第 5 章

鲲鹏应用迁移

5.1 应用迁移的原因

8min

5.1.1 不同架构下程序执行对比

通过一个简单的 C 程序,演示一下在不同架构下编译运行的对比,要对比的环境如表 5-1 所示。

表 5-1 运行环境对比

方式	编译环境	运行环境
方式 1	x86	x86
方式 2	Kunpeng	Kunpeng
方式 3	x86	Kunpeng
方式 4	Kunpeng	x86

1. 方式 1

步骤 1:准备好 x86 架构的运行环境,安装 CentOS 操作系统,并且安装好标准 C 开发环境,具体的步骤可以参考 4.2 节准备软件环境的内容,注意 CPU 架构选择 x86 架构。

步骤 2:创建/data/code/文件夹,然后创建 x86_demo.c,命令如下:

```
mkdir /data/code/
cd /data/code/
vim x86_demo.c
```

步骤 3:按 i 键进入编辑模式,输入代码,然后保存并退出,代码如下:

```
//Chapter5/x86_demo.c

int main(void)
{
    int a = 1;
```

```
    int b = 2;
    int c = 0;
    c = a + b;
    return c;
}
```

步骤 4：编译 x86_demo.c，生成编译后的文件 x86_demo，命令如下：

```
gcc -g -o x86_demo x86_demo.c
```

注意：这里使用了 gcc 的-g 选项，使用该选项在编译时会额外执行如下的操作：

(1) 创建符号表，符号表包含了程序中使用的变量名称的列表。

(2) 关闭所有的优化机制，以便程序执行过程中严格按照原来的 C 代码进行。

这样，在后续的反编译的时候，可以用汇编代码和 C 源代码进行对比，便于理解汇编后的代码。

步骤 5：运行 x86_demo，命令如下：

```
./x86_demo
```

因为这个演示程序没有输出，所以运行 x86_demo 也没有回显。

2. 方式 2

步骤 1：准备鲲鹏架构的 C 开发环境，参考 4.2 节准备软件环境的内容。

步骤 2：创建/data/code/文件夹，然后创建 kunpeng_demo.c，命令如下：

```
mkdir /data/code/
cd /data/code/
vim kunpeng_demo.c
```

步骤 3：按 i 键进入编辑模式，输入代码，然后保存并退出，代码如下：

```
//Chapter5/kunpeng_demo.c
int main(void)
{
    int a = 1;
    int b = 2;
    int c = 0;
    c = a + b;
    return c;
}
```

步骤 4：编译 kunpeng_demo.c，生成编译后的文件 kunpeng_demo，命令如下：

```
aarch64-redhat-Linux-gcc -g -o kunpeng_demo kunpeng_demo.c
```

步骤 5：运行 kunpeng_demo，命令如下：

```
./kunpeng_demo
```

同样没有回显。

3. 方式 3

步骤 1：登录鲲鹏架构服务器

步骤 2：从 x86 服务器复制编译好的 x86_demo 到本地，命令如下：

```
scp root@192.168.0.208:/data/code/x86_demo /data/code/
```

需要根据服务器的实际情况修改 x86 服务器的用户名和 IP。

步骤 3：运行 x86_demo，命令如下：

```
./x86_demo
```

系统会提示无法运行该文件，如图 5-1 所示。

```
[root@ecs-kunpeng code]# ./x86_demo
-bash: ./x86_demo: cannot execute binary file
[root@ecs-kunpeng code]#
```

图 5-1　鲲鹏架构运行 x86 程序

4. 方式 4

步骤 1：登录 x86 架构服务器

步骤 2：从鲲鹏服务器复制编译好的 kunpeng_demo 到本地，命令如下：

```
scp root@192.168.0.133:/data/code/kunpeng_demo /data/code/
```

需要根据实际情况修改 Kunpeng 服务器的用户名和 IP。

步骤 3：运行 kunpeng_demo，命令如下：

```
./kunpeng_demo
```

系统会提示无法运行该文件，如图 5-2 所示。

```
[root@ecs-x86 code]# ./kunpeng_demo
-bash: ./kunpeng_demo: cannot execute binary file
[root@ecs-x86 code]#
```

图 5-2　x86 架构运行鲲鹏程序

根据上面的 4 个小实验，可以得出这样的结论，x86 架构下编译的 C 程序无法在鲲鹏架构下直接运行；同样，鲲鹏架构下编译的 C 程序也无法在 x86 架构下运行。为什么会这样呢？在 5.1.2 节进行有针对性的分析。

5.1.2 不同架构下汇编指令分析

8min

1. 鲲鹏架构

针对 5.1.1 节在鲲鹏架构下编译的程序 kunpeng_demo，通过反汇编工具查看它的汇编指令，详细步骤如下：

步骤 1：安装反汇编工具 objdump。objdump 在工具包 binutils 中，可以通过 yum 安装该工具包，命令如下：

```
yum install -y binutils
```

步骤 2：反编译 kunpeng_demo，命令及回显如下(因为反编译后的汇编代码太多，这里只保留与 main 方法相关的汇编代码)：

```
[root@ecs-kunpeng code]# objdump -S kunpeng_demo

kunpeng_demo: file format elf64-littleaarch64
//此处省略几百行代码…
...

00000000004005b0 <main>:
int main(void)
{
  4005b0:    d10043ff        sub    sp, sp, #0x10
    int a = 1;
  4005b4:    52800020        mov    w0, #0x1                    //#1
  4005b8:    b9000fe0        str    w0, [sp,#12]
    int b = 2;
  4005bc:    52800040        mov    w0, #0x2                    //#2
  4005c0:    b9000be0        str    w0, [sp,#8]
    int c = 0;
  4005c4:    b90007ff        str    wzr, [sp,#4]
    c = a + b;
  4005c8:    b9400fe1        ldr    w1, [sp,#12]
  4005cc:    b9400be0        ldr    w0, [sp,#8]
  4005d0:    0b000020        add    w0, w1, w0
  4005d4:    b90007e0        str    w0, [sp,#4]
    return c;
  4005d8:    b94007e0        ldr    w0, [sp,#4]
}
  4005dc:    910043ff        add    sp, sp, #0x10
  4005e0:    d65f03c0        ret
//此处省略几百行代码…
```

这样，就得到了 C 语言代码对应的汇编代码。

下面对 main 函数的代码逐行分析，同时对鲲鹏架构指令和寄存器进行简单介绍。

1）sub sp，sp，＃0x10

sp 寄存器保持栈顶位置，这里向下扩展了 16 字节，这 16 字节可以用来给后面的变量分配内存空间，栈默认最小扩展空间是 16 字节，每次扩展空间是 16 字节的整数倍。

2）mov w0，＃0x1

把操作数 1 赋值给寄存器 w0。

3）str w0，[sp，＃12]

把 w0 寄存器的值传送到栈顶开始的第 12 字节对应的内存中。也就是给变量 a 赋值。

4）mov w0，＃0x2

把操作数 2 赋值给寄存器 w0。

5）str w0，[sp，＃8]

把 w0 寄存器的值传送到栈顶开始的第 8 字节对应的内存中，也就是给变量 b 赋值。

6）str wzr，[sp，＃4]

把零寄存器的值传送到栈顶开始的第 4 字节对应的内存中，也就是给变量 c 赋值。零寄存器的值总是 0。

7）ldr w1，[sp，＃12]

把栈顶开始的第 12 字节对应的内存数据传送给 w1 寄存器，也就是把变量 a 读到寄存器 w1。

8）ldr w0，[sp，＃8]

同上，把变量 b 读到寄存器 w0。

9）add w0，w1，w0

把 w0 和 w1 相加，存到 w0 寄存器中。

10）str w0，[sp，＃4]

把寄存器 w0 的值写到变量 c 中。

11）ldr w0，[sp，＃4]

把变量 c 中的值写回寄存器 w0，w0 用来作为返回值寄存器。

12）add sp，sp，＃0x10

恢复栈空间，释放内存。

注意：这里使用了 objdump 的-S 选项，该选项将代码段反汇编的同时，将反汇编代码和源代码交替显示。该选项需要 gcc 在编译时使用-g 的选项。

2. x86 架构

针对 5.1.1 节在 x86 架构下编译的程序 x86_demo，通过反汇编工具查看它的汇编指令，详细步骤如下：

步骤 1：安装反汇编工具 objdump。objdump 在工具包 binutils 中，可以通过 yum 安装

该工具包，命令如下：

```
yum install -y binutils
```

步骤 2：反编译 x86_demo，命令及回显如下（只保留与 main 方法相关的汇编代码）：

```
[root@ecs-x86 code]#objdump -S x86_demo

x86_demo: file format elf64-x86-64
//此处省略几百行代码…
00000000004004ed <main>:
int main(void)
{
  4004ed:    55                      push   %rbp
  4004ee:    48 89 e5                mov    %rsp,%rbp
    int a = 1;
  4004f1:    c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%rbp)
    int b = 2;
  4004f8:    c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
    int c = 0;
  4004ff:    c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%rbp)
    c = a + b;
  400506:    8b 45 f8                mov    -0x8(%rbp),%eax
  400509:    8b 55 fc                mov    -0x4(%rbp),%edx
  40050c:    01 d0                   add    %edx,%eax
  40050e:    89 45 f4                mov    %eax,-0xc(%rbp)
    return c;
  400511:    8b 45 f4                mov    -0xc(%rbp),%eax
}
  400514:    5d                      pop    %rbp
  400515:    c3                      retq
  400516:    66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40051d:    00 00 00
//此处省略几百行代码
```

对 main 函数的每行代码简要解释如下：

1）push %rbp

将调用函数的栈帧栈底地址入栈，即将 bp 寄存器的值压入调用栈中。

2）mov %rsp,%rbp

建立新的栈帧，将 main 函数的栈帧栈底地址放入 bp 寄存器中。sp 和 bp 是两个指针寄存器，一般的函数调用都会使用上述两个指令。

3）movl　$0x1,-0x4(%rbp)

把1传送给变量a(整型变量占用4字节,所以这里是-0x4)。

4）movl　$0x2,-0x8(%rbp)

把2传送给变量b。

5）movl　$0x0,-0xc(%rbp)

把0传送给变量c。

6）mov　-0x8(%rbp),%eax

把变量b的值传送给eax寄存器。

7）mov　-0x4(%rbp),%edx

把变量a的值传送给edx寄存器。

8）add　%edx,%eax

edx和eax寄存器相加并存入eax寄存器。

9）mov　%eax,-0xc(%rbp)

把eax寄存器的值传送给变量c。

10）mov　-0xc(%rbp),%eax

把变量c的值传送给寄存器eax,eax作为返回值寄存器。

11）pop　%rbp

恢复上一栈帧的bp。

5.1.3　应用需要迁移的原因

1. 汇编代码角度

从5.1.2节的汇编代码分析可以看出来,同样的C语言代码,在编译成不同架构下的程序后,得到的汇编代码是不同的,这些不同点主要体现在以下3个方面:

1）处理器指令

以简单的给变量赋值操作为例:

(1) 鲲鹏架构。

首先使用mov指令把操作数传送给寄存器,然后使用ldr指令把寄存器的值传送到内存。

(2) x86架构。

直接使用movl指令把操作数传送到内存。

鲲鹏架构和x86架构在具体的处理器指令设计上,是有重大区别的,同样的功能,两个架构的处理器指令实现的方式可能不一样。

2）寄存器

这一点也比较明显,两个架构下的寄存器不管是数量还是功能都有所不同。

(1) 鲲鹏架构。

ARM 64有34个寄存器,其中编号x0～x29是通用寄存器,x30为程序链接寄存器,

x31 比较特殊，它有时候用作 xzr 零寄存器，有时候是栈指针寄存器 sp，两者不能在一条指令里共存，另外两个寄存器是程序计数器 PC、状态寄存器 CPSR。

ARM 64 寄存器 x0～x30 及 xzr 零寄存器都是 64 位的，它们的低 32 位构成了 32 位寄存器，分别用 w0～w30 表示，用 wzr 表示 32 位下的零寄存器。

除此之外，ARM 64 还有浮点寄存器和向量寄存器，此处就不详细介绍了。

（2）x86-64 架构。

x86-64 架构下有 16 个 64 位的通用寄存器，这些寄存器支持访问低位，例如访问低 8 位、低 16 位、低 32 位。

16 个寄存器的名称和用途如表 5-2 所示。

表 5-2　x86-64 寄存器用途

63～0 位	31～0 位	15～0 位	7～0 位	用途
%rax	%eax	%ax	%al	返回值
%rbx	%ebx	%bx	%bl	被调用者保存
%rcx	%ecx	%cx	%cl	第 4 个参数
%rdx	%edx	%dx	%dl	第 3 个参数
%rsi	%esi	%si	%sil	第 2 个参数
%rdi	%edi	%di	%dil	第 1 个参数
%rbp	%ebp	%bp	%bpl	被调用者保存
%rsp	%esp	%sp	%spl	栈指针
%r8	%r8d	%r8w	%r8b	第 5 个参数
%r9	%r9d	%r9w	%r9b	第 6 个参数
%r10	%r10d	%r10w	%r10b	调用者保存
%r11	%r11d	%r11w	%r11b	调用者保存
%r12	%r12d	%r12w	%r12b	被调用者保存
%r13	%r13d	%r13w	%r13b	被调用者保存
%r14	%r14d	%r14w	%r14b	被调用者保存
%r15	%r15d	%r15w	%r15b	被调用者保存

3）指令长度

x86 下指令的长度是不一样的，短的只有 1 字节，而长的却有 15 字节，给寻址带来了一定的不便。

鲲鹏架构指令长度为固定的 32 位（ARM 工作状态），寻址方便，效率较高。

2. 计算技术栈角度

对于常用的使用高级语言编写的应用，计算技术栈一般分为两类，一类是编译型语言，另一类是解释型语言，这两种技术栈的示意图如图 5-3 所示。

1）编译型语言

编译型语言的代表是 C/C++等语言，使用编译型语言编写的源代码经过一系列的编译过程，最终才能生成可执行程序，大体流程如图 5-4 所示。

图 5-3　技术栈示意图

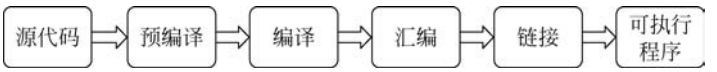

图 5-4　C 编译过程

因为不同架构下指令集不同，导致依赖于指令集的二进制机器码、汇编语言都不同，所以同一段程序，在不同的架构下，需要最终编译成和架构相适应的二进制机器码。这也就解释了 5.1 节最后两个实验不能成功的原因，毕竟架构不同，在特定架构下编译的二进制机器码，它需要执行的指令在另一个架构下根本就不存在。

2）解释型语言

解释型语言的代表是 Java/Python 等语言，从图 5-3 可以看出，Java 的源代码会被编译成字节码，字节码运行在 Java 虚拟机 JVM 上，JVM 具有与平台架构无关的指令集，同一段 Java 代码在不同的架构下都可以编译成相同的字节码。JVM 对字节码进行解释，转换为物理 CPU 对应的机器码进行实际执行。因为不同的指令集架构可以适配不同的 JVM 实现，所以 Java 等解释型语言只需编译一次，就可以到处运行，不同架构下的 JVM 屏蔽了指令集之间的差异。

如果一个应用是使用纯 Java 语言编写的，理论上基本可以跨平台运行，但是实际情况比较复杂，有些 Java 应用会引用 so 库文件，这些 so 库文件很有可能是通过编译型语言例如 C 来编写的，这时候就要考虑 so 库文件的移植。

5.2　编译型语言应用移植

根据 5.1 节的介绍，我们知道了采用编译型语言开发的应用在从一个架构迁移到另一个架构时需要移植，那么究竟怎么移植呢？先通过一个简单的 C 程序演示一下通常的移植过程。

8min

5.2.1 移植过程演示

1. x86 架构下运行效果

步骤 1：登录 x86 架构服务器，进入/data/code/文件夹，创建文件 transfer.c，指令如下：

```
cd /data/code/
vi transfer.c
```

步骤 2：在文件 transfer.c 中输入如下代码：

```
//Chapter5/transfer.c
#include <stdio.h>
int main(void)
{
    char a = -1;
    printf("a = %d\n",a);
    return 0;
}
```

这段代码按照预期，应该会打印出 a=-1 来。

步骤 3：编译 transfer.c，指令如下：

```
gcc -o transfer transfer.c
```

步骤 4：执行 transfer，查看输出结果，指令如下：

```
./transfer
```

输出结果如图 5-5 所示。

```
[root@ecs-x86 code]# ./transfer
a=-1
[root@ecs-x86 code]#
```

图 5-5 x86 架构输出结果

可以看出来，打印输出的结果就是希望看到的-1。

2. 鲲鹏架构下运行效果

在鲲鹏架构下同样编译并执行。

步骤 1：登录鲲鹏架构服务器，进入/data/code/文件夹，创建文件 transfer.c，指令如下：

```
cd /data/code/
vi transfer.c
```

步骤 2：在文件 transfer.c 中输入和 x86 架构下一样的代码，代码如下：

```
//Chapter5/transfer.c
#include <stdio.h>
int main(void)
{
    char a = -1;
    printf("a = %d\n",a);
    return 0;
}
```

步骤 3：编译 transfer.c，指令如下：

```
aarch64-redhat-Linux-gcc -o transfer transfer.c
```

步骤 4：执行 transfer，查看输出结果，指令如下：

```
./transfer
```

输出结果如图 5-6 所示。

这个结果和预期的结果不一样，输出的是 255。

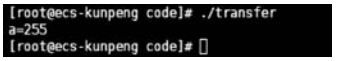

图 5-6　鲲鹏架构输出结果

3. 原因分析

－1 的二进制原码是 10000001，它的补码是除了符号外取反加 1，最后补码就是 11111111。在 x86 架构下，char 默认是有符号的，所以打印的时候正常打印－1，但是在鲲鹏架构下 char 默认是无符号的，这个二进制的 11111111 正好就是无符号的 255。

所以，出现这种情况的原因就是 x86 架构和鲲鹏架构对于 char 的默认处理不一样，一个是默认有符号，另一个是默认无符号。

4. 处理方式

对于这种情况，有两种处理方式，一种是修改源代码，把数据类型指定为有符号型，另一种就是在编译时指定参数，把默认无符号型改成默认为有符号型，下面演示说明。

1）修改编译参数

修改编译参数比较简单，只需要在 gcc 后面加入-fsigned-char 编译选项即可，命令如下：

```
aarch64-redhat-Linux-gcc -fsigned-char -o transfer transfer.c
```

需要注意的是，该选项会把源代码中所有的 char 类型变量都当作有符号类型，如果只是更改其中部分 char 变量，这种方式就不合适了。

修改后的执行效果如图 5-7 所示，可以看到得到了期望的输出。

2）修改源代码

修改源代码虽然有点复杂，但灵活性比较高，可以一劳永逸地解决问题，修改方法就是

```
[root@ecs-kunpeng code]# aarch64-redhat-linux-gcc -fsigned-char -o transfer transfer.c
[root@ecs-kunpeng code]# ./transfer
a=-1
[root@ecs-kunpeng code]#
```

图 5-7　鲲鹏架构修改编译参数后的输出结果

把 char 类型改成 signed char 即可。在/data/code/目录下新建 transfer_new.c，然后输入修改后的代码：

```
//Chapter5/transfer_new.c
#include <stdio.h>
int main(void)
{
    signed char a = -1;
    printf("a = %d\n",a);
    return 0;
}
```

编译的时候不用指定编译选项，直接编译即可，最后执行效果如图 5-8 所示，可以看到也得到了期望的输出。

```
[root@ecs-kunpeng data]# aarch64-redhat-linux-gcc -o transfer_new transfer_new.c
[root@ecs-kunpeng data]# ./transfer_new
a=-1
[root@ecs-kunpeng data]#
```

图 5-8　鲲鹏架构下 transfer_new 执行输出结果

5.2.2　移植总结

对于 C/C++为代表的编译型语言来说，移植方法一般包括两种，也就是源代码修改和编译选项修改，有时候使用其中一种方式，有时候两种方式需要同时使用。

1. 源代码修改

对于源代码修改的场景，主要分为以下几种：

1）对内嵌的汇编指令的修改

正如在 5.1 节介绍的那样，x86 架构和鲲鹏在汇编指令上完全不同，对于在编译型语言中内嵌的汇编指令，需要根据目标架构指令集的具体情况进行有针对性更改。

2）char 数据类型的修改

使用 5.2.1 节介绍的方法，把 char 数据类型更改为 signed char。

3）双精度浮点型转整型溢出处理的修改

详细见 5.2.3 节内容。

2. 编译选项修改

1）char 数据类型默认无符号

对于 char 数据类型默认无符号问题，除了上面介绍的直接更改代码方法外，还可以使

用修改编译选项来解决，就是在编译时指定-fsigned-char 选项，这一点在 5.2.1 节也演示了。当然，如果是从鲲鹏架构移植到 x86 架构，可以使用-funsigned-char 来保持 char 类型为无符号型。

2）指定编译 64 位应用

在指定应用编译为 64 位时，x86 架构下需要指定编译选项-m64，但是在鲲鹏架构下不支持这个选项，替代的选择是-mabi＝lp64。需要注意的是，不是所有的 gcc 版本都支持-mabi＝lp64 选项，只有在 4.9.4 及以后的版本才支持。

3）目标指令集

在执行编译时，可以指定目标的指令集，使用的是-march 编译选项，该选项可以指定的指令集类型非常多，大概有几十种，但是鲲鹏架构目前只对应一种类型，就是 ARMv8-a，在编译时可以指定编译选项为-march ＝ARMv8-a。

4）编译宏

gcc 预先内置了各种宏定义，这些宏定义有一部分是和架构有关系的，同时，gcc 也支持自定义宏，在代码里，也可以通过宏来对不同的架构做区别处理，例如，同样一段功能，通过宏定义的控制条件来区分不同架构下的实现方式，这样在编译时指定宏定义，就可以在同一个代码文件下适配多种架构。

5.2.3　移植常见问题

在进行实际的代码移植过程中，因为环境的多样性，会遇到多种问题，下面按照源码修改和嵌入式汇编两个类别，分别对可能出现的问题进行分析并给出建议的解决方法，需要特别注意的是，给出的解决方法仅供参考，本书不对实际的使用作任何担保。

1. 源码修改类问题

1）代码中汇编指令需要重写

■　现象描述：

ARM 的汇编语言与 x86 完全不同，需要重写，涉及使用嵌入汇编的代码，都需要针对 ARM 进行配套修改。

■　处理步骤：

需要重新实现汇编代码段。

■　示例：

在 x86 架构下，示例代码如下：

```
static inline long atomic64_add_and_return(long i, atomic64_t * v)
{
    long i = i;
    asm_volatile_(
        "lock ; "
        "xaddq %0, %1;"
        : "=r"(i)
```

```
        : "m"(v->counter), "0"(i));
    return i + __i;
}
static inline void prefetch(void * x)
{
    asm volatile("prefetcht0 %0" ::"m"( * (unsigned long * )x));
}
```

在鲲鹏平台下,使用 gcc 内置函数实现,示例代码如下:

```
static __inline__ long atomic64_add_and_return(long i, atomic64_t * v)
{
    return __sync_add_and_fetch(&((v)->counter), i);
}
#define prefetch(_x) __builtin_prefetch(_x)
```

以__sync_add_and_fetch 为例,编译后其反汇编对应代码如下:

```
<__sync_add_and_fetch>:
ldxr x2, [x0]
add x2, x2, x1
stlxr w3, x2, [x0]
```

2) 快速移植内联 SSE/SSE2 应用

■ 现象描述:

部分应用采用了 gcc 封装的用 SSE/SSE2 实现的函数,但是 gcc 目前没有提供对应的鲲鹏平台版本,需要实现对应函数。

■ 处理步骤:

目前已有开源代码实现了部分鲲鹏平台的函数,代码下载网址:https://GitHub.com/open-estuary/sse2neon.git,使用方法如下:

步骤 1:将已下载项目中的 SSE2NEON.h 文件复制到待移植项目中。

步骤 2 :在源文件中删除如下代码:

```
#include <xmmintrin.h>
#include <emmintrin.h>
```

步骤 3:在源代码中包含头文件 SSE2NEON.h。

3) 对结构体中的变量进行原子操作时程序异常 coredump

■ 现象描述:

程序调用原子操作函数对结构体中的变量进行原子操作,程序 coredump,堆栈如下:

```
Program received signal SIGBUS, Bus error.
0x000000000040083c in main () at /root/test/src/main.c:19
19 __sync_add_and_fetch(&a.count, step);
(gdb) disassemble
Dump of assembler code for function main:
0x0000000000400824 <+0>: sub sp, sp, #0x10
0x0000000000400828 <+4>: mov x0, #0x1 //#1
0x000000000040082c <+8>: str x0, [sp, #8]
0x0000000000400830 <+12>: adrp x0, 0x420000 <__libc_start_main@got.plt>
0x0000000000400834 <+16>: add x0, x0, #0x31      //将变量的地址放入 x0 寄存器
0x0000000000400838 <+20>: ldr x1, [sp, #8]       //指定 ldxr 取数据的长度(此处为 8 字节)
=> 0x000000000040083c <+24>: ldxr x2, [x0]       //ldxr 从 x0 寄存器指向的内存地址中取值
0x0000000000400840 <+28>: add x2, x2, x1
0x0000000000400844 <+32>: stlxr w3, x2, [x0]
0x0000000000400848 <+36>: cbnz w3, 0x40083c <main+24>
0x000000000040084c <+40>: dmb ish
0x0000000000400850 <+44>: mov w0, #0x0 //#0
0x0000000000400854 <+48>: add sp, sp, #0x10
0x0000000000400858 <+52>: ret
End of assembler dump.
(gdb) p /x $x0
$4 = 0x420039 //x0 寄存器存放的变量地址不在 8 字节地址对齐处
```

■　问题原因：

鲲鹏平台对变量的原子操作、锁操作等用到了 ldaxr、stlxr 等指令，这些指令要求变量地址必须按变量长度对齐，否则执行指令会触发异常，导致程序 coredump。一般是因为代码中对结构体进行强制字节对齐，导致变量地址不在对齐位置上，对这些变量进行原子操作、锁操作等会触发问题。

■　处理步骤：

代码中搜索 #pragma pack 关键字（该宏改变了编译器默认的对齐方式），找到使用了字节对齐的结构体，如果结构体中变量会被作为原子操作、自旋锁、互斥锁、信号量、读写锁的输入参数，则需要修改代码保证这些变量按变量长度对齐。

4）核数目硬编码

■　问题原因：

鲲鹏服务器相对于 x86 服务器，CPU 核数会有变化，如果模块代码针对处理器核数目硬编码，则会造成无法充分利用系统能力的情况，例如 CPU 核的利用率差异大或者绑核出现跨 numa 的情况。

■　处理步骤：

可以通过搜索代码中的绑核接口（sched_setaffinity）来排查绑核的实现是否存在 CPU 核数硬编码的情况。如果存在，则根据鲲鹏服务器实际核数进行修改，消除硬编码，可通过接口 sysconf(_SC_NPROCESSORS_CONF)获取实际核数再进行绑核。

5）双精度浮点型转整型时数据溢出，与 x86 平台表现不一致

■ 现象描述：

C/C++双精度浮点型数转整型数据时，如果超出了整型的取值范围，鲲鹏平台的表现与 x86 平台的表现不同。

```
long aa = (long)0x7FFFFFFFFFFFFFF;
long bb;
bb = (long)(aa * (double)10); //long -> double -> long
//x86: aa = 9223372036854775807, bb = -9223372036854775808
//arm64:aa = 9223372036854775807, bb = 9223372036854775807
```

■ 问题原因：

在两个平台下，是两套 CPU 架构，其中的算数逻辑单元的实现可能会有差异，操作系统、编译器的实现都会有所不同。x86（指令集）中的浮点到整型的转换指令，定义了一个 indefinite integer value——“不确定数值”（64bit：0x8000000000000000），大多数情况下 x86 平台确实都在遵循这个原则，但是在从 double 向无符号整型转换时，又出现了不同的结果。鲲鹏的处理则非常清晰和简单，在上溢出或下溢出时，保留整型能表示的最大值或最小值，开发者并不会面对不确定或无法预期的结果。

■ 处理步骤：

参考如下数据转换的表格，调整代码中的实现。

double 型数据向 long 转换，如表 5-3 所示。

表 5-3 double 型数据向 long 转换

CPU	double 值	转换为 long 变量保留值	说 明
x86	正值超出 long 范围	0x8000000000000000	indefinite integer value
x86	负值超出 long 范围	0x8000000000000000	indefinite integer value
鲲鹏	正值超出 long 范围	0x7FFFFFFFFFFFFFFF	鲲鹏为 long 变量赋值最大的正数
鲲鹏	负值超出 long 范围	0x8000000000000000	鲲鹏为 long 变量赋值最小的负数

double 型数据向 unsigned long 转换，如表 5-4 所示。

表 5-4 double 型数据向 unsigned long 转换

CPU	double 值	转换为 unsigned long 变量值	说 明
x86	正值超出 long 范围	0x0000000000000000	x86 为 long 变量赋值最小值 0
x86	负值超出 long 范围	0x8000000000000000	indefinite integer value
鲲鹏	正值超出 long 范围	0xFFFFFFFFFFFFFFFF	鲲鹏为 unsigned long 变量赋值最大值
鲲鹏	负值超出 long 范围	0x0000000000000000	鲲鹏为 unsigned long 变量赋值最小值

double 型数据向 int 转换，如表 5-5 所示。

表 5-5　double 型数据向 int 转换

CPU	double 值	转换为 int 变量值	说　明
x86	正值超出 int 范围	0x80000000	indefinite integer value
x86	负值超出 int 范围	0x80000000	indefinite integer value
鲲鹏	正值超出 int 范围	0x7FFFFFFF	鲲鹏为 int 变量赋值最大的正数
鲲鹏	负值超出 int 范围	0x80000000	鲲鹏为 int 变量赋值最小的负数

double 型数据向 unsigned int 转换，如表 5-6 所示。

表 5-6　double 型数据向 unsigned int 转换

CPU	double 值	转换为 unsigned int 变量值	说　明
x86	正值超出 unsigned int 范围	double 整数部分对 2^32 取余	x86 为 unsigned int 变量赋值最小的负值
x86	负值超出 unsigned int 范围	double 整数部分对 2^32 取余	x86 为 unsigned int 变量赋值最小的负值
鲲鹏	正值超出 unsigned int 范围	0xFFFFFFFF	鲲鹏为 unsigned int 变量赋值最大的正数
鲲鹏	负值超出 unsigned int 范围	0x00000000	鲲鹏为 unsigned int 变量赋值最小的负数

2. 嵌入式汇编类问题

1）替换 x86 pause 汇编指令

■　现象描述：

编译报错：Error：unknown mnemonic 'pause' -- 'pause'。

■　问题原因：

pause 指令给处理器提供提示，以提高 spin-wait 循环的性能，需替换为鲲鹏平台的 yield 指令。

■　处理步骤：

x86 平台实现样例：

```
static inline void PauseCPU()
{
    __asm__ __volatile__("pause"::: "memory");
}
```

鲲鹏平台实现样例：

```
static inline void PauseCPU()
{
    __asm__ __volatile__("yield"::: "memory");
}
```

2）替换 x86 pcmpestri 汇编指令

■ 现象描述：

编译报错 Error：unknown mnemonic 'pcmpestri'-- 'pcmpestri'。

■ 问题原因：

与 pcmpestrm 指令类似，pcmpestri 也是 x86 SSE4 指令集中的指令。根据指令介绍，其用途是根据指定的比较模式，判断字符串 str2 的字节是否在 str1 中出现，返回匹配到的位置索引（首个匹配结果为 0 的位置）。同样，对于该指令，需要彻底了解其功能，通过 C 代码重新实现其功能。

指令介绍：

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE4_2&expand=834

https://docs.microsoft.com/zh-cn/previous-versions/visualstudio/visualstudio-2010/bb531465(v=vs.100)。

■ 处理步骤：

如下代码段是 Impala 中对 pcmpestri 指令的调用，该调用参考 Intel 的_mm_cmpestri 接口实现将 pcmpestri 指令封装成 SSE4_cmpestri，代码如下：

```
template < int MODE >
static inline int SSE4_cmpestri(__m128i str1, int len1, __m128i str2, int len2)
{
    int result;
    __asm__ __volatile__("pcmpestri %5, %2, %1"
                         : "=c"(result)
                         : "x"(str1), "xm"(str2), "a"(len1),
                           "d"(len2), "i"(MODE)
                         : "cc");
    return result;
}
```

从指令介绍中看，不同的模式所执行的操作差异较大，完全实现指令功能所需代码行太多。结合代码中对接口的调用，实际使用到的模式为 PCMPSTR_EQUAL_EACH | PCMPSTR_UBYTE_OPS | PCMPSTR_NEG_POLARITY。即按照字节长度进行匹配，对 str1 与 str2 做对应位置字符是否相等判断，若相等，则将对应 bit 位置置 1，最后输出首次出现 1 的位置。根据该思路进行代码实现，代码如下：

```
#include < arm_neon.h >
template < int MODE >
static inline int SSE4_cmpestri(int32x4_t str1, int len1, int32x4_t str2, int len2)
{
    __oword a, b;
```

```
    a.m128i = str1;
    b.m128i = str2;
    int len_s, len_l;
    if (len1 > len2)
    {
        len_s = len2;
        len_l = len1;
    }
    else
    {
        len_s = len1;
        len_l = len2;
    }
    //本例替换的模式 STRCMP_MODE =
    //PCMPSTR_EQUAL_EACH | PCMPSTR_UBYTE_OPS | PCMPSTR_NEG_POLARITY
    int result;
    int i;
    for (i = 0; i < len_s; i++)
    {
        if (a.m128i_u8[i] == b.m128i_u8[i])
        {
            break;
        }
    }
    result = i;
    if (result == len_s)
    {
        result = len_l;
    }
    return result;
}
```

3）替换 x86 movqu 汇编指令

■　现象描述：

编译报错：unknown mnemonic 'movqu'-- 'movqu'。

■　问题原因：

movqu 为 x86 指令集中的指令，在鲲鹏上无法使用。该指令可以实现寄存器到寄存器，寄存器到地址的数据复制。x86 上 movqu 指令用法有两种：

第一种是将 xmm2 寄存器或者 128 位内存地址的内容复制到 xmm1 寄存器，代码如下：

```
MOVDQU xmm1, xmm2/m128
```

第二种是将 xmm1 寄存器的内容复制到 128 位内存地址或者 xmm2 寄存器，代码如下：

```
MOVDQU xmm2/m128, xmm1
```

参考资料：https://x86.puri.sm/html/file_module_x86_id_184.html。

■ 处理步骤：

对于第一种调用，可以用 NEON 指令 ld1 替代：

ld1 指令 Load multiple 1-element structures to one，two，three or four registers。

```
LD1 { Vt.T }, [Xn|SP]
```

可参考指令集手册的 9.98 节，下载网址：

http://infocenter.arm.com/help/topic/com.arm.doc.dui0802a/DUI0802A_armasm_reference_guide.pdf。

对于第二种调用，可以用 st1 指令来替代：

st1 指令 Store multiple 1-element structures from one，two three or four registers.

```
ST1 { Vt.T }, [Xn|SP]
```

可参考指令集手册的 9.202 节，下载网址：

http://infocenter.arm.com/help/topic/com.arm.doc.dui0802a/DUI0802A_armasm_reference_guide.pdf。

以下是一个简单的示例，代码如下：

```
/*x86 调用*/
void add_x86_asm(int *result, int *a, int *b, int len)
{
    __asm__("\n\t"
            "1: \n\t"
            "movdqu (%[a]), %%xmm0 \n\t"
            "movdqu (%[b]), %%xmm1 \n\t"
            "pand %%xmm0, %%xmm1 \n\t"
            "movdqu %%xmm1, (%[result]) \n\t"
            :[result]"+r"(result)
            :[a]"r"(a), [b]"r"(b), [len]"r"(len)
            :"memory", "xmm0", "xmm1");
    return;
}
/*鲲鹏调用*/
void and_neon_asm(int *result, int *a, int *b, int len)
{
    int num = {0};
    __asm__("\n\t"
            "1: \n\t"
```

```
            "ld1 {v0.16b}, [ %[a]], #16 \n\t"
            "ld1 {v1.16b}, [ %[b]], #16 \n\t"
            "and v0.16b, v0.16b, v1.16b \n\t"
            "subs %[len], %[len], #4 \n\t"
            "st1 {v0.16b}, [ %[result]], #16 \n\t"
            "bgt 1b \n\t"
            : [ result ] "+r"(result)
            : [ a ] "r"(a), [ b ] "r"(b), [ len ] "r"(len)
            : "memory", "v0", "v1");
    return;
}
```

4）替换 x86 pand 汇编指令

■　现象描述：

编译报错：unknown mnemonic 'pand' --'pand'。

■　问题原因：

pand 是 x86 指令集中的指令，无法在鲲鹏设备上使用。其功能是按位进行 and 运算，使用方法有两种：

第一种用法是对寄存器 xmm2 或内存地址中 128 位内容与 xmm1 进行按位与运算，结果存放于 xmm2 中，指令用法如下：

```
PAND xmm1, xmm2/m128
```

第二种用法是对寄存器 mm2 或内存地址中 64 位内容与 mm1 进行按位与运算，结果存放于 mm2 中，指令用法如下：

```
PAND mm1, mm2/m64
```

指令使用方法参考：https://c9x.me/x86/html/file_module_x86_id_230.html。

■　处理步骤：

对于以上两种情况，在鲲鹏上均可以用 NEON 指令 AND 替换，采用 64 或者 128 位长度的向量寄存器存放数据，代码如下：

```
AND Vd.<T>, Vn.<T>, Vm.<T>
```

Bitwise AND (vector). Where <T> is 8B or 16B (though an assembler shouldaccept any valid format)。

其中 Vn、Vm 为待操作的寄存器，Vd 是目的寄存器，<T>即是选择寄存器位数。

参考指令集手册的 9.7 节，下载网址：http://infocenter.arm.com/help/topic/com.arm.doc.dui0802a/DUI0802A_armasm_reference_guide.pdf。

下面是一个简单的使用 NEON 指令 AND 对数据进行按位与操作的过程，供参考，代码如下：

```
/*
 * 功能:对数组 a 和数组 b 进行按位与运算,结果放置到 result 中
 * neon 指令每次处理 16 字节长度数据,所以数据长度为 16 字节整数倍
 */
void and_neon_asm(int *result, int *a, int *b, int len)
{
    __asm__("\n\t"
            "1: \n\t"
            "ld1 {v0.16b}, [%[a]], #16 \n\t"
            "ld1 {v1.16b}, [%[b]], #16 \n\t"
            "and v0.16b, v0.16b, v1.16b \n\t"
            "subs %[len], %[len], #4 \n\t"
            "st1 {v0.16b}, [%[result]], #16 \n\t"
            "bgt 1b \n\t"
            : [result] "+r"(result)
            : [a] "r"(a), [b] "r"(b), [len] "r"(len)
            : "memory", "v0", "v1");
    return;
}
```

5）替换 x86 pxor 汇编指令

■ 现象描述：

编译报错：unknown mnemonic 'pxor'--'pxor'。

■ 问题原因：

pxor 是 x86 指令集中的指令，无法在鲲鹏设备上使用。其功能是按位进行 xor 运算，使用方法有两种：

第一种用法是对寄存器 xmm2 或内存地址中 128 位内容与 xmm1 进行按位异或运算，结果存放于 xmm1 中，指令用法如下：

```
PXOR xmm1, xmm2/m128
```

第二种用法是对寄存器 mm2 或内存地址中 64 位内容与 mm1 进行按位异或运算，结果存放于 mm1 中，指令用法如下：

```
PXOR mm1, mm2/m64
```

指令用法参考网址：https://c9x.me/x86/html/file_module_x86_id_272.html。

■ 处理步骤：

对于以上两种情况，在鲲鹏上均可以用 NEON 指令 EOR 替换，采用 64 或者 128 位长

度的向量寄存器存放数据，代码如下：

```
EOR Vd.<T>, Vn.<T>, Vm.<T>
```

Bitwise exclusive OR (vector). Where <T> is 8B or 16B (an assembler shouldaccept any valid arrangement)。其中 Vn、Vm 为待操作的寄存器，Vd 是目的寄存器，<T>是选择寄存器位数。参考指令集手册的 9.29 节，下载网址为 http://infocenter.arm.com/help/topic/com.arm.doc.dui0802a/DUI0802A_armasm_reference_guide.pdf。

下面是一个简单的使用 NEON 指令 EOR 对数据进行按位异或操作的过程，参考代码如下：

```
/*
 * 功能:对数组 a 和数组 b 进行按位异或运算,结果放置到 result 中
 * NEON 指令每次处理 16 字节长度数据,所以数据长度为 16 字节的整数倍
 */
void eor_neon_asm(int *result, int *a, int *b, int len)
{
    __asm__("\n\t"
            "1: \n\t"
            "ld1 {v0.16b}, [%[a]], #16 \n\t"
            "ld1 {v1.16b}, [%[b]], #16 \n\t"
            "eor v0.16b, v0.16b, v1.16b \n\t"
            "subs %[len], %[len], #4 \n\t"
            "st1 {v0.16b}, [%[result]], #16 \n\t"
            "bgt 1b \n\t"
            :[result] "+r"(result)
            :[a] "r"(a), [b] "r"(b), [len] "r"(len)
            :"memory", "v0", "v1");
    return;
}
```

6）替换 x86 pshufb 指令

■　现象描述：

编译报错：unknown mnemonic 'pshufb'--'pshufb'。

■　问题原因：

pshufb(Packed Shuffle Bytes)指令的功能是根据第二个操作数指定的控制掩码对第一个操作数执行散列操作，产生一个组合数。它是 x86 平台的汇编指令，在鲲鹏平台上需要进行替换。x86 上的指令用法如下：

```
pshufb xmm1, xmm2/m128
```

■　处理步骤：

pshufb 指令对应的 SSE intrinsic 函数是 _mm_shuffle_epi8，因此 pshufb 在鲲鹏上的替换可以分为两步：

步骤 1：将 pshufb 汇编指令替换成 SSE intrinsic。

x86 上实现样例，代码如下：

```
__asm__("pshufb %1, %0" : "+x" (mmdesc) : "xm" (shuf_mask));
```

在鲲鹏上先替换成 SSE intrinsic 函数，代码如下：

```
_mm_shuffle_epi8(mmdesc, shuf_mask);
```

步骤 2：移植内联 SSE 函数_mm_shuffle_epi8。gcc 目前没有提供对应的鲲鹏平台版本，因此需要实现对应函数，代码如下：

```
FORCE_INLINE __m128i _mm_shuffle_epi8(__m128i a, __m128i b)
{
    uint8x16_t tbl = vreinterpretq_u8_m128i(a);
    uint8x16_t idx = vreinterpretq_u8_m128i(b);
    uint8_t __attribute__((aligned(16))) mask[16] = {0x8F, 0x8F, 0x8F, 0x8F, 0x8F, 0x8F,
0x8F, 0x8F,
                                                     0x8F, 0x8F, 0x8F, 0x8F, 0x8F, 0x8F,
0x8F, 0x8F};
    uint8x16_t idx_masked = vandq_u8(idx, vld1q_u8(mask));
    return vreinterpretq_m128i_u8(vqtbl1q_u8(tbl, idx_masked));
}
```

7）替换 x86 cpuid 汇编指令

■ 现象描述：

编译报错：/tmp/ccfaVZfw.s：Assembler messages:/tmp/ccfaVZfw.s:34：Error：unknown mnemonic 'cpuid' -- 'cpuid'。

■ 问题原因：

cpuid 是 x86 平台上专有的获取 cpuid 信息的汇编指令，在鲲鹏平台上需要重写。在鲲鹏平台上，midr_el1 寄存器里存放的是 cpuid 信息，可以通过读寄存器获取 cpuid。

■ 处理步骤：

x86 实现样例，代码如下：

```
unsigned int s1 = 0;
unsigned int s2 = 0;
char cpu[32] = {0};
asm volatile(
    "movl $0x01, %%eax; \n\t"
    "xorl %%edx, %%edx; \n\t"
```

```
    "cpuid; \n\t"
    "movl %%edx, %0; \n\t"
    "movl %%eax, %1; \n\t"
    : "=m"(s1), "=m"(s2));
snprintf(cpu, sizeof(cpu), "%08X%08X", htonl(s2), htonl(s1));
```

midr_el1 是 64 位寄存器，其中高 32 位为预留位，其值为 0。读出来是一个 32 位的值。鲲鹏平台上可替换成的代码如下：

```
unsigned int s1 = 0;
unsigned int s2 = 0;
char cpu[32] = {0};
asm volatile(
    "mrs %0, midr_el1"
    : "=r"(s1)
    :
    : "memory");
snprintf(cpu, sizeof(cpu), "%08X%08X", htonl(s1), htonl(s2));
```

8）替换 x86 xchgl 汇编指令

■　现象描述：

编译报错：{standard input}：Assembler messages:{standard input}:1222：Error：unknown mnemonic 'xchgl'--'xchgl x1,[x19,112]'。

■　问题原因：

xchgl 是 x86 上的汇编指令，作用是交换寄存器/内存变量和寄存器的值，如果交换的两个变量中有内存变量，则会对内存变量增加原子锁操作。鲲鹏上可用 GCC 的原子操作接口__atomic_exchange_n 替换。__atomic_exchange_n 的第 3 个入参是内存屏障类型，使用者可以根据自身代码逻辑选择不同的屏障。当对多线程访问临界区的逻辑不清晰时，建议使用__ATOMIC_SEQ_CST 屏障，避免由屏障使用不当带来一致性问题。

■　处理步骤：

x86 实现样例，代码如下：

```
inline int nBasicAtomicInt::fetchAndStoreOrdered(int newValue)
{
    /* 原子操作，把_value 的值和 newValue 交换，且返回_value 原来的值 */
    asm volatile("xchgl %0, %1"
                 : "=r"(newValue), "+m"(m_value)
                 : "0"(newValue)
                 : "memory");
    return newValue;
}
```

鲲鹏上可替换成的代码如下：

```
inline int nBasicAtomicInt::fetchAndStoreOrdered(int newValue)
{
    /* 原子操作，把_value 的值和 newValue 交换，且返回_value 原来的值 */
    return __atomic_exchange_n(&_q_value, newValue, __ATOMIC_SEQ_CST);
}
```

9）替换 x86 cmpxchgl 汇编指令

■ 现象描述：

编译报错：{standard input}：Assembler messages:{standard input}:1222：Error：unknown mnemonic 'cmpxchgl '

■ 问题原因：

与 xchgl 类似，cmpxchgl 是 x86 上的汇编指令，其作用是比较并交换操作数。鲲鹏上无对应指令，可用 GCC 的原子操作接口__atomic_compare_exchange_n 进行替换。

■ 处理步骤：

x86 实现样例，代码如下：

```
inline bool nBasicAtomicInt::testAndSetOrdered(int expectedValue, int newValue)
{
    unsigned char ret;
    /* 原子操作，原来 m_value 的值如果等于 expectedValue，则把 newValue
 * 载入 m_value，且返回 ret = true；如果不等于，则 m_value 的值不变，且返回 ret = false */
    asm volatile("lock\n"
                 "cmpxchgl %3, %2\n"
                 "sete %1\n"
                 : "=a"(newValue), "=qm"(ret), "+m"(m_value)
                 : "r"(newValue), "0"(expectedValue)
                 : "memory");
    return ret != 0;
}
```

鲲鹏上可替换成的代码如下：

```
inline bool nBasicAtomicInt::testAndSetOrdered(int expectedValue, int newValue)
{
    unsigned char ret;
    /* 原子操作，原来 m_value 的值如果等于 expectedValue，则把 newValue 载入_value，且返回
ret = true；如果不等于，则 m_value 的值不变，且返回 ret = false */
    return __atomic_compare_exchange_n(&m_value, &expectedValue, newValue, false,
                                       __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST);
}
```

10）替换 x86 rep 汇编指令

■ 现象描述：

编译报错：Error：unknown mnemonic 'rep' -- 'rep'。

■　问题原因：

rep 为 x86 平台的重复执行指令，需替换为鲲鹏平台的 rept 指令。

■　处理步骤：

修改方法参考如下：

x86 实现样例，代码如下：

```
#define nop __asm__ __volatile__("rep;nop": : :"memory")
```

鲲鹏平台实现样例，本样例实现空指令，参数 n 为循环次数，代码如下：

```
#define __nops(n) ".rept " #n "\nnop\n.endr\n"
#define nops(n) asm volatile(__nops(n))
```

11）替换 x86 bswap 汇编指令

■　现象描述：

编译报错：Error：unknown mnemonic 'bswap' -- 'bswap x3'。

■　问题原因：

bswap 是 x86 平台的字节序反序指令，需替换为鲲鹏平台的 rev 指令。

■　处理步骤：

在 x86 平台下的实现，代码如下：

```
static inline uint32_t bswap(uint32_t val)
{
    __asm__("bswap %0"
             : "=r"(val)
             : "0"(val));
    retrun val;
}
```

在鲲鹏平台下的实现，代码如下：

```
static inline uint32_t bswap(uint32_t val)
{
    __asm__("rev %w[dst], %w[src]"
             : [dst] "=r"(val)
             : [src] "r"(val));
    return val;
}
```

12）替换 x86 crc32 汇编指令

■　现象描述：

编译错误：Error：unknown mnemonic 'crc32q'-- 'crc32q (x3),x2'或 operand 1should be an integer register -- 'crc32b [sp,11],x0'或 unrecognized command line option '-msse4.2'。

■ 问题原因：

x86 平台使用的是 crc32b、crc32w、crc32l、crc32q 汇编指令完成 CRC32C 校验值计算功能，而鲲鹏平台使用 crc32cb、crc32ch、crc32cw、crc32cx 4 个汇编指令完成 CRC32C 校验值计算功能。

■ 处理步骤：

使用 crc32cb、crc32ch、crc32cw、crc32cx 取代 x86 的 CRC32 系列汇编指令，替换方法如表 5-7 所示，并在编译时添加编译参数-march=armv8+crc。

表 5-7 替换方法

指令	输入数据位宽/b	备 注
crc32cb	8	适用输入数据位宽为 8bit，可用于替换 x86 的 crc32b 汇编指令
crc32ch	16	适用输入数据位宽为 16bit，可用于替换 x86 的 crc32w 汇编指令
crc32cw	32	适用输入数据位宽为 32bit，可用于替换 x86 的 crc32l 汇编指令
crc32cx	64	适用输入数据位宽为 64bit，可用于替换 x86 的 crc32q 汇编指令

■ 示例：

在 x86 平台下的实现，代码如下：

```
static inline uint32_t crc32_u8(uint32_t crc, uint8_t v)
{
    __asm__("crc32b %1, %0"
            : "+r"(crc)
            : "rm"(v));
    return crc;
}
static inline uint32_t crc32_u16(uint32_t crc, uint16_t v)
{
    __asm__("crc32w %1, %0"
            : "+r"(crc)
            : "rm"(v));
    return crc;
}
static inline uint32_t crc32_u32(uint32_t crc, uint32_t v)
{
    __asm__("crc32l %1, %0"
            : "+r"(crc)
            : "rm"(v));
    return crc;
}
```

```
static inline uint32_t crc32_u64(uint32_t crc, uint64_t v)
{
    uint64_t result = crc;
    __asm__("crc32q %1, %0"
            : "+r"(result)
            : "rm"(v));
    return result;
}
```

在鲲鹏平台下的实现，代码如下：

```
static inline uint32_t crc32_u8(uint32_t crc, uint8_t value)
{
    __asm__("crc32cb %w[c], %w[c], %w[v]"
            : [c] "+r"(crc)
            : [v] "r"(value));
    return crc;
}
static inline uint32_t crc32_u16(uint32_t crc, uint16_t value)
{
    __asm__("crc32ch %w[c], %w[c], %w[v]"
            : [c] "+r"(crc)
            : [v] "r"(value));
    return crc;
}
static inline uint32_t crc32_u32(uint32_t crc, uint32_t value)
{
    __asm__("crc32cw %w[c], %w[c], %w[v]"
            : [c] "+r"(crc)
            : [v] "r"(value));
    return crc;
}
static inline uint32_t crc32_u64(uint32_t crc, uint64_t value)
{
    __asm__("crc32cx %w[c], %w[c], %x[v]"
            : [c] "+r"(crc)
            : [v] "r"(value));
    return crc;
}
```

13）替换 x86 rdtsc 汇编指令

■　现象描述：

编译报错：error：impossible constraint in 'asm'__asm__ __volatile__("rdtsc" ："=a" (lo)，"=d" (hi))；

■ 问题原因：

TSC 是时间戳计数器的缩写，它是 Pentium 兼容处理器中的一个计数器，它记录自启动以来处理器消耗的时钟周期数。在每个时钟到来时，该计数器自动加 1。因为 TSC 随着处理器周期速率的变化而变化，所以它提供了非常高的精确度。它经常被用来分析和检测代码。x86 平台 TSC 的值可以通过 rdtsc 指令来读取，而鲲鹏平台需要使用类似算法实现。

■ 处理步骤：

x86 平台实现样例，代码如下：

```
static inline uint64_t Rdtsc()
{
    uint32_t lo, hi;
    __asm__ __volatile__("rdtsc"
                         : "=a"(lo), "=d"(hi));
    return (uint64_t)hi << 32 | lo;
}
```

鲲鹏平台实现样例：

方法一：使用 Linux 提供的获取时间函数 clock_gettime 进行近似替换，代码如下：

```
#include <time.h>
static inline uint64_t Rdtsc()
{
    struct timespec tmp;
    clock_gettime(CLOCK_MONOTONIC, &tmp);
    return tmp.tv_sec * 2400000000 + (uint64_t)tmp.tv_nsec * 2.4; //2400000000 和 2.4 基
                                                                   //于服务器主频而定
}
```

方法二：鲲鹏有 Performance Monitors Control Register 系列寄存器，其中 PMCCNTR_EL0 类似于 x86 的 TSC 寄存器。但默认情况下用户态是不可读的，需要内核态使能后才能读取。具体可参考网址 http://iLinuxKernel.com/?p=1755。

a. 下载 read aarch64 TSC（http://www.iLinuxKernel.com/files/aarch64_tsc.tar.bz2），解压压缩包，在 aarch64_tsc 目录下执行 make 命令，安装相应内核驱动，生成文件，生成文件中包括一个文件名为 pmu.ko 的文件。

b. 执行 insmod pmu.ko 命令安装内核模块，使能内核态（初次执行即可）。

c. 代码替换。

示例代码如下：

```
static inline uint64_t Rdtsc()
{
    uint64_t count_num;
```

```
    uint64_t Current_Speed = 2400; //Current Speed = 2400MHz
    uint64_t External_Clock = 100; //External Clock = 100MHz
    __asm__ __volatile__("mrs %0, cntvct_el0"
                         : "=r"(count_num));
    return count_num * (Current_Speed / External_Clock);
}
```

其中 Cent Speed 和 External Clock 的值可由以下命令获取：

```
dmidecode |grep MHz
```

14）替换 x86 popcntq 汇编指令

■　现象描述：

编译报错：Error：unknown mnemonic 'popcnt'-- 'popcnt [sp,8],x0'。

■　问题原因：

popcnt 为 x86 平台的位 1 计数指令，鲲鹏平台无对应指令，需使用替换算法实现。

■　处理步骤：

x86 平台实现样例，代码如下：

```
static inline uint64_t POPCNT_popcnt_u64(uint64_t a)
{
    uint64_t result;
    __asm__("popcnt %1, %0"
            : "=r"(result)
            : "mr"(a)
            : "cc");
    return result;
}
```

鲲鹏平台实现样例，代码如下：

```
#include <arm_neon.h>
static inline uint64_t POPCNT_popcnt_u64(uint64_t x)
{
    uint64_t count_result = 0;
    uint64_t count[1];
    uint8x8_t input_val, count8x8_val;
    uint16x4_t count16x4_val;
    uint32x2_t count32x2_val;
    uint64x1_t count64x1_val;
    input_val = vld1_u8((unsigned char *)&x);
    count8x8_val = vcnt_u8(input_val);
    count16x4_val = vpaddl_u8(count8x8_val);
```

```
    count32x2_val = vpaddl_u16(count16x4_val);
    count64x1_val = vpaddl_u32(count32x2_val);
    vst1_u64(count, count64x1_val);
    count_result = count[0];
    return count_result;
}
```

15）替换 x86 atomic 原子操作函数

■ 现象描述：

部分应用会通过封装汇编指令实现原子操作，如原子加及原子减。由于指令集差异，x86 上所使用的原子操作指令在 ARM 平台并不能保证原子性，因此需要进行相应替换。

① atomic_add 指令

函数功能：对整数变量进行原子加。

处理步骤：

x86 平台实现样例，代码如下：

```
static inline void atomic_add(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "addl %1, %0"
                 : "+m"(v->counter)
                 : "ir"(i));
}
```

在鲲鹏上进行替换：

第 1 种方法：使用 GCC 自带原子操作替换，代码如下：

```
static inline void atomic_add(atomic_t *v)
{
    __sync_add_and_fetch(&((*v).counter), 1);
}
```

第 2 种方法：使用内联汇编替换，代码如下：

```
static inline void atomic_add(atomic_t *v)
{
    unsigned int tmp;
    int result, i;
    i = 1;
    __asm__ volatile(" prfm pstl1strm, %2\n"
                     "1: ldaxr %w0, %2\n"        //加载数据到寄存器
                     " add %w0, %w0, %w3\n"      //加操作
                     " stlxr %w1, %w0, %2\n"     //加后的数据写入内存并判断是否写入成功
                     " cbnz %w1, 1b"             //若写入内存失败，则重新执行加操作
```

```
                        : "=&r"(result), "=&r"(tmp), "+Q"(v->counter)
                        : "Ir"(i));
}
```

② atomic_sub 指令

函数功能：对整数变量进行原子减。

处理步骤：

x86 平台实现样例，代码如下：

```
static inline void atomic_sub(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "subl %1, %0"
                 : "+m"(v->counter)
                 : "ir"(i));
}
```

在鲲鹏上进行替换：

第 1 种方法：使用 GCC 自带原子操作替换，代码如下：

```
static inline void atomic_sub(atomic_t *v)
{
    __sync_sub_and_fetch(&((*v).counter), 1);
}
```

第 2 种方法：使用内联汇编替换，代码如下：

```
static inline void atomic_sub(atomic_t *v)
{
    unsigned int tmp;
    int result, i;
    i = 1;
    __asm__ volatile(" prfm pstl1strm, %2\n"
                     "1: ldaxr %w0, %2\n"     //加载数据到寄存器
                     " sub %w0, %w0, %w3\n"   //减操作
                     " stlxr %w1, %w0, %2\n" //减后的数据写入内存并判断是否写入成功
                     " cbnz %w1, 1b"          //若写入内存失败，则重新执行加操作
                     : "=&r"(result), "=&r"(tmp), "+Q"(v->counter)
                     : "Ir"(i));
}
```

③ atomic_dec_and_test 指令

函数说明：对整数进行减操作，并判断执行原子减后结果是否为 0。

处理步骤：

x86 平台实现样例，代码如下：

```
static inline int atomic_dec_and_test(atomic_t * v)
{
    unsigned char c;
    asm volatile(LOCK_PREFIX "decl % 0; sete % 1"
                 : " + m"(v - > counter), " = qm"(c)
                 :
                 : "memory");
    return c != 0;
}
```

在鲲鹏上进行替换：

第 1 种方法：使用 GCC 自带原子操作函数替换，代码如下：

```
static inline int atomic_dec_and_test(atomic_t * v)
{
    __sync_sub_and_fetch(&(( * v).counter), 1);
    return ( * v).counter == 0;
}
```

第 2 种方法：使用内联汇编替换，代码如下：

```
static inline int atomic_dec_and_test(atomic_t * v)
{
    unsigned long tmp;
    int result, val, i;
    i = 1;
    prefetchw(&v - > counter);
    __asm__ volatile(
        "\n\t"
        "1: ldaxr % 0, [ % 4]\n\t"
        " sub % 1, % 0, % 5\n\t"
        " stlxr % w2, % 1, [ % 4]\n\t"
        " cbnz % w2, 1b\n\t "
        : " = &r"(result), " = &r"(val), " = &r"(tmp), " + Qo"(v - > counter)
        : "r"(&v - > counter), "Ir"(i)
        : "cc");
    return ( * v).counter == 0;
}
```

④ atomic_inc_and_test 指令

函数说明：对整数进行加操作，并判断返回结果是否为 0。

处理步骤：

x86 平台实现样例，代码如下：

```
static inline int atomic_inc_and_test(atomic_t * v)
{
    unsigned char c;
    asm volatile(LOCK_PREFIX "incl % 0; sete % 1"
                 : " + m"(v -> counter), " = qm"(c)
                 :
                 : "memory");
    return c != 0;
}
```

在鲲鹏上进行替换：

第 1 种方法：使用 GCC 自带原子操作函数替换，代码如下：

```
static inline int atomic_inc_and_test(atomic_t * v)
{
    __sync_add_and_fetch(&(( * v).counter), 1);
    return ( * v).counter == 0;
}
```

第 2 种方法：使用内联汇编替换，代码如下：

```
# define prefetchw(x) __builtin_prefetch(x, 1)
static inline int atomic_inc_and_test(atomic_t * v)
{
    unsigned long tmp;
    int result, val, i;
    i = 1;
    prefetchw(&v -> counter);
    __asm__ volatile(
        "\n\t"
        "1: ldaxr % 0, [ % 4]\n\t"          //@result, tmp
        " add % 1, % 0, % 5\n\t"            //@result,
        " stlxr % w2, % 1, [ % 4]\n\t"      //@tmp, result,tmp
        " cbnz % w2, 1b\n\t "               //@tmp
        : " = &r"(result), " = &r"(val), " = &r"(tmp), " + Qo"(v -> counter)
        : "r"(&v -> counter), "Ir"(i)
        : "cc");
    return ( * v).counter == 0;
}
```

⑤ atomic64_add_and_return 指令

函数说明：对两个长整数进行加操作，并将结果作为返回值返回。

处理步骤：

需要重新实现汇编代码段。

在 x86 平台实现样例，代码如下：

```
static inline long atomic64_add_and_return(long i, atomic64_t * v)
{
    long i = i;
    asm_volatile_(
        "lock ; "
        "xaddq % 0, % 1;"
        : " = r"(i)
        : "m"(v -> counter), "0"(i));
    return i + __i;
}
static inline void prefetch(void * x)
{
    asm volatile("prefetcht0 % 0" ::"m"( * (unsigned long * )x));
}
```

在鲲鹏平台下，使用 GCC 内置函数实现，代码如下：

```
static __inline__ long atomic64_add_and_return(long i, atomic64_t * v)
{
    return __sync_add_and_fetch(&((v) -> counter), i);
}
```

16）替换 x86 pcmpestrm 汇编指令

■ 现象描述：

编译报错 Error：unknown mnemonic 'pcmpestrm' -- 'pcmpestrm'。

■ 问题原因：

pcmpestrm 指令是 x86 指令集中 SSE4 中的指令。根据指令介绍，其用途是根据指定的比较模式，判断字符串 str2 的字节是否在字符串 str1 中出现，将每个字节的对比结果返回（最大长度为 16 字节）。该指令是典型的 x86 复杂指令，通过一条指令即可完成复杂的字符串匹配功能，鲲鹏架构中无类似实现。对于这种指令，需要彻底了解其功能，通过 C 代码重新实现其功能。

指令介绍：

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE4_2&expand=835。

https://docs.microsoft.com/zh-cn/previous-versions/visualstudio/visualstudio-2010/bb514080(v=vs.100)。

■　处理步骤：

以下代码段是 Impala 中对 pcmpestrm 指令的调用，该调用参考 Intel 的_mm_cmpestrm 接口实现将 pcmpestrm 指令封装成 SSE4_cmpestrm，代码如下：

```
template < int MODE >
static inline __m128i SSE4_cmpestrm(__m128i str1, int len1, __m128i str2, int len2)
{
#ifdef __clang__
    register volatile __m128i result asm("xmm0");
    __asm__ __volatile__("pcmpestrm % 5, % 2, % 1"
                         : " = x"(result)
                         : "x"(str1), "xm"(str2), "a"(len1),
                           "d"(len2), "i"(MODE)
                         : "cc");
#else
    __m128i result;
    __asm__ __volatile__("pcmpestrm % 5, % 2, % 1"
                         : " = Yz"(result)
                         : "x"(str1), "xm"(str2),
                           "a"(len1), "d"(len2), "i"(MODE)
                         : "cc");
#endif
    return result;
}
```

从指令介绍中看，不同的模式所执行的操作差异较大，完全实现指令功能所需代码行太多。结合代码中对接口的调用，实际使用到的模式为 PCMPSTR_EQUAL_ANY | PCMPSTR_UBYTE_OPS。即按照字节长度进行匹配，对比字符串 str2 中的每个字符是否在字符串 str1 中出现，若出现，则将对应 bit 位置置 1。

根据识别到的功能进行代码实现，代码如下：

```
#include < arm_neon.h >
typedefunion __attribute__((aligned(16))) __oword
{
    int32x4_t m128i;
    uint8_tm128i_u8[16];
}
__oword;
template < intMODE >
staticinlineuint16_tSSE4_cmpestrm(int32x4_tstr1, intlen1, int32x4_tstr2, intlen2)
{
    __oword a, b;
    a.m128i = str1;
    b.m128i = str2;
    uint16_t result = 0;
```

```
    uint16_t i = 0;
    uint16_t j = 0;
    //Impala 中用到的模式 STRCHR_MODE = PCMPSTR_EQUAL_ANY | PCMPSTR_UBYTE_OPS
    for (i = 0; i < len2; i++)
    {
        for (j = 0; j < len1; j++)
        {
            if (a.m128i_u8[j] == b.m128i_u8[i])
            {
                result | = (1 << i);
            }
        }
    }
    return result;
}
```

注意：无直接替代指令的场景，需要结合指令功能、所需功能共同分析，切忌生搬硬套直接代码复制及替换。

注意：5.2.3 节移植常见问题内容引用自华为《鲲鹏代码迁移参考手册》4.2 节和 4.3 节，网址为 http://ic-openlabs.huawei.com/chat/download/鲲鹏代码迁移参考手册.pdf。

5.3 解释型语言应用移植

对于以 Java 语言为代表的解释型语言来说，应用迁移相对简单一些，因为 Java 的虚拟机 JVM 屏蔽了不同处理器之间指令集架构的区别，所以，纯 Java 语言编写的程序不需要重新编译，而对于调用了编译型语言 so 库的应用来说，需要重新编译。下面通过具体示例，来演示一下纯 Java 语言应用迁移和包含了编译型语言的 Java 应用迁移。

5.3.1 纯 Java 语言应用迁移

通过编写一个简单的输入及输出的纯 Java 应用，分别在 x86 架构和鲲鹏架构下运行，看一看需要哪些步骤。

步骤 1：登录 x86 架构服务器，安装 openjdk 1.8，在命令行输入命令如下：

```
yum install -y java-1.8.0-openjdk
```

如果安装了其他版本号的 JDK 也是可以的，这段代码对 JDK 版本没有特别要求，常用的版本都可以，安装成功后可以通过命令查看版本信息，查看命令如下：

```
[root@ecs-x86 code]# java -version
openjdk version "1.8.0_272"
```

```
OpenJDK RunTime Environment (build 1.8.0_272-b10)
OpenJDK 64-Bit Server VM (build 25.272-b10, mixed mode)
```

步骤 2：因为要编写 Java 的代码并且进行编译，所以需要安装 Java 的开发环境，使用 yum 安装 java-devel，命令如下：

```
yum install -y java-devel
```

步骤 3：进入/data/code/文件夹，创建文件 IoTest.java，指令如下：

```
cd /data/code/
vi IoTest.java
```

步骤 4：在 IoTest.java 中输入的代码如下：

```
//Chapter5/IoTest.java
import java.util.Scanner;

public class IoTest{
    public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      String input = sc.nextLine();
      System.out.println(input);
      sc.close();
    }
}
```

该段代码的作用是接受用户的输入，然后把输入打印出来。

步骤 5：编译该 IoTest.java 文件，得到 IoTest.class 文件，命令如下：

```
javac IoTest.java
```

然后输入 ll 命令查看编译后的结果：

```
[root@ecs-x86 code]# ll
total 8
-rw-r--r-- 1 root root 592 Dec 1 21:08 IoTest.class
-rw-r--r-- 1 root root 208 Dec 1 21:08 IoTest.java
```

可以看到字节码文件 IoTest.class。

步骤 6：运行 IoTest.class，命令如下：

```
java IoTest
```

根据设计思路，输入“Hello Kunpeng!”，可以看到它同样会输出该字符串：

```
[root@ecs-x86 code]#java IoTest
Hello Kunpeng!
Hello Kunpeng!
```

在 x86 架构下编译及运行没问题了，把这个编译好的.class 文件复制到鲲鹏架构的服务器上，看一看是否可以正常运行。

步骤 7：使用 SCP 命令把 IoTest.class 复制到鲲鹏架构服务器上，命令及回显如下：

```
[root@ecs-x86 code]#scp IoTest.class root@172.16.0.155:/data/code
The authenticity of host '172.16.0.155 (172.16.0.155)' can't be established.
ECDSA key fingerprint is SHA256:kyhTbYOrHIYp/VNbnfMkTJmeV/wfxV2DTo6MCDc/Cos.
ECDSA key fingerprint is MD5:d8:91:a3:1d:26:9e:c1:f8:1b:a0:65:d0:9c:d1:c5:32.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.16.0.155' (ECDSA) to the list of known hosts.
root@172.16.0.155's password:
IoTest.class
100% 592   1.7MB/s 00:00
```

注意：使用的 IP 地址和密码需要根据实际的信息修改。

步骤 8：登录鲲鹏服务器，安装 aarch64 架构的 openjdk 1.8，命令如下：

```
yum install -y java-1.8.0-openjdk.aarch64
```

步骤 9：进入/data/code/文件夹，运行 IoTest.class，命令如下：

```
cd /data/code/
java IoTest
```

可以成功运行，同样输入“Hello Kunpeng!”，得到和 x86 架构下一样的运行结果：

```
[root@ecs-kunpeng code]#java IoTest
Hello Kunpeng!
Hello Kunpeng!
```


12min

5.3.2 依赖编译型语言的 Java 应用迁移


16min

对于大型项目来说，很多时候不仅仅使用一种语言来开发应用，有时候会使用多种语言进行混合编程，例如著名的开源项目 Netty，主体开发语言是 Java，但是在部分项目里还使用了 C 语言。出现这种情况的一个原因是 Netty 在 Linux 下的异步/非阻塞网络传输中，使用了 Epoll——一个基于 I/O 事件通知的高性能多路复用机制。Netty 是通过 JNI 方式提供 Native

Socket Transport 的，在 Netty 的 transport-native-epoll 项目中，有相关调用的 C 代码。

除此之外，在 Netty 的依赖项目 netty-tcnative-parent 中，也有 JNI 方式提供的 C 语言调用。

笔者负责的一款基于 Java 的物联网平台中也使用了 Netty，在进行应用迁移时经过多次尝试，解决了多个问题，最后迁移成功，这里通过 Netty 项目，演示一下依赖编译型语言的 Java 应用的迁移。

1. 迁移过程分析

Netty 是开源的项目，在获得所有的源代码后，可以通过对代码进行重新编译的方式来执行迁移。因为代码里有 Java 和 C 语言，并且 Netty 项目是通过 Pom 进行项目组织管理的，在迁移时不但要安装 C 的编译环境，还要安装 openjdk 和 Maven。

2. 安装依赖项

要安装的依赖项较多，大部分可以通过 yum 安装，命令如下：

```
yum install gcc gcc-c++ make cmake3 libtool autoconf automake ant wget git openssl openssl-
devel apr-devel ninja-build java-1.8.0-openjdk.aarch64 -y
```

安装依赖项的时间有点长，根据系统中已安装的软件情况，可能需要几分钟到十几分钟，最后回显如下：

```
Installed:
  ant.noarch 0:1.9.4-2.el7      apr-devel.aarch64 0:1.4.8-7.el7      cmake3.aarch64 0:3.
14.6-2.el7
  java-1.8.0-openjdk.aarch64 1:1.8.0.272.b10-1.el7_9   ninja-build.aarch64 0:1.7.2-2.el7

Dependency Installed:
  cmake3-data.noarch 0:3.14.6-2.el7 java-1.8.0-openjdk-devel.aarch64 1:1.8.0.272.b10
-1.el7_9
  java-1.8.0-openjdk-headless.aarch64 1:1.8.0.272.b10-1.el7_9 libarchive.aarch64 0:3.
1.2-14.el7_7
  libtirpc.aarch64 0:0.2.4-0.16.el7 libuv.aarch64 1:1.30.1-1.el7
  python3.aarch64 0:3.6.8-18.el7 python3-libs.aarch64 0:3.6.8-18.el7
  python3-pip.noarch 0:9.0.3-8.el7 python3-setuptools.noarch 0:39.2.0-10.el7
  rhash.aarch64 0:1.3.4-2.el7 xalan-j2.noarch 0:2.7.1-23.el7
  xerces-j2.noarch 0:2.11.0-17.el7_0 xml-commons-apis.noarch 0:1.4.01-16.el7
  xml-commons-resolver.noarch 0:1.2-15.el7

Updated:
  git.aarch64 0:1.8.3.1-23.el7_8

Dependency Updated:
  apr.aarch64 0:1.4.8-7.el7 perl-Git.noarch 0:1.8.3.1-23.el7_8

Complete!
```

因为后续步骤在编译 libressl-static 模块的时候需要 cmake 版本号大于 3，并且需要 ninja，这里提前做好软连接，命令如下：

```
ln -s /usr/bin/cmake3 /usr/bin/cmake
ln -s /usr/bin/ninja-build /usr/bin/ninja
```

3. 安装 Maven

Java 应用的编译打包需要 Maven，安装步骤如下：

步骤 1：下载 Maven 3.6.3 安装包，为了提高下载速度，可以使用国内的下载源，命令如下：

```
wget https://mirrors.tuna.tsinghua.edu.cn/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz
```

步骤 2：解压 Maven 安装包，命令如下：

```
tar -zvxf apache-maven-3.6.3-bin.tar.gz
```

步骤 3：移动 Maven 到指定目录，命令如下：

```
mv apache-maven-3.6.3 /opt/tools/
```

步骤 4：配置环境变量，修改/etc/profile 文件，在文件最后增加 Maven 的环境信息，增加的内容如下：

```
MAVEN_HOME=/opt/tools/apache-maven-3.6.3
PATH=$MAVEN_HOME/bin:$JAVA_HOME/bin:$PATH
export MAVEN_HOME JAVA_HOME PATH
```

步骤 5：使环境变量生效，命令如下：

```
source /etc/profile
```

步骤 6：由于 Maven 中央仓库的下载速度受限，所以这里配置 Maven 的镜像仓库网址为国内的镜像，要修改的配置文件路径为/opt/tools/apache-maven-3.6.3/conf/settings.xml，在该文件的<mirrors>节中添加新的镜像，添加的内容如下：

```
<mirror>
  <id>huaweimaven</id>
  <name>huawei maven</name>
  <URL>https://mirrors.huaweicloud.com/repository/maven/</URL>
  <mirrorOf>central</mirrorOf>
</mirror>
```

步骤 7：查看 Maven 是否安装成功，命令及回显如下：

```
[root@ecs-kunpeng ~]# mvn -v
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: /opt/tools/apache-maven-3.6.3
Java version: 11.0.8, vendor: N/A, RunTime: /usr/lib/jvm/java-11-openjdk-11.0.8.10-0.
el7_8.aarch64
Default locale: en_US, platform encoding: UTF-8
OS name: "Linux", version: "4.18.0-80.7.2.el7.aarch64", arch: "aarch64", family: "UNIX"
```

如果出现类似上面的回显，表示 Maven 安装配置成功了。

4. 处理鲲鹏架构中 char 类型为无符号型的默认设置

直接对代码中 char 类型进行更改风险较高，工作量也很大，这里通过设置 gcc 和 g++的编译选项来处理，也就是把这两个编译器的编译加上-fsigned-char 的选项。

1）修改 gcc 编译选项

步骤 1：确认 gcc 的位置，命令及回显如下：

```
[root@ecs-kunpeng ~]# command -v gcc
/usr/bin/gcc
```

根据系统不同，位置可能有差异，笔者本机的位置在/usr/bin/gcc。

步骤 2：修改 gcc 的名字为 gcc-ori，命令如下：

```
mv/usr/bin/gcc /usr/bin/gcc-ori
```

步骤 3：创建/usr/bin/gcc 文件，命令如下：

```
vi /usr/bin/gcc
```

步骤 4：编辑/bin/gcc 文件，输入内容如下：

```
#! /bin/sh
/usr/bin/gcc-ori -fsigned-char "$@"
```

步骤 5：给/bin/gcc 添加执行权限，命令如下：

```
chmod +x /usr/bin/gcc
```

步骤 6：查看 gcc 是否可以成功执行，命令及回显如下：

```
[root@ecs-kunpeng ~]# gcc --version
gcc-ori (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
Copyright (C) 2015 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

如果看到类似上面的回显,表示 gcc 修改成功了。

2）修改 g++编译选项

步骤 1：确认 g++的位置,命令及回显如下：

```
[root@ecs-kunpeng ~]# command -v g++
/usr/bin/g++
```

本机位置是/usr/bin/g++,不同的服务器位置可能不同。

步骤 2：修改 g++的名字为 g++-ori,命令如下：

```
mv /usr/bin/g++ /usr/bin/g++-ori
```

步骤 3：创建/usr/bin/g++文件,命令如下：

```
vi /usr/bin/g++
```

步骤 4：编辑/bin/g++文件,输入内容如下：

```
#! /bin/sh
/usr/bin/g++-ori -fsigned-char "$@"
```

步骤 5：给/bin/g++添加执行权限，命令如下：

```
chmod +x /usr/bin/g++
```

步骤 6：查看 g++是否可以成功执行，命令及回显如下：

```
[root@ecs-kunpeng ~]# g++ --version
g++-ori (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

如果看到类似上面的回显,表示 g++修改成功了。

5. 加速编译准备

在正式编译以前,需要先下载 3 个安装包。后面的编译过程需要从多个网站下载安装包,这些网站的服务器一般都在境外,下载速度较慢,可能会因为下载不成功导致编译失败。

步骤 1：下载 apr-1.6.5,进入/data/soft/文件夹,下载命令如下：

```
wget https://mirrors.tuna.tsinghua.edu.cn/apache/apr/apr-1.6.5.tar.gz
```

步骤 2：下载 libressl-3.1.1，下载命令如下：

```
wget https://mirrors.tuna.tsinghua.edu.cn/OpenBSD/LibreSSL/libressl-3.1.1.tar.gz
```

步骤 3：下载 openssl-1.1.1g，下载命令如下：

```
wget https://www.openssl.org/source/openssl-1.1.1g.tar.gz
```

6. 编译 netty-tcnative-2.0.34

步骤 1：进入/data/soft/下载 netty-tcnative 源码包，下载命令如下：

```
wget https://GitHub.com/netty/netty-tcnative/archive/netty-tcnative-parent-2.0.34.Final.tar.gz
```

步骤 2：解压源码包，并进入解压后目录，命令如下：

```
tar -zxvf netty-tcnative-parent-2.0.34.Final.tar.gz
cd netty-tcnative-netty-tcnative-parent-2.0.34.Final/
```

步骤 3：修改 pom 文件，注释掉对 apr 的下载，对于 2.0.34 版本来说，注释行在第 474 行，修改后的该段配置如下：

```
<configuration>
                    <target if="${linkStatic}">
                      <!-- Add the ant tasks from ant-contrib -->
                      <taskdef resource="net/sf/antcontrib/antcontrib.properties" />

                      <if>
                        <available file="${aprBuildDir}" />
                        <then>
                          <echo message="APR was already downloaded, skipping the build step." />
                        </then>
                        <else>
                          <echo message="Downloading and unpacking APR" />

                          <property name="aprTarGzFile" value="apr-${aprVersion}.tar.gz" />
                          <property name="aprTarFile" value="apr-${aprVersion}.tar" />
                         <!-- <get src="http://archive.apache.org/dist/apr/${aprTarGzFile}"
dest="${project.build.directory}/${aprTarGzFile}" verbose="on" /> -->
                           <checksum file="${project.build.directory}/${aprTarGzFile}"
algorithm="SHA-256" property="${aprSha256}" verifyProperty="isEqual" />
```

```
                        <gunzip src="${project.build.directory}/${aprTarGzFile}" dest=
"${project.build.directory}" />
                        <!-- Use the tar command (rather than the untar ant task) in order to
preserve file permissions. -->
                         <exec executable="tar" failonerror="true" dir="${project.
build.directory}/" resolveexecutable="true">
                          <arg line="xfvz ${aprTarGzFile}" />
                        </exec>
                      </else>
                    </if>
                  </target>
                </configuration>
```

注释掉该行后，mvn 编译时将不再从这里下载。

步骤 4：进入 libressl-static 目录，修改 pom 文件，注释掉对 libssl 的下载，注释行在第 263 行，修改后的该段配置如下：

```
<configuration>
                  <target>
                    <!-- Add the ant tasks from ant-contrib -->
                    <taskdef resource="net/sf/antcontrib/antcontrib.properties" />

                    <if>
                      <available file="${libresslCheckoutDir}" />
                      <then>
                          <echo message="LibreSSL was already downloaded, skipping the build
step." />
                      </then>
                      <else>
                        <echo message="Downloading LibreSSL" />

                         <!-- <get src="https://ftp.openbsd.org/pub/OpenBSD/LibreSSL/
${libresslArchive}" dest="${project.build.directory}/${libresslArchive}" verbose="on"
/> -->
                         <checksum file="${project.build.directory}/${libresslArchive}"
algorithm="SHA-256" property="${libresslSha256}" verifyProperty="isEqual" />
                         <exec executable="tar" failonerror="true" dir="${project.
build.directory}/" resolveexecutable="true">
                          <arg value="xfv" />
                          <arg value="${libresslArchive}" />
                        </exec>
                      </else>
                    </if>
                  </target>
                </configuration>
```

步骤 5：进入 openssl-static 目录，修改 pom 文件，注释掉对 openssl 的下载，注释行在第 334 行和第 338 行，修改后的该段配置如下：

```
<configuration>
                    <target>
                      <!-- Add the ant tasks from ant-contrib -->
                      <taskdef resource="net/sf/antcontrib/antcontrib.properties" />

                      <if>
                        <available file="${openssllBuildDir}" />
                        <then>
                        <echo message="OpenSSL was already downloaded, skipping the build step." />
                        </then>
                        <else>
                          <echo message="Downloading OpenSSL" />

                          <condition property="openssllFound">
                          <http URL="https://www.openssl.org/source/openssl-${openssllVersion}.
tar.gz" />
                          </condition>
                           <if>
                            <equals arg1="${openssllFound}" arg2="true" />
                            <then>
                              <!-- Download the openssl source. -->
                              <!-- <get src="https://www.openssl.org/source/openssl-
${openssllVersion}.tar.gz" dest="${project.build.directory}/openssl-
${openssllVersion}.tar.gz" verbose="on" /> -->
                            </then>
                            <else>
                             <!-- Download the openssl source from the old directory -->
                              <!-- <get src="https://www.openssl.org/source/old/
${openssllMinorVersion}/openssl-${openssllVersion}.tar.gz" dest="${project.build.
directory}/openssl-${openssllVersion}.tar.gz" verbose="on" /> -->
                            </else>
                          </if>
                          <checksum file="${project.build.directory}/openssl-${openssllVers
ion}.tar.gz" algorithm="SHA-256" property="${openssllSha256}" verifyProperty="isEqual" />

                          <!-- Use the tar command (rather than the untar ant task) in order to
preserve file permissions. -->
                           <exec executable="tar" failonerror="true" dir="${project.
build.directory}/" resolveexecutable="true">
                             <arg line="xfvz openssl-${openssllVersion}.tar.gz" />
                           </exec>
                        </else>
                      </if>
                    </target>
                  </configuration>
```

步骤 6：注释掉对 boringssl-static 的编译(在第 603 行)，因为 boringssl-static 需要从谷歌服务器获取资源，由于无法获取成功，这里就取消对它的编译，但不影响后续的使用(如果确实要用，可以把获取源码网址改为 GitHub 上的源码网址，这里就不演示了)。编辑源代码主目录的 pom 文件，修改后的该段配置如下：

```
        <modules>
          <module>openssl-dynamic</module>
          <module>openssl-static</module>
        <!--  <module>boringssl-static</module>  -->
          <module>libressl-static</module>
        </modules>
```

步骤 7：提前创建好 openssl-static 和 libressl-static 项目的 target 目录，命令如下：

```
mkdir /data/soft/netty-tcnative-netty-tcnative-parent-2.0.34.Final/openssl-static/
target/
mkdir /data/soft/netty-tcnative-netty-tcnative-parent-2.0.34.Final/libressl-static/
target/
```

步骤 8：复制预先下载的文件到 target 目录，命令如下：

```
cp /data/soft/apr-1.6.5.tar.gz /data/soft/netty-tcnative-netty-tcnative-parent-2.0.
34.Final/openssl-static/target/
cp /data/soft/openssl-1.1.1g.tar.gz /data/soft/netty-tcnative-netty-tcnative-parent-
2.0.34.Final/openssl-static/target/
cp /data/soft/apr-1.6.5.tar.gz /data/soft/netty-tcnative-netty-tcnative-parent-2.0.
34.Final/libressl-static/target/
cp /data/soft/libressl-3.1.1.tar.gz /data/soft/netty-tcnative-netty-tcnative-parent-
2.0.34.Final/libressl-static/target/
```

步骤 9：进入主目录，执行编译，命令如下：

```
cd /data/soft/netty-tcnative-netty-tcnative-parent-2.0.34.Final/
mvn install
```

最后编译成功的回显如下：

```
[INFO] Reactor Summary for Netty/TomcatNative [Parent] 2.0.34.Final:
[INFO]
[INFO] Netty/TomcatNative [Parent] ..................... SUCCESS [ 3.050 s]
[INFO] Netty/TomcatNative [OpenSSL - Dynamic] ........... SUCCESS [ 6.017 s]
[INFO] Netty/TomcatNative [OpenSSL - Static] ............ SUCCESS [ 23.875 s]
[INFO] Netty/TomcatNative [LibreSSL - Static] ........... SUCCESS [01:13 min]
[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:47 min
[INFO] Finished at: 2020-12-01T22:22:46+08:00
[INFO] ------------------------------------------------------------------------
```

7. 编译 netty-all-4.1.52

步骤 1：进入/data/soft/下载 netty-all 源码包，下载命令如下：

```
wget https://GitHub.com/netty/netty/archive/netty-4.1.52.Final.tar.gz
```

步骤 2：解压源码包，并进入解压后目录，命令如下：

```
tar -zxvf netty-4.1.52.Final.tar.gz
cd netty-netty-4.1.52.Final/
```

步骤 3：处理 jni.h 问题。在后续的编译中，可能会出现找不到 jni.h 和 jni_md.h 的错误，如图 5-9 所示。

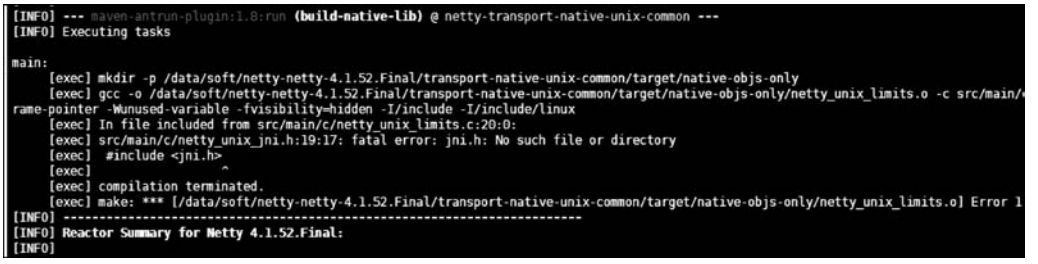

图 5-9　编译错误

出现这种错误的原因是 C 编译器找不到头文件的位置，所以需要直接告诉编译器头文件在哪里，就是通过 C 编译器选项 CFLAGS 传过去头文件的路径。本机的 jni.h 文件在/usr/lib/jvm/java/include 目录下，jni_md.h 文件在/usr/lib/jvm/java/include/Linux/目录下。编辑 transport-native-UNIX-common 下的 pom 文件，命令如下：

```
vim /data/soft/netty-netty-4.1.52.Final/transport-native-UNIX-common/pom.xml
```

要修改的 CFLAGS 选项在第 198 行和第 263 行，在值的后面加上头文件的位置，代码如下：

```
-I/usr/lib/jvm/java/include -I/usr/lib/jvm/java/include/Linux/
```

修改后的效果如下所示，注意修改后的字符串也是全部在 value 值的引号里面，代码如下：

```
<configuration>
                    <target>
                      <exec executable="${exe.make}" failonerror="true" resolveexecutable="
true">
                        <env key="CC" value="${exe.compiler}" />
                        <env key="AR" value="${exe.archiver}" />
                        <env key="LIB_DIR" value="${nativeLibOnlyDir}" />
                        <env key="OBJ_DIR" value="${nativeObjsOnlyDir}" />
                        <env key="JNI_PLATFORM" value="${jni.platform}" />
                        <env key="CFLAGS" value="-O3 -Werror -Wno-attributes -fPIC -
fno-omit-frame-pointer -Wunused-variable -fvisibility=hidden -I/usr/lib/jvm/java/
include -I/usr/lib/jvm/java/include/Linux/" />
                        <env key="LDFLAGS" value="-Wl,--no-as-needed -lrt" />
                        <env key="LIB_NAME" value="${nativeLibName}" />
                      </exec>
                    </target>
                  </configuration>
```

步骤 4：编译 netty-all，进入源码主目录，执行编译，命令如下：

```
mvn install -DskipTests
```

该命令将跳过测试过程，经过十几分钟的编译后，可以看到成功编译的回显如下所示：

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for Netty 4.1.52.Final:
[INFO]
[INFO] Netty/Dev-Tools ..................................................... SUCCESS
[ 3.865 s]
[INFO] Netty ............................................................... SUCCESS
[ 27.323 s]
[INFO] Netty/Common ........................................................ SUCCESS
[ 24.739 s]
[INFO] Netty/Buffer ........................................................ SUCCESS
[ 9.388 s]
[INFO] Netty/Resolver ...................................................... SUCCESS
[ 3.082 s]
[INFO] Netty/Transport ..................................................... SUCCESS
[ 10.379 s]
[INFO] Netty/Codec ......................................................... SUCCESS
[ 12.151 s]
[INFO] Netty/Codec/DNS ..................................................... SUCCESS
[ 5.560 s]
[INFO] Netty/Codec/HAProxy ................................................. SUCCESS
[ 3.563 s]
```

```
[INFO] Netty/Handler ..... SUCCESS
[ 10.909 s]
[INFO] Netty/Codec/HTTP ..... SUCCESS
[ 10.898 s]
[INFO] Netty/Codec/HTTP2 ..... SUCCESS
[ 11.358 s]
[INFO] Netty/Codec/Memcache ..... SUCCESS
[ 3.634 s]
[INFO] Netty/Codec/MQTT ..... SUCCESS
[ 8.049 s]
[INFO] Netty/Codec/Redis ..... SUCCESS
[ 3.343 s]
[INFO] Netty/Codec/SMTP ..... SUCCESS
[ 2.808 s]
[INFO] Netty/Codec/Socks ..... SUCCESS
[ 4.009 s]
[INFO] Netty/Codec/Stomp ..... SUCCESS
[ 3.104 s]
[INFO] Netty/Codec/XML ..... SUCCESS
[ 4.603 s]
[INFO] Netty/Handler/Proxy ..... SUCCESS
[ 4.802 s]
[INFO] Netty/Resolver/DNS ..... SUCCESS
[ 5.999 s]
[INFO] Netty/Transport/RXTX ..... SUCCESS
[ 1.964 s]
[INFO] Netty/Transport/SCTP ..... SUCCESS
[ 5.172 s]
[INFO] Netty/Transport/UDT ..... SUCCESS
[ 4.782 s]
[INFO] Netty/Example ..... SUCCESS
[ 6.317 s]
[INFO] Netty/Transport/Native/UNIX/Common ..... SUCCESS
[ 7.705 s]
[INFO] Netty/Testsuite ..... SUCCESS
[ 4.095 s]
[INFO] Netty/Transport/Native/UNIX/Common/Tests ..... SUCCESS
[ 2.327 s]
[INFO] Netty/Transport/Native/Epoll ..... SUCCESS
[ 18.026 s]
[INFO] Netty/Transport/Native/KQueue ..... SUCCESS
[ 4.162 s]
[INFO] Netty/Resolver/DNS/macOS ..... SUCCESS
[ 6.547 s]
[INFO] Netty/All-in-One ..... SUCCESS
[ 1.893 s]
```

```
[INFO] Netty/Tarball ..... SUCCESS
[ 0.299 s]
[INFO] Netty/Testsuite/Autobahn ..... SUCCESS
[ 4.716 s]
[INFO] Netty/Testsuite/Http2 ..... SUCCESS
[ 3.054 s]
[INFO] Netty/Testsuite/OSGI ..... SUCCESS
[ 5.161 s]
[INFO] Netty/Testsuite/Shading ..... SUCCESS
[ 6.329 s]
[INFO] Netty/Testsuite/NativeImage ..... SUCCESS
[ 8.765 s]
[INFO] Netty/Transport/BlockHound/Tests ..... SUCCESS
[ 2.125 s]
[INFO] Netty/Microbench ..... SUCCESS
[ 23.107 s]
[INFO] Netty/BOM ..... SUCCESS
[ 0.004 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 04:50 min
[INFO] Finished at: 2020-12-01T22:52:37+08:00
[INFO] ------------------------------------------------------------------------
```

鲲鹏架构的 jar 包就在各个项目的 target 目录下，例如 transport-native-epoll 项目下的 jar 包，显示如下：

```
[root@ecs-kunpeng transport-native-epoll]# cd target/
[root@ecs-kunpeng target]# ll
total 948
drwxr-xr-x 2    root root    4096 Dec    1 22:51    antrun
-rw-r--r-- 1  root root    14821 Dec    1 22:51    checkstyle-cachefile
-rw-r--r-- 1  root root    6059 Dec    1 22:51    checkstyle-checker.xml
-rw-r--r-- 1  root root    11358 Dec    1 22:51    checkstyle-header.txt
-rw-r--r-- 1  root root    81 Dec      1 22:51    checkstyle-result.xml
drwxr-xr-x 4    root root    4096 Dec    1 22:51    classes
drwxr-xr-x 2    root root    4096 Dec    1 22:51    dependency-maven-plugin-markers
drwxr-xr-x 3    root root    4096 Dec    1 22:51    dev-tools
drwxr-xr-x 4    root root    4096 Dec    1 22:51    generated-sources
drwxr-xr-x 3    root root    4096 Dec    1 22:51    generated-test-sources
drwxr-xr-x 2    root root    4096 Dec    1 22:51    japicmp
drwxr-xr-x 2    root root    4096 Dec    1 22:51    maven-archiver
drwxr-xr-x 3    root root    4096 Dec    1 22:51    maven-status
drwxr-xr-x 8    root root    4096 Dec    1 22:51    native-build
```

```
-rw-r--r--  1    root root    123723 Dec    1 22:51    netty-transport-native-epoll
-4.1.52.Final.jar
-rw-r--r--  1    root root    154034 Dec    1 22:51    netty-transport-native-epoll
-4.1.52.Final-Linux-aarch_64.jar
-rw-r--r--  1    root root    87454 Dec     1 22:51    netty-transport-native-epoll
-4.1.52.Final-sources.jar
-rw-r--r--  1    root root    276508 Dec    1 22:51    netty-transport-native-epoll
-4.1.52.Final-tests.jar
-rw-r--r--  1    root root    226782 Dec    1 22:51    netty-transport-native-epoll
-4.1.52.Final-test-sources.jar
drwxr-xr-x 4      root root    4096 Dec      1 22:51    test-classes
drwxr-xr-x 3      root root    4096 Dec      1 22:51    UNIX-common-lib
```

包含 aarch_64 的 jar 包就是鲲鹏架构下适用的 jar 包。

5.4　容器迁移

5.4.1　容器简介

容器是一种轻量级、可移植、自包含的软件打包技术，使应用程序几乎可以在任何地方以相同的方式运行。和虚拟机的硬件虚拟化不同，它基于操作系统级别的虚拟化技术，可以高效地利用服务器资源，具有如下特点：

1. 速度快

容器创建和启动速度都很快，基本可以做到秒级启动，这一点对于服务器的弹性使用很重要，在需要的时候可以随时快速创建容器，而在不需要时可以销毁容器释放资源。

2. 资源占用低

和虚拟机相比，容器没有 hypervisor 层，也没有自己的操作系统，大大降低了对内存、硬盘等资源的占用。

3. 标准化

容器基于开放技术标准，可以在所有主流的 Linux 发行版中运行。

4. 可移植性好

容器封装了所有运行应用程序所必需的相关细节，例如应用依赖及操作系统等，这就使得镜像从一个环境移植到另外一个环境更加灵活。

5. 安全性

容器之间的进程是相互隔离的，使用的资源亦是如此，一个容器的升级或者变化不会影响其他容器。

6. 镜像版本化

每个容器的镜像都由版本控制，可以追踪不同版本的容器，监控版本之间的差异。

5.4.2 容器和镜像、仓库之间的关系

在使用容器的时候，容器、镜像、仓库是关系非常紧密的几个概念，需要对比说明。

镜像：镜像是一个只读的模板，一个独立的文件系统，包括运行容器所需的数据，可以用来创建新的容器。镜像可以从仓库拉取，也可以推送镜像到仓库。

容器：容器是基于镜像创建的，是独立运行的一个或一组应用。同一个镜像可以创建多个容器，容器可以启动、暂停、停止、删除，但是对创建它的镜像没有影响。容器也可以保存当前状态，提交后可作为新镜像。

仓库：仓库是存储镜像的场所，可以查询、提交、提取镜像，目前最大的开源仓库是dockerhub。

从仓库提取镜像，然后使用镜像创建容器的关系如图 5-10 所示。

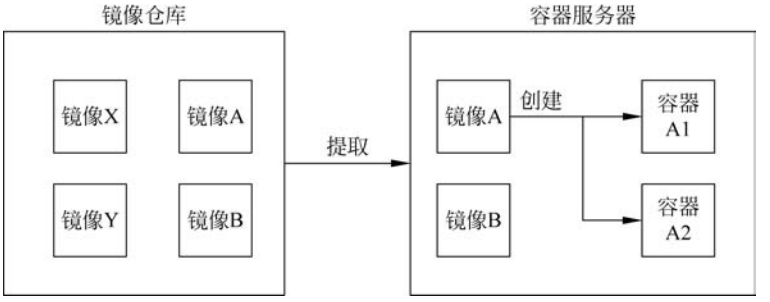

图 5-10 从仓库到容器

同样，从容器提交镜像，然后推送镜像到仓库的关系也可以用图 5-11 来表示。

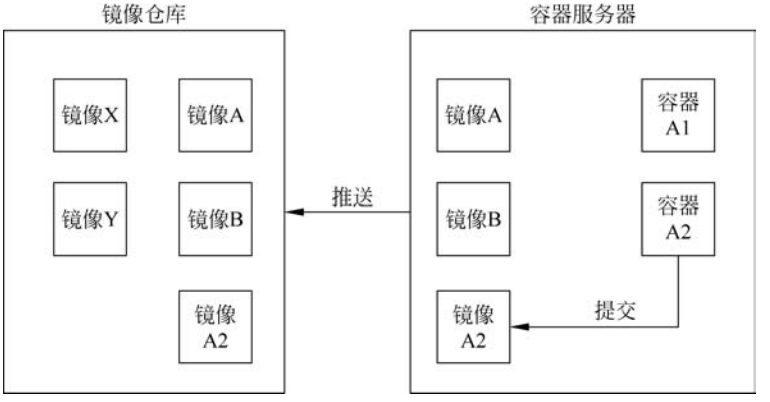

图 5-11 从容器到仓库

5.4.3 容器的基本操作

作为事实上的容器标准，Docker 被广泛使用，这里就以 Docker 为例，演示在鲲鹏架构

下容器的常用功能。

1. Docker 的安装

步骤 1：系统环境检查。Docker 对系统环境有一定的要求，对于 CentOS 7，要求 64 位系统，内核版本 3.10 或以上；对于 CentOS 6.5 或以上，要求 64 位系统，内核版本为 2.6.32-431 或者以上。检查内核版本，命令及回显如下：

```
[root@ecs-kunpeng ~]# uname -r
4.18.0-80.7.2.el7.aarch64
```

可以看到本机内核版本是 4.18，满足安装条件。

步骤 2：安装 Docker，命令如下：

```
yum install -y docker
```

安装成功的回显信息如下：

```
Installed:
  docker.aarch64 2:1.13.1-203.git0be3e21.el7.centos

Dependency Installed:
  PyYAML.aarch64 0:3.10-11.el7
  atomic-registries.aarch64 1:1.22.1-33.gitb507039.el7_8
  audit-libs-python.aarch64 0:2.8.5-4.el7
  checkpolicy.aarch64 0:2.5-8.el7
  container-seLinux.noarch 2:2.119.2-1.911c772.el7_8
  container-storage-setup.noarch 0:0.11.0-2.git5eaf76c.el7
  containers-common.aarch64 1:0.1.40-11.el7_8
  device-mapper-event.aarch64 7:1.02.170-6.el7
  device-mapper-event-libs.aarch64 7:1.02.170-6.el7
  device-mapper-persistent-data.aarch64 0:0.8.5-3.el7_9.2
  docker-client.aarch64 2:1.13.1-203.git0be3e21.el7.centos
  docker-common.aarch64 2:1.13.1-203.git0be3e21.el7.centos
  fuse-overlayfs.aarch64 0:0.7.2-6.el7_8
  fuse3-libs.aarch64 0:3.6.1-4.el7
  libaio.aarch64 0:0.3.109-13.el7
  libcgroup.aarch64 0:0.41-21.el7
  libnl.aarch64 0:1.1.4-3.el7
  libsemanage-python.aarch64 0:2.5-14.el7
  libyaml.aarch64 0:0.1.4-11.el7
  lvm2.aarch64 7:2.02.187-6.el7
  lvm2-libs.aarch64 7:2.02.187-6.el7
  oci-register-machine.aarch64 1:0-6.git2b44233.el7
  oci-systemd-hook.aarch64 1:0.2.0-1.git05e6923.el7_6
  oci-umount.aarch64 2:2.5-3.el7
```

```
  policycoreutils-python.aarch64 0:2.5-34.el7
  python-IPy.noarch 0:0.75-6.el7
  python-backports.aarch64 0:1.0-8.el7
  python-backports-ssl_match_hostname.noarch 0:3.5.0.1-1.el7
  python-dateutil.noarch 0:1.5-7.el7
  python-ethtool.aarch64 0:0.8-8.el7
  python-inotify.noarch 0:0.9.4-4.el7
  python-ipaddress.noarch 0:1.0.16-2.el7
  python-pytoml.noarch 0:0.1.14-1.git7dea353.el7
  python-setuptools.noarch 0:0.9.8-7.el7
  python-six.noarch 0:1.9.0-2.el7
  python-syspurpose.aarch64 0:1.24.42-1.el7.centos
  setools-libs.aarch64 0:3.3.8-4.el7
  slirp4netns.aarch64 0:0.4.3-4.el7_8
  subscription-manager.aarch64 0:1.24.42-1.el7.centos
  subscription-manager-rhsm.aarch64 0:1.24.42-1.el7.centos
  subscription-manager-rhsm-certificates.aarch64 0:1.24.42-1.el7.centos
  usermode.aarch64 0:1.111-6.el7
  yajl.aarch64 0:2.0.4-4.el7

Dependency Updated:
  device-mapper.aarch64 7:1.02.170-6.el7 device-mapper-libs.aarch64 7:1.02.170-6.el7
  policycoreutils.aarch64 0:2.5-34.el7

Complete!
```

步骤 3：启动 Docker 服务，命令如下：

```
systemctl start docker
```

步骤 4：查看 Docker 服务是否启动成功，命令及回显如下：

```
[root@ecs-kunpeng ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset:
disabled)
   Active: active (running) since Wed 2020-12-02 07:42:14 CST; 8s ago
     Docs: http://docs.docker.com
Main PID: 2219 (dockerd-current)
   CGroup: /system.slice/docker.service
           ├─2219 /usr/bin/dockerd-current --add-RunTime docker-runc=/usr/libexec/
docker/docker-runc-curr...
           ├─2227 /usr/bin/docker-containerd-current -l
UNIX:///var/run/docker/libcontainerd/docker-conta...
```

```
Dec 02 07:42:13 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
13.881610880+08:00" level=wa...ht"
Dec 02 07:42:13 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
13.881627730+08:00" level=wa...ce"
Dec 02 07:42:13 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
13.881953785+08:00" level=in...t."
Dec 02 07:42:13 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
13.930829233+08:00" level=in...se"
Dec 02 07:42:14 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
14.003211627+08:00" level=in...ss"
Dec 02 07:42:14 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
14.036062406+08:00" level=in...e."
Dec 02 07:42:14 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
14.114063481+08:00" level=in...on"
Dec 02 07:42:14 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
14.114099591+08:00" level=in...3.1
Dec 02 07:42:14 ecs-kunpeng dockerd-current[2219]: time="2020-12-02T07:42:
14.121241863+08:00" level=in...ck"
Dec 02 07:42:14 ecs-kunpeng systemd[1]: Started Docker Application Container Engine.
Hint: Some lines were ellipsized, use -l to show in full.
```

可以看到服务状态为 active(running),表示启动成功,可以正常运行了。

步骤 5：运行测试容器,命令及回显如下：

```
[root@ecs-kunpeng ~]# docker run hello-world
Unable to find image 'hello-world:latest' locally
Trying to pull repository docker.io/library/hello-world ...
latest: Pulling from docker.io/library/hello-world
256ab8fe8778: Pull complete
Digest: sha256:e7c70bb24b462baa86c102610182e3efcb12a04854e8c582838d92970a09f323
Status: Downloaded newer image for docker.io/hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.(arm64v8).
3. The Docker daemon created a new container from that image which runs the executable that
produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
```

```
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

如果看到类似上面的回显，表明镜像下载和容器运行都成功了。

2. 容器的使用

下面演示获取镜像并创建容器的过程，最后把容器提交成一个新的镜像。

步骤 1：获取 ARM64v8 架构下的精简的 Debian 镜像，命令及提取成功的回显如下：

```
[root@ecs-kunpeng ~]# docker pull arm64v8/debian:buster-slim
Trying to pull repository docker.io/arm64v8/debian ...
buster-slim: Pulling from docker.io/arm64v8/debian
29ade854e0dc: Pull complete
Digest: sha256:5d0f4e33abe44c7fca183c2c7ea7b2084d769aef3528ffd630f0dffda0784089
Status: Downloaded newer image for docker.io/arm64v8/debian:buster-slim
```

步骤 2：查看已经提取成功的镜像，命令如下：

```
[root@ecs-kunpeng ~]# docker images
REPOSITORY                  TAG           IMAGE ID       CREATED         SIZE
docker.io/arm64v8/debian    buster-slim   9db65bac9886   2 weeks ago     63.4 MB
docker.io/hello-world       latest        a29f45ccde2a   11 months ago   9.14 kB
```

可以看到刚提取的镜像 arm64v8/debian:buster-slim。

步骤 3：使用镜像 arm64v8/debian:buster-slim 启动一个容器并进入，容器名称为 debian4make，命令及回显如下：

```
[root@ecs-kunpeng ~]# docker run -it --name debian4make arm64v8/debian:buster-slim /
bin/bash
root@6145bfbeb7ec:/#
```

可以看到，启动后就直接进入了 id 为 6145bfbeb7ec 的容器内部。

步骤 4：进入容器后，需要安装后期编译 C 源代码会用到的一些依赖，命令如下：

```
apt-get update
apt-get install -y wget  gcc libc6-dev  make
```

安装成功后的回显如下：

```
126 added, 0 removed; done.
Setting up libgcc-8-dev:arm64 (8.3.0-6) ...
Setting up cpp (4:8.3.0-1) ...
```

```
Setting up libc6-dev:arm64 (2.28-10) ...
Setting up gcc-8 (8.3.0-6) ...
Setting up gcc (4:8.3.0-1) ...
Processing triggers for libc-bin (2.28-10) ...
Processing triggers for ca-certificates (20200601~deb10u1) ...
Updating certificates in /etc/ssl/certs...
0 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
```

步骤 5：安装成功后退出容器，命令如下：

```
Exit
```

查看容器状态，命令及回显如下：

```
[root@ecs-kunpeng ~]# docker ps -a
CONTAINER ID    IMAGE       COMMAND       CREATED       STATUS       PORTS       NAMES
6145bfbeb7ec    arm64v8/debian:buster-slim    "/bin/bash"    36 minutes ago
Exited (0) 20 seconds ago    debian4make
e963b90f37bb    hello-world    "/hello"    40 minutes ago    Exited (0) 40 minutes ago
fervent_austin
```

可以看到刚才运行的容器 debian4make 为 exited 状态。

步骤 6：使用 debian4make 创建一个新镜像，新镜像的名字为 arm64v8/debian4make，命令如下：

```
docker commit  -m "base image for make"debian4make arm64v8/debian4make
```

步骤 7：查看镜像列表，命令及回显如下：

```
[root@ecs-kunpeng ~]# docker images
REPOSITORY                TAG              IMAGE ID          CREATED          SIZE
arm64v8/debian4make       latest           39c77398e5cd      11 seconds ago   184 MB
docker.io/arm64v8/debian  buster-slim      9db65bac9886      2 weeks ago      63.4 MB
docker.io/hello-world     latest           a29f45ccde2a      11 months ago    9.14 kB
```

可以看到新的镜像已经创建成功了。

5.4.4　容器迁移的流程

从 5.4.3 节的示例可以知道，在同一个架构下，很容易实现容器的迁移，也就是把容器提交为镜像，然后把镜像推送到仓库，其他的服务器可以从仓库下载这个镜像，然后从这个

镜像创建容器，这样就实现了容器的迁移，过程如图 5-12 所示。

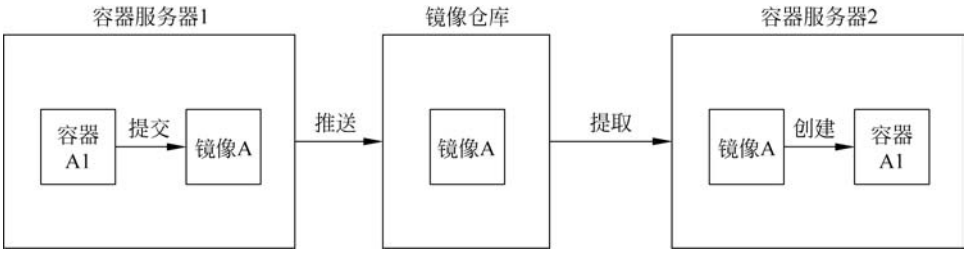

图 5-12　同架构容器迁移

但是，在不同的架构下，直接使用其他架构的镜像会出现错误，因为镜像包含的软件本身也与架构相关，在一个架构下生成的镜像，在其他的架构中是无法直接运行的。但是，镜像也可以使用 Dockerfile 生成，可以修改一个架构下的 Dockerfile 文件，使其在其他架构下也可以生成相同功能的镜像。

这里通过在 x86 架构和鲲鹏架构下分别构建一个 Redis 5.0.9 的镜像，演示不同架构之间容器的迁移。

1. x86 架构 Redis 镜像构建

步骤 1：准备 x86 架构服务器的容器环境，参考 5.4.3 节 Docker 的安装部分。

步骤 2：创建/data/redis/文件夹，然后在该文件夹内创建 Dockerfile 文件，命令如下：

```
mkdir -p /data/redis/
cd /data/redis/
vim Dockerfile
```

步骤 3：在 Dockerfile 文件内输入构建指令，保存并退出，指令如下：

```
#Chapter5/x86/Dockerfile
FROM centos/s2i-core-centos7

ENV REDIS_DOWNLOAD_URL https://GitHub.com/redis/redis/archive/5.0.9.tar.gz
ENV REDIS_TAR_NAME redis.tar.gz
RUN set -eux; \
    \
    yum update -y; \
    yum install -y \
        wget \
        gcc \
        libc6-dev \
        make \
    ; \
    yum clean all ;\
```

```
    rm -rf /var/cache/yum/* ;\
    wget -O "$REDIS_TAR_NAME" "$REDIS_DOWNLOAD_URL"; \
    mkdir -p /usr/src/redis; \
    tar -xzf "$REDIS_TAR_NAME" -C /usr/src/redis --strip-components=1; \
    rm "$REDIS_TAR_NAME"; \
    \
    make -C /usr/src/redis -j "$(nproc)" all; \
    make -C /usr/src/redis install; \
    \
    rm -r /usr/src/redis;

RUN mkdir /data
VOLUME /data
WORKDIR /data

EXPOSE 6379
CMD ["redis-server"]
```

构建文件指令简介：

FROM：构建基于系统的镜像，这里使用的是基于 CentOS 的精简镜像，体积较小。

ENV：设置环境变量，这里把下载网址和文件名称设置成环境变量，方便制作不同 Redis 版本的镜像。

RUN：要执行的指令。本构建文件中，RUN 指令执行的过程如下：

(1) 使用 yum 安装必需的依赖项。

(2) 删除了 yum 的缓存文件。

(3) 使用 wget 下载 redis 的 tar 源码包。

(4) 解压源码包到源码目录。

(5) 删除 tar 包。

(6) 使用 make 命令编译 redis 源码，编译时使用-j"$nproc"指定使用的核心数量。

(7) 使用 make install 安装。

(8) 删除源码目录。

VOLUME：创建挂载点/data。

WORKDIR：设置工作目录，这里把/data 设置为工作目录。

EXPOSE：声明服务端口，这里把 Redis 提供服务的 6379 端口声明为服务端口。

CMD：设置容器启动后默认执行的命令及其参数，这里默认启动 Redis 服务。

步骤 4：创建 Redis 镜像，命令如下：

```
docker build -t x86/centos_redis:5.0.9 .
```

构建过程所需时间较长，最后构建成功后的回显如下：

```
 + rm -r /usr/src/redis
 ---> 69f51df30aaa
Removing intermediate container b77560406af5
Step 5/9 : RUN mkdir /data
 ---> Running in 48aaaeec41e1

 ---> b477231b861b
Removing intermediate container 48aaaeec41e1
Step 6/9 : VOLUME /data
 ---> Running in 21539c2883cf
 ---> 284a49c147a3
Removing intermediate container 21539c2883cf
Step 7/9 : WORKDIR /data
 ---> 2794df97710e
Removing intermediate container 7e0a9535a2ef
Step 8/9 : EXPOSE 6379
 ---> Running in 754a0ea5f09c
 ---> 1678d17aff35
Removing intermediate container 754a0ea5f09c
Step 9/9 : CMD redis-server
 ---> Running in 65a55abf814b
 ---> b4dae1100ba2
Removing intermediate container 65a55abf814b
Successfully built b4dae1100ba2
```

步骤 5：查看新构建的镜像，命令及回显如下：

```
[root@ecs-x86 redis]#docker images
REPOSITORY                                TAG         IMAGE ID         CREATED          SIZE
x86/centos_redis                          5.0.9       b4dae1100ba2     3 minutes ago    485 MB
docker.io/centos/s2i-core-centos7         latest      d7831cbce893     2 months ago     239 MB
docker.io/hello-world                     latest      bf756fb1ae65     11 months ago    13.3 kB
```

可以看到新的镜像 x86/centos_redis:5.0.9。

步骤 6：创建/data/redis/data/文件夹，使用 x86/centos_redis:5.0.9 镜像运行容器，容器名称为 x86_redis，命令如下：

```
mkdir -p /data/redis/data/
docker run -p 6379:6379 --name x86_redis -d -v /data/redis/data/:/data/ x86/centos_
redis:5.0.9 redis-server --appendonly yes
```

创建成功后，查看容器运行状态，命令及回显如下：

```
[root@ecs-x86 redis]#docker ps
CONTAINER ID        IMAGE        COMMAND        CREATED        STATUS        PORTS        NAMES
```

```
46664ccdeaa2    x86/centos_redis:5.0.9    "container-entrypo..."    11 seconds ago
Up 11 seconds    0.0.0.0:6379->6379/tcp  x86_redis
```

容器状态为 Up，表示已经正常运行了。

步骤 7：使用 redis-cli 连接 redis 容器，然后输入 ping，正常会返回 PONG，命令及回显如下：

```
[root@ecs-x86 redis]# docker exec -it x86_redis redis-cli
127.0.0.1:6379> ping
PONG
```

步骤 8：测试 set/get 方法，指令及回显如下：

```
127.0.0.1:6379> set x86 hello
OK
127.0.0.1:6379> get x86
"hello"
```

这表明 x86 架构下使用镜像 x86/centos_redis:5.0.9 运行容器成功。

2. 鲲鹏架构 Redis 镜像构建

步骤 1：准备鲲鹏架构服务器的容器环境，可以使用 5.4.3 节安装的环境。

步骤 2：创建/data/redis/文件夹，然后在该文件夹内创建 Dockerfile 文件，命令如下：

```
mkdir -p /data/redis/
cd /data/redis/
vim Dockerfile
```

步骤 3：在 Dockerfile 文件内输入构建指令，保存并退出，指令如下：

```
#Chapter5/kunpeng/Dockerfile
FROM debian:buster-slim

ENV REDIS_DOWNLOAD_URL https://GitHub.com/redis/redis/archive/5.0.9.tar.gz
ENV REDIS_TAR_NAME redis.tar.gz
RUN set -eux; \
    \
    apt-get update -y; \
    apt-get install -y \
        wget \
        gcc \
        libc6-dev \
        make \
    ; \
    rm -rf /var/lib/apt/lists/*; \
```

```
    wget -O "$REDIS_TAR_NAME" "$REDIS_DOWNLOAD_URL"; \
    mkdir -p /usr/src/redis; \
    tar -xzf "$REDIS_TAR_NAME" -C /usr/src/redis --strip-components=1; \
    rm "$REDIS_TAR_NAME"; \
    \
    make -C /usr/src/redis -j "$(nproc)" all; \
    make -C /usr/src/redis install; \
    \
    rm -r /usr/src/redis;

RUN mkdir /data
VOLUME /data
WORKDIR /data

EXPOSE 6379
CMD ["redis-server"]
```

可以看出，鲲鹏架构的 Dockerfile 文件指令和 x86 架构的指令基本类似，主要区别在两个部分，一个是构建基于的镜像，这里选择的是 Debian 系统；另一个是 apt-install，用来取代 CentOS 中的 yum。除此之外，其他指令基本一致。

步骤 4：创建 Redis 镜像，命令如下：

```
docker build -t arm64v8/debian_redis:5.0.9 .
```

构建过程也需要较长时间，最后出现 Successfully built 表示构建成功了显示信息如下：

```
+ rm -r /usr/src/redis
---> 6fdb1ec669cc
Removing intermediate container cd41e189c725
Step 5/9 : RUN mkdir /data
---> Running in d44db4119267

---> 968e107257eb
Removing intermediate container d44db4119267
Step 6/9 : VOLUME /data
---> Running in 04c910bca4a3
---> 4c42fb187503
Removing intermediate container 04c910bca4a3
Step 7/9 : WORKDIR /data
---> 2e3e5269a671
Removing intermediate container 74b002db0f66
Step 8/9 : EXPOSE 6379
---> Running in c952a041ba8e
---> 22fbe952967d
```

```
Removing intermediate container c952a041ba8e
Step 9/9 : CMD redis-server
 ---> Running in 83e5d465fcc7
 ---> b85563db0232
Removing intermediate container 83e5d465fcc7
Successfully built b85563db0232
```

步骤 5：查看新构建的镜像，命令及回显如下：

```
[root@ecs-kunpeng redis]# docker images
REPOSITORY                  TAG             IMAGE ID        CREATED          SIZE
arm64v8/debian_redis        5.0.9           b85563db0232    46 minutes ago   211 MB
arm64v8/debian4make         latest          39c77398e5cd    2 hours ago      184 MB
docker.io/arm64v8/debian    buster-slim     9db65bac9886    2 weeks ago      63.4 MB
docker.io/debian            buster-slim     9db65bac9886    2 weeks ago      63.4 MB
docker.io/hello-world       latest          a29f45ccde2a    11 months ago    9.14 kB
```

可以看到新的镜像 arm64v8/debian_redis:5.0.9。

步骤 6：创建/data/redis/data/文件夹，使用 arm64v8/ debian_redis:5.0.9 镜像运行容器，容器名称为 kunpeng_redis，命令如下：

```
mkdir -p /data/redis/data/
docker run -p 6379:6379 --name kunpeng_redis -d -v /data/redis/data/:/data/ arm64v8/
debian_redis:5.0.9 redis-server --appendonly yes
```

创建成功后，查看容器运行状态，命令及回显如下：

```
[root@ecs-kunpeng redis]# docker ps
CONTAINER ID        IMAGE                       COMMAND                  CREATED          STATUS          PORTS                    NAMES
0ea91f3487a7        arm64v8/debian_redis:5.0.9  "redis-server --ap..."   9 seconds ago
Up 8 seconds        0.0.0.0:6379->6379/tcp      kunpeng_redis
```

可以看到 kunpeng_redis 已经成功运行。

步骤 7：使用 redis-cli 连接 redis 容器，输入 ping，正常会返回 PONG，然后测试 set/get 方法，命令及回显如下：

```
[root@ecs-kunpeng redis]# docker exec -it kunpeng_redis redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> set kunpeng hello
OK
127.0.0.1:6379> get kunpeng
"hello"
```

这表明鲲鹏架构下 redis 服务运行成功了。

3. 基于本地镜像构建 Redis 镜像

在使用 Dockerfile 构建新镜像时，除了可以基于仓库的镜像，也可以基于本地的镜像。这里创建一个生成 Redis 5.0.9 镜像的 Dockerfile 文件，使用在 5.4.3 节中创建的 arm64v8/centos4make 镜像作为基础镜像。下面列出简化的步骤。

步骤 1：进入鲲鹏服务器的/data/redis/文件夹。

步骤 2：创建 Dockerfile 文件，文件指令如下：

```
#Chapter5/kunpeng2/Dockerfile
FROM arm64v8/debian4make

ENV REDIS_DOWNLOAD_URL https://GitHub.com/redis/redis/archive/5.0.9.tar.gz
ENV REDIS_TAR_NAME redis.tar.gz
RUN set -eux; \
    \
    wget -O "$REDIS_TAR_NAME" "$REDIS_DOWNLOAD_URL"; \
    mkdir -p /usr/src/redis; \
    tar -xzf "$REDIS_TAR_NAME" -C /usr/src/redis --strip-components=1; \
    rm "$REDIS_TAR_NAME"; \
    \
    make -C /usr/src/redis -j "$(nproc)" all; \
    make -C /usr/src/redis install; \
    \
    rm -r /usr/src/redis;

RUN mkdir /data
VOLUME /data
WORKDIR /data

EXPOSE 6379
CMD ["redis-server"]
```

步骤 3：创建 Redis 镜像，命令如下：

```
docker build -t arm64v8/debian4make_redis:5.0.9 .
```

构建成功后的回显如下：

```
+ rm -r /usr/src/redis
---> fa697e87a302
Removing intermediate container 1ecd6c9f1674
Step 5/9 : RUN mkdir /data
---> Running in fbd9a2288e5c
```

```
 ---> 86b5b8cbe004
Removing intermediate container fbd9a2288e5c
Step 6/9 : VOLUME /data
 ---> Running in 04f7a7775884
 ---> 60b90212ed66
Removing intermediate container 04f7a7775884
Step 7/9 : WORKDIR /data
 ---> 58b93b6722f7
Removing intermediate container 9d4de3c9f8b6
Step 8/9 : EXPOSE 6379
 ---> Running in 291bed3e86e2
 ---> 837ab0382a13
Removing intermediate container 291bed3e86e2
Step 9/9 : CMD redis-server
 ---> Running in ed2d79416b1a
 ---> 9dd914c78911
Removing intermediate container ed2d79416b1a
Successfully built 9dd914c78911
```

后续的创建容器、测试 Redis 服务的过程和本节第 2 段“鲲鹏架构 Redis 镜像构建”的步骤 6、7、8 类似，就不再演示了，有兴趣的读者可以自己试一下。