Как и зачем LuaJIT нарушает DRY?

Антон Солдатов, IPONWEB

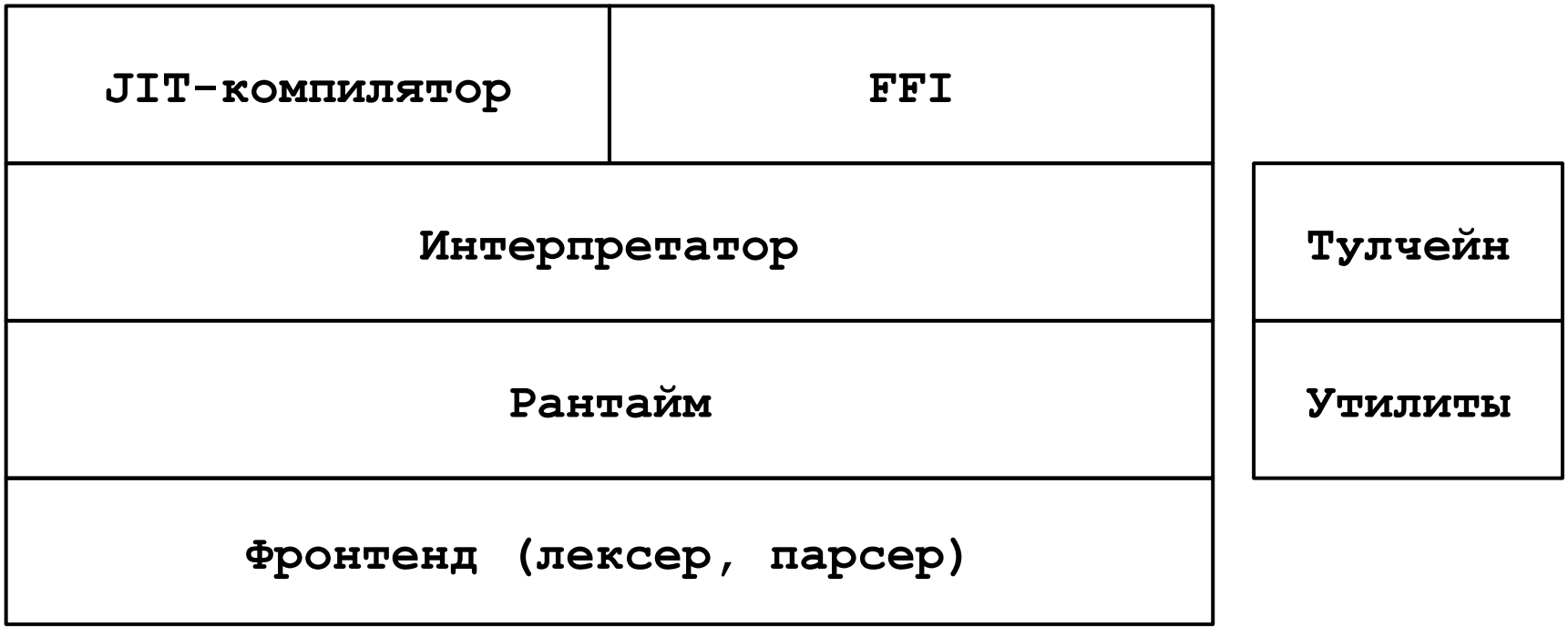
Lua in Moscow @ DevConf, 17 июня 2017 г.

Don't Repeat Yourself

- **Навязанное дублирование.** Разработчики чувствуют, что у них нет выбора – им кажется, что дублирования требует среда окружения.
- **Неосознанное дублирование.** Разработчики не осознают, что они тиражируют информацию.
- **Нетерпеливое дублирование.** Разработчики ленятся и осуществляют дублирование, потому что им кажется, что так проще.
- **Коллективное дублирование.** Фрагмент информации тиражируется несколькими членами одной команды разработчиков (или нескольких команд).

Эндрю Хант, Дэвид Томас. Программист-прагматик.

Из чего состоит LuaJIT?



Из чего состоит LuaJIT: Интерпретатор

- Реализован на DynASM, но отдельно для каждой платформы
- Главный цикл виртуальной машины
- Вход в виртуальную машину и возврат из неё
- Вход в JIT-код и возврат из него
- Запуск JIT-компилятора
- Реализация семантик байт-кодов
- Частичная реализация некоторых функций стандартной библиотеки Lua

Из чего состоит LuaJIT: Рантайм

- Реализация системы типов Lua
- Реализация функций стандартной библиотеки Lua
- Реализация C API
- Сборщик мусора

Из чего состоит LuaJIT: JIT-компилятор

- Рекордер (преобразует байт-код в IR)
- Оптимизатор IR (выполняет машинно-независимые оптимизации)
- Ассемблер машинного кода (свой для каждой целевой платформы)

Где LuaJIT нарушает DRY?

- Fast functions
- Обход таблиц
- Индексирование таблиц
- Листер байт-кода

Fast Functions

- Средство реализации функций стандартной библиотеки Lua
- Частично реализованы внутри интерпретатора на ассемблере ради быстродействия
- Частично – внутри рантайма на Си (для соблюдения полноты семантики)
- Семантика многих из них дополнительно реализована в рекордере

Часть семантики – в коде интерпретатора

src/vm_x64.dasc:

```
|.ffunc coroutine_yield
|  mov L:RB, SAVE_L
|  test aword L:RB->cframe, CFRAME_RESUME
|  jz ->fff_fallback
|  mov L:RB->base, BASE
|  lea RD, [BASE+NARGS:RD*8-8]
|  mov L:RB->top, RD
|  xor RDd, RDd
|  mov aword L:RB->cframe, RD
|  mov al, LUA_YIELD
|  mov byte L:RB->status, al
|  jmp ->vm_leave_unw
```

Часть семантики – в коде рантайма

src/lib_base.c:

```
LJLIB_ASM(coroutine_yield)
{
  lj_err_caller(L, LJ_ERR_CYIELD);
  return FFH_UNREACHABLE;
}
```

JIT-компиляция стандартной библиотеки Lua

```
$ grep "static void LJ_FASTCALL recff_" lj_ffrecord.c
static void LJ_FASTCALL recff_nyi(jit_State *J, RecordFFData *rd)
static void LJ_FASTCALL recff_assert(jit_State *J, RecordFFData *rd)
static void LJ_FASTCALL recff_type(jit_State *J, RecordFFData *rd)
static void LJ_FASTCALL recff_getmetatable(jit_State *J, RecordFFData
*rd)
static void LJ_FASTCALL recff_setmetatable(jit_State *J, RecordFFData
*rd)
static void LJ_FASTCALL recff_rawget(jit_State *J, RecordFFData *rd)
static void LJ_FASTCALL recff_rawset(jit_State *J, RecordFFData *rd)
static void LJ_FASTCALL recff_rawequal(jit_State *J, RecordFFData *rd)
...
```

Обход таблиц

- Встроенная функция `next` реализована как fast function (интерпретатор + рантайм)
- Внутренний интерфейс `lj_tab_next` для поддержки `next` и `lua_next`
- Специализированный байт-код `ITERN` реализует ту же семантику, но использует обход с помощью сквозного индекса

Индексирование таблиц

- Байт-коды `TGET*`
- `rawget` реализована как fast function (интерпретатор + рантайм)
- Внутренние интерфейсы `lj_tab_get`, `lj_tab_getinth` и `lj_tab_getstr` для `lua_rawget`, `lua_rawgeti` и т.д.
- Поддержка индексирования в JIT-компиляторе:
  - Рекордер для байт-кодов
  - Рекордер для `rawget`
  - Поддержка генерации машинного кода

Индексирование таблиц: `lj_tab_get`

```
cTValue *lj_tab_get(lua_State *L, GCtab *t, cTValue *key)
{
  /* ... */
    n = hashkey(t, key);
    do {
      if (lj_obj_equal(&n->key, key))
        return &n->val;
    } while ((n = nextnode(n)));
  }
  return niltv(L);
}
```

Индексирование таблиц: asm_href

```
/* Inlined hash lookup.
** Specialized for key type and for const keys.
** The equivalent C code is:
**   Node *n = hashkey(t, key);
**   do {
**     if (lj_obj_equal(&n->key, key))
**       return &n->val;
**   } while ((n = nextnode(n)));
**   return niltv(L);
*/
static void asm_href(ASMState *as, IRIns *ir, IROp merge)
{
        /* ~150 строк кодогенератора для x86/x64 */
}
```

Листер байт-кода

```
$ luajit -bl -e \
> 'local s = 0; for i = 1,10 do s=s+i end; print(s)'
-- BYTECODE -- 0x00050fe0:0-1
0001    KSHORT   0   0
0002    KSHORT   1   1
0003    KSHORT   2  10
0004    KSHORT   3   1
0005    FORI     1 => 0008
0006 => ADDVV    0   0   4
0007    FORL     1 => 0006
0008 => GGET     1   0      ; "print"
0009    MOV      2   0
0010    CALL     1   1   2
0011    RET0     0   1
```

Листер байт-кода: Что внутри?

src/jit/bc.lua:

```
  local ma, mb, mc = band(m, 7), band(m, 15*8), band(m, 15*128)
  local a = band(shr(ins, 8), 0xff)
  local oidx = 6*band(ins, 0xff)
  local op = sub(bcnames, oidx+1, oidx+6)
  local s = format("%04d %s %-6s %3s ",
    pc, prefix or "  ", op, ma == 0 and "" or a)
  local d = shr(ins, 16)
  if mc == 13*128 then -- BCMjump
    return format("%s=> %04d\n", s, pc+d-0x7fff)
  end
```

Листер байт-кода: Альтернативный подход?

src/lj_bc.h:

```
/* Macros to get instruction fields. */
#define bc_op(i)      ((BCOp)((i)&0xff))
#define bc_a(i)       ((BCReg)(((i)>>8)&0xff))
#define bc_b(i)       ((BCReg)((i)>>24))
#define bc_c(i)       ((BCReg)(((i)>>16)&0xff))
#define bc_d(i)       ((BCReg)((i)>>16))
#define bc_j(i)       ((ptrdiff_t)bc_d(i)-BCBIAS_J)
```

Выводы

- Нарушение принципа DRY уменьшает поддерживаемость кода
- Основной довод в пользу нарушения DRY в LuaJIT – выигрыш в производительности
- Если выигрыша в производительности нет, то целесообразно обсуждать необходимость кода, нарушающего DRY

Ссылки

- http://luajit.org
- https://github.com/LuaJIT/LuaJIT/tree/master
- https://www.youtube.com/watch?v=8Q0KLTma_FA (Peter Cawley. LuaJIT, Something Interesting Inside)
- http://www.corsix.org/content/what-is-dynasm
- https://corsix.github.io/dynasm-doc/

Спасибо! Вопросы?

asoldatov@iponweb.net