本文来自Odoo 13官方文档之开发者文档系列文章

本文介绍 Odoo 的代码指南。旨在提升应用代码的质量。适当的编码提升可读性、缓解可维护性、有助于调试工、降低复杂度并提升可靠性。这些指南应当用于每个新模块及新的开发。

> ⚠警告
>
> 在修改**稳定版本**已有文件时原文件样式严格替代其它样式指南。换句话说不要修改已有文件来应用这样指南。它避免了扰乱各代码行的审阅历史。应保持最小化的Diff。更多详情请参见 **拉取请求(PR)指南**。

> ⚠警告
>
> 在**master (开发)版本**中修改已有文件时仅对所修改代码或大部分处于审阅状态文件应用那些指南。换句话说仅在进行大幅变化时修改已有文件结构。这时首先进行一个**move** 提交，然后对相关的功能应用修改。

模块结构

目录

模块组织在重要的目录中。包含业务逻辑；查看一下应当会了解模块的用途。

data/: 演示和数据xml

models/: 模型定义

controllers/: 包含控制器(HTTP 路由)

views/: 包含视图和模板

static/: 包含网页资源，分离为 *css/, js/, img/, lib/, ...*

其它组成模块的可选目录。

wizard/: 重组临时模型(`models.TransientModel`) 及其视图

report/: 包含基于SQL视图的可打印报表和模型。Python对象和XML视图也在该目录中

tests/: 包含Python测试类

文件命名

文件命名对于在odoo插件中快速查找信息非常重要。本节讲解如何在标准odoo模块中进行文件的命名。作为示例我们使用 育苗 应用。它包含两个主要模型*plant.nursery* 和 *plant.order*。

有关*models*， 通过属于相同主模型的模型组来分离逻辑。每组处于基于主模型命名的给定文件中。如果仅有一个模型，其名称与模块名相同。每个继承模型应放在自己的文件中来有助于理解所受影响的模型。

```
addons/plant_nursery/
|-- models/
|   |-- plant_nursery.py (first main model)
|   |-- plant_order.py (another main model)
|   |-- res_partner.py (inherited Odoo model)
```

有关*security*和访问权限及规则时应使用两个主要文件。第一个是访问权限应在`ir.model.access.csv`文件中进行定义。用户组在`<module>_groups.xml`中进行定义。访问规则在`<model>_security.xml`中定义。

```
addons/plant_nursery/
|-- security/
|   |-- ir.model.access.csv
|   |-- plant_nusery_groups.xml
|   |-- plant_nusery_security.xml
|   |-- plant_order_security.xml
```

有关*views*, 后台视图应类似模型进行分离并由`_views.xml`进行后缀。后台视图有列表、表单、看板、活动、图表、透视表... 视图。 要缓解视图中按模型的分离不关联具体动作的主菜单可提取为可选的`<module>_menus.xml`文件。模板 (QWeb页面主要用于门户/网站展示)和资源包 (JS 和 CSS资源的导入) 放在单独的文件中。它们分别为`<model>_templates.xml`和`assets.xml`文件。

```
addons/plant_nursery/
|-- views/
|   | -- assets.xml (import of JS / CSS)
|   | -- plant_nursery_menus.xml (optional definition of main menus)
|   | -- plant_nursery_views.xml (backend views)
|   | -- plant_nursery_templates.xml (portal templates)
|   | -- plant_order_views.xml
|   | -- plant_order_templates.xml
|   | -- res_partner_views.xml
```

有关*data*, 按照用途(演示或数据)和主模型分离它们。用户名将为由`_demo.xml`或`_data.xml`后缀的main_model名称。例如具有带有演示和数据的应用，针对与mail模型相关的主模型及子类型、活动和邮件模板：

```
addons/plant_nursery/
|-- data/
|   |-- plant_nursery_data.xml
|   |-- plant_nursery_demo.xml
|   |-- mail_data.xml
```

有关*controllers*, 通常属于单个控制器的所有控制器包含在一个名为`<module_name>.py`的文件中。Odoo中老的惯例是将该文件命名为`main.py`，但现在认为已过时。如果需要从另一个模块中继承已有控制器应在`<inherited_module_name>.py`中进行。例如，在应用中添加门户控制器在`portal.py`中完成。

```
addons/plant_nursery/
|-- controllers/
|   |-- plant_nursery.py
|   |-- portal.py (inheriting portal/controllers/portal.py)
|   |-- main.py (deprecated, replaced by plant_nursery.py)
```

有关*静态文件*, Javascript文件全局遵循与python模型相同的逻辑。每个组件应使用一个有意义的名称放在自己的文件中。 例如，活动控件位于mail模块的`activity.js`中。也可以创建子目录来架构‘package’ (参见web模块获取更多详情)。相同的逻辑应对JS控件的模板（静态XML文件）及它们的样式（scss文件）进行应用。不要在Odoo之外链接数据 (图像、库)：不要使用图片的URL而又在基代码中拷贝它。

有关*向导*, 命名惯例与python模型相同：`<transient>.py`和`<transient>_views.xml`。两者都放在wizard目录中。来自老的odoo应用的w 命名对临时模型使用wizard关键字。

```
addons/plant_nursery/
|-- wizard/
|   |-- make_plant_order.py
|   |-- make_plant_order_views.xml
```

有关*统计报表* python / SQL视图和经典视图命名如下：

```
```

```
addons/plant_nursery/
|-- report/
|   |-- plant_order_report.py
|   |-- plant_order_report_views.xml
```

有关*可打印报表*主要包含数据准备和Qweb模板命名如下：

```
addons/plant_nursery/
|-- report/
|   |-- plant_order_reports.xml (report actions, paperformat, ...)
|   |-- plant_order_templates.xml (xml report templates)
```

因此我们的Odoo模块的完整目录树如下：

```
addons/plant_nursery/
|-- __init__.py
|-- __manifest__.py
|-- controllers/
|   |-- __init__.py
|   |-- plant_nursery.py
|   |-- portal.py
|-- data/
|   |-- plant_nursery_data.xml
|   |-- plant_nursery_demo.xml
|   |-- mail_data.xml
|-- models/
|   |-- __init__.py
|   |-- plant_nursery.py
|   |-- plant_order.py
|   |-- res_partner.py
|-- report/
|   |-- __init__.py
|   |-- plant_order_report.py
|   |-- plant_order_report_views.xml
|   |-- plant_order_reports.xml (report actions, paperformat, ...)
|   |-- plant_order_templates.xml (xml report templates)
|-- security/
|   |-- ir.model.access.csv
|   |-- plant_nusery_groups.xml
|   |-- plant_nusery_security.xml
|   |-- plant_order_security.xml
|-- static/
|   |-- img/
|   |   |-- my_little_kitten.png
|   |   |-- troll.jpg
|   |-- lib/
|   |   |-- external_lib/
|   |-- src/
|   |   |-- js/
|   |   |   |-- widget_a.js
|   |   |   |-- widget_b.js
|   |   |-- scss/
|   |   |   |-- widget_a.scss
|   |   |   |-- widget_b.scss
|   |   |-- xml/
|   |   |   |-- widget_a.xml
|   |   |   |-- widget_a.xml
|-- views/
|   |-- assets.xml
|   |-- plant_nursery_menus.xml
|   |-- plant_nursery_views.xml
|   |-- plant_nursery_templates.xml
|   |-- plant_order_views.xml
|   |-- plant_order_templates.xml
|   |-- res_partner_views.xml
|-- wizard/
|   |--make_plant_order.py
|   |--make_plant_order_views.xml
```

文件名应仅包含 `[a-z0-9_]` (小写字母数字及 `_`)

> ⚠警告
>
> 使用正确的文件权限： 文件夹755 及文件 644。

XML文件

格式

要在XML中声明记录，推荐使用 **record** 标记符 (使用 *<record>*) :

在 `model` 前放置 `id` 属性

对于字段声明，首先为 `name` 属性。然后将值要么放到 `field` 标签中，要么放到 `eval` 属性中，最终其它属性 (widget, options, ...)按重要性排序。

尝试通过模型对记录分组。在动作/菜单/视图之间依赖的情况下，可能无法应用这一惯例。

使用在下一点中定义的命名规范

标签 *<data>*仅用于通过 `noupdate=1` 设置不可更新的数据。如果在文件中仅存在不可更新数据，可对 `<odoo>` 标签设置 `noupdate=1` 且不设置 `<data>` 标签。

```
<record id="view_id" model="ir.ui.view">
    <field name="name">view.name</field>
    <field name="model">object_name</field>
    <field name="priority" eval="16"/>
    <field name="arch" type="xml">
        <tree>
            <field name="my_field_1"/>
            <field name="my_field_2" string="My Label" widget="statusbar" statusbar_visible="draft,sent,progress,done" />
        </tree>
    </field>
</record>
```

Odoo支持自定义标签来作为语法糖:

menuitem: 使用它来作为声明 `ir.ui.menu` 的快捷方式

template: 使用它来声明仅要求视图中 `arch` 版块的 QWeb视图。

report: 用于声明报表动作

act_window: 在record标记符无法实现时使用它

前4个标签的推荐度高于 *record* 标记。

XML ID和命名

安全、视图和动作

使用如下模式:

对于菜单：`<model_name>_menu`，或是什么子菜单用 `<model_name>_menu_do_stuff` 。

对于视图： `<model_name>_view_<view_type>`，其中的*view_type* 为 `kanban`，`form`，`tree`，`search`，…

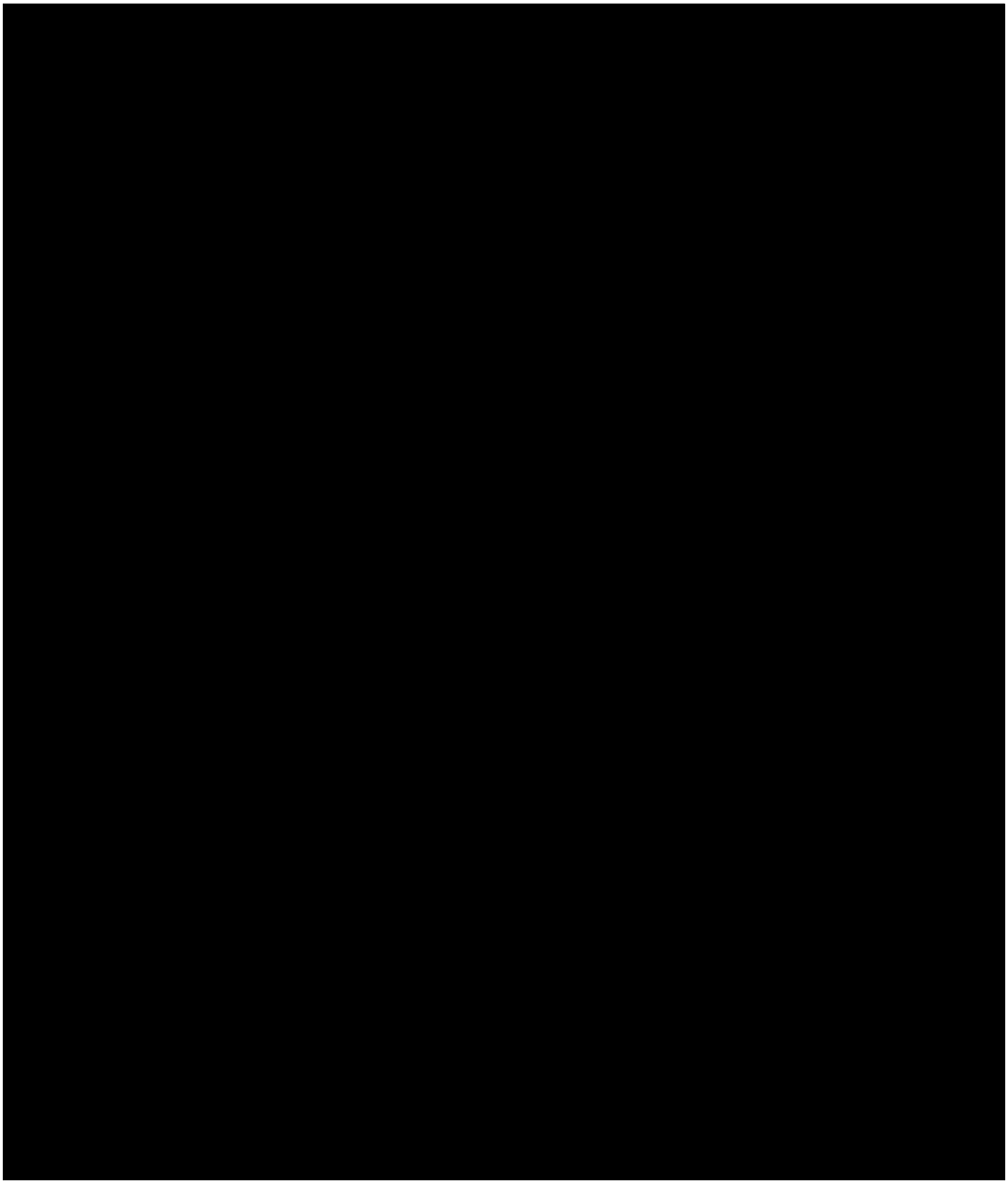
对于动作: 主动作为 `<model_name>_action`。其它使用 `_<detail>` 作为后缀，其中*detail* 为简洁地解释动作的小写字符串。仅用于多个动作对模型进行声明时。

对于窗口动作： 通过具体的视图信息如 `<model_name>_action_view_<view_type>` 对动作名进行后缀。

对于组: `<model_name>_group_<group_name>` 其中 *group_name* 是组的名称，通常为 ‘user’，‘manager’，…

对于规则: `<model_name>_rule_<concerned_group>` 其中*concerned_group* 是相关组的短名称（‘user’ 对应 ‘model_name_group_user’，‘public’ 对应公共用户，‘company’ 对应多租户规则, …).

名称应与xml id相同，使用点号替换下划线。动作应有一个真实命名，因为它用作显示名。



```
<!-- views  -->
<record id="model_name_view_form" model="ir.ui.view">
    <field name="name">model.name.view.form</field>
    ...
</record>

<record id="model_name_view_kanban" model="ir.ui.view">
    <field name="name">model.name.view.kanban</field>
    ...
</record>

<!-- actions -->
<record id="model_name_action" model="ir.act.window">
    <field name="name">Model Main Action</field>
    ...
</record>

<record id="model_name_action_child_list" model="ir.actions.act_window"
>
    <field name="name">Model Access Childs</field>
</record>

<!-- menus and sub-menus -->
<menuitem
    id="model_name_menu_root"
    name="Main Menu"
    sequence="5"
/>
<menuitem
    id="model_name_menu_action"
    name="Sub Menu 1"
    parent="module_name.module_name_menu_root"
    action="model_name_action"
    sequence="10"
/>

<!-- security -->
<record id="module_name_group_user" model="res.groups">
    ...
</record>

<record id="model_name_rule_public" model="ir.rule">
    ...
</record>

<record id="model_name_rule_company" model="ir.rule">
    ...
</record>
```

继承XML

继承视图的Xml Id应使用与原记录相同的ID。它有助于一眼看到所有的继承。因最终的 Xml Id会使用所创建的模块作为前缀，所以不会产生重叠。

命名应包含一个 `.inherit.{details}` 后缀来有且于在查看名称时理解重载的目的。

```
<record id="model_view_form" model="ir.ui.view">
    <field name="name">model.view.form.inherit.module2</field>
    <field name="inherit_id" ref="module1.model_view_form"/>
    ...
</record>
```

新的主要视图不要求继承后缀，因为它们是基于第一个的新记录。

```

```

```
<record id="module2.model_view_form" model="ir.ui.view">
    <field name="name">model.view.form.module2</field>
    <field name="inherit_id" ref="module1.model_view_form"/>
    <field name="mode">primary</field>
    ...
</record>
```

Python

PEP8选项

使用linter可帮助显示语法及句法警告或错误。Odoo源代码尽力遵守Python标准，但其中有一些可予以忽略。

E501: 行内容过长

E301: 应有一个空行，但无空行

E302: 应有两个空行，但仅有一个

导入

导入的顺序为

外部库 (每行一个并在python stdlib中分离)

`odoo`的导入

来自Odoo模块的导入 (很少见，仅在需要时进行)

在以上3组中，导入的行按字母排序。

```
# 1 : imports of python lib
import base64
import re
import time
from datetime import datetime
# 2 : imports of odoo
import odoo
from odoo import api, fields, models, _ # alphabetically ordered
from odoo.tools.safe_eval import safe_eval as eval
# 3 : imports from odoo addons
from odoo.addons.website.models.website import slug
from odoo.addons.web.controllers.main import login_redirect
```

编程规范(Python)

每个python文件的第一行应带有`# -*- coding: utf-8 -*-`.

易读性高于**简洁性**或使用语法或惯例。

不要使用`.clone()`

```
# 不妥
new_dict = my_dict.clone()
new_list = old_list.clone()
# 妥
new_dict = dict(my_dict)
new_list = list(old_list)
```

Python字典：创建及更新

```
# -- creation empty dict
my_dict = {}
my_dict2 = dict()

# -- creation with values
# 不妥
my_dict = {}
my_dict['foo'] = 3
my_dict['bar'] = 4
# 妥
my_dict = {'foo': 3, 'bar': 4}

# -- update dict
# 不妥
my_dict['foo'] = 3
my_dict['bar'] = 4
my_dict['baz'] = 5
# 妥
my_dict.update(foo=3, bar=4, baz=5)
my_dict = dict(my_dict, **my_dict2)
```

使用有意义的变量/类/方法名

无用变量：临时变量可通常为对象给定名称来让代码更清晰，但那并不表示应当总是创建临时变量：

```
# 无意义
schema = kw['schema']
params = {'schema': schema}
# 更简化
params = {'schema': kw['schema']}
```

多个返回点在更为简化时是OK的

```
# 有点复杂并带有冗余的临时变量
def axes(self, axis):
        axes = []
        if type(axis) == type([]):
                axes.extend(axis)
        else:
                axes.append(axis)
        return axes

 # 更清晰
def axes(self, axis):
        if type(axis) == type([]):
                return list(axis) # 克隆axis
        else:
                return [axis] # 单元素列表
```

了解内置函数：至少应对所有的Python内置函数有一个基本的了解

```
value = my_dict.get('key', None) # very very redundant
value = my_dict.get('key') # good
```

同时，`if 'key' in my_dict` 和 `if my_dict.get('key')` 的含义大相径庭，确保要正确使用。

学习列表推导式：使用列表推导式、字段解析式及使用 `map`，`filter`，`sum`，... 的基本操作。它们会让代码更易于阅读。

```
# 不太好
cube = []
for i in res:
        cube.append((i['id'],i['name']))
# 更佳
cube = [(i['id'], i['name']) for i in res]
```

集合也是布尔型：在python中，很多对象在布尔上下文（如if）中运行时具有“类布尔型”的值。其中有集合(列表、字典、集合...)，在为空时为 “假”，在包含内容时为“真”：

```
bool([]) is False
bool([1]) is True
bool([False]) is True
```

因此可以编写 `if some_collection:` 来代替 `if len(some_collection):`.

对可遍历内容进行遍历：

```
# 创建临时列表并查找bar
for key in my_dict.keys():
        "do something..."
# 更佳
for key in my_dict:
        "do something..."
# 访问键、值对
for key, value in my_dict.items():
        "do something..."
```

使用 dict.setdefault

```
# 更长.. 更难以阅读
values = {}
for element in iterable:
    if element not in values:
        values[element] = []
    values[element].append(other_value)

# 更好.. 使用 dict.setdefault 方法
values = {}
for element in iterable:
    values.setdefault(element, []).append(other_value)
```

作为一个优秀开发者，对代码添加文档 (方法中的docstring，复杂代码部分的简单注释)

这些指南以外，以下链接也会有帮助：

http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html (有点老，但非常相关)

Odoo中的编程

避免创建生成器和装饰器：仅使用Odoo API所自带的

和在python中一样，使用 `filtered`，`mapped`，`sorted`，... 方法来改善可读性、优化性能。

让方法批量生效

在添加函数时，确保其可通过遍历self的每条记录来处理多条记录。

```
def my_method(self)
    for record in self:
        record.do_cool_stuff()
```

对于性能问题，（例如）在开发‘stat 按钮’时，不要在循环中执行 `search` 或 `search_count`。推荐使用 `read_group` 方法来在一个请求中计算所有值。

```

```

```
def _compute_equipment_count(self):
""" Count the number of equipement per category """
    equipment_data = self.env['hr.equipment'].read_group([('category_id'
, 'in', self.ids)], ['category_id'], ['category_id'])
    mapped_data = dict([(m['category_id'][0], m['category_id_count']) fo
r m in equipment_data])
    for category in self:
        category.equipment_count = mapped_data.get(category.id, 0)
```

传送上下文

上下文是无法修改的 `frozendict` 。 要通过不同的上下文调用方法，应使用 `with_context` 方法：

```
records.with_context(new_context).do_stuff() # 替换了所有的上下文
records.with_context(**additionnal_context).do_other_stuff() # additionn
al_context的址覆盖原生上下文
```

> ⚠警告
>
> 在上下文中传递参数可能会伴随危险的副作用。
>
> 因为值是自动传送的，所以可能会出现一些预期外的行为。在上下文中通过*default_my_field*键调用 `create()` 方法会对相应的模型设置*my_field*的默认值。但在创建期间，其它对象 (如创建sale.order时的sale.order.line)有一个*my_field* 字段名会创建，也会设置它们的默认值。

如需创建一个影响对象行为的关键上下文，选择一个好名称，并最终使用模块的名称作为前缀来隔离影响。一个好例子是 `mail` 模块的键：*mail_create_nosubscribe*, *mail_notrack*, *mail_notify_user_signature*, ...

不要跳过ORM

在ORM可完成相应任务时不应直接使用数据库游标！这样可以传递所有的ORM功能，可能是事务、访问权限等等。

并且很有可能会让代码更难以阅读、安全性更低。

```
# 极其极其错误
self.env.cr.execute('SELECT id FROM auction_lots WHERE auction_id in ('
+ ','.join(map(str, ids))+') AND state=%s AND obj_price > 0', ('draft',
))
auction_lots_ids = [x[0] for x in self.env.cr.fetchall()]

# 无注入，但仍错误
self.env.cr.execute('SELECT id FROM auction_lots WHERE auction_id in %s
'\
            'AND state=%s AND obj_price > 0', (tuple(ids), 'draft',))
auction_lots_ids = [x[0] for x in self.env.cr.fetchall()]

# 更佳
auction_lots_ids = self.search([('auction_id','in',ids), ('state','=',
'draft'), ('obj_price','>',0)])
```

禁止SQL注入!

在手动使用SQL查询时要注意不要引入SQL注入漏洞。在用户输入没有正确进行过滤或引用有问题时就会出现漏洞，让攻击者可以使用预期外的SQL查询语句y (如绕过过滤或执行 UPDATE或DELETE命令)。

最好的方式是永远永远不要使用Python字符串拼接 (+) 或字符串参数插值 (%) 来传递变量到SQL查询字符串中。

第二个原因，也同样重要，决定如何格式化查询参数是数据库抽象层(psycopg2)的任务，而不是你的任务！例如psycopg2知道在你传递值列表时需要将其格式化为逗号分隔列表，使用括号包裹！

```
# 以下非常不妥：
#   - 它是一个SQL注入漏洞
#   - 可读性差
#   - 格式化id列表不是你的任务
self.env.cr.execute('SELECT distinct child_id FROM account_account_cons
ol_rel ' +
           'WHERE parent_id IN ('+','.join(map(str, ids))+')')

# 更佳
self.env.cr.execute('SELECT DISTINCT child_id '\
           'FROM account_account_consol_rel '\
           'WHERE parent_id IN %s',
           (tuple(ids),))
```

这非常重要，请注意在重构时，最重要的是不要拷贝这些模式！

以下是一个易记忆的示例， 帮助我们记住问题所在 (但不要拷贝其中的代码)。在继续之前，请确保阅读pyscopg2的在线文档在学习正确的使用方式：

> 查询字符串的问题 (http://initd.org/psycopg/docs/usage.html#the-problem-with-the-query-parameters)
>
> 如何通过psycopg2传递参数 (http://initd.org/psycopg/docs/usage.html#passing-parameters-to-sql-queries)
>
> 高级参数类型 (http://initd.org/psycopg/docs/usage.html#adaptation-of-python-values-to-sql-types)

考虑可扩展性

函数和方法不应包含太多逻辑： 更推荐使用一些简短简单的方式，而不是使用几个大而复杂的方法。一个黄金准则是在方法有一个以上任务时尽快进行分割(参见 http://en.wikipedia.org/wiki/Single_responsibility_principle).

应避免在方法中硬编码业务逻辑，因此妨碍了轻易地通过子模块进行继承：

```
# 不要这么做
# 修改作用域或条件表示重载整个方法
def action(self):
    ...  # 长方法
    partners = self.env['res.partner'].search(complex_domain)
    emails = partners.filtered(lambda r: arbitrary_criteria).mapped('em
ail')

# 更好但也不要这么做
# 修改逻辑强制复制一部分代码
def action(self):
    ...
    partners = self._get_partners()
    emails = partners._get_emails()

# 更佳
# 最小化重载
def action(self):
    ...
    partners = self.env['res.partner'].search(self._get_partner_domain(
))
    emails = partners.filtered(lambda r: r._filter_partners()).mapped(
'email')
```

以上代码是出于示例是对可扩展函数进行，但必须考虑可读性并进行权衡。

同样相应地对函数命名：简短、适当命名的函数是可读/可维护代码和紧致文档的起点。

这一推荐对类、文件、模块和包也同样适用。(还可参见
http://en.wikipedia.org/wiki/Cyclomatic_complexity)

不要执行事务

Odoo框架负责为所有的RPC调用提供事务性上下文。原则是新的数据库游标在每个RPC调用的开始时打开，并在调用返回时、传送RPC客户端回复前执行，大概是这样：

```
def execute(self, db_name, uid, obj, method, *args, **kw):
    db, pool = pooler.get_db_and_pool(db_name)
    # 创建事务游标
    cr = db.cursor()
    try:
        res = pool.execute_cr(cr, uid, obj, method, *args, **kw)
        cr.commit() # 一切正常，执行
    except Exception:
        cr.rollback() # 错误，按原子性回滚所有内容
        raise
    finally:
        cr.close() # 保持关闭手动打开的游标
    return res
```

如果在RPC调用执行时发生任何错误，事务会自动进行原子级回滚，保留系统状态。

相似地，系统还在测试套装执行过程中提供独立的事务，因此它可以进行回滚或不依赖于服务端启动选项。

结果是如果手动在任何地方调用 `cr.commit()`，大机率会将系统分割成各种方式，因为你会产生部分提交，因而有部分和不利落的回滚，导致其它问题：

- 不连续的业务数据，通常会有数据丢失
- 工作流去同步，永久文档卡死
- 无法利落地回滚测试，并会开始污染数据库，以及导致错误 (即使在事务中未发生错误也会这样)

以下是一些非常简单的规则：
应**永不**自己调用call `cr.commit()` ，**除非**你显式地创建了自己的数据库游标！并非需要这么做的场景非常罕见!

顺便说一下如果你真的创建了自己的游标，那么需要处理错误用例及适当的回滚，以及在完成时恰当地关闭游标。

和通常认为的不同，你甚至不需要在以下场景中调用 `cr.commit()` ： - 在*models.Model* 对象的 `_auto_init()` 方法中：这由插件初始化方法处理，或者在创建自定义模型由ORM事务处理； - 在报表中: `commit()` 也由框架处理，你甚至可以在报表中更新数据库 - 在*models.Transient*方法中：这些方法和 *models.Model* ones的调用完全一致，在事务中及在结束处相应的 `cr.commit()/rollback()` - 等等(如果存在疑虑查看通用规则！)

此后服务端框架之外的所有 `cr.commit()` 调用必须有**显式注释**说明它们绝对有必要，为什么它们确实正确，以及为什么它们没有毁坏事务。否则可以并应当删除它们!

正确地使用翻译方法

Odoo使用了一个 GetText样式的方法，名为“下划线” `_( )` ，来代码中使用的静态字符串需要在运行时使用上下文的语言翻译。这种伪方法在代码中通过如下导入进行访问：

```
```

```
from odoo import _
```

在使用时必须遵循一些非常重要的规则来让其生效并避免使用一些无用的垃圾填充翻译。

基本上，这一方法应公用在代码中手动编写的静态字符串，它不会翻译字段值，如商品名等。这时必须要在相应字段上使用翻译标记。

规则非常简单：调用下划线方法应总是以 `_('literal string')` 的形式而不能是其它形式：

```
# 妥：普通字符串
error = _('This record is locked!')

# 妥：包含格式化模式的字符串
error = _('Record %s cannot be modified!') % record

# 也ok：多行字面量字符串
error = _("""This is a bad multiline example
             about record %s!""") % record
error = _('Record %s cannot be modified' \
          'after being validated!') % record

# 不妥：尝试在字符串格式化之后翻译
#       (注意括号!)
# 这<strong>不会</strong>起作用且会使翻译混乱!
error = _('Record %s cannot be modified!' % record)

# 不妥：动态字符串、字符串拼接等禁止使用!
# 这不会起作用且会使翻译混乱!
error = _("'" + que_rec['question'] + "' \n")

# 不妥：字段值会自动由框架翻译
# 这没有用且不会以你所认为的方式生效：
error = _("Product %s is out of stock!") % _(product.name)
# 并且以下当然不会生效，已进行过解释：
error = _("Product %s is out of stock!" % product.name)

# 不妥：字段值自动由框架翻译
# 这没有用且不会按你所认为的方式生效：
error = _("Product %s is not available!") % _(product.name)
# 并且以下当然不会生效，已进行过解释：
error = _("Product %s is not available!" % product.name)

# 取而代之你可以使用如下，这样所有内容都会翻译，
# 包含产品名，如果其字段定义相应地设置了翻译标记：
error = _("Product %s is not available!") % product.name
```

同时，记住翻译器要具有传递给下划线函数的字面量值才能生效，因此请尽量让它们更容易理解并保持杂散字符和格式化最小化。翻译器必须知道%s 或 %d这样的格式化模式，新行等需要被保留，但以有意义和明显地方式使用它们很重要：

```
# 不妥：让翻译难以处理
error = "'" + question + _("' \nPlease enter an integer value ")

# 更佳 (还要注意括号的位置!)
error = _("Answer to question %s is not valid.\n" \
          "Please enter an integer value.") % question
```

通常在Odoo中， 在操作字符串时更推荐使用 `%` 而非 `.format()` (在字符串中仅有一个变量供替换时)，且推荐使用 `%(varname)` 而非位置参数 (在有多个变量供替换时)。这会让社区翻译翻译者更易于翻译。

> **符号和惯例**

模型名 (使用点号标记符，前缀模块名)：

在定义Odoo模型时：使用名称的单数形式 (*res.partner* 和 *sale.order*来取代 *res.partnerS* 和 *saleS.orderS*)

在定义Odoo 临时模型 (向导)时： 使用 `<related_base_model>.<action>` ，其中 *related_base_model* 是与临时模型相关的基模型 (在 *models/*中定义)， *action* 是临时模型所做内容的短名称。避免使用 *wizard* 一词。例如： `account.invoice.make` , `project.task.delegate.batch` , ...

在定义*report* 模型(如SQL视图) 时: 按照临时模型规范使用 `<related_base_model>.report.<action>` 。

Odoo Python类 : 使用驼峰(面向对象样式)。

```
class AccountInvoice(models.Model):
    ...
```

变量名：

对模型变量使用驼峰

对普通变量使用下划线小写字母标记。

在包含记录id或id列表时对变量名使用后缀 *_id* 或 *_ids* 。不要使用 `partner_id` 来包含res.partner的记录

```
Partner = self.env['res.partner']
partners = Partner.browse(ids)
partner_id = partners[0].id
```

`One2Many` 和 `Many2Many` 字段应总是带有 *_ids* 作为后缀 (例: sale_order_line_ids)

`Many2One` 字段应带有 *_id* 作为后缀 (示例 : partner_id, user_id, ...)

方法规范

计算字段：计算方法的模式为 *_compute_<field_name>*

搜索方法：搜索方法的模式为 *_search_<field_name>*

默认方法：默认方法的模式为*_default_<field_name>*

选择方法：选择方法的模式为 *_selection_<field_name>*

Onchange方法 : onchange方法的模式为 *_onchange_<field_name>*

约束方法：约束方法的模式为 *_check_<constraint_name>*

动作方法：对象动作方法的前缀为*action_*。因其仅使用一条记录，在方法的开始处添加`self.ensure_one()` 。

模型中属性排序应为

私有属性 (`_name` , `_description` , `_inherit` , ...)

默认方法和 `_default_get`

字段声明

计算、后向和搜索方法的排序与声明顺序相同

选择方法 (用于返回针对选择字段计算值的方法)

约束方法 (`@api.constrains`) 和 onchange方法 (`@api.onchange`)

CRUD方法 (ORM 重载)

s动作方法

最后是其它业务方法。

```
class Event(models.Model):
    # 私有属性
    _name = 'event.event'
    _description = 'Event'

    # 默认方法
    def _default_name(self):
        ...

    # 字段声明
    name = fields.Char(string='Name', default=_default_name)
    seats_reserved = fields.Integer(oldname='register_current', string=
'Reserved Seats',
        store=True, readonly=True, compute='_compute_seats')
    seats_available = fields.Integer(oldname='register_avail', string=
'Available Seats',
        store=True, readonly=True, compute='_compute_seats')
    price = fields.Integer(string='Price')
    event_type = fields.Selection(string="Type", selection='_selection_
type')

    # 计算和搜索字段，和字段声明的顺序一样
    @api.depends('seats_max', 'registration_ids.state', 'registration_i
ds.nb_register')
    def _compute_seats(self):
        ...

    @api.model
    def _selection_type(self):
        return []

    # 约束和onchange
    @api.constrains('seats_max', 'seats_available')
    def _check_seats_limit(self):
        ...

    @api.onchange('date_begin')
    def _onchange_date_begin(self):
        ...

    # CRUD方法(和 name_get, name_search, ...) 重载
    def create(self, values):
        ...

    # 动作方法
    def action_validate(self):
        self.ensure_one()
        ...

    # 业务方法
    def mail_user_confirm(self):
        ...
```

Javascript和CSS

静态文件组织

Odoo插件具有架构各个文件的一些规范。我们这讲解如何组织网页资源的更多详情。

第一件事是要知道Odoo服务会对位于*static/*文件夹中的所有文件（静态地）提供服务， 但会使用插件名前缀。例如，如果文件位于 *addons/web/static/src/js/some_file.js*，那就可以通过 url *your-odoo-server.com/web/static/src/js/some_file.js*来静态获取

规范为根据如下结构组织代码：

static: 通用的所有静态文件

static/lib: 这是js库应该处于的位置，位于子文件夹中。 因此，例如， *jquery* 库中的所有文件位于 *addons/web/static/lib/jquery*中

static/src: 通用静态源代码文件夹

static/src/css: 所有css文件

static/src/fonts

static/src/img

static/src/js

static/src/js/tours: 终端用户导览文件 (教程，非测试)

static/src/scss: scss 文件

static/src/xml: 所有会在JS中渲染的qweb模板

static/tests: 这里放置所胡测试相关的文件。

static/tests/tours: 这里放置所有导览测试文件 (非教程)。

Javascript编码指南

`use strict;` 推荐在所有javascript文件中使用

使用一种linter (jshint, ...)

永远不要添加最小化混淆 Javascript库

对类声明使用驼峰名

更精确的JS指南在github wiki中进行详细讲解。还可以通过查看Javascript手册来查看Javascript中已有的API。

CSS编码指南

对所有类添加 *o_ <module_name>* 前缀，其中*module_name* 是模块的技术名称(‘sale’ ，‘im_chat’ , ...) 或模块所保留的主路由(主要针对website模块，如 ‘o_forum’ 对应 *website_forum* 模块)。 这一规则的唯一例外是网页客户端：它仅使用*o_* 前缀。

避免使用*id* 标签

使用Bootstrap原生类

使用下划线小写字母标记来命名类

Git

配置 git

根据既往体验及口述传统，如下内容经过了很长时间沉淀来让提供更为有益:

确保在本地git中同时定义user.email 和 user.name

```
git config --global <var> <value>
```

确保在Github个人资料中添加完整姓。请自由添加团队名、头像、喜爱的名言等等

提交信息结构

提交信息有4个部分： 标签、模块、短描述和完整描述。尽量让你的消息遵循如下推荐结构

```
[TAG] module: 以短名描述修改 (理想情况为 < 50 个字符)

修改描述的长版本，包括修改的基本原因，
或引入功能的概述。

请花费更多的时间在描述所做修改的原因
而不是修改的内容。通常修改内容可通过阅读
diff来了解。仅在有技术选择或决策时才说明内容。
这时说明为什么做出这一决策。

使用参考结束消息，如任务或bug号、PR号,及
OPW工单, 遵循所推荐的格式:
task-123 (与任务相关)
Fixes #123  (在Github上关闭相关的issue)
Closes #123  (在Github上关闭相关的PR)
opw-123 (相关工单)
```

标签和模块名

标签用作提交的前缀。应当为如下内容之一

[FIX] 用于bug修复：多用于稳定版本但在开发版本中修复近期漏洞也同样有效;

[REF] 用于重构：在功能进行重度重写时;

[ADD] 用于添加新模块;

[REM] 用于删除资源：删除无用代码、删除视图、删除模块, ...;

[REV] 用于恢复提交：如果提交导致问题或不需要使用这一标签进行恢复;

[MOV] 用于移动文件：使用git move并且不修改所移动文件的内容，否则Git 会丢失文件的追踪和历史信息；也在将代码从一个文件移动到另一个文件时使用;

[REL] 用于发行版提交；新的主版本或小稳定版本;

[IMP] 用于改进：大部分在开发版本中做的修改是与其它标签无关的增量改进;

[MERGE] 用于合并提交：在漏洞修改的包转发中使用，但在用作包含一些分离提交的功能的提交;

[CLA] 用于签署Odoo个人贡献者证书；

[I18N] 用于在翻译文件中的修改；

标签后为修改的模块名。使用技术名称作为函数名称可能会随时间改变。如果修改了一些模块，列举它们或使用多个名称来说明它是跨模块的。除非实在必要或更为简单避免在同一次提交中对多个模块修改代码。掌握模块的历史修改可能会变得困难。

提交信息头

在标签和模块名之后有一个有意义的提交信息头。 应当有清晰的含义并包含修改的原因。不要使用“bugfix” 或 “improvements”这样的单个单词。尽量限制头部长度为约50个字符来保持可读性。

提交消息头在拼接了`if applied, this commit will <header>`时应该会成为有效的句子。例如`[IMP] base: prevent to archive users linked to active partners`是正确的，因其成为了有效的兔子`if applied, this commit will prevent users to archive...`.

提交信息完整描述

在消息描述中指明修改所影响的代码部分 (模块名, lib, 切面对象, ...) 及修改的描述。

首先说明**为什么**要修改代码。对于数十年（或3天）后回顾你的提交的人最重要的是你为什么这么做。这是修改的目的。

所做的修改可在提交本身中看到。如果包含一些技术选择，在提交信息的原因后进行说明也是一个好主意。对于 Odoo R&D 开发人员来说“PO 团队要求我这么做”并不是有效原因。

请避免同时进行影响多个模块的提交。尽量将影响不同的模块分割为不同的提交。如果需要对给定模块的修改进行恢复的话这会很有用。

在略显啰嗦时不要犹豫。大部分人只会看提交信息来根据几句话来判断你所做的所有修改。不要有任何压力。

你对功能花费了几个小时、几天或几周。花一点时间来平静下来，编写清晰、易于理解的提交信息。

如果你是Odoo R&D 开发人员，原因应当是你所处理的任务的用途。完成的详情是提交信息的核心部分。**如果你执行的任务缺乏目的性和确定性请考虑在继续之前让其变得更为清晰。**

最后以下是一些正确提交信息的示例：

```
[REF] models: 使用 `parent_path` 来实现 parent_store

 这会替换此前修改的预排序树状遍历(MPTT) 为
 字段 `parent_left`/`parent_right`[...]

[FIX] account: 删除 frenglish

 [...]

 关闭 #22793
 修改 #22769

[FIX] website: 删除未使用的alert div, 修复input-group-btn的样式

 Bootstrap的 CSS根据input-group-btn元素
 成为其父级的第一个/最后一个子级。
 对于不显示的或无用警告情况并非如此。
```

> 使用长描述来说明*为什么*而非*是什么*，*是什么*在diff中可以看到