

调不尽的内存泄漏，用不完的 Valgrind

Table of Contents

- [1. Valgrind 安装](#)
- [2. Valgrind 包含的工具](#)
 - [2.1. memcheck](#)
 - [2.2. cachegrind](#)
 - [2.3. helgrind](#)
 - [2.4. Callgrind](#)
 - [2.5. Massif](#)
 - [2.6. LACKY](#)
- [3. Valgrind 的参数](#)
 - [3.1. 常用选项](#)
 - [3.2. 适用于 Memcheck 工具的相关选项:](#)
- [4. Valgrind 的使用](#)
- [5. Valgrind 的示例](#)
 - [5.1. 使用未初始化的内存](#)
 - [5.1.1. 内存读写越界](#)
 - [5.1.2. src 和 dst 内存覆盖](#)
 - [5.1.3. 动态内存管理错误](#)
 - [5.1.4. 内存泄漏](#)
 - [5.1.5. 非法读写](#)
 - [5.1.6. 无效指针](#)
 - [5.1.7. 重复释放](#)
- [6. Valgrind 的局限](#)
- [7. 附录 A: 参数指令](#)
 - [7.1. 基本选项:](#)
 - [7.2. 错误相关选项:](#)
 - [7.3. MALLOC\(\) 相关的选项:](#)
 - [7.4. 非通用选项:](#)
 - [7.5. 调试 VALGRIND 选项:](#)
 - [7.5.1. 内存检查选项:](#)
 - [7.5.2. CACHETRIND 选项:](#)
 - [7.6. CALLGRIND 选项:](#)
 - [7.7. HELGRIND 选项:](#)
 - [7.8. LACKY 选项:](#)

1 Valgrind 安装

```
wget http://valgrind.org/downloads/valgrind-3.11.0.tar.bz2
tar -jxvf ~/Downloads/valgrind-*.tar.bz2
cd valgrind-3.11.0
./autogen.sh
./configure && make && sudo make install
```

2 Valgrind 包含的工具

Valgrind 支持很多工具：memcheck, addrcheck, cachegrind, Massif, helgrind 和 Callgrind 等。在运行 Valgrind 时，你必须指明想用的工具，如果省略工具名，默认运行 memcheck。

2.1 memcheck

memcheck 探测程序中内存管理存在的问题。它检查所有对内存的读/写操作，并截取所有的 malloc/new/free/delete 调用。因此 memcheck 工具能够探测到以下问题：

- 使用未初始化的内存
- 读/写已经被释放的内存
- 读/写内存越界
- 读/写不恰当的内存栈空间
- 内存泄漏
- 使用 `malloc/new/new[]` 和 `free/delete/delete[]` 不匹配。
- `src` 和 `dst` 的重叠

2.2 cachegrind

cachegrind 是一个 cache 剖析器。它模拟执行 CPU 中的 L1, D1 和 L2 cache，因此它能很精确的指出代码中的 cache 未命中。如果你需要，它可以打印出 cache 未命中的次数，内存引用和发生 cache 未命中的每一行代码，每一个函数，每一个模块和整个程序的摘要。如果你要求更细致的信息，它可以打印出每一行机器码的未命中次数。在 x86 和 amd64 上，cachegrind 通过 CPUID 自动探测机器的 cache 配置，所以在多数情况下它不再需要更多的配置信息了。

2.3 helgrind

它主要用来检查多线程程序中出现的竞争问题。Helgrind 寻找内存中被多个线程访问，而又没有一贯加锁的区域，这些区域往往是线程之间失去同步的地方，而且会导致难以发掘的错误。Helgrind 实现了名为“Eraser”的竞争检测算法，并做了进一步改进，减少了报告错误的次数。

2.4 Callgrind

Callgrind 收集程序运行时的一些数据，函数调用关系等信息，还可以有选择地进行 cache 模拟。在运行结束时，它会把分析数据写入一个文件。`callgrind_annotate` 可以把这个文件的内容转化成可读的形式。

一般用法：

```
valgrind --tool=callgrind ./sec_infod
```

会在当前目录下生成 `callgrind.out.[pid]`，如果我们想结束程序，可以

```
killall callgrind
```

然后我们可以用

```
callgrind_annotate --auto=yes callgrind.out.[pid] > log  
vi log
```

2.5 Massif

堆栈分析器，它能测量程序在堆栈中使用了多少内存，告诉我们堆块，堆管理块和栈的大小。Massif 能帮助我们减少内存的使用，在带有虚拟内存的现代系统中，它还能够加速我们程序的运行，减少程序停留在交换区中的几率。

2.6 lackey

lackey 是一个示例程序，以其为模版可以创建你自己的工具。在程序结束后，它打印出一些基本的关于程序执行统计数据。

3 Valgrind 的参数

用法：

```
valgrind [options] prog-and-args [options]:
```

3.1 常用选项

适用于所有 Valgrind 工具。

- `--tool=<name>` 最常用的选项。运行 valgrind 中名为 `toolname` 的工具。默认 memcheck。

- `-h --help`

显示所有选项的帮助，包括内核和选定的工具两者。

- `--version`

显示 valgrind 内核的版本，每个工具都有各自的版本。

- `-q --quiet`

安静地运行，只打印错误信息。

- `--verbose`

更详细的信息。

- `--trace-children=<yes|no>`

跟踪子线程？ [default: no]

- `--track-fds=<yes|no>`

跟踪打开的文件描述？ [default: no]

- `--time-stamp=<yes|no>`

增加时间戳到 LOG 信息? [default: no]

- `--log-fd=<number>`

输出 LOG 到描述符文件 [2=stderr]

- `--log-file=<file>`

将输出的信息写入到 `filename.PID` 的文件里, PID 是运行程序的进程 ID

- `--log-file-exactly=<file>`

输出 LOG 信息到 file, LOG 信息输出

- `--xml=yes`

将信息以 xml 格式输出, 只有 memcheck 可用

- `--num-callers=<number>`

`show <number> callers in stack traces`

- `--error-exitcode=<number>`

如果发现错误则返回错误代码 [0=disable]

- `--db-attach=<yes|no>`

当出现错误, valgrind 会自动启动调试器 gdb。[default: no]

- `--db-command=<command>`

启动调试器的命令行选项[gdb -nw %f %p]

3.2 适用于 Memcheck 工具的相关选项:

- `--leak-check=<no|summary|full>`

要求对 leak 给出详细信息? Leak 是指, 存在一块没有被引用的内存空间, 或没有被释放的内存空间, 如 summary, 只反馈一些总结信息, 告诉你有多少个 malloc, 多少个 free 等; 如果是 full 将输出所有的 leaks, 也就是定位到某一个 malloc/free。

- `--show-reachable=<yes|no>`

如果为 no, 只输出没有引用的内存 leaks, 或指向 malloc 返回的内存块中部某处的 leaks [default: no]

更详细的参数指令见附录 A。

4 Valgrind 的使用

首先, 在编译程序的时候打开调试模式 (gcc 编译器的-g 选项)。如果没有调试信息, 即使最好的 valgrind 工具也将中能够猜测特定的代码是属于哪一个函数。打开调试选项进行编译后再用 valgrind 检查, valgrind 将会给你一个详细的报告, 比如哪一行代码出现了内存泄漏。

当检查的是 C++ 程序的时候，还应该考虑另一个选项 `-fno-inline`。它使得函数调用链很清晰，这样可以减少你在浏览大型 C++ 程序时的混乱。比如在使用这个选项的时候，用 `memcheck` 检查 `openoffice` 就很容易。当然，你可能不会做这项工作，但是使用这一选项使得 `valgrind` 生成更精确的错误报告和减少混乱。

一些编译优化选项(比如`-O2` 或者更高的优化选项)，可能会使得 `memcheck` 提交错误的未初始化报告，因此，为了使得 `valgrind` 的报告更精确，在编译的时候最好不要使用优化选项。

如果程序是通过脚本启动的，可以修改脚本里启动程序的代码，或者使用 `--trace-children=yes` 选项来运行脚本。

5 Valgrind 的示例

5.1 使用未初始化的内存

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int x;
    if(x == 0) {
        printf("X is zero");
    }
    return 0;
}
```

运行：

```
gcc -g uninit-mem.c -o uninit-mem
valgrind --tool=memcheck ./uninit-mem
```

我们可以看到：

- 左边显示类似行号的数字 (10297) 表示的是 Process ID。
- 最上面的红色方框表示的是 `valgrind` 的版本信息。
- 中间的红色方框表示 `valgrind` 通过运行被测试程序，发现的内存问题。通过阅读这些信息，可以发现：
 - 这是一个对内存的非法写操作，非法写操作的内存是 4 bytes。
 - 发生错误时的函数堆栈，以及具体的源代码行号。
 - 非法写操作的具体地址空间。
- 最下面的红色方框是对发现的内存问题和内存泄漏问题的总结。内存泄漏的大小 (40 bytes) 也能够被检测出来。

5.1.1 内存读写越界

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int len=5;
    int i;
    int *pt=(int*)malloc(len*sizeof(int));
```

```

int *p=pt;
for(i=0;i<len;i++)
{p++;}
*p=5;
printf( "%d" , *p);

return;
}

```

5.1.2 src 和 dst 内存覆盖

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char x[50];
    int i;
    for(i=0;i<50;i++)
    {
        x[i]=i;
    }
    strncpy(x+20, x, 20); //Good
    strncpy(x+20, x, 21); //Overlap
    x[39]=' \' ;
    strcpy(x, x+20); //Good
    x[39]=40;
    x[40]=' \' ;
    strcpy(x, x+20); //Overlap

    return 0;
}

```

5.1.3 动态内存管理错误

常见的内存分配方式分三种：静态存储，栈上分配，堆上分配。全局变量属于静态存储，它们是在编译时就被分配了存储空间，函数内的局部变量属于栈上分配，而最灵活的内存使用方式当属堆上分配，也叫做内存动态分配了。常用的内存动态分配函数包括：malloc, alloc, realloc, new 等，动态释放函数包括 free, delete。

一旦成功申请了动态内存，我们就需要自己对其进行内存管理，而这又是最容易犯错误的。常见的内存动态管理错误包括：

- 申请和释放不一致
- 由于 C++ 兼容 C，而 C 与 C++ 的内存申请和释放函数是不同的，因此在 C++ 程序中，就有两套动态内存管理函数。一条不变的规则就是采用 C 方式申请的内存就用 C 方式释放；用 C++ 方式申请的内存，用 C++ 方式释放。也就是用 malloc/alloc/realloc 方式申请的内存，用 free 释放；用 new 方式申请的内存用 delete 释放。在上述程序中，用 malloc 方式申请了内存却用 delete 来释放，虽然这在很多情况下不会有大问题，但这绝对是潜在的问题。
- 申请和释放不匹配
- 申请了多少内存，在使用完成后就要释放多少。如果没有释放，或者少释放了就是内存泄露；多释放了也会产生问题。上述程序中，指针 p 和 pt 指向的是同一块内存，却被先后释放两次。

- 释放后仍然读写，本质上说，系统会在堆上维护一个动态内存链表，如果被释放，就意味着该块内存可以继续被分配给其他部分，如果内存被释放后再访问，就可能覆盖其他部分的信息，这是一种严重的错误。

下面的一段程序，就包括了内存动态管理中常见的错误。

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    char *p=(char*)malloc(10);
    char *pt=p;
    int i;
    for(i=0; i<10; i++)
    {
        p[i]='z';
    }
    delete p;
    p[1]='a';
    free(pt);

    return 0;
}
```

5.1.4 内存泄漏

```
#include <stdlib.h>

int main()
{
    char *x = (char*)malloc(20);
    char *y = (char*)malloc(20);
    x=y;
    free(x);
    free(y);

    return 0;
}
```

5.1.5 非法读写

```
int main()
{
    int i, *x;
    x = (int *)malloc(10*sizeof(int));
    for (i=0; i<11; i++)
        x[i] = i;
    free(x);

    return 0;
}
```

5.1.6 无效指针

```
#include <stdlib.h>

int main()
{
    char *x = malloc(10);
    x[10] = 'a';
    free(x);

    return 0;
}
```

5.1.7 重复释放

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *x = malloc(10);
    free(x);
    free(x);

    return 0;
}
```

6 Valgrind 的局限

Valgrind 不对静态数组 (分配在栈上) 进行边界检查。如果在程序中声明了一个数组：

```
int main(int argc, char *argv[])
{
    char x[10];
    x[11] = 'a';
    return 0;
}
```

Valgrind 则不会警告你，你可以把数组改为动态在堆上分配的数组，这样就可能进行边界检查了。这个方法好像有点得不偿失的感觉。

Valgrind 占用了更多的内存—可达两倍于你程序的正常使用量。如果你用 Valgrind 来检测使用大量内存的程序就会遇到问题，它可能会用很长的时间来运行测试。大多数情况下，这都不是问题，即使速度慢也仅是检测时速度慢，如果你用 Valgrind 来检测一个正常运行时速度就很慢的程序，这下问题就大了。Valgrind 不可能检测出你在程序中犯下的所有错误—如果你不检查缓冲区溢出，Valgrind 也不会告诉你代码写了它不应该写的内存。

7 附录 A：参数指令

7.1 基本选项：

这些选项对所有工具都有效。

- -h --help

显示所有选项的帮助，包括内核和选定的工具两者。

- `--help-debug`

和 `--help` 相同，并且还能显示通常只有 Valgrind 的开发人员使用的调试选项。

- `--version`

显示 Valgrind 内核的版本号。工具可以有他们自己的版本号。这是一种保证工具只在它们可以运行的内核上工作的一种设置。这样可以减少在工具和内核之间版本兼容性导致奇怪问题的概率。

- `-q --quiet`

安静的运行，只打印错误信息。在进行回归测试或者有其它的自动化测试机制时会非常有用。

- `-v --verbose`

显示详细信息。在各个方面显示你的程序的额外信息，例如：共享对象加载，使用的重置，执行引擎和工具的进程，异常行为的警告信息。重复这个标记可以增加详细的级别。

- `-d`

调试 Valgrind 自身发出的信息。通常只有 Valgrind 开发人员对此感兴趣。重复这个标记可以产生更详细的输出。如果你希望发送一个 bug 报告，通过 `-v -v -d -d` 生成的输出会使你的报告更加有效。

- `--tool=<toolname> [default: memcheck]`

运行 `toolname` 指定的 Valgrind，例如，Memcheck, Addrcheck, Cachegrind, 等等。

- `--trace-children=<yes|no> [default: no]`

当这个选项打开时，Valgrind 会跟踪到子进程中。这经常会导致困惑，而且通常不是你所期望的，所以默认这个选项是关闭的。

- `--track-fds=<yes|no> [default: no]`

当这个选项打开时，Valgrind 会在退出时打印一个打开文件描述符的列表。每个文件描述符都会打印出一个文件是在哪里打开的栈回溯，和任何与此文件描述符相关的详细信息比如文件名或 socket 信息。

- `--time-stamp=<yes|no> [default: no]`

当这个选项打开时，每条信息之前都有一个从程序开始消逝的时间，用天，小时，分钟，秒和毫秒表示。

- `--log-fd=<number> [default: 2, stderr]`

指定 Valgrind 把它所有的消息都输出到一个指定的文件描述符中去。默认值 2，是标准错误输出 (stderr)。注意这可能会干扰到客户端自身对 stderr 的使用，Valgrind 的输出与客户程序的输出将穿插在一起输出到 stderr。

- `--log-file=<filename>`

指定 Valgrind 把它所有的信息输出到指定的文件中。实际上，被创建文件的文件名是由 filename、'.' 和进程号连接起来的（即 <filename>.<pid>），从而每个进程创建不同的文件。

- `--log-file-exactly=<filename>`

类似于 `--log-file`，但是后缀 ".pid" 不会被添加。如果设置了这个选项，使用 Valgrind 跟踪多个进程，可能会得到一个乱七八糟的文件。

- `--log-file-qualifier=<VAR>`

当和 `--log-file` 一起使用时，日志文件名将通过环境变量 \$VAR 来筛选。这对于 MPI 程序是有益的。更多的细节，查看手册 2.3 节 "注解"。

- `--log-socket=<ip-address:port-number>`

指定 Valgrind 输出所有的消息到指定的 IP，指定的端口。当使用 1500 端口时，端口有可能被忽略。如果不能建立一个到指定端口的连接，Valgrind 将输出写到标准错误 (stderr)。这个选项经常和一个 Valgrind 监听程序一起使用。更多的细节，查看手册 2.3 节 "注解"。

7.2 错误相关选项：

这些选项适用于所有产生错误的工具，比如 Memcheck，但是 Cachegrind 不行。

- `--xml=<yes|no> [default: no]`

当这个选项打开时，输出将是 XML 格式。这是为了使用 Valgrind 的输出做为输入的工具，例如 GUI 前端更加容易些。目前这个选项只在 Memcheck 时生效。

- `--xml-user-comment=<string>`

在 XML 开头 附加用户注释，仅在指定了 `--xml=yes` 时生效，否则忽略。

- `--demangle=<yes|no> [default: yes]`

打开 / 关闭 C++ 的名字自动解码。默认打开。当打开时，Valgrind 将尝试着把编码过的 C++ 名字自动转回初始状态。这个解码器可以处理 g++ 版本为 2.X,3.X 或 4.X 生成的符号。一个关于名字编码解码重要的事实是，禁止文件中的解码函数名仍然使用他们未解码的形式。Valgrind 在搜寻可用的禁止条目时不对函数名解码，因为这将使禁止文件内容依赖于 Valgrind 的名字解码机制状态，会使速度变慢，且无意义。

- `~ --num-callers=<number> [default: 12]`

默认情况下，Valgrind 显示 12 层函数调用的函数名有助于确定程序的位置。可以通过这个选项来改变这个数字。这样有助在嵌套调用的层次很深时确定程序的位置。注意错误信息通常只回溯到最顶上的 4 个函数。（当前函数，和它的 3 个调用者的位置）。所以这并不影响报告的错误总数。这个值的最大值是 50。注意高的设置会使 Valgrind 运行得慢，并且使用更多的内存，但是在嵌套调用层次比较高的程序中非常实用。

- `--error-limit=<yes|no> [default: yes]`

当这个选项打开时，在总量达到 10,000,000，或者 1,000 个不同的错误，Valgrind 停止报告错误。这是为了避免错误跟踪机制在错误很多的程序下变成一个巨大的性能负担。

- `--error-exitcode=<number>` [default: 0]

指定如果 Valgrind 在运行过程中报告任何错误时的退出返回值，有两种情况；当设置为默认值(零)时，Valgrind 返回的值将是它模拟运行的程序的返回值。当设置为非零值时，如果 Valgrind 发现任何错误时则返回这个值。在 Valgrind 做为一个测试工具套件的部分使用时这将非常有用，因为使测试工具套件只检查 Valgrind 返回值就可以知道哪些测试用例 Valgrind 报告了错误。

- `--show-below-main=<yes|no>` [default: no]

默认地，错误时的栈回溯不显示 `main()` 之下的任何函数 (或者类似的函数像 glibc 的 `_libc_start_main()`，如果 `main()` 没有出现在栈回溯中)；这些大部分都是令人厌倦的 C 库函数。如果打开这个选项，在 `main()` 之下的函数也将会显示。

- `--suppressions=<filename>` [default: `$PREFIX/lib/valgrind/default.supp`]

指定一个额外的文件读取不需要理会的错误；你可以根据需要使用任意多的额外文件。

- `--gen-suppressions=<yes|no|all>` [default: no]

当设置为 yes 时，Valgrind 将会在每个错误显示之后自动暂停并且打印下面这一行：`--Print suppression ? ~---` [Return/N/n/Y/y/C/c] 个提示的行为和 `--db-attach` 选项 (见下面) 相同。如果选择是，Valgrind 会打印出一个错误的禁止条目，你可以把它剪切然后粘帖到一个文件，如果不希望在将来再看到这个错误信息。当设置为 all 时，Valgrind 会对每一个错误打印一条禁止条目，而不向用户询问。这个选项对 C++ 程序非常有用，它打印出编译器调整过的名字。注意打印出来的禁止条目是尽可能的特定的。如果需要把类似的条目归纳起来，比如在函数名中添加通配符。并且，有些时候两个不同的错误也会产生同样的禁止条目，这时 Valgrind 就会输出禁止条目不止一次，但是在禁止条目的文件中只需要一份拷贝(但是如果多于一份也不会引起什么问题)。并且，禁止条目的名字像 <在这儿输入一个禁止条目的名字>；名字并不是很重要，它只是和 `-v` 选项一起使用打印出所有使用的禁止条目记录。

- `--db-attach=<yes|no>` [default: no]

当这个选项打开时，Valgrind 将会在每次打印错误时暂停并打出如下一行：`-- Attach to debugger ? -` [Return/N/n/Y/y/C/c] 按下回车，或者 N、回车，n、回车，Valgrind 不会对这个错误启动调试器。按下 Y、回车，或者 y、回车，Valgrind 会启动调试器并设定在程序运行的这个点。当调试结束时，退出，程序会继续运行。在调试器内部尝试继续运行程序，将不会生效。按下 C、回车，或者 c、回车，Valgrind 不会启动一个调试器，并且不会再次询问。注意：`--db-attach=yes` 与 `--trace-children=yes` 有冲突。你不能同时使用它们。Valgrind 在这种情况下不能启动。

- `--db-command=<command>` [default: `gdb -nw %f %p`]

通过 `--db-attach` 指定如何使用调试器。默认的调试器是 `gdb`。默认的选项是一个运行时扩展 Valgrind 的模板。`%f` 会用可执行文件的文件名替换，`%p` 会被可执行文件的进程 ID 替换。这指定了 Valgrind 将怎样调用调试器。默认选项不会因为在构造时是否检测到了 GDB 而改变，通常是 `/usr/bin/gdb`。使用这个命令，你可以指定一些调用其它的调试器来替换。给出的这个命令字串可以包括一个或多个 `%p %f` 扩展。每一个 `%p` 实例都被解释成将调试的进程的 PID，每一个 `%f` 实例都被解释成要调试的进程的可执行文件路径。

- `--input-fd=<number>` [default: 0, `stdin`]

使用 `--db-attach=yes` 和 `--gen-suppressions=yes` 选项，在发现错误时，Valgrind 会停下来读取键盘输入。默认地，从标准输入读取，所以关闭了标准输入的程序会有问题。这个选项允许你指定一个文件描述符来替代标准输入读取。

- `--max-stackframe=<number>` [default: 2000000]

栈的最大值。如果栈指针的偏移超过这个数量，Valgrind 则会认为程序是切换到了另外一个栈执行。如果在程序中有大量的栈分配的数组，你可能需要使用这个选项。valgrind 保持对程序栈指针的追踪。如果栈指针的偏移超过了这个数量，Valgrind 假定你的程序切换到了另外一个栈，并且 Memcheck 行为与栈指针的偏移没有超出这个数量将会不同。通常这种机制运转得很好。然而，如果你的程序在栈上申请了大的结构，这种机制将会表现得愚蠢，并且 Memcheck 将会报告大量的非法栈内存访问。这个选项允许把这个阈值设置为其它值。应该只在 Valgrind 的调试输出中显示需要这么做时才使用这个选项。在这种情况下，它会告诉你应该指定的新的阈值。普遍地，在栈中分配大块的内存是一个坏的主意。因为这很容易用光你的栈空间，尤其是在内存受限的系统或者支持大量小堆栈的线程的系统上，因为 Memcheck 执行的错误检查，对于堆上的数据比对栈上的数据要高效很多。如果你使用这个选项，你可能希望考虑重写代码在堆上分配内存而不是在栈上分配。

7.3 MALLOC() 相关的选项:

对于使用自有版本的 `malloc()` (例如 Memcheck 和 massif)，下面的选项可以使用。

- `--alignment=<number>` [default: 8]

默认 Valgrind 的 `malloc()`,`realloc()`, 等等，是 8 字节对齐地址的。这是大部分处理器的标准。然而，一些程序可能假定 `malloc()` 等总是返回 16 字节或更多对齐的内存。提供的数值必须在 8 和 4096 区间之内，并且必须是 2 的幂数。

7.4 非通用选项:

这些选项可以用于所有的工具，它们影响 Valgrind core 的几个特性。大部分人不会用到这些选项。

- `--run-libc-freeres=<yes|no>` [default: yes]

GNU C 库 (`libc.so`)，所有程序共用的，可能会分配一部分内存自己用。通常在程序退出时释放内存并不麻烦，这里没什么问题，因为 Linux 内核在一个进程退出时会回收进程全部的资源，所以这只是会造成速度慢。glibc 的作者认识到这样会导致内存检查器，像 Valgrind，在退出时检查内存错误的报告 glibc 的内存泄漏问题，为了避免这个问题，他们提供了一个 `_libc_freeres()` 例程特别用来让 glibc 释放分配的所有内存。因此 Memcheck 在退出时尝试着去运行 `_libc_freeres()`。不幸的是，在 glibc 的一些版本中，`_libc_freeres` 是有 bug 会导致段错误的。这在 Red Hat 7.1 上有特别声明。所以，提供这个选项来决定是否运行 `_libc_freeres`。如果你的程序看起来在 Valgrind 上运行得很好，但是在退出时发生段错误，你可能需要指定 `--run-libc-freeres=no` 来修正，这将可能错误的报告 `libc.so` 的内存泄漏。

- `--sim-hints=hint1, hint2, ...`

传递杂凑的提示给 Valgrind，轻微的修改模拟行为的非标准或危险方式，可能有助于模拟奇怪的特性。默认没有提示打开。小心使用！目前已知的提示有：

- `lax-ioctls`: 对 `ioctl` 的处理非常不严格，唯一的假定是大小是正确的。不需要在写时缓冲区完全的初始化。没有这个，用大量的奇怪的 `ioctl` 命令来使用一些设备驱动将会非常烦

人。

- enable-inner: 打开某些特殊的效果，当运行的程序是 Valgrind 自身时。
 - --kernel-variant=variant1, variant2, ...
- 处理系统调用和 ioctls 在这个平台的默认核心上产生不同的变量。这有助于运行在改进过的内核或者支持非标准的 ioctls 上。小心使用。如果你不理解这个选项做的是什么那你几乎不需要它。已经知道的变量有：
- bproc: 支持 X86 平台上的 sys_bproc 系统调用。这是为了运行在 BProc，它是标准 Linux 的一个变种，有时用来构建集群。
- --show-emwarns=<yes|no> [default: no]

当这个选项打开时，Valgrind 在一些特定的情况下将对 CPU 仿真产生警告。通常这些都是不引人注意的。

- --smc-check=<none|stack|all> [default: stack]

这个选项控制 Valgrind 对自我修改的代码的检测。Valgrind 可以不做检测，可以检测栈中自我修改的代码，或者任意地方检测自我修改的代码。注意默认选项是捕捉绝大多数情况，到目前我们了解的情况为止。使用 all 选项时会极大的降低速度。(但是用 none 选项运行极少影响速度，因为对大多数程序，非常少的代码被添加到栈中)

7.5 调试 VALGRIND 选项：

还有一些选项是用来调试 Valgrind 自身的。在运行一般的东西时不应该需要的。如果你希望看到选项列表，使用 --help-debug 选项。

7.5.1 内存检查选项：

- --leak-check=<no|summary|yes|full> [default: summary]

当这个选项打开时，当客户程序结束时查找内存泄漏。内存泄漏意味着有用 malloc 分配内存块，但是没有用 free 释放，而且没有指针指向这块内存。这样的内存块永远不能被程序释放，因为没有指针指向它们。如果设置为 summary，Valgrind 会报告有多少内存泄漏发生了。如果设置为 full 或 yes，Valgrind 给出每一个独立的泄漏的详细信息。

- --show-reachable=<yes|no> [default: no]

当这个选项关闭时，内存泄漏检测器只显示没有指针指向的内存块，或者只能找到指向块中间的指针。当这个选项打开时，内存泄漏检测器还报告有指针指向的内存块。这些块是最有可能出现内存泄漏的地方。你的程序可能，至少在原则上，应该在退出前释放这些内存块。这些有指针指向的内存块和没有指针指向的内存块，或者只有内部指针指向的块，都可能产生内存泄漏，因为实际上没有一个指向块起始的指针可以拿来释放，即使你想去释放它。

- --leak-resolution=<low|med|high> [default: low]

在做内存泄漏检查时，确定 memcheck 将怎么样考虑不同的栈是相同的情况。当设置为 low 时，只需要前两层栈匹配就认为是相同的情况；当设置为 med，必须要四层栈匹配，当设置为 high 时，所有层次的栈都必须匹配。对于 hardcore 内存泄漏检查，你很可能需要使用 --

`--leak-resolution=high` 和 `--num-callers=40` 或者更大的数字。注意这将产生巨量的信息，这就是为什么默认选项是四个调用者匹配和低分辨率的匹配。注意 `--leak-resolution=` 设置并不影响 `memcheck` 查找内存泄漏的能力。它只是改变了结果如何输出。

- `--freelist-vol=<number>` [default: 5000000]

当客户程序使用 `free(C 中)` 或者 `delete(C++)` 释放内存时，这些内存并不是马上就可以用来再分配的。这些内存将被标记为不可访问的，并被放到一个已释放内存的队列中。这样做的目的是，使释放的内存再次被利用的点尽可能的晚。这有利于 `memcheck` 在内存块释放后这段重要的时间检查对块不合法的访问。这个选项指定了队列所能容纳的内存总容量，以字节为单位。默认的值是 5000000 字节。增大这个数目会增加 `memcheck` 使用的内存，但同时也增加了对已释放内存的非法使用的检测概率。

- `--workaround-gcc296-bugs=<yes|no>` [default: no]

当这个选项打开时，假定读写栈指针以下的一小段距离是 `gcc 2.96` 的 bug，并且不报告为错误。距离默认为 256 字节。注意 `gcc 2.96` 是一些比较老的 Linux 发行版 (RedHat 7.X) 的默认编译器，所以你可能需要使用这个选项。如果不是必要请不要使用这个选项，它可能会使一些真正的错误溜掉。一个更好的解决办法是使用较新的，修正了这个 bug 的 `gcc/g++` 版本。

- `--partial-loads-ok=<yes|no>` [default: no]

控制 `memcheck` 如何处理从地址读取时字长度，字对齐，因此哪些字节是可以寻址的，哪些是不可以寻址的。当设置为 yes 是，这样的读取并不抛出一个寻址错误。而是从非法地址读取的 V 字节显示为未定义，访问合法地址仍然是像平常一样映射到内存。设置为 no 时，从部分错误的地址读取与从完全错误的地址读取同样处理：抛出一个非法地址错误，结果的 V 字节显示为合法数据。注意这种代码行为是违背 ISO C/C++ 标准，应该被认为是有问题的。如果可能，这种代码应该修正。这个选项应该只是做为一个最后考虑的方法。

- `--undef-value-errors=<yes|no>` [default: yes]

控制 `memcheck` 是否检查未定义值的危险使用。当设为 yes 时，`Memcheck` 的行为像 `Addrcheck`，一个轻量级的内存检查工具，是 `Valgrind` 的一个部分，它并不检查未定义值的错误。使用这个选项，如果你不希望看到未定义值错误。

7.5.2 CACHEGRIND 选项：

手动指定 I1/D1/L2 缓冲配置，大小是用字节表示的。这三个必须用逗号隔开，中间没有空格，例如：`valgrind --tool=cachegrind --I1=65535,2,64` 你可以指定一个，两个或三个 I1/D1/L2 缓冲。如果没有手动指定，每个级别使用普通方式 (通过 CPUID 指令得到缓冲配置，如果失败，使用默认值) 得到的配置。

- `--I1=<size>, <associativity>, <line size>`

指定第一级指令缓冲的大小，关联度和行大小。

- `--D1=<size>, <associativity>, <line size>`

指定第一级数据缓冲的大小，关联度和行大小。

- `--L2=<size>, <associativity>, <line size>`

指定第二级缓冲的大小，关联度和行大小。

7.6 CALLGRIND 选项:

- `--heap=<yes|no>` [default: yes]

当这个选项打开时，详细的追踪堆的使用情况。关闭这个选项时，`massif.pid.txt` 或 `massif.pid.html` 将会非常的简短。

- `--heap-admin=<number>` [default: 8]

每个块使用的管理字节数。这只能使用一个平均的估计值，因为它可能变化。`glibc` 使用的分配器每块需要 4~15 字节，依赖于各方面的因素。管理已经释放的块也需要空间，尽管 `massif` 不计算这些。

- `--stacks=<yes|no>` [default: yes]

当打开时，在剖析信息中包含栈信息。多线程的程序可能有多个栈。

- `--depth=<number>` [default: 3]

详细的堆信息中调用过程的深度。增加这个值可以给出更多的信息，但是 `massif` 会更使这个程序运行得慢，使用更多的内存，并且产生一个大的 `massif.pid.txt` 或者 `massif.pid.hp` 文件。

- `--alloc-fn=<name>`

指定一个分配内存的函数。这对于使用 `malloc()` 的包装函数是有用的，可以用它来填充原来无效的上下文信息。(这些函数会给出无用的上下文信息，并在图中给出无意义的区域)。指定的函数在上下文中被忽略，例如，像对 `malloc()` 一样处理。这个选项可以在命令行中重复多次，指定多个函数。

- `--format=<text|html>` [default: text]

产生 `text` 或者 `HTML` 格式的详细堆信息，文件的后缀名使用. `txt` 或者. `html`。

7.7 HELGRIND 选项:

- `--private-stacks=<yes|no>` [default: no]

假定线程栈是私有的。

- `--show-last-access=<yes|some|no>` [default: no]

显示最后一次字访问出错的位置。

7.8 LACKEY 选项:

- `--fnname=<name>` [default: `_dl_runtime_resolve()`]

对 `<name>` 函数计数。

- `--detailed-counts=<no|yes>` [default: no]

对读取，存储和 alu 操作计数。

Author: phenix3443

Email: phenix3443+github@gmail.com

Created: 2020-04-26 日 10:53