データ処理のための プログラミング言語［Ⅰ］
――Java 言語編――

Enjoy Data Processing［Ⅰ］: Java Language

石崎一明

1. はじめに

本稿では，データ処理との関わりを念頭に置きながらJava 言語の特徴を説明する．Java 言語の大きな特徴として，静的型付け(用語)オブジェクト指向(用語)言語，型安全性，言語によるスレッド(用語)サポート，メモリ管理機能，プラットホーム非依存かつ高性能，が挙げられる．これらについて概説した後，データの入出力・処理に関するコードの例とともに，言語の特徴を解説する．最後に，大規模データ処理フレームワークと Java 言語の関係，近年の Java 言語の変化や今後の方向性について触れる．

2. Java 言語概要

Java 言語[1]は，1995 年に Sun Microsystems 社（現 Oracle 社）が発表した言語である．現在も進化を続け，2018 年 12 月時点で，最新バージョンは Java 11 である．

目　　次

石崎一明　日本アイ・ビー・エム株式会社東京基礎研究所
E-mail　ishizaki@jp.ibm.com
Kazuaki ISHIZAKI, Nonmember（IBM Research-Tokyo, IBM Japan, Tokyo, 103-8510 Japan）.
電子情報通信学会誌　Vol.102 No.6 pp.583-587 2019 年 6 月

Java の処理系は，Java Runtime Environment（JRE），Java Developers Kit（JDK），などと呼ばれる．1. で述べた Java 言語の特徴は，2018 年現在では，あって当然，というものが多い．しかし，Java 言語発表当時としては先進的な試みが多く，広く実用的に利用される言語としては初めてとなる機能が採用された．

Java 言語は静的型付け言語であり，基本的に実行前にプログラムで使われる全ての変数の型(用語)が決定される必要がある．これは，最近よく使われている Python や Ruby のような動的型付け(用語)言語と大きく異なる点である．どちらにも利点があり，道具であるプログラミング言語は，実行するプログラムの目的によって使い分けるのがよい，と筆者は考える．

型安全性は，プログラムで使われる変数の型が型システム(用語)の規則に反していない（エラーや例外が発生しない）ことを保証することで，言語のセマンティクスどおりにプログラムが実行され，範囲外へのメモリアクセスなどの不正な動作を行わないことを保証する．Java 言語では実行前プログラム検査と実行時の型や値の検査を組み合わせて，言語の型安全性を保証している．例えば，配列の長さより大きな配列要素がアクセスされた場合，Java 処理系は例外を発生してプログラムの実行を停止する．様々なデータにアクセスするデータ処理のプログラムにおいて，言語が不正メモリアクセスを起こさないことを保証していることは，C/C++ 言語などの安全でない言語に比べてプログラマの負担を軽減できる．

Java 言語は，その言語仕様において Java スレッド間のモニタを用いた同期手法をサポートしている．更に，処理系もマルチスレッド(用語)を用いた場合でも性能のボトルネックが発生しないよう，改良が行われ続けている．また，マルチスレッドを用いた際のプログラミングにおいて，スレッド間で読み書きする値がいつどのようにプログラムから観測可能かを定めた，メモリモデルも

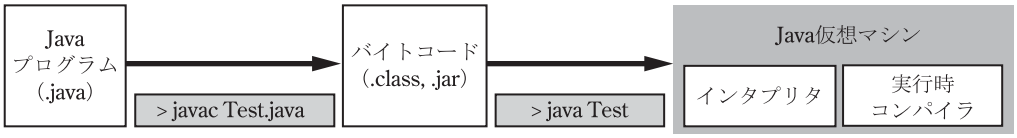

図1　Java プログラムの実行形式

定義されている[2]．これにより，高速かつ安全なスレッドプログラミングが可能である．

プログラマによるメモリ管理を容易にするために，使われなくなったメモリを処理系が自動的に回収するガーベージコレクション（GC）を導入した．この結果，言語仕様から明示的なメモリ解放の操作（C 言語では free 関数に当たる）をなくし，プログラムでのメモリ管理を簡単にしている．今となっては GC を持つ言語が多数であるが，長年の研究成果を基に初めて広く使われるようになったのは Java 言語が最初である．スレッドを用いたプログラムでのメモリ管理，特にメモリ解放は一般には容易ではないが，GC がある Java 言語では容易に実現できる．

Java 言語は，プラットホーム独立であることを念頭に置き，Java 仮想マシン（Java Virtual Machine）上で実行することを前提に設計された．図1に示すように実行形式として，Java 言語で書かれたプログラムは，javac コマンドによって，機種非依存のバイトコードに変換される．Java プログラムは，多くの場合バイトコード形式で class ファイルや jar ファイルによって配布される．プログラム実行時には，バイトコード列が仮想マシン上で実行される．仮想マシン上でバイトコードがインタプリタ実行される．頻繁に実行されるメソッドについては，実行時コンパイラがバイトコード列を対象プロセッサの機械語に変換し，機械語が実行されることによって高性能を達成している．

実行時コンパイラも Java 言語登場以前から研究は行われていた．Java 言語が登場した後，研究開発の速度が急加速した．筆者らのグループによるものを含めて多くの研究論文が発表され，その成果が Java 処理系に実装された．

3. データ処理を行う Java プログラムの実際例

3.1　データ入出力

Java 言語では，基本処理，各種入出力，文字列処理，ハッシュテーブル(用語)などの基本的なコレクションフレームワーク(用語)，並行コレクションフレームワーク，などの豊富なライブラリ群が標準で用意されている．入出力ライブラリでは，Stream と Channel の抽象化された2種類の入出力形式を持つ．歴史的には，初期の段階で Stream が提供され，後に Channel も提供された．

Stream は，Java 言語の最初のバージョンから導入されている Blocking I/O で，様々な種類の入出力を抽象化している．ファイル，標準入出力，ネットワーク，文字列，などを同一の API で扱うことができる．Stream 列には，バイト単位と文字単位の2種類がある．更に，プログラマが独自の入出力ルーチンを作り，Stream として扱うことも可能である．例えば，後で述べる Ha-

■ 用　語　解　説

静的型付け　プログラム実行前に，プログラム中全ての変数，引数，関数の戻り値などの型を決定する手法．

オブジェクト指向　データと操作を含むオブジェクト同士のやり取りを基にプログラムを記述する考え方．

スレッド　プロセッサ上で実行される処理の単位の一つ．プロセスより細かい単位で，各スレッドが独自に保有する内容が少ないので，スレッド間の切り替えが高速にできることが特徴である．Java 言語では，多くの処理系は Java スレッドとカーネルスレッドを1：1対応させているが，ユーザスレッドを用いる処理系もある．ユーザスレッドとはプロセス内で実装されたスレッド機構であり，OS を介さずライブラリなどによってスレッド切り替えを行う点が特徴である．OS カーネルのスケジューラによって切り替えが行われるカーネルスレッドに比べて，高速にスレッド間の切り替えが可能である．

型　プログラムで計算する値の種類（例えば，整数値を示す int や Integer など）．

動的型付け　プログラム実行中に，プログラム中の変数，引数，関数の戻り値などの型を決定する手法．

型システム　型を用いてプログラムがある振舞い（例えば，バッファオーバフロー）を起こさないことを保証する手法．

マルチスレッド　スレッドを複数使用して，一つのプログラムを実行する方法．一つのプロセスの中で複数のスレッドが使用される．実行する CPU が複数のコアを持つならば，同時に複数スレッドを実行可能である．

ハッシュテーブル　キーと値のペアを保持するデータ構造．キーの値に基づいた整数値を添え字とした複数の配列に値を保存する．検索・追加操作を高速に処理可能である．

コレクションフレームワーク　オブジェクトの集合を操作するためのクラスを集めたもの．

オブジェクト　プログラム内で使用するために，クラスを実体化したもの．

型推論　プログラム中の変数や関数の戻り値に明示的に型が記述されなかった場合，最も一般的（汎用的）な型を決定すること．

クラス　オブジェクト指向において，データと操作を含むオブジェクトを定義する方法．

doop の分散ファイルシステムである Hadoop Distributed File System（HDFS）も Stream を用いて入出力処理することも可能である．

Channel は，Java 1.4 から新しい IO の API（NIO）として導入された Non-blocking I/O である．ファイルやネットワークのハードウェアデバイスを抽象化している．高性能を得るために実装され，現在では非同期の入出力 API も提供している．

まず例題として，ファイルを読み込んで，1 単語目をキー・行を値として，逐次的にハッシュテーブルに格納するプログラムを図 2 に示す．ここでは，入出力に Stream を使用した．

Java 言語では明示的なメモリの解放が必要ないので，プログラムを単純に書くことが可能で，メモリ管理が容易である．例えば，6 行目で new で確保して変数 br に保存されたオブジェクト(用語)，変数 line が保持する行の内容を示す String オブジェクト，words が保持する List<String>オブジェクト，は使用終了後 Java 処理系の GC によって自動的に回収される．

標準的な Java 処理系では，各メソッドは，最初のうちは仮想マシンがバイトコードを 1 命令ずつ解釈しながら実行される．実行中に特定された頻繁に実行されるメソッドは，実行時コンパイラによって対象プロセッサの機械語に変換され，高速に実行可能である．例えば，行数が十分多ければ，ループ内の readline()，split()，put()メソッドは，機械語に変換されて実行される．

Java 言語では，これまで変数の型宣言などに関わるコード量の多さが，プログラミングの容易さを阻害する一因として指摘されてきた．Java 8 以降ではコードを簡潔に書くことができる変更が取り入れられている．例えば，変数の宣言を var と書くことで，型推論(用語)を利用して変数の明示的な型宣言を減らすことが可能である．また，6 行目では try-resource と呼ばれる構文を利用して，ファイルに関する終了処理を記述しなくとも，自動的に行う．これにより，正常終了した場合も，例外で異常終了した場合もファイルのクローズなどの終了処理漏れを防ぐことができる．

3.2　データの処理

Java 言語は，言語仕様でスレッドをサポートしており，標準ライブラリはマルチスレッド上でも正しく動作する．また，マルチスレッドを利用して並列実行を容易に記述するために，並列ストリームなどのライブラリも標準で用意されている．これらを用いて，マルチスレッド化された高速なプログラムを容易に記述できる．

具体例として，図 2 のプログラムの各行の処理を並列実行するように書き換える．並列実行する内容は，入力ファイルのうち行の先頭が“Valid”で始まる行について，スペースで区切られた 2 単語以降の小数について二乗和平方根を計算するものとする．その後，単語をキー・計算結果を値としてハッシュテーブルに格納するように変更した．書き換えたプログラムを，図 3 に示す．

ここでは，Java 8 から導入されたストリームとラムダ式を用いてプログラムを記述した．ストリームとラムダ式を組み合わせることで，Java 言語ではデータの並列処理を非常に簡潔に記述することが可能である．8 行目の Files.lines()メソッドで，ファイルの各行に対して生成されたストリームを，10 行目の parallel()で並列に実行することを指示しているだけである．プログラマはスレッドの開始，終了，破棄，などの管理を明示的に行う必要がない．標準ライブラリ内部で，スレッドプールによって管理されている．

ラムダ式においても，型推論が適用されている．この結果，11，12 行目の変数 line の型を記述する必要がなく，簡潔にプログラムを記述することができる．

ハッシュテーブルとして，図 2 で使われていた HashMap の代わりに，標準ライブラリの中で並列アクセス

```
1: import java.io.*;
2: import java.util.*;
3:
4: public class Stream {
5:  public static void main(String[] args) {
6:    var map = new HashMap<String, String>();
7:    try (var br = new BufferedReader(new FileReader("test.txt"))) {
8:      var line = "";
9:      while ((line = br.readLine()) != null) {
10:       var words = line.split(" ");
11:       map.put(words[0], line);
12:     }
13:   } catch (IOException e) {
14:     e.printStackTrace();
15:   }
16: }
17:}
```

図 2 ファイルを読み込んで，逐次的に行をハッシュテーブルに追加する Java プログラム

```
1: import java.io.*;
2: import java.nio.file.*;
3: import java.util.concurrent.*;
4:
5: public class Lambda {
6:   public static void main(String[] args) {
7:     var map = new ConcurrentHashMap<String, Double>();
8:     try (var stream = Files.lines(Paths.get("test.txt"))) {
9:     stream
10:     .parallel()
11:     .filter(line -> line.startsWith("Valid"))
12:     .forEach(line -> {
13:       var words = line.split(" ");
14:       var s = 0.0;
15:       for (var i = 1; i < words.length; i++) {
16:         s += Math.pow(Double.valueOf(words[i]), 2);
17:       }
18:       map.put(words[0], Math.sqrt(s));
19:      });
20:    } catch (IOException e) {
21:      e.printStackTrace();
22:    }
23:  }
24:}
```

図3　ファイルを読み込んで，条件に一致した行の小数について求められた二乗和平方根をハッシュテーブルに追加，を並列に実行する Java プログラム

をサポートした java.util.concurrent.ConcurrentHashMap クラス(用語)を使用している．このクラスは，ハッシュテーブルの読み書きを並列に高速に処理可能である．

8 行目の Files.lines() は Channel によるファイル入力を行っており，ライブラリ内部の実装で最適化が施されている．その結果，並列ストリームと組み合わせて高速な処理が可能である．

4. 大規模データ処理における Java 言語と，最近の動向について

大規模データ処理のための分散ファイルシステムとプログラミングフレームワークは，Google 社の Google File System[3] と MapReduce[4] をきっかけに注目を集めた．論文によると，どちらも C++ を用いて実装されている．オープンソースでは，サーチエンジン Lucene[5] のためのクローラの分散実行・ファイル環境が Google File System と MapReduce を基に 2004 年から実装された．2006 年には，この分散ファイル・実行環境は Apache Hadoop[6] として独立プロジェクトとなった．これらの実装には Java 言語が用いられた．開発者の Doug Cutting は，「実装言語として Java 言語を選択したのは，再利用性のため」，とインタビューの中で述べている[7]．異なるハードウェアやプラットホーム間でのコードの再利用性と動的コンパイルによる性能の高さの両立，ライブラリの豊富さによる再利用性の高さ，が Java 言語を選択した理由と思われる．その後，多くの大規模データ処理フレームワークが Hadoop の MapReduce や HDFS を利用した．その結果，これらのフレームワークは，Java 若しくは Java 仮想マシン上で動く Scala などの言語（JVM 言語と呼ばれる）を使用している．

データ処理では，圧縮・展開処理など一部の処理は C や C++ 言語による最適化された実装を利用することがある．Java では，登場初期から他言語とのやり取りが考慮されており，他言語で書かれた関数を呼ぶ際の規約として Java Native Interface（JNI）[8] が定義されている．JNI を用いて書かれた関数は，同じバイナリーを異なる Java のバージョンで実行可能である．

Java 言語では，マルチスレッドを言語自身でサポートし，実行環境もマルチスレッドプログラムを高速に処理できるように実装されている．マルチスレッドを利用したアルゴリズムを容易に使用できるよう，言語やライブラリによる並列プログラミングに必要な基本的な同期命令，プログラムの挙動を明確にするメモリモデル，などが用意されている．とはいえ，実際にはマルチスレッドを利用したアルゴリズムの高速な実装を短期間で行うことは容易ではない．標準ライブラリに用意されたマップ・リスト・キューなどの実装（java.util.concurrent パッケージのクラス），若しくは優秀なプログラマによって開発された実装，を使用することを開発効率の面から見てお勧めする．例えば，よく使われるハッシュテーブルは java.util.concurrent.ConcurrentHashMap クラスが用意されている．Java のバージョンが新しくなるとともに，実装が改良され，機能も増加している．

最近のプロセッサでは，データ処理などの高速化のためにキャッシュメモリやSingle Instruction Multiple Data（SIMD）命令の効率的活用が必須となってきている．これらをJava言語で実現するために，新しい仕様・実装が議論されている．主な議論をここで三つ挙げる．いずれも，メモリアクセスを効率的に行うことで高速化を目指している．

（1） Project Valhalla[9]：オブジェクトのメモリレイアウト・ジェネリクスの最適化
（2） Project Panama[10]：二次元配列などのメモリレイアウト・他言語とのデータの受け渡しの最適化
（3） Vector API[11]：Java言語からSIMD命令を利用するためのライブラリインタフェース

Sun社によるJava言語の実装は，当初は一般には公開されていなかった．一方，幾つかの機関・団体によって，オープンソースによるJava仮想マシンが発表された．研究目的で公開された有名な実装の一つが，Jikes Research Virtual Machine[12]である．その後，Sun社は自社のJava言語の実装を2006年にオープンソースとして公開した．これは，後にOpenJDK[13]として非常に活発に開発が進められ，現在ではJava言語の参照実装となっている．IBM社も，自社のJava言語の実装を2017年にオープンソースとして公開して，OpenJ9[14]として開発が進められている．

Java言語には，その実装が言語仕様とAPI仕様を満たしているかを確認するための膨大なテストケースTest Compatibility Kit[15]が存在する．このテストによって，Java言語の実装ごとの互換性が保たれている．

Java言語はこれまでメジャーリリースが2〜4年ごと，と他の言語に比べて長く，新機能の取り込みが迅速には行われていない，と言われていた．この意見を踏まえOpenJDKコミュニティでは，Java 11からメジャーリリースを半年ごとに行うように改善した[16]．この改善により，Java言語に新機能がより早く取り込まれることが期待される．一方で，Java言語では，新機能を実装する際にJava仮想マシンになるべく大きな変更を及ぼさないよう慎重に議論を重ね，既存開発者への影響が少なくなるよう考慮している．この慎重な議論と考慮も，Java言語が長く使われて続けている理由の一つであると，筆者は考える．

5. ま と め

本稿では，Java言語の特徴について実際のデータの入出力・処理に関するコードとともに解説した．更に，大規模データ処理におけるJava言語と，Java言語の最近の状況について紹介した．Java言語は，発表から20年以上たっても使い続けられ，進化を続けている．これからもその特徴を守りながらも，プログラミングの容易さと最新のハードウェアを容易に利用できるよう進化を続けていくと，筆者は考えている．本稿を読まれた読者の方々が，Java言語にも関心を持って頂ければ幸いである．

文 献

（1） J. Gosling, B. Joy, G.L. Steele, G. Bracha, and A. Buckley, The Java Language Specification, Java SE 8 Edition, Addison-Wesley Professional, 2014.
（2） “JSR 133 : Java™ memory model and thread specification revision, ”［online］, https://jcp.org/en/jsr/detail?id=133
（3） S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” Proc. the Nineteenth ACM Symposium on Operating Systems Principles, pp. 29-43, 2003.
（4） J. Dean and S. Ghemawat, “MapReduce : Simplified data processing on large clusters,” Proc. the 6th Conference on Symposium on Opearting Systems Design & Implementation, pp. 107-113, 2004.
（5） Apache Software Foundation, “Apache Lucene,”［online］, http://lucene.apache.org/
（6） Apache Software Foundation, “Apache Hadoop, ”［online］, https://hadoop.apache.org/
（7） 五味明子，“Hadoop成功のカギはオープンソースにあり，”［online］, https://gihyo.jp/news/interview/2014/07/1501
（8） S. Liang, Java Native Interface : Programmer's Guide and Reference, Addison-Wesley Longman Publishing, 1999.
（9） “Project Valhalla, ”［online］, https://openjdk.java.net/projects/valhalla/
（10） “Project Panama : Interconnecting JVM and native code, ”［online］, https://openjdk.java.net/projects/valhalla/
（11） “JEP 338 : Vector API（Incubator）,”［online］, http://openjdk.java.net/jeps/338
（12） “Jikes RVM,”［online］, https://www.jikesrvm.org/
（13） “OpenJDK,”［online］, https://openjdk.java.net/
（14） “OpenJ9,”［online］, https://www.eclipse.org/openj9/
（15） “TCK tool and documentation,”［online］, https://jcp.org/en/resources/tdk
（16） 伊藤 敬，“JDK：新しいリリースモデル解説（ver. 2.0），”［online］, https://www.slideshare.net/oracle4engineer/jdk-ver20

（2018年12月11日受付 2019年1月18日最終受付）

石崎（いしざき） 一明（かずあき）

1992早大大学院理工学研究科修士課程了．博士（情報科学）．同年日本アイ・ビー・エム株式会社東京基礎研究所入社．以来，Java実行時コンパイラなどの最適化コンパイラに関する研究開発に従事．Apache Sparkコミッタ．情報処理学会シニア会員．日本ソフトウェア科学会理事．ACM Distinguished Member．