DM2.0

DMLib_API 仕様書

Version 1.0

作成者

DM2.0 コンソーシアム

2020/03/23

更新履歴

Version	日付	内容	担当
1.0	2020/3/23	初版	DM2.0 コンソーシアム

目次

図 一覧

表 一覧

1. はじめに

本書は、ダイナミックマップ 2.0 コンソーシアムにて開発した『DM2.0 プラットフォーム（以下、DM2.0PF）』に接続するためのライブラリ（以下、DMLib）の API についての仕様書である。

1.1. 本書の概要および位置づけ

本書は、DM2.0PF のデータベースシステム（以下、DB システム）を利用するための API 仕様についてまとめ、DMLib を組み込んだ際の実装例について記述する事を目的とする。

1.2. 前提知識

本書は、DB システムを利用するための API 仕様について記述されている。そのためデータベースを扱うための前提知識であるクエリ言語については別紙である、『クエリ言語仕様書』を参照する事を推奨する。また DMLib はプログラミング言語として C++を採用しているため、読者にはプログラミング知識が必要とされる。C++のプログラミング知識に関しては、一般的なプログラミング言語解説書を参照する事を推奨する。

2. DM2.0PF における DMLib の位置づけ

本章では DM2.0PF における DMLib の位置づけや利用形態について説明する。

2.1. ダイナミックマップとは？

近年の交通分野では、車両に搭載されたセンサにより走行環境を認識し、ドライバへの警告や自動で危険を回避する高度な安全運転支援システム、自律走行を可能とする自動走行システムの研究・開発が加速している。また、交通流の円滑化、環境負荷軽減などを目的として、車両と道路インフラが連携する、協調 ITS(Intelligent Transport Systems)の利用も活発化している。しかし、単体のセンサで認識できる範囲は非常に限定的（レーザレーダの場合、射程は約 120m）であり、手前の物体に遮られて奥の物体が検知できない等の問題もある。複数の車両や道路インフラの間で、道路上の事象を検出したセンサ情報を共有できるようにすることが、交通サービスを発展させていく上でますます重要となっている。

車載システムの進化に伴い、道路地図の高度化も必要となっている。これまでの車載システムでは、道路地図を目的地までの経路を案内するナビゲーションに主に用いてきた。それに加えて今後は、自動運転システムが自車両の位置を推定するための情報として、周辺の地形・建物の凹凸などの空間特徴量を提供することや、車両や道路インフラから得られたセンサ情報に対して、交通ルール上の意味付け（どのレーン上の事象なのか、そのレーンと自分のいるレーンとはどういう関係なのか等）を提供することなど、新たな役割が道路地図に期待されるようになっている。そのようなニーズに応えるための高精度道路地図の作成に向けて、国内外の様々な組織で仕様検討や試作が行われている。

高精度の道路地図上に、センサなどから得た交通データ（動的情報，準動的情報，準静的情報）を重ねて、位置参照方式を用いてお互いに紐づけられるようにしたデータ集合は「**ダイナミックマップ**」と呼ばれている。ヨーロッパで提案された当初は、車両周辺の情報を扱うローカルダイナミックマップとして検討されてきたが、現在は広域を扱うように概念が拡張されており、ダイナミックマップは、自動走行システム等の高度な交通サービスを支えるために必要な情報基盤と位置づけられている。以下はダイナミックマップを構成する各種情報の代表例である。

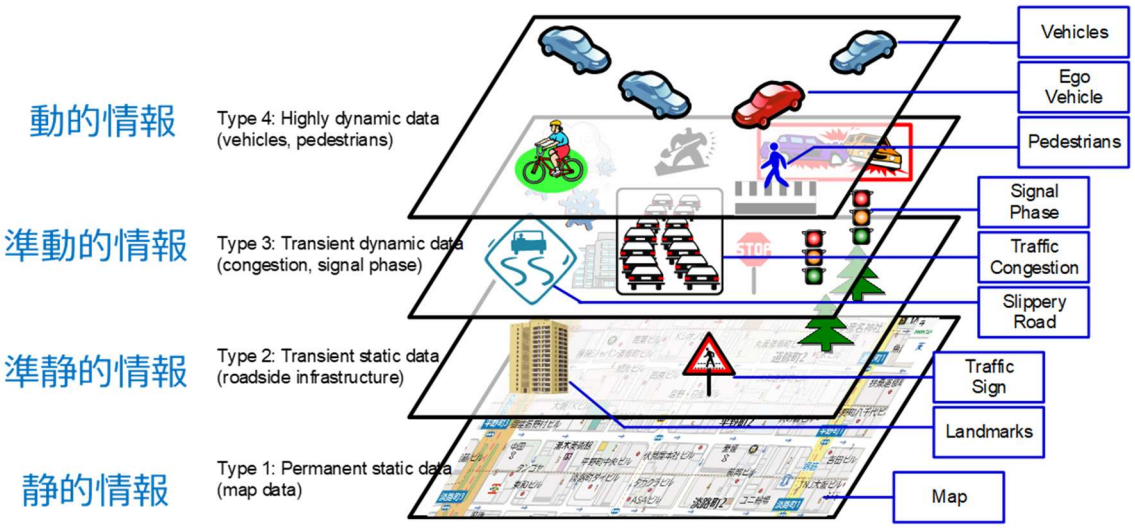

図 1 ダイナミックマップのイメージ

- 動的情報：移動体（車両、人など）の位置情報、信号の情報など
- 準動的情報：渋滞、道路工事、路面状態など
- 準静的情報：標識、ランドマーク、路側設備など
- 静的情報：道路地図

2.2.ダイナミックマップを用いたアプリケーション開発

ダイナミックマップを利用した交通サービスを発展させていくには次のような課題がある。交通サービスを実現するアプリケーションには、多くのセンサや地図データなど参照すべき情報が多数存在し、それぞれの情報に対するアクセス手段が異なっている点が課題である。またそれぞれのアプリケーションが多数存在する情報源にアクセスする場合には通信帯域を圧迫してしまう課題が発生する。加えて個々のアプリケーションの類似的な処理が共通モジュール化されないためソフトウェアの再利用が行われず、開発コストを下げることが出来ないといった課題もあり、これらを解決する必要がある（図 2 左）。

DM2.0PF とはダイナミックマップ 2.0 コンソーシアムが開発している交通情報管理のための共通開発基盤であり、上記で説明した課題の解決を目指している。DM2.0PF では交通参加者が通信により情報を交換する仕組みを共通化し、アプリケーションに対してそれらの情報を体系的に操作できるインターフェースを提供する。これにより複数種類の情報源に対して、アプリケーションは単一のアクセスで取得する事が可能となり、データ提供者(センサなど)とデータ利用者(アプリケーション)を疎結合にし、交通システム全体の複雑度を低減（掛け算から足し算に）する事が可能となる（図 2 右）。

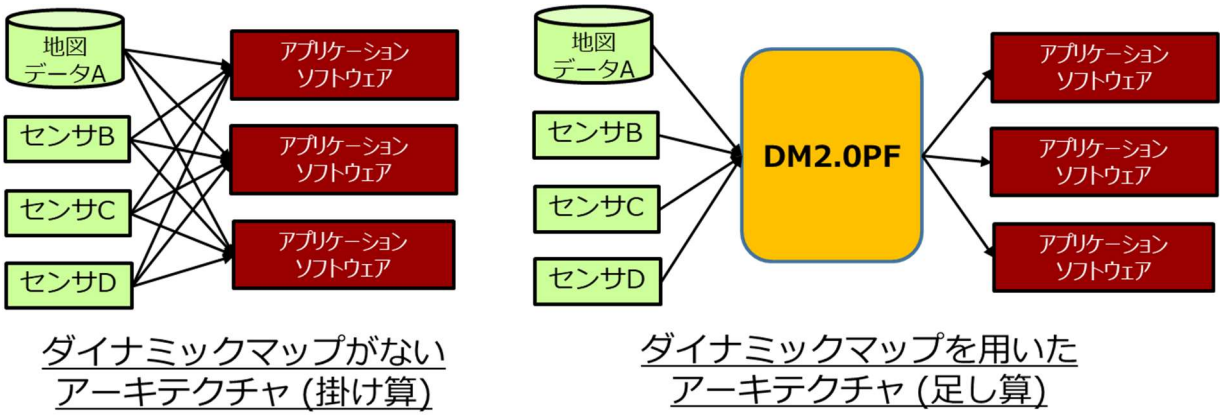

図 2 共通開発基盤としての DM2.0PF

このような共通開発基盤を整備する事で、多くのデータ提供者から情報を集約する事が出来、かつそれらの情報を活用する事で、交通サービスの発展につながる。この共通開発基盤こそが DM2.0PF が目指す姿である。

DM2.0PF は移動体の動的情報や、道路インフラの情報など多種にわたる情報を集約・管理し、交通管理のためのアプリケーション（図 3 参照）に対して体系的にデータを提供する。

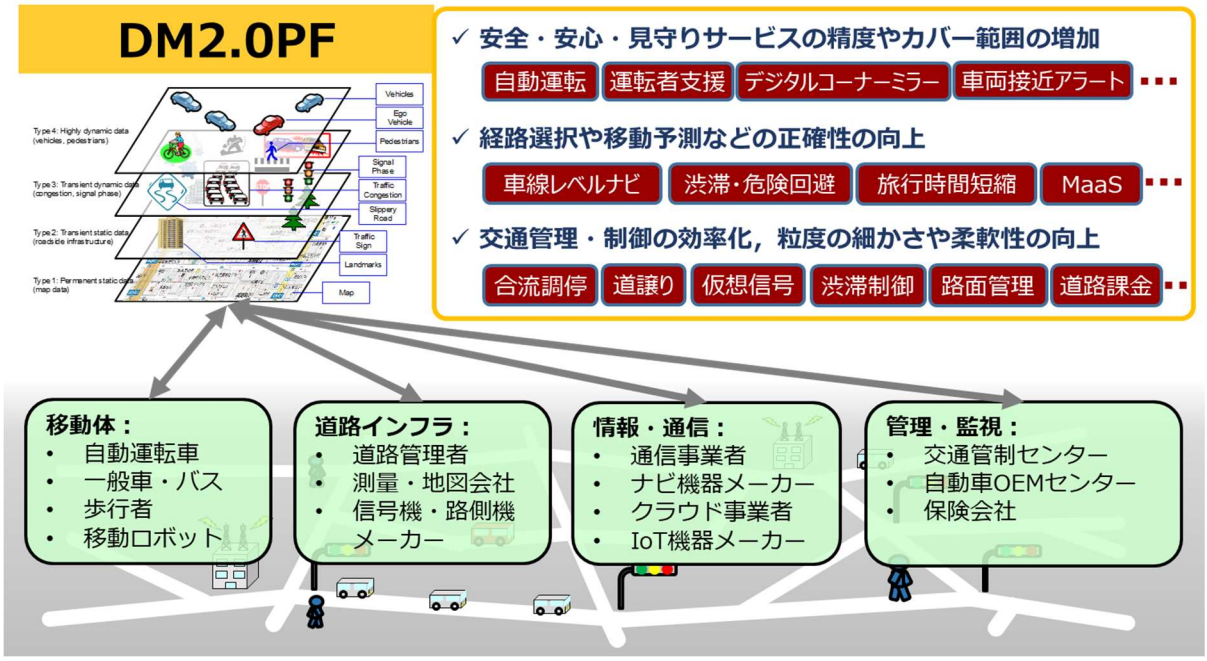

図 3 DM2.0PF が拓く新たな交通社会

2.3. DMLib の位置づけ

DM2.0PF はクラウド／エッジ／車両の三層構造から構成され、それぞれのノードに DB システムと通信部を配置している。全てのノードの DB システムと通信部を包括したものを DM2.0PF と総称する。DM2.0PF の三層構造については『ダイナミックマップ 2.0 プラットフォーム通信アーキテクチャ(ACCEAN)の研究』に説明があるので本書では説明は省略する。

本書の対象としている部分は DM2.0PF を利用して、交通アプリケーションを開発するためのライブラリ DMLib である。DMLib を利用する事により、DM2.0PF の DB システムにアクセスする事が可能となる。

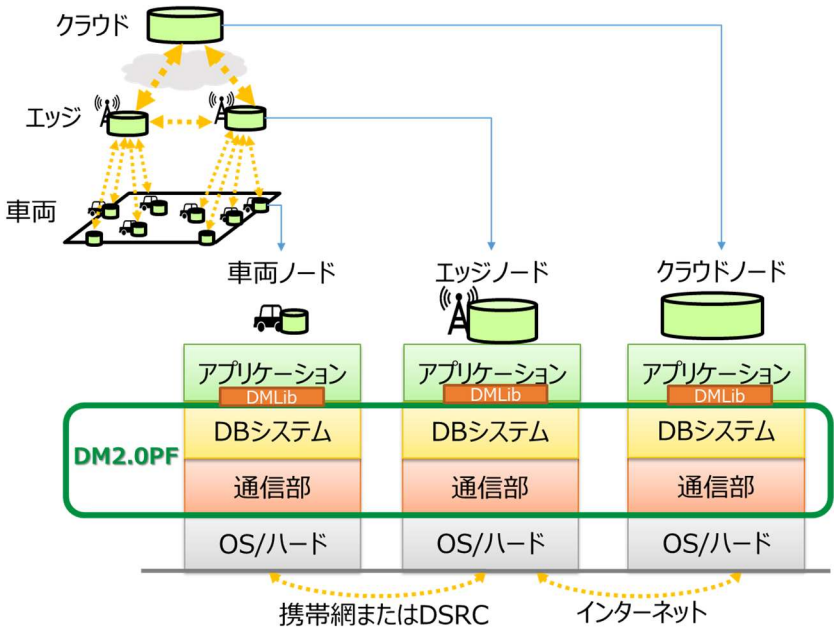

図 4 DM2.0PF の構成と DMLib の位置づけ

2.4. DMLib の利用形態

DMLib の利用形態は大きく分けて 2 つに分類する事が出来る。

1 つ目はデータ提供者としての利用方法で、DM2.0PF の DB システムに対して車両の位置情報やセンシング情報、信号の情報など動的情報の提供者として利用するパターンである。

2 つ目はデータ利用者としての利用方法で、DM2.0PF の DB システムに対して集約された動的、静的情報を DMLib のインターフェースを介して利用するパターンである。

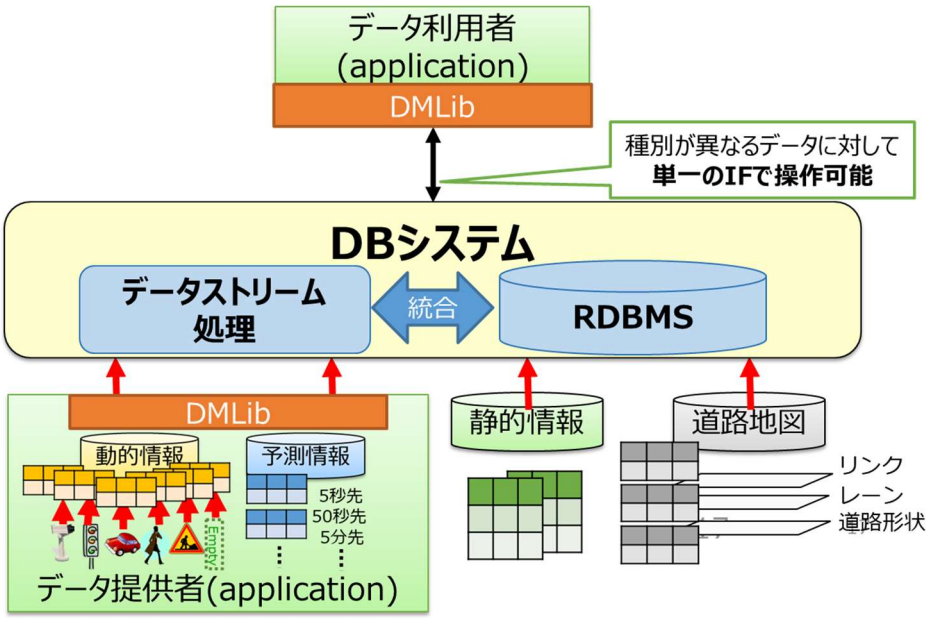

図 5 DB システムと DMLib との関係

DB システムでは動的情報や静的情報など、全ての情報がリレーション(データベースのテーブル)として表現・管理され、データ利用者は SQL に似たクエリ言語を使って、リレーションの中から、静的情報に対する検索、あるいは静的情報と動的情報を統合した検索など切り替えることが可能である。

2.5.データ操作要求の種類

DMLib のインターフェースを使って扱えるクエリは、２種類存在する。

データ操作要求に対して同期で結果を返す、主に静的情報に対する検索を**ワンショットクエリ**と呼んでいる。

対して予め DB システムでデータ操作要求を登録しておき、特定のイベント発生や条件に合うデータが発生する度に非同期で結果を返す、データ要求に動的情報を含む検索を**継続クエリ**と呼ぶ。継続クエリの活用例として“自車両が走行しているレーン前方 200m 以内に存在する他車両の位置情報が欲しい”といった要求を DB システムに登録した場合、前方レーンに走行中の他車両の位置情報を継続的かつリアルタイムに得る事が出来、無限に発生し続ける動的情報に対する検索および監視を行う事が出来る。クエリ言語仕様については『クエリ言語仕様書』を参照すること。

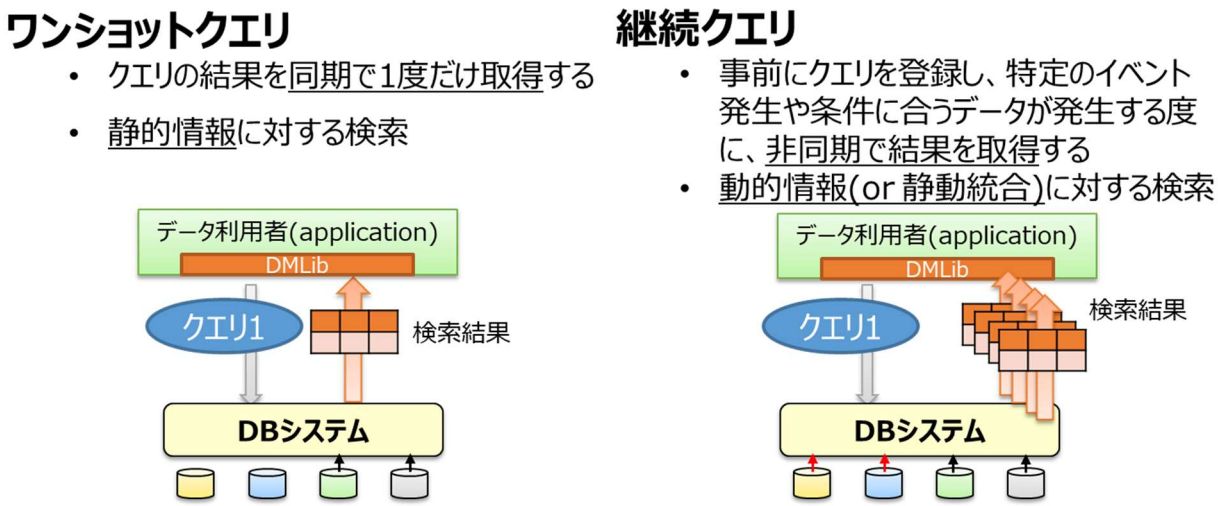

図 6 DMLib で扱うデータ操作要求

3. DMLib

DMLib は DB システムとの通信やデータ変換処理を隠蔽し、組み込んだアプリケーションに対して DM2.0PF の利用手段を提供するものである。本章では DMLib の概要や API について記載する。

3.1. DMLib 概要

DMLib は DB システムに対してデータ要求や操作をするためのライブラリである。DMLib を組み込む事によって、アプリケーションは以下の操作が可能となる。

表 1 DMLib で出来る操作

#	操作	概要
1	ワンショットクエリの発行	DB システムに対して静的情報の問い合わせやストリーム定義の生成および破棄の操作を実施する事が出来る。クエリ言語仕様については『クエリ言語仕様書』を参照すること。
2	継続クエリの登録	DB システムに対して動的情報、もしくは静動的統合情報を取得するための処理を登録する事が出来る。クエリ言語仕様については『クエリ言語仕様書』を参照すること。
3	継続クエリの結果受信	#2 で登録した継続クエリの結果を受信する事が出来る。
4	継続クエリのキャンセル	#2 で登録した継続クエリの処理をキャンセル(終了)する事が出来る。
5	ストリームデータの送信	動的情報を DB システムに送信する事が出来る。

DB システムと DMLib との間で利用されるプロトコルは以下のようになっている。尚、コネクション取得前に暗号化を有効にする API をコールする事で、通常時に TCP 通信していた部分を SSL 通信、UDP 通信していた部分を DTLS 通信する。

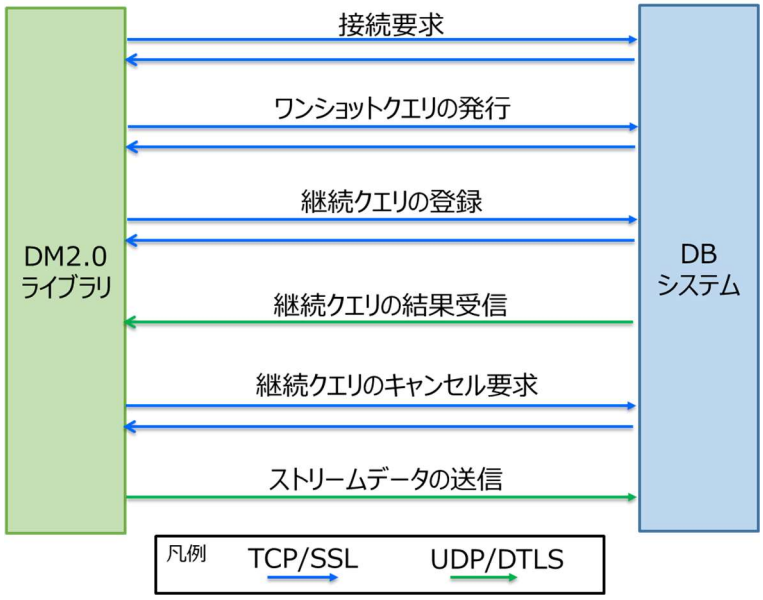

図 7 DMLib の利用プロトコル

3.2. DM ライブラリ API

DM ライブラリは DB システムとの通信やデータ変換処理を隠蔽し、組み込んだアプリケーションに対して DM プラットフォームの利用手段を提供するものである。以下に DMLib によって提供される API を記載する。

以下に DM ライブラリの主なインターフェース定義を記載する。

3.2.1. DmManager クラス

表 2 DmManager クラス IF 定義一覧

Return type	Method (args…、 exceptions…)	Description
Connection*	***getDBConnection(string ip, int port, string username, string password)*** **・Args** ip　：DB システムの IP アドレス port：DB システムの Listen しているポート番号 username：ユーザ名 password：パスワード **・Exceptions** ConnectionFailedException:接続失敗	DB システムに対して TCP 接続する。 ip、 port の引数は省略可能で、省略した場合は初期設定(ローカルホスト+規定ポート) にて接続を行う。ユーザ名とパスワードはユーザ認証時に使用する。
DatagramSocket*	***getDatagramSocket(string ip、 int port, string username, string password)*** **・Args** ip　：DB システムの IP アドレス port：DB システムの Listen しているポート番号 username：ユーザ名 password：パスワード **・Exceptions** ConnectionFailedException:接続失敗	DB システムに対して UDP 接続するためのオブジェクトを取得する。 ip、 port の引数は省略可能で、省略した場合は初期設定(ローカルホスト+規定ポート)にて設定した IP アドレスにて接続を行う。ユーザ名とパスワードはユーザ認証時に使用する。
void	***initEcryptionSettings(const string &inCertFilePath, const string &inPrivateKeyFilePath, const string &inPemPass)*** **・Args** inCertFilePath　：サーバ証明書ファイルパス inPrivateKeyFilePath：秘密鍵ファイルパス inPemPass：秘密鍵に設定した PEM パスワード	暗号化通信を有効にする。コネクションを取得する前に本 API をコールすると以降暗号化通信可能なコネクションを取得できるようになる。ファイルパスは絶対パスを指定する。

void	***initTcpSslTimeoutSetting(const int TcpSslTimeoutSec)*** **・Args** TcpSslTimeoutSec　：TCP および SSL のタイムアウト時間	TCP および SSL による DB システムへの要求に対して、レスポンスが返ってくるまでのタイムアウト時間を設定する。 デフォルト：60[秒]
void	***initDtlsTimeoutSetting(const int DtlsTimeoutSec)*** **・Args** DtlsTimeoutSec　：DTLS の認証タイムアウト時間	DTLS の認証タイムアウト時間を設定する。DTLS の受信開始もしくは最終受信時間から次の受信時間までのタイムアウト時間となる。 デフォルト：3600[秒]

3.2.2. Connection クラス

表 3 Connection クラス IF 定義一覧

Return type	Method (args…, exceptions…)	Description
ResultSet	***execute(string query)*** **・Args** query　：DB システムに問い合わせるクエリ文字列 **・Exceptions** ConnectionTimeoutException:タイムアウトエラー SQLException:クエリの文法エラー	DB システムに対してワンショットクエリを発行する TCP 通信にて同期処理でクエリ結果が返却される
void	***disconnect()*** **・Exceptions** ConnectionTimeoutException:タイムアウトエラー	TCP 通信の接続を切断する
unsigned int	***registerQuery(string query, FUNC_CALLBACK func)*** **・Args** query ：DB システムに問い合わせるクエリ文字列 func　：クエリ結果が返却された場合にコールされるコールバック関数 **・Exceptions** ConnectionTimeoutException:タイムアウトエラー SQLException:クエリの文法エラー	DB システムに対して継続クエリを発行する TCP 通信にて同期処理でクエリ管理番号が返却される。 継続クエリのクエリ結果は第 2 引数に指定したコールバック関数に UDP 通信にて非同期で返却される
unsigned int	***registerQuery(string query, std::function<void(ResultSet)> func)*** **・Args**	DB システムに対して継続クエリを発行する

	query ：DB システムに問い合わせるクエリ文字列 func ：クエリ結果が返却された場合にコールされるコールバック関数 **・Exceptions** ConnectionTimeoutException:タイムアウトエラー SQLException:クエリの文法エラー	TCP 通信にて同期処理でクエリ管理番号が返却される。 継続クエリのクエリ結果は第 2 引数に指定したコールバック関数に UDP 通信にて非同期で返却される
unsigned int	***registerQuery(string query, CallBackListener* cbl)*** **・Args** query ：DB システムに問い合わせるクエリ文字列 cbl ：CallBackListener クラスを継承したクラスを指定する **・Exceptions** ConnectionTimeoutException:タイムアウトエラー SQLException:クエリの文法エラー	DB システムに対して継続クエリを発行する TCP 通信にて同期処理でクエリ管理番号が返却される。 継続クエリのクエリ結果は第 2 引数に指定したクラスのコールバック関数に UDP 通信にて非同期で返却される
void	***cancelQuery(unsigned int managementId)*** **・Exceptions** ConnectionTimeoutException:タイムアウトエラー	継続クエリのキャンセル要求を発行する 第 1 引数に指定したクエリ管理番号の継続クエリを終了し、コールバックされなくなる
bool	***reconnect()*** **・Exceptions** ConnectionFailedException:接続失敗	DB システムに対して再接続する。 既に設定済みの設定にて再度接続を行う。(getDBConnection()にて引数が未指定の場合は初期設定)

表 4 Connection クラス 型定義一覧

Type	Define	Description
FUNC_CALLBACK	***Typedef void (*FUNC_CALLBACK)(ResultSet *)***	継続クエリのクエリ結果受信時にコールバックされる関数の型 引数に ReusltSet を持つ関数を宣言する必要がある

3.2.3. ResultSet クラス

表 5 ResultSet クラス IF 定義一覧

Return type	Method (args…, exceptions…)	Description

ResultSetMetaData	***getResultSetMetaData()***	クエリ結果のメタデータを取得
bool	***next()***	次のクエリ結果が存在するかを返却
void	***close()***	クエリ結果をクリアする
long	***getEpochTime(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータのタイムスタンプ情報をエポックミリ秒で取得する
long	***getEpochTime(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータのタイムスタンプ情報をエポックミリ秒で取得する
bool	***isNull(string key)*** ・**Args** key：カラム文字列	クエリ結果の key 文字列に該当するデータが NULL かどうかを取得する
bool	***isNull(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号	クエリ結果のカラムインデックス番号に該当するデータが NULL かどうかを取得する
string	***getString(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを string 型で取得
string	***getString(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを string 型で取得
int	***getInt(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを int 型で取得
int	***getInt(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号	クエリ結果のカラムインデックス番号に該当するデータを int 型で取得

	・Exceptions CastException：型変換エラー	
double	***getDouble(string key)*** **・Args** key：カラム文字列 **・Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを double 型で取得
double	***getDouble(int columnIndex)*** **・Args** columnIndex：カラムインデックス番号 **・Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを double 型で取得
long	***getLong(string key)*** **・Args** key：カラム文字列 **・Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを long 型で取得
long	***getLong(int columnIndex)*** **・Args** columnIndex：カラムインデックス番号 **・Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを long 型で取得
bool	***getBool(string key)*** **・Args** key：カラム文字列 **・Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを bool 型で取得
bool	***getBool(int columnIndex)*** **・Args** columnIndex：カラムインデックス番号 **・Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを bool 型で取得
Point	***getPoint(string key)*** **・Args** key：カラム文字列 **・Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを Point 型で取得

Point	***getPoint(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを Point 型で取得
LineString	***getLineString(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを LineString 型で取得
LineString	***getLineString(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを LineString 型で取得
Polygon	***getPolygon(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを Polygon 型で取得
Polygon	***getPolygon(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを Polygon 型で取得
vector<string>	***getVectorString(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを string 型配列で取得
vector<string>	***getVectorString(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを string 型配列で取得
vector<int>	***getVectorInt(string key)*** ・**Args** key：カラム文字列	クエリ結果の key 文字列に該当するデータを int 型配列で取得

	・**Exceptions** CastException：型変換エラー	
vector<int>	***getVectorInt(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを int 型配列で取得
vector<double>	***getVectorDouble(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを double 型配列で取得
vector<double>	***getVectorDouble(int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを double 型配列で取得
vector<long>	***getVectorLong(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを long 型配列で取得
vector<long>	***getVectorLong (int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを long 型配列で取得
vector<bool>	***getVectorBool(string key)*** ・**Args** key：カラム文字列 ・**Exceptions** CastException：型変換エラー	クエリ結果の key 文字列に該当するデータを bool 型配列で取得
vector<bool>	***getVectorBool (int columnIndex)*** ・**Args** columnIndex：カラムインデックス番号 ・**Exceptions** CastException：型変換エラー	クエリ結果のカラムインデックス番号に該当するデータを bool 型配列で取得

3.2.4. ResultSetMetaData クラス

表 6 ResultSetMetaData クラス IF 定義一覧

Return type	Method (args…, exceptions…)	Description
string	***getColumnName(int col)*** ・**Args** col：カラムインデックス番号	指定したカラムインデックス番号のカラム名を取得
string	***getColumnType(int col)*** ・**Args** col：カラムインデックス番号	指定したカラムインデックス番号のカラムの型を文字列で取得
int	***getColumnSize()***	クエリ結果のカラム数を返却
unsigned int	***getManagementId()***	クエリ管理番号を取得

3.2.5. DatagramSocket クラス

表 7 DatagramSocket クラス IF 定義一覧

Return type	Method (args…, exceptions…)	Description
bool	***sendStreamData(string streamName, vector<Tuple> tuples)*** ・**Args** streamName：ストリームデータのテーブル名 tuples　　：ストリームデータの集合 ・**Exceptions** ConnectionFailedException:送信失敗	DB システムに対して TCP 接続する。ip、 port の引数は省略可能で、省略した場合は初期設定(ローカルホスト+規定ポート) にて接続を行う。

3.2.6. Tuple クラス

表 8 Tuple クラス IF 定義一覧

Return type	Method (args…, exceptions…)	Description
bool	***setValue(int index, any value, long timestamp)*** ・**Args**	タプルにデータをセットする。

	index：データをセットするインデックス番号 value：セットしたいデータ timestamp：現在の EPOCH ミリ秒	timestamp はタイムゾーン依存しない設計とし、標準時間の EPOCH ミリ秒を採用する。 （プローブデータ送出間隔が 100ms なのを考慮し、秒ではなく、ミリ秒を採用） また timestamp 引数は省略可能で、その場合は DM ライブラリ内で現在時間を付与する。

3.2.7. DmUtil クラス

表 9 DmUtil クラス IF 定義一覧

Return type	Method (args…, exceptions…)	Description
long	***getTimeMillisec()***	標準時間の EPOCH ミリ秒を取得する
string	***getTimeMillisecStr()***	標準時間の EPOCH ミリ秒を文字列で取得する
Point	***getPointFromEWKT(string input)*** **・Args** key：カラム文字列 **・Exceptions** CastException：型変換エラー	input 文字列(EWKT)に該当するデータを Point 型で取得
LineString	***getLineStringFromEWKT(string input)*** **・Args** key：カラム文字列 **・Exceptions** CastException：型変換エラー	input 文字列(EWKT)に該当するデータを LineString 型で取得
Polygon	***getPolygonFromEWKT(string input)*** **・Args** key：カラム文字列 **・Exceptions** CastException：型変換エラー	input 文字列(EWKT)に該当するデータを Polygon 型で取得

4. DMLib 利用方法

本章では実際に前章で説明した DMLib の API を利用した場合の実装例を記載する。ただし本章で記載したコードを実行するには、ダイナミックマップ 2.0 コンソーシアムが開発した DM2.0PF が必要である。そのため利用者が限られてしまうが、DMLib の利用イメージを示すために本章に記載する。

4.1. 想定条件

本項では DMLib を利用するための想定条件について記載する。DB システムにはユーザ認証機能が存在するが、以下の実装例ではユーザ名”test_user”,パスワード”test_user”が事前に設定されていることを想定する。また DB システムの接続先 IP とポートに関しては、localhost とデフォルトポートである場合は省略可能であるが、必要に応じて API で指定することも可能である。

4.2. ワンショットクエリを扱う記述例

以下に DB システムへのコネクション接続からクエリの発行、結果取得までの実装例を記載する。

実装例

```
#include <is/DmManager.h>
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
// 暗号化を有効にする場合はinitEncryptionSettings()を事前にコールする
// DmManager::initEncryptionSettings(certFile, keyFile, pemPass);
// サーバに接続（ip,portは省略可、省略した場合はlocalhostのデフォルトportに接続する）
Connection* con = DmManager::getDBConnection(ip, port, “test_user”, “test_user”);
try {
    //サーバにデータを送信
    ResultSet rs = con->execute(“クエリ文字列”);
    ResultSetMetaData rsmd = rs.getResultSetMetaData();
    int colNum = rsmd.getColumnSize();
    int rowsNum = 0;
    while (rs.next()) {
        int i = 0;
        //受信したデータを表示
        while (i < colNum) {
            if (i != 0) cout << ",";
            cout << rs.getString(i++);
        }
        cout << endl;
        rowsNum++;
    }
    cout << "----- RowsNum:" << rowsNum << " -----" << endl;
    rs.close();
} catch (SQLException e) {
        cout << "ERROR catch :" << e.getMessage() << endl;
}
con->disconnect();
delete con;
```

4.3.継続クエリを扱う記述例

以下に継続クエリの登録と終了を実施するための実装例を記載する。尚、継続クエリの API は用途に応じて 3 種類の使い方が存在する。

① C 言語仕様の API

② C++言語仕様の関数ポインタを利用した API

③ オブジェクト指向な API

それぞれ以下で解説する。

実装例（①C 言語仕様の API）

```
#include <is/DmManager.h>
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
void callbackFunc(ResultSet rs)
{
        cout << "callback call" << endl;
}
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
// 暗号化を有効にする場合はinitEncryptionSettings()を事前にコールする
// DmManager::initEncryptionSettings(certFile, keyFile, pemPass);
// サーバに接続（ip,portは省略可、省略した場合はlocalhostのデフォルトportに接続する）
Connection* con = DmManager::getDBConnection(ip, port, “test_user”,”test_user”);
unsigned int mngId = 0;
try {
        // サーバに継続クエリを登録
        mngId = con->registerQuery(“クエリ文字列”, (FUNC_CALLBACK)callbackFunc);
    } catch(SQLException e) {
            cout << "ERROR catch :" << e.getMessage() << endl;
    }
    con->disconnect();
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
// 以下、継続クエリの通知が不要になったら実行
// サーバに再接続
con->reconnect();
// 継続クエリの終了
con->cancelQuery(mngId);
// TCP切断
con->disconnect();
// 接続オブジェクトの破棄
delete con;
```

C言語の仕様を用いているため、幅広い開発者に馴染みがあるのがメリットであるが、コールバック関数が static である必要性があるのがデメリットとなる。

実装例（②C++言語仕様の関数ポインタも利用した API）

```
#include <is/DmManager.h>
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
class CallBackClass {
public:
  void callbackClassFunc(ResultSet rs) {
            ResultSetMetaData rsmd = rs.getResultSetMetaData();
            int colNum = rsmd.getColumnSize();
            int rowsNum = 0;
            while (rs.next()) {
                        int i = 0;
                        //受信したデータを表示
                        while (i < colNum) {
                                    if (i != 0) cout << ",";
                                    cout << rs.getString(i++);
                        }
                        cout << endl;
                        rowsNum++;
            }
            cout << "----- RowsNum:" << rowsNum << " -----" << endl;
  }
};
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
// サーバに接続（ip,portは省略可、省略した場合はlocalhostのデフォルトportに接続する）
Connection* con = DmManager::getDBConnection(ip, port, “test_user”, “test_user”);
unsigned int mngId = 0;
try {
       // サーバに継続クエリを登録
       CallBackClass cbc;
       mngId = con->registerQuery(“クエリ文字列”, std::bind(&CallBackClass::callbackClassFunc, std::ref(cbc), std::placeholders::_1));
    } catch(SQLException e) {
           cout << "ERROR catch :" << e.getMessage() << endl;
    }
    con->disconnect();
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
// 以下、継続クエリの通知が不要になったら実行
// サーバに再接続
con->reconnect();
// 継続クエリの終了
con->cancelQuery(mngId);
// TCP切断
con->disconnect();
// 接続オブジェクトの破棄
delete con;
```

コールバック関数をクラスで定義し、registerQuery()の引数にて関数ポインタを指定できるため、自由にクラスの関数を登録出来、static な関数でなくてもよい点がメリットであるが、registerQuery()の引数の記述が複雑化する点がデメリットとなる。

実装例（③オブジェクト指向な API）

```
#include <is/DmManager.h>
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
class CallBackListenerClass : public CallBackListener {
public:
  // インターフェースクラスと同様の関数名で定義する
  void callBack(ResultSet rs) {
            ResultSetMetaData rsmd = rs.getResultSetMetaData();
            int colNum = rsmd.getColumnSize();
            int rowsNum = 0;
            while (rs.next()) {
                        int i = 0;
                        //受信したデータを表示
                        while (i < colNum) {
                                    if (i != 0) cout << ",";
                                    cout << rs.getString(i++);
                        }
                        cout << endl;
                        rowsNum++;
            }
            cout << "----- RowsNum:" << rowsNum << " -----" << endl;
  }
};
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
// サーバに接続（ip,portは省略可、省略した場合はlocalhostのデフォルトportに接続する）
Connection* con = DmManager::getDBConnection(ip, port, “test_user”, “test_user”);
unsigned int mngId = 0;
try {
        // サーバに継続クエリを登録
        CallBackListenerClass* cblc = new CallBackListenerClass();
        mngId = con->registerQuery(“クエリ文字列”, cblc);
     } catch(SQLException e) {
            cout << "ERROR catch :" << e.getMessage() << endl;
     }
     con->disconnect();
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
// 以下、継続クエリの通知が不要になったら実行
// サーバに再接続
con->reconnect();
// 継続クエリの終了
con->cancelQuery(mngId);
// TCP切断
con->disconnect();
// 接続オブジェクトの破棄
delete con;
```

インターフェースクラスを用いる事で registerQuery()の引数の記述を簡素化できるのがメリットであるが、1 クラスにつき 1 コールバック関数しか実装できない点がデメリットとなる。

4.4.ストリームデータ送信を行う記述例

以下にストリームデータを送信するための実装例を記載する。

尚、ストリームデータを送信する場合は DB システムのスキーマ設定にて定義されているストリームデータを送信する必要があるが、実装例では以下のストリーム定義がされている前提で例を示す。

表 10 ストリーム定義例「test_stream」

id(int)	long_data(long)	double_data(double)	str1(string)	str2(string)
1	123	12.3	test1	test2
…	…	…	…	…
…	…	…	…	…

実装例

```
#include <is/DmManager.h>
#include <string>
using namespace std;
～～～～～～～～～～～～(途中省略)～～～～～～～～～～～～
// 暗号化を有効にする場合はinitEncryptionSettings()を事前にコールする
// DmManager::initEncryptionSettings(certFile, keyFile, pemPass);
// Tupleの生成
Tuple tuple(5);
long now = DmUtil::getTimeMillisec();
tuple.setValue(0, (int)1, now);
tuple.setValue(1, (long)123, now);
tuple.setValue(2, (double)12.3, now);
tuple.setValue(3, (string)“test1”, now);
tuple.setValue(4, (string)“test2”, now);
vector<Tuple> tuples;
tuples.push_back(tuple);
// UDP送信オブジェクトの取得（ip,portは省略可、省略した場合はlocalhostのデフォルトportに接続する）
DatagramSocket* sendSock = DmManager::getDatagramSocket(ip, port, “test_user”,”test_user”);
// DBシステムにストリームデータを送信する
sendSock->sendStreamData("test_stream", tuples);
// 送信を終えたらソケットをcloseし、オブジェクトを破棄
sendSock->closeSocket();
delete sendSock;
```

5. 終わりに

成果物を公開した目的は、ダイナミックマップを使ったアプリ開発のためのプログラミング API 仕様について、幅広い領域からフィードバックを得ることにある。本文書に関するご意見・質問は下記の窓口へご送信願いたい。

連絡先：
ダイナミックマップ 2.0 コンソーシアム事務局
〒464-8601 名古屋市千種区不老町 NIC 508
dm2-sec@nces.i.nagoya-u.ac.jp
https://www.nces.i.nagoya-u.ac.jp/dm2/index.html

謝辞

本書はダイナミックマップ 2.0 コンソーシアムの 2016 年度～2019 年度の成果物である。また、本研究の一部は、文部科学省「革新的イノベーション創出プログラム(COI STREAM)」の助成を受けている。

6. 参考文献

- 「クエリ言語仕様書」,ダイナミックマップ 2.0 コンソーシアム
  https://www.nces.i.nagoya-u.ac.jp/dm2/query_20170906.pdf
- 「ダイナミックマップ 2.0 プラットフォーム通信アーキテクチャ(ACCEAN)の研究」,ダイナミックマップ 2.0 コンソーシアム
  https://www.nces.i.nagoya-u.ac.jp/dm2/ACCEAN20180806.pdf