WebAssembly on the PostgreSQL

言語拡張機能でWASMを動かしてみた

日本情報通信株式会社
齋藤晃

概要

WebAssemblyが最初にアナウンスされてから7年以上が経過し、現在ではDockerやブロックチェーン、PaaSのサーバレス環境等、WEBブラウザ外の様々な技術に組み込まれながら発展を続けています。

PostgreSQLは優れた拡張性を備えており、その一つにUDF用の言語拡張機能が存在します。WebAssemblyを組み込むことも比較的容易に実現できました。

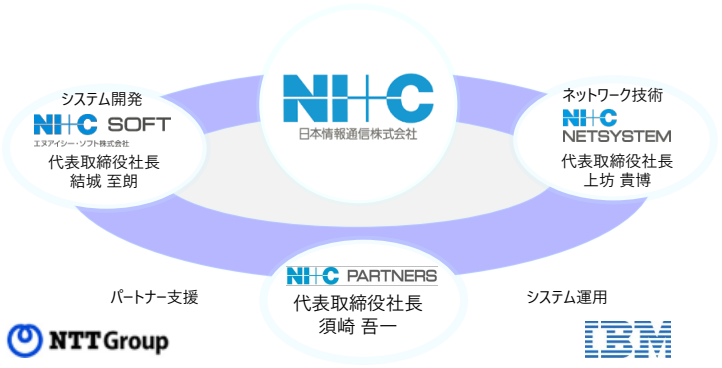
どのような実装を行うことで、それを実現したのか。
そして、その結果どのような効果が得られたのか。
言語拡張機能の解説を交えながら説明します。

自己紹介

- 齋藤晃
  - 出身：青森県平川市
  - 所属：日本情報通信株式会社
  - github：https://github.com/asiwjp

所属会社の紹介

設立	1985年12月18日
資本金	40億円
株主	日本電信電話株式会社(**65%**) 日本アイ・ビー・エム株式会社(**35%**)
売上高	439億円(FY22連結ベース)
社員数	1,286名(グループ会社計:FY23 4現在)
代表 取締役	代表取締役社長　　桜井　伝治 代表取締役副社長　須崎　吾一
事業内容	・システムインテグレーションサービス ・ハードウェア機器、ソフトウェア製品の販売
お取引先	約2,400社

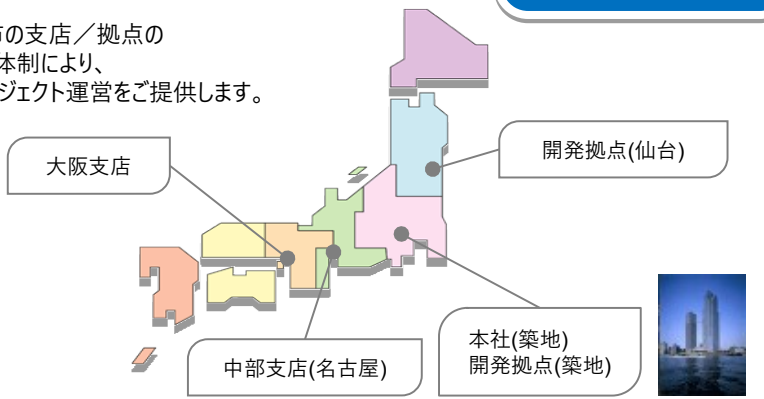


主な会社資格

- 国際規格 ISO/IEC27001
  情報セキュリティマネジメントシステム
- 国際規格 ISO14001
  環境マネジメントシステム
  ▷本社・大阪支店
  ▷エヌアイシー・ソフト株式会社/本社
  ▷エヌアイシー・ネットシステム株式会社/本社
  ▷エヌアイシー・パートナーズ株式会社/本社
- 国際規格 ISO9001
  品質マネジメントシステム
  ▷システムサービス部門
- JNX 認定サービス・プロバイダー
- NTTコミュニケーションズ・ビジネスパートナー
- IBMプレミア・ビジネス・パートナー
- IIJグローバルソリューションズ パートナー
- NTTコムウェア Premium Excellent Partner
- JPNIC正規会員
  インターネット・サービス・プロバイダー
  IPアドレス管理指定業者
- VMware ソリューション
  プロバイダ エンタープライズ
  VMware サービス プロバイダ
  プロフェッショナル
- Redhat Enterprise Linux アドバンスト・
  ビジネス・パートナー
- Citrix Solution Advisor
  （Silver Solution Advisor）
- 届出電気通信事業者（総務省）
- 一般建設業（電気通信工事業）
- 労働者派遣事業（派13-307734）

サポート体制

全国主要都市の支店／拠点の
営業・サポート体制により、
きめ細かなプロジェクト運営をご提供します。



注意書き

- 資料中の拡張機能は、齋藤個人が検証用途で作成した物です。
- 日本情報通信株式会社が、開発・公開しているものではありませんので、ご注意ください。

WebAssemblyの概要

WebAssemblyの概要

- JavaScriptの高速化を目指したモノ。
- 仮想命令コードのバイナリ（WASM）
  - 所謂中間コード。
  - ほとんどのランタイムでJITコンパイル。C言語並みに高速（らしい）
- プラットフォーム非依存
  - 仕様としては。ランタイムの有無次第。
- プログラミング言語非依存
  - 特定の言語に依存しない。コンパイラ次第。
- W3Cで仕様策定
  - 1企業の影響を受けにくい

WebAssemblyの概要

- ランタイムの種類
  - WEBブラウザ
    - Firefox/Chrome/Edge/Safari・・・最近は標準搭載
  - 非WEBブラウザ
    - Wasmtime・・・スタンダード？
    - Wasmer・・・Wasmtimeからフォークしたらしい
    - 他いろいろ

非ブラウザ版には、共有ライブラリ(so/dll)が提供されており、PostgreSQLの組み込みなどにも利用できます。

PostgreSQLの言語拡張機能

PostgreSQLの言語拡張機能

- UDFの手続型言語(PL)を、利用者が追加できます。
- 「PL」は、「PL」と言語ハンドラーの登録で利用可能になります。

```
CREATE FUNCTION my_lang_handler()
  RETURNS language_handler
  AS 共有ライブラリ名
  LANGUAGE C;
```

言語ハンドラー関数の登録
特殊な戻り値型(language_handler)を持ったC関数を登録

```
CREATE TRUSTED PROCEDURAL LANGUAGE my_lang
   HANDLER my_lang_handler;
```

PLの登録
登録時に言語ハンドラーをマップ

```
CREATE FUNCTION my_func() LANGUAGE my_lang
～;
```

UDFの登録
登録したPLを言語として、UDFを作成

```
SELECT my_func();
```

UDFの呼出し
UDF登録時のPLに対応した言語ハンドラーが呼び出されUDFが処理される。

PostgreSQLの言語拡張機能

- WebAssebmlyをPLとして追加すると？
  - 利用者は・・・
    - 使い慣れた言語でUDFを作成できる。（コンパイラがあれば）
    - C言語を用いずに、高速なUDFを作成できる。（かもしれない）
    - そのくせ、どこでも動く。（ランタイムがあれば）

なんだか（）書きだらけですが、メリットがありそうな気もします。
勉強にはなりますし、とりあえず実装してみることにしました。

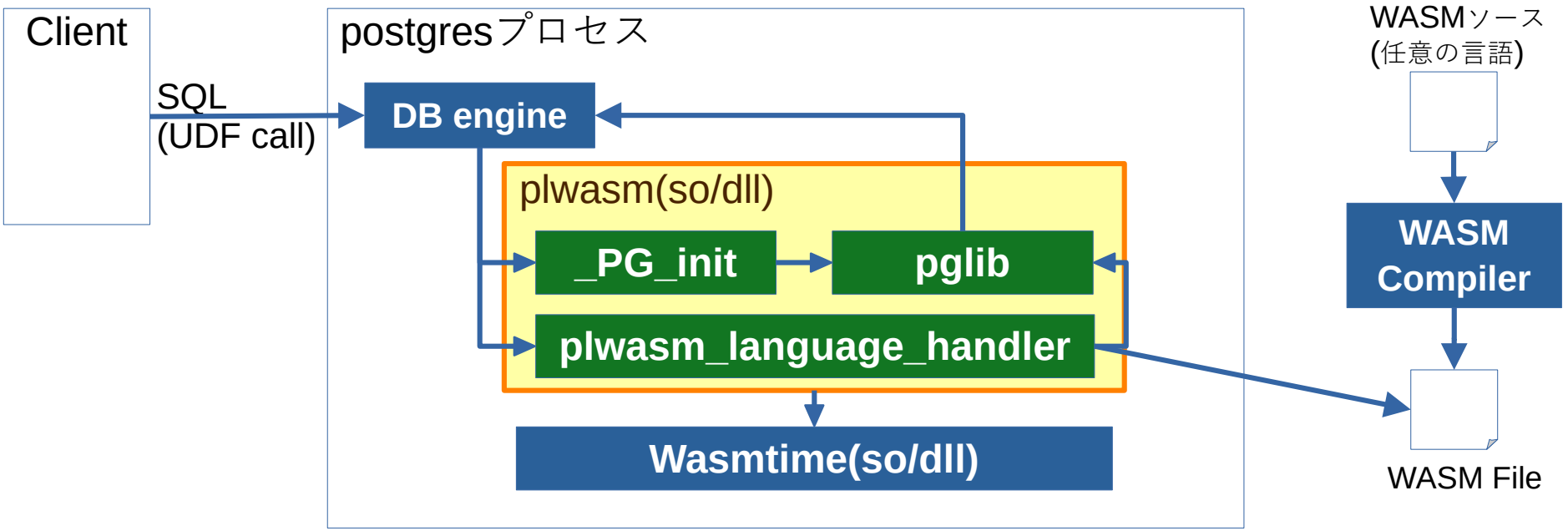
基本設計

方針

- 言語の拡張機能(EXTENSION)として提供。名前はPL/wasm。
- 実体はplwasm.so/dll。
- UDFの言語として、”plwasm”を指定可能にする。
- UDFのdefinition(*)にて、wasmファイルと関数名をJSONで指定。
- ランタイムは、とりあえずWasmtimeを採用。

* CREATE FUNCTION ～ as ‘xxx’の、xxxです。

アーキテクチャ

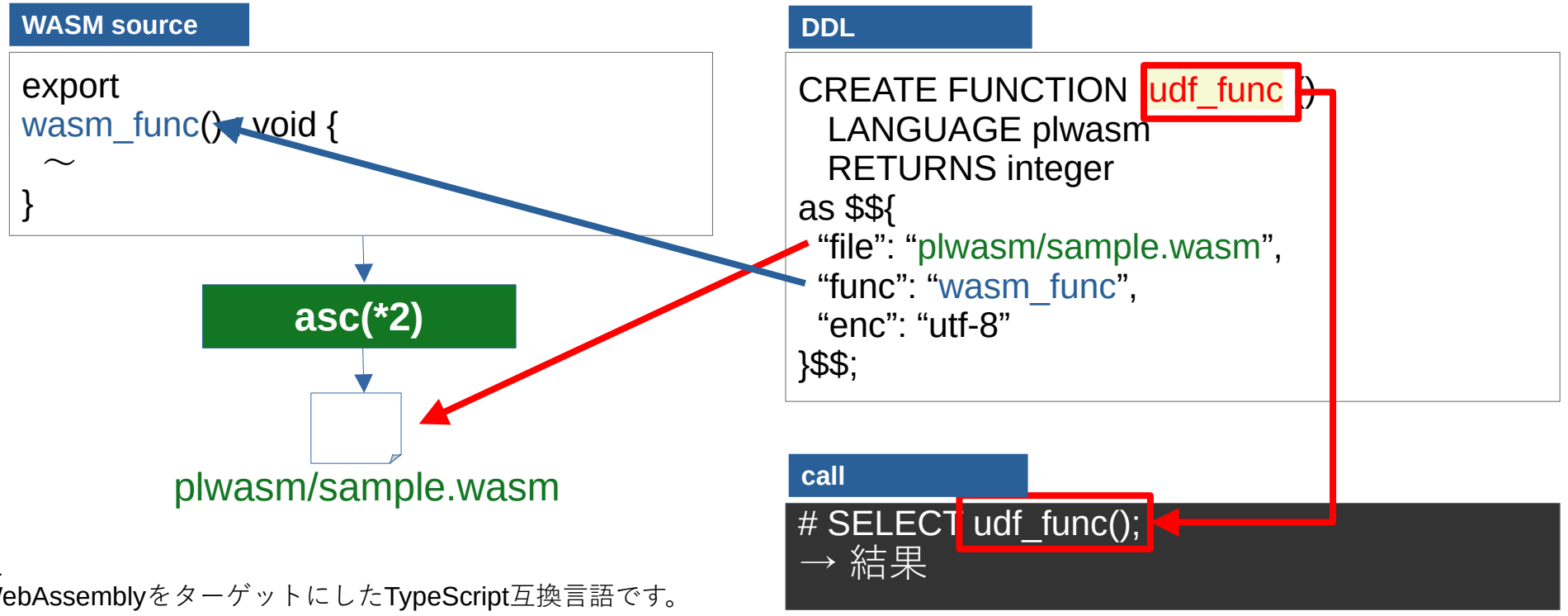


主な構成要素の概要

DB engine	・・・ SQL parserやexecutor等。簡略化のためにDB engineと記載。
_PG_init	・・・ plwasm拡張機能の初期化関数。(後述)
plwasm_language_handler	・・・ plwasmの言語ハンドラー関数。(後述)
pglib	・・・ PostgreSQLとのI/Fを提供するWASMモジュール。(後述)
WASM Compiler	・・・ ソース言語毎のWASMコンパイラー。

利用イメージ

この例では、WASMのソース言語として、AssemblyScript(*1)を使用します。



*1
WebAssemblyをターゲットにしたTypeScript互換言語です。
(https://www.assemblyscript.org)

*2
AssemblyScriptのコンパイラーです。

実装の詳細

(_PG_init)

_PG_init

_PG_initは、共有ライブラリ毎に宣言できるPostgreSQLの特別な関数です。

PostgreSQLは共有ライブラリ(so/dll)のロードを行うと、ライブラリの中から
_PG_init関数を検索して、呼び出します。

この挙動は、拡張機能の初期化処理に利用できるので、以下の処理を行います。

(1) GUC(*)の処理
(2) Wasmtimeランタイムの初期化
(3) キャッシュ用のハッシュテーブル作成
(4) pglib WASMモジュール生成

ちなみに、_PG_initと対になる終了時のエントリポイントもあったようですが、
現在では廃止されています。

*Grand Unified Configurationの略で、PostgreSQLの設定管理システムのことです。
postgresql.confやSQL、C関数等から、設定を変更したり、参照可能にする仕組みを提供します。

_PG_init - (1) GUCの処理

まずはじめにGUC項目を定義します。
これはPostgreSQLのパラメータであり、簡単に言えばpostgresql.confの項目です。

普段はoffにしたい機能（トレースや処理時間計測機能など）が幾つかあるので、
GUC項目として設定項目を定義することにしました。

```
DefineCustomBoolVariable("plwasm.trace",〜);
DefineCustomIntVariable("plwasm.trace_threshold",〜);
...
```

```
postgres=# select name,vartype From pg_settings where name like
          name           | vartype
-------------------------+---------
 plwasm.trace            | bool
 plwasm.trace_threshold | integer
(2 行)
```

GUCの関数群は、Guc.hに定義されています。
これらの関数を用いると、項目を定義する以外にも、初期値の指定や、値のValidation、
ALTER DATABASE SETコマンドにも対応できます。

_PG_init - (2) WASMランタイムの初期化

Wasmtimのランタイムでは、初期化用のC関数が提供されていますので、
_PG_initで、これらを用いた初期化を行い、結果をグローバル変数に入れておきます。

本解説では省略します。

_PG_init - (3)キャッシュ用ハッシュテーブル生成

続いてキャッシュ用のハッシュテーブル(以降ハッシュTBL)を、メモリ内に作成します。
これは生成に時間がかかるWASMモジュールや、インスタンスの格納先として使います。

メモリの使用にあたっては、PostgreSQLのMemoryContextへの理解が必要です。
PostgreSQLのAPIで確保したメモリは、基本的にいずれかのContextに属します。

主なMemoryContext

```
TopLevelContext          … ルートコンテキスト。postgresプロセスの終了まで存続。
 |- CacheMemoryContext   … 主にシステムテーブルキャッシュ。postgresプロセスの終了まで存続。
 |- TopTransactionContext … トランザクションの終了まで存続
 ...etc...
```

子のContextに属するメモリは、親のContextの終了と共に解放されてしまうので、
メモリを使用する際には、生存期間を意識したContextの選定が重要になります。

また、MemoryContextは拡張機能自らが、作成することができます。

_PG_init - (3)キャッシュ用ハッシュテーブル生成

キャッシュはトランザクションを跨いで生存させたいのと、サイズ把握(*1)もしたいので、
まずは、TopMemoryContext(*2)を親として、新しいMemoryContextを作成しました。

```
ectx->memctx = AllocSetContextCreate(TopMemoryContext,
                "PL/WebAssembly cache context", ALLOCSET_SMALL_SIZES);
```

ハッシュTBL作成には、hash_create(hsearch.h)を利用しました。
この関数にHASH_CONTEXT定数と、作成したContextを指定することで、
希望したContextに、ハッシュTBLが属することになります。

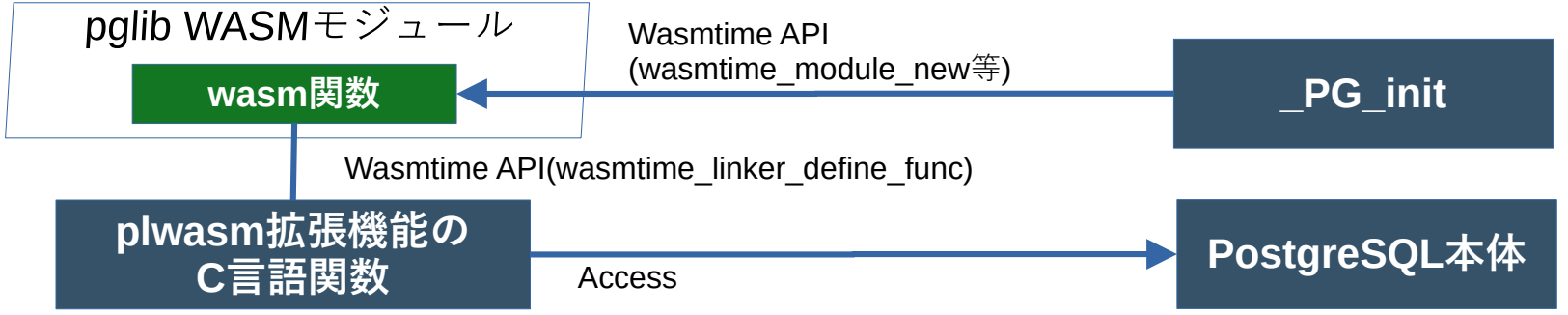
```
hsctl_modules.hcxt = ectx->memctx; // 作成したメモリコンテキストを指定
hsctl_modules.keysize = sizeof(Oid);
hsctl_modules.entrysize = sizeof(plwasm_hs_entry_cache_wasm_module_t);
ectx->wasm_module_cache = hash_create(
      "plwasm-modules", 1024, &hsctl_modules,
      HASH_ELEM | HASH_BLOBS | HASH_CONTEXT);
```

*1 pg_backend_memory_contextsシステムビューが提供されています。(PostgreSQL14以降)
*2 CacheMemoryContextでも同様の効果は得られますが、、、システムキャッシュから分離しました。

_PG_init - (4) pglib WASMモジュール生成

最後にpglib WASMモジュールの生成を行います。

このモジュールは、PostgreSQLに対するI/F(SQLコマンドを発行する等)を、
WASMの関数として提供するものです。



WasmtimeのAPIを利用することで、C関数をWASM関数として定義しています。
WASMモジュール自体を作成するのも、WasmtimeのAPIです。

SQLコマンドの発行自体は、SPI関数を利用しています。
SPI関数は、公式ドキュメントの”サーバプログラミングインターフェース”の章に
掲載されていますので、興味のある方は覗いてみてください。

実装の詳細

(言語ハンドラー：plwasm_language_handler)

言語ハンドラー

_PG_initが終了すると、メインとなる言語ハンドラーが呼び出されます。

PL/wasmの言語ハンドラーでは、次のような順で処理を行います。

(1)UDF definitionの解析
(2)WASMモジュール生成
(3)WASMインスタンス生成
(4)WASM関数コール
(5)UDF戻値の設定

尚、
・WASMモジュール　　＝　WASMをJITコンパイルした後の実行可能バイナリ
・WASMインスタンス　＝　WASMモジュールの実行インスタンス
です。

言語ハンドラー – (1)UDF definitionの解析

WASMのファイルや関数名などの情報は、UDF definitionから読み取ります。

まずはUDFのOidを入手します。
言語ハンドラーの引数である関数コール情報（以下のfcinfo）の中から取り出せます。

```
pg_proc_oid = fcinfo->flinfo->fn_oid;
```

今度は、SearchSysCache1関数に、PDOCOIDとOidを指定して
pg_procシステムテーブルのレコードを取得します。

```
pl_tuple = SearchSysCache1(PROCOID, ObjectIdGetDatum(pg_proc_oid));
```

更に、SysCacheGetAttr関数で、prosrc列の値をUDFのdefinitionとして取り出します。

```
src = SysCacheGetAttr(PROCOID, pl_tuple, Anum_pg_proc_prosrc, &isnull);
```

この流れは、PostgreSQLのソース内にある、拡張機能のサンプルそのままです。
src\test\modules\plsample\plsample.c

言語ハンドラー – (1)UDF definitionの解析

UDFのdefinitionはJSONですので、パースが必要です。

PostgreSQLのソースから、以下のJson関数を見つけました。
・jsonb_in　　・・・文字列→Jsonb型の組込変換関数。
・findJsonbValueFromContainer　　・・・Jsonbからプロパティを探す関数。

これらを駆使して、JSONのパースを行います。

jsonb_inは、SQL用に提供される組込関数ですが、
この手の関数は、DirectFunctionCall～等を用いるとC言語から呼び出せます。
※～は、関数の引数の数やCollation有無などに応じたバリエーションがあります。

```
jb_root = (Jsonb*) DirectFunctionCall1Coll(jsonb_in, DEFAULT_COLLATION_OID, (Datum)source);
```

言語ハンドラー - (2)WASMモジュールの生成

WASMファイルの情報を得たら、そのファイルからWASMモジュールを生成します。

生成にはWasmtimeのAPI（wasmtime_module_new）を使用しますが、
時間がかかるので、生成したモジュールはハッシュTBLにキャッシュします。

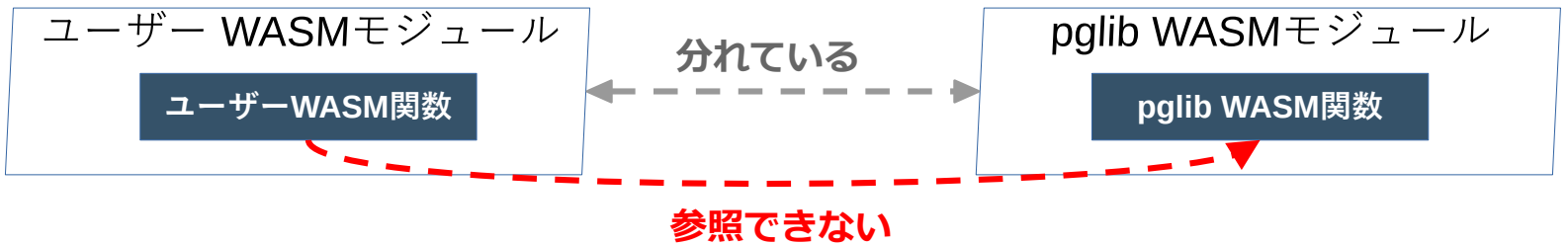
```
cache_entry = (plwasm_hs_entry_cache_wasm_module_t*) hash_search(
    cctx->ectx->wasm_module_cache, // _PG_initで作成したハッシュテーブル
    &fn_oid,
    HASH_ENTER,
    &found);
```

HASH_ENTERを渡すと、ハッシュキーのエントリーが無かった場合に(found=false)、
新しいエントリーのポインター(cache_entry)が返されますので値を設定し保存します。

言語ハンドラー - (3)WASMインスタンスの生成

今度はWASMモジュールを実行できるように、
メモリを初期化する等の様々な準備を行って、インスタンスを作成します。

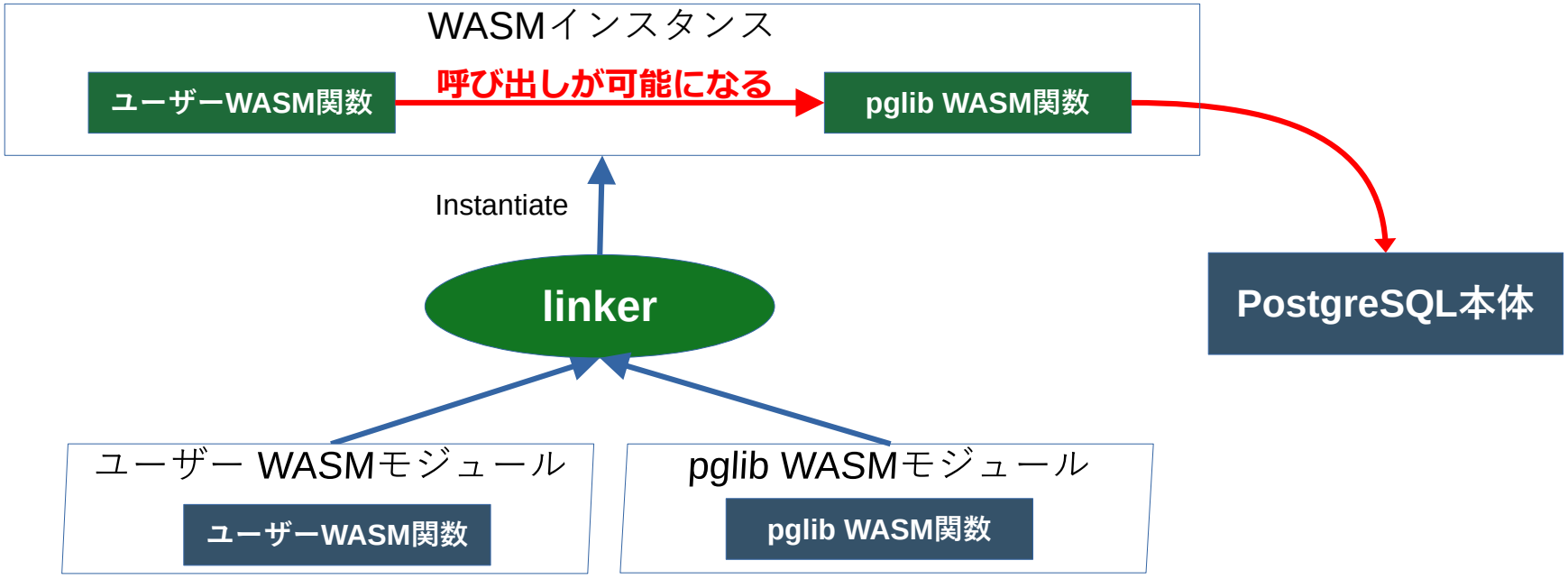
ただ、まだこの段階では、
ユーザーが用意したWASMモジュールと、pglib WASMモジュールが分れたままで、
ユーザー側からpglib側を参照できません。



SQLコマンドを実行できないWASMは、あまり役には立たなさそうです。
それどころか、参照先の解決付加を理由として、インスタンス化が失敗します。
両モジュールを結合する手段が必要です。

言語ハンドラー - (3)WASMインスタンスの生成

WasmtimeのlinkerをWASMインスタンスの生成に利用することにしました。



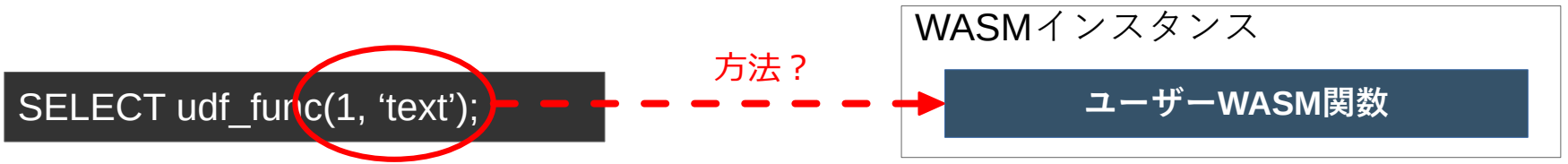
これで両者のWASM関数がインスタンス内に取り込まれ、
ユーザーのWASM関数から、SQLコマンドを発行できるようになります。

言語ハンドラー - (4)WASM関数コール

インスタンスが出来たので、いよいよその中の関数を呼出します。

呼出し自体は、WasmtimeのAPI(wasmtime_func_call)を用いれば簡単ですが、
ここでも考慮すべき点が、、、

UDFの引数を、ユーザーのWASM関数に引き渡す方法を考える必要があります。



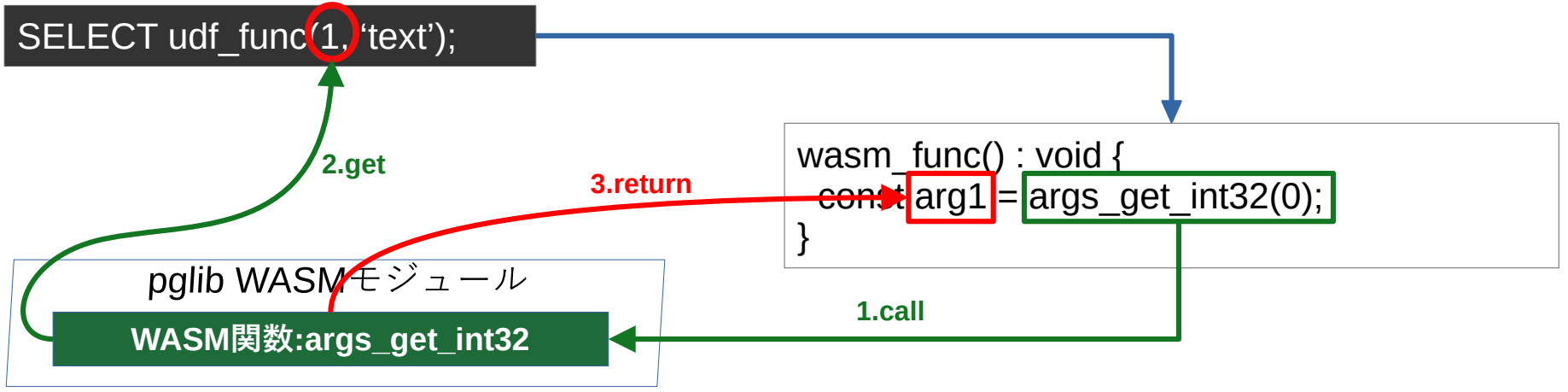
WASMの関数には、引数を持たせることができますが、
WASMのデータ型は、32 or 64bitの整数または浮動小数点数しかありません。

PostgreSQLには数値以外にも文字列等もありますし、NULLもあります。

言語ハンドラー - (4)WASM関数コール

pglibモジュールにて、UDF引数を扱うWASM関数を提供することにしました。

以下は、integer型のUDF引数を扱う例です。

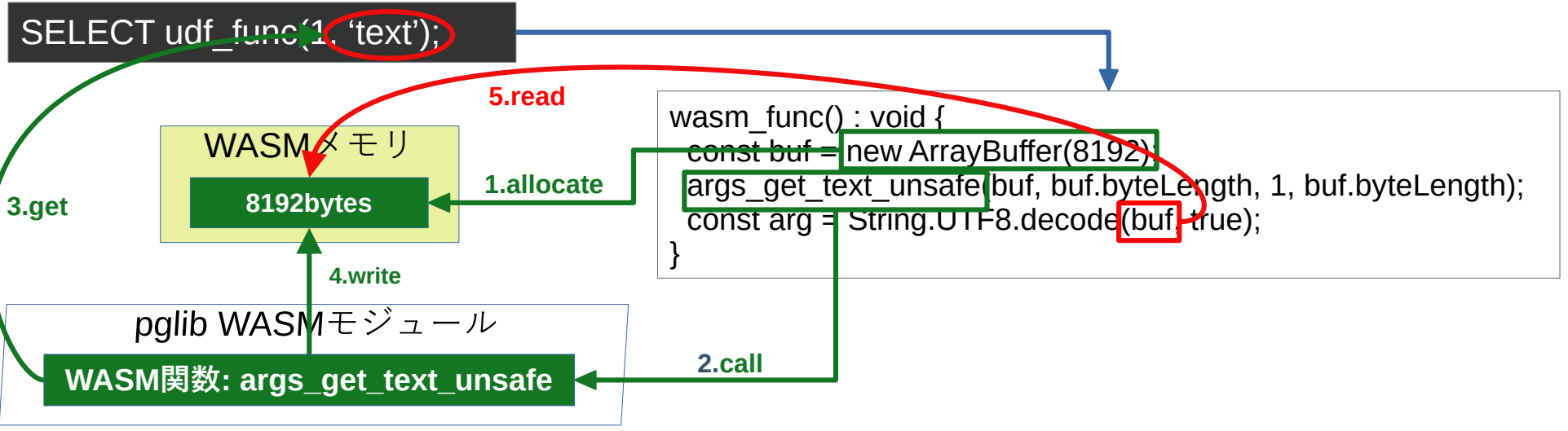


この方式なら、データ型毎の関数などを用意する手間はありますが、
多数の型をサポートしつつも、ある程度一貫性のあるI/Fを作れそうです。

言語ハンドラー - (4)WASM関数コール

text型など、64bitに収まらないデータ型に対しては、もう一工夫が必要です。
WASMインスタンス毎に所有するメモリを利用します。

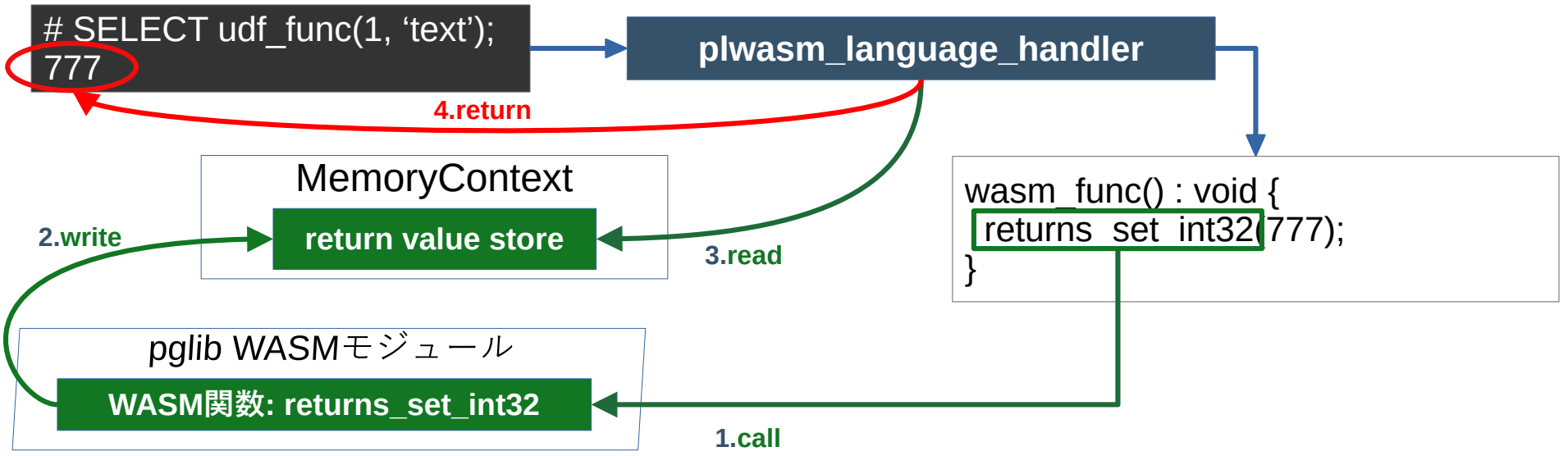
以下は、text型を扱うargs_get_text_unsafeの例です。



WASMメモリには、WasmtimeのAPIでアドレスを取得して、読み書きを行います。
文字符号化方式は、ソース言語毎に異なるので、UDFで宣言可能にしています。

言語ハンドラー - (5)UDF戻り値の設定

戻り値の返却も、UDF引数と同様の方式でpglibから関数を提供します。

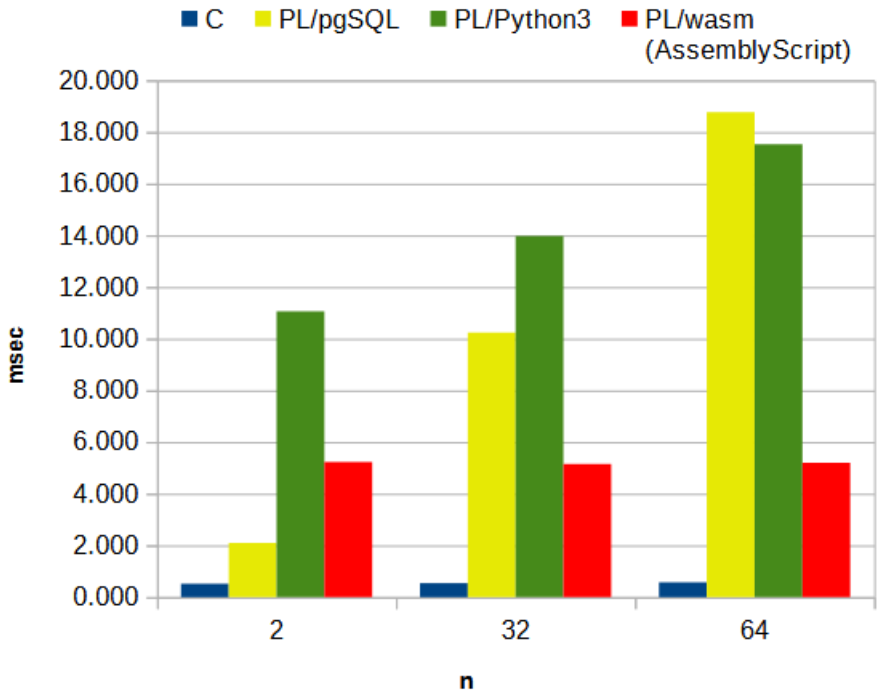


UDF引数との違いは、PostgreSQLのMemoryContextを介している点です。

パフォーマンス

パフォーマンス

フィボナッチ関数(f)による演算性能の測定(*1)。　f(n) x 1000回



・なんだかいまいち・・・？

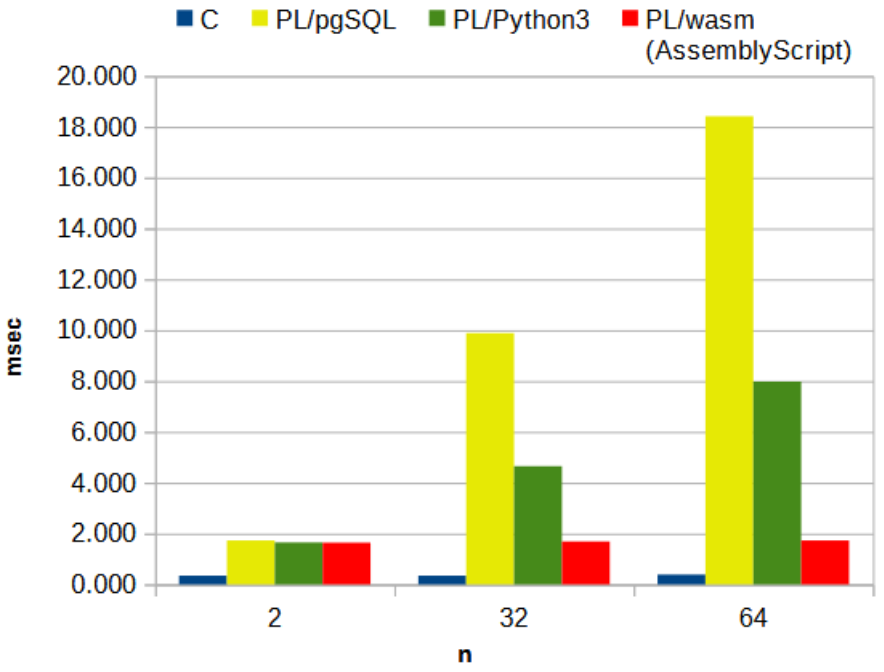
・ただし、nの拡大（計算量の増加）に対する処理時間の増加が、極端に少ないという特徴が・・・。

・結果的に、nが大きくなるにつれてPL/wasmが、他のPLを上回る。(*2)

*1 Pythonのメモ化等、言語固有の最適化は行っていません。
*2 記載していませんが、Python3単独での測定も行っています。PL/Python3と比べると、いずれのnでも倍以上高速でしたが、nの拡大に伴い処理時間が増加する傾向は同じで、途中からPL/wasmのほうが高速になりました。

パフォーマンス

ファーストショットを除いてみました。



・PLの中では、n=2でこそ大差ないが、
n > 2でわかりやすく差が現れる。

・nの拡大による時間増加がないことから、
呼出し準備のオーバーヘッドが殆どで、
純粋な関数本体の処理時間は、
Cにほぼ近いのではないかと推測。

一般的には、SQL等によるI/Oコストのほうが大きいと思うので、あくまでも参考です。

最後に感想など

実装してみて

・PLの追加は、（I/Fを考えず単純にWASMを呼ぶだけであれば）
実質１、２日で実装できてしまいましたので、思った以上に簡単でした。
本体ソースにサンプルソースがあるのは大きく、
「あれ？動いちゃった？」という感じです。

・もちろん、本資料に記載した内容をすべてとなると、数週間かかっていますが、
PostgreSQLのソースを読んだり、WasmtimeやAssemblyScriptについて調べたり、、
実装以外の時間の方が長かった様な気がします。

・PostgreSQLの関数は、綺麗にドキュメント化されているわけではないので、
内部関数を直接呼んでいるような感じでした。
一度ソースを眺めてみて、少しでも構造を知っていたほうが、実装はスムーズです。

・ソースを眺めるとか、ハードルは多少高いのですが、動いた時はちょっと感動します。
また、ちょっと知った気になれて楽しいです。
時間があれば、試してみてはいかがでしょうか。

PL/wasmの今後の予定

PL/wasmでは、今後、以下に取り込んでみようと考えています。

- 呼出しオーバーヘッドの削減による高速化
- Wasmtime以外のランタイムサポート
- PostgreSQLの様々な型のサポート
- トリガーのサポート
- MacOSのサポート
- CI環境やUnitTestの準備
- いろんなソース言語でのお試し

何だかいろいろあります。関わってみたい方は、githubまで。

EOF