効率化と安全性のためのロック制御

－ －

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 1](#_Toc379631166)

[ 目的 1](#_Toc379631167)

[ 【要件】リソースアクセスの最適化 1](#_Toc379631168)

[▼ リード・ライトロックの活用 1](#_Toc379631169)

[▼ 最適化の前提 1](#_Toc379631170)

[▼ 要件定義 2](#_Toc379631171)

[▼ クラス設計 4](#_Toc379631172)

[▼ プログラミングイメージ 4](#_Toc379631173)

[ リソースマネージャを通したリソースアクセスのイメージ 6](#_Toc379631174)

[ 【要件】マルチスレッドで利用する共通処理の最適化 7](#_Toc379631175)

[▼ スレッドセーフなシングルトンパターンの活用 7](#_Toc379631176)

[▼ 要件定義 7](#_Toc379631177)

[▼ クラス設計 8](#_Toc379631178)

[▼ プログラミングイメージ 10](#_Toc379631179)

[ 処理実装サンプル 13](#_Toc379631180)

[▼ 準備①：インクルードファイルと基本マクロ 13](#_Toc379631181)

[▼ 準備②：スレッドIDクラス 14](#_Toc379631182)

[▼ 準備③：軽量スピンロッククラス 15](#_Toc379631183)

[▼ 準備④：メモリブロック確保クラス 17](#_Toc379631184)

[▼ リード・ライトロッククラス 20](#_Toc379631185)

[▼ リード・ライトロッククラスの使用サンプル 27](#_Toc379631186)

[▼ シングルトンクラス 33](#_Toc379631187)

[▼ シングルトンクラスの使用サンプル 44](#_Toc379631188)

[▼ 管理シングルトンクラス 50](#_Toc379631189)

[▼ 管理シングルトンクラスの使用サンプル 59](#_Toc379631190)

# 概略

ゲームプログラミングで有効なマルチスレッド処理を実践するために、具体的な処理要件に対する基本的な仕組みを設計する。

とりわけロック制御にフォーカスし、処理効率とプログラミング効率（手軽さ、安全性）を最適化した仕組みを確立する。

# 目的

本書は、別紙の「マルチスレッドプログラミングの基礎」を踏まえ、実際にゲーム開発で効果的なマルチスレッドプログラミングを実践することを目的とする。

なお、各要件に対して、Visual C++ 2013 で動作する完全なサンプルプログラムも合わせて記載している。本書は、実践的な処理設計そのものを目的とするものではなく、そうしたプログラミングの過程をトレースして、体感的にマルチスレッドプログラミングを修得することを目的としている。

# 【要件】リソースアクセスの最適化

リソースマネージャを通したリソースアクセスをマルチスレッドに最適化する。

## リード・ライトロックの活用

本件は、リード・ライトロック、スコープドロックパターンを活用する事例として説明する。

## 最適化の前提

リソースマネージャの処理は、ゲームループ中のある一時点でのみ、集中的にリソースの追加（可視化）、破棄、再配置を行う。

* 他の多くの処理はリソースを読み取り専用で扱うため、リソースマネージャの書き換えタイミングと競合する機会が極力なようにする。
* リソースマネージャの処理以外にも、リソースの内容更新を行う処理がある。それは、内部の値の変更だけで、リソースそのものの追加や削除を行う事はない。

リソースマネージャは、グラフィックデータの他、インスタンス（姿勢データ）、サウンドデータ、ゲームデータなど、ゲーム中でメモリ管理する要素の多くを扱う。

* リソースマネージャが備えるスレッドセーフ機構を多くの場面で有効活用し、マルスレッドを安全かつ効率的に処理する。

## 要件定義

本件の具体的な処理要件を定義する。

プログラミング構造上、リソースアクセス時は必然的にロックする。

プログラミング構造上、リソースアクセス時のロック解放は自動的に行われる。

* C++11のミューテックスロック用クラス「std::lock\_guard」と同様に、デストラクタでロック解放する仕組みとする。
* 処理ブロック（スコープ）から抜ける時にデストラクタが呼び出されてロックを解放する。「スコープドロックパターン」（Scoped Lock Pattern）と呼ばれるプログラミング手法。
* これにより、コーディングミスや、例外発生、不測のreturnがあっても確実にロック解放される。
* 明示的なロック解放も可能。その場合、デストラクタではロック解放しない。

複数の処理が同時にリソースの読み込みを行うことを可能とする。

* リソースアクセス処理の多くは、読み取り専用で更新を必要としない。

リソースの書き込み・破棄は、同時に行われる全てのリソースアクセス処理をブロックする。

* 読み込みも書き込みもブロックする。書き込み時は一切同時アクセスできない。

上記のロックを実現するために、リード・ライトロック機構を用いる。

* リードロックは重複してロック取得可能。
* リードロック中はライトロック取得をブロックする。
* ライトロック中は全てのロック取得をブロックする。

リード・ライトロックは、個々のリソース個別に対して行うことを可能とする。

* 何千という大量のロックオブジェクトを扱うことを可能とする。
* そのため、OSが（ハンドルを発行して）管理するような同期オブジェクトは用いない。

ロック取得待機時には、並列処理に最適化するため、スピンロック＋スリープを行う。

メインスレッド（メインループ）でしか更新しないことが判明しているリソースに対しては、オプションにより、メインスレッドでのリードロックをスルーできるものとする。

* これにより、少しでも高速化する。
* 待機が発生しない状況でも、ロックの取得には若干の処理時間を要するため。

基本的にライトロックを優先とする。

* リードロックが混み合っている時にライトロック取得が待機に入ったら、その後のリードロック取得要求は、ライトロックが解放されるまで待機する。
* オプションにより、この挙動は無効にできる。

以上のリード・ライトロック処理要件を実現するために、独自のリード・ライトロック機構を実装する。

* POSIXスレッドライブラリにはリード・ライトロックがあるが、使用しない。
* 限定的な用法で用いることを前提に、回帰ロックやデッドロック検出などの安全性重視の機能は盛り込まず、パフォーマンスを最優先にした設計とする。

リード・ライトロックの動作イメージ：



ライトロック優先動作のイメージ：



メインスレッドがリードロックをスルーする動作のイメージ：



## クラス設計

要件定義に基づいて、リード・ライトロックのクラスを設計する。

リード・ライトロッククラスのクラス図:



## プログラミングイメージ

リード・ライトロックを使用したプログラミングのイメージを示す。

【リードロック】

|  |
| --- |
| //リード・ライトロックオブジェクト  CRWLock s\_rwlock;  //リードロック処理ブロック  {  CRWLockR lock(s\_rwlock);//リードロックオブジェクト（ロック取得）  …処理…  }  //ブロックを抜ける時に、リードロックオブジェクトのデストラクタにより、自動的にロック解放 |

【ライトロック】

|  |
| --- |
| //リード・ライトロックオブジェクト  CRWLock s\_rwlock;  //リードロック処理ブロック  {  CRWLockW lock(s\_rwlock);//ライトロックオブジェクト（ロック取得）  …処理…  }  //ブロックを抜ける時に、ライトロックオブジェクトのデストラクタにより、自動的にロック解放 |

【明示的なロック解放】

|  |
| --- |
| //リードロック処理ブロック  {  CRWLockR lock(s\_rwlock);  …処理…  lock.unlock(); //明示的なロック解放  }  //ブロックを抜けてリードロックオブジェクトのデストラクタが呼び出されても、多重ロック解放にはならない |

【スピンロックカウントの指定】

|  |
| --- |
| //リードロック処理ブロック  {  CRWLockR lock(s\_rwlock, 2000);//デフォルトは1000 ※ライトロックも同様  …処理… |

【メインループ（スレッド）時にロック取得しないリードロック】

|  |
| --- |
| //メインスレッドIDオブジェクト  //※コンストラクタにより、現在のスレッドのIDを自動的に保持  CThreadID s\_mainThreadID("MainThread");  //リード・ライトロックオブジェクト  //※コンストラクタにスレッドIDオブジェクトを渡すことにより、  //　CRWLockR\_AsNecessary() 使用時のスレッドが同じなら、ロック取得をしない  //※スレッドIDを指定しない場合は、インスタンス生成時点のスレッドを対象にする  CRWLock s\_rwlock(s\_mainThreadID);  //共通処理  {  //現在のスレッドIDを取得  CThisThreadID current\_thread\_id;    //リードロック処理ブロック  {  //必要に応じてリードロック取得  //※コンストラクタにスレッドIDオブジェクトを受け渡すことにより、  //　リード・ライトロックオブジェクトが保持しているスレッドIDと一致するなら、  //　ライトロックと衝突がないものと信じて、即時（高速に）リードロックを取得する  //　ライトロックの状態を確認しないで処理するので注意。  //※スレッドIDを指定しない場合は、インスタンス生成時点のスレッドを対象にする  CRWLockR\_AsNecessary lock(s\_rwlock, current\_thread\_id);  …処理… |

【メインループ（スレッド）時にロック取得しないリードロック】※シンプル版

|  |
| --- |
| //リード・ライトロックオブジェクト  CRWLock s\_rwlock;  //共通処理  {  //リードロック処理ブロック  {  //必要に応じてリードロック取得  //※リード・ライトオブジェクト生成時のスレッドと、現在のスレッドで判定  CRWLockR\_AsNecessary lock(s\_rwlock);  …処理… |

【ライトロックを優先にしない】

|  |
| --- |
| //リード・ライトロックオブジェクト  //※ライトロックを優先にしないオプションを指定（デフォルトは優先にする）  CRWLock s\_rwlock(CRWLock::NOT\_WLOCK\_PRIORITIZED); |

### リソースマネージャを通したリソースアクセスのイメージ

リソース一つ一つがリード・ライトロックオブジェクトを持ち、リソースの取得と同時にリードまたはライトロックを行う。明示的にアンロックせずとも、処理ブロックを抜ける際に自動的にアンロックする。

|  |
| --- |
| //リソースマネージャのインスタンス（シングルトン）取得  //※CSingletonUsing<> テンプレートは、インスタンス生成を行わないシングルトンテンプレートクラス  CSingltonUsing<CResMan> res\_man;  //モデルデータのリード処理ブロック  {  //リソースをリソースIDで検索して取得、同時にリードロック  //※リソースIDは、ファイルパスのCRC(32bit)をベースにした64bitのID  CResR<CModel> model(res\_man, res\_id);  //※ライトロックは CResW<>、必要に応じてリードロックは CResR\_AsNecessary<>  //リソース存在チェックの後、処理可能  //※ロック取得待機中にリソースが破棄された可能性もある  if(model.isExist())  {  model->member1(); //リードロック時は、const CModel\* 型の変数として振る舞う（アロー演算子が使える）  model->member2(); //ライトロック時は、CModel\* 型の変数として振る舞う（アロー演算子が使える）  …処理…  }    //明示的なロック解放  //※ロック解放するとリソースにアクセスできなくなる  //※リソース取得に失敗している時は、何もせず処理をスルーする  model.unlock();  }  //明示的なロック解放がなかった場合、処理ブロック終了時に自動的にロック解放  //関数に渡すときは必ず参照渡しにする  void func(CResR<CModel>& model)  {  …処理… |

# 【要件】マルチスレッドで利用する共通処理の最適化

「リソースマネージャ」や「シーンマネージャ」のような共通処理を、マルチスレッドに最適化する。

また、もっと手軽な共通処理にも対応する。例えば「複数のスレッドが共通の集計オブジェクトを利用して処理」といったことを安全に行えるようにする。この時、明示的な共通オブジェクトの生成を行わなくても、各スレッドがクラスにアクセスするだけで、最初のスレッドがアクセスした時点でインスタンスを生成する。極力余計なコードを書かずに済む処理構造とする。

## スレッドセーフなシングルトンパターンの活用

本件は、CallOnce、TLS（スレッドローカルストレージ）を活用した、効果的なシングルトンパターンの事例として説明する。

## 要件定義

本件の具体的な処理要件を定義する。

任意のクラスに対して適用可能な、汎用的なシングルトンパターンのテンプレートクラスを作成し、シングルトンの仕組みを標準化する。

プロキシーパターンの性質を兼ね備え、対象クラスの機能を完全に透過的に利用できるものとする。

あらかじめスタティックなメモリ領域を確保しておき、動的メモリを使用しない。

* ヒープが足りない状況でも確実にインスタンスを生成できる。
* いつ初期化されても、「初期化のタイミングを変えたらヒープメモリが断片化した」といった類いの問題を起こさない。

スタティック領域を使用しつつも、初期化・終了処理のタイミング（順序）を任意にコントロールできるものとする。

マルチスレッドで手軽に扱えるシングルトンとする。

* 初めてアクセスする際に自動的にインスタンスを生成する。
* どんなスレッドが初回アクセスしても安全にインスタンスを生成する。
* 競合して二重に初期化されるようなことがないものとする。

「○○マネージャ」のような特に重要なシングルトンを安全に扱うために、「インスタンスの生成・削除」と「アクセス」の機能を分割可能とする。

* ヘルパークラスを通してのみ扱える動作モードを用意し、通常のシングルトンとして扱えないようにできる。
* インスタンス生成・削除用のヘルパークラスとアクセス専用のヘルパークラスを用意する。
* ヘルパークラスもまたプロキシーであり、通常のシングルトンと同様に操作できる。

非スレッドセーフなクラスを扱うことを考慮し、マルチスレッドの利用を禁止（警告）するモードにも対応する。

* インスタンス生成時のスレッド以外のスレッドからのアクセスを禁止することを可能とする。

シングルトンに対する利用状況を確認するデバッグ機能を標準実装する。

* アクセス数の確認、アクセスしているスレッド数の確認、アクセス中の処理の確認、特定のアクセスに対してブレークする、といった機能をもつ。

## クラス設計

要件定義に基づいて、クラスを設計する。

まずは一般的なシングルトンパターンを示す。

一般的なシングルトンパターンのクラス図:



シングルトンをより使い易くするために、プロキシーパターンと組み合わせたテンプレートクラスを設計する。

スレッドセーフな汎用シングルトンパターン＋プロキシーパターンのクラス図:



これにより、いつアクセスがあってもスレッドセーフなインスタンス生成を行う事ができる。

しかし、このままでは、マネージャ系オブジェクトの生成・終了のタイミング調整がしにくい。その対応のためにと、明示的な生成・終了のためのメソッドをシングルトンクラス本体に追加してしまうと、誰でもそのような操作を行うことができるため、安全性が低下してしまう。

これらの対応として、「生成・終了用」と「アクセス用」の二つのヘルパークラスを追加する。

汎用シングルトンクラス＋ヘルパークラスのクラス図:



## プログラミングイメージ

シングルトンを使用したプログラミングのイメージを示す。

【シングルトン対象クラス】

|  |
| --- |
| //共通処理テストクラス　※継承は必要ないが、幾つかのメンバー定義が必要  class CTest  {  //コンストラクタ／デストラクタを private にするための friend 宣言  friend class CSingleton<CTest>;  public:  //シングルトン設定  static const CSingleton::E\_ATTR SINGLETON\_ATTR = CSingleton::ATTR\_AUTO\_CREATE;//属性：自動生成  static const bool IS\_THREAD\_SAFE = false;//スレッドセーフ：無効　※誤ったスレッドからアクセスしたら  // アサーション違反  static const char\* CLASS\_NAME;//クラス名：MAKE\_SINGLETON\_INSTANCE()マクロで自動設定  public:  //専用処理  void printTest()  {  printf("test\n");  }  private:  //コンストラクタ  CTest()  {  printf("CTest::CTest()\n");  }  //デストラクタ  ~CTest()  {}  }; |

【シングルトン使用処理】

|  |
| --- |
| //通常処理１  void func1()  {  //CTest シングルトン  //※初回のアクセス時にインスタンスが生成され、  //　以後はそのインスタンスが使用される  CSingleton<CTest> test;    //処理  //※アロー演算子でCTestクラスのメンバーにアクセスする  test->printTest();  …処理…  //通常処理２  //※どの処理からでも同じ使い方  void func2()  {  //CTest シングルトン  CSingleton<CTest> test;    //処理  test->printTest();  …処理…  //CTest のインスタンスを直接生成しようとするとコンパイルエラー  CTest o;//NG  CTest\* p = new CTest();//NG |

【シングルトンヘルパークラス使用処理】

|  |
| --- |
| //共通処理テストクラス  class CTest  {  …（略）…  //シングルトン設定  static const CSingleton::E\_ATTR SINGLETON\_ATTR = CSingleton::ATTR\_MANUAL\_CREATE\_AND\_DELETE;//属性：手動生成・破棄  …（略）…  //初期化処理  void gameInitializer()  {  …処理…  //シングルトンイニシャライザーによる、明示的なインスタンス生成  CSingletonInitializer<CTest> testInit("initializer");//処理名を与える  testInit.initialize();  …処理…  //通常処理１  void func1()  {  //CTest シングルトン使用オブジェクト  //※このとき、インスタンスが生成されていなければアサーション違反  CSingletonUsing<CTest> test("func1");//処理名を与える    //処理  //※通常のシングルトンオブジェクトと同様に、アロー演算子でCTestクラスのメンバーにアクセスする  test->printTest();  …処理…    //通常のシングルトンオブジェクトとして使おうとするとアサーション違反  CSingleton<CTest> test\_direct;//NG    //トラップ  //※特定の処理名、スレッド名のアクセスがあったらブレーク（AND条件）  test.setDebugTrapName("funcX");  test.setDebugTrapThreadName("ThreadX");    //現在アクセス中の処理の処理名を表示  //※イニシャライザーの処理名も含まれる  test.printUsingList();  //出力イメージ：  // ----------------------------------------  //Using List: [CTest]  // "func1" on "ThreadB"(0xcfa88983)  // "CGmaeLoop::test" on "MainThread"(0xc9281f44)  // "baseFunc" on "ThreadB"(0xcfa88983)  // "CGameLoop::main" on "MainThread"(0xc9281f44)  // "initializer" on "MainThread"(0xc9281f44)  //(num=5, max=9)  //----------------------------------------    //デバッグ情報を表示  test.printDebugInfo();  //出力イメージ：  //----------------------------------------  //Debug Info: [CTest]  // ClassAttribute = INITIALIZER\_NEEDED  // ClassIsThreadSafe = IS\_THREAD\_SAFE  // ClassisProtected = IS\_PROTECTED  // ClassisCreated = IS\_CREATED  // RefCount = 5 (max=9)  // RefCountOnThisThread = 2  // ThreadCount = 2 (max=4)  // CreatedThread = "MainThread" (ID=0xc9281f44)  // InitializerNname = "initializer"  // DebugTrap = "funcX" (Thread="ThreadX")  //----------------------------------------  …処理…  //終了処理  void gameFinalizer()  {  …処理…  //シングルトンイニシャライザーによる、明示的なインスタンス破棄  CSingletonInitializer<CTest> testFinal("finalizer");//処理名を与える  testFinal.finalize();  …処理… |

# 処理実装サンプル

リード・ライトロッククラスとシングルトンクラスの実装サンプルを示す。

かなり長いが、Visual C++ 2013 で完全に動作する状態である。

C++11の仕様に依存したコードである。

アトミック型、スレッドID、右辺値参照（ムーブコンストラクタ）、ミューテックス（CallOnce）、ラムダ式、可変長テンプレート引数、static\_assert を使用している。サンプルプログラム中でこれらを使用している箇所は赤字で表記する。ほか、有効なプログラミングテクニックの説明も随所に記載している。

## 準備①：インクルードファイルと基本マクロ

アサーション、静的アサーション、ブレークポイント、TLSのマクロを用意し、以降のプログラムで使用する。

**【インクルードファイル】**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  //C++11ライブラリ  #include <thread>//スレッド ※スレッドIDとスリープ用  #include <mutex>//ミューテックス ※CallOnce用  #include <atomic>//アトミック型  #include <chrono>//時間計測用 |

**【基本マクロ】**

|  |
| --- |
| //--------------------------------------------------------------------------------  //基本マクロ  //ブレークポイント  #include <windows.h>//ブレークポイント用  #define BREAK\_POINT() DebugBreak()  //デバッグプリント  #define DEBUG\_PRINT(msg, ...) printf(msg, \_\_VA\_ARGS\_\_);  #define DEBUG\_FPRINT(fp, msg, ...) fprintf(fp, msg, \_\_VA\_ARGS\_\_);  #define DEBUG\_FLUSH() fflush(stdout)  #define DEBUG\_FFLUSH(fp) fflush(fp)  //#define DEBUG\_PRINT(msg, ...)  //#define DEBUG\_FPRINT(fp, msg, ...)  //#define DEBUG\_FLUSH()  //#define DEBUG\_FFLUSH(fp) fflush(fp)  //文字列化マクロ  #define TO\_STRING(s) #s  #define TO\_STRING\_EX(s) TO\_STRING(s)  //アサーション  #define ASSERT(expr, msg, ...) \  if(!(expr)) \  { \  fprintf(stderr, "Assertion failed! : " #expr "\n" \  " " \_\_FILE\_\_ "(" TO\_STRING\_EX(\_\_LINE\_\_) ")\n" \  " " msg "\n", \_\_VA\_ARGS\_\_); \  BREAK\_POINT(); \  }  //#define ASSERT(expr, msg, ...)//削除用  //#include <assert.h>//assert用  //#define ASSERT(expr, msg, ...) assert(expr)//VC++標準版  //静的アサーション  //※C++11仕様  //※日本語使用不可  #define STATIC\_ASSERT(expr, msg) static\_assert(expr, msg)  //#define STATIC\_ASSERT(expr, msg)//削除用  //スレッドローカルストレージ修飾子  //※C++11仕様偽装（VC++2013では未対応につき）  //※中身はWindows仕様  #define thread\_local \_\_declspec(thread) |

## 準備②：スレッドIDクラス

スレッドIDの保持、受け渡し、比較を行うためのクラス。スレッド名も同時に扱う。

TLSを利用して、スレッドIDの取得を高速化している。

**【スレッドIDクラス】**

|  |
| --- |
| //----------------------------------------  //スレッドIDクラス  //※TLSを活用して高速化  //スレッドID型  //typedef std::thread::id THREAD\_ID;//C++11 ※この型では扱わず、ハッシュ値を使用する  typedef std::size\_t THREAD\_ID;//C++11  static const THREAD\_ID INVALID\_THREAD\_ID = 0xffffffff;//無効なスレッドID  //現在のスレッドID取得関数  //inline THREAD\_ID GetThisThreadID(){ return std::this\_thread::get\_id(); } //C++11  inline THREAD\_ID GetThisThreadID(){ return std::this\_thread::get\_id().hash(); } //C++11  //スレッドIDクラス  class CThreadID  {  public:  //アクセッサ  const THREAD\_ID getID() const { return m\_threadId; }//スレッドIDを取得  const char\* getName() const { return m\_threadName; }//スレッド名を取得  public:  //アクセッサ（static）  static THREAD\_ID getThisID(){ return m\_thisThreadID; }//現在のスレッドのスレッドIDを取得  static const char\* getThisName(){ return m\_thisThreadName; }//現在のスレッドのスレッド名を取得  public:  //メソッド  bool isThisThread() const { return m\_threadId == getThisID(); }//現在のスレッドと同じスレッドか判定  private:  //メソッド(static)  static void setThisThread()//現在のスレッドのスレッドIDをセット  {  if (m\_thisThreadID == INVALID\_THREAD\_ID)  m\_thisThreadID = GetThisThreadID();  }  static void resetThisThread(const char\* name)//現在のスレッドのスレッドIDをリセット  {  m\_thisThreadID = GetThisThreadID();  m\_thisThreadName = name;  }  public:  //オペレータ（許可）  bool operator==(const CThreadID& o) const { return m\_threadId == o.getID(); }//ID一致判定  bool operator!=(const CThreadID& o) const { return m\_threadId != o.getID(); }//ID不一致判定  bool operator==(const THREAD\_ID& id) const { return m\_threadId == id; }//ID一致判定  bool operator!=(const THREAD\_ID& id) const { return m\_threadId != id; }//ID不一致判定  CThreadID& operator=(const CThreadID& o) //コピー演算子  {  m\_threadId = o.m\_threadId;  m\_threadName = o.m\_threadName;  return \*this;  }  private:  //オペレータ（禁止）  CThreadID& operator=(const THREAD\_ID& id) { return \*this; }//コピー演算子（禁止）  public:  //コピーコンストラクタ（許可）  explicit CThreadID(const CThreadID& o) :  m\_threadId(o.m\_threadId),  m\_threadName(o.m\_threadName)  {  }  private:  //コピーコンストラクタ（禁止）  explicit CThreadID(const THREAD\_ID& id){}  public:  //コンストラクタ  //※スレッド名を指定し、内部で現在のスレッドIDを取得して保持  //※TLSにも記録  CThreadID(const char\* name)  {  resetThisThread(name);  m\_threadId = m\_thisThreadID;  m\_threadName = m\_thisThreadName;  }  //デフォルトコンストラクタ  //※既にTLSに記録済みのスレッドID（と名前）を取得  CThreadID()  {  setThisThread();  m\_threadId = m\_thisThreadID;  m\_threadName = m\_thisThreadName;  }  private:  //フィールド  THREAD\_ID m\_threadId;//スレッドID（オブジェクトに保存する値）  const char\* m\_threadName;//スレッド名（オブジェクトに保存する値）  static thread\_local THREAD\_ID m\_thisThreadID;//現在のスレッドのスレッドID(TLS)  static thread\_local const char\* m\_thisThreadName;//現在のスレッド名(TLS)  };  //static変数のインスタンス化  thread\_local THREAD\_ID CThreadID::m\_thisThreadID = INVALID\_THREAD\_ID;//スレッドID(TLS)  thread\_local const char\* CThreadID::m\_thisThreadName = nullptr;//スレッド名(TLS) |

## 準備③：軽量スピンロッククラス

「std::atomic\_flag」を利用した軽量なスピンロッククラスを用意する。

**【軽量スピンロッククラス】**

|  |
| --- |
| //--------------------------------------------------------------------------------  //軽量スピンロック  //----------------------------------------  //軽量スピンロック  //※手軽に使えるスピンロック  //※一定回数のスリープごとにスリープ（コンテキストスイッチ）を行う  //※容量は4バイト(std::atomic\_flag一つ分のサイズ)  //※プログラミング上の安全性は低いので気がるに使うべきではない  //　　⇒ロック取得状態を確認せずにアンロックする  #define SPIN\_LOCK\_USE\_ATOMIC\_FLAG//std::atomic\_flag版（高速）  //#define SPIN\_LOCK\_USE\_ATOMIC\_BOOL//std::atomic\_bool版（軽量）  class CSpinLock  {  public:  //定数  static const int DEFAULT\_SPIN\_COUNT = 1000;//スピンロックカウントのデフォルト値  public:  //ロック取得  void lock(const int spin\_count = DEFAULT\_SPIN\_COUNT)  {  int spin\_count\_now = 0;  #ifdef SPIN\_LOCK\_USE\_ATOMIC\_FLAG  while (m\_lock.test\_and\_set())//std::atomic\_flag版（高速）  {  #else//SPIN\_LOCK\_USE\_ATOMIC\_FLAG  bool prev = false;  while (m\_lock.compare\_exchange\_weak(prev, true))//std::atomic\_bool版（軽量）  {  prev = false;  #endif//SPIN\_LOCK\_USE\_ATOMIC\_FLAG  if (spin\_count == 0 || ++spin\_count\_now % spin\_count == 0)  std::this\_thread::sleep\_for(std::chrono::milliseconds(0));//スリープ（コンテキストスイッチ）  }  }  //ロック解放  void unlock()  {  #ifdef SPIN\_LOCK\_USE\_ATOMIC\_FLAG  m\_lock.clear();//std::atomic\_flag版（高速）  #else//SPIN\_LOCK\_USE\_ATOMIC\_FLAG  m\_lock.store(false);//std::atomic\_bool版（軽量）  #endif//SPIN\_LOCK\_USE\_ATOMIC\_FLAG  }  public:  //コンストラクタ  CSpinLock()  {  }  //デストラクタ  ~CSpinLock()  {  }  private:  //フィールド  #ifdef SPIN\_LOCK\_USE\_ATOMIC\_FLAG  std::atomic\_flag m\_lock;//ロック用フラグ（高速）  #else//SPIN\_LOCK\_USE\_ATOMIC\_FLAG  std::atomic\_bool m\_lock;//ロック用フラグ（軽量）  #endif//SPIN\_LOCK\_USE\_ATOMIC\_FLAG  }; |

## 準備④：メモリブロック確保クラス

シングルトンクラス内でシングルトンにアクセス中の処理を記録する処理を行っている。動的なメモリ割り当てを使用せず、上限を決めて扱うものとする。常に一定の構造体のメモリ割り当てを行うので、固定サイズの領域が固定の数用意されたメモリ割り当て処理を使用する。

この処理のために、汎用のメモリブロック確保処理を用いる。

テンプレート引数で指定した個数とサイズの領域を持つ単純なクラス。

配置new/delete と可変長テンプレート引数を組み合わせた効果的な処理を行っている。

**【固定メモリブロックアロケータクラス】**

|  |
| --- |
| //--------------------------------------------------------------------------------  //固定メモリブロックアロケータクラス  //※現状は可読性重視だが、実際にはテンプレートのインスタンス化による  //　プログラムサイズの肥大を考慮し、非テンプレートの共通処理を  //　切り出す必要がある  //----------------------------------------  //クラス宣言  template<std::size\_t N, std::size\_t S>  class CBlockAllocator;  //----------------------------------------  //固定メモリブロックアロケータクラス専用配置new/delete処理  //※クラス内で使用するためのものなので、直接使用は禁止  //※クラス内でdeleteすることで、デストラクタの呼び出しにも対応  //配置new  template<std::size\_t N, std::size\_t S>  void\* operator new(const std::size\_t size, CBlockAllocator<N, S>& allocator){ return allocator.alloc(size); }  //配置delete  template<std::size\_t N, std::size\_t S>  void operator delete(void\* p, CBlockAllocator<N, S>& allocator){ allocator.free(p); }  //----------------------------------------  //固定メモリブロックアロケータクラス  //※スレッドセーフ対応  template<std::size\_t N, std::size\_t S>  class CBlockAllocator  {  public:  //型宣言  typedef unsigned char byte;//バッファ用  typedef unsigned int b32;//フラグ用  public:  //定数  static const std::size\_t BLOCKS\_NUM = N;//ブロック数  static const std::size\_t BLOCK\_SIZE = S;//メモリブロックサイズ  static const std::size\_t FLAG\_SIZE = ((N + 31) >> 5);//フラグサイズ（１サイズで32ビットのフラグ）  static const std::size\_t FLAG\_SURPLUS\_BITS = N % 32;//余剰フラグ数（32の倍数からはみ出したフラグの数）  private:  //メソッド  //メモリ確保状態リセット  void reset()  {  //ロック  m\_lock.lock();  //ゼロクリア  memset(m\_used, 0, sizeof(m\_used));  //ブロック数の範囲外のフラグは最初から立てておく  if (FLAG\_SURPLUS\_BITS > 0)  {  b32 parmanent = 0xffffffff >> FLAG\_SURPLUS\_BITS;  parmanent <<= FLAG\_SURPLUS\_BITS;  m\_used[FLAG\_SIZE - 1] = parmanent;  }  //ロック解放  m\_lock.unlock();  }  //メモリブロック確保  //※使用中フラグの空きを検索してフラグを更新し、  //　確保したインデックスを返す  int assign()  {  //ロック  m\_lock.lock();  //確保済みインデックス準備  int index = -1;//初期状態は失敗状態  //空きフラグ検索  b32\* used\_p = m\_used;  int bit\_no = 0;  for (int arr\_idx = 0; arr\_idx < FLAG\_SIZE; ++arr\_idx, ++used\_p)  {  //32bitごとの空き判定  b32 bits\_now = \*used\_p;  if (bits\_now != 0xffffffff)  {  //32bitのフラグのどこかに空きがあるので、  //最初に空いているフラグ（ビット）を検索  b32 bits = bits\_now;  if ((bits & 0xffff) == 0xffff){ bit\_no += 16; bits >>= 16; }  //下位16ビット判定（空きがなければ16ビットシフト）  if ((bits & 0xff) == 0xff){ bit\_no += 8; bits >>= 8; }  //下位8ビット判定（空きがなければ8ビットシフト）  if ((bits & 0xf) == 0xf){ bit\_no += 4; bits >>= 4; }  //下位4ビット判定（空きがなければ4ビットシフト）  if ((bits & 0x3) == 0x3){ bit\_no += 2; bits >>= 2; }  //下位2ビット判定（空きがなければ2ビットシフト）  if ((bits & 0x1) == 0x1){ ++bit\_no; } //下位1ビット判定（空きがなければ上位1ビットで確定）  bits\_now |= (1 << bit\_no);//論理和用のビット情報に変換  \*used\_p = bits\_now;//論理和  index = arr\_idx \* 32 + bit\_no;//メモリブロックのインデックス算出  //確保成功  break;  }  }  //ロック解放  m\_lock.unlock();  //終了  return index;  }  //メモリブロック解放  //※指定のインデックスの使用中フラグをリセット  void release(const int index)  {  //インデックスの範囲チェック  if (index < 0 || index >= BLOCKS\_NUM)  return;  //ロック  m\_lock.lock();  //フラグ解放  const int arr\_idx = index >> 5;//使用中フラグの配列番号  const int bit\_no = index & 31;//ビット番号  m\_used[arr\_idx] &= ~(1 << bit\_no);//論理積  //ロック解放  m\_lock.unlock();  }  public:  //メモリ確保  void\* alloc(const std::size\_t size)  {  //【アサーション】要求サイズがブロックサイズを超える場合は即時確保失敗  ASSERT(size <= BLOCK\_SIZE, "CBlockAllocator::alloc(%d) cannot allocate. Size must has been under %d.",  size, BLOCK\_SIZE);  if (size > BLOCK\_SIZE)  {  return nullptr;  }  //空きブロックを確保して返す  const int index = assign();  //【アサーション】全ブロック使用中につき、確保失敗  ASSERT(index >= 0, "CBlockAllocator::alloc(%d) cannot allocate. Buffer is full. (num of blocks is %d)",  size, BLOCKS\_NUM);  //確保したメモリを返す  return index < 0 ? nullptr : &m\_buff[index];  }  //メモリ解放  void free(void \* p)  {  //【アサーション】nullptr時は即時解放失敗  ASSERT(p != nullptr, "CBlockAllocator::free() cannot free. Pointer is null.");  if (!p)  return;  //ポインタからインデックスを算出  const byte\* top\_p = reinterpret\_cast<byte\*>(m\_buff);//バッファの先頭ポインタ  const byte\* target\_p = reinterpret\_cast<byte\*>(p);//指定ポインタ  const int diff = (target\_p - top\_p);//ポインタの引き算で差のバイト数算出  const int index = (target\_p - top\_p) / BLOCK\_SIZE;//ブロックサイズで割ってインデックス算出  //【アサーション】ポインタが各ブロックの先頭を指しているかチェック  //　　　　　　　　⇒多重継承とキャストしているとずれることがるのでこの問題は無視して解放してしまう  //ASSERT(diff % BLOCK\_SIZE == 0, "CBlockAllocator::free() cannot free. Pointer is illegal."); //【アサーション】メモリバッファの範囲外なら処理失敗（release関数内で失敗するのでそのまま実行）  ASSERT(index >= 0 && index < BLOCKS\_NUM, "CBlockAllocator::free() cannot free. Pointer is different.");  //算出したインデックスでメモリ解放  release(index);  }  //コンストラクタ呼び出し機能付きメモリ確保  //※C++11の可変長テンプレートパラメータを活用  template<class T, typename... Tx>  T\* create(Tx... nx)  {  return new(\*this) T(nx...);  }  //デストラクタ呼び出し機能付きメモリ解放  //※解放後、ポインタに nullptr をセットする  template<class T>  void remove(T\*& p)  {  p->~T();//明示的なデストラクタ呼び出し（デストラクタ未定義のクラスでも問題なし）  operator delete(p, \*this);//配置delete呼び出し  p = nullptr;//ポインタにはnullptrをセット  }  public:  //コンストラクタ  CBlockAllocator()  {  //使用中フラグリセット  reset();  }  //デストラクタ  ~CBlockAllocator()  {}  private:  //フィールド  byte m\_buff[BLOCKS\_NUM][BLOCK\_SIZE];//ブロックバッファ  b32 m\_used[FLAG\_SIZE];//使用中フラグ  CSpinLock m\_lock;//ロック  }; |

## リード・ライトロッククラス

前述の要件定義に基づく実装例を示す。

クラス内クラスを定義している点に注意。

また、try\_lockの仕組みが未実装。

**【リード・ライトロッククラス】**

|  |
| --- |
| //--------------------------------------------------------------------------------  //リード・ライトロッククラス  //※容量節約のために、POSIXスレッドライブラリ版のように、現在のスレッドのロック状態を保持しない  //※必ずロッククラス CRWLock::LockR, CRWLock::LockR\_AsNecessary, CRWLock::LockW を使用し、  //　そこに現在のロック状態を保持する  //----------------------------------------  //リード・ライトロッククラス  class CRWLock  {  public:  //定数  enum E\_WLOCK\_PRIORITY//ライトロック優先度  {  WLOCK\_PRIORITIZED,//ライトロック優先  NOT\_WLOCK\_PRIORITIZED,//ライトロック優先しない  ALL\_WLOCK//全てライトロックにする（リードロックも内部的にライトロックになる）  };  public:  //----------------------------------------  //【クラス内クラス】クラス宣言  class RLock;  class RLockAsNecessary;  class WLock;  //----------------------------------------  //【クラス内クラス】ロッククラス　※継承専用  class Lock  {  friend class CRWLock;  public:  //アクセッサ  bool isWriteLock() const { return m\_isWriteLock; }//ライトロックモードか？  bool isUnlocked() const { return m\_isUnlocked; }//現在アンロック状態か？（ロック状態なのが普通）  public:  //メソッド  //明示的なリードロック  //※明示的なロック解放後用メソッド  //※通常はコンストラクタでロックするので使用しない  //※コンストラクタ時と同じ挙動になるため、  //　「必要に応じてリードロック」を行っていたなら、再びその挙動となる  void rlock(const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT)  {  if (!m\_isUnlocked)  return;  m\_lock.rlock(spin\_count, m\_ignoreThreadId);  m\_isWriteLock = false;  m\_isUnlocked = false;  }  //明示的なライトロック  //※明示的なロック解放後用メソッド  //※通常はコンストラクタでロックするので使用しない  void wlock(const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT)  {  if (!m\_isUnlocked)  return;  m\_lock.wlock(spin\_count);  m\_isWriteLock = true;  m\_isUnlocked = false;  }  //明示的なアンロック  //※通常はデストラクタでアンロックするので使用しない  void unlock()  {  if (m\_isUnlocked)  return;  if (m\_isWriteLock)  m\_lock.wunlock();  else  m\_lock.runlock();  m\_isUnlocked = true;  }  private:  //オペレータ（禁止）  Lock& operator=(const Lock& o){ return \*this; }//コピー演算子  private:  //コピーコンストラクタ（禁止）  explicit Lock(const Lock& o) :m\_lock(o.m\_lock){}  public:  //ムーブコンストラクタ  inline Lock(Lock&& lock) :  m\_lock(lock.m\_lock),  m\_ignoreThreadId(lock.m\_ignoreThreadId),  m\_isWriteLock(lock.m\_isWriteLock),  m\_isUnlocked(lock.m\_isUnlocked)  {  //移動元のロックオブジェクトはアンロック扱いにして、  //デストラクタでアンロックしてしまうことを防ぐ  lock.m\_isUnlocked = true;  }  protected:  //コンストラクタ　※各種ロッククラスからのみ使用  //リードロック用コンストラクタ  inline Lock(const RLock\*, CRWLock& lock, const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  m\_lock(lock),  m\_ignoreThreadId(INVALID\_THREAD\_ID),  m\_isWriteLock(false),  m\_isUnlocked(false)  {  //リードロック  m\_lock.rlock(spin\_count, m\_ignoreThreadId);  }  //必要に応じてリードロック用コンストラクタ  //※リードロックをスキップするためのスレッドIDを受け渡す  //※リード・ライトロックオブジェクト生成時のスレッドと一致している時にリードロックしない  inline Lock(const RLockAsNecessary\*, CRWLock& lock, const CThreadID& ignore\_thread\_id,  const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  m\_lock(lock),  m\_ignoreThreadId(ignore\_thread\_id.getID()),  m\_isWriteLock(false),  m\_isUnlocked(false)  {  //必要に応じてリードロック  m\_lock.rlock(spin\_count, m\_ignoreThreadId);  }  inline Lock(const RLockAsNecessary\*, CRWLock& lock, const THREAD\_ID ignore\_thread\_id,  const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  m\_lock(lock),  m\_ignoreThreadId(ignore\_thread\_id),  m\_isWriteLock(false),  m\_isUnlocked(false)  {  //必要に応じてリードロック  m\_lock.rlock(spin\_count, m\_ignoreThreadId);  }  //ライトロック用コンストラクタ  inline Lock(const WLock\*, CRWLock& lock, const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  m\_lock(lock),  m\_ignoreThreadId(INVALID\_THREAD\_ID),  m\_isWriteLock(true),  m\_isUnlocked(false)  {  //ライトロック  m\_lock.wlock(spin\_count);  }  public:  //デストラクタ  //※アンロック  ~Lock()  {  unlock();  }  private:  //フィールド  CRWLock& m\_lock;//リード・ライトオブジェクトの参照  THREAD\_ID m\_ignoreThreadId;//リードロックを無視するスレッドID  bool m\_isWriteLock;//ライトロックモードか？  bool m\_isUnlocked;//アンロック状態か？  };  public:  //----------------------------------------  //【クラス内クラス】リードロッククラス  class RLock : public Lock  {  private:  //オペレータ（禁止）  RLock& operator=(const RLock& o){ return \*this; }//コピー演算子  private:  //コピーコンストラクタ（禁止）  explicit RLock(const Lock& o) :Lock(o){}  public:  //ムーブコンストラクタ  inline RLock(RLock&& lock) :  Lock(std::move(lock))  {}  public:  //コンストラクタ  inline RLock(CRWLock& lock, const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  Lock(this, lock, spin\_count)  {}  //デストラクタ  inline ~RLock()  {}  };  //----------------------------------------  //【クラス内クラス】必要に応じてリードロッククラス  class RLockAsNecessary : public Lock  {  private:  //オペレータ（禁止）  RLockAsNecessary& operator=(const RLockAsNecessary& o){ return \*this; }//コピー演算子  private:  //コピーコンストラクタ（禁止）  explicit RLockAsNecessary(const RLockAsNecessary& o) :Lock(o){}  public:  //ムーブコンストラクタ  inline RLockAsNecessary(RLockAsNecessary&& lock) :  Lock(std::move(lock))  {}  public:  //コンストラクタ  //※リードロックをスキップするためのスレッドIDを受け渡す  //※リード・ライトロックオブジェクト生成時のスレッドと一致している時にリードロックしない  inline RLockAsNecessary(CRWLock& lock, const CThreadID& ignore\_thread\_id,  const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  Lock(this, lock, ignore\_thread\_id, spin\_count)  {}  inline RLockAsNecessary(CRWLock& lock, const THREAD\_ID ignore\_thread\_id,  const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  Lock(this, lock, ignore\_thread\_id, spin\_count)  {}  //※スレッドIDを渡さず、現在のスレッドIDを自動的に適用する場合のコンストラクタ  inline RLockAsNecessary(CRWLock& lock, const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  Lock(this, lock, CThreadID::getThisID(), spin\_count)  {}  //デストラクタ  inline ~RLockAsNecessary()  {}  };  //----------------------------------------  //【クラス内クラス】ライトロッククラス  class WLock : public Lock  {  private:  //オペレータ（禁止）  WLock& operator=(const WLock& lock){ return \*this; }//コピー演算子  private:  //コピーコンストラクタ（禁止）  explicit WLock(const WLock& lock) :Lock(lock){}  public:  //ムーブコンストラクタ  inline WLock(WLock&& lock) :  Lock(std::move(lock))  {}  public:  //コンストラクタ  inline WLock(CRWLock& lock, const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT) :  Lock(this, lock, spin\_count)  {}  //デストラクタ  inline ~WLock()  {}  };  //----------------------------------------  //【クラス内クラス】（以上で終了）    public:  //【ロックオブジェクトを返すロックメソッド】  //※右辺値参照を使用することにより、「コピーコンストラクタ」と  //　区別して「ムーブコンストラクタ」を記述できる  //※これを利用して、関数の内部で作成したオブジェクトの内容を  //　呼び出し元のオブジェクトに移動する。  //　ムーブコンストラクタでは、移動元オブジェクトを  //　アンロック済み扱いにしているるので、関数終了時に  //　ローカルオブジェクトのデストラクタが呼び出されても、  //　アンロックしてしまうことがない。    //【ロックオブジェクトを返すロックメソッド】リードロック  RLock rLock(const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT)  {  RLock lock(\*this, spin\_count);  return std::move(lock);  }  //【ロックオブジェクトを返すロックメソッド】必要に応じてリードロック  RLockAsNecessary rLockAsNecessary(const CThreadID& ignore\_thread\_id,  const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT)  {  RLockAsNecessary lock(\*this, ignore\_thread\_id, spin\_count);  return std::move(lock);  }  RLockAsNecessary rLockAsNecessary(const THREAD\_ID ignore\_thread\_id,  const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT)  {  RLockAsNecessary lock(\*this, ignore\_thread\_id, spin\_count);  return std::move(lock);  }  RLockAsNecessary rLockAsNecessary(const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT)  {  RLockAsNecessary lock(\*this, spin\_count);  return std::move(lock);  }  //【ロックオブジェクトを返すロックメソッド】ライトロック  WLock wLock(const int spin\_count = CSpinLock::DEFAULT\_SPIN\_COUNT)  {  WLock lock(\*this, spin\_count);  return std::move(lock);  }  private:  //メソッド　※ロッククラスからのみ使用  //リードロック取得  //※「必要に応じてリードロック」の機能を兼ねる  void rlock(const int spin\_count, const THREAD\_ID ignore\_thread\_id)  {  //全てライトロックにするモード用処理  if (m\_wlockPrioritized == ALL\_WLOCK)  {  wlock(spin\_count);  return;  }  //リードロックスキップチェック  //※リード・ライトロックオブジェクトインスタンス生成時、および、リードロック時に、  //　共にスレッドIDが指定されていた場合、かつ、それが同じIDであった場合、  //　リードロックをスキップする  if (ignore\_thread\_id != INVALID\_THREAD\_ID && m\_ignoreThreadId == ignore\_thread\_id)  //&& m\_writeLock.load() == false)  //↑ライトロック状態はチェックしない  //※プログラムを信頼して高速化を徹底する  //※仮にこの判定を行っても、m\_lock で保護しない  //　限りは結局不確実なブロックになる  {  m\_readLock.fetch\_add(1);  return;  }  //リードロック予約カウントアップ  //※制御上は必要ないが、問題追跡時の参考用  m\_readLockReserved.fetch\_add(1);  //リードロック待機ループ  while (1)  {  //内部変数更新ロック取得  m\_lock.lock(spin\_count);  //内部変数更新ロックを取得できたので、ライトロックの状態をチェック  if (m\_writeLock.load() == false &&//ライトロック状態ではない  (m\_wlockPrioritized != WLOCK\_PRIORITIZED ||//ライトロック優先モードではない  m\_wlockPrioritized == WLOCK\_PRIORITIZED && m\_writeLockReserved.load() == 0))  //ライトロック優先モードなら、  //ライトロック予約がないこともチェック  {  //リードロックOK  m\_readLock.fetch\_add(1);//リードロックカウントアップ  m\_lock.unlock();//内部変数更新ロック解放  break;  }  //内部変数更新ロック解放  //※リードロックが取得できるまで再び待機  m\_lock.unlock();//内部変数更新ロック解放  }  //リードロック予約カウントダウン  //※制御上は必要ないが、問題追跡時の参考用  m\_readLockReserved.fetch\_sub(1);  }  //ライトロック取得  void wlock(const int spin\_count)  {  //ライトロック予約カウントアップ  m\_writeLockReserved.fetch\_add(1);  //ライトロック待機ループ  while (1)  {  //内部変数更新ロック取得  m\_lock.lock(spin\_count);  //内部変数更新ロックを取得できたので、リードロックとライトロックの状態をチェック  if (m\_readLock.load() == 0 && m\_writeLock.load() == false)  {  //ライトロックOK  m\_writeLock.store(true);//ライトロックON  m\_lock.unlock();//内部変数更新ロック解放  break;  }  //内部変数更新ロック解放  //※ライトロックが取得できるまで再び待機  m\_lock.unlock();//内部変数更新ロック解放  }  //ライトロック予約カウントダウン  m\_writeLockReserved.fetch\_sub(1);  }  //リードロック解放  void runlock()  {  //全てライトロックにするモード用処理  if (m\_wlockPrioritized == ALL\_WLOCK)  {  wunlock();  return;  }  //リードロックカウントダウン  m\_readLock.fetch\_sub(1);  }  //ライトロック解放  void wunlock()  {  //ライトロックOFF  m\_writeLock.store(false);  }  public:  //アクセッサ  THREAD\_ID getIgnoreThreadID() const { return m\_ignoreThreadId; }//「必要に応じてリードロック」用のスレッドID  void setIgnoreThreadID(const THREAD\_ID thread\_id) { m\_ignoreThreadId = thread\_id; }  //「必要に応じてリードロック」用のスレッドIDを更新  int getReadLockReserved() const { return m\_readLockReserved.load(); }//リードロック予約カウンタ  // ※制御上は必要ないが、問題追跡時の参考用  int getReadLock() const { return m\_readLock.load(); }//リードロックカウンタ  int getWriteLockReserved() const { return m\_writeLockReserved.load(); }//ライトロック予約カウンタ  bool getWriteLock() const { return m\_writeLock.load(); }//ライトロックフラグ  E\_WLOCK\_PRIORITY getWlockPrioritized() const { return m\_wlockPrioritized; }//ライトロック優先度  public:  //コンストラクタ  CRWLock(const THREAD\_ID ignore\_thread\_id, const E\_WLOCK\_PRIORITY wlock\_prioritized) :  m\_ignoreThreadId(ignore\_thread\_id),  m\_readLockReserved(0),  m\_readLock(0),  m\_writeLockReserved(0),  m\_writeLock(false),  m\_wlockPrioritized(wlock\_prioritized)  {}  CRWLock(const CThreadID& ignore\_thread\_id, const E\_WLOCK\_PRIORITY wlock\_prioritized) :  CRWLock(ignore\_thread\_id.getID(), wlock\_prioritized)  {}  CRWLock(const CThreadID& ignore\_thread\_id) :  CRWLock(ignore\_thread\_id.getID(), WLOCK\_PRIORITIZED)  {}  CRWLock(const THREAD\_ID ignore\_thread\_id) :  CRWLock(ignore\_thread\_id, WLOCK\_PRIORITIZED)  {}  CRWLock(const E\_WLOCK\_PRIORITY wlock\_prioritized) :  CRWLock(CThreadID::getThisID(), wlock\_prioritized)  {}  CRWLock() :  CRWLock(CThreadID::getThisID(), WLOCK\_PRIORITIZED)  {}  //デストラクタ  ~CRWLock()  {}  private:  //フィールド  THREAD\_ID m\_ignoreThreadId;//「必要に応じてリードロック」用のスレッドID  std::atomic<int> m\_readLockReserved;//リードロック予約カウンタ ※制御上は必要ないが、問題追跡時の参考用  std::atomic<int> m\_readLock;//リードロックカウンタ  std::atomic<int> m\_writeLockReserved;//ライトロック予約カウンタ  std::atomic<bool> m\_writeLock;//ライトロックフラグ  CSpinLock m\_lock;//内部変数更新用ロックフラグ  E\_WLOCK\_PRIORITY m\_wlockPrioritized;//ライトロック優先度  }; |

**【クラス内クラスのエイリアス】**

|  |
| --- |
| //「リードロッククラス」「必要に応じてリードロッククラス」「ライトロッククラス」の別名を設定  typedef CRWLock::RLock CRWLockR;  typedef CRWLock::RLockAsNecessary CRWLockR\_AsNecessary;  typedef CRWLock::WLock CRWLockW;  //【C++11スタイル】「リードロッククラス」「必要に応じてリードロッククラス」「ライトロッククラス」の別名を設定  //using CRWLockR = CRWLock::RLock;  //using CRWLockR\_AsNecessary = CRWLock::RLockAsNecessary;  //using CRWLockW = CRWLock::WLock; |

## リード・ライトロッククラスの使用サンプル

この自作リード・ライトクラスを使用し、別紙の「マルチスレッドプログラミングの基礎」の「POSIXライブラリ版リード・ライトロック」で示したサンプルと同じプログラムを実行する。

ここまでに作成したスレッドIDクラス、リード・ライトロッククラスを使用している箇所を赤字で示す。

**【リード・ライトロックを使用したサンプルプログラム】**

|  |
| --- |
| #include <random> //乱数生成用  //メインスレッドID  static CThreadID s\_mainThread("MainThread");  //リード・ライトロックオブジェクト  static CRWLock s\_lock;//【デフォルト】現在のスレッドに基づく、ライトロックを優先にする  //static CRWLock s\_lock(CRWLock::NOT\_WLOCK\_PRIORITIZED);//現在オンスレッドに基づく、ライトロックを優先にしない  //static CRWLock s\_lock(s\_mainThread);//スレッドIDを明示的に渡す  //static CRWLock s\_lock(s\_mainThread, CRWLock::NOT\_WLOCK\_PRIORITIZED);//上記二つの組み合わせ  //static CRWLock s\_lock(CRWLock::ALL\_WLOCK);//リードロックがライトロックとして振る舞う  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  thread\_local int s\_tlsData = 0;  //書き込みスレッド  void threadFuncW(const char\* name)  {  //開始  printf("- begin:(W)%s -\n", name);  fflush(stdout);  //乱数  std::random\_device seed\_gen;  std::mt19937 rnd(seed\_gen());  std::uniform\_int\_distribution<int> sleep\_time(0, 499);  //若干ランダムでスリープ（0～499 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //処理  for (int i = 0; i < 3; ++i)  {  //ライトロック取得  CRWLockW lock(s\_lock);  //データ表示（前）  printf("(W)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(W)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //ライトロック解放  lock.unlock();  //若干ランダムでスリープ（0～499 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  }  //終了  printf("- end:(W)%s -\n", name);  fflush(stdout);  }  //読み込みスレッド  void threadFuncR(const char\* name)  {  //開始  printf("- begin:(R)%s -\n", name);  fflush(stdout);  //乱数  std::random\_device seed\_gen;  std::mt19937 rnd(seed\_gen());  std::uniform\_int\_distribution<int> sleep\_time(0, 499);  //若干ランダムでスリープ（0～499 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //処理  for (int i = 0; i < 3; ++i)  {  //リードロック取得  CRWLockR lock(s\_lock);  //データ表示（前）  printf("(R)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //若干ランダムでスリープ（0～499 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //データ表示（後）  printf("(R)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //リードロック解放  lock.unlock();  //若干ランダムでスリープ（0～499 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  }  //終了  printf("- end:(R)%s -\n", name);  fflush(stdout);  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  std::thread thread\_obj1 = std::thread(threadFuncR, "太郎");  std::thread thread\_obj2 = std::thread(threadFuncR, "次郎");  std::thread thread\_obj3 = std::thread(threadFuncR, "三郎");  std::this\_thread::sleep\_for(std::chrono::microseconds(1));  std::thread thread\_obj4 = std::thread(threadFuncW, "松子");  std::thread thread\_obj5 = std::thread(threadFuncW, "竹子");  std::thread thread\_obj6 = std::thread(threadFuncW, "梅子");  //スレッド終了待ち  {  auto begin = std::chrono::high\_resolution\_clock::now();  thread\_obj1.join();  thread\_obj2.join();  thread\_obj3.join();  thread\_obj4.join();  thread\_obj5.join();  thread\_obj6.join();  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(  std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  printf("Time = %.6f sec\n", duration);  }    //リード／ライトロックの取得と解放を大量に実行して時間を計測①：ライトロックのテスト  {  auto begin = std::chrono::high\_resolution\_clock::now();  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  CRWLockW lock(s\_lock);  lock.unlock();  }  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(  std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  printf("Read-WriteLock:wlock \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //リード／ライトロックの取得と解放を大量に実行して時間を計測②：リードロックのテスト  {  auto begin = std::chrono::high\_resolution\_clock::now();  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  CRWLockR lock(s\_lock);  lock.unlock();  }  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(  std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  printf("Read-WriteLock:rlock \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //リード／ライトロックの取得と解放を大量に実行して時間を計測③：「必要に応じてリードロック」のテスト  //※このケースでは全てロックがスルーされる  {  CThreadID current\_thread\_id;  auto begin = std::chrono::high\_resolution\_clock::now();  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  CRWLockR\_AsNecessary lock(s\_lock, current\_thread\_id);  // CRWLockR\_AsNecessary lock(s\_lock);//※スレッドIDの指定を省略したら現在のスレッドとして処理する  lock.unlock();  }  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(  std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  printf("Read-WriteLock:rlockAsNecessary \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //--------------------------------------------------------------------------------  //ロックオブジェクト使用方法サンプル  {  //「リードロック」の記述パターン  CRWLockR lock1(s\_lock); //パターン①：コンストラクタでロック  CRWLockR lock2(s\_lock.rLock()); //パターン②：rLock() メソッドでロック  CRWLockR lock3 = s\_lock.rLock();//パターン③：rLock() メソッドでロック　※代入演算子使用  auto lock4 = s\_lock.rLock(); //パターン④：rLock() メソッドでロック　※auto型使用  //※いずれも、明示的に .unlock() メソッドを呼ばなくても、  //　処理ブロックから抜けるときに自動的にロック解放する  }  {  //「必要に応じてリードロック」の記述パターン  CRWLockR\_AsNecessary lock1(s\_lock); //パターン①：コンストラクタでロック  CRWLockR\_AsNecessary lock2(s\_lock.rLockAsNecessary()); //パターン②：rLockAsNecessary () メソッドでロック  CRWLockR\_AsNecessary lock3 = s\_lock.rLockAsNecessary();//パターン③：rLockAsNecessary () メソッドでロック  // ※代入演算子使用  auto lock4 = s\_lock.rLockAsNecessary(); //パターン④：rLockAsNecessary () メソッドでロック  // ※auto型使用  //※いずれも、明示的に .unlock() メソッドを呼ばなくても、  //　処理ブロックから抜けるときに自動的にロック解放する  }  {  //「ライトロック」の記述パターン  CRWLockW lock1(s\_lock); //パターン①：コンストラクタでロック  lock1.unlock();  CRWLockW lock2(s\_lock.wLock()); //パターン②：rLock() メソッドでロック  lock2.unlock();  CRWLockW lock3 = s\_lock.wLock();//パターン③：rLock() メソッドでロック　※代入演算子使用  lock3.unlock();  auto lock4 = s\_lock.wLock(); //パターン④：rLock() メソッドでロック　※auto型使用  //※いずれも、明示的に .unlock() メソッドを呼ばなくても、  //　処理ブロックから抜けるときに自動的にロック解放する  //※ライトロックは多重呼び出しでデッドロックするので、上のサンプルでは明示的なアンロックを併用している  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(R)太郎 -  - begin:(R)次郎 -  - begin:(R)三郎 -  - begin:(W)松子 -  - begin:(W)竹子 -  - begin:(W)梅子 -  (W)松子: [BEFORE] commonData=0, tlsData=0  (W)松子: [AFTER] commonData=1, tlsData=1 ←ライトロック中は完全に排他（完了）  (W)梅子: [BEFORE] commonData=1, tlsData=0  (W)梅子: [AFTER] commonData=2, tlsData=1 ←ライトロック中は完全に排他（完了）  (W)竹子: [BEFORE] commonData=2, tlsData=0  (W)竹子: [AFTER] commonData=3, tlsData=1 ←ライトロック中は完全に排他（完了）  (W)松子: [BEFORE] commonData=3, tlsData=1  (W)松子: [AFTER] commonData=4, tlsData=2 ←ライトロック中は完全に排他（完了）  (R)三郎: [BEFORE] commonData=4, tlsData=0  (R)太郎: [BEFORE] commonData=4, tlsData=0 ←リードロックは同時取得可  (R)次郎: [BEFORE] commonData=4, tlsData=0 ←リードロックは同時取得可  (R)三郎: [AFTER] commonData=4, tlsData=0  (R)次郎: [AFTER] commonData=4, tlsData=0  (R)太郎: [AFTER] commonData=4, tlsData=0 ←リードロック中はライトロックをブロック  (W)松子: [BEFORE] commonData=4, tlsData=2  (W)松子: [AFTER] commonData=5, tlsData=3 ←ライトロック中は完全に排他（完了）  (W)竹子: [BEFORE] commonData=5, tlsData=1  (W)竹子: [AFTER] commonData=6, tlsData=2 ←ライトロック中は完全に排他（完了）  (W)梅子: [BEFORE] commonData=6, tlsData=1  - end:(W)松子 - ←ライトロックが優先的に処理された  (W)梅子: [AFTER] commonData=7, tlsData=2 ←ライトロック中は完全に排他（完了）  (W)竹子: [BEFORE] commonData=7, tlsData=2  (W)竹子: [AFTER] commonData=8, tlsData=3 ←ライトロック中は完全に排他（完了）  (W)梅子: [BEFORE] commonData=8, tlsData=2  - end:(W)竹子 -  (W)梅子: [AFTER] commonData=9, tlsData=3 ←ライトロック中は完全に排他（完了）  (R)次郎: [BEFORE] commonData=9, tlsData=0  (R)三郎: [BEFORE] commonData=9, tlsData=0 ←リードロックは同時取得可  (R)太郎: [BEFORE] commonData=9, tlsData=0 ←リードロックは同時取得可  - end:(W)梅子 - ←ライトロックが優先的に処理された  (R)三郎: [AFTER] commonData=9, tlsData=0  (R)太郎: [AFTER] commonData=9, tlsData=0  (R)三郎: [BEFORE] commonData=9, tlsData=0 ←リードロックは同時取得可  (R)次郎: [AFTER] commonData=9, tlsData=0  (R)太郎: [BEFORE] commonData=9, tlsData=0 ←リードロックは同時取得可  (R)太郎: [AFTER] commonData=9, tlsData=0  (R)三郎: [AFTER] commonData=9, tlsData=0  (R)次郎: [BEFORE] commonData=9, tlsData=0  - end:(R)太郎 -  - end:(R)三郎 -  (R)次郎: [AFTER] commonData=9, tlsData=0  - end:(R)次郎 –  Time = 3.700204 sec ←処理時間  Read-WriteLock:wlock \* 10000000 = 0.404023 sec ←1千万回ループによる処理時間計測（ライトロック）  Read-WriteLock:rlock \* 10000000 = 0.440026 sec ←1千万回ループによる処理時間計測（リードロック）  Read-WriteLock:rlockAsNecessary \* 10000000 = 0.163008 sec ←1千万回ループによる処理時間計測（必要に応じて  　　　　　　　　　　　　　　　　　　リードロック）  ※「必要に応じてリードロック」がかなり高速なことがわかる |

※リード・ライトロックオブジェクト生成時に、パラメータ CRWLock::ALL\_WLOCK を渡して、リードロックをライトロックの振る舞いにした場合の実行結果（完全な排他になり、安直にミューテックスで排他制御したのと同様の状態）

↓

|  |
| --- |
| - begin:(R)太郎 -  - begin:(R)次郎 -  - begin:(R)三郎 -  - begin:(W)松子 -  - begin:(W)竹子 -  - begin:(W)梅子 -  (R)次郎: [BEFORE] commonData=0, tlsData=0  (R)次郎: [AFTER] commonData=0, tlsData=0 ←完全に排他（完了）  (W)梅子: [BEFORE] commonData=0, tlsData=0  (W)梅子: [AFTER] commonData=1, tlsData=1 ←完全に排他（完了）  (W)竹子: [BEFORE] commonData=1, tlsData=0  (W)竹子: [AFTER] commonData=2, tlsData=1 ←完全に排他（完了）  (R)次郎: [BEFORE] commonData=2, tlsData=0  (R)次郎: [AFTER] commonData=2, tlsData=0 ←完全に排他（完了）  (R)三郎: [BEFORE] commonData=2, tlsData=0  (R)三郎: [AFTER] commonData=2, tlsData=0 ←完全に排他（完了）  (W)梅子: [BEFORE] commonData=2, tlsData=1  (W)梅子: [AFTER] commonData=3, tlsData=2 ←完全に排他（完了）  (R)太郎: [BEFORE] commonData=3, tlsData=0  (R)太郎: [AFTER] commonData=3, tlsData=0 ←完全に排他（完了）  (W)松子: [BEFORE] commonData=3, tlsData=0  (W)松子: [AFTER] commonData=4, tlsData=1 ←完全に排他（完了）  (R)三郎: [BEFORE] commonData=4, tlsData=0  (R)三郎: [AFTER] commonData=4, tlsData=0 ←完全に排他（完了）  (W)竹子: [BEFORE] commonData=4, tlsData=1  (W)竹子: [AFTER] commonData=5, tlsData=2 ←完全に排他（完了）  (W)梅子: [BEFORE] commonData=5, tlsData=2  (W)梅子: [AFTER] commonData=6, tlsData=3 ←完全に排他（完了）  (R)三郎: [BEFORE] commonData=6, tlsData=0  - end:(W)梅子 -  (R)三郎: [AFTER] commonData=6, tlsData=0 ←完全に排他（完了）  (W)松子: [BEFORE] commonData=6, tlsData=1  - end:(R)三郎 -  (W)松子: [AFTER] commonData=7, tlsData=2 ←完全に排他（完了）  (R)太郎: [BEFORE] commonData=7, tlsData=0  (R)太郎: [AFTER] commonData=7, tlsData=0 ←完全に排他（完了）  (W)竹子: [BEFORE] commonData=7, tlsData=2  (W)竹子: [AFTER] commonData=8, tlsData=3 ←完全に排他（完了）  (W)松子: [BEFORE] commonData=8, tlsData=2  - end:(W)竹子 -  (W)松子: [AFTER] commonData=9, tlsData=3 ←完全に排他（完了）  (R)太郎: [BEFORE] commonData=9, tlsData=0  (R)太郎: [AFTER] commonData=9, tlsData=0 ←完全に排他（完了）  (R)次郎: [BEFORE] commonData=9, tlsData=0  - end:(W)松子 -  (R)次郎: [AFTER] commonData=9, tlsData=0  - end:(R)太郎 -  - end:(R)次郎 -  Time = 5.624323 sec ←処理時間  Read-WriteLock:wlock \* 10000000 = 0.416024 sec ←1千万回ループによる処理時間計測（ライトロック）  Read-WriteLock:rlock \* 10000000 = 0.440026 sec ←1千万回ループによる処理時間計測（リードロック）  Read-WriteLock:rlockAsNecessary \* 10000000 = 0.436025 sec ←1千万回ループによる処理時間計測（必要に応じて  　　　　　　　　　　　　　　　　　　リードロック）  ※並行処理ができず、非効率になって処理時間がかかっていることがわかる  ※「必要に応じてリードロック」も含めて、どのロックも同様の速度になっている |

## シングルトンクラス

前述の要件定義に基づく実装例を示す。

処理共通化のために、先のクラス図よりももう少しクラスが増えている。

なお、前述のスレッドIDクラスも使用している。

**【クラス宣言】**

|  |
| --- |
| //--------------------------------------------------------------------------------  //シングルトンクラス  //※現状は可読性重視だが、実際にはテンプレートのインスタンス化による  //　プログラムサイズの肥大を考慮し、非テンプレートの共通処理を  //　切り出す必要がある  //----------------------------------------  //クラス宣言  //※friend宣言のための事前宣言  //共通シングルトンテンプレート　※継承専用  template<class T, class U>  class CSingletonCommon;  //通常シングルトンテンプレートクラス  template<class T>  class CSingleton;  //管理シングルトンテンプレートクラス  template<class T>  class CManagedSingleton;  //【シングルトン用ヘルパー】シングルトンプロキシーテンプレートクラス　※継承専用  template<class T>  class CSingletonProxy;  //【シングルトン用ヘルパー】シングルトンイニシャライザーテンプレートクラス  template<class T>  class CSingletonInitializer;  //【シングルトン用ヘルパー】シングルトンアクセステンプレートクラス  template<class T>  class CSingletonUsing; |

**【シングルトン用定数定義】**

|  |
| --- |
| //----------------------------------------  //シングルトン処理用定数定義  class CSingletonConst  {  public:  //定数定義  //シングルトン属性  enum E\_ATTR  {  ATTR\_AUTO\_CREATE, //自動生成（破棄しない）  ATTR\_AUTO\_CREATE\_AND\_DELETE, //自動生成と自動破棄  ATTR\_MANUAL\_CREATE\_AND\_DELETE,//手動生成と手動破棄  };  //スレッドセーフ宣言  enum E\_IS\_THREAD\_SAFE  {  IS\_THREAD\_SAFE = true,//スレッドセーフ宣言  IS\_NOT\_THREAD\_SAFE = false,//非スレッドセーフ宣言  };  //管理シングルトン宣言  enum E\_IS\_MANAGED\_SINGLETON  {  IS\_NORMAL\_SINGLETON = false,//通常シングルトン宣言  IS\_MANAGED\_SINGLETON = true,//管理シングルトン宣言  };  //シングルトン生成済み状態  enum E\_IS\_CREATED  {  IS\_NOT\_CREATED = false,//未生成  IS\_CREATED = true,//生成済み  };  //強制処理指定  enum E\_IS\_FORCED  {  IS\_FORCED = true,//強制  IS\_NORMAL = false,//通常  };  //ファイナライズ処理指定  enum E\_IS\_FINALIZE  {  IS\_FINALIZE = true,//ファイナライズ  IS\_RELEASE = false,//参照カウンタリリース  };  //シングルトンイニシャライザー指定  enum E\_IS\_INITIALIZER  {  IS\_INITIALIZER = true,//シングルトンイニシャライザー  IS\_USING = false,//シングルトンアクセス  };  public:  //データ型    //【テンプレートクラスの挙動を宣言し、静的アサーション（static\_assert）を行うための構造体】  //コンパイル動作設定用構造体：通常シングルトン宣言  struct AUTO\_SINGLETON\_TYPE  {  static const E\_IS\_MANAGED\_SINGLETON THIS\_IS\_MANAGED\_SINGLETON = IS\_NORMAL\_SINGLETON;  };  //コンパイル動作設定用構造体：管理シングルトン宣言  struct MANAGED\_SINGLETON\_TYPE  {  static const E\_IS\_MANAGED\_SINGLETON THIS\_IS\_MANAGED\_SINGLETON = IS\_MANAGED\_SINGLETON;  };  public:  //メソッド  //【定数文字列化】シングルトン属性  static const char\* Attr\_ToStr(const E\_ATTR attr)  {  #define CASE\_TO\_STR(x) case ATTR\_##x: return #x; break;  switch (attr)  {  CASE\_TO\_STR(AUTO\_CREATE);  CASE\_TO\_STR(AUTO\_CREATE\_AND\_DELETE);  CASE\_TO\_STR(MANUAL\_CREATE\_AND\_DELETE);  }  #undef CASE\_TO\_STR  return "(unknown)";  }  //【定数文字列化】スレッドセーフ宣言  static const char\* IsThreadSafe\_ToStr(const E\_IS\_THREAD\_SAFE is\_thread\_safe)  {  #define CASE\_TO\_STR(x) case x: return #x; break;  switch (is\_thread\_safe)  {  CASE\_TO\_STR(IS\_THREAD\_SAFE);  CASE\_TO\_STR(IS\_NOT\_THREAD\_SAFE);  }  #undef CASE\_TO\_STR  return "(unknown)";  }  //【定数文字列化】管理シングルトン宣言  static const char\* IsManagedSingleton\_ToStr(const E\_IS\_MANAGED\_SINGLETON is\_managed\_singleton)  {  #define CASE\_TO\_STR(x) case x: return #x; break;  switch (is\_managed\_singleton)  {  CASE\_TO\_STR(IS\_MANAGED\_SINGLETON);  CASE\_TO\_STR(IS\_NORMAL\_SINGLETON);  }  #undef CASE\_TO\_STR  return "(unknown)";  }  //【定数文字列化】シングルトン生成済み状態  static const char\* IsCreated\_ToStr(const E\_IS\_CREATED is\_created)  {  #define CASE\_TO\_STR(x) case x: return #x; break;  switch (is\_created)  {  CASE\_TO\_STR(IS\_CREATED);  CASE\_TO\_STR(IS\_NOT\_CREATED);  }  #undef CASE\_TO\_STR  return "(unknown)";  }  //【定数文字列化】強制処理指定  static const char\* IsForced\_ToStr(const E\_IS\_FORCED is\_forced)  {  #define CASE\_TO\_STR(x) case x: return #x; break;  switch (is\_forced)  {  CASE\_TO\_STR(IS\_FORCED);  CASE\_TO\_STR(IS\_NORMAL);  }  #undef CASE\_TO\_STR  return "(unknown)";  }  //【定数文字列化】ファイナライズ指定  static const char\* IsFinalize\_ToStr(const E\_IS\_FINALIZE is\_finalize)  {  #define CASE\_TO\_STR(x) case x: return #x; break;  switch (is\_finalize)  {  CASE\_TO\_STR(IS\_FINALIZE);  CASE\_TO\_STR(IS\_RELEASE);  }  #undef CASE\_TO\_STR  return "(unknown)";  }  //【定数文字列化】イニシャライザー指定  static const char\* IsInitializer\_ToStr(const E\_IS\_INITIALIZER is\_initializer)  {  #define CASE\_TO\_STR(x) case x: return #x; break;  switch (is\_initializer)  {  CASE\_TO\_STR(IS\_INITIALIZER);  CASE\_TO\_STR(IS\_USING);  }  #undef CASE\_TO\_STR  return "(unknown)";  }  //【定数文字列化】論理値  static const char\* Bool\_ToStr(const bool flag)  {  return flag ? "TRUE" : "FALSE";  }  }; |

**【共通シングルトンテンプレートクラス（継承専用）】**

|  |
| --- |
| //----------------------------------------  //共通シングルトンテンプレートクラス　※継承専用  template<class T, class U>  class CSingletonCommon  {  //フレンドクラス  friend class CSingleton<T>;  friend class CManagedSingleton<T>;  friend class CSingletonProxy<T>;  friend class CSingletonInitializer<T>;  friend class CSingletonUsing<T>;  public:  //シングルトン設定をテンプレート引数のクラスに基づいて静的に確定  //※クラス T には、定数 CLASS\_NAME, SINGLETON\_ATTR, IS\_THREAD\_SAFE が定義されている必要がある。  //※クラス U には、定数 THIS\_IS\_MANAGED\_SINGLETON が定義されている必要がある。  //static const char\* T::CLASS\_NAME;//クラス名 ※char\*型のため、クラスの静的な値として反映できない  static const CSingletonConst::E\_ATTR THIS\_SINGLETON\_ATTR = T::SINGLETON\_ATTR;//シングルトン属性  static const CSingletonConst::E\_IS\_THREAD\_SAFE THIS\_IS\_THREAD\_SAFE = T::THIS\_IS\_THREAD\_SAFE;//スレッドセーフ宣言  static const CSingletonConst::E\_IS\_MANAGED\_SINGLETON THIS\_IS\_MANAGED\_SINGLETON = U::THIS\_IS\_MANAGED\_SINGLETON;  //管理シングルトン宣言    //【静的アサーション】通常シングルトンでは、「手動生成属性：ATTR\_MANUAL\_CREATE\_AND\_DELETE」を使用すると  //　　　　　　　　　　アサーション違反  STATIC\_ASSERT(THIS\_IS\_MANAGED\_SINGLETON == CSingletonConst::IS\_NORMAL\_SINGLETON &&  THIS\_SINGLETON\_ATTR != CSingletonConst::ATTR\_MANUAL\_CREATE\_AND\_DELETE ||  THIS\_IS\_MANAGED\_SINGLETON != CSingletonConst::IS\_NORMAL\_SINGLETON,  "CSingleton<T> is not supported ATTR\_MANUAL\_CREATE\_AND\_DELETE in THIS\_SINGLETON\_ATTR.");  //【静的アサーション】管理シングルトンでは、「手動生成属性：ATTR\_MANUAL\_CREATE\_AND\_DELETE」以外を使用すると  //　　　　　　　　　　　　アサーション違反  STATIC\_ASSERT(THIS\_IS\_MANAGED\_SINGLETON == CSingletonConst::IS\_MANAGED\_SINGLETON &&  THIS\_SINGLETON\_ATTR == CSingletonConst::ATTR\_MANUAL\_CREATE\_AND\_DELETE ||  THIS\_IS\_MANAGED\_SINGLETON != CSingletonConst::IS\_MANAGED\_SINGLETON,  "CManagedSingleton<T> is only supported ATTR\_MANUAL\_CREATE\_AND\_DELETE in THIS\_SINGLETON\_ATTR.");  private:  //インスタンス生成・破棄用配置new/delete  //※new は内部で持つ static バッファのポインタを返すだけ  //※delete は何もしない  //※コンストラクタ、デストラクタを実行して、オブジェクトの初期化・終了処理を実行することが目的  void\* operator new(const size\_t size){ return m\_buff; }  void operator delete(void\*){}  public:  //アクセッサ  const char\* getClassName() const { return T::CLASS\_NAME; }//クラス名取得  CSingletonConst::E\_ATTR getAttr() const { return THIS\_SINGLETON\_ATTR; }//シングルトン属性取得  const char\* getAttr\_Named() const { return CSingletonConst::Attr\_ToStr(getAttr()); }  //シングルトン属性名取得（デバッグ用）  CSingletonConst::E\_IS\_THREAD\_SAFE isThreadSafe() const { return THIS\_IS\_THREAD\_SAFE; }//スレッドセーフ宣言取得  const char\* isThreadSafe\_Named() const { return CSingletonConst::IsThreadSafe\_ToStr(isThreadSafe()); }  //スレッドセーフ宣言名取得（デバッグ用）  CSingletonConst::E\_IS\_MANAGED\_SINGLETON isManagedSingleton() const { return THIS\_IS\_MANAGED\_SINGLETON; }  //管理シングルトン宣言取得  const char\* isManagedSingleton\_Named() const  { return CSingletonConst::IsManagedSingleton\_ToStr(isManagedSingleton()); }  //管理シングルトン宣言名取得（デバッグ用）  CSingletonConst::E\_IS\_CREATED isCreated() const  { return m\_this.load() != nullptr ? CSingletonConst::IS\_CREATED : CSingletonConst::IS\_NOT\_CREATED; }  //クラス(T)インスタンス生成済み状態取得  const char\* isCreated\_Named() const { return CSingletonConst::IsCreated\_ToStr(isCreated()); }  //クラス(T)インスタンス生成済み状態名取得（デバッグ用）  private:  T\* getThis() { return m\_this.load(); }//クラス(T)インスタンスの参照を取得（禁止）  const T\* getThis() const { return m\_this.load(); }//クラス(T)インスタンスのconst参照を取得（禁止）  public:  int getRefCount() const { return m\_refCount.load(); }//参照カウンタ取得  int getRefCountMax() const { return m\_refCountMax.load(); }//参照カウンタの最大到達値を取得  int getRefCountOnThread() const { return m\_refCountOnThread; }//現在のスレッド内の参照カウンタ数を取得  int getThreadCount() const { return m\_threadCount.load(); }//参照スレッド数取得  int getThreadCountMax() const { return m\_threadCountMax.load(); }//参照スレッド数の最大到達値を取得  THREAD\_ID getCreatedThreadID() const { return m\_createdThreadId; }  //クラス(T)インスタンスを生成したスレッドのスレッドIDを取得  const char\* getCreatedThreadName() const { return m\_createdThreadName; }  //クラス(T)インスタンスを生成したスレッドのスレッド名を取得  public:  //オペレータ  T\* operator->(){ return m\_this.load(); }  //アロー演算子：シングルトンクラスがクラス(T)のインスタンスを偽装（代理）する  const T\* operator->() const { return m\_this.load(); }//const アロー演算子：（同上）  private:  //オペレータ（禁止）  T& operator\*(){ return \*m\_this.load(); }//ポインタ演算子（禁止）  const T& operator\*() const { return \*m\_this.load(); }//constポインタ演算子（禁止）  operator T\*(){ return m\_this.load(); }//クラス T\* キャスト演算子（禁止）  operator const T\*() const { return m\_this.load(); }//クラスconst T\* キャスト演算子（禁止）  operator T&(){ return \*m\_this.load(); }//クラス T& キャスト演算子（禁止）  operator const T&() const { return \*m\_this.load(); }//クラス const T& キャスト演算子（禁止）  CSingletonCommon<T, U>& operator=(const CSingletonCommon<T, U>&){ return \*this; }//コピー演算子（禁止）  CSingletonCommon<T, U>& operator=(const CSingletonCommon<T, U>\*){ return \*this; }//コピー演算子（禁止）  CSingletonCommon<T, U>& operator=(const T&){ return \*this; }//コピー演算子（禁止）  CSingletonCommon<T, U>& operator=(const T\*){ return \*this; }//コピー演算子（禁止）  private:  //コピーコンストラクタ（禁止）  explicit CSingletonCommon(CSingletonCommon<T, U>&){}  explicit CSingletonCommon(CSingletonCommon<T, U>\*){}  explicit CSingletonCommon(T&){}  explicit CSingletonCommon(T\*){}  private:  //参照カウンタのカウントアップ  //※最初のカウントアップ時にインスタンスを生成  //※なるべくロックフリーにするためにカウンタの判定は演算後の値のみを用いている  //　...が、結局インスタンスの生成を確実に待たせる必要があるので、結局全体をロックする。  //※「自動生成属性：ATTR\_AUTO\_CREATE」の時は、CallOnceにより初期化の衝突を保護  //　... が、当初の予定に反してスレッドセーフをより強化したので、CallOnceは廃止。  //　参考用に CallOnce を使用する場合の処理をコメントとして残す。  //※C++11仕様の可変長テンプレートを使用し、クラス T をインスタンス化する際に、  //　任意のパラメータを与えることを可能にしている。  //　C++11非対応のコンパイラでは、クラス T にデフォルトコンストラクタしか  //　使えないものとする。  template<typename... Tx>  bool addRef(Tx... nx)  {  //インスタンス生成用ラムダ式  //※可変長テンプレート引数を使用した関数の仮引数を、ラムダ式の外部参照に指定している  auto creator = [&nx...](bool& is\_created)  {  if (m\_this.load() == nullptr)  {  //クラス内 配置 new を利用し、static 領域を割り当て  //※可変長テンプレート引数を使用した関数の仮引数を受け渡し  m\_this.store(new T(nx...));  //生成時情報を記録  {  CThreadID thread\_id;//現在のスレッド情報  m\_createdThreadId = thread\_id.getID();//スレッドID記録  m\_createdThreadName = thread\_id.getName();//スレッド名記録  is\_created = true;//生成OK  }  }  };  //参照カウンタカウントアップ（共通処理）  return addRefCore(creator);  }  //参照カウンタのカウントアップ  //※インスタンスを生成しない  bool addRefWitoutCreate()  {  //インスタンス生成用ラムダ式（ダミー）  auto creator\_dummy = [](bool& is\_created)  {  };  //参照カウンタカウントアップ（共通処理）  return addRefCore(creator\_dummy);  }  //参照カウンタのカウントアップ（共通処理部）  //※実際のインスタンス生成処理部はラムダ式で受け取る  //※ラムダ式を受け取るためにテンプレート関数化している  template<typename L>  bool addRefCore(L& creator)  {  //カウントアップ済み判定  if (m\_isCounted)  return false;  m\_isCounted = true;    //ロック取得  m\_instanceLock.lock();  //参照カウンタをカウントアップ  bool is\_allow\_create = false;//インスタンス生成許可フラグ  {  const int ref\_count\_prev = m\_refCount.fetch\_add(1);//カウントアップ  const int ref\_count\_now = ref\_count\_prev + 1;  if (ref\_count\_prev == 0)  is\_allow\_create = true;//初めてのカウントアップ時にインスタンス生成  //最大到達値を記録  if (m\_refCountMax.load() < ref\_count\_now)  m\_refCountMax.store(ref\_count\_now);  }  //スレッド内参照数をカウントアップ  {  const int ref\_count\_on\_thread\_prev = m\_refCountOnThread++;  if (ref\_count\_on\_thread\_prev == 0)  {  //初めてのカウントアップ時に参照スレッド数をカウントアップ  const int thread\_count\_prev = m\_threadCount.fetch\_add(1);  const int thread\_count\_now = thread\_count\_prev + 1;  //最大到達値を記録  if (m\_threadCountMax.load() < thread\_count\_now)  m\_threadCountMax.store(thread\_count\_now);  }  }  //インスタンス生成  bool is\_created = false;//生成済みフラグ  if (is\_allow\_create)  {  //※CallOnce廃止  // //「自動生成属性：ATTR\_AUTO\_CREATE」の時は CallOnce で生成  // if (THIS\_SINGLETON\_ATTR == CSingletonConst::ATTR\_AUTO\_CREATE)  // {  // std::call\_once(m\_once, creator, is\_created);  // }  // //「自動生成属性：ATTR\_AUTO\_CREATE」以外の時は普通に生成  // else  // {  // creator(is\_created);  // }  //常に普通にインスタンス生成  creator(is\_created);  }  //ロック解放  m\_instanceLock.unlock();  //インスタンス生成を行ったかどうかを返す  return is\_created;  }  //参照カウンタのカウントダウン  //※最後のカウントダウン時にインスタンスを破棄  //※なるべくロックフリーにするためにカウンタの判定は演算後の値のみを用いている  //　...が、結局インスタンスの生成を確実に待たせる必要があるので、結局全体をロックする。  bool release()  {  //カウントアップ済み判定  if (!m\_isCounted)  return false;  m\_isCounted = false;  //ロック取得  m\_instanceLock.lock();  //参照カウンタをカウントダウン  bool is\_allow\_delete = false;//インスタンス破棄許可フラグ  {  const int ref\_count\_prev = m\_refCount.fetch\_sub(1);//カウントダウン  const int ref\_count\_now = ref\_count\_prev - 1;  if (ref\_count\_now == 0)  is\_allow\_delete = true;//最後のカウントダウン時にインスタンス破棄  }    //スレッド内参照数をカウントダウン  {  const int ref\_count\_on\_thread\_now = --m\_refCountOnThread;  if (ref\_count\_on\_thread\_now == 0)  {  //カウント 0 時に参照スレッド数をカウントダウン  if (m\_threadCount.load() > 0)  {  m\_threadCount.fetch\_sub(1);  }  }  }  //インスタンス破棄  bool is\_deleted = false;//破棄済みフラグ  if (is\_allow\_delete)  {  //「自動生成・破棄属性：ATTR\_AUTO\_CREATE\_AND\_DELETE」の時だけ自動破棄  if (THIS\_SINGLETON\_ATTR == CSingletonConst::ATTR\_AUTO\_CREATE\_AND\_DELETE)  {  //インスタンス破棄（共通処理利用）  is\_deleted = deleteThisCore();  }  }  //ロック解放  m\_instanceLock.unlock();  //インスタンス破棄を行ったかどうかを返す  return is\_deleted;  }  //手動破棄  //※参照カウンタは更新しない  bool deleteThis()  {  //カウントアップ済み状態解除  m\_isCounted = false;    //ロック取得  m\_instanceLock.lock();  //インスタンス破棄（共通処理利用）  const bool is\_deleted = deleteThisCore();  //ロック解放  m\_instanceLock.unlock();  //インスタンス破棄を行ったかどうかを返す  return is\_deleted;  }  //破棄（共通部）  bool deleteThisCore()  {  //インスタンス破棄  bool is\_deleted = false;//破棄済みフラグ  if (m\_this.load() != nullptr)  {  //インスタンスを破棄  delete m\_this.load();  m\_this.store(nullptr);  //生成時情報をリセット  m\_createdThreadId = INVALID\_THREAD\_ID;//スレッドID  m\_createdThreadName = nullptr;//スレッド名  is\_deleted = true;//破棄OK  }  //インスタンス破棄を行ったかどうかを返す  return is\_deleted;  }  private:  //通常シングルトン用コンストラクタ  //※コンストラクタで自動的に参照カウンタをカウントアップし、インスタンスを生成する。  //※C++11仕様の可変長テンプレートを使用し、クラス T をインスタンス化する際に、  //　コンストラクタに任意のパラメータを与えることを可能にしている。  //　C++11非対応のコンパイラでは、クラス T にデフォルトコンストラクタしか  //　使えないものとする。  template<typename... Tx>  CSingletonCommon(CSingleton<T>\*, Tx... nx) :  m\_isCounted(false)  {  //【アサーション】通常シングルトンでは、「手動生成属性：ATTR\_MANUAL\_CREATE\_AND\_DELETE」を使用すると  // アサーション違反（処理続行可）  ASSERT(THIS\_SINGLETON\_ATTR != CSingletonConst::ATTR\_MANUAL\_CREATE\_AND\_DELETE,  "CSingleton<T> is not supported ATTR\_MANUAL\_CREATE\_AND\_DELETE in THIS\_SINGLETON\_ATTR.");    //【アサーション】インスタンス生成済みの場合、スレッドセーフじゃないクラスに対して、  // 生成時と異なるスレッドからアクセスするとアサーション違反（処理続行可）  CThreadID this\_thread;  ASSERT(m\_createdThreadId == INVALID\_THREAD\_ID ||  THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE ||  THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE &&  m\_createdThreadId == this\_thread.getID(),  "CSingleton<T> is not thread safe. Create thread and this thread are different.");  //参照カウンタをカウントアップ  //※初めてのカウントアップならインスタンスを生成する  addRef(nx...);  }  //管理シングルトン用コンストラクタ  //※コンストラクタで参照カウンタを更新せず、インスタンスを生成しない。  CSingletonCommon(CManagedSingleton<T>\*) :  m\_isCounted(false)  {  //【アサーション】管理シングルトンでは、「手動生成属性：ATTR\_MANUAL\_CREATE\_AND\_DELETE」以外を使用すると  // アサーション違反（処理続行可）  ASSERT(THIS\_SINGLETON\_ATTR == CSingletonConst::ATTR\_MANUAL\_CREATE\_AND\_DELETE,  "CManagedSingleton<T> is only supported ATTR\_MANUAL\_CREATE\_AND\_DELETE in THIS\_SINGLETON\_ATTR.");    //【アサーション】インスタンス生成済みの場合、スレッドセーフじゃないクラスに対して、  // 生成時と異なるスレッドからアクセスするとアサーション違反（処理続行可）  CThreadID this\_thread;  ASSERT(m\_createdThreadId == INVALID\_THREAD\_ID ||  THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE ||  THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE &&  m\_createdThreadId == this\_thread.getID(),  "CManagedSingleton<T> is not thread safe. Create thread and this thread are different.");  }  //デストラクタ  //※デストラクタで自動的に参照カウンタをカウントダウンする。  //※「自動生成属性：ATTR\_AUTO\_CREATE\_AND\_DELTE」の時は、参照カウンタが 0 で自動破棄する。  ~CSingletonCommon()  {  //参照カウンタをカウントダウン  //※最後のカウントダウンならインスタンスを破棄する  release();  }  private:  //フィールド  bool m\_isCounted;//カウントアップ済み  //static std::once\_flag m\_once;//CallOnceフラグ ※CallOnce廃止  static CSpinLock m\_instanceLock;//生成・破棄処理ロックフラグ  static std::atomic<T\*> m\_this;//クラス T のインスタンス（ポインタ）  static char m\_buff[sizeof(T)];//クラス T のインスタンス用の static 領域  static std::atomic<int> m\_refCount;//参照カウンタ  static std::atomic<int> m\_refCountMax;//参照カウンタの最大到達値  static thread\_local int m\_refCountOnThread;//【TLS】現在のスレッド内の参照カウンタ  static std::atomic<int> m\_threadCount;//参照スレッド数  static std::atomic<int> m\_threadCountMax;//参照スレッド数の最大到達値  static THREAD\_ID m\_createdThreadId;//インスタンス生成時のスレッドのスレッドID  static const char\* m\_createdThreadName;//インスタンス生成時のスレッドのスレッド名  }; |

**【共通シングルトンテンプレートクラス（継承専用）：補助マクロ】**

|  |
| --- |
| //----------------------------------------  //friend クラス指定補助マクロ  //※このマクロを直接使用せず、SINGLETON\_FRIEND か MANAGED\_SINGLETON\_FRIEND を使用する  #define SINGLETON\_COMMON\_FRIEND(T, U) \  friend class CSingletonCommon<T, U>;  //----------------------------------------  //シングルトン設定補助マクロ  //※このマクロを直接使用せず、SINGLETON\_ATTR か MANAGED\_SINGLETON\_ATTR を使用する  #define SINGLETON\_COMMON\_ATTR(attr, is\_thread\_safe) \  static const char\* CLASS\_NAME; \  static const CSingletonConst::E\_ATTR SINGLETON\_ATTR = attr; \  static const CSingletonConst::E\_IS\_THREAD\_SAFE THIS\_IS\_THREAD\_SAFE = is\_thread\_safe;  //----------------------------------------  //シングルトンクラスの static インスタンス生成用マクロ  //※このマクロを直接使用せず、MAKE\_SINGLETON\_INSTANCE か MAKE\_MANAGED\_SINGLETON\_INSTANCE\_ALL を使用する  #define MAKE\_SINGLETON\_COMMON\_INSTANCE(T, U) \  /\*std::once\_flag CSingletonCommon<T, U>::m\_once;\*//\*CallOnceフラグ ※CallOnce廃止\*/ \  CSpinLock CSingletonCommon<T, U>::m\_instanceLock;/\*生成・破棄処理ロックフラグ\*/ \  std::atomic<T\*> CSingletonCommon<T, U>::m\_this(nullptr);/\*クラス T のインスタンス（ポインタ）\*/ \  char CSingletonCommon<T, U>::m\_buff[sizeof(T)];/\*クラス T のインスタンス用の static 領域\*/ \  std::atomic<int> CSingletonCommon<T, U>::m\_refCount(0);/\*参照カウンタ\*/ \  std::atomic<int> CSingletonCommon<T, U>::m\_refCountMax(0);/\*参照カウンタの最大到達値\*/ \  thread\_local int CSingletonCommon<T, U>::m\_refCountOnThread = 0;/\*【TLS】現在のスレッド内の参照カウンタ\*/ \  std::atomic<int> CSingletonCommon<T, U>::m\_threadCount(0);/\*参照スレッド数\*/ \  std::atomic<int> CSingletonCommon<T, U>::m\_threadCountMax(0);/\*参照スレッド数の最大到達値\*/ \  THREAD\_ID CSingletonCommon<T, U>::m\_createdThreadId = INVALID\_THREAD\_ID; \  /\*インスタンス生成時のスレッドのスレッドID\*/ \  const char\* CSingletonCommon<T, U>::m\_createdThreadName = nullptr;/\*インスタンス生成時のスレッドのスレッド名\*/ \  const char\* T::CLASS\_NAME = #T;/\*クラス名\*/ |

**【通常シングルトンテンプレートクラス】**

|  |
| --- |
| //----------------------------------------  //通常シングルトンテンプレートクラス  template<class T>  class CSingleton : public CSingletonCommon<T, CSingletonConst::AUTO\_SINGLETON\_TYPE>  {  public:  //メソッド    //デバッグ情報表示  void printDebugInfo(FILE\* fp = stdout)  {  CThreadID thread\_id;  DEBUG\_FPRINT(fp, "----------------------------------------\n");  while (m\_instanceLock.lock()){}//ロック取得  DEBUG\_FPRINT(fp, "Debug Info: [%s] on \"%s\"(0x%08x)\n",  getClassName(), thread\_id.getName(), thread\_id.getID());  DEBUG\_FPRINT(fp, " ClassAttribute = %s\n", getAttr\_Named());  DEBUG\_FPRINT(fp, " ClassIsThreadSafe = %s\n", isThreadSafe\_Named());  DEBUG\_FPRINT(fp, " ClassIsManaged = %s\n", isManagedSingleton\_Named());  DEBUG\_FPRINT(fp, " ClassIsCreated = %s\n", isCreated\_Named());  DEBUG\_FPRINT(fp, " RefCount = %d (max=%d)\n", getRefCount(), getRefCountMax());  DEBUG\_FPRINT(fp, " RefCountOnThisThread = %d\n", getRefCountOnThread());  DEBUG\_FPRINT(fp, " ThreadCount = %d (max=%d)\n", getThreadCount(), getThreadCountMax());  DEBUG\_FPRINT(fp, " CreatedThread = \"%s\"(0x%08x)\n",  getCreatedThreadName(), getCreatedThreadID());  m\_instanceLock.unlock();//ロック解放  DEBUG\_FPRINT(fp, "----------------------------------------\n");  DEBUG\_FFLUSH(fp);  }  public:  //コンストラクタ  //※コンストラクタで自動的に参照カウンタをカウントアップし、インスタンスを生成する。  //※C++11仕様の可変長テンプレートを使用し、クラス T をインスタンス化する際に、  //　コンストラクタに任意のパラメータを与えることを可能にしている。  //　C++11非対応のコンパイラでは、クラス T にデフォルトコンストラクタしか  //　使えないものとする。  template<typename... Tx>  CSingleton(Tx... nx) :  CSingletonCommon<T, CSingletonConst::AUTO\_SINGLETON\_TYPE>::CSingletonCommon(this, nx...)  {  }  //デストラクタ  //※デストラクタで自動的に参照カウンタをカウントダウンする。  //※「自動生成属性：ATTR\_AUTO\_CREATE\_AND\_DELTE」の時は、参照カウンタが 0 で自動破棄する。  ~CSingleton()  {  }  }; |

**【通常シングルトンテンプレートクラス：補助マクロ】**

|  |
| --- |
| //----------------------------------------  //通常シングルトンクラス用 friend クラス指定補助マクロ  //※シングルトン対象クラス内に記述する  //※シングルトン対象クラス自身のクラス名を渡す  #define SINGLETON\_FRIEND(T) \  SINGLETON\_COMMON\_FRIEND(T, CSingletonConst::AUTO\_SINGLETON\_TYPE);  //----------------------------------------  //通常シングルトンクラス用シングルトン設定補助マクロ  //※シングルトン対象クラス内に記述する（privateスコープでもよい）  //※シングルトン属性とスレッドセーフ宣言を渡す  #define SINGLETON\_ATTR(attr, is\_thread\_safe) \  SINGLETON\_COMMON\_ATTR(attr, is\_thread\_safe);  //通常シングルトンクラス用シングルトン属性：自動生成のみ＋スレッドセーフ宣言  #define SINGLETON\_ATTR\_AUTO\_CREATE\_WITH\_THREAD\_SAFE() \  SINGLETON\_COMMON\_ATTR(CSingletonConst::ATTR\_AUTO\_CREATE, CSingletonConst::IS\_THREAD\_SAFE);  //通常シングルトンクラス用シングルトン属性：自動生成のみ＋非スレッドセーフ宣言  #define SINGLETON\_ATTR\_AUTO\_CREATE\_WITHOUT\_THREAD\_SAFE() \  SINGLETON\_COMMON\_ATTR(CSingletonConst::ATTR\_AUTO\_CREATE, CSingletonConst::IS\_NOT\_THREAD\_SAFE);  //通常シングルトンクラス用シングルトン属性：自動生成／自動削除＋スレッドセーフ宣言  #define SINGLETON\_ATTR\_AUTO\_CREATE\_AND\_DELETE\_WITH\_THREAD\_SAFE() \  SINGLETON\_COMMON\_ATTR(CSingletonConst::ATTR\_AUTO\_CREATE\_AND\_DELETE, CSingletonConst::IS\_THREAD\_SAFE);  //通常シングルトンクラス用シングルトン属性：自動生成／自動削除＋非スレッドセーフ宣言  #define SINGLETON\_ATTR\_AUTO\_CREATE\_AND\_DELETE\_WITHOUT\_THREAD\_SAFE() \  SINGLETON\_COMMON\_ATTR(CSingletonConst::ATTR\_AUTO\_CREATE\_AND\_DELETE, CSingletonConst::IS\_NOT\_THREAD\_SAFE);  //----------------------------------------  //通常シングルトンクラス用 static インスタンス生成用マクロ  //※.cpp ファイル中に記述する  //※シングルトン対象クラスのクラス名を渡す  #define MAKE\_SINGLETON\_INSTANCE(T) \  MAKE\_SINGLETON\_COMMON\_INSTANCE(T, CSingletonConst::AUTO\_SINGLETON\_TYPE) |

## シングルトンクラスの使用サンプル

シングルトンクラスの使用サンプルを示す。

シングルトンクラス、スレッドIDクラスを使用している箇所を赤字で示す。

**【インクルード】**

|  |
| --- |
| //----------------------------------------  //C++11ライブラリ  #include <random>//乱数  #include <chrono>//時間 |

**【シングルトン化対象クラス】**

|  |
| --- |
| //----------------------------------------  //共通処理クラス①：通常シングルトン用  class CData1  {  //コンストラクタ／デストラクタを private にするための friend 宣言  SINGLETON\_FRIEND(CData1);  public:  //定数  //シングルトン属性：自動生成のみ＋スレッドセーフ宣言  SINGLETON\_ATTR\_AUTO\_CREATE\_WITH\_THREAD\_SAFE();  //シングルトン属性：自動生成のみ＋非スレッドセーフ宣言  //SINGLETON\_ATTR\_AUTO\_CREATE\_WITHOUT\_THREAD\_SAFE();  //シングルトン属性：自動生成／自動削除＋スレッドセーフ宣言  //SINGLETON\_ATTR\_AUTO\_CREATE\_AND\_DELETE\_WITH\_THREAD\_SAFE();  //シングルトン属性：自動生成／自動削除＋非スレッドセーフ宣言  //SINGLETON\_ATTR\_AUTO\_CREATE\_AND\_DELETE\_WITHOUT\_THREAD\_SAFE();  public:  //アクセッサ  int getData() const { m\_data.load(); }  public:  //メソッド  //カウントアップ  void addData()  {  int data\_prev = m\_data.fetch\_add(1);  printf("addCount() %d -> %d\n", data\_prev, data\_prev + 1);  fflush(stdout);  }  //カウントダウン  void subData()  {  int data\_prev = m\_data.fetch\_sub(1);  printf("subCount() %d -> %d\n", data\_prev, data\_prev - 1);  fflush(stdout);  }  //プリント  void print(const char\* name, const char\* thread)  {  printf("print() Data=%2d [%s][%s] (FIRST:%s)\n", m\_data, name, thread, m\_firstThreadName);  fflush(stdout);  }  private:  //コンストラクタ  //※シングルトン以外でこのクラスを使えないように private にする  CData1(const char\* thread\_name):  m\_data(0),  m\_firstThreadName(thread\_name)  {  printf("[CONSTRUCTOR] (FIRST:%s)\n", m\_firstThreadName);  fflush(stdout);  }  //デストラクタ  ~CData1()  {  printf("[DESTRUCTOR] (FIRST:%s)\n", m\_firstThreadName);  fflush(stdout);  }  private:  std::atomic<int> m\_data;//データ  const char\* m\_firstThreadName;//最初にアクセスしたスレッド名  }; |

**【シングルトン化対象クラスのstaticメンバーをインスタンス化】**

|  |
| --- |
| //クラス内のstaticメンバーのインスタンスを定義する必要あり  MAKE\_SINGLETON\_INSTANCE(CData1);//通常シングルトン用staticインスタンス生成用マクロ |

**【テスト用処理】**

|  |
| --- |
| //----------------------------------------  //スレッド関数①-A  void threadFunc1A(const char\* thread\_name)  {  //スレッドIDとスレッド名をセット  CThreadID thread\_id(thread\_name);  //シングルトン  //※CData1 のコンストラクタ引数を指定  CSingleton<CData1> data(thread\_id.getName());  //乱数  std::random\_device rd;  std::mt19937 engine(rd());  std::uniform\_int\_distribution<int> sleep\_time(60, 100);  //スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(engine)));  //カウントアップ  //※アロー演算子でCData1のメンバー関数に普通にアクセス  data->addData();  //プリント  //※アロー演算子でCData1のメンバー関数に普通にアクセス  data->print("threadFunc1A", thread\_id.getName());  }  //----------------------------------------  //スレッド関数①-B  void threadFunc1B(const char\* thread\_name)  {  //スレッドIDとスレッド名をセット  CThreadID thread\_id(thread\_name);  //シングルトン  //※CData1 のコンストラクタ引数を指定  CSingleton<CData1> data(thread\_id.getName());  //スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  //乱数  std::random\_device rd;  std::mt19937 engine(rd());  std::uniform\_int\_distribution<int> sleep\_time(60, 100);  //スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(engine)));  //カウントダウン  //※アロー演算子でCData1のメンバー関数に普通にアクセス  data->subData();  //プリント  //※アロー演算子でCData1のメンバー関数に普通にアクセス  data->print("threadFunc1B", thread\_id.getName());  }  //----------------------------------------  //テスト①：通常シングルトンテスト  void test1()  {  printf("----------------------------------------------------------------------\n");  printf("【通常シングルトンテスト】\n");  //【コンパイルエラー】直接インスタンス生成  {  //普通にインスタンス生成  //CData1 data("illegal-data");//←NG:コンパイルエラー  //CData1\* data = new CData1("illegal-data");//←NG:コンパイルエラー  //※シングルトン対象クラスはコンストラクタが private 宣言されているので  //　直接インスタンスを生成できない  }  //スレッド生成  std::thread th1 = std::thread(threadFunc1A, "THREAD-A");  std::thread th2 = std::thread(threadFunc1B, "THREAD-B");  std::thread th3 = std::thread(threadFunc1A, "THREAD-C");  std::thread th4 = std::thread(threadFunc1B, "THREAD-D");  std::thread th5 = std::thread(threadFunc1A, "THREAD-E");  std::thread th6 = std::thread(threadFunc1B, "THREAD-F");  //スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  //途中状態のシングルトンの情報を表示  {  //スレッドID  CThreadID thread\_id;  //シングルトン  //※CData1 のコンストラクタ引数を指定  CSingleton<CData1> data(thread\_id.getName());  //プリント  //※アロー演算子でCData1のメンバー関数に普通にアクセス  data->print("main", thread\_id.getName());  //デバッグ情報表示  data.printDebugInfo();  }  //スレッド終了待ち  th1.join();  th2.join();  th3.join();  th4.join();  th5.join();  th6.join();  //終了状態のシングルトンの情報を表示  {  //スレッドID  CThreadID thread\_id;  //シングルトン  //※CData1 のコンストラクタ引数を指定  CSingleton<CData1> data(thread\_id.getName());  //プリント  //※アロー演算子でCData1のメンバー関数に普通にアクセス  data->print("main", thread\_id.getName());  //デバッグ情報表示  data.printDebugInfo();  }    printf("----------------------------------------------------------------------\n");  } |

**【テストメイン】**

|  |
| --- |
| //----------------------------------------  //テストメイン  int main(const int argc, const char\* argv[])  {  //メインスレッドIDとスレッド名をセット  CThreadID main\_thread\_id("MainThread");    //テスト①：通常シングルトン  test1();    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| ----------------------------------------------------------------------  【通常シングルトンテスト】  [CONSTRUCTOR] (FIRST:THREAD-A) ←CData1のコンストラクタ呼び出し（THREAD-Aスレッドが先着）  addCount() 0 -> 1  print() Data= 1 [threadFunc1A][THREAD-C] (FIRST:THREAD-A)  addCount() 1 -> 2  print() Data= 2 [threadFunc1A][THREAD-E] (FIRST:THREAD-A)  addCount() 2 -> 3  print() Data= 3 [threadFunc1A][THREAD-A] (FIRST:THREAD-A)  print() Data= 3 [main][MainThread] (FIRST:THREAD-A)  ----------------------------------------  Debug Info: [CData1] on "MainThread"(0x40dbbc18) ←[CData1]シングルトンのデバッグ情報表示（MainThreadで実行）  ClassAttribute = AUTO\_CREATE ←自動生成属性  ClassIsThreadSafe = IS\_THREAD\_SAFE ←スレッドセーフ宣言されている  ClassIsManaged = IS\_NORMAL\_SINGLETON ←通常シングルトン  ClassIsCreated = IS\_CREATED ←インスタンス生成済み  RefCount = 4 (max=6) ←現在4つの処理が同時アクセス中（最高6つだった）  RefCountOnThisThread = 1 ←現在のスレッドには1つの処理が存在  ThreadCount = 4 (max=6) ←現在4つのスレッドから同時アクセス中（最高6つだった）  CreatedThread = "THREAD-A"(0x0dfe7524) ←インスタンスを生成したスレッドは「THREAD-A」  ---------------------------------------- ※同一スレッド内の処理数が多いということは、  subCount() 3 -> 2 深い呼び出しのネストの中で何度も使用されているということ  print() Data= 2 [threadFunc1B][THREAD-B] (FIRST:THREAD-A) 　非効率な処理を疑うことができる。  subCount() 2 -> 1  print() Data= 1 [threadFunc1B][THREAD-D] (FIRST:THREAD-A)  subCount() 1 -> 0  print() Data= 0 [threadFunc1B][THREAD-F] (FIRST:THREAD-A)  print() Data= 0 [main][MainThread] (FIRST:THREAD-A) ←すべてのスレッドが問題なくシングルトンにアクセスできた  ---------------------------------------- （全てのスレッドが正常に動作すれば、Data の値が 0 になるテスト）  Debug Info: [CData1] on "MainThread"(0x40dbbc18)  ClassAttribute = AUTO\_CREATE  ClassIsThreadSafe = IS\_THREAD\_SAFE  ClassIsManaged = IS\_NORMAL\_SINGLETON  ClassIsCreated = IS\_CREATED  RefCount = 1 (max=6)  RefCountOnThisThread = 1  ThreadCount = 1 (max=6)  CreatedThread = "THREAD-A"(0x0dfe7524)  ---------------------------------------- ※処理は完了したが、CData1は消えない。  ---------------------------------------------------------------------- |

シングルトン対象クラスのシングルトン属性を「自動生成／削除」に変更した場合の実行結果を示す。（SINGLETON\_ATTR\_AUTO\_CREATE\_AND\_DELETE\_WITH\_THREAD\_SAFEを指定）

↓（実行結果）

|  |
| --- |
| ----------------------------------------------------------------------  【通常シングルトンテスト】  [CONSTRUCTOR] (FIRST:THREAD-A) ←CData1のコンストラクタ呼び出し（THREAD-Aスレッドが先着）  addCount() 0 -> 1  print() Data= 1 [threadFunc1A][THREAD-A] (FIRST:THREAD-A)  addCount() 1 -> 2  print() Data= 2 [threadFunc1A][THREAD-C] (FIRST:THREAD-A)  addCount() 2 -> 3  print() Data= 3 [threadFunc1A][THREAD-E] (FIRST:THREAD-A)  print() Data= 3 [main][MainThread] (FIRST:THREAD-A)  ----------------------------------------  Debug Info: [CData1] on "MainThread"(0x17476d44)  ClassAttribute = AUTO\_CREATE\_AND\_DELETE  ClassIsThreadSafe = IS\_THREAD\_SAFE  ClassIsManaged = IS\_NORMAL\_SINGLETON  ClassIsCreated = IS\_CREATED  RefCount = 4 (max=6)  RefCountOnThisThread = 1  ThreadCount = 4 (max=6)  CreatedThread = "THREAD-A"(0xaf998f2e) ←インスタンスを生成したスレッドは「THREAD-A」  ----------------------------------------  subCount() 3 -> 2  print() Data= 2 [threadFunc1B][THREAD-B] (FIRST:THREAD-A)  subCount() 2 -> 1  print() Data= 1 [threadFunc1B][THREAD-D] (FIRST:THREAD-A)  subCount() 1 -> 0  print() Data= 0 [threadFunc1B][THREAD-F] (FIRST:THREAD-A)  [DESTRUCTOR] (FIRST:THREAD-A) ←参照がなくなったので、シングルトン消滅  [CONSTRUCTOR] (FIRST:MainThread) ←再びアクセスがあったので再生成（MainThreadスレッド）  print() Data= 0 [main][MainThread] (FIRST:MainThread)  ----------------------------------------  Debug Info: [CData1] on "MainThread"(0x17476d44)  ClassAttribute = AUTO\_CREATE\_AND\_DELETE  ClassIsThreadSafe = IS\_THREAD\_SAFE  ClassIsManaged = IS\_NORMAL\_SINGLETON  ClassIsCreated = IS\_CREATED  RefCount = 1 (max=6)  RefCountOnThisThread = 1  ThreadCount = 1 (max=6)  CreatedThread = "MainThread"(0x17476d44) ←インスタンスを生成したスレッドが「MainThrad」に変わっている  ----------------------------------------  [DESTRUCTOR] (FIRST:MainThread) ←参照がなくなったので、シングルトン消滅  ---------------------------------------------------------------------- |

シングルトン対象クラスのシングルトン属性を「非スレッドセーフ宣言」に変更した場合の実行結果を示す。（SINGLETON\_ATTR\_AUTO\_CREATE\_WITHOUT\_THREAD\_SAFEを指定）

↓（実行結果）

|  |
| --- |
| ----------------------------------------------------------------------  【通常シングルトンテスト】 ※インスタンスを生成した THREAD-A 以外からアクセスすると、アサーション違反に  [CONSTRUCTOR] (FIRST:THREAD-A) 　なるが、処理はそのまま続行する。  Assertion failed! : m\_createdThreadId == INVALID\_THREAD\_ID || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE && m\_createdThreadId == this\_thread.getID()  test.cpp(1097)  CSingleton<T> is not thread safe. Create thread and this thread are different.  addCount() 0 -> 1  Assertion failed! : m\_createdThreadId == INVALID\_THREAD\_ID || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE && m\_createdThreadId == this\_thread.getID()  test.cpp(1097)  CSingleton<T> is not thread safe. Create thread and this thread are different.  print() Data= 1 [threadFunc1A][THREAD-A] (FIRST:THREAD-A)  Assertion failed! : m\_createdThreadId == INVALID\_THREAD\_ID || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE && m\_createdThreadId == this\_thread.getID()  test.cpp(1097)  CSingleton<T> is not thread safe. Create thread and this thread are different.  Assertion failed! : m\_createdThreadId == INVALID\_THREAD\_ID || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE && m\_createdThreadId == this\_thread.getID()  test.cpp(1097)  CSingleton<T> is not thread safe. Create thread and this thread are different.  addCount() 1 -> 2  print() Data= 1 [threadFunc1A][THREAD-C] (FIRST:THREAD-A)  Assertion failed! : m\_createdThreadId == INVALID\_THREAD\_ID || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE && m\_createdThreadId == this\_thread.getID()  test.cpp(1097)  CSingleton<T> is not thread safe. Create thread and this thread are different.  subCount() 2 -> 1  print() Data= 1 [threadFunc1B][THREAD-B] (FIRST:THREAD-A)  Assertion failed! : m\_createdThreadId == INVALID\_THREAD\_ID || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE && m\_createdThreadId == this\_thread.getID()  test.cpp(1097)  CSingleton<T> is not thread safe. Create thread and this thread are different.  subCount() 1 -> 0  print() Data= 1 [threadFunc1B][THREAD-D] (FIRST:THREAD-A)  print() Data= 1 [main][MainThread] (FIRST:THREAD-A)  ----------------------------------------  Debug Info: [CData1] on "MainThread"(0xbf50bc5b)  ClassAttribute = AUTO\_CREATE  ClassIsThreadSafe = IS\_NOT\_THREAD\_SAFE  ClassIsManaged = IS\_NORMAL\_SINGLETON  ClassIsCreated = IS\_CREATED  RefCount = 0 (max=1)  RefCountOnThisThread = 0  ThreadCount = 0 (max=1)  CreatedThread = "THREAD-A"(0x0e663599)  ----------------------------------------  addCount() 0 -> 1  print() Data= 1 [threadFunc1A][THREAD-E] (FIRST:THREAD-A)  subCount() 1 -> 0  print() Data= 0 [threadFunc1B][THREAD-F] (FIRST:THREAD-A)  Assertion failed! : m\_createdThreadId == INVALID\_THREAD\_ID || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_THREAD\_SAFE || THIS\_IS\_THREAD\_SAFE == CSingletonConst::IS\_NOT\_THREAD\_SAFE && m\_createdThreadId == this\_thread.getID()  test.cpp(1097)  CSingleton<T> is not thread safe. Create thread and this thread are different.  print() Data= 0 [main][MainThread] (FIRST:THREAD-A)  ----------------------------------------  Debug Info: [CData1] on "MainThread"(0xbf50bc5b)  ClassAttribute = AUTO\_CREATE  ClassIsThreadSafe = IS\_NOT\_THREAD\_SAFE  ClassIsManaged = IS\_NORMAL\_SINGLETON  ClassIsCreated = IS\_CREATED  RefCount = 0 (max=1)  RefCountOnThisThread = 0  ThreadCount = 0 (max=1)  CreatedThread = "THREAD-A"(0x0e663599)  ----------------------------------------  ---------------------------------------------------------------------- |

## 管理シングルトンクラス

ヘルパークラスを使用した管理シングルトンの実装例を示す。

なお、前述のリード・ライトロッククラスも使用している。

**【管理シングルトンテンプレートクラス】**

|  |
| --- |
| //----------------------------------------  //管理シングルトンテンプレートクラス  template<class T>  class CManagedSingleton : public CSingletonCommon<T, CSingletonConst::MANAGED\_SINGLETON\_TYPE>  {  //フレンドクラス  friend class CSingletonProxy<T>;  friend class CSingletonInitializer<T>;  friend class CSingletonUsing<T>;  public:  //シングルトン設定をテンプレート引数のクラスに基づいて静的に確定  //※クラス T には、定数 SINGLETON\_USING\_LIST\_MAX が定義されている必要がある  static const std::size\_t THIS\_SINGLETON\_USING\_LIST\_MAX = T::SINGLETON\_USING\_LIST\_MAX;//使用中処理リスト数    //【静的アサーション】T::SINGLETON\_USING\_LIST\_MAX は 1 以上の指定が必要  STATIC\_ASSERT(THIS\_SINGLETON\_USING\_LIST\_MAX > 0, "class THIS\_SINGLETON\_USING\_LIST\_MAX is under 0.");  public:  //アクセッサ  const char\* getInitializerName() const { return m\_initializerName.load(); }  //インスタンスを生成したイニシャライザー名取得  int getInitializerExists() const { return m\_initializerExists.load(); }//イニシャライザー数取得  CRWLock& getRWLock() { return m\_lock; }//リード・ライトロック取得  const char\* getDebugTrapName() const { return m\_debugTrapName.load(); }//デバッグ用トラップ対象処理名取得  void setDebugTrapName(const char\* name){ m\_debugTrapName.store(const\_cast<char\*>(name)); }  //デバッグ用トラップ対象処理名更新  void resetDebugTrapName(){ m\_debugTrapName.store(nullptr); }//デバッグ用トラップ対象処理名リセット  const char\* getDebugTrapThreadName() const { return m\_debugTrapThreadName.load(); }  //デバッグ用トラップ対象スレッド名取得  void setDebugTrapThreadName(const char\* name){ m\_debugTrapThreadName.store(const\_cast<char\*>(name)); }  //デバッグ用トラップ対象スレッド名更新  void resetDebugTrapThreadName(){ m\_debugTrapThreadName.store(nullptr); }  //デバッグ用トラップ対象スレッド名リセット  private:  void setInitializerName(const char\* name){ m\_initializerName.store(const\_cast<char\*>(name)); }  //インスタンスを生成したイニシャライザー名更新  void resetInitializerName(){ m\_initializerName.store(nullptr); }  //インスタンスを生成したイニシャライザー名リセット  int addInitializerExists(){ return m\_initializerExists.fetch\_add(1); }//イニシャライザー数をカウントアップ  int subInitializerExists(){ return m\_initializerExists.fetch\_sub(1); }//イニシャライザー数をカウントダウン  public:  //オペレータ  operator CRWLock& (){ return m\_lock; }//リード・ライトロックキャストオペレータ  private:  //メソッド  //使用中処理リストに処理情報を追加  //※処理名、スレッドID、スレッド名を渡す  void addUsingList(const char\* name, const THREAD\_ID thread\_id, const char\* thread\_name,  const CSingletonConst::E\_IS\_INITIALIZER is\_initializer)  {  //トラップ対象処理名＆スレッド名チェック  //※文字列の一致はポインタで判定（文字列リテラルが同じならポインタが一致する⇒コンパイルオプション依存）  const char\* trap\_name = m\_debugTrapName.load();  const char\* trap\_thread\_name = m\_debugTrapThreadName.load();  if ((trap\_name || trap\_thread\_name) &&  (!trap\_name || (name && name == trap\_name)) &&  (!trap\_thread\_name || (thread\_name && thread\_name == trap\_thread\_name)))  {  //メッセージ  DEBUG\_PRINT("Singleton catch the trap!! (\"%s\", Thread=\"%s\")\n", trap\_name, trap\_thread\_name);    //ブレークポイント  BREAK\_POINT();  }  //使用中情報作成  //※THIS\_SINGLETON\_USING\_LIST\_MAX で指定された数まで同時に記録可能  //　可変長テンプレート引数を活用したインスタンス生成  m\_thisUsingInfo = m\_usingListBuff.create<USING\_INFO>(m\_usingList.load(), name, thread\_id, thread\_name,  is\_initializer);  if (m\_thisUsingInfo)  {  //【参考】ロックフリーなスタックプッシュ（先頭ノード追加）アルゴリズム  //m\_thisUsingInfo->m\_next = m\_usingList.load();  //while (!m\_usingList.compare\_exchange\_weak(m\_thisUsingInfo->m\_next, m\_thisUsingInfo)){}  //※破棄処理側をロックフリーにできないので、このアルゴリズムを使用しない  //使用中情報リスト追加  m\_usingListLock.lock();//ロック取得  m\_thisUsingInfo->m\_next = m\_usingList.load();  m\_usingList.store(m\_thisUsingInfo);//先頭ノードに挿入（連結リスト）  m\_usingListLock.unlock();//ロック解放  }  }  //使用中処理リストから処理情報削除  //※m\_thisUsingInfo をリストから検索して削除  void deleteUsingList()  {  if (m\_thisUsingInfo)  {  //【参考】ロックフリーなスタックポップ（先頭ノード取り出し）アルゴリズム  //USING\_INFO\* old\_head = m\_usingList.load();  //while (old\_head && !m\_usingList.compare\_exchange\_weak(old\_head, old\_head->next));  //※この要件には適合しないので、ロックを使用する  //使用中情報リスト削除  m\_usingListLock.lock();//ロック取得  USING\_INFO\* now = m\_usingList.load();  if (now && now == m\_thisUsingInfo)  {  //先頭ノードの場合  m\_usingList.store(m\_thisUsingInfo->m\_next);  }  else  {  //先頭以降のノードの場合  while (now && now->m\_next != m\_thisUsingInfo){ now = now->m\_next; }  if (now)  now->m\_next = m\_thisUsingInfo->m\_next;  }  m\_usingListBuff.remove(m\_thisUsingInfo);//削除  m\_usingListLock.unlock();//ロック解放  }  }  public:  //使用中処理リスト表示  void printUsingList(const char\* name, FILE\* fp = stdout)  {  CThreadID thread\_id;  DEBUG\_FPRINT(fp, "----------------------------------------\n");  DEBUG\_FPRINT(fp, "Using List: [%s] by \"%s\" on \"%s\"(0x%08x)\n",  getClassName(), name, thread\_id.getName(), thread\_id.getID());  m\_usingListLock.lock();//ロック取得  USING\_INFO\* info = m\_usingList;  while (info)  {  DEBUG\_FPRINT(fp, " \"%s\" %s\ton \"%s\"(0x%08x)\n",  info->m\_name,  CSingletonConst::IsInitializer\_ToStr(info->m\_isInitializer ? CSingletonConst::IS\_INITIALIZER :  CSingletonConst::IS\_USING),  info->m\_threadName,  info->m\_threadId  );  info = info->m\_next;  }  DEBUG\_FPRINT(fp, "(num=%d, max=%d)\n", m\_usingListNum, m\_usingListNumMax);  m\_usingListLock.unlock();//ロック解放  DEBUG\_FPRINT(fp, "----------------------------------------\n");  DEBUG\_FFLUSH(fp);  }  //デバッグ情報表示  void printDebugInfo(const char\* name, FILE\* fp = stdout)  {  CThreadID thread\_id;  DEBUG\_FPRINT(fp, "----------------------------------------\n");  m\_instanceLock.lock();//ロック取得  DEBUG\_FPRINT(fp, "Debug Info: [%s] by \"%s\" on \"%s\"(0x%08x)\n",  getClassName(), name, thread\_id.getName(), thread\_id.getID());  DEBUG\_FPRINT(fp, " ClassAttribute = %s\n", getAttr\_Named());  DEBUG\_FPRINT(fp, " ClassIsThreadSafe = %s\n", isThreadSafe\_Named());  DEBUG\_FPRINT(fp, " ClassIsManaged = %s\n", isManagedSingleton\_Named());  DEBUG\_FPRINT(fp, " ClassIsCreated = %s\n", isCreated\_Named());  DEBUG\_FPRINT(fp, " RefCount = %d (max=%d)\n", getRefCount(), getRefCountMax());  DEBUG\_FPRINT(fp, " RefCountOnThisThread = %d\n", getRefCountOnThread());  DEBUG\_FPRINT(fp, " ThreadCount = %d (max=%d)\n", getThreadCount(), getThreadCountMax());  DEBUG\_FPRINT(fp, " CreatedThread = \"%s\"(0x%08x)\n",  getCreatedThreadName(), getCreatedThreadID());  DEBUG\_FPRINT(fp, " InitializerName = \"%s\"\n", getInitializerName());  DEBUG\_FPRINT(fp, " InitializerExists = %d\n", getInitializerExists());  DEBUG\_FPRINT(fp, " DebugTrap = \"%s\" on \"%s\"\n",  getDebugTrapName(), getDebugTrapThreadName());  m\_instanceLock.unlock();//ロック解放  DEBUG\_FPRINT(fp, "----------------------------------------\n");  DEBUG\_FFLUSH(fp);  }  private:  //コンストラクタ  //※コンストラクタで参照カウンタを更新せず、インスタンスを生成しない。  CManagedSingleton() :  CSingletonCommon<T, CSingletonConst::MANAGED\_SINGLETON\_TYPE>::CSingletonCommon(this)  {  }  //デストラクタ  //※デストラクタで自動的に参照カウンタをカウントダウンする。  ~CManagedSingleton()  {  }  private:  //構造体：使用中処理情報型  struct USING\_INFO  {  USING\_INFO\* m\_next;//次のノード  const char\* m\_name;//処理名  const THREAD\_ID m\_threadId;//処理スレッドのスレッドID  const char\* m\_threadName;//処理スレッドのスレッド名  bool m\_isInitializer;//イニシャライザーフラグ  //コンストラクタ  USING\_INFO(USING\_INFO\* next, const char\* name, const THREAD\_ID thread\_id, const char\* thread\_name,  const CSingletonConst::E\_IS\_INITIALIZER is\_initializer) :  m\_next(next),  m\_name(name),  m\_threadId(thread\_id),  m\_threadName(thread\_name),  m\_isInitializer(is\_initializer == CSingletonConst::IS\_INITIALIZER)  {  }  //デストラクタ  ~USING\_INFO()  {  }  };  private:  //フィールド  USING\_INFO\* m\_thisUsingInfo;//自身の使用中処理情報  static std::atomic<char\*> m\_initializerName;//インスタンスを生成したイニシャライザー名  static std::atomic<int> m\_initializerExists;//イニシャライザー数（普通は 0 か 1）  static CRWLock m\_lock;//リード・ライトロック  static std::atomic<char\*> m\_debugTrapName;//デバッグ用トラップ対象処理名  static std::atomic<char\*> m\_debugTrapThreadName;//デバッグ用トラップ対象スレッド名  static std::atomic<int> m\_usingListNum;//使用中処理リストの使用数  static std::atomic<int> m\_usingListNumMax;//使用中処理リストの使用数の最大到達値  static std::atomic<USING\_INFO\*> m\_usingList;//使用中処理リスト  static CSpinLock m\_usingListLock;//使用中処理リストのロック用フラグ  static CBlockAllocator<THIS\_SINGLETON\_USING\_LIST\_MAX, sizeof(USING\_INFO)> m\_usingListBuff;  //使用中処理リストの領域  }; |

**【管理シングルトンテンプレートクラス：補助マクロ】**

|  |
| --- |
| //----------------------------------------  //管理シングルトンクラス用 friend クラス指定補助マクロ  //※シングルトン対象クラス内に記述する  //※シングルトン対象クラス自身のクラス名を渡す  #define MANAGED\_SINGLETON\_FRIEND(T) \  SINGLETON\_COMMON\_FRIEND(T, CSingletonConst::MANAGED\_SINGLETON\_TYPE);  //----------------------------------------  //管理シングルトンクラス用シングルトン設定補助マクロ  //※シングルトン対象クラス内に記述する（privateスコープでもよい）  //※シングルトン属性とスレッドセーフ宣言を渡す  #define MANAGED\_SINGLETON\_ATTR(is\_thread\_safe, using\_list\_num) \  SINGLETON\_COMMON\_ATTR(CSingletonConst::ATTR\_MANUAL\_CREATE\_AND\_DELETE, is\_thread\_safe); \  static const std::size\_t SINGLETON\_USING\_LIST\_MAX = using\_list\_num;  //管理シングルトンクラス用シングルトン属性：手動生成＋スレッドセーフ宣言  #define MANAGED\_SINGLETON\_ATTR\_WITH\_THREAD\_SAFE(using\_list\_num) \  MANAGED\_SINGLETON\_ATTR(CSingletonConst::IS\_THREAD\_SAFE, using\_list\_num);  //管理シングルトンクラス用シングルトン属性：手動生成＋非スレッドセーフ宣言  #define MANAGED\_SINGLETON\_ATTR\_WITHOUT\_THREAD\_SAFE(using\_list\_num) \  MANAGED\_SINGLETON\_ATTR(CSingletonConst::IS\_NOT\_THREAD\_SAFE, using\_list\_num);  //----------------------------------------  //管理シングルトンクラス用 static インスタンス生成用マクロ  //※このマクロを直接使用せず、MAKE\_MANAGED\_SINGLETON\_INSTANCE\_ALL を使用する  #define MAKE\_MANAGED\_SINGLETON\_INSTANCE(T) \  std::atomic<char\*> CManagedSingleton<T>::m\_initializerName(nullptr);/\*インスタンスを生成したイニシャライザー名\*/\  std::atomic<int> CManagedSingleton<T>::m\_initializerExists(0);/\*イニシャライザー数\*/ \  CRWLock CManagedSingleton<T>::m\_lock;/\*リード・ライトロック\*/ \  std::atomic<char\*> CManagedSingleton<T>::m\_debugTrapName(nullptr);/\*デバッグ用トラップ対象処理名\*/ \  std::atomic<char\*> CManagedSingleton<T>::m\_debugTrapThreadName(nullptr);/\*デバッグ用トラップ対象スレッド名\*/ \  std::atomic<int> CManagedSingleton<T>::m\_usingListNum(0);/\*使用中処理リストの使用数\*/ \  std::atomic<int> CManagedSingleton<T>::m\_usingListNumMax(0);/\*使用中処理リストの使用数の最大到達値\*/ \  std::atomic< CManagedSingleton<T>::USING\_INFO\* > CManagedSingleton<T>::m\_usingList(nullptr);/\*使用中処理リスト\*/\  CSpinLock CManagedSingleton<T>::m\_usingListLock;/\*使用中処理リストのロック用フラグ\*/ \  CBlockAllocator<CManagedSingleton<T>::THIS\_SINGLETON\_USING\_LIST\_MAX, sizeof(CManagedSingleton<T>::USING\_INFO)> \  CManagedSingleton<T>::m\_usingListBuff;/\*使用中処理リストの領域\*/ |

**【ヘルパー：シングルトンプロキシーテンプレートクラス】**

|  |
| --- |
| //----------------------------------------  //【シングルトン用ヘルパー】シングルトンプロキシーテンプレートクラス　※継承専用  template<class T>  class CSingletonProxy  {  public:  //クラスの属性、動作モードをテンプレート引数に基づいて静的に決定  //※クラス T には、定数 CLASS\_NAME, SINGLETON\_ATTR, IS\_THREAD\_SAFE が定義されている必要がある。  //※クラス U には、定数 THIS\_IS\_MANAGED\_SINGLETON が定義されている必要がある。  static const CSingletonConst::E\_ATTR THIS\_SINGLETON\_ATTR = T::SINGLETON\_ATTR;//シングルトン属性  static const CSingletonConst::E\_IS\_THREAD\_SAFE THIS\_IS\_THREAD\_SAFE = T::THIS\_IS\_THREAD\_SAFE;//スレッドセーフ宣言  public:  //アクセッサ  const char\* getName() const { return m\_name; }//処理名取得  const CThreadID& getThreadID() const { return m\_threadId; }//現在のスレッドID取得  const char\* getClassName() const { return m\_singleton.getClassName(); }//現在のスレッド名取得　※委譲  CSingletonConst::E\_ATTR getAttr() const { return m\_singleton.getAttr(); }//シングルトン属性取得　※委譲  const char\* getAttr\_Named() const { return m\_singleton.getAttr\_Named(); }  //シングルトン属性名取得（デバッグ用）※委譲  CSingletonConst::E\_IS\_THREAD\_SAFE isThreadSafe() const { return m\_singleton.isThreadSafe(); }  //スレッドセーフ宣言取得　※委譲  const char\* isThreadSafe\_Named() const { return m\_singleton.isThreadSafe\_Named(); }  //スレッドセーフ宣言名取得（デバッグ用）　※委譲  CSingletonConst::E\_IS\_MANAGED\_SINGLETON isManagedSingleton() const { return m\_singleton.isManagedSingleton(); }  //管理シングルトン宣言取得　※委譲  const char\* isManagedSingleton\_Named() const { return m\_singleton.isManagedSingleton\_Named(); }  //管理シングルトン宣言名取得（デバッグ用）　※委譲  CSingletonConst::E\_IS\_CREATED isCreated() const { return m\_singleton.isCreated(); }  //クラス(T)インスタンス生成済み状態取得　※委譲  const char\* isCreated\_Named() const { return m\_singleton.isCreated\_Named(); }  //クラス(T)インスタンス生成済み状態名取得（デバッグ用）　※委譲  private:  CManagedSingleton<T>& getSinlgeton() { return m\_singleton; }//シングルトン取得  const CManagedSingleton<T>& getSinlgeton() const { return m\_singleton; }//constシングルトン取得  T\* getThis() { return m\_singleton.getThis(); }//クラス(T)インスタンスの参照を取得（禁止）　※委譲  const T\* getThis() const { return m\_singleton.getThis(); }  //クラス(T)インスタンスのconst参照を取得（禁止）　※委譲  public:  int getRefCount() const { return m\_singleton.getRefCount(); }//参照カウンタ取得　※委譲  int getRefCountMax() const { return m\_singleton.getRefCountMax(); }//参照カウンタの最大到達値を取得　※委譲  int getRefCountOnThread() const { return m\_singleton.getRefCountOnThread(); }  //現在のスレッド内の参照カウンタ数を取得　※委譲  int getThreadCount() const { return m\_singleton.getThreadCount(); }//参照スレッド数取得　※委譲  int getThreadCountMax() const { return m\_singleton.getThreadCountMax(); }  //参照スレッド数の最大到達値を取得　※委譲  THREAD\_ID getCreatedThreadID() const { return m\_singleton.getCreatedThreadID(); }  //クラス(T)インスタンスを生成したスレッドのスレッドIDを取得　※委譲  const char\* getCreatedThreadName() const { return m\_singleton.getCreatedThreadName(); }  //クラス(T)インスタンスを生成したスレッドのスレッド名を取得　※委譲  const char\* getInitializerName() const { return m\_singleton.getInitializerName(); }  //インスタンスを生成したイニシャライザー名取得　※委譲  int getInitializerExists() const { return m\_singleton.getInitializerExists(); }  //イニシャライザー数取得　※委譲  CRWLock& getRWLock() { return m\_singleton.getRWLock(); }//リード・ライトロック取得　※委譲  const char\* getDebugTrapName() const { return m\_singleton.getDebugTrapName(); }  //デバッグ用トラップ対象処理名取得　※委譲  void setDebugTrapName(const char\* name){ m\_singleton.setDebugTrapName(name); }  //デバッグ用トラップ対象処理名更新　※委譲  void resetDebugTrapName(){ m\_singleton.resetDebugTrapName(); }//デバッグ用トラップ対象処理名リセット　※委譲  const char\* getDebugTrapThreadName() const { return m\_singleton.getDebugTrapThreadName(); }  //デバッグ用トラップ対象スレッド名取得　※委譲  void setDebugTrapThreadName(const char\* name){ m\_singleton.setDebugTrapThreadName(name); }  //デバッグ用トラップ対象スレッド名更新　※委譲  void resetDebugTrapThreadName(){ m\_singleton.resetDebugTrapThreadName(); }  //デバッグ用トラップ対象スレッド名リセット　※委譲  public:  //メソッド  void printUsingList(FILE\* fp = stdout){ m\_singleton.printUsingList(m\_name, fp); }//使用中処理リスト表示　※委譲  void printDebugInfo(FILE\* fp = stdout){ m\_singleton.printDebugInfo(m\_name, fp); }//デバッグ情報表示　※委譲  public:  //オペレータ  T\* operator->(){ return m\_singleton; }  //アロー演算子：シングルトンクラスがクラス(T)のインスタンスを偽装（代理）する　※委譲  const T\* operator->() const { return m\_singleton; }//const アロー演算子：（同上）　※委譲  operator CRWLock& (){ return m\_singleton; }//リード・ライトロックキャストオペレータ　※委譲  protected:  //オペレータ（禁止）  T& operator\*(){ return \*m\_singleton; }//ポインタ演算子（禁止）　※委譲  const T& operator\*() const { return \*m\_singleton; }//constポインタ演算子（禁止）　※委譲  operator T\*(){ return m\_singleton; }//クラス T\* キャスト演算子（禁止）　※委譲  operator const T\*() const { return m\_singleton; }//クラスconst T\* キャスト演算子（禁止）　※委譲  operator T&(){ return \*m\_singleton; }//クラス T& キャスト演算子（禁止）　※委譲  operator const T&() const { return \*m\_singleton; }//クラス const T& キャスト演算子（禁止）　※委譲  CSingletonProxy<T>& operator=(const CSingletonProxy<T>&){ return \*this; }//コピー演算子（禁止）  CSingletonProxy<T>& operator=(const CSingletonProxy<T>\*){ return \*this; }//コピー演算子（禁止）  CSingletonProxy<T>& operator=(const T&){ return \*this; }//コピー演算子（禁止）  CSingletonProxy<T>& operator=(const T\*){ return \*this; }//コピー演算子（禁止）  private:  //コピーコンストラクタ（禁止）  explicit CSingletonProxy(CSingletonInitializer<T>&){}  explicit CSingletonProxy(CSingletonInitializer<T>\*){}  explicit CSingletonProxy(T&){}  explicit CSingletonProxy(T\*){}  private:  //デフォルトコンストラクタ（禁止）  CSingletonProxy(){}  protected:  //コンストラクタ  CSingletonProxy(const char\* name) :  m\_singleton(),  m\_name(name),  m\_threadId(),  m\_isAddRef(false)  {}  //デストラクタ  ~CSingletonProxy()  {}  protected:  //フィールド  CManagedSingleton<T> m\_singleton;//管理シングルトン  const char\* m\_name;//処理名  CThreadID m\_threadId;//スレッドID  bool m\_isAddRef;//参照カウンタカウントアップ済み  }; |

**【ヘルパー：シングルトンプロキシーテンプレートクラス：補助マクロ】**

|  |
| --- |
| //----------------------------------------  //シングルトンプロキシークラス用 static インスタンス生成用マクロ  //※このマクロを直接使用せず、MAKE\_MANAGED\_SINGLETON\_INSTANCE\_ALL を使用する  #define MAKE\_SINGLETON\_PROXY\_INSTANCE(T) \  // |

**【ヘルパー：シングルトンイニシャライザーテンプレートクラス】**

|  |
| --- |
| //----------------------------------------  //【シングルトン用ヘルパー】シングルトンイニシャライザーテンプレートクラス  template<class T>  class CSingletonInitializer : public CSingletonProxy<T>  {  //【静的アサーション】CSingletonInitializer<T>は、「手動生成属性：ATTR\_MANUAL\_CREATE\_AND\_DELETE」以外利用不可  STATIC\_ASSERT(THIS\_SINGLETON\_ATTR == CSingletonConst::ATTR\_MANUAL\_CREATE\_AND\_DELETE,  "CSingletonInitializer<T> is only supported ATTR\_MANUAL\_CREATE\_AND\_DELETE in THIS\_SINGLETON\_ATTR.");  public:  //アクセッサ  bool isExist() const { return CManagedSingleton<T>::isExistInitializer(); }//イニシャライザーが存在するか？  private:  void setInitializerName(const char\* name) { return m\_singleton.setInitializerName(name); }  //イニシャライザー名を更新　※委譲  void resetInitializerName() { return m\_singleton.resetInitializerName(); }//イニシャライザー名をリセット　※委譲  private:  //オペレータ（禁止）  CSingletonInitializer<T>& operator=(const CSingletonInitializer<T>&){ return \*this; }//コピー演算子（禁止）  CSingletonInitializer<T>& operator=(const CSingletonInitializer<T>\*){ return \*this; }//コピー演算子（禁止）  CSingletonInitializer<T>& operator=(const T&){ return \*this; }//コピー演算子（禁止）  CSingletonInitializer<T>& operator=(const T\*){ return \*this; }//コピー演算子（禁止）  private:  //コピーコンストラクタ（禁止）  explicit CSingletonInitializer(CSingletonInitializer<T>&){}  explicit CSingletonInitializer(CSingletonInitializer<T>\*){}  explicit CSingletonInitializer(T&){}  explicit CSingletonInitializer(T\*){}  public:  //メソッド  //初期化（手動インスタンス生成）  //※C++11仕様の可変長テンプレートを使用し、クラス T をインスタンス化する際に、  //　任意のパラメータを与えることを可能にしている。  //　C++11非対応のコンパイラでは、クラス T にデフォルトコンストラクタしか  //　使えないものとする。  template<typename... Tx>  bool initialize(Tx... nx)  {  //【アサーション】既にインスタンスが生成済みならアサーション違反（処理続行可）  ASSERT(!isCreated(),  "CSingletonInitializer<T> cannot create. Singleton instance is already exist.");    //生成（参照カウンタカウントアップ）  bool is\_created = false;//生成したか？  if (!m\_isAddRef)//まだ参照カウンタカをカウントアップしていない時に処理実行  {  m\_isAddRef = true;//参照カウンタカウントアップ状態ON  is\_created = m\_singleton.addRef(nx...);//生成（参照カウンタカウントアップ）  //生成されたか？  if (is\_created)  {  //関連情報をセット  setInitializerName(m\_name);//インスタンスを生成したイニシャライザー名  CRWLock& lock(getRWLock());  lock.setIgnoreThreadID(getThreadID().getID());//リード・ライトロックの対象スレッドIDを更新  }  }  //インスタンスが生成されたかどうかを返す  return is\_created;  }  //破棄（手動破棄）  bool finalize(CSingletonConst::E\_IS\_FORCED is\_forced = CSingletonConst::IS\_NORMAL){  return releaseCore(CSingletonConst::IS\_FINALIZE, is\_forced); }  //（明示的な）参照カウンタリリース　※破棄しない  bool release(){ return releaseCore(CSingletonConst::IS\_RELEASE, CSingletonConst::IS\_NORMAL); }  private:  //参照カウンタリリース（共通処理）  bool releaseCore(CSingletonConst::E\_IS\_FINALIZE is\_finalize, CSingletonConst::E\_IS\_FORCED is\_forced)  {  //リリース済みなら即終了  //※ファイナライズ時はリリース済みでも実行する  if (!m\_isAddRef && is\_finalize != CSingletonConst::IS\_FINALIZE)  return false;  //【アサーション】既に破棄済みならアサーション違反（処理続行可）  //※強制実行時はアサーション違反としない  ASSERT(isCreated() == CSingletonConst::IS\_CREATED ||  is\_forced == CSingletonConst::IS\_FORCED,  "CSingletonInitializer<T> cannot delete. Singleton instance is already deleted.");  //【アサーション】ファイナライズ時にまだが参照が残っているならアサーション違反（処理続行可）  //※強制時刻時はアサーション違反としない  const int LAST\_COUNT = getRefCount() - (m\_isAddRef ? 1 : 0);  ASSERT(is\_finalize == CSingletonConst::IS\_FINALIZE && LAST\_COUNT == 0 ||  is\_finalize != CSingletonConst::IS\_FINALIZE ||  is\_forced == CSingletonConst::IS\_FORCED,  "CSingletonInitializer<T> will finalize, yet still using singleton.");  //参照カウンタリリース  bool is\_deleted = false;//削除済みフラグ  if (m\_isAddRef || is\_finalize == CSingletonConst::IS\_FINALIZE)  //参照カウンタをカウントアップしていたか、強制手動破棄指定時に処理実行  {  m\_isAddRef = false;//参照カウンタカウントアップ状態OFF  if (is\_finalize == CSingletonConst::IS\_FINALIZE)//ファイナライズ（手動破棄）か？  is\_deleted = m\_singleton.deleteThis();//手動破棄  else  is\_deleted = m\_singleton.release(); //リリース  //破棄されたか？  if (is\_deleted)  {  //関連情報を破棄  resetInitializerName();//インスタンスを生成したイニシャライザー名  resetDebugTrapName();//デバッグ用トラップ対象処理名  resetDebugTrapThreadName();//デバッグ用トラップ対象スレッド名  }  }  //インスタンスが破棄されたかどうかを返す  return is\_deleted;  }  public:  //コンストラクタ  CSingletonInitializer(const char\* name, const CSingletonConst::E\_IS\_FORCED is\_forced =  CSingletonConst::IS\_NORMAL) :  CSingletonProxy(name),  m\_isFirst(false)  {  //イニシャライザー数カウントアップ  const int exists\_prev = m\_singleton.addInitializerExists();  //使用中処理リスト追加  m\_singleton.addUsingList(m\_name, m\_threadId.getID(), m\_threadId.getName(),  CSingletonConst::IS\_INITIALIZER);    //最初のイニシャライザーか判定  if (exists\_prev == 0)  {  m\_isFirst = true;//最初のイニシャライザーインスタンス  }  //【アサーション】他にもイニシャライザーがいるならアサーション違反（処理続行可）  //※強制破棄指定時はアサーション違反としない  ASSERT(m\_isFirst == true || is\_forced == CSingletonConst::IS\_FORCED,  "CSingletonInitializer<T>: already exist!");  }  //デストラクタ  ~CSingletonInitializer()  {  //イニシャライザー数カウントダウン  m\_singleton.subInitializerExists();    //使用中処理リストから破棄  m\_singleton.deleteUsingList();    //参照カウンタをリリース  release();  }  private:  //フィールド  bool m\_isFirst;//最初のイニシャライザーインスタンスか？  }; |

**【ヘルパー：シングルトンイニシャライザーテンプレートクラス：補助マクロ】**

|  |
| --- |
| //----------------------------------------  //シングルトンイニシャライザークラス用 static インスタンス生成用マクロ  //※このマクロを直接使用せず、MAKE\_MANAGED\_SINGLETON\_INSTANCE\_ALL を使用する  #define MAKE\_SINGLETON\_INITIALIZER\_INSTANCE(T) \  // |

**【ヘルパー：シングルトンアクセステンプレートクラス】**

|  |
| --- |
| //----------------------------------------  //【シングルトン用ヘルパー】シングルトンアクセステンプレートクラス  template<class T>  class CSingletonUsing : public CSingletonProxy<T>  {  //【静的アサーション】CSingletonInitializer<T>は、「手動生成属性：ATTR\_MANUAL\_CREATE\_AND\_DELETE」以外利用不可  STATIC\_ASSERT(THIS\_SINGLETON\_ATTR == CSingletonConst::ATTR\_MANUAL\_CREATE\_AND\_DELETE,  "CSingletonUsing<T> is only supported ATTR\_MANUAL\_CREATE\_AND\_DELETE in THIS\_SINGLETON\_ATTR.");  private:  //オペレータ（禁止）  CSingletonUsing<T>& operator=(const CSingletonUsing<T>&){ return \*this; }//コピー演算子（禁止）  CSingletonUsing<T>& operator=(const CSingletonUsing<T>\*){ return \*this; }//コピー演算子（禁止）  CSingletonUsing<T>& operator=(const T&){ return \*this; }//コピー演算子（禁止）  CSingletonUsing<T>& operator=(const T\*){ return \*this; }//コピー演算子（禁止）  private:  //コピーコンストラクタ禁止  explicit CSingletonUsing(CSingletonUsing<T>&){}  explicit CSingletonUsing(CSingletonUsing<T>\*){}  explicit CSingletonUsing(T&){}  explicit CSingletonUsing(T\*){}  public:  //コンストラクタ  CSingletonUsing(const char\* name) :  CSingletonProxy(name)  {  //【アサーション】インスタンスが存在していない時はアサーション違反  ASSERT(m\_singleton.isCreated() == CSingletonConst::IS\_CREATED,  "CSingletonUsing<T> cannot use. Singleton instance not exist.");  if (isCreated() != CSingletonConst::IS\_CREATED)  {  return;  }    //参照カウンタカウントアップ状態ON  m\_isAddRef = true;    //参照カウンタをカウントアップ  m\_singleton.addRefWitoutCreate();    //使用中処理リスト追加  m\_singleton.addUsingList(m\_name, m\_threadId.getID(), m\_threadId.getName(), CSingletonConst::IS\_USING);  }  //デストラクタ  ~CSingletonUsing()  {  if (m\_isAddRef)  {  //参照カウンタカウントアップ状態OFF  m\_isAddRef = false;    //参照カウンタをカウントダウン  m\_singleton.release();  //使用中処理リスト破棄  m\_singleton.deleteUsingList();  }  }  }; |

**【ヘルパー：シングルトンアクセステンプレートクラス：補助マクロ】**

|  |
| --- |
| //----------------------------------------  //シングルトンアクセスクラス用 static インスタンス生成用マクロ  //※このマクロを直接使用せず、MAKE\_MANAGED\_SINGLETON\_INSTANCE\_ALL を使用する  #define MAKE\_SINGLETON\_USING\_INSTANCE(T) \  // |

**【管理シングルトン＆ヘルパーテンプレートクラス：総合補助マクロ】**

|  |
| --- |
| //----------------------------------------  //シングルトンアクセスヘルパークラス用 static インスタンス生成用マクロ  //※.cpp ファイル中に記述する  //※シングルトン対象クラスのクラス名を渡す  #define MAKE\_MANAGED\_SINGLETON\_INSTANCE\_ALL(T) \  MAKE\_SINGLETON\_COMMON\_INSTANCE(T, CSingletonConst::MANAGED\_SINGLETON\_TYPE); \  MAKE\_MANAGED\_SINGLETON\_INSTANCE(T); \  MAKE\_SINGLETON\_INITIALIZER\_INSTANCE(T); \  MAKE\_SINGLETON\_USING\_INSTANCE(T); |

## 管理シングルトンクラスの使用サンプル

管理シングルトンクラスの使用サンプルを示す。

**【インクルード】**

|  |
| --- |
| //----------------------------------------  //C++11ライブラリ  #include <random>//乱数  #include <chrono>//時間 |

**【シングルトン化対象クラス】**

|  |
| --- |
| //----------------------------------------  //共通処理クラス②：管理シングルトン用  class CData2  {  //コンストラクタ／デストラクタを private にするための friend 宣言  MANAGED\_SINGLETON\_FRIEND(CData2);  public:  //定数  //シングルトン属性：手動生成＋スレッドセーフ宣言＋使用中情報数MAX=10件  MANAGED\_SINGLETON\_ATTR\_WITH\_THREAD\_SAFE(10);  //シングルトン属性：手動生成＋非スレッドセーフ宣言＋使用中情報数MAX=10件  //MANAGED\_SINGLETON\_ATTR\_WITHOUT\_THREAD\_SAFE(10);  public:  //アクセッサ  int getData() const { m\_data.load(); }  public:  //メソッド  //カウントアップ  void addData()  {  int data\_prev = m\_data.fetch\_add(1);  printf("addCount() %d -> %d\n", data\_prev, data\_prev + 1);  fflush(stdout);  }  //カウントダウン  void subData()  {  int data\_prev = m\_data.fetch\_sub(1);  printf("subCount() %d -> %d\n", data\_prev, data\_prev - 1);  fflush(stdout);  }  //プリント  void print(const char\* name, const char\* thread)  {  printf("print() Data=%2d [%s][%s] (FIRST:%s)\n", m\_data, name, thread, m\_firstThreadName);  fflush(stdout);  }  private:  //コンストラクタ  //※シングルトン以外でこのクラスを使えないように private にする  CData2(const char\* thread\_name) :  m\_data(0),  m\_firstThreadName(thread\_name)  {  printf("[CONSTRUCTOR] (FIRST:%s)\n", m\_firstThreadName);  fflush(stdout);  }  //デストラクタ  ~CData2()  {  printf("[DESTRUCTOR] (FIRST:%s)\n", m\_firstThreadName);  fflush(stdout);  }  private:  std::atomic<int> m\_data;//データ  const char\* m\_firstThreadName;//最初にアクセスしたスレッド名  }; |

**【シングルトン化対象クラスのstaticメンバーをインスタンス化】**

|  |
| --- |
| //クラス内のstaticメンバーのインスタンスを定義する必要あり  MAKE\_MANAGED\_SINGLETON\_INSTANCE\_ALL(CData2);//管理シングルトン用staticインスタンス生成用マクロ |

**【テスト用処理】**

|  |
| --- |
| //----------------------------------------  //スレッド関数②-A  void threadFunc2A(const char\* thread\_name)  {  //スレッドIDとスレッド名をセット  CThreadID thread\_id(thread\_name);  //シングルトンアクセス  //※処理名を指定  CSingletonUsing<CData2> data("threadFunc2A");  //乱数  std::random\_device rd;  std::mt19937 engine(rd());  std::uniform\_int\_distribution<int> sleep\_time(60, 100);  //スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(engine)));  //プリント  //※アロー演算子でCData2のメンバー関数に普通にアクセス  data->print("threadFunc2A:BEFORE", thread\_id.getName());  //リードロック  //※関数終了時に自動的にロック解放  CRWLockR lock(data);  //※管理シングルトンはリード・ライトロックを一つ保持している。  //　また、キャストオペレーターにより、そのままリード・ライトロックオブジェクトとして振る舞うことができる  //カウントアップ  //※アロー演算子でCData2のメンバー関数に普通にアクセス  data->addData();  //プリント  //※アロー演算子でCData2のメンバー関数に普通にアクセス  data->print("threadFunc2A:AFTER ", thread\_id.getName());  }  //----------------------------------------  //スレッド関数②-B  void threadFunc2B(const char\* thread\_name)  {  //スレッドIDとスレッド名をセット  CThreadID thread\_id(thread\_name);  //シングルトンアクセス  //※処理名を指定  CSingletonUsing<CData2> data("threadFunc2B");  //スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  //乱数  std::random\_device rd;  std::mt19937 engine(rd());  std::uniform\_int\_distribution<int> sleep\_time(60, 100);  //スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(engine)));  //プリント  //※アロー演算子でCData2のメンバー関数に普通にアクセス  data->print("threadFunc2B:BEFORE", thread\_id.getName());  //ライトロック  //※関数終了時に自動的にロック解放  CRWLockW lock(data);  //※管理シングルトンはリード・ライトロックを一つ保持している。  //　また、キャストオペレーターにより、そのままリード・ライトロックオブジェクトとして振る舞うことができる  //カウントダウン  //※アロー演算子でCData2のメンバー関数に普通にアクセス  data->subData();  //プリント  //※アロー演算子でCData2のメンバー関数に普通にアクセス  data->print("threadFunc2B:AFTER ", thread\_id.getName());  }  //----------------------------------------  //テスト②初期化関数  void initializeTest2()  {  //スレッドID  CThreadID thread\_id;    //シングルトンイニシャライザー  //※処理名を指定  CSingletonInitializer<CData2> data\_init("initializeTest2");    //インスタンス生成  //※CData2 のコンストラクタ引数を指定  data\_init.initialize(thread\_id.getName());  //デバッグ情報表示  data\_init.printDebugInfo();  data\_init.printUsingList();  }  //----------------------------------------  //テスト②終了関数  void finalizeTest2()  {  //シングルトンイニシャライザー  //※処理名を指定  CSingletonInitializer<CData2> data\_init("finalizeTest2");    //インスタンス破棄  data\_init.finalize();    //デバッグ情報表示  data\_init.printDebugInfo();  data\_init.printUsingList();  }  //----------------------------------------  //テスト②：管理シングルトンテスト  void test2()  {  printf("----------------------------------------------------------------------\n");  printf("【管理シングルトンテスト】\n");  //【コンパイルエラー】直接インスタンス生成  {  //普通にインスタンス生成  //CData2 data("illegal-data");//←NG:コンパイルエラー  //CData2\* data = new CData2("illegal-data");//←NG:コンパイルエラー  //※シングルトン対象クラスはコンストラクタが private 宣言されているので  //　直接インスタンスを生成できない  }  //【静的アサーション違反】通常シングルトンとして CData2 をインスタンス化  {  //通常シングルトン  //※CData2 のコンストラクタ引数を指定  //CSingleton<CData2> data("illegal-initialize");//←NG:静的アサーション違反  //※CData2は管理シングルトンとして宣言されているため、  //　通常シングルトンとして使用しようとするとコンパイルが通らない  //　しかし、安全ではあるものの、エラーの場所がかなりわかりにくいので、  //　静的アサーションをやめて実行時アサーションにしたほうが良いかもしれない  }  //【アサーション違反】初期化前にシングルトンにアクセス  {  //シングルトンアクセス  //※処理名を指定  CSingletonUsing<CData2> data("illegal-access");  }  //シングルトン初期化  initializeTest2();  //デバッグトラップ  //※指定の処理名とスレッド名が一致するシングルトンアクセスがあった場合、  //　ブレークポイントが発生する  //※どちらか片方の指定も可  {  CSingletonUsing<CData2> data("test");  data.setDebugTrapName("threadFunc2B");//処理名でトラップ  data.setDebugTrapThreadName("THREAD-F");//スレッド名でトラップ  }  //スレッド生成  std::thread th1 = std::thread(threadFunc2A, "THREAD-A");  std::thread th2 = std::thread(threadFunc2B, "THREAD-B");  std::thread th3 = std::thread(threadFunc2A, "THREAD-C");  std::thread th4 = std::thread(threadFunc2B, "THREAD-D");  std::thread th5 = std::thread(threadFunc2A, "THREAD-E");  std::thread th6 = std::thread(threadFunc2B, "THREAD-F");  //スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  //途中状態のシングルトンの情報を表示  {  //スレッドID  CThreadID thread\_id;  //シングルトンアクセス  //※処理名を指定  CSingletonUsing<CData2> data("main-1");  //プリント  //※アロー演算子でCData2のメンバー関数に普通にアクセス  data->print("main", thread\_id.getName());  //デバッグ情報表示  data.printDebugInfo();  data.printUsingList();  }  //スレッド終了待ち  th1.join();  th2.join();  th3.join();  th4.join();  th5.join();  th6.join();  //終了状態のシングルトンの情報を表示  {  //スレッドID  CThreadID thread\_id;  //シングルトンアクセス  //※処理名を指定  CSingletonUsing<CData2> data("main-2");  //プリント  //※アロー演算子でCData2のメンバー関数に普通にアクセス  data->print("main", thread\_id.getName());  //デバッグ情報表示  data.printDebugInfo();  data.printUsingList();  }  //シングルトン終了  finalizeTest2();    printf("----------------------------------------------------------------------\n");  } |

**【テストメイン】**

|  |
| --- |
| //----------------------------------------  //テストメイン  int main(const int argc, const char\* argv[])  {  //メインスレッドIDとスレッド名をセット  CThreadID main\_thread\_id("MainThread");  //テスト②：管理シングルトン  test2();    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| ----------------------------------------------------------------------  【管理シングルトンテスト】  Assertion failed! : m\_singleton.isCreated() == CSingletonConst::IS\_CREATED  test.cpp(1844)  CSingletonUsing<T> cannot use. Singleton instance not exist. ←シングルトン生成前にアクセスするとアサーション違反  [CONSTRUCTOR] (FIRST:MainThread) ←シングルトンイニシャライザーが明示的にインスタンスを生成  ---------------------------------------- ←[CData2]シングルトンのデバッグ情報表示  Debug Info: [CData2] by "initializeTest2" on "MainThread"(0x3dc23fc5) （MainThreadのinitializeTest2処理で実行）  ClassAttribute = MANUAL\_CREATE\_AND\_DELETE ←手動生成／削除属性  ClassIsThreadSafe = IS\_THREAD\_SAFE ←スレッドセーフ宣言  ClassIsManaged = IS\_MANAGED\_SINGLETON ←管理シングルトン宣言  ClassIsCreated = IS\_CREATED ←インスタンス生成済み  RefCount = 1 (max=1) ←現在1つの処理がアクセス中（最高1つ）  RefCountOnThisThread = 1 ←現在のスレッドには1つの処理が存在  ThreadCount = 1 (max=1) ←現在1つのスレッドからアクセス中（最高1つ）  CreatedThread = "MainThread"(0x40dbbc18) ←インスタンスを生成したスレッドは「MainThread」  InitializerName = "initializeTest2" ←インスタンスを生成した処理は「initializeTest2」  InitializerExists = 1 ←現在存在するイニシャライザーの数は1（普通は0か1）  DebugTrap = "(null)" on "(null)" ←デバッグトラップ名  ----------------------------------------  ---------------------------------------- ←[CData2]シングルトンの処理中リスト表示（MainThreadの  Using List: [CData2] by "initializeTest2" on "MainThread"(0x3dc23fc5) initializeTest2処理で実行）  "initializeTest2" IS\_INITIALIZER on "MainThread"(0x3dc23fc5)  (num=1, max=1)  ----------------------------------------  Singleton catch the trap!! ("threadFunc2B", Thread="THREAD-F") ←デバッグトラップ検出（ブレークポイント発生）  print() Data= 0 [threadFunc2A:BEFORE][THREAD-E] (FIRST:MainThread)  print() Data= 0 [threadFunc2A:BEFORE][THREAD-C] (FIRST:MainThread)  print() Data= 0 [threadFunc2A:BEFORE][THREAD-A] (FIRST:MainThread)←threadFunc2Aはリードロックにより同時実行  print() Data= 0 [main][MainThread] (FIRST:MainThread) 　できるが、threadFunc2Bは排他  ----------------------------------------  Debug Info: [CData2] by "main-1" on "MainThread"(0x3dc23fc5)  ClassAttribute = MANUAL\_CREATE\_AND\_DELETE  ClassIsThreadSafe = IS\_THREAD\_SAFE  ClassIsManaged = IS\_MANAGED\_SINGLETON  ClassIsCreated = IS\_CREATED  RefCount = 7 (max=7)  RefCountOnThisThread = 1  ThreadCount = 7 (max=7)  CreatedThread = "MainThread"(0x3dc23fc5)  InitializerName = "initializeTest2"  InitializerExists = 0  DebugTrap = "threadFunc2B" on "THREAD-F"  ----------------------------------------  ----------------------------------------  Using List: [CData2] by "main-1" on "MainThread"(0x3dc23fc5)  "main-1" IS\_USING on "MainThread"(0x3dc23fc5)  "threadFunc2B" IS\_USING on "THREAD-F"(0xa62e898b)  "threadFunc2A" IS\_USING on "THREAD-E"(0x25c6391a)  "threadFunc2B" IS\_USING on "THREAD-D"(0xe53ac732)  "threadFunc2A" IS\_USING on "THREAD-C"(0x9e69077a)  "threadFunc2B" IS\_USING on "THREAD-B"(0xe27a52fb)  "threadFunc2A" IS\_USING on "THREAD-A"(0x8282e39a)  (num=7, max=7)  ----------------------------------------  addCount() 1 -> 2  print() Data= 3 [threadFunc2A:AFTER ][THREAD-C] (FIRST:MainThread)  addCount() 0 -> 1  print() Data= 3 [threadFunc2A:AFTER ][THREAD-E] (FIRST:MainThread)  addCount() 2 -> 3  print() Data= 3 [threadFunc2A:AFTER ][THREAD-A] (FIRST:MainThread)  print() Data= 3 [threadFunc2B:BEFORE][THREAD-D] (FIRST:MainThread)←threadFunc2Bはライトロックにより完全に排他実行  subCount() 3 -> 2  print() Data= 2 [threadFunc2B:AFTER ][THREAD-D] (FIRST:MainThread)  print() Data= 2 [threadFunc2B:BEFORE][THREAD-B] (FIRST:MainThread) ←threadFunc2Bはライトロックにより完全に排他実行  subCount() 2 -> 1  print() Data= 1 [threadFunc2B:AFTER ][THREAD-B] (FIRST:MainThread)  print() Data= 1 [threadFunc2B:BEFORE][THREAD-F] (FIRST:MainThread) ←threadFunc2Bはライトロックにより完全に排他実行  subCount() 1 -> 0  print() Data= 0 [threadFunc2B:AFTER ][THREAD-F] (FIRST:MainThread)  print() Data= 0 [main][MainThread] (FIRST:MainThread)  ----------------------------------------  Debug Info: [CData2] by "main-2" on "MainThread"(0x3dc23fc5)  ClassAttribute = MANUAL\_CREATE\_AND\_DELETE  ClassIsThreadSafe = IS\_THREAD\_SAFE  ClassIsManaged = IS\_MANAGED\_SINGLETON  ClassIsCreated = IS\_CREATED  RefCount = 1 (max=7)  RefCountOnThisThread = 1  ThreadCount = 1 (max=7)  CreatedThread = "MainThread"(0x3dc23fc5)  InitializerName = "initializeTest2"  InitializerExists = 0  DebugTrap = "threadFunc2B" on "THREAD-F"  ----------------------------------------  ----------------------------------------  Using List: [CData2] by "main-2" on "MainThread"(0x3dc23fc5)  "main-2" IS\_USING on "MainThread"(0x3dc23fc5)  (num=1, max=7)  ----------------------------------------  [DESTRUCTOR] (FIRST:MainThread) ←シングルトンイニシャライザーが明示的にインスタンスを破棄  ----------------------------------------  Debug Info: [CData2] by "finalizeTest2" on "MainThread"(0x3dc23fc5)  ClassAttribute = MANUAL\_CREATE\_AND\_DELETE  ClassIsThreadSafe = IS\_THREAD\_SAFE  ClassIsManaged = IS\_MANAGED\_SINGLETON  ClassIsCreated = IS\_NOT\_CREATED  RefCount = 0 (max=7)  RefCountOnThisThread = 0  ThreadCount = 0 (max=7)  CreatedThread = "(null)"(0xffffffff)  InitializerName = "(null)"  InitializerExists = 1  DebugTrap = "(null)" on "(null)"  ----------------------------------------  ----------------------------------------  Using List: [CData2] by "finalizeTest2" on "MainThread"(0x3dc23fc5)  "finalizeTest2" IS\_INITIALIZER on "MainThread"(0x3dc23fc5)  (num=1, max=7)  ----------------------------------------  ---------------------------------------------------------------------- |

■■以上■■

索引

**索引項目が見つかりません。**

効率化と安全性のためのロック制御

以　上