mrubyの省メモリ化について考える

（当日版）

2018/08/31 SWEST 20

まつもと ゆきひろ
高橋征義
山根ゆりえ

今回のあらすじ

- これまでのあらすじ
- mrubyの「初期化時」における省メモリ化
- 省メモリ化する方法
- 省メモリ化できそうなところ
  - シンボルとメソッド表
- それ以外の手法について（参加者の方からご意見をいただきたいです）

これまでのあらすじ

- マイコンでmrubyを広めるには
- 国内外で入手しやすいマイコン上で動かす
    - 価格面、入手容易性
- （実行速度等より）メモリ消費が問題
    - 特にRAM

今回検討する
メモリ消費タイミング

- 初期化時メモリ消費　⇐　今回は主にこちら
- 実行時メモリ消費

mrubyの初期化時に
おける省メモリ化

「初期化時」とは

```c
#include <mruby.h>
#include <mruby/compile.h>

int main(void) {
  mrb_state *mrb = mrb_open();
  if (!mrb) { /* handle error */ }
  mrb_load_string(mrb, "puts 'hello world'");
  mrb_close(mrb);
  return 0;
}
```

このタイミングで
なんとかしたい

（こちらは対象外）

helloworld.c

mrb_open()の概略

```
{
  mrb_gc_init(mrb, &mrb->gc); /* GC */
  mrb_init_symtbl(mrb); /* シンボル表 */

  mrb_init_class(mrb); /* 基盤(Class) */
  mrb_init_object(mrb); /* 基盤(Object) */
  mrb_init_kernel(mrb); /* 基盤(Kernel) */

  mrb_init_XXXX(mrb); /*クラスライブラリ(C)*/
  mrb_init_mrblib(mrb); /*クラスライブラリ(Ruby)*/
  mrb_init_mrbgems(mrb); /*拡張ライブラリ*/
}
```

最適化できるのでは🤔

- やるべきことは事前に（実行前に）分かっている
    - クラスライブラリ、拡張ライブラリも分かっている

静的な情報はROMに置いたりできそう

なぜそうしていないのか

- あまり意識していなかった？
    - アプリケーション組込みだとあまり関係なさそう
- Rubyの言語的な性質による

Rubyの特徴

- 宣言がない
  - 変数の宣言はなく、代入で初期化される
  - クラスやメソッド定義も「式」として実行される
- クラスもオブジェクト
  - クラスはClassクラスのインスタンス
  - オープンクラス（実行中にもメソッド追加可能）

Rubyと宣言と式

- 変数宣言はなく、代入で初期化される
- クラスやメソッド定義も「式」として実行される
- →「初期化」と「実行」の区別があまりない
- 現状の実装でも「初期化」時に「実行」してる

クラスもオブジェクト

- String、IOなどの各クラスは、Classクラスのインスタンスとして実装されている
- 各クラスを生成する段階で、オブジェクトを作ることになる
- クラス定義後にメソッドを追加することも可能
- 「初期化」時に決まらない

初期化の範囲

```
{
  mrb_gc_init(mrb, &mrb->gc); /* GC */
  mrb_init_symtbl(mrb); /* シンボル表 */

  mrb_init_class(mrb); /* 基盤(Class) */
  mrb_init_object(mrb); /* 基盤(Object) */
  mrb_init_kernel(mrb); /* 基盤(Kernel)

  mrb_init_XXXX(mrb); /*クラスライブラリ(C)*/
  mrb_init_mrblib(mrb); /*クラスライブラリ(Ruby)*/
  mrb_init_mrbgems(mrb); /*拡張ライブラリ*/
}
```

普通に初期化

初期化の範囲

```
{
  mrb_gc_init(mrb, &mrb->gc); /* GC */
  mrb_init_symtbl(mrb); /* シンボル表 */

  mrb_init_class(mrb); /* 基盤(Class) */
  mrb_init_object(mrb); /* 基盤(Object) */
  mrb_init_kernel(mrb); /* 基盤(Kernel) */

  mrb_init_XXXX(mrb); /*クラスライブラリ(C)*/
  mrb_init_mrblib(mrb); /*クラスライブラリ(Ruby)*/
  mrb_init_mrbgems(mrb); /*拡張ライブラリ*/
}
```

やや特殊な処理

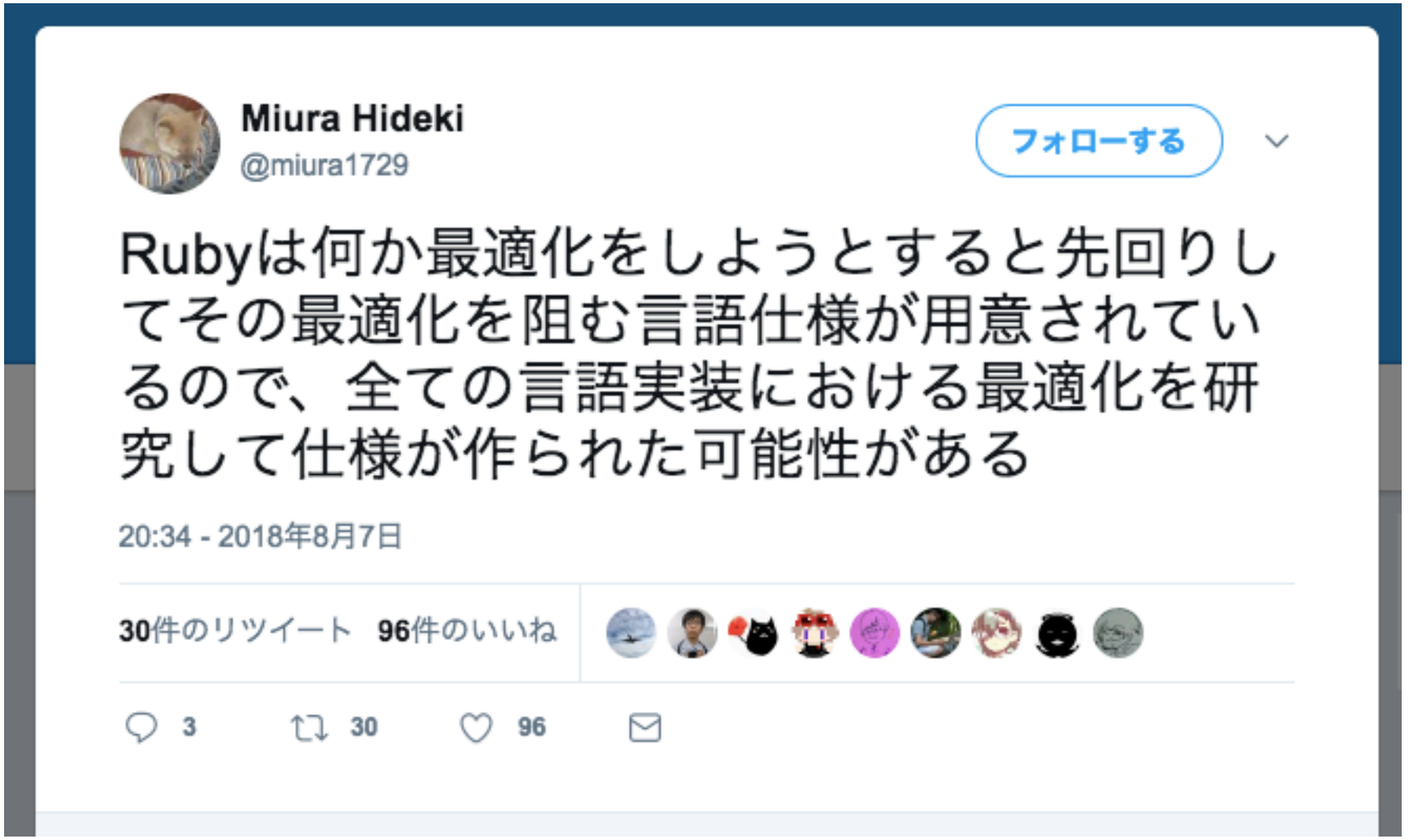
初期化の範囲

```
{
  mrb_gc_init(mrb, &mrb->gc); /* GC */
  mrb_init_symtbl(mrb); /* シンボル表 */

  mrb_init_class(mrb); /* 基盤(Class) */
  mrb_init_object(mrb); /* 基盤(Objec
  mrb_init_kernel(mrb); /* 基盤(Kernel

  mrb_init_XXXX(mrb); /*クラスライブラリ(C)*/
  mrb_init_mrblib(mrb); /*クラスライブラリ(Ruby)*/
  mrb_init_mrbgems(mrb); /*拡張ライブラリ*/
}
```

実行と変わらない

なぜmrubyは初期化時にもメモリを消費するのか

- Rubyには素朴な意味での「初期化」があまりない
- クラス・メソッド定義を含め、すべては実行文（式）
  - 実行時に評価されるのと同様の挙動になる
  - 素朴に実装すると、mruby処理系内でメソッド定義式と同等のC関数が実行されることになる



https://twitter.com/miura1729/status/1026793694991593472

省メモリ化
できそうなこと

もっとROMを使いたい

- ROMを活用する
    - 事前に決定している情報についてはROMに配置させるようにする
    - 実行時に追加・変更される可能性があっても、初期化時の置き場所と実行時の置き場所を分けて、うまいこと辻褄があうようにする

mrubyとROM

- すでに使っているもの
  - 文字列リテラルの確保(コンパイルオプションで最適化)
- バイトコードの実体
  - ただし、初期化時にはバイトコードに対応するirep構造体が作られ、実行時にはこちらが使われる

mrubyとROM

- 新たにROMに置けそうなもの
  - シンボル
  - メソッドテーブル
- 他には？

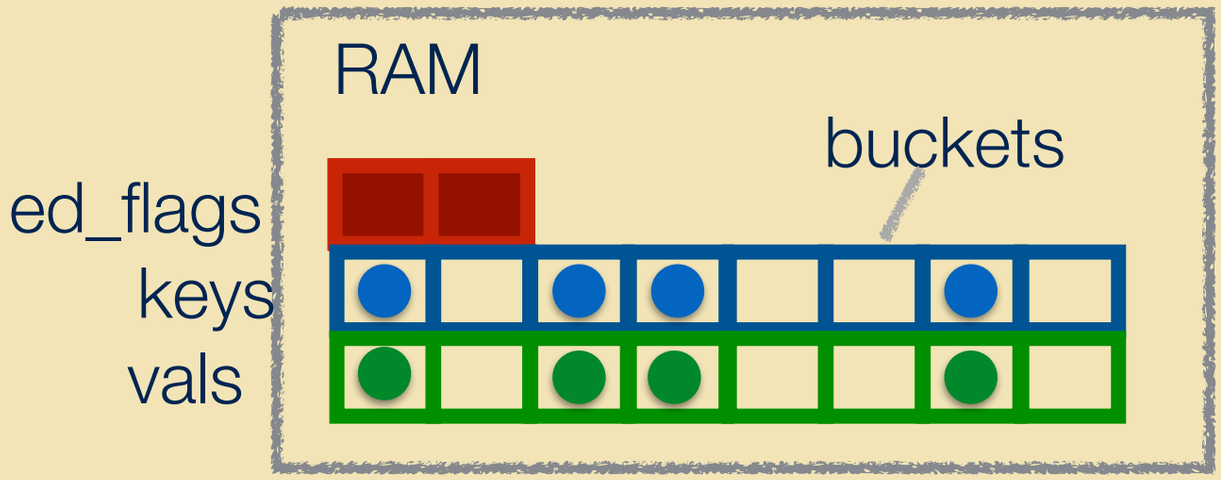
Rubyとシンボル

- シンボルは元々処理系の内部表現だった
- 最近は「変更できない(immutableな)文字列」として使われたりする
- 低コストで操作できる（？　最近は文字列も効率化されている？）

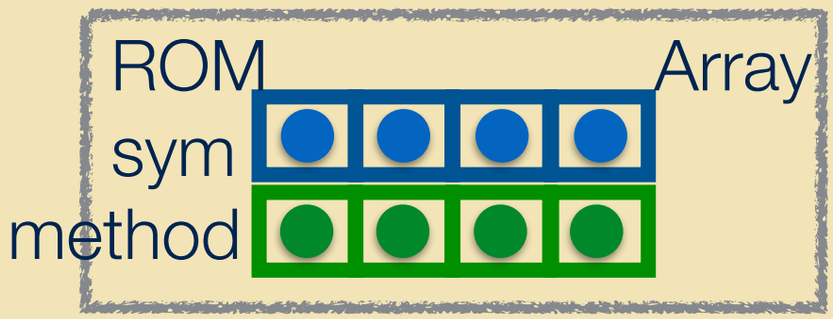
khash.h

- mrubyのhashmap/hashset構造体操作マクロ群
- いわゆるハッシュマップのデータを管理する
    - アクセスは速い（衝突がなければ O(1) ）
    - メモリ消費量は大きい（余計なbucketsが必要）
- mruby内では汎用Key-Value Store的に多用されている
    - シンボルやメソッドテーブルにも使われている

hash vs array




```
typedef struct kh_mt{
  khint_t n_buckets;
  khint_t size;
  khint_t n_occupied;
  uint8_t *ed_flags;
  mrb_sym *keys;
  mrb_method_t *vals;
} kh_mt_t;
```




```
struct method_table{
  mrb_sym sym;
  mrb_method_t method;
};
```

シンボルのROM化

- ソースに埋め込まれているシンボルをROMに入れる
- 事前に判別できる→khashも不要
  - mrubyではlexerの作成にgperfを使っている
  - これをシンボルにも利用し、名前→シンボルのインデックスもROMに置くようにする

シンボルの検出

- ソースコードをスキャンする
  - Cの解析はpreprocessorも絡むので厳密には困難なので、正規表現でざっくり解析する
- mrbファイルを作って抽出
  - Cで実装した部分については使えない
- mrubyを実際に起動して生成されたシンボルを動的に抽出
  - ターゲットが組込みのmrubyでは困難

メソッド表のROM化

- 各クラスにメソッドが定義されている
- 実装としては、各クラスオブジェクト構造体にメソッド表へのリンクがある
- メソッド表の実体はkhashで実装
- 実行時に追加されることもある
    - が、稀な場合も多いのでROM化したい

メソッド表のROM化

- メソッド表：メソッド名（シンボル）とメソッド定義の実体（アドレス）
    - ↑このためにもシンボルをROM化する必要があった
- 単なる配列か、バランス木等で実装できそう

（続き）

やりたいこと

できるか分からないこと

- もっとROMに置く
  - RClassのROM化
  - irepのROM化
- メモリ(RAM)使用量を減らす
  - バイトコード化(mruby 2.0)
  - khash.hをtree等に置き換える
  - irepを削る

RClassのROM化

- RClass: 各クラスの構造体
  - 共通ヘッダ
  - インスタンス変数テーブルへのポインタ
  - メソッドテーブルへのポインタ
  - スーパークラスへのポインタ

RClassのROM化

- クラスをfreezeしたことにすれば可能？

```
#define MRB_OBJECT_HEADER \
  enum mrb_vtype tt:8;\
  uint32_t color:3;\
  uint32_t flags:21;\
  struct RClass *c;\
  struct RBasic *gcnext

struct RClass {
  MRB_OBJECT_HEADER;
  struct iv_tbl *iv;
  struct kh_mt *mt;
  struct RClass *super;
};
```

irepのROM化

- irep: バイトコードの情報をいったん構造体にしたもの

```
typedef struct mrb_irep {
  uint16_t nlocals;
  uint16_t nregs;
  uint8_t flags;

  mrb_code *iseq;
  mrb_value *pool;
  mrb_sym *syms;
  struct mrb_irep **reps;

  struct mrb_locals *lv;
  /* debug info */
  mrb_bool own_filename;
  const char *filename;
  uint16_t *lines;
  struct mrb_irep_debug_info* debug_info;

  uint16_t ilen, plen, slen, rlen, refcnt;
} mrb_irep;
```

バイトコード化(mruby 2.0)

The new bytecode

We will reimplement VM to use 8bit instruction code. By bytecode, we mean real byte code. The whole purpose is reducing the memory consumption of mruby VM.

Instructions

Instructions are bytes. There can be 256 instructions. Currently we have 94 instructions. Instructions can take 0 to 3 operands.

operands

The size of operands can be either 8bits, 16bits or 24bits. In the table.1 below, the second field describes the size (and sign) of operands.

- B: 8bit
- sB: signed 8bit
- S: 16bit
- sS: signed 16bit
- W: 24bit

First two byte operands may be extended to 16bit. When those byte operands are bigger than 256, the instruction will be prefixed by `OP_EXT1` (means 1st operand is 16bit) or `OP_EXT2` (means 2nd operand is

バイトコード化(mruby 2.0)

table.1 Instruction Table

Instruction Name	Operand type	Semantics
OP_NOP	-	
OP_MOVE"	BB	R(a) = R(b)
OP_LOADL"	BB	R(a) = Pool(b)
OP_LOADI"	BsB	R(a) = mrb_int(b)
OP_LOADI_0'	B	R(a) = 0
OP_LOADI_1'	B	R(a) = 1
OP_LOADI_2'	B	R(a) = 2
OP_LOADI_3'	B	R(a) = 3
OP_LOADSYM"	BB	R(a) = Syms(b)
OP_LOADNIL'	B	R(a) = nil
OP_LOADSELF'	B	R(a) = self
OP_LOADT'	B	R(a) = true
OP_LOADF'	B	R(a) = false
OP_GETGV"	BB	R(a) = getglobal(Syms(b))

khashのTree化

- hashはメモリ消費量が大きいので何とかしたい
- 別のデータ構造で置き換える
  - splay tree / treap / red-black tree
- khashのAPIをもうちょっと一般化したい
  - kh_foreach{…}

データ構造比較

	khash	Tree	Array
挿入	O(1)	O(log n)	O(1)
検索	O(1)	O(log n)	O(n)
メモリ	大	中	小

irepのスリム化

```c
typedef struct mrb_irep {
  uint16_t nlocals;
  uint16_t nregs;
  uint8_t flags;

  mrb_code *iseq;
  mrb_value *pool;
  mrb_sym *syms;
  struct mrb_irep **reps;

  struct mrb_locals *lv;
  /* debug info */
  mrb_bool own_filename;
  const char *filename;
  uint16_t *lines;
  struct mrb_irep_debug_info* debug_info;

  uint16_t ilen, plen, slen, rlen, refcnt;
} mrb_irep;
```

mruby

```c
typedef struct IREP {
  uint16_t nlocals;   //!< # of local variables
  uint16_t nregs;   //!< # of register variables
  uint16_t rlen;    //!< # of child IREP blocks
  uint16_t ilen;    //!< # of irep
  uint16_t plen;    //!< # of pool

  uint8_t     *code;    //!< ISEQ (code) BLOCK
  mrbc_object  **pools; //!< array of POOL objects pointer.
  uint8_t     *ptr_to_sym;
  struct IREP **reps;   //!< array of child IREP's pointer.

} mrbc_irep;
typedef struct IREP mrb_irep;
```

mruby/c