Less is more

lessgo

v0.7.0 - 2016.11.03(草稿)

Index

1. 导语 Introduction 6
2. 特性 Features 6
3. 应用场景 Scenarios 6
4. 当前版本 Version 6
5. 安装 Install 7
构成说明 7
安装 7
6. 项目架构 Framework 7
7. 项目构建工具 Less 8
Go 版本需求 Requirements 8
安装 Installation 8
基本命令 Basic commands 9
less new 9
less run 10
帮助 Help 10
8. Lessgo 执行逻辑 11
9. 开始 Hello world 11
Hello world 11
10. 配置项 Configurations 13
应用程序全局配置 13
配置文件 13
配置说明 13
配置项 13
Xorm 数据库访问配置 14
配置文件 14
配置说明 14
配置项 14
Gorm 数据库访问配置 16
配置文件 16

配置说明 16
配置项 16
Sqlx 数据库访问配置 17
配置文件 17
配置说明 17
配置项 17
Directsql 访问数据库 17
配置文件 17
配置项 17
11. Serves HTTP 18
Lessgo Serves 启动过程 18
启动过程示意图 18
非常简洁的主单元代码 18
Lessgo.Run 运作 19
App.Run 运作 20
Panic 处理 21
Custom HTTP Errors 21
12. 处理器 Handlers 22
13. 中间件 Middleware 23
过滤的用法（举例） 23
支持动态配置的用法（举例） 23
14. 路由 Routing 24
源码路由 24
动态路由 25
15. 参数 Parameters 26
16. 上下文 Context 27
参数解析 27
文件上传 27
17. 静态文件 Static files 28
18. Pongo2 Template 28

Pongo2 调用 28
Pongo2 扩展 29
Pongo2 与 directsql 扩展调用(待更新) 30
Pongo2 技术文档 30
19. Markdown 30
20. 日志 Logger 30
21. Cookies 31
22. Sessions 31
23. 数据库访问 Database access 31
Xorm 数据库访问 31
驱动支持(具体以其官方为准) 32
配置项 32
技术文档 32
使用步骤示例 32
Gorm 数据库访问 32
驱动支持(具体以其官方为准) 33
配置项 33
技术文档 33
使用步骤示例 33
Sqlx 数据库访问 34
Directsql 访问数据库 34
依赖 34
驱动支持 34
配置项 34
原理 34
使用配置流程 35
SQL 配置类型 35
SQL 配置文件详解 36
路由(调用 SQL 的 API 设置) 36
客户端调用 36

使用示例 37
24. Issues 38
25. F&Q 38
26. About 39
贡献者名单 39
开源协议 39
27. Appendix 40

1. 导语 Introduction

Lessgo 是一款 Go 语言开发的简单、稳定、高效、灵活的 web 开发框架。它的项目组织形式经过精心设计，实现前后端分离、系统与业务分离，完美兼容 MVC 与 MVVC 等多种开发模式，非常利于企业级应用与 API 接口的开发。当然，最值得关注的是它突破性支持运行时路由重建，开发者可通过 Admin 后台轻松配置路由，并实现启用/禁用模块或操作、添加/移除中间件等！同时，以 ApiHandler 与 ApiMiddleware 为项目基本组成单元，可实现编译期或运行时的自由搭配组合，让开发变得更加灵活富有趣味性，赶快开始你的探索之旅吧，lessgo！

2. 特性 Features

- 使用简单、运行稳定高效（核心架构来自对 echo 真正意义的二次开发）
- 兼容流行系统模式如:MVC、MVVC、Restful...
- httprouter 真实路由配合强大的虚拟路由层，不仅性能优秀更可同时支持在源码或 admin 中动态配置
- 多异构数据库支持，用户可以选择 xorm 或者 gorm 两种引擎（当然愿意，用户还可以同时使用两种引擎）,如果习惯直接使用 SQL,可使用 Drectsql 配置化 SQL 执行引擎
- 优化的项目目录组织最佳实践，满足复杂企业应用需要
- 集成统一的系统日志(system、database 独立完整的日志)
- 提供 Session 管理（优化 beego 框架中的 session 包）
- 强大的前端模板渲染引擎（pongo2），扩展的 Drectsql 支持函数
- 天生支持运行时可更新的 API 测试网页（swagger2.0）
- 配置文件自动补填默认值，并按字母排序
- 支持热编译
- 支持热升级
- 另外灵活的扩展包中还包含 HOTP、TOTP、UUID 以及各种条码生成工具等常用工具包

3. 应用场景 Scenarios

- 网站
- web 应用
- Restful API 服务应用
- 企业应用

4. 当前版本 Version

- V0.7.0

发布日期：2016.06.01

5. 安装 Install

构成说明

- 核心框架：[lessgo](https://github.com/lessgo/lessgo)
- 框架部署工具：[less](https://github.com/lessgo/less)
- 框架扩展：[lessgoext](https://github.com/lessgo/lessgoext)
- 项目 Demo：[demo](https://github.com/lessgo/demo)
- 框架文档　[document](https://github.com/lessgo/doc)

安装

```
go get -u github.com/lessgo/lessgo
go get -u github.com/lessgo/less
go get -u github.com/lessgo/lessgoext/...
```

6. 项目架构 Framework

```
—Project 项目开发目录
├──config 配置文件目录
|    ├──app.config 系统应用配置文件
|    └──db.config 数据库配置文件
├──common 后端公共目录
|    └──... 如 utils 等其他
├──middleware 后端公共中间件目录
├──static 前端公共目录 (url: /static)
|    ├──tpl 公共 tpl 模板目录
|    ├──js 公共 js 目录 (url: /static/js)
|    ├──css 公共 css 目录 (url: /static/css)
|    ├──img 公共 img 目录 (url: /static/img)
|    └──plugin 公共 js 插件 (url: /static/plugin)
├──uploads 默认上传下载目录
├──router 源码路由配置
|    ├──sys_router.go 系统模块路由文件
|    ├──biz_router.go 业务模块路由文件
├──sys_handler 系统模块后端目录
|    ├──xxx 子模块目录
|    |    ├──example.go example 操作
|    |    └──... xxx 的子模块目录
|    └──... 其他子模块目录
```

```
├──sys_model 系统模块数据模型目录
├──sys_view 系统模块前端目录 (url: /sys)
|    ├──xxx 与 sys_handler 对应的子模块目录 (url: /sys/xxx)
|    |    ├──example.tpl 相应操作的模板文件
|    |    ├──example2.html 无需绑定操作的静态 html 文件
|    |    ├──xxx.css css 文件(可有多个)
|    |    ├──xxx.js js 文件(可有多个)
|    |    └──... xxx 的子模块目录
├──biz_handler 业务模块后端目录
|    ├──xxx 子模块目录
|    |    ├──example.go example 操作
|    |    └──... xxx 的子模块目录
|    └──... 其他子模块目录
├──biz_model 业务模块数据模型目录
├──biz_view 业务模块前端目录 (url: /biz)
|    ├──xxx 与 biz_handler 对应的子模块目录 (url: /biz/xxx)
|    |    ├──example.tpl 相应操作的模板文件
|    |    ├──example2.html 无需绑定操作的静态 html 文件
|    |    ├──xxx.css css 文件(可有多个)
|    |    ├──xxx.js js 文件(可有多个)
|    |    └──... xxx 的子模块目录
├──database 默认数据库文件存储目录
├──logger 运行日志输出目录
└──main.go 应用入口文件
```

7. 项目构建工具 Less

Less is a command line tool facilitating development with lessgo framework. It is modified from bee.

Go 版本需求 Requirements

- Go version >= 1.3

安装 Installation

Begin by installing `less` using `go get` command.

bash
go get github.com/lessgo/less

Then you can add `less` binary to PATH environment variable in your `~/.bashrc` or `~/.bash_profile` file:

bash
export PATH=$PATH:<your_main_gopath>/bin
> If you already have `less` installed, updating `less` is simple:

bash
go get -u github.com/lessgo/less

基本命令 Basic commands

Less provides a variety of commands which can be helpful at various stage of development. The top level commands include:

- new create an application base on lessgo framework
- run run the app which can hot compile

less new

Creating a new lessgo web application is no big deal, too.

bash
$ less new myapp
[INFO] Creating application...
/home/zheng/gopath/src/myapp/
/home/zheng/gopath/src/myapp/common/
/home/zheng/gopath/src/myapp/middleware/
/home/zheng/gopath/src/myapp/static/
/home/zheng/gopath/src/myapp/static/tpl/
/home/zheng/gopath/src/myapp/static/css/
/home/zheng/gopath/src/myapp/static/js/
/home/zheng/gopath/src/myapp/static/img/
/home/zheng/gopath/src/myapp/static/plugin/
/home/zheng/gopath/src/myapp/uploads/
/home/zheng/gopath/src/myapp/router/
/home/zheng/gopath/src/myapp/sys_handler/
/home/zheng/gopath/src/myapp/sys_handler/admin/
/home/zheng/gopath/src/myapp/sys_handler/admin/login/
/home/zheng/gopath/src/myapp/sys_model/
/home/zheng/gopath/src/myapp/sys_model/admin/
/home/zheng/gopath/src/myapp/sys_view/
/home/zheng/gopath/src/myapp/sys_view/admin/

/home/zheng/gopath/src/myapp/sys_view/admin/login/
/home/zheng/gopath/src/myapp/biz_handler/
/home/zheng/gopath/src/myapp/biz_handler/home/
/home/zheng/gopath/src/myapp/biz_model/
/home/zheng/gopath/src/myapp/biz_view/
/home/zheng/gopath/src/myapp/biz_view/home/
/home/zheng/gopath/src/myapp/middleware/test.go
/home/zheng/gopath/src/myapp/static/img/favicon.ico
/home/zheng/gopath/src/myapp/static/js/jquery.js
/home/zheng/gopath/src/myapp/static/js/jquery.min.map
/home/zheng/gopath/src/myapp/router/sys_router.go
/home/zheng/gopath/src/myapp/router/biz_router.go
/home/zheng/gopath/src/myapp/sys_handler/admin/index.go
/home/zheng/gopath/src/myapp/sys_handler/admin/login/index.go
/home/zheng/gopath/src/myapp/sys_model/admin/login.go
/home/zheng/gopath/src/myapp/sys_view/admin/login/index.tpl
/home/zheng/gopath/src/myapp/biz_handler/home/index.go
/home/zheng/gopath/src/myapp/biz_handler/home/websocket.go
/home/zheng/gopath/src/myapp/biz_view/home/index.tpl
/home/zheng/gopath/src/myapp/biz_view/home/websocket.js
/home/zheng/gopath/src/myapp/biz_view/home/jquery.githubRepoWidget2.js
/home/zheng/gopath/src/myapp/biz_view/home/index.css
/home/zheng/gopath/src/myapp/main.go
2016/08/08 15:45:47 [SUCC] New application successfully created!

less run

To run the application we just created, navigate to the application folder and execute `less run`.
bash
$ cd myapp
$ less run

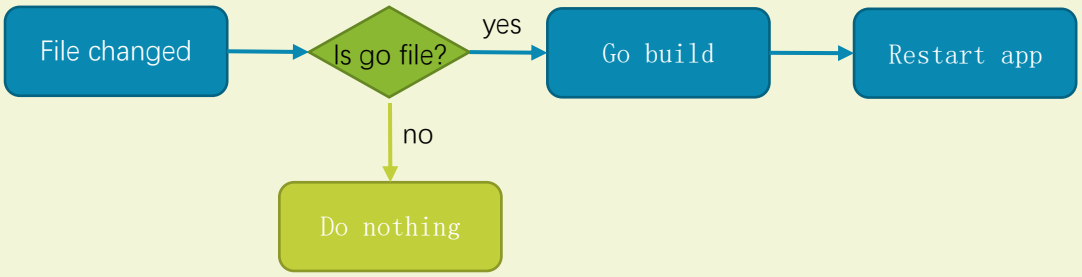
帮助 Help

If you happen to forget the usage of a command, you can always find the usage information by `less help <command>`.

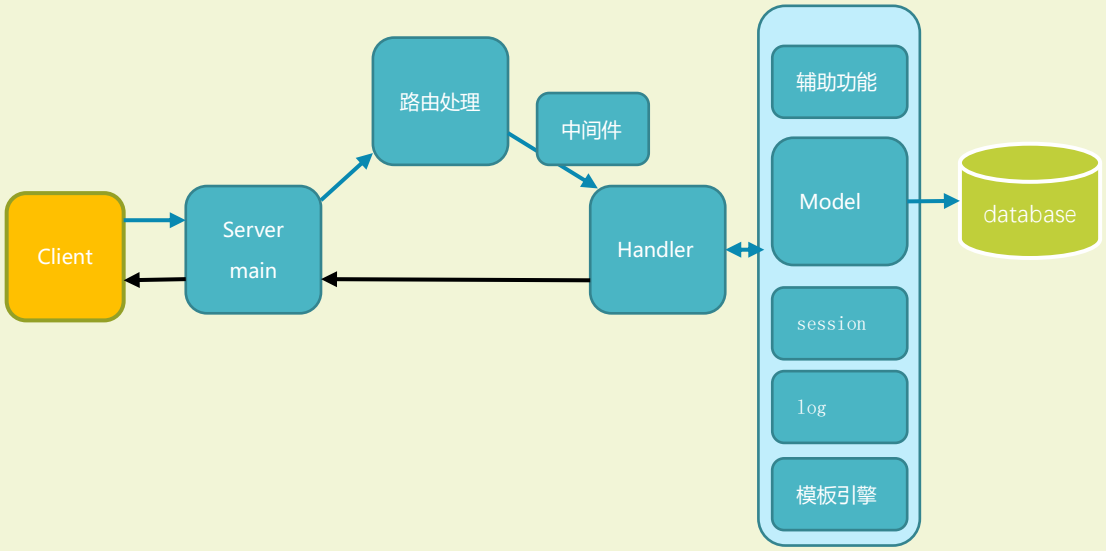
For instance, to get more information about the `run` command:

bash
$ less help run
usage: less run [appname] [watchall] [-main=*.go]

start the appname throw exec.Command
then start a notify watch for current dir
when the file has changed less will auto go build and restart the app, the flow chart is as follows：



8. Lessgo 执行逻辑



9. 开始 Hello world

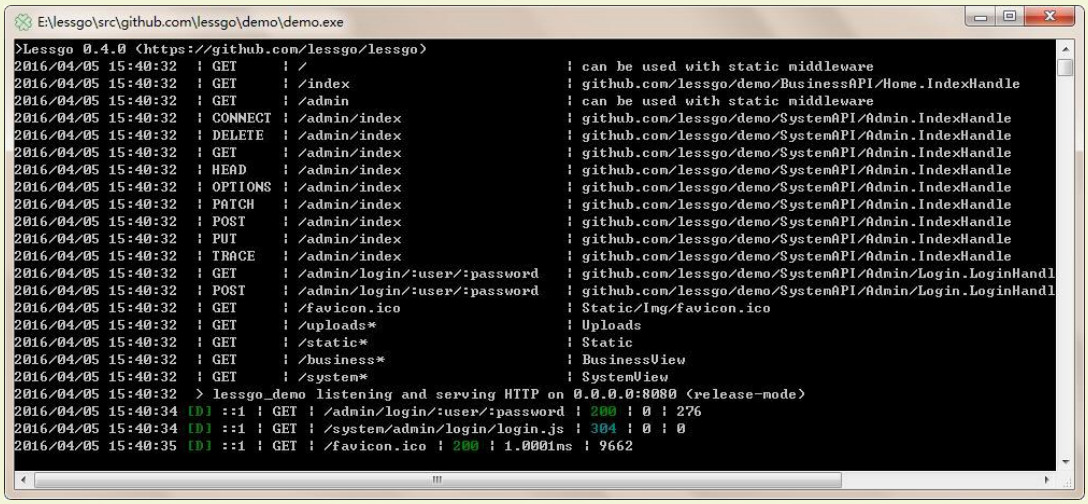
Hello world

```
import (
    "github.com/lessgo/lessgo"
```

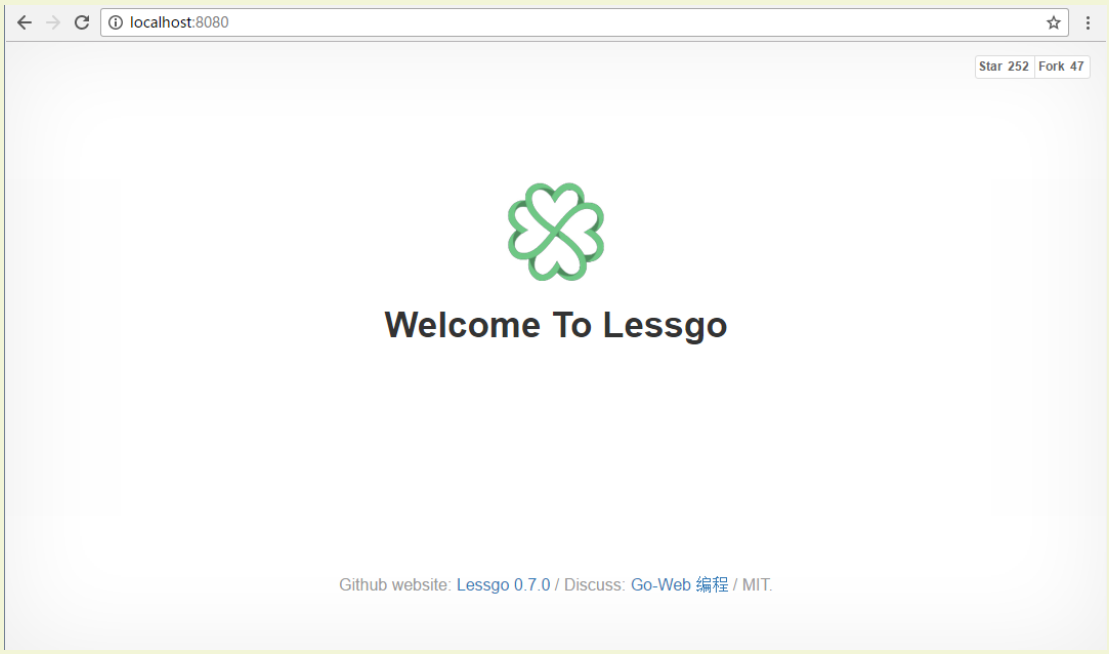
```
    "github.com/lessgo/lessgoext/swagger"
    _ "github.com/lessgo/demo/middleware"
    _ "github.com/lessgo/demo/router"
)
func main() {
    // 开启自动 api 文档，通过 config/apidoc_allow.myconfig 进行配置
    swagger.Reg()
    // 指定根目录 URL，本示例 /home 定位到 /biz_view/home 路径
    lessgo.SetHome("/home")
    // 开启网络服务，均使用 app.config 下的默认参数。

    lessgo.Run()
}
```

服务器运行界面如下:



浏览器打开 localhost:8080,运行成功，出现界面如下：



10. 配置项 Configurations

应用程序全局配置

配置文件

/config/app.config

配置说明

配置应用程序的全局信息。

配置项

[info] --框架基本信息
email=henrylee_cn@foxmail.com
license=MIT
licenseurl=https://github.com/lessgo/lessgo/raw/master/doc/LICENSE
version=0.4.0
description=A simple, stable, efficient and flexible web framework.
host=127.0.0.1:8080
contact=henrylee_cn@foxmail.com
termsofserviceurl=https://github.com/lessgo/lessgo

[filecache] --静态文件缓存配置项
maxcapmb=256 --最大缓存(内存 MB)
cachesecond=600 --缓存的刷新时间(秒)
singlefileallowmb=64 --单文件缓存限制(单文件大小超过 MB 的不缓存)

[listen] --http 服务器参数
enablehttps=false --允许 https
httpscertfile=
httpskeyfile=
graceful=false
address=0.0.0.0:8080 --IP 地址与端口
readtimeout=0 --读超时
writetimeout=0 --写超时

[log]
asyncchan=1000 并发数开关

level=debug 日志级别

[session]
enablesidinhttpheader=false
enablesidinurlquery=false
sessionautosetcookie=true
sessioncookielifetime=0
sessiondomain=
sessiongcmaxlifetime=3600
sessionname=lessgosessionID
sessionnameinhttpheader=Lessgosessionid
sessionon=true
sessionprovider=memory --保存方式: memory=内存
sessionproviderconfig=

[system]
appname=lessgo //应用名称
crossdomain=false //是否允许跨域访问
debug=true //调试模式
maxmemorymb=125 //最大内存

Xorm 数据库访问配置

配置文件

/config/xorm.config

配置说明

可以配置多个数据库连接，通过段 [xxxdb]分别配置不同的数据库，默认数据配置在 **[defaultdb]** 下。

配置项

[defaultdb] --默认数据库
name=preset --数据库名称
tablesnake=true --表名使用 snake 风格或保持不变
tablespace= --表命名空间
maxidleconns=1 --最大空闲连接数，超过该数量的会释放
showexectime=false --显示执行时间
showsql=false --显示执行的 SQL

tablefix=prefix --表命名空间是前缀还是后缀：prefix | suffix
connstring=Common/DB/sqlite.db --连接字符串
disablecache=false --禁止缓存
columnpace= -- 列命名空间
maxopenconns=1 --最大打开连接数
driver=sqlite3 --数据库驱动类型
columnsnake=true -- 列名使用 snake 风格或保持不变
columnfix=prefix 列命名空间是前缀还是后缀：prefix | suffix

;Mysql 连接配置段
[mysql]
columnfix=prefix
columnsnake=true
columnspace=
connstring=root:pwd@tcp(127.0.0.1:3306)/demodb?charset=utf8
disablecache=false
driver=mysql
maxidleconns=1
maxopenconns=1
name=mysql
showexectime=false
showsql=false
tablefix=prefix
tablesnake=true
tablespace=

;Postgres 连接配置段
[postgres]
columnfix=prefix
columnsnake=true
columnspace=
connstring=host=127.0.0.1 port=5432 user=postgres password=postgres dbname=demodb
disablecache=false
driver=postgres
maxidleconns=1
maxopenconns=1
name=postgres
showexectime=false
showsql=false
tablefix=prefix
tablesnake=true
tablespace=

;Mssql 连接配置段

```
[mssql]
columnfix=prefix
columnsnake=true
columnspace=
connstring=server=127.0.0.1;port=1433;database=demodb;user
id=sa;password=sa;Connection Timeout=90
disablecache=false
driver=mssql
maxidleconns=1
maxopenconns=1
name=mssql
showexectime=false
showsql=false
tablefix=prefix
tablesnake=true
tablespace=
```

Gorm 数据库访问配置

配置文件

/config/gorm.config

配置说明

可以配置多个数据库连接，通过段 [xxxdb]分别配置不同的数据库，默认数据配置在 **[defaultdb]** 下。

配置项

```
[defaultdb]
connstring=database/sqlite.db
driver=sqlite3
maxidleconns=1
maxopenconns=1
name=lessgo
showsql=false
```

Sqlx 数据库访问配置

配置文件

/config/sqlx.config

配置说明

可以配置多个数据库连接，通过段 [xxxdb]分别配置不同的数据库，默认数据配置在 **[defaultdb]** 下。

配置项

Directsql 访问数据库

配置文件

/config/directsql.config

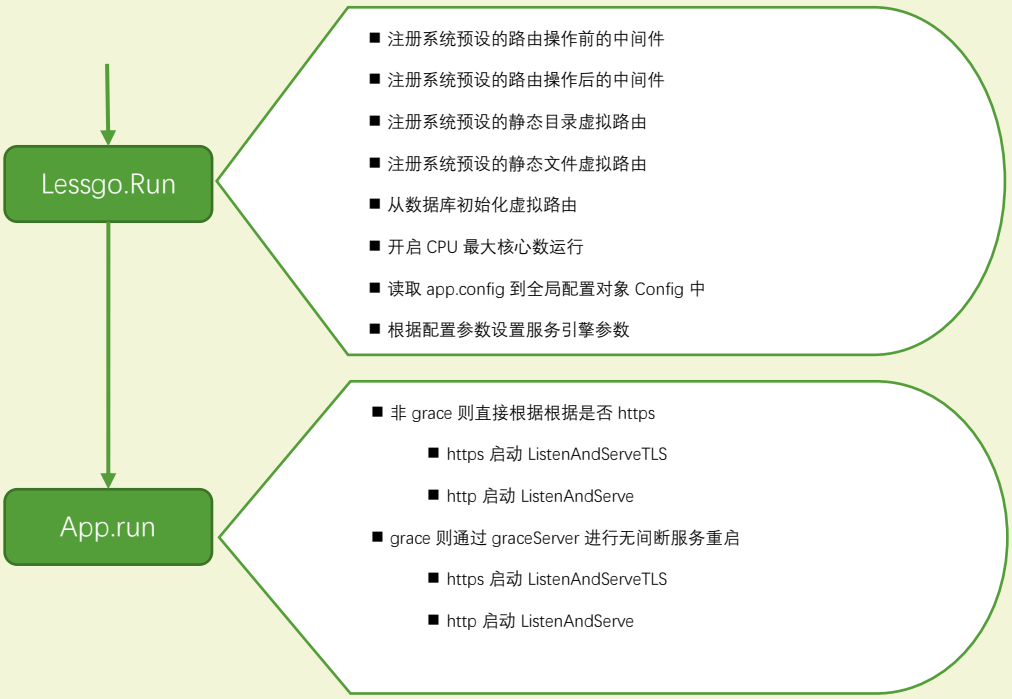
配置项

```
;SQL 配置文件扩展名，只能一个。
ext=.msql
;是否开始监控所有 roots 目录下的配置文件变化，改变自动处理(增加，删除，修改)
watch=true
;SQL 配置文件加载的根目录，可以个多个，定义后自动将真实文件名映射到前边名称
[roots]
biz=biz_model
sys=sys_model
```

11. Serves HTTP

Lessgo Serves 启动过程

启动过程示意图



非常简洁的主单元代码

```
package main
import (
	"github.com/lessgo/lessgo"
	"github.com/lessgo/lessgoext/swagger"
	_ "github.com/lessgo/demo/middleware"
	_ "github.com/lessgo/demo/router"
)

func main() {
	// 开启自动 api 文档
	// 参数为 true 表示自定义允许访问的 ip 前缀
	// 参数为 false 表示只允许局域网访问
	//swagger.Reg(true)
```

```
    // 指定根目录 URL
    lessgo.SetHome("/home")
    // 开启网络服务
    lessgo.Run()
}
```

Lessgo.Run 运作

```
//lessgo.go 运行服务
func Run() {
    // 添加系统预设的路由操作前的中间件
    registerBefore()
    // 添加系统预设的路由操作后的中间件
    registerAfter()
    // 添加系统预设的静态目录虚拟路由
    registerStatics()
    // 添加系统预设的静态文件虚拟路由
    registerFiles()
    // 从数据库初始化虚拟路由
    initVirtRouterConfig()
    // 重建路由
    ReregisterRouter()
    // 开启最大核心数运行
    runtime.GOMAXPROCS(runtime.NumCPU())
    // 配置服务器引擎
    var (
        tlsCertfile string
        tlsKeyfile  string
        mode        tring
        graceful    string
        protocol     = "HTTP"
    )
    if Config.Listen.EnableHTTPS {
        protocol = "HTTPS"
        tlsCertfile = Config.Listen.HTTPSCertFile
        tlsKeyfile = Config.Listen.HTTPSKeyFile
    }
    if Config.Debug {
        mode = "debug"
    } else {
        mode = "release"
    }
    if Config.Listen.Graceful {
        graceful = "(enable-graceful)"
```

```
	} else {
		graceful = "(disable-graceful)"
	}
	Log.Sys("> %s listening and serving %s on %v (%s-mode) %v", Config.AppName, protocol,
Config.Listen.Address, mode, graceful)
	// 启动服务
	lessgo.App.run(
		Config.Listen.Address,
		tlsCertfile,
		tlsKeyfile,
		time.Duration(Config.Listen.ReadTimeout),
		time.Duration(Config.Listen.WriteTimeout),
		Config.Listen.Graceful,
	)
}
```

最终通过 App.run 启动服务。

App.Run 运作

```
// app.go 启动 HTTP server.
func (this *App) run(address, tlsCertfile, tlsKeyfile string, readTimeout, writeTimeout
time.Duration, graceful bool) {
	server := &http.Server{
		Addr:         address,
		Handler:      this,
		ReadTimeout:  readTimeout,
		WriteTimeout: writeTimeout,
	}
	canHttps := tlsCertfile != "" && tlsKeyfile != ""
	var err error
	if !graceful {//如果非无间断启动则根据是否 https 分别启动服务
		if canHttps {
			err = server.ListenAndServeTLS(tlsCertfile, tlsKeyfile)
		} else {
			err = server.ListenAndServe()
		}

	} else {//通过 grace 进行无间断启动服务
		endRunning := make(chan bool, 1)
		graceServer := grace.NewServer(address, server, Log)
		if canHttps {
			go func() {
				time.Sleep(20 * time.Microsecond)
				if err = graceServer.ListenAndServeTLS(tlsCertfile, tlsKeyfile); err != nil {
```

```
                        err = fmt.Errorf("Grace-ListenAndServeTLS: %v, %d", err, os.Getpid())
                        time.Sleep(100 * time.Microsecond)
                        endRunning <- true
                    }
                }()
            } else {
                go func() {
                    // graceServer.Network = "tcp4"
                    if err = graceServer.ListenAndServe(); err != nil {
                        err = fmt.Errorf("Grace-ListenAndServe: %v, %d", err, os.Getpid())
                        time.Sleep(100 * time.Microsecond)
                        endRunning <- true
                    }
                }()
            }
            <-endRunning
        }
        if err != nil {
            Log.Fatal("%v", err)
            select {}
        }
}
```

Panic 处理

系统提供默认的 Panic 处理函数（app.go 中 defaultPanicStackFunc），可以可以通过该函数设置自己的 Panic 处理函数。

```
// 设置获取请求过程中恐慌 Stack 信息的函数(内部有默认实现)
func SetPanicStackFunc(fn func(rcv interface{}) string) {
    app.SetPanicStackFunc(fn)
}
```

Custom HTTP Errors

系统提供默认的 http 失败处理函数（app.go 中 defaultFailureHandler），一般来说，你需要定义自己的失败处理函数，方式如下：

```
// 设置失败状态默认的响应操作(内部有默认实现，app.go 中 func defaultFailureHandler)
func SetFailureHandler(fn func(c *Context, code int, errString string) error) {
    app.SetFailureHandler(fn)
}
```

12. 处理器 Handlers

```
一个复杂的 handler
var Index = ApiHandler{
	Desc:     "后台管理登录操作",
	Method: "POST|PUT",
	Params: []Param{
		{"user", "formData", true, "henry11111", "用户名"},
		{"user", "query", true, "henry22222", "用户名"},
		{"user", "path", true, "henry33333", "用户名"},
		{"password", "formData", true, "1111111111", "密码"},
		{"password", "query", true, "2222222222", "密码"},
		{"password", "path", true, "3333333333", "密码"},
	},
	Handler: func(c *Context) error {
		// 测试读取 cookie
		id := c.CookieParam(Config.Session.SessionName)
		c.Log().Info("cookie 中的%v: %#v", Config.Session.SessionName, id)

		// 测试 session
		c.Log().Info("从 session 读取上次请求的输入: %#v", c.GetSession("info"))

		c.SetSession("info", map[string]interface{}{
			"user":     c.FormParam("user"),
			"password": c.FormParam("password"),
		})
		c.Log().Info("path 用户名: %#v", c.PathParam("user"))
		c.Log().Info("query 用户名: %#v", c.QueryParam("user"))
		c.Log().Info("formData 用户名: %#v", c.FormParam("user"))
		c.Log().Info("path 密码: %#v", c.PathParam("password"))
		c.Log().Info("query 密码: %#v", c.QueryParam("password"))
		c.Log().Info("formData 密码: %#v", c.FormParam("password"))

		return c.Render(200,
			"sysview/admin/login/index.tpl",
			map[string]interface{}{
				"name":       c.FormParam("user"),
				"password":   c.FormParam("password"),
				"repeatfunc": admin.Login.Repeatfunc,
			},
		)
	},
}.Reg()
```

13. 中间件 Middleware

中间件 ApiMiddleware 与操作 ApiHandler 共同组成一条请求的执行链。常见用途有授权验证、登录验证等。中间件有两种三者定义形式：

最简的用法（举例）

```
var ShowHeader = lessgo.ApiMiddleware{
	Name:     "显示 Header",
	Desc:     "显示 Header 测试",
	Middleware: func(c *lessgo.Context) error {
		c.Log().Info("测试中间件-显示 Header：%v", c.Request().Header)
		return nil
	},
}.Reg()
```

过滤的用法（举例）

```
var CheckLogin = ApiMiddleware{
	Name: "登录验证",
	Desc: "登录验证，未登录时返回 -10",
	Params: []Param{
		{"token", "query", true, "", "用户登录 token"},
	},
	Middleware: func(next HandlerFunc) HandlerFunc {
		return func(c *Context) error {
			uid, islogin := CheckLogin(c.QueryParam("token"))
			if islogin {
				c.Set("uid", uid)
				return next(c)
			}
			return c. JSON (200, map[string] string{
				"code": "-10",
				"msg": "请先登录再进行操作",
			})
		}
	},
}.Reg()
```

支持动态配置的用法（举例）

```
var  lanPrefix = []string{
```

```
            "::",
            "127.",
            "192.168.",
            "10.",
        }
// The IPs which are allowed access.
var AllowIPPrefixes = lessgo.ApiMiddleware{
        Name:     "AllowIPPrefixes",
        Desc:     `The IP Prefixes which are allowed access.`,
        Config: lanPrefix,
        Middleware: func(confObject interface{}) lessgo.MiddlewareFunc {
            ips, _ := confObject.([]string)
            return func(next lessgo.HandlerFunc) lessgo.HandlerFunc {
                return func(c *lessgo.Context) error {
                    remoteAddress := c.RealRemoteAddr()
                    for i, count := 0, len(ips); i < count; i++ {
                        if strings.HasPrefix(remoteAddress, ips[i]) {
                            return next(c)
                        }
                    }

                    return c.Failure(http.StatusForbidden, errors.New(`Not allow your ip access:
`+c.RealRemoteAddr()))
                }
            }
        },
}.Reg()
```

运行期间，您可以通过 AllowIPPrefixes.SetConfig(confObject interface{})等方法来修改配置信息 lanPrefix。

14. 路由 Routing

源码路由

```
package router

import (
        "github.com/lessgo/lessgo"
        "github.com/lessgo/demo/bizhandler/home"
```

```
	"github.com/lessgo/demo/middleware"
)

func init() {
	lessgo.Root(
		lessgo.Leaf("/websocket", home.WebSocket, middleware.ShowHeader),
		lessgo.Branch("/home", "前台",
			lessgo.Leaf("/index", home.Index, middleware.ShowHeader),
		).Use(middleware.Print),
	)
}
```

动态路由

lessgo 支持从配置文件读取路由配置，支持运行时修改路由。路由配置文件在首次运行后，自动创建并保存于 config/ virtrouter.config，举例：

```
{
  "md5": "15204871614405858416",
  "virtrouter": {
    "id": "e93528e7-f0b9-4ba9-b137-4bfbab8ca30e",
    "type": 0,
    "prefix": "/",
    "middlewares": [],
    "enable": true,
    "dynamic": false,
    "hid": "7cb4e397",
    "children": [
      {
        "id": "6f6cd194-acf9-4e5d-a1d7-8226883624c1",
        "type": 1,
        "prefix": "/admin",
        "middlewares": [
          {
            "name": "登录验证",
            "config": ""
          }
        ],
        "enable": true,
        "dynamic": true,
        "hid": "c4abd3c5",
        "children": [
          {
```

```
                "id": "b84b5c3b-e76a-4aa0-ac97-df6299d72ecb",
                "type": 1,
                "prefix": "/login",
                "middlewares": [],
                "enable": true,
                "dynamic": true,
                "hid": "bf174920",
                "children": null
            }
        ]
    }
  ]
 }
}
```

用户无需重新编译程序仅通过修改上面配置就能实现修改路由的目的。
未来，lessgo 还会添加运行时修改路由的 http 接口，便于 admin 管理。

15. 参数 Parameters

ApiHandler. Params 和 ApiMiddleware. Params 的字段类型：

```
Param struct {
        Name        string          // (必填)参数名
        In          string          // (必填)参数出现位置
        Required bool               // (必填)是否必填
        Model       interface{} // (必填)参数值，API 文档中依此推断参数值类型，同时作为默认值，当为 nil 时表示 file 文件上传
        Desc        string          // (可选)参数描述
    }
```

它是基于 swagger 的参数规则进行定义的，目的是生成 swagger JSON 配置文件，实现自动化 API 文档。参数具体说明如下：
一、数据结构主要用于固定格式的服务器响应结构，适用于多个接口可能返回相同的数据结构，编辑保存后相关所有的引用都会变更。
支持的数据类型说明如下：
1、string:字符串类型
2、array:数组类型，子项只能是支持的数据类型中的一种，不能添加多个
3、object:对象类型，只支持一级属性，不支持嵌套，嵌套可以通过在属性中引入 ref 类型的对象或自定义数据格式
4、int:短整型
5、long:长整型
6、float:浮点型
7、double:浮点型
8、decimal:精确到比较高的浮点型
9、ref:引用类型，即引用定义好的数据结构

10、file:文件（Param.Model==nil）
*
二、参数位置
body：http 请求 body
cookie：本地 cookie
formData：表单参数
header：http 请求 header
path：http 请求 url,如 getInfo/{userId}
query：http 请求拼接，如 getInfo?userId={userId}
三、参数类型
自定义：目前仅支持自定义 json 格式，仅当"参数位置"为“body"有效

16. 上下文 Context

参数解析

```
// 获取 URL path 部分的参数，如“/get/:id”中的 id
func (c *Context) PathParam(key string) string
// 获取 URL query 部分的参数，如“/get?token=abc123”中的 token
func (c *Context) QueryParam(key string) string
// 获取提交表单中的参数
func (c *Context) FormParam(key string) string
// 获取 HTTP header 中的参数
func (c *Context) HeaderParam(key string) string
// 获取 cookie 中的参数
func (c *Context) CookieParam(key string) *http.Cookie
```

文件上传

```
// 获取客户端上传的文件句柄
func (c *Context) FormFile(key string) (multipart.File, *multipart.FileHeader, error)
// 获取客户端上传的文件并自动保存，返回该文件的 url；参数 cover 表示是否覆盖已存在的同名文件，若不覆盖则添加形式为“(2)”的序号后缀进行区分；参数 newfname 为自定义的保存文件名，注意自定义文件名不要含有扩展名，因为扩展名是根据原文件名自动添加的。
func (c *Context) SaveFile(key string, cover bool, newfname ...string) (fileUrl string, size int64, err error)
```

17. 静态文件 Static files

静态文件服务通过一下两个方法进行注册：

```
// 单独注册静态文件路由
func File(path, file string, middlewares ...interface{}) error
// 单独注册静态目录路由
func Static(prefix, root string, middlewares ...interface{}) error
```

另外，lessgo 默认注册的静态路由有：

```
/uploads // 对应项目中 uploads 目录，用于存储上传的文件
/static  // 对应项目中 static 目录，用于存放前台公共的静态文件，如 jQuery.js 文件等
/biz     // 对应项目中 biz_view 目录，用于存在业务前端静态文件
/sys     // 对应项目中 sys_view 目录，用于存在系统前端静态文件
/favicon.ico  // 对应项目中 static/img/ favicon.ico，是项目在前端显示的图标
```

静态文件服务支持缓存功能，通过在配置文件 config/app.config 中设置 debug=false，即关闭调试模式后启用。另外[filecache]分段中 cachesecond、maxcapmb、singlefileallowmb 三个配置项分别用于设置缓存时长、缓存总容量、允许缓存的单个文件大小。

18. Pongo2 Template

pongo 2 是一个语法与 Django 模板类似的 Go 语言模板引擎，并且完全兼容 Django 模板。pongo 2 提供了复杂和嵌套函数调用和强大的类 C 表达式。

lessgo 集成了 pongo2 作为默认的模板引擎。

Pongo2 调用

首先引入包 "github.com/lessgo/lessgo/pongo2"

调用代码：

```
var Pongo2 = lessgo.ApiHandler{
    Desc:   "模板测试",
    Method: "GET",
    Handler: func(c *lessgo.Context) error {
        return c.Render(200, "biz_view/home/pongo2.tpl", pongo2.Context{"bb": "上下文参数"})
    },
}.Reg()
```

模板文件：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
```

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <style type="text/css">
        .wrapper {
            background: #ECF0F5 !important;
        }
    </style>
</head>
<body>
 <div id="content1">
      <p>【数据库函数结果列表】</p>
       {{ bb }}
       {% set pp="{`aa`:`ppppppppp`}" %}
       {% for aa in SimpleData("biz/demo","selecttpl",pp) %}
           <p>名称:{{ aa.cnname }}</p>
       {% endfor %}
    </div>
    <div id="content2">
      <p>传入值</p>
       {{ bb }}
    </div>
</body>
</html>
```

Pongo2 扩展

Pongo2 支持扩展 filter、tag、以及通过其 context 传递任何值包括函数，lessgo 提供系统的 API 支持 pongo2 进行扩展 filter 和 context 变量传递（值或函数）

Lessgo 扩展 pongo2 的 API：

lessgo .TemplateVariable(name string, v interface{})

通过 TemplateVariable 注册的全局可用，即系统的所有模板中都可以使用。

说明，如果 v 的实际类型符合 pongo2 的 filter 格式，则注册为扩展的 filtetr，否则注册为 context 变量，可以是任意值或函数。

相关代码：

```
    switch d := v.(type) {
        case func(in *pongo2.Value, param *pongo2.Value) (out *pongo2.Value, err
    *pongo2.Error):  //注册为 Filter
            pongo2.RegisterFilter(name, d)
        case pongo2.FilterFunction: //注册为 Filter
            pongo2.RegisterFilter(name, d)
        default:  //全局的变量中（值，函数等）
            p.tplContext[name] = d
        }
```

特别注意：如果在调用 Render 时提供的参数名称与 TemplateVariable 中已经有的相同则使用 Render 提供的参数（即本次调用提供的参数自动覆盖全局的）。

Pongo2 与 directsql 扩展调用(待更新)

为了更好的通过服务端模板直接在后台集成数据绑定，系统提供了 3 个通过 directsql 执行 SQL 配置文件中 SQL 的函数，分别是：

```
//单个查询 参数 map[string]interface{} 返回 []map[string]interface{}
func SimpleData(modelId, sqlId string, mp string) *pongo2.Value
//获取单表分页数据 返回 []map[string]interface{}
func PagingData(modelId, sqlId string, mp string) *pongo2.Value
//多個查询 返回 map[string][]map[string]interface{}
func MultiData(modelId, sqlId string, mp string) *pongo2.Value
```

函数通过扩展 API 注册到系统中：

```
//获取单表数据 返回 []map[string]interface{}
lessgo.TemplateVariable("SimpleData", SimpleData)
//获取单表分页数据 返回 []map[string]interface{}
lessgo.TemplateVariable("PagingData", PagingData)
//获取多表数据 返回 map[string][]map[string]interface{}
lessgo.TemplateVariable("MultiData", MultiData)
```

在模板中使用：

```
{% for o in SimpleData("biz/users","selecttpl","{`para`:`我是 sql 参数值`}") %}
        <p>名称:{{ o.name }}</p>
    {% endfor %}
```

Pongo2 技术文档

请参考: https://github.com/flosch/pongo2

或者参考 畅雨 收集整理的 <<pongo2 文档集锦>>

19. Markdown

```
// 采用 github 风格的渲染方式，参数 file 为.md 文件全名，hasCatalog 表示是否自动添加目录索引
func (c *Context) Markdown(file string, hasCatalog ...bool) error
```

20. 日志 Logger

21. Cookies

```
// 从请求中读取 cookie
func (c *Context) CookieParam(key string) *http.Cookie
// 在响应中追加 cookie
func (c *Context) AddCookie(cookie *http.Cookie)
// 在响应中覆盖式地添加 cookie
func (c *Context) SetCookie(cookie *http.Cookie)
// 删除响应中的 cookie
func (c *Context) DelCookie()
```

22. Sessions

用户通过在配置文件 config/app.config 设置 sessionon=true 开启 session 功能。

```
// 向 session 中添加缓存信息
func (c *Context) SetSession(key interface{}, value interface{})
// 读取 session 中缓存信息
func (c *Context) GetSession(key interface{}) interface{}
// 删除 session 中指定缓存信息
func (c *Context) DelSession(key interface{})
// 更新 session ID
func (c *Context) SessionRegenerateID()
// 删除 session
func (c *Context) DestroySession()
```

23. 数据库访问 Database access

连接数据库并进行访问是绝大部分应用系统必须的功能，lessgo 通过开放的架构支持主流的 orm 以及 sql 方式访问数据库，主要包括：xorm，gorm，sqlx，以及自带扩展 directsql。具体实现见 github.com/lessgo/lessgoext/dbservice 目录。

Xorm 数据库访问

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。

Github： Github.com/go-xorm / xorm

驱动支持(具体以其官方为准)

目前支持的 Go 数据库驱动和对应的数据库如下：

- 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)

配置项

见配置章 xorm 节。

技术文档

对于 xorm 的使用请参考 xorm 相关文档，见 xorm 的官方网站文档 http://xorm.io/docs/。

使用步骤示例

1. 首先配置数据库连接文件，/config/xorm.config，具体配置信息见配置项
2. 系统启动时会根据配置项自动建立与数据库的连接
3. 在代码中使用：

```
import   "github.com/lessgo/lessgoext/dbservice/xorm"
...
 group := new(TGroup)
 has, err := xorm.DefaultDB().Id(id).Get(group)
 if err != nil {
       lessgo.Log.Error(err.Error())
       return nil, err
 }
...
```

代码中 **xorm.DefaultDB()**即为 xorm 访问数据的引擎，该示例为使用默认数据库，如果使用非默认数据库则：**xorm.GetDB("dbname")。**

Gorm 数据库访问

程序员友好的 GoLang ORM， 具有高易用性。
支持 CURD， 链式查询， 内嵌 struct，各种回调 callback 支持

支持 Rails 类似的 Update， Updates， FirstOrInit, FirstOrCreate 等功能
并且有自动的 CreatedAt, UpdatedAt， 软删除等等功能。

Github： github.com/jinzhu/gorm

驱动支持(具体以其官方为准)

目前支持的 Go 数据库驱动和对应的数据库如下：

- MySQL
- PostgreSQL
- Sqlite3
- Write Dialect for unsupported databases

配置项

```
[defaultdb]
connstring=database/sqlite.db
driver=sqlite3
maxidleconns=1
maxopenconns=1
name=lessgo
showsql=false
```

技术文档

对于 gorm 的使用请参考 gorm 相关文档, 见 gorm 的官方网站文档 http://jinzhu.me/gorm。

使用步骤示例

4. 首先配置数据库连接文件，/config/gorm.config，具体配置信息见配置项
5. 系统启动时会根据配置项自动建立与数据库的连接
6. 在代码中使用：

```
import    "github.com/lessgo/lessgoext/dbservice/gorm"
...
  ...
```

Sqlx 数据库访问

Directsql 访问数据库

Directsql 是类似 MyBatis 的 SQL 执行引擎。可以使用简单的 **XML** 用于配置和映射数据库中的表。通过 id 进行 SQL 调用并返回执行的结果。

依赖

Directsql 运行需要 xorm/core，Github： Github.com/go-xorm / core

驱动支持

目前支持的 Go 数据库驱动和对应的数据库如下：

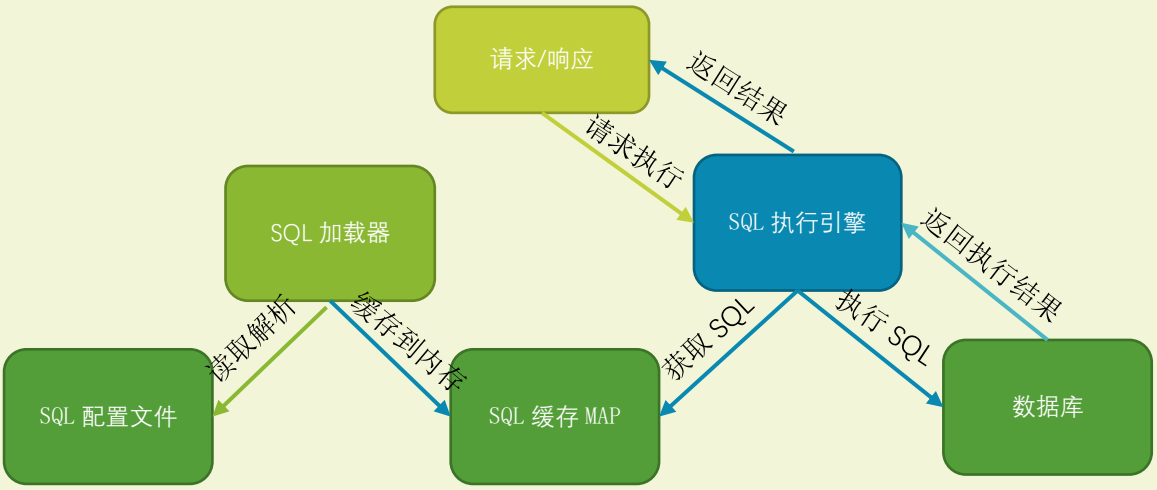
- Mysql: github.com/go-sql-driver/mysql
- MyMysql: github.com/ziutek/mymysql/godrv
- Postgres: github.com/lib/pq
- Tidb: github.com/pingcap/tidb
- SQLite: github.com/mattn/go-sqlite3
- MsSql: github.com/denisenkom/go-mssqldb
- MsSql: github.com/lunny/godbc
- Oracle: github.com/mattn/go-oci8 (试验性支持)

配置项

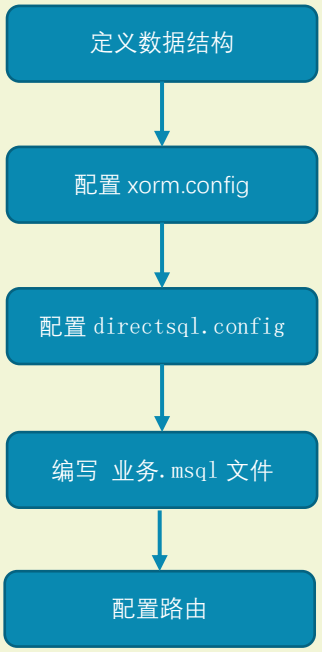
见配置章 directsql 节。

原理

应用启动时根据 xorm.config 的数据库连接信息建立数据库连接；SQL 加载器根据 directsql.config 中配置的 SQL 配置文件存放目录加载所有 SQL 配置文件到内存放入 MAP 中；根据 router 的设置响应请求根据配置文件路径以及 SQL 的 ID 与参数值进行调用，执行并返回 json 或 jsonp（带 callback 参数调用时返回 jsonp）结果集。



使用配置流程



SQL 配置类型

配置类型	内部类型	说明
select	ST_SELECT	0=普通查询
pagingselect	ST_PAGINGSELECT	//1=分页查询
multiselect	ST_MULTISELECT	//3=多结果集查询
delete	ST_DELETE	//4=删除
insert	ST_INSERT	//5=插入

update ST_UPDATE //6=更新
batchinsert ST_BATCHINSERT //7=批量插入(单数据集批量插入)
batchupdate ST_BATCHUPDATE //8=批量更新(单数据集批量更新)
batchcomplex ST_BATCHCOMPLEX //9=批量复合 SQL(一般多数据集批量插入或更新)---OK!
insertpro ST_INSERTPRO //10=插入升级版，可以在服务端生成 key 的 ID 并返回到客户端

SQL 配置文件详解

路由(调用 SQL 的 API 设置)

```
import github.com/lessgo/lessgoext/dbservice/directsql
    ...
lessgo.Root(
    ...
        // bos 执行 SQL 定义的路由
        lessgo.Leaf("/bos/*path", directsql.DirectSQL),
        //重新载入全部 SQL 配置文件
        lessgo.Leaf("/bom/reloadall", directsql.DirectSQLReloadAll),
        //重新载入某个具体的 SQL 配置文件
        lessgo.Leaf("/bom/reload/*path", directsql.DirectSQLReloadModel),
        ...
    )
```

客户端调用

三种类型调用参数：

- 简单 json 参数
  {"id":"001","name":"aaaaa"}
  其中：
  "callback":"可选参数，不为空时返回 JSONP",
  "start":"可选参数，分页查询时需要 开始记录数",
  "limted":"可选参数，分页查询时需要 每页记录数",
- 简单批量 json 参数
  [{"id":"001","name":"aaaaa"},{"id":"002","name":"bbbbb"},{"id":"003","name":"ccc"}]
- 复杂批量 json 参数
  {"main":[{"id":"01","name":"aaaaa"},{"id":"002","name":"bbbbb"}],

"sub1":[{"id":"0111","pid":"01","name":"sub11"},{"id":"0112","pid":"01","name":"sub12"}]
"sub2":[{"id":"0121","pid":"01","name":"sub21"},{"id":"0122","pid":"01","name":"sub22"}]
}

Ajax 请求：

使用示例

1. 创建数据库和表(针对 MySQL 数据库)

```
create database mydemo;
use mydemo;
CREATE TABLE users(id INT PRIMARY KEY AUTO_INCREMENT, NAME VARCHAR(20), age INT);
INSERT INTO users(NAME, age) VALUES('张三', 30);
INSERT INTO users(NAME, age) VALUES('李四',40);
```

2. 配置数据库连接文件 /config/xorm.config

```
;Mysql 连接配置段
[defautdb]
columnfix=prefix
columnsnake=true
columnspace=
connstring=root:pwd@tcp(127.0.0.1:3306)/ mydemo?charset=utf8
disablecache=false
driver=mysql
maxidleconns=100
maxopenconns=100
name=mysql
showexectime=false
showsql=false
tablefix=prefix
tablesnake=true
tablespace=
```

3. 配置 directsql.config

```
;SQL 配置文件扩展名，只能一个。
ext=.msql
;是否开始监控所有 roots 目录下的配置文件变化，改变自动处理(增加，删除，修改)
watch=true
;SQL 配置文件加载的根目录，可以个多个，定义后自动将真实文件名映射到前边名称
[roots]
biz=biz_model
sys=sys_model
```

4. 将路由信息添加到 router/sys_router.go 中

```
lessgo.Root(
```

```
        ...
        // bos 执行 SQL 定义的路由
        lessgo.Leaf("/bos/*path", directsql.DirectSQL),
        lessgo.Leaf("/bom/reloadall", directsql.DirectSQLReloadAll),
        lessgo.Leaf("/bom/reload/*path", directsql.DirectSQLReloadModel),
        ...
    )
```

5. 在 biz_model/下创建 user.msql 文件

```
<?xml version="1.0" encoding="utf-8"?>
<model id="users" database="">
<comment>
    <desc>DirectSQL 功能测试 SQL 定义</desc>
    <author></author>
    <date>2016-10-10</date>
    <history Editor="畅雨" Date="2016-10-20">完善并增加注释</history>
</comment>
<sql type="select" id="selectall" desc="查询 SQL">
    <cmd in="" out="">
        <![CDATA[   select * from users    ]]>
    </cmd>
</sql>
<!-- 下面可继续添加新的方法 -->
</model>
```

6. 执行测试
   - 启动应用
   - 打开浏览器，提交请求：http://localhost:8080/bos/biz/users/ selectall ，或者通过 ajax 进行 post 提交测试.

24. Issues

Application icon and sqlite/oracle gcc link error

25. F&Q

1. 找不到模板文件，找不到配置文件，nil 指针错误

   这种大多数情况是由于你采用了 `go run main.go` 这样的方式来运行你的应用，go run

是把文件编译之后放在了 tmp 下去运行，而 lessgo 的应用会读取应用的当前运行目录对应的 conf,view 去查找相应的配置文件和模板，因此要正确运行，请使用 `go build` 然后执行应用 `./app` 这种方式来运行。或者使用 `less run app` 来启动你的应用。

2. lessgo 可以应用于生产环境吗？

目前 lessgo 已经被广大用户应用于多种生产环境，包括 Web 应用和服务器应用，都是一些高并发高性能的应用，所以请放心使用。

3. lessgo 将来升级会影响现有的 API 吗？

lessgo 从 0.1 版本到现在基本保持了稳定的 API，很多应用都是可以无缝的升级到最新版本的 lessgo。将来升级重构都会尽量保持 lessgo 的 API 的稳定性。

4. lessgo 会持续开发吗？

很多人使用开源软件都有一个担心就是项目不能持续，目前我们 lessgo 开发组有三个人一直在贡献代码，我想我们能够坚持把这个项目做好，而且会持续不断的进行改进。

26. About

贡献者名单

贡献者	贡献概要
henrylee2cn	代码的主要实现者（第一作者）
changyu72	架构的主要设计者（第二作者）
LeSou	

开源协议

Lessgo 项目采用商业应用友好的 MIT 协议发布。

27. Appendix