効果的なデバッグログとアサーション

– 効果的なデバッグトレース手法 –

2014 年 2 月 24 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 24 日	板垣 衛	（初稿）

■ 目次

■ 概略

より開発効率を向上させるための、一歩進んだデバッグログとアサーションのシステムを設計する。

■ 目的

本書は、問題発生時に迅速に的確な原因追跡を行うことを可能とするシステムを設計し、開発効率を向上させることを目的とする。

開発環境以外のランタイム時に発生する問題を捕捉し、原因追求を可能とすることも目的とする。

■ マルチスレッドについて

本書のデバッグロギングシステムは、マルチスレッドシステムとして最適化するため、以降の説明にはマルチスレッドプログラミングの用語を用いている。用語の説明については、別紙の「マルチスレッドプログラミングの基礎」を参照。

本書では、「アトミック型」（または「インターロック操作」）、「セマフォ」、「モニター」（「条件変数」）といった要素を扱っている。

■ 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。

- マルチスレッド環境で安全に利用可能なデバッグログシステムとする。
- デバッグログの出力処理がメイン処理の実行のパフォーマンスを極力阻害しないものとする。
- スレッドの先着処理順に的確に表示しつつ、スレッドの処理をブロックしないものとする。
- 共通処理のログは「何の処理に使われたものか」がすぐにわかるようなシステムとする。

・ ログレベル（ログ表示のマスク）はランタイム時に動的に変更できるものとする。
・ スタッフごとに「既定のログレベル」を設定できるものとする。
・ アサーション違反発生時はブレークポイント割り込みを発生させるものとする。
・ アサーション違反発生時はコールスタックを表示するものとする。
・ 重大なログは画面にも表示し、ランタイム時にも問題発生を気づかせるようにするものとする。
・ ランタイムでのアサーション違反発生時は、ゲームを自動的にポーズするものとし、さらに、その後の処理続行も可能なものとする。
・ メッセージ出力時は文字コードを指定する事で、シフト JIS コードと UTF-8 コードのどちらも表示できるものとする。

▼ 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。

・ デバッグログは、メッセージをキューイングすることでゲーム本体のパフォーマンスに極力影響しないものとする。

・ キューイングされたメッセージは、ログ出力スレッドによって逐次出力する。
  ➢ ログ出力スレッドの優先度は、メインスレッドよりも低く設定する。

・ メッセージ出力時は、メッセージのレベル（重大度）とカテゴリを指定する。

・ メッセージ出力時は、（ランタイムの）ログレベルにより、出力するメッセージをレベルに応じて制限する。
  ➢ つまり、常にすべてのログがビルドされる。（プログラム上に実装される。）
  ➢ メッセージ出力のマスク判定はメッセージ出力関数内で行い、出力しない場合はメッセージのキューイングも行わない。

・ ランタイム時の「ログレベル」の変更は、デバッグメニューなどの外部のシステムから行えるものとする。

・ 規定の「ログレベル」は、「ユーザー設定ファイル」に記録される情報の一つとして扱い、ユーザーごとに規定を設定することを可能とする。
  ➢ 「ユーザー設定ファイル」は、ユーザーごとのゲームの挙動を設定したファイルで、ユーザー自身の PC 上に配置して扱う。ゲーム起動時のそのファイルが読み込まれる。
  ➢ 主に制作スタッフや QA スタッフが自身に関係のあるメッセージしか表示させないようにするために使用する。

・ ゲーム画面上への表示が必要なメッセージ（とくに重大度の高いものなど）については、ログ出力スレッドが画面表示用のキューにメッセージを再投入して扱う。
  - つまり、メッセージを「どう扱うか？」の判断は、やはりログ出力スレッドが行う。
  - 「どこまでのログを画面に表示させるか？」については、デバッグメニューやユーザー設定ファイルを通して、ランタイム時に設定できるものとする。

・ ゲーム画面にログを表示するシステムは、別途メインスレッド（メインループ）で稼働し、キューを拾って逐次処理する。
  - ログをピックアップしてキューから削除するシステムと、画面に表示するシステムは処理のレイヤーを分け、ゲームタイトルに応じた表示処理を行えるものとする。
  - 画面に大きくメッセージを表示するような場合は、メッセージを全文表示しないような工夫を行う。例えば、「エラー : XXX ¥t 理由…¥n…」といったメッセージの場合、画面上には「¥t」や「¥n」までしか表示しないといった対応を取る。または、40 文字で表示を打ち切るなど、タイトル固有の仕様で実装する。

・ メッセージのバッファはロギングシステムが用意し、（ゲーム側の）各スレッドが自身のスタックをほとんど使用しないものとする。
  - これにより、少ないスタックで動作するスレッドからもメッセージの長さをほとんど気にせずに、ログ出力できるものとする。
  - ロギングシステムはあらかじめ幾つかの固定バッファ（2KB×16 個ほど）を用意しておき、ログ出力の都度未使用のバッファが割り当てられる。
    - バッファを「16 個ほど」と多めに記したのは、それだけのスレッドが同時にプリントする可能性を考慮したものではなく、一気に大量のプリント文が列挙される可能性を考慮したもの。
  - ログ出力処理（ゲーム側がログを出力する処理）は、固定バッファへの書き込みが終了したら即座に処理が返る。
  - 固定バッファが全て使用中で割り当てができない時は、ゲーム側のスレッドがブロックして待機する。
    - これが起こらないように、十分な数のバッファを用意しておかなければならない。
    - バッファのサイズと数はゲームタイトルに応じて設定可能。
    - ブロック処理には、単純にはセマフォを活用。ゲーム終了時に確実に開放し、解放忘れのデッドロックを起こさないように注意。
  - メッセージをキューイングするためのバッファは別途用意されており、固定バッファからのキュー（バッファ）への移し替えが完了した時に、固定バッファが解放される。
    - このため、固定バッファの利用状況を監視し、キューに移し替えするだけのスレッドも用意する。
    - この固定バッファ監視スレッドの優先度はメインスレッドと同じか、それより高く設定する。（ゲーム側の処理をブロックすることにつながるため）

・ メッセージにはシーケンス番号を付け、マルチスレッドで確実に処理の到達順序を保証して出力するものとする。
  - アトミック型のシーケンス番号を用意し、ログ出力時には真っ先にシーケンス番号の発番を行う。
    - アトミック型もしくはインターロック操作を用いることにより、ロックフリーな発番を、各処理スレッド自身で行う。
  - 例えば、二つのスレッドがほぼ同時にログ出力処理を行った際、先着のスレッドの方が書き込みに時間がかかり、完了が後になる可能性がある。このような場合でも、ログ出力スレッドは、シーケンス番号どおりのメッセージが到着するのを待ってから出力する。
    - ただし、結局ログ出力処理によって、スレッドの本来の処理効率を損なう問題は残る。この対処として、シーケンス番号だけ先に発番して、後で（一通りの処理が完了してから）メッセージを書き込む処理にも対応する。
    - シーケンス番号の先行発番は複数行ってもよい。
    - 仮に「シーケンス番号を先行発番したが、対応するメッセージが出力されなかった」ということがあると、そこでログ出力が完全に停止してしまう。この問題の対処として、ログ出力スレッドは、一定時間待ってもキューイングされない場合はタイムアウトしてスキップするものとする。その待機時間はせいぜい 1～2 ミリ秒ほど。
    - タイムアウト時間を過ぎて書き込まれたメッセージもログ出力される。
    - ユーザーが挙動を完全に捕捉できるように、オプションにより、シーケンス番号付きでメッセージをログ出力するモードにも対応する。（デバッグメニューなどから ON/OFF できる）

・ 「コールポイントシステム」を用意し、共通処理のメッセージを呼び出し元に応じてマスクすることを可能とする。
  - 処理ブロック内（関数の先頭など）で、「コールポイント宣言」を行うことで、共通処理のログ出力時に「どのコールポイントを通過したか？」を判定できる。
  - コールポイントは階層的にスタックされ、「コールポイントに基づくログ出力マスク」のような処理は、直近のコールポイントに基づいて処理する。
  - コールポイントの情報は、ローカル変数の情報をそのまま扱う。
  - TLS（スレッドローカルストレージ）を活用し、スレッド間で干渉することなく、安全にコールポイントのスタックを扱う。

・ アサーションにもこのロギングシステムを活用する。
  - アサーション違反発生時は、ブレークポイント割り込みを発生させる。
    - Windows なら DebugBreak()関数の呼び出し、Unix 系なら SIGTRAP シグナルの発行など。
  - アサーション違反時は、ランタイムでも気がつけるように、重大なメッセージとして、ログ出力、画面表示も行う。

- ログ出力では、コールスタックも表示する。
  - ランタイムでシンボル情報を読み取ってコールスタックを表示するライブラリがある。後述する。
- ログ出力では、コールポイントのスタックも表示する。
  - 情報が冗長するようなら、コールスタックかコールポイントスタックのどちらかは表示しないように処理を変える。
- 【オプション】アサーション違反発生時は、自動的にゲーム（ゲームループ）をポーズ状態にし、「START」ボタンなどで続行可能とする。
- ブレークポイント割り込みを呼び出す直前に、キューイングされたログをフラッシュ（強制全出力）する。
  - メッセージを出力したあと、すぐにフラッシュし、フラッシュの完了を待つ。
  - コールポイントスタックの出力は、通常のメッセージ出力の仕組みを用いられるため、フラッシュを行う前に出力する。
  - コールスタックの出力は、Boost C++ ライブラリの backtrace を用いることを想定しており、「`std::ostream <<`」で出力する必要がある。このインターフェースには無理に対応せず、フラッシュの完了を待ってから、「`std:cerr << boost::trace(e);`」といった処理でそのまま出力する。
  - 処理順序としては、「メッセージ出力」→「コールポイントスタック出力」→「ログ出力フラッシュ」（完了待ち）→「コールスタック出力」→「ブレークポイント割り込み」（開発環境以外では無反応）→「ポーズ」となる。

・アサーション以外にも、任意の「ウォッチポイント」、「ブレークポイント」を設定可能とする。
- ウォッチポイントはアサーションと逆に「ブレークする条件」を式に記述する。
- ウォッチポイント、ブレークポイントは、メッセージをログ出力してブレークポイント割り込みを発生させる。
  - この時のメッセージの重大度も指定する。
  - ログレベルによってマスクされたメッセージのウォッチポイント、ブレークポイントは、ブレークポイント割り込みも反応しない。

・メッセージ出力時は、そのメッセージが「UTF-8」か「シフト JIS」か指定できるものとする。
- 文字コード変換は、ログ出力スレッドがメッセージを出力する際に行う。
- 一つのゲームシステム上で、両方の文字コードを意識的に使用するケースがあるため、この仕組みを用意する。
  - UTF-8 の方が欧州文字の扱いなど表現の問題が少ないが、シフト JIS の方がテキストのサイズが小さくなるため、ゲーム画面上で扱わないテキストやソースコード埋め込みのテキストは

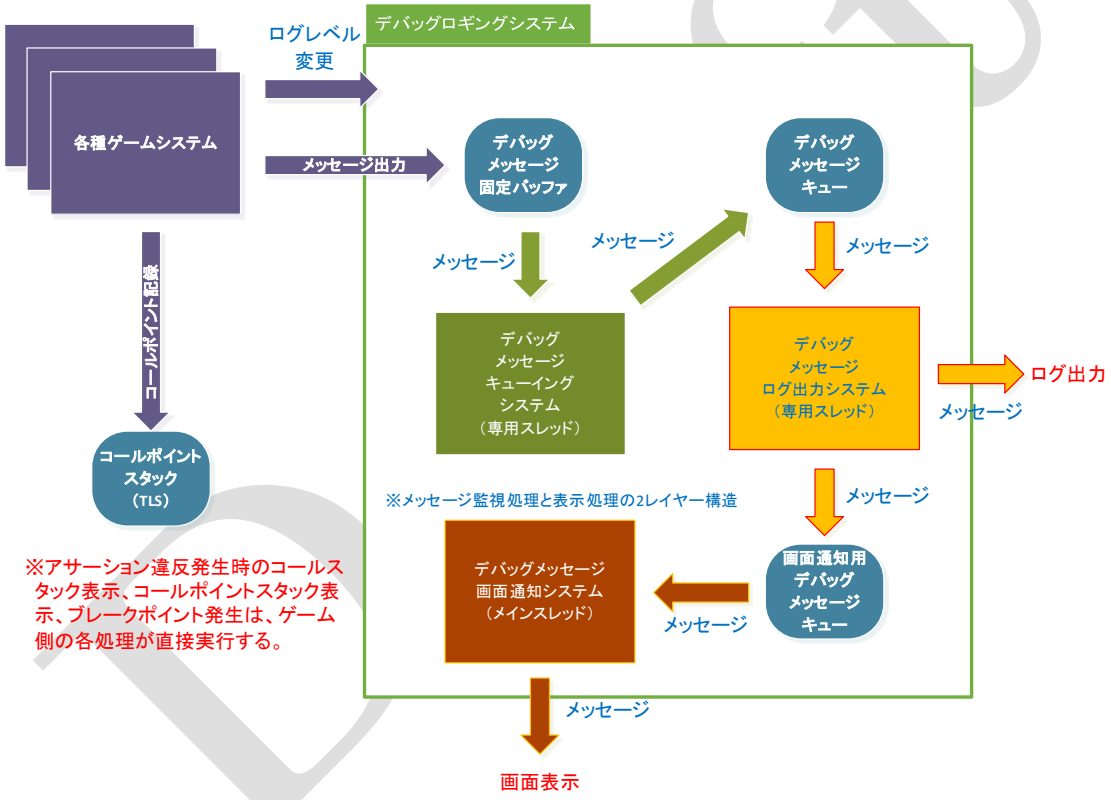
シフト JIS コードで扱われることも多い。

■ 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。

デバッグロギングシステムのシステム構成図：



■ 処理仕様

処理仕様として、各処理の使用方法を示す。

▼ デバッグメッセージ

デバッグメッセージは基本的にプリント文である。重大度とカテゴリを指定してプリン

トする。

　ログレベルの設定と画面通知レベルの設定が可能。ログレベル、画面通知レベルはカテゴリごとに設定する。全カテゴリの一括設定も可能。

　また、スレッド間の処理到達順序を保証したメッセージ出力のための「遅延メッセージ」に対応する。

　メッセージのシーケンス番号を表示に加えることも可能。

デバッグメッセージの使用イメージ：

```
//---------------------------------------
//デバッグメッセージ出力
//※メッセージレベルとカテゴリを指定

using debug_ptin;//関数や定数を簡潔に使用するためのネームスペース using 宣言

//Lv.1：通常メッセージ
printAsNormal(forAny, "通常メッセージ by %s¥n", name);
//Lv.0：冗長メッセージ
printAsVerbose(forAny, "冗長メッセージ by %s¥n", name);
//Lv.0：詳細メッセージ
printAsDetail(forAny, "詳細メッセージ by %s¥n", name);
//Lv.2：重要メッセージ
printAsImportant(forAny, "重要メッセージ by %s¥n", name);
//Lv.3：警告メッセージ
printAsWarning(forAny, "警告メッセージ by %s¥n", name);
//LV.4：重大（問題）メッセージ
printAsCritical(forAny, "重大メッセージ by %s¥n", name);
//Lv.5：絶対（必須）メッセージ
printAsAbsolute(forAny, "絶対メッセージ by %s¥n", name);

//レベルを引数指定
printLv(asNormal, forAny, "通常メッセージ by %s¥n", name);
//※数字で指定しても可
//※Lv.0～5 以外を指定した場合、ログレベルによらず何も出力されない
printLv(2, forAny, "重要メッセージ by %s¥n", name);

//---------------------------------------
//ログレベル

//ログレベル変更
//※指定の値以上のレベルのメッセージのみをログ出力する
setLogLv(asWarning, forAny);//警告以上のメッセージのみログ出力

//ログレベル取得
getLoglv(forAny);//戻り値としてログレベルが返る

//---------------------------------------
//画面通知レベル

//画面通知レベル変更
//※ログレベルより低いレベルを設定しても通知されない
setNoticeLv(asCritical, forAny);//重大メッセージのみ画面通知
//※各レベルの推奨表示カラー
//  Lv.1：通常      ... 黒
//  Lv.0：冗長/詳細 ... （表示されない）
//  Lv.2：重要      ... 青
//  Lv.3：警告      ... 紫
//  Lv.4：重大      ... 赤
//  Lv.5：絶対      ... （表示されない）

//画面通知レベル取得
```

```
getNoticeLvAs(forAny);//戻り値として画面通知レベルが返る

//----------------------------------------
//定数

//レベル定数
enum E_MESSAGE_LV
{
	asNormal = 1,//通常メッセージ
	asVerbose = 0,//冗長メッセージ
	asDetail = 0,//詳細メッセージ
	asImportant = 2,//重要メッセージ
	asWarning = 3,//警告メッセージ
	asCritical = 4,//重大メッセージ
	asAbsolute = 5,//絶対メッセージ（ログレベルに関係なく出力したいメッセージ）
	//以下、ログレベル／画面通知レベル変更用
	asSilent = 5,//静寂（絶対メッセージ以外出力しない）
	asSilentAbsolutely = 99,//絶対静寂（全てのメッセージを出力しない）
};

//カテゴリ定数
//※制作スタッフにとって分かり易く、
//  メッセージを仕込むプログラマーにとって
//  分かり易い程度の分類にする
//※forReserved**を任意の用途に割り当てて使用可
enum E_MESSAGE_CATEGOY
{
	forAny = 0,//なんでも（カテゴリなし）
	forLogic = 1,//プログラム関係
	forResource = 2,//リソース関係
	for3D = 3,//3D グラフィックス関係
	for2D = 4,//2D グラフィックス関係
	forSound = 5,//サウンド関係
	forScript = 6,//スクリプト関係
	forGameData = 7,//ゲームデータ関係
	forReserved01 = 8,//(予約 01)
	//...（略）...
	forReserved56 = 63,//(予約 56) ※0～63 まで使用可
	//以下、特殊なカテゴリ（プリント時専用）
	forCallppoint = 1001,//直近のコールポイントのカテゴリに合わせる（なければ forAny 扱い）
	forCriticalCallppoint = 1002,//直近の重大コールポイントのカテゴリに合わせる（なければ forAny 扱い）
	//以下、ログレベル／画面通知レベル変更用
	forEvery = 0xffffffff,//全部まとめて変更
};
```

一塊の複数のメッセージを扱う場合：

```
//----------------------------------------
//デバッグメッセージ出力
//※一塊の複数のメッセージを扱う場合

//画面通知をしないメッセージを明示する
//※prLog***()関数は、ログ出力専用
prLogAsCritical(forAny, "----------------------------------------¥n");
printAsCritical(forAny, "重大メッセージ：%s¥n", name);
prLogAsCritical(forAny, "...理由など...¥n");
prLogAsCritical(forAny, "----------------------------------------¥n");
//※一塊の複数のメッセージの内、一部しか画面通知に反応させたくない場合に使い分ける
//※逆に画面通知専用関数は、notice***()。
```

遅延デバッグメッセージの使用イメージ：

```
//----------------------------------------
//遅延デバッグメッセージ
//※スレッド間での処理到達の順序性を厳密に保証してプリントするための仕組み
```

```
//メッセージシーケンス番号予約
reservePrint reserved(asCritical, forAny, 3);//予約数を引数で指定する
//※reservePrint はクラスであり、内部で予約した番号（64bit）を保持する
//※予約の時点でレベルとカテゴリを指定し、ログレベルが適合せず出力されない場合は予約されない
//※2 つ以上の予約の際は、連番で予約されることを保証する
//   (std::atomic<long long>::fetch_add(n) などを使用して安全なロックフリー処理で確保する)

//...（処理）...

//遅延メッセージ出力
reserved.print("重大メッセージ１：%s", name);
reserved.print("重大メッセージ２：%s", name);
reserved.print("重大メッセージ３：%s", name);
//※メッセージレベルにより、予約が失敗している場合は、メッセージの出力は無視される
//※出力するごとに reserved の内部のカウンタが進み、予約数を超えたものは出力されない
//※この仕組みを利用し、一塊の複数のメッセージの途中で、他のスレッドのメッセージが
//  割り込むことがないようにすることができる

//メッセージ出力キャンセル
reserved.cancelPrint();
//※未使用の予約番号はキャンセルされ、欠番扱いになる
//※処理ブロックを抜ける際、reservePrint クラスのデストラクタにより、
//  キャンセル処理は自動実行される
```

デバッグメッセージ処理属性の使用イメージ：

```
//--------------------------------------
//デバッグメッセージ処理属性

//デバッグメッセージ処理属性をセット
setPrintAttr(withSeqNo);//属性追加：メッセージ出力時にシーケンス番号を付加
//表示例：
//  [23]重大メッセージ：func1
//  [24]通常メッセージ：func2

//デバッグメッセージ処理属性をリセット
resetPrintAttr(withSeqNo);//属性解除：メッセージ出力時のシーケンス番号の付加を取りやめ
```

▼ コールポイント

コールポイントは明示的なコールスタックのようなもので、アサーション違反のような問題発生時に呼び出し履歴を表示したり、通常のメッセージをログ出力する際に直前のコールポイントの情報を表示したりといった使い方をする。

また、「共通処理のアサーション判定を無効にする」といった、以降の処理でのメッセージやアサーションの挙動を制御することにも用いる。

● コールポイントの使用イメージ

まず、コールポイントの使用イメージを示す。

コールポイントの使用イメージ：

```
//関数１
void func1()
{
```

```
	callPoint call_point(forAny, "func1");//コールポイント設定（カテゴリと名前を与える）
	func2();//関数２呼び出し
	func3();//関数３呼び出し
}
//関数２
void func2()
{
	commonFunc();//共通関数呼び出し
}
//関数３
void func3()
{
	callPoint call_point(forLogic, "func3");//コールポイント設定（カテゴリと名前を与える）
	commonFunc();//共通関数呼び出し
}
//共通関数
void commonFunc()
{
	printCallPointStackAsCritical(forAny, "commonFunc");//コールポイントスタック表示（カテゴリと名前を与える）
}
```

↓（実行結果）

```
---------------------------------------
Call point stack at "commonFunc"
"func1" forAny
---------------------------------------
---------------------------------------
Call point stack at "commonFunc"
"func3" forLogic
"func1" forAny
---------------------------------------
```

「printCallPointStack***()」を実行すると、それまでに通過したコールポイントの名前とカテゴリが逆順（スタック順）で表示される。ここで与える名前はログ出力に用いられ、出力の有無はメッセージ出力と同様にカテゴリとログレベルに従って判定される。

なお、この関数は通常のデバッグプリント文の仕組みを通してログ出力スレッドに出力されるが、どのログレベルであっても画面通知されることはない。

● コールポイントの仕組み

コールポイントは、TLS(スレッドローカルストレージ)を利用した連結リストで管理する。このため、他のスレッドが干渉することのない、そのスレッドだけのコールスタックを作る事ができる。

「callPoint」クラスのコンストラクタで連結リストにつなぎ、デストラクタで自動的に外す。自身のスレッドに限定された処理なので、とくに何らかのメモリ確保もせず、ローカル変数をそのまま連結する。

TLS については別紙の「マルチスレッドプログラミングの基礎」を参照。

● コールポイントの高度な利用イメージ

コールポイントを利用することで、共通処理の挙動を変えることができる。

デバッグメッセージを出力する際は、前述のとおり「カテゴリ」を指定する。この時、「forCallpoint」カテゴリを指定すると、直近のコールポイントのカテゴリと同じ扱いになる。

また、コールポイントは一時的にログレベルを引き上げる機能にも対応する。既存の設定より引き上げるだけで、下げることはできない。

以上の機能を、例えば「メモリマネージャ内のメモリ確保失敗検出用のアサーション」に活用する。

メモリ確保失敗は重大な問題であるが、「メモリ確保失敗時は、次のフレームでリトライする」といった処理要件では、このアサーションが邪魔になる。通常の対応では、やむなくアサーションを外すか、アサーション実行判定のためのフラグを渡すなどの煩雑な対応になっていく。new 演算子を通すことを考えると、後者の対応はさらに難しい。

この対処として、コールポイントを利用する。

デバッグ機能だけで完結するので、通常の処理インターフェースを汚すこともない。

以下、このような利用例を具体的に示す。

コールポイントの高度な利用イメージ：

【memory_manager.cpp】

```
//メモリ確保
void* CMemoryManager::alloc(const std::size_tsize)
{
	//...（処理）...
	//メモリ確保成功アサーション
	assertAsCritical(forCallpoint, p != nullptr, "not enough memory!");
	//※「forCallpoint」カテゴリ指定により、コールポイント時のカテゴリに基づいてアサーションを判定する
	//...（処理）...
	return p;
}
```

【resource_manager.cpp】

```
//リソース構築
template<class T>
T* CResourceManager::createResource()
{
	//...（処理）...
	//インスタンス生成
	T* p = nullptr;
	{
		//コールポイントを設定
		callPoint call_point(forResource, "CResourceManager::createResource");
		//このコールポイントの間だけ、forResource カテゴリのログレベルを asSilentAbsolutely に設定
		call_point.setLogLv(asSilentAbsolutely, forResource);
		//インスタンス生成
		p = new(mem_man) T();
		//処理ブロックから抜けるとコールポイントが解除され、ログレベルも元に戻る
	}
	//...（処理）...
	return p;
}
```

これにより、通常はメモリマネージャ内のアサーションが判定されるが、リソース構築

処理の時だけ判定を行わないようにする。

● コールポイントの更なる活用：プロファイラとしての活用

コールポイントの仕組みをさらに拡張子、プロファイラとして活用する。

- コールポイントが開始と終了を自動的に処理することを利用し、処理時間を計測する。
- さらに、別途処理時間を集計する仕組みを設け、そこに記録していくようにする。
- 処理時間を記録する際は、別紙の「効率化と安全性のためのロック制御」で示すスレッド ID 管理の仕組みと組み合わせ、スレッド ID（スレッド名）＋処理名で記録する。
- コールポイントの親子関係も記録する。
- なお、処理時間を記録する際は、__FILE__ マクロと __LINE__ マクロを活用してユニークなキーとすると、集計しやすい。
- 「5 秒間隔」などの設定で集計を行えるようにし、重い処理をリアルタイムに画面表示するなどして活用する。
- 画面表示の際は、「カテゴリ」でマスクすることもできるため、特定のカテゴリに絞った確認も取りやすい。
- 以上の仕組みにより、例えば「描画スレッド」に限定した確認なども簡単に行える。
- リアルタイムのプロファイラであることが何より大きな特徴。プログラマーの開発環境以外でパフォーマンスの問題を確認した際にその場で調査できる。

▼ アサーション／ウォッチポイント／ブレークポイント

アサーションは普通によく使われるプログラミングの手段の一つであるが、標準関数のように（アサーション違反時に） abort() せず、ブレークポイント割り込みを発生させて、処理を継続できるものとする。

アサーションの処理は下記のとおり。

- 「評価式」を判定し、結果が「偽」なら、下記の処理を行う。「真」なら何もせず終了。
- 「評価式」、「任意のメッセージ」、「ソースファイル名」（__FILE__）、「ソースファイルのタイムスタンプ」（__TIMESTAMP__）、「関数名」（__FUNCSIG__など）、「行番号」（__LINE__）を表示。
- 「任意のメッセージ」のみを画面通知。
- コールポイントスタックを表示。
- コールスタックを表示。
- ブレークポイント割り込み。
- ゲームのポーズ要求。（ゲーム稼働中のみ）

アサーションも、デバッグメッセージと同様のレベルとカテゴリを扱い、時にはランタイム時に無効化させることができる。（本来推奨されるようなことではないが、前述のコールポイントのような利用に効果を発揮する）

アサーションと同様の機能で、もっとレベル（重大度）の低いチェックのために、「ウォッチポイント」の機能を設ける。「アサーション」は、「本当に起こってはいけないこと（誓約）」に用いるべきなので、きちんと使い分けて、アサーションの濫用を防ぐ。

「ウォッチポイント」の機能はほとんどアサーションと同じであるが、下記の点で異なる。

- 「評価式」が「真」の時に反応する。
- 画面通知は行わない。
- ゲームのポーズ要求は行わない。

ほか、コールポイント、コールスタック、レベル、カテゴリの扱いは同じ。

「ブレークポイント」は、「ウォッチポイント」の無条件版。

アサーション／ウォッチポイント／ブレークポイントの使用イメージ：

```
//アサーション
//※デバッグメッセージと同様に、レベルとカテゴリを指定する
//※続いて、評価式（偽なら停止）、メッセージ、メッセージのパラメータを指定する
assertAsCritical(forCallpoint, p != nullptr, "重大メッセージ:%s", name);
//※なお、アサーションは「重大」(asCritical) レベルにしか対応しない。

//ウォッチポイント
//※デバッグメッセージと同様に、レベルとカテゴリを指定する
//※続いて、評価式（真なら停止）、メッセージ、メッセージのパラメータを指定する
watchAsWarning(forCallpoint, p == nullptr, "警告メッセージ:%s", name);
//※ウォッチポイントは、「絶対」(asAbsolute) レベルを除く全てのレベル（0～4）に対応する

//ブレークポイント
//※デバッグメッセージと同様に、レベルとカテゴリを指定する
//※続いて、メッセージ、メッセージのパラメータを指定する
breakAsWarning(forImportant, "重要メッセージ:%s", name);
//※ブレークポイントは、「絶対」(asAbsolute) レベルを除く全てのレベル（0～4）に対応する
```

▼ コールスタック

開発環境以外のところでアサーション違反が発生した場合、コールスタックの表示が非常に役に立つ。

コールスタックは SDK の API などを用いて表示させることもできるが、Boost C++ のライセンスで公開されているソースコードがある。（Boost C++ のライブラリには組み込まれていない）

シンボル情報もたどって、きちんと名前で表示してくれるので便利。「例外」(try～catch) が必須であるほか、「std::ostream」「std::string」を内部で多用しているので、メモリには注意が必要。

backtrace.cpp と backtrace.hpp をそのままプロジェクトに組み込むだけですぐに使える。ゲーム機向けの SDK ではそのまま使えない可能性があるが、Windows や Linux には対応している。

ソースの配布元 URL は、ソースを入手した個人のブログしか見つけきれなかったので記載は省略。「boost backtrace」で検索すると見つかる。

boost::bactrace の使用イメージ：※例外のキャッチが必須

```
#include <boost/backtrace.hpp>

void debug::printInfo()
{
	//... (処理) ...
	//コールスタック表示
	try
	{
		throw boost::runtime_error("Test");//例外をスロー
	}
	catch (std::exception const &e)
	{
		std::cerr << boost::trace(e);//例外をキャッチしてバックトレースを std::ostream に出力
	}
	//... (処理) ...
}
```

↓（実行結果）

```
0132B88F: boost::stack_trace::trace +0x3f
0132E62A: boost::backtrace::backtrace +0x9a
0132E802: boost::runtime_error::runtime_error +0x62
013339B3: debug::printInfo +0x313
01333B90: debug::printInfo +0x40
01332487: debug::flushInfo +0x267
0133275E: debug::flushMessageBlock +0x3e
0132EDB2: debug::CMessageBlock::~CMessageBlock +0x42
01335549: sub +0x4e9
01335C2D: thread +0xed
01335F04: main +0x44
013369F9: __tmainCRTStartup +0x199
01336BED: mainCRTStartup +0xd
76FB495D: BaseThreadInitThunk +0xe
771F98EE: RtlInitializeExceptionChain +0x84
771F98C4: RtlInitializeExceptionChain +0x5a
```

▼ マルチスレッドでの状態監視について

システムの要件として、メッセージバッファの使用状態やキューイングの状態を監視する処理を行うものとしている。

このような処理は、ループでポーリングし続けるような処理ではなく、できるだけ「スリープ」することが望ましい。

「処理不要な時は可能な限り何もせずスリープ」し、「処理が必要になったら即座に反応する」という動作が望ましいので、ループしながらスリープを入れて反応が遅れるようなことも避けたい。

このような処理の対応としては、「モニター」の仕組みを用いると良い。POSIX スレッドライブラリや C++11 では「条件変数」が、WIN32 では「イベント」が使える。どちらも扱いに少しクセがあるが、「条件変数」の方が何かと柔軟に扱えてよい。

詳しくは別紙の「マルチスレッドプログラミングの基礎」を参照。

■■以上■■

■ 索引

索引項目が見つかりません。

効果的なデバッグログとアサーション

以　上