odoo其中一个最强大的功能就是能够基于已经创建的模块进行修改和升级扩展

能实现这个功能的**原因就是继承机制(Inheritance mechanisms),在现有的模块顶端作为修改层运作.这些修改可以在所有的工作层中实现:model, views, business logic. 我们将会重新创建一个全新的模块去扩展现有的模块,而不是改写现有的模块内容**

在本章笔记中,将学习如何写一个扩展模块,从而修改核心模块或者第三方模块.作为实例,我们将给在上章中创建的模块添加社交功能

我们的模块现在可以用于管理个人的任务,但是为其加入合作和社交功能岂不是更好吗?这样就可以共享某些任务并和其他人一起讨论

我们将通过继承之前创建的模块来实现这一目的,工作步骤如下:

1. 扩展模型结构, 添加新的字段
2. 调整业务逻辑是的当前用户只能查看和编辑个人的工作计划
3. 扩展视图,为界面添加新的字段内容
4. 为模块添加社交功能

我们首先开始创建新模块todo\_user的基本架构,和前两张笔记相同,首先创建模块目录和一些必须的python文件

编辑\_\_manifest\_\_.py 文件并添加以下内容

# -\*- coding: utf-8 -\*-  
  
{  
 "name":"Multi-user To-Do",  
 "description":"An extension to the To-Do app",  
 "author":"GaoHan",  
 "depends":['base','todo\_app'],  
 "data":[],  
}

注意在depends属性中添加todo\_app的值,意味着本模块和todo\_app有直接的依赖关系.这一点在系统的系统的继承机制中是非常重要的.至此,只要todo\_app更新,依赖它的所有模块,例如本模块,都会自动更新

接下来,安装本模块 在应用列表中找到本模块并安装 具体的方法在第一章有详细的步骤,这里就不多赘述.模块安装完成后,就可以添加新的功能

**扩展模型**

新的模型通过python类定义. 而扩展他们同样借助系统的继承机制和python类

**要想扩展一个已存在的模型,在创建python类中必须要添加\_inherit变量,**它的作用就是识别将要被扩展的模型id. 子模型将会继承父模型的所有功能,我们只需要声明要修改的地方即可

**事实上,odoo模型是存在于python模块之外,在中央注册表内.这个注册表可以通过模型方法访问 self.env[“模型名”]. 例如,如果想获得对res.partner模型对象的引用,只需要在代码中添加 self.env[“res.partner”]即可**

想要去改动一个odoo模型,只需要获取模型的注册类的引用,然后就地修改即可.这意味着只要新模型被使用,改动就会生效

当odoo服务被启动时,模块的加载序列是由相关性的, 一个附加模块所做的修改只会在随后加载的附加模块上可见. 所以,正确设定模块之间的依赖关系至关重要.

**为模型添加字段**

我们将为todo.task模型添加一些新的字段: 任务的负责人,和任务的截止日期

创建模型文件的方法相见之间的两张笔记 这里就不多赘述

1. 创建python包: models
2. 创建todo\_task.py文件并添加以下内容

from odoo import models,api,fields  
  
class TodoTask(models.Model):  
 \_inherit = "todo.task"  
  
 res\_user\_id = fields.Many2one("res.users","Responsible")  
 deadline = fields.Date("DeadLine")

TodoTask这个类名只对本文件生效,而且一般来说,它对其他模块是无意义的

**类中的\_inherit属性才是重点, 它会告诉odoo这个类会继承todo.task模型**

**小知识**

值得注意的是,在这个新类中并没有像之前一样写入\_name, \_description这样的变量.原因就是它会从我们设定的类中继承

而之后的代码便是新声明的两个字段 res\_user\_id代表为本任务计划负责的负责人. 而它的值是一个对已存在的模型res.user的引用 就像数据库中的外键一样

而deadline是任务计划的截止日期 是一个短日期类型 我们会在第五章的笔记中详解odoo系统中的各种字段类型

**调整已存在字段**

可以看到,向已存在的模型中添加新的字段非常直接.自从odoo8后,对已存在的模型的字段进行调整也是可能的. 添加一个相同名字的字段,然后修改要改动的参数即可

例如,如果我们要对name字段添加一个提示信息 我们可以在todo\_task.py的代码中添加以下内容:

name = fields.Char(help="What needs to be done?")

以上代码会将添加新的参数至原来的name变量中,但是 其他的变量和没有设置的参数则不会发生任何改变.这时如果我们重启服务器然后更新模块,将鼠标悬停在Description上 就会出现What needs to be done 的提示

**调整模型的方法**

继承机制在业务逻辑层同样有效.添加一个新的方法非常简单:只需要在子函数中声明一个新的函数即可

在扩展业务逻辑的操作中,如果子模型中的函数名和父模型的函数名相同,子模型函数会重写(覆盖)父模型的函数.但是使用python 的super()函数,可以在子函数中调用父函数

在实际的开发环境中,为了确保现有的调用可以正常运行,最好不要修改现有函数的参数. 如果需要添加函数, 将他们设置为可选择的,并设置一个默认值

由于原有的Clear All Done会无视当前用户修改数据,所以他不再适合我们现在的多用户分享型模块.我们需要把它修改为只会清楚当前用户的任务

所以,我们将会重载原来的方法:它将首先列出当前用户所有的已完成的任务,然后再使他们无效(归档):

@api.multi  
def do\_clear\_done(self):  
 domain =[("is\_done","=",True),  
 "|",  
 ("res\_user\_id","=",self.env.uid),  
 ("res\_user\_id","=",False)]  
  
 done\_recs=self.search(domain)  
 done\_recs.write({"active":False})  
 return True

我们首先使用search(domain)的方法将所有is\_done=True的记录选择出来.

而domain可以理解为:odoo系统特殊语法的过滤器

Domain是一个包含多种条件的,元组嵌套列表形式的,每一个元组代表一个条件.

在元组之间添加 “&”,”|” 用来表达 与(AND)/或(OR). 上面的代码”|”会影响后后面的两个元组.

**例如[(A),”|”,(B),(C)] 的结果集为 A条件 + (B条件OR C条件)**

关于domain我们会在后面第六章笔记中详细讨论

所以以上代码表示:筛选出所有is\_done = True, 负责人id为当前用户OR没有负责人的任务

我们刚才所写的代码完全重写了父方法,使该方法有了新的功能.但是这并不是我们真正想要做的.我们应该向现有的逻辑中添加新的选项,否则我们会破坏现有的特性.现有的逻辑可以使用super()函数调用

我们看一个例子: 我们应该重写一个更好的do\_toggle\_done()函数,让只有当前任务的负责人才可以触发该函数

@api.multi  
def do\_toggle\_done(self):  
 for task in self:  
 if task.res\_user\_id == self.env.user:  
 return super(TodoTask, self).do\_toggle\_done()  
 else:  
 raise ValidationError("Only the responsible can do this!")

这些就是最基本的重写和扩展已有的业务逻辑的方法接下来我们将学习如何扩展用视图界面

**扩展视图**

表单,列表,查找视图使用xml配置文件定义. 所以扩展视图的步骤为:

1. 定位要修改的元素🡪定位元素id
2. 修改值

<?xml version="1.0" encoding="utf-8" ?>  
<odoo>  
 <record id="view\_todo\_task\_inherited" model="ir.ui.view">  
 <field name="name">Todo Task Form - Extension</field>  
 <field name="model">todo.task</field>  
 <field name="inherit\_id" ref="todo\_app.view\_form\_todo\_task"/>  
 <field name="arch" type="xml">  
 </field>  
 </record>  
</odoo>

**Inherit\_id 配合使用ref属性可以准确定位到要继承的视图文件id**

更多的外部标签信息将会在第四章笔记中详细探讨

在一般情况下,定位xml配置内容的位置需要使用XPath标签(更多关于XPath的信息可以到python官方网站查询),但是odoo框架替我们简化了这一操作,方便我们在定位的字段附近添加新的字段

<field name="is\_done" positon="before">  
 <field name="deadline"/>  
</field>

以上代码表示:在父模块的视图字段is\_done的位置之前 添加子模块的字段deadline

**注意: 如果同一字段在同一视图中出现多次, 就必须使用XPath.**这是由于odoo会在字段出现一次后停止

通常,我们希望在原字段的附近添加新的字段,所以<field>标签会经常作为定位器来使用.但是任何其他的标签例如<sheet><group><div>也可以作为定位器使用.虽然使用<field>标签中的name属性最适合匹配元字段,但是特殊情况下,我们会使用例如CSS样式的其他方法

**小知识**

在odoo 9.0之前,string属性也可以用作定位器.但是由于语言翻译的限制,9.0过后不能被当作定位器使用

定位器中的position属性是多项可选的:

1. After 在定位的字段之后添加
2. Before 在定位的字段之前添加
3. Replace 新字段替代定位字段
4. Attribute 改变定位字段的属性

例如在我们的模型中有一个叫active的可见字段,我们将把它修改为不可见,代码如下

<field name="active" positon="attribute">  
 <attribute name="invisible">1</attribute>  
</field>

以上代码的含义为:

1. 找到父模块视图中active变量的对应字段并修改它的属性值
2. 修改的属性为invisible 值为1

将变量设置为不可见的方式要比直接替换节点的方式要好一些.在开发过程中应该尽量避免直接替换或者删除节点,因为这样有可能会同时删除依赖它的其他字段造成不必要的损失或错误

**扩展form视图**

将本章之前的代码结合起来,我们会得到一下结果:

1. 为模块添加两个新的字段

将负责人放在任务描述之后

将任务截止日期放在是否完成之前

1. 将原字段active设置为不可见字段

总体代码如下:

<?xml version="1.0" encoding="utf-8" ?>  
<odoo>  
 <record id="view\_form\_todo\_task\_inherited" model="ir.ui.view">  
 <field name="name">Todo Task Form - Extension</field>  
 <field name="model">todo.task</field>  
 <field name="inherit\_id" ref="todo\_app.view\_form\_todo\_task"/>  
 <field name="arch" type="xml">  
 <field name="name" positon="after">  
 <field name="res\_user\_id"/>  
 </field>  
  
 <field name="is\_done" positon="before">  
 <field name="deadline"/>  
 </field>  
  
 <field name="active" positon="attribute">  
 <attribute name="invisible">1</attribute>  
 </field>  
 </field>  
 </record>  
</odoo>

将代码完成后, 记得将配置文件路径写入\_\_manifest\_\_.py中

**小知识**

**扩展视图同样可以被扩展,但是在实际应用中不推荐这样使用**

**扩展列表和搜索视图**

列表和搜索试图同样通过XML文件定义,而他们的扩展方式和表单视图一样

在列表视图中,我们希望添加任务负责人res\_user\_id字段

<record id="view\_tree\_todo\_task\_inherited" model="ir.ui.view">  
<record id="view\_tree\_todo\_task\_inherited" model="ir.ui.view">  
 <field name="name">Todo Task Tree - Extension</field>  
 <field name="model">todo.task</field>  
 <field name="inherit\_id" ref="todo\_app.view\_tree\_todo\_task"/>  
 <field name="arch" type="xml">  
 <field name="name" position="after">  
 <field name="res\_user\_id"/>  
 </field>  
 </field>  
</record>

而对于查找视图,我们将会添加通过负责人查找功能,显示自己任务的过滤器和没有任何负责人的任务

<record id="view\_filter\_todo\_task\_inherited" model="ir.ui.view">  
 <field name="name">Todo Task Filter - Extension</field>  
 <field name="model">todo.task</field>  
 <field name="inherit\_id" ref="todo\_app.view\_filter\_todo\_task"/>  
 <field name="arch" type="xml">  
 <field name="name" position="after">  
 <field name="res\_user\_id" />  
 <filter name="filter\_my\_tasks"  
 string="My Tasks"  
 domain="[('res\_user\_id','in',[uid,False])]"/>  
 <filter name="filter\_not\_assigned"  
 string="Not Assigned"  
 domain="[('res\_user\_id','=',False)]"/>  
 </field>  
 </field>  
</record>

现在不要担心代码中的一些特殊语法,我们会在第六章笔记中详细解释

**更多的模型继承机制**

我们之前已经学习了基本的模型扩展,在官方文档中被称作**类继承(class inheritance).**这个是继承机制的最常用方法,并且把他简单理解为**原地扩展(in-place extension).** 选择一个已存在的,要进行功能扩展的模型,写入新的功能,这个过程并不会创建一个新的模型.**通过向\_inherit属性添加继承列表,我们还可以继承多个不同的模型.** 有了这个功能,我们就可以使用**混合类(mixin class).我们可以将混合类理解为一个可以存放各种扩展特性的容器.**

如果我们在创建继承类的时候使用和父类不一样的\_name变量的值,那么我们将会得到一个全新的,继承父类全部特性和字段的,拥有独立数据库数据的模型.官方文档称之为**原型继承(Prototype inheritance).** 标准继承相当于复制模型, 当你在子模型上做任何修改时,父模型不会有任何的改变

还有一种继承方式被称作**委托继承(delegation inheritance),使用\_inherits属性.**

**\*注意此处为\_inherits不是\_inherit.**它允许一个模型在观察角度上透明的包含另一个模型(it allows a model to contain other models in a transparent way for the observer),但是在后台, **每一个模型的数据是独立的.当开发人员向子模型添加新的特性时, 父模型不会做任何的改动.子父模型之间会创建一个数据记录的关联,并且父模型的数据是可以被子模型直接使用的.**

下面为详解:

1. **使用原型继承(prototype inheritance)复制特性**

在本章笔记的一开始我们介绍并使用\_inherit变量做继承.我们首先定义了一个继承自**todo.task**模型的类并且添加了一些功能,而类变量**\_name**并没有被设置,所以实际上,本模型还是**todo.task.**

然而,使用**\_name**变量会创建一个新的,完全复制父模型功能的模型.见代码:

# todo.task模型中嵌入mail.thread模型  
class TodoTask(models.Model):  
 \_name = "todo.task"  
 \_inherit = "mail.thread"

以上代码会将**mail.thread**模型中的功能**复制(copying)**到**todo.task**模型中. Mail.thread模型中涵盖odoo管理系统中的社交系统而且是可重复使用的.所以将它添加至任何模型都是非常容易的

这里**复制(copying)**意味着父模型的方法和字段,子模型同样可以使用.对于字段来说,这同样意味着字段会在目标模型中的数据库表中被创建和存储.**可以理解为子模型了复制了父模型的数据库表结构?** 父模型(inherited)的数据和子模型(inheriting)的数据是不相关的,他们只是共享定义

稍后,我们将会详细讨论如何使用mail.thread模型和其社交功能.在实践中,当我们将使用混合类(mixins), 基本不会从一般模型中继承,因为会引起相同的数据结构重复化

Odoo系统同时提供**授权继承(delegation inheritance)**机制防止数据结构的重复化. 通常它是继承常规模型的首选方法

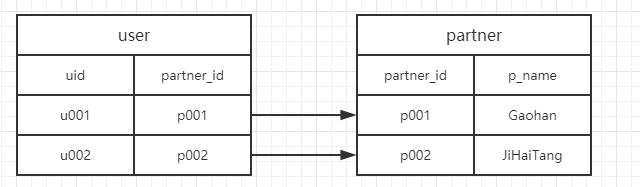
1. **使用授权继承(Delegation Inheritance)嵌入模型**

授权模型并不经常被使用,但是它可以提供非常方便的解决方案. 它通过使用**\_inherits**变量(\*注意不是**\_inherit**),在字段和模型之间创建字典映射

一个非常好的例子是标准的用户模型:res.user,其中嵌入了合作伙伴(partner)模型

# res.users模型中嵌入res.partner模型  
class User(models.Model):  
 \_name = "res.users"  
 \_inherits = {"res.partner":"partner\_id"}  
 partner\_id = fields.Many2one("res.partner")

通过授权继承,在res.users模型中嵌入父模型res.partner,使得当一个新的user实例被创建时,一个partner实例也被创建并且在user类的表中创建partner\_id字段指向partner实例.这种特性类似面向对象编程中的多态



通过授权机制,所有的父模型的字段和合作伙伴都可以访问,就像他们本身是user模型的属性一样.例如: 合作伙伴(partner)的姓名和地址字段就像用户(user)的字段一样,但是实际上他们存储在映射的合作伙伴(partner)模型里,并不会出现数据重复的现象

对比原型继承,这种继承方式的好处是不会出现重复的数据结构.比如多表共同使用的地址字段.任何一个需要使用地址字段的新模型都可以授权嵌入一个partner模型.而且如果有对partner的地址做任何的修改,所有被嵌入的模型都可以立刻访问到

**小知识**

**当使用授权继承时,字段会被继承,但是函数并不会**

**使用继承机制添加社交网络特性**

Odoo的社交模块(mail)提供了一套完整的表单底部追踪功能和消息版,也可以成为:**Open Chatter**

社交模块是由mail.thread模型提供的.要将这一功能嵌入任何的模块,我们需要执行以下操作:

1. 将模块设定依赖于mail模型
2. 创建类继承mail.thread
3. 在form视图中创建新的Chatter视图
4. 设置订阅者的记录规则
5. 创建依赖就是将mail模块添加进模块的依赖列表,依赖列表在\_\_manifest\_\_.py文件中

"depends":['base','todo\_app','mail'],

1. 创建继承,就是将模型\_name添加入\_inherit变量中.之前我们已经对\_inherit变量赋值过了.然而在odoo中 \_inherit变量也可以是一个存储继承模型名的列表

修改todo\_task.py文件:

\_inherit = ["todo.task","mail.thread"]

Mail.thread是一个抽象模型(Abstract models). 抽象模型无法创建数据库,也不能被直接使用. 但是他们可以作为mixin类,将某一个功能嵌入另外一个模块.我们可以把抽象模型看作一个能实现特定功能的模板. 创建抽象模型,必须要继承models.AbstractMode 而不是models.Model.

1. 视图方面,我们希望社交模块的界面出现在表单的底部,这是通过我们之前学到的扩展视图来完成的. 在扩展表单的配置文件中定位源表主体<sheet>标签,添加一下数据

<sheet position="after">  
 <div class="oe\_chatter" >  
 <field name="message\_follower\_ids" widget="mail\_followers"/>  
 <field name="message\_ids" widget="mail\_thread"/>  
 </div>  
</sheet>

这里的两个字段我们并没有声明,因为它是由mail.thread模型提供的

1. 最后已不是为订阅者设置记录规则.这里我们只需要保证每一个订阅者能过读取数据即可.我们确实需要模型的记录规则,这点我们将在下面介绍

**调整数据**

不同于视图,普通的数据记录并没有xml结构,也不能使用XPath去扩展.但是它们仍然可以被修改替换

加载数据记录使用的标签:<record id=”x” model=”y”>实际上是进行了一次对y模型的插入或升级动作. 如果模型x不存在的话,就重新创建一个x模型,否则将会升级/覆盖

由于其他模块的数据可以通过<model>.<identifer>的全局识别访问.所以我们的模块重写其他模块已经写入的数据是可能的

**小知识**

因为点是系统预留用来访问变量的,所以在创建识别id时不能使用点,应使用下划线

**修改菜单和动作记录**

作为一个例,我们将原来的顶部菜单名改为My To-Do

我们将在todo\_user/views/todo\_task.xml文件中添加以下代码:

<record id="todo\_app.menu\_todo\_task" model="ir.ui.menu">  
 <field name="name">My To-Do</field>  
</record>

我们也可以编辑顶部菜单的点击动作. 动作有一个可选择的上下文属性,他可以提供视图和过滤器的默认值. 我们将使用这个功能实现当点击顶部菜单时,默认只显示自己的任务列表. 同样在todo\_task.xml中添加以下代码:

<record id="todo\_app.action\_todo\_task" model="ir.actions.act\_window">  
 <field name="context">  
 {'search\_default\_filter\_my\_tasks':True}  
 </field>  
</record>

**扩展记录规则**

我们的模块包含一条记录规则那就是只有任务的创建者才能看到任务明细.但是现在有了新的社交特性,我们需要任务的订阅者也可访问他们.这一点,社交模块本身并不能实现这一功能. 同时,新的任务模块会向每个任务分配负责人,所以负责人有权限访问也是非常合情合理的这同我们之前扩展顶部菜单的过程非常相似: 重写todo\_app.todo\_task\_user\_rule去改变domain并赋予新值.

但可惜的是,这次并不能成功. 还记得我们之前修改的noupdate=1吗?既然那些数据不想允许升级,我们就需要换一种思路.那就是删除原来的记录然后替换上一个新的

为了确保所有的文件都是有组织的,我们将创建一个security目录,并在目录下创建todo\_access-rules.xml 然后添加以下代码:

<?xml version="1.0" encoding="UTF-8" ?>  
<odoo>  
 <data noupdate="1">  
 <record id="todo\_app.todo\_task\_user\_rule" model="ir.rule">  
 <field name="name">To Do Tasks for owner and followers</field>  
 <field name="model\_id" ref="model\_todo\_task"/>  
 <field name="groups" eval="[(4,ref('base.group\_user'))]"/>  
 <field name="domain\_force">  
 ['|',  
 ('res\_user\_id','in',[user.id,False]),  
 ('message\_follower\_ids','in',[user.partner\_id.id])]  
 </field>  
 </record>  
 </data>  
</odoo>

以上代码会找到父模型的todo\_task\_user\_rule并重写他的内容

通过使用domain,我们使任务表单能够被负责人访问.如果任务表没有指定任何负责人,则可以被所有人访问

**总结**

通过学习,目前我们已经能够创建新的模块并扩展他们.我们本章已就在前一章中创建的模块进行了功能和视图的扩展

在不同的工作层中添加新的特性才能使一个应用完整.在本章中,我们

1. 扩展原有模型,并为其添加新的字段和业务逻辑
2. 扩展视图文件,使得新的字段可以显示在新视图上
3. 学习如何通过继承其他模块的功能来扩展本模块,实例就是向todo模块中添加了社交的功能

学习了前三章笔记,我们对Odoo开发过程有了一个大概的了解,从Odoo的安装到创建模块再到扩展模块.在之后的学习中,我们将会着重学习特殊的领域,很多我们已经在之前的笔记中粗略了解过.再下一章中,我们将会接触数据的序列化,和进一步使用CSV,XML文件