



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

---

インストーラーのカスタマイズと機能強化

Vratislav Podzimek

Petr Bokoč



## インストーラーのカスタマイズと機能強化

Vratislav Podzimek  
vpodzime@redhat.com

Petr Bokoč  
pbokoc@redhat.com

## 法律上の通知

Copyright © 2015 Red Hat, Inc. and others.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-sa/3.0/). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## 概要

Anaconda は、Red Hat Enterprise Linux、Fedora、およびこれらの派生製品で使用するインストーラーです。本ガイドでは、これのカスタマイズに必要な情報を提供しています。このインストーラーの基本的機能の拡張を図る開発者には、Anaconda のアーキテクチャー、アドオン API およびヘルパー関数の情報が提供され、カスタマイズアドオンの作成に便利な例もあります。本ガイドは、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ー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。

---

# 目次

<b>1. Anaconda のカスタマイズについて .....</b>	<b>2</b>
<b>2. ISO イメージを使った作業 .....</b>	<b>2</b>
2.1. Red Hat Enterprise Linux ブートイメージの抽出	2
2.2. product.img ファイルの作成	3
2.3. カスタムブートイメージの作成	5
<b>3. ブートメニューのカスタマイズ .....</b>	<b>6</b>
3.1. BIOS ファームウェアのシステム	6
3.2. UEFI ファームウェアのシステム	9
<b>4. グラフィカルユーザーインターフェースのブランド化と色調節 .....</b>	<b>11</b>
4.1. グラフィカル要素のカスタマイズ	11
4.2. 製品名のカスタマイズ	13
<b>5. インストーラーアドオンの開発 .....</b>	<b>14</b>
5.1. Anaconda とアドオンについて	14
5.2. Anaconda のアーキテクチャー	16
5.3. ハブ & スポークモデル	16
5.4. スレッドと連絡	18
5.5. Anaconda アドオンの構造	19
5.6. Anaconda アドオンの作成	20
5.7. Anaconda アドオンのデプロイとテスト	36
<b>A. 改訂履歴 .....</b>	<b>37</b>
<b>索引 .....</b>	<b>37</b>

## 1. Anaconda のカスタマイズについて

Red Hat Enterprise Linux および Fedora のインストールプログラムである **Anaconda** の最新バージョンでは、多くの点が改善されており、カスタマイズ機能の強化がその 1 つに挙げられます。基本的なインストーラー機能を拡張するためのアドオン作成や、グラフィカルユーザーインターフェースの外観を変更するオプションなどがあります。

本ガイドでは、以下のカスタマイズ方法について説明しています。

- ✳ ブートメニュー - 事前設定のオプション、カラースキームおよび背景
- ✳ グラフィカルインターフェースの外観 - ロゴ、背景、製品名
- ✳ インストーラーの機能 - アドオン。これは、グラフィカルおよびテキストのユーザーインターフェースに新たな Kickstart コマンドと画面を追加することでインストーラーの機能を強化します。

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 と Fedora 17 およびそれ以降にのみ適用されることに注意してください。



### 重要

本ガイドでの手順は、Red Hat Enterprise Linux 7 もしくは同様のシステム向けに書かれています。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 ブートイメージの抽出

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。

Red Hat Enterprise Linux 7 ブートメディアは、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](#) から取得できます。Red Hat Enterprise Linux 7 イメージのダウンロードには、アカウントに十分なエンタイトルメントが必要になります。

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。これらのイメージは、本ガイドの手順を使用することで修正できます。**KVM Guest Image** や **Supplementary DVD** といった他のダウンロード可能なイメージは修正できません。**(Server や ComputeNode といった)** イメージのバリエーションはここでは関係なく、どのバリエーションでも使用することができます。

Binary DVD および Boot ISO ダウンロードに関する詳細な手順については、[『Red Hat Enterprise Linux 7 インストールガイド』](#)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. マウントしたイメージの全コンテンツを作業ディレクトリーにコピーします。-p オプションを使っ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。[「ブートメニューのカスタマイズ」](#) または [「インストーラーアドオンの開発」](#) に進みます。変更を加えたら、[「カスタムブートイメージの作成」](#) の指示に従って新規の修正済み ISO イメージを作成します。

## 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)	<code>/usr/share/anaconda/pixmaps/</code>
インストール進捗画面のバナー	<code>/usr/share/anaconda/pixmaps/rnotes/en/</code>
GUI スタイルシート	<code>/usr/share/anaconda/anaconda-gtk.css</code>
Installclasses (製品名変更用)	<code>/run/install/product/pyanaconda/installclasses/</code>
Anaconda アドオン	<code>/usr/share/anaconda/addons/</code>

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. **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。

6. **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。[「カスタムブートイメージの作成」](#)に進んで変更を含む新たな起動可能な ISO イメージを作成することができます。**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

ブートメディアに **product.img** ファイルを追加する代わりに、このファイルを別の場所に配置して、ブートメニューで **inst.updates=** ブートオプションを使用してこれを読み込むこともできます。この場合、イメージファイルの名前は好きなものにすることができ、配置場所はどこでも構いません (USB フラッシュドライブ、ハードディスク、HTTP、FTP または NFS サーバーなど。ただし、インストールシステムからアクセスできること)。

**Anaconda** ブートオプションの詳細については、[『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. 加えた変更がすべて作業ディレクトリーに含まれていることを確認します。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** を使って新規 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 .
```

上記の例を以下で説明します。

- ※ オプションに **LABEL=** ディレクティブを使用する場合は同一ディスク上のファイルを読み込む場所が必要となります。この場合、**-V**、**-volset**、および **-A** の各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ようにします。ブートローダー設定 (BIOS では **isolinux/isolinux.cfg**、UEFI では **EFI/BOOT/grub.cfg**) で **inst.stage2=LABEL=disk\_label** の節を使用して同一ディスクからインストーラーの第 2 ステージを読み込む場合は、ディスクの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は、ディスクラベルの空白をすべて **\x20** で置き換えます。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は、ブートローダー設定では **RHEL\x207.1** を使ってこのラベルを参照します。

- ※ **-o** オプションの値 (**-o ../NEWISO.iso**) を新規イメージのファイル名で置き換えます。この例の値では、**NEWISO.iso** というファイルを現在のディレクトリーの上のディレクトリーに作成します。

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することができます。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については『[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については『[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。これは、インストールイメージからシステムを起動した後に表示されるメニューです。通常、このメニューを使うと、**Install Red Hat Enterprise Linux, Boot from local drive** または **Rescue an installed system** といったオプションから選択することができます。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することができます。

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、**ISOLINUX** は BIOS ファームウェアのシステムで使用され、**GRUB2** は UEFI ファームウェアのシステムで使用されます。この 2 つは、Red Hat が提供する AMD64 および Intel 64 システム向けの全イメージに存在します。

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。Kickstart ファイルは、インストール開始前にインストーラーに提供する必要があります。通常これは、既存のブートオプションを手動で編集し、**inst.ks=** ブートオプションを追加することで行います。メディア上のブートローダー設定ファイルを編集する際には、このオプションを事前設定エントリーのいずれかに追加することができます。

ブートローダーのカスタマイズを開始する前に、[手順1「ISO イメージの抽出」](#)の手順に従って、編集する ISO イメージを作業ディレクトリーに展開します。編集が完了したら、[手順3「ISO イメージの作成」](#)の手順に従って新規の起動可能な 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。

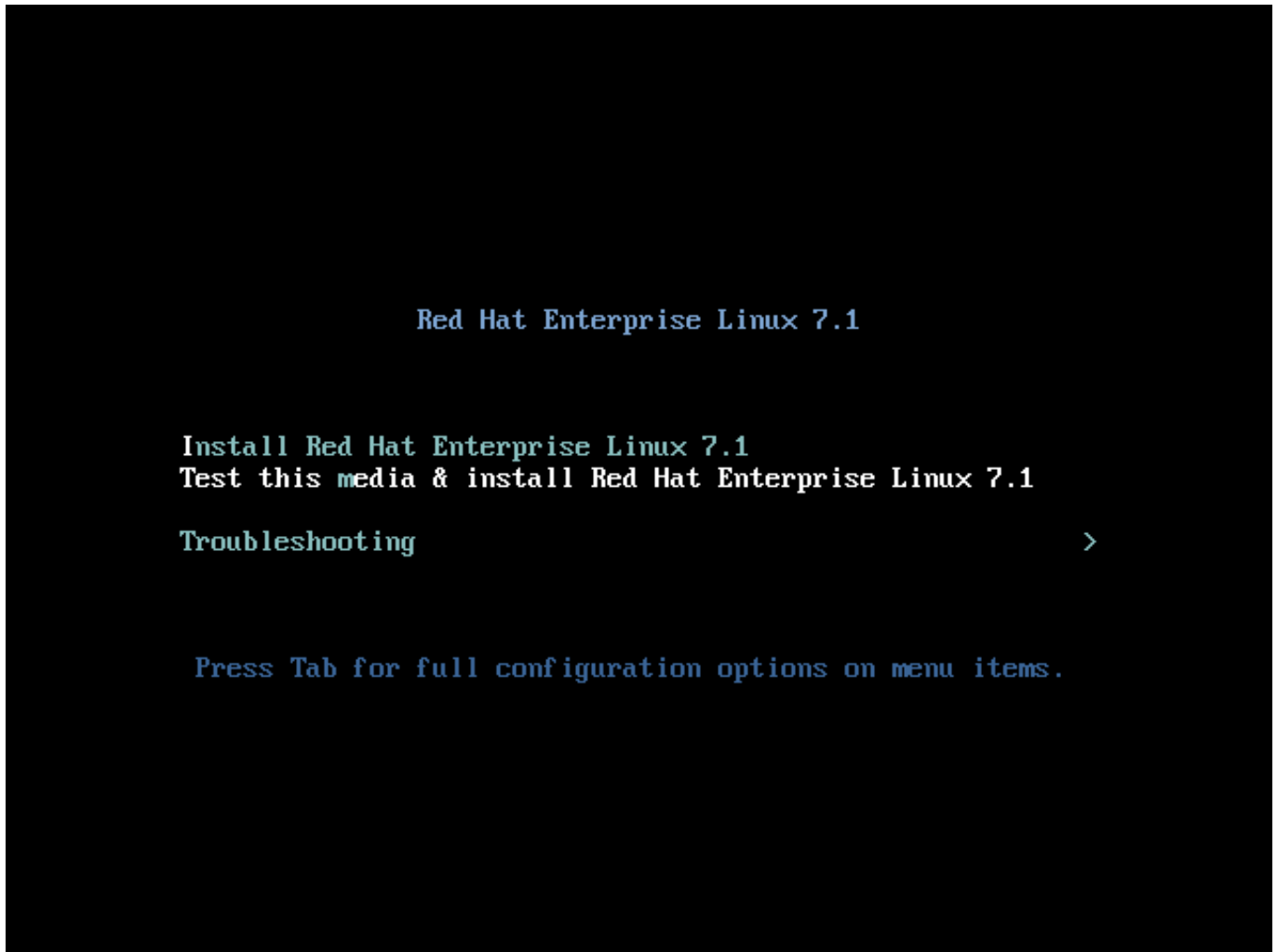


図1 ISOLINUX ブートメニュー

ブートメディア上の **isolinux/isolinux.cfg** 設定ファイルには、カラスキームとメニュー構造 (エン트리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 のデフォルトのメニューエン트리である **Test this media & Install Red Hat Enterprise Linux 7** が以下のブロックで定義されています。

```
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** - メニュー内でエントリが命名される方法を決定します。^ の文字はキーボードのショートカットを決定します (m のキー)。
- ✧ **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** - インストーラカーネルを読み込みます。通常は変更しないでください。

- ※ **append** - 追加のカーネルオプションが含まれます。**initrd=** と **inst.stage2** の各オプションは必須となり、この他のものを追加できます。

Anaconda 固有の使用可能なオプションは、『[Red Hat Enterprise Linux 7 インストールガイド](#)』に記載されています。注目すべきオプションには **inst.ks=** があり、このオプションを使用すると Kickstart 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ってこれを使用することができます。例えば、**kickstart.ks** という名前のファイルをイメージの root ディレクトリーに置く場合は、**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。これについては **dracut.cmdline(7)** man ページを参照してください。



### 重要

(上記の **inst.stage2=hd:LABEL=RHEL-7.1\x20x86\_64** オプションの場合のように) ディスクラベルを使って特定のドライブを参照する場合は、すべての空白を **\x20** で置き換えてください。

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 - submenu** ブロックの開始点と終了点を指定します。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** のステートメントの間にグループ化されています。2 目目のオプションの **menu exit** 行はサブメニューを終了してメインメニューに戻るものです。

- ※ **menu background** - メニューの背景です。(以下の **menu color** で) 単色とするか、PNG、JPEG または LSS16 形式のイメージとすることができます。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解像度の設定に寸法が一致するようにします。デフォルトの寸法は 640x480 です。
- ※ **menu color** - メニュー要素の色を指定します。完全な形式は以下のようになります。

```
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については、[Syslinux Wiki](#) を参照してください。

- ※ **menu help textfile** - 選択された場合にヘルプのテキストファイルを表示するメニューエントリーを作成します。

ISOLINUX 設定ファイルの完全なオプション一覧については、[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。

設定ファイル内では、Red Hat Enterprise Linux のデフォルトのメニューエントリー (**Test this media & install Red Hat Enterprise Linux 7.1**) は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、一重引用符または二重引用符 ( ' または " ) に入れます。 **--class** オプションを使うとメニューエントリーを異なるクラスにグループ化することができます。これは **GRUB2** テーマを使用して異なるスタイルを設定することが可能です。

### 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 固有の使用可能なオプションは、『[Red Hat Enterprise Linux 7 インストールガイド](#)』に記載されています。注目すべきオプションには **inst.ks=** があり、このオプションを使用すると Kickstart 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ってこれを使用することができます。例えば、**kickstart.ks** という名前のファイルをイメージの root ディレクトリーに置く場合は、**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。これについては **dracut.cmdline(7)** man ページを参照してください。



### 重要

(上記の **inst.stage2=hd:LABEL=RHEL-7.1\x20x86\_64** オプションの場合のように) ディスクラベルを使って特定のドライブを参照する場合は、すべての空白を **\x20** で置き換えてください。

- ※ **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 **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** - **submenu** ブロックの定義です。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、一重引用符または二重引用符 ( ' または " ) で囲みます。

**submenu** ブロックには上記で説明した **menuentry** 定義が 1 つ以上含まれ、ブロック全体は中括弧 ( { } ) で囲まれます。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。デフォルトを **3 番目** のエントリーとする場合は、**set default=2** を使用することになります。
- ※ **theme** - **GRUB2** テーマファイルを格納しているディレクトリーの場所です。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することができます。

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** に関する全般情報は、『[Red Hat Enterprise Linux 7 システム管理者のガイド](#)』も参照してください。

## 4. グラフィカルユーザーインターフェー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ース (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。

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、『[ISO イメージを使った作業](#)』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 イメージの背景は拡大縮小せず、常に正確な寸法で表示されます。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。このため、背景のイメージは背景色に「紛れる」ことになり、すべての解像度において背景はイメージを拡大縮小せずに機能します。

また **background-repeat** パラメーターを変更して背景をタイル表示にしたり、インストールしているすべてのシステムで同一の解像度を使用することが分かっている場合は、バー全体を占める背景イメージを使用することもできます。

**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 分ごとにバナー画像が画面下部で循環します。

上記で記載されているファイルはすべてカスタマイズが可能です。カスタマイズが終了したら、[「product.img ファイルの作成」](#) の指示に従ってカスタマイズ画像で **product.img** を作成し、[「カスタムブートイメージの作成」](#) にあるように変更を含めた新規の起動可能な 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** 属性を **20000** 以上に設定し、新規の installation class が最初に読み込まれるようにします。



### 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想しない動作をする場合があります。

カスタム installclass を作成したら、[「product.img ファイルの作成」](#)の指示に従ってカスタマイズを含む **product.img** ファイルを作成し、[「カスタムブートイメージの作成」](#)にあるように変更を含めた新規の起動可能な 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** は、Fedora、Red Hat Enterprise Linux、およびその他の派生製品で使用するオペレーティングシステムの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は以下の 4 つの主要ステップで構成されます。

- 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、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ース (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ースでは **VNC** を使ったり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、このインターフェー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は、Kickstart ファイルの使用です。これはプレーンテキストファイルで、インストールプロセスを進行するデータを格納できる shell のような構文を使用します。Kickstart ファイルを使用すると、インストールをすべてまたは部分的に自動化できます。すべての必須エラーを設定する特定のコマンドセットが必要になり、これが 1 つでも欠けているとインストールに對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、對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。他のインターフェー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ー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きました。これはユーザー作成などの一部の設定がこの時点で実行されるためです。これまでは **Firstboot** ツールをこの目的に使用し、新規インストールされた Red Hat Enterprise Linux システムを登録したり、**Kdump** を設定できるようになっていました。しかし **Firstboot** は、**Gtk2** や **pygtk2** などの既にメンテナンス対象外となったツールに依存しています<sup>[1]</sup>。このため、新規ツールである **Initial Setup** が開発されました。これは **Anaconda** からのコードを再利用します。このため、**Anaconda** 用に開発されたアドオンがスムーズに **Initial Setup** で再利用できます。これについては「[Anaconda アドオンの作成](#)」で詳述しています。

### 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ー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 の現行リリース (7.1 およびそれ以降) と Fedora<sup>[2]</sup> (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 kernel クラッシュダンプをサポートします。このアドオンは Kickstart で完全対応しており (`%addon com_redhat_kdump` コマンドとそのオプションを使用)、テキストベースおよびグラフィカルインターフェースで追加画面として完全に統合されます。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することができます。

### 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** を現行バージョンに開発することについての情報は、[Anaconda/NewInstaller Wiki page](#) 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](#) の章には Kickstart についての完全なドキュメントがあり、サポートされているコマンドとオプションの全一覧もあります。

- ※ 『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、グラフィカルおよびテキストユーザーインターフェースでのインストールプロセスを説明しています。
- ※ インストール後の設定に使用するツールについての情報は、[初期設定](#) の章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構成されています。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** - 元々は *anaconda* パッケージから *pyanaconda.storage* として分岐したものでした。ストレージ管理に関するアクティビティを処理します。
- ※ **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ー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 Project](#) の一部です。

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ー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ってインストールのパラメーターが決定されます。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。

ツリー構造の要素は *pykickstart* パッケージで定義されますが、**pyanaconda.kickstart** モジュールからの修正されたバージョンでいくつかは上書きすることができます。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ー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。これらのメソッドについては、[「Anaconda アドオンの作成」](#) で詳述しています。

## 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。

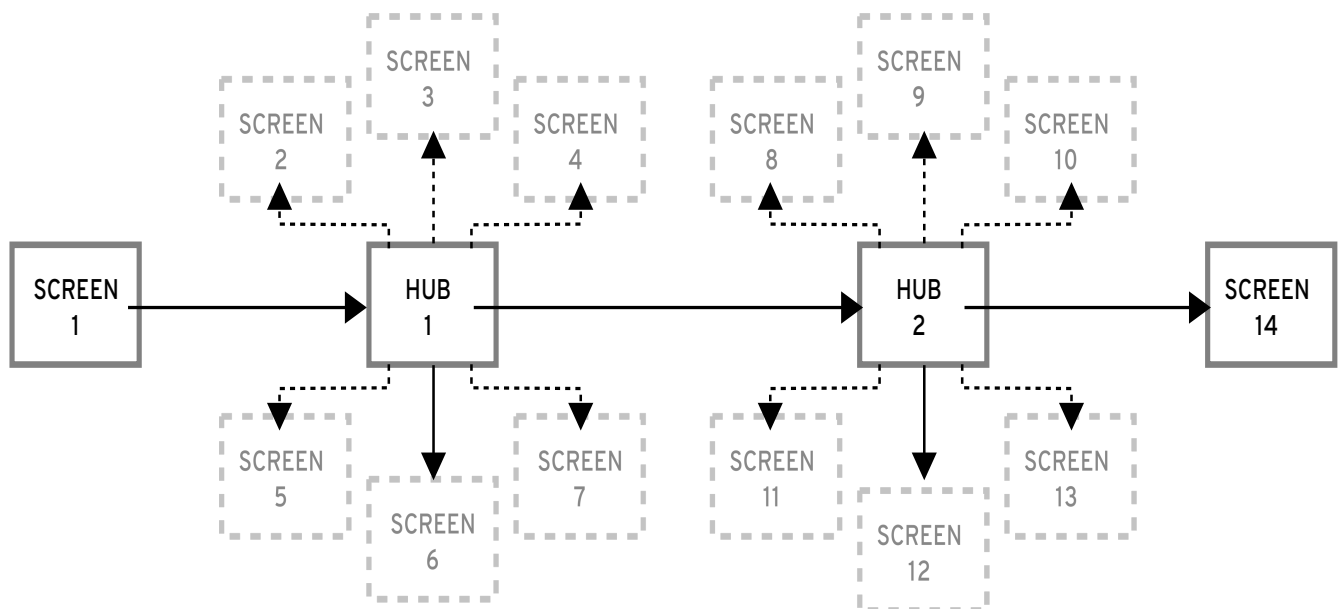


図2 ハブ & スポークモデルの図

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に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ース (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 プロパティがあり、これはハブで反映されます。

- ※ **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** - スポーク内で設定された値の概要を提供します (ハブのスポーク名の下に表示)。

ユーザーインターフェースを使いやすくするために、スポークは **カテゴリ** 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 UI 制御が含まれており、これは「[Anaconda のアーキテクチャー](#)」で説明したインメモリのツリー構造の 1 つ以上のサブツリーからの値を表示、修正できるものです。「[Anaconda アドオンの作成](#)」でも説明しますが、これはアドオンが提供するスポークにも該当します。

## 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。この問題を緩和するために、`pyanaconda.ui.gui.utils` モジュール内でいくつかのヘルパー関数とデコレーターが定義されています。

最も便利なのは `@gtk_action_wait` と `@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** と呼ばれるメッセージキューで処理され、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役割を果たします。

テキストベースのインターフェー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、アドオンはその名前に関係なく読み込まれるため、名前は任意のものにすることができます。唯一の要件は、特定のディレクトリー内に格納される必要があるということです。

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。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。この際、ドットではなくアンダースコア (`_`) を使用することで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。この規則を適用したディレクトリー名の例は、`com_example_hello_world` となります。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。GUI と TUI のサポートはオプションになります。各インターフェース (Kickstart、グラフィカルインターフェースおよびテキストインターフェース) は別個のサブパッケージにある必要があります、これらのサブパッケージは Kickstart では **ks**、グラフィカルインターフェースでは **gui**、テキストベースのインターフェースでは **tui** と命名する必要があります。**gui** と **tui** のパッケージは、**spokes** サブパッケージを格納している必要もあります。<sup>[3]</sup>

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ース (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 アドオン構造のサンプル

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

各パッケージには少なくとも 1 つのモジュールが含まれている必要があります。これらのモジュール名は任意のもので、API で定義される 1 つ以上のクラスから継承したクラスを定義します。これについては、「[Anaconda アドオンの作成](#)」で詳述します。

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 の実際のコンテンツの docstring に関する形式ではコンセンサスがありません。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、全インターフェース (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](#) リポジトリから取得できます。このリポジトリをクローンするか、ウェブインターフェー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、「[Anaconda アドオンの構造](#)」で説明したディレクトリ構造を作成してください。Kickstart サポートはすべてのアドオンで必須であることから、「[キックスターートのサポート](#)」をその次に行なってください。必要に応じて、「[グラフィカルユーザーインターフェース](#)」と「[テキスト形式のユーザーインターフェース](#)」を実行します。

### 5.6.1. キックスターートのサポート

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ー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
```

アドオン内での Kickstart サポートの重要なクラスは、**AddonData** と呼ばれるものです。このクラスは `pyanaconda.addons` 内で定義され、Kickstart ファイルからのデータを解析、保存するオブジェクトを表します。

引数は、**AddonData** クラスから継承し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](https://github.com/pyanaconda/pyanaconda/blob/master/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()

```

ここでは必要なメソッドのインポートで始まり、`__all__` 変数を定義します。これは **Anaconda** の収集したメソッドがアドオン固有の **HelloWorldData** ではなく **AddonData** クラスを取らないようにするために必要なものです。

次に、**AddonData** から継承した **HelloWorldData** クラスの定義が表示されています。これには、親の `__init__` を呼び出し、属性 `self.text` と `self.reverse` を **False** に初期化する `__init__` メソッドがあります。

`self.reverse` 属性は `handle_header` メソッド内に、`self.text` は `handle_line` 内に設定されます。`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ます。`handle_line` は各行の最初と最後にある空白のコンテンツ行を取り去り、これらを `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、そのデータを使ってインストールプロセスを進めることです。以下の 2 つの事前設定メソッドをこれに使用できます。

- ✦ **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"
```

Hello World アドオンに **setup** と **execute** メソッドが追加されると以下ようになります。

**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、保存済みのテキストをターゲットシステムの root (/) ディレクトリーに作成され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。

Kickstart サポートを提供するモジュールで必要となるコードの最後の部分とともに、データライフサイクルの最後のフェーズでは、新規 Kickstart ファイルが生成されます。これには「[Anaconda のアーキテクチャー](#)」で説明されているインストールプロセスの最後に設定される値が含まれています。これは、`__str__` メソッドを、インストールデータを保存しているツリー構造に反復して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が、有効な Kickstart 構文の保存済みデータを返す独自の `__str__` メソッドを定義する必要があることとなります。この返されるデータは、**pykickstart** を使用して解析可能である必要があります。

Hello World の例では、`__str__` メソッドは以下のようになります。

#### 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
```

Kickstart サポートモジュールに必要なメソッド (**handle\_header**、**handle\_line**、**setup**、**execute** および `__str__`) がすべて格納されたら、有効な **Anaconda** アドオンになります。以下のセクションに従って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ースのサポートを追加するか、「[Anaconda アドオンのデプロイとテスト](#)」に進んで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ース

本セクションでは、グラフィカルユーザーインターフェース (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ースのサポートのあるアドオンの開発を始める前に、*anaconda-widgets* と *anaconda-widgets-devel* のパッケージがインストールされていることを確認してください。これらには **SpokeWindow** のような **Anaconda** 固有の Gtk ウィジェットが含まれています。

#### 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。このクラス名が示すように、これは「[ハブ & スpoke モデル](#)」で説明されている **通常の spoke** の画面向けのクラスです。

**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** - spoke 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) spoke に公開されるべきものです。ただし、すべてを spoke に公開する場合 (非推奨) は、空のリストにします。
- ✦ **mainWidgetName** - **.glade** ファイルで定義されるメインウィンドウのウィジェットの ID が含まれます。[4]
- ✦ **uiFile** - **.glade** ファイルの名前を指定します。
- ✦ **category** - spoke が所属するカテゴリーのクラスを指定します。
- ✦ **icon** - spoke またはハブに使用されるアイコンの識別子を指定します。
- ✦ **title** - spoke またはハブに使用され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 つ目は `category` で、これには [com.example.hello\\_world.gui.categories](#) モジュールからの `HelloWorldCategory` クラスからインポートされた値があります。`HelloWorldCategory` クラスについては後述します。ここでは、アドオンへのパスは `sys.path` にあるので、`com_example_hello_world` パッケージからインポート可能になっていることに留意してください。

2 つ目は `title` です。これにはアンダースコアが 2 つ含まれています。1 つ目のアンダースコアは `N_` 関数名の一部で、これは変換する文字列をマークしますが、返されるのは変換されていない文字列のバージョンです (変換は後でなされます)。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** のグラフィカルインターフェースのオブジェクトの場合、`__init__` と `initialize` の 2 つのメソッドが新規インスタンスを初期化します。

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。このため、`__init__` メソッドは親の `__init__` メソッドを呼び出し、(例えば) GUI 以外の属性を初期化します。`initialize` メソッド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ースが初期化する際に呼び出され、スポークの完全な初期化を完了します。

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。親の **\_\_init\_\_** メソッドの1つでは **self.data** 属性に保存され、これによりクラス内の他のすべてのメソッドが構造を読み取り、修正できるようになります。

**HelloWorldData** クラスは「[キックスタートのサポート](#)」で既に定義済みであることから、**self.data** 内にアドオン向けのサブツリーが既にあり、その root (クラスのインスタンス) は **self.data.addons.com\_example\_hello\_world** として利用可能になっています。

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\_\_** と **initialize** のメソッドは両方ともスポークの作成時に重要なものです。ただし、スポークの主要な役割は、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。これを実行可能とするためには、以下の3つのメソッドが利用できます。

- ✦ **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** - ユーザーがスポークを離れ、UI 要素からの値を **self.data** 構造に保存し直す際に呼び出されます。
- ✦ **execute** - ユーザーがスポークを離れ、スポークの新規状態に基づいたランタイム変更を実行するために使用されます。

これらの関数は、Hello World アドオンに以下のように実装されます。

## 例10 refresh、apply および execute 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 execute 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** - スポークが完了したかどうかを判断します。
- ✧ **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行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 ready、completed および mandatory 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 according 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することができます。

Hello World アドオンでは **status** プロパティを以下のように定義します。

#### 例12 status プロパティの定義

```

@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ースと 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** 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。(Glade 定義のような) GUI サポートのあるアドオンの開発に必要な他のファイルは、*anaconda-widgets-devel* パッケージにあります。

グラフィカルインターフェースのサポートモジュールに必要なメソッドが備わったら、以下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ースのサポートを追加するか、[「Anaconda アドオンのテストとプロイ」とテスト](#)に進んで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 **Anaconda** で使用する **enlightbox** 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、サンプルのアドオンには新規ダイアログの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、**GUIObject** クラスを継承した特別な **HelloWorldDialog** で、これは [pyanaconda.ui.gui.\\_\\_init\\_\\_](#) で定義されます。

**dialog** クラスは、**self.window** でアクセス可能な内部の 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。これは **mainWidgetName** クラス属性を使用して同じ意味で設定されます。このため、このダイアログを定義するコードは以下のように非常にシンプルなものになります。

#### 例13 enlightbox ダイアログの定義

```

# 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** が提供するもうひとつの便利な機能は、インストール中と初回起動 ([「初回起動と初期設定」](#) の **Initial Setup** ユーティリティ) の両方で表示されるスポークを定義する機能です。**Anaconda** と **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。

(`@gtk_action_wait` や `@gtk_action_nowait` デコレーターのよう)に **pyanaconda** パッケージは他にも多くの高度な機能を提供しますが、それらは本ガイドの対象外となります。例については [インストーラーソース](#) 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ース

**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ースもサポートしています。このインターフェー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。テキストベースとグラフィカルインターフェースの違い、および TUI の制限に関する詳細は、[「Anaconda の概要」](#) を参照してください。

アドオンにテキストインターフェースのサポートを追加するには、[「Anaconda アドオンの構造」](#) にあるように **tui** ディレクトリ下にサブパッケージの新規セットを作成します。

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** および **Widget** の 3 つのメインクラスがあります。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 **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ー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** クラスをベースとしており、これ自体は前の章で説明した **GUIObject** クラスと同等のものです。各 TUI スポークは **NormalTUISpoke** クラスから継承した Python クラスで、API が定義する特別な引数やメソッドを上書きします。テキストインターフェースは GUI よりもシンプルなので、引数は以下の 2 つのみになります。

- ✦ **title** - GUI の **title** の場合と同様に、スポークのタイトルを指定します。
- ✦ **category** - 文字列としてスポークのカテゴリーを指定します。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 の場合とは異なる方法で処理されます。<sup>[5]</sup> 新規スポークには既存のカテゴリーを割り当てること推奨されます。新規のカテゴリーを作成すると **Anaconda** にパッチが必要になりますが、大きな利点はありません。

各スポークは以下のメソッドを上書きすることが期待されま

す。`__init__`、`initialize`、`refresh`、`refresh`、`apply`、`execute`、`input`、および `prompt` と、プロパティー (`ready`、`completed`、`mandatory`、および `status`)。これらについては「[グラフィカルユーザーインターフェース](#)」で説明しています。

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 スポークの定義

```
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.

: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 execute 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 successfully
    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: ")

```

親の `__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、この例でのコメントではスポーククラスのコンストラクターに渡す引数を分かりやすい方法で記述してあります。

**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。戻り値の意味についてはコメントで説明されています。アプリケーション (**App** クラスインスタンス) にこのスポークがユーザー入力を必要とするかどうかを指示します。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** のメソッドは、テキストインターフェース固有のもので、Kickstart や 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。上記の例では **いかなる** 値でもよく、これが内部属性 (**key**) として保存されます。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、スポークを離れる時に **apply** メソッドは自動的に呼び出されません。これは **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、**App** の **self.app.switch\_screen\*** メソッドの1つを呼び出します。

テキストベースインターフェースの制限により、TUI スポークは非常に似通った構造になる傾向があります。選択が必要なチェックもしくはエントリーリストをユーザーが設定するというものです。前の段落では、メソッドが利用可能なデータのプリントと処理を行う TUI スポークの実装方法について説明しました。ただし、これは [pyanaconda.ui.tui.spokes](#) パッケージの **EditTUISpoke** クラスを使用することでも実行できます。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ー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とテスト

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展開方法、**product.img** ファイルの作成方法、およびイメージの再パッケージ化方法についての詳細は、[「ISO イメージを使った作業」](#) を参照してください。



## A. 改訂履歴

<b>改訂 2-3.1</b>	<b>Thu Jul 20 2017</b>	<b>Kenzo Moriguchi</b>
翻訳ファイルを XML ソースバージョン 2-3 と同期		
<b>改訂 2-3</b>	<b>Mon May 15 2017</b>	<b>Petr Bokoč</b>
非同期のアップデート		
<b>改訂 2-2</b>	<b>Mon Nov 16 2015</b>	<b>Petr Bokoč</b>
Red Hat Enterprise Linux 7.2 GA 向けリリース用のバージョン		
<b>改訂 2-1</b>	<b>Mon Jun 22 2015</b>	<b>Petr Bokoč</b>
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		
<b>改訂 2-0</b>	<b>Fri Jun 19 2015</b>	<b>Petr Bokoč</b>
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		
<b>改訂 1-0</b>	<b>Wed Jan 15 2014</b>	<b>Vratislav Podzimek</b>
公式ドキュメントの一部となる最初のバージョン		
<b>改訂 0-0</b>	<b>Fri Dec 28 2012</b>	<b>Vratislav Podzimek</b>
Publican による初版作成		

## 索引

### シンボル

#### ブートメニュー

- BIOS システムのカスタマイズ, [BIOS ファームウェアのシステム](#)
- UEFI システムのカスタマイズ, [UEFI ファームウェアのシステム](#)

### G

#### grub2

- カスタム設定, [UEFI ファームウェアのシステム](#)

### I

#### ISO イメージ

- 抽出, [Red Hat Enterprise Linux ブートイメージの抽出](#)

#### ISO イメージの作成, [カスタムブートイメージの作成](#)

#### isolinux

- カスタム設定, [BIOS ファームウェアのシステム](#)

### M

#### MD5sum

- ISO イメージに埋め込み, [カスタムブートイメージの作成](#)

## P

**product.img**

- 作成, [product.img ファイルの作成](#)

---

- [1] 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ります。ブートメニューで **inst.kdump\_addon=on** オプションを使用するとこれを有効にできます。
- [3] アドオンが新規カテゴリーを定義する必要がある場合は、*gui* パッケージに *categories* サブパッケージを格納する場合もありますが、これは推奨されません。
- [4] SpokeWindow ウィジェットのインスタンスで、これは Anaconda 向けに作成されたカスタムウィジェットです。
- [5] これは将来変更される可能性が高く、より優れた (GUI) 方法になる可能性があります。