# Gura 開発者向けマニュアル

Updated: June 25, 2012

# 目次

## 1. この文書について

Gura の本体およびモジュールのビルド方法と、Gura のライブラリやインクルードファイルが提供する関数・マクロ・クラスについて説明します。

内容は Gura v0.3.0 の実装に基づきます。

## 2. ソースファイルの入手方法

Gura のソースファイルは、tar ボールのダウンロードまたはレポジトリからのチェックアウトで入手することができます。

### 2.1. tar ボールのダウンロード

ソースファイルをまとめた tar ボールが、SourceForge.JP のダウンロードページから取得できます。URL は以下の通りです。

http://sourceforge.jp/projects/gura/releases/

### 2.2. レポジトリからのチェックアウト

Gura のソースファイルは SourceForge.JP の Subversion レポジトリで管理されています。開発中のソースファイルは trunk レポジトリ内にあります。以下のようにチェックアウトしてください。

```
$ svn co http://svn.sourceforge.jp/svnroot/gura/trunk gura
```

編集権限を持っている場合は、以下の URI からチェックアウトします。

```
$ svn co https://svn.sourceforge.jp/svnroot/gura/trunk gura
```

## 3. ソースファイルのディレクトリ構成

ソースファイルのディレクトリは以下のようになっています。

| ディレクトリ | 内容 |
|---|---|
| application | Guraスクリプトで作成した実用的なアプリケーション |
| bin-x64 | 64bit版プログラムの格納ディレクトリ |
| bin-x86 | 32bit版プログラムの格納ディレクトリ |
| dist | インストーラや tar ボールを作成する作業ディレクトリ |
| doc | ドキュメント |
| guest | Windowsで使用するDLLファイルやインクルードファイルなど |
| include | Gura のインクルードファイル |
| lib | Gura のライブラリファイル |
| module | スクリプトモジュールファイル。<br>Windowsの場合、バイナリモジュールファイルもここに格納します。 |
| sample | Guraのサンプルスクリプト |
| scripts | 作業用スクリプト |
| src | ソースファイル |
| test | テスト用スクリプト |

## 4. 開発環境

以下の開発環境でビルドできます。Visual Studio は無償の Express 版も使用可能です。

- Windows (Visual Studio 2010)
- Ubuntu Linux (gcc)

# 5. ビルド方法

以下、Windows と Linux のコンソールプロンプトをそれぞれ ">" および "$" で表します。

## 5.1. Windows

### 5.1.1. 事前準備

guest ディレクトリにある setup.bat を実行します。レポジトリから必要なパッケージをエキスポートし、ビルドします。

### 5.1.2. Visual Studio C++

最上位ディレクトリにある Visual Studio ソリューションファイル gura.sln を Visual Studio 2010 で開き、アクティブソリューション構成を "Release" にしてビルドします。

各ディレクトリに以下のファイルが生成されます。

| ディレクトリ | ファイル |
|---|---|
| gura | gura.exe, guraw.exe, guraole.dll, libgura.dll |
| gura¥lib | libgura.lib |
| gura¥module | バイナリモジュール (*.gurd) |

## 5.2. Linux (gcc)

autoconf/automake 関連のファイルを使用します。コンソールを開き、カレントディレクトリを gura/src に移動してから以下のコマンドを実行してください。

```
$ ./configure
$ make
$ sudo make install
```

各ディレクトリに以下のファイルがインストールされます。

| ディレクトリ | ファイル |
|---|---|
| /usr/local/bin | gura |
| /usr/local/lib | libgura.so |
| /usr/local/lib/gura | スクリプトモジュール (*.gura) |
| /usr/local/include | gura.h |
| /usr/local/include/gura | gura.h からインクルードされるヘッダファイル |
| /usr/local/share/gura | サンプルスクリプトなど |

続けて、モジュールのインストールを行います。同じく src ディレクトリで以下のコマンドを実行してください。

```
$ gura build_modules.gura
$ sudo build_modules.gura install
```

これで、/usr/local/lib/gura にバイナリモジュールがインストールされます。エラーが出る場合は、必要なライブラリがシステムにインストールされていない可能性があります。エラーメッセージに必要な Debian パッケージ名が表示されるので、それに基づいてインストールしてください。

# 6. バイナリモジュール開発

Gura は、バイナリモジュールを開発するためのフレームワークを用意しています。以下、バイナリモジュール hoge を作る過程を見ていきます。

## 6.1. 雛形の作成

コンソールを開き、適当な作業用ディレクトリを作成した後、そのディレクトリ内で以下のコマンドを実行します。

```
$ gura -i lets_module hoge
```

ビルド用スクリプト build.gura とソースファイルの雛形 Module_hoge.cpp が生成されます。

階層構造の下にモジュールを作成するときは、親のモジュール名と本体のモジュール名を引数に指定します。以下に例を示します。

```
$ gura -i lets_module net hoge
```

## 6.2. ビルド方法

以下のコマンドを実行すると、ソースファイルのコンパイルおよびリンクを行ってバイナリモジュール hoge.gurd を生成します。

```
$ gura build.gura
```

バイナリモジュールを出力するディレクトリは開発環境によって異なり、以下のようになります。

| | |
|---|---|
| Visual Studio C++ | msc |
| gcc | gcc |

実際の開発プロセスではビルドとテストを繰り返すことになります。モジュールのサーチパスにはカレントディレクトリが含まれるので、バイナリモジュールがソースファイルと同じディレクトリに出力されると便利です。そのようなときは以下のように --here オプションをつけてビルドします。

```
$ gura build.gura --here
```

## 6.3. インストール方法

モジュールを Gura のディレクトリにインストールするときは、以下のコマンドを実行します。

```
$ sudo gura build.gura install
```

## 6.4. モジュールソースファイルの内部構成

自動生成した雛形をもとに、モジュールソースファイルの内部構成を見ていきます。以下のソースは、自動生成されたソースからコメントを取り除いたものです。

```
1  #include <gura.h>
2
```

```
 3  Gura_BeginModule(hoge)
 4
 5  Gura_DeclareFunction(test)
 6  {
 7      SetMode(RSLTMODE_Normal, FLAG_None);
 8      DeclareArg(env, "num1", VTYPE_number);
 9      DeclareArg(env, "num2", VTYPE_number);
10      SetHelp("adds two numbers and returns the result.");
11  }
12
13  Gura_ImplementFunction(test)
14  {
15      return Value(args.GetNumber(0) + args.GetNumber(1));
16  }
17
18  Gura_ModuleEntry()
19  {
20      Gura_AssignFunction(test);
21  }
22
23  Gura_ModuleTerminate()
24  {
25  }
26
27  Gura_EndModule(hoge, hoge)
28
29  Gura_RegisterModule(hoge)
```

1行　　全てのモジュールは gura.h をインクルードします。

3行　　Gura_BeginModuleマクロでモジュール実装の開始を宣言します。

5-11行　Gura_DeclareFunctionマクロで関数の宣言をし、戻り値や関数のタイプ、引数の名前や型、ヘルプなどを定義します。

13-16行　Gura_DeclareFunctionの内容で宣言した関数の実行内容をGura_ImplementFunctionマクロに続いて記述します。

18-21行　Gura_ModuleEntryでモジュールをインポートしたときに実行する内容を記述します。この中には、関数や変数のアサイン、シンボル定義、クラス定義などが含まれます。

23-25行　Gura_ModuleTerminateでモジュールを削除したときに実行する内容を記述します。

27行　　Gura_EndModuleマクロでモジュール実装の終了を宣言します。
Gura_BeginModuleからGura_EndModuleまでが一つのモジュールの実装単位になります。

29行　　Gura_RegisterModuleで、実装したモジュールの登録を行います。

# 7. C++ インターフェース

Gura のライブラリやインクルードファイルが提供する関数・マクロ・クラスについて説明します。

## 7.1. モジュールのフレームワークを構成する要素

`Gura_BeginModule(name)`

モジュールの開始を宣言します。name にはモジュール名を指定します。

`Gura_EndModule(name, alias)`

モジュールの終了を宣言します。name には Gura_BeginModule で指定したものと同じ名前を渡します。alias は通常は name と同じものを指定します。階層構造を持ったモジュールの場合、alias はモジュールのベース名を指定します。

`Gura_ModuleEntry()`

モジュールをインポートしたときに呼ばれる関数を定義します。関数の内容を、このマクロに続いて "{" と "}" の間に記述します。

この関数の中では以下の変数が参照できます。

| | |
|---|---|
| env | Environment インスタンスの参照です。 |
| sig | Signal インスタンスです。 |

`Gura_ModuleTerminate()`

モジュールを解放したときに呼ばれる関数を定義します。関数の内容を、このマクロに続いて "{" と "}" の間に記述します。

`Gura_RegisterModule(name)`

モジュールを登録します。name は Gura_BeginModule で指定したものを渡します。

## 7.2. シンボル定義

`Gura_DeclareUserSymbol(name)`

モジュール内で使用するシンボルを宣言します。シンボルはスクリプト全体で管理されるので、すでに Gura 本体で宣言されているものと同じ名前のシンボルをここで宣言しても、メモリ効率などに影響しません。

`Gura_RealizeUserSymbol(name)`

Gura_DeclareUserSymbol で宣言したシンボルを生成します。通常 Gura_ModuleEntry の関数内に記述します。

## 7.3. 関数定義

`Gura_DeclareFunction(funcName)`

関数の宣言をします。この内部には、関数の戻り値の扱い・動作モード・引数宣言・受け付けるアトリビュートシンボルの宣言が記述されます。

`Gura_ImplementFunction(funcName)`

関数の処理内容を、このマクロに続いて "{" と "}" の間に記述します。

`Gura_AssignFunction(name)`

通常 `Gura_ModuleEntry` の関数内に記述します。

### 7.4. クラス定義

`Gura_DeclareUserClass(className)`

クラスの宣言をします。

`Gura_ImplementUserClass(className)`

クラスの内容を、このマクロに続いて "{" と "}" の間に記述します。

`Gura_RealizeUserClass(classsName, str, pClassBase)`

通常 `Gura_ModuleEntry` の関数内に記述します。

### 7.5. メソッド定義

`Gura_DeclareMethod(className, methodName)`

メソッドの宣言をします。この内部には、メソッドの戻り値の扱い・動作モード・引数宣言・受け付けるアトリビュートシンボルの宣言が記述されます。

`Gura_ImplementMethod(className, methodName)`

メソッドの処理内容を、このマクロに続いて "{" と "}" の間に記述します。

`Gura_AssignMethod(className, methodName)`

通常 `Gura_ImplementUserClass` の関数内に記述します。

### 7.6. 引数宣言

引数宣言を行う C++のメンバ関数は、Function クラス内で以下のように定義されています。

```
Declaration *DeclareArg(Environment &env, const char *name, ValueType valType,
    OccurPattern occurPattern = OCCUR_Once, unsigned long flags = FLAG_None,
    Expr *pExprDefault = NULL);
```

引数の意味は以下の通りです。

| 引数 | 内容 |
|---|---|
| env | この引数にはGura_DeclareFunctionまたはGura_DeclareMethod内で暗黙的に定義される変数envを渡します。 |
| name | 引数の名前です。 |
| valType | Guraのプログラムで使われている型名に、VTYPE_ というプレフィックスをつけたものを指定します。例えば、number型の引数を定義する場合は、VTYPE_number を指定します。<br>任意の型を受け取る引数には、VTYPE_any を指定します。 |

| | |
|---|---|
| occurPattern | オプショナル引数や可変長引数の宣言をします。<br>OCCUR_Once　通常の引数指定<br>OCCUR_ZeroOrOnce　オプショナル引数。? をつけたのと同じ。<br>OCCUR_ZeroOrMore　0個以上の可変長引数。* をつけたのと同じ。<br>OCCUR_OnceOrMore　0個以上の可変長引数。+ をつけたのと同じ。 |
| flags | 引数のフラグを指定します。<br>FLAG_List　リストを受け取ります。引数名に [] をつけたのと同じです。<br>FLAG_NoMap　アトリビュート :nomap をつけたのと同じ。<br>FLAG_Nil　アトリビュート :nil をつけたのと同じ。<br>FLAG_Read　アトリビュート :r をつけたのと同じ。<br>FLAG_Write　アトリビュート :w をつけたのと同じ。 |
| pExprDefault | 引数に演算子 => でデフォルト値をつけたときの代入要素 |

例えば、以下のような Gura 関数の宣言を考えてみます。

```
Hoge(x:number, y:number, z[]:number)
```

これに対応する C++のプログラムは以下のようになります。

```
DeclareArg(env, "x", VTYPE_number, OCCUR_Once, FLAG_None, NULL);
DeclareArg(env, "y", VTYPE_number, OCCUR_Once, FLAG_None, NULL);
DeclareArg(env, "z", VTYPE_number, OCCUR_Once, FLAG_List, NULL);
```