链滴

非常叼的工单系统，工单结束，工作完成；非常叼的权限管控，精细到页面按钮及 API

作者：lanyulei
原文链接：https://ld246.com/article/1595081770806
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

最近在研究工单系统的时候，被我找到一个非常流弊的工单系统，我们都知道工单系统最麻烦的就是程和模版的维护，并且，在工单处理过程中很可能会添加一些操作，这些操作被称之为钩子。就按我前调研的结果来说，目前其实没有啥工单系统能实现的这么好的。

这个工单系统就把流程设计，模版设计等等做的非常不错，而且对权限的把控非常详细，包括API接、菜单、页面按钮权限，都可以灵活的控制，非常的不错。

Demo：http://fdevops.com/#/dashboard

Github：https://github.com/lanyulei/ferry

如果觉得不错就给作者一个star，你的star没准就是作者继续维护下去的动力呢。

功能介绍

系统管理

- 用户管理不仅仅包括了用户，还有角色、职位、部门的管理，方便后面的工单处理人扩展。
- 菜单管理，对菜单，页面按钮，甚至是API接口的管理。
- 登陆日志，对用户登陆和退出的日志记录。

流程中心

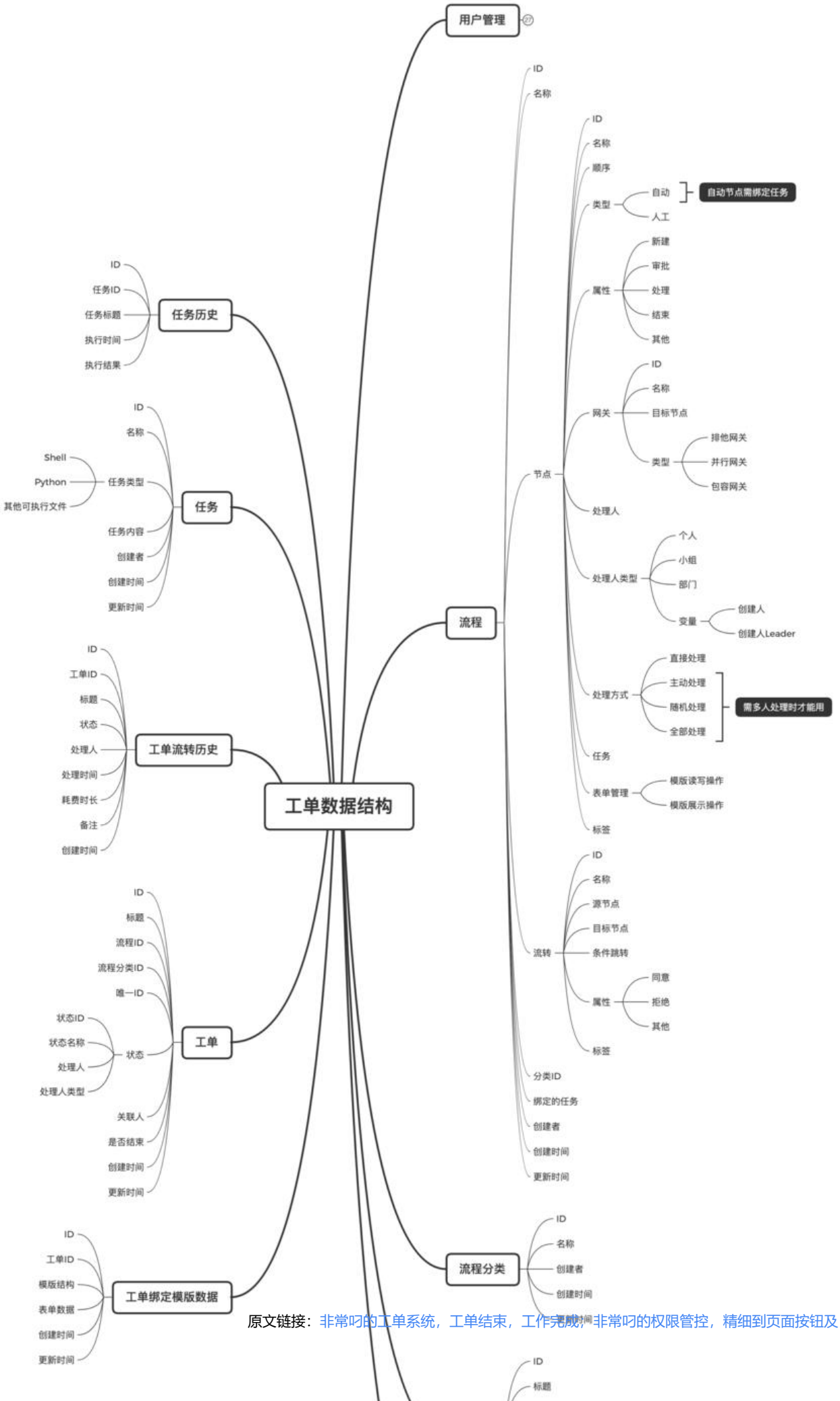
- 流程申请，对流程进行分类管理，方便维护与可视化。
- 工单列表，拆分了4个分类，包括：我待办的工单，我创建的工单，我相关的工单，还有所有工单。
- 转交工单，如果你当前有别的事情在处理就可以把工单转交给别人去处理。
- 结束工单，如果一个工单发展申请不对，权限足够的话，是可以直接结束工单的。
- 工单处理人的多样化，不仅可以个人处理，还可以是部门、角色、变量。
- 处理人变量，根据用户数据来自动获得是该谁处理，比如：创建人，创建人leader，HRBP等等。
- 会签，如果是多个选择人的话，并且勾选了会签功能，那么就需要这些负责人都处理完成后才会通。
- 任务管理，可以给任何阶段绑定任务，相当于流程中的钩子操作，实现的效果就是，工单完成，任也就执行完成了，减少很多的人力成本。
- 通知方式的灵活性，可以通过任务给每个阶段绑定通知方式，也可以给流程绑定全局通知。
- 网关，支持排他网关和并行网关，排他网关即通过条件判断，只要有一个条件通过，则可进入下一阶段；并行网关，即必须所有的阶段都完成处理，才可以进行下一个阶段
- 后面还会有很多的功能扩展，包括：加签，催办，子流程等等。

等等还有很多功能待研究。

数据结构设计

对于一个完整的工作流系统来说，我们需要有流程、模版、分组、用户、任务等等，并且这些东西都可以灵活定制的，因为如果不能灵活定制的话，对于普通的使用这来说是非常不方便的，所以对于一好的工作流系统，是必须要实现灵活性的。

下面直接来展示一下，数据结构的设计图。



原文链接：非常叼的工单系统，工单结束，工作完成，非常叼的权限管控，精细到页面按钮及 API

流程分类

```
type Classify struct {
    base.Model
    Name    string `gorm:"column:name; type: varchar(128)" json:"name" form:"name"`     // 分
名称
    Creator int    `gorm:"column:creator; type: int(11)" json:"creator" form:"creator"` // 创建者
}

func (Classify) TableName() string {
    return "process_classify"
}
```

流程

```
type Info struct {
    base.Model
    Name      string          `gorm:"column:name; type:varchar(128)" json:"name" form:"name"`
  // 流程名称
    Structure json.RawMessage `gorm:"column:structure; type:json" json:"structure" form:"struc
ure"` // 流程结构
    Classify  int             `gorm:"column:classify; type:int(11)" json:"classify" form:"classify"` // 分
ID
    Tpls      json.RawMessage `gorm:"column:tpls; type:json" json:"tpls" form:"tpls"`               //
模版
    Task      json.RawMessage `gorm:"column:task; type:json" json:"task" form:"task"`               /
 任务ID, array, 可执行多个任务，可以当成通知任务，每个节点都会去执行
    Creator   int            `gorm:"column:creator; type:int(11)" json:"creator" form:"creator"`    //
创建者
}

func (Info) TableName() string {
    return "process_info"
}
```

模版

```
type Info struct {
    base.Model
    Name          string          `gorm:"column:name; type: varchar(128)" json:"name" form:"name"
binding:"required"`                          // 模板名称
    FormStructure json.RawMessage `gorm:"column:form_structure; type: json" json:"form_stru
ture" form:"form_structure" binding:"required"` // 表单结构
    Creator       int            `gorm:"column:creator; type: int(11)" json:"creator" form:"creator"`
                                 // 创建者
    Remarks       string          `gorm:"column:remarks; type: longtext" json:"remarks" form:"rema
ks"`                                 // 备注
}

func (Info) TableName() string {
    return "tpl_info"
}
```

工单

```
type Info struct {
    base.Model
    Title        string          `gorm:"column:title; type:varchar(128)" json:"title" form:"title"`
     // 工单标题
    Process      int             `gorm:"column:process; type:int(11)" json:"process" form:"process"`
              // 流程ID
    Classify     int             `gorm:"column:classify; type:int(11)" json:"classify" form:"classify"`
       // 分类ID
    IsEnd        int            `gorm:"column:is_end; type:int(11); default:0" json:"is_end" form:"is_e
d"`          // 是否结束， 0 未结束，1 已结束
    State        json.RawMessage `gorm:"column:state; type:json" json:"state" form:"state"`
                // 状态信息
    RelatedPerson json.RawMessage `gorm:"column:related_person; type:json" json:"related_pe
son" form:"related_person"` // 工单所有处理人
    Creator      int             `gorm:"column:creator; type:int(11)" json:"creator" form:"creator"`
           // 创建人
}

func (Info) TableName() string {
    return "work_order_info"
}
```

工单绑定模版

```
type TplData struct {
    base.Model
    WorkOrder     int            `gorm:"column:work_order; type: int(11)" json:"work_order" form:
work_order"`          // 工单ID
    FormStructure json.RawMessage `gorm:"column:form_structure; type: json" json:"form_stru
ture" form:"form_structure"` // 表单结构
    FormData      json.RawMessage `gorm:"column:form_data; type: json" json:"form_data" form
"form_data"`                // 表单数据
}

func (TplData) TableName() string {
    return "work_order_tpl_data"
}
```

工单流转历史

```
type CirculationHistory struct {
    base.Model
    Title        string `gorm:"column:title; type: varchar(128)" json:"title" form:"title"`
  // 工单标题
    WorkOrder    int    `gorm:"column:work_order; type: int(11)" json:"work_order" form:"work_
rder"`              // 工单ID
    State        string `gorm:"column:state; type: varchar(128)" json:"state" form:"state"`
       // 工单状态
    Source       string `gorm:"column:source; type: varchar(128)" json:"source" form:"source"`
              // 源节点ID
    Target       string `gorm:"column:target; type: varchar(128)" json:"target" form:"target"`
           // 目标节点ID
    Circulation  string `gorm:"column:circulation; type: varchar(128)" json:"circulation" form:"cir
ulation"`       // 流转ID
```

```
    Processor    string `gorm:"column:processor; type: varchar(45)" json:"processor" form:"proc
ssor"`              // 处理人
    ProcessorId  int    `gorm:"column:processor_id; type: int(11)" json:"processor_id" form:"proc
ssor_id"`        // 处理人ID
    CostDuration string `gorm:"column:cost_duration; type: varchar(128)" json:"cost_duration"
orm:"cost_duration"` // 处理时长
    Remarks      string `gorm:"column:remarks; type: longtext" json:"remarks" form:"remarks"`
                // 备注
}

func (CirculationHistory) TableName() string {
    return "work_order_circulation_history"
}
```

任务

```
type Info struct {
    base.Model
    Name     string `gorm:"column:name; type: varchar(256)" json:"name" form:"name"`
// 任务名称
    TaskType string `gorm:"column:task_type; type: varchar(45)" json:"task_type" form:"task_ty
e"` // 任务类型
    Content  string `gorm:"column:content; type: longtext" json:"content" form:"content"`
// 任务内容
    Creator  int    `gorm:"column:creator; type: int(11)" json:"creator" form:"creator"`          //
建者
    Remarks  string `gorm:"column:remarks; type: longtext" json:"remarks" form:"remarks"`
 // 备注
}

func (Info) TableName() string {
    return "task_info"
}
```

任务执行历史

```
type History struct {
    base.Model
    Task         int    `gorm:"column:task; type: int(11)" json:"task" form:"task"`
     // 任务ID
    Name         string `gorm:"column:name; type: varchar(256)" json:"name" form:"name"`
                // 任务名称
    TaskType     int    `gorm:"column:task_type; type: int(11)" json:"task_type" form:"task_type"`
               // 任务类型, python, shell
    ExecutionTime string `gorm:"column:execution_time; type: varchar(128)" json:"execution_ti
e" form:"execution_time"` // 执行时间
    Result       string `gorm:"column:result; type: longtext" json:"result" form:"result"`
          // 任务返回
}

func (History) TableName() string {
    return "task_history"
}
```

安装部署

go >= 1.14
vue >= 2.6
npm >= 6.14

二次开发

后端

```
# 1. 获取代码
git https://github.com/lanyulei/ferry.git

# 2. 进入工作路径
cd ./ferry

# 3. 修改配置 ferry/config/settings.dev.yml
vi ferry/config/settings.dev.yml

# 配置信息注意事项:
1. 程序的启动参数
2. 数据库的相关信息
3. 日志的路径

# 4. 初始化数据库
go run main.go init -c=config/settings.dev.yml

# 5. 启动程序
go run main.go server -c=config/settings.dev.yml
```

前端

```
# 1. 获取代码
git https://github.com/lanyulei/ferry_web.git

# 2. 进入工作路径
cd ./ferry_web

# 3. 安装依赖
npm install

# 4. 启动程序
npm run dev
```

上线部署

后端

```
# 1. 进入到项目路径下进行交叉编译（centos）
env GOOS=linux GOARCH=amd64 go build

更多交叉编译内容，请访问 https://www.fdevops.com/2020/03/08/go-locale-configuration

# 2. config目录上传到项目根路径下，并确认配置信息是否正确
```

```
vi ferry/config/settings.yml

# 配置信息注意事项:
1. 程序的启动参数
2. 数据库的相关信息
3. 日志的路径

# 3. 创建日志路径及静态文件经历
mkdir -p log static/uploadfile

# 4. 初始化数据
./ferry init -c=config/settings.yml

# 5. 启动程序，推荐通过"进程管理工具"进行启动维护
nohup ./ferry server -c=config/settings.yml > /dev/null 2>&1 &
```

前端

```
# 1. 编译
npm run build:prod

# 2. 将dist目录上传至项目路径下即可。
mv dist web

# 3. nginx配置，根据业务自行调整即可
  server {
    listen 8001; # 监听端口
    server_name localhost; # 域名可以有多个，用空格隔开

    #charset koi8-r;

    #access_log  logs/host.access.log  main;
    location / {
      root /data/ferry/web;
      index index.html index.htm; #目录内的默认打开文件,如果没有匹配到index.html,则搜索index.h
m,依次类推
    }

    #ssl配置省略
    location /api {
      # rewrite ^.+api/?(.*)$ /$1 break;
      proxy_pass http://127.0.0.1:8002; #node api server 即需要代理的IP地址
      proxy_redirect off;
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    # 登陆
    location /login {
      proxy_pass http://127.0.0.1:8002; #node api server 即需要代理的IP地址
    }

    # 刷新token
```

```
    location /refresh_token {
      proxy_pass http://127.0.0.1:8002; #node api server 即需要代理的IP地址
    }

    # 接口地址
    location /swagger {
      proxy_pass http://127.0.0.1:8002; #node api server 即需要代理的IP地址
    }

    # 后端静态文件路径
    location /static/uploadfile {
      proxy_pass http://127.0.0.1:8002; #node api server 即需要代理的IP地址
    }

    #error_page  404              /404.html;    #对错误页面404.html 做了定向配置

    # redirect server error pages to the static page /50x.html
    #将服务器错误页面重定向到静态页面/50x.html
    #
    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
      root html;
    }
  }
```

自此项目就介绍完了，反正我觉得还是非常不错的，如果你也觉得不错，就给者一个star吧。