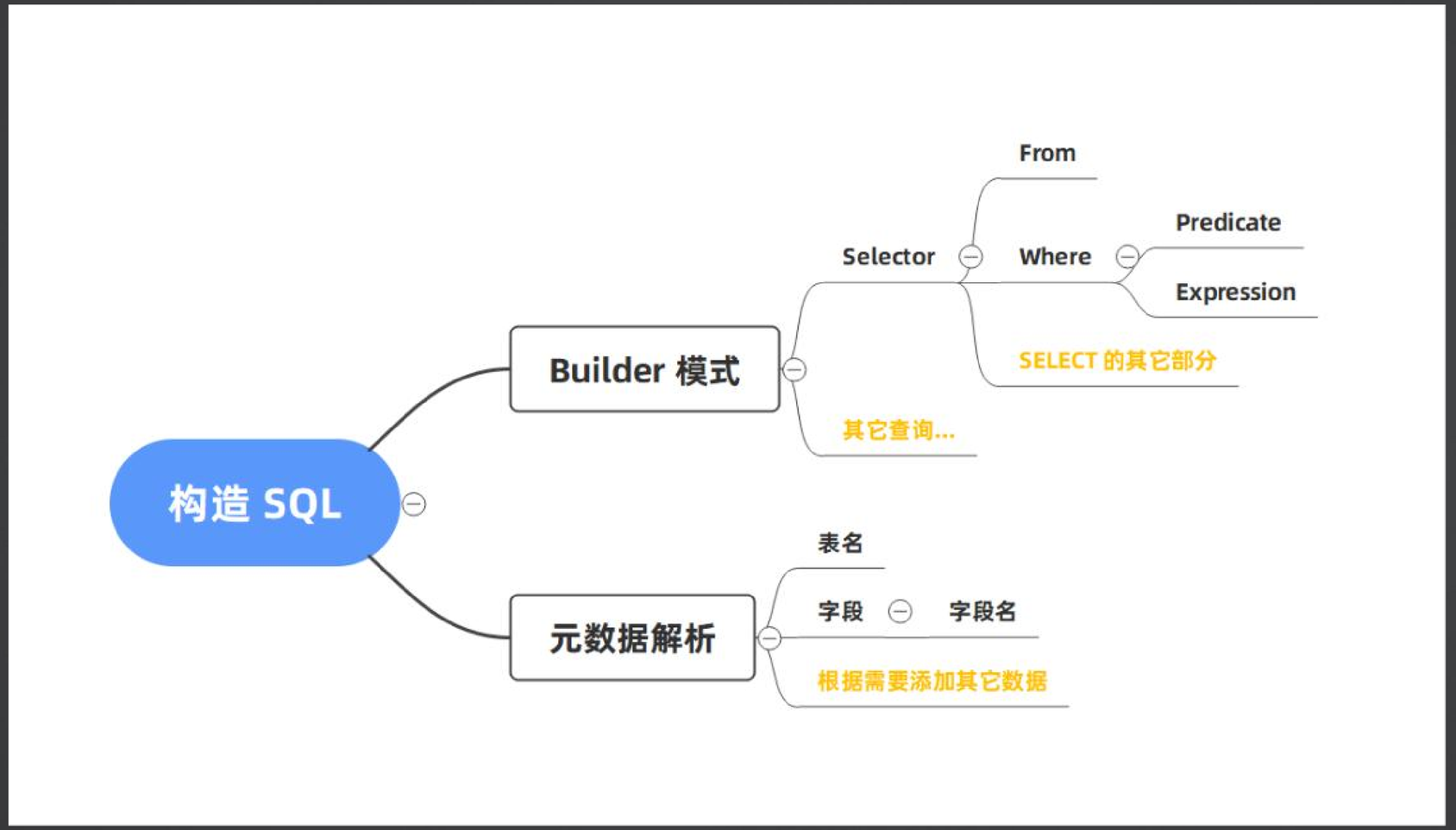
Go 实战训练营

极简版 ORM 框架设计与实现（下）

大明

示例代码：https://github.com/flycash/toy-orm
准备带着学员做的开源项目（完成了主体）：https://github.com/gotomicro/eorm

上节课回顾


构造 SQL
Builder 模式
Selector
From
Where
Predicate
Expression
SELECT 的其它部分
其它查询...
元数据解析
表名
字段
字段名
根据需要添加其它数据

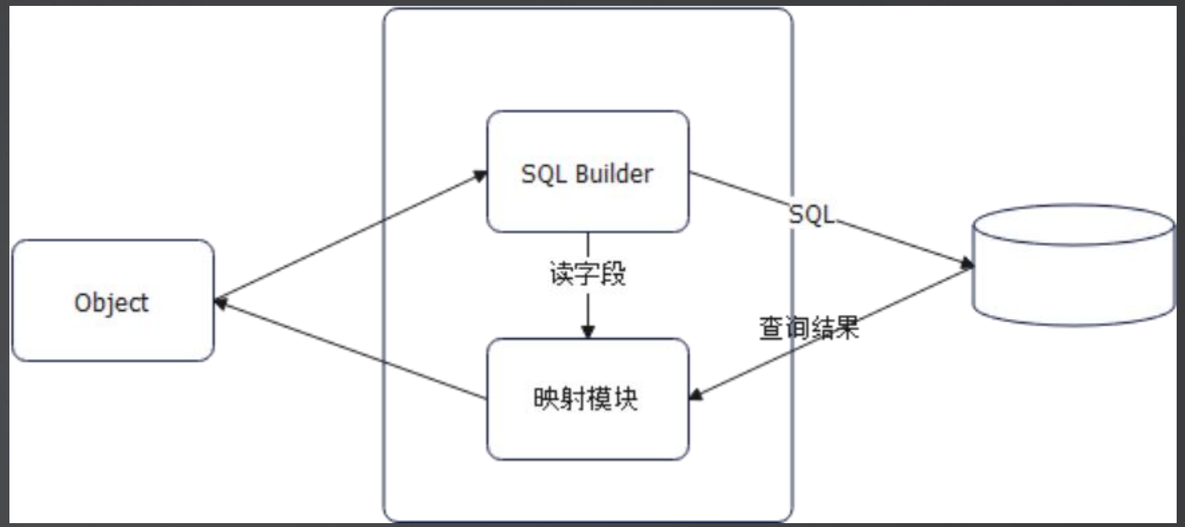
ORM SELECT —— 发起查询

Selector 到这一步，完成了：

- SQL 构建
- DB 构建
- 元数据定义

还剩下两个关键步骤：

- 执行查询
- 处理结果集



ORM SELECT —— 发起查询

极客时间 训练营

```go
type DB struct {
    db *sql.DB
    r  *registry
}

func NewDB(driver string, dsn string, opts
    db, err := sql.Open(driver, dsn)
    if err != nil {
        return nil, err
    }
    res := &DB{
        db: db,
        r:  &registry{},
    }
    for _, o := range opts {
        o(res)
```

```go
func (s *Selector[T]) Get(ctx context.Context) (*
    q, err := s.Build()
    if err != nil {
        return nil, err
    }
    row := s.db.db.QueryRow(q.SQL, q.Args...)
    if row.Err() != nil {
        return nil, row.Err()
    }

    t := new(T)
    // 处理结果集，也就是构造 T

    return t, nil
}
```

ORM 处理结果集 —— Row 和 Rows

Row：可以理解为只有一行的 Rows，而且是必须要有一行。没有的话，在调用 Row 的 Scan 的时候会返回 sql.ErrNoRow。

```
    row := db.QueryRowContext(context.Background(), query:"SELEC
    if row.Err() != nil {
        t.Fatal(row.Err())
    }
    tm := &TestModel{}
    err = row.Scan(&tm.Id, &tm.FirstName, &tm.Age, &tm.LastName)
    if err != nil {
        t.Fatal(err)
    }
    assert.Equal(t, expected:"changed", tm.FirstName)
```

ORM 处理结果集 —— Row 和 Rows

极客时间 训练营

Rows：

- 迭代器设计，需要在使用前调用 Next 方法
- Scan 支持的类型很多：如果拿捏不准用什么，就用 interface{} 的指针来作为接收器

```
for rows.Next() { 标准迭代器式设计
    tm := &TestModel{}
    err = rows.Scan(&tm.Id, &tm.FirstName, &tm.Age, &tm.LastName)
    if err != nil {            注意要用指针
        t.Fatal(err)
    }
    assert.Equal(t, expected: "Tom", tm.FirstName)
}
```

```
//    *string
//    *[]byte
//    *int, *int8, *int16, *int32, *int64
//    *uint, *uint8, *uint16, *uint32, *uint64
//    *bool
//    *float32, *float64
//    *interface{}
//    *RawBytes
//    *Rows (cursor value)
//    any type implementing Scanner (see Scanner docs)
//
```

ORM 处理结果集 —— sqlmock

在单元测试里面我们不希望依赖于真实的数据库，因为数据难以模拟，而且 error 更加难以模拟，所以我们采用 sqlmock 来做单元测试。

sqlmock 使用：

- 初始化：返回一个 mockDB，类型是 *sql.DB。还有 mock 用于构造模拟的场景；
- 设置 mock：基本上是 ExpectXXX WillXXX。严格依赖于顺序。

```go
func TestDB_BeginTx(t *testing.T) {
    mockDB, mock, err := sqlmock.New()
    if err != nil {
        t.Fatal(err)
    }
    defer func() { _ = mockDB.Close() }()

    db, err := openDB( driver: "mysql", mockDB)
    if err != nil {
        t.Fatal(err)
    }
    // Begin 失败
    mock.ExpectBegin().WillReturnError(errors.New( text: "begin failed"))
    tx, err := db.BeginTx(context.Background(), &sql.TxOptions{})
    assert.Equal(t, errors.New( text: "begin failed"), err)
    assert.Nil(t, tx)

    mock.ExpectBegin()
    tx, err = db.BeginTx(context.Background(), &sql.TxOptions{})
    assert.Nil(t, err)
    assert.NotNil(t, tx)
}
```

初始化

设置 mock

ORM 处理结果集 —— 简单的异常场景

极客时间 训练营

```
{
    // 查询返回错误
    name:    "query error",
    mockErr: errors.New(text: "invalid query"),
    wantErr: errors.New(text: "invalid query"),
    query:   "SELECT .*",
},
{
    name:     "no row",
    wantErr:  errors.New(text: "toy-orm: 未找到数据"),
    query:    "SELECT .*",
    mockRows: sqlmock.NewRows([]string{"id"}),
},
{
    name:    "too many column",
    wantErr: errors.New(text: "toy-orm: 列过多"),
    query:   "SELECT .*",
    mockRows: func() *sqlmock.Rows {
        res := sqlmock.NewRows([]string{"id", "first_name"
        res.AddRow([]byte("1"), []byte("Da"), []byte("18")
        return res
```

mock 采用正则表达式匹配

```
func (s *Selector[T]) Get(ctx context.Context) (*T, error) {
    q, err := s.Build()
    if err != nil {
        return nil, err
    }
    rows, err := s.db.db.QueryContext(ctx, q.SQL, q.Args...)
    if err != nil {
        return nil, err
    }
    if !rows.Next() {
        return nil, errors.New(text: "toy-orm: 未找到数据")
    }

    tp := new(T)
    meta, err := s.db.r.get(tp)
    if err != nil {
        return nil, err
    }
    cs, err := rows.Columns()
    if err != nil {
        return nil, err
    }
    if len(cs) > len(meta.fieldMap) {
        return nil, errors.New(text: "toy-orm: 列过多")
```

ORM 处理结果集 —— 构造结构体

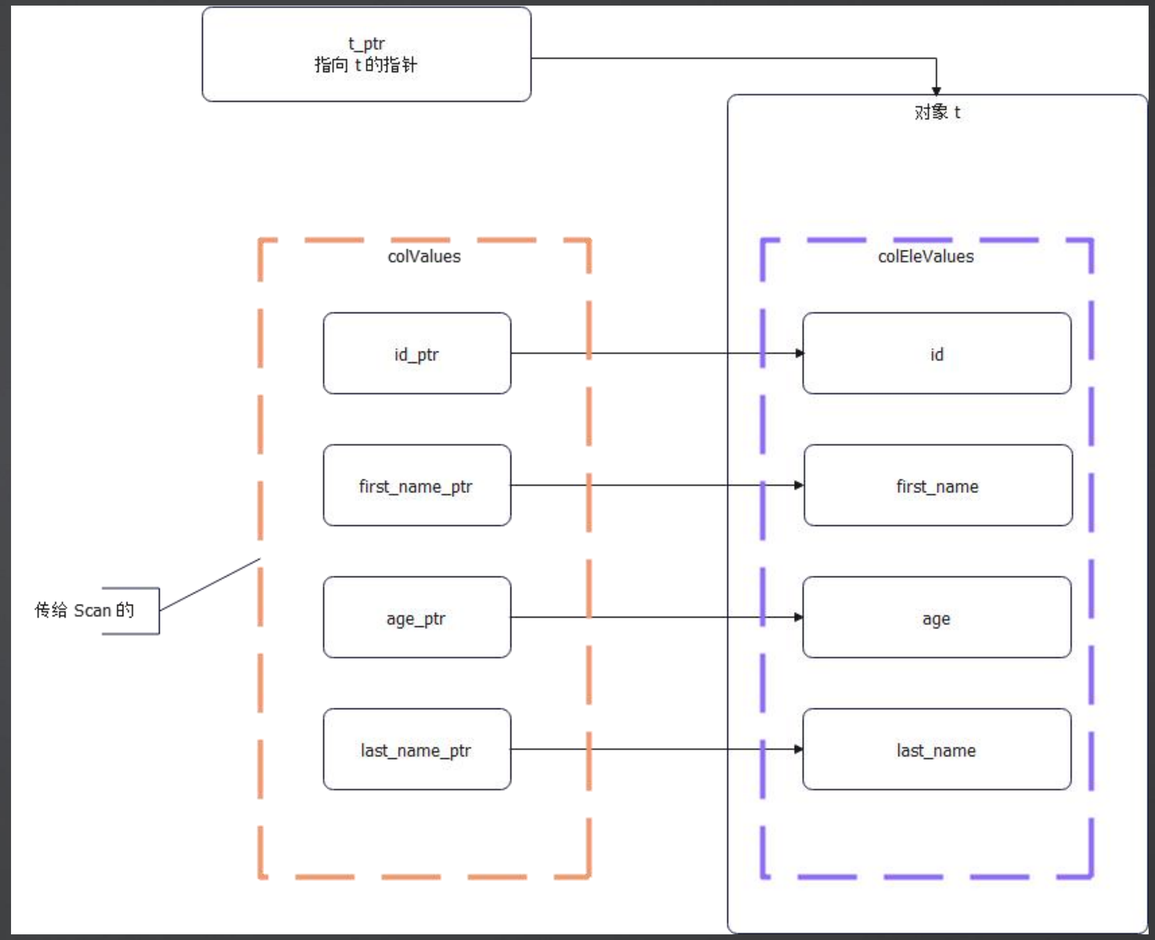
极客时间 训练营

```
{
    name:  "get data",
    query: "SELECT .*",
    mockRows: func() *sqlmock.Rows {
        res := sqlmock.NewRows([]string{"id", "first_name", "age", "last_name"})
        res.AddRow([]byte("1"), []byte("Da"), []byte("18"), []byte("Ming"))
        return res
    }(),
    wantVal: &TestModel{
        Id:        1,
        FirstName: "Da",
        Age:       18,
        LastName:  &sql.NullString{String: "Ming", Valid: true},
    },
}
```

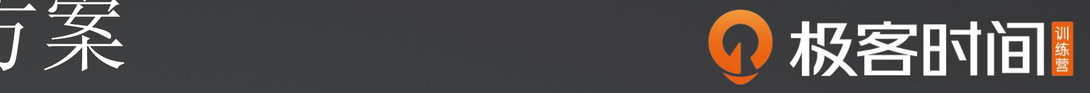
```
// colValues 和 colEleValues 实质上最终都指向同一个对象
colValues := make([]interface{}, len(cs))
colEleValues := make([]reflect.Value, len(cs))
for i, c := range cs {
    cm, ok := meta.columnMap[c]
    if !ok {
        return nil, fmt.Errorf( format: "toy-orm: 非法列名 %s", c)
    }
    val := reflect.New(cm.typ)
    colValues[i] = val.Interface()
    colEleValues[i] = val.Elem()
}
if err = rows.Scan(colValues...); err != nil {
    return nil, err
}

val := reflect.ValueOf(tp).Elem()
for i, c := range cs {
    cm := meta.columnMap[c]
    fd := val.FieldByName(cm.fieldName)
    fd.Set(colEleValues[i])
}
return tp, nil
```

ORM 处理结果集 —— 构造结构体

```go
// colValues 和 colEleValues 实质上最终都指向同一个对象
colValues := make([]interface{}, len(cs))
colEleValues := make([]reflect.Value, len(cs))
for i, c := range cs {
    cm, ok := meta.columnMap[c]
    if !ok {
        return nil, fmt.Errorf(format: "toy-orm: 非法列名 %s", c)
    }
    val := reflect.New(cm.typ)
    colValues[i] = val.Interface()
    colEleValues[i] = val.Elem()
}
if err = rows.Scan(colValues...); err != nil {
    return nil, err
}

val := reflect.ValueOf(tp).Elem()
for i, c := range cs {
    cm := meta.columnMap[c]
    fd := val.FieldByName(cm.fieldName)
    fd.Set(colEleValues[i])
}
return tp, nil
```



ORM 处理结果集 —— unsafe 方案



除了使用反射，还可以使用 unsafe 来构造结构体：

- 计算字段偏移量
- 计算对象起始地址
- 字段真实地址=对象起始地址 + 字段偏移量
- reflect.NewAt 在特定地址创建对象
- 调用 Scan

```
    cs, err := rows.Columns()
    if err != nil {
        return err
    }
    if len(cs) > len(u.meta.Columns) {
        return errs.ErrTooManyColumns
    }

    colValues := make([]interface{}, len(cs))
    for i, c := range cs {
        cm, ok := u.meta.ColumnMap[c]
        if !ok : errs.NewInvalidColumnError(c) ↵
        ptr := unsafe.Pointer(uintptr(u.addr) + cm.Offset)
        val := reflect.NewAt(cm.Typ, ptr)
        colValues[i]=val.Interface()
    }
    return rows.Scan(colValues...)
```

ORM 处理结果集 —— 性能比较

极客时间 训练营

```
 + 6dd36e2...dbb3895 main -> main (forced update)
➜  eorm git:(main) go test -bench=BenchmarkQuerier_Get -benchmem -benchtime=10000x
2022/06/25 00:53:35 ExpectPing will have no effect as monitoring pings is disabled. Use MonitorPingsOption to enable.
goos: linux
goarch: amd64
pkg: github.com/gotomicro/eorm
cpu: Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz
BenchmarkQuerier_Get/unsafe-12              10000            424128 ns/op            3847 B/op        116 allocs/op
BenchmarkQuerier_Get/reflect-12             10000           1226397 ns/op            4028 B/op        125 allocs/op
PASS
ok      github.com/gotomicro/eorm       16.563s
```

代码在 eorm：https://github.com/gotomicro/eorm

ORM SELECT 进一步提供更多功能

极客时间 训练营

- 指定列：普通列、聚合函数
- Order By、Group By、Offset、Limit、Having
- MultiGet：在 Get 的基础上，引入批量查询

有时间的同学可以尝试，思路都是类似的。

ORM INSERT —— 构造查询

极客时间 训练营

最基本的 INSERT 语句包含：

- 插入的列名
- 插入的数据

如果考虑 UPSERT 语句，那么就还有 ON DUPLICATE KEY UPDATE 部分。

剩下的语法，可支持也可不支持。

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    { {VALUES | VALUE} (value_list) [, (value_list)] ... }
    [AS row_alias[(col_alias [, col_alias] ...)]]
    [ON DUPLICATE KEY UPDATE assignment_list]

INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    SET assignment_list
    [AS row_alias[(col_alias [, col_alias] ...)]]
    [ON DUPLICATE KEY UPDATE assignment_list]

INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    { SELECT ...
      | TABLE table_name
      | VALUES row_constructor_list
    }
    [ON DUPLICATE KEY UPDATE assignment_list]

value:
```

ORM INSERT —— Beego ORM 设计

特点：

- 单个插入和批次插入是分离的
- 批量插入的时候可以控制每一批次的数量

```
// insert model data to database
// for example:
//  user := new(User)
//  id, err = Ormer.Insert(user)
//  user must be a pointer and Insert will set user's pk field
Insert(md interface{}) (int64, error)
InsertWithCtx(ctx context.Context, md interface{}) (int64, error)
// mysql:InsertOrUpdate(model) or InsertOrUpdate(model,"colu=colu+value")
// if colu type is integer : can use(+-*/), string : convert(colu,"value")
// postgres: InsertOrUpdate(model,"conflictColumnName") or InsertOrUpdate(model,"c
// if colu type is integer : can use(+-*/), string : colu || "value"
InsertOrUpdate(md interface{}, colConflitAndArgs ...string) (int64, error)
InsertOrUpdateWithCtx(ctx context.Context, md interface{}, colConflitAndArgs ...st
// insert some models to database
InsertMulti(bulk int, mds interface{}) (int64, error)
InsertMultiWithCtx(ctx context.Context, bulk int, mds interface{}) (int64, error)
```

ORM INSERT —— GORM 设计

特点：

- value 可以是单个，也可以是多个
- CreateInBatches 是辅助用户分批次插入

```go
// Create insert the value into database
func (db *DB) Create(value interface{}) (tx *DB) {
    if db.CreateBatchSize > 0 {
        return db.CreateInBatches(value, db.CreateBatchS
    }
```

```go
// CreateInBatches insert the value in batches into database
func (db *DB) CreateInBatches(value interface{}, batchSize int) (tx *DB) {
    reflectValue := reflect.Indirect(reflect.ValueOf(value))
```

ORM INSERT —— 采用的设计

特点：

- 采用泛型约束类型；
- 批量还是非批量完全取决于用户传入的个数；
- 放弃辅助用户切割，例如说 100 个切割成 10 批，每批 10 个。我认为这已经超出了 ORM 要控制的范畴。

```go
type Inserter[T any] struct {
    db     *DB
    values []*T
}
```

```go
func (i *Inserter[T]) Values(vals ...*T) *Inserter[T] {
    i.values = vals
    return i
}
```

ORM INSERT —— 测试用例

极客时间 训练营

```
{
    {
        name:    "no examples of values",
        builder: NewInserter[User](db).Values(),
        wantErr: errors.New( text: "toy-orm: 插入0行"),
    },
    {
        name:     "single example of values",
        builder:  NewInserter[User](db).Values(u),
        wantSql:  "INSERT INTO `user`(`id`,`first_name`,`ctime`) VALUES(?,?,?);",
        wantArgs: []interface{}{int64(12), "Tom", n},
    },

    {
        name:     "multiple values of same type",
        builder:  NewInserter[User](db).Values(u, u1),
        wantSql:  "INSERT INTO `user`(`id`,`first_name`,`ctime`) VALUES(?,?,?),(?,?,?);",
        wantArgs: []interface{}{int64(12), "Tom", n, int64(13), "Jerry", n},
    },
}
```

ORM INSERT —— 实现

极客时间 训练营

```
    var sb strings.Builder
    sb.WriteString( s: "INSERT INTO `")
    sb.WriteString(meta.tableName)
    sb.WriteString( s: "`(")
    for index, fd := range meta.fields {
        if index > 0 {
            sb.WriteByte( c: ',')
        }
        cm, _ := meta.fieldMap[fd]
        sb.WriteByte( c: '`')
        sb.WriteString(cm.columnName)
        sb.WriteByte( c: '`')
    }
    sb.WriteString( s: ")")
    sb.WriteString( s: " VALUES")
    args := make([]any, 0, len(i.values)*len(meta.fields))
    for index, val := range i.values {
        if index > 0 {
            sb.WriteByte( c: ',')
        }
        sb.WriteByte( c: '(')
        refVal := reflect.ValueOf(val).Elem()
        for j, v := range meta.fields {
```

ORM INSERT —— 执行语句

极客时间 训练营

要考虑：

- 要不要把 Id 放回去原本插入的对象里面
- 连续两次错误处理简直过于繁琐

```
}

func (i *Inserter[T]) Exec(ctx context.Context) (sql.Result, error) {
    q, err := i.Build()
    if err != nil {
```

```
    }
    res, err := NewInserter[TestModel](db).Values(&TestModel{FirstName: "Tom"}).
        Exec(context.Background())
    if err != nil {
        t.Fatal(err)
    }
    id, err := res.LastInsertId()
    if err != nil {
        t.Fatal(err)
    }
    assert.True(t, id > 0)
}
```

ORM INSERT —— Result 抽象

极客时间 训练营

```go
func (i *Inserter[T]) Exec(ctx context.Context) sql.Result {
    q, err := i.Build()
    if err != nil {
        return Result{
            err: err,
        }
    }
    res, err := i.db.db.ExecContext(ctx, q.SQL, q.Args...)
    return Result{
        err: err,
        res: res,
    }
}
```

```go
type Result struct {
    err error
    res sql.Result
}

func (r Result) LastInsertId() (int64, error) {
    if r.err != nil {
        return 0, r.err
    }
    return r.res.LastInsertId()
}

func (r Result) RowsAffected() (int64, error) {
    if r.err != nil {
        return 0, r.err
    }
    return r.res.RowsAffected()
}
```

ORM INSERT —— Result 抽象

极客时间 训练营

```
    }
    res, err := NewInserter[TestModel](db).Values(&TestModel{FirstName: "Tom"}).
        Exec(context.Background())
    if err != nil {
        t.Fatal(err)
    }
    id, err := res.LastInsertId()
    if err != nil {
        t.Fatal(err)
    }
    assert.True(t, id > 0)
}
```

```
res := NewInserter[TestModel](db).V
    Exec(context.Background())
id, err := res.LastInsertId()

if err != nil {
    t.Fatal(err)
}
assert.True(t, id > 0)
```

简化了操作

ORM 其余语句，或者更多语句特性

极客时间 训练营

自己动手，丰衣足食

思路都是一样的，课程时间有限，就只能大家课后自己尝试~

ORM 事务 —— Beego ORM 设计

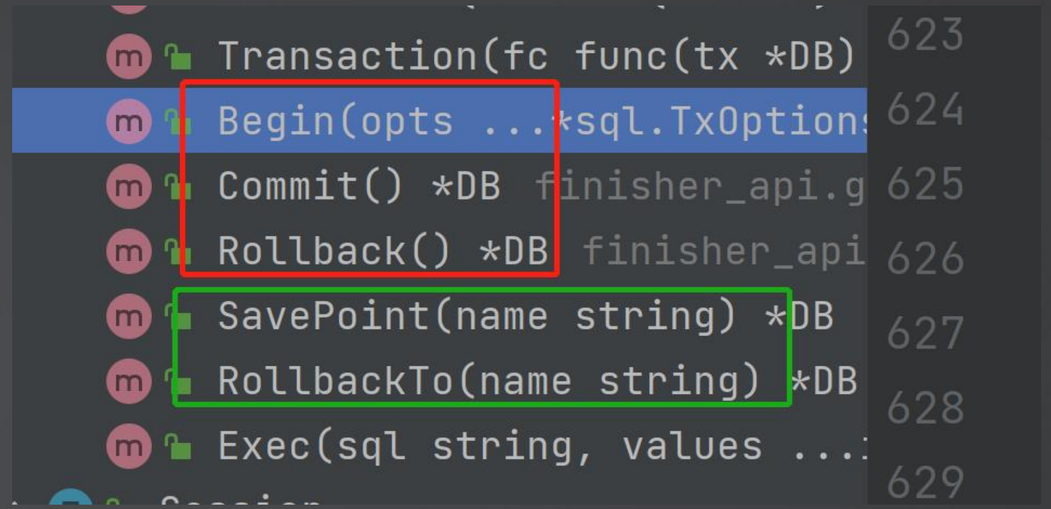
极客时间 训练营

```go
type TxBeginner interface {
    // self control transaction
    Begin() (TxOrmer, error)
    BeginWithCtx(ctx context.Context) (TxOrmer, error)
    BeginWithOpts(opts *sql.TxOptions) (TxOrmer, error)
    BeginWithCtxAndOpts(ctx context.Context, opts *sql.Tx

    // closure control transaction
    DoTx(task func(ctx context.Context, txOrm TxOrmer) er
    DoTxWithCtx(ctx context.Context, task func(ctx contex
    DoTxWithOpts(opts *sql.TxOptions, task func(ctx conte
    DoTxWithCtxAndOpts(ctx context.Context, opts *sql.Tx(
}
```

```go
// transaction ending
type txEnder interface {
    Commit() error
    Rollback() error

    // RollbackUnlessCommit if the transac
    // For example:
    // ```go
    //    txOrm := orm.Begin()
    //    defer txOrm.RollbackUnlessCommit
    //    err := txOrm.Insert() // do some
    //    if err != nil {
    //        return err
    //    }
    //    txOrm.Commit()
    // ```
    RollbackUnlessCommit() error
}
```

ORM 事务 —— GORM 设计

- DB 本身也可以被看做是事务
- 普通的事务开启、提交和回滚功能
- 额外实现了一个 SavePoint 的功能
- 事务闭包 API



ORM 事务

总结：

- 开启事务
- 回滚或者提交事务
- 闭包 API
- 不准备支持 SavePoint 的功能

```go
func (db *DB) Begin(ctx context.Context, opts *sql.TxOptions) (*Tx, error)
    tx, err := db.db.BeginTx(ctx, opts)
    if err != nil {
        return nil, err
    }
    return &Tx{
        r:  db.r,
        tx: tx,
    }, nil
}
```

```go
type Tx struct {
    tx *sql.Tx
    r  *registry
}

func (t *Tx) Commit() error {
    return t.tx.Commit()
}

func (t *Tx) Rollback() error {
    return t.tx.Rollback()
}
```

问题：无法用来构造 Selector

极客时间 训练营

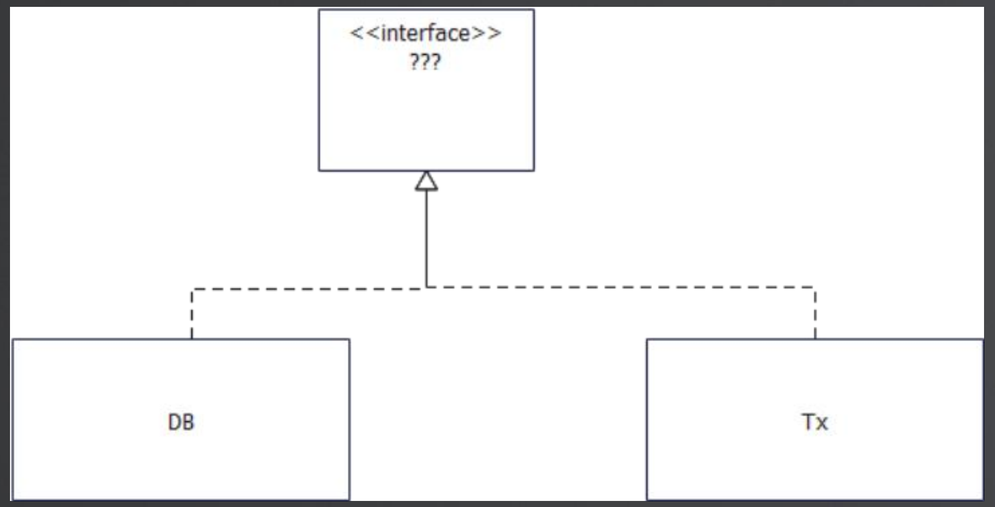
```
}
tx, err := db.Begin(context.Background(), &sql.TxOptions{})
if err != nil {
    t.Fatal()
}

NewSelector[TestModel](tx)
```

编译错误

```
func NewSelector[T any](db *DB) *Selector[T] {
    return &Selector[T]{
        db: db,
    }
}
```

ORM Session 抽象

我们需要一个 DB 和 Tx 的公共抽象。




```go
func (s *Selector[T]) Get(ctx context.Context) (*T, error) {
    q, err := s.Build()
    if err != nil { return nil, err }
    rows, err := s.db.db.QueryContext(ctx, q.SQL, q.Args...)
    if err != nil { return nil, err }
    if !rows.Next() {
        return nil, errors.New( text: "tov-orm: 未找到数据")
```

```go
    }
    res, err := i.db.db.ExecContext(ctx, q.SQL, q.Args...)
    return Result{
        err: err,
        res: res,
    }
```

ORM Session 抽象

Session 在 Web 里面有比较特殊的含义。

在 ORM 的语境下，一般代表一个上下文；也可以理解为一种分组机制，在这个分组内所有的查询会共享一些基本的配置。

```go
type Session interface {
    query(ctx context.Context, sql string, args ...any) (*sql.Rows, error)
    exec(ctx context.Context, sql string, args ...any) (sql.Result, error)
    registry() *registry
}
```

```go
func (t *Tx) query(ctx c
    return t.tx.QueryCor
}

func (t *Tx) exec(ctx cc
    return t.tx.ExecCont
}

func (t *Tx) registry()
    return t.r
}
```

```go
func (db *DB) query(ctx context.Context, sql st
    return db.db.QueryContext(ctx, sql, args...
}

func (db *DB) exec(ctx context.Context, sql str
    return db.db.ExecContext(ctx, sql, args...)
}

func (db *DB) registry() *registry {
    return db.r
}
```

ORM Session 重构 Selector

极客时间 训练营

```go
type Selector[T any] struct {
    sess Session
    sb   strings.Builder
    args []any
    mi   *ModelInfo

    tbl   string
    where []Predicate
}
```

```go
func (s *Selector[T]) Get(ctx context.Context) (*T, error) {
    q, err := s.Build()
    if err != nil { return nil, err }
    rows, err := s.sess.query(ctx, q.SQL, q.Args...)
    if err != nil { return nil, err }
    if !rows.Next() {
        return nil, errors.New( text: "toy-orm: 未找到数据")
    }

    tp := new(T)
    meta, err := s.sess.registry().get(tp)
    if err != nil { return nil, err }
    cs, err := rows.Columns()
    if err != nil { return nil, err }
    if len(cs) > len(meta.fieldMap) { return nil, errors.New("toy-orm: 列过多") }

    // TODO 性能优化
```

Q & A

极客时间 训练营

体验课：Go 工程师进阶 14 讲

◆ 课程安排 ◆

方法论

- 学好 Go，这些方法你必须知道
- Go 工程师必备技能与进阶指南

Go 架构实践 – 微服务

- Proxyless service mesh 演进
- 微服务的 DDD 实践
- 微服务框架解析
- 如何做好 DDD 战术设计
- 缓存一致性

Go 工程化实践

- 谈谈 Go 工程化：
  - Functional options and config for APIs
  - 怎么做好配置管理?

Go 高并发实践

- Go 并发编程模式及最佳实践
- Golang 在高并发场景下的技术实践

API 网关

- Go 语言微服务实战之 API 网关
- Go 项目实战之 API 设计
- Golang 实战 API 管理的高效玩法

扫码填写试讲调研问卷

联系学习助理免费领