ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 9

Імплементація ООП (завершення)

Весна 2021

Автоупаковка і розпакування

У C++ ви можете оголосити вектор **int**, якщо вам це потрібно, написавши

```
vector < int > intVec;
```

У Java неможливо оголосити **ArrayList** з **int**. Шаблони на C ++ можуть приймати будь-який тип як аргумент класу, навіть примітивні типи. У Java лише класи можуть служити аргументами для загальних джерел. Тип **int** - примітивний тип. Це означає, що це не клас. Однак творці Java зрозуміли цю проблему і надали класи обгортки для кожного з примітивних типів, щоб ми могли оголосити колекції **ints**, наприклад, обернувши кожен **int** як ціле число.

Отже, хоча ми не можемо оголосити **ArrayList** з **int**, ми можемо оголосити **ArrayList** з **Integer** наступним чином.

```
ArrayList<Integer> intList = new ArrayList<>();
```

Обгорнутий int створюється та додається до нашого списку наступним чином.

```
int x = 6;
Integer y = new Integer(x);
intList.add(y);
```

І повернення його знову вимагає написання такого коду.

```
x = intList.get(0).intValue();
```

Це старий спосіб обтікання та розгортання цілих чисел та інших примітивних типів у Java. Знову ж таки, програмісти роблять це досить часто, щоб їм хотілося отримати більш компактний спосіб загортання та розгортання примітивних типів. Програмісти Java позначають це як бокс і розпакування. Останні версії Java підтримують автоматичний бокс і розпакування. Отже, тепер на Java ви можете написати наступне.

```
int x = 6;
intList.add(x);
...
x = intList.get(0);
```

Значення змінної **x** автоматично вставляється в поле, коли воно додається до списку, і автоматично розпаковується під час вилучення. Java визначає, коли встановлювати та розпаковувати примітивні значення на основі типу значення та викликаного методу. Синтаксис набагато компактніший і легший для читання.

Обробка винятків у Java та C ++

Java та C ++ можуть створювати винятки та ловити їх, як і в багатьох мовах. Іноді винятки містяться в коді, який ми не пишемо, а використовуємо код. Наприклад, індексація після кінця вектора. В інших випадках ми можемо побажати зробити виняток.

У C ++ може бути використано буквально будь-який тип значення. На рисунку 9.1 показано, як за допомогою C ++ створюється об'єкт, який називається **`PyException`**. Цей код взятий з модуля **`PyRange.cpp`**. Коли **`indexOf`** викликається за кінець об'єкта діапазону, CoCo кидає об'єкт **`PyException`** зі значенням ітерації зупинки, як показано на рис. 9.1.

```
PyObject* PyRange::indexOf(int index) {
    int val = start + index * increment;
    if (increment > 0 && val >= stop) {
        throw new PyException(PYSTOPITERATIONEXCEPTION,"Stop Iteration");
    }
    if (increment < 0 && val <= stop) {
        throw new PyException(PYSTOPITERATIONEXCEPTION,"Stop Iteration");
    }
    return new PyInt(start + increment*index);
}
```

Рис.9.1 Викидання винятку в C ++

```
1       public PyObject indexOf(int index) throws PyException {
2           int val = start + index * increment;
3           if (increment > 0 && val >= stop) {
4               throw new PyException(
5               PyException.ExceptionType.PYSTOPITERATIONEXCEPTION,
6               "Stop Iteration");
7           }
8           if (increment < 0 && val <= stop) {
9               throw new PyException(
10              PyException.ExceptionType.PYSTOPITERATIONEXCEPTION,
11              "Stop Iteration");
12          }
13          return new PyInt(start + increment * index);
14      }
```

Рис.9.2 Викидання винятку в Java

```
case FOR_ITER:
    u = safetyPop();
    args = new vector<PyObject*>();
    try {
        v = u->callMethod("__next__", args);
        opStack->push(u);
        opStack->push(v);
    } catch (PyException* ex) {
        if (ex->getExceptionType() == PYSTOPITERATIONEXCEPTION) {
            PC = operand;
        } else
            throw ex;
    }
    try {
        delete args;
    } catch (...) {
        cerr <<
         "Delete of FOR_ITER args caused an exception for some reason." <<
         endl;
    }
    break;
```

Рис.9.3 Перехоплення винятку в C ++

```
case FOR_ITER:
    u = this.safetyPop();
    args = new ArrayList<PyObject>();
    try {
        v = u.callMethod("__next__", args);
        this.opStack.push(u);
        this.opStack.push(v);
    } catch (PyException ex) {
        if (ex.getExceptionType() == ExceptionType.PYSTOPITERATIONEXCEPTION) {
            this.PC = operand;
        } else {
            throw ex;
        }
    }
    break;
```

Рис.9.4 Перехоплення винятку в Java

Викиди, які виникають, можна перехопити, а версію винятку C++ перехопити в **`PyFrame.cpp`** в інструкції **`FOR _ITER`**. Код цього наведено на рис. 9.3. Щоб уловити виняток, його потрібно кинути в блок **`try`** або в якийсь код, викликаний із блоку **`try`**. Тоді тип цінності, зловленої в улові, повинен відповідати типу викинутої вартості. На рисунку 9.4 наведено версію Java для вилучення винятку. Немає великої різниці між C++ та Java у обробці винятків. Додатковий код на рис. 9.3 у рядках 14–20 потрібен, оскільки C++ не має збору сміття, тоді як Java має.

Рисунки 9.1 та 9.3 демонструють, як обробку винятків можна використовувати для реалізації ітерації в інтерпретаторі CoCo, тоді як Рис. 9.2 та 9.4 забезпечують версію Java для JCoCo. Коли досягається кінець ітерації, видається виняток ітерації зупинки, і коли вона потрапляє, вона сигналізує про закінчення ітерації.

Обробка винятків - це засіб обробки умов у межах програми, незалежно від того, планується вона чи не планується. Програми на C++ та Java можуть за потреби викидати та / або ловити винятки. Однак деякі проблеми в C++, такі як поділ на нульові помилки, не виникають як винятки. Заміна їх сигналами є наступною темою.

Сигнали

Версія C обробки винятків - це обробка сигналів. Програми C можуть генерувати сигнали, але частіше встановлюють обробник сигналів на місце обробки сигналів, генерованих операційною системою. Малюнок 9.5 містить витяг коду з **`main.cpp`**, де реалізований обробник сигналу та встановлений в **`main`**.

```
void sigHandler(int signum) {
  cerr << "\n\n";
  cerr << "************************************************************" << endl;
  cerr << "             An Uncaught Exception Occurred" << endl;
  cerr << "************************************************************" << endl;
  cerr << "Signal: ";
  switch (signum) {
      case SIGABRT:
          cerr << "Program Execution Aborted" << endl;
          break;
      case SIGFPE:
          cerr << "Arithmetic or Overflow Error" << endl;
          break;
      case SIGILL:
          cerr << "Illegal Instruction in Virtual Machine" << endl;
          break;
      case SIGINT:
          cerr << "Execution Interrupted" << endl;
          break;
      case SIGSEGV:
          cerr << "Illegal Memory Access" << endl;
          break;
      case SIGTERM:
          cerr << "Termination Requested" << endl;
          break;
  }
```

```
27    cerr << "-------------------------------------------------------" << endl;
28    cerr << "               The Exception's Traceback" << endl;
29    cerr << "-------------------------------------------------------" << endl;
30    for (int k=callStack.size()-1;k>=0;k--) {
31        cerr << "==========> At PC=" << (callStack[k]->getPC()-1) <<
32        " in this function. " << endl;
33        cerr << callStack[k]->getCode().prettyString("",true);
34    }
35    exit(0);
36  }
37
38  int main(int argc, char* argv[]) {
39    char* filename;
40
41    signal(SIGABRT,sigHandler);
42    signal(SIGFPE,sigHandler);
43    signal(SIGILL,sigHandler);
44    signal(SIGINT,sigHandler);
45    signal(SIGSEGV,sigHandler);
46    signal(SIGTERM,sigHandler);
47    ...
```

Рис.9.5 Обробка сигналів

Існує кілька типів сигналів, і код на рис. 9.5 написаний для вловлювання всіх сигналів, визначених стандартом C. Постійні типи сигналів визначаються у включенні, яке називається **`signal.h`**. Коли сигнал генерується, програма негайно переходить до обробника сигналу, передаючи йому значення сигналу, яке було сформовано. Обробник сигналу зазвичай пишеться для повідомлення про якийсь тип помилки, а потім припиняється. Обробник сигналу, представлений на рис. 9.5, робить це. Він друкує відстеження програми, а потім завершує.

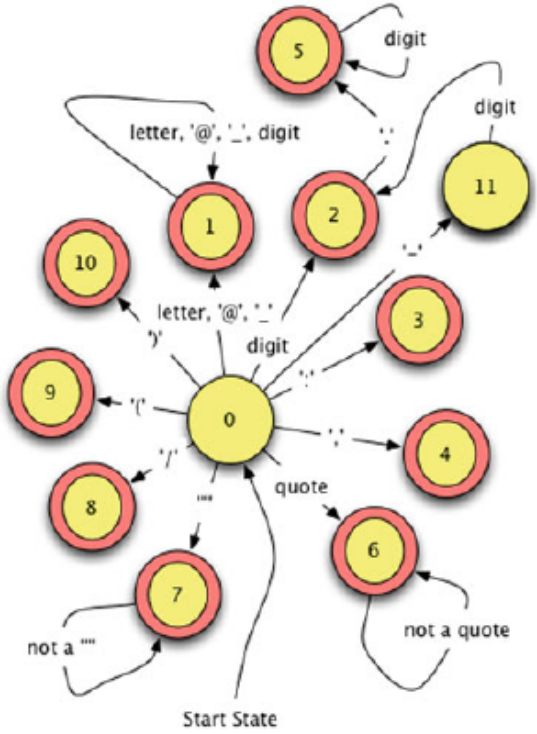
Глибоке JCoCo

Решта лекції стосуватиметься лише реалізації JCoCo на Java. Багато подібності існує із реалізацією C++ CoCo. JCoCo - це здебільшого надмножина реалізації CoCo. Якщо існують відмінності між двома реалізаціями, вони будуть зазначені. Насамперед JCoCo додає підтримку для створення програмованих класів.

Сканер

Віртуальна машина JCoCo зчитує файл CASM, як показано на рис. 7.1 (див. Лекцію 7). Віртуальна машина запускається за допомогою сканера для повернення маркерів файлу CASM. Загальноприйнятим є застосування сканера як кінцевого автомата. Кінцевий автомат складається з станів і переходів між станами залежно від символів, прочитаних із вхідного файлу. Кінцевий автомат приймає маркери файлу CASM. Кінцевий автомат, що використовується JCoCo, зображений на рис. 9.6.

Рис. 9.6 JCoCo сканер FSM



Коли парсер побудований, він спочатку створює сканер для зчитування маркерів програми **CASM**. Метод **getToken** **PyScanner** записаний як кінцевий автомат для отримання токенів програми **CASM**. Малюнок 9.7 містить методи **getToken** та **putBackToken** для сканера. Метод **putBackToken** здатний повернути один маркер, який використовується синтаксичним аналізатором, коли йому потрібно заглянути вперед один маркер, щоб визначити його наступну дію.

Сканер зчитує з потоку, який у випадку з JCoCo є **`PushbackInput-Stream`**, так що символ може бути непрочитаним. Сканер також відстежує свою позицію у файлі, щоб кожен маркер міг проходити по позиції, де він був знайдений у файлі вводу.

Стартовий стан дорівнює 0, як показано на рис. 9.6. У кінцевій машині можна взяти до уваги кілька речей. По-перше, ідентифікатори приймаються станом 1 і обмежуються буквами та цифрами, де першим символом є буква. Символи підкреслення та символи `@` розглядаються сканером, тому маркери, такі як `@x _1`, розпізнаються як ідентифікатори JCoCo, навіть якщо це нелегальний ідентифікатор у Python. Стан 2 приймає цілі числа. Стан 5 приймає числа з плаваючою комою, які повинні мати десяткову крапку. Наукові позначення з плаваючою комою не приймаються JCoCo.

Стани 6 і 7 відповідають за розпізнавання рядків. Ці стани продовжують читати, поки не буде знайдено одинарну або подвійну лапку, яка закінчує рядок. Однак рядки не можуть мати лапки або подвійні лапки, як вони визначені. Наприклад, рядок «**як справи?**» заборонений, оскільки в JCoCo немає реалізованого символу екранування, а друга лапка закінчує рядок. Повну реалізацію кінцевого автомата можна знайти в **PyScanner.java**, лише витяг коду наведений на рис. 9.8.

```
public PyToken getToken() {
    if (!this.needToken) {
        this.needToken = true;
        return this.lastToken;
    }

    try {
        next = this.in.read();
        if (next == -1) {
            type = TokenType.PYBADTOKEN;
            foundOne = true;
        }

        while (!foundOne) {
            this.colCount++;
            c = (char) next;
            switch (state) {
                case 0:
                    lex = "";
                    column = this.colCount;
                    line = this.lineCount;
                    if (isLetter(c)) state = 1;
                    else if (Character.isDigit(c)) state = 2;
                    else if (c == '-') state = 11;
                    ...
```

Рис.9.8 PyScanner getToken і putBackToken методи

```
                    break;
                ...
            }
            if (!foundOne) {
                lex += c;
                next = this.in.read();
                if (next == -1) {
                    type = TokenType.PYEOFTOKEN;
                    foundOne = true;
                }
            }
        }
        this.in.unread((char)next);
        this.colCount--;
    } catch(IOException e) {
        System.err.println(e.getMessage());
    }
    PyToken t = new PyToken(type, lex, line, column);
    this.lastToken = t;
    return t;
}

public void putBackToken() {
    needToken = false;
}
```

Вхідний потік містить метод повернення останнього символу, який використовується кодом сканера в рядку 38 циклу кінцевого автомата на рис. 9.8. Останній символ потрібно повернути назад, коли маркер повертається, оскільки останній символ не є частиною цього маркера. Розглянемо, наприклад, стан 1. Кінцевий автомат залишається в стані 1 до тих пір, поки символ все ще залишається буквою або цифрою. Якщо воно не є ні тим, ні іншим, для змінної **`foundOne`** встановлюється значення **`true`**, завершуючи цикл. Але цей останній символ може бути частиною наступного маркера, тому його повертають перед поверненням.

Перед тим, як **`getToken`** повертає маркер, маркер, який потрібно повернути, зберігається. Це використовується методом **`putBackToken`**. Якщо встановлено останній маркер, тоді **`needToken`** просто встановлюється як **`false`**. Коли викликається **`getToken`**, рядки 2–5 перевіряють, чи **`needToken`** хибний, і якщо так, повертаємо маркер, повернутий методом **`putBackToken`**. Зберігаючи маркер до його повернення, завжди запам'ятовується останній маркер, якщо його потрібно повернути знову.

Парсер

Маркери файлу **CASM** зчитуються синтаксичним аналізатором та аналізуються відповідно до граматичних правил у Додатку А. Кожен нетермінал **BNF** відповідає одній функції в парсері. Синтаксичний аналізатор повертає абстрактне дерево синтаксису, що представляє програму **CASM**. У цій реалізації абстрактне дерево синтаксису є **ArrayList** об'єктів **PyCode** та **PyClass**, що означає, що **ArrayList** оголошено як список **PyObjects**.

Рисунок 9.9 містить схему методу синтаксичного аналізу та частину коду, що викликається синтаксичним аналізом, який можна знайти в **`PyParser.java`**. Кожен метод парсера відповідає нетерміналу граматики. Реалізація методу кожного методу визначається правою частиною його правил. Вся реалізація парсера знаходиться в **`PyParser.java`**. Рисунки 9.9 та 9.10 містять два уривки цього коду. Вивчення правил для **`ClassFunctionList`**, **`FunDef`** та **`ConstPart`** дозволить пролити світло на реалізацію методів на рис. 9.9 та 9.10.

```
CoCoAssemblyProg ::= ClassFunctionListPart EOF
ClassFunctionListPart ::= ClassFunDef ClassFunctionList
ClassFunctionList ::= ClassFunDef ClassFunctionList | <null>
ClassFunDef ::= ClassDef | FunDef
FunDef ::= Function colon Identifier slash Integer
    ClassFunctionList ConstPart LocalsPart FreeVarsPart
    CellVarsPart GlobalsPart BodyPart
ClassDef ::= Class colon Identifier [ ( Identifier ) ]
    BEGIN ClassFunctionList END
ConstPart ::= <null> | Constants colon ValueList
```

Починаючи з нетерміналу **ClassFunctionList**, його правила говорять, що він або порожній (тобто **<null>**), або це **ClassFunDef**, за яким слідує **ClassFunctionList**. Звідки ми знаємо, якого правила дотримуватися? Відповідь можна знайти, заглянувши вперед на одну лексему. Якщо ми вивчаємо правило **FunDef**, воно повинно починатися з ключового слова **Function**, і це має бути наступним маркером, який слід прочитати у реалізації **ClassFunctionList**, якщо наступна частина програми не є визначенням класу. У цьому випадку нетермінал **ClassDef** вимагає ключове слово **Class**.

Щоб визначити, що робити, ми отримуємо наступний маркер у рядку 29 на рис. 9.9, відразу ж повертаємо його назад і перевіряємо, чи не було це ключовим словом **`Function`** або **`Class`**. Якщо це було будь-яке з них, то перше правило виконується за допомогою виклику **`ClassFunDef`**, а потім **`ClassFunctionList`**. Якщо функція або клас не є наступним маркером, ми повертаємо **`ArrayList`**, переданий методу, оскільки дотримуємося правила **`<null>`**.

Рис.9.9
PyParser.java
excerpt 1

```
1  public ArrayList<PyObject> parse() {
2      try {
3          return PyAssemblyProg();
4      } catch (PyException e) {
5          this.in.putBackToken();
6          PyToken tok = this.in.getToken();
7          // print error message
8          System.exit(0);
9      }
10     // unreachable
11     return null;
12 }
13 private ArrayList<PyObject> PyAssemblyProg() {
14     ArrayList<PyObject> code = ClassFunctionListPart();
15     PyToken tok = this.in.getToken();
16     if (tok.getType() != TokenType.PYEOFTOKEN) {
17         badToken(tok, "Excpected End Of File (EOF)");
18     }
19     return code;
20 }
21 private ArrayList<PyObject> ClassFunctionListPart() {
22     PyObject obj = ClassFunDef();
23     ArrayList<PyObject> codeList = new ArrayList<PyObject>();
24     codeList.add(obj);
25     codeList = ClassFunctionList(codeList);
26     return codeList;
27 }
```

```
28  private ArrayList<PyObject> ClassFunctionList(ArrayList<PyObject> codeList) {
29      PyToken tok = this.in.getToken();
30      this.in.putBackToken();
31
32      PyObject obj = null;
33      String lexeme = tok.getLex();
34      if (lexeme.equals("Function") || lexeme.equals("Class")) {
35          obj = ClassFunDef();
36          codeList.add(obj);
37          codeList = ClassFunctionList(codeList);
38      }
39      return codeList;
40  }
41  private PyObject ClassFunDef() {
42      PyToken tok = this.in.getToken();
43      this.in.putBackToken();
44      PyObject obj = null;
45      if (tok.getLex().equals("Function")) {
46          obj = FunDef();
47      } else if (tok.getLex().equals("Class")) {
48          obj = ClassDef();
49      } else {
50          // throw  exception
51      }
52      return obj;
53  }
```

Рис.9.10
PyParser.java
excerpt 2

```
private PyCode FunDef() {
    PyToken tok = this.in.getToken();
    if (!tok.getLex().equals("Function")) {
        badToken(tok, "Expected Function keyword.");
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals(":")) {
        badToken(tok, "Excpected a ':'.");
    }
    PyToken funName = this.in.getToken();
    if (funName.getType() != TokenType.PYIDENTIFIERTOKEN) {
        badToken(funName, "Expected an identifier");
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals("/")) {
        badToken(tok, "Expected a '/'");
    }
    PyToken numArgsToken = this.in.getToken();
    int numArgs = 0;
    if (numArgsToken.getType() != TokenType.PYINTEGERTOKEN) {
        badToken(numArgsToken, "Expected an integer token");
    }
    try {
        numArgs = Integer.parseInt(numArgsToken.getLex());
    } catch (NumberFormatException e) {
        System.err.println(e.getMessage());
        System.exit(0);
    }
```

```
29      ArrayList<PyObject> nestedClassFunctionList = new ArrayList<PyObject>();
30      nestedClassFunctionList = ClassFunctionList(nestedClassFunctionList);
31      ArrayList<PyObject> constants = ConstPart(nestedClassFunctionList);
32      ArrayList<String> locals = LocalsPart();
33      ArrayList<String> freevars = FreeVarsPart();
34      ArrayList<String> cellvars = CellVarsPart();
35      ArrayList<String> globals = GlobalsPart();
36      ArrayList<PyByteCode> instructions = BodyPart();
37      return new PyCode(funName.getLex(), nestedClassFunctionList, constants,
38              locals, freevars, cellvars, globals, instructions, numArgs);
39  }
40  private ArrayList<PyObject> ConstPart(ArrayList<PyObject> nestedCFList) {
41      ArrayList<PyObject> constants = new ArrayList<PyObject>();
42      PyToken tok = this.in.getToken();
43      if (!tok.getLex().equals("Constants")) {
44          this.in.putBackToken();
45          return constants;
46      }
47      tok = this.in.getToken();
48      if (!tok.getLex().equals(":")) {
49          badToken(tok, "Expected a ':'.");
50      }
51      constants = ValueList(constants, nestedCFList);
52      return constants;
53  }
```

Чому **ArrayList** з **PyObjects** передається методу **ClassFunctionList**? Цей **ArrayList** - це абстрактне дерево синтаксису “дотепер”, оскільки воно було прочитане до цього моменту в синтаксичному аналізаторі. Метод **ClassFunctionList** додає до цього **ArrayList**, якщо він знаходить іншу функцію або визначення класу.
Метод **FunDef** має дотримуватися лише одного правила. Він відповідає за побудову об’єкта **PyCode** для повернення до методу **ClassFunDef**. Коли викликається **FunDef**, ми вже перевірили, що першим маркером є ключове слово **Function**, щоб рядки 2–5 можна було пропустити.

Решта методу отримує маркери, перевіряє їх, чи є вони очікуваними маркерами, і викликає інші методи аналізатора, щоб прочитати решту визначення функції.

Метод **ConstPart** має дотримуватися двох правил, як метод **ClassFunctionList**. Знову ж таки, він повинен отримати маркер, щоб визначити, якому правилу слідувати. Якщо наступний маркер не є константами, тоді використовується порожнє правило, а метод **ConstPart** повертає порожній **ArrayList**. В іншому випадку він повертає **ArrayList** констант, що використовуються у функції. Кожен рядок константи використовується для побудови значення **PyObject** для цієї константи. **ArrayList nestedCFList** передається методу **ConstPart**, оскільки вкладений клас або функція самі по собі є константою, що зберігається у об'єкті **PyClass** або **PyCode** відповідно.

Коли константа типу **`g(g)`** з'являється у списку констант, вона говорить парсеру шукати код для неї у списку вкладених класів або функцій, переданих методу **`ConstPart`**.
Витяги коду на рис. 9.9 та 9.10 демонструють, що функції синтаксичного аналізатора є простою реалізацією правил у граматиці. Час від часу потрібен маркер пошуку, щоб визначити, якому правилу слідувати, але в іншому випадку синтаксичний аналізатор отримує маркери, коли це потрібно, і викликає інші нетермінальні методи, якщо це вказано правилом. Найскладніша частина написання синтаксичного аналізатора, ймовірно, визначає, що слід повернути. Це продиктовано інформацією, яка необхідна в дереві абстрактного синтаксису, що визначається передбачуваним використанням інформації у вихідному файлі.

Асемблер

Перш ніж CoCo зможе виконати код у функції, усі мітки повинні бути перетворені на цільові адреси в інструкціях. Мітки не мають сенсу для інтерпретатора байт-кодів. Мітки зручні для програмістів, але не для виконання коду. Етап складання шукає мітки та замінює будь-яку мітку переходу інструкції адресою, якій вона відповідає. Наприклад, розглянемо програму **CASM** на рис. 9.11. Мітка00 ідентифікує інструкцію зі зміщенням 11 основної функції. Мітка01 відображає зсув 18, а мітка02 - зсув 19. Інструкції в рядках 14, 17 і 23 повинні отримувати зсув, а не мітку, запланованих цілей. Це робота монтажника.

Асемблер досить простий для включення до коду синтаксичного аналізатора, коли аналізується частина тіла функції. У ньому є дві частини, які використовують **HashMap** для запам'ятовування, а потім оновлення цільових адрес у коді. Код асемблера міститься у двох із методів синтаксичного аналізу, методі **LabeledInstruction** та **BodyPart**. Тут наводяться граматичні правила, що оточують цей код.

```
<BodyPart> ::= BEGIN <InstructionList> END
<InstructionList> ::= <null> | <LabeledInstruction> <InstructionList>
<LabeledInstruction> ::= Identifier colon <LabeledInstruction> |
                         <Instruction> | <OpInstruction>
<Instruction> ::= STOP_CODE | NOP | POP_TOP | ROT_TWO | ROT_THREE | ...
```

Код **`LabeledInstruction`** додає кожну виявлену мітку на карту від міток до цілих зсувів. Рядки 32–39 на рис. 9.12 роблять це, коли виявляють, що інструкція містить мітку. Якщо код знаходить мітку, тоді рядок 34 додає мітку на карту, роблячи її вказівкою на зміщення, що називається індексом у коді.
Розташування цілей оновлюються в тілі функції за рядками 11–19 рис. 9.12. Якщо знайдено інструкцію, яка використовує мітку як свою ціль, вона видаляється і створюється нова інструкція з однаковим кодом операції з фактичною адресою цілі інструкції.

Байт-код

Об'єкт **PyByteCode** створюється для кожної інструкції, знайденої в програмі **CASM**. Визначення класу, частково визначене на рис. 9.13, показує перелік, який оголошується з усіма можливими кодами операцій. Ця конструкція переліку в Java є зручною та потужною. Перерахування насправді є визначенням класу, яке оголошує як ім'я кожного перерахованого значення, так і будь-які пов'язані з ним атрибути. У випадку переліку **PyOpCode** кожне ім'я інструкції асоціювало з ним кількість аргументів, які будуть надані з інструкцією. Кожна інструкція має нуль або один аргумент, які з’являються відразу за інструкцією у файлі **CASM.**

Посилаючись на рис. 9.11, більшість інструкцій у цьому прикладі мають один аргумент, за винятком команд **`GET _ITER, POP _TOP`** та **`POP _BLOCK`**, які мають нуль аргументів.
Перераховані значення зручні тим, що допомагають у написанні самодокументуючого коду. Перераховані значення в програмі будуються як об'єкти, по одному для кожного значення, перерахованого в декларації. На них може посилатися їх перелічене значення в коді. Наприклад, у **`PyFrame.java`** оператор **`switch`** вибирає між можливими значеннями інструкцій.

Рис.9.11 listiter.casm

```
Function: main/0
Constants: None, "Enter a list: "
Locals: x, lst, b
Globals: input, split, print
BEGIN
            LOAD_GLOBAL          0
            LOAD_CONST           1
            CALL_FUNCTION        1
            STORE_FAST           0
            LOAD_FAST            0
            LOAD_ATTR            1
            CALL_FUNCTION        0
            STORE_FAST           1
            SETUP_LOOP     label02
            LOAD_FAST            1
            GET_ITER
label00:    FOR_ITER       label01
            STORE_FAST           2
            LOAD_GLOBAL          2
            LOAD_FAST            2
            CALL_FUNCTION        1
            POP_TOP
            JUMP_ABSOLUTE  label00
label01:    POP_BLOCK
label02:    LOAD_CONST           0
            RETURN_VALUE
END
```

Рис.9.12
Асембльована
програма

```
private ArrayList<PyByteCode> BodyPart() {
    ArrayList<PyByteCode> instructions = new ArrayList<PyByteCode>();
    PyToken tok = this.in.getToken();
    this.target.clear();
    this.index = 0;
    if (!tok.getLex().equals("BEGIN")) {
        badToken(tok, "Expected a BEGIN keyword.");
    }
    instructions = InstructionList(instructions);
    //find the target of any labels in the byte code instructions
    for (int i = 0; i < instructions.size(); i++) {
        PyByteCode inst = instructions.get(i);
        String label = inst.getLabel();
        if (!label.equals("")) {
            String op = inst.getOpCodeName();
            instructions.remove(instructions.get(i));
            instructions.add(i, new PyByteCode(op, target.get(label)));
        }
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals("END")) {
        badToken(tok, "Expected a END keyword.");
    }
    return instructions;
}

private PyByteCode LabeledInstruction() {
    PyToken tok1 = this.in.getToken();
    String tok1Lex = tok1.getLex();
    PyToken tok2 = this.in.getToken();
    String tok2Lex = tok2.getLex();
    if (tok2Lex.equals(":")) {
        if (!this.target.containsKey(tok1Lex)) {
            this.target.put(tok1Lex, this.index);
        } else {
            badToken(tok1, "Duplicate label found.");
        }
        return LabeledInstruction();
    }
  // code omitted here.
}
```

Рис.9.13 Статична ініціалізація

```
class PyByteCode {
    enum PyOpCode {
        BINARY_ADD (0),
        LOAD_CONST (1),
        COMPARE_OP (1),
        CALL_FUNCTION (1),
        ...;
        private int args;
        PyOpCode(int args) {
            this.args = args;
        }
        public int args() {
            return this.args;
        }
    };

    private static HashMap<String, PyOpCode> OpCodeMap = createOpCodeMap();
    private static HashMap<String, Integer> ArgMap = createArgMap();

    private static HashMap<String, PyOpCode> createOpCodeMap() {
        HashMap<String, PyOpCode> map = new HashMap<String, PyOpCode>();
        for (PyOpCode opcode : PyOpCode.values()) {
            map.put(opcode.name(), opcode);
        }
        return map;
    }

    private static HashMap<String, Integer> createArgMap() {
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        for (PyOpCode opcode : PyOpCode.values()) {
            map.put(opcode.name(), opcode.args());
        }
        return map;
    }
    // code omitted here.
    public PyByteCode(String opcode, int operand)  {
        if (!OpCodeMap.containsKey(opcode)) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
                                  "Unknown opcode "+opcode);
        }

        this.opcode = OpCodeMap.get(opcode);
        this.operand = operand;
        this.label = "";
    }
    // code omitted here.
}
```

```
inst = this.code.getInstructions().get(this.PC);
switch (inst.getOpCode()) {
    case LOAD_FAST:
        u = this.locals.get(this.code.getLocals().get(operand));
        if (u == null) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
                    "NameError: name '" + this.code.getLocals().get(operand) ...
        }
        this.opStack.push(u);
        break;
    ...
```

Цей код отримує код операції з наступної інструкції. Оператор **`switch`** записується з кожним кодом операцій, переліченим у операторах **`case`**. Це робить код дуже чітким при його вивченні. Поведінка інструкцій пов'язана з назвою кожної інструкції.

Для побудови об'єктів **PyByteCode** з файлу **CASM** необхідно перевести рядок кодів операцій, як **"LOAD _FAST"**, у їх фактичні коди операцій, як **LOAD _FAST**. Для цього має бути спосіб знайти рядок і знайти відповідний код операції. Цей пошук здійснюється за час **O (1)** за допомогою **HashMap**. Хеш-карта створюється один раз, коли програма починається. Коли код виконується один раз і лише один раз при ініціалізації програми, це називається статичною ініціалізацією. Java і C ++ підтримують статичну ініціалізацію значень.

Усередині класу **PyByteCode** є дві статично виділені карти, які допомагають перекласти кожну інструкцію, прочитану парсером, в об'єкт **PyByteCode**. Код на рис. 9.13 з'являється в модулі **PyByteCode**. Дві змінні **OpCodeMap** та **ArgMap** статично ініціалізовані та доступні для всього коду, реалізованого в класі. **OpCodeMap** використовується, коли ім'я коду операції знайдено у файлі **CASM**. Він служить для підтвердження того, що це дійсна інструкція, та для надання перекладу на його перераховане значення. **ArgMap** забезпечує підрахунок кількості операндів, 0 або 1, дозволену для інструкції. Наприклад, пошук **"BINARY _ADD"** як **OpCodeMap ["BINARY _ADD"]** дасть перераховане значення **BINARY _ADD**.

Статична ініціалізація змінних може бути корисною, коли у вас є одноразовий код, який потрібно запустити під час ініціалізації програми. У цьому розділі показано, як це зробити за допомогою Java. Подібний код існує для C ++. Див. Модулі **`PyByteCode.cpp`** та **`PyByteCode.h`** у реалізації CoCo для прикладу, як це зробити за допомогою C ++ для отримання додаткової інформації. У цій версії Java дві функції, що створюють карти, виконуються при виклику статичної ініціалізації в рядках 17 та 18 коду на рис. 9.13.

Ієрархія класу та типу інтерфейсу JCoCo

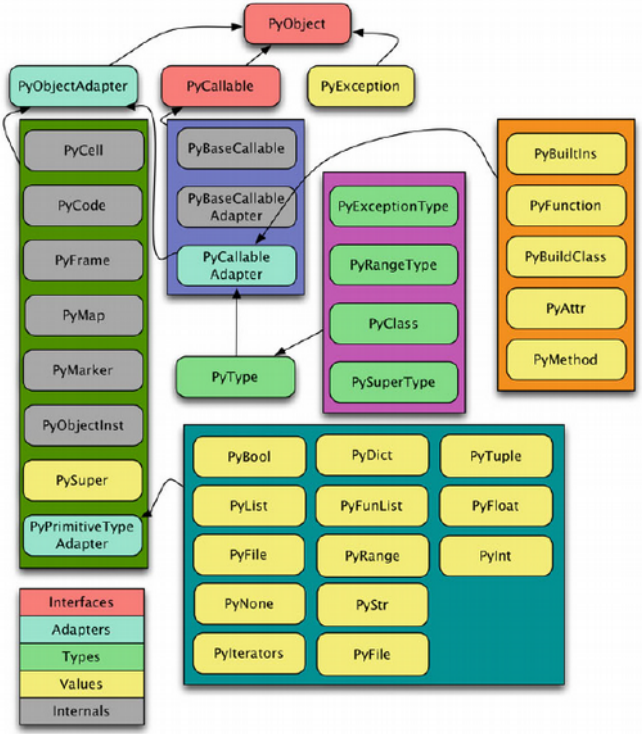
Реалізація JCoCo складається приблизно з п'ятдесяти шести класів та інтерфейсів. Приблизно п'ятдесят шість класів, тому що JCoCo продовжує рости та розвиватися. Наслідування класів використовується для повторного використання коду та поліморфізму протягом усієї реалізації. На малюнку 9.14 представлений погляд на ієрархію класів та інтерфейсів. **`PyBuiltIns`** представляють набір класів для всіх вбудованих функцій, що надаються JCoCo, які включають **`concat`**, **`print`**, **`fprint`**, **`tprint`**, **`iter`**, **`len`**, **`open`** та **`repr`**. Вбудовані функції **`fprint`** і **`tprint`** не є частиною Python.

Функція **`fprint`** - це функціональна версія друку, яка приймає одне значення та повертає екземпляр **`fprint`**. Вбудована функція tprint бере кортеж і друкує елементи кортежу з пробілами, що розділяють значення в кортежі. **`PyBuildClass`** схожий на вбудовану функцію, але доступний лише через інструкцію **`LOAD _BUILD _CLASS`**.

PyIterators представляє колекцію всіх класів ітераторів, які включають усі ітеративні значення, що підтримуються JCoCo, включаючи словники, списки, файли, списки переходів (які є функціональними списками, реалізованими як посилання **head** / **tail**). Ключ показує, що темно-сірі класи використовуються всередині реалізації JCoCo і недоступні для програм **CASM**. Ці класи є частиною внутрішньої реалізації JCoCo і недоступні програмісту. Деякі з них були описані раніше в цій главі. Клас **PyType** використовується усіма типами значень JCoCo, крім двох, при побудові об'єктів їх типу.

Тип винятків та діапазони об'єктів повинні були успадковуватися від класу **`PyType`**, тому поведінка виклику типу може бути замінена, оскільки ці два типи можна викликати для побудови або об'єкта винятку, або об'єкта діапазону відповідно.

Рис.9.14 Ієрархія типів JCoCo



Є два інтерфейси та три класи адаптерів. Коли це було можливо, адаптери писали, щоб дозволити спільний доступ коду між кількома класами. Розглянемо клас **PyPrimitiveTypeAdapter**. Цей клас визначає магічні методи **__repr__, __str__, __hash__, __iter__** та **__type__**. Ці методи викликаються методами **repr, str, hash, iter** і **type**.

Клас **PyException** є одним із цікавих прикладів необхідності багаторазового успадкування в Java. **PyException** успадковується від **RuntimeException**. Шляхом успадкування від **RuntimeException**, виключення JCoCo не потрібно оголошувати, що вони викидаються в кожному методі, який або створює або викликає щось, що може викликати виняток. Без успадкування від **RuntimeException** майже кожен метод у JCoCo повинен був би бути оголошений як можливий, що кидає **PyException**, що спричинило б безлад у коді.

Оскільки **PyException** успадковує від **RuntimeException**, він не може успадковувати від **PyObjectAdapter**. Натомість **PyException** реалізує інтерфейс **PyObject** і тому повинен повторно реалізувати методи, загальні для класу **PyObjectAdapter**. За умови багаторазового успадкування цього можна було б уникнути. Але це єдиний випадок, коли множинне успадкування було б корисним у цій колекції класів.

JCoCo підтримує ряд різних типів значень, включаючи цілі числа, плаваючі, словники, списки, рядки, булеві значення та деякі інші. Кожне з цих значень має пов’язаний із ним тип. Кожен об'єкт JCoCo має метод **`getType`**, який повертає його тип. Виявляється, навіть типи є об'єктами, і вони теж мають тип. Виклик **`getType`** для типу повертає тип із іменем **`type`**. Це має десь закінчитися, і це відбувається з об’єктом типу з іменем **`type`**. Тип типу - це тип. Це відбувається з перших двох рядків функції **`initTypes`** у вихідному файлі **`JCoCo.java`**. Об'єкт типу створюється в першому рядку і створюється сам із собою як власний ідентифікатор типу.

```
PyType typeType = new PyType("type", PyTypeId.PyTypeType);
PyTypes.put(PyTypeId.PyTypeType, typeType);
```

У наступному матеріалі лекцій ми глибше заглибимося в декілька класів та інтерфейсів JCoCo, щоб дослідити їх призначення та те, як вони вписуються в більшу реалізацію JCoCo.

Код

Функція **CASM** складається з більш ніж просто послідовності об'єктів **PyByteCode**. Існує назва функції, кількість аргументів, переданих функції, список констант, що використовуються функцією, локальні змінні, посилання на глобальну змінну та будь-які вкладені функції або класи, оголошені в цій функції **CASM**. Вся ця інформація та інше інкапсульовано в об’єкт **PyCode**.

З точки зору програмування Java, цей код не такий унікальний. Це контейнер для всієї інформації, яка поєднується з кожною функцією. Оголошення класу змінних екземпляра наведено на рис. 9.14.

Рис.9.14 Змінні екземпляра класу PyCode

```
1  package jcoco;
2
3  import java.util.ArrayList;
4  import jcoco.PyException.ExceptionType;
5  import jcoco.PyType.PyTypeId;
6
7  class PyCode  extends PyObjectAdapter {
8      private String name;
9      private ArrayList<PyObject> nestedClassFunctions;
10     private ArrayList<String> locals;
11     private ArrayList<String> freevars;
12     private ArrayList<String> cellvars;
13     private ArrayList<String> globals;
14     private ArrayList<PyObject> consts;
15     private ArrayList<PyByteCode> instructions;
16     private int argCount;
17     ... // methods and constructors omitted.
18 }
```

Об'єкти **PyCode** неможливо виконати. Неможливо запустити об’єкт **PyCode**. Для запуску коду потрібні дві речі: код і середовище, в якому він повинен запускатися. Середовище - це змінні, функції та інші значення, які вже визначені та ініціалізовані до виконання функції. Середовище і код надаються об'єктам **PyFunction** під час їх виконання, що більш докладно описано в розділі “Функції” (слайд 67).

Коли функція Python кодується як об'єкт **`PyCode`**, є два необхідні списки. Вони відображають можливий зміст середовища. Безкоштовні змінні - це змінні, на які посилаються, що існують у середовищі, а не в коді функції. Інший список, клітинні зразки, - це список змінних, які надходять із середовища або є частиною середовища внутрішньої функції, яка може бути модифікована і, отже, на неї слід непрямо посилатися. Це означає, що ми проходимо додатковий крок до посилань на **`Cellvars`**, щоб ми могли оновлювати їх значення, отримуючи доступ до них з іншого середовища. Див. розділ “Функції” (зі слайду 67) для прикладу.

Функції

Кожна функція у файлі **CASM** аналізується синтаксичним аналізатором, а об'єкт **PyCode** створюється в дереві абстрактного синтаксису для представлення коду, його імені та кількості аргументів, а також декларації констант, локальних значень, фріварів, комірок та глобалів, як описано в розділі "Код". Але об'єкти **PyCode** не можна викликати, як показано на рис. 9.13. Для виклику вам потрібен як код, так і середовище, в якому можна виконати код. Середовище заповнює прогалини, так би мовити. Фрівари не визначені в коді функції. Фрівари надходять з навколишнього середовища.

Код на рис. 9.15 надає конструктор та метод виклику для класу **`PyFunction`**. Конструктор будує так зване закриття від оточення та коду. Закриття ініціалізується у рядку 8–10, де ми перебираємо вільні змінні в коді, що відображають змінні клітинки у закритті від їхніх імен вільних змінних до значень змінних клітинок. До всіх змінних, до яких здійснюється доступ із закриття, здійснюється непрямий доступ через змінні комірки.

Рис.9.15 Конструктор PyFunction та метод _ _call _ _

```
public PyFunction(PyCode theCode, HashMap<String, PyObject> theGlobals,
        PyObject env) {
    PyTuple tuple = (PyTuple)env;
    this.cellvars = new HashMap<String, PyCell>();
    this.code = theCode;
    this.globals = theGlobals;

    for (int i = 0; i < theCode.getFreeVars().size(); i++) {
        this.cellvars.put(theCode.getFreeVars().get(i),
                          (PyCell)tuple.getVal(i));
    }

    PyFunction self = this;
    this.dict.put("__call__", new PyCallableAdapter() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args)  {
            return self.__call__(args);
        }
    });
}
@Override
public PyObject __call__(ArrayList<PyObject> args)  {
    if (args.size() != this.code.getArgCount()) {
        throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                        "Type Error: expected "+this.code.getArgCount() +
                        " arguments, got "+args.size());
    }
    PyFrame frame = new PyFrame(this.code, args, this.globals,
                           this.code.getConsts(), this.cellvars);
    PyObject result = frame.execute();
    return result;
}
```

Коли функцію викликають, викликається метод **`__call__`**. Коли це відбувається, створюється новий об'єкт **`PyFrame`**. **`PyFrame`** містить лічильник програми та простір для зберігання локальних змінних. Об'єкт **`PyFrame`** виконується за допомогою виклику методу **`execute`**.

Класи

Визначені користувачем класи в JCoCo - це колекції об’єктів **PyFunction** та вкладених класів. Клас містить ім'я класу (тобто його тип), його супер клас та список об'єктів **PyFunction** або **PyClass**, як показано на рис. 9.16. Це відображає реалізацію в Python. Наприклад, хоча це зазвичай не пишеться таким чином, якщо скласти два цілих числа, можна записати це.

```
z = int.__add__(x, y)
```

Цей код шукає магічний метод додавання в класі **int** і викликає його, передаючи два аргументи функції. Хоча додавання є методом, що викликається на цілочисельному об'єкті, його також можна викликати для класу, надаючи обидва цілих числа.

```
> python3.2
Python 3.2.5 (v3.2.5:cef745775b65, May 13 2013, 13:37:00)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int.__add__(4,6)
10
>>>
```

Коли **PyClass** побудований, йому передається список внутрішніх класів та один об'єкт **PyCode** для кожного з його методів. Об'єкти **PyCode** використовуються для побудови об'єктів **PyFunction** у рядках 22–23. Об'єкти **PyFunction** розміщуються у словнику **attr**. Змінна **attr** класу містить атрибути, які мають бути передані будь-якому екземпляру цього класу. Клас містить функції, які стануть методами будь-якого екземпляра класу.

Наприклад, функція **`__add__`** у класі **`int`** стане методом у екземплярі класу **`int`**.

Методи

Екземпляри класу створюються шляхом виклику їх класу. Наприклад, написання `Dog ("Mesa")` створює екземпляр класу `Dog`. Код, який виконується при виклику класу, показаний на рис. 4.39 у рядках 48–54, що викликає метод `initInstance` у рядках 36–47. У цьому коді всі `PyFunctions`, знайдені у змінній `attrs`, передаються екземпляру класу (тобто екземпляру об'єкта) як об'єкти `PyMethod`. Потім у рядку 52 викликається конструктор об’єкта для виконання будь-якої ініціалізації об’єкта. Продовжуючи наш приклад з попереднього розділу, це означає, що, як і в Python, наступний код можна розібрати та виконати в JCoCo.

```
public PyClass(String name, ArrayList<PyObject> nestedclassesandfuns,
            String baseClass, HashMap<String, PyObject> globals)  {
    super(name, PyTypeId.PyClassType);
    this.baseClass = baseClass;
    this.name = name;
    this.classesandfuns = nestedclassesandfuns;
    this.globals = (HashMap<String, PyObject>)globals;
    this.attrs.put("__name__", new PyStr(name));
    for (int i = 0; i < classesandfuns.size(); i++) {
        if (classesandfuns.get(i).getType().typeId() == PyTypeId.PyCodeType) {
            PyCode code = (PyCode) classesandfuns.get(i);
            ArrayList<PyObject> env = new ArrayList<PyObject>();
            for (int j = 0; j<code.getFreeVars().size(); j++) {
                String freeVar = code.getFreeVars().get(j);
                if (freeVar.equals("__class__")) {
                    env.add(new PyCell(this));
                } else {
                    throw new PyException(ExceptionType.PYMATCHEXCEPTION,
                     "Error: Found unexpected freevar in class declaration.");
                }
            }
            PyFunction fun = new PyFunction((PyCode) classesandfuns.get(i),
                globals, new PyTuple(env));
            this.attrs.put(fun.callName(), fun);
        } else if (classesandfuns.get(i).getType().typeId() ==
                PyTypeId.PyTypeType) {
            PyClass cls = (PyClass) classesandfuns.get(i);
            this.attrs.put(cls.getName(), cls);
        } else {
            throw new PyException(ExceptionType.PYMATCHEXCEPTION,
                        "TypeError: expected a Function or Class, got "+
                        classesandfuns.get(i).getType().str());
        }
    }
}
```

Рис.9.16 Конструктор PyClass та метод _ _call _ _

```
36  public void initInstance(PyObjectAdapter obj)  {
37      if (!baseClass.equals("")) {
38          ((PyClass)globals.get(baseClass)).initInstance(obj);
39      }
40      for (String name : this.attrs.keySet()) {
41          if (this.attrs.get(name).getType().typeId() ==
42                  PyTypeId.PyFunctionType) {
43              obj.dict.put(name, new PyMethod(name, obj,
44                  (PyCallable)this.attrs.get(name)));
45          }
46      }
47  }
48  @Override
49  public PyObject __call__(ArrayList<PyObject> args)  {
50      PyObjectAdapter obj = new PyObjectInst(this);
51      initInstance(obj);
52      ((PyMethod) obj.dict.get("__init__")).__call__(args);
53      return obj;
54  }
```

Винятки та відстеження JCoCo

Виконання методу для **PyFrame** виконується одним із двох способів. Або виконується інструкція **RETURN _VALUE**, або виникає виняток, який не обробляється в рамках цієї функції. Якщо виникає виняток, виконання переходить до рядка 18 коду на рис. 9.17. Усі навмисно викинуті винятки, викинуті JCoCo, є об'єктами **PyException**, тому блок **catch** у рядку 18 буде його ловити.

```
public PyObject execute() {
  this.PC = 0;
  boolean handled = false;
  JCoCo.pushFrame(this);
  while (true) {
    try {
      inst = this.code.getInstructions().get(this.PC);
      this.PC++;
      opcode = inst.getOpCode();
      operand = inst.getOperand();
      switch (opcode) {
        // instructions omitted
        case SETUP_EXCEPT:
          this.blockStack.push(-1 * operand);
          opStack.push(new PyMarker());
          break;
      }
    } catch (PyException ex) {
      int exitAddress;
      boolean found = false;
      while (!found && !this.blockStack.isEmpty()) {
        exitAddress = this.blockStack.pop();
        if (exitAddress < 0) {
          found = true;
          if (!opStack.isEmpty()) {
            PyObject obj = opStack.pop();
            while (!obj.str().equals("Marker") && !opStack.isEmpty()) {
              obj = opStack.pop();
            }
          }
```

```
31              this.opStack.push(ex.getTraceBack()); //The traceback at TOS2
32              this.opStack.push(ex); //The exception at TOS1
33              this.opStack.push(ex); //the exception at TOS
34              this.PC = -1 * exitAddress;
35              this.blockStack.push(0);
36            }
37          }
38          if (!found) {
39            ex.tracebackAppend(this);
40            throw ex;
41          }
42        } catch (Exception e) {
43          PyException ex =
44             new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
45             e.getMessage()+" while executing instruction "+inst.getOpCodeName());
46          ex.tracebackAppend(this);
47          throw ex;
48        }
49      }
50  }
```

Рис.9.17 Сінопсис обробки винятків у JCoCo

Винятки JCoCo використовують механізм обробки винятків Java для переходу до коду, що починається з рядка 18, щоразу, коли **PyException** передається в програму JCoCo, що є будь-яким винятком, навмисно викликаним програмою **CASM**. Після введення блоку **catch** у рядку 19 код шукає обробник винятків, який, можливо, був встановлений для обробки винятків у цьому об'єкті **PyFrame**. Може існувати обробник винятків, а може і ні. JCoCo включає стек блоків, що використовується для ітерації та обробки винятків. Стек блоків записує точки виходу з циклів та адреси обробників винятків. Для циклів точка виходу натискається на стек блоків на випадок, якщо виконується інструкція **BREAK _LOOP** для виходу з циклу.

Коли обробник винятків ставиться на місце, місце обробника позначається негативним значенням на стеці блоків, щоб диференціювати його від точок виходу з циклу. Рядки 13–16 показують, як обробник винятків встановлюється всередині кадру. Інструкція **SETUP _EXCEPT** штовхає адресу обробника винятків у стек блоків.

Коли виникає виняток, блок з блоку вискакує, поки не буде знайдена адреса блоку обробки винятків (тобто вискакує негативне значення). Адреса обробника винятків є запереченням цього негативного значення.

Коли виникає виняток, це може статися де завгодно в коді. Зокрема, в стеку операндів можуть залишитися операнди, оскільки виняток стався в середині якоїсь іншої роботи. Наприклад, під час підготовки до виклику функції може статися виняток, і аргументи можуть бути залишені в стеку операндів. Клас **`PyMarker`** служить для очищення стеку операндів після вилучення винятку. Коли обробник винятків встановлюється з інструкцією **`SETUP_EXCEPT`**, об'єкт **`PyMarker`** висувається в стек операндів. Якщо об'єкт **`PyMarker`** з'являється під час звичайного виконання інструкції, його просто викидають.

Якщо виникає виняток, код обробки винятків у рядку 25–29 вискакує аргументи зі стеку операндів, поки він не спорожниться або поки не буде знайдений об’єкт **`PyMarker`**, очищаючи таким чином стек операндів.

Якщо обробник винятків було знайдено в коді функції JCoCo, виняток надсилається у стек, щоб підготуватися до переходу до коду обробника винятків. Рядки 31–33 здаються трохи дивними, поки ви не пам’ятаєте, що JCoCo підтримує сумісність з Python 3.2, а розібраний код Python очікує, що на стек буде висунуто три операнди, коли обробник винятків почне виконуватися. Рядок 34 змушує виконання переходити до першої інструкції обробника винятків. Віртуальна машина Python також припускає, що є ще один блок обробки винятків, що натискається на `blockStack`. Це не потрібно JCoCo, але для підтримки сумісності рядок 35 виштовхує запис у стек блоків.

Якщо обробника винятків не знайдено, рядок 38 додає поточний кадр до зворотного відстеження винятку. **`Traceback`** - це список усіх об’єктів **`PyFrame`**, які з’являються, доки не буде знайдений обробник винятків. Якщо обробника винятків не знайдено, елемент керування повертається до основної функції, де друкується відстеження.

Винятки Java, що не належать до JCoCo, також можуть бути передані кодом JCoCo. Це може статися за одним із двох сценаріїв. JCoCo може мати помилку, і якщо це так, може бути створено виняток через деякі непередбачені обставини. Інша причина, через яку може бути створено стандартний виняток Java, полягає в тому, що програміст робить спробу незаконної операції, наприклад, арифметичної операції, що спричиняє переповнення цілих чисел, наприклад. Якщо трапляється будь-який із цих сценаріїв, тоді рядки 42–49 оброблятимуть ці винятки.

Коли виникає виняток Java, який не є JCoCo, створюється **`PyException`**, поточний кадр додається до його зворотного відстеження і **`PyException`** викидається. Якщо цей **`PyException`** ніде не виявлено в програмі **`CASM`**, тоді елемент керування повернеться до основної функції в **`JCoCo.java`**, а трасування зворотного винятку буде надруковано як джерело помилки.

Магічні методи

Рисунок 9.18 містить чотири методи, які визначені для кожного типу значення в JCoCo. Наприклад, будь-який об'єкт може бути перетворений у рядок, і всі об'єкти мають тип в ієрархії, який можна отримати. Зверніть увагу, що всі ці методи мають однаковий підпис. Кожна функція або метод у JCoCo (і Python) приймає список об'єктів як аргументи і повертає об'єкт. Кожен об'єкт JCoCo реалізує методи з цим підписом і лише з цим підписом. Методи **`__str__`** та **`__type__`** розробники Python називають магічними методами, оскільки їх автоматично викликають певні оператори в Python.

Наприклад, перетворення PyObject у рядок викликає **__str__** магічний метод, щоб отримати рядкове представлення об'єкта. Виклик типу об'єкта в програмі Python призводить до виклику методу **__type__** для отримання типу об'єкта.

Виклик **repr** для об'єкта в програмі Python викликає метод **__repr__**. Функція **repr** повертає рядкове представлення об'єкта, який при обчисленні дасть копію того самого об'єкта. Хеш-функцію також можна викликати на будь-якому об'єкті, але лише хеш-об'єкти реалізують магічний метод **__hash__** з чимось іншим, окрім створення виключення.

```
public PyObjectAdapter() {
    name = "PyObject()";
    type = PyType.PyTypeId.PyClassType;
    PyObjectAdapter self = this;
    this.dict.put("__str__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            return new PyStr(self.str());
        }
    });
    this.dict.put("__hash__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
            "TypeError: unhashable type: '" + self.getType().str() + "'");
        }
    });
    this.dict.put("__repr__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            return self.callMethod("__str__", args);
        }
    });
```

```
32      this.dict.put("__iter__", new PyBaseCallable() {
33          @Override
34          public PyObject __call__(ArrayList<PyObject> args) {
35              if (args.size() != 0) {
36                  throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
37                  "TypeError: expected 0 argument, got " + args.size());
38              }
39              throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
40              "TypeError: '" + self.getType().str() + "' object is not iterable");
41          }
42      });
43      this.dict.put("__type__", new PyBaseCallable() {
44          @Override
45          public PyObject __call__(ArrayList<PyObject> args) {
46              if (args.size() != 0) {
47                  throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
48                  "TypeError: expected 0 argument, got " + args.size());
49              }
50              return (PyObject) self.getType();
51          }
52      });
53  }
```

Рис.9.18 Конструктор PyObjectAdapter

Методи - це операції, які можна виконувати над об'єктами в ієрархії типів JCoCo. Магічні методи - це методи, які викликаються в результаті виклику вбудованої функції або перевантаження оператора в Python. Магічні методи **`__str__`**, **`__type__`**, **`__repr__`** та **`__hash__`** додаються до словника методів для всіх об'єктів конструктором **`PyObjectAdapter`**. На рисунку 9.18 показано методи, які підтримуються класом **`PyObjectAdapter`** для всіх підкласів. Але підкласи **`PyObjectAdapter`** можуть додати на карту підтримуваних операцій. Наприклад, конструктор об’єкта **`PyInt`** викликає метод **`funs`**, щоб додати цілу низку додаткових підтримуваних методів до цілочисельних об’єктів, як показано на рис. 9.19.

Ім'я методу, що називається просто іменем у коді на рис. 8.11 (лекція 8), шукається у словнику. Якщо його не знайдено, створюється виняток. В іншому випадку mbr змушений вказувати на код методу (тобто об’єкт **PyCallable**). Рядок 6 викликає метод **__call _ _** для цієї функції-члена або методу для поточного об'єкта, повертаючи все, що повертається із виклику, абоненту. Використання словника відображає імена (тобто рядки), надані JCoCo, у методи об’єктів, які реалізовані як об’єкти **PyCallable**, в межах JCoCo. Якщо об'єкт не має визначеного методу, код **callMethod** витончено обробляє це, викидаючи виняток, що призведе до друку трасування зворотної інструкції **CALL _FUNCTION**.

У JCoCo магічні методи викликаються як результат багатьох вказівок. Наприклад, магічний метод `__add__` викликається на об'єкті в результаті виконання інструкції `BINARY_ADD`. Інструкція `COMPARE_OP` викликає кілька різних магічних методів залежно від операнда порівняння інструкції.
Точно, який магічний метод викликається для даної інструкції, детально описано в документації, наведеній у Додатку А.

```
public static HashMap<String, PyCallable> funs() {
    HashMap<String, PyCallable> funs = new HashMap<String, PyCallable>();
    funs.put("__hash__", new PyCallableAdapter() {...});
    funs.put("__add__", new PyCallableAdapter() {...});
    funs.put("__sub__", new PyCallableAdapter() {...});
    funs.put("__mul__", new PyCallableAdapter() {...});
    funs.put("__pow__", new PyCallableAdapter() {...});
    funs.put("__truediv__", new PyCallableAdapter() {...});
    funs.put("__floordiv__", new PyCallableAdapter() {...});
    funs.put("__mod__", new PyCallableAdapter() {...});
    funs.put("__eq__", new PyCallableAdapter() {...});
    funs.put("__ne__", new PyCallableAdapter() {...});
    funs.put("__lt__", new PyCallableAdapter() {...});
    funs.put("__le__", new PyCallableAdapter() {...});
    funs.put("__gt__", new PyCallableAdapter() {...});
    funs.put("__ge__", new PyCallableAdapter() {...});
    funs.put("__float__", new PyCallableAdapter() {...});
    funs.put("__int__", new PyCallableAdapter() {...});
    funs.put("__bool__", new PyCallableAdapter() {...});
    funs.put("__str__", new PyCallableAdapter() {...});
    return funs;
}
```

Рис.9.19 Додаткові магічні методи PyInt

Підсумок ООП

У останніх трьох лекціях висвітлено об’єктно-орієнтоване імперативне програмування на Java із C ++, охоплене меншою мірою. Розширені методи, включаючи успадкування, поліморфізм, інтерфейси, узагальнення, автобоксинг та розпакування, внутрішні класи та кілька інших важливих тем були розглянуті на прикладах реалізації віртуальної машини JCoCo.

Java та C ++ є статично набраними мовами порівняно з Python, яка є динамічно набраною мовою. Статичне введення вимагає більше роботи програміста під час написання коду, але також забезпечує певний рівень впевненості у правильності набору коду. Помилки типу програми Python не виявляються до часу виконання. Компілятори C ++ та Java виявляють більшість помилок типу під час компіляції.

C ++ та Java мають багато синтаксису, але вони багато в чому різні мови. C ++ особливо підходить для реалізацій на низькому рівні та в режимі реального часу, де продуктивність є критичною і вам потрібен доступ до базового обладнання. C ++ дає вам повний контроль над тим, коли звільняються динамічно розподілені дані. Але з цією відповідальністю виникає давня проблема витоків пам'яті. Програми C ++ схильні до витоків пам'яті, і реалізація C ++ CoCo їх повна, оскільки важко точно визначити, коли можна звільнити простір у віртуальній машині без реалізації певної форми збору сміття. C ++ має багато чудових функцій програмування, таких як шаблони, велика стандартна бібліотека та підтримка компілятора для багатьох апаратних платформ.

Програми Java отримують вигоду від збору сміття, який вбудований в JVM. Java також забезпечує уніфіковану ієрархію типів для класів з `Object` у корені дерева.

C ++ не має вбудованої ієрархії класів. Java вбудувала в нього кілька приємних функцій програмування, таких як автобокс і розпакування, підтримка потоків, включаючи підтримку синхронізації всіх об'єктів Java, підтримку внутрішнього класу та підтримку відокремлення інтерфейсів від реалізації.

І C ++, і Java служать хорошими прикладами статично набраних, об'єктно-орієнтованих, імперативних мов програмування. Кожен з них має свої переваги та недоліки, але для віртуальних машин CoCo та JCoCo Java є найбільш підходящою мовою, що забезпечує збір сміття та уніфікований підхід до обробки винятків у віртуальній машині. У наступному розділі ми представляємо ще одну парадигму програмування під час вивчення іншої статично набраної мови.

На наступній лекції буде почато розгляд імплементації функціонального програмування.