プログラミング言語処理系論 (8)
Design and Implementation of Programming Language Processors

佐藤周行

（情報基盤センター/電気系専攻融合情報学コース）

今日の予定

- VM設計
  - VMをプラットフォームとする「実行」
  - Dispatcher Loop
  - Stack Machineの実際
    - JVM
    - CPython
  - 実際のVMの設計は次回に

Javaのクラスファイルの実行(一番本格的)

- クラスのロードと実行の定義は以下の http://docs.oracle.com/javase/specs/jls/se18/html/index.html
  - Chapter 12. Execution
  - Javaの言語定義の中にある…
- やることは
  - クラスのロード
  - リンク(名前解決を含む)
  - 初期化
  - Go

実行までの手順

12.1. Java Virtual Machine Startup

12.1.1　クラスのロード

12.1.2　リンク、ベリファイ

12.1.3　必要な初期化

12.1.4 mainメソッドのinvoke

「リンク」とは

- Verification
  - 命令がvalid/branchの行き先がvalid/命令が型チェックを通る
- Preparation
  - Staticフィールドの用意
- Resolution of Symbolic References
  - クラス内でクラス名、フィールド名、メソッド名でreferされている箇所を直接referする形に書き換える
- Initialization
  (やることはNativeなものと同じ)

VMでの命令列の実行:特徴づけ

- アーキテクチャの観点から
  - Stack machine vs. Register machine
  - ISAの観点
  - Data segment, Code segment
- 実行時ルーチンの観点から
  - 実行時の型チェック
  - スレッド管理

命令列の実行

- JVM Spec. Chap. 2.11より

Ignoring exceptions, the inner loop of a Java Virtual Machine interpreter is effectively

```
do {
    atomically calculate pc and fetch opcode at
pc;
    if (operands) fetch operands;
            execute the action for the opcode;
 } while (there is more to do);
```

- 普通は、巨大なループになっていて、
  - Interpreter Loop（Dispatcher Loop）
  - この部分に注目したのがPyPyのRpythonのMetainterpreter（次回やります）
- その中で、goto文が跳ね回っている
- クラシックなものではvmgen, CPythonのceval.cでも基本は同じ

アーキテクチャ:
Stack Machine vs. Register Machine

- Stack Machine
  - 各種演算（load/store, arithmetic, …）のオペランドにスタックを想定する
  - スタックに「プッシュする」「ポップする」演算がexplicit/implicitにある
  - フレームなど、スタックで効率よく実装できるものが多くある
  - コードが簡潔になる場合が多い

Stack型VMの典型的な命令

- Load/Store
  - aload n/astore n （frameのn番目の変数をstackにload/store)
- Arithmetic Operations
  - iadd (stack上の上から1番目と2番目のデータをポップして足して、stackの一番上に置く)
- Stack操作
  - pop, pop2, dup, dup2
- 制御命令
- オブジェクト操作
- メソッド呼び出し

opcode　　operand (0 or 1)

Register Machine

- Register Machine
  - 各種演算のオペランドにレジスタ(変数)を想定する
  - ハードウェア命令との親和性が高いと主張する
    - ハードウェアとの親和性を考えるとき、命令体系のレジスタセットとハードウェアのレジスタセットが異なる場合、「Register Allocation」が必要になる
  - レジスタの選択の自由度が上がる分、コードは一般に複雑。

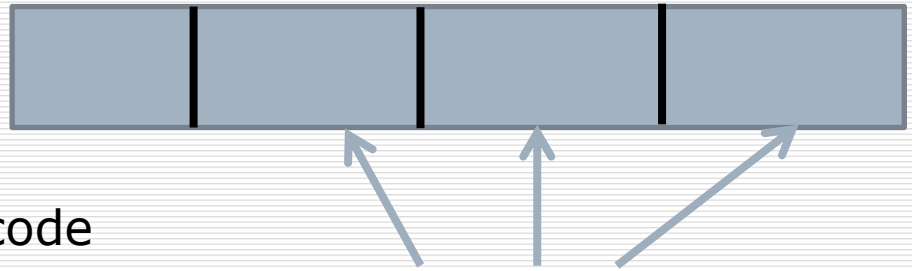
典型的な命令

- Load/Store
  - load address rn/store rn address（アドレスaddressのデータをregisterにload/store)
- Arithmetic Operations
  - add r0, r1, r2 (register r0, r1の値を足してr2に代入する)

ここで、オペランドがstack machineと比較して多く要求されていることに注意する

- 制御命令
- オブジェクト操作
- メソッド呼び出し

- 近年、register machineの実装はみないけど
- HW CPUでは、空間的な制約があり、レジスタ数が有限、命令長が有限
- SW VMでは、その制約がない



喧嘩は往々にして起こるもので

Y. Shi, K. Casey, M. A. Ertl, D. Gregg:
Virtual Machine Showdown: Stack Versus Registers, ACM Transactions on Architecture and Code Optimization, r(4), 2008.

この論文はクラシックなものだけど…

- JVMの命令列からregister machine用の命令列を生成して性能を比較して…
- このグループはvmgenの開発者です

- （課題8）
  JAVA　VMの仕様を読み、以下の観点から整理して特徴を記せ。手っ取り早い方法は、2章を要約することだろう
- JVMの仕様はここ
  https://docs.oracle.com/javase/specs/jvms/se18/jvms18.pdf

（1）VMの命令があつかうデータ
（2）メモリ（スタックとヒープ）管理
（3）スレッド管理
（4）オブジェクト管理

大枠

1 Introduction
2 The Structure of the Java Virtual Machine
3 Compiling for the Java Virtual Machine
4 The class File Format
5 Loading, Linking, and Initializing
6 The Java Virtual Machine Instruction Set
7 Opcode Mnemonics by Opcode

2. The Structure of the Java Virtual Machine

2.1. The class File Format

2.2. Data Types

- 2.3. Primitive Types and Values
  - 2.3.1. Integral Types and Values
  - 2.3.2. Floating-Point Types, Value Sets, and Values
  - 2.3.3. The returnAddress Type and Values
  - 2.3.4. The boolean Type
- 2.4. Reference Types and Values

- 2.5. Run-Time Data Areas
  - 2.5.1. The pc Register
  - 2.5.2. Java Virtual Machine Stacks
  - 2.5.3. Heap
  - 2.5.4. Method Area
  - 2.5.5. Run-Time Constant Pool
  - 2.5.6. Native Method Stacks
- 2.6. Frames
  - 2.6.1. Local Variables
  - 2.6.2. Operand Stacks
  - 2.6.3. Dynamic Linking
  - 2.6.4. Normal Method Invocation Completion
  - 2.6.5. Abrupt Method Invocation Completion

- 2.11. Instruction Set Summary
  - 2.11.1. Types and the Java Virtual Machine
  - 2.11.2. Load and Store Instructions
  - 2.11.3. Arithmetic Instructions
  - 2.11.4. Type Conversion Instructions
  - 2.11.5. Object Creation and Manipulation
  - 2.11.6. Operand Stack Management Instructions
  - 2.11.7. Control Transfer Instructions
  - 2.11.8. Method Invocation and Return Instructions
  - 2.11.9. Throwing Exceptions
  - 2.11.10. Synchronization

2.12. Class Libraries

2.13. Public Design, Private Implementation

データ領域の構成

- 2.5 Runtime Data Areas

- 2.5.1 The pc Register
- 2.5.2 Java Virtual Machine Stacks ← スタック
- 2.5.3 Heap ← ヒープ
- 2.5.4 Method Area
- 2.5.5 Runtime Constant Pool
- 2.5.6 Native Method Stacks

- 2.6 Frames
- 2.6.1. Local Variables
  2.6.2. Operand Stacks
  2.6.3. Dynamic Linking
  2.6.4. Normal Method Invocation Completion
  2.6.5. Abrupt Method Invocation Completion

JVMの特徴は何か

- 単に「Stack Machine」で片づけてはいけない
- Static Data Type Collection
- Specificationで、重点的に読むべきものは何か
  - Load/link – Class fileの扱い
  - OO言語のサポート
    - オブジェクト操作
    - その他、重点を置いている命令
      - スレッドサポート
  - Code/data/stack segmentの定義
    - Runtime Data Area
    - Frame

命令セットアーキテクチャ（ISA）

- VMで扱うデータの型について
  - 対象となる言語によって若干違い
  - 命令をデザインする時に必要
- JAVA VM
  - byte/char/short/int/long
  - float/double
  - boolean/return address
  - reference (class/array/interface)

実際に見てみる

- ISAは、VMの思想を理解する上で重要
- Javaの場合はdisassembleにjavapを使う

- 今回は、javapで遊んでみる
  - javap –c 逆アセンブル
- やれることは、前回アセンブリコードを出力したのとほぼ同じ

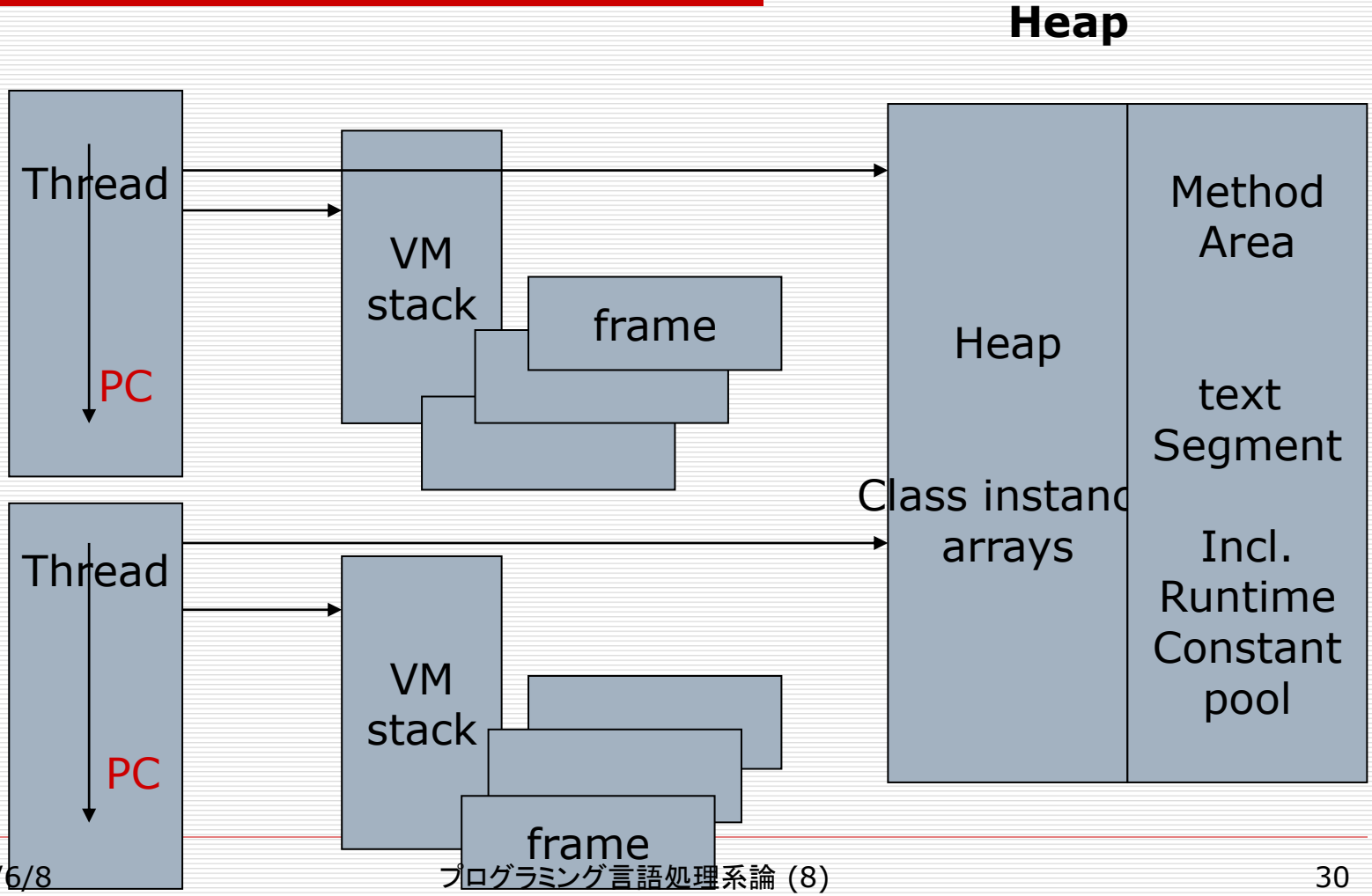
言語的に重要なもの

- オブジェクト, arrayの生成, 操作
  - new
  - field操作 getfield, putfield
  - array access *aload, *astore
- Method invocation
  - invokevirtual
  - invokestatic
  - invokespecial
  - Invokedynamic

VMとして考える必要のあるもの(続き)

- メモリ管理(code/data/stack segment)
  ＝ Runtime Data Area
  - pc Register
  - VM Stack
  - Heap
  - Method Area
  - Constant Pool
  - Native Method Stack
- Frame管理

JAVA VM Overview



JVMのメモリ管理

- JVM Spec §2.5 Runtime Data Areas
  - 2.5.1. PC Register (per thread)
  - 2.5.2. JVM Stacks
    - 通常のマシンのスタックに相当する
    - 定義がアブストラクトなため、いろいろな実装が可能
    - 操作はpush/pop及び、演算系
    - ふつうは、フレーム、ローカル変数、リターンを格納する
    - フレームの扱いはすぐ後で

JVMのメモリ管理(cont'd)

- 2.5.3 Heap
  - Class Instance
  - Array
  - Memory管理はGCで行う
    - Objects are never explicitly deallocated
- 2.5.4. Method Area
  - Per class structure

JVMのメモリ管理(cont'd)

- 2.5.5. Run-Time Constant Pool
  - Per class/interface runtime representation of the constant_pool table.
  - Constants
  - Method and field references (Will be resolved at run-time
  - Class/Interface がJVMによって作成されるごとに作成される
- 2.5.6 Native Method Stacks

- Nativeな環境でのプログラム実行に用意されている概念はだいたい用意されている
  - PC <-> PC
  - Stack <-> Stack
  - Heap <-> Heap (with GC)
  - Text Segment <-> Method Area
  - Symbol Table + RODATA <-> Runtime Constant Pool

Runtime Constant Poolのアクセス

- ldc (load constant)でアクセスする
  - 小さい定数は、コードの中に直接埋め込むことができる
  - 大きい定数や文字列は、Constant Poolに格納していちいちロードするしかない
- シンボルは、(数字も含めて)コンパイル時にresolveされている

Nativeな環境での例外…

- OSの腕の見せ所
- シグナルに関するC interfaceが用意されているのが普通

クラスファイルの管理

- クラスファイルのロード
  - 実は、Classとして用意されていて…
    - Java.lang.ClassLoader
      - getsystemClassLoader
      - getClassLoader
      - …
    - ファイルからのロード、ネットからのロードその他に対応

フレームの管理

- フレームの大きさはコンパイル時に決定する
- SPやBPのセット等はInvoke時になされる
- 引数はどう渡されるのか？
  - *load, *storeでアクセスできる「local variable」に引数が渡される
  - Local variableは、フレームの一部であるが、スタック上に作る必要はない！？
  - ヒープにlocal variableを作ってもよい（JVMのデザインの問題）

フレームに格納されるデータ

- リターンアドレス (*returnで使用)
- local variable (*load/*storeで使用)
- runtime constantへのreference
- Exception Table

Java VM Frameについて

- Frameに入っているもの
  - Local variables
    - Parameterを含む
    - Local var 0 はmethodを呼び出したオブジェクトをさす
  - Operand stack
  - Reference to the runtime constant pool (dynamic linkを可能にするため)

- Frameの生成・破壊
  - Methodが呼び出されるたびに生成される
  - Methodが「complete」すると破壊される

- Operand stackって？
  - JAVA VMの命令はスタック上にあるオペランドを操作する
    → これがスタックマシンのいわれ
- Reference to a constant poolって？
  - Dynamic Linkingのサポート

Dynamic linking translates these symbolic method references into concrete method references, loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the runtime location of these variables.

Calling Convention

- 関数呼び出しの制御をするには、呼び出し側と呼び出され側で引数をどう扱うかについての合意が必要だった
- JVM chap. 3.6 & 3.7
  - Static とvirtualでの違い
    - 第0引数にthisがくるかどうか
- → calling conventionを実際に観察する

コンパイルしてみる

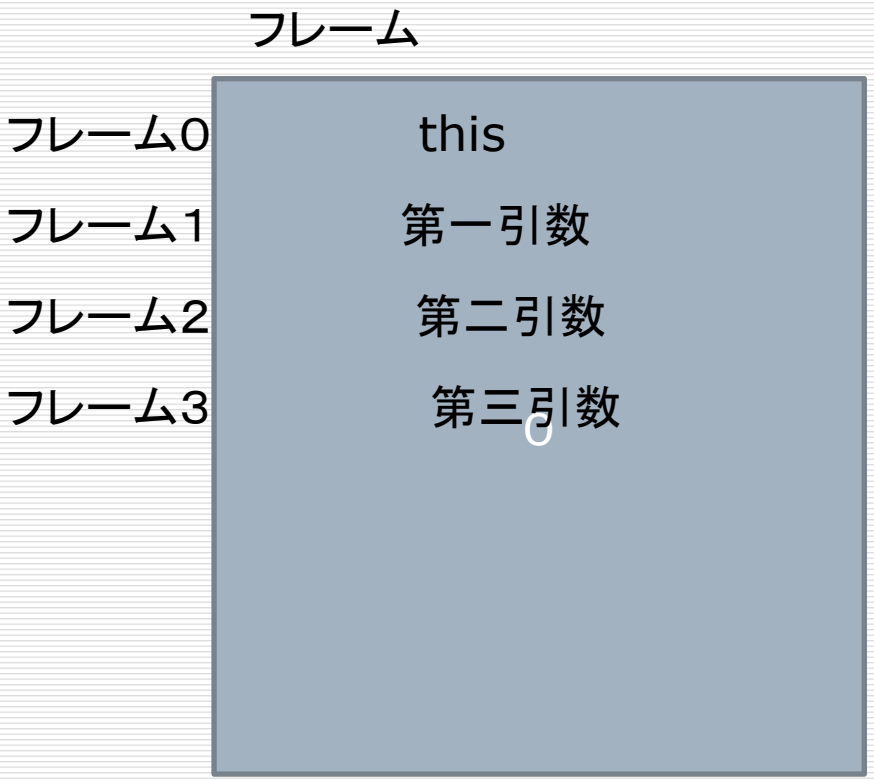
```
class b {
    static int content;

    b(int y) {
        content = y;
    }

    public int get_content(int z) {
        return content+z;
    }

    public void addxx(b x1) {
        int y;

        y = x1.get_content(100000);
        content += y;
    }
}
```



iload_n, aload_n, istore_n, aload_n等の命令が
フレーム内のデータにアクセスするために用意されている

コンパイル結果

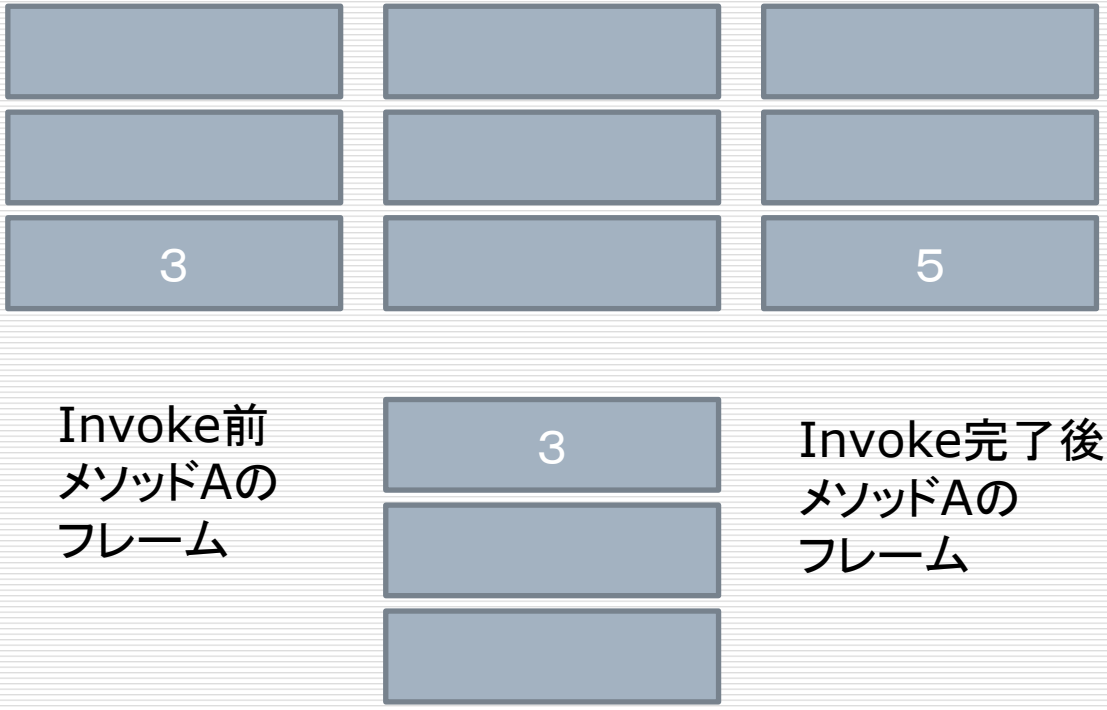
```
class b {
  static int content;

  b(int);
    Code:
       0: aload_0
       1: invokespecial #1                  //
        Method
        java/lang/Object."<init>":()V
       4: iload_1
       5: putstatic     #2
       8: return

  public int get_content(int);
    Code:
       0: getstatic     #2
        1: iload_0
        4: iadd
       3: ireturn

  public void addxx(x);
    Code:
       0: aload_1
       1: pop
       2: sipush     10000
       5: invokestatic  #3
       8: istore_2
       9: getstatic       #2
       // Field content:I
       12: iload_2
       13: iadd
      14: putstatic     #2                 //
       Field content:I
      17: return
}
```

Local Variable

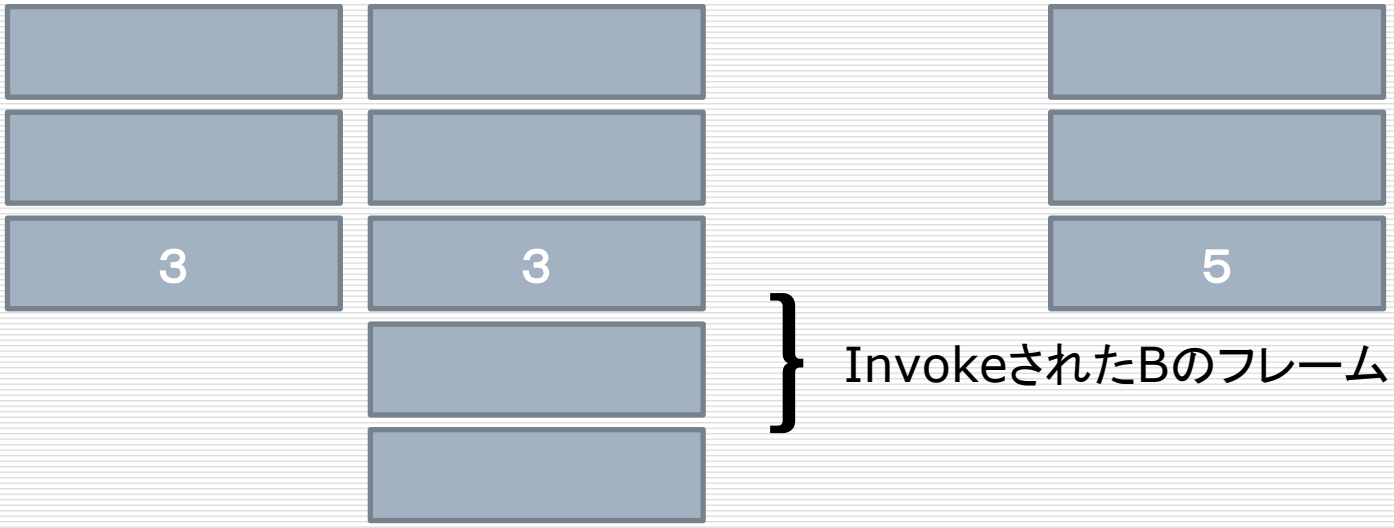


Invoke前
メソッドAの
フレーム

Invoke完了後の
メソッドAの
フレーム

Invokeされた時の
AとBのフレーム

ふつうはJVM　Stackをきれいに使う



Controlについて

- Exception Handling
  - 以下のプログラムを見てみる

```
class c_a_t {
   double throwOne(int i, int j) throws ArithmeticException {
     return i/j;
  }
void catchOne(int i) {
try {
    throwOne(3,i);
} catch (ArithmeticException e) {
    throwOne(100000, i+1);
}
}
}
```

Exception Tableの管理

- Class fileを作る時にException Tableに関する属性が決められる

from to target type

0 7 10 Class java/lang/Arith…

(0≦PC<7の時に例外が発生したら10に跳べ)

Concurrentyへの対応

- Synchronized
  - Monitorの導入
  - 特別なExceptionの対応
  - では、以下のコードを見てみましょう

```
class sync {
    void onlyMe() {
    synchronized(this) {
        System.out.println("abc");
    }
    }
}
```

- (課題8′)ceval.c を以下の観点から要約せよ。要約は、JVMの整理の仕方に倣ってみよ(以下のスライドで概要を示す)JVMの2章程度の内容でよい。
  - CPython VMの仮想機械としての構造
  - Pythonの実行をサポートするためのフレーム管理、メソッド呼び出し、ループ実行等の命令セットの特徴

PythonのVM

- JVMと同じ視点からの解析をしてみる
  - Note: PythonのVMは基本undocumented
- VMで扱うデータ型
- Python のサポート
- Data Areaへのアクセス
- Calling Convention

Pythonの解析ツール

- import dis
- Disassembleはdis.dis()でできる

データ型

- Javaのようなstrictな型チェックをコンパイル時にしない
  - Binary addはNumberを対象に (int/float/complexは実行時にチェック)
- データ型
  - Number
  - Iterator
  - Sequence
    - list, tuple, range
  - Text Sequence (=string)
  - Binary Sequence
  - Set
  - Mapping (Dictionary)
  - Context manager

ceval.c中のBINARY_ADDの部分

```
case TARGET(BINARY_ADD): {
        PyObject *right = POP();
        PyObject *left = TOP();
        PyObject *sum;
        if (PyUnicode_CheckExact(left) &&
                PyUnicode_CheckExact(right)) {
            sum = unicode_concatenate(tstate, left, right, f, next_instr);
            /* unicode_concatenate consumed the ref to left */
        }
        else {
            sum = PyNumber_Add(left, right);
            Py_DECREF(left);
        }
        Py_DECREF(right);
        SET_TOP(sum);
        if (sum == NULL)
            goto error;
        DISPATCH();
    }
```

- Objects/abstract.c中のPyNumber_Addがこれがまた…
  - Objectの持つ演算の取り出し
  - Subclassのチェック
  - float_add, long_addで必要に応じてまた型変換
    - float_add (floatobject.c)
    - long_add → x_add (longobject.c)

Pythonの言語サポート

- Coroutine関係
  - get_awaitable, get_aiter, …
- Build_(データ型)
  - build_{tuple, list, set, map, iter, …}
- Iterationの実行
  - get_iter, get_aiter
- 特殊なデータ型操作のサポート
  - map_add, set_add, list_append, …
- Binary_matrix_matrix (!)

Data Areaへのアクセス

- load_const
- load_name
  - Nameにassociateされた値のロード
- load_attr
  - Fieldへのアクセス
- load_global
  - reference by name
- load_fast
  - Frame上のvariableへのアクセス
- load_closure
- load_deref
- load_classderef

Calling Convention

- Caller側:
  - stack上 0: function reference (push)
  - stack上 1—n: 引数 (push)
  - Call_function (n)
- Callee側
  - load_fast 0—(n-1): 引数へのアクセス (frame)

Concurrency関係

- Thread based concurrency
- 実際は、ループの非同期実行 (coroutineで間に合うという強弁が…)
- Critical sectionの管理をgilで行う
  - threadが並列に動かない
- VM(プロセス)を複数持って管理

関係する命令

- GET_AITER, GET_ANEXT
- BEFORE_ASYNC_WITH, BEGIN_FINALLY, END_ASYNC_FOR, SETUP_ASYNC_WITH
- GET_AWAITABLE

- Compileでは、decorator (@asyncio.coroutine) またはqualifier (async) で指定

次回に続く

- CPythonは、ソースが解析できるので、
  - Dispatch Loopの全貌
  - 各命令の処理のしかた

が全部見えます

  - せっかくだから、次回少し見てみましょう
- で、JVMのように、仕様がしっかり書かれているものと、CPythonのように、ソースをみないとわからないものとどちらが勉強になるでしょうか？
- （そんなことより、ツールを使ってVM作ろうぜというのは、正しい方向だろう）