# BigQueryエミュレータの作り方

Merpay Architect
goccy

2022/10/15 Go Conference mini 2022 Autumn IN SENDAI

# goccy ( ごっしー )

Merpay Architect

@goccy54

goccy

| Repository | Star |
| --- | --- |
| goccy/go-json | 1.8k |
| goccy/go-yaml | 754 |
| goccy/go-reflect | 392 |
| goccy/go-graphviz | 408 |

# Agenda

1. 用語解説 ( BigQuery と ZetaSQL )
2. エミュレータ開発のMotivation と Goal
3. goccy/bigquery-emulator
4. How it works

# Agenda

1. 用語解説 ( BigQuery と ZetaSQL )
2. エミュレータ開発のMotivation と Goal
3. goccy/bigquery-emulator
4. How it works

# BigQuery

- GCP上で動作する、ペタバイト単位のデータに対して高速に分析できる
  フルマネージドDWH
- クエリエンジンとして **Dremel** を利用している

- Dremel や Cloud Spanner は SQL として **Google Standard SQL** を
  解釈するように作られている
- Google Standard SQL の Parser / Analyzer は **ZetaSQL** という名前で OSS として
  公開されている
- 実際には Cloud Spanner は ZetaSQL + 独自拡張 で実装されており、BigQuery の
  SQL は ZetaSQL でカバーされている印象

# ZetaSQL

- https://github.com/google/zetasql
- C++製
- SQL Parser / Analyzer などを提供している
  - BigQuery のクエリをすべて Parse できる ( はず )
  - Analyzer はクエリの対象となるテーブルのスキーマやカラムの型情報を見て Validation したり、型判別した上での AST を出力できる機能
- ビルドには Bazel が必要
- **Go から利用する方法は存在しない**

# Agenda

1. 用語解説 ( BigQuery と ZetaSQL )
2. **エミュレータ開発のMotivation と Goal**
3. goccy/bigquery-emulator
4. How it works

# Motivation

- 2022/01、BigQuery を使った仕組みをいくつか実装する可能性があった
- BigQuery にはエミュレータが存在しないことを知る
- IssueTracker には 2019/03 に必要性について言及されているが、2022/01 の時点でも存在しなかった
  - 2022/10 現在も Google からは何も提供されていない
- ZetaSQL で BigQuery の SQL を Parse/Analyze できることを知る
- ZetaSQL を Go から使いたい需要がそこそこありそうなことを知る
- **Go から ZetaSQL を扱えるようにして、それを使って BigQuery のエミュレータを作ればみんな嬉しいんじゃないか**

# Goal

- BigQuery 関連の OSS 開発を助けること
  - OSS の CI で BigQuery に直接つなぎにいくのは費用面でハードルが高い
  - 「エミュレータがあるならツールを作ろうか」という動きを増やしたい
- BigQuery を利用したローカルでの開発を助けること
  - 現状では、ローカルで良い感じに開発・テストするためには何らかの仕組みを作らなければならない
    - そのコストを割きたくないという理由でテストを書かないことがあるはず
  - エミュレータを提供することで API Endpoint を差し替えればあとは何も考えなくて良いという世界にしたい
- BigQuery クライアントから見て、本物と同じ動作を目指す

# Agenda

1. 用語解説 ( BigQuery と ZetaSQL )
2. エミュレータ開発のMotivation と Goal
3. goccy/bigquery-emulator
4. How it works

# goccy/bigquery-emulator

- https://github.com/goccy/bigquery-emulator
- Go 製
- ストレージに SQLite を使っているのでデータの永続化ができる
- Docker image / Binary / Go Install でインストールできる
  - シングルプロセスで動くのでバイナリ1つ、イメージ1つで動作する
- Go のテストで使う場合は、ライブラリとして使うことで httptest を利用してインプロセスでサーバを起動できる
- エミュレータなので、既存の Client SDK や bq などの CLI が endpoint を差し替えるだけでそのまま使える

# サポートしている機能（データ型 / 文 / 句）

- **Type**
  - GEOGRAPHY 型を除く全ての型（GEOGRAPHY は使っている例をあまり見なかったので後回し）
- **Statement**
  - 基本的なものの他に MERGE や UDF（User Defined Function）にも対応
- **Clause / Expression**
  - OVER - WINDOW / WITH / JOIN / GROUP BY - ROLLUP / QUALIFY など

# サポートしている機能（ 標準関数 ）

## 63% ( 206 / 329 )

| | |
|---|---|
| Aggregate | 15/15 |
| Statistical aggregate | 9/9 |
| Approximate aggregate | 0/4 |
| HyperLogLog++ | 0/4 |
| Numbering | 3/6 |
| Bit | 0/1 |

| | |
|---|---|
| Conversion | 17/18 |
| Mathematical | 41/41 |
| Navigation | 3/7 |
| Hash | 5/5 |
| String | 50/52 |
| JSON | 7/16 |

| | |
|---|---|
| Array | 8/8 |
| Date | 12/12 |
| Datetime | 10/10 |
| Time | 9/9 |
| Timestamp | 16/16 |
| Interval | 5/5 |

| | |
|---|---|
| Geography | 0/63 |
| Security | 0/1 |
| UUID | 1/1 |
| Net | 0/10 |
| Debugging | 0/1 |
| AEAD Encryption | 0/13 |

**※ サポート数 / 標準関数の数**

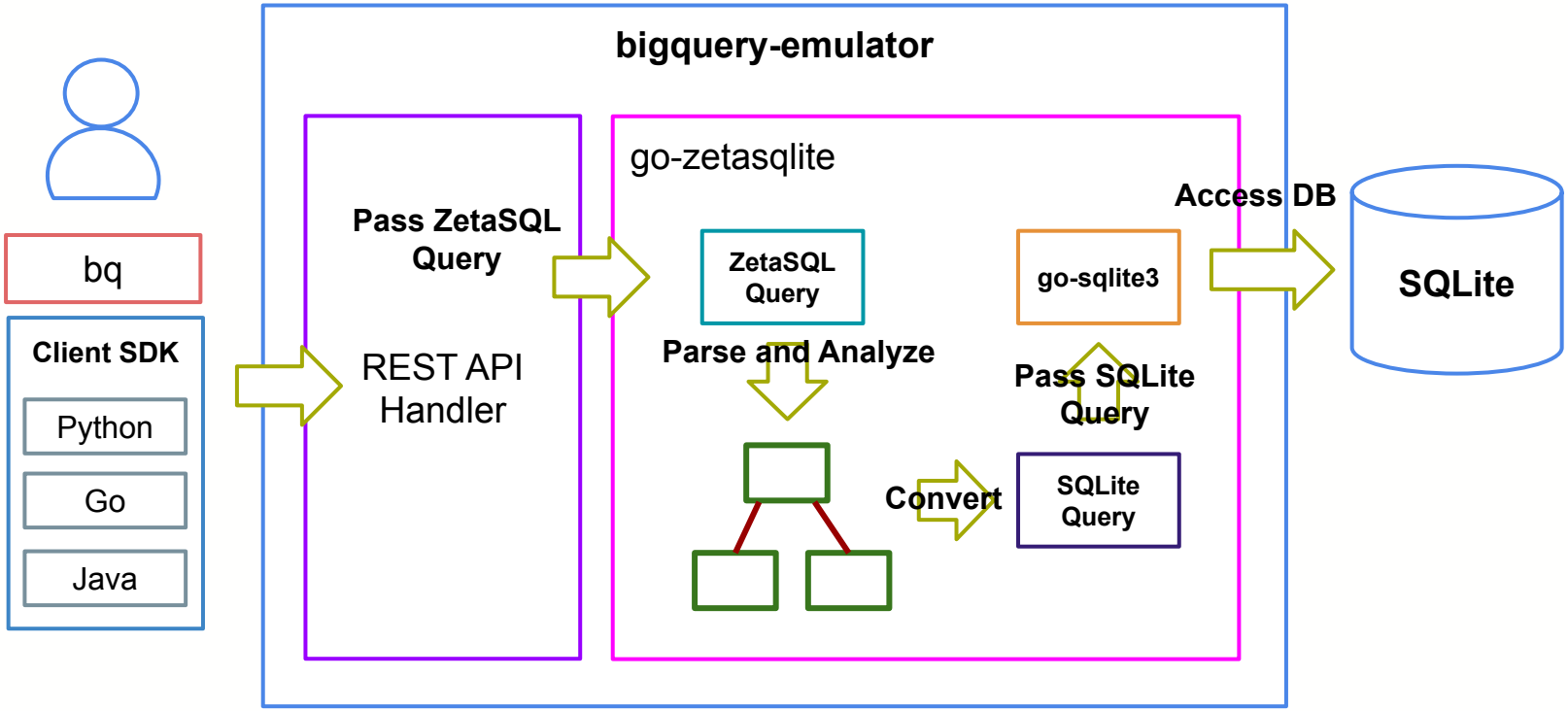
# 今後サポート予定の機能

- GEOGRAPHY 型
- 残りの標準関数
- 残りのクエリ句 e.g.) DECLARE, PIVOT
- INFORMATION SCHEMA
- BigQuery Storage API
- BigQuery Model
- JavaScript UDF ( powered by otto or goja )
- 様々な場所 (e.g. GCS ) からのデータの Import

# Agenda

1. 用語解説 ( BigQuery と ZetaSQL )
2. エミュレータ開発のMotivation と Goal
3. goccy/bigquery-emulator
4. How it works

# Architecture Overview



# BigQuery API

- 基本的には REST API 経由で JSON をやりとりする
  - BigQuery API | Google Cloud
- API Request/Response の構造は Go の Client SDK が利用している bigquery package を使っている
- クライアントの実装によって Response フィールドの何を参照するかが違う
  - 正確に仕様を満たしていないと、ものによって動いたり動かなかったりする
  - e.g.) Go の Client SDK では動いたけど bq で動かない
- ZetaSQLクエリの評価は go-zetasqlite で行い、bigquery-emulator リポジトリでは BigQuery API の仕様にあわせてクライアントとやりとりするだけ

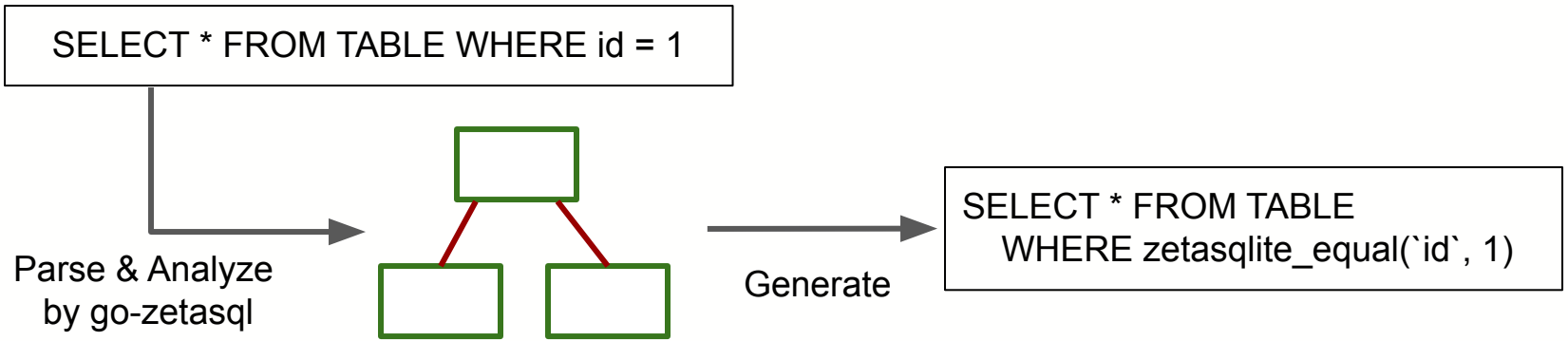
# goccy/go-zetasqlite

- https://github.com/goccy/go-zetasqlite
- ZetaSQL クエリを解釈・評価して結果を SQLite に保存できる

  Database Driver Library
  - Go の database/sql の I/F にあわせて ZetaSQL クエリを実行できる
- SQLite へのアクセスは mattn/go-sqlite3 を利用
- **ZetaSQL のクエリ評価ができるもので、BigQuery 専用ではない**
  - e.g.) project_id / dataset_id を table 名の前に付与することは必須じゃない
- ZetaSQL クエリの解析には自作の goccy/go-zetasql を利用している

# Architecture

- go-zetasql で ZetaSQL クエリを解析して AST を出力
- AST から SQLite のクエリを生成する
- Operator や Builtin Function は**すべて** 1 : 1 対応する独自関数を作成し、
  go-sqlite3 の機能で関数を登録して実現している
  - 独自関数は通常の関数の他に、集計関数や照合関数も登録できる



# ZetaSQL クエリを SQLite 向けに変換するときの注意

- SQLite に存在しない型の扱い
- ARRAY 型の値の展開
- WINDOW 関数
- ORDER BY による SQLite に存在しない型の比較と COLLATE

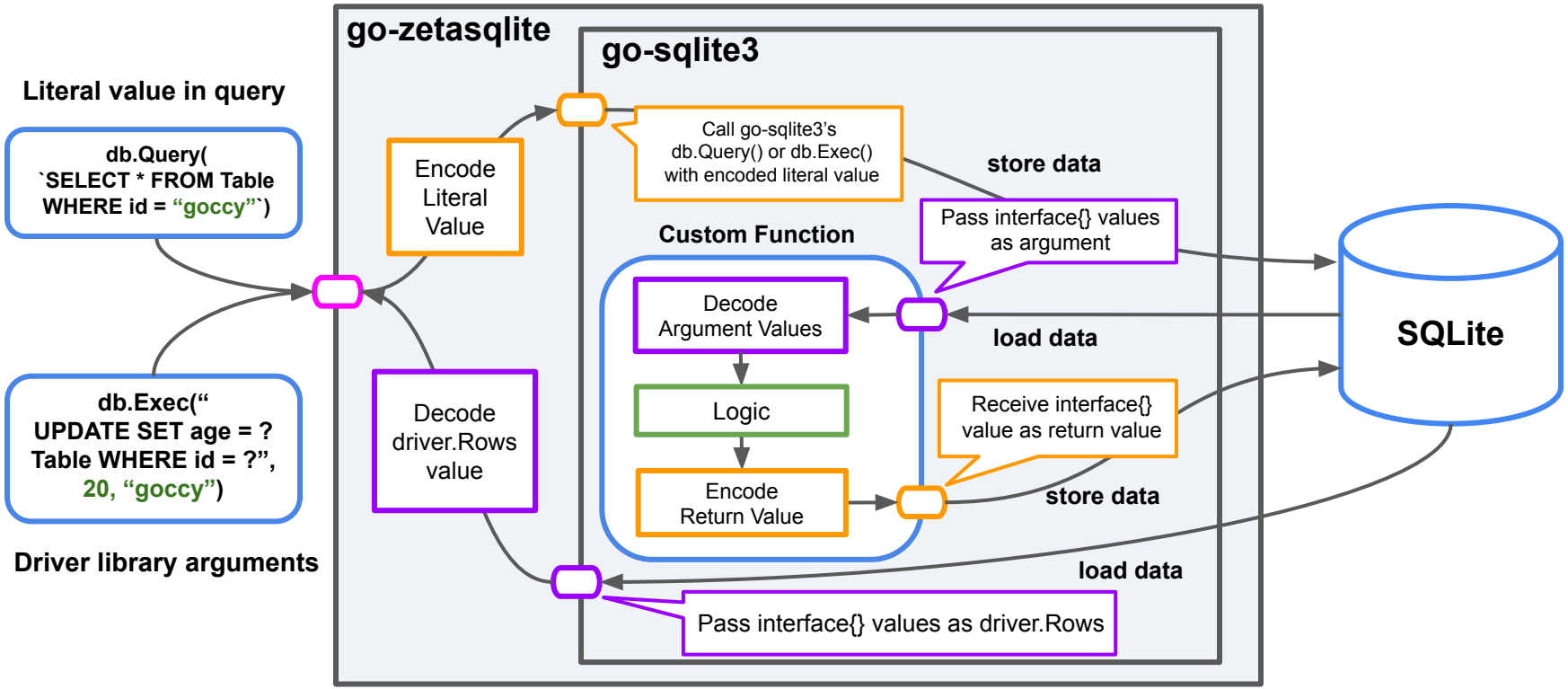
# ZetaSQL クエリを SQLite 向けに変換するときの注意

- SQLite に存在しない型の扱い
- ARRAY 型の値の展開
- WINDOW 関数
- ORDER BY による SQLite に存在しない型の比較と COLLATE

# SQLite に存在しない型の扱い

- BigQuery には STRUCT や ARRAY など、SQLite の型では対応できないものがある
  - BigQuery のデータ型一覧 Data types | BigQuery | Google Cloud
  - SQLite のデータ型一覧 Datatypes In SQLite
- BigQuery と SQLite の間で型変換を行う仕組みを作る必要がある
- go-zetasqlite では INT64 / FLOAT64 / BOOL 以外をエンコードして文字列に変換した上で TEXT 型として保存
  - INT64 => INT / FLOAT64 => DOUBLE / BOOL => BOOLEAN / OTHER => TEXT

# データ型の変換フロー


Literal value in query
db.Query(
`SELECT * FROM Table
WHERE id = “goccy”`)
db.Exec(“
UPDATE SET age = ?
Table WHERE id = ?”,
20, “goccy”)
Driver library arguments
go-zetasqlite
go-sqlite3
Encode
Literal
Value
Decode
driver.Rows
value
Call go-sqlite3’s
db.Query() or db.Exec()
with encoded literal value
store data
Pass interface{} values
as argument
Custom Function
Decode
Argument Values
Logic
Encode
Return Value
load data
Receive interface{}
value as return value
store data
load data
Pass interface{} values as driver.Rows
SQLite

# データ型の変換アルゴリズム

1. JSON形式で型情報と元のデータ値の組み合わせを作る
2. JSON を Base64 エンコードしてエスケープ不要な文字列を作る

例) DATE 型の値 ″2022-10-14″ を変換する場合

″2022-10-14″

{″header″ :″DATE″ ,″body″ :″2022-10-14″ }

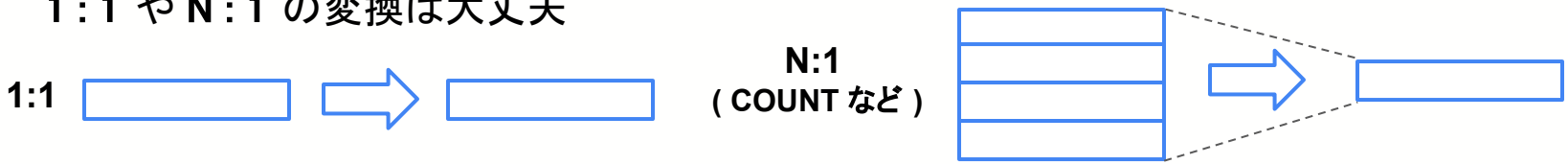
eyJoZWFkZXIiOiJEQVRFIiwiYm9keSI6IjIwMjItMTAtMTQifQo=

# ZetaSQL クエリを SQLite 向けに変換するときの注意

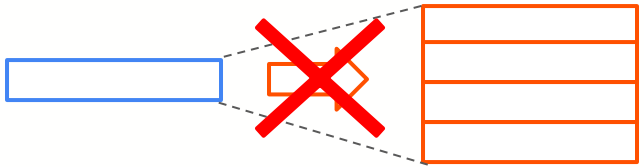
- SQLite に存在しない型の扱い
- **ARRAY 型の値の展開**
- WINDOW 関数
- ORDER BY による SQLite に存在しない型の比較と COLLATE

# ARRAY 型の値の展開

- ARRAY型のリテラルは1つの値を複数の行として SQLite に認識させなければならない場合があるが、通常のやり方ではできない
  - **1 : 1** や **N : 1** の変換は大丈夫




  - **1 : N** はダメ



- zetasqlite では ARRAY 値を JSON 形式にした上で SQLite の json_each 関数を使って分解して対応

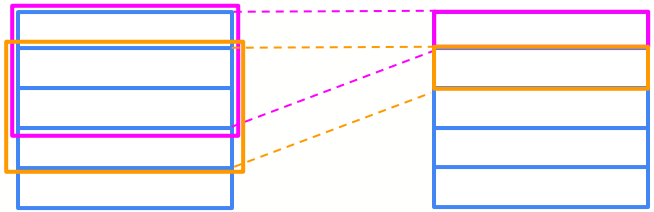
SELECT * FROM ( **SELECT json_each.value FROM json_each("[1, 2, 3]")** )

# ZetaSQL クエリを SQLite 向けに変換するときの注意

- SQLite に存在しない型の扱い
- ARRAY 型の値の展開
- **WINDOW 関数**
- ORDER BY による SQLite に存在しない型の比較と COLLATE

# WINDOW 関数

- 行のグループに対して値を計算して、各行に対して 1 つの結果を返す
  - 行のグループに対して 1 つの結果を返す集計関数とは異なる



- go-sqlite3 は 自作の WINDOW 関数を登録できない
- WINDOW 関数の挙動は集計関数とも異なるので BigQuery 独自の WINDOW 関数を実装することが難しい

# go-zetasqlite の WINDOW 関数

- 自作の集計関数は登録できるので、「行ごとに」「スキャン対象全体を行番号と共に集計関数に渡す」と同じ挙動になる
  - SQLite の ROW_NUMBER 関数で行番号を取得して集計関数の引数にオプションの形で渡す
  - WINDOW 関数を適応するスキャン対象の数を N とすると N^2 の計算量になるので、パフォーマンスはとても悪い
- 例) クエリのイメージ ( CUSTOM_WINDOW は自作の集計関数 )

```
(
  SELECT CUSTOM_WINDOW(col, window_rowid(`row_id`)) FROM (SELECT col FROM TABLE)
) FROM (
  SELECT *, ROW_NUMBER() OVER() AS `row_id` FROM TABLE
)
```

# ZetaSQL クエリを SQLite 向けに変換するときの注意

- SQLite に存在しない型の扱い
- ARRAY 型の値の展開
- WINDOW 関数
- ORDER BY による SQLite に存在しない型の比較と COLLATE

# ORDER BY による SQLite に存在しない型の比較と COLLATE

- ORDER BY / GROUP BY / LIMIT など、実現するには SQLite 組み込みの機能を使わなければいけない場合、そのままでは ORDER BY の比較対象に BigQuery 独自の型を使うことができない
- go-sqlite3 では独自の COLLATE（照合）関数を登録することができるので、比較の際はその関数を通すようにすると良い
- BigQuery には COLLATE 関数を利用して文字列比較の挙動を変更することができる
  - 文字列と一緒に照合情報を管理して照合関数を通して比較すれば同じ挙動になる
- go-zetasqlite では **zetasqlite_collate**　という関数を登録して実現

# goccy/go-zetasql

- ZetaSQL の Parser / Analyzer API などを Go から操作できるライブラリ
- cgo を使って ZetaSQL の C++ API をバインディングすることで実現
  - API の数は約 2700 個
- go get するだけで使える
  - ZetaSQL ライブラリを別途インストールせずに使える
  - ZetaSQL とその依存ライブラリのソースコードをすべて go get 時に cgo の仕組みでコンパイルしている
  - 初回インストール時は ( コンパイルするので ) 時間がかかる
- Static link できる
  - cgo を使っていてもシングルバイナリにできる！

# cgo

- Go から C/C++ の API を利用するための機能
  - go build 時に C/C++ のソースコードに対して C/C++ コンパイラ ( gcc や clang ) を使ってオブジェクトファイルを作り、Go のソースコードをコンパイルしたあとの結果とリンクして単一バイナリにできる
  - C++ のソースをバインドする場合は**extern C** を使って C の世界のシンボルに変換した上で Go から使う ( Go ⇔ C ⇔ C++ )
- C/C++ ライブラリを別でビルドしてもらって、それをリンクする前提で作ることもできるが、その場合は Go ライブラリの利用者が自分で対象のライブラリを install しなければならない
  - 要求しているバージョンを正しくインストールしてもらう保証がない
  - 自分でインストール方法を調べる必要がある

# cgo を使って C/C++ ライブラリをソースからビルドする

- オススメは C/C++ ライブラリのソースコードを Go のリポジトリに含んで cgo の仕組みでビルドすること
  - go build のタイミングで自動でビルドされるので、go get するだけでビルドが走って使えるようになる
  - go-sqlite3 も go-zetasql もこの方法
- Platform 依存のソースコードをビルド時に自動生成するライブラリの場合は、サポートしたい Platform で自動生成処理を走らせ、できあがったファイルを Go の Build Constraint で切り替えると良い
  - e.g.) GitHub Actions で macOS や Windows 向けに走らせる

# バインディング時のファイル構成

- C/C++ライブラリのソースコードとは別の場所にバインディング用のファイルを作り、ライブラリのコードはなるべくそのまま配置する
  - 特定のディレクトリ配下は、ライブラリのソースコードと自動生成されたPlatform依存のコードだけにするような構成
  - **ライブラリの更新がやりやすくなる**
- バインディング用の C/C++ ファイルからライブラリのソースコードを参照する際は #include を使う

# マルチパッケージ構成のライブラリのバインディング(1/2)

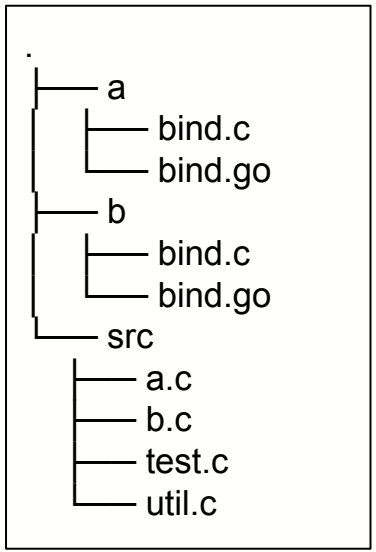
- 複数のパッケージから構成されるライブラリ (e.g. Graphviz / ZetaSQL )をバインドするのは特に難しいので具体例を使って説明
- 次のようなファイル構成のライブラリをバインドすることを考える
  - a.c と b.c は別々のパッケージ(ライブラリ)、テストする場合は test.c から作られるバイナリを利用するような構成 ( util.c は両方のライブラリから参照される便利関数が記述されているイメージ )

```
├── a.c ( library A のソースコード )
├── b.c ( library B のソースコード )
├── test.c ( main 関数がある )
└── util.c ( a.c と b.c から参照される関数がある)
```

- **バインドするために main 関数のある test.c をコンパイル対象に含めず、a.c と b.c を Go の別々のパッケージにして提供することを考える**

# マルチパッケージ構成のライブラリのバインディング(2/2)

- パッケージ a と b を作成してバインディング用のファイル bind.go と bind.c を作成するような構成を考える
  - src 配下にバインディング対象のソースをすべて配置




```
package a
/*
#cgo CFLAGS: -I../src
void FuncA();
*/
import "C"

func FuncA() {
  C.FuncA()
}
```

a/bind.go

```
#include "a.c"
#include "util.c"
```

a/bind.c

パッケージ b の bind.go と bind.c は
A を B に変えただけ
util.c には FuncA と FuncB から
参照される関数が定義されている

**Q.**

a と b を import するとどうなるか

# マルチパッケージのシンボル解決問題

- a と b パッケージを import したプログラムを作成すると
  **duplicated symbol error** になる
  - util.c が a と b の bind.c から参照されているのが原因
  - a と b パッケージそれぞれから作られたオブジェクトファイルに util.c の関数が含まれているので衝突してしまう
- cgo は **パッケージ毎に linker まで動く** ので、a と b パッケージそれぞれの bind.c に util.c を書かないと今度は **undefined symbol error** になる
- もともとのライブラリでは linker による symbol 解決は最後にまとめて1度しか動かないので大丈夫だが、cgo のようにパッケージごとに symbol 解決する方針だと問題になる

# シンボル解決テクニック ( その1 )

- ソースコードを include する手前で衝突するシンボルの名前を書き換える
  - util.c で定義されている共通関数の名前を bind.c 側で書き換えて衝突を避ける

```
#define UtilFunc UtilFuncA
#include "a.c"
#include "util.c"
#undef UtilFunc
```

a/bind.c

```
#define UtilFunc UtilFuncB
#include "b.c"
#include "util.c"
#undef UtilFunc
```

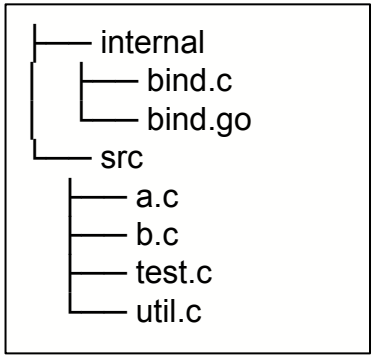
b/bind.c

**問題点**

- util.c を 2 度コンパイルすることになるため、コンパイル時間やバイナリサイズの増加につながる

# シンボル解決テクニック ( その2 )

- パッケージを分けることを諦めて、ひとつのパッケージにまとめる




```
package internal
/*
#cgo CFLAGS: -I../src
void FuncA();
void FuncB();
*/
import "C"

func FuncA() {
  C.FuncA()
}
func FuncB() {
  C.FuncB()
}
```

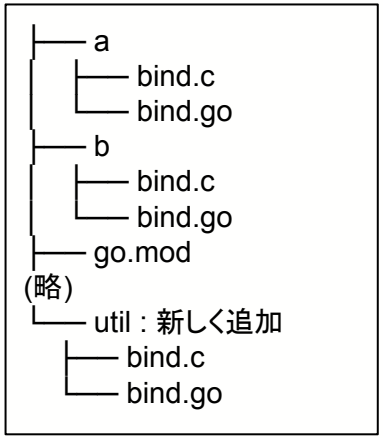
```
#include "a.c"
#include "b.c"
#include "util.c"
```

**問題点**

- bind.go と bind.c が巨大になると、大量のメモリを食ったりコンパイルに時間がかかったりする ( Go はパッケージごとにビルドを並列化するので、その恩恵を受けられない )

# シンボル解決テクニック ( その3 )

- 共通コードである util.c をパッケージとして切り出し、そのシンボルを
  パッケージ a と b から参照することで再利用するような構成




```
package a
/*
#cgo CFLAGS: -I../src
void FuncA();
*/
import "C"
import (
  "example/util"
)
//export export_UtilFuncA
func export_UtilFuncA(v C.int) {
  util.UtilFunc(int(v))
}
func FuncA() {
  C.FuncA()
}
```

a/bind.go

```
#include "_cgo_export.h"

#define UtilFunc export_UtilFuncA
#include "a.c"
#undef UtilFunc
```

a/bind.c

この組み合わせで C => Go の関数
呼び出しができる

util package で作られた関数を Go 側で
import したあと、export directive で C側に
その関数を公開する

# go-zetasql はどうやっているか

- 2 と 3 の組み合わせで実現している
- 3 は 1 と 2 の問題点を解決してくれるが、実装するのが一番大変なので、
  ZetaSQL が依存している 3rd party ライブラリの一部で利用するに
  留めている
- 現状ではソースコードのほとんどが単一パッケージとしてビルドされるため、ビルド
  時間がかかる原因になっている

- 今回のバインディングの話は goccy/cgo-multipkg-example にまとまっている

# おまけ: cgoを使っていても静的バイナリは作れる

- CGO_ENABLED=1 を付けると単一バイナリが作れないと勘違いしている例をたまに見かける
- **-linkmode external -extldflags "-static"** を付けてビルドすることで cgo が有効でも static link できる
  - go-sqlite3 や go-zetasql のようにソースコードを含む cgo ライブラリと相性が良い
  - macOS の場合は framework などを利用してしまうと static link できないが、Linux なら大丈夫！

# まとめ

- **BigQueryエミュレータがどのように作られたかを解説**
  - ZetaSQL から SQLite に変換する際に注意しなければいけなかったこと
  - cgo を使ってマルチパッケージ構成のライブラリをバインディングする際の注意
- **goccy/bigquery-emulator**
  - BigQuery エミュレータを様々なインストール方法で提供
  - テストやローカルでの開発に活用できる
- **goccy/go-zetasqlite**
  - BigQuery に依存せずに ZetaSQL クエリを評価できるデータベースドライバライブラリ
- **goccy/go-zetasql**
  - ZetaSQL の Parser / Analyzer / Formatter が使える
  - Linter や Formatter などのツール開発に便利