# Red Hat Enterprise Linux 8

## 在 RHEL 8 中开发 C 和 C++ 应用程序

设置开发人员工作站,并在 Red Hat Enterprise Linux 8 中开发和调试 C 和 C++ 应用程序

Last Updated: 2021-06-07

# Red Hat Enterprise Linux 8 在 RHEL 8 中开发 C 和 C++ 应用程序

设置开发人员工作站,并在 Red Hat Enterprise Linux 8 中开发和调试 C 和 C++ 应用程序

Olga Tikhomirova
otikhomi@redhat.com

Levi Valeeva
lvaleeva@redhat.com

Vladimír Slávik
Red Hat Customer Content Services

## 法律通告

Copyright © 2021 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.

## 摘要

本文档描述了让 Red Hat Enterprise Linux 8 成为应用程序开发的理想企业级平台的不同功能和工具。

# 目录

**使开源包含更多** .... 5

**对红帽文档提供反馈** .... 6

**第 1 章 设置开发工作站** .... 7
- 1.1. 先决条件 7
- 1.2. 启用调试和源存储库 7
- 1.3. 设置以管理应用程序版本 7
- 1.4. 设置以使用 C 和 C++ 开发应用程序 8
- 1.5. 设置以调试应用程序 8
- 1.6. 设置以测量应用程序的性能 9

**部分 I. 创建 C 或 C++ 应用程序** .... 10

**第 2 章 使用 GCC 构建代码** .... 11
- 2.1. 代码表单之间的关系 11
- 2.2. 将源文件编译到对象代码 12
- 2.3. 使用 GCC 启用 C 和 C++ 应用程序调试 12
- 2.4. 使用 GCC 的代码优化 13
- 2.5. 使用 GCC 强化代码的选项 13
- 2.6. 链接代码以创建可执行文件 14
- 2.7. 示例：使用 GCC 构建 C 程序 15
- 2.8. 示例：使用 GCC 构建 C++ 程序 15

**第 3 章 在 GCC 中使用代理** .... 17
- 3.1. 库命名惯例 17
- 3.2. 静态和动态链接 17
- 3.3. 使用 GCC 库 18
- 3.4. 在 GCC 中使用静态库 19
- 3.5. 在 GCC 中使用动态库 20
- 3.6. 通过 GCC 使用静态和动态库 21

**第 4 章 使用 GCC 创建库** .... 23
- 4.1. 库命名惯例 23
- 4.2. SONAME 机制 23
- 4.3. 使用 GCC 创建动态库 24
- 4.4. 使用 GCC 和 AR 创建静态库 25

**第 5 章 使用 MAKE 管理更多代码** .... 27
- 5.1. GNU MAKE 和 MAKEFILE 概述 27
- 5.2. 示例：使用 MAKEFILE 构建 C 程序 28
- 5.3. 文档资源 MAKE 29

**第 6 章 RHEL 7 后对 TOOLCHAIN 的更改** .... 31
- 6.1. RHEL 8 中的 GCC 的更改 31
- 6.2. RHEL 8 中 GCC 的安全性增强 32
- 6.3. RHEL 8 中 GCC 中破坏兼容性的更改 35
  - C++ ABI 的更改 std::string 和 std::list 35
  - GCC 不再构建 Ada、Go 和 Objective C/C++ 代码 35

**部分 II. 调试应用程序** .... 36

**第 7 章 使用调试信息启用调试** .... 37
- 7.1. 调试信息 37

7.2. 使用 GCC 启用 C 和 C++ 应用程序调试 37
7.3. DEBUGINFO 和 DEBUGSOURCE 软件包 38
7.4. 使用 GDB 为应用程序或库获取 DEBUGINFO 软件包 38
7.5. 手动为应用程序或库获取 DEBUGINFO 软件包 39

**第 8 章 使用 GDB 检查应用程序 INTERNAL STATE 42**
8.1. GNU DEBUGGER(GDB) 42
8.2. 将 GDB 附加到进程 42
8.3. 使用 GDB 通过程序代码步骤 43
8.4. 使用 GDB 显示程序内部值 45
8.5. 使用 GDB 中断点停止在定义的代码位置执行 45
8.6. 使用 GDB 监视点在数据访问和更改时停止执行 46
8.7. 使用 GDB 调试或线程程序 47

**第 9 章 记录应用程序互动 49**
9.1. 用于记录应用程序交互的工具 49
9.2. 使用 STRACE 监控应用程序的系统调用 50
9.3. 使用 LTRACE 监控应用程序的库功能 51
9.4. 使用 SYSTEMTAP 监控应用程序的系统调用 52
9.5. 使用 GDB 拦截应用程序系统调用 53
9.6. 使用 GDB 截取应用程序对信号的处理 54

**第 10 章 调试 CRASHED APPLICATION 55**
10.1. CORE DUMPS:它们的状态以及如何使用它们 55
10.2. 使用内核转储记录应用程序崩溃 55
10.3. 使用内核转储检查应用程序崩溃状态 56
10.4. 使用 COREDUMPCTL 创建并访问内核转储 58
10.5. 使用它转储进程内存 GCORE 59
10.6. 使用 GDB 转储受保护进程内存 60

**第 11 章 GDB 中破坏兼容性的更改 62**
gdbserver 现在使用 shell 启动 feriors 62
gcj 支持已删除 62
符号转储维护命令的新语法 62
线程数不再是全局的 63
值内容的内存可能会受限制 63
不再支持 Sun 版本的 stabs 格式 63
sysroot 处理更改 64
HISTSIZE 不再控制 GDB 命令历史大小 64
添加了完成限制 64
删除了 HP-UX XDB 兼容性模式 64
为线程处理信号 64
breakpoint 模式总是关闭并自动合并 64
不再支持 remotebaud 命令 64

**部分 III. 用于开发的额外工具集 65**

**第 12 章 使用 GCC TOOLSET 66**
12.1. 什么是 GCC TOOLSET 66
12.2. 安装 GCC TOOLSET 66
12.3. 从 GCC TOOLSET 安装单独的软件包 66
12.4. 卸载 GCC TOOLSET 66
12.5. 从 GCC TOOLSET 运行工具 67
12.6. 使用 GCC TOOLSET 运行 SHELL 会话 67
12.7. 相关信息 67

**第 13 章 GCC TOOLSET 9 .......... 68**
13.1. GCC TOOLSET 9 提供的工具和版本 68
13.2. GCC TOOLSET 9 中的 C++ 兼容性 68
13.3. GCC TOOLSET 9 中的 GCC 的具体设置 69
13.4. GCC TOOLSET 9 中的 BINUTILS 的具体设置 69

**第 14 章 GCC TOOLSET 10 .......... 71**
14.1. GCC TOOLSET 10 提供的工具和版本 71
14.2. GCC TOOLSET 10 中的 C++ 兼容性 71
14.3. GCC TOOLSET 10 中的 GCC 的具体设置 72
14.4. GCC TOOLSET 10 中的 BINUTILS 的具体设置 72

**第 15 章 使用 GCC TOOLSET 容器镜像 .......... 74**
15.1. GCC TOOLSET 容器镜像内容 74
15.2. 访问并运行 GCC TOOLSET 容器镜像 75
15.3. 示例：使用 GCC TOOLSET 10 TOOLCHAIN 容器镜像 76
15.4. 使用 GCC TOOLSET 10 PERFTOOLS 容器镜像中的 SYSTEMTAP 76

**第 16 章 编译器工具集 .......... 78**

**第 17 章 ANNOBIN 项目 .......... 79**
17.1. 使用 ANNOBIN 插件 79
17.1.1. 启用 annobin 插件 79
17.1.2. 将选项传递给 annobin 插件 80
17.2. 使用 ANNOCHECK 程序 80
17.2.1. 使用 annocheck 检查文件 81
17.2.2. 使用 annocheck 检查目录 81
17.2.3. 使用 annocheck 检查 RPM 软件包 81
17.2.4. 使用 annocheck 额外工具 82
17.2.4.1. 启用 built-by 工具 82
17.2.4.2. 启用 notes 工具 83
17.2.4.3. 启用 section-size 工具 83
17.2.4.4. 强化检查程序基础 83
17.2.4.4.1. 强化检查器选项 83
17.2.4.4.2. 禁用强化检查器 84
17.3. 删除冗余 ANNOBIN 备注 84

**部分 IV. 附加主题 .......... 85**

**第 18 章 编译器和开发工具中破坏兼容性的更改 .......... 86**
librtkaio 删除 86
从中删除的 Sun RPC 和 NIS 接口 glibc 86
32 位 Xen 的 nosegneg 库已被删除 86
make 新的 operator != 会对某些现有的 makefile 语法有不同的解释 86
用于 MPI 调试支持的 Valgrind 库已删除 86
从中移除开发标头和静态库 valgrind-devel 86

**第 19 章 在 RHEL 8 上运行 RHEL 6 或 7 应用程序的选项 .......... 88**

# 使开源包含更多

红帽承诺替换我们的代码、文档和网页属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。如需了解更多详细信息，请参阅 CTO Chris Wright 信息 。

# 对红帽文档提供反馈

我们感谢您对文档提供反馈信息。请让我们了解如何改进文档。要做到这一点：

- 关于特定内容的简单评论：
  1. 请确定您使用 *Multi-page HTML* 格式查看文档。另外，确定 **Feedback** 按钮出现在文档页的右上方。
  2. 用鼠标指针高亮显示您想评论的文本部分。
  3. 点在高亮文本上弹出的 **Add Feedback**。
  4. 按照显示的步骤操作。
- 要提交更复杂的反馈，请创建一个 Bugzilla ticket：
  1. 进入 Bugzilla 网站。
  2. 在 Component 中选择 **Documentation**。
  3. 在 **Description** 中输入您要提供的信息。包括文档相关部分的链接。
  4. 点 **Submit Bug**。

# 第 1 章 设置开发工作站

Red Hat Enterprise Linux 8 支持开发自定义应用程序。要允许开发人员完成此操作,必须使用所需工具和工具设置该系统。本章列出了开发以及要安装项目的最常用用例。

## 1.1. 先决条件

- 必须安装该系统,包括图形环境,并订阅。

## 1.2. 启用调试和源存储库

标准安装 Red Hat Enterprise Linux 无法启用 debug 和 source 软件仓库。这些软件仓库包含调试系统组件并测量其性能所需的信息。

**流程**

- 启用源和调试信息软件包频道：

```
# subscription-manager repos --enable rhel-8-for-$(uname -i)-baseos-debug-rpms
# subscription-manager repos --enable rhel-8-for-$(uname -i)-baseos-source-rpms
# subscription-manager repos --enable rhel-8-for-$(uname -i)-appstream-debug-rpms
# subscription-manager repos --enable rhel-8-for-$(uname -i)-appstream-source-rpms
```

**$(uname -i)** 部分自动替换为系统架构的匹配值：

| 构架名称 | 值 |
| --- | --- |
| 64-bit Intel 和 AMD | x86_64 |
| 64-bit ARM | aarch64 |
| IBM POWER | ppc64le |
| 64-bit IBM Z | s390x |

## 1.3. 设置以管理应用程序版本

有效的版本控制对所有多开发者项目都至关重要。Red Hat Enterprise Linux 由 Git 提供,它是一个分布式版本控制系统。

**流程**

1. 安装 **git** 软件包：

```
# yum install git
```

2. 可选：设置与您的 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@example.com"
```

将 *Full Name* 和 email@example.com *替换* 为您的实际名称和电子邮件地址。

3. 可选： 要更改由 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
```

使用用于启动所选文本编辑器的命令替换 *command*。

**其它资源**

- Linux 手册页,Git 和教程：

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

请注意,很多 Git 命令都有自己的手册页。例如, *git-commit(1)*。

- *Git 用户的手册* - Git 的 HTML 文档位于 **/usr/share/doc/git/user-manual.html**。
- pro Git - *Pro Git* 书的在线版本提供了 Git、概念及其用法的详细描述。
- 参考 - Git 的 Linux 手册页的在线版本

## 1.4. 设置以使用 C 和 C++ 开发应用程序

Red Hat Enterprise Linux 包含创建 C 和 C++ 应用程序的工具。

**先决条件**

- 必须启用 debug 和 source。

**流程**

1. 安装开发工具 **软件包组**,包括 GNU Compiler Collection(GCC)、GNU Debugger(GDB)和其他开发工具：

```
# yum group install "Development Tools"
```

2. 安装基于 LLVM 的工具链,包括 **clang** 编译器和 **lldb** debugger:

```
# yum install llvm-toolset
```

3. 可选： Fortran 依赖关系,安装 GNU Fortran 编译器：

```
# yum install gcc-gfortran
```

## 1.5. 设置以调试应用程序

Red Hat Enterprise Linux 提供多个调试和工具工具来分析和排除内部应用程序行为。

**先决条件**

- 必须启用 debug 和 source。

**流程**

1. 安装可用于调试的工具：

```
# yum install gdb valgrind systemtap ltrace strace
```

2. 安装 **yum-utils** 软件包以便使用 **debuginfo-install** 工具：

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

3. 运行 SystemTap 帮助程序脚本来设置环境。

```
# stap-prep
```

   请注意，**stap-prep** 安装与当前 *运行* 的内核相关的软件包,它们可能与实际安装的内核不相同。要确保 **stap-prep** 安装正确的 **kernel-debuginfo** 和 **kernel-headers** 软件包, 请使用 **uname -r** 命令检查当前的内核版本,并在需要时重启您的系统。

4. 确保 **SELinux** 策略允许相关应用程序不仅可以正常运行,而且在调试情况下运行。如需更多信息,请参阅 使用 SELinux。

**其它资源**

- 第 7 章 *使用调试信息启用调试*

## 1.6. 设置以测量应用程序的性能

Red Hat Enterprise Linux 包括几个可帮助开发人员找出导致应用程序性能丢失的原因的应用程序。

**先决条件**

- 必须启用 debug 和 source。

**流程**

1. 安装用于性能测量的工具：

```
# yum install perf papi pcp-zeroconf valgrind strace sysstat systemtap
```

2. 运行 SystemTap 帮助程序脚本来设置环境。

```
# stap-prep
```

   请注意，**stap-prep** 安装与当前 *运行* 的内核相关的软件包,它们可能与实际安装的内核不相同。要确保 **stap-prep** 安装正确的 **kernel-debuginfo** 和 **kernel-headers** 软件包, 请使用 **uname -r** 命令检查当前的内核版本,并在需要时重启您的系统。

3. 启用并启动 Performance Co-Pilot(PCP)收集器服务：

```
# systemctl enable pmcd && systemctl start pmcd
```

# 部分 I. 创建 C 或 C++ 应用程序

红帽提供了使用 C 和 C++ 语言创建应用程序的多个工具。本书的这部分列出了一些最常见的开发任务。

# 第 2 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。

## 2.1. 代码表单之间的关系

**先决条件**

- 了解编译和连接的概念

**可能的代码表单**

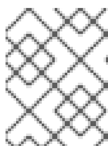
C 和 C++ 语言有三种代码格式：

- **使用 C 或者 C++} 语言编写的源代码**,以纯文本文件的形式出现。
  文件通常使用扩展,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。这是一个中间形式。*
  *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。*



***注意***

*还存在用于静态链接的库归档文件。这是使用 **.a** 文件名称扩展的对象代码变体。不建议使用静态链接。请查看 第 3.2 节 "静态和动态链接"。*

***处理 GCC 中的代码表单***

*从源代码生成可执行代码会分为两个步骤,这需要不同的应用程序或工具。GCC 可以用作编译器和链路器的智能驱动程序。这可让您在任何所需操作中使用单个 **gcc** 命令（编译和连接）。GCC 会自动选择操作及其序列：*

1. *源文件被编译到对象文件中。*
2. *对象文件和库链接（包括之前编译的源）。*

*可以运行 GCC 以便它只执行第 1 步、只执行第 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,它通常为每个操作单独运行 GCC,所以最好始终将编译和连接视为两个不同的操作,即使 GCC 可以同时执行。*

***其它资源***

- *第 2.2 节 "将源文件编译到对象代码"*
- *第 2.6 节 "链接代码以创建可执行文件"*

## 2.2. 将源文件编译到对象代码

要立即从源文件而不是可执行文件创建对象代码文件,必须指示 GCC 只创建对象代码文件作为其输出。此操作是大型项目的构建过程的基本操作。

### 先决条件

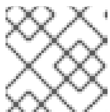
- C 或 C++ 源代码文件
- 在系统中安装 GCC

### 流程

1. 进入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
```

对象文件被创建,其文件名反映了原始源代码文件: **source.c** 结果为 **source.o**。



**注意**

使用 C++ 源代码,将 **gcc** 命令替换为 **g++** 以方便地处理 C++ Standard 库依赖项。

### 其它资源

- 第 2.5 节 "使用 GCC 强化代码的选项"
- 第 2.4 节 "使用 GCC 的代码优化"
- 第 2.7 节 "示例:使用 GCC 构建 C 程序"

## 2.3. 使用 GCC 启用 C 和 C++ 应用程序调试

由于调试信息较大,默认情况下不包含在可执行文件中。要启用 C 和 C++ 应用程序调试,您必须明确指示编译器创建它。

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执行代码,与原始源代码很难执行:变量可能会优化、循环未滚动,操作合并到相邻的源代码等。这会影响调试。要改进的调试体验,请考虑使用 **-Og** 选项设置优化功能。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,包括删除一些程序错误。
- 要在调试信息中包含宏定义,请使用 **-g3** 选项而不是 **-g**。
- **-fcompare-debug** GCC 选项测试由 GCC 编译的带有调试信息且没有调试信息的代码。如果得到的两个二进制文件相同,测试会传递。此测试可确保可执行代码不受任何调试选项的影响,这可以进一步确保 debug 代码中没有隐藏的错误。请注意,使用 **-fcompare-debug** 选项会显著增加编译时间。有关这个选项的详情,请查看 GCC 手册页。

### 其它资源

- 第 7 章 使用调试信息启用调试
- 使用 GNU Compiler Collection(GCC)- 调试您的选项
- 使用 GDB 进行 调试 - 在 Separate 文件中调试信息
- GCC 手册页：

```
$ man gcc
```

## 2.4. 使用 GCC 的代码优化

单个程序可以转换为多个机器指令。如果您在编译过程中为分析代码分配更多资源,则可以实现更最佳的结果。

使用 GCC,您可以使用 **-Olevel** 选项设置优化级别。这个选项接受一组值来代替 级别。

| level | 描述 |
| --- | --- |
| **0** | 优化编译速度 - 不优化代码优化（默认）。 |
| **1**, **2**, **3** | 优化代码执行速度（容量越大,速度越高）。 |
| **s** | 文件大小优化。 |
| **fast** | 与级别 **3** 设置相同,另外 **fast** 会考虑严格的标准合规性,以允许额外的优化。 |
| **g** | 调试体验优化。 |

对于发行版本构建,使用优化选项 **-O2**。

在开发过程中,该 **-Og** 选项在某些情况下可用于调试程序或库。因为一些程序错误清单只带有特定的优化级别,所以使用发行优化级别测试程序或库。

GCC 提供了大量启用单独优化的选项。如需更多信息,请参阅以下附加资源。

### 其它资源

- 使用 GNU Compiler Collection - 3.10 选项该控制选项
- GCC 的 Linux 手册页：

```
$ man gcc
```

## 2.5. 使用 GCC 强化代码的选项

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通常被利用的情况并提高安全性。选择正确的编译器选项可帮助生成更安全的程序及库,而无需更改源代码。

### 发行版本选项

对于针对 Red Hat Enterprise Linux 的开发人员,推荐的选项列表最少：

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

- 对于程序,添加 **-fPIE** 和 **-pie** 独立执行可执行选项。
- 对于动态链接的库,强制的 **-fPIC** （独立代码）选项会间接提高安全性。

**开发选项**

使用以下选项检测开发过程中的安全漏洞。这些选项和发行版本的选项一起使用：

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

**其它资源**

- 清理标准指南
- 使用 GCC 进行内存错误检测 - Red Hat Developers Blog post

## 2.6. 链接代码以创建可执行文件

连接是构建 C 或 C++ 应用程序时的最后步骤。将所有对象文件和库合并到可执行文件中。

**先决条件**

- 一个或多个对象文件
- GCC 必须安装到系统中

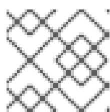
**流程**

1. 进入包含对象代码文件的目录。
2. 运行 **gcc**:

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

   从提供的对象文件和库创建一个名为 **executable-file** 的可执行文件。

   要连接其他库,请在对象文件列表后添加所需的选项。如需更多信息,请参阅 第 3 章 在 GCC 中使用代理。



**注意**

使用 C++ 源代码,将 **gcc** 命令替换为 **g++** 以方便地处理 C++ Standard 库依赖项。

**其它资源**

- 第 2.7 节 "示例：使用 GCC 构建 C 程序"
- 第 3.2 节 "静态和动态链接"

## 2.7. 示例：使用 GCC 构建 C 程序

本例演示了构建简单示例 C 程序的具体步骤。

### 先决条件

- 您必须了解如何使用 GCC。

### 流程

1. 创建目录 **hello-c** 并更改它：

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

2. 创建包含以下内容的 **hello.c** 文件：

```
#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 5.2 节 "示例：使用 Makefile 构建 C 程序"

## 2.8. 示例：使用 GCC 构建 C++ 程序

本例演示了构建最小 C++ 程序示例的步骤。

### 先决条件

- 您必须了解 **gcc** 和 **g++** 之间的区别。

### 流程

1. *创建目录 **hello-cpp** 并更改它：*

```
$ 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 3 章 在 GCC 中使用代理

本章论述了在代码中使用库。

## 3.1. 库命名惯例

库使用一个特殊的文件名惯例： 称为 **foo** 的库应该 作为文件 **libfoo.so** 或 **libfoo.a** 存在。本约定通过链接 GCC 的输入选项来自动理解,而不是由输出选项来理解：

- 当与库连接时,该库只能使用名称 **foo** 指定,其 **-l** 选项为 **-lfoo**:

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

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

**其它资源**

- 第 4.2 节 "soname 机制"

## 3.2. 静态和动态链接

在使用完全编译的语言构建应用程序时,开发人员可以选择使用静态或动态链接。本节列出了不同之处,特别是使用 Red Hat Enterprise Linux 中的 C 和 C++ 语言。要总结,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

静态连接使得生成的可执行文件的库部分。动态链接将这些库作为单独的文件。

动态和静态连接可以通过多种方式进行比较：

**资源使用**

静态链接会产生更大的可执行文件,其中包含更多代码。这个来自库的额外代码无法在系统的多个程序间共享,从而增加文件系统的使用量和内存用量。运行相同静态链接程序的多个进程仍可共享代码。另一方面,静态应用程序需要较少的运行时重新定位,从而减少启动时间,并需要较少的私有设置大小 (RSS)内存。由于位置独立代码(PIC)引入的开销,静态链接生成的代码比动态链接效率更高。

**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。这对于红帽作为 Red Hat Enterprise Linux 的一部分提供的库尤其重要,红帽在其中提供了安全更新。强烈建议您对任何这样的库进行静态连接。

**兼容性**

静态链接可能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。使用静态连接时,这个依赖关系变得不灵活,因此前向兼容性和向后兼容性都会丢失。静态链接保证只能用于构建可执行文件的系统。

**警告**

从 GNU C 库(**glibc**)连接静态库的应用程序仍需要 **glibc** 作为动态库存在于系统中。另外,应用程序运行时可用的 **glibc** 动态库变体在连接应用程序时必须是一个与存在的 glibc 相同的版本。因此,可以保证静态链接只适用于构建可执行文件的系统。

**支持覆盖范围**

红帽提供的大多数静态库都位于 CodeReady Linux Builder 频道中,但红帽不支持。

**功能**

一些库,特别是 GNU C 库(**glibc**)在静态链接时提供较少的功能。
例如,当静态链接时,**glibc** 不支持线程和同一程序中对 **dlopen()** 功能的调用。

由于列出了缺陷,应该避免以所有成本为单位进行静态连接,特别是针对整个应用程序以及 **glibc** 和 **libstdc++** 库。

### 静态链接的情况

在某些情况下,静态连接可能是合理的选择,例如：

- 使用没有启用动态链接的库。
- 在空的 **chroot** 环境或容器中运行代码需要完全静态连接。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 8:应用程序兼容性指南
- Package 清单中 CodeReady Linux Builder 软件仓库 的描述

## 3.3. 使用 GCC 库

库是一组代码,可在您的程序中重复使用。C 或 C++ 库由两个部分组成：

- 库代码
- 标头文件

### 编译使用库的代码

头文件描述了库的接口：库中的功能和变量。编译代码时需要标头文件中的信息。

通常,库的标头文件将放置在应用程序代码不同的目录中。要告诉 GCC 标题文件的位置,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** 选项可以多次使用来添加带有标头文件的多个目录。当查找标头文件时,会按照在 **-I** 选项中出现的顺序搜索这些目录。

### 链接使用库的代码

当连接可执行文件时,应用程序的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有不同的格式：

- 静态库可用作归档文件。它们包含一组对象文件。归档文件有一个文件名扩展 **.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。共享对象具有文件名扩展 **.so**。

要告诉 GCC,库的归档或共享对象文件是 **-L** 选项：

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

使用 库目录的实际路径替换 library_ path。

**-L** 选项可以多次用来添加多个目录。当查找库时,会按照 **-L** 选项的顺序搜索这些目录。

选项的顺序问题： GCC 无法与库 **foo** 链接,除非它知道使用这个库的目录。因此,在使用 **-l** 选项根据库链接前,使用 **-L** 选项指定库目录。

### 编译并链接使用库的一个步骤的代码

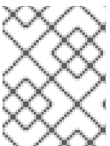
当情况允许在一个 **gcc** 命令中编译并链接代码时,对上述两种情况都使用选项。

### 其它资源

- 使用 GNU Compiler Collection(GCC)- 用于目录搜索的 3.15 选项
- 使用 GNU Compiler Collection (GCC) – 3.14 Options for Linking

## 3.4. 在 GCC 中使用静态库

静态库可作为包含对象文件的归档使用。连接后,它们就成为生成的可执行文件的一部分。

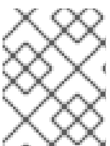


**注意**

出于安全考虑,红帽不建议使用静态链接。请查看 第 3.2 节 "静态和动态链接"。只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 必须安装在您的系统中。
- 您必须了解静态和动态链接。
- 您有一组源代码或对象文件组成一个有效程序,需要一些静态库 **foo** 且没有其它库。
- **foo** 库 作为文件 **libfoo.a** 提供,没有提供 **libfoo.so** 文件进行动态连接。



**注意**

大多数作为 Red Hat Enterprise Linux 一部分的库只支持动态链接。以下步骤只适用于 没有启用动态链接的库。请查看 第 3.2 节 "静态和动态链接"。

### 流程

要从源和对象文件链接程序,请添加静态连接的库 **foo**,它可找到为文件 **libfoo.a**:

1. 进入包含代码的目录。
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** 的目录路径替换 library_ path。

4. 要稍后运行程序,例如：

```
$ ./program
```

### 小心

与静态链接相关的 **-static** GCC 选项禁止所有动态链接。反之,使用 **-Wl,-Bstatic** 和 **-Wl,-Bdynamic** 选项更精确地控制链路器行为。请查看 第 3.6 节 "通过 GCC 使用静态和动态库"。

## 3.5. 在 GCC 中使用动态库

动态库可以作为独立可执行文件使用,需要在连接时间和运行时都需要。它们独立于应用程序的可执行文件。

### 先决条件

- GCC 必须安装到系统中。
- 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** 库必须作为 libfoo .so 文件 提供。

### 针对动态库链接程序

针对动态库 **foo** 链接程序：

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

当某个程序与动态库链接时,生成的程序必须总是在运行时载入库。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

### 在可执行文件中使用 rpath 值

**rpath** 是一个在链接时作为可执行文件的一部分保存的特殊值。之后,当程序从可执行文件加载时,运行时链接器将使用 **rpath** 值来定位库文件。

使用 GCC 连接时,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**

**-Wl,-rpath=** 选项中的逗号没有空格！

在以后运行该程序：

```
$ ./program
```

### 使用 LD_LIBRARY_PATH 环境变量

如果在程序的可执行文件中找不到 **rpath**,运行时链接器将使用 **LD_LIBRARY_PATH** 环境变量。每个程序都必须更改此变量的值。这个值应该代表共享库对象所在的路径。

要在没有设置 **rpath** 的情况下运行程序,请在路径 library_path 中包含库:

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

退出 **rpath** 值可提供灵活性,但需要在每次运行程序时设置 **LD_LIBRARY_PATH** 变量。

### 将库放到默认目录

运行时链接器配置将很多目录指定为动态库文件的默认位置。要使用这个默认行为,请将库复制到适当的目录中。

有关动态链接器行为的完整描述已超出本文档的范围。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
  ```

- 报告动态链接器识别的库,无需额外配置,其中包括目录：

  ```
  $ ldconfig -v
  ```

## 3.6. 通过 GCC 使用静态和动态库

有时需要静态链接一些库,有些库是动态链接。这种情形会带来一些挑战。

**先决条件**

- 了解静态和动态链接

## 简介

GCC 可以识别动态和静态库。当遇到 **-lfoo** 选项时, gcc 首先会尝试定位包含 foo 库动态链接版本的共享对象（**.so** 文件）,然后查找包含库静态版本的归档文件(**.a**)。因此,这个搜索可能会导致以下情况：

- 只找到共享对象,并动态 **使用** gcc 链接。
- 只会找到存档,静态找到针对它的 gcc 链接。
- 找到共享对象和存档,默认情况下, gcc 选择针对共享对象的动态链接。
- 找不到共享对象或存档,链接会失败。

由于这些规则,选择用于链接的静态或动态版本库的最佳方法是只有 gcc **找到的版本**。当指定 **-Lpath** 选项时,这可以通过使用或退出包含库版本的目录来控制。

另外,由于动态链接是默认的,所以必须明确指定连接的唯一情况是,具有两个版本的库应该静态链接。有两个可能的解决方案：

- 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, gcc 被指示使用 **-lfoo** 选项针对 foo 库链接。但是,可以指定到包含库的文件 **libfoo.a** 的完整路径：

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

从文件扩展 **.a** 中, gcc 会了解到这是一个与程序链接的库。但是,指定到库文件的完整路径并不灵活。

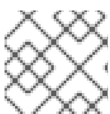
## 使用 -Wl 选项

gcc 选项 **-Wl** 是将选项传递给底层链接器的特殊选项。这个选项的语法与其他 gcc 选项不同。**-Wl** 选项后跟一个用逗号分开的链接选项列表,其他 gcc 选项则需要用空格分开的选项列表。

gcc **使用的**ld **链接程序** 提供选项 **-Bstatic** 和 **-Bdynamic** 来分别指定这个选项后面的库是静态链接还是动态链接。在将 **-Bstatic** 和库传递给链接器后,必须手动恢复默认的动态链接行为,才能让以下库与 **-Bdynamic** 选项动态链接。

要连接程序库, **首先** 静态链接库(**libfirst.a**), **第二个** 动态链接(**libsecond.so**):

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



**注意**

GCC 可以配置为使用默认 **ld** 以外的链接。

## 其它资源

- 使用 GNU Compiler Collection (GCC) — 3.14 Options for Linking
- binutils 2.27 - 2.1 命令线选项的文档

# 第 4 章 使用 GCC 创建库

本章论述了创建库的步骤,并解释了 Linux 操作系统为库使用的必要概念。

## 4.1. 库命名惯例

库使用一个特殊的文件名惯例：称为 foo 的库应该作为文件 **libfoo.so** 或 **libfoo.a** 存在。本约定通过链接 GCC 的输入选项来自动理解,而不是由输出选项来理解：

- 当与库连接时,该库只能使用名称 foo 指定,其 **-l** 选项为 **-lfoo**:

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

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

### 其它资源

- 第 4.2 节 “soname 机制”

## 4.2. SONAME 机制

动态载入的库（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

- 您必须了解动态链接和库。
- 您必须了解 ABI 兼容性的概念。
- 您必须了解库命名约定。
- 您必须了解符号链接。

### 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库的应用程序的情况下更新库。然而,这个概念会产生以下问题：

- 库的实际版本的识别
- 同一库的现有多个版本需要
- 信号每个多个版本的 ABI 兼容性

### soname 机制

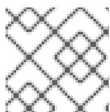
要解决这个问题,Linux 使用名为 soname 的机制。

**foo** 库版本 X.Y 在版本号中与其他值相同的 X 版本兼容 ABI。保持兼容性的小变化会增加 Y。破坏兼容性的主要更改会增加 X 的值。

实际的 **foo** 库版本 X.Y 作为文件 **libfoo.so.x.y**。在库文件中,记录一个带有值为 **libfoo.so.x** 的 soname 来指示兼容性。

当构建应用程序时,链接器通过搜索文件 **libfoo.so** 来查找库。必须存在具有这个名称的符号链接,指向实际库文件。然后连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,链接程序会创建应用程序,它使用 soname,而不是名称或文件名来声明对库的依赖。

当运行时动态链接程序在运行前链接应用程序时,它会从应用程序的可执行文件读取 soname。这个 soname 是 **libfoo.so.x**。必须存在具有这个名称的符号链接,指向实际库文件。无论版本的 Y 组件是什么,都允许加载库,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,有些库用它们的名称对版本进行编码。

### 从文件中读取 soname

显示库文件 **somelibrary** 的 soname

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

将 somelibrary 替换为您要检查的库的实际文件名。

## 4.3. 使用 GCC 创建动态库

允许动态链接的库(共享对象):

- 通过代码重复使用来管理资源
- 通过更轻松地更新库代码来提高安全性

按照以下步骤从源构建和安装动态库。

### 先决条件

- 您必须了解 soname 机制。
- GCC 必须安装到系统中。
- 您必须有库的源代码。

### 流程

1. 使用库源进入目录。
2. 使用所需独立代码选项 **-fPIC** 将每个源文件编译到对象文件中:

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

   对象文件的文件名与原始源代码文件相同,但它的扩展是 **.o**。
3. 从对象文件中链接共享库:

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

   使用的主要版本号是 X,次版本号为 Y。

4. 将 **libfoo.so.x.y** 文件复制到适当位置,系统动态链路器可以在其中找到该文件。在 Red Hat Enterprise Linux 中,库的目录为 **/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：

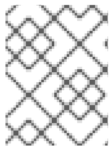
```
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

**其它资源**

- Linux 文档项目 - Program library HOjan - 3。共享代理

## 4.4. 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归档文件,可以创建用于静态链接的库。



**注意**

出于安全考虑,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。详情请查看 第 3.2 节 "静态和动态链接"。

**先决条件**

- GCC 和 binutils 必须安装在系统中。
- 您必须了解静态和动态链接。
- 具有作为库共享的功能的源文件可用。

**流程**

1. 使用 GCC 创建中间对象文件。

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

如果需要,请附加更多源文件。生成的对象文件共享文件名,但使用 **.o** 文件名扩展。

2. 使用 **binutils** 软件包中的 **ar** 工具将对象文件切换到静态库(Archive)。

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

文件 **libfoo.a** 已创建。

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中。

5. *当与库连接时, GCC 会自动从* ***.a*** *文件名扩展识别该程序库是静态链接的归档。*

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

***其它资源***

- *Linux 手册页, ar(1):*

```
$ man ar
```

# 第 5 章 使用 MAKE 管理更多代码

GNU make 工具程序（通常缩写）是 控制源文件生成的可执行文件的工具。自动 决定复杂程序的哪些部分已更改,需要重新编译。使用 名为 Makefiles 的配置文件来控制程序构建的方式。

## 5.1. GNU MAKE 和 MAKEFILE 概述

要从特定项目的源文件中创建可用的表单（通常可执行文件）,请执行几个必要的步骤。记录操作及其序列以便稍后重复它们。

Red Hat Enterprise Linux 包含 GNU **make**,这是为此设计的一个构建系统。

### 先决条件

- 了解编译和连接的概念

### GNU make

GNU **make** 读取包含构建过程说明的 Makefile。Makefile 包含多个 规则,它们描述了通过特定操作(recipe) 满足特定条件(target)的方法。规则可分层依赖于另一个规则。

在没有选项的情况下运行 **make** 会导致它在当前目录中查找 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 details

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。如果某个规则被执行,目标指定了输出。使用 Reipes 的行必须以 TAB 字符开头。

通常, Makefile 包含编译源文件的规则、用于链接结果对象文件的规则以及作为层次结构顶部入口点的目标。

考虑以下 **Makefile** 以构建由单个文件 **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
```

本例显示要达到目标 **all**,需要文件 **hello**。要获得 **hello**,一个需要 **hello.o** （由 **gcc**链接）,它从 **hello.c** 创建（由 **gcc**编译）。

目标 **all** 是默认目标,因为它是第一个以一段时间(.)开头的目标。当当前目录包含这个 **Makefile** 时,在没有参数的情况下运行 **make** 与运行 **make all** 相同。

### 典型的 makefile

一个更典型的 Makefile 使用变量来常规化步骤,并添加一个目标"干净" - 除源文件外,删除所有内容。

```
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)
```

在这样的 Makefile 中添加更多源文件只需要将其添加到定义了 SOURCE 变量的行中。

**其它资源**

- GNU make: 简介 - 2 一个 Makefile 简介
- 第 2 章 使用 GCC 构建代码

## 5.2. 示例：使用 MAKEFILE 构建 C 程序

按照本例中的步骤使用 Makefile 构建示例 C 程序。

**先决条件**

- 您必须了解 Makefile 和 **make** 的概念。

**流程**

1. 创建目录 **hellomake** 并改为此目录：

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

2. 创建包含以下内容的 **hello.c** 文件：

```
#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 **Makefile** 文件：

```
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 recipe 行必须以标签字符开头！从文档中复制上述文本时,横线过程可能会粘贴空格而不是标签页。如果发生了这种情况,请手动修正这个问题。

4. 运行 **make**:

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

这会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** 删除创建的文件：

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

**其它资源**

- 第 2.7 节 "示例：使用 GCC 构建 C 程序"
- 第 2.8 节 "示例：使用 GCC 构建 C++ 程序"

## 5.3. 文档资源 MAKE

有关 **make** 的详情请参考以下列出的资源。

**安装的文档**

- 使用 **man** 和 **info** 工具查看系统中安装的手动页面和信息页面：

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

**在线文档**

- *GNU Make Manual 由 Free Software Foundation 托管*
- *Red Hat Developer Toolset User Guide - 第 3 章。GNU make*

# 第 6 章 RHEL 7 后对 TOOLCHAIN 的更改

以下小节列出了自 Red Hat Enterprise Linux 7 中描述组件发行版本起的更改。另请参阅 Red Hat Enterprise Linux 8.0 发行注记。

## 6.1. RHEL 8 中的 GCC 的更改

在 Red Hat Enterprise Linux 8 中，GCC 工具链基于 GCC 8.2 发行系列。从 Red Hat Enterprise Linux 7 开始的显著变化包括：

- 添加了很多通用优化功能,如别名分析、向量器改进、相同的代码组合、流程间分析、存储合并优化通过等。
- 改进了 Address Sanitizer。
- 添加了用来检测内存泄漏的 Leak Sanitizer。
- 添加了用于检测未定义行为的 Undefined Behavior Sanitizer。
- 现在可使用 DWARF5 格式生成调试信息。这个功能是实验性的。
- 源代码覆盖分析工具 GCOV 已进行了各种改进。
- 添加了对 OpenMP 4.5 规格的支持。另外, OpenMP 4.0 规格的卸载功能现在被 C、C ++ 和 Fortran 编译器支持。
- 添加了新的警告并改进了诊断,以静态检测某些可能的编程错误。
- 现在,源位置被跟踪为范围而不是点,这样就可以进行更丰富的诊断。编译器现在提供"fix-it"提示,建议可能对代码进行修改。添加了一个检查程序来提供替代名称并可方便地检测拼写错误。

### Security

GCC 已被扩展，提供一些工具以确保增加生成的代码的强化。

如需了解更多详细信息,请参阅 第 6.2 节 “RHEL 8 中 GCC 的安全性增强” 。

### 构架和处理器支持

架构和处理器支持的改进包括：

- 为 Intel AVX-512 构架、多个微架构架构和 Intel Software Guard 扩展(SGX)添加了多个与架构相关的新选项。
- Code 生成可以针对 64 位 ARM 架构 LSE 扩展、ARMv8.2-A 16 位 Floating-Point 扩展(FPE)和 ARMv8.2-A、ARMv8.3-A 和 ARMv8.4-A 架构版本。
- 处理 ARM 和 64 位 ARM 架构的 **-march=native** 选项已被修复。
- 添加了对 64 位 IBM Z 架构的 z13 和 z14 处理器的支持。

### 语言和标准

与语言和标准有关的显著变化包括：

- C 语言编译代码时使用的默认标准已改为使用 GNU 扩展的 C17。
- C++ 语言编译代码时使用的默认标准已改为使用 GNU 扩展的 C++14。

- C++ 运行时程序库现在支持 C++11 和 C++14 标准。
- C++ 编译器现在以许多新功能实现 C++14 标准,如变量模板、非静态数据成员初始化工具、扩展 **constexpr** 规格、大小取消分配功能、通用 lambdas、变量边数数组、数字分隔器等。
- 改进了对 C 语言标准 C11 的支持：现在提供了 ISO C11 atomics、通用选择和线程存储。
- 新的 **__auto_type** GNU C 扩展提供了 C++11 **auto** 关键字功能的子集。
- ISO/IEC TS 18661-3:2015 标准指定的 **_FloatN** 和 **_FloatNx** 类型名称现在由 C 前端识别。
- C 语言编译代码时使用的默认标准已改为使用 GNU 扩展的 C17。这与使用 **--std=gnu17** 选项的效果相同。在以前的版本中，默认值是带有 GNU 扩展的 C89。
- GCC 现在可以使用 C++17 语言标准以及 C++20 标准中的某些功能对代码进行实验性编译。
- 现在，传递空类作为参数不包括在 Intel 64 和 AMD64 构架中，如平台 ABI 要求。传递或返回只删除复制的类,移动构造器现在使用相同的调用约定,与具有非商用副本或移动构造器的类相同。
- C++11 **alignof** operator 返回的值已被修正以匹配 C **_Alignof** operator 并返回最小协调。要查找首选的协调，请使用 GNU 扩展 **__alignof__**。
- 用于 Fortran 语言代码的 **libgfortran** 库的主要版本已改为 5。
- 对 Ada(GNAT)、GCC Go 和 Objective C/C++ 语言的支持已被删除。使用 Go Toolset 进行 Go 代码开发。

### 其它资源

- 另请参阅 Red Hat Enterprise Linux 8 发行注记 。
- 使用 Go 工具集

## 6.2. RHEL 8 中 GCC 的安全性增强

本节详细介绍了在 Red Hat Enterprise Linux 7.0 发行版本后，GCC 中与安全性相关的变化。

### 新警告

添加了这些警告选项：

| 选项 | 显示警告信息 |
| --- | --- |
| **-Wstringop-truncation** | 调用有绑定字符串操作功能,如 **strncat**、**strncpy** 和 **stpncpy**,这些功能可能会断开复制的字符串或使目的地不受影响。 |
| **-Wclass-memaccess** | 原始内存功能可能会以不安全的方式处理类型为非 trivial 类的对象,如 **memcpy** 或 **realloc**。<br><br>该警告有助于检测调用,绕过用户定义的构造器或复制分配 operator、损坏的虚拟表格指针、限定类型或参考的数据成员或者成员指针。该警告还会检测到可绕过数据成员的访问控制的调用。 |

| 选项 | 显示警告信息 |
|---|---|
| **-Wmisleading-indentation** | 代码缩进对于阅读代码的人可能会造成对代码块结构的误导。 |
| **-Walloc-size-larger-than=*size*** | 调用内存分配超过 *size* 的内存分配功能。还可用于通过乘以两个参数并使用属性 **alloc_size** 来指定分配的功能。 |
| **-Walloc-zero** | 调用内存分配功能,试图分配零内存。还可用于通过乘以两个参数并使用属性 **alloc_size** 来指定分配的功能。 |
| **-Walloca** | 所有对 **alloca** 功能的调用。 |
| **-Walloca-larger-than=*size*** | 请求内存大于 *size* 时调用 **alloca** 功能。 |
| **-Wvla-larger-than=*size*** | 可超过指定大小或者其绑定未知约束的 Variable Length Arrays(VLA)定义。 |
| **-Wformat-overflow=*level*** | 对格式化输出功能的 **sprintf** 系列调用时可能会出现特定和可能的缓冲溢出。有关 *level* 值的详情和说明，请参阅 *gcc(1)* 手册页。 |
| **-Wformat-truncation=*level*** | 对格式化输出功能的 **snprintf** 系列调用的特定和可能的输出截断。有关 *level* 值的详情和说明，请参阅 *gcc(1)* 手册页。 |
| **-Wstringop-overflow=*type*** | 对字符串处理功能（如 **memcpy** 和 **strcpy** ）调用的缓冲溢出。有关 *level* 值的详情和说明，请参阅 *gcc(1)* 手册页。 |

### *警告改进*

*改进了以下 GCC 警告：*

- *改进了 **-Warray-bounds** 选项,以检测更多绑定范围数组索引和指针偏移实例。例如,会检测到成灵活的阵列成员和字符串字面的负或过量索引。*
- *GCC 7 中引入的 **-Wrestrict** 选项已被改进,通过限制参数到标准内存和字符串操作功能（如 **memcpy** 和 **strcpy** ）来检测到多个对象的重叠访问实例。*
- ***-Wnonnull** 选项已被改进,以检测到一组更宽松的情况,将 null pointers 传递给希望使用非无效参数的功能（使用属性 **nonnull** 引用）。*

### *新的 UndefinedBehaviorSanitizer*

*添加了一个新的用于检测未定义行为的运行时清理程序，称为 UndefinedBehaviorSanitizer。以下选项需要加以注意：*

| 选项 | 检查 |
|---|---|
| **-fsanitize=float-divide-by-zero** | 检查浮点被被零除。 |
| **-fsanitize=float-cast-overflow** | 检查浮点类型到整数转换的结果是否溢出。 |

| 选项 | 检查 |
| --- | --- |
| **-fsanitize=bounds** | 启用阵列绑定控制并检测对边界外的访问。 |
| **-fsanitize=alignment** | 启用协调检查并检测各种没有对齐的对象。 |
| **-fsanitize=object-size** | 启用对象大小检查并检测到各种对边界外的访问。 |
| **-fsanitize=vptr** | 启用对 C++ 成员功能调用、成员访问以及指针到基本类别和派生类之间的一些转换。另外，检测引用的对象没有正确的动态类型。 |
| **-fsanitize=bounds-strict** | 启用对阵列绑定的严格的检查。这启用了 **-fsanitize=bounds** 并支持如灵活数组成员的数组。 |
| **-fsanitize=signed-integer-overflow** | 即使在使用通用向量的诊断操作中诊断异常溢出。 |
| **-fsanitize=builtin** | 在运行时诊断为 **__builtin_clz** 或 **__builtin_ctz** 前缀的内置无效参数。包括来自 **-fsanitize=undefined** 的检查。 |
| **-fsanitize=pointer-overflow** | 为指针嵌套执行 cheap run-time 测试。包括来自 **-fsanitize=undefined** 的检查。 |

### *AddressSanitizer 的新选项*

*这些选项已经被添加到 AddressSanitizer 中：*

| 选项 | 检查 |
| --- | --- |
| **-fsanitize=pointer-compare** | 指向不同内存对象的指针的警告。 |
| **-fsanitize=pointer-subtract** | 指向不同内存对象的指针的小数警告。 |
| **-fsanitize-address-use-after-scope** | 定义变量的范围后获取并使用其地址的 Thitize 变量。 |

### *其他清理程序和工具*

- *添加了选项 **-fstack-clash-protection**,以便在静态分配或动态分配堆栈空间时插入探测,以可靠检测到堆栈溢出,从而缓解依赖于操作系统提供的堆栈保护页面跳过的攻击向量。*
- *添加了一个新的选项 **-fcf-protection=[full|branch|return|none]** 来执行代码工具并提高程序安全性,方法是检查控制流传输指令的目标地址（如间接功能调用、函数返回、间接跳过）是否有效。*

### *其它资源*

- *有关为上述一些选项提供的值的详情和解释,请参见 gcc(1) 手册页：*

```
$ man gcc
```

## 6.3. RHEL 8 中 GCC 中破坏兼容性的更改

**C++ ABI 的更改 std::string 和 std::list**

**libstdc++** 库中的 **std::string** 和 **std::list** 类的 Application Binary Interface(ABI)在 RHEL 7(GCC 4.8)和 RHEL 8(GCC 8)间进行了更改,以符合 C++11 标准。**libstdc++** 库支持旧的和新的 ABI,但其他一些 C++ 系统库不支持。因此,需要重建针对这些库动态链接的应用程序。这会影响所有 C++ 标准模式,包括 C++98。它还会影响通过 Red Hat Developer Toolset 编译器为 RHEL 7 构建的应用程序,这些应用程序使旧的 ABI 保持与系统库的兼容性。

**GCC 不再构建 Ada、Go 和 Objective C/C++ 代码**

从 GCC 编译器中删除了 Ada(GNAT)、GCC Go 和 Objective C/C++ 语言构建代码的能力。

要构建 Go 代码,请使用 Go Toolset。

# 部分 II. 调试应用程序

调试应用程序是一个非常广泛的主题。这部分为开发人员提供了在多个情况下调试最常用的技术。

# 第 7 章 使用调试信息启用调试

要调试应用程序和库，需要调试信息。以下小节描述了如何获取这些信息。

## 7.1. 调试信息

在调试任何可执行代码时，两种类型的信息允许这些工具来比较二进制代码，进而进而是使用管理的二进制代码：

- 源代码文本
- 有关源代码文本与二进制代码的关系的描述

这些信息被称为调试信息。

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

要显示保存在 ELF 文件中的 DWARF 信息，请运行 **readelf -w file** 命令。

**小心**

STABS 是一个较老的、功能更低的格式，偶尔与 UNIX 一起使用。红帽不建议使用它。GCC 和 GDB 仅尽力提供 STABS 生产与消耗。其它工具，如 Valgrind 和 **elfutils** 无法用于 STABS。

**其它资源**

- DWARF 调试标准

## 7.2. 使用 GCC 启用 C 和 C++ 应用程序调试

由于调试信息较大，默认情况下不包含在可执行文件中。要启用 C 和 C++ 应用程序调试，您必须明确指示编译器创建它。

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执行代码，与原始源代码很难执行：变量可能会优化、循环未滚动，操作合并到相邻的源代码等。这会影响调试。要改进的调试体验，请考虑使用 **-Og** 选项设置优化功能。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，包括删除一些程序错误。
- 要在调试信息中包含宏定义，请使用 **-g3** 选项而不是 **-g**。
- **-fcompare-debug** GCC 选项测试由 GCC 编译的带有调试信息且没有调试信息的代码。如果得到的两个二进制文件相同，测试会传递。此测试可确保可执行代码不受任何调试选项的影响，这可以进一步确保 debug 代码中没有隐藏的错误。请注意，使用 **-fcompare-debug** 选项会显著增加编译时间。有关这个选项的详情，请查看 GCC 手册页。

**其它资源**

- 第 7 章 使用调试信息启用调试
- 使用 GNU Compiler Collection(GCC)- 调试您的选项

- 使用 GDB 进行 调试 - 在 Separate 文件中调试信息
- GCC 手册页：

```
$ man gcc
```

## 7.3. DEBUGINFO 和 DEBUGSOURCE 软件包

**debuginfo** 和 **debugsource** 软件包包含程序库的调试信息和调试源代码。对于 Red Hat Enterprise Linux 软件仓库中安装的应用程序和库,您可以从附加频道获取单独的 **debuginfo** 和 **debugsource** 软件包。

### 调试信息软件包类型

可用于调试的软件包有两种：

**debuginfo 软件包**

**debuginfo** 软件包提供了为二进制代码功能提供人类可读名称所需的调试信息。这些软件包包含包含 DWARF 调试信息的 **.debug** 文件。这些文件安装到 **/usr/lib/debug** 目录中。

**Debugsource 软件包**

**debugsource** 软件包包含用于编译二进制代码的源文件。安装对应的 **debuginfo** 和 **debugsource** 软件包后,如 GDB 或 LLDB 等调试程序可以将二进制代码的执行与源代码相关联。源代码文件安装到 **/usr/src/debug** 目录中。

### RHEL 7 的不同

在 Red Hat Enterprise Linux 7 中, **debuginfo** 软件包包含两种信息。Red Hat Enterprise Linux 8 将从 **debuginfo** 软件包中调试所需的源代码数据分成独立的 **debugsource** 软件包。

### 软件包名称

**debuginfo** 或 **debugsource** 软件包只针对具有相同名称、版本、发行和架构的二进制软件包提供有效的调试信息：

- 二进制软件包： **packagename-version-release.architecture.rpm**
- debuginfo 软件包： **packagename-debuginfo-version-release.architecture.rpm**
- Debugsource 软件包： **packagename-debugsource-version-release.architecture.rpm**

### 其它资源

- 第 7.1 节 "调试信息"
- 第 1.2 节 "启用调试和源存储库"

## 7.4. 使用 GDB 为应用程序或库获取 DEBUGINFO 软件包

调试代码需要调试信息。对于从软件包中安装的代码,GDB 会自动识别缺少的调试信息,解析软件包名称并为如何获得软件包提供具体建议。

### 先决条件

- 您要调试的应用程序或库必须安装在系统中。
- 必须在系统中安装 GDB 和 **debuginfo-install** 工具。

- 在系统中必须配置并启用提供 **debuginfo** 和 **debugsource** 软件包的频道。

**流程**

1. 启动附加到应用程序或您要调试的程序库的 GDB。GDB 自动识别缺少调试信息并建议运行命令。

```
$ gdb -q /bin/ls
Reading symbols from /bin/ls...Reading symbols from .gnu_debugdata for /usr/bin/ls...(no
debugging symbols found)...done.
(no debugging symbols found)...done.
Missing separate debuginfos, use: dnf debuginfo-install coreutils-8.30-6.el8.x86_64
(gdb)
```

2. 退出 GDB:输入 **q** 并确认 **Enter**。

```
(gdb) q
```

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

```
# dnf debuginfo-install coreutils-8.30-6.el8.x86_64
```

**dnf** 软件包管理工具提供更改概述,要求确认,并在确认、 下载和安装所有必要的文件后请求确认。

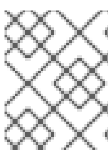
4. 如果 GDB 无法建议 **debuginfo** 软件包,请按照 第 7.5 节 "手动为应用程序或库获取 debuginfo 软件包" 中描述的步骤操作。

**其它资源**

- Red Hat Developer Toolset User Guide,"安装调试信息"部分
- 如何为 RHEL 系统下载或安装 debuginfo 软件包？ - 红帽知识库解决方案

## 7.5. 手动为应用程序或库获取 DEBUGINFO 软件包

您可以通过找到可执行文件,然后查找安装该文件的软件包来手动确定您需要安装哪些 **debuginfo** 软件包。



**注意**

红帽建议您 使用 GDB 来决定安装的软件包。只有在 GDB 无法建议安装软件包时使用这个手动步骤。

**先决条件**

- 该应用程序或库必须安装在系统中。
- 应用程序或库是从软件包中安装的。
- **debuginfo-install** 工具必须在系统中可用。
- 必须在系统中配置并启用提供 **debuginfo** 软件包的频道。

**流程**

1. 查找应用程序或库的可执行文件。

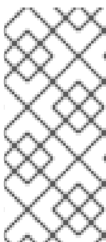
   a. 使用 **which** 命令查找应用程序文件。

   ```
   $ which less
   /usr/bin/less
   ```

   b. 使用 **locate** 命令查找库文件。

   ```
   $ locate libz | grep so
   /usr/lib64/libz.so.1
   /usr/lib64/libz.so.1.2.11
   ```

   如果调试的原始原因包括错误消息,请选择库的文件名中具有与错误消息中所述相同附加号码的结果。如果有疑问,请按照其余部分步骤进行尝试,从而使库文件名称不包含额外的数字。

   **注意**

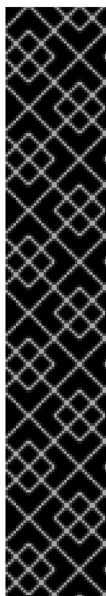
   **locate** 命令由 **mlocate** 软件包提供。安装并启用使用：

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

2. 搜索提供文件的软件包的名称和版本：

   ```
   $ rpm -qf /usr/lib64/libz.so.1.2.7
   zlib-1.2.11-10.el8.x86_64
   ```

   输出提供了 名称 为epoch-version. release.架构 格式的软件包详情。

   **重要**

   如果这个步骤没有生成任何结果,则无法决定哪个软件包提供了二进制文件。可能会有几个可能的情况：

   - 该文件从 当前 配置中未知软件包管理工具的软件包中安装。
   - 该文件从本地下载并手动安装的软件包安装。在这种情况下,无法自动决定合适的 **debuginfo** 软件包。
   - 软件包管理工具被错误配置。
   - 该文件不会从任何软件包安装。在这种情况下,不存在对应的 **debuginfo** 软件包。

   由于进一步的步骤取决于这种情况,您必须解决这种情况,或者中止这个过程。描述准确的故障排除步骤超出了此过程的范围。

3. 使用 **debuginfo-install** 工具安装 **debuginfo** 软件包。在命令中,使用在上一步中决定的软件包名称和其他详情：

   ```
   # debuginfo-install zlib-1.2.11-10.el8.x86_64
   ```

■

***其它资源***

- *Red Hat Developer Toolset User Guide,"安装调试信息"部分*
- *如何为 RHEL 系统下载或安装 debuginfo 软件包？ - 知识库文章*

# 第 8 章 使用 GDB 检查应用程序 INTERNAL STATE

要查找为什么应用程序无法正常工作，控制其执行，并使用 debugger 检查其内部状态。这部分论述了如何在这个任务中使用 GNU Debugger(GDB)。

## 8.1. GNU DEBUGGER(GDB)

Red Hat Enterprise Linux 包含 GNU debugger(GDB)，它可让您通过命令行用户界面调查程序内发生的情况。

对于 GDB 的图形前端，安装 Eclipse 集成开发环境。请参阅 使用 Eclipse。

### GDB 功能

单个 GDB 会话可以调试以下类型的程序：

- 多线程和请求程序
- 同时多个程序
- 通过 TCP/IP 网络连接在远程机器或带有 **gdbserver** 工具的容器上的程序

### 调试要求

要调试任何可执行代码，GDB 需要该特定代码的调试信息：

- 对于您开发的程序，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，必须安装其 debuginfo 软件包。

## 8.2. 将 GDB 附加到进程

要检查进程，GDB 必须 附加 到该进程。

### 先决条件

- 必须在系统中安装 GDB

### 使用 GDB 启动程序

当程序没有作为进程运行时，使用 GDB 启动它：

```
$ gdb program
```

使用到程序的文件名或路径替换 program。

GDB 设置至开始执行程序。在使用 **run** 命令开始执行进程前，您可以设置 breakpoints 和 **gdb** 环境。

### 将 GDB 附加到已在运行的进程中

将 GDB 附加到已作为一个进程运行的程序中：

1. 使用 **ps** 命令查找进程 ID(pid):

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

使用到程序的文件名或路径替换 program。

2. 将 GDB 附加到此过程中：

```
$ gdb -p pid
```

使用 **ps** 输出中的实际进程 ID 号替换 pid。

### 将已在运行的 GDB 附加到已在运行的进程中

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):

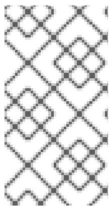
```
(gdb) shell ps -C program -o pid h
 pid
```

使用到程序的文件名或路径替换 program。

2. 使用 **attach** 命令将 GDB 附加到程序：

```
(gdb) attach pid
```

使用 **ps** 输出中的实际进程 ID 号替换 pid。



**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 Invoking GDB
- 使用 GDB 进行调试 - 4.7 调试 Already-running 过程

## 8.3. 使用 GDB 通过程序代码步骤

当 **GDB 调试程序** 被附加到某个程序后,您可以使用一些命令来控制该程序的执行。

### 先决条件

- 您必须有可用的调试信息：
  - 使用调试信息编译并构建该程序,或者
  - 已安装相关的 debuginfo 软件包
- 必须将 GDB 附加到程序才能调试

### 逐步实现代码的 GDB 命令

**r**（运行）

*开始程序的执行。如果 **run** 使用了任何参数来执行,则这些参数会像程序正常启动一样传递给可执行文件。用户通常在设置了 breakpoint 后发布这个命令。*

***start***

*开始执行程序,但在程序主功能开始时停止。如果 **start** 使用了任何参数来执行,则这些参数会像程序正常启动一样传递给可执行文件。*

***c (持续)***

*从当前状态继续执行程序。这个程序的执行将继续执行,直到以下条件之一被满足:*

- *已达到断开点。*
- *满足指定条件。*
- *这个程序收到信号。*
- *发生错误。*
- *程序终止。*

***n (next)***

*从当前状态继续执行程序,直到达到当前源文件中的下行代码。这个程序的执行将继续执行,直到以下条件之一被满足:*

- *已达到断开点。*
- *满足指定条件。*
- *这个程序收到信号。*
- *发生错误。*
- *程序终止。*

***s (步骤)***

***step** 命令还停止在当前源文件中的每个连续代码行中执行。但是,如果执行当前在包含 **函数调用的源行中停止**, GDB 在输入功能调用(而不是执行它)后会停止执行。*

***until 位置***

*继续执行操作,直到达到由 location 选项指定的代码位置。*

***fini (finish)***

*恢复程序的执行,并在从功能返回时停止执行。这个程序的执行将继续执行,直到以下条件之一被满足:*

- *已达到断开点。*
- *满足指定条件。*
- *这个程序收到信号。*
- *发生错误。*
- *程序终止。*

***q (quit)***

*终止执行并退出 GDB。*

**其它资源**

- 第 8.5 节 "使用 GDB 中断点停止在定义的代码位置执行"
- 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

## 8.4. 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做什么非常重要。GDB 提供了多个可以用来检查内部变量的命令。本节描述了这些命令中最有用的：

**p (print)**

显示给定参数的值。通常,参数是任何复杂性的变量名称,从简单单值到结构。参数也可以是当前语言中有效的表达式,包括使用程序变量和库功能,或者正在测试的程序中定义的功能。
您可以使用 **print** 命令使用 pretty-printer Python 或 Guile 脚本扩展 GDB,以自定义显示数据结构（比如类、 破坏）。

**bt (backtrace)**

显示用于访问当前执行点的功能调用链,或者显示在执行被终止前使用的函数链。这对调查带有故障原因的严重错误（比如分段错误）非常有用。
在 **backtrace** 命令中添加 **full** 选项也会显示本地变量。

您可以使用 **bt** 和 **info frame** 命令自定义显示数据时,使用 帧过滤器 Python 脚本扩展 GDB。术语 框架 是 指与单一功能调用关联的数据。

**info**

**info** 命令是一个通用命令,用于提供有关各种项目的信息。 它使用一个选项来指定要描述的项目。

- **info args** 命令显示目前选择的框架功能调用的选项。
- **info locals** 命令显示当前选择的帧中的本地变量。

对于可能的项目列表,请在 GDB 会话中运行 **help info** 命令：

```
(gdb) help info
```

**l (list)**

显示程序停止的源代码中的行。这个命令仅在程序执行停止时才可用。虽然这个命令并不严格显示内部状态, **list** 可帮助用户了解在程序执行的下一步中将会对内部状态进行哪些更改。

**其它资源**

- GDB Python API - Red Hat Developers Blog 条目
- 使用 GDB 进行调试 - 10.9 Pretty Printing

## 8.5. 使用 GDB 中断点停止在定义的代码位置执行

通常只调查少量代码。Breakpoints 是告诉 GDB 在代码的特定位置停止执行某个程序的一个标记。Breakpoints 与源代码行最常见关联。在这种情况下,放置一个 breakpoint 需要指定源文件和行号。

- 要 **设置断开点,请执行以下操作**:
  - 指定源代码 文件 的名称以及该文件中的 行:

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

  - 当 文件 不存在时,会使用当前执行点的源文件名称：

    ```
    (gdb) br line
    ```

  - 或者,使用函数名称将 breakpoint 放在它的启动中：

    ```
    (gdb) br function_name
    ```

- 在某个任务的某些迭代后,某个程序可能会遇到错误。指定要停止 **执行的额外条件**:

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

  使用 C 或 C++ 语言的条件替换 condition。文件 和 行 的含义与以上相同。

- 检查 所有 breakpoints 和 watchpoint 的状态：

  ```
  (gdb) info br
  ```

- 使用 **info br** 输出中显示的 数字 来 **删除** 中断点：

  ```
  (gdb) delete number
  ```

- **删除** 给定位置的 breakpoint:

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

**其它资源**

- 使用 GDB 调试 – 5.1 Breakpoints, Watchpoints, and Catchpoints

## 8.6. 使用 GDB 监视点在数据访问和更改时停止执行

在很多情况下,让程序执行直到某些数据更改或被访问为止。本节列出了最常见的用例。

**先决条件**

- 了解 GDB

**在 GDB 中使用监视点**

Watchpoints 是告诉 **GDB** 停止执行某个程序的标记。Watchpoints 与 data 关联：放置一个监视点需要指定描述变量、多个变量或内存地址的表达式。

- 要 **放置** 用于数据 **更改的监视点** （写）:

  ```
  (gdb) watch expression
  ```

使用描述要监视的 表达式 替换 expression。对于变量, 表达式 等于变量名称。

- 为 **数据** 访问设置监视点 （读取）:

```
(gdb) rwatch expression
```

- 为 **任何** 数据访问（读写） **放置** 一个监视点：

```
(gdb) awatch expression
```

- 检查 所有监视点和 breakpoint 的状态：

```
(gdb) info br
```

- 删除监视 点：

```
(gdb) delete num
```

使用 **info br** 命令报告的数量替换 **num** 选项。

**其它资源**

- 使用 GDB 进行调试 - 5.1.2 Setting Watchpoints

## 8.7. 使用 GDB 调试或线程程序

有些程序使用 forking 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**

- 您必须了解调用和线程的概念。

### 使用 GDB 调试已分叉的程序

当某个程序（父）生成其自身（子）的独立副本时, Forking 是一种情况。使用以下设置和命令来影响 GDB 在 fork 发生时的作用：

- **follow-fork-mode** 设置控制 GDB 是否跟随分叉后的父对象或子级。

**set follow-fork-mode parent**
  在 fork 后,调试父进程。这是默认值。

**set follow-fork-mode child**
  在 fork 后,调试子进程。

**show follow-fork-mode**
  显示 **follow-fork-mode** 的当前设置。

- **set detach-on-fork** 设置控制 GDB 是否对另一个进程（未遵循）进行控制或将其保留运行。

**set detach-on-fork on**
  未遵循的进程（取决于 **follow-fork-mode**的值）会被分离并独立运行。这是默认值。

**set detach-on-fork off**

GDB 保持对两个进程的控制。接下来的进程（取决于 follow-fork-mode的值）像通常一样调试,另一个会被挂起。

**show detach-on-fork**

显示 detach-on-fork 的当前设置。

### 使用 GDB 调试 Threaded Programs

GDB 能够调试独立的线程,并可以独立操作和检查它们。要使 GDB 只停止检查的线程,请使用命令 set non-stop on 和set target-async on。您可以将这些命令添加到 .gdbinit 文件中。启用该功能后,GDB 就可以进行线程调试。

GDB 使用 当前线程的概念。默认情况下,命令只应用到当前的线程。

**info threads**

使用 id 和gid 号显示线程列表,显示当前的线程。

**thread id**

使用指定的 id 将线程设置为当前的线程。

**thread apply ids command**

将 command 命令应用到ids 列出的所有线程。ids 选项是一个以空格分开的线程 ID 列表。特殊值all 会将命令应用到所有线程。

**break location thread id if condition**

在特定 location 的condition 中设置一个 breakpoint,它只针对线程数id。

**watch expression thread id**

仅为线程数 id 设置expression 定义的监视点。

**command&**

执行命令 command 并立即返回到 gdb 提示符(gdb),在后台继续任何代码执行。

**interrupt**

在后台停止执行。

### 其它资源

- 使用 GDB 进行调试 - 4.10 Debugging Programs 使用多 Threads
- 使用 GDB 进行调试 - 4.11 调试 Fork

# 第 9 章 记录应用程序互动

应用程序的可执行代码与操作系统代码和共享库交互。记录这些交互的活动日志可让您深入了解应用程序的行为,而无需调试实际应用程序代码。或者,分析应用程序的交互有助于找出错误清单的条件。

## 9.1. 用于记录应用程序交互的工具

Red Hat Enterprise Linux 提供多种工具来分析应用程序的交互。

strace

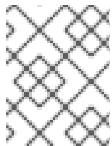
**strace** 工具主要启用应用程序所用系统调用（内核功能）的日志记录。

- **strace** 工具可提供有关调用的详细输出,因为 **strace** 在了解底层内核代码的情况下解释参数和结果。数字被转换为对应的恒定名称,将位符合并到标志列表,指向字符数组来解引用,以提供实际字符串,等等。可能不支持较新的内核功能。
- 您可以过滤追踪的调用来减少捕获的数据的数量。
- 使用 **strace** 不需要任何特定的设置,除了设置日志过滤器外。
- 使用 **strace** 追踪应用程序代码会导致应用程序的执行显著下降。因此, **strace** 不适合很多生产部署。作为替代,请考虑使用 **ltrace** 或者 SystemTap。
- Red Hat Developer Toolset 中可用的 **strace** 版本也可以执行系统调用 tampering。这个功能对调试非常有用。

ltrace

**ltrace** 工具启用将应用程序的用户空间调用记录到共享对象（动态库）中。

- **ltrace** 工具启用对任何库的追踪调用。
- 您可以过滤追踪的调用来减少捕获的数据的数量。
- 使用 **ltrace** 不需要任何特定的设置,除了设置日志过滤器外。
- **ltrace** 工具是轻量级和快速的,提供了 **strace** 的替代方案：可以追踪库中的 **glibc** 中的对应接口,并使用 **ltrace** 来追踪使用 **strace** 的内核功能。
- 因为 **ltrace** 没有处理类似 **strace** 的已知调用,所以它不会尝试解释传递给库功能的值。**ltrace** 输出只包含原始数字和指针。解释 **ltrace** 输出需要咨询输出中库的实际接口声明。



注意

在 Red Hat Enterprise Linux 8.0 中,一个已知问题会阻止 **ltrace** 追踪系统可执行文件。这个限制不适用于用户构建的可执行文件。

SystemTap

SystemTap 是一个工具平台,用于在 Linux 系统上模拟运行的进程和内核活动。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,脚本化日志记录意味着在初始设置阶段可以进行更多的工作。但是,脚本功能会扩展了 SystemTap 的功能,它超出了生成日志的范围。

- SystemTap 通过创建和插入内核模块来实现。SystemTap 的使用效率高,不会对系统或应用程序本身造成显著下降。
- SystemTap 附带一组使用示例。

GDB

GNU Debugger(GDB)主要用于调试,而不是记录。然而,在某些特性中,即使应用程序互动是关注的主要活动,也很有用。

- 使用 GDB,可以方便地将交互事件的捕获与立即调试后续执行路径合并。
- 在被其他工具初始发现有问题的情况后,GDB 最适合对罕见或单数事件进行分析。在出现频繁事件的任何场景中使用 GDB 将会变得效率低下甚至无法使用。

**其它资源**

- Red Hat Enterprise Linux SystemTap Beginners Guide
- Red Hat Developer Toolset User Guide

## 9.2. 使用 STRACE 监控应用程序的系统调用

**strace** 工具启用监控应用程序执行的系统（内核）调用。

**先决条件**

- 您必须在系统中安装了 **strace**。

**流程**

1. 识别要监控的系统调用。
2. 启动 **strace** 并将其附加到程序。
   - 如果您想要监控的程序没有运行,请启动 **strace** 并指定 *程序*:

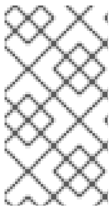
     ```
     $ strace -fvttTyy -s 256 -e trace=call program
     ```

   - 如果程序已在运行,找到其进程 id(pid)并把 **strace** 附加到它：

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

   - 使用要显示的系统调用替换 *call*。您可以多次使用该 **-e trace=*call*** 选项。如果省略, **strace** 将显示所有系统调用类型。详情请查看 strace(1) 手册页。
   - 如果您不想追踪所有 fork 进程或线程,请保留 **-f** 选项。
3. **strace** 工具显示应用程序的系统调用及其详情。
   在大多数情况下,如果没有为系统调用设置过滤,则应用程序及其库会发出大量调用, **strace** 输出会立即出现。

4. strace 工具在程序退出时退出。
   要在追踪程序退出前终止监控，按 Ctrl+C。
   - 如果 strace 启动程序，该程序会与 strace 一起终止。
   - 如果您将 strace 附加到已在运行的程序，该程序会与 strace 一起终止。
5. 分析应用程序的系统调用列表。
   - 日志中存在资源访问或可用性的问题，因为调用返回的错误。
   - 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行为的原因。
   - 如果应用程序崩溃，则可能需要在日志末尾获得重要信息。
   - 输出会包含大量不必要的信息。但是，您可以为系统调用构建一个更加精确的过滤器，并重复这个过程。



注意

查看输出并将其保存到文件中是很有好处的。使用 tee 命令达到此目的：

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

其它资源

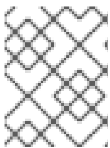
- strace(1) 手册页：

```
$ man strace
```

- 如何使用 strace 跟踪命令发出的系统调用？- 知识库文章
- Red Hat Developer Toolset User Guide - chapter strace

## 9.3. 使用 LTRACE 监控应用程序的库功能

ltrace 工具允许监控应用程序对库（共享对象）中可用功能的调用。



注意

在 Red Hat Enterprise Linux 8.0 中，一个已知问题会阻止 ltrace 追踪系统可执行文件。这个限制不适用于用户构建的可执行文件。

先决条件

- 您必须在系统中安装了 ltrace。

流程

1. 若有可能，识别相关的库和功能。
2. 启动 ltrace 并将其附加到程序。
   - 如果您想要监控的程序没有运行，请启动 ltrace 并指定program:

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

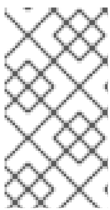
- 如果程序已在运行，找到其进程 id(pid)并把 ltrace 附加到它：

```
$ ps -C program
(...)
$ ltrace -f -l library -e function program -ppid
```

- 使用 -e、-f 和 -l 选项过滤输出：
  - 提供要显示为功能的 功能 名称。-e function 选项可以多次使用。如果省略，ltrace 会显示对所有功能的调用。
  - 您可以使用 -l library 选项指定整个库，而不指定功能。这个选项的行为与 -e function 选项类似。
  - 如果您不想追踪所有 fork 进程或线程，请保留 -f 选项。

  如需更多信息，请参阅 ltrace(1)_ 手册页。

3. ltrace 显示应用程序发出的库调用。
   在大多数情况下，如果没有设置过滤器，应用程序会发出大量调用，ltrace 输出会立即显示。
4. ltrace 程序退出时退出。
   要在追踪程序退出前终止监控，按 ctrl+C。
   - 如果 ltrace 启动程序，该程序会与 ltrace 一起终止。
   - 如果您将 ltrace 附加到已在运行的程序，该程序会与 ltrace 一起终止。
5. 分析应用程序进行的库调用列表。
   - 如果应用程序崩溃，则可能需要在日志末尾获得重要信息。
   - 输出会包含大量不必要的信息。但是，您可以构造一个更精确的过滤器并重复这个步骤。



**注意**

查看输出并将其保存到文件中是很有好处的。使用 tee 命令达到此目的：

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

**其它资源**

- ltrace(1) 手册页：

```
$ man ltrace
```

- Red Hat Developer Toolset User Guide - chapter ltrace

## 9.4. 使用 SYSTEMTAP 监控应用程序的系统调用

SystemTap 工具允许为内核事件注册自定义事件处理程序。与 **strace** 工具相比,更难以使用,但效率更高,并启用更复杂的处理逻辑。一个名为 **strace.stp** 的 SystemTap 脚本与 SystemTap 一起安装,并使用 SystemTap 提供 **strace** 功能的近似值。

**先决条件**

- SystemTap 和相关的内核软件包必须安装在系统中。

**流程**

1. 查找您要监控的进程 ID(pid):

```
$ ps -aux
```

2. 使用 **strace.stp** 脚本运行 SystemTap:

```
# stap /usr/share/systemtap/examples/process/strace.stp -x pid
```

**pid** 的值是进程 id。

该脚本被编译到内核模块中,然后载入该模块。这在输入命令和获取输出之间引入了一些延迟。

3. 当进程执行系统调用时,调用名称及其参数会输出到终端。

4. 当进程终止或按 **Ctrl**+**C** 键时,该脚本会退出。

## 9.5. 使用 GDB 拦截应用程序系统调用

GNU Debugger(GDB)可让您在程序执行过程中的不同情况下停止执行。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**

- 您必须了解 GDB 断开点的使用。
- GDB 必须附加到程序。

**流程**

1. 设置 catchpoint:

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

**catch syscall** 命令设置一种特殊的 breakpoint 类型,它会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 在任何系统调用中停止。

2. 开始执行程序。

   - 如果程序还没有启动执行,请启动它:

```
(gdb) r
```

- 如果停止了程序执行,恢复它:

```
(gdb) c
```

3. GDB 在执行程序执行任何指定的系统调用后停止执行。

### 其它资源

- 第 8.4 节 "使用 GDB 显示程序内部值"
- 第 8.3 节 "使用 GDB 通过程序代码步骤"
- 使用 GDB 进行调试 - 设置 Watchpoints

## 9.6. 使用 GDB 截取应用程序对信号的处理

GNU Debugger(GDB)可让您在程序执行过程中的不同情况下停止执行。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

- 您必须了解 GDB 断开点的使用。
- GDB 必须附加到程序。

### 流程

1. 设置 catchpoint:

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

**catch signal** 命令设置一种特殊的 breakpoint 类型,它会在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. 让程序运行。
   - 如果程序还没有启动执行,请启动它:

```
(gdb) r
```

   - 如果停止了程序执行,恢复它:

```
(gdb) c
```

3. GDB 在程序接收任何指定的信号后停止执行。

### 其它资源

- 第 8.4 节 "使用 GDB 显示程序内部值"
- 第 8.3 节 "使用 GDB 通过程序代码步骤"
- 使用 GDB 调试 – 5.1.3 Setting Catchpoints

# 第 10 章 调试 CRASHED APPLICATION

有时,无法直接调试应用程序。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,之后再分析它。

## 10.1. CORE DUMPS:它们的状态以及如何使用它们

core 转储是应用程序停止工作时应用程序内存的一部分副本,以 ELF 格式保存。它包含所有应用程序的内部变量和堆栈,允许检查应用程序的最终状态。当使用对应的可执行文件和调试信息增强时,可以使用调试器分析内核转储文件,其方式类似于分析正在运行的程序。

如果启用了这个功能,Linux 操作系统内核可以自动记录内核转储。或者,您可以向任何正在运行的应用程序发送信号来生成一个内核转储,而不考虑其实际状态。

**警告**

有些限制可能会影响生成内核转储的能力。查看当前的限制:

```
$ ulimit -a
```

## 10.2. 使用内核转储记录应用程序崩溃

要记录应用程序崩溃,请设置内核转储保存并添加有关系统的信息。

**流程**

1. 要启用内核转储,请确保 **/etc/systemd/system.conf** 文件包含以下行:

```
DumpCore=yes
DefaultLimitCORE=infinity
```

您还可以添加注释来描述之前是否有这些设置,以及前面的值。这可让您在以后根据需要撤销这些更改。注释是以 # 字符开头的行。

更改文件需要管理员级别的访问权限。

2. 应用新配置:

```
# systemctl daemon-reexec
```

3. 删除内核转储大小的限制:

```
# ulimit -c unlimited
```

要撤销这个更改,使用值为 **0** 而不是 **unlimited** 运行命令。

4. 安装提供 **sosreport** 工具来收集系统信息的 **sos** 软件包:

```
# yum install sos
```

5. 当应用程序崩溃时,内核转储由 **systemd-coredump** 生成和处理。

6. 创建一个 SOS 报告来提供有关系统的额外信息：

```
# sosreport
```

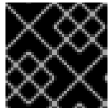
这会创建一个包含系统信息的 **.tar** 归档,比如配置文件的副本。

7. 找到并导出内核转储：

```
$ coredumpctl list executable-name
$ coredumpctl dump executable-name > /path/to/file-for-export
```

如果应用程序多次崩溃,第一个命令的输出会列出更多捕获的内核转储。在这种情况下,为第二个命令构造一个使用其他信息更精确的查询。详情请查看 coredumpctl(1) 手册页。

8. 将 core 转储和 SOS 报告传送到进行调试的计算机。传输可执行文件（如果已知）。



**重要**

当可执行文件未知时,对核心文件的后续分析会识别该文件。

9. 可选：在传输后删除内核转储和 SOS 报告,从而释放磁盘空间。

**其它资源**

- 文档中的 配置基本系统设置中的 systemd 简介
- 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？- 知识库文章

## 10.3. 使用内核转储检查应用程序崩溃状态

**先决条件**

- 您必须有一个内核转储文件,且来自发生崩溃的系统 sosreport。
- GDB 和 elfutils 必须安装在您的系统中。

**流程**

1. 要识别发生崩溃的可执行文件,使用内核转储文件运行 **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
```

输出包含一行中每个模块的详情,用空格分开。信息按以下顺序列出:

1. 映射该模块的内存地址
2. 模块的 build-id 以及它找到的内存的位置
3. 模块的可执行文件名,在未知时以 - 的形式显示,或者在 . 没有从文件中载入该模块时显示
4. 调试信息源,在可用时以文件名形式显示,当可执行文件本身中包含时作为 . 显示,或者在不存在时显示为 -
5. 主模块的共享库名称(soname)或 [exe]

在这个示例中,重要详情是文件名 **/usr/bin/sleep** 和包含文本 [exe] 的行中的 build-id **2818b2009547f780a5639c904cded443e564973e**。使用这些信息,您可以识别分析内核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。使用从核心文件提取的文件名。
   - 您还可以在系统中使用相同的可执行文件。基于 Red Hat Enterprise Linux 构建的每个可执行文件都包含一个带有唯一 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
     ```

     使用这些信息将远程系统中的可执行文件与您的本地副本匹配。在 core 转储中列出的本地文件的 build-id 和 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** 输出查找所需软件包的准确版本。

3. 获取可执行文件使用的共享库。使用与可执行文件相同的步骤。

4. 如果应用程序作为软件包发布,在 GDB 中载入可执行文件,为缺少的 debuginfo 软件包显示 hints。如需了解更多详细信息,请参阅 第 7.4 节 "使用 GDB 为应用程序或库获取 debuginfo 软件包"。

5. 要详细检查核心文件,请使用 GDB 加载可执行文件和内核转储文件:

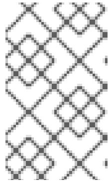
   ```
   $ gdb -e executable_file -c core_file
   ```

   有关缺少文件和调试信息的其他信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。

   如果应用程序的调试信息可作为文件提供,而不是软件包,请使用 **symbol-file** 命令在 GDB 中载入该文件:

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

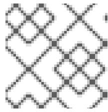
   使用实际文件名替换 program.debug。



注意

可能不需要为 core 转储中包含的所有可执行文件安装调试信息。大多数可执行文件都是应用程序代码使用的库。这些库可能不会直接造成您分析的问题,您不需要包括调试信息。

6. 使用 GDB 命令在应用程序崩溃时检查应用程序的状态。请查看 第 8 章 使用 GDB 检查应用程序 Internal State。



注意

在分析核心文件时,GDB 不会附加到正在运行的进程中。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

## 10.4. 使用 COREDUMPCTL 创建并访问内核转储

systemd 的 coredumpctl 工具可显著简化在发生崩溃的机器中使用内核转储的过程。该流程概述了如何捕获无响应过程的内核转储。

先决条件

- 系统必须配置为使用 systemd-coredump 进行内核转储处理。验证它为 true:

```
$ sysctl kernel.core_pattern
```

如果输出以以下内容开头,则配置正确：

```
kernel.core_pattern = |/usr/lib/systemd/systemd-coredump
```

流程

1. 根据可执行文件名称的已知部分查找 hung 进程的 PID:

```
$ pgrep -a executable-name-fragment
```

这个命令将以表单输出一行

```
PID command-line
```

使用 命令行 值验证PID 是否属于预期进程。

例如：

```
$ pgrep -a bc
5459 bc
```

2. 向进程发送中止信号：

```
# kill -ABRT PID
```

3. 验证内核是否已被 **coredumpctl** 捕获：

```
$ coredumpctl list PID
```

例如：

```
$ coredumpctl list 5459
TIME                          PID   UID   GID SIG COREFILE  EXE
Thu 2019-11-07 15:14:46 CET    5459  1000  1000   6 present   /usr/bin/bc
```

4. 根据需要进一步检查或使用核心文件。
您可以使用 PID 指定 core 转储和其他值。详情请 查看 coredumpctl(1) 手册页。

- 显示核心文件的详情：

```
$ coredumpctl info PID
```

- 在 GDB 调试器中载入内核文件：

```
$ coredumpctl debug PID
```

根据调试信息的可用性,GDB 会建议运行命令,例如：

```
Missing separate debuginfos, use: dnf debuginfo-install bc-1.07.1-5.el8.x86_64
```

有关此过程的详情请参考 第 7.4 节 “使用 GDB 为应用程序或库获取 debuginfo 软件包”。

- 要导出核心文件以便在其他位置进一步处理：

```
$ coredumpctl dump PID > /path/to/file_for_export
```

将 /path/to/file_for_export 替换为您要放置 core 转储的文件。

## 10.5. 使用它转储进程内存 GCORE

内核转储调试的工作流允许分析程序离线状态。在某些情况下,您可以将此工作流与仍在运行的程序一起使用,例如,使用这个过程很难访问环境。在进程仍然运行时,您可以使用 **gcore** 命令转储任何进程的内存。

**先决条件**

- 您必须了解内核转储是什么以及如何创建它们。
- GDB 必须在系统中安装。

**流程**

1. 找到进程 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
```

这会创建一个文件 文件名,并转储其中的进程内存。在转储内存时,进程的执行会停止。

3. 在 core 转储完成后,该过程会恢复正常执行。
4. 创建一个 SOS 报告来提供有关系统的额外信息：

```
# sosreport
```

这会创建一个包含系统信息的 tar 归档,比如配置文件的副本。

5. 将程序的可执行文件、内核转储和 SOS 报告传送到进行调试的计算机。
6. 可选：在传输后删除内核转储和 SOS 报告,从而释放磁盘空间。

**其它资源**

- 如何在不重启应用程序的情况下获得核心文件？ - 知识库文章

## 10.6. 使用 GDB 转储受保护进程内存

您可以将进程内存标记为不转储。这可保存资源,并在进程内存包含敏感数据时确保额外的安全性：例如,在企业或核算应用程序或整个虚拟机上。内核内核转储(kdump)和手动内核转储（gcore、GDB）都不会以这种方式标记的转储内存。

在某些情况下,无论这些保护是什么,您必须转储进程内存的整个内容。此流程演示了如何使用 GDB 调试程序进行此操作。

**先决条件**

- 您必须了解内核转储是什么。
- GDB 必须在系统中安装。
- GDB 必须已附加到带有受保护内存的进程。

**流程**

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 - 如何从您的计划中提取核心文件*

# 第 11 章 GDB 中破坏兼容性的更改

Red Hat Enterprise Linux 8 中提供的 GDB 版本包含很多与兼容性问题相关的更改,特别是直接从终端读取 GDB 输出的情况。以下小节详细介绍了这些更改。

不建议解析 GDB 的输出。使用 Python GDB API 或 GDB Machine Interface(MI)的首选脚本。

## gdbserver 现在使用 shell 启动 feriors

要启用扩展和变量替换命令行参数,GDBserver 现在与 GDB 一样在 shell 中启动下级。

使用 shell 禁用：

- 使用 **target extended-remote** GDB 命令时,使用 **set startup-with-shell off** 命令禁用 shell。
- 当使用 **target remote** GDB 命令时,使用 **--no-startup-with-shell** 选项的 GDBserver 禁用 shell。

> 例 11.1. 远程 GDB 推断器中的 shell 扩展示例
>
> 这个示例演示了在 Red Hat Enterprise Linux 版本 7 和 8 中通过 GDBserver 运行 **/bin/echo** /* 命令的不同：
>
> - 对于 RHEL 7:
>
> ```
> $ gdbserver --multi :1234
> $ gdb -batch -ex 'target extended-remote :1234' -ex 'set remote exec-file /bin/echo' -ex
> 'file /bin/echo' -ex 'run /*'
> /*
> ```
>
> - 对于 RHEL 8:
>
> ```
> $ gdbserver --multi :1234
> $ gdb -batch -ex 'target extended-remote :1234' -ex 'set remote exec-file /bin/echo' -ex
> 'file /bin/echo' -ex 'run /*'
> /bin /boot (...) /tmp /usr /var
> ```

## gcj 支持已删除

支持通过 GNU Compiler for Java(**gcj**)编译的调试 Java 程序已被删除。

## 符号转储维护命令的新语法

符号转储维护命令语法现在包含文件名前的选项。因此,在 RHEL 7 中使用 GDB 的命令无法在 RHEL 8 中正常工作。

例如,以下命令不再将符号存储在文件中,但会生成出错信息：

```
(gdb) maintenance print symbols /tmp/out main.c
```

符号转储维护命令的新语法是：

```
maint print symbols [-pc address] [--] [filename]
maint print symbols [-objfile objfile] [-source source] [--] [filename]
maint print psymbols [-objfile objfile] [-pc address] [--] [filename]
maint print psymbols [-objfile objfile] [-source source] [--] [filename]
maint print msymbols [-objfile objfile] [--] [filename]
```

### 线程数不再是全局的

在以前的版本中,GDB 只使用全局线程编号。该编号已扩展为以 **inferior_num.thread_num** 格式为每个下级显示,如 2.1。因此, **$_thread** 方便变量和 **InferiorThread.num** Python 属性中的线程数字在下级之间不再是唯一的。

GDB 现在为每个线程保存第二个线程 ID,称为全局线程 ID,这是之前版本中的线程数的新值。要访问全局线程号,请使用 **$_gthread** 方便变量和 **InferiorThread.global_num** Python 属性。

为了向后兼容,Machine Interface(MI)线程 ID 始终包含全局 ID。

**例 11.2. GDB 线程数更改示例**

在 Red Hat Enterprise Linux 7 上：

```
# debuginfo-install coreutils
$ gdb -batch -ex 'file echo' -ex start -ex 'add-inferior' -ex 'inferior 2' -ex 'file echo' -ex start -ex 'info
threads' -ex 'pring $_thread' -ex 'inferior 1' -ex 'pring $_thread'
(...)
  Id   Target Id         Frame
* 2    process 203923 "echo" main (argc=1, argv=0x7fffffffdb88) at src/echo.c:109
  1    process 203914 "echo" main (argc=1, argv=0x7fffffffdb88) at src/echo.c:109
$1 = 2
(...)
$2 = 1
```

在 Red Hat Enterprise Linux 8 中：

```
# dnf debuginfo-install coreutils
$ gdb -batch -ex 'file echo' -ex start -ex 'add-inferior' -ex 'inferior 2' -ex 'file echo' -ex start -ex 'info
threads' -ex 'pring $_thread' -ex 'inferior 1' -ex 'pring $_thread'
(...)
  Id   Target Id         Frame
  1.1  process 4106488 "echo" main (argc=1, argv=0x7fffffffce58) at ../src/echo.c:109
* 2.1  process 4106494 "echo" main (argc=1, argv=0x7fffffffce58) at ../src/echo.c:109
$1 = 1
(...)
$2 = 1
```

### 值内容的内存可能会受限制

在以前的版本中，GDB 不会限制为值内容分配的内存量。因此，调试不正确的程序可能会导致 GDB 分配过多的内存。添加了 **max-value-size** 设置来限制分配的内存量。这个限制的默认值为 64 KiB。因此，Red Hat Enterprise Linux 8 中的 GDB 不会显示太大的值，而是会报告这个值太大。

例如,打印一个定义为 **char s[128*1024];** 的值会产生不同的结果：

- 在 Red Hat Enterprise Linux 7 中, **$1 = 'A' <repeats 131072 times>**
- 在 Red Hat Enterprise Linux 8 中, **value requires 131072 bytes, which is more than max-value-size**

### 不再支持 Sun 版本的 stabs 格式

对 Sun 版本的 **stabs** 调试文件格式的支持已删除。GDB 仍支持 GCC 在 RHEL 中使用**gcc -gstabs** 选项生成的 **stabs** 格式。

### sysroot 处理更改

set sysroot path 命令在搜索调试所需文件时指定系统根。现在,为这个命令提供的目录名可能会带有字符串 target: 前缀,使 GDB 从目标系统中（本地和远程）读取共享的库。以前可用的remote: 前缀现在被视为 target:。另外,默认的系统根值已从空字符串改为 target:,以便向后兼容。

指定的系统 root 先于主可执行文件名,当 GDB 远程启动时,或者它附加到已在运行的进程（本地和远程）。这意味着,对于远程进程,默认值 target: 使 GDB 总是尝试从远程系统加载调试信息。要防止这种情况,请在 target remote 命令前运行 set sysroot 命令以便在远程符号前找到本地符号文件。

### HISTSIZE 不再控制 GDB 命令历史大小

在以前的版本中,GDB 使用 HISTSIZE 环境变量来决定命令历史记录应保留的时长。GDB 已被修改为使用 GDBHISTSIZE 环境变量。该变量只适用于 GDB。可能的值及其影响如下：

- 一个正数 - 使用这个大小的命令历史记录,
- -1 或者空字符串 - 保留所有命令的历史记录,
- 非数字值 - 忽略。

### 添加了完成限制

现在,可以使用 set max-completions 命令限制完成期间考虑的最大候选数。要显示当前的限制,请运行 show max-completions 命令。默认值为 200。这个限制可防止 GDB 生成过大的完成列表且变得无响应。

例如,输入 p <tab><tab> 后的输出为：

- 对于 RHEL 7: **Display all 29863 possibilities? (y or n)**
- 对于 RHEL 8: **Display all 200 possibilities? (y or n)**

### 删除了 HP-UX XDB 兼容性模式

HP-UX XDB 兼容模式的 **-xdb** 选项已从 GDB 中删除。

### 为线程处理信号

在以前的版本中,GDB 可以向当前的线程发送信号,而不是发送信号的线程。这个程序错误已被解决,GDB 现在会在恢复执行时将信号传递给正确的线程。

另外,现在 signal 命令总是正确地向当前的线程发送请求的信号。如果程序停止了信号并且用户切换了线程,GDB 需要确认。

### breakpoint 模式总是关闭并自动合并

已更改 breakpoint always-inserted 设置。已删除 auto 值和对应行为。默认值现在为 off。另外, off 值现在会导致 GDB 在所有线程停止前不会从目标中删除断点。

### 不再支持 remotebaud 命令

不再支持 set remotebaud 和 show remotebaud 命令。使用 set serial baud 和 show serial baud 命令替代。

# 部分 III. 用于开发的额外工具集

***除了可作为操作系统一部分的开发相关工具外,开发人员还可以在 Red Hat Enterprise Linux 中安装附加工具集。这些工具集可以包含不同语言、替代工具链或者系统工具的替代版本的工具。***

# 第 12 章 使用 GCC TOOLSET

## 12.1. 什么是 GCC TOOLSET

Red Hat Enterprise Linux 8 引进了 GCC Toolset，它是一个 Application Stream，其中包含最新的开发和性能分析工具版本。GCC Toolset 与 RHEL 7 的 Red Hat Developer Toolset 类似。

GCC Toolset 作为 Application Stream 提供，其格式为 **AppStream** 存储库中的软件集合。GCC Toolset 在 Red Hat Enterprise Linux 订阅级别协议中被完全支持。它的功能是完整的，并适用于生产环境。GCC Toolset 提供的应用程序和库不会取代 Red Hat Enterprise Linux 系统版本，不要覆盖它们，且不会自动成为默认选择或首选选择。使用名为软件集合的框架，一组额外的开发人员工具被安装到 **/opt/** 目录中，并在需要时用户使用 **scl** 工具显式启用。除非对特定工具或功能另有说明，GCC Toolset 可用于 Red Hat Enterprise Linux 支持的所有架构。

## 12.2. 安装 GCC TOOLSET

在系统上安装 GCC Toolset 会安装主工具和所有需要的依赖项。请注意，工具集的一些部分默认不会安装，且必须单独安装。

### 流程

- 安装 GCC Toolset 版本 N:

```
# yum install gcc-toolset-N
```

## 12.3. 从 GCC TOOLSET 安装单独的软件包

要只从 GCC Toolset 安装某些工具而不是整个工具集，请列出可用软件包，并使用 **yum** 软件包管理工具安装所选工具。这个过程对工具集没有默认安装的软件包也很有用。

### 流程

1. 列出 GCC Toolset 版本 N 中的可用软件包：

```
$ yum list available gcc-toolset-N-\*
```

2. 安装这些软件包：

```
# yum install package_name
```

使用空格分开的软件包列表替换 package_name。例如，要安装 **gcc-toolset-9-gdb-gdbserver** 和 **gcc-toolset-9-gdb-doc** 软件包：

```
# yum install gcc-toolset-9-gdb-gdbserver gcc-toolset-9-gdb-doc
```

## 12.4. 卸载 GCC TOOLSET

要从您的系统中删除 GCC Toolset，请使用 **yum** 软件包管理工具卸载它。

### 流程

- 卸载 GCC Toolset 版本 N:

```
# yum remove gcc-toolset-N\*
```

## 12.5. 从 GCC TOOLSET 运行工具

要从 GCC Toolset 运行工具,使用 **scl** 实用程序。

### 流程

- 要从 GCC Toolset 版本 N 运行工具：

```
$ scl enable gcc-toolset-N tool
```

## 12.6. 使用 GCC TOOLSET 运行 SHELL 会话

GCC Toolset 允许运行 GCC Toolset 工具版本而不是这些工具的系统版本的 shell 会话,而无需明确使用 **scl** 命令。当您需要多次以互动方式启动工具时（如设置或测试开发设置时）,这非常有用。

### 流程

- 要运行来自 GCC Toolset 版本 N 的工具版本的 shell 会话,覆盖这些工具的系统版本：

```
$ scl enable gcc-toolset-N bash
```

## 12.7. 相关信息

- Red Hat Developer Toolset User Guide

# 第 13 章 GCC TOOLSET 9

*本章提供与 GCC Toolset 版本 9 相关的信息,以及本版本中包含的工具。*

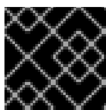
## 13.1. GCC TOOLSET 9 提供的工具和版本

*GCC Toolset 9 提供以下工具和版本：*

*表 13.1. GCC Toolset 9 中的工具版本*

| 名称 | 版本 | 描述 |
|---|---|---|
| GCC | 9.2.1 | 支持 C、C ++ 和 Fortran 的可移植编译器套件。 |
| GDB | 8.3 | 用于使用 C、C ++ 和 Fortran 编写的程序的命令行调试程序。 |
| Valgrind | 3.15.0 | 工具框架和多个用来配置集应用程序的工具,以便检测内存错误、识别内存管理问题并报告系统调用中任何不正确参数的使用情况。 |
| SystemTap | 4.1 | 一个追踪和模拟工具,用于监控整个系统的活动,而无需工具、重新编译、安装和重启。 |
| Dyninst | 10.1.0 | 用于在执行期间检测和使用用户空间可执行文件的库。 |
| binutils | 2.32 | 一组二进制工具和其他工具来检查和操作对象文件和二进制文件。 |
| elfutils | 0.176 | 一组二进制工具和其他工具来检查和操作 ELF 文件。 |
| dwz | 0.12 | 一个用来优化 ELF 共享库和 ELF 执行文件中的 DWARF 调试信息的工具。 |
| make | 4.2.1 | 一个依赖项跟踪构建自动化工具。 |
| strace | 5.1 | 用于监控程序使用和信号的系统调用的调试工具。 |
| ltrace | 0.7.91 | 用于显示对程序动态库调用的调试工具。它还可监控程序执行的系统调用。 |
| annobin | 9.08 | 构建安全检查工具。 |

## 13.2. GCC TOOLSET 9 中的 C++ 兼容性



*重要*

*这里介绍的兼容性信息只适用于 GCC Toolset 9 中的 GCC。*

*GCC Toolset 中的 GCC 编译器可使用以下 C++ 标准：*

*C++14*

这是 GCC Toolset 9 的 默认 语言标准设置,包含 GNU 扩展名,等同于明确使用选项 **-std=gnu++14**。
当所有使用 GCC 版本 6 或更高版本构建了带有相应标志的 C++ 对象时,支持使用 C++14 语言版本。

C++11

GCC Toolset 9 提供了这种语言标准。
当所有使用 GCC 版本 5 或更高版本构建了带有相应标志的 C++ 对象时,支持使用 C++11 语言版本。

C++98

GCC Toolset 9 提供了这种语言标准。无论是否使用 GCC Toolset、Red Hat Developer Toolset 和 RHEL 5、6、7 和 8 构建,都可以使用这个标准构建二进制文件、共享库和对象。

C++17, C++2a

这些语言标准仅作为实验性、不稳定且不受支持的能力在 GCC Toolset 9 中找到。此外,无法保证使用这些规则构建的对象、二进制文件和库的兼容性。

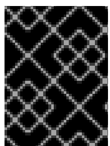
所有语言标准都可用于标准变体或 GNU 扩展中。

当将使用 GCC Toolset 构建的对象与 RHEL 工具链构建的对象(通常为 **.o** 或 **.a** 文件)混合时, GCC Toolset 工具链应该用于任何此项。这样可确保在链接时间解决所有仅由 GCC Toolset 提供的新库功能。

## 13.3. GCC TOOLSET 9 中的 GCC 的具体设置

### 库的静态链接

某些最新的库功能会静态链接到通过 GCC Toolset 构建的应用程序,以支持在多个 Red Hat Enterprise Linux 版本上执行。这会创建额外的安全风险,因为标准 Red Hat Enterprise Linux 勘误不会更改此代码。如果因为这种风险,开发人员需要重建其应用程序,红帽会使用安全勘误来进行通讯。



重要

由于这个额外的安全风险,强烈建议开发人员不要因为同样的原因静态连接其整个应用程序。

### 链接时在对象文件后指定库

在 GCC Toolset 中,库使用 linker 脚本链接,这些脚本可以通过静态归档指定一些符号。这需要保证与多个 Red Hat Enterprise Linux 版本兼容。但是, linker 脚本使用相应的共享对象文件的名称。因此,链接器使用不同于预期的符号处理规则,且在指定对象文件选项前指定了库添加选项时,无法识别对象文件所需的符号:

```
$ scl enable gcc-toolset-9 'gcc -lsomelib objfile.o'
```

使用来自 GCC Toolset 的库会导致 linker 错误消息 **undefined reference to symbol**。要防止这个问题,请遵循标准链接实践,并在指定对象文件选项后指定库:

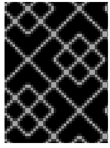
```
$ scl enable gcc-toolset-9 'gcc objfile.o -lsomelib'
```

请注意,在使用基本 Red Hat Enterprise Linux 版本 GCC 时,此建议也适用于此建议。

## 13.4. GCC TOOLSET 9 中的 BINUTILS 的具体设置

### 库的静态链接

某些最新的库功能会静态链接到通过 GCC Toolset 构建的应用程序,以支持在多个 Red Hat Enterprise Linux 版本上执行。这会创建额外的安全风险,因为标准 Red Hat Enterprise Linux 勘误不会更改此代码。如果因为这种风险,开发人员需要重建其应用程序,红帽会使用安全勘误来进行通讯。



重要

由于这个额外的安全风险,强烈建议开发人员不要因为同样的原因静态连接其整个应用程序。

### 链接时在对象文件后指定库

在 GCC Toolset 中,库使用 linker 脚本链接,这些脚本可以通过静态归档指定一些符号。这需要保证与多个 Red Hat Enterprise Linux 版本兼容。但是, linker 脚本使用相应的共享对象文件的名称。因此,链接器使用不同于预期的符号处理规则,且在指定对象文件选项前指定了库添加选项时,无法识别对象文件所需的符号：

```
$ scl enable gcc-toolset-9 'ld -lsomelib objfile.o'
```

使用来自 GCC Toolset 的库会导致 linker 错误消息 **undefined reference to symbol**。要防止这个问题,请遵循标准链接实践,并在指定对象文件选项后指定库：

```
$ scl enable gcc-toolset-9 'ld objfile.o -lsomelib'
```

请注意,在使用基本 Red Hat Enterprise Linux 版本的 binutils 时,建议也适用。

# *第 14 章 GCC TOOLSET 10*

***本章提供与 GCC Toolset 版本 10 相关的信息,以及此版本中包含的工具。***

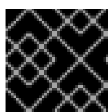
## *14.1. GCC TOOLSET 10 提供的工具和版本*

***GCC Toolset 10 提供以下工具和版本：***

***表 14.1. GCC Toolset 10 中的工具版本***

| 名称 | 版本 | 描述 |
|---|---|---|
| GCC | 10.2.1 | 支持 C、C ++ 和 Fortran 的可移植编译器套件。 |
| GDB | 9.2 | 用于使用 C、C ++ 和 Fortran 编写的程序的命令行调试程序。 |
| Valgrind | 3.16.0 | 工具框架和多个用来配置集应用程序的工具,以便检测内存错误、识别内存管理问题并报告系统调用中任何不正确参数的使用情况。 |
| SystemTap | 4.4 | 一个追踪和模拟工具,用于监控整个系统的活动,而无需工具、重新编译、安装和重启。 |
| Dyninst | 10.2.1 | 用于在执行期间检测和使用用户空间可执行文件的库。 |
| binutils | 2.35 | 一组二进制工具和其他工具来检查和操作对象文件和二进制文件。 |
| elfutils | 0.182 | 一组二进制工具和其他工具来检查和操作 ELF 文件。 |
| dwz | 0.12 | 一个用来优化 ELF 共享库和 ELF 执行文件中的 DWARF 调试信息的工具。 |
| make | 4.2.1 | 一个依赖项跟踪构建自动化工具。 |
| strace | 5.7 | 用于监控程序使用和信号的系统调用的调试工具。 |
| ltrace | 0.7.91 | 用于显示对程序动态库调用的调试工具。它还可监控程序执行的系统调用。 |
| annobin | 9.29 | 构建安全检查工具。 |

## *14.2. GCC TOOLSET 10 中的 C++ 兼容性*



***重要***

***这里介绍的兼容性信息只适用于 GCC Toolset 10 中的 GCC。***

***GCC Toolset 中的 GCC 编译器可使用以下 C++ 标准：***

***C++14***

这是 GCC Toolset 10 的 默认 语言标准设置,带有 GNU 扩展名,等同于明确使用选项 **-std=gnu++14**。
当所有使用 GCC 版本 6 或更高版本构建了带有相应标志的 C++ 对象时,支持使用 C++14 语言版本。

C++11

这个语言标准可在 GCC Toolset 10 中找到。
当所有使用 GCC 版本 5 或更高版本构建了带有相应标志的 C++ 对象时,支持使用 C++11 语言版本。

C++98

这个语言标准可在 GCC Toolset 10 中找到。无论是否使用 GCC Toolset、Red Hat Developer Toolset 和 RHEL 5、6、7 和 8 构建,都可以使用这个标准构建二进制文件、共享库和对象。

C++17

这个语言标准可在 GCC Toolset 10 中找到。

C++20

这个语言标准只在 GCC Toolset 10 中作为实验性、不稳定且不受支持的能力提供。另外,无法保证使用这个标准构建的对象、二进制文件和库的兼容性。

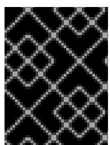
所有语言标准都可用于标准变体或 GNU 扩展中。

当将使用 GCC Toolset 构建的对象与 RHEL 工具链构建的对象(通常为 **.o** 或 **.a** 文件)混合时, GCC Toolset 工具链应该用于任何此项。这样可确保在链接时间解决所有仅由 GCC Toolset 提供的新库功能。

## 14.3. GCC TOOLSET 10 中的 GCC 的具体设置

### 库的静态链接

某些最新的库功能会静态链接到通过 GCC Toolset 构建的应用程序,以支持在多个 Red Hat Enterprise Linux 版本上执行。这会创建额外的安全风险,因为标准 Red Hat Enterprise Linux 勘误不会更改此代码。如果因为这种风险,开发人员需要重建其应用程序,红帽会使用安全勘误来进行通讯。



重要

由于这个额外的安全风险,强烈建议开发人员不要因为同样的原因静态连接其整个应用程序。

### 链接时在对象文件后指定库

在 GCC Toolset 中,库使用 linker 脚本链接,这些脚本可以通过静态归档指定一些符号。这需要保证与多个 Red Hat Enterprise Linux 版本兼容。但是, linker 脚本使用相应的共享对象文件的名称。因此,链接器使用不同于预期的符号处理规则,且在指定对象文件选项前指定了库添加选项时,无法识别对象文件所需的符号:

```
$ scl enable gcc-toolset-10 'gcc -lsomelib objfile.o'
```

使用来自 GCC Toolset 的库会导致 linker 错误消息 **undefined reference to symbol**。要防止这个问题,请遵循标准链接实践,并在指定对象文件选项后指定库:

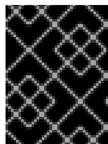
```
$ scl enable gcc-toolset-10 'gcc objfile.o -lsomelib'
```

请注意,在使用基本 Red Hat Enterprise Linux 版本 GCC 时,此建议也适用于此建议。

## 14.4. GCC TOOLSET 10 中的 BINUTILS 的具体设置

### 库的静态链接

某些最新的库功能会静态链接到通过 GCC Toolset 构建的应用程序,以支持在多个 Red Hat Enterprise Linux 版本上执行。这会创建额外的安全风险,因为标准 Red Hat Enterprise Linux 勘误不会更改此代码。如果因为这种风险,开发人员需要重建其应用程序,红帽会使用安全勘误来进行通讯。



**重要**

由于这个额外的安全风险,强烈建议开发人员不要因为同样的原因静态连接其整个应用程序。

### 链接时在对象文件后指定库

在 GCC Toolset 中,库使用 linker 脚本链接,这些脚本可以通过静态归档指定一些符号。这需要保证与多个 Red Hat Enterprise Linux 版本兼容。但是, linker 脚本使用相应的共享对象文件的名称。因此,链接器使用不同于预期的符号处理规则,且在指定对象文件选项前指定了库添加选项时,无法识别对象文件所需的符号：

```
$ scl enable gcc-toolset-10 'ld -lsomelib objfile.o'
```

使用来自 GCC Toolset 的库会导致 linker 错误消息 **undefined reference to symbol**。要防止这个问题,请遵循标准链接实践,并在指定对象文件选项后指定库：

```
$ scl enable gcc-toolset-10 'ld objfile.o -lsomelib'
```

请注意,在使用基本 Red Hat Enterprise Linux 版本的 binutils 时,建议也适用。

# 第 15 章 使用 GCC TOOLSET 容器镜像

GCC Toolset 10 组件在两个容器镜像中可用：

- GCC Toolset 10 Toolchain
- GCC Toolset 10 Perftools

GCC Toolset 容器镜像基于 **rhel8** 基础镜像,并可用于 RHEL 8 支持的所有架构：

- AMD 和 Intel 64 位构架
- 64 位 ARM 架构
- IBM Power Systems, Little Endian
- 64-bit IBM Z

## 15.1. GCC TOOLSET 容器镜像内容

GCC Toolset 10 容器镜像中提供的工具版本与 GCC Toolset 10 组件版本匹配。

### GCC Toolset 10 Toolchain 内容

**rhel8/gcc-toolset-10-toolchain** 镜像提供 GCC 编译器、GDB 调试器和其他与开发相关的工具。容器镜像由以下组件组成：

| 组件 | 软件包 |
| --- | --- |
| **gcc** | gcc-toolset-10-gcc |
| **g++** | gcc-toolset-10-gcc-c++ |
| **gfortran** | gcc-toolset-10-gcc-gfortran |
| **gdb** | gcc-toolset-10-gdb |

### GCC Toolset 10 Perftools 内容

**rhel8/gcc-toolset-10-perftools** 镜像提供很多用于调试、性能监控和进一步分析应用程序的工具。容器镜像由以下组件组成：

| 组件 | 软件包 |
| --- | --- |
| **Valgrind** | gcc-toolset-10-valgrind |
| **SystemTap** | gcc-toolset-10-systemtap |
| **Dyninst** | gcc-toolset-10-dyninst |
| **elfutils** | gcc-toolset-10-elfutils |

**其它资源**

- 要在 RHEL 7 上使用 GCC Toolset 组件，请使用 Red Hat Developer Toolset 为 RHEL 7 用户提供类似的开发工具 - Red Hat Developer Toolset 用户指南。
- 在 RHEL 7 上使用 Red Hat Developer Toolset 容器镜像的说明 - Red Hat Developer Toolset 镜像。

## 15.2. 访问并运行 GCC TOOLSET 容器镜像

下面的部分论述了如何访问和运行 GCC Toolset 容器镜像。

**先决条件**

- 已安装 podman。

**流程**

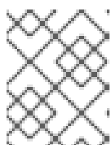
1. 使用您的客户门户网站凭证访问 Red Hat Container Registry：

```
$ podman login registry.redhat.io
Username: username
Password: ********
```

2. 以 root 用户身份运行相关命令来拉取所需的容器镜像：

```
# podman pull registry.redhat.io/rhel8/gcc-toolset-10-toolchain
```

```
# podman pull registry.redhat.io/rhel8/gcc-toolset-10-perftools
```



**注意**

在 RHEL 8.1 及更新的版本中，您可以设置系统以非 root 用户身份使用容器。详情请参阅以 root 用户或无根用户身份运行容器。

3. 可选：通过运行列出本地系统中所有容器镜像的命令来检查拉取是否成功：

```
# podman images
```

4. 通过在容器内启动 bash shell 来运行容器：

```
# podman run -it image_name /bin/bash
```

**-i** 选项会创建一个交互式会话；如果没有这个选项，shell 将打开并立即退出。

**-t** 选项会打开终端会话；如果没有这个选项，您就无法输入任何 shell。

**其它资源**

- 在 RHEL 8 中构建、运行和管理 Linux 容器
- 红帽博客文章 - 了解容器内部和外部的根用户

- Red Hat Container Registry 中的条目 - GCC Toolset 容器镜像

## 15.3. 示例：使用 GCC TOOLSET 10 TOOLCHAIN 容器镜像

本例演示了如何拉取和开始使用 GCC Toolset 10 Toolchain 容器镜像。

### 先决条件

- 已安装 podman。

### 流程

1. 使用您的客户门户网站凭证访问 Red Hat Container Registry:

```
$ podman login registry.redhat.io
Username: username
Password: ********
```

2. 将容器镜像拉取为 root:

```
# podman pull registry.redhat.io/rhel8/gcc-toolset-10-toolchain
```

3. 以 root 用户身份使用互动 shell 启动容器镜像：

```
# podman run -it registry.redhat.io/rhel8/gcc-toolset-10-toolchain /bin/bash
```

4. 正常运行 GCC Toolset 工具。例如,要验证 gcc 编译器版本,请运行：

```
bash-4.4$ gcc -v
...
gcc version 10.2.1 20200804 (Red Hat 10.2.1-2) (GCC)
```

5. 要列出容器中提供的所有软件包,请运行：

```
bash-4.4$ rpm -qa
```

## 15.4. 使用 GCC TOOLSET 10 PERFTOOLS 容器镜像中的 SYSTEMTAP

SystemTap 工具发布在 GCC Toolset 10 Perftools 容器镜像中。要使用这个工具,请按照以下步骤执行。

### 先决条件

- GCC Toolset 10 Perftools 容器镜像会被拉取。

### 流程

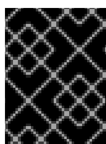
1. 使用超级用户权限运行镜像：

```
$ podman run -u root -it --privileged --ipc=host --net=host --pid=host
registry.redhat.io/rhel8/gcc-toolset-{gcct-ver}-perftools /bin/bash
```

如需了解更多有关 超级特权容器 的信息,请参阅 运行超级特权容器。

2. 确保容器中安装以下软件包或安装它们：

   - **kernel**
   - **kernel-devel**
   - **kernel-debuginfo**



**重要**

上面的 **kernel** 软件包的版本和发行号必须与主机上运行的内核的版本和发行号匹配。

   - 要检查主机系统内核的版本和发行号，请运行：

```
$ uname -r
4.18.0-193.el8.x86_64
```

   - 要安装与软件包相匹配的版本，请运行带有 **uname** 命令输出的软件包安装命令。例如：

```
# yum install kernel-devel-$(uname -r)
```

   - 要安装 **kernel-debuginfo** 软件包，首先以 root 用户身份运行以下命令来启用 **debug** 存储库：

```
# subscription-manager repos --enable=rhel-8-for-x86_64-baseos-debug-rpms
```

   要了解在 RHEL 系统上安装 **debuginfo** 软件包的更多信息，请参阅如何为 RHEL 系统下载或安装 debuginfo 软件包？

3. 可选：要避免重复这些步骤并在以后重复使用预先配置的容器，请考虑运行以下命令来保存它：

```
$ podman commit new-container-image-name
```

# 第 16 章 编译器工具集

*RHEL 8 提供以下编译器工具集作为 Application Streams:*

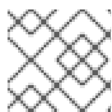
- *LLVM Toolset 11.0.0,它提供 LLVM 编译器基础架构框架、C 和 C++ 语言的 Clang 编译器、LLDB 调试器以及相关代码分析工具。请参阅 使用 LLVM Toolset 指南。*
- *rust Toolset 1.49.0,它提供 Rust 编程语言编译器* ***rustc*****、*****cargo*** *构建工具和依赖项管理器、* ***cargo-vendor*** *插件以及所需的库。请参阅 使用 Rust Toolset 指南。*
- *Go Toolset 1.15.7,它提供 Go 编程语言工具和程序库。Go 也称为* ***golang*****。***请参阅 使用 Go Toolset 指南。*

# 第 17 章 ANNOBIN 项目

Annobin 项目是 Setmark 规格项目的实现。水标规格项目旨在将标记添加到可执行和可链接格式(ELF)对象中,以确定它们的属性。Annobin 项目由 annobin 插件和 annockeck 程序组成。

annobin 插件扫描 GNU Compiler Collection(GCC)命令行、编译状态和编译过程,并生成 ELF 备注。ELF 记录如何构建二进制文件,并为 annocheck 程序提供执行安全强化检查的信息。

安全强化检查程序是 annocheck 程序的一部分,并默认启用。它检查二进制文件,以确定程序是否使用必要的安全强化选项构建并正确编译。annocheck 可以为 ELF 对象文件递归扫描目录、归档和 RPM 软件包。



**注意**

文件必须采用 ELF 格式。annocheck 不要处理任何其他二进制文件类型。

下面的部分描述了如何：

- 使用 annobin 插件
- 使用 annocheck 程序
- 删除冗余 annobin 备注

## 17.1. 使用 ANNOBIN 插件

下面的部分描述了如何：

- 启用 annobin 插件
- 将选项传递给 annobin 插件

### 17.1.1. 启用 annobin 插件

下面的部分论述了如何通过 gcc 和 clang 启用annobin 插件。

**流程**

- 要启用带有 gcc 的annobin 插件,请使用：

  ```
  $ gcc -fplugin=annobin
  ```

  - 如果 gcc 没有找到annobin 插件,请使用：

    ```
    $ gcc -iplugindir=/path/to/directory/containing/annobin/
    ```

    使用 包含 annobin 的目录的绝对路径替换 /path/to/directory/con contain/annobin/。

  - 要查找包含 annobin 插件的目录,请使用：

    ```
    $ gcc --print-file-name=plugin
    ```

- 要启用带有 clang 的annobin 插件,请使用：

```
$ clang -fplugin=/path/to/directory/containing/annobin/
```

使用 包含 annobin 的目录的绝对路径替换 /path/to/directory/con contain/annobin/。

### 17.1.2. 将选项传递给 annobin 插件

下面的部分论述了如何通过 gcc 或通过 clang 将选项传递给 annobin 插件。

**流程**

- 要将选项传递给带有 gcc 的annobin 插件,请使用：

  ```
  $ gcc -fplugin=annobin -fplugin-arg-annobin-option file-name
  ```

  使用 annobin 命令行参数替换 选项,并使用文件名 替换 file-name。

  **示例**

  - 要显示 annobin 正在做什么的更多详情,请使用：

    ```
    $ gcc -fplugin=annobin -fplugin-arg-annobin-verbose file-name
    ```

    用文件名替换 file-name。

- 要将选项传递给带有 clang 的annobin 插件,请使用：

  ```
  $ clang -fplugin=/path/to/directory/containing/annobin/ -Xclang -plugin-arg-annobin -Xclang
  option file-name
  ```

  使用 annobin 命令行参数替换 选项,并使用 包含 annobin 的目录的绝对路径替换 /path/to/directory/con contain/annobin/。

  **示例**

  - 要显示 annobin 正在做什么的更多详情,请使用：

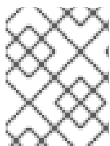
    ```
    $ clang -fplugin=/usr/lib64/clang/10/lib/annobin.so -Xclang -plugin-arg-annobin -Xclang
    verbose file-name
    ```

    用文件名替换 file-name。

## 17.2. 使用 ANNOCHECK 程序

下面的部分论述了如何使用 annocheck 检查：

- 文件
- 目录
- RPM 软件包
- annocheck 额外工具



**注意**

**annocheck** 为 ELF 对象文件递归扫描目录、归档和 RPM 软件包。文件必须采用 ELF 格式。**annocheck** 不要处理任何其他二进制文件类型。

### 17.2.1. 使用 annocheck 检查文件

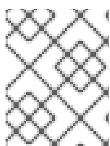
下面的部分论述了如何使用 **annocheck** 检查 ELF 文件。

**流程**

- 要检查文件,请使用：

```
$ annocheck file-name
```

  使用 文件名 替换 file-name。



**注意**

文件必须采用 ELF 格式。**annocheck** 不要处理任何其他二进制文件类型。**annocheck** 处理包含 ELF 对象文件的静态库。

**附加信息**

- 有关 **annocheck** 和可能的命令行选项的详情请参考 **annocheck** man page。

### 17.2.2. 使用 annocheck 检查目录

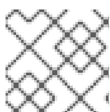
下面的部分论述了如何使用 **annocheck** 检查目录中的 ELF 文件。

**流程**

- 要扫描目录,请使用：

```
$ annocheck directory-name
```

  使用 目录名 替换 directory-name。**annocheck** 自动检查目录、其子目录以及目录中的所有归档和 RPM 软件包的内容。



**注意**

**annocheck** 只查找 ELF 文件。其他文件类型将被忽略。

**附加信息**

- 有关 **annocheck** 和可能的命令行选项的详情请参考 **annocheck** man page。

### 17.2.3. 使用 annocheck 检查 RPM 软件包

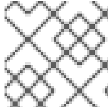
下面的部分论述了如何使用 **annocheck** 检查 RPM 软件包中的 ELF 文件。

**流程**

- 要扫描 RPM 软件包，请使用：

```
$ annocheck rpm-package-name
```

使用 RPM 软件包的名称替换 rpm- package-name。**annocheck** 递归扫描 RPM 软件包中的所有 ELF 文件。



**注意**

**annocheck** 只查找 ELF 文件。其他文件类型将被忽略。

- 要使用提供的 debug info RPM 扫描 RPM 软件包，请使用：

```
$ annocheck rpm-package-name --debug-rpm debuginfo-rpm
```

使用 RPM 软件包的名称替换 rpm- package-name，debuginfo-rpm 替换为与二进制 RPM 关联的 debug info RPM 的名称。

**附加信息**

- 有关 **annocheck** 和可能的命令行选项的详情请参考 **annocheck** man page。

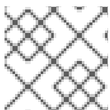
### 17.2.4. 使用 annocheck 额外工具

**annocheck** 包括多个用来检查二进制文件的工具。您可以使用命令行选项启用这些工具。

下面的部分描述了如何启用：

- **built-by** 工具
- **notes** 工具
- **section-size** 工具

您可以同时启用多个工具。



**注意**

默认启用强化检查程序。

#### 17.2.4.1. 启用 built-by 工具

您可以使用 **annocheck built-by** 工具查找构建二进制文件的编译器名称。

**流程**

- 要启用 **built-by** 工具，请使用：

```
$ annocheck --enable-built-by
```

**附加信息**

- 有关 **built-by** 工具的详情请参考 **--help** 命令行选项。

#### 17.2.4.2. 启用 notes 工具

您可以使用 annocheck notes 工具显示存储在 annobin 插件创建的二进制文件中的备注。

**流程**

- 要启用 notes 工具，请使用：

```
$ annocheck --enable-notes
```

  备注显示在按地址范围排序的序列中。

**附加信息**

- 有关 notes 工具的详情请参考 **--help** 命令行选项。

#### 17.2.4.3. 启用 section-size 工具

您可以使用 annocheck section-size 工具显示命名部分的大小。

**流程**

- 要启用 section-size 工具，请使用：

```
$ annocheck --section-size=name
```

  使用命名部分的 名称 替换 name。输出仅限于特定的部分。最后会生成一个累积结果。

**附加信息**

- 有关 section-size 工具的详情请参考 **--help** 命令行选项。

#### 17.2.4.4. 强化检查程序基础

默认启用强化检查程序。您可以使用 **--disable-hardened** 命令行选项禁用强化检查程序。

##### 17.2.4.4.1. 强化检查器选项

annocheck 程序检查以下选项：

- 使用 **-z now** linker 选项禁用 lazy 绑定。
- 该程序没有可执行内存的堆栈。
- GOT 表的重新定位仅限读取。
- 没有程序片段中的所有三个读、写和执行权限位集。
- 无法对可执行代码进行重新定位。
- 在运行时查找共享库的 runpath 信息只包括根于 /usr 的目录。
- 程序编译时启用了 annobin 备注。

- 该程序已启用 -fstack-protector-strong 选项编译。
- 该程序已使用 -D_FORTIFY_SOURCE=2 编译。
- 该程序已使用 -D_GLIBCXX_ASSERTIONS 编译。
- 该程序被编译为启用 -fexceptions。
- 该程序被编译为启用 -fstack-clash-protection。
- 该程序被编译为 -O2 或更高版本。
- 该程序没有以写入形式进行的重新定位。
- 动态可执行文件有一个动态片段。
- 共享库使用 -fPIC 或 -fPIE 编译。
- 动态可执行文件使用 -fPIE 编译并链接到 -pie。
- 如果可用,使用 -fcf-protection=full 选项。
- 如果可用,使用 -mbranch-protection 选项。
- 如果可用,使用 -mstackrealign 选项。

##### 17.2.4.4.2. 禁用强化检查器

下面的部分论述了如何禁用强化检查器。

**流程**

- 要在没有强化检查器的情况下扫描文件中的备注,请使用：

```
$ annocheck --enable-notes --disable-hardened file-name
```

使用 文件名 替换 file-name。

## 17.3. 删除冗余 ANNOBIN 备注

使用 annobin 会增加二进制文件的大小。要减少使用 annobin 编译的二进制文件的大小,您可以删除冗余 annobin 备注。要删除冗余 annobin 备注,使用 objcopy 程序,它是 binutils 软件包的一部分。

**流程**

- 要删除冗余 annobin 备注,请使用：

```
$ objcopy --merge-notes file-name
```

用文件名替换 file-name。

# *部分 IV. 附加主题*

# 第 18 章 编译器和开发工具中破坏兼容性的更改

## librtkaio 删除

在这个版本中, librtkaio 库已被删除。这个程序库为一些文件提供了高性能实时 I/O 访问权限,它们基于 Linux 内核 Asynchronous I/O 支持(KAIO)。

由于删除的结果：

- 使用 LD_PRELOAD 方法加载librtkaio 的应用程序会显示对缺少库的警告,而是加载librt 库并正确运行。
- 使用 LD_LIBRARY_PATH 方法加载librtkaio 的应用程序 加载librt 库并正常运行,没有任何警告。
- 使用 dlopen() 系统调用来访问librtkaio 的应用程序直接载入librt 库。

librtkaio 用户有以下选项：

- 使用上面描述的回退机制,而不对其应用程序进行任何更改。
- 将应用程序代码改为使用 librt 库, 它提供了一个兼容的 POSIX API。
- 将应用程序代码改为使用 libaio 库 , 该库提供了一个兼容的 API。

librt 和 libaio 可 在特定条件下提供功能和性能。

请注意, libaio 软件包的红帽兼容性级别为 2,而librtk 和删除了 librtkaio 级别 1。

如需了解更多详细信息,请参阅 https://fedoraproject.org/wiki/Changes/GLIBC223_librtkaio_removal

## 从中删除的 Sun RPC 和 NIS 接口 glibc

**glibc** 库不再为新应用程序提供 Sun RPC 和 NIS 接口。现在,这些接口仅适用于运行旧应用程序。开发人员必须更改其应用程序,才能使用 **libtirpc** 库,而不是使用 Sun RPC 和**libnsl2** 而不是 NIS。应用程序可从替换库中的 IPv6 支持中受益。

## 32 位 Xen 的 nosegneg 库已被删除

在以前的版本中, **glibc** i686 软件包包含一个替代的**glibc** 构建,它避免使用带有负偏移(**nosegneg**)的线程描述符寄存器。这个替代构建只用于 Xen Project hypervisor 的 32 位版本,但没有硬件虚拟化支持,从而可以降低完全半虚拟化的成本。现在,这些替代构建不再被使用,它们已被删除。

## make 新的 operator != 会对某些现有的 makefile 语法有不同的解释

在 GNU **make** 中添加了 **!=** shell 分配运算符,作为 **$(shell …)** 功能的替代来提高与 BSD makefile 的兼容性。因此,名称以声明标记结尾的变量,然后再分配（如 **variable!=value** ）现在被解释为 shell 分配。要恢复之前的行为,请在声明标记后添加一个空格,如 **variable! =value**。

有关运算符和功能之间的更多详情和区别,请查看 GNU **make** 手册。

## 用于 MPI 调试支持的 Valgrind 库已删除

由 **valgrind-openmpi** 软件包提供的Valgrind 的**libmpiwrap.so** wrapper 库已被删除。这个库启用了 Valgrind 使用 Message Passing Interface(MPI)调试程序。这个程序库专用于之前 Red Hat Enterprise Linux 版本中的 Open MPI 实现版本。

建议 **libmpiwrap.so** 用户从特定于其 MPI 实现和版本的上游源构建自己的版本。使用**LD_PRELOAD** 技术向 Valgrind 提供这些自定义构建的库。

## 从中移除开发标头和静态库 valgrind-devel

*在以前的版本中,用来包括用于开发自定义 valgrind 工具的开发文件 **valgrind-devel** 子软件包。在这个版本中删除了这些文件,因为它们没有保证的 API,且必须静态链接,且不被支持。**valgrind-devel** 软件包仍然包含 valgrind-aware 程序及标头文件（如 **valgrind.h**、**callgrind.h**、**drd.h**、**helgrind.h** 和 **memcheck.h** ）的开发文件,它们是稳定并被良好支持的。*

# 第 19 章 在 RHEL 8 上运行 RHEL 6 或 7 应用程序的选项

要在 Red Hat Enterprise Linux 8 上运行 Red Hat Enterprise Linux 6 或 7 应用程序,可以使用一组选项。系统管理员需要应用程序开发人员的详细指导。以下列表列出了红帽提供的选项、注意事项和资源。

**在具有匹配 RHEL 版本客户端操作系统的虚拟机中运行应用程序**

这个选项的资源成本很高,但环境与应用程序的要求紧密匹配,这个方法不需要很多额外考虑。这是当前推荐的选项。

**基于相应 RHEL 版本在容器中运行应用程序**

资源成本低于以前的情况,而配置要求则更严格。有关容器主机和客户机用户空间之间的关系的详情,请查看 Red Hat Enterprise Linux Container 兼容性列表。

**在 RHEL 8 上原生运行应用程序**

这个选项提供最低资源成本,但也提供最严格的要求。应用程序开发人员必须决定 RHEL 8 系统的正确配置。以下资源可帮助开发人员执行此任务:

- Red Hat Enterprise Linux 8:应用程序兼容性指南
- Red Hat Enterprise Linux 7:应用程序兼容性指南
- Red Hat Enterprise Linux 8.0 发行注记
- 使用 RHEL 8 时的注意事项

请注意,这个列表不是决定应用程序兼容性所需的完整的资源集合。这些只是已知不兼容更改列表以及红帽与兼容性相关的策略的起点。

另外, Kernel Application Binary Interface(kABI)是什么？知识中心的支持文章包含与内核和兼容性相关的信息。