効率化と安全性のためのロック制御

– より最適なマルチスレッドプログラミングのために –

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

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。

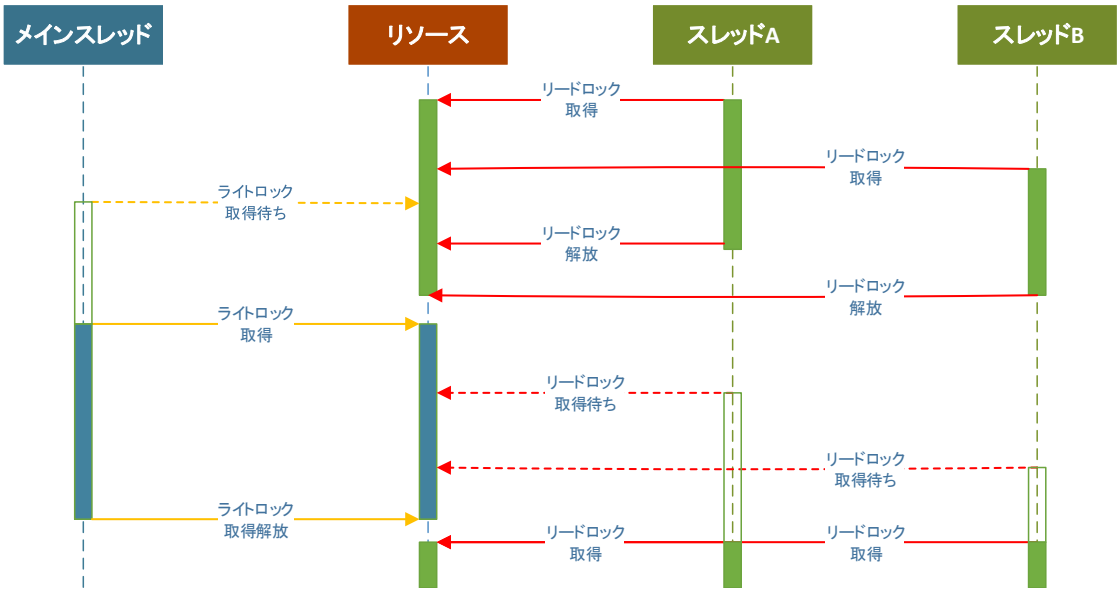
▼ 要件定義

本件の具体的な処理要件を定義する。

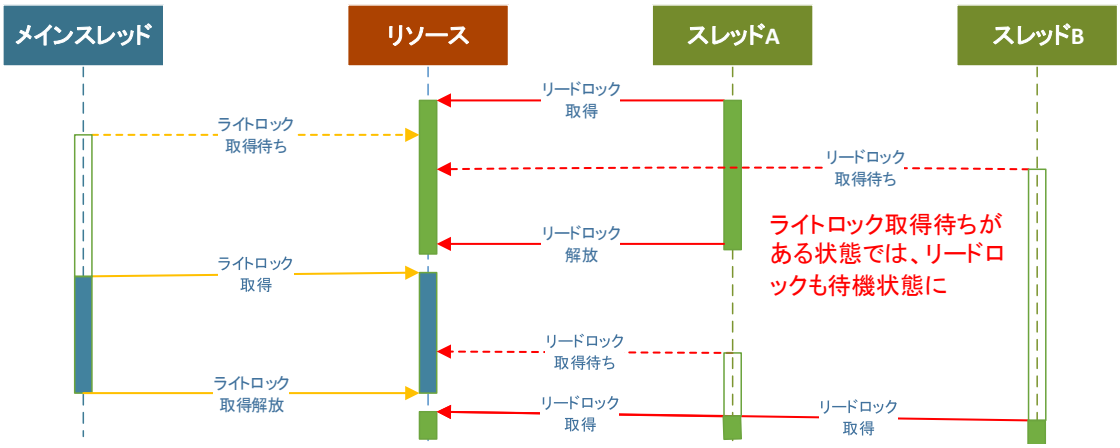
- プログラミング構造上、リソースアクセス時は必然的にロックする。
- プログラミング構造上、リソースアクセス時のロック解放は自動的に行われる。
  - C++11 のミューテックスロック用クラス「std::lock_guard」と同様に、デストラクタでロック解放する仕組みとする。
    - 処理ブロック（スコープ）から抜ける時にデストラクタが呼び出されてロックを解放する。「スコープドロックパターン」（Scoped Lock Pattern）と呼ばれるプログラミング手法。
  - これにより、コーディングミスや、例外発生、不測の return があっても確実にロック解放される。
  - 明示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。

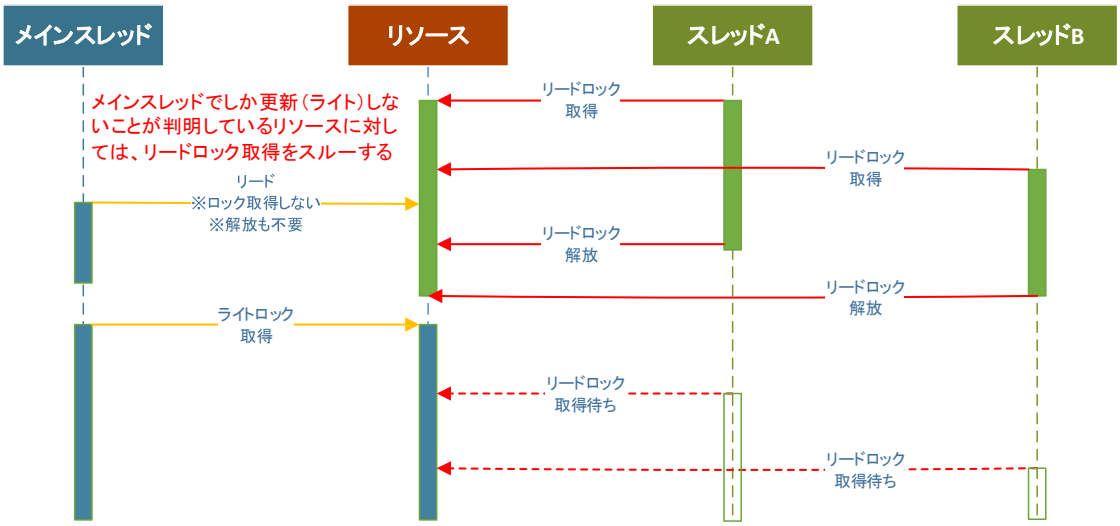
リード・ライトロックの動作イメージ：



ライトロック優先動作のイメージ：



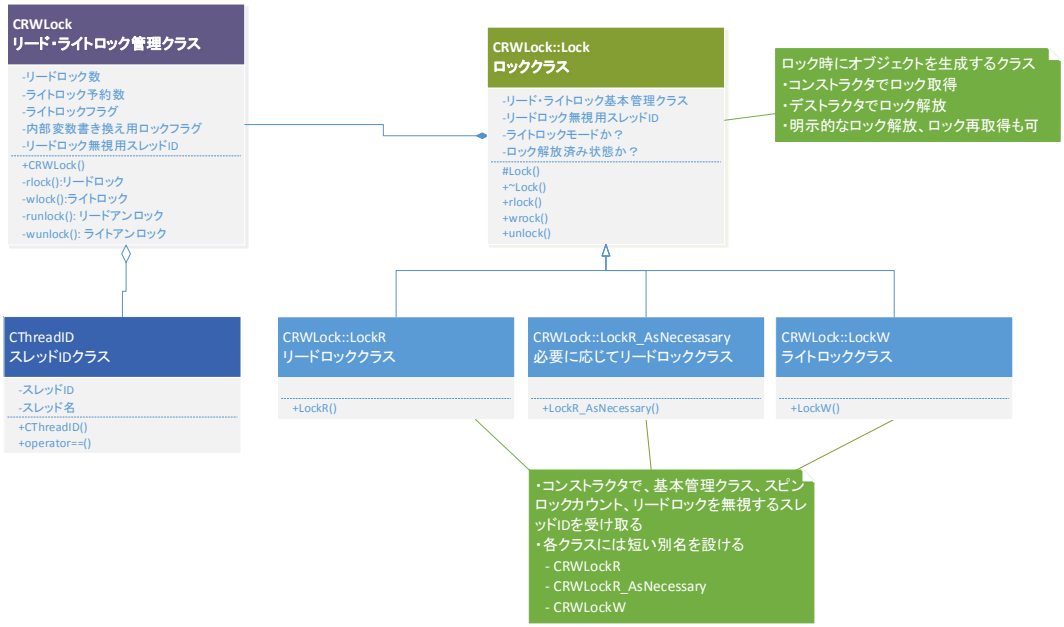
メインスレッドがリードロックをスルーする動作のイメージ：



▼ 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。

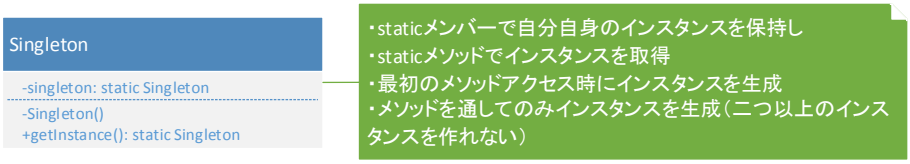
- 任意のクラスに対して適用可能な、汎用的なシングルトンパターンのテンプレートクラスを作成し、シングルトンの仕組みを標準化する。
- プロキシーパターンの性質を兼ね備え、対象クラスの機能を完全に透過的に利用できるものとする。
- あらかじめスタティックなメモリ領域を確保しておき、動的メモリを使用しない。
  - ヒープが足りない状況でも確実にインスタンスを生成できる。
  - いつ初期化されても、「初期化のタイミングを変えたらヒープメモリが断片化した」といった類いの問題を起こさない。
- スタティック領域を使用しつつも、初期化・終了処理のタイミング（順序）を任意にコントロールできるものとする。
- 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

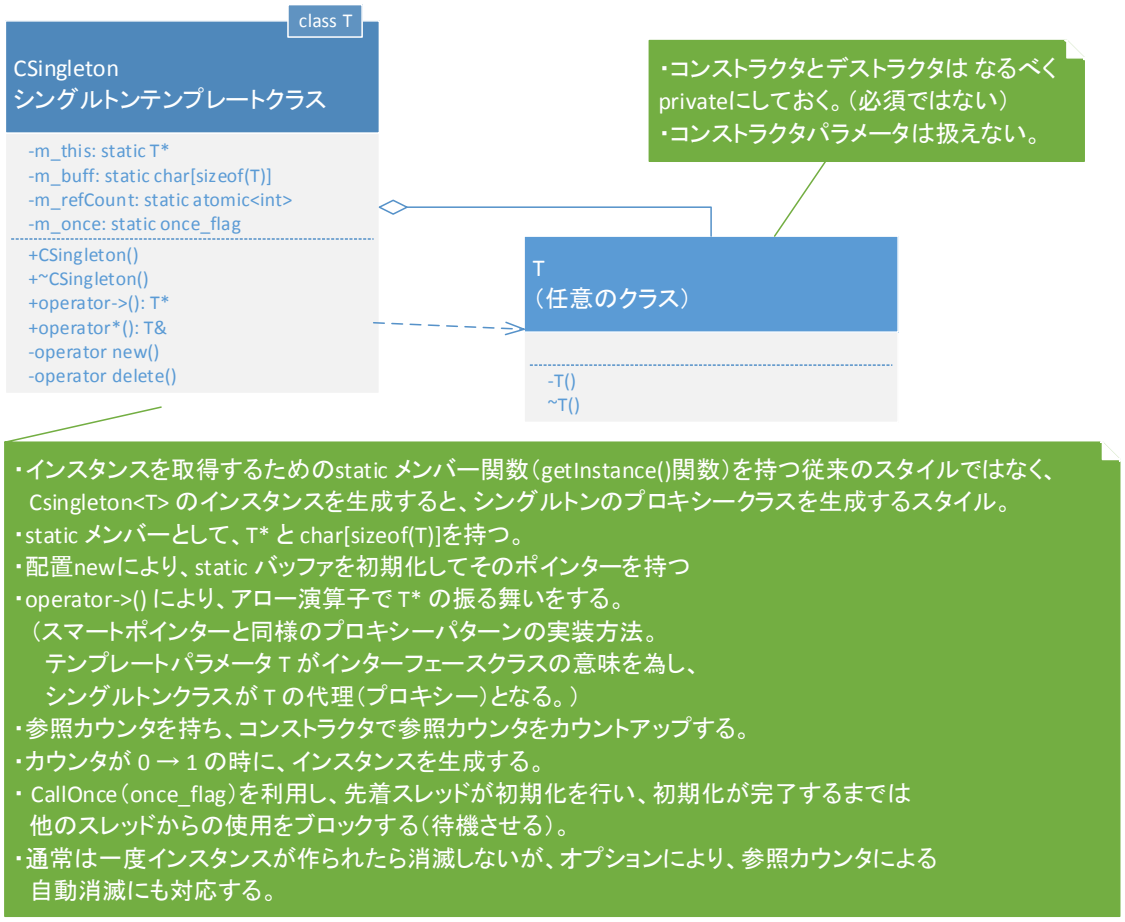
要件定義に基づいて、クラスを設計する。
まずは一般的なシングルトンパターンを示す。

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:

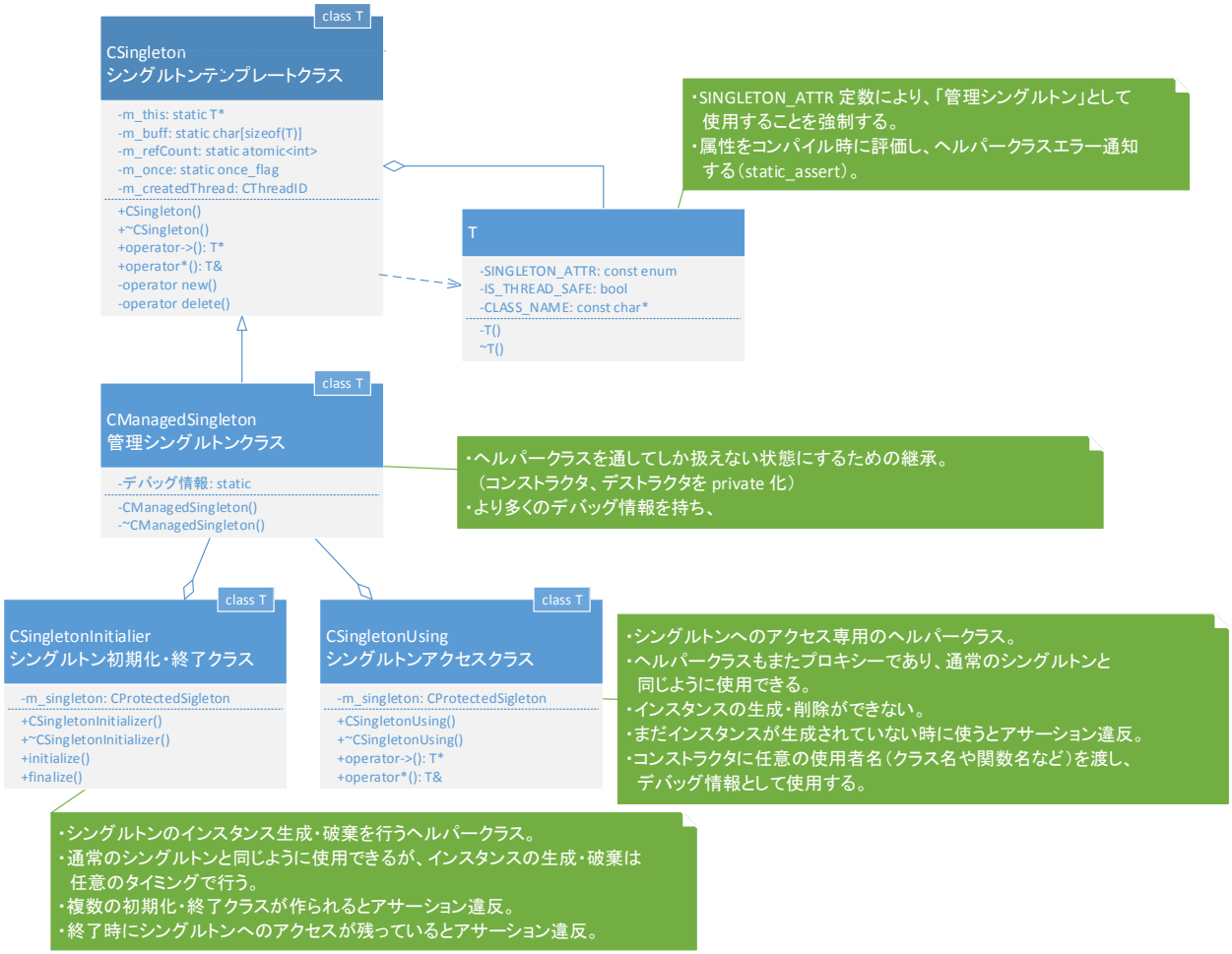


これにより、いつアクセスがあってもスレッドセーフなインスタンス生成を行う事ができる。

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