



下载APP



26 | GORM : 数据库的使用必不可少 (下)

2021-11-15 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >



讲述：叶剑峰

时长 17:25 大小 15.97M




你好，我是轩脉刃。

上一节课，我们梳理了 Gorm 的核心逻辑，也通过思维导图，详细分析了 Gorm 的源码搞清楚它是如何封装 database/sql 的。这节课我们就要思考和操作，如何将 Gorm 融合进入 hade 框架了。

Gorm 的使用分为两个部分，数据库的连接和数据库的操作。

对于数据库操作接口的封装，Gorm 已经做的非常好了，它在 gorm.DB 中定义了非常对数据库的操作接口，这些接口已经是非常易用了，而且每个操作接口在 [官方文档](#) 中都有对应的说明和使用教程。比如在 DB 的操作接口列表中，我们可以看到常用的增删改查的逻辑：



 复制代码

```
1 func (db *DB) Create(value interface{}) (tx *DB)
2
3 func (db *DB) Delete(value interface{}, conds ...interface{}) (tx *DB)
4
5 func (db *DB) Get(key string) (interface{}, bool)
6
7 func (db *DB) Update(column string, value interface{}) (tx *DB)
```

同时，[官方首页](#)的例子也把获取到 DB 后的增删改查操作显示很清楚了，建议你在浏览器收藏这个 Gorm 的说明文档，因为在具体的应用开发中，你会经常参考使用它的。


所以今天我们要做的事情，就是封装 Gorm 的数据库连接部分。

ORM 服务

按照“一切皆服务”的思想，我们也计划将 Gorm 封装为一个服务。而服务三要素是服务接口、服务提供者、服务实例化。我们先来定义 ORM 服务接口。

服务接口


这个服务接口并不复杂，它的唯一任务就是能够初始化出 gorm.DB 实例。回顾上节课说的 Gorm 初始化 gorm.DB 的方法：

 复制代码

```
1 dsn := "xxxxxxx"
2 db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
```

参数看起来就这两个部分 DSN 和 gorm.Config。

不过我们希望设计一个 hade 框架自定义的配置结构，将所有创建连接需要的配置项整合起来。所以除了 DSN 和 gorm.Config 这两个配置项，其实还需要加上连接池的配置，就是上节课说的 database/sql 中提供的对连接池的配置信息。再回顾一下这四个影响底层创建连接池设置的配置信息：

 复制代码

```
1 // 设置连接的最大空闲时长
```

```

2 func (db *DB) SetConnMaxIdleTime(d time.Duration)
3 // 设置连接的最大生命时长
4 func (db *DB) SetConnMaxLifetime(d time.Duration)
5 // 设置最大空闲连接数
6 func (db *DB) SetMaxIdleConns(n int)
7 // 设置最大打开连接数
8 func (db *DB) SetMaxOpenConns(n int)

```

所以可以定义这么一个 DBConfig 结构，将所有的创建 DB 相关的配置都放在这里面。代码在 framework/contract/orm.go 中：

[复制代码](#)

```


1 // DBConfig 代表数据库连接的所有配置
2 type DBConfig struct {
3     // 以下配置关于dsn
4     WriteTimeout string `yaml:"write_timeout"` // 写超时时间
5     Loc           string `yaml:"loc"`           // 时区
6     Port          int    `yaml:"port"`          // 端口
7     ReadTimeout   string `yaml:"read_timeout"`   // 读超时时间
8     Charset       string `yaml:"charset"`       // 字符集
9     ParseTime     bool  `yaml:"parse_time"`    // 是否解析时间
10    Protocol       string `yaml:"protocol"`      // 传输协议
11    Dsn            string `yaml:"dsn"`           // 直接传递dsn，如果传递了，其他关于d
12    Database       string `yaml:"database"`      // 数据库
13    Collation      string `yaml:"collation"`     // 字符序
14    Timeout        string `yaml:"timeout"`       // 连接超时时间
15    Username       string `yaml:"username"`      // 用户名
16    Password       string `yaml:"password"`      // 密码
17    Driver         string `yaml:"driver"`        // 驱动
18    Host           string `yaml:"host"`          // 数据库地址
19
20    // 以下配置关于连接池
21    ConnMaxIdle     int    `yaml:"conn_max_idle"` // 最大空闲连接数
22    ConnMaxOpen     int    `yaml:"conn_max_open"` // 最大连接数
23    ConnMaxLifetime string `yaml:"conn_max_lifetime"` // 连接最大生命周期
24    ConnMaxIdletime string `yaml:"conn_max_idletime"` // 空闲最大生命周期
25
26    // 以下配置关于gorm
27    *gorm.Config // 集成gorm的配置
28 }

```

其中 DSN 是一个复杂的字符串。但我们又不希望使用者直接设置这些复杂字符串来进行传递，所以这里设置了多个字段来生成这个 DSN。

另外上节课也说过，DSN 并没有一个标准的格式约定，不同的数据库可能有不同的解析，所以也同时保留直接设置 DSN 的权限，如果用户手动设置了 Dsn 字段，那么其他关于 Dsn 的字段设置均无效。

所以这里同时需要实现一个方法，使用 DBConfig 来生成最终使用的字符串 Dsn，使用上节课介绍的 github.com/go-sql-driver/mysql 库，就能很方便地实现了。我们继续写：

 复制代码

```
1 import (
2     "github.com/go-sql-driver/mysql"
3     ...
4 )
5
6 // FormatDsn 生成dsn
7 func (conf *DBConfig) FormatDsn() (string, error) {
8     port := strconv.Itoa(conf.Port)
9     timeout, err := time.ParseDuration(conf.Timeout)
10    if err != nil {
11        return "", err
12    }
13    readTimeout, err := time.ParseDuration(conf.ReadTimeout)
14    if err != nil {
15        return "", err
16    }
17    writeTimeout, err := time.ParseDuration(conf.WriteTimeout)
18    if err != nil {
19        return "", err
20    }
21    location, err := time.LoadLocation(conf.Loc)
22    if err != nil {
23        return "", err
24    }
25    driverConf := &mysql.Config{
26        User:      conf.Username,
27        Passwd:    conf.Password,
28        Net:       conf.Protocol,
29        Addr:      net.JoinHostPort(conf.Host, port),
30        DBName:    conf.Database,
31        Collation: conf.Collation,
32        Loc:       location,
33        Timeout:   timeout,
34        ReadTimeout: readTimeout,
35        WriteTimeout: writeTimeout,
36        ParseTime: conf.ParseTime,
37    }
38    return driverConf.FormatDSN(), nil
39 }
```


可以看到 Gorm 配置，我们使用结构嵌套的方式，将 `gorm.Config` 直接嵌套进入 `DBConfig` 中。你可以琢磨下这种写法，它有两个好处。

一是可以直接设置 `DBConfig` 来设置 `gorm.Config`。比如这个函数是可行的，它直接设置 `config.DryRun`，就是直接设置 `gorm.Config`：

```
1 func(container framework.Container, config *contract.DBConfig) error {
2     config.DryRun = true
3     return nil
4 }
```

[复制代码](#)

二是 `DBConfig` 继承了 `*gorm.Config` 的所有方法。比如这段代码，我们来理解一下：

```
1 config := &contract.DBConfig{}
2 db, err = gorm.Open(mysql.Open(config.Dsn), config)
```

[复制代码](#)

还记得 `gorm.Open` 的第二个参数是 `Option` 么，它是一个接口，需要实现 `Apply` 和 `AfterInitialize` 方法，而我们的 `DBConfig` 并没有显式实现这两个方法。但是它嵌套了实现了这两个方法的 `*gorm.Config`，所以，默认 `DB.Config` 也就实现了这两个方法。

```
1 type Option interface {
2     Apply(*Config) error
3     AfterInitialize(*DB) error
4 }
```

[复制代码](#)

现在，`gorm.Open` 的两个参数 `DSN` 和 `gorm.Config` 都封装在 `DBConfig` 中，而修改 `DBConfig` 的方法，我们封装为 `DBOption`。

如何让设置 `DBOption` 的方法更为优雅呢？这里就使用到上节课刚学到的 `Option` 可变参数的编程方法了。定义一个 `DBOption` 的结构，它代表一个可以对 `DBConfig` 进行设置的

方法，这个结构作为获取 ORM 服务 GetDB 方法的参数。在 framework/contract/orm.go 中：

[复制代码](#)

```
1 package contract
2
3 // ORMKey 代表 ORM的服务
4 const ORMKey = "hade:orm"
5
6 // ORMService 表示传入的参数
7 type ORMService interface {
8     GetDB(option ...DBOption) (*gorm.DB, error)
9 }
10
11 // DBOption 代表初始化的时候的选项
12 type DBOption func(container framework.Container, config *DBConfig) error
```

这样就能通过设置不同的方法来对 DBConfig 进行配置。

比如要设置 DBConfig 中 gorm.Config 的 DryRun 空跑字段，设计了这么一个方法在 framework/provider/orm/config.go 中：

[复制代码](#)

```
1 // WithDryRun 设置空跑模式
2 func WithDryRun() contract.DBOption {
3     return func(container framework.Container, config *contract.DBConfig) error {
4         config.DryRun = true
5         return nil
6     }
7 }
```

之后，在使用 ORM 服务的时候，我们就可以这样设置：

[复制代码](#)

```
1 gormService := c.MustMake(contract.ORMKey).(contract.ORMService)
2 // 可变参数为WithDryRun()
3 db, err := gormService.GetDB(orm.WithDryRun())
```

服务提供者

下一步来完成服务提供者，我们也并不需要过于复杂的设计，只要注意一下两点：

ORM 服务一定是要延迟加载的，因为这个服务并不是一个基础服务。如果设置为非延迟加载，在框架启动的时候就会去建立这个服务，这并不是我们想要的。所以我们设计 ORM 的 provider 的时候，需要将 IsDefer 函数设置为 true。

第二点考虑到我们后续会使用 container 中的配置服务，来创建具体的 gorm.DB 实例，传递一个 container 是必要的。

所以具体的服务提供者代码如下，在 framework/provider/orm/provider.go 中：

 复制代码

```
1 package orm
2
3 import (
4     "github.com/gohade/hade/framework"
5     "github.com/gohade/hade/framework/contract"
6 )
7
8 // GormProvider 提供App的具体实现方法
9 type GormProvider struct {
10 }
11
12 // Register 注册方法
13 func (h *GormProvider) Register(container framework.Container) framework.NewIn
14     return NewHadeGorm
15 }
16
17 // Boot 启动调用
18 func (h *GormProvider) Boot(container framework.Container) error {
19     return nil
20 }
21
22 // IsDefer 是否延迟初始化
23 func (h *GormProvider) IsDefer() bool {
24     return true
25 }
26
27 // Params 获取初始化参数
28 func (h *GormProvider) Params(container framework.Container) []interface{} {
29     return []interface{}{container}
30 }
31
32 // Name 获取字符串凭证
33 func (h *GormProvider) Name() string {
34     return contract.ORMKey
35 }
```

服务实例化


服务实例化是今天的重点内容，我们先把 Gorm 的配置结构和日志结构的准备工作完成，再写稍微复杂一点的具体 ORM 服务的实例 HadeGorm。

配置

前面定义了 hade 框架专属的 DBConfig 配置结构，如何设置它是一个需要讲究的问题。

虽然已经设计了一种修改配置文件的方式，就是通过 GetDB 中的 Option 参数来设置。但是每个字段都这么设置又非常麻烦，我们自然会想到使用配置文件来配置这个结构。另外如果要连接多个数据库，每个数据库都进行同样的配置，还是颇为麻烦，是不是可以有个默认配置呢？

于是我们的配置文件可以这样设计：在 database.yaml 中保存数据库的默认值，如果想对某个数据库连接有单独的配置，可以用内嵌 yaml 结构的方式来进行配置。看下面这个配置例子：

 复制代码

```
1 conn_max_idle: 10 # 通用配置，连接池最大空闲连接数
2 conn_max_open: 100 # 通用配置，连接池最大连接数
3 conn_max_lifetime: 1h # 通用配置，连接数最大生命周期
4 protocol: tcp # 通用配置，传输协议
5 loc: Local # 通用配置，时区
6
7 default:
8     driver: mysql # 连接驱动
9     dsn: "" # dsn，如果设置了dsn，以下的所有设置都不生效
10    host: localhost # ip地址
11    port: 3306 # 端口
12    database: coredemo # 数据库
13    username: jianfengye # 用户名
14    password: "123456789" # 密码
15    charset: utf8mb4 # 字符集
16    collation: utf8mb4_unicode_ci # 字符序
17    timeout: 10s # 连接超时
18    read_timeout: 2s # 读超时
19    write_timeout: 2s # 写超时
20    parse_time: true # 是否解析时间
21    protocol: tcp # 传输协议
22    loc: Local # 时区
```



```
23     conn_max_idle: 10 # 连接池最大空闲连接数
24     conn_max_open: 20 # 连接池最大连接数
25     conn_max_lifetime: 1h # 连接数最大生命周期
26
27 read:
28     driver: mysql # 连接驱动
29     dsn: "" # dsn, 如果设置了dsn, 以下的所有设置都不生效
30     host: localhost # ip地址
31     port: 3306 # 端口
32     database: coredemo # 数据库
33     username: jianfengye # 用户名
34     password: "123456789" # 密码
35     charset: utf8mb4 # 字符集
36     collation: utf8mb4_unicode_ci # 字符序
```

在这个 database.yaml 中，我们配置了 database.default 和 database.read 两个数据源。database.read 数据源，并没有设置诸如时区 loc、连接池 conn_max_open 配置，这些缺省的配置要从 database.yaml 的根结构中获取。

要实现这个也不难，先在 framework/provider/orm/service.go 中实现一个 GetBaseConfig 方法，来读取 database.yaml 根目录的结构：

[复制代码](#)

```
1 // GetBaseConfig 读取database.yaml根目录结构
2 func GetBaseConfig(c framework.Container) *contract.DBConfig {
3
4     configService := c.MustMake(contract.ConfigKey).(contract.Config)
5     logService := c.MustMake(contract.LogKey).(contract.Log)
6
7     config := &contract.DBConfig{}
8     // 直接使用配置服务的load方法读取,yaml文件
9     err := configService.Load("database", config)
10    if err != nil {
11        // 直接使用logService来打印错误信息
12        logService.Error(context.Background(), "parse database config error", nil)
13        return nil
14    }
15    return config
16 }
17
```

然后设计一个根据配置路径加载某个配置结构的方法。这里这个方法一定是在具体初始化某个 DB 实例的时候使用到，所以要封装为一个 Option 结构，写在

framework/provider/orm/config.go 中：

[复制代码](#)

```
1 // WithConfigPath 加载配置文件地址
2 func WithConfigPath(configPath string) contract.DBOption {
3     return func(container framework.Container, config *contract.DBConfig) error {
4         configService := container.MustMake(contract.ConfigKey).(contract.Config)
5         // 加载configPath配置路径
6         if err := configService.Load(configPath, config); err != nil {
7             return err
8         }
9         return nil
10    }
11 }
```

现在，对于使用者来说，要初始化一个配置路径为 database.default 的数据库，就可以这么使用：

[复制代码](#)

```
1 gormService := c.MustMake(contract.ORMKey).(contract.ORMService)
2 db, err := gormService.GetDB(orm.WithConfigPath("database.default"), orm.WithD
```

日志

配置项设计清楚了，我们再来思考下日志这块。上一章介绍过了，Gorm 是有自己的输出规范的，在初始化参数 gorm.Config 中定义了一个日志输出接口 Interface。我们来仔细看下这个接口的定义：

[复制代码](#)

```
1 const (
2     Silent LogLevel = iota + 1
3     Error
4     Warn
5     Info
6 )
7
8 // Interface logger interface
9 type Interface interface {
10     LogMode(LogLevel) Interface // 日志级别
11     Info(context.Context, string, ...interface{})
12     Warn(context.Context, string, ...interface{})
13     Error(context.Context, string, ...interface{})
14 }
```

```
14     Trace(ctx context.Context, begin time.Time, fc func() (sql string, rowsAffe
15 }
```

Gorm 接口的日志级别分类比较简单：Info、Warn、Error、Trace。恰巧，这几个日志级别都在我们 hade 框架定义的 7 个日志级别中，所以完全可以将 Gorm 的这几个级别，映射到 hade 的日志级别中。也就是说，Gorm 打印的 Info 级别日志输出到 hade 的 Info 日志中、error 日志输出到 hade 的 error 日志中。

至于 Gorm 提供的一个 LogMode 来调整日志级别，由于我们的 hade 框架已经可以通过配置进行日志级别设置了，所以 LogMode 函数对我们来说是没有什么意义的。

好，了解 Gorm 的日志接口之后，我们明确了接下来要做的事情：**实现一个 Gorm 的日志实现类，但是这个日志实现类中的每个方法都用 hade 的日志服务来实现。**

我们在 framework/provider/orm/logger.go 中定义一个 OrmLogger 结构，它带有一个 logger 属性，这个 logger 属性存放的是 hade 容器中的 log 服务：

[复制代码](#)

```
1 // OrmLogger orm的日志实现类，实现了gorm.Logger.Interface
2 type OrmLogger struct {
3     logger contract.Log // 有一个logger对象存放hade的log服务
4 }
5
6 // NewOrmLogger 初始化一个ormLogger,
7 func NewOrmLogger(logger contract.Log) *OrmLogger {
8     return &OrmLogger{logger: logger}
9 }
```

它实现了 Gorm 的 Logger.Interface 接口。其中 LogMode 什么都不做，Info、Error、Warn、Trace 分别对应 hade 容器中 log 服务的 Info、Error、Warn、Trace 方法：

[复制代码](#)

```
1 // Info 对接hade的info输出
2 func (o *OrmLogger) Info(ctx context.Context, s string, i ...interface{}) {
3     fields := map[string]interface{}{
4         "fields": i,
5     }
6     o.logger.Info(ctx, s, fields)
7 }
```

```

8 // Warn 对接hade的Warn输出
9 func (o *OrmLogger) Warn(ctx context.Context, s string, i ...interface{}) {
10     fields := map[string]interface{}{
11         "fields": i,
12     }
13     o.logger.Warn(ctx, s, fields)
14 }
15
16 // Error 对接hade的Error输出
17 func (o *OrmLogger) Error(ctx context.Context, s string, i ...interface{}) {
18     fields := map[string]interface{}{
19         "fields": i,
20     }
21     o.logger.Error(ctx, s, fields)
22 }
23
24 // Trace 对接hade的Trace输出
25 func (o *OrmLogger) Trace(ctx context.Context, begin time.Time, fc func() (sql
26     sql, rows := fc()
27     elapsed := time.Since(begin)
28     fields := map[string]interface{}{
29         "begin": begin,
30         "error": err,
31         "sql":    sql,
32         "rows":   rows,
33         "time":   elapsed,
34     }
35
36     s := "orm trace sql"
37     o.logger.Trace(ctx, s, fields)
38 }
39

```

这里稍微注意下 Trace 方法，Gorm 的 Trace 方法的参数中有传递时间戳 begin，这个时间戳代表 SQL 执行的开始时间，而在函数中使用 time.Now 获取到当前时间之后，两个相减，我们可以获取到这个 SQL 的实际执行时间，然后作为 hade 日志服务的 fields map 的一个字段输出。除了 Trace，其他几个基本上简单封装 hade 的日志服务方法就好了。

服务实例

好了，到现在 Gorm 的配置结构和日志结构也完成了。万事俱备，下面我们就开始写具体的 ORM 服务的实例 HadeGorm，在 framework/provider/orm/service.go 中。

首先，定义实现 contract.ORMService 的结构 HadeGorm。要明确一点，我们会使用这个结构来生成不同数据库的 gorm.DB 结构，**所以这个 HadeGorm 是一个与某个数据库**

设置无关的结构，而且它应该对单个数据库是一个单例模式，即在一个服务中，我从 HadeGorm 两次获取到的 default 数据库的 gorm.DB 是同一个。

设置 HadeGorm 结构如下：

[复制代码](#)

```
1 // HadeGorm 代表hade框架的orm实现
2 type HadeGorm struct {
3     container framework.Container // 服务容器
4     dbs        map[string]*gorm.DB // key为dsn, value为gorm.DB (连接池)
5
6     lock *sync.RWMutex
7 }
```

dbs 就是为了单例存在，它的 key 直接设计为一个 string，也就是连接数据库的 DSN 字符串，而 value 就是 gorm.DB 结构。

这样我们在拿到一个 DSN 的时候，从这个 map 中就能判断出是否已经实例化过这个数据库对应的 gorm.DB 了；如果没有实例化过，就实例化一个 gorm.DB，并且将这个实例挂到这个 map 中。**不过这个逻辑会对 dbs 有并发修改操作，所以这里要使用一个读写锁来锁住这个 dbs 的修改。**


对应实例化 HadeGorm 的方法为 NewHadeGorm，它的具体实现就是初始化 HadeGorm 中的每个字段。继续写入这段：

[复制代码](#)

```
1 // NewHadeGorm 代表实例化Gorm
2 func NewHadeGorm(params ...interface{}) (interface{}, error) {
3     container := params[0].(framework.Container)
4     dbs := make(map[string]*gorm.DB)
5     lock := &sync.RWMutex{}
6     return &HadeGorm{
7         container: container,
8         dbs:       dbs,
9         lock:      lock,
10    }, nil
11 }
```

重头戏在 GetDB 方法的实现上。

首先初始化 `orm.Config`，其中包括从配置中获取设置项，也包括初始化内部的 `Gorm`；然后将 `GetDB` 的 `option` 参数作用于初始化的 `orm.Config`，修改默认配置；通过 `orm.Config` 生成 DSN 字符串。

 复制代码

```
1 // 读取默认配置
2 config := GetBaseConfig(app.container)
3
4 logService := app.container.MustMake(contract.LogKey).(contract.Log)
5
6 // 设置Logger
7 ormLogger := NewOrmLogger(logService)
8 config.Config = &gorm.Config{
9     Logger: ormLogger,
10 }
11
12 // option对opt进行修改
13 for _, opt := range option {
14     if err := opt(app.container, config); err != nil {
15         return nil, err
16     }
17 }
```

之后根据 `dsn` 字符串判断数据库实例 `gorm.DB` 是否已经存在了。如果存在直接返回 `gorm.DB`，如果不存在需要实例化 `gorm.DB`，这一步逻辑稍微复杂一点：


根据配置项 `orm.Config` 中的不同驱动，来实例化 `gorm.DB`（支持 MySQL/Postgres/SQLite/SQL Server/ClickHouse）

根据配置项 `orm.Config` 中的连接池配置，设置 `gorm.DB` 的连接池

将实例化后的 `gorm.DB` 和 DSN 放入 map 映射中

返回实例化后的 `gorm.DB`

代码如下：

 复制代码

```
1 // 如果最终的config没有设置dsn,就生成dsn
2 if config.Dsn == "" {
3     dsn, err := config.FormatDsn()
4     if err != nil {
5         return nil, err
6     }
7 }
```

```
6         }
7         config.Dsn = dsn
8     }
9
10    // 判断是否已经实例化了gorm.DB
11    app.lock.RLock()
12    if db, ok := app.dbs[config.Dsn]; ok {
13        app.lock.RUnlock()
14        return db, nil
15    }
16    app.lock.RUnlock()
17
18    // 没有实例化gorm.DB, 那么就要进行实例化操作
19    app.lock.Lock()
20    defer app.lock.Unlock()
21
22    // 实例化gorm.DB
23    var db *gorm.DB
24    var err error
25    switch config.Driver {
26    case "mysql":
27        db, err = gorm.Open(mysql.Open(config.Dsn), config)
28    case "postgres":
29        db, err = gorm.Open(postgres.Open(config.Dsn), config)
30    case "sqlite":
31        db, err = gorm.Open(sqlite.Open(config.Dsn), config)
32    case "sqlserver":
33        db, err = gorm.Open(sqlserver.Open(config.Dsn), config)
34    case "clickhouse":
35        db, err = gorm.Open(clickhouse.Open(config.Dsn), config)
36    }
37
38    // 设置对应的连接池配置
39    sqlDB, err := db.DB()
40    if err != nil {
41        return db, err
42    }
43
44    if config.ConnMaxIdle > 0 {
45        sqlDB.SetMaxIdleConns(config.ConnMaxIdle)
46    }
47    if config.ConnMaxOpen > 0 {
48        sqlDB.SetMaxOpenConns(config.ConnMaxOpen)
49    }
50    if config.ConnMaxLifetime != "" {
51        liftTime, err := time.ParseDuration(config.ConnMaxLifetime)
52        if err != nil {
53            logger.Error(context.Background(), "conn max lift time error", map{
54                "err": err,
55            })
56        } else {
57            sqlDB.SetConnMaxLifetime(liftTime)
```

```
58     }
59 }
60
61 if config.ConnMaxIdleTime != "" {
62     idleTime, err := time.ParseDuration(config.ConnMaxIdleTime)
63     if err != nil {
64         logger.Error(context.Background(), "conn max idle time error", map{
65             "err": err,
66         })
67     } else {
68         sqlDB.SetConnMaxIdleTime(idleTime)
69     }
70 }
71
72 // 挂载到map中，结束配置
73 if err != nil {
74     app.dbs[config.Dsn] = db
75 }
76
77 return db, err
```

如果前面的内容都理解了，这段代码实现也没有什么难点了。唯一要注意的地方就是锁的使用，**由于对存在 gorm.DB 的 map 是读多写少，所以这里也是使用读写锁**，在读取的时候加了一个读锁，如果 map 中没有我们要的 gorm.DB，先把读锁解开，再加一个写锁，初始化完 gorm.DB、保存进入 map 映射后，再把写锁解开。这样能有效防止对 map 的并发读写。

完整的 GetDB 方法可以参考 GitHub 上的 [framework/provider/orm/service.go](https://github.com/jefferyzhang/go-framework/blob/master/provider/orm/service.go)。

最后记得去业务代码 main.go 中，把我们的 GormProvider 注入服务容器：

```
1 func main() {
2     // 初始化服务容器
3     container := framework.NewHadeContainer()
4     ...
5     container.Bind(&orm.GormProvider{})
6
7     ...
8
9     // 运行root命令
10    console.RunCommand(container)
11 }
```

[复制代码](#)

整个 Gorm 就已经结合到 hade 框架中了。

测试

下面来做一下测试。我们用真实的 MySQL 进行测试。当然你需要在本机 / 远端 / Docker 搭建一个 MySQL，至于怎么搭建，教程网上有很多了，这里就不详细描述。

我用的是 Mac，使用 homebrew 能很方便搭建一个 MySQL 服务。我的 MySQL 实例搭建在本机的 3306 端口，并且搭建完成之后，我创建了一个 coredemo 的 database 数据库：

```
~ ➤ mysql -ujianfengye -p123456789
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14
Server version: 8.0.22 Homebrew

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.


Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| coredemo |
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
5 rows in set (0.07 sec)

mysql> █
```

所以我的配置文件 config/development/database.yaml 配置如下：

```
1 conn_max_idle: 10 # 通用配置，连接池最大空闲连接数
2 conn_max_open: 100 # 通用配置，连接池最大连接数
3 conn_max_lifetime: 1h # 通用配置，连接数最大生命周期
4 protocol: tcp # 通用配置，传输协议
```

 复制代码

```
5 loc: Local # 通用配置, 时区
6
7 default:
8     driver: mysql # 连接驱动
9     dsn: "" # dsn, 如果设置了dsn, 以下的所有设置都不生效
10    host: localhost # ip地址
11    port: 3306 # 端口
12    database: coredemo # 数据库
13    username: jianfengye # 用户名
14    password: "123456789" # 密码
15    charset: utf8mb4 # 字符集
16    collation: utf8mb4_unicode_ci # 字符序
17    timeout: 10s # 连接超时
18    read_timeout: 2s # 读超时
19    write_timeout: 2s # 写超时
20    parse_time: true # 是否解析时间
21    protocol: tcp # 传输协议
22    loc: Local # 时区
```

我们想在 coredemo 数据库中增加一个 user 表, 按照 Gorm 的规范, 需要先定义一个数据结构 User。在 app/http/module/demo/model.go 中:

```
1 // User is gorm model
2 type User struct {
3     ID          uint
4     Name        string
5     Email       *string
6     Age         uint8
7     Birthday    *time.Time
8     MemberNumber sql.NullString
9     ActivatedAt sql.NullTime
10    CreatedAt    time.Time
11    UpdatedAt    time.Time
12 }
```

[复制代码](#)

然后在应用目录 app/http/module/demo/api_orm.go 中, 定义了一个新的路由方法 DemoOrm, 在这个方法中, 我们先从容器中获取到 gorm.DB 的实例, 然后使用 db.AutoMigrate 同步数据表 user。

如果第一次执行的时候, 数据库中没有表 user, 它会自动创建 user 表, 然后分别调用 db.Create、db.Save、db.First、db.Delete 来对 user 表进行增删改查操作:



```
1 // DemoOrm Orm的路由方法
2 func (api *DemoApi) DemoOrm(c *gin.Context) {
3     logger := c.MustMakeLog()
4     logger.Info(c, "request start", nil)
5
6     // 初始化一个orm.DB
7     gormService := c.MustMake(contract.ORMKey).(contract.ORMService)
8     db, err := gormService.GetDB(orm.WithConfigPath("database.default"))
9     if err != nil {
10         logger.Error(c, err.Error(), nil)
11         c.AbortWithError(50001, err)
12         return
13     }
14     db.WithContext(c)
15
16     // 将User模型创建到数据库中
17     err = db.AutoMigrate(&User{})
18     if err != nil {
19         c.AbortWithError(500, err)
20         return
21     }
22     logger.Info(c, "migrate ok", nil)
23
24     // 插入一条数据
25     email := "foo@gmail.com"
26     name := "foo"
27     age := uint8(25)
28     birthday := time.Date(2001, 1, 1, 1, 1, 1, 1, time.Local)
29     user := &User{
30         Name:      name,
31         Email:      &email,
32         Age:        age,
33         Birthday:   &birthday,
34         MemberNumber: sql.NullString{},
35         ActivatedAt: sql.NullTime{},
36         CreatedAt:   time.Now(),
37         UpdatedAt:   time.Now(),
38     }
39     err = db.Create(user).Error
40     logger.Info(c, "insert user", map[string]interface{}{
41         "id": user.ID,
42         "err": err,
43     })
44
45     // 更新一条数据
46     user.Name = "bar"
47     err = db.Save(user).Error
48     logger.Info(c, "update user", map[string]interface{}{
49         "err": err,
50         "id": user.ID,
51     })
52 }
```

```

52 // 查询一条数据
53 queryUser := &User{ID: user.ID}
54
55 err = db.First(queryUser).Error
56 logger.Info(c, "query user", map[string]interface{}{
57     "err": err,
58     "name": queryUser.Name,
59 })
60
61 // 删除一条数据
62 err = db.Delete(queryUser).Error
63 logger.Info(c, "delete user", map[string]interface{}{
64     "err": err,
65     "id": user.ID,
66 })
67 c.JSON(200, "ok")
68 }
69

```

记得修改 app/http/module/demo/api.go 中的路由注册：

 复制代码

```

1 func Register(r *gin.Engine) error {
2     api := NewDemoApi()
3     ...
4     r.GET("/demo/orm", api.DemoOrm)
5     return nil
6 }

```

现在，使用 ./hade build self 来重新编译 hade 文件，使用 ./hade app start 启动服务，并挂起在控制台，日志会输出到控制台。浏览器调用

<http://localhost:8888/demo/orm>，控制台打印日志如下：

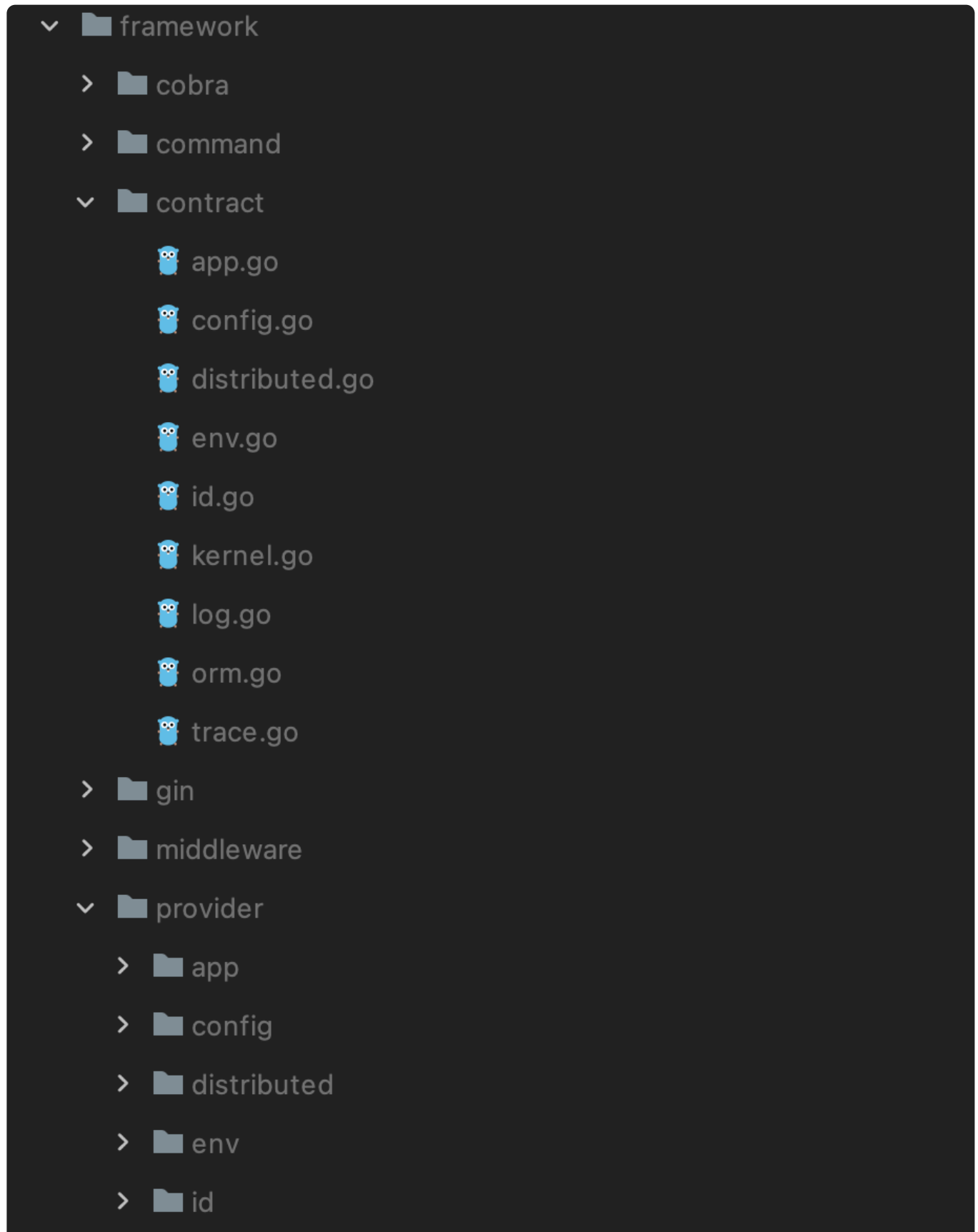
```

~/Documents/UGit/coredemo > geekbang/25 ● ./hade app start
[PID] 55601
app serve url: :8888
[Info] 2021-10-27T09:54:53+08:00 "request start" map[]
[Trace] 2021-10-27T09:54:53+08:00 "orm trace sql" map[begin:2021-10-27 09:54:53.158382 +0800 CST m=+51.490311603 error:<nil> rows:-1 sql:SELECT DATABASE() time:1.487197ms]
[Trace] 2021-10-27T09:54:53+08:00 "orm trace sql" map[begin:2021-10-27 09:54:53.159946 +0800 CST m=+51.491875598 error:<nil> rows:-1 sql:SELECT count(*) FROM information_schema.tables WHERE table_schema = 'coredemo' AND table_name = 'users' AND table_type = 'BASE TABLE' time:28.70403ms]
[Trace] 2021-10-27T09:54:53+08:00 "orm trace sql" map[begin:2021-10-27 09:54:53.188776 +0800 CST m=+51.520706195 error:<nil> rows:-1 sql:SELECT DATABASE() time:209.356µs]
[Trace] 2021-10-27T09:54:53+08:00 "orm trace sql" map[begin:2021-10-27 09:54:53.189041 +0800 CST m=+51.520970869 error:<nil> rows:-1 sql:SELECT column_name, is_nullable, data_type, character_maximum_length, numeric_precision, numeric_scale, datetime_precision FROM information_schema.columns WHERE table_schema = 'coredemo' AND table_name = 'users' time:5.62846ms]
[Info] 2021-10-27T09:54:53+08:00 "migrate ok" map[]
[Trace] 2021-10-27T09:54:53+08:00 "orm trace sql" map[begin:2021-10-27 09:54:53.194066 +0800 CST m=+51.526795548 error:<nil> rows:1 sql:INSERT INTO 'users' ('name','email','age','birthday','member_number','activated_at','created_at','updated_at') VALUES ('foo','foo@gmail.com',25,'2001-01-01 01:01:01',NULL,NULL,'2021-10-27 09:54:53.194','2021-10-27 09:54:53.194') time:17.394801ms]
[Info] 2021-10-27T09:54:53+08:00 "insert user" map[err:<nil> id:4]
[Trace] 2021-10-27T09:54:53+08:00 "orm trace sql" map[begin:2021-10-27 09:54:53.212322 +0800 CST m=+51.544251783 error:<nil> rows:1 sql:UPDATE 'users' SET 'name'='bar','email'='foo@gmail.com','age'=25,'birthday'='2001-01-01 01:01:01','member_number'='NULL','activated_at'='NULL','created_at'='2021-10-27 09:54:53.194','updated_at'='2021-10-27 09:54:53.213' WHERE 'id' = 4 time:5.363596ms]
[Info] 2021-10-27T09:54:53+08:00 "update user" map[err:<nil> id:4]
[Trace] 2021-10-27T09:54:53+08:00 "orm trace sql" map[begin:2021-10-27 09:54:53.217742 +0800 CST m=+51.549672214 error:<nil> rows:1 sql:SELECT * FROM 'users' WHERE 'users'.'id' = 4 ORDER BY 'users'.'id' LIMIT 1 time:495.384µs]
[Info] 2021-10-27T09:54:53+08:00 "query user" map[err:<nil> name:bar]
[Trace] 2021-10-27T09:54:53+08:00 "orm trace sql" map[begin:2021-10-27 09:54:53.218326 +0800 CST m=+51.550256343 error:<nil> rows:1 sql:DELETE FROM 'users' WHERE 'users'.'id' = 4 time:1.094781ms]
[Info] 2021-10-27T09:54:53+08:00 "delete user" map[err:<nil> id:4]

```

可以清晰地通过 trace 日志看到底层的 Insert/Update/Select/Delete 的操作，并且可以通过 time 字段看到这个请求的具体耗时。到这里 Gorm 融合 hade 框架就验证完成了。

本节课我们主要修改了 framework 目录下的 contract/orm.go 和 provider/orm 目录。目录截图如下，供对比查看，所有代码都已经上传到 [geekbang/25](https://github.com/geekbang/geekbang/tree/master/25) 分支了。



```
> kernel
> log
v orm
  config.go
  config_test.go
  logger.go
  provider.go
  service.go
```

小结

对于 Gorm 这样比较庞大的库，要把 Gorm 完美集成到 hade 框架，更好地支持业务对数据库频繁的增删改查操作，我们并不是一开始就动手修改代码，而是先把 Gorm 的实例化部分的源码都理清楚了，再动手集成才不会出现问题。

现在我们可以方便获取到 gorm.DB 了。但是在具体开发业务的时候，如何使用好 Gorm 来为业务服务，也是一个非常值得花心思研究的课题。好在我们的技术选型是目前 Golang 业界最火的 Gorm，网络上关于如何使用 Gorm 的课程有非常多了，在具体开发业务的时候，你可以自己参考和研究。

思考题

在 ORM 框架中，model 层的存放位置一直是个很有争论的话题。比如 geekbang/25 分支上 model 层的 User 结构，我存放在 app/http/module/demo 中，有同学会觉得 model 层放在 app/http/model 目录比较好么？具体 model 是否应该单独作为一个文件夹出来呢？

欢迎在留言区分享你的思考。感谢你的收听，如果你觉得今天的内容对你有所帮助，也欢迎分享给你身边的朋友，邀请他一起学习。我们下节课见~

分享给需要的人，Ta 订阅后你可得 20 元现金奖励

 赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | GORM: 数据库的使用必不可少 (上)

下一篇 27 | 缓存服务: 如何基于Redis实现封装?

训练营推荐

Java 学习包免费领 NEW

面试题答案均由大厂工程师整理

阿里、美团等
大厂真题

18 大知识点
专项练习

大厂面试
流程解析

可复用的
面试方法

面试前
要做的准备

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。