OCTA

ソフトマテリアルのための統合化シミュレータ

プラットフォームインターフェースライブラリ

libplatform

リファレンスマニュアル

Version 3.3.2

OCTA ユーザーズグループ

AUG. 08 2007

執筆者

第 1～5 章、7 章以降 西尾裕三

第 6 章 佐々木誠

改訂・付録B～C 西本正博

プログラム開発者

西尾裕三

西本正博

庄野敦子

山科純

佐々木誠

謝辞

本プログラム開発は、経済産業省の出資・補助を受け、新エネルギー・産業技術総合開発機構(NEDO)が(財)化学技術戦略推進機構に委託した、大学連携型産業科学技術研究開発プロジェクト「高機能材料設計プラットフォーム」通称「土井プロジェクト」の下で行われたものである。

本プログラムの改良は、独立行政法人科学技術振興機構の補助を受け、「多階層的バイオレオシミュレータの研究開発」における「バイオレオシミュレータ用プラットフォーム」機能改良ソフト開発の下で行われたものである。

目次

図目次

表目次

第１章　概要

libplatformは、UDFファイルと解析エンジン入出力のインタフェース機能をライブラリ化したものであり、解析エンジンとリンクして使用するものである。入出力インタフェースは、C++言語の機能を用いてカプセル化されており、解析エンジン設計者は、UDF定義に基づきインタフェースクラスを準備し、そのオブジェクトにストリーム形式でUDFデータを読み込み、同様に書き出すことができる。

V2.0以降では、主として高速化を実現するためサーバマシンを利用しないスタンドアローン用ライブラリとなっている。これに伴い、解析エンジンのI/OするUDFファイルはローカルに存在するテキストファイルとなった。オープンしたUDFファイル内容は、構文解析された後メモリー上にキャッシュされるため、高速なデータアクセスを実現している。

V3.0では、GOURMETからリモートサーバで稼動する解析エンジンを制御し、また解析エンジンの稼動状況を監視する機能が追加された。また、UDFの新機能であるグローバルデータの書込み機能をサポートした。

UDFデータのI/O方式として、UDFManagerインタフェースとIObjectインタフェースの２種類のインタフェースを提供している。UDFManagerインタフェースは、データ名の解決を実行時に行うレイトバインディング方式を採用し、キャッシュメモリに直接I/Oを行うものであり、GOURMETで使用されるPythonスクリプトと同等の操作を提供している。このため、UDFへの入門や簡単な構造のUDFデータを扱うツール類での利用に適している。一方、IObjectインタフェースはC++へのバインドクラスを自動生成して利用するアーリーバインディング方式を採用し、ストリーム形式でI/Oを行うものであり、複雑な構造のUDFデータを扱う解析エンジンでの利用に適している。

V3.2では、FORTRAN77,Fortran90,Fortran95およびANSI CからUDFManagerを利用するインタフェースが追加された。

V3.3では、「makeinterface」ツール（UDFファイルの定義情報を解析して、その定義内容と同じUDFオブジェクトをC++プログラムで入出力するためのインターフェースクラス定義を生成するツール）に改良を加え、エンジン開発者が長期間に渡ってUDFファイルをメンテナンスする作業をサポートするための機能として、複数の異なるバージョンのUDFファイル（通常少しずつデータ構造が変化していく）の差異を解析し、自動的にバージョンに合致したUDFファイルを入出力できるインタフェースクラス定義を生成するようにした。

第２章　制約事項

・ エンジン実装言語としてC++言語、C言語およびFortran言語の利用を前提としている。

・ IObjectインタフェースでは、C++言語でのデータ構造とのインタフェースを前提としている。具体的には、構造体型のデータ構造をインタフェースクラスに静的にマッピングする方法をサポートしている。

・ UDFManagerインタフェースでは、C言語およびFortran言語でのデータ構造とのインタフェースを前提としている。具体的には、UDFのデータ変数名を実行時に動的に解釈してデータ抽出する方法をサポートしている。

・ コンパイル・リンクに用いる環境は、Linux (kernel 2.2.9 以上) およびCygwin(gcc3.3 以上) とする。Linuxでのコンパイル・リンクには、gcc 3.2 以上、glibc 2.1.2 以上が必要である。Microsoft が提供する開発環境については、Microsoft Visual C++ (6.0 SP3 以上)および、Microsoft .net framework に対応している。

第３章　主要機能

OCTA全体の主要機能を図 3.1 に示す。この仕様書で記述されている範囲は、エンジンとシステムのインタフェース部分であり、枠内に示した部分である。

図 3.1 における各機能の定義を以下に示す。

・ ＵＤＦヘッダ情報取得・更新

エンジンプログラムが指定したUDFのヘッダ情報を取得または更新する。

・ 入力ファイル読込

エンジンプログラムが、指定したUDFのレコード内から特定のUDFシンボルの値を読みこむ。

・ 出力ファイル新規作成

エンジンプログラムが、既存の構造定義ファイルを指定して、UDFを新規作成する。

・ 出力ファイル書込み

エンジンプログラムが、指定したUDFのレコード内へ指定したUDFシンボルの値を追加または更新する。

・ エンジン制御情報取得

エンジンプログラムが、ユーザの指示するエンジン制御情報を取得する。

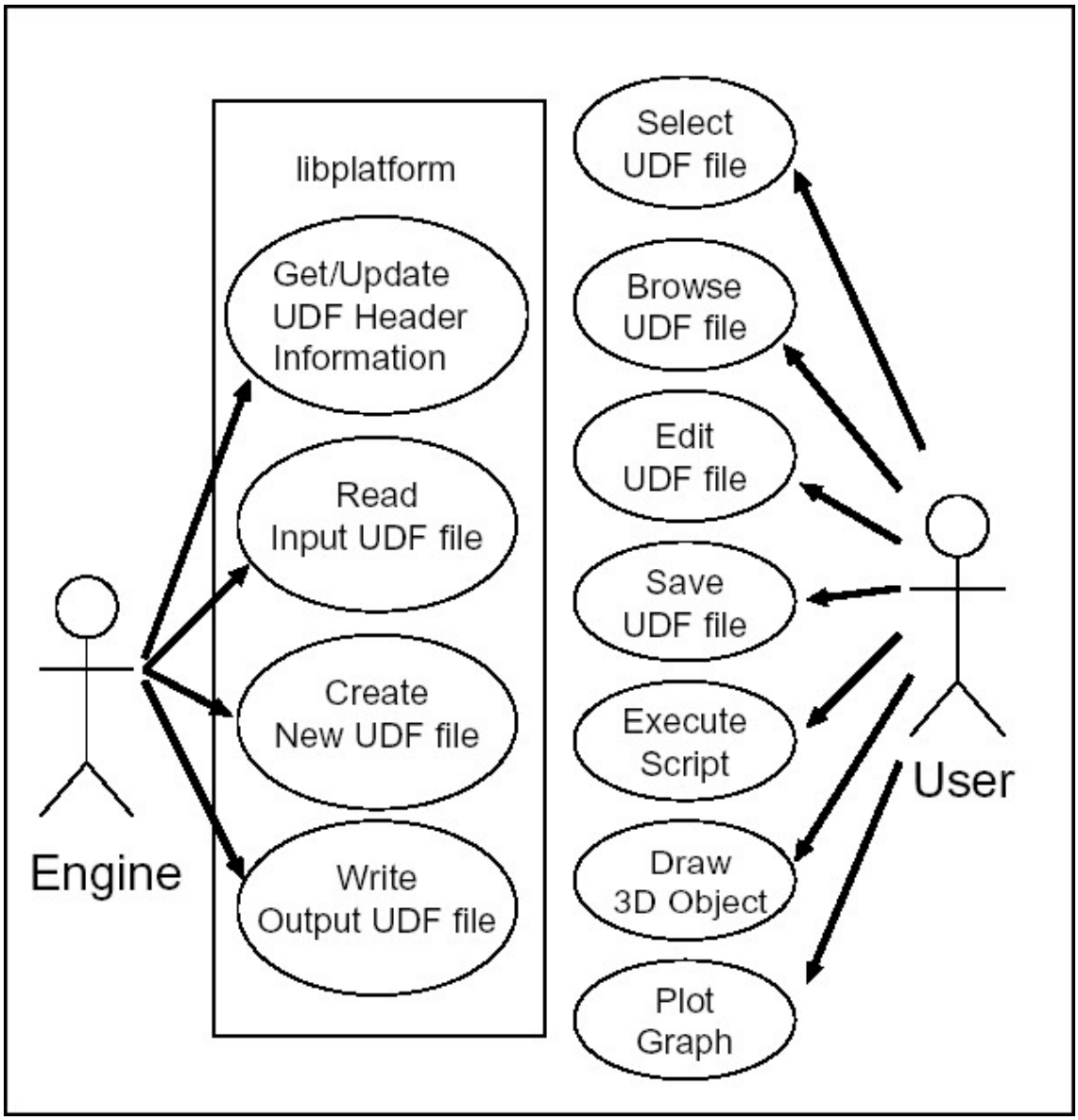

libplatform
Get/Update
UDF Header
Information
Read
Input UDF file
Create
New UDF file
Write
Output UDF file
Engine
Select
UDF file
Browse
UDF file
Edit
UDF file
Save
UDF file
Execute
Script
Draw
3D Object
Plot
Graph
User

図 3.1：主要機能

第４章　ライブラリ構造

プラットフォームライブラリは、以下のC++パッケージとFortran/C APIからなる。

- STL

  コンパイラベンダから提供される標準C++テンプレートライブラリ。使用しているクラスは以下の通りである。

  string, iostream, fstream, strstream, list, vector, map, algorithm

  プラットフォームライブラリでは、STL ライブラリを各パッケージで使用している。

- UDFhandle

  エンジンがUDF データを読み書きするための基本インタフェース。インタフェース定義は、udfhandle.h ヘッダーファイルに記述されている。

- Pfinterface

  UDFhandleを隠蔽するためのインタフェース。将来的なC、FORTRANエンジンへの対応拡張のため準備している。

  インタフェース定義は、pfinterface.h にある。

- UDFstream

  UDFとエンジン間のデータ交換方式を、C++言語のストリームとして隠蔽するためのパッケージ。

  インタフェース定義は、udfstream.h にある。

- ErrorHandling

  例外処理を行うパッケージ。

  インタフェース定義は、pfexception.h にある。

- UDFObject

  UDF のデータ構造をC++言語のクラスにマッピングするための基本インタフェースクラス定義のパッケージ。

  インタフェース定義は、udfobject.h にある。

- IObject

  エンジンのUDF インタフェースを実現するパッケージ。UDFObject を継承して、個々のエンジンで

用いるインタフェースオブジェクトを実装する。
インタフェース定義は、ICognacin.h、IMuffin_solid.h など
UDFファイルを解析してこのパッケージを自動生成するツール「makeinterface」を提供している。
このツールの利用方法については、第８ 章「インタフェースクラス自動生成ツール」を参照のこと。
簡便な入門用解説が付録A「makeinterface Tutorial」にまとめられている。

・UDFManager

UDFObject、IObjectとは別の方式（レイトバインディング方式）でUDFインタフェースを実現するパッケージ。このパッケージは、C言語、Fortran言語によるプログラミングを行うユーザを対象としており、UDFの利用を簡便に行なうためのものである。UDFManagerの利用者は、他のパッケージの内容を知っている必要は一切無い。
簡便な入門用解説が別マニュアル「UDF/PF Tutorial」にまとめられている。なお、このパッケージを利用して作成したソースコードは、共通ライブラリ化が困難であるため、ツール類などの軽微なプログラムでの利用にとどめることを推奨する。
インタフェース定義は、udfmanager.h にある。

・UObject

UDFManagerを使用して構造型データを簡便に扱うためのUDFインタフェースを実現するパッケージ。
インタフェース定義は、UCognacin.h、IMuffin_solid.h など。
UDFファイルを解析してこのパッケージを自動生成するツール「makeinterface」を提供している。
このツールの利用方法については、第８ 章「インタフェースクラス自動生成ツール」を参照のこと。
簡便な入門用解説が付録A「makeinterface Tutorial」にまとめられている。

・Location

UDFManagerを使用してUDFシンボルを簡便に指定するためのユーティリティーパッケージ。
インタフェース定義は、udflocation.h にある。

・C APIs

C言語からUDFManagerインタフェースを利用するための関数群。
インタフェース定義は、fc_interfaceC.h にある。
関数リファレンスが第６ 章「C言語、Fortran言語によるUDFファイルの入出力」にまとめられている。また入門用サンプルコードの解説が別冊「libplatform Cインタフェース利用例」にまとめられている。

・Fortran APIs

Fortran 言語からUDFManagerインタフェースを利用するための関数群。

インタフェース定義は、fc_interfaceF.hである。

関数リファレンスが第6 章「C言語、Fortran言語によるUDFファイルの入出力」にまとめられている。また入門用サンプルコードの解説が別冊「libplatform Fortranインタフェース利用例」にまとめられている。

図 4.1 に、パッケージ関連図を示す。点線矢印は、パッケージの依存関係を示している。矢印の方向が依存方向となっている（include している）

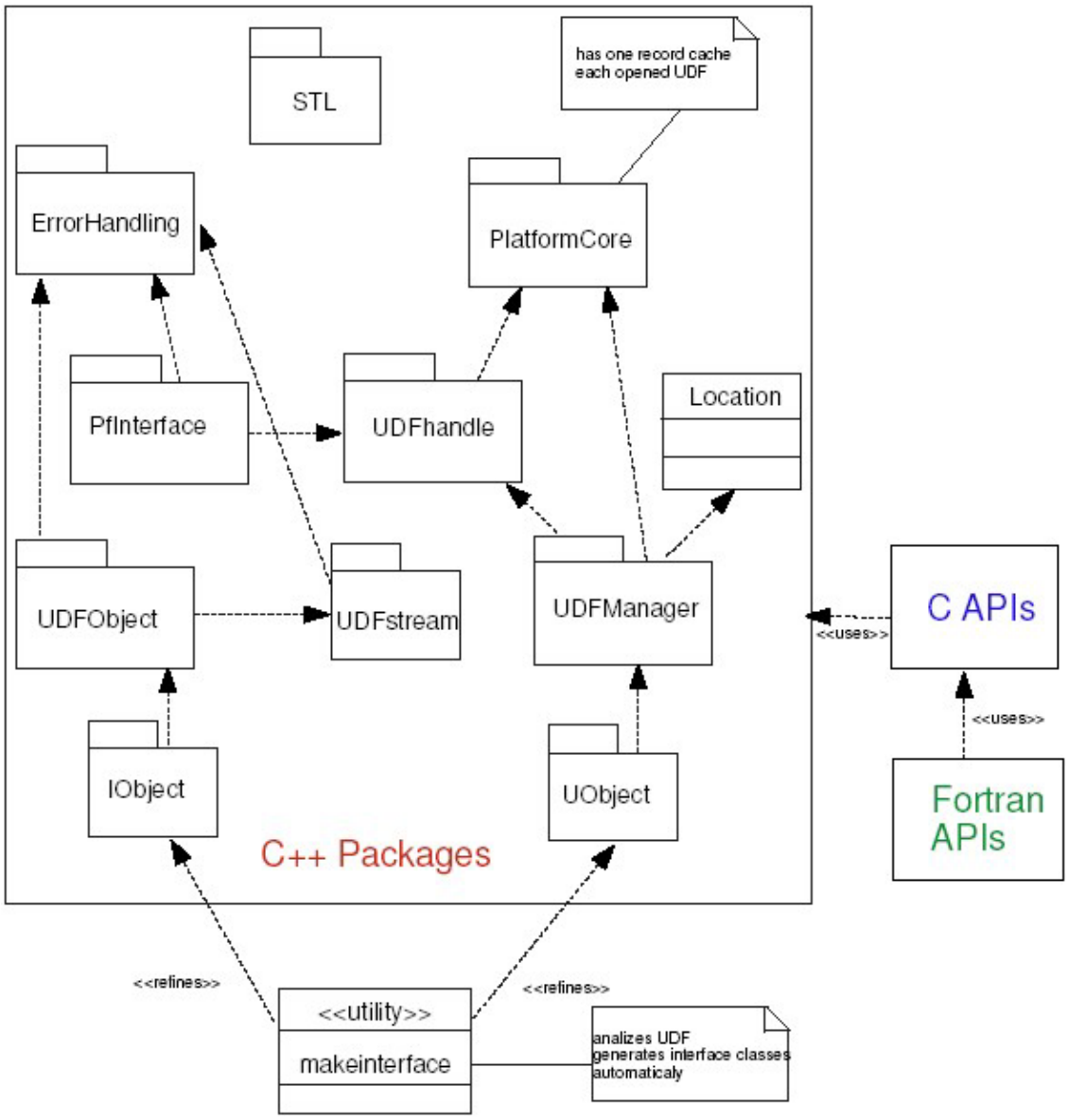


図 4.1：パッケージ間の関連

第５章　Ｃ＋＋パッケージインターフェース

以下に各パッケージのインタフェースを示す。

5.1　UDFhandle

エンジンがUDF データを読み書きするための基本インタフェース。通常PFinterface パッケージから使われるが、直接利用することも可能である。

インタフェース定義は、udfhandle.h である。

メソッドシグニチャ一覧は表 5.1，表 5.2，表 5.3 の通り。

表 5.1：UDFhandle メソッド一覧(1)

メソッドシグニチャ	機能
UDFhandle();	デフォルトコンストラクタ
UDFhandle(const string &udfname, const int recno = COMMON_RECORD);	コンストラクタ。インスタンスを構築し、指定したUDFファイル名、レコード番号で、open()を呼ぶ。
UDFhandle(const string &udfname, const string &recname);	コンストラクタ。インスタンスを構築し、指定したUDFファイル名、レコード名で、open()を呼ぶ。
~UDFhandle();	デストラクタ。参照回数が 0 ならば、UDFキャッシュを削除し、UDFhandleを未使用状態に戻す。
bool open(const string &udfname, const int recno = COMMON_RECORD);	指定したUDFファイル名、レコード番号でUDFキャッシュを構築する。
bool open(const string &udfname, const string &recname);	指定したUDFファイル名、レコードラベルでUDFキャッシュを構築する。
void close();	UDFキャッシュを削除し、UDFhandleを未使用状態に戻す。UDF出力用ハンドルの場合、UDFキャッシュ内容をUDFファイルへ出力する。

表 5.2：UDFhandle メソッド一覧(2)

メソッドシグニチャ	機能
int addReference();	参照回数をインクリメントする。
int removeReference();	参照回数をデクリメントする。
bool isOpen();	UDFキャッシュが構築されていれば、trueを返す。
bool inRecord();	レコードデータを出力中ならば、trueを返す。
void setRecordName(const string &recname);	レコードラベルを設定する。
int getCurrentRecord();	現在のレコード番号を返す。
int getTotalRecord();	現在の全レコード数を返す。
const char* newRecord(const char *prefix = NULL);	新規にレコードを作成しレコードラベルを返す。名称が指定されれば､#レコード番号を付加してレコードラベルを設定する。
void endRecord();	レコードデータ出力を終了する。
bool getIdList(const string &class_name, vector<int>& list) const;	指定したUDFクラス名のIDデータのリストを作成する。成功すればtrueを返す。
const char* getLocation(const string &class_name, int id);	指定したUDFクラス名、IDデータを持つUDFロケーション文字列を返す。
bool getKeyList(const string &class_name, vector<string>& list) const;	指定したUDFクラス名のKeyデータのリストを作成する。成功すればtrueを返す。
const char* getLocation(const string &class_name, const string &key);	指定したUDFクラス名､Keyデータを持つUDFロケーション文字列を返す。

表 5.3：UDFhandle メソッド一覧(3) ヘッダ情報関連

メソッドシグニチャ	機能
const string& getProjectName() const;	UDF ヘッダ情報の ProjectName を返す。
const string& getIOType() const;	UDF ヘッダ情報の IOType を返す。
const string& getEngineName() const;	UDF ヘッダ情報の EngineType を返す。
const string& getEngineVersion() const;	UDF ヘッダ情報の EngineVersion を返す。
const string& getComment() const;	UDF ヘッダ情報の Comment を返す。
const string& getAction() const;	UDF ヘッダ情報の Action FilePath を返す。
bool isCompatible (const string&version = ″″, const string&engine = ″″) const;	指定したversionが、UDFヘッダ情報EngineVersionと前方一致しなければfalseを返す。engineが指定されていれば､EngineVersionが同一でなければ､falseを返す。
void setProjectName(const string& text);	UDF ヘッダ情報の ProjectName を設定する。
void setIOType(const string& text);	UDF ヘッダ情報の IOType を設定する。
void setEngineName(const string& text);	UDF ヘッダ情報の EngineType を設定する。
void setEngineVersion(const string& text);	UDF ヘッダ情報の EngineVersion を設定する。
void setComment(const string& text);	UDF ヘッダ情報の Comment を設定する。
void setAction(const string& text);	UDF ヘッダ情報の Action FilePath を設定する。

5.2 Pfinterface

UDFhandleを隠蔽するためのインタフェース。Pfinterfaceパッケージは、C言語インタフェースから利用できるように、関数群となっており、UDFstreamと連携して、UDFデータの入出力を行う。

インタフェース定義は、pfinterface.h である。

メソッドシグニチャー覧は表 5.4 の通り。

5.3 UDFstream

UDFとエンジン間のデータ交換方式を、C++言語のストリームとして隠蔽するためのパッケージ。UDFhandleと連携して、UDFデータの入出力ストリームを実現する。

インタフェース定義は、udfstream.h にある。メソッドシグニチャ一覧は表 5.5 の通り。

UDFシンボルパターンとはUDF仕様書における「変数名」の集合を指定するもので、UDF内のデータ群を一意的な名前で指定できる文字列である。この文字列を使用して、UDF内の特定のデータ群をストリームとして読み書きすることを指定する。UDF変数が配列である場合、末尾の[]を指定しなければ、配列オブジェクトそのものを入力することを指定できる。

表 5.6 に、UDFシンボルパターンの例を示す。

5.4 ErrorHandling

例外処理を行うパッケージ。インタフェース定義は、pfexception.h である。

class ExceptionBase は、例外処理をカプセル化するための抽象クラスである。本リリースでは、以下の３種類の例外処理クラスを提供している。

class PFException; サーバモジュールのエラーをカプセル化する。
class UDFStreamException; UDFstream に関するエラーをカプセル化する。
class UDFObjectException; UDFObject に関するエラーをカプセル化する。

各クラスは、以下の２種類の診断メソッドを持っている。

・ ERROR_CODE WhatsWrong();
　内部エラーコードを返す。
・ virtual void Explain(DiagOutput & diag);
　エラー情報を、診断情報出力先に出力する。
　また、C 言語インタフェースで共通利用するために、以下の関数が定義されている。
・ const char *GetErrorMessage(ERROR_CODE code);
　エラーコードに対応する診断情報を返す。

class DiagOutput は、診断情報を出力するための抽象クラスである。本リリースでは、以下の３種類の診断情報出力クラスを提供している。

class DiagOutConsole; 診断情報を標準出力に出力する。
class DiagOutFile; 診断情報をログファイルに出力する。

各クラスは、出力先を考慮した以下のメッセージ出力メソッドを持っている。

・ virtual void DisplayMsg(const char *msg, DiagLevel level);

5.5 UDFObject

UDFのデータ構造をC++言語のクラスにマッピングするための基本インタフェースクラス定義のパッケージ。インタフェース定義は、udfobject.hである。

メソッドシグニチャ一覧は表 5.7～表 5.12 の通り。

5.6 IObject

エンジンのUDFインタフェースを実現するパッケージ。UDFObjectを継承して、個々のエンジンで用いるインタフェースオブジェクトを実装する。

インタフェース定義は、ICognacin.h、IMuffin_solid.h など"I"が先頭に付いたインタフェース名を用いる。インタフェース名は通常UDF名と同じにするが、ユーザが決めることもできる。

UDFファイルを解析してこのパッケージを自動生成するツール「makeinterface」を提供している。このツールの利用方法については、第 7 章「インタフェースクラス自動生成ツール」、付録 A「makeinterface チュートリアル」を参照のこと。

5.7 UDFManager

UDFObject、IObjectとは別の方式（レイトバインディング方式）でUDFインタフェースを実現するパッケージ。UDFManagerの利用者は、他のパッケージの内容を知っている必要は一切無い。簡便な入門用解説が別マニュアル「UDF/PF Tutorial」にまとめられている。

インタフェース定義は、udfmanager.h である。

メソッドシグニチャ一覧は表 5.13～表 5.21 の通り。

表 5.4：Pfinterface パッケージ関数一覧

メソッドシグニチャ	機能
UDFhandle* OpenUDF(const char *udfname, int record_no = COMMON_RECORD);	指定したUDFファイル、レコード番号をオープンし、UDFhandleを返す。
UDFhandle* CreateUDF(const char *udfname, const char *filename, bool def_copy = true);	指定したUDF定義ファイルの構造をもつUDFファイルを新規作成し、UDFhandleを返す。def_copyがfalseの場合、構造定義をinclude 指定するUDFファイルを新規作成する。
UDFhandle* CreateUDF(const char *udfname, const char **filenames, int nfiles, bool def_copy = true);	配列（filenames）で個数指定（nfiles）したUDF定義ファイルの構造をもつUDFファイルを新規作成し、UDFhandleを返す。def_copyがfalseの場合、構造定義をinclude指定するUDFファイルを新規作成する。
void CloseUDF(UDFhandle *fd);	指定したUDFhandleをクローズする。このUDFhandleが他で参照されている場合、何も行わない。
void AppendUDF(UDFhandle *fd, int record_no = COMMON_RECORD);	指定したUDFhandleにレコードを追加する。追加するレコード番号は、0 からシーケンシャルに増加する正整数でなければならない。
void SetStepName(UDFhandle *fd, const char *record_name);	指定したUDFhandleのレコードラベルを設定する。
size_t GetTotalStepNumber(UDFhandle *fd)	指定したUDFhandleの全レコード数を返す。
void SetPFControl(const char *statusFile)	GOURMETから起動されたエンジンには、エンジン制御ファイルパスがコマンドラインパラメータとして渡される。渡されたエンジン制御ファイルを設定する。この機能呼び出しの後、制御情報取得機能 PFControl() を使用することができる。一般的には解析エンジンのmain関数で使用する。
E_ENGINE_STATUS_TYPE PFControl(void)	エンジン制御情報を取得する。一般的には、解析エンジンの解析ループ内で使用し、取得した情報に従いエンジンが対応する処理を行うことになる。 返されるE_ENGINE_STATUSは、 e_ENGINE_RUN（実行）/ e_ENGINE_RESUME（再開）/ e_ENGINE_STOP（中断）/ e_ENGINE_ERROR（エンジン制御エラー） のいずれかである。e_ENGINE_SUSPEND（停止）は返されることはなく、この機能内で他の制御指定が行われるまでスリープする。

表 5.5：UDFstream メソッド一覧

メソッドシグニチャ	機能
UDFistream::UDFistream()	デフォルトコンストラクタ
UDFistream::UDFistream(UDFhandle* fd, const char *udfsymbol=NULL)	コンストラクタ。指定したUDFhandleの入力ストリームを構築する。UDFシンボルパターンが指定されていれば、同時に入力ストリームをオープンする。
UDFistream::~UDFistream()	デストラクタ
int UDFistream::open(const char *udfsymbol)	指定したUDFシンボルパターンの入力ストリームをオープンする。
void UDFistream::close()	入力ストリームをクローズする。UDFストリームは再利用可能である。すなわち、同じUDFistreamに対し、異なるUDFシンボルパターンを指定してオープンできる。
UDFostream::UDFostream()	デフォルトコンストラクタ
UDFostream::UDFostream(UDFhandle* fd, const char *udfsymbol=NULL)	コンストラクタ。指定したUDFhandleの出力ストリームを構築する。UDFシンボルが指定されていれば、同時に出力ストリームをオープンする。指定するUDFシンボルはトップオブジェクトでなければならない。
UDFostream::~UDFostream()	デストラクタ
int UDFostream::open(const char *udfsymbol= NULL)	指定したUDFシンボルの出力ストリームをオープンする。指定するUDFシンボルはトップオブジェクトでなければならない。
void UDFostream::close()	出力ストリームをクローズする。UDFストリームは再利用可能である。すなわち、同じUDFostreamに対し、異なるUDFシンボルパターンを指定してオープンできる。

表 5.6：UDF シンボルパターンの例

UDFシンボルパターン	指定されたUDF データストリーム	データ数
長さ 0 の文字列	UDFレコード内の全オブジェクト	トップオブジェクトの個数
molecules	molecules オブジェクト全体	1
molecules.bond[]	molecules.bond[] の全ての要素	要素数
molecules.bond	molecules.bond[] の全ての要素を持つ配列	1
molecules.bond[].atom1.posx	molecules.bond[] の全ての要素のatom1.posx	要素数

表 5.7：UDFObject メソッド一覧(1)

メソッドシグニチャ	機能
UDFObject()	デフォルトコンストラクタ
UDFObject(const UDFObject& other)	コピーコンストラクタ
~UDFObject()	デストラクタ
virtual const char *GetName()	UDF変数名取得。継承クラスでオバーライドする。
virtual void _Input(istream& is)	ストリーム入力スタブ。継承クラスでオバーライドする。
virtual void _Output(ostream& os)	ストリーム出力スタブ。継承クラスでオバーライドする。

表 5.8：UDFObject メソッド一覧(2) UDF 基本型

メソッドシグニチャ	機能
UDFshort::UDFshort	C++基本型short にバインド
UDFint::UDFint	C++基本型int にバインド
UDFlong::UDFlong	C++基本型long にバインド
UDFfloat::UDFfloat	C++基本型float にバインド
UDFsingle::UDFsingle	C++基本型float にバインド
UDFdouble::UDFdouble	C++基本型double にバインド
UDFstring::UDFstring	UDFstring 型
string UDFstring::GetValue()	値取得
operator UDFstring::string()	C++string クラスへの型変換演算子
void UDFstring::SetValue(string val)	値設定
template <class T> class UDFarray::UDFarray()	UDFarray 型テンプレート
size_t UDFarray::size()	STL::<list> 同名メソッド
size_t max_ UDFarray::size()	STL::<list> 同名メソッド
void UDFarray::push_back(T & x)	STL::<list> 同名メソッド
void UDFarray::clear()	STL::<list> 同名メソッド
void UDFarray::pop_back()	STL::<list> 同名メソッド
bool UDFarray::empty()	STL::<list> 同名メソッド
void UDFarray::reserve(size_t n)	STL::<vector> 同名メソッド
size_t UDFarray::capacity()	STL::<vector> 同名メソッド
T & UDFarray::operator[](size_t pos)	STL::<vector> 同名メソッド
const T & UDFarray::operator[](size_t pos)	STL::<vector> 同名メソッド
vector<T>::iterator UDFarray::begin()	STL::<vector> 同名メソッド
vector<T>::iterator UDFarray::end()	STL::<vector> 同名メソッド
vector<T>::iterator UDFarray::rbegin()	STL::<vector> 同名メソッド
vector<T>::iterator UDFarray::rend()	STL::<vector> 同名メソッド

表 5.9：UDFObject メソッド一覧(2)

メソッドシグニチャ	機能
void UDFarray::resize(size_t n, T x = T())	STL::<vector> 同名メソッド
vector<T>::iterator UDFarray::erase(vector<T>::iterator it)	STL::<vector> 同名メソッド
vector<T>::iterator UDFarray::erase(vector<T>::iterator first, vector<T>::iterator last)	STL::<vector> 同名メソッド
void UDFarray::assign(vector<T>:: const_iterator first, vector<T>::const_iterator last)	STL::<vector> 同名メソッド
void UDFarray::assign(size_t n, const T& x = T())	STL::<vector> 同名メソッド
vector<T>::iterator UDFarray::insert(vector<T>::iterator it, const T& x = T())	STL::<vector> 同名メソッド
void UDFarray::insert(vector<T>::iterator it, size_t n, const T& x)	STL::<vector> 同名メソッド
void UDFarray::insert(vector<T>::iterator it, vector<T>::const_iterator first, vector<T>::const_iterator last)	STL::<vector> 同名メソッド

表 5.10：UDFObject メソッド一覧(3) 比較演算子

メソッドシグニチャ	機能
bool operator==(const UDFObject& x, const UDFObject& y)	等値判定演算子
bool operator<(const UDFObject& x, const UDFObject& y)	大小判定演算子

表 5.11：UDFObject メソッド一覧(4) 入出力オペレータ

メソッドシグニチャ	機能
istream& operator >> (istream& is, UDFObject& object)	テキスト入力ストリーム演算子
ostream& operator << (ostream& os, UDFObject& object)	テキスト出力ストリーム演算子

表 5.12：UDFObject メソッド一覧(5) 例外処理クラス

メソッドシグニチャ	機能
UDFObjectException(int err, const char *info = NULL)	コンストラクタ。指定エラー番号、追加情報のUDFObjectExceptionを生成する。
void UDFObjectException::Explain(DiagOutput & diag);	DiagOutputにメッセージを表示する。
const char * UDFObjectException::ErrorMessage(int err);	指定エラー番号のメッセージ文字列を返す。

表 5.13：UDFManager メソッド一覧(1)

メソッドシグニチャ	機能
UDFManager(const string &udfname, int mode =READ)	コンストラクタ。インスタンスを構築し、指定したUDFファイルをオープンする。modeでREADまたはWRITEを指定する。
UDFManager(const string &udfname, const string &deffilename, bool def_copy = true)	コンストラクタ。インスタンスを構築し、指定したUDF定義ファイルを使用してUDFファイルを新規作成する。def_copyがfalseの場合、定義ファイルをインクルードする指定となる。
~UDFManager()	デストラクタ

表 5.14：UDFManager メソッド一覧(2) ヘッダ情報関連

メソッドシグニチャ	機能
const string& getProjectName() const;	UDFヘッダ情報のProjectNameを返す。
const string& getIOType() const;	UDFヘッダ情報のIOTypeを返す。
const string& getEngineName() const;	UDFヘッダ情報のEngineTypeを返す。
const string& getEngineVersion() const;	UDFヘッダ情報のEngineVersionを返す。
const string& getComment() const;	UDFヘッダ情報のCommentを返す。
const string& getAction() const;	UDFヘッダ情報のAction FilePathを返す。
bool isCompatible (const string& version = "", const string& engine ="") const;	指定したversion が、UDFヘッダ情報EngineVersionと前方一致しなければfalseを返す。engineが指定されていれば、EngineVersionが同一でなければ、falseを返す。
void setProjectName(const string& text);	UDFヘッダ情報のProjectNameを設定する。
void setIOType(const string& text);	UDFヘッダ情報のIOTypeを設定する。
void setEngineName(const string& text);	UDFヘッダ情報のEngineTypeを設定する。
void setEngineVersion(const string& text);	UDFヘッダ情報のEngineVersionを設定する。
void setComment(const string& text);	UDFヘッダ情報のCommentを設定する。
void setAction(const string& text);	UDFヘッダ情報のAction FilePathを設定する。

表 5.15：UDFManager メソッド一覧(3) レコード関連

メソッドシグニチャ	機能
size_t totalRecord()	レコード数を取得する。
bool jump(size_t record_no)	指定レコードへ移動する。
bool jump(const string &record_name)	指定レコードラベルへ移動する。
bool nextRecord()	次レコードへ移動する。
const string newRecord(const string prefix)	新規にレコードを作成しレコードラベルを返す。 名称が指定されていれば､#レコード番号を付加してレコードラベルを設定する。
bool write()	全レコードデータをオープンしているUDFファイルへSaveする。
bool write(const string &udfname)	全レコードデータを指定したUDFファイルへSaveAsする。

表 5.16：UDFManager メソッド一覧(4) シンボル関連

メソッドシグニチャ	機能
bool seek(const string &location)	現在位置を指定シンボル位置へ移動する。
bool seek(const Location &location)	現在位置を指定ロケーションへ移動する。
const string tell()	現在シンボル位置を取得する。
string type()	現在位置のシンボルタイプを取得する。
bool next()	次データへ移動する。
size_t size()	現在位置のシンボルの配列要素数を取得する。
size_t size(const string &location)	現在位置のシンボルの配列要素数を取得する。
size_t size(const string &location, const INDEX & index=INDEX())	指定シンボル・INDEX の配列要素数を取得する。

表 5.17：UDFManager メソッド一覧(5) データ取得関連

メソッドシグニチャ	機能
bool get(const Location &location, UDF_TYPE &value)	指定ロケーションのデータを取得する。
bool get(const string &location, UDF_TYPE &value, const INDEX & index=INDEX())	指定シンボル・INDEX のデータを取得する。
bool getArray(const Location &location, vector<UDF_TYPE > &array)	指定ロケーションのデータ配列を取得する。
bool getArray(const string &location, vector<UDF_TYPE> &array, const INDEX& index=INDEX())	指定シンボル・INDEX のデータ配列を取得する。
bool get(const string &location, UDF_TYPE *value, const INDEX& index=INDEX())	指定シンボル・INDEX のデータを取得する。

表 5.18：UDFManager メソッド一覧(6) 値取得簡易版

メソッドシグニチャ	機能
short shortValue(const string &location)	指定シンボルのデータを取得する。
int intValue(const string &location)	指定シンボルのデータを取得する。
long longValue(const string &location)	指定シンボルのデータを取得する。
float floatValue(const string &location)	指定シンボルのデータを取得する。
double doubleValue(const string &location)	指定シンボルのデータを取得する。
string stringValue (const string &location)	指定シンボルのデータを取得する。
vector<short> & shortArray(const string &location)	指定シンボルのデータ配列を取得する。
vector<int> & intArray(const string &location)	指定シンボルのデータ配列を取得する。
vector<long> & longArray(const string &location)	指定シンボルのデータ配列を取得する。
vector<float> & floatArray(const string &location)	指定シンボルのデータ配列を取得する。
vector<double> & doubleArray(const string &location)	指定シンボルのデータ配列を取得する。
vector<string> & stringArray(const string &location)	指定シンボルのデータ配列を取得する。
int i(const string &location)	intValue のエイリアス
double d(const string &location)	doubleValue のエイリアス
string s(const string &location)	stringValue のエイリアス
vector<int> & iarray(const string &location)	intArray のエイリアス
vector<double> & darray(const string &location)	doubleArray のエイリアス
vector<string> & sarray(const string &location)	stringArray のエイリアス

表 5.19：UDFManager メソッド一覧(7) データ設定関連

メソッドシグニチャ	機能
bool put(const Location &location, UDF_TYPE &value)	指定ロケーションのデータを設定する。
bool put(const string &location, UDF_TYPE &value, const INDEX &index=INDEX())	指定シンボル・INDEX のデータを設定する。
bool putArray(const Location &location, vector<UDF_TYPE> &array)	指定ロケーションのデータ配列を設定する。
bool putArray(const string &location, vector<UDF_TYPE> &array, const INDEX & index=INDEX())	指定シンボル・INDEX のデータ配列を設定する。

表 5.20：UDFManager メソッド一覧(8) キー・インデックス関連

メソッドシグニチャ	機能
vector<int> getIdList(const string &class_name)	指定したUDFクラス名のIDデータのリストを取得する。
const Location &getLocation(const string &class_name, int id)	指定したUDFクラス名、IDデータを持つシンボルのロケーションを取得する。
vector<string>& getKeyList(const string &class_name)	指定したUDFクラス名のKeyデータのリストを取得する。
const Location &getLocation(const string &class_name, const string &key)	指定したUDFクラス名Keyデータを持つシンボルのロケーションを取得する。

表 5.21：UDFManager メソッド一覧(9) 例外処理関連

メソッドシグニチャ	機能
UDFManager::UDFManagerException	UDFManager内での何らかのエラー
UDFManager::ObjectNotFoundException	該当データが見つからない
UDFManager::SymbolNotFoundException	指定シンボルが見つからない
UDFManager::SymbolNotDeterminedException	指定シンボルを決定できない
UDFManager::KeyNotFoundException	キーが見つからない
UDFManager::IndexNotFoundException	インデックスが見つからない
UDFManager::ParseException	UDFパースエラー
UDFManager::FileIOException	指定UDFファイルのIOエラー（オープンエラーも含む)
UDFManager::SystemException	ライブラリ内の未定義エラー
UDFManager::NotImplementedException	未実装機能を呼び出した

特に説明がない場合、bool を返すメソッドは、操作が成功すればtrue、失敗すればfalseを返す。

UDF_TYPEは、short, int, long, float, double, string のいずれかを示す。

「シンボル」の表記は、UDF変数文字列を表し、表 5.22 のようにUDF変数を厳密に指定する文字列を使用できる。配列を表す場合、"[]"が必要であることに注意。INDEXは繰り返し処理の場合に配列の添字を指定する為のヘルパークラスであり、５次元までのコンストラクタが準備されている。

表 5.22：UDFManager シンボル指定例

シンボル	INDEX	内容
molecules.bond[]	無指定	molecules.bond 配列
molecules.bond[0]	無指定	molecules.bond 配列の第 1 要素
molecules.bond[]	INDEX(0)	同上
molecules.bond[1].atom[0].posx	無指定	molecules.bond 配列の第 2 要素のatom 配列の第 1 要素の.posx
molecules.bond[].atom[].posx	INDEX(1,0)	同上

「ロケーション」は、上記のUDF配列文字列とINDEXを実装するヘルパークラスLocationのインスタンスであり、単独でUDF変数を厳密に指定することができる。Locationクラスのメソッドについては、５．９ Location を参照のこと。

5.8 UObject

UDFManagerを使用して構造型データを簡便に扱うためのUDFインタフェースを実現するパッケージ。

インタフェース定義は、UCognacin.h、UMuffin_solid.h など"U"が先頭に付いたインタフェース名を用いる。インタフェース名は通常UDF名と同じにするが、ユーザが決めることもできる。

UDFファイルを解析してこのパッケージを自動生成するツール「makeinterface」を提供している。このツールの利用方法については、第 7 章「インタフェースクラス自動生成ツール」を参照のこと。簡便な入門用解説が第 11 章「makeinterface チュートリアル」にまとめられている。

5.9 Location

UDFManagerを使用してUDFシンボルを簡便に指定するためのユーティリティーパッケージ。インタフェース定義は、udflocation.h である。

INDEX クラス：配列データアクセスのための添字情報のコンストラクタ

INDEX のコンストラクター覧を、表 5.23 に示す。

Location のメソッドシグニチャ一覧を、表 5.24、表 5.25 に示す。

表 5.23：INDEX コンストラクタ一覧

メソッド名	機能
INDEX()	デフォルトコンストラクタ
INDEX(const UINT idx1)	1 次元コンストラクタ
INDEX(const UINT idx1, const UINT idx2)	2 次元コンストラクタ
INDEX(const UINT idx1, const UINT idx2, const UINT idx3)	3 次元コンストラクタ
INDEX(const UINT idx1, const UINT idx2, const UINT idx3, const UINT idx4)	4 次元コンストラクタ
INDEX(const UINT idx1, const UINT idx2, const UINT idx3, const UINT idx4, const UINT idx5)	5 次元コンストラクタ

表 5.24：Location メソッド一覧

メソッドシグニチャ	機能
Location()	デフォルトコンストラクタ
Location(const Location & loc)	コピーコンストラクタ
Location(const string &path)	コンストラクタ。添字の入ったパス文字列を解析する
Location(const char *path)	コンストラクタ。添字の入ったパス文字列を解析する
Location(const string &path, const INDEX &index)	コンストラクタ。パス文字列と指定インデックスを解析する
const string & str()	添字の入ったパス文字列を返す
const Location & root()	トップロケーションに移動する
const Location & seek(const string &path)	指定ロケーションに移動する
const Location & up()	上の階層に移動する
const Location & down(const string &element)	下の指定属性に移動する
const Location sub(const string &element) const	下の指定属性のロケーションを返す。移動はしない。
Location array()	現在の配列ロケーションを返す
const Location & prev()	配列の 1 つ前の要素に移動する
const Location & next()	配列の 1 つ次の要素に移動する
Location array() const	現在の配列位置に移動する
const string & getPath()	シンボルパスを返す
const INDEX & getIndex()	インデックスを返す

表 5.25：Location メソッド一覧(2) 例外処理関連

メソッドシグニチャ	機能
Location::LocationException	Locationの使用エラー

第６章　C 言語、Fortran 言語による UDF ファイルの入出力

プラットフォームライブラリにはC言語およびFortran言語(FORTRAN77,Fortran90,Fortran95)からUDFファイルの入出力を行う関数群が含まれている。これらの関数群は以下のような機能を持つ。

・ UDFファイルを入出力用に開いたり閉じたりする。

・ 指定したUDFデータパスの単一データの読み書き。

・ 指定したUDFデータパスの配列データの読み書き。

配列データの読み書きは多次元配列データに対応している。

6.1　C言語によるUDFファイルの入出力関数

プラットフォームライブラリlibplatform にはC言語プログラムから呼び出す事のできる関数群が含まれている。表 6.1 にそれらの関数の仕様を示す。

表 6.1：C 言語による UDF 関数

---

int openUDFManager(const char *filename, int *udf_handle);

ファイル名が文字列filenameで表されるUDFファイルを読み込み専用として開く。udf_handleはこの関数の呼出し以降のC言語ソースでここで開かれたUDFファイルを指す整数値である。関数戻り値はUDFファイルを開くのに成功すれば 0、失敗すれば-1 となる。

int createUDFManager(const char *filename, const char *def_filename, int def_copy, int *udf_handle);

UDF定義ファイル名が文字列def_filenameで表されるUDFデータファイルfilenameを書き込みモードで開く。def_copyはUDF定義ファイルの内容を出力UDFファイルにコピーする場合に 1、UDF定義ファイル名でincludeする場合には 0 を与える。udf_handleはこの関数の呼出し以降のC言語ソースでここで開かれたUDFファイルを指す整数値である。関数戻り値はUDFデータを開くのに成功すれば 0、失敗すれば-1 となる。

closeUDFManager(const int udf_handle);

udf_handleで指定されていたUDFファイルをクローズする。この呼出し以降のソースでudf_handleによるUDF入出力はできない。戻り値はUDFファイルのクローズに成功すれば 0、失敗すれば-1 となる。

int totalRecord(const int udf_handle, int *result);

udf_handleで指定されるUDFファイル中の総データレコード数をresultに返す。関数戻り値はレコー

ド数の取得に成功すれば 0、失敗すれば-1 となる。

int jumpRecord(const int udf_handle, int record_no);

udf_handleで指定されるUDFファイルのレコード番号record_noの位置に移動する。関数戻り値はレコードの移動に成功すれば 0、失敗すれば-1 となる。

int newRecord(const int udf_handle, const int limit, char *label);

udf_handleで指定されるUDFファイルの末尾に新しいデータレコードを一つ追加する。新しいレコードのラベル文字が文字列labelとして返される。limitは文字列変数labelの文字数を表す。関数戻り値はレコード数の生成に成功すれば 0、文字列生成に失敗するかもしくは生成されたラベルの文字数がlimitより大きい場合 -1 となる。

int getIntValue(const int udf_handle, const char *target, int *result);

udf_handleで指定されるUDFファイルのデータパスtargetの位置から整数値resultを読み込む。関数戻り値はデータ取得に成功すれば 0、失敗すれば-1 となる。

int getFloatValue(const int udf_handle, const char *target, float *result);

udf_handleで指定されるUDFファイルのデータパスtargetの位置からfloat値 resultを読み込む。関数戻り値はデータ取得に成功すれば 0、失敗すれば-1 となる。

int getDoubleValue(const int udf_handle, const char *target, double *result);

udf_handleで指定されるUDFファイルのデータパスtargetの位置からdouble値 resultを読み込む。関数戻り値はデータ取得に成功すれば 0、失敗すれば-1 となる。

int getStringValue(const int udf_handle, const char *target,const int limit,char *result);

udf_handleで指定されるUDFファイルのデータパスtargetの位置から文字列値resultを読み込む。limitは文字列変数resultの文字数を表す。関数戻り値はデータ取得に成功すれば 0、データ取得に失敗するかもしくは取得する文字列長がlimitを越える場合-1 となる。

int getIntValueInArray(const int udf_handle, const char *target, int *result);

udf_handle で指定されるUDFファイルのデータパスtarget が配列内の位置を示している時にそこから一個の整数値resultを読み込む。関数戻り値はデータ取得に成功すれば 0、失敗すれば-1 となる。

int getFloatValueInArray(const int udf_handle, const char *target, float *result);

udf_handleで指定されるUDFファイルのデータパスtarget が配列内の位置を示している時にそこから一個のfloat値 resultを読み込む。関数戻り値はデータ取得に成功すれば 0、失敗すれば-1 となる。

int getDoubleValueInArray(const int udf_handle, const char *target, double *result);

udf_handleで指定されるUDFファイルのデータパスtargetが配列内の位置を示している時にそこから一個のdouble値 resultを読み込む。関数戻り値はデータ取得に成功すれば 0、失敗すれば-1 となる。

int getStringValueInArray(const int udf_handle, const char *target, const int limit, char *result);

udf_handleで指定されるUDFファイルのデータパスtargetが配列内の位置を示している時にそこから一個の文字列値resultを読み込む。limitは文字列変数resultの文字数上限(末尾ヌル文字を含む)を表す。関数戻り値はデータ取得に成功すれば0、データ取得に失敗するかもしくは取得する文字列長がlimitを越える場合-1 となる。

int getIntArray(const int udf_handle, const char *target, const int capacity, int *num_dim, int size_list[], int *result);

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[]" を含む)にそこから整数値配列resultを読み込む。capacityは配列resultに格納できるデータ数の上限。num_dim、size_listはそれぞれ配列の次元数と各次元のサイズで関数から返されるものである。関数戻り値はデータ取得に成功すれば0、データ取得に失敗したり配列サイズ上限capacityを越えるデータがあれば-1 となる。

int getFloatArray(const int udf_handle, const char *target, const int capacity, int *num_dim, int size_list[], float *result);

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）にそこからfloat値配列 result を読み込む。capacityは配列resultに格納できるデータ数の上限。num_dim、size_list はそれぞれ配列の次元数と各次元のサイズで関数から返されるものである。関数戻り値はデータ取得に成功すれば0、データ取得に失敗したり配列サイズ上限capacityを越えるデータがあれば-1 となる。

int getDoubleArray(const int udf_handle, const char *target, const int capacity, int *num_dim, int size_list[], double *result);

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）にそこからdouble値配列 resultを読み込む。capacityは配列resultに格納できるデータ数の上限。num_dim、size_list はそれぞれ配列の次元数と各次元のサイズで関数から返されるものである。関数戻り値はデータ取得に成功すれば0、データ取得に失敗したり配列サイズ上限capacityを越えるデータがあれば-1 となる。

int getStringArray(const int udf_handle, const char *target, const int capacity, const int limit, int *num_dim, int size_list[], char *result);

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）にそこから文字列値配列 resultを読み込む。capacityは配列resultに格納できるデータ数の上限、limitは各配列要素の文字数の上限である。num_dim、size_listはそれぞれ配列の次元数と各次元のサイズで関数から返されるものである。関数戻り値はデータ取得に成功すれば0、データ取得に失敗したり配列サイズ上限capacityを越えるデータや文字数がlimitを越える文字列があれば-1 となる。

int setIntValue(const int udf_handle, const char *target, int *source);

udf_handleで指定されるUDFデータのデータパスtargetの位置に整数値sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

int setFloatValue(const int udf_handle, const char *target, float *source);

udf_handleで指定されるUDFファイルのデータパスtargetの位置にfloat値 sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

int setDoubleValue(const int udf_handle, const char *target, double *source);

udf_handleで指定されるUDFデータのデータパスtargetの位置にdouble値 sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

int setStringValue(const int udf_handle, const char *target, const int limit, char *source);

udf_handleで指定されるUDFデータのデータパスtargetの位置に文字列値 sourceを与える。limitはここでは利用されていない。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

int setIntValueInArray(const int udf_handle, const char *target, int *source);

udf_handleで指定されるUDFデータのデータパスtargetが配列内の位置を示している時にそこに一個の整数値 sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

int setFloatValueInArray(const int udf_handle, const char *target, float *source);

udf_handleで指定されるUDFデータのデータパスtargetが配列内の位置を示している時にそこに一個のfloat値 sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

int setDoubleValueInArray(const int udf_handle, const char *target, double *source);

udf_handleで指定されるUDFファイルのデータパスtargetが配列内の位置を示している時にそこに一個のdouble値 sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

int setStringValueInArray(const int udf_handle, const char *target, const int limit, char *source);

udf_handleで指定されるUDFデータのデータパスtargetが配列内の位置を示している時にそこに一個の文字列値 sourceを与える。limitはここでは利用されていない。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

int setIntArray(const int udf_handle, const char *target, const int size_list[], int *source);

udf_handleで指定されるUDFデータのデータパスtargetが配列を示している時（文字列"[]"を含む）にそこに整数値配列sourceを書き込む。size_listは配列の各次元のサイズである（次元数はtargetで決まる）。関数戻り値はデータのセットに成功すれば0、データのセットに失敗した場合 -1 となる。

int setFloatArray(const int udf_handle, const char *target, const int size_list[], float *source);

udf_handleで指定されるUDFデータのデータパスtargetが配列を示している時（文字列"[]"を含む）にそこにfloat値配列 sourceを書き込む。size_listは配列の各次元のサイズである（次元数はtarget で決まる）。関数戻り値はデータのセットに成功すれば0、データのセットに失敗した場合 -1 となる。

int setDoubleArray(const int udf_handle, const char *target, const int size_list[], double *source);

udf_handleで指定されるUDFデータのデータパスtargetが配列を示している時（文字列"[]"を含む）にそこにdouble値配列 sourceを書き込む。size_listは配列の各次元のサイズである（次元数はtargetで決まる）。関数戻り値はデータのセットに成功すれば0、データのセットに失敗した場合-1となる。

int setStringArray(const int udf_handle, const char *target, const int size_list[], const int limit, char *source);

udf_handleで指定されるUDFデータのデータパスtargetが配列を示している時（文字列"[]"を含む）にそこに文字列値配列sourceを書き込む。size_listは配列の各次元のサイズである（次元数はtargetで決まる）。limitはsourceの各要素の文字列長である。関数戻り値はデータのセットに成功すれば0、データのセットに失敗した場合-1 となる。

int writeUDFData(const int udf_handle);

udf_handleで指定されるUDFデータをファイルに書き出す。関数戻り値は書き出しに成功すれば0、失敗した場合-1 となる。

int getSize(const int udf_handle, const char *arget, int *result);

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）配列データの大きさをresultに得る。多次元配列の場合には対象となる次元のサイズの積が得られる。

int getSizeList(const int udf_handle, const char *target, int *num_dim, int size_list[]);

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）配列データの次元数をnum_dimに、各配列次元のサイズを配列size_listにそれぞれ返す。

int getDimensions(const int udf_handle, const char *target, int *num_dim);

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）配列データの次元数をnum_dimに返す。

6.2 FortranによるUDFファイルの入出力関数

プラットフォームライブラリlibplatform にはFortranプログラムから呼び出す事のできる関数群が含まれている。表 6.2 にそれらの関数の仕様を示す。

表 6.2：Fortran による UDF 関数

integer openUDFManager(character*(*) filename, integer udf_handle)

ファイル名が文字列filenameで表されるUDFファイルを読み込み専用として開く。udf_handleはこの関数の呼出し以降のFortranソースでここで開かれたUDFファイルを指す整数値である。関数戻り値はUDFファイルを開くのに成功すれば 0、失敗すれば-1 となる。

integer createUDFManager(character*(*) filename, character*(*) def_filename,
integer def_copy, integer udf_handle);

UDF定義ファイル名が文字列def_filenameで表されるUDFデータファイルfilenameを書き込みモードで開く。def_copyはUDF定義ファイルの内容を出力UDFファイルにコピーする場合に 1、UDF定義ファイル名でincludeする場合には 0 を与える。udf_handleはこの関数の呼出し以降のC言語ソースでここで開かれたUDFファイルを指す整数値である。関数戻り値はUDFデータを開くのに成功すれば 0、失敗すれば-1 となる。

integer closeUDFManager(integer udf_handle)

udf_handleで指定されていたUDFファイルをクローズする。この呼出し以降のソースでudf_handleによるUDF入出力はできない。戻り値はUDFファイルのクローズに成功すれば 0、失敗すれば-1 となる。

integer totalRecord(integer udf_handle, integer result)

udf_handleで指定されるUDFファイル中の総データレコード数をresult に返す。関数戻り値はレコード数の取得に成功すれば 0、失敗すれば-1 となる。

integer jumpRecord(integer udf_handle, integer record_no)

udf_handleで指定されるUDFファイルのレコード番号record_noの位置に移動する。関数戻り値はレコードの移動に成功すれば 0、失敗すれば-1 となる。

integer newRecord(integer udf_handle, integer limit, character*(*) label)

udf_handleで指定されるUDFファイルの末尾に新しいデータレコードを一つ追加する。新しいレコードのラベル文字が文字列labelとして返される。limitは文字列変数labelの文字数を表す。関数戻り値はレコード数の生成に成功すれば 0、文字列生成に失敗するかもしくは生成されたラベルの文字数がlimitより大きい場合-1 となる。

integer getIntValue(integer udf_handle, character*(*) target, integer result)

udf_handleで指定されるUDFファイルのデータパスtargetの位置から整数値resultを読み込む。関数戻り値はデータ取得に成功すれば0、失敗すれば-1となる。

integer getFloatValue(integer udf_handle,character*(*) target, real result)

udf_handleで指定されるUDFファイルのデータパスtargetの位置からreal値 resultを読み込む。関数戻り値はデータ取得に成功すれば0、失敗すれば-1となる。

integer getDoubleValue(integer udf_handle, character*(*) target, double precision result)

udf_handleで指定されるUDFファイルのデータパスtargetの位置からdouble precision値 resultを読み込む。関数戻り値はデータ取得に成功すれば0、失敗すれば-1となる。

integer getStringValue(integer udf_handle, character*(*) target,integer limit, character*(*) result)

udf_handleで指定されるUDFファイルのデータパスtargetの位置から文字列値resultを読み込む。limitは文字列変数resultの文字数を表す。関数戻り値はデータ取得に成功すれば0、データ取得に失敗するかもしくは取得する文字列長がlimitを越える場合-1となる。

integer getIntValueInArray(integer udf_handle, character*(*) target, integer result)

udf_handleで指定されるUDFファイルのデータパスtargetが配列内の位置を示している時にそこから一個の整数値resultを読み込む。関数戻り値はデータ取得に成功すれば0、失敗すれば-1となる。

integer getFloatValueInArray(integer udf_handle, character*(*) target, real result)

udf_handleで指定されるUDFファイルのデータパスtarget が配列内の位置を示している時にそこから一個のreal値 resultを読み込む。関数戻り値はデータ取得に成功すれば0、失敗すれば-1となる。

integer getDoubleValueInArray(integer udf_handle, character*(*) target, double precision result)

udf_handleで指定されるUDFファイルのデータパスtargetが配列内の位置を示している時にそこから一個のdouble precision値 resultを読み込む。関数戻り値はデータ取得に成功すれば0、失敗すれば-1となる。

integer getStringValueInArray(integer udf_handle, character*(*) target, integer limit, character*(*) result)

udf_handleで指定されるUDFファイルのデータパスtargetが配列内の位置を示している時にそこから一個の文字列値resultを読み込む。limitは文字列変数 resultの文字数上限を表す。関数戻り値はデータ取得に成功すれば0、データ取得に失敗するかもしくは取得する文字列長がlimitを越える場合-1となる。

integer getIntArray(integer udf_handle, character*(*) target, integer capacity, integer num_dim, integer size_list(*), integer result(1:capacity))

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）にそこから整数値配列resultを読み込む。capacityは配列resultに格納できるデータ数の上限。num_dim、size_listはそれぞれ配列の次元数と各次元のサイズで関数から返されるものである。関数戻り値はデータ取得に成功すれば0、データ取得に失敗したり配列サイズ上限capacityを越えるデータがあれば-1となる。

integer getFloatArray(integer udf_handle, character*(*) target, integer capacity,
integer num_dim, integer size_list(*), real result(1:capacity))

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）にそこからfloat値配列 resultを読み込む。capacityは配列resultに格納できるデータ数の上限。num_dim、size_listはそれぞれ配列の次元数と各次元のサイズで関数から返されるものである。関数戻り値はデータ取得に成功すれば0、データ取得に失敗したり配列サイズ上限capacityを越えるデータがあれば-1 となる。

integer getDoubleArray(integer udf_handle, character*(*) target, integer capacity,
integer num_dim, integer size_list(*), double precision result(1:capacity))

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）にそこからdouble precision値配列 resultを読み込む。capacityは配列resultに格納できるデータ数の上限。num_dim、size_listはそれぞれ配列の次元数と各次元のサイズで関数から返されるものである。関数戻り値はデータ取得に成功すれば0、データ取得に失敗したり配列サイズ上限capacityを越えるデータがあれば-1 となる。

integer getStringArray(integer udf_handle, character*(*) target, integer capacity,integer limit,
integer num_dim, integer size_list(*), character□(limit) result(1:capacity))

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）にそこからdouble値配列 resultを読み込む。capacityは配列resultに格納できるデータ数の上限、limitは各配列要素の文字数の上限である。num_dim、size_list はそれぞれ配列の次元数と各次元のサイズで関数から返されるものである。関数戻り値はデータ取得に成功すれば0、データ取得に失敗したり配列サイズ上限capacityを越えるデータや文字数がlimitを越える文字列があれば-1 となる。

integer setIntValue(intger udf_handle, character*(*) target, integer source)

udf_handleで指定されるUDFデータのデータパスtargetの位置に整数値sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

integer setFloatValue(integer udf_handle, character*(*) target, real source)

udf_handleで指定されるUDFファイルのデータパスtargetの位置にreal値 sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

integer setDoubleValue(integer udf_handle, character*(*) target, double precision source)

udf_handleで指定されるUDFデータのデータパスtargetの位置にdouble precision値 sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

integer setStringValue(integer udf_handle, character*(*) target, integer limit,
character*(*) source)

udf_handleで指定されるUDFデータのデータパスtargetの位置に文字列値sourceを与える。limitはここでは利用されていない。関数戻り値はデータのセットに成功すれば0、失敗すれば-1 となる。

integer setIntValueInArray(integer udf_handle, character*(*) target, integer source)

udf_handleで指定されるUDFデータのデータパスtargetが配列内の位置を示している時にそこに一個の整数値sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1となる。

integer setFloatValueInArray(integer udf_handle, character*(*) target, real source)

udf_handleで指定されるUDFデータのデータパスtargetが配列内の位置を示している時にそこに一個のreal値sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1となる。

integer setDoubleValueInArray(integer udf_handle, character*(*) target,
double precision source)

udf_handleで指定されるUDFファイルのデータパスtargetが配列内の位置を示している時にそこに一個のdouble precision値 sourceを与える。関数戻り値はデータのセットに成功すれば0、失敗すれば-1となる。

integer setStringValueInArray(integer udf_handle, character*(*) target, integer limit,
character*(*) source)

udf_handleで指定されるUDFデータのデータパスtargetが配列内の位置を示している時にそこに一個の文字列値sourceを与える。limitはここでは利用されていない。関数戻り値はデータのセットに成功すれば0、失敗すれば-1となる。

integer setIntArray(integer udf_handle, character*(*) target, integer size_list(*),
integer source(*))

udf_handleで指定されるUDFデータのデータパスtargetが配列を示している時（文字列"[]"を含む）にそこに整数値配列sourceを書き込む。size_listは配列の各次元のサイズである（次元数はtargetで決まる）。関数戻り値はデータのセットに成功すれば0、データのセットに失敗した場合-1となる。

integer setFloatArray(integer udf_handle, character*(*) target, integer size_list(*),
real source(*))

udf_handleで指定されるUDFデータのデータパスtargetが配列を示している時（文字列"[]"を含む）にそこにreal値配列sourceを書き込む。size_listは配列の各次元のサイズである（次元数はtargetで決まる）。関数戻り値はデータのセットに成功すれば0、データのセットに失敗した場合-1となる。

integer setDoubleArray(integer udf_handle, character*(*) target, integer size_list(*),
double precision source(*))

udf_handleで指定されるUDFデータのデータパスtargetが配列を示している時（文字列"[]"を含む）にそこにdouble precision値配列sourceを書き込む。size_listは配列の各次元のサイズである（次元数はtargetで決まる）。関数戻り値はデータのセットに成功すれば0、データのセットに失敗した場合-1となる。

integer setStringArray(integer udf_handle, character*(*) target, integer size_list(*),
integer limit, character*(limit) source(*))

udf_handleで指定されるUDFデータのデータパスtargetが配列を示している時（文字列"[]"を含む）にそこに文字列値配列sourceを書き込む。size_listは配列の各次元のサイズである（次元数はtargetで決まる）。limitはsourceの各要素の文字列長である。関数戻り値はデータのセットに成功

すれば 0、データのセットに失敗した場合-1 となる。

integer writeUDFData(integer udf_handle)

udf_handleで指定されるUDFデータをファイルに書き出す。関数戻り値は書き出しに成功すれば 0、失敗した場合-1 となる。

integer getSize(integer udf_handle, character*(*) target, integer result)

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）配列データの大きさをresultに得る。多次元配列の場合には対象となる次元のサイズの積が得られる。

integer getSizeList(integer udf_handle, character*(*) target, integer num_dim,
integer size_list(1:num_dim))

udf_handleで指定されるUDFファイルのデータパスtargetが配列を示している時（文字列"[ ]"を含む）配列データの次元数をnum_dim に、各配列次元のサイズを配列size_list にそれぞれ返す。

第７章　ビルド方法

コンパイル・リンクに用いる環境は、Linux（kernel 2.2.9 以上）およびMicrosoft Windowsである。

Microsoft Windows上で構築を行うためには、CygwinパッケージまたはMicrosoft Visual Studioが必要である。Cygwinパッケージは、http://cygwin.com/ からダウンロードできる。

LinuxおよびCygwinでのコンパイル・リンクには、gcc 3.3 以上、glibc 2.1.2 以上が必要である。

7.1　ライブラリのビルド

OCTAインストーラにはLinux用のライブラリが付属しているが、アプリケーションをビルドする環境（コンパイラのバージョンなど）と一致していない場合は、リンクエラーが起こることがある。このような場合、ユーザのビルド環境でライブラリをソースコードからビルドする必要がある。

各環境別にビルド用のmakefileが提供されているため、ライブラリのビルドはきわめて簡単である。ライブラリのビルドは以下の手順で行う。

・ソースツリーの展開

GOURMETとAPIライブラリを構築するためのソースツリーを展開する。

src/ディレクトリにソースアーカイブを展開するためには、GOURMETのインストールディレクトリ（環境変数PF_FILESに設定されているディレクトリ）で以下のコマンドを実行する。

```
% cd $PF_FILES
% gunzip < gourmet2005_src.tar.gz | tar xvf -
```

・APIライブラリ構築

src/ディレクトリに移動した後、makeコマンドを実行（パラメータに何も指定しなければ、ライブラリのみがビルドされる）

```
% cd src
% make
```

ビルドが終了し（マシンによりかなり時間がかかることがる）、エラーメッセージが出ていないことを確認する。

・インストール

ライブラリ（$PF_FILES/lib/$PF_ARCH/libplatform.a）と、ヘッダファイル（$PF_FILES/include/以下）のインストールを行なう。ここで、$PF_ARCH はwin32、cygwin、linux などビルド・実行を

行う環境の名称である。

% make install

以上で、ライブラリのビルドとインストールが完了する。

7.2 アプリケーションのビルド

アプリケーション作成に必要なインタフェースライブラリ関係のインクルードファイルは $PF_FILES/include/ 以下、ライブラリは、$PF_FILES/lib/$PF_ARCH 以下にビルド環境別に提供されている。

・ Linux・Cygwin 版

Makefile例が$PF_FILES/tutorial/makeinterface/ 以下に準備されているので参考のこと。もっと複雑な例としては、OCTAエンジンのmakefileやライブラリのmakefileが参考になる。

・Microsoft Visual Studio用プロジェクト

Microsoft Visual Studio用プロジェクトが$PF_FILES/src/winbuild/libplatformに提供されている。ワークスペースファイル（libplatform.dsw）をMicrosoft Visual Studioで開き、ビルドすることにより$PF_FILES/lib/win32 にlibplatform.lib が生成される。（デバッグ版のライブラリ名はlibplatform_d.libである。）エンジンのビルドに必要なインクルードファイルは、$PF_FILES/include に置かれている。もしインクルードファイルが$PF_FILES/includeにない場合は、$PF_FILES/include/_copyInclude.bat を$PF_FILES/includeディレクトリで実行することにより必要なインクルードファイルが$PF_FILES/includeにコピーされる。

Windows Nativeのバイナリモジュールは、Cygwin+MinGW によってもビルドできる。

7.3 windows 用バイナリ配布時の注意事項

Cygwin環境で構築したwindows用バイナリアプリケーションを第三者に配布する場合、以下の点に注意する必要がある。

・cygwin1.dllの使用が前提とならないようにする

Cygwin は Gnu Public Liscence(GPL) により公開されており、cygwin1.dllの使用を前提とするアプリケーションは、GPL のもとで配布されることが要求されている。OCTA はフリーではあるが、OCTA使用許諾契約に基づいて配布されており、上記のような場合はライセンス上の問題が生ずる。

このような問題を回避するために、Cygwinパッケージに含まれているMinGWライブラリを用いることができる。Cygwin+MinGW パッケージでビルドしたバイナリは、Win32 Native API を用いるWindowsコマンドラインアプリケーションとなるため、実行時にcygwin1.dll を使用しない。OCTA2003 エンジンは、上記の方法で配布されている。

具体的な方法は、コンパイル・リンク時にgcc のパラメータとして、-m nocygwin を与えるだけである。

・第三者に配布しない場合はMinGWライブラリを用いる必要は無い

Windowのコマンドラインを使いにくいと考える多くの利用者は、MinGWパッケージを用いることなく自分でビルドしたlibplatformライブラリとアプリケーションを用いることができる。この場合は、自分でダウンロードしたCygwin環境を用いているため、ライセンス上の問題は無い。

第８章　インタフェースクラス自動生成ツール

8.1　概要

「makeinterface」は、指定されたUDFファイルの定義情報を解析して、その定義内容と同じUDFオブジェクトを入出力するためのインタフェースクラス定義を生成するツールである。このツールで生成されたクラス定義を用いることにより、UDFのトップオブジェクト（UDF data部で名称が記述されているデータ）を指定すれば、そのオブジェクトの構造全部を確実に、一度に入出力できる。

「makeinterface」は、標準提供されているテンプレートファイルを指定することにより、IObjectインタフェースクラスおよび UObjectインタフェースの両方のヘッダファイルを生成できる。

また、生成方法を指定するスクリプトファイルで、ユーザ定義クラスへのバインドや KEY/IDクラスとして扱うUDFオブジェクトを選定することが可能である。

「makeinterface Version 3.0」（OCTA2005）では、エンジン開発者が長期間に渡ってUDF入出力ファイルをメンテナンスする作業をサポートするための機能として、複数の異なるバージョンのUDFファイル（通常少しずつデータ構造が変化していく）の差異を解析し、自動的にバージョンに合致したUDFファイルを入出力できるインタフェースクラス定義を生成するようにした。UDF定義のバージョンアップ対応機能については付録Bに詳述した。

8.2　使用方法

Usage:

```
makeinterface -D udfdef_file1 [udfdef_file2 udfdef_file3 ...] -F [N]
              [-N interface_name ] [-I template_file]
              [-O output_file] [-C class_prefix] [-A attribute_prefix]
              [-S script_file]
```

Parameters:

-D udfdef_fileN: UDF定義ファイル名

指定されたUDFファイルのdef部の情報を解析してインターフェースクラスを作成する。

複数のUDFファイルを指定した場合、バージョン解析を行う。指定するUDFファイルの順序はバージョン順であると解釈する。変数名の自動変更（付録B参照）が発生するような時には後ろに指定する（新しい）UDFファイルのデータ名が優先的に使用される。

指定するUDF定義ファイルは¥include を含んでもよく、その場合カレントディレクトリに存在しなければ、環境変数UDF_DEF_PATHで指定されたディレクトリを探す。

-F [N]: ＵＤＦ書き出しメソッド(_Output)のバージョン固定機能

ＵＤＦ書き出しメソッド(_Output)を、-Dで指定されたＵＤＦファイルの内の１つに固定する。Nは-Dで指定したＵＤＦファイルのインデックス番号（ゼロから始まる）である。

Nを省略した場合は、最後のＵＤＦファイル定義に対応したＵＤＦ書き出しメソッドが生成される。

-N interface_name: インタフェース名称

%INTERFACE_NAME% マクロの置き換えとして使用される。このマクロは、ヘッダファイルのインクルード検査やcommon udfクラス名に用いられる。デフォルト名は、class_prefix + (udfdef file) の拡張子を除いた文字である。

-I template_file: テンプレートファイルパス名

指定されたテンプレートファイルのマクロ部分を置き換えてインターフェースクラスを作成する。

template_file を編集することにより、生成内容を変更することができる。デフォルト名は、template.txt。テンプレートファイルが、実行時カレントディレクトリになければ、makeinterfaceモジュールのあるディレクトリが探索される。

-O output_file: 出力するヘッダファイルパス名

デフォルト名は、interface_name + ".h" である。

-C class_prefix: 生成するインターフェースクラス名に付けるプレフィックス

デフォルトは、"I"でIObject インタフェースの生成を行う。"U"を指定すると、UObject インタフェースの生成を行う。

-A attribute_prefix: 生成するメンバ属性名に付けるプレフィックス

ディフォルトは、なし。

-S script_file: 生成方法指定ファイル（スクリプトファイル）

script_file を編集することにより、生成方法を変更することができる。デフォルトは、script.txt。スクリプトファイルが、実行時カレントディレクトリになければ、makeinterface モジュールのあるディレクトリが探索される。

8.3 生成テンプレートファイル

template_fileを編集することにより、インターフェースクラスの生成内容を変更することができる。

template_fileの中で、マクロ定義された内容が解析されたUDF名称に置き換えられ、マクロルールに従って繰り返し出力される。

マクロルールとは$で囲まれた文字列で、制御指定を行う役割を持ち、出力は行われない。

マクロルールは、以下のものが使用できる。

表 8.1：makeinterface マクロルール一覧

$CLASS_TEMPLATE_BEGIN$	個々のクラス定義の開始。次行から $CLASS_TEMPLATE_END$ の前行までの内容が、生成クラスの数だけ繰り返される。
$CLASS_TEMPLATE_END$	個々のクラス定義の終了
$ATTRIBUTE_TEMPLATE_BEGIN$	属性定義の開始。次行から $ATTRIBUTE_TEMPLATE_END$ の前行までの内容が、生成中クラスの属性数だけ繰り返される。
$ATTRIBUTE_TEMPLATE_END$	個々の属性定義の終了
IF_BASE_TYPE	条件分岐ルールの開始。生成中属性の種別が基本型である場合、次行から次の条件分岐ルールまたは、IF_TYPE_END の前行までの内容が生成される。
IF_USER_TYPE	条件分岐ルールの開始。生成中属性の種別がユーザ定義クラスである場合、次行から次の条件分岐ルールまたは、IF_TYPE_END の前行までの内容が生成される。
IF_ARRAY_TYPE	条件分岐ルールの開始。生成中属性の種別が基本型配列である場合、次行から次の条件分岐ルールまたは、IF_TYPE_ENDの前行までの内容が生成される。
$IF_USER_ARRAY_TYPE$	条件分岐ルールの開始。生成中属性の種別がユーザ定義クラス配列である場合、次行から次の条件分岐ルールまたは、IF_TYPE_ENDの前行までの内容が生成される。
IF_MAP_TYPE	条件分岐ルールの開始。生成中クラスの種別が、KEYまたはIDの場合、次行からIF_TYPE_ENDの前行までの内容が生成される。
IF_TYPE_END	条件分岐ルールの終了
$IF_VERSION_BRANCH_BEGIN$	$ATTRIBUTE_TEMPLATE_BEGIN$ ～ $ATTRIBUTE_TEMPLATE_END$ 内で複数行に渡ってバージョン分岐処理が必要な場合に使用する。バージョン分岐の開始を示す。複数のUDF定義ファイルが指定されていない時は、無視され、出力ファイルに現れない。
$IF_VERSION_BRANCH_END$	バージョン分岐の終了を示す。

マクロ定義は%で囲まれた文字列で、処理内容に応じて置換された文字列が出力される。また、LISTで終わるマクロ定義は行マクロであり、行頭から行末までがリストに対応する行数だけ出力される。従って行マクロは、入れ子にはできない。

表 8.2 のマクロ定義は、テンプレートファイルのどこでも使用できる。

表 8.2：makeinterface マクロ定義一覧(1)

%GENERATED_DATE%	自動生成を実行した日付、時刻。
%GENERATED_VERSION%	自動生成を実行したmakeinterfaceのバージョン。
%INTERFACE_NAME%	インタフェース名称。
%INCLUDE_LIST%	スクリプトでinclude指定したファイルのリスト。 インクルードファイル指定(include ” xxxx”) などに用いる。(3)スクリプトファイル参照。
%DEFINITION%	UDF定義ファイル名
%CLASS_LIST%	生成する全クラスのリスト。プロトタイプ宣言(class xxxx;) などに用いる。
%KEY_CLASS_LIST%	KEYクラスのリスト。スクリプトで、key指定したオブジェクトのインタフェースクラス名リスト。(8.4 スクリプトファイル　参照)。
%ID_CLASS_LIST%	IDクラスのリスト。スクリプトで、id指定したオブジェクトのインタフェースクラス名リスト。(8.4 スクリプトファイル　参照)。
%OBJECT_NAME%	現在生成中のオブジェクトに対応するUDF変数名。
%KEY_OBJECT_LIST%	KEYクラスのリスト。スクリプトで、key指定したオブジェクトのUDF変数名リスト。(8.4 スクリプトファイル　参照)。
%ID_OBJECT_LIST%	IDクラスのリスト。スクリプトで、id指定したオブジェクトのUDF変数名リスト。(8.4 スクリプトファイル　参照)。
%DEFINE_VERSION_LIST%	バージョン分岐処理のために必要なバージョン文字列の定義に置き換えられる。 #define UDF_VERSION_NN ″バージョン文字列″ の形式の行が、指定したUDF定義ファイル数分生成される。

表 8.3 のマクロ定義は、マクロルール$CLASS_TEMPLATE_BEGIN$から$CLASS_TEMPLATE_END$の間だけで使用できる。

表 8.3：makeinterface マクロ定義一覧(2)

%CLASS_NAME%	現在生成中のクラス名。
%ATTRIBUTE_PARAMETER_LIST%	コンストラクタで用いられるパラメータリスト。
%ATTRIBUTE_INITIALIZE_LIST%	コンストラクタで用いられる属性の初期化宣言リスト。
%ATTIBUTE_DEFINITION_LIST%	属性定義リスト。
%ATTRIBUTE_INPUT_LIST%	全属性の入力変数リスト。使用例：

	is << %ATTRIBUTE_INPUT_LIST%;
%ATTRIBUTE_OUTPUT_LIST%	全属性の出力変数リスト。使用例： os << %ATTRIBUTE_OUTPUT_LIST% << ″ ″;
%PREFIX%	上記出力時のフォーマット前置文字列
%POSTFIX%	上記出力時のフォーマット後置文字列。使用例： os << %PREFIX% %ATTRIBUTE_OUTPUT_LIST% %POSTFIX% << ″ ″;
%ATTRIBUTE_DEFAULT_LIST%	C/C++プログラムの代入文に置き換えられる。
%ATTRIBUTE_OUTPUT_LIST_WITHVERSION%	%ATTRIBUTE_OUTPUT_LIST%にバージョン分岐処理が追加される。

表 8.4 のマクロ定義は、マクロルール$ATTRIBUTE_TEMPLATE_BEGIN$ から$ATTRIBUTE_TEMPLATE_END$ の間だけで使用できる。

表 8.4：makeinterface マクロ定義一覧(3)

%ATTRIBUTE_NAME%	現在生成中の属性名。
%ATTRIBUTE_SYMBOL%	上記に対応するUDF 変数名。
%ATTRIBUTE_PARAMETER_LIST%	コンストラクタで用いられるパラメータリスト。

UDF 定義ファイルのバージョン分岐処理に用いるために、下記のバージョン識別文字列を使う。

_VERSION_STRING_	バージョン分岐処理に使用する。 UDF ヘッダー情報の EngineVersion に設定されている文字列をもつ std::string 型の変数。

これらのマクロを使用したテンプレートファイル例を以下に示す。

template3.txt (IObject インタフェースを生成する標準テンプレートファイル)

```
// This file is ganerated by makeinterface tool in Platform utility.
// You may edit this file to add any class or method, and to change output format etc.
//       template version V3.0
//                          Generated by: %GENERATED_VERSION%
//                          Generated date: %GENERATED_DATE%

#ifndef _%INTERFACE_NAME%_H_
#define _%INTERFACE_NAME%_H_

#include ″udfobject.h″
#include %INCLUDE_LIST%
```

```
%DEFINE_VERSION_LIST%

// UDF header informations.
extern string projectName;
extern string engineName;
extern string engineVersion;
extern string ioType;
extern string comment;
extern string action;
extern string currentVersion;
extern TAB tab;    // tab formatting object for ostream
#define _VERSION_STRING_ engineVersion

#if !defined(_IUdfInformation_H_)
class IUdfInformation {
public:
         IUdfInformation(const string& file="%DEFINITION%") : deffile(file) {}

         // UDF File Information
         const string& getDefinition() { return deffile; }

private:
         string deffile;
};
#endif

class %CLASS_LIST%;

$CLASS_TEMPLATE_BEGIN$
class %CLASS_NAME% : public UDFObject {
public:
         %CLASS_NAME%() : UDFObject() {
                  %ATTRIBUTE_DEFAULT_LIST%
         }
         virtual ~%CLASS_NAME%() {}
         virtual const char* GetName() const { return "%OBJECT_NAME%"; }
         %ATTRIBUTE_DEFINITION_LIST%

public:
         virtual void _Input(UDFistream& is) {
$ATTRIBUTE_TEMPLATE_BEGIN$
                  $IF_VERSION_BRANCH_BEGIN$
$IF_NATIVE_TYPE$
#if defined(USE_TEXT_TEMP)
                  is >> %ATTRIBUTE_NAME%;
#else
                  is.read((char *)& %ATTRIBUTE_NAME%, sizeof(%ATTRIBUTE_NAME%));
#if defined(DEBUG)
                  cout << %ATTRIBUTE_NAME% << " ";
#endif
#endif
$ELSE_OTHER_TYPE$
                  is >> %ATTRIBUTE_NAME%;
$IF_TYPE_END$
```

```
                $IF_VERSION_BRANCH_END$
$ATTRIBUTE_TEMPLATE_END$
$IF_MAP_TYPE$
#if defined(USE_COMMON_SUPORT)
                if (getRoot())     getRoot()->addMAP(this);
#endif
$IF_TYPE_END$
        }
        virtual void _Output(ostream& os) {
                os << tab << "{";
                tab.enter();
                os << %ATTRIBUTE_OUTPUT_LIST_WITHVERSION% << " ";
                tab.exit();
                os << tab << "}¥n";
        }
};
$CLASS_TEMPLATE_END$

#endif
```

8.4 スクリプトファイル

script_file でのスクリプト指定により、生成方法を変更することができる。スクリプトで指定した方法でtemplate file のマクロ定義の置換内容が変更される。

スクリプトは識別子と空白で区切られたパラメータからなる。現在のバージョンで使用できるものは表8.5の通り。

表 8.5：makeinterface スクリプト一覧

識別子	パラメータ	機能
bind	インタフェースクラス名、C++ユーザクラス名	1) %CLASS_LIST%がC++ユーザクラス名に置換される。 2) マクロルールによる生成は行われない。
default	データ名 = デフォルト値	%ATTRIBUTE_DEFAULT_LIST%マクロが初期値指定文に置き換えられる。
identify	置き換えられるデータ名= 置き換え後データ名	1) バージョン解析時に置き換えられたデータ名が使用される。 2) C++インターフェースクラス定義ファイルのデータ名が置き換えられる。
id	UDF定義部のIDクラス名	%ID_OBJECT_LIST%が、指定クラス名の列挙に置換される。
include	バインド後のコンパイルに必要なインクルードファイル	%INCLUDE_LIST%が、指定ファイル名に置換される。
key	UDF定義部のKEYクラス名	%KEY_OBJECT_LIST%が、指定クラス名の列挙に置換される。
simple	特定スクリプト処理を行なうクラス名	%SIMPLE_OBJECT_LIST%が、指定クラス名の列挙に置換される。
top	UDFデータ部のトップオブジェクト名	%TOP_OBJECT_LIST%が、指定オブジェクト名の列挙に置換される。

これらのスクリプトを使用したスクリプトファイル例を以下に示す。common_udf_script.txt（共通UDFを生成するスクリプト）

スクリプトファイル指定例

```
// This file is default script for makeinterface tool in Platform utility.
// To Do: bind IObject to user class
bind IVector3D Vector3d
// To Do: include user class header
include "Vector3d.h"
// To Do: sujestion to makeinterface tool
// top objects
top parameter
top mesh
top field
top dynamics_manager
// keymap objects
key PartialRegion
key PartialRegionCondition
key ScalarField
key VectorField
key TensorField
key Parameter
key Procedure
// idmap objects
id Vertex
id Edge
id Face
id Cell
// other objects
simple Mesh
simple StructuredMesh
simple UnStructuredMesh
```

8.5　現バージョンの制約事項

・ インタフェースクラスにユーザ定義クラスへのポインタをバインド指定した場合、対象インタフェースクラスがUDFArray の要素となっている状況には対応できない。
この場合、生成したインタフェースクラスを利用してUDF ファイル内の対応オブジェクトを読み込む時点でコンストラクタが異常終了となる。

第９章　本リリースで確認されているバグ

確認されているバグを以下に示す。

1. makeinterface ツールで、インタフェースクラスにユーザ定義クラスへのポインタをバインド指定した場合、対象インタフェースクラスがUDFArray の要素となっている状況では、生成したインタフェースクラスを利用してUDF ファイル内の対応オブジェクトを読み込む時点でコンストラクタが異常終了する。

第１０章　エラーメッセージ一覧

10.1　PFException関連

PFExceptionがThrowされた場合のエラーメッセージを以下に示す。

・ out of memory ...

　　メモリーが不足している。

・ illegal use of OpenUDF.

　　OpenUDF の呼び出しシーケンス、またはパラメータ指定が間違っている。

・ illegal use of CreateUDF.

　　CteateUDF の呼び出しシーケンス、またはパラメータ指定が間違っている。

・ illegal use of AppendUDF.

　　AppendUDF の呼び出しシーケンス、またはパラメータ指定が間違っている。

・ file not found.

　　指定したファイルが見つけられない。ファイルがUDF定義ファイルである場合、環境変数

　　UDF_DEF_PATHの指定が間違っている可能性がある。

10.2　LocationException関連

LocationExceptionがThrowされた場合のエラーメッセージを以下に示す。

・ illegal use of Location

　　Locationクラスの使用方法が間違っている。

10.3　UDFhandleException関連

UDFhandleException がThrow された場合のエラーメッセージを以下に示す。

・ FATAL ERROR: UDFistream read error occured.

10.4　UDFstreamException関連

UDFstreamExceptionがThrowされた場合のエラーメッセージを以下に示す。

・ FATAL ERROR: UDFistream read error occured.

UDFistreamから指定されたデータ群を規定の順序で読みこめない。インタフェースオブジェクトヘッダファイルの間違いまたは、プログラムミスの可能性が高い。

10.5 UDFStreamException関連

UDFStreamExceptionがThrowされた場合のエラーメッセージを以下に示す。

・ UDFistream read error occured.

UDFistreamから指定されたデータ群を規定の順序で読みこめない。IObjectインタフェースのヘッダファイルが、UDF定義と整合性が無い場合に表示されることが多い。

・ UDFostream write error occured.

UDFostreamへ指定されたデータ群を書き出せない。IObjectインタフェースのヘッダファイルが、UDF定義と整合性が無い場合に表示されることが多い。

10.6 UDFManagerException関連

UDFManagerExceptionがThrowされた場合のエラーメッセージを以下に示す。

・ illegal use of Location

Locationクラスの使用方法が間違っている。

10.7 UDFObjectException関連

UDFObjectException がThrow された場合のエラーメッセージを以下に示す。

・ UDFObject array size error occured.

UDFarrayオブジェクトを内部ストリームから読み取る場合の個数がおかしい。IObjectインタフェースのヘッダファイルが、UDF定義と整合性が無い場合に表示されることが多い。

10.8 UDFパーサ関連

Parse Errorメッセージは UDF書式またはデータの間違いの可能性がある。詳細は「OCTA UDF 文法書」を参照のこと。

10.9　内部エラー

Platform core library Error メッセージが表示された場合、ライブラリの内部エラーである。このエラーメッセージが出た場合、

http://octa.jp/

のOCTA BBSまでご連絡ください。

付録 A　添付資料：makeinterface チュートリアル

「インタフェースクラス生成方法の入門」

makeinterfaceツールの機能概要

makeinterfaceツールは、指定したUDFファイルのデータを入出力するインタフェースクラス定義を自動生成するツールです。

UDFのデータ構造が複雑になると、C++で入出力を行うプログラムを作成することが面倒になってきますが、makeinterfaceツールを使えば少しの指定のみでこれを自動的に行うことができます。

ここで行うチュートリアルの内容は以下のとおりです。

1. 簡単なクラス定義の自動生成
   指定したUDFのヘッダ情報を出力するIUdfInformation クラスを自動生成し、実際に動かしてみます。
   このチュートリアルを終了すると、makeinterfaceツールの基本的な使い方が理解できます。

2. UObjectインタフェースの生成
   UDFManagerを使用するインタフェースクラスを自動生成するための汎用テンプレート
   (uobject_template.txt)を利用し、実際に動かしてみます。このチュートリアルを終了すると、
   UDFManager をより高度に利用する方法が理解できます。

3. IObject インタフェースの生成
   PFinterfaceを使用するインタフェースクラスを自動生成するための汎用テンプレート
   (template.txt)を利用し、実際に動かしてみます。

このチュートリアルを終了すると、主要なエンジンで使用されているPFinterfaceインタフェースを使用して、UDFファイルのI/O を実装する方法が理解できます。
また、UDFデータからC++変数へのマッピングを指示する方法も理解できます。チュートリアルの実行に必要な全てのソース・ファイルは、$PF_HOME/tutorial/makeinterface 以下に準備してあります。
さあ、チュートリアルを始めましょう。

A.1　簡単なクラス定義の自動生成

A.1.1　簡単なクラス定義の自動生成

指定したUDFのヘッダ情報を出力するIUdfInformationクラスを自動生成し、実際に動かしてみます。makeinterfaceツールの基本的な使い方が理解できます。

info_template.txt の内容を見てください。

これは、指定したUDF のヘッダ情報を出力するインタフェースクラスヘッダを生成するためのテンプレートです。

以下のシンボルはマクロであり、自動生成時に置き換えられます。

%GENERATED_DATE%　自動生成された日付、時刻

%INTERFACE_NAME%　インタフェースクラス名

%DEFINITION%　　UDF定義ファイル名

```
// This file is ganerated by makeinterface tool in Platform utility.
// You may edit this file to add any class or method, and to change output format etc.
// Generated date: %GENERATED_DATE%
#ifndef _%INTERFACE_NAME%_H_
#define _%INTERFACE_NAME%_H_
#include "udfobject.h"
#include "pfinterface.h"
class %INTERFACE_NAME% {
public:
        %INTERFACE_NAME%(const string& file="%DEFINITION%") : deffile("%DEFINITION%")
                {
                        UDFhandle *udfin = OpenUDF(file.c_str());
                        recnum = udfin->getTotalRecord();
                        project = udfin->getProjectName();
                        engine = udfin->getEngineName();
                        version = udfin->getEngineVersion();
                        iotype = udfin->getIOType();
                        comment = udfin->getComment();
                        CloseUDF(udfin);
                }
        // UDF File Information Access method
        const string& getDefinition() { return deffile; }
        const int getRecordNum() { return recnum; }
        const string& getProjectName() { return project; }
        const string& getEngineName() { return engine; }
        const string& getEngineVersion() { return version; }
        const string& getIOType() { return iotype; }
        const string& getComment() { return comment; }
private:
        string deffile;
```

```
        size_t recnum;
        string project;
        string engine;
        string version;
        string iotype;
        string comment;
};
#endif
```

A.1.2　makeinterface を使用して、自動生成してみます。

以下のコマンドを実行します。

```
makeinterface -D myudf.udf -I info_template.txt -N IUdfInformation
```

ここで、パラメータの意味は以下の通りです。

-D UDFファイル名

-I テンプレートファイル名

-N インタフェース名

A.1.3　生成された IUdfInformation.h の内容を見てください。

マクロの内容が置き換えられていることを確認してください。

```
// This file is ganerated by makeinterface tool in Platform utility.
// You may edit this file to add any class or method, and to change output format etc.
// Generated date: Mon Apr 16 19:56:01 2001
#ifndef _IUdfInformation_H_
#define _IUdfInformation_H_
#include "udfobject.h"
#include "pfinterface.h"
class IUdfInformation {
public:
        IUdfInformation(const string& file="myudf.udf") : deffile("myudf.udf")
                {
                        UDFhandle *udfin = OpenUDF(file.c_str());
                        recnum = udfin->getTotalRecord();
                        project = udfin->getProjectName();
                        engine = udfin->getEngineName();
                        version = udfin->getEngineVersion();
                        iotype = udfin->getIOType();
                        comment = udfin->getComment();
                        CloseUDF(udfin);
                }
        // UDF File Information Access method
        const string& getDefinition() { return deffile; }
        const int getRecordNum() { return recnum; }
        const string& getProjectName() { return project; }
        const string& getEngineName() { return engine; }
```

```
        const string& getEngineVersion() { return version; }
        const string& getIOType() { return iotype; }
        const string& getComment() { return comment; }
private:
        string deffile;
        size_t recnum;
        string project;
        string engine;
        string version;
        string iotype;
        string comment;
};
#endif
```

A.1.4　次に **udfinfo test.cpp** の内容を見てください。

このプログラムは、IUdfInformationクラスを使用するテスト用のプログラムです。makeinterfaceで自動生成したヘッダファイルをインクルードし、UdfInformationクラスのインスタンスを構築して、UDFのヘッダ情報を問い合わせます。

```
#if defined(WIN32)
#pragma warning(disable:4786)
#endif
// makeinterface で自動生成したヘッダファイル
#include "IUdfInformation.h"
void UDFInfoTest(const char *input_udfpath)
{
        // 自動生成されたUDF情報クラスのインスタンスを作ります。
        // コンストラクタでUDF内容を解析して各情報を取得し、UDFファイルは閉じられます。
        IUdfInformation info(input_udfpath);
        cout << "====== " << input_udfpath << " =======" << endl;
        // UDF情報クラスインスタンスへ問い合わせます。
        cout << "UDF definition:¥t" << info.getDefinition() << endl;
        cout << "total record number:¥t" << info.getRecordNum() << endl;
        cout << "Project name:¥t" << info.getProjectName() << endl;
        cout << "Engine name:¥t" << info.getEngineName() << endl;
        cout << "Engine version:¥t" << info.getEngineVersion() << endl;
        cout << "IN/OUT type:¥t" << info.getIOType() << endl;
        cout << "Comment:¥t" << info.getComment() << endl;
        cout << "=============" << endl;
}
```

A.1.5　テストプログラムをビルドして、実行してみます。

以下のコマンドでテストプログラムをビルドします。

```
make testinterface
```

以下のコマンドでテストプログラムを実行します。

```
testinterface -T1 -I myudf.udf
```

指定したmyudf.udfのヘッダ情報が、標準出力に表示されます。ここで、-T1 はチュートリアル１を指定するためのオプションです。

```
====== myudf.udf =======
UDF definition: myudf.udf
total record number: 0
Project name: makeinterface tutorial
Engine name: myEngine
Engine version: TestVersion
IN/OUT type: IN
Comment: This file is sample UDF file for makeinterface tutorial.
=============
```

「自分でプログラミングするのと何が違うの」と思われましたか？その通りです。但し、大きな違いがあります。makeinterface では、テンプレートをプログラムするのです。これにより、変数名が異なるだけの繰り返しプログラミングから開放されます。

以上のように、makeinterfaceツールを使用する手順は以下の通りとなります。

1. UDFファイルを作成する。
2. テンプレートをプログラミングする。
3. makeinterfaceツールで、インタフェースクラスのヘッダファイルを自動生成する。
4. インタフェースクラスを利用するプログラムを作成する。
5. プログラムをビルドする。

次に、makeinterfaceの一般的な使い方として、標準提供されているテンプレートを使用する方法を説明します。

A.2　UObjectインタフェースの生成

UDFManagerを使用するインタフェースクラスを自動生成するための汎用テンプレート（uobject_template.txt）を利用し、実際に動かしてみます。UDFManagerをより高度に使用する方法が理解できます。

UDFManagerでは、UDF変数名を指定して入出力を行えるため、簡便に使用することができますが、複雑な構造を持つデータの場合、対応するC++クラスを定義する必要があるため、プログラム量が非常に多くなります。

UDFファイルには、データ構造も定義されているので、この情報を利用してC++クラスの自動生成を行う

ことが可能です。これを行うための汎用テンプレート（uobject_template.txt）が、準備されています。

A.2.1　uobject_template.txt の内容を見てください。

このテンプレートの内容を全部理解する必要はありませんが、簡単に説明します。中核となるマクロは、$CLASS_TEMPLATE_BEGIN$ から$CLASS_TEMPLATE_END$ までで、UDFファイルの中で定義されているデータ構造に対応するC++インタフェースクラスの定義が生成されます。

この中では、

%CLASS_NAME% インタフェースクラス名

%ATTRIBUTE_DEFINITION_LIST% 全ての属性定義

などが置換されます。

他のマクロの機能の説明については、ご興味があれば第８章「インタフェースクラス自動生成ツール」をご覧下さい。

A.2.2　uobject_test.cpp の内容を見てください。

これは、UDFManagerを利用し、指定したUDFファイルのデータを読み込み、同じ内容を、指定したレコード数だけ指定した別のUDFファイルへ出力するプログラムです。

先頭でインクルードしているヘッダファイル、IUdfInformation.hは、チュートリアル１で生成したヘッダファイル、Umyudf.hは、myudf.udfの定義に対応して生成するインタフェースクラスのヘッダファイルです。

UmoleculeはUDFファイルのデータに対応するインタフェースクラスであり、このインスタンス molecule へUDF のmoleculeデータを読み込みます。

出力は、指定されたレコード回のループを行い、インタフェースオブジェクトインスタンスの内容をそのまま出力していますが、実際には、計算結果をインタフェースオブジェクトへ設定する処理がこの間に入ることになります。

```
#if defined(WIN32)
#pragma warning(disable:4786)
#endif
// UDFManager 入出力サンプル
// 2001/1/10 JRI Y.Nishio
#include "IUdfInformation.h"
#include "Umyudf.h"
//*************************************/
// To do:
// 入出力するオブジェクトを宣言してください。
//*************************************/
static Umolecule molecule;
//*************************************/
// 読み込みテスト
```

```
// seek() を使うので、uf はconstではない
static void ValueInput(UDFManager & uf)
{
        //************************************/
        // To do:
        // 入力するUDFトップオブジェクト名を指定してください。
        //************************************/
        molecule.get(uf, Location("molecule"));
        // 確認のため、読み込んだオブジェクトを表示する。
        cout << "molecule:" << molecule << endl;
}
// UDF読み込みエントリー
void UObjectInputTest(const char *udfpath, const int restart_step)
{
        try {
                UDFManager uf(udfpath, UDFManager::READ);
                UINT recnum = uf.totalRecord();
                cout << "record number:" << recnum << endl;
                if (restart_step >= 0 && restart_step < (int)recnum )
                uf.jump(restart_step);
                ValueInput(uf);
        } catch (Location::LocationException &e) {
                cerr << e.what() << endl;
        } catch (UDFManager::UDFManagerException &e) {
                cerr << e.what() << endl;
        }
}
// 計算結果を書き込む
// インタフェースオブジェクトの内容をそのまま書き出す
static void SimpleOut(const UDFManager & uf)
{
        try {
                // makeinterfaceでUObjectにput methodを定義させると、
                // 構造型の出力も、以下のように非常に簡単になります。
                molecule.put(uf, Location("molecule"));
        } catch (Location::LocationException &e) {
                cerr << e.what() << endl;
        } catch (UDFManager::UDFManagerException &e) {
                cerr << e.what() << endl;
        }
}
// UDF 出力エントリー
// IUdfInformation から取得した定義ファイル内容を使い、
// end_step レコードまでudfpath で指定したファイルへ出力する。
void UObjectOutputTest(const char *udfpath, const int end_step)
{
        try {
                IUdfInformation info;
                UDFManager uf(info.getDefinition(), UDFManager::READ);
                for(int i=0; i < end_step; i++) {
                        string recname = uf.newRecord("record");
                        cout << recname << " Output" << endl;
                        SimpleOut(uf);
                }
```

```
                uf.write(udfpath);
        } catch (UDFManager::UDFManagerException &e) {
                cerr << e.what() << endl;
        }
}
```

A.2.3　テストプログラムをビルドして、実行してみます。

以下のコマンドでテストプログラムをビルドします。

make testinterface

Makefile には、makeinterface での生成処理が記述されているので、myudf.udfを修正すれば自動的にUmyudf.hが更新されるようになっています。

以下のコマンドでテストプログラムを実行します。

testinterface -T2 -I myudf.udf -O testout2.udf -e2

指定したmyudf.udf のデータ内容が、標準出力に表示されます。ここで、-T2 はチュートリアル２を指定するためのオプションです。また、-O で指定したファイルに、-e で指定したレコードまでデータが出力されています。出力されたtestout2.udf を確認してみてください。

このように、makeinterface を使用すれば、UDF データの入出力についてプログラミングを行う必要がほとんどなくなります。

A.2.4　UDFManager に準備されている多くのメソッドを利用して、uobject_test.cpp の処理内容を変更してみるのも理解の助けになるでしょう。

メソッド仕様については、第5 章「パッケージインタフェース」5.7 UDFManager および5.8 UObject をご覧下さい。

さて、最後のチュートリアルを始めましょう。

インタフェースライブラリでは、UDFManagerパッケージの他にIObjectパッケージが提供されています。

このパッケージでは、UDFデータの入出力をストリームとして扱うことができ、他のデータ形式への出力も簡便に行うことができます。また、UDFデータからC++変数へのマッピングを指示することができます。

A.3　IObjectインタフェースの生成

PFinterfaceを使用するインタフェースクラスを自動生成するための汎用テンプレート（template.txt）を利用し、実際に動かしてみます。

主要なエンジンで使用されているPFinterfaceインタフェースを使用して、UDFファイルのI/O を実装する方法が理解できます。C++クラスの自動生成の考え方は、チュートリアル２と同じです。これを行うための汎用テンプレート（template.txt）が、準備されています。

また、UDFデータからC++変数へのマッピングを指示する方法も理解できます。これを指示するためにはスクリプトファイル(script.txt) を使用します。

A.3.1 template.txt の内容を見てください。

このテンプレートの内容を全部理解する必要はありませんが、簡単に説明します。最初に、UDF定義ファイル名を記憶させるためのIUdfInformationクラスが定義されています。

このクラスは、チュートリアル１で説明したものに置き換えることも可能ですが、ここでは単純なものを使用しています。

中核となるマクロは、(チュートリアル２と同じく) $CLASS_TEMPLATE_BEGIN$ から$CLASS_TEMPLATE_END$

までで、UDFファイルの中で定義されているデータ構造に対応するC++インタフェースクラスの定義が生成されます。

この中では、

%CLASS_NAME% インタフェースクラス名

%ATTRIBUTE_DEFINITION_LIST% 全ての属性定義

などが置換されます。

他のマクロの機能の説明については、ご興味があれば第７ 章「インタフェースクラス自動生成ツール」をご覧下さい。

A.3.2 script.txt の内容を見てください。

このファイルは、UDFデータからC++変数へのマッピングを指示するものです。bind IVector3D Vector3d

は、「UDF 定義のVector3D をインタフェースオブジェクトにマッピングする時に、Vector3d クラスとして宣言しなさい」と指示しています。

これにより、事前に構築されている共通ライブラリを使用することが可能になり、プログラム内で、Vector3dクラスのインスタンスを直接利用できます。

include "Vector3d.h"は、上記のライブラリを使用するためのインクルードファイルを指示します。この指示により、テンプレートファイルのマクロ%INCLUDE_LIST% が置換されます。

A.3.3 iobject test.cpp の内容を見てください。

これは、PFinterfaceを利用し、指定したUDFファイルのデータを読み込み、同じ内容を、指定したレコード数だけ指定した別のUDFファイルへ出力するプログラムです。(機能的にはチュートリアル２と全く同じです)

先頭でインクルードしているヘッダファイル、Imyudf.h は、myudf.udf の定義に対応して生成するインタフェースクラスのヘッダファイルです。

Imolecule はUDF ファイルのデータに対応するインタフェースクラスであり、このインスタンス molecule へUDF のmolecule データを読み込みます。PFinterface パッケージでは、ストリーム形式で入出力するため、入力前に指定したオブジェクトが存在するかどうかを判定する必要があります。

出力は、指定されたレコード回のループを行い、インタフェースオブジェクトインスタンスの内容をそのまま出力していますが、実際には、計算結果をインタフェースオブジェクトへ設定する処理がこの間に入ることになります。

```
#if defined(WIN32)
#pragma warning(disable:4786)
#endif
// UDF ファイル入出力サンプル
// 2000/10/27 JRI Y.Nishio
#include "pfinterface.h"
#include "Imyudf.h"
//************************************/
// To do:
// 入出力するオブジェクトを宣言してください。
//************************************/
static Imolecule molecule;
//************************************/
// 読み込みテスト
static void ValueInput(UDFhandle * udfin)
{
        UDFistream is(udfin);
        //************************************/
        // To do:
        // 入力するUDF オブジェクト名を指定してください。
        //************************************/
        is.open("molecule");
        int num;
        is >> num; if (num>0) is >> molecule;
        is.close();
        // 確認のため読み込んだデータを標準出力へ書き出す
        cout << "====== molecule =======" << endl;
        cout << molecule << endl;
}
// UDF 読み込みエントリー
void IObjectInputTest(const char *udfpath, const int restart_step)
{
        UDFhandle *udfin = OpenUDF(udfpath, restart_step);
        // レコード数取得
        size_t recnum = udfin->getTotalRecord();
        cout << "input file total record number:" << recnum << endl;
        ValueInput(udfin);
        CloseUDF(udfin);
}
// 計算結果を書き込む
// ここではインタフェースオブジェクト内容をそのまま書き出す
static void RecordOut(UDFhandle * udfout)
```

```
{
        UDFostream os(udfout);
        //*************************************/
        // To do:
        // 出力するUDF オブジェクト名を指定してください。
        //*************************************/
        os.open("molecule"); // オブジェクトを書き込むストリーム
        os << molecule;
        os.close();
}
void IObjectOutputTest(const char *udfpath, const int end_step)
{
        // 新規作成時は、登録するUDF 名称とローカルに存在するUDF 定義ファイルを指定する
        // 定義ファイル名を、インタフェースオブジェクトから取得する
        IUdfInformation udfinfo;
        UDFhandle *udfout = CreateUDF(udfpath, udfinfo.getDefinition().c_str());
        AppendUDF(udfout);
        for(int step= 0; step <= end_step; step++) {
                AppendUDF(udfout, step);
                cout << "==========" << udfpath << ":" << step << endl;
                RecordOut(udfout);
        }
        CloseUDF(udfout);
}
```

A.3.4　テストプログラムをビルドして、実行してみます。

以下のコマンドでテストプログラムをビルドします。

make testinterface

Makefile には、makeinterface での生成処理が記述されているので、myudf.udf を修正すれば自動的にImyudf.h

が更新されるようになっています。

以下のコマンドでテストプログラムを実行します。

testinterface -T3 -I myudf.udf -O testout2.udf -e2

指定したmyudf.udf のデータ内容が、標準出力に表示されます。ここで、-T3 はチュートリアル３を指定するためのオプションです。また、-O で指定したファイルに、-e で指定したレコードまでデータが出力されています。出力されたtestout3.udf を確認してみてください。

このように、makeinterface を使用すれば、UDF データの入出力についてほとんどプログラミングを行う必要がなくなります。

A.3.5　UDFObjectパッケージに準備されている多くのメソッドを利用して、iobject_test.cppの処理内容を変更してみるのも理解の助けになるでしょう。

メソッド仕様については、第5 章「パッケージインタフェース」5.5 UDFObject および5.6 IObject をご覧下さい。

以上で、makeinterface チュートリアルは終了です。

makeinterface はUDFの入出力だけではなく、フォーマット変換プログラムや、統計プログラムを自動生成することも可能です。ご活用ください。

付録 B　添付資料：UDF バージョンアップ対応機能の解説

「makeinterface V3.0」では、エンジン開発者が長期間に渡って UDF 入出力ファイルをメンテナンスする作業をサポートするための機能として、複数の異なるバージョンの UDF ファイル（通常少しずつデータ構造が変化していく）の差異を解析し、自動的にバージョンに合致した UDF ファイルを入出力できるインタフェースクラス定義を生成するようにした。

新たに追加した機能は以下のとおり、

(1) 複数の異なるバージョンの UDF ファイルを指定することができる。
(2) バージョン間のデータ定義の差異を解析し､バージョン分けされたインターフェースクラス定義を生成する。
(3) 配列でない基本データ型データにデフォルト値を設定する。

B.1　バージョン分岐の概要

複数バージョンのＵＤＦファイルの順序付けは、コマンドの-D オプションに指定されたＵＤＦ定義ファイルの順序に基づいて行う。この順序は実際のＵＤＦのバージョンアップ履歴と一致している必要がある。

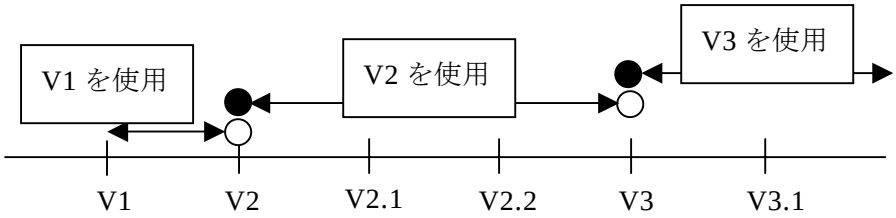
一方、バージョン分岐処理は、ヘッダーファイルの"EngineVersion"を利用する。

例えば、次のようなバージョンアップ履歴があるとする：

UDF バージョン （EngineVersion 内容）	変更内容
V1	最初のＵＤＦ定義ファイル
V2	ＵＤＦ定義の変更
V2.1	エンジンプログラムの改良のみでＵＤＦ定義の変更なし
V2.2	エンジンプログラムの改良のみでＵＤＦ定義の変更なし
V3	ＵＤＦ定義の変更
V3.1	エンジンプログラムの改良のみでＵＤＦ定義の変更なし

このような変更履歴のあるＵＤＦ定義ファイルがあるとき、本ツールに指定するＵＤＦ定義ファイルは、ＵＤＦ定義の変更があった V1,V2,V3 のＵＤＦ定義ファイルである。

各バージョンで使用されるＵＤＦ定義ファイルは以下のようになるはずである：



上図に示すＵＤＦ定義の使用方法を考慮して、本ツールでは、文字列の大小比較を使用するバージョン分岐処理を生成する。

以下の例は、データ名 a,b,c が下表に示すバージョンのＵＤＦファイルで使用されるとき、出力されるインターフェース定義を示している。

UDF バージョン	使用されるデータ名
"V1"	a,b
"V2"	b,c
"V3"	a,c

バージョン"V1"の例

```
¥begin{header}
¥begin{def}
EngineVersion:string;
¥end{def}
¥begin{data}
EngineVersion:"V1"
¥end{data}
¥end{header}
¥begin{def}
class x:{
  a:int
  b:double
}
¥end{def}
```

バージョン"V2"の例

```
¥begin{header}
¥begin{def}
EngineVersion:string;
¥end{def}
¥begin{data}
EngineVersion:"V2"
¥end{data}
¥end{header}
¥begin{def}
class x:{
  b:double
  c:string
}
```

```
¥end{def}
```

バージョン"V3"の例

```
¥begin{header}
¥begin{def}
EngineVersion:string;
¥end{def}
¥begin{data}
EngineVersion:"V3"
¥end{data}
¥end{header}
¥begin{def}
class x:{
   a:int
   c:string
}
¥end{def}
```

出力されるインターフェース定義

（分かりやすいように_Input メソッド内はテキスト入力部分のみを記述してある。）

```
#include "udfobject.h"

#define UDF_VERSION_00 "V1"
#define UDF_VERSION_01 "V2"
#define UDF_VERSION_02 "V3"

// UDF header informations.
extern string projectName;
extern string engineName;
extern string engineVersion;
extern string ioType;
extern string comment;
extern string action;
extern string currentVersion;
extern TAB tab;      // tab formatting object for ostream
#define _VERSION_STRING_ engineVersion

#if !defined(_IUdfInformation_H_)
class IUdfInformation {
public:
      IUdfInformation(const string& file="v3.udf") : deffile(file) {}
      // UDF File Information
      const string& getDefinition() { return deffile; }
private:
      string deffile;
};
#endif

class Ix;

class Ix : public UDFObject {
public:
```

```
    Ix() : UDFObject() {
        a=0;
        b=0.0;
    }
    virtual ~Ix() {}
    virtual const char* GetName() const { return "x"; }
    UDFint    a;
    UDFdouble    b;
    UDFstring    c;

public:
    virtual void _Input(UDFistream& is) {
        if(_VERSION_STRING_.compare(UDF_VERSION_01) < 0
        || _VERSION_STRING_.compare(UDF_VERSION_02) >= 0){
            is >> a;
        }
        if(_VERSION_STRING_.compare(UDF_VERSION_02) < 0){
            is >> b;
        }
        if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
            is >> c;
        }
    }
    virtual void _Output(ostream& os) {
        os << tab << "{";
        tab.enter();
        os << a << " ";
        os << c << " ";
        tab.exit();
        os << tab << "}¥n";
    }
};
```

データ"a" は不連続なバージョンで使用されるために、バージョン分岐に or 演算 ("||") を使用している。出力に用いる UDF 定義は最終バージョンを使用する。

B.2　バージョン解析の方法

以下では UDF ファイルのデータ構造が変更されるケース毎に、生成されるインターフェースクラス定義の違いについて説明する。

UDF は基本データ型（int,double,string 等）のほかに、構造体型（ユーザー定義型）を扱うことができる。基本データ型および構造体型について、データ構造を変更するケースは以下のようになるものと考えられる。

(1) 構造体内でデータの順序が変更される。

(2) トップレベルまたは構造体内で新たな名前でデータが追加される、あるいは削除される

(3) トップレベルまたは構造体内で既存のデータの型が変更される

(4) トップレベルまたは構造体内で既存のデータの名前が変更される

上記の４ケースが重なりあって新たなデータ構造が作成されていくと考えられる。

(1)[構造体内のデータ順序変更]

このケースでは、並びが同じデータを最大限使用するようにバージョン分岐処理を行うインターフェースクラスを生成する。

変更前のＵＤＦ定義例

```
¥begin{header}
¥begin{def}
EngineVersion:string;
¥end{def}
¥begin{data}
EngineVersion:"V0"
¥end{data}
¥end{header}
¥begin{def}
class a:{
   p:int
   q:int
   r:double
   s:string
}
¥end{def}
```

変更後のＵＤＦ定義例

```
¥begin{header}
¥begin{def}
EngineVersion:string;
¥end{def}
¥begin{data}
EngineVersion:"V1"
¥end{data}
¥end{header}
¥begin{def}
class a:{
   s:string
   r:double
   p:int
   q:int
}
¥end{def}
```

出力されるインターフェース定義

（分かりやすいように_Input メソッド内はテキスト入力部分のみを記述してある。）

```
#define UDF_VERSION_00 "V0"
#define UDF_VERSION_01 "V1"

// UDF header informations.
extern string engineVersion;
extern TAB tab;      // tab formatting object for ostream
#define _VERSION_STRING_ engineVersion

#if !defined(_IUdfInformation_H_)
class IUdfInformation {
public:
        IUdfInformation(const string& file="v1.udf") : deffile(file) {}

        // UDF File Information
        const string& getDefinition() { return deffile; }

private:
        string deffile;
};
#endif

class Ia;

class Ia : public UDFObject {
public:
        Ia() : UDFObject() {
                r=0.0;
        }
        virtual ~Ia() {}
        virtual const char* GetName() const { return "a"; }
        UDFstring           s;
        UDFdouble           r;
        UDFint    p;
        UDFint    q;

public:
        virtual void _Input(UDFistream& is) {
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        is >> s;
                        is >> r;
                }
                is >> p;
                is >> q;
                if(_VERSION_STRING_.compare(UDF_VERSION_01) < 0){
                        is >> r;
                        is >> s;
                }
        }
        virtual void _Output(ostream& os) {
                os << tab << "{";
                tab.enter();
```

```
                        if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                                os << s << " ";
                                os << r << " ";
                        }
                        os << p << " ";
                        os << q << " ";
                        if(_VERSION_STRING_.compare(UDF_VERSION_01) < 0){
                                os << r << " ";
                                os << s << " ";
                        }
                        tab.exit();
                        os << tab << "}¥n";
        }
};
```

(2)[データの追加あるいは削除]

このケースでは、データ入出力時の処理として、以下のように各バージョンの入出力時に対応してバージョン分岐処理を行うインターフェースクラスを生成する。

下記例では、最初の変更で DPD が追加され、次の変更で Lennard_Jones_EV が追加された３バージョンのＵＤＦ定義から出力されたインターフェースクラスの一部である。

```
virtual void _Input(UDFistream& is) {
        is >> Name;
        is >> Potential_Type;
        is >> Site1_Name;
        is >> Lennard_Jones;
        if(_VERSION_STRING_.compare(UDF_VERSION_02) >= 0){
                is >> Lennard_Jones_EV;
        }
        is >> Gay_Berne;
        is >> GB_LJ;
        if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                is >> DPD;
        }
        is >> Table_Pair_Potential;
        is >> User_Pair_Interaction;
}
```

(3) [データ型の変更]

このケースの場合は幾分複雑な状況が発生する。基本データの場合および構造体型（ユーザー定義型）に分けて説明する。

・基本データの場合

ある構造体の基本データ型属性データの型を変更した場合、C++のクラス内では同じデータ名で異なるタ

イプの変数を記述することができない。従って古い方の UDF ファイル（コマンド引数の-D オプションに指定する UDF 定義ファイルで前の方に指定した UDF ファイル）のデータ名を以下の例のように a.q -->> a.q_00 に変更したインターフェースクラスファイルを出力する。

変更前の UDF 定義

```
¥begin{header}
¥begin{def}
EngineVersion:string;
¥end{def}
¥begin{data}
EngineVersion:"V0"
¥end{data}
¥end{header}
¥begin{def}
class a:{
        q:int
}
¥end{def}
```

変更後の UDF 定義

```
¥begin{header}
¥begin{def}
EngineVersion:string;
¥end{def}
¥begin{data}
EngineVersion:"V1"
¥end{data}
¥end{header}
¥begin{def}
class a:{
        q:string
}
¥end{def}
```

生成されるインターフェースクラス定義は以下のようになる。

```
#define UDF_VERSION_00 "V0"
#define UDF_VERSION_01 "V1"

extern string engineVersion;
#define _VERSION_STRING_ engineVersion

class Ia;
class Ia : public UDFObject {
public:
        Ia() : UDFObject() {
                q_00=0;
        }
        virtual ~Ia() {}
        virtual const char* GetName() const { return "a"; }
        UDFstring               q;
```

```
        UDFint    q_00;

public:
        virtual void _Input(UDFistream& is) {
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        is >> q;
                }
                if(_VERSION_STRING_.compare(UDF_VERSION_01) < 0){
                        is >> q_00;
                }
        }
        virtual void _Output(ostream& os) {
                os << tab << "{";
                tab.enter();
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        os << q << " ";
                }
                if(_VERSION_STRING_.compare(UDF_VERSION_01) < 0){
                        os << q_00 << " ";
                }
                tab.exit();
                os << tab << "}¥n";
        }
};
```

ただし、整数系データ型（short,int,long,INDEX,INDEXREF）、実数系データ型（single,float,double）および文字列系データ型（string,select,KEY,KEYREF）のそれぞれの系内のデータ型変更ではデータ名の自動変更は引き起こされない。

名前変更は、コマンド引数の-D オプションに指定する UDF 定義ファイルの順序番号が付加される。

・ユーザー定義型データの場合

ユーザー定義型データのユーザー定義型名が変更される場合の処理は以下のようになる。

次の例のように　mesh:MeshNew -> mesh:Mesh などとユーザー定義型名が変更された場合、基本データ型と同様に簡単にデータ名が自動変更されると、ユーザーが扱わなければならないC++クラスの種類が増えてしまう。ユーザー定義型の場合、ユーザー定義型名が変わってもユーザー定義型内の属性データは変わっていない場合もありうる。そのため、ユーザー定義型名が変更された場合でも、その内容が同じ場合には、データ名を自動変更しないことにした。

ユーザー定義型名が変わり、その内容が異なる場合には、データ名は自動変更される。このような事態が不都合な場合は、ユーザーがあらかじめデータ名を変更しておくことを推奨する。

変更前の UDF 定義

```
Grid_Density:{
        mesh:MeshNew                                    "Information of mesh"
        boundary_condition:BoundaryConditionNew "Boundary condition"
        phi:ScalarField                                 "Scalar field data of phi"
} "Output of grid density calculated from the positions of atoms"
```

変更後の UDF 定義

```
Grid_Density:{
        mesh:Mesh                                   "Information of mesh"
        boundary_condition:BoundaryCondition "Boundary condition"
        atom_name[]:string                          "List of Atom name"
        phi:ScalarField                             "Scalar field data of phi"
} "Output of grid density calculated from the positions of atoms"
```

MeshNew と Mesh、BoundaryConditionNew と BoundaryCondition がそれぞれ同じ内容の属性をもつ場合に生成されるインターフェースクラス定義は以下のとおり：

```
class IGrid_Density : public UDFObject {
public:
        IGrid_Density() : UDFObject() {
        }
        virtual ~IGrid_Density() {}
        virtual const char* GetName() const { return "Grid_Density"; }
        IMesh     mesh;
        IBoundaryCondition          boundary_condition;
        UDFarray<UDFstring>         atom_name;
        IScalarField          phi;

public:
        virtual void _Input(UDFistream& is) {
                is >> mesh;
                is >> boundary_condition;
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        is >> atom_name;
                }
                is >> phi;
        }
        virtual void _Output(ostream& os) {
                os << tab << "{";
                tab.enter();
                os << mesh << " ";
                os << boundary_condition << " ";
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        os << atom_name << " ";
                }
                os << phi << " ";
                tab.exit();
                os << tab << "}¥n";
        }
};
```

もし、MeshNew と Mesh のユーザー定義型の属性が異なる場合には、以下のようなインターフェースクラス定義が生成される。

```
class IGrid_Density : public UDFObject {
public:
        IGrid_Density() : UDFObject() {
        }
```

```
        virtual ~IGrid_Density() {}
        virtual const char* GetName() const { return "Grid_Density"; }
        IMesh       mesh;
        IMeshNew            mesh_00;
        IBoundaryCondition          boundary_condition;
        UDFarray<UDFstring>         atom_name;
        IScalarField        phi;

public:
        virtual void _Input(UDFistream& is) {
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        is >> mesh;
                }
                if(_VERSION_STRING_.compare(UDF_VERSION_01) < 0){
                        is >> mesh_00;
                }
                is >> boundary_condition;
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        is >> atom_name;
                }
                is >> phi;
        }
        virtual void _Output(ostream& os) {
                os << tab << "{";
                tab.enter();
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        os << mesh << " ";
                }
                if(_VERSION_STRING_.compare(UDF_VERSION_01) < 0){
                        os << mesh_00 << " ";
                }
                os << boundary_condition << " ";
                if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                        os << atom_name << " ";
                }
                os << phi << " ";
                tab.exit();
                os << tab << "}¥n";
        }
};
```

(4)[データ名変更]

既存のデータの名前が変更された場合、それまであったデータが削除され、同じ位置・データ型で新たなデータが追加されたと解釈される。従って、ＵＤＦファイルのバージョン解析においては、(1)に帰着される。

しかし、データ名変更のケースによっては、名前は変更したがそれまでと全く同じように扱いたいという場合がある。その場合には、後述する「スクリプトファイルに追加された識別子」の「identify 識別子によるデータの同一視指定」を参照されたい。

B.3 使用方法

以下に新機能を搭載した makeintarface ツールの使用方法を述べる。

コマンド:

makeinterface -D udfdef_file1 [udfdef_file2 udfdef_file3 ...] -F [N]
[-N interface_name] [-I template_file]
[-O output_file] [-C class_prefix] [-A attribute_prefix]
[-S script_file]

引数:

-D udfdef_fileN: UDF 定義ファイル名

指定された UDF ファイルの def 部の情報を解析してインターフェースクラスを作成する。
複数の UDF ファイルを指定した場合、バージョン解析を行う。指定する UDF ファイルの順序はバージョン順であると仮定している。変数名の自動変更が発生するような時には後ろに指定する（新しい）UDF ファイルのデータ名が優先的に使用される。
指定する UDF 定義ファイルは¥include を含んでもよく、その場合カレントディレクトリに存在しなければ、環境変数 UDF_DEF_PATH で指定されたディレクトリを探す。

-F [N]: ＵＤＦ書き出しメソッド(_Output)のバージョン固定機能

ＵＤＦ書き出しメソッド(_Output)を、-D で指定されたＵＤＦファイルの内の１つに固定する。
N は-D で指定したＵＤＦファイルのインデックス番号（ゼロから始まる）である。
N を省略した場合は、最後のＵＤＦファイル定義に対応したＵＤＦ書き出しメソッドが生成される。

-N interface_name: インタフェース名称

%INTERFACE_NAME% マクロの置き換えとして使用される。このマクロは、ヘッダファイルのインクルード検査や common udf クラス名に用いられる。デフォルト名は、class_prefix + (udfdef file) の拡張子を除いた文字である。

-I template_file: テンプレートファイルパス名

指定されたテンプレートファイルのマクロ部分を置き換えてインターフェースクラスを作成する。template_file を編集することにより、生成内容を変更することができる。デフォルト名は、template.txt。テンプレートファイルが、実行時カレントディレクトリになければ、makeinterface モジュールのあるディレクトリが探索される。

-O output_file: 出力するヘッダファイルパス名

デフォルト名は、interface_name + ".h" である。

-C class_prefix: 生成するインターフェースクラス名に付けるプレフィックス

デフォルトは、"I"で IObject インタフェースの生成を行う。"U"を指定すると、UObject インタ

フェースの生成を行う。

-A attribute_prefix: 生成するメンバ属性名に付けるプレフィックス

ディフォルトは、なし。

-S script_file: 生成方法指定ファイル（スクリプトファイル）

script_file を編集することにより、生成方法を変更することができる。デフォルトは、script.txt。スクリプトファイルが、実行時カレントディレクトリになければ、makeinterface モジュールのあるディレクトリが探索される。

テンプレートファイルに追加されたマクロ

UDF バージョンアップ対応機能のために、新規に追加されたマクロは以下のとおり。

新マクロルール

$IF_VERSION_BRANCH_BEGIN$	$ATTRIBUTE_TEMPLATE_BEGIN$ ～ $ATTRIBUTE_TEMPLATE_END$ 内で複数行に渡ってバージョン分岐処理が必要な場合に使用する。 バージョン分岐の開始を示す。複数の UDF 定義ファイルが指定されていない時は、無視され、出力ファイルに現れない。
$IF_VERSION_BRANCH_END$	バージョン分岐の終了を示す。

新マクロ定義

%DEFINE_VERSION_LIST%	#define UDF_VERSION_NN "バージョン文字列" に変換される。指定した UDF 定義ファイル数行生成される。
%ATTRIBUTE_DEFAULT_LIST%	C/C++プログラムの代入文に置き換えられる。
%ATTRIBUTE_OUTPUT_LIST_WITHVERSION%	%ATTRIBUTE_OUTPUT_LIST%にバージョン分岐処理が追加される。

バージョン識別文字列

_VERSION_STRING_	バージョン分岐処理に使用する。 UDF ヘッダー情報の EngineVersion に設定されている文字列をもつ std::string 型の変数。

新マクロを用いたテンプレート例を以下に示す。

```
//           template version V3.0
//                                Generated by: %GENERATED_VERSION%
//                                Generated date: %GENERATED_DATE%
#ifndef _%INTERFACE_NAME%_H_
#define _%INTERFACE_NAME%_H_

#include "udfobject.h"
```

```
#include %INCLUDE_LIST%

%DEFINE_VERSION_LIST%

// UDF header informations.
extern string engineVersion;

extern TAB tab;      // tab formatting object for ostream
#define _VERSION_STRING_ engineVersion

#if !defined(_IUdfInformation_H_)
class IUdfInformation {
public:
        IUdfInformation(const string& file="%DEFINITION%") : deffile(file) {}
        // UDF File Information
        const string& getDefinition() { return deffile; }
private:
        string deffile;
};
#endif

class %CLASS_LIST%;

$CLASS_TEMPLATE_BEGIN$
class %CLASS_NAME% : public UDFObject {
public:
        %CLASS_NAME%() : UDFObject() {
                %ATTRIBUTE_DEFAULT_LIST%
        }
        virtual ~%CLASS_NAME%() {}
        virtual const char* GetName() const { return "%OBJECT_NAME%"; }
        %ATTRIBUTE_DEFINITION_LIST%

public:
        virtual void _Input(UDFistream& is) {
$ATTRIBUTE_TEMPLATE_BEGIN$
                $IF_VERSION_BRANCH_BEGIN$
$IF_NATIVE_TYPE$
#if defined(USE_TEXT_TEMP)
                is >> %ATTRIBUTE_NAME%;
#else
                is.read((char *)& %ATTRIBUTE_NAME%, sizeof(%ATTRIBUTE_NAME%));
#if defined(DEBUG)
                cout << %ATTRIBUTE_NAME% << " ";
#endif
#endif
$ELSE_OTHER_TYPE$
                is >> %ATTRIBUTE_NAME%;
$IF_TYPE_END$
                $IF_VERSION_BRANCH_END$
$ATTRIBUTE_TEMPLATE_END$
$IF_MAP_TYPE$
#if defined(USE_COMMON_SUPORT)
                if (getRoot())          getRoot()->addMAP(this);
```

```
#endif
$IF_TYPE_END$
        }
        virtual void _Output(ostream& os) {
                os << tab << "{";
                tab.enter();
                os << %ATTRIBUTE_OUTPUT_LIST_WITHVERSION% << " ";
                tab.exit();
                os << tab << "}¥n";
        }
};
$CLASS_TEMPLATE_END$
#endif
```

スクリプトファイルに追加された識別子

UDF バージョンアップ対応機能のために、新たに追加された識別子は以下のとおり。

識別子	パラメータ	機能
default	データ名 = デフォルト値	%ATTRIBUTE_DEFAULT_LIST% マクロが初期値指定文に置き換えられる。
identify	置き換えられるデータ名= 置き換え後データ名	1) バージョン解析時に置き換えられたデータ名が使用される。 2) C++インターフェースクラス定義ファイルのデータ名が置き換えられる。

default 識別子は UDF 定義のバージョンアップ変更によって配列ではないデータ定義が増減した場合、あるバージョンの UDF ファイルにおいてデータ入出力対象になっていないデータにデフォルト値を与えておくために必要である。デフォルト値指定が無いバージョン分岐処理対象データには、数値の場合：0、文字列の場合："" がそれぞれ設定される。配列データはサイズゼロの配列になるので、デフォルト値指定はできない。

identify 識別子は、データの位置および型は変更しないけれどもある事情により名前のみを変更する必要があるときに使用する。identify 識別子の使用例を以下に示す。

バージョンアップ前の UDF 定義の一部分

```
class NodeDensityBias:{
  Molecular_Name:string "Molecular name to apply node density biased Monte Carlo"
  UDF_Name:string       "UDF file name to read segment volume fraction"
  Start_ID:int          "Start ID of segment in SUSHI UDF"
  End_ID:int            "End ID of segment in SUSHI UDF"
  Scale_Param:double    "Scaling parameter: ¥n(number of atoms of MD)/(one segment of SCF)"
  Max_Retry:int         "Maximum number of retry"
} "Input for node density bias Monte Carlo"
```

バージョンアップ後の UDF 定義の一部分

```
class NodeDensityBias:{
   Molecular_Name:string "Molecular name to apply node density biased Monte Carlo"
   UDF_Name:string       "UDF file name to read segment volume fraction"
   Start_Index:int       "Start Index of segment in SUSHI UDF"
   End_Index:int         "End Index of segment in SUSHI UDF"
   Scale_Param:double    "Scaling parameter: ¥n(number of atoms of MD)/(one segment of SCF)"
   Max_Retry:int         "Maximum number of retry"
} "Input for node density bias Monte Carlo"
```

上記の２つの UDF から makeinterface を使ってインターフェースクラス定義ファイルを生成すると、データ項目の増減があったものとみなして、クラス宣言およびデータ出力部分として以下のプログラムが生成される。

クラス宣言部分

```
class INodeDensityBias : public UDFObject {
public:
        INodeDensityBias() : UDFObject() {
                Start_Index=0;
                End_Index=0;
                Start_ID=0;
                End_ID=0;
        }
        virtual ~INodeDensityBias() {}
        virtual const char* GetName() const { return "NodeDensityBias"; }
        UDFstring       Molecular_Name;
        UDFstring       UDF_Name;
        UDFint  Start_Index;
        UDFint  End_Index;
        UDFint  Start_ID;
        UDFint  End_ID;
        UDFdouble       Scale_Param;
        UDFint  Max_Retry;
        .....
```

データ出力部分部分

```
virtual void _Output(ostream& os) {
        os << tab << "{";
        tab.enter();
        os << Molecular_Name << " ";
        os << UDF_Name << " ";
        if(_VERSION_STRING_.compare(UDF_VERSION_01) >= 0){
                os << Start_Index << " ";
                os << End_Index << " ";
        }
        if(_VERSION_STRING_.compare(UDF_VERSION_01) < 0){
                os << Start_ID << " ";
                os << End_ID << " ";
        }
        os << Scale_Param << " ";
```

```
        os << Max_Retry << " ";
        tab.exit();
        os << tab << "}¥n";
}
```

自動生成ツールはデータの意味まで分からないので上記のプログラムを生成せざるを得ない。エンジン開発者が Start_ID を Start_Index に名前を変更しただけでデータ定義変更後も変更前と同じように扱いたい時、identify 識別子を以下のように指定すれば、次のように Start_Index を Start_ID と同一視したインターフェースクラス定義ファイルを生成することができる。

identify 識別子によるデータの同一視指定例

```
identify NodeDensityBias.Start_ID = NodeDensityBias.Start_Index
identify NodeDensityBias.End_ID = NodeDensityBias.End_Index
```

出力されたインターフェースクラス定義ファイルの一部

```
class INodeDensityBias : public UDFObject {
public:
        INodeDensityBias() : UDFObject() {
        }
        virtual ~INodeDensityBias() {}
        virtual const char* GetName() const { return "NodeDensityBias"; }
        UDFstring       Molecular_Name;
        UDFstring       UDF_Name;
        UDFint  Start_Index;
        UDFint  End_Index;
        UDFdouble       Scale_Param;
        UDFint  Max_Retry;
        .....
```

```
virtual void _Output(ostream& os) {
        os << tab << "{";
        tab.enter();
        os << Molecular_Name << " ";
        os << UDF_Name << " ";
        os << Start_Index << " ";
        os << End_Index << " ";
        os << Scale_Param << " ";
        os << Max_Retry << " ";
        tab.exit();
        os << tab << "}¥n";
}
```

新識別子を用いたスクリプトファイルの例を以下に示す。

```
// This file is default script for makeinterface tool in Platform utility.
// To Do:       bind IObject to user class

bind IVector3D Vector3d
```

```
bind IVector3d Vector3d
bind ISymMat3x3 SymMat3x3
bind ICell SymMat3x3
bind IEnergy_element EnergyTable
// etc.

// To Do:     include user class header
include "Vector3d.h"
include "symmat3x3.h"
include "energytable.h"
// etc..

top Simulation_Conditions
top Initial_Structure
top Molecular_Attributes
top Interactions
top React_Conditions
top Set_of_Molecules
top Steps
top Time
top Statistics_Data
top Structure
top Grid_Density
top Grid_Density1
top Unit_Parameter

default CellDeformation.Deform_Atom = 1
default CellDeformation.Method = ""
default dpd.lambda = 0.65
default dpdbond.C = 4.0

identify NodeDensityBias.Start_ID = NodeDensityBias.Start_Index
identify NodeDensityBias.End_ID = NodeDensityBias.End_Index
```

（注）デフォルト値設定時のインターフェースクラス定義ファイル

文字列型のデータについては、以下のように SetValue()を使ってデータをセットするインターフェースクラス定義ファイルが生成される.（2007 年以降）

```
    ICellDeformation() : UDFObject() {
        Method.SetValue("");
        Deform_Atom=1;
    }
```

付録C　添付資料：テキストUDF形式以外の出力

C.1　バイナリUDF形式の出力

インターフェースライブラリにおいてバイナリUDF形式の書き出しを指定するために下記のメソッドが追加されています。

UDFhandleクラス（udfhandle.h）

```
bool setBinaryMode();
bool isBinaryMode();
```

UDFManagerクラス（udfmanager.h）

```
bool setBinaryMode();
bool isBinaryMode();
```

また、UDFManagerおよびIObjectインターフェースライブラリをリンクしたエンジンは、出力ファイルの拡張子を、".bdf"にするとバイナリUDF形式で書き出すようになります。

C.2　分割レコードUDF形式の出力

インターフェースライブラリにおいて分割レコードUDF形式の書き出しを指定するために下記のメソッドが追加されています。

UDFhandleクラス（udfhandle.h）

```
bool setDivideMode();
bool isDivideMode();
```

UDFManagerクラス（udfmanager.h）

```
bool setDivideMode();
bool isDivideMode();
```

また、UDFManagerおよびIObjectインターフェースライブラリをリンクしたエンジン

は、出力ファイルの拡張子を、"duf"にすると分割レコード UDF 形式で書き出すようになります。