在之前的章节中,我们了解了如何创建一个新的模块.在第二章中,我们创建了一个全新的应用,而在第三章中,我们了解了odoo的继承机制并且学习了如何利用这个机制扩展我们的模块.在第四章中,我们讨论了如何给我们的模块添 加演示数据

在之前的笔记中,我们接触到创建一个odoo后端应用程序所需的所有层.现在,和之后的章节中,我们要更深入的学习这些工作层:模型(model),视图(view),和业务逻辑(businesss logic).

在本章中,我们将会学习如何设计一个应用中的数据结构和如何表达他们之间的关系

**将应用特性融入模块中**

我们将使用一个实例去解释内容

Odoo的继承特性提供了一个有效的扩展机制.它允许开发人员扩展一个第三方的模块而不是直接改动它的源代码.这样的组合性使**面相模块开发(model-oriented development)**成为可能.这样大的应用可以拆分成为小的特性快,而其内容足够使他们独立

这种方式可以减少系统在开发和应用上的复杂性.从技术层面上来讲,将大问题分割成小问题会使问题整体更容易去解决并且对**incremental feature development(功能增长型开发)**很友好.而在用户体验方面,我们可以选择只激活那些他们需要的特性.所以我们将通过添加新的模块来改进我们的todo应用使它变得完整

**创建todo\_ui模块**

在之前的章节中,我们首先创建了一个记录个人任务的应用并且随后将它扩展为任务可多人共享

现在我们将会升级我们的模块中的用户界面,为其添加一个看板(Kanban board).看板是一个简单的,可以将数据组织为一栏的工作流工具.当设计完成时,这些栏目将由左向右排列.我们将根据任务的完成情况:等待(waiting),准备好(ready),已开始(started),已完成(Done) 来将任务融入栏目中

我们将从添加数据结构开始一步一步完成我们的设想.我们首先需要添加**阶段(Stage)**,而且要为其添加标签,使得这些任务可以被分类.在本章中,我们只会着重研究数据结构.而他们的用户界面特性我们将会在第六章中学习.看板将会在第九章中学习.

第一件要明白的事情是我们的数据结构将如何设计才能支持我们的模型.我们现在已经有了核心的内容:To-do Task.每一个任务(Task)只能同时存在一个任务进度,而且单个任务可以同时挂载一个或者多个标签.我们需要添加两个额外的模块,而他们之间的关系是这样的:

1. 每一个都有一个阶段,每一个阶段可以包含很多任务
2. 每一个任务可以拥有很多标签,而每一个标签可以属于多个任务

这意味着任务和阶段之间是多对一关联,和标签之间是多对多关系.另一方面,他们之间的逆关联为:阶段和任务之间是一对多关联,标签和任务之间是多对多关联

我们将从创建新的todo\_ui模块开始,并为其添加todo\_stages和todo\_tags模型,我们将使用custom-addons来存放我们的模型.

**创建模型**

为了使应用有一个看板界面(Kanban),我们首先需要阶段(Stage).看板是看板中的栏目.每一个任务将嵌入到这些看板中

在models目录中创建todo\_model.py文件,并写入以下代码:

from odoo import models, fields, api  
  
class Tag(models.Model):  
 \_name = "todo.task.tag"  
 \_description = "To-do Tag"  
  
 name = fields.Char('Name', 40, translate=True)  
  
class Stage(models.Model):  
 \_name = "todo.task.stage"  
 \_description = "To-do Stage"  
 \_order = "sequence.name"  
  
 name = fields.Char('Name',40,translate=True)  
 sequence = fields.Integer("Sequence")

以上代码中,我们创建了两个将会在todo-task引用的新模型

在stage类中,它继承了models.Model类,这意味将会在odoo系统中创建一个id为todo,task.stage的模型. 在stage模型中还存在两个字段:name和sequence.此外还有一些带下划线前缀的模型属性.我们接下来将讨论这些内容

**模型属性**

模型类使用一些额外的属性(变量)去控制他们的行为,以下为一些常用的属性:

1. \_name: \_name变量是odoo模型的是内部标识,在创建模型时,必须含有此变量
2. \_description: \_description是对本模型的用户友好型描述.是可选变量,但是非常推荐添加
3. \_order: 当模型数据被浏览时,\_order定义了默认的数据排列顺序.他的本体其实是一条字符串格式的sql查询语句,所以他的值你可以任意填写.字符串是可翻译的并且支持多对一字段

为了模型的完整性,这里还有一些额外的属性适用于其他的实例:

1. \_rec\_name: 当纪录被相关字段引用时,描述这条记录,通常默认使用name字段.但是这条属性允许我们时候其他字段
2. \_table:表示存储本模型数据的数据库表名.通常它是自动生成的.但是可以通过本属性分配特殊名称

正如我们之前提到的,我们还可以添加\_inherit和\_inherits属性

**模型和python类**

Odoo模型通过python类描述.在之前的代码中,我们创建了一个python类:Stage,继承自models.Model类,而那些代码定了一个名为todo.task.stage的odoo模型

模型被存储的在核心注册表中,在旧的odoo版本中被称为池(pool).它是一个字典型的容器确保当前实例中的所有模型都是可以通过使用模型名称访问.具体来讲,模型方法中的代码可以通过使用self.env[‘x’]获得一个代表x模型的引用.

可以发现模型名是非常重要的因为它们是访问注册表的键.模型名的命名规则是使用小写英文字符用点连接的形式,例如:todo.task.stage 其他来自核心模型的例子如:project.project, project.task, project.task.type.我们要沿用之前创建的模型名todo.task作为前缀而不是todo.tasks. 由于历史原因,有一些核心模型并不遵循这一规则,例如:res.users, 但这不是规则

模型名必须全局唯一.由于这个原因,名称的第一个单词必须和模块以来的应用的模块一致. 例如我们的todo. 其他的例子例如:project crm或者sale

Python类,附属于声明该类的python文件.类的标识只能够在此文件中生效.由于这个原因,类的标识并没有强制要求在关联的主程序中预先声明.例如,将todo.task.stage命名为模型名是没有问题的,因为在其他模块上很有可能不会存在相同的名称

类标识的命名方法有两中:snake\_case 或者 CamelCase. 在之前,odoo使用snake\_case,并且现在仍然可以使用这种命名格式找到对应的类.但是目前的趋势是使用CamelCase,因为这是python语言官方的编写格式.整个笔记都已经在使用这种命名方式

**瞬时和抽象模型**

在我们之前的学习和多数的odoo模型中,类都是继承自models.Model类.这些类在数据库中都有是持久化的:数据库将为类本身以及类数据创建表来存放并且直到被删除为止.

但是odoo系统仍然提供了两种额外的模型种类:**瞬时类和抽象类**

1. **瞬时模型(Transient models)**继承自model.TransientModel类,而且可以被当作wizard-style的用户交互.他们的数据虽然仍存储在数据库中,但是它是暂时的.瞬时类的表数据会阶段性的被清除. 。例如，在Settings | Translations菜单中发现的“Load a Language”对 话框窗口，使用一个临时模型来存储用户选择并实现向导逻辑
2. **抽象模型(Abstract models)**继承自model.AbstractModel类,并且不会有任何的数据存储.这种类被当作用于其他模型中,可重复使用的特性容器.例如我们之前使用的mail.thread就是一个抽象模型.

**创建字段**

在创建一个新模型后,下一步是为模型添加字段.odoo支持所有的数据基本类型.比如字符串,整形,浮点型,Bool类型,时间类型,和图片/二进制类型

一些字段名是特殊的.那是因为他们被ORM(面向关系模型)留用其他目的了,或者因为某些预安装的系统默认特性使用了这些字段名

让我们看学习下odoo中的几种字段类型

**基础字段类型**

现在我们创建一个Stage模型,我们将会为它添加一些额外的字段,添加以下代码:

class Stage(models.Model):  
 \_name = "todo.task.stage"  
 \_description = "To-do Stage"  
 \_order = "sequence.name"  
  
 # 字符串型字段  
 name = fields.Char('Name', 40, translate=True)  
 description = fields.Text('Description')  
 state = fields.Selection( [('Draft', 'New'), ('Open', 'Started'), ('Done', 'Closed')],'State')  
 docs = fields.Html('Documentation')  
  
 # 数字型字段  
 sequence = fields.Integer("Sequence")  
 percent\_complete = fields.Float("%Complete", (3, 2))  
  
 # 时间型  
 date\_effective = fields.Date("Effective Date")  
 date\_changed = fields.Datetime("Last Changed")  
  
 # 其他类型  
 fold = fields.Boolean("Folded?")  
 image = fields.Binary("Image")

在这里我们创建一些非关系型字段,这些字段都有一些可选择的参数

在实际应用中,第一个参数通常是字段的标题(显示在页面上),使用string=””的参数.这是一个可选择的参数,如果不提供的话,系统默认使用字段名代替

对于日期型的字段,有个命名规则.例如:我们应该使用date\_effecitve而不是effective\_date这种格式.同样的规则也可以用在其他的字段上,例如amount\_, price\_或者qty\_  
以下是每一个字段需求的标准位置参数:

1. Char字段的第二个参数是字符串的最大长度.同样也是可选参数.如果没有业务上的要求,不建议使用该参数.使用定长的例子:社保卡号码
2. Text 不同于字符串类型,他可以容纳多行的上下文,与Char有相同的参数
3. **Selection 是一个下拉形选择列表. 第一个参数是一个包含选项内容的列表,第二个参数是字符串类型的标题.选项内容列表是一个可嵌套多个元组的列表.每一个元组都代表一个选项.格式为(‘value’,’Title’).value是将要被存储在数据库中的值, 而Title是在用户界面看到的值所对应的描述.当字段被继承时,可以通过传入selection\_add参数传入新的选项**
4. HTML 是以text字段类型存储的,但是加载到用户界面的方法却是特别的.
5. Integer 参数为一个字符串类型的描述
6. Float 有第二个可选参数:一个(x,y)元组和字段的精度:x是数字个数的总数,y时小数精度
7. Date和Datetime分别对应短时间和长时间 长时间精确到时分秒
8. Boolean 布尔类型只有string一个参数
9. Binary 二进制类型也只有一个参数

除了以上字段意外,还有许多关系型字段,这些内容将会在本章稍后介绍.但是现在, 我们仍然要对上述字段以及其参数进行扩展

**常见的字段属性**

字段在定义的时候可以创建属性.根据字段类型,几种属性可以根据位置传递而不使用参数关键字.例如;name=field.Char(“Name”,40),参数部分就等同于(String=”Name’,size=40)

所有的可用的属性都可以通过参数传递.一下是一些常用的可用属性和对应的参数关键字

1. String 是字段的默认标签,用于用户界面.除了选择和关系字段外,都位于参数第一位置.
2. Default 是字段的默认值. 它可以是一个静态值,例如字符串.也可以一段可以调用的赋值函数.命名函数和匿名函数皆可
3. Size 只对应字符串字段 用于设置字符串的最大值,最好不用设置
4. Translate 之对应字符串,长文本,和页面字段,使当前字段内容可被翻译
5. Help 在鼠标悬停在字段上时显示帮助信息
6. Readonly = True 让字段的值默认只读 后台可以修改,但是前台不可以
7. Required=True 说明改字段是一个必填字段,通过数据库中的NOT NULL设置
8. Index=True 在数据表中为该字段创建索引
9. Copy=False 当使用重复记录功能时,字段会被忽略 默认值是True
10. Groups 允许当前字段只能被制定用户组成员访问 格式为用逗号分隔的安全组XMLid列表 例如 groups=’base.group\_user, base.group\_system’

**特殊字段名**

在ORM中,odoo系统预留了一些字段:

1. Id 是自动生成的每一条记录的唯一标识,可以作为数据库的主键.它是自动加载到每一个模型中的
2. Create\_uid 记录创建者的id
3. Create\_date 记录创建的时间
4. Write\_uid 最后一次修改本记录人员的id
5. Write\_date 最后一次修改的日期

这些信息可以通过在客户端开启开发模式查看

**模型之间的关系**

我们在回过头看一下模块设计,拥有一下的关系:

1. 每一个任务都有一个阶段: 这是一个多对一模型,可以理解为外键关联. 其逆关系为:多对一关系.意思是多个任务可以同时处在同一阶段中
2. 每一个任务可以包含多个标签: 这是一个多对多关系. 所以逆关系同样为多对多关系.原因是一个标签也可以用于标识多个任务记录

我么将下面的代码写入todo\_ui/todo\_model.py文件中

class TodoTask(models.Model):  
 \_inherit = "todo.task"  
 stage\_id = fields.Many2one("todo.task.stage", string="Stage")  
 tag\_ids = fields.Many2many("todo.task.tag", string="Tag")

上述的代码只显示了这些字段的基本语法:设置关系模型和字段的显示标题.关系模型的字段命名规则为添加\_id或者\_ids

作为练习,可以尝试在关系模型上创建对应的逆关系

1. 多对一的的逆关系为一对多: 每一个阶段可以对应多个任务.这种关系应当在Stage类中定义
2. 多对多的逆关系为多对多: 每个标签可以对应多个任务,这种关系应当在Tag类中定义

我们仔细分析一下关系字段的定义

**多对一关系(Many-to-One)**

多对一关系字段可以接受两个位置参数:关系模型(对应的模型关键字)和标题.它将在数据库表中创建一个与之关联的外键

关系字段同样还可以包含下面的参数:

1. Ondelete 定义了当关系模型记录被删除时,本条记录将如何操作.默认值是null.意味着当关系字段被删除时,本条字段值将会被设置为空.其他的值域选项为**:restrict**(当删除时提示错误信息)和**cascade**(关系字段记录删除,本条记录同样删除)
2. Context 是一个数据的字段web端视图用
3. Domain 之前介绍过,列表中嵌套元组的过滤器
4. Auto\_join=True 允许关系映射(ORM)在查询的时候使用SQL join语法.如果允许,关系字段的访问权限将会被忽略,但是相应的会提高数据的查询速度

**多对多关系(Many-to-Many)**

多对多关系同样可以至少接受两个关系参数,关系模型关键字和标题

在数据库层面上,这种关系不会对现有的表中添加任何新字段.它将自动的创建一个新的关系表,表中只包含两表的ID值来表达他们的对应关系.关系表的表名和字段名是自动生成的.关系表的表名由两个表名相加并包含一个\_rel后缀组成.

在某些情况下,我们也许需要改写这些默认的设置.

其中一个情况是由于关系模型的表名长于PostgreSQL规定的63个字符. 这时我们就必须手动的为关系表赋一个表名.

另一个情况是,当我们为同样的两个模型创建第二张关系表.这时就要手动写入关系表表名.

想要重写这一默认规则有两种方法:使用位置参数或者关键字参数

tag\_ids = fields.Many2many(  
 "todo.task.tag", #关系模型  
 "todo\_task\_tag\_rel", #关系表表名  
 "task\_id", #本条数据的字段名  
 "tag\_id", #关系模型数据的字段名  
 string="Tags"  
)

**注意**

额外的参数为可选择的,我们可以只设置关系表名而字段名可以由系统自动生成.

其实也可以使用关键字参数,一些开发人员会认为这种方式可读性强

tag\_ids = fields.Many2many(  
 comodel\_name="todo.task.tag", #关联模型  
 relation="todo\_task\_tag\_rel", #关系表表名  
 column1="task\_id", #本条数的字段名  
 column2="tag\_id", #关联模型数据的字段名  
 string="Tags"  
)

**和多对一一样,多对多模型同样支持domain和context参数**

**一对多关系(One-to-Many)**

在我们的例子中,一对多关系可以使我们简单的列出一个阶段中的所有任务.在Stage中需要添加一下代码:

task\_ids = fields.One2many(  
 comodel\_name="todo.task",  
 inverse\_name="stage\_id",  
 string="Tasks in this stage")

一对多关系可以接受三个可选参数:关联模型,本条记录在数据库中的字段名,和显示标题.第一和第二个参数对应的关键字名为:**comodel\_name 和 inverse\_name**

其他可选参数和多对一字段一样:context domain ondelete auto\_join

**分层关系(Hierarchic Relationships)**

父子树关系(Parent-child tree relationship)使用同一模型中的多对一关系,导致每一条子记录都关联其父记录.一对多的逆关系能使父记录追踪子记录更简单

Odoo提供了对层次结构的该机支持.用于更快的浏览数中的修堤姐妹,以及在域表达式中使用额外的child\_of运算符进行更简单的搜索

为了额启用这些特性,我们需要设置\_parent\_store标志属性,并将其添加到模型的帮助字段:parent\_left和parent\_right.注意,这个额外的操作是在存储和执行时间上进行的,所以更适合频繁的读取而不是写入

我们将为todo\_model.py文件中添加如下代码:

\_parent\_store = True  
\_parent\_name = "parent\_id" # the default  
parent\_id = fields.Many2one(  
 "todo.task.tag",  
 "Parent Tag",  
 ondelete="restrict"  
)  
parent\_left=fields.Integer("Parent Left",index=True)  
parent\_right=fields.Integer("Parent Right",index=True)  
  
child\_ids = fields.One2many(  
 "todo.task.tag",  
 "parent\_id",  
 "Child Tags"  
)

\_parent\_store = True 添加层次搜索支持

\_parent\_name = “parent\_id” 引用的父类字段

**使用动态引用字段**

常规关系字段引用一个固定的模型,而引用字段类型(Reference filed type)则不会被这一规则限制并且支持动态关系,同一字段可以引用多个模型的字段

例如: 我们可以向todo-task中添加一个refers to字段,而这个字段既可以引用User也可以引用Partner,以下是代码:

refers\_to = fields.Reference(  
 [("res.user","User"),("res.partner","Partner")],  
 string="Refers To"  
)

显而易见的是,引用字段类型和选择(Selection)类型相似,只不过这里的选项是模型.而在用户界面中,用户首先选择一个可用的模型,再从模型中选择记录.

同样也可以使用一种更灵活的方法:

Referenceakbe Models配置表用于配置在Referece字段中使用的模型.它可以在Database Structure菜单中使用.在创建任何reference字段时,我们可以调用refereceable\_models()方法获取所有注册的可以引用的模型

refers\_to = fields.Reference(  
 referenceable\_models,  
 string="Refers To"  
)

**计算字段**

字段可以通过函数计算赋值,而不仅仅是通过读取数据库中的数据赋值.通过计算的字段生命和通常字段相同,不同的是传入的可以调用的函数参数

在大多数的事件中,计算字段通常需要业务逻辑的支持,所以我们将在第七章笔记中深入研究.但是我们在这里也会简单讨论,将业务逻辑设计的越简单越好

例如:Stage有一个fold字段,我们将在模块中添加一个Folded?标记来对应Stage

在todo\_model.py中添加以下代码:

stage\_fold = fields.Boolean(  
 string="Stage Folded?",  
 compute="\_compute\_stage\_fold"  
)  
  
@api.depends("stage\_id.fold")

def \_compute\_stage\_fold(self):  
 for task in self:  
 task.stage\_fold = task.stage\_id.fold

以上代码创建一个新的stage\_fold字段和用于为其赋值的\_compute\_stage\_fold函数.函数名通过字符串传递,但同时也可以通过函数标识传递.而在这种情况下,我们要确保在字段声明前,函数及已经被定义

**在函数计算需要用到其他字段时,需要使用@api.denpeds()装饰器**.它的作用是让服务器知道何时去重新计算存储或缓存值. 装饰器至少接受一个字段名类型的参数,并且使用点符号来表明字段关系

计算型函数需要为一个或多个字段赋值,如果没有赋值,将会报错.由于这里的self是一个记录对象,我们的计算只是为了获得stage\_id.fold的值.结果是通过该值赋给计算字段,即stage\_fold.

**查找和写入计算字段**

我们刚刚创建的计算字段只能读取,但是不能被查找或写入.要启用这些功能,我们需要为其嵌入特殊的功能.和计算函数一样,我们需要:

1. 创建查找函数,嵌入查找逻辑
2. 创建逆函数,嵌入写入逻辑

我们的计算字段stage\_fold将变为:

stage\_fold = fields.Boolean(  
 string="Stage Folded?",  
 compute="\_compute\_stage\_fold",  
 search="\_search\_stage\_fold",  
 inverse="\_write\_stage\_fold"  
)

def \_search\_stage\_fold(self, operator, value):  
 return [("stage\_id.fold", operator, value)]  
def \_write\_stage\_fold(self):  
 self.stage\_id.fold = self.stage\_id

搜索函数会在(field,operator,value)中的任意条件出现在搜索域中出现时被调用

Inverse函数执行计算的反向逻辑,周到计算的源字段上写的值.在我们的例子中,这代表在stage\_id.fold字段上写回

**储存计算字段**

计算字段的值也可也被存储在数据库中,通过设置store的值为True.当他们的依赖的值改变的时候,将会被重新计算.由于他们的值被存储,他们可以向其他普通字段一样被搜索,这样搜索函数就不是必须的

**模型约束**

为了增强数据的完整性,模型也支持两种约束:SQL和 python

SQL约束是基于PostgreSQL的数据库表约束,这种约束通过\_sql\_constraints定义

格式为[(约束识别,SQL约束语句,错误信息提示)]

例如:我们不希望有两个名字相同的被激活的任务:

\_sql\_constraints = [("todo\_task\_name\_unique",  
 "UNIQUE(name,active)",  
 "Task title must be Unique!")]

[Python约束则可以使用专用的代码去确认条件.检查函数必须使用@api.constraints](mailto:Python约束则可以使用专用的代码去确认条件.检查函数必须使用@api.constraints)装饰器,表明需要检查的字段.函数将会在字段修改的时候被自动调用

例如:要检测任务名至少含有5个字节的长度,我们要使用下列的约束:

**总结**

我们彻底解释了模型和字段,并使用他们扩展了todo app

学习了如何去定义模型之间的关系,包括层级的父子关系

最后,我们了解了简单的计算字段和python约束

下一张我们将了解用户界面和后台模型的特性,是他们能通过试图访问.