RPM Python (and friends)

Paul Nasrat

Summary

- **Background of rpm python bindings**
- **Reasons for interacting with RPM**
- **RPM python itself**
- **Friends**

Background

- **RPM is the RPM Package Manager**
- **Python is a dynamically typed scripting language**
- **rpmlib bindings for C**
- **anaconda for installer in python - bindings created**
- **later used in up2date**

The rpm python bindings are hand crafted python extensions in C. They provide a subset of rpmlib, particularly lacking interaction with rpmbuild.

Uses

Installation	**anaconda uses rpm-python for installation**
Dependancy Solvers	**Both yum and up2date use rpm-python. apt-rpm now uses rpmlib (C++)**
Repository generation and checking	**yum-arch, metadata, treetool**
Build systems	**mach**

rpm - macros

Macros

Macros are important in RPM, they can tell RPM where the database is, amongst other things. You normally encounter macros in spec files `%define foo bar`

```
import rpm
rpm.addMacro("_dbPath","/usr/lib/rpmdb/i386-redhat-linux/redhat")
solvets = rpm.TransactionSet()
# We discuss transaction sets next
rpm.delMacro("_dbPath")
```

RPM macros are a stack. Adding one pushes the new value onto the stack, removing it pops it. Thus in the above example %_dbPath is reset to the default value.

Transaction Sets (rpm.ts)

The transaction set will open the RPM database as needed so in most case you will not need to explicitly open the database. The transaction set is the workhorse of RPM.

```
import rpm
ts=rpm.TransactionSet()
```

As well as using macros to select the installation path that RPM will use. RPM will effectivel chroot to that path and use the default database, usually located in /var/lib/rpm within the chroot. This comes in useful with installers, etc.

```
import rpm
# rpm.ts is an abbreviation for rpm.TransactionSet
ts=rpm.ts("/mnt/sysimage")
```

rpm.ts - Flags and VSFlags

Flags

A transaction set can be modified or setup with various flags. There are two main types of flags `ts.Flags`relates to general transaction flags such as `rpm.RPM_TRANS_FLAG_TEST` and `rpm.RPM_TRANS_FLAG_NOSCRIPTS`. Transaction flags are set using `ts.setFlags(flag)`. setFlags returns the flags prior to changing should you wish to restore them later.

VSFlags

`ts.VSFlags` are flags relating to verification of signatures. Flags are bitmasks and you can combine by binary or, e.g. `myflags = rpm.RPMVSF_NORSA | rpm.RPMVSF_NODSA`. Verification flags are set using `ts.setVSFlags(vsflags)`, or by passing them as the second argument to `rpm.ts`.

rpm.hdr

Headers

This represents an RPM header object. The header contains metadata about a package. A header can be returned from the RPM database or from an RPM package on disk. Below we extract a header object from an RPM file.

```
import rpm, os
# rpm.ts is an alias for rpm.TransactionSet
ts = rpm.ts()
fdno = os.open("/tmp/foo-1.0-1.i386.rpm", os.O_RDONLY)
hdr = ts.hdrFromFdno(fdno)
os.close(fdno)
```

rpm.hdr - getting information

Once we have a header object what can we do with it?

Accessing keys

The simple explanation is that the header is presented as a dictionary. Keys can be accessed by tag number, constants exist for these, or by name

```
> name = hdr[rpm.RPMTAG_NAME]
'foo'
> hdr['name']
'foo'
```

Match Iterator (rpm.mi)

To get a RPM header from the database you must use dbMatch which returns a match iterator. Then you loop over the match iterator for each matching header. In it's simplest form

```
import rpm
ts=rpm.ts()
mi=ts.dbMatch("name","kernel")
for hdr in mi:
    print "%s-%s-%s" % (hdr['name'], hdr['version'], hdr['release'])
```

Calling `dbMatch()` with no arguments returns all matches within the RPM database, thus is equivalent to `rpm -qa`

Match iterators can be told to use globs or regular expressions rather than searching on a header key.

```
import rpm
ts=rpm.ts()
# Equivalent to rpm -qa
mi=ts.dbMatch()
# Apply regex
mi.pattern("name", rpm.RPMMIRE_GLOB, "kernel*")
for hdr in mi:
   print "%s-%s-%s" % (hdr['name'], hdr['version'], hdr['release'])
```

Callbacks

Callbacks

To get progress status and hook in at other stages in a transaction you need to use a callback

```
class simpleCallback:
   def __init__(self):
      self.fdnos = {}

   def callback(self, what, amount, total, mydata, wibble):
    """what -- callback type, amount --
      if what == rpm.RPMCALLBACK_TRANS_START:
         pass

      elif what == rpm.RPMCALLBACK_INST_OPEN_FILE:
         hdr, path = mydata
         print "Installing %s\r" % (hdr["name"])
         fd = os.open(path, os.O_RDONLY)
         nvr = '%s-%s-%s' % ( hdr['name'], hdr['version'], hdr['release'] )
         self.fdnos[nvr] = fdno
         return fd

 elif what == rpm.RPMCALLBACK_INST_CLOSE_FILE:
         hdr, path = mydata
         nvr = '%s-%s-%s' % ( hdr['name'], hdr['version'], hdr['release'] )
         os.close(self.fdnos[nvr])

      elif what == rpm.RPMCALLBACK_INST_PROGRESS:
         hdr, path = mydata
         print "%s:  %.5s%% done\r" % (hdr["name"], (float(amount) / total) * 100)
```

Installing an rpm

Installing an rpm

Installing or upgrading an rpm consists of adding a header and the package to the transaction set

```
rpmloc = "/path/to/foo-1.0.1.noarch.rpm"
fdno = os.open(rpmloc, os.O_RDONLY)
hdr = ts.hdrFromFdno(fdno)
os.close(fdno)
ts.addInstall(hdr,(hdr,rpmloc), "u")
cb = simpleCallback()
ts.run(cb.callback,”)
```

Now we can see our callback in action:

```
Installing foo
foo:  33.33% done
foo:  68.68% done
foo:  100.0% done
```

Erasing an RPM

Erasing an RPM

Having installed an RPM it is only natural want to know how to remove one. This is done by calling `ts.addErase()` by name.

```
import rpm
ts=rpm.ts()
ts.addErase("foo")
```

If you want to remove a particular package instance you can do this by getting the package instance by querying the database.

```
mi = ts.dbMatch(rpm.RPMTAG_NAME, "kernel")
mi.pattern(rpm.RPMTAG_ARCH, rpm.RPMMIRE_DEFAULT, "i386")
for idx in mi:
   instance = mi.instance()
   ts.addErase(instance)
```

Putting it all together I

As a system administrator I want to go through all files that rpm thinks are config files and find which ones have changed since installation. I'm only intrested if the file contents have changed not the timestamps or permissions

We start off importing all the modules we need and setting up our transaction set:

Example 1. Initial setup

```
import rpm, md5, os

ts = rpm.TransactionSet()
mi = ts.dbMatch
configs = []
```

Putting it all together II

- **Iterate over the headers**
- **For each header extract file list, md5s and flags**

```
for h in mi:
   name=h['filenames']
   fileflags=h['fileflags']
   md5sums=h['filemd5s']
   total=len(names)
```

- **Find all the config files and pass them to a function**

```
for i in xrange(total):
   if (fileflags[i] & rpm.RPMFILE_CONFIG):
      if isModified(names[i], md5sums[i]):
         configs.append(names[i])
```

Putting it all together III

- **Compare the md5sum against that on the system**
- **Return true if the file is modified**

```
def isModified(fileName, fileMD5):
   m = md5.new()
   f = open(fileName,"r")
   data = f.read()
   f.close()
   m.update(data)
   if fileMD5 != m.hexdigest():
      return 1
   else:
      return 0
```

perl and RPM

There are several modules relating to perl and RPM, available via CPAN [http://cpan.org]

perl-RPM2

The RPM2 module allows you to query RPM files and the RPM database

- **Open the RPM database**
- **Create an iterator**
- **Iterate and print nvre**

```
use RPM2;

my $db = RPM2->open_rpm_db();

my $i = $db->find_all_iter();
while (my $pkg = $i->next) {
   print $pkg->as_nvre, "\n";
}
```

perl and RPM contd.

RPM::Specfile

This module provides methods for creating RPM Spec files. Currently rpm-python lacks Spec file support.

- **Simple getters and setters**

```
$oldurl = $spec->url;
$spec->url('http://www.python.org');
```

- **Lists are handled by index, push or clear methods**
- **`write_specfile($file)` handles outputting the final Spec file**
- **cpanflute2 provides a good example of `RPM::Specfile` and is useful for those wishing to create RPMs from CPAN**

lua and RPM

lua

Lua [http://www.lua.org/] support has been added to rpm for both scriptlets and macros. Some reasons for adding lua support include:

- **Many scripts execute simple operations which an internal interpreter requires no forking at all**
- **Internal scripts reduce or eliminate external dependancies related to script slots**
- **Internal scripts operate even under very minimal environments, e.g.stripped chroot, LFS, installer**
- **Syntax errors in internal scripts are detected at build not run time**

How it works:

Just use `-p <lua>` in any script slot.

```
%pre -p lua
print("Hello, Lua RPM World!")
```

Inline macros are done using `%{lua: print ("Requirs: foo > 1.1") }`. Lua provides better support for multiline macros than the standard RPM macro parser.

Further Resources

For more information try the following first:

- **rpm-list [https://listman.redhat.com/mailman/listinfo/rpm-list/]**
- **rpm-python-list [https://lists.dulug.duke.edu/mailman/listinfo/rpm-python-list]**
- **#rpm on freenode**

Recommended Reading:

- **RPM API docs [http://www.rpm.org/rpmapi-4.1/]**
- **Maximum RPM [http://www.rpm.org/max-rpm/]**
- **yum source code [http://linux.duke.edu/projects/yum/]**
- **anaconda source code [http://fedora.redhat.com/projects/anaconda-installer/]**
- **up2date source code**

Acknowledgements:

This talk benefitted from discussion, assistance and code from a number of sources. In particular I'd like to thank the following specifically:

- **Jeff Johnson**
- **Jeremy Katz**
- **Adrian Likins**
- **Gustavo Niemeyer**
- **Seth Vidal**
- **Everyone on rpm-python and rpm-list and #rpm**