YARVの ソースを 読んでみた (VM 編)

http://d.hatena.ne.jp/hzkr/19000101
の
まとめ

2007/04/02 (Mon) / d:id:hzkr

YARV とは

- プログラミング言語 Ruby の処理系
  - のひとつ
  - ささだこういち さんによる実装
  - Ruby 1.9 の処理系として本家にマージされました
    - *(Yet Another ではなくなった？)*
- 特徴
  - Rubyコードを仮想マシン語にコンパイルして実行
  - 速い！
- URI
  - http://www.atdot.net/yarv/
  - http://svn.ruby-lang.org/repos/ruby/trunk/

参考資料

- YARV Maniacs
  - http://jp.rubyist.net/magazine/?0006-YarvManiacs
- Ruby ソースコード完全解説
  - http://i.loveruby.net/ja/rhg/
- Ruby リファレンスマニュアル
  - http://www.ruby-lang.org/ja/man/

流れ（mainからyarvまで）

- main @ main.c
  - ruby_init @ eval.c
    - 組み込みモジュールの初期化など
  - ruby_options @ eval.c
    - この辺りで構文解析。本家Rubyと共通。
      (Rubyのコード文字列を、NODE型の木構造に変換)
  - ruby_run @ eval.c
    - ruby_exec @ eval.c
      - ruby_exec_internal @ eval.c
        - yarvcore_eval_parsed @ yarvcore.c

ここから
YARV 評価器
スタート

流れ（yarv評価器内）

- yarvcore_eval_parsed @ yarvcore.c
  - th_compile_from_node @ yarvcore.c
    - 構文木を、YARVマシン語列に変換（コンパイル）
    - yarv_iseq_new_with_opt @ iseq.c
      - iseq_compile @ compile.c
        - iseq_compile_each @ compile.c
          構文木→マシン語列の変換関数
        - iseq_setup @ compile.c
          最適化などなど
  - yarvcore_eval_iseq @ yarvcore.c
    - マシン語列を、実行

ここを
読むよ

このスライドの、この後の流れ

- VMのデータ構造
  - VM
  - スレッド
  - スタック
  - フレーム
- 実行開始！
- メインループ！
- 命令定義ファイル

VMのデータ構造：VM

- struct rb_vm_struct @ yarvcore.h
  - rb_thread_lock_t　global_interpreter_lock;
  - rb_thread_struct* main_thread;
  - rb_thread_struct* running_thread;
  - st_table*　living_threads;
  - …略…
- VMは「スレッドの集まり」
- ある時点で稼働中のスレッドは常に1個
  - == running_thread
  - == global_interpreter_lock をロックしてるスレッド
    - http://www.atdot.net/~ko1/w3ml/w3ml.cgi/yarv-dev/msg/631
    - http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-dev/30202

VMのデータ構造：スレッド

- rb_thread_struct @ yarvcore.h
  - VALUE* stack;
  - rb_control_frame_t* cfp;
  - native_thread_data_t native_thread_data;
  - …略…
- スレッドは
  - 計算用「スタック」
  - 現在の「制御フレーム」
  - あと、YARVスレッドはネイティブスレッドで実装されてるのでそのデータ

VMのデータ構造：スタック

- VALUE* stack
  - = ALLOC_N(VALUE, RUBY_VM_STACK_SIZE);
- ただの配列
  - YARVはスタックマシンなので、
    ここに値をpushしたりpopしたりして計算
  - ちなみに RUBY_VM_STACK_SIZE は
    128*1024 でした (2007/04/01現在)

VMのデータ構造：制御フレーム

- struct rb_control_frame_t @ yarvcore.h
  - VALUE* pc: 命令ポインタ
  - VALUE* sp: スタックポインタ
  - rb_iseq_t* iseq: 現在の関数/ブロックの命令列
  - VALUE* lfp: ローカル変数テーブル
  - などなどなどなど…
- VMの今の状態を表す
- 関数/ブロック呼び出しごとにスタック的に制御フレームを積んでいく感じ

実行開始！

実行開始！

色々あるけど

th_eval_body

から実行開始

- メインスレッドの場合
  - yarvcore_eval_iseq @ yarvcore.c
    - *rb_thread_eval @ vm.c*
      - th_eval_body @ vm.c
- Thread.new で作る新規スレッド
  - thread_s_new @ thread.c
    - *thread_create_core @ thread.c*
      - native_thread_create @ thread_(pthread|win32).c
        - thread_start_func_1 @ thread_(pthread|win32).c
        - thread_start_func_2 @ thread.c
        - th_invoke_proc
        - invoke_block
        - th_eval_body @ vm.c

th_eval_body (要約)

```
VALUE
th_eval_body(rb_thread_t* th)
{
  if( ... ) {
    vm_loop_start:
      th_eval(...);
      if( th->state != 0 )
        goto exception_handler;
  }
  else {
      ...
    exception_handler:
      ...
  }
}
```

th_eval がメインループ

例外発生時か
Ruby実行終了時に
return

例外catchするハンドラを
ここで地道に検索

ハンドラを見つけたら
制御フレーム巻き戻して
goto vm_loop_start

return from th_eval

- 例外発生時
  - YARVの"throw"命令
- Rubyコード実行終了時
  - YARVの"finish"命令

  - メソッド終了時 (YARVの"leave"命令)
    には、いちいち th_eval を抜けたりしない

メインループ！

th_eval

- 命令1個読んでswitch&実行,の無限ループ
  - …するコードを #include

```
VALUE
th_eval( rb_thread_t* th, VALUE initial )
{
    INSN_DISPATCH();
    #include "vm.inc"
    END_INSN_DISPATCH();
}
```

vm.inc

- マクロ展開するとだいたいこんな感じ
  - （スレッデッドコード最適化OFFの場合）

```
while(1)
  switch(*cfp->pc)
  {
  case YARVINSN_leave: ...
  case YARVINSN_finish: ...
  case YARVINSN_branchif: ...
  ...                          // などなど...
  }
```

- 「命令定義ファイル(insns.def)」から
  Rubyスクリプトで生成される！

命令定義ファイル

insns.def

- 各YARV命令の実装を専用記法で書いた物

- 命令の
  - 名前
  - 引数リスト
  - スタックからPOPする変数名のリスト
  - スタックにPUSHする変数名
  - 実際に実行する処理（ここはC言語で書く）

insns.def

- 例 : getlocal
- 指定された番号のローカル変数の値を
  スタックに積む命令
  - *ローカル変数にアクセスするときに使われてる命令*

```
DEFINE_INSN
getlocal                          ← 命令の名前
(lindex_t idx)                    ← 命令の引数(ローカル変数番号)
()                                ← スタックからPOPする値(なし)
(VALUE val)                       ← スタックにPUSHする値
{
  val = *(GET_LFP() - idx);       ← 実装(制御フレームからローカル
                                    変数領域を取得してそこの値をget)
}
```

insns.def

- 例 : tostring
- スタックトップにある値をString化して
  スタックに置き直す命令
  - *“#{...}” とかで使われてる命令*

```
DEFINE_INSN
tostring                              ← 命令の名前
()                                    ← 命令の引数(なし)
(VALUE val)                           ← スタックからPOPする値
(VALUE val)                           ← スタックにPUSHする値
{
  val = rb_obj_as_string(val);        ← 実装(オブジェクトの表現などは
                                          従来のRubyと同じなので、
                                          従来の実装と同じ関数でOK)
}
```

insns.def

- 例：jump
- 指定された距離だけpc（次に実行する命令のアドレス）を動かす命令
  - *whileやifなどなどで使われてる命令*

```
DEFINE_INSN
jump                          ← 命令の名前
(OFFSET dst)                  ← 命令の引数(ジャンプ距離)
()                            ← スタックからPOPする値(なし)
()                            ← スタックにPUSHする値(なし)
{
  RUBY_VM_CHECK_INTS();       ← 各種ジャンプ命令のタイミングで
                                割り込みチェック＆スレッド切替してるみたい
  JUMP(dst);                  ← 実装 (cfp->pc += dst)
}
```

insns.def

- 例 : putobject
- 指定されたオブジェクトをスタックに積む
  - 1 とか true とか即値を書いたときに使われる命令
  - C実装の部分が空でちょっとかっこいい

```
DEFINE_INSN
putobject                ← 命令の名前
(VALUE val)              ← 命令の引数(オブジェクト)
()                       ← スタックからPOPする値(なし)
(VALUE val)              ← スタックにPUSHする値
{
}
```

まとめ

- YARVの、VM実装
  - …の部分のコードを読んだ結果をまとめました
  - 超ダイジェスト版なので、物足りない方はぜひぜひ
    http://svn.ruby-lang.org/repos/ruby/trunk/
    を読みましょう!!