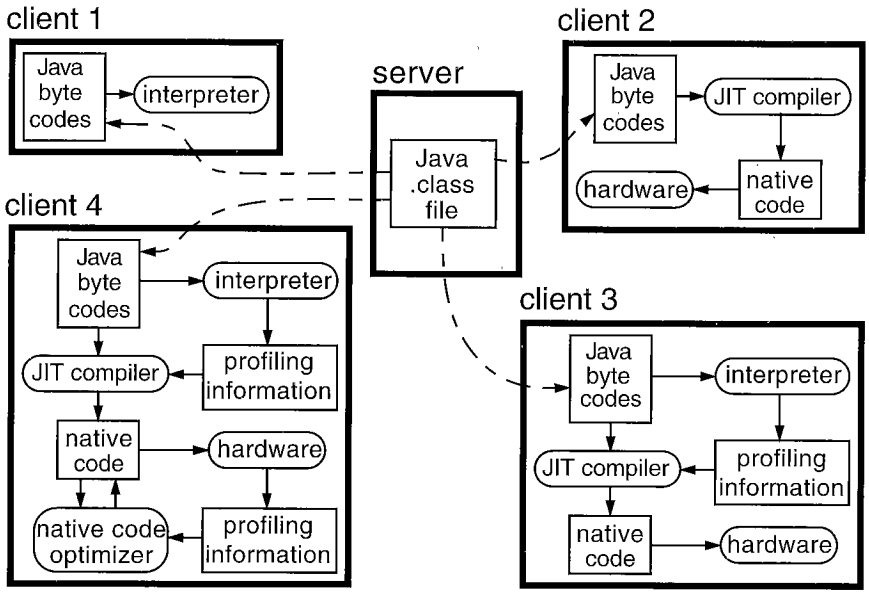


図-3　さまざまな実行形態

いくつかある．単純な例としては，マルチメディアの配達というのは，予測できない制約されない中断時間には耐えられないということがある．それらは単純な記憶管理（ガベジコレクション）の多くのアルゴリズムなども関連する．一方で，スマートカードなどの小さな組込みのシステムなどは，長い休止を避けるような洗練された記憶管理技法をサポートするほど，十分なメモリを持っていないかもしれない．

■on the flyコンパイル

もう1つの重要な研究領域は，「on the fly」コンパイルである．この言葉が意味するのは，単純なJITコンパイルではなく，実行時のコンパイルということである．JITコンパイルはバイトコードを，そのクラスがロードされる時に機械語に変換することである．

実行時のコンパイルには，Javaの実行性能を最大にするのに重要な2つの側面がある．まず第1は，Javaの処理系はコンパイルリソースを割り付ける最適戦略を決定するために，実行時プロフィルのフィードバックを用いることができるということである．これは頻繁に実行されないコードに対する最適化の努力を多大に払わないで済むようにさせる．第2の点は，Javaの処理系は実行時の情報の収集と，キャッシングの戦略を用いて，そのコードの実体の特定の動作の振舞いについて，コンパイルされたコードを最適化できるということである．たとえば，原理的には複数のメソッド定義のどれかを呼び出すような仮想的なメソッド呼び出しを，そのプログラム実行の間で常にある1つの定義だけを呼び出すようにできるかもしれない．コンパイルされたコードはまずそうしたケースに際して，インラインテストを用いることで，賭けをするようにカスタマイズできる．つまり，そのメソッドの差し立てを平均よりももっと速くすることができる．その結果，on the flyコンパイルを用いるJavaコードは，最もよいスタティック・スタンドアローン・コンパイルで生成されたコードよりも速く実行できるかもしれない．

これらを**図-3**にまとめた．client1から4へ順に，高度なものとなる．

■パラレリズム

3番目の領域は，パラレリズムである．Javaはプログラムの中の並列性を移植性のある仕方で陽に提供する，という点で，かなり普通のプログラミング言語とは違ったものになっている．Javaバーチャルマシンの構造の中に，並列性に関するチャンスがあることを指摘しておきたい．

今仮に，JVMは2つの（あるいはそれ以上の）プロセッサを使えるとしよう．それらはユーザスレッドを実行するためにすべて割り付けられるべきだろうか？

おそらく1つのプロセッサは専用にするべきだろう．あるいは少なくともコンパイルの作業に使えるようにするべきだ．また，記憶域管理についてはどうだろうか？ユーザプロセスと同時にガベジコレクタが実行するようなコンカレントガベジコレクションを本当に使うのはなおも難しいことかもしれない．なぜならば，極端に注意深く設計されたハードウェアのサポートなしでは，同期のためのオーバーヘッドは無視できないものになっているからだ．しかし，パラレルガベジコレクタを設計するのは，それほど困難さはないかもしれない．その場合，ユーザプロセスはガベジコレクションの間，停止しなければならない．けれども，ガベジコレクションアルゴリズム自身は，多くのプロセッサを同時に利用できるのである．

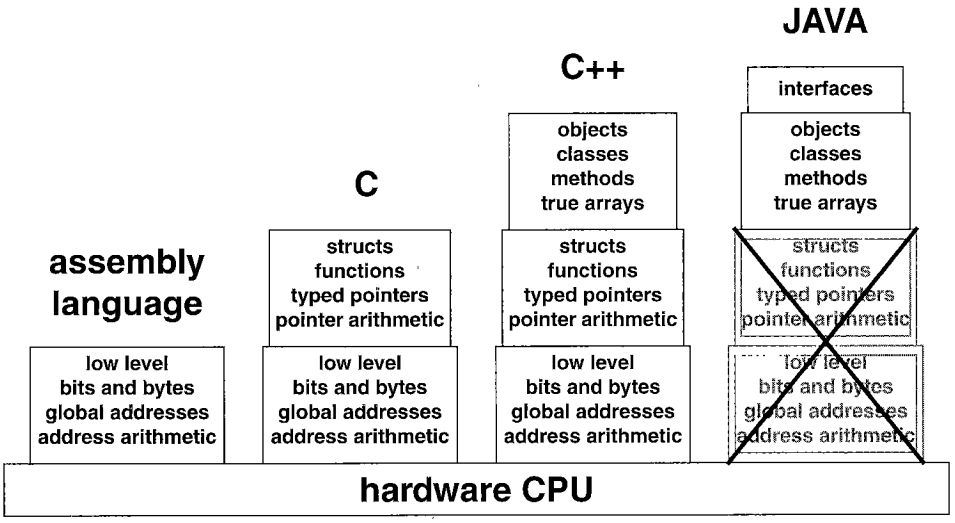


図-4　システムプログラミングの進化

結論

Javaは成功したものだと信じている．そして複雑さをうまく管理できるようにしていると思うので，より広く使われ続けるだろう．古いプログラミング・パラダイムを捨て，注意深く選んだいくつかの新しいパラダイムをサポートすることで，アプリケーションの作成と展開の過程を簡単化した．

システムプログラミング言語の進化を調べることも有益である．以下のようになる（図-4）．

①アセンブリ言語は大変に低レベルだった．その機械が行うことすべてをすることができた．けれどもその結果のコードは抽象度が足りず，また移植性がなかった．その基本的なデータモデルはビットを語に編成したものだった．プログラマはすべてのアドレス計算をコードしなければならなかった．

②C言語はシステムプログラムにおいて大きくステップを進めるものだった．オペレーティングシステム全体を含むたくさんのコードは，おおよそ移植できるものとなった．それは移植性があるというより，容易に移植できると呼ぼう．

C言語はいくつかの便利なデータ抽象化を提供していた．アセンブリ言語のビットの処理はバイトの配列に編成された．それは次に，レコード構造，あるいはバイトではなくデータ項目の要素からなる配列へ自動的にうまく分けられていった．そしてもし必要であれば，常に低レベルのビットを参照することができた．キャストを1つか2つすることでコンパイラはすべてのものにアクセスすることを許した．もちろん，ビッグインディアンとリトルインディアンの機械の両者でうまく動くようなコードを書くのは困難なことだった．またこの言語は必要な秩序を保つ手助けはほとんど与えていなかった．

③C++はレコード構造上に，オブジェクト指向プログラミングをサポートした．これは大変高レベルの抽象度の組み立てを許し，また真の移植性にそれ自身を導くような秩序のあるプログラミングスタイルを許した．けれども，キャストをすると，その下にあるレコード構造を，バイトの配列を，あるいは生のビットなどを，なおも参照することができた．

④Javaは完全に高レベルのオブジェクト指向設計まで上り詰めて，その下のはしごを全部蹴飛ばして捨てたようなものだ．

Javaでは，すべてのものはオブジェクトであって，それと2，3の秩序のあるプリミティブ型だけだ．オブジェクトの生のビットを見ることは許されていない．コードは真に移植性がある．その型システムによって高レベルの抽象化が強いられているので，キャストはあるが限られていて，その抽象度をはみ出すことはできない．コードは安全であり保守も容易である．JVMが，常にビットの組は「参照のように偶然見えるような整数」ではなく，参照であることを知らせるので，自動的に記憶域管理をすることができる．クラスの振舞いは，APIによって，記述される．JavaはAPIの処理系に関する健全な競争型のマーケットを生成してきた．

抽象データ型，オブジェクト，定義済みのインタフェース，自動記憶域管理，などは古いアイディアだが，Javaと共にそれらはより広範に受け入れられ，そして，実社会で毎日，何百，何十万の人々に用いられている．

（平成10年2月16日受付）