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，在 Red Hat Enterprise Linux 8 中开发和调试 C 和 C++ 应用程序

Last Updated: 2024-07-09

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，在 Red Hat Enterprise Linux 8 中开发和调试 C 和 C++ 应用程序

法律通告

摘要

使用 Red Hat Enterprise Linux 8 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. 单击顶部导航栏中的 **Create**。
3. 在 **Summary** 字段中输入描述性标题。
4. 在 **Description** 字段中输入您对改进的建议。包括到文档相关部分的链接。
5. 点对话框底部的 **Create**。

第 1 章 建立一个开发工作站

Red Hat Enterprise Linux 8 支持自定义应用程序的开发。要允许开发人员这样做，必须设置带有必要的工具和实用程序的系统。本章列出了开发的最常见用例和要安装的项目。

1.1. 先决条件

- 必须安装系统，包括图形环境和订阅。

1.2. 启用调试和资源存储库

Red Hat Enterprise Linux 的标准安装不会启用调试和资源存储库。这些存储库包含调试系统组件并测量其性能所需的信息。

步骤

- 启用源并调试信息软件包通道：

```
# 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) 部分会自动替换为您系统构架的匹配值：

架构名称	订阅价值
64 位 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

有效的版本控制对于所有多开发人员项目非常重要。Red Hat Enterprise Linux 附带了 Git，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
```

使用用来启动选定的文本编辑器的命令替换 *command*。

其他资源

- Git 的 Linux 手册页和教程：

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

请注意，许多 Git 命令都有自己的手册页。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* 书的线上版本提供了 Git 、其概念及用法的详细描述。
- Reference - Git 的 Linux 手册页的线上版本

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

- 必须启用调试和资源存储库。

流程

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

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

2. 安装包括 **clang** 编译器和 **lldb** 调试器的基于 LLVM 的工具链：

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

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

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

1.5. 设置以调试应用程序

Red Hat Enterprise Linux 提供多种调试和检测工具，来分析和故障排除内部应用程序行为。

先决条件

- 必须启用调试和资源存储库。

流程

1. 安装用于调试的工具：

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

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

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

3. 运行 SystemTap 助手脚本来设置环境。

```
# stap-prep
```

请注意，**stap-prep** 安装与当前 *运行的* 内核相关的软件包，其可能与实际安装的内核不同。要确保 **stap-prep** 是否安装了正确的 **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，并在需要时重启系统。

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

其他资源

- 第 3.1 节 “启用带有调试信息的调试”

1.6. 设置以测量应用程序的性能

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

先决条件

- 必须启用调试和资源存储库。

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 助手脚本来设置环境。

```
# stap-prep
```

请注意，**stap-prep** 安装与当前 *运行的* 内核相关的软件包，其可能与实际安装的内核不同。要确保 **stap-prep** 是否安装了正确的 **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，并在需要时重启系统。

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

2.1. 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。

2.1.1. 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

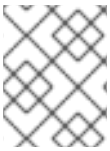
C 和 C++ 语言有三种形式的代码：

- 用 C 或 C++ 语言编写的 **源代码**，以纯文本文件形式呈现。
  文件通常使用如 **.c**,**.cc**,**.cpp**,**.h**,**.hpp**,**.i**,**.inc** 的扩展名。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
  ```

- **目标代码**，是使用 *编译器* *编译* 源代码创建的。这是一种中间形式。
  目标代码文件使用 **.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。共享目标(library)可执行文件使用 **.so** 文件名扩展名。



注意

也存在用于静态链接的库存档文件。这是使用 **.a** 文件名扩展名的目标代码的变体。不建议使用静态链接。请参阅 第 2.2.2 节 “静态和动态链接”。

GCC 中代码表单的处理

从源代码生成可执行代码分为两步执行，这需要不同的应用或工具。GCC 可用作编译器和链接器的智能驱动程序。这允许您将单个 **gcc** 命令用于任何必要的操作（编译和链接）。GCC 自动选择操作及其顺序：

1. 源文件编译成目标文件。
2. 目标文件和库被链接（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。这由输入的类型和请求的输出类型来决定的。

由于较大的项目需要一个通常为每个操作单独运行 GCC 的构建系统，因此最好始终考虑将编译和链接作为两个不同的操作，即使 GCC 可以同时执行这两项操作。

其他资源

- 第 2.1.2 节 “将源文件编译成目标代码”

- 第 2.1.6 节 “链接代码以创建可执行文件”
- 示例：使用 GCC 构建一个 C 程序（在一个步骤中进行编译和链接）
- 示例：使用 GCC 构建一个 C 程序（在两个步骤中进行编译和连接）

2.1.2. 将源文件编译成目标代码

要从源代码文件而不是可执行文件立即创建目标代码文件，必须指示 GCC 仅将目标代码文件创建为其输出。此操作代表了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。
- 在系统上安装了 GCC

流程

1. 进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
```

创建目标文件，其文件名反了映原始源代码文件： **source.c** 会生成 **source.o**。

注意

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

其他资源

- 第 2.1.5 节 “使用 GCC 强化代码的选项”
- 第 2.1.4 节 “使用 GCC 进行代码优化”
- 第 2.1.7 节 “Example:使用 GCC 构建一个 C 程序（在一个步骤中编译和链接）”

2.1.3. 使用 GCC 启用 C 和 C++ 应用程序的调试

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。要用它启用 C 和 C++ 应用程序的调试，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 启用调试信息的创建，请使用 **-g** 选项：

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

- 由编译器和链接器执行的优化可能会产生难以与原始源代码相关的可执行代码：变量可能被优化、循环被展开、操作被合并到周围操作中，等等。这会对调试产生负面影响。为提高调试体验，请考虑使用 **-Og** 选项设置优化。但是，更改优化级别会更改可执行代码，并可能会改变实际 d 行为，包括删除一些 bug。
- 要在调试信息中也包含宏定义，请使用 **-g3** 选项，而不是 **-g** 选项。

- **-fcompare-debug** GCC 选项测试使用带有调试信息的 GCC 编译的代码，而无需调试信息。如果生成的两个二进制文件相同，则测试通过。此测试确保可执行代码没有受到任何调试选项的影响，这进一步确保调试代码中没有隐藏的 bug。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

- 第 3.1 节 "启用带有调试信息的调试"
- 使用 GNU Compiler Collection(GCC)- 用于调试程序的选项
- 使用 GDB 进行调试 - 在独立文件中调试信息
- GCC 手册页：

```
$ man gcc
```

2.1.4. 使用 GCC 进行代码优化

一个程序可以转换为多个计算机指令序列。如果在编译过程中分配更多资源来分析代码，您可以获得更优的结果。

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

级别	描述
0	编译速度的优化 - 无代码优化（默认）。
1, 2, 3	优化以加快代码执行速度（数量越多，速度越快）。
s	文件大小的优化。
fast	与级别 **3** 设置一样，**fast** 忽略严格的标准合规性，以允许额外的优化。
g	调试体验的优化。

对于版本构建，请使用优化选项 **-O2**。

在开发过程中，**-Og** 选项在某些情况下对于调试程序或库很有用。由于某些 bug 只在某些优化级别上出现，因此请使用版本优化级别测试程序或库。

GCC 提供了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
```

2.1.5. 使用 GCC 强化代码的选项

当编译器将源代码转换为目标代码时，它可以添加各种检查来防止经常被利用的情况，并提高安全性。选择正确的编译器选项有助于生成更安全的程序和库，而无需更改源代码。

发行版本选项

对于以 Red Hat Enterprise Linux 为目标的开发人员，推荐使用以下选项列表：

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

- 对于程序，添加 **-fPIE** 和 **-pie** 位置独立可执行文件选项。
- 对于动态链接库，强制 **-fPIC** (Position Independent Code)选项会间接提高安全性。

开发选项

使用以下选项来检测开发过程中的安全漏洞：使用这些选项与发行版本的选项相结合：

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

其他资源

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

2.1.6. 链接代码以创建可执行文件

构建 C 或 C++ 应用程序时，链接是最后一步。链接将所有目标文件和库合并到一个可执行文件中。

先决条件

- 一个或多个目标文件。
- 系统上必须安装了 GCC

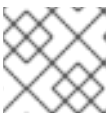
流程

1. 进到包含目标代码文件的目录。
2. 运行 **gcc** ：

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

从提供的目标文件和库创建一个名为 ***executable-file***的可执行文件。

要链接其他库，请在目标文件列表后添加所需的选项。如需更多信息，请参阅 第 2.2 节 “将库与 GCC 一起使用”。



注意

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

其他资源

- 第 2.1.7 节 “Example:使用 GCC 构建一个 C 程序（在一个步骤中编译和链接）”
- 第 2.2.2 节 “静态和动态链接”

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

此示例演示了构建一个简单示例 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

- 您必须了解如何使用 GCC。

流程

1. 创建一个目录 **hello-c**，并进到其中：

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

2. 创建包含以下内容的文件 **hello.c** ：

```
#include <stdio.h>

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

3. 使用 GCC 编译和链接代码：

```
$ 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!
```

其他资源

- 第 2.4.2 节 “Example:使用 Makefile 构建一个 C 程序”

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

此示例演示了构建一个简单示例 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

- 您必须了解如何使用 GCC。

流程

1. 创建一个目录 **hello-c**，并进到其中：

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

2. 创建包含以下内容的文件 **hello.c** ：

```
#include <stdio.h>

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

3. 使用 GCC 编译代码：

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

目标文件 **hello.o** 已创建。

4. 从目标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!
```

其他资源

- 第 2.4.2 节 “Example:使用 Makefile 构建一个 C 程序”

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

本例显示了构建最小 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。

流程

1. 创建 **hello-cpp** 目录，并进到其中：

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

2. 创建包含以下内容的文件 **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!
```

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

本例显示了构建最小 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。

流程

1. 创建 **hello-cpp** 目录，并进到其中：

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

2. 创建包含以下内容的文件 **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
```

目标文件 **hello.o** 已创建。

4. 从目标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!
```

2.2. 将库与 GCC 一起使用

了解在代码中使用库。

2.2.1. 库命名惯例

对库使用特殊文件名惯例：名为 **foo** 的库应该以文件 **lib*foo*.so** 或 **lib*foo*.a** 的形式存在。通过链接 GCC 的输入选项（而非输出选项）可自动理解这一惯例：

- 当链接到库时，只能将其名称 **foo** 和 **-l** 作为 **-l*foo*** 来指定库：

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

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

其他资源

- 第 2.3.2 节 “soname 机制”

2.2.2. 静态和动态链接

开发人员在使用完全编译的语言构建应用程序时可以选择使用静态或动态链接。务必要了解静态和动态链接之间的区别，特别是在 Red Hat Enterprise Linux 上使用 C 和 C++ 语言的上下文中。总之，红帽不建议对 Red Hat Enterprise Linux 的应用程序使用静态链接。

静态和动态链接的比较

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

可以通过多种方式比较动态和静态链接：

资源使用

静态链接会导致更大的可执行文件，其中包含更多代码。这些额外的代码来自不能在系统上的多个程序之间共享的库，这会增加运行时文件系统的使用率和内存的使用率。运行同一静态链接的程序的多个进程仍将共享代码。
另一方面，静态应用需要较少的运行时重定位，从而减少启动时间，并且需要较少的专用常驻集大小(RSS)内存。由于位置无关代码(PIC)引入的开销，为静态链接生成的代码比动态链接更高效。

安全性

可以更新提供 ABI 兼容的动态链接库，而无需根据这些库更改可执行文件。这对于由红帽提供的、作为 Red Hat Enterprise Linux 的一部分的库来说尤为重要，红帽提供安全更新。强烈反对对任何此类库的静态链接。

兼容性

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

警告

静态链接 GNU C 库的应用程序(**glibc**)仍然需要 **glibc** 作为动态库存在于系统中。此外，在应用程序的运行时可用的 **glibc** 的动态库变体在连接应用程序时必须与当前版本按位相同。因此，静态链接保证仅在构建可执行文件的系统上工作。

支持范围

红帽提供的大多数静态库都位于 *CodeReady Linux Builder* 通道中，不受红帽的支持。

功能

某些库，特别是 GNU C 库(**glibc**)，在静态链接时提供减少的功能。
例如，当静态链接时，**glibc** 不支持线程和在同一程序中对 **dlopen（）** 函数的任何形式的调用。

由于所列出的缺点，应该不惜一切代价避免静态链接，特别是对于整个应用程序以及 **glibc** 和 **libstdc++** 库。

静态链接的情况

在某些情况下静态链接可能是一种合理的选择，例如：

- 使用未对动态链接启用的库。
- 对于在空的 **chroot** 环境或容器中运行的代码，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。

其他资源

- Red Hat Enterprise Linux 8:应用程序兼容性指南
- *软件包清单* 中的 CodeReady Linux Builder 存储库 的描述

2.2.3. 将一个库与 GCC 一起使用

库是可在您的程序中重复使用的代码软件包。C 或 C++ 库由两个部分组成：

- 库代码
- 头文件

使用库编译代码

头文件描述库的接口：库中的函数和变量。编译代码需要头文件中的信息。

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

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

使用头文件目录的实际路径替换 *include_path*。

-I 选项可多次使用，以添加包含头文件的多个目录。查找头文件时，会按照它们在 **-I** 选项中出现的顺序搜索这些目录。

链接使用库的代码

链接可执行文件时，应用程序的目标代码和库的二进制代码都必须提供。静态和动态库的代码以不同的格式存在：

- 静态库作为存档文件提供。它们包含一组目标文件。存档文件具有文件扩展名 **.a**。
- 动态库作为共享目标提供。它们是一种可执行文件的形式。共享目标具有文件扩展名 **.so**。

要告诉 GCC 库的存档或共享目标文件的位置，请使用 **-L** 选项：

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

使用库目录的实际路径替换 *library_path*。

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

选项的顺序很重要：GCC 不能链接库 **foo**，除非其知道此库的目录。因此，在使用 **-l** 选项链接库之前，请使用 **-L** 选项来指定库目录。

在一个步骤中编译和链接使用库的代码

当允许在一个 **gcc** 命令中编译并链接代码时，请同时对上述两种情况使用选项。

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

2.2.4. 将一个静态库与 GCC 一起使用

静态库作为包含目标文件的存档提供。链接后，它们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 2.2.2 节 “静态和动态链接”。仅在需要时才使用静态链接，特别是对红帽提供的库。

先决条件

- GCC 必须安装在您的系统上。
- 您必须了解静态和动态链接。
- 您有一组组成有效程序的源或目标文件，需要一些静态库 **foo** ，但没有其他库。
- **foo** 库作为 **libfoo.a** 文件提供，对于动态链接，不提供文件 **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。以下步骤仅适用于*没有*为动态链接启用的库。请参阅 第 2.2.2 节 “静态和动态链接”。

流程

要从源和目标文件链接程序，请添加静态链接库 **foo**，该库可作为 **libfoo.a** 文件找到：

1. 进到包含您代码的目录。
2. 编译带有 **foo** 库的头的程序源文件：

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

使用包含 **foo** 库的头文件的目录的路径替换 *header_path*。

3. 将程序与 **foo** 库链接：

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

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

4. 要稍后运行该程序，只需：

```
$ ./program
```

警告

与静态链接有关的 **-static** GCC 选项禁止所有动态链接。相反，请使用 **-Wl、-Bstatic** 和 **-Wl,-Bdynamic** 选项更精确地控制链接器行为。请参阅 第 2.2.6 节 “将静态库和动态库与 GCC 一起使用”。

2.2.5. 将一个动态库与 GCC 一起使用

动态库作为独立的可执行文件提供，在链接时和运行时需要。它们独立于您应用程序的可执行文件。

先决条件

- GCC 必须安装在系统上。
- 组成有效程序的一组源或目标文件需要一些动态库 **foo**，但不需要其他库。
- **foo** 库必须作为文件 *libfoo.so* 提供。

将程序与动态库链接

要将程序与动态库 **foo** 链接：

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

当程序链接了动态库时，生成的程序必须总是在运行时加载库。定位库有两个选项：

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

使用存储在可执行文件中的 rpath 值

在其被链接时，**rpath** 是一个作为可执行文件的一部分保存的特殊值。之后，从可执行文件加载程序时，运行时链接器将使用 **rpath** 值来定位库文件。

与 GCC 链接时，要将路径 *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=** 选项中的逗号后面添加空格。

要稍后运行程序：

```
$ ./program
```

使用 LD_LIBRARY_PATH 环境变量

如果在程序的可执行文件中没有找到 **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目标所在的路径。

要运行没有 **rpath** 设置的程序，库要存在于路径 *library_path* 中：

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

省略 **rpath** 值提供了灵活性，但每次程序运行时，都需要设置 **LD_LIBRARY_PATH** 变量。

将库放入默认目录

运行时链接器配置指定多个目录来作为动态库文件的默认位置。要使用此默认行为，请将库复制到合适的目录中。

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

- 动态链接器的 Linux 手册页：

  ```
  $ man ld.so
  ```

- **/etc/ld.so.conf** 配置文件的内容：

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

- 动态链接器识别的库的报告，无需额外配置，其包括目录：

```
$ ldconfig -v
```

2.2.6. 将静态库和动态库与 GCC 一起使用

有时需要静态链接一些库，有时需要动态链接一些库。这种情况带来了一些挑战。

先决条件

- 了解静态和动态链接

简介

gcc 识别动态和静态库。遇到 **-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目标（一个 **.so** 文件），然后查找包含库的静态版本的存档文件(**.a**)。因此，这个搜索可能会导致以下情况：

- 只找到了共享目标，gcc 会动态链接它。
- 只找到了归档，gcc 会静态链接它。
- 共享目标和存档都都找到了，默认情况下，gcc 会选择对共享目标的动态链接。
- 共享目标和存档都未找到，链接失败。

由于这些规则，选择用于链接的库的静态或动态版本的最佳方法是让 gcc 只找到该版本。在指定 **-L*path*** 选项时，可以使用或省略包含库版本的目录，来在某种程度上控制它。

此外，由于动态链接是默认的，因此链接被明确指定的唯一情形是存在两个版本的库都应被静态链接时。有两种可能的解决方案：

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

通过文件指定静态库

通常，使用 **-l*foo*** 选项指示 gcc 链接到 foo 库。但是，可以指定包含库的 **lib*foo*.a** 文件的全路径：

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

从文件扩展名 **.a**，gcc 将理解为这是一个与程序链接的库。但是，指定库文件的全整路径是一个不太灵活的方法。

使用 -Wl 选项

gcc 选项 **-Wl** 是一个将选项传给底层链接器的特殊选项。此选项的语法与其他 gcc 选项不同。**Wl** 选项后跟一个以逗号分隔的链接选项列表，而其他 gcc 选项则需要以空格分隔的选项列表。

gcc 使用的 ld 链接器提供选项 **-Bstatic** 和 **-Bdynamic**，来指定此选项后面的库是否应分别被静态链接或动态链接。在将 **-Bstatic** 和库传给链路器后，必须为以下使用 **-Bdynamic** 选项动态链接的库手动恢复默认的动态链接行为：

要链接程序，请静态链接库**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 的文档 - 2.1 命令行选项

2.3. 使用 GCC 创建库

了解创建库以及 Linux 操作系统用于库的必要概念的步骤。

2.3.1. 库命名惯例

对库使用特殊文件名惯例：名为 **foo** 的库应该以文件 **lib*foo*.so** 或 **lib*foo*.a** 的形式存在。通过链接 GCC 的输入选项（而非输出选项）可自动理解这一惯例：

- 当链接到库时，只能将其名称 **foo** 和 **-l** 作为 **-l*foo*** 来指定库：

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

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

其他资源

- 第 2.3.2 节 “soname 机制”

2.3.2. soname 机制

动态加载的库（共享对象）使用一个名为 *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（共享目标）作为一个独立的可执行文件存在。这使得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：

- 库的实际版本的标识
- 需要存在同一库的多个版本
- 表示多个版本中每个版本的 ABI 兼容性

soname 机制

为了解决这个问题，Linux 使用一种称为 soname 的机制。

foo 库版本 *X.Y* 与版本号为 *X* 的其他版本 ABI 兼容。保持兼容性的次更改会增加数字 *Y*。破坏兼容性的主更改会增加数字 *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，soname 是使用值 **libfoo.so.x** 记录的，以表示兼容性。

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

当运行时动态链接器在运行前链接应用程序时，它会从应用的可执行文件中读取 soname。该 soname 是 **libfoo.so.*x***。具有此名称的符号链接必须存在，指向实际的库文件。这允许加载库，而不考虑版本的 *Y* 组件，因为 soname 不会改变。

注意

版本号 *Y* 组件号不仅限于一个数字。此外，一些库将其版本编码到其名称中。

从文件中读取 soname

要显示库文件 **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*。

2.3.3. 使用 GCC 创建动态库

动态链接的库（共享目标）允许：

- 通过代码重用来节约资源
- 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 必须安装在系统上。
- 您必须有库的源代码。

流程

1. 进到有库源文件的目录。
2. 使用位置独立代码选项 **-fPIC** 将每个源文件编译成目标文件：

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

   目标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. 链接目标文件的共享库：

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

使用的主版本号是 X，次版本号是 Y。

4. 将 ***libfoo.so.x.y*** 文件复制到合适的位置，其中系统的动态链接器可以找到它。在 Red Hat Enterprise Linux 中，库的目录是 **/usr/lib64** ：

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

请注意，您需要 root 权限才能操作此目录中的文件。

5. 为 soname 机制创建符号链接结构：

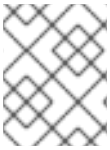
```
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

其他资源

- Linux 文档项目 - 程序库 HOWTO - 3。共享库

2.3.4. 使用 GCC 和 ar 创建静态库

通过将目标文件转换为特殊类型的存档文件，可以创建用于静态链接的库。



注意

出于安全原因，红帽不建议使用静态链接。只在需要时才使用静态链接，特别是红帽提供的库。详情请查看 第 2.2.2 节 “静态和动态链接”。

先决条件

- 系统上必须安装了 GCC 和 binutils 。
- 您必须了解静态和动态链接。
- 提供了要作为库共享的函数的源文件。

流程

1. 使用 GCC 创建中间目标文件。

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

如果需要，可附加更多源文件。生成的目标文件共享文件名，但使用 **.o** 文件扩展名。

2. 使用 **binutils** 软件包中的 **ar** 工具将目标文件转换为静态库（存档）。

```
$ 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合适的目录。

5. 与库链接时，GCC 将自动从 **.a** 文件扩展名中识别出库是一个用于静态链接的存档。

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

其他资源

- *ar(1)* 的 Linux 手册页：

```
$ man ar
```

2.4. 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。

2.4.1. GNU make 和 Makefile 概述

要从特定项目的源文件创建一个可用的表单（通常是可执行文件），请执行几个必要的步骤。记录操作及其顺序，以便稍后可重复这些操作。

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

先决条件

- 了解编译和链接的概念

GNU make

GNU **make** 读取 Makefile，其中包含描述构建过程的说明。Makefile 包含多个 *规则*，它们描述了满足带有特定操作（*recipe*）的某一条件（*target*）的方法。规则可以在层次上依赖于另一条规则。

运行不带任何选项的 **make** 可使其在当前的目录中查找 Makefile，并尝试到达默认目标。实际的 Makefile 文件名可以是 **Makefile**、**makefile** 和 **GNUmakefile** 中的一个。默认目标由 Makefile 内容决定。

Makefile 详情

makefile 使用相对简单的语法来定义 *变量* 和 *规则*，这些变量和规则由一个 *target* 和一个 *recipe* 组成。目标指定执行规则后的输出是什么。带有 recipe 的行必须以 TAB 字符开头。

通常，Makefile 包含用于编译源文件的规则、用于链接生成的目标文件的规则，以及充当层次结构顶部的入口点的目标。

请考虑以下用来构建由单个文件 **hello.c** 组成的 C 程序的 **Makefile**。

```
all: hello

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

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

本例演示了要到达目标 **all**，需要文件 **hello**。要获得 **hello**，您需要 **hello.o**（由 **gcc** 链接），后者是从 **hello.c** 创建的（由 **gcc**编译）。

目标 **all** 是默认目标，因为它是不是以句点(.)开头的第一个目标。当当前目录包含这个 **Makefile** 时，运行不带任何参数的 **make** 与运行 **make all** 一样。

典型的 makefile

更为典型的 Makefile 使用变量来概括步骤，并添加一个目标"clean" - 删除除源文件以外的任何内容。

```
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 添加更多源文件只需要将它们添加到定义了 SOURCE 变量的行中。

其他资源

- GNU make：简介 - 2 Makefile 简介
- 第 2.1 节 “使用 GCC 构建代码”

2.4.2. Example:使用 Makefile 构建一个 C 程序

按照本示例中的步骤，使用 Makefile 构建一个示例 C 程序。

先决条件

- 您必须理解 Makefile 的概念和 **make**。

流程

1. 创建一个 **hellomake** 目录，并进到此目录：

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

2. 创建包含以下内容的文件 **hello.c** ：

```
#include <stdio.h>

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

3. 创建包含以下内容的文件 **Makefile** ：

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

all: $(SOURCE) $(EXE)

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

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

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

重要

Makefile recipe 行必须以制表符字符开头！从文档中复制上面的文本时，剪切和粘贴过程可能会粘贴空格而不是制表符。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
```

这将创建一个可执行文件 **hello**。

5. 运行可执行文件 **hello** ：

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

6. 运行 Makefile 目标 **clean** 以删除创建的文件：

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

其他资源

- 第 2.1.7 节 “Example:使用 GCC 构建一个 C 程序（在一个步骤中编译和链接）”

- 第 2.1.9 节 “Example:使用 GCC 构建一个 C++ 程序（在一个步骤中编译和链接）”

2.4.3. make 的文档资源

有关 **make** 的更多信息，请参阅以下列出的资源。

安装的文档

- 使用 **man** 和 **info** 工具查看安装在系统上的手册页和信息页：

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

在线文档

- 由自由软件基金会主办的 GNU Make 手册

2.5. 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。

2.5.1. 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：

- 添加了大量常规优化，如别名分析、向量改进、相同代码折叠、流程间分析、存储合并优化传递等。
- 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，C、C++ 和 Fortran 编译器现在支持 OpenMP 4.0 规范的卸载功能。
- 为静态检测某些可能的编程错误增加了新的警告和改进的诊断。
- 源位置现在作为范围而不是点进行跟踪，这允许更丰富的诊断。编译器现在提供"fix-it"提示，建议可能的代码修改。添加了拼写检查器以提供替代名称并容易检测拼写错误。

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 2.5.2 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：

- 添加了多个 Intel AVX-512 架构、多个微架构和 Intel Software Guard 扩展(SGX)的新架构特定选项。
- 代码生成可以针对 64 位 ARM 架构 LSE 扩展、ARMv8.2-A 16 位浮点扩展(FPE)和 ARMv8.2-A、ARMv8.3-A 及 ARMv8.4-A 架构版本。
- 已修复了 ARM 和 64 位 ARM 架构上的 **-march=native** 选项的处理。
- 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++ 编译器现在实现了带有许多新功能的 C++14 标准，如，变量模板、非静态数据成员初始化器、扩展的 **constexpr** 规范器、大小的取消分配函数、通用 lambda、变量长度数组、数字分隔器等。
- 改进了对 C 语言标准 C11 的支持：ISO C11 原子、通用选择和线程本地存储现已提供。
- 新的 **__auto_type** GNU C 扩展提供了 C 语言中 C++11 **auto** 关键字的功能子集。
- 由 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** 选项的作用相同。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。传递或返回一个仅带有已删除副本和移动构造器的类，现在使用与带有非平凡副本或移动构造器的类相同的调用惯例。
- C++11 **alignof** 运算符返回的值已被修正，以匹配 C **_Alignof** 运算符并返回最小对齐。要找到首选对齐，请使用 GNU 扩展 **__alignof__**。
- 适用于 Fortran 语言代码的 **libgfortran** 库的主要版本已变为 5。
- 删除了对 Ada(GNAT)、GCC Go 和目标 C/C++ 语言的支持。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

2.5.2. RHEL 8 中 GCC 的安全性增强

以下是与 Red Hat Enterprise Linux 7.0 发布以来与安全性相关的 GCC 的更改。

新警告

添加了这些警告选项：

选项	显示警告信息
-Wstringop-truncation	调用有界字符串操作函数，如 **strncat**、**strncpy** 和 **stpncpy**，它们可能会截断复制的字符串或使目的地保持不变。
-Wclass-memaccess	原始功能（如 **memcpy** 或 **realloc**）可能会以不安全的方式处理类型为非 trivial 类的对象。 警告有助于检测绕过用户定义的构造器或复制-赋值运算符、损坏虚拟表指针、常量限定类型或引用的数据成员或成员指针的调用。该警告还会检测到可绕过数据成员的访问控制的调用。
-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。也适用于通过将两个参数相乘来指定分配的函数，也适用于使用属性 **alloc_size** 修饰的任何函数。
-Walloc-zero	调用内存分配功能,试图分配零内存。也适用于通过将两个参数相乘来指定分配的函数，也适用于使用属性 **alloc_size** 修饰的任何函数。
-Walloca	所有对 **alloca** 功能的调。
-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*	对格式化输出函数的 **sprintf** 系列调用中的一定和可能的缓冲区溢出。有关 *level* 值的详情和说明，请参阅 *gcc(1)* 手册页。
-Wformat-truncation=*level*	对格式化输出函数的 **snprintf** 系列调用中的一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** 的调用中的缓冲区溢出。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，以检测更多来自边界外数组索引和指针偏移的实例。例如，检测到对灵活的数组成员和字符串文字的负索引或过度索引。
- GCC 7 中引入的 **-Wrestrict** 选项已被增强，可以通过对标准内存和字符串操作函数（如 **memcpy** 和 **strcpy** ）的限制限定参数来检测更多对目标的重叠访问的实例。

- 改进了 **-Wnonnull** 选项，以检测将空指针传给期望非空参数（以属性 **nonnull** 修饰）的函数的更广泛的情况。

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。
-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。这可启用 **-fsanitize=bounds**，以及灵活的数组成员式数组的检测。
-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 **__builtin_clz** 或 **__builtin_ctz** 前缀内置的无效参数。包括 **-fsanitize=undefined=undefined** 的检查。
-fsanitize=pointer-overflow	为指针嵌套执行 cheap run-time 测试。包括 **-fsanitize=undefined=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	对指向不同内存对象的指针减法的警告。
-fsanitize-address-use-after-scope	清理在定义了变量的作用域之后其地址被占用的变量。

其他清理程序和工具

- 添加了选项 **-fstack-clash-protection**，以便在静态或动态分配堆栈空间时插入探测，来可靠地检测堆栈溢出，从而减少依赖于操作系统提供的堆栈保护页的攻击向量。

- 添加了一个新的选项 **-fcf-protection=[full|branch|return|none]** 来执行代码检测，并通过检查控制流传输指令的目标地址（如间接函数调用、函数返回、间接跳过）是否有效来来提高程序安全性。

其它资源

- 有关提供给以上某些选项的值的详情和解释，请参阅 *gcc(1)* 手册页：

```
$ man gcc
```

2.5.3. RHEL 8 中 GCC 的兼容性破坏的变化

std::string 和 std::list 中 C++ ABI 的更改
std::string 的应用程序二进制接口(ABI)和 **libstdc++** 库中的**std::list** 类在 RHEL 7(GCC 4.8)和 RHEL 8(GCC 8)之间的变化符合 C++11 标准。**libstdc++** 库同时支持旧的和新的 ABI，但些其他一些 C++ 系统库不支持。因此，动态链接这些库的应用程序需要重建。这会影响所有 C++ 标准模式，包括 C++98。它还影响了使用 RHEL 7 的 Red Hat Developer Toolset 编译器构建的应用程序，这些应用程序保留了旧的 ABI，以保持与系统库的兼容性。

GCC 不再构建 Ada、Go 和 Objective C/C++ 代码
在 Ada(GNAT)、GCC Go 和目标 C/C++ 语言中构建代码的能力已从 GCC 编译器中删除。

要构建 Go 代码，请使用 Go Toolset。

第 3 章 调试应用程序

调试应用程序是一个非常广泛的话题。本节为开发人员提供了在各种情况下进行调试的最通用的技术。

3.1. 启用带有调试信息的调试

要调试应用和库，需要调试信息。以下章节描述了如何获取此信息。

3.1.1. 调试信息

在调试任何可执行代码时，两种类型的信息允许程序员通过工具和扩展来理解二进制代码：

- 源代码文本
- 源代码文本如何与二进制代码相关的描述

此类信息称为调试信息。

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

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、功能较弱的格式，有时与 UNIX 一起使用。红帽不鼓励使用它。GCC 和 GDB 仅在尽最大努力的基础上提供 STABS 生产和消费。一些其他工具，如 Valgrind 和 **elfutils**，不适用于 STABS。

其他资源

- DWARF 调试标准

3.1.2. 使用 GCC 启用 C 和 C++ 应用程序的调试

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。要用它启用 C 和 C++ 应用程序的调试，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 启用调试信息的创建，请使用 **-g** 选项：

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

- 由编译器和链接器执行的优化可能会产生难以与原始源代码相关的可执行代码：变量可能被优化、循环被展开、操作被合并到周围操作中，等等。这会对调试产生负面影响。为提高调试体验，请考虑使用 **-Og** 选项设置优化。但是，更改优化级别会更改可执行代码，并可能会改变实际 d 行为，包括删除一些 bug。
- 要在调试信息中也包含宏定义，请使用 **-g3** 选项，而不是 **-g** 选项。
- **-fcompare-debug** GCC 选项测试使用带有调试信息的 GCC 编译的代码，而无需调试信息。如果生成的两个二进制文件相同，则测试通过。此测试确保可执行代码没有受到任何调试选项的影响，这进一步确保调试代码中没有隐藏的 bug。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

- 第 3.1 节 “启用带有调试信息的调试”
- 使用 GNU Compiler Collection(GCC)- 用于调试程序的选项
- 使用 GDB 进行调试 - 在独立文件中调试信息
- GCC 手册页：

```
$ man gcc
```

3.1.3. debuginfo 和 debugsource 软件包

debuginfo 和 **debugsource** 软件包包含程序和库的调试信息和调试源代码。对于安装在 Red Hat Enterprise Linux 存储库的软件包中的应用程序和库，您可以从其它渠道获得单独的**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 软件包提供为二进制代码功能提供的人类可读名称所需的调试信息。这些软件包包含 **.debug** 文件，其中包含 DWARF 调试信息。这些文件被安装到 **/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。分别安装了 **debuginfo** 和 **debugsource** 软件包后，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。源代码文件被安装到 **/usr/src/debug** 目录中。

与 RHEL 7 的不同

在Red Hat Enterprise Linux 7 中，**debuginfo** 软件包包含这两种类型的信息。Red Hat Enterprise Linux 8 将调试信息所需的源代码数据从 **debuginfo** 软件包分割为单独的 **debugsource** 软件包。

软件包名称

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

- 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**

其他资源

- 第 3.1.1 节 “调试信息”
- 第 1.2 节 “启用调试和资源存储库”

3.1.4. 使用 GDB 获取应用程序或库的 debuginfo 软件包

需要调试信息来调试代码。对于从软件包安装的代码，GNU Debugger(GDB)自动识别缺少的调试信息，解析软件包名称并提供有关如何获取软件包的具体建议。

先决条件

- 要调试的应用程序或库必须安装在系统上。
- GDB 和 **debuginfo-install** 工具必须安装在系统上。
- 必须在系统中配置和启用提供 **debuginfo** 和 **debugsource** 软件包的软件仓库。详情请参阅 启用调试和源存储库。

流程

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

```
$ gdb -q /bin/ls
Reading symbols from /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更改摘要、要求确认，并在确认后，下载并安装所有必需的文件。

4. 如果 GDB 没有推荐 **debuginfo** 软件包，请按照 第 3.1.5 节 “手动为应用程序或库获取 debuginfo 软件包” 中描述的流程操作。

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. 手动为应用程序或库获取 debuginfo 软件包

您可以通过定位可执行文件，然后查找安装它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 来确定安装的软件包。只有在 GDB 无法建议要安装的软件包时，才使用此手动流程。

先决条件

- 应用程序或库必须安装在系统上。
- 应用程序或库是从软件包安装的。
- **debuginfo-install** 工具必须在系统上可用。

- 提供 **debuginfo** 软件包的渠道必须在系统上配置了且启用了。

流程

1. 查找应用程序或库的可执行文件。

   a. 使用 **which** 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
   ```

   如果调试的最初原因包含错误消息，请选择库的文件名中与错误消息中提及的具有相同的额外数字的结果。若有疑问，请使用文件名不包含其他数字的结果来尝试遵循该流程的其余部分。

注意

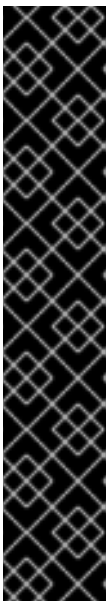
locate 命令由 **mlocate** 软件包提供。要安装并启用它：

```
# yum 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
   ```

   输出以 *name*:*epoch*-*version*.*release*.*architecture* 格式提供已安装软件包的详细信息。



重要

如果此步骤没有生成任何结果，则无法确定哪一个软件包提供了二进制文件。有几个可能的情况：

- 文件是从 *current* 配置中包管理工具不知道的包安装的。
- 文件从本地下载和手动安装的包中安装的。在这种情况下，自动确定合适的 **debuginfo** 软件包是不可能的。
- 您的软件包管理工具配置错误。
- 文件不是从任何软件包安装的。在这种情况下，没有对应的 **debuginfo** 软件包存在。

因为后续步骤依赖于此，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不在本流程范围内。

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

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

要找出应用不能正常工作的原因，请控制其执行并使用调试器检查其内部状态。本节描述了如何对此任务使用 GNU Debugger(GDB)。

3.2.1. GNU 调试器(GDB)

Red Hat Enterprise Linux 包含 GNU 调试器(GDB)，允许您通过命令行用户界面调查程序内部发生了什么情况。

GDB 功能

单个 GDB 会话可以调试以下类型的程序：

- 多线程和分叉程序
- 同时有多个程序
- 远程机器上或带有 **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，您必须安装它们的 debuginfo 软件包。

3.2.2. 将 GDB 附加到进程

为了检查进程，必须将 GDB *附加到* 进程。

先决条件

- GDB 必须安装在系统上

使用 GDB 启动程序

当该程序没有作为进程运行时，使用 GDB 启动它：

```
$ gdb program
```

使用程序的文件名或路径替换 *program*。

设置 GDB 以开始程序的执行。您可以使用 **run** 命令在开始执行进程前设置断点和 **gdb** 环境。

将 GDB 附加到已在运行的进程

要将 GDB 附加到已作为进程运行的程序：

1. 使用 **ps** 命令找到进程 ID(*pid*)：

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

使用程序的文件名或路径替换 *program*。

2. 将 GDB 附加到此进程：

```
$ 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

要将已在运行的 GDB 附加到已在运行的程序：

1. 使用 **shell** GDB 命令运行 **ps** 命令，并找到程序的进程 ID(*pid*)：

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

使用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
```

其他资源

- 使用 GDB 进行调试 - 2.1 调用 GDB
- 使用 GDB 进行调试 - 4.7 调试已在运行的进程

3.2.3. 使用 GDB 逐步浏览程序代码

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

- 您必须有所需的调试信息：
  - 程序使用调试信息编译和构建，或者

- 相关的 debuginfo 软件包已安装

- GDB 必须附加到要调试的程序

单步调试代码的 GDB 命令

r (run)

开始程序的执行。如果使用任何参数执行 **run**，则这些参数将被传给可执行文件，如同程序已正常启动了一样。用户通常在设置断点后发出此命令。

开始

开始程序的执行，但会在程序主函数的开头停止。如果使用任何参数执行 **start**，则这些参数将被传给可执行文件，就像程序已正常启动了一样。

c （continue）

从当前状态继续程序的执行。程序的执行将继续，直到以下其中一个条件变为 true：

- 达到了断点。
- 满足指定的条件。
- 程序收到了信号。
- 出现了错误。
- 程序终止了。

n (next)

从当前状态继续程序的执行，直到达到当前源文件中的下一行代码。程序的执行将继续，直到以下其中一个条件变为 true：

- 达到了断点。
- 满足指定的条件。
- 程序收到了信号。
- 出现了错误。
- 程序终止了。

s （step）

step 命令还会在当前源文件中的每一行代码处停止执行。但是，如果执行当前在包含 **function call** 的源行处停止，则 GDB 会在输入 function call （而不是执行它）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）

恢复程序的执行，并在执行从函数返回时停止。程序的执行将继续，直到以下其中一个条件变为 true：

- 达到了断点。
- 满足指定的条件。

- 程序收到了信号。
- 出现了错误。
- 程序终止了。

q (quit)

终止执行并退出 GDB。

其他资源

- 第 3.2.5 节 “使用 GDB 断点在定义的代码位置停止执行”
- 使用 GDB 进行调试 - 启动程序
- 使用 GDB 进行调试 - 继续和步进

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做什么非常重要。GDB 提供了多个您可用来检查内部变量的命令。以下是这些命令中最有用的命令：

p （print）

显示给定的参数的值。通常，参数是任何复杂程度的变量的名称，从简单的单个值到结构。参数也可以是在当前语言中一个有效的表达式，包括程序变量和库函数的使用，或者在测试的程序中定义的函数。
可以使用 *pretty-printer* Python 或 Guile 脚本来扩展 GDB， 以便使用 **print** 命令来自定义对数据结构（如类、结构）的显示。

bt （backtrace）

显示用于到达当前执行点的函数调用链，或者显示在执行终止之前所使用的函数链。这对于调查原因不明的严重 bug（如分段错误）非常有用。
向 **backtrace** 命令中添加 **full** 选项也会显示本地变量。

可以使用 *frame filter* Python 脚本来扩展 GDB，以便使用 **bt** 和 **info frame** 命令自定义显示的数据。术语 *frame* 指的是与单个函数调用关联的数据。

info

info 命令是提供关于各种条目的信息一个通用命令。它使用指定要描述的条目的一个选项。

- **info args** 命令显示当前所选帧的函数调用的选项。
- **info locals** 命令显示当前选定的帧中的局部变量。

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

```
(gdb) help info
```

l （list）

显示源代码中程序停止的行。此命令仅在程序执行停止时才可用。虽然严格来说不是用来显示内部状态的命令，但 **list** 帮助用户了解在程序执行的下一步中内部状态将发生哪些变化。

其他资源

- GDB Python API -红帽开发者博客条目
- 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，仅调查代码的一小部分。断点是标记，告知 GDB 在代码的特定位置停止程序的执行。断点通常与源代码行关联。在这种情况下，放置断点需要指定源文件和行号。

- **放置断点** ：
  - 指定源代码 *文件的名称以及该文件中 的行* ：

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

  - 如果 *文件* 不存在，则使用当前执行点的源文件名称：

    ```
    (gdb) br line
    ```

  - 或者，使用函数名称将 breakpoint 放置到其启动时：

    ```
    (gdb) br function_name
    ```

- 在任务发生多次迭代后，程序可能会遇到错误。指定暂停执行的额外 **条件** ：

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

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

- 要 **检查** 所有断点和监控点的状态：

  ```
  (gdb) info br
  ```

- 要使用 **info br** 的输出中显示的 *number* **删除** 断点：

  ```
  (gdb) delete number
  ```

- 要在给定位置 **删除** 断点：

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

其他资源

- 使用 GDB 进行调试 - 断点、监视点和捕捉点

3.2.6. 使用 GDB 监视点停止在数据访问和更改时执行

在很多情况下，让程序执行直到某些数据改变了或被访问是有好处的。以下示例是最常见的用例。

先决条件

- 了解 GDB

在 GDB 中使用观察点

Watchpoints 是指示 GDB 停止执行程序的标记。Watchpoints 与数据相关联：放置观察点需要指定描述变量、多个变量或内存地址的表达式。

- 为数据 **更改** **放置** 监视点（写入）：

```
(gdb) watch expression
```

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

- 为 **数据访问** **放置** 监视点（读取）：

```
(gdb) rwatch expression
```

- 为 **任何** 数据访问 **放置** 监视点（读写）：

```
(gdb) awatch expression
```

- 要 **检查** 所有监视点和断点的状态：

```
(gdb) info br
```

- **删除** 观察点：

```
(gdb) delete num
```

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

其他资源

- 使用 GDB 调试 - 设置 Watchpoints

3.2.7. 使用 GDB 调试或线程程序

些程序使用分叉或线程来实现并行代码执行。调试多个同步执行路径需要特殊考虑。

先决条件

- 您必须了解进程分叉和线程的概念。

使用 GDB 调试程序

分叉是一种程序（**父**）创建自身（**子**）的一个独立副本的情况。使用以下设置和命令来影响 GDB 在发生分叉时执行的操作：

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

  设置 follow-fork-mode parent

  分叉后，调试父进程。这是默认值。

  设置 follow-fork-mode child

  分叉后，调试子进程。

显示 follow-fork-mode

显示 **follow-fork-mode** 的当前设置。

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

设置 detach-on-fork on

不遵循的进程（取决于 **follow-fork-mode 的值）将被**分离并独立运行。这是默认值。

设置 detach-on-fork off

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

显示 detach-on-fork

显示当前 **detach-on-fork 的设置**。

使用 GDB 调试线程程序

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

GDB 使用 *当前线程* 的概念。默认情况下，命令仅应用于当前线程。

info 线程

显示 **ID** 和 **gid** 编号的线程列表，指示当前的线程。

线程 *ID*

使用指定 **id** 作为当前线程设置线程。

线程应用 *ids 命令*

将 ***命令*** 应用到 ***ids*** 列出的所有线程。***ids*** 选项是空格分隔的线程 ID 列表。一个特殊值 **all** 将命令应用到所有线程。

如果 *条件*中断 *位置* 线程 *id*

在特定位置设置断点，并且仅针对线程编号 ***ID 具有 特定条件***。

watch *表达式* 线程 *ID*

设置仅用于线程编号 ***ID*** 的 ***表达式*** 定义的观察点。

命令&

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

中断

在后台暂停执行。

其他资源

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

3.3. 记录应用程序互动

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

3.3.1. 用于记录应用程序交互的工具

红帽企业 Linux 提供用于分析应用程序的交互的多种工具。

strace

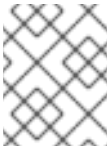
The **strace** 工具主要支持记录供应用使用的系统调用（内核功能）。

- The **strace** 工具可以提供有关调用的详细输出，因为 **strace** 知道底层内核代码来解读参数和结果。数字转换为对应的恒定名称，展开为标记列表的组合标志，指向字符数组的指针被解引用以提供实际字符串等。可能缺少对更新内核功能的支持。
- 您可以过滤跟踪的调用来减少捕获的数据量。
- 使用 **strace** 不需要任何特定设置，除了设置日志过滤器外。
- 使用 **strace** 跟踪应用代码导致应用的执行出现显著下降。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。此功能对于调试非常有用。

ltrace

ltrace 工具支持将应用的用户空间调用记录到共享对象（动态库）。

- **ltrace** 工具启用对任何库的追踪调用。
- 您可以过滤跟踪的调用来减少捕获的数据量。
- 使用 **ltrace** 不需要任何特定设置，除了设置日志过滤器外。
- **ltrace** 工具是轻量级且快速的，它提供了 **strace的** 替代选择：可以使用 **l trace** 跟踪库中的对应接口，而不是通过 trace 跟踪内核功能 **。**
- 因为 **ltrace** 不会处理一组已知的调用，如 **trace,** 所以它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。对 **ltrace** 输出的解释需要咨询输出中存在的库的实际接口声明。



注意

在 Red Hat Enterprise Linux 8 中，一个已知问题会阻止 **ltrace** 追踪系统可执行文件。此限制不适用于用户构建的可执行文件。

SystemTap

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

- 与使用 **strace** 和 **ltrace** 相比，编写日志意味着在初始设置阶段有更多工作。但是，该脚本功能将 SystemTap 的用处扩展至生成日志之外。
- SystemTap 通过创建和插入内核模块来工作。SystemTap 的使用效率更高，不会自行造成系统或应用执行的显著下降。
- SystemTap 附带了一组使用示例。

GDB

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

- 借助 GDB，可以将交互事件的捕获和后续执行路径的即时调试轻松结合在一起。
- 在其他工具最初识别问题的情况下，GDB 最适合分析对不常见的事件或单数事件的响应。在任何有频繁事件的情况下使用 GDB 都会变得效率低下甚至不可能。

其他资源

- SystemTap 入门
- Red Hat Developer Toolset 用户指南

3.3.2. 使用 strace 监控应用的系统调用

The **strace** 工具启用监控应用执行的系统（内核）调用。

先决条件

- 必须在系统中安装了 **strace**。

流程

1. 确定要监控的系统调用。
2. Start **strace** 并将其附加到 程序。
   - 如果您要监控的程序没有运行，请启动 **追踪** 并指定 *程序* ：

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

   - 如果程序已在运行，请查找其进程 ID(*pid*) **并附加至** 该程序：

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

   - 将 *call* 替换为要显示的系统调用。您可以多次使用 **-e trace=*call*** 选项。如果遗漏，**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分叉的进程或线程，请退出 **-f** 选项。
3. **strace** 工具显示应用程序发出的系统调用及其详细信息。
   在大多数情况下，如果未设置系统调用过滤器，应用及其库都会立即发出大量调用和 **strace** 输出。
4. 当程序退出时，The **strace** 工具会退出。
   要在 traced 程序退出前终止监控，请按 **Ctrl+C**。
   - If **strace** 启动程序，程序与 **strace** 一起终止。
   - 如果您 **将 trace 附加到** 已在运行的程序，程序会一起终止 。
5. 分析应用执行的系统调用列表。

- 当调用返回错误时，日志中会出现与资源访问或可用性相关的问题。
- 传递给系统调用和调用序列模式的值可让您了解应用程序的原因。
- 如果应用崩溃，重要信息或许位于日志的末尾。
- 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精确的过滤器，并重复该过程。

注意

益处是查看输出并将其保存到文件中。使用 **tee** 命令实现这一点：

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

其他资源

- The *strace(1)* 手册页：

```
$ man strace
```

- 我如何使用 strace 跟踪命令发出的系统调用？ - 知识库文章.
- Red Hat Developer Toolset 用户指南 - 第 strace

3.3.3. 使用 ltrace 监控应用程序的库功能调用

ltrace 工具支持监控应用程序对库中可用功能（共享对象）的调用。

注意

在 Red Hat Enterprise Linux 8 中，一个已知问题会阻止 **ltrace** 追踪系统可执行文件。此限制不适用于用户构建的可执行文件。

先决条件

- 必须在系统中安装了 **ltrace**。

流程

1. 识别感兴趣的库和功能（如果可能）。
2. 启动 **ltrace** 并将其附加到 程序。
   - 如果您要监控的程序没有运行，请启动 **ltrace** 并指定 *程序* ：

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

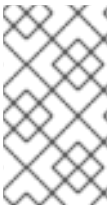
   - 如果程序已在运行，请查找其进程 ID(*pid*)并附加 **ltrace** 到其中：

```
$ ps -C program
(...)
$ ltrace -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 *function* 的函数名称。**e *函数*** 选项可多次使用。如果遗漏，**ltrace** 会显示对所有功能的调用。
  - 您可以使用 **-l *库*** 选项指定整个库，而不是指定函数。这个选项的行为与 **-e *函数*** 选项类似。
  - 如果您不想跟踪任何分叉的进程或线程，请退出 **-f** 选项。

  如需更多信息，请参阅 *ltrace*(1)_ 手册页。

3. **ltrace** 显示应用发出的库调用。
   在大多数情况下，如果未设置过滤器，应用会立即发出大量调用和 **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 enter **ctrl+C**。
   - 如果 **ltrace** 启动了程序，程序会一起使用 **ltrace** 终止。
   - 如果您将 **ltrace** 附加到已在运行的程序，程序会一起终止 **ltrace**。
5. 分析应用执行的库调用列表。
   - 如果应用崩溃，重要信息或许位于日志的末尾。
   - 输出中包含大量不必要的信息。但是，您可以构建一个更精确的过滤器并重复该过程。



注意

益处是查看输出并将其保存到文件中。使用 **tee** 命令实现这一点：

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

其他资源

- *ltrace(1)* 手册页：

```
$ man ltrace
```

- Red Hat Developer Toolset 用户指南 - 第 ltrace

3.3.4. 使用 SystemTap 监控应用的系统调用

SystemTap 工具启用为内核事件注册自定义事件处理程序。与 **strace** 工具相比，它更难使用，但更有效，并支持更复杂的处理逻辑。SystemTap 脚本名为 **trace.stp**，与 SystemTap 一同安装，并且使用 SystemTap 提供对 **strace** 功能的估测。

先决条件

- 系统必须安装 SystemTap 和相应的内核软件包。

流程

1. 查找您要监控的进程的进程 ID(*pid*)：

```
$ ps -aux
```

2. 使用 the **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，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，调用名称及其参数将打印到终端。

4. 当进程终止或按 **Ctrl+C** 时，脚本将退出。

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出现的各种情况下停止执行。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 断点的使用情况。
- GDB 必须附加到程序。

流程

1. 设置捕获点：

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

捕获系统调用的命令 设置一种特殊的断点类型，在程序执行系统调用时停止执行。

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

2. 开始执行 程序。

   - 如果程序还没有开始执行，请启动它：

```
(gdb) r
```

   - 如果程序执行被停止，恢复它：

```
(gdb) c
```

3. GDB 在程序执行任何指定系统调用后停止执行。

其他资源

- 第 3.2.4 节 “使用 GDB 显示程序内部值”
- 第 3.2.3 节 “使用 GDB 逐步浏览程序代码”

- 使用 GDB 进行调试 - 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 断点的使用情况。
- GDB 必须附加到程序。

流程

1. 设置捕获点：

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

命令 **捕获信号** 会设置一种特殊的断点类型，它会在程序收到信号时暂停执行。 ***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。
   - 如果程序还没有开始执行，请启动它：

```
(gdb) r
```

   - 如果程序执行被停止，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

- 第 3.2.4 节 “使用 GDB 显示程序内部值”
- 使用 GDB 逐步浏览程序代码
- 使用 GDB 调试 – 5.1.3 Setting Catchpoints

3.4. 调试崩溃应用程序

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

3.4.1. 内核转储：它们是什么以及如何使用它们

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

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

警告

某些限制可能会影响生成内核转储的功能。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，请设置核心转储保存并添加系统信息。

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，描述之前是否存在这些设置以及之前的值。如果需要，这将允许您稍后撤销这些更改。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
   ```

   若要反转这一更改，可运行值为 **0** 的 命令，而不运行 ***无限*** 值。

4. 安装 **sos** 软件包，它提供用于收集系统信息的 **sosreport** 工具：

   ```
   # yum install sos
   ```

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

6. 创建 SOS 报告以提供有关系统的更多信息：

   ```
   # sosreport
   ```

   这将创建一个包含您系统信息的 **.tar** 存档，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
```

如果应用多次崩溃，第一个命令的输出会列出更多捕获的核心转储。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. 将核心转储和 SOS 报告传送到要进行调试的计算机。如果已知，也传输可执行文件。

重要

当可执行文件未知时，core 文件的后续分析会识别该文件。

9. 可选：传输后删除核心转储和 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
- 如何在应用程序崩溃或分段故障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

3.4.3. 使用内核转储检查应用程序崩溃状态

先决条件

- 您必须从发生崩溃的系统中有一个核心转储文件和 sosreport。
- 在您的系统上必须安装 GDB 和 elfutils。

流程

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

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

输出中包含一行中各个模块的详细信息，用空格分隔。此信息按以下顺序列出：

1. 映射模块的内存地址
2. 模块的 build-id 及其在内存中的位置
3. 模块的可执行文件名称，如果未从文件加载为 **-** 时显示为 **-** when unknown 或 等 。

4. 调试信息源（如果可用时显示为文件名），如 **包含在** 可执行文件本身中，或 （ 如果根本不存在）

5. 主模块的共享库名称(*soname*)或 **[exe]**

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

2. 获取崩溃的可执行文件。

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

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

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

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

3. 获取由可执行文件使用的共享库。将与 相同的步骤用于 可执行文件。

4. 如果应用程序以软件包形式分发，请在 GDB 中加载可执行文件，以显示缺少 debuginfo 软件包的提示。如需了解更多详细信息，请参阅 第 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 加载可执行文件和核心转储文件：

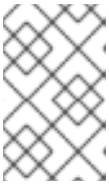
   ```
   $ gdb -e executable_file -c core_file
   ```

   有关缺少文件和调试信息的更多消息可帮助您确定调试会话中缺少的内容。如果需要，返回到上一步骤。

   如果应用的调试信息可作为文件而不是软件包提供，请使用 symbolic **-file** 命令加载这个文件到 GDB 中：

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

   使用实际文件名替换 *program.debug*。

   注意

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

6. 在应用崩溃时，使用 GDB 命令检查应用的状态。请参阅使用 GDB 检查应用内部状态。

   注意

   分析核心文件时，GDB 未附加到正在运行的进程。控制执行的命令无效。

其他资源

- 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

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机器中使用内核转储的工作。此流程概述了如何捕获未响应进程的内核转储。

先决条件

- 系统必须配置为使用 **systemd-coredump 进行** 内核转储处理。验证这是正确的：

```
$ sysctl kernel.core_pattern
```

如果输出以以下内容开始，则配置是正确的：

```
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，查找挂起进程的 PID：

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

此命令将以以下形式输出一行：

```
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. 验证内核是否已由 **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 core 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。

   - 显示核心文件的详细信息：

     ```
     $ coredumpctl info PID
     ```

   - 加载 GDB 调试器中的核心文件：

     ```
     $ coredumpctl debug PID
     ```

     根据调试信息的可用性，GDB 将建议运行的命令，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 3.1.4 节 “使用 GDB 获取应用程序或库的 debuginfo 软件包”。

   - 导出核心文件以便在其他位置进行进一步处理：

     ```
     $ 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

内核转储调试的工作流允许分析程序的状态离线。在某些情况下，您可以将此工作流用于仍在运行的程序，例如难以通过 进程访问环境。您可以使用 **gcore** 命令转储仍在运行的任何进程的内存。

先决条件

- 您必须了解什么是核心转储，以及它们的创建方式。
- 系统上必须安装 GDB。

流程

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

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

2. 转储这个过程的内存：

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

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

3. 内核转储完成后，进程会恢复正常执行。

4. 创建 SOS 报告以提供有关系统的更多信息：

   ```
   # sosreport
   ```

这将创建一个 tar 存档，其中包含系统的相关信息，如配置文件副本。

5. 将程序的可执行文件、核心转储和 SOS 报告传送到要进行调试的计算机。
6. 可选：传输后删除核心转储和 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。这样可以节省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，银行或记帐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)都不会转储以这种方式标记的内存。

在某些情况下，您必须转储进程内存的所有内容，无论这些保护如何。此流程演示了如何使用 GDB 调试器执行此操作。

先决条件

- 您必须了解什么是核心转储。
- 系统上必须安装 GDB。
- GDB 必须已连接到具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称替换 *core-file*。

其他资源

- 使用 GDB 进行调试 - 如何从您的程序生成核心文件

3.5. GDB 中的兼容性破坏更改

Red Hat Enterprise Linux 8 中提供的 GDB 版本包含许多改变，这些更改会破坏兼容性，特别是在直接从终端读取 GDB 输出的情况下。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 机器接口(MI)的首选脚本。

GDBserver 现在使用 shell 启动 inferior

为了在 underferior 命令行参数中启用扩展和变量替换，GDBserver 现在在 shell 中启动与 GDB 相同的扩展和变量替换。

使用 shell 禁用：

- 使用 **目标 extend-remote** GDB 命令时，通过 **set start-with-shell off** 命令禁用 shell。
- 当使用 **目标远程** GDB 命令时,请使用 gdbserver 的 **--no-startup-with-shell** 选项禁用 shell。

例 3.1. 远程 GDB 底层中 shell 扩展的示例

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

删除了对调试使用 GNU Compiler for Java(**gcj**)编译的 Java 程序的支持。

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 中工作。

例如，以下命令不再将符号存储在文件中，而是生成错误消息：

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

转储维护命令的符号的新语法为：

```
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]
```

线程号不再是全局的

在以前的版本中，GDB 只使用全局线程编号。数字已扩展为以 inferior **_num.thread_num** 形式显示，如 **2.1**。因此，$ **_thread** convenience 变量和 **InferiorThread.num** Python 属性中的线程数字在 inferiors 之间不再是唯一的。

GDB 现在为每个线程存储第二个线程 ID，称为全局线程 ID，这是上一个发行版中线程编号的新数量。要访问全局线程号，请使用 **$_gthread convenience** 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 3.2. 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。已添加 **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];** 的值会生成不同的结果：

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

不再支持 stabs 格式的 Sun 版本

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

sysroot 处理更改

当搜索调试所需文件时，使用 **set sysroot *path*** 命令指定系统根。现在，为这个命令提供的目录名可能会有字符串 **target:** 前缀，它使 GDB 从目标系统中（本地和远程）对共享的库。以前可用的 **remote:** 前缀现在被视为 **目标：**此外，默认的系统根值已从空字符串更改为 **target：** 用于向后兼容。

当 GDB 远程启动进程时，或者当它连接到已经运行的进程（本地和远程）时，会预先指定系统 root 的文件名。这意味着，对于远程进程，默认值为 **target：** GDB 始终会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，这样本地符号文件会在远程符号文件之前被发现。

HISTSIZE 不再控制 GDB 命令历史记录大小
在以前的版本中，GDB 使用 **HISTSIZE** 环境变量来确定应保留命令历史记录的时长。GDB 已改为使用 **GDBHISTSIZE** 环境变量。该变量只适用于 GDB。可能的值及其影响有：

- 正数 - 使用这个大小的命令历史记录，
- **-1** 或空字符串 - 保留所有命令的历史记录，
- 非数字值 - 忽略.

添加完成限制
现在可以使用 **set max-completions** 命令来限制在完成过程中考虑的最大候选数。若要显示当前限制，请运行 **show max-completions** 命令。默认值为 200。这个限制可防止 GDB 生成太大的完成列表，并且变得无响应。

例如，输入 **p <tab><tab>** 的输出：

- 对于 RHEL 7：**显示所有 29863 可能性？（y 或 n）**
- 在 RHEL 8 中：**显示所有 200 个可能？（y 或 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。

处理线程信号
在以前的版本中，GDB 可以向当前线程发送信号，而不是实际发送信号的线程。这个程序错误已被解决，GDB 现在总是在恢复执行时将信号传递给正确的线程。

此外，**sign** 命令现在始终正确地将请求的信号传送到当前线程。如果程序因信号和用户切换线程而停止，GDB 将请求确认。

断点模式始终插入并自动合并
breakpoint always-inserted 设置已被更改。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 不会从目标中删除断点，直到所有线程都停止。

remotebaud 命令不再被支持
set remotebaud 和 **show remotebaud** 命令不再被支持。使用 **设置的 serial baud** 并改为 **显示 serial baud** 命令。

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。以下提供类别以及常见的命令行工具。

注意

这不是命令行工具的完整列表。用于调试容器应用的工具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 访问权限，因为它们与系统级别的资源（如网络、systemd 服务或硬件性能计数器）进行交互，这些在无根容器中受到限制。

Rootless 容器在不需要提升特权的情况下运行，在用户命名空间中运行，以改进与主机系统的安全性和隔离。它们通过降低特权升级漏洞的风险，与主机提供有限的交互，降低攻击面并提高安全性。

rootful 容器以提升的特权运行，通常以 root 用户身份运行，从而授予系统资源和功能的完整访问权限。虽然 rootful 容器提供更大的灵活性和控制，但它们会带来安全风险，因为他们可能会升级特权升级并暴露主机系统漏洞。

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

控制容器内的 systemd 服务，允许启动、停止、启用和禁用操作。

journalctl

查看 systemd 服务生成的日志，以帮助对容器问题进行故障排除。

网络工具

ip

管理容器内的网络接口、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。

traceroute

标识路径数据包到达目的地，可用于诊断网络问题。

流程和性能工具

ps

列出容器中当前运行的进程。

top

通过容器内的流程提供对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，有助于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，辅助性能分析。

iostat

监控容器中块设备的输入/输出统计信息。

GDB (GNU Debugger)

一个命令行调试器，有助于检查和调试程序，方法是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。如需更多信息，请参阅 Red Hat OpenShift 容器中调试应用程序 的文章。

strace

截获并记录程序发出的系统调用，通过显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，有助于在不同根目录内测试或进行故障排除。

特定于 podman 的工具

Podman 日志

batch-retrieves 执行时一个或多个容器存在的任何日志。

Podman inspect

显示容器和镜像的低级别信息，如名称或 ID 标识。

Podman 事件

监控并打印 Podman 中发生的事件。每个事件都包括一个时间戳、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在容器内运行的进程的健康状态或就绪状态。

Podman 顶部

显示容器的运行进程。

Podman exec

在 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 导出

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
  - 内核转储、**sosreport**、**gdb**、**ps**、**核心**.
- Kubernetes 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**, systemctl ,**podman exec**/**kill**/**restart**,**podman insect**,**podman top**,**podman exec**,**podman export**,**paunch**
- 外部链接
  - 调试 Docker 容器的十个提示

第 4 章 开发的其他工具集

4.1. 使用 GCC 工具集

4.1.1. 什么是 GCC Toolset

Red Hat Enterprise Linux 8 引进了 GCC Toolset，它是一个 Application Stream，其中包含更多最新版本的开发和性能分析工具。GCC Toolset 与适用于 RHEL 7 的红帽开发人员工具集 类似。

GCC Toolset 以 **AppStream** 存储库中的软件集合的形式作为 Application Stream 提供。在 Red Hat Enterprise Linux 订阅级别协议中完全支持 GCC 工具集，其功能完整，并适用于生产用途。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，也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，另一组开发人员工具安装到 **/opt/** 目录中，用户利用 **scl** 实用程序根据需要明确启用。除非对特定工具或功能另有说明，否则 GCC 工具集适用于红帽企业 Linux 支持的所有架构。

4.1.2. 安装 GCC Toolset

在系统上安装 GCC Toolset 会安装主要工具和所有必需的依赖项。请注意，工具集的某些部分默认未安装，必须单独安装。

流程

- 要安装 GCC Toolset 版本 *N* ：

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

4.1.3. 从 GCC Toolset 安装单个软件包

要仅从 GCC Toolset 而不是整个工具集安装某些工具，请列出可用的软件包并使用 **yum** 软件包管理工具安装选定的软件包。此流程也适用于默认情况下没有使用工具集安装的软件包。

流程

1. 列出 GCC Toolset 版本 *N* 中可用的软件包：

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

2. 安装这些软件包中的任何一个：

```
# yum install package_name
```

使用空格分隔的软件包列表替换 *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
```

4.1.4. 卸载 GCC 工具集

要从您的系统中删除 GCC 工具集，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\*
```

4.1.5. 从 GCC Toolset 运行工具

要从 GCC Toolset 运行工具，请使用 the **scl** 实用程序。

流程

- 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

GCC Toolset 允许运行使用 GCC 工具集工具版本而不是这些工具的系统版本的 shell 会话，而无需显式使用 **scl** 命令。这在您需要多次以交互方式启动工具时很有用，例如在设置或测试开发设置时。

流程

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

```
$ 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 9

了解特定于 GCC Toolset 版本 9 以及这个版本中包含的工具的信息。

4.2.1. GCC Toolset 9 提供的工具和版本

GCC Toolset 9 提供了以下工具和版本：

表 4.1. GCC Toolset 9 中的工具版本

Name	版本	描述
GCC	9.2.1	便携式编译器套件，支持 C、C++ 和 Fortran。
GDB	8.3	命令行调试器，适用于使用 C、C++ 和 Fortran 编写的程序。
Valgrind	3.15.0	检测框架和许多工具来对应用进行性能分析，以检测内存错误、识别内存管理问题并报告系统调用中未使用的参数。

Name	版本	描述
SystemTap	4.1	跟踪和探测工具，可监控整个系统的活动，无需检测、重新编译、安装和重新启动。
Dyninst	10.1.0	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实用程序，用于检查和操作对象文件和二进制文件。
elfutils	0.176	用于检查和操作 ELF 文件的二进制工具和其他实用程序的集合。
dwz	0.12	用于优化 ELF 共享库和 ELF 可执行文件中包含的 DWARF 调试信息的工具。
make	4.2.1	依赖项跟踪构建自动化工具。
strace	5.1	用于监控程序使用的系统调用并发出信号的调试工具。
ltrace	0.7.91	用于显示对程序所进行的动态库的调用的调试工具。它还可以监控程序执行的系统调用。
annobin	9.08	构建安全检查工具.

4.2.2. GCC Toolset 9 中的 C++ 兼容性

重要

此处提供的兼容性信息仅适用于 GCC Toolset 9 中的 GCC。

GCC Toolset 中的 GCC 编译器可以使用以下 C++ 标准：

C++14

这是 GCC Toolset 9 **的默认** 语言标准设置，其 GNU 扩展等同于使用选项 **-std=gnu++14**。
当使用 GCC 版本 6 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++14 语言版本。

C++11

GCC Toolset 9 中提供此语言标准。
当使用 GCC 版本 5 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++11 语言版本。

C++98

GCC Toolset 9 中提供此语言标准。通过使用 GCC Toolset、Red Hat Developer Toolset 以及 RHEL 5、6、7 和 8 的 GCC 构建二进制文件、共享库和对象，都可以自由混合。

C++17, C++2a

这些语言标准仅在 GCC Toolset 9 中以实验、不稳定和不受支持的功能形式提供。此外，不能保证使用这些标准构建的对象、二进制文件和库的兼容性。

所有语言标准均可在符合标准的变体或 GNU 扩展中找到。

将与 GCC Toolset 构建的对象与通过 RHEL 工具链构建的对象（particularly. **o 或. a** 文件）混合时，应将 GCC Toolset 工具链用于任何链接。这可确保任何仅由 GCC Toolset 提供的更新库功能在链接时得以解决。

4.2.3. GCC Toolset 9 中的 GCC 细节

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。

重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会识别对象文件所需的符号：

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

以这种方式使用 GCC Toolset 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

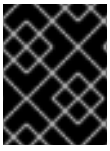
```
$ scl enable gcc-toolset-9 'gcc objfile.o -lsomelib'
```

请注意，在使用基本 Red Hat Enterprise Linux 的 **GCC** 版本时，这个建议也适用。

4.2.4. GCC Toolset 9 中的 binutils 细节

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。



重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会识别对象文件所需的符号：

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

以这种方式使用 GCC Toolset 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

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

请注意，在使用基本 Red Hat Enterprise Linux 的 **binutils** 版本时，这个建议也适用。

4.3. GCC TOOLSET 10

了解特定于 GCC Toolset 版本 10 以及此版本中包含的工具的信息。

4.3.1. GCC Toolset 10 提供的工具和版本

GCC Toolset 10 提供以下工具和版本：

表 4.2. GCC Toolset 10 中的工具版本

Name	版本	描述
GCC	10.2.1	便携式编译器套件，支持 C、C++ 和 Fortran。
GDB	9.2	命令行调试器，适用于使用 C、C++ 和 Fortran 编写的程序。
Valgrind	3.16.0	检测框架和许多工具来对应用进行性能分析，以检测内存错误、识别内存管理问题并报告系统调用中未使用的参数。
SystemTap	4.4	跟踪和探测工具，可监控整个系统的活动，无需检测、重新编译、安装和重新启动。
Dyninst	10.2.1	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实用程序，用于检查和操作对象文件和二进制文件。
elfutils	0.182	用于检查和操作 ELF 文件的二进制工具和其他实用程序的集合。
dwz	0.12	用于优化 ELF 共享库和 ELF 可执行文件中包含的 DWARF 调试信息的工具。
make	4.2.1	依赖项跟踪构建自动化工具。
strace	5.7	用于监控程序使用的系统调用并发出信号的调试工具。
ltrace	0.7.91	用于显示对程序所进行的动态库的调用的调试工具。它还可以监控程序执行的系统调用。
annobin	9.29	构建安全检查工具.

4.3.2. GCC Toolset 10 中的 C++ 兼容性

重要

此处提供的兼容性信息仅适用于 GCC Toolset 10 中的 GCC。

GCC Toolset 中的 GCC 编译器可以使用以下 C++ 标准：

C++14

这是 GCC Toolset 10 **的默认** 语言标准设置，其 GNU 扩展等同于使用 **-std=gnu++14**。
当使用 GCC 版本 6 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++14 语言版本。

C++11

此语言标准在 GCC Toolset 10 中可用。
当使用 GCC 版本 5 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++11 语言版本。

C++98

此语言标准在 GCC Toolset 10 中可用。通过使用 GCC Toolset、Red Hat Developer Toolset 以及 RHEL 5、6、7 和 8 的 GCC 构建二进制文件、共享库和对象，都可以自由混合。

C++17

此语言标准在 GCC Toolset 10 中可用。

C++20

GCC Toolset 10 中仅以实验、不稳定和不受支持的功能提供此语言标准。此外，还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。

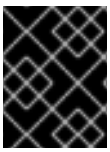
所有语言标准均可在符合标准的变体或 GNU 扩展中找到。

将与 GCC Toolset 构建的对象与通过 RHEL 工具链构建的对象（particularly. **o 或. a** 文件）混合时，应将 GCC Toolset 工具链用于任何链接。这可确保任何仅由 GCC Toolset 提供的更新库功能在链接时得以解决。

4.3.3. GCC Toolset 10 中的 GCC 细节

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。



重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会识别对象文件所需的符号：

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

以这种方式使用 GCC Toolset 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

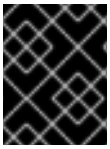
```
$ scl enable gcc-toolset-10 'gcc objfile.o -lsomelib'
```

请注意，在使用基本 Red Hat Enterprise Linux 的 **GCC** 版本时，这个建议也适用。

4.3.4. GCC Toolset 10 中的 binutils 细节

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。



重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会识别对象文件所需的符号：

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

以这种方式使用 GCC Toolset 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

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

请注意，在使用基本 Red Hat Enterprise Linux 的 **binutils** 版本时，这个建议也适用。

4.4. GCC TOOLSET 11

了解特定于 GCC Toolset 版本 11 以及此版本中包含的工具的信息。

4.4.1. GCC Toolset 11 提供的工具和版本

GCC Toolset 11 提供以下工具和版本：

表 4.3. GCC Toolset 11 中的工具版本

Name	版本	描述
GCC	11.2.1	便携式编译器套件，支持 C、C++ 和 Fortran。

Name	版本	描述
GDB	10.2	命令行调试器，适用于使用 C、C++ 和 Fortran 编写的程序。
Valgrind	3.17.0	检测框架和许多工具来对应用进行性能分析，以检测内存错误、识别内存管理问题并报告系统调用中未使用的参数。
SystemTap	4.5	跟踪和探测工具，可监控整个系统的活动，无需检测、重新编译、安装和重新启动。
Dyninst	11.0.0	执行期间检测和使用用户空间可执行文件的库。
binutils	2.36.1	一组二进制工具和其他实用程序，用于检查和操作对象文件和二进制文件。
elfutils	0.185	用于检查和操作 ELF 文件的二进制工具和其他实用程序的集合。
dwz	0.14	用于优化 ELF 共享库和 ELF 可执行文件中包含的 DWARF 调试信息的工具。
make	4.3	依赖项跟踪构建自动化工具。
strace	5.13	用于监控程序使用的系统调用并发出信号的调试工具。
ltrace	0.7.91	用于显示对程序所进行的动态库的调用的调试工具。它还可以监控程序执行的系统调用。
annobin	10.23	构建安全检查工具.

4.4.2. GCC Toolset 11 中的 C++ 兼容性

重要

此处所提供的兼容性信息仅适用于 GCC Toolset 11 中的 GCC。

GCC Toolset 中的 GCC 编译器可以使用以下 C++ 标准：

C++14

此语言标准在 GCC Toolset 11 中提供。
当使用 GCC 版本 6 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++14 语言版本。

C++11

此语言标准在 GCC Toolset 11 中提供。
当使用 GCC 版本 5 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++11 语言版本。

C++98

此语言标准在 GCC Toolset 11 中提供。通过使用 GCC Toolset、Red Hat Developer Toolset 以及 RHEL 5、6、7 和 8 的 GCC 构建二进制文件、共享库和对象，都可以自由混合。

C++17

此语言标准在 GCC Toolset 11 中提供。
这是 GCC Toolset 11 的默认语言标准设置，它使用 GNU 扩展，相当于使用 **-std=gnu++17** 选项明确设置。

当使用 GCC 版本 10 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++17 语言版本。

c++20 和 C++23

此语言标准在 GCC Toolset 11 中仅作为实验性、不稳定和不受支持的功能提供。此外，还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，请将命令行选项 **-std=c++20** 添加到 g++ 命令行。

要启用 C++23 支持，将命令行选项 **-std=c++2b** 添加到 g++ 命令行。

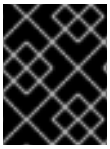
所有语言标准均可在符合标准的变体或 GNU 扩展中找到。

将与 GCC Toolset 构建的对象与通过 RHEL 工具链构建的对象（particularly. **o 或．a** 文件）混合时，应将 GCC Toolset 工具链用于任何链接。这可确保任何仅由 GCC Toolset 提供的更新库功能在链接时得以解决。

4.4.3. GCC Toolset 11 中的具体信息

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。



重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会识别对象文件所需的符号：

```
$ scl enable gcc-toolset-11 'gcc -lsomelib objfile.o'
```

以这种方式使用 GCC Toolset 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

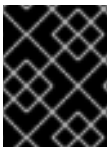
```
$ scl enable gcc-toolset-11 'gcc objfile.o -lsomelib'
```

请注意，在使用基本 Red Hat Enterprise Linux 的 **GCC** 版本时，这个建议也适用。

4.4.4. GCC Toolset 11 中的 binutils 细节

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。



重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会识别对象文件所需的符号：

```
$ scl enable gcc-toolset-11 'ld -lsomelib objfile.o'
```

以这种方式使用 GCC Toolset 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

```
$ scl enable gcc-toolset-11 'ld objfile.o -lsomelib'
```

请注意，在使用基本 Red Hat Enterprise Linux 的 **binutils** 版本时，这个建议也适用。

4.5. GCC TOOLSET 12

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

4.5.1. GCC Toolset 12 提供的工具和版本

GCC Toolset 12 提供以下工具和版本：

表 4.4. GCC Toolset 12 中的工具版本

Name	版本	描述
GCC	12.2.1	便携式编译器套件，支持 C、C++ 和 Fortran。
GDB	11.2	命令行调试器，适用于使用 C、C++ 和 Fortran 编写的程序。
binutils	2.38	一组二进制工具和其他实用程序，用于检查和操作对象文件和二进制文件。
dwz	0.14	用于优化 ELF 共享库和 ELF 可执行文件中包含的 DWARF 调试信息的工具。
annobin	11.08	构建安全检查工具.

4.5.2. 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 编译器可以使用以下 C++ 标准：

C++14

这个语言标准包括在 GCC Toolset 12 中。
当使用 GCC 版本 6 或更高版本构建了使用相应标记编译的所有 C++ 对象时，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 GCC 版本 5 或更高版本构建了使用相应标记编译的所有 C++ 对象时，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。通过使用 GCC Toolset、Red Hat Developer Toolset 以及 RHEL 5、6、7 和 8 的 GCC 构建二进制文件、共享库和对象，都可以自由混合。

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，它相当于使用 **-std=gnu++17** 选项进行显式使用。

当使用 GCC 版本 10 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++17 语言版本。

c++20 和 C++23

这个语言标准只在 GCC Toolset 12 中作为实验性、不稳定和不受支持的功能提供。此外，还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，请将命令行选项 **-std=c++20** 添加到 g++ 命令行。

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

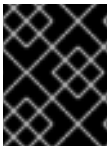
所有语言标准均可在符合标准的变体或 GNU 扩展中找到。

将与 GCC Toolset 构建的对象与通过 RHEL 工具链构建的对象（particularly. **o 或. a** 文件）混合时，应将 GCC Toolset 工具链用于任何链接。这可确保任何仅由 GCC Toolset 提供的更新库功能在链接时得以解决。

4.5.3. GCC Toolset 12 中的 GCC 的具体信息

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。



重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会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 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

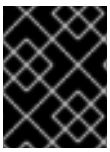
```
$ scl enable gcc-toolset-12 'gcc objfile.o -lsomelib'
```

请注意，在使用基本 Red Hat Enterprise Linux 的 **GCC** 版本时，这个建议也适用。

4.5.4. GCC Toolset 12 中 binutils 的细节

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。



重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会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 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

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

请注意，在使用基本 Red Hat Enterprise Linux 的 **binutils** 版本时，这个建议也适用。

4.5.5. GCC Toolset 12 中 annobin 的细节

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**

4.6. 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。

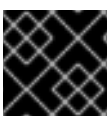
4.6.1. GCC Toolset 13 提供的工具和版本

GCC Toolset 13 提供以下工具和版本：

表 4.5. GCC Toolset 13 中的工具版本

Name	版本	描述
GCC	13.2.1	便携式编译器套件，支持 C、C++ 和 Fortran。
GDB	12.1	命令行调试器，适用于使用 C、C++ 和 Fortran 编写的程序。
binutils	2.40	一组二进制工具和其他实用程序，用于检查和操作对象文件和二进制文件。
dwz	0.14	用于优化 ELF 共享库和 ELF 可执行文件中包含的 DWARF 调试信息的工具。
annobin	12.32	构建安全检查工具.

4.6.2. 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 编译器可以使用以下 C++ 标准：

C++14

这个语言标准在 GCC Toolset 13 中提供。
当使用 GCC 版本 6 或更高版本构建了使用相应标记编译的所有 C++ 对象时，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 GCC 版本 5 或更高版本构建了使用相应标记编译的所有 C++ 对象时，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。通过使用 GCC Toolset、Red Hat Developer Toolset 以及 RHEL 5、6、7 和 8 的 GCC 构建二进制文件、共享库和对象，都可以自由混合。

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 GCC 版本 10 或更高版本构建了使用相应标记编译的所有 C++ 对象时，支持使用 C++17 语言版本。

c++20 和 C++23

在 GCC Toolset 13 中，这些语言标准仅作为实验性、不稳定和不支持的功能提供。此外，还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**。

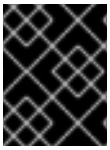
所有语言标准均可在符合标准的变体或 GNU 扩展中找到。

将与 GCC Toolset 构建的对象与通过 RHEL 工具链构建的对象（particularly. **o 或．a** 文件）混合时，应将 GCC Toolset 工具链用于任何链接。这可确保任何仅由 GCC Toolset 提供的更新库功能在链接时得以解决。

4.6.3. GCC Toolset 13 中 GCC 的具体内容

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。



重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与

预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会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 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

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

请注意，在使用基本 Red Hat Enterprise Linux 的 **GCC** 版本时，这个建议也适用。

4.6.4. GCC Toolset 13 中 binutils 的具体内容

库的静态链接

某些较新的库功能已静态链接到使用 GCC 工具集构建的应用程序中，以支持在多个版本的 Red Hat Enterprise Linux 中执行。这会产生另外一个小的安全风险，因为标准的 Red Hat Enterprise Linux 勘误表不会改变这个代码。如果开发人员出于此风险而需要重建其应用，红帽将使用安全勘误对此进行沟通。

重要

由于这种额外的安全风险，开发人员强烈建议不要出于相同的原因将整个应用程序静态链接。

在链接时在对象文件后指定库

在 GCC Toolset 中，库使用链接器脚本链接，这些脚本可能通过静态存档指定一些符号。这需要确保与红帽企业 Linux 的多个版本兼容。但是，链接器脚本使用对应共享对象文件的名称。因此，链接器使用与预期不同的符号处理规则，在指定对象文件选项前，在指定对象文件选项前不会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 中的库会导致 **未定义引用的 符号链接器错误消息。**要防止这个问题，请遵循标准链接实践，并在指定对象文件的选项后指定添加库的选项：

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

请注意，在使用基本 Red Hat Enterprise Linux 的 **binutils** 版本时，这个建议也适用。

4.6.5. 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**

4.7. 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 **rhel8** 基础镜像，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

4.7.1. 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

rhel8/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

4.7.2. 访问并运行 GCC Toolset 容器镜像

下面的部分论述了如何访问并运行 GCC Toolset 容器镜像。

先决条件

- Podman 已安装。

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-13-toolchain
```

注意

在 RHEL 8.1 及更新的版本中，您可以将您的系统设置为以非 root 用户身份使用容器。详情请参阅 设置 rootless 容器。

3. 可选：运行列出本地系统上所有容器镜像的命令，检查拉取是否成功：

```
# podman images
```

4. 通过在容器内启动 bash shell 运行容器：

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

-i 选项创建一个交互式会话；如果没有此选项，shell 将打开并立即退出。

t 选项打开终端会话；没有此选项，您就无法在 shell 中键入任何内容。

其他资源

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

4.7.3. Example: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。

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 用户身份拉取容器镜像：

```
# podman pull registry.redhat.io/rhel8/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/rhel8/gcc-toolset-13-toolchain /bin/bash
```

4. 按照预期运行 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 13.1.1 20231102 (Red Hat 13.1.1-4) (GCC)
```

5. 要列出容器中提供的所有软件包，请运行：

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

4.8. 编译器工具集

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

- LLVM Toolset 提供 LLVM 编译器基础架构框架、Clang 编译器用于 C 和 C++ 语言、LLDB 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** 构建工具和依赖关系管理器、**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。

4.9. ANNOBIN 项目

Annobin 项目是 Watermark 规范项目的实施。水印规范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。

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 注释。ELF 注释记录了二进制文件的构建方式，并为 **annocheck** 程序提供信息，以执行安全强化检查。

安全强化检查器是 **annocheck** 程序的一部分，默认启用。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并正确编译。**nocheck** 能够递归扫描 ELF 对象文件的目录、存档和 RPM 软件包。

注意

文件必须采用 ELF 格式。**anocheck** 不会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

4.9.1. 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

4.9.1.1. 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** 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 **annobin** 插件，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。

4.9.1.2. 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。

流程

- 要使用 **gcc** 将选项传递给 **annobin** 插件，请使用：

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

  将 *选项* 替换为 **annobin** 命令行参数，并使用文件名称替换 *file-name*。

  示例

- 要显示关于它的 **annobin** 用途的附加详情，请使用：

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

用文件名替换 *file-name*。

- 要使用 **clang** 将选项传递给 **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

- 要显示关于它的 **annobin** 用途的附加详情，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*。

4.9.2. 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

无检查会以 递归方式扫描 ELF 对象文件的目录、存档和 RPM 软件包。文件必须采用 ELF 格式。ano **check** 不会处理任何其他二进制文件类型。

4.9.2.1. 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 *文件的名称替换 file-name*。

注意

文件必须采用 ELF 格式。**anocheck** 不会处理任何其他二进制文件类型。**nocheck** 进程包含 ELF 对象文件的静态库。

其他信息

- 有关 **annocheck** 和可能的命令行选项的详情，请查看 **annocheck** man page。

4.9.2.2. 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
```

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

注意

annocheck 只查找 ELF 文件。忽略其他文件类型。

其他信息

- 有关 **annocheck** 和可能的命令行选项的详情，请查看 **annocheck** man page。

4.9.2.3. 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*。**不检查** 以递归方式扫描 RPM 软件包中的所有 ELF 文件。

注意

annocheck 只查找 ELF 文件。忽略其他文件类型。

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

其他信息

- 有关 **annocheck** 和可能的命令行选项的详情，请查看 **annocheck** man page。

4.9.2.4. 使用 annocheck 额外工具

annocheck 包含用于检查二进制文件的多个工具。您可以通过 命令行选项启用这些工具。

下面的部分论述了如何启用：

- **内置** 工具
- **备注** 工具
- **section-size** 工具

您可以同时启用多个工具。

注意

默认启用强化检查器。

4.9.2.4.1. 通过工具启用构建

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

流程

- 要启用 **内置的工具，** 请使用：

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

其他信息

- 有关 **内置工具** 的更多信息，请参阅 **--help** 命令行选项。

4.9.2.4.2. 启用 备注 工具

您可以使用 **annocheck 备注** 工具显示由 **annobin** 插件创建的二进制文件内存储的注释。

流程

- 要启用 **备注** 工具，请使用：

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

备注按地址范围排序的顺序显示。

其他信息

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

4.9.2.4.3. 启用 section-size 工具

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

流程

- 要启用 **section-size** 工具，请使用：

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

使用 named 部分的名称替换 *name*。输出仅限于特定的部分。最终产生累计结果。

其他信息

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

4.9.2.4.4. 强化检查器基础知识

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

4.9.2.4.4.1. 强化检查器选项

annocheck 程序检查以下选项：

- lazy binding 被禁用，使用 **-z now** linker 选项。
- 程序没有可执行的内存区域中的堆栈。
- GOT 表的重新定位设置为只读。
- 没有程序段设置全部三个读取、写入和执行权限位。
- 没有针对可执行代码的重新定位。
- 在运行时查找共享库的 runpath 信息仅包含根于 /usr 的目录。
- 程序在启用了 **annobin** 注释的情况下被编译。
- 该程序编译时启用了 **-fstack-protector-strong** 选项。
- 该程序使用 **-D_FORTIFY_SOURCE=2** 进行编译。
- 该程序使用 **-D_GLIBCXX_ASSERTIONS** 进行编译。
- 该程序编译时已启用 **-f 例外**。
- 该程序编译时 **启用了 -fstack-clash-protection**。
- 该程序编译为 **-O2** 或更高版本。
- 该程序没有任何可写位置的重新定位。
- 动态可执行文件具有动态分段。
- 共享库使用 **-fPIC 或 -fPIE** 进行编译。
- 使用 **-fPIE** 编译动态可执行文件并链接到 **-pie**。

- 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。

4.9.2.4.4.2. 禁用强化检查程序

下面的部分论述了如何禁用强化检查器。

流程

- 要在没有强化检查器的文件中扫描备注，请使用：

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

使用 *文件的名称替换 file-name*。

4.9.3. 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 an **nobin 编译的二进制文件的大小，** 您可以删除冗余 **annobin** 备注。要删除冗余的 **annobin** 注释，请使用 **objcopy** 程序，该程序是 **binutils** 软件包的一部分。

流程

- 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*。

4.9.4. GCC Toolset 12 中 annobin 的细节

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

已删除 librtkaio

在这个版本中，lib **rtkaio** 库已被删除。此库为某些文件提供了高性能实时异步 I/O 访问，该文件基于 Linux 内核异步 I/O 支持(KAIO)。

删除后：

- 使用 **LD_PRELOAD** 加载 **librtkaio** 的应用会显示有关缺少库的警告，加载 **librt** 库并正确运行。
- 使用 **LD_LIBRARY_PATH** 方法加载 **librtkaio** 的应用程序会载入 **librt** 库，并在没有任何警告的情况下正确运行。
- 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：

- 使用上述回退机制，无需更改其应用。
- 更改其应用程序的代码以使用 **librt** 库，该库提供兼容的 POSIX 兼容 API。
- 更改其应用的代码以使用 **libaio** 库，该库提供兼容的 API。

librt 和 **libaio** 都可以在特定条件下提供可比的功能和性能。

请注意，lib **aio** 软件包的红帽兼容性级别为 2，而 **librtkaio 和** 移除的 **librtkaio** 级别 1。

详情请查看 https://fedoraproject.org/wiki/Changes/GLIBC223_librtkaio_removal

从 glibc中删除 Sun RPC 和 NIS 接口

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传统应用程序。开发人员必须更改其应用程序以使用 **libtirpc** 库，而不是 Sun RPC 和 **libnsl2** 而不是 NIS。应用可从替换库中的 IPv6 支持中受益。

删除了适用于 32 位 Xen 的 nosegneg 库

在以前的版本中，gli **bc i** 686 软件包包含一个替代 **glibc** 构建，它避免使用带有负偏移(**nosegneg**)的线程描述符片段注册。此备选构建仅在没有硬件虚拟化支持的 Xen 项目系统管理程序的 32 位版本中使用，作为降低完全半虚拟化成本的优化。这些替代构建不再使用，它们已被删除。

make 的新操作 != 会与特定的现存 makefile 语法有不同的解释

!= shell 分配运算符已添加到 GNU **make** 中，作为 **$(shell ... 。）** 功能的替代方案，以提高与 BSD makefile 的兼容性。因此，名称以感叹号结尾的变量，紧接在 **变量！=value** 等分配后立即解析为 shell 分配。要恢复之前的行为,请在声明标记后添加一个空格,比如 **variable! =value**。

有关运算符和函数之间的更多详细信息和差异，请参阅 GNU **make** manual。

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** 包装器库已被删除。这个库启用了 **Valgrind** 使用 Message Passing Interface(MPI)调试程序。这个库专用于之前版本的 Red Hat Enterprise Linux 中的 Open MPI 实施版本。

建议 **libmpiwrap.so** 用户从特定于其 MPI 实施和版本的上游源构建自己的版本。利用 **LD_PRELOAD** 技术，将这些自定义的库提供给 **Valgrind**。

开发标头和静态库已从 valgrind-devel 中删除
在以前的版本中，**valgrind-devel** 子软件包用于包含用于开发自定义 valgrind 工具的开发文件。在这个版本中会删除这些文件，因为它们没有保证的 API，所以必须静态链接，且不受支持。**valgrind-devel** 软件包仍包含用于 valgrind-aware 程序和标头文件的开发文件，如 **valgrind.h**、**callgrind.h**、**drd.h**、**helgrind.h** 和 **memcheck.h**，它们稳定且受到良好支持。

5.2. 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，可以使用一系列选项。系统管理员需要应用开发人员提供的详细指导。以下列表概述了红帽提供的选项、注意事项和资源。

使用匹配的 RHEL 版本客户机操作系统在虚拟机中运行应用程序

此选项的资源成本较高，但环境与应用的要求紧密匹配，这种方法不需要很多额外的注意事项。这是当前推荐的选项。

根据相应的 RHEL 版本在容器中运行应用程序

资源成本低于之前的情形中，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详情，请参阅 Red Hat Enterprise Linux Container Compatibility Matrix 。

在 RHEL 8 中原生运行应用程序

这个选项提供最低的资源成本，但也有最严格的要求。应用程序开发人员必须确定 RHEL 8 系统的正确配置。以下资源可帮助开发人员完成此任务：

- 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，此列表不是确定应用程序兼容性所需的一整套资源。这些仅是从已知不兼容更改列表和红帽与兼容性相关的政策开始。

此外，*什么是内核应用程序二进制接口(kABI)？*知识库支持文章包含与内核和兼容性相关的信息。