CEDEC
CESA Developers Conference 2009

Squirrelを使ったゲーム開発 Part Ⅱ

株式会社スクウェア・エニックス
北出　智
神尾隆司

アジェンダ

- 今回の発表の背景
  - Wiiウェア「光と闇の姫君と世界征服の塔
    ファイナルファンタジー・クリスタルクロニクル」
- Squirrelとは
- 開発事例
  - 「光と闇の姫君と世界征服の塔
    ファイナルファンタジー・クリスタルクロニクル」（※以下 光と闇の姫君)
    での実例と工夫
  - 前作「小さな王様と約束の国
    ファイナルファンタジー・クリスタルクロニクル」（※以下 小さな王様)
    との違いや、引き継いだノウハウなど
- まとめ
- Q&A

背景

発表の背景

- Squirrelを使った、新たなゲームが完成
- 前作での利点、課題はどうなったか
- 新たな工夫について
- Squirrelをもっと多くの方に！

「光と闇の姫君」の立ち上がり

- 前作「小さな王様」のシステムで新作を
- 前作よりさらに小規模で・・・プログラマー２人
- 前作からゲーム内容を変える・・・魔王側、詳細は？

Squirrelを使った試行錯誤、効率的な開発

続編だけど、ゲームは違う！

- 「光と闇の姫君と世界征服の塔
  ファイナルファンタジー・クリスタルクロニクル」は、
  どんなゲームか？

- 「小さな王様と約束の国
  ファイナルファンタジー・クリスタルクロニクル」と、
  具体的にどう違ったのか？

簡単にゲームの内容をご紹介！

国づくりRPG

小さな王様と約束の国

ファイナルファンタジー・クリスタルクロニクル

2008/03/25 発売

世界征服RPG

光と闇の姫君と世界征服の塔

ファイナルファンタジー・クリスタルクロニクル

2009/06/30 発売

Squirrel をまだ知らないという方のために

ここで簡単にご紹介！

Squirrel

Squirrel概要

- 作者　AlbertoDemichelis さん
- ライセンス　zlib/libpng license
- 拡張子は.nut
- C/C++ライク
- オブジェクト指向
- 組み込みが容易
- ６０００行ほどのＣ＋＋コードで書かれている
- 型は動的型
- 例外処理

Squirrel概要

- Unicode 対応
- 64bit 対応
- alloc / free などが簡単に置き換え出来る様になっている
- Squirrel から C++、C++ から Squirrel の呼び出しも可能
- 読み方は、、、、

Luaとの比較

- 参照カウンタを使ったGC（ガベージコレクション）
- クラスの正式サポート、継承も可能
- Integer と Float の区別がある
- 配列とテーブルが別扱い
- ビット演算サポート

- 関数の戻り値は1つまで

ゲーム実装を考慮された設計になっている！

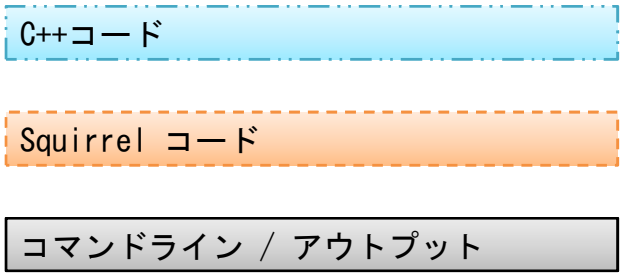
使用事例

- 小さな王様と約束の国 ファイナルファンタジー・クリスタルクロニクル
  Final Fantasy Crystal Chronicles: My Life as a King
  [国づくりRPG / Wii(WiiWare)]
- X-Blades [アクション]
- その他ゲーム
- オープンソースや、フリーソフトなど
  - SquirrelShellというplatformに依存しないshellも

- 光と闇の姫君と世界征服の塔ファイナルファンタジー・クリスタルクロニクル
  Final Fantasy Crystal Chronicles: My Life as a Darklord
  [世界征服RPG / Wii(WiiWare)]

コメント

- 行コメント
  //
- グループコメント
  /*～*/



型

- null
- 10 //Integer
- 1.0 //Float
- “hoge” //String
- Array, Table, Instance,….

```
local i = 10;
local f = 0.1;
local s = "hoge";
local a = [];
local t = {};
```

※1.0f と書いてしまうとエラーになるので注意

変数

- ローカル

```
local i = 0;
```

```
foo <- 20; //global
local foo = 10; //local

print("foo=" + foo + "¥n");
print("::foo=" + ::foo + "¥n");
```

- グローバル

```
i = 123;    // error
::i = 123;  // error
```

```
i <- 123;   //ok
```

```
::i <- 123; //ok
```

```
foo=10
::foo=20
```

キャスト

- .toxxx() で型変換できる

```
local i = 10
local f = 0.1
local sf = "1.0";
local s = "hoge";
local a = [];
local t = {};

class Foo {}
local ci = Foo();

print( "i=" + typeof( i.tofloat() ) +"¥n");
print( "f=" + typeof( f.tointeger() ) +"¥n");

print( "i=" + i.tofloat() +"¥n");
print( "f=" + f.tointeger() +"¥n");
print( "sf=" + sf.tofloat() +"¥n");
//error print( "s=" + s.tofloat() +"¥n");
print( "Foo=" + Foo.tostring() +"¥n");
print( "ci=" + ci.tostring() +"¥n");
print( "a=" + a.tostring() +"¥n");
print( "t=" + t.tostring() +"¥n");
```

```
i=float
f=integer
i=10
f=0
sf=1
Foo=(class : 0x003AF380)
ci=(instance : 0x003AF440)
a=(array : 0x003AF320)
t=(table : 0x003AF4A0)
```

関数

- 関数

```
function testFunc()
{
}
```

- 引数に型が無いのでコメントを書きましょう
- 戻り値は1つだけ

```
function fooFunc( bar, hoge )
{
    return bar + hoge;
}
```

※return を返さない関数の、戻り値を受け取ると null が入ります

制御文

- if, for, white, do~while, …

```
if( exp ){

}
else{

}
```

```
for( local i=0; i<10; i++ ){

}
```

```
while( exp ){

}
```

```
do{

} while( exp );
```

- switch

```
switch( exp ){
    case 1:
        break;
    case "foo":
        break;
    default:
        break;
}
```

※default は一番下に書かないとエラーになります
※case に同じ値があってもエラーが出ません

```
switch( exp ){
    default:  //error
        break;
    case 1:
        break;
}
```

```
switch( exp ){
    case 1:
        break;
    case 1:  //not error
        break;
    case "foo":
        break;
}
```

Array

- 配列、idx は 0～
- 動的に追加、削除が可能

```
local idx = 0;
local val = 123;

local fooArray = array(3); // null が入った３件の配列ができる

fooArray = array(0);    // 0件の配列を生成
fooArray = [];          // 0件の配列を生成
fooArray = [ 1, 2, 10, 20 ];        // 初期化付きで、配列を生成
fooArray = [ 1, 2, Foo(), { i=0 } ]; // 初期化付きで、配列を生成
fooArray[idx] = 0; // idx 番目の配列にアクセス
fooArray.append( val ); //後ろに val を追加する
fooArray.remove( idx );//idx 番目の配列を削除
```

Table

- キーと値のセット
- 動的に追加、削除が可能
- テーブルは Squirrel の根幹といってもよい機能

```
local fooTable = { a=1, b=2 };
fooTable.hoge <- 1;
fooTable[0] <- "two";
fooTable["1"] <- 3;
fooTable.rawset( "bar", 0.1 );

print( fooTable.hoge + "¥n" );    //ok
print( fooTable.bar + "¥n" );    //ok

print( fooTable[0] + "¥n" );    //ok
print( fooTable["1"] + "¥n" );  //ok

print( fooTable["0"] );//error
print( fooTable[1] );  //error

foreach( key, val in fooTable ){
    print( key +"="+ val +"¥n" );
}
```

```
a=1
b=2
1=3
hoge=1
bar=0.1
0=two
```

foreach

- foreach を使うと配列とテーブルを回すのに便利
- array

```
foreach( i, val in fooArray ){

}
```

```
val = fooArray[i]
```

- table

```
foreach( key, val in fooTable ){

}
```

```
val = fooTable[key]
```

Class

- クラスが使えます！
- 継承もできます
- コンストラクタはありますが、デストラクタはありません
- メンバ変数、関数
- static なメンバ変数、関数

```
class Foo
{
    constructor()
    {
    }

    function check()
    {
        print("foo!¥n");
    }
}

class Bar extends Foo
{
    static _baseBar = ::Foo;

    constructor()
    {
        _baseBar.constructor();
    }

    function check()
    {
        print("bar!¥n");
    }
}

Local bar = Bar();
```

例外

- スクリプト中のエラーを捕まえることができる

```
try
{
    i++;    // error
}
catch( str )
{
    print( str );

    // 後始末処理
}
```

```
the index 'i' does not exist
```

- 例外を処理オブジェクト単位でうまく使うことで、エラー発生時の時の再読み込み＆再実行を適切な範囲で行うこともできる
- throwで自前のオブジェクトを投げることもできる

便利なルートテーブル

- グローバルのものは、getroottable()を使ってroottable を探せば見つかる!!

- 関数や、クラス、定数が、定義済みかどうかもここを調べるとわかります。

```
foreach( key, val in getroottable() ){
    print("key=" + key + ", val=" + val + "¥n");
}
```

```
key=sqrt, val=(function : 0x003AD440)
key=stdin, val=(instance : 0x003ACF68)
key=print, val=(function : 0x003AAA58)
key=date, val=(function : 0x003AD2B0)
key=casti2f, val=(function : 0x003AC0A8)
key=getstackinfos, val=(function : 0x003AA520)
key=_intsize_, val=4
key=writeclosuretofile, val=(function : 0x003AC928)
key=collectgarbage, val=(function : 0x003AB080)
key=stdout, val=(instance : 0x003AC9F0)
key=setroottable, val=(function : 0x003AA700)
key=seterrorhandler, val=(function : 0x003AA388)
key=assert, val=(function : 0x003AA9D8)
key=atan2, val=(function : 0x003ADD58)
key=stderr, val=(instance : 0x003ACFF0)
key=_charsize_, val=1
key=asin, val=(function : 0x003AD5F0)
key=atan, val=(function : 0x003ADCC8)
key=setdebughook, val=(function : 0x003AA410)
key=ceil, val=(function : 0x003ADF18)
key=cos, val=(function : 0x003AD560)
key=log10, val=(function : 0x003ADBA8)
key=RAND_MAX, val=32767
…
```

その他の機能

- thread
- instanceof
- typeof
- clone
- generator
- などなど、、、

初期化処理、終了処理

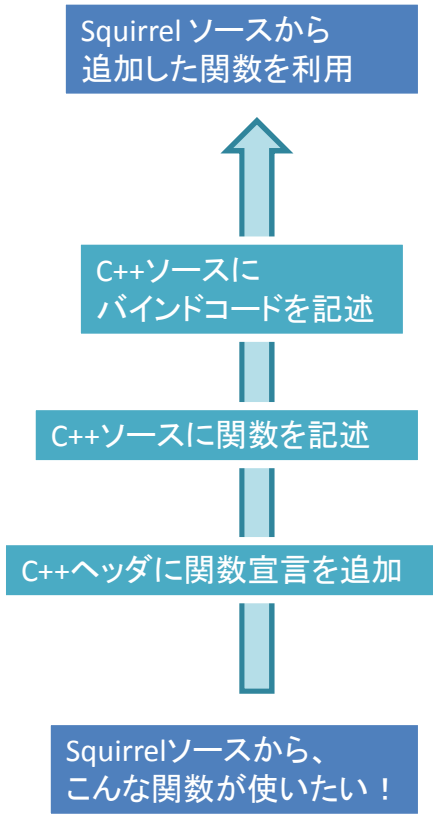
```
#include <squirrel.h>
#include <sqstdio.h>
#include <sqstdaux.h>
int main(int argc, char* argv[]) {
            // スタックサイズ1024でsquirrelVMオープン
            HSQUIRRELVM v = sq_open(1024);
            // print関数をセット
            sq_setprintfunc(v, myPrintFunc);
            // rootテーブルの登録
            sq_pushroottable(v);
            // 各ライブラリの登録
            sqstd_register_bloblib(v);
            sqstd_register_iolib(v);
            sqstd_register_systemlib(v);
            sqstd_register_mathlib(v);
            sqstd_register_stringlib(v);
            // エラーハンドラーを初期化
            sqstd_seterrorhandlers(v);

            // ゲームの処理
            ...

            // VMのクローズ
            sq_close(v);
            return 0;
}
```

```
void myPrintFunc(HSQUIRRELVM v, const SQChar* s, ...) {
      va_list arglist;
      va_start(arglist, s);
      vprintf(s, arglist);
      va_end(arglist);
}
```

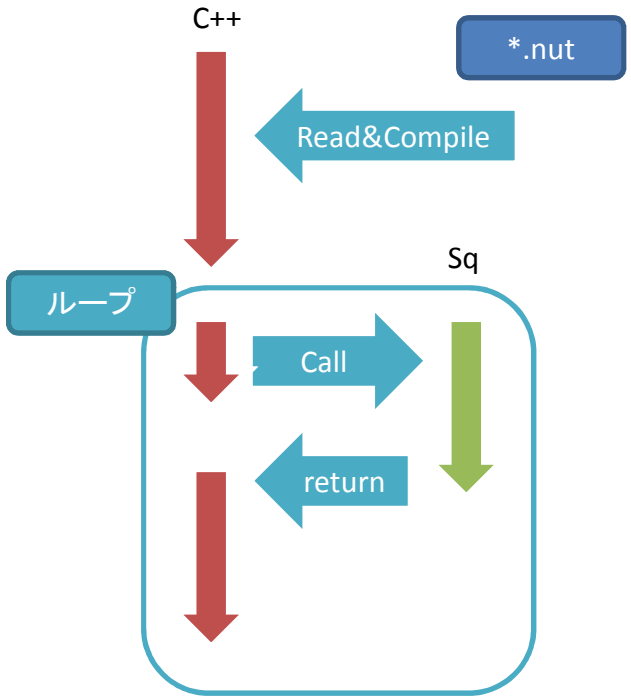
バインド

- Squirrel から C++ コードを呼び出すには、bind と呼ばれる作業が必要
- バインドは単純作業な上に、結構面倒くさい

- 引き続き、SqPlus を使用
- Sqrad, sqBind など新しいバインダも登場



実行の流れ



スクリプトのコンパイルと実行

```
// メモリに読み込んだスクリプトのコンパイルと実行
void run_script(HSQUIRRELVM v, char* cmd, char* fileName) {
    sq_compilebuffer(v, cmd, (int)strlen(cmd)*(int)sizeof(SQChar), fileName, true);
    sq_pushroottable(v);
    sq_call(v, 1, true, false);
}
```

```
/// @file test.nut

function foo() {
    print("foo called¥n");
}

// fooをすぐに実行
foo();

//[EOF]
```

```
foo called
```

- ゲーム開発においては、リソースの読み込み方法は様々
  コンパイルには、読み込みと分離されたsq_compile_bufferが便利

参考URL

- 公式サイト

http://www.squirrel-lang.org/

- 公式wiki

http://wiki.squirrel-lang.org/

- 日本語の非公式サイト
  - マニュアルの和訳等、検索したら出てきます

チュートリアル的なURL

- 公式wikiから
  - 組み込みの基本

  http://wiki.squirrel-lang.org/default.aspx/SquirrelWiki/EmbeddingGettingStarted.html
  - C関数の呼び出し

  http://wiki.squirrel-lang.org/default.aspx/SquirrelWiki/SquirrelCallToCpp.html

- 公式のOnline Documentから
  - squirrelの組み込み

  http://squirrel-lang.org/doc/squirrel2.html#d0e3487
  - C関数の登録

  http://squirrel-lang.org/doc/squirrel2.html#d0e3768
  - スクリプトファイルのコンパイル

  http://squirrel-lang.org/doc/squirrel2.html#d0e3719
  - squirrel関数の呼び出し

  http://squirrel-lang.org/doc/squirrel2.html#d0e3754

光と闇の姫君と世界征服の塔
ファイナルファンタジー・クリスタルクロニクル
の開発事例

「光と闇の姫君～」を取り巻く環境

- プログラマー２人
- 新作を立ち上げる
- 目に見えるリアルタイムバトル

- 前作システムの活用
- ゲームスクリプト部分の再設計、実装

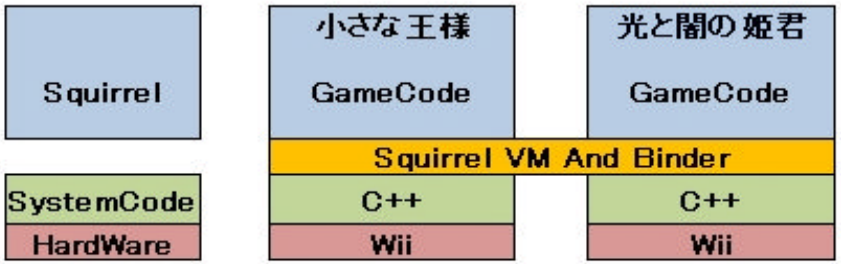
「光と闇の姫君～」の実装方針

- Squirrelをゲーム実装にフル活用する
  - 前作「小さな王様～」システムの踏襲
    - プログラマーの経験
    - Squirrelでのゲーム実装により、実行中の試行錯誤が容易
- 開発効率を上げる仕組みを強化する
- 見通しのよい設計で問題を未然に防ぐ

システム概要

- 小さな王様のシステムを原則そのまま利用！

- システム部分がC++、ゲーム部分が Squirrel で書かれている
- メインループも Squirrel 側
- ゲームデータにも Squirrel を使用

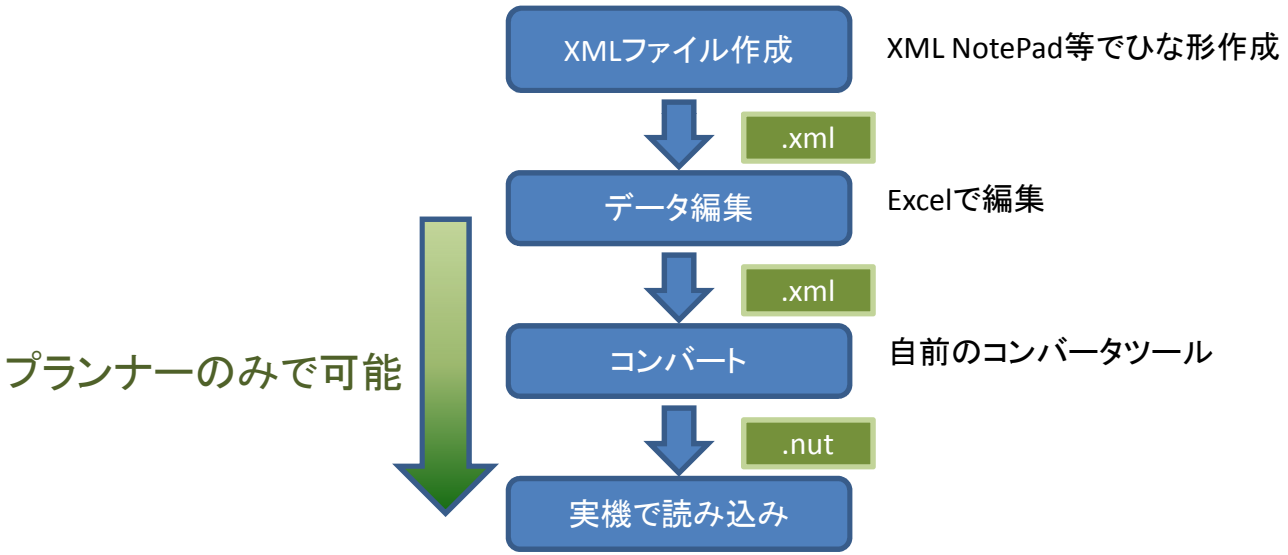


仕組みの強化

- try catchでのランタイムエラーハンドリングと復帰の強化
- ゲーム起動中における、データとコードのリロードの強化
- クロスコンパイルによるゲーム起動時間短縮

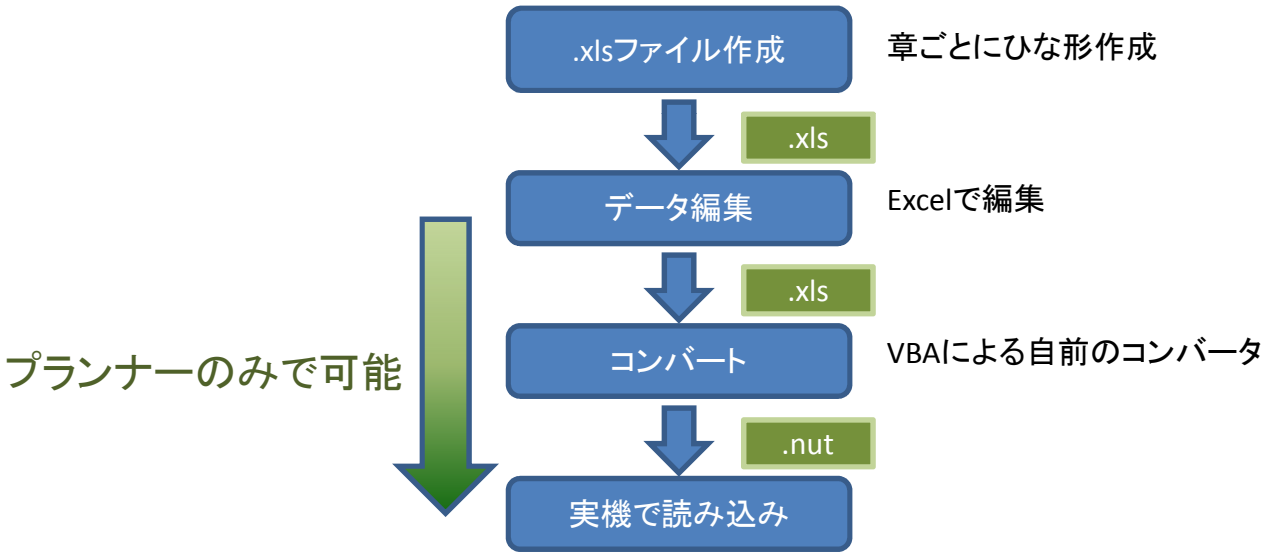
ゲームデータの作成と利用

- .nutファイルに、配列やテーブルとして書き出し、直に利用した



イベントデータの作成と利用

- Excelシートに打ち込んだイベントコマンド列を VBAで直接 .nutに変換した



データのリロード

- ゲームを起動したままデータを調整したい！

ゲームデータを.nutファイル化してあれば、
該当ファイルの読み込みとコンパイルで
以前のデータを上書き更新できる

デバッグメニューに仕込んでおく

コードのリロード

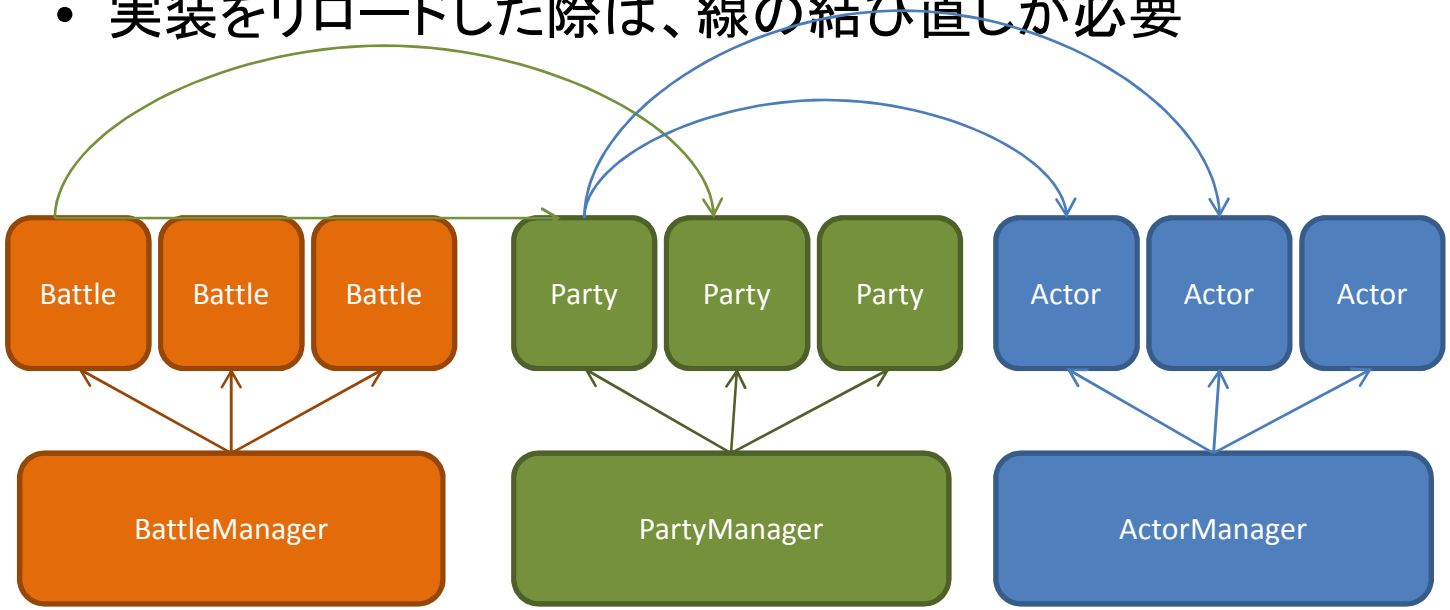
- ゲームを起動したままコードを更新したい！

- class定義の更新だけなら、データのリロードと同様（.nutファイル再読み込みとコンパイル）
- ただし、多くの場合、以前のinstanceが存在しているため、即座に反映させるには工夫が必要

新しいinstanceへの置き換え、参照の更新等

「光と闇の姫君～」のバトルまわりの構造

- 各マネージャがインスタンスを管理する
- 実装をリロードした際は、線の結び直しが必要



※適切なクラス分けとしっかりとした管理が前提　：　見通しのよい設計

デバッグメニューのリロード

- ゲームを起動したまま、デバッグ機能を追加できるので、大変便利！

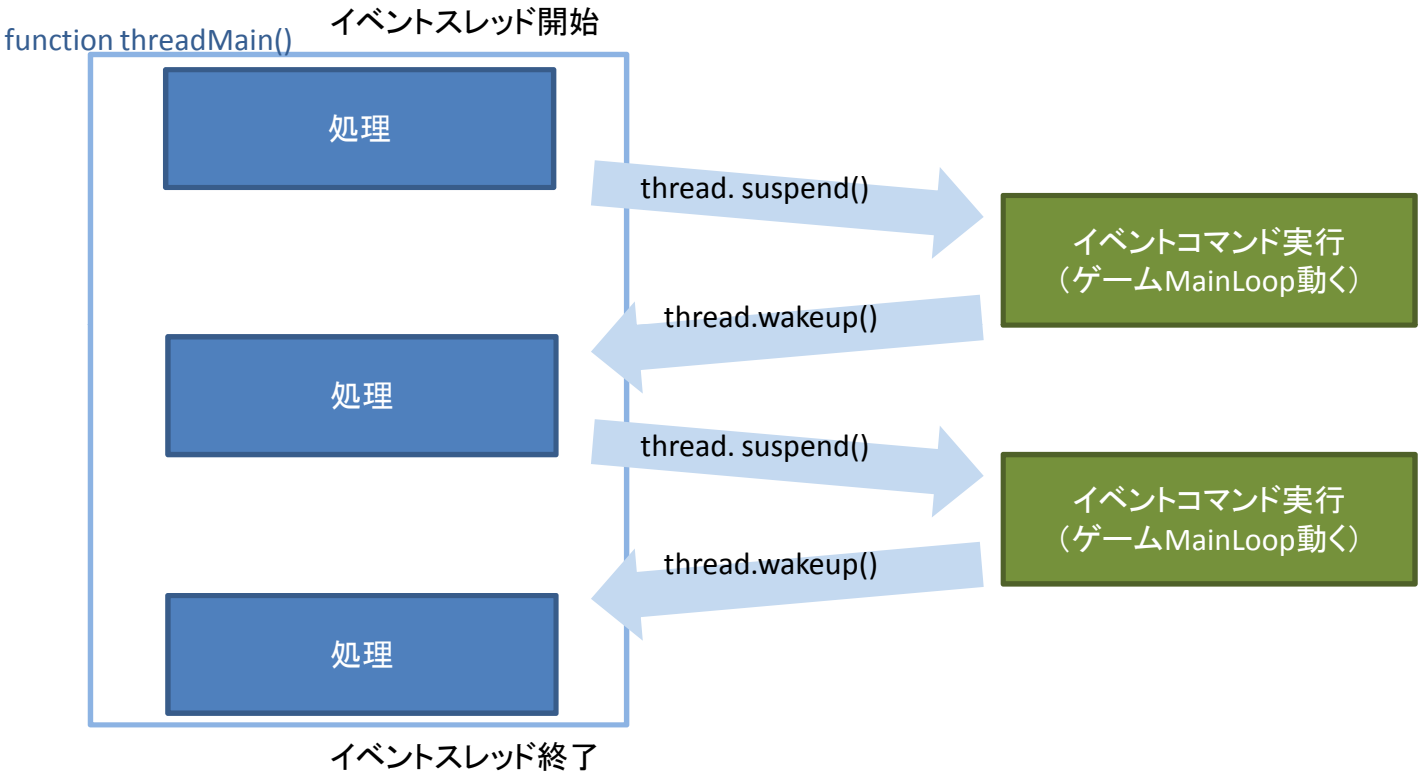
クロスコンパイルとバイナリコード実行

- クロスコンパイル
  - sqobject.cppのSafeWrite関数に渡すデータをEndian変換するだけ
    （呼び出し元はsqstd_writeclosuretofile）

- バイナリコード実行
  - コンパイル済みバイナリコードのde-serializeと実行
    sq_readclosure
    sq_call

  - 比較：スクリプトファイルのコンパイルと実行
    sq_compilebuffer
    sq_call

スレッドを使ったイベント

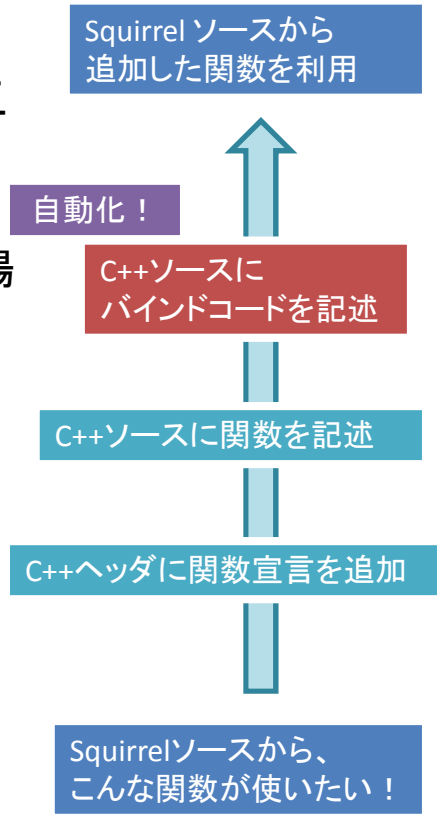


実装テクニックとノウハウ

ツールを作ってバインドコードを自動生成

- 自動化のメリット
  - C++ヘッダにある関数はSquirrel 側で使える様になっているか？引数は？などが、C++のヘッダーファイルだけ見ればわかる様になる
  - バインドするメンバを増やしたり減らしたりする場合は、C++ヘッダを書き換えて、ツールを実行してコンパイルするだけでいい
- ツールを作る上での工夫
  - ツールがファイルを更新するのは、自動生成内容が変わったときだけ

C++コードの不要なリコンパイルを避ける

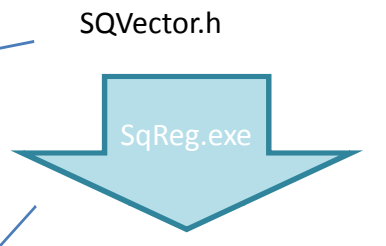


ツールを作ってバインドコードを自動生成

```
class SQVector
{
public:
    static void BindSquirrel();

    float x, y, z;
    SQREG_CVAR( x,"x");
    SQREG_CVAR( y,"y");
    SQREG_CVAR( z,"z");

    SQVector();
    SQVector( float x, float y, float z );
    void Set( float x, float y, float z );
    SQREG_CFUNC( Set, "Set" );
}

SQREG_CLASS( SQVector, "SQVector" );
```

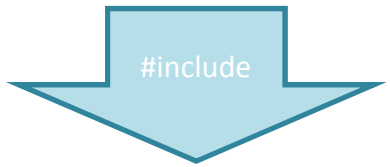
ヘッダファイルに
キーワードを記述！

```
[SqReg.rb] / version 1.2.0
 command: SqReg.rb ../SQVector.h
 read : ../SQVector.h
 write: SQVector.sqreg <- SQVector.sqreg.tmp
```



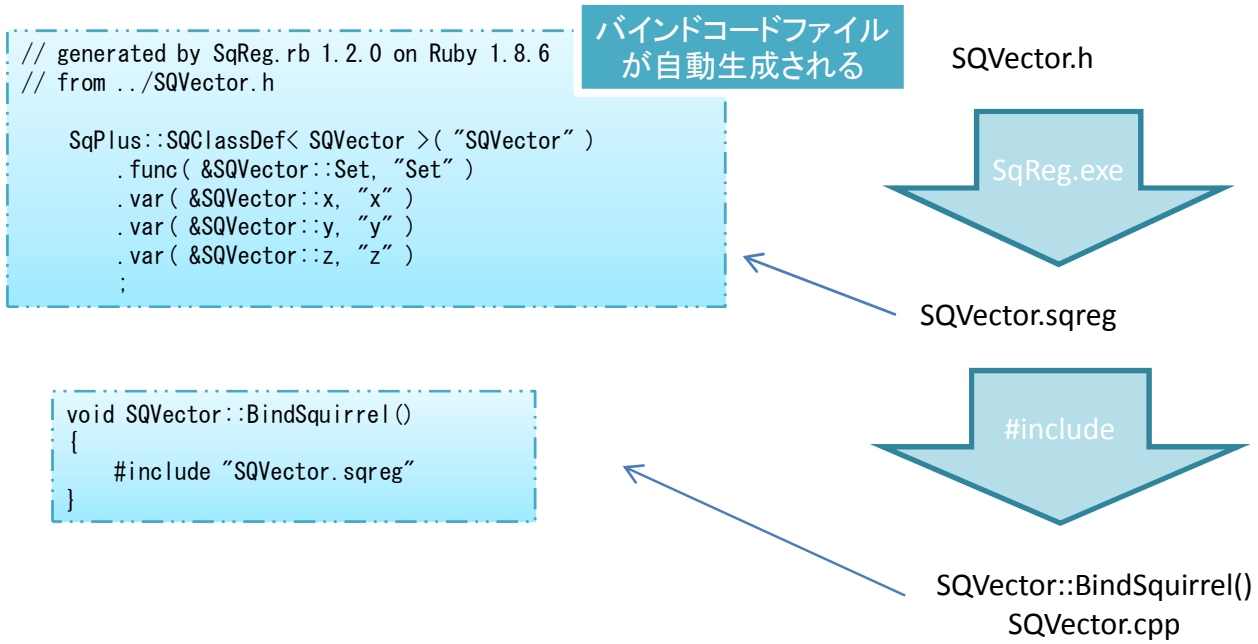
SQVector.h

SQVector.sqreg



SQVector::BindSquirrel()
SQVector.cpp

ツールを作ってバインドコードを自動生成



スタックダンプ

- Squirrel 中で起きたエラーについては、スタックダンプを見ることで、ソースのどこで止まったかが簡単にわかります。

```
AN ERROR HAS OCCURED [the index 'vec' does not exist]

CALLSTACK
*FUNCTION [SetPos()] Test/GameObject.nut line [321]
*FUNCTION [InitMapObject()] Test/GameManager.nut line [1253]
*FUNCTION [Update()] Test/GameManager.nut line [300]
*FUNCTION [Update()] Test/Test.nut line [61]
*FUNCTION [Update()] RootMenu.nut line [134]
*FUNCTION [ScriptUpdate()] ScriptMain.nut line [10]

LOCALS
[y] 0
[x] 150
[this] INSTANCE
[@ITERATOR@] 1
[objectData] TABLE
[i] 0
[object] INSTANCE
[this] INSTANCE
[this] INSTANCE
[this] INSTANCE
[this] INSTANCE
[this] TABLE
```

設計ノウハウ

- Squirrel 中のエラーは、スタックダンプのおかげで、原因の特定が非常に簡単です。
- C++で止まってしまうと、スタックダンプが出ないので、これらの恩恵を受けられません。

- C++コード中で止まっても、スタックダンプが出る様に！
  - C++で生成したクラスのポインタをそのまま Squirrel に見せる場合、try～catch を使って、Squirrelのスタックダンプを呼び出すか、呼びだせる様にしておくのがお勧めです。
    VMの中でアドレス例外などが起きると、スタックダンプが出ない場合があり、C++デバッガからのトレースが面倒です。
  - ポインターをそのままSquirrelに出さず、アドレスを id に置き換えて管理する方法も有効です。この場合は、C++側の関数で id が適正かどうか事前にチェックして、問題があればスタックダンプを出すことができます

C++コード中で止まった！

- C++側でアドレス例外などで落ちた場合、
  Squirrel のスタックダンプが出ずに停止してしまう場合があります、
  そんな場合には下記の様な関数を1つ用意しておくと便利です

```
void
ScriptManager::PrintCallStack()
{
    sqstd_printcallstack( SquirrelVM::GetVMPtr() );
}
```

- IDEでPC(プログラムカウンタ)をここに飛ばせば、
  いつでも Squirrel のスタックダンプを print させることができます

- ASSERT や、割り込み例外などの処理に足しておくと、
  C++側でおきたエラーが、どのスクリプトから呼びだされたものかが簡単に解ります

メモリー

- メモリーブロック数に注意！！
  100B以下程度の細かいブロックが沢山できる
  - Alloc/Free の負荷増大と、メモリ浪費の原因に
  - この状態で頻繁な realloc の発生は、かなりのボトルネックになる
- データ構造に Table を用いるのは便利、でも Array よりも メモリーを消費する
- GCの実行タイミングとデフラグメントにも注意

Squirrel には専用のメモリ空間を！！

GCの実行タイミング

- GCの実行はフレームレートに影響する
  - 影響しにくい場所で
  - 画面の切り替えや、フェードのタイミングで
- リソースの入れ替え前
  - デフラグメントを減らす
- ヒープメモリーが無くなったとき
  - ほおっておくとゴミが貯まっていき、
    いずれメモリーが無くなる場合も
- GCを任意のタイミングで実行できる様に、
  デバッグメニュー等にボタンを用意しておくと便利

気をつけるべき重い処理

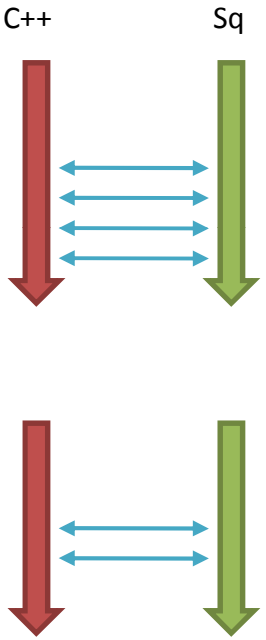
- Squirrel と C++ の頻繁な往復
- 文字列の加工
  - realloc が頻発
  - フレームレートに影響するほど
- コピーの発生する引数や戻り値の受け渡し
  - String
  - 同様にヒープ負荷が影響
- GC
  - 毎フレーム実行するには重い

往復を減らす

- 目的に特化したI/Fを用意して、スクリプトとC++の往復を減らす

```
local player = SQPlayer();
local vec = player.GetPos(); // retval SQVector
vec.y += speed;
player.SetPos( vec );
```

```
local player = SQPlayer();
player.AddPosY( speed ); // C++側で posVec.y += speed される
```



ランタイムエラー

エラーの多くがランタイムで起こります！

- 変数やメンバが無い
- typo
- シンタックスエラー
- 型が違う
- などなど、、、

ランタイムエラーを減らすには

- ネーミングルールを統一する
- タイプしにくい名前は避ける
- 補完機能などを利用する

- コマンドラインツールや、IDEを使ってシンタックスを事前にチェック
  - 動的シンボルのチェックはできない、、、

- データラベルなどは、名前を規則化して、ツールなどでチェック
  - key 参照などで、ラベルを動的に作っている場合もチェックできない

エラー回復の流れ

- もしスクリプトエラーが発生した場合は、スクリプトの呼び出しを一時中止します。
  - 空回りでもいいですし、埋め込みのブレークポイントなどでも構いません
- その間に、問題のソースファイル(.nut)を修正
- タイムスタンプが更新されたスクリプトファイルを読み込み直して、コンパイル、処理を復帰させます。

- 前述の通り、class インスタンスの様な場合は、これだけでは回復できません、工夫が必要です。

デバッガ

- デバッガ、あります！
  - Eclipse / SQDev
  - Squirrel Visual Studio 2008 Integration

- でも、、、、
  - プロトコルをラップして、Wii でも使える様に書き換えることができた。
  - 開発ハードウェアの通信速度上の問題で、実用できなかった。
  - デバッガが無くても、優秀なスタックダンプと print デバッグだけで十分なんとかなる！

debugHookの利用

- enabledebuginfo(true);
  した状態でコンパイルしてあれば、ライン毎のコールバックを設定可能

つまり、簡単なデバッガを作成できる

巻末資料

まとめ

今回の試み

- 前作システムで、まったく違ったゲーム作成
  - Squirrelスクリプトをごっそり入れ替え
- 更なる小規模開発
  - Squirrel利用のメリットを最大限に活用
  - 効率化の仕組みを強化
- 再び、Squirrelをフル活用

今回確認できた点

- 前回に続き、Squirrelスクリプトをフルに使った実装でゲームを完成させることができた
  + Squirrelを使った試行錯誤の有用性を確認
- スクリプト部分のみを入れ替え、新しいゲームを作成できた
- 前回の工夫をより強化して、レスポンスの良い開発環境が得られた

前回発表時の課題について

- ランタイムエラー
  - try catchの強化、スクリプトリロードの強化で対応
- メモリ使用量
  - 見通しのよい設計で問題を回避、リークも防いだ
    （ただし仕様は前作より小さい）
- 完全なメモリマップの掌握が困難
  - 特に変わらず

「光と闇の姫君～」でも苦労した点

- try,catchが難しい部分での実行時エラーはゲームの再起動につながった
- 実用可能な速度でデバッガ(SQDev)が動かないため、自前の変数表示デバッグで行っていた
- メモリ使用量の変化を追いづらい

Squirrel利用の利点

- ✓ データとコードの両面で、素早いリトライが可能
- ✓ 強力なコールスタックにより問題の把握が容易
- ✓ IDEが無くても制作や、デバッグが可能
- ✓ 難解なメモリー破壊の様なバグが起きにくい
- ✓ ゲームとの相性がよい

Squirrel利用の注意点

- ✓メモリーには注意
  - ✓メモリーマップの掌握が困難
  - ✓ガベージコレクタのため使用量の把握と管理が難しい
- ✓デバッグのコスト増
  - ✓ランタイムエラーによる製品テストのコスト増に注意
  - ✓ランタイムエラーを減らす工夫を
  - ✓エラー時にハングしないように工夫を
- ✓コメントを書きましょう
  - ✓引数や、戻り値に型が無いため、コメントを書きましょう

今後

今後も是非使いたいです！

新作の立ち上げには特に！！！

Q&A

Thank You!

光と闇の姫君と世界征服の塔 ファイナルファンタジー・クリスタルクロニクル™
小さな王様と約束の国 ファイナルファンタジー・クリスタルクロニクル®

巻末資料

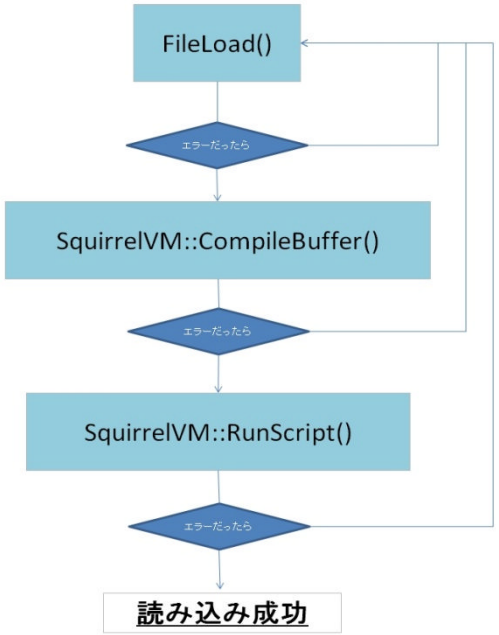
VMスタックサイズの調整

- sqvm.cpp _stack.resize() にブレークを張って、何度も通る用ならVMのスタックはあらかじめ大き目に、頻繁な realloc を避けることができます

SquirrelVM.cpp

```
void SquirrelVM::Init()
{
//          _VM = sq_open(1024);
            _VM = sq_open(8*1024);
        ...
```

import の実装



- 比較的簡単に実装可能
- これがあるとソースを分けて管理できる
- import したファイルをプログラムで管理し、タイムスタンプをチェックしながら再読み込み出来る様にしておくと、エラーの回復がし易くなります

```
SQImport( “Ability.nut” );
SQImport( “Eenmy.nut” );

function initialize()
{
    local enemy = Eenmy();
}
```

固定サイズのスモールヒープ

- Squirrel 用のメモリにおいて、ブロック数の増大によりパフォーマンスが問題になっていないか、動作中のヒープから100B以下程度のメモリーブロックの数を調べる

- もし沢山できていれば、それらにメモリーから固定サイズのスモールヒープとして専用領域を適切なだけ割り当てることで改善が期待できます

- 固定ヒープのメリット
  - ブロック数増大によるヒープ処理オーバーヘッドの軽減
  - ブロック数増大によるヒープブロックサイズのメモリ消費軽減

77

固定サイズのスモールヒープ

```
struct SmallBlock
{
    unsigned char buffer[SMALL_BLOCK_SIZE];
    SmallBlock*   prev;
    SmallBlock*   next;
};
```

- heapSmall は固定サイズに特化した専用のメモリ空間をもったヒープ

```
void *sq_vm_malloc(SQUnsignedInteger size){
    void* mem;
    if( size <= SMALL_BLOCK_SIZE ){
        mem = g_heapSmall.alloc();
    }
    if( mem == NULL ){
        mem = g_heapSquirrel.allocAlign(size, ALIGN);
    }
    return mem;
}

void sq_vm_free(void *p, SQUnsignedInteger size){

    if( g_heapSmall.isValid( p ) ){
        g_heapSmall.free(p);
    }
    else{
        g_heapSquirrel.free(p);
    }
}
```

定義済みか調べる

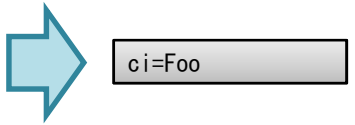
- roottable を使って、#ifdef の様なことが可能
- ::hogehoge が定義されているか？

```
if( "hogehoge" in getroottable() ){
    print("hogehoge 発見！値は= " + ::hogehoge + "¥n");
}
```

roottable を使って class の名前を取得する

- roottable を引くことで、クラスなどの名前も取得できます

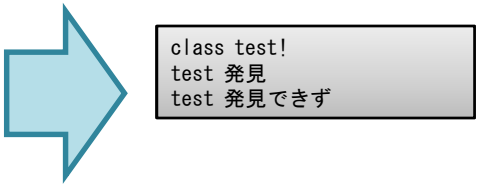
```
// isntance か getclass の値を入れたら、クラス名が返る
function getClassName( classdef )
{
    if( typeof(classdef) != "class" ){
        return "*NotClass*";
    }

    foreach( name, item in getroottable() ){
        if( item == classdef ){
            return name;
        }
    }
    return "*UnknownClass*";
}

class Foo {}
local ci = Foo(); print( getClassName( ci.getclass() ) + "¥n");
print( "ci=" + getClassName( ci.getclass() ) + "¥n");
```



定義済みの class を解放する

```
class test
{
    constructor()
    {
        print("class test!¥n");
    }
}

local foo = test();

if( "test" in getroottable() ){
    print("test 発見¥n");
}
else{
    print("test 発見できず¥n");
}

delete getroottable()["test"]

if( "test" in getroottable() ){
    print("test 発見¥n");
}
else{
    print("test 発見できず¥n");
}

//local bar = test();   //error
```

- メモリーを目的に開放する場合は、必要に応じてGCも実行してください。



新しいバインダ: sqBind

- MIT License
- シンプルなバインダ。バインディングした際のコードが軽量
- クラス、メソッド、staticメソッド、メンバ変数、グローバル関数、enum、nativeなタイプをサポート

```
class CTest {
private:
            int m_val;
public:
            CTest() { m_val = 2; }
            int getVal() { return m_val; }
};
CTest g_cTest;
CTest& getCTestInstance() {
            return g_cTest;
}

#include "sqbind.h"
...
HSQUIRRELVM v = sq_open(1024);
sqbind_function( v, "getCTestInst", getCTestInstance );

SqBind<CTest>::init(v,_SC("CTest"));
sqbind_method( v, "getVal", &CTest::getVal );
```

新しいバインダ: Sqrat

- zlib/libpng license
- ヘッダのみで構成されたシンプルなバインダ
- staticなメンバも含め、メンバ変数、メンバ関数、グローバル関数、変数、定数やenumをサポート

```
// 前のスライドのCTestクラスが定義されているとする

#include "sqrat.h"

...

using namespace Sqrat;

HSQUIRRELVM v = sq_open(1024);
Table myTable(v);
myTable.Func("getCTestInst", &getCTestInstance);
RootTable(v).Bind("MyTable", myTable);

Class<CTest> cTestClass(v);
cTestClass.Func("getVal", &CTest::getVal);
RootTable(v).Bind("CTest", cTestClass);
```

VMの初期化:SqPlus

```
    // Squirrlの初期化
    SquirrelVM::Init();

#ifdef  SQUIRREL_DEBUG
    sq_enabledebuginfo( SquirrelVM::GetVMPtr(), true );

    // エラーの時にstack情報を表示させる
    sq_notifyallexceptions( SquirrelVM::GetVMPtr(), true );
#endif

    // printf 関数
    sq_setprintfunc( SquirrelVM::GetVMPtr(), &Printf );

    // コンパイルエラー関数
    sq_setcompilererrorhandler( SquirrelVM::GetVMPtr(), &CompilerErrorPrintf );

    m_rootVMPtr = SquirrelVM::GetVMPtr();
    m_rootTable = SquirrelVM::GetRootTable();

    ScriptGlobal::BindSquirrel();
    SQVector::BindSquirrel();
    SQCamera::BindSquirrel();
    ...
```

関数のバインド：SqPlus

- C++のシンボル名と、バインド名は違っていても動作する

```
class SQGlobal
{
public:
    static void FooFunc( int val, bool flag );
}
```

```
void BarFunc( SQVector* vec )
{
}
```

```
SqPlus::RegisterGlobal( SQGlobal::FooFunc, _T( "SQFoo") );
SqPlus::RegisterGlobal( BarFunc, _T( "SQBar") );
```

```
SQFoo( 1, true );

local vec = SQVector();
vec.x = 1.0;
SQBar( val );
```

クラスのバインド：SqPlus

```
namespace script
{
    class SQVector
    {
    public:
        float x, y, z;
        SQVector();
        void Set( float x, float y, float z );
    };

}   // namespace script

DECLARE_INSTANCE_TYPE_NAME( script::SQVector, SQVector );
```

```
using namespace script;

SqPlus::SQClassDef< SQVector >( "SQVector" )
    .func( &SQVector::Set, "Set" )
    .var( &SQVector::x, "x" )
    .var( &SQVector::y, "y" )
    .var( &SQVector::z, "z" )
    ;
```

```
local vec = SQVector();
vec.x = 0.0;
vec.y = 1.0;
vec.z = 0.0;

obj.SetPos( vec );
```

namespace などを使っていても大丈夫

クラスのバインド：SqPlus

- C++からバインドしたクラスをextendsして、Squirrel 上で派生することも可能

```
SqPlus::SQClassDef< SQMath >( " SQMath " )
    .staticFunc( &SQMath::DegToRad, "DegToRad" )
    ;
```

```
class Math extends SQMath
{
    static PI = 3.1415926535;
}
```

```
local rad = 2 * Math.PI;
Math.DegToRad( rad );
```

定数のバインド:SqPlus

- enum や マクロなどをバインドします。
- 変数も可能

```
class Camera
{
public:
    enum MODE
    {
        MODE_NORMAL
    };
};
```

```
SqPlus::BindConstant( static_cast<int>(Camera::MODE_NORMAL), "SQCAMERA_MODE_NORMAL" );
```

```
local camera = SQCamera();
camera.SetMode( ::SQCAMERA_MODE_NORMAL );
```

スクリプトの呼び出し：SqPlus

- 読み込んだスクリプトを C++ から呼び出します
- ここでは ScriptMain()
- C++からの呼び出しは、スタック操作がややこしいです

```
sq_pushroottable( SquirrelVM::GetVMPtr() );
sq_pushstring( SquirrelVM::GetVMPtr(), "ScriptMain", -1 );
sq_get( SquirrelVM::GetVMPtr(), -2 );
sq_pushroottable( SquirrelVM::GetVMPtr() );
SQRESULT result = sq_call( SquirrelVM::GetVMPtr(), 1, SQFalse, SQTrue );
sq_pop( SquirrelVM::GetVMPtr(), 2 );
```

C関数のsquirrelへの登録（バインド）

- C関数登録の例

```
SQInteger register_global_func(HSQUIRRELVM v,SQFUNCTION f,const char *fname)
{
    sq_pushroottable(v);
    sq_pushstring(v,fname,-1);
    sq_newclosure(v,f,0,0); // 新たに関数を作成
    sq_createslot(v,-3); // roottableに追加
    sq_pop(v,1); // roottableをpopしてスタックを元通りに
}
```

- 登録されるのは、以下のような関数

```
SQInteger testFunc0(HSQUIRRELVM v) {
    sq_pushinteger(v, 111);    // 適当な戻り値
    return 1;                  // 戻り値の数が1なので1
}

void testFunc1(HSQUIRRELVM v) {
    return 0;                  // 戻り値の数0
}
```

Squirrel関数の呼び出し

- Squirrel関数を呼び出す例

```
sq_pushroottable(v);
sq_pushstring(v, “foo”,-1); // fooという関数を呼びたい
sq_get(v,-2); // roottableから関数を取得
sq_pushroottable(v); // 環境(this)としてroottableをセット
sq_pushinteger(v, 1);  // 引き数セット
sq_pushfloat(v, 10.0); // 引き数セット
sq_pushstring(v, "test_string",-1); // 引き数セット
sq_call(v,4,SQFalse);  // 呼び出し
sq_pop(v,2); // roottableをpopしてスタックを元通りに
```

C++側から配列を得るサンプル

```
/// 配列を返す関数の例
SQInteger getArraySample(HSQUIRRELVM v) {
            // 引数の取得例
            SQInt32 flagTest;
            sq_getinteger(v, -1, &flagTest);

            // 配列の作成例
            int32 size = 16;        // ここでは適当なサイズ
            sq_newarray(v, size);   // 配列を作成してVMのスタックに積む
            for ( int32 i = 0; i < size; ++i ) {
                        // 配列に要素を追加する例
                        int32 val = i*2;        // ここでは適当な値
                        sq_pushinteger(v, i);   // index
                        sq_pushinteger(v, val);// 値を入れる
                        if(!SQ_SUCCEEDED(sq_rawset(v, -3))) {
                                    assert(0);
                        }
            }
            return 1;   // 返り値は１つ
}
```

clone

- オブジェクトを代入すると通常は参照が渡される。
- 配列やテーブルを明示的にコピーしたい場合は、clone を使う

```
local fooArray = [];

fooArray.append( 10 );

local barArray = clone( fooArray );// 複製

barArray.append( 20 );

foreach( i, val in barArray ){
    print( "barArray["+i+"]=" + val + "¥n");
}

foreach( i, val in fooArray ){
    print( "fooArray["+i+"]=" + val + "¥n");
}
```

```
barArray[0]=10
barArray[1]=20
fooArray[0]=10
```

※ただし、cloneは浅いコピー。再帰的に入れ子構造をコピーしてはくれない

メンバー変数について注意

- メンバー変数は、コンストラクタで初期化しないと static の様にふるまうので注意

```
class Foo
{
    _barArray = []; // static ?

    constructor()
    {
    }

    function push( val )
    {
        _barArray.append( val );
    }

    function print()
    {
        foreach( i, val in _barArray ){
            ::print("i=" + i + ", val=" + val + "¥n");
        }
    }
}
```

```
local foo = Foo();
local foo2 = Foo();

foo.push( 10 );
foo2.push( 20 );
foo.print();
```

```
i=0, val=10
i=1, val=20
```

※Integer, Float, Bool なら問題ないが、配列、テーブル、クラスinstance等ではNG

メンバー関数

- this や、:: を使ってスコープを区別することができます

```
class Foo
{
    constructor()
    {
    }

    function test()
    {
        this.print();
    }

    function print()
    {
        ::print("test()!¥n");
    }
}


local foo = Foo();
foo.test();
```

static メンバー

```
class Hoge
{
    static _staticIndex = 123;  // static 変数
    _var = null;

    constructor()
    {
        _var = 456;
    }

    function    check()
    {
        print("b "  + _staticIndex + "¥n" ); //ok
    }

    static function test()  // static 関数
    {
        print("c¥n");
//      print("_var=" + _var ); //error _var は static じゃない
    }
}
```

```
// static 変数の参照
print("a" +  Hoge._staticIndex + "¥n");

Hoge.test();    // static 関数の呼び出し

local hoge = Hoge();
hoge.check();
```

```
a 123
c
b 123
```

typeof

- 変数の型を調べます

```
local i = 10;
local f = 0.1;
local s = "hoge";
local a = [];
local t = {};

class Foo {}
local ci = Foo();

print( "i=" + typeof( i ) +"¥n");
print( "f=" + typeof( f ) +"¥n");
print( "s=" + typeof( s ) +"¥n");
print( "Foo=" + typeof( Foo ) +"¥n");
print( "ci=" + typeof( ci ) +"¥n");
print( "a=" + typeof( a ) +"¥n");
print( "t=" + typeof( t ) +"¥n");
```

```
i=integer
f=float
s=string
Foo=class
ci=instance
a=array
t=table
```

instanceof

- instance の正体を調べます

```
class Foo {}
class Bar extends Foo {}

local a = Bar();

if( a instanceof ::Bar ){
    print("b=Bar¥n");
}
if( a instanceof ::Foo ){
    print("a=Foo¥n");
}

if( a.getclass() == ::Foo ){
    print("a=Foo¥n");
}
if( a.getclass() == ::Bar ){
    print("a=Bar¥n");
}
```

```
a=Bar
a=Foo
a=Bar
```

C++から Squirrel のシンボルの値を取得：SqPlus

- グローバルなら、RootTable から簡単に取得できます

```
::TXT_ID_FOO <- 10056;
```

```
int textId =  SquirrelVM::GetRootTable().GetInt( "TXT_ID_FOO" );
```

ごく簡単な printf 実装例

```
function printf( msg, ... )
{
    local arg = array(7);
    local i = 0;
    while( i < 7 ){
        if( i < vargc ){
            arg[i] = vargv[i];
        }
        i += 1;
    }
    print( format( msg, arg[0], arg[1], arg[2], arg[3], arg[4], arg[5], arg[6] )  );
}

local i = 10;
local f = 0.1;
printf("%s, %d, %f, %s¥n", "str", i, f, "aaa"  );
```

```
str, 10, 0.100000, aaa
```

※これで、printf と書いてもエラーにならなくなります

スレッドを使ったイベント実装例

```
/// イベントコマンドクラス
class EventCommand
{
            /// 開始処理
            function start() {
            }

            /// 毎フレームの処理
            function proc() {
                        if ( 終了check ) {
                                    return true;            // コマンド終了
                        }
                        return false;           // コマンド続行
            }
}

/// スレッド（コルーチン）を持つイベントクラスの例
class ContextEvent
{
            _thread = null;         ///< 実行部分となるスレッド
            _currentCommand = null;///< 現在実行中のEventCommand

            /// コンストラクタ
            constructor() {
                        _thread = null;
                        _currentCommand = null;
            }
```

スレッドを使ったイベント実装例

```
/// イベント前処理          :オーバーライド用
function startMain() {
}
/// イベント実行処理        :オーバーライド用
function procMain() {
}
/// イベント後処理          :オーバーライド用
function endMain() {
}
/// イベントを開始する
function start() {
            startMain(); // 各イベントの前処理を呼ぶ
            _thread = ::newthread(procThread);      // 実行関数を指定してスレッド生成
            _thread.call(this);                     // 開始
            // 終わった？
            if ( _thread.getstatus() == "idle" ) {// "idle" or "running" or "suspended"
                        end();       // 後処理
            }
}
/// スレッド実行
function procThread(event) {
            try {
                        event.procMain();          // 各イベントの実行処理を呼ぶ
            }
            catch ( ex ) {
                        // 例外発生
                        // エラー内容をPrintしてイベント終了。ゲームは続行
                        //...
            }
            return;
}
```

スレッドを使ったイベント実装例

```
function execCommand(command) { /// イベントコマンドを実行してスレッド休止
        if ( command ) {
                _currentCommand = command;
                _currentCommand.start();
        }
        return suspend();                       // スレッドを待ち状態にする
}
function resume() {/// スレッド再開
        if ( _thread.getstatus() == "suspended" ) {// "idle" or "running" or "suspended"
                try {
                        _thread.wakeup();       // スレッド再開
                }
                catch ( ex ) {
                        // 例外発生
                        // エラー内容をPrintしてイベント終了。ゲームは続行
                        //...
                }
        }
        if ( _thread.getstatus() == "idle" ) { // "idle" or "running" or "suspended"
                end();          // 後処理
        }
}
/// イベントの毎フレームの更新
function proc(world) {
        if ( _currentCommand ) {
                if ( _currentCommand.proc() ) {
                        // コマンドは終了した
                        _currentCommand = null;
                        resume();       // スレッド再開
                }
        }
}
```

スレッドを使ったイベント実装例

```
                /// 後処理
                function end() {
                        endMain();      // 各イベントの後処理を呼ぶ
                        // イベントManagerの管理から外す
                        //...
                }
}

/// イベント実装例
class TestEvent extends ContextEvent
{
                /// イベント前処理              :オーバーライド
                function startMain() {
                        // 必要であれば実装
                }
                /// イベント実行処理            :オーバーライド
                function procMain() {
                        local command = null;

                        //...

                        // コマンドその１を生成、実行してスレッド休止
                        command = EventComamnd001();
                        command.setParam(1.0);
                        execCommand(command);

                        //...
                }
                /// イベント後処理              :オーバーライド
                function endMain() {
                        // 必要であれば実装
                }
}
```

debugHookを利用したコード例

```
::g_targetSrc <- null; ///< デバッグ用、ブレークポイント設定ソース
::g_targetLine <- -1; ///< デバッグ用、ブレークポイントLine

// debugHookをセットする
setdebughook(hook_func);

/// ブレークポイントでローカル変数の値を書き出すdebugHook関数の例
function hook_func(event_type, sourcefile, line, funcname) {
    if ( event_type == 'l' ) {    // 各行のコールバック
        if ( (line == ::g_targetLine) && (sourcefile == ::g_targetSrc) ) { // check file and line
            ::print("Break Point Hit !¥n");
            local stackInfos = getstackinfos(2);    // hook_funcの呼び出し元を調べる
            if ( stackInfos ) {
                ::print("func:" + stackInfos.func + "¥n");
                    foreach ( key, val in stackInfos.locals ) {
                    ::print(key + "=" + val + "¥n");    // 値をprint
                    if ( key == "this" ) {  // thisの内容はprintする
                        if ( typeof(val) == "instance" ) {
                            foreach ( key_class, val_class in val.getclass() ) { // class定義からメンバーを調べ書き出す
                                if ( typeof(val_class) != "function" ) {    // functionは除く
                                    ::print("    " + key_class + "=" + val[key_class] + "¥n");
                                }
                            }
                        }
                    }
                }
            }
            // ボタン入力待ちのLoop、あるいはC++側の（埋め込み）ブレークポイントを呼び出す
            //...
        }
    }
}
```