セーブデータのためのシリアライズ処理

– 互換性維持とデバッグ効率向上のために –

2014 年 2 月 27 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 27 日	板垣 衛	（初稿）

■ 目次

■ 概略

ゲームのセーブデータのためのシリアライズ処理を設計する。
バージョン互換性に強く、かつ、生産性の高いセーブデータ書き込み／読み込みシステムを構築する。

ゲームタイトル固有のセーブデータ構造に特化したプログラミングは必要とするが、その手間を極力抑えるための仕組みを策定する。そのプログラミングスタイルは、Boost C++ ライブラリの boost::serialization を参考にする。

■ 目的

本書は、柔軟で生産性の高いシリアライズ処理を作ることにより、ゲーム制作の効率を向上させることを目的とする。

ゲーム制作の過程でセーブデータの構造が変わるのは当然としても、その互換性が保証されることにより、セーブデータ依存の制作作業や QA 作業に支障をきたさずに済む。本書のシステムはそのための汎用処理であり、また、プログラマーの手間もできる限り簡略化する。
さらには、このシステムを通して、作業用のセーブデータの量産を簡単に行えるようにすることも目的とする。

■ 用語の規定とシリアライズの基礎

本書の構成にあたって、まずは用語を規定する。
これにより、本書の前提となるシリアライズの基礎も説明する。
一般的なシリアライズ関係の用語の説明から、本書独自の用語の規定までを含む。

▼ シリアライズ（Serialize）

「シリアライズ」とは、メモリ上のデータを、「後で復元すること」を目的に、保存可能な形式に変換することである。
シリアライズの主な用途は、「ファイルへの保存」、「データ通信」である。
なお、日本語では「直列化」と訳される。また、（主にファイルに保存する意味で）「永

続化」と呼ばれることも多い。

本書においては、「ゲームのセーブデータを作成すること」をシリアライズと呼ぶ。
ファイルに直接書き込むことまでを意味せず、ファイルに書き込み可能なイメージをメモリ上に作成することを指すものとする。

▼ デシリアライズ（Deserialize）

「デシリアライズ」とは、シリアライズされたデータをメモリ上に復元することである。
「アンシリアライズ」（Unserialize）と呼ばれることもある。

本書においては、「ゲームのセーブデータからゲームデータを復元すること」をデシリアライズと呼ぶ。
ファイルから直接読み込むことまでを意味せず、メモリ上のファイルイメージを元に復元することを指すものとする。

なお、デシリアライズの際は、純粋にセーブ時の状態を再現するのではなく、可能な限り、現在の構造に適合する形で復元する、バージョン互換維持処理を伴う。

▼ セーブデータ

ゲームの進行状態が記録されたファイルのこと。
シリアライズによって作成されたデータのことであり、文脈によってはメモリ上のファイルイメージを意味するが、基本的には保存されたファイルのことである。

▼ セーブ（Save）

セーブデータを作成すること。
シリアライズからファイルへの保存をひとまとめにした処理を意味する。

▼ ロード（Load）

セーブデータを読み込んで、ゲームの進行状態を復元すること。
ファイルの読み込みからデシリアライズをひとまとめにした処理を意味する。

▼ アーカイブ（Archive）

Boost C++ ライブラリの boost::serialization クラスに基づいてこの用語を用いる。

本来の用語としては、データを「貯蔵」することであり、複数のファイルを一つにまとめるようなことを意味するが、本書においてはシリアライズしたデータの「保存形式」を意味する。

「保存形式」には、大きく分けて「バイナリ」と「テキスト」があり、「バイナリアーカイブ」や「テキストアーカイブ」などと呼ぶ。なお、boost::serialization では標準的に「XML アーカイブ」も扱う。

「アーカイブ」という言葉は基本的には名詞であり、文脈によっては「ファイル」と同義としても用いられるが、「アーカイブする」「アーカイブしたデータ」のように動詞としても用いられる。

▼ シリアライズとアーカイブの違い

処理として「シリアライズ」と「アーカイブ」は厳密には異なる。

シリアライズは保存する情報（とその順序）の指定であり、実際に保存を行うのはアーカイブである。

boost::serializetion では、シリアライズがユーザー定義処理で、アーカイブが汎用処理である。

▼ コレクター（Collector）

セーブデータを構成するために、分散するデータを収集する処理を担う仕組みを意味する。また、「収集」（動詞）の意味では「コレクト」（Collect）という用語も用いる。

これは、本書が独自に規定する用語である。セーブデータ処理要件に柔軟に対応するための仕組みとして導入する。

なお、これはあくまでもシリアライズの対象データを集めることを意味するものであり、boost::serialization の「非侵入型（non-intrusive）シリアライズ」のような意味ではない。（解説：boost::serialization は、シリアライズ対象クラス本体にシリアライズ処理を記述するのが基本であるが、外部に記述することもできる。標準ライブラリが提供するクラスなど、直接シリアライズ処理を記述できないものをシリアライズするために用意されている仕組み。）

▼ ディストリビュータ（Distributor）

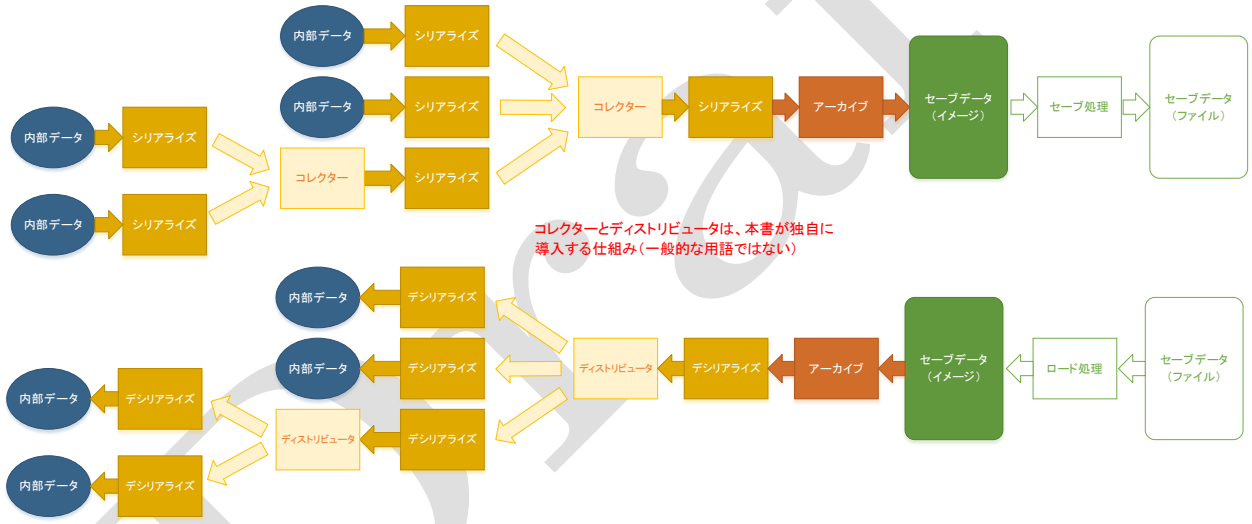
セーブデータの復元のために、読み込んだデータを各所に分散配置する処理を担う仕組みを意味する。また、「分配」（動詞）の意味では「ディストリビュート」（Distribute）という用語も用いる。

これは、本書が独自に規定する用語である。セーブデータ処理要件に柔軟に対応するための仕組みとして導入する。

▼ 各用語の関係と処理イメージ

上記の各用語の関係を、処理イメージにあてはめると下記のとおりとなる。

用語の関係と基本処理イメージ：



「コレクター」と「ディストリビュータ」により、boost::serialization などの一般的なシリアライズ処理と異なり、直接関連性のない複数のデータをひとまとめに扱うようにする。これにより、ゲームプログラムの自由度とセーブデータの自由度の両方を獲得する。

■ 【参考】Boost C++ ライブラリの boost::serialization

本システムの要件を定義する前に、本システムの参考とする Boost C++ライブラリの boost::serialization を簡単に説明する。

なお、直接 boost::serialization を使用しないのは、「コレクター」と「ディストリビュータ」のような仕組みを導入して、柔軟な処理要件に対応するためである。

▼ Boost C++ライブラリの準備

Boost C++ライブラリは、ヘッダーをインクルードするだけでそのまま使用できるテンプレートクラスを多く含むが、一部のライブラリはプラットフォーム向けのビルドが必要である。boost::serialization もそのようなライブラリの一つである。

なお、ビルド方法の説明については省略する。

▼ シリアライズのサンプル①：基本的な動作

実際に boost::serialization を使用したサンプルを示す。

単純なクラスをシリアライズしてテキストアーカイブに保存し、またデシリアライズする。「serialize」テンプレート関数が、シリアライズとデシリアライズの処理で共用される。

基本的なシリアライズとデシリアライズのサンプル：

【インクルード】

```
#include <fstream>//ファイル入出力ストリーム
#include <boost/serialization/serialization.hpp>//シリアライズ
#include <boost/archive/text_oarchive.hpp>//テキスト出力アーカイブ
#include <boost/archive/text_iarchive.hpp>//テキスト入力アーカイブ
```

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス１
class CTest1
{
public:
	//アクセッサ
	int getData1() const { return m_data1; }//データ１取得
	void setData1(const int value){ m_data1 = value; }//データ１更新
	float getData2() const { return m_data2; }//データ２取得
	void setData2(const float value){ m_data2 = value; }//データ２更新
	char getData3(const int index) const { return m_data3[index]; }//データ３取得
	void setData3(const int index, const int value){ m_data3[index] = value; }//データ３更新
private:
	//シリアライズ用テンプレート関数
	//※シリアライズとデシリアライズの共通処理
	//※private スコープにした上でフレンドクラスを設定する
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("serialize:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ１ ※アーカイブに & 演算子でデータを渡す
		arc & m_data2 & m_data3;//データ２ & データ３ ※連続してデータを渡すことも可能
	}
private:
	//フィールド
	int m_data1;//データ１：int 型
	float m_data2;//データ２：float 型
	char m_data3[3];//データ３：配列型
};
//--------------------
//シリアライズ設定
BOOST_CLASS_VERSION(CTest1, 99);//バージョン設定
BOOST_CLASS_TRACKING(CTest1, boost::serialization::track_never);//VC++でエラー C4308 を回避するための設定
```

【シリアライズ】

```
//--------------------
//シリアライズテスト１：テキストアーカイブ
void serializeTest1()
{
	printf("--------------------¥n");
	printf("シリアライズ：テキストアーカイブ¥n");
	//オブジェクト生成
	CTest1 obj;
	//テストデータをセット
	obj.setData1(123);
	obj.setData2(4.56f);
	obj.setData3(0, 7);
	obj.setData3(1, 8);
	obj.setData3(2, 9);
	//実行前の状態表示
	printf("data1=%d¥n", obj.getData1());
	printf("data2=%.2f¥n", obj.getData2());
	printf("data3={%d, %d, %d}¥n", obj.getData3(0), obj.getData3(1), obj.getData3(2));
	//シリアライズ
	std::ofstream stream("test1.txt");//ファイル出力ストリーム生成
	boost::archive::text_oarchive arc(stream);//出力アーカイブ生成：テキストアーカイブ
	arc << obj;//シリアライズ
	stream.close();//ストリームをクローズ
}
```

【デシリアライズ】

```
//--------------------
//デシリアライズテスト１：テキストアーカイブ
void deserializeTest1()
{
	printf("--------------------¥n");
	printf("デシリアライズ：テキストアーカイブ¥n");
	//オブジェクト生成
	CTest1 obj;
	//デシリアライズ
	std::ifstream stream("test1.txt");//ファイル入力ストリーム生成
	boost::archive::text_iarchive arc(stream);//入力アーカイブ生成：テキストアーカイブ
	arc >> obj;//デシリアライズ
	stream.close();//ストリームをクローズ
	//結果を表示
	printf("data1=%d¥n", obj.getData1());
	printf("data2=%.2f¥n", obj.getData2());
	printf("data3={%d, %d, %d}¥n", obj.getData3(0), obj.getData3(1), obj.getData3(2));
}
```

【テスト】

```
//--------------------
//シリアライズ＆デシリアライズテスト１：テキストアーカイブ
void test1()
{
	//シリアライズ
	serializeTest1();
	//デシリアライズ
	deserializeTest1();
}
```

↓（実行結果）

```
--------------------
シリアライズ：テキストアーカイブ
data1=123
data2=4.56
data3={7, 8, 9}
serialize:version=99    ←シリアライズ実行（バージョンはクラスに設定されているもの）
--------------------
```

```
デシリアライズ：テキストアーカイブ
serialize:version=99    ←デシリアライズ実行　※シリアライズと同じ関数（バージョンはファイルから読み込まれたもの）
data1=123               ←正しくデータが復元されている
data2=4.56              ←（同上）
data3={7, 8, 9}         ←（同上）
```

【test1.txt】※シリアライズの結果、出力されたファイル

```
22 serialization::archive 10 0 99 123 4.5599999 3 7 8 9
```

▼ シリアライズ用テンプレート関数の仕組み

シリアライズ用テンプレート関数は、シリアライズ処理とデシリアライズ処理の両方に兼用する。このテンプレート関数は、与えられたアーカイブクラスによって実体化されるため、アーカイブクラスの「operator&()」の実装に応じて、読み込み処理にも書き込み処理にも対応することができる。

様々な型に対応するために、このオペレータ自体もテンプレート関数となっている。

ライブラリのコードを一部抜粋：

【boost/archive/detail/interface_oarchive.hpp】※一部抜粋

```
template<class Archive>
class interface_oarchive
{
    …（略）…
    // the << operator
    template<class T>
    Archive & operator<<(T & t){
        this->This()->save_override(t, 0);
        return * this->This();
    }
    // the & operator
    template<class T>
    Archive & operator&(T & t){
        return *(this->This()) << const_cast<const T &>(t);
    }
};
```

【boost/archive/detail/interface_iarchive.hpp】※一部抜粋

```
template<class Archive>
class interface_iarchive
{
    …（略）…
    // the >> operator
    template<class T>
    Archive & operator>>(T & t){
        this->This()->load_override(t, 0);
        return * this->This();
    }
    // the & operator
    template<class T>
    Archive & operator&(T & t){
        return *(this->This()) >> t;
    }
};
```

▼ シリアライズのサンプル②：バイナリアーカイブに変更

サンプル①をバイナリアーカイブに変更したサンプルを示す。
シリアライズ対象クラスには変更なし。

バイナリアーカイブを使用したシリアライズとデシリアライズのサンプル：

【インクルード】

```
#include <boost/archive/text_oarchive.hpp>//バイナリ出力アーカイブ
#include <boost/archive/text_iarchive.hpp>//バイナリ入力アーカイブ
```

【シリアライズ】

```
//--------------------
//シリアライズテスト２：バイナリアーカイブ
void serializeTest2()
{
	printf("--------------------¥n");
	printf("シリアライズ：バイナリアーカイブ¥n");
	…（略：テスト１と同じ）…
	std::ofstream stream("test2.bin");//ファイル出力ストリーム生成
	boost::archive::binary_oarchive arc(stream);//出力アーカイブ生成：バイナリアーカイブ
	…（略：テスト１と同じ）…
}
```

【デシリアライズ】

```
//--------------------
//デシリアライズテスト２：バイナリアーカイブ
void deserializeTest2()
{
	printf("--------------------¥n");
	printf("デシリアライズ：バイナリアーカイブ¥n");
	…（略：テスト１と同じ）…
	std::ifstream stream("test2.bin");//ファイル入力ストリーム生成
	boost::archive::binary_iarchive arc(stream);//入力アーカイブ生成：バイナリアーカイブ
	…（略：テスト１と同じ）…
}
```

【テスト】

```
//--------------------
//シリアライズ＆デシリアライズテスト２：バイナリアーカイブ
void test2()
{
	//シリアライズ
	serializeTest2();
	//デシリアライズ
	deserializeTest2();
}
```

↓（実行結果）　※テキストアーカイブと同じ結果

```
--------------------
シリアライズ：バイナリアーカイブ
data1=123
data2=4.56
data3={7, 8, 9}
serialize:version=99
--------------------
デシリアライズ：バイナリアーカイブ
serialize:version=99
data1=123
data2=4.56
data3={7, 8, 9}
```

【test2.bin】※シリアライズの結果、出力されたファイル

```
00000000  16 00 00 00 73 65 72 69  61 6C 69 7A 61 74 69 6F  ....serializatio
00000010  6E 3A 3A 61 72 63 68 69  76 65 0D 0A 00 04 04 04  n::archive......
00000020  08 01 00 00 00 00 63 00  00 00 7B 00 00 00 85 EB  ......c...{.....
00000030  91 40 03 00 00 00 07 08  09                       .@.......
```

▼ シリアライズのサンプル③：XML アーカイブに変更

サンプル①を XML アーカイブに変更したサンプルを示す。

XML はシリアライズの際に「データ名」が必要なため、シリアライズ処理にも変更を加える。「BOOST_SERIALIZABLE_NPV()」というマクロを使用してデータ項目を指定することで、データ名も自動的に指定される。

XML アーカイブを使用したシリアライズとデシリアライズのサンプル：

【インクルード】

```
#include <boost/archive/xml_oarchive.hpp>//XML 出力アーカイブ
#include <boost/archive/xml_iarchive.hpp>//XML 入力アーカイブ
```

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス３
class CTest3
{
	…（略：テスト１と同じ）…
	void serialize(Archive& arc, const unsigned int version)
	{
		…（略：テスト１と同じ）…
		arc & BOOST_SERIALIZATION_NVP(m_data1);//データ１ ※アーカイブに & 演算子でデータを渡す
		arc & BOOST_SERIALIZATION_NVP(m_data2) & BOOST_SERIALIZATION_NVP(m_data3);
                                                    //データ２& データ３ ※連続してデータを渡すことも可能
	}
	…（略：テスト１と同じ）…
};
//--------------------
//シリアライズ設定
BOOST_CLASS_VERSION(CTest3, 99);//バージョン設定
```

【シリアライズ】

```
//--------------------
//シリアライズテスト３：XML アーカイブ
void serializeTest3()
{
	printf("--------------------¥n");
	printf("シリアライズ：XML アーカイブ¥n");
	…（略：テスト１と同じ）…
	std::ofstream stream("test3.xml");//ファイル出力ストリーム生成
	boost::archive::xml_oarchive arc(stream);//出力アーカイブ生成：XML アーカイブ
	arc << BOOST_SERIALIZATION_NVP(obj);//シリアライズ
	…（略：テスト１と同じ）…
}
```

【デシリアライズ】

```
//--------------------
//デシリアライズテスト３：XML アーカイブ
void deserializeTest3()
{
	printf("--------------------¥n");
	printf("デシリアライズ：XML アーカイブ¥n");
	…（略：テスト１と同じ）…
```

```
	std::ifstream stream("test3.xml");//ファイル入力ストリーム生成
	boost::archive::xml_iarchive arc(stream);//入力アーカイブ生成：XML アーカイブ
	arc >> BOOST_SERIALIZATION_NVP(obj);//デシリアライズ
	…（略：テスト１と同じ）…
}
```

【テスト】

```
//--------------------
//シリアライズ＆デシリアライズテスト３：XML アーカイブ
void test3()
{
	//シリアライズ
	serializeTest3();
	//デシリアライズ
	deserializeTest3();
}
```

↓（実行結果）　※テキストアーカイブと同じ結果

```
--------------------
シリアライズ：XML アーカイブ
data1=123
data2=4.56
data3={7, 8, 9}
serialize:version=99
--------------------
デシリアライズ：XML アーカイブ
serialize:version=99
data1=123
data2=4.56
data3={7, 8, 9}
```

【test3.xml】※シリアライズの結果、出力されたファイル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="10">
<obj class_id="0" tracking_level="0" version="99">	←変数名がそのまま XML の項目名になっている
	<m_data1>123</m_data1>			←メンバー名がそのまま XML の項目名になっている
	<m_data2>4.5599999</m_data2>
	<m_data3>
		<count>3</count>
		<item>7</item>
		<item>8</item>
		<item>9</item>
	</m_data3>
</obj>
```

▼ シリアライズのサンプル④：複雑な構造のデータ

シリアライズ対象メンバーに構造体を含む場合、全ての構造体に「serialize」テンプレート関数を実装することで連鎖的にシリアライズできる。

また、ポインタ変数は復元時に自動的にメモリ確保される。

さらに、継承したクラスの場合、子クラスのシリアライズ関数の中から親クラスのシリアライズ関数を呼び出すことも可能。（継承のサンプルは省略する）

複雑な構造のデータのシリアライズとデシリアライズのサンプル：

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス４
```

```
//内部クラス
class CTest4sub
{
public:
	//アクセッサ
	int getVal1() const { return m_val1; }//値 1 取得
	void setVal1(const int value){ m_val1 = value; }//値 1 更新
	int getVal2() const { return m_val2; }//値 2 取得
	void setVal2(const int value){ m_val2 = value; }//値 2 更新
private:
	//シリアライズ用テンプレート関数
	//※シリアライズとデシリアライズの共通処理
	//※private スコープにした上でフレンドクラスを設定する
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("sub::serialize:version=%d¥n", version);//バージョンを表示
		arc & BOOST_SERIALIZATION_NVP(m_val1)//値 1
		    & BOOST_SERIALIZATION_NVP(m_val2);//値 2
	}
private:
	//フィールド
	int m_val1;//値 1
	int m_val2;//値 2
};
//データクラス
class CTest4
{
public:
	//アクセッサ
	int getData1() const { return m_data1; }//データ 1 取得
	void setData1(const int value){ m_data1 = value; }//データ 1 更新
	float getData2() const { return m_data2; }//データ 2 取得
	void setData2(const float value){ m_data2 = value; }//データ 2 更新
	char getData3(const int index) const { return m_data3[index]; }//データ 3 取得
	void setData3(const int index, const int value){ m_data3[index] = value; }//データ 3 更新
	CTest4sub& getData4(){ return m_data4; }//データ 4 取得
	CTest4sub& getData5(const int index){ return m_data5[index]; }//データ 5 取得
	CTest4sub* getData6(){ return m_data6; }//データ 6 取得
	void setData6(CTest4sub* obj){ m_data6 = obj; }//データ 6 更新
	CTest4sub* getData7(){ return m_data7; }//データ 7 取得
	void setData7(CTest4sub* obj){ m_data7 = obj; }//データ 7 更新
private:
	//シリアライズ用テンプレート関数
	//※シリアライズとデシリアライズの共通処理
	//※private スコープにした上でフレンドクラスを設定する
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("serialize:version=%d¥n", version);//バージョンを表示
		arc & BOOST_SERIALIZATION_NVP(m_data1)//データ 1
		    & BOOST_SERIALIZATION_NVP(m_data2)//データ 2
		    & BOOST_SERIALIZATION_NVP(m_data3)//データ 3
		    & BOOST_SERIALIZATION_NVP(m_data4)//データ 4
		    & BOOST_SERIALIZATION_NVP(m_data5)//データ 5
		    & BOOST_SERIALIZATION_NVP(m_data6)//データ 6
		    & BOOST_SERIALIZATION_NVP(m_data7);//データ 7
	}
private:
	//フィールド
	int m_data1;//データ 1：int 型
	float m_data2;//データ 2：float 型
	char m_data3[3];//データ 3：配列型
```

```
	CTest4sub m_data4;//データ４：構造体
	CTest4sub m_data5[2];//データ５：構造体の配列
	CTest4sub* m_data6;//データ６：構造体のポインタ
	CTest4sub* m_data7;//データ７：構造体のポインタ（ヌル時の動作確認用）
};
//--------------------
//シリアライズ設定
BOOST_CLASS_VERSION(CTest4, 99);//バージョン設定
BOOST_CLASS_VERSION(CTest4sub, 100);//バージョン設定
```

【シリアライズ】※扱うデータ項目が増えている以外はとくに変わった処理はなし

```
//--------------------
//シリアライズテスト４：複雑な構造のデータ（XML アーカイブ）
void serializeTest4()
{
	printf("--------------------¥n");
	printf("シリアライズ：複雑な構造のデータ（XML アーカイブ）¥n");
	//オブジェクト生成
	CTest4 obj;
	//テストデータをセット
	obj.setData1(123);
	obj.setData2(4.56f);
	obj.setData3(0, 7);
	obj.setData3(1, 8);
	obj.setData3(2, 9);
	obj.getData4().setVal1(10);
	obj.getData4().setVal2(11);
	obj.getData5(0).setVal1(12);
	obj.getData5(0).setVal2(13);
	obj.getData5(1).setVal1(14);
	obj.getData5(1).setVal2(15);
	obj.setData6(new CTest4sub());//ポインタ変数にはメモリ確保したオブジェクトをセット
	obj.getData6()->setVal1(16);
	obj.getData6()->setVal2(17);
	obj.setData7(nullptr);//ヌルポインタをセット
	//実行前の状態表示
	printf("data1=%d¥n", obj.getData1());
	printf("data2=%.2f¥n", obj.getData2());
	printf("data3={%d, %d, %d}¥n", obj.getData3(0), obj.getData3(1), obj.getData3(2));
	printf("data4={val1=%d, val2=%d}¥n", obj.getData4().getVal1(), obj.getData4().getVal2());
	printf("data5[0]={val1=%d, val2=%d}¥n", obj.getData5(0).getVal1(), obj.getData5(0).getVal2());
	printf("data5[1]={val1=%d, val2=%d}¥n", obj.getData5(1).getVal1(), obj.getData5(1).getVal2());
	printf("data6={val1=%d, val2=%d}¥n", obj.getData6()->getVal1(), obj.getData6()->getVal2());
	printf("data7=0x%p¥n", obj.getData7());
	//シリアライズ
	std::ofstream stream("test4.xml");//ファイル出力ストリーム生成
	boost::archive::xml_oarchive arc(stream);//出力アーカイブ生成：XML アーカイブ
	arc << BOOST_SERIALIZATION_NVP(obj);//シリアライズ
	stream.close();//ストリームをクローズ
}
```

【デシリアライズ】※扱うデータ項目が増えている以外はとくに変わった処理はなし

```
//--------------------
//デシリアライズテスト４：複雑な構造のデータ（XML アーカイブ）
void deserializeTest4()
{
	printf("--------------------¥n");
	printf("デシリアライズ：複雑な構造のデータ（XML アーカイブ）¥n");
	//オブジェクト生成
	CTest4 obj;
	//デシリアライズ
	std::ifstream stream("test4.xml");//ファイル入力ストリーム生成
	boost::archive::xml_iarchive arc(stream);//入力アーカイブ生成：XML アーカイブ
	arc >> BOOST_SERIALIZATION_NVP(obj);//デシリアライズ
	stream.close();//ストリームをクローズ
```

```
	//結果を表示
	printf("data1=%d¥n", obj.getData1());
	printf("data2=%.2f¥n", obj.getData2());
	printf("data3={%d, %d, %d}¥n", obj.getData3(0), obj.getData3(1), obj.getData3(2));
	printf("data4={val1=%d, val2=%d}¥n", obj.getData4().getVal1(), obj.getData4().getVal2());
	printf("data5[0]={val1=%d, val2=%d}¥n", obj.getData5(0).getVal1(), obj.getData5(0).getVal2());
	printf("data5[1]={val1=%d, val2=%d}¥n", obj.getData5(1).getVal1(), obj.getData5(1).getVal2());
	printf("data6={val1=%d, val2=%d}¥n", obj.getData6()->getVal1(), obj.getData6()->getVal2());
	printf("data7=0x%p¥n", obj.getData7());
}
```

【テスト】

```
//--------------------
//シリアライズ＆デシリアライズテスト４：複雑な構造のデータ（XML アーカイブ）
void test4()
{
	//シリアライズ
	serializeTest4();
	//デシリアライズ
	deserializeTest4();
}
```

↓（実行結果）

```
--------------------
シリアライズ：複雑な構造のデータ（XML アーカイブ）
data1=123
data2=4.56
data3={7, 8, 9}
data4={val1=10, val2=11}
data5[0]={val1=12, val2=13}
data5[1]={val1=14, val2=15}
data6={val1=16, val2=17}
data7=0x00000000
serialize:version=99
sub::serialize:version=100
sub::serialize:version=100
sub::serialize:version=100
sub::serialize:version=100
--------------------
デシリアライズ：複雑な構造のデータ（XML アーカイブ）
serialize:version=99
sub::serialize:version=100
sub::serialize:version=100
sub::serialize:version=100
sub::serialize:version=100
data1=123
data2=4.56
data3={7, 8, 9}
data4={val1=10, val2=11}	←内部のオブジェクトもきちんと復元している
data5[0]={val1=12, val2=13}	←配列も大丈夫
data5[1]={val1=14, val2=15}
data6={val1=16, val2=17}	←ポインタも大丈夫（自動的に new されている）
data7=0x00000000	←ヌルポインタも再現
```

【test4.xml】※シリアライズの結果、出力されたファイル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="10">
<obj class_id="0" tracking_level="0" version="99">
	<m_data1>123</m_data1>
	<m_data2>4.5599999</m_data2>
	<m_data3>
		<count>3</count>
		<item>7</item>
		<item>8</item>
```

```
            <item>9</item>
      </m_data3>
      <m_data4 class_id="1" tracking_level="1" version="100" object_id="_0">
            <m_val1>10</m_val1>
            <m_val2>11</m_val2>
      </m_data4>
      <m_data5>
            <count>2</count>
            <item object_id="_1">
                  <m_val1>12</m_val1>
                  <m_val2>13</m_val2>
            </item>
            <item object_id="_2">
                  <m_val1>14</m_val1>
                  <m_val2>15</m_val2>
            </item>
      </m_data5>
      <m_data6 class_id_reference="1" object_id="_3">
            <m_val1>16</m_val1>
            <m_val2>17</m_val2>
      </m_data6>
      <m_data7 class_id="-1"></m_data7>
</obj>
```

▼ シリアライズのサンプル⑤：非侵入型（non-intrusive）

上記のように、内部で保持するオブジェクトをシリアライズするためには、手間をかけてそれぞれのクラスにシリアライズ処理を実装しなければならない。しかし、これがライブラリのクラスであった場合、独自にシリアライズ処理を追加することができない。このような場合、シリアライズ処理をクラス内の関数ではなく、外部関数として実装することで対応できる。これを「非侵入型」（non-intrusive）と呼ぶ。

非侵入型シリアライズ関数のプロトタイプ：

```
namespace boost{
      namespace serialization{//このネームスペースである必要がある（オーバーロード関数扱いになり自動的に呼び出される）
            template <class Archive>
            void serialize(Archive& ar, TYPE& obj, const unsigned int version)
            {
                  ar & obj.***
                     & obj.***
                     & obj.***;
            }
      }
}
```

サンプル④の「CTest4sub」を非侵入型に変更したサンプル：

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス４
//内部クラス
class CTest4sub
{
      …（略）…
private:
#if 0//元の処理を無効化
      //シリアライズ用テンプレート関数
      //※シリアライズとデシリアライズの共通処理
```

```
	//※private スコープにした上でフレンドクラスを設定する
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("sub::serialize:version=%d¥n", version);//バージョンを表示
		arc & BOOST_SERIALIZATION_NVP(m_val1)//値 1
		    & BOOST_SERIALIZATION_NVP(m_val2);//値 2
	}
private:
#else
public://外部関数から扱えるのはパブリックなメンバーに限られる
#endif
	//フィールド
	int m_val1;//値 1
	int m_val2;//値 2
};
```

【非侵入型シリアライズ関数】

```
namespace boost{
	namespace serialization{
		template <class Archive>
		void serialize(Archive& ar, CTest4sub& sub, const unsigned int version)
		{
			printf("[non-intrusive]sub::serialize:version=%d¥n", version);//バージョンを表示
			arc & BOOST_SERIALIZATION_NVP(m_val1)//値 1
			    & BOOST_SERIALIZATION_NVP(m_val2);//値 2
		}
	}
}
```

↓（実行結果）

```
--------------------
シリアライズ：複雑な構造のデータ（XML アーカイブ）
data1=123
data2=4.56
data3={7, 8, 9}
data4={val1=10, val2=11}
data5[0]={val1=12, val2=13}
data5[1]={val1=14, val2=15}
data6={val1=16, val2=17}
data7=0x00000000
serialize:version=99
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
--------------------
デシリアライズ：複雑な構造のデータ（XML アーカイブ）
serialize:version=99
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
data1=123
data2=4.56
data3={7, 8, 9}
data4={val1=10, val2=11}
data5[0]={val1=12, val2=13}
data5[1]={val1=14, val2=15}
data6={val1=16, val2=17}
data7=0x00000000
```

【test4.xml】※シリアライズの結果、出力されたファイル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

```
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="10">
<obj class_id="0" tracking_level="0" version="99">
	<m_data1>123</m_data1>
	<m_data2>4.5599999</m_data2>
	<m_data3>
		<count>3</count>
		<item>7</item>
		<item>8</item>
		<item>9</item>
	</m_data3>
	<m_data4 class_id="1" tracking_level="1" version="100" object_id="_0">
		<sub.m_val1>10</sub.m_val1>
		<sub.m_val2>11</sub.m_val2>
	</m_data4>
	<m_data5>
		<count>2</count>
		<item object_id="_1">
			<sub.m_val1>12</sub.m_val1>
			<sub.m_val2>13</sub.m_val2>
		</item>
		<item object_id="_2">
			<sub.m_val1>14</sub.m_val1>
			<sub.m_val2>15</sub.m_val2>
		</item>
	</m_data5>
	<m_data6 class_id_reference="1" object_id="_3">
		<sub.m_val1>16</sub.m_val1>
		<sub.m_val2>17</sub.m_val2>
	</m_data6>
	<m_data7 class_id="-1"></m_data7>
</obj>
```

▼ 標準ライブラリのクラスのシリアライズ

「std::string」や「std::vector」などの標準ライブラリのデータをシリアライズすために、Boost C++ライブラリはそれらのシリアライズ用処理を多数用意している。下記のファイルをインクルードすれば使える。

標準ライブラリ用シリアライズ処理のインクルード：※主な例

```
#include <boost/serialization/string.hpp>//std::string 用
#include <boost/serialization/vector.hpp>//std::vector 用
#include <boost/serialization/list.hpp>//std::list 用
#include <boost/serialization/map.hpp>//std::map 用
#include <boost/serialization/set.hpp>//std::set 用
#include <boost/serialization/deque.hpp>//std::deque 用
```

▼ シリアライズのサンプル⑥：バージョン互換処理

例えば、新しくデータ項目が追加された後で、その項目が存在しない古いデータを読み込む場合がある。

そのようなデータを読み込むためのバージョン互換処理は、シリアライズ用テンプレート関数に渡ってくるバージョンをチェックして、処理を振り分ければよい。

シリアライズテンプレート関数のバージョン互換処理サンプル：

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス１
class CTest1
{
	…（略）…
	//シリアライズ用テンプレート関数
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("serialize:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ１
		arc & m_data2 & m_data3;//データ２ & データ３
		if(version >= 101)
		{
			//Ver.101 以上の場合
			arc & m_data4;//データ４
		}
		else
		{
			//Ver.101 以前の場合
			//※古いバージョンが渡ってくるのはデシリアライズの時しかありえないので、
			//  直接値を更新してしまっても問題ない
			m_data4 = 99;//データ４
		}
	}
private:
	//フィールド
	int m_data1;//データ１：int 型
	float m_data2;//データ２：float 型
	char m_data3[3];//データ３：配列型
	int m_data4;//データ４　※Ver.101 より追加
};
//--------------------
//シリアライズ設定
BOOST_CLASS_VERSION(CTest1, 101);//バージョン設定
```

▼ シリアライズのサンプル⑦：セーブ処理とロード処理を分ける

複雑なバージョン互換処理が必要な場合、セーブ処理とロード処理を分けて扱うことですっきりした処理構造にすることができる。セーブ処理にはバージョン互換処理が必要ないためである。

「BOOST_SERIALIZATION_SPLIT_MEMBER()」というマクロをクラス内に配置すると、セーブ処理とロード処理を分けて扱うクラスとなる。その場合、「serialize()」関数の代わりに「save()」関数と「save()」関数を定義する。

基本的なシリアライズとデシリアライズのサンプル：

```
#include <boost/serialization/split_member.hpp>//BOOST_SERIALIZATION_SPLIT_MEMBER 用　※このインクルードが必要

//--------------------
//シリアライズ対象クラス１
class CTest1
{
	…（略）…
```

```
	//シリアライズ用テンプレート関数
	friend class boost::serialization::access;
#if 0
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("serialize:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ 1
		arc & m_data2 & m_data3;//データ 2 & データ 3
	}
#else
	BOOST_SERIALIZATION_SPLIT_MEMBER();
	template<class Archive>
	void save(Archive& arc, const unsigned int version) const
	{
		printf("save:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ 1
		arc & m_data2 & m_data3;//データ 2 & データ 3
	}
	template<class Archive>
	void load(Archive& arc, const unsigned int version)
	{
		printf("load:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ 1
		arc & m_data2 & m_data3;//データ 2 & データ 3
		if(version >= 101)
		{
			//Ver.101 以上の場合
			arc & m_data4;//データ 4
		}
		else
		{
			//Ver.101 以前の場合
			m_data4 = 99;//データ 4
		}
	}
#endif
	…（略）…
```

↓（実行結果）

```
--------------------
シリアライズ：テキストアーカイブ
data1=123
data2=4.56
data3={7, 8, 9}
save:version=99
--------------------
デシリアライズ：テキストアーカイブ
load:version=99
data1=123
data2=4.56
data3={7, 8, 9}
```

■ 要件定義

以上を踏まえて、ゲームのセーブデータのためのシリアライズ処理に対する要件を定義する。

▼ 基本要件

本書が扱うシステムの基本要件は下記のとおり。

・ ゲームタイトル固有のセーブデータを、形式にとらわれることなく自由に扱うことができるシリアライズ処理とする。

・ バージョンを管理し、セーブデータの構造が変わっても、（できる限り）古いデータの読み込みが可能なシステムとする。
  - もしくは、そのための簡潔な仕組みを提供するシステムとする。

・ プログラマーの労力をできる限り抑えるシステムとする。
  - シリアライズとデシリアライズの処理コードを共通化できるものとする。
  - これはコードの簡略化を意味するが、プログラムサイズを小さくすることを意味するものではない。生産性を優先し、テンプレートによるプログラムサイズの肥大は許容するものとする。

・ シリアライズの際は、ゲームシステム中に分散する多数のデータをひとまとめにしてセーブデータを構成できるものとする。
  - このための任意の処理を、簡潔に定義可能とする。
  - この処理をシリアライズと区別し、データの「収集」を意味して「コレクト」と呼び、それを担う処理を「コレクター」と呼ぶものとする。

・ デシリアライズの際は、必要に応じてデータオブジェクトの追加・更新・削除を適切に行えるものとする。
  - このための任意の処理を、簡潔に定義可能とする。
    - 追加・削除の判断をシステムが自動的に行わない。
    - Boost 版のような自動的な new も行わない。
  - この処理をデシリアライズと区別し、データの「分配」を意味して「ディストリビュート」と呼び、それを担う処理を「ディストリビュータ」と呼ぶものとする。
  - ディストリビュートの際は、復元対象となるデータ、もしくは、復元対象となり得るデータを全て一掃して再構築するものとする。
    - そういう処理を任意に記述できるものとする。
  - デシリアライズ（ディストリビュート）により、稼働中のゲームに影響を与える可能性があるが、それは使う側の問題である。

・ シリアライズは、メモリ上のバッファに対してセーブデータのイメージを作成するものとする。
  - 直接ファイルに出力するようなことはしない。
  - 固定バッファの範囲内で処理するものとする。

  - 途中でバッファオーバーフローした場合、シリアライズは失敗とする。
    - 部分的な成功はなく、一切が失敗とする。

- デシリアライズは、メモリ上のセーブデータイメージから取り込むものとする。
  - 直接ファイルから読み込むようなことはしない。

- シリアライズの際は、元のデータに一切変更を加えないことを保証するものとする。
  - const オブジェクトとして処理し、シリアライズ処理中に直接データを更新するような誤ったコーディングがあったらコンパイルエラーになるものとする。
  - 【要対処】デシリアライズ時のバージョン互換のために、直接データを更新する処理が必要になることもあるが、デシリアライズ処理とシリアライズ処理のコードを共通化するとそれができなくなる。
    - この対処として、デシリアライズ時専用の処理を別メソッドで追加できるものとする。

- シリアライズ処理に変更を加えることなく、バイナリアーカイブとテキストアーカイブを扱えるものとする。
  - 【可能であれば】テキストアーカイブは JSON 形式とする。
  - テキストアーカイブにより、手作業によるセーブデータの量産を可能とし、QA 作業等の効率化に寄与する。

- 【できれば】デシリアライズ時のオプションにより、部分ロードを可能とする。
  - 例えば、プレイヤーキャラの成長状態のデータと、所持アイテムのデータ、ゲームの進行状態（章とフラグ）のデータを、それぞれ別々のセーブデータから読み込んで組み合わせることを可能とする。これにより、デバッグ効率を向上させる。
  - 部分ロードでは、ディストリビュータは部分ロードの対象データのみ（例えば所持アイテム系データのみ）を一掃して再構築するように挙動を変えるものとする。
    - そういう処理を任意に記述できるものとする。
  - 【要対処】部分ロードにより、データの不整合が生じる可能性がある。例えば、所持アイテムを部分ロードすることにより、プレイヤーキャラが装備していた武器が失われ、参照できなくなるなど。デシリアライズおよびディストリビュートの基本的な仕組みとして、このような問題についてはサポートしないものとする。
    - この対処は、ディストリビュータの処理で任意に対応する。
    - 部分ロードであることを判別して整合性の解消を行う。

- セーブデータに要求されることの多い「チェックサム」や「暗号化」は、本システムの対象外とする。
  - そうした処理は、本システムが作成したセーブデータイメージに対して実施するものとする。
  - ロード時は、そうした処理を通過した後のセーブデータイメージを本システムに受け

渡すものとする。

▼ 要求仕様／要件定義

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

● シリアライズシステムの構成要素

シリアライズシステムは、下記の要素で構成するものとする。
各要素の基本要件も合わせて示す。

【汎用処理】アーカイブ

シリアライズシステムの本体。

- 汎用クラスであり、ゲームタイトル固有の処理を実装することはない。
- セーブデータ読み書きのためのバッファを受け取って処理する。
- ユーザー定義処理（ゲームタイトル固有の処理）であるシリアライズの実行を担い、セーブデータの作成・復元を実行する。
- アーカイブには複数の種類があり、使用するアーカイブによってセーブデータの保存形式が異なる。
- 下記の基本クラスで構成する。なお、これらのクラスは直接使用しない。
  - ・ アーカイブ書き込みクラス
  - ・ アーカイブ読み込みクラス
  - ・ バイナリ形式アーカイブクラス
  - ・ テキスト形式アーカイブクラス
- 上記の基本クラスを組み合わせて、下記のクラスを構成する。
  - ・ バイナリ形式アーカイブ書き込みクラス
  - ・ バイナリ形式アーカイブ読み込みクラス
  - ・ テキスト形式アーカイブ読み込みクラス
  - ・ テキスト形式アーカイブ読み込みクラス

【ユーザー定義処理】シリアライズ

シリアライズとデシリアライズのための共通処理コード。

- ユーザー定義関数であり、ゲームタイトル固有の処理を扱う。

- テンプレート関数として定義する。
  - ・ これにより、コンパイル時にシリアライズのコードとデシリアライズのコードをそれぞれ生成する。
- 基本的に、データクラス本体にシリアライズ処理を内蔵せず、外部定義関数として構成する。
  - ・ Boost 版でいうところの「非侵入型（non-intrusive)」のみを扱うものとする。
- データクラス一つに対して一つのシリアライズ関数を定義する。
  - ・ セーブデータに必要なオブジェクトの数だけ連鎖的に多数のシリアライズが実行される。
- シリアライズ処理の中では、シリアライズの対象とするデータ項目の指定を行うのが基本。
  - ・ Boost 版と同様。
  - ・ 同じクラス内のメンバーでも、シリアライズ対象に指定されなかった項目はシリアライズされない。
  - ・ 対象項目がクラスの場合、そのクラスに対して定義されたシリアライズ処理が連鎖的に実行される。

【ユーザー定義処理】セーブ専用処理

シリアライズと同じだが、シリアライズ時にしか実行されない。（デシリアライズ時には実行されない）

- この処理を「セーブ」としたのは Boost 版の模倣。

【ユーザー定義処理】ロード専用処理

シリアライズと同じだが、デシリアライズ時にしか実行されない。（シリアライズ時には実行されない）

- この処理を「ロード」としたのは Boost 版の模倣。

【ユーザー定義処理】ロード前処理

デシリアライズの際、セーブデータを読み込む前に実行される処理。

- 読み込み先のインスタンスを生成したりなど、必要に応じてデシリアライズの準備を行う。
- Boost 版では読み込み先のポインタがヌルなら自動的に new するが、それは対応しない。
- Boost 版にはない処理。

【ユーザー定義処理】ロード後処理

デシリアライズの際、セーブデータを読み込んだ後に実行される処理。

- 読み込んだデータに基づいてポインタの参照先を割り当てるなど、事後処理や整合性解消処理などを行う。
- Boost 版にはない処理。

【ユーザー定義処理】ロード結果通知

デシリアライズの結果、セーブデータと現在のデータ構造との不整合を通知する処理。

- シリアライズ対象データとして指定されているが、セーブデータになかったデータ項目を通知。
- セーブデータには存在するが、シリアライズ対象データとして指定されておらず、読み込めなかったデータ項目を通知。
- Boost 版にはない処理。

【ユーザー定義処理】コレクター

シリアライズの際、他のオブジェクトをいっしょにセーブデータに保存するために収集する処理。

- 直接関連性のないオブジェクトをひとまとめにするために用いる。
  - ・ クラスのメンバー変数としてのクラスなら普通のシリアライズで処理できる。
- オブジェクトを集めてシリアライズを実行する処理を任意に記述する。
- コレクターで集めたデータを復元するためのディストリビュータと必ず一対で定義する。
- Boost 版にはない処理。

【ユーザー定義処理】ディストリビュータ

デシリアライズの際、コレクターによってセーブデータに記録されたオブジェクトを分配（復元）するための処理。

- コレクターによって記録されたオブジェクトの情報が通知される。
  - ・ コレクターが記録した情報の数だけ繰り返し呼び出される。
- 対象オブジェクトに見合ったインスタンス生成の処理とそれに対するデシリアライズを任意に記述する。
- 必ずコレクターと一対で定義する。
- Boost 版にはない処理。

【ユーザー定義処理】ディストリビュータ前処理

一連のディストリビュータが実行される前に実行される処理。

- 例えば、プールアロケータを用いるオブジェクトを扱う場合、ディストリビュータではプールからのメモリ割り当てを行うが、その前のプール自体の初期化を、前処理として実施する。

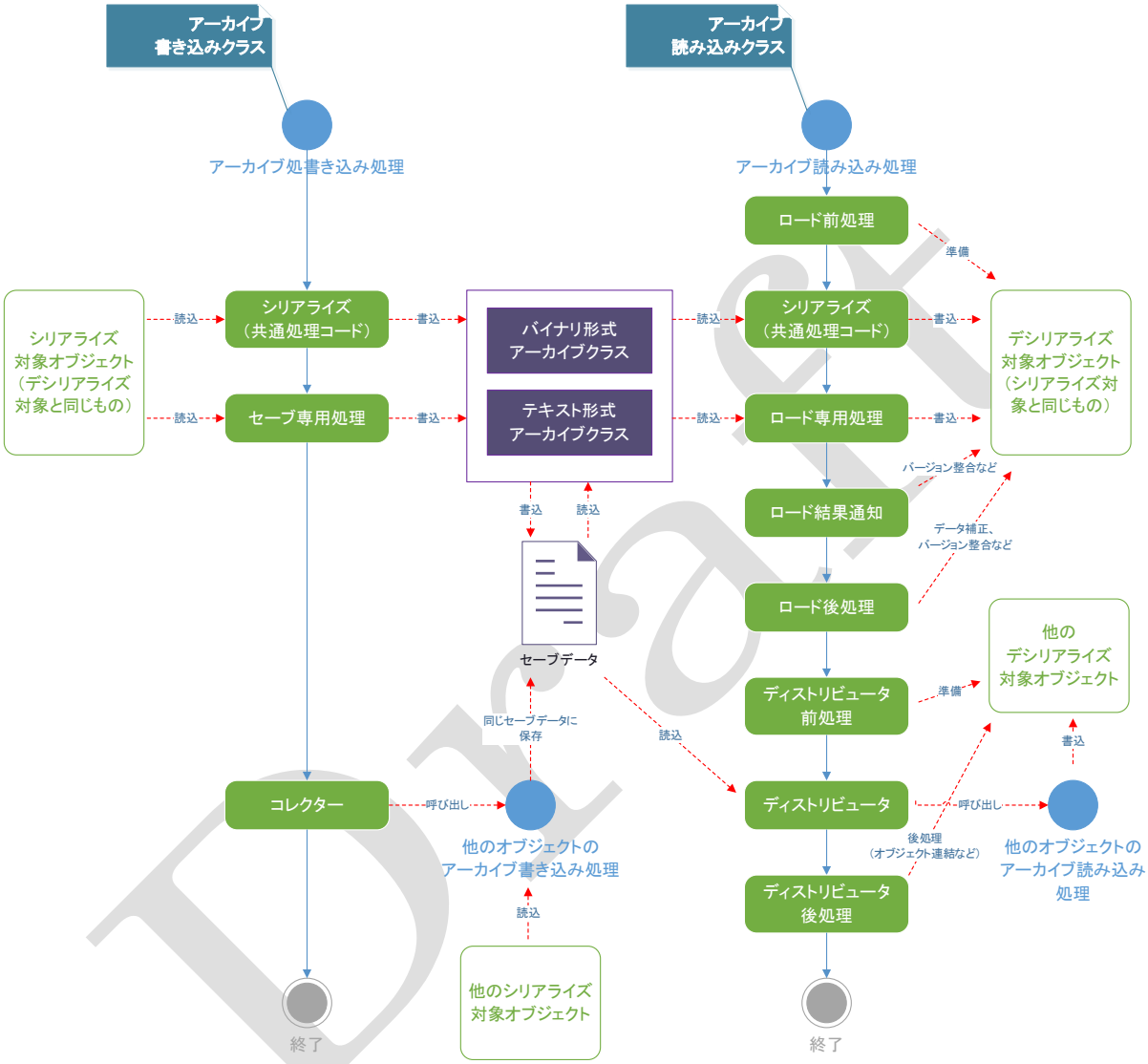
【ユーザー定義処理】ディストリビュータ後処理

一連のディストリビュータが実行された後に実行される処理。

- 例えば、オブジェクト間の関連付けや、部分ロード時の不整合解消などを行う。

構成要素の関係図

以上の構成要素の関係図を示す。



● プログラミングモデルに対する基本要件

- 全般的にテンプレートを活用することで、コンパイル時にユーザー定義処理が共通処理に取り込まれてコード生成されるものとする。
  - これにより、ユーザー定義処理は必要最小限のコーディングで済むものとする。
  - テンプレートクラスの特殊化、および、テンプレート関数のオーバーロードを利用すると、明示的にテンプレート引数で型を与えることなく、共通処理にユーザー定義処理を渡すことができるため、コーディングを簡略化できる。（Boost 版と同様の手法）

- これによるコードサイズの肥大は注意すべきものとなるが、コーディングの利便性の方を重視するものとする。

● ユーザー定義処理のプログラミングに関する要件

- ユーザー定義処理は、Boost 版と同様の簡潔な記述でコーディングできるものとする。
- ユーザー定義処理の種類が多いが、不要なものは定義しなくても良いものとする。
- セーブする対象のクラス（構造体）に、シリアライズ／デシリアライズのための処理を記述しなくてもよいものとする。
  - Boost 版ではクラス内にシリアライズのコードを記述するのが基本だが、それには対応しない。
  - セーブ／ロード処理だけを切り分けてまとめることで、簡潔なコードになるようにする。
- ユーザー定義処理は、Boost 版の非侵入型（non-intrusive）と同様の外部関数のみに対応するものとする。
- ユーザー定義処理は、Boost 版と異なり、テンプレート関数ではなく、テンプレートクラスによる関数オブジェクトで実装するものとする。
  - 【理由】テンプレートクラスを用いると、特殊化（と SFINAE 機構）を利用して、ユーザー定義処理の実装状態を静的に（コンパイル時に）チェックすることができるため。

以下、Boost 版と比べて余計に記述しなければならない部分を赤字で示す。若干構造が異なるぐらいで、コード量には大差ない。

【Boost 版】※非侵入型のテンプレート関数

```
namespace boost{
    namespace serialization{
        template<class Archive>
        void serialize(Archive& arc, const USER_DATA& obj,const unsigned int version)
        {
            arc & obj.m_a;
            arc & obj.m_b;
            arc & obj.m_c;
        }
    }
}
```

【本システム】※テンプレートクラスによる関数オブジェクト

```
namespace serial{
    template<class Archive>
    struct serialize<Archive, USER_DATA> {//←特殊化のためにテンプレート引数に型を明示する必要がある
        void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
        {
            arc & pair("data1", obj.m_a);
            arc & pair("data2", obj.m_b);
            arc & pair("data3", obj.m_c);
        }
    }
};
```

- バージョンは、Boost 版と同様に、クラス（構造体）ごとに設定可能とする。
- バージョンの扱いは、Boost 版と異なり、メジャーバージョンとマイナーバージョンの組み合わせで定義できるものとする。

【Boost 版】※BOOST_CLASS_VERSION マクロで定義

```
struct USER_DATA
{
	int m_a
	float m_b;
	char m_c[3];
};
BOOST_CLASS_VERSION(USER_DATA, 2);//USER_DATA Ver.2.0
```

【本システム】※SERIALIZE_VERSION_DEF マクロで定義

```
SERIALIZE_VERSION_DEF(USER_DATA, 2, 1);//USER_DATA Ver.2.1
```

- デシリアライズ時には、Boost 版と同様に、セーブデータから読み込んだバージョンがユーザー定義処理に渡されるものとする。
- ユーザー定義処理に渡されるバージョンは、Boost 版と異なり、メジャーバージョンとマイナーバージョンを格納した「バージョン定義クラス」で渡されるものとする。
- ユーザー定義処理に渡されるバージョンは、Boost 版と異なり、セーブデータから読み込んだバージョンだけではなく、現在のバージョンも渡されるものとする。
  - セーブデータのバージョンが現在のバージョンと同じかどうかを素早くチェックするぐらいしか使い道がないが、少しだけ便利になるので。

【Boost 版】

```
template<class Archive>
void serialize(Archive& arc, const USER_DATA& obj,const unsigned int version)
```

【本システム】

```
template<class Archive>
struct serialize<Archive, USER_DATA> {
	void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
	{
		if(ver < now_ver)//オブジェクト同士を直接比較可能
```

- ユーザー定義処理内では、Boost 版と同様に、アーカイブオブジェクトに対する「&」演算子でシリアライズ対象のデータ項目（メンバー変数）を指定するものとする。
  - テンプレートにより、アーカイブ読み込み時とアーカイブ書き込み時で「&」演算子の振る舞いを変えることができるため、シリアライズとデシリアライズの処理コードを共通化できる。
- シリアライズ対象データ項目を指定する際は、Boost 版と異なり、必ず「名称」を与えるものとする。
  - バイナリデータ時にもデータ項目の識別子として保存する。
  - この「名称」（を CRC にしたもの）を記録することで、本システムの要点である「ディストリビュータ」の処理を成立させることができる。
    - ユーザー定義処理内で、「名称」に対する分配先（復元先）のデータを指定することができる。

- なお、Boost 版でも XML アーカイブを扱う場合は名称を必要とするが、それはマクロによって自動的に与えられる。本システムでは、「コレクター」と「ディストリビュータ」（後述）のように、複数の処理で名前を合わせて処理する必要があるため、必ず明示的に名前を指定するものとする。

【Boost 版】

```
template<class Archive>
void serialize(Archive& arc, const USER_DATA& obj,const unsigned int version)
{
	arc & obj.m_a;
	arc & obj.m_b & obj.m_c;//演算子の連結も可
}
```

【Boost 版の XML アーカイブ時】※マクロを使用して自動的に XML の項目名を与えている

```
template<class Archive>
void serialize(Archive& arc, const USER_DATA& obj,const unsigned int version)
{
	arc & BOOST_SERIALIZATION_NVP(obj.m_a);
	arc & BOOST_SERIALIZATION_NVP(obj.m_b) & BOOST_SERIALIZATION_NVP(obj.m_c);//演算子の連結も可
}
```

【本システム】※pair()関数を使用して名前を与える

```
template<class Archive>
struct serialize<Archive, USER_DATA> {
	void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
	{
		arc & pair("data1", obj.m_a);//「&」演算子でシリアライズ対象データ項目を指定
					//※（pair 関数を使用し、必ず名前とペアで指定）
		arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);//演算子の連結も可
	}
};
```

➢ シリアライズの実行は、Boost 版と同様に、アーカイブオブジェクトとシリアライズ対象オブジェクトを「<<」演算子でつないで実行するものとする。

- この時もやはり「名称」を与える。

【Boost 版】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//ファイル出力ストリーム生成
std::ofstream os("file.bin");
//アーカイブ書き込みオブジェクト生成
boost::archive::binary_oarchive arc(os);
//シリアライズ
arc << user_data;
//ストリームをクローズ
os.close();
```

【Boost 版の XML アーカイブ時】

```
//アーカイブ書き込みオブジェクト生成
boost::archive::xml_oarchive arc(os);
//シリアライズ
arc << BOOST_SERIALIZATION_NVP(user_data);
```

【本システム】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//バッファ
static const std::size_t SAVE_DATA_BUFF_SIZE = 32 *1024;
char save_data_buff[SAVE_DATA_BUFF_SIZE];
```

```
static const std::size_t WORK_BUFF_SIZE = 32 *1024;
char work_buff[WORK_BUFF_SIZE];
//アーカイブ書き込みオブジェクト生成
serial::COBinaryArchive arc(save_data_buff, SAVE_DATA_BUFF_SIZE, work_buff, WORK_BUFF_SIZE);
//シリアライズ
arc << serial::pair("UserData", user_data);//実行時には名前を与える
```

- デシリアライズの実行は、Boost 版と同様に、アーカイブオブジェクトとシリアライズ対象オブジェクトを「>>」演算子でつないで実行するものとする。
  - この時もやはり「名称」を与える。
  - 以下、シリアライズ時と異なる箇所を赤字で示す。

【Boost 版】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//ファイル入力ストリーム生成
std::ifstream is("file.bin");
//アーカイブ読み込みオブジェクト生成
boost::archive::binary_iarchive arc(is);
//デシリアライズ
arc >> user_data;
//ストリームをクローズ
is.close();
```

【Boost 版の XML アーカイブ時】

```
//アーカイブ読み込みオブジェクト生成
boost::archive::xml_iarchive arc(is);
//デシリアライズ
arc >> BOOST_SERIALIZATION_NVP(user_data);
```

【本システム】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//セーブデータバッファ取得
std::size_t save_data_size = 0;
void* save_data = getLoadedSaveDataFile(&save_data_size);
//バッファ
static const std::size_t WORK_BUFF_SIZE = 32 *1024;
char work_buff[WORK_BUFF_SIZE];
//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//デシリアライズ
arc >> serial::pair("UserData", user_data);//実行時には名前を与える
```

- 「<<」演算子および「>>」演算子が指定できるのは、ユーザー定義処理が定義されたクラスもしくは構造体に限定するものとする。
  - テンプレートの特殊化などに頼った処理構造のため、任意のクラス（構造体）として識別する必要がある。
  - int 型や char の配列などのデータを直接シリアライズすることができない。

- Boost 版と同様に、クラスのプライベートメンバーにアクセスするための手段を用意するものとする。
  - Boost 版と異なり、非侵入型の処理に対してフレンド化する手段を提供する。
  - 簡潔に定義できるように、専用マクロを用意する。

【Boost 版】※侵入型（クラス／構造体内に直接シリアライズ処理を実装）のみ対応

```
class USER_DATA
{
private:
	int m_a
	float m_b;
	char m_c[3];
	friend class boost::serialization::access;//フレンド化
	//シリアライズ処理
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		arc & obj.m_a;
		arc & obj.m_b & obj.m_c;
	}
};
```

【本システム】※SERIALIZE_FRIEND マクロで定義

```
class USER_DATA
{
private:
	int m_a
	float m_b;
	char m_c[3];
	FRIEND_SERIALIZE(USER_DATA);//フレンド化　※内部でテンプレートパラメータを明示するために、
	                            //　　自身のクラス名を記述する必要あり（残念）
};
//シリアライズ処理
namespace serial{
	template<class Archive>
	struct serialize<Archive, USER_DATA> {
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
		}
	};
}
```

- ユーザー定義処理のシリアライズ処理（共通処理コード）ではデータを直接書き換えることを禁止するものとする。
  - データの書き換えは「ロード後処理」を利用するものとする。
  - ロード後処理は、必要な時だけ定義する。

【問題のあるシリアライズ処理】

```
//シリアライズ処理
namespace serial{
	template<class Archive>
	struct serialize<Archive, USER_DATA> {//↓USER_DATA に const が付く
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
			//バージョン整合処理
			if (ver >= CVersion(1, 2))//バージョンが 1.2 以上なら m_d 読み込み（もしくは書き込み）
			{
				arc & pair("data4", obj.m_d);
			}
			else//バージョン 1.2 未満ならセーブデータに m_d がないので直接値を代入
			{
				obj.m_d = 123;//初期値代入
				//　　↑コンパイルエラー！
```

```
                        //※ロードの時だけ実行されることを意図したコードだが、
                        //  シリアライズとの共通処理内ではデータの更新が許可されない
                }
        }
    };
}
```

↓（変更後）

```
//シリアライズ処理
namespace serial{
    template<class Archive>
    struct serialize<Archive, USER_DATA> {
        void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
        {
            arc & pair("data1", obj.m_a);
            arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
            //バージョン整合処理
            if (ver >= CVersion(1, 2))//バージョンが 1.2 以上なら m_d 読み込み（もしくは書き込み）
            {
                arc & pair("data4", obj.m_d);
            }
        }
    };
}
//ロード後処理
namespace serial{
    template<class Archive>
    struct afterLoad<Archive, USER_DATA> {//↓USER_DATA に const が付かない
        void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
        {
            //バージョン 1.2 未満ならセーブデータに m_d がないので直接値を代入
            if (ver < CVersion(1, 2))
            {
                obj.m_d = 123;//初期値代入
                //      ↑コンパイル OK
            }
        }
    };
}
```

- 同様に「ロード前処理」でもデータの書き換えを可能とする。
  - 事前のメモリ割り当てなどを行うことが可能。
  - ロード前処理は、必要な時だけ定義する。

【ロード前処理】

```
//ロード前処理
namespace serial{
    template<class Archive>
    struct beforeLoad<Archive, USER_DATA> {//↓USER_DATA に const が付かない
        void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
        {
            obj.m_d = new EXTRA_DATA;//データを読み込むための領域を事前に割り当てておく
        }
    };
}
```

- シリアライズ対象に指定されたデータ項目が記録されていないセーブデータを読み込んでも処理がスキップされるだけで、他のデータが問題なく読み込めるものとする。
  - 構造体にメンバーが追加された後で古いセーブデータを読み込む場合など。

【シリアライズ処理】

```
//シリアライズ処理
namespace serial{
	template<class Archive>
	struct serialize<Archive, USER_DATA> {
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
			arc & pair("data4", obj.m_d);//←バージョン 1.2 以降に追加されたメンバー
			//	（特殊な処理をせずとも問題なし）
		}
	};
}
```

- シリアライズ対象のデータ項目が記録されていないセーブデータを読み込んだ場合、「通知処理」により、問題が発生したデータ項目が通知されるものとする。
  - プログラム上で新たにデータ項目が追加された後に、古いセーブデータを読み込んだ場合などに起こる。
  - 問題が検出されたデータ項目の数だけ呼び出される。
  - （そのオブジェクトの）一通りのデータ項目の読み込みが済んだ後、「ロード後処理」の前に呼び出される。
  - 通知処理は、必要な時だけ定義する。

【保存先の指定があるが、セーブデータになくロードできなかったデータ項目の通知処理】

```
//保存先の指定があるが、セーブデータになくロードできなかったデータ項目の通知
namespace serial{
	template<class Archive>
	struct noticeUnloadedItem<Archive, USER_DATA> {//↓USER_DATA に const が付かない
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
			, const CItemBase& unloaded_item)
		{	//↑読み込めなかったデータ項目が渡される
			//data4 はバージョン 1.2 未満のセーブデータには存在しない
			if (unloaded_item == "data4")//引数 unloaded_item で問題の項目をチェック
			{
				obj.m_d = 123;//初期値代入
			}
			//バージョンで比較して処理をせずとも、確実な整合処理を行うことができる
		}
	};
}
```

- シリアライズ対象のデータ項目ではない、セーブデータにだけ存在するデータ項目があっても問題なく動作し、専用の「通知」処理で通知も行われるものとする。
  - プログラム上でデータ項目を削除、もしくは改名した後に、古いセーブデータを読み込んだ場合などに起こる。
  - 問題が検出されたデータ項目の数だけ呼び出される。
  - 読み込みの途中、問題が見つかったタイミングで呼び出されるため、正しい読み込み先を指定

し直して再度読み込みすることができる。

- 通知処理は、必要な時だけ定義する。

【セーブデータにはあったが、保存先の指定がなく、ロードできなかったデータ項目の通知処理】

```
//セーブデータにはあったが、保存先の指定がなく、ロードできなかったデータ項目の通知処理
namespace serial{
	template<class Archive>
	struct noticeUnrecognizedItem<Archive, USER_DATA> {//↓USER_DATA に const が付かない
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
		                                              , const CItemBase& unrecognized_item)
		{                                             //↑読み込めなかったデータ項目が渡される
			//data_x は現在 data5 に改名
			if (unrecognized_item == "data_x")//引数 unrecognized_item で問題の項目をチェック
			{
				arc & pair("data5", obj.m_e5);//読み込み先を指定し直して再実行
				//「&」演算子で改めてシリアライズ対象項目を指定し直すことで
				//データの読み込み先を委譲する

				//この関数内では obj の値を更新してもよい
				obj.m_preblemItem = unrecognized_item.m_nameCrc;//問題があった項目を記録
			}
		}
	};
}
```

- プログラム上の構造体のメンバーの順序とそれをシリアライズ対象項目として指定する順序、それと、（古い）セーブデータ上のデータ項目の順序は、それぞれ違っていても問題なく読み込みできるものとする。
  - プログラム上で構造体のメンバーの順序を変更しても不整合が起こらないものとする。
  - このための任意の処理や設定をユーザー定義する必要はないものとする。
    - 順序が異なるデータ項目をシステムに伝える手続きは不要。
  - データ項目名に基づいて自動的に整合性の解消が行われるものとする。
  - なお、書き込みはシリライズ対象に指定したデータ項目の順に行われるものとする。
- プログラム上の構造体のメンバーの型や配列のサイズが変更されても、（古い）セーブデータを問題なく読み込みできるものとする。
  - この変更によって発生する問題は居所的なものであり、それ以外のデータは正しく読み込めるものとする。
  - このための任意の処理や設定をユーザー定義する必要はないものとする。
    - 型や配列サイズが異なるデータ項目をシステムに伝える手続きは不要。
  - 問題のあったデータも、読み込み可能な範囲で読み込むものとする。
    - 例えば、int → short に変わった項目は、2 バイトの読み込みを行う。
    - また、例えば、short → int に変わった項目は、2 バイトの読み込み、残りバイトを 0 で埋める。
    - リトルエンディアンなら前詰め、ビッグエンディアンなら後詰めで処理する。
    - 配列の要素ごとにこの変換を行う。
    - 配列の要素数が縮小されている場合は、余りを読み捨てる。

・ 配列の要素数が拡大されている場合は、残りを無視する。(0 で初期化したりもしない)

・ オーバーフローによるデータ破壊、セーブデータの読み込み位置のずれといった、他のデータにも影響が及ぶことはないようにする。

➢ ユーザー定義処理の「コレクター」と「ディストリビュータ」を定義することにより、対象オブジェクトと直接関連性のないデータを一つのセーブデータとしてまとめて扱えるものとする。

- コレクターとディストリビュータは、必ずセットで定義する。
- コレクターとディストリビュータは、必要な時だけ定義する。
- コレクターの処理で、複数のデータのシリアライズを行うと、ディストリビュータはそのデータの数だけ呼び出される。
- ディストリビュータには、「ディストリビュート前処理」と「ディストリビュート後処理」があり、それぞれ一連のディストリビュート処理の前後に 1 回ずつ実行される。

・ デシリアライズ前のデータの削除や、デシリアライズ後のデータ整合処理などに用いる。

【コレクター】

```
//コレクター
template<class Arc>
struct collector<Arc, USER_DATA> {
	void operator()(Arc& arc, const USER_DATA& obj, const CVersion& ver)
	{
		//USER_DATA の後に連続して下記のデータもいっしょにシリアライズする

		//インベントリのデータを収集
		CSingleton<CInventory> inventory;
		for (auto item_data : *inventory)//ループ処理で全要素を取得
		{
			//アイテムデータを一つずつシリアライズ
			arc << pair("item", *item_data);
		}
		//進行＆フラグデータをシリアライズ
		CSingleton<CPhaseAndFlags> phase_and_flags;
		arc << pair("phase_and_flags", *phase_and_flags);
	}
};
```

【ディストリビュート前処理】

```
//ディストリビュート前処理
template<class Arc>
struct beforeDistribute<Arc, USER_SAVE_DATA> {//↓USER_DATA に const が付かない
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
			const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
			//↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
			//  渡さてくる
	{
		//データを読み込む前に、読み込み先を一旦空にする

		//インベントリデータクリア
		CSingleton<CInventory> inventory;
		inventory.destroy();
		//フェーズ＆進行データクリア
		CSingleton<CPhaseAndFlags> phase_and_flags;
		phase_and_flags.destroy();
	}
};
```

【ディストリビュータ】

```
//ディストリビュータ
template<class Arc>
struct distributor<Arc, USER_DATA> {//↓USER_DATAにconstが付かない
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
		const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
		                                                    const CItemBase& target_item)
		//↑objが配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
		//  渡される（ディストリビュート前処理と同じ
		                                                    //↑対象データ項目が渡される
	{
		//USER_DATAの後に連続して下記のデータもいっしょにデシリアライズされる

		//コレクターでシリアライズしたデータの数だけ繰り返し実行されるので、
		//データ項目名を判定して一つずつ処理する
		//※無視したらデータは読み込まれずに次のデータに進む

		if (target_item == "item")//インベントリデータか？
		{
			//インベントリデータ復元
			CSingleton<CInventory> inventory;
			ITEM_DATA item_data;
			arc >> pair("item", item_data);//デシリアライズ
			inventory->regist(item_data);//インベントリにデータを登録
		}
		else if (target_item == "phase_and_flags")//進行データか？
		{
			//フェーズ＆進行データ復縁
			CSingleton<CPhaseAndFlags> phase_and_flags;
			arc >> pair("phase_and_flags", *phase_and_flags);//デシリアライズ
		}
	}
};
```

【ディストリビュート後処理】

```
//ディストリビュート後処理
template<class Arc>
struct afterDistribute<Arc, USER_DATA> {//↓USER_DATAにconstが付かない
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
		const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
		//↑objが配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
		//  渡される（ディストリビュート前処理と同じ
	{
		//全ての読み込みが完了
		//インベントリデータが更新されたので、キャラが所持する武器データの参照を再設定する
		CSingleton<CCharaList> chara_list;
		for (auto& chara_data : *chara_list)//ループ処理で全キャラを取得
		{
			chara_data->attachItems();//アイテムを参照し直す
		}
	}
};
```

- セーブデータの部分ロードは、ディストリビュータを工夫することによって実現可能とする。
  - 例えば、「インベントリデータは『セーブデータ A』から読み込み、進行データは『セーブデータ B』から読み込んで組み合わせる」といったことができると、QA や制作のためのセーブデータの組み合わせパターンを用意する手間を軽減できる。
  - このような処理は、ここまでに示したシステムの応用で十分対応可能となる。

【部分ロード実行】

```
//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//部分ロード設定
setLoadTarget("phase_and_flags");//部分ロード指定：進行データのみをロード
resetLoadTarget();//部分ロードを解除して全体ロードする場合
//デシリアライズ
arc >> serial::pair("UserData", user_data);
```

【部分ロードのための処理】※シリアライズの仕組みとは直接関係のない処理

```
//部分ロード設定
crc32_t s_loadTarget;//ロード対象データ項目名（CRC）
void setLoadTarget(const char* name)//ロード対象データ項目名をセット
{
	s_loadTarget = calcCRC32(name));
}
void resetLoadTarget()//ロード対象データ項目をリセット
{
	s_loadTarget = 0;
}
bool isLoadTarget(const char* name)//ロード対象データ項目か？
{
	return s_loadTarget == 0 || s_loadTarget == calcCRC32(name_crc);
}
```

【ディストリビュート前処理】※部分ロード処理を追加

```
//ディストリビュート前処理
template<class Arc>
struct beforeDistribute<Arc, USER_DATA> {
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
			const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
	{
		//データを読み込む前に、読み込み先を一旦空にする
		//※部分ロード時は、部分ロードの対象データのみを破棄

		if(isLoadTarget("item"))
		{
			//インベントリデータクリア
			CSingleton<CInventory> inventory;
			inventory.destroy();
		}
		if(isLoadTarget("phase_and_flags"))
		{
			//フェーズ＆進行データクリア
			CSingleton<CPhaseAndFlags> phase_and_flags;
			phase_and_flags.destroy();
		}
	}
};
```

【ディストリビュータ】※部分ロード処理を追加

```
//ディストリビュータ
template<class Arc>
struct distributor<Arc, USER_DATA> {
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
			const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
				const CItemBase& target_item)
	{
		//USER_DATA の後に連続して下記のデータもいっしょにデシリアライズされる

		if (target_item == "item" && isLoadTarget(target_item))//インベントリデータか？
		{
			//インベントリデータ復元
			CSingleton<CInventory> inventory;
			ITEM_DATA item_data;
```

```
                arc >> pair("item", item_data);//デシリアライズ
                inventory->regist(item_data);//インベントリにデータを登録
            }
            else if (target_item == "phase_and_flags" && isLoadTarget(target_item))//進行データか？
            {
                //フェーズ＆進行データ復縁
                CSingleton<CPhaseAndFlags> phase_and_flags;
                arc >> pair("phase_and_flags", *phase_and_flags);//デシリアライズ
            }
        }
};
```

【ディストリビュート後処理】※部分ロード処理を追加

```
//ディストリビュート後処理
template<class Arc>
struct afterDistribute<Arc, USER_DATA> {
    void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
    {
        //全ての読み込みが完了
        //インベントリデータが更新されているなら、キャラが所持する武器データの参照を再設定する
        if(isLoadTarget("item"))
        {
            CSingleton<CCharaList> chara_list;
            for (auto& chara_data : *chara_list)//ループ処理で全キャラを取得
            {
                chara_data->attachItems();//アイテムを参照し直す
            }
        }
    }
};
```

- 細かい互換性維持を捨てて、シリアライズ対象項目をひとくくりに扱う手段も用意するものとする。
  - ここまでの要件で示したとおり、すべてのシリアライズ対象項目を「&」演算子で指定さえすれば、バージョン互換性に優れた利便性の高いセーブデータシステムにすることができるが、反面、コードサイズの肥大化やセーブデータサイズの肥大化、処理の低速化が懸念される。
  - この対処として、データ項目の型を無視してバイトデータのかたまりとして記録する手段を用意するものとする。

【構造体まるごとをバイトデータのかたまりとして扱う】

```
struct USER_DATA
{
    int m_a
    float m_b;
    char m_c[3];
    struct SUB
    {
        int x;
        float y;
        short z[5];
    } m_sub;//サブ構造体のメンバー
};

template<class Archive>
struct serialize<Archive, USER_DATA> {
    void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
    {
        arc & pair("data1", obj.m_a);
        arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
```

```
            arc & pairBin("sub", obj.m_sub);//構造体まるごとをバイトデータとして扱う
            //※本来はサブ構造体を１項目ずつ指定するか、
            //  struct serialize<Archive, USER_DATA::SUB> を定義する
      }
};
```

- ただし、この「pairBin」の仕組みが使えるのは「&」演算子に対してのみ。「<<」演算子、および、「>>」演算子は、その仕組み上「型」が指定されている必要があるため、バイトデータ扱いにできない。（通常でも、これらの演算子には、ユーザー定義処理に対応したクラスもしくは構造体しか指定できない）

【>>演算子、<<演算子に pairBin の指定は不可】

```
//アーカイブ書き込みオブジェクト生成
serial::COBinaryArchive arc(save_data_buff, SAVE_DATA_BUFF_SIZE, work_buff, WORK_BUFF_SIZE);
//シリアライズ
arc << serial::pairBin("UserData", user_data);//←NG

//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//デシリアライズ
arc >> serial::pairBin("UserData", user_data);//←NG
```

➢ （主にテキストアーカイブのために）char 型の配列と文字列を区別してシリアライズするための仕組みを用意するものとする。

- これにより、テキストアーカイブ時に、char 型の配列は数値で扱われ、文字列は文字列として扱われるようにする。
- ポインタ型の変数も扱えるようにする。
- ポインタ型の場合、配列の要素数を自動的に読み取ることができないので、要素数を明示する必要がある。

【文字列を扱う】

```
struct USER_DATA
{
      char m_data[10];//数値データの配列
      char m_str[10];//文字列データ
      char* m_dataP;//数値データ（ポインタ型）
      char* m_strP;//文字列データ（ポインタ型）
};

template<class Archive>
struct serialize<Archive, USER_DATA> {
      void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
      {
            arc & pair("data", obj.m_data);//数値の配列（通常通りの指定方法）
            arc & pairStr("str", obj.m_str);//文字列
            arc & pairArray("data_arr", obj.m_dataP, 10);//ポインタを数値の配列として扱う
            //arc & pair( “data_p, obj_m_dataP);//この場合は配列ではなく char 型の値一つ（のポインタ）
                                                //として扱う
            arc & pairStr("str_p", obj.m_strP);//文字列ポインタ（文字列長は自動的に求める／ヌル時は 0）
            //※ポインタ型は、デシリアライズ時には先に領域が割り当てられている必要がある
      }
};
```

➢ （主にセーブデータの起点として）コレクターとディストリビュータだけを使用したい場合、クラスのインスタンスを用意しなくてもシリアライズ／デシリアライズを実行できるものとする。

- セーブデータに多数のデータを集めることだけを目的としたクラスの場合、それ自体はシリアライズする必要がない場合がある。

【インスタンスのないクラスでシリアライズ／デシリアライズ】

```
//セーブデータ用クラス
class SAVE_DATA_TOP{};//中身なし

//コレクター
template<class Arc>
struct collector<Arc, SAVE_DATA_TOP> {
	void operator()(Arc& arc, const USER_DATA& obj, const CVersion& ver)
	{
		//多数のシリアライズ処理
		//…（略）…
	}
};
//ディストリビュータ
template<class Arc>
struct distributor<Arc, SAVE_DATA_TOP> {
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
			const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
				const CItemBase& target_item)
	{
		//多数のデシリアライズ処理
		//…（略）…
	}
};

//アーカイブ書き込みオブジェクト生成
serial::COBinaryArchive arc(save_data_buff, SAVE_DATA_BUFF_SIZE, work_buff, WORK_BUFF_SIZE);
//シリアライズ
arc << serial::pair<SAVE_DATA_TOP>("SaveData");//←インスタンス不要（クラス名と名前だけ指定）

//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//デシリアライズ
arc >> serial::pair<SAVE_DATA_TOP>("SaveData";//←インスタンス不要（クラス名と名前だけ指定）
```

● アーカイブに関する要件

アーカイブクラスがシリアライズ／デシリアライズ処理の本体。

➢ アーカイブクラスもまたテンプレートで構成する。

- 基本の「アーカイブ読み込み／書き込みクラス」に対して、「アーカイブ形式クラス」を与えることでインスタンス化するものとする。

➢ 「アーカイブ形式クラス」は、後からの追加が可能なものとする。

- アーカイブ読み込み／書き込みクラスは、特定のアーカイブ形式クラスに特化しない、（できるだけ）汎用的な処理構造とする。

・ 例えば、バイナリ形式の読み込み処理の場合、順次データを読み込みながら処理をするため、バッファのカレントポインタを操作するが、テキスト形式は事前のパースが必要な可能性がある。もし、共通処理中にバッファのシークがあると、バイナリ形式以外の形式が扱えなくなる可能性がある。読み込んだセーブデータイメージをどのように扱うかは、完全にアーカイブ形式クラスに任せ、抽象化したインターフェースでデータの入出力を求める構造とする。

■ 処理仕様

▼ プログラミングイメージ

プログラミングのイメージは要件定義に示したとおり。

事前にユーザー定義処理を定義し、シリアライズしたいオブジェクトをアーカイブオブジェクトに渡すだけで良い。

以下、要件定義の内容の繰り返しになるが、シリアライズ／デシリアライズに必要なプログラミング要素を簡単に列挙する。詳しい説明については要件定義を参照。

● シリアライズ／デシリアライズの基本形

最低限シリアライズに必要な要素。

【クラス／構造体の定義とフレンド宣言】※フレンド宣言はprivateメンバーへのアクセスが必要な時のみ

```
//シリアライズ対象のクラス／構造体
struct USER_DATA
{
	int m_a
	float m_b;
	char m_c[3];
	struct SUB
	{
		int x;
		float y;
		short z[5];
	} m_sub;//サブ構造体のメンバー
	char m_data[10];//数値データの配列
	char m_str[10];//文字列データ
	char* m_dataP;//数値データ（ポインタ型）
	char* m_strP;//文字列データ（ポインタ型）
	FRIEND_SERIALIZE(USER_DATA);//【必要があれば】フレンド化設定（privateメンバーにアクセスする場合のみ）
};
```

【クラス／構造体のバージョン設定】※バージョン設定の必要がある時のみ

```
SERIALIZE_VERSION_DEF(USER_DATA, 2, 1);//【必要があれば】USER_DATA Ver.2.1
```

【シリアライズ／デシリアライズ共通処理を定義】

```
//シリアライズ＆デシリアライズ共通処理
namespace serial {
```

```
	template<class Archive>
	struct serialize<Archive, USER_DATA> {
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);//「&」演算子でシリアライズ対象データ項目を指定
							//※（pair 関数を使用し、必ず名前とペアで指定）
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);//演算子の連結も可
			arc & pairBin("sub", obj.m_sub);//構造体まるごとをバイトデータとして扱う
			arc & pair("data", obj.m_data);//数値の配列（通常通りの指定方法）
			arc & pairStr("str", obj.m_str);//文字列
			arc & pairArray("data_arr", obj.m_dataP, 10);//ポインタを数値の配列として扱う
			//arc & pair( “data_p, obj_m_dataP);//この場合は配列ではなく char 型の値一つ（のポインタ）として扱う
			arc & pairStr("str_p", obj.m_strP);//文字列ポインタ（文字列長は自動的に求める／ヌル時は 0）
		}
	};
};
```

【シリアライズ実行】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//バッファ
static const std::size_t SAVE_DATA_BUFF_SIZE = 32 *1024;
char save_data_buff[SAVE_DATA_BUFF_SIZE];
static const std::size_t WORK_BUFF_SIZE = 32 *1024;
char work_buff[WORK_BUFF_SIZE];
//アーカイブ書き込みオブジェクト生成
serial::COBinaryArchive arc(save_data_buff, SAVE_DATA_BUFF_SIZE, work_buff, WORK_BUFF_SIZE);
//シリアライズ
arc << serial::pair("UserData", user_data);//実行時には名前を与える
```

【デシリアライズ実行】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//セーブデータバッファ取得
std::size_t save_data_size = 0;
void* save_data = getLoadedSaveDataFile(&save_data_size);
//バッファ
static const std::size_t WORK_BUFF_SIZE = 32 *1024;
char work_buff[WORK_BUFF_SIZE];
//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//デシリアライズ
arc >> serial::pair("UserData", user_data); //実行時には名前を与える
```

● ロード前処理

デシリアライズの際、データを読み込む準備を行いたい場合、「ロード前処理」を定義する。

【ロード前処理】

```
//ロード前処理
namespace serial{
	template<class Archive>
	struct beforeLoad<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			obj.m_dataP = new char[10];//ロード前にメモリ割り当て
			obj.m_strP = new char[10 + 1];//ロード前にメモリ割り当て
		}
	};
}
```

● ロード後処理

デシリアライズの際、バージョン判定や読み込んだデータの状態などに基づいてデータを更新したい場合、「ロード後処理」を定義する。

【ロード後処理】

```
//ロード後処理
namespace serial{
	template<class Archive>
	struct afterLoad<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			//バージョン1.2未満ならセーブデータに m_b がないので直接値を代入
			if (ver < CVersion(1, 2))//バージョン判定
			{
				obj.m_b = 1.23f;//初期値代入
			}

			//読み込みの結果、 m_strP が空文字列なら nullptr にする
			if(obj.m_strP[0] == '¥0')
			{
				delete obj.m_strP;//削除
				obj.m_strP = nullptr;
			}
		}
	};
}
```

● 通知処理①：セーブデータにないデータ

「明示的なバージョン判定に頼らず、古いセーブデータに存在しないデータがあったら初期値をセットしたい」という処理を行うには、そのようなデータの通知を受け取る処理を定義する。

【セーブデータに存在しないデータ項目の通知】

```
//保存先の指定があるが、セーブデータになくロードできなかったデータ項目の通知
namespace serial{
	template<class Archive>
	struct noticeUnloadedItem<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
		                                              , const CItemBase& unloaded_item)
		{                                             //↑読み込めなかったデータ項目が渡される
			//data2(m_b)はバージョン1.2未満のセーブデータには存在しない
			if (unloaded_item == "data2")//引数 unloaded_item で問題の項目をチェック
			{
				obj.m_b = 123;//初期値代入
			}
		}
	};
}
```

● 通知処理②：セーブデータにしかないデータ

「データ名を変更したが、古いセーブデータの古い名前で読み込みたい」という処理を行うには、そのようなデータの通知を受け取る処理を定義する。

【セーブデータにしか存在しないデータ項目の通知】

```
//セーブデータにはあったが、保存先の指定がなく、ロードできなかったデータ項目の通知処理
namespace serial{
	template<class Archive>
	struct noticeUnrecognizedItem<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
                                                    , const CItemBase& unrecognized_item)
		{                                           //↑読み込めなかったデータ項目が渡される
			//data_x は現在 data3(m_c)に改名
			if (unrecognized_item == "data_x")//引数 unrecognized_item で問題の項目をチェック
			{
				arc & pair("data3", obj.m_c);//読み込み先を指定し直して再実行
				//「&」演算子で改めてシリアライズ対象項目を指定し直すことで
				//データの読み込み先を委譲する

				//この関数内では obj の値を更新してもよい
				obj.m_preblemItem = unrecognized_item.m_nameCrc;//問題があった項目を記録
			}
		}
	};
}
```

● コレクターとディストリビュータ

クラス（構造体）のメンバーではないデータをセーブデータにひとまとめにするには、「コレクター」と「ディストリビュータ」をセットで定義する。

【コレクター】

```
//コレクター
template<class Arc>
struct collector<Arc, USER_DATA> {
	void operator()(Arc& arc, const USER_DATA& obj, const CVersion& ver)
	{
		//USER_DATA の後に連続して下記のデータもいっしょにシリアライズする

		//インベントリのデータを収集
		CSingleton<CInventory> inventory;
		for (auto item_data : *inventory)//ループ処理で全要素を取得
		{
			//アイテムデータを一つずつシリアライズ
			arc << pair("item", *item_data);
		}
		//進行＆フラグデータをシリアライズ
		CSingleton<CPhaseAndFlags> phase_and_flags;
		arc << pair("phase_and_flags", *phase_and_flags);
	}
};
```

【ディストリビュート前処理】

```
//ディストリビュート前処理
template<class Arc>
struct beforeDistribute<Arc, USER_DATA> {
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
		const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
```

```
                //↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
                //  渡さてくる
    {
        //データを読み込む前に、読み込み先を一旦空にする

        //インベントリデータクリア
        CSingleton<CInventory> inventory;
        inventory.destroy();
        //フェーズ＆進行データクリア
        CSingleton<CPhaseAndFlags> phase_and_flags;
        phase_and_flags.destroy();
    }
};
```

【ディストリビュータ】

```
//ディストリビュータ
template<class Arc>
struct distributor<Arc, USER_DATA> {//↓USER_DATA に const が付かない
    void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
                                                        const CItemBase& target_item)
                //↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
                //  渡される（ディストリビュート前処理と同じ
                                                        //↑対象データ項目が渡される
    {
        //USER_DATA の後に連続して下記のデータもいっしょにデシリアライズされる

        //コレクターでシリアライズしたデータの数だけ繰り返し実行されるので、
        //データ項目名を判定して一つずつ処理する
        //※無視したらデータは読み込まれずに次のデータに進む

        if (target_item == "item")//インベントリデータか？
        {
            //インベントリデータ復元
            CSingleton<CInventory> inventory;
            ITEM_DATA item_data;
            arc >> pair("item", item_data);//デシリアライズ
            inventory->regist(item_data);//インベントリにデータを登録
        }
        else if (target_item == "phase_and_flags")//進行データか？
        {
            //フェーズ＆進行データ復縁
            CSingleton<CPhaseAndFlags> phase_and_flags;
            arc >> pair("phase_and_flags", *phase_and_flags);//デシリアライズ
        }
    }
};
```

【ディストリビュート後処理】

```
//ディストリビュート後処理
template<class Arc>
struct afterDistribute<Arc, USER_DATA> {//↓USER_DATA に const が付かない
    void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
                //↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
                //  渡される（ディストリビュート前処理と同じ
    {
        //全ての読み込みが完了
        //インベントリデータが更新されたので、キャラが所持する武器データの参照を再設定する
        CSingleton<CCharaList> chara_list;
        for (auto& chara_data : *chara_list)//ループ処理で全キャラを取得
        {
            chara_data->attachItems();//アイテムを参照し直す
        }
    }
```

```
};
```

なお、これらの処理を工夫することで、データの部分ロードに対応することができる。例えば、「『セーブデータ A』をロード後、インベントリだけ『セーブデータ B』の状態にする」といった制御が可能。詳しい方法は要件定義で説明。

▼ クラス設計

本システムのクラス図を示す。

▼ 処理フロー

デシリアライズ時のプール処理（シリアライズ関数を抜けた後にまとめて実行される仕組み）

▼ シリアライズの特殊な仕組み：ネストした構造体

デシリアライズ時のプール処理（シリアライズ関数を抜けた後にまとめて実行される仕組み）

▼ デシリアライズの特殊な仕組み①：定義順序に依存しない読み込み

デシリアライズ時のプール処理（シリアライズ関数を抜けた後にまとめて実行される仕組み）

▼ デシリアライズの特殊な仕組み②：ネストした構造体

デシリアライズ時のプール処理（シリアライズ関数を抜けた後にまとめて実行される仕組み）

▼ デシリアライズの特殊な仕組み③：セーブデータにしかないデータの委譲

デシリアライズ時のプール処理（シリアライズ関数を抜けた後にまとめて実行される仕組み）

■ データ仕様

▼ シリアライズデータの基本構造

■ 処理実装サンプル：シリアライズの使用サンプル

▼ サンプルプログラムについて

▼ 【準備】ゲームのデータ管理システムのサンプル

▼ データ管理システムのクラス図

▼ データ管理システムのサンプルプログラム：定義部

【】

```
```

【】

```
```

【】

```
```

▼ データ管理システムのサンプルプログラム：テスト処理部

【】

【】

【】

▼ データ管理システムに対するシリアライズ処理

【】

【】

【】

▼ システムの依存関係

■ 処理実装サンプル：シリアライズの実装

▼ サンプルプログラムについて

▼ インクルードとネームスペース

【】

【】

【】

▼ バージョンクラス

【】

【】

【】

▼ シリアライズ関数オブジェクト用テンプレートクラス

【】

【】

【】

▼ 型操作クラス

特殊化を利用して、（ゲームタイトル固有の）任意のクラスを追加することも可能。

【】

【】

【】

▼ データ項目管理クラス

【】

【】

【】

▼ 処理結果クラス

【】

【】

【】

▼ アーカイブ読み書きクラス

【】

【】

【】

▼ アーカイブ形式クラス

【】

【】

【】

▼ システムの依存関係

■■以上■■

■ 索引

索引項目が見つかりません。

セーブデータのためのシリアライズ処理

以　上