Red Hat Enterprise Linux 8

Anaconda のカスタマイズ

Red Hat Enterprise Linux 8 でのインストーラーの外観の変更およびカスタムアドオンの作成

Last Updated: 2021-09-12

Red Hat Enterprise Linux 8 Anaconda のカスタマイズ

Red Hat Enterprise Linux 8 でのインストーラーの外観の変更および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

Copyright © 2021 | You need to change the HOLDER entity in the en-US/Customizing_Anaconda.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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, the Red Hat 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 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、および派生製品が使用す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。

目次

多様性を受け入れるオープンソースの強化 . . . 3

RED HAT ドキュメントへのフィードバック (英語のみ) . . . 4

第1章 ANACONDA のカスタマイズの概要 . . . 5
1.1. ANACONDA のカスタマイズの概要 5
1.2. 事前カスタマイズタスクの実行 5
1.2.1. ISO イメージの使用 5
1.2.2. RH ブートイメージのダウンロード 5
1.2.3. Red Hat Enterprise Linux ブートイメージの抽出 6

第2章 ブートメニューのカスタマイズ . . . 7
2.1. ブートメニューのカスタマイズ 7
2.2. BIOS ファームウェアが搭載されたシステム 7
2.3. UEFI ファームウェアを使用しているシステム 10

第3章 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 . . . 13
3.1. グラフィカル要素のカスタマイズ 13
3.2. 製品名のカスタマイズ 15
3.3. デフォルト設定のカスタマイズ 15
3.3.1. デフォルトの設定ファイルの設定 15
3.3.2. 製品設定ファイルの設定 20
3.3.3. カスタム設定ファイルの設定 21

第4章 インストーラーアドオンの開発 . . . 22
4.1. ANACONDA およびアドオンの概要 22
4.2. ANACONDA のアーキテクチャー 23
4.3. ANACONDA ユーザーインターフェース 24
4.4. ANACONDA スレッド間の通信 25
4.5. ANACONDA モジュールおよび D-BUS ライブラリー 26
4.6. HELLO WORLD アドオンの例 26
4.7. ANACONDA アドオン構造 26
4.8. ANACONDA サービスと設定ファイル 27
4.9. GUI アドオンの基本機能 28
4.10. アドオングラフィカルユーザーインターフェース (GUI) のサポートの追加 29
4.11. アドオン GUI の高度な機能 34
4.12. TUI アドオンの基本機能 35
4.13. シンプルな TUI SPOKE の定義 36
4.14. NORMALTUISPOKE を使用したテキストインターフェーススポークの定義 39
4.15. ANACONDA アドオンのデプロイおよびテスト 41

第5章 ポストカスタマイズタスクの完了 . . . 43
5.1. プロダクトの IMG ファイルの作成 43
5.2. カスタムブートイメージの作成 45

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社 の CTO、Chris Wright のメッセージ を参照してください。

RED HAT ドキュメントへのフィードバック (英語のみ)

ご意見ご要望をお聞かせください。ドキュメントの改善点はございませんか。改善点を報告する場合は、以下のように行います。

- 特定の文章に簡単なコメントを記入する場合は、以下の手順を行います。
    1. ドキュメントの表示が **Multi-page HTML** 形式になっていて、ドキュメントの右上端に **Feedback** ボタンがあることを確認してください。
    2. マウスカーソルで、コメントを追加する部分を強調表示します。
    3. そのテキストの下に表示される **Add Feedback** ポップアップをクリックします。
    4. 表示される手順に従ってください。
- より詳細なフィードバックを行う場合は、Bugzilla のチケットを作成します。
    1. Bugzilla の Web サイトにアクセスします。
    2. Component で **Documentation** を選択します。
    3. **Description** フィールドに、ドキュメントの改善に関するご意見を記入してください。ドキュメントの該当部分へのリンクも記入してください。
    4. **Submit Bug** をクリックします。

第1章 ANACONDA のカスタマイズの概要

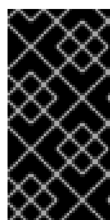
1.1. ANACONDA のカスタマイズの概要

Red Hat Enterprise Linux および Fedora インストールプログラム **Anaconda**最新バージョンに多くの改善が追加されました。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。アドオンを作成して、ベースインストーラー機能を拡張し、グラフィカルユーザーインターフェースの外観を変更できるようになりました。

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ースの外観: 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ースおよびテキストユーザーインターフェー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

また、本ガイドは、Red Hat Enterprise Linux 8 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 8 または同様のシステム用に記述されています。他のシステムでは、使用されるツールやアプリケーション (カスタム ISO イメージを作成する **genisoimage** など) は異なるため、手順の調整が必要になる場合があります。

1.2. 事前カスタマイズタスクの実行

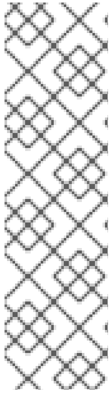
1.2.1. ISO イメージの使用

本セクションでは、以下の方法について説明します。

- Red Hat ISO を展開します。
- カスタマイズを含む新規ブートイメージを作成します。

1.2.2. RH ブートイメージのダウンロード

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。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 8 ブートメディアを取得できます。



注記

- アカウントには、Red Hat Enterprise Linux 8 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。
- **Binary DVD** または **Boot ISO** イメージをダウンロードし、任意のイメージバリアント (Server または ComputeNode) を使用することができます。
- KVM Guest Image や Supplementary DVD などの他の利用可能なダウンロードを使用してインストーラーをカスタマイズすることはできません。**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。

Binary DVD および Boot ISO のダウンロードの詳細は、『Red Hat Enterprise Linux 8 高度な RHEL インストールの実行』を参照してください。

1.2.3. Red Hat Enterprise Linux ブートイメージの抽出

以下の手順に従って、ブートイメージの内容を抽出します。

手順

1. **/mnt/iso** ディレクトリーが存在し、現在そこにマウントされていないことを確認してください。
2. 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 は、ダウンロードしたブートイメージのパスです。

3. ISO イメージの内容を配置する作業ディレクトリーを作成します。

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

4. 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
```

5. イメージをアンマウントします。

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

関連情報

- 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 8を参照してください。

第2章 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の概要と、そのカスタマイズ方法を説明します。

前提条件:

ブートイメージのダウンロードおよび抽出の詳細は、「Red Hat Enterprise Linux ブートイメージの抽出」 を参照してください。

ブートメニューのカスタマイズには、以下のハイレベルなタスクが必要です。

1. 前提条件を完了します。
2. ブートメニューをカスタマイズします。
3. カスタムブートイメージを作成します。

2.1. ブートメニューのカスタマイズ

ブートメニュー は、インストールイメージを使用してシステムを起動すると表示されるメニューです。通常、このメニューでは、**Install Red Hat Enterprise Linux**、**Boot from local drive**、または **Rescue an installed system** などのオプションを選択できます。Boot メニューをカスタマイズするには、以下を行います。

- デフォルトのオプションをカスタマイズします。
- オプションを追加します。
- 視覚的スタイル (色と背景) を変更します。

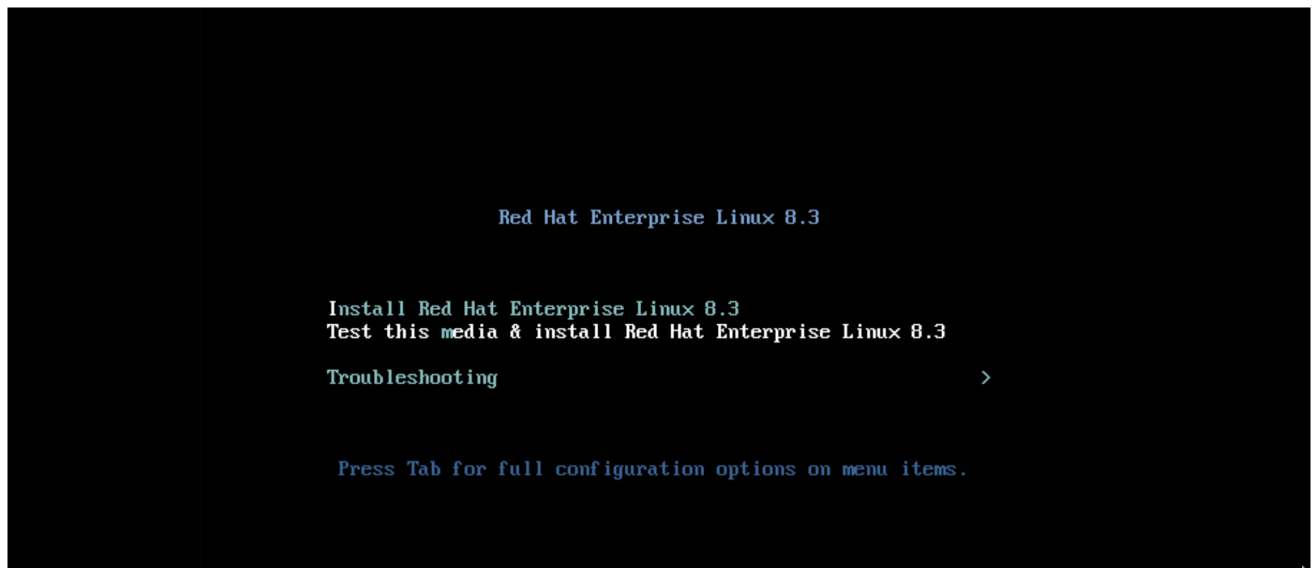
インストールメディアは、**ISOLINUX** および **GRUB2** ブートローダーで構成されます。**ISOLINUX** ブートローダーは BIOS ファームウェアのあるシステムで使用され、 **GRUB2** ブートローダーは、UEFI ファームウェアが搭載されているシステムで使用されます。AMD64 システムおよび Intel 64 システム用のすべての Red Hat イメージに、ブートローダーが両方存在します。

起動オプションのカスタマイズは、特にキックスタートで役に立ちます。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、**inst.ks=** ブートオプションを追加するために、既存の起動オプションを手動で編集して行います。メディア上のブートローダー設定ファイルを編集する場合は、このオプションを事前設定されたエントリーのいずれかに追加できます。

2.2. BIOS ファームウェアが搭載されたシステム

ISOLINUX ブートローダーは、BIOS ファームウェアのあるシステムで使用されます。

図2.1 ISOLINUX ブートメニュー



ブートメディアの **isolinux/isolinux.cfg** 設定ファイルには、色スキームとメニュー構造 (Enterly および submenu) を設定するため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である **Test this media & Install Red Hat Enterprise Linux 8** は、以下のブロックで定義されています。

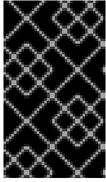
```
label check
  menu label Test this ^media & install Red Hat Enterprise Linux 8.4.0.
  menu default
  kernel vmlinuz
  append initrd=initrd.img inst.stage2=hd:LABEL=RHEL-8-4-0-BaseOS-x86_64 rd.live.check
quiet
```

詳細は以下のようになります。

- **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**: 一覧の最初のオプションではない場合でも、デフォルトの選択を行います。
- **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。
  適用可能なオプションの詳細は、 Anaconda 参照してください Red Hat Enterprise Linux 8 Performing a Standard RHEL Installation Guide.

  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。ブート ISO イメージにキックスタートファイルを配置し、inst.ks= オプションを使用してその場所を指定できます。たとえば、**kickstart.ks** 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-8-2-0-BaseOS-x86_64:/kickstart.ks** を使用します。

  また、**dracut.cmdline(8)** に一覧表示されている **dracut** オプションを使用することもできます。

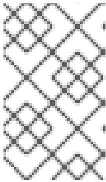


重要

上記の **inst.stage2=hd:LABEL=RHEL-8-2-0-BaseOS-x86_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に含まれていないその他の重要なオプションは、以下のとおりです。

- **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起動メニューが表示される時間を指定します。デフォルト値は **600** で、メニューが 60 秒間表示されます。この値を **0** に設定すると、timeout オプションが無効になります。



注記

タイムアウトを低い値 (例: **1**) に設定すると、ヘッドレスインストールを実行するときに便利です。これにより、デフォルトのタイムアウトが完了するのを回避するのに役立ちます。

- **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。2 つのオプションを持つ単純なサブメニュー (続行するオプション、メインメニューに戻るオプション) は次のようになります。

```
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-8-2-0-BaseOS-x86_64 rescue quiet

menu separator

label returntomain
  menu label Return to ^main menu
  menu exit

menu end
```

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似ていますが、**menu begin** および **menu end** ステートメント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**: メニュー背景は単色 (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
```

このコマンドで最も重要な部分は以下の通りです。

- element*:の色が適用される要素を決定します。*

- **フォアグラウンド** および **背景**: 実際の色を指定します。
  色は、16 進数形式の **#AARRGGBB** で記述されます。これにより、透明度も決まります。
- **00** は完全に透過的です。
- **ff** は完全に不透明です。
- **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。

関連情報

- 完全な一覧については、 ISOLINUX 設定ファイルのオプション。Syslinux Wiki を参照してください。

2.3. UEFI ファームウェアを使用しているシステム

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。

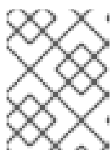
ブートメディアの **EFI/BOOT/grub.cfg** 設定ファイルには、事前設定されたメニューエントリーの一覧と、外観とブートメニュー機能を制御する他のディレクティブが含まれます。

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 8.4.0)**は、以下のブロックで定義されます。

```
menuentry 'Test this media & install Red Hat Enterprise Linux 8.4' --class fedora --class gnu-linux
--class gnu --class os {
    linuxefi /images/pxeboot/vmlinuz inst.stage2=hd:LABEL=RHEL-8-4-0-BaseOS-x86_64
rd.live.check quiet
    initrdefi /images/pxeboot/initrd.img
}
```

ここで、

- **menuentry**: エントリーのタイトルを定義します。これは引用符または二重引用符 (**'** または **"**) で指定されます。**--class オプションを使用して、メニューエントリーを別のクラスにグループ化し、** GRUB2 それらは以下のようになります。



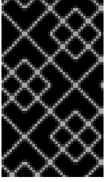
注記

上記の例で示すように、各メニューエントリー定義は中括弧 (**{}**) で囲む必要があります。

- **linuxefi**: ブートするカーネル (上記の例では**/images/pxeboot/vmlinuz**) と、その他の追加オプション (存在する場合) 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できます。適用可能なオプションの詳細は、 Anaconda、を参照してください。 Red Hat Enterprise Linux 8 Performing an advanced RHEL installation.

  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。ブート ISO イメージにキックスタートファイルを配置し、inst.ks= オプションを使用してその場所を指定できます。たとえば、**kickstart.ks** 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-8-2-0-BaseOS-x86_64:/kickstart.ks** を使用します。

また、**dracut.cmdline(8)** に一覧表示されている **dracut** オプションを使用することもできます。



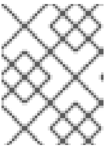
重要

上記の **inst.stage2=hd:LABEL=RHEL-8-2-0-BaseOS-x86_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されるその他のオプションは以下のとおりです。

- **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ブートメニューに表示される期間を指定します。デフォルト値は **60** です。つまり、メニューが 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

この設定によりデフォルトのブートエントリーがすぐに有効になるため、ヘッドレスインストールの実行時にタイムアウトを **0** に設定すると便利です。

- **submenu**: submenu ブロックでは、メインメニューにサブメニューを表示し、その下にエントリーをグループ化できます。デフォルト設定の **Troubleshooting** サブメニューには、既存のシステムをレスキューする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-8-2-0-BaseOS-x86_64
xdriver=vesa nomodeset quiet
    initrdefi /images/pxeboot/initrd.img
  }
  menuentry 'Submenu option 2' {
    linuxefi /images/vmlinuz inst.stage2=hd:LABEL=RHEL-8-2-0-BaseOS-x86_64 rescue quiet
    initrdefi /images/initrd.img
  }
}
```

- **set default**: デフォルトエントリーを決定します。エントリー番号は **0** から始まります。3 番目のエントリーをデフォルトのエントリーに設定する場合は、**set default=2** を使用します。

- テーマ: 含まれるディレクトリーを決定します。 **GRUB2** テーマファイル。テーマを使用して、ブートローダーの視覚的な要素 (背景、フォント、および特定の要素の色) をカスタマイズできます。

関連情報

- ブートメニューのカスタマイズの詳細は、GNU GRUB Manual 2.00 を参照してください。

- *一般情報： GRUB2。を参照してください。* Red Hat Enterprise Linux 8 Managing, monitoring and updating the kernel.

第3章 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

Anaconda ユーザーインターフェースのカスタマイズには、グラフィカル要素のカスタマイズや製品名のカスタマイズが含まれます。

本セクションでは、グラフィカル要素と製品名をカスタマイズする方法を説明します。

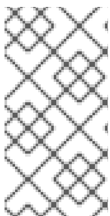
前提条件

1. ISO イメージをダウンロードして展開している。
2. 独自のブランディングマネージメントを作成している。

ブートイメージのダウンロードおよび抽出の詳細は、「Red Hat Enterprise Linux ブートイメージの抽出」を参照してください。

ユーザーインターフェースのカスタマイズには、以下のハイレベルなタスクが必要です。

1. 前提条件を完了します。
2. カスタムブランディング資料を作成する(グラフィカル要素をカスタマイズする予定の場合)
3. グラフィカル要素のカスタマイズ(カスタマイズする予定の場合)
4. 製品名のカスタマイズ(カスタマイズを計画している場合)
5. product.img ファイルを作成。
6. カスタムブートイメージを作成



注記

カスタムブランディング資料を作成するには、まずデフォルトのグラフィカル要素ファイルタイプと寸法を参照します。適切なカスタム資料を作成できます。デフォルトのグラフィカル要素の詳細は、「グラフィカル要素のカスタマイズ」セクションで説明されているサンプルファイルを参照してください。

3.1. グラフィカル要素のカスタマイズ

グラフィカル要素をカスタマイズするには、カスタマイズ可能な要素をカスタムブランドの資料に変更または置き換え、コンテナーファイルを更新します。

インストーラーのカスタマイズ可能なグラフィカル要素は、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ます。このディレクトリーには、以下のカスタマイズ可能なファイルが含まれます。

```
pixmaps
├─ anaconda-password-show-off.svg
├─ anaconda-password-show-on.svg
├─ right-arrow-icon.png
├─ sidebar-bg.png
├─ sidebar-logo.png
└─ topbar-bg.png
```

さらに、**/usr/share/anaconda/** ディレクトリーには、**anaconda-gtk.css** という名前の CSS スタイルシートが含まれており、メインの UI 要素のファイル名とパラメーター (サイドバーとトップバーのロゴおよび背景) を決定します。このファイルには、要件に従ってカスタマイズできる以下の内容が含まれます。

```
/* theme colors/images */

@define-color product_bg_color @redhat;

/* logo and sidebar classes */

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

/* Add a logo to the sidebar */

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

/* This is a placeholder to be filled by a product-specific logo. */

.product-logo {
   background-image: none;
   background-color: transparent;
}

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

CSS ファイルの最も重要な部分は、解決に基づいてスケーリング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、バックグラウンドには透過的な背景があり、スタイルシートは **@define-color** 行に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。

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、セクション 2.2 の「product.img ファイルの作成」の手順に従い、カスタムグラフィックスを備えた独自の product.img を作成します。その後、セクション 2.3 の「カスタムブートイメージの作成」を参照し、変更が含まれる新しいブート可能な ISO イメージを作成します。

3.2. 製品名のカスタマイズ

プロダクト名をカスタマイズするには、カスタム **.buildstamp** ファイルを作成する必要があります。これを行うには、以下の内容で新しいファイル **.buildstamp.py** を作成します。

```
[Main]
Product=My Distribution
Version=8.4
BugURL=https://bugzilla.redhat.com/
IsFinal=True
UUID=202007011344.x86_64
[Compose]
Lorax=28.14.49-1
```

My Distribution を、インストーラーで表示する名前に変更します。

カスタム .buildstamp ファイルを作成したら、「プロダクトの img ファイルの作成」 セクションの手順に従い、カスタマイズを含む新しい product.img ファイルを作成します。また、「カスタムブートイメージの作成」 セクションに従い、を含む変更が含まれる新しい起動可能な ISO ファイルを作成します。

3.3. デフォルト設定のカスタマイズ

独自の設定ファイルを作成し、これを使用してインストーラーの設定をカスタマイズすることができます。

3.3.1. デフォルトの設定ファイルの設定

Anaconda 設定ファイルは、**.ini** ファイル形式で記述できます。Anaconda 設定ファイルは、セクション、オプション、およびコメントで構成されています。各セクションは、**[section]** ヘッダー、**#** 文字で始まるコメント、および **オプション** を定義するためのキーで定義されます。生成される設定ファイルは、**configparser** 設定ファイルパーサーで処理されます。

/etc/anaconda/anaconda.conf にあるデフォルトの設定ファイルには、文書化されたセクションおよびオプションが含まれています。このファイルは、インストーラーの完全なデフォルト設定を提供します。製品設定ファイルの設定は、**/etc/anaconda/product.d/** から変更でき、カスタム設定ファイルは **/etc/anaconda/conf.d/** から変更できます。

以下の設定ファイルは、RHEL 8.4 のデフォルト設定を説明します。

```
# Anaconda configuration file for Red Hat Enterprise Linux.

[Product]
product_name = Red Hat Enterprise Linux

[Anaconda]
# Run Anaconda in the debugging mode.
debug = False

# Enable Anaconda addons.
addons_enabled = True

# List of enabled Anaconda DBus modules for RHEL.
kickstart_modules =
    org.fedoraproject.Anaconda.Modules.Timezone
```

```
    org.fedoraproject.Anaconda.Modules.Network
    org.fedoraproject.Anaconda.Modules.Localization
    org.fedoraproject.Anaconda.Modules.Security
    org.fedoraproject.Anaconda.Modules.Users
    org.fedoraproject.Anaconda.Modules.Payloads
    org.fedoraproject.Anaconda.Modules.Storage
    org.fedoraproject.Anaconda.Modules.Services

[Installation System]
# Should the installer show a warning about enabled SMT?
can_detect_enabled_smt = False

[Installation Target]
# Type of the installation target.
type = HARDWARE

# A path to the physical root of the target.
physical_root = /mnt/sysimage

# A path to the system root of the target.
system_root = /mnt/sysroot

# Should we install the network configuration?
can_configure_network = True

[Network]
# Network device to be activated on boot if none was configured so.
# Valid values:
#
#   NONE                   No device
#   DEFAULT_ROUTE_DEVICE   A default route device
#   FIRST_WIRED_WITH_LINK  The first wired device with link
#
default_on_boot = NONE

[Payload]
# Default package environment.
default_environment =

# List of ignored packages.
ignored_packages =

# Enable installation of latest updates.
enable_updates = True

# List of .treeinfo variant types to enable.
# Valid items:
#
#   addon
#   optional
#   variant
#
enabled_repositories_from_treeinfo = addon optional variant

# Enable installation from the closest mirror.
enable_closest_mirror = True
```

```
# Default installation source.
# Valid values:
#
#    CLOSEST_MIRROR  Use closest public repository mirror.
#    CDN             Use Content Delivery Network (CDN).
#
default_source = CLOSEST_MIRROR

# Enable ssl verification for all HTTP connection
verify_ssl = True

[Security]
# Enable SELinux usage in the installed system.
# Valid values:
#
#  -1  The value is not set.
#   0  SELinux is disabled.
#   1  SELinux is enabled.
#
selinux = -1

[Bootloader]
# Type of the bootloader.
# Supported values:
#
#   DEFAULT   Choose the type by platform.
#   EXTLINUX  Use extlinux as the bootloader.
#
type = DEFAULT

# Name of the EFI directory.
efi_dir = default

# Hide the GRUB menu.
menu_auto_hide = False

# Are non-iBFT iSCSI disks allowed?
nonibft_iscsi_boot = False

# Arguments preserved from the installation system.
preserved_arguments =
    cio_ignore rd.znet rd_ZNET zfcp.allow_lun_scan
    speakup_synth apic noapic apm ide noht acpi video
    pci nodmraid nompath nomodeset noiswmd fips selinux
    biosdevname ipv6.disable net.ifnames net.ifnames.prefix
    nosmt

[Storage]
# Enable dmraid usage during the installation.
dmraid = True

# Enable iBFT usage during the installation.
ibft = True

# Do you prefer creation of GPT disk labels?
```

```
gpt = False

# Tell multipathd to use user friendly names when naming devices during the installation.
multipath_friendly_names = True

# Do you want to allow imperfect devices (for example, degraded mdraid array devices)?
allow_imperfect_devices = False

# Default file system type. Use whatever Blivet uses by default.
file_system_type =

# Default partitioning.
# Specify a mount point and its attributes on each line.
#
# Valid attributes:
#
#   size <SIZE>     The size of the mount point.
#   min <MIN_SIZE> The size will grow from MIN_SIZE to MAX_SIZE.
#   max <MAX_SIZE> The max size is unlimited by default.
#   free <SIZE>     The required available space.
#
default_partitioning =
    /     (min 1 GiB, max 70 GiB)
    /home (min 500 MiB, free 50 GiB)
    swap

# Default partitioning scheme.
# Valid values:
#
#    PLAIN       Create standard partitions.
#    BTRFS       Use the Btrfs scheme.
#    LVM         Use the LVM scheme.
#    LVM_THINP  Use LVM Thin Provisioning.
#
default_scheme = LVM

# Default version of LUKS.
# Valid values:
#
#   luks1  Use version 1 by default.
#   luks2  Use version 2 by default.
#
luks_version = luks2

[Storage Constraints]
# Minimal size of the total memory.
min_ram = 320 MiB

# Minimal size of the available memory for LUKS2.
luks2_min_ram = 128 MiB

# Should we recommend to specify a swap partition?
swap_is_recommended = True

# Recommended minimal sizes of partitions.
# Specify a mount point and a size on each line.
```

```
min_partition_sizes =
    /      250 MiB
    /usr   250 MiB
    /tmp   50  MiB
    /var   384 MiB
    /home  100 MiB
    /boot  200 MiB

# Required minimal sizes of partitions.
# Specify a mount point and a size on each line.

# Allowed device types of the / partition if any.
# Valid values:
#
#  LVM        Allow LVM.
#  MD         Allow RAID.
#  PARTITION  Allow standard partitions.
#  BTRFS      Allow Btrfs.
#  DISK       Allow disks.
#  LVM_THINP  Allow LVM Thin Provisioning.
#
root_device_types =

# Mount points that must be on a linux file system.
# Specify a list of mount points.
must_be_on_linuxfs = / /var /tmp /usr /home /usr/share /usr/lib

# Paths that must be directories on the / file system.
# Specify a list of paths.
must_be_on_root = /bin /dev /sbin /etc /lib /root /mnt lost+found /proc

# Paths that must NOT be directories on the / file system.
# Specify a list of paths.
must_not_be_on_root =

[User Interface]
# The path to a custom stylesheet.
custom_stylesheet =

# The path to a directory with help files.
help_directory =

# A list of spokes to hide in UI.
# FIXME: Use other identification then names of the spokes.
hidden_spokes =

[License]
# A path to EULA (if any)
#
# If the given distribution has an EULA & feels the need to
# tell the user about it fill in this variable by a path
# pointing to a file with the EULA on the installed system.
#
# This is currently used just to show the path to the file to
# the user at the end of the installation.
eula =
```

3.3.2. 製品設定ファイルの設定

製品設定ファイルには、製品を識別するセクションが1 つまたは2 つあります。**[Product]** セクションは、プロダクトの製品名を指定します。**[Base Product]** セクションは、ベース製品 (存在する場合) の製品名を指定します。たとえば、Red Hat Enterprise Linux は Red Hat Virtualization のベース製品です。

インストーラーは、指定された製品の設定ファイルを読み込む前に、ベース製品の設定ファイルを読み込みます。たとえば、まずRed Hat Enterprise Linux の設定を読み込み、次に Red Hat Virtualization の設定を読み込みます。

Red Hat Virtualization の製品設定ファイルの例を参照してください。

```
[Product]
product_name = Red Hat Virtualization

[Base Product]
product_name = Red Hat Enterprise Linux

[Storage]
default_scheme = LVM_THINP
default_partitioning = VIRTUALIZATION

[Storage Constraints]
root_device_types = LVM_THINP
must_not_be_on_root = /var
req_partition_sizes =
 /var   10 GiB
 /boot  1  GiB

[User Interface]
help_directory = /usr/share/anaconda/help/rhv
```

Red Hat Enterprise Linux の製品設定ファイルの例を参照してください。

```
# Anaconda configuration file for Red Hat Enterprise Linux.

[Product]
product_name = Red Hat Enterprise Linux

[Anaconda]
# List of enabled Anaconda DBus modules for RHEL.
kickstart_modules =
    org.fedoraproject.Anaconda.Modules.Timezone
    org.fedoraproject.Anaconda.Modules.Network
    org.fedoraproject.Anaconda.Modules.Localization
    org.fedoraproject.Anaconda.Modules.Security
    org.fedoraproject.Anaconda.Modules.Users
    org.fedoraproject.Anaconda.Modules.Payloads
    org.fedoraproject.Anaconda.Modules.Storage
    org.fedoraproject.Anaconda.Modules.Services
    org.fedoraproject.Anaconda.Modules.Subscription

[Installation System]
# Show a warning if SMT is enabled.
can_detect_enabled_smt = True
```

```
[Network]
default_on_boot = DEFAULT_ROUTE_DEVICE

[Payload]
ignored_packages =
    ntfsprogs
    btrfs-progs
    dmraid

enable_updates = False
enable_closest_mirror = False
default_source = CDN

[Bootloader]
efi_dir = redhat

[Storage]
file_system_type = xfs

[User Interface]
help_directory = /usr/share/anaconda/help/rhel

[License]
eula = /usr/share/redhat-release/EULA
```

プロダクトのインストーラー設定をカスタマイズするには、製品設定ファイルを作成する必要があります。上記の例のような内容で、**my-distribution.conf** という名前の新しいファイルを作成します。**[Product]** セクションの product_name を、製品の名前 (例: My Distribution) に変更します。製品名は、**.buildstamp** ファイルで使用される名前と同じでなければなりません。

カスタム .buildstamp ファイルを作成したら、「プロダクトの img ファイルの作成」 セクションの手順に従い、カスタマイズを含む **new product.img** ファイルを作成します。また、「カスタムブートイメージの作成」 セクションに従い、を含む変更が含まれる新しい起動可能な ISO ファイルを作成します。

3.3.3. カスタム設定ファイルの設定

インストーラー設定を製品名とは別にカスタマイズするには、カスタム設定ファイルを作成する必要があります。これを行うには、??? の例のような内容で **100-my-configuration.conf** という名前の新規ファイルを作成し、**[Product]** および **[Base Product]** セクションを省略します。

カスタム .buildstamp ファイルを作成したら、「プロダクトの img ファイルの作成」 セクションの手順に従い、カスタマイズを含む **new product.img** ファイルを作成します。また、「カスタムブートイメージの作成」 セクションに従い、を含む変更が含まれる新しい起動可能な ISO ファイルを作成します。

第4章 インストーラーアドオンの開発

本セクションでは、Anaconda とそのアーキテクチャーの詳細と、独自のアドオンの開発方法を説明します。Anaconda およびそのアーキテクチャーの詳細は、Anaconda バックエンドおよびアドオンが機能するさまざまなプラグインポイントを理解するのに役立ちます。また、アドオンの開発に対応するのに役立ちます。

4.1. ANACONDA およびアドオンの概要

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、**Gtk** ウィジェット (C で記述)、**systemd** ユニット、**dracut** ライブラリーなどの一部の追加ファイルが含まれる、一連の Python モジュールおよびスクリプトです。同時に、これらは、作成される (ターゲット) システムのパラメーターを設定してから、このシステムをマシンにセットアップできるツールを形成します。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。

1. 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)
2. パッケージおよびデータのインストール
3. ブートローダーのインストールおよび設定
4. 新規インストールシステムの設定

Anaconda を使用すると、以下の 3 つの方法で Fedora、Red Hat Enterprise Linux、および派生製品をインストールできます。

グラフィカルユーザーインターフェース (GUI) の使用:

これは、最も一般的なインストール方法です。インターフェースを使用すると、インストールを開始する前に、設定がほとんどまたは何も必要のない状態でシステムを対話的にインストールできます。この方法は、複雑なパーティションレイアウトの設定など、一般的なすべてのユースケースを取り上げます。

グラフィカルインターフェースは **VNC** を介したリモートアクセスをサポートします。これにより、グラフィックカードやモニターが接続されていないシステムでも GUI を使用できます。

テキストユーザーインターフェース (TUI) の使用:

TUI はモノクロラインプリンターと同様に動作します。これにより、カーソルの移動、色、その他の高度な機能に対応しないシリアルコンソールで作業できます。テキストモードは限定されており、ネットワーク設定、言語オプション、インストール (0パッケージ) ソースなど、最も一般的なオプションのみをカスタマイズできます。このインターフェースでは、手動パーティション設定などの高度な機能は利用できません。

キックスタートファイルの使用:

キックスタートファイルは、シェルのような構文を持つプレーンテキストファイルで、インストールプロセスを実行するためのデータを含めることができます。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、すべての必要なエリアを構成する一連のコマンドが必要になります。1 つ以上のコマンドが見つからない場合は、インストールに対話が必要です。

インストーラー自体の自動化以外に、キックスタートファイルには、インストールプロセス中に特定の時点で実行するカスタムスクリプトを含めることができます。

4.2. ANACONDA のアーキテクチャー

Anaconda Python モジュールおよびスクリプトのセットです。また、外部パッケージおよびライブラリーも使用します。このツールセットの主要コンポーネントには、以下のパッケージが含まれます。

- **pykickstart**: キックスタートファイルを解析し、検証します。また、インストールを動作させる値を保存するデータ構造も提供します。
- **yum**: パッケージをインストールして依存関係を解決するパッケージマネージャー
- **blivet**: 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します。
- **pyanaconda**: 使用するユーザーインターフェースおよびモジュールが含まれます。Anaconda キーボードやタイムゾーンの選択、ネットワーク設定、ユーザー作成など。システム指向の機能を実行するためのさまざまなユーティリティーも提供します。
- **python-meh**: クラッシュ発生時に追加のシステム情報を収集して保存する例外ハンドラーを含みます。また、この情報を **libreport** ライブラリーに渡します。それ自体は ABRT プロジェクト に含まれます。
- **dasbus**: **D-Bus** ライブラリーと、anaconda のモジュールと外部コンポーネントとの通信を有効にします。
- **python-simpleline**: Aaconda テキストモードでユーザーの対話を管理するテキスト UI フレームワークライブラリー
- **gtk**: GUI を作成および管理する Gnome ツールキットライブラリー

前述したパッケージへの分割以外に、**Anaconda** は内部的にユーザーインターフェースと、別個のプロセスとして実行され、**D-Bus** ライブラリーを使用して通信するモジュールセットに分けられています。これらのモジュールは以下のとおりです。

- **Boss**: 内部モジュール検出、ライフサイクル、およびコーディネーションを管理します。
- **Localization**: ロケールを管理します。
- **Network**: ネットワークを処理します。
- **Payloads**: **rpm**、**ostree**、**tar** などの異なる形式でインストール用のデータを処理します。ペイロードはインストール用のデータのソースを管理します。ソースは CD-ROM、HDD、NFS、URL などの形式によって異なる可能性があります。
- **Security**: セキュリティー関連機能を管理します。
- **Service**: サービスを処理します。
- **Storage**: **blivet**を使用してストレージを管理します。
- **Subscription**: **subscription-manager** ツールと Insights を処理します。
- **Timezone**: 時間、日付、ゾーン、および時刻の同期を処理します。
- **Users**: ユーザーおよびグループを作成します。

各モジュールは、処理するキックスタート部分を宣言し、キックスタートから設定を適用してインストール環境とインストール済みシステムに適用します。

*Anaconda の Python コード部分 (**pyanaconda**) は、ユーザーインターフェースを所有する「メイン」プロセスとして起動します。指定のキックスタートデータはすべて **pykickstart** モジュールと **Boss** モジュールを使用して解析され、その他のすべてのモジュールを検出し、起動します。その後、メインプロセスは宣言された機能に応じてキックスタートデータをモジュールに送信します。モジュールはデータを処理し、インストール環境に設定を適用し、必要な選択がすべて行われているかどうかを UI が検証します。インストールされていない場合は、対話式インストールモードでデータを指定する必要があります。必要な選択がすべて行われると、インストールを開始できます。モジュールは、インストール済みシステムにデータを書き込むことができます。*

4.3. ANACONDA ユーザーインターフェース

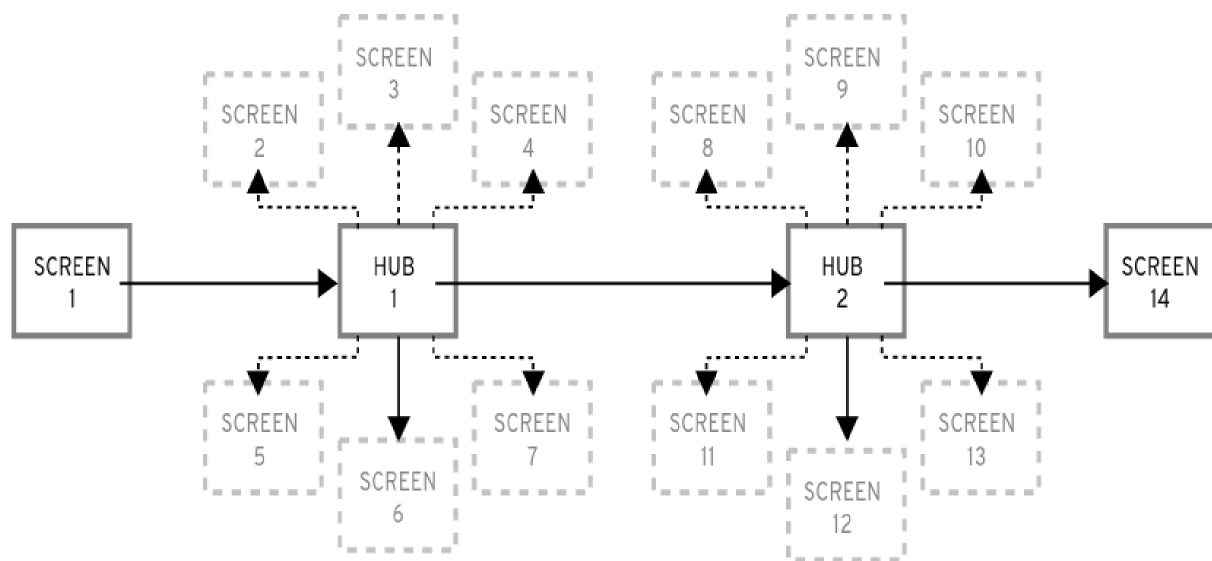
Anaconda ユーザーインターフェース (UI) には、ハブおよびスポークモデルとして知られる非リニア構造があります。

*利点は、 **Anaconda** ハブおよびスポークモデルは次のとおりです。*

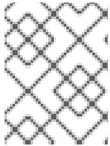
- *インストーラー画面をフォローする柔軟性。*
- *デフォルト設定を保持する柔軟性。*
- *設定された値の概要を説明します。*
- *拡張性をサポートします。ハブは、何も並べ替えることなく追加でき、複雑な順序の依存関係を解決できます。*
- *グラフィカルおよびテキストモードでのインストールをサポートします。*

*以下の図は、インストーラーレイアウトと **ハブ** と **スポーク** (スクリーン) 間の対話を可能にします。*

図4.1 ハブおよびスリープモデル



*この図では、画面 2-13 は **通常のスポーク** と呼ばれ、スクリーン 1 および 14 は **スタンドアロンのスポーク** です。スタンドアロンのスリープとは、スタンドアロンのドアまたはハブの前後に使用できる画面です。たとえば、インストール先頭の **Welcome** 画面では、残りのインストールに言語を選択するように求められます。*



注記

- ***インストールの概要*** *は、Anaconda の唯一のハブです。インストールを開始する前に設定されたオプションの概要が示されます。*

各スポークには、ハブを反映する以下の事前定義 **プロパティー** *があります。*

- ***ready***: *到着できるかどうかを示します。たとえば、インストーラーがパッケージソースを設定している場合は、スポークがグレーで色化され、設定が完了するまでアクセスできません。*
- ***completed***: *スポークが完了しているかどうかをマークします (必要なすべての値が設定されます)。*
- ***mandatory***: *インストールを続行する前に会議にアクセスする***必要がある***かどうかを判断します。たとえば、自動ディスクパーティション設定を使用しても、***インストール先***スポークに移動する必要があります。*
- ***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ースを明確にするために、スポークは **カテゴリー** *に分類されます。たとえば、ローカライゼーションカテゴリーグループは、***キーボードレイアウト*** の選択、言語サポート、タイムゾーン設定などに使用できます。*

各スリープには UI 制御が含まれており、1 つ以上のモジュールから値を表示し、変更できます。アドオンが提供するスポークにも同様のことが言えます。

4.4. ANACONDA スレッド間の通信

インストールプロセス中に実行する必要のあるアクションの一部には時間がかかる場合があります。たとえば、既存のパーティションのディスクをスキャンしたり、パッケージメタデータをダウンロードしたりできます。待機して応答しなくなるには、以下を実行します。 **Anaconda** *これらのアクションを別のスレッドで実行します。*

Gtk *Toolkit は、複数のスレッドからの要素の変更をサポートしません。メインのイベントループ* **Gtk** *その主なスレッドで稼働します。* **Anaconda** *実行するプロセスが必要です。したがって、GUI に関連するすべてのアクションはメインスレッドで実行する必要があります。これを行うには、***GLib.idle_add*** を使用しますが、これは常に簡単でも望ましい訳ではありません。このスクリプトで定義されているいくつかのヘルパー関数およびデコレーターは、* **pyanaconda.ui.gui.utils** *モジュールは難易度に追加される場合があります。*

@gtk_action_wait *および* ***@gtk_action_nowait*** *デコレーターにより、デコードされた関数またはメソッドが呼び出されたときに、メインスレッドで実行される Gtk のメインループに自動的にキューに置かれるように、デコレートされた関数またはメソッドが変更されます。戻り値はそれぞれ呼び出し元またはドロップされた値に返されます。*

*スポークやハブの通信では、準備ができ、ブロックされていないときにスポークが通知します。***hubQ*** メッセージキューはこの機能を処理し、メインのイベントループを定期的に確認します。スポークがアクセス可能になると、メッセージをキューに送付し、メッセージをブロックしないようにします。*

*スポークがステータスを更新するか、フラグを完了する必要がある場合にも、同じことが当てはまります。***Configuration and Progress*** ハブには、***progressQ*** と呼ばれる異なるキューがあり、インストール進捗の更新を転送するためのメディアとして機能します。*

これらのメカニズムは、テキストベースのインターフェースにも使用されます。テキストモードではメインループはありませんが、キーボード入力にかかる時間がほとんどありません。

4.5. ANACONDA モジュールおよび D-BUS ライブラリー

Anaconda のモジュールは、独立したプロセスとして実行されます。**D-Bus** API を使用してこれらのプロセスと通信するには、**dasbus** ライブラリーを使用します。

D-Bus'API を介したメソッドへの呼び出しは非同期ですが、'dasbus ライブラリーでは Python の同期メソッド呼び出しに変換できます。以下のプログラムのいずれかを書き込むこともできます。

- 非同期呼び出しとリターンハンドラーを含むプログラム
- 呼び出しが完了するまで呼び出しを待機する同期呼び出しを使用するプログラム

スレッドおよび通信の詳細は、「Anaconda スレッド間の通信」 を参照してください。

また、Anaconda は、モジュールで実行している Task オブジェクトを使用します。タスクには、追加のスレッドで自動的に実行される **D-Bus** API とメソッドがあります。タスクを正常に実行するには、**sync_run_task** および **async_run_task** ヘルパー関数を使用します。

4.6. HELLO WORLD アドオンの例

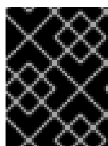
Anaconda 開発者が、GitHub で利用可能な「Hello World」というサンプルを公開しています。https://github.com/rhinstaller/hello-world-anaconda-addon/ その他のセクションの説明は、これで再現しています。

4.7. ANACONDA アドオン構造

Ansible は、 **Anaconda アドオンは、__init__.py** およびその他のソースディレクトリー（サブパッケージ）のディレクトリーを含む Python パッケージです。Python では各パッケージ名を一度だけインポートできるため、パッケージトップレベルのディレクトリー名には、一意のものを指定します。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。

アドオンに推奨される命名規則は Java パッケージまたは D-Bus サービス名に似ています。

ディレクトリー名が Python パッケージの有効な識別子になるようにするには、アドオン名を、ドットではなくアンダースコア (_) を使用して組織の逆引きドメイン名にプレフィックスを付ける必要があります。例: **com_example_hello_world**



重要

各ディレクトリーに **__init__.py** ファイルを作成してください。このファイルがないディレクトリーは、無効な Python パッケージとみなされます。

アドオンを作成する場合は、以下を確認してください。

- 各インターフェース (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) のサポートは個別のサブパッケージで利用可能です。このサブパッケージには、グラフィカルインターフェースには **gui**、テキストベースのインターフェースには **tui** という名前が付けられています。
- **gui** パッケージおよび **tui** パッケージには、**spoke** サブパッケージが含まれています。[1]
- パッケージに含まれるモジュールには任意の名前があります。

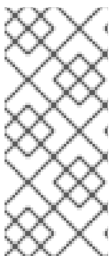
- **gui/** ディレクトリーおよび **tui/** ディレクトリーには、任意の名前を持つ Python モジュールが含まれます。
- アドオンの実際の作業を実行するサービスがあります。このサービスは、Python またはその他の言語で記述できます。
- サービスは、D-Bus およびキックスタートのサポートを実装します。
- アドオンには、サービスの自動起動を有効にするファイルが含まれます。

以下は、全インターフェース (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例です。

例4.1 アドオン構造の例

```
com_example_hello_world
├─ 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。



注記

すべてのアドオンは、docstring 規則の Python の PEP 8 と PEP 257 のガイドラインに従う必要があります。docstring の実際のコンテンツの形式には、 **Anaconda**; 唯一の要件は、人間が判読できることです。アドオンに自動生成されるドキュメントを使用する予定の場合には、docstrings はこれを行うために使用するツールキットのガイドラインに従う必要があります。

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は、カテゴリーサブパッケージを含めることができます。ただし、これは推奨されません。

4.8. ANACONDA サービスと設定ファイル

Anaconda サービスと設定ファイルは data/ ディレクトリーに含まれます。これらのファイルはアドオンサービスを起動し、D-Bus を設定する必要があります。

Anaconda Hello World アドオンの例を以下に示します。

例4.2 addon-name.conf の例:

```
<!DOCTYPE busconfig PUBLIC
"-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
"http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
    <policy user="root">
```

```
        <allow own="org.fedoraproject.Anaconda.Addons.HelloWorld"/>
        <allow send_destination="org.fedoraproject.Anaconda.Addons.HelloWorld"/>
    </policy>
    <policy context="default">
        <deny own="org.fedoraproject.Anaconda.Addons.HelloWorld"/>
        <allow send_destination="org.fedoraproject.Anaconda.Addons.HelloWorld"/>
    </policy>
</busconfig>
```

このファイルは、インストール環境の **/usr/share/anaconda/dbus/confs/** ディレクトリーに置く必要があります。**org.fedoraproject.Anaconda.Addons.HelloWorld** は D-Bus 上の addon のサービスの場所に対応する必要があります。

例4.3 addon-name.service の例:

```
[D-BUS Service]
# Start the org.fedoraproject.Anaconda.Addons.HelloWorld service.
# Runs org_fedora_hello_world/service/main.py
Name=org.fedoraproject.Anaconda.Addons.HelloWorld
Exec=/usr/libexec/anaconda/start-module org_fedora_hello_world.service
User=root
```

このファイルは、インストール環境の **/usr/share/anaconda/dbus/services/** ディレクトリーに置く必要があります。**org.fedoraproject.Anaconda.Addons.HelloWorld** は D-Bus 上の addon のサービスの場所に対応する必要があります。**Exec=** で始まる行の値は、インストール環境でサービスを開始する有効なコマンドである必要があります。

4.9. GUI アドオンの基本機能

アドオンのキックスタートサポートと同様に、GUI サポートでは、アドオンのすべての部分に、API で定義される特定のクラスから継承されたクラスを定義するモジュールが少なくとも 1 つ含まれる必要があります。グラフィカルアドオンサポートの場合、追加すべき唯一のクラスは **NormalSpoke** クラスで、これはスクリーンの通常スポークタイプのクラスとして **pyanaconda.ui.gui.spokes** に定義されています。詳細は、「Anaconda ユーザーインターフェース」 を参照してください。

NormalSpoke から継承された新しい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最上位オブジェクトを一覧表示します。これは、子オブジェクトでスポークに (再帰的に) 公開される必要があります。すべてをスポークに公開する必要がある場合 (非推奨) は、一覧は空でなければなりません。
- **mainWidgetName**: **.glade** ファイルで定義されているようにメインウィンドウウィジェット (Add Link) の ID が含まれます。
- **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。

4.10. アドオングラフィカルユーザーインターフェース (GUI) のサポートの追加

本セクションでは、以下の大まかな手順を実行することで、アドオンのグラフィカルユーザーインターフェース (GUI) にサポートを追加する方法を説明します。

1. Normalspoke クラスに必要な属性を定義します。
2. **__init__** と **initialize** メソッドを定義します。
3. **refresh**、**apply**、および **execute** メソッドを定義します。
4. **status** および **ready**、**completed** および **mandatory** のプロパティーを定義します。

前提条件

- アドオンには、キックスタートのサポートが含まれています。「Anaconda アドオン構造」 を参照してください。
- **Anaconda** 固有の Gtk ウィジェット (**SpokeWindow** など) が含まれる anaconda-widgets および anaconda-widgets-devel パッケージをインストールします。

手順

- 以下の例に従って、アドオングラフィカルユーザーインターフェース (GUI) のサポートを追加するために必要なすべての定義で以下のモジュールを作成します。

例4.4 Normalspoke クラスに必要な属性の定義

```
# 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(FirstbootSpokeMixIn, 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. Since it is also inherited form
    the FirstbootSpokeMixIn, it will also appear in the Initial Setup (successor
    of the Firstboot tool).

    :see: pyanaconda.ui.common.UIObject
    :see: pyanaconda.ui.common.Spoke
    :see: pyanaconda.ui.gui.GUIObject
    :see: pyanaconda.ui.common.FirstbootSpokeMixIn
    :see: pyanaconda.ui.gui.spokes.NormalSpoke

    """
```

```
    # 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* ***spoke*** *クラスをエクスポートします。この後に、以前* *「GUI アドオンの基本機能」* *で言及した、属性の定義を含む定義の最初の行が続きます。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。この他に、以下の 2 つの重要な属性があります。*

- ***category****。この値は、***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。アドオンへのパスが* ***sys.path*** *にある* ***HelloWorldCategory****。これにより、***com_example_hello_world*** *パッケージから値をインポートできます。***category*** *属性は* ***N_ function*** *名の一部で、変換用の文字列をマークしますが、変換は後の段階で行われるため、変換されていない文字列のバージョンを返します。*

- ***タイトル****。定義内にアンダースコアが1 つ含まれています。***title*** *属性アンダースコアは、タイトル自体の先頭をマークし、***Alt+H*** *キーボードショートカットを使用してスポークに到達できるようにします。*

通常、クラス定義のヘッダーとクラス ***attributes*** *の定義に続くのは、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ースオブジェクトの場合、新しいインスタンスの初期化には* ***__init__*** *メソッドおよび* ***initialize*** *メソッドの 2 つのメソッドがあります。*

*このようなメソッドが2 つある理由は、***spoke*** *の初期化に時間がかかる可能性があるため、あるタイミングで GUI オブジェクトがメモリーに作成され、別のタイミングで完全に初期化される可能性があるためです。したがって、***__init__*** *メソッドは親の* ***_init__*** *メソッドのみを呼び出し、たとえば、GUI 以外の属性を初期化する必要があります。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ー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。*

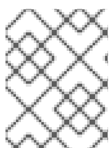
Hello World add-on *の例で、以下のようにこの 2 つのメソッドを定義します。***__init__*** *メソッドに渡される引数の数および説明をメモしてください。*

例4.5 __init__ と初期化メソッドの定義

```
def __init__(self, data, storage, payload):
```

```
        """
        :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

        """

        NormalSpoke.init(self, data, storage, payload)
        self._hello_world_module = HELLO_WORLD.get_proxy()

    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 separate thread.
        :see: pyanaconda.ui.common.UIObject.initialize
        """
        NormalSpoke.initialize(self)
        self._entry = self.builder.get_object("textLines")
        self._reverse = self.builder.get_object("reverseCheckButton")
```

__init__ メソッドに渡されるデータパラメーターは、すべてのデータが保存されるキックスタートファイルのメモリー内ツリーのような表示になります。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。



注記

storage object は RHEL8 以降利用できなくなりました。アドオンがストレージ設定と対話する必要がある場合は、**Storage DBus** モジュールを使用します。

HelloWorldData クラスは 「Hello World アドオンの例」 ですでに定義されているため、このアドオンの self.data にサブツリーがすでに存在します。クラスのインスタンスである root は **self.data.addons.com_example_hello_world** として利用できます。

ancestor の **__init__** が実行するもう 1 つのアクションは、**spoke's .glade** で GtkBuilder のインスタンスを初期化し、これを **self.builder** として保存することです。**initialize** メソッドはこれを使用して、キックスタートファイルの %addon セクションにあるテキストを表示し、変更するために使用される **GtkTextEntry** を取得します。

__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、スポークの主な役割は、スポークの値の表示と設定を変更または確認したいユーザーがアクセスすることです。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**: スポークがアクセスされようとするときに呼び出されます。このメソッドは、スポーク (主に UI 要素) の状態を更新し、表示されるデータが内部データ構造と一致するようにします。これにより、self.data 構造に保存されている現在の値が表示されるようにします。

- **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**: ユーザーがスポークを離れる場合に呼び出され、スポークの新しい状態に基づいてランタイムの変更を実行する際に使用されます。

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4.6 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
    internal data structures.
    :see: pyanaconda.ui.common.UIObject.refresh
    """
    lines = self._hello_world_module.Lines
    self._entry.get_buffer().set_text("".join(lines))
    reverse = self._hello_world_module.Reverse
    self._reverse.set_active(reverse)

def apply(self):
    """
    The apply method that is called when user leaves the spoke. It should
    update the D-Bus service with values set in the GUI elements.
    """
    buf = self._entry.get_buffer()
    text = buf.get_text(buf.get_start_iter(),
                        buf.get_end_iter(),
                        True)
    lines = text.splitlines(True)
    self._hello_world_module.SetLines(lines)

    self._hello_world_module.SetReverse(self._reverse.get_active())

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

複数の追加のメソッドを使用して、スポークの状態を制御できます。

- **ready**: スポークへアクセスできるかどうかを判断します。値が「False」の場合は **spoke** へアクセスできません。たとえば、パッケージソースを設定する前に **Package Selection** スポークにアクセスできません。
- **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別します。たとえば、自動パーティションを使用する場合でも、常にアクセスする必要がある* ***Installation Destination*** *スポークが必須かどうか判別します。*

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メソッドの実装例です。これには、****HelloWorldData*** *クラスの text 属性に特定の値を設定する必要があります。*

例4.7 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 reports whether the spoke is ready, that is, can be visited
    or not. The spoke is made (in)sensitive based on the returned value of the ready
    property.

    :rtype: bool

    """

    # this spoke is always ready
    return True


@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が定義された後、スポークはそのアクセス可能性と完全性を制御できますが、内部で構成された値のサマリーを提供することはできません。スポークにアクセスして、スポークがどのように構成されているかを確認する必要がありますが、これは望ましくない場合があります。このため、****status*** *という追加のプロパティーが存在します。このプロパティーには、設定された値の短いサマリーを含む1 行のテキストが含まれ、スポークタイトルの下のハブに表示することができます。*

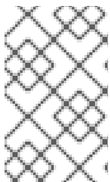
status プロパティーは、以下のように ***Hello World*** *の例のアドオンで定義されます。*

例4.8 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
"""
lines = self._hello_world_module.Lines
if not lines:
    return _("No text added")
elif self._hello_world_module.Reverse:
    return _("Text set with {} lines to reverse").format(len(lines))
else:
    return _("Text set with {} lines").format(len(lines))
```

例で説明しているプロパティーをすべて定義した後に、アドオンには、グラフィカルユーザーインターフェース (GUI) とキックスタートを示す完全なサポートがあります。



注記

ここで示した例は非常にシンプルで、制御を含むものはありません。GUI で機能的かつインタラクティブなスポークを開発するには、Python Gtk プログラミングに関する知識が必要です。

主な制限の 1 つとして、それぞれのスポークに独自のメインウィンドウ (**SpokeWindow** ウィジェットのインスタンス) が必要である点が挙げられます。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** など、GUI サポートでアドオンの開発に必要な他のファイルは、**anaconda-widgets-devel** パッケージで見つけることができます。

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、必要な方法をすべて含むと、以下のセクションを使用して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およびテスト」を引き続き使用してアドオンをテストすることができます。

4.11. アドオン GUI の高度な機能

pyanaconda パッケージには、ヘルパー関数やユーティリティー関数が複数含まれ、ハブやスポークで使用されるコンストラクトも含まれます。そのほとんどは、**pyanaconda.ui.gui.utils** パッケージにあります。

Hello World アドオンの例は、Anaconda も使用する **englightbox** コンテンツマネージャーの使用方法を示しています。このコンテンツマネージャーはウィンドウをライトボックスに入れ、可視性を高め、ユーザーの基礎となるウィンドウとの対話を防ぐことに重点を置くことができます。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。

ダイアログクラスは、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4.9 englightbox Dialog の定義

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

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

Defining an englightbox Dialog のサンプルコードは、ダイアログのインスタンスを作成してから、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ダイアログを実行します。コンテキストマネージャーにはスポークのウィンドウへの参照があり、ダイアログのライトボックスをインスタンス化するためにダイアログのウィンドウだけを必要とします。

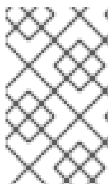
Anaconda が提供するもう 1 つの便利な機能は、インストール時および最初の再起動後に表示されるスポークを定義する機能です。**Initial Setup** ユーティリティーは、「アドオングラフィカルユーザーインターフェース (GUI) のサポートの追加」 で説明されています。Anaconda と初期セットアップの両方でスポークを使用できるようにするには、**pyanaconda.ui.common** モジュールで定義される最初の継承クラスとして、特別な **FirstbootSpokeMixIn** クラス (別称 **mixin**) を継承する必要があります。

初期セットアップでのみ特定のスポークを利用可能にする場合は、このスポークは代わりに **FirstbootOnlySpokeMixIn** クラスを継承する必要があります。

pyanaconda パッケージは、**@gtk_action_wait** および **@gtk_action_nowait** デコレーターなど、より高度な機能を提供しますが、これらは本書の対象外となっています。その他の例については、インストーラーのソースを参照してください。

4.12. TUI アドオンの基本機能

Anaconda は、テキストベースのインターフェース (TUI) にも対応しています。このインターフェースは機能がさらに制限されていますが、一部のシステムでは、インタラクティブインストールの唯一の選択肢となる場合があります。テキストベースのインターフェ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 およびアドオンの概要」 を参照してください。



注記

テキストインターフェースのサポートをアドオンに追加するには、「Anaconda アドオン構造」 の説明に従って、tui ディレクトリー内に新しいサブパッケージのセットを作成します。

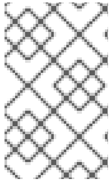
インストーラーのテキストモードサポートは **simpleline** ライブラリーに基づいており、非常にシンプルなユーザーの対話のみを許可します。テキストモードインターフェースは、

- カーソルの移動には対応していません。代わりに、ラインプリンターのように動作します。
- 視覚的機能拡張 (異なる色やフォントの使用など) はサポートしません。

内部的には、**simpleline** ツールキットには、**App**、**UIScreen** および **Widget** の 3 つの主要クラスがあります。ウィジェットは、画面に出力される情報が含まれるユニットです。これらは、App クラスの単一のインスタンスによって切り替えられる UIScreens に配置されます。基本的な要素に加え、**hubs**、**spoke`s and `dialogs** はすべて、グラフィカルインターフェースと同じような方法でさまざまなウィジェットを含んでいます。

アドオンで最も重要なクラスは、**NormalTUISpoke** および **pyanaconda.ui.tui.spokes** パッケージで定義される他のさまざまなクラスです。これらのクラスはすべて **TUIObject** クラスをベースとしています。このクラス自体は、??? で説明されている **GUIObject** クラスと同じものになります。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ースは 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 引数と同様にスポークのタイトルを決定します。
- ***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

TUI は GUI とは異なる方法でカテゴリーを処理します。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。

各スポークは、複数のメソッドをオーバーライドすることも想定されています。複数のメソッドとは **init**、***initialize***、***refresh***、***refresh***、***apply***、***execute***、***input***、***prompt***、および ***properties*** (***ready***、***completed***、***mandatory***、および ***status***) になります。

関連情報

- 「アドオングラフィカルユーザーインターフェース (GUI) のサポートの追加」 を参照してください。

4.13. シンプルな TUI SPOKE の定義

以下の例は、Hello World サンプルアドオンのシンプルなText User Interface (TUI) スポークの実装を示しています。

前提条件

- 「Anaconda アドオン構造」 の説明に従って、tui ディレクトリーに新しいサブパッケージセットを作成している。

手順

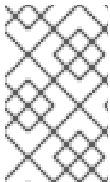
- 以下の例に従って、アドオン Text User Interface (TUI) のサポートを追加するために必要なすべての定義を含むモジュールを作成します。

例4.10 シンプルな TUI Spoke の定義

```
def init(self, app, data, storage, payload, instclass):
    """
    :see: simpleline.render.screen.UIScreen
    :param data: data object passed to every spoke to load/store data
                from/to it
    :type data: pykickstart.base.BaseHandler
    :param storage: dummy object storing storage-related information
     :type storage: blivet.Blivet
    :param payload: object storing packaging-related information
    :type payload: pyanaconda.packaging.Payload

    """

    NormalTUISpoke.init(self, data, storage, payload)
    self.title = N_("Hello World")
    self._container = None
    self._hello_world_module = HELLO_WORLD.get_proxy()
```

```
        self._reverse = False
        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 separate thread.

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

        """

        NormalTUISpoke.initialize(self)
        self._reverse = self._hello_world_module.Reverse
        self._entered_text = "".join(self._hello_world_module.Lines)

    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
        internal data structures.

        :see: pyanaconda.ui.common.UIObject.refresh
        :see: simpleline.render.screen.UIScreen.refresh
        :param args: optional argument that may be used when the screen is
                     scheduled
        :type args: anything

        """

        self._reverse = self._hello_world_module.Reverse
        self._entered_text = "".join(self._hello_world_module.Lines)

        self._container = ListColumnContainer(columns=1)
        self.window.add(self._container)

        self._container.add(CheckboxWidget(title="Reverse", completed=self._reverse),
                            callback=self._change_reverse)
        self._container.add(EntryWidget(title="Hello world text", value=self._entered_text),
                            callback=self._change_lines)

        self._window.add_separator()

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

        """

        self._hello_world_module.SetReverse(self._reverse)
        lines = self._entered_text.splitlines(True)
        self._hello_world_module.SetLines(lines)

    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
        :type args: anything
        :param key: user's input
        :type key: unicode
        :return: if the input should not be handled here, return it, otherwise
                 return InputState.PROCESSED or InputState.DISCARDED if the input was
                 processed successfully or not respectively
        :rtype: enum InputState

        """

        if self._container.process_user_input(key):
            return InputState.PROCESSED_AND_REDRAW
        else:
            return super().input(args=args, key=key)
```



注記

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。

上の例では、以下のようになります。

- ***initialize*** メソッドでは、スポークの内部属性のデフォルト値を設定します。これは、refresh メソッドにより更新され、apply メソッドによって内部データ構造を更新するために使用されます。
- ***execute*** メソッドの目的は、GUI の同等のメソッドの目的と同じです。この場合、このメソッドは効果がありません。
- ***input*** メソッドはテキストインターフェー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。***input*** メソッドはユーザーの対話を行います。
- ***input*** メソッドは、入力された文字列を処理し、そのタイプと値に応じてアクションを取ります。上記の例は値を要求してから、それを内部属性 (キー) として保存します。より複雑なアドオンでは、通常、文字をアクションとして解析したり、数値を整数に変換したり、追加の画面を表示したり、ブール値を切り替えたりするなど、重要なアクションを実行する必要があります。

- この入力を別の画面で処理する必要がある場合に備えて、入力クラスの **return** 値は、**InputState** enum または **input** 文字列自体のいずれか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。スポークの画面を閉じる (非表示にする) 場合も同様です。**close** メソッドから明示的に呼び出す必要があります。

別のスポークで入力した追加情報が必要な場合など、別の画面を表示するには、別の **TUIObject** をインスタンス化し、**ScreenHandler.push_screen_modal()** を使用してそれを表示します。

テキストベースのインターフェースの制限により、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構成されます。

4.14. NORMALTUISPOKE を使用したテキストインターフェーススポークの定義

「シンプルな TUI Spoke の定義」 の例では、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 **Normal 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ことで、設定する必要のあるフィールドと属性を指定するだけで、一般的な TUI スポークを実装できます。以下の例で説明します。

前提条件

- 「Anaconda アドオン構造」 で説明されているように、**TUI** ディレクトリーに新しいサブパッケージのセットを追加している。

手順

- 以下の例に従って、アドオン Text User Interface (TUI) のサポートを追加するために必要なすべての定義を含むモジュールを作成します。

例4.11 NormalTUISpoke を使用したテキストインターフェーススポークの定義

```
class HelloWorldEditSpoke(NormalTUISpoke):
    """Example class demonstrating usage of editing in TUI"""

    category = HelloWorldCategory

    def init(self, data, storage, payload):
        """
        :see: simpleline.render.screen.UIScreen
        :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
        """
        NormalTUISpoke.init(self, data, storage, payload)

        self.title = N_("Hello World Edit")
```

```
        self._container = None
        # values for user to set
        self._checked = False
        self._unconditional_input = ""
        self._conditional_input = ""

    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: simpleline.render.screen.UIScreen.refresh
        :param args: optional argument that may be used when the screen is
                     scheduled
        :type args: anything
        """
        super().refresh(args)
        self._container = ListColumnContainer(columns=1)

        # add ListColumnContainer to window (main window container)
        # this will automatically add numbering and will call callbacks when required
        self.window.add(self._container)

        self._container.add(CheckboxWidget(title="Simple checkbox", completed=self._checked),
                            callback=self._checkbox_called)
        self._container.add(EntryWidget(title="Unconditional text input",
                                        value=self._unconditional_input),
                            callback=self._get_unconditional_input)

        # show conditional input only if the checkbox is checked
        if self._checked:
            self._container.add(EntryWidget(title="Conditional password input",
                                            value="Password set" if self._conditional_input
                                            else ""),
                                callback=self._get_conditional_input)

        self._window.add_separator()


    @property
    def completed(self):
        # completed if user entered something non-empty to the Conditioned input
        return bool(self._conditional_input)
    @property
    def status(self):
        return "Hidden input %s" % ("entered" if self._conditional_input
                                    else "not entered")

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

関連情報

- **HelloWorldEditSpoke** の詳細については、「NormalTUISpoke を使用したテキストインターフェーススポークの定義」 を参照してください。

4.15. ANACONDA アドオンのデプロイおよびテスト

独自の Anaconda アドオンをインストール環境にデプロイしてテストできます。これを行うには、次の手順を実行します。

前提条件

- アドオンを作成している。
- **D-Bus** ファイルにアクセスできる。

手順

1. ディレクトリーの作成 **DIR** 好みの希望箇所で開催されます。
2. **Add-on** python ファイルを **DIR/usr/share/anaconda/addons/**.
3. **D-Bus** サービスファイルをコピーします。 **DIR/usr/share/anaconda/dbus/services/**.
4. **D-Bus** サービス設定ファイルを **/usr/share/anaconda/dbus/confs/** にコピーします。
5. Create the **updates** イメージ。
   アクセス権は、 **DIR** ディレクトリー：

   ```
   cd DIR
   ```

   次を見つけます。 **updates** イメージ。

   ```
   find . | cpio -c -o | pigz -9cv > DIR/updates.img
   ```

6. ISO ブートイメージの内容を抽出します。
7. 作成されたものを使用します。 **updates** イメージ：
   a. 次の追加： **updates.img** ファイルを展開された ISO コンテンツの **images** ディレクトリーに置く。
   b. イメージを再パッケージ化します。
   c. 提供用に Web サーバーを設定します。 **updates**HTTP 経由で Anaconda インストーラーへの .img ファイル。
   d. load **updates**以下の仕様を起動オプションに追加して、システムの起動時に .img ファイルを指定します。

      ```
      inst.updates=http://your-server/whatever/updates.img to boot options.
      ```

既存のブートイメージを展開し、**product.img** ファイルを作成してイメージを再パッケージ化する方法は、「Red Hat Enterprise Linux ブートイメージの抽出」 を参照してください。

*[1] **gui** パッケージにも、 **categories** 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は、サブパッケージは推奨されません。*

第5章 ポストカスタマイズタスクの完了

カスタマイズを行うには、以下のタスクを実行します。

- product.img イメージファイルを作成します (グラフィカルカスタマイズにのみ適用)。
- カスタムのブートイメージを作成します。

本セクションでは、product.img イメージファイルを作成し、カスタムブートイメージを作成する方法を説明します。

5.1. プロダクトの IMG ファイルの作成

product.img イメージファイルは、実行時に既存のインストーラーファイルを置き換える新しいインストーラーファイルを含むアーカイブです。

システムの起動時に、 **Anaconda** 起動メディアの images/ ディレクトリーから product.img ファイルを読み込みます。その後、このディレクトリー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で同じ名前が付けられたファイルを置き換えます。置き換えると、インストーラーをカスタマイズします (例: デフォルトのイメージをカスタムイメージに置き換えるためなど)。

注記: **product.img** イメージには、インストーラーと同じディレクトリー構造が含まれている必要があります。インストーラーディレクトリー構造の詳細は、表 表5.1「インストーラーのディレクトリー構造およびカスタムコンテンツ」 を参照してください。

表5.1 インストーラーのディレクトリー構造およびカスタムコンテンツ

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/**
GUI スタイルシート	**/usr/share/anaconda/anaconda-gtk.css**
Anaconda アドオン	**/usr/share/anaconda/addons/**
製品設定ファイル	**/etc/anaconda/product.d/**
カスタム設定ファイル	**/etc/anaconda/conf.d/**
Anaconda DBus サービスの conf ファイル	**/usr/share/anaconda/dbus/confs/**
Anaconda DBus サービスファイル	**/usr/share/anaconda/dbus/services/**

以下の手順では、**product.img** ファイルの作成方法を説明します。

手順

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。

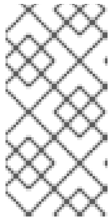
```
$ cd /tmp
```

2. サブディレクトリー product/ を作成します。

```
$ mkdir product/
```

3. 置き換え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。たとえば、インストールシステムの **/usr/share/anaconda/addons** ディレクトリーにあるアドオンをテストする場合は、作業ディレクトリーに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を表示するには、インストールを起動し、仮想コンソール 1(**Ctrl**+**Alt**+**F1**)**に切り替えてから** 2 番目に切り替えます。 **tmux** ウィンドウ(**Ctrl**+**b 2**)。ファイルシステムの閲覧に使用できるシェルプロンプトが開きます。

4. カスタムファイル（この例ではカスタムアドオン）を配置する **Anaconda**）新規作成したディレクトリーで、以下を実行します。

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

5. インストーラーに追加するすべてのファイルについて、ステップ 3 および 4 を繰り返します (ディレクトリー構造を作成してカスタムファイルをそこに配置します)。

6. ディレクトリーのルートに **.buildstamp** ファイルを作成します。**.buildstamp** ファイルは、システムバージョン、製品、およびその他のパラメーターを説明します。以下は、Red Hat Enterprise Linux 8.4 の **a.buildstamp** ファイルの例です。

```
[Main]
Product=Red Hat Enterprise Linux
Version=8.4
BugURL=https://bugzilla.redhat.com/
IsFinal=True
UUID=202007011344.x86_64
[Compose]
Lorax=28.14.49-1
```

IsFinal パラメーターは、イメージが製品のリリース(GA)バージョン (**True**) か、Alpha、Beta、または内部マイルストーン (**False**) などのプレリリースであるかを指定します。

7. **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。

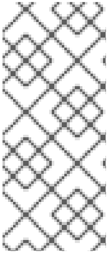
```
$ cd product
```

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

これにより、**product / ディレクトリーの上に product.img** ファイルが 1 レベル上に作成されます。

8. **product.img** ファイルを、展開した ISO イメージの **images/** ディレクトリーに移動します。

これで product.img ファイルが作成され、作成するカスタマイズがそれぞれのディレクトリーに配置されるようになりました。



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、「Anaconda Boot Options」を参照してください。

5.2. カスタムブートイメージの作成

ブートイメージと GUI レイアウトをカスタマイズしたら、変更を含む新しいイメージを作成します。

カスタムブートイメージを作成するには、以下の手順に従います。

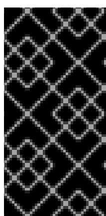
手順

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。たとえば、アドオンをテストする場合は、**images/** ディレクトリーに **product.img** を配置してください。
2. 現在の作業ディレクトリーが、展開した ISO イメージの最上位ディレクトリーであることを確認します (例: **/tmp/ISO/iso**)。
3. **genisoimage** を使用して新しい ISO イメージを作成します。

```
# genisoimage -U -r -v -T -J -joliet-long -V "RHEL-8.2 Server.x86_64" -volset "RHEL-8.2 Server.x86_64" -A "RHEL-8.2 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 **/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。たとえば、**RHEL 8.2** ラベルが付いた ISO イメージを作成する場合は、ブートローダー設定に **RHEL\x207.1** を使用する必要があります。

- **-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。
このコマンドの詳細は、**genisoimage(1)** の 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。MD5 checksu を使用しないと、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) が失敗し、インストールがハングする可能性があることに注意してください。*

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

    上記の例では、 **../NEWISO.iso** *を、直前の手順で作成したファイル名と ISO イメージの場所に置き換えます。*

    これで、新しい ISO イメージを物理メディアまたはネットワークサーバーに書き込んで物理ハードウェアで起動することや、仮想マシンのインストールを開始できるようになりました。

関連情報

- *ブートメディアまたはネットワークサーバーの準備手順は、『高度な RHEL インストールの実行』参照してください。*
- *ISO イメージを使用した仮想マシンの作成手順は、「仮想化の設定および管理」を参照してください。*