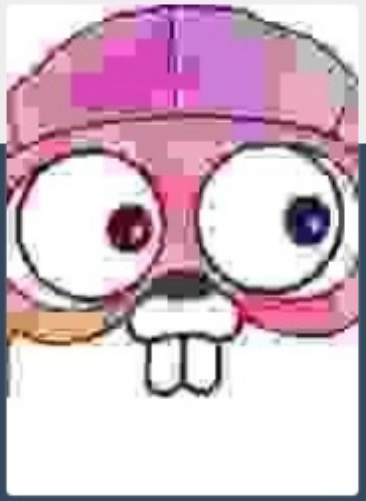
Xorm操作指南中文版 v0.6.5



xorm是一个简单而强大的Go语言ORM库. 通过它可以使数据库操作非常简便。xorm的目标并不是让你完全不去学习SQL，我们认为SQL并不会为ORM所替代，但是ORM将可以解决绝大部分的简单SQL需求。xorm支持两种…

下载手机APP
畅享精彩阅读

目　录

致谢

当前文档 《Xorm操作指南中文版 v0.6.5》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2019-11-17。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：go-xorm https://github.com/go-xorm/manual-zh-CN

文档地址：http://www.bookstack.cn/books/go-xorm-0.6.5

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

xorm

xorm是一个简单而强大的Go语言ORM库. 通过它可以使数据库操作非常简便。xorm的目标并不是让你完全不去学习SQL，我们认为SQL并不会为ORM所替代，但是ORM将可以解决绝大部分的简单SQL需求。xorm支持两种风格的混用。

特性

- 支持Struct和数据库表之间的灵活映射，并支持自动同步
- 事务支持
- 同时支持原始SQL语句和ORM操作的混合执行
- 使用连写来简化调用
- 支持使用Id, In, Where, Limit, Join, Having, Table, SQL, Cols等函数和结构体等方式作为条件
- 支持级联加载Struct
- Schema支持（仅Postgres）
- 支持缓存
- 支持根据数据库自动生成xorm的结构体
- 支持记录版本（即乐观锁）
- 内置SQL Builder支持
- 通过EngineGroup支持读写分离和负载均衡

驱动支持

xorm当前支持的驱动和数据库如下:

- Mysql: github.com/go-sql-driver/mysql
- MyMysql: github.com/ziutek/mymysql/godrv
- Postgres: github.com/lib/pq
- Tidb: github.com/pingcap/tidb
- SQLite: github.com/mattn/go-sqlite3
- MsSql: github.com/denisenkom/go-mssqldb
- MsSql: github.com/lunny/godbc

- Oracle: github.com/mattn/go-oci8 (试验性支持)
- ql: github.com/cznic/ql (试验性支持)

安装

```
1. go get xorm.io/xorm
```

文档

- 操作指南
- GoWalker代码文档
- Godoc代码文档

讨论

请加入QQ群: 280360085 进行讨论。

贡献

如果您也想为Xorm贡献您的力量，请查看 CONTRIBUTING

LICENSE

BSD Licensehttp://creativecommons.org/licenses/BSD/

创建 ORM 引擎

所有操作均需要事先创建并配置 ORM 引擎才可以进行。XORM支持两种 ORM 引擎，即 Engine 引擎和 Engine Group 引擎。一个 Engine 引擎用于对单个数据库进行操作，一个 Engine Group 引擎用于对读写分离的数据库或者负载均衡的数据库进行操作。Engine 引擎和 EngineGroup 引擎的API基本相同，所有适用于 Engine 的API基本上都适用于 EngineGroup，并且可以比较容易的从 Engine 引擎迁移到 EngineGroup引擎。

创建 Engine 引擎

单个ORM引擎，也称为Engine。一个APP可以同时存在多个Engine引擎，一个Engine一般只对应一个数据库。Engine通过调用 `xorm.NewEngine` 生成，如：

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

var engine *xorm.Engine

func main() {
    var err error
    engine, err = xorm.NewEngine("mysql", "root:123@/test?charset=utf8")
}
```

or

```
import (
    _ "github.com/mattn/go-sqlite3"
    "github.com/go-xorm/xorm"
)

var engine *xorm.Engine

func main() {
    var err error
    engine, err = xorm.NewEngine("sqlite3", "./test.db")
}
```

一般情况下如果只操作一个数据库，只需要创建一个 `engine` 即可。 `engine` 是GoRoutine安全的。

创建完成 `engine` 之后，并没有立即连接数据库，此时可以通过 `engine.Ping()` 来进行数据库的连接测试是否可以连接到数据库。另外对于某些数据库有连接超时设置的，可以通过起一个定期Ping的Go程来保持连接鲜活。

对于有大量数据并且需要分区的应用，也可以根据规则来创建多个Engine，比如：

```
var err error
for i:=0;i<5;i++ {
    engines[i], err = xorm.NewEngine("sqlite3", fmt.Sprintf("./test%d.db", i))
}
```

engine可以通过engine.Close来手动关闭，但是一般情况下可以不用关闭，在程序退出时会自动关闭。

NewEngine传入的参数和 `sql.Open` 传入的参数完全相同，因此，在使用某个驱动前，请查看此驱动中关于传入参数的说明文档。以下为各个驱动的连接符对应的文档链接：

- sqlite3
- mysql dsn
- mymysql
- postgres

在engine创建完成后可以进行一些设置，如：

日志

日志是一个接口，通过设置日志，可以显示SQL，警告以及错误等，默认的显示级别为INFO。

- `engine.ShowSQL(true)`，则会在控制台打印出生成的SQL语句；
- `engine.Logger().SetLevel(core.LOG_DEBUG)`，则会在控制台打印调试及以上的信息；

如果希望将信息不仅打印到控制台，而是保存为文件，那么可以通过类似如下的代码实现，`NewSimpleLogger(w io.Writer)` 接收一个io.Writer接口来将数据写入到对应的设施中。

```
f, err := os.Create("sql.log")
if err != nil {
    println(err.Error())
    return
}
engine.SetLogger(xorm.NewSimpleLogger(f))
```

当然，如果希望将日志记录到syslog中，也可以如下：

```
logWriter, err := syslog.New(syslog.LOG_DEBUG, "rest-xorm-example")
if err != nil {
    log.Fatalf("Fail to create xorm system logger: %v\n", err)
}

logger := xorm.NewSimpleLogger(logWriter)
logger.ShowSQL(true)
engine.SetLogger(logger)
```

连接池

engine内部支持连接池接口和对应的函数。

- 如果需要设置连接池的空闲数大小，可以使用`engine.SetMaxIdleConns()`来实现。
- 如果需要设置最大打开连接数，则可以使用`engine.SetMaxOpenConns()`来实现。

创建 Engine Group 引擎

通过创建引擎组EngineGroup来实现对从数据库(Master/Slave)读写分离的支持。在创建引擎章节中，我们已经介绍过了，在xorm里面，可以同时存在多个Orm引擎，一个Orm引擎称为Engine，一个Engine一般只对应一个数据库，而EngineGroup一般则对应一组数据库。EngineGroup通过调用xorm.NewEngineGroup生成，如:

```
import (
    _ "github.com/lib/pq"
    "github.com/xormplus/xorm"
)

var eg *xorm.EngineGroup

func main() {
    conns := []string{
        "postgres://postgres:root@localhost:5432/test?sslmode=disable;", // 第一个默认是master
        "postgres://postgres:root@localhost:5432/test1?sslmode=disable;", // 第二个开始都是slave
        "postgres://postgres:root@localhost:5432/test2?sslmode=disable",
    }

    var err error
    eg, err = xorm.NewEngineGroup("postgres", conns)
}
```

或者

```
import (
    _ "github.com/lib/pq"
    "github.com/xormplus/xorm"
)

var eg *xorm.EngineGroup

func main() {
    var err error
    master, err := xorm.NewEngine("postgres", "postgres://postgres:root@localhost:5432/test?sslmode=disable")
    if err != nil {
        return
    }

    slave1, err := xorm.NewEngine("postgres", "postgres://postgres:root@localhost:5432/test1?sslmode=disable")
    if err != nil {
        return
    }

    slave2, err := xorm.NewEngine("postgres", "postgres://postgres:root@localhost:5432/test2?sslmode=disable")
    if err != nil {
        return
    }
```

```
24.
25.         slaves := []*xorm.Engine{slave1, slave2}
26.     eg, err = xorm.NewEngineGroup(master, slaves)
27. }
```

创建完成EngineGroup之后，并没有立即连接数据库，此时可以通过eg.Ping()来进行数据库的连接测试是否可以连接到数据库，该方法会依次调用引擎组中每个Engine的Ping方法。另外对于某些数据库有连接超时设置的，可以通过起一个定期Ping的Go程来保持连接鲜活。EngineGroup可以通过eg.Close()来手动关闭，但是一般情况下可以不用关闭，在程序退出时会自动关闭。

- NewEngineGroup方法

```
1. func NewEngineGroup(args1 interface{}, args2 interface{}, policies ...GroupPolicy) (*EngineGroup, error)
```

前两个参数的使用示例如上，有两种模式。模式一：通过给定DriverName，DataSourceName来创建引擎组，每个引擎使用相同的Driver。每个引擎的DataSourceNames是[]string类型，第一个元素是Master的DataSourceName，之后的元素是Slave的DataSourceName。模式一：通过给定*xorm.Engine*，*[]*xorm.Engine来创建引擎组，每个引擎可以使用不同的Driver。第一个参数为Master的*xorm.Engine*，第二个参数为*Slave*的*[]*xorm.Engine。NewEngineGroup方法，第三个参数为policies，为Slave给定负载策略，该参数将在负载策略章节详细介绍，如示例中未指定，则默认为轮询负载策略。

- Master方法

  ```
  1. func (eg *EngineGroup) Master() *Engine
  ```

  返回Master数据库引擎

- Slave方法

  ```
  1. func (eg *EngineGroup) Slave() *Engine
  ```

  依据给定的负载策略返回一个Slave数据库引擎

- Slaves方法

  ```
  1. func (eg *EngineGroup) Slaves() []*Engine
  ```

  返回所以Slave数据库引擎

- GetSlave方法

  ```
  1. func (eg *EngineGroup) GetSlave(i int) *Engine
  ```

依据一组Slave数据库引擎[]*xorm.Engine下标返回指定Slave数据库引擎。通过给定DriverName，DataSourceName来创建引擎组，则DataSourceName的第二个元素的数据库为下标0的Slave数据库引擎。

- SetPolicy方法

```
1. func (eg *EngineGroup) SetPolicy(policy GroupPolicy) *EngineGroup
```

设置引擎组负载策略

负载策略

通过xorm.NewEngineGroup创建EngineGroup时，第三个参数为policies，我们可以通过该参数来指定Slave访问的负载策略。如创建EngineGroup时未指定，则默认使用轮询的负载策略。

xorm中内置五种负载策略，分别为随机访问负载策略，权重随机访问负载策略，轮询访问负载策略，权重轮询访问负载策略和最小连接数访问负载策略。开发者也可以通过实现GroupPolicy接口，来实现自定义负载策略。

- 随机访问负载策略

```
import (
    _ "github.com/lib/pq"
    "github.com/xormplus/xorm"
)

var eg *xorm.EngineGroup

func main() {
    conns := []string{
        "postgres://postgres:root@localhost:5432/test?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test1?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test2?sslmode=disable",
    }

    var err error
    eg, err = xorm.NewEngineGroup("postgres", conns, xorm.RandomPolicy())
}
```

- 权重随机访问负载策略

```
import (
    _ "github.com/lib/pq"
    "github.com/xormplus/xorm"
)

var eg *xorm.EngineGroup

func main() {
    conns := []string{
        "postgres://postgres:root@localhost:5432/test?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test1?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test2?sslmode=disable",
    }

    var err error
    //此时设置的test1数据库和test2数据库的随机访问权重为2和3
    eg, err = xorm.NewEngineGroup("postgres", conns, xorm.WeightRandomPolicy([]int{2, 3}))
}
```

- 轮询访问负载策略

```
import (
    _ "github.com/lib/pq"
    "github.com/xormplus/xorm"
)

var eg *xorm.EngineGroup

func main() {
    conns := []string{
        "postgres://postgres:root@localhost:5432/test?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test1?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test2?sslmode=disable",
    }

    var err error
    eg, err = xorm.NewEngineGroup("postgres", conns, xorm.RoundRobinPolicy())
}
```

- 权重轮询访问负载策略

```
import (
    _ "github.com/lib/pq"
    "github.com/xormplus/xorm"
)

var eg *xorm.EngineGroup

func main() {
    conns := []string{
        "postgres://postgres:root@localhost:5432/test?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test1?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test2?sslmode=disable",
    }

    var err error
    //此时设置的test1数据库和test2数据库的轮询访问权重为2和3
    eg, err = xorm.NewEngineGroup("postgres", conns, xorm.WeightRoundRobinPolicy([]int{2, 3}))
}
```

- 最小连接数访问负载策略

```
import (
    _ "github.com/lib/pq"
    "github.com/xormplus/xorm"
)

var eg *xorm.EngineGroup

func main() {
```

```
    conns := []string{
        "postgres://postgres:root@localhost:5432/test?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test1?sslmode=disable;",
        "postgres://postgres:root@localhost:5432/test2?sslmode=disable",
    }

    var err error
    eg, err = xorm.NewEngineGroup("postgres", conns, xorm.LeastConnPolicy())
}
```

- 自定义负载策略

你也可以通过实现 `GroupPolicy` 接口来实现自定义负载策略。

```
type GroupPolicy interface {
    Slave(*EngineGroup) *Engine
}
```

2.定义表结构体

xorm支持将一个struct映射为数据库中对应的一张表。映射规则可以查看：名称映射规则。

名称映射规则

名称映射规则主要负责结构体名称到表名和结构体field到表字段的名称映射。由core.IMapper接口的实现者来管理，xorm内置了三种IMapper实现： `core.SnakeMapper` ， `core.SameMapper` 和 `core.GonicMapper` 。

- SnakeMapper 支持struct为驼峰式命名，表结构为下划线命名之间的转换，这个是默认的Maper；
- SameMapper 支持结构体名称和对应的表名称以及结构体field名称与对应的表字段名称相同的命名；
- GonicMapper 和SnakeMapper很类似，但是对于特定词支持更好，比如ID会翻译成id而不是i_d。

当前SnakeMapper为默认值，如果需要改变时，在engine创建完成后使用

```
1. engine.SetMapper(core.SameMapper{})
```

同时需要注意的是：

- 如果你使用了别的命名规则映射方案，也可以自己实现一个IMapper。
- 表名称和字段名称的映射规则默认是相同的，当然也可以设置为不同，如：

```
1. engine.SetTableMapper(core.SameMapper{})
2. engine.SetColumnMapper(core.SnakeMapper{})
```

When a struct auto mapping to a database’s table, the below table describes how they change to each other:

go type’s kind	value method	xorm type
implemented Conversion	Conversion.ToDB / Conversion.FromDB	Text
int, int8, int16, int32, uint, uint8, uint16, uint32		Int
int64, uint64		BigInt
float32		Float
float64		Double
complex64, complex128	json.Marshal / json.UnMarshal	Varchar(64)
[]uint8		Blob
array, slice, map except []uint8	json.Marshal / json.UnMarshal	Text
bool	1 or 0	Bool
string		Varchar(255)
time.Time		DateTime
cascade struct	primary key field value	BigInt
struct	json.Marshal / json.UnMarshal	Text
Others		Text

前缀映射，后缀映射和缓存映射

- 通过 `core.NewPrefixMapper(core.SnakeMapper{}, "prefix")` 可以创建一个在SnakeMapper的基础上在命名中添加统一的前缀，当然也可以把SnakeMapper{}换成SameMapper或者你自定义的Mapper。

例如，如果希望所有的表名都在结构体自动命名的基础上加一个前缀而字段名不加前缀，则可以在engine创建完成后执行以下语句：

```
1. tbMapper := core.NewPrefixMapper(core.SnakeMapper{}, "prefix_")
2. engine.SetTableMapper(tbMapper)
```

执行之后，结构体 `type User struct` 默认对应的表名就变成了 `prefix_user` 了，而之前默认的是 `user`

- 通过 `core.NewSuffixMapper(core.SnakeMapper{}, "suffix")` 可以创建一个在SnakeMapper的基础上在命名中添加统一的后缀，当然也可以把SnakeMapper换成SameMapper或者你自定义的Mapper。
- 通过 `core.NewCacheMapper(core.SnakeMapper{})` 可以创建一个组合了其它的映射规则，起到在内存中缓存曾经映射过的命名映射。

使用Table和Tag改变名称映射

如果所有的命名都是按照IMapper的映射来操作的，那当然是最理想的。但是如果碰到某个表名或者某个字段名跟映射规则不匹配时，我们就需要别的机制来改变。xorm提供了如下几种方式来进行：

- 如果结构体拥有 `TableName() string` 的成员方法，那么此方法的返回值即是该结构体对应的数据库表名。
- 通过 `engine.Table()` 方法可以改变struct对应的数据库表的名称，通过sturct中field对应的Tag中使用 `xorm:"'column_name'"` 可以使该field对应的Column名称为指定名称。这里使用两个单引号将Column名称括起来是为了防止名称冲突，因为我们在Tag中还可以对这个Column进行更多的定义。如果名称不冲突的情况，单引号也可以不使用。

到此名称映射的所有方法都给出了，一共三种方式，这三种是有优先级顺序的。

- 表名的优先级顺序如下：
    - `engine.Table()` 指定的临时表名优先级最高
    - `TableName() string` 其次
    - `Mapper` 自动映射的表名优先级最后
- 字段名的优先级顺序如下：
    - 结构体tag指定的字段名优先级较高
    - `Mapper` 自动映射的表名优先级较低

Column属性定义

我们在field对应的Tag中对Column的一些属性进行定义，定义的方法基本和我们写SQL定义表结构类似，比如：

```
type User struct {
    Id   int64
    Name string  `xorm:"varchar(25) notnull unique 'usr_name' comment('姓名')"`
}
```

对于不同的数据库系统，数据类型其实是有些差异的。因此xorm中对数据类型有自己的定义，基本的原则是尽量兼容各种数据库的字段类型，具体的字段对应关系可以查看字段类型对应表。对于使用者，一般只要使用自己熟悉的数据库字段定义即可。

具体的Tag规则如下，另Tag中的关键字均不区分大小写，但字段名根据不同的数据库是区分大小写：

name	当前field对应的字段的名称，可选，如不写，则自动根据field名字和转换规则命名，如与其它关键字冲突，请使用单引号括起来。
pk	是否是Primary Key，如果在一个struct中有多个字段都使用了此标记，则这多个字段构成了复合主键，单主键当前支持int32,int,int64,uint32,uint,uint64,string这7种Go的数据类型，复合主键支持这7种Go的数据类型的组合。
当前支持30多种字段类型，详情参见本文最后一个表格	字段类型
autoincr	是否是自增
[not]null 或notnull	是否可以为空
unique或unique(uniquename)	是否是唯一，如不加括号则该字段不允许重复；如加上括号，则括号中为联合唯一索引的名字，此时如果有另外一个或多个字段和本unique的uniquename相同，则这些uniquename相同的字段组成联合唯一索引
index或index(indexname)	是否是索引，如不加括号则该字段自身为索引，如加上括号，则括号中为联合索引的名字，此时如果有另外一个或多个字段和本index的indexname相同，则这些indexname相同的字段组成联合索引
extends	应用于一个匿名成员结构体或者非匿名成员结构体之上，表示此结构体的所有成员也映射到数据库中，extends可加载无限级
-	这个Field将不进行字段映射
->	这个Field将只写入到数据库而不从数据库读取
<-	这个Field将只从数据库读取，而不写入到数据库
created	这个Field将在Insert时自动赋值为当前时间
updated	这个Field将在Insert或Update时自动赋值为当前时间
deleted	这个Field将在Delete时设置为当前时间，并且当前记录不删除
version	这个Field将会在insert时默认为1，每次更新自动加1
default 0或default(0)	设置默认值，紧跟的内容如果是Varchar等需要加上单引号
json	表示内容将先转成Json格式，然后存储到数据库中，数据库中的字段类型可以为Text或者二进制

comment	设置字段的注释（当前仅支持mysql）

另外有如下几条自动映射的规则：

- 1.如果field名称为 `Id` 而且类型为 `int64` 并且没有定义tag，则会被xorm视为主键，并且拥有自增属性。如果想用 `Id` 以外的名字或非int64类型做为主键名，必须在对应的Tag上加上 `xorm:"pk"` 来定义主键，加上 `xorm:"autoincr"` 作为自增。这里需要注意的是，有些数据库并不允许非主键的自增属性。
- 2.string类型默认映射为 `varchar(255)` ，如果需要不同的定义，可以在tag中自定义，如： `varchar(1024)`
- 3.支持 `type MyString string` 等自定义的field，支持Slice， Map等field成员，这些成员默认存储为Text类型，并且默认将使用Json格式来序列化和反序列化。也支持数据库字段类型为Blob类型。如果是Blob类型，则先使用Json格式序列化再转成[]byte格式。如果是[]byte或者[]uint8，则不做转换二十直接以二进制方式存储。具体参见 Go与字段类型对应表
- 4.实现了Conversion接口的类型或者结构体，将根据接口的转换方式在类型和数据库记录之间进行相互转换，这个接口的优先级是最高的。

```
type Conversion interface {
  FromDB([]byte) error
  ToDB() ([]byte, error)
}
```

- 5.如果一个结构体包含一个Conversion的接口类型，那么在获取数据时，必须要预先设置一个实现此接口的struct或者struct的指针。此时可以在此struct中实现 `BeforeSet(name string, cell xorm.Cell)` 方法来进行预先给Conversion赋值。例子参见 testConversion

下表为xorm类型和各个数据库类型的对应表：

xorm	mysql	sqlite3	postgres	remark
BIT	BIT	INTEGER	BIT	
TINYINT	TINYINT	INTEGER	SMALLINT	
SMALLINT	SMALLINT	INTEGER	SMALLINT	
MEDIUMINT	MEDIUMINT	INTEGER	INTEGER	
INT	INT	INTEGER	INTEGER	
INTEGER	INTEGER	INTEGER	INTEGER	
BIGINT	BIGINT	INTEGER	BIGINT	
CHAR	CHAR	TEXT	CHAR	
VARCHAR	VARCHAR	TEXT	VARCHAR	
TINYTEXT	TINYTEXT	TEXT	TEXT	
TEXT	TEXT	TEXT	TEXT	
MEDIUMTEXT	MEDIUMTEXT	TEXT	TEXT	
LONGTEXT	LONGTEXT	TEXT	TEXT	

BINARY	BINARY	BLOB	BYTEA	
VARBINARY	VARBINARY	BLOB	BYTEA	
DATE	DATE	NUMERIC	DATE	
DATETIME	DATETIME	NUMERIC	TIMESTAMP	
TIME	TIME	NUMERIC	TIME	
TIMESTAMP	TIMESTAMP	NUMERIC	TIMESTAMP	
TIMESTAMPZ	TEXT	TEXT	TIMESTAMP with zone	timestamp with zone info
REAL	REAL	REAL	REAL	
FLOAT	FLOAT	REAL	REAL	
DOUBLE	DOUBLE	REAL	DOUBLE PRECISION	
DECIMAL	DECIMAL	NUMERIC	DECIMAL	
NUMERIC	NUMERIC	NUMERIC	NUMERIC	
TINYBLOB	TINYBLOB	BLOB	BYTEA	
BLOB	BLOB	BLOB	BYTEA	
MEDIUMBLOB	MEDIUMBLOB	BLOB	BYTEA	
LONGBLOB	LONGBLOB	BLOB	BYTEA	
BYTEA	BLOB	BLOB	BYTEA	
BOOL	TINYINT	INTEGER	BOOLEAN	
SERIAL	INT	INTEGER	SERIAL	auto increment
BIGSERIAL	BIGINT	INTEGER	BIGSERIAL	auto increment

Go与字段类型对应表

如果不使用tag来定义field对应的数据库字段类型，那么系统会自动给出一个默认的字段类型，对应表如下：

go type’s kind	value method	xorm type
implemented Conversion	Conversion.ToDB / Conversion.FromDB	Text
int, int8, int16, int32, uint, uint8, uint16, uint32		Int
int64, uint64		BigInt
float32		Float
float64		Double
complex64, complex128	json.Marshal / json.UnMarshal	Varchar(64)
[]uint8		Blob
array, slice, map except []uint8	json.Marshal / json.UnMarshal	Text
bool	1 or 0	Bool
string		Varchar(255)
time.Time		DateTime
cascade struct	primary key field value	BigInt
struct	json.Marshal / json.UnMarshal	Text
Others		Text

表结构操作

xorm提供了一些动态获取和修改表结构的方法，通过这些方法可以动态同步数据库结构，导出数据库结构，导入数据库结构。

如果您只是需要一个工具，可以直接使用 `go get github.com/go-xorm/cmd/xorm` 来安装xorm命令行工具。

获取数据库信息

- DBMetas()

xorm支持获取表结构信息，通过调用 `engine.DBMetas()` 可以获取到数据库中所有的表，字段，索引的信息。

- TableInfo()

根据传入的结构体指针及其对应的Tag，提取出模型对应的表结构信息。这里不是数据库当前的表结构信息，而是我们通过struct建模时希望数据库的表的结构信息

表操作

- CreateTables()

创建表使用 `engine.CreateTables()` ，参数为一个或多个空的对应Struct的指针。同时可用的方法有Charset()和StoreEngine()，如果对应的数据库支持，这两个方法可以在创建表时指定表的字符编码和使用的引擎。Charset()和StoreEngine()当前仅支持Mysql数据库。

- IsTableEmpty()

判断表是否为空，参数和CreateTables相同

- IsTableExist()

判断表是否存在

- DropTables()

删除表使用 `engine.DropTables()` ，参数为一个或多个空的对应Struct的指针或者表的名字。如果为string传入，则只删除对应的表，如果传入的为Struct，则删除表的同时还会删除对应的索引。

创建索引和唯一索引

- CreateIndexes

根据struct中的tag来创建索引

- CreateUniques

根据struct中的tag来创建唯一索引

同步数据库结构

同步能够部分智能的根据结构体的变动检测表结构的变动，并自动同步。目前有两个实现：

- Sync

Sync将进行如下的同步操作：

```
* 自动检测和创建表，这个检测是根据表的名字
* 自动检测和新增表中的字段，这个检测是根据字段名
* 自动检测和创建索引和唯一索引，这个检测是根据索引的一个或多个字段名，而不根据索引名称
```

调用方法如下：

```
err := engine.Sync(new(User), new(Group))
```

- Sync2

Sync2对Sync进行了改进，目前推荐使用Sync2。Sync2函数将进行如下的同步操作：

```
* 自动检测和创建表，这个检测是根据表的名字
* 自动检测和新增表中的字段，这个检测是根据字段名，同时对表中多余的字段给出警告信息
* 自动检测，创建和删除索引和唯一索引，这个检测是根据索引的一个或多个字段名，而不根据索引名称。因此这里需要注意，如果在一个有大量数据的表中引入新的索引，数据库可能需要一定的时间来建立索引。
* 自动转换varchar字段类型到text字段类型，自动警告其它字段类型在模型和数据库之间不一致的情况。
* 自动警告字段的默认值，是否为空信息在模型和数据库之间不匹配的情况

以上这些警告信息需要将`engine.ShowWarn` 设置为 `true` 才会显示。
```

调用方法和Sync一样：

```
err := engine.Sync2(new(User), new(Group))
```

Dump数据库结构和数据

如果需要在程序中Dump数据库的结构和数据可以调用

```
engine.DumpAll(w io.Writer)
```

和

```
engine.DumpAllToFile(fpath string)
```
。

DumpAll方法接收一个io.Writer接口来保存Dump出的数据库结构和数据的SQL语句，这个方法导出的SQL语句并不能通用。只针对当前engine所对应的数据库支持的SQL。

Import 执行数据库SQL脚本

如果你需要将保存在文件或者其它存储设施中的SQL脚本执行，那么可以调用

```
engine.Import(r io.Reader)
```

和

```
engine.ImportFile(fpath string)
```

同样，这里需要对应的数据库的SQL语法支持。

插入数据

插入数据使用Insert方法，Insert方法的参数可以是一个或多个Struct的指针，一个或多个Struct的Slice的指针。

如果传入的是Slice并且当数据库支持批量插入时，Insert会使用批量插入的方式进行插入。

- 插入一条数据，此时可以用Insert或者InsertOne

```
1. user := new(User)
2. user.Name = "myname"
3. affected, err := engine.Insert(user)
4. // INSERT INTO user (name) values (?)
```

在插入单条数据成功后，如果该结构体有自增字段(设置为autoincr)，则自增字段会被自动赋值为数据库中的id。这里需要注意的是，如果插入的结构体中，自增字段已经赋值，则该字段会被作为非自增字段插入。

```
1. fmt.Println(user.Id)
```

- 插入同一个表的多条数据，此时如果数据库支持批量插入，那么会进行批量插入，但是这样每条记录就无法被自动赋予id值。如果数据库不支持批量插入，那么就会一条一条插入。

```
1. users := make([]User, 1)
2. users[0].Name = "name0"
3. ...
4. affected, err := engine.Insert(&users)
```

- 使用指针Slice插入多条记录，同上

```
1. users := make([]*User, 1)
2. users[0] = new(User)
3. users[0].Name = "name0"
4. ...
5. affected, err := engine.Insert(&users)
```

- 插入多条记录并且不使用批量插入，此时实际生成多条插入语句，每条记录均会自动赋予Id值。

```
1. users := make([]*User, 1)
2. users[0] = new(User)
3. users[0].Name = "name0"
4. ...
5. affected, err := engine.Insert(users...)
```

- 插入不同表的一条记录

```
1. user := new(User)
```

```
user.Name = "myname"
question := new(Question)
question.Content = "whywhywhwy?"
affected, err := engine.Insert(user, question)
```

- 插入不同表的多条记录

```
users := make([]User, 1)
users[0].Name = "name0"
...
questions := make([]Question, 1)
questions[0].Content = "whywhywhwy?"
affected, err := engine.Insert(&users, &questions)
```

- 插入不同表的一条或多条记录

```
user := new(User)
user.Name = "myname"
...
questions := make([]Question, 1)
questions[0].Content = "whywhywhwy?"
affected, err := engine.Insert(user, &questions)
```

这里需要注意以下几点：

- 这里虽然支持同时插入，但这些插入并没有事务关系。因此有可能在中间插入出错后，后面的插入将不会继续。此时前面的插入已经成功，如果需要回滚，请开启事务。
- 批量插入会自动生成 `Insert into table values (),(),()` 的语句，因此各个数据库对SQL语句有长度限制，因此这样的语句有一个最大的记录数，根据经验测算在150条左右。大于150条后，生成的sql语句将太长可能导致执行失败。因此在插入大量数据时，目前需要自行分割成每150条插入一次。

创建时间Created

Created可以让您在数据插入到数据库时自动将对应的字段设置为当前时间，需要在xorm标记中使用created标记，如下所示进行标记，对应的字段可以为time.Time或者自定义的time.Time或者int,int64等int类型。

```
type User struct {
    Id int64
    Name string
    CreatedAt time.Time `xorm:"created"`
}
```

或

```
type JsonTime time.Time
func (j JsonTime) MarshalJSON() ([]byte, error) {
    return []byte(`"`+time.Time(j).Format("2006-01-02 15:04:05")+`"`), nil
}

type User struct {
    Id int64
    Name string
    CreatedAt JsonTime `xorm:"created"`
}
```

或

```
type User struct {
    Id int64
    Name string
    CreatedAt int64 `xorm:"created"`
}
```

在Insert()或InsertOne()方法被调用时，created标记的字段将会被自动更新为当前时间或者当前时间的秒数（对应为time.Unix()），如下所示：

```
var user User
engine.Insert(&user)
// INSERT user (created...) VALUES (?...)
```

最后一个值得注意的是时区问题，默认xorm采用Local时区，所以默认调用的time.Now()会先被转换成对应的时区。要改变xorm的时区，可以使用：

```
engine.TZLocation, _ = time.LoadLocation("Asia/Shanghai")
```

查询和统计数据

所有的查询条件不区分调用顺序，但必须在调用Get，Exist，Sum，Find，Count，Iterate，Rows这几个函数之前调用。同时需要注意的一点是，在调用的参数中，如果采用默认的 `SnakeMapper` 所有的字符字段名均为映射后的数据库的字段名，而不是field的名字。

查询条件方法

查询和统计主要使用 `Get` , `Find` , `Count` , `Rows` , `Iterate` 这几个方法，同时大部分函数在调用 `Update` , `Delete` 时也是可用的。在进行查询时可以使用多个方法来形成查询条件，条件函数如下：

- Alias(string)

给Table设定一个别名

```
1. engine.Alias("o").Where("o.name = ?", name).Get(&order)
```

- And(string, …interface{})

和Where函数中的条件基本相同，作为条件

```
1. engine.Where(...).And(...).Get(&order)
```

- Asc(…string)

指定字段名正序排序，可以组合

```
1. engine.Asc("id").Find(&orders)
```

- Desc(…string)

指定字段名逆序排序，可以组合

```
1. engine.Asc("id").Desc("time").Find(&orders)
```

- ID(interface{})

传入一个主键字段的值，作为查询条件，如

```
1. var user User
2. engine.ID(1).Get(&user)
3. // SELECT * FROM user Where id = 1
```

如果是复合主键，则可以

```
1. engine.ID(core.PK{1, "name"}).Get(&user)
2. // SELECT * FROM user Where id =1 AND name= 'name'
```

传入的两个参数按照struct中pk标记字段出现的顺序赋值。

- Or(interface{}, …interface{})

和Where函数中的条件基本相同，作为条件

- OrderBy(string)

按照指定的顺序进行排序

- Select(string)

指定select语句的字段部分内容，例如：

```
engine.Select("a.*, (select name from b limit 1) as name").Find(&beans)

engine.Select("a.*, (select name from b limit 1) as name").Get(&bean)
```

- SQL(string, …interface{})

执行指定的Sql语句，并把结果映射到结构体。有时，当选择内容或者条件比较复杂时，可以直接使用Sql，例如：

```
engine.SQL("select * from table").Find(&beans)
```

- Where(string, …interface{})

和SQL中Where语句中的条件基本相同，作为条件

```
engine.Where("a = ? AND b = ?", 1, 2).Find(&beans)

engine.Where(builder.Eq{"a":1, "b": 2}).Find(&beans)

engine.Where(builder.Eq{"a":1}.Or(builder.Eq{"b": 2})).Find(&beans)
```

- In(string, …interface{})

某字段在一些值中，这里需要注意必须是[]interface{}才可以展开，由于Go语言的限制，[]int64等不可以直接展开，而是通过传递一个slice。第二个参数也可以是一个*builder.Builder 指针。示例代码如下：

```
// select from table where column in (1,2,3)
engine.In("cloumn", 1, 2, 3).Find()

// select from table where column in (1,2,3)
engine.In("column", []int{1, 2, 3}).Find()

// select from table where column in (select column from table2 where a = 1)
engine.In("column", builder.Select("column").From("table2").Where(builder.Eq{"a":1})).Find()
```

- Cols(…string)

只查询或更新某些指定的字段，默认是查询所有映射的字段或者根据Update的第一个参数来判断更新的字段。例如：

```
engine.Cols("age", "name").Get(&usr)
// SELECT age, name FROM user limit 1
```

```
engine.Cols("age", "name").Find(&users)
// SELECT age, name FROM user
engine.Cols("age", "name").Update(&user)
// UPDATE user SET age=? AND name=?
```

- AllCols()

查询或更新所有字段，一般与Update配合使用，因为默认Update只更新非0，非""，非bool的字段。

```
engine.AllCols().Id(1).Update(&user)
// UPDATE user SET name = ?, age =?, gender =? WHERE id = 1
```

- MustCols(…string)

某些字段必须更新，一般与Update配合使用。

- Omit(…string)

和cols相反，此函数指定排除某些指定的字段。注意：此方法和Cols方法不可同时使用。

```
// 例1：
engine.Omit("age", "gender").Update(&user)
// UPDATE user SET name = ? AND department = ?
// 例2：
engine.Omit("age, gender").Insert(&user)
// INSERT INTO user (name) values (?) // 这样的话age和gender会给默认值
// 例3：
engine.Omit("age", "gender").Find(&users)
// SELECT name FROM user //只select除age和gender字段的其它字段
```

- Distinct(…string)

按照参数中指定的字段归类结果。

```
engine.Distinct("age", "department").Find(&users)
// SELECT DISTINCT age, department FROM user
```

注意：当开启了缓存时，此方法的调用将在当前查询中禁用缓存。因为缓存系统当前依赖Id，而此时无法获得Id

- Table(nameOrStructPtr interface{})

传入表名称或者结构体指针，如果传入的是结构体指针，则按照IMapper的规则提取出表名

- Limit(int, …int)

限制获取的数目，第一个参数为条数，第二个参数表示开始位置，如果不传则为0

- Top(int)

相当于Limit(int, 0)

- Join(string,interface{},string)

第一个参数为连接类型，当前支持INNER, LEFT OUTER, CROSS中的一个值，第二个参数为string类型的表名，表对应的结构体指针或者为两个值的[]string，表示表名和别名，第三个参数为连接条件

```
1. 详细用法参见 [5.Join的使用](5.join.md)
```

- GroupBy(string)

Groupby的参数字符串

- Having(string)

Having的参数字符串

临时开关方法

- NoAutoTime()

如果此方法执行，则此次生成的语句中Created和Updated字段将不自动赋值为当前时间

- NoCache()

如果此方法执行，则此次生成的语句则在非缓存模式下执行

- NoAutoCondition()

禁用自动根据结构体中的值来生成条件

```
1. engine.Where("name = ?", "lunny").Get(&User{Id:1})
2. // SELECT * FROM user where name='lunny' AND id = 1 LIMIT 1
3. engine.Where("name = ?", "lunny").NoAutoCondition().Get(&User{Id:1})
4. // SELECT * FROM user where name='lunny' LIMIT 1
```

- UseBool(…string)

当从一个struct来生成查询条件或更新字段时，xorm会判断struct的field是否为0,””,nil，如果为以上则不当做查询条件或者更新内容。因为bool类型只有true和false两种值，因此默认所有bool类型不会作为查询条件或者更新字段。如果可以使用此方法，如果默认不传参数，则所有的bool字段都将会被使用，如果参数不为空，则参数中指定的为字段名，则这些字段对应的bool值将被使用。

- NoCascade()

是否自动关联查询field中的数据，如果struct的field也是一个struct并且映射为某个Id，则可以在查询时自动调用Get方法查询出对应的数据。

Get方法

查询单条数据使用 `Get` 方法，在调用Get方法时需要传入一个对应结构体的指针，同时结构体中的非空field自动成为查询的条件和前面的方法条件组合在一起查询。

如：

1）根据Id来获得单条数据：

```
1. user := new(User)
2. has, err := engine.Id(id).Get(user)
3. // 复合主键的获取方法
4. // has, errr := engine.Id(xorm.PK{1,2}).Get(user)
```

2）根据Where来获得单条数据：

```
1. user := new(User)
2. has, err := engine.Where("name=?", "xlw").Get(user)
```

3）根据user结构体中已有的非空数据来获得单条数据：

```
1. user := &User{Id:1}
2. has, err := engine.Get(user)
```

或者其它条件

```
1. user := &User{Name:"xlw"}
2. has, err := engine.Get(user)
```

返回的结果为两个参数，一个 `has` 为该条记录是否存在，第二个参数 `err` 为是否有错误。不管err是否为nil，has都有可能为true或者false。

Exist系列方法

判断某个记录是否存在可以使用 `Exist` ， 相比 `Get` ， `Exist` 性能更好。

```
has, err := testEngine.Exist(new(RecordExist))
// SELECT * FROM record_exist LIMIT 1
```

```
has, err = testEngine.Exist(&RecordExist{
        Name: "test1",
    })
// SELECT * FROM record_exist WHERE name = ? LIMIT 1
```

```
has, err = testEngine.Where("name = ?", "test1").Exist(&RecordExist{})
// SELECT * FROM record_exist WHERE name = ? LIMIT 1
```

```
has, err = testEngine.SQL("select * from record_exist where name = ?", "test1").Exist()
// select * from record_exist where name = ?
```

```
has, err = testEngine.Table("record_exist").Exist()
// SELECT * FROM record_exist LIMIT 1
```

```
has, err = testEngine.Table("record_exist").Where("name = ?", "test1").Exist()
// SELECT * FROM record_exist WHERE name = ? LIMIT 1
```

与Get的区别

Get与Exist方法返回值都为bool和error，如果查询到实体存在，则Get方法会将查到的实体赋值给参数

```
user := &User{Id:1}
has,err := testEngine.Get(user)    // 执行结束后，user会被赋值为数据库中Id为1的实体
has,err = testEngine.Exist(user)    // user中仍然是初始声明的user，不做改变
```

建议

如果你的需求是：判断某条记录是否存在，若存在，则返回这条记录。

建议直接使用Get方法。

如果仅仅判断某条记录是否存在，则使用Exist方法，Exist的执行效率要比Get更高。

Find方法

查询多条数据使用 `Find` 方法，Find方法的第一个参数为 `slice` 的指针或 `Map` 指针，即为查询后返回的结果，第二个参数可选，为查询的条件struct的指针。

1）传入Slice用于返回数据

```
everyone := make([]Userinfo, 0)
err := engine.Find(&everyone)

pEveryOne := make([]*Userinfo, 0)
err := engine.Find(&pEveryOne)
```

2）传入Map用户返回数据，map必须为 `map[int64]Userinfo` 的形式，map的key为id，因此对于复合主键无法使用这种方式。

```
users := make(map[int64]Userinfo)
err := engine.Find(&users)

pUsers := make(map[int64]*Userinfo)
err := engine.Find(&pUsers)
```

3）也可以加入各种条件

```
users := make([]Userinfo, 0)
err := engine.Where("age > ? or name = ?", 30, "xlw").Limit(20, 10).Find(&users)
```

4）如果只选择单个字段，也可使用非结构体的Slice

```
var ints []int64
err := engine.Table("user").Cols("id").Find(&ints)
```

Join的使用

- Join(string,interface{},string)

第一个参数为连接类型，当前支持INNER, LEFT OUTER, CROSS中的一个值，第二个参数为string类型的表名，表对应的结构体指针或者为两个值的[]string，表示表名和别名，第三个参数为连接条件。

以下将通过示例来讲解具体的用法：

假如我们拥有两个表user和group，每个User只在一个Group中，那么我们可以定义对应的struct

```
type Group struct {
    Id int64
    Name string
}
```

```
type User struct {
    Id int64
    Name string
    GroupId int64 `xorm:"index"`
}
```

OK。问题来了，我们现在需要列出所有的User，并且列出对应的GroupName。利用extends和Join我们可以比较优雅的解决这个问题。代码如下：

```
type UserGroup struct {
    User `xorm:"extends"`
    Name string
}

func (UserGroup) TableName() string {
    return "user"
}

users := make([]UserGroup, 0)
engine.Join("INNER", "group", "group.id = user.group_id").Find(&users)
```

这里我们将User这个匿名结构体加了xorm的extends标记（实际上也可以是非匿名的结构体，只要有extends标记即可），这样就减少了重复代码的书写。实际上这里我们直接用Sql函数也是可以的，并不一定非要用Join。

```
users := make([]UserGroup, 0)
engine.Sql("select user.*, group.name from user, group where user.group_id = group.id").Find(&users)
```

然后，我们忽然发现，我们还需要显示Group的Id，因为我们需要链接到Group页面。这样又要加一个字段，算了，不如我们把Group也加个extends标记吧，代码如下：

```
type UserGroup struct {
```

```
    User `xorm:"extends"`
    Group `xorm:"extends"`
}

func (UserGroup) TableName() string {
    return "user"
}

users := make([]UserGroup, 0)
engine.Join("INNER", "group", "group.id = user.group_id").Find(&users)
```

这次，我们把两个表的所有字段都查询出来了，并且赋值到对应的结构体上了。

这里要注意，User和Group分别有Id和Name，这个是重名的，但是xorm是可以区分开来的，不过需要特别注意UserGroup中User和Group的顺序，如果顺序反了，则有可能会赋值错误，但是程序不会报错。

这里的顺序应遵循如下原则：

```
结构体中extends标记对应的结构顺序应和最终生成SQL中对应的表出现的顺序相同。
```

还有一点需要注意的，如果在模板中使用这个UserGroup结构体，对于字段名重复的必须加匿名引用，如：

对于不重复字段，可以 `{{.GroupId}}` ，对于重复字段 `{{.User.Id}}` 和 `{{.Group.Id}}`

这是2个表的用法，3个或更多表用法类似，如：

```
type Type struct {
    Id int64
    Name string
}

type UserGroupType struct {
    User `xorm:"extends"`
    Group `xorm:"extends"`
    Type `xorm:"extends"`
}

users := make([]UserGroupType, 0)
engine.Table("user").Join("INNER", "group", "group.id = user.group_id").
    Join("INNER", "type", "type.id = user.type_id").
    Find(&users)
```

同时，在使用Join时，也可同时使用Where和Find的第二个参数作为条件，Find的第二个参数同时也允许为各种bean来作为条件。Where里可以是各个表的条件，Find的第二个参数只是被关联表的条件。

```
engine.Table("user").Join("INNER", "group", "group.id = user.group_id").
    Join("INNER", "type", "type.id = user.type_id").
    Where("user.name like ?", "%"+name+"%").Find(&users, &User{Name:name})
```

当然，如果表名字太长，我们可以使用别名：

```
engine.Table("user").Alias("u").
    Join("INNER", []string{"group", "g"}, "g.id = u.group_id").
    Join("INNER", "type", "type.id = u.type_id").
    Where("u.name like ?", "%"+name+"%").Find(&users, &User{Name:name})
```

Iterate方法

Iterate方法提供逐条执行查询到的记录的方法，他所能使用的条件和Find方法完全相同

```
    err := engine.Where("age > ? or name=?)", 30, "xlw").Iterate(new(Userinfo), func(i int, bean
1. interface{})error{
2.     user := bean.(*Userinfo)
3.     //do somthing use i and user
4. })
```

Count方法

统计数据使用 `Count` 方法，Count方法的参数为struct的指针并且成为查询条件。

```
1. user := new(User)
2. total, err := engine.Where("id >?", 1).Count(user)
```

Rows方法

Rows方法和Iterate方法类似，提供逐条执行查询到的记录的方法，不过Rows更加灵活好用。

```
user := new(User)
rows, err := engine.Where("id >?", 1).Rows(user)
if err != nil {
}
defer rows.Close()
for rows.Next() {
    err = rows.Scan(user)
    //...
}
```

Sum系列方法

求和数据可以使用 `Sum` , `SumInt` , `Sums` 和 `SumsInt` 四个方法，Sums系列方法的参数为struct的指针并且成为查询条件。

- Sum 求某个字段的和，返回float64

```
type SumStruct struct {
    Id int64
    Money int
    Rate float32
}

ss := new(SumStruct)
total, err := engine.Where("id >?", 1).Sum(ss, "money")
fmt.Printf("money is %d", int(total))
```

- SumInt 求某个字段的和，返回int64

```
type SumStruct struct {
    Id int64
    Money int
    Rate float32
}

ss := new(SumStruct)
total, err := engine.Where("id >?", 1).SumInt(ss, "money")
fmt.Printf("money is %d", total)
```

- Sums 求某几个字段的和， 返回float64的Slice

```
ss := new(SumStruct)
totals, err := engine.Where("id >?", 1).Sums(ss, "money", "rate")

fmt.Printf("money is %d, rate is %.2f", int(total[0]), total[1])
```

- SumsInt 求某几个字段的和， 返回int64的Slice

```
ss := new(SumStruct)
totals, err := engine.Where("id >?", 1).SumsInt(ss, "money")

fmt.Printf("money is %d", total[0])
```

更新数据

更新数据使用 `Update` 方法，Update方法的第一个参数为需要更新的内容，可以为一个结构体指针或者一个Map[string]interface{}类型。当传入的为结构体指针时，只有非空和0的field才会被作为更新的字段。当传入的为Map类型时，key为数据库Column的名字，value为要更新的内容。

`Update` 方法将返回两个参数，第一个为 更新的记录数，需要注意的是 `SQLITE` 数据库返回的是根据更新条件查询的记录数而不是真正受更新的记录数。

```
1. user := new(User)
2. user.Name = "myname"
3. affected, err := engine.Id(id).Update(user)
```

这里需要注意，Update会自动从user结构体中提取非0和非nil得值作为需要更新的内容，因此，如果需要更新一个值为0，则此种方法将无法实现，因此有两种选择：

- 1.通过添加Cols函数指定需要更新结构体中的哪些值，未指定的将不更新，指定了的即使为0也会更新。

```
1. affected, err := engine.Id(id).Cols("age").Update(&user)
```

- 2.通过传入map[string]interface{}来进行更新，但这时需要额外指定更新到哪个表，因为通过map是无法自动检测更新哪个表的。

```
1. affected, err := engine.Table(new(User)).Id(id).Update(map[string]interface{}{"age":0})
```

乐观锁Version

要使用乐观锁，需要使用version标记

```
type User struct {
    Id int64
    Name string
    Version int `xorm:"version"`
}
```

在Insert时，version标记的字段将会被设置为1，在Update时，Update的内容必须包含version原来的值。

```
var user User
engine.Id(1).Get(&user)
// SELECT * FROM user WHERE id = ?
engine.Id(1).Update(&user)
// UPDATE user SET ..., version = version + 1 WHERE id = ? AND version = ?
```

更新时间Updated

Updated可以让您在记录插入或每次记录更新时自动更新数据库中的标记字段为当前时间，需要在xorm标记中使用updated标记，如下所示进行标记，对应的字段可以为time.Time或者自定义的time.Time或者int,int64等int类型。

```
type User struct {
    Id int64
    Name string
    UpdatedAt time.Time `xorm:"updated"`
}
```

在Insert(), InsertOne(), Update()方法被调用时，updated标记的字段将会被自动更新为当前时间，如下所示：

```
var user User
engine.Id(1).Get(&user)
// SELECT * FROM user WHERE id = ?
engine.Id(1).Update(&user)
// UPDATE user SET ..., updaetd_at = ? WHERE id = ?
```

如果你希望临时不自动插入时间，则可以组合NoAutoTime()方法：

```
engine.NoAutoTime().Insert(&user)
```

这个在从一张表拷贝字段到另一张表时比较有用。

删除数据

删除数据 `Delete` 方法，参数为struct的指针并且成为查询条件。

```
1. user := new(User)
2. affected, err := engine.Id(id).Delete(user)
```

`Delete` 的返回值第一个参数为删除的记录数，第二个参数为错误。

注意：当删除时，如果user中包含有bool,float64或者float32类型，有可能会使删除失败。具体请查看 FAQ

软删除Deleted

Deleted可以让您不真正的删除数据，而是标记一个删除时间。使用此特性需要在xorm标记中使用deleted标记，如下所示进行标记，对应的字段必须为time.Time类型。

```
type User struct {
    Id int64
    Name string
    DeletedAt time.Time `xorm:"deleted"`
}
```

在Delete()时，deleted标记的字段将会被自动更新为当前时间而不是去删除该条记录，如下所示：

```
var user User
engine.Id(1).Get(&user)
// SELECT * FROM user WHERE id = ?
engine.Id(1).Delete(&user)
// UPDATE user SET ..., deleted_at = ? WHERE id = ?
engine.Id(1).Get(&user)
// 再次调用Get，此时将返回false, nil，即记录不存在
engine.Id(1).Delete(&user)
// 再次调用删除会返回0, nil，即记录不存在
```

那么如果记录已经被标记为删除后，要真正的获得该条记录或者真正的删除该条记录，需要启用Unscoped，如下所示：

```
var user User
engine.Id(1).Unscoped().Get(&user)
// 此时将可以获得记录
engine.Id(1).Unscoped().Delete(&user)
// 此时将可以真正的删除记录
```

执行SQL查询

Query

也可以直接执行一个SQL查询，即Select命令。在Postgres中支持原始SQL语句中使用 ` 和 ？ 符号。

```
1. sql := "select * from userinfo"
2. results, err := engine.Query(sql)
```

当调用 `Query` 时，第一个返回值 `results` 为 `[]map[string][]byte` 的形式。

`Query` 的参数也允许传入 `*builder.Buidler` 对象

```
1. // SELECT * FROM table
2. results, err := engine.Query(builder.Select("*").From("table"))
```

QueryInterface

和 `Query` 类似，但是返回值为 `[]map[string]interface{}`

QueryString

和 `Query` 类似，但是返回值为 `[]map[string]string`

执行SQL命令

也可以直接执行一个SQL命令，即执行Insert， Update， Delete 等操作。此时不管数据库是何种类型，都可以使用 ` 和 ? 符号。

```
sql = "update `userinfo` set username=? where id=?"
res, err := engine.Exec(sql, "xiaolun", 1)
```

事务处理

当使用事务处理时，需要创建Session对象。在进行事物处理时，可以混用ORM方法和RAW方法，如下代码所示：

```
session := engine.NewSession()
defer session.Close()
// add Begin() before any action
err := session.Begin()
user1 := Userinfo{Username: "xiaoxiao", Departname: "dev", Alias: "lunny", Created: time.Now()}
_, err = session.Insert(&user1)
if err != nil {
    session.Rollback()
    return
}
user2 := Userinfo{Username: "yyy"}
_, err = session.Where("id = ?", 2).Update(&user2)
if err != nil {
    session.Rollback()
    return
}

_, err = session.Exec("delete from userinfo where username = ?", user2.Username)
if err != nil {
    session.Rollback()
    return
}

// add Commit() after all actions
err = session.Commit()
if err != nil {
    return
}
```

- 注意如果您使用的是mysql，数据库引擎为innodb事务才有效，myisam引擎是不支持事务的。

缓存

xorm内置了一致性缓存支持，不过默认并没有开启。要开启缓存，需要在engine创建完后进行配置，如：启用一个全局的内存缓存

```
1. cacher := xorm.NewLRUCacher(xorm.NewMemoryStore(), 1000)
2. engine.SetDefaultCacher(cacher)
```

上述代码采用了LRU算法的一个缓存，缓存方式是存放到内存中，缓存struct的记录数为1000条，缓存针对的范围是所有具有主键的表，没有主键的表中的数据将不会被缓存。如果只想针对部分表，则：

```
1. cacher := xorm.NewLRUCacher(xorm.NewMemoryStore(), 1000)
2. engine.MapCacher(&user, cacher)
```

如果要禁用某个表的缓存，则：

```
1. engine.MapCacher(&user, nil)
```

设置完之后，其它代码基本上就不需要改动了，缓存系统已经在后台运行。

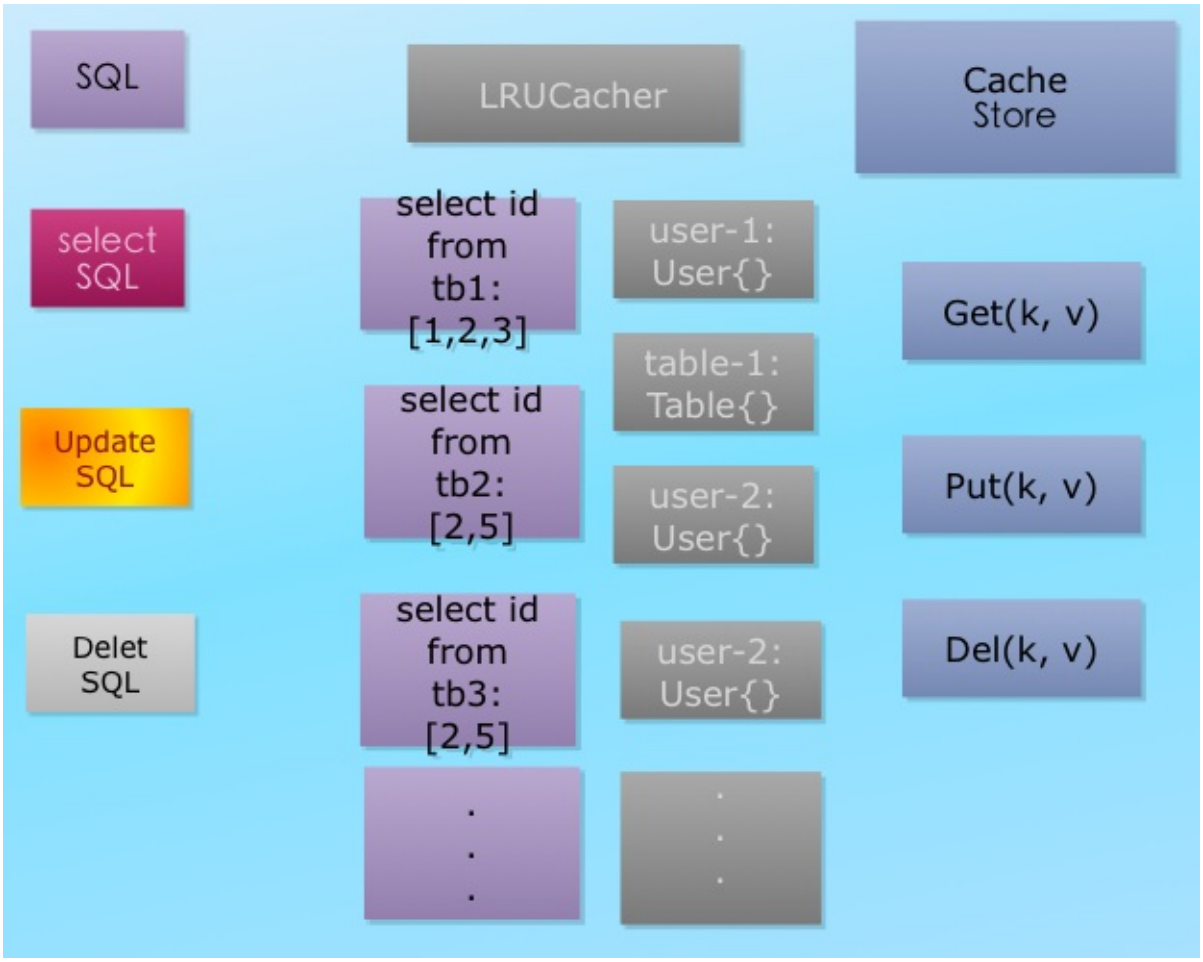
当前实现了内存存储的CacheStore接口MemoryStore，如果需要采用其它设备存储，可以实现CacheStore接口。

不过需要特别注意不适用缓存或者需要手动编码的地方：

1. 当使用了 `Distinct` ， `Having` ， `GroupBy` 方法将不会使用缓存
2. 在 `Get` 或者 `Find` 时使用了 `Cols` ， `Omit` 方法，则在开启缓存后此方法无效，系统仍旧会取出这个表中的所有字段。
3. 在使用Exec方法执行了方法之后，可能会导致缓存与数据库不一致的地方。因此如果启用缓存，尽量避免使用Exec。如果必须使用，则需要在使用了Exec之后调用ClearCache手动做缓存清除的工作。比如：

```
1. engine.Exec("update user set name = ? where id = ?", "xlw", 1)
2. engine.ClearCache(new(User))
```

缓存的实现原理如下图所示：


SQL
LRUCacher
Cache Store
select SQL
select id from tb1: [1,2,3]
user-1: User{}
Get(k, v)
table-1: Table{}
Update SQL
select id from tb2: [2,5]
user-2: User{}
Put(k, v)
Delet SQL
select id from tb3: [2,5]
user-2: User{}
Del(k, v)

事件

xorm支持两种方式的事件，一种是在Struct中的特定方法来作为事件的方法，一种是在执行语句的过程中执行事件。

在Struct中作为成员方法的事件如下：

- BeforeInsert()

在将此struct插入到数据库之前执行

- BeforeUpdate()

在将此struct更新到数据库之前执行

- BeforeDelete()

在将此struct对应的条件数据从数据库删除之前执行

- `func BeforeSet(name string, cell xorm.Cell)`

在 Get 或 Find 方法中，当数据已经从数据库查询出来，而在设置到结构体之前调用，name为数据库字段名称，cell为数据库中的字段值。

- `func AfterSet(name string, cell xorm.Cell)`

在 Get 或 Find 方法中，当数据已经从数据库查询出来，而在设置到结构体之后调用，name为数据库字段名称，cell为数据库中的字段值。

- AfterInsert()

在将此struct成功插入到数据库之后执行

- AfterUpdate()

在将此struct成功更新到数据库之后执行

- AfterDelete()

在将此struct对应的条件数据成功从数据库删除之后执行

在语句执行过程中的事件方法为：

- Before(beforeFunc interface{})

临时执行某个方法之前执行

```
1. before := func(bean interface{}){
2.     fmt.Println("before", bean)
3. }
4. engine.Before(before).Insert(&obj)
```

- After(afterFunc interface{})

临时执行某个方法之后执行

```
after := func(bean interface{}){
    fmt.Println("after", bean)
}
engine.After(after).Insert(&obj)
```

其中beforeFunc和afterFunc的原型为func(bean interface{}).

xorm 工具

xorm 是一组数据库操作命令行工具。

源码安装

```
1. go get xorm.io/cmd/xorm
```

同时你需要安装如下依赖：

- github.com/go-xorm/xorm
- Mysql: github.com/go-sql-driver/mysql
- MyMysql: github.com/ziutek/mymysql/godrv
- Postgres: github.com/lib/pq
- SQLite: github.com/mattn/go-sqlite3

** 对于sqlite3的支持，你需要自己进行编译 `go build -tags sqlite3` 因为sqlite3需要cgo的支持。

命令列表

有如下可用的命令：

- **reverse** 反转一个数据库结构，生成代码
- **shell** 通用的数据库操作客户端，可对数据库结构和数据操作
- **dump** Dump数据库中所有结构和数据到标准输出
- **source** 从标注输入中执行SQL文件
- **driver** 列出所有支持的数据库驱动

reverse

Reverse command is a tool to convert your database struct to all kinds languages of structs or classes. After you installed the tool, you can type

`xorm help reverse`

to get help

example:

sqlite: `xorm reverse sqite3 test.db templates/goxorm`

mysql: `xorm reverse mysql root:@/xorm_test?charset=utf8 templates/goxorm`

mymysql: `xorm reverse mymysql xorm_test2/root/ templates/goxorm`

postgres: `xorm reverse postgres "dbname=xorm_test sslmode=disable" templates/goxorm`

will generated go files in `./model` directory

Template and Config

Now, xorm tool supports go and c++ two languages and have go, goxorm, c++ three of default templates. In template directory, we can put a config file to control how to generating.

```
1. lang=go
2. genJson=1
```

lang must be go or c++ now.genJson can be 1 or 0, if 1 then the struct will have json tag.

Shell

Shell command provides a tool to operate database. For example, you can create table, alter table, insert data, delete data and etc.

`xorm shell sqlite3 test.db` will connect to the sqlite3 database and you can type `help` to list all the shell commands.

Dump

Dump command provides a tool to dump all database structs and data as SQL to your standard output.

`xorm dump sqlite3 test.db` could dump sqlite3 database test.db to standard output. If you want to save to file, justtype `xorm dump sqlite3 test.db > test.sql` .

Source

`xorm source sqlite3 test.db < test.sql` will execute sql file on the test.db.

Driver

List all supported drivers since default build will not include sqlite3.

LICENSE

BSD License http://creativecommons.org/licenses/BSD/

常见问题

- 如何使用Like？

答：

```
1. engine.Where("column like ?", "%"+char+"%").Find
```

- 怎么同时使用xorm的tag和json的tag？

答：使用空格

```
1. type User struct {
2.     Name string `json:"name" xorm:"name"`
3. }
```

- 我的struct里面包含bool类型，为什么它不能作为条件也没法用Update更新？

答：默认bool类型因为无法判断是否为空，所以不会自动作为条件也不会作为Update的内容。可以使用UseBool函数，也可以使用Cols函数

```
1. engine.Cols("bool_field").Update(&Struct{BoolField:true})
2. // UPDATE struct SET bool_field = true
```

- 我的struct里面包含float64和float32类型，为什么用他们作为查询条件总是不正确？

答：默认float32和float64映射到数据库中为float,real,double这几种类型，这几种数据库类型数据库的实现一般都是非精确的。因此作为相等条件查询有可能不会返回正确的结果。如果一定要作为查询条件，请将数据库中的类型定义为Numeric或者Decimal。

```
1. type account struct {
2. money float64 `xorm:"Numeric"`
3. }
```

- 为什么Update时Sqlite3返回的affected和其它数据库不一样？

答：Sqlite3默认Update时返回的是update的查询条件的记录数条数，不管记录是否真的有更新。而Mysql和Postgres默认情况下都是只返回记录中有字段改变的记录数。

- xorm有几种命名映射规则？

答：目前支持SnakeMapper, SameMapper和GonicMapper三种。SnakeMapper支持结构体和成员以驼峰式命名而数据库表和字段以下划线连接命名；SameMapper支持结构体和数据库的命名保持一致的映射。GonicMapper在SnakeMapper的基础上对一些特定名词，比如ID的映射会映射为id，而不是像SnakeMapper那样为i_d。

- xorm支持复合主键吗？

答：支持。在定义时，如果有多个字段标记了pk，则这些字段自动成为复合主键，顺序为在struct中出现的顺序。在使用Id方法时，可以用 `Id(xorm.PK{1, 2})` 的方式来用。

- xorm如何使用Join？

答：一般我们配合Join()和extends标记来进行，比如我们要对两个表进行Join操作，我们可以这样：

```
type Userinfo struct {
    Id int64
    Name string
    DetailId int64
}

type Userdetail struct {
    Id int64
    Gender int
}

type User struct {
    Userinfo `xorm:"extends"`
    Userdetail `xorm:"extends"`
}

var users = make([]User, 0)
err := engine.Table(&Userinfo{}).Join("LEFT", "userdetail", "userinfo.detail_id = userdetail.id").Find(&users)
```

请注意这里的Userinfo在User中的位置必须在Userdetail的前面，因为他在join语句中的顺序在userdetail前面。如果顺序不对，那么对于同名的列，有可能会赋值出错。

当然，如果Join语句比较复杂，我们也可以直接用Sql函数

```
err := engine.Sql("select * from userinfo, userdetail where userinfo.detail_id = userdetail.id").Find(&users)
```

- 如果有自动增长的字段，Insert如何写？答：Insert时，如果需要自增字段填充为自动增长的数值，请保持自增字段为0；如果自增字段为非0，自增字段将会被作为普通字段插入。
- 如果设置数据库时区？答：

```
location, err = time.LoadLocation("Asia/Shanghai")
engine.TZLocation = location
```

案例

- Gogs - github.com/gogits/gogs
- Gowalker - github.com/Unknwon/gowalker
- Gobuild.io - github.com/shxsun/gobuild
- Sudo China - github.com/insionng/toropress
- Godaily - github.com/govc/godaily
- GoCMS - github.com/zzboy/GoCMS
- GoBBS - gobbs.domolo.com
- go-blog - github.com/easykoo/go-blog

更新日志

- **v0.6.5**
    - 通过 engine.SetSchema 来支持 schema，当前仅支持Postgres
    - vgo 支持
    - 新增 FindAndCount 函数
    - 通过 NewEngineWithParams 支持数据库特别参数
    - 修正部分Bug
- **v0.6.4**
    - 自动读写分离支持
    - Query/QueryString/QueryInterface 支持与 Where/And 合用
    - Get 支持获取非结构体变量
    - Iterate 支持 BufferSize
    - 修正部分Bug
- **v0.6.3**
    - 合并单元测试到主工程
    - 新增 `Exist` 方法
    - 新增 `SumInt` 方法
    - Mysql新增读取和创建字段注释支持
    - 新增 `SetConnMaxLifetime` 方法
    - 修正了时间相关的Bug
    - 修复了一些其它Bug
- **v0.6.2**
    - 重构Tag解析方式
    - Get方法新增类似Scan的特性
    - 新增 QueryString 方法
- **v0.6.0**
    - 去除对 ql 的支持
    - 新增条件查询分析器 github.com/go-xorm/builder，从因此 `Where, And, Or` 函数将可以用 `builder.Cond` 作为条件组合
    - 新增 Sum, SumInt, SumInt64 和 NotIn 函数
    - Bug修正
- **v0.5.0**
    - logging接口进行不兼容改变
    - Bug修正
- **v0.4.5**

    - bug修正
    - extends 支持无限级
    - Delete Limit 支持

- **v0.4.4**

    - Tidb 数据库支持
    - QL 试验性支持
    - sql.NullString支持
    - ForUpdate 支持
    - bug修正

- **v0.4.3**

    - Json 字段类型支持
    - oracle实验性支持
    - bug修正

- **v0.4.2**

    - 事物如未Rollback或Commit，在关闭时会自动Rollback
    - Gonic 映射支持
    - bug修正

- **v0.4.1**

    - 添加deleted标记作为软删除.

- **v0.4.0 RC1** 新特性:

    - 移动xorm cmd github.com/go-xorm/cmd
    - 在重构一般DB操作核心库 github.com/go-xorm/core
    - 移动测试github.com/XORM/tests github.com/go-xorm/tests

      改进:

    - Prepared statement 缓存
    - 添加 Incr API
    - 指定时区位置

- **v0.3.2** 改进:

    - Add AllCols & MustCols function
    - Add TableName for custom table name

      Bug 修复:

    - 46

- 51
- 53
- 89
- 86
- 92

- **v0.3.1**

  新特性:

  - 支持 MSSQL DB 通过 ODBC 驱动 (github.com/lunny/godbc);
  - 通过多个pk标记支持联合主键;
  - 新增 Rows() API 用来遍历查询结果，该函数提供了类似sql.Rows的相似用法，可作为 Iterate() API 的可选替代;
  - ORM 结构体现在允许内建类型的指针作为成员，使得数据库为null成为可能;
  - Before 和 After 支持

  改进:

  - 允许 int/int32/int64/uint/uint32/uint64/string 作为主键类型
  - 查询函数 Get()/Find()/Iterate() 在性能上的改进

- **v0.2.3** : 改善了文档;提供了乐观锁支持;添加了带时区时间字段支持;Mapper现在分成表名Mapper和字段名Mapper，同时实现了表或字段的自定义前缀后缀;Insert方法的返回值含义从id, err更改为affected, err，请大家注意;添加了UseBool 和 Distinct函数。

- **v0.2.2** : Postgres驱动新增了对lib/pq的支持;新增了逐条遍历方法Iterate;新增了SetMaxConns(go1.2+)支持，修复了bug若干;
- **v0.2.1** : 新增数据库反转工具，当前支持go和c++代码的生成，详见 Xorm Tool README; 修复了一些bug.
- **v0.2.0** : 新增 缓存支持，查询速度提升3-5倍; 新增数据库表和Struct同名的映射方式; 新增Sync同步表结构;
- **v0.1.9** : 新增 postgres 和 mymysql 驱动支持; 在Postgres中支持原始SQL语句中使用 ` 和 ? 符号; 新增Cols, StoreEngine, Charset 函数;SQL语句打印支持io.Writer接口，默认打印到控制台;新增更多的字段类型支持，详见 映射规则;删除废弃的MakeSession和Create函数。
- **v0.1.8** : 新增联合index，联合unique支持，请查看 映射规则。
- **v0.1.7** : 新增IConnectPool接口以及NoneConnectPool, SysConnectPool, SimpleConnectPool三种实现，可以选择不使用连接池，使用系统连接池和使用自带连接池三种实现，默认为SysConnectPool,

即系统自带的连接池。同时支持自定义连接池。Engine新增Close方法，在系统退出时应调用此方法。

- **v0.1.6** ： 新增Conversion，支持自定义类型到数据库类型的转换；新增查询结构体自动检测匿名成员支持；新增单向映射支持；
- **v0.1.5** ： 新增对多线程的支持；新增Sql()函数；支持任意sql语句的struct查询；Get函数返回值变动；MakeSession和Create函数被NewSession和NewEngine函数替代；
- **v0.1.4** ： Get函数和Find函数新增简单的级联载入功能；对更多的数据库类型支持。
- **v0.1.3** ： Find函数现在支持传入Slice或者Map，当传入Map时，key为id；新增Table函数以为多表和临时表进行支持。
- **v0.1.2** ： Insert函数支持混合struct和slice指针传入，并根据数据库类型自动批量插入，同时自动添加事务
- **v0.1.1** ： 添加 Id, In 函数，改善 README 文档
- **v0.1.0** ： 初始化工程s