HiBase API の 概要...

HiBaseAPI Overview...

HiBase へようこそ...

このドキュメントは、アプリケーションプログラムから ***HiBase*** のデータベースを操作する時に利用する、アプリケーションプログラムインタフェース（HiBase API）の詳細（設計とその使用法）を説明します。

各内容の説明を行なう前に、この章では、***HiBase*** APIの概要と、本ドキュメントおよび、サンプルプログラム中での ***HiBase*** APIの表記、および、HiBase の使うデータ型について解説します。

この章で説明するのは、次の項目です。

- ◇ ***HiBase*** APIの概略
- ◇ ***HiBase*** APIのクラス／データ型などの表記

HiBase の クラスライブラリ

HiBase は、Holon の開発した「リレーショナルデータベース・管理システム」です。***HiBase*** を使用することで、「イントラネット」をはじめとした各種「データベース・システム」、Client/Server 環境での「グループウェア」の運用、及び、その開発を容易に行うことができます。

HiBase は、開発言語として「C++」を利用しています。そして、***HiBase*** のデータベースを操作する全ての機能を、「クラス」とその「メンバ関数」で表現し、これらを「クラスライブラリ」の形にまとめています。つまり、***HiBase*** を利用するアプリケーションプログラムからは、**「*HiBase* ＝クラスライブラリ」**と見ることができます。

この「***HiBase*** ＝クラスライブラリ」は、百を超える「クラス」と数千の「メンバ関数」が定義されている大規模なものです。ただし、アプリケーションプログラムが直接利用する部分はある程度限定されていることから、***HiBase*** は、アプリケーションプログラムから ***HiBase*** のデータベースを操作する時に利用する「クラス」とその「メンバ関数」を「***HiBase*** API」と定義しています。

本ドキュメントは、この「***HiBase*** API（アプリケーションプログラムインタフェース)」を説明するものです。

以下に、「***HiBase***／クラスライブラリ」の構造を示します。

HiBase API

「***HiBase*** API」は、***HiBase*** の「データベース」を操作するための全ての機能を覆っています。

具体的には、「HBoss クラス」で ***HiBase*** との接続を行ない、「HDBHandle クラス」が、HBossを中継する形で、トランザクション処理などの基本機能を提供します。

その他のサービスクラス

先に述べた通り、「***HiBase*** ＝クラスライブラリ」は、百を超える「クラス」と数千の「メンバ関数」が定義されている大規模なもので、「***HiBase*** API」の他にも、多数の「クラス」や「メンバ関数」が存在します。

本ドキュメントでは、***HiBase*** APIに加え、本ドキュメントおよび、サンプルプログラム中で利用している、「クラス」や「メンバ関数」を「サービスクラス（HDir/ HFile/ HStream クラス）」として追加説明しています。

「HDir（ディレクトリを表現するクラス）」、「HFile（ファイルを表現するクラス）」は、***HiBase*** が利用するディスクエレメントを抽象化し、物理的なアクセスを実行するクラスです。
これらのクラスには、「ディレクトリ（フォルダ）」や「ファイル」を操作する「メンバ関数」、及び、***HiBase*** の運用プログラム等を開発する際、***HiBase*** の利用環境の構築に利用される「スタティックメンバ関数」等が定義されています。

また、「HStream （ファイル／メモリを ストリームに 抽象化するクラス）」は、***HiBase*** が利用する「メモリ」を「ストリーム」に抽象化して表現する「基本クラス」です。
ストリーム操作を理解することで、プログラム中での、ファイル、動的メモリ（ハンドル、ポインタメモリ）、静的メモリ（スタック内メモリ）等の効率的な管理が可能となります。

HiBase API の表記 および メンバデータについて ...

HiBase Conventions...

以下、本ドキュメントおよび、サンプルプログラム中での ***HiBase*** APIの表記と、***HiBase*** APIのメンバデータについて説明します。

これらには、***HiBase*** の標準的な規約が含まれていますので、***HiBase*** のコードが理解しやすくなり、コードに一貫性を保つことができます。

ここでは、次の項目の命名およびコーディングの標準的な規約を解説します。

- ◇ クラスの名前、および、ヘダーファイルの扱い
- ◇ データ型
- ◇ その他のメンバデータ

クラスの名前

HiBase のクラス名は、大文字の「H」で始まります。以降、そのクラスの種類を表現する名前が付いています。

例：

クラス名	クラスの種類
HDBHandle	データベースへのアクセスを行なうクラス （データベース接続／管理、DD/D、DML関連）

本ドキュメントは、各クラス利用時に必要な「ヘダーファイル」を随時解説しています。

データ型

HiBase は移植性を高める目的で標準のCのデータ型も含めた、専用のデータ型を宣言しています。

HiBase は次のデータ型を使用します。

HiBase のデータ型	対応するC の型
JByte	char
UByte	unsigined char
JWord	short
UWord	unsigined short
JLong	long
ULong	unsigined long

その他のデータ型

整数型の類似形のデータ型の定義は、以下のとおりです。

種類	用途（実際の型）
JBool	2値（true / false）
JState	3値（true / false / undefine）
ECode	error code（short）
DNbr	データベース番号（long）
FNbr	ファイル番号（long）
BNb	ブロック番号（long）
RNbr	レコード番号（long）
VNbr	バリュー番号（long）
PNb	パケット番号（long）
SetID	集合番号（long）

ErrorCode 型（エラーコード）の詳細については、巻末の附録を参照してください。
また、***HiBase*** APIのメンバデータ（DD/D関連のデータ構造／検索条件式データ構造など）については、各々関連する章の解説を参照してください。

なお、以上の内容は、「HiTypes.h」、「HiECode.h」に定義されています。

この章では、「***HiBase*** API」の「クラス」と各「メンバ関数」について説明します。

「***HiBase*** API」は、***HiBase*** のデータベースを操作するための機能を提供するもので、***HiBase*** データベースの機能全てをを覆っています。「***HiBase*** API」を理解すれば、***HiBase*** の全般的なな理解ができます。

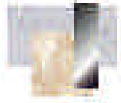
この章で解説する主要な項目は、次の通りです。

HBoss クラス　-　***HiBase*** への接続

　ソケットの準備

HDBHandle クラス　-　***HiBase*** にアクセス

　HiBase の開始／終了
　データベースの管理
　データベースファイルの管理（DD/D 関連）
　トランザクション処理（DML 関連）



HiBase APIについて ...

「***HiBase*** API」は、***HiBase***を操作するための基本的な機能を提供するもので、HiBase開発環境の最も重要なパーツです。

具体的には、以下のような「クラス」が用意されています。

HBoss クラス　- ***HiBase*** への接続

HiBaseを利用するアプリケーションプログラムは、まず最初に「データベースソケット」を生成します。「データベースソケット」とは、「HiBaseAPI」と「HiBase本体」を結合するために設けられた３つのクラスです。これら３つのクラスは、同じHBossクラスから派生する継承クラスで、
HBSBoss.....シングルスレッドで実行するHiBaseを起動する
HBMBoss....マルチスレッドで実行するHiBaseを起動する
HXBoss........HiBaseサーバーに接続する
があります。
アプリケーションプログラムは、実行環境に応じて上記のなかから適切な「データベースソケット」を選択して生成します。

HDBHandle クラス　- ***HiBase*** にアクセス

「HDBHandle クラス」は、***HiBase***のデータベースを操作するためのいわば「HiBaseAPIの実体」といえます。***HiBase***を利用するアプリケーションプログラムは、「HDBHandleクラス」を生成し、その後、目的に応じたメンバ関数を呼び出し、最後に「HDBHandleクラス」を削除して、アプリケーションプログラムを終了するといったスタイルになります。
「HDBHandleクラス」の機能は、データベースの定義情報情報を扱う「DD/D関連」機能、データを扱う「DML関連」機能に大別できます。

DD/ D 関連機能

HiBaseのデータベースは、「データベース」、「データベースファイル」、「レコード」、「項目」、「キー」という構成要素からなり、それぞれ定義情報が存在します。

「DD/D関連」機能は、これら「定義情報」をHiBaseで管理する機能、言い換えるとデータベースをデータベースで管理する機能といえます。

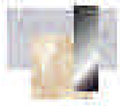
DML 関連

「DML 関連」のAPIは、アプリケーションプログラム（の利用者）のデータを、レコードの形で ***HiBase*** のデータベースに出し入れする機能（トランザクション処理）を提供します。

具体的には、データベースファイル内の「レコード」の追加／変更／読み出し／削除／検索等を行うメンバ関数が用意されています。

データベースを利用するアプリケーションプログラムは、頻繁に「検索」をおこなうのが普通です。つまり、検索は、データベースを利用するアプリケーションプログラムの「ボトルネック処理」といえます。***HiBase*** の大きな特長は、データベース処理の中心とも言える、このデータ検索に「集合」という概念を利用している点です。

HiBase は、レコードを構成している「項目」を「キー」に指定して高速な検索をおこないます。更に、検索された「レコード群」に識別子を付けて「集合」として保持し、後続の検索に再利用します。つまり、***HiBase*** は、実際に検索を行わずとも、集合同士の演算や「検索条件」から、新たな「集合= 検索結果」を得ることができるといった、いわば「検索処理」のキャッシュ化を行うことで、データベース処理の高速化を図る設計になっています。



1-1. HBoss クラス

- ***HiBase*** への接続

HiBase を利用するアプリケーションプログラムは、まず最初に「データベースソケット」を生成します。「データベースソケット」は、***HiBase*** の利用法によって、３種類に分れています。

1-1-1.データベースソケットの生成

アプリケーションプログラムが ***HiBase*** を利用する場合、「シングルモード/シングルスレッドモード」、「シングルモード/マルチスレッドモード」または「マルチモード」のいずれかの「利用モード」を決定します。

「シングルモード/シングルスレッド」は、アプリケーションプログラムにシングルスレッドHiBaseをリンクする「利用モード」です。
「シングルモード/マルチスレッド」は、アプリケーションプログラムにマルチスレッドHiBaseをリンクする「利用モード」です。
また、「マルチモード」は、アプリケーションプログラムをHiBase サーバーにネットワークを経由して接続する「利用モード」です。
上記のうち、普通使われるのは、「シングルモード/シングルスレッド」、もしくは、「マルチモード」です。しかし、サーバーとして働くアプリケーションプログラムは「シングルモード/マルチスレッド」を選択することになります。これは、マルチスレッドデータベースを抱えた高度なサーバーアプリケーションプログラムの開発を意味します。

利用モードの決定は、生成する「データベースソケット」を決定します。次に、「利用モード」に応じた「データベースソケット」の生成を説明します。

１-１-１-１．コンストラクタ

〔シングルモード /シングルスレッド〕

HBSBoss:: #include <HBSBoss.h>

HBSBoss(const JByte* thePath = "HiBase.conf");

〔シングルモード /マルチスレッド〕

HBMBoss:: #include <HBMBoss.h>

HBMBoss(const JByte* thePath = "HiBase.conf");

〔マルチモード〕

HXBoss:: #include <HXBoss.h>

HXBoss(const JByte* thePath = "HiBase.conf");

【説明】

各「利用モード」に応じた「データベースソケット」を生成する。

【パラメータ】

const JByte* thePath = "HiBase.conf"

thePath に、「環境ファイル」へのパスを指定します。

省略時は、デフォルト値（カレントディレクトリのHiBase.conf ）が指定されます。また、パスを指定する規則はUNIX のパス指定文法にしたがい、ディレクトリをデリミッタ ” / ” で区切って指定します。

以下は、***HiBase*** の環境ファイルの例です。

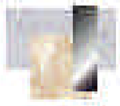
【環境ファイルのサンプル】

```
DBPATH    = "DBSample"
WORKPATH  = "DBTemp"
CASHSIZE  = 300
DDD       = on
SECURITY  = on
VALUE     = on
TLOG      = off
CLOG      = on
READONLY  = off

SERVER    = “hibase.hln.co.jp”
PORT      = 3330
```

【参照】

◇　HDBHandleクラス;　コンストラクタ



1-2. HDBHandle クラス - ***HiBase*** にアクセス

「HDBHandle クラス」は、***HiBase*** のデータベースを操作するための「HiBaseAPI」で、***HiBase*** の全機能を覆っています。***HiBase*** を利用するアプリケーションプログラムは、最初に「HDBHandleクラス」のインスタンス（オブジェクト）を生成し、目的に応じたメンバ関数を呼び出し、最後に、このインスタンスを削除して、アプリケーションプログラムを終了するといったスタイルを取ります。

「HDBHandle クラス」は、***HiBase*** との接続に関連するAPI、データベースの生成に関連するAPI、データベースの定義を操作する「DD/D 関連」のAPI、及び、データベースファイル内のデータを操作する「 DML 関連」のAPI に大別することができます。

メンバ関数一覧

Create	データベースの生成（特殊用途）
Erase	データベースの削除（特殊用途）
DBD_GetInfo	データベース情報の取得
Open	データベース開始宣言
Close	データベース終了宣言
Flush	キャッシュの書きだし（特殊用途）
CreateDBFile	データベースファイルの生成
EraseDBFile	データベースファイルの削除
GetDBSchema	データベーススキーマを読む
GetDBFileSchema	データベースファイルのスキーマを読む
TRCommit	トランザクションの開始/終了（更新の保証）
TRCancel	トランザクションの開始/終了（更新の取り消し）
CreateKey	データベースファイルのスキーマにキーを追加
DeleteKey	データベースファイルのスキーマからキーを削除
MaxRNbr	データベースファイルの最大発番レコード番号を取得
FileSize	データベースファイルのレコード総数を取得
GetRecord	レコードの読み込み
InsRecord	レコードの追加
DelRecord	レコードの削除
UpdRecord	レコードの更新

SetMake	集合の作成（=レコードの検索）
SetCancel	集合の削除
SetSort	集合のソート
SortCancel	ソートの取り消し
SetSize	集合のサイズ（=検索で見つかったレコード数）
SetRGet	集合内のレコードの読み込み
SetRAdd	集合にレコードを追加
SetRDel	集合からレコードを削除
Info_Set	集合情報（サイズ、条件式）を取得
Info_SetNof	HiBase内の集合の総数を取得
KSLocate	キー順読み込みの開始位置と開始宣言
KSRead	キー順読み込み
KSCancel	キー順読み込みの終了
PSLocate	高速順次読み込みの開始宣言
PSRead	高速順次読み込み
PSCancel	高速順次読み込みの終了
DBU_Insert	利用者アカウントの追加
DBU_Delete	利用者アカウントの削除
DBU_Get	利用者アカウントの読み込み

１-２-１-１．コンストラクタ

HDBHandle::

#include <HDBHandle.h>
#include <HBoss.h>

HDBHandle (HBoss* theBoss);

【説明】

「***HiBase*** API（HDBHandleクラス）」のインスタンス（オブジェクト）を生成します。

【パラメータ】

HDBHandleを生成する際に渡すパラメータは、利用モードに応じて以下の３つクラスから選択します。

HBSBoss	シングルモード/シングルスレッド
HBMBoss	シングルモード/マルチスレッド
HXBoss	マルチモード

【例】

【シングルモードでシングルスレッドHiBaseにリンク】

```
HBSBoss*     theBoss= new HBSBoss(“hibase.conf”);
HDBHandle*   theHandle = new HDBHandle(theBoss);
```

【シングルモードでマルチスレッドHiBaseにリンク】

```
HBMBoss*     theBoss= new HBMBoss(“hibase.conf”);
HDBHandle*   theHandle = new HDBHandle(theBoss);
```

【マルチモードでHiBaseサーバにネットワークを経てリンク】

```
HXBoss*      theBoss= new HXBoss(“hibase.conf”);
HDBHandle*   mDBHandle = new HDBHandle(mSocket, fileNbr);
```

【参照】

◇　HBossクラス;　コンストラクタ

１-２-１. *HiBase* の開始／終了

HiBase を利用するアプリケーションプログラムは､最初にOpen 関数を呼び出してデータベースアクセスの開始宣言をし、最後にClose 関数を呼び出して終了宣言をします。

１-２-１-２. データベースアクセスの開始／終了

〔データベースアクセスの開始〕

HDBHandle:: Open #include <HDBHandle.h>

ECode Open(const JByte* uid, const JByte* pwd);

【説明】

HiBase のデータベースアクセスを開始します。

本関数は、データベースの使用開始宣言であるため、通常、アプリケーションプログラムの最初に、一度だけ実行します。この呼び出しの時HiBase はデータベースに登録されているユーザアカウント情報を利用して認証を実施し、認証に合格したユーザだけがHiBase のアクセスに参加します。合格しなかったユーザにはエラーコードが返され、以降全ての要求がエラーになります。

【パラメータ】

uid : ユーザ ID（デフォルトは GUEST）へのポインタ

pwd: パスワード（デフォルトはブランク）へのポインタ

【例】

以下に、***HiBase*** 開始のサンプルプログラムを示します。

```
// HDBHandle (HiBase API) をつくる
HDBHandle* theHandle = new HDBHandle(theBoss);
// データベースアクセスを開始する
ec = theHandle->Open(“HOLON”);
```

〔 データベースアクセスの終了 〕

HDBHandle:: Close #include <HDBHandle.h>

ECode Close();

【 説 明 】

HiBase のデータベースアクセスを終了します。

本関数は、データベースの使用終了宣言であるため、通常、アプリケーションプログラムの最後に、一度だけ実行します。

1 - 2 - 2 . データベースの管理

アプリケーションプログラムが ***HiBase*** の「データベース」を生成／削除する場合は、Create／Erase 関数を呼び出します。

また、既に「データベース」が生成されているかを確認する場合は、DBD_GetInfo関数を利用して、データベース情報を取得します。

1 - 2 - 2 - 1 . データベースの生成／削除

〔 データベースの生成 〕

HDBHandle:: Create #include <HDBHandle.h>

ECode Create(const JByte* name, const JByte* desc);

【 説 明 】

HiBase の「データベース」を生成します。

【 パラメータ 】

name とdescで、生成するデータベースに付ける名前とメモを指定します。

【参照】

◇ DBD_GetInfo

〔データベースの削除〕

HDBHandle:: Erase #include <HDBHandle.h>

ECode Erase();

【説明】

HiBase の「データベース」を削除します。

【参照】

◇ DBGetInfo

【重要事項および使用制限】

HDBHandle::Create HDBHandle::Erase は、データベースの生成と削除を行います。データベースは、普通、組織の共有情報を管理する性格を持つため、本関数は非常に危険な性格を持っています。ご利用の際は、本関数の「特殊で危険」という性格を認識して、充分な注意を払ってください。
なお、本関数は、シングルモード / シングルスレッド（「データベースソケット」に HBSBoss を選択した場合）でのみ有効に機能します。シングルモード / マルチスレッドやマルチモードで利用する場合は、HiBase からエラーコードが返されます。これは HiBase の使用制限としております。

１-２-２-２．データベース情報の取得

〔データベース情報の取得〕

HDBHandle:: DBD_GetInfo #include <HDBHandle.h>

ECode　DBD_GetInfo（JByte* name, JByte* desc, JLong& n, FNbr* fList);

【説明】

HiBase の「データベース情報」を返します。

【パラメータ】

name - データベース名称、desc - メモ、n - 定義されているデータベースファイル数、fList - 定義されているデータベースファイル番号のリストが、HiBase から返されます。なお、fList に NULL を指定すれば、データベースファイル番号のリストが返されません。

1-2-3. データベースファイルの管理 （DD/D 関連）

HiBase の「データベース」に「データベースファイル」を追加する場合、HDBHandle::CreateDBFileを利用します。この時、「データベースファイルの定義情報（ファイルスキーマ）」が必要です。これは、データベースファイルの名前、識別子〔ファイル番号)、含まれる項目の一覧、含まれるキー項目の一覧、を指定する一連の記述で、これらの記述をHSchemaクラスに格納し、HDBHandle::CreateDBFileを呼びだします。

反対に、既に生成されている「データベースファイル」を削除する場合は、HDBHandle::EraseDBFile 関数を利用します。

1-2-3-1. データベースファイルの生成

HDBHandle:: CreateDBFile #include <HDBHandle.h>

ECode CreateDBFile(HSchema* theSchema);

【説明】

HiBase のデータベース内に、theSchemaで示すデータベースファイル（複数の指定が可能）を生成します。生成するデータベースファイルの定義はtheSchemaに指定します。

【パラメータ】

HSchema* theSchema;

HSchemaクラスは、データベース定義情報のコンテナ（格納庫）クラスとして存在します。このHSchemaクラスに一つもしくは複数のデータベースファイル記述を格納し、CreateDBFile に渡します。

【参照】

◇ HSchema クラス

【例】

【プログラム】... データベースファイルの生成

```
HBSBoss*   theBoss = new HBSBoss("../HiBase.conf");
HDBHandle* theHandle = new HDBHandle(theBoss);
ECode      ec = theHandle->Open("M.Nishibayashi", "mizuo-n");
if (ec == ecNormal) {
     HFile*     theFile = new HFile(NULL, "../HiBase.sch");
     HSchema*   dbSchema = HSchema::MakeSchema(theFile);
     ec = theHandle->CreateDBFile(dbSchema);
     delete theFile;
     delete dbSchema;
     theHandle->Close();
}
delete theHandle;
delete theBoss;
```

【スキーマ記述ファイル（DDL)】....../HiBase.sch ファイルの内容

```
[FILE:2], NAME = "TEST-FILE#2", DESC = "HiBaseの学習用ファイル２", BLOCKSIZE = (2048, 4096)
[ITEM:1], NAME = "ファイル名", DESC = "ファイル名称", TYPE = String, LENGTH = 16
[ITEM:2], NAME = "ファイル日付", DESC = "ファイル日付", TYPE = Date, LENGTH = 8
[ITEM:3], NAME = "登録日付", DESC = "データベースへの登録日", TYPE = Date, LENGTH = 8
[ITEM:4], NAME = "サイン", DESC = "ファイルクリエータ", TYPE = String, LENGTH = 4
[ITEM:5], NAME = "タイプ", DESC = "ファイルタイプ", TYPE = String, LENGTH = 4
[ITEM:6], NAME = "内容", DESC = "ファイル本体", TYPE = Text, LENGTH = 32000
[KEY:1], NAME = "ファイル名", DESC = "ファイル名称", TYPE = String, BIND = [1]
[KEY:2], NAME = "ファイル日付", DESC = "ファイル日付", TYPE = Date, BIND = [2]
[KEY:3], NAME = "登録日付", DESC = "データベースへの登録日", TYPE = Date, BIND = [3]
[KEY:4], NAME = "サイン", DESC = "ファイルクリエータ", TYPE = String, BIND = [4]
[KEY:5], NAME = "タイプ", DESC = "ファイルタイプ", TYPE = String, BIND = [5]
```

1 - 2 - 3 -1. データベースファイルの削除

HDBHandle:: EraseDBFile #include <HDBHandle.h>

ECode EraseDBFile(FNbr fNbr);

【説 明】

HiBase のデータベースから、fNbr で示す「データベースファイル」を削除します。

【パラメータ】

FNbr fNbr

削除すべきデータベースファイルの番号を指定します。

1-2-4. トランザクション処理（DML 関連）

「***HiBase*** API」のDML 関連のメンバ関数は、アプリケーションプログラム（の利用者）のデータを、レコードの形で ***HiBase*** のデータベースに出し入れする機能（トランザクション処理）を提供します。アプリケーションプログラムが「トランザクション処理」を行なう場合、通常は、本APIと「スキーマ変換クラス(HRecord)」を組み合わせて利用します。

一般的に、アプリケーションプログラムがデータベースファイル内にレコードを追加する場合、「スキーマ変換クラス（HRecord)」を使ってレコードを準備し、HDBHandle::InsRecord 関数を呼び出します。

また、レコードの読み出し／更新／削除などを行なう場合は、「SetMakeメンバ関数」を使って「集合」を作成し、SetRGet 関数で検索された集合内のレコード番号を獲得後、GetRecord／UpdRecord／DelRecordの各関数でレコードの操作（読み出し／更新／削除）をおこないます。

HiBase は、レコードを構成している「項目」を「キー」に指定して検索を行い、検索された「レコード群」を「集合」として扱うことで、処理の高速化を計っています。この集合の操作をおこなうために、一連の「Set関数群」が「***HiBase*** API」に用意されています。

更に、「***HiBase*** API」は、全データを対象とする「一括処理」を想定し、「キーシーケンシャルアクセス」と「高速シーケンシャルアクセス」機能があり、キーの値の順番に（キーシーケンシャルアクセス）もしくは物理的な格納の順番に（高速シーケンシャルアクセス)、データベース内の全てのデータを順次読み出すことが出来ます。

1-2-4-1. トランザクションの確立／キャンセル

〔トランザクションの確立〕

HDBHandle:: TRCommit　　#include <HDBHandle.h>

ECode　TRCommit();

【説明】

HiBase のデータベースの「トランザクション」を確立します。

〔トランザクションのキャンセル〕

HDBHandle:: TRCancel #include <HDBHandle.h>

ECode TRCancel();

【説明】

HiBase のデータベースの「トランザクション」をキャンセルします。

1-2-4-2．キーの追加／削除

〔キーの追加〕

HDBHandle:: CreateKey #include <HDBHandle.h>

ECode CreateKey(FNbr fNbr, JWord kid);

【説明】

fNbr で示すデータベースファイル内に、kid で示す「キー」を追加します。

【パラメータ】

FNbr fNbr

fNbr に、追加する「データベースファイル番号」（1～64000）を指定します。

JWord kid

kidに、追加する「キー番号」（1～250）を指定します。

〔キーの削除〕

HDBHandle:: DeleteKey #include <HDBHandle.h>

ECode　DeleteKey(FNbr fNbr, JWord kid);

【説明】

fNbr で示すデータベースファイルから、kid で示す「キー」を削除します。

【パラメータ】

FNbr fNbr

fNbr に、削除する「データベースファイル番号」（1～64000）を指定します。

JWord kid

kidに、削除する「キー番号」（1～250）を指定します。

１-２-４-３．データベースファイルの情報取得

〔レコード発番数の取得〕

HDBHandle:: MaxRNbr #include <HDBHandle.h>

ECode MaxRNbr (FNbr fNbr, RNbr* rNbr);

【説明】

fNbr で示すデータベースファイルの、「レコード発番数」を取得します。

HiBase のデータベースファイルの「レコード」は、データベースファイル内でユニークな番号（レコード ID）で管理されています。

本関数は、データベースが発番した「レコード ID」の最新の番号（最大番号）を返すものです。

【パラメータ】

FNbr fNbr

fNbr に、レコード発番数を取得する「データベースファイル番号」（1～64000）を指定します。

【戻り値】

RNbr* rNbr

rNbrに、fNbr で示すデータベースファイルが、現在までに発行した「レコード ID」の最大番号が返されます。

〔レコード総数の取得〕

HDBHandle:: FileSize #include <HDAP.h> / #include <CDAP.h>

ECode FileSize(FNbr fNbr, JLong* rQty);

【説明】

fNbr で示すデータベースファイルの、「レコード総数」を取得します。

【パラメータ】

FNbr fNbr

fNbr に、レコード総数を取得する「データベースファイル番号」（1～64000）を指定します。

【戻り値】

JLong* rQty

rQtyに、fNbr で示すデータベースファイル内で管理されている「レコード」の総数が返されます。

1-2-4-4. レコード操作

〔レコードの直接読み出し〕

HDBHandle:: GetRecord #include <HDBHandle.h>>

```
ECode  GetRecord(FNbr fNbr, RNbr  rNbr,
                          HRecord* oRec);
```

【説明】

fNbr で示すデータベースファイルから、rNbr で示す「レコード」を読み出します。

読み出されたレコードは、通常「スキーマ変換クラス（HRecord)」を利用して編集します。

「スキーマ変換」についての詳細は、後の『API 関連／HRecordクラス』を参照してください。

【パラメータ】

FNbr fNbr

fNbr に、読み出す「データベースファイル番号」(1〜64000）を指定します。

RNbr rNbr

rNbrに、読み出す「レコード番号」を指定します。

【戻り値】

HRecord* oRec

oRecに、fNbr で示すデータベースファイルから読み出されたレコードが格納されます。

【参照】

◇ HRecordクラス; コンストラクタ／Fetch 関数ファミリー

〔レコードの追加〕

HDBHandle:: InsRecord #include <HDBHandle.h>>

ECode　InsRecord(FNbr fNbr, RNbr& rNbr, const HRecord* iRec);

【説明】

fNbr で示すデータベースファイル内に、iRec で示す「レコード」を追加します。

通常、アプリケーションプログラムがデータベースファイル内に「レコード」を追加する場合は、本関数の実行に先立って、「スキーマ変換クラス（HRecord）」で編集した「レコード」を準備しておきます。

【パラメータ】

FNbr fNbr

fNbr に、追加する「データベースファイル番号」（1～64000）を指定します。

const HRecord* iRec

iRec に、「スキーマ変換クラス（HRecord）」を利用して編集された「レコード」を指定します。

「スキーマ変換」についての詳細は、後の『API 関連／HRecordクラス』を参照してください。

【戻り値】

RNbr& rNbr

rNbrに、fNbr で示すデータベースファイルにレコードを追加した際に付けられた「レコード番号」が返されされます。

【 例 】

以下に、レコード追加のサンプルプログラムを示します。

【 レコードの追加 】

```
HRecord theRec;                         // スキーマクラスを準備
long nbr = 1;
char aItem[32], bItem[32], cItem[32];
RNbr rNbr;                              // レコード番号
ECode ec;                               // エラーコード

theRec.Clear();                         // レコードを初期化

        //  項目１を long integerで設定
theRec.Append("[1] %4i", &nbr);
sprintf(aItem, "A%05ld", nbr);

        //  項目２を文字列（c 文字列）で設定
theRec.Append("[2] %sn", aItem);
sprintf(bItem, "B%05ld", nbr);

        //  項目３を文字列（c 文字列）で設定
theRec.Append("[3] %sn", bItem);
sprintf(cItem, "C%05ld", nbr);

        //  項目４を文字列（c 文字列）で設定
theRec.Append("[4] %sn", cItem);

ec = hdl->InsRecord(file, rNbr, &theRec);          // レコードを追加
```

【 参 照 】

◇　HRecordクラス;　コンストラクタ／Append 関数ファミリー

〔レコードの削除〕

HDBHandle:: DelRecord #include <HDBHandle.h>>

ECode DelRecord(FNbr fNbr, RNbr rNbr);

【説明】

fNbr で示すデータベースファイルから、rNbr で示す「レコード」を削除します。

アプリケーションプログラムがデータベースファイルから「レコード」を削除する場合、一般的に、「Set関数群」と本関数を組み合わせて使うのが普通です。

データ検索についての詳細は、後の『集合の操作』を参照してください。

【パラメータ】

FNbr fNbr

fNbr に、削除する「データベースファイル番号」（1～64000）を指定します。

RNbr rNbr

rNbrに、削除する「レコード番号」を指定します。

【例】

以下に、レコード削除のサンプルプログラムを示します。

【レコードの削除】

```
long rQty, i;
RNbr rNbr;

//
//      集合は既に作られて、その識別子が sid に入っているものとする
//      また、データベースハンドルが hdl に、
//      ファイル番号が file に入っているものとする
//

//      集合のサイズを知る
ec = hdl->SetSize(file, sid, &rQty);
if ((ec == ecNormal) && (rQty > 0)) {
        for (i = 0; (i < rQty) && (ec == ecNormal); i ++) {

                // i番目のレコード番号を得る
                ec = hdl->SetRGet(file, sid, i, &rNbr, NULL);
                if (ec == ecNormal) {
                        //レコードを削除
                        ec = hdl->DelRecord(file, rNbr);
                }
        }
        hdl->SetCancel(sid);
}
```

【参照】

◇ SetMake／SetRget
◇ HRecordクラス

〔レコードの更新〕

HDBHandle:: UpdRecord #include <HDBHandle.h>>

```
ECode   UpdRecord(FNbr fNbr, RNbr  rNbr,
                              const HRecord* iRec);
```

【説明】

fNbr で示すデータベースファイル内の、iRec で示す「レコード」を更新します。

アプリケーションプログラムが「レコード」を更新する場合、「SetMake関数」を使ってデータ検索を行ない、見つかったレコードを「スキーマ変換クラス(HRecord)」で変更した後、本関数を実行します。

データ検索についての詳細は、後の『集合の操作』を参照してください。

【パラメータ】

FNbr fNbr

fNbr に、更新する「データベースファイル番号」(1〜64000)を指定します。

RNbr rNbr

rNbrに、更新する「レコード番号」を指定します。

const DBRec* iRec

iRec に、「スキーマ変換クラス(HXRecord)」を利用して編集された「レコード」を指定します。

「スキーマ変換」についての詳細は、後の『拡張API 関連/HXRecord クラス』を参照してください。

【例】

以下に、レコード更新のサンプルプログラムを示します。

【レコードの更新】

```
long rQty, i;
RNbr rNbr;

//
//	集合は既に作られて、その識別子が sid に入っているものとする
//	また、データベースハンドルが hdl に、
//	ファイル番号が file に入っているものとする
//

//	集合のサイズを知る
ec = hdl->SetSize(file, sid, &rQty);
if ((ec == ecNormal) && (rQty > 0)) {
	HXRecord theRec(hdl, file);
	for (i = 0; (i < rQty) && (ec == ecNormal); i ++) {

		// i番目のレコード番号を得る
		ec = hdl->SetRGet(file, sid, i, &rNbr, (DBRec*) theRec);
		if (ec == ecNormal) {

			// ２番目の項目を"HiBase Database"に変更
			theRec.Update("[2] %sn;", "HiBase Database");

			// レコードを更新
			ec = hdl->UpdRecord(file, rNbr, &theRec);
		}
	}
	hdl->SetCancel(sid);
}
```

【参照】

◇ SetMake／SetRget
◇ HRecordクラス;　コンストラクタ／Update 関数ファミリー

１-２-４-５．集合の操作（データ検索）

「***HiBase*** API（HDBHandle クラス）」に用意されている、データ検索関連のメンバ関数は、以下の通りです。

集合の作成

〔母集合の生成〕

HDBHandle:: SetMakel #include <HDBHandle.h>

ECode SetMake(FNbr fNbr, SetID& sid, JLong& rQty, const JByte* iSearch);

【説明】

fNbr で示すデータベースファイルから、iSearch で示す「検索条件」にマッチするレコード群を探しだし、「集合」を作ります。

【パラメータ】

FNbr fNbr

fNbr に、集合を作成する「データベースファイル番号」（1～64000）を指定します。

const JByte* iSearch

検索条件式を渡します。

【戻り値】

SetID& sid

集合ID が返されます。

JLong& rQty

集合に含まれるレコードの個数（見つかったレコード数）が返されます。

検索条件式の文法

SetMakeで使用する検索条件式（const JByte* iSearch で指定）の詳細を説明します。

基本フォーマット

検索条件式 ： ＝　条件　＜　｛and ｜ or ｜ not｝ 条件 ＜...＞ ＞;

条件 ： ＝　条件式 ｜　｛集合名｜ ALL｝

条件式 ： ＝　［ 項目 または キー ］　＝　値指定

【説 明】

検索条件式は、「条件式」もしくは「ALL（全てのレコード）」もしくは「集合名」を「and／or／not」で連結し、最後にセミコロン（；）を付けます。

（上記表記の「<>」 は省略可能、「{}」 はその内の何れかを選択、「... 」は繰り返しを表します。）

「条件式」は、検索に利用する「項目」または「キー」の指定として、「項目 または キー名」もしくは、「項目 または キー番号」を 中括弧 （［ ］）で鋏み、その後ろに「イコール （＝）」を置き、その後ろに検索対象とする「値指定」を並べたもので、項目もしくはキーの値を指定するものです。

「集合名」は、すでに作られた集合をこの検索に利用するものです。

「ALL」は、集合の母集団（全てのレコード）をこの検索に利用するものです。

｛**and** ｜ **or** ｜ **not**｝

複数の「条件」を「and／or／not」で繋ぐことにより、複数の「条件」を以下の演算で組み合わせます。

◇　論理積　=　and
◇　論理和　=　or
◇　論理差　=　not

集合名 ： ＝ #集合識別子

既に作られている集合を検索条件式に指定する場合は、対象となる集合の集合名を指定します。集合名は、集合識別子（SetMakeで返される long 値）を10進文字列に変換したものを、「#」の後ろに連結して作り出します。

全て ： ＝ [ALL]

「データベースファイル」内の全ての「レコード群」を対象とする場合、「ALL」を中括弧 （[ ]）で括って指定します。

[項目 または キー指定] ： ＝ { 名前 ｜ 番号 }

名前 ： ＝ 項目 または キー名（文字列）

番号 ： ＝ < { K ¦ I } > : 項目 または キー番号

検索に利用する「項目」または「キー」の指定は、「名前」もしくは「番号」を、中括弧 （[ ]）で括ります。
「番号」で指定する場合の「キーマーク（K:)」、「項目マーク（I:)」は省略可能です。

【「項目検索」を指示する場合】

◇ 名前で指定

→ 「項目名（文字列)」を中括弧 （[ ]）で括る

// ＜例＞ [住所] = " 東京 ";

◇ 番号で指定

→ 「I : 」の後ろに「項目番号（1～255)」を置き、
中括弧 （[ ]）で括る（ ただし「I : 」は省略可能 ）

// ＜例＞ [I: 1] = -10;

【「キー検索」を指示する場合】

◇ 名前で指定

→ 「キー名（文字列)」を中括弧 （[ ]）で括る

// ＜例＞ [住所] = " 東京 ";

◇ 番号で指定

→ 「K: 」の後ろに「キー番号（1～250)」を並べ、
中括弧 （[ ]）で括る（ ただし「K : 」は省略可能 ）

// ＜例＞ [K: 1] = -10;

＝　値指定 ：＝　{ 完全一致 ¦ 前方一致 ¦ 後方一致 ¦ 中間一致 ¦ 範囲 ¦ 列挙 }

検索対象とする「値指定」は、指定の開始マークとして「イコール （＝)」を置き、その後ろに、検索対象とする「値」及び「形式」を、下記フォーマットのいずれかで指定します。

完全一致 ：＝　値
前方一致 ：＝　値 ?
後方一致 ：＝　? 値
中間一致 ：＝　? 値 ?
範囲　　 ：＝　値 - 値
列挙　　 ：＝　値 <, 値 <...>>

「一致条件（前方一致、後方一致、中間一致)」は、「文字列」の値にのみ有効な形式です。
「範囲」を指定する場合は、「ハイフン（-)」で区切り、２個の値を指定します。
「列挙（一致列)」を指定する場合は、「カンマ（,)」で区切り、複数個の値を指定します。

値　　　 ：＝　{ 数値 ¦ " 文字列 " }

「値」は、文字列／数値とも256byte 未満の指定が可能です。
「文字列」の値は、ダブルクォーテーション（" "）で括って指定します。また、値の最後を示す「ターミネータ」及び、レコードの最後を示す「拡張ターミネータ」を検知する必要がある場合は、「エスケープ（\)」を指定します。

【「値指定」の形式】

// 文字列の一致条件

◇ 前方一致 ＝ " 文字列の値 " ?
→ 「＝」の後ろにダブルクォーテーション（" "）で括った「文字列」を置き、その後ろに「クェスチョンマーク（？)」を置く

// ＜例＞ [n] = "鈴木 "?;

◇ 後方一致 ＝ ? " 文字列の値 "
→ 「＝」の後ろに「クェスチョンマーク（？)」を置き、その後ろにダブルクォーテーション（" "）で括った「文字列」を置く

// ＜例＞ [n] = ? "正男 ";

◇ 中間一致 ＝ ? " 文字列の値 " ?
→ 「＝」の後ろに、ダブルクォーテーション（" "）で括った「文字列」を「クェス

チョンマーク（？)」で鋏んで置く

// ＜例＞ [n] = ? "東京 "?;

// 値の形式

◇ 単独一致（イコール） → 「＝」の後ろに「値」を置く

// ＜例＞ [n] = 100;

// [n] = " 鈴木";

◇ 範囲一致（スルー）

→ 「＝」の後ろに、「ハイフン（-)」で区切って２個の「値」を置く

// ＜例＞ [n] = 100 - 200;

// [n] = "100" - "200";

◇ 一致列（イコールリスト）

→「＝」の後ろに、「カンマ（,)」で区切って複数個の「値」を置く

// ＜例＞ [n] = 100,200,300,400;

// [n] = " 東京", "大阪";

【 例 】

以下に、検索条件のパラメータ指定のサンプルを示します。

【 数値の検索条件例 】

◇ 項目番号１の値が「-10」

→ [1] = -10;

◇ キー番号２の値が「-10」

→ [K2] = -10;

◇ キー番号３の値が、「B00100」、「B00110」、「B00120」、「B00130」のいずれか

→ [K3] = "B00100", "B00110", "B00120", "B00130";

◇ 項目名「売上金額」の値が、「1,000,000」から「3,000,000」の範囲のレコードを検索する

→ [売上金額] =1000000 -2000000;

【文字列の検索条件例】

◇ 項目番号１の値が「鈴木」のレコードを検索する

→ [1] = " 鈴木";

◇ キー番号２の値が「鈴木」のレコードを検索する

→ [K2] = "鈴木";

◇ 項目名「コード番号」の値が、文字列数字「B00100」、「B00110」、「B00120」、「B00130」のいずれかレコードを検索する

→ [コード番号] = "B00100", "B00110", "B00120", "B00130";

◇ 項目名「名前」の値が、「鈴木」で始るレコードを検索する

→ [名前] = "鈴木 "? ;

◇ キー名「名前」の値が、「鈴木」で終わるレコードを検索する

→ [名前] = ? "鈴木 ";

◇ キー番号４の値が、「鈴木」含むレコードを検索する

→ [K4] = ? "鈴木 "? ;

◇ 項目名「名前」の値が、「鈴木」ではないレコードを検索する

→ [all] not [名前] = "鈴木 ";

◇ キー番号５の値が、「鈴木」ではないレコードを検索する

→ [all] not [K:5] = "鈴木 " ;

◇ 項目名「趣味」の値が「スポーツ」か「読書」で、項目名「性別」の値が「女性」のレコードを検索する

→ [趣味] = " スポーツ", "読書" and [性別] = "女性";

集合内のソート

〔集合のソート〕

HDBHandle:: SetSort #include <HDBHandle.h>

ECode SetSort (FNbr fNbr, SetID sid, const JByte* iSort);

【説 明】

fNbr で示すデータベースファイル内の、sid で示す「集合」を、iSort で示すソート条件でソート（集合内のレコードの並びを変える）します。

【パラメータ】

FNbr fNbr

fNbr に、集合を作成する「データベースファイル番号」（1～64000）を指定します。

SetID sid

sid に、ソートする「集合番号」を指定します。

const JByte* iSort

iSort に、「ソート条件」を指定します。

ソート条件の文法

基本フォーマット

ソート条件 ： ＝ 項目指定 ＜ , 項目指定 ＞;

項目指定 ： ＝ [項目番号] % <a / d><s / i>

【 説 明 】

ソート条件は、項目指定（ソートの対象となるデータベースファイルの項目）をカンマ区切りで並べたものです。複数の項目指定がなされた場合、ソート条件は、左から右に優先順位が付けられます。
項目指定は、項目番号を「中カッコ（[]）」で囲み、その後ろにアンパサント記号（%）とソートオプションを付加したものです。
ソートオプションは以下のようになっています。

<a / d>....aは昇順、dは降順ソート（デファオルトは昇順）
<s / i>...sはケース付、i はケース無視のソート（デファオルトはケース付）

＜例＞ 項目1を「昇順」、項目2を「降順、ケース無視」

```
SetSort(fileNumber,  sid,   "[1], [2] %di; ");
```

集合内のレコード操作

〔レコード数の取得〕

HDBHandle:: SetSize #include <HDBHandle.h>

ECode SetSize (FNbr fNbr, SetID sid, JLong& rQty);

【説 明】

fNbr で示すデータベースファイルの、sid で示す集合内の「レコード」の個数を得ます。

【パラメータ】

FNbr fNbr

fNbr に、レコード数を取得する「データベースファイル番号」(1〜64000)を指定します。

SetID sid

sid に、レコード数を取得する「集合番号」を指定します。

【戻り値】

JLong& rQty

rQty に、sid で示す集合内に含まれる「レコード」の個数が返されます。

〔レコードの読み出し〕

HDBHandle:: SetRGet #include <HDBHandle.h>

```
ECode  SetRGet (FNbr fNbr, SetID  sid, JLong idx,
                RNbr& rNbr, HRecord* oRec = NULL);
```

【説明】

fNbr で示すデータベースファイルの、sid で示す「集合」から、idx で示すインデックスの「レコード」を読み出します。

読み出されたレコードは、「スキーマ変換クラス（HRecord)」を利用して編集します。

「スキーマ変換」についての詳細は、後の『API 関連／HRecordクラス』を参照してください。

【パラメータ】

FNbr fNbr

fNbr に、読み出す「データベースファイル番号」(1〜64000）を指定します。

SetID sid

sid に、読み出す「集合番号」を指定します。

JLong idx

rNbrに、読み出すレコードの「インデックス」を指定します。

「インデックス」は、先頭レコードを「0（ゼロ)」とてナンバリングします。

【戻り値】

RN&r rNbr

rNbrに、読み出す「レコード番号」が 返されます。

HRecord* oRec

oRecに、sid で示す集合から読み出された「レコード」が 返されます。

【例】

以下に、レコード読み出しのサンプルプログラムを示します。

【レコードの読み出し】

```
long rQty, i;
RNbr rNbr;

struct {
        long nbr;               // item #1
        char aItem[32];         // item #2
        char bItem[32];         // item #3
        char cItem[32];         // item #4
} theData;

//
//  集合は既に作られて、その識別子が sid に入っているものとする
//  また、データベースハンドルが hdl に、
//  ファイル番号が file に入っているものとする
//

//  集合のサイズを得る
ec = hdl->SetSize(file, sid, &rQty);
if ((ec == ecNormal) && (rQty > 0)) {
        HXRecord theRec(hdl, file);
        for (i = 0; (i < rQty) && (ec == ecNormal); i ++) {

        //  i番目のレコードを読み出す
                ec = hdl->SetRGet(file, sid, i, &rNbr, &theRec);
                if (ec == ecNormal) {

                        //  「外部データ構造」に変換
                        theRec.Fetch("[1] %4i, [2]-[4] %32sn;", &theData);
                }
        }
        hdl->SetCancel(sid);
}
```

【参照】

◇ HRecordクラス

〔レコードの集合への追加〕

HDBHandle:: SetRAdd #include <HDBHandle.h>

ECode SetRAdd (FNbr fNbr, SetID sid, RNbr rNbr);

【説明】

fNbr で示すデータベースファイルの、sid で示す「集合」内に、rNbr で示す「レコード」を追加します。

【パラメータ】

FNbr fNbr

fNbr に、追加する「データベースファイル番号」(1〜64000)を指定します。

SetID* sid

sid に、追加する「集合番号」を指定します。

Nbr rNbr

rNbrに、追加する「レコード番号」を指定します。

〔レコードの集合からの削除〕

HDBHandle:: SetRDel #include <HDBHandle.h>

ECode SetRDel (FNbr fNbr, SetID sid, RNbr rNbr);

【説明】

fNbr で示すデータベースファイルの、sid で示す集合から、rNbr で示すレコードを削除します。

【パラメータ】

FNbr fNbr

fNbr に、削除する「データベースファイル番号」(1～64000）を指定します。

SetID* sid

sid に、削除する「集合番号」を指定します。

Nbr rNbr

rNbrに、削除する「レコード番号」を指定します。

〔集合の削除〕

HDBHandle:: SetCancel #include <HDBHandle.h>

ECode SetCancel (FNbr fNbr, SetID sid);

【説明】

fNbr で示すデータベースファイル内の、sid で示す「集合」を削除します。

【パラメータ】

FNbr fNbr

fNbr に、削除する「データベースファイル番号」（1〜64000）を指定します。

SetID* sid

sid に、削除する「集合番号」を指定します。

〔集合の削除〕

HDBHandle:: SortCancel #include <HDBHandle.h>

ECode SortCancel (FNbr fNbr, SetID sid);

【説明】

fNbr で示すデータベースファイル内の、sid で示すソートされている「集合」のソートを取り消します。

【パラメータ】

FNbr fNbr

fNbr に、削除する「データベースファイル番号」(1〜64000)を指定します。

SetID* sid

sid に、削除する「集合番号」を指定します。

本ページ 空白

１-２-４-６．シーケンシャル処理

「***HiBase*** API（HDBHandle クラス)」は、全レコードを対象とする「一括処理」のための、以下の「シーケンシャル処理」関連の機能を用意しています。

キーシーケンシャルアクセス

〔読み出し開始位置の設定〕

HDBHandle:: KSLocate #include <HDBHandle.h>

```
ECode  KSLocate(FNbr fNbr, SetID& sid, JWord kid,
                                const UByte* pStr);
```

【説明】

fNbr で示すデータベースファイルに、「キーシーケンシャルアクセス」の開始位置を設定します。kid で示す「キー番号」の、pStr で示す「キーの値」が、最初の読み出し位置になります。

「キーシーケンシャルアクセス機能」とは、１つの「キー」の値の順番に、「レコード」を順次読み出す機能です。本関数は、KSRead関数で上記機能を実行する際の条件設定を行なうものです。

【パラメータ】

FNbr fNbr

fNbr に、キーシーケンシャルアクセスを行なう「データベースファイル番号」（1～64000）を指定します。

JWord kid

kid に、読み出す「キー番号」（1～250）を指定します。

const UByte* pStr

pStr に、キーシーケンシャルアクセスを開始する「キー」の値を指定します。

【戻り値】

SetID& sid

sid に、読み出し時に作成する「集合番号」が返されます。

「キーシーケンシャルアクセス」を実行する場合は、この値をパラメータとして KSRead関数に渡します。

【例】

KSReadの『キー順次読み出しのサンプルプログラム』を参照してください。

【参照】

◇ KSRead

〔キー順次読み出し〕

HDBHandle:: KSRead #include <HDBHandle.h>

ECode KSRead (FNbr fNbr, SetID sid, UByte* pStr, JWord& rQty, RNbr* rList);

【説明】

fNbr で示すデータベースファイルに対し、「キーシーケンシャルアクセス」を実行します。該当する「キーの値」、および、「レコード番号」が読み出されます。

本関数を繰り返し呼び出すと、KSLocate 関数のkid で指定した「キー番号」の、pStr で指定した値以降の「レコード番号列」が、sidで示す「集合」に、順次降順で読み出されます。

本機能は、全データを対象に、１つの「キー」の値の順番に、「レコード」を順次読み出す場合に利用します。

そのため、本関数の実行に先立って、KSLocate 関数を利用し、読み出しの条件（読み出すキー番号／開始値）を指定しておく必要があります。

なお、本関数は、「レコード番号」のみを読み出すため、「レコード本体」が必

要な場合は、続けて GetRecord 関数を呼び出し、「レコード」の読み出しを実行する必要があります。読み出されたレコードは、通常「スキーマ変換クラス（HXRecord）」を利用して編集します。

「スキーマ変換」についての詳細は、後の『 拡張API 関連／HXRecord クラス 』を参照してください。

【 パラメータ 】

FNbr fNbr

fNbr に、キーシーケンシャルアクセスを行なう「データベースファイル番号」（1〜64000）を指定します。

SetID　sid

sidに、KSLocate 関数で返された、読み出す「集合番号」を指定します。

【 戻り値 】

UByte* pStr

pStr に、読み出された「キーの値」が返されます。

JWord& rQty

rQty に、読み出されたキーの値を持つ「キーの個数」が返されます。

RNbr* rList

rList に、読み出された「レコード番号リスト」が返されます。

【例】

以下に、キーシーケンシャルアクセスのサンプルプログラムを示します。

【キー順次読み出し】

```
struct {
        long nbr;           // item #1
        char aItem[32];     // item #2
        char bItem[32];     // item #3
        char cItem[32];     // item #4
} theData;

SetID sid;
short rQty, i;
RNbr rList[512];


// 読み出し開始位置をキーの値（"[2]=A00001"）で設定
ec = hdl->KSLocate(file, &sid, 2, "\pA00001");
while (ec == ecNormal) {
        HRecord theRec(hdl, file);

//次の値とその値を持つレコード件数及びレコード番号リストを読み出す
        ec = hdl->KSRead(file, sid, pVal, rQty, rList);
        if (ec == ecNormal) {
                for (JWord i = 0; i < rQty; i ++) {
                        ec = hdl->GetRecord(file, rList[i], & theRec);
                        if (ec == ecNormal) {
                                theRec.Fetch("[1] %4i, [2]-[4] %32sn;", &theRec);
                                ..................................
                        }
                }
        }
        hdl->KSCancel(sid);
}
```

【参照】

- ◇ KSLocate／KSCancel
- ◇ GetRecord
- ◇ HRecordクラス

〔読み出し終了〕

HDBHandle:: KSCancel #include <HDBHandle.h>

ECode KSCancel(FNbr fNbr, SetID sid);

【説明】

fNbr で示すデータベースファイル内の、sidで示す「集合」で実行されている「キーシーケンシャルアクセス」を終了します。

【パラメータ】

FNbr fNbr

fNbr に、キーシーケンシャルアクセスを終了する「データベースファイル番号」(1〜64000）を指定します。

SetID sid

sidに、KSRead 関数で指定した、キーシーケンシャルアクセス中の「集合番号」を指定します。

【例】

KSReadの『キー順次読み出しのサンプルプログラム』を参照してください。

【参照】

◇ KSRead、KSLocate

高速シーケンシャルアクセス

〔読み出し開始の宣言〕

HDBHandle:: PSLocate #include <HDBHandle.h>

ECode　PSLocate(FNbr fNbr, SetID& sid);

【説明】

fNbr で示すデータベースファイルに対し、「高速シーケンシャルアクセス」の開始を宣言します。

「高速シーケンシャルアクセス機能」とは、データベース内の「全レコード」を、格納されている順番に、高速に読み出す機能です。本関数は、PSRead関数で上記機能を実行するための宣言を行なうものです。

【パラメータ】

FNbr fNbr

fNbr に、高速シーケンシャルアクセスを行なう「データベースファイル番号」（1〜64000）を指定します。

SetID& sid

sid に、読み出し時に作成される「集合番号」が返されます。

「高速シーケンシャルアクセス」を実行する場合は、この値をパラメータとしてPSRead関数に渡します。

【例】

KSReadの『順不同読み出しのサンプルプログラム』を参照してください。

【参照】

◇ PSRead

〔順不同高速読み出し〕

HDBHandle:: PSRead #include <HDBHandle.h>

ECode PSRead (FNbr fNbr, SetID sid, RNbr* rNbr, HRecord* oRec);

【説明】

fNbr で示すデータベースファイルに対し、「高速シーケンシャルアクセス」を実行します。

本関数の実行に先立って、PSLocate 関数を利用し、読み出しの開始を宣言しておく必要があります。

本機能は、データベース内の全データを対象に、「レコード」を高速に読み出す場合に利用します。本関数を繰り返し呼び出すと、データベース内の「レコード」が、格納順に、順次oRec に読み出されます。

【パラメータ】

FNbr fNbr

fNbr に、高速シーケンシャルアクセスを行なう「データベースファイル番号」(1〜64000)を指定します。

SetID sid

sidに、PSLocate 関数で返された「集合番号」を指定します。

【戻り値】

RNbr* rList

rList に、読み出された「レコード番号リスト」が返されます。

HRecord* oRec

oRec に、読み出された「レコード」が返されます。

読み出されたレコードは、「スキーマ変換クラス(HRecord)」を利用して編集します。

「スキーマ変換」についての詳細は、後の『API 関連/HRecordクラス』を参照してください。

【例】

以下に、高速シーケンシャルアクセスのサンプルプログラムを示します。

【順不同高速読み出し】

```
struct {
        long nbr;           // item #1
        char aItem[32];     // item #2
        char bItem[32];     // item #3
        char cItem[32];     // item #4
} theData;

SetID sid;
RNbr rNbr;

// 読み出しの宣言
ec = hdl->PSLocate(file, &sid);
while (ec == ecNormal) {
        HXRecord theRec(hdl, file);

//次の値とその値を持つレコード件数及びレコード番号リストを読み出す
        ec = hdl->PSRead(file, sid, rNbr, &theRec);
        if (ec == ecNormal) {
                theRec.Fetch("[1] %4i, [2]-[4] %32sn;", &theData);
                                ..................................
        }
        hdl->PSCancel(sid);
}
```

【参照】

◇ PSLocate／PSCancel
◇ HXRecordクラス

〔読み出し終了〕

HDBHandle:: PSCancel #include <HDBHandle.h>

ECode PSCancel (FNbr fNbr, SetID sid);

【説明】

fNbr で示すデータベースファイル内の、sidで示す「集合」で実行されている「高速シーケンシャルアクセス」を終了します。

【パラメータ】

FNbr fNbr

fNbr に、高速シーケンシャルアクセスを終了する「データベースファイル番号」（1〜64000）を指定します。

SetID sid

sidに、PSRead 関数で指定した、高速シーケンシャルアクセス中の「集合番号」を指定します。

【例】

PSReadの『順不同高速読み出しのサンプルプログラム』を参照してください。

【参照】

◇ PSRead

本ページ 空白

Chapter 2

HiBase API

スキーマの操作

この章では、スキーマ操作に関連する機能を説明します。

「***HiBase*** API」を利用してデータベースの操作を実施するとき、データベースのスキーマの操作が必要になります。例えば、データベースにファイルを生成するときはデータベースファイルのスキーマが必要です。また、データベースから返されたレコードはHiBaseの内部的なスキーマの状態ですから、これをアプリケーションプログラムに都合のいい外部スキーマに変換する必要が有ります。

「スキーマの変換（HRecord クラス）」

「スキーマの生成（HSchima クラス）」

スキーマの変換 - アプリケーションプログラムのデータ構造へ...

HRecord クラス - レコードにアクセス

項目データの操作

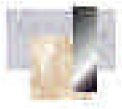
その他のメンバ関数

スキーマ変換のパラメータフォーマット

スキーマの生成 - DDLからスキーマへ...

HSchema クラス

DDL（データベース定義文法）



HiBase API スキーマの操作 ...

スキーマ変換クラス

HRecord クラス

「スキーマ変換クラス（HRecord)」は、***HiBase*** の「内部形式レコード」をアプリケーションプログラムの要求する「外部データ構造」に変換する機能を提供します。

スキーマの生成クラス

HSchema クラス

「スキーマ生成クラス（HSchema)」は、DDL で記述されたスキーマ定義を HiBaseAPIに渡すデータ構造に翻訳します。

Chapter 2 - 1

スキーマ変換

- アプリケーションプログラムのデータ構造へ...

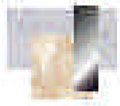
この章では、「スキーマ変換クラス（HRecord）」の「メンバ関数」について説明します。

「スキーマ変換クラス（HRecord）」は、***HiBase*** の「内部形式レコード」をアプリケーションプログラムの要求する「外部データ構造」に変換し、***HiBase*** データベースの「レコード」を「項目」単位で編集する機能を提供します。

この章で解説する主要な項目は、次の通りです。

HXRecord クラス - レコードにアクセス

- 項目データの操作
- その他のメンバ関数
- スキーマ変換のパラメータフォーマット



2-1. HRecord クラス　- レコードにアクセス

「スキーマ変換クラス（HRecord）」は、***HiBase*** の「内部形式レコード」と、アプリケーションプログラムの要求する「外部データ構造」を、双方向に変換する機能を提供します。

具体的には、Append／Fetch／Update／Deleteなどのメンバ関数群で、「項目データ」の追加／読み出し／更新／削除を行ないます。

メンバ関数

Append	AppendItem
Fetch	FetchItem
Update	UpdateItem
Delete	DeleteItem
ValueCount	ValueSize
ValueType	DBRec
Length	Clear

【 HRecord の機能 】

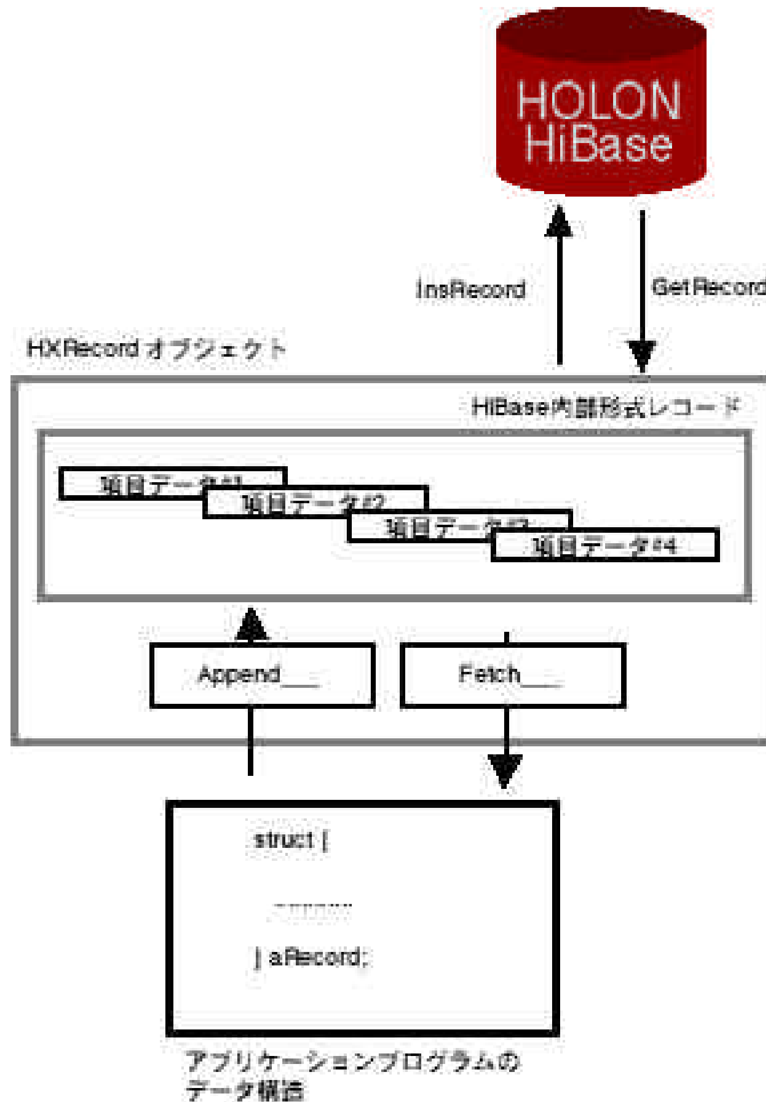
HRecord クラスは、レコードのの追加（HDBHandle::InsRecord）、読み出し（HDBHandle::GetRecord、HDBHandle::GetRGet、HDBHandle::PSRead）、更新（HDBHandle::InsRecord）の時、アプリケーションプログラムから必ず利用されるクラスです。

HiBaseAPIは、HiBaseデータベースから目的のレコードを内部形式のまま入出力します。言い換えると、レコードというコンテナをデータベースから出したり入れたりしていると考えることができます。そしてこのコンテナに荷物（項目の値）を出し入れするのが、ここで説明しているHRecord（スキーマ変換クラス）です。

アプリケーションプログラムが、データベースにデータを追加するときは、まず、HRecordオブジェクトを作り、ここに項目の値を積んでゆきます（Append)。予定の項目の値を積み終わったならば、HiBaseAPIのHDBHandle::InsRecordを呼びだしてこの荷物の詰まったコンテナをHiBaseに送り、HiBaseはこれをデータベースに格納します。

反対に、データを読み出すときは、まずHiBaseAPIのHDBHandle::GetRecordを呼びだし、レコードであるコンテナをHRecordオブジェクトに取りだします。その後、HRecord は荷物である項目の値をアプリケーションプログラムに渡します(Fetch)。

【 HRecord の概念図 】



２-１-１-１．コンストラクタ

HRecord:: #include <HRecord.h>

HRecord ();
HRecord (JWord allocSize);

【説明】

HRecord クラスのインスタンスを生成します。

【パラメータ】

JWord allocSize

allocSize に、***HiBase*** の「内部形式レコード」を格納する「メモリープール」のサイズ（Byte 単位）を指定します。

「メモリープール」は、1024 Bytes がデフォルトで確保され、データが追加されるに従って自動的に拡張されますが、予めサイズがわかっている「レコード」を扱う場合は、このパラメータを利用することで、わずかですが、効率が良くなります。

【例】

Appendの『項目データ追加のサンプルプログラム』、Fetchの『項目データ読み出しのサンプルプログラム』を参照してください。

【参照】

◇ Append／Fetch
◇ HDBHandleクラス; InsRecord／GetRecord ／SetRGet

2-1-1.項目データの操作

アプリケーションプログラムが ***HiBase*** の「データベース」のデータ編集を行なう場合は、Append／Fetch／Update／Deleteの各メンバ関数群を利用して、「項目データ」の追加／読み出し／更新／削除を実行します。

2-1-1-2. 項目データの追加

「項目データ」を追加する場合は、以下の４種類のいずれかのAppend メンバ関数を利用します。

〔複数項目データの追加〕

HRecord:: Append #include <HRecord.h>

ECode Append (const JByte* fb, const void* iRec);

ECode Append (const JByte* fb, HStream& iRec);

【説明】

iRecで渡されたデータを、fb の「スキーマ変換フォーマット」に従ってHRecord内に取り込みます。

【パラメータ】

const JByte* fb

fbに、「スキーマ変換フォーマット」を指定します。

「スキーマ変換フォーマット」とは、***HiBase*** を利用するアプリケーションプログラムの要求する「外部データ構造」を、パラメータで指示したものです。

「スキーマ変換フォーマット」についての詳細は、後の『スキーマ変換のパラメータフォーマット』を参照してください。

const void* iRec

iRecに、追加する「項目データ」へのポインタを指定します。

HStream& iRec

追加する「項目データ」をストリーム（HStream）で指定することもできます。

【例】

以下に、項目データ追加のサンプルプログラムを示します。

【レコードの追加】

```
HRecord theRec(hdl, file);                  // スキーマクラスを準備
long nbr = 1;
char aItem[32], bItem[32], cItem[32];
RNbr rNbr;                                              // レコード番号
ECode ec;                                               // エラーコード

theRec.Clear();                                  // レコードを初期化

        //   項目１を long integerで設定
theRec.Append("[1] %4i", &nbr);
sprintf(aItem, "A%05ld", nbr);
        //   項目２を文字列（c 文字列）で設定
theRec.Append("[2] %sn", aItem);
sprintf(bItem, "B%05ld", nbr);
        //   項目３を文字列（c 文字列）で設定
theRec.Append("[3] %sn", bItem);
sprintf(cItem, "C%05ld", nbr);
        //   項目４を文字列（c 文字列）で設定
theRec.Append("[4] %sn", cItem);

ec = hdl->InsRecord(file, rNbr, theRec);   // レコードを追加
```

【参照】

◇　スキーマ変換のパラメータフォーマット
◇　HDBHandleクラス;　InsRecord

〔１項目データの追加〕

HXRecord:: AppendItem #include <HRecord.h>

```
ECode   AppendItem (JWord iid, JWord iType,
            const JByte* pData, JLong lData, HDBHandle* theHdl = NULL);
ECode   AppendItem (JWord iid, JWord iType,
            const JHdl hData, JLong lData, HDBHandle* theHdl = NULL);
ECode   AppendItem (JWord iid, JWord iType,
            HFile& iFile, HDBHandle* theHdl = NULL);
ECode   AppendItem (JWord iid, JWord iType,
            HStream& iStream, HDBHandle* theHdl = NULL);
```

【説 明】

iid（項目番号）で示す項目に、iType で示すデータ型で、データを取り込みます。データの格納形式は、ポインタ（pData)、ハンドル（hData)、ファイル（iFile)、ストリーム（iStream）を選ぶことができます。

本関数は、大きなメモリーを必要とするデータ項目（マルチメディアデータ型を持つ項目）の為に用意されたものです。しかしながら、一般データ型で使用することもできます。

一般データ型で使用する場合、最後のパラメータ「HDBHandle* theHdl」を省略することが出来ます。

【パラメータ】

JWord iid

iidに、追加する項目の「項目番号」(1～255）を指定します。

同一項目番号に対して追加を繰り返した場合、データは「マルチバリュー」として格納されます。

「マルチバリュー」についての詳細は、後の『スキーマ変換のパラメータフォーマット』を参照してください。

JWord　iType

iTypeには、「HiBaseデータ型」のいずれかを指定します。
「HiBaseデータ型」に関しては、付録の「App 1. HiBaseデータベース項目のデータ型」を参照してください。

const char* pData

JHdl hData

HFile& iFile

HStream& iStream

追加すべき項目データを、ポインター、ハンドル、ファイル、ストリームの形式で渡すことができます。

JLong lData

lDataに、「項目のデータ長」を指定します。

HDBHandle*　　theHdl = NULL

マルチメディアデータ型を取り扱う場合、HiBaseAPI（HDBHandler）オブジェクトへのポインタを指定します。一般データ型を取り扱う場合、このパラメータは省略できます。

【参照】

◇　HDBHandleクラス;　InsRecord

2-1-1-3．項目データ読み出し

「項目データ」を読み出す場合は、以下の４種類のいずれかのFetchメンバ関数を利用します。

〔複数項目データの読み出し〕

HRecord:: Fetch #include <HRecord.h>

```
ECode  Fetch (const JByte* fb, void* oRec,
                  JLong* oLen=NULL);
ECode  Fetch (const JByte* fb, HStream& oRec);
```

【説明】

本関数は、アプリケーションプログラムが ***HiBase*** のデータベースファイルから「レコード」を読み出す場合、最も汎用的に利用するメンバ関数です。

「メモリープール」に展開されている ***HiBase*** の「内部形式レコード」を、fbで示す「外部データ構造」に変換し、oRecに格納します。

本関数の実行に先立って、「***HiBase*** API（HDBHandleクラス)」のGetRecord 或いは、SetRGet関数で ***HiBase*** の「内部形式レコード」を読み出し、これを「スキーマ変換クラス（HRecord)」のコンストラクタで「メモリープール」に展開しておく必要があります。
GetRecord ／SetRGet関数についての詳細は、『API 関連／HDBHandleクラス 』を参照してください。

【パラメータ】

const JByte* fb

fbに、「スキーマ変換フォーマット」を指定します。

「スキーマ変換フォーマット」とは、***HiBase*** を利用するアプリケーションプログラムの要求する「外部データ構造」を、パラメータで指示したものです。

「スキーマ変換フォーマット」についての詳細は、後の『スキーマ変換のパラメータフォーマット』を参照してください。

【戻り値】

void* oRec
HStream& oRec

oRecに、fbで示す「外部データ構造」で読み出す「項目データ」の受け取り場所を指定します。メモリーへのポインターの他にHStreamオブジェクトを選択できます。

JLong* oLen=NULL

変換されたデータの長さを返します。
なお、このパラメータは省略することができます。

【例】

以下に、項目データ追加のサンプルプログラムを示します。

【レコードの読み出し】

```
long rQty, i;
RNbr rNbr;

struct {
        long nbr;           // item #1
        char aItem[32];     // item #2
        char bItem[32];     // item #3
        char cItem[32];     // item #4
} theData;

//
//  集合は既に作られて、その識別子が sid に入っているものとする
//  また、データベースハンドルが hdl に、
//  ファイル番号が file に入っているものとする
//

//  集合のサイズを得る
ec = hdl->SetSize(file, sid, &rQty);
if ((ec == ecNormal) && (rQty > 0)) {
        HRecord theRec(hdl, file);
        for (i = 0; (i < rQty) && (ec == ecNormal); i ++) {

        //  i番目のレコードを読み出す
                ec = hdl->SetRGet(file, sid, i, &rNbr, theRec);
                if (ec == ecNormal) {

                        //  「外部データ構造」に変換
                        theRec.Fetch("[1] %4i, [2]-[4] %32sn;", &theData);
                }
        }
        hdl->SetCancel(sid);
}
```

【参照】

◇　スキーマ変換のパラメータフォーマット

◇　HDBHandleクラス;　GetRecord／
SetMakeKey／SetMakeItem／SetRGet

〔１項目データの読み出し〕

HXRecord:: FetchItem #include <HRecord.h>

```
ECode    FetchItem (JWord iid, JWord idx, JWord iType,
         JByte* pData, JLong& lData, HDBHandle* theHdl = NULL);
ECode    FetchItem (JWord iid, JWord idx, JWord iType,
         JHdl hData, JLong& lData, HDBHandle* theHdl = NULL);
ECode    FetchItem (JWord iid, JWord idx, JWord iType,
         HFile& oFile, HDBHandle* theHdl = NULL);
ECode    FetchItem (JWord iid, JWord idx, JWord iType,
         HStream& oStream, HDBHandle* theHdl = NULL);
```

【説明】

iid（項目番号）で示す項目を、iType で示すデータ型で、項目値を読み出します。データの格納形式は、ポインタ（pData)、ハンドル（hData)、ファイル（iFile)、ストリーム（iStream）を選ぶことができます。

本関数は、大きなメモリーを必要とするデータ項目（マルチメディアデータ型を持つ項目）の為に用意されたものです。しかしながら、一般データ型で使用することもできます。

一般データ型で使用する場合、最後のパラメータ「HDBHandle* theHdl」を省略することができます。

【パラメータ】

JWord iid

iidに、読み出す「項目番号」(1～255）を指定します。

JWord idx

idxに、読み出す「バリューインデックス」(0～255）を指定します。

「バリューインデックス」とは、iidで示す「項目番号」に、複数の値（「マルチバリュー」)が格納されている場合に付けられる「値の番号」です。

「0」が指定された場合、１番目の値が読み出されます。

なお、「マルチバリュー」中の任意「値」を読み出す場合、本関数の実行に先立って、「Value メンバ関数群（ValueCount、ValueSize、ValueType)」を利用することで、同一項目に対して、幾つの「値」が入っているのか、各値のサイズ／データタイプを調べ、事前に対象とする「値」の情報を得ることができます。

「Value メンバ関数群（ValueCount、ValueSize、ValueType)」についての詳細は、後の『項目データの情報取得』を参照してください。

JWord　iType

iType に、HiBase データベース項目のデータ型の識別子を指定します。
データ型の識別子に関しては、付録「App 1 - HiBaseデータベース項目のデータ型」を参照してください。

HDBHandle* theHdl = NULL

マルチメディアデータ型を取り扱う場合、HiBaseAPI（HDBHandler）オブジェクトへのポインタを指定します。一般データ型を取り扱う場合、このパラメータは省略できます。

【戻り値】

JByte* pData

JHdl hData

HFile& oFile

HStream& oStream

項目データを、ポインター、ハンドル、ファイル、ストリームの形式で受け取ることができます。

JLong& lData

読み出されたデータ長が返されます。

【参照】

◇　ValueCount／ValueSize ／ValueType
◇　HDBHandleクラス;　GetRecord／SetRGet

2-1-1-4．項目データの更新

「項目データ」を更新する場合は、以下の２種類のいずれかのUpdateメンバ関数を利用します。

〔複数項目データの更新〕

HXRecord:: Update #include <HRecord.h>

ECode Update (const JByte* fb, const void* iRec);

ECode Update (const JByte* fb, HStream& iRec);

【説明】

本関数は、複数の「項目データ」を一度に更新する場合に利用します。

この関数が呼び出された場合、fbで示す「外部データ構造」で表現された「項目データ」を、***HiBase***の「内部形式レコード」に変換して、「メモリープール」を更新します。本関数実行後、「***HiBase*** API（HDBHandleクラス）」のUpdRecord関数を呼び出し、この「メモリープール」へのポインタをパラメータとして渡して、「レコード」の更新を実行します。

UpdRecord関数についての詳細は、『API 関連／HDBHandleクラス』を参照してください。

【パラメータ】

const JByte* fb

fbに、「スキーマ変換フォーマット」を指定します。

「スキーマ変換フォーマット」についての詳細は、後の『スキーマ変換のパラメータフォーマット』を参照してください。

const void* iRec
HStream& iRec

iRecに、更新する「項目データ」へのポインタもしくはHStreamオブジェクトを指定します。

【例】

以下に、項目データ更新のサンプルプログラムを示します。

【レコードの更新】

```
long rQty, i;
RNbr rNbr;

//
//      集合は既に作られて、その識別子が sid に入っているものとする
//      また、データベースハンドルが hdl に、
//      ファイル番号が file に入っているものとする
//

//      集合のサイズを知る
ec = hdl->SetSize(file, sid, &rQty);
if ((ec == ecNormal) && (rQty > 0)) {
        HRecord theRec(hdl, file);
        for (i = 0; (i < rQty) && (ec == ecNormal); i ++) {

                //  i番目のレコード番号を得る
                ec = hdl->SetRGet(file, sid, i, &rNbr, theRec);
                if (ec == ecNormal) {

                        //  ２番目の項目を"HiBase Database"に変更
                        theRec.Update("[2] %sn;", "HiBase Database");

                        //  レコードを更新
                        ec = hdl->UpdRecord(file, rNbr, (DBRec*) theRec);
                }
        }
        hdl->SetCancel(sid);
}
```

【参照】

◇ スキーマ変換のパラメータフォーマット

◇ HDBHandleクラス;　UpdRecord／GetRecord
SetMake／SetRGet

２-１-１-５．項目データの削除

「項目データ」を削除する場合は、以下の２種類のいずれかのDelete メンバ関数を利用します。

〔複数項目データの削除〕

HXRecord:: Delete #include <HRecord.h>

ECode　Delete (const JByte* fb);

【説明】

本関数は、複数の「項目データ」を一度に削除する場合に利用します。

この関数が呼び出された場合、fbで示す「外部データ構造」で表現された「項目データ」を、「メモリープール」から削除します。

本関数の実行に先立って、「メモリープール」に展開されているデータや、そのサイズなどを確認する場合は、DBRec ／Length 等の関数を利用します。
また、「メモリープール」内の全てのデータを削除する場合は、Clear 関数を利用することができます。

上記関数についての詳細は、後の『メモリープール内の情報取得』を参照してください。

【パラメータ】

const JByte* fb

fbに、「スキーマ変換フォーマット」を指定します。

「スキーマ変換フォーマット」についての詳細は、後の『スキーマ変換のパラメータフォーマット』を参照してください。

【参照】

◇　スキーマ変換のパラメータフォーマット
◇　DBRec ／Length／Clear
◇　HDBHandleクラス;　GetRecord／UpdRecord／DelRecord

〔１項目データの削除〕

HXRecord:: DeleteItem #include <HRecord.h>

ECode DeleteItem (JWord iid, JWord idx,

HDBHandle* theHdl = NULL);

【説明】

一度に１つの「項目データ」を削除する場合に利用します。「外部データ構造」で表現されたデータの内、iidで示す「項目番号」の、idxで示す「バリューインデックス」のデータを削除します。

本関数は、大きなメモリーを必要とするデータ項目（マルチメディアデータ型を持つ項目）の為に用意されたものです。しかしながら、一般データ型で使用することもできます。

一般データ型で使用する場合、最後のパラメータ「HDBHandle* theHdl」を省略することが出来ます。

【パラメータ】

JWord iid

iidに、削除する「項目番号」（1～255）を指定します。

JWord idx

idxに、削除する「バリューインデックス」（0～255）を指定します。

「1～255」が指定された場合はその値のみが削除されますが、「0」が指定された場合、全ての値が削除されます。

なお、「マルチバリュー」中の任意「値」を削除する場合、本関数の実行に先立って、「Value メンバ関数群（ValueCount、ValueSize、ValueType）」を利用することで、同一項目に対して、幾つの「値」が入っているのか、各値のサイズ／データタイプを調べ、事前に対象とする「値」の情報を得ることができます。

「Value メンバ関数群（ValueCount、ValueSize、ValueType）」についての詳細は、後の『項目データの情報取得』を参照してください。

HDBHandle* theHdl = NULL

マルチメディアデータ型を取り扱う場合、HiBaseAPI（HDBHandler）オブジェクトへのポインタを指定します。一般データ型を取り扱う場合、このパラメータは省略できます。

【参照】

◇ ValueCount／ValueSize ／ValueType
◇ HDBHandleクラス;　GetRecord／UpdRecord／DelRecord

２-１-１-６．項目データの情報取得

〔バリュー数の取得〕

<u>**HXRecord:: ValueCount**</u> #include <HRecord.h>

JWord　ValueCount (JWord iid);

【説明】

iidで示す「項目」の「値」の数を取得します。

HiBase は、すべての「項目」に、複数（最大255個）の「値」を格納することができます。FetchItem／UpdateItem／DeleteItem等の関数で任意「値」を対象とする場合、事前に本関数を利用することで、対象項目に幾つの「値」が入っているのかを調べることができます。

同一項目に対して、複数の値（「マルチバリュー」）が格納されている場合は、「2」以上が返されます。

FetchItem／UpdateItem／DeleteItem等の関数についての詳細は、前の『項目データの操作』を参照してください。

【パラメータ】

JWord iid

iidに、値の数を取得する「項目番号」（1～255）を指定します。

【参照】

◇　FetchItem／UpdateItem／DeleteItem

〔バリューサイズの取得〕

HXRecord:: ValueSize #include <HRecord.h>

JWord　ValueSize (JWord iid, JWord idx = 0);

【説明】

iidで示す「項目」の、idxで示す「バリューインデックス」の「値」の、長さ（Byte 単位）を取得します。（標準データ長; 128byte 未満／バイナリデータ長; 16 M Byte 未満が返されます。）

FetchItem／UpdateItem／DeleteItem等の関数で任意「値」を対象とする場合、事前に本関数を利用することで、対象値のサイズを調べることができます。

FetchItem／UpdateItem／DeleteItem等の関数についての詳細は、前の『項目データの操作』を参照してください。

【パラメータ】

JWord iid

iidに、値の長さを取得する「項目番号」（1〜255）を指定します。

JWord idx = 0

iidで示す「項目番号」に、複数の値（「マルチバリュー」）が格納されている場合は、idxに、値の長さを取得する「バリューインデックス」（0〜255）を指定します。

「0」が指定された場合、１番目の値の長さが返されます。

【参照】

◇　FetchItem／UpdateItem／DeleteItem

〔バリュータイプの取得〕

HXRecord:: ValueType #include <HRecord.h>

JWord　ValueType (JWord iid, JWord idx = 0);

【説明】

iidで示す「項目」の、idxで示す「バリューインデックス」の「値」の、「データ型」を取得します。

FetchItem／UpdateItem／DeleteItem等の関数で任意「値」を対象とする場合、事前に本関数を利用することで、対象値の「データ型」を調べることができます。

FetchItem／UpdateItem／DeleteItem等の関数についての詳細は、前の『項目データの操作』を参照してください。

【パラメータ】

JWord iid

iidに、データ型を取得する「項目番号」(1〜255) を指定します。

JWord idx = 0

iidで示す「項目番号」に、複数の値（「マルチバリュー」）が格納されている場合は、idxに、データ型を取得する「バリューインデックス」(0〜255) を指定します。

「0」が指定された場合、１番目の値のデータ型が返されます。

【戻り値】

「HiBase データベース項目のデータ型」の識別子が返されます。
「HiBase データベース項目のデータ型」に関しては、付録「App 1 - HiBaseデータベース項目のデータ型」を参照してください。

【参照】

◇ FetchItem／UpdateItem／DeleteItem
◇ App 1 - HiBaseデータベース項目のデータ型

2-1-2.その他のメンバ関数

「スキーマ変換クラス（HXRecord）」には、アプリケーションプログラムの開発を支援するため、上記関数の他、以下のメンバ関数が用意されています。

2-1-2-1．メモリプール内の情報取得

〔DBRec* オペレータ関数；メモリプール内レコードの取得〕

HRecord:: DBRec* #include <HRecord.h>

operator DBRec*();

【説明】

「スキーマ変換クラス（HRecord）」の「メモリープール」内にある「内部形式レコード」へのポインタを取得します。

「スキーマ変換クラス（HRecord）」で項目データを追加／更新した後、通常は、「***HiBase*** API（HDBHandleクラス）」のInsRecord／UpdRecord関数を呼び出し、***HiBase***の「データベースファイル」に「レコード」を追加／更新します。この際、本関数をパラメータとして利用することができます。

InsRecord／UpdRecord関数についての詳細は、『基本API 関連／HDBHandleクラス』を参照してください。

【参照】

◇ Append／Update
◇ HDBHandleクラス; InsRecord／UpdRecord

〔メモリプール使用レングスの取得〕

HXRecord:: Length #include <HRecord.h>

JWord　Length();

【説明】

「スキーマ変換クラス（HRecord)」の「メモリープール」の使用レングス（Byte単位）を取得します。

「***HiBase*** API（HDBHandleクラス)」のInsRecord／UpdRecord関数実行時、事前に本関数を利用することで、「メモリープール」のサイズを調べることができます。

InsRecord／UpdRecord関数についての詳細は、『基本API 関連／HDBHandleクラス』を参照してください。

【参照】

- ◇ Append／Update
- ◇ HDBHandleクラス;　InsRecord／UpdRecord

２-１-２-２．メモリの初期化

〔メモリプールのクリア〕

HXRecord:: Clear #include <HRecord.h>

```
ECode   Clear ();
```

【説明】

「スキーマ変換クラス（HXRecord)」の「メモリープール」を初期化（クリア）します。

一般的に、アプリケーションプログラムがレコード編集する場合、「スキーマ変換クラス（HXRecord)」で項目データを追加／更新した後、「基本***HiBase*** API（HDBHandleクラス)」のInsRecord／UpdRecord関数を呼び出し、***HiBase***の「データベースファイル」に「レコード」を追加／更新するといった作業を繰り返します。本関数は、こうした「シリアルリユース」を繰り返す際、レコード編集の先頭で実行する関数です。

本関数の実行に先立って、「メモリープール」に展開されているデータや、そのサイズなどを確認する場合は、DBRec ／Length 等の関数を利用します。
また、「メモリープール」内の任意データのみを削除する場合は、Delete関数を利用することができます。

上記関数についての詳細は、後の『項目データの削除 』、『メモリープール内の情報取得 』を参照してください。

【参照】

◇ Append／Update／Delete
◇ DBRec ／Length
◇ HDBHandleクラス; GetRecord／UpdRecord／DelRecord

2-1-3.スキーマ変換フォーマットの文法

「スキーマ変換クラス（HRecord)」のAppend／Fetch／Update／Delete メンバ関数群を利用する場合は、各関数の「第１パラメータ」に「スキーマ変換フォーマット」を指定する必要があります。

「スキーマ変換フォーマット」とは、***HiBase*** を利用するアプリケーションプログラムの要求する「外部データ構造」を、下記「基本フォーマット」に従って、パラメータ群で指示したものです。

以下、「スキーマ変換フォーマット」(基本フォーマット、及び、各パラメータ）について説明します。

基本フォーマット

「スキーマ変換」のパラメータは、以下の基本フォーマットで表現されます。

【基本フォーマット】

fb ：＝　項目指定　＜，　項目指定　＜，　........＞　＞＜ex-terminator＞;

項目指定：＝　［項目番号］％

＜length ＞＜type ＞＜＜terminator＞｜＜ex-terminator ＞＞

【説 明】

「スキーマ変換フォーマット」を指定する場合は、「項目指定」もしくは、「ex-terminator（拡張ターミネータ指定)」をカンマ（，）で区切って並べ、最後にセミコロン（；）を付けます。

「項目指定」は、「項目番号」を 中括弧 （［ ］）で鋏み、その後ろに「編集マーク （%)」を置き、その後ろに、length （データ長)、type （データ型)、terminator （ターミネータ指定)、もしくは、ex-terminator（拡張ターミネータ指定）を並べます。

【例】

以下に、スキーマ変換のパラメータ指定のサンプルを示します。

下記サンプルは、項目番号 1 のデータを、20 バイトの C 言語仕様（null terminate）の文字列として扱うことを表現しています。

【スキーマ変換のパラメータ指定例】

「Append／Fetch／Update／Delete メンバ関数群」にフォーマットを指定する際は、上記「基本フォーマット」に従って指示されたパラメータ群を、ダブルクォーテーション（" "）で括ります。

【パラメータ】

[項目番号]

対象とする「項目番号」（1～255）を指定します。

< length >

対象とする項目の「データ長」（標準データ長; 128byte 未満／バイナリデータ長; 16 M Byte 未満）を指定します。

「terminator（ターミネータ）」を指定する場合は、省略することができます。

< type >

対象とする項目の「データ型識別記号」として、以下のいずれかを指定します。

【データ型識別記号】

```
s    //  hitString （文字列 ／256byte 未満）
n    //  hitNumber（文字列数値 ／256byte 未満）
i    //  hitInteger（バイナリ数値 ／C 言語の long もしくは short）
d    //  hitDate（日付 ／yyyy/mm/dd の形式）
```

t　　// hitTime（時刻 ／hh:mm:ss の形式）
o　　// hitDateTime（日付＆時刻 ／yyyy/mm/dd hh:mm:ss の形式）
c　　// hitChar（文字）
f　　// hitFlag（フラグ）
b　　// hitBinary （不定型）

※ここで指定できるデータ型は、上記の一般データ型に限られます。
マルチメディアデータ型を、スキーマ変換フォーマットで取り扱うことはできません。

＜ terminator ＞

「type （データ型）」が、s（文字列）、n（文字列数字）、d（日付）、t（時刻）、o（日付＆時刻）の場合、以下の「ターミネータ」を指定することができます。

【ターミネータタイプ】

n　　// hitNTerm（null terminator）
　データの最後がヌル文字（バイナリのゼロ）で終了する

t　　// hitTTerm（TAB terminator）
　データの最後が HT（holizontal TAB = 0x09）で終了する

r　　// hitRTerm（return terminator）
　データの最後が CR（carriage return = 0x0d）で終了する

例えば、下図のような、レコードがデータベースに存在して、

YAMADA	TARO	TOKYO	25

これを、以下のようなタブ区切り形式で編集したい場合は、

YAMADA→TARO→TOKYO→25↓

[1] %st, [2] %st, [3] %st, [4] %nr;　のように指定します。
（タブターミネータを指定し、データ長は可変であるため、省略します。）

また、以下のようなデータ構造に編集する場合は、

```
struct {
        char    name[32];
        char    family[32];
        char    address[50];
        int     age;
} Jinji;
```

[1] %32sn, [2] %32sn, [3] %50sn, [4] %2i;　のように指定します。

（C 言語の規則に従ってヌルターミネータを指定し、固定長データであることから、データ長も指定します。）

＜ ex-terminator ＞

「type （データ型）」が、 s（文字列）、n（文字列数字）、d（日付）、t（時刻）、o（日付＆時刻）の場合、レコードの最後を示す「拡張ターミネータ」を指定することができます。

「拡張ターミネータ」を指定する場合は、「拡張ターミネータマーク（ / ）」を置き、その後、「拡張ターミネータ」とする 1 文字、もしくは、「エスケープ（\）x」に続く16進文字（例えば、\x0d はリターンを表わす）を、クォーテーション（ ' ）で囲みます。

【 拡張ターミネータ指定 】

ex-terminator：＝ /'<char>'　もしくは　/ ' \x<hex-char>'

省略形

連続する「項目」の編集指定が同じ場合は、連続する「項目番号」をマイナス記号（-）で繋ぐことで、指定を省略することができます。

【連続項目の省略指定】

[iid]-[jid]　　//　i〜j 番目の項目を同じ内容で指定

以下に、省略形のサンプルを示します。

【省略指定例】

◇　[1] %st, [2] %st, [3] %st, [4] %nr;　を省略
　　→　[1] - [3] %st, [4] %nr;

◇　[1] %32sn, [2] %32sn, [3] %50sn, [4] %2i;　を省略
　　→　[1] - [2] %32sn, [3] %50sn, [4] %2i;

マルチバリュー指定

HiBase は、すべての「項目」に、複数（最大255個）の「値」を格納することができます。同一項目に対して、複数の値（「マルチバリュー」）が格納されている場合に、特定のオカレンスのみを対象とする場合は、項目番号に「バリューインデックス」を付加します。

（マルチバリューにするために、特別な宣言は必要ありません。また、「キー」に指定されている「項目」に、複数の「値」を格納した場合、それぞれの「値」に対して、自動的に「キー」が生成されます。）

【マルチバリュー指定】

[iid(i)]　　//　i 番目のバリューを指定
[iid(i-j)]　　//　i〜j 番目のバリューを指定

[iid(ex-terminator)]　　// 不定回数のマルチバリューを指定

以下に、バリュー指定のサンプルを示します。

【マルチバリュー時のバリュー指定例】

◇　１番項目の４番目のバリューのみを指定
→　[1 (4)]

◇　１番項目の２番目から５番目までのバリューを指定
→　[1 (2 - 5)]

◇　１番項目が、不定回数繰り返されるバリューを指定
（繰り返しの最後にリターン（0x0d）が出現する）
→　[1 (/'\x0d')]

【参照】

◇　スキーマ変換クラス;　HRecord クラス;
／Append／Fetch／Update／Delete

拡張 HiBase API
CLASSES ...

Chapter 2 - 2

スキーマの生成

この章で解説する主要な項目は、次の通りです。

HSchema クラス

「スキーマ生成クラス（HSchema)」は、DDL で記述されたスキーマ定義をHiBaseAPIに渡すデータ構造に翻訳します。

2-2. HSchema クラス

「スキーマ生成クラス（HSchema）」は、DDLで記述されたスキーマ定義をHiBaseAPIに渡すデータ構造に翻訳します。

メンバ関数

MakeSchema... スタティック関数

SchemaTo

【HSchema の機能】

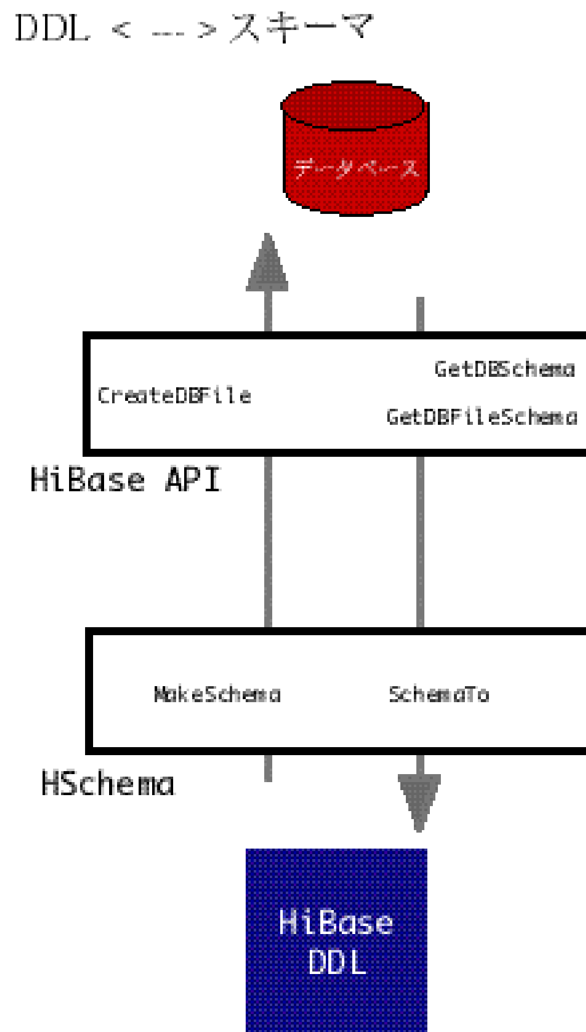
HSchemaクラスは、HiBaseのデータベース定義記述言語（HiBase DDL）を翻訳し、スキーマオブジェクト（HSchema クラスのインスタンス）を作り出します。

HiBaseAPI には、データベースファイルを生成する HDBHandle::CreateDBFile が提供されています。アプリケーションプログラムは、データベースファイルを作り出すとき、HSchema::MakeSchemaでスキーマオブジェクトを作り、作り出したオブジェクトをHDBHandle::CreateDBFileに渡します。

また、HiBaseデータベースに存在するデータベースのスキーマは、HDBHandle::GetDBSchema（データベース全体のスキーマを取りだす）もしくはHDBHandle::GetDBFileSchema（データベースファイルのスキーマを取りだす）で取りだすことができます。この時スキーマオブジェクトが返されるのでこれをHSchemaのメンバー関数であるHSchema::SchemaToを呼びだして、データベース定義記述言語の状態に

戻すことができます。

【 HSchema の概念図 】



２-２-１．スキーマの生成

HSchema ::MakeSchema #include <HSchema.h>

```
static HSchema* MakeSchema (const JByte*  pMem, JLong lMem = 0);
static HSchema* MakeSchema (JByte** hMem, JWord nSeg = 0);
static HSchema* MakeSchema (HFile* inFile);
static HSchema* MakeSchema (HStream* inStream);
```

【説明】

HiBaseの「DDL（データベース定義記述言語）」を翻訳して、スキーマオブジェクト（HSchemaのインスタンス）を作り出します。

DDLは、メモリーへのポインタ、ポインターリスト、ファイル、ストリームで渡すことができます。

【例】

```
char*     argv[] = {
    "[DB], NAME = \"TEST-Database\", DESC = \"testing database\"",
    "[FILE:1], NAME = \"TEST-FILE-1\", LABEL = \"file001\", BLOCKSIZE = (2048, 4096)",
    "[ITEM:1], NAME = \"ID\", TYPE = String, LENGTH = 32",
    "[ITEM:2], NAME = \"DateTime\", TYPE = DateTime, LENGTH = 32",
    "[ITEM:3], NAME = \"File-Name\", TYPE = String, LENGTH = 64",
    "[ITEM:11], NAME = \"File\",    TYPE = File,    LENGTH = 4000",
    "[ITEM:12], NAME = \"MacFile\", TYPE = MacFile, LENGTH = 4000",
    "[ITEM:13], NAME = \"TEXT\",    TYPE = Text,    LENGTH = 4000",
    "[ITEM:14], NAME = \"HTML\",    TYPE = HTML,    LENGTH = 4000",
    "[ITEM:15], NAME = \"PDF\",     TYPE = PDF,     LENGTH = 4000",
    "[ITEM:16], NAME = \"JPEG\",    TYPE = JPEG,    LENGTH = 4000",
    "[ITEM:17], NAME = \"GIF\",     TYPE = GIF,     LENGTH = 4000",
    "[ITEM:18], NAME = \"PNG\",     TYPE = PNG,     LENGTH = 4000",
    "[ITEM:19], NAME = \"PICT\",    TYPE = PICT,    LENGTH = 4000",
    "[ITEM:20], NAME = \"BMP\",     TYPE = BMP,     LENGTH = 4000",
    "[KEY:1], NAME = \"ID\", TYPE = String, BIND = [1]",
    "[KEY:2], NAME = \"DateTime\", TYPE = DateTime, BIND = [2]",
    "[KEY:3], NAME = \"File-Name\", TYPE = String, BIND = [3]",
    NULL
};
HSchema*  theSchema = HSchema::MakeSchema(argv);
```

2 - 2 -2. スキーマから DDL へ

HSchema ::SchemaTO #include <HSchema.h>

```
ECode   SchemaTo(JByte*  pMem, JLong& lMem);
ECode   SchemaTo(HFile* oFile);
ECode   SchemaTo(HStream* oStream);
```

【説明】

スキーマオブジェクト（HSchema クラスのインスタンス）から、「HiBase DDL（データ記述言語）」に逆翻訳します。

「HiBase DDL（データ記述言語）」の格納場所は、メモリ、ファイル、ストリームを指定することができます。

本関数は、「データベース定義」のバックアップを必要とするアプリケーションプログラムを想定しています。

2-2-3.データベース記述言語の文法

以下、「HiBase DDL（データ記述言語）」の文法を説明します。

「HiBase DDL（データ記述言語）」とは、「データベース」、「データベースファイル」、「項目」、「キー」の特性を指定するものです。

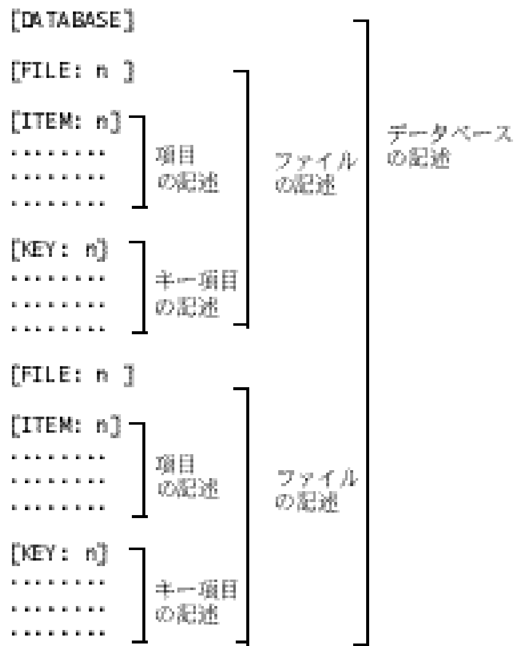
基本フォーマット

「データベース定義ファイル」のパラメータは、以下の基本フォーマットで表現されます。

【基本フォーマット】

```
[DATABASE] , name = " ", desc = " "
 [FILE: n] , name = " ", desc = " ", label = " ", BlockSize = (xxx, xxx)
  [ITEM: n] , name = " ", desc = " ", label = " ", type = , length =
          ..................................................
  [KEY: n] , name = " ", desc = " ", label = " ", type = xx , bind = ...
          ..................................................
```

【基本構造】



【説明】

[DataBase]の行は「データベース」、[File: n]の行は「データベースファイル」、[Item: n]の行は「項目」、[Key: n]の行は「キー項目」の定義をおこないます。

各々の「n 」に、「データベースファイル番号」(1～64000)、「項目番号」(1～255)、「キー番号」(1～250)を指定します。

(上記表記の「... 」は繰り返しを表します。)

[ITEM: n]、[KEY: n]の各行は、「データベースファイル」内に登録されている「項目」及び、「キー項目」の数に応じて繰り返します。

[FILE: n] 以下の行(データベースファイルブロック: [FILE: n]、[ITEM: n]、[KEY: n])は、「データベース」内の「データベースファイル」の数に応じて繰り返します。

【パラメータ】

name = " "

「データベース名」、「データベースファイル名」、「項目名」、「キー名」(64byte 未満)を、ダブルクォーテーション(" ")で括って指定します。
省略時は、ブランクとなります。

desc = " "

「データベースメモ」、「データベースファイルメモ」、「項目メモ」、「キーメモ」(256byte 未満)を、ダブルクォーテーション(" ")で括って指定します。
省略時は、ブランクとなります。

label = " ",

「データベースファイル」、「項目」、「キー項目」には、ラベル(外部参照名称)を付けることができます。これは、外部から参照する名前を「name= ””」で示す名前と別に設定するときに用います。

BlockSize = (xxx, xxx)

「データベースファイル」の「インデックスファイル(@Index)」、及び、「データファイル(@Data:データベースファイルの物理的な存在)」の「物理ブロックサイズ」(byte 単位)をカンマで区切り、括弧で括って ((,)) 指定します。
省略時(ゼロ指定時)は、「2048byte」、「4096byte」が採用されます。

type = xxx

「項目」、「キー」の「データ型」として、以下のいずれかを指定します。

【項目のデータ型】

一般データ型

DateTime	日付＆時間（(yyyy/mm/dd hh:mm:ss の形式）
Date	日付（yyyy/mm/dd の形式）
Time	時間（hh:mm:ss の形式）
String	文字列（256byte 未満）
Number	１０進数字（256byte 未満）
Integer	バイナリ数字（2 bytes もしくは 4 bytes）
Char	文字
Flag	２値（true 、false）
Binary	不定

マルチメディアデータ型

File	標準ファイル
MacFile	マッキントッシュファイル
Text	標準文章
HTML	HTML文章
PDF	PDF文章
JPEG	絵（JPEG）
GIF	絵（GIF）
PNG	絵（PNG）
PICT	絵（PICT）
BMP	絵（BMP）

【キーのデータ型】

String	//文字列（256byte 未満）
Number	// 数値（256byte 未満）
Date	//日付（yyyy/mm/dd の形式）
Time	// 時間（hh:mm:ss の形式）
DateTime	//日付＆時間（(yyyy/mm/dd hh:mm:ss の形式）
Integer	バイナリ数字（2 bytes もしくは 4 bytes）
Char	文字
Flag	２値（true 、false）
Binary	不定

上記データ型に加え、「キー」の「データ型」として「ユニークキー（Unique)」の指定を追加することができます。（上記「データ型」の後に「Unique」を指定します。）

```
//      ユニーク値

        Unique          // ユニークキー
```

「ユニークキー」を指定した場合、InsRecord 関数でレコードを追加する際、追加するキー値の重複を調べ、重複するならば、エラーを返します。つまり、値の重複を排除し、ユニーク値だけからなるデータベースファイルを維持することができます。

length = xxx

「項目」の「データ長」（一般データ型：256byte 未満／マルチメディアデータ型：制限なし）を指定します。

bind = [項目番号] + [項目番号] + ...

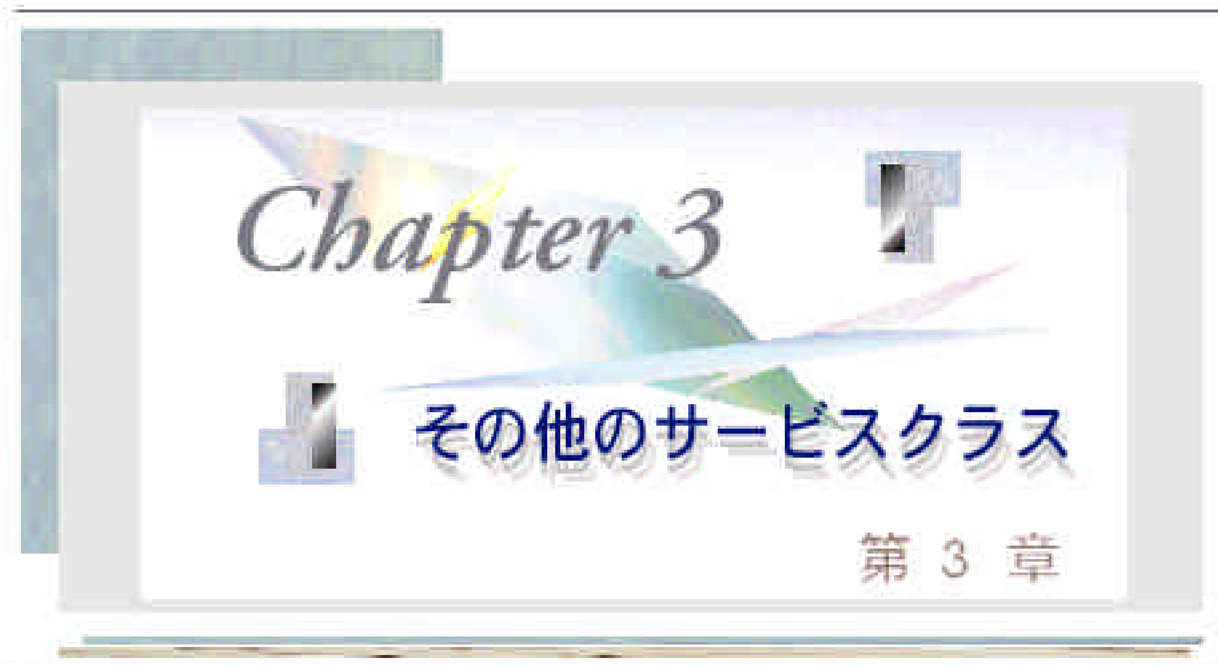
HiBase の「キー」は、最大５つまでの「項目」を連結することができます。
「キー」が作られる元の「項目番号」を、中括弧 （[ ]）で鋏んで指定します。複数の「項目」と連結するする場合は、これを「プラス（＋)」で繋ぎます。

【例】

以下に、「データベース定義ファイル」のサンプルを示します。

【データベース定義ファイル】

```
[DATABASE:1], NAME = "テスト用データベース", DESC = "テスト"
 [FILE:1], NAME = "DBFILE#1", DESC = "練習1", BLOCKSIZE = (2048, 4096)
  [ITEM:1], NAME = "品目", DESC = "名称", TYPE = String, LENGTH = 6
  [ITEM:2], NAME = "個数", DESC = "", TYPE = Number, LENGTH = 6
  [ITEM:3], NAME = "納入先", DESC = "", TYPE = String, LENGTH = 6
  [KEY:1], NAME = " 品目", DESC = "", TYPE = String, BIND = [1]
  [KEY:2], NAME = " 個数", DESC = "", TYPE = Number, BIND = [2]
  [KEY:3], NAME = " 店舗名", DESC = "", TYPE = String, BIND = [3]
 [FILE:2], NAME = "DBFILE#2", DESC = "練習2", BLOCKSIZE = (2048, 4096)
  [ITEM:1], NAME = "資料名", DESC = "", TYPE = String, LENGTH = 16
  [ITEM:2], NAME = "日付", DESC = " 作成日", TYPE = Date, LENGTH = 8
  [ITEM:3], NAME = "内容", DESC = "", TYPE = Text, LENGTH = 32000
  [KEY:1], NAME = " 作成日", DESC = "", TYPE = Date, BIND = [2]
```



この章では、本ドキュメント及び、サンプルプログラム中で利用している、***HiBase*** のアプリケーションプログラム開発に有効な「サービスクラス」、及びその「メンバ関数」について説明します。

「サービスクラス」は、***HiBase*** が利用する「ディスクエレメント（ディレクトリ、ファイル)」や「メモリ」を抽象化し、これに物理的なアクセスを実行するクラスです。
これらには、***HiBase*** の利用環境の構築に使用される「スタティックメンバ関数」等も含まれます。

なお、ストリーム操作を理解することで、プログラム中での、ファイル、動的メモリ（ハンドル、ポインタメモリ)、静的メモリ（スタック内メモリ）等の効率的な管理が可能となります。

この章で解説する主要な項目は、次の通りです。

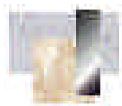
HDir / HFile / HStream クラス

- 開発を支援するその他のメンバ関数...

ディレクトリを表現するクラス（HDir）

ファイルを表現するクラス（HFile）

ファイル／メモリを ストリームに 抽象化するクラス（HStream）



3 - 1 . HDir / HFile / HStream クラス

- 開発を支援するその他のメンバ関数...

「***HiBase*** =クラスライブラリ」は、百を超える「クラス」と数千の「メンバ関数」が定義されている大規模なものですが、「基本 ***HiBase*** API」、「拡張 ***HiBase*** API」の他にも、多数の「クラス」や「メンバ関数」が存在します。

以下、これらの中で特に有効な「サービスクラス」(本ドキュメント及び、サンプルプログラム中で利用しているクラスやメンバ関数)について説明します。

HiBase の「サービスクラス」には、「HDirクラス:ディレクトリを表現するクラス、「HFile クラス:ファイルを表現するクラス」、及び、「HStream クラス:ファイル/メモリを ストリームに 抽象化するクラス」が定義されています。

「HDir(ディレクトリを表現するクラス)」、「HFile(ファイルを表現するクラス)」は、***HiBase*** が利用するディスクエレメントを抽象化し、物理的なアクセスを実行するクラスです。
これらのクラスには、「ディレクトリ(フォルダ)」や「ファイル」を操作する「メンバ関数」、及び、***HiBase*** の運用プログラム等を開発する際、***HiBase*** の利用環境の構築に使用される「スタティックメンバ関数」等が定義されています。

「HStream (ファイル/メモリを ストリームに 抽象化するクラス)」は、ファイル、動的メモリ(ハンドル、ポインタメモリ)、静的メモリ(スタック内メモリ)等を「ストリーム」に抽象化して表現する「抽象的(abstruction)基本クラス」です。
アプリケーションプログラムがこれらを生成する場合、「HStream クラス」のインスタンス(オブジェクト)を直接生成することはありませんが、「HFileStream(ファイルストリーム)」、「HROMStream(読出だけに利用するメモリのストリーム)」、「HDMemStream(動的メモリのストリーム)」、「HSMemStream(静的メモリのストリーム)」等オブジェクトを生成することで、これらの読み出し/書き込み操作が可能となります。

メンバ関数

HDir ::

IsExsit	
Create	Erase
IsExsitDir	IsExsitFile
Dir_Size	Dir_Fetch
CreateDir	CreateFile
EraseDir	EraseFile
Dir_AttachAll	
SystemDir	ApplicationDir
CurrentDir	HolonDir

HFile ::

IsExsit	IsOpen
Create	Erase
Open	Close
Flush	
GetFileSize	SetFileSize
GetFileSign	SetFileSign
GetFileTOD	SetFileTOD
GetFileVisible	SetFileVisible
GetFileType	
Copy	Move
GetPosition	SetPosition
Read	Write
OpenReadStream	OpenWriteStream
FlushStream	

HStream ::

GetSize	IsMore
GetPosition	SetPosition
ReadByte	ReadBytes
ReadIntger	ReadLong
ReadString	ReadCString
ReadPString	ReadLine
UnRead	
WriteByte	WriteBytes
WriteIntger	WriteLong
WriteString	WriteCString
WritePString	WriteLine
WriteTerminal	

3-1-1.ディレクトリを表現するクラス

「HDir：ディレクトリを表現するクラス」は、任意「ディレクトリオブジェクト（フォルダ）」を基準に、これと、これの管理する「ディスクエレメント（ディレクトリ、ファイル）」を表現するクラスです。

本クラスには、「ディレクトリオブジェクト（フォルダ）」やその管理下の「ファイルオブジェクト」を操作する「メンバ関数」、及び、***HiBase***が利用環境として使用する「ディレクトリ」を返す「スタティックメンバ関数」等が定義されています。

3-1-1-1．コンストラクタ／デストラクタ

〔HDirクラスの生成（コンストラクタ）〕

HDir :: #include <HDir.h>

HDir (HDir* theMother, const JByte* theName);

〔HDirクラスの削除（デストラクタ）〕

HDir :: #include <HDir.h>

HDir ();

【説明】

「HDir：ディレクトリを表現するクラス」のインスタンス（オブジェクト）を生成、或いは、削除します。

アプリケーションプログラムが本クラスを利用する場合は、まず「HDirクラス」のインスタンス（オブジェクト）を生成する必要があります。（この際、本クラスが対象とする「ディレクトリオブジェクト（フォルダ）」を特定します。）
その後、目的のメンバ関数を呼び出し、最後に、このインスタンスを削除するスタイルを取ります。

【パラメータ】

HDir* theMother

theMotherに、本クラスが対象とする「ディレクトリオブジェクト（フォルダ）」の親ディレクトリを指定します。

本パラメータに指定された親ディレクトリが存在しない場合は、「エラーコード」が返され、本クラスは生成されません。

const JByte* theName

theNameに、本クラスが対象とする（Create 関数で生成する）「ディレクトリオブジェクト名（フォルダ名）」を指定します。

「ディレクトリオブジェクト（フォルダ）」の生成については、後の『対象ディレクトリの作成／削除 ; Create 』を参照してください。

なお、本パラメータに指定された名称が、先に指示された親ディレクトリ内で既に使用されている場合は、Create 関数実行時、「エラーコード」が返され、新しいディレクトリは生成されません。

３-１-１-２．対象ディレクトリ自身の操作

本クラスが対象とする「ディレクトリオブジェクト（フォルダ）」自身を操作するメンバ関数は、以下の通りです。

存在の取得

〔対象ディレクトリ有無の取得〕

HDir:: IsExist #include <HDir.h>

JBool IsExist() const;

【説明】

対象とする「ディレクトリオブジェクト（フォルダ）」の存在有無を取得します。

本関数を利用することで、既に対象「ディレクトリ（フォルダ）」が存在するかどうかを調べることができます。
存在しない場合、Create 関数を呼び出し、「ディレクトリ（フォルダ）」の生成が可能です。

「ディレクトリオブジェクト（フォルダ）」の生成については、後の『対象ディレクトリの作成／削除 ; Create 』を参照してください。

【戻り値】

戻り値として、「true」または、「false」が返されます。

◇ 対象「ディレクトリ」が存在する場合 = TRUE
◇ 対象「ディレクトリ」が存在しない場合 = FALSE

【参照】

◇ HDir クラス; Create ／Erase
／スタティック関数（***HiBase*** が利用するディレクトリの操作）

対象ディレクトリの作成／削除

〔対象ディレクトリの作成〕

HDir:: Create #include <HDir.h>

ECode Create();

〔対象ディレクトリの削除〕

HDir:: Erase #include <HDir.h>

ECode Erase ();

【説明】

対象とする「ディレクトリオブジェクト（フォルダ)」を生成、或いは、削除します。

なお、本関数の実行に先立って「HDir クラス:: IsExist 関数」を呼び出すことで、対象「ディレクトリ（フォルダ)」の存在有無を事前に確認することができます。

IsExist 関数についての詳細は、前の『存在の取得 ; IsExist 』を参照してください。

【参照】

◇ HDir クラス; IsExist
／スタティック関数（***HiBase*** が利用するディレクトリの操作）

３-１-１-３．対象ディレクトリ内のディレクトリ／ファイルの操作

対象ディレクトリ（フォルダ）の管理下にある「ディレクトリオブジェクト（フォルダ)」又は、「ファイルオブジェクト」を操作するメンバ関数は、以下の通りです。

存在の取得

〔ディレクトリ有無の取得〕

HDir:: IsExistDir #include <HDir.h>

JBool IsExistDir (const JByte* theName);

〔ファイル有無の取得〕

HDir:: IsExistFile #include <HDir.h>

JBool IsExistFile (const JByte* theName);

【説明】

対象とする「ディレクトリオブジェクト（フォルダ)」内の、theName で示す「ディレクトリオブジェクト（フォルダ)」又は、「ファイルオブジェクト」の存在有無を取得します。

本関数を利用することで、対象「ディレクトリ（フォルダ)」内に、任意「ディレクトリ（フォルダ)」又は、「ファイル」が存在するかどうかを調べることができます。（存在しない場合、「HDir クラス:: CreateDir 又は、CreateFile 関数」を呼び出すことで、これらの生成が可能です。）

なお、存在の確認されたオブジェクトを、対象「ディレクトリ（フォルダ)」内に登録する場合は「HDir クラス:: Dir_AttachAll 関数」を利用します。
また、登録されているオブジェクト自体を取得する場合は「HDir クラス:: Dir_Fetch 関数」を利用します。

「ディレクトリ（フォルダ)」又は、「ファイル」の生成については『対象ディレクトリ内への作成 ; CreateDir, CreateFile 』を、オブジェクトの登録については『対象ディレクトリ内への登録; Dir_AttachAll 』を、オブジェクトの取得については『存在の取得; Dir_Fetch 』を参照してください。

【パラメータ】

const JByte* theName

theNameに、存在有無を取得する「ディレクトリオブジェクト名（フォルダ名)」又は、「ファイルオブジェクト名」を指定します。

【戻り値】

戻り値として、「true」または、「false」が返されます。

◇ 「ディレクトリ」又は「ファイル」が存在する場合　＝　TRUE
◇ 「ディレクトリ」又は「ファイル」が存在しない場合　＝　FALSE

【参照】

◇ HDir クラス;　CreateDir ／CreateFile ／Dir_Fetch／Dir_AttachAll
／スタティック関数（***HiBase*** が利用するディレクトリの操作）

〔登録ディレクトリ、ファイル数の取得〕

HDir:: Dir_Size　#include <HDir.h>

JLong　Dir_Size () const;

【説明】

対象とする「ディレクトリオブジェクト（フォルダ)」内に登録されている、「ディレクトリオブジェクト（フォルダ)」及び、「ファイルオブジェクト」の総数を取得します。

本関数を利用することで、対象「ディレクトリ（フォルダ)」が管理する「ディレクトリ（フォルダ)」及び、「ファイル」の総数を調べることができます。

なお、「ディレクトリ（フォルダ)」及び、「ファイル」を、対象「ディレクトリ（フォルダ)」内に登録する場合は「HDir クラス:: Dir_AttachAll 関数」を利用します。

「ディレクトリ（フォルダ)」又は、「ファイル」の登録については、後の『対象ディレクトリ内への登録; Dir_AttachAll 』を参照してください。

【参照】

◇ HDir クラス; Dir_AttachAll
／スタティック関数（*HiBase* が利用するディレクトリの操作）

〔登録ディレクトリ、ファイルの取得〕

HDir:: Dir_Fetch #include <HDir.h>

HDirElement* Dir_Fetch (JLong idx) const;

【説明】

対象とする「ディレクトリオブジェクト（フォルダ）」内に登録されている、idx で示す「ディレクトリオブジェクト（フォルダ）」又は、「ファイルオブジェクト」を取得します。

「ディレクトリ（フォルダ）」及び、「ファイル」を、対象「ディレクトリ（フォルダ）」内に登録する場合は「HDir クラス:: Dir_AttachAll 関数」を利用します。

「ディレクトリ（フォルダ）」又は、「ファイル」の登録については、後の『対象ディレクトリ内への登録; Dir_AttachAll 』を参照してください。

【パラメータ】

JLong idx

idx に、取得する「ディレクトリオブジェクト（フォルダ）」又は、「ファイルオブジェクト」の「インデックス番号」を指定します。

【参照】

◇ HDir クラス; Dir_AttachAll
／スタティック関数（*HiBase* が利用するディレクトリの操作）

対象ディレクトリ内への作成

〔ディレクトリの作成〕

HDir:: CreateDir #include <HDir.h>

JBool　CreateDir (const JByte* theName);

〔ファイルの作成〕

HDir:: CreateFile #include <HDir.h>

JBool　CreateFile (const JByte* theName);

【説 明】

対象とする「ディレクトリオブジェクト（フォルダ）」内に、theName で示す「ディレクトリオブジェクト（フォルダ）」又は、「ファイルオブジェクト」を生成します。

なお、本関数の実行に先立って「HDir クラス:: IsExistDir 又は、IsExistFile 関数」を呼び出すことで、これらの存在有無を事前に確認することができます。

また、生成後、これらを対象「ディレクトリ（フォルダ）」内に登録する場合は「HDir クラス:: Dir_AttachAll 関数」を利用します。

IsExistDir 又は、IsExistFile 関数についての詳細は『存在の取得 ; IsExistDir, IsExistFile 』を、オブジェクトの登録については『対象ディレクトリ内への登録; Dir_AttachAll 』を参照してください。

【パラメータ】

const JByte* theName

theNameに、生成する「ディレクトリオブジェクト名（フォルダ名）」又は、「ファイルオブジェクト名」を指定します。

なお、本パラメータに指定された名称が、先に指示された親ディレクトリ内で既に使用されている場合は、「エラーコード」が返され、新しいオブジェクトは生成されません。

【参 照】

◇ HDir クラス;　IsExistDir ／IsExistFile ／Dir_AttachAll
／スタティック関数（***HiBase*** が利用するディレクトリの操作）

対象ディレクトリからの削除

〔ディレクトリの削除〕

HDir:: EraseDir #include <HDir.h>

JBool　EraseDir (const JByte* theName);

〔ファイルの削除〕

HDir:: EraseFile #include <HDir.h>

JBool　EraseFile (const JByte* theName);

【説明】

対象とする「ディレクトリオブジェクト（フォルダ）」内の、theName で示す「ディレクトリオブジェクト（フォルダ）」又は、「ファイルオブジェクト」を削除します。

なお、本関数の実行に先立って「HDir クラス:: IsExistDir 又は､IsExistFile 関数」を呼び出すことで、これらの存在有無を事前に確認することができます。

IsExistDir 又は､IsExistFile 関数についての詳細は、前の『存在の取得 ; IsExistDir, IsExistFile 』を参照してください。

【パラメータ】

const JByte* theName

theNameに、削除する「ディレクトリオブジェクト名（フォルダ名）」又は、「ファイルオブジェクト名」を指定します。

【参照】

◇　HDir クラス;　IsExistDir ／IsExistFile ／CreateDir／CreateFile
／スタティック関数（***HiBase*** が利用するディレクトリの操作）

対象ディレクトリ内への登録

〔ディレクトリ、ファイルの登録〕

HDir:: CreateDir #include <HDir.h>

JLong　Dir_AttachAll ();

【説明】

対象とする「ディレクトリオブジェクト（フォルダ)」内にある、全ての「ディレクトリオブジェクト（フォルダ)」及び、「ファイルオブジェクト」を、対象「ディレクトリ」に登録します。

本関数は、対象「ディレクトリ（フォルダ)」内に存在する全オブジェクトを、対象「ディレクトリ（フォルダ)」の管理下に置くものです。

本関数の実行に先立って「HDir クラス:: IsExistDir 又は、IsExistFile 関数」を呼び出すことで、「ディレクトリ（フォルダ)」及び、「ファイル」の存在有無を事前に確認することができます。
また、「HDir クラス:: Dir_Size 関数」を呼び出すことで、既に登録されているオブジェクトの総登録数を確認することができます。

なお、登録されたオブジェクトを取得する場合は「HDir クラス:: Dir_Fetch 関数」を利用します。

IsExistDir 又は、IsExistFile 関数についての詳細は『存在の取得 ; IsExistDir, IsExistFile 』を、オブジェクトの登録数については『存在の取得; Dir_Size 』を、オブジェクトの取得については『存在の取得; Dir_Fetch 』を参照してください。

【参照】

◇ HDir クラス;　IsExistDir ／IsExistFile ／Dir_Size ／Dir_Fetch
／スタティック関数（***HiBase*** が利用するディレクトリの操作）

3-1-1-4．スタティック関数 （*HiBase* が利用するディレクトリの操作）

HiBase の利用環境を構築する際に使用する「ディレクトリ（フォルダ）」を対象とする「スタティックメンバ関数」は、以下の通りです。

〔システムフォルダの取得〕

HDir:: SystemDir #include <HDir.h>

static HDir* SystemDir ();

〔実行プログラムの存在ディレクトリの取得〕

HDir:: ApplicationDir #include <HDir.h>

static HDir* ApplicationDir ();

〔カレントディレクトリの取得〕

HDir:: CurrentDir #include <HDir.h>

static HDir* CurrentDir ();

〔ホロンフォルダの取得〕

HDir:: HolonDir #include <HDir.h>

static HDir* HolonDir ();

【説明】

HiBase が利用環境として使用する「ディレクトリ（フォルダ）」を、本クラスが定義する「ディレクトリオブジェクト（フォルダ）」で表現し、返します。

順に、「システムフォルダ」、現在実行中のプログラムが存在するディレクトリ（フォルダ）、現在カレントなディレクトリ（フォルダ）、「ホロンフォルダ（システムフォルダの第１階層）」を対象とします。

本関数は、***HiBase*** の運用プログラム等を開発する際、***HiBase*** の利用環境の構築に利用されます。
なお、上記ディレクトリは、***HiBase*** の「環境設定ファイル」や「ロード／アンロードファイル（データベース定義ファイル）」保存時のパラメータ指定に利用されます。

【参照】

◇ HBase, HNetClient／HSchemaLoadUnloadクラス; コンストラクタ

3-1-2.ファイルを表現するクラス

「HFile：ファイルを表現するクラス」は、任意「ファイルオブジェクト」を抽象化し、物理的なファイルアクセスを実行するクラスです。

本クラスは、***HiBase*** の物理的なファイルアクセスを、最も深いレベルで抽象化しており、Macintosh／Windows／UNIX の全プラットフォームに対し（一部の例外を除き）アクセスの共通化を実現しています。

本クラスを利用する場合は、「コンストラクタ」で対象ファイルを特定後、目的の「メンバ関数」を利用して、ファイルへのアクセスを実行します。

3-1-2-1. コンストラクタ ／ デストラクタ

〔HFile クラスの生成（コンストラクタ）〕

HFile :: #include <HFile.h>

HFile (HDir* theMother, const JByte* theName);

〔HFile クラスの削除（デストラクタ）〕

HFile :: #include <HFile.h>

HFile ();

【説明】

「HFile：ファイルを表現するクラス」のインスタンス（オブジェクト）を生成、或いは、削除します。

アプリケーションプログラムが本クラスを利用する場合は、まず「HFile クラス」のインスタンス（オブジェクト）を生成する必要があります。
この際、本クラスが対象とする「ファイルオブジェクト」を特定し、その親となる「ディレクトリ（ HDir クラスのオブジェクト)」をパラメータで通知するスタイルを取ります。

【パラメータ】

HDir* theMother

theMotherに、本クラスが対象とする「ファイルオブジェクト」の親ディレクトリ（ HDir クラスのオブジェクト）を指定します。

「ディレクトリ（ HDir クラスのオブジェクト)」の生成については、前の『ディレクトリを表現するクラス ; HDir クラス 』を参照してください。

なお、本パラメータに指定された親ディレクトリが存在しない場合は、「エラーコード」が返され、本クラスは生成されません。

const JByte* theName

theNameに、本クラスが対象とする（Create 関数で生成する）「ファイルオブジェクト名」を指定します。

「ファイルオブジェクト」の生成については、後の『対象ファイルの作成／削除 ; Create 』を参照してください。

なお、本パラメータに指定された名称が、先に指示された親ディレクトリ内で既に使用されている場合は、Create 関数実行時、「エラーコード」が返され、新しいファイルは生成されません。

【参照】

◇ HDir クラス; Create
◇ HFile クラス; Create

３-１-２-２．対象ファイル自身の操作

本クラスが対象とする「ファイルオブジェクト」自身を操作するメンバ関数は、以下の通りです。

状態の取得

〔対象ファイル有無の取得〕

HFile:: IsExist #include <HFile.h>

JBool　IsExist() const;

【説明】

対象とする「ファイルオブジェクト」の存在有無を取得します。

本関数を利用することで、既に対象「ファイル」が存在するかどうかを調べることができます。
存在しない場合、Create 関数を呼び出し、「ファイルオブジェクト」の生成が可能です。

「ファイルオブジェクト」の生成については、後の『対象ファイルの作成／削除 ; Create』を参照してください。

【戻り値】

戻り値として、「true」または、「false」が返されます。

◇　対象「ファイル」が存在する場合　＝　TRUE
◇　対象「ファイル」が存在しない場合　＝　FALSE

【参照】

◇　HFile クラス;　Create ／Erase

〔対象ファイルオープン状態の取得〕

HFile:: IsOpen #include <HFile.h>

JBool IsOpen() const;

【説明】

対象とする「ファイルオブジェクト」のオープン状態を取得します。

本関数を利用することで、対象「ファイル」が既にオープンしているかどうかを調べることができます。
オープンしていない場合、Open 関数を呼び出し、「ファイルオブジェクト」のオープンが可能です。

「ファイルオブジェクト」のオープンについては、後の『対象ファイルのオープン／クローズ ; Open 』を参照してください。

【戻り値】

戻り値として、「true」または、「false」が返されます。

◇ 対象「ファイル」がオープンしている場合 = TRUE
◇ 対象「ファイル」がオープンしていない場合 = FALSE

【参照】

◇ HFile クラス; Create ／Erase／Open／Close

対象ファイルの作成／削除

〔対象ファイルの作成〕

HFile:: Create #include <HFile.h>

```
ECode  Create();
```

〔対象ファイルの削除〕

HFile:: Erase #include <HFile.h>

```
ECode  Erase ();
```

【説 明】

対象とする「ファイルオブジェクト」を生成、或いは、削除します。

なお、本関数の実行に先立って「HFile クラス:: IsExist 又は、IsOpen 関数」を呼び出すことで、対象「ファイル」の存在有無や、オープン状態を事前に確認することができます。

IsExist 又は、IsOpen 関数についての詳細は、前の『存在の取得 ; IsExist, IsOpen 』を参照してください。

【参 照】

◇ HFile クラス; IsExist／IsOpen

対象ファイルのオープン／クローズ

〔対象ファイルのオープン〕

HFile:: Open #include <HFile.h>

ECode　Open (JBool readOnly = false);

〔対象ファイルのクローズ〕

HFile:: Close #include <HFile.h>

ECode　Close ();

【説明】

対象とする「ファイルオブジェクト」をオープン、或いは、クローズします。

Open 関数には、読み出し／書き込みの両者を許すオープン（readOnly = false)、読み出しのみを許すオープン（readOnly = true）の２つオープンタイプがあります。

なお、ファイルの読み出し／書き込みを実行する場合は、本関数実行後、「HFile クラス:: Read 又は、Write 関数」を利用します。
ファイルの読み出し／書き込みについての詳細は、後の『対象ファイルの読み出し／書き出し；Read, Write 』を参照してください。

本関数の実行に先立って「HFile クラス:: IsExist 又は、IsOpen 関数」を呼び出すことで、対象「ファイル」の存在有無や、オープン状態を事前に確認することができます。
IsExist 又は、IsOpen 関数についての詳細は、前の『存在の取得 ; IsExist, IsOpen 』を参照してください。

【パラメータ】

JBool readOnly = true or false

readOnly = に、オープンタイプとして、「true」または、「false」を指定します。

◇ 読み出しのみを許す場合 = TRUE
◇ 読み出し／書き込みの両者を許す場合 = FALSE

【参照】

◇ HFile クラス; IsExist／IsOpen／Read／Write

対象ファイルのフラッシュ

〔対象ファイルのフラッシュ〕

HFile:: Flush #include <HFile.h>

ECode　Flush ();

【説明】

対象とする「ファイルオブジェクト」の保持する「キャッシュ」をフラッシュします。

本関数は、対象「ファイル」のメモリ内に残っている書き出しデータを、ディスクに反映するものです。

なお、ファイルの読み出し／書き込みを実行する場合は、「HFile クラス:: Read 又は、Write 関数」を利用します。

ファイルの読み出し／書き込みについての詳細は、後の『対象ファイルの読み出し／書き出し；Read, Write』を参照してください。

【参照】

◇　HFile クラス;　Read／Write

対象ファイルの情報の取得／設定

〔ファイルサイズの取得〕

HFile:: GetFileSize #include <HFile.h>

ECode　GetFileSize (JLong& size);

〔ファイルサイズの設定〕

HFile:: SetFileSize #include <HFile.h>

ECode　SetFileSize (JLong　size);

【説明】

対象とする「ファイルオブジェクト」のサイズを、取得、或いは、設定します。

GetFileSize 関数は、ファイルサイズ（Byte 単位）を size に返します。
また、SetPosition 関数は、size で示すファイルサイズに設定します。

なお、ファイルの読み出し／書き込みを実行する場合は、「HFile クラス:: Read 又は、Write 関数」を利用します。

ファイルの読み出し／書き込みについての詳細は、後の『対象ファイルの読み出し／書き出し；Read, Write』を参照してください。

【パラメータ】

JLong& size

size に、設定するファイルサイズ（Byte 単位）を指定します。

【参照】

◇　HFile クラス;　Read／Write

〔クリエータ&ファイルタイプの取得〕

HFile:: GetFileSign #include <HFile.h>

ECode GetFileSign(ULong& fSign, ULong& fType) const;

〔クリエータ&ファイルタイプの設定〕

HFile:: SetFileSign #include <HFile.h>

ECode SetFileSign (ULong fSign, ULong fType);

【説明】

対象とする「ファイルオブジェクト」の、fSign で示す「クリエータ」と、fType で示す「ファイルタイプ」を、取得、或いは、設定します。

なお、ファイルの読み出し／書き込みを実行する場合は、「HFile クラス:: Read 又は、Write 関数」を利用します。

ファイルの読み出し／書き込みについての詳細は、後の『対象ファイルの読み出し／書き出し；Read, Write』を参照してください。

【パラメータ】

ULong& fSign

fSign に、設定する「クリエータ」を指定します。

ULong& fType

fType に、設定する「ファイルタイプ」を指定します。

なお、***HiBase*** では、以下のクリエータ、ファイルタイプを利用します。

【クリエータ＆ファイルタイプ】

```
sign_SimpleText     'ttxt'     // simple text file
sign_CodeWarrior    'CWIE'     // code warrior file
```

【参照】

◇ HFile クラス; Read／Write

〔作成日時＆修正日時の取得〕

HFile:: GetFileTOD #include <HFile.h>

ECode GetFileTOD(ULong& cTOD,ULong& mTOD)const;

〔作成日時＆修正日時の設定〕

HFile:: SetFileTOD #include <HFile.h>

ECode SetFileTOD (ULong cTOD, ULong mTOD);

【説明】

対象とする「ファイルオブジェクト」の、cTOD で示す「作成日時」と、mTOD で示す「修正日時」を、取得、或いは、設定します。

なお、ファイルの読み出し／書き込みを実行する場合は、「HFile クラス:: Read 又は、Write 関数」を利用します。

ファイルの読み出し／書き込みについての詳細は、後の『対象ファイルの読み出し／書き出し；Read, Write』を参照してください。

【パラメータ】

ULong& cTOD

cTOD に、設定する「作成日時」を指定します。

ULong& mTOD

mTOD に、設定する「修正日時」を指定します。

【作成日時＆修正日時】

yyyy/mm/dd hh:mm:ss // 日付＆時刻の形式

【参照】

◇ HFile クラス; Read／Write

〔表示／非表示タイプの取得〕

HFile:: GetFileVisible #include <HFile.h>

JBool　GetFileVisible() const;

〔表示／非表示タイプの設定〕

HFile:: SetFileVisible #include <HFile.h>

void　SetFileVisible(JBool visFlag);

【説 明】

対象とする「ファイルオブジェクト」の表示／非表示タイプを、取得、或いは、設定します。

GetFileVisible関数は、「表示ファイル = true」或いは、「非表示ファイル = false」のいずれかを返します。
また、SetFileVisible 関数は、visFlag で示すタイプに設定します。

なお、ファイルの読み出し／書き込みを実行する場合は、「HFile クラス:: Read 又は、Write 関数」を利用します。

ファイルの読み出し／書き込みについての詳細は、後の『対象ファイルの読み出し／書き出し； Read, Write 』を参照してください。

【パラメータ】

JBool visFlag

visFlag に、表示／非表示タイプとして、「true」または、「false」を指定します。

◇ 対象「ファイル」を「表示ファイル」にする場合 = TRUE
◇ 対象「ファイル」を「非表示ファイル」にする場合 = FALSE

【戻り値】

戻り値として、「true」または、「false」が返されます。

【参 照】

◇ HFile クラス; Read／Write

〔MIME タイプの取得〕

HFile:: GetFileType #include <HFile.h>

JWord　GetFileType (JByte* theType) const;

【説明】

対象とする「ファイルオブジェクト」の、MIME タイプを取得します。

なお、ファイルの読み出し／書き込みを実行する場合は、「HFile クラス:: Read 又は、Write 関数」を利用します。

ファイルの読み出し／書き込みについての詳細は、後の『対象ファイルの読み出し／書き出し； Read, Write 』を参照してください。

【戻り値】

JByte* theType

theType に、以下の「MIME タイプ」が返されます。

```
mime_Text_Plain    = 1,     //  plain
mime_Text_HTML     = 2,     //  html
mime_Text_rtf      = 3,     //  rich text
mime_Text_tsv      = 4,     //  tab separated value
mime_Image_gif     = 101,   //  gif
mime_Image_jpeg    = 102,   //  jpeg
mime_Image_tiff    = 103,   //  tiff
mime_Appl_octet    = 200,   //  default application
mime_Appl_java     = 201,   //  java applet
mime_Appl_pdf      = 202    //  acrobat file
```

【参照】

◇ HFile クラス;　Read／Write

３-１-２-３．対象ファイルへの操作

本クラスが対象とする「ファイルオブジェクト」へ、任意「ファイルオブジェクト」をコピー、又は、移動するメンバ関数が定義されています。

対象ファイルへコピー／移動

〔対象ファイルにコピー〕

HFile:: Copy #include <HFile.h>

ECode　Copy (HFile* theFile);

〔対象ファイルへ移動〕

HFile:: Move #include <HFile.h>

ECode　Move (HFile* theFile);

【説明】

theFile で示す「ファイルオブジェクト」を、対象とする「ファイルオブジェクト」にコピー、或いは、移動します。

Move 関数は、コピー後、theFileで示すオリジナルの「ファイルオブジェクト」を削除します。

なお、本関数の実行に先立って「HFile クラス:: GetFileメンバ関数群」を利用することで、任意「ファイルオブジェクト」の情報（サイズ、クリエータ、ファイルタイプ、作成日時、修正日時、表示状態、MIME タイプ等）を事前に確認することができます。

「ファイルオブジェクト」の情報についての詳細は、前の『対象ファイルの情報の取得／設定; GetFileSize, GetFileSign, GetFileTOD, GetFileVisible, GetFileType』を参照してください。

【パラメータ】

HFile* theFile

theFile に、コピー、或いは、移動する「ファイルオブジェクト」を指定します。

【参照】

◇　HFile クラス;　GetFileSize／GetFileSign／GetFileTOD
／GetFileVisible／GetFileType

３-１-２-４．対象ファイルのアクセス操作

本クラスが対象とする「ファイルオブジェクト」に対し、物理的なアクセスを実行する場合は、以下のメンバ関数を利用します。

アクセス位置の取得／設定

〔アクセス位置の取得〕

HFile:: GetPosition #include <HFile.h>

ECode　GetPosition (JLong& pos) const;

〔アクセス位置の設定〕

HFile:: SetPosition #include <HFile.h>

ECode　SetPosition (JLong& pos) const;

【説明】

対象とする「ファイルオブジェクト」の現在のアクセス位置を、取得、或いは、設定します。

GetPosition 関数は、現在のアクセス位置をpos に返します。
また、SetPosition 関数は、現在のアクセス位置を、pos で示す位置に設定します。

なお、ファイルの読み出し／書き込みを実行する場合は、「HFile クラス:: Read 又は、Write 関数」を利用します。

ファイルの読み出し／書き込みについての詳細は、後の『対象ファイルの読み出し／書き出し；Read, Write』を参照してください。

【パラメータ】

JLong pos

pos に、設定するアクセス位置（Byte 単位）を指定します。

【参照】

◇　HFile クラス;　Read／Write

対象ファイルの読み出し／書き込み

〔ファイルの読み出し〕

HFile:: Read #include <HFile.h>

ECode　Read (void* p, JLong len) const;

〔ファイルの書き込み〕

HFile:: Write #include <HFile.h>

ECode　Write (const void* p, JLong len) const;

【説明】

対象とする「ファイルオブジェクト」に対し、読み出し／書き込みを実行します。

Read 関数は、現在のアクセス位置からlen バイト分を、pで示すメモリに読み出ます。
また、Write 関数は、pで示すメモリから、現在のアクセス位置にlenバイト分を書き込みます。

「ファイルオブジェクト」には、読み出し／書き込みの両者を許すタイプ、及び、読み出しのみを許すタイプ（readOnly ）の２つオープンタイプがあります。そのため、本関数の実行に先立って「HFile クラス:: Open 関数」を呼び出し、いずれかのタイプを指定し、対象ファイルをオープンしておく必要があります。
(「HFile クラス:: IsExist　又は､IsOpen 関数」を呼び出すことで、対象ファイルの存在有無や､オープン状態の確認が可能です｡)

Open 関数、IsExist , IsOpen 関数についての詳細は、前の『対象ファイルのオープン／クローズ；Open 』又は、『存在の取得 ; IsExist, IsOpen 』を参照してください。

なお、本関数の実行に先立って「HFile クラス:: GetPosition, SetPosition 関数、又は、GetFileメンバ 関数群」を利用することで、対象ファイルの現在のアクセス位置、及び、ファイルの情報（サイズ、クリエータ、ファイルタイプ、作成日時、修正日時､表示状態､MIME タイプ等）を事前に確認、又は設定することができます。

アクセス位置の取得については『アクセス位置の取得／設定; GetPosition , SetPosition 』を、「ファイルオブジェクト」の情報についての詳細は『対象ファイルの情報 の取得／設定; GetFileSize, GetFileSign, GetFileTOD, GetFileVisible, GetFileType 』を参照してください。

【パラメータ】

void* p

p に、読み出す、或いは、書き出すメモリへのポインタを指定します。

JLong len

len に、読み出す、或いは、書き出すデータの長さ（Byte 単位）を指定します。

【参照】

◇　HFile クラス;　Open／IsExist／IsOpen
／GetPosition／SetPosition
／GetFileSize／GetFileSign／GetFileTOD
／GetFileVisible／GetFileType

3-1-2-5．対象ファイルをアクセスするためのストリーム操作

本クラスが対象とする「ファイルオブジェクト」をアクセスするための「入出力ストリーム」を操作する場合は、以下のメンバ関数を利用します。

入出力ストリームの生成

〔入力ストリームの生成〕

HFile:: OpenReadStream #include <HFile.h>

HStream*　OpenReadStream();

〔出力ストリームの生成〕

HFile:: OpenWriteStream #include <HFile.h>

HStream*　OpenWriteStream();

【説明】

対象とする「ファイルオブジェクト」に対し、「HFileStream（ファイルストリームクラス)」のインスタンス（オブジェクト）を生成します。

本関数は、対象ファイルを「ストリーム」に抽象化し、物理的なアクセスを実行するため、「HStream：ファイル／メモリを ストリームに 抽象化するクラス」の派生クラスである「HFileStream（ファイルストリームクラス)」から、「HStream オブジェクト（「ストリーム」の基本クラス)」を生成するものです。

「HFileStream オブジェクト（ファイルストリーム)」には、書き込み用の「入力ストリーム」と、読み出し用の「出力ストリーム」の２つタイプがあります。OpenReadStream関数は「入力ストリーム」を、OpenWriteStream関数は「出力ストリーム」を生成します。

なお、本関数で生成された「HFileStream オブジェクト（ファイルストリーム)」へのアクセス（ストリームの読み出し／書き込み等）を実行する場合は、本関数実行後、「HStream クラス:: Readメンバ関数群、 又は、Writeメンバ関数群」を利用します。

ストリームの読み出し／書き込みについての詳細は、後の『ファイル／メモリを ストリームに 抽象化するクラス；HStream クラス:: Readメンバ関数群、又は、Write メンバ関数群 』を参照してください。

【参照】

◇　HStream クラス;

ストリームのフラッシュ

〔ストリームのフラッシュ〕

HFile:: FlushStream #include <HFile.h>

void FlushStream();

【説 明】

対象とする「ファイルオブジェクト」の指定された「HFileStream オブジェクト(ファイルストリーム)」が保持する「キャッシュ」をフラッシュします。

本関数は、対象ファイルの「ストリーム」内に残っている書き出しデータを、ディスクに反映するものです。

なお、「ファイルストリーム」へのアクセス（ストリームの読み出し／書き込み等）を実行する場合は、本関数実行後、「HStream クラス:: Readメンバ関数群、又は、Write メンバ関数群」を利用します。

ストリームの読み出し／書き込みについての詳細は、後の『ファイル／メモリをストリームに 抽象化するクラス ; HStream クラス:: Readメンバ関数群、又は、Write メンバ関数群 』を参照してください。

【参 照】

◇ HStream クラス;

3-1-3.ファイル／メモリを ストリームに 抽象化するクラス

「HStream：ファイル／メモリを ストリームに 抽象化するクラス」は、ファイル、動的メモリ（ハンドル、ポインタメモリ）、静的メモリ（スタック内メモリ）等を「ストリーム」に抽象化し、物理的なアクセスを実行するクラスです。
ただし、アプリケーションプログラムがこれらを生成する場合、「HStream クラス」のインスタンス（オブジェクト）を直接生成することはありません。

「HStream オブジェクト（「ストリーム」の基本クラス）」を利用する場合は、「HFileStream（ファイルストリーム）」、「HROMStream（読出だけに利用するメモリのストリーム）」、「HDMemStream（動的メモリのストリーム）」、「HSMemStream（静的メモリのストリーム）」の各クラスのインスタンス（オブジェクト）を生成した後、「HStreamクラス」のメンバ関数を呼び出し、ストリームへのアクセス（読み出し／書き込み等）を実行します。

3-1-3-1. コンストラクタ ／ デストラクタ

〔HStream クラスの生成（コンストラクタ）〕

HStream :: #include <HStream.h>

```
HStream ();
```

〔HStream クラスの削除（デストラクタ）〕

HStream :: #include <HStream.h>

```
HStream ();
```

【説明】

「HStream：ファイル／メモリを ストリームに 抽象化するクラス」のインスタンス（オブジェクト）を生成、或いは、削除します。ただし、アプリケーションプログラムが、本クラスを直接生成することはありません。

「HStream オブジェクト（「ストリーム」の基本クラス）」を生成する場合は、「ファイルストリーム =HFileStreamクラス」、「読出だけに利用するメモリのストリーム =HROMStreamクラス」、「動的メモリのストリーム =HDMemStreamクラス」或いは、「静的メモリのストリーム =HSMemStreamクラス」を利用し、各クラスのインスタンス（オブジェクト）を生成する必要があります。

「HFileStream（ファイルストリーム）」、「HROMStream（読出だけに利用するメモリのストリーム）」、「HDMemStream（動的メモリのストリーム）」、「HSMemStream（静的メモリのストリーム）」の生成については、後の『ストリームの生成／HFileStream、HROMStream、HDMemStream、HSMemStreamクラス』を参照してください。

３-１-３-２．ストリームの生成

アプリケーションプログラムが「HStream：ファイル／メモリを ストリームに抽象化するクラス」を利用する場合は、「ファイルストリーム ＝HFileStreamクラス」、「読出だけに利用するメモリのストリーム ＝HROMStreamクラス」、「動的メモリのストリーム ＝HDMemStreamクラス」或いは、「静的メモリのストリーム ＝HSMemStreamクラス」を使い分け、「HStream オブジェクト（「ストリーム」の基本クラス)」のインスタンス（オブジェクト）を生成する必要があります。

〔HFileStream クラス（ファイルストリーム）〕

HFileStream :: #include <HStream.h>

HFileStream ();

〔HROMStream クラス（読出専用メモリのストリーム）〕

HROMStream :: #include <HStream.h>

HROMStream ();

〔HDMemStream クラス（動的メモリのストリーム）〕

HStream :: #include <HStream.h>

HDMemStream ();

〔HROMStream クラス（静的メモリのストリーム）〕

HSMemStream :: #include <HStream.h>

HSMemStream ();

【説明】

「ファイルストリーム」、「読出だけに利用するメモリのストリーム」、「動的メモリ（ハンドル、ポインタメモリ）のストリーム」或いは、「静的メモリ（スタック内メモリ）のストリーム」の、各「HStream オブジェクト（「ストリーム」の基本クラス)」のインスタンス（オブジェクト）を生成します。

ただし、「HFileStream オブジェクト（ファイルストリーム)」を利用する場合は、「HFile クラス:: OpenReadStream関数 又は、OpenWriteStream関数」を利用し、任意ファイルの「入力ストリーム」或いは、「出力ストリーム」を生成します。
(「HFileStream オブジェクト（ファイルストリーム)」を直接生成することはありません。)

「HFileStream オブジェクト（ファイルストリーム）」の生成については、前の『ファイルを表現するクラス／HFile クラス:: OpenReadStream, OpenWriteStream』を参照してください。

【参照】

◇ HFile クラス;　OpenReadStream／OpenWriteStream ／FlushStream

３-１-３-３．ストリーム有効サイズの取得

対象とする各「ストリーム（HStream オブジェクト）」の有効サイズを取得する場合は、GetSize 関数を利用します。

〔ストリーム有効サイズの取得〕

HStream:: GetSize #include <HStream.h>

```
JLong   GetSize() const;
```

【説明】

対象とする「ストリーム（HStream オブジェクト）」の有効サイズ（KByte単位）を取得します。

なお、ストリーム読み出し／書き込みを実行する場合は、「HStream クラス:: Read メンバ関数群 又は、Write メンバ関数群」を利用します。

ストリーム読み出し／書き込みについての詳細は、後の『ストリームの読み出し／キャンセル； ReadByte, ReadIntger, ReadLong, ReadBytes, ReadLine, ReadString, ReadCString, ReadPString, UnRead』、『ストリームの書き込み； WriteByte, WriteIntger, WriteLong, WriteBytes, WriteLine, WriteString, WriteCString, WritePString, WriteTerminal』を参照してください。

【参照】

◇ HStream クラス;　ReadByte ／ReadIntger／ReadLong
／ReadBytes／ReadLine
／ReadString／ReadCString ／ReadPString／UnRead
／WriteByte ／WriteIntger／WriteLong
／WriteBytes／WriteLine
／WriteString ／WriteCString／WritePString
／WriteTerminal

３-１-３-４．ストリームのアクセス位置の操作

対象とする各「ストリーム（HStream オブジェクト）」の到達状態、及び、現アクセス位置を取得する場合は、以下のメンバ関数を利用します。

到達状態の取得

〔最終位置への到着状態の取得〕

HStream:: IsMore #include <HStream.h>

JBool IsMore() const;

【説明】

対象とする「ストリーム（HStream オブジェクト）」の到着状態を取得します。

本関数を利用することで、対象「ストリーム」が、現在最後の位置に到着しているかどうかを調べることができます。
なお、「HStream クラス:: GetPosition又は、SetPosition関数」を呼び出すことで、「ストリーム」の現在のアクセス位置の取得、設定が可能です。

ストリーム読み出し／書き込みを実行する場合は、「HStream クラス:: Read メンバ関数群 又は、Write メンバ関数群」を利用します。
ストリーム読み出し／書き込みについての詳細は、後の『ストリームの読み出し／キャンセル；ReadByte, ReadIntger, ReadLong, ReadBytes, ReadLine, ReadString, ReadCString, ReadPString, UnRead』、『ストリームの書き込み；WriteByte, WriteIntger, WriteLong, WriteBytes, WriteLine, WriteString, WriteCString, WritePString, WriteTerminal』を参照してください。

【戻り値】

戻り値として、「true」または、「false」が返されます。

◇ 対象「ストリーム」が最後の位置に到着していない場合 ＝ TRUE
◇ 対象「ストリーム」が最後の位置に到着している場合 ＝ FALSE

【参照】

◇ HStream クラス; GetPosition／SetPosition
／ReadByte／ReadIntger／ReadLong／ReadBytes
／ReadLine／ReadString／ReadCString／ReadPString
／WriteByte／WriteIntger／WriteLong／WriteBytes
／WriteLine／WriteString／WriteCString
／WritePString／WriteTerminal

現アクセス位置の取得／設定

〔現アクセス位置の取得〕

HStream:: GetPosition #include <HStream.h>

```
JLong　GetPosition() const;
```

〔現アクセス位置の設定〕

HStream:: SetPosition #include <HStream.h>

```
void　SetPosition(JLong pos);
```

【説明】

対象とする「ストリーム（HStream オブジェクト）」の現在のアクセス位置を、取得、或いは、設定します。

GetPosition 関数は、現在のアクセス位置（KByte単位）を返します。
また、SetPosition 関数は、現在のアクセス位置を、pos で示す位置に設定します。

なお、「HStream クラス:: IsMore 関数」を呼び出すことで、現在「ストリーム」が、既に最後の位置に到着しているかどうかを調べることができます。

ストリーム読み出し／書き込みを実行する場合は、「HStream クラス:: Read メンバ関数群 又は、Write メンバ関数群」を利用します。
ストリーム読み出し／書き込みについての詳細は、後の『ストリームの読み出し／キャンセル； ReadByte, ReadIntger, ReadLong, ReadBytes, ReadLine, ReadString, ReadCString, ReadPString, UnRead』、『ストリームの書き込み； WriteByte, WriteIntger, WriteLong, WriteBytes, WriteLine, WriteString, WriteCString, WritePString, WriteTerminal』を参照してください。

【パラメータ】

JLong pos

pos に、設定するアクセス位置（KByte単位）を指定します。

【参照】

◇　HStream クラス;　IsMore ／ReadByte／ReadIntger／ReadLong／ReadBytes／ReadLine ／ReadString／ReadCString／ReadPString／WriteByte ／WriteIntger／WriteLong／WriteBytes／WriteLine ／WriteString ／WriteCString／WritePString／WriteTerminal

３-１-３-５．ストリームの読み出し／キャンセル

対象とする各「ストリーム（HStream オブジェクト)」を読み出す場合は、以下の「Readメンバ関数群」を利用します。

〔ストリーム読み出し（１バイト分）〕

HStream:: ReadByte #include <HStream.h>

UByte　ReadByte　();

〔ストリーム読み出し（２バイト整数）〕

HStream:: ReadInteger #include <HStream.h>

JWord　ReadInteger ();

〔ストリーム読み出し（４バイト整数）〕

HStream:: ReadLong #include <HStream.h>

JLong　ReadLong ();

【説 明】

対象とする「ストリーム（HStream オブジェクト)」から、１バイト分、２バイト分（２バイト整数）或いは、４バイト分（４バイト整数）のデータを読み出します。

なお、本関数の実行に先立って「HStream クラス:: IsMore 又はGetPosition, SetPosition関数」を呼び出すことで、事前に「ストリーム」の現在の到達状態、及び、アクセス位置の取得、設定が可能です。

ストリームの到達状態、及び、アクセス位置についての詳細は、『ストリームのアクセス位置の操作；HStream クラス:: IsMore、又は、GetPosition, SetPosition 』を参照してください。

【参 照】

◇ HStream クラス;　IsMore ／GetPosition／SetPosition
／ReadBytes／ReadLine
／ReadString／ReadCString ／ReadPString／UnRead
／WriteByte ／WriteIntger／WriteLong
／WriteBytes／WriteLine
／WriteString ／WriteCString／WritePString
／WriteTerminal

〔ストリーム読み出し（任意バイト分）〕

HStream:: ReadBytes #include <HStream.h>

void ReadBytes (void* p, JLong l);

〔ストリーム読み出し（１行分）〕

HStream:: ReadLine #include <HStream.h>

void ReadLine (void* p, JWord& l, JBool lf = false);

【説明】

対象とする「ストリーム（HStream オブジェクト）」から、任意バイト分、或いは、１行分のデータを読み出します。

ReadBytes 関数はlで示す長さ分の「ストリーム」の内容を、ReadLine 関数は１行分（MacintoshではCRまで、DOSではCR/LFまで）の内容を、p で示すメモリに読み出します。

なお、本関数の実行に先立って「HStream クラス:: IsMore 又はGetPosition, SetPosition関数」を呼び出すことで、事前に「ストリーム」の現在の到達状態、及び、アクセス位置の取得、設定が可能です。

ストリームの到達状態、及び、アクセス位置についての詳細は、『ストリームのアクセス位置の操作；HStream クラス:: IsMore、又は、GetPosition, SetPosition 』を参照してください。

【パラメータ】

void* p

p に、読み出すメモリへのポインタを指定します。

JLong l

l に、読み出すデータの長さ（Byte 単位）を指定します。

【参照】

◇ HStream クラス; IsMore ／GetPosition／SetPosition
／ReadByte ／ReadIntger／ReadLong
／ReadString／ReadCString ／ReadPString／UnRead
／WriteByte ／WriteIntger／WriteLong
／WriteBytes／WriteLine
／WriteString ／WriteCString／WritePString

〔ストリーム読み出し（文字列データ）〕

HStream:: ReadString #include <HStream.h>

JWord ReadString (void* p);

〔ストリーム読み出し（文字列データ→C データ型）〕

HStream:: ReadCString #include <HStream.h>

void ReadCString (JByte* cStr);

〔ストリーム読み出し（文字列データ→Pascal データ型）〕

HStream:: ReadPString #include <HStream.h>

void ReadPString (UByte* pStr);

【説明】

対象とする「ストリーム（HStream オブジェクト）」から、文字列のデータを読み出します。

ReadString 関数は、文字列データを、p で示すメモリに読み出します。
ReadCString 関数は、「ストリーム」から文字列データを読み出し、C データ型（最後にヌル値）で、cStr に返します。
ReadPString 関数は、「ストリーム」から文字列データを読み出し、Pascal データ型（先頭に文字列の長さ）で、pStr にで返します。

なお、本関数の実行に先立って「HStream クラス:: IsMore 又はGetPosition, SetPosition関数」を呼び出すことで、事前に「ストリーム」の現在の到達状態、及び、アクセス位置の取得、設定が可能です。

ストリームの到達状態、及び、アクセス位置についての詳細は、『ストリームのアクセス位置の操作；HStream クラス:: IsMore、又は、GetPosition, SetPosition 』を参照してください。

【パラメータ】

void* p

p に、読み出すメモリへのポインタを指定します。

【戻り値】

JByte* cStr

ReadCString 関数で「ストリーム」から読み出された文字列データを、cStr にC データ型（最後にヌル値）で返します。

JByte* cStr

ReadPString 関数で「ストリーム」から読み出された文字列データを、pStr に Pascal データ型（先頭に文字列の長さ）で返します。

【参照】

◇ HStream クラス;　IsMore／GetPosition／SetPosition
／ReadByte／ReadIntger／ReadLong
／ReadBytes／ReadLine／UnRead
／WriteByte／WriteIntger／WriteLong
／WriteBytes／WriteLine
／WriteString／WriteCString／WritePString
／WriteTerminal

〔ストリーム読み出しのキャンセル〕

HStream:: Unread #include <HStream.h>

```
void   Unread (JLong l = 1);
```

【説明】

対象とする「ストリーム（HStream オブジェクト）」の「読み出し」をキャンセルします。

本関数は、現在のアクセス位置を１バイト分戻し、読み出しをキャンセルするものです。

なお、本関数の実行に先立って「HStream クラス:: IsMore 又はGetPosition, SetPosition関数」を呼び出すことで、事前に「ストリーム」の現在の到達状態、及び、アクセス位置の取得、設定が可能です。

ストリームの到達状態、及び、アクセス位置についての詳細は、『ストリームのアクセス位置の操作 ; HStream クラス:: IsMore、又は、GetPosition, SetPosition 』を参照してください。

【参照】

◇ HStream クラス; IsMore ／GetPosition／SetPosition
／ReadByte ／ReadIntger／ReadLong
／ReadBytes／ReadLine
／ReadString／ReadCString ／ReadPString
／WriteByte ／WriteIntger／WriteLong
／WriteBytes／WriteLine
／WriteString ／WriteCString／WritePString
／WriteTerminal

3-1-3-6．ストリームの書き込み

対象とする各「ストリーム（HStream オブジェクト）」に書き込みを実行する場合は、以下の「Write メンバ関数群」を利用します。

〔ストリーム書き込み（1バイト分）〕

HStream:: WriteByte #include <HStream.h>

void WriteByte (UByte n);

〔ストリーム書き込み（2バイト整数）〕

HStream:: WriteInteger #include <HStream.h>

void WriteInteger (JWord n);

〔ストリーム書き込み（4バイト整数）〕

HStream:: WriteLong #include <HStream.h>

void WriteLong (JLong n);

【説明】

対象とする「ストリーム（HStream オブジェクト）」に、1バイト分、2バイト分（2バイト整数）或いは、4バイト分（4バイト整数）のデータを書き込みます。

なお、本関数の実行に先立って「HStream クラス:: IsMore 又はGetPosition, SetPosition関数」を呼び出すことで、事前に「ストリーム」の現在の到達状態、及び、アクセス位置の取得、設定が可能です。

ストリームの到達状態、及び、アクセス位置についての詳細は、『ストリームのアクセス位置の操作；HStream クラス:: IsMore、又は、GetPosition, SetPosition 』を参照してください。

【パラメータ】

UByte n ／ JWord n ／ JLong n

n に、書き込むデータ（順に1バイト、2バイト整数、4バイト整数）を指定します。

〔ストリーム書き込み（任意バイト分）〕

HStream:: WriteBytes #include <HStream.h>

void WriteBytes (const void* p, JLong l);

〔ストリーム書き込み（1行分）〕

HStream:: WriteLine #include <HStream.h>

void WriteLine (void* p, JWord l, JBool lf = false);

【説 明】

対象とする「ストリーム（HStream オブジェクト）」に、任意バイト分、或いは、１行分のデータを書き込みます。

WriteBytes 関数は l で示す長さ分の、WriteLine 関数は１行分（Macintoshでは CRを、DOSではCR/LFを付加）の、p で示すメモリの内容を書き込みます。

なお、本関数の実行に先立って「HStream クラス:: IsMore 又はGetPosition, SetPosition関数」を呼び出すことで、事前に「ストリーム」の現在の到達状態、及び、アクセス位置の取得、設定が可能です。
ストリームの到達状態、及び、アクセス位置についての詳細は、『ストリームのアクセス位置の操作；HStream クラス:: IsMore、又は、GetPosition, SetPosition 』を参照してください。

【パラメータ】

const void* p ／ void* p

p に、書き込む内容を示すメモリへのポインタを指定します。

JLong l

l に、書き込むデータの長さ（Byte 単位）を指定します。

【参 照】

◇ HStream クラス; IsMore ／GetPosition／SetPosition
／ReadByte ／ReadIntger／ReadLong
／ReadBytes／ReadLine
／ReadString／ReadCString ／ReadPString／UnRead
／WriteByte ／WriteIntger／WriteLong
／WriteString ／WriteCString／WritePString
／WriteTerminal

〔ストリーム書き込み（文字列データ）〕

HStream:: WriteString #include <HStream.h>

```
void  WriteString (const void* p, JWord l);
```

〔ストリーム書き込み（文字列データ= C データ型）〕

HStream:: WriteCString #include <HStream.h>

```
void  WriteCString (const JByte* cStr);
```

〔ストリーム書き込み（文字列データ= Pascal データ型）〕

HStream:: WritePString #include <HStream.h>

```
void  WritePString (const UByte* pStr);
```

【説明】

対象とする「ストリーム（HStream オブジェクト）」に、文字列のデータを書き込みます。

WriteString 関数はp で示すメモリの文字列データを、WriteCString 関数はcStr で示すメモリの文字列データ（C データ型）を、WritePString 関数はpStr で示すメモリの文字列データ（Pascal データ型）を書き込みます。

なお、本関数の実行に先立って「HStream クラス:: IsMore 又はGetPosition, SetPosition関数」を呼び出すことで、事前に「ストリーム」の現在の到達状態、及び、アクセス位置の取得、設定が可能です。

ストリームの到達状態、及び、アクセス位置についての詳細は、『ストリームのアクセス位置の操作；HStream クラス:: IsMore、又は、GetPosition, SetPosition 』を参照してください。

【パラメータ】

const void* p ／ void* p

p に、書き込む内容を示すメモリへのポインタを指定します。

JWord l

l に、書き込むデータの長さ（Byte 単位）を指定します。

const JByte* cStr

ReadCString 関数で書き込む文字列データ（C データ型）を指定します。

const JByte* cStr

ReadPString 関数で書き込む文字列データ（Pascal データ型）を指定します。

【参照】

◇ HStream クラス; IsMore／GetPosition／SetPosition
／ReadByte／ReadIntger／ReadLong
／ReadBytes／ReadLine
／ReadString／ReadCString／ReadPString／UnRead
／WriteByte／WriteIntger／WriteLong
／ReadBytes／ReadLine
／WriteTerminal

〔ストリーム書き込み（ターミネータ）〕

HStream:: WriteLineTerminal #include <HStream.h>

void WriteLineTerminal (JBool lf = false);

【説 明】

対象とする「ストリーム（HStream オブジェクト）」の行の終端に、「ターミネータ」を書き込みます。（MacintoshではCRを、DOSではCR/LFを）を書き込む。「読み出し」をキャンセルします。

本関数は、「ターミネータ」として、Macintoshでは「CR」を、DOSでは「CR/LF」を書き込むものです。

なお、本関数の実行に先立って「HStream クラス:: IsMore 又はGetPosition, SetPosition関数」を呼び出すことで、事前に「ストリーム」の現在の到達状態、及び、アクセス位置の取得、設定が可能です。

ストリームの到達状態、及び、アクセス位置についての詳細は、『ストリームのアクセス位置の操作；HStream クラス:: IsMore、又は、GetPosition, SetPosition 』を参照してください。

【参 照】

◇ HStream クラス; IsMore ／GetPosition／SetPosition
／ReadByte ／ReadIntger／ReadLong
／ReadBytes／ReadLine
／ReadString／ReadCString ／ReadPString／UnRead
／WriteByte ／WriteIntger／WriteLong
／WriteBytes／WriteLine
／WriteString ／WriteCString／WritePString

Appendix

附録

App 1. HiBase データベース項目のデータ型

HiBase データベースで利用できるデータ型は以下の通りです。

一般データ型

識別子	定義番号	識別記号	説明
hitDateTime	1	o	日付＆時間
hitDate	2	d	日付
hitTime	3	t	時間
hitFlag	4	f	２値（true， false）
hitChar	5	c	文字
hitString	6	s	文字列
hitNumber	7	n	10 進数字
hitInteger	8	i	バイナリ数字
hitBinary	9	b	不定型
hitPacket	15	p	内部使用

■一般データ型は、256バイト以下の長さを持つ項目につけるデータ型を言います。

データベースレコードに項目値を追加するとき、または、項目値を取りだすとき、スキーマ変換クラス（HRecord）の全ての関数を利用する事ができます。

マルチメディアデータ型

識別子	定義番号	説明
hitFile	0x40	DOS ファイル
hitMacFile	0x4a	Macintosh ファイル
hitJPEG	0x60	絵/JPEG
hitGIF	0x61	絵/JPEG
hitPNG	0x62	絵/JPEG
hitMacPICT	0x6a	絵/JPEG
hitWinBMP	0x6b	絵/JPEG
hitText	0x50	文章/一般
hitHTML	0x51	文章/HTML
hitPDF	0x52	文章/PDF

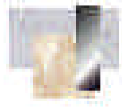
■マルチメディアデータ型は､長さに制約を設けない項目につけるデータ型を言います。

データベースレコードに項目値を追加するとき、または､項目値を取りだすとき、スキーマ変換クラスの、HRecord::FetchItem（項目値を取りだす）、HRecord::AppendItem（項目値を追加する）､HRecord::UpdateItem（項目値を置き換える）､を利用してください。

ユーザ定義型

識別子	定義番号	説明
	１２８～２５４	ユーザ定義型

■HiBaseに、127個までのユーザ定義型（オリジナルなデータ型）を追加することができます



App 2. HiBase API の エラーコード（ECode 型）

\- Error Definitions for HiSystems...

「***HiBase*** =クラスライブラリ」の中で使用されている「エラーコード（ErrorCode 型）」は、以下のとおりです。

プロセスコントロール 関連

```
//      error code  ( 0 )       ----------------------------------------------------

#define  ecNormal          0        //    normal end

//      process control (not error)

#define  ecEndOfFile       1        //    end of file
#define  ecEndOfRecord     2        //    end of record
#define  ecEndOfRepeat     3        //    end of repeat
#define  ecEndOfItem       4        //    end of item
#define  ecEndOfData       5        //    end of data
#define  ecKeyExist        6        //    key already exist
```

ファイル I / O 関連

```
//      file I/O error  ( -10 ～ -99 )

#define  ecFileNotOpen    -10       //    file not open
#define  ecFileDblOpen    -11       //    file already open
#define  ecFileOpened     -12       //    file opened (for create / erase)

#define  ecPathError      -13       //    file path error
#define  ecGetHFSInfo     -14       //    file get info
#define  ecSetHFSInfo     -15       //    file set info

#define  ecDirectory      -16       //    directory error
#define  ecCreateDir      -17       //    create directory error
#define  ecEraseDir       -18       //    erase  directory error
#define  ecSpec2Path      -19       //    mother directory error
```

```
#define  ecCreateFile    -20    //  create file error
#define  ecEraseFile     -21    //  erase  file error
#define  ecOpenFile      -22    //  open   file error
#define  ecCloseFile     -23    //  close  file error
#define  ecFlushFile     -24    //  flush  file error
#define  ecRenameFile    -25    //  rename file error

#define  ecSetPosition   -26    //  set position error
#define  ecGetPosition   -27    //  get position error
#define  ecReadFile      -28    //  read   file error
#define  ecWriteFile     -29    //  write  file error
#define  ecGetFileSize   -30    //  get file size error
#define  ecSetFileSize   -31    //  set file size error
```

キー、レコード、スキーマＩ/Ｏ ハンドリングエラー 関連

【キーハンドリングエラー】

```
//    key handling error （-100 ～ -199）

#define  ecKeyDuplicate   -101   //  key already exist thought unique key
#define  ecKeyPathError   -102   //  key path error
#define  ecKeyValueError  -103   //  key value error
#define  ecKeyNoRecord    -104   //  key no record
#define  ecKeyPath        -105   //  key path error
```

【レコードハンドリングエラー】

```
//    record handling error （-200 ～ -299）

#define  ecGetData     -201   //  get    data from HCioMap
#define  ecInsData     -202   //  insert data from HCioMap
#define  ecDelData     -203   //  delete data from HCioMap
#define  ecUpdData     -204   //  update data from HCioMap
#define  ecNullPacket  -205   //  null packet error
```

【スキーマＩ/Ｏハンドリングエラー】

```
//    schema I/O handling error （-300 ～ -399）

#define  ecFormat      -301   //  format error
#define  ecDataFormat  -302   //  data format error
#define  ecDataLength  -303   //  length error
```

データベース、データベースファイル 関連

【データベースエラー】

```
//      HDBM （-400 ～ -499）

#define   ecOpened        -401    //   opened
#define   ecDDAddFile     -402    //   DDAddFile  error
#define   ecDDAddItem     -403    //   DDAddItem  error
#define   ecDDAddKey      -404    //   DDAddKey   error
#define   ecDDGetKey      -405    //   DDGetKey   error
#define   ecKeyCheck      -406    //   DDKeyCheck error

#define   ecNoDDFile      -407    //   no DDFile
#define   ecNoDDItem      -408    //   no DDItem
#define   ecNoDDKey       -409    //   no DDKey
```

【データベースファイルエラー】

```
//      HDBFile （-500 ～ -599）

#define   ecNotOpen       -501    //   not open
#define   ecNoKey         -502    //   no key
#define   ecWrongKey      -503    //   wrong key
#define   ecHoldRecord    -504    //   record hold
#define   ecWrongOpr      -505    //   wrong operator
#define   ecNoSet         -506    //   set not found
```

【その他】

```
//      HBase （-600 ～ -699）

#define   ecDDGet         -601    //   error from DDGet
#define   ecDDAdd         -602    //   error from DDAdd
#define   ecDDUpd         -603    //   error from DDUpd
#define   ecDBCreate      -604    //   error from DBCreate
#define   ecDBDelete      -605    //   error from DBDelete

//      HDAPI/HCAPI （ -1000 ～ - 1099）

#define   ecRootPath      -1001   //   no root path
#define   ecConnection    -1002   //   connection error
```

ロード、アンロード 関連

```
//      HLdUld （-1100 ～ -1199）

#define ecWrongUnload       -1101   //  unload error
#define ecWrongLoad         -1102   //  load error
#define ecWrongPath         -1103   //  wrong path
#define ecWrongTag          -1104   //  wrong tag
#define ecWrongDBDefine     -1105   //  wrong db define
#define ecWrongFileDefine   -1106   //  wrong file define
#define ecWrongItemDefine   -1107   //  wrong item define
#define ecWrongKeyDefine    -1108   //  wrong key  define
#define ecWrongKWD          -1109   //  wrong keyword
#define ecUnloadFile        -1110   //  unload file error
```

仮想テーブル 関連

```
//      HVTable （-1200 ～ -1299）

#define ecNoRNbr            -1201   //  no rNbr
#define ecNoList            -1202   //  no list
#define ecPwr_notActive     -1203   //  no active
#define ecPwr_badID         -1204   //  bad id
#define ecPwr_noDesc        -1205   //  no desc
#define ecPwr_noKeyName     -1206   //  no key value
#define ecPwr_noItemName    -1207   //  no item name
#define ecPwr_noSetName     -1208   //  no set name
#define ecPwr_Syntax        -1209   //  syntax error
#define ecPwr_noEMark       -1210   //  no E mark
#define ecPwr_noFOper       -1211   //  no opr
```

ユーティリティー 関連

```
//      Utilities （-1300 ～ -1399）

#define ecFileNotFound  -1301   //  file not found
#define ecLaunchApp     -1302   //  fail to launch application
```

コントロールハンドリングエラー 関連

```
//      Control Handling error （-2100～ -2199）

#define ecDMOpen         -2101   //  file open
#define ecDMClose        -2102   //  file close
#define ecDMFlush        -2103   //  file flush
#define ecDMKeyCreate    -2104   //  key create
#define ecDMKeyDelete    -2105   //  key delete
#define ecDMMaxRNbr      -2106   //  max rNbr
#define ecDMFileSize     -2107   //  file size
#define ecDMNewSetID     -2108   //  get new set-id
#define ecDMGetRecord    -2109   //  get record
#define ecDMInsRecord    -2110   //  insert record
#define ecDMDelRecord    -2111   //  delete record
#define ecDMUpdRecord    -2112   //  update record
#define ecDMSetMakeAll   -2113   //  set make all
#define ecDMSetMakeKey   -2114   //  set make by key
#define ecDMSetMakeItm   -2115   //  set make by item
#define ecDMSetScanItm   -2116   //  set scan by item
#define ecDMSetCalc      -2117   //  set calc
#define ecDMSetSort      -2118   //  set sort
#define ecDMSetSize      -2119   //  set size
#define ecDMSetRGet      -2120   //  set record to be get
#define ecDMSetRAdd      -2121   //  set record to be add
#define ecDMSetRDel      -2122   //  set record to be delete
#define ecDMSetCancel    -2123   //  set cancel
#define ecDMKSLocate     -2124   //  key sequence locate
#define ecDMKSRead       -2125   //  key sequence read
#define ecDMKSCancel     -2126   //  key sequence cancel
#define ecDMPSLocate     -2127   //  phisical sequence locate
#define ecDMPSRead       -2128   //  phisical sequence read
#define ecDMPSCancel     -2129   //  phisical sequence cancel

#define ecDMGetValue     -2131   //  get value
#define ecDMInsValue     -2132   //  insert value
#define ecDMDelValue     -2133   //  delete value
#define ecDMUpdValue     -2134   //  update value
#define ecDMMaxValueSize -2135   //  max value size

#define ecCmdCode        -2140   //  command error
#define ecFileBusy       -2141   //  file busy
#define ecWrongRequest   -2142   //  wrong request
#define ecDriver         -2143   //  river error
```

その他

【ネットワークエラー】

```
//      NetWork （-2200 〜 -2299）

#define   ecTimeOut          -2201      //   time out error
#define   ecRProtocol        -2211      //   request protocol error
#define   ecThreadDie        -2212      //   request protocol error
#define   ecLaunchCGI        -2213      //   error for launch application
#define   ecCGIProcPSN       -2214      //   CGI error (PSN)
#define   ecCGIProcCAE       -2215   //   CGI error (create AppleEvent)
#define   ecCGIProcSAE       -2216   //   CGI error (send AppleEvent)
#define   ecCGIProcAESize    -2217   //   CGI error (size of param)
#define   ecCGIProcAEGet     -2218   //   CGI error (get param)
```

【CGI、他】

```
//      CGI Toolkit （-3000 〜 -3499）

#define   ecPutResult    -3001     //   put param (keyDirectObject) error
#define   ecAESuspend    -3002     //   AppleEvent suspend error

//      throw code   ------------------------------------------------

#define   tcWrongWrite   -10       //      wrong write
```

HiBase 関連ドキュメント & サンプルプログラム一覧

HiBase のアプリケーションプログラム開発 に関連するドキュメント、及び、サンプルプログラムを参照する場合は、以下を参考にしてください。

◇ ***HiBase*** **アプリケーションプログラミングの基礎知識**

プログラミング・ガイド　（PrGuide.pdf）

Documentsフォルダ内

◇ ***HiBase*** **API の詳細（設計とその使用法）**

プログラム・リファレンス　（PrRefs.pdf）

Documentsフォルダ内

◇ ***HiBase*** **API を利用したサンプルプログラム**

PDBDefine.Prj, DBLoad.Prj, DBSearch.Prj, DBDelete.Prj, DBUnload.Prj, DBTest.Prj　Develop:: Samplesフォルダ内

◇ ***HiBase*** **API for Java （設計とその使用法）**

プログラム・リファレンス for Java）））（PrRefs(Java).pdf）

Documentsフォルダ内

◇ ***HiBase*** **基本アプリケーション&インタフェースプログラムインストール**

スタートアップ・マニュアル（StartUp.pdf）

Documentsフォルダ内

◇ ***HiBase*** **の運用（基本アプリケーション操作）**

オペレーション・マニュアル（Operation.pdf）

Documentsフォルダ内

◇ **CGI 開発の支援**

CGI 開発支援解説 for HCGI & HACGI））））（HCGI&HACGI.pdf）

Documentsフォルダ内

文書履歴

作成	１９９８年１２月	伊藤由美子
第一改定	１９９９年9月	西林瑞夫

HiBase プログラム・リファレンス

発行日：　１９９8年12月

改定：　１９９９年9月

発　行：　ホロン株式会社