IBM

developerWorks 中国 技术主题 Linux 文档库

Linux 内核中的 GCC 特性

了解用于 C 语言的 GCC 扩展

Linux® 内核使用 GNU Compiler Collection (GCC) 套件的几个特殊功能。这些功能包括提供快捷方式和简化以及向编译器提供优化提示等等。了解这些特殊的 GCC 特性，学习如何在 Linux 内核中使用它们。

M. Tim Jones 是一名嵌入式软件工程师，他是 Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming（现在已经是第 2 版）、AI Application Programming（第 2 版）和 BSD Sockets Programming from a Multilanguage Perspective 等书的作者。他的工程背景非常广泛，从同步宇宙飞船的内核开发到嵌入式系统架构设计，再到网络协议的开发。Tim 是位于科罗拉多州 Longmont 的 Emulex Corp. 的一名顾问工程师。

2009 年 4 月 07 日

GCC 和 Linux 是出色的组合。尽管它们是独立的软件，但是 Linux 完全依靠 GCC 在新的体系结构上运行。Linux 还利用 GCC 中的特性（称为*扩展*）实现更多功能和优化。本文讨论一些重要的扩展，讲解如何在 Linux 内核中使用它们。

GCC 当前的稳定版本（版本 4.3.2）支持 C 标准的三个版本：

- International Organization for Standardization (ISO) 最初的 C 语言标准（ISO C89 或 C90）

带修正 1 的 ISO C90

当前的 ISO C99（这是 GCC 使用的默认标准，本文也假设采用这种标准）

注意：本文假设使用 ISO C99 标准。如果指定比 ISO C99 版本旧的标准，那么可能无法使用本文描述的一些扩展。可以在命令行上使用 `-std` 选项指定 GCC 使用的实际标准。可以通过 GCC 手册查看哪个标准版本支持哪些扩展（见 参考资料 中的链接）。

可以以几种方式对可用的 C 扩展进行分类。本文把它们分为两大类：

功能性 扩展提供新功能。

优化 扩展帮助生成更高效的代码。

可应用的版本
本文主要关注在 2.6.27.1 Linux 内核和 GCC 的 4.3.2 版本中使用 GCC 扩展。每个 C 扩展引用 Linux 内核源代码中的一个文件，可以在其中找到示例。

功能性扩展

先讨论一些扩展标准 C 语言的 GCC 扩展。

类型发现

GCC 允许通过变量的引用识别类型。这种操作支持*泛型编程*。在 C++、Ada 和 Java™ 语言等许多现代编程语言中都可以找到相似的功能。Linux 使用 `typeof` 构建 `min` 和 `max` 等依赖于类型的操作。清单 1 演示如何使用 `typeof` 构建一个泛型宏（见 ./linux/include/linux/kernel.h）。

清单 1. 使用 typeof 构建一个泛型宏

```
#define min(x, y) ({                            \
        typeof(x) _min1 = (x);                  \
        typeof(y) _min2 = (y);                  \
        (void) (&_min1 == &_min2);              \
        _min1 < _min2 ? _min1 : _min2; })
```

范围扩展

GCC 支持范围，在 C 语言的许多方面都可以使用范围。其中之一是 switch/case 块中的 case 语句。在复杂的条件结构中，通常依靠嵌套的 if 语句实现与清单 2（见 ./linux/drivers/scsi/sd.c）相同的结果，但是清单 2 更简洁。使用 switch/case 也可以通过使用跳转表实现进行编译器优化。

清单 2. 在 case 语句中使用范围

```
static int sd_major(int major_idx)
{
        switch (major_idx) {
        case 0:
                return SCSI_DISK0_MAJOR;
        case 1 … 7:
                return SCSI_DISK1_MAJOR + major_idx - 1;
        case 8 … 15:
                return SCSI_DISK8_MAJOR + major_idx - 8;
        default:
                BUG();
                return 0;       /* shut up gcc */
        }
}
```

还可以使用范围进行初始化，如下所示（见 ./linux/arch/cris/arch-v32/kernel/smp.c）。在这个示例中，spinlock_t 创建一个大小为 LOCK_COUNT 的数组。数组的每个元素初始化为 SPIN_LOCK_UNLOCKED 值。

```
/* Vector of locks used for various atomic operations */
spinlock_t cris_atomic_locks[] = { [0 ... LOCK_COUNT - 1] = SPIN_LOCK_UNLOCKED};
```

范围还支持更复杂的初始化。例如，以下代码指定数组中几个子范围的初始值。

```
int widths[] = { [0 … 9] = 1, [10 … 99] = 2, [100] = 3 };
```

零长度的数组

在 C 标准中，必须定义至少一个数组元素。这个需求往往会使代码设计复杂化。但是，GCC 支持零长度数

组的概念，这对于结构定义尤其有用。这个概念与 ISO C99 中灵活的数组成员相似，但是使用不同的语法。

下面的示例在结构的末尾声明一个没有成员的数组（见 ./linux/drivers/ieee1394/raw1394-private.h）。这允许结构中的元素引用结构实例后面紧接着的内存。在需要数量可变的数组成员时，这个特性很有用。

```
struct iso_block_store {
        atomic_t refcount;
        size_t data_size;
        quadlet_t data[0];
};
```

判断调用地址

在许多情况下，需要判断给定函数的调用者。GCC 提供用于此用途的内置函数 __builtin_return_address。这个函数通常用于调试，但是它在内核中还有许多其他用途。

如下面的代码所示，__builtin_return_address 接收一个称为 level 的参数。这个参数定义希望获取返回地址的调用堆栈级别。例如，如果指定 level 为 0，那么就是请求当前函数的返回地址。如果指定 level 为 1，那么就是请求进行调用的函数的返回地址，依此类推。

```
void * __builtin_return_address( unsigned int level );
```

在下面的示例中（见 ./linux/kernel/softirq.c），local_bh_disable 函数在本地处理器上禁用软中断，从而禁止在当前处理器上运行 softirqs、tasklets 和 bottom halves。使用 __builtin_return_address 捕捉返回地址，以便在以后进行跟踪时使用这个地址。

```
void local_bh_disable(void)
{
        __local_bh_disable((unsigned long)__builtin_return_address(0));
}
```

常量检测

在编译时，可以使用 GCC 提供的一个内置函数判断一个值是否是常量。这种信息非常有价值，因为可以构造出能够通过常量叠算（constant folding）优化的表达式。__builtin_constant_p 函数用来检测常量。

__builtin_constant_p 的原型如下所示。注意，__builtin_constant_p 并不能检测出所有常量，因为 GCC 不容易证明某些值是否是常量。

```
int __builtin_constant_p( exp )
```

Linux 相当频繁地使用常量检测。在清单 3 所示的示例中（见 ./linux/include/linux/log2.h），使用常量检测优化 roundup_pow_of_two 宏。如果发现表达式是常量，那么就使用可以优化的常量表达式。如果表达式不是常量，就调用另一个宏函数把值向上取整到 2 的幂。

清单 3. 使用常量检测优化宏函数

```
#define roundup_pow_of_two(n)                   \
(                                               \
        __builtin_constant_p(n) ? (         \
                (n == 1) ? 1 :                  \
                (1UL << (ilog2((n) - 1) + 1))   \
                                ) :             \
        __roundup_pow_of_two(n)                 \
)
```

函数属性

GCC 提供许多函数级属性，可以通过它们向编译器提供更多数据，帮助编译器执行优化。本节描述与功能相关联的一些属性。下一节描述 影响优化的属性。

如清单 4 所示，属性通过其他符号定义指定了别名。可以以此帮助阅读源代码参考，了解属性的使用方法（见 ./linux/include/linux/compiler-gcc3.h）。

清单 4. 函数属性定义

```
# define __inline__     __inline__      __attribute__((always_inline))
# define __deprecated           __attribute__((deprecated))
```

```
# define __attribute_used__     __attribute__((__used__))
# define __attribute_const__     __attribute__((__const__))
# define __must_check            __attribute__((warn_unused_result))
```

清单 4 所示的定义是 GCC 中可用的一些函数属性。它们也是在 Linux 内核中最有用的函数属性。下面解释如何使用这些属性：

always_inline 让 GCC 以内联方式处理指定的函数，无论是否启用了优化。

deprecated 指出函数已经被废弃，不应该再使用。如果试图使用已经废弃的函数，就会收到警告。还可以对类型和变量应用这个属性，促使开发人员尽可能少使用它们。

__used__ 告诉编译器无论 GCC 是否发现这个函数的调用实例，都要使用这个函数。这对于从汇编代码中调用 C 函数有帮助。

__const__ 告诉编译器某个函数是无状态的（也就是说，它使用传递给它的参数生成要返回的结果）。

warn_unused_result 让编译器检查所有调用者是否都检查函数的结果。这确保调用者适当地检验函数结果，从而能够适当地处理错误。

下面是在 Linux 内核中使用这些属性的示例。deprecated 示例来自与体系结构无关的内核（./linux/kernel/resource.c），const 示例来自 IA64 内核源代码（./linux/arch/ia64/kernel/unwind.c）。

```
int __deprecated __check_region(struct resource
    *parent, unsigned long start, unsigned long n)

static enum unw_register_index __attribute_const__
    decode_abreg(unsigned char abreg, int memory)
```

优化扩展

现在，讨论有助于生成更好的机器码的一些 GCC 特性。

分支预测提示

在 Linux 内核中最常用的优化技术之一是 __builtin_expect。在开发人员使用有条件代码时，常常知道最可能执行哪个分支，而哪个分支很少执行。如果编译器知道这种预测信息，就可以围绕最可能执行的分支生成最优的代码。

如下所示，__builtin_expect 的使用方法基于两个宏 likely 和 unlikely（见 ./linux/include/linux/compiler.h）。

```
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
```

通过使用 __builtin_expect，编译器可以做出符合提供的预测信息的指令选择决策。这使执行的代码尽可能接近实际情况。它还可以改进缓存和指令流水线。

例如，如果一个条件标上了 “likely”，那么编译器可以把代码的 True 部分直接放在分支指令后面（这样就不需要执行分支指令）。通过分支指令访问条件结构的 False 部分，这不是最优的方式，但是访问它的可能性不大。按照这种方式，代码对于最可能出现的情况是最优的。

清单 5 给出一个使用 likely 和 unlikely 宏的函数（见 ./linux/net/core/datagram.c）。这个函数预测 sum 变量将是零（数据包的 checksum 是有效的），而且 ip_summed 变量不等于 CHECKSUM_HW。

清单 5. likely 和 unlikely 宏的使用示例

```
unsigned int __skb_checksum_complete(struct sk_buff *skb)
{
        unsigned int sum;

        sum = (u16)csum_fold(skb_checksum(skb, 0, skb->len, skb->csum));
        if (likely(!sum)) {
                if (unlikely(skb->ip_summed == CHECKSUM_HW))
                        netdev_rx_csum_fault(skb->dev);
                skb->ip_summed = CHECKSUM_UNNECESSARY;
        }
```

```
        return sum;
}
```

预抓取

另一种重要的性能改进方法是把必需的数据缓存在接近处理器的地方。缓存可以显著减少访问数据花费的时间。大多数现代处理器都有三类内存:

一级缓存通常支持单周期访问

二级缓存支持两周期访问

系统内存支持更长的访问时间

为了尽可能减少访问延时并由此提高性能，最好把数据放在最近的内存中。手工执行这个任务称为*预抓取*。GCC 通过内置函数 __builtin_prefetch 支持数据的手工预抓取。在需要数据之前，使用这个函数把数据放到缓存中。如下所示，__builtin_prefetch 函数接收三个参数:

数据的地址

rw 参数，使用它指明预抓取数据是为了执行读操作，还是执行写操作

locality 参数，使用它指定在使用数据之后数据应该留在缓存中，还是应该清除

```
void __builtin_prefetch( const void *addr, int rw, int locality );
```

Linux 内核经常使用预抓取。通常是通过宏和包装器函数使用预抓取。清单 6 是一个辅助函数示例，它使用内置函数的包装器（见 ./linux/include/linux/prefetch.h）。这个函数为流操作实现预抓取机制。使用这个函数通常可以减少缓存缺失和停顿，从而提高性能。

清单 6. 范围预抓取的包装器函数

```
#ifndef ARCH_HAS_PREFETCH
#define prefetch(x) __builtin_prefetch(x)
#endif

static inline void prefetch_range(void *addr, size_t len)
{
#ifdef ARCH_HAS_PREFETCH
        char *cp;
        char *end = addr + len;

        for (cp = addr; cp < end; cp += PREFETCH_STRIDE)
                prefetch(cp);
#endif
}
```

变量属性

除了本文前面讨论的函数属性之外，GCC 还为变量和类型定义提供了属性。最重要的属性之一是 aligned 属性，它用于在内存中实现对象对齐。除了对于性能很重要之外，某些设备或硬件配置也需要对象对齐。aligned 属性有一个参数，它指定所需的对齐类型。

下面的示例用于软件暂停（见 ./linux/arch/i386/mm/init.c）。在需要页面对齐时，定义 PAGE_SIZE 对象。

```
char __nosavedata swsusp_pg_dir[PAGE_SIZE]
        __attribute__ ((aligned (PAGE_SIZE)));
```

清单 7 中的示例说明关于优化的两点：

packed 属性打包一个结构的元素，从而尽可能减少它们占用的空间。这意味着，如果定义一个 char 变量，它占用的空间不会超过一字节（8 位）。位字段压缩为一位，而不会占用更多存储空间。

这段源代码使用一个 __attribute__ 声明进行优化，它用逗号分隔的列表定义多个属性。

清单 7. 结构打包和设置多个属性

```
static struct swsusp_header {
        char reserved[PAGE_SIZE - 20 - sizeof(swp_entry_t)];
        swp_entry_t image;
```

```
        char    orig_sig[10];
        char    sig[10];
} __attribute__((packed, aligned(PAGE_SIZE))) swsusp_header;
```

结束语

本文只讨论了在 Linux 内核中可以使用的几个 GCC 特性。可以通过 GNU GCC 手册进一步了解针对 C 和 C++ 语言的所有扩展（见 参考资料 中的链接）。另外，尽管 Linux 内核经常使用这些扩展，但是也可以在用户自己的应用程序中使用它们。随着 GCC 的发展，肯定会出现新的扩展，它们会进一步改进性能和增加 Linux 内核的功能。

参考资料

学习

GNU Compiler Collection 提供关于 GCC 的所有信息。在这里，可以找到新闻和最新的 GCC 源代码（包括以前的和最新的发行版）。还可以找到每个版本的详细历史和 在线文档，这可以提供 GCC 和所有扩展的详细信息。

Experimental GCC extensions 是 GNU 维护的扩展列表，标准的发行版还不包含这些扩展。可以通过它了解正在开发的 GCC 扩展。

阅读文章 “认识 GCC 4”（developerWorks，2008 年 10 月），了解关于 GCC 的第四个主要版本（GCC4）的更多信息。这篇文章介绍 GCC4 以及它相对于前四个次要版本的改进。

在 developerWorks Linux 专区 寻找为 Linux 开发人员（包括 Linux 新手入门）准备的更多参考资料，查阅我们 最受欢迎的文章和教程。

在 developerWorks 上查阅所有 Linux 技巧 和 Linux 教程。

随时关注 developerWorks 技术活动和网络广播。

获得产品和技术

使用可直接从 developerWorks 下载的 IBM 试用软件 构建您的下一个 Linux 开发项目。

讨论

通过博客、论坛、podcast 和空间，加入 developerWorks 社区。