

Go Conference
2022 Spring

database/sqlの仕組みについて

Takuma Shibuya

自己紹介

Takuma Shibuya

Twitter @sivchari
GitHub sivchari

Company: Cyber Agent

Contributor

Go
Kubernetes
golangci-lint

Go Conference 2021 Autumn
Go COnference 2022 Spring



内容

- database/sqlの概要
- database/sqlの内部実装
- goroutine safeなコネクションプールの管理

01 database/sqlの概要

database/sqlって何？

- Goが提供するDB操作を行うための標準パッケージ
 - 実際に利用する際はdatabase/sqlと任意のSQL Driverを用い、database/sqlを介して利用する
 - database/sqlはSQL(or SQL-Like)のためのinterfaceを提供している
 - MySQLのように特定の実装をしていない
 - 具体的な実装は各SQL Driverが実装する(interfaceを満たせばOK)
(e.g. go-sql-driver(MySQL), pq(PostgreSQL), sqlite3(SQLite3))
- 公式のwikiには56個掲載されている

ORM

- GORM
 - GORMが提供しているDriverがdatabase/sqlを満たすように実装している
- Ent
 - database/sqlの実装をラップして拡張している

→ ORMはdatabase/sqlとサポートするdriverをラップすることでinterfaceを満たしながら機能を拡張している

02 database/sqlの内部実装

サンプルコード

```
package main

import (
    "context"
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    ctx := context.Background()
    db, _ := sql.Open("mysql", "dsn")
    _ = db.PingContext(ctx)
}
```


サンプルコード

```
package main

import (
    "context"
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    ctx := context.Background()
    db, _ := sql.Open("mysql", "dsn")
    _ = db.PingContext(ctx)
}
```

Blank import

Go言語仕様

インポート宣言は、インポートするパッケージとインポートされるパッケージの依存関係を宣言するものである。パッケージがそれ自身を直接または間接的にインポートすることや、エクスポートされた識別子を一切参照せずにパッケージを直接インポートすることは違法です。パッケージの副作用(初期化)のためだけにパッケージをインポートするには、明示的なパッケージ名として空白の識別子を使用します。

Blank import

Go言語仕様

インポート宣言は、インポートするパッケージとインポートされるパッケージの依存関係を宣言するものである。パッケージがそれ自身を直接または間接的にインポートすることや、エクスポートされた識別子を一切参照せずにパッケージを直接インポートすることは違法です。パッケージの副作用(初期化)のためだけにパッケージをインポートするには、明示的なパッケージ名として空白の識別子を使用します。

パッケージの副作用(初期化)

- Goのinitは組み込みの初期化関数
- 依存先のパッケージにinitがある場合先に依存先が解決される
- initを同一パッケージで複数書いた場合上を書いてあるnitから順番に解決される

サンプルコード

```
package main

import (
    "context"
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    ctx := context.Background()
    db, _ := sql.Open("mysql", "dsn")
    _ = db.PingContext(ctx)
}
```

mysql/driver.go

```
func init() {  
    sql.Register("mysql", &MySQLDriver{})  
}
```

mysql/driver.go

```
func init() {  
    sql.Register("mysql", &MySQLDriver{})  
}
```

sql.go

```
// Register makes a database driver available by the provided name.  
// If Register is called twice with the same name or if driver is nil,  
// it panics.
```

```
// 和訳
```

```
// Registerは与えられたname引数から利用可能なdatabase driverを作成します。
```

```
// もしRegisterが同じname引数で2度呼ばれたり、nilであればpanicします。
```

```
func Register(name string, driver driver.Driver) {  
    driversMu.Lock()  
    defer driversMu.Unlock()  
    if driver == nil {  
        panic("sql: Register driver is nil")  
    }  
    if _, dup := drivers[name]; dup {  
        panic("sql: Register called twice for driver " + name)  
    }  
    drivers[name] = driver  
}
```


sql.go

```
// Register makes a database driver available by the provided name.  
// If Register is called twice with the same name or if driver is nil,  
// it panics.
```

```
// 和訳
```

```
// Registerは与えられたname引数から利用可能なdatabase driverを作成します。
```

```
// もしRegisterが同じname引数で2度呼ばれたり、nilであればpanicします。
```

```
func Register(name string, driver driver.Driver) {  
    driversMu.Lock()  
    defer driversMu.Unlock()  
    if driver == nil {  
        panic("sql: Register driver is nil")  
    }  
    if _, dup := drivers[name]; dup {  
        panic("sql: Register called twice for driver " + name)  
    }  
    drivers[name] = driver  
}
```

sql.go

```
// Register makes a database driver available by the provided name.  
// If Register is called twice with the same name or if driver is nil,  
// it panics.
```

```
// 和訳
```

```
// Registerは与えられたname引数から利用可能なdatabase driverを作成します。
```

```
// もしRegisterが同じname引数で2度呼ばれたり、nilであればpanicします。
```

```
func Register(name string, driver driver.Driver) {  
    driversMu.Lock()  
    defer driversMu.Unlock()  
    if driver == nil {  
        panic("sql: Register driver is nil")  
    }  
    if _, dup := drivers[name]; dup {  
        panic("sql: Register called twice for driver " + name)  
    }  
    drivers[name] = driver  
}
```

sql.go

```
// Register makes a database driver available by the provided name.  
// If Register is called twice with the same name or if driver is nil,  
// it panics.
```

```
// 和訳
```

```
// Registerは与えられたname引数から利用可能なdatabase driverを作成します。
```

```
// もしRegisterが同じname引数で2度呼ばれたり、nilであればpanicします。
```

```
func Register(name string, driver driver.Driver) {  
    driversMu.Lock()  
    defer driversMu.Unlock()  
    if driver == nil {  
        panic("sql: Register driver is nil")  
    }  
    if _, dup := drivers[name]; dup {  
        panic("sql: Register called twice for driver " + name)  
    }  
    drivers[name] = driver  
}
```

sql.go

```
// Register makes a database driver available by the provided name.  
// If Register is called twice with the same name or if driver is nil,  
// it panics.
```

```
// 和訳
```

```
// Registerは与えられたname引数から利用可能なdatabase driverを作成します。
```

```
// もしRegisterが同じname引数で2度呼ばれたり、nilであればpanicします。
```

```
func Register(name string, driver driver.Driver) {  
    driversMu.Lock()  
    defer driversMu.Unlock()  
    if driver == nil {  
        panic("sql: Register driver is nil")  
    }  
    if _, dup := drivers[name]; dup {  
        panic("sql: Register called twice for driver " + name)  
    }  
    drivers[name] = driver  
}
```

同じname引数で2度呼んでみる

```
package main

import (
    "database/sql"

    // initで"mysql"をkeyとしてすでにRegisterが呼ばれている。
    "github.com/go-sql-driver/mysql"
)

func init() {
    sql.Register("mysql", &mysql.MySQLDriver{})
}

func main() {
}
```

同一DB DriverのRegisterを防ぐ

- go-mysql
- go-sql-driver

サンプルコード

```
package main

import (
    "context"
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    ctx := context.Background()
    db, _ := sql.Open("mysql", "dsn")
    _ = db.PingContext(ctx)
}
```

sql.Open

```
func Open(driverName, dataSourceName string) (*DB, error) {  
    driversMu.RLock()  
    driveri, ok := drivers[driverName]  
    driversMu.RUnlock()  
    if !ok {  
        return nil, fmt.Errorf("sql: unknown driver %q (forgotten import?)", driverName)  
    }  
  
    if driverCtx, ok := driveri.(driver.DriverContext); ok {  
        connector, err := driverCtx.OpenConnector(dataSourceName)  
        if err != nil {  
            return nil, err  
        }  
        return OpenDB(connector), nil  
    }  
  
    return OpenDB(dsnConnector{dsn: dataSourceName, driver: driveri}), nil  
}
```


sql.Open

```
func Open(driverName, dataSourceName string) (*DB, error) {
    driversMu.RLock()
    driveri, ok := drivers[driverName]
    driversMu.RUnlock()
    if !ok {
        return nil, fmt.Errorf("sql: unknown driver %q (forgotten import?", driverName)
    }

    if driverCtx, ok := driveri.(driver.DriverContext); ok {
        connector, err := driverCtx.OpenConnector(dataSourceName)
        if err != nil {
            return nil, err
        }
        return OpenDB(connector), nil
    }

    return OpenDB(dsnConnector{dsn: dataSourceName, driver: driveri}), nil
}
```

sql.Open

```
func Open(driverName, dataSourceName string) (*DB, error) {
    driversMu.RLock()
    driveri, ok := drivers[driverName]
    driversMu.RUnlock()
    if !ok {
        return nil, fmt.Errorf("sql: unknown driver %q (forgotten import?", driverName)
    }

    if driverCtx, ok := driveri.(driver.DriverContext); ok {
        connector, err := driverCtx.OpenConnector(dataSourceName)
        if err != nil {
            return nil, err
        }
        return OpenDB(connector), nil
    }

    return OpenDB(dsnConnector{dsn: dataSourceName, driver: driveri}), nil
}
```

OpenDB

- driver.DriverContextを満たしていればDBコネクションを取得する
- 実際にconnectionを取得するopenNewConnectionでは内部で抽象化されたConnectを呼ぶ
 - Driverの実装に依存するため、Ping/PingContextを呼ぶようにコメントされている

サンプルコード

```
package main

import (
    "context"
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    ctx := context.Background()
    db, _ := sql.Open("mysql", "dsn")
    _ = db.PingContext(ctx)
}
```

Ping

```
func (db *DB) Ping() error {  
    return db.PingContext(context.Background())  
}
```

PingContext

```
func (db *DB) PingContext(ctx context.Context) error {  
    var dc *driverConn  
    var err error  
    var isBadConn bool  
    for i := 0; i < maxBadConnRetries; i++ {  
        dc, err = db.conn(ctx, cachedOrNewConn)  
        isBadConn = errors.Is(err, driver.ErrBadConn)  
        if !isBadConn {  
            break  
        }  
    }  
    if isBadConn {  
        dc, err = db.conn(ctx, alwaysNewConn)  
    }  
    if err != nil {  
        return err  
    }  
  
    return db.pingDC(ctx, dc, dc.releaseConn)  
}
```

PingContext

```
func (db *DB) PingContext(ctx context.Context) error {  
    var dc *driverConn  
    var err error  
    var isBadConn bool  
    for i := 0; i < maxBadConnRetries; i++ {  
        dc, err = db.conn(ctx, cachedOrNewConn)  
        isBadConn = errors.Is(err, driver.ErrBadConn)  
        if !isBadConn {  
            break  
        }  
    }  
    if isBadConn {  
        dc, err = db.conn(ctx, alwaysNewConn)  
    }  
    if err != nil {  
        return err  
    }  
  
    return db.pingDC(ctx, dc, dc.releaseConn)  
}
```

PingContext

```
func (db *DB) PingContext(ctx context.Context) error {  
    var dc *driverConn  
    var err error  
    var isBadConn bool  
    for i := 0; i < maxBadConnRetries; i++ {  
        dc, err = db.conn(ctx, cachedOrNewConn)  
        isBadConn = errors.Is(err, driver.ErrBadConn)  
        if !isBadConn {  
            break  
        }  
    }  
    if isBadConn {  
        dc, err = db.conn(ctx, alwaysNewConn)  
    }  
    if err != nil {  
        return err  
    }  
  
    return db.pingDC(ctx, dc, dc.releaseConn)  
}
```


PingContext

```
func (db *DB) PingContext(ctx context.Context) error {  
    var dc *driverConn  
    var err error  
    var isBadConn bool  
    for i := 0; i < maxBadConnRetries; i++ {  
        dc, err = db.conn(ctx, cachedOrNewConn)  
        isBadConn = errors.Is(err, driver.ErrBadConn)  
        if !isBadConn {  
            break  
        }  
    }  
    if isBadConn {  
        dc, err = db.conn(ctx, alwaysNewConn)  
    }  
    if err != nil {  
        return err  
    }  
  
    return db.pingDC(ctx, dc, dc.releaseConn)  
}
```

db.pingDC

```
func (db *DB) pingDC(ctx context.Context, dc *driverConn, release func(error)) error {  
    var err error  
    if pinger, ok := dc.ci.(driver.Pinger); ok {  
        withLock(dc, func() {  
            err = pinger.Ping(ctx)  
        })  
    }  
    release(err)  
    return err  
}
```

db.pingDC

```
func (db *DB) pingDC(ctx context.Context, dc *driverConn, release func(error)) error {  
    var err error  
    if pinger, ok := dc.ci.(driver.Pinger); ok {  
        withLock(dc, func() {  
            err = pinger.Ping(ctx)  
        })  
    }  
    release(err)  
    return err  
}
```

db.pingDC

```
func (db *DB) pingDC(ctx context.Context, dc *driverConn, release func(error)) error {  
    var err error  
    if pinger, ok := dc.ci.(driver.Pinger); ok {  
        withLock(dc, func() {  
            err = pinger.Ping(ctx)  
        })  
    }  
    release(err)  
    return err  
}
```

releaseConn

```
func (dc *driverConn) releaseConn(err error) {  
    dc.db.putConn(dc, err, true)  
}
```

03

goroutine safeなコネクションプールの管理

goroutine safe

sql.DB

DB is a database handle representing a pool of zero or more underlying connections. It's safe for concurrent use by multiple goroutines.

sql.DBはgoroutine safeであることを保証している(e.g. net/http)

もし対策していないとリクエストごとのgoroutineがコネクションプールを操作するためraceが発生する可能性がある。

E.g.

<https://github.com/golang/go/blob/b55a2fb3b0d67b346bac871737b862f16e5a6447/src/net/http/server.go#L3010>

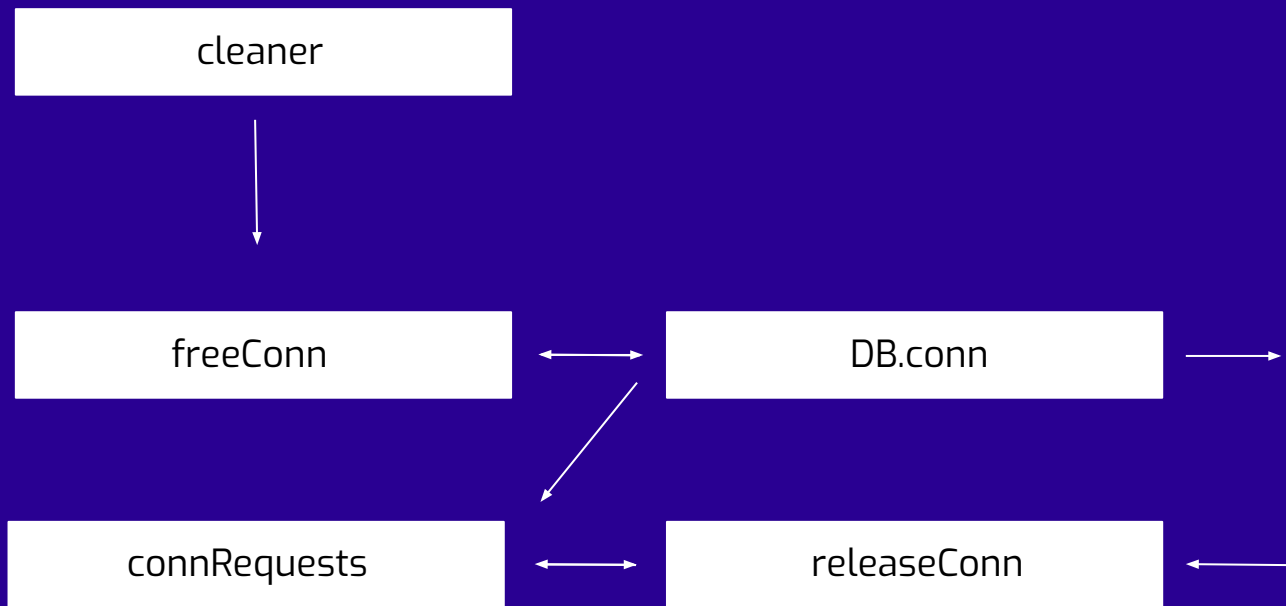
Goals of the sql and sql/driver package

- 並行処理をうまく処理する。ユーザーはデータベースの接続ごとのスレッドセーフ (goroutine safe)の問題を気にするべきではなく、コネクションプールを自分で管理すべきでもない。
- sql.DBはインスタンスを共有することが可能であるべき。
- 複数のgoroutineが余分な同期を取る必要がない。

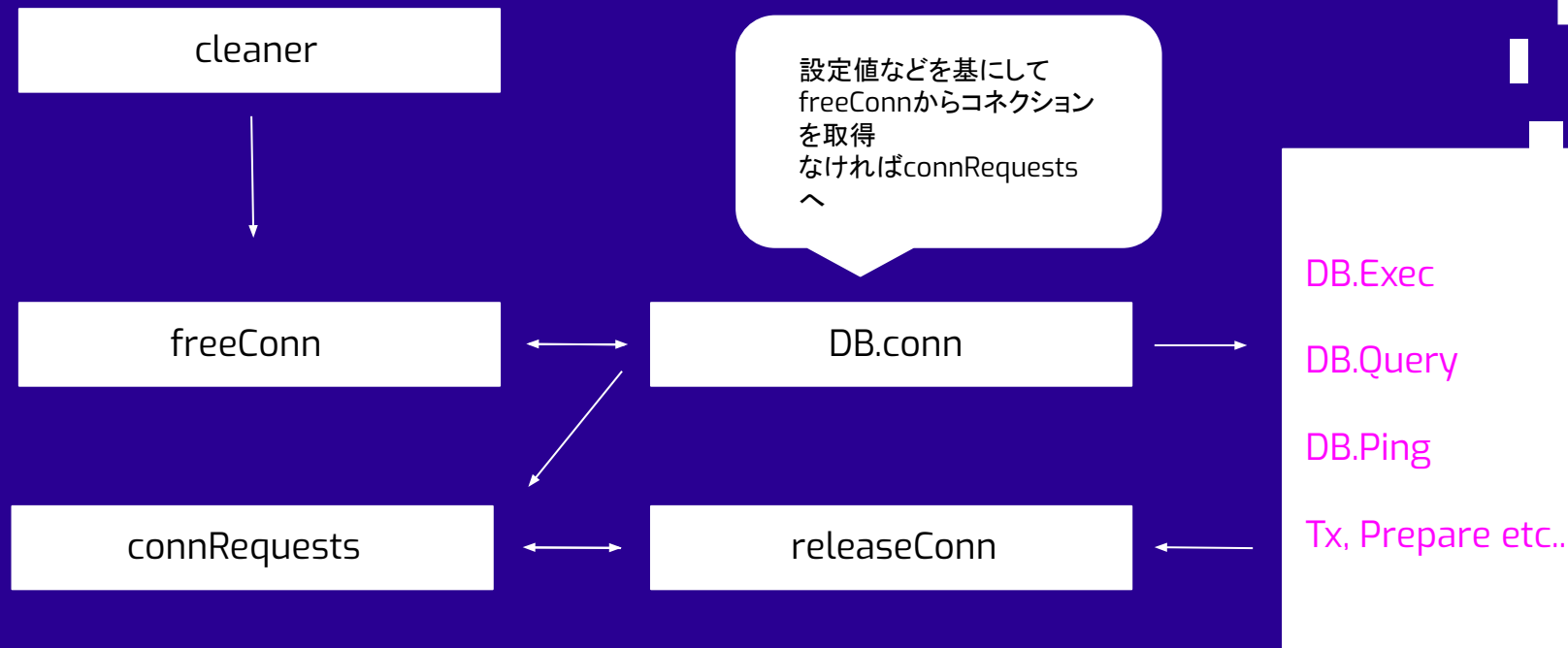
登場人物

- `mu sync.Mutex` `sql.DB`をgoroutine safeにするための `Mutex`
- `freeConn []*driverConn` idle状態のconnection, 空きがあればここから使う
- `connRequests map[uint64]chan connRequest` `freeConn`に空きがない時の待ち行列
- `nextRequest uint64` 待ち行列から次に実行する `connRequest`
- `cleanerCh chan struct{}` `SetConnMaxIdleTime`, `SetConnMaxLifeTime`の値などで`freeConn`をcleanにする
- `func (db *DB) connectionCleaner() {}` `cleanerCh`, `SetConnMaxXXX`の値などで定期実行する

相関図



相関図



DB.Exec/ExecContext → DB.exec → db.conn

```
func (db *DB) exec(ctx context.Context, query string, args []any, strategy connReuseStrategy) (Result, error) {  
    dc, err := db.conn(ctx, strategy)  
    if err != nil {  
        return nil, err  
    }  
    return db.execDC(ctx, dc, dc.releaseConn, query, args)  
}
```

DB.Query/QueryContext → DB.query → db.conn

```
func (db *DB) query(ctx context.Context, query string, args []any, strategy connReuseStrategy)
(*Rows, error) {
    dc, err := db.conn(ctx, strategy)
    if err != nil {
        return nil, err
    }

    return db.queryDC(ctx, nil, dc, dc.releaseConn, query, args)
}
```

DB.conn

```
if strategy == cachedOrNewConn && last >= 0 {
    // Reuse the lowest idle time connection so we can close
    // connections which remain idle as soon as possible.
    conn := db.freeConn[last]
    db.freeConn = db.freeConn[:last]
    conn.inUse = true
    if conn.expired(lifetime) {
        db.maxLifetimeClosed++
        db.mu.Unlock()
        conn.Close()
        return nil, driver.ErrBadConn
    }
    db.mu.Unlock()

    // Reset the session if required.
    if err := conn.resetSession(ctx); errors.Is(err, driver.ErrBadConn) {
        conn.Close()
        return nil, err
    }

    return conn, nil
}
```

DB.conn

```
if strategy == cachedOrNewConn && last >= 0 {  
    // Reuse the lowest idle time connection so we can close  
    // connections which remain idle as soon as possible.  
    conn := db.freeConn[last]  
    db.freeConn = db.freeConn[:last]  
    conn.inUse = true  
    if conn.expired(lifetime) {  
        db.maxLifetimeClosed++  
        db.mu.Unlock()  
        conn.Close()  
        return nil, driver.ErrBadConn  
    }  
    db.mu.Unlock()  
  
    // Reset the session if required.  
    if err := conn.resetSession(ctx); errors.Is(err, driver.ErrBadConn) {  
        conn.Close()  
        return nil, err  
    }  
  
    return conn, nil  
}
```

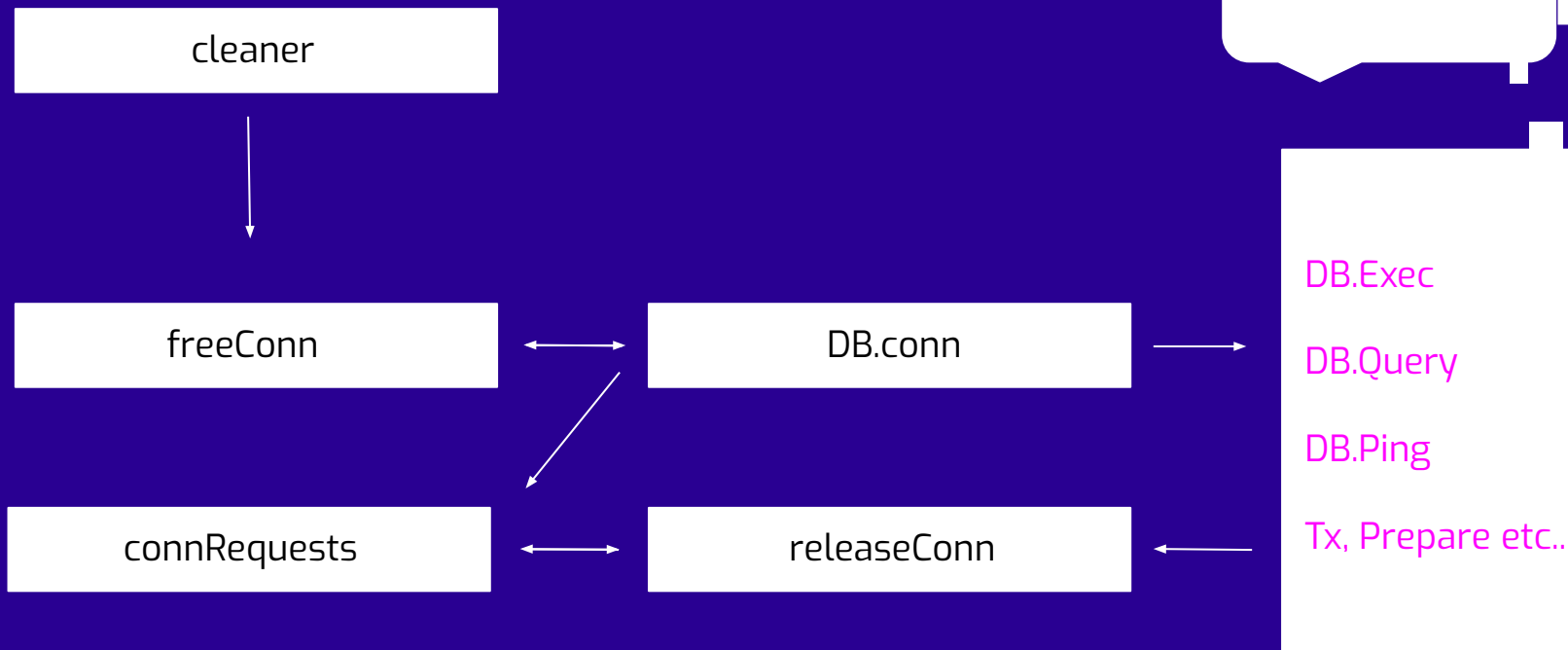
DB.conn

```
if db.maxOpen > 0 && db.numOpen >= db.maxOpen {  
    // Make the connRequest channel. It's buffered so that the  
    // connectionOpener doesn't block while waiting for the req to be read.  
    req := make(chan connRequest, 1)  
    reqKey := db.nextRequestKeyLocked()  
    db.connRequests[reqKey] = req  
    db.waitCount++  
    db.mu.Unlock()  
}
```

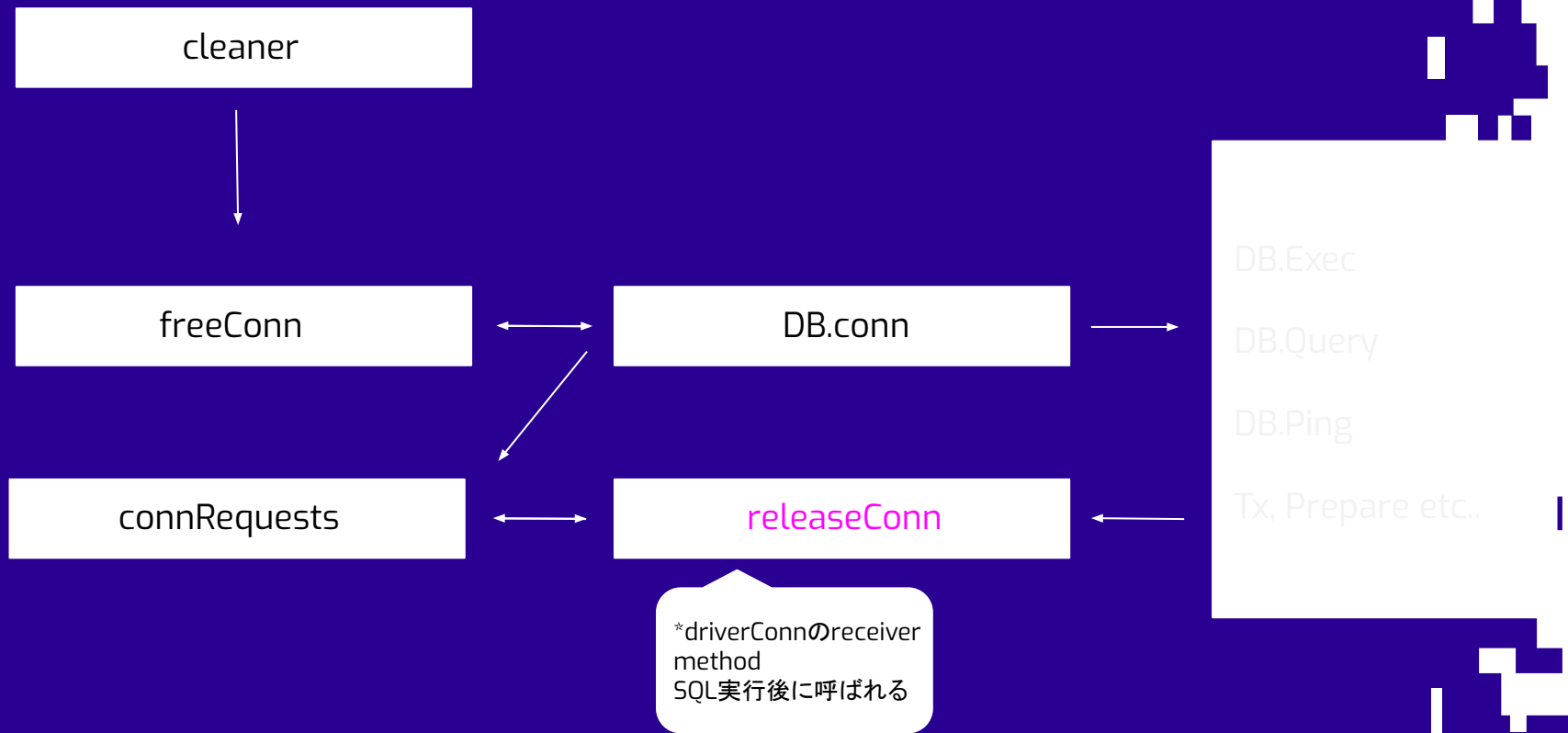

DB.conn

```
if db.maxOpen > 0 && db.numOpen >= db.maxOpen {  
    // Make the connRequest channel. It's buffered so that the  
    // connectionOpener doesn't block while waiting for the req to be read.  
    req := make(chan connRequest, 1)  
    reqKey := db.nextRequestKeyLocked()  
    db.connRequests[reqKey] = req  
    db.waitCount++  
    db.mu.Unlock()  
}
```

相関図



相関図



releaseConn

```
func (dc *driverConn) releaseConn(err error) {  
    dc.db.putConn(dc, err, true)  
}
```

putConn

```
dc.inUse = false
dc.returnedAt = nowFunc()
// nowFunc returns the current time; it's overridden in tests. var nowFunc = time.Now

added := db.putConnDBLocked(dc, nil)
db.mu.Unlock()

if !added {
    dc.Close()
    return
}
```

putConn

```
dc.inUse = false
dc.returnedAt = nowFunc()
// nowFunc returns the current time; it's overridden in tests. var nowFunc = time.Now

added := db.putConnDBLocked(dc, nil)
db.mu.Unlock()

if !added {
    dc.Close()
    return
}
```

putConn

```
dc.inUse = false
dc.returnedAt = nowFunc()
// nowFunc returns the current time; it's overridden in tests. var nowFunc = time.Now

added := db.putConnDBLocked(dc, nil)
db.mu.Unlock()

if !added {
    dc.Close()
    return
}
```

db.putConnDBLocked

```
if db.maxOpen > 0 && db.numOpen > db.maxOpen {
    return false
}
if c := len(db.connRequests); c > 0 {
    var req chan connRequest
    var reqKey uint64
    for reqKey, req = range db.connRequests {
        break
    }
    delete(db.connRequests, reqKey) // Remove from pending requests.
    if err == nil {
        dc.inUse = true
    }
    req <- connRequest{
        conn: dc,
        err:  err,
    }
    return true
} else if err == nil && !db.closed {
    if db.maxIdleConnsLocked() > len(db.freeConn) {
        db.freeConn = append(db.freeConn, dc)
        db.startCleanerLocked()
        return true
    }
    db.maxIdleClosed++
}
return false
```


db.putConnDBLocked

```
if db.maxOpen > 0 && db.numOpen > db.maxOpen {  
    return false  
}
```

db.putConnDBLocked

```
if c := len(db.connRequests); c > 0 {  
    var req chan connRequest  
    var reqKey uint64  
    for reqKey, req = range db.connRequests {  
        break  
    }  
    delete(db.connRequests, reqKey) // Remove from pending requests.  
    if err == nil {  
        dc.inUse = true  
    }  
    req <- connRequest{ // DB.connでselect で受け取っている  
        conn: dc,  
        err:  err,  
    }  
    return true  
}
```

db.putConnDBLocked

```
} else if err == nil && !db.closed {  
    if db.maxIdleConnsLocked() > len(db.freeConn) {  
        db.freeConn = append(db.freeConn, dc)  
        db.startCleanerLocked()  
        return true  
    }  
    db.maxIdleClosed++  
}
```

db.putConnDBLocked

```
} else if err == nil && !db.closed {  
    if db.maxIdleConnsLocked() > len(db.freeConn) {  
        db.freeConn = append(db.freeConn, dc)  
        db.startCleanerLocked()  
        return true  
    }  
    db.maxIdleClosed++ // DB.Stats用  
}
```

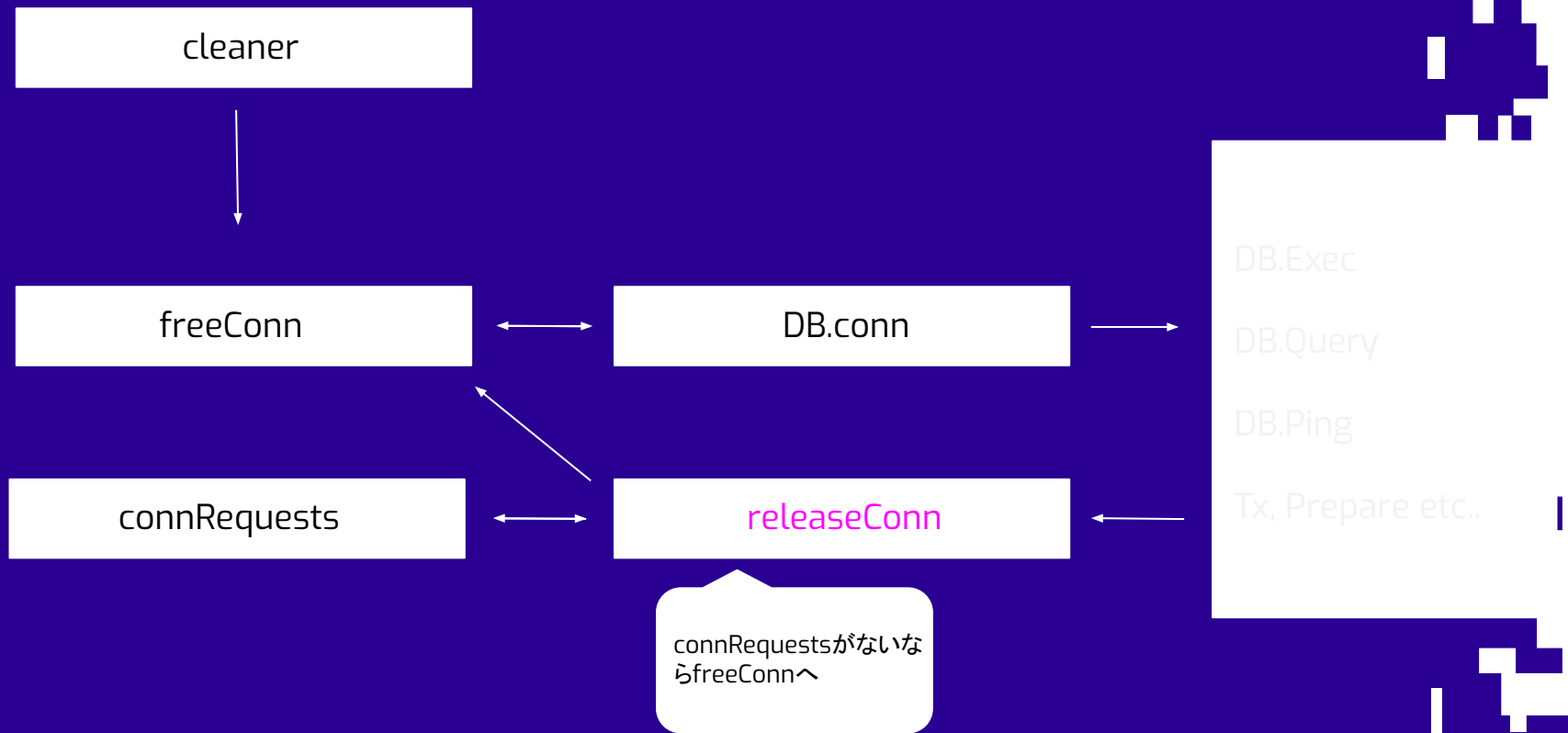
putConn

```
dc.inUse = false
dc.returnedAt = nowFunc()
// nowFunc returns the current time; it's overridden in tests. var nowFunc = time.Now

added := db.putConnDBLocked(dc, nil)
db.mu.Unlock()

if !added {
    dc.Close()
    return
}
```

相関図



timer/cleanerChを受け取ると実行

相関図

cleaner

freeConn

DB.conn

connRequests

releaseConn

connRequestsがないならfreeConnへ

DB.Exec

DB.Query

DB.Ping

Tx, Prepare etc..

DB.startCleanerLocked

```
// startCleanerLocked starts connectionCleaner if needed.
func (db *DB) startCleanerLocked() {
    if (db.maxLifetime > 0 || db.maxIdleTime > 0) && db.numOpen > 0 && db.cleanerCh == nil {
        db.cleanerCh = make(chan struct{}, 1)
        go db.connectionCleaner(db.shortestIdleTimeLocked())
    }
}
```


DB.startCleanerLocked

```
// startCleanerLocked starts connectionCleaner if needed.
func (db *DB) startCleanerLocked() {
    if (db.maxLifetime > 0 || db.maxIdleTime > 0) && db.numOpen > 0 && db.cleanerCh == nil {
        db.cleanerCh = make(chan struct{}, 1)
        go db.connectionCleaner(db.shortestIdleTimeLocked())
    }
}
```

DB.connectionCleaner

```
func (db *DB) connectionCleaner(d time.Duration) {  
    for {  
        select {  
        case <-t.C:  
        case <-db.cleanerCh: // maxLifetime was changed or db was closed.  
        }  
  
        d, closing := db.connectionCleanerRunLocked(d)  
        db.mu.Unlock()  
        for _, c := range closing {  
            c.Close()  
        }  
    }  
}
```

DB.connectionCleaner

```
func (db *DB) connectionCleaner(d time.Duration) {  
    for {  
        select {  
        case <-t.C:  
        case <-db.cleanerCh: // maxLifetime was changed or db was closed.  
        }  
  
        d, closing := db.connectionCleanerRunLocked(d)  
        db.mu.Unlock()  
        for _, c := range closing {  
            c.Close()  
        }  
    }  
}
```

DB.connectionCleaner

```
func (db *DB) connectionCleaner(d time.Duration) {  
    for {  
        select {  
        case <-t.C:  
        case <-db.cleanerCh: // maxLifetime was changed or db was closed.  
        }  
  
        d, closing := db.connectionCleanerRunLocked(d)  
        db.mu.Unlock()  
        for _, c := range closing {  
            c.Close()  
        }  
    }  
}
```

DB.connectionCleanerRunLocked

```
func (db *DB) connectionCleanerRunLocked(d time.Duration) (time.Duration, []*driverConn) {  
    var closing []*driverConn  
    if db.maxIdleTime > 0 {  
        for i := last; i >= 0; i-- {  
            // snip  
            closing = db.freeConn[i:i]  
        }  
    }  
  
    if db.maxLifetime > 0 {  
        for i := 0; i < len(db.freeConn); i++ {  
            if c.createdAt.Before(expiredSince) {  
                closing = append(closing, c)  
            }  
        }  
    }  
    return d, closing  
}
```

DB.connectionCleanerRunLocked

```
func (db *DB) connectionCleanerRunLocked(d time.Duration) (time.Duration, []*driverConn) {  
    var closing []*driverConn  
    if db.maxIdleTime > 0 {  
        for i := last; i >= 0; i-- {  
            // snip  
            closing = db.freeConn[i:i]  
        }  
    }  
  
    if db.maxLifetime > 0 {  
        for i := 0; i < len(db.freeConn); i++ {  
            if c.createdAt.Before(expiredSince) {  
                closing = append(closing, c)  
            }  
        }  
    }  
    return d, closing  
}
```

DB.connectionCleaner

```
func (db *DB) connectionCleaner(d time.Duration) {  
    for {  
        select {  
        case <-t.C:  
        case <-db.cleanerCh: // maxLifetime was changed or db was closed.  
        }  
  
        d, closing := db.connectionCleanerRunLocked(d)  
        db.mu.Unlock()  
        for _, c := range closing {  
            c.Close()  
        }  
    }  
}
```

まとめ

- database/sqlの拡張性の高さ
 - Interfaceをどのように使うか
 - 抽象的に書くことであらゆるDBに対応できる
- Goのtips的な書き方
 - time.NowをnowFuncとして置いておく
 - for rangeで最後のconnを取得する
- Goの言語仕様への理解
 - Blank import

Thank you for watching !