－ －

2014年1月20日 初版

板垣 衛

改訂履歴

|  |  |  |  |
| --- | --- | --- | --- |
| 版 | リリース | 担当 | 改訂内容 |
| 初版 | 2014年1月20日 | 板垣 衛 | （初版） |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目次

[ 概略 1](#_Toc377261752)

[ 目的 1](#_Toc377261753)

[ 基本用語 1](#_Toc377261754)

[▼ 「ゲームデータ」 1](#_Toc377261755)

[ 要件定義 2](#_Toc377261756)

[▼ 基本要件 2](#_Toc377261757)

[▼ 要求仕様／要件定義 2](#_Toc377261758)

[ 仕様概要 3](#_Toc377261759)

[▼ 環境 3](#_Toc377261760)

[▼ ワークフロー 4](#_Toc377261761)

[ データ仕様 4](#_Toc377261762)

[▼ DB/Excel 5](#_Toc377261763)

[▼ 拡張JSON 5](#_Toc377261764)

[ JSON形式の採用について 5](#_Toc377261765)

[ 拡張JSON仕様 6](#_Toc377261766)

[▼ データ定義JSON 8](#_Toc377261767)

[▼ 中間JSON① 8](#_Toc377261768)

[▼ 中間JSON② 9](#_Toc377261769)

[▼ フォーマット定義JSON 9](#_Toc377261770)

[▼ データ型定義リスト 14](#_Toc377261771)

[▼ 計算式用拡張関数定義リスト 16](#_Toc377261772)

[▼ チェック用JSON 17](#_Toc377261773)

[▼ C言語ソース 19](#_Toc377261774)

[▼ バイナリデータ 22](#_Toc377261775)

[ 値の演算と組み込み関数 24](#_Toc377261776)

[▼ 計算式解析関数：expr() 25](#_Toc377261777)

[ 計算式 25](#_Toc377261778)

[▼ 計算式の構文 26](#_Toc377261779)

[▼ 応演算子 27](#_Toc377261780)

[▼ 組み込み関数 28](#_Toc377261781)

[▼ 拡張関数 28](#_Toc377261782)

[ 処理仕様 29](#_Toc377261783)

[▼ プリプロセッサ 29](#_Toc377261784)

[▼ データ変換ツール 30](#_Toc377261785)

[▼ データ変換実行バッチ 33](#_Toc377261786)

[▼ データ変換処理（C# Tips） 35](#_Toc377261787)

[ JSON.Netの使用 35](#_Toc377261788)

[ JSON解析済みオブジェクトをC#スクリプト内で使用 36](#_Toc377261789)

[▼ データ取り込み処理（ランタイム） 38](#_Toc377261790)

[▼ データアクセス処理（ランタイム） 38](#_Toc377261791)

[ 計算式処理仕様 38](#_Toc377261792)

[▼ 計算式解析処理 38](#_Toc377261793)

[▼ 計算式データの構造 42](#_Toc377261794)

[▼ 計算式実行処理（ランタイム） 43](#_Toc377261795)

[ データ変換作業環境の改善 43](#_Toc377261796)

[▼ バッチファイル の利用 43](#_Toc377261797)

[▼ SCons の利用 43](#_Toc377261798)

# 概略

本書における「ゲームデータ」とは、大まかには、グラフィック関係データとサウンド関係データ以外のデータ全般を指す。

その多くはプランナーが扱うデータで、ゲームを制御するための設定やパラメータなどのことである。多彩なデータを扱い、ゲーム固有のデータ構造となるものが多い。

本書は、「ゲームデータ」の入力フォーマットと実機上のデータフォーマット、および、その変換・取り込み処理に関する基本仕様を規定する。

# 目的

汎用化したゲームデータ処理により、バージョン整合処理や数式解析処理などの高度な処理も標準化し、作業の効率化と安全性の向上を目的とする。

# 基本用語

## 「ゲームデータ」

本書においては、「ゲームデータ」という用語を下記の意味で扱う。

グラフィック関係データとサウンド関係データを除くデータ全般。一般的には、これらのデータを含むリソース全般をまとめて「ゲームデータ」と呼ぶが、本書では区別して扱う。

グラフィック関係データとサウンド関係データであっても、それを制御するためのゲーム固有のデータは「ゲームデータ」に類する。

何らかの処理設定やデバッグ用データなど、プログラマーが扱うデータもまた「ゲームデータ」である。

* 基本的には、予め定義された静的なデータを指し、ファイル（リソース）として扱われる。
* メモリ上で内容が変動する動的なデータやセーブデータなどは、「ゲームデータ」の範疇に含まない。しかし、例えば「ゲームデータを読み込んでセーブデータを復元する」といった、動的なデータを再現するためのゲームデータの活用はありえる。
* PlayStation系では「インストールデータ」を指して「ゲームデータ」と呼ぶが、本書における「ゲームデータ」はそれとは別物である。

# 要件定義

## 基本要件

本書が扱うシステムの基本要件は下記の通り。

ゲームデータの入力データをテキストファイル形式で扱う。

ExcelやDBで管理するデータの場合、直接実機向けのデータに変換せず、いったんテキストファイルに変換して扱う。これにより、全てのゲームデータに対して、一貫したデータ変換とフォーマットを使用するものとする。なお、Excel/DBに関する仕様は本書の範疇外とし、別途仕様を策定する。

テキストファイルをバイナリデータに変換する汎用ツールを作成する。

バイナリデータを実機（ゲーム）に取り込むための汎用処理を作成する。

テキストファイルの取り込みは、実機上では行わないものとする。（パーサーを実装しない。）

## 要求仕様／要件定義

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

テキストファイルは、JSONを基本フォーマットとして統一する。

テキストファイルの文字コードはUTF-8とし、日本語や欧州文字に対応する。

テキストファイルは、下記の拡張仕様（JSONが非対応の仕様）に対応する。

* JavaScript形式のコメント文を使用できる。（例：// comment、/\* comment \*/）
* C言語形式の#include文と#define文を使用できる。
* データ部に四則演算を用いることができる。（例：{ ”age”: 30 + 3, … } ）
* データ部にCRC変換やゲーム用計算式変換などの特殊な関数を使用できる。（例：{ ”id”: CRC(“c0010”), … }, { “condition”: Expr(“IsFlag(\“AlreadyMetOldMan\”) == True && GetChapter() >= 2”), … } ）

注： JSONデータをMongoDBなどのドキュメント指向データベース（BSON形式で保存される）で扱う場合、これらの拡張仕様が使えないので注意。同様の情報を扱うための別の仕様も合わせて策定する。

テキストファイルからバイナリデータに変換するための変換設定もまたJSON形式のテキストデータとして定義する。期待される値の範囲など、エラー判定用の設定も可能。

不定長の配列をメンバーに持つ構造体にも対応。

専用のデータ変換ツールを通して、バイナリデータを出力する。

データ変換ツールは、下記の仕様に対応する。

* CUIツールとして構成し、単純に一つのテキストファイルを一つのバイナリファイルに変換する。これは、任意のバッチ処理や他のツールからの呼び出しなどに対応しやすい形式である。
* エンディアンの指定、ポインターのビット数（32 or 64）指定に対応。
* 【できれば】シフトJISなどのエンコーディングの指定に対応。
* データ構造／内容のエラーを検出した場合、出力ファイルは作成されず、エラーが通知される。

バイナリデータは、メモリ上のイメージとしてほぼそのまま取り込める。

* 文字列のポインター変換などの処理も取り込み時に同時に行われる。

バイナリデータの構造が変更され、実機上の構造とずれた場合、取り込み処理は自動的にその事を検出し、項目毎のデータ取り込み処理を行う。要は、データ構造が変わっても、プログラムが極力正常に動作するようにする。

# 仕様概要

## 環境

OS：Windows系PC（XP以上）32bit/64bit

必須ツール①：テキストエディタ　※なんでもよい

必須ツール②：MinGW(GCC)　※プリプロセッサ

必須ツール③：変換ツール　※独自開発

使用ツール（オプション）：Python + SCons

## ワークフロー



DB／Excel 元データ　※この仕様書では扱わない。

データ定義JSON テキストで記述されたデータ

中間JSON プリプロセッサを通した後の「データ定義JSON」

フォーマット定義JSON データ変換のためのフォーマットとルールを定義

データ型定義リスト 「フォーマット定義JSON」に付随するリスト　※「データ型」の仕様を定義

計算式用拡張関数定義リスト

「フォーマット定義JSON」に付随するリスト　※計算式で使用する「拡張関数」を定義

変換ツール 「データ定義JSON」を「バイナリデータ」や「C言語ソース」に変換するツール

バイナリデータ バイナリデータ（C言語構造体と一致）

C言語ソース 構造体とバージョン整合用の構造定義情報

チェック用JSON バイナリデータと同じ内容を表すJSON　※内容確認／二次利用向け

実機 ゲームプログラム　※バイナリデータのみを扱える

# データ仕様

以下、前述の「ワークフロー」に基づいて、各データの内容と構造について説明。

## DB/Excel

DB／Excelに関する仕様は本書では扱わない。別途仕様を策定。

なお、想定としては、ドキュメント指向DB（MongoDBなど）＋RDB（PostgreSQLなど）で管理し、Excelへのエクスポート／インポートでデータ編集する。DBには独自のバージョン管理とロック機構を備え、チーム開発を支援する。

DBに記録されるデータもJSON形式とするが、Excel化を考慮したデータ構造とするため、本書で示すデータ構造とは必ずしも一致しない。DB上のデータを変換してデータ定義JSONを出力する。

DBに記録されるJSONは、一切の拡張仕様が使用できない。しかし、この場合、直接テキストを編集することはないので、ほぼ問題はない。

## 拡張JSON

「データ定義JSON」に適用する記述仕様。

JSONの基本仕様に独自の拡張仕様を加えたもの。

「フォーマット定義JSON」と、それに付随する「データ型定義リスト」「計算式用拡張関数定義リスト」にも、この「拡張JSON」の記述仕様を適用する。

### JSON形式の採用について

データフォーマットにはJSON形式を採用する。採用理由は下記の通り。

* テキストファイルによる柔軟で素早いデータ策定が見込める。
* 「Excel⇒バイナリデータ出力」よりも、「Excel⇒テキスト⇒バイナリデータ」とすることで、ツール（Excel）の用意に先行してデータ策定と処理を作成できる。
* 「テキスト⇒バイナリデータ」の変換ツールを汎用化できる。
* XMLのように、メンバー名を持った柔軟な構造体（オブジェクト）を扱える。
* 配列が扱える。
* XMLよりも軽量で視覚的にも分かり易い。
* XMLの「<タグ>～</タグ>」のような記述の冗長性がなく、開始タグ（「タグ：」）のみを記述する。
* 反面、XMLの「属性」のような複雑なデータ記述はできない。その方がシンプルで良い。
* XMLのような一般的なデータ記述仕様であり、多くのライブラリ（Javaや.Net）やデータベース（MongoDBやCouchDB）などが対応している。
* Ajaxで利用が広がり注目を集めた。テキスト形式かつ軽量な点はhttp通信にも向いている。XMLの代替として注目。
* JavaScript準拠のフォーマットであり、JaveScriptやそれをサポートする多数の言語（ライブラリ）でそのままデータを取り込めるため、データの二次利用がし易い。

### 拡張JSON仕様

* 基本仕様：
* JSON仕様： <http://ja.wikipedia.org/wiki/JavaScript_Object_Notation>
* ファイルの拡張子は .json。
* エンコーディングはUnicode。基本的にUTF-8。
* キーと値のペアでデータを記述するのが基本。
* 値は、整数、小数、真偽値（true / false）、文字列、null を記述可能。文字列はダブルクォーテーションで囲む。
* 整数の記法は10進法に限られる。「0127」のような8進記法、「0x12af」のような16進記法には対応しない。
* 小数の記法には「1.0e-10」のような指数表記が可能。
* 文字列には「\n」「\r」「\\」などのエスケープシーケンスが記述可能。

【注意】 エスケープ文字を含む一部のシフトJISコードの文字（「ソ」「能」「表」など）は正常に扱えない事に注意。エンコーディングはあくまでもUnicode。

* 複数のデータを並べる場合は、「,」（コンマ）で区切って列挙。

【注意】 MicorosfoftのJscriptなど、最後のデータには「,」が付いているとエラーになるパーサーもあるので注意。

* 配列を記述可能。
* オブジェクト（構造体）を記述可能。
* キーと値の記述：
* キーと値のペアは、「”キー”＋：＋値」の書式で記述。
* キーには文字列のみが指定可能で、必ずダブルクォーテーションで囲む。

例：

|  |
| --- |
| “id”: 123,  “name”: “名前”,  “value”: 1.23 |

* 配列の記述：
* 「［ 」～「 ］」（角括弧）で囲む事で表現する。

例：

|  |
| --- |
| “array”: [ 1, 2, 3 ]  “data”: [ “山田”, “田中”, “佐藤” ] |

多重配列も可能：

|  |
| --- |
| “array”:  [  [1, 2, 3],  [4, 5, 6]  ] |

* オブジェクトの記述：
* 「｛ 」～「 ｝」（波括弧）で囲む事で表現する。
* 一つのデータ項目は、「“キー”」+「：」+「値」のペアを記述。「キー」は構造体のメンバー名と同じ意味。重複禁止。ダブルクォーテーションで囲んで記述。

例：

|  |
| --- |
| {  “id” : 123 ,  “name” : “山田”,  “param” :  {  “str” : 10,  “vit” : 20  }  } |

* データテーブルの記述：
* オブジェクトを配列で列挙する。

例：

|  |
| --- |
| “table”:  [  {"id":1, "name":"山田", "param": {"str": 10, "vit": 2.0, "ext": [1, 2, 3], "is\_cleared": true } },  {"id":2, "name":"田中", "param": {"str": 11, "vit": 2.1, "ext": [4, 5, 6], "is\_cleared": false } },  {"id":3, "name":"佐藤", "param": {"str": 12, "vit": 2.2, "ext": [7, 8, 9], "is\_cleared": true } }  ] |

* 独自拡張仕様：
* C言語のコンパイラに依存し、多数のプリプロセッサ用の記述をサポート。
* JavaScript形式（C++形式）のコメント（「//」もしくは「/\*」～「\*/」）を使用可能。
* #include 文を使用可能。
* #define マクロ、#undef を使用可能。
* #pragma once を使用可能。
* #if, #ifdef, #ifndef ~ #elif, #else ~ #endif を使用可能。
* 文字列化演算子「#」、トークン連結演算子「##」を使用可能。
* 値に8進表記（0 + 0～7の数字）、16進表記（0x + 0～f の数字）、2進表記（0b + 0～1の数字）を指定可能。書式はC言語準拠。なお、2進表記の書式は C++14 仕様準拠。
* 値に四則演算などの演算子を記述可能。
* 組み込み関数（pow(), sqrt(), crc(), expr() など）を使用可能。
* キーのダブルクォーテーション表記は省略可能。

例：

|  |
| --- |
| //データ定義JSONサンプル  //共通定義のインクルード  #include “common.jsonh”  //BASE\_\*\*マクロが定義されている  //キー定義  #define KEY\_ID “id”  #define KEY\_NAME “name”  //パラメータ計算マクロ  #define STR(x) BASE\_STR + x  #define VIT(x) BASR\_VIT + x  //名前用文字列作成マクロ  #define TO\_NAME(s) #s  /\*\*  \* キャラデータリスト  \*/  “table”:  [  { KEY\_ID: CRC(“c0010”), KEY\_NAME: TO\_NAME(山田), param: {"str": STR(1), "vit": VIT(1) },  “condition”: expr(“getChapter() >= 20”) },  { KEY\_ID: CRC(“c0020”), KEY\_NAME: TO\_NAME(田中), param: {"str": STR(2), "vit": VIT(2) } },  { KEY\_ID: CRC(“c0030”), KEY\_NAME: TO\_NAME(佐藤), param: {"str": STR(3), "vit": VIT(3) } }  ] |

## データ定義JSON

データを定義するためのJSON。文章中では、短縮して「JSONデータ」と表記する場合もあるが、「データ定義JSON」と同義。

記述仕様は「拡張JSON」形式。

オブジェクトの配列（いわゆる「テーブル」）で定義するのが基本形。「［ 」で始まり「 ］」で終わる。その内部には、「｛ 」~「 ｝」で定義されたオブジェクト（構造体）を列挙する。

一つのファイルでは一つのデータ（テーブル）だけを扱う。ネストしたデータ構造も使用可能。

サンプルは、前述の「拡張JSON」の仕様を参照。

## 中間JSON①

データ定義JSONにプリプロセッサを通した状態の中間データ。プリプロセッサには、フリーのC言語コンパイラ MinGW(GCC) を使用。

プリプロセッサの機能に頼り、#line文が埋め込まれた状態で出力する。これにより、変換ツールがエラーを検出した際に、プリプロセス前の行位置や、インクルードファイル内のエラーであることを表示できる。

この状態ではJSONパーサーに通せないので注意。

例：前述の「拡張JSON」からの変換結果

|  |
| --- |
| ※空行が多いのは、コメントや#define が除去された結果。  エラーメッセージ出力時に行番号がずれないように、残っている行の位置は変化しないため、空行が多くなる。  # 1 “common.jsonh”1  （インクルードファイルが展開される。）  （インクルードファイルから元のファイルに戻る。最初の # 5 は、この位置から元のファイルの 5 行目が再開することを示す。）  # 5 "data.json" 2  （コンパイラによっては、「#line」という書式のものもある。この出書式は GCC 準拠。）  “table”:  [  { “id”: CRC(“c0010”), “name”: “山田”, param: {"str": 10 + 1, "vit": 20 + 1 },  “condition”: expr(“getChapter() >= 20”) },  { “id”: CRC(“c0020”), “name”: “田中”, param: {"str": 10 + 2, "vit": 20 + 2 } },  { “id”: CRC(“c0030”), “name”: “佐藤”, param: {"str": 10 + 3, "vit": 20 + 3 } }  ] |

## 中間JSON②

中間JSON①に対して、値の演算と組み込み関数を処理した結果のJSON。#line文も除去する。なお、expr() 関数は処理しない。

中間JSON①への変換＋中間JSON②への変換により、JSONパーサーに通せる状態になる。

データ変換ツールの実行中にメモリ内で処理されるが、確認用にファイル出力も可能。

例：前述の「中間JSON①」からの変換結果

|  |
| --- |
| [  { “id”: 745853103, “name”: “山田”, param: {"str": 11, "vit": 21 },  “condition”: expr(“getChapter() >= 20”) },  { “id”: 123306860, “name”: “田中”, param: {"str": 12, "vit": 22 } },  { “id”: 507687469, “name”: “佐藤”, param: {"str": 13, "vit": 23 } }  ] |

## フォーマット定義JSON

データフォーマットを定義するためのJSON。データ変換ツールに変換方法を指定するために用いる。データ定義JSONファイルを入力して、バイナリデータファイルとC言語ソースファイルを出力するためのフォーマットとルールが定義される。

基本的な記述仕様は「拡張JSON」形式に従う。記述内容は、フォーマット定義のための設定項目が定められている。下記のサンプルでその仕様を示す。赤い字で書かれた項目が、サンプル中で初出の定義項目であり、詳しい説明を併記する。

例：

|  |
| --- |
| //データフォーマット定義JSONサンプル  {  “name”: “CharaData”, //データフォーマット名　※構造一致照合用。  “majorVer”: 1, //データフォーマットメジャーバージョン　※構造一致照合用。  “minorVer”: 0, //データフォーマットマイナーバージョン　※構造一致照合用。  “comment”: “キャラパラメータ構造定義”, //コメント　※C言語ソース用。    “headerFileName”: “charaData.h”, //ソースファイル名　※C言語ヘッダーファイル出力用。  “declFileName”: “charaDataDecle.cpp”, //ソースファイル名　※C言語ソースファイル（バージョン整合用  //　　　　　　　　　　構造定義）出力用。  “isUsePragmaOnce”: true, //#pramgma once 使用指定　※C言語ヘッダーファイル出力用。  “headerIncludeFiles”: [“types.h”], //インクルードファイル　※C言語ヘッダーファイルに適用。  //　　　　　　　　　　　　複数指定可。  “declIncludeFiles”: [“gameDataDecl.h”], //インクルードファイル　※C言語ソースファイル（バージョン整合用  //　　　　　　　　　　　　構造定義）に適用。  //　　　　　　　　　　　　複数指定可。  “headerNamespace”: [“charaDataDef”], //ネームスペース　※C言語ヘッダーファイルに適用。  //　　　　　　　　　ネストする場合は配列で複数指定。  “declNamespace”: [“charaDataDecl”], //ネームスペース　※C言語ソースファイル（バージョン整合用  //　　　　　　　　　構造定義）に適用。  //　　　　　　　　　ネストする場合は配列で複数指定。  “funcNameForGetDecl”: “getGameDataDecl”, //バージョン整合用構造定義情報の取得関数名  //　　　　　　　　※C言語ソースファイル（バージョン整合用  //　　　　　　　　　構造定義）に適用。    “struct”: //構造体  {  “name”: “T\_CHARA”, //構造体名  “comment”: “キャラ構造体”, //コメント　※C言語ソース用。    “isMakeSource”: true, //構造体の定義をC言語ヘッダーに出力するか？  //　※規定値は true。汎用構造体などの定義済みの構造体を用いる場合は  //　　 false を指定する。    “primaryKey”: “id”, //主キー項目　※メンバー名で指定。主キーの昇順にデータが並べ替えされる。  //　　　　　　　重複検出でエラー。ネストした構造体のメンバーは指定不可。  “secondaryKey”: “name”, //副キー項目　※メンバー名で指定。検索用のインデックステーブルが作成される。  //　　　　　　　重複検出でエラー。ネストした構造体のメンバーは指定不可。    “indexes”: [“kana”], //インデックス項目　※キーで指定。並べ替え用のインデックステーブル作成用。  //　　　　　　　　　　複数キー指定可。複合キー指定不可。重複検出なし。  //　　　　　　　　　　ネストした構造体のメンバーは指定不可。    “members”: //メンバー　※デフォルトでは、定義順がデータの並び順になるので注意。  [  //メンバー定義：基本形  {  “name”: “id”, //メンバー名  “comment”: “識別ID”, //コメント　※C言語ソース用。  “key”: “id”, //対象キー　※JSONデータ上のキー。  //　　　　　　省略時は “name” と同じとみなす。  “type”: “crc”, //データ型　※s8,u8,s16,u16,s32,u32,s64,u64,f31,f64,ptr,struct,  //　　　　　　str,expr,crc,crcs を指定可。  //　　　　　　crc,crcs は文字列が32bit整数に変換される。  //　　　　　　expr は文字列がT\_EXPR 型の計算式データに変換  //　　　　　　される。  //　　　　　　str と expr はポインターに変換され、バイナリ  //　　　　　　データにはオフセット値として記録される。  “typeName”: “CRC32” //データ型名　※C言語ソース用  //　　　　　　　特別に指定したい場合だけ指定。  //　　　　　　　通常は “type” に対応した型から自動判定される。  },  //メンバー定義：文字列型の場合  {  “name”: “name”, //メンバー名  “comment”: “名前”, //コメント  “type”: “str” //データ型　※文字列のデータ型は、C言語ソース上では  //　　　　　　 const char\* などのポインター型に置き換わる。  //　　　　　　文字列データはバイナリデータの後部にまとめられ、  //　　　　　　ポインターはその位置を指す。  //　　　　　　バイナリデータ上では、文字列データのオフセット値が  //　　　　　　記録される。  },  //メンバー定義：計算式型の場合  {  “name”: “condition”, //メンバー名  “comment”: “有効化条件式”,//コメント  “type”: “expr” //データ型　※計算式のデータ型は、C言語ソース上では  //　　　　　　 const T\_EXPR\* 型に置き換わる。  //　　　　　　実際のデータはバイナリデータの後部にまとめられ、  //　　　　　　ポインターはその位置を指す。  //　　　　　　バイナリデータ上では、計算式データのオフセット値が  //　　　　　　記録される。  //　　　　　※JSONデータに記述された計算式（文字列）を解析して、  //　　　　　　計算式データ（バイナリ）に変換して記録する。  //　　　　　　この解析の際にエラー判定も行う。  //　　　　　※計算式内で使用される関数は、予め用意された組み込み  //　　　　　　関数（crcやpowなど）のほか、実機側の処理で  //　　　　　　計算式用に用意された拡張関数（getChapterなどの  //　　　　　　ゲーム依存の関数）を指定できる。  //　　　　　　正しい名前とパラメータで拡張関数を使用しているか  //　　　　　　どうかは、変換ツール実行時に渡される  //　　　　　　「拡張関数定義リスト」に基づいて判定する。  },  //メンバー定義：値の場合　※エラー判定のサンプル  {  “name”: “power”, //メンバー名  “comment”: “力”, //コメント  “key”: “param.power”, //対象キー　※JSONデータ上のネストしたデータは  //　　　　　　「.」で区切って指定。  “type”: “i8”, //データ型  “default”: 1, //省略時の規定値　※JSONデータ上で記述されなかった場合の規定値。  “isRequired”: false, //入力必須項目？　※JSONデータ上で記述が必須か？  //　　　　　　　　※エラー判定用、省略時はエラー判定なし  // （false指定と同じ）。ただし、  //　　　　　　　　　primayKey, secondaryKey, indexes に  //　　　　　　　　　指定された項目は必然的に入力必須となる。  “min”: 0, //最小値　※エラー判定用、省略時はエラー判定なし。  “max”: 100 //最大値　※エラー判定用、省略時はエラー判定なし。  //※データ型に応じた最小値～最大値の範囲チェックは  //　デフォルトで行われる。例えば、i8 なら -128～127の  //　範囲外の値が指定されたらエラー。  },  //メンバー定義：固定長配列の場合  {  “name”: “tol”, //メンバー名  “comment”: “耐性”, //コメント  “key”: “param.tol”, //対象キー  “type”: “f32”, //データ型  “isArray”: true, //配列か？　※規定値は false。  //　　　　　　true 指定されたデータが、データ定義JSON上で  //　　　　　　配列として定義されていなければエラー。その逆も同様。  “arraySize”: [ 10 ] //配列の要素数　※配列の次元数分の要素数を指定。  //　　　　　　　　二次元配列なら [ 5, 10 ] のように記述する。  //　　　　　　　　JSONデータ上と要素数が一致しない場合はエラー。  },  //メンバー定義：不定長配列の場合  {  “name”: “abilities”, //メンバー名  “comment”: “アビリティ”, //コメント  “key”: “abilities”, //対象キー  “type”: “u32”, //データ型  “isVariableArray”: true, //不定長配列の指定　※isArray の指定を省略可。  “arraySizeName”: “abilitiesNum”,  //配列の要素数を記録するメンバー名  “arraySizeType”: “i8” //配列の要素数を記録するメンバーのデータ型  //　※省略時は i32とみなす  //※不定長配列の場合、構造体にはポインターと要素数の二つのメンバーが定義される。  //　通常、要素数のメンバーが先に、続いてポインターが並ぶ。  //　　（例）i8 abilityNum;  //　　　　　u32\* abilities;  //※実際のデータはバイナリデータの後部にまとめられ、ポインターはその位置を指す。  //　バイナリデータ上では、データのオフセット値が記録される。  },  //メンバー：ネストした構造体の場合  {  “name”: “param”, //メンバー名  “comment”: “パラメータ”, //コメント  “key”: “param”, //対象キー  “type”: “struct”, //データ型　※C言語の構造体をネストする場合は、  //　　　　　　“struct” を指定して、データ型名に構造体名を指定。  “typeName”: “T\_PARAM” //データ型名  //※実際の構造体は、”substructs” で定義する。  },  //メンバー定義：固定値の場合  {  “name”: “fixed”, //メンバー名  “comment”: “固定値”, //コメント  “key”: null, //対象キー　※対象キーに null を指定することで、JSONデータ側に  //　　　　　　存在しないメンバーを定義することが可能。  “type”: “i16”, //データ型  “default”: 1, //省略時の規定値　※固定値の指定に使用  },  //メンバー定義：無視項目の場合  {  “key”: “dummy”, //対象キー  “type”: “ignore”, //データ型　※データ型に “ignore” を指定すると、対象キーを無視する。  }  ],    “membersOrder”: //メンバーの並び順　※”members”の定義順と変えたい時だけ記述する。  //　　　　　　　　　　これを指定する場合、”members”の全項目を指定しなければエラー。  //　　　　　　　　　　“name” および “arraySizeName” を全て列挙する。  //　　　　　　　　　　アラインメントを考慮した配置にしたい場合などに使用する。  [  “id”, //ID：crc  “power”, //力：i8  “abilitiesNum”, //アビリティ(Num)：i8  “fixed”, //固定値：i16  “tol”, //耐性：f32[10]  “name”, //名前：str\*  “condition”, //有効化条件：T\_EXPR\*  “abilities”, //アビリティ：u32\*  “param” //パラメータ：T\_PARAM  ]  },    “substructs”: //ネストした構造体　※「struct」とほどぼ同様の構造だが、配列で複数の構造体を定義する。  [  {  “name”: “T\_PARAM”, //構造体名  “comment”: “パラメータ構造体”, //コメント    “isMakeSource”: true, //構造体の定義をC言語ヘッダーに出力するか？  “isInternalStructure”: true, //親の構造体の中にこの構造体を定義するかどうか？    “members”: //メンバー  [  {  “name”: “atk”, //メンバー名  “comment”: “攻撃力”,//コメント  “key”: “atk”, //対象キー　※JSONデータ上のネストしたデータのキーだが、  //　　　　　　親キーは指定しない。  //　　　　　※メンバー名と同じなら省略可能。  “type”: “i16” //データ型  },  {  “name”: “def”, //メンバー名  “comment”: “守備力”,//コメント  “type”: “i16” //データ型  },  {  “name”: “specials”, //メンバー名  “comment”: “特殊能力”, //コメント  “type”: “struct”, //データ型　※さらにネストした構造体も指定可能  “typeName”: “T\_SPECIAL\_PARAM”, //データ型名  “isVariableArray”: true, //不定長配列の指定  “arraySizeName”: “specialsNum” //配列の要素数  }  ],    “membersOrder”: //メンバーの並び順  [  “atk”, //攻撃力:i16  “def”, //防御力:i16  “specialNum”, //特殊能力(Num):s32  “special” //特殊能力:T\_SPECIAL\_PARAM\*  ]  },  {  “name”: “T\_SPECIAL\_PARAM”,  “comment”: “特殊パラメータ構造体”,//コメント    “isMakeSource”: true, //構造体の定義をC言語ヘッダーに出力するか？  “isInternalStructure”: false, //親の構造体の中にこの構造体を定義するかどうか？    “members”: //メンバー  [  {  “name”: “dark”, //メンバー名  “comment”: “闇”, //コメント  “type”: “u32” //データ型  },  {  “name”: “shine”, //メンバー名  “comment”: “光”, //コメント  “type”: “u32” //データ型  }  ]  }  ],    //エラー判定用ルール　※メンバーごとの min, max, isReuired 以外のルールを設定したい場合に用いる。  //　　　　　　　　　　※複数のルールを指定可。  //　　　　　　　　　　※メンバーの値を計算結果などで書き換えたい場合にも利用可能。  “rules”:  [  {  //ルール　※エラーメッセージ判定用のC#処理を記述。一塊の文字列として定義する。  //　　　　　エラーがある場合はエラーメッセージを return し、問題が無い場合は null を return。  //　　　　※一つのオブジェクト（構造体）が取り込まれる毎に実行され、  //　　　　　取り込んだオブジェクトは変数 obj として渡される。  //　　　　　なお、この時の obj は、strや可変長配列などの情報はポインター化（オフセット化）  //　　　　　されていないため、そのままメンバーにアクセスできる。crcなどの組み込み関数、  //　　　　　exprによる計算式解析もまだ行われていない状態。ルールを一通りパスした後に  //　　　　　それらの処理を行う。  //　　　　※エラーメッセージ出力時は、データ定義位置の行番号と、主キーの情報もいっしょに出力される。  //　　　　※obj のメンバーに値を代入すれば、そのまま実際の値として反映される。  //　　　　※行が連結されてしまうので、「//」によるコメントの記述は禁止  “rule”:  “ \  var atk = obj.param.atk; \  var def = obj.param.def; \  if(atk < def) \  { \  return \“ATK(\”+ atk + \”)は、DEF(\”+ def + \“) 以上の値にして下さい。\”; \  } \  obj.tol[2] = obj.tol[0] + obj.tol[1]; /\*値を書き換え可能\*/ \  return null; \  “  },  {  //ルール  “rule”:  “ \  if(obj.tol[0] > 0 && obj.tol[1] > 0) \  { \  return　\“「耐性」は、どれか一つだけ入力して下さい。\”; \  } \  return null; \  “  }  ],  } |

## データ型定義リスト

データフォーマットで使用するデータ型を定義するためのJSON。

基本的な記述仕様は「拡張JSON」形式に従う。記述内容は、データ型定義のための設定項目が定められている。下記のサンプルでその仕様を示す。赤い字で書かれた項目が、サンプル中で初出の定義項目であり、詳しい説明を併記する。

例：

|  |
| --- |
| //データ型定義リスト  //※「データ型」の内容を定義する。  //※プリミティブな型のみに対応し、構造体や配列は定義できない。  [  //u16\_ex 型  {  “type”: “u16\_ex”, //データ型  “typeName”: “unsigned int”, //データ型名　※C言語ソース用  “baseType”: “int”, //基本データ型　※int（整数） / float（浮動小数点） / dec（固定小数点）  //　　　　　　　　bool（真偽値）/ str（文字列）/ expr（計算式）  //　　　　　　　　ptr（ポインター）のいずれかで指定する。  //　　　　　　　※ptrは便宜上存在。任意のデータを扱うことができないが、  //　　　　　　　　構造体にポインター型のメンバーを含めたい場合に  //　　　　　　　　使用する。  “isUnsigned”: true, //符号無し指定　※基本データ型が int の場合のみ指定可。  “size”: 2, //データサイズ　※基本データ型が int の場合は 1,2,4,8 のいずれか。  //　　　　　　　　float の場合は 2, 4, 8 のいずれか。  //　　　　　　　　dec の場合は 2, 4, 8 のいずれか。  //　　　　　　　　str/scpr/ptr の場合はサイズ指定不要。ポインターの  //　　　　　　　　サイズ（32/64bit）になる。  “decBits”: 8, //固定小数の小数部のビット数　※基本データ型が dec の場合のみ指定可。  //　　　　　　　　　　　　　　　省略時はデータサイズの半分 -1 の  //　　　　　　　　　　　　　　　ビット数。2バイトなら 7bit、4バイト  //　　　　　　　　　　　　　　　なら15bit、8バイトなら31bit。  “min”: -10000, //最小値　※オプションで指定可。省略時は基本データ型、符号無し指定、  //　　　　　データサイズから自動判定。  “max”: 10000, //最大値　※（同上）  “default”: 1 //規定値　※オプションで指定可。省略時は 0 。  },  //crc 型  {  “type”: “crc”, //データ型  “typeName”: “unsinged int”, //データ型名  “baseType”: “int”, //基本データ型  “isUnsigned”: true, //符号無し指定  “size”: 4, //データサイズ  “func”: “crc” //組み込み関数　※バイナリデータに変換する際に適用する  //　　　　　　　　組み込み関数を指定する。  },  //文字列型  {  “type”: “str”, //データ型  “typeName”: “const char\*”, //データ型名  “baseType”: “str” //基本データ型　※str が指定されたデータは、JSONデータをバイナリ  //　　　　　　　　データに変換した際、データ後部に実際のデータ  //　　　　　　　　（文字列）をまとめ、その参照を扱うようになる。  },  //計算式型  {  “type”: “expr”, //データ型  “typeName”: “T\_EXPR\*”, //データ型名  “baseType”: “expr”, //基本データ型　※expr が指定されたデータは、JSONデータをバイナリ  //　　　　　　　　データに変換した際、データ後部に実際のデータ  //　　　　　　　　（T\_EXPR型の計算式データ）をまとめ、その参照を  //　　　　　　　　扱うようになる。  “func”: “expr” //組み込み関数　※expr()組み込み関数は、計算式が記述された  //　　　　　　　　文字列を、T\_EXPR型（不定長のデータ部を含む）の  //　　　　　　　　バイナリデータに変換する。  }  ] |

データ型定義リストは、複数の定義ファイルを変換ツール実行時に指定することが可能。

もっとも基本的な定義リストには、下記のデータ型が定義される。

bool 真偽値：1バイト：true/false

i8 符号付き8bit整数：1バイト：-128 ～ 127

u8 符号無し8bit整数：1バイト：0 ～ 255

i16 符号付き16bit整数：2バイト：-32,768 ～ 32,767

u16 符号なし16bit整数：2バイト：0 ～ 65,535

i32 符号付き32bit整数：4バイト：-2,147,483,648 ～ 2,147,483,647

u32 符号なし32bit整数：4バイト：0 ～ 4,294,967,295

i64 符号付き64bit整数：8バイト：-9,223,372,036,854,775,808

～ 9,223,372,036,854,775,807

u64 符号なし64bit整数：8バイト：0 ～ 18,446,744,073,709,551,615

f32 32bit浮動小数点：4バイト：1 + 8 + 23 bits（符号＋指数＋仮数）

f64 64bit浮動小数点：8バイト：1 + 11 + 52 bits（符号＋指数＋仮数）

str 文字列：ポインター（オフセット値）＋実データ（char[]型）

expr 計算式：ポインター（オフセット値）＋実データ（T\_EXPR型＋データ長）

crc 符号なし32bit整数：4バイト：32bit CRC値

crcs 符号なし32bit整数：4バイト：32bit CRC値（文字列を小文字化して算出）

## 計算式用拡張関数定義リスト

「データ定義JSON」内の「計算式」で使用する拡張関数を定義するためのJSON。

なお、これはあくまでも「計算式」の中で、ランタイム時に実行される関数を指定するためのものであり、バイナリデータ変換時には処理されないことに注意。（組み込み関数の場合は、パラメータに動的な要素がない限りは、データ変換時にも処理される。）

基本的な記述仕様は「拡張JSON」形式に従う。記述内容は、拡張関数定義のための設定項目が定められている。下記のサンプルでその仕様を示す。赤い字で書かれた項目が、サンプル中で初出の定義項目であり、詳しい説明を併記する。

例：

|  |
| --- |
| //拡張関数定義リスト  //※「拡張関数」の名前とパラメータを定義する。  [  //フラグを更新：bool setFlag(“フラグ名”, bool)  {  “func”: “getChapter”, //関数名  “args”: [ “str”, “bool” ], //パラメータ　※データ型を列挙  “return”: “bool”, //戻り値　※データ型を一つ指定  },  //現在の章を取得：u32 getChapter()  {  “func”: “getChapter”, //関数名  “args”: [ ], //パラメータ　※パラメータがない場合は空の配列を指定するか、  //　　　　　　　args 自体を指定しない。  “return”: “u32”, //戻り値  },  //デバッグプリント：bool debugPrint()  {  “func”: “debugPRint”, //関数名  “args”: [ “str”, “…”], //パラメータ　※不定長パラメータには “…”（ピリオド三つ）を指定する。  “return”: “bool”, //戻り値  }  ] |

## チェック用JSON

バイナリデータ出力が成功した時にだけ一緒に出力される。

チェック用JSONは拡張仕様を排除したJSON仕様のフォーマットのため、データの二次利用にも活用できる。

バイナリデータの構造に合わせた構造だが、文字列や計算式、不定長配列などのポインター（オフセット）要素は展開されず、そのまま本来の位置に記述される。crcなどの組み込み関数は計算結果が出力され、計算式のようなバイナリデータはBASE64エンコードされた文字列が出力される。出力されるデータの並び順は、指定された「主キー」に基づいて並べ替えされた状態となる。「副キー」と「インデックス」に指定されたインデックステーブルも別ファイルに出力され、内容を確認することができる。以下にそれらのサンプルを示す。

例：データ定義JSON ⇒ チェック用JSON＆インデックスリスト（主キー = ”id”、副キー = “name”、インデックス = “kana” を設定し、かつ、”kana”は実機用バイナリデータに出力されないものとする）

|  |
| --- |
| //キャラ定義  #include “header.jsonh”  [  //キャラ：山田  {  “id”: “c0010”, //ID：主キー  “name”: “山田”, //名前：副キー  “kana”: “やまだ”, //読み：インデックス  “condition”: “getChapter() >= 20”,  “param”:  {  “atk”: 10,  “def”: 20  },  “abilities”: [ “aaa”, “bbb” ]  },  //キャラ：田中  {  “id”: “c0020”, //ID：主キー  “name”: “田中”, //名前：副キー  “kana”: “たなか”, //読み：インデックス  “param”:  {  “atk”: 11,  “def”: 21  },  },  //キャラ：佐藤  {  “id”: “c0030”, //ID：主キー  “name”: “佐藤”, //名前：副キー  “kana”: “さとう”, //読み：インデックス  “param”:  {  “atk”: 12,  “def”: 22  },  “abilities”: [ “xxx” ]  }  ] |

↓ （変換後）

チェック用JSON： ※ ”id”がCRC値に変換され、”condition”のバイナリコードがBASE64に変換され、リストの並びが主キー（”id”）の順に変わり、kana”が消滅し、abilitiesなどの入力が省略された項目が網羅されている。

|  |
| --- |
| [  {  “id”: 123306860,  “name”: “田中”,  “condition”: null,  “param”:  {  “atk”: 11,  “def”: 21  },  “abilities”: [ ]  },  {  “id”: 507687469,  “name”: “佐藤”,  “condition”: null,  “param”:  {  “atk”: 12,  “def”: 22  },  “abilities”: [ “xxx” ]  },  {  “id”: 745853103,  “sortOrder”: 1,  “condition”: “Z2V0Q2hhcHRlcigpID49IDIweA==”,  “param”:  {  “atk”: 10,  “def”: 20  },  “abilities”: [ “aaa”, “bbb” ]  },  ] |

副キー検索用インデックステーブル： ※ ”name” の CRC順。”key”は”name”のCRC値を、”index”は出力データ（チェック用JSON）の位置を示す。

|  |
| --- |
| [  { “key”: 230889854, “index”: 2 },  { “key”: 558509826, “index”: 1 },  { “key”: 2850539082, “index”: 0 }  ] |

並べ替え用インデックステーブル： ※ ”kana” 順（さとう→たなか→やまだ）。数値は出力データ（チェック用JSON）の位置を示す。

|  |
| --- |
| [ 1, 0, 2 ] |

## C言語ソース

構造体が定義されたヘッダーファイルと、バージョン整合用の構造定義情報が定義されたソースファイル。

データ変換ツールのオプションにより出力可能。その際、「データ定義JSON」は必要なく、「フォーマット定義JSON」に基づいて作成される。

「バージョン整合用構造定義」は、T\_GAME\_DATA\_DECL型のデータで定義されるstaticなデータ。この情報を取得するための関数 getGameDataDecl() も合わせて自動生成される。

以下に自動生成されたファイルのサンプルを示す。T\_GAME\_DATA\_DECL型の扱いについてもサンプルで示す。なお、この内容は、前述の「フォーマット定義JSON」のサンプルに基づく。

例：ヘッダーファイル：charaData.h

|  |
| --- |
| #pragma once  #ifndef \_\_CHARA\_DATA\_H\_\_  #define \_\_CHARA\_DATA\_H\_\_  //ATTENTION!  //Do not modify this file manually.  #include “types.h”  namespace charaDataDef  {  //キャラパラメータ構造定義  //Name: CharaData  //Version: 1.0  //Update: 204.1.10 12:34:56  //特殊パラメータ構造体  struct T\_SPECIAL\_PARAM  {  unsigned int dark; //闇  unsigned int shine; //光  };  //キャラ構造体  struct T\_CHARA  {  //パラメータ構造体  struct T\_PARAM  {  short atk; //攻撃力  short def; //守備力  int specialNum; //特殊能力 (Num)  T\_SPECIAL\_PARAM\* special; //特殊能力  };    CRC id; //識別ID  char power; //力  char abilitiesNum; //アビリティ (Num)  short fixed; //固定値  float tol[10]; //耐性  const char\* name; //名前  const T\_EXPR\* condition; //有効化条件  unsigned int\* abilities; //アビリティ  T\_PARAM param; //パラメータ  };  }//namespace charaDataDef  #endif//\_\_CHARA\_DATA\_H\_\_ |

例：バージョン整合用構造定義ファイル：charaDataDecl.cpp

|  |
| --- |
| //ATTENTION!  //Do not modify this file manually.  #include “gameDataDecl.h”  /\*  ※このインクルード内で下記の構造体が定義されている。  　本来このコメントの部分はファイル出力されないが、  　このサンプルでは内容を分かり易くするために記述している。  //ゲームデータ定義  struct T\_GAME\_DATA\_DECL  {  //ゲームデータ構造体定義  struct STRUCT  {  //ゲームデータ構造体メンバー定義  struct MEMBER  {  unsigned int memberNameCrc; //メンバー名CRC  unsigned int dataNameCrc; //構造体名CRC　※型種別 = struct の場合に指定。  unsigned char baseType:3; //型種別　※0 = int, 1 = float, 2 = dec, 3 = bool, 4 = str、  //　　　　　5 = expr, 6 = ptr, 7 = struct。  unsigned char isUnsigned:1; //符号無し型か？　※型種別 = int の場合のみ指定可。  unsigned char size:4; //型のサイズ　※int の場合は、1,2,4,8 のいずれか。  //　　　　　　　float の場合は、2,4,8 のいずれか。  //　　　　　　　dec の場合は 2,4,8 のいずれか。  //　　　　　　　bool の場合は 1。  //　　　　　　　str / expr / ptr /struct の場合は 0。  unsigned char padding[3]; //パディング　※0xdbで埋める  unsigned int isVariableArray:1;//不定長配列か？　※true で不定長配列。この時、メンバーの実際の  //　　　　　　　　　　データ型はポインターとなる。  unsigned int arraySize:16; //配列要素数　※配列でない時や不定長の時は 0。  };    unsigned int structNameCrc; //構造体名CRC  unsigned short membersNum; //構造体メンバー数  MEMBER\* members; //構造体メンバー定義の参照  };    unsigned int formatNameCrc; //データフォーマット名CRC  unsigned short majorVer; //データフォーマットメジャーバージョン  unsigned short minorVer; //データフォーマットマイナーバージョン  unsigned short structsNum; //構造体数  STRUCT\* structs; //構造体定義の参照  };  \*/  namespace charaDataDecl  {  //キャラパラメータ構造定義  //Name: CharaData  //Version: 1.0  //Update: 204.1.10 12:34:56  //データ構造定義  //構造体のメンバー定義  static const int s\_structMembersNum = 9 + 4 + 2; //全構造体の全メンバー数  static T\_GAME\_DATA\_DECL::STRUCT::MEMBER s\_structMembers[s\_structMembersNum] =  {  //T\_CHARA構造体  { 0xe66c3671, 0x00000000, 0, true, 4, false, 0 }, //識別ID:”id”:crc  { 0xab8a01a0, 0x00000000, 0, false, 1, false, 0 }, //力:”power”:i8  { 0x9c504212, 0x00000000, 0, false, 1, false, 0 }, //アビリティ(Num):”abiriliesNum”:i8  { 0x9ec9ce32, 0x00000000, 0, false, 2, false, 0 }, //固定値:”fixed”:i16  { 0x5e237e06, 0x00000000, 4, false, 0, false, 0 }, //名前:”name”:str  { 0xbdd68843, 0x00000000, 5, false, 0, false, 0 }, //有効化条件:”condition”:expr  { 0x0ad84385, 0x00000000, 1, false, 1, false, 10 }, //耐性:”tol”:f32  { 0xb8388da4, 0x00000000, 0, true, 4, true, 0 }, //アビリティ:”abilities”:u32\*  { 0xa4fa7c89, 0x22a2e1dc, 7, false, 0, false, 0 }, //パラメータ:”param”:T\_PARAM  //T\_PARAM構造体  { 0x27677c27, 0x00000000, 0, false, 2, false, 0 }, //攻撃力:”atk”:i16  { 0x0cc4e161, 0x00000000, 0, false, 2, false, 0 }, //防御力:”def”:i16  { 0xcd1415d7, 0x00000000, 0, false, 4, false, 0 }, //特殊能力(Num):”specialsNum”:i32  { 0x4c6b3fe3, 0x84f96e44, 7, false, 0, true, 0 }, //特殊能力:”specials”:T\_SPECIAL\_PARAM\*  //T\_SPECIAL\_PARAM構造体  { 0x1b7cbdfb, 0x00000000, 0, true, 4, false, 0 }, //闇:”dark”:u32  { 0x076291bf, 0x00000000, 0, true, 4, false, 0 }, //光:”shine”:u32  };  //構造体の定義  static const int s\_structsNum = 3; //全構造体数  static T\_GAME\_DATA\_DECL::STRUCT s\_structs[s\_structsNum] =  {  { 0x0ed35394, 9, &s\_structMembers[0] }, //T\_CHARA  { 0x22a2e1dc, 4, &s\_structMembers[9] }, //T\_PARAM  { 0x84f96e44, 2, &s\_structMembers[13] }, //T\_SPECIAL\_PARAM  };  //フォーマット情報定義（バージョンを含む）  static T\_GAME\_DATA\_DECL s\_decl =  {  0xe1e4a645, //フォーマット名CRC：CharaData  1, //フォーマットメジャーバージョン  0, //フォーマットマイナーバージョン  s\_structsNum,//構造体数  &s\_structs[0]//構造体定義の参照  };  //データ構造定義取得関数  const T\_GAME\_DATA\_DECL\* getGameDataDecl(){return &s\_decl;}  }//namespace charaDataDecl |

「バージョン整合用構造定義」データは、以下の二つの目的で使用される。

1. バージョン整合判定：

バイナリデータ取り込み時に、バイナリデータのデータ構造と実機側のデータ構造が一致しているかどうかの判定に用いる。バイナリデータのヘッダー部にも同様の「バージョン整合用構造定義」データが記録されており、両者を比較する。両者が一致する場合はバイナリデータをまるごとコピーできるが、不一致だった場合は、両者の情報を比較しながら、名前が一致する項目のデータを一つずつコピーする。

1. ポインター補正：

バイナリデータ取り込み後、文字列、計算式、不定長配列のデータは、データのオフセットが記録されているため、ポインターに修正する。

## バイナリデータ

「データ定義JSON」が変換された実機向けのデータ。

基本構造は下記の通り。





















「データ部」内のデータには、「可変長データ」「計算式データ」「文字列データ」が含まれる。それらは、それぞれ「可変長データ部」「計算式データ部」「文字列データ部」を指すポインター型で定義されているが、「バイナリデータ」内では、オフセット値が記録されているため、それをポインターに変換する必要がある。

それぞれのオフセット値は、いずれもバイナリデータの先頭からのオフセット値になっているため、単純に先頭ポインターを加算するだけで変換できる。オフセット値が 0 のデータは nullptr。

どのデータ項目をポインターに変換する必要があるかは、ソースコードに組み込まれている「バージョン整合用構造定義」に基づく。

また、データ取り込み時には、この「バージョン整合用構造定義」とバイナリデータ内の「構造定義部」の内容を比較し、そのままコピー可能かどうか判定する。構造が不一致の場合、全体サイズを計算してメモリを確保し、必要な情報を一つずつコピーする。この時、ヘッダー部の各種オフセット値は書き換えられるが、「構造定義部」はそのままとなる。

データ取り込み時に、「デバッグデータ部」を取り込むかどうか指定することが可能。取り込まなかった場合は、ヘッダー部内の「デバッグデータ部のオフセット」を0がになり、デバッグテキストの参照ができなくなる。

# 値の演算と組み込み関数

拡張JSONの仕様により、「データ定義JSON」の「値」には、演算と組み込み関数の使用が可能。データ変換時に演算が行われ、算出された値がバイナリデータに出力される。

使用可能な演算子と組み込み関数は、後述する「計算式」とほぼ同じ。仕様も「計算式」の説明に記述。

演算と組み込み関数の扱いについて、「計算式」との違いは下記の通り。

拡張JSON上では、「+」演算子による文字列の連結が可能。計算式では不可。

計算式では、expr() 関数が使用不可。expr() 関数はデータ定義JSON専用。

## 計算式解析関数：expr()

関数名：expr

パラメータ：計算式文字列

戻り値：T\_EXPR型（計算式型）

用例：expr(“getChapter() >= 20 && getFlag(\”flag1\”) == On”)

計算式の文字列を、ランタイム時に実行可能なバイナリの計算式データに変換して返す。計算式に関する仕様は後述。

# 計算式

「データ定義JSON」内で、ランタイム時に実行するための計算式を使用できる。

例えば、会話テキストの選択条件として「○章以降なら」「XXXフラグが立っていたならら」「△章時にフラグＡ=OFFでフラグＢ=ONなら」といった条件を与えたい場合などに使用する。汎用的なデータであるため、Excelシートの会話テキストに並べて条件判定式として記述するようなことが可能。

例：



複雑な判定条件が必要になっても、データシートを拡張せずに対応できる点も利点。拡張関数さえ登録すれば、どのようなタイトルでも、タイトル依存の情報を扱うことができる。

章判定やフラグ判定などの基本的な要素は専用の入力欄を設けて、実機用のデータにする際に計算式化するような方法も有効。この場合、判定要素の追加が必要になってExcelシートを拡張することがあっても、実機用のデータはそのまま（一つの計算式があるだけ）にして、少しでも手間を軽減することができる。

以降、計算式データに関する仕様を示す。

## 計算式の構文

計算式の構文は、下記の要素で成り立つ。

値（整数、小数、真偽値）

* 整数には、10進表記、8進表記（0 + 0~7）、16進表記（0x + 0~f）、2進表記（0b + 0~1）が使用可能。
* 真偽値は、真値に true, yes, on のいずれか、偽値に false, no, off のいずれかを記述可能。大文字／小文字は問わない。

文字列

* 文字列はダブルクォーテーションもしくはシングルクォーテーションのどちらかで囲む。
* 文字列内にはエスケープ文字を使用可能。使用可能なエスケープ文字は下記のとおり。

\”（ダブルクォーテーション）

\’（シングルクォーテーション）

\n（改行）／ \r（改行）／ \r\n（改行） ※どれも単一の改行として扱われる

\\（\マーク）

単項演算子（+, -, ~, !）※C言語準拠

四則演算子（+, -, \* , /, %）※C言語準拠

ビット演算子（&, |, ^, ~, <<, >>, >>>）※C言語／Java準拠

判定式（==, !=, >, <, >=, <=）※C言語準拠

論理演算子（&&, ||）※C言語準拠

三項演算子（x ? y : z）※C言語準拠

かっこ（丸かっこのみ）※C言語準拠

組み込み関数　※あらかじめ用意されている関数。

拡張関数　※タイトル固有の関数。

「,」（コンマ）　※関数のパラメータの区切り用。

「;」（セミコロン）　※演算と無関係に複数の関数を呼び出したい時用。

なお、下記の要素には対応しない。

変数

代入および代入を伴う演算（=, +=, -=, \*=, /=, ++, --）

## 応演算子

文字列の演算には対応しない。例えば、「+」演算子で文字列連結といったことはできない。基本的に、文字列操作のような、メモリ操作を要する処理は計算式内では実行できない。

リスト中の一部の説明「…なら右値を評価しない」とあるのは、例えば、「getChapter() >= 30 && getFlag(“flag”) == ON」という計算式があった場合、論理演算子「&&」の左値「getChaptere() >= 30」の結果が偽だった時点で、「&&」の結果が偽と確定するため、右値の「getFlag(“flag”) == ON」が評価されず、関数「getFlag()」も呼び出されないことを意味する。これはC言語と同じ挙動である。評価が省略されることにより、処理が少しでも効率化される。

## 組み込み関数

下記の組み込み関数に対応する。どれもありきたりの関数なので、細かい仕様は省略し、関数を列挙する。

crc(“str”) 文字列”str”のCRC値を算出

crcs(“str”) 文字列”str”を小文字化した文字列のCRC値を算出

pow(x, y) x の y 乗を算出

sqrt(x) x の平方根を算出

abs(x) x の絶対値を返す

sign(x) x の符号を返す（x=負…-1、x=0…0、x=正…1）

sin(x) x のsin値を算出　※xはラジアン

cos(x) x のcos値を算出　※xはラジアン

tan(x) x のtan値を算出　※xはラジアン

asin(x) x のasin値を算出　※xはsin値

acos(x) x のacos値を算出　※xはcos値

atan(x) x のatan値を算出　※xはtan値

atan2(x, y) x, y の atan2 値を算出

toRad(x) 度をラジアンに変換

toDeg(x) ラジアンを度に変換

pi() π（3.141592653589）を返す

なお、組み込み関数は、できる限りデータ変換時に計算結果を算出して値に置き換えるようにする。できない場合はランタイム時に計算される。基本的に、拡張関数がパラメータに含まれない限りは算出可能である。（例：crc(“c0010”)⇒可、pow(4,3)⇒可、sqrt(2.0 \* 2.0)⇒可、toRad(45)⇒可、cos(toRad(45))⇒可、pow(getUserLevel(),2)⇒不可）

## 拡張関数

拡張関数も使い方は組み込み関数と同様。

タイトル固有の関数を使用することができる。

データ変換時には、「拡張関数定義リスト」に基づいて、関数名とパラメータが正しいかどうかがチェックされる。

# 処理仕様

データ変換および実機でのデータ取り込みの処理仕様を示す。

## プリプロセッサ

プリプロセッサにはMinGW(GCC) を使用。

ダウンロードサイト： <http://sourceforge.net/projects/mingw/files/Installer/>

（インストール方法は省略）

実行手順①： MinGWのパスを通す。バッチファイルなどで、実行時にだけパスに追加する方法でも可。

（例）

SET PATH=C:\MinGW\bin;%PATH%

実行手順②： コマンドラインコンパイラ g++.exe を使用し、プリプロセッサを実行。

（例）

$ g++.exe –x c++ -E –c data.json –o data.i.json

【使用オプション】

-x c++ C++ 言語として処理

-E Cプリプロセッサだけを実行

-P #line 文を出力しない

-c 入力ファイルを指定

-o 出力ファイルを指定

-Idir インクルードファイルの検索パスを指定（複数指定可）

-include file

　 強制インクルードファイルを指定

-Dmacro

-Dmacro=defn

　 マクロを定義（=defn が無いときは 1 になる）

-M 依存ファイル（インクルードファイル）のリストアップ

-Mf file

　 依存ファイルのリストをファイルに出力

## データ変換ツール

開発言語：C#コンソールアプリケーション（.Net Framework4.0以上）

選定理由： 開発のし易さ、JSONパーサーの利用、dynamic型（遅延バインド）の利用。

使用ライブラリ：JSON. Net

<http://json.codeplex.com/downloads/get/744406>

ツール名：gdconv.exe　※Game Data CONVertor の意。

使用方法：（例）

$ gdconv.exe --be --p64 --sp8 --dbgcrc --dbgexpr --utf8 --dbgutf8 -t types.json -t ext\_types.json -f funcs.json -f ext\_funcs.json –d char\_format.json -i chara\_data.json -o chara\_data.bin -c chara\_data\_chk.json

【対応オプション】　※競合オプションは後に指定されたものが有効

--le リトルエンディアン　※デフォルト

--be ビッグエンディアン

--p32 ポインターサイズ：32bit（4バイト）　※デフォルト

--p64 ポインターサイズ：64bit（8バイト）

--sp1 構造体パッキング（アラインメント）：1バイト

--sp2 構造体パッキング（アラインメント）：2バイト

--sp4 構造体パッキング（アラインメント）：4バイト

--sp8 構造体パッキング（アラインメント）：8バイト　※デフォルト

--sp16 構造体パッキング（アラインメント）：16バイト

--dbgcrc crc/crcs変換前テキストをデバッグデータ化　※デフォルト

--dbgexpr 計算式変換前テキストをデバッグデータ化　※デフォルト

--nodbg デバッグデータなし

--utf8 エンコーディング：UTF-8　※デフォルト

--sjis エンコーディング：シフトJIS

--dbgutf8 デバッグデータエンコーディング：UTF-8　※デフォルト

--dbgsjis デバッグデータエンコーディング：シフトJIS

--makesrc ソースファイル生成　※-c,-o,-c,-m オプションは無視

--srcutf8 ソースファイルエンコーディング：UTF-8　※デフォルト

--srcsjis ソースファイルエンコーディング：シフトJIS

-t file データ型定義リストファイル指定　※複数指定可

-f file 計算式用拡張関数定義リストファイル指定　※複数指定可

-d file フォーマット定義JSONファイル指定　※必須

-i file データ定義JSONファイル（入力ファイル）指定

-o file バイナリファイル（出力ファイル）指定

-c file チェック用JSONファイル（出力ファイル）指定

-m file 中間JSON②ファイル（出力ファイル）指定

※C言語ヘッダーファイル（出力ファイル）とC言語ソース（構造定義情報）ファイル（出力ファイル）はデータ定義JSON内で指定。

※-c オプションがないとチェック用JSONは出力されない。

※-d オプションがないとエラー。

※--makesrc オプション非指定時は、-c,-o オプションがないとエラー。

各データ仕様および上記オプションに基づいてツールを作成。ツール作成時の注意点は下記のとおり。

主キー、副キーの重複エラー判定は、文字列とCRC値の両方で行う。

計算式データ、文字列データ、デバッグデータは、バイナリデータ化の際、同じ内容なら同じ実データを参照するようにする。

途中エラーが一つでも検出されたら、一切のファイル出力を行わず、その時点で処理を終了する。

途中エラーが一つでも検出されたら、コマンドは 1 を返す。（正常時は 0 を返す。）

内部での処理順序は下記のとおり。

データ型定義リスト解析（中間JSON②化→JSON解析→内部データ化）

⇒ フォーマット定義JSON解析（中間JSON②化→JSON解析→内部データ化）

⇒ フォーマット定義の整合性チェック

* 存在しない型が使用されていたらエラー、ネストした構造体が定義されていなければエラー、などをチェック。

【以下、C言語ソース出力時は行わない処理】

⇒ 計算式拡張関数定義リスト解析（中間JSON②化→JSON解析→内部データ化）

⇒ データ定義JSON解析（中間JSON②化→JSON解析→内部データ化）

* 中間JSON②作成と同時に、CRC値化された文字列をデバッグ用データとして蓄積。
* 同CRC値のデータが蓄積済みなら、文字列が一致するかチェックして不一致ならエラー。
* crcs()関数使用箇所は、小文字化した文字列で扱う（元の文字列は失われる）。

⇒ データのフォーマット照合

* フォーマット定義に存在しないキーが使用されていたらエラー、主キー／副キー／インデックス対象項目が定義されていなければエラー、など。

⇒ 不定長データを分割

* 基本のデータと、不定長データ（の型）の数だけリストが作られる。

⇒ 計算式（expr関数）解析処理（計算式データに変換して蓄積、元の計算式文字列をデバッグ用データとして蓄積）

* 計算式文字列のCRC値で計算式データ蓄積済みかどうかをチェックする。デバッグ用データも同様。
* 同CRC値のデバッグ用データが蓄積済みなら、計算式の文字列が一致するかチェックして不一致ならエラー。

⇒ 文字列解析処理（文字列データとして蓄積）

* 文字列のCRC値で文字列データ蓄積済みかどうかをチェックする。
* 同CRC値のデータが蓄積済みなら、文字列が一致するかチェックして不一致ならエラー。

⇒ 主キーによるデータソート

* 重複があったらエラー。
* 処理対象は基本のデータリストのみ。不定長データのリストはなにもしない。

⇒ 検索インデックス作成（副キー）

* 重複があったらエラー。

⇒ 並べ替えインデックス作成

⇒ バイナリデータのメモリサイズ計算

⇒ 不定長データ、計算式データ、文字列データのオフセット値化

* 処理対象は基本のデータリストと不定長データのリスト全て。

⇒ バイナリデータ出力

⇒ チェック用JSON出力

⇒ 中間JSON②出力

* 対象は、データ定義JSONの中間JSON②のみ。

【以下、C言語ソース出力時のみ行う処理】

⇒ C言語ヘッダーファイル出力

⇒ C言語ソース（構造定義情報）ファイル出力

## データ変換実行バッチ

プリプロセッサからデータ変換を実行するバッチのサンプル。

ビルドツールを使用して効率化する手法については後述。

サンプル:

|  |
| --- |
| @ECHO OFF  rem --- バッチファイルの存在するパス（ドライブ＆ディレクトリ）にカレントディレクトリを変更 ---  CD /D %~dp0  rem --- MinGW(GCC)のパスを通す ---  SET MINGW\_BIN\_DIR=D:\MinGW\bin  SET PATH=%MINGW\_BIN\_DIR%;%PATH%  rem --- プリプロセッサオプション ---  SET PP\_FLAGS=-x c++ -E -P  SET PP\_INC\_DIRS=-I . -I "lib/common"  rem SET PP\_INCLUDES=-include "common.hjson"  SET PP\_INCLUDES=  rem --- データ変換オプション ---  SET GD\_FLAGS=--le --p32 --sp8 --dbgcrc -dbgexpr --utf8 --dbgutf8  SET GD\_MKSRC\_FLAGS=--makesrc --srcutf8  SET GD\_TYPE\_FILE01=settings/common\_type  SET GD\_TYPE\_FILE02=settings/ext\_type  SET GD\_FUNC\_FILE01=settings/ext\_func01  SET GD\_FUNC\_FILE02=settings/ext\_func02  SET GD\_FORMAT\_FILE=settings/chara\_fomat  SET GD\_FORMAT\_INC\_FILE=include/chara\_format.h  SET GD\_FORMAT\_SRC\_FILE=src/chara\_format.cpp  rem --- 開始 ---  ECHO \*\*\*\*\* データ変換開始 \*\*\*\*\*  ECHO.  SET RESULT\_TOTAL=0  rem --- 型定義リストのプリプロセッサ実行 ---  CALL :PREPROCESSOR %GD\_TYPE\_FILE01%  CALL :PREPROCESSOR %GD\_TYPE\_FILE02%  rem --- 拡張関数定義リストのプリプロセッサ実行 ---  CALL :PREPROCESSOR %GD\_FUNC\_FILE01%  CALL :PREPROCESSOR %GD\_FUNC\_FILE02%  rem --- フォーマット定義JSONのプリプロセッサ実行 ---  CALL :PREPROCESSOR %GD\_FORMAT\_FILE%  rem --- ソースファイル作成 ---  CALL :GDSRCCONV %GD\_FORMAT\_FILE%  rem --- データ変換実行 ---  CALL :GDCONV %GD\_FORMAT\_FILE% data/data01 bin/data01  CALL :GDCONV %GD\_FORMAT\_FILE% data/data02 bin/data02  CALL :GDCONV %GD\_FORMAT\_FILE% data/data03 bin/data03  rem --- 終了 ---  ECHO.  ECHO エラー数: %RESULT\_TOTAL%  CALL :PAUSE\_MSG \*\*\*\*\* データ変換終了 \*\*\*\*\*  EXIT /B %RESULT\_TOTAL%  rem ========== プリプロセッサ実行 ==========  :PREPROCESSOR  SET JSON\_FILE=%1.json  SET JSON\_PP\_FILE=%2  IF "%JSON\_PP\_FILE%" == "" SET JSON\_PP\_FILE=%1  SET JSON\_PP\_FILE=%JSON\_PP\_FILE%.i.json  ECHO.  ECHO ----------------------------------------  ECHO プリプロセッサ実行: "%JSON\_FILE%" to "%JSON\_PP\_FILE%"  CALL :RESET\_ERRORLEVEL  CALL :RUN\_CMD g++.exe %PP\_FLAGS% %PP\_INC\_DIRS% %PP\_INCLUDES% -c "%JSON\_FILE%" -o "%JSON\_PP\_FILE%"  SET RESULT=%ERRORLEVEL%  IF NOT "%RESULT%" == "0" ECHO エラー！  IF NOT "%RESULT%" == "0" SET /A RESULT\_TOTAL=RESULT\_TOTAL+1  EXIT /B %RESULT%  rem ========== ソースファイル生成実行 ==========  :GDSRCCONV  SET FORMAT\_FILE=%1.i.json  ECHO.  ECHO ----------------------------------------  ECHO ソースファイル生成: "%FORMAT\_FILE%"  CALL :RESET\_ERRORLEVEL  CALL :RUN\_CMD gdconv.exe %GD\_MKSRC\_FLAGS% -t "%GD\_TYPE\_FILE01%.i.json" -t "%GD\_TYPE\_FILE02%.i.json" -f "%GD\_FUNC\_FILE01%.i.json" -f "%GD\_FUNC\_FILE02%.i.json" -d "%FORMAT\_FILE%"  SET RESULT=%ERRORLEVEL%  IF NOT "%RESULT%" == "0" ECHO エラー！  IF NOT "%RESULT%" == "0" SET /A RESULT\_TOTAL=RESULT\_TOTAL+1  EXIT /B %RESULT%  rem ========== データ変換実行 ==========  :GDCONV  SET FORMAT\_FILE=%1.i.json  SET JSON\_FILE=%2.json  SET JSON\_PP\_FILE=%2.i.json  SET JSON\_MID\_FILE=%2.m.json  SET JSON\_CHECK\_FILE=%2.c.json  SET BIN\_FILE=%3.bin  ECHO.  ECHO ----------------------------------------  ECHO データ変換: "%JSON\_FILE%" to "%BIN\_FILE%"  CALL :RESET\_ERRORLEVEL  CALL :RUN\_CMD g++.exe %PP\_FLAGS% %PP\_INC\_DIRS% %PP\_INCLUDES% -c "%JSON\_FILE%" -o "%JSON\_PP\_FILE%"  SET RESULT=%ERRORLEVEL%  IF NOT "%RESULT%" == "0" ECHO エラー！  IF NOT "%RESULT%" == "0" SET /A RESULT\_TOTAL=RESULT\_TOTAL+1  CALL :RUN\_CMD gdconv.exe %GD\_FLAGS% -t "%GD\_TYPE\_FILE01%.i.json" -t "%GD\_TYPE\_FILE02%.i.json" -f "%GD\_FUNC\_FILE01%.i.json" -f "%GD\_FUNC\_FILE02%.i.json" -d "%FORMAT\_FILE%" -i "%JSON\_PP\_FILE%" -o "%BIN\_FILE%" -c "%JSON\_CHECK\_FILE%" -m "%JSON\_MID\_FILE%"  SET RESULT=%ERRORLEVEL%  IF NOT "%RESULT%" == "0" ECHO エラー！  IF NOT "%RESULT%" == "0" SET /A RESULT\_TOTAL=RESULT\_TOTAL+1  EXIT /B %RESULT%  rem ========== 標準サブルーチン ==========  rem --- 標準サブルーチン：コマンドラインを表示して実行 ---  :RUN\_CMD  ECHO ^> %\*  %\*  EXIT /B %ERRORLEVEL%  rem --- 標準サブルーチン：メッセージ付きポーズ ---  :PAUSE\_MSG  SET MSG=%\*  IF "%MSG%" == "" SET MSG=続行するには何かキーを押してください．．．  ECHO.%MSG%  PAUSE >NUL  EXIT /B  rem --- ERRORLEVELリセット ---  :RESET\_ERRORLEVEL  DATE /t >NUL  EXIT /B |

## データ変換処理（C# Tips）

C#でJSONを活用する方法を解説。

### JSON.Netの使用

準備： まずC#のプロジェクトにて、「参照設定」に、JSON.NetのNewtonsoft.Json.dll を追加しておく。

処理例：C#プログラムソース

|  |
| --- |
| using System;  //JSON.Netのネームスペース使用指定  using Newtonsoft.Json;  using Newtonsoft.Json.Linq;  //JSONパーサーテスト  void test()  {  //JSONテキスト  string json\_text = @"  {  ""data1"":  [  { ""id"": ""c0010"", ""name"": ""田中"", },  { ""id"": ""c0020"", ""name"": ""山田"", },  { ""id"": ""c0030"", ""name"": ""佐藤"", },  ],  ""data2"":  [  [1, 2, 3],  [4, 5, 6],  [7, 8, 9, 10],  ]  }  ";    //JSONテキストを解析  //※本来は JObject 型を返すが、dynamic型（動的型付け）でないと、  //　遅延バインディングが行われないため、コンパイル時にエラーになってしまう。  dynamic json\_obj = JObject.Parse(json\_text);  //dynamic型を使用することにより、型宣言せずに、  //JSON内で定義されたメンバーに直接アクセスが可能  Console.WriteLine();  Console.WriteLine("data1[0].id=\"{0}\"", json\_obj.data1[0].id);  Console.WriteLine("data1[2].name=\"{0}\"", json\_obj.data1[2].name);  Console.WriteLine("data2[2][3]={0}", json\_obj.data2[2][3]);  //foreach も使用可  Console.WriteLine();  Console.WriteLine("data1=[");  foreach(dynamic rec in json\_obj.data1)  {  Console.WriteLine(" {{ id=\"{0}\", name=\"{1}\" }},", rec.id, rec.name);  }  Console.WriteLine("]");    //foreach のネストも使用可  Console.WriteLine();  Console.WriteLine("data2=[");  foreach (dynamic array1 in json\_obj.data2)  {  Console.Write(" [");  foreach (dynamic array2 in array1)  {  Console.Write(array2 + ",");  }  Console.WriteLine("]");  }  Console.WriteLine("]");  } |

↓ （処理結果）

|  |
| --- |
| data1[0].id="c0010"  data1[2].name="佐藤"  data2[2][3]=10  data1=[  { id="c0010", name="田中" },  { id="c0020", name="山田" },  { id="c0030", name="佐藤" },  ]  data2=[  [1,2,3,]  [4,5,6,]  [7,8,9,10,]  ] |

### JSON解析済みオブジェクトをC#スクリプト内で使用

準備： まずC#のプロジェクトにて、「参照設定」に「Microsoft.CSharp」を追加しておく。

処理例：C#プログラムソース

|  |
| --- |
| using System;  //JSON.Netのネームスペース使用指定  using Newtonsoft.Json;  using Newtonsoft.Json.Linq;  //C#コンパイラ使用指定  using Microsoft.CSharp;  using System.CodeDom.Compiler;  using System.Reflection;  //C#スクリプト＋JSON＋dynamic型テスト  void test()  {  //JSONテキスト  string json\_text = @"  {  ""name"": ""田中"",  ""data"": 10  }  ";    //JSONオブジェクト生成  dynamic json\_obj = JObject.Parse(json\_text);    //C#スクリプト  string cs\_source = @"  using System;  using Newtonsoft.Json;  using Newtonsoft.Json.Linq;  public class CScriptTest  {  public string checkJsonData(dynamic json\_obj)  {  //スクリプト外部から受け渡された型情報不明なJSON解析済みオブジェクトに直接アクセス  string msg = ""スクリプト内のメッセージテキスト("" + json\_obj.data + "")"";  json\_obj.name += ""さん"";  json\_obj.data += 1;  return msg;  }  }  ";    //コンパイラパラメータ生成  CompilerParameters param = new CompilerParameters();    //コンパイラパラメータ設定  param.GenerateExecutable = false; // 実行ファイル（.exe）を作らない  param.GenerateInMemory = true; // メモリ上にプログラム（スクリプト）を生成  param.IncludeDebugInformation = false; // デバッグ情報を付加しない    //スクリプト内で使用する.Netアセンブリをコンパイラパラメータに追加  //※以下の設定により、JSON.Net と dynamic 型が使用可能になる  param.ReferencedAssemblies.Add("System.dll");  param.ReferencedAssemblies.Add("System.Core.dll");  param.ReferencedAssemblies.Add("Microsoft.CSharp.dll");  param.ReferencedAssemblies.Add("Newtonsoft.Json.dll");    //C#コンパイラ生成  CSharpCodeProvider provider = new CSharpCodeProvider();    //C#スクリプトをコンパイル  CompilerResults script\_prog = provider.CompileAssemblyFromSource(param, cs\_source);  if (script\_prog.Errors.Count > 0)  {  //コンパイルエラーあり  Console.WriteLine("コンパイルエラー数 = " + script\_prog.Errors.Count);  foreach (var err in script\_prog.Errors)  {  Console.WriteLine("コンパイルエラー:" + err.ToString());  }  }  else  {  //コンパイルエラーなし    //スクリプト内の型（クラス）情報を取得  Type script\_class = script\_prog.CompiledAssembly.GetType("CScriptTest");    //インスタンスを生成（コンストラクタ呼び出し：パラメータなしコンストラクタ呼び出しのケース）  dynamic script\_obj = script\_class.GetConstructors()[0].Invoke(new Object[0]);    //実行前状態表示  Console.WriteLine();  Console.WriteLine("実行前：json\_objd.name=\"{0}\", .data={1}", json\_obj.name, json\_obj.data);    //スクリプト内のメソッド呼び出し　※JSONテキスト解析済みオブジェクトを受け渡し  string ret\_msg = script\_obj.checkJsonData(json\_obj);    //実行後結果表示  Console.WriteLine();  Console.WriteLine("実行後：json\_objd.name=\"{0}\", .data={1}", json\_obj.name, json\_obj.data);  if(ret\_msg != null)  {  Console.WriteLine(" 戻り値=\"{0}\"", ret\_msg);  }  }  } |

↓ （処理結果）

|  |
| --- |
| 実行前：json\_objd.name="田中", .data=10  実行後：json\_objd.name="田中さん", .data=11  戻り値="スクリプト内のメッセージテキスト(10)" |

## データ取り込み処理（ランタイム）

## データアクセス処理（ランタイム）

# 計算式処理仕様

## 計算式解析処理

データ変換の処理中に行われる「計算式解析処理」について、詳しく説明する。

基本的な処理の流れは、「字句解析（トークン分解）」⇒「抽象構文木作成」⇒「ポーランド記法によるバイナリデータ化」。





















## 計算式データの構造

## 計算式実行処理（ランタイム）

# データ変換作業環境の改善

## バッチファイル の利用

## SCons の利用

SCons

Python

以上

索引

Ｇ

GCC 29

Ｊ

JSON 6

Ｍ

MinGW 29

Ｐ

Python 43

Ｓ

SCons 43

け

計算式

逆ポーランド記法 40

ポーランド記法 40

ゲームデータ 1

以　上