blog.csdn.net

Openstack : 10、openstack中pbr的作用和解释 - CSDN博客

56-71 分钟

A library for managing setuptools packaging needs in a consistent manner.

pbr reads and then filters the *setup.cfg* data through a setup hook to fill in default values and provide more sensible behaviors, and then feeds the results in as the arguments to a call to *setup.py* - so the heavy lifting of handling python packaging needs is still being done by *setuptools*.

Note that we don't support the *easy_install* aspects of setuptools: while we depend on setup_requires, for any install_requires we recommend that they be installed prior to running *setup.py install* - either by hand, or by using an install tool such as *pip*.

翻译:

pbr -python 合理编译工具

这是一个一致的管理python setuptools 的工具库

pbr模块读入setup.cfg文件的信息，并且给setuptools 中的setup hook 函数填写默认参数，提供更加有意义的行为，然后使用setup.py来条用，因此setuptools工具包依然是必须的。

注意，我们并不支持setuptools包中的easy_install工具集，当我们依赖于安装需求前提软件，我们推荐使用setup.py install方式或者pip方式安装。

What It Does

PBR can and does do a bunch of things for you:

- **Version**: Manage version number based on git revisions and tags
- **AUTHORS**: Generate AUTHORS file from git log
- **ChangeLog**: Generate ChangeLog from git log
- **Manifest**: Generate a sensible manifest from git files and some standard files
- **Sphinx Autodoc**: Generate autodoc stub files for your whole module
- **Requirements**: Store your dependencies in a pip requirements file
- **long_description**: Use your README file as a long_description
- **Smart find_packages**: Smartly find packages under your root package

翻译：

它能干什么

PBR包可以做以下事情

版本：可以基于git版本和标签信息管理版本号

作者:从git的日志信息产生作者信息

更改日志：从git日志中产生软件包日志

manifest:从git以及其他标准文档中产生一个manifest文件

Sphinx Autodoc:自动产生stub files

需求：生成requirements需求文件

详细描述：使用你的README文件作为包的描述

聪明找包：从你的包的根目录下聪明的找到包

Version

Versions can be managed two ways - postversioning and preversioning. Postversioning is the default, and preversioning is enabeld by setting `version` in the setup.cfg `metadata` section. In both cases version

strings are inferred from git.

If a given revision is tagged, that’s the version.

If it’s not, then we take the last tagged version number and increment it to get a minimum target version.

We then walk git history back to the last release. Within each commit we look for a Sem-Ver: pseudo header, and if found parse it looking for keywords. Unknown symbols are not an error (so that folk can’t wedge pbr or break their tree), but we will emit an info level warning message. Known symbols: `feature`, `api-break`, `deprecation`, `bugfix`. A missing Sem-Ver line is equivalent to `Sem-Ver: bugfix`. The `bugfix` symbol causes a patch level increment to the version. The `feature` and `deprecation` symbols cause a minor version increment. The `api-break` symbol causes a major version increment.

If postversioning is in use, we use the resulting version number as the target version.

If preversioning is in use - that is if there is a version set in *setup.cfg* metadata - then we check that that version is higher than the target version we inferred above. If it is not, we raise an error, otherwise we use the version from *setup.cfg* as the target.

We then generate dev version strings based on the commits since the last release and include the current git sha to disambiguate multiple dev versions with the same number of commits since the release.

翻译:

版本：版本可以使用两种方式管理-postversioning，preversioning管理。postversioning是默认方式，preversioning通过在setup.cfg文件的metadata项目下面设置版本信息控制。两种情况下，都从git中获取版本的字符串信息。

如果指定修改设置了标签，那么就是版本。如果没有，我们则使用最后一次被标签记录的版本号。在其上递增获取目标版本。

我们然后便利git的历史信息，回到最后的发布。在每次提交，我们寻找Sem-Ver:假的header，如果找到解析的关键字.不认识的符号非错误（因此人们不会破坏pbr和树形结构），但是我们会发出一个警告信息。已知的符号：feature,apt-break,deprecation,bugfix.一个缺失的Sem-Ver 行等于Sem-Ver:bugfix.bugfix符号导致该版本上增加一个patch。feature和deprecation符号导致最小的版本递增。api-break符号导致最大的版本递增。

如果使用postversioning，我们使用resulting 版本号作为目标版本。

如果使用preversioning，那就是说在setup.cfg的metadata栏目中有设置版本信息，那么我们将检查比以上我们提到的目标版本更加大的版本号。如果没有，我们将引发一个错误，否则我们使用setup.cfg中的定义的版本号作为目标版本号。

基于最后发布的提交，我们产生一个dev 版本的字符串。然后使用目前git的sha字符串来区分多个dev 版本中使用相同号码的提交。

Note

pbr expects git tags to be signed for use in calculating versions

The versions are expected to be compliant with Linux/Python Compatible Semantic Versioning 3.0.0.

The `version.SemanticVersion` class can be used to query versions of a package and present it in various forms
- `debian_version()`, `release_string()`, `rpm_string()`, `version_string()`, or `version_tuple()`.

注意:

pbr通过git的标签来计算版本。

版本与inux/Python Compatible Semantic Versioning 3.0.0.兼容。

`version.SemanticVersion类可以通过查询包的版本号，然后通过不同形式展现: debian_version(), release_string(), rpm_string(), version_string(), or version_tuple().`

AUTHORS and ChangeLog

Why keep an *AUTHORS* or a *ChangeLog* file when git already has all of the information you need? *AUTHORS* generation supports filtering/combining based on a standard .*mailmap* file.

作者以及变更日志

有什么必要重作一份，当git已经有了作者与变更日志的记录呢？能够依据.mailmap文件自动过滤组合成作者信息。

Manifest

Just like *AUTHORS* and *ChangeLog*, why keep a list of files you wish to include when you can find many of these in git. *MANIFEST.in* generation ensures almost all files stored in git, with the exception of *.gitignore*, *.gitreview* and *.pyc* files, are automatically included in your distribution. In addition, the generated *AUTHORS* and *ChangeLog* files are also included. In many cases, this removes the need for an explicit ‘MANIFEST.in’ file

就像作者和变更日志，有什么必要维护一份文件列表信息，当git已经在.mainfest 中维护了一份除了.gitignore,gitreview,.pyc等文件的其他的信息。在多个情况下，可以除掉MAINFEST.in文件。

Sphinx Autodoc

Sphinx can produce auto documentation indexes based on signatures and docstrings of your project but you have to give it index files to tell it to autodoc each module: that’s kind of repetitive and boring. PBR will scan your project, find all of your modules, and generate all of the stub files for you.

Sphinx documentation setups are altered to generate man pages by default. They also have several pieces of information that are known to setup.py injected into the sphinx config.

See the pbr section for details on configuring your project for autodoc.

Sphinx 自动文档

Sphinx 可以自动产生文档索引基于你的项目中的标签与文档字符串(代码中""" ..."""部分).但是你得告诉它每个模块的索引文件位置：这种 工作重复性且无聊。PBR会自动扫描你的项目，找到你所有模块，产生stub 文件。

Sphnix默认设置为更改则自动更新。他们有几处信息告诉setup.py，被注入到sphinx设置中。

查看pbr 段落设置详细信息来设置你的项目

Requirements

You may not have noticed, but there are differences in how pip *requirements.txt* files work and how distutils wants to be told about requirements. The pip way is nicer because it sure does make it easier to populate a virtualenv for testing or to just install everything you need. Duplicating the information, though, is super lame. To solve this issue, *pbr* will let you use *requirements.txt*-format files to describe the requirements for your project and will then parse these files, split them up appropriately, and inject them into the *install_requires*, *tests_require* and/or *dependency_links* arguments to *setup*. Voila!

You can also have a requirement file for each specific major version of Python. If you want to have a different package list for Python 3 then just drop a *requirements-py3.txt* and it will be used instead.

Finally, it is possible to specify groups of optional dependencies, or "extra" requirements, in your *setup.cfg* rather than *setup.py*.

要求

你可能没有注意，但是pip 的requirements.txt 工作方式与 distutils的告知requirements需求是有差异的。pip方式更好，因为它保证操作virtualenv，以及安装所有软件更加容易，容易产生重复信息，但，非常糟糕。为了解决这个问题，pbr会让你使用requirements.txt-格式的文件去描述你的项目的前提需求，然后解析这些文档，将它们恰当分割开来，然后将它们注入到 安装需求 (install requires)，测试需求(tests require),依赖路径的设置。真不错。

你可以为特别版本的python设置需求文档。如果你希望python 3有不同的安装包列表你可以写一个requirements-py3.txt，它会被使用。

最后，可以设置可选的依赖，或者"特别"需求。在你的setup.cfg文件中。

long_description

There is no need to maintain two long descriptions- and your README file is probably a good long_description. So we'll just inject the contents of your README.rst, README.txt or README file

into your empty long_description. Yay for you.

日志描述

没有必要维护两个详细描述-你的README文件就应该是很好的描述。所以我们将你的README.rst，或者README.txt 或者README文件中的内容提取出来，作为项目描述。

Usage

pbr is a setuptools plugin and so to use it you must use setuptools and call `setuptools.setup()`. While the normal setuptools facilities are available, pbr makes it possible to express them through static data files.

使用

pbr是setuptools的插件，所以你必须使用setuptools，然后调用setuptools.setup()函数。当普通的setuptools设施可用，pbr通过静态的数据文档使得需求被更加清晰表达。

setup.py

pbr only requires a minimal *setup.py* file compared to a standard setuptools project. This is because most configuration is located in static configuration files. This minimal *setup.py* file should look something like this:

pbr只需要最小化的setup.py 文件，跟普通的使用setuptools的项目相比。这是因为设置都在setup.cfg里面。setup.py文件如下。

Note

It is necessary to specify `pbr=True` to enabled *pbr* functionality.

Note

While one can pass any arguments supported by setuptools to *setup()*, any conflicting arguments supplied in *setup.cfg* will take precedence.

注意：

必须设置pbr=True,来使用pbr.

当使用setuptools的setup函数来进行设置时，如果与位于setup.cfg中的信息产生冲突，则setup.cfg则优先。

setup.cfg

The *setup.cfg* file is an ini-like file that can mostly replace the *setup.py* file. It is based on the distutils2 *setup.cfg* file. A simple sample can be found in *pbr*'s own *setup.cfg* (it uses its own machinery to install itself):

setup.cfg文档

这个文档像ini 文档。它基于项目distutils2的setup.cfg文件设置。下面是一个例子(这个设置中项目使用自己的机制去安装)

There are a number of sections in these documents. These are:

- metadata
- files
- entry_points
- pbr

setup.cfg文件中有几个段落:

- metadata
- files
- entry_points
- pbr

files

The `files` section defines the install location of files in the package using three fundamental keys: `packages`, `namespace_packages`, and `data_files`.

`packages` is a list of top-level packages that should be installed. The behavior of packages is similar to `setuptools.find_packages` in that it recurses the python package hierarchy below the given top level and installs all of it. If `packages` is not specified, it defaults to the value of the `name` field given in the `[metadata]`section.

`namespace_packages` is the same, but is a list of packages that provide namespace packages.

`data_files` lists files to be installed. The format is an indented block that contains key value pairs which specify target directory and source file to install there. More than one source file for a directory may be indicated with a further indented list. Source files are stripped of leading directories.
Additionally, *pbr* supports a simple file globbing syntax for installing entire directory structures, thus:

段落 files

files段落定义了包中的文件位置，有三个基本的设置键：packages,namespace_packages,以及data_files.

packages指定了必须安装的数个最高级别的包的列表。这个像setuptools中的函数find_packages.这里面它进入python的包体系中，在最高的级别路径下安装它。如果packages没有被指明，则默认为metadata段落中的name键值。

will result in */etc/neutron* containing *api-paste.ini* and *dhcp-agent.ini*, both of which pbr will expect to find in the *etc* directory in the root of the source tree. Additionally, *neutron.init* from that dir will be installed in */etc/init.d*. All of the files and directories located under *etc/pbr* in the source tree will be installed into */etc/pbr*.

Note that this behavior is relative to the effective root of the environment into which the packages are installed, so depending on available permissions this could be the actual system-wide */etc* directory or just a top-level *etc* subdirectory of a virtualenv.

上面的配置会 将本项目的**etc/api-paste.ini**拷贝**/etc/neutron** 文件夹中，而将项目中的**etc/pbr**中的所有文件拷贝到**/etc/pbr**中。项目中的**neutron.init**将会拷贝到**/etc/init.d**中。

注意:

这种行为是根据在那个环境下包被安装，以及它们相应的根目录。(也就是说拷贝也许不是到/etc/neutron中，或者某个子目录下面的etc/neutron中)。所以基于权限，文件会被拷贝到根目录下的etc或者某个子目录下的etc

entry_points

The *entry_points* section defines entry points for generated console scripts and python libraries.

The general syntax of specifying entry points is a top level name indicating the entry point group name, followed by one or more key value pairs naming the entry point to be installed. For instance:

段落entry_points

这个段落定义了命令行命令以及python的库lib的进入点。

里面的内容分成子段落，子段落的头行设置了一组进入点的最高级目录的名称，里面定义了键值对，描述了会被安装的进入点。

Will cause a console script called *pbr* to be installed that executes the *main* function found in *pbr.cmd*. Additionally, two entry points will be installed for *pbr.config.drivers*, one called *plain* which maps to the *Plain* class in *pbr.cfg.driver* and one called *fancy* which maps to the *Fancy* class in *pbr.cfg.driver*.

这段设置会产生一个pbr的脚本，执行pbr.cmd中的main函数，而pbr.config.dirvers则会安装两个进入点，为plain以及fancy。对应Plain,和Fancy函数。

pbr

The pbr section controls pbr specific options and behaviours.

The `autodoc_tree_index_modules` is a boolean option controlling whether pbr should generate an index of modules using `sphinx-apidoc`. By default, *setup.py* is excluded. The list of excluded modules

can be specified with the `autodoc_tree_excludes` option. See the sphinx-apidoc man page for more information.

The `autodoc_index_modules` is a boolean option controlling whether *pbr* should itself generates documentation for Python modules of the project. By default, all found Python modules are included; some of them can be excluded by listing them in `autodoc_exclude_modules`. This list of modules can contains *fnmatch* style pattern (e.g. *myapp.tests.**) to exclude some modules.

The `warnerrors` boolean option is used to tell Sphinx builders to treat warnings as errors which will cause sphinx-build to fail if it encounters warnings. This is generally useful to ensure your documentation stays clean once you have a good docs build.

pbr

pbr段落控制pbr相关的参数以及行为

autodoct_tree_index_modules 是一个布尔型参数，描述pbr是否应该为sphinx-apidoc中的模块自动产生索引。默认的，setup.py排除在外。可以在autodoc_tree-excludes的设置中设定一系列被排除在外的模块。参考sphinx-apidoc man page 获取更多帮助信息。

autodoc_index_modules是一个布尔型参数,控制pbr是否自己为项目的python模块产生文档。默认的，所以找到的python模块包含在内；他们有些被autodoc_exclude_modules排除掉了。这个排除文件模块列表可以包含fnmatch 风格的设置(例如myapp.tests.*)

warnerrors 是一个布尔型参数，用来告诉sphinx 将警告信息看待为出错信息，如果当它遇到警告就会失败。如果你想让你的文档干净良好这样做很必要。

Note

When using `autodoc_tree_excludes` or `autodoc_index_modules` you may also need to set `exclude_patterns` in your Sphinx configuration file (generally found at *doc/source/conf.py* in most OpenStack projects) otherwise Sphinx may complain about documents that are not in a toctree. This is especially true if the `warnerrors=True` option is set. See the Sphinx build configuration file documentation for more information on configuring Sphinx.

注意

当使用autodoc_tree_excludes 或者autodoc_index_modules。你也需要在sphinx的配置文件中设置exclude_patterns(一般在项目中的 /doc/source/conf.py找到)。否则sphinx将会告知文档不在toctree中。当warnerrors=True开启后这个也需要为真。参考Sphinx build configuration file 文档获取更多如何配置sphinx.

Requirement files should be given one of the below names. This order is also the order that the requirements are tried in (where *N* is the Python major version number used to install the package):

- requirements-pyN.txt
- tools/pip-requires-py3

- requirements.txt
- tools/pip-requires

  Only the first file found is used to install the list of packages it contains.

  Note

  The ‘requirements-pyN.txt’ file is deprecated - ‘requirements.txt’ should be universal. You can use Environment markers for this purpose.

  项目需求

  需求文档必须以以下名字命名。列举顺序也就是python进行尝试时候的顺序。(这里面N代表python安装包时候的主要版本数字)

- requirements-pyN.txt
- tools/pip-requires-py3
- requirements.txt
- tools/pip-requires

Environment markers

Environment markers are conditional dependencies which can be added to the requirements (or to a group of extra requirements) automatically, depending on the environment the installer is running in. They can be added to requirements in the requirements file, or to extras defined in *setup.cfg*, but the format is slightly different for each.

环境标签

环境标签是条件依赖，根据运行环境来设定需求文件。它们可以被加入进requirements文档或者在setup.cfg中定义,但是格式有些微不同。

For `requirements.txt`:

```
argparse; python_version=='2.6'
```

例如在requirements.txt中

```
argparse; python_version=='2.6'
```

This will result in the package depending on `argparse` only if it's being installed into Python 2.6

For extras specifed in *setup.cfg*, add an `extras` section. For instance, to create two groups of extra requirements with additional constraints on the environment, you can use:

这会导致包依赖于argparse包，仅仅当在python2.6的情况下

而在setup.cfg中，在extras段里面，例如，创建依赖环境的两个需求组，你可以使用以下配置：