

DebPacker 完全新手教程：从零开始打包你的第一个 Debian 软件包

欢迎来到 DebPacker 的世界！如果你是一名 Linux 新手，或是想将自己开发的程序打包成 `.deb` 安装包以便在其他 Debian/Ubuntu/UOS 系统上轻松安装，那么这份教程就是为你量身定制的。

我们将一步步带你安装 DebPacker，配置项目，最终生成一个可用的 `.deb` 软件包。整个过程不需要你写一行控制文件，完全在图形界面中完成。

第一部分：DebPacker 是什么？

DebPacker (得包) 是一个基于 PySide6 的图形化工具，它封装了 `fpm` 或 `dpkg-deb` 的复杂命令，让你能够：

- 通过拖拽添加文件，并指定它们在系统中的安装位置。
- 管理软件依赖（比如你的程序需要 Python 或某个库）。
- 添加安装/卸载前后执行的脚本（如创建用户、注册 systemd 服务）。
- 一键生成标准的 `.deb` 安装包。

有了它，你再也不用手动编写复杂的 `debian/` 目录，也不用死记硬背 `fpm` 的参数。

第二部分：安装 DebPacker

前提条件

- 你的电脑运行的是 **Debian、Ubuntu、Deepin、UOS** 等 Debian 系 Linux 系统。
- 你拥有 `sudo` 权限（能安装软件）。

1. 下载或获取 DebPacker 源码

将 DebPacker 的代码文件夹（比如 DebPacker）复制到你的桌面或任意目录。如果你还没有，可以从开发者那里获取。

假设你的目录路径为 ~/Desktop/DebPacker。

2. 安装系统依赖

打开终端（快捷键 Ctrl+Alt+T），依次执行：

安装 Python 虚拟环境支持（如果系统没有）

```
sudo apt update
```

```
sudo apt install python3-venv python3-pip
```

安装 fpm（可选，但推荐；DebPacker 也支持 dpkg-deb 后备）

```
sudo apt install ruby ruby-dev build-essential
```

```
sudo gem install fpm
```

如果 `gem install fpm` 因 Ruby 版本过旧失败，你仍然可以使用 DebPacker 的 `dpkg-deb` 模式（我们稍后会说明）。

3. 创建 Python 虚拟环境并安装 PySide6（图形界面依赖）

在终端中进入 DebPacker 目录：

```
cd ~/Desktop/DebPacker
```

```
python3 -m venv venv
```

```
source venv/bin/activate
```

```
pip install PySide6
```

如果创建虚拟环境失败（提示 `ensurepip` 不可用），请先安装 `python3-venv` 包（上面已包括）。

4. 启动 DebPacker

```
python run.py
```

如果一切正常，你会看到 DebPacker 的主窗口（如下图描述）：

- 左侧是基本信息填写区（包名、版本等）
- 中央是文件映射表格
- 右侧是依赖和脚本管理
- 底部是日志输出区域

第三部分：准备你的软件文件

在开始打包前，你需要将自己的程序文件整理到一个目录中，并想好它们在目标系统上的安装路径。

示例场景

假设你有一个名为 `myapp` 的 Python 脚本，内容如下（保存为 `myapp.py`）：

```
#!/usr/bin/env python3  
print("Hello, world!")
```

你希望：

- 安装到 `/usr/local/bin/myapp` 以使用户直接运行。
- 在安装时自动赋予可执行权限。

准备文件结构

我们建议按系统路径组织文件，方便映射：

```
~/myapp-package/  
├── usr/  
│   ├── local/  
│   │   └── bin/  
│   │       └── myapp.py    (你的程序文件)
```

这样，在 DebPacker 中只要将整个 myapp-package/ 目录映射到 /（根目录），文件就能自动放到正确位置。但我们也可以更灵活。

第四部分：使用 DebPacker 创建一个打包项目

1. 新建项目

点击工具栏的“新建”按钮（或菜单 文件 → 新建项目），所有输入框被清空。

2. 填写软件包的基本信息（左侧面板）

3. 添加文件映射（中央表格）

文件映射决定了：哪些本地文件→安装到系统的哪个位置。

- **手动添加行**：右键表格 → “添加行”，然后双击单元格输入源路径和目标路径。
- **拖拽添加**：直接从文件管理器拖拽文件或文件夹到表格中。
 - 如果是单个文件，会弹出对话框让你输入目标路径（例如 `/usr/local/bin/myapp.py`）。
 - 如果是文件夹，会递归添加所有文件，并自动生成目标路径（以包名作为子目录）。

对于我们的示例，你可以将 `myapp.py` 拖入表格，目标路径设为 `/usr/local/bin/myapp.py`。

朗小技巧：如果目标路径都是 `/usr/local/bin/` 开头，你可以先添加一行，然后右键 → “批量设置目标前缀”，输入 `/usr/local/bin/`，表格中该行的目标路径会自动变为前缀 + 源文件名。

4. 设置文件权限（可选）

DebPacker 本身不直接设置权限，但我们可以通过安装后脚本（`postinst`）来 `chmod` `+x` 可执行文件。下面会提到。

5. 管理依赖（右侧“依赖”标签）

如果你的程序需要其他软件包才能运行（例如 `python3`、`libc6`），在这里添加。

- 点击 “添加依赖”，弹出对话框：
 - 包名： `python3`
 - 运算符：留空（表示任意版本）
 - 版本：留空
- 点击确定。

依赖会在安装包时由 `dpkg` 检查，如果系统缺少，会提示用户安装。

6. 编写生命周期脚本（右侧“脚本”标签）

脚本在安装/卸载软件包的不同阶段执行，常用于自动配置。

- **postinst（安装后）**：例如创建用户、注册 `systemd` 服务、设置权限。
- **preinst（安装前）**：检查条件。
- **prerm（卸载前）**：停止服务。

- **postrm (卸载后)** : 清理残留。

对于我们的示例, 我们需要让 `myapp.py` 变为可执行, 并创建一个不带 `.py` 后缀的启动命令。我们可以这样写 `postinst` 脚本:

```
#!/bin/sh

set-e

chmod +x /usr/local/bin/myapp.py

# 可选: 创建一个符号链接, 方便执行
ln -sf /usr/local/bin/myapp.py /usr/local/bin/myapp

exit 0
```

在脚本编辑器中选中 `postinst`, 粘贴上述内容。

你也可以点击“插入模板”快速生成常用片段 (如 `systemd` 服务)。

7. 保存项目

点击“**保存**”按钮, 将当前配置保存为 `.fpmproj` 文件, 方便以后再次打开修改。

第五部分：执行打包

- 点击工具栏的“**▶ 打包**”按钮。
- 日志区域会显示打包过程 (复制文件、调用 `fpm` 或 `dpkg-deb` 等)。
- 如果成功, 会弹出提示框, 并在输出目录生成 `.deb` 文件。

第六部分：安装测试

在终端中, 导航到输出目录, 执行:

```
sudo dpkg -i myapp_1.0.0_all.deb
```

如果提示缺少依赖，执行 `sudo apt install -f` 自动修复。

安装后，测试运行：

```
myapp
```

```
# 或
```

```
myapp.py
```

如果输出了 `Hello, world!`，说明打包成功！

第七部分：常见问题与解答

1. 打包失败，提示 `fpm: command not found`

- 说明系统未安装 `fpm`，你可以选择：
 - 安装 `fpm` （ `sudo gem install fpm` ）
 - 或在 DebPacker 设置中切换为 `dpkg-deb` 模式（如果支持）。目前 DebPacker 默认优先使用 `fpm`，若不可用则提示。

2. 安装后点击开始菜单图标无反应

- 检查 `.desktop` 文件中的 `Exec` 路径是否正确。
- 检查 `/usr/bin/` 下的启动脚本是否有执行权限。
- 检查是否缺少 `PySide6` 等依赖。

3. 我的程序需要图形界面，但无法启动

- 确保你的程序依赖的图形库已安装（如 `PyQt/PySide`）。
- 在 `.desktop` 文件中设置 `Terminal=false` 。

4. 打包后文件权限不对

- 在 `postinst` 脚本中手动设置权限（如 `chmod 755`）。

5. 打包时提示“目标路径已存在”

- 可能之前生成过同名的 `.deb` 文件，要么删除旧文件，要么修改版本号。

第八部分：进阶技巧

1. 打包整个目录树

如果你有完整的目录结构（如 `usr/`、`etc/`），只需将根目录映射到 `/`，即可一次性打包所有文件。

2. 添加配置文件标记

在文件映射表格中，勾选“配置文件”列，这样该文件在升级时不会被覆盖（用户配置得以保留）。

3. 批量生成不同架构的包

在打包前修改“架构”字段，可分别生成 `amd64`、`arm64` 或 `all` 包。

4. 使用虚拟环境打包复杂 Python 应用

你可以在 `postinst` 中创建虚拟环境并安装所需的 `pip` 包，但考虑到网络和权限，推荐在软件包依赖中声明所需 Python 包，并由用户自行安装。

结语

恭喜！你已经掌握了使用 DebPacker 打包软件的全流程。无论是命令行小工具还是图形应用，DebPacker 都能帮你轻松转化为.deb 安装包，方便分发给其他用户。

如果你在实践过程中遇到任何问题，欢迎查阅软件内置的帮助文档或向开发者反馈。祝打包愉快！