Red Hat Enterprise Linux 7

開発者ガイド

Red Hat Enterprise Linux 7 のアプリケーション開発ツールの概要

Last Updated: 2019-07-28

Red Hat Enterprise Linux 7 開発者ガイド

Red Hat Enterprise Linux 7 のアプリケーション開発ツールの概要

Vladimír Slávik
Red Hat Customer Content Services
vslavik@redhat.com

法律上の通知

Copyright © 2019 Red Hat, Inc.

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

本書では、アプリケーション開発に最適なエンタープライズプラットフォームとして、Red Hat Enterprise Linux 7.5 を活用するためのさまざまな機能とユーティリティーについて説明します。

目次

前書き 8

パート I. 開発ワークステーションの設定 9

第1章 オペレーティングシステムのインストール 10
関連資料 10

第2章 アプリケーションのバージョンを管理するための設定 11
関連資料 11

第3章 C および C++ を使用してアプリケーションを開発するための設定 12
関連資料 12

第4章 アプリケーションをデバッグするための設定 13
関連資料 13

第5章 アプリケーションのパフォーマンスを測定するための設定 14
関連資料 14

第6章 JAVA を使用してアプリケーションを開発するための設定 15

第7章 PYTHON を使用してアプリケーションを開発するための設定 16
Red Hat Software Collections パッケージに対応する Python バージョン 16
関連資料 16

第8章 C# および .NET CORE を使用してアプリケーションを開発するための設定 17
関連資料 17

第9章 コンテナーアプリケーションの開発のための設定 18
関連資料 18

第10章 WEB アプリケーションの開発のための設定 19
Red Hat Software Collections パッケージに対応する Ruby on Rails 19
関連資料 19

パート II. アプリケーションでの他の開発者との共同作業 20

第11章 GIT の使用 21
インストールされているドキュメント 21
オンラインのドキュメント 21

パート III. ユーザーへのアプリケーションの公開 22

第12章 配信オプション 23
RPM パッケージ 23
Software Collections 23
コンテナー 23
関連資料 23

第13章 アプリケーションでのコンテナーの作成 24
前提条件 24
ステップ 24
関連資料 25

第14章 パッケージからのアプリケーションのコンテナー化 26
前提条件 26

ステップ 26
追加情報 26

パート IV. C または C++ アプリケーションの作成 27

第15章 GCC でのビルドコード 28
15.1. コード形式間の関係 28
前提条件 28
考えられるコード形式 28
GCC でのコード形式の処理 28
関連資料 29
15.2. ソースファイルのオブジェクトコードへのコンパイル 29
前提条件 29
ステップ 29
関連資料 29
15.3. GCC を使用した C および C++ アプリケーションのデバッグの有効化 29
GCC を使用したデバッグ情報の作成の有効化 29
関連資料 30
15.4. GCC でのコードの最適化 30
GCC でのコードの最適化 30
関連資料 31
15.5. GCC でのコードのハード化 31
リリースバージョンのオプション 31
開発オプション 31
関連資料 31
15.6. 実行可能なファイルを作成するためのコードのリンク 31
前提条件 31
ステップ 31
関連資料 32
15.7. 各種 RED HAT 製品との C++ の互換性 32
関連資料 32
15.8. 例: GCC での C プログラムの構築 33
前提条件 33
ステップ 33
関連資料 33
15.9. 例: GCC での C++ プログラムの構築 33
前提条件 33
ステップ 34

第16章 GCC でのライブラリーの使用 35
16.1. ライブラリーの命名規則 35
関連資料 35
16.2. 静的リンクおよび動的リンク 35
静的リンクおよび動的リンクの比較 35
静的リンクの理由 36
関連資料 37
16.3. GCC でのライブラリーの使用 37
ライブラリーを使用するコードのコンパイル 37
ライブラリーを使用するコードのリンク 37
1 つの手順でライブラリーを使用するコードをコンパイルとリンクする方法 38
関連資料 38
16.4. GCC での静的ライブラリーの使用 38
前提条件 38
ステップ 38

16.5. GCC での動的ライブラリーの使用 39
前提条件 39
プログラムの動的ライブラリーへのリンク 39
実行可能ファイルに保存された rpath の値を使用する方法 39
LD_LIBRARY_PATH 環境変数を使用する方法 40
ライブラリーのデフォルトのディレクトリーへの配置 40
16.6. GCC で静的および動的ライブラリー両方の使用 40
前提条件 40
はじめに 40
ファイルで静的ライブラリーを指定する方法 41
-Wl オプションの使用 41
関連資料 42

第17章 GCC でのライブラリーの作成 43
17.1. ライブラリーの命名規則 43
関連資料 43
17.2. SONAME のメカニズム 43
前提条件 43
問題の概要 43
soname のメカニズム 43
ファイルからの soname の読み込み 44
17.3. GCC での動的ライブラリーの作成 44
前提条件 44
ステップ 44
関連資料 45
17.4. GCC および AR での静的ライブラリーの作成 45
前提条件 45
ステップ 45
関連資料 46

第18章 MAKE でのさらなるコードの管理 47
18.1. GNU MAKE および MAKEFILE の概要 47
前提条件 47
GNU make 47
Makefile の詳細 47
一般的な Makefile 48
関連資料 48
18.2. 例: MAKEFILE の使用した C プログラムの構築 48
前提条件 48
ステップ 48
関連資料 49
18.3. MAKE のドキュメントリソース 49
インストールされているドキュメント 49
オンラインのドキュメント 50

第19章 C および C++ アプリケーション開発での ECLIPSE IDE の使用 51
C および C++ アプリケーション開発での Eclipse IDE の使用 51
関連資料 51

パート V. デバッグアプリケーション 52

第20章 実行中のアプリケーションのデバッグ 53
20.1. デバッグ情報を使用したデバッグの有効化 53
20.1.1. デバッグの情報 53

関連資料 53
20.1.2. GCC を使用した C および C++ アプリケーションのデバッグの有効化 53
GCC を使用したデバッグ情報の作成の有効化 53
関連資料 54
20.1.3. Debuginfo パッケージ 54
前提条件 54
Debuginfo パッケージ 54
20.1.4. GDB を使用したアプリケーションまたはライブラリー向けの debuginfo パッケージ取得 54
前提条件 54
手順 55
関連資料 55
20.1.5. 手動でのアプリケーションまたはライブラリー向けの debuginfo パッケージ取得 55
前提条件 55
手順 56
関連資料 57
20.2. GDB を使用したアプリケーションの内部状況の検証 57
20.2.1. GNU デバッガー (GDB) 57
GDB 機能 57
デバッグの要件 57
20.2.2. プロセスへの GDB のアタッチ 57
前提条件 58
GDB でのプログラムの起動 58
すでに実行中のプロセスへの GDB のアタッチ 58
すでに実行中のプロセスに実行中の GDB をアタッチする手順 58
関連資料 59
20.2.3. GDB でのプログラムコードの活用 59
前提条件 59
コードを活用するための GDB コマンド 59
関連資料 60
20.2.4. GDB でのプログラム内部値の表示 60
前提条件 60
プログラムの内部の状態を表示するための GDB コマンド 60
関連資料 61
20.2.5. 定義したコードの場所で実行を停止するための GDB ブレークポイントの使用 61
前提条件 61
GDB でのブレークポイントの使用 61
関連資料 62
20.2.6. データへのアクセスや変更が合った場合に実行を停止するための GDB ウォッチポイントの使用 62
前提条件 62
GDB でのウォッチポイントの章 62
関連資料 63
20.2.7. GDB でのフォーク用またはスレッド化されたプログラムのデバッグ 63
前提条件 63
GDB でのフォークされたプログラムのデバッグ 63
GDB でのスレッド化されたプログラムのデバッグ 64
関連資料 64
20.3. アプリケーションの対話の記録 64
20.3.1. アプリケーションの対話の記録に役立つツール 65
関連資料 66
20.3.2. strace でのアプリケーションのシステム呼び出し監視 66
前提条件 66
ステップ 66
関連資料 67

20.3.3. ltrace でのアプリケーションのライブラリー関数呼び出しの監視 67
前提条件 67
ステップ 67
関連資料 68
20.3.4. SystemTap でのアプリケーションのシステム呼び出し監視 69
前提条件 69
ステップ 69
関連資料 69
20.3.5. GDB を使用したアプリケーションシステム呼び出しの遮断 70
前提条件 70
GDB でのシステム呼び出しでプログラム実行の停止 70
関連資料 70
20.3.6. アプリケーションによるシグナルの処理を遮断するための GDB の使用 70
前提条件 70
GDB でのシグナル受信時のプログラム実行停止 71
関連資料 71

第21章 クラッシュしたアプリケーションのデバッグ . . . 72
21.1. コアダンプ 72
前提条件 72
説明 72
21.2. コアダンプでのアプリケーションのクラッシュの記録 72
ステップ 72
関連資料 73
21.3. コアダンプを使用したアプリケーションのクラッシュの状態の検証 73
前提条件 73
ステップ 73
関連資料 75
21.4. GCORE を使用したプロセスメモリーのダンプ 75
前提条件 75
ステップ 75
関連資料 76
21.5. GDB での保護されたプロセスメモリーのダンプ 76
前提条件 76
ステップ 76
関連資料 76

パート VI. パフォーマンスの監視 . . . 77

第22章 VALGRIND . . . 78
22.1. VALGRIND ツール 78
22.2. VALGRIND の使用 79
22.3. その他の情報 79

第23章 OPROFILE . . . 80
23.1. OPROFILE の使用 80
23.2. OPROFILE のドキュメント 82

第24章 SYSTEMTAP . . . 83
24.1. 追加情報 83

第25章 PERFORMANCE COUNTERS FOR LINUX (PCL) ツールおよび PERF . . . 84
25.1. PERF ツールコマンド 84
25.2. PERF の使用方法 84

付録A 改訂履歴 87

前書き

本書は、アプリケーション開発に最適なエンタープライズプラットフォームとして、Red Hat Enterprise Linux 7 を活用するためのさまざまな機能とユーティリティーについて説明します。

パート I. 開発ワークステーションの設定

Red Hat Enterprise Linux 7 は、カスタムアプリケーションの開発をサポートします。開発者がカスタムアプリケーションの開発ができるように、必要なツールやユーティリティーを使用して、システムを設定する必要があります。本章では、開発に関する最も一般的なユースケース、インストールする項目について紹介します。

第1章 オペレーティングシステムのインストール

特定の開発ニーズを設定する前に基盤のシステムを設定する必要があります。

1. **ワークステーション** などに Red Hat Enterprise Linux をインストールします。『Red Hat Enterprise Linux インストールガイド』の手順に従います。

2. インストール時には「ソフトウェアの選択」の内容に注意してください。**Development and Creative Workstation** のシステムプロファイルを選択して、開発のニーズに適したアドオンをインストールできるようにします。以下のセクションに適切なアドオンを記載します。セクションごとに、さまざまな種類の開発にフォーカスしています。

3. ドライバーなど、 Linux カーネルに密に連携するアプリケーションを開発する場合は、インストール時に、**kdump** で自動クラッシュダンプを有効化してください。

4. システムをインストールしたら、システムを登録し、必要なサブスクリプションを割り当てます。Red Hat Enterprise Linux システム管理ガイドの「システム登録およびサブスクリプション管理」を参照してください。
   各種開発に特化した以下のセクションでは、該当の開発を行うために割り当てる必要がある特定のサブスクリプションを記載します。

5. 最新の開発ツールやユーティリティーは、Red Hat Software Collections として入手できます。Red Hat Software Collections へのアクセス方法は、『Red Hat Software Collections Release Notes』の「Installation」を参照してください。

関連資料

- Red Hat Enterprise Linux インストールガイド - Subscription Manager
- Red Hat Subscription Management
- Red Hat Enterprise Linux 7 パッケージマニフェスト

第2章 アプリケーションのバージョンを管理するための設定

複数の開発者が関わるプロジェクトではすべて、効果的な改訂管理が必須になります。Red Hat Enterprise Linux は **Git** という分散型のバージョン管理システムが同梱され、配信されています。

1. インストール時に **開発ツール** アドオンを選択して、**Git** をインストールします。

2. または、システムのインストール後に、Red Hat Enterprise Linux のリポジトリーから **git** パッケージをインストールします。

   ```
   # yum install git
   ```

3. Red Hat がサポートする **Git** の最新版を取得するには、Red Hat Software Collections から **rh-git29** のコンポーネントをインストールします。

   ```
   # yum install rh-git29
   ```

4. **Git** コミットに関連付けるフルネームとメールアドレスを設定します。

   ```
   $ git config --global user.name "full name"
   $ git config --global user.email "email_address"
   ```

   full name と **email_address** は、実際の名前とメールアドレスに置き換えます。

5. **Git** で開始するデフォルトのテキストエディターを変更するには、**core.editor** の設定オプションの値を設定します。

   ```
   $ git config --global core.editor command
   ```

   command は、選択のテキストエディターの起動に使用するコマンドに置き換えます。

関連資料

- *11章Git の使用*
- 『Red Hat Software Collections 3.0 Relase Notes』:「4.5 Git」

第3章 C および C++ を使用してアプリケーションを開発するための設定

Red Hat Enterprise Linux は、完全にコンパイルされた C および C++ のプログラミング言語を使用すると、最適に開発をサポートできます。

1. インストール時に **開発ツール** および **デバッグツール** のアドオンを選択して、**GNU コンパイラーコレクション (GCC)** と **GNU デバッガー (GDB)** などの開発ツールをインストールします。

2. **GCC**、**GDB** および関連のツールの最新版は、Red Hat Developer Toolset ツールチェーンのコンポーネントとして入手できます。

```
# yum install devtoolset-7-toolchain
```

3. Red Hat Enterprise Linux リポジトリーには、C および C++ アプリケーションの開発に幅広く使用されるライブラリーが多数含まれます。**yum** パッケージマネージャーを使用して、アプリケーションに必要なライブラリーの開発パッケージをインストールします。

4. グラフィカルインターフェースベースの開発の場合は、**Eclipse** 統合の開発環境をインストールします。C および C++ 言語は直接サポートされます。**Eclipse** は、Red Hat Developer Tools の一部として利用できます。実際のインストール手順については、「Using Eclipse」を参照してください。

関連資料

- 『Red Hat Developer Toolset User Guide』:「List of Components」

第4章 アプリケーションをデバッグするための設定

Red Hat Enterprise Linux には、内部のアプリケーションの動作を分析してトラブルシューティングを行うためのデバッグおよびインストルメンテーションツールが複数搭載されています。

1. **デバッグツール** および **Desktop Debugging and Performance Tools** アドオンを選択して、**GNU Debugger (GDB)**、**Valgrind**、**SystemTap**、**ltrace**、**strace** などのツールをインストールします。

2. **GDB**、**Valgrind**、**SystemTap**、**strace** および **ltrace** の最新版については、Red Hat Developer Toolset をインストールしてください。これをインストールすると、**memstomp** もインストールされます。

```
# yum install devtoolset-7
```

3. **memstomp** ユーティリティーは、Red Hat Developer Toolset の一部としてのみ入手可能です。Developer Toolset 全体のインストールは望まれないが、**memstomp** が必要な場合は、Red Hat Developer Toolset からこのコンポーネントだけインストールしてください。

```
# yum install devtoolset-7-memstomp
```

4. **debuginfo-install** ツールを使用するには、**yum-utils** パッケージをインストールします。

```
# yum install yum-utils
```

5. Red Hat Enterprise Linux の一部として利用可能なアプリケーションやライブラリーをデバッグするには、**debuginfo-install** ツールを使用して、適切な debuginfo およびソースパッケージを Red Hat Enterprise Linux リポジトリーからインストールします。コアのダンプファイル分析についても、これは該当します。

6. **SystemTap** アプリケーションでで必要なカーネルの debuginfo およびソースパッケージをインストールします。『SystemTap ビギナーズガイド』の「SystemTap のインストール」 を参照してください。

7. カーネルダンプを取得するには、**kdump** をインストールして設定します。『Kernel Crash Dump Guide』の「Installing and Configuring kdump」 を参照してください。

8. **SELinux** ポリシーの設定が、適切なアプリケーションが通常の場合だけでなく、デバッグの状況でも実行できるようになっていることを確認します。『SELinux ユーザーおよび管理者のガイド』の「問題の修正」を参照してください。

関連資料

- 「デバッグ情報を使用したデバッグの有効化」

- 『SystemTap ビギナーズガイド』

第5章 アプリケーションのパフォーマンスを測定するための設定

Red Hat Enterprise Linux には、開発者がアプリケーションのパフォーマンス低下の原因を特定できるように支援するアプリケーションが複数含まれています。

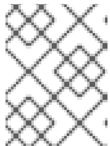
1. インストール時に **デバッグツール、開発ツール** および **パフォーマンスツール** のアドオンを選択して、**OProfile**, **perf** および **pcp** のツールをインストールします。

2. 一部の種類のパフォーマンス分析を可能にする **SystemTap** ツールと、パフォーマンス測定のモジュールを含む **Valgrind** をインストールします。

```
# yum install valgrind systemtap systemtap-runtime
```

3. 環境設定用の **SystemTap** ヘルパースクリプトを実行します。

```
# stap-prep
```



注記

このスクリプトを実行すると、サイズが非常に大きいカーネル debuginfo パッケージがインストールされます。

4. より頻繁にバージョンが更新される **SystemTap**、**OProfile** および **Valgrind** については、Red Hat Developer Toolset package **perftools** をインストールしてください。

```
# yum install devtoolset-7-perftools
```

関連資料

- 『Red Hat Developer Toolset User Guide』：「IV. Performance Monitoring Tools」

第6章 JAVA を使用してアプリケーションを開発するための設定

Red Hat Enterprise Linux は Java でのアプリケーションの開発をサポートします。

1. システムのインストール時に、**Java Platform** アドオンを選択して、デフォルトの Java バージョンとして OpenJDK をインストールします。
   または、『Red Hat JBoss Developer Studio Installation Guide』の「Installing OpenJDK on Red Hat Enterprise Linux」の説明に従い、OpenJDK を個別にインストールします。

2. 統合型のグラフィカル開発環境の場合は、Java 開発を幅広くサポートする Eclipse ベースの Red Hat JBoss Developer Studio をインストールしてください。『Red Hat JBoss Developer Studio Installation Guide』の説明に従ってください。

第7章 PYTHON を使用してアプリケーションを開発するための設定

Python 言語バージョン 2.7.5 は Red Hat Enterprise Linux の一部として提供されています。

1. **Python** インタープリターおよびライブラリーの新規バージョンは Red Hat Software Collections パッケージとして入手できます。以下の表を参照して希望のバージョンのパッケージをインストールしてください。

```
# yum install package
```

Red Hat Software Collections パッケージに対応する Python バージョン

バージョン	パッケージ
Python 2.7.13	**python27**
Python 3.4.2	**rh-python34**
Python 3.5.1	**rh-python35**
Python 3.6.3	**rh-python36**

2. **Python** 言語での開発をサポートする **Eclipse** 統合型開発環境をインストールします。**Eclipse** は Red Hat Developer Tools の一部として入手できます。実際のインストール手順は、「Using Eclipse」を参照してください。

関連資料

- Red Hat Software Collections Hello-World: 「Python」
- 「Red Hat Software Collections 3.0 Components」

第8章 C# および .NET CORE を使用してアプリケーションを開発するための設定

Red Hat は .NET Core ランタイム環境を対象とする C# 言語を使用したアプリケーションの開発をサポートします。

- ランタイム、コンパイラー、追加のツールを含む **.NET Core for Red Hat Enterprise Linux** をインストールします。『.NET Core Getting Started Guide』の「Install .NET Core」の説明に従ってください。

.NET Core 2.1 for Red Hat Enterprise Linux は、C# の他に、 ASP.NET、F#、および Visual Basic をサポートします。

関連資料

- 「.NET Core for Red Hat Enterprise Linux Overview」

第9章 コンテナーアプリケーションの開発のための設定

Red Hat は Red Hat Enterprise Linux、Red Hat OpenShift などの Red Hat 製品をもとにしたコンテナーアプリケーションの開発をサポートします。

- **Red Hat Container Development Kit** (CDK) をインストールします。CDK は、Red Hat Enterprise Linux 仮想マシンのシングルノード Red Hat OpenShift クラスターを提供します。『Red Hat Container Development Kit Getting Started Guide』の「Installing CDK」 を参照してください。

- また、**Red Hat Development Suite** も Java、C、および C++ でのコンテナーアプリケーションの開発に適しています。このスイートには、**Red Hat JBoss Developer Studio**、**OpenJDK**、**Red Hat Container Development Kit** や他のマイナーなコンポーネントが含まれています。**DevSuite** をインストールするには、『Red Hat Development Suite Installation Guide』の説明に従ってください。

関連資料

- Red Hat JBoss Developer Studio - 「Getting Started with Container and Cloud-based Development」

- 『Product Documentation for Red Hat Container Development Kit』

- Red Hat カスタマーポータル: 『Product Documentation for OpenShift Container Platform』

- Red Hat Enterprise Linux Atomic Host: 「Overview of Containers in Red Hat Systems」

第10章 WEB アプリケーションの開発のための設定

Red Hat Enterprise Linux は、Web アプリケーションの開発をサポートするだけでなく、デプロイメントのプラットフォームとしての役割を果たします。

Web 開発のトピックは幅が広いので、単純な説明だけでは理解は困難です。本セクションでは、Red Hat Enterprise Linux での Web アプリケーション開発パスとしてサポートされいる、最適な方法のみを説明します。

- 従来の Web アプリケーションの開発環境を設定するには、**Apache** Web サーバー、**PHP** ランタイム、**MariaDB** データベースサーバーおよびツールをインストールします。

```
# yum install httpd mariadb-server php-mysql php
```

  または、これらのアプリケーションのより新しいバージョンは、Red Hat Software Collections のコンポーネントとして入手できます。

```
# yum install httpd24 rh-mariadb102 rh-php71
```

- **Ruby on Rails** を使用した Web アプリケーションの開発には、以下の表に従い、希望のバージョンを含む Red Hat Software Collections のパッケージをインストールします。

```
# yum install package
```

Red Hat Software Collections パッケージに対応する Ruby on Rails

バージョン	パッケージ
Ruby on Rails 4.1.5	**rh-ror41**
Ruby on Rails 4.2.6	**rh-ror42**
Ruby on Rails 5.0.1	**rh-ror50**

関連資料

- 『Red Hat Software Collections 3.0 Release Notes』:「Ruby on Rails」
- Red Hat Software Collections: 「Hello World in Ruby」
- 「Advanced Linux Commands Cheat Sheet (setting up a LAMP stack)」 : Red Hat Developers Portal Cheat Sheet

パート II. アプリケーションでの他の開発者との共同作業

第11章 GIT の使用

複数の開発者が関わるすべてのプロジェクトでは、効果的な改訂管理が必須になります。改訂管理を効果的に行うことで、チーム内の開発者全員が組織的かつ規則的な方法でコードを作成、見直し、改訂、記録できるようになります。Red Hat Enterprise Linux 7.5 は、オープンソースのバージョン管理システム **Git** が含まれた状態で配信されています。

Git 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。この改訂管理システムに関する情報は、以下に記載のリソースを参照してください。

インストールされているドキュメント

- **Git** とチュートリアルの Linux の man ページ:

  ```
  $ man git
  $ man gittutorial
  $ man gittutorial-2
  ```

  Git コマンドの多くには、独自の man ページがあるのでご留意ください。

- **Git ユーザーマニュアル**: **Git** の HTML ドキュメントは **/usr/share/doc/git-1.8.3/user-manual.html** にあります。

オンラインのドキュメント

- Pro Git: オンライン版の**Pro Git** ブックでは、**Git**、コンセプト、用途が詳細にわたり説明されています。

- リファレンス: オンライン版の **Git** の Linux man ページ

パート III. ユーザーへのアプリケーションの公開

ユーザーにアプリケーションを公開する方法は複数存在します。本ガイドでは、最も一般的な方法を説明します。

- アプリケーションを RPM にパッケージ化
- アプリケーションをソフトウェアコレクションにパッケージ化
- アプリケーションをコンテナーにパッケージ化

第12章 配信オプション

Red Hat Enterprise Linux は、他社アプリケーションを 3 種類の方法で配信します。

RPM パッケージ

RPM パッケージは、ソフトウェアの配信/インストールするための従来の方式です。

- 複数のツールおよび広く普及したナレッジが含まれる成熟した技術
- アプリケーションはシステムの一部としてインストールされる
- インストールツールは依存関係の解決を大きく支援する
- パッケージのバージョン 1 つしかインストールできないので、複数のアプリケーションバージョンのインストールが困難である

RPM パッケージを作成する方法は、『RPM Packaging Guide』の「Packaging Software」の説明に従うようにしてください。

Software Collections

Software Collection は、アプリケーションの別バージョン用に特別に用意された RPM パッケージです。

- Red Hat が使用/サポートするパッケージ方法
- RPM パッケージメカニズム上に構築される
- 複数のアプリケーションバージョンを一度にインストールできる

詳しい情報は、『Red Hat Software Collections Packaging Guide』の「1.2 What Are Software Collections?」を参照してください。

Software Collection パッケージを作成する方法は、『Red Hat Software Collections Packaging Guide』の「Packaging Software Collections」を参照してください。

コンテナー

Docker 形式のコンテナーは、軽量に仮想化する方法の 1 つです。

- アプリケーションは、複数の個別バージョンおよびインスタンスとして存在できる
- RPM パッケージおよび Software Collection から簡単に用意できる
- システムとの対話が正確に制御できる
- アプリケーションの分離でセキュリティーが強化される
- コンテナーアプリケーションまたはそのコンポーネントで複数のインスタンスのオーケストレーションが可能になる

関連資料

- 『Red Hat Software Collections Packaging Guide 』:「1.2 What Are Software Collections?」

第13章 アプリケーションでのコンテナーの作成

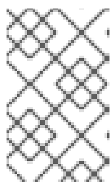
以下のセクションでは、Docker 形式のコンテナーイメージをローカルで構築したアプリケーションから作成する方法について説明します。デプロイメントにオーケストレーションを使用する場合には、アプリケーションをコンテナーとして提供すると有益です。または、効果的にコンテナー化することで、依存関係の矛盾が解決されます。

前提条件

- コンテナーを理解していること
- アプリケーションをソースからローカルで構築しておくこと

ステップ

1. 使用するベースイメージを決定します。



注記

Red Hat は、基盤として Red Hat Enterprise Linux を使用するベースイメージから開始することを推奨します。詳細情報は、「Base Image in the Red Hat Container Catalog」を参照してください。

2. ワークスペース用のディレクトリーを作成します。
3. アプリケーションが必要なファイルをすべて含むディレクトリーとして、アプリケーションを用意します。このディレクトリーをワークスペースディレクトリー内に配置します。
4. コンテナーの作成に必要な手順を説明する Dockerfile を記述します。
コンテンツの追加、デフォルトの実行コマンドの設定、必要なポートの開放および他の機能の追加など、Dockerfile の作成方法については、「Dockerfile Reference」を参照してください。

my-program/ ディレクトリーを含む最小限の Dockerfile の例:

```
FROM registry.access.redhat.com/rhel7
USER root
ADD my-program/ .
```

この Dockerfile をワークスペースディレクトリーに配置します。

5. Dockerfile からコンテナーイメージを構築します。

```
# docker build .
(...)
Successfully built container-id
```

この手順では、新規作成されたコンテナーイメージの **container-id** をメモするようにしてください。

6. イメージにタグを追加して、コンテナーイメージの保存先のレジストリーを識別します。「Getting Started with Containers: Tagging Images」を参照してください。

```
# docker tag container-id registry:port/name
```

container-id は、以前の手順の出力に表示された値に置き換えます。

registry は、イメージをプッシュするレジストリーのアドレスに、**port** はレジストリーのポートに (必要に応じて省略)、**name** はイメージの名前に置き換えます。

たとえば、イメージの名前が **myimage** のローカルシステムで **docker-distribution** サービスを使用してレジストリーを実行する場合には、タグ **localhost:5000/myimage** を使用することで、対象のイメージがレジストリーに配置する準備が整います。

7. イメージの使用を希望する人が後でレジストリーからプルできるように、対象イメージをレジストリーにプッシュします。

```
# docker push registry:port/name
```

タグの部分を、以前の手順で使用した値と同じものに置き換えます。

独自の Docker レジストリーを実行する方法は、「Getting Started with Containers – Working with Docker registries」を参照してください。

関連資料

- OpenShift Container Platform:「Creating Images」
- Red Hat Enterprise Linux Atomic Host:「コンテナー開発の推奨プラクティス」
- 「Dockerfile Reference」
- Docker ドキュメント:「Get Started, Part 2: Containers」
- Red Hat Enterprise Linux Atomic Host:「コンテナーの使用ガイド」
- 「Base Images」: Red Hat Container Catalog listing

第14章 パッケージからのアプリケーションのコンテナー化

複数の理由から、RPM にパッケージ化されたアプリケーションをコンテナーとして配信すると有益な場合があります。

前提条件

- コンテナーを理解していること
- 1 つ以上の RPM パッケージとしてパッケージ化されたアプリケーション

ステップ

RPM パッケージのアプリケーションをコンテナー化するには、『コンテナー使用ガイド』の「Docker イメージの作成」を参照してください。

追加情報

- OpenShift Container Platform:「Creating Images」
- Red Hat Enterprise Linux Atomic Host:「コンテナーの使用ガイド」
- Red Hat Enterprise Linux Atomic Host の製品ドキュメント
- Docker ドキュメント:「Get Started, Part 2: Containers」
- Docker ドキュメント:「Dockerfile reference」
- 「Base Images」: Red Hat Container Catalog listing

パート IV. C または C++ アプリケーションの作成

Red Hat では C および C++ 言語を使用してアプリケーションを構築するために、複数のツールをご用意しています。本書の以下の箇所に、最も一般的な開発タスクの一部を記載しています。

第15章 GCC でのビルドコード

本章では、ソースコードを実行可能なコードに変換する必要のある状況を扱います。

15.1. コード形式間の関係

前提条件

- コンパイルとリンクのコンセプトを理解していること

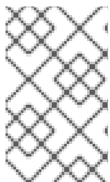
考えられるコード形式
C および C++ 言語を使用する場合は、コード形式が 3 つあります。

- C または C++ 言語で記述された **ソースコード**。プレーンテキストファイルとして公開します。これらのファイルは通常、**.c**、**.cc**、**.cpp**、**.h**、**.hpp**、**.i**、**.inc** などの拡張子を使用します。サポートされる拡張子およびその解釈にかんする全一覧は、gcc の man ページを参照してください。

  ```
  $ man gcc
  ```

- **コンパイラー** でソースコードを**コンパイルして**作成する **オブジェクトコード**。これは中間形式です。
  オブジェクトコードファイルは、**.o** の拡張子を使用します。

- **リンカー**でオブジェクトコードを**リンクして**作成する **実行可能なコード**
  Linux アプリケーションの実行可能なファイルは、ファイル名の拡張子を使用しません。共有オブジェクト (ライブラリー) の実行可能なファイルは、**.so** のファイル名の拡張子を使用します。



注記

静的にリンクするためのライブラリーのアーカイブファイルも存在します。これはオブジェクトコードのバリアントで、**.a** ファイル名の拡張子を使用します。静的リンクは推奨されません。「静的リンクおよび動的リンク」を参照してください。

GCC でのコード形式の処理
ソースコードから実行可能なコードを生成するには、2 つの手順を実行してください。各手順では、異なるアプリケーションまたはツールが必要です。GCC は、コンパイラーおよびリンカーどちらにも、インテリジェントドライバーとして使用可能です。これにより、必要なアクションに **gcc** のコマンド 1 つだけで対応できます。GCC は自動的に必要なアクション (コンパイルおよびリンク) とそのシーケンスを選択します。

1. ソースファイルは、オブジェクトファイルにコンパイルされます。
2. オブジェクトファイルおよびライブラリーはリンクされます (以前にコンパイルしたソールも含む)。

ステップ 1 だけ、ステップ 2 だけ、ステップ 1 と 2 両方というように、GCC を実行することができます。これは、入力タイプや要求する出力タイプにより決定されます。

大規模なプロジェクトには、アクション毎に個別に GCC を実行するビルドシステムが必要なため、GCC が両方同時に実行できる場合でも 2 つの異なるアクションとしてコンパイルとリンクを実行するように検討するほうが良いでしょう。

関連資料

- 「ソースファイルのオブジェクトコードへのコンパイル」
- 「実行可能なファイルを作成するためのコードのリンク」

15.2. ソースファイルのオブジェクトコードへのコンパイル

実行可能ファイルから直接作成するのではなく、ソースファイルからオブジェクトコードファイルを作成するには、 GCC にオブジェクトコードファイルのみを出力として作成するように指示する必要があります。このアクションは、大規模なプロジェクトのビルドプロセスの基本操作となります。

前提条件

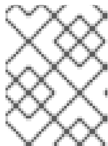
- C または C++ のソースコードファイル
- GCC をシステムにインストールしておくこと

ステップ

1. ソースコードファイルが含まれるディレクトリーに移動します。
2. **-c** オプションを指定して **gcc** を実行します。

```
$ gcc -c source.c another_source.c
```

   オブジェクトファイルは、オリジナルのソースコードファイルを反映したファイル名を使用して作成されます。 **source.c** は **source.o** になります。



注記

C++ ソースコードの場合は、標準 C++ ライブラリーの依存関係を都合よく処理するために、**gcc** コマンドを **g++** に置き換えてください。

関連資料

- 「GCC でのコードのハード化」
- 「GCC でのコードの最適化」
- 「例: GCC での C プログラムの構築」

15.3. GCC を使用した C および C++ アプリケーションのデバッグの有効化

デバッグの情報が大きい場合には、デフォルトでは実行可能ファイルは含まれません。GCC を使用した C および C++ アプリケーションのデバッグを有効化するには、コンパイラーに対して、ファイルを作成するように、明示的に指示する必要があります。

GCC を使用したデバッグ情報の作成の有効化

コードのコンパイルおよびリンク時には GCC でデバッグ情報の作成を有効化するには、**-g** オプションを使用します。

```
$ gcc ... -g ...
```

- コンパイラーとリンカーで最適化を行うと、実行可能なコードを、元のソースコードと関連付けることが難しくなります。変数の最適化、ループのアンロール、周りの操作へのマージなどが行われる可能性があります。デバッグの体験を向上するには、**-Og** オプションを指定して、最適化を設定することを考慮してください。ただし、最適化レベルを変更すると、実行可能なコードが変更され、バグを取り除くための実際の動作が変更される可能性があります。

- **-fcompare-debug** GCC オプションでは、GCC でコンパイルしたコードを、デバッグ情報を使用して (または、デバッグ情報を使用せずに) テストします。このテストでは、出力されたバイナリーファイル 2 つが同一であれば合格します。このテストを行うことで、実行可能なコードはデバッグオプションによる影響は受けないようにするだけでなく、デバッグコードにバグが含まれないようにします。**-fcompare-debug** オプションを使用するとコンパイレーションの時間が大幅に伸びます。このオプションに関する詳細情報は、GCC の man ページを参照してください。

関連資料

- 「デバッグ情報を使用したデバッグの有効化」
- GNU コンパイラーコレクション (GCC) の使用:「Options for Debugging Your Program」
- GDB でのデバッグ:「Debugging Information in Separate Files」
- GCC の man ページ:

```
$ man gcc
```

15.4. GCC でのコードの最適化

1 つのプログラムは、複数の機械語命令シーケンスに変換可能です。コンパイル時にコード分析用のリソースがより多く割り当てられると、より最適な結果が得られます。

GCC でのコードの最適化

GCC では、**-Olevel** オプションを使用して最適化レベルを設定できます。このオプションでは、**level** の部分に各種値を指定することができます。

レベル	説明
0	コンピレーション速度の最適化: コードの最適化なし (デフォルト)
1、2、3	コード実行速度を高めるための最適化の作業量増加
s	作成されるファイルサイズの最適化
fast	レベル 3 以上は、厳密な標準準拠を無視して追加の最適化を可能にします
g	デバッグ作業の最適化

リリースビルドの場合の最適化オプションは、**-O2** を推奨します。

開発中は、場合によってはプログラムやライブラリーのデバッグを行えるように **-Og** オプションのほうが便利です。バグによっては、特定の最適化レベルでのみ出現するので、リリースの最適化レベルでプログラムまたはライブラリーをテストするようにしてください。

GCC では、個別の最適化を有効にするオプションが多数含まれています。詳細情報は、以下の追加リソースを参照してください。

関連資料

- GNU コンパイラーコレクションの使用:「3.10 Options That Control Optimization」
- GCC の Linux man ページ:

```
$ man gcc
```

15.5. GCC でのコードのハード化

コンパイラーで、ソースコードをオブジェクトコードに変換する場合には、さまざまなチェックを追加して、一般的に悪用される状況などを回避し、セキュリティーを強化することができます。適切なコンパイラーオプションセットを選択して、ソースコードを変更せずに、よりセキュアなプログラムやライブラリーを制作することができます。

リリースバージョンのオプション
Red Hat Enterprise Linux を使用する開発者には、以下のオプション一覧が最小限必要なオプションとなります。

```
$ gcc ... -O2 -g -Wall -Wl,-z,now,-z,relro -fstack-protector-strong -D_FORTIFY_SOURCE=2 ...
```

- プログラムには、**-fPIE** および **-pie** の位置独立コードオプションを追加します。
- 動的にリンクされたライブラリーには、必須の **-fPIC** (位置独立コード) オプションを使用すると間接的にセキュリティーが強化されます。

開発オプション
開発時にセキュリティーの欠陥を検出する場合には、以下のオプションを推奨します。これらのオプションは、リリースバージョンのオプションと合わせて使用してください。

```
$ gcc ... -Walloc-zero -Walloca-larger-than -Wextra -Wformat-security -Wvla-larger-than ...
```

関連資料

- 『Defensive Coding Guide』
- 『Memory Error Detection Using GCC』: Red Hat 開発者のブログ投稿

15.6. 実行可能なファイルを作成するためのコードのリンク

C または C++ アプリケーション構築の最後の手順は、リンクです。リンクをすることで、オブジェクトファイルやライブラリーをすべて実行可能なファイルに統合します。

前提条件

- 1 つまたは複数のオブジェクトファイル
- GCC をシステムにインストールしておくこと

ステップ

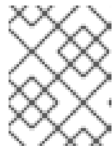
1. オブジェクトコードファイルを含むディレクトリーに移動します。

2. **gcc** を実行します。

```
$ gcc ... objfile.o another_object.o ... -o executable-file
```

executable-file という名前の実行可能なファイルが、指定したオブジェクトファイルとライブラリーをベースに作成されます。

追加のライブラリーををリンクするには、オブジェクトファイルの一覧の前に、必要なオプションを追加します。「*16章GCC でのライブラリーの使用*」を参照してください。



注記

C++ ソースコードの場合は、標準 C++ ライブラリーの依存関係を都合よく処理するために、**gcc** コマンドを **g++** に置き換えてください。

関連資料

- 「例: GCC での C プログラムの構築」
- *16章GCC でのライブラリーの使用*

15.7. 各種 RED HAT 製品との C++ の互換性

Red Hat エコシステムには、Red Hat Enterprise Linux および Red Hat Developer Toolset で提供される、GCC コンパイラーおよびリンカーのバージョンが複数含まれます。これらのバージョン間の C++ ABI の互換性は以下のとおりです。

- GCC 4.8 がベースになり、Red Hat Enterprise Linux 7 の一部として直接提供されている **システムコンパイラー** は、C++98 標準仕様 (C++03 としても知られています)、およびそのバリアント (GNU 拡張あり) へのコンパイルおよびリンクのみをサポートします。
- **-std=C++98** または **-std=gnu++98** オプションで明示的に構築された C++98 準拠のバイナリーまたはライブラリーは、使用するコンパイラーのバージョンにかかわらず、自由に組み合わせることができます。
- Red Hat Developer Toolset では、同じメジャーバージョンの GCC を使用して構築された適切なフラグで、すべての C++ オブジェクトがコンパイルされている場合に限り、C++11 および C++14 言語の使用および併用がサポートされます。
- Red Hat Developer Toolset および Red Hat Enterprise Linux ツールチェーンで構築された C++ ファイルをリンクする場合には、Red Hat Developer Toolset バージョンのコンパイラーとリンカーが推奨されます。
- Red Hat Enterprise Linux 6 および 7、そして Red Hat Developer Toolset バージョン 4.1 までのコンパイラーにおけるデフォルト設定は **-std=gnu++98** です。つまり、GNU 拡張がある C++98 です。
- Red Hat Developer Toolset 6、6.1、7、および 7.1 のコンパイラーにおけるデフォルト設定は **-std=gnu++14** です。つまり、GNU 拡張がある C++14 です。

関連資料

- 「Application Compatibility GUIDE」

- 「What gcc versions are available in Red Hat Enterprise Linux?」 - ナレッジベースソリューション
- 『Red Hat Developer Toolset User Guide』 - 「C++ Compatibility」

15.8. 例: GCC での C プログラムの構築

以下の例では、最小限の C プログラムのサンプルを構築する手順を説明します。

前提条件

- GCC の使用方法を理解していること

ステップ

1. **hello-c** ディレクトリーを作成して、そのディレクトリーに移動します。

```
$ mkdir hello-c
$ cd hello-c
```

2. 以下の内容を含む **hello.c** ファイルを作成します。

```
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

3. GCC でコードをコンパイルします。

```
$ gcc -c hello.c
```

オブジェクトファイル **hello.o** が作成されます。

4. オブジェクトファイルから作成した、実行可能なファイル **helloworld** をリンクします。

```
$ gcc hello.o -o helloworld
```

5. 作成された実行可能なファイルを実行します。

```
$ ./helloworld
Hello, World!
```

関連資料

- 「例: Makefile の使用した C プログラムの構築」

15.9. 例: GCC での C++ プログラムの構築

以下の例では、最小限の C++ プログラムのサンプルを構築する手順を説明します。

前提条件

- GCC の使用方法を理解していること

- GCC の使用方法を理解していること
- **gcc** と **g++** の相違点を理解していること

ステップ

1. **hello-cpp** ディレクトリーを作成して、そのディレクトリーに移動します。

   ```
   $ mkdir hello-cpp
   $ cd hello-cpp
   ```

2. 以下の内容を含む **hello.cpp** ファイルを作成します。

   ```
   #include <iostream>

   int main() {
     std::cout << "Hello, World!\n";
     return 0;
   }
   ```

3. **g++** でコードをコンパイルします。

   ```
   $ g++ -c hello.cpp
   ```

   オブジェクトファイル **hello.o** が作成されます。

4. オブジェクトファイルから作成した、実行可能なファイル **helloworld** をリンクします。

   ```
   $ g++ hello.o -o helloworld
   ```

5. 作成された実行可能なファイルを実行します。

   ```
   $ ./helloworld
   Hello, World!
   ```

第16章 GCC でのライブラリーの使用

この章では、コード内でのライブラリーの使用について説明します。

16.1. ライブラリーの命名規則

特別なファイルの命名規則をライブラリーに使用します。**foo** として知られるライブラリーは、**libfoo.so** または **libfoo.a** ファイルとして存在する必要があります。この規則は、リンクする GCC の入力オプションで、自動的に理解されますが、出力オプションでは理解されません。

- ライブラリーにリンクする場合には、**-lfoo** のように、**-l** オプションと **foo** の名前でしか、ライブラリーを指定することはできません。

```
$ gcc ... -lfoo ...
```

- ライブラリーの作成時には、**libfoo.so** または **libfoo.a** など、完全なファイル名を指定する必要があります。

関連資料

- 「soname のメカニズム」

16.2. 静的リンクおよび動的リンク

開発者には、完全にコンパイルした言語でアプリケーションを構築する際に、静的または動的のリンクを使用することが選択できます。本セクションでは、Red Hat Enterprise Linux で C 言語および C++ 言語した場合の違い (特にコンテキストについて) を説明します。つまり、Red Hat は、Red Hat Enterprise Linux 用のアプリケーションで静的リンクを使用することを推奨しません。

静的リンクおよび動的リンクの比較

静的リンクは、結果として作成された実行ファイルのライブラリーの一部になります。動的リンクは、このライブラリーを別々のファイルとして保持します。

動的リンクおよび静的リンクは、いくつかの方法で比較できます。

リソースの使用

静的リンクにでは、含まれるコードが増えるため、実行ファイルが大きくなります。ライブラリーから追加されるこのコードは、システム内の複数のプログラムで共有できるため、ランタイム時のファイルシステムの使用と、メモリーの使用が増えます。静的にリンクされた同じプログラムを実行する複数のプロセスは、依然としてそのコードを共有します。
一方、静的なアプリケーションでは、必要なランタイムの再配置がより少なくなるため、起動時が短くなり、必要なプライベートの RSS (Resident Set Size) メモリーも少なくなります。静的リンクに生成されるコードは、PIC (Position-Independent Code) により導入されるオーバーヘッドにより、動的リンクよりも効果的になります。

セキュリティー

ABI 互換性を提供する動的にリンクされたライブラリーは、そのライブラリーに依存する実行ファイルを変更せずに更新できます。これは、特に、Red Hat Enterprise Linux の一部として Red Hat がセキュリティー更新を提供するライブラリーで重要になります。このようなライブラリーには静的リンクを使用しないことが強く推奨されます。
さらに、ロードアドレスのランダム化などのセキュリティー対策は、静的にリンクされた実行ファイルで使用することはできません。これにより、アプリケーションのセキュリティーが低減します。

互換性

静的リンクは、オペレーティングシステムが提供するライブラリーのバージョンに依存しない実行ファイルを提供するように見えます。ただし、大概のライブラリーは、他のライブラリーに依存しています。静的リンクでは、この依存関係は柔軟性に欠けるとなり、結果として、前方互換性と後方互換性が失われます。静的リンクは、実行ファイルが構築されたシステムでのみ動作することが保証されます。

警告

GNU C ライブラリー (**glibc**) から静的にライブラリーをリンクするアプリケーションでは、システムに動的ライブラリーとして存在する **glibc** が依然として必要になります。さらに、アプリケーションのランタイム時に利用できる **glibc** の動的ライブラリーのバリアントは、アプリケーションをリンクしている間に存在するものとビット単位で同じバージョンである必要があります。そのため、静的リンクは、実行ファイルが構築されたシステムでのみ動作することが保証されます。

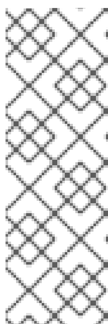
サポート範囲

Red Hat が提供するほとんどの静的ライブラリーは **Optional** チャンネルにあり、Red Hat ではサポートされていません。

機能

いくつかのライブラリー (特に GNU C ライブラリー (**glibc**)) は、静的にリンクすると提供する機能が少なくなります。
たとえば、静的にリンクすると、**glibc** は、スレッドと、同じプログラム内の **dlopen()** 関数への呼び出しの形式をサポートしません。

記載されたデメリットにより、静的リンクは、特にアプリケーション全体、**glibc** ライブラリー、および **libstdc++** ライブラリーに対しては、使用しないようにする必要があります。



注記

compat-glibc パッケージは Red Hat Enterprise Linux 7 に含まれますが、ランタイムパッケージではないため、何も実行する必要がありません。これは、リンクのためのヘッダーファイルとダミーのライブラリーが含まれる開発パッケージでしかありません。これにより、以前の Red Hat Enterprise Linux バージョンを実行するパッケージのコンパイルおよびリンクができるようになります (ヘッダーやライブラリーに **compat-gcc-*** を使用)。このパッケージの使用の詳細は、**rpm -qpi compat-glibc-*** を実行します。

静的リンクの理由

静的リンクは、次のようないくつかのケースでは合理的な選択が可能です。

- 動的リンクが使用できないライブラリー
- 完全に静的なリンクは、空の **chroot** 環境またはコンテナーでコードを実行するのに必要になる場合がありますが、**glibc-static** パッケージを使用した静的リンクは、Red Hat ではサポートされていません。

関連資料

- Red Hat Enterprise Linux 7: Application Compatibility GUIDE

16.3. GCC でのライブラリーの使用

ライブラリーは、プログラムで再利用可能なコードのパッケージです。C または C++ ライブラリーは、以下の 2 つの部分で構成されます。

- ライブラリーコード
- ヘッダーファイル

ライブラリーを使用するコードのコンパイル
ヘッダーファイルでは、ライブラリーで提供する関数や変数など、ライブラリーのインターフェースを記述します。コードをコンパイルする場合に、ヘッダーファイルからの情報が必要です。

通常、ライブラリーのヘッダーファイルは、アプリケーションのコードとは別のディレクトリーに配置されます。ヘッダーファイルの場所を GCC に指示するには、**-I** オプションを使用します。

```
$ gcc ... -Iinclude_path ...
```

include_path は、ヘッダーファイルのディレクトリーへの実際のパスに置き換えます。

-I オプションは、複数回使用して、ヘッダーファイルを含むディレクトリーを複数追加することができます。ヘッダーファイルを検索する場合は、**-I** オプションで表示順に、これらのディレクトリーが検索されます。

ライブラリーを使用するコードのリンク
実行可能なファイルをリンクする場合には、アプリケーションのオブジェクトコードおよび、ライブラリーのバイナリーコードの両方が利用できる状態でなければなりません。静的および動的ライブラリーのコードは、形式が異なります。

- 静的なライブラリーは、アーカイブファイルとして提供します。静的なライブラリーには、一連のオブジェクトファイルが含まれます。アーカイブファイルのファイル名の拡張子は、**.a** となっています。
- 動的なライブラリーは共有オブジェクトとして提供します。動的なライブラリーの形式は実行可能ファイルで、共有オブジェクトのファイル名の拡張子は **.so** となっています。

アーカイブファイルまたは共有オブジェクトファイルの場所を GCC に渡すには、**-L** オプションを使用します。

```
$ gcc ... -Llibrary_path -lfoo ...
```

library_path はライブラリーのディレクトリーへの実際のパスに置き換えます。

-l オプションは、複数回使用して、ディレクトリーを複数追加することができます。ライブラリーを検索する場合は、**-L** オプションで表示順に、これらのディレクトリーが検索されます。

オプションの指定順は重要です。対象のライブラリーがディレクトリーにリンクされていることが分からないと、GCC は、ライブラリー **foo** をリンクできません。そのため、**-L** オプションを使用して先にライブラリーディレクトリーを指定してから、**-l** オプションでライブラリーをリンクするようにしてください。

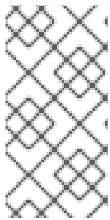
1 つの手順でライブラリーを使用するコードをコンパイルとリンクする方法
gcc コマンド 1 つでコードをコンパイルリンクできる場合には、上記のオプションを一度に使用します。

関連資料

- GNU コンパイラーコレクション (GCC) の使用:「3.15 Options for Directory Search」
- GNU コンパイラーコレクション (GCC) の使用:「3.14 Options for Linking」

16.4. GCC での静的ライブラリーの使用

静的ライブラリーは、オブジェクトファイルを含むアーカイブファイルとして提供します。リンク後には、これは、作成される実行ファイルの一部になります。

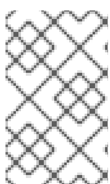


注記

Red Hat は、さまざまな理由から静的リンクを使用することは推奨していません。「静的リンクおよび動的リンク」を参照してください。特に Red Hat が提供するライブラリーに対してリンクする場合など、必要な場合にのみ、静的リンクを使用するようにしてください。

前提条件

- 「GCC がシステムにインストールされていること」
- 静的および動的リンクについて理解していること
- 有効なプログラムを構成するソースまたはオブジェクトファイルセット。静的ライブラリー **foo** だけを必要とします。
- **foo** ライブラリーが **libfoo.a** ファイルとして利用でき、動的リンクに **libfoo.so** ファイルが提供されていない。



注記

Red Hat Enterprise Linux の一部であるほとんどのライブラリーは、動的リンクでのみサポートされています。以下の手順は、動的リンクで **有効ではない** ライブラリーにのみ有効です。「GCC で静的および動的ライブラリー両方の使用」を参照してください。

ステップ

ソースとオブジェクトファイルからのプログラムをリンクするには、静的にリンクされたライブラリー **foo** (**libfoo.a** として検索可能) を追加します。

1. コードが含まれるディレクトリーに移動します。
2. **foo** ライブラリーのヘッダーで、プログラムソースファイルをコンパイルします。

   ```
   $ gcc ... -Iheader_path -c ...
   ```

   header_path は、**foo** ライブラリーのヘッダーファイルを含むディレクトリーへのパスに置き換えます。
3. プログラムを **foo** ライブラリーとリンクします。

```
$ gcc ... -Llibrary_path -lfoo ...
```

library_path は **libfoo.a** ファイルを含むディレクトリーへのパスに置き換えます。

4. あとでプログラムを実行するには、以下を実行するだけです。

```
$ ./program
```

注意

静的リンクに関連する **-static** GCC オプションでは、動的リンクを使用できません。代わりに、**-Wl,-Bstatic** オプションおよび **-Wl,-Bdynamic** オプションを使用して、リンカーの動作を正確に制御してください。「GCC で静的および動的ライブラリー両方の使用」 を参照してください。

16.5. GCC での動的ライブラリーの使用

動的ライブラリーは、スタンドアロンの実行可能ファイルとして提供します。このファイルは、リンク時、ランタイム時に必要です。これらのファイルは、アプリケーションの実行ファイルからは独立しています。

前提条件

- GCC をシステムにインストールしておくこと
- 有効なプログラムを構成するソースまたはオブジェクトファイルセット。動的ライブラリー **foo** だけを必要とします。
- **foo** ライブラリーは **libfoo.so** として利用できること

プログラムの動的ライブラリーへのリンク

動的ライブラリー **foo** にプログラムをリンクする手順:

```
$ gcc ... -Llibrary_path -lfoo ...
```

プログラムを動的ライブラリーにリンクすると、作成されるプログラムは常にランタイム時にライブラリーを読み込む必要があります。ライブラリーの場所を特定するオプションは 2 つあります。

- 実行可能ファイルに保存された **rpath** の値を使用する方法
- ランタイム時に **LD_LIBRARY_PATH** 変数を使用する方法

実行可能ファイルに保存された rpath の値を使用する方法

rpath は、リンク時に実行可能ファイルの一部として保存される特別な値です。その後、実行可能ファイルからプログラムを読み込む時に、ランタイムリンカーが **rpath** の値を使用してライブラリーファイルの場所を特定します。

GCC とリンクし、**library_path** のパスを **rpath** として保存します。

```
$ gcc ... -Llibrary_path -lfoo -Wl,-rpath=library_path ...
```

library_path のパスは、**libfoo.so** ファイルを含むディレクトリーを参照する必要があります。

注意

-Wl,-rpath= オプションのコンマの後に、スペースがあるので注意してください。

あとでプログラムを実行するには、以下を実行します。

```
$ ./program
```

LD_LIBRARY_PATH 環境変数を使用する方法
プログラムの実行可能ファイルに **rpath** がない場合は、ランタイムリンカーは **LD_LIBRARY_PATH** の環境変数を使用します。この変数の値は、共有オブジェクトがあるパスに合わせて、プログラム毎に変更する必要があります。

rpath セットがなく、ライブラリーが library_path パスにある状態で、プログラムを実行します。

```
$ export LD_LIBRARY_PATH=library_path:$LD_LIBRARY_PATH
$ ./program
```

rpath の値を空白にすると柔軟性が出てきますが、プログラムを実行するたびに、**LD_LIBRARY_PATH** の変数を設定する必要があります。

ライブラリーのデフォルトのディレクトリーへの配置
ランタイムのリンカー設定では、複数のディレクトリーを動的ライブラリーファイルのデフォルトの場所として指定します。このデフォルトの動作を使用するには、ライブラリーを適切なディレクトリーにコピーしてください。

動的リンカーの動作に関する全説明は、本書の対象外です。詳しい情報は、以下のリソースを参照してください。

- 動的リンカーの Linux man ページ:

  ```
  $ man ld.so
  ```

- **/etc/ld.so.conf** 設定ファイルの内容:

  ```
  $ cat /etc/ld.so.conf
  ```

- 追加設定なしに動的リンカーにより認識されるライブラリーのレポート (ディレクトリーを含む):

  ```
  $ ldconfig -v
  ```

16.6. GCC で静的および動的ライブラリー両方の使用

場合によっては、静的ライブラリーと動的ライブラリーの両方をリンクする必要があります。このような場合には、課題が浮上します。

前提条件

- 静的および動的リンクについて理解していること

はじめに
gcc は、動的および静的のライブラリーを認識します。**-lfoo** オプションが指定されている場合、**gcc**

は、動的にリンクされた **foo** ライブラリーを含む共有オブジェクト (**.so** ファイル) の場所をまず特定し、静的ライブラリーを含むアーカイブファイル (**.a**) を検索します。そのため、今回の検索の結果、以下の状態になります。

- 共有オブジェクトのみが見つかり、**gcc** がそのオブジェクトに動的にリンクする
- アーカイブファイルのみが見つかり、**gcc** がそのファイルに静的にリンクする
- 共有オブジェクトとアーカイブファイルの両方が見つかり、**gcc** はデフォルト設定のとおりに共有オブジェクトに動的にリンクする
- 共有オブジェクトもアーカイブファイルも見つからず、リンクに失敗する

このようなルールがあるので、リンクするために、静的または動的ライブラリーを選択する場合は **gcc** が検索可能なバージョンのみを指定するようにします。これは、**-Lpath** オプションで指定する場合に、静的/動的ライブラリーを含むディレクトリーを追加するか、追加しないかである程度制御することができます。

また、動的リンクがデフォルトの設定であるため、明示的にリンクを指定する必要があるのは、静的/動的の両バージョンのライブラリーを静的にリンクする必要がある場合のみです。考えられる方法は以下の 2 つです。

- **-l** オプションではなく、ファイルパスで静的ライブラリーを指定する
- **-Wl** を使用してリンカーに操作を渡す

ファイルで静的ライブラリーを指定する方法

通常、**gcc** は、**-lfoo** オプションを指定して、**foo** ライブラリーにリンクするように設定しますが、代わりに、ライブラリーを含む **libfoo.a** への完全パスを指定することも可能です。

```
$ gcc ... path/to/libfoo.a ...
```

ファイルの拡張子 **.a** から、**gcc** は、このファイルはプログラムとリンクするためのライブラリーであることを理解します。ただし、ライブラリーパスへの完全なパスを指定するのは柔軟な方法ではありません。

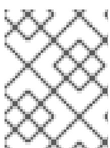
-Wl オプションの使用

gcc オプションの **-Wl** は、基盤のリンカーにオプションを渡すための特別なオプションです。このオプションの構文は、他の **gcc** オプションとは異なります。このオプションの後に、リンカーのオプションのコンマ区切りのリストを指定して、スペースで区切る **gcc** オプションと混同されないようにします。

gcc が使用する **ld** リンカーには、**-Bstatic** と **-Bdynamic** のオプションがあり、このオプションの後に来るライブラリーが静的、または動的にリンクすべきかどうかを指定します。**-Bstatic** とライブラリーをリカーに渡した後、以降のライブラリーを **-Bdynamic** オプションで動的にリンクするには、デフォルトの動的リンクの動きを手動で復元する必要があります。

プログラムをリンクするには、**まず** 静的に (**libfirst.a**) ライブラリーをリンクして、**ルギに** 動的に (**libsecond.so**) リンクします。

```
$ gcc ... -Wl,-Bstatic -lfirst -Wl,-Bdynamic -lsecond ...
```



注記

gcc は、デフォルトの **ld** 以外のリンカーを使用するように設定できます。**-Wl** オプションは、**gold** リンカーにも適用されます。

関連資料

- GNU コンパイラーコレクション (GCC) の使用: 「3.14 Options for Linking」
- binutils 2.27 のドキュメント: 「2.1 Command Line Options」

第17章 GCC でのライブラリーの作成

本章では、ライブラリーの作成手順と、Linux オペレーティングシステムで使用するために必要なライブラリーのコンセプトを説明します。

17.1. ライブラリーの命名規則

特別なファイルの命名規則をライブラリーに使用します。**foo** として知られるライブラリーは、**libfoo.so** または **libfoo.a** ファイルとして存在する必要があります。この規則は、リンクする GCC の入力オプションで、自動的に理解されますが、出力オプションでは理解されません。

- ライブラリーにリンクする場合には、**-lfoo** のように、**-l** オプションと **foo** の名前でしか、ライブラリーを指定することはできません。

```
$ gcc ... -lfoo ...
```

- ライブラリーの作成時には、**libfoo.so** または **libfoo.a** など、完全なファイル名を指定する必要があります。

関連資料

- 「soname のメカニズム」

17.2. SONAME のメカニズム

動的に読み込んだライブラリー (共有オブジェクト) は、**soname** と呼ばれるメカニズムを使用して、複数の互換性のあるライブラリーを管理します。

前提条件

- 「動的リンクとライブラリーについて理解していること」
- ABI の互換性のコンセプトを理解していること
- 「ライブラリーの命名規則を理解していること」
- シンボリックリンクについて理解していること

問題の概要
動的に読み込んだライブラリー (共有オブジェクト) は、独立した実行可能ファイルとして存在します。そのため、依存するアプリケーションを更新せずに。ライブラリーを更新することができます。ただし、このコンセプトでは、以下の問題が発生します。

- 実際のライブラリーバージョンを特定すること
- 同じライブラリーに対して複数のバージョンを存在させる必要があること
- 複数のバージョンでそれぞれ ABI の互換性を示すこと

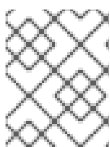
soname のメカニズム
この問題を解決するには、Linux では soname と呼ばれるメカニズムを使用します。

ライブラリー **foo** の **X.Y** バージョンは、バージョン番号 (**X**) が同じ値でマイナーバージョンが異なるバージョンと、ABI の互換性があります。互換性を確保してマイナーな変更を加えると、**Y** の数字が増加します。互換性がなくなるような、メジャーな変更を加えると、**X** の数字を増やします。

実際の **foo** ライブラリーバージョン **X.Y** は、**libfoo.so.x.y** ファイルとして存在します。ライブラリーファイルの中に、soname が **libfoo.so.x** の値として記録され、互換性を指定します。

アプリケーションを構築すると、リンカーは **libfoo.so** ファイルを検索して、ライブラリーを特定します。この名前のシンボリックリンクが存在し、実際のライブラリーファイルを参照する必要があります。次にリンカーは、ライブラリーファイルから soname を読み込み、アプリケーションの実行可能ファイルに記録します。最後に、リンカーにより、ファイル名でなく、soname を使用してライブラリー上で依存関係を宣言するように、アプリケーションが作成されます。

ランタイムの動的リンカーが実行前にアプリケーションをリンクすると、soname がアプリケーションの実行可能ファイルから読み込まれます。この soname は **libfoo.so.x** と呼ばれ、この名前のシンボリックリンクが存在し、実際のライブラリーファイルを参照する必要があります。これにより、soname が変更されないので、バージョンの **Y** の部分の有無にかかわらず、ライブラリーを読み込むことができるようになります。



注記

バージョン番号の **Y** の部分は、1 つの数字である必要はありません。また、ライブラリーによっては、名前がバージョンに組み込まれているものもあります。

ファイルからの soname の読み込み
somelibrary ライブラリーファイルの soname を表示します。

```
$ objdump -p somelibrary | grep SONAME
```

somelibrary は、検証するライブラリーの実際のファイル名に置き換えます。

17.3. GCC での動的ライブラリーの作成

ライブラリーを動的にリンクすると (共有オブジェクト)、コードを再利用して、リソースを確保し、ライブラリーコードの更新が簡単になり、セキュリティーの強化を図ることができます。このセクションでは、ソースから動的ライブラリーを構築してインストールする手順を説明します。

前提条件

- soname メカニズムの理解
- GCC をシステムにインストールしておくこと
- ライブラリーのソースコード

ステップ

1. ライブラリーソースのディレクトリーに移動します。
2. 位置独立コードオプション **-fPIC** でオブジェクトファイルに各ソースファイルをコンパイルします。

   ```
   $ gcc ... -c -fPIC some_file.c ...
   ```

   オブジェクトファイルはオリジナルのソースコードファイルと同じファイル名ですが、拡張子は **.o** となっています。
3. オブジェクトファイルから共有ライブラリーをリンクします。

```
$ gcc -shared -o libfoo.so.x.y -Wl,-soname,libfoo.so.x some_file.o ...
```

使用するメジャーバージョン番号は X で、マイナーバージョン番号は Y です。

4. **libfoo.so.x.y** ファイルを、システムの動的リンカーが検索できる適切な場所にコピーします。Red Hat Enterprise Linux では、ライブラリーのディレクトリーは **/usr/lib64** となります。

```
# cp libfoo.so.x.y /usr/lib64
```

このディレクトリー内のファイルを操作するには、root パーミッションが必要な点に注意してください。

5. soname メカニズムのシンボリックリンク構造を作成します。

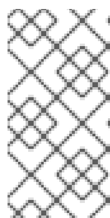
```
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

関連資料

- Linux ドキュメントプロジェクト「Program Library HOWTO の 3. Shared Libraries」

17.4. GCC および AR での静的ライブラリーの作成

オブジェクトファイルを特別なアーカイブファイルに変換して、静的にリンクするためにライブラリーを作成することが可能です。



注記

Red Hat は、セキュリティーの理由から静的リンクを使用することは推奨していません。特に Red Hat が提供するライブラリーに対してリンクする場合など、必要な場合にのみ、静的リンクを使用するようにしてください。「静的リンクおよび動的リンク」を参照してください。

前提条件

- GCC と binutils がシステムにインストールされていること
- 静的および動的リンクについて理解していること
- 関数を含むソースファイルをライブラリーとして共有すること

ステップ

1. GCC で仲介となるオブジェクトファイルを作成します。

```
$ gcc -c source_file.c ...
```

必要に応じて、さらにソースファイルを追加します。作成されるオブジェクトファイルはファイル名を共有しますが、ファイル名の拡張子は **.o** を使用 s います。

2. **binutils** パッケージからの **ar** ツールを使用して、オブジェクトファイルを静的ライブラリー (アーカイブ) に変換します

```
$ ar rcs libfoo.a source_file.o ...
```

libfoo.a ファイルが作成されます。

3. **nm** コマンドを使用して、作成されたアーカイブを検証します。

```
$ nm libfoo.a
```

4. 静的ライブラリーファイルを適切なディレクトリーにコピーします。

5. ライブラリーにリンクする場合には、GCC は自動的に **.a** のファイル名の拡張子 (ライブラリーが静的リンクのアーカイブであること) を認識します。

```
$ gcc ... -lfoo ...
```

関連資料

- **ar** ツールの Linux man ページ:

```
$ man ar
```

第18章 MAKE でのさらなるコードの管理

GNU の make ユーティリティー (通称 **make**) は、ソースファイルからの実行可能なファイルの生成を制御するツールです。**make** は自動的に、複雑なプログラムのどの部分が変更され、再度コンパイルする必要があるのかを判断します。**make** は Makefiles と呼ばれる設定ファイルを使用して、プログラムを構築する方法を制御します。

18.1. GNU MAKE および MAKEFILE の概要

特定のプロジェクトのソースファイルから使用可能な形式 (通常は実行可能ファイル) を作成するには、必要な手順を完了します。後で反復できるように、アクションとそのシーケンスを記録します。

Red Hat Enterprise Linux には、この目的用に設計されたビルドシステム、GNU **make** が含まれています。

前提条件

- コンパイルとリンクのコンセプトを理解していること

GNU make

GNU **make** はビルドプロセスの命令が含まれる Makefiles を読み込みます。Makefile には、特定のアクション (**recipe**) で特定の条件 (**target**) を満たす方法を記述する複数の **rules** が含まれています。ルールは、別のルールに階層的に依存することができます。

オプションを指定せずに **make** を実行すると、現在のディレクトリーで Makefile を検索し、デフォルトのターゲットに到達しようと試みます。実際の Makefile ファイル名は **Makefile**、**makefile** および **GNUmakefile** です。デフォルトのターゲットは、Makefile の内容で決まります。

Makefile の詳細

Makefiles は比較的単純な構文を使用して **variables** と **rules** を定義します。Makefiles は **target**と **recipe** で構成されます。ターゲットでは、ルールが実行された場合にどのような出力が表示されるのかを指定します。レシピの行は、TAB 文字で開始する必要があります。

通常、Makefile は、ソースファイルをコンパイルするためのルール、作成されるオブジェクトファイルをリンクするためにのルール、階層上部のエントリーポイントとしての役割を果たすターゲットで構成されます。

単一のファイル (**hello.c**) で構成される C プログラムを構築するには、以下の **Makefile** を検討してください。

```
all: hello

hello: hello.o
	gcc hello.o -o hello

hello.o: hello.c
	gcc -c hello.c -o hello.o
```

上記の例では、ターゲット **all** に到達するには、ファイル **hello** iが必要です。**hello** を取得するには、**hello.o** (**gcc** でリンク) が必要で、**hello.c** (**gcc** でコンパイル) をもとに作成します。

ターゲットの **all** は、ピリオド (.) で開始されない最初のターゲットであるため、デフォルトのターゲットとなっています。この **Makefile** が現在のディレクトリーに含まれている場合には、引数なしに **make** を実行するのは、**make all** するのと同じ作業です。

一般的な Makefile
より一般的な Makefile は、この手順を正規化する変数を使用し、ターゲット「clean」を追加して、ソースファイル以外のすべてを削除します。

```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

$(EXE): $(OBJ)
	$(CC) $(OBJ) -o $@

%.o: %.c
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -rf $(OBJ) $(EXE)
```

このような Makefile にソースファイルをさらに追加するには、SOURCE 変数が定義されている行に追加するだけです。

関連資料

- GNU make: 概要: 「2 An Introduction to Makefiles」
- *15章GCC でのビルドコード*

18.2. 例: MAKEFILE の使用した C プログラムの構築

以下の例の手順に従い、Makefile を使用して C プログラムを構築します。

前提条件

- 「Makefiles および **make** を理解していること」

ステップ

1. **hellomake** ディレクトリーを作成して、そのディレクトリーに移動します。

```
$ mkdir hellomake
$ cd hellomake
```

2. 以下のコンテンツで **hello.c** ファイルを作成します。

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

3. 以下の内容で **Makefile** ファイルを作成します。

```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

$(EXE): $(OBJ)
	$(CC) $(OBJ) -o $@

%.o: %.c
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -rf $(OBJ) $(EXE)
```

注意

Makefile レシピの行は、Tab 文字で開始する必要があります。ブラウザーから上記のテキストをコピーする場合は、スペースが挿入される可能性があるので、これは手動で変更してください。

4. **make** を実行します。

```
$ make
gcc -c -Wall hello.c -o hello.o
gcc hello.o -o hello
```

このコマンドで **hello** 実行可能ファイルが作成されます。

5. この実行可能ファイル **hello** を実行します。

```
$ ./hello
Hello, World!
```

6. Makefile のターゲット **clean** を実行して、作成されたファイルを削除します。

```
$ make clean
rm -rf hello.o hello
```

関連資料

- 「例: GCC での C プログラムの構築」
- 「例: GCC での C++ プログラムの構築」

18.3. MAKE のドキュメントリソース

make の詳細情報は、以下に記載のリソースを参照してください。

インストールされているドキュメント

- **man** および **info** ツールを使用して、お使いのシステムにインストールされている man ページ

と情報ページを表示します。

```
$ man make
$ info make
```

オンラインのドキュメント

- Free Software Foundation が提供する GNU Make Manual
- 『Red Hat Developer Toolset User Guide』の「Chapter 3. GNU make」

第19章 C および C++ アプリケーション開発での ECLIPSE IDE の使用

開発者によっては、複数のコマンドラインツールではなく、IDE の使用を好むがあります。Red Hat は、C および C++ 開発のサポートのある Eclipse IDE を提供します。

C および C++ アプリケーション開発での Eclipse IDE の使用

Eclipse IDE と、C および C++ アプリケーションの用途に関する詳細ジョイ法は、本書の対象外です。以下のリンクのリソースを参照してください。

関連資料

- 「Using Eclipse」
- Eclipse ドキュメント:『C/C++ Development User Guide』

パート V. デバッグアプリケーション

デバッグアプリケーションのトピックは幅広く、本セクションでは、複数の状況でデバッグを行うための最も一般的な手法を説明します。

第20章 実行中のアプリケーションのデバッグ

本章では、開発者が直接アクセスできるマシンで、必要に応じて何度でも起動でき、アプリケーションをデバッグする方法を紹介します。

20.1. デバッグ情報を使用したデバッグの有効化

アプリケーションやライブラリーをデバッグするには、デバッグ情報が必要です。以下のセクションでは、この情報を取得する方法を説明します。

20.1.1. デバッグの情報

実行可能なコードをデバッグする場合に、ツールやプログラマーは 2 種類の情報を使用して、バイナリーコードを理解することができます。

- ソースコードテキスト
- ソースコードテキストがバイナリーコードにどのように関連しているのかの説明

上記がデバッグ情報と呼ばれるものです。

Red Hat Enterprise Linux は、実行可能なバイナリー、共有ライブラリー、または debuginfo ファイルに ELF 形式を使用します。この ELF ファイルでは、DWARF 形式を使用してデバッグ情報を保持します。

DWARF シンボルは、**readelf -w file** コマンドを使用して読み込みます。

注意

STABS は UNIX で使用される場合もあります。STABS は、機能が少ない旧式の形式です。Red Hat は、STABS の使用は推奨していません。GCC および GDB は、STABS の実稼働および使用はベストエフォートでのみサポートされます。Valgrind および elfutils などの他のツールでは、STABS のサポートはありません。

関連資料

- 「DWARF Debugging Standard」

20.1.2. GCC を使用した C および C++ アプリケーションのデバッグの有効化

デバッグの情報が大きい場合には、デフォルトでは実行可能ファイルは含まれません。GCC を使用した C および C++ アプリケーションのデバッグを有効化するには、コンパイラーに対して、ファイルを作成するように、明示的に指示する必要があります。

GCC を使用したデバッグ情報の作成の有効化
コードのコンパイルおよびリンク時には GCC でデバッグ情報の作成を有効化するには、**-g** オプションを使用します。

```
$ gcc ... -g ...
```

- コンパイラーとリンカーで最適化を行うと、実行可能なコードを、元のソースコードと関連付けることが難しくなります。変数の最適化、ループのアンロール、周りの操作へのマージなどが行われる可能性があります。デバッグの体験を向上するには、**-Og** オプションを指定して、

最適化を設定することを考慮してください。ただし、最適化レベルを変更すると、実行可能なコードが変更され、バグを取り除くための実際の動作が変更される可能性があります。

- **-fcompare-debug** GCC オプションでは、GCC でコンパイルしたコードを、デバッグ情報を使用して (または、デバッグ情報を使用せずに) テストします。このテストでは、出力されたバイナリーファイル 2 つが同一であれば合格します。このテストを行うことで、実行可能なコードはデバッグオプションによる影響は受けないようにするだけでなく、デバッグコードにバグが含まれないようにします。**-fcompare-debug** オプションを使用するとコンピレーションの時間が大幅に伸びます。このオプションに関する詳細情報は、GCC の man ページを参照してください。

関連資料

- 「デバッグ情報を使用したデバッグの有効化」
- GNU コンパイラーコレクション (GCC) の使用: 「Options for Debugging Your Program」
- GDB でのデバッグ: 「Debugging Information in Separate Files」
- GCC の man ページ:

```
$ man gcc
```

20.1.3. Debuginfo パッケージ

Debuginfo パッケージには、プログラムとライブラリーのデバッグ情報と、デバッグソースコードが含まれます。

前提条件

- 「デバッグ情報を理解していること」

Debuginfo パッケージ
Red Hat Enterprise Linux リポジトリーのパッケージにインストールされているアプリケーションやライブラリーの場合は、別のチャンネルで提供されている別の **debuginfo** パッケージにとしてデバッグ情報とデバッグソースコードを取得することができます。debuginfo パッケージには **.debug** ファイルが含まれており、その中には、バイナリーパッケージのコンパイルに使用する DWARF debuginfo とソースファイルがあります。Debuginfo パッケージのコンテンツは、**/usr/lib/debug** ディレクトリーにインストールされます。

debuginfo パッケージでは、名前、バージョン、リリース、アーキテクチャーが同じバイナリーパッケージでのみ有効なデバッグ情報が提供されます。

- バイナリーパッケージ: **packagename-version-release.architecture.rpm**
- Debuginfo パッケージ: **packagename-debuginfo-version-release.architecture.rpm**

20.1.4. GDB を使用したアプリケーションまたはライブラリー向けの debuginfo パッケージ取得

GNU デバッガー (GDB) は、自動的に足りないデバッグ情報を認識して、パッケージ名を解決します。

前提条件

- デバッグするアプリケーションまたはライブラリーがシステムにインストールされていること

- システムに GDB がインストールされていること
- システムに **debuginfo-install** ツールがインストールされていること

手順

1. デバッグするアプリケーションまたはライブラリーにアタッチされている GDB を起動します。GDB は自動的に、足りないデバッグ情報を認識して、実行するコマンドを提案します。

```
$ gdb -q /bin/ls
Reading symbols from /usr/bin/ls...Reading symbols from /usr/bin/ls...(no debugging symbols found)...done.
(no debugging symbols found)...done.
Missing separate debuginfos, use: debuginfo-install coreutils-8.22-21.el7.x86_64
(gdb)
```

2. これ以上何もせずに、GDB を終了します。**q** を入力して、**Enter** を押します。

```
(gdb) q
```

3. GDB が提案するコマンドを実行して、必要な debuginfo パッケージをインストールします。

```
# debuginfo-install coreutils-8.22-21.el7.x86_64
```

アプリケーションまたはライブラリーの debuginfo パッケージをインストールすると、依存関係の debuginfo パッケージもすべてインストールされます。

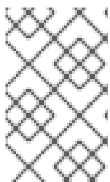
4. GDB が debuginfo パッケージを提案できない場合には、「手動でのアプリケーションまたはライブラリー向けの debuginfo パッケージ取得」の手順に従ってください。

関連資料

- 『Red Hat Developer Toolset User Guide』の「Installing Debugging Information」
- 「How can I download or install debuginfo packages for RHEL systems?」 : Red Hat ナレッジベースのソリューション

20.1.5. 手動でのアプリケーションまたはライブラリー向けの debuginfo パッケージ取得

実行可能ファイルの場所を特定して、そのファイルをインストールするパッケージを検索し、インストール用の debuginfo パッケージを手動で判断できます。



注記

GDB を使用してインストールするパッケージを決定すること を推奨します。GDB によりインストールするパッケージが提案されない場合にのみ、この手動の手順を使用してください。

前提条件

- アプリケーションまたはライブラリーをシステムにインストールしておくこと
- **debuginfo-install** ツールをシステムで利用できるようにしておくこと

手順

1. アプリケーションまたはライブラリーの実行可能ファイルを検索します。

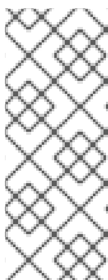
   a. **which** コマンドを使用して、アプリケーションファイルを検索します。

   ```
   $ which nautilus
   /usr/bin/nautilus
   ```

   b. **locate** コマンドを使用して、ライブラリーファイルを検索します。

   ```
   $ locate libz | grep so
   /usr/lib64/libz.so
   /usr/lib64/libz.so.1
   /usr/lib64/libz.so.1.2.7
   ```

   デバッグの理由がエラーメッセージに含まれていた場合には、ライブラリーのファイル名に同じ追加の数字が含まれる結果を選択します。分からない場合は、ライブラリーのファイル名に追加の数字が含まれていない結果を使用して、残りの手順を実行してください。

   注記

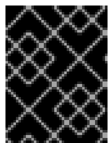
   locate コマンドは **mlocate** パッケージで提供されます。このパッケージをインストールして使用を有効にするには以下を実行します。

   ```
   # yum install mlocate
   # updatedb
   ```

2. ファイルパスを使用して、対象のファイルを提供するパッケージを検索します。

   ```
   # yum provides /usr/lib64/libz.so.1.2.7
   Loaded plugins: product-id, search-disabled-repos, subscription-manager
   zlib-1.2.7-17.el7.x86_64 : The compression and decompression library
   Repo        : @anaconda/7.4
   Matched from:
   Filename    : /usr/lib64/libz.so.1.2.7
   ```

   この出力では、**name-version.distribution.platform** の形式でパッケージ一覧が表示されます。**yum** の出力で表示されるバージョンは実際にインストールされているバージョンでない場合があるので、この手順では、パッケージの **名前** のみが重要です。

   重要

   この手順では結果が返されないので、どのパッケージがバイナリーファイル用のパッケージであるか、この手順が失敗したかの判断ができません。

3. **rpm** の低レベルのパッケージ管理ツールを使用して、どのパッケージバージョンがシステムにインストールされているのかを確認します。パッケージ名を引数として使用します。

   ```
   $ rpm -q zlib
   zlib-1.2.7-17.el7.x86_64
   ```

この出力では、**name-versiondistribution.platform** の形式でインストールされたパッケージの詳細が表示されます。

4. **debuginfo-install** ユーティリティーを使用して debuginfo パッケージをインストールします。このコマンドでは、前の手順で判断したパッケージ名などの情報を使用します。

```
# debuginfo-install zlib-1.2.7-17.el7.x86_64
```

アプリケーションまたはライブラリーの debuginfo パッケージをインストールすると、依存関係の debuginfo パッケージもすべてインストールされます。

関連資料

- 『Red Hat Developer Toolset User Guide』:「1.5.4. Installing Debugging Information」
- 「How can I download or install debuginfo packages for RHEL systems?」:ナレッジベースのアーティクル

20.2. GDB を使用したアプリケーションの内部状況の検証

アプリケーションが正しく機能しない理由を特定するには、実行を管理して、デバッガーの内部状況を検証します。このセクションでは、このタスクにおける GNU デバッガー (GDB) の使用方法を説明します。

20.2.1. GNU デバッガー (GDB)

デバッガーは、コード実行の制御や、コードの状態の検証を有効にするツールです。この機能は、プログラム内で何が発生しているのか、またその発生理由についての調査に使用します。

Red Hat Enterprise Linux には、コマンドラインユーザーインターフェースで機能を使用できる GNU デバッガー (GDB) が含まれます。

GDB へのグラフィカルフロントエンドについては、Eclipse 統合開発環境をインストールします。「Using Eclipse」を参照してください。

GDB 機能
単一の GDB セッションで、以下をデバッグできます。

- マルチスレッドやフォーク用のプログラムをデバッグ
- 一度に複数のプログラムをデバッグ
- TCP/IP ネットワーク接続経由で接続された **gdbserver** ユーティリティーを使用するコンテナー内または、リモートマシンのプログラムをデバッグ

デバッグの要件
実行可能なコードをデバッグするには、GDB では、適切なデバッグ情報が必要です。

- 自分で開発したプログラムの場合は、コードの構築時にデバッグ情報を作成できます。
- パッケージからインストールしたシステムプログラムの場合は、適切な debuginfo パッケージをインストールする必要があります。

20.2.2. プロセスへの GDB のアタッチ

プロセスを検証するには、GDB はプロセスに **アタッチ** する必要があります。

前提条件

- 「GDB をシステムにインストールしておくこと」

GDB でのプログラムの起動
プログラムがプロセスとして実行されていない場合は、GDB でプログラムを起動します。

```
$ gdb program
```

program は、ファイル名またはプログラムへのパスに置き換えます。

GDB は、プログラムの実行を開始するように設定します。ブレークポイントと **gdb** 環境を設定してから、**run** コマンドでプロセスの実行を開始するようにしてください。

すでに実行中のプロセスへの GDB のアタッチ
プロセスとしてすでに実行中のプログラムに GDB をアタッチするには、以下を実行します。

1. **ps** コマンドで、プロセス id (**pid**) を検索します。

   ```
   $ ps -C program -o pid h
    pid
   ```

   program は、ファイル名またはプログラムへのパスに置き換えます。

2. このプロセスに GDB をアタッチします。

   ```
   $ gdb -p pid
   ```

   pid は、**ps** の出力にある実際のプロセス id の数字に置き換えます。

すでに実行中のプロセスに実行中の GDB をアタッチする手順
すでに実行中のプロセスに実行中の GDB をアタッチします。

1. **shell** GDB コマンドを使用して、**ps** コマンドを実行し、プログラムのプロセス id (**pid**) を特定します。

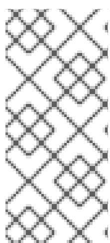
   ```
   (gdb) shell ps -C program -o pid h
    pid
   ```

   program は、ファイル名またはプログラムへのパスに置き換えます。

2. **attach** コマンドを使用して、GDB をプログラムにアタッチします。

   ```
   (gdb) attach pid
   ```

   pid は、**ps** の出力にある実際のプロセス id の数字に置き換えます。



注記

場合によっては GDB は適切な実行可能ファイルを検索できない可能性があります。**file** コマンドを使用して、パスを指定してください。

```
(gdb) file path/to/program
```

関連資料

- GDB でのデバッグ: 「2.1 Invoking GDB」
- GDB でのデバッグ: 「4.7 Debugging an Already-running Process」

20.2.3. GDB でのプログラムコードの活用

GDB デバッガーがプログラムにアタッチされたら、複数のコマンドを使用して、プログラムの実行を制御できます。

前提条件

- 「GDB をシステムにインストールしておくこと」
- 必要なデバッグ情報を利用できる状態にしておくこと
  - プログラムはコンパイルされ、デバッグ情報で構築されていること、または
  - 適切な debuginfo パッケージがインストールされていること
- GDB がデバッグするプログラムにアタッチされていること

コードを活用するための GDB コマンド

r (run)

プログラムの実行を開始します。引数を指定して **run** を実行すると、プログラムが通常通り開始されたかのように、その引数が実行可能ファイルに渡されます。ユーザーは通常、ブレークポイントを設定してから、このコマンドを実行します。

start

プログラムの実行を開始し、プログラムの主な機能の開始時点で停止します。引数を指定して **start** を実行すると、これらの引数は、プログラムが通常に起動したかのように、実行可能ファイルに渡されます。

c (continue)

現在の状態からプログラムの実行を継続します。プログラムの実行は、以下のいずれかが真になるまで継続されます。

- ブレークポイントに到達した場合
- 指定の条件を満たした場合
- プログラムからシグナルを受け取った場合
- エラーが発生した場合
- プログラムが終了された場合

n (next)

現在のソースファイルでコードが次の行に移動するまで、現在の状態からプログラムの実行を続けます。プログラムの実行は、以下の内容が真になるまで継続されます。

- ブレークポイントに到達した場合
- 指定の条件を満たした場合

- プログラムからシグナルを受け取った場合
- エラーが発生した場合
- プログラムが終了された場合

s (step)

step コマンドは、現在のソースファイルのコード行ごとに実行を停止させます。ただし、実行が **関数呼び出し** が含まれるソースの行で停止中の場合には、GDB は (関数呼び出しを実行するのではなく)、関数呼び出しに入る前に実行を停止します。

until location

location で指定したコードの場所に到達するまで、実行が継続されます。

fini (finish)

プログラムの実行を再開し、実行が関数から戻り値として返された時点で停止します。プログラムの実行は、以下のいずれかが真になるまで継続されます。

- ブレークポイントに到達した場合
- 指定の条件を満たした場合
- プログラムからシグナルを受け取った場合
- エラーが発生した場合
- プログラムが終了された場合

q (quit)

実行を中断して、GDB を終了します。

関連資料

- 「定義したコードの場所で実行を停止するための GDB ブレークポイントの使用」
- GDB でのデバッグ: 「4.2 Starting your Program」
- GDB でのデバッグ: 「5.2 Continuing and Stepping」

20.2.4. GDB でのプログラム内部値の表示

プログラムの内部変数の値を表示することは、プログラムが何を実行しているかを理解する場合に重要です。GDB には、内部変数の検証に使用可能なコマンドが複数含まれています。このセクションでは、これらのコマンドの中で最も有用なものを説明します。

前提条件

- GDB デバッガーを理解していること

プログラムの内部の状態を表示するための GDB コマンド

p (print)

指定した引数の値を表示します。引数には通常、単純な値 1 つや構造など、あらゆる複雑性の変数名を指定できます。引数には、プログラム変数やライブラリー関数での用途、テストするプログラムに定義する関数など、現在の言語で有効な表現も指定できます。

pretty-printer Python または Guile スクリプトを使用して GDB を拡張し、 **print** コマンドを使用して、(クラス、構造など) データ構造をカスタマイズ表示することができます。

bt (backtrace)

現在の実行ポイントに到達するために使用する関数呼び出しチェーンや、実行が中断されるまで使用する関数チェーンを表示します。これは、原因を特定しにくい、深刻なバグ (セグメント障害など) を調査する場合に便利です。
backtrace コマンドに **full** オプションを追加すると、ローカルの変数も表示されます。

frame filter Python スクリプトを使用して GDB を拡張し、 **bt** と **info frame** コマンドでデータをカスタマイズして表示することができます。**frame** は、単一の関数呼び出しに関連付けられたデータを参照します。

info

info コマンドは、様々な項目に関する情報を表示するための汎用コマンドです。このコマンドでは、説明を確認する項目をオプションとして指定できます。

- **info args** コマンドは、現在選択されているフレームの関数呼び出しオプションを表示します。
- **info locals** コマンドは、現在選択されているフレームにローカルの変数を表示します。

指定可能なオプションを一覧表示するには、GDB セッションで **help info** のコマンドを実行します。

```
(gdb) help info
```

l (list)

プログラムが停止したソースコードの行を表示します。このコマンドは、プログラムの実行が停止された場合のみ利用できます。**list** は、内部の状態を厳密に表示するコマンドではありませんが、プログラム実行の次のステップでどのような変更が内部状態に加えられるのかを理解することができます。

関連資料

- 「The GDB Python API」: Red Hat 開発者のブログ投稿
- GDB でのデバッグ: 「10.9 Pretty Printing」

20.2.5. 定義したコードの場所で実行を停止するための GDB ブレークポイントの使用

多くの場合、特定のコードの行に到達するまでプログラムを実行させると有益です。

前提条件

- GDB を理解していること

GDB でのブレークポイントの使用
ブレークポイントは、プログラムの実行を停止するように GDB に指示を出すためのマーカーです。ブレークポイントは一般的に、ソースコードの行と関連付けられており、ブレークポイントを配置するには、ソースファイルと行数を指定する必要があります。

- **ブレークポイントを配置する方法**:
  - ソースコードの **ファイル** 名と、そのファイルの **行** を指定します。

```
(gdb) br file:line
```

- file がない場合には、現在実行中のポイントのソースファイル名を使用します。

```
(gdb) br line
```

- または、関数名を使用して、起動時にブレークポイントを配置します。

```
(gdb) br function_name
```

- タスクを特定の回数反復すると、プログラムにエラーが発生する可能性があります。実行を停止するには、追加の **条件** を指定します。

```
(gdb) br file:line if condition
```

  condition は、C または C++ 言語の条件に置き換えます。 **file** と **line** は、上記と同様に、ファイル名および行数に置き換えます。

- 全ブレークポイントおよびウォッチポイントの状態を **検証** するには以下を実行します。

```
(gdb) info br
```

- **info br** の出力で表示された **数字** を使用して、ブレークポイントを **削除するには** 以下を実行します。

```
(gdb) delete number
```

- 指定の場所のブレークポイントを **削除** するには以下を実行します。

```
(gdb) clear file:line
```

関連資料

- GDB でのデバッグ: 「5.1 Breakpoints, Watchpoints, and Catchpoints」

20.2.6. データへのアクセスや変更が合った場合に実行を停止するための GDB ウォッチポイントの使用

多くの場合、特定のデータが変更されるまで、または特定のデータにアクセスがあるまで、プログラムを実行させるようにすると有益です。このセクションでは、一般的なコマンドについて説明します。

前提条件

- GDB の理解

GDB でのウォッチポイントの章
ウォッチポイントは、プログラム実行を停止するように GDB に指示を出すマーカーです。ウォッチポイントは、データと関連付けられており、ウォッチポイントを配置するには、変数を記述する表現、複数の変数、またはメモリーアドレスを指定する必要があります。

- データを **変更** (書き込み) するために、ウォッチポイントを **配置** します。

```
(gdb) watch expression
```

expression は、ウォッチの対象を記述する表現に置き換えます。変数の場合は、**expression** は変数名と同じです。

- データに **アクセス** (読み取り) するために、ウォッチポイントを **配置** します。

```
(gdb) rwatch expression
```

- **あらゆる** データにアクセス (読み取りおよび書き込み) するために、ウォッチポイントを **配置** します。

```
(gdb) awatch expression
```

- 全ウォッチポイントおよびブレークポイントの状態を **検証** するには以下を実行します。

```
(gdb) info br
```

- ウォッチポイントを **削除** するには以下を実行します。

```
(gdb) delete num
```

num オプションは、**info br** コマンドで報告された数字に置き換えます。

関連資料

- GDB でのデバッグ: 「5.1.2 Setting Watchpoints」

20.2.7. GDB でのフォーク用またはスレッド化されたプログラムのデバッグ

プログラムによっては、コードを並行実行するためにフォークまたはスレッドを使用するものがあります。複数同時に実行パスをデバッグするには、特別な留意点があります。

前提条件

- GDB デバッガーを理解していること
- フォークおよびスレッドプロセスのコンセプトを理解していること

GDB でのフォークされたプログラムのデバッグ
フォークとは、プログラム (**親**) により、独立したコピー (**子**) を作成する状況のことを指します。以下の設定およびコマンドを使用して、フォークを行う場合に GDB にどのような動作をさせるかを決定します。

- **follow-fork-mode** 設定で、フォークの後に GDB が親、子どちらに従うのかを制御します。

  set follow-fork-mode parent
  フォークの後に、親のプロセスをデバッグします。これはデフォルトです。

  set follow-fork-mode child
  フォークの後に子のプロセスをデバッグします。

  show follow-fork-mode
  follow-fork-mode の現在の設定を表示します。

- **set detach-on-fork** 設定では、GDB が (フォローしていない) 他のプロセスを制御するか、そのまま実行させるかを制御します。

set detach-on-fork on

フォローしていないプロセス (**follow-fork-mode** の値による) は切り離され、別個で実行されます。これはデフォルトです。

set detach-on-fork off

GDB は両プロセスを制御します。フォローしているプロセス (**follow-fork-mode** の値による) は通常通りにデバッグされ、他は一時停止されます。

show detach-on-fork

detach-on-fork の現在の設定を表示します。

GDB でのスレッド化されたプログラムのデバッグ
GDB には、個別のスレッドをデバッグし、そのスレッドを個別に操作および検査する機能があります。GDB を使用して、検証したスレッドのみを停止させるには、**set non-stop on** および **set target-async on** のコマンドを使用します。これらのコマンドは、**.gdbinit** ファイルに追加することができます。この機能をオンにすると、GDB でスレッドのデバッグを行う準備が整います。

GDB は current thread のコンセプトを使用します。デフォルトでは、コマンドは現在のスレッドのみに適用されます。

info threads

現在のスレッドを指す **id** と **gid** の数字を使用してスレッドの一覧を表示します。

thread id

id を指定して現在のスレッドとして、スレッドを設定します。

thread apply ids command

command コマンドを **ids** で表示されたスレッドすべてに適用します。**ids** オプションは、スペースで区切られたスレッド ID です。特別な値 **all** でこのコマンドを全スレッドに適用します。

break location thread id if condition

スレッド番号 **id** に対してのみ、特定の **condition** の特定 **location** で、ブレークポイントを設定します。

watch expression thread id

スレッド番号 **id** に対してのみ、**expression** で定義したウォッチポイントを設定します。

command&

コマンド **command** を実行して、すぐに gdb プロンプト **(gdb)** に戻りますが、バックグラウンドでコードの実行が続行されます。

interrupt

バックグラウンドでの実行が停止されます。

関連資料

- GDB でのデバッグ: 「4.10 Debugging Programs with Multiple Threads」
- GDB でのデバッグ: 「4.11 Debugging Forks」

20.3. アプリケーションの対話の記録

アプリケーションの実行可能コードは、オペレーティングシステムや共有ライブラリーのコードと対話します。これらの対話に関するアクティビティーログを記録すると、実際のアプリケーションコードをデバッグせずに、アプリケーションの動作を詳細にわたり知ることができます。また、アプリケーションの対話を分析すると、バグが発生した状況をピンポイントで特定しやすくなります。

20.3.1. アプリケーションの対話の記録に役立つツール

Red Hat Enterprise Linux には、アプリケーションの対話を分析するための複数のツールが含まれます。

strace

strace ツールは主に、アプリケーションが使用するシステム呼び出し (カーネルの関数) のログ記録を有効化します。

- **strace** は、パラメーターを解釈し、下層のカーネルコードのナレッジを結果として提供するので、**strace** 出力は詳細にわたり、呼び出しに関する詳しい情報を提供します。数字は、適切な定数名に変換され、ビット単位で統合されたフラグはフラグ一覧に拡張され、文字配列に対するポインターは実際の文字列を提供するために逆参照されます。最新のカーネル機能のサポートに欠ける場合があります。
- トレースされた呼び出しをフィルタリングして、取得するデータ量を減らすことができます。
- [command]`strace を使用するために、ログフィルターの設定以外に、特別な設定は必要ありません。
- **strace** でアプリケーションコードを追跡すると、アプリケーションの実行速度が大幅に遅くなるため、多くの場合、**strace** は、実稼働デプロイメントに適していません。代わりに、**ltrace** または SystemTap の使用を検討してください。
- Red Hat Developer Toolset で利用可能な **strace** のバージョンでは、システム呼び出しで異なる結果を出すことができるので、この機能は、デバッグに役立ちます。

ltrace

ltrace ツールは、アプリケーションの共有オブジェクト (動的ライブラリー) へのユーザー空間呼び出しをロギングできるようにします。

- **ltrace** は、ライブラリーへの呼び出しをトレースできるようにします。
- トレースされた呼び出しをフィルタリングして、取得するデータ量を減らすことができます。
- **ltrace** を使用するために、ログフィルターの設定以外に、特別な設定は必要ありません。
- **ltrace** は、軽量、高速で、**strace** の代わりとして使用できます。**strace** でカーネルの関数をトレースする代わりに、**ltrace** で **glibc** など、ライブラリー内の適切なインターフェースをトレースできます。
- **ltrace** は **strace** などの既知の呼び出しを処理しないので、ライブラリーの関数に渡す値の情報は提供されません。**ltrace** の出力には、未処理の数字やポインターだけが含まれます。**ltrace** 出力を解釈するには、出力に含まれるライブラリーの実際のインターフェースの宣言を確認する必要があります。

SystemTap

SystemTap は、Linux システム上で実行中のプロセスおよびカーネルアクティビティをプローブするための有用なインストルメンテーションプラットフォームです。SystemTap は、独自のスクリプト言語を使用してカスタムのイベントハンドラーをプログラミングします。

- **strace** や **ltrace** の使用と比較して、ロギングをスクリプト化すると、初期の設定フェーズでの作業量が増加しますが、スクリプト機能があると、SystemTap は、ログの生成以外の利便性が増します。

- SystemTap は、カーネルモジュールを作成、挿入することで機能します。SystemTap は効率的で、システムやアプリケーションの実行速度を大幅に下げることはありません。
- SystemTap には使用例が複数含まれています。

GDB

GNU デバッガーは主に、ロギングではなく、デバッグを行いますが、GNU デバッガーの機能の中で、アプリケーションの対話が主なアクティビティーとなるシナリオでも、有用なものがあります。

- GDB では、対話イベントを取得して、後に続く実行パスを即時にデバッグするように、都合よく組み合わせることができます。
- GDB は、他のツールで問題の状況を最初に特定してから、頻度の低いイベントや単一のイベントを分析するのが一番適しています。イベントの発生頻度が高い場合には、GDB の使用が非効率であったり、使用ができなくなったりします。

関連資料

- 『Red Hat Enterprise Linux SystemTap ビギナーズガイド』
- 『Red Hat Developer Toolset User Guide』

20.3.2. strace でのアプリケーションのシステム呼び出し監視

strace ツールでは、アプリケーションを実行するシステム (カーネル) 呼び出しの監視を有効化します。

前提条件

- システムに **strace** がインストールされていること

ステップ

1. 監視するシステム呼び出しを特定します。
2. 監視するプログラムが実行されていない場合は、**strace** を起動して、**プログラム** を指定します。

```
$ strace -fvttTyy -s 256 -e trace=call program
```

call は、表示されるシステム呼び出しに置き換えます。**-e trace=call** オプションは複数回使用できます。何も指定しない場合は、**strace** は全システム呼び出しの種類を表示します。詳しい情報は **strace(1)** の man ページを参照してください。

プログラムがすでに実行中の場合には、プロセス id (**pid**) を検索して、その id に **strace** をアタッチします。

```
$ ps -C program
(...)
$ strace -fvttTyy -s 256 -e trace=call -ppid
```

フォークしたプロセスまたはスレッドをトレースしない場合には、**-f** オプションは指定しないでください。

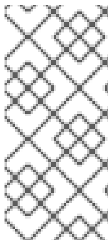
3. **strace** は、アプリケーションで作成したシステム呼び出しとその詳細を表示します。
多くの場合、アプリケーションとそのライブラリーは大量の呼び出しを作成し、これらのシステム呼び出しにフィルターが設定されていない場合には、**strace** の出力が直後に表示されます。

4. **strace** は、プログラムが終了すると、終了します。
トレースしたプログラムが終了する前にモニタリングを中断するには、**ctrl+C** を押してください。

    - **strace** でプログラムを起動した場合には、プログラムは **strace** とともに中断されます。
    - 実行中のプログラムに **strace** をアタッチした場合には、プログラムは **strace** とともに中断されます。

5. アプリケーションが実行したシステム呼び出しの一覧を分析します。

    - リソースのアクセスや可用性の問題は、エラーを返した呼び出しとしてログに表示されます。
    - システム呼び出しや呼び出しシーケンスのパターンに渡す値により、アプリケーションの動作の原因が分かります。
    - アプリケーションがクラッシュした場合に、重要な情報はおそらく、ログの最後に表示されます。
    - 出力には、不必要な情報が多く含まれていますが、より正確なフィルターを構築して、繰り返し手順を実行してください。



注記

出力を確認することも、ファイルに保存することもどちらも有用です。これには、**tee** コマンドを使用します。

```
$ strace ... |& tee your_log_file.log
```

関連資料

- **strace(1)** man ページ
- 「How do I use strace to trace system calls made by a command?」 : ナレッジベースアーティクル
- 『Red Hat Developer Toolset User Guide』 - 「strace」

20.3.3. ltrace でのアプリケーションのライブラリー関数呼び出しの監視

ltrace ツールは、ライブラリー (共有オブジェクト) で利用可能な関数に対するアプリケーションの呼び出しを監視できます。

前提条件

- システムに **ltrace** がインストールされていること

ステップ

1. 可能であれば、対象のライブラリーおよび関数を特定します。

2. 監視するプログラムが実行されていない場合には、**ltrace** を起動して、**プログラム** を指定します。

```
$ ltrace -f -l library -e function program
```

-e および **-l** のオプションを使用して、出力をフィルタリングします。

- function として表示する関数名を提示します。**-e function** オプションは複数回使用できます。何も指定しない場合は、**ltrace** は全関数への呼び出しを表示します。
- 関数を指定するのではなく、**-l library** オプションでライブラリー全体を指定することができます。このオプションは、**-e function** オプションと同様に動作します。

詳細情報は、**ltrace**(1)_ man ページを参照してください。

プログラムがすでに実行中の場合は、プロセス id (**pid**) を検索して、その id を指定して **ltrace** を実行します。

```
$ ps -C program
(...)
$ ltrace ... -ppid
```

フォークしたプロセスまたはスレッドをトレースしない場合には、**-f** オプションは指定しないでください。

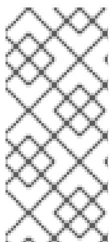
3. **ltrace** はアプリケーションのライブラリー呼び出しを表示します。
多くの場合、アプリケーションは大量の呼び出しを作成し、フィルターが設定されていない場合には、**ltrace** の出力がすぐに表示されます。

4. **ltrace** は、プログラムが終了すると、終了します。
トレースしたプログラムが終了する前にモニタリングを中断するには、**ctrl+C** を押してください。

- **ltrace** でプログラムを起動した場合には、プログラムは **ltrace** とともに中断されます。
- **ltrace** をすでに実行中のプログラムにアタッチした場合には、プログラムは **ltrace** とともに中断されます。

5. アプリケーションが実行したライブラリー呼び出しの一覧を分析します。

- アプリケーションがクラッシュした場合に、重要な情報はおそらく、ログの最後に表示されます。
- 出力には、不必要な情報が多く含まれていますが、より正確なフィルターを構築して、繰り返し手順を実行してください。



注記

出力を確認することも、ファイルに保存することもどちらも有用です。これには、**tee** コマンドを使用します。

```
$ ltrace ... |& tee your_log_file.log
```

関連資料

- strace(1) man ページ

- strace(1) man ページ
- 『Red Hat Developer Toolset User Guide』 - 「ltrace」

20.3.4. SystemTap でのアプリケーションのシステム呼び出し監視

SystemTap ツールでは、カーネルイベントにカスタムイベントハンドラーを登録できるようになります。**strace** と比較すると、使いにくいですが、効率性が高く、より複雑な処理ロジックを使用できます。

前提条件

- システムに SystemTap がインストールされていること

ステップ

1. 以下の内容を含む **my_script.stp** ファイルを作成します。

```
probe begin
{
  printf("waiting for syscalls of process %d \n", target())
}

probe syscall.*
{
  if (pid() == target())
    printf("%s(%s)\n", name, argstr)
}

probe process.end
{
  if (pid() == target())
    exit()
}
```

2. 監視するプロセスのプロセス ID (**pid**) を検索します。

```
$ ps -aux
```

3. スクリプトで SystemTap を実行します。

```
# stap my_script.stp -x pid
```

pid には、プロセス id を指定します。

スクリプトはカーネルモジュールにコンパイルされてから読み込まれるため、コマンドを入力してから出力が表示されるまで若干の遅延があります。

4. プロセスでシステム呼び出しが実行されると、呼び出し名とパラメーターがターミナルに出力されます。
5. プロセスが中断された場合や、**Ctrl+C** が押された場合に、スクリプトは終了します。

関連資料

- 『SystemTap ビギナーズガイド』

- SystemTap タップセットリファレンス
- **strace** 機能に似た SystemTap スクリプトは **/usr/share/systemtap/examples/process/strace.stp** にあります。

20.3.5. GDB を使用したアプリケーションシステム呼び出しの遮断

GDB は、プログラムの実行中に発生するさまざまな状況において実行を指定できます。プログラムがシステム呼び出しを実行時に実行を停止するには、GDB **catchpoint** を使用します。

前提条件

- GDB ブレークポイントを理解していること
- GDB がプログラムにアタッチされていること

GDB でのシステム呼び出しでプログラム実行の停止

1. キャッチポイントを設定します。

   ```
   (gdb) catch syscall syscall-name
   ```

   catch syscall コマンドは、プログラムによりシステム呼び出しが行われた時に実行を停止する特別なブレークポイントを設定します。

   syscall-name オプショは、呼び出し名を指定します。様々なシステム呼び出しに対して複数のキャッチポイントを指定することができます。**syscall-name** オプションに何も指定しない場合には、システム呼び出しがあると、GDB が停止してしまいます。

2. プログラムにより、実行が開始されていない場合には、開始してください。

   ```
   (gdb) r
   ```

   プログラムの実行が一時停止されているだけの場合は、再開してください。

   ```
   (gdb) c
   ```

3. GDB は、指定のシステム呼び出しがプログラムにより実行された後に実行を一時停止します。

関連資料

- 「GDB でのプログラム内部値の表示」
- 「GDB でのプログラムコードの活用」
- GDB でのデバッグ: 「Setting Watchpoints」

20.3.6. アプリケーションによるシグナルの処理を遮断するための GDB の使用

GDB は、プログラムの実行中に発生するさまざまな状況において実行を指定できます。プログラムがオペレーティングシステムからシグナルを受信した時に実行を停止するには、GDB **catchpoint** を使用します。

前提条件

- GDB ブレークポイントを理解していること

- GDB がプログラムにアタッチされていること

GDB でのシグナル受信時のプログラム実行停止

1. キャッチポイントを設定します。

```
(gdb) catch signal signal-type
```

   catch signal コマンドは、プログラムでシグナルを受信した場合に、実行を一時停止する特別なブレークポイントを設定します。**signal-type** オプションは、シグナルのタイプを指定します。すべてのシグナルを取得するには、特別な値 **'all'** を使用します。

2. プログラムにより、実行が開始されていない場合には、開始してください。

```
(gdb) r
```

   プログラムの実行が一時停止されているだけの場合は、再開してください。

```
(gdb) c
```

3. GDB は、プログラムが指定のシグナルを受信すると実行を停止します。

関連資料

- 「GDB でのプログラム内部値の表示」
- 「GDB でのプログラムコードの活用」
- GDB でのデバッグ: 「5.1.3 Setting Catchpoints」

第21章 クラッシュしたアプリケーションのデバッグ

アプリケーションを直接デバッグできない場合があります。そのような場合には、中断時のアプリケーションの情報を収集して、後で分析することができます。

21.1. コアダンプ

以下のセクションでは、**コアダンプ** の概要、その用途について説明します。

前提条件

- デバッグ情報を理解していること

説明

コアダンプは、アプリケーション機能が停止した時点のアプリケーションメモリーの一部を ELF 形式で保存したコピーのことです。コアダンプには、アプリケーションの内部変数、スタックすべてが含まれ、アプリケーションの最終的な状態を検査することができます。適切な実行ファイルやデバッグ情報が指定されている場合には、実行中のプログラムを分析するのに似た方法で、デバッガーを使用してコアダンプファイルを分析することができます。

Linux オペレーティングシステムのカーネルは、コアダンプの自動記録が有効化去れている場合には、コアダンプを自動的に記録できます。また、実行中のアプリケーションにシグナルを送り、実際の状態に関係なくコアダンプを生成することも可能です。

警告

コアダンプの生成機能に影響を与える制限が何点かあります。

21.2. コアダンプでのアプリケーションのクラッシュの記録

アプリケーションのクラッシュを記録するには、コアダンプの保存内容を設定し、システムに関する情報を追加します。

ステップ

1. コアダンプを有効にします。**/etc/systemd/systemd.conf** ファイルを編集し、**DefaultLimitCORE** を含む行を以下のように変更します。

```
DefaultLimitCORE=infinity
```

2. システムを再起動します。

```
# shutdown -r now
```

3. コアダンプサイズの制限を削除します。

```
# ulimit -c unlimited
```

この変更を元に戻すには、**unlimited** の代わりに 0 を指定してコマンドを実行します。

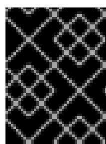
4. アプリケーションがクラッシュすると、コアダンプが生成されます。コアダンプのデフォルトの場所は、クラッシュ発生時の作業ディレクトリーとなっています。

5. システムに関する追加情報を指定するには、SOS レポートを作成します。

```
# sosreport
```

これにより、設定ファイルのコピーなど、お使いのシステムに関する情報が含まれる tar アーカイブが作成されます。

6. デバッグを行うコンピューターに、コアダンプと SOS レポートを移動します。また、実行可能ファイルがある場合にはそのファイルも移動します。



重要

実行可能ファイルが不明な場合は、あとでコアファイルを分析するときに特定します。

7. オプション: コアダンプと SOS レポートに移動後には、ディスク容量を開放するために削除します。

関連資料

- 「アプリケーションがクラッシュまたはセグメンテーション違反が発生した時にコアファイルのダンプを有効にする」: ナレッジベースアーティクル
- 「Red Hat Enterprise Linux 4.6 以降における sosreport の役割と取得方法」 : ナレッジベースアーティクル

21.3. コアダンプを使用したアプリケーションのクラッシュの状態の検証

前提条件

- コアダンプファイルおよび sosreport
- GDB および elfutils がシステムにインストールされていること

ステップ

1. クラッシュが発生した実行可能ファイルを特定するには、コアダンプファイルを指定して、**eu-unstrip** コマンドを実行します。

```
$ eu-unstrip -n --core=./core.9814
0x400000+0x207000 2818b2009547f780a5639c904cded443e564973e@0x400284
/usr/bin/sleep /usr/lib/debug/bin/sleep.debug [exe]
0x7fff26fff000+0x1000 1e2a683b7d877576970e4275d41a6aaec280795e@0x7fff26fff340 . -
linux-vdso.so.1
0x35e7e00000+0x3b6000
374add1ead31ccb449779bc7ee7877de3377e5ad@0x35e7e00280 /usr/lib64/libc-2.14.90.so
/usr/lib/debug/lib64/libc-2.14.90.so.debug libc.so.6
0x35e7a00000+0x224000
3ed9e61c2b7e707ce244816335776afa2ad0307d@0x35e7a001d8 /usr/lib64/ld-2.14.90.so
/usr/lib/debug/lib64/ld-2.14.90.so.debug ld-linux-x86-64.so.2
```

出力には、行ごとに各モジュールの詳細が、スペースで区切られて表示されます。以下の順番で情報が表示されます。

1. モジュールがマッピングされているメモリーアドレス
2. モジュールの build-id および、その id が見つかったメモリーの場所
3. 不明な場合は **-**、モジュールがファイルから読み込まれていない場合には **.** として表示される、モジュールの実行可能ファイル名
4. 実行可能ファイル自体に含まれている場合には **.**、ファイルが存在しない場合には **-** のファイル名で表示されるデバッグ情報のソース
5. 主要なモジュールの共有ライブラリー名 (soname) または **[exe]**

この例では、重要な情報は **/usr/bin/sleep** と、**[exe]** テキストに含まれる行の build-id **2818b2009547f780a5639c904cded443e564973e** です。この情報を使用して、コアダンプの分析に日宇町な実行可能ファイルを特定することができます。

2. クラッシュした実行可能ファイルを取得します。
   - 可能であれば、クラッシュが発生したシステムからコピーします。コアファイルから取得したファイル名を使用します。
   - または、お使いのシステムで、同じ実行可能ファイルを使用します。Red Hat Enterprise Linux でビルドされた実行可能ファイルはそれぞれ、一意の buid-id の値と中期が含まれます。build-id がローカルで利用可能で適切な実行可能ファイルであるこかどうかを判断します。

     ```
     $ eu-readelf -n executable_file
     ```

     この情報を使用して、リモートシステムの実行可能ファイルとローカルコピーを一致させます。ローカルファイルの build-id と、コアダンプに表示される build-id は必ず一致させてください。
   - 最後に、アプリケーションが RPM パッケージからインストールされた場合には、パッケージから実行可能ファイルを取得できます。**sosreport** の出力を使用して、必要なパッケージと合致するバージョンを検索します。
3. 実行可能ファイルで使用する共有ライブラリーを取得します。実行可能ファイルと同じ手順を使用します。
4. アプリケーションがパッケージとして配信されている場合は、GDB で実行可能ファイルを読み込み、足りない debuginfo パッケージに関するヒントを表示します。詳細は、「GDB を使用したアプリケーションまたはライブラリー向けの debuginfo パッケージ取得」を参照してください。
5. コアファイルを詳細にわたり検証するには、GDB で実行可能ファイルとコアダンプファイルを読み込みます。

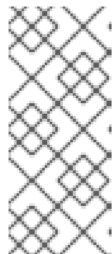
   ```
   $ gdb -e executable_file -c core_file
   ```

   足りないファイルやデバッグ情報に関するさらなるメッセージで、デバッグセッションで足りない情報の特定に役立ちます。必要に応じて、以前の手順に戻ります。

   アプリケーションのデバッグ情報がパッケージのではなく、ファイルとして提供される場合には、**symbol-file** コマンドを使用して、このファイルを GDB で読み込みます。

```
(gdb) symbol-file program.debug
```

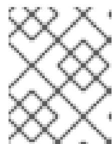
program.debug は実際のファイル名に置き換えます。



注記

コアダンプに含まれるすべての実行可能ファイルにデバッグ情報をインストールする必要はありません。これらの実行可能ファイルの多くは、アプリケーションコードで使用するライブラリーです。これらのライブラリーが、分析中の問題の直接原因でない可能性があるので、ライブラリーのデバッグ情報を含める必要はありません。

6. GDB コマンドを使用して、クラッシュした時点のアプリケーションの状態を検証します。「GDB を使用したアプリケーションの内部状況の検証」を参照してください。



注記

コアファイルを分析する場合は、実行中のプロセスに、GDB はアタッチしません。実行を制御するコマンドを使用しても効果がありません。

関連資料

- GDB でのデバッグ: 「2.1.1 Choosing Files」
- GDB でのデバッグ: 「18.1 Commands to Specify Files」
- GDB でのデバッグ: 「18.3 Debugging Information in Separate Files」

21.4. GCORE を使用したプロセスメモリーのダンプ

コアダンプのデバッグに関するワークフローでは、プログラムの状態をオフラインで分析できます。対象のプロセスで環境にアクセスするのが困難な場合など、実行中のプログラムでこのワークフローを使用すると有益な場合があります。**gcore** コマンドを使用して、実行中にプロセスのメモリーをダンプすることができます。

前提条件

- コアダンプを理解していること
- システムに GDB がインストールされていること

ステップ

gcore を使用してプロセスメモリーをダンプします。

1. プロセス id (**pid**) を検索します。**ps**、**pgrep** および **top** などのツールを使用します。

```
$ ps -C some-program
```

2. このプロセスのメモリーをダンプします。

```
$ gcore -o filename pid
```

このコマンドで、**filename** を作成し、このファイルにプロセスメモリーをダンプします。

3. コアダンプが完了したら、プロセスで通常の実行が再開されます。
4. システムに関する追加情報を指定するには、SOS レポートを作成します。

```
# sosreport
```

   これにより、設定ファイルのコピーなど、お使いのシステムに関する情報が含まれる tar アーカイブが作成されます。

5. デバッグを行うコンピューターに、プログラムの実行可能ファイル、コアダンプ、SOS レポートを移動します。
6. オプション: コアダンプと SOS レポートに移動後には、ディスク容量を開放するために削除します。

関連資料

- 「アプリケーションを再起動せずにコアファイルを取得する方法」: ナレッジベースアーティクル

21.5. GDB での保護されたプロセスメモリーのダンプ

プロセスのメモリーをダンプしないように、マークすることができます。カーネルコアダンプ (**kdump**) や手動のコアダンプ (**gcore**、GDB) は、このようにマークされたメモリーをダンプしません。

このように保護されていても、プロセスメモリーの全コンテンツをダンプする必要がある場合があります。以下の手順では、GDB デバッガーを使用して、全コンテンツをダンプする方法を説明します。

前提条件

- コアダンプを理解していること
- システムに GDB がインストールされていること
- GDB が保護されているメモリープロセスにアタッチされていること

ステップ

1. **/proc/PID/coredump_filter** ファイルの設定を無視するように GDB を設定します。

```
(gdb) set use-coredump-filter off
```

2. メモリーページのフラグ **VM_DONTDUMP** を無視するように GDB を設定します。

```
(gdb) set dump-excluded-mappings on
```

3. メモリーをダンプします。

```
(gdb) gcore core-file
```

   core-file は、メモリーをダンプする先のファイル名に置き換えます。

関連資料

- GDB でのデバッグ: 「How to Produce a Core File from Your Program」

パート VI. パフォーマンスの監視

開発者は、パフォーマンスに最も大きな影響を与えるプログラムの部分に注目するためにプログラムのプロファイルを作成します。収集されるデータのタイプには、プロセッサーの時間を最も多く消費するプログラムのセクションや、メモリーが割り振られる場所などがあります。プロファイルでは、実際のプログラム実行からデータを収集します。そのため、収集されるデータの質は、プログラムが実施する実際のタスクに影響されます。プロファイル時に実施されるタスクは、実際の使用を表すものでなければなりません。これにより、プログラムの実際の使用に起因する問題への対応を開発時に確実に行うことができるようになります。

Red Hat Enterprise Linux には、プロファイルデータを収集する各種ツール (**Valgrind**、**OProfile**、**perf** および **SystemTap**) が多数含まれています。各ツールは、以下のセクションで説明されているように、特殊なプロファイルの実行に適しています。

第22章 VALGRIND

Valgrind は、アプリケーションを詳細にわたりプロファイリングするために使用可能な動的分析ツールを構築するインストルメンテーションフレームワークです。デフォルトのインストールには、5 つの標準ツールが含まれます。**Valgrind** ツールは通常、メモリー管理やスレッドの問題を調査するのに使用します。 **Valgrind** スイートには、必要に応じて新規プロファイリングツールを構築可能なツールも含まれます。

Valgrind は、初期化されていないメモリーの使用、メモリーの不適切な割り振り/解放、およびシステム呼び出しの不適切な引数などのエラーをチェックするための、ユーザー空間バイナリーのインストルメンテーションを提供します。Valgrind のプロファイルツールは、大半のバイナリーで標準ユーザーによる使用が可能ですが、他のプロファイラーと比較すると、**Valgrind** プロファイルの実行速度は大幅に遅くなります。バイナリーのプロファイルを作成するために、**Valgrind** は、特定の仮想マシンでそれを実行すると、すべてのバイナリー命令を傍受できるようになります。**Valgrind** のツールは、ユーザー空間プログラムにおけるメモリー関連の問題を検出する場合に最も役立ちます。ただし、これは、時間固有の問題、カーネルスペースのインストルメンテーションやデバッグには適していません。

Valgrind は、調査中のプログラムやライブラリー向けに **debuginfo** パッケージがインストールされている場合に、最も有用で正確なレポートを提供します。 「デバッグ情報を使用したデバッグの有効化」を参照してください。

22.1. VALGRIND ツール

Valgrind スイートは、以下のツールで構成されています。

memcheck

このツールは、プログラムでメモリー管理の問題を以下ように検出します。

- メモリーへの読み込みと書き込みをすべて確認
- **malloc**、**free**、**new**、または **delete** への呼び出しのようなメモリー操作を傍受します。

その他の方法では、メモリー管理問題を検出するのが難しいため、**memcheck** が、おそらく最も使用される **Valgrind** ツールです。このような問題は、大概、長期間検出されず、診断が難しいクラッシュの原因となります。

cachegrind

cachegrind は、CPU 内の I1、D1、および L2 キャッシュを緻密にシミュレーションして、コード内のキャッシュミスの原因を正確に指摘するキャッシュプロファイラーです。このプロファイラーは、ソースコードの各行に累積される命令、キャッシュミス数、メモリー参照を表示します。また、**cachegrind** は関数別、モジュール別、およびプログラム全体の要約や、マシンごとの数を表示することもできます。

callgrind

cachegrind と同様に、**callgrind** は、キャッシュの動作をモデル化できます。ただし、**callgrind** の主な目的は、実行したコードのコールグラフデータを記録することです。

massif

massif はヒーププロファイラーで、プログラムが使用するヒープメモリーの量を測定し、ヒープブロック、ヒープ管理オーバーヘッド、スタックサイズに関する情報を提供します。ヒーププロファイルは、ヒープメモリーの使用率を減少させる方法を特定する場合に有用です。仮想メモリーを使用するシステムでは、ヒープメモリーの使用率が最適化されたプログラムは、メモリーがなくなることは少なく、必要とするページングの量が少ないため処理も早くなります。

helgrind

POSIX pthreads スレッドプリミティブを使用するプログラムでは、**helgrind** は、以下のような同期エラーを検出します。

- POSIX pthreads API の誤用
- ロックの順序付けの問題から生じる潜在的なデッドロック
- データレース (ロックが適切でない状態でのメモリーへのアクセス)

Valgrind は、独自のプロファイリングツールを開発することができます。これと合わせて、Valgrind には **lackey** ツールが含まれており、独自のツールを作成するためのテンプレートとして使用可能なサンプルとして使用できます。

22.2. VALGRIND の使用

valgrind パッケージとその依存関係は、Valgrind プロファイルの実行に必要なツールをすべてインストールします。Valgrind でプログラムをプロファイリングするには、以下を使用します。

```
$ valgrind --tool=toolname program
```

toolname の引数の一覧については、「Valgrind ツール」を参照してください。Valgrind ツールスイート以外に、**toolname** では **none** も有効な引数として使用できます。この引数では、プロファイリングせずに Valgrind でプログラムを実行できます。Valgrind 自体のデバッグやベンチマーク化をする時に、これは便利です。

Valgrind に、固有のファイルに情報すべてを送信するように指示することも可能です。これには、**--log-file=filename** のオプションを使用します。実行可能ファイル **hello** のメモリー使用状況を確認して、**output** にプロファイル情報を送信するには、以下を使用します。

```
$ valgrind --tool=memcheck --log-file=output hello
```

Valgrind に関する詳しい情報は、「その他の情報」や、ツールの Valgrind スイートに関する他のドキュメントを参照してください。

22.3. その他の情報

Valgrind の詳細情報は、**man valgrind** を参照してください。Red Hat Enterprise Linux には、包括的な **Valgrind ドキュメント** が PDF および HTML 形式で含まれています。これらのドキュメントは以下の場所にあります。

- **/usr/share/doc/valgrind-version/valgrind_manual.pdf**
- **/usr/share/doc/valgrind-version/html/index.html**

第23章 OPROFILE

OProfile は、維持負担の少ないシステム全体のパフォーマンス監視ツールで、**oprofile** パッケージに含まれています。システムのプロセッサー上にあるパフォーマンス監視ハードウェアを使用して、メモリーの参照タイミング、第 2 レベルのキャッシュ要求の回数、受け取ったハードウェア割り込みの回数など、システム上のカーネルと実行可能ファイルに関する情報を取得します。**OProfile** は、Java Virtual Machine (JVM) で実行されるアプリケーションのプロファイリングも実行できます。

以下は、**OProfile** が提供するツールの選択です。

ophelp

システムプロセッサーで使用可能なイベントとその簡単な説明を表示します。

operf

主なプロファイリングツール。**operf** ツールは、Linux Performance Event サブシステムを使用します。これにより、プロファイリングの正確なターゲティングが可能になり、**OProfile** が、システムのパフォーマンス監視ハードウェアを使用するその他のツールと一緒に動作できるようになります。
以前使用していた **opcontrol** ツールとは異なり、初期設定は必要ありません。**--system-wide** オプションが使用されている場合を除き、root 権限がなくても使用できます。

ocount

イベント発生の絶対数をカウントするためのツール。これは、プロセスごと、CPU ごと、またはスレッドごとに、システム全体のイベントをカウントできます。

opimport

サンプルデータベースファイルをシステム用に外部のバイナリ形式からネイティブの形式に変換します。異なるアーキテクチャーからのサンプルデータベースを解析する時にのみこのオプションを使用してください。

opannotate

アプリケーションがデバッグシンボルでコンパイルされている場合は、実行可能ファイル用の注釈付きのソースを作成します。

opreport

プロファイルデータを取得します。

23.1. OPROFILE の使用

operf は、推奨されるプロファイリングデータ収集ツールです。このツールは、初期設定が必要なく、すべてのオプションはコマンドラインで渡されます。レガシーの **opcontrol** ツールとは異なり、**operf** は **root** 権限なしに実行できます。**operf** ツールの使用方法に関する詳しい情報は、**『システム管理者ガイド』**の「operf の使用」の章を参照してください。

例23.1 ocount の使用

次の例は、**sleep** ユーティリティーの実行時に、**ocount** を使用してイベント量のカウントを示しています。

```
$ ocount -e INST_RETIRED -- sleep 1

Events were actively counted for 1.0 seconds.
```

```
Event counts (actual) for /bin/sleep:
Event Count % time counted
INST_RETIRED 683,011 100.00
```

例23.2 operf の基本的な使用状況

以下の例では、**operf** ツールを使用して、**ls -l** ~ コマンドからプロファイリングデータを収集します。

1. **ls** コマンドのデバッグ情報をインストールします。

```
# debuginfo-install -y coreutils
```

2. プロファイリングの実行

```
$ operf ls -l ~
Profiling done.
```

3. 収集したデータを分析します。

```
$ opreport --symbols
CPU: Intel Skylake microarchitecture, speed 3.4e+06 MHz (estimated)
Counted cpu_clk_unhalted events () with a unit mask of 0x00 (Core cycles when at least
one thread on the physical core is not in halt state) count 100000
samples % image name symbol name
161 81.3131 no-vmlinux /no-vmlinux
3 1.5152 libc-2.17.so get_next_seq
3 1.5152 libc-2.17.so strcoll_l
2 1.0101 ld-2.17.so _dl_fixup
2 1.0101 ld-2.17.so _dl_lookup_symbol_x
[...]
```

例23.3 Java プログラムのプロファイリングでの operf の使用

以下の例では、**operf** ツールを使用して Java (JIT) プログラムからのプロファイリングデータを収集し、次に **opreport** を使用して、シンボル毎にデータを出力します。

1. この例で使用するデモの Java プログラムをインストールします。これは、**java-1.8.0-openjdk-demo** パッケージの一部で、**Optional** チャンネルに含まれます。**Optional**チャンネルの使用方法については、「Enabling Supplementary and Optional Repositories」を参照してください。**Optional** チャンネルを有効にしたら以下のパッケージをインストールします。

```
# yum install java-1.8.0-openjdk-demo
```

2. **OProfile** の **oprofile-jit** パッケージをインストールして、Java プログラムからプロファイルデータを収集できるようにします。

```
# yum install oprofile-jit
```

3. **OProfile** データのディレクトリーを作成します。

```
$ mkdir ~/oprofile_data
```

4. デモプログラムが含まれるディレクトリーに移動します。

```
$ cd /usr/lib/jvm/java-1.8.0-openjdk/demo/applets/MoleculeViewer/
```

5. プロファイリングを開始します。

```
$ operf -d ~/oprofile_data appletviewer \
-J"-agentpath:/usr/lib64/oprofile/libjvmti_oprofile.so" example2.html
```

6. ホームディレクトリーに移動して、収集したデータを分析します。

```
$ cd
```

```
$ opreport --symbols --threshold 0.5
```

出力例は以下のとおりです。

```
$ opreport --symbols --threshold 0.5
Using /home/rkratky/oprofile_data/samples/ for samples directory.
CPU: Intel Ivy Bridge microarchitecture, speed 3600 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask
of 0x00 (No unit mask) count 100000
samples  %        image name               symbol name
14270    57.1257  libjvm.so                /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.51-
1.b16.el7_1.x86_64/jre/lib/amd64/server/libjvm.so
3537     14.1593  23719.jo                 Interpreter
690      2.7622  libc-2.17.so             fgetc
581      2.3259  libX11.so.6.3.0          /usr/lib64/libX11.so.6.3.0
364      1.4572  libpthread-2.17.so       pthread_getspecific
130      0.5204  libfreetype.so.6.10.0    /usr/lib64/libfreetype.so.6.10.0
128      0.5124  libc-2.17.so             __memset_sse2
```

23.2. OPROFILE のドキュメント

OProfile に関する詳細情報は、**oprofile**(1) man ページを参照してください。Red Hat Enterprise Linux には、**OProfile** に関する包括的なガイドが含まれており、このガイドは **file:///usr/share/doc/oprofile-version/** にあります。

OProfile Manual

OProfile の詳細にわたる設定や用途の説明が含まれる包括的なマニュアルは、**file:///usr/share/doc/oprofile-version/oprofile.html** にあります。

OProfile Internals

OProfile の内部での機能に関する情報は、**file:///usr/share/doc/oprofile-version/internals.html** にあり、**OProfile** アップストリームへの貢献に興味のあるプログラマーには役立つ情報が含まれています。

第24章 SYSTEMTAP

SystemTap は、Linux システムで実行中のプロセスやカーネルアクティビティーをプローブするための便利なインストルメンテーションプラットフォームです。プローブを実行するには、以下の手順に従います。

1. どのシステムイベント (たとえば、仮想ファイルシステムの読み込み、パケット送信) が特定のアクション (たとえば、印刷、解析、またはデータ操作) を開始するかを指定する **SystemTap スクリプト** を書き込みます。

2. SystemTap がスクリプトを C プログラミングに変換し、さらにカーネルモジュールにコンパイルします。

3. SystemTap がこのカーネルモジュールを読み込んで、実際のプローブを実行します。

SystemTap スクリプトは、通常のシステム運用に最小限の割り込みでシステム運用を監視しシステム問題を診断する際に便利なものです。インストルメント化されたコードを再コンパイルしたり再インストールすることなく、実行中のシステムテスト仮説をすばやくインストルメント化できます。**kernel-space** をプローブする SystemTap スクリプトをコンパイルするために、SystemTap は以下の 3 つの異なる **カーネル情報パッケージ** からの情報を使用します。

- **kernel-variant-devel-version**
- **kernel-variant-debuginfo-version**
- **kernel-debuginfo-common-arch-version**

これらのカーネル情報パッケージは、プローブ対象のカーネルと一致する必要があります。さらに、複数のカーネル用に SystemTap スクリプトをコンパイルするには、各カーネルのカーネル情報パッケージがインストールされている必要もあります。

24.1. 追加情報

SystemTap の詳細情報は、以下の Red Hat ドキュメントを参照してください。

- 『SystemTap ビギナーズガイド』
- SystemTap タップセットリファレンス

第25章 PERFORMANCE COUNTERS FOR LINUX (PCL) ツールおよび PERF

Performance Counters for Linux (PCL) は、パフォーマンスデータを収集、分析するフレームワークを提供する、カーネルベースのサブシステムです。Red Hat Enterprise Linux 7 には、データやユーザー空間ツール **perf** を収集するこのカーネルサブシステムが含まれており、収集したパフォーマンスデータを分析します。 PCL サブシステムを使用して、リタイアした説明やプロセッサークロックサイクルなど、ハードウェアイベントを測定することができます。主なページの障害やコンテキストスイッチなど、ソフトウェアイベントを測定することも可能です。たとえば、PCL カウンターは、リタイアしたプロセスの命令数やプロセッサーのクロックサイクルをもとに、 *(IPC) を算出することができます。IPC の割合が低い場合は、コードがCPU をあまり使用していないことが分かります。他のハードウェアイベントを使用して、CPU パフォーマンスの低さを診断することも可能です。*

パフォーマンスカウンターは、サンプルを記録するように設定することもできます。サンプルの相対的頻度を使用して、コードのどの領域がパフォーマンスに最も影響があるかを特定することができます。

25.1. PERF ツールコマンド

*Useful **perf** コマンドには以下が含まれます。*

perf stat

*この **perf** コマンドは、実行した命令や、使用したクロックサイクルなど、一般的なパフォーマンスイベントの全体的な統計が分かります。オプションを指定すると、デフォルトの計測イベント以外のイベントも選択することができます。*

perf record

*この **perf** コマンドは、パフォーマンスデータを **perf report** を使用して後ほど分析できるように、ファイルに記録します。*

perf report

*この **perf** コマンドは、ファイルからパフォーマンスデータを読み込み、記録されたデータを分析します。*

perf list

*この **perf** コマンドは、特定のマシンで利用可能なイベントを一覧表示します。これらのイベントは、パフォーマンス監視ハードウェアや、システムのソフトウェア設定により異なります。*

perf** コマンドの完全一覧を取得するには、**perf help** を使用します。**perf** コマンドごとに、**man** ページの情報を取得するには、**perf help **command** *を使用します。*

25.2. PERF の使用方法

プログラム実行の統計やサンプルと収集するための基本的なPCL インフラストラクチャーは、比較的単純です。以下のセクションでは、全体的な統計やサンプルのシンプルな例を紹介します。

***make** およびその子についての統計情報を収集するには、以下のコマンドを使用します。*

```
# perf stat -- make all
```

***perf** コマンドは、多くの異なるハードウェアおよびソフトウェアカウンターを収集し、以下の情報を表示します。*

```
Performance counter stats for 'make all':
```

```
 244011.782059  task-clock-msecs         #      0.925 CPUs
         53328  context-switches         #      0.000 M/sec
           515  CPU-migrations           #      0.000 M/sec
       1843121  page-faults              #      0.008 M/sec
  789702529782  cycles                   #   3236.330 M/sec
 1050912611378  instructions             #      1.331 IPC
  275538938708  branches                 #   1129.203 M/sec
    2888756216  branch-misses            #      1.048 %
    4343060367  cache-references         #     17.799 M/sec
     428257037  cache-misses             #      1.755 M/sec

 263.779192511  seconds time elapsed
```

perf ツールはサンプルを記録することもできます。たとえば、**make** コマンドおよびその子に関するデータを記録するには、以下のコマンドを実行します。

```
# perf record -- make all
```

これで収集されたサンプル数とサンプルが保存されているファイルが表示されます。

```
[ perf record: Woken up 42 times to write data ]
[ perf record: Captured and wrote 9.753 MB perf.data (~426109 samples) ]
```

Performance Counters for Linux (PCL) ツールが OProfile と競合

OProfile と Performance Counters for Linux (PCL) は同じハードウェアの Performance Monitoring Unit (PMU) を使用します。PCL **perf** コマンドを使用する際に OProfile が実行中である場合は、OProfile 開始時に以下のようなエラーメッセージが表示されます。

```
Error: open_counter returned with 16 (Device or resource busy). /usr/bin/dmesg may provide
additional information.

Fatal: Not all events could be opened.
```

perf コマンドを使用するには、まず OProfile をシャットダウンします。

```
# opcontrol --deinit
```

その後に **perf.data** を分析してサンプルの相対頻度を測定することができます。レポート出力には、コマンド、オブジェクト、サンプルの機能などが含まれます。**perf report** を使って **perf.data** の分析を出力します。たとえば、以下のコマンドは最も時間がかかる実行可能ファイルのレポートを作成します。

```
# perf report --sort=comm
```

出力は以下のようになります。

```
# Samples: 1083783860000
#
# Overhead          Command
# ........  ...............
#
    48.19%         xsltproc
    44.48%        pdfxmltex
     6.01%             make
```

```
0.95%             perl
0.17%       kernel-doc
0.05%          xmllint
0.05%              cc1
0.03%               cp
0.01%            xmlto
0.01%               sh
0.01%          docproc
0.01%               ld
0.01%              gcc
0.00%               rm
0.00%              sed
0.00%   git-diff-files
0.00%             bash
0.00%   git-diff-index
```

左のコラムはサンプルの相対頻度を示しています。この出力では、**make** が **xsltproc** および **pdfxmltex** で最も多くの時間を消費していることを示しています。**make** が完了する時間を短縮するには、**xsltproc** と **pdfxmltex** にフォーカスします。**xsltproc** が実行する機能を一覧表示するには、以下のコマンドを実行します。

```
# perf report -n --comm=xsltproc
```

以下が生成されます。

```
comm: xsltproc
# Samples: 472520675377
#
# Overhead  Samples                    Shared Object  Symbol
# ........ ..........  .............................  ......
#
    45.54%215179861044  libxml2.so.2.7.6              [.] xmlXPathCmpNodesExt
    11.63%54959620202  libxml2.so.2.7.6              [.] xmlXPathNodeSetAdd__internal_alias
     8.60%40634845107  libxml2.so.2.7.6              [.] xmlXPathCompOpEval
     4.63%21864091080  libxml2.so.2.7.6              [.] xmlXPathReleaseObject
     2.73%12919672281  libxml2.so.2.7.6              [.] xmlXPathNodeSetSort__internal_alias
     2.60%12271959697  libxml2.so.2.7.6              [.] valuePop
     2.41%11379910918  libxml2.so.2.7.6              [.] xmlXPathIsNaN__internal_alias
     2.19%10340901937  libxml2.so.2.7.6              [.] valuePush__internal_alias
```

付録A 改訂履歴

改訂番号 7-6.1、2018 年 10 月 30 日 (火)、Vladimír Slávik

7.6 GA リリース用のビルド

改訂番号 7-6、2018 年 8 月 21 日 (火)、Vladimír Slávik

7.6 ベータ版のビルド

改訂番号 7-5.1、2018 年 4 月 10 日 (火)、Vladimír Slávik

7.5 GA リリース向けに作成

改訂番号 7-5、2018 年 1 月 9 日 (火)、Vladimír Slávik

新規バージョン 7.5 のベータ版のプレビューを公開

改訂番号 7-4.1、2017 年 8 月 22 日 (火)、Vladimír Slávik

関連製品の新規リリース用に更新

改訂番号 7-4、2017 年 7 月 26 日 (水)、Vladimír Slávik

7.4 GA リリース用にビルド。開発用のワークステーション設定に関する章を追加

改訂番号 1-12、2017 年 5 月 26 日 (金)、Vladimír Slávik

古くなった情報を削除するために更新

改訂番号 7-3.9、2017 年 5 月 15 日 (月)、Robert Krátký

7.4 ベータリリース向けのビルド