Red Hat Developer Tools 2019.3

Rust Toolset の使用

Rust Toolset 1.35.0 のインストールおよび使用

Last Updated: 2020-09-17

Red Hat Developer Tools 2019.3 Rust Toolset の使用

Rust Toolset 1.35.0 のインストールおよび使用

Peter Macko
pmacko@redhat.com

Kevin Owen

Vladimir Slavik

法律上の通知

概要

Rust Toolset は、Red Hat Enterprise Linux プラットフォームで開発者向けの Red Hat 製品です。Rust Toolset ユーザーガイド では、この製品の概要、ツールの Rust Toolset バージョンの呼び出しおよび使用方法、および詳細な情報を含むリソースへのリンクを説明します。

目次

第1章 RUST TOOLSET

1.1. RUST TOOLSET について

Rust Toolset は、Red Hat Enterprise Linux プラットフォームで開発者向けの Red Hat 製品です。Rust Toolset 1.41 は、Rust プログラミング言語コンパイラー **rustc**、**cargo** ビルドツールおよび依存マネージャー、**cargo-vendor** プラグイン、**rustfmt** プラグイン、および必要なライブラリーを提供します。

Rust Toolset は、Red Hat Enterprise Linux 7 の一部として配布されており、Red Hat Enterprise Linux 8 でモジュールとして利用できます。

以下のコンポーネントは、Rust Toolset の一部として利用できます。

表1.1 Rust Toolset コンポーネント

名前	バージョン	説明
rust	1.35.0	LLVM 用の Rust コンパイラーフロントエンド。
cargo	1.35.0	Rust のビルドシステムおよび依存関係マネージャー。
cargo-vendor	0.1.23	ベンダー crates.io 依存関係に対する cargo サブコマンド。
rustfmt	1.35.0	Rust コードの自動フォーマットを行うためのツール。

1.2. 互換性

Rust Toolset は、Red Hat Enterprise Linux 7 および Red Hat Enterprise Linux 8 では、以下のアーキテクチャーで利用できます。

- 64 ビット Intel および AMD アーキテクチャー
- 64 ビット ARM アーキテクチャー
- IBM Power Systems アーキテクチャー
- IBM Power Systems アーキテクチャーのリトルエンディアンバリアント
- IBM Z Systems アーキテクチャー

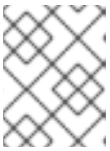
1.3. RED HAT ENTERPRISE LINUX 7 で RUST TOOLSET へのアクセス

Rust Toolset は、Red Hat Developer Tools コンテンツセットの一部として配布されるオファリングです。Red Hat Enterprise Linux 7 のデプロイメントをご利用いただけます。Rust Toolset をインストールするには、Red Hat Subscription Management を使用して Red Hat Developer Tools および Red Hat Software Collections リポジトリーを有効にし、システムに Red Hat Developer Tools GPG キーを追加します。

1. **rhel-7-variant-devtools-rpms** リポジトリーを有効にするには、以下を行います。

```
# subscription-manager repos --enable rhel-7-variant-devtools-rpms
```

variant を、Red Hat Enterprise Linux システムのバリアント (**server** または **workstation**) に置き換えます。



注記

Red Hat Enterprise Linux Server を使用して、最も幅広い開発ツールにアクセスすることを検討してください。

2. **rhel-variant-rhscl-7-rpms** リポジトリーを有効にします。

```
# subscription-manager repos --enable rhel-variant-rhscl-7-rpms
```

variant を、Red Hat Enterprise Linux システムのバリアント (**server** または **workstation**) に置き換えます。

3. Red Hat Developer Tools キーをシステムに追加します。

```
# cd /etc/pki/rpm-gpg
# wget -O RPM-GPG-KEY-redhat-devel https://www.redhat.com/security/data/a5787476.txt
# rpm --import RPM-GPG-KEY-redhat-devel
```

サブスクリプションがシステムに割り当てられ、リポジトリーを有効にしたら、「Rust Toolset のインストール」 の説明に従って Red Hat Rust Toolset をインストールできます。

関連資料

- Red Hat Subscription Management を使用してシステムを登録し、サブスクリプションに関連付ける方法は、Red Hat Subscription Management のガイドを参照してください。
- Red Hat Software Collections のサブスクリプションに関する詳しい説明は、**Red Hat Developer Toolset User Guide** の セクション 1.4 を参照してください。Red Hat Developer Toolset へのアクセスの取得

1.4. RUST TOOLSET のインストール

Rust Toolset は、Red Hat Enterprise Linux に含まれる標準のパッケージ管理ツールを使用してインストール、更新、アンインストール、および検査できる RPM パッケージのコレクションとして配布されています。Rust Toolset を Red Hat Enterprise Linux 7 システムにインストールするには、Red Hat Developer Tool コンテンツセットへのアクセスを提供する有効なサブスクリプションが必要です。Red Hat Enterprise Linux 7 システムを適切なサブスクリプションに関連付け、Rust Toolset にアクセスする方法は、「Red Hat Enterprise Linux 7 で Rust Toolset へのアクセス」 を参照してください。

重要

Rust Toolset をインストールする前に、利用可能なすべての Red Hat Enterprise Linux 更新をインストールします。

1. オペレーティングシステムの Rust Toolset に含まれるすべてのコンポーネントをインストールします。

- Red Hat Enterprise Linux 7 で、**rust-toolset-1.35** パッケージをインストールします。

```
# yum install rust-toolset-1.35
```

- Red Hat Enterprise Linux 8 に、**rust-toolset** モジュールをインストールします。

```
# yum module install rust-toolset
```

これにより、すべての開発およびデバッグツール、および依存するパッケージがシステムにインストールされます。特に、Rust Toolset には Clang および LLVM Toolset の依存関係があります。

1.5. 関連資料

Rust プログラミング言語の詳細な説明と、その機能はすべて本ガイドの対象外です。詳細は、以下に記載のドキュメントを参照してください。

インストール可能なドキュメント

- HTML 形式で、**The Rust Programming Language** および API ドキュメントをインストールします。
  - Red Hat Enterprise Linux 7 で、**rust-toolset-1.35-rust-doc** パッケージをインストールします。

```
# yum install rust-toolset-1.35-rust-doc
```

本書は、**/opt/rh/rust-toolset-1.35/usr/share/doc/rust/html/index.html** でご利用いただけます。

すべての crates の API ドキュメントは、**/opt/rh/rust-toolset-1.35/usr/share/doc/rust/html/std/index.html** で HTML 形式で入手できます。

  - Red Hat Enterprise Linux 8 で、**rust-doc** パッケージをインストールします。

```
# yum install rust-doc
```

本書は、**/usr/share/doc/rust/html/index.html** でご利用いただけます。

すべての crates の API ドキュメントは、**/usr/share/doc/rust/html/std/index.html** で HTML 形式で入手できます。

オンラインドキュメント

- Rust documentation: アップストリームの Rust ドキュメント

第2章 CARGO

cargo は、Rust プログラミング言語を使用して開発するためのツールです。**cargo** は以下のロールに対応します。

- Rust コンパイラー **rustc** のビルドツールおよびフロントエンド。

注記

rustc よりも **cargo** の使用を検討してください。

- パッケージおよび依存関係マネージャー。
  cargo により、Rust プロジェクトは特定のバージョン要件で依存関係を宣言できます。**cargo** は完全な依存関係グラフを解決し、必要に応じてパッケージをダウンロードし、プロジェクト全体をビルドおよびテストします。

Rust Toolset には、**cargo 1.35.0** が同梱されています。

2.1. CARGO のインストール

- Rust Toolset をインストールします。「Rust Toolset のインストール」 を参照してください。

注記

Rust Toolset の一部として、**cargo** がインストールされます。

Red Hat Enterprise Linux 7 では、**rust-toolset-1.35-cargo** パッケージで **cargo** が提供されます。

Red Hat Enterprise Linux 8 では、**rust-toolset** モジュールにより **cargo** が提供されます。

2.2. 新規プロジェクトの作成

コマンドラインで Rust プログラムを作成するには、以下のように **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo new --bin project_name'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo new --bin project_name
```

これにより、**Cargo.toml** という名前のテキストファイルを含む **project_name** ディレクトリーと、**main.rs** という名前のテキストファイルを含むサブディレクトリー **src** が作成されます。

プロジェクトを設定し、依存関係を追加するには、ファイル **Cargo.toml** を編集します。「プロジェクト依存関係の設定」 を参照してください。

プロジェクトコードを編集するには、必要に応じて、**main.rs** ファイルを編集し、**src** サブディレクトリーに新しいソースファイルを追加します。

プログラムではなく cargo パッケージのプロジェクトを作成するには、以下のようにコマンドラインで **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo new --lib project_name'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo new --lib project_name
```

Red Hat Enterprise Linux 7 で **scl** ユーティリティーを使用してコマンドを実行すると、利用可能な Rust Toolset バイナリーで実行することができることに注意してください。これにより、Rust Toolset **cargo** コマンドでシェルセッションを直接実行できるようになります。

```
$ scl enable rust-toolset-1.35 'bash'
```

例2.1 cargo を使用したプロジェクトの作成

helloworld という名前の新規 Rust プロジェクトを作成します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo new --bin helloworld'
    Created binary (application) helloworld project
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo new --bin helloworld
    Created binary (application) helloworld project
```

結果を確認します。

```
$ cd helloworld
$ tree
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
```

プロジェクトメタデータを追跡するためのファイルと、メインのソースコードファイル **main.rs** が含まれるサブディレクトリー **src** とともに、ディレクトリー **helloworld** がプロジェクト **Cargo.toml** 用に作成されます。

ソースコードファイル **main.rs** は **cargo** によって、サンプル hello world プログラムに初期化されています。

注記

tree ツールは、デフォルトの Red Hat Enterprise Linux リポジトリーから利用できます。インストールするには、以下を行います。

```
# yum install tree
```

2.3. プロジェクトのビルド

コマンドラインで Rust プロジェクトをビルドするには、プロジェクトディレクトリーに移動し、以下のように **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo build'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo build
```

これにより、プロジェクトのすべての依存関係が解決され、不足している依存関係がダウンロードされ、**rustc** コンパイラーを使用してプロジェクトをコンパイルします。

デフォルトでは、プロジェクトはデバッグモードでビルドおよびコンパイルされます。リリースモードでプロジェクトをビルドするには、以下のように **--release** オプションを指定して **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo build --release'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo build --release
```

例2.2 cargo を使用したプロジェクトのビルド

この例では、例2.1「cargo を使用したプロジェクトの作成」 従って Rust プロジェクト **helloworld** を正常に作成していることを前提としています。

ディレクトリーに移動して**helloworld**、プロジェクトをビルドします。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo build'
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.51 secs
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo build
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.51 secs
```

結果を確認します。

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    └── debug
        ├── build
        ├── deps
        │   └── helloworld-b7c6fab39c2d17a7
        ├── examples
        ├── helloworld
        ├── helloworld.d
        ├── incremental
        └── native

8 directories, 6 files
```

ディレクトリー **target** から開始して、サブディレクトリー構造が作成されました。このプロジェクトはデバッグモードでビルドされているため、実際のビルド出力は追加のサブディレクトリー **debug** に含まれます。実際に生成される実行可能ファイルは **target/debug/helloworld** です。

注記

tree ツールは、デフォルトの Red Hat Enterprise Linux リポジトリーから利用できます。インストールするには、以下を行います。

```
# yum install tree
```

2.4. プログラムの確認

cargo が管理する Rust プログラムを構築するには、コマンドラインでプロジェクトディレクトリーに移動し、以下のように **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo check'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo check
```

注記

実行可能なコードが必要ない場合は、Rust プログラム有効性の検証に **cargo build** ではなく **cargo check** を使用することを検討してください。この **cargo check** コマンドは、実行可能なコードを生成しないため、**cargo build** コマンドを使用した完全なプロジェクトビルドよりも高速です。

デフォルトでは、プロジェクトはデバッグモードでチェックされます。リリースモードでプロジェクトを確認するには、以下のように **--release** オプションを指定して **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo check --release'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo check --release
```

例2.3 cargo でのプログラムの確認

この例では、例2.2「cargo を使用したプロジェクトのビルド」 従って Rust プロジェクト **helloworld** を正常に構築していることを前提としています。

ディレクトリー **helloworld** に移動し、プロジェクトを確認します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo check'
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.5 secs
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo check
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.5 secs
```

プロジェクトがチェックされ、**cargo build** コマンドの出力に似ています。ただし、実行ファイルは生成されません。これを確認するには、現在の時間を実行ファイルのタイムスタンプと比較します。

```
$ date
Fri Oct 13 08:53:21 CEST 2017
$ ls -l target/debug/helloworld
-rwxrwxr-x. 2 vslavik vslavik 252624 Oct 13 08:48 target/debug/helloworld
```

2.5. プログラムの実行

コマンドラインで **cargo** によってプロジェクトとして管理される Rust プログラムを実行するには、以下のようにプロジェクトディレクトリーに移動し、**cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo run'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo run
```

プログラムがビルドされていない場合、**cargo** はプログラムを実行する前にビルドを実行します。

注記

開発中は、**cargo** を使用して Rust プログラムを実行することを検討してください。これは、ビルドモードとは独立して、出力パスを正しく解決します。

デフォルトでは、プロジェクトはデバッグモードでビルドされます。実行前にリリースモードでプロジェクトをビルドするには、以下のように **--release** オプションを指定して **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo run --release'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo run --release
```

例2.4 cargo でのプログラムの実行

この例では、例2.2「cargo を使用したプロジェクトのビルド」 従って Rust プロジェクト **helloworld** を正常に構築していることを前提としています。

ディレクトリー **helloworld** に移動し、プロジェクトを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo run'
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/helloworld
Hello, world!
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/helloworld
Hello, world!
```

cargo はまずプロジェクトを再構築し、作成された実行ファイルを実行します。

この例では、最後のビルド以降にソースコードに変更がありませんでした。そのため、**cargo** は実行可能ファイルを再構築する必要はなく、現時点では受け入れるだけでした。

2.6. プロジェクトテストの実行

コマンドラインで **cargo** プロジェクトでテストを実行するには、プロジェクトディレクトリーに移動し、以下のように **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo test'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo test
```

デフォルトでは、プロジェクトはデバッグモードでテストされています。リリースモードでプロジェクトをテストするには、以下のように **--release** オプションを指定して **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo test --release'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo test --release
```

例2.5 cargo を使用したプロジェクトのテスト

この例では、例2.2「cargo を使用したプロジェクトのビルド」 従って Rust プロジェクト **helloworld** を正常に構築していることを前提としています。

ディレクトリー **helloworld** に移動し、以下のソースコードが含まれるファイル **src/main.rs** を編集します。

```
fn main() {
    println!("Hello, world!");
}

#[test]
fn my_test() {
    assert_eq!(21+21, 42);
}
```

テストとして **my_test** マークされた関数が追加されました。

ファイルを保存し、テストを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo test'
   Compiling helloworld v0.1.0 (file:///home/vslavik/Documentation/rusttest/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.26 secs
     Running target/debug/deps/helloworld-9dd6b83647b49aec

running 1 test
test my_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo test
```

```
   Compiling helloworld v0.1.0 (file:///home/vslavik/Documentation/rusttest/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.26 secs
     Running target/debug/deps/helloworld-9dd6b83647b49aec

running 1 test
test my_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

cargo はまずプロジェクトを再構築し、次にプロジェクトにあるテストを実行します。テスト **my_test** に成功しました。

2.7. プロジェクト依存関係の設定

cargo プロジェクトの依存関係を指定するには、プロジェクトディレクトリーの **Cargo.toml** ファイルを編集します。このセクション **[dependencies]** には、プロジェクトの依存関係の一覧が含まれます。各依存関係は、次の形式で新しい行に一覧表示されます。

crate_name = **version**

Rust コードパッケージは crates と呼ばれます。

例2.6 プロジェクトへの依存関係の追加および cargo で構築

この例では、例2.2「cargo を使用したプロジェクトのビルド」 従って Rust プロジェクト **helloworld** を正常に構築していることを前提としています。

ディレクトリー **helloworld** に移動し、以下のソースコードが含まれるファイル **src/main.rs** を編集します。

```
extern crate time;

fn main() {
    println!("Hello, world!");
    println!("Time is: {}", time::now().rfc822());
}
```

このコードには、外部のクレート **時間** が必要になりました。以下のコードが含まれるように **Cargo.toml** ファイルを編集して、この依存関係をプロジェクト設定に追加します。

```
[package]
name = "helloworld"
version = "0.1.0"
authors = ["Your Name <yourname@example.com>"]

[dependencies]
time = "0.1"
```

cargo run コマンドを実行してプロジェクトをビルドし、作成された実行ファイルを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo run'
```

```
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading time v0.1.38
 Downloading libc v0.2.32
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/helloworld`
Hello, world!
Time is: Fri, 13 Oct 2017 11:08:57
```

- Red Hat Enterprise Linux 8 の場合:

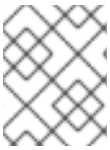
```
$ cargo run
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading time v0.1.38
 Downloading libc v0.2.32
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/helloworld`
Hello, world!
Time is: Fri, 13 Oct 2017 11:08:57
```

cargo は **time** crate とその依存関係 (crate **libc**) をダウンロードし、ローカルに保管し、依存関係を含むすべてのプロジェクトのソースコードをビルドし、最終的に生成される実行ファイルを実行します。

関連資料

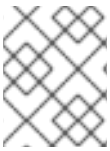
- 依存関係の指定: 公式の **cargo** ドキュメント。

2.8. プロジェクトドキュメントの構築



注記

rustdoc ではなく **cargo doc** の使用を考慮してください。このコマンド **cargo doc** は、**rustdoc** ユーティリティーを使用します。



注記

cargo doc パブリック関数、変数、メンバーに対してのみ、ドキュメントコメントを抽出します。

Rust コードには、ドキュメントへの抽出用のコメントを含めることができます。コメントは Markdown 言語に対応しています。

cargo ツールを使用してプロジェクトのドキュメンテーションを構築するには、プロジェクトディレクトリーに移動し、**cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo doc --no-deps'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo doc --no-deps
```

これにより、プロジェクトのソースコードにある特別なコメントから保存したドキュメントが抽出され、HTML 形式でドキュメントが書き込まれます。

- サードパーティーライブラリーなど、生成されたドキュメントの依存関係を含める **--no-deps** オプションを省略します。
- ブラウザーで生成されたドキュメントを開く **--open** オプションを追加します。

例2.7 プロジェクトドキュメントの構築

この例では、例2.6「プロジェクトへの依存関係の追加および cargo で構築」 に従って依存関係で Rust プロジェクト **helloworld** を正常にビルドしていることを前提としています。

ディレクトリー **helloworld** に移動し、以下のソースコードが含まれるファイル **src/main.rs** を編集します。

```
//! This is a hello-world program.
extern crate time;

/// Prints a greeting to `stdout`.
pub fn print_output() {
    println!("Hello, world!");
    println!("Time is: {}", time::now().rfc822());
}

/// The program entry point.
fn main() {
    print_output();
}
```

このコードには、パブリック関数 **print_output()** が含まれるようになりました。**helloworld** プログラム、**print_output()** 関数、および **main()** 関数にはドキュメントコメントがあります。

cargo doc コマンドを実行し、プロジェクトのドキュメンテーションを構築します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo doc --no-deps'
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo doc --no-deps
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
```

インストールされていない場合は、**ツリー** ツール (デフォルトの Red Hat Enterprise Linux リポジトリーで利用できます) をインストールします。

```
# yum install tree
```

結果を確認します。

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
...
    └── doc
...
        ├── helloworld
        │   ├── fn.print_output.html
        │   ├── index.html
        │   ├── print_output.v.html
        │   └── sidebar-items.js
...
        └── src
            └── helloworld
                └── main.rs.html

12 directories, 32 files
```

cargo はプロジェクトのドキュメンテーションを構築します。このドキュメントを表示するには、ブラウザー **target/doc/helloworld/index.html** でファイルを開きます。生成されたドキュメントには、公開されていないため、**main()** 関数に関する記述は含まれません。

--no-deps オプションを指定せずに **cargo doc** コマンドを実行して、依存関係ライブラリーの **time** および **libc** を含むプロジェクトのドキュメントを構築します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo doc'
 Documenting libc v0.2.32
 Documenting time v0.1.38
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 3.41 secs
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo doc
 Documenting libc v0.2.32
 Documenting time v0.1.38
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 3.41 secs
```

以下の **tree** コマンドを使用して、フォルダー構造を確認します。

```
$ tree
...
92 directories, 11804 files
$ ls -d target/doc/*/
target/doc/helloworld/  target/doc/implementors/  target/doc/libc/  target/doc/src/  target/doc/time/
```

作成されるドキュメントは、依存関係ライブラリーの **time** および **libc** にも対応し、それぞれ **target/doc/** ディレクトリーに別のサブディレクトリーとして存在します。

関連資料
cargo doc ツールおよびその機能の詳しい説明は、本書の対象外となっています。詳細は、以下に記載のドキュメントを参照してください。

- 公式 Rust プログラミング言語のドキュメントの 便利なドキュメントコメント

2.9. プロジェクト依存関係のベンダー

プロジェクト依存関係のベンダーとは、オフライン再分配および再分散のための依存関係のローカルコピーを作成することを意味します。ベンダーの依存関係は、インターネットに接続せずに **cargo** ビルドツールで使用することができます。

依存関係をベンダーする **cargo vendor** コマンドは、**cargo** プラグイン **cargo-vendor** により提供されます。

cargo-vendor 0.1.23 をインストールするには、次のコマンドを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
# yum install rust-toolset-1.35-cargo-vendor
```

- Red Hat Enterprise Linux 8 の場合:

```
# dnf install cargo-vendor
```

cargo プロジェクトのベンダーの依存関係には、プロジェクトディレクトリーに移動し、以下のように **cargo** ツールを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo vendor'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo vendor
```

これによりディレクトリー **vendor** が作成され、このディレクトリーへのすべての依存関係のソースがダウンロードされます。追加の設定手順はコマンドラインに表示されます。

cargo vendor コマンドは、プラットフォームに依存しない結果の依存関係を収集します。潜在的なすべてのターゲットプラットフォームの依存関係クレートがダウンロードされます。

重要

この **cargo vendor** コマンドは、**cargo** ツールの実験的かつ非実験的なプラグインです。

例2.8 プロジェクト依存関係のベンダー

この例では、例2.6「プロジェクトへの依存関係の追加および cargo で構築」 に従って依存関係で Rust プロジェクト **helloworld** を正常にビルドしていることを前提としています。

ディレクトリー **helloworld** に移動し、**cargo vendor** コマンドを実行して依存関係のあるプロジェクトをベンダーします。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo vendor'
 Downloading kernel32-sys v0.2.2
 Downloading redox_syscall v0.1.31
 Downloading winapi-build v0.1.1
 Downloading winapi v0.2.8
   Vendoring kernel32-sys v0.2.2 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/kernel32-sys-0.2.2) to vendor/kernel32-sys
   Vendoring libc v0.2.32 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/libc-0.2.32) to vendor/libc
   Vendoring redox_syscall v0.1.31 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/redox_syscall-0.1.31) to vendor/redox_syscall
   Vendoring time v0.1.38 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/time-0.1.38) to vendor/time
   Vendoring winapi v0.2.8 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/winapi-0.2.8) to vendor/winapi
   Vendoring winapi-build v0.1.1 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/winapi-build-0.1.1) to vendor/winapi-build
To use vendored sources, add this to your .cargo/config for this project:

    [source.crates-io]
    replace-with = "vendored-sources"

    [source.vendored-sources]
    directory = "/home/vslavik/helloworld/vendor"
```

- Red Hat Enterprise Linux 8 の場合:

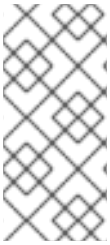
```
$ cargo vendor
```

結果を確認します。

```
$ ls
Cargo.lock  Cargo.toml  src  target  vendor
$ tree vendor
vendor
├── kernel32-sys
│   ├── build.rs
│   ├── Cargo.toml
│   ├── README.md
│   └── src
│       └── lib.rs
├── libc
│   ├── appveyor.yml
│   ├── Cargo.toml
...
75 directories, 319 files
```

vendor ディレクトリーには、**helloworld** プログラムを構築するために必要なすべての依存関係 crates のコピーが含まれます。Windows オペレーティングシステムでプロジェクトをビルドするための Crates は、Red Hat Enterprise Linux でこのコマンドを実行してもベンダーが提供されていま

す。



注記

tree ツールは、デフォルトの Red Hat Enterprise Linux リポジトリーから利用できます。インストールするには、以下を行います。

```
# yum install tree
```

2.10. 関連資料

cargo ツールおよびその機能の詳しい説明は、本書の対象外となっています。詳細は、以下に記載のドキュメントを参照してください。

Cargo ドキュメント

- **cargo**(1): **cargo** ツールの man ページは、その使用方法の詳細情報を提供します。Rust Toolset に含まれるバージョンの man ページを表示するには、次のコマンドを実行します。
  - Red Hat Enterprise Linux 7 の場合:

    ```
    $ scl enable rust-toolset-1.35 'man cargo'
    ```

  - Red Hat Enterprise Linux 8 の場合:

    ```
    $ man cargo
    ```

- **Rust の Package Manager、Cargo** HTML ブックは、パッケージとして提供されています。
  - Red Hat Enterprise Linux 7 の場合:

    ```
    # yum install rust-toolset-1.35-cargo-doc
    ```

    HTML は以下で利用可能です。**/opt/rh/rust-toolset-1.35/usr/share/doc/cargo/html/index.html**

  - Red Hat Enterprise Linux 8 の場合:

    ```
    # yum install cargo-doc
    ```

    HTML は以下で利用可能です。**/usr/share/doc/cargo/html/index.html**

オンラインの Cargo ドキュメント

- 公式 Cargo ガイド

関連項目

- 1章*Rust Toolset*: Rust Toolset の概要およびそのシステムへのインストール方法の詳細

第3章 RUSTFMT

rustfmt ツールは、Rust ソースコードの自動フォーマットを提供します。

Rust Toolset には、**rustfmt 1.35.0** が同梱されています。

3.1. RUSTFMT のインストール

- Red Hat Enterprise Linux 7 の場合:

```
# yum install rust-toolset-1.35-rustfmt
```

- Red Hat Enterprise Linux 8 の場合:

```
# dnf install rustfmt
```

3.2. RUSTFMT をスタンドアロンツールとして使用

rust ソースファイルとそのすべての依存関係を **rustfmt** ツールでフォーマットするには、以下を実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'rustfmt source-file'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ rustfmt source-file
```

source-file を、ソースファイルへのパスに置き換えます。

デフォルトでは、**rustfmt** は詳細を表示したりバックアップを作成したりせずに、影響を受けるファイルを変更します。動作を変更するには、**--write-mode value** オプションを使用します。詳細は、**rustfmt** のヘルプメッセージを参照してください。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'rustfmt --help'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ rustfmt --help
```

また、**rustfmt** はファイルの代わりに標準入力を受け付け、出力を標準出力で提供します。

3.3. CARGO での RUSTFMT の使用

cargo crate のすべてのソースファイルをフォーマットするには、次のコマンドを実行します。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'cargo fmt'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ cargo fmt
```

rustfmt フォーマットオプションを変更するには、プロジェクトディレクトリーに設定ファイル **rustfmt.toml** を作成し、そこに設定を指定します。詳細は、**rustfmt** のヘルプメッセージを参照してください。

- Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'rustfmt --config-help'
```

- Red Hat Enterprise Linux 8 の場合:

```
$ rustfmt --config-help
```

3.4. 関連資料

- **rustfmt** のヘルプメッセージがあります。
  - Red Hat Enterprise Linux 7 の場合:

```
$ scl enable rust-toolset-1.35 'rustfmt --help'
$ scl enable rust-toolset-1.35 'rustfmt --config-help'
```

  - Red Hat Enterprise Linux 8 の場合:

```
$ rustfmt --help
$ rustfmt --config-help
```

- **Configurations.md** での **RustIfmt の設定**:
  - Red Hat Enterprise Linux 7 の場所:
    /opt/rh/rust-toolset-1.35/usr/share/doc/rust-toolset-1.35-rustfmt-1.35.0/Configurations.md
  - Red Hat Enterprise Linux 8 の場所:
    /usr/share/doc/rustfmt/Configurations.md

第4章 RHEL 7 用の RUST TOOLSET のあるコンテナーイメージ

Rust Toolset は、Red Hat Container Registry からダウンロードできるコンテナーイメージとして利用できます。

4.1. イメージの内容

devtools/rust-toolset-rhel7 イメージは、以下のパッケージに対応するコンテンツを提供します。

コンポーネント	バージョン	パッケージ
Rust	1.35.0	rust-toolset-1.35-rust
Cargo	1.35.0	rust-toolset-1.35-cargo
Vendor plug-in for Cargo	0.1.23	rust-toolset-1.35-cargo-vendor

4.2. イメージへのアクセス

devtools/rust-toolset-rhel7 イメージをプルするには、**root** で以下のコマンドを実行します。

```
# podman pull registry.access.redhat.com/devtools/rust-toolset-rhel7
```

4.3. 関連資料

- Rust Toolset コンテナーイメージ: Red Hat Container Catalog のエントリー
- Red Hat Software Collections コンテナーイメージの使用

第5章 RED HAT DEVELOPER TOOLS 2019.3 の RUST TOOLSET の変更点

本章では、以前のリリース以降の Rust Toolset の主な変更点を紹介します。

5.1. RUST

Go のバージョンが **1.31.1** から **1.35.0** に更新されました。以下は、主な変更点です。

- Rust 1.32 の変更点
  - **dbg!** マクロ: ファイル、行、および式値を使用して、出力デバッグが容易になります。
  - 最終的なモジュール改善: "use" ステートメントはより均一です。例: **enum** バリアントの直接インポートを許可します。
  - マクロの改善: マクロはリテラルトークンに一致でき、"?" 演算子が完全に 0 または 1 回繰り返しに一致するようになりました。
  - ライブラリーの安定化: 整数はバイトアレイへの変換をサポートします。
- Rust 1.33 の変更点：
  - **const fn** 改善: 定数関数の実装は、パターンのデストラクション、**let** バインディング、割り当てなどに対応するようになりました。
  - pinning: タイプレベルのメモリーは、メモリーのオブジェクトが移動しないことを保証します。
  - _ としてインポート: トレイト名なしで範囲におけるトレイトメソッドを、競合が発生しないようにして利用するための匿名インポート。
  - ライブラリーの安定化: 整数の **const** メソッドが多数あります。
- Rust 1.34 の変更点:
  - 代替の **cargo** レジストリー: パブリックの crates.io 以外のレジストリーの使用を許可し、相互に混在させることができます。
  - ドキュメントテストにおける **?**: **Try** 演算子は「return」タイプを指定する doc テストで使用できます。
  - **TryFrom** および **TryInto**: フォールト可能な変換方法を提供します。
  - **fn before_exec** が **unsafe fn pre_exec** を優先して非推奨となりました: 分岐した pre-exec 環境には追加の安全制約があります。これは、安全でない呼び出しとして示唆されます。
  - ライブラリーの安定化: ゼロ以外の署名された整数タイプ **successors** および **from_fn** イテレーターコンストラクター
- Rust 1.35 の変更点:
  - **Box<dyn Fn*>** に実装されている **Fn*** 閉鎖トレイト: **Box<dyn FnOnce>** の特に注目すべき特性で 、以前に抽象的に呼び出すことはできません。
  - 引数なしでの **dbg!()** の呼び出し: 1.32 から更新すると、引数がない場合に file:line のみを出力できるようになりました。

- ライブラリーの安定化: フロート copysign、範囲制約、RefCell map_split

5.2. CARGO

この **cargo** ツールがバージョン 1.31.0 から 1.35.0 に更新されました。

5.3. CARGO-VENDOR

cargo-vendor ツールのバージョンが 0.1.22 から 0.1.23 に更新されました。