**Kasan - Linux 内核的内存检测工具**



滕 瑞  
2016 年 8 月 30 日发布

[Weibo](http://service.weibo.com/share/share.php?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Flinux%2F1608_tengr_kasan%2Findex.html%25&title=Kasan%20-%20Linux%20%E5%86%85%E6%A0%B8%E7%9A%84%E5%86%85%E5%AD%98%E6%A3%80%E6%B5%8B%E5%B7%A5%E5%85%B7&language=zh_cn)[Google+](https://plus.google.com/share?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Flinux%2F1608_tengr_kasan%2Findex.html&t=Kasan%20-%20Linux%20%E5%86%85%E6%A0%B8%E7%9A%84%E5%86%85%E5%AD%98%E6%A3%80%E6%B5%8B%E5%B7%A5%E5%85%B7)[用电子邮件发送本页面](mailto:?subject=Kasan%20-%20Linux%20%E5%86%85%E6%A0%B8%E7%9A%84%E5%86%85%E5%AD%98%E6%A3%80%E6%B5%8B%E5%B7%A5%E5%85%B7&body=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Flinux%2F1608_tengr_kasan%2Findex.html)

[Comments](https://www.ibm.com/developerworks/cn/linux/1608_tengr_kasan/index.html#icomments)

[0](https://www.ibm.com/developerworks/cn/linux/1608_tengr_kasan/index.html" \l "icomments)

**引言**

Kasan 是 Kernel Address Sanitizer 的缩写，它是一个动态检测内存错误的工具，主要功能是检查内存越界访问和使用已释放的内存等问题。Kasan 集成在 Linux 内核中，随 Linux 内核代码一起发布，并由内核社区维护和发展。

**背景**

Kasan 可以追溯到 LLVM 的 sanitizers 项目（https://github.com/google/sanitizers），这个项目包含了 AddressSanitizer，MemorySanitizer，ThreadSanitizer 和 LeakSanitizer 等工具。但这些工具只能检测用户空间的内存问题。通过在编译时加入指定的选项，就可以给用户程序加入 Address Sanitizer 功能。

**清单 1. 用户空间内存错误代码实例**

|  |  |
| --- | --- |
| 1 | // To compile: g++ -O -g -fsanitize=address use-after-free.c int main(int argc, char \*\*argv) { int \*array = new int[10]; delete [] array; return array[argc]; // BOOM } |

当运行以上有内存使用错误的程序时，加入 Address Sanitizer 功能的的版本会报告如下的错误信息，而没有任何选项的版本则会正常结束程序。

**清单 2. Address Sanitizer 运行结果**

|  |  |
| --- | --- |
| 1 | tengrui@virtualbox:~/workspace/cc$ g++ -O -g -fsanitize=address use-after-free.cc tengrui@virtualbox:~/workspace/cc$ ./a.out ================================================================= ==4206==ERROR: AddressSanitizer: heap-use-after-free on address 0x60400000dfd4 at pc 0x0000004007d4 bp 0x7ffdfdd414f0 sp 0x7ffdfdd414e0 READ of size 4 at 0x60400000dfd4 thread T0 #0 0x4007d3 in main /home/tengrui/workspace/cc/use-after-free.cc:4 #1 0x7f8aa150882f in \_\_libc\_start\_main (/lib/x86\_64-linux-gnu/libc.so.6+0x2082f) #2 0x4006b8 in \_start (/home/tengrui/workspace/cc/a.out+0x4006b8) 0x60400000dfd4 is located 4 bytes inside of 40-byte region [0x60400000dfd0,0x60400000dff8) freed by thread T0 here: #0 0x7f8aa194abca in operator delete[](void\*) (/usr/lib/x86\_64-linux-gnu/libasan.so.2+0x99bca) #1 0x4007a7 in main /home/tengrui/workspace/cc/use-after-free.cc:3 previously allocated by thread T0 here: #0 0x7f8aa194a5d2 in operator new[](unsigned long) (/usr/lib/x86\_64-linux-gnu/libasan.so.2+0x995d2) #1 0x400797 in main /home/tengrui/workspace/cc/use-after-free.cc:2 SUMMARY: AddressSanitizer: heap-use-after-free /home/tengrui/workspace/cc/use-after-free.cc:4 main Shadow bytes around the buggy address: 0x0c087fff9ba0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c087fff9bb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c087fff9bc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c087fff9bd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c087fff9be0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa =&gt;0x0c087fff9bf0: fa fa fa fa fa fa fa fa fa fa[fd]fd fd fd fd fa 0x0c087fff9c00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c087fff9c10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c087fff9c20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c087fff9c30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c087fff9c40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa Shadow byte legend (one shadow byte represents 8 application bytes): Addressable: 00 Partially addressable: 01 02 03 04 05 06 07 Heap left redzone: fa Heap right redzone: fb Freed heap region: fd Stack left redzone: f1 Stack mid redzone: f2 Stack right redzone: f3 Stack partial redzone: f4 Stack after return: f5 Stack use after scope: f8 Global redzone: f9 Global init order: f6 Poisoned by user: f7 Container overflow: fc Array cookie: ac Intra object redzone: bb ASan internal: fe ==4206==ABORTING |

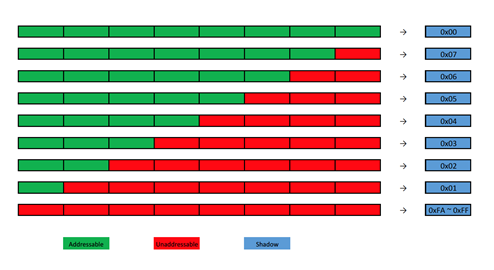
Andrey Ryabinin 借鉴了 AddressSanitizer 的思想，并在 Linux 内核中实现了 Kernel Address Sanitizer。所以 Kasan 也可以看成是用于内核空间的 Address Sanitizer。

**原理**

Kasan 的原理是利用“额外”的内存来标记那些可以被使用的内存的状态。这些做标记的区域被称为影子区域（shadow region）。了解 Linux 内存管理的读者知道，内存中的每个物理页在内存中都会有一个 struct page 这样的结构体来表示，即每 4KB 的页需要 40B 的结构体，大约 1% 的内存用来表示内存本身。Kasan 与其类似但“浪费”更为严重，影子区域的比例是 1:8，即总内存的九分之一会被“浪费”。用官方文档中的例子，如果有 128TB 的可用内存，需要有额外 16TB 的内存用来做标记。

做标记的方法比较简单，将可用内存按照 8 子节的大小分组，如果每组中所有 8 个字节都可以访问，则影子内存中相应的地方用全零（0x00）表示；如果可用内存的前 N（1 到 7 范围之间）个字节可用，则影子内存中响应的位置用 N 表示；其它情况影子内存用负数表示该内存不可用。

**图 1. Kasan 内存布局原理**



**使用**

Kasan 是内核的一部分，使用时需要重新配置、编译并安装内核。Kasan 在 Linux 内核 4.0 版本时被引入内核，所以选择的内核代码需要高于 4.0 版本。另外，最基本的 Kasan 功能需要 GCC4.9.2 支持，更多的支持则需要 GCC5.0 及以上版本。

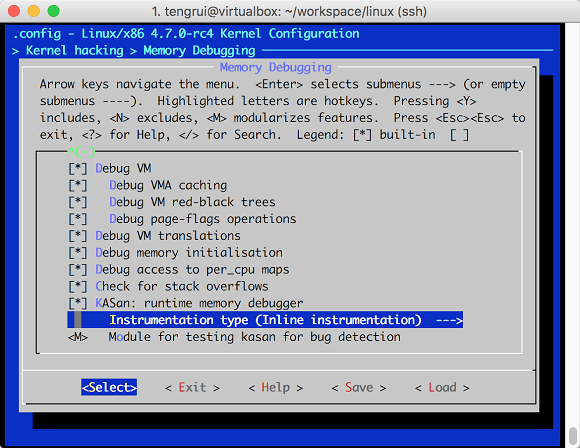
首先是配置和编译内核。

运行如下命令启动图形配置界面：

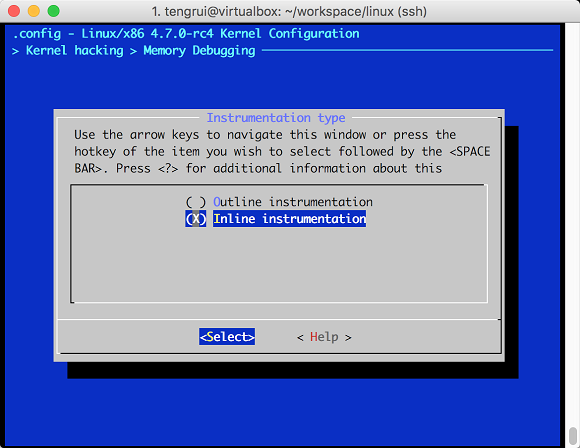
**清单 3. Linux 图形配置命令**

|  |  |
| --- | --- |
| 1 | make menuconfig |

**图 2. Kasan 内核选项配置界面**



**图 3. Kasan 模式选项**



然后重新编译并安装内核即可，除了通用的编译和安装命令，在 Fedora 这种发行版本中，还需要更新 grub。

**清单 4. Linux 内核编译、安装命令**

|  |  |
| --- | --- |
| 1 | make menuconfig make sudo make modules\_install sudo make install |

**清单 5. Grub 配置命令**

|  |  |
| --- | --- |
| 1 | sudo grub2mkconfig – o /boot/grub/grub.cfg |

其它发行版本请参考相关文档。

**测试**

学习 Linux 的同学一定对 Linus 的名言“Talk is cheap, show me the code.”耳熟能详。由于 Kasan 这部分代码一直在变化之中，更重要的是代码也比较难懂，本文暂时不去讨论具体的实现细节，而是从测试的角度研究其原理。

幸运的是 Linux 内核的源码中已经包含了针对 Kasan 的测试代码，其位置在 linux/lib/test\_kasan.c。编译内核或者单独编译 lib 模块的时候，会生成 test\_kasan.ko 模块。当向内核插入该模块的时候，就会执行测试代码。

例如，下面的代码模拟了内存越界的情况：申请了 124 字节的空间，却写访问第 125 个字节的内容，则会造成越界访问的问题。

**清单 6. Kasan 内存右侧越界测试代码**

|  |  |
| --- | --- |
| 1 | static noinline void \_\_init kmalloc\_oob\_right(void) { char \*ptr; size\_t size = 124; pr\_info("out-of-bounds to right\n"); ptr = kmalloc(size, GFP\_KERNEL); if (!ptr) { pr\_err("Allocation failed\n"); return; } pr\_info("ptr address: 0x%lx\n", ptr); ptr[size] = 'x'; pr\_info("ptr[size] address: 0x%lx\n", ptr + size); kfree(ptr); } |

当运行以上测试代码的时候，在内核日志中会详细打印以下内容：

**清单 7. 内核日志**

|  |  |
| --- | --- |
| 1 | [18319.272272] kasan test: kmalloc\_oob\_right out-of-bounds to right [18319.272288] kasan test: kmalloc\_oob\_right ptr address: 0xffff8800d40b9798 [18319.272292] ===================================================== [18319.272996] BUG: KASAN: slab-out-of-bounds in kmalloc\_oob\_right+0xb4/0xdb [test\_kasan] at addr ffff8800d40b9814 [18319.274250] Write of size 1 by task insmod/4852 [18319.274992] =========================================== ================================== [18319.275982] BUG kmalloc-128 (Tainted: G B OE ): kasan: bad access detected [18319.276103] ---------------------------------------------------- [18319.276103] INFO: Allocated in 0xffff8800d40b9d08 age=18446723238827703540 cpu=0 pid=0 [18319.276103] kmalloc\_oob\_right+0x53/0xdb [test\_kasan] [18319.276103] \_\_\_slab\_alloc+0x4da/0x540 [18319.276103] \_\_slab\_alloc+0x20/0x40 [18319.276103] kmem\_cache\_alloc\_trace+0x1f8/0x270 [18319.276103] kmalloc\_oob\_right+0x53/0xdb [test\_kasan] [18319.276103] kmalloc\_tests\_init+0x9/0xf25 [test\_kasan] [18319.276103] do\_one\_initcall+0xa9/0x230 [18319.276103] do\_init\_module+0x1d0/0x4de [18319.276103] load\_module+0x74d0/0x9ca0 [18319.276103] SYSC\_finit\_module+0x190/0x1d0 [18319.276103] SyS\_finit\_module+0xe/0x10 [18319.276103] entry\_SYSCALL\_64\_fastpath+0x1e/0xa8 [18319.276103] INFO: Freed in 0x10044bcf2 age=18446723238827703542 cpu=0 pid=0 [18319.276103] load\_elf\_binary+0x219/0x4400 [18319.276103] \_\_slab\_free+0x17f/0x2d0 [18319.276103] kfree+0x18a/0x1d0 [18319.276103] load\_elf\_binary+0x219/0x4400 [18319.276103] search\_binary\_handler+0x151/0x420 [18319.276103] do\_execveat\_common.isra.36+0xfd9/0x1d20 [18319.276103] SyS\_execve+0x3a/0x50 [18319.276103] do\_syscall\_64+0x19c/0x3b0 [18319.276103] return\_from\_SYSCALL\_64+0x0/0x6a [18319.276103] INFO: Slab 0xffffea0003502e00 objects=17 used=14 fp=0xffff8800d40b8748 flags=0x1ffff0000004080 [18319.276103] INFO: Object 0xffff8800d40b9790 @offset=6032 fp=0xcccccccccccccccc [18319.276103] Redzone ffff8800d40b9788: 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZ [18319.276103] Object ffff8800d40b9790: cc cc cc cc cc cc cc cc 6b 6b 6b 6b 6b 6b 6b 6b ........kkkkkkkk [18319.276103] Object ffff8800d40b97a0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk [18319.276103] Object ffff8800d40b97b0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk [18319.276103] Object ffff8800d40b97c0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk [18319.276103] Object ffff8800d40b97d0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk [18319.276103] Object ffff8800d40b97e0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk [18319.276103] Object ffff8800d40b97f0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk [18319.276103] Object ffff8800d40b9800: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk [18319.276103] Redzone ffff8800d40b9810: 6b 6b 6b 6b 6b 6b 6b a5 kkkkkkk. [18319.276103] Padding ffff8800d40b9950: f0 bc 44 00 01 00 00 00 ..D..... [18319.276103] CPU: 0 PID: 4852 Comm: insmod Tainted: G B OE 4.7.0-rc4+ #25 [18319.276103] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006 [18319.276103] 0000000000000000 00000000bfaee01f ffff8800d40a7810 ffffffff81b6fd61 [18319.276103] ffff88011f603440 ffff8800d40b9790 ffff8800d40a7840 ffffffff8157b472 [18319.276103] ffff88011f603440 ffffea0003502e00 ffff8800d40b9790 ffffffffc03780db [18319.276103] Call Trace: [18319.276103] [&lt;ffffffff81b6fd61&gt;] dump\_stack+0x63/0x82 [18319.276103] [&lt;ffffffff8157b472&gt;] print\_trailer+0x112/0x1a0 [18319.276103] [&lt;ffffffffc03780db&gt;] ? kmalloc\_oob\_right+0xdb/0xdb [test\_kasan] [18319.276103] [&lt;ffffffff815815f4&gt;] object\_err+0x34/0x40 [18319.276103] [&lt;ffffffff81583a82&gt;] kasan\_report\_error+0x222/0x540 [18319.276103] [&lt;ffffffff8146af5c&gt;] ? power\_down+0xc4/0xc4 [18319.276103] [&lt;ffffffffc03780db&gt;] ? kmalloc\_oob\_right+0xdb/0xdb [test\_kasan] [18319.276103] [&lt;ffffffff81584031&gt;] \_\_asan\_report\_store1\_noabort+0x61/0x70 [18319.276103] [&lt;ffffffffc03780b4&gt;] ? kmalloc\_oob\_right+0xb4/0xdb [test\_kasan] [18319.276103] [&lt;ffffffffc03780b4&gt;] kmalloc\_oob\_right+0xb4/0xdb [test\_kasan] [18319.276103] [&lt;ffffffffc03780e4&gt;] kmalloc\_tests\_init+0x9/0xf25 [test\_kasan] [18319.276103] [&lt;ffffffff81002299&gt;] do\_one\_initcall+0xa9/0x230 [18319.276103] [&lt;ffffffff810021f0&gt;] ? initcall\_blacklisted+0x180/0x180 [18319.276103] [&lt;ffffffff815830c6&gt;] ? kasan\_unpoison\_shadow+0x36/0x50 [18319.276103] [&lt;ffffffff815830c6&gt;] ? kasan\_unpoison\_shadow+0x36/0x50 [18319.276103] [&lt;ffffffff8158313e&gt;] ? kasan\_kmalloc+0x5e/0x70 [18319.276103] [&lt;ffffffff815830c6&gt;] ? kasan\_unpoison\_shadow+0x36/0x50 [18319.276103] [&lt;ffffffff815831d7&gt;] ? \_\_asan\_register\_globals+0x87/0xa0 [18319.276103] [&lt;ffffffff8146b8a3&gt;] do\_init\_module+0x1d0/0x4de [18319.276103] [&lt;ffffffff812eea50&gt;] load\_module+0x74d0/0x9ca0 [18319.276103] [&lt;ffffffff812e3d00&gt;] ? m\_show+0x4a0/0x4a0 [18319.276103] [&lt;ffffffff812e7580&gt;] ? module\_frob\_arch\_sections+0x20/0x20 [18319.276103] [&lt;ffffffff815c421d&gt;] ? rw\_verify\_area+0xbd/0x2b0 [18319.276103] [&lt;ffffffff81534d05&gt;] ? \_\_vmalloc\_node\_range+0x485/0x630 [18319.276103] [&lt;ffffffff815d4a89&gt;] ? kernel\_read\_file\_from\_fd+0x49/0x80 [18319.276103] [&lt;ffffffff812f15d0&gt;] SYSC\_finit\_module+0x190/0x1d0 [18319.276103] [&lt;ffffffff812f1440&gt;] ? SYSC\_init\_module+0x220/0x220 [18319.276103] [&lt;ffffffff814d1790&gt;] ? vma\_is\_stack\_for\_task+0x90/0x90 [18319.276103] [&lt;ffffffff815d1036&gt;] ? vfs\_getattr+0x26/0x30 [18319.276103] [&lt;ffffffff812f162e&gt;] SyS\_finit\_module+0xe/0x10 [18319.276103] [&lt;ffffffff828bb8b6&gt;] entry\_SYSCALL\_64\_fastpath+0x1e/0xa8 [18319.276103] Memory state around the buggy address: [18319.276103] ffff8800d40b9700: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc [18319.276103] ffff8800d40b9780: fc fc fc 00 00 00 00 00 00 00 00 00 00 00 00 00 [18319.276103] &gt;ffff8800d40b9800: 00 00 04 fc fc fc fc fc fc fc fc fc fc fc fc fc [18319.276103] ^ [18319.276103] ffff8800d40b9880: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc [18319.276103] ffff8800d40b9900: fc fc fc fc fc fc fc fc fc fc fc fc fc 00 00 00 [18319.276103] ================================================================== [18319.333357] kasan test: kmalloc\_oob\_right ptr[size] address: 0xffff8800d40b9814 |

其中能直观看出 Kasan 原理的是 Memory state around the buggy address，在一堆 fc 字节中有一串 00 以及一个 04。如前所述，每个 00 代表 8 个可用字节，04 代表该对应的地址中前四个字节可用。这里共有 15 个 00，一个 04。15 x 8 + 4 = 124，正是代码中申请的 124 字节。当测试代码往第 125 个字节中写入数据的时候，Kasan 就会检测到该行为并报告一系列的相关信息。其中 fc 表示的是 slub 对象中的红色区域，其它填充值的意义可以参考以下定义。

**清单 8. 填充值的定义**

|  |  |
| --- | --- |
| 1 | #define KASAN\_FREE\_PAGE 0xFF /\* page was freed \*/ #define KASAN\_PAGE\_REDZONE 0xFE /\* redzone for kmalloc\_large allocations \*/ #define KASAN\_KMALLOC\_REDZONE 0xFC /\* redzone inside slub object \*/ #define KASAN\_KMALLOC\_FREE 0xFB /\* object was freed (kmem\_cache\_free/kfree) \*/ #define KASAN\_GLOBAL\_REDZONE 0xFA /\* redzone for global variable \*/ |

该测试代码包含了许多其它的测试用例，有兴趣的读者可以参考如下代码清单有选择地编译并运行。

**清单 9. Kasan 测试用例集**

|  |  |
| --- | --- |
| 1 | static int \_\_init kmalloc\_tests\_init(void) { kmalloc\_oob\_right(); kmalloc\_oob\_left(); kmalloc\_node\_oob\_right(); #ifdef CONFIG\_SLUB kmalloc\_pagealloc\_oob\_right(); #endif kmalloc\_large\_oob\_right(); kmalloc\_oob\_krealloc\_more(); kmalloc\_oob\_krealloc\_less(); kmalloc\_oob\_16(); kmalloc\_oob\_in\_memset(); kmalloc\_oob\_memset\_2(); kmalloc\_oob\_memset\_4(); kmalloc\_oob\_memset\_8(); kmalloc\_oob\_memset\_16(); kmalloc\_uaf(); kmalloc\_uaf\_memset(); kmalloc\_uaf2(); kmem\_cache\_oob(); kasan\_stack\_oob(); kasan\_global\_oob(); ksize\_unpoisons\_memory(); copy\_user\_test(); return -EAGAIN; } |

**对比**

和 Kasan 功能类似的工具还有 kmemcheck，它比 Kasan 更早加入内核，但是运行速度没有 Kasan 快，这是因为 Kasan 利用了编译器的特性，可以将代码编译为內联模式。但 Kasan 也有自己的不足，目前 Kasan 不能检测出读取未初始化内存的错误，而这一点 kmemcheck 是支持的。

此外，内核还包含了一些配置选项可以打开其它的内存检测功能，如 SLAB\_DEBUG 和 DEBUG\_SLAB 选项可以激活 redzones 和 poisoning 功能，用来检测申请和释放内存的错误。当打开 DEBUG\_PAGEALLOC 选项后，可以检测部分释放后使用内存的情况。

这些都是内核代码质量的保证工具，当提交代码的时候，综合使用以上工具可以预防自己的补丁引入一些低级的错误。

**结束语**

本文介绍了 Kasan 的配置及使用方法，并通过运行 Kasan 的测试用例说明了 Kasan 的原理。对于内核开发者来说，该工具不仅可以用来检测自己代码。对该工具有兴趣的读者，也可以给该工具增加新功能或发现并修复其中的 BUG。

**相关主题**

* 参考 [Clang 文档](http://clang.llvm.org/docs/AddressSanitizer.html) 中关于 AddressSanitizer 的内容。
* 参考 [LWN 文档：The kernel address sanitizer](https://lwn.net/Articles/612153/)：Linux 每周新闻关于 Kasan 的介绍文章。
* 参考 [LWN文档：Kernel address sainitzer (KASan) - dynamic memory error deetector](https://lwn.net/Articles/611410/)：Linux 每周新闻中关于 Kasan 的建议及补丁的收录。
* 参考 [Linux 内核文档](https://www.kernel.org/doc/Documentation/kmemcheck.txt)，了解关于 memcheck 的介绍。
* 参考 [LinuxCon North America 2015 KernelAddressSanitizer.pdf](http://119.90.25.37/events.linuxfoundation.org/sites/events/files/slides/LinuxCon%20North%20America%202015%20KernelAddressSanitizer.pdf)：Andrey Konovalov 和 Dmitry Vyukov 在 LinuxCon North America 2015 会议上的演讲稿。
* [IBM developerWorks 中国 linux 专区](http://www.ibm.com/developerworks/cn/linux/)：为使用 linux 的开发人员准备的技术信息和资料。这里提供产品下载、how-to 信息、支持资源以及免费技术库，包含 2000 多份技术文章、教程、最佳实践、IBM Redbook 和在线产品手册。