5.10. GCC-4.9.2 - 第2遍

GCC 软件包包含 GNU 编译器集合，其中有 C 和 C++ 编译器。

大概编译时间: 7.7 SBU
所需磁盘空间: 2.6 GB

5.10.1. 安装 GCC

我们第一次编译 GCC 的时候安装了一些内部系统头文件。其中的一个 `limits.h` 会反过来包括对应的系统头文件 `limits.h`，在我们的例子中，是 `/tools/include/limits.h`。但是，第一次编译 gcc 的时候 `/tools/include/limits.h` 并不存在，因此 GCC 安装的内部头文件只是部分的自包含文件，并不包括系统头文件的扩展功能。这足以编译临时 libc，但是这次编译 GCC 要求完整的内部头文件。使用和正常情况下 GCC 编译系统使用的相同的命令创建一个完整版本的内部头文件：

```
cat gcc/limitx.h gcc/glimits.h gcc/limity.h > \
  `dirname $($LFS_TGT-gcc -print-libgcc-file-name)`/include-fixed/limits.h
```

再一次更改 GCC 的默认动态链接器的位置，使用安装在 `/tools` 的那个。

```
for file in \
 $(find gcc/config -name linux64.h -o -name linux.h -o -name sysv4.h)
do
  cp -uv $file{,.orig}
  sed -e 's@/lib\(64\)\?\(32\)\?/ld@/tools&@g' \
      -e 's@/usr@/tools@g' $file.orig > $file
  echo '
#undef STANDARD_STARTFILE_PREFIX_1
#undef STANDARD_STARTFILE_PREFIX_2
#define STANDARD_STARTFILE_PREFIX_1 "/tools/lib/"
#define STANDARD_STARTFILE_PREFIX_2 ""' >> $file
  touch $file.orig
done
```

和第一次编译 GCC 一样，它要求 GMP、MPFR 和 MPC 软件包。解压 tar 包并把它们重名为到所需的文件夹名称：

```
tar -xf ../mpfr-3.1.2.tar.xz
mv -v mpfr-3.1.2 mpfr
tar -xf ../gmp-6.0.0a.tar.xz
mv -v gmp-6.0.0 gmp
tar -xf ../mpc-1.0.2.tar.gz
mv -v mpc-1.0.2 mpc
```

再次创建独立的编译文件夹：

```
mkdir -v ../gcc-build
cd ../gcc-build
```

在开始编译 GCC 之前，记住取消所有会覆盖默认优化选项的环境变量。

准备编译 GCC：

```
CC=$LFS_TGT-gcc                                \
CXX=$LFS_TGT-g++                               \
AR=$LFS_TGT-ar                                 \
RANLIB=$LFS_TGT-ranlib                         \
../gcc-4.9.2/configure                         \
    --prefix=/tools                            \
    --with-local-prefix=/tools                 \
    --with-native-system-header-dir=/tools/include \
    --enable-languages=c,c++                   \
    --disable-libstdcxx-pch                    \
    --disable-multilib                         \
    --disable-bootstrap                        \
    --disable-libgomp
```

新配置选项的含义：

`--enable-languages=c,c++`
这个选项确保编译了 C 和 C++ 编译器。

`--disable-libstdcxx-pch`
不为 `libstdc++` 编译预编译的头文件(PCH)。这会花费很多时间，却对我们没有用处。

`--disable-bootstrap`
对于原生编译的 GCC，默认是做一个"引导"构建。这不仅会编译 GCC，而且会多次编译。它用第一次编译的程序去第二次编译自己，然后同样进行第三次。比较第二次和第三次迭代确保它可以完美复制自身。这也意味着已经成功编译。但是，LFS 的构建方法能够提供一个稳定的编译器，而不需要每次都重新引导。

编译软件包：

```
make
```

安装软件包：

```
make install
```

作为画龙点睛，这里创建一个符号链接。很多程序和脚本执行 `cc` 而不是 `gcc` 来保持程序的通用性，因而在所有并不总是安装了 GNU C 编译器的 Unix 类型的系统上都可以使用。运行 `cc` 使得系统管理员不用考虑要安装那种 C 编译器：

```
ln -sv gcc /tools/bin/cc
```

注意

到了这里，必须停下来确认新工具链的基本功能(编译和链接)都是像预期的那样正常工作。运行下面的命令进行全面的检查：

```
echo 'main(){}' > dummy.c
cc dummy.c
readelf -l a.out | grep ': /tools'
```

如果一切工作正常的话，这里应该没有错误，最后一个命令的输出形式会是：

```
[Requesting program interpreter: /tools/lib/ld-linux.so.2]
```

注意 `/tools/lib`、或者 64 位机器的 `/tools/lib64` 会以动态链接器的前缀出现。

如果输出不是像上面那样或者根本就没有输出，那么可能某些地方出错了。调查并回溯这些步骤，找出问题所在并改正它。在继续之前必须解决这个问题。首先，使用 `gcc` 而不是 `cc` 再次进行全面的检查。如果能运行，就丢失了 `/tools/bin/cc` 符号链接。像上面介绍的那样新建符号链接。下一步，确认 `PATH` 是正常的。这能通过运行 `echo $PATH` 检验，验证 `/tools/bin` 在列表的前面。如果 `PATH` 是错误的，这意味着你可能不是以 `lfs` 用户的身份登录或者前面 4.4,"设置环境"中某些地方出现了错误。

一旦一切都顺利，清理测试文件：

```
rm -v dummy.c a.out
```

该软件包的详细信息在 Section 6.17.2, "GCC 软件包内容"

翻译团队:LCTT 译者/校对:ictlyh,dongfengweixiao