**在本章节笔记中,主要记录以下内容:**

1. **Odoo系统的文件结构**
2. **Odoo系统的模块扩展机制**
3. **确定Odoo开发学习路线**
4. **尝试开发一个小的应用**

**Odoo的架构为类MVC (Model View Controller).**

也就是在不改变底层业务逻辑的前提下, 通过配置文件改变/改进界面以及用户交互

Model 层 : 决定应用数据结构

View层 : 决定用户交互界面

Controller层: 决定应用的业务逻辑

**重要概念**

**理解应用(applications)和模块(modules)**

应用和模块作为常见的Odoo关键字而被开发者耳熟能详,但是他们的区别是 什么呢?

**Module addons(附加模块)**

附加模块是支撑起整个Odoo应用的基础. 一个模块可以为Odoo添加/修改 已有的功能. 模块以目录的形式表现,内部**必须包含一个名\_\_manifest\_\_.py 的说明文件(manifest)以及其他功能性文件.**

**Applications(应用)**

Odoo主要通过安装应用的方式加载功能

如果模块比较复杂,且为Odoo系统本身增加新的功能,则需要把模块做成一个应用.如果你的模块只是为了对现有的功能进行更改, 则不是应用.

**无论一个模块是否在说明文件(\_\_manifest\_\_.py)中被定义. 技术上它都不会影响附加模块的表现, 它只是用于在Odoo的应用列表中做高亮用**

**修改和扩展现有模块**

接下来会开发一个新的, 依赖较少的模块

这个模块不会是一个典型的模块. **绝大多数的时间,二次开发都是在修改和扩展现有的模块**

**一般来讲,用直接修改源码的方式来修改已存在的模块是一个坏的习惯**,特别是修改Odoo官方提供的模块. 这样做会很难区分官方的源码和开发者的改动,并且由于改动有可能会重写Odoo原有的方法,会使得系统的升级会变得困难.

**所以,如果需要对模块进行修改,我们应该创建扩展模块(extension modules)并安装在我们想修改的模块旁边.**事实上, Odoo的其中一个强大之处就是他的继承机制. 继承机制可以允许客户扩展官方和社区的第三方模块.继承机制适用于本系统的所有结构: model, view, controller

在本章笔记中,将会创建一个全新的,对现有模块无任何扩展的模块. 目的就是为了了解创建模块的步骤和模块本身的结构. 本章笔记只会简要的学习,更深层的理解将会在后面陆续学习.

一旦熟悉了模块的创建,对后面学习继承机制以及模块扩展有很大的帮助.

二次开发学习路线: **学习创建新模块 --> 扩展旧模块**

**为了能够开发生产用的Odoo系统,开发人员必须熟悉开发的流程:**

1. **管理开发环境**
2. **应用更新后的代码**
3. **检查结果**

**创建模块的基本框架**

创建存放模块的目录

一个odoo模块从本质上来说就是一个python包

所以odoo模块中必须包含以下文件:

\_\_init\_\_.py

\_\_manifest\_\_.py

编辑\_\_manifest\_\_.py文件 内容见pycharm

如果一个模块对其他模块有依赖设置,那么在安装本模块时, 被依赖的模 块也会被系统自动安装.因此非常建议在设计模块的时候添加有用的依 赖设定.**即使没有任何依赖,也要添加基础模块(base)**

在设定依赖的时候,一定要确认在模块中用到的其他模块被写入依赖环境,否 则模块会安装失败,失败的原因为: missing dependencies

在本次写的测试模块中,没有对其他模块的依赖, 所以设定依赖为base

一个odoo模块有非常多的属性, 在开发中最好多使用这些属性去提高模块 的完成度

Summary : 模块的副标题

Version : 模块版本

License: 证书

Website: 与本模块相关的网站

Category: 本模块所属的类别,

类别的位置在: Settings -> users -> groups -> group by XXX

# -\*- coding: utf-8 -\*-  
  
{  
 'name': 'To-Do Application',  
 'description': 'Manage your personal To-Do tasks',  
 'author': 'GaoHan',  
 'version': 'V1.0',  
 'depends': ['base','product','sale'],  
 'application': True,  
 'data': [],  
}

在模块创建完成后, 我们要确认模块存储的路径能够被系统识别

路径配置可以在openerp-server.conf中设置

模块的安装和更新都可以在系统页面中的application中查看

在二次开发中,

**如果对python的文件有修改,则必须要重启服务器后,再更新 模块才能生效**

**而如果只对配置文件xml有修改,则只需要更新模块即可**

**Model layer**

现在odoo能够识别刚才创建的新模块, 现在开始往里添加模型

模型描述了业务对象,比如机遇,销售,业务伙伴等等.一个模型拥有许多的属性来描述它特殊的业务逻辑

一个模型由一个基于python类的odoo模板定义. 模型将会直接映射至数据库模型,而odoo则会自动在安装模块的时候执行这些操作,这就是ORM思想

我们的模块是一个非常简单的任务记录模块,主要功能是记录任务内容和检查他们是否完成

**创建数据模型**

odoo官方的开发文档中指出,所有的模型文件必须放在一个名为models的python包下

# 说明本模块使用的是UTF-8编码

# -\*- coding: utf-8 -\*-

from odoo import models, fields, api  
  
# 声明一个新的模块,继承自model.Model  
class TodoTask(models.Model):

# 本模块在系统中的全局识别id  
 \_name = "todo.task"

# 本模块的友好名称  
 \_description = "To-do Task"  
 # 本模块的三个字段  
 name = fields.Char("Description", required=True)  
 is\_done = fields.Boolean("Done?")  
 active = fields.Boolean("Active?", default=True)

在创建好魔性文件后,必须在\_\_init\_\_.py中导入

并且在重启服务器和更新模块后, 可以在数据结构页面中找到todo.task

进入todo.task模块可以看到很多我们并没有声明的字段, 这些字段是odoo系统在新模型安装时自动生成的,字段如下:

Id: 模型中每一条数据的识别id

Create\_date/create\_uid: 模型中每一条数据的生成人id和生成日期

Write\_date/write\_uid: 模型中每一条数据最后一此修改的日期和操作人员

\_last\_update: 并发行检查 不常用

**View Layer**

视图层通过使用xml配置文件,调用web框架生成HTML文件来创建用户操作界面.例如用户界面可以显示一个存放所有用户信息的列表.odoo系统提供很多不同种类的视图,如列表(tree),表单(form),还有过滤器search视图

官方的开发文档建议xml文件存放在views目录下

**创建菜单和目录**

在views目录下创建todo\_menu.xml文件

<?xml version="1.0" encoding="utf-8" ?>  
<odoo>

定义一个打开todo.task模型的窗口动作,并启用列表 和表单视图  
 <act\_window id="action\_todo\_task"  
 name="To-do Task"  
 res\_model="todo.task"  
 view\_mode="tree,form"/>  
 定义一个顶部菜单 动作可根据实际情况填写

action为点击后打开的窗口id  
 <menuitem id="menu\_todo\_task" name="Todos" action="action\_todo\_task"/>  
</odoo>

在所有的配置文件中 id属性是全局唯一的,可以被其他元素引用.在上面的代码中, menuitem元素引用了打开窗口的这个元素

配置完xml文件后,需将该文件路径添加至引用模块的data路径中

**小知识**

如果在模块的升级过程中出现XML错误,不要慌张. 在\_\_manifest\_\_.py文件中的data属性里摘除最后一次编辑的xml文件引用然后再次更新即可. 随后再认真查看服务器日志里的错误信息并找出错误来源

Odoo系统支持多种视图,但是最常用,最重要的只有三种:列表,表单,查找

我们会随后在模块中逐一添加他们

**创建表单(form)视图**

**所有的视图都存储在数据库中的ir.ui.view模型中**

要想在一个模块中增加一个新的视图, 首先要在xml文件中声明一个<record>标签描述视图.当模块在安装或升级的时候,视图便会自动加载到数据库中

<?xml version="1.0" encoding="utf-8" ?>  
<odoo>  
 <record id="view\_form\_todo\_task" model="ir.ui.view">  
 <field name="name">Todo-Task Form</field>  
 <field name="model">todo.task</field>  
 <field name="arch" type="xml">  
 <form>  
 <group>  
 <field name="name"/>  
 <field name="is\_done"/>  
 <field name="active" readonly="1"/>  
 </group>  
 </form>  
 </field>  
 </record>  
</odoo>

xml文件完成后要添加进\_\_manifest\_\_.py文件中,否则该配置文件不会被加载

本配置文件首先想ir.ui.view模型中添加一条新的记录,id为view\_form\_todo\_task

而本模块的表单视图名为Todo-Task From .视图名只是一条简单的,让用户易懂的描述表单的信息,不需要是唯一值.事实上,视图名是完全可以被省略的,系统会自动以模块名+视图类别作为视图名

在配置文件中,最重要的属性是name=”arch”,它包含视图的定义.

<form>标签定义视图类型为表单,表单内容包含三个字段

**业务记录表单**

刚才的配置文件只能提供一个基础的表单视图,还有改进的空间. **odoo提供一个可以模仿纸质表单的表单模型.这种类型的表单视图包含两个标签:**

**<header> 来存放动作案件**

**<sheet>来存放字段内容**

我们可以替换刚才的配置成为一下内容

**添加动作按键**

**表单可以通过按键执行动作. 通过按键可以打开一个新的窗口/新的表单或者执行在本模型中定义的python代码**

按键可以放置在表单的任意位置,但是在一个档案类型的表单视图中,建议将按键放置于<header>标签下

对于我们的应用,我们将对todo.task模型添加两个按键

<header>  
 <button name="do\_toggle\_done"  
 type="object"  
 string="Toggle Done"  
 class="oe\_highlight"/>  
  
 <button nane="do\_clear\_done"  
 type="object"  
 string="Clear All Done"/>  
</header>

<button>标签的基础属性有:

String : 按键上显示的名字

Type: 按键的动作类型

Name: 动作的唯一标识

Class: 按键的显示样式 css/bootstrap

**使用分组(group)来组织表单**

<group>标签可以实现表单内容的布局.在<group>标签中嵌套另一个<group>标签可以在外标签中创建一个两列的布局. <group>标签推荐使用一个name属性描述它们,以方便之后的模块扩展功能

我们将使用这个标签来重新组织内容布局.修改后的<sheet>标签内容如下:

<sheet>  
 <group name="group\_top">  
 <group name="group\_left">  
 <field name="name"/>  
 </group>  
 <group name="group\_right">  
 <field name="is\_done"/>  
 <field name="active" readonly="1"/>  
 </group>  
 </group>  
</sheet>

整个表单文件如下:

<form>  
 <header>  
 <button name="do\_toggle\_done"  
 type="object"  
 string="Toggle Done"  
 class="oe\_highlight"/>  
  
 <button nane="do\_clear\_done"  
 type="object"  
 string="Clear All Done"/>  
 </header>  
 <sheet>  
 <group name="group\_top">  
 <group name="group\_left">  
 <field name="name"/>  
 </group>  
 <group name="group\_right">  
 <field name="is\_done"/>  
 <field name="active" readonly="1"/>  
 </group>  
 </group>  
 </sheet>  
</form>

**小知识**

一定要记住,对于要写入数据库的修改,必须要先进行模块升级才能生效

而配置文件中的按键因为没有写入业务逻辑所以暂时不能生效

**添加列表和搜索视图**

<tree>标签用来创建一个列表视图 列表视图可以分层显示内容,但是大部分的时间里,他们只会用来显示普通的列表

我们添加以下内容至xml文件

<record id="view\_tree\_todo\_task" model="ir.ui.view">  
 <field name="name">To-do Task Tree</field>  
 <field name="name">todo.task</field>  
 <field name="arch" type="xml">  
 <tree colors="decoration-muted:is\_done==True">  
 <field name="name"/>  
 <field name="is\_done"/>  
 </tree>  
 </field>  
</record>

上面的内容定义了列表显示的字段:name , is\_done 并且在显示is\_done==True的字段时,条目背景变为灰色. 这个功能是由bootStrap提供的.

Odoo系统自带的搜索功能在列表的右上角. 他能够搜索所有在<search>视图中定义的过滤器. 和之前一样,我们添加一下内容至xml文件

<record id="view\_filter\_todo\_task" model="ir.ui.view">  
 <field name="name">To-do Task Filter</field>  
 <field name="model">todo.task</field>  
 <field name="arch" type="xml">  
 <search>  
 <field name="name"/>  
 <filter string="Not Done"  
 domain="[('is\_done','=','False')]"/>  
 <filter string="Done"  
 domain="[('is\_done','!=','False')]"/>  
 </search>  
 </field>  
</record>

<search>标签下的<field>标签定义了一个在搜索框中输入时也会被搜索的字段

<filter>标签预定义了一个过滤器条件触发器,当用户点击搜索时,便会触发

**Logic Layer**

添加业务逻辑需要修改python代码,但是首先我们必须引入新的odoo API

from odoo import models, fields, api

Toggle Done的按键逻辑非常简单, 用来切换is\_done的 True False值.

对于记录的逻辑,必须使用@api.multi这个装饰器 而self将指代的是一个记录集合,我们必须遍历每一个记录. 在类中添加以下代码:

@api.multi  
def do\_toggle\_done(self):  
 for task in self:  
 task.is\_done = not task.is\_done  
 return True

代码将会循环遍历所有的todo-task记录并且改变每一个记录的is\_done的值

这个函数本身不需要任何的返回值, 但是实际操作中需要加入一个为True的返回值.原因是系统本身使用XML-RPC方法去调用函数,而这个方法不支持返回值为空的函数调用

对于Clear All Done这个按键,我们需要它有更深层的意义.它将会遍历所有的记录并且找出所有有效的,且已已经完成的记录,并且使他们变为无效. 一般表单按键只会影响到当前的选择的表单,但是这次我们将他设计为影响所有数据而不只是当前选中的数据

@api.model  
def do\_clear\_done(self):  
 dones = self.search(["is\_done","=",True])  
 dones.write({"active":False})  
 return True

当函数带有api.model的装饰器后,self指代模型本身,并不指代任何的记录. 我们会创建一个容器存储所有状态为已完成的记录,之后我们会将这些记录标记为失效

**系统APi自带的search方法会根据参数返回相应的值.这些参数称为domain,它以元组嵌套列表的方式表达** 我们会在之后的第六章笔记中详细说明domains

而**write函数会同时对所有的记录集合写入相同的内容,写入的内容格式为python字典的键值对类型.在这里使用write函数比遍历集合后赋值要更有效率**

**设置访问安全**

如何控制一个模块的访问规则呢?首先要了解设定访问规则需要了解的信息

在odoo客户端页面进入 Setting -> Security -> Access control list

在本页面可以看到模型的访问控制. 它指定了组和其能够执行的动作

这一系列的信息是由模型提供的.通过一个数据型的文件将内容录入进 ir,model.access这个模型中. 本模型将所有的权限交给员工组. 而员工组是odoo系统中最基本的分组而且几乎所有的用户都属于这个组

而实现这一行为需要首先在模型的根目录下**创建security目录,和ir.model.access.csv文件(文件名固定)** 并添加以下内容:

id,name,model\_id:id,group\_id:id,perm\_read,perm\_write,perm\_create,perm\_unlink  
access\_todo\_task\_group\_user,todo.task.user,model\_todo\_task,base.group\_user,1,1,1,1

文件的第一行为列表字段

Id : csv文件中每一行记录的识别id 全局唯一

Name: 对规则的描述 官方使用”模型名.用户组”的方式命名

Model\_id: 访问控制将会被实施的模型的唯一识别id 一般由ORM自行分配

对于todo.task来说 识别id为 : model\_todo\_task

Group\_id: 拥有访问权限的用户组的识别id

Perm\_XX: 分别对应对数据的增,删,改,查,操作

当编写完后,必须将文件路径添加入\_\_manifest\_\_.py的data属性中

**角色层的访问安全**

Odoo是一个多用户系统,并且我们希望to-do task对不同的用户显示它们自己的内容. 幸运的是, odoo也提供角色层的访问规则. 同样也在Access Control List中

记录规则在ir.rule模型定义.和通常一样,他需要一个唯一辨识的名称.我们同时需要一个能让它生效的模型和限制访问权限的值域(domain). 值域过滤器使用domain自己的规则并且在odoo系统通用

最后,规则可以是全局的或者只针对某一个用户组.而在我们的案例中, 它应该是是一个全局的规则,但是为了举一个最常用的实例, 我们会将他设定为一个专门为员工组的访问规则

我们要首先在security目录中创建todo\_access\_rules.xml

<?xml version="1.0" encoding="UTF-8" ?>  
<odoo>  
 <data noupdate="1">  
 <record id="todo\_task\_user\_rule" model="ir.rule">  
 <field name="name">Todo Tasks only for owner</field>  
 <field name="model\_id" ref="model\_todo\_task" />  
 <field name="domain\_force">  
 [('create\_uid','=','user.id')]  
 </field>  
 <field name="groups" eval="[(4, ref('base.group\_user'))]"/>  
 </record>  
 </data>  
</odoo>

文中的noupdate=”1”属性, 这意味着该标签下的数据将不会再模块更新时升级

但是当开发环境中,还是应该将值改为0直到内容满意为止

在group字段中,还存在一段特殊的代码.这是一个一对多的关系字段. 在本例中,(4,x)是一个元组,表示将x附到记录. 而在本例中的x是一个对员工组的引用,我们将会在第四章笔记中更多的讨论这种语法

当配置文件完成后,要添加入\_\_manifest\_\_.py文件中

**总结**

我们从头开始创建了一个新的模块,涵盖了:

1. 模块中最常用的元素:模型

2. 三个最基本的视图类型:列表,表单和查找

3. 模型中的业务逻辑函数

4. 访问权限

我们熟悉了创建模块的步骤,其中包括了模块的升级和重启服务器以确保改动生效.永远要记得:

1. 当向模型添加新的字段时,必须要升级
2. 当对python代码有改动时,必须重启服务器
3. 当改动xml或者csv文件时,必须升级改动的模块
4. 当不确定时,既重启服务器也要升级模块

在下一张笔记中,我们将学习如何对现有的模块进行升级和添加新的功能