net/http

简介

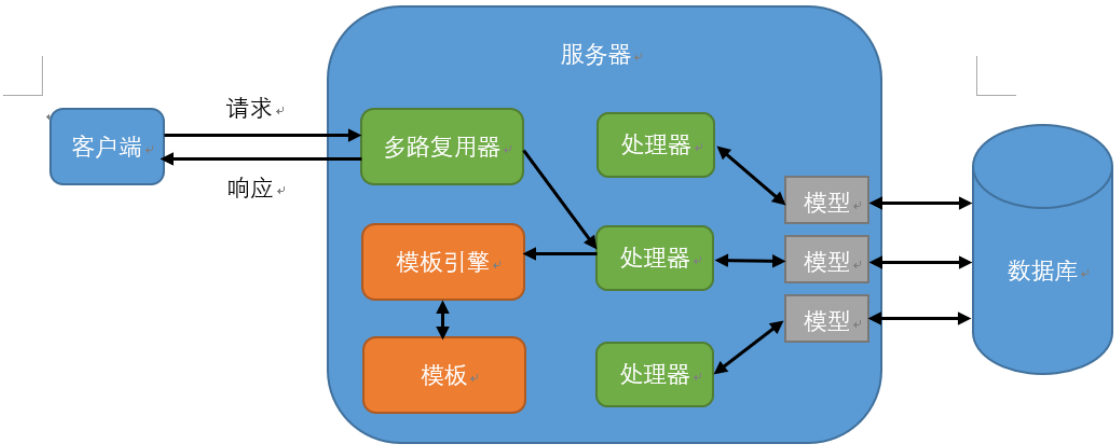
Web 应用简介

Web 应用在我们的生活中无处不在。看看我们日常使用的各个应用程序，它们要么是 Web 应用，要么是移动 App 这类 Web 应用的变种。无论哪一种编程语言，只要它能够开发出与人类交互的软件，它就必然会支持 Web 应用开发。对一门崭新的编程语言来说，它的开发者首先要做的一件事，就是构建与互联网(Internet)和万维网(World
Wide Web)交互的库(library)和框架，而那些更为成熟的编程语言还会有各种五花八门
的 Web 开发工具。

Go 是一门刚开始崭露头角的语言，它是为了让人们能够简单而高效地编写后端系统而创建的。这门语言拥有众多的先进特性，如函数式编程方面的特性、内置了对并发编程的支持、现代化的包管理系统、垃圾收集特性、以及一些包罗万象威力强大的标准库，而且如果需要我们还可以引入第三方开源库。

使用 Go 语言进行 Web 开发正变得日益流行，很多大公司都在使用，如 Google、Facebook、腾讯、百度、阿里巴巴、京东、小米以及 360、美团、滴滴以及新浪等。

Web 应用的工作原理



Web 应用简单工作流程

Hello World

下面，就让我们使用 Go 语言创建一个简单的 Web 应用。

1) 在 GOPATH 下的 src 目录下创建一个 webapp 的文件夹，并在该目录中创建一个
main.go 的文件，代码如下

```go
package main
import (
"fmt"
"net/http"
)
// 创建处理器函数
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello world!", r.URL.Path)
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

2） 在终端执行以下命令（使用 Vscode 开发工具时）：

a）方式一（建议使用）：在 webapp 目录中右键→在命令提示符中打开
执行 go build main.go 命令；然后在当前目录中就会生成一个 main.exe 的
二进制可执行文件；最后再执行 ./main.exe 就可以启动服务器
b）方式二：在 webapp 目录中右键→在命令提示符中打开

3） 在浏览器地址栏输入 http://localhost:8080，在浏览器中就会显示 Hello World! /
在浏览器地址栏输入 http://localhost:8080/hello，在浏览器中就会显示 Hello World!
/hello

Web服务器的创建

2.1 简介

Go 提供了一系列用于创建 Web 服务器的标准库。通过 net/http 包调用 ListenAndServe 函数并传入网络地址以及负责处理请求的处理器(handler)作为参数就可以了。

如果网络地址参数为空字符串，那么服务器默认使用 80 端口进行网络连接；如果处理器参数为 nil，那么服务器将使用默认的多路复用器 DefaultServeMux，当然，我们也可以通过调用 NewServeMux 函数创建一个多路复用器。多路复用器接收到用户的请求之后根据请求的 URL 来判断使用哪个处理器来处理请求，找到后就会重定向到对应的处理器来处理请求，

2.2 使用默认的多路复用器（DefaultServeMux）

HandleFunc方法和Handle方法区别

使用handle函数接收处理器，只要某个结构体实现了 Handler 接口中的 ServeHTTP 方法
那么它就是一个处理器

```
type MyHandler struct{}

func (m *MyHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    fmt.Fprintln(w, "通过自己创建的处理器处理请求！")
}

func main() {
    myHandler := MyHandler{}
    http.Handle("/myHandler", &myHandler)
    //创建路由
    http.ListenAndServe(":8080", nil)
}
```

通过 Server 结构对服务器进行更详细的配置

```
type MyHandler struct{}

func (m *MyHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    fmt.Fprintln(w, "通过详细配置处理器来处理请求！")
}

func main() {
    myHandler := MyHandler{}
    server := http.Server{
        Addr: ":8080",
        Handler: &myHandler,
        ReadTimeout: 2 * time.Second,
    }
    server.ListenAndServe()
}
```

创建自己的多路复用器

```
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "通过自己创建的多路复用器处理请求！")
}

func main() {
    //创建多路复用器
    mux := http.NewServeMux()

    mux.HandleFunc("/", handler)
    //创建路由
    http.ListenAndServe(":8080", mux)
}
```

请求报文
get/hello/index.jsp HTTP/1.1 //没有请求体

post/hello/index.jsp HTTP/1.1
name:zs
pwd:123

//响应报文
HTTP/1.1 200 OK

处理请求

获取请求行，请求头，请求体，请求参数

```
// 创建处理器函数
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "你发送的请求的请求地址是：", r.URL.Path)
    fmt.Fprintln(w, "你发送的请求的请求地址后的查询字符串(请求参数)
是：", r.URL.RawQuery)
    fmt.Fprintln(w, "请求头中的所有信息：", r.Header)
    fmt.Fprintln(w, "请求头中Accept-Encoding的信息是",
r.Header["Accept_Encoding"])
    fmt.Fprintln(w, "请求头中Accept-Encoding的属性值是",
r.Header["Accept_Encoding"])

    //获取请求体内容长度
    len := r.ContentLength
    //创建byte切片
    body := make([]byte, len)
    //将请求体中的内容读到body中
```

```
    r.Body.Read(body)
    fmt.Fprintln(w, "请求体中的内容有：", string(body))

    //解析表单，在调用r.Form之前必须执行该操作
    r.ParseForm()
    //获取请求参数
    //如果form表单的action属性的URL地址中也有与form表单参数名相同的请求参数，
    //那么参数值都可以得到，并且form表单中的参数值在URL参数值前面
    fmt.Fprintln(w, "请求参数有：", r.Form)
    fmt.Fprintln(w, "POST请求的form表单中的请求参数有：", r.PostForm)
    //通过直接调用FormValue方法和PostFormValue直接获取请求参数的值
    fmt.Fprintln(w, "请求参数username的值为：", r.FormValue("username"))
    fmt.Fprintln(w, "请求参数username的值为：", r.PostFormValue("username"))
}
```

给客户端响应

```
func handler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("你的请求我已经收到"))

    html := `
    <html>
        <head>
            <title>测试响应内容为网页</title>
            <meta charset="utf-8"/>
        </head>
        <body>
            我是以网页的形式响应过来的!
        </body>
    </html>`
    w.Write([]byte(html))

    //设置响应头中内容的类型
    w.Header().Set("Content-Type", "application/json")
    user := User{
        ID: 1,
        Username: "admin",
        Password: "123456",
    }
```

```
    //将 user 转换为 json 格式
    json, _ := json.Marshal(user)
    w.Write(json)

    //让客户端重定向
    //以下操作必须要在 WriteHeader 之前进行
    w.Header().Set("Location", "https:www.baidu.com")
    //设置响应的状态码
    w.WriteHeader(302)
}
```

模板与渲染

`html/template`包实现了数据驱动的模板，用于生成可防止代码注入的安全的HTML内容。它提供了和`text/template`包相同的接口，Go语言中输出HTML的场景都应使用`html/template`这个包。

在一些前后端不分离的Web架构中，我们通常需要在后端将一些数据渲染到HTML文档中，从而实现动态的网页（网页的布局和样式大致一样，但展示的内容并不一样）效果。

我们这里说的模板可以理解为事先定义好的HTML文档文件，模板渲染的作用机制可以简单理解为文本替换操作-使用相应的数据去替换HTML文档中事先准备好的标记。

很多编程语言的Web框架中都使用各种模板引擎，比如Python语言中Flask框架中使用的jinja2模板引擎。

Go语言的模板引擎

Go语言内置了文本模板引擎`text/template`和用于HTML文档的`html/template`。它们的作用机制可以简单归纳如下：

1. 模板文件通常定义为`.tmpl`和`.tpl`为后缀（也可以使用其他的后缀），必须使用`UTF8`编码。
2. 模板文件中使用`{{`和`}}`包裹和标识需要传入的数据。
3. 传给模板这样的数据就可以通过点号（`.`）来访问，如果数据是复杂类型的数据，可以通过{ { .FieldName }}来访问它的字段。
4. 除`{{`和`}}`包裹的内容外，其他内容均不做修改原样输出。

模板引擎的使用

Go语言模板引擎的使用可以分为三部分：定义模板文件、解析模板文件和模板渲染.

定义模板文件

其中，定义模板文件时需要我们按照相关语法规则去编写，后文会详细介绍。

解析模板文件

上面定义好了模板文件之后，可以使用下面的常用方法去解析模板文件，得到模板对象：

```
func (t *Template) Parse(src string) (*Template, error)
func ParseFiles(filenames ...string) (*Template, error)
func ParseGlob(pattern string) (*Template, error)
```

当然，你也可以使用`func New(name string) *Template`函数创建一个名为`name`的模板，然后对其调用上面的方法去解析模板字符串或模板文件。

模板渲染

渲染模板简单来说就是使用数据去填充模板，当然实际上可能会复杂很多。

```
func (t *Template) Execute(wr io.Writer, data interface{})
error
func (t *Template) ExecuteTemplate(wr io.Writer, name string,
data interface{}) error
```

基本示例

定义模板文件

我们按照Go模板语法定义一个`hello.tmpl`的模板文件，内容如下：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello</title>
</head>
<body>
    <p>Hello {{.}}</p>
</body>
</html>
```

解析和渲染模板文件

然后我们创建一个`main.go`文件，在其中写下HTTP server端代码如下：

```
// main.go

func sayHello(w http.ResponseWriter, r *http.Request) {
    // 解析指定文件生成模板对象
    tmpl, err := template.ParseFiles("./hello.tmpl")
    if err ≠ nil {
        fmt.Println("create template failed, err:", err)
        return
    }
    // 利用给定数据渲染模板，并将结果写入w
    tmpl.Execute(w, "沙河小王子")
}
func main() {
    http.HandleFunc("/", sayHello)
    err := http.ListenAndServe(":9090", nil)
    if err ≠ nil {
        fmt.Println("HTTP server failed,err:", err)
        return
    }
}
```

将上面的`main.go`文件编译执行，然后使用浏览器访问`http://127.0.0.1:9090`就能看到页面上显示了“Hello 沙河小王子”。 这就是一个最简单的模板渲染的示例，Go语言模板引擎详细用法请往下阅读。

模板语法

{{.}}

模板语法都包含在`{{`和`}}`中间，其中`{{.}}`中的点表示当前对象。

当我们传入一个结构体对象时，我们可以根据`.`来访问结构体的对应字段。例如：

```
// main.go

type UserInfo struct {
    Name   string
    Gender string
    Age    int
}
```

```go
func sayHello(w http.ResponseWriter, r *http.Request) {
    // 解析指定文件生成模板对象
    tmpl, err := template.ParseFiles("./hello.tmpl")
    if err != nil {
        fmt.Println("create template failed, err:", err)
        return
    }
    // 利用给定数据渲染模板，并将结果写入w
    user := UserInfo{
        Name:   "小王子",
        Gender: "男",
        Age:    18,
    }
    tmpl.Execute(w, user)
}
```

模板文件`hello.tmpl`内容如下：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello</title>
</head>
<body>
    <p>Hello {{.Name}}</p>
    <p>性别: {{.Gender}}</p>
    <p>年龄: {{.Age}}</p>
</body>
</html>
```

同理，当我们传入的变量是map时，也可以在模板文件中通过`.`根据key来取值。

注释

```
{{/* a comment */}}
注释，执行时会忽略。可以多行。注释不能嵌套，并且必须紧贴分界符始止。
```

pipeline

pipeline是指产生数据的操作。比如{{.}}、{{.Name}}等。Go的模板语法中支持使用管道符号|链接多个命令，用法和unix下的管道类似：|前面的命令会将运算结果(或返回值)传递给后一个命令的最后一个位置。

注意：并不是只有使用了|才是pipeline。Go的模板语法中，pipeline的概念是传递数据，只要能产生数据的，都是pipeline。

变量

我们还可以在模板中声明变量，用来保存传入模板的数据或其他语句生成的结果。具体语法如下：

```
$obj := {{.}}
```

其中$obj是变量的名字，在后续的代码中就可以使用该变量了。

移除空格

有时候我们在使用模板语法的时候会不可避免的引入一下空格或者换行符，这样模板最终渲染出来的内容可能就和我们想的不一样，这个时候可以使用{{-语法去除模板内容左侧的所有空白符号， 使用-}}去除模板内容右侧的所有空白符号。

例如：

```
{{- .Name -}}
```

注意：-要紧挨{{和}}，同时与模板值之间需要使用空格分隔。

条件判断

Go模板语法中的条件判断有以下几种：

```
{{if pipeline}} T1 {{end}}

{{if pipeline}} T1 {{else}} T0 {{end}}

{{if pipeline}} T1 {{else if pipeline}} T0 {{end}}
```

range

Go的模板语法中使用range关键字进行遍历，有以下两种写法，其中pipeline的值必须是数组、切片、字典或者通道。

```
{{range pipeline}} T1 {{end}}
如果pipeline的值其长度为0，不会有任何输出

{{range pipeline}} T1 {{else}} T0 {{end}}
如果pipeline的值其长度为0，则会执行T0。
```

with

```
{{with pipeline}} T1 {{end}}
如果pipeline为empty不产生输出，否则将dot设为pipeline的值并执行T1。不修改外面的dot。

{{with pipeline}} T1 {{else}} T0 {{end}}
如果pipeline为empty，不改变dot并执行T0，否则dot设为pipeline的值并执行T1。
```

预定义函数

执行模板时，函数从两个函数字典中查找：首先是模板函数字典，然后是全局函数字典。一般不在模板内定义函数，而是使用Funcs方法添加函数到模板里。

预定义的全局函数如下：

```
and
    函数返回它的第一个empty参数或者最后一个参数;
    就是说"and x y"等价于"if x then y else x"; 所有参数都会执行;
or
    返回第一个非empty参数或者最后一个参数;
    亦即"or x y"等价于"if x then x else y"; 所有参数都会执行;
not
    返回它的单个参数的布尔值的否定
len
    返回它的参数的整数类型长度
index
    执行结果为第一个参数以剩下的参数为索引/键指向的值;
    如"index x 1 2 3"返回x[1][2][3]的值; 每个被索引的主体必须是数组、切片或者字典。
print
    即fmt.Sprint
printf
    即fmt.Sprintf
println
    即fmt.Sprintln
```

```
html
    返回与其参数的文本表示形式等效的转义HTML。
    这个函数在html/template中不可用。
urlquery
    以适合嵌入到网址查询中的形式返回其参数的文本表示的转义值。
    这个函数在html/template中不可用。
js
    返回与其参数的文本表示形式等效的转义JavaScript。
call
    执行结果是调用第一个参数的返回值，该参数必须是函数类型，其余参数作为调用该函数的参数;
    如"call .X.Y 1 2"等价于go语言里的dot.X.Y(1, 2);
    其中Y是函数类型的字段或者字典的值，或者其他类似情况;
    call的第一个参数的执行结果必须是函数类型的值（和预定义函数如print明显不同）;
    该函数类型值必须有1到2个返回值，如果有2个则后一个必须是error接口类型;
    如果有2个返回值的方法返回的error非nil，模板执行会中断并返回给调用模板执行者该错误;
```

比较函数

布尔函数会将任何类型的零值视为假，其余视为真。

下面是定义为函数的二元比较运算的集合：

```
eq      如果arg1 == arg2则返回真
ne      如果arg1 != arg2则返回真
lt      如果arg1 < arg2则返回真
le      如果arg1 <= arg2则返回真
gt      如果arg1 > arg2则返回真
ge      如果arg1 >= arg2则返回真
```

为了简化多参数相等检测，eq（只有eq）可以接受2个或更多个参数，它会将第一个参数和其余参数依次比较，返回下式的结果：

```
{{eq arg1 arg2 arg3}}
```

比较函数只适用于基本类型（或重定义的基本类型，如"type Celsius float32"）。但是，整数和浮点数不能互相比较。

自定义函数

Go的模板支持自定义函数。

```
func sayHello(w http.ResponseWriter, r *http.Request) {
    htmlByte, err := ioutil.ReadFile("./hello.tmpl")
    if err ≠ nil {
        fmt.Println("read html failed, err:", err)
        return
    }
    // 自定义一个夸人的模板函数
    kua := func(arg string) (string, error) {
        return arg + "真帅", nil
    }
    // 采用链式操作在Parse之前调用Funcs添加自定义的kua函数
    tmpl, err :=
template.New("hello").Funcs(template.FuncMap{"kua":
kua}).Parse(string(htmlByte))
    if err ≠ nil {
        fmt.Println("create template failed, err:", err)
        return
    }

    user := UserInfo{
        Name:   "小王子",
        Gender: "男",
        Age:    18,
    }
    // 使用user渲染模板，并将结果写入w
    tmpl.Execute(w, user)
}
```

我们可以在模板文件`hello.tmpl`中按照如下方式使用我们自定义的`kua`函数了。

```
{{kua .Name}}
```

嵌套template

我们可以在template中嵌套其他的template。这个template可以是单独的文件，也可以是通过`define`定义的template。

举个例子： `t.tmpl`文件内容如下：

```
<!DOCTYPE html>
```

```
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>tmpl test</title>
</head>
<body>

    <h1>测试嵌套template语法</h1>
    <hr>
    {{template "ul.tmpl"}}
    <hr>
    {{template "ol.tmpl"}}
</body>
</html>

{{ define "ol.tmpl"}}
<ol>
    <li>吃饭</li>
    <li>睡觉</li>
    <li>打豆豆</li>
</ol>
{{end}}
```

ul.tmpl文件内容如下：

```
<ul>
    <li>注释</li>
    <li>日志</li>
    <li>测试</li>
</ul>
```

我们注册一个templDemo路由处理函数.

```
http.HandleFunc("/tmpl", tmplDemo)
```

tmplDemo函数的具体内容如下：

```go
func tmplDemo(w http.ResponseWriter, r *http.Request) {
    tmpl, err := template.ParseFiles("./t.tmpl", "./ul.tmpl")
    if err != nil {
        fmt.Println("create template failed, err:", err)
        return
    }
    user := UserInfo{
        Name:   "小王子",
        Gender: "男",
        Age:    18,
    }
    tmpl.Execute(w, user)
}
```

注意：在解析模板时，被嵌套的模板一定要在后面解析，例如上面的示例中`t.tmpl`模板中嵌套了`ul.tmpl`，所以`ul.tmpl`要在`t.tmpl`后进行解析。

block

```
{{block "name" pipeline}} T1 {{end}}
```

`block`是定义模板`{{define "name"}} T1 {{end}}`和执行`{{template "name" pipeline}}`缩写，典型的用法是定义一组根模板，然后通过在其中重新定义块模板进行自定义。

定义一个根模板`templates/base.tmpl`，内容如下：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <title>Go Templates</title>
</head>
<body>
<div class="container-fluid">
    {{block "content" . }}{{end}}
</div>
</body>
</html>
```

然后定义一个`templates/index.tmpl`，“继承”`base.tmpl`：

```
{{template "base.tmpl"}}

{{define "content"}}
    <div>Hello world!</div>
{{end}}
```

然后使用`template.ParseGlob`按照正则匹配规则解析模板文件，然后通过`ExecuteTemplate`渲染指定的模板：

```
func index(w http.ResponseWriter, r *http.Request){
    tmpl, err := template.ParseGlob("templates/*.tmpl")
    if err != nil {
        fmt.Println("create template failed, err:", err)
        return
    }
    err = tmpl.ExecuteTemplate(w, "index.tmpl", nil)
    if err != nil {
        fmt.Println("render template failed, err:", err)
        return
    }
}
```

如果我们的模板名称冲突了，例如不同业务线下都定义了一个`index.tmpl`模板，我们可以通过下面两种方法来解决。

1. 在模板文件开头使用`{{define 模板名}}`语句显式的为模板命名。
2. 可以把模板文件存放在`templates`文件夹下面的不同目录中，然后使用`template.ParseGlob("templates/**/*.tmpl")`解析模板。

修改默认的标识符

Go标准库的模板引擎使用的花括号`{{`和`}}`作为标识，而许多前端框架（如`Vue`和`AngularJS`）也使用`{{`和`}}`作为标识符，所以当我们同时使用Go语言模板引擎和以上前端框架时就会出现冲突，这个时候我们需要修改标识符，修改前端的或者修改Go语言的。这里演示如何修改Go语言模板引擎默认的标识符：

```
template.New("test").Delims("{[", "]}").ParseFiles("./t.tmpl")
```

text/template与html/tempalte的区别

`html/template`针对的是需要返回HTML内容的场景，在模板渲染过程中会对一些有风险的内容进行转义，以此来防范跨站脚本攻击。

例如，我定义下面的模板文件：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello</title>
</head>
<body>
    {{.}}
</body>
</html>
```

这个时候传入一段JS代码并使用`html/template`去渲染该文件，会在页面上显示出转义后的JS内容。`<script>alert('嘿嘿嘿')</script>` 这就是`html/template`为我们做的事。

但是在某些场景下，我们如果相信用户输入的内容，不想转义的话，可以自行编写一个safe函数，手动返回一个`template.HTML`类型的内容。示例如下：

```
func xss(w http.ResponseWriter, r *http.Request){
    tmpl,err :=
template.New("xss.tmpl").Funcs(template.FuncMap{
        "safe": func(s string)template.HTML {
            return template.HTML(s)
        },
    }).ParseFiles("./xss.tmpl")
    if err ≠ nil {
        fmt.Println("create template failed, err:", err)
        return
    }
    jsStr := `<script>alert('嘿嘿嘿')</script>`
    err = tmpl.Execute(w, jsStr)
    if err ≠ nil {
        fmt.Println(err)
    }
}
```

这样我们只需要在模板文件不需要转义的内容后面使用我们定义好的safe函数就可以了。

```
{{ . | safe }}
```

Gin框架介绍及使用

Gin是一个用Go语言编写的web框架。它是一个类似于martini但拥有更好性能的API框架，由于使用了httprouter，速度提高了近40倍。 如果你是性能和高效的追求者，你会爱上Gin。

Gin框架介绍

Go世界里最流行的Web框架，Github上有32K+star。 基于httprouter开发的Web框架。 中文文档齐全，简单易用的轻量级框架。

Gin框架安装与使用

安装

下载并安装Gin：

```
go get -u github.com/gin-gonic/gin
```

第一个Gin示例：

```
package main

import (
    "github.com/gin-gonic/gin"
)

func main() {
    // 创建一个默认的路由引擎
    r := gin.Default()
    // GET：请求方式；/hello：请求的路径
    // 当客户端以GET方法请求/hello路径时，会执行后面的匿名函数
    r.GET("/hello", func(c *gin.Context) {
        // c.JSON：返回JSON格式的数据
        c.JSON(200, gin.H{
            "message": "Hello world!",
        })
    })
    // 启动HTTP服务，默认在0.0.0.0:8080启动服务
    r.Run()
}
```

将上面的代码保存并编译执行，然后使用浏览器打开127.0.0.1:8080/hello就能看到一串JSON字符串。

RESTful API

REST与技术无关，代表的是一种软件架构风格，REST是Representational State Transfer的简称，中文翻译为“表征状态转移”或“表现层状态转化”。

推荐阅读阮一峰 理解RESTful架构

简单来说，REST的含义就是客户端与Web服务器之间进行交互的时候，使用HTTP协议中的4个请求方法代表不同的动作。

- GET用来获取资源
- POST用来新建资源
- PUT用来更新资源
- DELETE用来删除资源。

只要API程序遵循了REST风格，那就可以称其为RESTful API。目前在前后端分离的架构中，前后端基本都是通过RESTful API来进行交互。

例如，我们现在要编写一个管理书籍的系统，我们可以查询对一本书进行查询、创建、更新和删除等操作，我们在编写程序的时候就要设计客户端浏览器与我们Web服务端交互的方式和路径。按照经验我们通常会设计成如下模式：

请求方法	URL	含义
GET	/book	查询书籍信息
POST	/create_book	创建书籍记录
POST	/update_book	更新书籍信息
POST	/delete_book	删除书籍信息

同样的需求我们按照RESTful API设计如下：

请求方法	URL	含义
GET	/book	查询书籍信息
POST	/book	创建书籍记录
PUT	/book	更新书籍信息
DELETE	/book	删除书籍信息

Gin框架支持开发RESTful API的开发。

```
func main() {
    r := gin.Default()
    r.GET("/book", func(c *gin.Context) {
```

```
        c.JSON(200, gin.H{
            "message": "GET",
        })
    })

    r.POST("/book", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "POST",
        })
    })

    r.PUT("/book", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "PUT",
        })
    })

    r.DELETE("/book", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "DELETE",
        })
    })
}
```

开发RESTful API的时候我们通常使用Postman来作为客户端的测试工具。

Gin渲染

HTML渲染

我们首先定义一个存放模板文件的templates文件夹，然后在其内部按照业务分别定义一个posts文件夹和一个users文件夹。 posts/index.html文件的内容如下：

```
{{define "posts/index.html"}}
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>posts/index</title>
</head>
```

```
<body>
    {{.title}}
</body>
</html>
{{end}}
```

users/index.html文件的内容如下：

```
{{define "users/index.html"}}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>users/index</title>
</head>
<body>
    {{.title}}
</body>
</html>
{{end}}
```

Gin框架中使用LoadHTMLGlob()或者LoadHTMLFiles()方法进行HTML模板渲染。

```
func main() {
    r := gin.Default()
    r.LoadHTMLGlob("templates/**/*")
    //r.LoadHTMLFiles("templates/posts/index.html",
"templates/users/index.html")
    r.GET("/posts/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "posts/index.html", gin.H{
            "title": "posts/index",
        })
    })

    r.GET("users/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "users/index.html", gin.H{
            "title": "users/index",
        })
    })
```

```
    r.Run(":8080")
}
```

自定义模板函数

定义一个不转义相应内容的`safe`模板函数如下：

```
func main() {
    router := gin.Default()
    router.SetFuncMap(template.FuncMap{
        "safe": func(str string) template.HTML{
            return template.HTML(str)
        },
    })
    router.LoadHTMLFiles("./index.tmpl")

    router.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.tmpl", "<a
href='https://liwenzhou.com'>李文周的博客</a>")
    })

    router.Run(":8080")
}
```

在`index.tmpl`中使用定义好的`safe`模板函数：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <title>修改模板引擎的标识符</title>
</head>
<body>
<div>{{ . | safe }}</div>
</body>
</html>
```

静态文件处理

当我们渲染的HTML文件中引用了静态文件(.css js文件 图片等)时，我们只需要按照以下方式在渲染页面前调用`gin.Static`方法即可。

```
func main() {
    r := gin.Default()
    r.Static("/static", "./static")
    r.LoadHTMLGlob("templates/**/*")
   // ...
    r.Run(":8080")
}
```

使用模板继承

Gin框架默认都是使用单模板，如果需要使用`block template`功能，可以通过`"github.com/gin-contrib/multitemplate"`库实现，具体示例如下：

首先，假设我们项目目录下的templates文件夹下有以下模板文件，其中`home.tmpl`和`index.tmpl`继承了`base.tmpl`：

```
templates
├── includes
│   ├── home.tmpl
│   └── index.tmpl
├── layouts
│   └── base.tmpl
└── scripts.tmpl
```

然后我们定义一个`loadTemplates`函数如下：

```
func loadTemplates(templatesDir string) multitemplate.Renderer
{
    r := multitemplate.NewRenderer()
    layouts, err := filepath.Glob(templatesDir +
"/layouts/*.tmpl")
    if err ≠ nil {
        panic(err.Error())
    }
    includes, err := filepath.Glob(templatesDir +
"/includes/*.tmpl")
    if err ≠ nil {
        panic(err.Error())
    }
    // 为layouts/和includes/目录生成 templates map
    for _, include := range includes {
        layoutCopy := make([]string, len(layouts))
        copy(layoutCopy, layouts)
        files := append(layoutCopy, include)
```

```
        r.AddFromFiles(filepath.Base(include), files...)
    }
    return r
}
```

我们在`main`函数中

```
func indexFunc(c *gin.Context){
    c.HTML(http.StatusOK, "index.tmpl", nil)
}

func homeFunc(c *gin.Context){
    c.HTML(http.StatusOK, "home.tmpl", nil)
}

func main(){
    r := gin.Default()
    r.HTMLRender = loadTemplates("./templates")
    r.GET("/index", indexFunc)
    r.GET("/home", homeFunc)
    r.Run()
}
```

补充文件路径处理

关于模板文件和静态文件的路径，我们需要根据公司/项目的要求进行设置。可以使用下面的函数获取当前执行程序的路径。

```
func getCurrentPath() string {
    if ex, err := os.Executable(); err == nil {
        return filepath.Dir(ex)
    }
    return "./"
}
```

JSON渲染

```
func main() {
    r := gin.Default()

    // gin.H 是map[string]interface{}的缩写
    r.GET("/someJSON", func(c *gin.Context) {
        // 方式一：自己拼接JSON
```

```
        c.JSON(http.StatusOK, gin.H{"message": "Hello world!"})
    })
    r.GET("/moreJSON", func(c *gin.Context) {
        // 方法二：使用结构体
        var msg struct {
            Name    string `json:"user"`
            Message string
            Age     int
        }
        msg.Name = "小王子"
        msg.Message = "Hello world!"
        msg.Age = 18
        c.JSON(http.StatusOK, msg)
    })
    r.Run(":8080")
}
```

XML渲染

注意需要使用具名的结构体类型。

```
func main() {
    r := gin.Default()
    // gin.H 是map[string]interface{}的缩写
    r.GET("/someXML", func(c *gin.Context) {
        // 方式一：自己拼接JSON
        c.XML(http.StatusOK, gin.H{"message": "Hello world!"})
    })
    r.GET("/moreXML", func(c *gin.Context) {
        // 方法二：使用结构体
        type MessageRecord struct {
            Name    string
            Message string
            Age     int
        }
        var msg MessageRecord
        msg.Name = "小王子"
        msg.Message = "Hello world!"
        msg.Age = 18
        c.XML(http.StatusOK, msg)
    })
    r.Run(":8080")
}
```

YMAL渲染

```
r.GET("/someYAML", func(c *gin.Context) {
    c.YAML(http.StatusOK, gin.H{"message": "ok", "status":
http.StatusOK})
})
```

protobuf渲染

```
r.GET("/someProtoBuf", func(c *gin.Context) {
    reps := []int64{int64(1), int64(2)}
    label := "test"
    // protobuf 的具体定义写在 testdata/protoexample 文件中。
    data := &protoexample.Test{
        Label: &label,
        Reps:  reps,
    }
    // 请注意，数据在响应中变为二进制数据
    // 将输出被 protoexample.Test protobuf 序列化了的数据
    c.ProtoBuf(http.StatusOK, data)
})
```

获取参数

获取querystring参数

querystring指的是URL中?后面携带的参数，例如：/user/search?username=小王子&address=沙河。 获取请求的querystring参数的方法如下：

```
func main() {
    //Default返回一个默认的路由引擎
    r := gin.Default()
    r.GET("/user/search", func(c *gin.Context) {
        username := c.DefaultQuery("username", "小王子")
        //username := c.Query("username")
        address := c.Query("address")
        //输出json结果给调用方
        c.JSON(http.StatusOK, gin.H{
            "message":  "ok",
            "username": username,
            "address":  address,
        })
    })
```

```
    r.Run()
}
```

获取form参数

当前端请求的数据通过form表单提交时，例如向`/user/search`发送一个POST请求，获取请求数据的方式如下：

```
func main() {
    //Default返回一个默认的路由引擎
    r := gin.Default()
    r.POST("/user/search", func(c *gin.Context) {
        // DefaultPostForm取不到值时会返回指定的默认值
        //username := c.DefaultPostForm("username", "小王子")
        username := c.PostForm("username")
        address := c.PostForm("address")
        //输出json结果给调用方
        c.JSON(http.StatusOK, gin.H{
            "message":  "ok",
            "username": username,
            "address":  address,
        })
    })
    r.Run(":8080")
}
```

获取JSON参数

当前端请求的数据通过JSON提交时，例如向`/json`发送一个JSON格式的POST请求，则获取请求参数的方式如下：

```
r.POST("/json", func(c *gin.Context) {
    // 注意：下面为了举例子方便，暂时忽略了错误处理
    b, _ := c.GetRawData()  // 从c.Request.Body读取请求数据
    // 定义map或结构体
    var m map[string]interface{}
    // 反序列化
    _ = json.Unmarshal(b, &m)

    c.JSON(http.StatusOK, m)
})
```

更便利的获取请求参数的方式，参见下面的 **参数绑定** 小节。

获取path参数

请求的参数通过URL路径传递，例如：`/user/search/小王子/沙河`。 获取请求URL路径中的参数的方式如下。

```
func main() {
    //Default返回一个默认的路由引擎
    r := gin.Default()
    r.GET("/user/search/:username/:address", func(c
*gin.Context) {
        username := c.Param("username")
        address := c.Param("address")
        //输出json结果给调用方
        c.JSON(http.StatusOK, gin.H{
            "message":  "ok",
            "username": username,
            "address":  address,
        })
    })

    r.Run(":8080")
}
```

参数绑定

为了能够更方便的获取请求相关参数，提高开发效率，我们可以基于请求的`Content-Type`识别请求数据类型并利用反射机制自动提取请求中`QueryString`、`form表单`、`JSON`、`XML`等参数到结构体中。 下面的示例代码演示了`.ShouldBind()`强大的功能，它能够基于请求自动提取`JSON`、`form表单`和`QueryString`类型的数据，并把值绑定到指定的结构体对象。

```
// Binding from JSON
type Login struct {
    User     string `form:"user" json:"user"
binding:"required"`
    Password string `form:"password" json:"password"
binding:"required"`
}

func main() {
    router := gin.Default()

    // 绑定JSON的示例 ({"user": "q1mi", "password": "123456"})
    router.POST("/loginJSON", func(c *gin.Context) {
```

```
        var login Login

        if err := c.ShouldBind(&login); err == nil {
            fmt.Printf("login info:%#v\n", login)
            c.JSON(http.StatusOK, gin.H{
                "user":     login.User,
                "password": login.Password,
            })
        } else {
            c.JSON(http.StatusBadRequest, gin.H{"error":
err.Error()})
        }
    })

    // 绑定form表单示例 (user=q1mi&password=123456)
    router.POST("/loginForm", func(c *gin.Context) {
        var login Login
        // ShouldBind()会根据请求的Content-Type自行选择绑定器
        if err := c.ShouldBind(&login); err == nil {
            c.JSON(http.StatusOK, gin.H{
                "user":     login.User,
                "password": login.Password,
            })
        } else {
            c.JSON(http.StatusBadRequest, gin.H{"error":
err.Error()})
        }
    })

    // 绑定QueryString示例 (/loginQuery?
user=q1mi&password=123456)
    router.GET("/loginForm", func(c *gin.Context) {
        var login Login
        // ShouldBind()会根据请求的Content-Type自行选择绑定器
        if err := c.ShouldBind(&login); err == nil {
            c.JSON(http.StatusOK, gin.H{
                "user":     login.User,
                "password": login.Password,
            })
        } else {
            c.JSON(http.StatusBadRequest, gin.H{"error":
err.Error()})
        }
```

```
    })

    // Listen and serve on 0.0.0.0:8080
    router.Run(":8080")
}
```

ShouldBind会按照下面的顺序解析请求中的数据完成绑定：

1. 如果是 GET 请求，只使用 Form 绑定引擎（query）。
2. 如果是 POST 请求，首先检查 content-type 是否为 JSON 或 XML，然后再使用 Form（form-data）。

文件上传

单个文件上传

文件上传前端页面代码：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <title>上传文件示例</title>
</head>
<body>
<form action="/upload" method="post" enctype="multipart/form-data">
    <input type="file" name="f1">
    <input type="submit" value="上传">
</form>
</body>
</html>
```

后端gin框架部分代码：

```
func main() {
    router := gin.Default()
    // 处理multipart forms提交文件时默认的内存限制是32 MiB
    // 可以通过下面的方式修改
    // router.MaxMultipartMemory = 8 << 20  // 8 MiB
    router.POST("/upload", func(c *gin.Context) {
        // 单个文件
        file, err := c.FormFile("f1")
        if err != nil {
            c.JSON(http.StatusInternalServerError, gin.H{
```

```
                "message": err.Error(),
            })
            return
        }

        log.Println(file.Filename)
        dst := fmt.Sprintf("C:/tmp/%s", file.Filename)
        // 上传文件到指定的目录
        c.SaveUploadedFile(file, dst)
        c.JSON(http.StatusOK, gin.H{
            "message": fmt.Sprintf("'%s' uploaded!",
file.Filename),
        })
    })
    router.Run()
}
```

多个文件上传

```
func main() {
    router := gin.Default()
    // 处理multipart forms提交文件时默认的内存限制是32 MiB
    // 可以通过下面的方式修改
    // router.MaxMultipartMemory = 8 << 20  // 8 MiB
    router.POST("/upload", func(c *gin.Context) {
        // Multipart form
        form, _ := c.MultipartForm()
        files := form.File["file"]

        for index, file := range files {
            log.Println(file.Filename)
            dst := fmt.Sprintf("C:/tmp/%s_%d", file.Filename,
index)
            // 上传文件到指定的目录
            c.SaveUploadedFile(file, dst)
        }
        c.JSON(http.StatusOK, gin.H{
            "message": fmt.Sprintf("%d files uploaded!",
len(files)),
        })
    })
    router.Run()
}
```

重定向

HTTP重定向

HTTP 重定向很容易。 内部、外部重定向均支持。

```
r.GET("/test", func(c *gin.Context) {
    c.Redirect(http.StatusMovedPermanently,
"http://www.sogo.com/")
})
```

路由重定向

路由重定向，使用`HandleContext`：

```
r.GET("/test", func(c *gin.Context) {
    // 指定重定向的URL
    c.Request.URL.Path = "/test2"
    r.HandleContext(c)
})
r.GET("/test2", func(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{"hello": "world"})
})
```

Gin路由

普通路由

```
r.GET("/index", func(c *gin.Context) {...})
r.POST("/login", func(c *gin.Context) {...})
r.PUT("/login", func(c *gin.Context) {...})
r.DELETE("/login", func(c *gin.Context) {...})
```

此外，还有一个可以匹配所有请求方法的`Any`方法如下：

```
r.Any("/test", func(c *gin.Context) {...})
```

为没有配置处理函数的路由添加处理程序，默认情况下它返回404代码，下面的代码为没有匹配到路由的请求都返回`views/404.html`页面。

```
r.NoRoute(func(c *gin.Context) {
        c.HTML(http.StatusNotFound, "views/404.html", nil)
    })
```

路由组

我们可以将拥有共同URL前缀的路由划分为一个路由组。习惯性一对{}包裹同组的路由，这只是为了看着清晰，你用不用{}包裹功能上没什么区别。

```
func main() {
    r := gin.Default()
    userGroup := r.Group("/user")
    {
        userGroup.GET("/index", func(c *gin.Context) {...})
        userGroup.GET("/login", func(c *gin.Context) {...})
        userGroup.POST("/login", func(c *gin.Context) {...})

    }
    shopGroup := r.Group("/shop")
    {
        shopGroup.GET("/index", func(c *gin.Context) {...})
        shopGroup.GET("/cart", func(c *gin.Context) {...})
        shopGroup.POST("/checkout", func(c *gin.Context) {...})
    }
    r.Run()
}
```

路由组也是支持嵌套的，例如：

```
shopGroup := r.Group("/shop")
    {
        shopGroup.GET("/index", func(c *gin.Context) {...})
        shopGroup.GET("/cart", func(c *gin.Context) {...})
        shopGroup.POST("/checkout", func(c *gin.Context) {...})
        // 嵌套路由组
        xx := shopGroup.Group("xx")
        xx.GET("/oo", func(c *gin.Context) {...})
    }
```

通常我们将路由分组用在划分业务逻辑或划分API版本时。

路由原理

Gin框架中的路由使用的是httprouter这个库。

其基本原理就是构造一个路由地址的前缀树。

Gin中间件

Gin框架允许开发者在处理请求的过程中，加入用户自己的钩子（Hook）函数。这个钩子函数就叫中间件，中间件适合处理一些公共的业务逻辑，比如登录认证、权限校验、数据分页、记录日志、耗时统计等。

定义中间件

Gin中的中间件必须是一个`gin.HandlerFunc`类型。`func(*gin.Context)`类型

记录接口耗时的中间件

例如我们像下面的代码一样定义一个统计请求耗时的中间件。

```
// StatCost 是一个统计耗时请求耗时的中间件
func StatCost() gin.HandlerFunc {
    return func(c *gin.Context) {
        start := time.Now()
        c.Set("name", "小王子") // 可以通过c.Set在请求上下文中设置值，后续的处理函数能够取到该值
        // 调用该请求的剩余处理程序，后续程序处理完后回来继续执行
        c.Next()
        // 不调用该请求的剩余处理程序
        // c.Abort()
        // 计算耗时
        cost := time.Since(start)
        log.Println(cost)
    }
}

func main() {
    r := gin.Default()
    r.Get("/index", StatCost(), f1)
    r.GET("/index", f1)
}
```

记录响应体的中间件

我们有时候可能会想要记录下某些情况下返回给客户端的响应数据，这个时候就可以编写一个中间件来搞定。

```
type bodyLogWriter struct {
    gin.ResponseWriter               // 嵌入gin框架ResponseWriter
    body               *bytes.Buffer // 我们记录用的response
```

```
}

// Write 写入响应体数据
func (w bodyLogWriter) Write(b []byte) (int, error) {
    w.body.Write(b)                  // 我们记录一份
    return w.ResponseWriter.Write(b) // 真正写入响应
}

// ginBodyLogMiddleware 一个记录返回给客户端响应体的中间件
// https://stackoverflow.com/questions/38501325/how-to-log-response-body-in-gin
func ginBodyLogMiddleware(c *gin.Context) {
    blw := &bodyLogWriter{body: bytes.NewBuffer([]byte{}), ResponseWriter: c.Writer}
    c.Writer = blw // 使用我们自定义的类型替换默认的

    c.Next() // 执行业务逻辑

    fmt.Println("Response body: " + blw.body.String()) // 事后按需记录返回的响应
}
```

跨域中间件cors

推荐使用社区的https://github.com/gin-contrib/cors 库，一行代码解决前后端分离架构下的跨域问题。

注意： 该中间件需要注册在业务处理函数前面。

这个库支持各种常用的配置项，具体使用方法如下。

```
package main

import (
  "time"

  "github.com/gin-contrib/cors"
  "github.com/gin-gonic/gin"
)

func main() {
  router := gin.Default()
  // CORS for https://foo.com and https://github.com origins, allowing:
  // - PUT and PATCH methods
```

```
  // - Origin header
  // - Credentials share
  // - Preflight requests cached for 12 hours
  router.Use(cors.New(cors.Config{
    AllowOrigins:     []string{"https://foo.com"},  // 允许跨域发来请求的网站
    AllowMethods:     []string{"GET", "POST", "PUT", "DELETE", "OPTIONS"},  // 允许的请求方法
    AllowHeaders:     []string{"Origin", "Authorization", "Content-Type"},
    ExposeHeaders:    []string{"Content-Length"},
    AllowCredentials: true,
    AllowOriginFunc: func(origin string) bool {  // 自定义过滤源站的方法
      return origin == "https://github.com"
    },
    MaxAge: 12 * time.Hour,
  }))
  router.Run()
}
```

当然你可以简单的像下面的示例代码那样使用默认配置，允许所有的跨域请求。

```
func main() {
  router := gin.Default()
  // same as
  // config := cors.DefaultConfig()
  // config.AllowAllOrigins = true
  // router.Use(cors.New(config))
  router.Use(cors.Default())
  router.Run()
}
```

注册中间件

在gin框架中，我们可以为每个路由添加任意数量的中间件。

为全局路由注册

```
func main() {
    // 新建一个没有任何默认中间件的路由
    r := gin.New()
    // 注册一个全局中间件
    r.Use(StatCost())
```

```
    r.GET("/test", func(c *gin.Context) {
        name := c.MustGet("name").(string) // 从上下文取值
        log.Println(name)
        c.JSON(http.StatusOK, gin.H{
            "message": "Hello world!",
        })
    })
    r.Run()
}
```

为某个路由单独注册

```
// 给/test2路由单独注册中间件（可注册多个）
    r.GET("/test2", StatCost(), func(c *gin.Context) {
        name := c.MustGet("name").(string) // 从上下文取值
        log.Println(name)
        c.JSON(http.StatusOK, gin.H{
            "message": "Hello world!",
        })
    })
```

为路由组注册中间件

为路由组注册中间件有以下两种写法。

写法1：

```
shopGroup := r.Group("/shop", StatCost())
{
    shopGroup.GET("/index", func(c *gin.Context) {...})
    ...
}
```

写法2：

```
shopGroup := r.Group("/shop")
shopGroup.Use(StatCost())
{
    shopGroup.GET("/index", func(c *gin.Context) {...})
    ...
}
```

中间件注意事项

gin默认中间件

`gin.Default()`默认使用了`Logger`和`Recovery`中间件，其中：

- `Logger`中间件将日志写入`gin.DefaultWriter`，即使配置了`GIN_MODE=release`。
- `Recovery`中间件会recover任何`panic`。如果有panic的话，会写入500响应码。

如果不想使用上面两个默认的中间件，可以使用`gin.New()`新建一个没有任何默认中间件的路由。

gin中间件中使用goroutine

当在中间件或`handler`中启动新的`goroutine`时，**不能使用**原始的上下文（c *gin.Context），必须使用其只读副本（`c.Copy()`）。防止c被修改

运行多个服务

我们可以在多个端口启动服务，例如：

```
package main

import (
    "log"
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "golang.org/x/sync/errgroup"
)

var (
    g errgroup.Group
)

func router01() http.Handler {
    e := gin.New()
    e.Use(gin.Recovery())
    e.GET("/", func(c *gin.Context) {
        c.JSON(
            http.StatusOK,
            gin.H{
                "code":  http.StatusOK,
                "error": "Welcome server 01",
```

```
            },
        )
    })

    return e
}

func router02() http.Handler {
    e := gin.New()
    e.Use(gin.Recovery())
    e.GET("/", func(c *gin.Context) {
        c.JSON(
            http.StatusOK,
            gin.H{
                "code":  http.StatusOK,
                "error": "Welcome server 02",
            },
        )
    })

    return e
}

func main() {
    server01 := &http.Server{
        Addr:         ":8080",
        Handler:      router01(),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    server02 := &http.Server{
        Addr:         ":8081",
        Handler:      router02(),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }
    // 借助errgroup.Group或者自行开启两个goroutine分别启动两个服务
    g.Go(func() error {
        return server01.ListenAndServe()
    })

    g.Go(func() error {
```

```
        return server02.ListenAndServe()
    })

    if err := g.Wait(); err != nil {
        log.Fatal(err)
    }
}
```

GORM入门指南

gorm是一个使用Go语言编写的ORM框架。它文档齐全，对开发者友好，支持主流数据库。

注意：本文发布于2020-02-11，文中使用的gorm版本有极大概率与你正在使用的不一致，为了更好的使用gorm请移步官方中文文档：https://gorm.io/zh_CN/docs/

gorm介绍

Github GORM

中文官方网站内含十分齐全的中文文档，有了它你甚至不需要再继续向下阅读本文。

安装

```
go get -u github.com/jinzhu/gorm
```

连接数据库

连接不同的数据库都需要导入对应数据的驱动程序，GORM已经贴心的为我们包装了一些驱动程序，只需要按如下方式导入需要的数据库驱动即可：

```
import _ "github.com/jinzhu/gorm/dialects/mysql"
// import _ "github.com/jinzhu/gorm/dialects/postgres"
// import _ "github.com/jinzhu/gorm/dialects/sqlite"
// import _ "github.com/jinzhu/gorm/dialects/mssql"
```

连接MySQL

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/mysql"
)

func main() {
  db, err := gorm.Open("mysql",
"user:password@(localhost)/dbname?
charset=utf8mb4&parseTime=True&loc=Local")
  defer db.Close()
}
```

连接PostgreSQL

基本代码同上，注意引入对应`postgres`驱动并正确指定`gorm.Open()`参数。

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/postgres"
)

func main() {
  db, err := gorm.Open("postgres", "host=myhost port=myport
user=gorm dbname=gorm password=mypassword")
  defer db.Close()
}
```

连接Sqlite3

基本代码同上，注意引入对应`sqlite`驱动并正确指定`gorm.Open()`参数。

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/sqlite"
)

func main() {
  db, err := gorm.Open("sqlite3", "/tmp/gorm.db")
  defer db.Close()
}
```

连接SQL Server

基本代码同上，注意引入对应`mssql`驱动并正确指定`gorm.Open()`参数。

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/mssql"
)

func main() {
  db, err := gorm.Open("mssql",
"sqlserver://username:password@localhost:1433?database=dbname")
  defer db.Close()
}
```

GORM基本示例

注意：

1. 本文以MySQL数据库为例，讲解GORM各项功能的主要使用方法。
2. 往下阅读本文前，你需要有一个能够成功连接上的MySQL数据库实例。

Docker快速创建MySQL实例

很多同学如果不会安装MySQL或者懒得安装MySQL，可以使用一下命令快速运行一个MySQL8.0.19实例，当然前提是你要有docker环境…

在本地的`13306`端口运行一个名为`mysql8019`，root用户名密码为`root1234`的MySQL容器环境：

```
docker run --name mysql8019 -p 13306:3306 -e
MYSQL_ROOT_PASSWORD=root1234 -d mysql:8.0.19
```

在另外启动一个`MySQL Client`连接上面的MySQL环境，密码为上一步指定的密码`root1234`：

```
docker run -it --network host --rm mysql mysql -h127.0.0.1 -
P13306 --default-character-set=utf8mb4 -uroot -p
```

创建数据库

在使用GORM前手动创建数据库`db1`：

```
CREATE DATABASE db1;
```

GORM操作MySQL

使用GORM连接上面的db1进行创建、查询、更新、删除操作。

```
package main

import (
    "fmt"
    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/mysql"
)

// UserInfo 用户信息
type UserInfo struct {
    ID uint
    Name string
    Gender string
    Hobby string
}


func main() {
    db, err := gorm.Open("mysql",
"root:root1234@(127.0.0.1:13306)/db1?
charset=utf8mb4&parseTime=True&loc=Local")
    if err≠ nil{
        panic(err)
    }
    defer db.Close()

    // 自动迁移
    db.AutoMigrate(&UserInfo{})

    u1 := UserInfo{1, "七米", "男", "篮球"}
    u2 := UserInfo{2, "沙河娜扎", "女", "足球"}
    // 创建记录
    db.Create(&u1)
    db.Create(&u2)
    // 查询
    var u = new(UserInfo)
    db.First(u)
    fmt.Printf("%#v\n", u)
```

```
    var uu UserInfo
    db.Find(&uu, "hobby=?", "足球")
    fmt.Printf("%#v\n", uu)

    // 更新
    db.Model(&u).Update("hobby", "双色球")
    // 删除
    db.Delete(&u)
}
```

GORM Model定义

在使用ORM工具时，通常我们需要在代码中定义模型（Models）与数据库中的数据表进行映射，在GORM中模型（Models）通常是正常定义的结构体、基本的go类型或它们的指针。 同时也支持`sql.Scanner`及`driver.Valuer`接口（interfaces）。

gorm.Model

为了方便模型定义，GORM内置了一个`gorm.Model`结构体。`gorm.Model`是一个包含了`ID`， `CreatedAt`， `UpdatedAt`， `DeletedAt`四个字段的Golang结构体。

```
// gorm.Model 定义
type Model struct {
  ID        uint `gorm:"primary_key"`
  CreatedAt time.Time
  UpdatedAt time.Time
  DeletedAt *time.Time
}
```

你可以将它嵌入到你自己的模型中：

```
// 将 `ID`, `CreatedAt`, `UpdatedAt`, `DeletedAt`字段注入到`User`模型中
type User struct {
  gorm.Model
  Name string
}
```

当然你也可以完全自己定义模型：

```
// 不使用gorm.Model，自行定义模型
type User struct {
  ID   int
  Name string
}
```

模型定义示例

```
type User struct {
  gorm.Model
  Name         string
  Age          sql.NullInt64
  Birthday     *time.Time
  Email        string  `gorm:"type:varchar(100);unique_index"`
  Role         string  `gorm:"size:255"` // 设置字段大小为255
  MemberNumber *string `gorm:"unique;not null"` // 设置会员号
(member number) 唯一并且不为空
  Num          int     `gorm:"AUTO_INCREMENT"` // 设置 num 为自增
类型
  Address      string  `gorm:"index:addr"` // 给address字段创建名
为addr的索引
  IgnoreMe     int     `gorm:"-"` // 忽略本字段
}
```

结构体标记（tags）

使用结构体声明模型时，标记（tags）是可选项。gorm支持以下标记：

支持的结构体标记（Struct tags）

结构体标记（Tag）	描述
Column	指定列名
Type	指定列数据类型
Size	指定列大小，默认值255
PRIMARY_KEY	将列指定为主键
UNIQUE	将列指定为唯一
DEFAULT	指定列默认值
PRECISION	指定列精度
NOT NULL	将列指定为非 NULL
AUTO_INCREMENT	指定列是否为自增类型
INDEX	创建具有或不带名称的索引，如果多个索引同名则创建复合索引
UNIQUE_INDEX	和 INDEX 类似，只不过创建的是唯一索引
EMBEDDED	将结构设置为嵌入
EMBEDDED_PREFIX	设置嵌入结构的前缀
-	忽略此字段

关联相关标记（tags）

结构体标记（Tag）	描述
MANY2MANY	指定连接表
FOREIGNKEY	设置外键
ASSOCIATION_FOREIGNKEY	设置关联外键
POLYMORPHIC	指定多态类型
POLYMORPHIC_VALUE	指定多态值
JOINTABLE_FOREIGNKEY	指定连接表的外键
ASSOCIATION_JOINTABLE_FOREIGNKEY	指定连接表的关联外键
SAVE_ASSOCIATIONS	是否自动完成 save 的相关操作
ASSOCIATION_AUTOUPDATE	是否自动完成 update 的相关操作
ASSOCIATION_AUTOCREATE	是否自动完成 create 的相关操作
ASSOCIATION_SAVE_REFERENCE	是否自动完成引用的 save 的相关操作
PRELOAD	是否自动完成预加载的相关操作

主键、表名、列名的约定

主键 (Primary Key)

GORM 默认会使用名为ID的字段作为表的主键。

```
type User struct {
  ID   string // 名为`ID`的字段会默认作为表的主键
  Name string
}

// 使用`AnimalID`作为主键
type Animal struct {
  AnimalID int64 `gorm:"primary_key"`
  Name     string
  Age      int64
}
```

表名 (Table Name)

表名默认就是结构体名称的复数，例如：

```
type User struct {} // 默认表名是 `users`
```

```
// 将 User 的表名设置为 `profiles`
func (User) TableName() string {
  return "profiles"
}

func (u User) TableName() string {
  if u.Role == "admin" {
    return "admin_users"
  } else {
    return "users"
  }
}

// 禁用默认表名的复数形式，如果置为 true，则 `User` 的默认表名是 `user`
db.SingularTable(true)
```

也可以通过`Table()`指定表名：

```
// 使用User结构体创建名为`deleted_users`的表
db.Table("deleted_users").CreateTable(&User{})

var deleted_users []User
db.Table("deleted_users").Find(&deleted_users)
//// SELECT * FROM deleted_users;

db.Table("deleted_users").Where("name = ?", "jinzhu").Delete()
//// DELETE FROM deleted_users WHERE name = 'jinzhu';
```

GORM还支持更改默认表名称规则：

```
gorm.DefaultTableNameHandler = func (db *gorm.DB,
defaultTableName string) string  {
  return "prefix_" + defaultTableName;
}
```

列名（Column Name）

列名由字段名称进行下划线分割来生成

```
type User struct {
  ID        uint      // column name is `id`
  Name      string    // column name is `name`
  Birthday  time.Time // column name is `birthday`
  CreatedAt time.Time // column name is `created_at`
}
```

可以使用结构体tag指定列名：

```
type Animal struct {
  AnimalId    int64     `gorm:"column:beast_id"`         // set
column name to `beast_id`
  Birthday    time.Time `gorm:"column:day_of_the_beast"` // set
column name to `day_of_the_beast`
  Age         int64     `gorm:"column:age_of_the_beast"` // set
column name to `age_of_the_beast`
}
```

时间戳跟踪

CreatedAt

如果模型有 `CreatedAt`字段，该字段的值将会是初次创建记录的时间。

```
db.Create(&user) // `CreatedAt`将会是当前时间

// 可以使用`Update`方法来改变`CreateAt`的值
db.Model(&user).Update("CreatedAt", time.Now())
```

UpdatedAt

如果模型有`UpdatedAt`字段，该字段的值将会是每次更新记录的时间。

```
db.Save(&user) // `UpdatedAt`将会是当前时间

db.Model(&user).Update("name", "jinzhu") // `UpdatedAt`将会是当前
时间
```

DeletedAt

如果模型有`DeletedAt`字段，调用`Delete`删除该记录时，将会设置`DeletedAt`字段为当前时间，而不是直接将记录从数据库中删除。

GORM CRUD指南

注意：本文发布于2020-02-16，文中使用的gorm版本有极大概率与你正在使用的不一致，为了更好的使用gorm请移步官方中文文档：https://gorm.io/zh_CN/docs/

CRUD通常指数据库的增删改查操作，本文详细介绍了如何使用GORM实现创建、查询、更新和删除操作。

本文中的`db`变量为`*gorm.DB`对象，例如：

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/mysql"
)

func main() {
  db, err := gorm.Open("mysql", "user:password@/dbname?
charset=utf8&parseTime=True&loc=Local")
  defer db.Close()

  // db.Xx
}
```

创建

创建记录

首先定义模型：

```
type User struct {
    ID           int64
    Name         string
    Age          int64
}
```

使用使用`NewRecord()`查询主键是否存在，主键为空使用`Create()`创建记录：

```
user := User{Name: "q1mi", Age: 18}

db.NewRecord(user) // 主键为空返回`true`
db.Create(&user)   // 创建user
db.NewRecord(user) // 创建`user`后返回`false`
```

默认值

可以通过 `tag` 定义字段的默认值，比如：

```
type User struct {
  ID   int64
  Name string `gorm:"default:'小王子'"`
  Age  int64
}
```

注意：通过tag定义字段的默认值，在创建记录时候生成的 SQL 语句会排除没有值或值为 零值 的字段。 在将记录插入到数据库后，Gorm会从数据库加载那些字段的默认值。

举个例子：

```
var user = User{Name: "", Age: 99}
db.Create(&user)
```

上面代码实际执行的SQL语句是`INSERT INTO users("age") values('99');`，排除了零值字段`Name`，而在数据库中这一条数据会使用设置的默认值`小王子`作为Name字段的值。

注意：所有字段的零值，比如`0`， `""`，`false`或者其它`零值`，都不会保存到数据库内，但会使用他们的默认值。 如果你想避免这种情况，可以考虑使用指针或实现`Scanner/Valuer`接口，比如：

使用指针方式实现零值存入数据库

```
// 使用指针
type User struct {
  ID   int64
  Name *string `gorm:"default:'小王子'"`
  Age  int64
}
user := User{Name: new(string), Age: 18)}
db.Create(&user)  // 此时数据库中该条记录name字段的值就是''
```

使用Scanner/Valuer接口方式实现零值存入数据库

```go
// 使用 Scanner/Valuer
type User struct {
    ID int64
    Name sql.NullString `gorm:"default:'小王子'"` //
sql.NullString 实现了Scanner/Valuer接口
    Age  int64
}
user := User{Name: sql.NullString{"", true}, Age:18}
db.Create(&user)  // 此时数据库中该条记录name字段的值就是''
```

扩展创建选项

例如PostgreSQL数据库中可以使用下面的方式实现合并插入， 有则更新， 无则插入。

```go
// 为Instert语句添加扩展SQL选项
db.Set("gorm:insert_option", "ON CONFLICT").Create(&product)
// INSERT INTO products (name, code) VALUES ("name", "code") ON
CONFLICT;
```

查询

一般查询

```go
// 根据主键查询第一条记录
db.First(&user)
//// SELECT * FROM users ORDER BY id LIMIT 1;

// 随机获取一条记录
db.Take(&user)
//// SELECT * FROM users LIMIT 1;

// 根据主键查询最后一条记录
db.Last(&user)
//// SELECT * FROM users ORDER BY id DESC LIMIT 1;

// 查询所有的记录
db.Find(&users)
//// SELECT * FROM users;

// 查询指定的某条记录(仅当主键为整型时可用)
db.First(&user, 10)
//// SELECT * FROM users WHERE id = 10;
```

Where 条件

普通SQL查询

```
// Get first matched record
db.Where("name = ?", "jinzhu").First(&user)
//// SELECT * FROM users WHERE name = 'jinzhu' limit 1;

// Get all matched records
db.Where("name = ?", "jinzhu").Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu';

// <>
db.Where("name <> ?", "jinzhu").Find(&users)
//// SELECT * FROM users WHERE name <> 'jinzhu';

// IN
db.Where("name IN (?)", []string{"jinzhu", "jinzhu
2"}).Find(&users)
//// SELECT * FROM users WHERE name in ('jinzhu','jinzhu 2');

// LIKE
db.Where("name LIKE ?", "%jin%").Find(&users)
//// SELECT * FROM users WHERE name LIKE '%jin%';

// AND
db.Where("name = ? AND age ⩾ ?", "jinzhu", "22").Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu' AND age ⩾ 22;

// Time
db.Where("updated_at > ?", lastWeek).Find(&users)
//// SELECT * FROM users WHERE updated_at > '2000-01-01
00:00:00';

// BETWEEN
db.Where("created_at BETWEEN ? AND ?", lastWeek,
today).Find(&users)
//// SELECT * FROM users WHERE created_at BETWEEN '2000-01-01
00:00:00' AND '2000-01-08 00:00:00';
```

Struct & Map查询

```
// Struct
db.Where(&User{Name: "jinzhu", Age: 20}).First(&user)
//// SELECT * FROM users WHERE name = "jinzhu" AND age = 20
LIMIT 1;

// Map
db.Where(map[string]interface{}{"name": "jinzhu", "age":
20}).Find(&users)
//// SELECT * FROM users WHERE name = "jinzhu" AND age = 20;

// 主键的切片
db.Where([]int64{20, 21, 22}).Find(&users)
//// SELECT * FROM users WHERE id IN (20, 21, 22);
```

提示：当通过结构体进行查询时，GORM将会只通过非零值字段查询，这意味着如果你的字段值为`0`，`''`，`false`或者其他零值时，将不会被用于构建查询条件，例如：

```
db.Where(&User{Name: "jinzhu", Age: 0}).Find(&users)
//// SELECT * FROM users WHERE name = "jinzhu";
```

你可以使用指针或实现 Scanner/Valuer 接口来避免这个问题.

```
// 使用指针
type User struct {
  gorm.Model
  Name string
  Age  *int
}

// 使用 Scanner/Valuer
type User struct {
  gorm.Model
  Name string
  Age  sql.NullInt64  // sql.NullInt64 实现了 Scanner/Valuer 接口
}
```

Not 条件

作用与 Where 类似的情形如下：

```
db.Not("name", "jinzhu").First(&user)
//// SELECT * FROM users WHERE name <> "jinzhu" LIMIT 1;
```

```
// Not In
db.Not("name", []string{"jinzhu", "jinzhu 2"}).Find(&users)
//// SELECT * FROM users WHERE name NOT IN ("jinzhu", "jinzhu
2");

// Not In slice of primary keys
db.Not([]int64{1,2,3}).First(&user)
//// SELECT * FROM users WHERE id NOT IN (1,2,3);

db.Not([]int64{}).First(&user)
//// SELECT * FROM users;

// Plain SQL
db.Not("name = ?", "jinzhu").First(&user)
//// SELECT * FROM users WHERE NOT(name = "jinzhu");

// Struct
db.Not(User{Name: "jinzhu"}).First(&user)
//// SELECT * FROM users WHERE name <> "jinzhu";
```

Or条件

```
db.Where("role = ?", "admin").Or("role = ?",
"super_admin").Find(&users)
//// SELECT * FROM users WHERE role = 'admin' OR role =
'super_admin';

// Struct
db.Where("name = 'jinzhu'").Or(User{Name: "jinzhu
2"}).Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu' OR name =
'jinzhu 2';

// Map
db.Where("name = 'jinzhu'").Or(map[string]interface{}{"name":
"jinzhu 2"}).Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu' OR name =
'jinzhu 2';
```

内联条件

作用与Where查询类似，当内联条件与多个立即执行方法一起使用时，内联条件不会传递给后面的立即执行方法。

```
// 根据主键获取记录（只适用于整形主键）
db.First(&user, 23)
//// SELECT * FROM users WHERE id = 23 LIMIT 1;
// 根据主键获取记录，如果它是一个非整形主键
db.First(&user, "id = ?", "string_primary_key")
//// SELECT * FROM users WHERE id = 'string_primary_key' LIMIT
1;

// Plain SQL
db.Find(&user, "name = ?", "jinzhu")
//// SELECT * FROM users WHERE name = "jinzhu";

db.Find(&users, "name <> ? AND age > ?", "jinzhu", 20)
//// SELECT * FROM users WHERE name <> "jinzhu" AND age > 20;

// Struct
db.Find(&users, User{Age: 20})
//// SELECT * FROM users WHERE age = 20;

// Map
db.Find(&users, map[string]interface{}{"age": 20})
//// SELECT * FROM users WHERE age = 20;
```

额外查询选项

```
// 为查询 SQL 添加额外的 SQL 操作
db.Set("gorm:query_option", "FOR UPDATE").First(&user, 10)
//// SELECT * FROM users WHERE id = 10 FOR UPDATE;
```

FirstOrInit

获取匹配的第一条记录，否则根据给定的条件初始化一个新的对象（仅支持 struct 和 map 条件）

```
// 未找到
db.FirstOrInit(&user, User{Name: "non_existing"})
//// user -> User{Name: "non_existing"}

// 找到
db.Where(User{Name: "Jinzhu"}).FirstOrInit(&user)
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
db.FirstOrInit(&user, map[string]interface{}{"name": "jinzhu"})
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
```

Attrs

如果记录未找到，将使用参数初始化 struct.

```
// 未找到
db.Where(User{Name: "non_existing"}).Attrs(User{Age:
20}).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = 'non_existing';
//// user -> User{Name: "non_existing", Age: 20}

db.Where(User{Name: "non_existing"}).Attrs("age",
20).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = 'non_existing';
//// user -> User{Name: "non_existing", Age: 20}

// 找到
db.Where(User{Name: "Jinzhu"}).Attrs(User{Age:
30}).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = jinzhu';
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
```

Assign

不管记录是否找到，都将参数赋值给 struct.

```
// 未找到
db.Where(User{Name: "non_existing"}).Assign(User{Age:
20}).FirstOrInit(&user)
//// user → User{Name: "non_existing", Age: 20}

// 找到
db.Where(User{Name: "Jinzhu"}).Assign(User{Age:
30}).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = jinzhu';
//// user → User{Id: 111, Name: "Jinzhu", Age: 30}
```

FirstOrCreate

获取匹配的第一条记录，否则根据给定的条件创建一个新的记录（仅支持 struct 和 map 条件）

```
// 未找到
db.FirstOrCreate(&user, User{Name: "non_existing"})
//// INSERT INTO "users" (name) VALUES ("non_existing");
//// user → User{Id: 112, Name: "non_existing"}

// 找到
db.Where(User{Name: "Jinzhu"}).FirstOrCreate(&user)
//// user → User{Id: 111, Name: "Jinzhu"}
```

Attrs

如果记录未找到，将使用参数创建 struct 和记录.

```
 // 未找到
db.Where(User{Name: "non_existing"}).Attrs(User{Age:
20}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'non_existing';
//// INSERT INTO "users" (name, age) VALUES ("non_existing",
20);
//// user → User{Id: 112, Name: "non_existing", Age: 20}

// 找到
db.Where(User{Name: "jinzhu"}).Attrs(User{Age:
30}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'jinzhu';
//// user → User{Id: 111, Name: "jinzhu", Age: 20}
```

Assign

不管记录是否找到，都将参数赋值给 `struct` 并保存至数据库.

```
// 未找到
db.Where(User{Name: "non_existing"}).Assign(User{Age:
20}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'non_existing';
//// INSERT INTO "users" (name, age) VALUES ("non_existing",
20);
//// user → User{Id: 112, Name: "non_existing", Age: 20}

// 找到
db.Where(User{Name: "jinzhu"}).Assign(User{Age:
30}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'jinzhu';
//// UPDATE users SET age=30 WHERE id = 111;
//// user → User{Id: 111, Name: "jinzhu", Age: 30}
```

高级查询

子查询

基于 `*gorm.expr` 的子查询

```
db.Where("amount > ?",
db.Table("orders").Select("AVG(amount)").Where("state = ?",
"paid").SubQuery()).Find(&orders)
// SELECT * FROM "orders"  WHERE "orders"."deleted_at" IS NULL
AND (amount > (SELECT AVG(amount) FROM "orders"  WHERE (state =
'paid')));
```

选择字段

`Select`，指定你想从数据库中检索出的字段，默认会选择全部字段。

```
db.Select("name, age").Find(&users)
//// SELECT name, age FROM users;

db.Select([]string{"name", "age"}).Find(&users)
//// SELECT name, age FROM users;

db.Table("users").Select("COALESCE(age,?)", 42).Rows()
//// SELECT COALESCE(age,'42') FROM users;
```

排序

Order，指定从数据库中检索出记录的顺序。设置第二个参数 reorder 为 true ，可以覆盖前面定义的排序条件。

```
db.Order("age desc, name").Find(&users)
//// SELECT * FROM users ORDER BY age desc, name;

// 多字段排序
db.Order("age desc").Order("name").Find(&users)
//// SELECT * FROM users ORDER BY age desc, name;

// 覆盖排序
db.Order("age desc").Find(&users1).Order("age",
true).Find(&users2)
//// SELECT * FROM users ORDER BY age desc; (users1)
//// SELECT * FROM users ORDER BY age; (users2)
```

数量

Limit，指定从数据库检索出的最大记录数。

```
db.Limit(3).Find(&users)
//// SELECT * FROM users LIMIT 3;

// -1 取消 Limit 条件
db.Limit(10).Find(&users1).Limit(-1).Find(&users2)
//// SELECT * FROM users LIMIT 10; (users1)
//// SELECT * FROM users; (users2)
```

偏移

Offset，指定开始返回记录前要跳过的记录数。

```
db.Offset(3).Find(&users)
//// SELECT * FROM users OFFSET 3;

// -1 取消 Offset 条件
db.Offset(10).Find(&users1).Offset(-1).Find(&users2)
//// SELECT * FROM users OFFSET 10; (users1)
//// SELECT * FROM users; (users2)
```

总数

Count, 该 model 能获取的记录总数。

```
db.Where("name = ?", "jinzhu").Or("name = ?", "jinzhu
2").Find(&users).Count(&count)
//// SELECT * from USERS WHERE name = 'jinzhu' OR name =
'jinzhu 2'; (users)
//// SELECT count(*) FROM users WHERE name = 'jinzhu' OR name =
'jinzhu 2'; (count)

db.Model(&User{}).Where("name = ?", "jinzhu").Count(&count)
//// SELECT count(*) FROM users WHERE name = 'jinzhu'; (count)

db.Table("deleted_users").Count(&count)
//// SELECT count(*) FROM deleted_users;

db.Table("deleted_users").Select("count(distinct(name))").Count
(&count)
//// SELECT count( distinct(name) ) FROM deleted_users; (count)
```

注意 Count 必须是链式查询的最后一个操作 ，因为它会覆盖前面的 SELECT，但如果里面使用了 count 时不会覆盖

Group & Having

```
rows, err := db.Table("orders").Select("date(created_at) as
date, sum(amount) as total").Group("date(created_at)").Rows()
for rows.Next() {
  ...
}

// 使用Scan将多条结果扫描进事先准备好的结构体切片中
type Result struct {
    Date time.Time
    Total int
}
var rets []Result
db.Table("users").Select("date(created_at) as date, sum(age) as
total").Group("date(created_at)").Scan(&rets)
```

```
rows, err := db.Table("orders").Select("date(created_at) as
date, sum(amount) as
total").Group("date(created_at)").Having("sum(amount) > ?",
100).Rows()
for rows.Next() {
  ...
}

type Result struct {
  Date  time.Time
  Total int64
}
db.Table("orders").Select("date(created_at) as date,
sum(amount) as
total").Group("date(created_at)").Having("sum(amount) > ?",
100).Scan(&results)
```

连接

Joins，指定连接条件

```
rows, err := db.Table("users").Select("users.name,
emails.email").Joins("left join emails on emails.user_id =
users.id").Rows()
for rows.Next() {
  ...
}

db.Table("users").Select("users.name,
emails.email").Joins("left join emails on emails.user_id =
users.id").Scan(&results)

// 多连接及参数
db.Joins("JOIN emails ON emails.user_id = users.id AND
emails.email = ?", "jinzhu@example.org").Joins("JOIN
credit_cards ON credit_cards.user_id =
users.id").Where("credit_cards.number = ?",
"411111111111").Find(&user)
```

Pluck

Pluck，查询 model 中的一个列作为切片，如果您想要查询多个列，您应该使用 Scan

```
var ages []int64
db.Find(&users).Pluck("age", &ages)

var names []string
db.Model(&User{}).Pluck("name", &names)

db.Table("deleted_users").Pluck("name", &names)

// 想查询多个字段？ 这样做：
db.Select("name, age").Find(&users)
```

扫描

Scan，扫描结果至一个 struct.

```
type Result struct {
  Name string
  Age  int
}

var result Result
db.Table("users").Select("name, age").Where("name = ?",
"Antonio").Scan(&result)

var results []Result
db.Table("users").Select("name, age").Where("id > ?",
0).Scan(&results)

// 原生 SQL
db.Raw("SELECT name, age FROM users WHERE name = ?",
"Antonio").Scan(&result)
```

链式操作相关

链式操作

Method Chaining，Gorm 实现了链式操作接口，所以你可以把代码写成这样：

```
// 创建一个查询
tx := db.Where("name = ?", "jinzhu")

// 添加更多条件
if someCondition {
  tx = tx.Where("age = ?", 20)
} else {
  tx = tx.Where("age = ?", 30)
}

if yetAnotherCondition {
  tx = tx.Where("active = ?", 1)
}
```

在调用立即执行方法前不会生成Query语句，借助这个特性你可以创建一个函数来处理一些通用逻辑。

立即执行方法

Immediate methods ，立即执行方法是指那些会立即生成SQL语句并发送到数据库的方法，他们一般是CRUD方法，比如：

Create， First， Find， Take， Save， UpdateXXX， Delete， Scan， Row， Rows…

这有一个基于上面链式方法代码的立即执行方法的例子：

```
tx.Find(&user)
```

生成的SQL语句如下：

```
SELECT * FROM users where name = 'jinzhu' AND age = 30 AND
active = 1;
```

范围

Scopes，Scope是建立在链式操作的基础之上的。

基于它，你可以抽取一些通用逻辑，写出更多可重用的函数库。

```
func AmountGreaterThan1000(db *gorm.DB) *gorm.DB {
  return db.Where("amount > ?", 1000)
}

func PaidWithCreditCard(db *gorm.DB) *gorm.DB {
```

```
  return db.Where("pay_mode_sign = ?", "C")
}

func PaidWithCod(db *gorm.DB) *gorm.DB {
  return db.Where("pay_mode_sign = ?", "C")
}

func OrderStatus(status []string) func (db *gorm.DB) *gorm.DB {
  return func (db *gorm.DB) *gorm.DB {
    return db.Scopes(AmountGreaterThan1000).Where("status IN
(?)", status)
  }
}

db.Scopes(AmountGreaterThan1000,
PaidWithCreditCard).Find(&orders)
// 查找所有金额大于 1000 的信用卡订单

db.Scopes(AmountGreaterThan1000, PaidWithCod).Find(&orders)
// 查找所有金额大于 1000 的 COD 订单

db.Scopes(AmountGreaterThan1000, OrderStatus([]string{"paid",
"shipped"})).Find(&orders)
// 查找所有金额大于 1000 且已付款或者已发货的订单
```

多个立即执行方法

Multiple Immediate Methods，在 GORM 中使用多个立即执行方法时，后一个立即执行方法会复用前一个**立即执行方法**的条件（不包括内联条件）。

```
db.Where("name LIKE ?", "jinzhu%").Find(&users, "id IN (?)",
[]int{1, 2, 3}).Count(&count)
```

生成的 Sql

```
SELECT * FROM users WHERE name LIKE 'jinzhu%' AND id IN (1, 2,
3)

SELECT count(*) FROM users WHERE name LIKE 'jinzhu%'
```

更新

更新所有字段

`Save()`默认会更新该对象的所有字段，即使你没有赋值。

```
db.First(&user)

user.Name = "七米"
user.Age = 99
db.Save(&user)

////  UPDATE `users` SET `created_at` = '2020-02-16 12:52:20',
`updated_at` = '2020-02-16 12:54:55', `deleted_at` = NULL,
`name` = '七米', `age` = 99, `active` = true  WHERE
`users`.`deleted_at` IS NULL AND `users`.`id` = 1
```

更新修改字段

如果你只希望更新指定字段，可以使用`Update`或者`Updates`

```
// 更新单个属性，如果它有变化
db.Model(&user).Update("name", "hello")
//// UPDATE users SET name='hello', updated_at='2013-11-17
21:34:10' WHERE id=111;

// 根据给定的条件更新单个属性
db.Model(&user).Where("active = ?", true).Update("name",
"hello")
//// UPDATE users SET name='hello', updated_at='2013-11-17
21:34:10' WHERE id=111 AND active=true;

// 使用 map 更新多个属性，只会更新其中有变化的属性
db.Model(&user).Updates(map[string]interface{}{"name": "hello",
"age": 18, "active": false})
//// UPDATE users SET name='hello', age=18, active=false,
updated_at='2013-11-17 21:34:10' WHERE id=111;

// 使用 struct 更新多个属性，只会更新其中有变化且为非零值的字段
db.Model(&user).Updates(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18, updated_at = '2013-
11-17 21:34:10' WHERE id = 111;

// 警告：当使用 struct 更新时，GORM只会更新那些非零值的字段
```

```
// 对于下面的操作，不会发生任何更新，"", 0, false 都是其类型的零值
db.Model(&user).Updates(User{Name: "", Age: 0, Active: false})
```

更新选定字段

如果你想更新或忽略某些字段，你可以使用 `Select`，`Omit`

```
db.Model(&user).Select("name").Updates(map[string]interface{}
{"name": "hello", "age": 18, "active": false})
//// UPDATE users SET name='hello', updated_at='2013-11-17
21:34:10' WHERE id=111;

db.Model(&user).Omit("name").Updates(map[string]interface{}
{"name": "hello", "age": 18, "active": false})
//// UPDATE users SET age=18, active=false, updated_at='2013-
11-17 21:34:10' WHERE id=111;
```

无Hooks更新

上面的更新操作会自动运行 model 的 `BeforeUpdate`，`AfterUpdate` 方法，更新 `UpdatedAt` 时间戳，在更新时保存其 `Associations`，如果你不想调用这些方法，你可以使用 `UpdateColumn`， `UpdateColumns`

```
// 更新单个属性，类似于 `Update`
db.Model(&user).UpdateColumn("name", "hello")
//// UPDATE users SET name='hello' WHERE id = 111;

// 更新多个属性，类似于 `Updates`
db.Model(&user).UpdateColumns(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18 WHERE id = 111;
```

批量更新

批量更新时Hooks（钩子函数）不会运行。

```
db.Table("users").Where("id IN (?)", []int{10,
11}).Updates(map[string]interface{}{"name": "hello", "age":
18})
//// UPDATE users SET name='hello', age=18 WHERE id IN (10,
11);

// 使用 struct 更新时，只会更新非零值字段，若想更新所有字段，请使用
map[string]interface{}
db.Model(User{}).Updates(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18;

// 使用 `RowsAffected` 获取更新记录总数
db.Model(User{}).Updates(User{Name: "hello", Age:
18}).RowsAffected
```

使用SQL表达式更新

先查询表中的第一条数据保存至user变量。

```
var user User
db.First(&user)
db.Model(&user).Update("age", gorm.Expr("age * ? + ?", 2, 100))
//// UPDATE `users` SET `age` = age * 2 + 100, `updated_at` =
'2020-02-16 13:10:20'  WHERE `users`.`id` = 1;

db.Model(&user).Updates(map[string]interface{}{"age":
gorm.Expr("age * ? + ?", 2, 100)})
//// UPDATE "users" SET "age" = age * '2' + '100', "updated_at"
= '2020-02-16 13:05:51' WHERE `users`.`id` = 1;

db.Model(&user).UpdateColumn("age", gorm.Expr("age - ?", 1))
//// UPDATE "users" SET "age" = age - 1 WHERE "id" = '1';

db.Model(&user).Where("age > 10").UpdateColumn("age",
gorm.Expr("age - ?", 1))
//// UPDATE "users" SET "age" = age - 1 WHERE "id" = '1' AND
quantity > 10;
```

修改Hooks中的值

如果你想修改 `BeforeUpdate`, `BeforeSave` 等 Hooks 中更新的值，你可以使用 `scope.SetColumn`, 例如:

```
func (user *User) BeforeSave(scope *gorm.Scope) (err error) {
  if pw, err := bcrypt.GenerateFromPassword(user.Password, 0);
err == nil {
    scope.SetColumn("EncryptedPassword", pw)
  }
}
```

其它更新选项

```
// 为 update SQL 添加其它的 SQL
db.Model(&user).Set("gorm:update_option", "OPTION (OPTIMIZE FOR
UNKNOWN)").Update("name", "hello")
//// UPDATE users SET name='hello', updated_at = '2013-11-17
21:34:10' WHERE id=111 OPTION (OPTIMIZE FOR UNKNOWN);
```

删除

删除记录

警告 删除记录时，请确保主键字段有值，GORM 会通过主键去删除记录，如果主键为空，GORM 会删除该 model 的所有记录。

```
// 删除现有记录
db.Delete(&email)
//// DELETE from emails where id=10;

// 为删除 SQL 添加额外的 SQL 操作
db.Set("gorm:delete_option", "OPTION (OPTIMIZE FOR
UNKNOWN)").Delete(&email)
//// DELETE from emails where id=10 OPTION (OPTIMIZE FOR
UNKNOWN);
```

批量删除

删除全部匹配的记录

```
db.Where("email LIKE ?", "%jinzhu%").Delete(Email{})
//// DELETE from emails where email LIKE "%jinzhu%";

db.Delete(Email{}, "email LIKE ?", "%jinzhu%")
//// DELETE from emails where email LIKE "%jinzhu%";
```

软删除

如果一个 model 有 DeletedAt 字段，他将自动获得软删除的功能！ 当调用 Delete 方法时， 记录不会真正的从数据库中被删除， 只会将DeletedAt 字段的值会被设置为当前时间

```
db.Delete(&user)
//// UPDATE users SET deleted_at="2013-10-29 10:23" WHERE id = 111;

// 批量删除
db.Where("age = ?", 20).Delete(&User{})
//// UPDATE users SET deleted_at="2013-10-29 10:23" WHERE age = 20;

// 查询记录时会忽略被软删除的记录
db.Where("age = 20").Find(&user)
//// SELECT * FROM users WHERE age = 20 AND deleted_at IS NULL;

// Unscoped 方法可以查询被软删除的记录
db.Unscoped().Where("age = 20").Find(&users)
//// SELECT * FROM users WHERE age = 20;
```

物理删除

```
// Unscoped 方法可以物理删除记录
db.Unscoped().Delete(&order)
//// DELETE FROM orders WHERE id=10;
```

GORM Gen使用指南

发布于2023/10/19 ,更新于2023/10/19 19:28:53

| Golang

> 承蒙大家厚爱，我的《Go语言之路》的纸质版图书已经上架京东，有需要的朋友请点击 此链接 购买。

Gen是一个基于GORM的安全ORM框架，其主要通过代码生成方式实现GORM代码封装。使用Gen框架能够自动生成Model结构体和类型安全的CRUD代码，极大提升CRUD效率。

Gen介绍

Gen是由字节跳动无恒实验室与GORM作者联合研发的一个基于GORM的安全ORM框架，主要通过代码生成方式实现GORM代码封装。

Gen框架在GORM框架的基础上提供了以下能力：

- 基于原始SQL语句生成可重用的CRUD API
- 生成不使用`interface{}`的100%安全的DAO API
- 依据数据库生成遵循GORM约定的结构体Model
- 支持GORM的所有特性

简单来说，使用Gen框架后我们无需手动定义结构体Model，同时Gen框架也能帮我们生成类型安全的CRUD代码。

更多详细介绍请查看Gen官方文档。

此外，Facebook开源的ent也是社区中常用的类似框架，大家可按需选择使用。

如何使用Gen

Gen框架的使用非常简单，如果你熟悉GORM框架，那么你可以通过以下教程快速上手。

安装依赖

```
go get -u gorm.io/gen
```

快速指南

想要在项目中使用Gen框架，通常只需三步。本节将通过一个简单示例快速带大家熟悉Gen框架的使用。

首先，我们假设数据库中已经有一张`book`表，建表语句如下。

```
CREATE TABLE book
(
    `id`     bigint unsigned NOT NULL AUTO_INCREMENT COMMENT
'主键',
    `title`  varchar(128) NOT NULL COMMENT '书籍名称',
    `author` varchar(128) NOT NULL COMMENT '作者',
    `price`  int NOT NULL DEFAULT '0' COMMENT '价格',
    `publish_date` datetime COMMENT '出版日期',
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='书籍表';
```

> 本教程演示的为先有数据表的业务场景，通常这也是比较主流的工程实现流程。

定义Gen配置

配置即代码。我们通常会在项目的`cmd`目录下定义好Gen框架生成代码的配置。例如，我们的项目名称为`gen_demo`，那么我们就在`gen_demo/cmd/gen/generate.go`文件。

```
package main

// gorm gen configure

import (
    "fmt"

    "gorm.io/driver/mysql"
    "gorm.io/gorm"

    "gorm.io/gen"
)

const MySQLDSN = "root:root1234@tcp(127.0.0.1:13306)/db2?
charset=utf8mb4&parseTime=True"

func connectDB(dsn string) *gorm.DB {
    db, err := gorm.Open(mysql.Open(dsn))
    if err != nil {
        panic(fmt.Errorf("connect db fail: %w", err))
    }
    return db
}

func main() {
```

```
    // 指定生成代码的具体相对目录(相对当前文件)，默认为: ./query
    // 默认生成需要使用WithContext之后才可以查询的代码，但可以通过设置gen.WithoutContext禁用该模式
    g := gen.NewGenerator(gen.Config{
        // 默认会在 OutPath 目录生成CRUD代码，并且同目录下生成 model 包
        // 所以OutPath最终package不能设置为model，在有数据库表同步的情况下会产生冲突
        // 若一定要使用可以通过ModelPkgPath单独指定model package的名称
        OutPath: "../../dal/query",
        /* ModelPkgPath: "dal/model"*/

        // gen.WithoutContext：禁用WithContext模式
        // gen.WithDefaultQuery：生成一个全局Query对象Q
        // gen.WithQueryInterface：生成Query接口
        Mode: gen.WithDefaultQuery | gen.WithQueryInterface,
    })

    // 通常复用项目中已有的SQL连接配置db(*gorm.DB)
    // 非必需，但如果需要复用连接时的gorm.Config或需要连接数据库同步表信息则必须设置
    g.UseDB(connectDB(MySQLDSN))

    // 从连接的数据库为所有表生成Model结构体和CRUD代码
    // 也可以手动指定需要生成代码的数据表
    g.ApplyBasic(g.GenerateAllTable()...)

    // 执行并生成代码
    g.Execute()
}
```

> 为什么要放到 `cmd` 目录下？👉 Go官方模块布局说明

生成代码

进入项目下的`cmd/gen`目录下，执行以下命令。

```
go run generate.go
```

上述命令会在项目目录下生成`dal`目录，其中`dal/query`中是CRUD代码，`dal/model`下则是生成Model结构体。

```
├── cmd
│   └── gen
│       └── generate.go
├── dal
│   ├── model
│   │   └── book.gen.go
│   └── query
│       ├── book.gen.go
│       └── gen.go
├── go.mod
├── go.sum
└── main.go
```

我们可以在dal下新建db.go文件，保存如下初始化数据库连接的代码。

```go
package dal

import (
    "fmt"

    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

var DB *gorm.DB

func ConnectDB(dsn string) *gorm.DB {
    db, err := gorm.Open(mysql.Open(dsn))
    if err != nil {
        panic(fmt.Errorf("connect db fail: %w", err))
    }
    return db
}
```

注意：通常不建议直接修改Gen框架生成的代码。

使用生成的代码

Gen会生成基础的查询方法，并且绑定到结构体上，我们可以在项目中使用了它们。

```go
package main

import (
```

```
    "context"
    "fmt"
    "gen_demo/dal"
    "gen_demo/dal/model"
    "gen_demo/dal/query"
    "time"
)

// gen demo

// MySQLDSN MySQL data source name
const MySQLDSN = "root:root1234@tcp(127.0.0.1:13306)/db2?
charset=utf8mb4&parseTime=True"

func init() {
    dal.DB = dal.ConnectDB(MySQLDSN).Debug()
}

func main() {
    // 设置默认DB对象
    query.SetDefault(dal.DB)

    // 创建
    b1 := model.Book{
        Title:       "《七米的Go语言之路》",
        Author:      "七米",
        PublishDate: time.Date(2023, 11, 15, 0, 0, 0, 0,
time.UTC),
        Price:       100,
    }
    err :=
query.Book.WithContext(context.Background()).Create(&b1)
    if err ≠ nil {
        fmt.Printf("create book fail, err:%v\n", err)
        return
    }

    // 更新
    ret, err := query.Book.WithContext(context.Background()).
        Where(query.Book.ID.Eq(1)).
        Update(query.Book.Price, 200)
    if err ≠ nil {
        fmt.Printf("update book fail, err:%v\n", err)
```

```
        return
    }
    fmt.Printf("RowsAffected:%v\n", ret.RowsAffected)

    // 查询
    book, err :=
query.Book.WithContext(context.Background()).First()
    // 也可以使用全局Q对象查询
    //book, err :=
query.Q.Book.WithContext(context.Background()).First()
    if err != nil {
        fmt.Printf("query book fail, err:%v\n", err)
        return
    }
    fmt.Printf("book:%v\n", book)

    // 删除
    ret, err =
query.Book.WithContext(context.Background()).Where(query.Book.I
D.Eq(1)).Delete()
    if err != nil {
        fmt.Printf("delete book fail, err:%v\n", err)
        return
    }
    fmt.Printf("RowsAffected:%v\n", ret.RowsAffected)
}
```

通过上述教程，基本即可掌握Gen框架的基本使用，大家可点击查看Gen官方最佳实践示例代码。

自定义SQL查询

Gen框架使用模板注释的方法支持自定义SQL查询，我们只需要按对应规则将SQL语句注释到interface的方法上即可。Gen将对其进行解析，并为应用的结构生成查询API。

通常建议将自定义查询方法添加到`model`模块下。

注释语法

Gen 为动态条件 SQL 支持提供了一些约定语法，分为三个方面：

- 返回结果
- 模板占位符
- 模板表达式

返回结果

占位符	含义
gen.T	用于返回数据的结构体，会根据生成结构体或者数据库表结构自动生成
gen.M	表示map[string]interface{},用于返回数据
gen.RowsAffected	用于执行SQL进行更新或删除时候,用于返回影响行数
error	返回错误（如果有）

示例

```
// dal/model/querier.go

package model

import "gorm.io/gen"

// 通过添加注释生成自定义方法

type Querier interface {
    // SELECT * FROM @@table WHERE id=@id
    GetByID(id int) (gen.T, error) // 返回结构体和error

    // GetByIDReturnMap 根据ID查询返回map
    //
    // SELECT * FROM @@table WHERE id=@id
    GetByIDReturnMap(id int) (gen.M, error) // 返回 map 和 error

    // SELECT * FROM @@table WHERE author=@author
    GetBooksByAuthor(author string) ([]*gen.T, error) // 返回数据切片和 error
}
```

在Gen配置处（cmd/gen/generate.go）添加自定义方法绑定关系。

```
// 通过ApplyInterface添加为book表添加自定义方法
g.ApplyInterface(func(model.Querier) {},
g.GenerateModel("book"))
```

重新生成代码后，即可使用自定义方法。

```
// 使用自定义的GetBooksByAuthor方法
rets, err :=
query.Book.WithContext(context.Background()).GetBooksByAuthor("
七米")
if err ≠ nil {
    fmt.Printf("GetBooksByAuthor fail, err:%v\n", err)
    return
}
for i, b := range rets {
    fmt.Printf("%d:%v\n", i, b)
}
```

模板占位符

名称	描述
@@table	转义和引用表名
@@<name>	从参数中转义并引用表/列名
@<name>	参数中的SQL查询参数

示例

```
// Filter 自定义Filter接口
type Filter interface {
  // SELECT * FROM @@table WHERE @@column=@value
  FilterWithColumn(column string, value string) (gen.T, error)
}

// 为`Book`添加 `Filter`接口
g.ApplyInterface(func(model.Filter) {},
g.GenerateModel("book"))
```

模板表达式

Gen 为动态条件 SQL 提供了强大的表达式支持，目前支持以下表达式：

- if/else
- where
- set
- for

示例

```
```

```
// Searcher 自定义接口
type Searcher interface {
    // Search 根据指定条件查询书籍
    //
    // SELECT * FROM book
    // WHERE publish_date is not null
    // {{if book ≠ nil}}
    //   {{if book.ID > 0}}
    //     AND id = @book.ID
    //   {{else if book.Author ≠ ""}}
    //     AND author=@book.Author
    //   {{end}}
    // {{end}}
    Search(book *gen.T) ([]*gen.T, error)
}

// 通过ApplyInterface添加为book表添加Searcher接口
g.ApplyInterface(func(model.Searcher) {},
g.GenerateModel("book"))
```

重新生成代码后，即可直接使用自定义的`Search`方法进行查询。

```
b := &model.Book{Author: "Q1mi"}
rets, err =
query.Book.WithContext(context.Background()).Search(b)
if err ≠ nil {
    fmt.Printf("Search fail, err:%v\n", err)
    return
}
for i, b := range rets {
    fmt.Printf("%d:%v\n", i, b)
}
```

数据库到结构体

Gen支持根据GORM约定依据数据库生成结构体，在之前的示例中我们已经使用过类似的代码。

```
// 根据`users`表生成对应结构体`User`
g.GenerateModel("users")

// 基于`users`表生成名为`Employee`的结构体
g.GenerateModelAs("users", "Employee")

// 在生成结构体时还可指定额外的生成选项
// gen.FieldIgnore("address"): 忽略 address 字段
// gen.FieldType("id", "int64"): id字段使用 int64 类型
g.GenerateModel("users", gen.FieldIgnore("address"),
gen.FieldType("id", "int64"))

// 为连接的数据库中的所有表生成对应结构体
g.GenerateAllTable()
```

方法模板

当从数据库生成结构体时，还可以为它们生成事先配置的模板方法，例如：

```
type CommonMethod struct {
    ID   int32
    Name *string
}

func (m *CommonMethod) IsEmpty() bool {
    if m == nil {
        return true
    }
    return m.ID == 0
}

func (m *CommonMethod) GetName() string {
    if m == nil || m.Name == nil {
        return ""
    }
    return *m.Name
}

// 当生成 `People` 结构体时添加 IsEmpty 方法
g.GenerateModel("people",
gen.WithMethod(CommonMethod{}.IsEmpty))

// 生成`User`结构体时添加 `CommonMethod` 的所有方法
```

```
g.GenerateModel("user", gen.WithMethod(CommonMethod{}))
```

最终将生成类下面的代码。

```
// Generated Person struct
type Person struct {
  // ...
}

func (m *Person) IsEmpty() bool {
  if m == nil {
    return true
  }
  return m.ID == 0
}


// Generated User struct
type User struct {
  // ...
}

func (m *User) IsEmpty() bool {
  if m == nil {
    return true
  }
  return m.ID == 0
}

func (m *User) GetName() string {
  if m == nil || m.Name == nil {
    return ""
  }
  return *m.Name
}
```

数据映射

可以自行指定字段类型和数据库列类型之间的数据类型映射。

在某些业务场景下，这个功能非常有用，例如，我们希望将数据库中数字列在生成结构体时都定义为`int64`类型。

```
```

```
var dataMap = map[string]func(gorm.ColumnType) (dataType
string){
  // int mapping
  "int": func(columnType gorm.ColumnType) (dataType string) {
    if n, ok := columnType.Nullable(); ok && n {
      return "*int32"
    }
    return "int32"
  },

  // bool mapping
  "tinyint": func(columnType gorm.ColumnType) (dataType string)
{
    ct, _ := columnType.ColumnType()
    if strings.HasPrefix(ct, "tinyint(1)") {
      return "bool"
    }
    return "byte"
  },
}

g.WithDataTypeMap(dataMap)
```

从 SQL语句生成结构体

Gen 支持遵循 GORM 约定从 sql 生成结构体，具体用法如下。

```
package main

import (
    "gorm.io/gen"
    "gorm.io/gorm"
    "gorm.io/rawsql"
)

func main() {
    g := gen.NewGenerator(gen.Config{
        OutPath: "../query",
        Mode:    gen.WithoutContext | gen.WithDefaultQuery |
gen.WithQueryInterface, // generate mode
    })
    // https://github.com/go-
gorm/rawsql/blob/master/tests/gen_test.go
    gormdb, _ := gorm.Open(rawsql.New(rawsql.Config{
```

```
        //SQL:      rawsql,     // create table sql
        FilePath: []string{
            //"./sql/user.sql", // create table sql file
            "./test_sql", // create table sql file directory
        },
    }))
    g.UseDB(gormdb) // reuse your gorm db

    // Generate basic type-safe DAO API for struct `model.User`
following conventions

    g.ApplyBasic(
        // Generate struct `User` based on table `users`
        g.GenerateModel("users"),

        // Generate struct `Employee` based on table `users`
        g.GenerateModelAs("users", "Employee"),
    )
    g.ApplyBasic(
        // Generate structs from all tables of current database
        g.GenerateAllTable()...,
    )
    // Generate the code
    g.Execute()
}
```

关于Gen框架的更多技巧，推荐查看官方文档。