Red Hat Enterprise Linux 9

在 RHEL 9 中开发 C 和 C++ 应用程序

在 Red Hat Enterprise Linux 9 中设置开发人员工作站并开发和调试 C 和 C++ 应用程序

Last Updated: 2024-07-09

Red Hat Enterprise Linux 9 在 RHEL 9 中开发 C 和 C++ 应用程序

在 Red Hat Enterprise Linux 9 中设置开发人员工作站并开发和调试 C 和 C++ 应用程序

法律通告

Copyright © 2024 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

使用 Red Hat Enterprise Linux 9 中提供的不同功能和工具来开发和调试 C 和 C++ 应用程序。

目录

对红帽文档提供反馈

我们感谢您对我们文档的反馈。让我们了解如何改进它。

通过 Jira 提交反馈（需要帐户）

1. 登录到 Jira 网站。
2. 点顶部导航栏中的 **Create**
3. 在 **Summary** 字段中输入描述性标题。
4. 在 **Description** 字段中输入对改进的建议。包括文档相关部分的链接。
5. 点对话框底部的 **Create**。

第 1 章 设置开发工作站

Red Hat Enterprise Linux 9 支持自定义应用程序的开发。要允许开发人员这样做，必须使用所需工具和实用程序设置系统。本章列出了用于开发的最常见用例和要安装的项目。

1.1. 先决条件

- 必须安装该系统，包括图形环境和订阅。

1.2. 启用调试和源存储库

Red Hat Enterprise Linux 的标准安装并不会启用 debug 和 source 软件仓库。这些仓库包含调试系统组件并测量其性能所需的信息。

步骤

- 启用源和调试信息频道：**$(uname -i)** 部分会自动替换为与您的系统构架匹配的值：

架构名称	Value
64 位 Intel 和 AMD	x86_64
64-bit ARM	aarch64
IBM POWER	ppc64le
64-bit IBM Z	s390x

1.3. 设置来管理应用程序版本

有效的版本控制对于所有多开发人员项目至关重要。Red Hat Enterprise Linux 由 Git（一个分布式版本控制系统）提供。

步骤

1. 安装 **git** 软件包：

```
# dnf install git
```

2. 可选：设置与您的 Git 提交关联的全名和电子邮件地址：

```
$ 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 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
```

使用使用命令替换 *command*，以启动选定的文本编辑器。

其他资源

- Git 和教程的 Linux man page:

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

  请注意，很多 Git 命令都有自己的 man page。例如，请参阅 *git-commit(1)*。

- *Git 用户手册* - Git 的 HTML 文档位于 **/usr/share/doc/git/user-manual.html**。
- Pro Git - *Pro Git* 文档的在线版本提供了 Git、其概念及其用法的详细描述。
- 参考 - Git 的在线版本的 Linux man page

1.4. 设置以使用 C 和 C++ 开发应用程序

Red Hat Enterprise Linux 包括用于创建 C 和 C++ 应用程序的工具。

先决条件

- 必须启用 debug 和 source 存储库。

步骤

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

```
# dnf group install "Development Tools"
```

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

```
# dnf install llvm-toolset
```

3. 可选：对于 Fortran 依赖项，请安装 GNU Fortran 编译器：

```
# dnf install gcc-gfortran
```

1.5. 设置调试应用程序

Red Hat Enterprise Linux 提供多个调试和检测工具，用于分析和解决内部应用程序行为。

先决条件

- 必须启用 debug 和 source 存储库。

步骤

1. 安装可用于调试的工具：

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

2. 安装 **dnf-utils** 软件包以使用 **debuginfo-install** 工具：

```
# dnf install dnf-utils
```

3. 运行 SystemTap 帮助程序脚本以设置环境。

```
# stap-prep
```

请注意，**stap-prep** 安装与当前*运行的*内核相关的软件包，这可能与实际安装的内核不同。为确保 **stap-prep** 安装了正确的 **kernel-debuginfo** 和 **kernel-headers** 软件包，请使用 **uname -r** 命令仔细检查当前的内核版本，如果需要，重启系统。

4. 确保 **SELinux** 策略允许相关应用程序正常运行，在需要进行调试的情况下也可以运行。如需更多信息，请参阅使用 SELinux。

1.6. 设置以测量应用程序的性能

Red Hat Enterprise Linux 包含多个应用程序，可帮助开发人员识别应用程序性能丢失的原因。

先决条件

- 必须启用 debug 和 source 存储库。

步骤

1. 安装用于性能测量的工具：

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

2. 运行 SystemTap 帮助程序脚本以设置环境。

```
# stap-prep
```

请注意，**stap-prep** 安装与当前*运行的*内核相关的软件包，这可能与实际安装的内核不同。为确保 **stap-prep** 安装了正确的 **kernel-debuginfo** 和 **kernel-headers** 软件包，请使用 **uname -r** 命令仔细检查当前的内核版本，如果需要，重启系统。

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

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

第 2 章 创建 C 或 C++ 应用程序

红帽提供多种使用 C 和 C++ 语言创建应用程序的工具。本书的这一部分列出了一些最常见的开发任务。

2.1. RHEL 9 中的 GCC

Red Hat Enterprise Linux 9 提供了 GCC 11 作为标准的编译器。

GCC 11 的默认语言标准设置为 C++17。这等同于明确使用命令行选项 **-std=gnu++17**。

后续语言标准（如 C++20 等）和这些语言标准的库功能仍被视为实验性。

其他资源

- 移植到 GCC 11
- 使用 GCC 11 将代码移植到 C++17

2.2. 使用 GCC 构建代码

了解源代码必须转换为可执行代码的情况。

2.2.1. 代码表单之间的关系

先决条件

- 了解编译和链接的概念

可能的代码形式

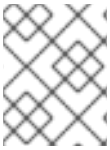
C 和 C++ 语言具有三种代码：

- 用 C 或 C++ 语言编写的 **源代码**，以纯文本文件形式呈现。
  文件通常会使用 **.c**, **.cc**, **.cpp**, **.h**, **.hpp**, **.i**, **.inc** 作为扩展名。有关支持的扩展及其解释的完整列表，请参阅 gcc 手册页：

  ```
  $ man gcc
  ```

- **对象代码（Object code）**，通过使用*编译器*对源代码*进行编译*来创建。这是一个中间形式。
  对象代码文件使用 **.o** 扩展。

- **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。共享对象（库）可执行文件使用 **.so** 文件名扩展名。



注意

还有用于静态链接的库归档文件。这是使用 **.a** 文件名扩展名的对象代码的变体。不建议使用静态链接。请参阅 第 2.3.2 节 “静态和动态链接”。

在 GCC 中处理代码形式

从源代码生成可执行代码分为两个步骤，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。这样，您可以使用一个单一的 **gcc** 命令用于任何必需的操作（编译和链接）。GCC 自动选择操作及其序列：

1. 源文件是编译到对象文件中。
2. 对象文件和库已链接（包括之前编译的源）。

可以运行 GCC 以便它只执行编译、只执行链接或在单个步骤中进行编译和链接。这由输入和请求的输出类型决定。

因为大型项目会需要一个构建系统，它通常会对每个操作独立运行 GCC，因此最好总是将编译和链接看做为两个不同的独立操作，即使 GCC 可以同时执行这两个操作。

2.2.2. 将源文件编译到对象代码

要从源文件而非可执行文件创建对象代码文件，必须指示 GCC 仅创建对象代码文件，作为其输出。此操作代表了更大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，其文件名反映了原始源代码文件： **source.c** 将生成 **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++ 标准库依赖项。

2.2.3. 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。要启用 C 和 C++ 应用的调试，您必须明确指示编译器创建它。

要启用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导致很难与原始源代码相关的执行代码：变量可能会被优化、循环展开、操作被合并到周围的操作中，等等。这会对调试有负面影响。为了改进调试体验，请考虑使用 **-Og** 选项设置优化。但是，更改优化等级会改变可执行代码，并可能会更改实际行为，包括删除一些错误。
- 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 GCC 使用调试信息和没有调试信息编译的代码。如果生成的两个二进制文件相同，则测试通过。此测试可确保可执行代码不受任何调试选项的影响，这会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。

其他资源

- 使用 GNU Compiler Collection(GCC)- 用于调试程序的选项
- 使用 GDB 进行调试 - Debugging Information in Separate Files
- GCC 手册页：

```
$ man gcc
```

2.2.4. GCC 的代码优化

可以将单个程序转换为多个计算机说明序列。如果您分配更多资源以在编译过程中分析代码，您可以实现更最佳的结果。

使用 GCC，您可以使用 **-O*level*** 选项设置优化级别。此选项接受一组值以代替 *level*。

级别	描述
0	优化编译速度 - 无代码优化（默认）。
1, 2, 3	优化以提高代码执行速度（数量越大，速度越快）。
s	优化文件大小。
fast	与 **3** 级设置相同，**fast** 会忽略严格的标准合规性检查，允许进行额外的优化。
g	优化调试体验。

对于发行版本构建，请使用优化选项 **-O2**。

在开发过程中，**-Og** 选项在某些情况下用于调试程序或库。因为有些程序错误清单只适用于特定的优化级别，所以使用发行版本优化级别测试程序或库。

GCC 提供大量选项来启用单个优化。如需更多信息，请参阅以下额外资源。

其他资源

- 使用 GNU Compiler Collection - 控制优化的选项
- GCC 的 Linux 手册页：

```
$ man gcc
```

2.2.5. 使用 GCC 的强化代码选项

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以防止经常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和库，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，建议使用以下选项列表：

```
$ 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** Position 独立可执行文件选项。
- 对于动态链接库，强制 **-fPIC** （独立代码）选项间接增加安全性。

开发选项

使用以下选项检测开发过程中的安全漏洞。这些选项与发行版本选项一起使用：

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

其他资源

- 防御代码指南
- 使用 GCC 进行内存错误检测 - 红帽开发人员博客文章

2.2.6. 连接代码以创建可执行文件

连接是构建 C 或 C++ 应用程序时的最后一步。将所有对象文件和库链接到可执行文件中。

先决条件

- 一个个或多个对象文件
- 必须在系统上安装 GCC

步骤

1. 更改到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中创建一个名为 ***executable-file*** 的可执行文件。

   要链接附加库，请在对象文件列表后添加所需选项。

注意

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

2.2.7. 例如：使用 GCC 构建一个 C 程序（在一个步骤中编译和链接）

此示例显示构建简单示例 C 程序的确切步骤。

在本例中，编译和链接代码在一个步骤中完成。

先决条件

- 您必须使用 GCC。

步骤

1. 创建一个目录 **hello-c** 并修改它：

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

2. 使用以下内容创建文件 **hello.c** ：

```
#include <stdio.h>

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

3. 使用 GCC 编译和链接代码：

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

这会编译代码，创建目标文件 **hello.o**，并从目标文件链接可执行文件 **helloworld**。

4. 运行生成的可执行文件：

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

2.2.8. 例如：使用 GCC 构建一个 C 程序（编译和连接在两个步骤中）

此示例显示构建简单示例 C 程序的确切步骤。

在本例中，编译和链接代码是两个独立的步骤。

先决条件

- 您必须使用 GCC。

步骤

1. 创建一个目录 **hello-c** 并修改它：

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

2. 使用以下内容创建文件 **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. 从对象文件中链接可执行文件 **helloworld**:

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

5. 运行生成的可执行文件：

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

2.2.9. 例如：使用 GCC 构建一个 C++ 程序（在一个步骤中编译和链接）

这个示例显示了构建示例最小 C++ 程序的确切步骤。

在本例中，编译和链接代码在一个步骤中完成。

先决条件

- 您必须了解 **gcc** 和 **g++** 之间的区别。

步骤

1. 创建一个目录 **hello-cpp** 并修改它：

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

2. 使用以下内容创建文件 **hello.cpp** ：

```
#include <iostream>

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

3. 使用 **g++** 编译和链接代码：

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

这会编译代码，创建目标文件 **hello.o**，并从目标文件链接可执行文件 **helloworld**。

4. 运行生成的可执行文件：

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

2.2.10. 例如：使用 GCC 构建一个 C++ 程序（编译和连接在两个步骤中）

这个示例显示了构建示例最小 C++ 程序的确切步骤。

在本例中，编译和链接代码是两个独立的步骤。

先决条件

- 您必须了解 **gcc** 和 **g++** 之间的区别。

步骤

1. 创建一个目录 **hello-cpp** 并修改它：

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

2. 使用以下内容创建文件 **hello.cpp** ：

```
#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. 从对象文件中链接可执行文件 **helloworld**:

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

5. 运行生成的可执行文件：

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

2.3. 将库与 GCC 搭配使用

了解在代码中使用库。

2.3.1. 库命名规则

用于库的一个特殊文件名称规则：例如称为 **foo** 的库一般会有文件 **lib*foo*.so** 或 **lib*foo*.a**。通过 GCC 的链接输入选项（而非输出选项）自动理解此约定：

- 当对库进行链接时，库可以通过它的名称 **foo** 以及参数 **-l** 指定（**-l*foo***）：

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

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

2.3.2. 静态和动态链接

开发人员可以选择使用完整编译语言构建应用时使用静态或动态链接。了解静态和动态链接之间的区别很重要，特别是在 Red Hat Enterprise Linux 上使用 C 和 C++ 语言的上下文中。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。动态连接会将这些库保留为单独的文件。

可使用多种方式比较动态和静态链接：

资源使用

静态连接会导致包含更多代码的大型可执行文件。这个来自库的额外代码不能在系统上的多个程序间共享，在运行时增加文件系统的使用情况和内存使用情况。运行相同静态链接程序的多个进程仍将共享代码。
另一方面，静态应用程序需要较少的运行时重新定位，从而缩短启动时间，并需要较少的专用 RSS（resident set size）内存。因为使用独立于位置的代码(PIC)带来的开销，静态链路生成的代码比动态连接效率更高。

安全性

可动态链接库（提供 ABI 兼容性），无需更改可执行文件，具体取决于这些库。这对于由红帽提供的、作为 Red Hat Enterprise Linux 的一部分的库尤为重要，因为红帽会提供安全更新。强烈建议不要对任何此类库进行静态链接。

兼容性

静态连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。使用静态链路时，这个依赖关系会变得不灵活，因此向前和向后兼容都会丢失。静态连接保证仅在构建可执行文件的系统中工作。

警告

从 GNU C 库链接静态库（**glibc**）的应用程序仍然需要在系统中存在 **glibc** 作为动态库。另外，在应用程序运行时提供 **glibc** 的动态库变体必须是与应用程序链接时所用的相同版本。因此，静态连接可以保证仅在构建可执行文件的系统中工作。

支持覆盖范围

红帽提供的大多数静态库都在 *CodeReady Linux Builder* 频道中，它们不受红帽的支持。

功能

有些库，例如 GNU C 库（**glibc**）在静态链接时提供的功能会减少。

例如，当静态链接时，**glibc** 不支持线程以及在同一个程序中的任何形式的 **dlopen()** 调用。

因为存在这里列出的缺陷，应尽可能避免使用静态链接，特别是整个应用程序以及 **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：

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

2.3.3. 链接时间优化

链接时间优化(LTO)可让编译器通过在链接时使用其中间表示，在程序的所有转换单元中执行各种优化。因此，您的可执行文件和库会较小并更快地运行。另外，您可以使用 LTO 在编译时分析软件包源代码，从而改进了潜在的编码错误的各种 GCC 诊断。

已知问题

- 违反一个定义规则(ODR)会产生 **-Wodr** 警告
  ODR 违规会导致未定义的行为产生一个 **-Wodr** 警告。这通常指向您的程序中的一个错误。默认启用 **-Wodr** 警告。
- LTO 会导致内存消耗增加
  当程序处理翻译单元由以下组成时，编译器会消耗更多内存。在内存有限的系统中，在构建程序时禁用 LTO 或者降低并行级别。
- GCC 会删除以前未使用的功能
  GCC 可能会删除它认为未使用的功能，因为编译器无法识别 asm() 语句引用哪些符号。因此，可能会出现编译错误。要防止这种情况，请在您的程序中使用的符号中添加 **__attribute__((used))**。
- 使用 **-fPIC** 选项编译会导致错误
  由于 GCC 不会解析 asm() 语句的内容，因此使用 **-fPIC** 命令行选项编译代码可能会导致错误。要防止这种情况，在编译您的翻译单元时使用 **-fno-lto** 选项。
  请参阅 LTO FAQ - Symbol usage from assembly 语言 。
- 使用 **.symver** 指令的符号进行版本与 LTO 不兼容
  在 asm()语句中使用 **.symver** 指令实现符号版本控制与 LTO 不兼容。不过，可以使用 **symver** 属性实施符号版本控制。例如：

  __attribute__ ((_symver_ ("<*symbol*>@VERS_1"))) void <*symbol*>_v1 (void) { }

其他资源

- GCC Manual - 功能属性
- GCC Wiki - Link Time Optimization

2.3.4. 将库与 GCC 搭配使用

库是可以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

使用库的编译代码

标头文件描述了库的接口：库中可用的函数和变量。编译代码需要来自标头文件的信息。

通常，库的标头文件将放在与应用程序代码不同的目录中。要告知 GCC 哪个标头文件，请使用 **-I** 选项：

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

使用到标头文件目录的实际路径替换 *include_path*。

可以多次使用 **-I** 选项来添加包含标头文件的多个目录。查找标头文件时，会按照它们在 **-I** 选项中的顺序搜索这些目录。

使用库的链接代码

在连接可执行文件时，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形式：

- 静态库作为存档文件提供。它们包含一组对象文件。归档文件名为 **.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。共享对象具有文件名扩展名 **.so**。

要告知 GCC 一个库的归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 选项可以多次使用来添加多个目录。查找库时，会按照其 **-L** 选项顺序搜索这些目录。

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，在使用链接库的 **-l** 选项前，使用 **-L** 选项指定库目录。

在一个步骤中使用库编译和链接代码

当允许代码在一个 **gcc** 命令中编译和链接时，请一次性使用上述两个情况的选项。

其他资源

- 使用 GNU Compiler Collection(GCC) - 目录查询的选项
- 使用 GNU Compiler Collection(GCC)- 链接的选项

2.3.5. 在 GCC 中使用静态库

静态库作为包含对象文件的存档提供。链接后，它们将成为生成的可执行文件的一部分。

注意

出于安全原因，红帽不建议使用静态链接。请参阅 第 2.3.2 节 “静态和动态链接”。仅在必要时使用静态链接，特别是红帽提供的库。

先决条件

- 必须在您的系统中安装 GCC。
- 您必须了解静态和动态链接。
- 您有一组源或对象文件组成一个有效程序，需要一些静态库 **foo** 且没有其他库。
- **foo** 库作为一个文件 **libfoo.a** 提供，且没有为动态链接提供文件 **libfoo.so**。

注意

作为 Red Hat Enterprise Linux 一部分的大多数库都只支持动态链接。以下步骤只适用于 *没有* 为动态链接启用的库。

步骤

要从源和目标文件链接程序，添加一个静态链接的库 **foo**，它会是一个 **libfoo.a** 文件：

1. 更改到包含您的代码的目录。
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** 的目录替换 *library_path*。

4. 要在以后运行该程序，只需：

```
$ ./program
```

警告

与静态链路相关的 **-static** GCC 选项，用于禁止所有动态链接。应该使用 **-Wl,-Bstatic** 和 **-Wl,-Bdynamic** 选项来更精确地控制链接器行为。请参阅 第 2.3.7 节 “在 GCC 中使用静态和动态库”。

2.3.6. 使用 GCC 的动态库

动态库可作为独立的可执行文件提供，在链接时间和运行时需要。它们独立于应用程序的可执行文件。

先决条件

- 必须在系统中安装 GCC。

- 一组源或对象文件组成一个有效程序，需要一些动态库 **foo**，没有其他库。
- **foo** 库必须作为一个文件 *libfoo.so* 可用。

将程序与动态库连接

要根据动态库 **foo** 链接程序：

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

当某个程序与动态库相关联时，生成的程序必须始终加载库。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 Executable 文件中的 rpath 值

rpath 是一个特殊值，保存在可执行文件时作为可执行文件的一部分保存。之后，当从可执行文件加载程序时，运行时链接器将使用 **rpath** 值来定位库文件。

当使用 GCC 链接时，将路径 *library_path* 存储为 **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。

重要

不要在 **-Wl,-rpath=** 选项的逗号后添加一个空格。

稍后运行程序：

```
$ ./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**，则运行时链接器将使用 **LD_LIBRARY_PATH** 环境变量。必须为每个程序更改此变量的值。这个值应该代表共享库对象所在的路径。

要在没有设置 **rpath** 的情况下运行安装程序，库存在于路径 *library_path* 中：

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

不使用 **rpath** 值可提供一定的灵活性，但每次程序运行时都需要设置 **LD_LIBRARY_PATH** 变量。

将库放在默认目录中

运行时链路器配置将多个目录指定为动态库文件的默认位置。要使用这个默认行为，请将您的库复制到适当的目录中。

动态链路程序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：

- 动态链路器的 Linux 手册页：
  -

```
$ man ld.so
```

- **/etc/ld.so.conf** 配置文件的内容：

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

- 报告由动态链路器识别的库而无需额外的配置，其中包括目录：

```
$ ldconfig -v
```

2.3.7. 在 GCC 中使用静态和动态库

有时需要静态和一些库来动态链接一些库。这种情况给企业带来一些挑战。

先决条件

- 了解静态和动态链接

简介

GCC 同时识别动态和静态库。当遇到 **-l*foo*** 选项时，gcc 将首先尝试查找包含 foo 库的动态链接版本的共享对象（ **.so** 文件），然后查找包含该库静态版本的存档文件(**.a**)。因此，以下情况可能会导致此搜索的结果：

- 仅找到共享对象，并动态 gcc 链接它。
- 只找到归档，并以静态方式使用 gcc 链接。
- 可以找到共享对象和存档，默认情况下，gcc 选择对共享对象的动态链接。
- 未找到共享对象或存档，并且链接失败。

由于这些规则，选择要链接库的静态或动态版本的最佳方法是，只有 gcc 找到的版本。在指定 **-L*path*** 选项时，可以通过使用或不使用包含库版本的目录来进行一定的控制。

另外，因为动态链接是默认设置的，所以当存在这两个版本的库时，需要明确指定链接的唯一情形是静态链接。有两种可能的解决方法：

- 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 linker

通过文件指定静态库

通常，gcc 指示针对带有 **-l*foo*** 选项的 foo 库进行链接。但是，可以指定包含库的文件 **lib*foo*.a** 的完整路径：

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

通过文件扩展名 **.a**，gcc 将了解到这是与程序链接的库。但是，指定库文件的完整路径是一个不太灵活的方法。

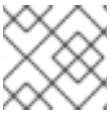
使用 -Wl 选项

gcc 选项 **-Wl** 是用于将选项传递给底层链路器的特殊选项。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 提供了选项 **-Bstatic** 和 **-Bdynamic** 以指定此选项后的库是否应静态链接或动态链接。将 **-Bstatic** 和库传递给 linker 后，必须手动恢复默认的动态链接行为，才能将以下库与 **-Bdynamic** 选项动态链接。

要链接程序，静态链接库 **first**(**libfirst.a**)，动态链接库 **second**(**libsecond.so**)：

```
$ gcc ... -Wl,-Bstatic -l first -Wl,-Bdynamic -l second ...
```



注意

gcc 可以配置为使用默认 ld 以外的链接器。

其他资源

- 使用 GNU Compiler Collection (GCC) — 3.14 Options for Linking
- binutils 2.27 文档 - 2.1 命令行选项文档

2.4. 使用 GCC 创建库

了解创建库的步骤，以及 Linux 操作系统用于库的必要概念。

2.4.1. 库命名规则

用于库的一个特殊文件名称规则：例如称为 **foo** 的库一般会有文件 **lib*foo*.so** 或 **lib*foo*.a**。通过 GCC 的链接输入选项（而非输出选项）自动理解此约定：

- 当对库进行链接时，库可以通过它的名称 **foo** 以及参数 **-l** 指定（**-l*foo***）：

  ```
  $ gcc ... -l foo ...
  ```

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

2.4.2. soname 机制

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

- 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。

问题介绍

动态加载的库（共享对象）作为独立的可执行文件存在。这样，可以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

- 需要同一库的多个版本
- 代表每个版本的 ABI 兼容性

soname 机制

为了解决这个问题，Linux 使用名为 soname 的机制。

foo 库版本 *X.Y* 与在版本号中的 *X* 值相同的其它版本是 ABI 兼容。不会影响兼容性的次要更改会增加数字 *Y*。会影响兼容性的主要变化会增加数字 *X*。

实际的 **foo** 库版本 *X.Y* 作为文件 **libfoo.so.*x.y*** 存在。在库文件中，使用值 **libfoo.so.x** 记录 soname 来指示兼容性。

构建应用程序时，链接程序通过搜索文件 **libfoo.so** 来查找库。必须存在带有此名称的符号链接，指向实际库文件。然后，链接器会从库文件读取 soname，并将其记录到应用程序可执行文件。最后，链路器创建使用 soname（而不是文件名）声明对库的依赖项的应用。

运行时动态链路器会在运行之前链接应用程序时，它会从应用程序的可执行文件读取 soname。这个 soname 是 **libfoo.so.*x***。必须存在带有此名称的符号链接，指向实际库文件。无论某个版本的 *Y* 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，一些库会使用其名称对它们的版本进行编码。

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*。

2.4.3. 使用 GCC 创建动态库

动态链接的库（共享对象）允许：

- 通过代码重复利用的资源保留
- 通过更新库代码来提高安全性

按照以下步骤从源构建并安装动态库。

先决条件

- 您必须了解 soname 机制。
- 必须在系统中安装 GCC。
- 您必须具有库的源代码。

步骤

1. 进入包含库源的目录。

2. 使用独立代码选项 **-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 ...
```

使用的主版本号为 X 和次版本号 Y。

4. 将 ***libfoo.so.x.y*** 文件复制到一个适当的位置，该系统的动态链路程序可以找到它。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 权限来处理此目录中的文件。

5. 为 soname 机制创建符号链接结构：

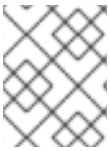
```
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

其他资源

- Linux 文档项目 - 程序库 HOWTO - 3.共享库

2.4.4. 使用 GCC 和 ar 创建静态库

通过将对象文件转换为特殊归档文件，可以为静态链接创建库。



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 2.3.2 节 “静态和动态链接”。

先决条件

- 在系统中必须安装 GCC 和 binutils。
- 您必须了解静态和动态链接。
- 提供了作为库共享的功能的源文件。

步骤

1. 使用 GCC 创建中间对象文件。

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

如果需要，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 ...
```

创建 **libfoo.a** 文件。

3. 使用 **nm** 命令检查生成的存档：

```
$ nm libfoo.a
```

4. 将静态库文件复制到适当的目录中。

5. 当链接到库时，GCC 会自动识别库为静态链接的 **.a** 文件名扩展。

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

其他资源

- *ar(1)* 的 Linux man page:

```
$ man ar
```

2.5. 使用 MAKE 管理更多代码

GNU make 程序（通常缩写为 **make**）是一个控制从源文件生成可执行文件的工具。**make** 自动确定复杂程序的哪个部分已更改，需要重新编译。**make** 使用名为 Makefile 的配置文件来控制构建程序的方式。

2.5.1. GNU make 和 Makefile 概述

要从特定项目的源文件创建可用格式（通常是可执行文件），请执行几个必要的步骤。记录操作及其序列以便以后重复它们。

Red Hat Enterprise Linux 包含 GNU **make**，它是一个专为此目的而设计的构建系统。

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*，用于描述一种使用特定操作（*recipe* ）满足特定条件（*目标*）的方法。规则可以分层依赖于另一规则。

在不带任何选项的情况下运行 **make**，它会在当前目录中查找 Makefile，并尝试访问默认目标。实际 Makefile 文件名可以是 **Makefile**、**makefile** 和 **GNUmakefile** 之一。默认目标从 Makefile 内容确定。

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。目标指定规则执行时的输出是什么。配方（recipes）的行必须以 TAB 字符开头。

通常，Makefile 包含编译源文件的规则、用于链接生成的对象文件的规则，以及用作层次结构顶部的入口点的目标。

考虑以下 **Makefile**，用于构建由一个文件 **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，在达到目标 **all**，需要文件 **hello**。若要获取 **hello**，一个需要 **hello.o** （由 **gcc**链接），它通过 **hello.c** 创建（由 **gcc** 编译）。

目标 **all** 是默认目标，因为它是没有以句点(.)开头的第一个目标。当当前目录包含此 **Makefile** 时，不使用任何参数运行 **make** 与运行 **make all** 是完全相同的。

典型的 makefile

更为典型的 Makefile 使用变量对步骤进行规范化，并添加目标"clean" - 删除除源文件外的所有内容。

```
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)
```

向此类 Makefile 添加更多源文件时，只需要将它们添加到定义 SOURCE 变量所在的行中。

其他资源

- GNU make：简介 - 2 Makefile 简介

2.5.2. 例如：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 的概念并 **make**。

步骤

1. 创建 **hellomake** 目录并改为此目录：

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

2. 创建一个包含以下内容的 **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：

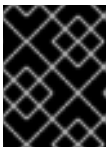
```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

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

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

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



重要

Makefile 方法行必须以 tab 字符开头！在文档中复制以上文本时，剪切和粘贴过程可能会粘贴空格而不是制表符（tab）。如果发生这种情况，请手动纠正问题。

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** 以删除创建的文件：

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

2.5.3. make 文档

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 man page 和信息页面：

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

在线文档

- GNU Make Manual 由自由软件基金会托管

第 3 章 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进行调试的最常见技术。

3.1. 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。以下小节介绍了如何获取此信息。

3.1.1. 调试信息

在调试任何可执行代码时，有两种信息允许工具，并通过扩展程序员的信息来理解二进制代码：

- 源代码文本
- 源代码文本如何与二进制代码关联

此类信息称为调试信息。

Red Hat Enterprise Linux 将 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，使用 DWARF 格式来保存调试信息。

要显示存储在 ELF 文件中的 DWARF 信息，请运行 **readelf -w *file*** 命令。

重要

STABS 是一种较旧的、功能较多的格式，有时与 UNIX 一起使用。红帽不建议使用它。GCC 和 GDB 只"尽力而为"的方式提供 STABS 生产和消耗。其他一些工具，如 Valgrind 和 **elfutils** 无法用于 STABS。

其他资源

- DWARF 调试标准

3.1.2. 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。要启用 C 和 C++ 应用的调试，您必须明确指示编译器创建它。

要启用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导致很难与原始源代码相关的执行代码：变量可能会被优化、循环展开、操作被合并到周围的操作中，等等。这会对调试有负面影响。为了改进调试体验，请考虑使用 **-Og** 选项设置优化。但是，更改优化等级会改变可执行代码，并可能会更改实际行为，包括删除一些错误。
- 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 GCC 使用调试信息和没有调试信息编译的代码。如果生成的两个二进制文件相同，则测试通过。此测试可确保可执行代码不受任何调试选项的影响，这会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。

其他资源

- 使用 GNU Compiler Collection(GCC)- 用于调试程序的选项
- 使用 GDB 进行调试 - Debugging Information in Separate Files
- GCC 手册页：

```
$ man gcc
```

3.1.3. debuginfo 和 debugsource 软件包

debuginfo 和 **debugsource** 软件包包含用于程序和库的调试信息和调试源代码。对于在 Red Hat Enterprise Linux 软件仓库的软件包中安装的应用程序和库，您可以从附加频道获得独立的 **debuginfo** 和 **debugsource** 软件包。

调试信息软件包类型

有两种类型的软件包可用于调试：

debuginfo 软件包

debuginfo 软件包提供必要的调试信息，以便为二进制代码功能提供人类可读的名称。这些软件包包含 **.debug** 文件，其中包含 DWARF 调试信息。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。安装相应的 **debuginfo** 和 **debugsource** 软件包后，GDB 或 LLDB 等调试器可以与源代码的二进制代码执行相关。源代码文件安装到 **/usr/src/debug** 目录中。

3.1.4. 使用 GDB 获取应用程序或库的 debuginfo 软件包

调试代码需要调试信息。对于从软件包安装的代码，GNU Debugger(GDB)会自动识别缺少的调试信息，解析软件包名称，并提供了有关如何获取软件包的建议。

先决条件

- 必须在系统上安装您要调试的应用程序或库。
- 在系统中必须安装 GDB 和 **debuginfo-install** 工具。详情请参阅设置调试应用程序。
- 必须在系统上配置和启用提供 **debuginfo** 和 **debugsource** 软件包的存储库。详情请参阅 启用调试和源存储库。

步骤

1. 启动附加到您要调试的应用程序或库的 GDB。GDB 会自动识别缺少的调试信息并建议一个命令运行。

```
$ 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：键入 **q**，使用 **Enter** 进行确认。

```
(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了更改摘要，要求确认，一旦被您确认后会下载和安装所有需要的文件。

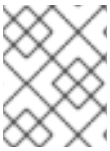
4. 如果 GDB 无法建议 **debuginfo** 软件包，请按照 手动应用程序或库获取 debuginfo 软件包中所述的步骤操作。

其他资源

- 如何为 RHEL 系统下载或安装 debuginfo 软件包？ - 红帽知识库解决方案

3.1.5. 手动获取应用程序或库的 debuginfo 软件包

您可以通过查找可执行文件来确定您需要安装哪些 **debuginfo** 软件包，然后查找安装它的软件包。



注意

红帽建议您使用 GDB 来决定要安装的软件包。只有在 GDB 无法建议安装软件包时，才使用这个手动过程。

先决条件

- 必须在系统中安装应用程序或库。
- 应用程序或库是从软件包安装的。
- 系统中必须有 **debuginfo-install** 工具。
- 必须在系统中配置和启用提供 **debuginfo** 软件包的频道。

步骤

1. 查找应用程序或库的可执行文件。
   a. 使用 **which** 命令来查找应用文件。

```
$ 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，请选择库在其文件名称中具有相同额外数字的结果，如错误消息中所述。如果有疑问，请尝试遵循库文件名不包含额外数字的结果。

注意

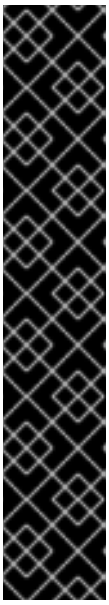
locate 命令由 **mlocate** 软件包提供。要安装它并启用其使用：

```
# dnf install mlocate
# updatedb
```

2. 搜索提供该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
```

输出提供了安装软件包（*name*:*epoch*-*version*.*release*.*architecture* 的形式）的详情。



重要

如果此步骤没有生成任何结果，则无法确定提供该二进制文件的软件包。有几个可能的情况：

- 文件从*当前*配置中包管理工具不知道的包安装。
- 该文件使用本地下载并手动安装的软件包进行安装。在这种情况下，无法自动决定合适的 **debuginfo** 软件包。
- 您的软件包管理工具配置错误。
- 该文件没有从任何软件包安装。在这种情况下，不存在相关的 **debuginfo** 软件包。

因为后续步骤依赖于这种情况，所以您必须解决这种情况或中止这个过程。描述确切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，使用在上一步中确定的软件包名称和其他详情：

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

其他资源

- 如何为 RHEL 系统下载或安装 debuginfo 软件包？ - 知识库文章

3.2. 使用 GDB 检查应用程序内部状态

若要查找应用无法正常工作的原因，请使用调试器控制其执行并检查其内部状态。本节论述了如何将 GNU Debugger(GDB)用于此任务。

3.2.1. 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。

GDB 功能

单个 GDB 会话可以调试以下类型的程序：

- 多线程和 fork 程序
- 一次多个程序
- 远程机器上的程序或带有 **gdbserver** 工具的容器中的程序通过 TCP/IP 网络连接

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，您必须安装其 debuginfo 软件包。

3.2.2. 将 GDB 附加到进程

要检查进程，GDB *必须附加到*进程中。

先决条件

- 在系统中必须安装 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 设置为开始执行程序。在使用 **run** 命令开始执行进程前，您可以设置断点和 **gdb** 环境。

将 GDB 附加到已经运行的进程

将 GDB 附加到已作为进程运行的程序中：

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：

   ```
   $ gdb -p pid
   ```

   使用 **ps** 输出中的实际进程 ID 编号替换 *pid*。

将已在运行的 GDB 附加到已经运行的进程

将已在运行的 GDB 附加到已经运行的程序中：

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

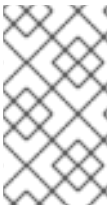
```
(gdb) shell ps -C program -o pid h
 pid
```

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 编号替换 *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
```

其他资源

- 使用 GDB 进行调试 - 2.1 Invoking GDB
- 使用 GDB 进行调试 - 4.7 调试 Already-running Process

3.2.3. 使用 GDB 检查程序代码

将 GDB 调试器附加到程序后，您可以使用一些命令来控制程序的执行。

先决条件

- 您必须具有所需的调试信息：
  - 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 必须附加到程序中才能被调试

GDB 命令逐步完成代码

r (run)

开始执行程序。如果使用任何参数执行 **run**，这些参数会像正常启动程序一样将这些参数传递给可执行文件。在设置断点后用户通常会发出此命令。

start

开始执行程序，但在程序的主函数的开头停止。如果使用任何参数执行 **start**，这些参数将传送到可执行文件，就如程序启动正常一样。

c (continue)

继续从当前状态执行程序。该程序的执行将继续进行，直到以下其中之一变为 true：

- 已有一个断点。
- 满足指定的条件。

- 程序收到信号。
- 发生错误。
- 程序终止。

n (next)

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。该程序的执行将继续进行，直到以下其中之一变为 true：

- 已有一个断点。
- 满足指定的条件。
- 程序收到信号。
- 发生错误。
- 程序终止。

s (step)

step 命令还会在当前源文件中的每个后续代码行下停止执行。但是，如果执行目前在包含 **函数调用** 的源行停止，GDB 会在输入函数调用（而不是执行）后停止执行。

until *location*

继续执行，直到达到 *location* 选项指定的代码位置。

fini (finish)

恢复执行程序并在执行从功能返回时停止执行。该程序的执行将继续进行，直到以下其中之一变为 true：

- 已有一个断点。
- 满足指定的条件。
- 程序收到信号。
- 发生错误。
- 程序终止。

q (quit)

终止执行并退出 GDB。

其他资源

- 使用 GDB 进行调试 – 启动您的程序
- 使用 GDB 进行调试 – 继续并逐步执行

3.2.4. 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 提供多个命令，可用于检查内部变量。以下是这些命令中最有用的命令：

p (print)

显示所给定参数的值。通常，参数是任何复杂性的变量名称，从简单的单一值到结构。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函数，或者在正在测试的程序中定义的函数。
可以使用 *pretty-printer* Python 或 Guile 脚本对 GDB 进行扩展，以使用 **print** 命令自定义显示数据结构（如类、结构）等。

bt (backtrace)

显示用于到达当前执行点的功能调用链，或者使用直到执行终止前所使用的功能链。这在调查严重错误（如分段错误）时非常有用，并带有严重原因。
在 **backtrace** 命令中添加 **full** 选项也会显示本地变量。

可以使用 *帧过滤* Python 脚本扩展 GDB，以使用 **bt** 和 **info** 框架命令对显示的数据进行自定义显示。术语 *帧（frame）* 指的是与单个功能调用关联的数据。

info

info 命令是提供有关各种项目的通用命令。它取指定要描述的项目的选项。

- **info args** 命令显示当前选择帧的功能调用选项。
- **info locals** 命令在当前选定的框中显示本地变量。

如需可能的项目列表，请在 GDB 会话中运行命令 **help info** ：

```
(gdb) help info
```

l (list)

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可用。虽然不是严格显示内部状态的命令，但 **list** 有助于用户了解执行程序在下一步中将发生对内部状态的更改。

其他资源

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

3.2.5. 使用 GDB 断点在定义的代码位置停止执行

通常，只调查一小部分代码。断点（breakpoints）是用来告诉 GDB 在代码中某个特定位置停止执行程序的标记。断点与源代码行最常关联。在这种情况下，放置断点需要指定源文件和行号。

- **要放置断点** ：
  - 指定源代码 *file* 的名称以及在该文件中的 *line* ：

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

  - 如果 *file* 不存在，则使用当前执行点的源文件的名称：

    ```
    (gdb) br line
    ```

  - 或者，使用函数名称将断点放在其开始上：

```
(gdb) br function_name
```

- 在任务进行一定迭代后，程序可能会遇到错误。要指定额外的 **condition** 停止执行：

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

  使用 C 或 C++ 语言条件替换 *condition*。*file* 和 *line* 如以上是相同的。

- 检查所有断点和监视点的状态：

  ```
  (gdb) info br
  ```

- 使用 **info br** 输出中的*数字*来**删除**断点：

  ```
  (gdb) delete number
  ```

- **删除**给定位置的断点：

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

其他资源

- 使用 GDB 进行调试 - 断点、观察点和捕获点

3.2.6. 使用 GDB 观察点停止对数据访问和更改执行

在很多情况下，在某些数据更改或访问前，让程序执行得很好。以下示例是最常见的用例。

先决条件

- 了解 GDB

在 GDB 中使用监视点

Watchpoints 是用来告诉 **GDB** 停止执行某个程序的标记。Watchpoints 与数据相关联：放置监视点需要指定一个表达式来描述变量、多个变量或内存地址。

- 为数据 **change** (写) **放置**一个观察点：

  ```
  (gdb) watch expression
  ```

  使用描述您要监视的表达式替换 *expression*。对于变量，*expression* 等于变量的名称。

- 为数据 **access** (读) **放置**一个观察点：

  ```
  (gdb) rwatch expression
  ```

- 要针对 **任何** 数据访问 **放置** 监视点（读取和写入）：

  ```
  (gdb) awatch expression
  ```

- 检查所有观察点和断点的状态：

```
(gdb) info br
```

- **删除**一个监视点：

```
(gdb) delete num
```

将 ***num*** 选项替换为 **info br** 命令报告的编号。

其他资源

- 使用 GDB 调试 - 设置 Watchpoints

3.2.7. 使用 GDB 调试 fork 或线程程序

有些程序使用分叉或线程来实现并行代码执行。调试多个同时执行路径需要特殊考虑。

先决条件

- 您必须了解进程分叉和线程的概念。

使用 GDB 调试 fork 程序

当程序（**父**）创建本身的独立副本（**子**）时，分叉是一个状况。使用以下设置和命令影响 GDB 在进行分叉时的作用：

- **follow-fork-mode** 设置控制 GDB 在分叉后是否遵循父项或子项。

  设置 follow-fork-mode 父项

  在分叉后，调试父进程。这是默认值。

  设置 follow-fork-mode 子项

  在分叉后，调试子进程。

  显示后续模式

  显示 **follow-fork-mode** 的当前设置。

- **set detach-on-fork** 设置控制 GDB 是否控制其他进程（未跟随）进程，还是保留它继续运行。

  设置 detach-on-fork on

  未遵循的进程（取决于 **follow-fork-mode** 值 ）将独立分离并运行。这是默认值。

  set detach-on-fork off

  GDB 控制这两个进程。其后的进程（取决于 **follow-fork-mode** 的值）会正常进行调试，而另一个被暂停。

  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 能够单独调试单个线程，并单独操作并检查它们。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。在打开该功能后，GDB 已准备好进行线程调试。

GDB 使用 *当前线程* 的概念。默认情况下，命令仅应用到当前的线程。

info 线程

显示带有相应 **id** 和 **gid** 的线程列表，代表当前的线程。

线程 *ID*

将指定 **id** 的线程设置为当前的线程。

线程应用 *ids command*

将 ***command*** 命令应用到 ***ids*** 列出的所有线程。 ***ids*** 选项是以空格分隔的线程 ID 列表。特殊值 **all** 将命令应用到所有线程。

break *location* thread *id* if *condition*

只对线程号 ***id*** 在带有特定***条件***的特定***位置***设置一个断点。

watch *expression* thread *id*

仅为线程编号 ID ***id*** 设置由 ***expression*** 定义的观察点。

command&

执行 **command** 命令并立即返回到 gdb 提示符 **(gdb)**，然后在后台继续执行任何代码。

interrupt

在后台停止执行。

其他资源

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

3.3. 记录应用程序交互

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以有足够的了解应用程序的行为，而无需调试实际的应用程序代码。另外，分析应用程序的交互可帮助识别错误清单的条件。

3.3.1. 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 工具主要启用由应用程序使用的系统调用（内核功能）。

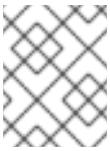
- **strace** 工具可以提供关于调用的详细输出，因为 **strace** 会解释参数以及了解底层内核代码的结果。将数字转换为相应的常量名称，位组合标志扩展至标志列表，指向字符数组的指针以提供实际字符串等。可能缺少对更多最新内核功能的支持。
- 您可以过滤 traced 调用来减少捕获的数据量。
- 除了设置日志过滤器外，使用 **strace** 不需要任何特定的设置。
- 使用 **strace** 跟踪应用程序代码会导致应用程序的执行速度显著慢。因此， **strace** 不适用于许多生产部署。作为替代方案，请考虑使用 **ltrace** 或 SystemTap。
- Red Hat Developer Toolset 中提供的 **strace** 版本也可以执行系统调用修改。此功能可用于调试。

ltrace

ltrace 工具允许将应用程序的用户空间调用记录到共享对象（动态库）。

- **ltrace** 工具允许对任何库进行追踪调用。

- 您可以过滤 traced 调用来减少捕获的数据量。
- 除了设置日志过滤器外，使用 **ltrace** 不需要任何特定的设置。
- **ltrace** 工具非常轻便且快速，提供 **strace** 的替代选择：可以跟踪库中相应的接口，如 **glibc**，使用 **ltrace** 而不是使用 **strace** 跟踪内核功能。
- 因为 **ltrace** 不会象 **strace** 一样处理一组已知的调用，所以不会试图解释传递给库函数的值。**ltrace** 输出仅包含原始数字和指针。**ltrace** 输出的解释需要咨询输出中存在库的实际接口声明。



注意

在 Red Hat Enterprise Linux 9 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提供的可执行文件。

SystemTap

SystemTap 是一个检测平台，用于探测在 Linux 系统上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** 进行比较，使用脚本进行日志处理意味着在初始设置阶段有更多工作。但是，脚本功能除生成日志外，扩展 SystemTap 的有用性。
- SystemTap 通过创建和插入内核模块来工作。SystemTap 的使用效率更高，且不会自行创建系统或应用程序的执行速度。
- SystemTap 附带一组用法示例。

GDB

GNU Debugger(GDB)主要用于调试，而不是处理日志记录。但是，它的某些功能在应用程序交互是值得关注的主要活动的情况中发挥用处。

- 使用 GDB，可以方便地将交互事件的捕获与后续执行路径的立即调试。
- 在其他工具的初始识别问题后，GDBDB 最适合分析响应不常或分散事件。在任何带有频繁事件的情况中使用 GDB 变得效率不低甚至不可能。

其他资源

- SystemTap 入门
- Red Hat Developer Toolset 用户指南

3.3.2. 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

- 必须在系统中安装了 **strace**。

步骤

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

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

   - 如果程序已在运行，找到其进程 ID(*pid*)并将 **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 **-e trace=*call*** 选项。如果没有使用，**strace** 将显示所有系统调用类型。如需更多信息，请参阅 *strace(1)* 手册页。
   - 如果您不想跟踪任何 fork 进程或线程，请退出 **-f** 选项。
3. **strace** 工具显示由应用发出的系统调用及其详情。
   在大多数情况下，应用程序及其库会立即出现大量调用和 **strace** 输出，如果没有设置系统调用的过滤器。
4. 当程序退出时，**strace** 工具会退出。
   要在 traced 程序退出前终止监控，请按 **Ctrl+C**。
   - 如果 **strace** 启动程序，则程序会与 **strace** 一起终止。
   - 如果您将 **strace** 附加到已在运行的程序中，则程序会与 **strace** 一起终止。
5. 分析应用所完成的系统调用列表。
   - 当调用返回错误时，日志中会出现资源访问或可用性的问题。
   - 传递给系统调用和调用序列模式的值可以了解应用程序的行为。
   - 如果应用程序崩溃，重要信息可能会在日志的末尾。
   - 输出中包含大量不必要的信息。但是，您可以为感兴趣的系统调用构建更加精确的过滤器，并重复这个过程。

注意

最好查看输出并将其保存到文件中。使用 **tee** 命令实现这一目的：

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

其他资源

- *strace(1)* 手册页：

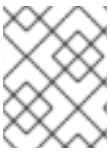
  ```
  $ man strace
  ```

- 如何使用 strace 来跟踪命令发出的系统调用？ - 知识库文章

- Red Hat Developer Toolset 用户指南 - strace 章

3.3.3. 使用 ltrace 监控应用程序的库功能调用

ltrace 工具可监控应用程序对库中可用函数的调用（共享对象）。



注意

在 Red Hat Enterprise Linux 9 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提供的可执行文件。

先决条件

- 在系统中必须安装 **ltrace**。

步骤

1. 如果可能，请确定感兴趣的库和功能。
2. 启动 **ltrace** 并将其附加到程序。
   - 如果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
     ```

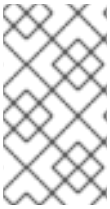
   - 如果程序已在运行，找到其进程 ID(*pid*)并为其附加 **ltrace** ：

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

   - 使用 **-e**、**-f** 和 **-l** 选项过滤输出：
     - 提供要显示的函数名称作为 *function*。**-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，而不是指定功能。这个选项的行为与 **-e *function*** 选项类似。
     - 如果您不想跟踪任何 fork 进程或线程，请退出 **-f** 选项。

     如需更多信息，请参阅 *ltrace*(1)_ manual page。
3. **ltrace** 显示应用程序发出的库调用。
   在大多数情况下，如果没有设置任何过滤器，应用程序会立即显示大量调用和 **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 **ctrl+C**。
   - 如果 **ltrace** 启动程序，则程序会与 **ltrace** 一起终止。
   - 如果您将 **ltrace** 附加到已运行程序，则程序会与 **ltrace** 一起终止。
5. 分析应用所完成的库调用列表。

- 如果应用程序崩溃，重要信息可能会在日志的末尾。
- 输出中包含大量不必要的信息。但是，您可以构建更加精确的过滤器并重复这个过程。



注意

最好查看输出并将其保存到文件中。使用 **tee** 命令实现这一目的：

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

其他资源

- *ltrace(1)* 手册页：

```
$ man ltrace
```

- Red Hat Developer Toolset User Guide - ltrace

3.3.4. 使用 SystemTap 监控应用程序的系统调用

SystemTap 工具为内核事件注册自定义事件处理程序。与 **strace** 工具相比，使用它较复杂，但它的效率更高，并可以启用更复杂的处理逻辑。名为 **strace.stp** 的 SystemTap 脚本与 SystemTap 一起安装，并可实现使用 SystemTap 的 **strace** 大致相同的功能。

先决条件

- 必须在系统中安装 SystemTap 和相关的内核软件包。

步骤

1. 找到您要监控的进程 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。

脚本编译到内核模块中，然后载入该模块。这会在输入命令并获取输出之间引入小延迟。

3. 当进程执行系统调用时，调用名称及其参数会打印到终端中。
4. 当进程终止或按 **Ctrl+C** 时，该脚本会退出。

3.3.5. 使用 GDB 截获应用程序系统调用

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 断点的使用。
- GDB 必须附加到程序。

步骤

1. 设置 catchpoint:

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

命令 **catch syscall** 设置了一个特殊的断点，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追踪点。使用 ***syscall-name*** 选项会使 GDB 在任意系统调用中停止。

2. 开始执行程序。
   - 如果程序还没有启动执行，请启动它：

```
(gdb) r
```

   - 如果程序执行停止，请恢复它：

```
(gdb) c
```

3. GDB 在程序执行任何指定的系统调用后停止执行。

其他资源

- 使用 GDB 调试 - 设置 Watchpoints

3.3.6. 使用 GDB 截获应用程序处理信号

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 断点的使用。
- GDB 必须附加到程序。

步骤

1. 设置 catchpoint:

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

命令 **catch signal** 设定一种特殊类型的断点，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"** 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
```

- 如果程序执行停止，请恢复它：

```
(gdb) c
```

3. GDB 在程序收到任何指定信号后停止执行。

其他资源

- 使用 GDB 调试 – 5.1.3 Setting Catchpoints

3.4. 调试 CRASHED 应用程序

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

3.4.1. 核心转储：它们是什么以及如何使用它们

内核转储(core dump)是应用程序在应用程序停止工作时的一部分的副本，它以 ELF 格式存储。它包含所有应用的内部变量和堆栈，可启用检查应用程序的最终状态。当遵守相应的可执行文件和调试信息时，可以通过调试器来分析核心转储文件，类似于分析正在运行的程序。

如果启用了此功能，Linux 操作系统内核可以自动记录核心转储。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功能。查看当前的限制：

```
$ ulimit -a
```

3.4.2. 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系统的信息。

步骤

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，说明之前是否存在这些设置，以及前面的值是什么。如果需要，这将使您稍后撤销这些更改。注释是以 **#** 字符开头的行。

更改该文件需要管理员级别访问权限。

2. 应用新配置：

```
# systemctl daemon-reexec
```

3. 删除内核转储大小的限制：

```
# ulimit -c unlimited
```

要反向更改，请使用值 **0** 而不是 ***unlimited*** 来运行 命令。

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

```
# dnf install sos
```

5. 当应用程序崩溃时，会生成核心转储，并由 **systemd-coredump** 处理。

6. 创建 SOS 报告以提供系统的附加信息：

```
# sosreport
```

这会创建一个 **.tar** 存档包含您的系统信息，如配置文件副本。

7. 查找并导出内核转储：

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

如果应用程序多次崩溃，则第一个命令的输出列出了更多捕获的内核转储。在这种情况下，使用其他信息为第二个命令构建更精确的查询。详情请查看 *coredumpctl(1)* 手册页。

8. 将内核转储和 SOS 报告传送到进行调试的计算机。另外，也传输可执行文件（如果该文件已知）。

重要

当可执行文件未知时，然后对核心文件进行后续分析会标识它。

9. 可选：传输后删除内核转储和 SOS 报告，以释放磁盘空间。

其他资源

- 文档 *配置基本系统设置* 中的 管理 systemd
- 当应用程序崩溃或分段错误时，如何启用核心文件转储 - 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？ - 一个知识库文章

3.4.3. 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，以及发生崩溃的系统的 sosreport。

- 在您的系统中必须安装 GDB 和 elfutils。

步骤

1. 要识别发生崩溃的可执行文件，请运行带内核转储文件的 **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
```

   输出中包含一行中每个模块的详细信息，用空格分开。该信息按以下顺序列出：

   1. 模块映射的内存地址
   2. 模块的 build-id 以及它在找到的内存中的位置
   3. 模块的可执行文件名称，如果未知显示为 **-**，如果模块没有从文件中加载则显示为 **.**。
   4. 调试信息的来源，在可用时显示为一个文件名，当包括在可执行文件本身中时显示为 **.**，或在不存在时显示 **-**
   5. 主模块的共享库名称(*soname*)或 **[exe]**

   在本例中，重要的细节是文件名 **/usr/bin/sleep** 以及在包括文本 **[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，在发生崩溃的系统中复制它。使用从核心文件中提取的文件名。
   - 您还可以在系统中使用相同的可执行文件。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。确定相关本地可用可执行文件的构建 ID：

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

     使用此信息将远程系统中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，如果应用程序从 RPM 软件包安装，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. 获取可执行文件使用的共享库。对可执行文件使用的步骤相同。

4. 如果应用程序以软件包的形式发布，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 3.1.4 节 “使用 GDB 获取应用程序或库的 debuginfo 软件包”。

5. 要详细检查核心文件，使用 GDB 加载可执行文件和内核转储文件：

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

有关缺失的文件和调试信息的进一步消息可帮助您识别调试会话中缺少什么。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而不是软件包提供，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

对于核心转储中包含的所有可执行文件，可能不需要安装调试信息。其中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为问题贡献，您不需要为它们包含调试信息。

6. 使用 GDB 命令在应用程序崩溃时检查应用程序状态。请参阅使用 GDB 检查应用程序内部状态。

注意

在分析内核文件时，GDB 不会附加到正在运行的进程。用于控制执行的命令无效。

其他资源

- 使用 GDB 进行调试 - 2.1.1 Choosing Files
- 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

3.4.4. 使用 coredumpctl 创建和访问内核转储

systemd 的 **coredumpctl** 工具可显著简化崩溃发生的机器中的核心转储。此流程概述了如何捕获无响应进程的内核转储。

先决条件

- 该系统必须配置为使用 **systemd-coredump** 进行内核转储处理。验证这一点是否正确：

```
$ sysctl kernel.core_pattern
```

如果输出以以下内容开头，则配置是正确的：

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

步骤

1. 根据可执行文件名称的已知部分，查找挂起进程的 PID：

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

这个命令会以以下形式输出一行

```
PID command-line
```

使用 *命令行* 值来验证 *PID* 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 **coredumpctl** 是否捕获了内核：

```
$ 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. 可以根据需要进一步检查或使用核心文件。
您可以根据 PID 和其他值指定内核转储。详情请查看 *coredumpctl(1)* 手册页。

- 显示核心文件的详情：

```
$ coredumpctl info PID
```

- 要在 GDB 调试器中载入核心文件：

```
$ coredumpctl debug PID
```

根据调试信息的可用性，GDB 将会建议运行命令，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，请参阅使用 GDB 获取应用程序或库的 debuginfo 软件包。

- 导出核心文件以便进一步处理其他位置：

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

使用您要放置内核转储的文件替换 */path/to/file_for_export*。

3.4.5. 使用 gcore转储进程内存

核心转储调试的工作流可以离线分析程序的状态。在某些情况下，您可以将此工作流与仍然运行的程序搭配使用，比如很难通过进程访问环境。您可以使用 **gcore** 命令在仍在运行时转储任何进程的内存。

先决条件

- 您必须了解什么是核心转储，以及如何创建它们。
- 在系统中必须安装 GDB。

步骤

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

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

2. 转储这个进程的内存：

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

这会创建一个文件 ***filename*** 并转储进程内存。转储内存时，进程的执行将停止。

3. 在内核转储完成后，进程会恢复正常执行。
4. 创建 SOS 报告以提供系统的附加信息：

```
# sosreport
```

这会创建一个 tar 存档，其中包含您的系统信息，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：传输后删除内核转储和 SOS 报告，以释放磁盘空间。

其他资源

- 如何在不重启应用程序的情况下获取核心文件？ - 知识库文章

3.4.6. 使用 GDB 转储受保护的进程内存

您可以将进程内存标记为不转储。这可节省资源并确保进程内存包含敏感数据时的其他安全：例如，在银行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）都不会转储标记为这种方式的内存。

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 调试器进行此操作。

先决条件

- 您必须了解什么是核心转储。
- 在系统中必须安装 GDB。
- GDB 必须已附加到带有受保护内存的进程。

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*。

其他资源

- 使用 GDB 进行调试 - 如何从您的程序减少核心文件

3.5. GDB 中兼容性破坏的更改

Red Hat Enterprise Linux 9 中提供的 GDB 版本包含几个破坏兼容性的更改。以下部分详细介绍了这些变化。

命令

- **gdb -P python-script.py** 命令不再支持。
  改为使用 **gdb -ex 'source python-script.py'** 命令。
- **gdb *COREFILE*** 命令不再支持。
  改为使用 **gdb *EXECUTABLE* --core *COREFILE*** 命令加载核文件中指定的可执行文件。
- GDB 现在默认设置输出样式。
  这个新更改可能会破坏尝试解析 GDB 输出的脚本。使用 **gdb -ex 'set style enabled off'** 命令禁用脚本中的样式。
- 命令现在根据语言为符号定义语法。
  info functions、**info types**、**info variables** 和 **rbreak** 命令现在根据 **set language** 命令所选择的语言为实体定义语法。通过将其设置为 **set language auto** 表示 GDB 将自动选择显示实体的语言。
- **set print raw frame-arguments** 和 **show print raw frame-arguments** 命令已被弃用。
  这些命令被 **set print raw-frame-arguments** 和 **show print raw-frame-arguments** 命令替代。旧的命令可能会在以后的版本中删除。
- 以下 TUI 命令现在区分大小写：
  - **focus**
  - **winheight**
  - **+**
  - **-**

- >
- <

- **help** 和 **apropos** 命令现在只显示一次命令信息。
  现在，这些命令只显示一次命令的文档，即使该命令有一个或多个别名。这些命令现在显示命令名称，然后是其所有别名，最后是命令的描述。

MI 解释器

- MI 解释器的默认版本现在是 3。
  MI 3 中更改了有关多位置断点（其在 MI 2 中语法不正确）信息的输出。这会影响以下命令和事件：
  - **-break-insert**
  - **-break-info**
  - **=breakpoint-created**
  - **=breakpoint-modified**

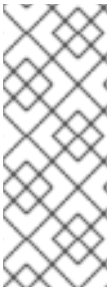
  使用 **-fix-multi-location-breakpoint-output** 命令在之前的 MI 版本中启用此行为。

Python API

- 以下符号现已弃用：
  - **gdb.SYMBOL_VARIABLES_DOMAIN**
  - **gdb.SYMBOL_FUNCTIONS_DOMAIN**
  - **gdb.SYMBOL_TYPES_DOMAIN**
- **gdb.Value** 类型有一个新的构造器，用于从 Python 缓冲对象和 **gdb.Type** 构造 **gdb.Value**。
- 现在，当没有过滤器或使用 **backtrace** 命令的 **-no-filters** 选项时，Python 帧过滤代码打印的帧信息与 **backtrace** 命令打印的帧信息一致。

3.6. 在容器中调试应用程序

您可以使用为故障排除的不同方面量身定制的各种命令行工具。以下提供了类别以及常用的命令行工具。



注意

这不是命令行工具的完整列表。调试容器应用程序的工具的选择主要基于容器镜像和您的用例。

例如，**systemctl**,**journalctl**,**ip**,**netstat**,**ping**,**traceroute**,**perf**,**iostat** 工具可能需要 root 权限，因为它们与系统级资源（如网络、systemd 服务或硬件性能计数器）进行交互，出于安全原因，他们在无根容器中受到限制。

无根容器的操作不需要提升特权，在用户命名空间中以非 root 用户身份运行，以提供改进的安全及与主机系统的隔离。它们通过缓解特权升级漏洞的风险，提供与主机有限的交互，降低了攻击面，并提高了安全性。

有根容器使用提升的特权运行，通常以 root 用户身份运行，对系统资源和功能授予完整的访问权限。虽然有根容器提供更大的灵活性和控制，但由于潜在的特权升级和主机系统暴露给漏洞，它们会带来安全风险。

有关根和无根容器的更多信息，请参阅 设置无根容器、升级到无根容器，以及 对无根容器的特殊考虑。

systemd 和进程管理工具

systemctl

在容器内控制 systemd 服务，允许启动、停止、启用和禁用操作。

journalctl

查看 systemd 服务产生的日志，帮助故障排除容器问题。

网络工具

ip

在容器中管理网络接口、路由和地址。

netstat

显示网络连接、路由表和接口统计信息。

ping

验证容器或主机之间的网络连接性。

traceroute

标识数据包到达目的地的路径，用于诊断网络问题。

进程和性能工具

ps

列出容器中当前运行的进程。

top

提供对容器内按进程的资源使用情况的实时洞察。

htop

用于监控资源使用率的交互式进程查看器。

perf

CPU 性能分析、追踪和监控，帮助识别系统或应用程序中的性能瓶颈。

vmstat

报告容器内的虚拟内存统计信息，帮助性能分析。

iostat

监控容器中块设备的输入/输出统计信息。

GDB (GNU 调试器)

一个命令行调试器，通过允许用户跟踪和控制其执行、检查变量，并在运行时分析内存和注册器来帮助检查和调试程序。如需更多信息，请参阅 在 Red Hat OpenShift 容器中调试应用程序 文章。

strace

拦截并记录程序发出的系统调用，通过揭示程序与操作系统之间的交互来帮助进行故障排除。

安全和访问控制工具

sudo

启用执行具有升级权限的命令。

chroot

更改命令的根目录，帮助在不同根目录中进行测试或故障排除。

podman 专用工具

podman logs

批量检索一个或多个容器在执行时存在的任何日志。

podman inspect

显示有关按名称或 ID 标识的容器和镜像的低级信息。

podman events

监控并打印 Podman 中发生的事件。每个事件都包括时间戳、类型、状态、名称（如果适用）和镜像（如果适用）。默认日志机制是 **journald**。

podman run --health-cmd

使用健康检查来确定容器中运行的进程的健康状态或就绪状态。

podman top

显示容器的运行进程。

podman exec

在容器中运行命令或附加到正在运行的容器，对于更好地了解容器中发生的情况非常有用。

podman export

当容器出现故障时，基本上无法知道发生了什么。从容器中导出文件系统结构将允许检查可能不在挂载卷中的其他日志文件。

其他资源

- 在 Red Hat OpenShift 容器中调试应用程序
  - **gdb**
- 调试 Crashed 应用程序
  - Core dump、**sosreport**、**gdb**、**ps**、**core** 。
- 对 Kubernetes 进行故障排除
  - docker exec + env, **netstat**,**kubectl**,**etcdctl**,**journalctl**, docker logs
- 容器化服务的提示和技巧
  - watch, **podman logs**,**systemctl**,**podman exec/kill/restart**,**podman insect**,**podman top**,**podman exec**,**podman export**,**paunch**
- 外部链接
  - 调试 Docker 容器的十个技巧

第 4 章 用于开发的额外工具集

4.1. 使用 GCC 工具集

4.1.1. 什么是 GCC Toolset

Red Hat Enterprise Linux 9 继续对 GCC Toolset 的支持，它是一个包含更新的开发和性能分析工具的应用程序流。GCC Toolset 与 RHEL 7 的 Red Hat Developer Toolset 类似。

GCC Toolset 以 **AppStream** 存储库中的一个软件集合的形式作为 Application Stream 提供。GCC Toolset 在 Red Hat Enterprise Linux 订阅级别协议中被完全支持，其功能完整并适用于生产环境。GCC Toolset 提供的应用程序和库不会替换 Red Hat Enterprise Linux 系统版本，不会覆盖它们，且不会自动成为默认或首选选择。使用称为软件集合的框架，将额外的开发人员工具安装到 **/opt/** 目录中，用户可使用 **scl** 工具根据需要明确启用。除非对特定工具或功能另有说明，否则 GCC Toolset 可供 Red Hat Enterprise Linux 支持的所有架构使用。

4.1.2. 安装 GCC 工具集

在系统中安装 GCC Toolset 会安装主工具和所有必要的依赖项。请注意，默认情况下工具集的某些部分不会被安装，您需要单独安装它们。

步骤

- 安装 GCC Toolset 版本 *N* ：

```
# dnf install gcc-toolset-N
```

4.1.3. 从 GCC Toolset 安装单个软件包

如果只需安装 GCC Toolset 中的特定工具而不是安装整个工具集，请列出可用的软件包并使用 **dnf** 软件包管理工具安装所选工具。对于默认情况下没有通过 toolset 安装的软件包，这个过程也很有用。

步骤

1. 列出 GCC Toolset 版本 *N* 中可用的软件包：

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

2. 安装这些软件包：

```
# dnf install package_name
```

使用要安装的软件包列表替换 *package_name*。例如，要安装 **gcc-toolset-9-gdb-gdbserver** 和 **gcc-toolset-9-gdb-doc** 软件包：

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

4.1.4. 卸载 GCC Toolset

要从系统中删除 GCC Toolset，请使用 **dnf** 软件包管理工具卸载它。

步骤

- 卸载 GCC Toolset 版本 *N* ：

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

4.1.5. 从 GCC Toolset 运行工具

要运行来自 GCC Toolset 的工具，请使用 **scl**。

步骤

- 要从 GCC Toolset 版本 *N* 运行工具：

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

4.1.6. 使用 GCC Toolset 运行 shell 会话

在无需明确使用 **scl** 命令的情况下，GCC Toolset 允许运行一个使用 GCC Toolset 工具版本的 shell 会话，而不是这些工具的系统版本。当您需要多次交互启动工具时（如设置或测试开发设置）时，这非常有用。

步骤

- 要运行一个 shell 会话，其中 GCC Toolset 版本 *N* 中的工具版本会覆盖这些工具的系统版本：

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

4.1.7. 其他资源

- Red Hat Developer Toolset 用户指南

4.2. GCC TOOLSET 12

了解特定于 GCC Toolset 版本 12 的信息以及此版本中包含的工具。

4.2.1. GCC Toolset 12 提供的工具和版本

GCC Toolset 12 提供以下工具和版本：

表 4.1. GCC Toolset 12 中的工具版本

Name	版本	描述
GCC	12.2.1	可移植编译器套件，支持 C、C++ 和 Fortran。
GDB	11.2	对使用 C、C++ 和 Fortran 编写的程序的命令行调试器。
binutils	2.38	用于检查和操作对象文件和二进制文件的二进制工具集合。

Name	版本	描述
dwz	0.14	可优化 ELF 共享库和 ELF 可执行文件中包含的 DWARF 调试信息的工具，以获得大小。
annobin	11.08	构建安全检查工具。

4.2.2. GCC Toolset 12 中的 C++ 兼容性

重要

这里给出的兼容性信息只适用于 GCC Toolset 12 中的 GCC。

GCC Toolset 中的 GCC 编译器可使用以下 C++ 标准：

C++14

这个语言标准包括在 GCC Toolset 12 中。
当使用相应标志编译的所有 C++ 对象使用 GCC 版本 6 或更高版本构建时，支持使用 C++14 语言版本。

C++11

这个语言标准包括在 GCC Toolset 12 中。
当使用相应标志编译的所有 C++ 对象使用 GCC 版本 5 或更高版本构建时，支持使用 C++11 语言版本。

C++98

这个语言标准包括在 GCC Toolset 12 中。无论使用 GCC Toolset、Red Hat Developer Toolset 和 RHEL 5、6、7 和 8 的 GCC 都可以自由混合使用这个标准构建的二进制文件、共享库和对象。

C++17

这个语言标准包括在 GCC Toolset 12 中。
这是 GCC Toolset 12 的默认语言标准设置，它相当于明确使用 **-std=gnu++17** 选项。

当使用相应标志编译的所有 C++ 对象使用 GCC 版本 10 或更高版本构建时，支持使用 C++17 语言版本。

c++20 和 C++23

这个语言标准在 GCC Toolset 12 中提供，但应视为是实验性的、不稳定和不受支持的功能。另外，无法保证使用此标准构建的对象、二进制文件和库的兼容性。
要启用 C++20 支持，请在 g++ 命令行中添加命令行选项 **-std=c++20**。

要启用 C++23 支持，请在 g++ 命令行中添加命令行选项 **-std=c++23**。

所有语言标准均有标准兼容变体和 GNU 扩展。

当使用 GCC Toolset 和这些 RHEL 工具链（特别是 **.o** 或 **.a** 文件）混合对象构建时，GCC Toolset 工具链应该用于所有链接。这样可确保仅由 GCC Toolset 提供的较新的库功能在链接时得以解决。

4.2.3. GCC Toolset 12 中的 GCC 的具体信息

库的静态链接

某些最新的库功能被静态链接到使用 GCC Toolset 构建的应用程序，以支持在多个 Red Hat Enterprise Linux 版本上执行。这会产生一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误没有改变这个代码。如果因这个风险开发人员需要重建其应用程序，红帽将使用一个安全勘表进行沟通。

重要

由于这个额外的安全风险，强烈建议开发人员不要静态地将整个应用程序链接在一起。

在进行链接时，在对象文件后指定库

在 GCC Toolset 中，库使用 linker 脚本链接，这些脚本可通过静态归档指定某些符号。这是为了确保与多个 Red Hat Enterprise Linux 版本兼容所必需的。但是，链接器脚本使用对应的共享对象文件的名称。因此，linker 使用不同于预期的符号处理规则，在指定程序库选项前不识别对象文件所需的符号：

```
$ scl enable gcc-toolset-12 'gcc -lsomelib objfile.o'
```

以这种方式使用 GCC Toolset 的库会导致 linker 错误消息 **undefined reference to symbol**。要防止这个问题，请按照标准链接实践操作，并在指定对象文件的选项后添加库：

```
$ scl enable gcc-toolset-12 'gcc objfile.o -lsomelib'
```

请注意，这个建议也适用于使用 **GCC** 的基本 Red Hat Enterprise Linux 版本。

4.2.4. GCC Toolset 12 中的 binutils 的具体信息

库的静态链接

某些最新的库功能被静态链接到使用 GCC Toolset 构建的应用程序，以支持在多个 Red Hat Enterprise Linux 版本上执行。这会产生一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误没有改变这个代码。如果因这个风险开发人员需要重建其应用程序，红帽将使用一个安全勘表进行沟通。

重要

由于这个额外的安全风险，强烈建议开发人员不要静态地将整个应用程序链接在一起。

在进行链接时，在对象文件后指定库

在 GCC Toolset 中，库使用 linker 脚本链接，这些脚本可通过静态归档指定某些符号。这是为了确保与多个 Red Hat Enterprise Linux 版本兼容所必需的。但是，链接器脚本使用对应的共享对象文件的名称。因此，linker 使用不同于预期的符号处理规则，在指定程序库选项前不识别对象文件所需的符号：

```
$ scl enable gcc-toolset-12 'ld -lsomelib objfile.o'
```

以这种方式使用 GCC Toolset 的库会导致 linker 错误消息 **undefined reference to symbol**。要防止这个问题，请按照标准链接实践操作，并在指定对象文件的选项后添加库：

```
$ scl enable gcc-toolset-12 'ld objfile.o -lsomelib'
```

请注意，这个建议也适用于使用 **binutils** 的基本 Red Hat Enterprise Linux 版本。

4.2.5. GCC Toolset 12 中的 annobin 的具体内容

在某些情况下，由于 GCC Toolset 12 中的 **annobin** 和 **gcc** 之间的同步问题，您的编译可能会失败，并显示类似如下的错误消息：

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

要临时解决这个问题，请从 **annobin.so** 文件中创建一个符号链接到 **gcc-annobin.so** 文件中：

```
# cd /opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin
# ln -s annobin.so gcc-annobin.so
```

使用您系统中使用的构架替换 *architecture* ：

- **aarch64**
- **i686**
- **ppc64le**
- **s390x**
- **x86_64**

4.3. GCC TOOLSET 13

了解特定于 GCC Toolset 版本 13 的信息以及此版本中包含的工具。

4.3.1. GCC Toolset 13 提供的工具和版本

GCC Toolset 13 提供以下工具和版本：

表 4.2. GCC Toolset 13 中的工具版本

Name	版本	描述
GCC	13.2.1	可移植编译器套件，支持 C、C++ 和 Fortran。
GDB	12.1	对使用 C、C++ 和 Fortran 编写的程序的命令行调试器。
binutils	2.40	用于检查和操作对象文件和二进制文件的二进制工具集合。
dwz	0.14	可优化 ELF 共享库和 ELF 可执行文件中包含的 DWARF 调试信息的工具，以获得大小。
annobin	12.32	构建安全检查工具。

4.3.2. GCC Toolset 13 中的 C++ 兼容性

重要

此处显示的兼容性信息只适用于 GCC Toolset 13 中的 GCC。

GCC Toolset 中的 GCC 编译器可使用以下 C++ 标准：

C++14

这个语言标准在 GCC Toolset 13 中提供。
当使用相应标志编译的所有 C++ 对象使用 GCC 版本 6 或更高版本构建时，支持使用 C++14 语言版本。

C++11

这个语言标准在 GCC Toolset 13 中提供。
当使用相应标志编译的所有 C++ 对象使用 GCC 版本 5 或更高版本构建时，支持使用 C++11 语言版本。

C++98

这个语言标准在 GCC Toolset 13 中提供。无论使用 GCC Toolset、Red Hat Developer Toolset 和 RHEL 5、6、7 和 8 的 GCC 都可以自由混合使用这个标准构建的二进制文件、共享库和对象。

C++17

这个语言标准在 GCC Toolset 13 中提供。
这是 GCC Toolset 13 的默认语言标准设置，带有 GNU 扩展，相当于明确使用 **-std=gnu++17** 选项。

当使用相应标志编译的所有 C++ 对象使用 GCC 版本 10 或更高版本构建时，支持使用 C++17 语言版本。

c++20 和 C++23

这些语言标准仅在 GCC Toolset 13 中作为实验性、不稳定的和不支持的功能提供。另外，无法保证使用此标准构建的对象、二进制文件和库的兼容性。
要启用 C++20 标准，请在 g++ 命令行中添加命令行选项 **-std=c++20**。

要启用 C++23 标准，请在 g++ 命令行中添加命令行选项 **-std=c++23**。

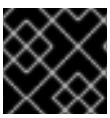
所有语言标准均有标准兼容变体和 GNU 扩展。

当使用 GCC Toolset 和这些 RHEL 工具链（特别是 **.o** 或 **.a** 文件）混合对象构建时，GCC Toolset 工具链应该用于所有链接。这样可确保仅由 GCC Toolset 提供的较新的库功能在链接时得以解决。

4.3.3. GCC Toolset 13 中 GCC 的具体内容

库的静态链接

某些最新的库功能被静态链接到使用 GCC Toolset 构建的应用程序，以支持在多个 Red Hat Enterprise Linux 版本上执行。这会产生一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误没有改变这个代码。如果因这个风险开发人员需要重建其应用程序，红帽将使用一个安全勘表进行沟通。



重要

由于这个额外的安全风险，强烈建议开发人员不要静态地将整个应用程序链接在一起。

在进行链接时，在对象文件后指定库

在 GCC Toolset 中，库使用 linker 脚本链接，这些脚本可通过静态归档指定某些符号。这是为了确保与多个 Red Hat Enterprise Linux 版本兼容所必需的。但是，链接器脚本使用对应的共享对象文件的名称。因此，linker 使用不同于预期的符号处理规则，在指定程序库选项前不识别对象文件所需的符号：

```
$ scl enable gcc-toolset-13 'gcc -lsomelib objfile.o'
```

以这种方式使用 GCC Toolset 的库会导致 linker 错误消息 **undefined reference to symbol**。要防止这个问题，请按照标准链接实践操作，并在指定对象文件的选项后添加库：

```
$ scl enable gcc-toolset-13 'gcc objfile.o -lsomelib'
```

请注意，这个建议也适用于使用 **GCC** 的基本 Red Hat Enterprise Linux 版本。

4.3.4. GCC Toolset 13 中 binutils 的具体内容

库的静态链接

某些最新的库功能被静态链接到使用 GCC Toolset 构建的应用程序，以支持在多个 Red Hat Enterprise Linux 版本上执行。这会产生一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误没有改变这个代码。如果因这个风险开发人员需要重建其应用程序，红帽将使用一个安全勘表进行沟通。

重要

由于这个额外的安全风险，强烈建议开发人员不要静态地将整个应用程序链接在一起。

在进行链接时，在对象文件后指定库

在 GCC Toolset 中，库使用 linker 脚本链接，这些脚本可通过静态归档指定某些符号。这是为了确保与多个 Red Hat Enterprise Linux 版本兼容所必需的。但是，链接器脚本使用对应的共享对象文件的名称。因此，linker 使用不同于预期的符号处理规则，在指定程序库选项前不识别对象文件所需的符号：

```
$ scl enable gcc-toolset-13 'ld -lsomelib objfile.o'
```

以这种方式使用 GCC Toolset 的库会导致 linker 错误消息 **undefined reference to symbol**。要防止这个问题，请按照标准链接实践操作，并在指定对象文件的选项后添加库：

```
$ scl enable gcc-toolset-13 'ld objfile.o -lsomelib'
```

请注意，这个建议也适用于使用 **binutils** 的基本 Red Hat Enterprise Linux 版本。

4.3.5. GCC Toolset 13 中 annobin 的具体内容

在某些情况下，由于 GCC Toolset 13 中 **annobin** 和 **gcc** 之间的同步问题，您的编译可能会失败，并显示类似如下的错误消息：

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-13/root/usr/lib/gcc/architecture-linux-gnu/13/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

要临时解决这个问题，请从 **annobin.so** 文件中创建一个符号链接到 **gcc-annobin.so** 文件中：

```
# cd /opt/rh/gcc-toolset-13/root/usr/lib/gcc/architecture-linux-gnu/13/plugin
# ln -s annobin.so gcc-annobin.so
```

使用您系统中使用的构架替换 *architecture* ：

- **aarch64**
- **i686**
- **ppc64le**
- **s390x**
- **x86_64**

4.4. 使用 GCC TOOLSET 容器镜像

仅 支持 GCC Toolset 13 容器镜像。之前 GCC Toolset 版本的容器镜像已弃用。

GCC Toolset 13 组件在 **GCC Toolset 13 Toolchain**容器镜像中提供。

GCC Toolset 容器镜像基于 **rhel9** 基础镜像，并可用于 RHEL 9 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

4.4.1. GCC Toolset 容器镜像内容

GCC Toolset 13 容器镜像中提供的工具版本与 GCC Toolset 13 组件版本 匹配。

GCC Toolset 13 Toolchain 内容

rhel9/gcc-toolset-13-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-13-gcc
g++	gcc-toolset-13-gcc-c++
gfortran	gcc-toolset-13-gcc-gfortran
gdb	gcc-toolset-13-gdb

4.4.2. 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

- podman 已安装。

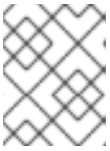
步骤

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. 作为 root 运行相关命令来拉取所需的容器镜像：

```
# podman pull registry.redhat.io/rhel9/gcc-toolset-13-toolchain
```



注意

您还可以设置您的系统，以非 root 用户身份使用容器。详情请参阅设置无根容器。

3. 可选：运行列出本地系统中的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；没有此选项，shell 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，则无法输入任何 shell。

其他资源

- 在 RHEL 9 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 root
- Red Hat Container Registry - GCC Toolset 容器镜像中的条目

4.4.3. 例如：使用 GCC Toolset 13 Toolchain 容器镜像

本例演示了如何拉取并开始使用 GCC Toolset 13 Toolchain 容器镜像。

先决条件

- podman 已安装。

步骤

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 用户身份拉取容器镜像：

```
# podman pull registry.redhat.io/rhel9/gcc-toolset-13-toolchain
```

3. 以 root 用户身份使用交互式 shell 启动容器镜像：

```
# podman run -it registry.redhat.io/rhel9/gcc-toolset-13-toolchain /bin/bash
```

4. 按预期运行 GCC 工具集工具。例如，要验证 **gcc** 编译器版本，请运行：

```
bash-4.4$ gcc -v
...
gcc version 13.1.1 20231102 (Red Hat 13.1.1-4) (GCC)
```

5. 要列出容器提供的所有软件包，请运行：

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

4.5. 编译器工具集

RHEL 9 提供以下编译器工具集作为应用程序流：

- LLVM Toolset 提供 LLVM 编译器基础架构框架、Clang 编译器、ClangDB 调试器以及用于代码分析的相关工具。
- Rust Toolset 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** 插件和所需的库。
- Go Toolset 提供 Go 编程语言工具和库。Go 也称为 **golang**。

有关用法的详情和信息，请参阅 Red Hat Developer Tools 页中的编译器工具集用户指南。

4.6. ANNOBIN 项目

Annobin 项目是 Watermark 规格项目的实现。水位线规格项目旨在向可执行文件和可链接格式(ELF)对象添加标记，以确定其属性。Annobin 项目由 **annobin** 插件和 **annockeck** 程序组成。

anobin 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 备注记录了二进制文件的构建方式，并为 **annocheck** 程序提供信息来执行安全强化检查。

安全强化检查程序是 **annocheck** 程序的一部分，默认是启用的。它检查二进制文件，以确定是否使用必要的安全强化选项构建程序。**annocheck** 能够为 ELF 对象文件递归扫描目录、存档和 RPM 软件包。

注意

这些文件必须采用 ELF 格式。**anocheck** 不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

4.6.1. 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

4.6.1.1. 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。

步骤

- 要启用带有 **annobin** 插件的 **gcc**，请使用：

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

  - 如果 **gcc** 找不到 **annobin** 插件，请使用：

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

    将 */path/to/directory/containing/annobin/* 替换为包含 **annobin** 的目录的绝对路径。

  - 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 **gcc**，请使用：

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

  将 */path/to/directory/containing/annobin/* 替换为包含 **annobin** 的目录的绝对路径。

4.6.1.2. 将选项传递给 annobin 插件

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

步骤

- 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 *option*，并使用文件名替换 *file-name*。

  示例

- 要显示 **annobin** 正在进行的操作的其他详情，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 *option*，并将 */path/to/directory/containing/annobin/* 替换为包含 **annobin** 的目录的绝对路径。

示例

- 要显示 **annobin** 正在进行的操作的其他详情，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*。

4.6.2. 使用 annocheck 程序

下面的部分论述了如何使用 **annocheck** ：

- 文件
- 目录
- RPM 软件包
- **annocheck** 额外工具

注意

annocheck 递归扫描 ELF 对象文件的目录、存档和 RPM 软件包。文件必须采用 ELF 格式。**annocheck** 不处理任何其他二进制文件类型。

4.6.2.1. 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。

步骤

- 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。**anocheck** 不处理任何其他二进制文件类型。**annocheck** 会处理包含 ELF 对象文件的静态库。

附加信息

- 有关 **annocheck** 和可用命令行选项的详情请参考 **annocheck** man page。

4.6.2.2. 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。

步骤

- 要扫描目录，请使用：

```
$ annocheck directory-name
```

使用目录名称替换 *directory-name*。**annocheck** 会自动检查目录的内容、其子目录以及该目录中的任何存档和 RPM 软件包。

注意

annocheck 仅查找 ELF 文件。其他文件类型将被忽略。

附加信息

- 有关 **annocheck** 和可用命令行选项的详情请参考 **annocheck** man page。

4.6.2.3. 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。

步骤

- 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*。**nocheck** 会以递归方式扫描 RPM 软件包中的所有 ELF 文件。

注意

annocheck 仅查找 ELF 文件。其他文件类型将被忽略。

- 要使用提供 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*，使用与二进制 RPM 关联的调试信息 RPM 的名称替换 *debuginfo-rpm*。

附加信息

- 有关 **annocheck** 和可用命令行选项的详情请参考 **annocheck** man page。

4.6.2.4. 使用 annocheck 额外的工具

annocheck 包括多个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

强化检查程序被默认启用。

4.6.2.4.1. 启用 built-by 工具

您可以使用 **annocheck built-by** 工具来查找用于构建二进制文件的编译器的名称。

步骤

- 要启用 **built-by** 工具，请使用：

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

附加信息

- 有关 **built-by** 工具的更多信息，请参阅 **--help** 命令行选项。

4.6.2.4.2. 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注释。

步骤

- 要启用 **notes** 工具，请使用：

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

注释按地址范围排序。

附加信息

- 有关 **notes** 工具的详情，请查看 **--help** 命令行选项。

4.6.2.4.3. 启用 section-size 工具

您可以使用 **annocheck section-size** 工具显示指定部分的大小。

步骤

- 要启用 **section-size** 工具，请使用：

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

  使用指定部分的名称替换 *name*。输出仅限于特定的部分。结束时会生成累积结果。

附加信息

- 有关 **section-size** 工具的更多信息，请参阅 **--help** 命令行选项。

4.6.2.4.4. 强化检查程序基础知识

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

4.6.2.4.4.1. 强化检查程序选项

annocheck 程序检查以下选项：

- 使用 **-z now** linker 选项禁用 lazy 绑定。
- 该程序没有堆栈在内存中的可执行区域。
- GOT 表的重新定位被设置为只读。
- 没有程序片段设置所有三个读取、写入和执行权限。
- 没有针对可执行代码重新定位。
- 在运行时查找共享库的 runpath 信息仅包含在 /usr 中根目录。
- 程序在启用了 **annobin** 备注的情况下编译。
- 程序在启用了 **-fstack-protector-strong** 选项的情况下被编译。
- 程序使用 **-D_FORTIFY_SOURCE=2** 编译。
- 程序使用 **-D_GLIBCXX_ASSERTIONS** 编译。
- 程序被编译时启用了 **-fexceptions**。
- 程序在启用了 **-fstack-clash-protection** 的情况下被编译。
- 程序于 **-O2** 或更高版本编译。
- 程序没有处于写入状态的重新定位状态。
- 动态可执行文件具有动态片段。
- 共享库使用 **-fPIC** 或 **-fPIE** 编译。
- 动态可执行文件使用 **-fPIE** 编译并通过 **-pie** 链接。

- 如果可用，则使用 **-fcf-protection=full** 选项。
- 如果可用，则使用 **-mbranch-protection** 选项。
- 如果可用，则使用 **-mstackrealign** 选项。

4.6.2.4.4.2. 禁用强化检查程序

下面的部分论述了如何禁用强化检查程序。

步骤

- 要在没有强化检查程序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*。

4.6.3. 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，您可以删除冗余的 **annobin** 备注。要删除冗余的 **annobin** 注释，请使用 **objcopy** 程序，这是 **binutils** 软件包的一部分。

步骤

- 要删除冗余的 **annobin** 备注，请使用：

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

用文件名替换 *file-name*。

4.6.4. GCC Toolset 12 中的 annobin 的具体内容

在某些情况下，由于 GCC Toolset 12 中的 **annobin** 和 **gcc** 之间的同步问题，您的编译可能会失败，并显示类似如下的错误消息：

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

要临时解决这个问题，请从 **annobin.so** 文件中创建一个符号链接到 **gcc-annobin.so** 文件中：

```
# cd /opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin
# ln -s annobin.so gcc-annobin.so
```

使用您系统中使用的构架替换 *architecture* ：

- **aarch64**
- **i686**
- **ppc64le**

- **s390x**
- **x86_64**

第 5 章 补充主题

5.1. 编译器和开发工具中的兼容性破坏更改

非常量 PTHREAD_STACK_MIN、MINSIGSTKSZ 和 SIGSTKSZ 宏

为了更好地支持需要一个用于可扩展向量寄存器的变量堆栈大小、**PTHREAD_STACK_MIN**、**MINSIGSTKSZ** 和 **SIGSTKSZ** 宏的常量值已变为一个非常量值，如 **sysconf** 调用。

您不再以将 **PTHREAD_STACK_MIN**、**MINSIGSTKSZ** 和 **SIGSTKSZ** 宏视为像常量值的方式使用它们。**PTHREAD_STACK_MIN**、**MINSIGSTKSZ** 和 **SIGSTKSZ** 宏返回的值现在是长数据类型，并可能在与无符号值（如 **size_t** ）进行比较时产生编译器警告。

库合并到 libc.so.6

有了这个更新，以下库已合并到 **libc** 库中，以提供更顺畅的原位升级体验，支持进程在任何时候安全使用线程，并简化内部实现：

- **libpthread**
- **libdl**
- **libutil**
- **libanl**

另外，**libresolv** 库的一部分已移到 libc 中，以支持将名称切换服务(NSS)文件和域名系统(DNS)插件直接移到 **libc** 库中。NSS 文件和 DNS 插件现在直接构建到 **libc** 库中，并可在升级期间使用或跨 **chroot** 或容器边界使用。它们在 **chroot** 或容器边界的使用，支持从这些源安全地查询身份管理(IdM)数据。

zdump 工具的新位置

/usr/bin/zdump 是 **zdump** 工具的新位置。

弃用 sys_siglist、_sys_siglist 和 sys_sigabbrev 符号

sys_siglist、**_sys_siglist** 和 **sys_sigabbrev** 符号仅被导出为兼容性符号，以支持旧的二进制文件。所有程序都应该使用 **strsignal** 符号。

使用 **sys_siglist**、**_sys_siglist** 和 **sys_sigabbrev** 符号会产生问题，如复制重新定位和对阵列访问没有显式绑定检查的容易出错的应用程序二进制接口(ABI)。

这个更改可能会影响从源构建某些软件包。要修复问题，请重写程序以使用 **strsignal** 符号。例如：

```
#include <signal.h>
#include <stdio.h>

static const char *strsig (int sig)
{
  return sys_siglist[sig];
}

int main (int argc, char *argv[])
{
```

```
  printf ("%s\n", strsig (SIGINT));
  return 0;
}
```

应该调整为：

```
#include <signal.h>
#include <stdio.h>
#include <string.h>

static const char *strsig (int sig)
{
  return strsignal(sig);
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strsig (SIGINT));
  return 0;
}
```

或者，使用 **glibc-2.32** GNU 扩展 **sigabbrev_np** 或 **sigdescr_np** ：

```
#define _GNU_SOURCE
#include <signal.h>
#include <stdio.h>
#include <string.h>

static const char *strsig (int sig)
{
  const char *r = sigdescr_np (sig);
  return r == NULL ? "Unknown signal" : r;
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strsig (SIGINT));
  printf ("%s\n", strsig (-1));
  return 0;
}
```

这两个扩展都是 async-signal-safe 和 multithread-safe。

弃用 sys_errlist、_sys_errlist、sys_nerr 和 _sys_nerr 符号

sys_errlist、**_sys_errlist**、**sys_nerr** 和 **_sys_nerr** 符号仅被导出兼容性符号，以支持旧的二进制文件。所有程序都应该使用 **strerror** 或 **strerror_r** 符号。

使用 **sys_errlist**、**_sys_errlist**、**sys_nerr** 和 **_sys_nerr** 符号会导致问题，如复制重新定位和对阵列访问没有显式绑定检查的易出问题的 ABI。

这个更改可能会影响从源构建某些软件包。要解决这个问题，请使用 **strerror** 或 **strerror_r** 符号重写程序。例如：

```
#include <stdio.h>
```

```
#include <errno.h>

static const char *strerr (int err)
{
  if (err < 0 || err > sys_nerr)
    return "Unknown";
  return sys_errlist[err];
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

应该调整为：

```
#include <stdio.h>
#include <errno.h>

static const char *strerr (int err)
{
  return strerror (err);
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

或者，使用 **glibc-2.32** GNU 扩展 **strerrorname_np** 或 **strerrordesc_np** ：

```
#define _GNU_SOURCE
#include <stdio.h>
#include <errno.h>
#include <string.h>

static const char *strerr (int err)
{
  const char *r = strerrordesc_np (err);
  return r == NULL ? "Unknown error" : r;
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

这两个扩展都是 async-signal-safe 和 multithread-safe。

用户空间内存分配器, malloc, 更改

mallwatch 和 **tr_break** 符号现已弃用，不再在 **mtrace** 函数中使用。您可以使用 GDB 中 **mtrace** 函数中的条件断点来实现类似的功能。

__morecore 和 **__after_morecore_hook malloc** hook 和默认实现 **__default_morecore** 已从 API 中删除。现有应用程序将继续链接到这些符号，但接口不再对 **malloc** 产生任何影响。

现在，默认在 C 主库中禁用了 **malloc** 中的调试功能，如 **MALLOC_CHECK_** 环境变量（或 **glibc.malloc.check** 可调整变量）、**mtrace** （） 和 **mcheck** （）。要使用这些功能，请预先加载新的 **libc_malloc_debug.so** 调试 DSO。

弃用的函数 **malloc_get_state** 和 **malloc_set_state** 已从核心 C 库移到 **libc_malloc_debug.so** 库。仍使用这些函数的传统应用程序现在必需使用 **LD_PRELOAD** 环境变量在其环境中预先加载 **libc_malloc_debug.so** 库。

弃用的内存分配 hook **__malloc_hook**、**__realloc_hook**、**__memalign_hook** 和 **__free_hook** 现在已从 API 中删除。存在兼容性符号来支持旧程序，但新应用程序可以不再链接到这些符号。这些 hook 不再对 **glibc** 功能有任何影响。调试 DSO **libc_malloc_debug.so** 的**malloc** 目前支持 hook，并可预先加载以恢复旧程序的功能。但是，这是一个过渡措施，可能会在以后的 GNU C 库发行版本中删除。您可以通过编写和预先加载您自己的 **malloc** 组成库来从这些 hook 中移植出去。

lazy 绑定失败终止了进程

如果在 **dlopen** 函数期间发生了 lazy 绑定失败，则在 ELF 构造器执行过程中，进程现在被终止。在以前的版本中，动态加载程序从 **dlopen** 返回 **NULL**，并显示 **dlerror** 消息中捕获的 lazy 绑定错误。通常，这是不安全的，因为无法在任意函数调用中重置堆栈。

弃用 stime 功能

stime 函数不再对新链接的二进制文件可用，其声明已从 **<time.h>** 中删除。对于设置系统时间的程序，使用 **clock_settime** 函数。

popen 和 system 函数不再运行 fork 处理程序

虽然可能违反 POSIX，但 **pthread_atfork** 文档中关于 **atfork** 处理程序的 POSIX 原理是在多线程进程中的 fork 调用后处理不一致的排它锁锁状态。在 **popen** 和 **system** 函数中，无法直接访问用户定义的排它锁。

C++ 标准库中已弃用的功能

- 如果使用小于字符串的当前容量的参数调用，**std::string::reserve (n)** 将不再减少字符串的容量。可以通过调用不带参数的 **reserve()**来减少字符串的容量，但该形式已弃用。应该改为使用等同的 **shrink_to_fit** （）。

- 非标准 **std::__is_nullptr_t** 类型特征已弃用。应该改为使用标准的 **std::is_null_pointer** 特征。