Red Hat Enterprise Linux 7

Anaconda カスタマイズガイド

インストーラーの表示の変更とカスタムアドオンの作成

Last Updated: 2023-01-29

Red Hat Enterprise Linux 7 Anaconda カスタマイズガイド

インストーラーの表示の変更とカスタムアドオンの作成

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

概要

Anaconda は、Red Hat Enterprise Linux、Fedora、およびそれらの派生物で使用されるインストーラーです。本ガイドでは、カスタマイズに必要な情報を記載しています。 インストーラーの基本機能を拡張したい開発者は、Anaconda アーキテクチャー、そのアドオン API、提供されているヘルパー関数、およびカスタムアドオンの作成に役立つ例に関する情報を見つけることができます。 本ガイドでは、起動メニューのカラースキームや背景、グラフィカルユーザーインターフェイス内のブランド化や色調節などのインストーラーの視覚的側面をカスタマイズするための手順も説明しています。 Red Hat は、Red Hat Enterprise Linux インストールメディアの変更またはカスタマイズ

をサポートしていませ ん。システムテンプレートやライブ CD などの OS イメージをカスタマイズするには、キックスタート を使用してインフラストラクチャー内に一貫したシステムを構築するか、Red Hat Enterprise Linux Image Builder を使用してカスタム OS イメージを作成します。

目次

1. ANACONDA のカスタマイズの概要

Red Hat Enterprise Linux および Fedora インストールプログラムである **Anaconda** では、最新バージョンに多くの改良が追加されました。これらの改善の 1 つは、カスタマイズの可能性が強化されました。基本的なインストーラー機能を拡張するためにアドオンを作成できます。また、グラフィカルユーザーインターフェイスの外観を変更するオプションなどがあります。

本書では、以下をカスタマイズする方法について説明します。

- ブートメニュー: 事前設定されたオプション、色スキーム、および背景
- グラフィカルインターフェイスの外観: ロゴ、背景、製品名
- インストーラーの機能: グラフィカルユーザーインターフェイスおよびテキストユーザーインターフェイスに新しいキックスタートコマンドと新しい画面を追加して、インストーラーを強化できるアドオン

本ガイドで説明しているトピックの中には、多くの知識を必要とするものもあります。特に、カスタムの **Anaconda** アドオンの開発には Python の知識、ブートメニュー変更にはプレーンテキストの設定ファイルの編集、インストーラーの外観のカスタマイズにはコンピューターグラフィックスと CSS (カスケードスタイルシート) の知識がそれぞれ必要になります。

また、本ガイドは、Red Hat Enterprise Linux 7 および Fedora 17 以降にのみ適用されることに注意してください。

重要

本ガイドで説明されている手順は、Red Hat Enterprise Linux 7 または同様のシステム用に記述されています。これ以外のシステムでは、(カスタム ISO イメージを作成する **genisoimage** など) 使用するツールやアプリケーションが異なり、手順を調節する必要がある場合があります。

2. ISO イメージを使った作業

本セクションでは、Red Hat が提供する ISO イメージの抽出と、本ガイドにある手順による変更を含む新規ブートイメージの作成方法について説明します。

2.1. Red Hat Enterprise Linux Boot イメージの抽出

インストーラーのカスタマイズを開始する前に、Red Hat が提供するブートイメージをダウンロードします。このイメージは、本ガイドでの手順を実行するために必要なものです。

アカウントにログインした後に、Red Hat カスタマーポータル から Red Hat Enterprise Linux 7 ブートメディアを取得できます。アカウントには、Red Hat Enterprise Linux 7 イメージをダウンロードするのに十分なエンタイトルメントが必要です。

カスタマーポータルから **Binary DVD** または **Boot ISO** イメージをダウンロードします。本ガイドの手順を使用して変更することができます。**KVM Guest Image** や **Supplementary DVD** などの利用可能なダウンロードも可能です。イメージの **バリアント** (**Server**や **ComputeNode** など) は重要ではありませんが、いずれのバリアントも使用できます。

Binary DVD および Boot ISO ダウンロードに関する詳細なダウンロード手順および説明は、『Red Hat Enterprise Linux 7』を参照してください。

選択した iso イメージのダウンロードが完了したら、以下の手順に従ってコンテンツを抽出し、修正できるようにします。

手順1 ISO イメージの抽出

1. ダウンロードしたイメージをマウントします。

```
# mount -t iso9660 -o loop path/to/image.iso /mnt/iso
```

path/to/image.iso をダウンロードした ISO へのパスで置き換えます。ターゲットとするディレクトリー (**/mnt/iso**) が存在し、そこに他のものがマウントされていないことを確認します。

2. ISO イメージのコンテンツを配置する作業ディレクトリーを作成します。

```
$ mkdir /tmp/ISO
```

3. マウントされたイメージのすべてのコンテンツを新しい作業ディレクトリーにコピーします。**-p** オプションを使用して、ファイルおよびディレクトリーのパーミッションと所有権を保持するようにしてください。

```
# cp -pRf /mnt/iso /tmp/ISO
```

4. イメージをアンマウントします。

```
# umount /mnt/iso
```

展開が終了したら、ISO イメージは **/tmp/ISO** に抽出され、ここでコンテンツの修正ができます。続ける「ブートメニューのカスタマイズ」また「インストーラーアドオンの開発」.変更が完了したら、の手順を使用して、変更された新しい ISO イメージを作成します。「カスタムブートイメージの作成」.

2.2. product.img ファイルの作成

product.img イメージファイルは、インストーラーのランタイムに既存ファイルを置き換える、または新たなファイルを追加する、ファイルを含むアーカイブです。起動中の Anaconda はブートメディアの **images/** ディレクトリーからこのファイルを読み込みます。そしてこのファイル内にあるファイルを使用してインストーラーのファイルシステム内にある同一名のファイルを置換します。インストーラーのカスタマイズにはこれが必要になります (例えば、デフォルトのイメージをカスタムのもので置き換える場合)。**product.img** イメージに含まれるディレクトリーは、インストーラーと同じディレクトリー構造である必要があります。

本ガイドで説明する 2 つのトピックでは、製品のイメージを作成する必要があります。以下の表では、ディレクトリー構造内におけるイメージファイルの正しい場所を示しています。

表1 アドオンおよび Anaconda Visuals の場所

カスタムコンテンツのタイプ	ファイルシステムの場所
Pixmaps (ロゴ、サイドバー、トップバー、など)	**/usr/share/anaconda/pixmaps/**

カスタムコンテンツのタイプ	ファイルシステムの場所
インストール進捗画面のバナー	**/usr/share/anaconda/pixmaps/rnotes/en/**
GUI スタイルシート	**/usr/share/anaconda/anaconda-gtk.css**
Installclasses (製品名変更用)	**/run/install/product/pyanaconda/installclasses/**
Anaconda アドオン	**/usr/share/anaconda/addons/**

以下の方法で有効な **product.img** ファイルを作成します。

手順2 product.img の作成

1. **/tmp** などの作業ディレクトリーに移動し、**product/** という名前のサブディレクトリーを作成します。

```
$ cd /tmp
```

```
$ mkdir product/
```

2. 置換するファイルの場所と同じディレクトリー構造を作成します。例えば、アドオンをテストする場合、これがインストールシステムの **/usr/share/anaconda/addons** に配置されているとすると、作業ディレクトリー内で同じ構造を作成します。

```
$ mkdir -p product/usr/share/anaconda/addons
```

注記

インストーラーのランタイムファイルシステムをブラウズするには、インストールを起動し、仮想コンソール 1 に切り替え (**Ctrl**+**Alt**+**F1**) その後に 2 つ目の **tmux** ウィンドウに切り替えます (**Ctrl**+**b 2**)。これでファイルシステムをブラウズするためのシェルプロンプトが開きます。

3. カスタマイズしたファイル (この例では、Anaconda 用のカスタムアドオン) を新規作成したディレクトリーに配置します。

```
$ cp -r ~/path/to/custom/addon/ product/usr/share/anaconda/addons/
```

4. インストーラーに追加するすべてのファイルについて、上記の 2 つのステップを繰り返します (ディレクトリー構造の作成および変更済みファイルの移動)。

5. このディレクトリーの root に **.buildstamp** ファイルを作成します。これは、**product.img** ファイルになります。このファイルは、**product.img** ファイルになります。以下は、Red Hat Enterprise Linux 7.4 からの **.buildstamp** ファイルの例です。

```
[Main]
Product=Red Hat Enterprise Linux
Version=7.4
BugURL=https://bugzilla.redhat.com/
```

```
IsFinal=True
UUID=201707110057.x86_64
[Compose]
Lorax=19.6.92-1
```

IsFinal パラメーター。そのイメージが製品のリリース (一般公開) バージョンである (**True**) か、アルファ、ベータ、内部マイルストーンなどのプレリリース版である (**False**) かを指定します。

6. **product/** ディレクトリーに移動し、**product.img** アーカイブを作成します。

```
$ cd product
```

```
$ find . | cpio -c -o | gzip -9cv > ../product.img
```

これで **product.img** ファイルが **product/** ディレクトリーの 1 つ上のレベルに作成されます。

7. **product.img** ファイルを抽出した ISO イメージの **images/** ディレクトリーに移動します。

この手順が完了したら、カスタムファイルは適切なディレクトリーに配置されます。続行できます「カスタムブートイメージの作成」変更を含む新しいブータブル ISO イメージを作成します。**product.img** ファイルはインストーラーの起動時に自動的に読み込まれます。

注記

ブート用メディアに **product.img** ファイルを追加する代わりに、別の場所にこのファイルを追加し、ブートメニューで **inst.updates=** ブートオプションを使用して読み込むことができます。この場合、イメージファイルは任意の名前を指定することができ、インストールシステムからこの場所にアクセス可能な限り、任意の場所 (USB フラッシュドライブ、ハードディスク、HTTP、FTP、または NFS サーバー) に配置できます。

Anaconda 起動オプションの詳細は『Red Hat Enterprise Linux 7 インストールガイド』を参照してください。

2.3. カスタムブートイメージの作成

Red Hat が提供するブートイメージのカスタマイズが完了したら、加えた変更を含む新規イメージを作成します。以下の手順に従います。

手順3 ISO イメージの作成

1. すべての変更が作業ディレクトリーに含まれていることを確認してください。例えば、アドオンをテストする場合は、**product.img** を **images/** ディレクトリーに配置します。

2. 作業ディレクトリーが抽出した ISO イメージのトップレベルのディレクトリーであることを確認します。例えば、 **/tmp/ISO/iso**。

3. **genisoimage** を使用して、新規 ISO イメージを作成します。

```
# genisoimage -U -r -v -T -J -joliet-long -V "RHEL-7.1 Server.x86_64" -volset "RHEL-7.1 Server.x86_64" -A "RHEL-7.1 Server.x86_64" -b isolinux/isolinux.bin -c isolinux/boot.cat -no-emul-boot -boot-load-size 4 -boot-info-table -eltorito-alt-boot -e images/efiboot.img -no-emul-boot -o ../NEWISO.iso .
```

上記の例では、以下のようになります。

- 同じディスクにファイルを読み込む場所を必要とするオプションに **LABEL=** ディレクティブを使用している場合は、**-V**、**-volset**、**-A** のオプションの値がイメージのブートローダー設定と一致することを確認してください。ブートローダーの設定 (BIOS の場合は **isolinux/isolinux.cfg**、UEFI の場合は **EFI/BOOT/grub.cfg**) が **inst.stage2=LABEL=*disk_label*** スタンザを使用して同じディスクからインストーラーの 2 番目のステージを読み込む場合は、ディスクラベルが一致する必要があります。

重要

ブートローダー設定ファイルで、ディスクラベルのすべてのスペースを **\x20** に置き換えます。例えば、**RHEL 7.1** というラベルの ISO イメージを作成する場合、ブートローダー設定では **RHEL\x207.1** を使用してこのラベルを参照します。

- **-o** オプションの値 (**-o ../NEWISO.iso**) を、新しいイメージのファイル名に置き換えます。この例の値は、現在のディレクトリーの上に **NEWISO.iso** ファイルを作成します。

このコマンドに関する詳細は、**genisoimage(1)** man ページを参照してください。

4. MD5 チェックサムをイメージに埋め込みます。このステップを実行しないと、イメージ検証チェック (ブートローダー設定の **rd.live.check** オプション) に失敗し、インストールを続行できなくなります。

```
# implantisomd5 ../NEWISO.iso
```

この例では、*../NEWISO.iso* をこの前のステップで作成した ISO イメージのファイル名と場所で置き換えます。

ここまでの手順が完了したら、この新規 ISO イメージを物理メディアやネットワークサーバーに書き込んで物理的ハードウェア上で起動するか、これを使用して仮想マシンのインストールを開始できます。ブートメディアおよびネットワークサーバーの準備方法は 『Red Hat Enterprise Linux 7 インストールガイド』 を、ISO イメージを使った仮想マシンの作成方法は 『Red Hat Enterprise Linux 7 仮想化スタートガイド』 を参照してください。

3. ブートメニューのカスタマイズ

本セクションでは、*ブートメニュー*のカスタマイズに必要な情報を提供しています。通常、このメニューでは、**Install Red Hat Enterprise Linux**、**Boot from local drive**、**Rescue an installed system** などのオプションを選択できます。これらのオプションはカスタマイズ可能で、新たなオプションを追加したり、(色や背景などの) 視覚スタイルを変更できます。

インストールメディアには 2 つのブートローダーがあります。**ISOLINUX** ブートローダーは BIOS ファームウェアのシステムで使用され、**GRUB2** ブートローダーは UEFI ファームウェアが搭載されているシステムで使用されます。いずれも、Red Hat が提供する AMD64 システムおよび Intel 64 システム用のイメージすべてに存在します。

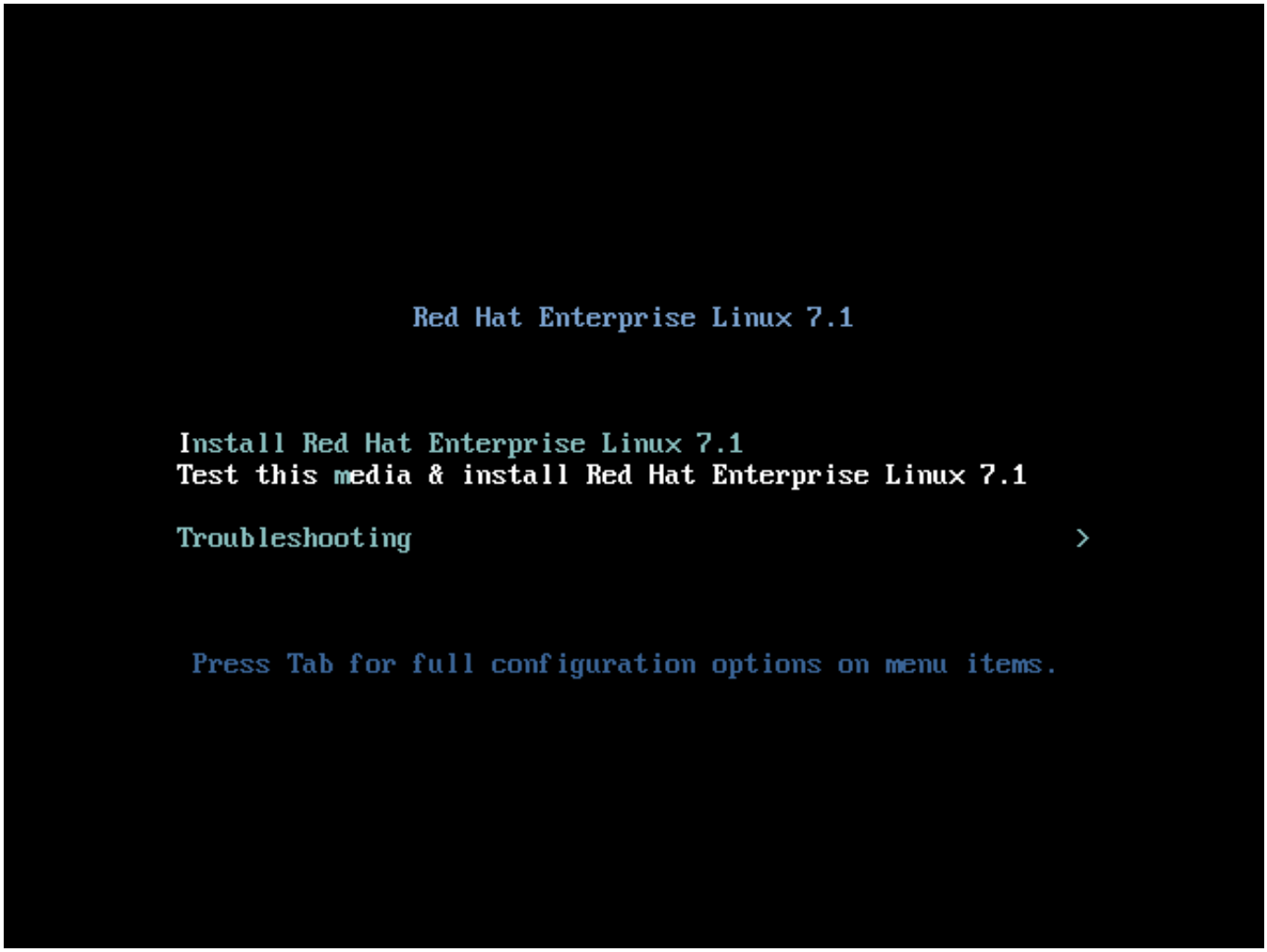
ブートメニューオプションのカスタマイズは、特に Kickstart で便利なものです。インストールを開始する前に、キックスタートファイルをインストーラーに提供する必要があります。通常、これは既存の起動オプションを手動で編集し、**inst.ks=** 起動オプションを追加して実行できます。メディア上のブートローダー設定ファイルを編集する場合は、このオプションを事前に設定されたエントリーのいずれかに追加できます。

ブートローダーのカスタマイズを開始する前に、以下に従ってください。手順1「ISO イメージの抽出」変更する ISO イメージを作業ディレクトリーに解凍します。変更が完了したら、次の手順に従います。手順 3「ISO イメージの作成」新しい起動可能な ISO イメージを作成します。

3.1. BIOS ファームウェアのシステム

BIOS ファームウェアのシステムでは、**ISOLINUX** ブートローダーが使用されます。

図1 ISOLINUX ブートメニュー



[D]

ブートメディア上の **isolinux/isolinux.cfg** 設定ファイルには、カラースキームとメニュー構造 (エントリーおよびサブメニュー) を設定するディレクティブが含まれています。

設定ファイルでは、Red Hat Enterprise Linux のデフォルトメニューエントリーである **Test this media & Install Red Hat Enterprise Linux 7** は、以下のブロックで定義されます。

```
label check
  menu label Test this ^media & install Red Hat Enterprise Linux 7.1
  menu default
  kernel vmlinuz
  append initrd=initrd.img inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64 rd.live.check quiet
```

上記の例で注目すべきオプションは以下の通りです。

- **menu label**: メニューでエントリーの名前を指定します。**^** 文字は、キーボードショートカット (**m** キー) を決定します。
- **menu default** - このオプションがデフォルトで選択されるようにします。ただし、これはリスト内の最初のオプションではありません。

- **kernel**: インストーラーカーネルを読み込みます。多くの場合、変更は不要です。
- **append**: 追加のカーネルオプションが含まれます。**initrd=** オプションおよび **inst.stage2** オプションは必須です。他のオプションを追加することもできます。

  Anaconda に固有する使用可能なオプションについては、『Red Hat Enterprise Linux 7 インストールガイド』 に記載されています。主なオプションは **inst.ks=** で、キックスタートファイルの場所を指定できます。つまり、起動 ISO イメージ上に Kickstart ファイルを配置して、このオプションによりこれを使用できます。たとえば、**kickstart.ks** というファイルをイメージのルートディレクトリーに配置し、**inst.ks=hd:LABEL=RHEL-7.1\x20x86_64:/kickstart.ks** を使用します。

  また、**dracut** オプションを使用することもできます。詳細は、man ページの **dracut.cmdline(7)** を参照してください。

重要

上記の **inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64** オプションにあるように、特定のドライブを参照するディスクラベルを使用する場合は、すべての空白を **\x20** に置き換えます。

メニューエントリー定義の一部ではない他の重要なオプションには以下のものがあります。

- **timeout**: デフォルトのメニューエントリーが自動的に使用されるまでのブートメニューの表示時間を指定します。デフォルト値は **600** で、メニューは 60 秒間表示されます。この値を **0** に設定すると、タイムアウトは完全に無効となります。

注記

ヘッドレスインストールを実行する場合は、この値を **1** のような低いものにすると、デフォルトの 60 秒間のタイムアウトを待機する必要がなくなります。

- **menu begin** および **menu end**: *サブメニューブロックの開始と終了を決定するため、トラブルシューティングや サブメニューでのグループ化などのオプションを追加できます。これにより、サブメニューにトラブルシュートなどのオプションを追加したり、それらをグループ化することができます。オプションが 2 つのシンプルなサブメニューの例 (1 つは続行する、もう 1 つはメインメニューに戻る) は以下のようになります。*

```
menu begin ^Troubleshooting
  menu title Troubleshooting

label rescue
  menu label ^Rescue a Red Hat Enterprise Linux system
  kernel vmlinuz
  append initrd=initrd.img inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64 rescue quiet

menu separator

label returntomain
  menu label Return to ^main menu
  menu exit

menu end
```

上記の例で分かるように、サブメニューエントリーの定義は通常のメニューエントリーと同様のものですが、**menu begin** と **menu end** ステートメント間でグループ化されます。2 つ目のオプションの **menu exit** 行はサブメニューを終了し、メインメニューに戻ります。

- **menu background**: メニュー背景です。単色 (以下の **menu clor** の色を参照)、または PNG、PNG、または LSS16 形式のイメージのいずれかになります。イメージを使用する場合は、**set resolution** ステートメントを使用して、その位置がセットされた解決に対応していることを確認してください。デフォルトの機能は 640x480 です。

- **menu color**: メニュー要素の色を指定します。完全なフォーマットは以下のとおりです。

  > **menu color *element ansi foreground background shadow***

  このコマンドの重要な部分は、*element* (色を適用する要素を指定) と *foreground* および *background* (実際の色を指定) になります。色は 16 進形式で ***#AARRGGBB*** 表記で示されます。最初の 2 桁 (*AA*) で不透明度を指定します (**00** が完全な透明度、**ff** が完全な不透明度)。

  利用可能な要素、ANSI 値、シャドウ設定、その他の視覚に関するカスタマイズオプションは、Syslinux Wiki を参照してください。

- **menu help *textfile***: メニューエントリーを作成します。このエントリーを選択すると、ヘルプテキストファイルが表示されます。

ISOLINUX 設定ファイルの完全なオプション一覧は、Syslinux Wiki を参照してください。

3.2. UEFI ファームウェアのシステム

UEFI ファームウェアのシステムでは GRUB2 ブートローダーが使用されます。

GRUB2 設定ファイルはブートメディア上にある **EFI/BOOT/grub.cfg** です。設定ファイルには事前設定済みのメニューエントリーと、ブートメニューの外観と機能を制御する他のディレクティブが含まれています。

設定ファイルでは、Red Hat Enterprise Linux のデフォルトメニューエントリー (**Test this media & install Red Hat Enterprise Linux 7.1**) が以下のブロックで定義されます。

```
menuentry 'Test this media & install Red Hat Enterprise Linux 7.1' --class fedora --class gnu-linux
--class gnu --class os {
  linuxefi /images/pxeboot/vmlinuz inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64 rd.live.check quiet
  initrdefi /images/pxeboot/initrd.img
}
```

上記の例で注目すべきオプションは以下の通りです。

- **menuentry**: メニューエントリーを定義するオプションです。エントリーのタイトルは引用符または二重引用符 (' または ") です。**--class** オプションを指定すると、メニューエントリーを異なる クラス にグループ化できます。このクラスは、GRUB2 テーマを使用してスタイルを変更できます。

  注記

  各メニューエントリーの定義は上記の例のように中括弧 (**{}**) で囲む必要があります。

- **linuxefi**: 起動するカーネル (上記の例では **/images/pxeboot/vmlinuz**) と追加オプションを定義します。これらのオプションをカスタマイズしてブートエントリーの動作を変更します。

  Anaconda に固有する使用可能なオプションについては、『Red Hat Enterprise Linux 7 インストー

ルガイド』 に記載されています。主なオプションは **inst.ks=** で、キックスタートファイルの場所を指定できます。つまり、起動 ISO イメージ上に Kickstart ファイルを配置して、このオプションによりこれを使用できます。たとえば、**kickstart.ks** というファイルをイメージのルートディレクトリーに配置し、**inst.ks=hd:LABEL=RHEL-7.1\x20x86_64:/kickstart.ks** を使用します。

また、**dracut** オプションを使用することもできます。詳細は、man ページの **dracut.cmdline(7)** を参照してください。

重要

上記の **inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64** オプションにあるように、特定のドライブを参照するディスクラベルを使用する場合は、すべての空白を **\x20** に置き換えます。

- **initrdefi**: 読み込む初期 RAM ディスク (initrd) イメージの場所。

grub.cfg 設定ファイルで使用する他のオプションには以下のものがあります。

- **set timeout**: デフォルトのメニューエントリーが自動的に使用されるまでのブートメニューの表示時間を指定します。デフォルト値は **60** で、メニューは 60 秒間表示されます。この値を **-1** に設定すると、タイムアウトを完全に無効にします。

注記

ヘッドレスインストールを実行する場合は、この値を **0** にすると、デフォルトのブートエントリーが直ちにアクティベートされます。

- **submenu**: *サブメニューブロック*の定義。これを使用するとサブメニューが作成でき、その下にあるエントリーをメインメニューに表示するのではなく、グループ化できます。デフォルト設定では **Troubleshooting** サブメニューがあり、これには既存システムを修復するためのエントリーが含まれます。

  エントリーのタイトルは引用符または二重引用符 (' または ") です。

  submenu ブロックには上記のように 1 つ以上の **menuentry** 定義が含まれ、ブロック全体は中括弧 (**{}**) で囲まれています。以下に例を示します。

```
submenu 'Submenu title' {
  menuentry 'Submenu option 1' {
    linuxefi /images/vmlinuz inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64 xdriver=vesa nomodeset
quiet
    initrdefi /images/pxeboot/initrd.img
  }
  menuentry 'Submenu option 2' {
    linuxefi /images/vmlinuz inst.stage2=hd:LABEL=RHEL-7.1\x20x86_64 rescue quiet
    initrdefi /images/initrd.img
  }
}
```

- **set default**: デフォルトで選択されるエントリーを指定します。エントリー番号は **0** から始まることに注意してください。**3 番目** のエントリーをデフォルトエントリーに設定する場合は、**set default=2** を設定します。

- **theme**: GRUB2 テーマファイルを含むディレクトリーの場所テーマを使用することで、背景やフォント、特定要素の色などのブートローダーの視覚的側面をカスタマイズできます。

  テーマファイル形式の完全な説明は本ガイドのスコープ外となります。カスタムテーマの作成に関する情報は、GNU GRUB Manual 2.00 を参照してください。

ブートメニューのカスタマイズに関する詳細情報はGNU GRUB Manual 2.00 を参照してください。**GRUB2** に関する一般的な情報は、『Red Hat Enterprise Linux 7 システム管理者のガイド』 を参照してください。

4. グラフィカルユーザーインターフェイスのブランド化と色調節

本セクションでは、**Anaconda** インストーラーのグラフィカルユーザーインターフェイス (GUI) の外観を変更する方法について説明します。

Anaconda の外観をカスタマイズするために変更可能な GUI の要素は複数あります。カスタマイズには **product.img** ファイルを作成し、これに *installclass* (インストーラーで表示される製品名を変更する) と独自のブランド化資料を格納します。この **product.img** ファイルはインストールイメージではありません。これは完全インストール ISO イメージを補完するもので、カスタマイズを読み込んで使用することで、デフォルトでブートイメージに含まれているファイルを上書きします。

見る「ISO イメージを使った作業」Red Hat が提供するブートイメージの抽出、**product.img** ファイルの作成、およびこのファイルを ISO イメージに追加する方法については、を参照してください。

4.1. グラフィカル要素のカスタマイズ

インストーラーのグラフィカル要素で変更可能なものは、インストーラーランタイムファイルシステムの **/usr/share/anaconda/pixmaps/** ディレクトリーに保存されています。このディレクトリーには以下のファイルが格納されています。

```
pixmaps
├── anaconda-selected-icon.svg
├── dialog-warning-symbolic.svg
├── right-arrow-icon.png
├── rnotes
│   └── en
│       ├── RHEL_7_InstallerBanner_Andreas_750x120_11649367_1213jw.png
│       ├── RHEL_7_InstallerBanner_Blog_750x120_11649367_1213jw.png
│       ├── RHEL_7_InstallerBanner_CPAccess_CommandLine_750x120_11649367_1213jw.png
│       ├── RHEL_7_InstallerBanner_CPAccess_Desktop_750x120_11649367_1213jw.png
│       ├── RHEL_7_InstallerBanner_CPAccess_Help_750x120_11649367_1213jw.png
│       ├── RHEL_7_InstallerBanner_Middleware_750x120_11649367_1213jw.png
│       ├── RHEL_7_InstallerBanner_OPSEN_750x120_11649367_1213cd.png
│       ├── RHEL_7_InstallerBanner_RHDev_Program_750x120_11649367_1213cd.png
│       ├── RHEL_7_InstallerBanner_RHELStandardize_750x120_11649367_1213jw.png
│       └── RHEL_7_InstallerBanner_Satellite_750x120_11649367_1213cd.png
├── sidebar-bg.png
├── sidebar-logo.png
└── topbar-bg.png
```

また、**/usr/share/anaconda/** ディレクトリーには **anaconda-gtk.css** という名前の CSS スタイルシートが格納されており、これがファイル名や、ロゴおよびサイドバー/トップバーの背景といったメインの UI 要素のパラメーターを決定します。このファイルには以下のコンテンツが含まれます。

```
/* vendor-specific colors/images */
```

```
@define-color redhat #021519;

/* logo and sidebar classes for RHEL */

.logo-sidebar {
    background-image: url('/usr/share/anaconda/pixmaps/sidebar-bg.png');
    background-color: @redhat;
    background-repeat: no-repeat;
}

.logo {
    background-image: url('/usr/share/anaconda/pixmaps/sidebar-logo.png');
    background-position: 50% 20px;
    background-repeat: no-repeat;
    background-color: transparent;
}

AnacondaSpokeWindow #nav-box {
    background-color: @redhat;
    background-image: url('/usr/share/anaconda/pixmaps/topbar-bg.png');
    background-repeat: no-repeat;
    color: white;
}

AnacondaSpokeWindow #layout-indicator {
    color: black;
}
```

この CSS ファイルで最も重要な点は、解像度に基づいて拡大縮小を処理する方法です。PNG イメージの背景はスケーリングされず、常に実際の画面に表示されます。代わりに、背景には透過度のある背景があり、スタイルシートは **@define-color** の行で一致する背景色を定義します。そのため、バックグラウンド **イメージ** は背景の **色** にフェードします。これは、イメージのスケーリングを必要とせずに、すべての解像度でバックグラウンドが機能することを意味します。

また、**background-repeat** パラメーターをバックグラウンドのタイル配置に変更することもできます。インストールするすべてのシステムが同じディスプレイの解像度を持つことを保証している場合は、バー全体を埋めるバックグラウンドイメージを使用できます。

rnotes/ ディレクトリーにはバナーのセットを格納します。インストール中は、ほぼ 1 分ごとにバナーイメージが画面下部で循環します。

上記のファイルはどれでもカスタマイズできます。これを行ったら、の指示に従ってください「product.img ファイルの作成」カスタムグラフィックスを使用して独自の **product.img** を作成し、「カスタムブートイメージの作成」変更を含む新しいブータブル ISO イメージを作成します。

4.2. 製品名のカスタマイズ

前のセクションで説明したグラフィカル要素とは別に、インストール中に表示される製品名をカスタマイズすることもできます。この製品名は、全画面の右上に表示されます。

製品名を変更するには、*installation class* を作成する必要があります。以下の例のようなコンテンツで **custom.py** という名前の新規ファイルを作成します。

例1 カスタム Installclass の作成

```
from pyanaconda.installclass import BaseInstallClass
```

```
from pyanaconda.product import productName
from pyanaconda import network
from pyanaconda import nm

class CustomBaseInstallClass(BaseInstallClass):
    name = "My Distribution"
    sortPriority = 30000
    if not productName.startswith("My Distribution"):
        hidden = True
    defaultFS = "xfs"
    bootloaderTimeoutDefault = 5
    bootloaderExtraArgs = []

    ignoredPackages = ["ntfsprogs"]

    installUpdates = False

    _l10n_domain = "comps"

    efi_dir = "redhat"

    help_placeholder = "RHEL7Placeholder.html"
    help_placeholder_with_links = "RHEL7PlaceholderWithLinks.html"

    def configure(self, anaconda):
        BaseInstallClass.configure(self, anaconda)
        BaseInstallClass.setDefaultPartitioning(self, anaconda.storage)

    def setNetworkOnbootDefault(self, ksdata):
        if ksdata.method.method not in ("url", "nfs"):
            return
        if network.has_some_wired_autoconnect_device():
            return
        dev = network.default_route_device()
        if not dev:
            return
        if nm.nm_device_type_is_wifi(dev):
            return
        network.update_onboot_value(dev, "yes", ksdata)

    def __init__(self):
        BaseInstallClass.__init__(self)
```

上記のファイルは (デフォルトのファイルシステムなどの) インストーラーのデフォルト値を指定しますが、この手順に関連する部分は以下のブロックになります。

```
class CustomBaseInstallClass(BaseInstallClass):
    name = "My Distribution"
    sortPriority = 30000
    if not productName.startswith("My Distribution"):
        hidden = True
```

ここでの *My Distribution* をインストーラーで表示したい名前に置き換えます。また、**sortPriority** 属性が **20000** を超えていることを確認します。これにより、新規インストールクラスが最初に読み込まれるようになります。

警告

このファイルの他の属性やクラス名を変更しないでください。変更すると、インストーラーが予期しない動作をする場合があります。

カスタムインストールクラスを作成したら、次の手順に従います。「product.img ファイルの作成」カスタマイズを含む新しい **product.img** ファイルを作成し、「カスタムブートイメージの作成」変更を含む新しいブータブル ISO ファイルを作成します。

5. インストーラーアドオンの開発

5.1. Anaconda とアドオンについて

5.1.1. Anaconda の概要

Anaconda: Fedora、Red Hat Enterprise Linux、およびその他の派生製品に使用されるオペレーティングシステムインストーラーです。これは Python モジュールとスクリプトのセットで、**Gtk** ウィジェット (C で作成)、**systemd** ユニット、および **dracut** ライブラリーといったファイルも含まれています。これらが一体となることで、このツールを使用するとターゲットとなるシステムのパラメーターを設定でき、そのシステムをマシンにセットアップします。インストールプロセスには、主に 4 つのステップがあります。

- インストール先の準備 (通常はディスクのパーティション設定)
- パッケージおよびデータのインストール
- ブートローダーのインストールと設定
- 新規にインストールされたシステムの設定

インストーラーを制御し、インストールオプションを指定する方法は 3 つあります。最も一般的なアプローチは *グラフィカルユーザーインターフェイス* (GUI) を使う方法です。これを使用するとインストール前の設定がほとんどまたは全く要らずにシステムを対話式にインストールできます。複雑なパーティションレイアウトの設定なども含めたすべての一般的なユースケースをカバーしています。

グラフィカルインターフェイスでは **VNC** を使ったリモートアクセスもサポートしており、これを使用するとグラフィックスカードやモニターのないシステムでも GUI の使用が可能になります。ただし、この方法が望ましくない場合もあり、また対話式インストールを希望する場合もあるでしょう。そのようなケースでは、*テキストモード* (TUI) が使用できます。TUI はモノクロのラインプリンターのように動作し、カーソルやカラー、他の高度な機能に対応していないシリアルコンソールでも機能します。テキストモードは、ネットワーク設定や言語オプション、インストール (パッケージ) ソースなどの非常に一般的なオプションしかカスタマイズできないという制限があります。手動によるパーティション設定といったような高度な機能はこのインターフェイスでは利用できません。

Anaconda を使用した 3 つ目のインストール方法では、Kickstart ファイルを使用します。キックスタートファイルを使用すると、インストールを部分的または完全に自動化できます。インストールを完全に自動化するには、すべての必須のエリアを設定する特定のセットのコマンドが必要となります。必要なコマンドが

1 つ以上欠けていると、インストールに対話が必要となります。すべての必須コマンドが揃っていれば、対話せずにインストールが完全に自動で実行されます。

Kickstart では最大限のオプションが提供され、TUI や GUI では不十分となるユースケースをカバーします。**Anaconda** の機能はすべて、Kickstart でサポートされる必要があります。他のインターフェイスは利用可能な全オプションのサブセットに従うだけで、これにより他のインターフェイスがクリアに保たれます。

5.1.2. 初回起動と初期設定

新規にインストールされたシステムの初回起動は、従来はインストールプロセスの一部とみなされていました。以前は、Firstboot **ツール** がこの目的で使用されていたため、新たにインストーラー Red Hat Enterprise Linux システムを登録したり、**Kdump** を設定したりしていました。ただし、**Firstboot** は **Gtk2** や **pygtk2** などのメンテナーンスされていないツールに依存しています。[1] このため、**Initial Setup** と呼ばれる新しいツールが開発され、**Anaconda** のコードを再利用します。これにより、**Anaconda** 向けに開発されたアドオンを **Initial Setup** で簡単に再利用できます。このトピックについては、「Anaconda アドオンの作成」 .

5.1.3. Anaconda と初期設定アドオン

新規オペレーティングシステムのインストールは非常に複雑なユースケースで、ユーザーはそれぞれ微妙に異なることを望みます。多様な望みすべてに合うようにインストーラーを設計すると、稀にしか使用しない機能が散りばめられることになります。このため、インストーラーを現在の形式に書き換えた際に、アドオンに対応するようになりました。

Anaconda のアドオンを使うと、特定のユースケースによっては、グラフィカルおよびテキストベースのユーザーインターフェイスに新たな設定画面と独自の Kickstart コマンドおよびオプションを追加できます。各アドオンは Kickstart に対応している必要があります。GUI と TUI はオプションですが、利用できると非常に便利です。

Red Hat Enterprise Linux (7.1 以降) および Fedora の現在のリリース [2] (21 およびそれ以降) では、**Kdump** アドオンがデフォルトで含まれます。これは、インストール中のカーネルクラッシュダンピングの設定のサポートを追加します。このアドオンには、Kickstart (**%addon com_redhat_kdump** コマンドおよびそのオプション) のフルサポートがあり、テキストベースのインターフェイスとグラフィカルインターフェイスの追加画面として完全に統合されています。本ガイドで詳述している手順を使用すると、他のアドオンも同様に開発して、デフォルトのインストーラーに追加できます。

5.1.4. 追加情報

Anaconda および **初期設定** についての追加情報は、以下のリンクからアクセスできます。

- Fedora Project Wiki の Anaconda ページ にはインストーラーについての詳細情報があります。
- **Anaconda** の、現行バージョンへの開発に関する情報は、Anaconda/NewInstaller の Wiki ページ にあります。
- 『Red Hat Enterprise Linux 7 インストールガイド』のキックスタートを使ったインストールの章では、対応しているコマンドとオプションの一覧など、キックスタートの全ドキュメントを紹介します。
- 『Red Hat Enterprise Linux 7 インストールガイド』のAnaconda を使用したインストールの章では、グラフィカルユーザーインターフェイスのインストールプロセスが説明されています。
- インストール後の設定に使用するツールの情報は、初期設定を参照してください。

5.2. Anaconda のアーキテクチャー

Anaconda は Python モジュールとスクリプトのセットで設定されています。外部パッケージとライブラリーも使用し、このインストーラー専用のものもあります。このツールセットの主要コンポーネントは以下のパッケージになります。

- **pykickstart** - Kickstart ファイルを解析、検証するとともに、インストールを実行する値を保存するデータ構造も提供します。
- **yum** - パッケージのインストールと依存関係の解決を処理するパッケージマネージャーです。
- **blivet** - もともと anaconda パッケージから pyanaconda.storage として分割されました。ストレージ管理に関連するすべてのアクティビティーを処理するために使用されます
- **pyanaconda** - キーボードとタイムゾーンの選択、ネットワーク設定、ユーザー作成、さらには多数のユーティリティーやシステム指向の機能など、Anaconda 固有の機能向けのユーザーインターフェイスやモジュールを含むパッケージです。
- **python-meh** - クラッシュ時に追加のシステム情報を収集、保存し、この情報を **libreport** ライブラリーに渡す例外ハンドラーを含みます。このライブラリー自体は ABRT プロジェクト に含まれます。

インストールプロセス中のデータのライフサイクルはシンプルです。Kickstart ファイルが提供されていれば、**pykickstart** モジュールがこれを処理してツリー構造としてメモリーにインポートします。Kickstart ファイルが提供されない場合は、代わりに空のツリー構造が作成されます。インストールが対話式の場合 (必須の Kickstart コマンドの一部しか使用されなかった場合)、この構造は対話式インターフェイスでユーザーが選択したもので更新されます。

必須の選択がすべて決定されるとインストールプロセスが開始され、ツリー構造に保存されている値を使用してインストールのパラメーターが決定します。これらの値は Kickstart ファイルとしても書き込まれ、インストールされるシステムの **/root/** ディレクトリーに保存されます。このため、この自動生成の Kickstart ファイルを再使用することで、このインストールを自動複製することが可能になります。

ツリーのような構造の要素は pykickstart パッケージによって定義されますが、それらの一部は **pyanaconda.kickstart** モジュールの修正版によってオーバーライドできます。この動作を決定する重要なルールは、選択データの保存場所がなく、インストールプロセスはデータ駆動型であり、最大限この処理に依存しているということです。このため、以下の点が確保されます。

- インストーラーの全機能が Kickstart でサポートされる **必要がある**
- インストールプロセスで、変更がターゲットシステムに書き込まれる単一の明確な時点がある。この時点の前では、永続的な変更 (例: ストレージのフォーマット) はなされません
- ユーザーインターフェイスでなされた手動での変更は作成される Kickstart ファイルに反映され、複製が可能

インストールが **データ駆動型** であることで、インストールおよび設定論理はツリー構造内のアイテムのメソッド内に位置することになります。必要に応じてインストールのランタイム環境を修正するためにアイテムはすべてセットアップされ (**setup** メソッド)、それを実行することで (**execute** メソッド) ターゲットシステム上で変更がなされます。これらの方法については、「Anaconda アドオンの作成」 .

5.3. ハブ & スポークモデル

Anaconda と他のオペレーティングシステムインストーラーの大きな違いは、前者は線状ではなく、*ハブおよびスポーク* モデルと呼ばれる点にあります。

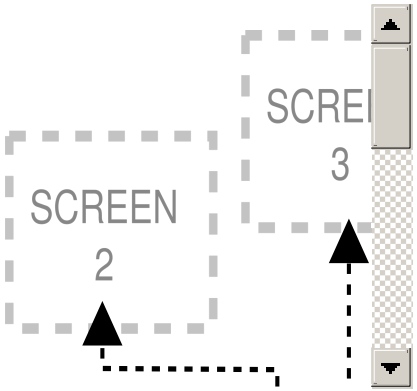
Anaconda がハブおよびスポークモデルとなっていることで、以下の利点があります。

- ユーザーは画面上でインストールを進める際に決まった順序に従う必要がない。

- 設定するオプションについて理解しているかどうかに関わらず全画面を開く必要がない。
- 特定のボタンがクリックされるまで、希望する値を設定してもマシンには実際には何もされないというトランザクションモードが機能する。
- 設定された値の概要を表示する方法がある。
- 並び替えや複雑な並び順の依存関係を解決せずに新たなスポークをハブに追加できるので、拡張性が高い。
- インストーラーのグラフィカルとテキストと両方のモードに使用可能。

以下の図ではインストーラーのレイアウトと、ハブおよび *スポーク* (screens) 間の関係を示しています。

図2 ハブ & スポークモデルの図



この図では、screens 2-13 は *通常のスポーク* とよばれ、screens 1 と 14 は *スタンドアロンスポーク* と呼ばれます。スタンドアロンスポークは、その後に続くスタンドアロンスポークもしくはハブの前 (またはその前のスタンドアロンスポークもしくはハブの後) に開く必要のある画面になります。例えば、インストール開始時の **ようこそ** 画面がこれに当たります。この画面では、残りのインストールに使用する言語を選択する必要があります。

注記

本セクションの残りの部分で言及する画面は、インストーラーのグラフィカルユーザーインターフェイス (GUI) の画面になります。

ハブおよびスポークモデルの中心点となるのが *ハブ* です。デフォルトでは以下の 2 種類のハブがあります。

- **インストールの概要** ハブ。インストール前に設定したオプションの概要を表示します。
- **設定および進捗状況** ハブ。**インストールの概要** で **インストールの開始** をクリックすると表示され、インストールプロセスの進捗状況が確認できるほか、追加オプションの設定ができます (root パスワードの設定やユーザーアカウントの作成など)。

スポークには以下の事前設定 *プロパティー* があり、これはハブで反映されます。以下のとおりです。

- ready: スポークが開けるかどうかを指定します。例えば、あるパッケージソースをインストーラーが設定している場合、そのスポークは準備ができておらずグレー表示され、設定が完了するまでアクセスできません。
- completed: スポークが完了 (必須の値がすべて設定済み) か未完了かをマークします。

- mandatory: インストールの続行にそのスポークを開いてユーザーが確認する **必要がある** かどうかを決定します。例えば、自動ディスクパーティション設定を使用する場合でも、**インストール先** スポークは開く必要があります。

- status: (ハブのスポーク名で表示される) 内で設定された値の短いサマリーを提供します。

ユーザーインターフェイスを使いやすくするために、スポークは *カテゴリー* 別にグループ化されます。例えば、**ローカリゼーション** カテゴリーにはキーボードレイアウトの選択や言語サポート、タイムゾーンの設定といったスポークが集められます。

各スポークには、メモリー内のツリー状構造の 1 つ以上のサブツリーの値を表示および変更できる UI コントロールが含まれています。「Anaconda のアーキテクチャー」 .として「Anaconda アドオンの作成」アドオンによって提供されるスポークにも同じことが当てはまります。

5.4. スレッドと連絡

既存パーティションのディスクスキャンやパッケージメタデータのダウンロードなど、インストール中に実行するアクションには時間のかかるものもあります。これらを待機すること避け、可能な範囲で応答できるようにするために、**Anaconda** ではこれらのアクションを個別のスレッドで実行します。

Gtk ツールキットはマルチスレッドからの要素変更をサポートしません。**Gtk** のメインイベントループは **Anaconda** プロセス自体のメインスレッドで実行され、GUI に関与するすべてのコード実行アクションもメインスレッドで実行される必要があります。これを行う唯一の対応方法は **GLib.idle_add** を使うことですが、これは必ずしも容易ではなく、推奨されません。この問題を軽減するために、いくつかのヘルパー関数とデコレーターが pyanaconda.ui.gui.utils モジュールで定義されています。

最も便利なものは **@gtk_action_wait** および **@gtk_action_nowait** デコレーターです。これらは、装飾された関数やメソッドを変更し、これらが呼び出されると自動的に Gtk のメインループにキュー登録され、戻り値が発信者に返されるか切断されるようにします。

マルチスレッドを使用する理由の 1 つは、他の画面で設定を実行中の間 (**インストール先** でパッケージメタデータをダウンロード中の場合など)、別の画面でユーザーが設定を進めることができるという利点です。実行中のスポークでの設定が終了したら、このスポークはその旨を連絡してブロックされないようにする必要があります。これは、**hubQ** というメッセージキューによって処理され、メインイベントループで定期的にチェックされています。スポークがアクセス可能になると、その旨とブロックされないべきであるとのメッセージをこのキューに送信します。

スポークがステータスまたは完了フラグをリフレッシュする必要がある場合にも、同様のことが適用されます。**設定および進捗状況** ハブには **progressQ** と呼ばれる別のキューがあり、インストールの進捗状況更新を送信するロールを果たします。

テキストベースのインターフェイスでは状況はより複雑になりますが、ここでもこれらのメカニズムは必要になります。テキストモードではメインループがなく、多くの場合にキーボード入力を待機することになります。

5.5. Anaconda アドオンの構造

Anaconda アドオンは Python パッケージで、これには **__init__.py** と他のソースディレクトリー (サブパッケージ) を格納しているディレクトリーが含まれています。Python は各パッケージ名のインポートを 1 回しか許可しないので、パッケージのトップレベルのディレクトリー名は一意のものである必要があります。同時に、アドオンは名前に関係なく読み込まれるため、任意の名前を指定できます。必須要件は、特定のディレクトリーに配置する必要があることです。

このため、アドオンで推奨される命名規則は、Java パッケージや D-Bus サービス名と同様のものになります。ドットの代わりにアンダースコア (_) で組織に予約されているドメイン名を持つアドオン名に接頭辞を付けます。これにより、ディレクトリー名が Python パッケージの有効な識別子となります。つまり、アド

オン名の前に自分の組織のリバースドメイン名を追加します (**com_example_hello_world**)。この規則は、Python パッケージおよびモジュール名の recommended naming scheme (推奨命名スキーム) に準拠しています。

重要

各ディレクトリー内に **__init__.py** ファイルを作成することを忘れないでください。このファイルがないディレクトリーは、有効な Python パッケージとはみなされません。

アドオンの作成時には、インストーラーでサポートされる機能は Kickstart でもサポートされる**必要がある**ことに注意してください。各インターフェイス (Kickstart、グラフィカルインターフェイス、グラフィカルインターフェイス、テキストインターフェイス) のサポートは個別のサブパッケージで指定する必要があります。このサブパッケージには、キックスタートの場合は **ks**、グラフィカルインターフェイスには **gui**、テキストベースのインターフェイスには **tui** という名前を指定する必要があります。**gui** と **tui** パッケージには、**spokes** サブパッケージが含まれる必要があります。[3]

これらのパッケージ内のモジュール名は任意のものにすることができます。**ks**/、**gui**/ および **tui**/ のディレクトリー内の Python モジュール名はどんなものでも構いません。

すべてのインターフェイス (Kickstart、GUI および TUI) をサポートするアドオンのディレクトリー構造のサンプルは以下のようになります。

例2 アドオン構造の例

```
com_example_hello_world
├── ks
│   └── __init__.py
├── gui
│   ├── __init__.py
│   └── spokes
│       └── __init__.py
└── tui
    ├── __init__.py
    └── spokes
        └── __init__.py
```

各パッケージには、API で定義される 1 つ以上のクラスから継承されるクラスを定義する任意の名前を持つモジュールが少なくとも 1 つ含まれる必要があります。これについては、「Anaconda アドオンの作成」.

すべてのアドオンにおける docstring 規則は、Python の PEP 8 および PEP 257 ガイドに従ってください。**Anaconda** の docstring の実際のコンテンツの形式には合意はありません。唯一の要件は、人間が判読できることです。唯一の要件は、ヒューマンリーダブルであることです。アドオンで自動生成ドキュメントを使用する場合は、これに使用するツールキットのガイドラインに docstring が準拠するようにしてください。

5.6. Anaconda アドオンの作成

以下のセクションでは、Hello World という名前のサンプルアドオンを作成およびテストするプロセスについて説明します。このサンプルアドオンは、全インターフェイス (Kickstart、GUI および TUI) をサポートします。このサンプルアドオンのソースは rhinstaller/hello-world-anaconda-addon リポジトリーから取得できます。このリポジトリーをクローンするか、ウェブインターフェイスでソースを開くことが推奨されます。

もう 1 つ確認するリポジトリーは、rhinstaller/anaconda です。ここにはインストーラーのソースコードがあり、本セクションでこのコードの一部が参照されます。

アドオン自体の開発を開始する前に、次の説明に従ってディレクトリー構造を作成することから始めます。「Anaconda アドオンの構造」.次に、「キックスタートのサポート」、キックスタートのサポートはすべてのアドオンで必須です。その後、オプションで続行できます「グラフィカルユーザーインターフェイス」と「テキスト形式のユーザーインターフェイス」必要に応じて。

5.6.1. キックスタートのサポート

Kickstart サポートはアドオンで最初に開発すべき点です。グラフィカルおよびテキストベースのインターフェイスのサポートなど他のパッケージはこれに依存します。まず、これまでに作成した **com_example_hello_world/ks/** ディレクトリーに移動し、**__init__.py** ファイルがあることを確認して、さらに **hello_world.py** という名前の Python スクリプトを追加します。

ビルトインの Kickstart コマンドとは異なり、アドオンは独自の *セクション* で使用されます。Kickstart ファイル内でのアドオンの使用はそれぞれ **%addon** ステートメントで開始され、**%end** で終了します。**%addon** 行にはアドオンの名前 (**%addon com_example_hello_world** など) と、オプションで引数一覧も含めます (アドオンがこれらに対応している場合)。

Kickstart ファイルでのアドオンの使用例は以下のようになります。

例3 Kickstart ファイルでのアドオンの使用

```
%addon ADDON_NAME [arguments]
first line
second line
...
%end
```

アドオンのキックスタートサポートのキークラスは **AddonData** と呼ばれます。このクラスは pyanaconda.addons で定義され、キックスタートファイルからのデータを解析および保存するためのオブジェクトを表します。

引数は、**AddonData** クラスから継承されたアドオンクラスのインスタンスに、リストとして渡されます。最初の行と最後の行の間にあるものはすべて、一度に一行ずつアドオンのクラスに渡されます。Hello World のアドオンサンプルをシンプルにするために、このブロック内のすべての行を単一行にまとめ、元の行を空白で区切ります。

サンプルのアドオンでは、**%addon** 行からの引数リストの処理のメソッドとセクション内の行を処理するメソッドのあるクラスを **AddonData** から継承する必要があります。**pyanaconda/addons.py** モジュールにはこれに使用可能な以下の 2 つのメソッドが含まれています。

- **handle_header**: **%addon** 行のリスト (およびエラー報告用の行番号) を取ります。
- **handle_line**: **%addon** と **%end** のステートメント間のコンテンツの単一行を取ります。

以下では、上記のメソッドを使用する Hello World アドオンの例を表示します。

例4 handle_header と handle_line の使用

```
from pyanaconda.addons import AddonData
from pykickstart.options import KSOptionParser
```

```
# export HelloWorldData class to prevent Anaconda's collect method from taking
# AddonData class instead of the HelloWorldData class
# :see: pyanaconda.kickstart.AnacondaKSHandler.__init__
__all__ = ["HelloWorldData"]

HELLO_FILE_PATH = "/root/hello_world_addon_output.txt"

class HelloWorldData(AddonData):
    """
    Class parsing and storing data for the Hello world addon.

    :see: pyanaconda.addons.AddonData

    """

def __init__(self, name):
    """
    :param name: name of the addon
    :type name: str

    """

    AddonData.__init__(self, name)
    self.text = ""
    self.reverse = False

def handle_header(self, lineno, args):
    """
    The handle_header method is called to parse additional arguments in the
    %addon section line.

    :param lineno: the current linenumber in the kickstart file
    :type lineno: int
    :param args: any additional arguments after %addon <name>
    :type args: list

    """

    op = KSOptionParser()
    op.add_option("--reverse", action="store_true", default=False,
    dest="reverse", help="Reverse the display of the addon text")
    (opts, extra) = op.parse_args(args=args, lineno=lineno)

    # Reject any additoinal arguments. Since AddonData.handle_header
    # rejects any arguments, we can use it to create an error message
    # and raise an exception.
    if extra:
        AddonData.handle_header(self, lineno, extra)

    # Store the result of the option parsing
    self.reverse = opts.reverse

def handle_line(self, line):
    """
    The handle_line method that is called with every line from this addon's
    %addon section of the kickstart file.
```

```
        :param line: a single line from the %addon section
        :type line: str

        """

        # simple example, we just append lines to the text attribute
        if self.text is "":
            self.text = line.strip()
        else:
            self.text += " " + line.strip()
```

この例では、必要なメソッドのインポートと **__all__** 変数の定義から始めます。これは、Anaconda の collect メソッドがアドオン固有の **HelloWorldData** ではなく **AddonData** クラスを取得しないようにするのに必要な変数を定義します。

この例は、親の **__init__** を呼び出して、**self.text** と **self.reverse** 属性を **False** に初期化する **__init__** メソッドで、**AddonData** から継承したクラス **HelloWorldData** の定義を示しています。

self.reverse 属性が **handle_header** メソッドに反映され、**self.text** が **handle_line** に反映されます。**handle_header** メソッドは **pykickstart** が提供する **KSOptionParser** のインスタンスを使用して、**%addon** 行で使用される追加オプションを解析し、**handle_line** は各行の最初と終了時に空白のコンテンツ行を取り除き、**self.text** に追加します。

上記のコードは、 Kickstart ファイルからデータを読み取るという、インストールプロセスにおけるデータライフサイクルの第 1 フェーズをカバーしています。次のステップは、そのデータを使用してインストールプロセスを進めることです。この場合、事前定義された方法が 2 つあります。

- **setup**: インストール処理の開始前に呼び出され、インストールランタイム環境の変更に使用されます。
- **execute**: 処理の最後に呼び出され、ターゲットシステムの変更に使用されます。

これら 2 つのメソッドを使用するには、新たなインポートと定数をモジュールに追加する必要があります。例を示します。

例5 setup および execute メソッドのインポート

```
import os.path

from pyanaconda.addons import AddonData
from pyanaconda.constants import ROOT_PATH

HELLO_FILE_PATH = "/root/hello_world_addon_output.txt"
```

setup および **execute** メソッドを含む Hello World アドオンの更新例を以下に示します。

例6 setup および execute メソッドの使用

```
def setup(self, storage, ksdata, instclass, payload):
    """
    The setup method that should make changes to the runtime environment
    according to the data stored in this object.
```

```
        :param storage: object storing storage-related information
        (disks, partitioning, bootloader, etc.)
        :type storage: blivet.Blivet instance
        :param ksdata: data parsed from the kickstart file and set in the
        installation process
        :type ksdata: pykickstart.base.BaseHandler instance
        :param instclass: distribution-specific information
        :type instclass: pyanaconda.installclass.BaseInstallClass
        :param payload: object managing packages and environment groups
        for the installation
        :type payload: any class inherited from the pyanaconda.packaging.Payload
        class
        """

        # no actions needed in this addon
        pass

    def execute(self, storage, ksdata, instclass, users, payload):
        """
        The execute method that should make changes to the installed system. It
        is called only once in the post-install setup phase.

        :see: setup
        :param users: information about created users
        :type users: pyanaconda.users.Users instance

        """

        hello_file_path = os.path.normpath(ROOT_PATH + HELLO_FILE_PATH)
        with open(hello_file_path, "w") as fobj:
            fobj.write("%s\n" % self.text)
```

上記の例では、**setup** メソッドは何もせず、Hello World アドオンはインストールランタイム環境に変化を加えません。**execute** メソッドは、保存したテキストを、ターゲットシステムのルート (/) ディレクトリーで作成したファイルに書き込みます。

上記の例で最も重要な情報は、それら 2 つのメソッドに渡される引数の量と意味です。これらは、例の中の docstring に記載されています。

データライフサイクルの最終フェーズ、およびキックスタートサポートを提供するモジュールで必要なコードの最後の部分は、インストール時に設定された値を含む新しいキックスタートファイルを、説明したようにインストールプロセスの最後に生成します。の「Anaconda のアーキテクチャー」.これは、インストールデータを格納するツリーのような構造構造で **__str__** メソッドを呼び出すことで実行されます。つまり、**AddonData** から継承されたクラスは、有効なキックスタート構文で保存されたデータを返す独自の **__str__** メソッドを定義する必要があります。この返されるデータは、**pykickstart** を使用して再度解析する必要があります。

Hello World の例では、**__str__** メソッドを以下の例のようになります。

例7 __str__ メソッドの定義

```
    def __str__(self):
        """
```

```
        What should end up in the resulting kickstart file, i.e. the %addon
        section containing string representation of the stored data.

        """

        addon_str = "%%addon %s" % self.name

        if self.reverse:
            addon_str += "--reverse"

        addon_str += "\n%s\n%%end" % self.text
        return addon_str
```

キックスタートサポートモジュールに必要なメソッドすべて (**handle_header**、**handle_line**、**setup**、**execute**、および **__str__**) が含まれると、有効な **Anaconda** アドオンになります。次のセクションに進み、グラフィカルおよびテキストベースのユーザーインターフェイスのサポートを追加するか、次のセクションに進むことができます。「Anaconda アドオンのデプロイおよびテスト」アドオンをテストします。

5.6.2. グラフィカルユーザーインターフェイス

本セクションでは、グラフィカルユーザーインターフェイス (GUI) のサポートをアドオンに追加する方法を説明します。これを開始する前に、前のセクションで説明した Kickstart のサポートがアドオンに含まれていることを確認してください。

注記

グラフィカルインターフェイスをサポートするアドオンの開発を開始する前に、**SpokeWindow** などの **Anaconda** 固有の Gtk ウィジェットを含む anaconda-widgets および anaconda-widgets-devel パッケージを必ずインストールしてください。

5.6.2.1. 基本的機能

アドオンの Kickstart サポートと同様に、GUI サポートでもアドオンのすべてのパートで、API で定義された特定のクラスから継承されたクラスの定義があるモジュールが少なくとも 1 つ含まれている必要があります。グラフィカルサポートの場合は、推奨される唯一のクラスは **NormalSpoke** で、これは **pyanaconda.ui.gui.spokes** で定義されます。クラス名の通り、**ノーマルスポーク** タイプのスクリーン用のクラスです。「ハブ & スポークモデル」 .

NormalSpoke から継承した新たなクラスを実装するには、API が必要とする以下のクラス属性を定義する必要があります。

- **builderObjects**: スポークの **.glade** ファイルからすべてのトップレベルオブジェクトを一覧表示します。これらのオブジェクトは、その子オブジェクトとともに (反復的に) スポークに公開されるべきものです。ただし、すべてをスポークに公開する場合 (非推奨) は、空のリストにします。

- **mainWidgetName**: メインウィンドウウィジェットの ID が含まれます。 [4] **.glade** ファイルで定義されている設定

- **uiFile**: **.glade** ファイルの名前が含まれます。

- **category**: スポークが属するカテゴリーのクラスが含まれます。

- **icon**: ハブ上のスポークに使用するアイコンの識別子が含まれます。

- **title**: ハブ上のスポークに使用するタイトルを定義します。

以下に必須の定義すべてを含むモジュール例を示します。

例8 Normalspoke クラスに必須の属性の定義

```
# will never be translated
_ = lambda x: x
N_ = lambda x: x

# the path to addons is in sys.path so we can import things from org_fedora_hello_world
from org_fedora_hello_world.gui.categories.hello_world import HelloWorldCategory
from pyanaconda.ui.gui.spokes import NormalSpoke

# export only the spoke, no helper functions, classes or constants
__all__ = ["HelloWorldSpoke"]

class HelloWorldSpoke(NormalSpoke):
    """
    Class for the Hello world spoke. This spoke will be in the Hello world
    category and thus on the Summary hub. It is a very simple example of
    a unit for the Anaconda's graphical user interface.

    :see: pyanaconda.ui.common.UIObject
    :see: pyanaconda.ui.common.Spoke
    :see: pyanaconda.ui.gui.GUIObject

    """

    ### class attributes defined by API ###

    # list all top-level objects from the .glade file that should be exposed
    # to the spoke or leave empty to extract everything
    builderObjects = ["helloWorldSpokeWindow", "buttonImage"]

    # the name of the main window widget
    mainWidgetName = "helloWorldSpokeWindow"

    # name of the .glade file in the same directory as this source
    uiFile = "hello_world.glade"

    # category this spoke belongs to
    category = HelloWorldCategory

    # spoke icon (will be displayed on the hub)
    # preferred are the -symbolic icons as these are used in Anaconda's spokes
    icon = "face-cool-symbolic"

    # title of the spoke (will be displayed on the hub)
    title = N_("_HELLO WORLD")
```

__all__ 属性はスポーククラスのエクスポートに使用され、これまでに説明した属性の定義を含む定義の最初の行がその後に続きます。これらの属性値は、**com_example_hello_world/gui/spokes/hello.glade** ファイルで定義されるウィジェットを参照します。

他に注目すべき 2 つの属性があります。1 番目の値は **category** で、**com_example_hello_world.gui.categories** モジュールの **HelloWorldCategory** クラスからインポートされます。の**HelloWorldCategory**class については後で説明しますが、ここでは、com_example_hello_world パッケージからインポートできるように、アドオンへのパスが sys.path にあることに注意してください。

2 つ目の notable 属性は **title** で、定義に 2 つのアンダースコアが含まれています。最初の 1 つは、翻訳の文字列のマークをマークする **N_** 関数名の一部ですが、変換されていないバージョンの文字列を返します (変換は後で行われます)。2 つ目のアンダースコアはタイトル自体の始まりをマークし、**Alt+H** キーボードのショートカットを使用してスポークに移動できるようにします。

クラス定義とクラス属性定義の後に続くのは、通常、クラスのインスタンスを初期化するコンストラクターです。**Anaconda** グラフィカルインターフェイスオブジェクトの場合、**__init__** メソッドと **initialize** メソッドという 2 つの方法があります。

これら 2 つの関数があるのは、GUI オブジェクトがメモリーで作成される時期とこれが完全に初期化される時期 (長時間かかる場合があります) が別になる可能性があるためです。そのため、**__init__** メソッドは親の **__init__** メソッドのみを呼び出してから、GUI 以外の属性を初期化します。一方、インストーラーのグラフィカルユーザーインターフェイスの初期化時に呼び出される **initialize** メソッドは、スポークの完全な初期化を完了する必要があります。

Hello World アドオンの例では、これら 2 つのメソッドは以下のように定義されます (**__init__** メソッドに渡される属性の数と記述に注意):

例9 __init__ および initialize メソッドの定義

```
def __init__(self, data, storage, payload, instclass):
    """
    :see: pyanaconda.ui.common.Spoke.__init__
    :param data: data object passed to every spoke to load/store data
    from/to it
    :type data: pykickstart.base.BaseHandler
    :param storage: object storing storage-related information
    (disks, partitioning, bootloader, etc.)
    :type storage: blivet.Blivet
    :param payload: object storing packaging-related information
    :type payload: pyanaconda.packaging.Payload
    :param instclass: distribution-specific information
    :type instclass: pyanaconda.installclass.BaseInstallClass

    """

    NormalSpoke.__init__(self, data, storage, payload, instclass)

def initialize(self):
    """
    The initialize method that is called after the instance is created.
    The difference between __init__ and this method is that this may take
    a long time and thus could be called in a separated thread.

    :see: pyanaconda.ui.common.UIObject.initialize

    """

    NormalSpoke.initialize(self)
    self._entry = self.builder.get_object("textEntry")
```

data パラメーターが **__init__** メソッドに渡されていることに留意してください。これは、すべてのデータが保存されている Kickstart ファイルのインメモリーのツリー構造を表します。ancestor の **__init__** メソッドのいずれかで、**self.data** 属性に格納されます。これにより、クラス内の他のすべてのメソッドで構造の読み取りおよび修正が可能になります。

なぜなら**HelloWorldData**クラスはすでに定義されています「キックスタートのサポート」、すでにサブツリーがあります**self.data**このアドオンのルート (クラスのインスタンス) は次のように利用できます。**self.data.addons.com_example_hello_world** .

親の **__init__** は他にも、スポークの **.glade** ファイルがある **GtkBuilder** のインスタンスを初期化し、これを **self.builder** として保存します。これは、kickstart ファイルの %addon セクションからのテキストの表示およびその修正に使用される **GtkTextEntry** の取得に **initialize** で使用されます。

__init__ および **initialize** メソッドは両方とも、スポークの作成時に重要となります。ただし、スポークの主要なロールは、ユーザーがこのスポークで表示、設定される値を変更または見直すことです。これを有効にするには、その他の 3 つの方法を使用できます。

- **refresh**: スポークをユーザーが表示する際に呼び出されます。このメソッドはスポークの状態を更新し (主に UI 要素)、**self.data** 構造に保存されている現行値を表示します。
- **apply**: スポークが残っている場合に呼び出され、UI 要素の値を **self.data** 構造に戻す際に使用されます。
- **execute**: スポークが残され、スポークの新しい状態に基づいてランタイム変更を実行する場合に呼び出されます。

これらの関数は、以下のように Hello World アドオンのサンプルに実装されます。

例10 更新、適用、および実行メソッドの定義

```
def refresh(self):
    """
    The refresh method that is called every time the spoke is displayed.
    It should update the UI elements according to the contents of
    self.data.

    :see: pyanaconda.ui.common.UIObject.refresh

    """

    self._entry.set_text(self.data.addons.org_fedora_hello_world.text)

def apply(self):
    """
    The apply method that is called when the spoke is left. It should
    update the contents of self.data with values set in the GUI elements.

    """

    self.data.addons.org_fedora_hello_world.text = self._entry.get_text()

def execute(self):
  """
  The excecute method that is called when the spoke is left. It is
```

```
    supposed to do all changes to the runtime environment according to
    the values set in the GUI elements.

    """

    # nothing to do here
    pass
```

スポークの状態の管理には、以下のメソッドを使用できます。

- **ready**: スポークを表示する準備ができているかどうかを判断します。値が false の場合は、スポークにアクセスできません (例: パッケージソース設定前の **Package Selection** スポークなど)。
- **completed**: スポークが完了しているかどうかを確認します。
- **mandatory**: スポークが必須かどうかを判断します (例: 自動パーティション設定を使用する場合でも、**インストール先** スポークは表示する必要があります)。

これらの属性はすべて、インストールプロセスの現在の状態に基づいて動的に決定する必要があります。以下は Hello World アドオンでのこれらの属性の実装例です。値によっては **HelloWorldData** クラスの **text** 属性で設定する必要があるものもあります。

例11 準備完了、完了、および必須メソッドの定義

```
@property
def ready(self):
    """
    The ready property that tells whether the spoke is ready (can be visited)
    or not. The spoke is made (in)sensitive based on the returned value.

    :rtype: bool

    """

    # this spoke is always ready
    return True

@property
def completed(self):
    """
    The completed property that tells whether all mandatory items on the
    spoke are set, or not. The spoke will be marked on the hub as completed
    or uncompleted acording to the returned value.

    :rtype: bool

    """

    return bool(self.data.addons.org_fedora_hello_world.text)

@property
def mandatory(self):
    """
    The mandatory property that tells whether the spoke is mandatory to be
```

```
        completed to continue in the installation process.

        :rtype: bool

        """

        # this is an optional spoke that is not mandatory to be completed
        return False
```

これらのプロパティーを定義したら、スポークはアクセスと完全性を制御できますが、その中で設定した値の概要を提供することはできません。これを確認するにはそのスポークを表示する必要がありますが、これは推奨されません。このため、**status** と呼ばれる別のプロパティーがあり、これには設定した値の簡潔な概要がテキスト 1 行で含まれています。これはハブ内のスポークタイトルの下で表示できます。

status プロパティーは、以下のように Hello World の例のアドオンで定義されます。

例12 status プロパティーの定義

```
@property
def status(self):
    """
    The status property that is a brief string describing the state of the
    spoke. It should describe whether all values are set and if possible
    also the values themselves. The returned value will appear on the hub
    below the spoke's title.

    :rtype: str

    """

    text = self.data.addons.org_fedora_hello_world.text

    # If --reverse was specified in the kickstart, reverse the text
    if self.data.addons.org_fedora_hello_world.reverse:
        text = text[::-1]

    if text:
        return _("Text set: %s") % text
    else:
        return _("Text not set")
```

本章にあるプロパティーをすべて定義したら、アドオンはグラフィカルユーザーインターフェイスと Kickstart に完全対応となります。ここでの例は非常に簡素化されたもので、管理機能はないことに注意してください。関数、GUI での対話式スポークの開発には Python Gtk プログラミングの知識が必要になります。

制限のうちで注意すべきものは、各スポークには **SpokeWindow** ウィジェットのインスタンスのメインウィンドウが必要になるという点です。このウィジェットは、**Anaconda** に固有の他のウィジェットとともに、anaconda-widgets パッケージにあります。GUI をサポートするアドオンの開発に必要なその他のファイル (Glade 定義など) は、anaconda-widgets-devel パッケージにあります。

クラフィカルインターフェイスサポートモジュールに必要なメソッドがすべて含まれたら、次のセクションに進んでテキストベースのユーザーインターフェイスのサポートを追加するか、「Anaconda アドオンのデプロイおよびテスト」アドオンをテストします。

5.6.2.2. 高度な機能

pyanaconda にはスポークやハブが使用するヘルパーやユーティリティー関数、コンストラクトが含まれており、これらはここまでのセクションで説明されていません。ほとんどのものは、**pyanaconda.ui.gui.utils** にあります。

サンプルの Hello World アドオン では **englightbox** で使用する englightbox コンテンツマネージャーの使用方法が説明されています。このマネージャーはウィンドウをライトボックスにおいて視認性を高めてフォーカスし、ユーザーが下層のウィンドウと対話しないようにします。この機能を示すために、サンプルアドオンには新しいダイアログウィンドウを開くボタンが含まれています。ダイアログ自体は、**pyanaconda.ui.gui.__init__** で定義される **GUIObject** クラスから継承される特別な **HelloWorldDialog** です。

dialog クラスは、**self.window** 属性を介してアクセス可能な内部 Gtk ダイアログを実行および破棄する **run** メソッドを定義します。この属性は、同じ意味の **mainWidgetName** クラス属性を使用して設定されます。そのため、以下の例のようにダイアログを定義するコードは非常にシンプルです。

例13 englightbox Dialog の定義

```
# every GUIObject gets ksdata in __init__
dialog = HelloWorldDialog(self.data)

# show dialog above the lightbox
with enlightbox(self.window, dialog.window):
    dialog.run()
```

上記のコードはダイアログのインスタンスを作成し、**enlightbox** コンテキストマネージャーを使用してライトボックス内でそのダイアログを実行します。コンテキストマネージャーは、スポークのウィンドウとダイアログのウィンドウのライトボックスをインスタンス化するためにこれらのウィンドウへの参照を必要とします。

Anaconda が提供するもう 1 つの便利な機能は、インストール中と **最初** の再起動後の両方に表示されるスポークを定義する機能です (「初回起動と初期設定」)。Anaconda および Initial Setup の両方でスポークを利用可能にするには、特別な **FirstbootSpokeMixIn** (より正確には mixin) を、**pyanaconda.ui.common** モジュールで定義されている最初の継承クラスとして継承する必要があります。

特定のスポークを Initial Setup で **のみ** 利用可能とするには、**FirstbootOnlySpokeMixIn** クラスを継承します。

pyanaconda パッケージによって提供されるさらに多くの高度な機能があります (**@gtk_action_wait**と**@gtk_action_nowait**デコレータ) ですが、このガイドの範囲外です。例についてはインストーラーのソース を参照してください。

5.6.3. テキスト形式のユーザーインターフェイス

Anaconda は Kickstart および GUI のほかに、テキストベースのインターフェイスもサポートしています。このインターフェイスは機能面では制限がありますが、システムによってはこれが唯一の対話式インストール方法となることがあります。テキストベースインターフェイスとグラフィカルインターフェイスの違い、および TUI の制限の詳細については、次を参照してください。「Anaconda の概要」 .

アドオンにテキストインターフェイスのサポートを追加するには、次の説明に従って、**tui** ディレクトリーの下に一連の新しいサブパッケージを作成します。「Anaconda アドオンの構造」.

インストーラーでのテキストモードのサポートは **simpleline** ユーティリティーをベースとしており、これは非常にシンプルなユーザーの対話のみを可能にするものです。これはカーソルの操作は**できず** (ラインプリンターのような動作になります)、色およびフォントのカスタマイズといった視覚的拡張機能もありません。

内部的には、**simpleline** ツールキットには、**App**、**UIScreen**、**Widget** の 3 つの主要クラスがあります。画面に表示 (プリント) する情報を格納しているユニットである Widget は、**App** クラスの単一インスタンスで切り替えられる UIScreens に配置されます。基本的な要素のほかには **hubs**、**spokes** および **dialogs** があり、これらはすべてグラフィカルインターフェイスと同様の各種ウィジェットを格納しています。

アドオンで最重要となるクラスは **NormalTUISpoke** と **pyanaconda.ui.tui.spokes** パッケージで定義される他の各種クラスです。これらのクラスはすべて **TUIObject** クラスに基づいています。それ自体は、前の章で説明した **GUIObject** クラスと同等です。各 TUI スポークは、**NormalTUISpoke** クラスを継承する Python クラスであり、API で定義される特別な引数とメソッドをオーバーライドします。テキストインターフェイスは GUI よりも簡単なため、引数は以下の 2 つのみになります。

- **title**: GUI の **title**引数の場合と同様に、スポークのタイトルを指定します。
- **category**: スポークのカテゴリーを文字列として判別します。カテゴリー名はどこにも表示されず、グループ化にのみ使用されます。

注記

カテゴリーは GUI とは異なる処理です。[5] 既存のカテゴリーを新しいスポークに割り当てることが推奨されます。新しいカテゴリーを作成するには、**Anaconda** にパッチを適用する必要があり、ほとんどメリットがありません。

また、各スポークは **__init__**、**initialize**、**refresh**、**refresh**、**apply**、**execute**、**input**、**prompt**、およびプロパティー (**ready**、**completed**、**mandatory**、および **status**) をオーバーライドすることが期待されています。これらはすべて、「グラフィカルユーザーインターフェイス」.

以下ではシンプルな TUI スポークを Hello World のサンプルアドオンに実装する例です。

例14 シンプルな TUI Spoke の定義

```
def __init__(self, app, data, storage, payload, instclass):
    """
    :see: pyanaconda.ui.tui.base.UIScreen
    :see: pyanaconda.ui.tui.base.App
    :param app: reference to application which is a main class for TUI
                screen handling, it is responsible for mainloop control
                and keeping track of the stack where all TUI screens are
                scheduled
    :type app: instance of pyanaconda.ui.tui.base.App
    :param data: data object passed to every spoke to load/store data
                 from/to it
    :type data: pykickstart.base.BaseHandler
    :param storage: object storing storage-related information
                    (disks, partitioning, bootloader, etc.)
    :type storage: blivet.Blivet
    :param payload: object storing packaging-related information
    :type payload: pyanaconda.packaging.Payload
```

```
        :param instclass: distribution-specific information
        :type instclass: pyanaconda.installclass.BaseInstallClass

        """

        NormalTUISpoke.__init__(self, app, data, storage, payload, instclass)
        self._entered_text = ""

    def initialize(self):
        """
        The initialize method that is called after the instance is created.
        The difference between __init__ and this method is that this may take
        a long time and thus could be called in a separated thread.

        :see: pyanaconda.ui.common.UIObject.initialize

        """

        NormalTUISpoke.initialize(self)

    def refresh(self, args=None):
        """
        The refresh method that is called every time the spoke is displayed.
        It should update the UI elements according to the contents of
        self.data.

        :see: pyanaconda.ui.common.UIObject.refresh
        :see: pyanaconda.ui.tui.base.UIScreen.refresh
        :param args: optional argument that may be used when the screen is
                     scheduled (passed to App.switch_screen* methods)
        :type args: anything
        :return: whether this screen requests input or not
        :rtype: bool

        """

        self._entered_text = self.data.addons.org_fedora_hello_world.text
        return True

    def apply(self):
        """
        The apply method that is called when the spoke is left. It should
        update the contents of self.data with values set in the spoke.

        """

        self.data.addons.org_fedora_hello_world.text = self._entered_text

    def execute(self):
        """
        The excecute method that is called when the spoke is left. It is
        supposed to do all changes to the runtime environment according to
        the values set in the spoke.

        """
```

```
        # nothing to do here
        pass

    def input(self, args, key):
        """
        The input method that is called by the main loop on user's input.

        :param args: optional argument that may be used when the screen is
                     scheduled (passed to App.switch_screen* methods)
        :type args: anything
        :param key: user's input
        :type key: unicode
        :return: if the input should not be handled here, return it, otherwise
                 return True or False if the input was processed succesfully or
                 not respectively
        :rtype: bool|unicode

        """

        if key:
            self._entered_text = key

        # no other actions scheduled, apply changes
        self.apply()

        # close the current screen (remove it from the stack)
        self.close()
        return True

    def prompt(self, args=None):
        """
        The prompt method that is called by the main loop to get the prompt
        for this screen.

        :param args: optional argument that can be passed to App.switch_screen*
                     methods
        :type args: anything
        :return: text that should be used in the prompt for the input
        :rtype: unicode|None

        """

        return _("Enter a new text or leave empty to use the old one: ")
```

ancestor の **__init__** のみを呼び出す場合は **__init__** メソッドを上書きする必要はありませんが、この例のコメントでは、一般的な方法でスポーククラスのコンストラクターへ渡された引数を記述します。

initialize メソッドはスポークの内部引数のデフォルト値を設定し、これは **refresh** メソッドで更新され、Kickstart データの更新に **apply** メソッドが使用します。この 2 つのメソッドが GUI のものと違う点は、**refresh** メソッドの戻り値のタイプ (None ではなく bool) と、それらが取る追加の **args** 引数のみです。返された値の意味はコメントで説明されています。このスポークにユーザー入力が必要かどうかに関係なく、アプリケーション (**App** クラスインスタンス) に指示します。追加の **args** 引数は、スポークに追加情報を渡す予定の場合に使用されます。

execute メソッドは、GUI の同等のメソッドと同じ目的で、この場合はメソッドは何もしません。

input と **prompt** メソッドはテキストインターフェイスに固有のものです。キックスタートまたは GUI には同等のものはありません。この 2 つのメソッドはユーザーの対話に使用されます。

prompt メソッドは、スポークのコンテンツがプリントされた後に表示されるプロンプトを返します。このプロンプトに文字列を入力すると、これが **input** メソッドに渡されて処理されます。**input** メソッドはこの文字列のタイプと値に応じてアクションを実行します。上記の例は**任意**の値を要求してから、それを内部属性 (**key**) として保存します。より複雑なアドオンでは通常、**c** を "continue" または **r** を "refresh" として解析する、数字を整数に変換する、新たな画面を表示するもしくはブール値を切り替えるなど、簡単ではないアクションを実行する必要があります。

input クラスの戻り値は、**INPUT_PROCESSED** か **INPUT_DISCARDED** の定数 (これらは両方とも **pyanaconda.constants_text** モジュールで定義) となるか、**もしくは** 入力文字列そのもの (この入力が別の画面で処理される場合) である必要があります。

グラフィカルモードとは対照的に、スポークを離れる際に **apply** メソッドは自動的に呼び出されません。**input** メソッドから明示的に呼び出す必要があります。同じことがスポークの画面閉鎖 (非表示) にも該当し、これは **close** メソッドの呼び出しで実行します。

別の画面を表示するには (例えば、別のスポークで入力された追加情報が必要な場合など)、別の **TUIObject** をインスタンス化し、**self.app.switch_screen*** の **App** メソッドの 1 つを呼び出します。

テキストベースのインターフェイスの制限により、TUI スポークは非常によく似た構造を持つ傾向があり、ユーザーがチェックまたはチェックを外して入力する必要があるチェックボックスまたはエントリーの一覧で設定されます。以前の段落では、メソッドが利用可能かつ提供されるデータの出力と処理に対処する TUI スポークを実装する方法が示されました。しかし、**pyanaconda.ui.tui.spokes** パッケージから **EditTUISpoke** クラスを使用し、これを実行する別の方法があります。このクラスを継承すると、設定するフィールドと属性を指定するだけで通常の TUI スポークを実装できます。以下の例ではこの方法を示しています。

例15 EditTUISpoke を使ったテキストインターフェイスのスポークの定義

```
class _EditData(object):
    """Auxiliary class for storing data from the example EditSpoke"""

    def __init__(self):
        """Trivial constructor just defining the fields that will store data"""

        self.checked = False
        self.shown_input = ""
        self.hidden_input = ""

class HelloWorldEditSpoke(EditTUISpoke):
    """Example class demonstrating usage of EditTUISpoke inheritance"""

    title = _("Hello World Edit")
    category = "localization"

    # simple RE used to specify we only accept a single word as a valid input
    _valid_input = re.compile(r'\w+')

    # special class attribute defining spoke's entries as:
    # Entry(TITLE, ATTRIBUTE, CHECKING_RE or TYPE, SHOW_FUNC or SHOW)
    # where:
    #   TITLE specifies descriptive title of the entry
    #   ATTRIBUTE specifies attribute of self.args that should be set to the
    #             value entered by the user (may contain dots, i.e. may specify
```

```
#           a deep attribute)
#   CHECKING_RE specifies compiled RE used for deciding about
#              accepting/rejecting user's input
#   TYPE may be one of EditTUISpoke.CHECK or EditTUISpoke.PASSWORD used
#       instead of CHECKING_RE for simple checkboxes or password entries,
#       respectively
#   SHOW_FUNC is a function taking self and self.args and returning True or
#            False indicating whether the entry should be shown or not
#   SHOW is a boolean value that may be used instead of the SHOW_FUNC
#
#   :see: pyanaconda.ui.tui.spokes.EditTUISpoke
edit_fields = [
    Entry("Simple checkbox", "checked", EditTUISpoke.CHECK, True),
    Entry("Always shown input", "shown_input", _valid_input, True),
    Entry("Conditioned input", "hidden_input", _valid_input,
    lambda self, args: bool(args.shown_input)),
    ]

def __init__(self, app, data, storage, payload, instclass):
    EditTUISpoke.__init__(self, app, data, storage, payload, instclass)

    # just populate the self.args attribute to have a store for data
    # typically self.data or a subtree of self.data is used as self.args
    self.args = _EditData()

@property
def completed(self):
    # completed if user entered something non-empty to the Conditioned input
    return bool(self.args.hidden_input)

@property
def status(self):
    return "Hidden input %s" % ("entered" if self.args.hidden_input
                                else "not entered")

def apply(self):
    # nothing needed here, values are set in the self.args tree
    pass
```

補助クラス **_EditData** は、ユーザーが入力した値を保存するデータコンテナーとして機能します。**HelloWorldEditSpoke** クラスはチェックボックス 1 つとエントリー 2 つの簡単なスポークを定義し、これらはすべて **Entry** クラスとしてインポートされる **EditTUISpokeEntry** のインスタンスになります。最初のインスタンスはスポークが表示されるたびに表示され、2 つ目のインスタンスは 1 つ目に空以外の値が含まれる場合にのみ表示されます。

EditTUISpoke クラスについての詳細は、上記の例にあるコメントを参照してください。

5.7. Anaconda アドオンのデプロイおよびテスト

新規のアドオンをテストするには、それをインストール環境に読み込む必要があります。アドオンはインストールランタイム環境の **/usr/share/anaconda/addons/** ディレクトリーから収集します。独自のアドオンをこのディレクトリーに追加するには、同じディレクトリー構造で **product.img** ファイルを作成し、ブートメディアに配置する必要があります。

既存のブートイメージの解凍、**product.img** ファイルの作成、およびイメージの再パッケージ化に関する具体的な手順については、次を参照してください。「ISO イメージを使った作業」 .

A. 更新履歴

改訂 2-11 Mon Aug 19 2019 Jaromír Hradílek
7.7 GA 公開用ドキュメントの準備

改訂 2-10 Wed May 29 2019 Sharon Moroney
7.7 ベータ公開用ドキュメントの準備

改訂 2-9 Tue Oct 30 2018 Vladimír Slávik
7.6 GA 公開用ドキュメントの準備

改訂 2-8 Tue Aug 21 2018 Vladimír Slávik
7.6 ベータ公開用ドキュメントの準備

改訂 2-7 Fri Apr 6 2018 Petr Bokoč
7.5 GA 公開用ドキュメントの準備

改訂 2-6 Fri Dec 15 2017 Petr Bokoč
7.5 ベータ公開用ドキュメントの準備

改訂 2-5 Tue Aug 1 2017 Petr Bokoč
7.4 GA 公開用ドキュメントの準備

改訂 2-4 Tue May 16 2017 Petr Bokoč
7.4 ベータ公開用ドキュメントの準備

改訂 2-3 Mon May 15 2017 Petr Bokoč
非同期の更新

改訂 2-2 Mon Nov 16 2015 Petr Bokoč
Red Hat Enterprise Linux 7.2 GA 向けリリース用のバージョン

改訂 2-1 Mon Jun 22 2015 Petr Bokoč
Anaconda アドオン開発セクションの編集終了
Anaconda 視覚要素カスタマイズセクションの追加

改訂 2-0 Fri Jun 19 2015 Petr Bokoč
Anaconda カスタマイズガイドにガイド名を変更
ISO イメージの展開および再パッケージ化についてのセクションを追加
ブートメニューのカスタマイズについてのセクションを追加
Anaconda アドオン開発セクションを編集

改訂 1-0 Wed Jan 15 2014 Vratislav Podzimek
公式ドキュメントの一部となる最初のバージョン

改訂 0-0 Fri Dec 28 2012 Vratislav Podzimek
Publican による初版作成

索引

シンボル

G

---

[1] Firstboot は、Gtk2 や pygtk2 など、メンテナーンス対象外となったツールに依存しています Firstboot はレガシーツールですが、これのために作成されたサードパーティーのモジュールがあるため、サポートが継続されています。

[2] Fedora では、アドオンはデフォルトで無効になっています。起動メニューで **inst.kdump_addon=on** オプションを使用して有効にできます。

[3] アドオンが新しいカテゴリーを定義する必要がある場合、gui パッケージには カテゴリー サブパッケージも含まれる場合がありますが、これはお勧めできません。

[4] Anaconda インストーラー用に作成されたカスタムウィジェットである SpokeWindow ウィジェットのインスタンス

[5] これは、今後も改善された方法 (GUI) に固定される可能性が高いです。