Red Hat Enterprise Linux 7

开发人员指南

RHEL 7 中的应用程序开发工具简介

Last Updated: 2023-03-25

Red Hat Enterprise Linux 7 开发人员指南

RHEL 7 中的应用程序开发工具简介

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

法律通告

摘要

本文档描述了使 Red Hat Enterprise Linux 7 成为应用程序开发的理想企业平台的不同功能和实用程序。

目录

前言

本文档描述了使 Red Hat Enterprise Linux 7 成为应用程序开发的理想企业平台的不同功能和实用程序。

在 RHEL 7.9 发布后，Red Hat Enterprise Linux 7 将停止更新。查看 Red Hat Enterprise Linux 7 的 Red Hat Software Collections 产品生命周期 以了解有关 RHEL 7 产品当前状态的信息。

部分 I. 设置开发工作站

Red Hat Enterprise Linux 7 支持开发自定义应用程序。要允许开发人员这样做，必须使用必要的工具和实用程序设置系统。本章列出了开发的最常见用例和要安装的项目。

第 1 章 安装操作系统

在为特定开发需求设置前，必须设置底层系统。

1. 在 **工作站** 变体中安装 Red Hat Enterprise Linux。按照 Red Hat Enterprise Linux 安装指南中的说明。
2. 安装时，注意 软件选择。选择 **Development 和 Creative Workstation** 系统配置文件，并启用适合您的开发需求的附加组件。以下各节中列出了相关的附加组件，侧重于各种类型的开发。
3. 要开发与 Linux 内核（如驱动程序）互操作的应用程序，请在安装过程中使用 **kdump** 启用自动崩溃转储。
4. 系统本身安装后，进行注册并附加所需的订阅。按照 Red Hat Enterprise Linux 系统管理员指南、第 7 章、注册系统和管理订阅中 的说明操作。
   以下小节列出了必须为相应开发类型附加的特定订阅。
5. Red Hat Software Collections 提供了最新版本的开发工具和实用程序。有关访问 Red Hat Software Collections 的说明，请参阅 Red Hat Software Collections 发行注记，第 2 章。

其它资源

- Red Hat Enterprise Linux 安装指南 - Subscription Manager
- Red Hat Subscription Management
- Red Hat Enterprise Linux 7 软件包清单

第 2 章 设置来管理应用版本

有效的版本控制是所有多开发人员项目的基础。Red Hat Enterprise Linux 与 **Git** 一同发布，它是一个分布式版本控制系统。

1. 在系统安装过程中选择 **Development Tools** Add-on 以安装 **Git**。
2. 另外，还可在安装系统后从 Red Hat Enterprise Linux 软件仓库安装 **git** 软件包。

```
# yum install git
```

3. 要获取红帽支持的最新版本，请从 Red Hat Software Collections 安装 **rh-git227** 组件。

```
# yum install rh-git227
```

4. **设置与 Git 提交关联的完整名称和电子邮件地址：**

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

将 *全名* 和 *email_address* 替换为您的实际名称和电子邮件地址。

5. **要更改 Git 启动的默认文本编辑器，请设置 core.editor 配置选项的值：**

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

使用 *命令* 替换 命令，以启动选定的文本编辑器。

其它资源

- **第 11 章 *使用 Git***

第 3 章 使用 C 和 C++ 设置开发应用

Red Hat Enterprise Linux 最适合支持使用完全编译的 C 和 C++ 编程语言进行开发。

1. 在系统安装过程中，选择 开发工具和 调试工具附加组件，以安装 GNU 编译器集合(GCC) 和 GNU Debugger(GDB) 和其他开发工具。

2. GCC、GDB 和相关工具的最新版本作为 Red Hat Developer Toolset 工具链组件的一部分提供。

```
# yum install devtoolset-9-toolchain
```

注：Red Hat Developer Toolset 作为 Software Collection 提供。scl 工具允许您在 Red Hat Developer Toolset 二进制文件中使用该命令，等同于 Red Hat Enterprise Linux 系统。

3. Red Hat Enterprise Linux 软件仓库包含许多广泛用于 C 和 C++ 应用程序的库。使用 yum 软件包管理器安装应用程序所需的库的开发软件包。

4. 对于基于图形的开发环境，请安装 Eclipse 集成开发环境。C 和 C++ 语言直接支持。Eclipse 作为 Red Hat Developer Tools 的一部分提供。有关实际安装过程，请参阅使用 Eclipse。

其它资源

- Red Hat Developer Toolset 用户指南 - Red Hat Developer Toolset components 第 1 章

第 4 章 设置调试应用程序

红帽企业 Linux 提供多种调试和工具，用于分析和故障排除内部应用程序行为。

1. 在系统安装过程中，选择 调试工具和 桌面调试和性能工具 附加组件，以安装 GNU Debugger(GDB)、Valgrind、 SystemTap 、Ltrace、strace 和其他工具。

2. 对于 GDB、Valgrind、 SystemTap 、strace 和 ltrace 的最新版本，请安装 Red Hat Developer Toolset。这也会安装 memstomp。

```
# yum install devtoolset-9
```

注：Red Hat Developer Toolset 作为 Software Collection 提供。scl 工具允许您在 Red Hat Developer Toolset 二进制文件中使用该命令，等同于 Red Hat Enterprise Linux 系统。

3. memstomp 工具仅作为 Red Hat Developer Toolset 的一部分提供。如果安装整个开发人员工具集并非必要，并且需要 memstomp，则仅从 Red Hat Developer Toolset 安装其组件。

```
# yum install devtoolset-9-memstomp
```

4. 安装 yum-utils 软件包以使用 debuginfo-install 工具：

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

5. 要调试作为 Red Hat Enterprise Linux 的一部分提供的应用程序和库，请使用 debuginfo-install 工具从 Red Hat Enterprise Linux 仓库安装对应的 debuginfo 和源软件包。这也适用于内核转储文件分析。

6. 安装 SystemTap 应用所需的内核调试信息和源软件包。请参阅 SystemTap 初学指南、第 2.1.1 章、安装 SystemTap。

7. 要捕获内核转储，安装和配置 kdump。按照 内核 Crash 转储指南、第 7.2 章、安装和配置 kdump 中的说明操作。

8. 确保 SELinux 策略允许相关应用程序正常运行，而且在调试情况下也一样。请参阅 SELinux 用户和管理员指南，第 11.3 节，修复问题。

其它资源

- 第 20.1 节 “使用调试信息启用调试”
- SystemTap 入门指南

第 5 章 设置应用程序的性能

红帽企业 Linux 包括多个应用程序，可帮助开发人员确定应用程序性能丢失的原因。

1. 在系统安装过程中，选择 调试工具、开发工具 和 Performance Tools 附加组件，以安装工具 OProfile、perf 和 pcp。

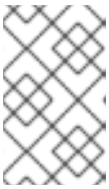
2. 安装 SystemTap 工具，允许某些类型的性能分析和 Valgrind，其中包括用于性能测量的模块。

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

注：Red Hat Developer Toolset 作为 Software Collection 提供。scl 工具允许您在 Red Hat Developer Toolset 二进制文件中使用该命令，等同于 Red Hat Enterprise Linux 系统。

3. 运行 SystemTap 帮助程序脚本来设置 SystemTap 环境。

```
# stap-prep
```



注意

运行此脚本将安装非常大的内核调试信息软件包。

4. 要更频繁地更新 SystemTap、OProfile 和 Valgrind 版本，请安装 Red Hat Developer Toolset 软件包 perftools。

```
# yum install devtoolset-9-perftools
```

其它资源

- Red Hat Developer Toolset 用户指南 - 第 IV 部分，性能监控工具

第 6 章 使用 JAVA 设置开发应用程序

Red Hat Enterprise Linux 支持在 Java 中开发应用程序。

1. 在系统安装过程中，选择 Java Platform Add-on 以安装 OpenJDK 作为默认的 Java 版本。

   或者，按照 Red Hat CodeReady Studio 第 2.2 章 在 RHEL 上安装 OpenJDK 1.8.0 中的说明，以单独安装 OpenJDK。

2. 对于集成的图形开发环境，请安装基于 Eclipse 的 Red Hat CodeReady Studio，它提供对 Java 开发的广泛支持。按照《 Red Hat CodeReady Studio 安装指南》 中的说明操作。

第 7 章 使用 PYTHON 设置开发应用程序

Python 语言版本 2.7.5 作为 Red Hat Enterprise Linux 的一部分提供。

- Python 解释器和库的新版本（包括较新版本的 Python 2.7）作为 Red Hat Software Collections 软件包提供。根据下表所示，使用所需版本安装 软件包。

```
# yum install package
```

与 Red Hat Software Collections 软件包对应的 Python 版本

版本	软件包
Python 2.7	python27
Python 3.6	rh-python36
Python 3.8	rh-python38

python27 软件集合是 RHEL 7 中 Python 2 软件包的更新版本。

2. 安装支持以 Python 语言进行开发的 Eclipse 集成开发环境。Eclipse 作为 Red Hat Developer Tools 的一部分提供。有关实际安装过程，请参阅使用 Eclipse。

其它资源

- Red Hat Software Collections Hello-World - Python
- Red Hat Software Collections

第 8 章 使用 C# 和 .NET CORE 设置到开发应用程序

红帽支持开发以 .NET Core 为目标的应用程序。

- 为 Red Hat Enterprise Linux 安装 .NET Core，其中包括运行时、编译器和其他工具。按照 .NET Core 入门指南 中的说明进行操作。

其它资源

- .NET Core for Red Hat Enterprise Linux Overview
- .NET Core for Red Hat Enterprise Linux 文档

第 9 章 设置开发容器化应用

红帽支持基于 Red Hat Enterprise Linux、Red Hat OpenShift 以及许多其他红帽产品开发容器化应用程序。

- Red Hat Container Development Kit (CDK)提供了一个 Red Hat Enterprise Linux 虚拟机，它运行单节点 Red Hat OpenShift 3 集群。它不支持 OpenShift 4。按照《红帽容器开发套件入门指南》第 1.4 章安装 CDK 中的说明操作。

- Red Hat CodeReady Containers (CRC)将最小的 OpenShift 4 集群引入到本地计算机上，为开发和测试提供最小的环境。CodeReady Containers 主要面向在开发人员的桌面上运行。

- Red Hat Development Suite 为在 Java、C 和 C++ 中容器化应用程序的开发提供工具。它由 Red Hat JBoss Developer Studio、OpenJDK、Red Hat Container Development Kit 以及其他次要组件组成。要安装 DevSuite，请遵循《红帽开发套件安装指南》 中的说明操作。

- .NET Core 3.1 是一个通用的开发平台，用于构建在 OpenShift Container Platform 版本 3.3 及更新版本中运行的高质量应用程序。有关安装和使用说明，请参阅 Red Hat OpenShift Container Platform 中使用 .NET Core 3.1 的第 2 章。

其它资源

- Red Hat CodeReady Studio - 容器和基于云的开发入门

- Red Hat Container Development Kit 产品文档

- OpenShift Container Platform 产品文档

- Red Hat Enterprise Linux Atomic Host - 红帽系统中的容器概述

第 10 章 设置到开发 WEB 应用程序

通过作为部署的平台，红帽企业 Linux 支持开发 Web 应用程序。

Web 开发主题太大，可通过几个简单的指令捕获。本节只提供在 Red Hat Enterprise Linux 中开发网络应用程序的最佳支持路径。

- 要为开发传统 Web 应用程序设置您的环境，请安装 Apache Web 服务器、PHP 运行时和 MariaDB 数据库服务器和工具。

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

另外，这些应用程序的最新版本也可以作为 Red Hat Software Collections 的组件提供。

```
# yum install httpd24 rh-mariadb102 rh-php73
```

其它资源

- Red Hat Software Collections
- Red Hat Developer portal Cheat Sheet - 高级 Linux Commands Cheat Sheet（设置 LAMP 堆栈）

部分 II. 使用其他开发人员在主机上协作

本文档部分介绍了版本控制系统 Git。

第 11 章 使用 GIT

有效修订控制对所有多开发人员项目来说是必不可少的。它可让团队中的所有开发人员以系统方式创建、审核、修改和文档代码。Red Hat Enterprise Linux 7 带有开源修订控制系统 Git。

Git 及其功能的详细描述已超出本书范围。有关此修订控制系统的更多信息，请参阅以下列出的资源。

安装的文档

- Git 和教程的 Linux 手册页：

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

请注意，很多 Git 命令都有自己的 man page。

- *Git 用户的手册* - Git 的 HTML 文档位于 /usr/share/doc/git-1.8.3/user-manual.html。

在线文档

- *Pro Git* 书的在线版本提供了 Git、其概念及其用法的详细描述 - Pro Git Book

- Git 的 Linux 手册页的在线版本 - Pro Git 参考表

部分 III. 使应用程序可供用户使用

可以通过多种方式将应用程序提供给用户。本指南描述了最常见的方法：

- 将应用程序打包为 RPM 软件包
- 将应用程序打包为软件集合
- 将应用程序打包为容器

第 12 章 分发选项

Red Hat Enterprise Linux 为第三方应用程序提供三种发行方式。

RPM 软件包

RPM 软件包是分发和安装软件的传统方法。

- RPM 软件包是一个成熟的技术，具有多个工具，广泛地解雇知识。
- 应用程序作为系统的一部分安装。
- 安装工具大大有助于解决依赖项。

注意

只能安装一个软件包版本，使多个应用程序版本安装比较困难。

要创建 RPM 软件包，请按照 RPM 包指南中的说明，打包软件。

Software Collections

Software Collection 是一个特别准备的 RPM 软件包，用于替代应用程序版本。

- Software Collection 是由红帽提供的使用和支持的打包方法。
- 它基于 RPM 软件包机制构建。
- 可以同时安装应用程序的多个版本。

如需更多信息，请参阅 Red Hat Software Collections Packaging Guide, Are Software Collections?

要创建软件集合软件包，请按照 Red Hat Software Collections Pack Guide 中的说明 进行打包软件集合。

容器

Docker 格式的容器是一种轻量级虚拟化方法。

- 应用程序可以存在于多个独立版本和实例中。
- 它们可以通过 RPM 软件包或 Software Collection 轻松准备。
- 与系统的交互可以精确控制。
- 应用的隔离会增加安全性。
- 应用或其组件容器化允许编排多个实例。

其它资源

- Red Hat Software Collections Packaging Guide - 什么是 Software Collections?

第 13 章 使用应用程序创建容器

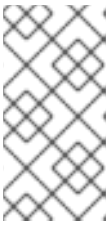
本节论述了如何从本地构建的应用程序创建 docker 格式的容器镜像。在您希望使用编排进行部署时，使您的应用作为容器可用。另外，容器化可以有效地解决依赖项冲突。

先决条件

- 了解容器
- 从源本地构建的应用程序

步骤

1. 确定要使用的基础镜像。



注意

红帽建议从使用 Red Hat Enterprise Linux 作为其基础的基础镜像开始。如需更多信息，请参阅 Red Hat Container Catalog 中的基础镜像。

2. 创建工作空间目录。
3. 将应用程序准备为包含所有应用程序所需文件的目录。将该目录放到工作区目录中。
4. 编写 Dockerfile，用于描述创建容器所需的步骤。

如需有关如何包含您的内容、设置要运行的默认命令以及打开必要的端口和其他功能的信息，请参阅 Dockerfile 参考。

此示例显示包含 my-program/ 目录的最小 Dockerfile：

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

将此 Dockerfile 放入工作空间目录中。

5. 从 Dockerfile 构建容器镜像：

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

在这一步中，请注意新创建的容器镜像的 ***container-id***。

6. 向镜像添加标签，以标识想存储容器镜像的 registry。请参阅开始使用容器 - 标记镜像。

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

将 ***container-id*** 替换为上一步输出中显示的值。

使用您 要将镜像推送到的 registry 地址替换 registry，使用 registry 的端口（如果需要，*使用名称*）替换 registry。

例如，如果您在本地系统中使用 ***docker-distribution*** *服务来运行* ***registry***，*并且名为* ***myimage*** *的镜像，标签* ***localhost:5000/myimage*** *会启用该镜像推送到* ***registry***。

7. *将镜像推送到* ***registry***，*以便稍后从该* ***registry*** *中拉取。*

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

将 ***tag*** *部分替换为与上一步中使用的值相同的值。*

要运行自己的 ***Docker registry***，*请参阅开始使用容器 - 使用* ***Docker registry***。

其它资源

- ***OpenShift Container Platform - Develop:****镜像*

- ***Red Hat Enterprise Linux Atomic Host* - *容器开发推荐做法***
- ***Dockerfile* *参考***
- ***Docker* *文档* - *入门，第 2 部分：容器***
- ***Red Hat Enterprise Linux Atomic Host* - *开始使用容器***
- ***Red Hat Container Catalog* *列表* - *基础镜像***

第 14 章 从软件包中容器化应用程序

出于多种原因，分发 **RPM** 包中打包为容器的应用可能会有一定优势。

先决条件

- 了解容器
- 打包为一个或多个 **RPM** 软件包的应用程序

步骤

要从 **RPM** 软件包容器化应用程序，请参阅开始使用容器 - 创建 **Docker** 镜像。

其它信息

- **OpenShift Container Platform** - 创建镜像
- **Red Hat Enterprise Linux Atomic Host** - 开始使用容器
- **Red Hat Enterprise Linux Atomic Host** 产品文档
- **Docker** 文档 - 入门，第 **2** 部分：容器
- **Docker** 文档 - **Dockerfile** 参考
- **Red Hat Container Catalog** 列表 - 基础镜像

部分 IV. 创建 C 或 C++ 应用程序

红帽提供了多种工具，可用于使用 C 和 C++ 语言创建应用程序。本书的这一部分列出了一些最常见的开发任务。

第 15 章 使用 GCC 构建代码

本章论述了源代码必须转换为可执行代码的情况。

15.1. CODE FORMS 之间的关系

先决条件

- 了解编译和链接的概念

可能的代码表单

使用 C 和 C++ 语言时，有三种类型的代码：

- 使用 C 或 C++ 语言编写的源代码，以纯文本文件形式存在。

  文件通常使用扩展名，如 .c,.cc,.cpp,.h,.hpp,.i,.inc。有关支持的扩展及其解释的完整列表，请查看 gcc 手册页：

  ```
  $ man gcc
  ```

- 对象代码，通过使用 编译器 编译 源代码来创建。这是一种中间形式。

  对象代码文件使用 .o 扩展名。

- 可执行代码,通过带有一个 linker 的 linking 对象代码来创建。

  Linux 应用程序可执行文件不使用任何文件名扩展名。共享对象(library)可执行文件使用 .so 文件名扩展名。

注意

也存在用于静态链接的库存档文件。这是使用 .a 文件名扩展名的对象代码变体。不建议使用静态链接。请参阅 第 16.2 节 “静态和动态链接”。

在 GCC 中处理代码表单

从源代码生成可执行代码需要两个步骤，它们需要不同的应用程序或工具。**GCC** 可用作编译器和链路器的智能驱动程序。这可让您对任何所需操作使用单个命令 **gcc**。**GCC** 自动选择所需操作（编译和链接），以及它们的顺序：

1. 源文件编译成对象文件。
2. 对象文件和库链接（包括之前编译的源）。

可以运行 **GCC** 以便只发生第 **1** 步、只发生第 **2** 步或执行第 **1** 步和第 **2** 步。这由输入类型和请求的输出类型决定。

因为大型项目需要一个构建系统，它通常会为每个操作单独运行 **GCC**，因此始终将编译和链接视为两个不同的操作，即使 **GCC** 可以同时执行。

其它资源

- 第 **15.2** 节 “编译源文件到对象代码”
- 第 **15.6** 节 “链接代码来创建可执行文件”

15.2. 编译源文件到对象代码

要从源代码文件而不是可执行文件立即创建对象代码文件，必须指示 **GCC** 仅创建对象代码文件作为其输出。此操作代表了大型项目的构建流程的基本操作。

先决条件

- ***C* 或 *C++* 源代码文件.**
- 系统中安装了 **GCC**

步骤

1. 更改到包含源代码文件的目录。

2. 使用 **-c** 选项运行 **gcc** ：

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

创建对象文件，其文件名反映原始源代码文件：**source .c** 会生成 **source.o**。

注意

使用 **C++** 源代码，将 **gcc** 命令替换为 **g++**，以方便处理 **C++** 标准库依赖项。

其它资源

- 第 15.5 节 “使用 GCC 强化代码”
- 第 15.4 节 “使用 GCC 进行代码优化”
- 第 15.8 节 “Example:使用 GCC 构建 C 程序”

15.3. 使用 GCC 启用 C 和 C++ 应用程序

由于调试信息较大，因此默认情况下，不会包含在可执行文件中。要为您的 **C** 和 **C++** 应用程序进行调试，您必须明确指示编译器创建调试信息。

使用 **GCC** 启用调试信息

要在编译和链接代码时使用 **GCC** 创建调试信息，请使用 **-g** 选项：

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

- 由编译器和链接器执行的优化可能会导致可执行代码难以与原始源代码相关：变量可以被优化，取消滚动，将操作合并到周围的代码中。这会影响调试。要改进调试体验，请考虑使用 **-Og** 选项设置优化。但是，更改优化级别会更改可执行代码，并可能会更改实际行为，以便删除一些程序错误。
- **f compare-debug GCC** 选项测试 **GCC** 使用调试信息编译的代码，无需调试信息。如果生成

*的两个二进制文件相同，则测试将通过。此测试可确保可执行代码不受任何调试选项的影响，进一步确保调试代码中没有隐藏的错误。请注意，使用 **-fcompare-debug** 选项可显著增加编译时间。有关这个选项的详情，请查看 **GCC** 手册页。*

其它资源

- *第 **20.1** 节 “使用调试信息启用调试”*
- *使用 **GNU Compiler Collection(GCC)- 3.10** 选项用于调试您的程序*
- *使用 **GDB** 进行调试 — **18.3 Debugging Information in Separate Files***
- ***GCC** 手册页：*

```
$ man gcc
```

15.4. 使用 GCC 进行代码优化

单个程序可以转换为多个计算机指令序列。如果为编译期间为分析代码分配了更多资源，则可以实现最佳结果。

*使用 **GCC** 进行代码优化*

*在 **GCC** 中，可以使用 **-O**级别 选项来设置优化级别。这个选项接受一组值来代替 级别。*

级别	描述
0	优化编译速度 - 无代码优化（默认）
1, 2, 3	增加代码执行速度的优化工作
s	优化生成文件大小
fast	级别 3 加上忽略严格标准合规性以允许进行额外的优化
g	优化调试体验

*对于版本构建，建议使用优化选项 **-O2**。*

在开发过程中，**-Og** 选项在某些情况下用于调试程序或库。因为有些错误清单只具有某些优化级别，所以请确保使用发行版本优化级别测试程序或库。

GCC 提供了大量选项，以实现单个优化。如需更多信息，请参阅以下附加资源。

其它资源

- 使用 **GNU Compiler Collection - 3.11** 选项可控制优化
- **GCC** 的 **Linux man page**：

```
$ man gcc
```

15.5. 使用 GCC 强化代码

当编译器将源代码转换为对象代码时，它可以添加各种检查来防止经常被利用的情况，从而提高安全性。选择正确的编译器选项集可帮助生成更安全的程序和库，而无需更改源代码。

发行版本选项

对于目标为 **Red Hat Enterprise Linux** 的开发人员，推荐使用以下选项列表：

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

- 对于程序，添加 **-fPIE** 和 **-pie** 位置独立的可执行文件选项。
- 对于动态链接的库，强制 **-fPIC (Position Independent Code)**选项间接提高了安全性。

开发选项

建议您在开发过程中检测安全漏洞。使用这些选项以及发行版本的选项：

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

其它资源

- 防御编码指南
- **Red Hat Developer** 博客文章 - 使用 **GCC** 的内存错误检测

15.6. 链接代码来创建可执行文件

构建 **C** 或 **C++** 应用程序时，链接是最后一步。链接将所有对象文件和库组合到一个可执行文件中。

先决条件

- 一个或多个对象文件
- **GCC** 已安装在系统上

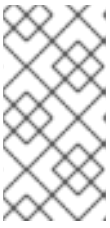
步骤

1. 更改到含有对象代码文件的目录。
2. 运行 **gcc** ：

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

从提供的对象文件和库中创建名为可执行文件可执行文件。

要链接其他库，请在对象文件列表前添加所需的选项。请参阅 **第 16 章 在 GCC 中使用 Libraries**。



注意

使用 **C++** 源代码，将 **gcc** 命令替换为 **g++**，以方便处理 **C++** 标准库依赖项。

其它资源

- 第 15.8 节 “Example:使用 GCC 构建 C 程序”
- 第 16 章 在 GCC 中使用 Libraries

15.7. 各种红帽产品的 C++ 兼容性

红帽生态系统包括多个 Red Hat Enterprise Linux 和 Red Hat Developer Toolset 提供的 GCC 编译器和 linker 版本。两者之间的 C++ ABI 兼容性如下：

- 基于 GCC 4.8 且作为 Red Hat Enterprise Linux 7 的一部分提供 的系统编译器 仅支持编译和连接 C++98 标准（也称为 C++03），及其使用 GNU 扩展的功能。
- 任何符合 C++98 的二进制文件或库都使用选项 -std=c++98 或 -std=gnu++98 明确构建，无论使用的编译器是什么版本是什么。
- 只有在使用 Red Hat Developer Toolset 中的编译器并使用相应标记构建时，才会支持使用 C++11 和 C++14 语言版本来构建的 C++ 对象。
- 当使用 Red Hat Developer Toolset 和 Red Hat Enterprise Linux 工具链构建 C++ 文件时，首选编译器和 linker 的红帽开发人员工具集版本。
- Red Hat Enterprise Linux 6 和 7 中编译器的默认设置是 -std=gnu++98 到 4.1。也就是说，使用 GNU 扩展的 C++98。
- Red Hat Developer Toolset 6、6.1、7、7.1、8.1、9、9、9、9、9、10 和 10 中的编译器的默认设置是 -std=gnu++14。也就是说，使用 GNU 扩展的 C++14。

其它资源

- 应用程序兼容性 GUIDE
- 知识库解决方案 - Red Hat Enterprise Linux 中提供哪些版本？

- ***Red Hat Developer Toolset User Guide - C++ 兼容性***

15.8. EXAMPLE:使用 GCC 构建 C 程序

此示例显示构建最小 C 程序示例的具体步骤。

先决条件

- *了解 GCC 的使用*

步骤

1. *创建一个目录 hello-c，并改为其位置：*

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

2. *创建包含以下内容的文件 hello.c ：*

```
#include <stdio.h>

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

3. *使用 GCC 编译代码：*

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

对象文件 hello.o 已创建。

4. *从对象文件链接可执行文件 helloworld:*

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

5. 运行生成的可执行文件：

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

其它资源

- 第 18.2 节 “Example:使用 Makefile 构建 C 程序”

15.9. EXAMPLE:使用 GCC 构建 C++ 程序

本例显示了构建最小 C++ 程序的具体步骤。

先决条件

- 了解 GCC 的使用
- 了解 gcc 和 g++之间的区别

步骤

1. 创建一个目录 hello-cpp，并改为其位置：

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

2. 创建包含以下内容的 hello.cpp 文件：

```
#include <iostream>

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

3. 使用 g++ 编译代码：

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

对象文件 **hello.o** 已创建。

4. 从对象文件链接可执行文件 **helloworld:**

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

5. 运行生成的可执行文件：

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

第 16 章 在 GCC 中使用 LIBRARIES

本章论述了如何在代码中使用库。

16.1. 库命名约定

特殊文件名规则用于库：名为 **foo** 的库应该作为文件 **libfoo.so** 或 **libfoo.a** 存在。这个约定由 **gcc** 的链接输入选项自动理解，但输出选项并不包括：

- 当链接到库时，只能使用 **-l** 选项的 **-l** 选项指定 库的名称为 **-lfoo** ：

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

- 在创建库时，必须指定完整文件名 **libfoo.so** 或 **libfoo.a**。

其它资源

- 第 17.2 节 “soname Mechanism”

16.2. 静态和动态链接

开发人员可以选择使用完全编译的语言构建应用程序时使用静态或动态链接。这部分列出了区别，特别是在 **Red Hat Enterprise Linux** 中使用 **C** 和 **C++** 语言的情况。总之，红帽不建议在 **Red Hat Enterprise Linux** 的应用程序中使用静态链接。

静态和动态链接的比较

静态链接使库成为生成的可执行文件的一部分。动态链接将这些库保留为单独的文件。

可以通过多种方式比较动态和静态链接：

资源使用

静态链接会导致更大的可执行文件，其中包含更多代码。这个来自库的额外代码不能在系统上的多个程序间共享，这会增加文件系统在运行时的使用和内存用量。运行同一静态链接程序的多个进程仍将共享代码。

另一方面，静态应用需要较少的运行时重定位，从而减少启动时间，并且需要较少的专用常驻集

大小(RSS)内存。由于位置独立代码(PIC)引入的开销，用于静态链接的代码比动态链接更高效。

安全性

可更新提供 ABI 兼容性的动态链接库，而无需根据这些库更改可执行文件。这对于作为 Red Hat Enterprise Linux 的一部分提供的库来说尤为重要，红帽在此提供安全更新。对于任何此类库，强烈建议使用静态链接。

另外，负载均衡器随机化等安全措施无法与静态链接的可执行文件搭配使用。这进一步降低了生成的应用程序的安全性。

兼容性

静态链接似乎提供独立于操作系统提供的库版本的可执行文件。但是，大多数库依赖于其他库。使用静态链接时，此依赖项变得不灵活，因此会丢失正向和向后兼容性。静态链接可保证仅在构建可执行文件的系统上工作。

警告

从 GNU C 库(glibc)链接静态库的应用程序仍要求 glibc 作为动态库存在于系统中。另外，在应用程序运行时提供的 glibc 的动态库变体在连接应用程序时必须是一个完全相同的版本。因此，静态链接可保证仅在构建可执行文件的系统上工作。

支持覆盖范围

红帽提供的大多数静态库都位于 Optional 频道中，且不受红帽支持。

功能

某些库（特别是 GNU C 库(glibc)）在静态链接时提供减少的功能。

例如，当静态链接时，glibc 不支持 在同一程序中对 the dlopen（） 函数进行任何类型的调用。

由于列出的缺点，应不加成本避免静态链接，特别是整个应用程序和 glibc 和 libstdc++ 库。

注意

compat-glibc 软件包包含在 **Red Hat Enterprise Linux 7** 中，但它不是运行时软件包，因此不需要进行任何操作。它只是一个开发软件包，其中包含用于链接的标头文件和 **dummy** 库。这允许编译和链接软件包在旧的 **Red Hat Enterprise Linux** 版本中运行（使用 **compat-gcc-*** 依赖于这些标头和库）。有关使用这个软件包的更多信息，请运行：**rpm -qpi compat-glibc-***。

静态链路的原因

静态链接在某些情况下可能是合理的选择，例如：

- 未为动态链接启用的库
- 在空的 **chroot** 环境或容器中运行代码需要完全静态的链接。但是，红帽不支持使用 **glibc-static** 软件包的静态链接。

其它资源

- **Red Hat Enterprise Linux 7:**应用程序兼容性 **GUIDE**

16.3. 在 GCC 中使用库

库是可在您的程序中重复使用的代码软件包。**C** 或 **C++** 库由两个部分组成：

- 库代码
- 标头文件

使用库编译代码

标头文件描述库的接口：库中提供的函数和变量。编译代码需要标题文件中的信息。

通常，库的标题文件将放置在与您的应用代码不同的目录中。要告诉 **GCC** 标头文件的位置，请使用 **-I** 选项：

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

使用标头文件目录的实际路径替换 **include_path**。

I 选项可 多次使用，以添加包含标题文件的多个目录。查找标头文件时，会在 **-I** 选项中按照其外观的顺序搜索这些目录。

使用库链接代码

当链接可执行文件时，您的应用程序对象代码和库的二进制代码都必须可用。静态和动态库的代码以不同的格式显示：

- 静态库可用作存档文件。它们包含一组对象文件。归档文件的文件名为 **.a**。
- 动态库作为共享对象提供。它们是可执行文件的一种形式。共享对象的文件名为 **.so**。

要告诉 **GCC** 库的存档或共享对象文件的位置，请使用 **-L** 选项：

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

使用库目录的实际路径替换 **library_path**。

L 选项可 多次使用，以添加多个目录。查找库时，系统将按照其 **-L** 选项的顺序搜索这些目录。

选项顺序很重要：**GCC** 无法链接到库 **foo**，除非它知道这个库的目录。因此，使用 **-L** 选项来指定库目录，然后再使用 **-l** 选项链接到库。

在同一步骤中编译和链接代码使用库

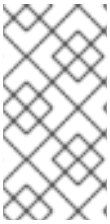
当一个 **gcc** 命令中允许编译并链接代码时，请一次性对上述两种情况使用 选项。

其它资源

- 使用 **GNU Compiler Collection(GCC)- 3.16 Options for Directory Search**
- 使用 **GNU Compiler Collection(GCC)- 3.15** 选项进行链接

16.4. 在 GCC 中使用静态库

静态库可用作包含对象文件的存档。链接后，它们将成为生成的可执行文件的一部分。

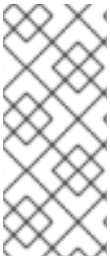


注意

由于各种原因，红帽不建议使用静态链接。请参阅 第 16.2 节 “静态和动态链接”。仅在需要时才使用静态链接，特别是红帽提供的库。

先决条件

- GCC 已安装在您的系统中
- 了解静态和动态链接
- 组成有效程序的一组源或对象文件，需要一些静态库 **foo**，且没有其他库
- **foo** 库作为 **libfoo.a** 文件提供，不提供用于动态链接的文件 **libfoo.so**。



注意

大多数作为 **Red Hat Enterprise Linux** 一部分的库都只支持动态链接。以下步骤仅适用于 没有为 动态链接启用的库。请参阅 第 16.6 节 “在 GCC 中使用 Both Static 和 Dynamic Libraries”。

步骤

要从源和对象文件中链接程序，请添加静态链接库 **foo**，名为 **libfoo.a** ：

1. 更改到包含您代码的目录。
2. 使用 **foo** 库的标头编译程序源文件：

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

使用包含 **foo** 库的标头文件的目录路径替换 **header_path**。

3. 将程序与 **foo** 库链接：

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

使用包含文件 **libfoo.a** 的目录的路径替换 **library_path**。

4. 要执行该程序，请运行：

```
$ ./program
```

小心

与 静态 链接相关的静态 **GCC** 选项禁止所有动态链接。相反，请使用 **-Wl**、**-Bstatic** 和 **-Wl,-Bdynamic** 选项更精确地控制链接器行为。请参阅 第 16.6 节 “在 GCC 中使用 Both Static 和 Dynamic Libraries”。

16.5. 在 GCC 中使用动态库

动态库以独立可执行文件的形式提供，在链接时间和运行时都是必需的。它们与您的应用程序的可执行文件保持独立。

先决条件

- **GCC** 已安装在系统上
- 组成有效程序的一组源或对象文件，需要一些动态库 **foo**，且没有其他库
- **foo** 库作为一个文件 **libfoo.so**

链接程序 **Against** 是一个动态库

根据动态库 **foo** 关联程序：

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

当程序与动态库链接时，生成的程序必须在运行时始终加载库。查找库有两个选项：

- 使用存储在可执行文件本身中的 **rpath** 值
- 在运行时使用 **LD_LIBRARY_PATH** 变量

使用存储在可执行文件中的 **rpath** 值

rpath 是一个特殊值，在链接文件时保存为可执行文件的一部分。之后，从可执行文件加载程序时，运行时链接器将使用 **rpath** 值来查找库文件。

使用 **GCC** 链接时，将路径 **library_path** 保存为 **rpath** ：

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

路径 **library_path** 必须指向包含文件 **libfoo.so** 的目录。

小心

在 **-Wl,-rpath=** 选项中逗号后没有空格

要稍后运行程序，请执行：

```
$ ./program
```

使用 **LD_LIBRARY_PATH** 环境变量

如果在程序的可执行文件中没有找到 **rpath**，则运行时链接器将使用 **LD_LIBRARY_PATH** 环境变量。必须根据共享库对象所在路径更改每个程序的值。

要运行没有设置 **rpath** 的程序，且 **library_path** 中存在库，请执行：

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

-

省略 **rpath** 值可提供灵活性，但每次程序运行时，需要设置 **LD_LIBRARY_PATH** 变量。

将库放入默认目录

运行时链接器配置指定多个目录作为动态库文件的默认位置。要使用此默认情况，请将您的库复制到适当的目录中。

有关动态链路器行为的完整描述超出了本文档的范围。如需更多信息，请参阅以下资源：

- 动态链接器的 **Linux** 手册页：

```
$ man ld.so
```

- **/etc/ld.so.conf** 配置文件的内容：

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

- 由动态链路器识别的库报告，无需额外配置，其中包括目录：

```
$ ldconfig -v
```

16.6. 在 GCC 中使用 BOTH STATIC 和 DYNAMIC LIBRARIES

有时需要静态地和某些库动态链接一些库。

先决条件

- 了解静态和动态链接

简介

GCC 识别 动态和静态库。遇到 **-lfoo** 选项后，**gcc** 将首先尝试查找包含 **foo** 库动态链接版本的共享对象（**a .so** 文件），然后查找包含库静态版本的存档文件**(.a)**。因此，这个搜索可能会导致以下情况：

- 只有找到共享对象，并动态地找到到它对应的 **gcc** 链接

- 只找到归档，并静态地找到指向它的 **gcc** 链接
- 找到共享对象和存档；默认情况下选择与共享对象进行动态链接
- 找不到共享对象或存档，且链接失败

由于这些规则，选择用于链接的库的静态或动态版本的最佳方法是仅让 **gcc** 找到该版本。在指定 **-L** 路径选项时，可以使用 或在指定 **-L**路径选项时保留包含库版本的目录，以控制 这一点。

此外，由于动态链接是默认的，因此必须明确指定链接的唯一情况是当一个存在两个版本的库都应静态链接。有两种可能的解决方案：

- 通过文件路径而不是 **-l** 选项指定静态库
- 使用 **-Wl** 选项将选项传递给链接器

通过文件指定静态库

通常，使用 **-l foo** 选项指示 **gcc** 与库**foo**链接。但是，可以指定到文件 **libfoo.a.a** 包含库的完整路径：

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

从文件 **extension .a**，**gcc** 将理解这是一个与程序链接的库。但是，指定库文件的完整路径是一个不太灵活的方法。

使用 **-Wl** 选项

gcc 选项 **-Wl** 是将选项传递给底层链接器的特殊选项。此选项的语法与其他 **gcc** 选项不同。**Wl** 选项加上以逗号分隔的 **linker** 选项列表，而其他 **gcc** 选项则需要以空格分隔的选项列表。**gcc** 使用的 **ld linker** 提供 **-Bstatic** 和 **-Bdynamic** 选项，以指定以下这个选项是否应分别链接或动态链接。将 **-Bstatic** 和库传递给 **linker** 后，默认的动态链接行为必须手动恢复，才能使以下库与 **-Bdynamic** 选项动态链接动态链接动态链接。

要链接程序，请运行以静态方式链接库**(lib first.a)**和 第二个 动态**(libsecond.so)**，请运行：

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

注意

GCC 可以配置为使用 default ld 以外的链接器。Wl 选项也适用于 gold 链接器。

其它资源

- 使用 GNU Compiler Collection(GCC)- 3.15 选项进行链接
- binutils 2.32 - 2.1 命令行选项的文档

第 17 章 使用 GCC 创建库

本章介绍创建库的步骤，并解释 Linux 操作系统用于库的必要概念。

17.1. 库命名约定

特殊文件名规则用于库：名为 foo 的库应该作为文件 libfoo.so 或 libfoo.a 存在。这个约定由 gcc 的链接输入选项自动理解，但输出选项并不包括：

- 当链接到库时，只能使用 -l 选项的 -l 选项指定 库的名称为 -lfoo ：

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

- 在创建库时，必须指定完整文件名 libfoo.so 或 libfoo.a。

其它资源

- 第 17.2 节 “soname Mechanism”

17.2. SONAME MECHANISM

动态加载的库（共享对象）使用名为 soname 的机制来管理库的多个兼容版本。

先决条件

- 了解动态链接和库
- 了解 ABI 兼容性概念
- 了解库命名规则
- 了解符号链接

问题简介

动态加载的库（共享对象）作为一个独立的可执行文件存在。这样便可在不更新依赖于它的应用程序的情况下更新库。但是，这个概念会出现以下问题：

- 识别库的实际版本
- 同一库的多个版本需要存在
- 每个多个版本的 **ABI** 兼容性的信号

soname Mechanism

为了解决这个问题，**Linux** 使用一种称为 **soname** 的机制。

库 **foo** 版本 **X.Y** 与其它版本兼容，与版本号中的 **X** 值相同。保持兼容性的小更改会增加数字 **Y**。破坏兼容性的主要变化会增加 **X**。

实际库 **foo** 版本 **X.Y** 作为文件 **libfoo.so.x** 存在。**y.**在库文件中，记录了一个 **soname**，值为 **libfoo.so.x** 以表示兼容性。

构建应用程序时，链接器通过搜索文件 **libfoo.so** 来查找库。必须存在具有此名称的符号链接，指向实际库文件。然后，链接器从库文件中读取 **soname**，并将其记录到应用程序可执行文件中。最后，**linker** 创建应用程序，以便使用 **soname** 而不是名称或文件名来声明对库的依赖项。

当运行时的动态链路器在运行前链接应用程序时，它会从应用程序的可执行文件中读取 **soname**。该 **soname** 是 **libfoo.so.x**。必须存在具有此名称的符号链接，指向实际库文件。无论版本的 **Y** 组件是什么，都允许载入库，因为 **soname** 不会改变。

注意

版本号的 **Y** 组件不仅限于一个数字。另外，一些库在其名称中对版本进行编码。

从文件中读取 **soname**

显示库文件 **somelibrary** 的 **soname**：

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

使用您要检查的库的实际文件名替换 **somelibrary**。

17.3. 使用 GCC 创建动态库

通过更轻松地更新库代码，动态链接的库（共享对象）允许通过代码重复使用和增强安全性。这部分论述了从源构建和安装动态库的步骤。

先决条件

- 了解 **soname** 机制
- **GCC** 已安装在系统上
- 库的源代码

步骤

1. 使用库源更改到 目录。
2. 使用独立于位置的代码选项 **-fPIC** 将每个源文件编译到对象文件中：

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

   对象文件具有与原始源代码文件相同的文件名，但它们的扩展名是 **.o**。
3. 从对象文件链接共享库：

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

   使用的主版本号是 **X**，次版本号为 **Y**。
4. 将 **libfoo.so.x.y** 文件复制到系统动态链路器可以找到它的适当位置。在 **Red Hat Enterprise**

Linux 中，库的目录是 **/usr/lib64** ：

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

请注意，您需要 **root** 权限才能操作此目录中的文件。

5. 为 **soname** 机制创建符号链接结构：

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

其它资源

- **Linux** 文档项目 **-** 计划库 **HOWTO - 3.**共享库

17.4. 使用 GCC 和 AR 创建静态库

通过将对象文件转换为特殊类型的存档文件，可以创建用于静态链接的库。

注意

出于安全原因，红帽不建议使用静态链接。只在需要时才使用静态链接，特别是红帽提供的库。请参阅 **第 16.2 节 "静态和动态链接"**。

先决条件

- **GCC 和 binutils** 已安装在系统中
- 了解静态和动态链接
- 包含要作为库共享函数的源文件

步骤

1. 使用 **GCC** 创建中间对象文件。

-

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

根据需要附加更多源文件。生成的对象文件共享文件名，但使用 **.o** 文件名称扩展名。

2. 使用 **binutils** 软件包中的 **ar** 工具，将对象文件转换为静态库（存档）。

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

文件 **libfoo.a** 被创建。

3. 使用 **nm** 命令检查生成的归档：

```
$ nm libfoo.a
```

4. 将静态库文件复制到适当的目录。

5. 与库链接时，**GCC** 将自动从 **.a** 文件扩展名中识别库是静态链接的存档。

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

其它资源

- **ar** 工具的 **Linux** 手册页：

```
$ man ar
```

第 18 章 使用 MAKE 管理更多代码

GNU Make 实用程序通常缩写为 Make，是控制从源文件生成可执行文件的工具。自动决定 复杂程序中的哪些部分已更改，需要重新编译。make 使用名为 Makefile 的配置文件来控制程序的构建方式。

18.1. GNU MAKE 和 MAKEFILE 概述

要从特定项目的源文件创建可用的表单（通常是可执行文件），请执行几个必要的步骤。记录操作及其顺序，以便稍后重复操作。

Red Hat Enterprise Linux 包含 GNU make，这是专为此目的设计的构建系统。

先决条件

- 了解编译和链接的概念

GNU make

GNU make 读取 Makefile，其中包含描述构建过程的说明。Makefile 包含多个 规则，它们描述通过特定操作（方法）满足特定条件（目标）的方式。规则可以分层依赖于另一条规则。

运行不带任何选项的 make 即可在当前目录中查找 Makefile 并尝试访问默认目标。实际的 Makefile 文件名可以是 Makefile、makefile 和 GNUmakefile 之一。默认目标由 Makefile 内容决定。

Makefile 详情

makefile 使用相对简单的语法来定义 变量 和 规则,这些变量和规则由一个 target 和一个 recipe 组成。如果执行规则，则目标指定输出。具有配方的行必须以 tab 字符开头。

通常，Makefile 包含用于编译源文件的规则、用于链接结果对象文件的规则，以及充当层次结构顶部的入口点的目标。

请考虑以下 Makefile 用于构建由单个文件 hello.c 组成的 C 程序。

```
all: hello

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

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

这将指定，要访问 所有目标，需要 **hello** 文件。要获得 **hello**，您需要 **hello.o** （由 **gcc** 链接），后者从 **hello.c** 创建（由 **gcc**编译）。

目标 **all** 是默认目标，因为它是不是以句点开头的第一个目标。如果当前目录包含这个 **Makefile**，则不带任何参数运行 **make** 会完全运行 **make**。

典型的 **Makefile**

较典型的 **Makefile** 使用变量来常规调整步骤，并添加目标**"clean"**，这将删除除源文件以外的所有对象。

```
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)
```

向此类 **Makefile** 添加更多源文件需要将它们添加到定义 **SOURCE** 变量的行。

其他资源

- **GNU make**：简介 - **2 Makefile** 简介
- 第 **15** 章 使用 **GCC** 构建代码

18.2. EXAMPLE:使用 MAKEFILE 构建 C 程序

*按照以下示例中的步骤，使用 **Makefile** 来构建示例 **C** 程序。*

先决条件

- *了解 **Makefile** 和 **make***

步骤

1. *创建 **hellomake** 目录并更改到此目录：*

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

2. *创建包含以下内容的文件 **hello.c** ：*

```
#include <stdio.h>

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

3. *创建包含以下内容的文件 **Makefile** ：*

```
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)
```

小心

Makefile 配方行必须以制表符字符开头。从浏览器中复制以上文本时，您可以粘贴空格。手动更正此更改。

4. 运行 **make** ：

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

这将创建一个可执行文件 **hello**。

5. 运行可执行文件 **hello** ：

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

6. 运行 **Makefile** 目标 **clean** 以删除创建的文件：

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

其它资源

- 第 15.8 节 “Example:使用 GCC 构建 C 程序”
- 第 15.9 节 “Example:使用 GCC 构建 C++ 程序”

18.3. 制作的文档 资源

有关 **make** 的更多信息，请参阅以下列出的资源。

安装的文档

- 使用 **man** 和信息 工具 查看系统中安装的 **man page** 和信息页面：

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

在线文档

- *由自由软件基金会托管的 **GNU Make** 手册*
- ***Red Hat Developer Toolset** 用户指南 - **GNU make***

第 19 章 使用 ECLIPSE IDE 进行 C 和 C++ 应用程序开发

有些开发人员更喜欢使用 IDE 而不是一组命令行工具。红帽为开发 C 和 C++ 应用程序提供 Eclipse IDE。

使用 Eclipse 开发 C 和 C++ 应用程序

Eclipse IDE 及其用于开发 C 和 C++ 应用程序的详细描述超出了本文档的范围。请参考下面链接的资源。

其它资源

- 使用 Eclipse
- Eclipse 文档 - C/C++ 开发用户指南

部分 V. 调试应用程序

调试应用程序是一个广泛的主题。这部分为开发人员提供了在多种情况下用于调试的最常见技术。

第 20 章 调试正在运行的应用程序

本章介绍了用于调试应用（可根据需要多次）在开发人员直接访问的机器上执行的技术。

20.1. 使用调试信息启用调试

要调试应用和库，需要调试信息。以下小节描述了如何获取此信息。

20.1.1. 调试信息

在调试任何可执行代码时，两种信息可以允许工具，并扩展程序员以理解二进制代码：

- 源代码文本
- 有关源代码文本与二进制代码的关系的描述

这称为调试信息。

Red Hat Enterprise Linux 对可执行二进制文件、共享库或调试信息文件使用 **ELF** 格式。在这些 **ELF** 文件中，**DWARF** 格式用于保存调试信息。

DWARF 符号由 **readelf -w file** 命令读取。

小心

UNIX 偶尔会使用 **STABS**。**STABS** 是一种较旧、更少的格式。红帽不鼓励其使用。**GCC** 和 **GDB** 只支持 **STABS** 生产与消费。**Valgrind** 和 **elfutils** 等其他工具根本不支持 **STABS**。

其它资源

- **DWARF** 调试标准

20.1.2. 使用 GCC 启用 C 和 C++ 应用程序

由于调试信息较大，因此默认情况下，不会包含在可执行文件中。要为您的 **C** 和 **C++** 应用程序进行调试，您必须明确指示编译器创建调试信息。

使用 **GCC** 启用调试信息

要在编译和链接代码时使用 **GCC** 创建调试信息，请使用 **-g** 选项：

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

- 由编译器和链接器执行的优化可能会导致可执行代码难以与原始源代码相关：变量可以被优化，取消滚动，将操作合并到周围的代码中。这会影响调试。要改进调试体验，请考虑使用 **-Og** 选项设置优化。但是，更改优化级别会更改可执行代码，并可能会更改实际行为，以便删除一些程序错误。

- **f compare-debug GCC** 选项测试 **GCC** 使用调试信息编译的代码，无需调试信息。如果生成的两个二进制文件相同，则测试将通过。此测试可确保可执行代码不受任何调试选项的影响，进一步确保调试代码中没有隐藏的错误。请注意，使用 **-fcompare-debug** 选项可显著增加编译时间。有关这个选项的详情，请查看 **GCC** 手册页。

其它资源

- 第 **20.1** 节 “使用调试信息启用调试”

- 使用 **GNU Compiler Collection(GCC)- 3.10** 选项用于调试您的程序

- 使用 **GDB** 进行调试 **— 18.3 Debugging Information in Separate Files**

- **GCC** 手册页：

  ```
  $ man gcc
  ```

20.1.3. debuginfo Packages

debuginfo 软件包包含程序和库的调试信息和调试源代码。

先决条件

- 了解调试信息

debuginfo Packages

对于从 **Red Hat Enterprise Linux** 软件仓库安装的软件包中安装的应用程序和库，您可以获得调试信息和调试源代码作为单独的通过另一个频道提供的调试信息软件包。**debuginfo** 软件包包含 **.debug** 文件，其中包含 **DWARF debuginfo** 和用于编译二进制软件包的源文件。**debuginfo** 软件包内容安装到 **/usr/lib/debug** 目录中。

debuginfo 软件包仅对具有相同名称、版本、发行版本和架构的二进制软件包提供调试信息有效：

- 二进制 软件包：**packagename-version-release.architecture.rpm**
- **debuginfo package: packagename-debuginfo-version-release.architecture.rpm**

20.1.4. 使用 GDB 获取应用程序或库的调试信息软件包

GNU Debugger(GDB)会自动识别缺少的调试信息并解决软件包名称。

先决条件

- 您要调试的应用程序或库安装在系统中
- **GDB** 已安装在系统中
- **debuginfo-install** 工具安装在系统中

步骤

1. 启动连接到要调试的应用程序或库的 **GDB**。**GDB** 自动识别缺少的调试信息，并建议要运行的命令。

```
$ 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. 退出 **GDB**，但不继续操作：键入 **q** 和 **Enter**。

```
(gdb) q
```

3. 运行 **GDB** 建议的命令来安装所需的 **debuginfo** 软件包：

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

为应用程序或库安装 **debuginfo** 软件包会为所有依赖项安装 **debuginfo** 软件包。

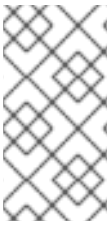
4. 如果 **GDB** 无法推荐 **debuginfo** 软件包，请按照 第 **20.1.5** 节 “手动 获取 应用程序或库的调试信息软件包” 中的步骤操作。

其它资源

- **Red Hat Developer Toolset** 用户指南 - 第 **1.5** 章，安装调试信息
- 红帽知识库解决方案 - 如何为 **RHEL** 系统下载或安装 **debuginfo** 软件包？

20.1.5. 手动 获取 应用程序或库的调试信息软件包

要手动选择（安装） **debuginfo** 软件包（用于安装），请找到可执行文件 并查找该软件包来安装该软件包。



注意

使用 **GDB** 确定安装的软件包更好。只有在 **GDB** 无法建议安装软件包时，才使用此手动步骤。

先决条件

- 应用程序或库必须安装在系统中
- **debuginfo-install** 工具必须在系统中可用

步骤

1. 查找应用程序或库的可执行文件。

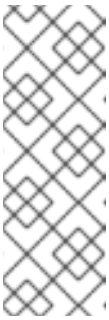
   a. 使用 哪个命令 查找应用文件。

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

   b. 使用 **locate** 命令查找库文件。

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

   如果调试包含的错误消息的原始原因，请在其文件名中选择库具有相同额外数字的结果。若有疑问，请尝试遵循库文件名不包含其他数字的结果。

   注意

   locate 命令由 **mlocate** 软件包提供。安装它并启用其使用：

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

2. 使用文件路径，搜索提供该文件的软件包。

   ```
   # 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
   ```

   输出以格式 名称提供软件包列表 **-**版本。分发。架构。在这一步中，只有 软件包名称 很重要，因为 **yum** 输出中显示的版本可能不是实际安装的版本。

重要

如果此步骤没有生成任何结果，则无法决定哪个软件包提供二进制文件，这个过程会失败。

3. 使用 **rpm** 低级别软件包管理工具查找系统上安装哪个软件包版本。使用软件包名称作为参数：

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

输出提供以格式 名称（版本 ）的已安装软件包的详细信息。分发.架构。

4. 使用 **debuginfo -install** 实用程序安装 **debuginfo** 软件包。在命令中，使用软件包名称以及您在上一步中确定的其他详情：

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

为应用程序或库安装 **debuginfo** 软件包会为所有依赖项安装 **debuginfo** 软件包。

其它资源

- **Red Hat Developer Toolset** 用户指南 - 安装调试信息
- 知识库文章 - 如何下载或安装 **RHEL** 系统的 **debuginfo** 软件包？

20.2. 使用 GDB 检查应用程序的内部状态

若要找出应用无法正常工作的原因，请控制其执行并使用调试器检查其内部状态。本节论述了如何将 **GNU Debugger(GDB)**用于此任务。

20.2.1. GNU Debugger(GDB)

debugger 是一个可以控制代码执行和检查代码状态的工具。这个功能用于调查程序中出现的情况以及原因。

Red Hat Enterprise Linux 包含 GNU debugger(GDB)，它通过命令行用户界面提供此功能。

对于到 GDB 的图形 frontend，请安装 Eclipse 集成开发环境。请参阅使用 Eclipse。

GDB 能力

单个 GDB 会话可以调试：

- 多线程和分叉程序
- 一次多个程序
- 远程机器或带有 gdbserver 实用工具通过 TCP/IP 网络连接连接的容器中的程序

调试要求

要调试任何可执行代码，GDB 需要相应的调试信息：

- 对于由您开发的程序，您可以在构建代码时创建调试信息。
- 对于从软件包安装的系统程序，必须安装对应的 debuginfo 软件包。

20.2.2. 将 GDB 附加到进程

为了检查进程，必须将 GDB 附加到 进程。

先决条件

- 在系统中必须安装 GDB

使用 GDB 启动程序

当该程序没有作为进程运行时，使用 GDB 启动它：

```
$ gdb program
```

使用文件名或程序路径替换 程序。

GDB 开始执行程序。您可以使用 **run** 命令开始执行进程，然后设置断点和 **gdb** 环境。

将 GDB 附加到 Already Running Process

将 **GDB** 附加到已作为进程运行的程序：

1. 使用 **ps** 命令查找进程 **id(pid)**：

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

使用文件名或程序路径替换 程序。

2. 将 **GDB** 附加到此过程：

```
$ gdb program -p pid
```

使用到 程序的 文件名或路径替换 **program**，将 **pid** 替换为 **ps** 输出中的实际进程 **ID** 号。

将 Already Running GDB 附加到 Already Running Process

将已在运行的 **GDB** 附加到已在运行的程序：

1. 使用 **shell GDB** 命令运行 **ps** 命令并查找程序的进程 **id(pid)**：

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

使用文件名或程序路径替换 程序。

2. 使用 **attach** 命令将 **GDB** 附加到程序：

```
(gdb) attach pid
```

将 **pid** 替换为 **ps** 输出中的实际进程 **ID** 号。

注意

在某些情况下，**GDB** 可能无法找到对应的可执行文件。使用 **file** 命令指定路径：

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

其它资源

- 使用 **GDB** 进行调试 **- 2.1** 调用 **GDB**
- 使用 **GDB** 进行调试 **- 4.7** 调试 **Already-running** 进程

20.2.3. 使用 GDB 逐步处理程序代码

GDB 调试器附加到程序后，您可以使用多个命令来控制程序的执行。

先决条件

- 在系统中必须安装 **GDB**
- 您必须有所需的调试信息可用：
  - 程序使用调试信息编译和构建，或者
  - 已安装相关的 **debuginfo** 软件包
- **GDB** 已附加到要调试的程序

通过代码逐步进行 **GDB** 命令

r (run)

开始执行 程序。如果通过参数执行了 运行，这些参数将作为程序正常启动，传递到可执行文

件。用户通常在设置断点后发出此命令。

开始

启动程序的执行过程，并在主要功能开始时停止。如果以参数 开头，则这些参数将被传递到可执行文件，就像程序正常启动一样。

c （续）

继续从当前状态执行 程序。程序的执行将继续，直到以下条件之一变为 **true**：

- 到达断点
- 满足指定条件
- 程序收到信号
- 发生错误
- 程序终止

n (next)

此命令的另一个常见名称为 步骤。继续从当前状态执行程序，直到达到当前源文件中的下一行代码。程序的执行将继续，直到以下条件之一变为 **true**：

- 到达断点
- 满足指定条件
- 程序收到信号
- 发生错误

- 程序终止

s （步骤）

这个命令的另一个常见名称为 步骤。**step** 命令在当前源文件中每个连续代码行停止执行。但是，如果当前在包含 函数调用 的源行停止执行，**GDB** 会在输入函数调用后停止执行（而不是执行它）。

直到 位置

继续执行，直到达到位置选项指定的代码 位置。

Fini （芬兰）

恢复执行程序，并在执行从函数返回时停止执行。程序的执行将继续，直到以下条件之一变为 **true**：

- 到达断点
- 满足指定条件
- 程序收到信号
- 发生错误
- 程序终止

q (quit)

终止执行并退出 **GDB**。

其它资源

- 第 20.2.5 节 “使用 GDB Breakpoints 在定义的代码位置停止执行”
- 使用 **GDB** 进行调试 - 4.2 启动您的程序

- 使用 GDB 进行调试 - 5.2 Continuing and Stepping

20.2.4. 使用 GDB 显示程序内部值

显示程序内部变量的值对于了解程序正在执行的操作非常重要。GDB 提供了多个命令，可用于检查内部变量。这部分论述了这些命令的最有用的信息。

先决条件

- 了解 GDB 调试器

显示程序的内部状态的 GDB 命令

P （打印）

显示给定参数的值。通常，参数是任何复杂性的变量的名称，从简单的单个值到结构。参数也可以是在当前语言中有效的表达式，包括使用程序变量和库函数，或者正在测试的程序中定义的函数。

可以使用 print 命令对数据结构（如类、结构）进行自定义显示，使用设计器 Python 或 Guile 脚本来扩展 GDB。

BT （回溯）

显示用于到达当前执行点的函数调用链，或在执行被信号前使用的功能链。这对于调查带有困难原因的严重错误（如分段错误）非常有用。

在 backtrace 命令中添加 完整 选项也会显示本地变量。

可以使用 框架过滤 Python 脚本来扩展 GDB，以自定义使用 bt 和 info 帧 命令显示的数据。术语 帧 指的是与单个函数调用关联的数据。

info

info 命令是一种通用命令，用于提供关于各种项目的信息。它取指定项目的选项。

- info args 命令显示目前为所选帧的函数调用的参数。
- info locals 命令在当前选定的帧中显示本地变量。

如需可能的项目列表，请在 **GDB** 会话中运行命令 帮助信息 ：

```
(gdb) help info
```

l （列表）

显示程序停止的源代码中的 行。此命令仅在程序执行停止时才可用。虽然不能严格显示内部状态的命令，但 **list** 有助于用户了解程序的执行下一步中对内部状态的更改。

其它资源

- **Red Hat Developer** 博客条目 - **GDB Python API**
- 使用 **GDB** 进行调试 - **10.9 Pretty Printing**

20.2.5. 使用 GDB Breakpoints 在定义的代码位置停止执行

在很多情况下，让程序在达到某一行代码之前会执行非常有优势。

先决条件

- 了解 **GDB**

在 **GDB** 中使用 **Breakpoints**

breakpoints 是告知 **GDB** 停止执行程序的标记。断点最常与源代码行关联：放置断点需要指定源文件和行号。

- 放置断点 ：
  - 指定源代码 文件的名称以及该文件中 的行 ：

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

  - 如果文件不存在，则使用当前执行点的源文件的名称：

    ```
    (gdb) br line
    ```

- 或者，使用函数名称来放置 **breakpoint**：

```
(gdb) br function_name
```

- 在任务发生多次迭代后，程序可能会遇到错误。指定暂停执行的额外 条件 ：

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

使用 **C** 或 **C++** 语言中的条件替换 条件。文件 和 行 的含义与上方相同。

- 检查 所有断点和监控点的状态：

```
(gdb) info br
```

- 使用 **info br** 的输出中显示的 数量 来删除 断点：

```
(gdb) delete number
```

- 在给定位置 删除 断点：

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

其它资源

- 使用 **GDB** 调试 — **5.1 Breakpoints, Watchpoints, and Catchpoints**

20.2.6. 使用 GDB Watchpoints 在数据访问和更改时停止执行

在很多情况下，让程序执行直到访问某些数据或被访问为止。本节列出了最常见的监视点。

先决条件

- 了解 **GDB**

在 **GDB** 中使用 **Watchpoints**

Watchpoints 是指示 **GDB** 停止执行程序的标记。**Watchpoints** 与数据关联：放置监视点需要指定描述变量、多个变量或内存地址的表达式。

- 为数据 更改 放置 监视点（写入）：

```
(gdb) watch expression
```

将 **expression** 替换为描述您要监视内容的表达式。对于变量，表达式 等于变量的名称。

- 为 数据访问 放置 监视点（读取）：

```
(gdb) rwatch expression
```

- 为 任何 数据访问 放置 监视点（读写）：

```
(gdb) awatch expression
```

- 检查 所有监视点和断点的状态：

```
(gdb) info br
```

- 删除 观察点：

```
(gdb) delete num
```

将 **num** 选项替换为 **info br** 命令报告的编号。

其它资源

- 使用 **GDB** 进行调试 **- 5.1.2** 设置 **Watchpoints**

20.2.7. 使用 GDB 调试验证或线程程序

些程序使用分叉或线程来实现并行代码执行。调试多个同步执行路径需要特殊考虑。

先决条件

- *了解 **GDB** 调试器*
- *了解进程分叉和线程的概念*

*使用 **GDB** 调试已验证程序*

***Forking** 是程序（父级）创建自身（子级）的独立副本的情况。使用以下设置和命令将 **GDB** 的反应影响到发生 **fork** 的修改：*

- ***following -fork-mode** 设置控制 **GDB** 是否紧跟 父项还是分叉后面的子级。*

  *设置 **follow-fork-mode parent***

  在分叉后，调试父进程。这是默认值。

  *设置 **follow-fork-mode child***

  在分叉后，调试子进程。

  *显示 **follow-fork-mode***

  *显示以下 **fork-mode** 的当前设置。*

- ***set detach-on-fork** 设置控制 **GDB** 是否保持对其他（未跟随）进程的控制，或使其保持运行。*

  *设置 **detach-on-fork on***

  不后的进程（取决于后续模式的值）将分离并运行。这是默认值。

  *设置 **detach-on-fork off***

  ***GDB** 保持对这两个进程的控制。随后的进程（取决于 **follow-fork-mode**的值）像往常一样被调试，而另一个则暂停。*

  *显示 **detach-on-fork***

  *显示 **detach-on-fork** 的当前设置。*

*使用 **GDB** 调试线程程序*

GDB 能够调试各个线程，并且能够独立操作和检查它们。要使 GDB 仅停止检查的线程，请使用命令设置不停止并在其 上设置 target-async。您可以将这些命令添加到 .gdbinit 文件。启用该功能后，GDB 已准备好执行线程调试。

GDB 使用 当前线程 的概念。默认情况下，命令仅应用于当前线程。

info 线程

显示具有 **id** 和 **gid** 号的线程列表，表示当前的线程。

线程 **ID**

将指定 **id** 设为当前线程的线程。

线程应用 **ids** 命令

将命令 应用到按 **ID** 列出的所有线程。 **ids** 选项是空格分隔的线程 **ID** 列表。特殊值 **all** 会将命令应用到所有线程。

如果 条件中断 位置 线程 **id**

在 特定位置 设置一个断点，且仅限于线程号 **id** 的 特定条件。

watch 表达式 线程 **ID**

设定仅由 表达式 定义的监视点，用于线程编号 **ID**。

命令**&**

执行 命令，并立即返回到 GDB 提示符 (gdb)，并在后台继续执行代码。

中断

在后台停止执行。

其它资源

- 使用 GDB 进行 调试 - 使用多个线程调试程序 4.10
- 使用 GDB 进行调试 - 4.11 Debugging Forks

20.3. 记录应用程序互动

应用程序的可执行代码与操作系统和共享库的代码交互。记录这些交互的活动日志可在不调试实际应用程序代码的情况下充分洞察应用的行为。另外，分析应用程序的交互可帮助确定存在错误清单的条件。

20.3.1. 记录应用程序交互的有用工具

红帽企业 Linux 提供用于分析应用程序的交互的多种工具。

strace

strace 工具可跟踪（和修改）应用程序与 Linux 内核之间的交互：系统调用、信号发送和进程状态更改。

- strace 输出的详细内容，并说明调用良好，因为 strace 解释底层内核代码的参数和结果。将数字转换为对应的常数名称、对标记列表扩展的位明组合标志，指向字符数组的指针以提供实际字符串，等等。但是，对最新内核功能的支持可能会缺乏。
- 使用 strace 不需要任何特定设置，除了设置日志过滤器外。
- 使用 strace 跟踪应用代码可能会导致应用程序的执行速度显著下降。因此，strace 不适用于很多生产部署。作为替代方案，在这样的情形中使用 SystemTap。
- 您可以限制 traced 系统调用和信号列表，以减少捕获的数据量。
- strace 仅捕获内核用户空间交互，而不 trace 库调用，例如：考虑对追踪库调用使用 ltrace。

ltrace

ltrace 工具支持将应用的用户空间调用记录到共享对象（动态库）。

- ltrace 可启用对任何库的追踪调用。
- 您可以过滤跟踪的调用来减少捕获的数据量。

- 使用 **ltrace** 不需要任何特定设置，除了设置日志过滤器外。

- **ltrace** 是轻量级而快速的，为 **strace** 提供替代 **strace** 的替代方案：可以使用 **ltrace** 追踪库 中的对应 接口，而不是通过 **strace** 跟踪内核功能。但请注意，**l trace** 在 **syscall** 跟踪时可能不太精确。

- **ltrace** 能够对有限库调用集进行解码参数：相关配置文件中定义了原型的调用。作为 **ltrace** 软件包的一部分，提供了一些 **libacl**、**libc** 和 **libm** 调用和系统调用的原型。**ltrace** 输出主要仅包含原始数字和指针。**ltrace** 输出的解释通常需要咨询输出中存在的库的实际接口声明。

SystemTap

SystemTap 是一个检测平台，用于在 **Linux** 系统上探测运行中的进程和内核活动。**SystemTap** 使用自己的脚本语言来编程自定义事件处理程序。

- 与使用 **strace** 和 **ltrace** 相比，编写日志意味着在初始设置阶段有更多工作。但是，该脚本功能将 **SystemTap** 的用处扩展至生成日志之外。

- **SystemTap** 通过创建和插入内核模块来工作。**SystemTap** 的使用效率更高，不会自行造成系统或应用执行的显著下降。

- **SystemTap** 附带了一组使用示例。

GDB

GNU Debugger 主要用于调试，而不是日志记录。但是，其某些功能即使在应用程序交互是关注的主要活动的情况下也使其有用。

- 借助 **GDB**，可以将交互事件的捕获和后续执行路径的即时调试轻松结合在一起。

- 在其他工具的初始识别问题后，**GDB** 最适合分析对非经常性或具体事件的响应。在任何有频繁事件的情况下使用 **GDB** 都会变得效率低下甚至不可能。

其它资源

- *Red Hat Enterprise Linux SystemTap Beginners Guide*
- *Red Hat Developer Toolset 用户指南*

20.3.2. 通过 strace 监控应用程序的系统调用

strace 工具可跟踪（和可选修改）与 **Linux** 内核之间的交互：系统调用、信号发送和进程状态更改。

先决条件

- **strace** 安装在系统中
  - 要安装 **strace**，以 **root** 用户身份运行：

```
# yum install strace
```

步骤

请注意，**strace** 的追踪规格语法提供正则表达式和 **syscall** 类，以帮助识别系统调用。

1. 运行 或附加到您想要监控的进程。
   - 如果您要监控的程序没有运行，请启动 追踪 并指定 程序 ：

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

上述示例中使用的选项不是强制性的。在需要时使用：

- **f** 选项是 "跟进分叉"的缩写。这个选项跟踪由 **fork**、**vfork** 和克隆系统调用创建的子项。
- **v** 或 **-e abbrev=none** 选项可禁用输出的缩写，省略了各种结构字段。
- **tt** 选项是 **-** **t** 选项的一个变体，使用绝对时间戳为每一行加上前缀。通过 **-tt** 选

项，打印的时间包括 **microseconds**。

- **t** 选项打印行末尾的每个系统调用所花费的时间长度。

- **yy** 选项是 **-y** 选项的一个变体，它可以打印与文件描述符编号关联的路径。**yy** 选项不仅打印路径，还打印与套接字文件描述符和设备描述符关联的协议特定信息，或者与设备文件描述符关联的字符设备编号。

- **s** 选项控制 要打印的最大字符串大小。请注意，文件名不被视为字符串，并且始终全部打印。

- **-e trace** 控制追踪的系统调用集合。

使用要显示的系统调用列表替换 **call**。如果缺少 调用，**strace** 将显示所有系统调用。**strace(1)** 手册页中提供了部分系统调用的 **Shorthands**。

- 如果程序已在运行，找到其进程 **ID(pid)**，并为它附加 **strace** ：

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

- 如果您不想跟踪任何已分叉的进程或线程，请不要使用 **-f** 选项。

2. **strace** 显示由应用发出的系统调用及其详细信息。

在大多数情况下，如果未设置系统调用过滤器，应用及其库都会立即发出大量调用和 **strace** 输出。

3. 当所有 **traced** 进程退出时，**strace** 会退出。要在跟踪程序退出前终止监控，请按 **Ctrl+C**。

- 如果 **strace** 启动 程序，它将向正在启动的程序发送终止信号**(SIGINT)**。但请注意，该程序依次可能忽略该信号。

- 如果您 将 **trace** 附加到 已在运行的程序，程序会一起终止 。

4. 分析应用执行的系统调用列表。

   - 当调用返回错误时，日志中会出现与资源访问或可用性相关的问题。
   - 传递给系统调用和调用序列模式的值可让您了解应用程序的原因。
   - 如果应用程序崩溃，重要信息可能是在日志的末尾。
   - 输出中包含大量额外的信息。但是，您可以构建一个更精确的过滤器并重复该过程。

备注

- 益处是查看输出并将其保存到文件中。要做到这一点，请运行 **tee** 命令：

```
$ strace ...-o |tee your_log_file.log>&2
```

- 要查看与不同进程对应的独立输出，请运行：

```
$ strace ... -ff -o your_log_file
```

  带有进程 **ID** 的进程的输出**(pid)**将保存在 **your_log_file.pid** 中。

其它资源

- **strace(1)** 手册页。
- 知识库文章 **-** 如何使用 **strace** 跟踪命令发出的系统调用？
- **Red Hat Developer Toolset User Guide -** **strace**

20.3.3. 通过 ltrace 监控应用程序的库函数调用

ltrace 工具支持对应用所进行的调用的监控到库中可用的功能（共享对象）。

先决条件

- ltrace 在系统中安装

步骤

1. 识别感兴趣的库和功能（如果可能）。

2. 如果您要监控的程序没有运行，请启动 ltrace 并指定 程序 ：

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

使用 -e 和 -l 选项过滤输出：

- 请提供函数名称，使其显示为 函数。e 函数 选项可多次使用。如果左侧，l trace 将显示所有功能的调用。

- 您可以使用 -l 库 选项指定整个库，而不是指定函数。这个选项的行为与 -e 函数 选项类似。

如需更多信息，请参阅 ltrace(1)_ 手册页。

如果程序已在运行，请查找其进程 ID(pid)并附加 ltrace 到其中：

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

如果您不想跟踪任何已分叉的进程或线程，请离开 -f 选项。

3. **ltrace** 显示应用发出的库调用。

   在大多数情况下，如果没有设置过滤器，应用程序将产生大量调用，如果未设置过滤器，则立即出现 **ltrace** 输出。

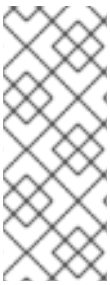
4. 当程序 退出时，**ltrace** 会退出。

   要在 **traced** 程序退出前终止监控，请按 **enter ctrl+C**。

   - 如果 **ltrace** 启动了程序，程序会一起使用 **ltrace** 终止。
   - 如果您将 **ltrace** 附加到已在运行的程序，程序会一起终止 **ltrace**。

5. 分析应用执行的库调用列表。

   - 如果应用程序崩溃，重要信息可能是在日志的末尾。
   - 输出中包含大量不必要的信息。但是，您可以构建一个更精确的过滤器并重复该过程。



注意

益处是查看输出并将其保存到文件中。使用 **tee** 命令实现这一点：

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

其它资源

- **strace(1)** 手册页
- **Red Hat Developer Toolset User Guide - ltrace**

20.3.4. 使用 SystemTap 监控应用程序的系统调用

SystemTap 工具支持为内核事件注册自定义事件处理程序。与 **strace** 相比，使用起来更难于使用，但 **SystemTap** 效率更高，并启用更复杂的处理逻辑。

先决条件

- 在系统中安装了 **SystemTap**

步骤

1. 使用以下内容创建 **my_script.stp** 文件：

```
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. 查找您要监控的进程的进程 **ID(pid)**：

```
$ ps -aux
```

3. 使用脚本运行 **SystemTap** ：

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

pid 的值是进程 **ID**。

脚本编译到之后载入的内核模块。这在输入命令和获取输出之间引入了一些延迟。

4. 当进程执行系统调用时，调用名称及其参数将打印到终端。

5. 当进程终止或按 **Ctrl+C** 时，脚本将退出。

其它资源

- **SystemTap** 入门指南
- **SystemTap Tapset** 参考
- 一个大型 **SystemTap** 脚本（约**roximates strace** 功能）作为 **/usr/share/systemtap/examples/process/strace.stp** 可用。要运行脚本：

```
# stap --example strace.stp -x pid
```

或

```
# stap --example strace.stp -c "cmd args ..."
```

20.3.5. 使用 GDB 互动应用程序系统调用

GDB 启用在执行程序的过程中停止各种情况的执行。要在程序执行系统调用时停止执行，请使用 **GDB** 捕获点。

先决条件

- 了解 **GDB** 断点
- **GDB** 已附加到程序

使用 **GDB** 在系统调用时停止程序执行

1. 设置捕获点：

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

命令 **catch syscall** 设置一个特殊类型的断点，在程序执行系统调用时停止执行。

syscall-name 选项指定调用的名称。您可以为各种系统调用指定多个捕获点。退出 **syscall-name** 选项会导致 **GDB** 在任何系统调用时停止。

2. 如果程序还没有开始执行，请启动它：

```
(gdb) r
```

如果程序执行只停止，请恢复它：

```
(gdb) c
```

3. **GDB** 在程序执行任何指定系统调用后停止执行。

其它资源

- 第 **20.2.4** 节 “使用 **GDB** 显示程序内部值”
- 第 **20.2.3** 节 “使用 **GDB** 逐步处理程序代码”
- 使用 **GDB** 进行调试 **- 5.1.3 Setting Catchpoints**

20.3.6. 使用 GDB 通过应用程序中断信号

GDB 启用在执行程序的过程中停止各种情况的执行。要在程序收到操作系统信号时停止执行，请使用 **GDB** 捕获点。

先决条件

- 了解 **GDB** 断点

- *GDB 已附加到程序*

使用 GDB 停止接收信号的程序执行

1. *设置捕获点：*

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

命令 catch signal 会设置一个特殊类型的断点，从而在收到程序收到信号时停止执行。signal-type 选项指定信号的类型。使用特殊值 "all" 来捕获所有信号。

2. *如果程序还没有开始执行，请启动它：*

```
(gdb) r
```

如果程序执行只停止，请恢复它：

```
(gdb) c
```

3. *GDB 会在程序收到任何指定的信号后停止执行。*

其它资源

- *第 20.2.4 节 “使用 GDB 显示程序内部值”*
- *第 20.2.3 节 “使用 GDB 逐步处理程序代码”*
- *使用 GDB 调试 - 5.3.1 Setting Catchpoints*

第 21 章 调试崩溃应用程序

有时，无法直接调试应用。在这些情况下，您可以在应用程序终止时收集有关应用程序的信息，然后对其进行分析。

21.1. 内核转储

这部分论述了 内核转储 是什么以及如何使用它。

先决条件

- 了解调试信息

描述

在应用程序停止工作时，核心转储是部分应用程序内存的副本，以 **ELF** 格式存储。它包含应用的所有内部变量和堆栈，可以检查应用的最终状态。使用相应的可执行文件和调试信息进行增强时，可以使用调试器以类似于分析正在运行的程序的方式分析核心转储文件。

如果启用了此功能，**Linux** 操作系统内核会自动记录内核转储。或者，您可以向任何正在运行的应用发送信号，以生成核心转储，而不考虑其实际状态。

警告

某些限制可能会影响生成内核转储的功能。

21.2. 使用内核转储记录应用程序崩溃

要记录应用程序崩溃，请设置核心转储保存并添加系统信息。

步骤

1. 启用内核转储。编辑文件 **/etc/systemd/system.conf**，并将包含 **DefaultLimitCORE** 的行改为以下内容：

```
DefaultLimitCORE=infinity
```

2. 重启系统：

```
# shutdown -r now
```

3. 删除内核转储大小的限制：

```
# ulimit -c unlimited
```

要反转这个更改，请运行值为 **0** 的命令，而不是 无限。

4. 当应用程序崩溃时，会生成一个内核转储。内核转储的默认位置是应用在崩溃时的工作目录。

5. 创建 **SOS** 报告以提供有关系统的更多信息：

```
# sosreport
```

这将创建一个 **tar** 存档，其中包含系统的相关信息，如配置文件副本。

6. 将核心转储和 **SOS** 报告传送到要进行调试的计算机。如果已知，也传输可执行文件。

重要

如果可执行文件未知，则后续的对核心文件的分析将识别该文件。

7. 可选：在传输内核转储和 **SOS** 报告来释放磁盘空间后，请删除它们。

其它资源

- 知识库文章 - 如何在应用程序崩溃或分段错误时启用核心文件转储
-

知识库文章 - 什么是 **sosreport** 以及如何在 **Red Hat Enterprise Linux 4.6** 及之后的版本中创建？

21.3. 使用内核转储检查应用程序 RASH 状态

先决条件

- 您有一个内核转储文件和 **SOS** 报告
- 在系统中安装了 **GDB** 和 **elfutils**

步骤

1. 要识别发生崩溃的可执行文件，请使用核心转储文件运行 **eu-unstrip** 命令：

```
$ 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
```

输出包含一个行中每个模块的详情，用空格分开。此信息按以下顺序列出：

1. 映射模块的内存地址
2. 模块的 **build-id** 及其在内存中的位置
3. 模块的可执行文件名称，如果未从文件加载为 **-** 时显示为 **- when unknown** 或 等 。
4. 调试信息源（如果可用时显示为文件名），如 包含在 可执行文件本身中，或 （ 如果根本不存在）

5. 主模块的共享库名称**(soname)**或 **[exe]**

在本例中，重要的详细信息是文件名 **/usr/bin/sleep** 和 **build-id 2818b2009547f780a5639c904cded443e564973e on the text [exe]**。使用此信息，您可以识别分析核心转储所需的可执行文件。

2. 获取崩溃的可执行文件。

- 如果可能，请从发生崩溃的系统中复制它。使用从 **core** 文件提取的文件名。
- 或者，在您的系统上使用相同的可执行文件。在 **Red Hat Enterprise Linux** 上构建的每个可执行文件都包含带有唯一 **build-id** 值的备注。确定相关的本地可用可执行文件的 **build-id**：

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

使用此信息将远程系统上的可执行文件与您的本地副本匹配。内核转储中列出的本地文件和 **build-id** 的 **build-id** 必须匹配。

- 最后，如果应用程序是从 **RPM** 软件包安装的，您可以从 软件包中获取可执行文件。使用 **sosreport** 输出来查找所需软件包的确切版本。

3. 获取由可执行文件使用的共享库。将与 相同的步骤用于 可执行文件。

4. 如果应用程序以软件包形式分发，请在 **GDB** 中加载可执行文件，以显示缺少 **debuginfo** 软件包的提示。如需了解更多详细信息，请参阅 第 **20.1.4** 节 **"使用 GDB 获取应用程序或库的调试信息软件包"**。

5. 要详细检查核心文件，使用 **GDB** 加载可执行文件和核心转储文件：

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

有关缺少文件和调试信息的其他消息可帮助您识别调试会话缺少的内容。如果需要，返回到上一步骤。

如果调试信息以文件而不是软件包形式提供，请使用 **symbol-file** 命令在 **GDB** 中载入此文件：

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

使用实际文件名替换 **program.debug**。

注意

可能不需要为核心转储中包含的所有可执行文件安装调试信息。这些可执行文件中大部分是由应用代码使用的库。这些库可能不会直接导致您要分析的问题，您不需要为它们包含调试信息。

6.

在应用崩溃时，使用 **GDB** 命令检查应用的状态。请参阅 **第 20.2 节 “使用 GDB 检查应用程序的内部状态”**。

注意

分析核心文件时，**GDB** 未附加到正在运行的进程。控制执行的命令无效。

其它资源

- 使用 **GDB** 进行调试 **- 2.1.1** 选择文件
- 使用 **GDB** 进行调试 **— 18.1 Commands to Specify Files**
- 使用 **GDB** 进行调试 **— 18.3 Debugging Information in Separate Files**

21.4. 使用 GCORE转储进程内存

内核转储调试的工作流允许分析程序的离线状态。在某些情况下，将这个工作流与仍在运行的程序配合使用时很有用，例如难以使用该进程访问环境时。您可以使用 **gcore** 命令转储仍在运行的任何进程的内存。

先决条件

- 了解内核转储
- GDB 已安装在系统中

步骤

使用 **gcore** 转储进程内存：

1. 查找进程 **id(pid)**。使用 **ps**、**pgrep** 和 **top** 等工具 ：

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

2. 转储这个过程的内存：

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

这会创建 文件名，并转储其中的进程内存。在内存被转储时，进程的执行将被停止。

3. 内核转储完成后，进程会恢复正常执行。

4. 创建 **SOS** 报告以提供有关系统的更多信息：

```
# sosreport
```

这将创建一个 **tar** 存档，其中包含系统的相关信息，如配置文件副本。

5. 将程序的可执行文件、核心转储和 **SOS** 报告传送到要进行调试的计算机。

6. 可选：在传输内核转储和 **SOS** 报告以回收磁盘空间后，删除它们。

其他资源

- 知识库文章 - 如何在不重启应用程序的情况下获取核心文件？

21.5. 使用 GDB 转储保护进程内存

您可以将进程内存标记为不会转储。这可节省资源并确保进程内存包含敏感数据，以确保其他安全性。内核内核转储(kdump)和手动内核转储(gcore，GDB)都不会转储以这种方式标记的内存。

在某些情况下，无论这些保护是什么，都需要转储进程内存的所有内容。此流程演示了如何使用 GDB 调试器执行此操作。

先决条件

- 了解内核转储
- GDB 已安装在系统中
- GDB 附加到带有保护内存的进程

步骤

1. 将 GDB 设置为忽略 /proc/PID/coredump_filter 文件中的设置：

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

2. 将 GDB 设置为忽略内存页面标记 VM_DONTDUMP ：

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

3. 转储内存：

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

使用您要转储内存的文件名替换 core-file。

其它资源

- *使用 **GDB** 进行调试 - **10.19 How to Produce a Core file from your program***

部分 VI. 监控性能

开发人员概况程序侧重于对性能有最大影响的计划区域。收集的数据类型包括程序使用哪个部分处理器时间以及分配内存的位置。分析从实际程序执行收集数据。因此，由程序执行的实际任务影响到所收集的数据质量。分析期间执行的任务应该是实际用途的代表；这样可确保在开发过程中解决实际使用程序时出现问题。

Red Hat Enterprise Linux 包含多个不同的工具（**Valgrind**、**O Profile**、**perf** 和 **SystemTap**）来收集性能分析数据。每个工具都适合执行特定类型的配置集运行，如以下部分所述。

第 22 章 VALGRIND

Valgrind 是一种用于构建动态分析工具的工具框架，可用于详细分析应用程序。默认安装已提供五种标准工具：**Valgrind** 工具通常用于调查内存管理和线程问题。**Valgrind** 提供了用户空间二进制文件支持，以检查是否有错误，比如使用未初始化的内存、内存不足/空内存的分配/空内存参数以及系统调用不正确的参数。其性能分析工具可用于大多数二进制文件；但是，与其他 **profilers** 相比，**Valgrind** 配置集运行会非常慢。要分析二进制文件，**Valgrind** 在特殊虚拟机内运行它，允许 **Valgrind** 截获所有二进制指令。**Valgrind**的工具最常用于查找用户空间程序中的与内存相关问题；它不适用于调试特定于时间的问题或内核空间检测和调试。

当为正在调查的程序或库安装 **debuginfo** 软件包时，**Valgrind** 报告非常有用和准确。请参阅 第 **20.1** 节 **"使用调试信息启用调试"**。

22.1. VALGRIND 工具

Valgrind 套件由以下工具组成：

Memcheck

此工具检测程序中的内存管理问题：

- 通过检查内存的所有读取和写入
- 通过截获内存操作，如对 **malloc** 的调用、可用、新 或删除

Memcheck 或许是最常用的 **Valgrind** 工具，因为内存管理问题难以使用其他手段进行检测。在较长的时间段内，这些问题往往不会检测到，从而导致出现难以诊断的崩溃。

在未选择特定工具时，**Mer Memcheck** 函数作为默认工具。

cachegrind

cachegrind 是一个缓存配置集，通过对 **CPU** 中 **I1**、**D1** 和 **L2** 缓存的详细模拟来准确识别代码中缓存丢失的缓存源。它显示缓存未命中、内存引用和指令的数量到源代码的各行；**cachegrind** 还提供每个功能、按模块和整个程序摘要，甚至可以为每个单独机器指令显示计数。

Callgrind

与 **cachegrind** 类似，**callgrind** 可以模型缓存行为。但是，**callgrind** 的主要用途是记录所执行代码的调用数据。

衍生（如果是）

衍生版本 是堆配置集；它测量程序使用的堆内存量，提供有关堆块、堆管理开销和堆栈大小的信息。堆配置文件器在找到降低堆内存使用情况的方式非常有用。在使用虚拟内存的系统上，优化堆内存使用的程序不太可能耗尽内存，并且因为需要较少的分页速度可能更快。

Helgrind

在使用 **POSIX pthreads** 线程原始的程序中，**helgrind** 会检测到同步错误。这些错误包括：

- **POSIX pthreads API** 的 **Misuses**
- 从锁定排序问题导致潜在的死锁
- 数据争用（即访问内存而无需充分锁定）

22.2. 使用 VALGRIND

valgrind 软件包及其依赖项安装执行 **Valgrind** 配置集运行的所有必要工具。要对带有 **Valgrind** 的程序进行性能分析，请使用：

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

有关 工具 名称 的参数列表，请参阅 第 22.1 节 “Valgrind 工具”。除了 **Valgrind** 工具套件外，无 也是工具名称的有效参数；此参数允许您在 **Valgrind** 下运行程序，而无需执行任何性能分析。 这可用于调试或基准测试 **Valgrind** 本身。

您还可以指示 **Valgrind** 将所有信息发送到特定文件。为此，请使用选项 **--log-file=filename**。例如，要检查可执行文件 **hello** 的内存用量并将配置集信息发送到 输出，请使用：

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

有关 **Valgrind** 的更多信息，以及 **Valgrind** 工具套件的其它可用文档，请参阅 第 22.3 节 “其他信息”。

22.3. 其他信息

有关 **Valgrind** 的更多信息，请参阅 **man valgrind**。**Red Hat Enterprise Linux** 还提供 **PDF** 和 **HTML** 中的综合 **Valgrind** 文档 一书：

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

第 23 章 OPROFILE

OProfile 是一个由 **oprofile** 软件包提供的、全系统范围内的性能监控工具的较低的开销。它使用系统处理器中的性能监控硬件来检索系统中内核和可执行文件的信息，如引用内存时、第二级缓存请求数量以及收到的硬件中断数量。**OProfile** 还能够对在 **Java** 虚拟机**(JVM)**中运行的应用程序进行性能分析。

以下是 **OProfile** 提供的工具选择：

ophelp

显示系统处理器的可用事件，以及每个事件的简短描述。

operf

主要分析工具。**operf** 工具使用 **Linux** 性能事件子系统，它允许 **OProfile** 使用系统的性能监控硬件与其他工具进行操作。

与之前使用的 **opcontrol** 工具不同，不需要初始设置，除非使用了 **--system-wide** 选项，否则在没有 **root** 权限的情况下可以使用它。

ocount

用于计算出现绝对事件数的工具。它可以计算整个系统、每个进程、每个 **CPU** 或每个线程的事件。

opimport

将外部二进制格式中的示例数据库文件转换为系统的原生格式。仅在分析不同架构的示例数据库时使用这个选项。

opannotate

如果应用程序使用调试符号编译，则为可执行文件创建一个注解的源。

opreport

读取记录的性能数据，并生成根据配置集规格指定的摘要。可以使用不同的配置集规格从同一配置集数据生成不同的报告。

23.1. 使用 OPROFILE

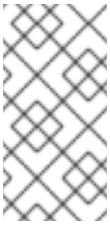
operf 是收集性能分析数据的推荐工具。工具不需要任何初始配置，并且所有选项都在命令行中传递给它。与传统的 **opcontrol** 工具不同，**operf** 可以运行没有 **root** 特权。有关如何使用 **operf** 工具的详细信息，请参阅 《系统管理员指南》 中的使用 **operf** 章节。

例 **23.1.** 使用 **ocount**

以下示例显示了在执行 **sleep** 实用程序期间使用 **ocount** 的事件量：

```
$ 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
```



注意

事件是特定于处理器的处理器实施。可能需要设置选项 **perf_event_paranoid**，或者将计数限制为仅用户空间事件。

例 **23.2.** 基本 **operf** 使用量

在以下示例中，**o perf** 工具用于从 **ls -l ~** 命令收集性能分析数据。

1. 为 **ls** 命令安装调试信息：

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

2. 运行性能分析：

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

3. 分析收集的数据：

```
$ 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.** 使用 **operf** 配置文件 **Java** 计划*

*在以下示例中，**operf** 工具用于从 **Java(JIT)**程序收集性能分析数据，而 **opreport** 工具则用于针对每个**symbol** 数据输出。*

1. *安装本示例中使用的演示 **Java** 程序。它是 **java-1.8.0-openjdk-demo** 软件包的一部分，该软件包包含在 **Optional** 频道中。如需有关如何使用 **Optional** 频道的说明，请参阅添加 **Optional** 和 **Supplementary** 软件仓库。启用 **Optional** 频道时，安装软件包：*

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

2. *为 **OProfile** 安装 **oprofile-jit** 软件包，以便能够从 **Java** 程序收集性能分析数据：*

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

3. *为 **OProfile** 数据创建一个目录：*

```
$ mkdir ~/oprofile_data
```

4. *使用演示程序更改到该目录：*

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

5. *启动配置集：*

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

6. *进入主目录并分析收集的数据：*

```
$ cd
```

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

示例输出可能类似如下：

```
$ 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 文档

有关 **OProfile** 的更多信息，请参阅 **oprofile(1)**手册页。**Red Hat Enterprise Linux** 还为 **file:///usr/share/doc/oprofile-**版本/ 中的 **OProfile** 提供了两个综合指南：

oprofile Manual

有关设置和使用 **OProfile** 的详细手册，请访问 **file:///usr/share/doc/oprofile-version/oprofile.html**

Oprofile 内部

有关 **OProfile** 的内部工作的文档，对于有兴趣贡献 **OProfile** 上游的编程人员很有用，请参阅 **file:///usr/share/doc/oprofile-**版本/**internals.html**

第 24 章 SYSTEMTAP

SystemTap 是一个实用的工具平台，可用于探测在 Linux 系统上运行的进程和内核活动。执行探测：

1. 编写指定系统事件（例如，虚拟文件系统读取、数据包传输）的 SystemTap 脚本 应触发指定的操作（例如，打印、解析或以其他操作数据）。

2. SystemTap 将脚本转换为 C 程序，它将编译到内核模块中。

3. SystemTap 加载内核模块来执行实际探测。

SystemTap 脚本对于监控系统操作以及诊断系统问题时非常有用，通过最小的入侵进入系统正常的操作。您可以快速检测正在运行的系统测试假设，而无需重新编译和重新安装代码。要编译可探测 内核空间的 SystemTap 脚本，SystemTap 使用三个不同的 内核信息软件包 ：

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

这些内核信息软件包必须与内核匹配才能探测。另外，为了为多个内核编译 SystemTap 脚本，还必须安装每个内核的内核信息软件包。

24.1. 其它信息

有关 SystemTap 的详情，请参阅以下红帽文档：

- SystemTap 启动者指南
- SystemTap Tapset 参考

第 25 章 LINUX(PCL)工具和 PERF 的性能计数器

Linux 性能计数器 (PCL)是一种基于内核的子系统，提供用于收集和分析性能数据的框架。Red Hat Enterprise Linux 7 包含这个内核子系统来收集数据和用户空间工具 perf，以分析所收集的性能数据。PCL 子系统可用于测量硬件事件，包括已停用的指令和处理器时钟周期。它还可以测量软件事件，包括主要的页面错误和上下文切换。例如，PCL 计数器可以计算进程已停用和处理器时钟周期中的 Per Clock (IPC)。低 IPC 比率表示代码使用不佳的 CPU。其他硬件事件也可用于诊断 CPU 性能不佳。

性能计数器也可以配置为记录示例。可以使用相对的示例数量来识别哪些代码区域对性能有最大影响。

25.1. PERF 工具命令

有用的 perf 命令包括以下内容：

perf stat

此 perf 命令为常见性能事件提供总体统计信息，包括执行的指令和消耗的时钟周期。选项允许选择默认测量事件以外的事件。

perf 记录

此 perf 命令将性能数据记录到文件中，稍后可以使用 perf 报告 进行分析。

perf 报告

此 perf 命令从文件中读取性能数据，并分析记录的数据。

perf list

此 perf 命令列出特定机器上可用的事件。这些事件将根据系统的性能监控硬件和软件配置而有所不同。

使用 perf help 获取 perf 命令的完整列表。若要检索每个 perf 命令的 man page 信息，请使用 perf help 命令。

25.2. 使用 PERF

使用基本的 PCL 基础架构收集统计数据或程序执行示例相对简单。本节提供了完整的统计数据和抽样示例。

要收集 **make** 及其子项的统计信息，请使用以下命令：

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

perf 命令收集了大量不同的硬件和软件计数器。然后输出以下信息：

```
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 工具也可以记录示例。例如，要记录 **make** 命令及其子项上的数据，请使用：

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

这将打印保存样本的文件，以及所收集的样本数量：

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

Linux(PCL)工具的性能计数器与 OProfile 冲突

Linux(PCL)的 OProfile 和性能计数器均使用相同的硬件性能监控单元**(PMU)**。如果在尝试使用 **PCL perf** 命令时运行 **OProfile**，则启动 **OProfile** 时会出现类似以下内容的错误消息：

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

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

要使用 **perf** 命令，首先关闭 **OProfile**：

```
# opcontrol --deinit
```

然后，您可以分析 **perf.data**，以确定示例相对频率。报告输出包括示例的命令、对象和功能。使用 **perf** 报告 输出 **perf.data** 分析。例如，以下命令可生成消耗最多时间的可执行文件报告：

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

生成的输出：

```
# 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
```

左侧的列显示了相对示例数量。此输出显示，在 **xsltproc** 和 **pdfxmltex** 中花费了大部分时间。为缩短完成的时间，请关注 **xsltproc** 和 **pdfxmltex**。要列出 **xsltproc** 执行的功能，请运行：

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

这会生成：

```
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
```

*附录 **A.** 修订历史记录*

*修订 **7-6.1，2018 年 10 月 30 日，Vladimír Slávik***

*为 **7.6 GA** 版本构建。*

*修订 **7-6，2018 年 8 月 21 日，Vladimír Slávik***

*为 **7.6 Beta** 版本构建。*

*修订 **7-5.1, Tue Apr 10 2018, Vladimír Slávik***

*为 **7.5 GA** 版本构建。*

*修订 **7-5，2018 年 1 月 9 日，Vladimír Slávik***

*预览用于 **7.5 Beta** 的新图书版本*

*修订 **7-4.1，2017 年 8 月 22 日，Vladimír Slávik***

更新链接产品的新版本。

*修订 **7-4、Wed Jul 26 2017、Vladimír Slávik***

*为 **7.4 GA** 版本构建。关于设置用于开发的工作站的新章节。*

*修订 **1-12, 2012 年 5 月 26 日，Vladimír Slávik***

更新以移除过期的信息。

*版本 **7-3.9，Mon May15 2017 年 5 月 15 日 Robert Krátk":{***

*为 **7.4 Beta** 版本构建。*