Odoo中文社区（odoo.net.cn）开发系列丛书

Odoo 10中文开发指南

快速更新你的开发技能，构建强大的Odoo 10业务应用程序

上海-老杨（杨浔波） **著**

QQ：291525012
E-mail：yangxunbo@qq.com
Wechat：yangxunbo1986

优秀靠谱的odoo服务及咨询公司名单列表：

上海先安信息科技有限公司
上海开阖软件有限公司
造源信息科技（上海）有限公司
上海龙坤信息科技有限公司
上海寰享网络科技公司
上海卓忆科技发展有限公司
上海亘盈信息技术有限公司
青岛欧度软件技术有限责任公司
昆山一百计算机有限公司信息
广州尚鹏信息科技有限公司

前言

Odoo是一个强大的商业应用开源平台。在此基础上，构建了一套紧密集成的应用程序，涵盖了从CRM到销售到股票和会计的所有业务领域。Odoo有一个动态和不断增长的社区，不断增加功能、连接器和其他商业应用。

Odoo 10开发要点提供了一个逐步指导Odoo开发的指南，让读者能够快速的爬上学习曲线，并在Odoo应用平台上变得富有成效。

前两章的目的是让读者熟悉Odoo，学习建立开发环境的基本技术，熟悉模块开发方法和工作流。

以下各章节详细解释了Odoo addon模块开发所需的关键开发主题，如继承和扩展、数据文件、模型、视图、业务逻辑等等。

最后，最后一章解释了在部署Odoo实例时应该考虑什么。

本书教学大纲

第1章，开始了Odoo开发，从开发环境的设置开始，从源代码安装Odoo，并学习如何管理Odoo服务器实例。

第2章，构建您的第一个Odoo应用程序，指导我们创建第一个Odoo模块，涵盖涉及的所有不同层:模型、视图和业务逻辑。

第3章，继承——扩展现有的应用程序，解释现有的继承机制，以及如何使用它们创建扩展模块，在其他现有模块上添加或修改功能。

第4章，模块数据，包括最常用的Odoo数据文件格式(XML和CSV)，外部标识符概念，以及如何在模块和数据导入/导出中使用数据文件。

第5章，模型构建应用程序数据，详细讨论模型层，使用模型和字段的类型，包括关系和计算字段。

第6章，视图——设计用户界面，包括视图层，详细解释了几种类型的视图以及可以用来创建动态和直观的用户界面的所有元素。

第7章，ORM应用程序逻辑——支持业务流程，在服务器端引入编程业务逻辑，探索ORM概念和特性，并解释如何使用向导进行更复杂的用户交互。

第8章，编写测试和调试代码，讨论如何向addon模块添加自动化测试，以及调试模块业务逻辑的技术。

第9章，QWeb和看板视图，通过Odoo QWeb模板，使用它创建丰富的看板。

第10章，创建QWeb报告，讨论使用基于QWeb的报告引擎，以及生成友好的PDF报告所需要的一切。

第11章，创建网站前端功能，介绍了Odoo网站开发，包括web控制器实现和使用QWeb模板构建前端web页面。

第l2章 ,外部API——与其他系统集成 ,解释了如何从外部应用程序中使用Odoo服务器逻辑，并引入了一个受欢迎的客户端编程库，也可以作为命令行客户端使用。

第l3章，部署清单——现场直播，向我们展示了如何为生产黄金时间准备一个服务器，解释应该注意哪些配置，以及如何配置Nginx反向代理以提高安全性和可伸缩性。

本书的环境基础

我们将在Ubuntu或Debian系统上安装我们的Odoo服务器，但我们希望您使用您的操作系统和编程工具，无论是Windows、Mac还是其他。

我们将提供一些关于在Ubuntu服务器上设置虚拟机的指导。您应该选择使用的虚拟化软件，例如VirtualBox或VMWare Player;两者都是免费的。如果您使用的是Ubuntu或Debian工作站，则不需要虚拟机。

正如您已经指出的，我们的Odoo安装将使用Linux，因此我们将不可避免地使用命令行。但是，你应该能够按照所给的指令行事，即使不熟悉它。

预期Python编程语言的基本知识。如果你不喜欢它，我们建议你学习快速教程，让你开始。我们还将使用XML，因此我们希望熟悉标记语法。

本书面向对象

这本书的目标是开发人员，他们有开发商业应用程序的经验，他们愿意快速地用Odoo来生产。

读者应该了解MVC应用程序设计和Python编程语言的知识。熟悉web技术、HTML、CSS和JavaScript也会有所帮助。

示例

在这本书中，你会发现许多不同种类的信息的文本样式。以下是这些风格的一些例子，以及它们的含义的解释。

文本中的代码单词、数据库表名、文件夹名称、文件名、文件扩展名、路径名、虚拟url、用户输入和Twitter句柄如下:文本中的代码字如下所示:“创建一个新的数据库，使用createdb命令。”

代码块设置如下:

```
@api.multi
def do_toggle_done(self):
    for task in self:
        task.is_done = not task.is_done
    return True
```

当我们希望将您的注意力吸引到代码块的某个特定部分时，相关的行或项以粗体设置:

```
@api.multi
def do_toggle_done(self):
    for task in self:
        task.is_done = not task.is_done
    return True
```

任何命令行输入或输出如下:

```
$ ~/odoo-dev/odoo/odoo-bin.py -d demo
```

新的术语和**重要的词**用粗体显示。例如，在屏幕上、菜单或对话框中看到的单词出现在这样的文本中:“在登录时，你会看到**Apps**菜单，显示可用的应用程序。”

警告或重要的音符出现在这样的盒子里。

提示和技巧就像这样。

1

开始使用Odoo开发

在进入Odoo开发之前，我们需要建立我们的开发环境，并学习它的基本管理任务。

在本章中，我们将学习如何设置工作环境，在这里我们将构建我们的Odoo应用程序。我们将学习如何设置Debian或Ubuntu系统来托管开发服务器实例，以及如何从GitHub源代码中安装Odoo。然后，我们将学习如何设置与Samba的文件共享，这将允许我们从运行Windows或任何其他操作系统的工作站运行Odoo文件。

Odoo是使用Python编程语言构建的，它使用PostgreSQL数据库进行数据存储;这些是Odoo主机的两个主要需求。要从源代码运行Odoo，我们首先需要安装它依赖的Python库。然后可以从GitHub下载Odoo源代码。虽然我们可以下载ZIP文件或tarball，但我们会看到，如果我们使用Git版本控制应用程序获取源代码会更好;它也会帮助我们把它安装在我们的Odoo主机上。

为Odoo服务器设置一个主机

一个Debian / Ubuntu系统被推荐用于Odoo服务器。你仍然可以在你最喜欢的桌面系统中工作，无论是Windows、Mac还是Linux。

Odoo可以在各种操作系统上运行，那么为什么要以牺牲其他操作系统为代价来选择Debian呢?因为Debian被认为是Odoo团队的参考部署平台;它有最好的支持。如果我们使用Debian / Ubuntu，它将更容易找到帮助和额外的资源。

它也是大多数开发人员工作的平台，大多数部署都是在这个平台上进行的。因此，不可避免的是，Odoo开发人员将会对Debian / Ubuntu平台感到满意。即使你是Windows背景的，你也要对它有所了解，这一点很重要。

在本章中，您将学习如何在基于debianbased的系统上设置和处理Odoo，只使用命令行。对于那些有Windows系统的家庭，我们将介绍如何设置虚拟机来托管Odoo服务器。作为一个额外的奖励，您将在这里学到的技术也将允许您在云服务器中管理Odoo，在那里您唯一的访问将通过**Secure Shell (SSH)**来访问。

请记住，这些指示是为了建立一个新的发展系统。如果您想在现有的系统中尝试其中的一些，总是提前进行备份，以便在出现问题时恢复它。

为Debian主机提供的服务

如前所述，我们需要一个基于debian-based的Odoo服务器主机。如果这是您第一次使用Linux，您可能会注意到Ubuntu是基于debianbased的Linux发行版，所以它们非常相似。

Odoo可以保证使用当前稳定版本的Debian或Ubuntu。在写作的时候，这些是Debian 8“Jessie”和Ubuntu 16.04.1 LTS(Xenial Xerus)。这两环境都有Python 2.7，这是运行Odoo的必要条件。值得一提的是，Odoo并不支持Python 3，因此需要Python 2。

如果你已经在运行Ubuntu或另一个基于debian-based的发行版，你就可以设置;这也可以作为Odoo的主机。
对于Windows和Mac操作系统，安装Python、PostgreSQL和所有依赖项;接下来，直接从源程序运行Odoo。然而 ,这可能是一个挑战 ,所以我们的建议是使用运行Debian或Ubuntu服务器的虚拟机。您可以选择您喜欢的虚拟化软件，以在虚拟机中获得一个工作的Debian系统。

如果您需要一些指导 ,这里有一些关于虚拟化软件的建议。有几个选项 ,比如Microsoft hyper - v(在某些版本的Windows系统中可用)、Oracle VirtualBox和VMWare工作站播放器(Mac的VMWare Fusion)。VMWare工作站的球员可能是更容易使用,并且免费下载可以在https://my.vmware.com/web/vmware/downloads上找到。

对于使用的Linux映像，安装Ubuntu服务器要比Debian更加友好。如果您从Linux开始，我建议您尝试使用现成的映像。TurnKey Linux提供了多种格式的易于使用的预安装映像 ,包括ISO。ISO格式将与您所选择的任何虚拟化软件一起工作，即使是在您可能拥有的裸金属机器上。一个很好的选择可能是第三方LAPP镜像,包括Python和PostgreSQL,可以在`http://www.turnkeylinux.org/lapp`找到。

一旦安装并启动，您应该能够登录到命令行shell。

为Odoo创建一个用户帐户

如果您正在登录使用超级用户`root`帐户，那么您的第一个任务应该是创建一个正常的用户帐户来使用您的工作，因为它被认为是不好的工作实践作为`root`。特别是，如果您将其作为`root`来启动，那么Odoo服务器将拒绝运行。

如果您正在使用Ubuntu，那么您可能不需要这个，因为安装过程必须已经通过创建一个用户来指导您。

首先，确保安装sudo。我们的工作用户将需要它。如果作为root登录，执行以下命令:

```
# apt-get update && apt-get upgrade  # 安装系统更新

# apt-get install sudo  # 确保安装“sudo”
```

下一组命令将创建一个odoo用户：

```
# useradd -m -g sudo -s /bin/bash odoo  # 创建一个具有sudo能力的“odoo”用户

# passwd odoo # 请求并为新用户设置密码
```

你可以将`odoo`转换为你想要的任何用户名。`-m`选项确保创建其主目录。`-g sudo`选项将它添加到*sudoers*列表中，以便它可以作为`root`运行命令。`-s /bin/bash`选项将默认的shell设置为bash，这比默认的`sh`要好。
现在我们可以作为新用户登录，并设置Odoo。

从源程序中安装Odoo

可以在`nightly.odoo.com`,上找到现成的Odoo软件包，如Windows(`.exe`)、Debian(`.deb`)、CentOS(`.rpm`)和源代码tarballs(`.tar.gz`)。

作为开发人员，我们希望直接从GitHub存储库中安装它们。这将使我们对版本和更新有更多的控制。

为了保持整洁，在我们的主目录home内建立一个 `/odoo-dev`子目录以便进行工作。

在整本书中，我们假设`/odoo-dev` 是您的Odoo服务器安装的目录。

首先，确保您已经登录为我们现在或在安装过程中创建的用户，而不是作为`root`用户。
假设您的用户是`odoo`，请使用以下命令确认:

```
$ whoami
odoo
$ echo $HOME
/home/odoo
```

现在我们可以使用这个脚本了。它向我们展示了如何将Odoo从源代码安装到Debian / Ubuntu系统中。

首先，安装基本的依赖项，以使我们开始:

```
$ sudo apt-get update && sudo apt-get upgrade  #安装系统更新
$ sudo apt-get install git  # 安装Git
$ sudo apt-get install npm  # 安装NodeJs及其包管理器
$ sudo ln -s /usr/bin/nodejs /usr/bin/node  # 调用节点运行nodejs
$ sudo npm install -g less less-plugin-clean-css  #安装less编译器
```

从版本9.0开始，Odoo web客户端需要在系统中安装`less` CSS预处理器，以便正确地呈现web页面。要安装这个，我们需要节点。Node.js和npm。

接下来，我们需要获得Odoo源代码并安装它的所有依赖项。Odoo源代码包括一个实用脚本，在`odoo/setup/`目录中，帮助我们在Debian / Ubuntu系统中安装所需的依赖项:

```
$ mkdir ~/odoo-dev  # Create a directory to work in
$ cd ~/odoo-dev  # Go into our work directory
$ git clone https://github.com/odoo/odoo.git -b 10.0 --depth=1  # Get Odoo
source code
$ ./odoo/setup/setup_dev.py setup_deps  # Installs Odoo system dependencies
$ ./odoo/setup/setup_dev.py setup_pg  # Installs PostgreSQL & db superuser
for unix user
```

最后，Odoo应该准备好使用。~符号是我们的主目录(例如，`/home/odoo`)的快捷方式。`git -b 10.0`选项告诉Git明确下载Odoo的10.0分支。在写的时候，这是多余的，因为10.0是默认的分支;然而，这可能会改变，因此它可能使脚本成为未来的证明。`--depth=1`选项告诉Git只下载最后一个版本，而不是完整的变更历史，使下载变得更小更快。

要启动一个Odoo服务器实例，只需运行:

```
$ ~/odoo-dev/odoo/odoo-bin
```

在Odoo 10中，在以前的版本中`odoo.py`脚本被`odoo-bin`替换，用于启动服务器。

在默认情况下，Odoo实例侦听端口8069，因此如果我们将浏览器指向`http://<server-address>:8069`，我们将到达这些实例。当我们第一次访问它时，它向我们展示了一个创建新数据库的助手，如下面的截图所示:

作为开发人员，我们需要使用几个数据库，因此从命令行创建它们更方便，因此我们将学习如何做到这一点。现在在终端按*Ctrl + C*停止Odoo服务器并返回命令提示符。

初始化一个新的Odoo数据库

为了能够创建一个新的数据库，您的用户必须是一个PostgreSQL超级用户。下面的命令为当前的Unix用户创建一个PostgreSQL超级用户:

```
$ sudo createuser --superuser $(whoami)
```

要创建一个新的数据库，请使用`createdb`命令。让我们创建一个`demo`数据库:

```
$ createdb demo
```

要使用Odoo数据模式初始化该数据库，我们应该使用`-d`选项在空数据库上运行Odoo:

```
$ ~/odoo-dev/odoo/odoo-bin -d demo
```

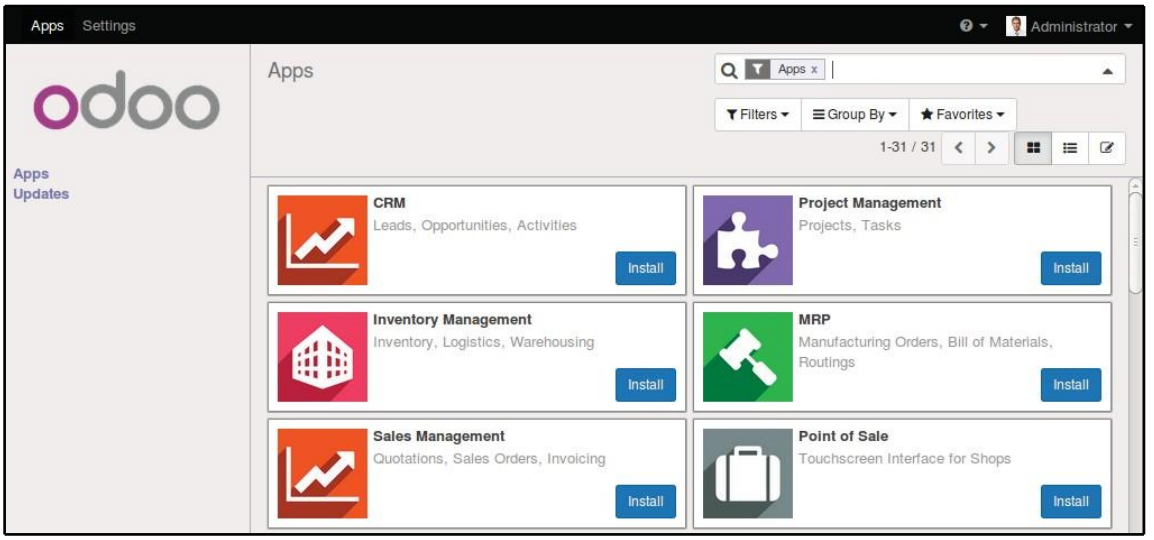
这将花费几分钟来初始化一个`demo`数据库，它将以一个信息日志消息结束，**Modules loaded**。

注意，它可能不是最后一个日志消息，它可以在最后三到四行。这样，服务器就可以准备好侦听客户端请求了。

默认情况下，这将用演示数据初始化数据库，这通常对开发数据库非常有用。若要初始化没有演示数据的数据库`--without-demo-data=all`。

现在打开`http://<server-name>:8069`，你的浏览器会被显示在登录屏幕上。如果您不知道您的服务器名称，在终端中键入`hostname`命令，以便找到它或`ifconfig`命令来查找IP地址。
如果您在虚拟机中托管Odoo，您可能需要设置一些网络配置，以便能够从主机系统访问它。最简单的解决方案是将虚拟机网络类型从NAT改为桥接。这样，客户虚拟机就不会共享主机IP地址，而是拥有自己的IP地址。也可以使用NAT，但这需要您配置端口转发，这样您的系统就知道一些端口，比如8069，应该由虚拟机来处理。如果您遇到麻烦，希望这些细节将帮助您在您所选择的虚拟化软件的文档中找到相关信息。

默认管理员帐户是`admin`，其密码`admin`。登录后，你会看到**Apps**菜单，显示可用的应用程序:



当您想要停止Odoo服务器实例并返回到命令行时，请在bash提示符中按*Ctrl* + *C*。按下向上箭头键会给你带来先前的shell命令，所以这是一个快速启动Odoo的方法，同样的选项。按下*Ctrl* + *C*键和向上的箭头键，*Enter*是一个经常使用的组合，在开发期间重新启动Odoo服务器。

管理数据库

我们已经了解了如何从命令行创建和初始化新的Odoo数据库。有更多的命令值得管理数据库。

我们已经知道如何使用`createdb`命令创建空的数据库，但是它也可以通过复制现有的数据库创建一个新的数据库`--template`选项
确保您的Odoo实例被停止，并且您没有打开的其他连接我们刚刚创建的`demo`数据库，然后运行这个：

```
$ createdb --template=demo demo-test
```

实际上，每次创建数据库时，都会使用模板。如果没有指定，则使用预定义的`templatel`。

要列出系统中的现有数据库，可以使用`-l`选项使用PostgreSQL psql实用程序:

```
$ psql -l
```

运行它将列出我们迄今为止创建的两个数据库: `demo`和`demo-test`。该列表还将显示每个数据库中使用的编码。默认值是UTF-8，这是Odoo数据库所需的编码。
要删除不再需要的数据库(或者需要重新创建)来使用`dropdb`命令:

```
$ dropdb demo-test
```

现在您知道了使用数据库的基础知识。了解更多关于PostgreSQL,请参考官方文档:`http://www.postgresql.org/docs/`.

WARNING:
删除数据库命令将不可挽回地破坏您的数据。使用此命令时要小心，并且在使用此命令之前，总是要对重要的数据库进行备份。

一个关于Odoo产品版本

在写作的时候，Odoo的最新稳定版本是10版本，在GitHub上以10.0的形式标注。这是我们将在本书中使用的版本。

值得注意的是，Odoo数据库在Odoo主要版本之间不兼容。这意味着，如果您在一个以前的主要版本的Odoo创建的数据库上运行一个Odoo 10服务器，那么它就不会起作用。在使用该产品的后续版本之前，需要在数据库中使用非琐碎的迁移工作。

对于addon模块也是如此:作为一个通用规则，为Odoo主版本开发的addon模块将不会与其他版本一起工作。当从web下载一个社区模块时，确保它针对您正在使用的Odoo版本。

另一方面，主要的发行版(9.0,10.0)预计会收到频繁的更新，但这些更新大部分应该是bug修复。它们被保证为“API稳定”，意味着模型数据结构和视图元素标识符将保持稳定。这很重要，因为它意味着上游核心模块中不兼容的更改将不会导致自定义模块的破坏。

请注意，`master`分支中的版本将导致下一个主要的稳定版本，但在此之前，它不是“API稳定”，您不应该使用它来构建自定义模块。这样做就像在流沙上移动:你不能确定什么时候会引入一些改变，这会破坏你的定制模块。

更多的服务器配置选项

Odoo服务器支持相当多的其他选项。我们可以查看所有可用选项`--help`:

```
$ ./odoo-bin --help
```

我们将在以下部分回顾一些最重要的选项。让我们从查看当前活动选项如何保存在配置文件中开始。

Odoo 服务器配置文件

大多数选项都可以保存在配置文件中。默认情况下，Odoo将使用`.odoorc`文件在您的主目录。在Linux系统中，它的默认位置是在`home` (`$HOME`)中，在Windows发行版中，它与用于启动Odoo的可执行文件处于同一目录。

在以前的Odoo / OpenERP版本中，缺省配置文件的名称为`.openerp-serverrc`。对于向后兼容性，Odoo 10仍然会使用它，如果它现在没有发现`.odoorc`文件。

在一个干净的安装上`.odoorc`配置文件不是自动创建的。我们应该使用`--save`选项创建默认配置文件，如果它还不存在，并将当前的实例配置存储到其中:

```
$ ~/odoo-dev/odoo/odoo-bin --save --stop-after-init  #保存配置文件
```

在这里，我们还使用了`--stop-after-init`选项，在服务器完成其操作后停止服务器。在运行测试或要求运行模块升级以检查安装是否正确时，通常使用此选项。

现在我们可以检查在这个默认配置文件中保存的内容:

```
$ more ~/.odoorc  # 显示配置文件
```

这将显示所有可使用其默认值的配置选项。在下一次启动Odoo实例时，编辑它们将是有效的。键入q退出并返回到提示符。

我们还可以选择使用一个特定的配置文件，使用`--conf=<filepath>`选项。配置文件不需要您刚才看到的所有选项。只有那些真正改变了默认值的才需要在那里。

改变监听端口

`--xmlrpc-port=<port>`命令选项允许我们更改服务器实例的监听端口，从默认的8069。这可以用于在同一台机器上同时运行多个实例。

让我们试试这个。打开两个终端窗口。首先，运行这个:

```
$ ~/odoo-dev/odoo/odoo-bin --xmlrpc-port=8070
```

在第二个终端上运行以下命令:

```
$ ~/odoo-dev/odoo/odoo-bin --xmlrpc-port=8071
```

在这里，两个Odoo实例在同一个服务器上监听不同的端口!这两个实例可以使用相同或不同的数据库，这取决于所使用的配置参数。这两个版本可以运行相同或不同版本的Odoo。

数据库过滤选项

与Odoo一起开发时，它经常与几个数据库一起工作，有时甚至使用不同的Odoo版本。在同一个端口上停止和启动不同的服务器实例，并在不同的数据库之间切换，会导致web客户端会话的行为不正确。

使用在私有模式下运行的浏览器窗口访问我们的实例可以帮助避免这些问题。

另一个好的做法是在服务器实例上启用数据库过滤器 ,以确保它只允许我们想要处理的数据库的请求，而忽略所有其他的。这是用`--db-filter`选项完成的。它接受一个正则表达式作为有效数据库名称的过滤器。匹配一个确切的名字,表达应该开始`^`和结束`$`。

例如，为了只允许`demo`数据库，我们将使用这个命令:

```
$ ~/odoo-dev/odoo/odoo-bin --db-filter=^demo$
```

管理服务器日志消息

`--log-level` 选项允许我们设置log verbosity。这对于了解服务器上正在发生的事情非常有用。例如，要启用调试日志级别，请使用`--log-level=debug` 选项。

以下日志级别特别有趣：

`debug_sql` 检查服务器生成的SQL查询

`debug_rpc` 详细说明服务器接收到的请求

`debug_rpc_answer`详细说明服务器发送的响应

默认情况下，日志输出被定向到标准输出(您的控制台屏幕)，但是它可以直接指向一个日志文件，其中包含`--logfile=<filepath>`选项。

最后，当一个异常被提起时，`--dev=all`选项将打开Python调试器(`pdb`)。对服务器错误进行事后分析是很有用的。注意，它对logger verbosity没有任何影响。更多细节在Python调试器命令可以在`https://docs.python.org/2/library/pdb.html#debugger-commands`找到调试器命令。

从您的工作站

您可能正在使用一个Debian / Ubuntu系统在本地虚拟机或网络服务器上运行Odoo。但您可能更喜欢在您的个人工作站使用您喜欢的文本编辑器或IDE进行开发工作。对于Windows工作站的开发人员来说，这可能是常见的情况。但对于那些需要在本地网络上使用Odoo服务器的Linux用户来说，情况也是如此。

解决这个问题的方法是在Odoo主机中启用文件共享，这样就可以方便地从我们的工作站编辑文件。对于Odoo服务器操作，比如服务器重启，我们可以在我们最喜欢的编辑器旁边使用SSH shell(比如在Windows上的PuTTY)。

使用Linux文本编辑器

迟早，我们需要从shell命令行编辑文件。在许多Debian系统中，默认的文本编辑器是vi . 如果您对它不满意，您可能可以使用更友好的选择。在Ubuntu系统中，默认的文本编辑器是nano。你可能更喜欢它，因为它更容易使用。如果您的服务器上没有可用的，可以安装:

```
$ sudo apt-get install nano
```

在下面的章节中，我们将假设nano是首选的编辑器。如果您喜欢其他的编辑器，请随意调整相应的命令。

安装和配置Samba

Samba服务帮助使Linux文件共享服务与Microsoft Windows系统兼容。我们可以在Debian / Ubuntu服务器上安装这个命令:

```
$ sudo apt-get install samba samba-common-bin
```

samba包安装了文件共享服务，`smbpasswd`工具需要使用`samba-common-bin`包。默认情况下，允许访问共享文件的用户需要注册。我们需要注册我们的用户`odoo`，并为其文件共享访问设置密码：

```
$ sudo smbpasswd -a odoo
```

在此之后，我们将被要求使用一个密码来访问共享目录，而`odoo`用户将能够访问其主目录的共享文件，尽管它只会被读取。我们想要有写访问权限，因此我们需要编辑Samba配置文件，以更改以下内容:

```
$ sudo nano /etc/samba/smb.conf
```

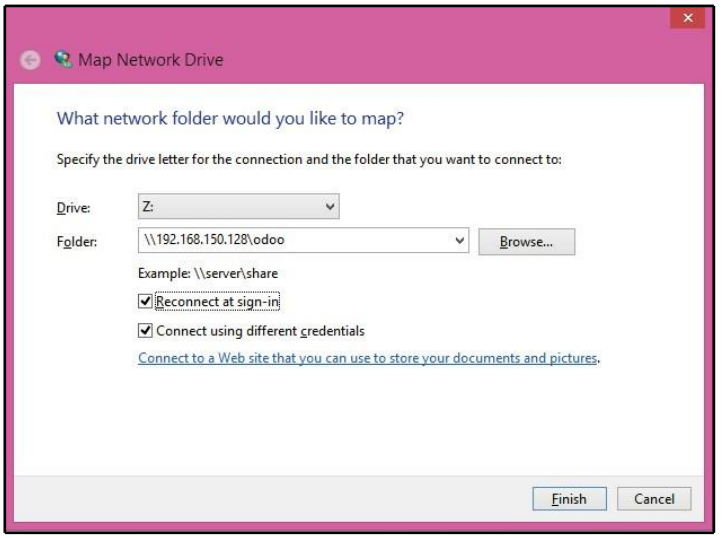
在配置文件中，查找`[homes]`部分。编辑其配置行，使其与以下设置匹配:

```
[homes]
    comment = Home Directories
    browseable = yes
    read only = no
    create mask = 0640
    directory mask = 0750
```

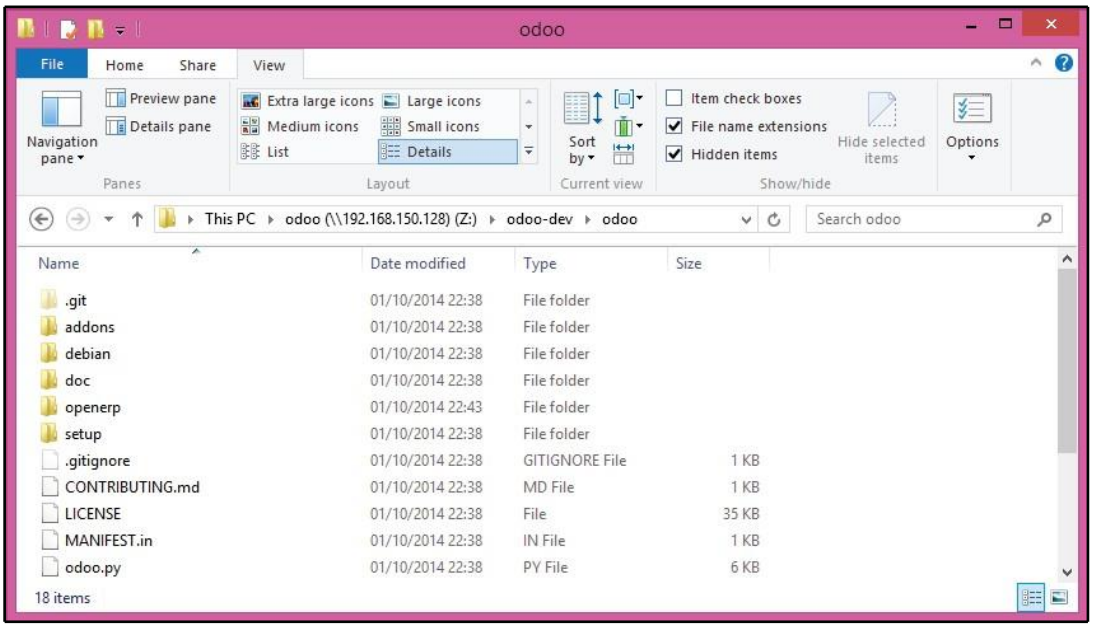
对于配置更改生效，重新启动服务:

```
$ sudo /etc/init.d/smbd restart
```

要从Windows中访问这些文件，我们可以使用与`smbpasswd`定义的特定的用户名和密码，将网络驱动器映射为`\\<my- server-name>\odoo`。当试图与`odoo`用户登录时，您可能会遇到Windows向用户名(例如`MYPC\odoo`)添加计算机域的问题。为了避免这种情况，请在登录时使用一个空域(例如，`\odoo`):



如果我们现在使用Windows资源管理器打开映射驱动器 我们将能够访问和编辑`odoo`用户的主目录的内容:

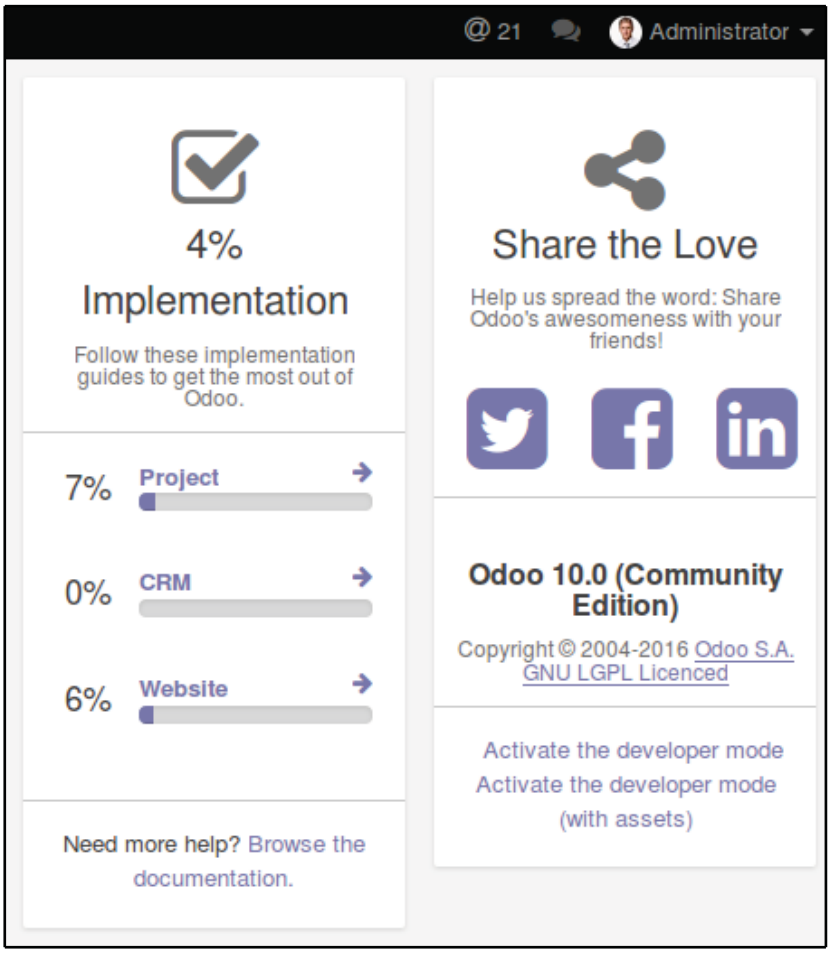


Odoo包括一些对开发人员非常有用的工具，我们将在本书中使用它们。它们是技术特性和开发模式。默认情况下，这些都是禁用的，所以这是一个学习如何启用它们的好时机。

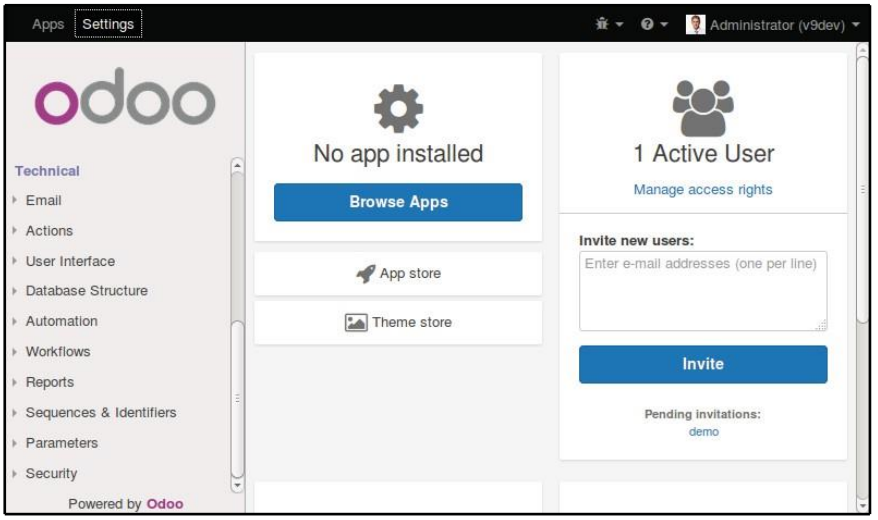
激活开发工具

开发工具提供高级服务器配置和特性。其中包括顶部菜单栏中的一个调试菜单，以及**Settings**菜单中的其他菜单选项，特别是**Technical**菜单。

这些工具在默认情况下是禁用的，并且为了启用它们，我们需要以admin身份登录。在顶部菜单栏中，选择**Settings**菜单。在底部右侧，在Odoo版本之下，您将找到两个选项来启用开发者模式;它们中的任何一个都可以**Debug**和**Technical**。第二个选项，**Activate the developer mode (with assets)**，也禁用了web客户端使用的JavaScript和CSS，这样可以更容易地调试客户端行为:



在此之后，页面被重新加载，您应该在顶部菜单栏中看到一个bug图标，就在会话用户头像和提供调试模式选项的名称之前。在顶部菜单的**Settings**选项中，我们应该看到一个新的**Technical**菜单部分，可以访问许多Odoo实例内部:



Technical菜单选项允许我们检查和编辑存储在数据库中的所有Odoo配置，从用户界面到安全性和其他系统参数。在这本书中，你将会学到更多。

安装第三方模块

在一个Odoo实例中提供新的模块，这样它们就可以安装，这是一个新手经常会感到困惑的东西。但不一定非得如此，让我们来揭开它的神秘面纱吧。

Finding community modules

互联网上有很多的Odoo模块。`apps.odoo.com`是一个可以在你的系统上下载和安装的模块目录。**Odoo Community Association (OCA)**协调社区贡献，并在`https://github.com/OCA/`上维护了GitHub上的一些模块存储库。

要在Odoo安装中添加一个模块，我们可以将它复制到`addons`目录中，并与官方模块一起使用。在我们的案例中，`addons`目录位于`~/odoo- dev/odoo/addons/`。对于我们来说，这可能不是最好的选择，因为我们的Odoo安装是基于版本控制的代码存储库，我们将希望它与GitHub存储库保持同步。

幸运的是，我们可以为模块使用额外的位置，这样我们就可以将自定义模块保存在不同的目录中，而不需要将它们与官方的模块混合在一起。

作为一个例子，我们将从这本书中下载代码，在GitHub中提供，并使那些addon模块在我们的Odoo安装中可用。

要从GitHub获得源代码，运行以下命令:

```
$ cd ~/odoo-dev
$ git clone https://github.com/dreispt/todo_app.git -b 10.0
```

我们使用`-b`选项确保我们正在下载10.0版本的模块。

在此之后，我们将在`/odoo`目录旁边有一个新的`/todo_app`目录，其中包含模块。现在我们需要让Odoo知道这个新的模块目录。

配置插件的路径

Odoo服务器有一个名为`addons_path`的配置选项，用于设置服务器应该在哪里查找模块。默认情况下，这指向`/addons`目录，在那里，Odoo服务器正在运行。

我们不仅可以提供一个目录，还可以提供一个目录列表，其中可以找到模块。这允许我们将自定义模块保存在一个不同的目录中，而不让它们与正式的addons混合。

让我们以一个包含我们的新模块目录的addons路径启动服务器:

```
$ cd ~/odoo-dev/odoo
$ ./odoo-bin -d demo --addons-path="../todo_app,./addons"
```

如果您仔细查看服务器日志，您将注意到一条正在使用的addons路径的行: `INFO? odoo: addons paths: [...]`. 确认它包含我们的`todo_app`目录。

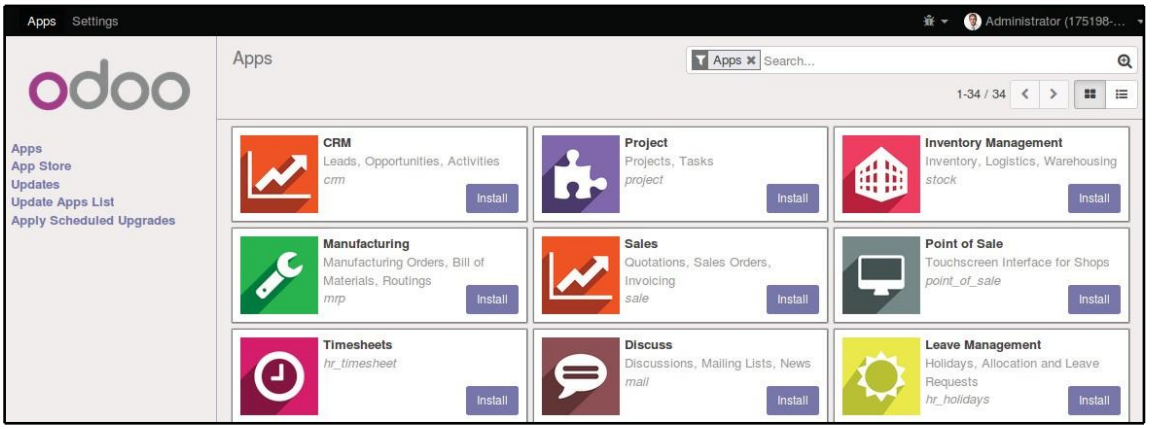
更新应用程序列表

我们还需要让Odoo在这些新模块提供安装之前更新它的模块列表。

为此，我们需要启用开发者模式，因为它提供了**Update Apps List**菜单选项。它可以在**Apps**头部菜单中找到。

更新模块列表后，我们可以确认新的模块可供安装。使用**Apps**菜单选项查看本地模块列表。搜索`todo`，您应该看到可以使用的新模块。

注意，第二个**App Store**菜单选项显示的是Odoo应用商店的模块列表，而不是本地模块:



摘要

在本章中，我们学习了如何设置一个Debian系统来托管Odoo并将其安装到GitHub源代码中。我们还学习了如何创建Odoo数据库并运行Odoo实例。为了允许开发人员在其个人工作站上使用他们最喜欢的工具，我们解释了如何在Odoo主机中配置文件共享。

我们现在应该有一个运行良好的Odoo环境，并对管理数据库和实例感到满意。

有了这个，我们就可以直接开始行动了。在下一章中，我们将从头开始创建我们的第一个Odoo模块，并理解它所涉及的主要元素。

所以让我们开始吧!

2

构建您的第一个Odoo应用程序

在Odoo中开发大部分时间意味着创建我们自己的模块。在本章中，我们将创建我们的第一个Odoo应用程序，并学习使它可用于Odoo并安装它所需的步骤。

在著名的`http://todomvc.com/` 项目的启发下，我们将构建一个简单的To-Do应用程序。它应该允许我们添加新任务，标记它们完成，最后清除所有已完成任务的任务列表。

我们将开始学习开发工作流程的基础:为您的工作建立一个新实例，创建并安装一个新模块，并更新它以应用您在开发迭代中所做的更改。

Odoo遵循一个类似于mvc的架构，我们将在实现To-Do应用程序的过程中仔细检查这些层。

model层，定义应用程序数据的结构

view层，描述用户界面

controller层，支持应用程序的业务逻辑

接下来，我们将学习如何设置访问控制安全性，最后，我们将向模块添加一些描述和品牌信息。

注意这里提到的术语控制器的概念不同于Odoo web开发控制器。这些是web页面可以调用来执行操作的程序端点。

通过这种方法，您将能够逐步了解构成应用程序的基本构建块，并体验从头构建一个Odoo模块的迭代过程。

基本概念

您可能刚刚开始使用Odoo，所以现在很明显是解释Odoo模块的好时机，以及它们是如何在Odoo开发中使用的。

理解应用程序和模块

关于Odoo模块和应用程序很常见。但它们之间的区别到底是什么呢?

Module addons是Odoo应用程序的构建块。一个模块可以向Odoo添加新功能，或者修改现有的功能。它是一个目录，包含一个名为`_manifest_.py`文件，加上实现其特性的其余文件。

Applications是将主要特性添加到Odoo的方式。它们提供了功能领域的核心元素，如会计或HR，基于附加的addon模块修改或扩展特性。因此，它们在Odoo **Apps**菜单中突出显示。

如果您的模块是复杂的，并向Odoo添加新的或主要功能，您可以考虑将其作为应用程序创建。如果您的模块只是对Odoo中的现有功能进行了更改，那么它很可能不是一个应用程序。

模块是否为应用程序，在清单中定义。从技术上讲，它对addon模块的行为没有任何特别的影响。它只是**Apps**列表中的一个亮点。

修改和扩展模块

在我们将要学习的示例中，我们将创建一个新的模块，它的依赖关系尽可能少。

然而，这不是典型的情况。大多数情况下，我们将修改或扩展已经存在的模块。

一般来说，通过直接修改源代码来修改现有模块是一种很糟糕的做法。这对于Odoo提供的官方模块尤其如此。这样做不允许您在原始的模块代码和修改之间有清晰的分离，这使得应用升级变得困难，因为它们将覆盖修改。

相反，我们应该创建在我们想要修改的模块旁边安装的扩展模块，实现我们需要的更改。事实上，Odoo的主要优势之一是**inheritance**机制，它允许定制模块扩展现有模块，无论是正式的还是社区的。继承在所有级别都是可能的：数据模型、业务逻辑和用户界面层。

在本章中，我们将创建一个全新的模块，不扩展任何现有的模块，只关注模块创建中涉及的不同部分和步骤。我们将对每个部分进行简单的研究，因为每个部分将在后面的章节中详细研究。

一旦我们熟悉了创建一个新模块，我们就可以深入到继承机制，这将在第3章中介绍，继承-扩展现有的应用程序。

为了使Odoo得到有效的开发，我们应该对开发工作流程感到满意:管理开发环境，应用代码更改，并检查结果。本节将介绍这些基础知识。

创建模块基本骨架

按照第一章的说明，开始使用Odoo开发，我们应该有Odoo服务器在`~/odoo-dev/odoo/`。为了使事情保持整洁，我们将在它旁边创建一个新的目录来托管我们的自定义模块，在`~/odoo-dev/custom-addons`中。

Odoo包括一个`scaffold`命令来自动创建一个新的模块目录，它的基本结构已经就绪。您可以通过以下命令了解更多信息:

```
$ ~/odoo-dev/odoo/odoo-bin scaffold --help
```

当您开始编写下一个模块时，您可能想要记住这一点，但我们现在不会使用它，因为我们更喜欢手动创建模块的所有结构。

一个Odoo addon模块是一个包含`_manifest_.py`描述文件。

在以前的版本中，这个描述符文件被命名为_openerp_.py。这个名称仍然被支持，但已被弃用。

它还需要是Python可输入的，所以它也必须有一个_init_.py文件。

模块的目录名是它的技术名称。我们将使用todo_app来完成它。技术名称必须是一个有效的Python标识符:它应该以字母开头，只能包含字母、数字和下划线字符。
下面的命令将创建模块目录并创建一个空__init_.py文件在~/odoo-dev/custom-addons/todo_app/_init_.py里面

如果你想直接从命令行做到这一点，这就是你要使用的:

```
$ mkdir ~/odoo-dev/custom-addons/todo_app
$ touch ~/odoo-dev/custom-addons/todo_app/_init__.py
```

接下来，我们需要创建清单文件。它应该只包含一个包含十几个可能属性的Python字典;其中，只有name属性是必需的。description描述属性，用于更长的描述，并且author属性提供更好的可视性和建议。

我们现在应该添加一个_manifest_.py文件在这个文件边上是_init_.py文件，这个_manifest_.py文件内容如下:

```
{
    'name': 'To-Do Application', 'description':
    'Manage your personal To-Do tasks.',
    'author': 'Daniel Reis',
    'depends': ['base'],
    'application': True,
}
```

depends属性可以拥有其他需要的模块列表。当安装此模块时，Odoo将自动安装它们。这不是一个强制性的属性，但建议总是拥有它。如果不需要特殊的依赖关系，那么我们应该依赖于核心base模块。
您应该注意确保在这里显式地设置所有依赖项;否则，该模块可能无法在干净的数据库中安装(由于缺少依赖关系)，或者在随后加载其他必需的模块时加载错误。

对于我们的应用程序，我们不需要任何特定的依赖项，因此我们依赖于`base`模块。

为了简洁，我们选择使用非常少的描述符键，但是，在一个真实的应用场景中，我们建议您也使用额外的键，因为它们与Odoo应用程序商店相关:

`summary` 显示为模块的副标题。

`version` 默认情况下，是`1.0`。它应该遵循语义版本规则(参见 http://semver.org/以获得详细信息)。

`license` 标识符默认是`LGPL-3`。

`website` 是一个URL，用于查找关于模块的更多信息。这可以帮助人们找到更多的文档或问题跟踪器来归档bug和建议。

`category` 是模块的功能类别，默认为未分类。在**Application**字段下拉列表中，可以在安全组表单(**Settings** | **User** | **Groups**)中找到现有类别的列表。

这些其他描述符键也可用:

`installable` 默认为`True`，但可以设置为`False`来禁用模块。

`auto_install` 如果设置为`True`，该模块将自动安装，提供所有依赖项已经安装。它是用来做胶水的模块。

由于Odoo 8.0，而不是`description`键，我们可以使用`README.rst`或`README.md`文件在模块的顶部目录。

对许可证

为你的工作选择一个许可是非常重要的，你应该仔细考虑什么是你最好的选择，以及它的含义。Odoo模块最常用的许可证是**GNU Lesser General Public License(LGLP)**和**Affero General Public License(AGPL)**。LGPL更加宽容，允许商业派生工作，而不需要共享相应的源代码。AGPL是一个更强的开源许可，它需要派生的工作和服务托管来共享它们的源代码。在https://www.gnu.org/licenses/了解更多关于GNU许可证。

添加到addons路径

现在我们有一个简约的新模块,我们想让它可用Odoo实例。

为此，我们需要确保包含模块的目录位于addons路径中，然后更新Odoo模块列表。

这两个动作在前一章都有详细的解释，但是在这里，我们将继续简要概述一下需要做什么。

我们将在工作目录中定位，并以适当的addons路径配置启动服务器:

```
$ cd ~/odoo-dev
$ ./odoo/odoo-bin -d todo --addons-path="custom-addons,odoo/addons" --save
```

`--save`选项保存您在配置文件中使用的选项。每次重新启动服务器时，都要避免重复它们:
运行`./odoo-bin`和最后一个保存的选项将被使用。
仔细查看服务器日志。它应该有一个信息`INFO ? odoo: addons paths:[...]`行，它应该包括我们`custom-addons`目录。
记住,也要包含您可能正在使用的其他addon目录。例如,如果您还有一个`~/odoo-dev/extra`的目录，其中包含了额外的模块 您可能希望将它们包括在`--addons-path`选项中:

```
--addons-path="custom-addons,extra,odoo/addons"
```

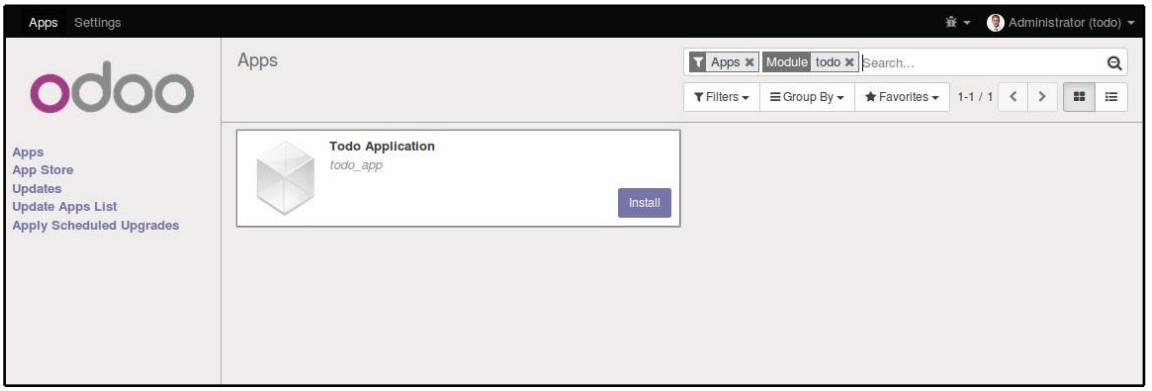
现在我们需要Odoo实例来确认刚刚添加的新模块。

安装新模块

在**Apps**菜单中，选择**Update Apps List**选项。这将更新模块列表，添加自上次更新后可能添加的任何模块。请记住，我们需要为该选项启用的开发模式是可见的。这是完成的**Settings**指示板，在链接的底部右侧，在Odoo版本号信息下面。

确保您的web客户端会话与正确的数据库一起工作。您可以在右上角检查:数据库名称在括号中显示，在用户名之后。使用正确的数据库执行的一种方法是使用附加选项启动服务器实例`--db-filter=^MYDB$.`

Apps选项显示了可用模块的列表。默认情况下，它只显示应用程序模块。由于我们已经创建了一个应用程序模块，所以我们不需要删除该过滤器来查看它。键入`todo`在搜索中，您应该看到我们的新模块，准备安装:



现在点击模块的**Install**按钮，我们准备好了!

升级模块

开发一个模块是一个迭代的过程，您需要对源文件进行修改，并在Odoo中进行可见。

在许多情况下，这是通过升级模块来完成的:在**Apps**列表中查找模块，一旦它已经安装，您将有一个可用的**Upgrade**按钮。

然而，当更改仅在Python代码中进行时，升级可能没有效果。不需要进行模块升级，需要重新启动应用程序服务器。由于Odoo只装载一次Python代码，因此，任何后来的代码更改都需要重新启动服务器。

在某些情况下，如果模块更改在数据文件和Python代码中，那么您可能需要两个操作。对于新的Odoo开发人员来说，这是一个常见的混淆来源。

但幸运的是，还有更好的方法。使我们对模块有效的所有更改的最安全、最快的方法是停止并重新启动服务器实例，请求将我们的模块升级到我们的工作数据库。

在服务器实例正在运行的终端中，使用*Ctrl + C*来停止它。然后，使用以下命令启动服务器并升级`todo_app`模块:

```
$ ./odoo-bin -d todo -u todo_app
```

`-u` 选项(或者，从`--update`的更新)需要`-d`选项，并接受一个逗号分隔的模块列表进行更新。例如，我们可以使用`-u todo_app,mail`。当一个模块被更新时，所有其他安装的模块也会被更新。这对于维护继承机制的完整性是至关重要的。

在本书中 ,当您需要应用模块中的工作时 ,最安全的方法是用前面的命令重新启动Odoo实例。按下向上箭头键会给你带来以前使用过的命令。所以 ,大多数时候 ,你会发现自己使用*Ctrl + C*，然后输入键组合。

不幸的是，两个更新模块列表和卸载模块都是无法通过命令行获得的操作。这些需要通过**Apps**菜单中的web界面完成。

服务器开发模式

在Odoo 10中，一个新的选项被引入提供开发友好的特性。要使用它，可以使用附加选项启动服务器实例`--dev=all`。

这使得一些方便的特性能够加速我们的开发周期。最重要的是:

> 自动重新加载Python代码，一旦保存了Python文件，避免手动服务器重新启动
> 直接从XML文件读取视图定义，避免手动模块升级

`--dev`选项接受一个以逗号分隔的选项列表，尽管`all`选项在大多数情况下都是合适的。我们还可以指定我们喜欢使用的调试器。默认情况下，使用Python调试器`pdb`。有些人可能更喜欢安装和使用其他调试器。这里也支持`ipdb`和`pudb`。

model层

现在，Odoo知道了我们的新模块，让我们从添加一个简单的模型开始。

模型描述业务对象，例如机会、销售订单或合作伙伴(客户、供应商等)。模型有一个属性列表，也可以定义它的特定业务。

模型是使用从一个Odoo模板类派生的Python类实现的。它们直接转换到数据库对象，Odoo在安装或升级模块时自动处理这些问题。对此负责的机制是**Object Relational Model** (**ORM**)。

我们的模块将是一个非常简单的应用程序，以保存待办事项。这些任务将有一个单独的文本字段用于描述和一个复选框来标记它们。我们应该稍后添加一个按钮来清理已完成任务的待办事项列表。

创建数据模型

Odoo开发准则规定，模型的Python文件应该放在一个模型子目录中。为了简单起见，我们不会在这里跟踪它，所以我们创建一个`todo_model.py`文件位于`todo_app`模块的主目录中。

添加以下内容:

```
# -*- coding: utf-8 -*-
from odoo import models, fields
class TodoTask(models.Model):
    _name = 'todo.task'
    _description = 'To-do Task'
    name = fields.Char('Description', required=True)
    is_done = fields.Boolean('Done?')
    active = fields.Boolean('Active?', default=True)
```

第一行是一个特殊的标记，它告诉Python解释器这个文件有UTF-8，这样它就可以预期和处理non-ASCII字符。我们不会使用任何东西，但无论如何这是一个很好的实践。
第二行是Python代码导入语句，可以从Odoo内核中获取`models`和`fields`对象。

第三行声明了我们的新模型。这是一个来自`models.Model`的类。

下一行设置了定义标识符的`_name`属性，该属性将在整个Odoo中使用，以引用该模型。注意，在本例中，实际的Python类名称`TodoTask`对其他Odoo模块没有意义。`_name`值将用作标识符。

注意，这个和下面的行是缩进的。如果您不熟悉Python，您应该知道这一点很重要:缩进定义了一个嵌套的代码块，因此这四行应该都是同样的缩进。

然后我们有`_description`模型属性。它不是强制性的，但它为模型记录提供了一个用户友好的名称，可以用于更好的用户消息。

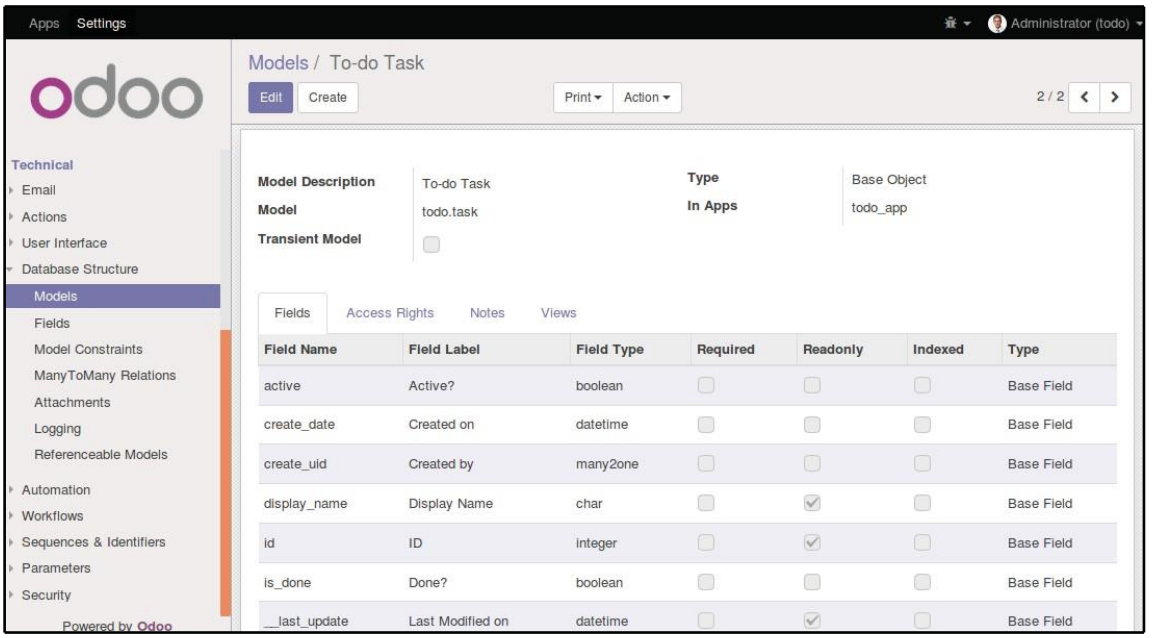
最后三行定义了模型的字段。值得注意的是，`name`和`active`是特殊的字段名。在默认情况下，Odoo在引用其他模型时将使用`name`字段作为记录的标题。`active`字段用于激活记录，默认情况下只显示活动记录。我们将使用它清除完成的任务，而不实际从数据库中删除它们。

现在，这个文件还没有被模块使用。我们必须告诉Python在`_init_.py`文件中使用模块加载它。让我们编辑它来添加以下一行:

```
from . import todo_model
```

就是这样!对于我们的Python代码更改生效，服务器实例需要重新启动(除非使用`--dev`模式)。

我们不会看到任何菜单选项来访问这个新模型，因为我们还没有添加它们。不过，我们可以使用**Technical**菜单来检查新创建的模型。在**Settings**顶部菜单，去**Technical** | **Database Structure** | **Models**，搜索`todo.task`模型在列表上，点击它看它的定义:



如果一切顺利，模型和字段就会被创建。如果您在这里看不到它们，可以尝试使用一个模块升级来重启服务器，如前所述。

我们还可以看到一些我们没有声明的字段。这些是保留的字段Odoo自动添加到每个新模型。他们如下:

`id`是模型中每个记录的惟一数字标识符。

`create_date`和`create_uid`分别指定创建记录和创建记录的时间。

`write_date`和`write_uid`在记录最后修改和修改时确认。

`__last_update`是一个没有实际存储在数据库中的助手。它用于并发检查。

添加自动化测试

编程最佳实践包括对代码进行自动化测试。对于像Python这样的动态语言来说，这一点更为重要。因为没有编译步骤，所以您不能确定没有语法错误，直到代码由解释器运行。一个好的编辑器可以帮助我们提前发现这些问题，但是不能帮助我们确保代码执行得像自动化测试一样。

Odoo支持两种方法来描述测试:使用YAML数据文件或使用Python代码，基于Unittest2库。YAML测试是旧版本的遗留问题，不推荐使用。我们更喜欢使用Python测试，并将一个基本的测试用例添加到我们的模块中。

测试代码文件应该以`test_`开头，并且应该从`tests/_init_.py`导入。但是`tests`目录(或Python子模块)不应该从模块的顶部`_init_.py`导入，因为它将在测试执行时自动被发现和加载。

测试必须放置在一个`tests/`子目录中。添加一个`tests/_init_.py`文件如下:

```
from . import test_todo
```

现在添加实际的测试代码，在`tests/test_todo.py`文件:

```
# -*- coding: utf-8 -*-
from odoo.tests.common import TransactionCase

class TestTodo(TransactionCase):

    def test_create(self):
        "Create a simple Todo"
        Todo = self.env['todo.task']
        task = Todo.create({'name': 'Test Task'})
        self.assertEqual(task.is_done, False)
```

这就添加了一个简单的测试用例来创建一个新to-do task，并验证所做的工作是否**Is Done**?字段具有正确的默认值。

现在我们要运行我们的测试。这是通过在安装模块时添加`--test-enable`选项来实现的:

```
$ ./odoo-bin -d todo -i todo_app --test-enable
```

Odoo服务器将在升级后的模块中查`tests/`子目录，并运行它们。如果其中任何一个测试失败，服务器日志将显示这一点。

view层

视图层描述用户界面。视图是使用XML定义的，它由web客户端框架使用，以生成具有数据感知的HTML视图。
我们有菜单项可以激活可以呈现视图的动作。例如, **Users**菜单项处理一个名为**Users**的操作，依次呈现一系列视图。有一些可用的视图类型，例如列表和表单视图，并且提供的筛选器选项也由特定类型的视图(search视图)定义。

Odoo开发指南指出定义用户界面的XML文件应该放在一个`views/`子目录中。

让我们开始为To-Do应用程序创建用户界面。

添加菜单项

现在我们有了存储数据的模型，我们应该在用户界面上提供它。

为此，我们应该添加一个菜单选项来打开要做的`To-do Task`模型，这样它就可以被使用。

创建`views/todo_menu.xml`文件定义一个菜单项和由它执行的操作:

```
<?xml version="1.0"?>
<odoo>
  <!-- Action to open To-do Task list -->
  <act_window id="action_todo_task"
    name="To-do Task"
    res_model="todo.task"
    view_mode="tree,form" />
  <!-- Menu item to open To-do Task list -->
  <menuitem id="menu_todo_task"
    name="Todos"
    action="action_todo_task" />
</odoo>
```

用户界面(包括菜单选项和操作)存储在数据库表中。XML文件是用于在安装或升级模块时将这些定义加载到数据库中的数据文件。前面的代码是一个Odoo数据文件，描述两个记录添加到Odoo:

> `<act_window>`元素定义了一个客户端窗口操作，可以打开`todo.task`模型，并在这个顺序中启用了`tree`和`form`视图。
>
> `<menuitem>`定义了一个顶级菜单项，调用`action_todo_task`操作，这是以前定义的。

这两个元素都包含`id`属性。这个id属性也被称为**XML ID**，它非常重要:它被用来唯一标识模块内部的每个数据元素，并可以由其他元素来引用它。在这种情况下，`<menuitem>`元素需要引用操作来处理，并且需要使用`<act_window>`ID。在第4章，模块数据中，XML id被更详细地讨论。

我们的模块还不知道新的XML数据文件。这是通过将其添加到`_manifest_.py`文件中的`data`属性完成的。它保存由模块加载的文件列表。将此属性添加到manifest的字典:

```
'data': ['views/todo_menu.xml'],
```

现在我们需要重新升级模块以使这些更改生效。去**Todos**顶部菜单，你应该看到我们的新菜单选项:

即使我们没有定义我们的用户界面视图，点击**Todos**菜单会为我们的模型打开一个自动生成的表单，允许我们添加和编辑记录。

Odoo很好，可以自动生成它们，这样我们就可以立即开始使用我们的模型了。

到目前为止还好!现在让我们改进我们的用户界面。试着逐步改进，如下一节所示，做频繁的模块升级，不要害怕尝试。您可能还想尝试`--dev=all` 服务选项。使用它，视图定义可以直接从XML文件中读取，这样您的更改就可以在不需要模块升级的情况下立即可用到Odoo。

如果因为XML错误而升级失败，请不要惊慌!请注释掉最后编辑的XML部分，或者从`_manifest_.py`中删除XML文件并重复升级。服务器应该正确启动。现在仔细阅读服务器日志中的错误消息:它应该指出问题所在。

Odoo支持多种类型的视图 ,但最重要的三个视图是: `tree` (通常称为列表视图)、`form`和`search`视图。我们将为我们的模块添加一个示例。

创建form视图

所有视图都存储在数据库中，在`ir.ui.view`模型中。为了向模块添加视图，我们声明了一个`<record>`元素，描述了XML文件中的视图，当模块安装时，它将被加载到数据库中。

添加这个新的`views/todo_view.xml`文件来定义我们的表单视图:

```
<?xml version="1.0"?>
<odoo>
  <record id="view_form_todo_task" model="ir.ui.view">
    <field name="name">To-do Task Form</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
      <form string="To-do Task">
        <group>
          <field name="name"/>
          <field name="is_done"/>
          <field name="active" readonly="1"/>
        </group>
      </form>
    </field>
  </record>
</odoo>
```

请记住将这个新文件添加到清单文件中的数据键，否则，我们的模块将不知道它，它将不会被加载。

这将向`ir.ui.view`模型添加一个记录，并使用标识符`view_form_todo_task`。视图是`todo.task`模型，命名为`To-do Task Form`。这个名字只是为了获取信息;它不必是唯一的，但它应该允许一个人很容易地识别它所指的记录。实际上，这个名称可以完全省略，在这种情况下，它将由模型名称和视图类型自动生成。

最重要的属性是`arch`，它包含视图定义，在上面的XML代码中突出显示。`<form>`标记定义了视图类型，在本例中，它包含三个字段。我们还向活动字段添加了一个属性，使其仅读取。

业务form文档视图

前面的部分提供了一个基本的表单视图，但是我们可以对它进行一些改进。对于文档模型，Odoo有一个模仿纸质页面的演示样式。该表单包含两个元素: `<header>`包含操作按钮和`<sheet>`以包含数据字段。

我们现在可以替换上一节中定义的基本`<form>`:

```
<form>
  <header>
  <!-- Buttons go here-->
  </header>
  <sheet>
    <!-- Content goes here: -->
    <group>
      <field name="name"/>
      <field name="is_done"/>
      <field name="active" readonly="1"/>
    </group>
  </sheet>
</form>
```

添加action按钮

表单可以有执行动作的按钮。这些按钮可以运行窗口操作，例如打开另一个表单或运行在模型中定义的Python函数。

它们可以放在表单的任何位置 ,但是对于文档样式表单 ,推荐的位置是`<header>`部分。

对于我们的应用程序，我们将添加两个按钮来运行`todo.task`模型的方法:

```
<header>
  <button name="do_toggle_done" type="object"
    string="Toggle Done" class="oe_highlight" />
  <button name="do_clear_done" type="object"
    string="Clear All Done" />
</header>
```

按钮的基本属性包括以下内容:

`string` 文本显示在按钮上

`type` 它执行的动作

`name` 这个动作的标识符是什么

`class` 是否可以使用CSS样式的可选属性，比如在普通HTML中

使用组来组织表单

`<group>`标记允许您组织表单内容。在`<group>`元素内放置`<group>`元素，在外部组内创建一个两列布局。建议组元素有一个`name`属性，以便其他模块可以更容易地扩展它们。

我们将使用这个来更好地组织我们的内容。让我们改变表格的`<sheet>`内容来匹配这个:

```
<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>
```

完整的form视图

在这一点上，`todo.task`表单视图应该是这样的:

```
<form>
  <header>
    <button name="do_toggle_done" type="object"
      string="Toggle Done" class="oe_highlight" />
    <button name="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>
```

请记住，对于要加载到Odoo数据库的更改，需要进行模块升级。要查看web客户端的变化，需要重新加载表单:要么单击打开它的菜单选项，要么重新加载浏览器页面(大多数浏览器中的*F5*)。

操作按钮还不能工作，因为我们还需要添加它们的业务逻辑。

添加列表和search视图

当在列表模式中查看模型时，使用`<tree>`视图。树视图能够显示在层次结构中组织的行，但大多数时候，它们被用来显示普通列表。

我们可以将以下`tree`视图定义添加到`todo_view.xml`:

```
    <record id="view_tree_todo_task" model="ir.ui.view">
      <field name="name">To-do Task Tree</field>
      <field name="model">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>
```

这定义了一个只有两列的列表: `name`和`is_done`。我们还添加了一个漂亮的触摸:完成任务的行(`is_done==True`)显示为灰色。这是通过使用Bootstrap类`muted`来完成的。检查`http://getbootstrap.com/css/#helper-classes-colors`更多信息引导和它的背景颜色。

在列表的右上角，Odoo显示一个搜索框。它搜索的字段和可用的过滤器由`<search>`视图定义。

如前所述，我们将将其添加到`todo_view.xml`:

```
  <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>
```

`<field>`元素定义在搜索框中输入的字段。`<filter>`元素添加预定义的筛选条件，可以通过使用特定语法定义的用户单击进行切换。

业务逻辑层

现在我们将在按钮上添加一些逻辑。这是用Python代码完成的，在模型的Python类中使用方法。

添加业务逻辑

我们应该编辑`todo_model.py`Python文件以向类添加按按钮调用的方法。首先，我们需要导入新的API，因此将其添加到Python文件顶部的import语句:

```
from odoo import models, fields, api
```

Toggle Done按钮的作用将非常简单:只需切换**Is Done**? 标记。对于记录的逻辑，使用`@api.multi`装饰器。在这里，`self`将代表一个记录集，然后我们应该对每个记录进行循环。

在`TodoTask`类内，添加以下内容:

```
@api.multi
def do_toggle_done(self):
    for task in self:
        task.is_done = not task.is_done
    return True
```

代码循环遍历所有的to-do记录，并为每个任务记录修改`is_done`字段，从而改变其值。这个方法不需要返回任何东西，但是我们应该至少返回一个`True`值。原因是，客户机可以使用XML-RPC调用这些方法，而这个协议不支持只返回`None`值的服务器函数。

对于**Clear All Done**按钮，我们想更进一步。它应该查找已经完成的所有活动记录，并使它们不活动。通常情况下，表单按钮只会在选定的记录上执行，但在这种情况下，我们希望它也能根据当前的记录采取行动:

```
@api.model
def do_clear_done(self):
    dones = self.search([('is_done', '=', True)])
    dones.write({'active': False})
    return True
```

在以`@api.model`为装饰的方法中，`self`变量代表了没有特别记录的模型。我们将构建一个包含所有被标记的任务的`dones`记录集。然后，我们将`active`标记设置为`False`。

`search`方法是一种返回满足某些条件的记录的API方法。这些条件是在一个域中编写的，它是三胞胎的列表。我们将在第6章中详细讨论域，视图——设计用户界面。

`write`方法在记录集的所有元素上同时设置值。使用字典描述要写入的值。在这里使用`write`比遍历记录集更有效地将值逐个分配给每一个。

添加测试

现在我们应该为业务逻辑添加测试。理想情况下，我们希望每一行代码都能被至少一个测试用例覆盖。在`tests/test_todo.py`中，再添加几行代码到`test_create()`方法:

```
# def test_create(self):
        # ...
        # Test Toggle Done
        task.do_toggle_done()
        self.assertTrue(task.is_done)
        # Test Clear Done
        Todo.do_clear_done()
        self.assertFalse(task.active)
```

如果我们现在运行测试，并且正确地编写了模型方法，那么在服务器日志中应该不会看到错误消息:

```
$ ./odoo-bin -d todo -i todo_app --test-enable
```

设置访问安全

您可能已经注意到，在加载时，我们的模块在服务器日志中得到一个警告消息:

```
The model todo.task has no access rules, consider adding one.
```

消息非常清楚:我们的新模型没有访问规则，因此除了管理超级用户之外，其他任何人都不能使用它。作为一个超级用户，`admin`忽略了数据访问规则，这就是为什么我们能够在没有错误的情况下使用表单。但我们必须在其他用户使用我们的模型之前解决这个问题。

我们还需要解决的另一个问题是，我们希望to-do任务对每个用户都是私有的。Odoo支持行级访问规则，我们将使用它来实现这一点。

测试访问安全

事实上，由于缺少访问规则，我们的测试现在应该失败了。他们不是因为他们和管理用户做的。因此，我们应该更改它们，以便它们使用演示用户。

为此，我们应该编辑`tests/test_todo.py`文件添加`setUp`方法:

```
# class TestTodo(TransactionCase):

    def setUp(self, *args, **kwargs):
        result = super(TestTodo, self).setUp(*args, \
        **kwargs)
        user_demo = self.env.ref('base.user_demo')
        self.env= self.env(user=user_demo)
        return result
```

第一个指令调用父类的`setUp`代码。下一个改变环境，用于运行测试，`self.env`，到一个新的使用演示用户。我们已经编写的测试不需要进一步的更改。

我们还应该添加一个测试用例，以确保用户只能看到自己的任务。为此，首先，在顶部添加一个额外的导入:

```
from odoo.exceptions import AccessError
```

接下来，向测试类添加一个额外的方法:

```
    def test_record_rule(self): "Test
        per user record rules" Todo =
        self.env['todo.task']
        task = Todo.sudo().create(('name': 'Admin Task'})
        with self.assertRaises(AccessError):
            Todo.browse([task.id]).name
```

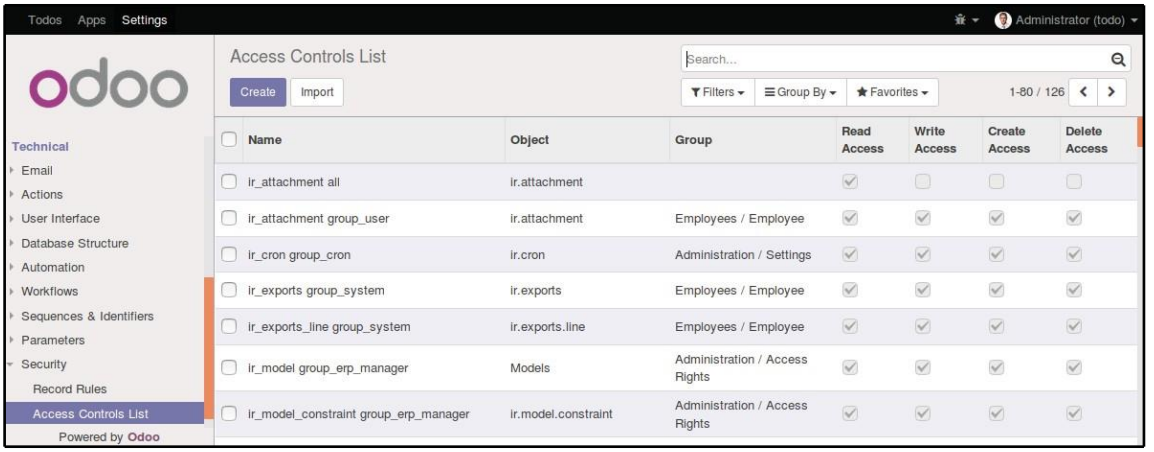
由于我们的`env`方法现在使用了演示用户，所以我们使用`sudo()`方法将上下文更改为admin用户。然后我们使用它来创建一个不应该被演示用户访问的任务。

当尝试访问这个任务数据时，我们期望提高`AccessError`异常。如果我们现在运行测试，

它们应该失败，所以让我们来处理它。

添加访问控制安全

要获取向模型添加访问规则所需的信息的图片，请使用web客户端并前往**Settings** | **Technical** | **Security** | **Access Controls List**:



这里我们可以看到一些模型的ACL。它表示，每个安全组允许记录哪些操作。

此信息必须由模块提供，使用数据文件将行加载到`ir.model.access`模型中。我们将在模型上添加完全访问员工组。员工是基本的准入群体，几乎每个人都属于。

这是使用一个名为`security/ir.model.access.csv`的CSV文件完成的。让我们添加以下内容:

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlin
k
acess_todo_task_group_user,todo.task.user,model_todo_task,base.group_user,1
,1,1,1
```

文件名对应于将数据加载到的模型，文件的第一行有列名。这些是CSV文件中提供的列:

`id` 是记录外部标识符(也称为XML ID)。在我们的模块中应该是唯一的。

`name` 是一个描述标题。它只提供信息，最好是保持独特。官方模块通常使用与模型名称和组相隔离的字符串。按照这个惯例，我们使用了`todo.task.user`。

`model_id` 是我们提供访问的模型的外部标识符。模型具有由ORM自动生成的XML ID:对于`todo.task`，标识符是`model_todo_task`。

`group_id` 标识要授予权限的安全组。最重要的是由基本模块提供的。Employee组就是这样一个例子，它有一个标识符`base.group_user`。

`Perm` 字段标记对授予`read`, `write`, `create`,或`unlink` (删除)访问的权限。

我们不能忘记在`_manifest_.py`描述符的`data`属性中添加对这个新文件的引用。它应该是这样的:

```
'data': [
    'security/ir.model.access.csv',
    'views/todo_view.xml',
    'views/todo_menu.xml',
],
```

和以前一样，升级这些添加的模块以生效。警告消息应该消失，我们可以通过登录用户`demo` (密码也为`demo`)来确认权限是否OK。如果我们现在运行我们的测试，他们应该只会失败`test_record_rule`测试用例。

行级访问规则

我们可以**Technical**菜单中找到**Record Rules**选项，以及**Access Control List**。

记录规则在`ir.rule`模型中定义。像往常一样，我们需要提供一个独特的名字。我们还需要它们操作的模型以及用于访问限制的域过滤器。域筛选器使用在Odoo中使用的通常的元组语法列表。

通常，规则适用于某些特定的安全组。在我们的案例中，我们将把它应用于Employees组。如果它适用于没有安全组，特别是它被认为是全局的(`global`字段自动被设置为`True`)。全球规则是不同的，因为它们强加了一些非全球规则无法推翻的限制。

要添加记录规则，我们应该创建一个`security/todo_access_rules.xml`文件包含以下内容:

```
<?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>
```

注意到`noupdate="1"`属性。这意味着该数据将不会在模块升级中更新。这将允许它在以后定制，因为模块升级不会破坏用户做出的改变。但是请注意，在开发过程中也会出现这种情况，因此在开发过程中，您可能需要设置`noupdate="0"`，直到您对数据文件感到满意为止。

在`groups`字段中，您还将找到一个特殊的表达式。它是一对多关系字段，它们有一个特殊的语法来操作。在本例中，(4、x)元组指示将x附加到记录，这里x是对Employees组的引用，由`base.group_user`标识。在第四章，模块数据中，详细讨论了这种一对多的书写特殊语法。

如前所述，我们必须将文件添加到_manifest_.py，然后才能加载到模块中:

```
'data':
  [ 'security/ir.model.access.csv',
  'security/todo_access_rules.xml',
  'todo_view.xml',
  'todo_menu.xml',
],
```

如果我们做对了，我们可以运行模块测试，现在它们应该通过了。

更好地描述模块

我们的模块看起来不错。为什么不添加一个图标来让它看起来更好呢?为此，我们只需要向模块添加一个带有该图标的static/description/icon.png文件。

我们将重新使用现有**Notes**应用程序的图标，因此我们应该将odoo/addons/static/description/icon.png文件复制到custom-addons/todo_app/static/description目录中。

下面的命令应该为我们提供这样的技巧:

```
$ mkdir -p ~/odoo-dev/custom-addons/todo_app/static/description
$ cp ~/odoo-dev/odoo/addons/note/static/description/icon.png ~/odoo-
dev/custom-addons/todo_app/static/description
```

现在，如果我们更新模块列表，我们的模块应该显示为新的图标。我们还可以为它添加一个更好的描述来解释它的作用，以及它有多伟大。这可以在_manifest_.py文件的description键中完成。但是，首选的方法是将一个README.rst文件添加到模块根目录。

摘要

我们从一开始就创建了一个新的模块，涵盖了模块中最常用的元素:模型、三种基本视图类型(表单、列表和搜索)、模型方法中的业务逻辑以及访问安全性。

在此过程中，我们熟悉了模块的开发过程，包括模块升级和应用服务器重新启动，以使逐步的变化在Odoo中有效。

记住，在添加模型字段时，需要进行升级。在改变Python代码(包括清单文件)时，需要重新启动。在更改XML或CSV文件时，需要进行升级;同样，当有疑问时，两者都要做:重新启动服务器并升级模块。

在下一章中，您将学习如何构建在现有的模块上叠加以添加功能的模块。

3

继承-扩展现有的应用程序

Odoo最强大的功能之一是在不直接修改底层对象的情况下添加功能。

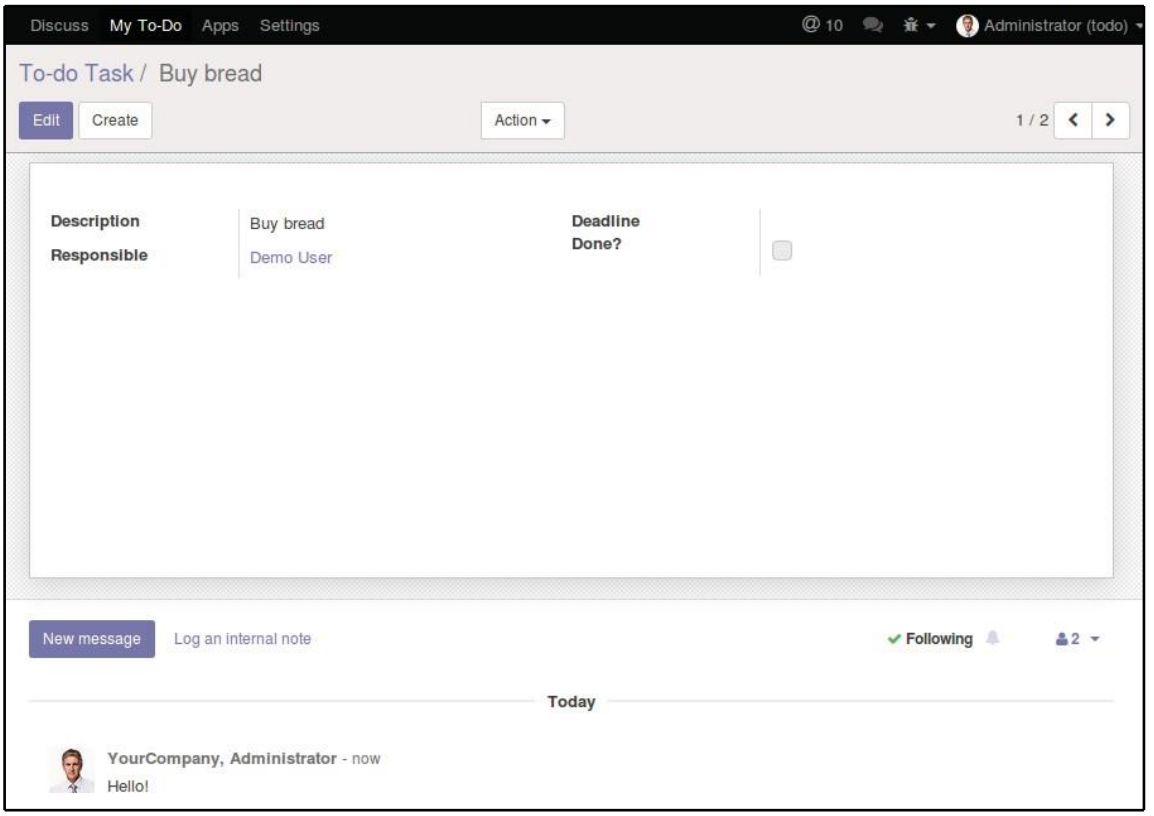
这是通过继承机制实现的，它是在现有对象之上的修改层。这些修改可以在所有级别进行:模型、视图和业务逻辑。我们不直接修改现有的模块，而是创建一个新的模块来添加预期的修改。

在本章中，您将学习如何编写自己的扩展模块，使您能够利用现有的核心或社区应用程序。作为一个相关的示例，您将学习如何向您自己的模块中添加Odoo的社交和消息传递特性。

向To-Do应用程序添加共享功能

我们的To-Do应用程序现在允许用户私下管理他们自己的待办事项。通过添加协作和社交网络功能，将应用程序提升到另一个层次难道不是很好吗?我们将能够分享任务并与他人讨论。

我们将使用一个新模块来扩展之前创建的To-Do应用程序，并使用继承机制添加这些新特性。以下是我们在本章末尾期望达到的目标:



这将是我们为实现的功能扩展的工作计划:

扩展任务模型，例如负责该任务的用户

修改业务逻辑，只在当前用户的任务上运行，而不是用户能够看到的所有任务

扩展视图，将必要的字段添加到视图中

添加社交网络功能:消息墙和追随者

我们将开始在`todo_app`模块旁边创建一个新的`todo_user`模块的基本框架。在第1章的安装示例中，开始使用Odoo开发，我们将在`~/odoo-dev/custom-addons/`托管我们的模块。我们应该为模块添加一个新的`todo_user`目录，其中包含一个空的`__init_.py`文件。

现在创建`todo_user/_manifest_.py`，包含以下代码:

```
(  'name': 'Multiuser To-Do',
   'description': 'Extend the To-Do app to multiuser.',
   'author': 'Daniel Reis',
   'depends': ['todo_app'], }
```

我们在这里没有做过这个，但是包括`summary`和`category`键在发布模块到Odoo在线应用商店时很重要。

注意，我们添加了对`todo_app`模块的显式依赖。这对于继承机制的正常工作是必要和重要的。从现在开始，当`todo_app`模块被更新时，所有依赖于它的模块，如`todo_user`模块，也将被更新。

接下来,安装它。在**Apps**下使用**Update Apps List**菜单选项更新模块列表就足够了;在**Apps**列表中找到新模块并单击它的**Install**按钮。注意，这次您需要删除默认的**Apps**过滤器，以便在列表中看到新模块，因为它没有被标记为应用程序。有关发现和安装模块的详细说明，请参阅第1章，开始使用Odoo开发。

现在，让我们开始添加新特性。

扩展模型

新模型通过Python类定义。扩展它们也可以通过Python类进行，但是在odoo特有的继承机制的帮助下。

为了扩展现有模型，我们使用带有`_inherit`属性的Python类。这标识了要扩展的模型。这个新类继承了父Odoo模型的所有特性，我们只需要声明我们想要引入的修改。

事实上，Odoo模型存在于我们特定的Python模块之外，在一个中央注册表中。这个注册表可以从使用`self.env[<model name>]`的模型方法访问。例如，为了获得对代表`res.partner`模型的对象的引用，我们将编写`self.env['res.partner']`。

为了修改一个Odoo模型，我们得到一个对它的注册表类的引用，然后对其进行就地修改。这意味着这些修改也将在其他地方可用，在这个新模型被使用的地方。

在Odoo服务器启动时，加载该序列的模块是相关的:一个附加模块所做的修改只会在随后加载的附加组件上可见。因此，正确设置模块依赖关系非常重要，确保提供我们使用的模型的模块包含在我们的依赖树中。

在模型中添加字段

我们将扩展`todo.task`模型，为它添加几个字段:负责任务的用户和截止日期。

编码风格指南建议拥有一个每个Odoo模型的`models/`子目录。因此，我们应该从创建模型子目录开始，使其成为Python- importable。

编辑`todo_user/_init_.py`文件的内容:

```
from .import models
```

用以下代码创建`todo_user/models/_init_.py`:

```
from . import todo_task
```

前一行指示Python在同一目录中查找名为`odoo_task.py`的文件并导入它。您通常在目录中的每个Python文件都有一个`from`行:

现在创建`todo_user/models/todo_task.py`文件来扩展原来的模型:

```
# -*- coding: utf-8 -*-
from odoo import models, fields, api
class TodoTask(models.Model):
    _inherit = 'todo.task'
    user_id = fields.Many2one('res.users', 'Responsible')
    date_deadline = fields.Date('Deadline')
```

类名`TodoTask`是这个Python文件的本地名称，通常与其他模块无关。`_inherit`类属性是这里的关键:它告诉Odoo这个类继承并因此修改了`todo.task`模型。

注意，`_name`属性没有出现。不需要它，因为它已经继承自父模型。

接下来的两行是常规字段声明。`user_id`字段表示来自用户模型`res.users`的用户。它是一个`Many2one`字段，相当于一个数据库术语的外键。`date_deadline`是一个简单的日期字段。在第5章中，模型——构造应用程序数据，我们将更详细地解释Odoo中可用的字段类型。

为了将新字段添加到模型的支持数据库表中 ,我们需要执行模块升级。如果一切按照预期进行，那么在**Technical** | **Database Structure** | **Models**菜单选项中检查`todo.task`模型时，您应该看到新的字段。

修改现有的字段

如您所见，将新字段添加到现有模型非常简单。从Odoo8开始，修改现有继承字段的属性也是可能的。它是通过添加一个具有相同名称和设置值的字段来实现的，只需要更改属性。

例如，为`name`字段添加一个帮助工具提示，我们将把这一行添加到`todo_task.py`中，前面描述过:

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

这将使用指定的属性修改字段，将未修改的所有其他属性保留在这里。如果我们升级模块，进入to-do任务表单并在**Description**字段上暂停鼠标指针;将显示工具提示文本。

修改模型方法

继承也适用于业务逻辑级别。添加新方法很简单:在继承类中声明它们的函数。

要扩展或更改现有逻辑，可以通过使用完全相同的名称声明方法来覆盖相应的方法。新方法将取代以前的方法，而且它还可以扩展继承类的代码，使用Python的`super()`方法调用父方法。它可以在调用`super()`方法之前和之后，在原有的逻辑上添加新的逻辑。

最好避免更改方法的函数签名(也就是说，保留相同的参数)，以确保现有的调用将继续正常工作。如果需要添加额外的参数，请将它们设置为可选的关键字参数(带有默认值)。

最初的**Clear All Done**操作不适合我们的任务共享模块，因为它清除了所有的任务，而不考虑它们的用户。我们需要修改它，使它只清除当前的用户任务。

为此，我们将重写(或替换)原来的方法，使用一个新版本，它首先找到当前用户完成的任务列表，然后激活它们:

```
@api.multi
def do_clear_done(self):
    domain = [('is_done', '=', True),
              '|', ('user_id', '=', self.env.uid),
                   ('user_id', '=', False)]
    dones = self.search(domain)
    dones.write({'active': False})
    return True
```

为了清晰起见，我们首先构建过滤器表达式，用于查找要清除的记录。

这个过滤器表达式遵循odoo特有的语法，称为`domain`:它是一个条件列表，其中每个条件都是一个元组。
这些条件隐式地与AND (`&`)操作符连接。对于OR操作，一个管道，`|`，被用在一个元组的地方，并且它加入下两个条件。我们将在第6章中详细介绍一些域，视图-设计用户界面。

这里使用的域筛选了所有已完成的任务(`'is_done', '=', True`)，它们要么有当前用户作为负责的(`'user_id', '=', self.env.uid`)，要么没有当前用户设置(`'user_id', '=', False`)。

然后，我们使用`search`方法将记录集与所做的记录进行操作，最后，在它们上做一个批量写入，将`active`字段设置为`False`。这里的Python `False`值表示数据库`NULL`值。

在这种情况下，我们完全重写了父方法，用新的实现替换它，但这不是我们通常想要做的。相反，我们应该将现有的逻辑扩展到一些额外的操作。否则，我们可能会破坏已经存在的特性。
为了让重写方法保留已经存在的逻辑，我们使用Python的`super()`构造调用该方法的父版本。让我们来看一个例子。

我们可以改进`do_toggle_done()`方法，使它只对分配给当前用户的任务执行操作。这是实现这一目标的代码:

```
from odoo.exceptions import ValidationError
# ...
# class TodoTask(models.Model):
# ...
@api.multi
def do_toggle_done(self):
    for task in self:
        if task.user_id != self.env.user:
            raise ValidationError(
                'Only the responsible can do this!')
    return super(TodoTask, self).do_toggle_done()
```

继承类中的方法从一个`for`循环开始，检查是否要切换到另一个用户的任务。如果这些检查通过，它将使用`super()`调用父类方法。如果没有提出错误，我们应该使用这个Odoo构建-在异常中。最相关的是`ValidationError`，这里使用，`UserError`。

这些是用于覆盖和扩展模型类中定义的业务逻辑的基本技术。接下来，我们将了解如何扩展用户界面视图。

扩展的观点

表单、列表和搜索视图都是使用arch XML结构定义的。为了扩展视图，我们需要一种方法来修改此XML。这意味着定位XML元素，然后在这些点上进行修改。

继承的视图允许这样做。继承的视图声明如下:

```
<record id="view_form_todo_task_inherited"
  model="ir.ui.view">
  <field name="name">Todo Task form - User
    extension</field>
  <field name="model">todo.task</field>
  <field name="inherit_id"
    ref="todo_app.view_form_todo_task"/>
  <field name="arch" type="xml">
    <!-- ...match and extend elements here! ... -->
  </field
</record>
```

`inherit_id`字段通过使用特殊的`ref`属性来表示其外部标识符，从而将视图扩展。外部标识符将在第四章模块数据中详细讨论。

作为XML，定位XML元素的最佳方法是使用XPath表达式。例如，使用前一章中定义的表单视图，一个用于定位`<field name="is_done">`元素的XPath表达式是`//field[@name]='is_done'`。这个表达式可以找到任何带有`name`属性的`field`元素，它等于`is_done`。你可以找到关于XPath的更多信息：`https://docs.python.org/2/library/xml.etree.elementtree.html#xpath-support`.

如果XPath表达式匹配多个元素，则只有第一个元素会被修改。因此，它们应该尽可能地具体化，使用惟一的属性。使用`name`属性是确保我们找到要使用扩展点的准确元素的最简单方法。因此，在我们的视图XML元素中设置它们是很重要的。

一旦扩展点被定位，您可以修改它或者在它附近添加XML元素。作为一个实际示例，在`is_done`字段之前添加`date_deadline`字段，我们将在`arch`中编写以下代码:

```
<xpath expr="//field[@name]='is_done'" position="before">
  <field name="date_deadline" />
</xpath>
```

幸运的是 ,Odoo为这个提供了快捷方式 ,因此大多数时候我们完全可以避免使用XPath语法。与前面的XPath元素相反，我们可以使用与元素类型类型相关的信息来定位和它的独特属性，而不是前面的XPath，我们这样写:

```
<field name="is_done" position="before">
  <field name="date_deadline" />
</field>
```

请注意，如果字段在同一视图中出现不止一次，那么您应该始终使用XPath语法。这是因为Odoo将在字段的第一次出现时停止，它可能会将您的更改应用到错误的字段。

通常，我们希望在现有的字段旁边添加新字段，因此`<field>`标记通常会被用作定位器。但是任何其他标记都可以使用: `<sheet>`，`<group>`，`<div>`，等等。名称属性通常是匹配元素的最佳选择，但有时，我们可能需要使用其他东西:例如CSS `class`元素。Odoo将会找到第一个元素，它至少包含了所有指定的属性。

在9.0版本之前，字符串属性(用于显示的标签文本)也可以用作扩展定位器。从9.0开始，这是不允许的。这个限制与在这些字符串上运行的语言转换机制有关。

定位器元素使用的`position`属性是可选的，可以具有以下值:

`after`将内容添加到父元素，在匹配的节点之后。

`before`在匹配的节点之前添加内容。

`inside` (默认值)附加在匹配节点内的内容。

`replace` 替换匹配的节点。如果使用空内容，则删除一个元素。由于Odoo 10还允许将一个元素与其他标记包在一起，通过在内容中使用$0来表示被替换的元素。

`attributes` 修改匹配元素的XML属性。这是在元素内容`<attribute name="attr-name">`元素和新的属性值设置中使用的。

例如，在任务表单中，我们有`active`字段，但是让它可见是没有用的。我们可以从用户那里隐藏它。这可以通过设置`invisible`属性来实现:

```
<field name="active" position="attributes">
  <attribute name="invisible">1</attribute>
</field>
```

设置`invisible`属性以隐藏元素是使用`replace`定位器删除节点的好方法。应该避免删除节点，因为它可以根据被删除的节点作为一个占位符来添加其他元素。

扩展form视图

将所有以前的表单元素组合在一起，我们可以添加新字段并隐藏`active`。扩展to-do任务表单的完整继承视图如下:

```
<record id="view_form_todo_task_inherited"
  model="ir.ui.view">
  <field name="name">Todo Task form - User
    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" position="after">
      <field name="user_id">
    </field>
    <field name="is_done" position="before">
      <field name="date_deadline" />
    </field>
    <field name="active" position="attributes">
      <attribute name="invisible">1</attribute>
    </field>
  </field>
</record>
```

这应该被添加到我们模块中的`views/todo_task.xml`文件中，在`<odoo>`元素中，如前一章所示。

继承的视图也可以继承，但是由于这会产生更复杂的依赖关系，所以应该避免。只要可能，您应该倾向于从最初的视图继承。

另外，我们不应该忘记将数据属性添加到`_manifest_.py`描述符文件中:

```
'data': ['views/todo_task.xml'],
```

扩展tree和search视图

Tree和search视图扩展也被定义为使用`arch` XML结构，它们可以以与form视图相同的方式扩展。我们将通过扩展list和search视图来继续我们的示例。

对于list视图，我们想要将`user`字段添加到它:

```
<record id="view_tree_todo_task_inherited"
  model="ir.ui.view">
  <field name="name">Todo Task tree - User
  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="user_id" />
    </field>
  </field
</record>
```

对于search视图，我们将为用户自己的任务和未分配给任何人的任务添加用户和预定义过滤器的搜索:

```
<record id="view_filter_todo_task_inherited"
  model="ir.ui.view">
  <field name="name">Todo Task tree - User
    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="user_id" />
        <filter name="filter_my_tasks" string="My Tasks"
          domain="[('user_id','in',[uid,False])]" />
        <filter name="filter_not_assigned" string="Not
          Assigned" domain="[('user_id','=',False)]" />
      </field>
    </field
  </record>
```

不要过于担心这些视图的特定语法。我们将在第6章中详细介绍它们，视图——设计用户界面。

更多的模型继承机制

我们已经看到了模型的基本扩展，即官方文档中的**class inheritance**。这是最常见的继承，最容易想到的是**in-place extension**。你取一个模型并扩展它。当您添加新特性时，它们被添加到现有模型中。一个新的模型没有被创建。我们还可以继承多个父模型，将值列表设置为`_inherit`属性。有了这个，我们可以使用**mixin classes**。Mixin classes是实现泛型特性的模型，我们可以添加到其他模型中。它们不被期望直接使用，而且就像一个准备被添加到其他模型的特性容器。

如果我们还使用与父模型不同的`_name`属性，我们将得到一个新的模型，该模型重用继承的特性，但使用它自己的数据库表和数据。官方文件称此为**prototype inheritance**。在这里，你拿一个模型，创建一个全新的，它是旧版本的副本。当您添加新特性时，它们被添加到新模型中。现有的模型没有改变。

还有**delegation inheritance**方法，使用`_inherits`属性。它允许模型以透明的方式包含其他模型，而在幕后，每个模型都处理自己的数据。你取一个模型并扩展它。当您添加新特性时，它们被添加到新模型中。现有模块没有更改。新模型中的记录与原始模型中的记录有关联，原始模型的字段被公开，可以直接在新模型中使用。

让我们更详细地探讨这些可能性。

复制具有原型继承的特性

我们在扩展模型之前使用的方法仅使用_inherit属性。我们定义了一个继承了todo.task模型并添加了一些特性的类。类属性_name没有显式设置;隐式,这是todo.task。

但是，使用_name属性允许我们创建一个新的模型，复制继承的特性。这是一个例子:

```
from odoo import models
class TodoTask(models.Model):
    _name = 'todo.task'
    _inherit = 'mail.thread'
```

这扩展了todo.task模型，将其复制到mail.thread模型的特性中。mail.thread模型实现了Odoo消息和追随者特性，并且是可重用的，因此可以很容易地将这些特性添加到任何模型中。

复制意味着继承的方法和字段也将在继承模型中可用。对于字段，这意味着它们也将被创建并存储在目标模型的数据库表中。原始(继承的)和新(继承)模型的数据记录是不相关的。只有定义是共享的。

稍后，我们将详细讨论如何使用此功能将mail.thread及其社交网络特性添加到模块中。在实践中，当使用mixin时，我们很少从常规模型中继承，因为这会导致相同数据结构的重复。

Odoo还提供了一种委托继承机制，避免了数据结构的重复，因此在继承常规模型时，它通常是首选的。让我们更详细地看一下。

使用委托继承嵌入模型

委托继承的使用较少，但它可以提供非常方便的解决方案。它通过使用字典映射继承的模型和连接到它们的字段来使用_inherits属性(注意附加的 s)。

一个很好的例子就是标准用户模型，`res.users`;它有一个嵌入的合作伙伴模型:

```
from odoo import models, fields
class User(models.Model):
    _name = 'res.users'
    _inherits = ('res.partner': 'partner_id'}
    partner_id = fields.Many2one('res.partner')
```

有了授权继承，`res.users`模型嵌入了继承的模型`res.partner`，这样当创建一个新的`User`类时，也会创建一个合作伙伴，并将它保存在`User`类的`partner_id`字段中。它与面向对象编程中的多态性概念有一些相似之处。

通过委托机制，从继承的模型和伙伴的所有字段都可用，就像它们是`User`字段一样。例如，伙伴`name`和`address`字段被公开为`User`字段，但实际上，它们被存储在链接的合作伙伴模型中，并且没有发生数据重复。

与原型继承相比，这一点的优点是，无需重复数据结构，比如多个表中的地址。任何需要包含地址的新模型都可以将其委托给嵌入的伙伴模型。如果在伙伴地址字段中引入了修改，那么这些都可以立即应用到所有嵌入它的模型中!

请注意，带授权继承的字段是继承的，但方法不是。

添加社交网络功能

社交网络模块(技术名称`mail`)提供了在许多表单底部和**Followers**功能底部发现的消息板，以及关于消息和通知的逻辑。这是我们经常想要添加到我们的模型中，所以让我们学习如何去做。

社交网络信息功能是由`mail`模块的`mail.thread`模型提供的。要将其添加到自定义模型，我们需要执行以下操作:

模块依赖于`mail`吗

有从`mail.thread`继承的类吗

在窗体视图中添加了追随者和线程小部件吗

我们还需要为关注者建立记录规则。

让我们按照这个清单。

关于第一点，我们的扩展模块需要对模块`_manifest_.py`清单文件额外的`mail`依赖性:

```
'depends': ['todo_app', 'mail'],
```

关于第二点，`mail.thread`的继承是使用我们以前使用的`_inherit`属性来完成的。但是我们的to-do任务扩展类已经使用了`_inherit`属性。幸运的是，它可以接受一个模型列表来继承，因此我们可以使用它来使它也包括`mail.thread`的继承:

```
_name = 'todo.task'
_inherit = ['todo.task', 'mail.thread']
```

`mail.thread`是一个抽象模型。**Abstract models**和常规模型一样，只不过它们没有数据库表示;没有为它们创建实际的表。抽象模型并不意味着直接使用。相反，它们将被用作mixin类，正如我们刚才所做的那样。我们可以把它们看作是具有现成功能的模板。为了创建一个抽象类，我们只需要使用`models.AbstractModel`而不是`models.Model`来定义它们。

对于第三点，我们希望将社交网络小部件添加到表单的底部。这是通过扩展窗体视图定义来完成的。我们可以重用已经创建的继承视图`view_form_todo_task_inherited`，并将其添加到它的`arch`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>
```

这里添加的两个字段并没有被我们明确声明，但是它们是由`mail.thread`模型提供的。

最后一步，即第4步，是为追随者设置记录规则:行级访问控制。只有当我们的模型被要求限制其他用户访问记录时，才需要这样做。在这种情况下，我们希望每个to-do任务记录也能被它的任何追随者看到。

我们已经在to-do任务模型上定义了一个记录规则，因此我们需要修改它来添加新的需求。这是我们下一节要做的事情之一。

修改数据

与视图不同，常规数据记录没有XML `arch`结构，不能使用XPath表达式进行扩展。但它们仍然可以被修改，以替换字段中的值。

`<record id="x" model="y">`数据加载元素实际上在模型y上执行`insert`或`update`操作:如果model x不存在，则创建;否则，它将被更新/写入。

由于其他模块中的记录可以使用`<model>.<identifier>`全局标识符访问，所以我们的模块可以覆盖其他模块之前编写的内容。

请注意，由于点是预留的，以将模块名与对象标识符分隔开来，因此不能在标识符名称中使用它。相反，使用下划线选项。

修改菜单和动作记录

例如，让我们将`todo_app`模块创建的菜单选项改为`My To-Do`。为此，我们可以将以下内容添加到`todo_user/views/todo_task.xml`文件:

```
<!-- Modify menu item -->
<record id="todo_app.menu_todo_task" model="ir.ui.menu">
  <field name="name">My To-Do</field>
</record>
```

我们还可以修改菜单项中使用的操作。动作有一个可选的上下文属性。它可以为视图字段和过滤器提供默认值。我们将使用它在默认情况下启用**My Tasks**过滤器，在本章前面定义:

```
<!-- Action to open To-Do Task list -->
<record model="ir.actions.act_window"
  id="todo_app.action_todo_task">
  <field name="context">
    ('search_default_filter_my_tasks': True}
  </field>
</record>
```

修改安全记录规则

To-Do应用程序包括一个记录规则，以确保每个任务只对创建它的用户可见。但是现在，随着社会功能的增加，我们也需要任务的追随者来访问它们。社交网络模块本身并没有处理这个问题。

另外，现在任务可以让用户分配给他们，因此，让访问规则来处理负责任的用户而不是创建任务的用户更有意义。
该计划与我们为菜单项所做的一样:覆盖`todo_app.todo_task_user_rule`，将`domain_force`字段修改为一个新值。

该公约是在安全`security`中保留与安全相关的文件，因此我们将创建一个`security/todo_access_rules.xml`文件，内容如下:

```
<?xml version="1.0" encoding="utf-8"?>
<odoo>
  <data noupdate="1">
    <record id="todo_app.todo_task_per_user_rule"
      model="ir.rule">
      <field name="name">ToDo 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">
        ['|',('user_id','in', [user.id,False]),
        ('message_follower_ids','in',
        [user.partner_id.id])]
      </field>
    </record>
  </data>
</odoo>
```

这覆盖`todo_task_per_user_rule`记录规则，从`todo_app`模块。新的域筛选器现在可以让负责用户、`user_id`或对每个人都可见的任务，如果负责的用户没有设置(等于`False`);所有的任务追随者都可以看到它。

记录规则运行在可用的`user`变量的上下文中，并表示当前会话用户的记录。因为追随者是合作伙伴,不是`users`, `user.id`,相反,我们需要使用`user.partner_id`。

groups字段是一个to-many关系。在这些字段中编辑数据使用一种特殊的符号。这里使用的代码4是附加到相关记录的列表。通常使用的是代码6，取而代之的是将相关记录完全替换为一个新列表。我们将在第四章，模块数据中详细讨论这个符号。

记录元素的`noupdate="1"`属性意味着这个记录数据只会写在安装操作上，并且在模块升级时将被忽略。这允许它是自定义的，不需要冒着过度编写定制的风险，并且在将来某个时候进行模块升级时丢失它们。

在开发时使用`<data noupdate="1">`处理数据文件可能会很麻烦,因为以后对XML定义的编辑会被模块升级所忽略。为了避免这种情况，可以重新安装模块。 这更容易通过使用 `-i` 的命令行完成

像往常一样，我们不能忘记将新文件添加到`__manifest__.py`的数据属性中:

```
'data': ['views/todo_task.xml', 'security/todo_access_rules.xml'],
```

摘要

现在，您应该能够通过扩展现有模块来创建自己的模块。

为了演示如何做到这一点，我们扩展了在前一章中创建的To-Do模块，并向构成应用程序的几个层添加新特性。

我们扩展了一个Odoo模型来添加新字段并扩展其业务逻辑方法。接下来，我们修改视图，使新字段可用。最后，我们学习了如何从其他模型继承特性，并使用它们将社交网络功能添加到To-Do应用程序。

前三章概述了Odoo开发的常见活动，从Odoo安装到模块创建和扩展。

下一章将重点讨论Odoo发展的一个具体领域 ,我们在第一章中简要介绍了其中的大部分内容。在接下来的章节中，我们将更详细地讨论数据序列化以及XML和CSV数据文件的使用。

4

模块数据

大多数Odoo模块定义，比如用户接口和安全规则，实际上都是存储在特定数据库表中的数据记录。在运行时，在模块中发现的XML和CSV文件不被Odoo应用程序使用。相反，它们是将这些定义加载到数据库表中的一种方法。

因此，Odoo模块的一个重要部分是关于使用文件来表示(序列化)数据，以便以后可以加载到数据库中。

模块也可以有默认和演示数据。数据表示允许将其添加到我们的模块中。此外，了解Odoo数据表示格式对于在项目实现的上下文中导出和导入业务数据非常重要。

在我们进入实际案例之前，首先探索外部标识符概念，这是Odoo数据表示的关键。

了解外部标识符

external identifier (也称为XML ID)是一个可读的字符串标识符，它唯一地标识了Odoo中的特定记录。当将数据加载到Odoo时，它们非常重要。

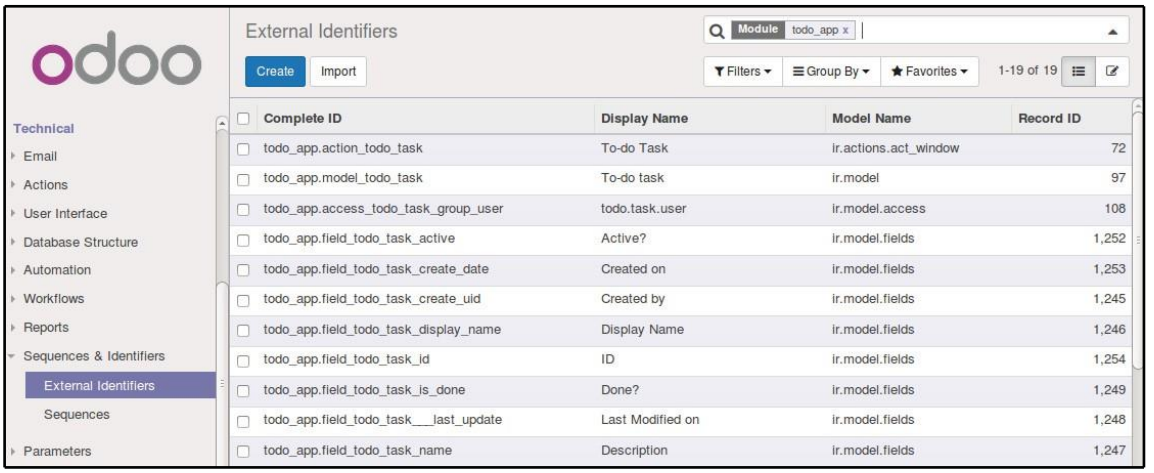
其中一个原因是当升级一个模块时，它的数据文件将再次被加载到数据库中，我们需要检测已经存在的记录，以便更新它们，而不是创建新的重复记录。

支持相关数据的另一个原因:数据记录必须能够引用其他数据记录。实际的数据库标识符是在模块安装期间由数据库自动分配的序号。外部标识符提供了一种方法来引用相关的记录，而不需要事先知道它将被分配的数据库ID，从而允许我们定义Odoo数据文件中的数据关系。

Odoo负责将外部标识符名称转换为分配给它们的实际数据库id。这背后的机制很简单:Odoo保存了一个表，表中有命名的外部标识符与其相应的数字数据库id之间的映射:
`ir.model.data`模型。

要检查现有的映射，请访问**Settings**顶部菜单的**Technical**部分，并在**Sequences & Identifiers**下选择“**External Identifiers**”菜单项。

例如，如果我们访问**External Identifiers**列表并通过`todo_app`模块过滤它，我们将看到前面创建的模块生成的外部标识符:

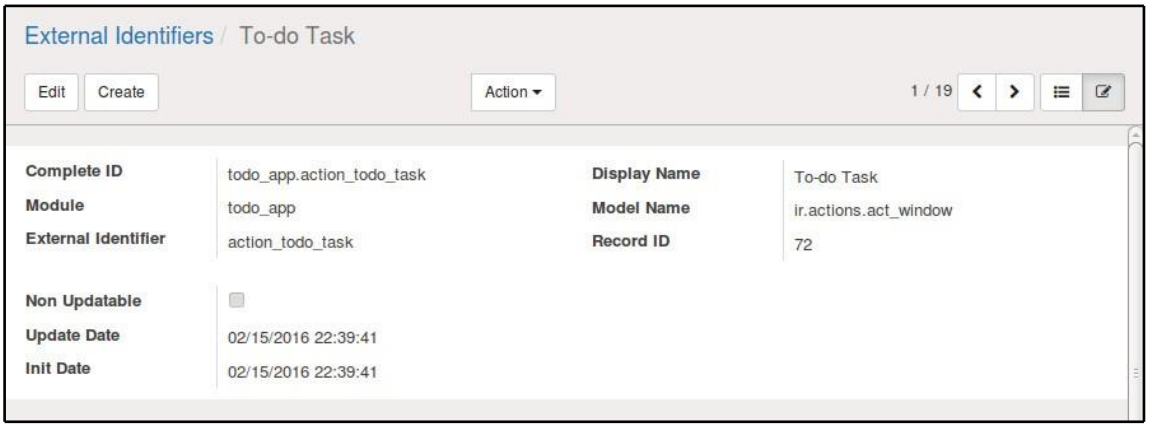


我们可以看到外部标识符有一个**Complete ID**标签。注意它是如何由模块名和标识符名称组成的，例如，`todo_app.action_todo_task`。

外部标识符只需要在一个Odoo模块内惟一，这样就不会有两个模块相互冲突的风险，因为它们不小心选择了相同的标识符。要构建一个全局惟一标识符，Odoo将模块名称与实际的外部标识符名称连接在一起。这是在**Complete ID**字段中可以看到的。

当在数据文件中使用外部标识符时，您可以选择使用完整的标识符或仅使用外部标识符名。通常使用外部标识符名称比较简单，但是完整的标识符使我们能够从其他模块引用数据记录。这样做时，请确保这些模块包含在模块依赖项中，以确保这些记录在我们之前已经加载。

在列表的顶部，我们有`todo_app.action_todo_task`完整的标识符。这是我们为模块创建的菜单操作，它也在相应的菜单项中引用。点击它，我们就会看到它的细节；`todo_app`模块中的`action_todo_task`外部标识符映射到`ir.actions.act_window`模型中的特定记录ID，在本例中为`72`:



除了为记录提供一种方便地引用其他记录的方法之外，外部标识符还允许您避免重复导入的数据重复。如果外部标识符已经存在，则将更新现有记录;您不需要创建一个新的记录。这就是为什么在后续的模块升级中，先前加载的记录是更新的而不是复制的。

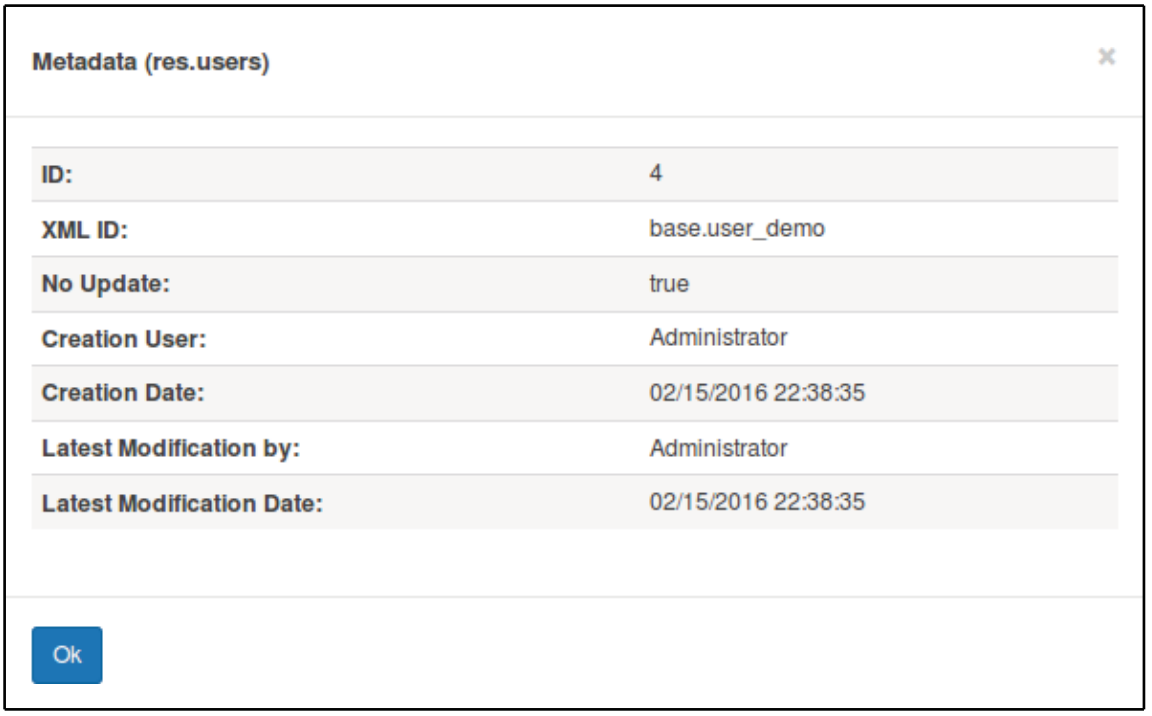
发现外部标识符

在为模块准备定义和演示数据文件时，我们经常需要查找引用中需要的现有外部标识符。

我们可以使用前面显示的**External Identifiers**菜单，但是**Developer**菜单可以提供更方便的方法。正如您可能在第1章中看到的，开始使用Odoo开发时，**Developer**菜单在**Settings**仪表板中被激活，在右下角的选项中。

要查找数据记录的外部标识符，在相应的表单视图中，从**Developer**菜单中选择**View Metadata**选项。这将显示一个与记录的数据库ID和外部标识符(也称为XML ID)的对话框。

举个例子，查看演示用户ID，我们可以导航到表单视图，**Settings**|**Users**，选择**View Metadata**选项，这将显示:



为了找到视图元素的外部标识符，例如表单、树、搜索或操作，**Developer**菜单也是一个很好的帮助来源。为此，我们可以使用它的**Manage Views**选项，或者使用**Edit <view type>**选项打开所需视图的信息。然后，选择他们的**View Metadata**选项。

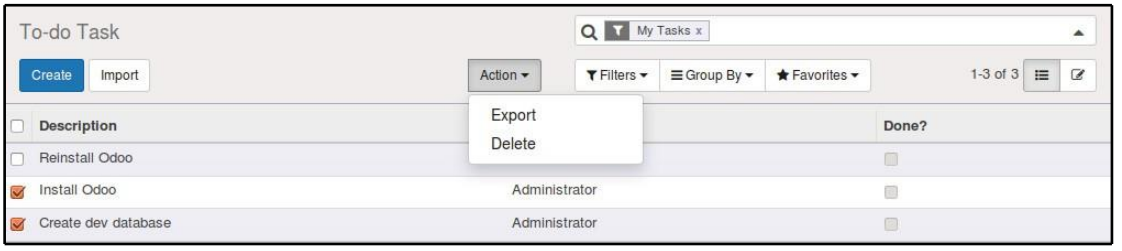
导出和导入数据

我们将开始探索如何从Odoo的用户界面导出和导入数据，然后我们将讨论如何在addon模块中使用数据文件的技术细节。

导出数据

数据导出是任何列表视图中可用的标准特性。要使用它，我们必须首先通过选择最左边的对应的复选框来选择要导出的行，然后从**More**按钮中选择**Export**选项。

这里有一个例子，使用最近创建的to-do任务:

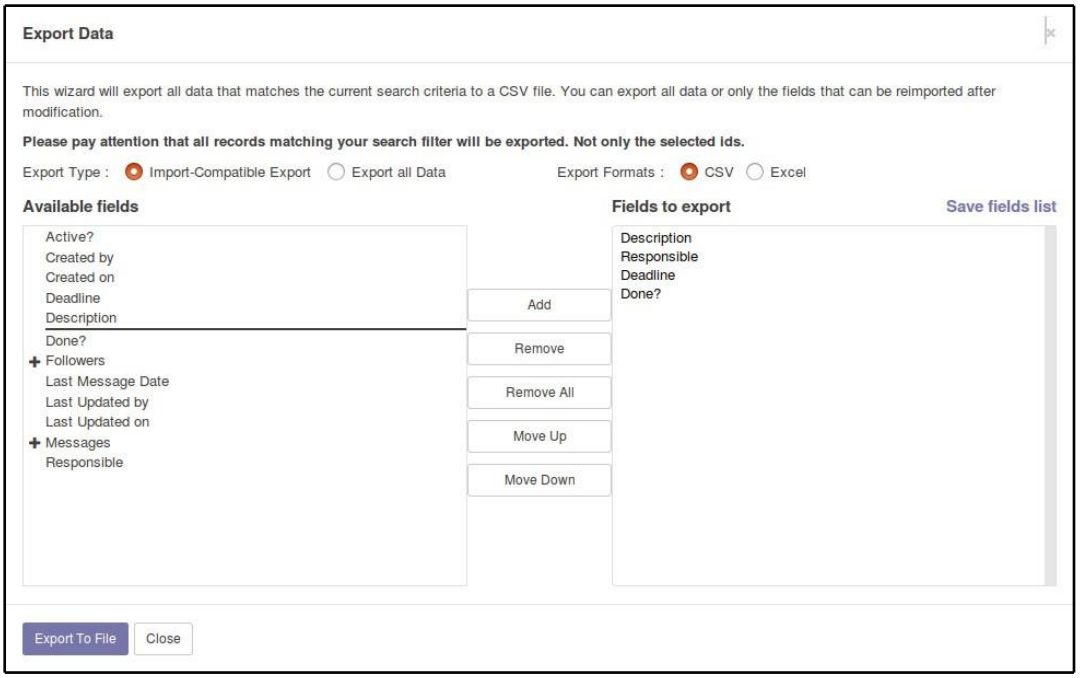


我们也可以勾选列标题中的复选框。它将同时检查所有的记录，并将导出符合当前搜索条件的所有记录。

在以前的Odoo版本中，只有在屏幕上看到的记录(当前页面)才能实际导出。从Odoo 9开始，在头文件中修改和勾勾的复选框将输出与当前筛选器匹配的所有记录，而不仅仅是当前显示的那些。这对于输出不适合屏幕的大量记录非常有用。

Export选项把我们带到了一个对话框，在那里我们可以选择输出什么。**Import-Compatible Export**选项确保导出的文件可以导入到Odoo。我们需要使用这个。

导出格式可以是CSV或Excel。我们希望CSV文件能更好地理解导出格式。接下来，选择要导出的列，并单击**Export To File**按钮。这将开始下载一个带有导出数据的文件:



如果我们遵循这些说明并选择前面截图中显示的字段，那么我们应该得到一个类似于此的CSV文本文件:

```
"id","name","user_id/id","date_deadline","is_done"
"todo_user.todo_task_a","Install
Odoo","base.user_root","2015-01-30","False"
" export .todo_task_9","Create my first
module","base.user_root","","False"
```

注意 ,Odoo自动导出一个额外的id列。这是分配给每个记录的外部标识符。如果没有指定的模块，则会自动生成使用`_export_`代替实际模块名称的新功能。新的标识符只被分配到没有一个的记录，而且从那里开始，它们被绑定到相同的记录。这意味着随后的导出将保留相同的外部标识符。

导入数据

首先，我们必须确保启用了导入特性。因为Odoo 9是默认启用的。如果不是，这个选项是可以从**Settings**级菜单，**General Settings**选项。在**Import**丨**Export**部分，有一个应该启用的**a Allow users to import data from CSV/XLS/XLSX/ODS files**复选框。

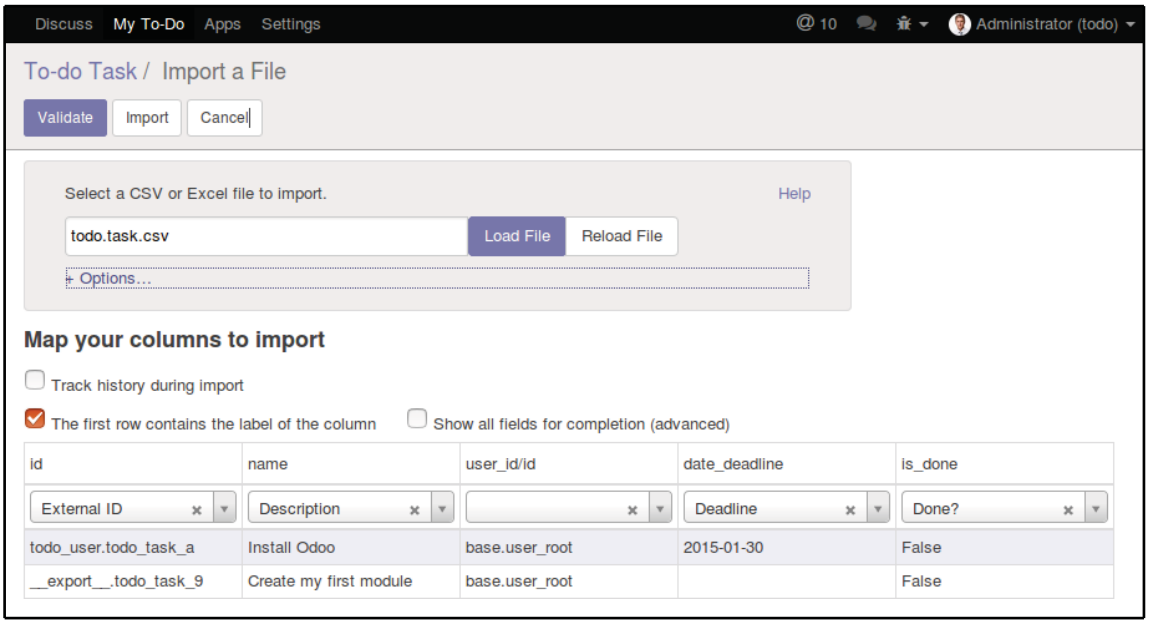
这个特性是由**Initial Setup Tools** addon提供的(`base_setup`是技术名称)。的实际效果bbb...复选框是安装或卸载`base_setup`。

有了这个选项，列表视图就显示了在列表顶部的**Create**按钮旁边的**Import**选项。

让我们先对我们的待办事项数据进行批量编辑。打开我们刚刚在电子表格或文本编辑器中下载的CSV文件，并更改一些值。另外，添加一些新行，将`id`列空出来。

如前所述，第一列`id`为每一行提供惟一标识符。这使得已经存在的记录可以更新，而不是在我们将数据导入Odoo时复制它们。对于添加到CSV文件的新行，我们可以选择提供我们选择的外部标识符，或者将空白`id`列留空，将为它们创建一个新的记录。
在将更改保存到CSV文件之后，单击**Import**选项(在**Create**按钮旁边)，我们将会看到导入助手。

在那里，我们应该选择磁盘上的CSV文件位置并单击**Validate**以检查其正确性的格式。因为要导入的文件是基于Odoo导出的，所以很有可能它是有效的:



现在我们可以点击**Import**，就这样;我们的修改和新记录本应载入Odoo。

CSV数据文件中的相关记录

在前面的示例中，负责每个任务的用户是用户模型中的一个相关记录，使用many-to-one (或一个外键)关系。它的列名是`user_id/id`，字段值是相关记录的外部标识符，例如管理员用户的`base.user_root`。

只建议在导出和导入相同的数据库时使用数据库id。通常，您倾向于使用外部标识符。

如果使用外部标识符或 `/.id` 使用数据库(数值)id，关系列应该有 `/id` 附加到它们的名称。或者，冒号(:)可以在相同效果的地方使用。

类似地，也支持many-to-many关系。many-to-many关系的一个例子是用户和组之间的关系:每个用户可以有许多组，每个组可以有许多用户。这种类型的字段的列名应该有`/id`附加。字段值接受以逗号分隔的外部标识符列表，并使用双引号包围。

例如，to-do任务关注者在待办事项和合作伙伴之间有many-to-many的关系。它的列名应该是`follower_ids/id`和一个字段值，有两个追随者可以是这样的:

```
" export .res_partner_1, export  .res_partner_2"
```

最后，还可以通过CSV导入一对多关系。这种类型关系的典型示例是带有几行代码的文档头。注意，one-to-many关系始终是many-to-one关系的逆。每个文档头可以有许多行。而且每条直线都有一个头。

我们可以在公司模型中看到这样一个关系的例子(在**Settings**菜单中有表单视图):每个公司都可以拥有多个银行帐户，每个帐户都有自己的详细信息;相反，每个银行账户记录都属于和只有一个公司的many-to-one关系。

我们可以把公司的银行账户连同他们的银行账户一起放在一个文件里。这里有一个例子，我们将一个公司和三家银行一起装入:

```
id,name,bank_ids/id,bank_ids/acc_number,bank_ids/state
base.main_company,YourCompany,_export__.res_partner_bank_4,123456789,bank
,,_export__.res_partner_bank_5,135792468,bank
,,_export__.res_partner_bank_6,1122334455,bank
```

我们可以看到前两列，`id`和`name`，在第一行中有值，在接下来的两行中是空的。他们有记录头记录的数据，也就是公司的数据。

其他三列都是带有`bank_ids/`的前缀，在这三行中都有值。他们为公司的银行账户提供了三个相关的数据。第一行有公司和第一家银行的数据，接下来的两行数据只针对额外的公司和银行。

这些都是与导出和从GUI导入的过程中的基本要素。在新的Odoo实例中设置数据或准备将数据文件包含在Odoo模块中是很有用的。接下来，我们将学习更多关于使用模块中的数据文件。

模块数据

模块使用数据文件将其配置加载到数据库、默认数据和演示数据中。这可以使用CSV和XML文件完成。对于完整性，也可以使用YAML文件格式，但它很少用于加载数据;因此，我们不会讨论这个问题。

模块使用的CSV文件与我们看到的和用于导入特性的CSV文件完全相同。当在模块中使用它们时，一个额外的限制是文件名必须与加载数据的模型的名称相匹配，这样系统就可以推断出应该导入数据的模型。

数据CSV文件的一个常见用法是访问安全定义，加载到`ir.model.access`模型中。他们通常使用命名为`ir.model.access.csv`的CSV文件。

演示数据

Odoo addon模块可以安装演示数据，这样做被认为是很好的实践。这有助于为测试中使用的模块和数据集提供使用示例。使用`____manifest__.py`清单文件的`demo`属性声明模块的演示数据。就像`data`属性一样，它是一个文件名列表，包含模块内相应的相对路径。
现在是向`todo_user`模块添加一些演示数据的时候了。我们可以先从to-do任务中导出一些数据，如前一节所解释的那样。该约定是将数据文件放置在`data/`子目录中。因此，我们应该将这些数据文件保存在`todo_user` addon模块中作为`data/todo.task.csv`。由于该数据将由我们的模块拥有，所以我们应该编辑id值以删除标识符中的`_export_`前缀。

例如，我们的`todo.task.csv`数据文件可能是这样的:

```
id,name,user_id/id,date_deadline
todo_task_a,"Install Odoo","base.user_root","2015-01-30"
todo_task_b","Create dev database","base.user_root",""
```

我们不能忘记将这个数据文件添加到`_manifest_.py`清单的`demo`属性:

```
'demo': ['data/todo.task.csv'],
```

下次我们更新模块时，只要它安装了已启用的演示数据，文件的内容将被导入。注意，每当执行模块升级时，这些数据将被重新导入。

XML文件也用于加载模块数据。让我们进一步了解XML数据文件可以做哪些CSV文件不能做的事情。

XML数据文件

虽然CSV文件提供了一种简单紧凑的格式来表示数据，但XML文件更强大，并对加载过程提供了更多的控制。它们的文件名不需要匹配要加载的模型。这是因为XML格式要丰富得多，而且该信息由文件中的XML元素提供。

我们已经在前几章中使用了XML数据文件。用户界面组件，如视图和菜单项，实际上是存储在系统模型中的记录。模块中的XML文件是用来将这些记录加载到服务器中的方法。
为了展示这一点，我们将向`todo_user`模块添加第二个数据文件，`data/todo_data.xml`，有以下内容:

```
<?xml version="1.0"?>
<odoo>
  <!-- Data to load -->
  <record model="todo.task" id="todo_task_c">
    <field name="name">Reinstall Odoo</field>
    <field name="user_id" ref="base.user_root" />
    <field name="date_deadline">2015-01-30</field>
    <field name="is_done" eval="False" />
  </record>
</odoo>
```

这个XML等价于我们刚才看到的CSV数据文件。

XML数据文件有一个`<odoo>` top元素，其中我们可以有几个与CSV数据行对应的`<record>`元素。

在版本9.0中引入了数据文件中的`<odoo>` top元素，并取代前者`<openerp>`标签。top元素中的`<data>`部分仍然被支持，但现在是可选的。事实上，现在是`<odoo>`和 `<data>`是等价的，因此我们可以使用其中的一个作为XML数据文件的top元素。

一个`<record>`元素有两个强制属性，即`model`和`id` (记录的外部标识符)，并包含一个`<field>`标记，用于每个字段的写入。
注意，字段名中的斜杠符号在这里是不可用的;我们不能使用`<field name="user_id/id">`。相反，`ref`特殊属性用于引用外部标识符。我们稍后将讨论关系到多个域的值。

数据noupdate属性

在重复数据加载时，从上一次运行中载入的记录将被重写。重要的是要记住，这意味着升级一个模块将覆盖数据库内部可能发生的任何手工更改。值得注意的是，如果视图被自定义修改，那么这些更改将随着下一个模块的升级而丢失。正确的方法是为我们需要的变更创建继承的视图，如第3章所述，继承-扩展现有的应用程序。

这种重新导入行为是默认的，但是它可以被更改，这样当一个模块升级时，一些数据文件记录就被保留了。这是由`<odoo>`或`<data>`元素的`noupdate="1"`属性完成的。这些记录将在安装addon模块时创建，但在随后的模块升级中，将不会对它们进行任何操作。

这允许您确保手动定制的自定义在模块升级中是安全的。它通常与记录访问规则一起使用，允许它们适应特定于实现的需求。

在同一个XML文件中可以有多个`<data>`部分。我们可以利用这一点来分离数据 ,只导入一个，使用`noupdate="1"`，并在每次升级中重新导入数据，使用`noupdate="0"`。

noupdate标志存储在每个记录的**External Identifier**信息中。使用**Non Updatable**复选框可以在**Technical**菜单中直接使用**External Identifier**表单手动编辑它。

在开发模块时，noupdate属性可能很棘手，因为稍后对数据做出的更改将被忽略。一个解决方案是，不再使用-u选项升级模块，而是使用-i选项重新安装它。使用-i选项从命令行重新安装，忽略数据记录上的noupdate标志。

在XML中定义记录

每个<record>元素都有两个基本属性、id和model，并包含为每个列赋值的<field>元素。如前所述，id属性对应于记录的外部标识符，而模型属性对应于写入记录的目标模型。<field>元素有几种不同的赋值方法。让我们详细地看一下它们。

设置字段值

<record>元素定义了一个数据记录，并包含了在每个字段上设置值的<field>元素。

field元素的name属性标识要写入的字段。

写入的值是元素内容:字段的开头和结束标记之间的文本。对于日期和日期，字符串有"YYYY-mm-dd"和"YYYY-mm-dd HH:MM:SS"将被正确转换。但是对于布尔字段，任何非空值都将被转换为True，而"0"和"False"值被转换为False。

从数据文件中读取布尔False值的方法在Odoo 10中得到了改进。在以前的版本中，任何非空值，包括"0"和"False"都被转换为True。对于使用下面讨论的eval属性的布尔，建议使用。

使用表达式设置值

定义字段值的一种更为复杂的方法是`eval`属性。它计算Python表达式并将结果值赋给该字段。

这个表达式是在一个context中被评估的，除了Python内置的外，还有一些额外的标识符可用。让我们看一看。

为了处理日期，下列模块可用: `time`, `datetime`, `timedelta`,和`relativedelta`。它们允许您计算日期值，这是在演示和测试数据中经常使用的，因此使用的日期接近于模块安装日期。例如，要设置一个值到昨天，我们将使用这个:

```
<field name="date_deadline"
  eval="(datetime.now() + timedelta(-1)).strftime('%Y-%m-%d')" />
```

在评估上下文中还有`ref()`函数，它用于将外部标识符转换为相应的数据库ID，可用于为关系字段设置值。例如，我们以前使用它来为`user_id`设置值:

```
<field name="user_id" eval="ref('base.group_user')" />
```

为关系字段设置值

我们刚刚看到了如何在一个many-to-one关系字段(比如`user_id`)中设置一个值，使用带有`ref()`功能的`eval`属性。但还有一种更简单的方法。

`<field>`元素还具有一个`ref`属性，可以使用外部标识符来设置many-to-one字段的值。有了这个，我们就可以为`user_id`设置值:

```
<field name="user_id" ref="base.user_demo" />
```

对于one-to-many和many-to-many的字段，需要一个相关id列表，因此需要不同的语法;Odoo在这类字段上提供了一种特殊的语法。

下面的示例从官方的Fleet应用程序中，替换了一个`tag_ids`字段的相关记录列表:

```
<field name="tag_ids"
  eval="[(6,0,
    [ref('vehicle_tag_leasing'),
    ref('fleet.vehicle_tag_compact'),
    ref('fleet.vehicle_tag_senior')]
  )]" />
```

要在一个多字段上写，我们使用一个三元组列表。每个三重是一个写命令，根据使用的代码做不同的事情:

`(0,_ ,('field': value})` 创建一个新的记录并将其链接到这个记录

`(l,id,('field': value})` 更新已链接的记录上的值

`(2,id,_)` 取消链接并删除相关记录

`(3,id,_)` 取消链接，但不删除相关记录

`(4,id,_)` 链接一个已经存在的记录

`(5,_,_)` 取消链接，但不会删除所有链接的记录

`(6,_,[ids])` 用提供的列表替换链接记录的列表

前面列表中使用的下划线符号代表无关的值，通常填充为`0`或`False`。

常用模型的快捷方式

如果我们回到第2章，构建您的第一个Odoo应用程序，我们将在XML文件中找到除`<record>`之外的元素，比如`<act_window>`和`<menuitem>`。

对于经常使用的经常使用`<record>`元素的模型，这些都是方便快捷的快捷方式。他们将数据加载到支持用户界面的基础模型中，稍后将在第6章中详细讨论——设计用户界面。

作为参考，下列快捷方式可以使用相应的模型加载数据到:

`<act_window>`是窗口操作模型`ir.actions.act_window`

`<menuitem>`是菜单项的模型，`ir.ui.menu`

`<report>`是报告操作模型，`ir.actions.report.xml`

`<template>`是用于存储在模型`ir.ui.view`中的QWeb模板

`<url>`是URL操作模型`ir.actions.act_url`

XML数据文件中的其他操作

到目前为止，我们已经了解了如何使用XML文件添加或更新数据。但是XML文件还允许执行其他类型的操作，这些操作有时需要设置数据。特别是，他们可以删除数据，执行任意的模型方法，并触发工作流事件。

删除记录

要删除数据记录，我们使用`<delete>`元素，为它提供ID或搜索域以查找目标记录。例如，使用搜索域来查找记录的删除如下:

```
<delete
  model="ir.rule"
  search="
  [('id','=',ref('todo_app.todo_task_user_rule'))]"
/>
```

因为在这种情况下，我们知道要删除的特定ID，我们可以直接使用它来达到同样的效果:

```
<delete model="ir.rule" id="todo_app.todo_task_user_rule" />
```

触发功能和工作流程

XML文件还可以通过`<function>`元素在其加载过程中执行方法。这可以用来设置演示和测试数据。例如，CRM应用程序使用它来建立演示数据:

```
<function
  model="crm.lead"
  name="action_set_lost"
  eval="[ref('crm_case_7'), ref('crm_case_9')
        , ref('crm_case_11'), ref('crm_case_12')]
        , {'install_mode': True}" />
```

这调用了`crm.lead`模型的`action_set_lost`方法，通过`eval`属性传递了两个参数。第一个是要使用的id列表，接下来是要使用的上下文。

XML数据文件可以执行操作的另一种方式是通过`<workflow>`元素触发Odoo工作流。例如，工作流可以更改销售订单的状态，或者将其转换为发票。`sale`应用不再使用工作流，但这个示例仍然可以在演示数据中找到:

```
<workflow model="sale.order"
  ref="sale_order_4"
  action="order_confirm" />
```

现在，`model`属性是不言自明的，`ref`标识了我们正在执行的工作流实例。`action`是发送到这个工作流实例的工作流信号。

摘要

您已经了解了关于数据序列化的所有要点，并更好地理解了前面章节中看到的XML方面。我们还花了一些时间来理解外部标识符，特别是在一般的数据处理和模块配置中的核心概念。详细解释了XML数据文件。您了解了用于在字段上设置值和执行操作的几个选项，比如删除记录和调用模型方法。还解释了CSV文件和数据导入/导出特性。这些都是Odoo初始设置或对数据进行大规模编辑的宝贵工具。

在下一章中 ,我们将详细介绍如何构建Odoo模型 ,并了解更多关于构建用户界面的知识。

5

模型—构造应用程序数据

在前面的章节中，我们有一个关于为Odoo创建新模块的端到端概述。在第2章，构建您的第一个Odoo应用程序，我们构建了一个全新的应用程序，在第3章，继承—扩展现有的应用程序，我们探索了继承，以及如何使用它为我们的应用程序创建一个扩展模块。在第4章模块数据中，我们讨论了如何向模块中添加初始和演示数据。

在这些超视图中，我们涉及到构建Odoo的后端应用程序的所有层。现在，在接下来的章节中，是时候解释这些层，它们组成了更详细的应用程序:模型、视图和业务逻辑。

在本章中，您将了解如何设计支持应用程序的数据结构，以及如何表示它们之间的关系。

将应用程序特性组织成模块

和以前一样，我们将使用一个示例来帮助解释概念。

Odoo的继承特性提供了一个有效的扩展机制。它允许你扩展现有的第三方应用而不直接改变它们。这种可组合性还允许以模块为导向的开发模式，大型应用程序可以分成更小的功能，足够丰富，可以独立运行。

这有助于限制复杂性，无论是在技术层面还是在用户体验层面上。从技术的角度来说，将一个大问题分解成更小的部分可以更容易地解决问题，并且对增量特性的开发更加友好。从用户体验的角度来看，我们可以选择只激活他们真正需要的功能，以简化用户界面。因此，我们将通过附加的addon模块改进我们的To-Do应用程序，以最终形成一个完整的应用程序。

引入todo_ui模块

在前面的章节中,我们首先创建了一个应用程序为个人行动计划,然后扩展它的行动计划可以与他人共享。

现在我们想通过改进它的用户界面，包括一个看板，把我们的应用程序带到下一个层次。看板是一个简单的工作流工具，它组织列中的项目，这些项目从左栏向右流动，直到完成。我们将把我们的任务组织成列，根据他们的阶段，例如**Waiting**，**Ready**，**Started**，或者**Done**。

我们将从添加数据结构开始，以实现这一愿景。我们需要添加阶段，也很好地添加对标记的支持，让任务按主题分类。在本章中，我们将只关注数据模型。这些特性的用户界面将在第6章中讨论，视图——设计用户界面，以及在第9章，QWeb和看板视图中的看板视图。

首先要弄清楚的是我们的数据是如何构建的，这样我们就能设计出支持模型。我们已经有了中心实体:待办事项。每个任务将一次处于一个阶段，任务也可以有一个或多个标记。我们将需要添加这两个额外的模型，它们将拥有这些关系:

每个任务都有一个阶段，每个阶段都有很多任务
每个任务可以有多个标记，每个标记可以附加到许多任务

这意味着任务具有与多个阶段的many-to-one关系，以及与标签的many-to-many关系。另一方面，反向关系是:阶段与任务和标记之间的关系是一个many-to-many关系。

我们将从创建新的`todo_ui`模块开始，并添加任务阶段和待办事项标签模型。

我们已经使用`~/odoo-dev/custom-addons/`目录来托管我们的模块。我们应该为新的addons创建一个新的`todo_ui`目录。从shell中，我们可以使用以下命令:

```
$ cd ~/odoo-dev/custom-addons
$ mkdir todo_ui
$ cd todo_ui
```

我们开始添加`_manifest_.py`清单文件，其中包含以下内容:

```
(
  'name': 'User interface improvements to the To-Do app',
  'description': 'User friendly features.',
  'author': 'Daniel Reis',
  'depends': ['todo_user'] }
```

我们还应该添加一个`_init_.py`文件。现在空着是完全可以的。

现在我们可以在Odoo工作数据库中安装这个模块，并开始使用这些模型。

创建模型

对于有看板的to-do级任务，我们需要阶段。阶段是板列，每一个任务都适合于其中一个列:

编辑`todo_ui/ init .py`导入`models`子模块:

```
from . import models
```

创建`todo_ui/models`目录并添加一个`_init_.py`文件:

```
from . import todo_model
```

现在让我们添加`todo_ui/models/todo_model.py` Python代码文件:

```
# -*- coding: utf-8 -*-
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')
```

在这里，我们创建了两个新模型，这些模型将在待办事项中被引用。

专注于任务阶段，我们有一个Python类, `Stage`，基于`models.Model`类，它定义了一个叫做`todo.task.stage`的新的Odoo模型。我们还有两个字段: `name`和`sequence`。我们可以看到一些新到我们的模型属性(前缀加上下划线)。让我们仔细看看。

模型属性

模型类可以使用其他属性来控制它们的某些行为。这些是最常用的属性:

`_name` 我们正在创建的Odoo模型的内部标识符。在创建新模型时必须强制执行。

`_description` 为模型的记录提供一个用户友好的标题，当模型在用户界面中被查看时显示。可选的,但是建议您这样做。

`_order` 设置当模型的记录被浏览或在列表视图中显示时使用的默认顺序。它是一个作为SQL order by子句的文本字符串 ,因此它可以是您可以在那里使用的任何东西，尽管它有一个聪明的行为并支持可翻译的和many-to-one个字段名。

为了完整性起见，在高级案例中还可以使用更多的属性:

> `_rec_name` 指示字段作为记录描述在相关字段引用时使用，例如many-to-one关系。默认情况下，它使用`name`字段，该字段是模型中常见的字段。但是这个属性允许我们使用任何其他字段来实现这个目的。
>
> `_table`is 支持模型的数据库表的名称。通常情况下，它是自动计算的，并且是模型名称，用点替换的点。但是可以设置指定一个特定的表名。

我们也可以有`_inherit`和`_inherits`属性，如第3章中解释的，继承-扩展现有的应用程序。

模型和Python类

Odoo模型由Python类表示。在前面的代码中，我们有一个基于`models.Model`类的Python类`Stage`，它定义了一个叫做`todo.task.stage`的新的Odoo模型。

Odoo模型被保存在一个中央注册中心，也被称为`pool`在旧的Odoo版本。它是一个字典，它可以引用实例中所有可用的模型类，并且可以通过模型名称引用它。具体地说，模型方法中的代码可以使用`self.env['x']`获得一个代表model `x`的类的引用。

您可以看到模型名称很重要，因为它们是访问注册表所用的键。模型名称的约定是使用与点相连的小写字母的列表，例如`todo.task.stage`。核心模块的其他例子是`project.project`, `project.task`,或`project.task.type`。我们应该使用单数形式的`todo.task`模型，而不是`todo.tasks`。出于历史原因，可以找到一些不遵循这一原则的核心模型，比如`res.users`，但它不是规则。

模型名称必须是全局惟一的。因此，第一个单词应该与模块涉及的主要应用程序相对应。在我们的例子中，它是`todo`。核心模块的其他例子是`project`, `crm`,或`sale`。

另一方面，Python类是在声明它们的Python文件的本地。用于它们的标识符只对该文件中的代码有意义。因此，类标识符不需要被它们所关联的主要应用程序预先确定。例如，为`todo.task.stage`模型命名我们的类阶段是没有问题的。在其他模块上，没有可能与可能的类具有相同的名称。

可以使用两种不同的类标识符约定: `snake_case`或`CamelCase`。从历史上看，Odoo代码使用了snake案例，而且仍然可以找到使用这个约定的类。但趋势是使用骆驼案例，因为它是由PEP8编码规范定义的Python标准。您可能已经注意到，我们使用后一种形式。

瞬态和抽象模型

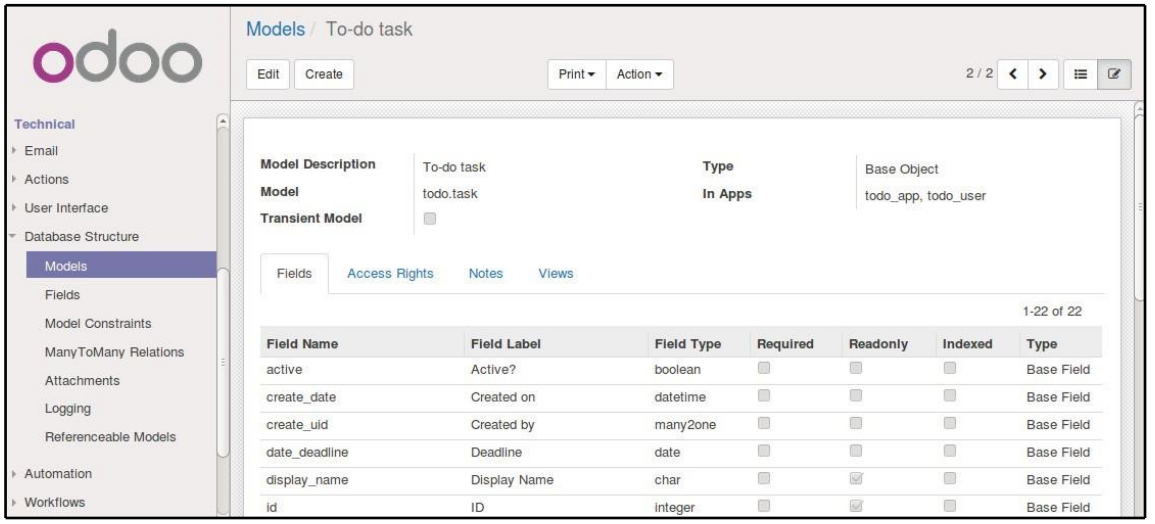
在前面的代码中，在绝大多数的Odoo模型中，类是基于`models.Model`类的。这些类型的模型具有永久的数据库持久性:为它们创建数据库表，并存储它们的记录，直到被显式删除。

但是Odoo还提供了另外两种模型类型: 瞬态和抽象模型。

> **Transient models**基于`models.TransientModel`类，并用于向导式的用户交互。他们的数据仍然存储在数据库中，但预计将是临时的。真空作业定期从这些表中清除旧数据。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。
>
> **Abstract models**基于`models.AbstractModel`类，没有任何数据存储。它们充当可重用特性集，与其他模型混合使用，使用Odoo继承功能。例如，`mail.thread`是一个抽象模型，由`Discuss` addon提供，用于向其他模型添加消息和追随者特性。

检查现有的模型

通过Python类创建的模型和字段可以通过用户界面获得它们的元数据。在**Settings**顶部菜单中，导航到**Technical** | **Database Structure** | **Models**菜单项。在这里，您将找到数据库中可用的所有模型的列表。点击列表中的模型将会打开一个包含其细节的表单:



这是一个很好的工具来检查模型的结构，因为在一个地方，您可以看到来自不同模块的所有自定义的结果。在这种情况下，正如您在**In Apps**字段的右上角看到的，这个模型的`todo.task`定义来自`todo_app`和`todo_user`模块。

在较低的区域，我们有一些信息标签可用:对模型的**Fields**的快速引用，对安全组授予的**Access Rights**，以及对该模型可用的**Views**。

我们可以从**Developer**菜单中找到模型的**External Identifier**, **View Metadata**选项。模型外部标识符(或XML id)由ORM自动生成，但很容易预测:对于`todo.task`模型，外部标识符是`model_todo_task`。

Models格式是可编辑的!可以在这里创建和修改模型、字段和视图。您可以使用它来构建原型，然后在模块中持久化它们。

创建字段

创建新模型后，下一步是向它添加字段。Odoo支持预期的所有基本数据类型，例如文本字符串、整数、浮点数、布尔值、日期、数据时间和图像/二进制数据。

一些字段名称是特殊的，因为它们是由ORM保留的特殊目的，或者因为默认使用一些默认字段名而内置了一些特性。

让我们来探索一下Odoo中可用的几种类型的字段。

基本的字段类型

我们现在有一个`Stage`模型，我们将扩展它来添加一些额外的字段。我们应该编辑`todo_ui/models/todo_model.py`文件并添加额外的字段定义，使其看起来像这样:

```
class Stage(models.Model):
    _name = 'todo.task.stage'
    _description = 'To-do Stage'
    _order = 'sequence,name'
    # String fields:
    name = fields.Char('Name', 40) desc
    = fields.Text('Description') state
    = fields.Selection(
        [('draft','New'), ('open','Started'),
        ('done','Closed')],'State')
    docs = fields.Html('Documentation')
    # Numeric fields:
    sequence = fields.Integer('Sequence') perc_complete
    = fields.Float('% Complete', (3, 2)) # Date fields:
    date_effective = fields.Date('Effective Date')
    date_changed = fields.Datetime('Last Changed') #
    Other fields:
    fold = fields.Boolean('Folded?')
    image = fields.Binary('Image')
```

在这里，我们有一个在Odoo中可用的非关系字段类型的示例，它们的位置参数是每个人所期望的。

在大多数情况下，第一个参数是字段标题，对应于`string`字段参数;这被用作用户界面标签的默认文本。它是可选的，如果没有提供，标题将自动从字段名中生成。

对于日期字段名，有一种约定，使用日期作为前缀。例如，我们应该使用`date_effective`字段而不是`effective_date`。类似的约定也适用于其他领域，如`amount_`，`price_`,或`qty_`。

这些是每个字段类型所期望的标准位置参数:

`Char` 期待第二个、可选、参数大小，以获得最大文本大小。建议不要使用它，除非有需要它的业务需求，比如具有固定长度的社会安全号码。

`Text` 与`Char`不同，它可以持有多行文本内容，但期望相同的参数。

`Selection` 是一个下拉选择列表。第一个参数是可选择选项的列表，第二个参数是字符串标题。选择项是(`'value', 'Title'`)元组的列表，用于存储在数据库中的值和相应的用户界面描述。当通过继承扩展时，`selection_add`参数可以将新项目追加到现有的选择列表中。

`Html` 存储为文本字段，但在用户界面上有特定的处理，用于HTML内容表示。出于安全原因，默认情况下它们是经过清理的，但是这种行为可以被重写。

`Integer` 只需要字段标题的字符串参数。

`Float` 有第二个可选参数，一个`(x,y)`元组和字段的精度: `x`是数字的总数;其中`y`是小数。

`Date` 和 `Datetime` 字段只希望字符串文本作为位置参数。由于历史原因，ORM以字符串格式处理它们的值。辅助函数应该用于将它们转换为实际的日期对象。同时，datetime值在UTC时间内存储在数据库中，但在本地时间显示，使用用户的时区首选项。在第6章中，我们将更详细地讨论这个问题——设计用户界面。

`Boolean` 如您所料，持有`True`或`False`值，并且只对字符串文本有一个位置参数。

`Binary` 存储文件如二进制数据，并且只期望字符串参数。可以使用`base64`编码的字符串来处理它们。

除了这些之外，我们还拥有关系字段，这将在本章后面介绍。但是现在，还有更多关于这些字段类型及其属性的知识。

常见的字段属性

字段具有可以在定义它们时设置的属性。根据字段类型，一些属性可以通过位置传递，没有参数关键字，如前一节所示。

例如，example, `name=fields.Char('Name', 40)`可以使用位置参数。使用关键字参数，同样可以写成`name=fields.Char(size=40, string='Name')`。关键字参数的更多信息可以在Python官方文档中找到: `https://docs.python.org/2/tutorial/controlflow.html#keyword-arguments`.
。

所有可用的属性都可以作为关键字参数传递。这些是通常可用的属性和相应的参数关键字:

`string` 是字段默认标签，用于用户界面。除了选择和关系字段之外，它是第一个位置参数，所以大部分时间它不被用作关键字参数。

`Default` 为字段设置默认值。它可以是一个静态值，例如字符串，或可调用引用，一个命名函数或一个匿名函数(lambda表达式)。

`size` 只适用于`Char`字段，可以设置允许的最大大小。当前的最佳实践是，除非真的需要，否则不要使用它。

`translate` 只适用于`Char`、`Text`和`Html`字段，并使字段内容可以翻译，对不同的语言持有不同的值。

`help` 为显示给用户的工具提示提供文本。

`readonly=True` 默认情况下，在用户界面上不编辑字段。这不是在API级别执行的;它只是一个用户界面设置。

`required=True` 在用户界面中默认设置字段。这是通过在列上添加`NOT NULL`约束来在数据库级别执行的。

`index=True` 将在字段上创建数据库索引。

`copy=False` 当使用重复记录功能，`copy()`ORM方法时，字段被忽略。默认情况下，非关系型字段是`copyable`。

`groups` 允许将字段的访问和可见性限制为一些组。它期望一个逗号分隔的安全组XML id列表，比如`groups='base.group_user,base.group_system'`

`states` 根据`state`字段的值，期望一个字典映射的UI属性值。例如:可以使用的`states={'done':[('readonly',True)]}`属性是`readonly`、`required`和`invisible`。

请注意，`states`字段属性与视图中的`attrs`属性相当。请注意，视图支持`states`属性，但它有不同的用法:它接受一个逗号分隔的状态列表，以控制元素何时应该可见。

为了完整性，有时在Odoo主要版本之间升级时使用另外两个属性:

`deprecated=True` 在使用字段时记录警告。

`oldname='field'` 当一个字段重命名为新版本时使用，使旧字段中的数据自动复制到新字段中。

特殊字段名称

ORM保留了一些字段名。

`id`字段是一个自动编号，唯一标识每个记录，并用作数据库主键。它会自动添加到每个模型中。

在新模型上自动创建以下字段，除非设置`_log_access=False`模型属性:

`create_uid` 是为创建记录的用户吗

`create_date` 当记录创建为`write_uid`的日期和时间时，是否为最后一个用户修改记录

`write_date` 记录修改后的最后一次日期和时间

此信息可从web客户端获得，导航到**Developer Mode**菜单并选择**View Metadata**选项。

默认情况下，一些API内置的特性期望特定的字段名。我们应该避免使用这些字段名，而不是用于预期目的。有些甚至是保留的，不能用于其他目的:

`name`被默认用作记录的显示名称。通常它是一个`Char`，但也可以是一个`Text`或`Many2one`字段类型。我们仍然可以使用`_rec_name`模型属性设置一个用于显示名称的字段。

`Active`级的`Boolean`，允许有未激活的记录。与`active==False`的记录将自动被排除在查询之外。要访问它们，必须将`('active','=',False)`条件添加到搜索域，或者`'active_test': False`应该添加到当前上下文。

`Sequence`类型的`Integer`,如果出现在列表视图中,允许手动定义记录的顺序。为了正常工作，您不应该忘记使用它与模型的`_order`属性。

类型为`Selection`的`State`，表示记录生命周期的基本状态，并可由state的字段属性使用，以动态修改视图:某些表单字段可以在特定的记录状态中生成`readonly`, `required`,或`invisible`。

`parent_id`, `parent_left`,和`parent_right`是`Integer`类型的，对父/子层次关系有特殊的意义。我们将在下一节详细讨论它们。

到目前为止，我们讨论了非关系字段。但是应用程序数据结构的一个重要部分是描述实

体之间的关系。现在我们来看看这个。

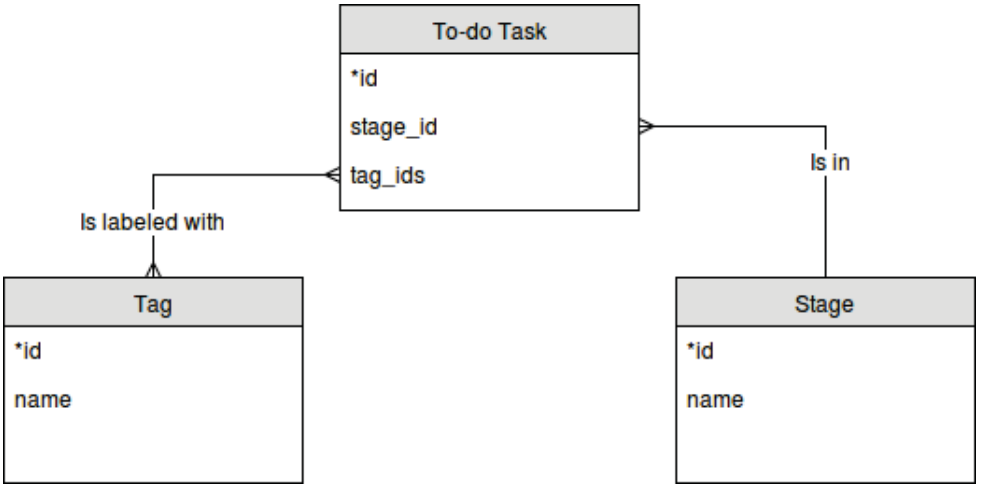
之间的关系模型

再看一下我们的模块设计，我们有这样的关系:

> 每个任务都有一个阶段。这是一个many-to-one的关系，也称为外键。逆是one-to-many关系，意思是每个阶段都可以有很多任务。

> 每个任务可以有多个标记。这是一个many-to-many关系。当然，反向关系也是many-to-many的，因为每个标记都可以有很多任务。

下面的实体关系图可以帮助可视化我们将要在模型上创建的关系。以三角形结尾的线代表了关系的许多方面:



让我们将相应的关系字段添加到`todo_model.py`文件中的待办事项:

```
class TodoTask(models.Model):
    _inherit = 'todo.task'
    stage_id = fields.Many2one('todo.task.stage', 'Stage')
    tag_ids = fields.Many2many('todo.task.tag', string='Tags')
```

前面的代码显示了这些字段的基本语法，设置了相关的模型和字段的标题`string`。关系字段名的约定分别是对字段名的`_id`或`_ids`，分别是to-one和to-many关系。

作为练习，您可以尝试将相应的反向关系添加到相关模型:

> `Many2one`关系的逆是一个分阶段的`One2many`字段，因为每个阶段都可以有很多任务。我们应该把这个字段添加到Stage类。
>
> `Many2many`关系的逆也是标签上的`Many2many`字段，因为每个标签也可以用于许多任务。

让我们仔细研究一下关系字段定义。

Many-to-one的关系

`Many2one`关系接受两个位置参数:相关模型(对应于`comodel`关键字参数)和标题`string`。它在数据库表中创建一个带外键的字段。

还有一些命名参数可以用于这类字段:

> `ondelete` 定义删除相关记录时会发生什么。它的默认值为`null`，这意味着在删除相关记录时将设置空值。其他可能的值是`restrict`，提高了防止删除的错误，并且`cascade`也删除了这个记录。
>
> - `context` 是数据字典，对web客户端视图有意义，在处理关系时携带信息。例如，设置默认值。在第六章中可以更好地解释，视图——设计用户界面。
>
> `domain` 是一个域表达式，一个元组列表，用来筛选关系字段可用的记录。
>
> `auto_join=True` 允许ORM在使用此关系进行搜索时使用SQL连接。如果使用，访问安全规则将被绕过，用户可以访问安全规则不允许的相关记录，但SQL查询将会更高效，运行更快。

Many-to-many的关系

`Many2many`最小签名接受与相关模型的一个参数，并推荐使用字段标题提供`string`参数。

在数据库级别，它不向现有表添加任何列。相反，它会自动创建一个新的关系表，只有两个ID字段与相关表的外键。关系表名和字段名是自动生成的。关系表名是两个表名，其中添加了一个下划线，后面加上了`_rel`。

在某些情况下，我们可能需要重写这些自动默认值。

其中一个例子是相关模型有长名称，而自动生成的关系表的名称太长，超过了63个字符的PostgreSQL限制。在这些情况下，我们需要手动选择关系表的名称，以符合表名称的大小限制。

另一个例子是，我们需要在同一模型之间建立第二个many-to-many关系。在这些情况下，我们需要手动为关系表提供一个名称 ,以便它不会与已经用于第一个关系的表名发生冲突。

有两种方法可以手动覆盖这些值:要么使用位置参数，要么使用关键字参数。

使用位置参数来定义字段定义:

```
# Task <-> Tag relation (positional args):
tag_ids = fields.Many2many(
    'todo.task.tag',      # related model
    'todo_task_tag_rel',  # relation table name
    'task_id',            # field for "this" record
    'tag_id',             # field for "other" record
    string='Tags')
```

注意，附加参数是可选的。我们可以为关系表设置名称，并让字段名使用自动默认值。

我们可以使用关键字参数，有些人喜欢可读性:

```
# Task <-> Tag relation (keyword args):
tag_ids = fields.Many2many(
    comodel_name='todo.task.tag',  # related model
    relation='todo_task_tag_rel',# relation table name
    column1='task_id',      # field for "this" record
    column2='tag_id',       # field for "other" record
    string='Tags')
```

就像many-to-one字段一样，many-to-many字段也支持`domain`和`context`关键字属性。

在ORM设计中有一个限制，关于抽象模型，当您强制关系表和列的名称时，它们就不能被干净地继承了。在抽象模型中不应该这样做。

`Many2many`关系的倒数也是`Many2many`级。如果我们还在`Tags`模型中加入`Many2many`域，那么Odoo就会发现这种many-to-many关系与`Task`模型中的一个相反。

任务和标记之间的反向关系可以像这样实现:

```
class Tag(models.Model):
    _name = 'todo.task.tag'
    # Tag class relationship to Tasks:
    task_ids = fields.Many2many(
        'todo.task',    # related model
        string='Tasks')
```

One-to-many的逆关系

`Many2one`的逆可以添加到关系的另一端。这对实际的数据库结构没有影响，但是允许我们轻松地从**many**相关记录的**one**端浏览。一个典型的用例是文档头和它的行之间的关系。

在我们的示例中，阶段上的One2many反向关系允许我们轻松地列出该阶段的所有任务。将这种逆关系添加到阶段的代码是:

```
class Stage(models.Model):
    _name = 'todo.task.stage'
    # Stage class relationship with Tasks:
    tasks = fields.One2many(
        'todo.task',   # related model
        'stage_id', # field for "this" on related model
        'Tasks in this stage')
```

One2many接受三个位置参数:相关模型、该模型中的字段名称，以及标题字符串。前两个位置参数对应于comodel_name和inverse_name关键字参数。

附加的关键字参数与Many2one: context、domain、ondelete (在此关系的**many**方面)和auto_join相同。

分级的关系

父-子树关系用与相同模型的Many2one关系表示，因此每个记录都引用其父节点。而反向One2many可以让父母很容易地跟踪孩子。

Odoo为这些层次结构的数据结构提供了改进的支持，用于更快地浏览树的兄弟姐妹，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，所以最好在你期望阅读的时候比写作的时候更频繁，比如一棵分类树的例子。

重新访问`todo_model.py`文件中定义的`Tags`模型，我们现在应该像这样编辑它:

```
class Tags(models.Model):
    _name = 'todo.task.tag'
    _description = 'To-do Tag'
    _parent_store = True
    # _parent_name = 'parent_id'
    name = fields.Char('Name')
    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)
```

在这里，我们有一个基本模型，有一个`parent_id`字段来引用父记录，另一个`_parent_store`属性添加层次搜索支持。这样做时，还必须添加`parent_left`和`parent_right`字段。

引用父类的字段预计将被命名为`parent_id`，但只要我们在`_parent_name`属性中声明，任何其他字段名称都可以使用。

此外，在记录的直接子项中添加一个字段通常很方便:

```
child_ids = fields.One2many(
    'todo.task.tag', 'parent_id', 'Child Tags')
```

使用动态关系的参考字段

常规关系字段引用一个固定的模型。参考字段类型没有这种限制并支持动态关系，因此同一个字段可以引用多个模型。

例如，我们可以使用它在To-do Tasks添加`Refers to`字段，它可以引用一个`User`或`Partner`:

```
# class TodoTask(models.Model):
    refers_to = fields.Reference(
    [('res.user', 'User'), ('res.partner', 'Partner')],
    'Refers to')
```

如您所见，字段定义类似于选择字段，但这里的选择列表保留了可以使用的模型。在用户界面上，用户将首先从av可用列表中选择一个模型，然后从该模型中选择一个记录。

可以采用另一种级别的灵活性: **Referenceable Models**配置表用于配置在**Reference**字段中使用的模型。它可以在**Settings | Technical | Database Structure**菜单中使用。在创建这样一个字段时，我们可以在`referenceable_models()`模块中使用`odoo.addons.res.res_request`函数的帮助，将其设置为在那里注册的任何模型。

使用**Referenceable Models**配置，`Refers to`字段的改进版本如下所示:

```
from odoo.addons.base.res.res_request import referenceable_models
# class TodoTask(models.Model):
    refers_to =
        fields.Reference( referenceable_mo
        dels, 'Refers to')
```

注意，在Odoo 9.0中，该函数使用了略微不同的拼写，并且仍然使用旧的API。在第9.0版本中，在使用之前显示的代码之前，我们必须在Python文件的顶部添加一些代码来包装它，以便它使用新的API:

```
from openerp.addons.base.res import res_request
    def referenceable_models(self):
        return res_request.referencable_models(
            self, self.env.cr, self.env.uid, context=self.env.context)
```

计算字段

字段可以具有由函数计算的值，而不是简单地读取数据库存储的值。一个计算字段就像一个常规字段一样声明，但是它具有定义用于计算它的函数的额外的`compute`参数。

在大多数情况下，计算字段涉及到编写一些业务逻辑，因此在第7章，ORM应用程序逻辑支持业务流程中，我们将进一步开发这个主题。我们仍然会在这里解释它们，但是将保持业务逻辑的尽可能简单。

让我们来看一个例子: Stage有一个`fold`字段。我们会在To-do Tasks的基础上加上**Folded**?标记对应的阶段。

我们应该在`todo_model.py`文件中编辑`TodoTask`模型来添加以下内容:

```
# class TodoTask(models.Model):
    stage_fold = fields.Boolean(
        '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
```

前面的代码添加了一个新的`stage_fold`字段和用于计算它的`_compute_stage_fold`方法。函数名作为字符串传递，但是它也可以作为可调用引用传递给它(没有引号的函数标识符)。在这种情况下，我们应该确保在字段之前，在Python文件中定义函数。

当计算依赖于其他字段时，需要`@api.depends` decorator，因为它通常这样做。它让服务器知道何时重新计算存储或缓存的值。一个或多个字段名被接受为参数，并使用点符号来跟踪字段关系。

计算函数将赋值给字段或字段以进行计算。如果没有，就会出错。由于`self`是一个记录对象，我们这里的计算只是为了获得**Folded?**使用`stage_id.fold`字段。结果是通过将该值(写入)赋给计算字段，即`stage_fold`。

对于这个模块，我们目前还没有工作，但是您可以在任务表单上快速编辑，以确认计算字段是否按预期工作:使用**Developer Mode**选择**Edit View**选项，并直接在表单XML中添加字段。别担心:它会被下一次升级的干净模块视图所取代。

在计算字段中搜索和写入

我们刚刚创建的计算字段可以读取，但不能搜索或写入。为了启用这些操作，我们首先需要为它们实现专门的功能。除了`compute`函数，我们还可以设置一个`search`函数，实现搜索逻辑，以及`inverse`函数，实现写逻辑。

使用这些，我们计算的字段声明变成这样:

```
# class TodoTask(models.Model):
    stage_fold = fields.Boolean(
        string='Stage Folded?',
        compute='_compute_stage_fold',
        # store=False,  # the default
        search='_search_stage_fold',
        inverse='_write_stage_fold')
```

支持功能是:

```
def _search_stage_fold(self, operator, value):
    return [('stage_id.fold', operator, value)]

def _write_stage_fold(self):
    self.stage_id.fold = self.stage_fold
```

当在搜索域表达式中找到该字段的(`field, operator, value`)条件时，调用`search`函数。它接收了`operator`和`value`的搜索，并期望将原始的搜索元素转换成另一个域搜索表达式。

`inverse`函数执行计算的反向逻辑，找到在计算的源字段上写的值。在我们的例子中，这意味着在`stage_id.fold`字段上写回。

存储计算字段

计算字段的值也可以存储在数据库中，根据它们的定义设置`store = True`。当它们的依赖项发生变化时 ,它们将被重新计算。由于现在存储了这些值 ,因此可以像普通字段一样搜索它们，而不需要搜索函数。

相关领域

我们在前一节中实现的计算字段只是将一个相关记录的值复制到一个模型的自己的字段中。然而，这是一种常见的用法，可以由Odoo自动处理。

使用相关字段也可以实现同样的效果。它们直接在一个模型上提供，这些字段属于一个相关的模型，可以使用点符号链访问。这使得它们在不能使用点符号的情况下可用，比如UI表单视图。

要创建一个相关的字段，我们声明一个需要类型的字段，就像使用常规计算的字段一样，但是我们不需要计算，而是使用点符号字段链的相关属性来达到期望的字段。

To-do Tasks是在可定制的阶段组织的，这些阶段将转换成基本状态。我们将使状态值直接在任务模型上可用，以便在下一章中可以用于一些客户端逻辑。

与`stage_fold`类似，我们将在任务模型中添加一个计算字段，但是这次使用了更简单的相关字段:

```
# class TodoTask(models.Model):
    stage_state = fields.Selection(
        related='stage_id.state',
        string='Stage State')
```

在幕后，相关字段只是计算字段，方便地实现`search`和`inverse`方法。这意味着我们可以在不需要编写任何附加代码的情况下搜索和写入它们。

模型约束

为了加强数据完整性，模型还支持两种类型的约束:SQL和Python

SQL约束被添加到数据库表定义中，并由PostgreSQL直接执行。它们是使用`_sql_constraints`级属性定义的。它是一个元组列表:约束标识符名;约束的SQL;以及要使用的错误消息。

一个常见的用例是为模型添加惟一的约束。假设我们不想允许两个具有相同标题的活动任务:

```
# class TodoTask(models.Model):
    _sql_constraints =
        [ ('todo_task_name_uniq
        ',
         'UNIQUE (name, active)',
         'Task title must be unique!')]
```

Python约束可以使用任意代码来检查条件。检查功能应该用`@api.constraints`来装饰，表示检查中涉及的字段的列表。当其中任何一个被修改时，验证被触发，如果条件失败，将引发一个异常。

例如，为了验证一个任务名至少有5个字符长，我们可以添加以下约束:

```
from odoo.exceptions import ValidationError #
class TodoTask(models.Model):
    @api.constrains('name')
    def _check_name_size(self):
        for todo in self:
            if len(todo.name) < 5:
                raise ValidationError('Must have 5
                chars!')
```

摘要

我们对模型和字段进行了详细的解释，使用它们来扩展To- Do应用程序，并在任务上进行标记和阶段。您学习了如何定义模型之间的关系，包括分层的父/子关系。最后，我们看到了使用Python代码计算字段和约束的简单示例。

在下一章中，我们将研究这些后端模型特性的用户界面，使它们在与应用程序交互的视图中可用。