

TaiShan 服务器代码移植参考手册

文档版本	V4.0
发布日期	2020-1-11

前言

概述





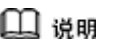
本文提供将软件从 x86 Linux 平台移植到 ARM Linux 平台的移植指导，以及移植过程中遇到的相关问题处理方法，包括编译环境准备、编译脚本和源码修改两部分内容。

读者对象

本文档主要适用于执行软件移植的研发工程师和技术支持工程师。

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 危险	用于警示紧急的危险情形，若不可避免，将会导致人员死亡或严重的人身伤害。
 警告	用于警示潜在的危险情形，若不可避免，可能会导致人员死亡或严重的人身伤害。
 注意	用于警示潜在的危险情形，若不可避免，可能会导致中度或轻微的人身伤害。
 注意	用于传递设备或环境安全警示信息，若不可避免，可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “注意”不涉及人身伤害。
 说明	用于突出重要/关键信息、最佳实践和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。

修改记录

文档版本	发布日期	修改说明
01	2019-08-03	第一次正式发布。
02	2019-08-09	修改 char 默认数据类型的编译选项错误
03	2019-08-30	1、增加 double 到整型数据类型转换时数据溢出，与 x86 平台表现对比描述 2、增加编译选项匹配鲲鹏流水线（tsv110）

目 录

前言.....	ii
1 简介.....	1
1.1 编程语言简介.....	1
1.2 基于编译型语言开发的应用程序	3
1.3 基于解释型语言开发的应用程序	3
2 准备工作.....	4
3 移植相关问题处理.....	5
3.1 编译脚本移植类问题	5
3.1.1 -m64 编译选项	5
3.1.2 char 数据类型的符号.....	5
3.2 源码修改类问题.....	6
3.2.1 代码中汇编指令需要重写	6
3.2.2 替换 x86 CRC32 汇编指令.....	7
3.2.3 替换 x86 bswap 汇编指令.....	9
3.2.4 替换 x86 rep 汇编指令.....	9
3.2.5 快速移植内联 SSE/SSE2 应用	10
3.2.6 弱内存序导致程序执行结果和预期不一致	10
3.2.7 对结构体中的变量进行原子操作时程序异常 coredump.....	11
3.2.8 核数目硬编码.....	12
3.2.9 双精度浮点型转整型时数据溢出，与 X86 平台表现不一致	12
4 编译优化项	1
4.1 gcc 编译器优化浮点运算精度.....	1
4.2 增加编译选项匹配 Kunpeng 处理器架构，提升性能.....	2
4.3 增加编译选项匹配 Kunpeng 处理器流水线，提升性能.....	2
5 代码归一	3
5.1 代码归一介绍.....	3
5.2 解释型语言代码归一	3
5.3 编译型语言代码归一，编译宏控制.....	5
5.4 运行态自动适配.....	6

5.5 32 位到 64 位代码迁移归一	6
5.6 其他归一建议	8
5.6.1 依赖组件迁移或替换后归一	8
5.6.2 核心部件替换	9

1 简介

- 1.1 编程语言简介
- 1.2 基于编译型语言开发的应用程序
- 1.3 基于解释型语言开发的应用程序

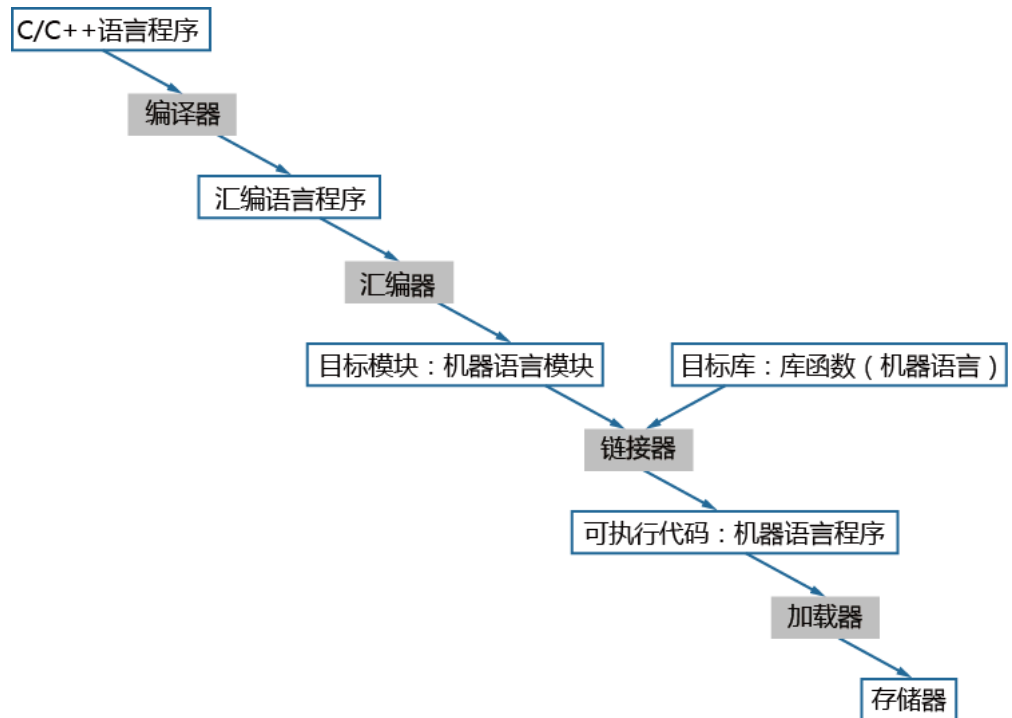
1.1 编程语言简介

按照翻译方式的不同，高级语言通常可以分为两类：一类是编译翻译，一类是解释翻译，分别对应着编译型语言和解释型语言。

- 编译型语言

典型的如 C、C++ 语言，都属于编译型语言，源代码到执行的过程概括如图 1-1 所示。C/C++ 编译好的程序是机器指令，由操作系统加载到存储器（一般为内存）后由 CPU 直接执行。

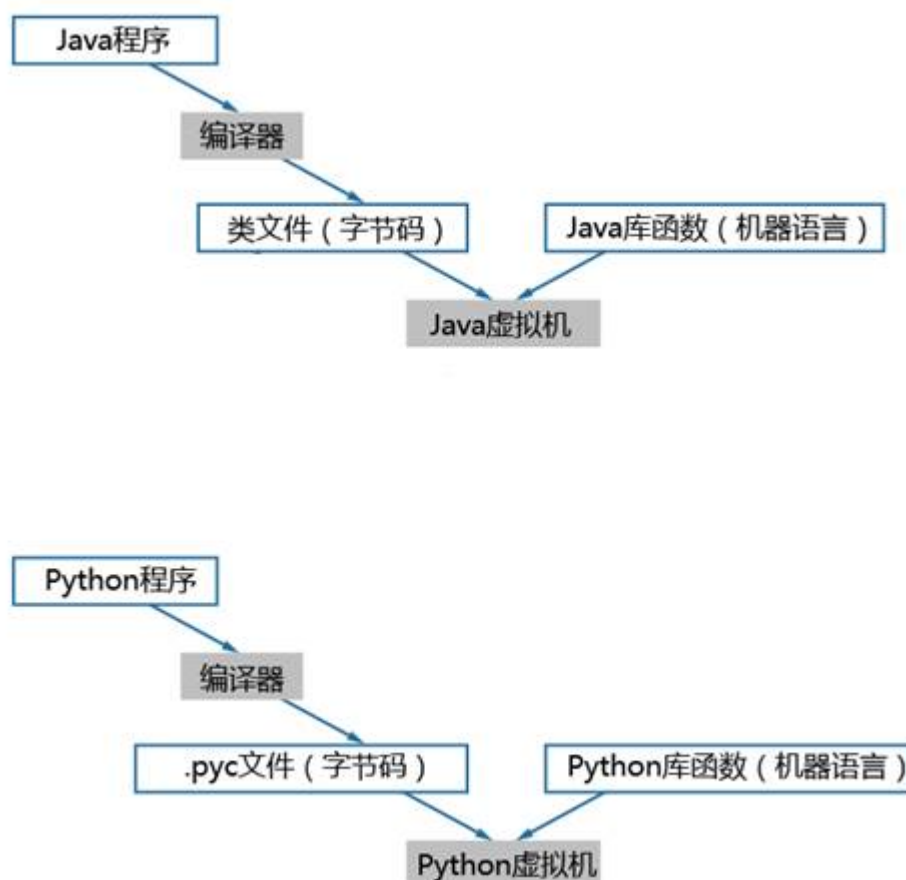
图1-1 编译型语言执行过程



- 解释型语言

典型的如 Java、Python 语言，都属于解释型语言，源代码到执行的过程概括如图 1-2 所示。Java/Python 编译好的程序是平台无关的字节码，由虚拟机解释执行，虚拟机完成平台差异的屏蔽。

图1-2 解释型语言执行过程



1.2 基于编译型语言开发的应用程序

基于编译型语言开发的应用程序，例如 C/C++ 语言应用程序，其编译后得到可执行程序，可执行程序执行时依赖的指令是 CPU 架构相关的。因此，基于 x86 架构编译的 C/C++ 语言应用程序，无法直接在 TaiShan 服务器运行，需要进行移植编译，移植编译过程中遇到的问题可以参考第 2、3 章提供的方法解决。

1.3 基于解释型语言开发的应用程序

基于解释型语言开发的应用程序，是 CPU 架构不相关的，例如 Java、Python，将这类应用程序移植到 TaiShan 服务器，无需修改和重新编译，按照与 x86 一致的方式部署和运行应用程序即可。Java 应用程序 jar 包内，可能包含基于 C/C++ 语言开发的 so 库文件，这类 so 库需要移植编译，移植编译 so 库遇到的问题可以参考第 2、3 章提供的方法解决，使用编译得到的 so 库重新打包 jar 包。

2 准备工作

C/C++程序移植需要安装编译器，推荐使用 gcc7.3 及以上版本（最低不低于 4.8.5），下载安装参考链接：

gcc7.3 版本下载地址：<http://ftp.gnu.org/gnu/gcc/gcc-7.3.0/>

安装步骤参考：<https://gcc.gnu.org/install/>

3 移植相关问题处理

3.1 编译脚本移植类问题

3.2 源码修改类问题

3.1 编译脚本移植类问题

3.1.1 -m64 编译选项

现象描述

告警信息：gcc: error: unrecognized command line option ‘-m64’

可能原因

-m64 是 x86 64 位应用编译选项，m64 选项设置 int 为 32bits 及 long、指针为 64 bits，为 AMD 的 x86 64 架构生成代码。在 ARM64 平台无法支持。

处理步骤

将 ARM64 平台对应的编译选项设置为-mabi=lp64。

3.1.2 char 数据类型的符号

现象描述

告警信息：warning: comparison is always false due to limited range of data type

可能原因

char 变量在不同 CPU 架构下默认符号不一致，在 x86 架构下为 signed char，在 ARM64 平台为 unsigned char，移植时需要指定 char 变量为 signed char。

处理步骤

在编译选项中加入“-fsigned-char”选项，指定 ARM64 平台下的 char 为有符号数。

3.2 源码修改类问题

3.2.1 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：

```
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 ARM64 平台下，使用 gcc 内置函数实现：

```
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所示：
<__sync_add_and_fetch >:
ldxr    x2, [x0]
```

```
add    x2, x2, x1
stlxr   w3, x2, [x0]
```

3.2.2 替换 x86 CRC32 汇编指令

现象描述

编译错误: unknown mnemonic `crc32q' -- `crc32q (x3),x2'或 operand 1 should be an integer register -- `crc32b (x1),x0'

或 unrecognized command line option '-msse4.2'。

可能原因

x86 使用的是 crc32b 和 crc32q 汇编指令完成 CRC32C 校验值计算功能，而 ARM64 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

请使用 crc32cb、crc32ch、crc32cw、crc32cx 取代 x86 的 CRC32 系列汇编指令，替换方法如表 3-1 所示，并在编译时添加编译参数-mcpu=generic+crc。

表3-1 替换方法

指令	输入数据位宽 (bit)	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。
crc32cw	32	适用输入数据位宽为 32bit。
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 下的实现：

```
#define CRC32CB(crc, value) __asm__("crc32b\t" "(%1), %0" : "=r"(crc) :  
"r"(value), "0"(crc))  
#define CRC32CP(crc, value) __asm__("crc32q\t" "(%1), %0" : "=r"(crc) :  
"r"(value), "0"(crc))  
  
uint32_t crc32c(uint32_t crc, void const *buf, unsigned int len) {  
    uint64_t crc0 = crc;  
    unsigned char const *next = buf;
```

```

{
    unsigned char const * const end = next + (len - (len & 7));
    while (next < end) {
        CRC32CP(crc0, next);
        next += 8;
    }
    len &= 7;
}
while (len) {
    CRC32CB(crc0, next);
    next++;
    len--;
}

return crc0;
}

```

- 在 ARM64 平台下的实现:

```

#define CRC32CX(crc, value)
__asm__ ("crc32cx %w[c], %w[c], %x[v]": [c]"r"(crc): [v]"r"(value))
#define CRC32CW(crc, value)
__asm__ ("crc32cw %w[c], %w[c], %w[v]": [c]"r"(crc): [v]"r"(value))
#define CRC32CH(crc, value)
__asm__ ("crc32ch %w[c], %w[c], %w[v]": [c]"r"(crc): [v]"r"(value))
#define CRC32CB(crc, value)
__asm__ ("crc32cb %w[c], %w[c], %w[v]": [c]"r"(crc): [v]"r"(value))

uint32_t crc32c_arm64_le_hw(uint32_t crc, const uint8_t *p, unsigned int
len)
{
    int64_t length = len;

    while ((length -= sizeof(uint64_t)) >= 0) {
        CRC32CX(crc, *((uint64_t *)p));
        p += sizeof(uint64_t);
    }

    if (length & sizeof(uint32_t)) {
        CRC32CW(crc, *((uint32_t *)p));
        p += sizeof(uint32_t);
    }

    if (length & sizeof(uint16_t)) {
        CRC32CH(crc, *((uint16_t *)p));

```

```
p += sizeof(uint16_t);  
}  
  
if (length & sizeof(uint8_t))  
CRC32CB(crc, *p);  
  
return crc;  
}
```

3.2.3 替换 x86 bswap 汇编指令

现象描述

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

可能原因

bswap 是 x86 的字节序反序指令，需替换为 ARM64 的 rev 指令。

处理步骤

x86 指令实现的 bswap 如下：

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

替换为 ARM64 指令后如下：

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

3.2.4 替换 x86 rep 汇编指令

现象描述

编译报错：unknown mnemonic `rep' -- `rep'。

可能原因

rep 为 x86 的重复执行指令，需替换为 ARM64 的 rept 指令。

处理步骤

替换方法如下：

替换前：

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

替换后：

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

```
#define nops(n) asm volatile(__nops(n))
```

3.2.5 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 ARM64 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 ARM64 平台的函数，代码下载地址：

<https://github.com/open-estuary/sse2neon.git>。

使用方法如下：

步骤 1 将下载项目中的 SSE2NEON.h 文件拷贝到待移植项目中。

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。

----结束

3.2.6 弱内存序导致程序执行结果和预期不一致

现象描述

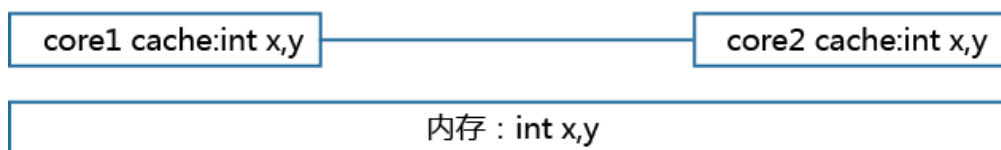
弱内存序导致程序执行结果和预期不一致。

可能原因

ARM64 平台是弱内存序，原理如下：

1. 同一份数据，在 cache 里面存在多份，需要 CPU 之间进行同步。

MESI协议进行同步



2. 代码编写顺序和执行顺序可能不一样。

开发人员写如下代码

```
int x = 0;
int y = 0;
x = 1;
y = 1;
```

CPU1 上可能的执行顺序与预期不一致：

```
y = 1; // y先执行
x = 1;
```

CPU2 上的线程在执行如下逻辑，就可能现问题

```
if ( y==1 ) {
    // x 可能为0
    assert(x==1);
}
```

CPU 内部是流水线执行，在执行到 $x=1$ 时，如果 x 在内存，那么 CPU 就会等待 x 导入到 cache，在等待的过程中如果 y 已经在 cache 中了，那么 CPU 会执行 $y=1$ ，这样就导致后面的语句先执行。

对系统的影响

- 影响无锁编程的代码。
- 对于使用信号量机制写的互斥代码，因为信号量函数已经带了内存屏障的指令，所以无影响。

处理步骤

找到使用无锁编程的代码，检查是否用内存屏障指令保证了数据的一致性。

使用内存屏障指令保证对共享数据的访问和预期一致。

示例：

```
int x = 0;
int y = 0;
x = 1;
smp_wmb(); // 等待 x=1 执行完成
y = 1;
```

```
其它线程在执行如下逻辑，就可能现问题
if ( y==1 ) {
    smp_rmb(); // 保证读的数据是最新的
    assert(x==1);
}
```

3.2.7 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.
0x00000000040083c 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:
0x000000000400824 <+0>:      sub     sp, sp, #0x10
0x000000000400828 <+4>:      mov     x0, #0x1                                // #1
0x00000000040082c <+8>:      str     x0, [sp, #8]
0x000000000400830 <+12>:     adrp    x0, 0x420000 <__libc_start_main@got.plt>
0x000000000400834 <+16>:     add     x0, x0, #0x31 // 将变量的地址放入到 x0 寄存器
0x000000000400838 <+20>:     ldr     x1, [sp, #8] // 指定 ldxr 取数据的长度（此处为 8 字节）
=> 0x00000000040083c <+24>:     ldxr    x2, [x0] // ldxr 从 x0 寄存器指向的内存地址中取值
0x000000000400840 <+28>:     add     x2, x2, x1
0x000000000400844 <+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

```

可能原因

ARM64 平台对变量的原子操作、锁操作等用到了 `ldaxr`、`stlxx` 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。

3.2.8 核数目硬编码

TaiShan 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 `core` 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

您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 TaiShan 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)`）来获取实际核数再进行绑核。

3.2.9 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，TaiShan 平台的表现与 x86 平台的表现不同。

```

long aa = (long)0x7FFFFFFFFFFFFFFF;
long bb;
bb = (long)(aa*(double)10); // long->double->long
//x86: aa=9223372036854775807, bb=-9223372036854775808
//arm64:aa=9223372036854775807, bb=9223372036854775807

```

可能原因

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：

CPU	double 值	转为 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 数据向 unsigned long 转换：

CPU	double 值	转为 int 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	0xFFFFFFFFFFFFFFF	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 数据向 int 转换：

CPU	double 值	转为 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 数据向 unsigned int 转换：

CPU	double 值	转为 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

4 编译优化项

- 4.1 gcc 编译器优化浮点运算精度
- 4.2 增加编译选项匹配 Kunpeng 处理器架构，提升性能
- 4.3 增加编译选项匹配 Kunpeng 处理器流水线，提升性能

4.1 gcc 编译器优化浮点运算精度

现象描述

编译优化选项设置-O2 级别及以上时，相同的浮点数乘加运算在 x86 平台和 ARM64 平台的运算结果，在小数点后 16 位存在差异。

可能原因

ARM64 平台编译优化选项设置为-O2 级别及以上，进行浮点数的乘加运算($a+=b*c$)，运算结果的精度只能精确到小数点后 16 位。在配置-O2 选项时，gcc 使用融合指令 *fmadd* 完成乘加运算，而不是 *fadd* 和 *fmul*。

fmadd 将浮点数的乘法和加法看成不可分的一个操作，不对中间结果进行舍入，从而导致计算结果有所差别。

对系统的影响

编译优化选项设置-O2 级别及以上时，浮点乘加运算的性能有提升，但是运算的精度受到影响。

处理步骤

添加编译选项-*ffp-contract=off* 可以关闭该优化。

4.2 增加编译选项匹配 Kunpeng 处理器架构，提升性能

在编译时增加编译选项指定处理器架构为 armv8，使编译器按照 Kunpeng 处理器的架构和微架构生成可执行程序，提升性能。

处理步骤

编译选项中添加 `-march=armv8-a`。

4.3 增加编译选项匹配 Kunpeng 处理器流水线，提升性能

如果使用了 gcc 9.1 以上的版本，在编译时增加编译选项指定使用 tsv110 流水线，使编译器按照 Kunpeng 处理器的流水线编排指令执行顺序，充分利用流水线的指令集并行，提升性能。

处理步骤

编译选项中添加 `-mtune=tsv110`。

5 代码归一

- 5.1 代码归一介绍
- 5.2 解释型语言代码归一
- 5.3 编译型语言代码归一，编译宏控制
- 5.4 运行态自动适配
- 5.5 32 位到 64 位代码迁移归一
- 5.6 其他归一建议

5.1 代码归一介绍

鲲鹏服务器底层芯片架构和传统 X86 芯片有差异，业务软件迁移过程中要对代码进行适配修改。修改后的代码可能会维护 2 个代码分支，这样增加了后期代码维护成本，这样是不可取的。将 2 个分支的代码合一到一套代码的过程称为代码归一。本章提供一些通用常见的代码归一方法和建议。

从技术上讲，鲲鹏迁移后所有场景的 2 套代码都能实现代码归一，业务决策者会在归一工作量和后期维护成本之间权衡是否选择归一或者什么时候归一。建议业务软件迁移后尽快归一，越早越好。

5.2 解释型语言代码归一

解释型语言编译后生成平台无关的字节码，由虚拟机解释执行，虚拟机完成平台差异的屏蔽，所以基于解释型语言开发的应用程序，与底层芯片架构无关的，无须迁移可以将同一套代码放在鲲鹏和 X86 服务器上运行。这里以 java 代码为例。

处理步骤

无须特别归一，同一套代码可以运行在鲲鹏和 X86 平台上。如 java 代码编译后的 jar 包可以直接在鲲鹏上运行。

注意：

- (1) Java 应用程序 jar 包内，可能包含基于 C/C++ 语言开发的 so 库文件，这类 so 库需要移植编译，移植编译 so 库遇到的问题可以参考第 2、3 章提供的方法解决，使用编译得到的 so 库重新打包 jar 包。如果存在这类 so 库的代码，要实行归一。

- (2) 归一代码在不同芯片架构服务器上运行的 JDK 版本建议保持一致，以免出现不兼容或者性能差异问题。

5.3 编译型语言代码归一，编译宏控制

基于编译型语言开发的应用程序，其编译后得到可执行程序，可执行程序执行时依赖的指令是芯片架构相关的。因此，软件迁移后鲲鹏架构上的代码和 x86 架构的代码会不一致，代码归一需要利用合适的措施修改代码适配。这里以 C/C++ 代码为例。

处理步骤

通过编译宏控制是 C/C++ 中较为通用的代码片段隔离方法，同样适用于不同芯片架构中的代码归一。一般方法是，在代码中用不同的宏将对应芯片的代码包含起来，在编译时选择或者自动识别选用的宏类型，如果不是被启用的宏，其代码就不会被编译。

示例：

下面示例代码是 CRC 指令在 X86 和 ARM 平台的不同实现方法，可以通过编译宏控制来进行区分隔离，代码可以放到同一个文件中，实现代码归一。

```
// x86环境下会编译如下代码块
#ifdef X86_64
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;
}
#endif

// ARM 环境下会编译如下代码块
#ifdef ARM_AARCH64
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;
}
#endif
```

C/C++语言中通过条件编译`#ifdef`, `#endif` 等关键词实现将代码块编译区分, 在启动编译前可以在 `makefile` 或者业务配置文件里面打开对应芯片的宏定义 (如示例代码中的宏 `ARM_AARCH64` 和宏 `X86_64`), 这样就能实现代码合一。

5.4 运行态自动适配

上层业务代码如果要根据不同的芯片类型设置不同的参数, 或者走不同的代码逻辑路径, 可以通过实时查询芯片类型来判断, 这样代码实现了自动适配芯片类型。与编译宏控制方法的区别是, 这类代码和编译无关, 不同芯片架构都可以编译。

处理步骤

简单通用的方法是实时查询 CPU 的类型, 根据查询结果来判断。如, linux 系统可以通过 `lscpu` 命令来实现:

执行命令: `lscpu | grep Archit | awk '{print $2}'`

1. 鲲鹏服务器显示结果:

```
[root@gaoqiao16u ~]# lscpu | grep Archit | awk '{print $2}'
aarch64
```

2. X86 服务器显示结果:

```
[root@gaoqiao16u ~]# lscpu | grep Archit | awk '{print $2}'
X86_64
```

编程语言有类似系统调用 `system()` 函数, 通过这类函数执行上述 shell 命令, 根据执行结果判断代码分支执行路径。

```
// 自动识别芯片类型, 伪代码
strCmdLine = "lscpu | grep Archit | awk '{print $2}' ";
strExeResult = system(strCmdLine);

if ("aarch64" == strExeResult)    //如果是ARM芯片业务代码放在这里
{
    .....
}
if ("X86_64" == strExeResult)    //如果是X86芯片业务代码放在这里
{
    .....
}
```

除了 `lscpu` 命令以外还有很多其他方法可以判断底层芯片类型, 根据业务情况自行选择。

5.5 32 位到 64 位代码迁移归一

现象描述

X86 服务器支持 32 位和 64 位程序, 鲲鹏服务器目前只支持 64 位程序, 业务从 32 位系统迁移到 64 位系统涉及到很多基础数据结构及数据类型的修改, 很多业务维护 2 套

代码，给后面维护带来不变。本章节基于 C/C++ 语言为例说明 32 位程序升级到 64 位程序需要关注的地方，代码归一方式可以采用第 5 章中描述的方法完成。

处理步骤

64 位 CPU 与 32 位 CPU 的不同，主要体现在基本数据类型的长度差别。下面是基本数据类型在 32 和 64 位上的差别：

类型	ILP32	LP64
char	8	8
short	16	16
int	32	32
long long	64	64
long	32	64
size_t	32	64
pointer	32	64

由于基础类型的变化，可能会导致部分数据结构在字节对齐和空间上有差别，需要特别注意。

示例：

```
typedef struct A{
    void* p;
    int i;
}B;
在32位系统上，指针长度为4，B 4字节对齐，sizeof(B) 等于8；
在64位系统上，指针长度为8，B 8字节对齐，sizeof(B) 等于16。
```

修改建议：

格式化字符串

(1.1) 指针的格式化使用%p，不允许使用%x，%lx。%p 对于用指针类型，同时适用于 32 位和 64 位系统。%x 对应于 32 位整数，%lx 对应于 64 位整数。不能用于指针的格式化。

(1.2) size_t 类型的格式化使用%zd 或%zx,不允许使用%d,%ld,%x,%lx。%zd 对于用 size_t 类型，同时适用于 32 位和 64 位系统。%d,%x 对应于 32 位整数，%ld,%lx 对应于 64 位整数。不能用于 size_t 的格式化。

(1.3) 打印变长数据，对齐时要考虑数据长度。不能假设指针为 8

指针转换

(2.1) 指针的长度必须使用 sizeof 计算，不允许假设指针的长度为 4 或者 8。

(2.2) 不允许指针与 UINT32 之间相互赋值。包括函数参数传递。64 位中 UINT32 是 32 位的，指针是 64 位的。如果要定义的变量可能是不确定的指针，使用 VOID*；如果要定义的变量既可以是指针，也可以是整形，使用 UINTPTR。

(2.3)不允许 `size_t` 与 `UINT32` 之间相互赋值。包括函数参数传递。64 位中 `UINT32` 是 32 位的，`size_t` 是 64 位的。

(2.4)对于指针与整形通用的场景，使用 `UINTPTR`。某些数据结构，例如双链表的用户自定义字段，既允许存入指针，也允许存入整形。应该使用 `UINTPTR` 定义变量。`UINTPTR` 是变长的基本数据类型，在 32 位系统上等价于 `UINT32`，而在 64 位系统上等价于 `UINT64`。

(2.5)指针偏移运算，将指针转换为 `char *` 之后做加减运算。不允许转换成 `UINT32` 后计算。为了将指针偏移一定的大小，通常会将指针转换成整形，在做加减运算。32 位系统上一般使用 `UINT32` 强转，这种做法在 64 位系统上不再正确。应该使用 `char*` 或者 `UINTPTR` 强转。

数据结构对齐

计算数据结构的成员的偏移，需要考虑字节对齐和填充，不能直接计算每个成员的大小之后作为偏移。

各对端通信模块的接口中数据接口字节对齐要一致。

指令修改

部分业务代码中间嵌入了汇编代码，直接对寄存器进行操作，32 位和 64 位部分汇编指令寄存处长度不一致，要做适配修改。可以按照本文第 3 章指导进行移植修改。

采用编译宏的方式进行代码归一。

5.6 其他归一建议

5.6.1 依赖组件迁移或替换后归一

现象描述

X86 代码迁移到鲲鹏上以后，依赖组件不一致，导致代码不能归一

处理步骤

目标是将鲲鹏和 X86 上代码业务组件整合为一致，实现代码归一。方法有：

- (1)如果 X86 上组件时开源的代码可获得，重新在鲲鹏上编译打包验证，迁移到鲲鹏上。
- (2)如果 X86 上组件闭源不能迁移，可以考虑用其他功能类似的组件替代。这个组件可以同时运行在鲲鹏和 X86 上。

5.6.2 核心部件替换

现象描述

X86 代码迁移到鲲鹏上以后，由于核心部件，如数据库、中间件等变化，导致适配这些部件的接口代码不一样，导致代码归一困难。

处理步骤

项目管理者需要决策不同接口代码归一和替换核心组件的代价和收益，从而评估采用哪种方法。参考第 5 章方法，完成接口代码归一；寻找核心部件的替代者(能够同时运行在鲲鹏和 X86)，部署到两种服务器上，实现代码归一。

注意，部分核心组件不能支持 ARM 或者商业闭源软件不支持 ARM，替换成为归一的主要途径。例如，某业务系统在 X86 上使用 altibase 内存数据库，altibase 是商业闭源软件且不支持 ARM 系统，那么可以采用 Redis 等第三方开源数据库替换，同时支持 X86 和鲲鹏服务器，实现代码归一。