スクリプトの生産性向上のための名前付きデータ参照

– 画一化された手続きによるデータ参照手法 –

2014 年 2 月 24 日　初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 24 日	板垣 衛	（初稿）

■ 目次

■ 概略

スクリプトやデバッグコマンドの機能追加を行い易くするために、「名前」でデータの参照が可能な画一的なプログラミングインターフェースを設計する。

また、マルチスレッドでの利用も考慮し、安全性とパフォーマンスを確保する。

■ 目的

本書は、スクリプトやデバッグコマンドの機能追加要件に迅速に対応できるプログラミング環境を構築することにより、全体的な生産性の向上を目的とする。

なお、本書が示すプログラムのサンプルは、別紙の「効率化と安全性のためのロック制御」に記載した「リード・ライトロック」の仕組みを使用する。

本書自体が、その別紙の延長線上にある内容である。

■ 名前付きデータ参照の設計

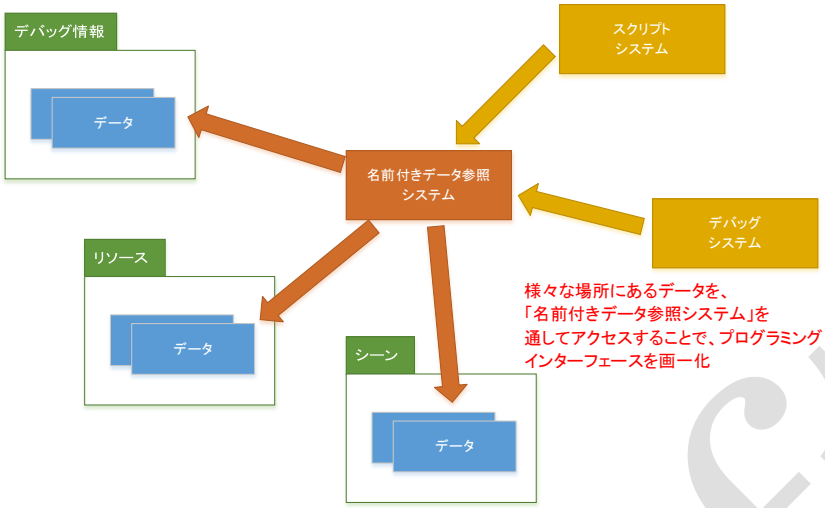
▼ 【目的】データ参照に対するインターフェースの画一化

様々なデータに対するアクセスのための共通インターフェースを設計し、アクセス手法を画一化する。

基本的に、「名前」でデータを指定してアクセスする。

「名前」とデータの関連づけの必要はあるが、それだけで済むため、データアクセス要件が追加されるたびにスクリプト用関数の追加やデバッグ機能のための専用関数の追加といった手間を軽減する。

画一化されたデータアクセスのイメージ：



▼ 要件定義

本件の具体的な処理要件を定義する。

- 「名前」と「変数」との関連づけを行い、名前で変数の読み込みと書き込みができるものとする。
  - グローバル変数、static 変数と言った静的変数だけでなく、リソースデータなどのインスタンスの情報も扱えるものとする。
- 「名前」とアクセッサ処理との関連づけを行い、名前で取得処理／更新処理を呼び出せるものとする。
- 高速アクセスのために、一度変数を参照したら、そのポインタを保持するものとする。
  - ただし、参照を保持し続けるとメモリ再配置やデータ削除で問題を起こすので、それを任意にリセットできるものとする。
  - 保持している参照をまとめてリセットする機能を設け、毎フレーム、ゲームループのメモリ再配置処理前に実行する。
- リード・ライトロックを標準で備え、マルチスレッドでの安全なアクセスを保証するものとする。
- 構造体へのアクセスに対応し、構造体へのアクセス中はロックを維持するものとする。
- ロックを効率化するために、四則演算、インクリメント、デクリメント、ビット論理演算に対応するものとする。

・ データアクセスの際、名前に加えて「シーン ID」などの任意の副キーを使用可能とする。

▼ プログラミングイメージ

画一化されたデータ参照のプログラミングのイメージを示す。

【データ取得】

```
CNamedRefSingleton named_ref;//名前付きデータ参照シングルトン取得
                             //※RefK()はキー生成用のクラス。名前をキーにする以外に、副キーの指定も可。
bool enabled = named_ref->getBool(RefK("DebugCpuMeter"));//CPU 使用率表示 ON/OFF 状態を取得(bool 値)
int hp = named_ref->getInt(RefK("CharaInfo::HP", chara_id));//指定のキャラ ID（副キー）のキャラ情報の HP を取得
{
     CRefR<CCharaInfo> obj(named_ref, RefK("CharaInfo", chara_id));
                                                  //指定のキャラ ID のキャラ情報（構造体）に読み込みアクセス
     hp = obj->hp; //構造体のメンバーにはアロー演算子でアクセス
     str = obj->str;
}//処理ブロックを抜ける時に自動的にロック解除
```

【データ更新】

```
CNamedRefSingleton named_ref;//名前付きデータ参照シングルトン取得
named_ref->set(RefK("DebugCpuMeter"), true);//CPU 使用率表示 ON/OFF 状態を更新(bool 値)
named_ref->setInt(RefK("CharaInfo::HP", chara_id), 123);//指定のキャラ ID のキャラ情報の HP を更新
{
     CRefW<CCharaInfo> obj(named_ref, RefK("CharaInfo", chara_id));
                                                  //指定のキャラ ID のキャラ情報（構造体）に書き込みアクセス
     obj->hp += 10; //構造体のメンバーにはアロー演算子でアクセス
     obj->mp += 5;
}//処理ブロックを抜ける時に自動的にロック解除
```

【データ参照の登録①：変数直接参照タイプ】

```
struct refDebugCpuMeter{ typedef bool TYPE; TYPE* operator()(const RefK& key) const { return &TEST::s_debugCpuMeter; } };
                                                  //CPU 使用率表示状態参照用関数オブジェクト
CNamedRefSingleton named_ref;//名前付きデータ参照シングルトン取得
named_ref->regist(RefK("DebugCpuMeter"), refDebugCpuMeter());//名前と変数参照用の関数オブジェクトの関連を登録
```

【データ参照の登録②：構造体参照タイプ】

```
struct refCharaInfo{ typedef CCharaInfo TYPE; TYPE* operator()(const RefK& key) const {
                                                      return searchCharaInfo(key.m_sub); } };
                                                  //キャラ情報参照用関数オブジェクト
                                                  //※内部でキャラ情報をキャラ ID で検索する関数を使用
CNamedRefSingleton named_ref;//名前付きデータ参照シングルトン取得
named_ref->regist(RefK("CharaInfo", chara_id), refCharaInfo());//名前と構造体参照用の関数オブジェクトの関連を登録
```

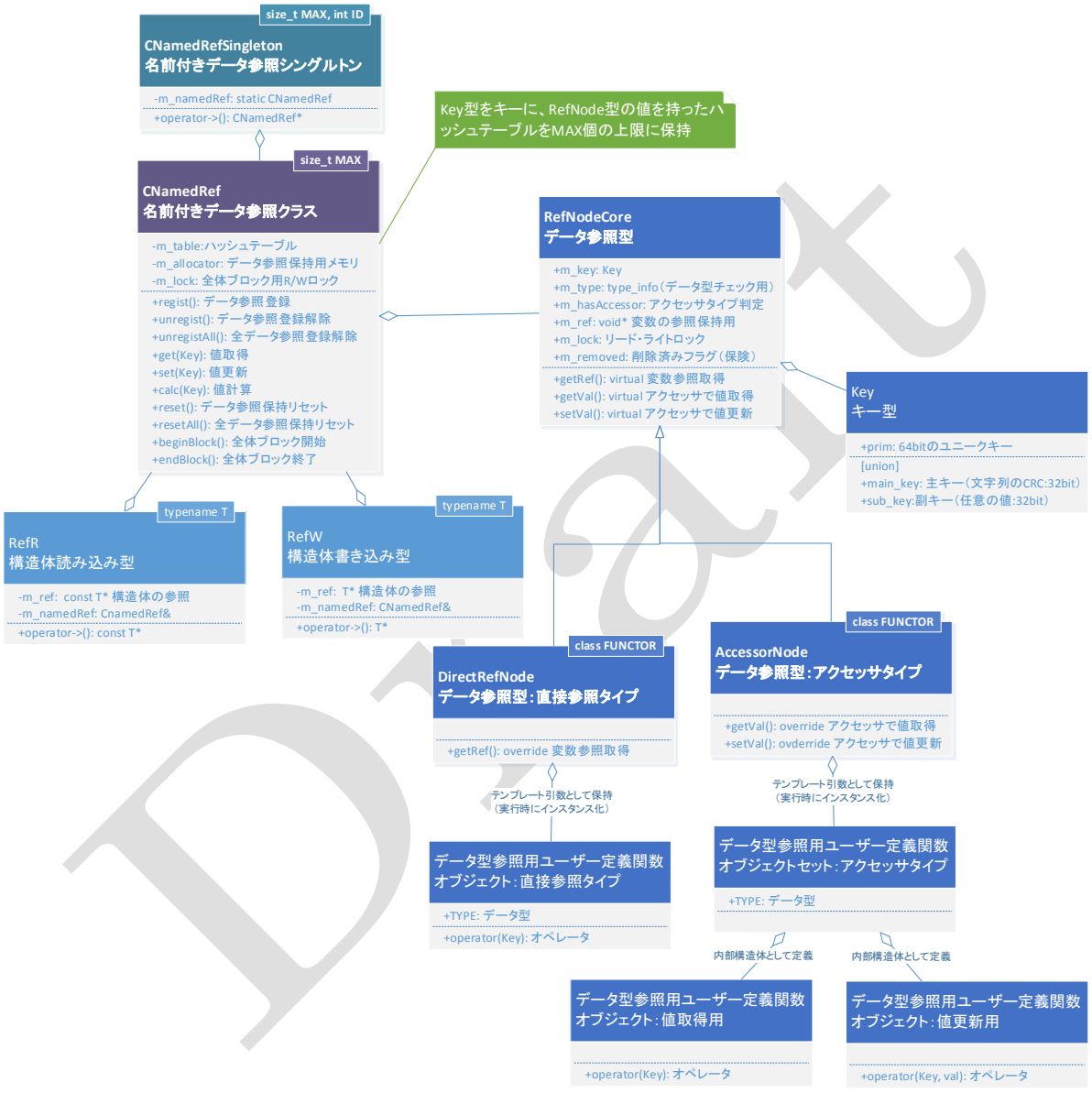
【データ参照の登録③：アクセッサタイプ】

```
//シーンオブジェクトの HP アクセス用関数オブジェクト
struct refSceneObjHP{
     typedef int TYPE;
     typedef CCharaInfo OBJ;
     static OBJ* search(const RefK& key){ return searchCharaInfo(key.m_sub); }
     //取得用処理
     struct getter{ TYPE operator()(const RefK& key) const { const OBJ* obj = search(key);
                                                      return obj ? obj->getHP() : 0; } };
     //更新用処理
     struct setter{ void operator()(const RefK& key, const TYPE val) { OBJ* obj = search(key);
                                                      obj->setHP(val); } };
};
CNamedRefSingleton named_ref;//名前付きデータ参照シングルトン取得
named_ref->registAcc(RefK("CharaInfo::HP"), refCharaInfoHP());//名前とアクセッサ処理用の関数オブジェクトの関連を登録
```

▼ クラス設計

要件定義に基づいて、名前付きデータ参照のクラスを設計する。

名前付きデータ参照のクラス図：



▼ 処理上の注意

リソースデータなど、メモリ再配置や途中削除の可能性があるデータを参照する場合、削除処理／メモリ再配置処理と並行して参照が行われることがないように注意。

このクラス自体はスレッドセーフであり、値取得と更新のアトミック性は保証するが、

参照先データの状態変化（再配置／削除）との衝突を防ぐものではない。
この問題の対応として、下記の機能を実装する。

・ メインループのメモリ再配置／削除処理のタイミングで、「名前付きデータ参照」へのアクセス全体をブロックする。
  - 全体ブロックは、ハッシュテーブルに対するリード・ライトロックで対応する。
  - 基本処理として、ハッシュテーブルの追加・削除が発生する時はライトロックを行い、それ以外のアクセスはリードロックを行う。
  - 任意の期間（メモリ再配置／削除処理中）の全体ブロックは、このリード・ライトロックを使用し、明示的にライトロック取得することで実現する。
  - 単純に全体ブロック（明示的なライトロック）を行うと、ブロック中に一切の操作が出来なくなってしまうので、明示的ライトロックを行ったスレッドは、一切の操作でロックされないものとする。

・ メインループのメモリ再配置／削除処理のタイミングで、「名前付きデータ参照」が内部で保持する参照先のポインタを全てリセットする。
  - 全体ブロック解放時に同時に自動実行する。

■ 処理実装サンプル

名前付きデータ参照クラスの実装サンプルを示す。
Visual C++ 2013 で完全に動作し、C++11 仕様に依存したコードである。

▼ 【前提】共通処理コード

別紙の「効率化と安全性のためのロック制御」に記載したソースコードを使用する。
「基本マクロ」、「スレッド ID クラス」、「軽量スピンロッククラス」、「プールアロケータクラス」、「リード・ライトロッククラス」については、別紙参照。

▼ 名前付きデータ参照クラス

要件に基づく実装例を示す。
暫定で C++11 標準ライブラリのハッシュテーブルクラス unordered_map をそのまま使用する。実際にゲームで使用する際は、ヒープを使用しない独自の処理に実装を変えるか、Allocator を定義した方が良い。また、別紙の「様々なメモリ管理手法と共通アロケー

タインターフェース」で説明する、「多態性を持ったグローバル new / delete」を用いるやり方も有効。

unordered_map 使用箇所を赤字で表記する。

【インクルード】

```
#include <atomic>//アトミック型
#include <unordered_map>//【暫定】C++11 標準ライブラリのハッシュテーブルライブラリ
```

【名前付きデータ参照クラス】

```
//---------------------------------------
//名前付きデータ参照クラス
//※テンプレートパラメータで記録可能な参照の最大数を指定
template<std::size_t MAX, int SPIN = CSpinLock::DEFAULT_SPIN_COUNT>
class CNamedRefT
{
public:
	//定数
	static const int MAX_NODES = MAX;//記録可能な参照の最大数を指定
	static const int SPIN_COUNT = SPIN;//スピンロックのピンカウント数
public:
	//型
	typedef unsigned long long KeyPrim;//キー型（ハッシュテーブル用のプリミティブ型）
	//キー型
	struct Key
	{
		union
		{
			KeyPrim m_prim;//プリミティブ型キー
			struct
			{
				unsigned int m_main;//主キー
				unsigned int m_sub;//副キー
			};
		};
		//キャストオペレータ
		operator KeyPrim() const { return m_prim; }
		//比較オペレータ
		bool operator==(const Key& key) const { return m_prim == key.m_prim; }
		bool operator!=(const Key& key) const { return m_prim != key.m_prim; }
		bool operator>(const Key& key) const { return m_prim > key.m_prim; }
		bool operator<(const Key& key) const { return m_prim < key.m_prim; }
		bool operator>=(const Key& key) const { return m_prim >= key.m_prim; }
		bool operator<=(const Key& key) const { return m_prim <= key.m_prim; }
		//代入オペレータ
		Key& operator=(const Key& key) const { return m_prim = key.m_prim; }
		//メソッド
		//CRC 値算出関数
		unsigned int CRC32(const char* name)
		{
			//ダミー処理
			return reinterpret_cast<unsigned long>(name);
		}
		//コンストラクタ
		//※explicit 宣言しない ⇒ Key 型を引数に取る関数に文字列を渡しても OK
		Key(const Key& key) :
			m_prim(key.m_prim)
		{}
		Key(const KeyPrim key_prim) :
			m_prim(key_prim)
		{}
		Key(const unsigned int main_key, const unsigned int sub_key) :
			m_main(main_key),
			m_sub(sub_key)
```

```
            {}
            Key(const unsigned int key) :
                  m_main(key),
                  m_sub(0)
            {}
            Key(const char* name, const unsigned int sub_key):
                  m_main(CRC32(name)),
                  m_sub(sub_key)
            {}
            Key(const char* name) :
                  m_main(CRC32(name)),
                  m_sub(0)
            {}
            Key(const unsigned int main_key, const char* sub_key) :
                  m_main(main_key),
                  m_sub(CRC32(sub_key))
            {}
            Key(const char* name, const char* sub_key) :
                  m_main(CRC32(name)),
                  m_sub(CRC32(sub_key))
            {}
            //デストラクタ
            ~Key()
            {}
      };
private:
      //--------------------
      //データ参照型：共通部
      struct RefNodeCore
      {
            //メソッド
            virtual void* getRefByFunctor() const = 0;//関数オブジェクトを使用して変数の参照を取得
            virtual void getValByFunctor(void* val) const = 0;//関数オブジェクトを使用して変数の値を取得
            virtual void setValByFunctor(const void* val) = 0;//関数オブジェクトを使用して変数の値を更新
            //データの参照を取得
            template<typename T>
            void* getRefCore() const
            {
                  if (m_removed.load() == true)//【保険処理】削除済みチェック
                        return nullptr;
                  ASSERT(typeid(T) == m_type, "This type is not match.(this=%s, request=%s)",
                                                                        m_type.name(), typeid(T).name());
                  if (typeid(T) != m_type)
                        return nullptr;
                  if (m_ref.load() == nullptr)//２回目以降のアクセスを高速にするために、参照を記憶する
                        m_ref.store(getRefByFunctor());
                  return m_ref.load();
            }
            //データの参照を取得
            template<typename T>
            T* getRef(){ return static_cast<T*>(getRefCore<T>()); }
            //データの参照を取得
            template<typename T>
            const T* getRef() const { return static_cast<T*>(getRefCore<T>()); }
            //データの値を取得
            template<typename T>
            T getVal() const
            {
                  if (m_removed.load() == true)//【保険処理】削除済みチェック
                        return static_cast<T>(0);
                  if (m_hasAccessor)
                  {
                        //アクセッサがある場合
                        ASSERT(typeid(T) == m_type, "This type is not match.");
                        if (typeid(T) != m_type)
```

```
                        return static_cast<T>(0);
                    T val = static_cast<T>(0);
                    getValByFunctor(reinterpret_cast<void*>(&val));
                    return val;
                }
                //アクセッサがない場合
                void* p = getRefCore<T>();
                return p == nullptr ? static_cast<T>(0) : *static_cast<T*>(p);
            }
            //データの値を更新
            template<typename T>
            void setVal(const T val)
            {
                if (m_removed.load() == true)//【保険処理】削除済みチェック
                    return;
                if (m_hasAccessor)
                {
                    //アクセッサがある場合
                    ASSERT(typeid(T) == m_type, "This type is not match.");
                    if (typeid(T) != m_type)
                        return;
                    setValByFunctor(reinterpret_cast<const void*>(&val));
                }
                //アクセッサがない場合
                void* p = getRefCore<T>();
                if (p){ *static_cast<T*>(p) = val; }
            }
            //記憶している参照をリセット
            //※メモリ再配置や参照先の削除があったら呼び出す必要あり
            void resetRef(){ m_ref.store(nullptr); }
            //リードロック取得
            void rLock() const { CRWLockHelper(m_lock).rLock(SPIN_COUNT); }
            //ライトロック取得
            void wLock() const { CRWLockHelper(m_lock).wLock(SPIN_COUNT); }
            //リードロック解放
            void rUnlock() const { CRWLockHelper(m_lock).rUnlock(); }
            //ライトロック解放
            void wUnlock() const { CRWLockHelper(m_lock).wUnlock(); }
            //コンストラクタ
            RefNodeCore(const Key& key, const type_info& type, const bool has_accessor) :
                m_key(key),
                m_type(type),
                m_hasAccessor(has_accessor),
                m_ref(nullptr),
                m_lock()
            {
                m_removed.store(false);
            }
            //デストラクタ
            virtual ~RefNodeCore()
            {
                m_removed.store(true);//【保険処理】削除済みにする
                m_ref.store(nullptr);//【念のため】変数の参照をクリア
            }
            //フィールド
            const Key m_key;//キー
            const type_info& m_type;//型情報　※不正アクセス検出用
            const bool m_hasAccessor;//アクセッサタイプか？
            std::atomic<bool> m_removed;//【保険用】削除済みフラグ
            mutable std::atomic<void*> m_ref;//変数の参照
            mutable CRWLock m_lock;//リード・ライトロックオブジェクト
        };
        //--------------------
        //データ参照型：実装部：直接参照タイプ
        //※直接参照タイプは、一度関数オブジェクトで変数の参照を取得したらそれを保持し、
```

```
	//  関数オブジェクトの再実行を行わないため、２回目以降のアクセスが早い。
	//※明示的に「参照をリセット」すると、次回また関数オブジェクトを実行する。
	template <typename F>
	struct DirectRefNode : public RefNodeCore
	{
		//型
		typedef typename F::TYPE TYPE;//データ型
		typedef typename F NAMED_REF_FUNCTOR;//変数の参照取得用関数オブジェクト型
		//【オーバーライド】関数オブジェクトを使用して変数の参照を取得
		void* getRefByFunctor() const override
		{
			NAMED_REF_FUNCTOR functor;
			return functor(m_key);
		}
		//【オーバーライド】【無効】関数オブジェクトを使用して変数の値を取得
		void getValByFunctor(void* val) const override {}
		//【オーバーライド】【無効】関数オブジェクトを使用して変数の値を更新
		void setValByFunctor(const void* val) override {}
		//コンストラクタ
		DirectRefNode(const Key& key) :
			RefNodeCore(key, typeid(TYPE), false)
		{}
		//デストラクタ
		~DirectRefNode() override
		{}
	};
	//--------------------
	//データ参照型：実装部：アクセッサタイプ
	//※直接参照タイプの方が、参照を記憶できる分早い。
	//※アクセッサタイプは、毎回必ずアクセッサを通す。
	template<typename F>
	struct AccessorNode : public RefNodeCore
	{
		//型
		typedef typename F::TYPE TYPE;//データ型
		typedef typename F::getter GETTER_FUNCTOIR;//データの値取得用関数オブジェクト型
		typedef typename F::setter SETTER_FUNCTOIR;//データの値更新用関数オブジェクト型
		//【オーバーライド】【無効】関数オブジェクトを使用して変数の参照を取得
		void* getRefByFunctor() const override{ return nullptr; }
		//【オーバーライド】関数オブジェクトを使用してデータの値を取得
		void getValByFunctor(void* val) const override
		{
			GETTER_FUNCTOIR functor;
			*static_cast<TYPE*>(val) = functor(m_key);
		}
		//【オーバーライド】関数オブジェクトを使用してデータの値を更新
		void setValByFunctor(const void* val) override
		{
			SETTER_FUNCTOIR functor;
			functor(m_key, *static_cast<const TYPE*>(val));
		}
		//コンストラクタ
		AccessorNode(const Key& key) :
			RefNodeCore(key, typeid(TYPE), true)
		{}
		//デストラクタ
		~AccessorNode() override
		{}
	};
public:
	//--------------------
	//関数オブジェクト用基本型
	//※必ず継承して使用する
	template<typename T>
	struct BaseRefF
```

```
	{
		typedef typename T TYPE;//データ型定義
	};
private:
	//--------------------
	//メモリサイズ計算用のダミー関数オブジェクト
	struct getRefFunctorDummy : BaseRefF<int>{ TYPE* operator()(const Key& key) const { return nullptr; } };
                                                    //【ダミー】データ参照用関数オブジェクト：直接参照タイプ
	struct accessorFunctorDummy : BaseRefF<int>{//【ダミー】データ参照用関数オブジェクト：アクセッサタイプ
		struct getter { TYPE operator()(const Key& key) const { return 0; } };
		struct setter { void operator()(const Key& key, const TYPE val) const {} };
	};
	using DirectRefNodeDummy = DirectRefNode<getRefFunctorDummy>;//【ダミー】データ参照型：直接参照タイプ
	using AccessorNodeDummy = AccessorNode<accessorFunctorDummy>;//【ダミー】データ参照型：アクセッサタイプ
public:
	//定数
	#define MAX_SIZE(x, y) ((x) > (y) ? (x) : (y))
	static const std::size_t REF_NODE_SIZE_MAX = MAX_SIZE(sizeof(DirectRefNodeDummy), sizeof(AccessorNodeDummy));
                                                                                    //データ参照型の最大サイズ
	#undef MAX_SIZE
private:
	//型
	//--------------------
	//ハッシュテーブル型
	//【暫定】C++11 標準ライブラリのハッシュテーブル型を使用
	//※できれば、ヒープを使用しない独自の実装に変えた方が安定する
	using NamedRefTable = std::unordered_map<KeyPrim, RefNodeCore*>;
public:
	//クラス内クラス定義
	//※構造体参照用

	//--------------------
	//変数読み込み参照用クラス：構造体参照用
	template<typename T>
	class RefR
	{
	public:
		//オペレータ
		const T* operator->() const { return m_ref; }//アロー演算子（本来のデータ型のプロキシー）
		T operator*() const { return *m_ref; }//ポインタ型
	public:
		//メソッド
		bool isExist() const { return m_isExist; }//参照先が存在するか？
	public:
		//コンストラクタ
		RefR(CNamedRefT<MAX>& map, const Key key) :
			m_namedRef(map.find(key, &m_isExist)),
			m_ref(nullptr)
		{
			//変数の参照を取得
			if (m_namedRef)
			{
				m_ref = m_namedRef->getRef<T>();
				if (m_ref)
				{
					//変数の参照に成功したらリードロック取得
					m_namedRef->rLock();
				}
			}
		}
		//デストラクタ
		~RefR()
		{
			if (m_namedRef && m_ref)
			{
```

```
                        //リードロック解放
                        m_namedRef->rUnlock();
                }
        }
    private:
        //フィールド
        const RefNodeCore* m_namedRef;//変数参照型
        const T* m_ref;//変数参照
        bool m_isExist;//参照先が存在するか？
    };
    //--------------------
    //変数書き込み参照用クラス：構造体参照用
    template<typename T>
    class RefW
    {
    public:
        //オペレータ
        T* operator->(){ return m_ref; }//アロー演算子（本来のデータ型のプロキシー）
        const T* operator->() const { return m_ref; }//アロー演算子（本来のデータ型のプロキシー）
        T operator*() const { return *m_ref; }//ポインタ型
    public:
        //メソッド
        bool isExist() const { return m_isExist; }//参照先が存在するか？
    public:
        //コンストラクタ
        RefW(CNamedRefT<MAX>& map, const Key key) :
            m_namedRef(map.find(key, &m_isExist)),
            m_ref(nullptr)
        {
            //変数の参照を取得
            if (m_namedRef)
            {
                m_ref = m_namedRef->getRef<T>();
                if (m_ref)
                {
                    //変数の参照に成功したらライトロック取得
                    m_namedRef->wLock();
                }
            }
        }
        //デストラクタ
        ~RefW()
        {
            if (m_namedRef && m_ref)
            {
                //ライトロック解放
                m_namedRef->wUnlock();
            }
        }
    private:
        //フィールド
        RefNodeCore* m_namedRef;//変数参照型
        T* m_ref;//変数参照
        bool m_isExist;//参照先が存在するか？
    };
private:
    //--------------------
    //全体ブロック：読み込みロック
    class WholeBLockR
    {
    public:
        //コンストラクタ
        WholeBLockR(CRWLock& lock, const bool locked, const THREAD_ID locked_thread_id) :
            m_lock(lock)
        {
```

```
                //ロック中状態
                //※全体ブロック中、かつ、全体ブロックを行ったスレッドと同じスレッドならロックしない
                m_locked = !(locked == true && CThreadID() == locked_thread_id);
                if (m_locked)
                    CRWLockHelper(m_lock).rLock(SPIN_COUNT);
            }
            //デストラクタ
            ~WholeBLockR()
            {
                if (m_locked)
                    CRWLockHelper(m_lock).rUnlock();
            }
        private:
            CRWLock& m_lock;//リード・ライトロック
            bool m_locked;//ロック中
        };
        //--------------------
        //全体ブロック：書き込みロック
        class WholeBLockW
        {
        public:
            //コンストラクタ
            WholeBLockW(CRWLock& lock, const bool locked, const THREAD_ID locked_thread_id) :
                m_lock(lock)
            {
                //ロック中状態
                //※全体ブロック中、かつ、全体ブロックを行ったスレッドと同じスレッドならロックしない
                m_locked = !(locked == true && CThreadID() == locked_thread_id);
                if (m_locked)
                    CRWLockHelper(m_lock).wLock(SPIN_COUNT);
            }
            //デストラクタ
            ~WholeBLockW()
            {
                if (m_locked)
                    CRWLockHelper(m_lock).wUnlock();
            }
        private:
            CRWLock& m_lock;//リード・ライトロック
            bool m_locked;//ロック中
        };
        //クラス内クラス定義：ここまで
        //--------------------
    public:
        //メソッド
        //変数参照用関数オブジェクト登録：直接参照タイプ
        template<typename F>
        bool regist(const Key key, F functor, bool* is_exist = nullptr)
        {
            //全体ライトロック取得（解放は自動）
            WholeBLockW lock(m_wholeLock, m_wholeLocked.load(), m_wholeLockThreadID.load());
            //ハッシュテーブル登録済みチェック
            NamedRefTable::const_iterator node = m_namedRefList.find(key);
            if (node != m_namedRefList.end())
            {
                if (is_exist)
                    *is_exist = true;
                return false;
            }
            if (is_exist)
                *is_exist = false;
            //ノードのメモリ確保
            DirectRefNode<F>* ref = m_allocator.create< DirectRefNode<F> >(key);
            if (!ref)
                return false;
```

```
		//ハッシュテーブルに追加登録
		m_namedRefList.insert(std::make_pair(key, ref));
		return true;
	}
	//変数参照用関数オブジェクト登録：アクセッサタイプ
	template<typename F>
	bool registAcc(const Key key, F functor_set, bool* is_exist = nullptr)
	{
		//全体ライトロック取得（解放は自動）
		WholeBLockW lock(m_wholeLock, m_wholeLocked.load(), m_wholeLockThreadID.load());
		//ハッシュテーブル登録済みチェック
		NamedRefTable::const_iterator node = m_namedRefList.find(key);
		if (node != m_namedRefList.end())
		{
			if (is_exist)
				*is_exist = true;
			return false;
		}
		if (is_exist)
			*is_exist = false;
		//ノードのメモリ確保
		AccessorNode<F>* ref = m_allocator.create< AccessorNode<F> >(key);
		if (!ref)
			return false;
		//ハッシュテーブルに追加登録
		m_namedRefList.insert(std::make_pair(key, ref));
		return true;
	}
	//変数参照用関数オブジェクト登録解除
	//※スレッドセーフではないのでタイミングに厳重注意
	//※念のためロックを取得してはいるものの、他からのアクセスをブロックしている状況だと、
	//  削除後に処理が走るので結局問題を起こす
	bool unregist(const Key key, bool* is_exist = nullptr)
	{
		//全体ライトロック取得（解放は自動）
		WholeBLockW lock(m_wholeLock, m_wholeLocked.load(), m_wholeLockThreadID.load());
		//変数参照情報取得
		//ハッシュテーブル登録済みチェック
		NamedRefTable::iterator node = m_namedRefList.find(key);
		if (node == m_namedRefList.end())
		{
			if (is_exist)
				*is_exist = false;
			return false;
		}
		if (is_exist)
			*is_exist = true;
		//登録解除と削除
		RefNodeCore* ref = node->second;//ハッシュテーブルの情報を参照
		if (ref)
		{
			//ref->wLock();//【不要】ライトロック取得
			node->second = nullptr;//ハッシュテーブルから情報を破棄
			//ref->wUnlock();//【不要】ライトロック解放
			m_allocator.destroy(ref);//メモリ解放
		}
		m_namedRefList.erase(key);//ハッシュテーブルから登録解除
		return true;
	}
	//全ての変数参照用関数オブジェクト登録解除
	//※スレッドセーフではないのでタイミングに厳重注意
	//※念のためロックを取得してはいるものの、他からのアクセスをブロックしている状況だと、
	//  削除後に処理が走るので結局問題を起こす
	void unregistAll()
	{
```

```
		//全体ライトロック取得（解放は自動）
		WholeBLockW lock(m_wholeLock, m_wholeLocked.load(), m_wholeLockThreadID.load());
		//全ノード処理
		for (auto node : m_namedRefList)
		{
			RefNodeCore* ref = node.second;//ハッシュテーブルの情報を参照
			if (ref)
			{
				//ref->wLock();//【不要】ライトロック取得
				node.second = nullptr;//ハッシュテーブルから情報を破棄
				//ref->wUnlock();//【不要】ライトロック解放
				m_allocator.destroy(ref);//メモリ解放
			}
		}
		m_namedRefList.clear();//ハッシュテーブルの全ノードをクリア
	}
	//変数の参照をクリア
	void resetRef(const Key key, bool* is_exist = nullptr)
	{
		//変数参照情報取得
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return;
		//変数の参照をクリア
		ref->resetRef();
	}
	//変数の参照を全てクリア
	void resetRefAll()
	{
		//全体リードロック取得（解放は自動）
		WholeBLockR lock(m_wholeLock, m_wholeLocked.load(), m_wholeLockThreadID.load());
		//全変数の参照をクリア
		for (auto node : m_namedRefList)
		{
			RefNodeCore* ref = node.second;
			if (ref)
			{
				//変数の参照をクリア
				ref->resetRef();
			}
		}
	}
	//全体アクセスブロック開始
	bool beginBlock()
	{
		//ブロック中状態判定
		if (m_wholeLocked.load() == true)
			return false;
		//全体ライトロック取得
		CRWLockHelper(m_wholeLock).wLock(SPIN_COUNT);
		m_wholeLocked.store(true);//全体ブロック中状態開始
		m_wholeLockThreadID.store(CThreadID().getID());//現在のスレッドID記録
		return true;
	}
	//全体アクセスブロック終了
	//※内部で全変数参照のクリアを実行
	//※ブロックを行ったスレッドしか解除できない
	bool endBlock()
	{
		//ブロック中状態＆スレッド判定
		if (m_wholeLocked.load() == false || CThreadID() != m_wholeLockThreadID.load())
			return false;
		//全参照をクリア
		resetRefAll();
		//全体ライトロック解法
```

```
            CRWLockHelper(m_wholeLock).wUnlock();
            m_wholeLocked.store(false);//全体ブロック中状態解除
            m_wholeLockThreadID.store(INVALID_THREAD_ID);//現在のスレッド ID 記録クリア
            return true;
      }
private:
      //変数参照情報取得：共通部
      inline const RefNodeCore* findCore(const Key key, bool* is_exist) const
      {
            //全体リードロック取得（解放は自動）
            WholeBLockR lock(m_wholeLock, m_wholeLocked.load(), m_wholeLockThreadID.load());
            //ハッシュテーブル検索
            NamedRefTable::const_iterator node = m_namedRefList.find(key);
            if (node == m_namedRefList.end())
            {
                  //見つからなかった場合
                  if (is_exist)
                        *is_exist = false;
                  return nullptr;
            }
            const RefNodeCore* ref = node->second;
            if (!ref)
            {
                  //見つからなかった場合
                  if (is_exist)
                        *is_exist = false;
                  return nullptr;
            }
            //見つかった場合
            if (is_exist)
                  *is_exist = true;
            return ref;
      }
      //変数参照情報取得：const 用
      inline const RefNodeCore* find(const Key key, bool* is_exist) const
      {
            const RefNodeCore* ref = findCore(key, is_exist);
            return ref;
      }
      //変数参照情報取得：非 const 用
      inline RefNodeCore* find(const Key key, bool* is_exist)
      {
            const RefNodeCore* ref = findCore(key, is_exist);
            return const_cast<RefNodeCore*>(ref);
      }
public:
      //----------
      //値を取得：型指定用
      //※is_exist で参照先の変数の存在を確認可能
      template<typename T>
      T get(const Key key, bool* is_exist = nullptr) const
      {
            const RefNodeCore* ref = find(key, is_exist);
            if (!ref)
                  return static_cast<T>(0);
            ref->rLock();
            T val = ref->getVal<T>();
            ref->rUnlock();
            return val;
      }
      //bool 型用
      bool getBool(const Key key, bool* is_exist = nullptr) const
      {
            return get<bool>(key, is_exist);
      }
```

```
	//int 型用
	int getInt(const Key key, bool* is_exist = nullptr) const
	{
		return get<int>(key, is_exist);
	}
	//float 型用
	float getFloat(const Key key, bool* is_exist = nullptr) const
	{
		return get<float>(key, is_exist);
	}
	//const char*型用
	const char* getStr(const Key key, bool* is_exist = nullptr) const
	{
		return get<const char*>(key, is_exist);
	}
	//----------
	//値を更新：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool set(const Key key, const T val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T now_val = ref->getVal<T>();
		if (prev_val)
			*prev_val = now_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//bool 型用
	bool setBool(const Key key, const bool val, bool* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return set<bool>(key, val, prev_val, is_exist);
	}
	//int 型用
	bool setInt(const Key key, const int val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return set<int>(key, val, prev_val, is_exist);
	}
	//float 型用
	bool setFloat(const Key key, const float val, float* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return set<float>(key, val, prev_val, is_exist);
	}
	//const char*型用
	bool setStr(const Key key, const char* str, const char** prev_val = nullptr, bool* is_exist = nullptr)
	{
		return set<const char*>(key, str, prev_val, is_exist);
	}
	//----------
	//CAS (Compare And Swap)：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool cas(const Key key, const T compare_val, const T swap_val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T now_val = ref->getVal<T>();
```

```
		if (prev_val)
			*prev_val = now_val;
		if (ref->getVal<T>() == compare_val)
			ref->setVal<T>(swap_val);
		ref->wUnlock();
		return true;
	}
	//bool 型用
	bool casBool(const Key key, const bool compare_val, const bool swap_val, bool* prev_val = nullptr,
	                                                                          bool* is_exist = nullptr)
	{
		return cas<bool>(key, compare_val, swap_val, prev_val, is_exist);
	}
	//int 型用
	bool casInt(const Key key, const int compare_val, const int swap_val, int* prev_val = nullptr,
	                                                                       bool* is_exist = nullptr)
	{
		return cas<int>(key, compare_val, swap_val, prev_val, is_exist);
	}
	//float 型用
	bool casFloat(const Key key, const float compare_val, const float swap_val, float* prev_val = nullptr,
	                                                                             bool* is_exist = nullptr)
	{
		return cas<float>(key, compare_val, swap_val, prev_val, is_exist);
	}
	//----------
	//値をインクリメント：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool inc(const Key key, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val += static_cast<T>(1);
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool incInt(const Key key, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return inc<int>(key, prev_val, is_exist);
	}
	//----------
	//値をデクリメント：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool dec(const Key key, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val -= static_cast<T>(1);
		ref->setVal<T>(val);
```

```
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool decInt(const Key key, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return dec<int>(key, prev_val, is_exist);
	}
	//----------
	//値を加算：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool add(const Key key, const T add_val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val += add_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool addInt(const Key key, const int add_val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return add<int>(key, add_val, prev_val, is_exist);
	}
	//float 型用
	bool addFloat(const Key key, const float add_val, float* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return add<float>(key, add_val, prev_val, is_exist);
	}
	//----------
	//値を減算：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool sub(const Key key, const T sub_val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val -= sub_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool subInt(const Key key, const int sub_val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return sub<int>(key, sub_val, prev_val, is_exist);
	}
	//float 型用
	bool subFloat(const Key key, const float sub_val, float* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return sub<float>(key, sub_val, prev_val, is_exist);
```

```
	}
	//----------
	//値を乗算：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool mul(const Key key, const T mul_val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val *= mul_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool mulInt(const Key key, const int mul_val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return mul<int>(key, mul_val, prev_val, is_exist);
	}
	//float 型用
	bool mulFloat(const Key key, const float mul_val, float* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return mul<float>(key, mul_val, prev_val, is_exist);
	}
	//----------
	//値を除算：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool div(const Key key, const T div_val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val /= div_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool divInt(const Key key, const int div_val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return div<int>(key, div_val, prev_val, is_exist);
	}
	//float 型用
	bool divFloat(const Key key, const float div_val, float* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return div<float>(key, div_val, prev_val, is_exist);
	}
	//----------
	//値を剰余算：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool mod(const Key key, const T mod_val, T* prev_val = nullptr, bool* is_exist = nullptr)
```

```
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val %= mod_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool modInt(const Key key, const int mod_val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return mod<int>(key, mod_val, prev_val, is_exist);
	}
	//----------
	//値を反転：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool reverse(const Key key, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		ref->setVal<T>(~val);
		ref->wUnlock();
		return true;
	}
	//bool 型用
	bool reverse(const Key key, bool* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		bool val = ref->getVal<bool>();
		if (prev_val)
			*prev_val = val;
		ref->setVal<bool>(val ? false : true );
		ref->wUnlock();
		return true;
	}
	//bool 型用
	bool reverseBool(const Key key, bool* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return reverse(key, prev_val, is_exist);
	}
	//int 型用
	bool reverseInt(const Key key, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return reverse<int>(key, prev_val, is_exist);
	}
	//----------
	//値を論理積：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
```

```
	bool and(const Key key, const T and_val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val &= and_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool andInt(const Key key, const int and_val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return and<int>(key, and_val, prev_val, is_exist);
	}
	//----------
	//値を論理和：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool or(const Key key, const T or_val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val |= or_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool orInt(const Key key, const int or_val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
		return or<int>(key, or_val, prev_val, is_exist);
	}
	//----------
	//値を排他的論理和：型指定用
	//※prev_val で変更前の値を確認可能
	//※is_exist で参照先の変数の存在を確認可能
	template<typename T>
	bool xor(const Key key, const T xor_val, T* prev_val = nullptr, bool* is_exist = nullptr)
	{
		RefNodeCore* ref = find(key, is_exist);
		if (!ref)
			return false;
		ref->wLock();
		T val = ref->getVal<T>();
		if (prev_val)
			*prev_val = val;
		val ^= xor_val;
		ref->setVal<T>(val);
		ref->wUnlock();
		return true;
	}
	//int 型用
	bool xorInt(const Key key, const int xor_val, int* prev_val = nullptr, bool* is_exist = nullptr)
	{
```

```
            return xor<int>(key, xor_val, prev_val, is_exist);
      }
      //----------
      //参照先が存在するか？
      bool isExist(const Key key) const
      {
            bool is_exist;
            find(key, &is_exist);
            return is_exist;
      }
public:
      //コンストラクタ
      CNamedRefT()
      {
            //ハッシュテーブルの要素数を最大数で予約
            m_namedRefList.reserve(MAX_NODES);
            //全体ブロック状態解除
            m_wholeLocked.store(false);
            m_wholeLockThreadID.store(INVALID_THREAD_ID);
      }
      //デストラクタ
      ~CNamedRefT()
      {
            //ハッシュテーブルから全ノード削除
            unregistAll();
      }
private:
      //フィールド
      CPoolAllocatorWithBuff<REF_NODE_SIZE_MAX, MAX_NODES> m_allocator;//プールアロケータ（最大数以上の確保は不可）
      NamedRefTable m_namedRefList;//【暫定】C++11 標準ライブラリのハッシュテーブル
      mutable CRWLock m_wholeLock;//全体ブロック用リード・ライトロック
      mutable std::atomic<bool> m_wholeLocked;//全体ブロック中
      mutable std::atomic<THREAD_ID> m_wholeLockThreadID;//全体ブロックスレッド ID
};
```

【名前付きデータ参照シングルトン】

```
//---------------------------------------
//名前付きデータ参照クラスシングルトン
//※テンプレートパラメータで記録可能な参照の最大数を指定
//※インスタンス識別 ID を指定することで、複数のシングルトンを作成可能
template <std::size_t MAX, int ID = 0, int SPIN = CSpinLock::DEFAULT_SPIN_COUNT>
class CNamedRefSingletonT
{
public:
      //型
      typedef CNamedRefT<MAX, SPIN> NamedRef;//名前付きデータ参照クラス
public:
      //定数
      static const int INSTANCE_ID = ID;//インスタンス識別 ID
public:
      //オペレータ
      operator NamedRef&() { return m_namedRef; }//キャストオペレータ
      const NamedRef* operator->() const { return &m_prosMap; }//アロー演算子（名前付きデータ参照クラスのプロキシー）
      NamedRef* operator->(){ return &m_namedRef; }//アロー演算子（名前付きデータ参照クラスのプロキシー）
private:
      //フィールド
      static NamedRef m_namedRef;//【static】名前付きデータ参照クラス本体
};
```

【データ型のエイリアス】

```
//---------------------------------------
//データ型のエイリアス
static const std::size_t NAMED_REF_MAX = 256;//名前付きデータ参照テーブルの最大数
using CNamedRefSingleton = CNamedRefSingletonT<NAMED_REF_MAX>;//名前付きデータ参照クラスシングルトン
using CNamedRef = CNamedRefSingleton::NamedRef;//名前付きデータ参照クラス
```

```
using RefK = CNamedRef::Key;//名前付きデータ参照キー
template <typename T>
using CRefR = CNamedRef::RefR<T>;//名前付きデータ参照：構造体参照クラス（読み込み用）
template <typename T>
using CRefW = CNamedRef::RefW<T>;//名前付きデータ参照：構造体参照クラス（書き込み用）
template <typename T>
using RefF = CNamedRef::BaseRefF<T>;//データ参照用関数オブジェクトの基本型
```

【データ参照用関数オブジェクト作成マクロ：直接参照タイプ】

```
//--------------------
//データ参照用関数オブジェクト作成マクロ：直接参照タイプ
#define NAMED_REF_FUNCTOR(name, code, type) ¥
    struct name : RefF<type> ¥
    { ¥
    TYPE* operator()(const RefK& key) const ¥
        { ¥
            code; ¥
        } ¥
    };
//データ参照用関数オブジェクト作成マクロ：直接参照タイプ：簡易版
#define NAMED_REF_FUNCTOR_DIRECT(name, var, type) ¥
    NAMED_REF_FUNCTOR(name, return &var, type)
```

【データ参照用関数オブジェクト作成マクロ：アクセッサタイプ】

```
//--------------------
//データ参照用関数オブジェクト作成マクロ：アクセッサタイプ
#define NAMED_ACCESSOR_FUNCTOR(name, getter_code, setter_code, type) ¥
    struct name : RefF<type> ¥
    { ¥
        struct getter ¥
        { ¥
            TYPE operator()(const RefK& key) const ¥
            { ¥
                getter_code; ¥
            } ¥
        }; ¥
        struct setter ¥
        { ¥
            void operator()(const RefK& key, const TYPE val) const ¥
            { ¥
                setter_code; ¥
            } ¥
        }; ¥
    };
//データ参照用関数オブジェクト作成マクロ：アクセッサタイプ：簡易版
#define NAMED_ACCESSOR_FUNCTOR_SIMPLE(name, obj, getter_method, setter_method, type) ¥
    NAMED_ACCESSOR_FUNCTOR(name, return obj getter_method(), obj setter_method(val), type })
```

【シングルトンインスタンス化】

```
//---------------------------------------
//名前付きデータ参照クラスシングルトンインスタンス化
CNamedRef CNamedRefSingleton::m_namedRef;
```

▼ 名前付きデータ参照クラスの使用サンプル

名前付きデータ参照クラスの使用サンプルを示す。

名前付きデータ参照クラスを使用している箇所を赤字で示す。

【データ用データ定義】

```
//---------------------------------------
//テスト用データ定義
```

```
namespace TEST
{
	static bool s_debugCpuMeter = false;//【デバッグ用】CPU 使用率表示状態
	static int s_debugData1 = 1;//【デバッグ用】データ 1
	static float s_debugData2 = 2.f;//【デバッグ用】データ 2
	static const char* s_debugData3 = "DATA3";//【デバッグ用】データ 3
	static short s_debugData4 = 4;//【デバッグ用】データ 4
	static int s_debugData5 = 5;//【デバッグ用】データ 5
	//キャラ情報
	static CCharaInfo s_charaInfoList[] =
	{
		{ 10, "太郎", 11, 111.1f, 1111 },
		{ 20, "次郎", 22, 222.2f, 2222 },
		{ 30, "三郎", 33, 333.3f, 3333 }
	};
	//キャラ情報検索関数
	static CCharaInfo* searchCharaInfo(const int id)
	{
		for (auto& o : s_charaInfoList)
		{
			if (o.id == id)
				return &o;
		}
		return nullptr;
	}
}

//---------------------------------------
//補助関数
const char* boolToStr(const bool val){ return val ? "true" : "false"; }//bool 値を文字列化
```

【テスト関数①：名前付きデータ参照を登録】

```
//---------------------------------------
//テスト関数①：名前付きデータ参照を登録
void testRegistNamedRef()
{
	//変数参照用関数オブジェクトを定義：直接参照タイプ
	//・条件①：構造体内に「typedef データ型 TYPE;」を定義するか、「RefF<データ型>」を継承する。
	//・条件②：「TYPE* operator()(const RefK& key) const { return 変数のポインタ; }」を定義する。
	//・条件③：メンバー変数があってはいけない
	//struct refDebugCpuMeter : RefF<bool>{ TYPE* operator()(const RefK& key) const {
	//	return &TEST::s_debugCpuMeter; } };//【デバッグ用】CPU 使用率表示状態参照用
	//struct refDebugParam1 : RefF<int>{ TYPE* operator()(const RefK& key) const {
	//	return &TEST::s_debugData1; } };//【デバッグ用】データ 1 参照用
	//struct refDebugParam2 : RefF<float>{ TYPE* operator()(const RefK& key) const {
	//	return &TEST::s_debugData2; } };//【デバッグ用】データ 2 参照用
	//struct refDebugParam3 : RefF<const char*>{ TYPE* operator()(const RefK& key) const {
	//	return &TEST::s_debugData3; } };//【デバッグ用】データ 3 参照用
	//struct refDebugParam4 : RefF<short>{ TYPE* operator()(const RefK& key) const {
	//	return &TEST::s_debugData4; } };//【デバッグ用】データ 4 参照用
	//struct refCharaInfo : RefF<CCharaInfo>{ TYPE *operator()(const RefK& key) const {
	//	return TEST::searchCharaInfo(key.m_sub); } };//キャラ情報参照用
	//※上記の構造体と同じことを「NAMED_REF_FUNCTOR」マクロで代用できる
	NAMED_REF_FUNCTOR_DIRECT(refDebugCpuMeter, TEST::s_debugCpuMeter, bool);//【デバッグ用】CPU 使用率表示状態参照用
	NAMED_REF_FUNCTOR_DIRECT(refDebugParam1, TEST::s_debugData1, int);//【デバッグ用】データ 1 参照用
	NAMED_REF_FUNCTOR_DIRECT(refDebugParam2, TEST::s_debugData2, float);//【デバッグ用】データ 2 参照用
	NAMED_REF_FUNCTOR_DIRECT(refDebugParam3, TEST::s_debugData3, const char*);//【デバッグ用】データ 3 参照用
	NAMED_REF_FUNCTOR_DIRECT(refDebugParam4, TEST::s_debugData4, short);//【デバッグ用】データ 4 参照用
	NAMED_REF_FUNCTOR(refCharaInfo, return TEST::searchCharaInfo(key.m_sub), CCharaInfo);//キャラ情報参照用

	//変数参照用関数オブジェクトを定義：アクセッサタイプ
	//・条件①：構造体内に「typedef データ型 TYPE;」を定義するか、「RefF<データ型>」を継承する。
	//・条件②：「struct getter」を定義し、そのメンバーとして
	//	「TYPE operator()(const RefK& key) const { return 変数の値; }」を定義する。
	//・条件③：「struct setter」を定義し、そのメンバーとして
```

```
	//	「void operator()(const RefK& key, const TYPE val){ 変数 = val; }」を定義する。
	//・条件④：メンバー変数があってはいけない
	//struct refDebugParam5 : RefF<int>{//【デバッグ用】データ5アクセス用
	//	struct getter{ TYPE operator()(const RefK& key) const { return TEST::s_debugData5; } };
	//	struct setter{ void operator()(const RefK& key, const TYPE val) { TEST::s_debugData5 = val; } };
	//};
	//※上記の構造体と同じことを「NAMED_ACCESSOR_FUNCTOR」マクロで代用できる
	NAMED_ACCESSOR_FUNCTOR(refDebugParam5, return TEST::s_debugData5, TEST::s_debugData5 = val, int);
	                                                                    //【デバッグ用】データ5アクセス用
	//※共通関数を作成したりなど、複雑な処理が必要ならマクロは使えない
	struct refCharaInfoData3 : RefF<int>{//キャラ情報のData3アクセス用
		typedef CCharaInfo OBJ;
		static OBJ* search(const RefK& key){ return TEST::searchCharaInfo(key.m_sub); }
		struct getter{ TYPE operator()(const RefK& key) const {
			const OBJ* obj = search(key);  return obj ? obj->getData3() : 0; } };
		struct setter{ void operator()(const RefK& key, const TYPE val) {
			OBJ* obj = search(key); obj->setData3(val); } };
	};

	//名前付きデータ参照クラスシングルトン
	CNamedRefSingleton named_ref;

	//変数参照を登録：直接参照タイプ
	named_ref->regist(RefK("DebugCpuMeter"), refDebugCpuMeter());//【デバッグ用】CPU使用率表示状態
	named_ref->regist(RefK("DebugParam1"), refDebugParam1());//【デバッグ用】データ1
	named_ref->regist(RefK("DebugParam2"), refDebugParam2());//【デバッグ用】データ2
	named_ref->regist(RefK("DebugParam3"), refDebugParam3());//【デバッグ用】データ3
	named_ref->regist(RefK("DebugParam4"), refDebugParam4());//【デバッグ用】データ4
	named_ref->regist(RefK("CharaInfo", 10), refCharaInfo());//キャラ情報：ID=10
	named_ref->regist(RefK("CharaInfo", 20), refCharaInfo());//キャラ情報：ID=20
	named_ref->regist(RefK("CharaInfo", 30), refCharaInfo());//キャラ情報：ID=30
	//変数参照を登録：アクセッサタイプ
	named_ref->registAcc(RefK("DebugParam5"), refDebugParam5());//【デバッグ用】データ5
	named_ref->registAcc(RefK("CharaInfo::Data3", 10), refCharaInfoData3());//キャラ情報：Data3：ID=10
	named_ref->registAcc(RefK("CharaInfo::Data3", 20), refCharaInfoData3());//キャラ情報：Data3：ID=20
	named_ref->registAcc(RefK("CharaInfo::Data3", 30), refCharaInfoData3());//キャラ情報：Data3：ID=30
}
```

【テスト関数②：名前付きデータ参照を使ってデータを表示】

```
//-----------------------------------
//テスト関数②：名前付きデータ参照を使ってデータを表示
void testPrintNamedRef()
{
	CNamedRefSingleton named_ref;
	printf("DebugCpuMeter=%s¥n", boolToStr(named_ref->getBool(RefK("DebugCpuMeter"))));
	printf("DebugParam1=%d¥n", named_ref->getInt(RefK("DebugParam1")));
	printf("DebugParam2=%.3f¥n", named_ref->getFloat(RefK("DebugParam2")));
	printf("DebugParam3=¥"%s¥"¥n", named_ref->getStr(RefK("DebugParam3")));
	printf("DebugParam4=%d¥n", named_ref->get<short>(RefK("DebugParam4")));
	printf("DebugParam5=%d¥n", named_ref->getInt(RefK("DebugParam5")));
	for (int id : {10, 20, 30})
	{
		CRefR<CCharaInfo> obj(named_ref, RefK("CharaInfo", id));//処理ブロックを抜けるまでロックを取得（自動開放）
		if (obj.isExist())
			printf("CharaInfo[id=%d]: {name=¥"%s¥", data1=%d, data2=%.3f, data3=%d}¥n",
				obj->id, obj->name, obj->data1, obj->data2, named_ref->getInt(RefK("CharaInfo::Data3", id)));
	}
}
```

【テスト関数③：名前付きデータ参照を使ってデータを更新】

```
//-----------------------------------
//テスト関数③：名前付きデータ参照を使ってデータを更新
void testUpdateNamedRef()
{
	CNamedRefSingleton named_ref;
```

```
	named_ref->set(RefK("DebugCpuMeter"), true);
	named_ref->set(RefK("DebugParam1"), 123);
	named_ref->set(RefK("DebugParam2"), 456.789f);
	named_ref->set(RefK("DebugParam3"), "New String");
	named_ref->set<short>(RefK("DebugParam4"), 12345);
	named_ref->set(RefK("DebugParam5"), 67890);
	{
		int id = 20;
		CRefW<CCharaInfo> obj(named_ref, RefK("CharaInfo", id));//処理ブロックを抜けるまでロックを取得（自動開放）
		if (obj.isExist())
		{
			obj->data1 = 321;
			obj->data2 = 987.654f;
		}
		named_ref->setInt(RefK("CharaInfo::Data3", id), 987654);
	}
}
```

【テスト関数④：名前付きデータ参照を使ってデータを演算１】

```
//---------------------------------------
//テスト関数④：名前付きデータ参照を使ってデータを演算１
void testCalcNamedRef1()
{
	CNamedRefSingleton named_ref;
	named_ref->reverseBool(RefK("DebugCpuMeter"));
	named_ref->incInt(RefK("DebugParam1"));
	named_ref->addFloat(RefK("DebugParam2"), 100.f);
	named_ref->mul<short>(RefK("DebugParam4"), 2);
	named_ref->reverseInt(RefK("DebugParam5"));
	{
		int id = 20;
		named_ref->xorInt(RefK("CharaInfo::Data3", id), 0xffff);
	}
}
```

【テスト関数⑤：名前付きデータ参照を使ってデータを演算２（演算１の前の状態に戻す逆計算）】

```
//---------------------------------------
//テスト関数⑤：名前付きデータ参照を使ってデータを演算２（演算１の前の状態に戻す逆計算）
void testCalcNamedRef2()
{
	CNamedRefSingleton named_ref;
	named_ref->reverseBool(RefK("DebugCpuMeter"));
	named_ref->decInt(RefK("DebugParam1"));
	named_ref->subFloat(RefK("DebugParam2"), 100.f);
	named_ref->div<short>(RefK("DebugParam4"), 2);
	named_ref->reverseInt(RefK("DebugParam5"));
	{
		int id = 20;
		named_ref->xorInt(RefK("CharaInfo::Data3", id), 0xffff);
	}
}
```

【テスト関数⑥：名前付きデータ参照を使ってデータを演算３（論理演算）】

```
//---------------------------------------
//テスト関数⑥：名前付きデータ参照を使ってデータを演算３（論理演算）
void testCalcNamedRef3()
{
	CNamedRefSingleton named_ref;
	named_ref->setInt(RefK("DebugParam1"), 0x33);
	named_ref->setInt(RefK("DebugParam5"), 0xf0);
	named_ref->andInt(RefK("DebugParam1"), 0x7);
	named_ref->orInt(RefK("DebugParam5"), 0xf);
}
```

【テスト関数⑦：名前付きデータ参照が内部で保持する参照をリセット】

```
//---------------------------------------
//テスト関数⑦：名前付きデータ参照が内部で保持する参照をリセット
//※見た目に何の影響はないが、次回なんらかの処理時に参照を再取得する。
//※データが消えたり登録が削除されたりするわけではない。
//※2 回目以降のアクセスを高速化するために内部で保持している参照（ポインタ）をクリアするだけ。
//※メモリ再配置や、データの削除が行われた場合に実行する必要がある。
//※サンプルでは全体リセットだが、個別のリセットも可能。
//※アクセッサタイプには何の影響もない。
void testResetRefAll()
{
	CNamedRefSingleton named_ref;
	named_ref->resetRefAll();
}
```

【テスト関数⑧：全体ブロック開始】

```
//---------------------------------------
//テスト関数⑧：全体ブロック開始
void testBeginBlock()
{
	CNamedRefSingleton named_ref;
	named_ref->beginBlock();
}
```

【テスト関数⑨：全体ブロック終了】

```
//---------------------------------------
//テスト関数⑨：全体ブロック終了
void testEndBlock()
{
	CNamedRefSingleton named_ref;
	named_ref->endBlock();
}
```

【テスト関数⑩：一部の名前付きデータ参照を登録解除】

```
//---------------------------------------
//テスト関数⑩：一部の名前付きデータ参照を登録解除
void testUnregistNamedRef()
{
	CNamedRefSingleton named_ref;
	named_ref->unregist(RefK("DebugParam1"));
	named_ref->unregist(RefK("DebugParam3"));
	named_ref->unregist(RefK("DebugParam5"));
	named_ref->unregist(RefK("CharaInfo", 20));
	named_ref->unregist(RefK("CharaInfo::Data3", 20));
}
```

【テスト関数⑪：全ての名前付きデータ参照を登録解除】

```
//---------------------------------------
//テスト関数⑪：全ての名前付きデータ参照を登録解除
void testUnregistRefAll()
{
	CNamedRefSingleton named_ref;
	named_ref->unregistAll();
}
```

【マルチスレッドテスト関数①：大量同時更新テスト】

```
//---------------------------------------
//マルチスレッドテスト①：大量同時更新テスト
void readyTestThread1()//準備
{
	CNamedRefSingleton named_ref;
	named_ref->setInt(RefK("DebugParam1"), 0);//テストのために 0 クリア
}
void testThread1(const int n)//処理
{
```

```
	CNamedRefSingleton named_ref;
	for (int i = 0; i < 10000; ++ i)
		named_ref->addInt(RefK("DebugParam1"), n);//10000 回 n ずつ加算
}
```

【マルチスレッドテスト関数②:ブロック中のアクセステスト】

```
//----------------------------------------
//マルチスレッドテスト②:ブロック中のアクセステスト
void testThread2()
{
	CNamedRefSingleton named_ref;
	printf("----- THREAD:BEGIN -----¥n");
	int param1 = named_ref->getInt(RefK("DebugParam1"));//データ取得
	printf("----- THREAD:SLEEP(param1=%d) -----¥n", param1);
	std::this_thread::sleep_for(std::chrono::milliseconds(100));
                                                    //スリープ(この間にメイン側でデータ参照の登録が解除される)
	param1 = named_ref->getInt(RefK("DebugParam1"));//データ取得
	printf("----- THREAD:END(param1=%d) -----¥n", param1);
}
```

【テスト用メイン】

```
//----------------------------------------
//テスト
int main(const int argc, const char* argv[])
{
	printf("sizeof(CNamedRef)=%d¥n", sizeof(CNamedRef));
	printf("sizeof(CNamedRefSingleton)=%d¥n", sizeof(CNamedRefSingleton));

	//テスト①:名前付きデータ参照を登録
	printf("----- regist -----¥n");
	testRegistNamedRef();
	//テスト②:名前付きデータ参照を使ってデータを表示
	testPrintNamedRef();

	//テスト①:もう一度登録しても登録済みのキーは登録に失敗するだけ
	testRegistNamedRef();

	//テスト③:名前付きデータ参照を使ってデータを更新
	printf("----- set value -----¥n");
	testUpdateNamedRef();
	testPrintNamedRef();

	//テスト④:名前付きデータ参照を使ってデータを演算1
	printf("----- calc value (1) -----¥n");
	testCalcNamedRef1();
	testPrintNamedRef();

	//テスト⑤:名前付きデータ参照を使ってデータを演算2
	printf("----- calc value (2) -----¥n");
	testCalcNamedRef2();
	testPrintNamedRef();

	//テスト⑥:名前付きデータ参照を使ってデータを演算3
	printf("----- calc value (3) -----¥n");
	testCalcNamedRef3();
	testPrintNamedRef();

	//テスト関数⑦:名前付きデータ参照が内部で保持する参照をリセット
	printf("----- reset-ref all -----¥n");
	testResetRefAll();
	testPrintNamedRef();

	//スレッドテスト①
	printf("----- thread test -----¥n");
	std::thread th1ready = std::thread(readyTestThread1);//準備 (param1 を 0 クリア)
```

```
	th1ready.join();
	std::thread th1[10];
	int n = 1;
	for (auto& th1tmp : th1)
	{
		th1tmp = std::thread(testThread1, n++);
	}
	for (auto& th1tmp : th1)
	{
		th1tmp.join();
	}
	testPrintNamedRef();

	//テスト関数⑧：全体ブロック開始
	printf("----- begin block -----¥n");
	testBeginBlock();
	testPrintNamedRef();

	//スレッドテスト②
	std::thread th2 = std::thread(testThread);
	std::this_thread::sleep_for(std::chrono::milliseconds(500));//スリープ

	//テスト③（再）：名前付きデータ参照を使ってデータを更新
	printf("----- set value : 2nd time -----¥n");
	testUpdateNamedRef();
	testPrintNamedRef();

	//テスト関数⑨：全体ブロック終了
	printf("----- end block -----¥n");
	testEndBlock();
	testPrintNamedRef();

	//テスト関数⑩：一部の名前付きデータ参照を登録解除
	printf("----- unregist -----¥n");
	testUnregistNamedRef();
	testPrintNamedRef();

	//テスト関数⑪：全ての名前付きデータ参照を登録解除
	printf("----- unregist all -----¥n");
	testUnregistRefAll();
	testPrintNamedRef();

	//スレッド終了待ち
	th2.join();

	return EXIT_SUCCESS;
}
```

↓（実行結果）

```
sizeof(CNamedRef)=14448
sizeof(CNamedRefSingleton)=1
----- regist -----
DebugCpuMeter=false                                          ←名前付きデータ参照を通してデータ表示
DebugParam1=1                                                ←※初期値が表示されていることを確認
DebugParam2=2.000                                            ←（同上）
DebugParam3="DATA3"                                          ←（同上）
DebugParam4=4                                                ←（同上）
DebugParam5=5                                                ←（同上）
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}   ←（同上）
CharaInfo[id=20]: {name="次郎", data1=22, data2=222.200, data3=2222}   ←（同上）
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}   ←（同上）
----- set value -----
DebugCpuMeter=true                                           ←名前付きデータ参照を通してデータ更新
DebugParam1=123                                              ←※値が更新されていることを確認
DebugParam2=456.789                                          ←（同上）
```

```
DebugParam3="New String"                                          ←（同上）
DebugParam4=12345                                                 ←（同上）
DebugParam5=67890                                                 ←（同上）
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=987654} ←（同上）
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}
----- calc value (1) -----
DebugCpuMeter=false                                               ←名前付きデータ参照を通して演算
DebugParam1=124                                                   ←※値が更新されていることを確認
DebugParam2=556.789                                               ←（同上）
DebugParam3="New String"
DebugParam4=24690                                                 ←（同上）
DebugParam5=-67891                                                ←（同上）
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=1043961}←（同上）
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}
----- calc value (2) -----
DebugCpuMeter=true                                                ←名前付きデータ参照を通して演算
DebugParam1=123                                                   ←※(1)の逆演算で値が元に戻っていることを確認
DebugParam2=456.789                                               ←（同上）
DebugParam3="New String"
DebugParam4=12345                                                 ←（同上）
DebugParam5=67890                                                 ←（同上）
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=987654} ←（同上）
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}
----- calc value (3) -----
DebugCpuMeter=true
DebugParam1=3                                                     ←名前付きデータ参照を通して演算
DebugParam2=456.789                                                 ※論理演算の結果を確認
DebugParam3="New String"
DebugParam4=12345
DebugParam5=255                                                   ←（同上）
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=987654}
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}
----- reset-ref all -----
DebugCpuMeter=true                                                ←記憶している参照（ポインタ）をリセットしても
DebugParam1=3                                                     ←変わらず値が参照できることを確認
DebugParam2=456.789                                               ←（同上）
DebugParam3="New String"                                          ←（同上）
DebugParam4=12345                                                 ←（同上）
DebugParam5=255                                                   ←（同上）
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}  ←（同上）※アクセッサタイプはポインタを
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=987654} ←（同上）　記憶しないので元々影響なし
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}  ←（同上）　（data3 がアクセッサタイプで参照）
----- thread test -----
DebugCpuMeter=true
DebugParam1=550000                                                ←10 個のスレッドに 1～10 の値を振り分け、
DebugParam2=456.789                                                 それぞれのスレッドで 10000 回ずつ加算
DebugParam3="New String"                                            ※アトミック性が保証され、正しく計算
DebugParam4=12345                                                     されていることを確認
DebugParam5=255
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=987654}
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}
----- begin block -----                                           ←全体ブロック開始
DebugCpuMeter=true                                                  ※メモリ再配置やリソース削除の前後を
DebugParam1=550000                                                    ブロックすることで、データの不正アクセスを
DebugParam2=456.789                                                   防ぐ
DebugParam3="New String"                                            ※ブロック中も、ブロックを行ったスレッド
DebugParam4=12345                                                     だけはアクセスできるので値が表示される
DebugParam5=255                                                     ※内容は何も変わっていない
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=987654}
```

```
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}
----- THREAD:BEGIN -----                                    ←ブロック中にスレッド開始
----- set value : 2nd time -----
DebugCpuMeter=true                                          ←ブロック中もブロックを実行したスレッドは
DebugParam1=123                                             ←値の更新が可能
DebugParam2=456.789                                         ←※「set value」を再実行したことにより、
DebugParam3="New String"
DebugParam4=12345                                           ←　元の値に戻っていることを確認
DebugParam5=67890                                           ←（同上）
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=987654} ←（同上）
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}
----- end block -----                                       ←全体ブロック終了
DebugCpuMeter=true                                            ※内容は何も変わっていない
----- THREAD:SLEEP(param1=123) -----                        ←ブロック終了に伴い、スレッド側の処理が
DebugParam1=123                                               反応して処理が進んだことを確認
DebugParam2=456.789
DebugParam3="New String"
DebugParam4=12345
DebugParam5=67890
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}
CharaInfo[id=20]: {name="次郎", data1=321, data2=987.654, data3=987654}
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}
----- unregist -----
DebugCpuMeter=true
DebugParam1=0                                               ←データ参照の登録を一部解除
DebugParam2=456.789                                           ※解除されたデータが参照できなくなっている
DebugParam3="(null)"                                        ←　ことを確認
DebugParam4=12345                                             ※結果が 0 であること以外に、アクセス関数
DebugParam5=0                                                   呼び出し時に bool* is_exist を渡すと
CharaInfo[id=10]: {name="太郎", data1=11, data2=111.100, data3=1111}    参照の有無を確認できる
CharaInfo[id=30]: {name="三郎", data1=33, data2=333.300, data3=3333}  ←（id=20 が削除されて表示されない）
----- unregist all -----
DebugCpuMeter=false                                         ←データ参照の登録を（残り）全部解除
DebugParam1=0                                                 ※解除されたデータが参照できなくなっている
DebugParam2=0.000                                           ←（同上）
DebugParam3="(null)"
DebugParam4=0                                               ←（同上）
DebugParam5=0
----- THREAD:END(param1=0) -----                            ←データ参照登録解除後のスレッドからのアクセス
                                                              ※データを参照できない
```

■■以上■■

■ 索引

索引項目が見つかりません。

スクリプトの生産性向上のための名前付きデータ参照

以　上