ORACLE

# JavaScript関連新機能

## Oracle Database 23c新機能セミナー

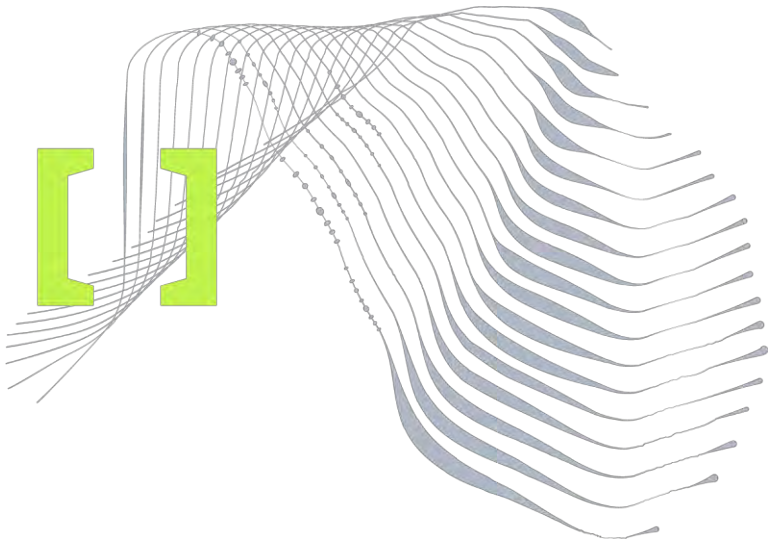
**藤田 慎二郎**

日本オラクル株式会社

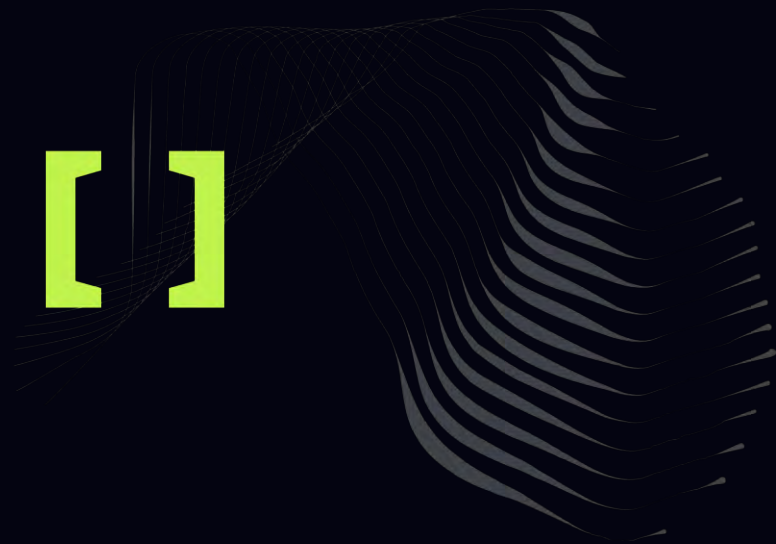
2023年10月20日

# Agenda

1. マルチリンガル・エンジンJavaScriptモジュールおよび環境
2. マルチリンガル・エンジン・モジュール・コール
3. マルチリンガル・エンジンの実行後デバッグ
4. マルチリンガル・エンジンJavaScript SODA API
5. マルチリンガル・エンジンJavaScriptでのJSONデータ型のサポート



# マルチリンガル・エンジンJavaScript モジュールおよび環境



# Oracle Database 21cで追加されたJavaScript機能

- 21cからDBMS_MLEパッケージが追加され、PL/SQLコード内でJavaScriptを実行できるようになりました。
- この機能を動的マルチリンガル・エンジン(MLE)実行と呼びます。

```
DECLARE
    l_ctx     DBMS_MLE.CONTEXT_HANDLE_T;
    l_snippet CLOB;
BEGIN
    l_ctx := DBMS_MLE.CREATE_CONTEXT();
    l_snippet := q'~
console.log(`The use of the q-quote operator`);
console.log(`greatly simplifies provision of code inline`);
~';
    DBMS_MLE.EVAL(l_ctx, 'JAVASCRIPT', l_snippet);
    DBMS_MLE.DROP_CONTEXT(l_ctx);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_MLE.DROP_CONTEXT(l_ctx);
        RAISE;
END;
/
```

qクォート(q')の中に直接JavaScriptのコードを記述

DBMS_MLE.EVALプロシージャでJavaScriptを実行

23cではPL/SQLの実装とJavaScriptの実装を分離できるように機能拡張

# マルチリンガル・エンジンJavaScriptモジュールおよび環境

- 概要
  - マルチリンガル・エンジン (MLE) モジュールと環境により、JavaScriptコードをDatabase内にスキーマ・オブジェクトとして保存できます。
  - MLEのJavaScriptによる実装はECMAScript 2022に準拠しています。
  - MLEのJavaScript モジュールはECMAScript 6 モジュールと等しく、モジュールの作成およびexport/importといった機能を提供します。
  - MLE環境はJavaScript実行時のオプション(js.strict)の設定や、作成済みMLEモジュールを他のMLEモジュールへimportする機能を提供します。

- メリット
  - 開発者はクライアント・サイドのJavaScript開発と同様の開発ワークフローを利用できます。
  - 複雑なプロジェクトを、各開発メンバーが個別に作業しやすいよう、より小さな単位に分解できます。分解されたそれぞれの処理を、MLEモジュールと環境により各々の開発者が個別に実装を進められます。

# マルチリンガル・エンジンJavaScriptモジュールおよび環境

- 基本構文

MLEモジュール

```
CREATE [ OR REPLACE ] MLE MODULE [IF NOT EXISTS][schema .] module_name
LANGUAGE [schema .] mle_language  [ VERSION version_string ]
( USING  BFILE ( directory_object_name , server_file_name )
   | ( CLOB | BLOB | BFILE ) selection_clause
   |  AS module_text )
```

MLE環境

```
CREATE [ OR REPLACE ] MLE ENV [IF NOT EXISTS][schema .] name
   (  [ CLONE [schema .] environment_name  ]
    |
    ( [ IMPORTS ( ( 'import_name' MODULE [schema .] mle_module_name)[,]... ) ]
    [ LANGUAGE OPTIONS option_string ] ) )
```

# マルチリンガル・エンジンJavaScriptモジュールおよび環境

- 実装例
  - 入力された値の階乗を計算するJavaScriptコードをMLEモジュールで実装した例

```
CREATE MLE MODULE IF NOT EXISTS factorial_mod
LANGUAGE JAVASCRIPT AS

export function factorial(num) {
  if ( num < 0 ) {
    return -1;
  } else if ( num == 0 ) {
    return 1;
  } else {
    return ( num * factorial( num - 1 ) );
  }
}
/
```

MLEモジュール名

JavaScriptコード

- MLE環境によるMLEモジュール(factorial_mod)のimportの実装例

MLE環境

```
CREATE MLE ENV my_factorial_env
IMPORTS('FACTORIAL_MOD' MODULE FACTORIAL_MOD)
/
```

MLE環境名

import時に指定する別名

import対象のMLEモジュール名

import先のMLE Module

```
CREATE OR REPLACE MLE MODULE mod_imp_fact
LANGUAGE JAVASCRIPT AS

import * as h from "FACTORIAL_MOD"

export function printAnswer(num){
    let ans = h.factorial(num);
    console.log(`The factorial of ${num} is:
${ans}`);
}
/
```

# 参考：JavaScriptをMLEで利用するために必要な権限

- JavaScriptコードの実行権限

```
GRANT EXECUTE ON JAVASCRIPT TO <role | user>
```

- 動的MLEの実行権限

```
GRANT EXECUTE DYNAMIC MLE TO <role | user>
```

- MLEスキーマ・オブジェクトを作成するための権限

```
GRANT CREATE MLE TO <role | user>
```

- MLEモジュール・コール(PL/SQLプロシージャ)を作成するための権限

```
GRANT CREATE PROCEDURE TO <role | user>
```

# 参考：MLEモジュール、MLE環境に関連したディクショナリ・ビュー

- USER_SOURCE
  - 作成済みMLEモジュールのソースコードを表示

```
SELECT LINE, TEXT FROM USER_SOURCE WHERE NAME = 'SHOW_SOURCE_MOD’;

 LINE TEXT
------- ------------------------------------------------------------
      1 export function hello(name) {
      2   console.log(`Hello, ${name}`)
      3 }
```

# 参考：MLEモジュール、MLE環境に関連したディクショナリ・ビュー

- USER_MLE_ENVS
  - 利用可能なMLE環境の情報を表示

```
SELECT ENV_NAME, LANGUAGE_OPTIONS
FROM USER_MLE_ENVS
WHERE ENV_NAME='MYENVOPT’
/

ENV_OWNER             ENV_NAME   LANGUAGE_OPTIONS
--------------------- ---------- ----------------------
JSDEV01               MYENVOPT   js.strict=true
```



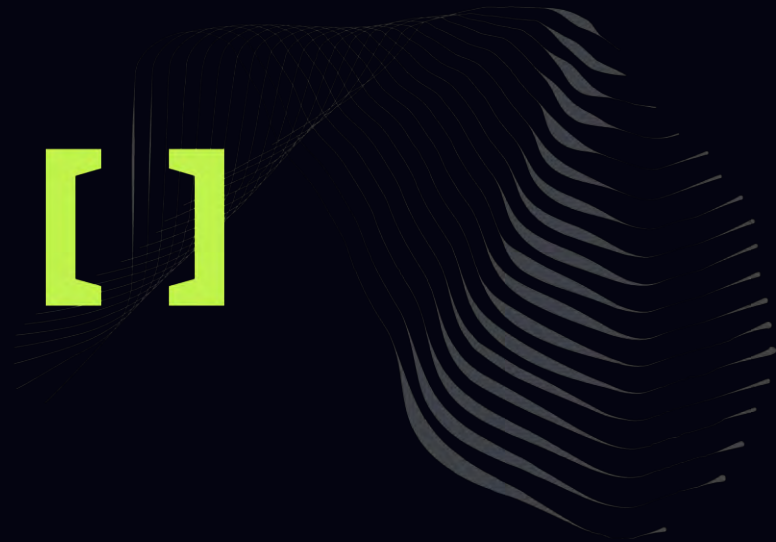

# 参考：MLEモジュール、MLE環境に関連したディクショナリ・ビュー

- USER_MLE_ENV_IMPORTS
  - 各MLE環境に設定されているimport可能なMLEモジュールの情報を表示

```
SELECT IMPORT_NAME, MODULE_OWNER, MODULE_NAME
    FROM USER_MLE_ENV_IMPORTS
    WHERE ENV_NAME='MYFACTORIALENV’;
/

IMPORT_NAME            MODULE_OWNER              MODULE_NAME
---------------------- ------------------------- ------------------
FACTORIAL_MOD          DEVELOPER1                FACTORIAL_MOD
```

# マルチリンガル・エンジン・モジュール・コール



# マルチリンガル・エンジン・モジュール・コール

- 概要
  - マルチリンガル・エンジン (MLE) モジュール・コールによりMLEモジュールに保存されたJavaScriptファンクションを、SQLやPL/SQLから実行できます。
  - MLEモジュール・コールを通してSQLやPL/SQLから実行するには、MLEモジュールにおいて対象のJavaScriptファンクションをexportします。

- メリット
  - MLEモジュール・コールを介することで、SQLやPL/SQLの処理においてJavaScriptで実装した処理を柔軟に組み込めます。

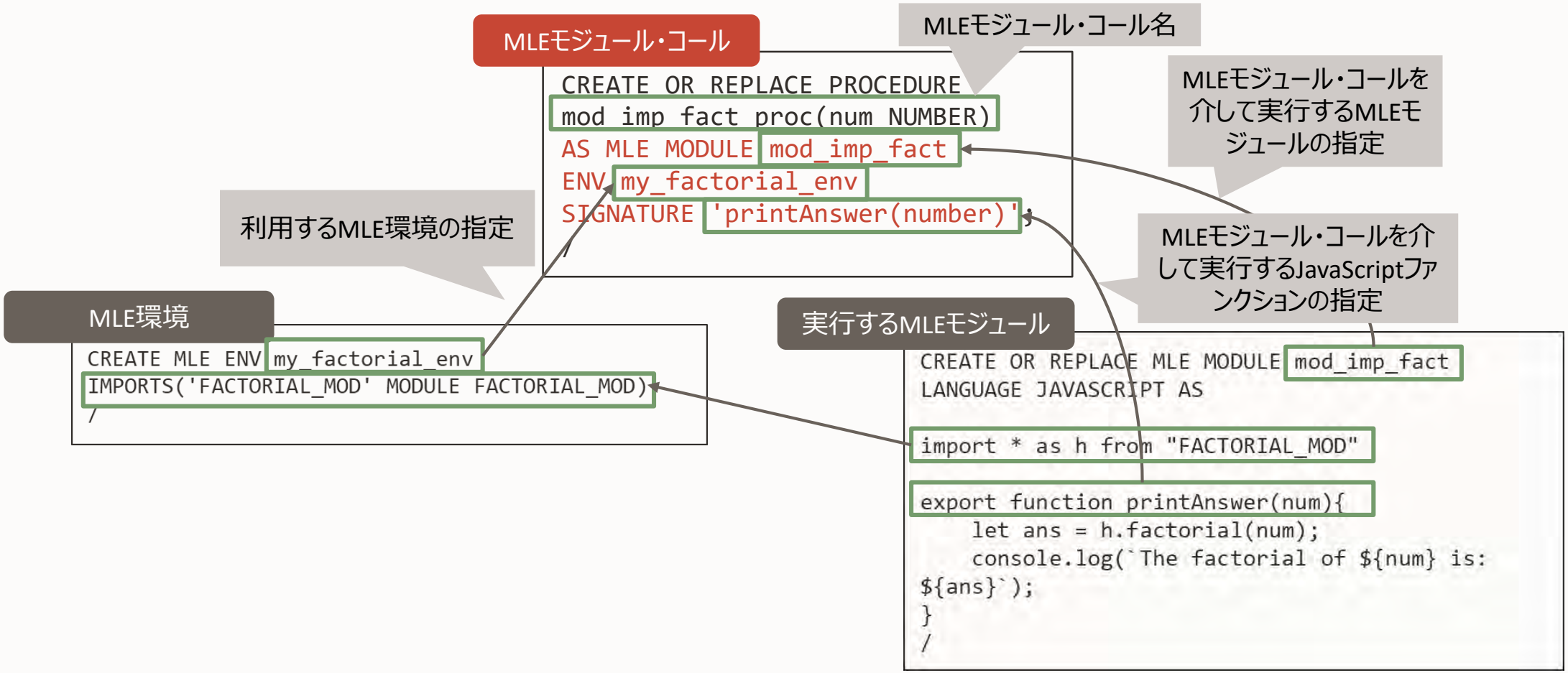
# マルチリンガル・エンジン・モジュール・コール

- 基本構文

```
CREATE [ OR REPLACE ] { FUNCTION | PROCEDURE } [ IF NOT EXISTS ]
  [ schema. ] call_spec_name [ ( param_declaration [, param_declaration]... ) ]
  [ RETURN datatype ]
  [ { invoker_rights clause
    | deterministic_clause
    | parallel_enable_clause
    | result_cache_clause }...
  ]
  { AS | IS } MLE MODULE [ schema. ] module_name
  [ ENV [ env_schema. ] mle_env ]
  SIGNATURE ' function_name_in_module
    [ ( mle_param_declaration [, mle_param_declaration]... ) ] ' ;
```

# マルチリンガル・エンジン・モジュール・コール

- 前項で作成した関数 printAnswer を実行するためのMLEモジュール・コール(PL/SQLプロシージャ)を作成する例



# 参考：作成したMLEモジュール・コールの実行

- 前スライドで作成したMLEモジュール・コール (PL/SQLプロシージャ)のmod_imp_fact_procを実行する例

```
SQL> set serveroutput on
SQL> EXEC mod_imp_fact_proc(3);
The factorial of 3 is: 6

PL/SQLプロシージャが正常に完了しました。
```




# 参考：インラインJavaScriptのファンクションおよびプロシージャ

MLEモジュールを作成せずに、インラインJavaScriptとしてファンクションやプロシージャに直接JavaScriptのコードを記述することも出来ます。

基本構文：

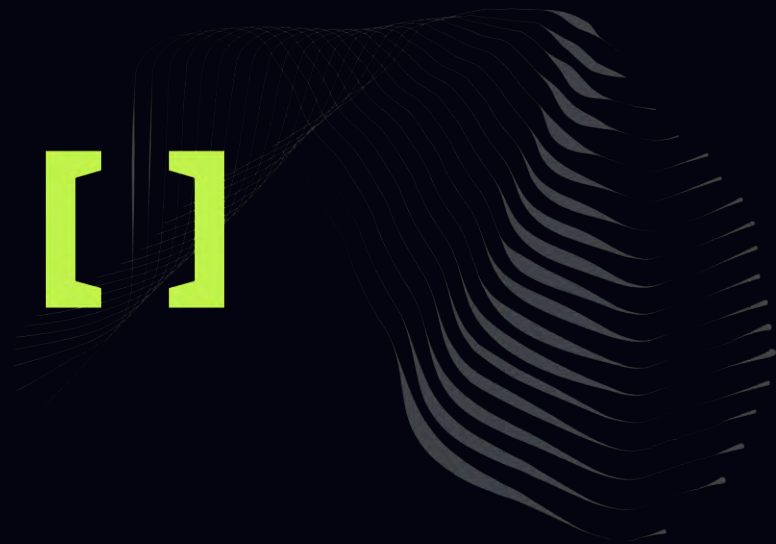
```
CREATE [ OR REPLACE ] { FUNCTION | PROCEDURE } [ IF NOT EXISTS ]
  [ schema. ] call_spec_name [ ( param_declaration [, param_declaration]... ) ]
  [ RETURN datatype ]
  [ { invoker_rights clause
    | deterministic_clause
    | result_cache_clause }...
  ]
  { AS | IS } MLE LANGUAGE language_name js_function_body_as_string_literal;
```

実装例：

```
CREATE FUNCTION IF NOT EXISTS sum_inline("num1" NUMBER, "num2" NUMBER) return NUMBER
AS MLE LANGUAGE JAVASCRIPT
q'~
    return num1+num2;
~';
/
```

qクォート(q')の中に直接JavaScriptのコードを記述

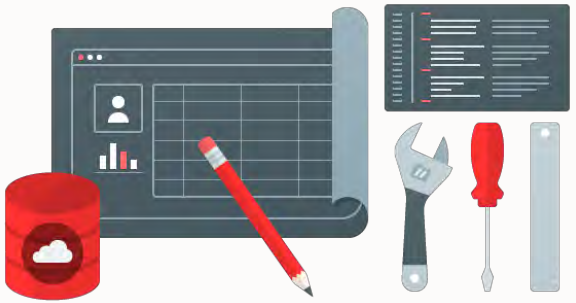
# マルチリンガル・エンジンの実行後デバッグ



# マルチリンガル・エンジンの実行後デバッグ

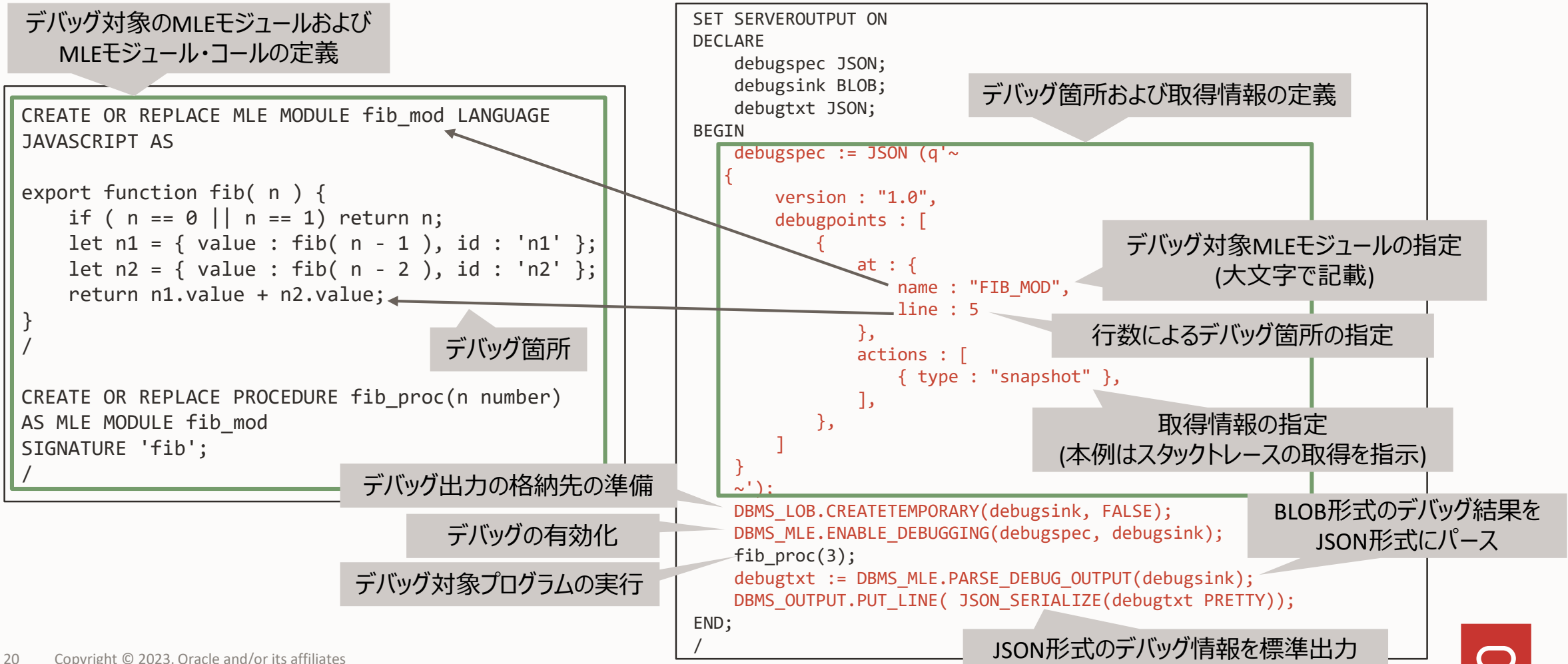
- 概要
  - マルチリンガル・エンジン(MLE)は、プログラム実行時の状態を収集することにより、開発者がJavaScriptコードをデバッグする機能を提供します。これはPost-Execution Debuggingと呼ばれる機能です。
  - コードが実行された後、収集されたデータはプログラムの動作を分析し、バグを発見して修正するために使用できます。
  - Post-Execution DebuggingによりJavaScriptコード実行時の状態を収集する際、コードを変更する必要はありません。

- メリット
  - コードを変更することなく実行時の情報を収集し、プログラムの各ステップの状態を確認できるため、開発者は効率的にデバッグ作業を行えます。



# マルチリンガル・エンジンの実行後デバッグ

- MLEモジュールとして実装したJavaScriptコードの指定行について、スタックトレースを取得する例



# マルチリンガル・エンジンの実行後デバッグ

- MLEモジュール…iptコードの指定行について、スタックトレースを取得する例

デバッグ対象のMLE
MLEモジュール・

```
CREATE OR REPLAC
JAVASCRIPT AS

export function
    if ( n == 0
    let n1 = { v
    let n2 = { v
    return n1.va
}
/

CREATE OR REPLAC
AS MLE MODULE fi
SIGNATURE 'fib';
/
```

出力結果

```
[
    [
        {
            at : {
                name : "fib",
                line : 5
            }
            values : {
                n : 2,
                n1 : {
                    value : 1,
                    id : 'n1'
                },
                n2 : {
                    value : 0,
                    id : 'n2'
                }
            }
        },
        {
            at : {
                name : "fib",
                line : 3
            },
            values : {
                n : 3
            }
        }
    ],
    [
        {
            at : {
                name : "fib",
                line : 5
            },
            values : {
                n : 3,
                n1 : {
                    value : 1,
                    id : 'n1'
                },
                n2 : {
                    value : 1,
                    id : 'n2'
                }
            }
        }
    ]
]
```

```
GUAGE

: 'n1' };
: 'n2' };

ッグ箇所

umber)

の格納先の準備

ッグの有効

プログラムの実行
```

```
SET SERVEROUTPUT ON
DECLARE
    debugspec JSON;
    debugsink BLOB;
    debugtxt JSON;
BEGIN
    debugspec := JSON (q'~
    {
        version : "1.0",
        debugpoints : [
            {
                at : {
                    name : "FIB_MOD",
                    line : 5
                },
                actions : [
                    { type : "snapshot" },
                ],
            },
        ]
    }
    ~');
    DBMS_LOB.CREATETEMPORARY(debugsink, FALSE);
    DBMS_MLE.ENABLE_DEBUGGING(debugspec, debugsink);
    fib_proc(3);
    debugtxt := DBMS_MLE.PARSE_DEBUG_OUTPUT(debugsink);
    DBMS_OUTPUT.PUT_LINE( JSON_SERIALIZE(debugtxt PRETTY));
END;
/
```

デバッグ箇所および取得情報の定義

デバッグ対象MLEモジュールの指定
(大文字で記載)

行数によるデバッグ箇所の指定

取得情報の指定
(本例はスタックトレースの取得を指示)

BLOB形式のデバッグ結果を
JSON形式にパース

JSON形式のデバッグ情報を標準出力

# 参考：デバッグ設定(Debugpoint)のJSONフォーマットの種類

特定変数の状態確認

```
actions: [
  { type: "watch",
    id: <string[]> | <string>,
    [depth : <number>] }
]
```

- **id**: 状態確認したい変数名
- **depth**: 対象変数の中身がJavaScriptオブジェクトなどで入れ子構造になっている場合、どこまで深くログとして出力するかを制御

スタックトレースの取得

```
actions: [
  { type: "snapshot",
    [framesLimit: <number>],
    [depth : <number>] }
]
```

- **framesLimit**: スタックトレースとしてどこまでログ出力するかを制御(ex: a()→b()→c()という呼び出し順の場合、framesLimitが1であればc()の情報のみ出力)。デフォルトは制限なし
- **depth**: 変数の中身がJavaScriptオブジェクトなどで入れ子構造になっている場合、どこまで深くログとして出力するかを制御

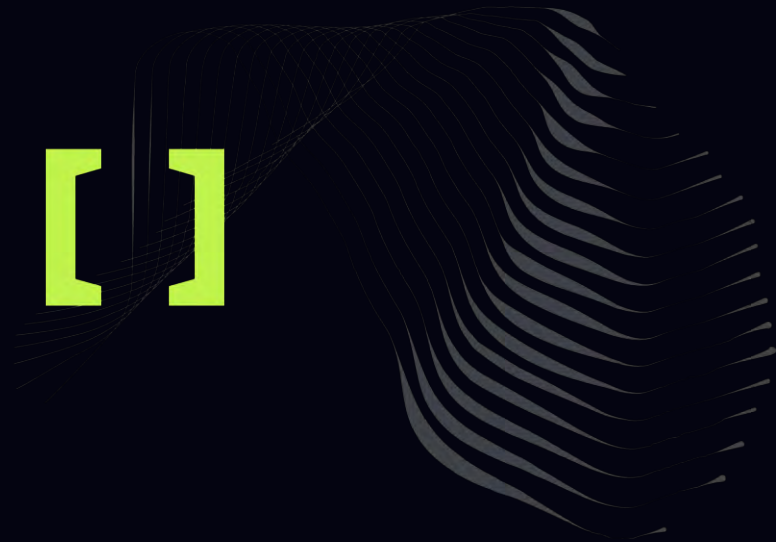
# 参考：マルチリンガル・エンジンの実行後デバッグの利用に必要な権限

- MLEモジュールを定義したDBユーザ以外がデバッグする場合、そのデバッグするDBユーザに以下の権限を付与する必要があります。

```
GRANT COLLECT DEBUG INFO ON <module> TO <role | user>
```



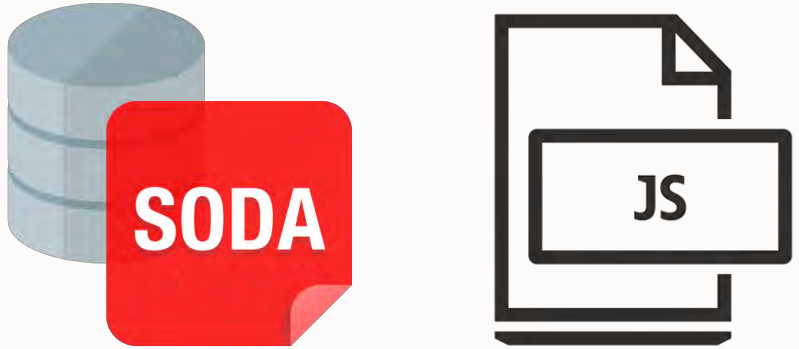

# マルチリンガル・エンジンJavaScript SODA API



# マルチリンガル・エンジンJavaScript SODA API

- 概要
  - Simple Oracle Document Access (SODA)はOracle Databaseにドキュメント(特にJSON)のコレクションを作成および格納し、それらを取得し、クエリできるようにするNoSQLスタイルのAPIのセットです。
  - マルチリンガル・エンジン (MLE) JavaScriptの導入により、SODAドキュメントのJavaScriptサポートがクライアントサイドおよびサーバーサイドの開発で利用できるようになりました。

- メリット
  - JavaScriptでSODA APIをサポートすることで、開発者はJSONをリレーショナルまたはNoSQLで使用する方法を選択できるようになり、開発プロセスが簡素化され、コードの移植性が向上します。



# マルチリンガル・エンジンJavaScript SODA API

- ①SODAコレクションの新規作成、②ドキュメントの新規挿入、③コレクション内のドキュメント表示を実装した例

```
CREATE OR REPLACE MLE MODULE intro_soda_mod LANGUAGE JAVASCRIPT AS

// Oracle Database接続で使うSQL Driverのimport
import oracledb from "mle-js-oracledb";

export function testSODA() {
    // Oracle Databaseに接続
    const connection = oracledb.defaultConnection();
    // SODAオブジェクト作成
    const db = connection.getSodaDatabase();
    // SODAコレクションを新規作成
    const col = db.createCollection("MyJSONCollection");

    // 新規追加するJSONドキュメントを作成
    const doc = {
        "employee_id": 100,
        "job_id": "AD_PRES",
        "last_name": "King",
        "first_name": "Steven",
        "email": "SKING",
        "manager_id": null,
        "department_id": 90
        };

    // ドキュメントをコレクションに挿入
    col.insertOne(doc);

    // 繰返し処理によってコレクション内の各ドキュメントを取得するためにカーソルを使用
    const c = col.find().getCursor();
    let resultDoc;
    while (resultDoc = c.getNext()) {
        const content = resultDoc.getContent();
        console.log(`
            ----------------------------------------
            key:            ${resultDoc.key}
            content (select fields):
            - employee_id   ${content.employee_id}
            - job_id        ${content.job_id}
            - name          ${content.first_name} ${content.last_name}
            version:        ${resultDoc.version}
            last modified:  ${resultDoc.lastModified}
            created on:     ${resultDoc.createdOn}
            media type:     ${resultDoc.mediaType}`
        );
    }
    // SODAドキュメントのカーソルをクローズ
    c.close();
    // トランザクションをcommit
    connection.commit();
}
/
```

# 参考：MLE JavaScript SODA API利用時のMLEモジュール・コール作成の注意点

MLEモジュール・コール作成時に AUTHID CURRENT_USER 句が必要です。
定義者権限で作成した場合はSODA APIの実行においてエラーになるため、
実行者権限で実行できるようMLEモジュール・コールを作成します。

```
CREATE OR REPLACE PROCEDURE intro_soda_proc()
AUTHID CURRENT_USER AS MLE MODULE intro_soda_mod
SIGNATURE 'testSODA';
/
```

# 参考：マルチリンガル・エンジンJavaScript SODA APIの利用に必要な権限

- MLE JavaScriptに必要な権限に加え、以下のロールを付与する必要があります。

```
GRANT SODA_APP TO <role | user>
```




# 参考：MLEモジュールのJavaScriptからOracle Databaseに接続するSQL Driver

- MLE JavaScriptに組み込まれている mle-js-oracledb モジュールを以下のようにimportし、Oracle Databaseへの接続や問合せを実行します。

```
CREATE OR REPLACE MLE MODULE employee_mod LANGUAGE JAVASCRIPT AS

// mle-js-oracledb をimport
import oracledb from "mle-js-oracledb";

export function getEmployee(employee_id) {
    // defaultConnectionメソッドを使い、コネクションを取得
    connection = oracledb.defaultConnection();

    // executeメソッドの引数にSQLを指定して問合せを実行
    // SQL中の変数(:id)にセットする値と、結果のフォーマットを後続で指定
    const result = connection.execute(`
    SELECT employee_id, first_name, last_name, hire_date
    FROM hr.employees
    WHERE employee_id = :id`,
        [
            employee_id
        ],
        {
            outFormat: oracledb.OUT_FORMAT_OBJECT
        }
    );
```

```
    // 結果セット result 内の要素を1つずつ処理
    for (let row of result.rows) {
        console.log(`employee_id: ${row.EMPLOYEE_ID}
        first_name: ${row.FIRST_NAME}
        last_name: ${row.LAST_NAME}
        hire_date: ${row.HIRE_DATE}`);
    }
}
/
```

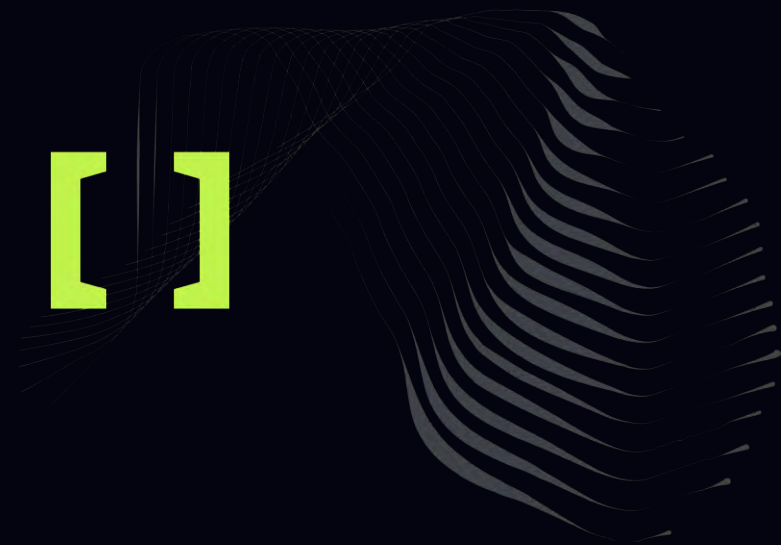
# 参考：MLE JavaScriptに組み込まれているモジュール

- MLE SQL Driver: mle-js-oracledb
- MLE Bindings: mle-js-bindings
- MLE PL/SQL Types: mle-js-plsqltypes
- MLE Fetch API polyfill: mle-js-fetch
- MLE Base64 Encoding: mle-encode-base64

各モジュールの詳細は以下ドキュメントをご参照ください。

Server-Side JavaScript API Documentation

# マルチリンガル・エンジンJavaScriptでのJSONデータ型のサポート



# マルチリンガル・エンジンJavaScriptでのJSONデータ型のサポート

- 概要
  - マルチリンガル・エンジン (MLE) JavaScriptはJSONデータ型をサポートします。
  - MLEモジュールに加え、動的MLEでもサポートします。
  - SQLやPL/SQLの処理過程で得られたJSONデータ型のデータをMLE JavaScriptに連携した際、MLE JavaScriptではそのままJavaScriptオブジェクトとして扱えます。
  - また逆にMLE JavaScriptからJavaScriptオブジェクトを連携した際は、SQLやPL/SQLではそのままJSONデータ型として扱えます。

- メリット
  - SQLやPL/SQLとMLE JavaScriptとの間でJSONデータ型のデータをやり取りするような処理において、パース処理を考慮する必要がなくなり、コードを簡素化できます。

# マルチリンガル・エンジンJavaScriptでのJSONデータ型のサポート

- PL/SQLから連携されたJSONデータ型のデータに対して、MLEモジュール内でデータを追加し、再度もとのPL/SQLに更新されたデータを連携する処理の実装例

```
CREATE OR REPLACE MLE MODULE
plsql_mle_json_mod LANGUAGE JAVASCRIPT AS

function plsqlMLEJson(jsonStr) {
    jsonStr.data.push({"department_id": 1050,
"department_name": "New Department 1050",
"manager_id": 205, "location_id": 2600});
    return jsonStr;
}
export {plsqlMLEJson}
/
```

呼び出しもとのPL/SQLで生成されたJSONデータに追加

JavaScriptファンクションを実行

```
CREATE OR REPLACE FUNCTION
plsql_mle_json_func(json_str JSON)
RETURN JSON
AS MLE MODULE plsql_mle_json_mod
SIGNATURE 'plsqlMLEJson';
/
```

MLEモジュール・コールを実行

```
set serveroutput on
DECLARE
    l_json        JSON;
    l_json_obj    json_object_t := json_object_t();
    l_json_array  json_array_t := json_array_t();
BEGIN
    l_json_array.append(json_object_t('{ "department_id": 1010,
"department_name": "New Department 1010", "manager_id": 200, "location_id":
1700 }'));
    l_json_obj.put('data', l_json_array);

    l_json := l_json_obj.to_json;
    dbms_output.put_line('json before: ' || json_serialize(l_json pretty));
    l_json := plsql_mle_json_func(l_json);
    dbms_output.put_line('-------------');
    dbms_output.put_line('json after: ' || json_serialize(l_json pretty));
END;
/
```

JSONデータ型のデータを生成

# マルチリンガル・エンジンJavaScriptでのJSONデータ型のサポート

- PL/SQLから連携されたJSONデータ型のデータに対して、MLEモジュール内でデータを追加し、再度もとのPL/SQLに更新されたデータを連携する処理の実装例

出力結果

```
json before: {
    "data" :
    [
        {
            "department_id" : 1010,
            "department_name" : "New Department 1010",
            "location_id" : 1700,
            "manager_id" : 200
        }
    ]
}
-------------
json after: {
    "data" :
    [
        {
            "department_id" : 1010,
            "department_name" : "New Department 1010",
            "location_id" : 1700,
            "manager_id" : 200
        },
        {
            "department_id" : 1050,
            "department_name" : "New Department 1050",
            "manager_id" : 205,
            "location_id" : 2600
        }
    ]
}
```

```
SIGNATURE 'plsqlMLEJson';
/
```

コールを実行

```
set serveroutput on
DECLARE
    l_json        JSON;
    l_json_obj    json_object_t := json_object_t();
    l_json_array  json_array_t := json_array_t();
BEGIN
    l_json_array.append(json_object_t('{ "department_id": 1010,
"department_name": "New Department 1010", "manager_id": 200, "location_id":
1700 }'));
    l_json_obj.put('data', l_json_array);

    l_json := l_json_obj.to_json;
    dbms_output.put_line('json before: ' || json_serialize(l_json pretty));
    l_json := plsql_mle_json_func(l_json);
    dbms_output.put_line('-------------');
    dbms_output.put_line('json after: ' || json_serialize(l_json pretty));
END;
/
```

JSONデータ型のデータを生成

# 参考：SQLおよびPL/SQLとMLE JavaScript間のデータ型マッピング

マッピングの一覧については以下マニュアルをご参照ください。

- MLE Type Conversions

# 参考：MLE JavaScript各新機能について動作確認した技術記事

- Multilingual Engine JavaScript Modules and Environments & Multilingual Engine Module Calls
  Oracle Database 23cで追加されたMLE JavaScriptの基本的な使い方紹介
- Multilingual Engine Post-Execution Debugging
  Oracle Database 23cで追加されたMLE JavaScript Post-Execution Debuggingを試してみた
- Multilingual Engine JavaScript SODA API
  Oracle Database 23cで追加されたMLE JavaScript SODA APIを試してみた
- Multilingual Engine JavaScript Support for JSON Data Type
  Oracle Database 23cで追加されたMLE JavaScriptにおけるJSONデータ型を扱う機能を試してみた

# ありがとうございました