在本章中,我们将会学习编写代码来支持模型的业务逻辑,并且你也可以学习它们是如何通过事件或者用户动作被激活的.使用Odoo的编程框架,我们可以写出复杂的逻辑和向导(wizard)来丰富用户交互体验

**创建一个向导**

假设我们的应用用户会频繁的对大量的数据设定截止日期和负责人字段.他们就需要一个小程序帮助他们.它可以允许批量的选择任务并且同时设定其负责人和截止日期.**向导(wizard)就是一个能过获取用户输入,并未下一步动作做准备的表单.**

我们可以从创建一个新的todo\_wizard模块开始:

# -\*- coding: utf-8 -\*-  
  
{  
 'name': 'To-Do Task Management Assistant',  
 'description': 'Mass edit tasks fields',  
 'author': 'GaoHan',  
 'depends': ['todo\_user'],  
 'data': [  
   
 ],  
}

**创建向导模型**

向导是以对话窗的形式向用户展示表单数据.这一基本逻辑将在向导逻辑中使用.

与定义通常模型非常相似,只不过**向导模型的需要继承models.TransientModel类而不是models.Model类**

这种瞬时态的模型在数据库中同样有表结构和数据存储.但是一旦向导的工作完成,数据就会消失.odoo系统会定时清理向导的数据表.

在向导的模型文件中,我们会定义用于和用户交互的字段:

1. 想要修改的任务列表
2. 想要修改/任命的负责人
3. 想要修改的截止日期

代码:

from odoo import models,fields,api  
  
class TodoWizard(models.TransientModel):  
 \_name="todo.wizard"  
 \_description = "To-do mass assignment"  
  
 task\_ids=fields.Many2many("todo.task", string="Tasks")  
 new\_deadline=fields.Date("New Deadline")  
 new\_res\_user\_id=fields.Many2one('res.user',string="New Responsible")

注意在瞬时模型中,不能使用一对多的关系字段.原因是它会在普通模型中产生多对一的逆关系,而这是不允许的,因为普通模型并没有垃圾回收机制.

**向导表单**

向导的表单和普通模型的表单只有两处不同,其余相同:

1. <footer> 向导的<footer>标签可以存放动作按键
2. Type=cancel

下面是配置文件代码:

<?xml version="1.0" encoding="utf-8" ?>  
<odoo>  
 <record id="view\_form\_todo\_wizard" model="ir.ui.view">  
 <field name="name">To-do Task Wizard</field>  
 <field name="model">todo.wizard</field>  
 <field name="arch" type="xml">  
 <form>  
 <div class="oe\_right">  
 <button type="object"  
 name="do\_count\_tasks"  
 sting="Count"/>  
 <button type="object"  
 name="do\_populate\_tasks"  
 sting="Get All"/>  
 </div>  
  
 <field name="task\_ids">  
 <tree>  
 <field name="name"/>  
 <field name="res\_user\_id"/>  
 <field name="deadline"/>  
 </tree>  
 </field>  
  
 <group>  
 <group>  
 <field name="new\_res\_user\_id"/>  
 </group>  
 <group>  
 <field name="new\_deadline"/>  
 </group>  
 </group>  
  
 <footer>  
 <button type="object" name="do\_mass\_update"  
 string="Mass Update" class="oe\_highlight"  
 attrs="{'invisible':[('new\_deadline','=',False),('new\_res\_user\_id','=',False)]}"/>  
 <button special="cancel" string="cancel"/>  
 </footer>  
 </form>  
 </field>  
 </record>  
  
 <act\_window id="todo\_app.action\_todo\_wizard"  
 name="To-Do Task Wizard"  
 src\_model="todo.task"  
 res\_model="todo.wizard"  
 view\_mode="form"  
 target="new"  
 multi="True"/>  
</odoo>

通过src\_model属性,<act\_window>将打开向导视图的窗口动作添加到todo.task表单的More按键中.设置target=new可以打开一个新的窗口

在批量修改按键Mass Update属性中,我们设置了只有在选择新截止日期和新负责人后才可见

**向导的业务逻辑**

接下来我们将定义刚才视图中确定的三个动作逻辑:除了cancel按钮外,我们还有三个按键需要键入.但是我们首先会专注于Mass Update按键

这个按键动作调用的函数为:do\_mass\_update,而我们将会在模型中定义这个函数,以下是代码:

@api.multi  
def do\_mass\_update(self):  
 self.ensure\_one()  
 if not (self.new\_deadline or self.new\_res\_user\_id):  
 raise exceptions.ValidationError("No data to update!")  
  
 \_logger.debug("Mass update on Todo Task %s", self.task\_ids.ids)  
  
 vals = {}  
 if self.new\_deadline:  
 vals['deadline'] = self.new\_deadline  
 if self.new\_res\_user\_id:  
 vals['res\_user\_id'] = self.new\_res\_user\_id  
 if vals:  
 self.task\_ids.write(vals)  
 return True

我们的代码每一次指挥执行一个向导实例,所以我们使用self.ensure\_none()来保证这是一个单例.这里的self指代向导表单上的数据

方法逻辑:

1. 首先确认是否有新的日期或者负责人的修改,如果没有任何修改,则报错.
2. 如果有修改,则写入服务器日志
3. 创建一个vals字典接收新日期和新负责人的值.然后对向导中选择的所有字段进行更新.这里的write方法可以对多个数据进行修改,要比循环语句更方便
4. 返回值.每一个odoo方法的结尾必须有返回值.而原因之前已经说过了,这里就不多赘述

接下来我们会详细的讨论有关日志方面的问题.

**日志**

大批量的修改数据很有可能出错,所以在修改的时候使用日志文件来记录是一个非常好的习惯.在刚才的代码中就初始化了python标准库中的\_logger变量.

可执行的日志记录

\_logger.debug("A DEBUG message")  
\_logger.info("An INFO message")  
\_logger.warning("A WARNING message")  
\_logger.error("An ERROR message")

**异常处理**

当系统运行错误时,我们希望终止运行并显示错误信息.这就是通过异常处理来实现的.Odoo提供了一些额外的异常处理类,以下是代码

from odoo import exceptions  
raise exceptions.Warning("Input warning messages")  
raise exceptions.ValidationError("Not Valid message")

警告信息(Warning)同样可以终止程序的运行,但是不如值错误(ValidationError)强烈.

我们可将Count按键的逻辑这样写:

@api.multi  
def do\_count\_task(self):  
 Task = self.env['todo.task']  
 count = Task.search\_count([('is\_done', '=', False)])  
 raise exceptions.Warning("There are %d tasks not done yet", count)

注意,看上去我们的方法装饰器应该使用@api.model,因为我们并没有访问模型数据.但是并不能这样做,因为本方法是通过按键触发

**向导对话框中的动作**

现在假设我们希望有个可以自动选取所有任务的按键.我们需要在视图中创建一个Get All的按键.它的代码会在后台接受一个所有有效任务的集合并且将他们赋值给一个多对多字段

这里有一个小技巧,在对话窗口中按下按键时,向导窗口会自动关闭.我们在Count按键中没有遇到这个问题是因为它用错误信息的方式显示消息,所以对话窗并没有关闭.

幸运的是,我们可以通过重新打开视图来解决这个问题.模型函数可以返回一个打开客户端窗口的动作.这个返回值是一个字典格式的,其中的键值对描述了系统将如何执行窗口动作.字典中的键和XML文件中的一样.

@api.multi  
def \_reopen\_form(self):  
 self.ensure\_one()  
 return {  
 "type": "ir.actions.act\_window",  
 "res\_model": self.\_name,  
 "res\_id": self.id, # 返回当前向导记录  
 "view\_type": "form",  
 "view\_mode": "form",  
 "target": "new"  
 }

Get All按键逻辑:

# 获取所有未完成任务  
@api.multi  
def do\_populate\_tasks(self):  
 self.ensure\_one()  
 Task = self.env['todo.task']  
 open\_tasks = Task.search([('is\_done','=',False)])  
 self.task\_ids = open\_tasks  
 return self.\_reopen\_form

这里我们可以知道如何引用其他模型,这里就是todo,task.向导表单中的数据以瞬时态存储在数据库中,并且可以和普通模型一样读写.这里我们将todo,task模型中所有未完成任务赋值给todo.wizard的task\_ids变量

**\*注意** 在api.multi装饰器下的self不能保证只包含一个记录,所以我们使用self.ensure\_one()函数,而不直接使用api.one装饰器.因为api.one装饰器会将返回值打包成一个列表.而客户端只接受字典返回值.

**\*使用ORM API**

在之前的所有笔记中,我们已经接触到了ORM API.接下来,我们将会详细了解它可以做什么.

**函数装饰器(<Method decorators)**

在我们的学习中,已经编写和使用了许多带有装饰器的函数,类似@api.multi这样的.这些装饰器非常重要,因为它们会告知服务器如何来处理被装饰的函数.我们来重新了解一下装饰器的使用方法和使用他们的时机

1. **@api.multi** 装饰器是所有API中最常使用的装饰器.用于处理记录集合(recordsets).**在这个装饰器下的self参数指代的是记录集**,而函数通常需要一个循环来遍历这个集合
2. **@api.one** 在很多情况下,函数只需要处理单一记录,而记录集里却包含多条数据.@api.one是不推荐使用的.相反,我们应该**使用@api.multi装饰器配合self.ensure\_one()来保证其单一性.**

虽然api.one不推荐使用,但是它仍然是被系统支持的.它的本质也是对记录集合进行迭代,每次给函数提供一条记录.在我们do\_mass\_update方法中,每次只处理一条记录.

1. @api.model 用户装饰静态类方法,他不会涉及到任何的记录集.虽然在这个装饰器下的self还是一个记录集,但内容是不相关的.**\*注意这种装饰器无法装饰按键函数**
2. @api.depends(field1) 用于计算字段函数识别需要执行哪些计算
3. @api.constrains(field1) 用于字段的有效性计算
4. @api.onchange(field1) 通过修改字段值来触发函数

通常,onchange方法可以向用户返回一条警告信息,见以下代码:

return {  
 "warning":{"title":"Warning!","message":"You have been warned"}  
}

警告信息是以字典嵌套字典的形式.在odoo的销售模型中,可以警告要出售的物品数量要少于实际库存数量,并停止用户继续生成销售订单

**重写ORM默认函数**

最常见的重写函数的例子是:重写create()和write()函数.主要是用于修改触发器逻辑.通过将我们的逻辑摆放在确切的位置,我们的代码可以在主程序之前或者之后运行

用我们设计的模型作为例子,我们可以这样定制我们的create()函数:

@api.model  
def create(self, vals):  
 # 将在父函数被调用之前执行的代码  
 pass  
  
 # 调用父函数create  
 new\_record = super(TodoTask, self).create(vals)  
  
 # 将在父函数被调用之前执行的代码  
 pass  
 return new\_record

个人理解: 主要还是利用了python函数的继承机制,通过调用父函数的方法,实现对旧函数的扩展. 在调用语句前后添加代码,可实现更多功能.write函数同理

@api.model  
def write(self, vals):  
 # 将在父函数被调用之前执行的代码  
 pass  
  
 # 调用父函数create  
 new\_record = super(TodoTask, self).write(vals)  
  
 # 将在父函数被调用之前执行的代码  
 pass  
 return True

以上便是一些常见的扩展例子,当然模型中的任何标准方法都可以用同样的方法继承和扩展.这种机制创造了更多的可能性,但是要记住,odoo系统同样提供了一些更好的更合适的方法:

1. 如果需要某个字段的值基于另一个字段.我们应该使用计算字段.例如:当某一个产品的价格变动,采购总价也会随之变动
2. 如果需要动态计算某个字段的默认值,我们可以使用计算字段+default关键字的方式
3. 如果需要在改变某个字段值同时修改其他字段,我们可以使用onchange装饰器.例如:当选择一个客户时,货币可以是客户的常用货币,并且之后可以手动修改.\***注意:onchange只会在表单视图中生效**
4. 如果需要审核字段值是否有效,我们需要@api.constraints装饰器.不同于普通计算字段,它们除了计算还需要提示错误信息

**RPC 客户端调用**

我们已经学习了如何创建记录集和修改他们.但是odoo系统中还有一些模型方法,他们可以实现一些特殊的目的,例如:

1. read([fields]) 类似browse方法.但是返回的不是一个recordset,而是一个包含参数字段所有数据的列表.每一行记录都是一个字典.也就是字典嵌套列表的格式
2. search\_read([domain],[fields],offset=0,limit=None,order=None) 功能同read,多了一个根据域表达式的搜索机制
3. load加载csv文件
4. export\_data 导出

接下来的函数是一些在客户端行常见的函数

1. name\_get()返回一个(id,name)格式的元组格式列表.
2. name\_search(name=’’,args=None,operator=’ilike’,limit=100)同样返回一个(id,name)格式的元组格式列表,只是name会严格按照name参数来搜索
3. name\_create(name)

**服务器环境**

服务器shell提供了一个self关键字来关联res.users这个模型.我们知道,self在odoo中一般指代一个记录集,而记录集(recordsets)通常携带当前的上下文环境,例如语言和时区等等的信息.**我们可以用过self.env来获取当前的上下文环境**

**当前用户环境信息(self.env)拥有以下属性:**

1. env.cr 当前的数据库游标
2. env.uid 当前会话的用户id
3. env.user 当前用户
4. env.context 当前会话的上下文字典

我们同样可以使用self.env来获取当前系统中已安装的模型.例:

self.env['res.partner'].search([('name','like','Ag')])

代码的前半部分返回一个res.partner模型的引用,并且使用这个引用通过search()方法进一步获取模型中的数据

**\*总结**

**在之前的章节中,我们学习了如何建立模型和设计视图.在本章中我们更进一步的讨论了如何添加业务逻辑并且使用记录集来操作模型数据**

**我们同样学习了业务逻辑是如何和用户界面交互的.并且通过创建向导使用户能对数据进一步控制**

**本章节非常重要.建议反复研读**