SWIG+Ruby

- 京都大学 法学部 2回生
- 小塚真啓 (KOZUKA Masahiro)

2002/3/22

0. Rubyとは?

- スクリプト言語でオブジェクト指向
- お手軽、直感的
- 今更語ることはないでしょう:-)

0. Rubyの拡張ライブラリ

- 既存のライブラリのラップをする場合、Cの知識があればそれほど難しくない(構造の設計は悩ましいけど)
- ただ、難しくはないが同じようなコードをたくさん書かないといけない。

0. SWIGとは?

- Simplified Wrapper and Interface Generator
- CやC++で書かれたライブラリをスクリプト言語から利用するためのコードを自動生成するツール
- Rubyにも対応

0. Ruby + SWIG

- ライブラリのラップで楽できる
- 既存の有益なライブラリをRubyで楽に利用できる
- ドキュメントが整備されてないのがちょっとつらい
- で、いくつかチュートリアルを書いてみました。
(http://kozuka.jp/doc/swig/)

0. チュートリアル

- チュートリアルのうち、いくつかを解説します。

- 1. 基本的な使い方
- 2. FOTRAN関数を使ったポインタの扱い
- 3. NArrayを題材としてRubyとCで引数の数を変える
- 4. 構造体をRubyクラスにラップする例

- ラッピング作業に焦点をあてます。インストール方法などは割愛します。

1. 基本的な使い方(1)

- 次の関数をRubyから使えるようにしてみます。

```
int add(int a, int b) {
  return(a + b);
}
```

1. 基本的な使い方(2)

- add関数をTestというモジュールの関数にする

```
%module test
int add(int a, int  b);
```

- 「%module test」でTestというモジュールを作る
- 列挙された関数がTestの関数になる
- 列挙はプロトタイプの形で

1. 基本的な使い方(3)

- 生成されるCのコード(抜粋)

```
static VALUE _wrap_add(VALUE self, VALUE varg0, VALUE varg1) {
  int arg0 ;
  int arg1 ;
  int result ;
  VALUE vresult = Qnil;

  arg0 = NUM2INT(varg0);
  arg1 = NUM2INT(varg1);
  result = (int )add(arg0,arg1);
  vresult = INT2NUM(result);
  return vresult;
}
```

- コンパイルする
- p Test.add(1,2) => 3 こんな感じで使えます。

2. FORTRANの関数(1)

- 以下のFORTRAN関数をRubyから使えるようにしてみる

```
INTEGER FUNCTION ADD(A,B)
  INTEGER A,B
  ADD=A+B
  RETURN
END
```

- Cからは以下のように見える

```
int add_(int *a, int *b);
```

2. FORTRANの関数(2)

- add_関数をTest2というモジュールの関数にする(だめな例)

```
%module test2

int add_(int *a, int *b);
```

2. FORTRAN関数(3)

- 生成されるコード(抜粋、書式を変更)

```
static VALUE _wrap_add_(VALUE self, VALUE varg0, VALUE varg1) {
    int *arg0 ;
    int *arg1 ;
    int result ;
    VALUE vresult = Qnil;

    arg0 = (int *)SWIG_ConvertPtr(varg0, SWIGTYPE_p_int); //int *のオブジェクトから変換
    arg1 = (int *)SWIG_ConvertPtr(varg1, SWIGTYPE_p_int); // 同上
    result = (int )add2(arg0,arg1);
    vresult = INT2NUM(result);
    return vresult;
}
```

- 使えないこともないが、すごく不便
- p Test2.add_(1, 2) # =>

in `add_': Expected int * (TypeError)

2. FORTRAN関数(4)

- SWIGは引数に与えられたRubyのオブジェクトを関数の引数にあう型へ変換するコードを埋め込む。

- ただ、デフォルトではその種類は少ない。
次に示すようにint *のようなものでもダメ。
- 「この引数の場合は次の変換コードを埋め込め」とtypemapという機能で指定することができる。
- 基本的なものはtypemapsとしてまとめられている。

- typemapsを使って先ほどの関数をラップし直します。

2. FORTRAN関数(5)

- 渡したいのはInteger ポインタへ自動変換してくれると助かる 以下のように書き直す

```
%module test2
%include typemaps.i

int add2(int *INPUT, int *INPUT);
```

- 引数名にint *INPUTを指定するとIntegerから変換してくれるようになる
- 他にもOUTPUTやINOUTが指定可能

2. FORTRAN関数(6)

- Integerからintへ変換し、そのポインタを与えるようになっている。

```
static VALUE _wrap_add2(VALUE self, VALUE varg0, VALUE varg1) {
    int *arg0, *arg1, temp, temp0, result ;
    VALUE vresult = Qnil;
    {
        temp = NUM2INT(varg0);
        arg0 = &temp;
    }
    {
        temp0 = NUM2INT(varg1);
        arg1 = &temp0;
    }
    result = (int )add2(arg0,arg1);
    vresult = INT2NUM(result);
    return vresult;
}
```

3. NArrayを使う (1)

- ご存知、多次元 数値 配列クラス
- ラップする関数にNarrayを渡すことができれば便利

3. NArrayを使う (2)

- 以下の関数を題材とします。

```
char *trans(char *original, int width, int height) {
  double cosTH, sinTH; int x, y, origin_x, origin_y, central_x, central_y;
  char *output; cosTH = 0.0; sinTH = 1.0;

  if (width % 2 == 0 || height % 2 == 0) return(NULL);
  central_x = (width - 1) / 2; central_y = (height -1) / 2;
  output = (char *)malloc(width * height * sizeof(char));
  for (y = 0; y < height; y++) {
    for (x = 0; x < width; x++) {
      origin_x = (x - central_x)*cosTH + (y - central_y)*sinTH + central_x;
      origin_y = -(x - central_y)*sinTH + (y - central_y)*cosTH + central_y;
      output[width * y + x] = original[width * origin_y + origin_x];
    }
  }
  return(output); }
```

3. NArrayを使う (3)

- 長いので分けて説明します。

```
%module test3
%{
  #include "narray.h“
  VALUE cNArray;
  typedef char char_2dimensions;
%}
%typemap(ruby,ignore) int FIRST_LEN(int *first_length) {
  first_length = &$target;
}
%typemap(ruby,ignore) int SECOND_LEN(int *second_length) {
  second_length = &$target;
}
```

- 長さはNArrayから得ることができるからignoreする
- NArrayはchar_2dimentionsから与える

3. NArrayを使う (4)

- -前から続く-

```
%typemap(ruby,in) char_2dimensions *ARRAY(VALUE source) {
  struct NARRAY *narray;
  source = na_cast_object($source, NA_BYTE);
  GetNArray(source, narray);
  $target = (char *)NA_PTR(narray, 0);
  *first_length = narray->shape[0];
  *second_length = narray->shape[1];
}
```

- {first,second}_lengthは先のignoreでtrans()の引数に結び付けている

3. NArrayを使う (5)

- -前から続く-

```
%typemap(ruby,out) char_2dimensions * {
  struct NARRAY *narray;
  int rank, i;
  int *shape;
  rank = 2; shape = ALLOC_N(int, rank);
  shape[0] = *first_length;
  shape[1] = *second_length;
  $target = na_make_object(NA_BYTE, rank, shape, cNArray);
  GetNArray($target, narray);
 memmove(narray->ptr, $source, narray->total);
}
char_2dimensions *trans
(char_2dimentions *Array, int FIRST_lEN, int SECOND_LEN);
```

- outは返り値にマッチ

3. NArrayを使う (6)

NArrayの内部データ構造

2002/3/22

4.構造体をクラスにラップする (1)

- GNU Scientific Library (GSL)を題材とする
- GSLのライブラリ関数の特徴は処理するデータの構造体を渡す点。
- gsl_vector構造体等をVectorクラスにラップしてVector処理の関数をメソッドにしていってやれば使いやすそう。

4.構造体をクラスにラップする (2)

- 構造体を取り扱うパターンは4つ。

  - self selfオブジェクトから構造体を取り出し、関数に与える。
    データの変更を行わない関数に用いる。

  - self_return selfオブジェクトから構造体を取り出し、直接関数に与える。
    データの変更を行う関数であれば破壊的なメソッドになる。

  - self_return_cp selfオブジェクトから構造体を取り出しコピーを関数に与える。
    データの変更を行う関数でも破壊的メソッドにならない。

  - self_return_alloc selfオブジェクトから取り出した構造体と同じサイズの構造体を作成し、関数に与える。
    memcpyに用いる。

4.構造体をクラスにラップする (3)

- 構造体を複製するために_memcpyという関数がある。
- ラップする場合、selfオブジェクトから元となる構造体を取り出しコピーした構造体をオブジェクトにラップして返すようなものが理想だろう。(深いコピーをするcloneとして)

- その場合、selfとself_return_allocを混ぜる。
- どちらもRubyからは引数をとらないのでignore
- しかし、ignoreは関数ごとに1つのみ
- 両方でignoreにselfオブジェクトから構造体をもらうコードを埋め込むのは不可能。

4.構造体をクラスにラップする (4)

- arginitとignoreに分離してしのぐ

```
%typemap(arginit) gsl_vector *self {
  Data_Get_Struct(self, $1_basetype, $1);
}
%typemap(ignore) gsl_vector *self,
                 gsl_vector *self_return {}
%typemap(ignore) gsl_vector *self_return_alloc {
  $1_basetype *self_vector;
  Data_Get_Struct(self, $1_basetype, self_vector);
  $1 = $1_basetype_alloc(self_vector->size);
}
%typemap(ignore) gsl_vector *self_return_cp {
  $1_basetype *self_vector;
  Data_Get_Struct(self, $1_basetype, self_vector);
  $1 = $1_basetype_alloc(self_vector->size);
  $1_basetype_memcpy($1, self_vector);
}
```

4.構造体をクラスにラップする (5)

- 関数の引数をメソッドの返り値にする

```
%typemap(argout) gsl_vector *self_return {
  $result = self;
}
```

- self_returnは自身の構造体が変わっているのでそのままself(オブジェクト)を渡す

```
%typemap(argout) gsl_vector *self_return_alloc,
                 gsl_vector *self_return_cp {
  $result = Data_Wrap_Struct(c$1_basetype, 0, $1_basetype_free, $1);
}
```

- self_return_alloc self_rerturn_cpは新たな構造体が割り当てられたのでクラスを作り直す

4.構造体をクラスにラップする (6)

```
%typemap(in) gsl_vector *input {
  Data_Get_Struct($input, $1_basetype, $1);
}
```

- inputは他のオブジェクトから構造体を受け取る

```
%typemap(out) gsl_vector * {
  $result = Data_Wrap_Struct(c$1_basetype, 0, $1_basetype_free, $1);
}
```

- 関数の返り値が構造体という場合に対応

5. まとめ

- SWIGはデフォルトのままではあまり使えない。
- が、強力なtypemap機能があるので工夫次第で非常に便利なツールとなる。