Paketerzeugung mit RPM

Tino Schwarze, TeeGee

September 2000

Inhaltsverzeichnis

1 RPM-Überblick

Die Aufgaben von RPM gliedern sich in

- Installation
- Deinstallation
- Update
- Konsistenzprüfung (Abhängigkeiten etc.)
- Abfrage (zu welchem Paket gehört eine Datei, was ist in einem Paket etc.)
- Erzeugen

von Softwarepaketen.

Diese Anleitung beschäftigt sich vorrangig mit dem letzen Punkt, dem Erzeugen von RPM-Paketen.

Im Kontext von RPM werden zwei Typen von RPM-Dateien unterschieden:

1. Binary RPMs

   Diese RPMs stellen ein fertiges Softwarepaket dar, das direkt installiert werden kann. Die Software ist meist für eine bestimmte Architektur kompiliert und nur auf dieser lauffähig. Nach dem Ausführen von

   ```
   rpm -i paket-x.y.z-r.i386.rpm
   ```

   sollte die Software installiert und einsatzbereit sein.

   Im Binary-RPM ist außerdem vermerkt, welche anderen Pakete die Software voraussetzt (Bibliotheken, Daemons etc.).

2. Source-RPMs

   Diese RPMs enthalten den kompletten Sourcecode für ein Binary RPM. Sie sind (theoretisch) unabhängig von der Architektur und müssen erst kompiliert werden, bevor sie einsatzfähig sind.

   Um ein Binary-RPM zu erhalten, genügt es meist,

   ```
   rpm -rebuild paket-x.y.z-r.src.rpm
   ```

   einzugeben, worauf man in einem speziellen Verzeichnis (`/usr/src/redhat/RPMS/{i386,alpha,...}/` bei RedHat, `/usr/src/packages/RPMS/{i386,alpha,...}/` bei SuSE) das Binary-RPM findet.

   Den Inhalt eines Source-RPMs kann man mit

```
rpm -i paket-x.y.z-r.src.rpm
```

installieren und dann im entsprechenden Quellverzeichnisbaum finden. (siehe Seite 3.1)

2 Erzeugen von RPM-Paketen

2.1 Grobablauf

Das Erzeugen eines RPMs gliedert sich grob in folgende Schritte:

1. Sammeln aller Sourcecodes, die zum Erzeugen des RPMs notwendig sind
2. evtl. Modifikation der Sourcecodes, damit sie auf dem Zielsystem kompilierbar sind (⇒ Patches)
3. Erzeugen der `spec`-Datei mit den Informationen für RPM, die zum Erzeugen des Paketes notwendig sind
4. Erzeugen des Paketes mit RPM
5. Hinzufügen von Installations-/Deinstallationsskripten zum RPM

2.2 Die `spec`-Datei

Die `spec`-Datei enthält eine Beschreibung des Paketes zusammen mit Informationen zur Erzeugung des Paketes und den durch das Paket installierten Dateien.

Die Datei wird gemäß allgemeiner Konvention benannt mit

„Paketname“-„Versionsnummer“-„Releasenummer“.spec

Innerhalb der `spec`-Datei werden folgende Syntax-Elemente verwendet:

- `# Kommentar`
- `Schlüsselwort: Wert`
- `%Makroname Makroparameter` - mittels dieser Schreibweise werden auch einzelne Sektionen der Datei getrennt
- `%{Makroname}` - Ersetzung von Werten (mehr zu Makros später)

Eine einfache `spec`-Datei „`bla-0.1.1-42.spec`“ könnte also ungefähr aussehen wie in Abbildung 1.

Die Bedeutung der einzelnen Schlüsselworte – auch Tags genannt – wird im nächsten Abschnitt erläutert.

```
Summary: Ein Paket, dass bla tut.
Name: bla
Version: 0.1.1
Release: 42
Copyright: GPL
Group: Applications/blubb
Source: http://www.somewhere.net/sources/bla-0.1.1.tar.gz
Patch: bla-xyz-bugfix.patch
BuildRoot: /var/tmp/%{name}-buildroot
%description
Das Programm bla macht bla. Es ist noch nicht weit fortgeschritten,
aber bereits stabil. ...

# Kommentar: keine besonderen Aktionen notwendig zum Entpacken des Pakets
%prep
%setup -q
# wir haben einen (imaginaeren) Patch fuer das Paket
%patch -p1
# jetzt kommen Anweisungen zum Kompilieren des Pakets (Shellskript)
%build
./configure --some-weird-option
make something
# wie wird das Paket installiert?
%install
# wir benutzen das BuildRoot-Feature; Pakete mit autoconf kommen damit
# sehr gut zurecht :-)
make DESTDIR=$RPM_BUILD_ROOT
# wie wird aufgeraeumt?
%clean
rm -rf $RPM_BUILD_ROOT
# welche Dateien gehoeren zum finalen RPM?
%files
/usr/bin/bla
/usr/man/man1/bla.1
# Historie muss auch sein!
%changelog
* Thu Jul 06 2000 Tino Schwarze <tisc@informatik.tu-chemnitz.de>
- first RPM release of bla (it took me some time to get the RPM to build,
  therefore the high release number)
```

Abbildung 1: Eine minimale spec-Datei

2.3 Die Tags im spec-File

Tag	Zweck
`Summary`	Einzeilige Kurzbeschreibung des Paketes. Diese Beschreibung wird von Installationstools neben dem Paketnamen angezeigt.
`Name`	Der Name des zu erzeugenden RPMs. RPM-Dateien folgen alle einer festen Namenskonvention (SuSE hält sich zum Beispiel nicht daran).
`Version`	Die Versionsnummer des Paketes. Die Versionsnummer wird Bestandteil des erzeugten RPM. Sie sollte gewissen Konventionen genügen, da RPM für Updates u. ä. Versionsnummern vergleichen muß.
`Release`	Die Releasenummer des Paketes. Die Releasenummer sollte jedesmal erhöht werden, wenn das RPM erstellt wird. Mit einer neuen Versionsnummer beginnt sie wieder bei 1.
`Copyright`	Das Copyright des Paketes. z. B.: GPL, BSD, distributable, commercial, etc.
`Group`	Einordnung des Paketes entsprechend seiner Bestimmung. Zum RPM Programm selbst gehört eine Datei `GROUPS`, die mögliche Gruppen auflistet (wie z.B. „Applications/Archiving").
`Source`	Gibt an, aus welchen Quellarchiven das Paket erstellt wird. Es können/müssen mehrere, durchnummerierte `Source`-Tags angegeben werden, wenn das Paket aus verschiedenen Quellen erstellt wird. Zu den Quellen können auch vorgefertigte Konfigurationsdateien u. ä. zählen.
`Patch`	Auflistung der Patches (durchnummeriert).
`BuildRoot`	Platz für temporäre Installation des Pakets. Bevor ein Paket erzeugt wird, muss es installiert werden. Damit das Paket nicht in das bestehende System installiert wird, kann man ein Buildroot verwenden (genaue Beschreibung später).

2.4 %prep-Sektion

Im Prinzip könnte man hier ein Shell-Skript hinschreiben, das die Sourcen auspackt und für das anschliessende `make` vorbereitet. Um Arbeit zu sparen gibt es dafür allerdings Makros.

Das `%setup`-Makro sorgt für das Auspacken der Sourcecodes und wechselt danach in das entpackte Verzeichnis. Es gibt einige Optionen für das Makro:

`-n name` Angabe des Verzeichnisses, das aus dem `tar`-Archiv ausgepackt wird (falls verschieden von `Paket-x.y.z`

-c verz erzeugt und wechselt in Verzeichnis verz vor dem Auspacken

-b # nur sinnvoll bei mehreren Quellarchiven: Quellarchiv Nummer # auspacken bevor in das Verzeichnis gewechselt wird

-a # dito, nur auspacken vom Quellarchiv Nummer # nachdem in das Verzeichnis gewechselt wird

-T verhindert das Auspacken von Quellarchiv 0, wird bei mehreren Quellarchiven benötigt

-D das Verzeichnis wird *nicht* gelöscht vor dem Auspacken - notwendig bei mehreren Quellarchiven

Das %patch-Makro kommt als nächstes zum Zuge. Sollen mehrere Patches eingespielt werden, benutzt man einfach die Makros `%patch0` bis `%patchX`. Als Option ist oft `-p#` notwendig, die mit der entsprechenden Option vom `patch`-Programm übereinstimmt, also die Anzahl der Verzeichnisse angibt, die abgeschnitten werden sollen.

2.5 %build-Sektion

Für diesen Abschnitt der `spec`-Datei existieren keine speziellen Makros. Sie enthält nur (mehr oder weniger umfangreiches) Shellskript, das im ersten Source-Verzeichnis ausgeführt wird und das Kompilieren des Pakets veranlassen soll.

Die Shellvariable `$RPM_OPT_FLAGS` kann verwendet werden, um systemspezifische Compiler-Optimierungen zu erhalten.

2.6 %install-Sektion

Auch hier gibt es keine speziellen Makros. In diesem Abschnitt steht notwendiges Shellskript, um das Paket zu installieren. Meistens ist dazu nur ein `make install` notwendig, manchmal muß noch Vor- oder Nacharbeit geleistet werden.

Die Shellvariable `$RPM_BUILD_ROOT` enthält den Wert von `BuildRoot` aus dem Kopf der `spec`-Datei und muß entsprechend verwendet werden. Siehe dazu später mehr.

2.7 %clean-Sektion

Diese Kommandos werden ausgeführt, um das Buildroot zu säubern.

2.8 `%{pre,post}{un,}install`-Sektionen

In diesen Sektionen können Shellskripte untergebracht werden, die vor oder nach der De-/Installation des fertigen RPMs ausgeführt werden.

Sektion	Ausführungszeitpunkt
`%preinstall`	direkt vor der Installation des RPMs
`%postinstall`	direkt nach der Installation des RPMs
`%preuninstall`	direkt vor der Deinstallation des RPMs
`%postuninstall`	direkt nach der Deinstallation des RPMs

2.9 `%files`-Sektion

Hier werden alle Dateien aufgelistet, die in das Binary RPM aufgenommen werden sollen. Es existieren einige Makros, um die Dateien genauer zu klassifizieren:

`%defattr (mode, owner, group)` Gibt die Standard-Attribute für Dateien an. `mode` ist eine Oktalzahl, wie sie `chmod` übergeben wird und `owner` und `group` sind die jeweiligen Nutzer- und Gruppennamen.

Möchte man ein Attribut nicht angeben, genügt ein „-“, um den Standardwert zu verwenden.

`%doc` Markiert die Datei als Dokumentation. Es wird dann automatisch in das entsprechende Verzeichnis kopiert.

`%config` Markiert die Datei als Konfigurationsdatei. Solche Dateien werden beim Installieren/Deinstallieren gesondert behandelt, um eventuelle Änderungen an der Konfiguration zu erhalten.

`%dir` Markiert den Dateinamen als Verzeichnis. Wird ein Verzeichnis ohne das `%dir`-Makro angegeben, werden *alle* Dateien in diesem Verzeichnis zum Paket hinzugefügt.

`%attr (mode, owner, group)` Wie `%defattr`; explizite Angabe der Dateiattribute. Dies ermöglicht es z. B. mit Hilfe eines Buildroots als „non-root“ RPMs zu erzeugen, die SUID-Binaries enthalten.

Man kann die Dateiliste auch extern halten und mit `%files -f datei` einlesen lassen. Dann sollte man die Liste allerdings entweder im Verlauf von `%build` automatisch erzeugen oder zu den Quelldateien hinzufügen.

3 Build-Prozeß

In diesem Abschnitt wird genauer auf das Erzeugen des RPMs eingegangen. Schritt für Schritt wird die Erzeugung eines RPMs erläutert. Dabei wird davon ausgegangen, daß eine `spec`-Datei schon erzeugt wurde.

3.1 Der Quellverzeichnisbaum

Es wird ein spezieller Verzeichnisbaum zum Erzeugen von RPMs benötigt. Meist ist dieser schon eingerichtet unter `/usr/src/{redhat,packages}/`, je nach Distribution.

Dort finden sich die folgenden Verzeichnisse:

`BUILD` Hier werde die Quellen kompiliert. Das Auspacken der Quellarchive etc. findet automatisch in diesem Verzeichnis statt. Das o. g. Paket `bla-0.1.1-42.src.rpm` würde also in `/usr/src/packages/BUILD/bla-0.1.1/` kompiliert. (Per Konvention sollte das TAR-Archiv `bla-0.1.1.tar.gz` das Verzeichnis `bla-0.1.1/` enthalten!)

`SOURCES` Hier werden alle Dateien abgelegt, die in einer `spec`-Datei unter `Source#:` angegeben werden. Deshalb sollte man darauf achten, möglichst eindeutige Namen für die Quellen zu vergeben und nicht `patch1`, `Makefile.neu` o. ä.

`SPECS` Hier werden die `spec`-Dateien abgelegt.

`RPMS` Hier landen die Binary RPMs, geordnet nach Architekturen. Es gibt pro Architektur ein Unterverzeichnis, also z. B. `i386`, `alpha` und `noarch` (für Pakete, die architekturunabhängig sind, wie Shellskripte, Tcl/Tk u. ä.).

`SRPMS` Hier werden die Source-RPMs untergebracht. Ein Source-RPM wird durch Angabe einer speziellen Option beim Erzeugen des Binary RPMs mit generiert.

3.2 Test-Build

Zuerst sollte man das Paket ohne RPM kompilieren könnnen. Dazu entpackt man alle Sourcen (`rpm -bp paket-x.y.z.spec` leistet gute Dienste) und versucht, das Paket „von Hand“ zu kompilieren. Falls dies nicht auf Anhieb gelingt, empfiehlt es sich, eine Kopie des Sourcetrees anzulegen (`paket-x.y.z.orig/`) und dann erst entsprechende Änderungen vorzunehmen.

RPM erlaubt es *nicht*, einen Source-Tree von Hand zu präparieren und dann davon ein RPM-Paket zu erzeugen! Es muß immer der Zyklus „Entpacken – Patchen – Kompilieren – Installieren – RPM(s) erzeugen“ durchlaufen werden! Diese „Restriktion“ wurde eingeführt, um sicherzustellen, daß sich ein Source-RPM immer kompilieren läßt.

Eventuelle Änderungen muß man z. B. im `BUILD`-Verzeichnis mit

```
diff -urN paket-x.y.z.orig paket-x.y.z >../SOURCES/paket-x.y.z.patch
```

in Patch-Form bringen. Dabei sollte man sicherstellen, daß keine Binaries in den Patch geraten – `make clean` vor dem `diff` ist sicherlich von Nutzen.

3.3 Erzeugen der %files-Liste

Die Verwendung eines Buildroots ist hier von Vorteil. Man sieht sich an, was alles durch `make install` der Software installiert wird und trägt alles notwendige in die `%files`-Liste ein.

3.4 Erzeugen des RPMs

Jetzt sollte die `spec`-Datei vollständig sein. Mittels

```
rpm -ba paket-x.y.z.spec
```

kann man nun versuchen, das Binary- und Source-RPM zu erstellen. Eventuell liefert die Option `-v` hilfreiche Erkenntnisse.

Wenn alles gut geht, kann man mit

```
rpm -i paket-x.y.z-r.i386.rpm
```

das neu erschaffene RPM installieren und auf Funktionsfähigkeit überprüfen. Eventuell müssen auch noch (Un-)Install-Skripte hinzugefügt werden oder es ist Feinschliff für die `%files`-Liste nötig. In diesem Fall wird man mit

```
rpm -e paket
```

das installierte Paket wieder los.

4 Buildroot näher erklärt

Die `Buildroot`-Option von RPM ist extrem nützlich für die Erzeugung von RPMs.

Normalerweise wird während der Erzeugung mit `make install` in das laufende System installiert. Dadurch wird es sehr schwierig, die Integrität des Binary-RPMs zu verifizieren.

Wenn man dagegen ein Buildroot verwendet, kann man diese Probleme umgehen, da das Paket in ein eigenständiges Verzeichnis (z. B. `/var/tmp/paket-x.y.z-buildroot/` installiert wird und somit vom System getrennt bleibt.

Der RPM sucht dann dort die Dateien für das Binary-RPM zusammen und entfernt transparent den zusätzlichen Pfad (er wird in der `%files`-Liste nicht mit angegeben).

Man muß jedoch in der `%install`-Sektion Sorge dafür tragen, daß das fertig kompilierte Paket auch in das Buildroot installiert wird. Programme, die `autoconf` verwenden, kennen die `make`-Variable `DESTDIR`, so daß mit

```
make install DESTDIR=$RPM_BUILD_ROOT
```

das Paket transparent verschoben wird. Bei anderen Programmen ist evtl. ein Patch für das `Makefile` oder etwas `sed`-Magie in der `%build`- oder `%install`-Sektion notwendig.