Go sql连接池原理源码解析

通过一个实际的查询语句来了解一下database/sql包的连接池实现原理

一、驱动注册

在初始化DB的地方导入mysql驱动

```
import _ "github.com/go-sql-driver/mysql"
```

导入此包的目的是执行github.com/go-sql-driver/mysql/driver.go中的init函数

```
func init() {
    // 调用database/sql/sql.go中的Register函数注册mysql驱动
    sql.Register("mysql", &MySQLDriver{})
}
```

二、初始化DB

open返回一个DB结构体（并发安全，就是这个DB管理了一个连接池），并启动一个goroutine监听openerCh通道，当有请求时获取新的连接，此时并没有创建连接

```
func Open(driverName, dataSourceName string) (*DB, error) {
    driversMu.RLock()
    // 获取刚刚注册的数据库驱动
    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
        }
        // 返回一个*DB
        return OpenDB(connector), nil
    }

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

func OpenDB(c driver.Connector) *DB {
   ctx, cancel := context.WithCancel(context.Background())
   db := &DB{
      connector:    c,
      openerCh:     make(chan struct{}, connectionRequestQueueSize),
      lastPut:      make(map[*driverConn]string),
      connRequests: make(map[uint64]chan connRequest),
      stop:         cancel,
   }

   // 开启一个goroutine，监听openerCh通道，当有请求时获取新的连接
   go db.connectionOpener(ctx)
   // 返回db
   return db
}
```

看一下这个DB结构体

```
type DB struct {
    // Atomic access only. At top of struct to prevent mis-alignment
    // on 32-bit platforms. Of type time.Duration.
    waitDuration int64 // Total time waited for new connections.

    connector driver.Connector
    // numClosed is an atomic counter which represents a total number of
    // closed connections. Stmt.openStmt checks it before cleaning closed
    // connections in Stmt.css.
    numClosed uint64

    mu           sync.Mutex // protects following fields
    freeConn     []*driverConn // 空闲连接，也就是连接池
    connRequests map[uint64]chan connRequest
    nextRequest  uint64 // Next key to use in connRequests.
    numOpen      int    // number of opened and pending open connections
    // Used to signal the need for new connections
    // a goroutine running connectionOpener() reads on this chan and
    // maybeOpenNewConnections sends on the chan (one send per needed connection)
    // It is closed during db.Close(). The close tells the connectionOpener
    // goroutine to exit.
    openerCh          chan struct{}
    closed            bool
    dep               map[finalCloser]depSet
    lastPut           map[*driverConn]string // stacktrace of last conn's put; debug only
    maxIdleCount      int                    // zero means defaultMaxIdleConns; negative means 0 最大空闲连接数
    maxOpen           int                    // <= 0 means unlimited 最大连接数
    maxLifetime       time.Duration          // maximum amount of time a connection may be reused
    maxIdleTime       time.Duration          // maximum amount of time a connection may be idle before being
    cleanerCh         chan struct{}
    waitCount         int64 // Total number of connections waited for.
    maxIdleClosed     int64 // Total number of connections closed due to idle count.
    maxIdleTimeClosed int64 // Total number of connections closed due to idle time.
    maxLifetimeClosed int64 // Total number of connections closed due to max connection lifetime limit.

    stop func() // stop cancels the connection opener.
}
```

三、获取连接

我们实际执行一个查询：

```
rows, err := db.Query("select * from test")
```

1. 调用下面的Query方法：

```
func (db *DB) QueryContext(ctx context.Context, query string, args ...interface{}) (*Rows, error) {
    var rows *Rows
    var err error
    // 重试机制
    for i := 0; i < maxBadConnRetries; i++ {
        // 这里默认采用cachedOrNewConn策略去获取conn
        rows, err = db.query(ctx, query, args, cachedOrNewConn)
        if err != driver.ErrBadConn {
            break
        }
    }
    if err == driver.ErrBadConn {
        // 如果获取连接失败，就采用alwaysNewConn的策略去获取新的连接
        return db.query(ctx, query, args, alwaysNewConn)
    }
    return rows, err
}

// Query executes a query that returns rows, typically a SELECT.
// The args are for any placeholder parameters in the query.
```

```
//
// Query uses context.Background internally; to specify the context, use
// QueryContext.
// 实际执行时调用这个方法
func (db *DB) Query(query string, args ...interface{}) (*Rows, error) {
    return db.QueryContext(context.Background(), query, args...)
}

func (db *DB) query(ctx context.Context, query string, args []interface{}, strategy connReuseStrategy) (*Rows
    // 获取一个链接
    dc, err := db.conn(ctx, strategy)
    if err != nil {
        return nil, err
    }

    // 用获取到的conn做实际的数据库操作
    return db.queryDC(ctx, nil, dc, dc.releaseConn, query, args)
}
```

2. 调用conn方法获取一个conn

```
// conn returns a newly-opened or cached *driverConn.
func (db *DB) conn(ctx context.Context, strategy connReuseStrategy) (*driverConn, error) {
    db.mu.Lock()
    if db.closed {
        db.mu.Unlock()
        return nil, errDBClosed
    }
    // Check if the context is expired.
    select {
    default:
    case <-ctx.Done():
        db.mu.Unlock()
        return nil, ctx.Err()
    }
    lifetime := db.maxLifetime

    // Prefer a free connection, if possible.
    // 从freeConn取一个空闲连接
    numFree := len(db.freeConn)
    if strategy == cachedOrNewConn && numFree > 0 {
        conn := db.freeConn[0]
        copy(db.freeConn, db.freeConn[1:])
        db.freeConn = db.freeConn[:numFree-1]
        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); err == driver.ErrBadConn {
            conn.Close()
            return nil, driver.ErrBadConn
        }

        return conn, nil
    }

    // Out of free connections or we were asked not to use one. If we're not
    // allowed to open any more connections, make a request and wait.
    // 如果建立的连接已经达到最大连接限制，加入等待队列
    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()

        waitStart := nowFunc()

        // Timeout the connection request with the context.
        // 阻塞在这里直到ctx.Done()或者有可用连接
        select {
        case <-ctx.Done():
            // Remove the connection request and ensure no value has been sent
            // on it after removing.
            db.mu.Lock()
            delete(db.connRequests, reqKey)
            db.mu.Unlock()

            atomic.AddInt64(&db.waitDuration, int64(time.Since(waitStart)))

            select {
            default:
            case ret, ok := <-req:
                if ok && ret.conn != nil {
                    db.putConn(ret.conn, ret.err, false)
                }
            }
            return nil, ctx.Err()
        case ret, ok := <-req:
            atomic.AddInt64(&db.waitDuration, int64(time.Since(waitStart)))

            if !ok {
                return nil, errDBClosed
            }
            // Only check if the connection is expired if the strategy is cachedOrNewConns.
            // If we require a new connection, just re-use the connection without looking
            // at the expiry time. If it is expired, it will be checked when it is placed
            // back into the connection pool.
            // This prioritizes giving a valid connection to a client over the exact connection
            // lifetime, which could expire exactly after this point anyway.
            if strategy == cachedOrNewConn && ret.err == nil && ret.conn.expired(lifetime) {
                db.mu.Lock()
                db.maxLifetimeClosed++
                db.mu.Unlock()
                ret.conn.Close()
                return nil, driver.ErrBadConn
            }
            if ret.conn == nil {
                return nil, ret.err
            }

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

    db.numOpen++ // optimistically
    db.mu.Unlock()
    // 直接获取一个连接
    ci, err := db.connector.Connect(ctx)
    if err != nil {
        db.mu.Lock()
        db.numOpen-- // correct for earlier optimism
        // 这里是向DB的openerCh通道丢一个空的结构体，让监听该通道的goroutine去获取新的连接
        db.maybeOpenNewConnections()
        db.mu.Unlock()
        return nil, err
    }
    db.mu.Lock()
    dc := &driverConn{
        db:         db,
        createdAt:  nowFunc(),
        returnedAt: nowFunc(),
        ci:         ci,
        inUse:      true,
    }
    db.addDepLocked(dc, dc)
    db.mu.Unlock()
    return dc, nil
}
```

总结一下连接获取的步骤：

1. 如果获取连接的策略为cachedOrNewConn且freeConn里面有空闲连接，直接返回拿到的conn
2. 如果已经建立的连接数达到最大上限且无空闲连接可用，则将请求加入db.connRequests等待队列，然后阻塞等到有连接可用时，这里拿到的是释放的连接，检查可用后返回
3. 如果不满足以上情况，则直接新建连接，创建成功则直接返回conn，失败则向DB的openerCh通道丢一个空的结构体，让监听该通道的goroutine去获取新的连接

四、释放连接

数据库连接在被使用完后需要释放，归还给连接池，以供后续请求复用

```
// putConn adds a connection to the db's free pool.
// err is optionally the last error that occurred on this connection.
func (db *DB) putConn(dc *driverConn, err error, resetSession bool) {
    if err != driver.ErrBadConn {
        if !dc.validateConnection(resetSession) {
            err = driver.ErrBadConn
        }
    }
    db.mu.Lock()
    if !dc.inUse {
        db.mu.Unlock()
        if debugGetPut {
            fmt.Printf("putConn(%v) DUPLICATE was: %s\n\nPREVIOUS was: %s", dc, stack(), db.lastPut[dc])
        }
        panic("sql: connection returned that was never out")
    }

    if err != driver.ErrBadConn && dc.expired(db.maxLifetime) {
        db.maxLifetimeClosed++
        err = driver.ErrBadConn
    }
    if debugGetPut {
        db.lastPut[dc] = stack()
    }
    dc.inUse = false
    dc.returnedAt = nowFunc()

    for _, fn := range dc.onPut {
        fn()
    }
    dc.onPut = nil

    if err == driver.ErrBadConn {
        // Don't reuse bad connections.
        // Since the conn is considered bad and is being discarded, treat it
        // as closed. Don't decrement the open count here, finalClose will
        // take care of that.
        db.maybeOpenNewConnections()
        db.mu.Unlock()
        dc.Close()
        return
    }
    if putConnHook != nil {
        putConnHook(db, dc)
    }
    // 归还连接
    added := db.putConnDBLocked(dc, nil)
    db.mu.Unlock()
    if !added {
        dc.Close()
        return
    }
}

func (db *DB) putConnDBLocked(dc *driverConn, err error) bool {
   if db.closed {
    return false
   }
   if db.maxOpen > 0 && db.numOpen > db.maxOpen {
    return false
   }
   // 如果等待队列有等待连接的请求，则将连接发放给他们，否则放回freeConn
   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
}
```