デバッグ制御システム

– コマンドベースの効率的なデバッグインターフェース作成 –

2014 年 2 月 18 日 初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 18 日	板垣 衛	（初稿）

■ 目次

■ 概略

デバッグ機能の追加実装や、デバッグ情報の扱いを簡単にするためのシステムを設計する。

■ 目的

本書は、デバッグ機能およびそれを実行するインターフェースを簡単に追加実装できるようシステムを構築することを目的とする。

また、インターフェースの一つとして、より便利な PC 上のデバッグ専用ツールと連携できる仕組みを確立する。

デバッグ機能とインターフェースの追加を簡単に行えることにより、コンテンツ製作者や QA スタッフにとって便利な機能も迅速に拡充できるよになる。それにより、全体的な開発効率と品質が向上し、プロジェクトの進行をスムーズにすることを最大の目的とする。

■ 要件定義

▼ 基本要件

本書が扱うシステムの基本要件は下記のとおり。

- デバッグメニューからデバッグ機能を実行できるものとする。
- デバッグメニューの項目は、外部データ化して扱えるものとし、分かり易い項目の配置や日本語の表示が柔軟に行えるものとする。
- PC 上のデバッグツールからゲームにネットワーク接続して、専用の GUI でデバッグ操作やデバッグ情報を扱うこともできるものとする。
- 制作スタッフや QA スタッフからの要請に迅速に応じることができるように、デバッグ機能の追加実装が、きわめて簡単に行えるシステムとする。

▼ 要求仕様／要件定義

以下、本書が扱うシステムの要件を定義する。なお、要件として不確定の要求仕様も併記する。

- デバッグ機能の実行は、ゲーム上のコマンドプロンプトから実行するコマンドとして構成することを基本とする。
  - 例えば、「CPU ON」というコマンドを実行すると、CPU 使用率メーターが画面に表示される。
    - この場合、「CPU」がコマンドで、「ON」がコマンドパラメータである。
- デバッグメニューシステムを用意し、メニュー項目に応じたデバッグコマンドが実行されるシステムとする。
  - メニューにはコマンドのリストが並び、選択して実行することができる。
  - メニュー項目は、コマンドがそのまま表示されるのではなく、見て分かり易い見出し（名前）をつけることができる。
    - 例えば、「CPU 使用率」と書かれたデバッグメニューを選択して決定ボタンを押すと、内部で「CPU ON」コマンドが実行され、CPU 使用率メーターが画面に表示される。
  - メニューの実装が済んでいないコマンドも、コマンドプロンプトから直接実行することができる。
- デバッグメニューシステムのメニュー構成は、外部データ化して定義可能とする。
  - メニューの表示項目（日本語＆英語）、メニューに対応した実行コマンド、コマンドパラメータ、サブメニューを扱えるものとする。
  - 同じメニュー項目でも、状況によってコマンドおよびコマンドパラメータが変わるようなものも扱えるものとする。
    - 例えば、CPU 使用率メーターが画面に表示されている状態で「CPU 使用率」メニューを実行すると、「CPU OFF」コマンドが実行されて CPU 使用率メーターが非表示になる。
  - デバッグメニューの定義は JSON 形式（テキスト）で定義する。
    - 別紙の「ゲームデータ仕様」のデータ仕様に準拠し、バイナリデータに変換して扱う。
- PC 上のデバッグツールから、ゲームにネットワーク接続して、リモートでデバッグコマンドを実行することに対応するものとする。
- デバッグコマンドの実行結果は、その実行方法に応じた出力を行うものとする。
  - コマンドによっては特に何も出力されない。
    - 例えば、「CPU」コマンドは、画面上の CPU メーターの表示 ON/OFF が行われるだけで、特に何も出力しない。

- コマンドプロンプトから実行したコマンドは、標準出力（デバッグコンソール）に出力される。
  - 例えば、「FRATE」コマンドの結果は、標準出力に「30 fps」といった結果が出力されて、内容を確認することができる。
- デバッグメニューから実行したコマンドは、標準出力、もしくは、デバッグメニュー自体に表示される。
  - 例えば、メニュー項目として「フレームレート [%s]」、「メニュー表示用コマンド」として「FRATE」が定義されたメニューには、「フレームレート [30 fp]」という項目が表示される。
- リモートから実行したコマンドは、接続元に実行結果が送信される。

・ デバッグ機能（コマンド）を簡単に実装するための中心機能として、デバッグコマンドインタープリタを用意するものとする。
- デバッグ機能を追加する際は、デバッグコマンドとそれに応じたコールバック関数を登録する。
  - コールバック関数は、main() 関数のように任意の数のパラメータを文字列で受け取れる形式。
- デバッグコマンドは内部で CRC 値化され、ハッシュテーブル、もしくは、バイナリサーチ可能な構造で記録される。

・ ゲーム内の各種スクリプトからもデバッグコマンドを実行可能とする。
- スクリプト開発を補助するデバッグ機能はデバッグコマンドとして登録して扱う事を基本とする。

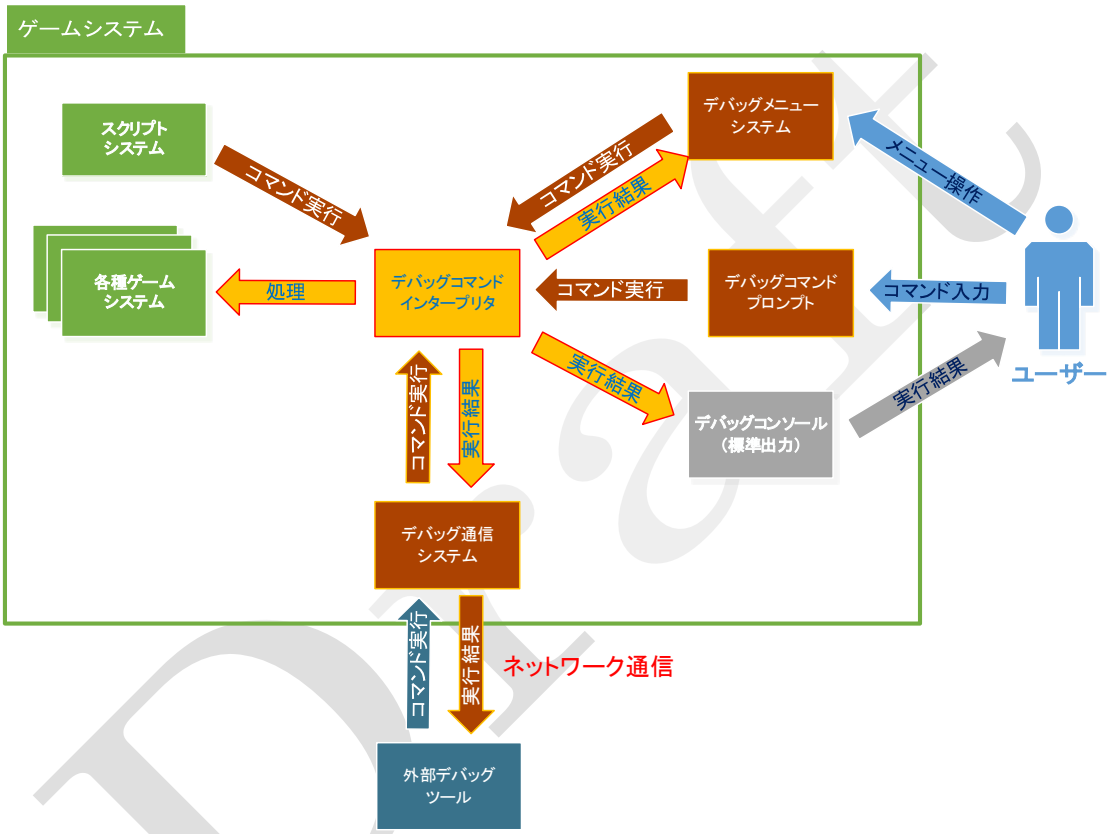
・ デバッグコマンドはリリースビルド時に実行されても無視される。

■ 仕様概要

▼ システム構成図

要件に基づくシステム構成図を示す。

デバッグシステム関係のシステム構成図：



なお、Release ビルドではデバッグ関係のシステムは全て消滅し、スクリプトシステム音デバッグコマンド実行処理（関数）は、呼び出されても何もしない処理になる。

■ データ仕様

【構想】

・ デバッグメニューシステムのデータ構造のイメージ（案）。

```
//デバッグメニュー定義データ構造
{
"DebugMenu":
[
```

```
{    //メインメニュー
     "id": CRC("main"), "items":
     [     { "caption": { "j": "CPU 使用率",
                          "e": "CPU ratio"},
             "cmd": { "cmd": CRC("CPU"), "params": "CHANGE" }//コマンド
           },
           { "caption": { "j": "フレームレート [%s]",
                          "e": "Frame-rate [%s]"},
             "disp_cmd": { "cmd": CRC("FRATE") }//メニュー表示用コマンド
           },
           { "caption": { "j": "サブメニュー",
                          "e": "Sub Menu"},
             "sub_menu": CRC("sub1")//サブメニュー指定
           }
     ],
},
{    //サブメニュー１
     "id": CRC("sub1"), "items":
     [     { "caption": { "j": "CPU 使用率 ex",
                          "e": "CPU ratio-ex"},
             "case_cmd": { "cmd": CRC("GET_CPU") },//分岐判定コマンド
             "switch_cmd"://分岐コマンド
             [ {"case": "ON", "when": { "cmd": CRC("CPU"), "params": "OFF"} },//分岐コマンド１
               {"case": "OFF", "when": { "cmd": CRC("CPU", "params": "ON"} }//分岐コマンド２
             ]
           }
     ]
}
]
}
```

■ 処理仕様

【構想】

- デバッグ通信システムはマルチスレッドで構成する。
  - セッション接続の待ち受けスレッドを用意する。
  - 複数のツールからの同時接続に対応するために、セッションの数だけスレッドを複製して通信する。
- デバッグコマンドプロンプト、デバッグメニュー、デバッグ通信システム、スクリプトシステムのすべてのコマンドと連携するために、デバッグコマンドは一旦キューイングして処理する。
  - これにより、通信スレッドからのデバッグコマンドの呼び出しを安全に行うようにする。
  - コマンドキューに登録されるコマンドには、出力用の FIFO ディスクリプタをセットにする。
    - stdout や stderr のようなもの。要するに出力メッセージ（ストリーム）を扱うキュー。
  - これにより、通信スレッドごとに的確にコマンドの結果を送信できるようにする。
  - FIFO バッファ、FIFO ディスクリプタの定義方法は現状未定。

・ デバッグコマンドインタープリタは、メインループのある時点でキューイングされたコマンドの存在を確認し、一括処理する。

・ コマンドインタープリタへのコマンドコールバック関数登録処理のイメージ（案）：

```
//デバッグコマンド用コールバック関数：CPU 使用率表示 ON/OFF
static void debugCmd_Cpu(DEBUG_DESC desc, const int argc, const char* argv[])
{
	if (argc <= 2)
		return;
	const char* p = argv[1];//パラメータ取得
	CSingletonUsing<CDebugState> debgu_state("debug");//デバッグ情報シングルトン取得
	if(stricmp(p, "ON") == 0) debgu_state->setCpu(true);//CPU 使用率表示 ON
	else if(stricmp(p, "OFF") == 0) debgu_state->setCpu(false);//CPU 使用率表示 OFF
}

//デバッグコマンド用コールバック関数：フレームレート表示
static void debugCmd_Frate(DEBUG_DESC desc, const int argc, const char* argv[])
{
	CSingletonUsing<CSystemInfo> sys_info("debug");//システム情報シングルトン取得
	DEBUG_OUTPUT(desc, "%d fps", sys_info->getFrate();//フレームレートを表示
}

//デバッグコマンドリスト登録
//※main()関数実行前にコマンドを CRC 値化
static DEBUG_COMMAND s_debugCmdList[] =
{
	{ AUTO_CALC_CRC("CPU"), debugCmd_Cpu },//CPU 使用率表示 ON/OFF
	{ AUTO_CALC_CRC("FRATE"), debugCmd_Frate },//フレームレート表示
};

//デバッグコマンド登録
//※main()関数実行前に登録処理を実行
static REGIST_DEBUG_COMMAND s_debugCmdRegister(s_debugCmdList);

//---------------------------------------
//上記の処理で使用している基本型と関数の定義

//基本データ型
typedef unsigned int CRC32;//CRC 値型
typedef int DEBUG_DESC;//デバッグ出力用ディスクリプタ

//デバッグ出力用関数
void DEBUG_OUTPUT(DEBUG_DESC desc, const char* fmt, ...)
{
	//... (略) ...
}

//デバッグコマンド型
typedef void (*DEBUG_COMMAND_CB)(DEBUG_DESC, const int, const char*[]);//コールバック関数型
struct DEBUG_COMMAND
{
	CRC32 m_cmdCrc;//コマンドの CRC 値
	DEBUG_COMMAND_CB m_cbFunc;//コマンドコールバック関数
};

//CRC 自動計算用構造体
//※main()関数実行前に計算を済ませて即結果を返すための構造体
struct AUTO_CALC_CRC
{
	operator CRC32 () const {return m_crc;}//キャストオペレータ
	AUTO_CALC_CRC(const char* str)//コンストラクタ
	{
		m_crc = CALC_CRC(str);//CRC 計算
	}
```

```
	CRC32 m_crc;//CRC 値
};

//デバッグコマンド自動登録用構造体
//※main()関数実行前に自動登録するための構造体
struct REGIST_DEBUG_COMMAND
{
	//コンストラクタ
	template<class T, size_t N>
	REGIST_DEBUG_COMMAND(T (&cmd_list)[N])
	{
		//全件処理
		for (auto node : cmd_list)
		{
			//...（略）...
		}
	}
};
```

■■以上■■

■ 索引

索引項目が見つかりません。

デバッグ制御システム

以　上