副本 生成 INSERT 语句

利用反射，将一个结构体实例转 `sertStmt(User{})` 有的单元测试。在这个作业里面，你只需要考虑生成一个能在 MySQL 上执行的语句。我们已经预先定义好了方法：

```
1 func InsertStmt(entity interface{}) (string, []interface{}, error) { }
```

为了减轻一些工作量，也为了便于维护，所以输入你只需要考虑有限的几种情况：

- nil
- 结构体
- 一级指针，但是指针指向的是结构体
- 组合，但是不能是指针形态的组合

实际上，在考虑到执行 INSERT 语句，数据库驱动只支持特定的一些类型，包括：

- 基本类型
- string, []byte 和 time.Time
- 实现了 driver.Valuer 和 sql.Scanner 两个接口的类型

具体可以参考测试用例。

从实现上来说，就是简单的遍历所有的字段，并且检查每个字段：

- 如果字段是普通的字段，那么直接进行处理，也就是认为它是一个列。它的名字作为列名，而对应的值作为插入时候的参数；
- 如果字段其实代表的是组合。注意，在 Go 里面，所谓组合其实就是一个比较特殊的字段，在课堂上演示过，所谓的匿名字段。如果字段类型此时是结构体，那么应该递归进去解析这个结构体的每一个字段；

> 以下文档是一个很全面的考虑，但是我们作业是极简版，所以简单定义一个 InsertStmt 方法就可以，主要还是以语法练习为主。
> 实际上在我们后面构造 ORM 框架的时候，采用的才是下面文档的设计。下面设计的核心要素是使用 Builder 模式，和我们作业的单一方法比起来要复杂很多，大概看一下就可以，ORM 模块的时候我们会详细分析这个文档。
> 另外你们也可以注意到，实际上在技术文档里面，把调研（也就是需求分析那一块）做好之后，其实设计是比较简单的事情。大多数同学写不出代码或者不知道怎么解决问题，其实都是卡在了搞清楚需求这一步。

背景

当我们希望把一个类型实例的数据存储进去数据库的时候，就必然面临着一个问题：如何将实例转化为一个 INSERT 语句，并且生成插入语句的参数。

在实际业务中，有很多种情况是需要考虑进去的：

- 批量插入
- 指定列，以及指定列的表达式
- 组合：一般的公司会考虑定义一些公共的结构体，要求所有的业务方必须组合这些结构体
- upsert：即所谓的 INSERT or UPDATE 语义，常见的场景则是数据不存在则插入，数据存在则删除

名词解释

- 多重组合：一个结构体同时组合了多个结构体
- 深层组合：一个结构体 A 组合了另一个结构体 B，而 B 本身也组合了 C
- 多级指针：指向指针的指针，如 **int
- 方言：SQL虽然有一个标准，但是往往数据库支持的语义可能有所差别，并且有些数据库会有自己独特的语法，因此不同数据库支持的 SQL 我们称之为方言
- upsert：即所谓的 INSERT or UPDATE语义，如果数据不存在，则插入；如果数据存在，则更新。存在与否的判断，一般是通过主键或者唯一索引来进行

需求分析

场景分析

这里基本上就是考虑三个因素：

- 开发者会如何构建输入。核心就是他们会使用指针还是非指针；
- 开发者如何定义模型。关键是他们是否会使用组合，以及如何使用组合；
- 开发者使用什么类型来定义字段。关键在于开发者如果用到了复杂结构体来作为字段类型，该如何处理；
- 开发者使用批量插入
- 开发者如何指定列，以及列的表达式
- 开发者希望执行 upsert。并且需要额外考虑以下情况
  - 指定冲突的列或者索引（在 PostgreSQL 或者 SQLite3）上
  - 指定更新的列，以及对应的值

场景	例子	说明
使用结构体作为输入	InsertStmt(User{})	能够正常处理
使用指针作为输入	InsertStmt(&User{})	能够正常处理
使用多级指针作为输入	InsertStmt(&&User{})	返回错误，不需要支持。很显然，正常情况下，开发者没有道理使用这种作为输入
使用 nil 作为输入	InsertStmt(nil)	不需要支持，这只能是开发者偶发性错误，没有在调用之前检测是否为 nil。返回错误会比较合适
使用基本类型或者内置类型作为输入	InsertStmt([]int)	不需要支持。正常的开发者都不会尝试这样使用，返回错误就可以

使用了组合	type Buyer struct {
	User
	}
		支持。这是常见用法。尤其是在一些公司有内部规范的情况下，例如部分公司可能会要求所有的模型都必须强制组合

		一个 BaseEntity，这个结构体里面包含基本的 Id, CreateTime, UpdateTime 等常见字段。在这种情况下，User 里面的字段会被认为是 Buyer 的一部分。因此生成的 SQL 里面也要包含这些字段

使用了组合，但是组合是指针形态	type Buyer struct {
	*User
	}
		不需要支持。大多数情况下，组合都会采用非指针的形态，尤其是在数据库模型定义这个层面上，这种定义方式并没有明显的优势，也找不出一个非它不可的场景。
		
		另外一个理由是，大多数时候，如果追求性能，我们会尝试用 unsafe 来取代反射操作，但是 unsafe 对指针类型的组合优势不大

使用了复杂结构体作为字段	type Buyer struct {
	User User
	}
		不做校验。严格来讲，这种定义方式和组合定义方式，在语义上是有区别的。组合意味着同一张表，这种更加接近关联关系。User 整体会被认为是一个参数，但是实际上这个参数在 driver 执行的时候会出错。不过我们并不会对这个执行校验。
		理由非常简单：driver 已经提供了这种保障。开发者应该知道 driver 支持什么，不支持什么。

使用了内置类型作为字段类型		同上
使用了 time.Time 作为字段		同上
使用了 driver.Valuer 作为字段类型		

使用了组合，但是有同名字段	type User struct {
	Id int64
	}
	
	type Buyer struct {
	Id int64
	User
	}
		User 和 Buyer 都定义了自己的 Id 字段，这种情况下只需要取一个，可以按照谁先就用谁的值的原则来构建 SQL。

批量插入	InsertBatch(u1, u2...)	也就是意味着用户可以在一个 SQL 语句里面插入多行。 插入多行需要注意一点，就是说有的插入的数据需要有同样的类型。
批量插入，但是不同类型	u := &User{} o := &Order{} InsertBatch(u, o)	返回错误

批量插入，不同类型但是字段完全相同	u1 := &UserV1{} u2 := &UserV2{} InsertBatch(u1, u2)	假定说 UserV1 和 UserV2 两个类型的字段一模一样，在这种场景下，依旧返回错误。 原则上来说，这种是可以正确生成 SQL 的，但是在用户层面上，他们不应该这么使用
批量插入，但是多批次	InsertBatch(users, 10)	假如说 users 传入了 1000 个实例，而且用户要求 10 个一批，那么 1000 个实例。 类似这种需求，其实不属于 ORM 层面上的需求，应该在应用层面上处理。 对于 ORM 来说，如果要支持该场景，那么需要解决： • 要不要开事务? • 部分批次失败，部分批次成功，怎么办? 所以ORM 框架不应该处理这种情况
插入，同时指定插入的列	Insert(user, ”age”, ”first_name”)	在一些场景之下，用户不希望插入所有的列，而是希望能够只插入部分的列。如果用户指定的列不存在，那么应该返回错误
插入复杂的列表达式	Insert(user, ”age”, ”create_time=now()”)	在插入的时候，用户希望使用 MySQL 函数 now() 来作为插入的值。其生成的语句类似于： INSERT xxx(col, col2) VALUES(val1, now()); 类似于 now() 这种表达式，是跟使用的数据库相关的，所以不需要对表达式的正确性进行校验，用户需要对此负责。
自增主键		用户在插入数据的时候，如果主键有值，那么应该使用主键的值，如果主键没有值，那么应该自增生成一个主键。

自增主键，但是 0 值	type User struct {
	ID int
	}
	
	Insert(&User{})
		在这种情况下，用户使用基本类型来作为主键类型，那么用户在没有设置值的情况下，它的默认值是 0, 0 应该被看做没有设置值，从而触发自增主键。

单个插入，获得自增主键		在单个插入的情况下，我们可以确定无疑获得自增主键，而且是必然正确的主键

批量插入，获得自增主键		用户可能期望，如果他一次性插入 100 条数据，假如说第一条的 ID 是 201，那么下一条是 202,203,204... 实际上，有些数据库它同一批次插入的 ID，都不是连续的 因此实际上在批量插入的时候不需要返回所有的 ID，只需要返回 last_insert_id，用户根据自己的数据库的配置，来计算其它 ID。 关于自增主键 ID 是否一定连续，可以参考 stackoverflow 的讨论
新增或者更新（Upsert）	InsertOrUpdate(users)	用户希望，如果要是数据冲突了（可能是主键冲突，也可能是唯一索引冲突），那么就执行更新。 在不同的方言下，upsert 的写法是不同的。所以需要考虑兼容不同的方言； 具体参考方言分析部分。
INSERT...SELECT		这种用法就是用户插入一个查询返回的数据。 这部分将在子查询部分进一步考虑。

方言分析

此处我们忽略了 Oracle，是因为 Oracle 缺乏开源免费版本，因此对测试非常不友好。

MySQL

MySQL 的 INSERT 总体上有三种形态，在它的文档里面有详细描述。
第一种是：

```
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]
```

比较值得注意的就是它采用了 `ON DUPLICATE KEY UPDATE` 来解决 upsert 的问题。这种形态也是我们最常见的形态。
第二种是：

```
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]
```

和第一种比起来，这里用了 `SET assignment_list` 的语法。
第三种形态是：

```
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]
```

这种形态使用了 SELECT 子句。

ON DUPLICATE KEY UPDATE

在 MySQL 的 `ON DUPLICATE KEY UPDATE` 部分，它后面可以跟着一个 `assinment_list` ，而 `assignment_list` 的定义是：

```
assignment:
    col_name =
          value
        | [row_alias.]col_name
        | [tbl_name.]col_name
        | [row_alias.]col_alias
assignment_list:
    assignment [, assignment] ...
```

关键是 assignment 有四种：

1. value：一个纯粹的值
2. row_alias.col_name：在使用了行别名的时候才会有的形态
3. tbl_name.col_name：指定更新为插入的值
4. row_alias.col_alias：这个和第二种比较像，所不同的是这里使用的是别名

在 ORM 框架中，我们不需要支持 2 和 4，因为在插入部分，我们就不会使用行的别名，所谓的 row_alias。

实际上 MySQL 的这个规范写得还是漏了一部分，也就是我们其实可以在 assignment 里面使用复杂的表达式，例如 `ON DUPLICATE KEY UPDATE update_time=now()` 又或者 `ON DUPLICATE KEY UPDATE epoch = epoch +1` 。ORM 框架需要将这种用法纳入考虑范围。

另外一个值得注意的点是：ON DUPLICATE KEY 是无法指定 KEY的，也就是说，假如我们的表上面定义了很多个唯一索引，那么任何一个唯一索引冲突（包含主键）都会引起执行更新。这和下面讨论的 PostgreSQL，SQLite3 非常不一样。

PostgreSQL

PostgreSQL 的语法在 INSERT 部分和 MySQL 的第一种形态接近。但是它的 upsert 部分采用的也是 ON CONFLICT 语法：

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
```

```
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] |
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
where conflict_target can be one of:
    ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ c
    ON CONSTRAINT constraint_name
and conflict_action is one of:
    DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAU
                    ( column_name [, ...] ) = ( sub-SELECT )
                  } [, ...]
              [ WHERE condition ]
```

ON CONFLICT 部分简单概括可以看做是：

- ON CONFLICT(col1, col2) DO NOTHING
- ON CONFLICT(co1, col2) DO UPDATE SET col1=xxx, ...

举例来说：

```sql
-- Do nothing 的例子
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;

-- 这个是指定了索引的例子，索引必须是唯一索引
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;

-- update 的例子，还在 Update 里面带了 where
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
    WHERE d.zipcode <> '21201';
```

SQLite3

SQLite3 的语法整体上要简单很多。在 INSERT 部分类似于 MySQL 的第一种形态，也就是我们所熟知的那种形态。而在 UPSERT 部分则是采用的是 ON CONFLICT 语法。完整的语法结构参考 INSERT 语句。

换句话说，SQLite3 的语法和 PostgreSQL 的语法更加接近。

框架分析

GORM 分析

GORM 的跟 Insert 有关的方法有：

- Create: 可以插入单个，也可以插入批量
- CreateInBatches：分批次插入，例如可以指定 10 个一批，那么 100 个就会拆成 10 批
- Save：接近于 INSERT or Update 的语义

```go
// Create insert the value into database
func (db *DB) Create(value interface{}) (tx *DB) {
    if db.CreateBatchSize > 0 {
        return db.CreateInBatches(value, db.CreateBatchSize)
    }
    tx = db.getInstance()
    tx.Statement.Dest = value
    return tx.callbacks.Create().Execute(tx)
}

// CreateInBatches insert the value in batches into database
func (db *DB) CreateInBatches(value interface{}, batchSize int) (tx *DB) {
    // ...
    return
}

// Save update value in database, if the value doesn't have primary key, wil
func (db *DB) Save(value interface{}) (tx *DB) {
    // ...
}
```

代码位于：gorm/gorm.go at master · go-gorm/gorm (github.com)
这里比较有特色的地方是 CreateInBatches，即 ORM 帮助用户解决了分批次插入的问题。但是整体上来说，我（们）认为，这个应该是应用层面上用户自己解决的问题，我们没有必要帮助用户解决。

> 注：GORM 最终拼出来一个 SQL 的过程是很复杂的，不必细究，因为后面我们 ORM 的思路和它不一样，我在 ORM 的小课上大概提到了 GORM 的设计思路，可以去看看

Beego ORM

Beego ORM 的接口定义是放在 beego/types.go at develop · beego/beego (github.com)
里面提供了方法：

- Insert 和 InsertWithCtx：插入单个
- InsertOrUpdate 和 InsertOrUpdateWithCtx：插入或者更新，同时支持指定冲突列。在实际过程中，用户会困惑与 MySQL 怎么指定冲突列，因为 MySQL 语法根本不支持冲突列
- InsertMulti 和 InsertMultiWithCtx：类似于 GORM 的 CreateInBatches，分批次插入

```go
// Data Manipulation Language
    type DML interface {
        // 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+va
        // if colu type is integer : can use(+-*/), string : convert(colu,"v
        // postgres: InsertOrUpdate(model,"conflictColumnName") or InsertOrU
        // if colu type is integer : can use(+-*/), string : colu || "value'
        InsertOrUpdate(md interface{}, colConflitAndArgs ...string) (int64,
```

```
        InsertOrUpdateWithCtx(ctx context.Context, md interface{}, colConfl
        // insert some models to database
        InsertMulti(bulk int, mds interface{}) (int64, error)
        InsertMultiWithCtx(ctx context.Context, bulk int, mds interface{}) (
}
```

从实现的角度来说， Beego ORM 的代码和 GORM 的代码一样很复杂，相比之下，GORM 的代码属于设计上的复杂——即职责被切分给了很多接口，而 Beego 的复杂，则在于所有的代码都混在了一起，难以搞懂每个步骤具体是干什么的。

功能需求

生成查询

将一个结构体转化为一个对应的 INSERT 查询，查询包括：

- SQL
- 参数

要支持以下选项：

- 单个或者批量：但是我们不会把一次提交的数据拆分成过个批次，也就是不管用户提交了多少数据，我们都是一批次插入进去
- 指定列：
  - 普通列
  - 复杂表达式在初期阶段不支持，在下一阶段支持
- 返回主键：只需要返回 last_insert_id
- upsert：支持 MySQL，PostgreSQL 和 SQlite3 的语法，并且在后两者中支持 UPDATE 和 DO NOTHING 两种动作。ON CONFLICT 部分可以只支持传入列名，而不支持指定冲突索引

方言支持

在 upsert 处涉及到了方言的处理，也就是说需要有依据方言来构建不同 SQL 的能力。因此需要设计一个综合的方言解决方案，该方案要求：

- 可以方便扩展新的方言
- 不同方言之间的实现相互隔离，互不影响

目前仅仅需要考虑支持 MySQL, PostgreSQL 和 SQLite3。

非功能需求

- 扩展性。要求我们在将来支持更加复杂的 upsert 语句的情况下，变更极小
- 耦合性。方言之间不存耦合

设计

总体设计

总体设计上，我们采用 Builder 模式，即定义一个新的 `Inserter` ：

```
type Inserter[T any] struct {

}

// 构建 SQL
```

```go
func (i *Inserter[T]) Build() (Query, error) {}


type Query struct {
  SQL string
  Args []any
}
```

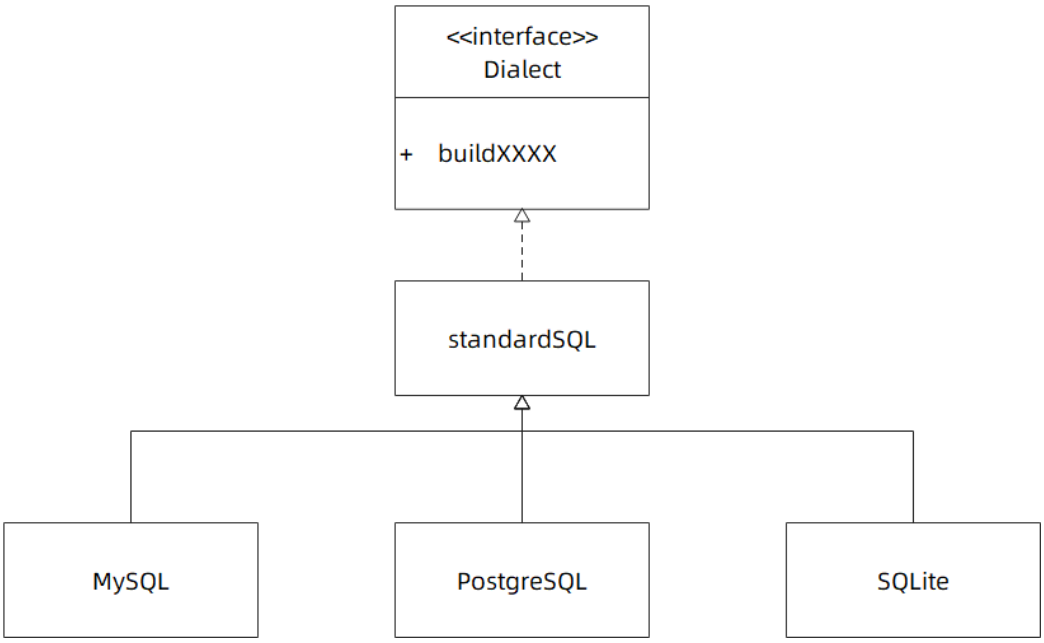
在 Builder 模式之下，任何复杂部分都可以拆成几个单一的方法。

与此同时，我们在 Inserter 上使用了泛型了，用来约束用户所能传入的类型。因此插入不同类型这种情况，是不会出现的，因为用户会得到编译错误。

方言

于此同时，为了解决方言的问题，我们需要进一步引入一个新的抽象 `Dialect` 。这个 `Dialect` 用于屏蔽不同方言之间的不同。

同时，除了为 Dialect 提供各种不同的实现以外，还会有一个基于标准 SQL 的实现，standardSQL。在引入了 standardSQL 之后，整体的方言抽象就变成了：



总结：

- 如果 SQL 的某一部分，不同方言之间有差异，那么就在 Dialect 里面新增一个方法
- 如果方言遵守 SQL 标准，那么我们不需要对它进行特殊处理
- 如果方言不遵守 SQL 标准，那么具体方言的实现就需要提供自己的实现

详细设计

Insert

在 INSERT 部分，需要考虑的就是两个问题： 如何指定插入的行，以及如何指定插入的列。

Values 方法

```go
func (i *Inserter[T]) Values(vals...*T) *Inserter {
```

```
  panic("implement me")
}
```

在这个方法里面，如果用户传入单个值，例如 `Values(&User)` ，那么就是插入一行；如果用户传入了多个值，例如 `Values(&user{}, &User{})` 那么就是批量插入。如果用户没有调用，或者调用了但是没有传值，例如 `Values()` 那么我们将在构建 SQL 的时候返回错误。

在这种设计之下，我们并没有区别单个插入还是批量插入。

> 实际上这里 vals 可以不用指针

Columns 方法

```
func (i *Inserter[T]) Columns(cols...string) *Inserter[T] {
  panic("implement me")
}
```

目前这种设计，我们放弃了支持复杂表达式，所以用户只能传入具体的列，而不能指定列的表达式，例如 now() 这种数据库方法调用。

Exec 方法

```
func (i *Inserter[T]) Exec(ctx context.Context) (sql.Result, error) {
  panic("implement me")
}
```

将会发起查询，并且返回结果。用户可以通过 `sql.Result` 拿到 `last_insert_id` ,也可以拿到受影响行数，即插入数量。

Upsert

在 Insert 部分，我们只是构建了 INSERT 的主要部分，但是如果用户想要使用 upsert 特性，那么就需要调用额外的方法。

为了支持 upsert，首先需要定义额外的结构体：

```
type Upsert struct {
  doNothing bool
  updateColumns []string
  conflictColumns []string
}
type UpsertBuilder[T any] struct {
  i *Inserter[T]
  conflictColumns []string
}

// Update 指定在冲突的时候要执行 update 的列
func (u *UpsertBuilder[T]) Update(cols...string) *Inserter[T] {
   i.upsert = Upsert{conflictColumns: u.conflictColumns, updateColumns: cols
   return i
}

```

```go
func (u *UpsertBuilder[T]) DoNothing() *Inserter[T] {
  i.upsert = Upsert{conflictColumns: u.conflictColumns, doNothing: true}
}

func (u *UpsertBuilder[T]) ConflictColumns(cols...string) *UpsertBuilder[T]
  u.conflictColumns = cols
}
```

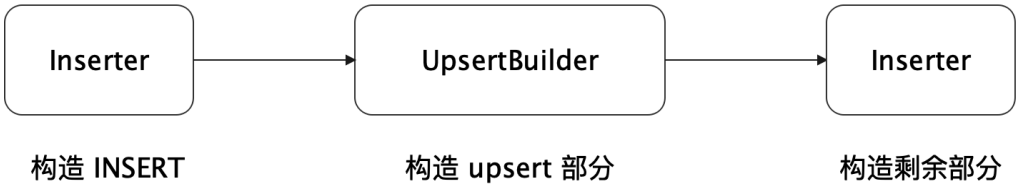
这里面明显分成两类方法：

- 中间方法：ConflictColumns，用户指定冲突的列，在将来我们可以增加对索引的支持
- 终结方法：DoNothing 和 Update，调用这两个方法后，重新回到 Inserter

很显然，在 Inserter 里面就是要暴露一个方法允许用户跳过去 UpsertBuilder 里面：

```go
func (i *Inserter[T]) Upsert() *UpsertBuilder[T] {
    return &UpsertBuilder {
      i: i
    }
}
```

使用起来的效果是：



例子

假如说我们有一个结构体：

```go
type User struct {
  ID uint64
  Email string
  FirstName string
  Age uint8
}
```

单个插入

```go
NewInserter[User]().Values(&User{ID: 1, Email: "xxx@xx"})
// INSERT INTO `user`(id, email, first_name, age) VALUES(?,?,?,?)
// []{1, "xxx@xx", "", 0}
```

批量插入

```go
NewInserter[User]().Values(&User{ID: 1, Email: "xxx@xx"},
&User{ID: 2, Email:"bb@aa", Age: 18})
// INSERT INTO `user`(id, email, first_name, age) VALUES(?,?,?,?),(?,?,?,?)
// []{1, "xxx@xx", "", 0, 2, "bb@aa", "", 18}
```

指定列

```go
NewInserter[User]().Values(&User{ID: 1, Email: "xxx@xx"}).Columns("id","ema
// INSERT INTO `user`(id, email) VALUES(?,?)
// []{1, "xxx@xx"}
```

不插入主键，只是在指定列的时候把主键排除掉

```
NewInserter[User]().Values(&User{Email: "xxx@xx", FirstName:"Deng"}).Columns
// INSERT INTO `user`(email, first_name, age) VALUES(?,?,?)
// []{"xxx@xx", "Deng", 0}
```

MySQL 冲突，更新列：

```
NewInserter[User]().Values(&User{ID: 1,Email: "xxx@xx", FirstName:"Deng"}).
Upsert().Update("first_name")
// INSERT INTO `user`(email, first_name, age) VALUES(?,?,?,?)
// ON DUPLICATE KEY UPDATE `first_name`=VALUES(`first_name`)
// []{1, "xxx@xx", "Deng", 0}
```

SQlLite 冲突，DoNothing

```
NewInserter[User]().Values(&User{ID: 1,Email: "xxx@xx", FirstName:"Deng"}).
Upsert().ConflictColumn("email")
// INSERT INTO `user`(email, first_name, age) VALUES(?,?,?,?)
// ON CONFLICT(email) DO NOTHING
// []{1, "xxx@xx", "Deng", 0}
```

测试

单元测试

影响最终结果的主要有两个因素：

- 输入的形态：nil, 非指针，指针，多级指针
- 结构体的定义：
  - 普通结构体
  - 组合
  - 指针式组合
  - 多重组合
  - 深层组合
  - 组合字段冲突（即含有同名字段）
- 指定列：
  - 直接指定列
  - 指定列的表达式：例如 now()
- Upsert 语句：
  - 指定冲突列
  - 不同方言：MySQL, SQLite 和 PostgreSQL
  - 冲突时候更新列的值：
    - 更新为一个具体值
    - 更新为插入的值

将这些因素进行交叉组合，得到所有的测试用例。注意，这些因素从设计的角度来看是各自独立的，所以可以考虑针对不同的因素分别设计测试用例，而不需要进行笛卡尔积。