**11-camke工具**

1. **camke**

https://mp.weixin.qq.com/s/B4jISWt7M1wBh-1WR93eew

1.1 **camke模板**

|  |
| --- |
| CMake #设置版本  cmake\_minimum\_required(VERSION 3.4.1)  project(myspdlog)  #设置目录-以此目录为基准设置下面编译的头，源，库位置、CMakeLists.txt所在目录  set(CURRENT\_DIR ${CMAKE\_SOURCE\_DIR})  #设置编译器  set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -std=gnu99")  #生成gdb的调试信息和不进行代码优化  set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -O0 -g")   set(CMAKE\_CXX\_STANDARD 11) # 添加编译选项，启用调试信息和生成调试符号 set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -g") # 添加链接选项，生成调试信息 set(CMAKE\_EXE\_LINKER\_FLAGS "${CMAKE\_EXE\_LINKER\_FLAGS} -g")   # 开源库 #include\_directories(./thirdparty/protobuf/include) #link\_directories(./thirdparty/protobuf/lib)  # 搜索指定的库文件 #find\_library(LIB\_PROTOBUF protobuf HINTS ./thirdparty/protobuf/lib)   #包含源文件 ，aux\_source\_directory将目录所有源文件自动加入  #aux\_source\_directory(${CMAKE\_SOURCE\_DIR}/src/common SOURCE\_COMMON)    # 添加头文件搜索路径 include\_directories(${CMAKE\_CURRENT\_SOURCE\_DIR}/libspdlog/) include\_directories(${CMAKE\_CURRENT\_SOURCE\_DIR}/libspdlog/include/) include\_directories(${CMAKE\_CURRENT\_SOURCE\_DIR}/libspdlog/include/adapter)    #包含源文件 ，aux\_source\_directory将目录所有源文件自动加入  aux\_source\_directory(${CMAKE\_CURRENT\_SOURCE\_DIR}/libspdlog/adapter/ SOURCE\_COMMON)  aux\_source\_directory(${CMAKE\_CURRENT\_SOURCE\_DIR}/libspdlog/ SOURCE\_API)    #生成IDPS可执行，这里直接放源文件也行如main.c；太多话，可以设置目录例如SOURCE\_COMMON。  add\_executable(myspdlog   ${CMAKE\_CURRENT\_SOURCE\_DIR}/spdlog\_test.c  ${SOURCE\_API}  ${SOURCE\_COMMON}   )    #链接我们需要的库，如lpthread系统库，预编译好的库lspdlog，需要2.5设置好路径  target\_link\_libraries( myapp  # -lspdlog  # -lpthread  # -lstdc++   )   # 设置输出路径 set\_target\_properties(myspdlog PROPERTIES  RUNTIME\_OUTPUT\_DIRECTORY ${CMAKE\_CURRENT\_SOURCE\_DIR}/../bin)  set\_target\_properties(client PROPERTIES ARCHIVE\_OUTPUT\_DIRECTORY ${CMAKE\_CURRENT\_SOURCE\_DIR}/lib)   # 设置 server 的构建产物输出目录 set\_target\_properties(server PROPERTIES LIBRARY\_OUTPUT\_DIRECTORY ${CMAKE\_CURRENT\_SOURCE\_DIR}/lib) set\_target\_properties(server PROPERTIES RUNTIME\_OUTPUT\_DIRECTORY ${CMAKE\_CURRENT\_SOURCE\_DIR}/bin)  # 设置 client 的构建产物输出目录 set\_target\_properties(client PROPERTIES LIBRARY\_OUTPUT\_DIRECTORY ${CMAKE\_CURRENT\_SOURCE\_DIR}/lib) set\_target\_properties(client PROPERTIES RUNTIME\_OUTPUT\_DIRECTORY ${CMAKE\_CURRENT\_SOURCE\_DIR}/bin)  # 将头文件复制到输出目录 configure\_file(${CMAKE\_CURRENT\_SOURCE\_DIR}/libspdlog/spdloglib.h ${CMAKE\_CURRENT\_SOURCE\_DIR}/../bin COPYONLY)  # 添加子目录，包含 gtest 的构建 add\_subdirectory(base64) add\_subdirectory(cjson) add\_subdirectory(spdlog) add\_subdirectory(sqlite3) add\_subdirectory(zlog) |

1.2 **注册安装**

|  |
| --- |
| CMake install(  TARGETS node1 node2 dagnode1 dagnode2 node\_server node\_client app\_control datarec\_demo  RUNTIME DESTINATION $ENV{INSTALL\_DIR}/bin  LIBRARY DESTINATION $ENV{INSTALL\_DIR}/lib)  install(  DIRECTORY config  DESTINATION DESTINATION $ENV{INSTALL\_DIR}) |

1.3 **生成静态库、动态库**

|  |
| --- |
| C++ # 创建名为 "mylib" 的静态库 add\_library(mylib STATIC ${SOURCE\_FILES})   # 创建名为"mylib"的动态库 add\_library(mylib SHARED ${SOURCE\_FILES})  # 指定静态库的公开头文件目录 target\_include\_directories(mylib PUBLIC ${CMAKE\_CURRENT\_SOURCE\_DIR}/include) |

1.4 **c++、c**

|  |
| --- |
| C++ # 设置当前源代码目录变量 set(CURRENT\_DIR ${CMAKE\_SOURCE\_DIR}) # 设置 C++ 编译器标志 set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -std=c++11") # 选择合适的C++标准 # 生成 GDB 调试信息和不进行代码优化 set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -O0 -g")   set(CURRENT\_DIR ${CMAKE\_SOURCE\_DIR})  #设置编译器  set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -std=gnu99")  #生成gdb的调试信息和不进行代码优化  set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -O0 -g") |

1.5 **添加其他模块下子模块**

A文件夹下cmake包含B文件夹下cmake

|  |
| --- |
| C++ add\_subdirectory(../../B B\_build) B\_build为当前目录下生成目录名称，随便起；当B下cmake是指定相对目录，B\_build不生效 |

1.6 **交叉编译**

|  |
| --- |
| CMake cmake\_minimum\_required(VERSION 3.0) project(MyProject)  set(CMAKE\_SYSTEM\_NAME Linux) set(CMAKE\_SYSTEM\_PROCESSOR arm)  # 设置编译选项 set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -g") set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -g")  # 指定交叉编译工具链位置 set(CROSS\_COMPILE\_PATH /home/lixiang/ljk\_test/my\_Tool/gcc-linaro-7.5.0-2019.12-x86\_64\_aarch64-linux-gnu) set(CMAKE\_C\_COMPILER ${CROSS\_COMPILE\_PATH}/bin/aarch64-linux-gnu-gcc) set(CMAKE\_CXX\_COMPILER ${CROSS\_COMPILE\_PATH}/bin/aarch64-linux-gnu-g++)  # 添加头文件目录 include\_directories(${CROSS\_COMPILE\_PATH}/aarch64-linux-gnu/include) #include\_directories(/path/to/your\_project/include)  # 添加库文件目录 link\_directories(${CROSS\_COMPILE\_PATH}/aarch64-linux-gnu/lib) #link\_directories(/path/to/your\_project/lib)  add\_executable(core\_test   core\_test.cpp)  # 链接你的项目所需要的库，例如：libmylib.a or -lmylib #target\_link\_libraries(MyProject mylib) |

1.7 **执行**

./cmake 生成makefile

make 编译

主要camke对大小写不敏感

1.8 **常用变量**

CMAKE\_SOURCE\_DIR：根路径

CMAKE\_CURRENT\_SOURCE\_DIR：当前路径

CMAKE\_MAJOR\_VERSION：cmake 主版本号

CMAKE\_MINOR\_VERSION：cmake 次版本号

CMAKE\_C\_FLAGS：设置 C 编译选项

CMAKE\_CXX\_FLAGS：设置 C++ 编译选项

PROJECT\_SOURCE\_DIR：工程的根目录

PROJECT\_BINARY\_DIR：运行 cmake 命令的目录

CMAKE\_CURRENT\_SOURCE\_DIR：当前CMakeLists.txt 所在路径

CMAKE\_CURRENT\_BINARY\_DIR：目标文件编译目录

EXECUTABLE\_OUTPUT\_PATH：重新定义目标二进制可执行文件的存放位置

LIBRARY\_OUTPUT\_PATH：重新定义目标链接库文件的存放位置

UNIX：如果为真，表示为UNIX-like的系统，包括AppleOSX和CygWin

WIN32：如果为真，表示为 Windows 系统，包括 CygWin

APPLE：如果为真，表示为 Apple 系统

CMAKE\_SIZEOF\_VOID\_P：表示void\*的大小（例如为4或者8），可以使用其来判断当前构建为32位还是64位

CMAKE\_CURRENT\_LIST\_DIR：表示正在处理的CMakeLists.txt文件所在目录的绝对路径

CMAKE\_ARCHIVE\_OUTPUT\_DIRECTORY：用于设置ARCHIVE目标的输出路径

CMAKE\_LIBRARY\_OUTPUT\_DIRECTORY：用于设置LIBRARY目标的输出路径

CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY：用于设置RUNTIME目标的输出路径

1.9 **常用命令**

(1) project命令：

命令语法:project(<projectname> [languageName1 languageName2 ...])

命令简述:用于指定项目的名称

使用范例:project(Main)

(2) cmake\_minimum\_required命令:

命令语法:cmake\_minimum\_requried(VERSION major[.minor[.patch)

命令简述:用于指定需要的CMake的最低版本

使用范例:cmake\_minimum\_requried(VERSION 2.8.3)

(3) aux\_source\_directory命令:

命令语法:aux\_source\_directory(<dir> <variable>)

命令简述:用于包含源文件目录，dir目录下的所有源文件的名字保存在变量variable中

使用范例：aux\_source\_directory(${CMAKE\_CURRENT\_SOURCE\_DIR}/src DIR\_SRCS)

(4) add\_executable命令:

命令语法：add\_executable(<name> [WIN32] [MACOSX\_BUNDLE][EXCLUDE\_FROM\_ALL] source1 source2 … sourceN)

命令简述:用于指定从一组源文件source1 source2 ... sourceN 编译出一个可执行文件且命名为name

使用范例:add\_executable(Main $(DIR\_SRCS))

(5) add\_library命令:

命令语法：add\_library([STATIC | SHARED | MODULE] [EXCLUDE\_FROM\_ALL] source1source2 … sourceN)

命令简述：用于指定从一组源文件 source1 source2 ... sourceN编译出一个库文件且命名为name

使用范例：add\_library(Lib $(DIR\_SRCS))

(6) add\_dependencies命令:

命令语法：add\_dependencies(target-name depend-target1 depend-target2 …)

命令简述：用于指定某个目标（可执行文件或者库文件）依赖于其他的目标。

这里的目标必须是add\_executable、add\_library、add\_custom\_target命令创建的目标

(7) add\_subdirectory命令:

命令语法：add\_subdirectory(source\_dir [binary\_dir] [EXCLUDE\_FROM\_ALL])

命令简述:用于添加一个需要进行构建的子目录

使用范例:add\_subdirectory(Lib)

(8) target\_link\_libraries命令:

命令语法：target\_link\_libraries(<target> [item1 [item2 […]]][[debug|optimized|general] ] …)

命令简述:用于指定target需要链接item1 item2 ...。这里target必须已经被创建，链接的item可以是已经存在的target（依赖关系会自动添加）

使用范例:target\_link\_libraries(Main Lib)

(9) set命令：

命令简述：用于设定变量 variable 的值为 value。如果指定了 CACHE 变量将被放入 Cache（缓存）中。

命令语法：set(<variable> <value> [[CACHE <type><docstring> [FORCE]] | PARENT\_SCOPE])

使用范例：set(ProjectName Main)

(10) unset命令：

命令语法：unset(<variable> [CACHE])

命令简述：用于移除变量 variable。如果指定了 CACHE 变量将被从 Cache 中移除。

使用范例：unset(VAR CACHE)

(11) message命令：

命令语法：message([STATUS|WARNING|AUTHOR\_WARNING|FATAL\_ERROR|SEND\_ERROR] “message todisplay”…)

命令简述:用于输出信息

使用范例:message(“Hello World”)

(12) include\_directories命令：

命令语法：include\_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 …)

命令简述：用于设定目录，这些设定的目录将被编译器用来查找 include 文件

使用范例：include\_directories(${PROJECT\_SOURCE\_DIR}/lib)

(13) find\_path命令：

命令语法：find\_path(<VAR> name1 [path1 path2 …])

命令简述：用于查找包含文件name1的路径，如果找到则将路径保存在VAR中（此路径为一个绝对路径），如果没有找到则结果为<VAR>-NOTFOUND.默认情况下，VAR会被保存在Cache中，这时候我们需要清除VAR才可以进行下一次查询（使用unset命令）

|  |
| --- |
| Bash find\_path(LUA\_INCLUDE\_PATH lua.h ${LUA\_INCLUDE\_FIND\_PATH}) if(NOT LUA\_INCLUDE\_PATH)  message(SEND\_ERROR "Header file lua.h not found") endif() |

(14) find\_library命令：

命令语法：find\_library(<VAR> name1 [path1 path2 …])

命令简述：用于查找库文件 name1 的路径，如果找到则将路径保存在 VAR 中（此路径为一个绝对路径），

如果没有找到则结果为 <VAR>-NOTFOUND。

一个类似的命令 link\_directories 已经不太建议使用了

(15) add\_definitions命令：

命令语法：add\_definitions(-DFOO -DBAR …)

命令简述：用于添加编译器命令行标志（选项），通常的情况下我们使用其来添加预处理器定义

使用范例：add\_definitions(-D\_UNICODE -DUNICODE)

(16) file命令：

命令简述：此命令提供了丰富的文件和目录的相关操作（这里仅说一下比较常用的）

使用范例：

目录的遍历

* GLOB 用于产生一个文件（目录）路径列表并保存在variable 中
* 文件路径列表中的每个文件的文件名都能匹配globbing expressions（非正则表达式，但是类似）
* 如果指定了 RELATIVE 路径，那么返回的文件路径列表中的路径为相对于 RELATIVE 的路径

|  |
| --- |
| Bash file(GLOB variable [RELATIVE path][globbing expressions]...) |

* 获取当前目录下的所有的文件（目录）的路径并保存到 ALL\_FILE\_PATH 变量中

|  |
| --- |
| Bash file(GLOB ALL\_FILE\_PATH ./\*) |

* 获取当前目录下的 .h 文件的文件名并保存到ALL\_H\_FILE 变量中
* 这里的变量CMAKE\_CURRENT\_LIST\_DIR 表示正在处理的 CMakeLists.txt 文件的所在的目录的绝对路径（2.8.3 以及以后版本才支持）

1.10 **设置选项**

1、添加宏定义到C文件

|  |
| --- |
| Bash add\_definitions(-DSERVER\_TEST) -D后面为添加宏 |

2、设置 strip

strip设置减少编译大小

|  |
| --- |
| Bash set(CMAKE\_C\_FLAGS\_RELEASE "${CMAKE\_C\_FLAGS\_RELEASE} -s") set(CMAKE\_BUILD\_TYPE "RELEASE") |

3、选择设置

|  |
| --- |
| Bash option(BUILD\_SERVER "Build test application" ON) if(BUILD\_SERVER)  #操作1 else()  #操作2 endif() |

2. **makefile**

2.1 **makefile模板**

2.2 **判断语句**

ifeq

|  |
| --- |
| Bash ifeq (变量1, 变量2)  # 条件成立时执行的操作 else  # 条件不成立时执行的操作 endif |

ifneq

|  |
| --- |
| Bash ifneq (变量1, 变量2)  # 条件成立时执行的操作 else  # 条件不成立时执行的操作 endif |

ifdef

|  |
| --- |
| Bash ifdef 变量  # 变量已定义时执行的操作 else  # 变量未定义时执行的操作 endif |

ifndef

|  |
| --- |
| Bash ifndef 变量  # 变量未定义时执行的操作 else  # 变量已定义时执行的操作 endif |

2.3 **VPATH和vpath**

（1）VPATH：VPATH是一个Makefile中的特殊变量，用于设置搜索路径。通过VPATH，可以指定Make命令在指定的目录中搜索依赖文件和目标文件。

|  |
| --- |
| Bash 例如：VPATH = dir1:dir2:dir3 |

（2）vpath：vpath是Makefile中的一个函数，用于临时设置搜索路径。与VPATH不同，vpath是一个函数调用，可以在规则中设置特定的搜索路径。

|  |
| --- |
| Bash 例如： vpath %.c src |

这样，对于所有以.c为扩展名的文件，在查找时会在src目录中搜索。

（3）总结：VPATH是一个全局的搜索路径设置，适用于整个Makefile，而vpath是局部的搜索路径设置，适用于特定的规则。在大多数情况下，推荐使用vpath来临时设置搜索路径，而不是修改全局的VPATH。

2.4 **多文件demo**

单文件源，单文件头

|  |
| --- |
| Bash project/  |- src/  | |- file1.c  | |- file2.c  |- include/  | |- header1.h  | |- header2.h  |- obj/  |- bin/ |

 makefile

|  |
| --- |
| Bash CC = gcc CFLAGS = -Wall -Wextra -g LDFLAGS = TARGET = bin/myprogram SRCDIR = src INCDIR = include OBJDIR = obj SOURCES := $(wildcard $(SRCDIR)/\*.c) OBJECTS := $(patsubst $(SRCDIR)/%.c,$(OBJDIR)/%.o,$(SOURCES)) .PHONY: all clean all: $(TARGET) $$(TARGET): $$(OBJECTS)     $(CC) $(LDFLAGS) $^ -o $@ $$(OBJDIR)/%.o: $$(SRCDIR)/%.c     $(CC) $(CFLAGS) -I$(INCDIR) -c $< -o $@ clean:     rm -f $(OBJECTS) $(TARGET) |

多文件源，多文件头

|  |
| --- |
| Bash project/  |- aes/  | |- file1.c  | |- file2.c  | |- header1.h  | |- header2.h |- hasfh/  | |- file1.c  | |- file2.c  | |- header1.h  | |- header2.h  |- obj/ |- bin/ |

makefile

|  |
| --- |
| Bash  CC = gcc CFLAGS = -Wall -Wextra -g LDFLAGS = TARGET = bin/myprogram SRCDIRS = aes hasfh INCDIRS = aes hasfh OBJDIR = obj SOURCES := $(wildcard $(addsuffix /\*.c, $(SRCDIRS))) OBJECTS := $(patsubst %.c,$(OBJDIR)/%.o,$(notdir $(SOURCES))) .PHONY: all clean all: $(TARGET) $$(TARGET): $$(OBJECTS)     $(CC) $(LDFLAGS) $^ -o $@ $$(OBJDIR)/%.o: $$(addsuffix /%.c, $(SRCDIRS))     $(CC) $(CFLAGS) -I$(addsuffix /, $(INCDIRS)) -c $< -o $@ clean:     rm -f $(OBJECTS) $(TARGET) |

多文件源、多文件头

|  |
| --- |
| Bash project/  |- aes/  | |- file1.c  | |- file2.c  | |- header1.h  | |- header2.h |- hasfh/  | |- file1.c  | |- file2.c  | |- header1.h  | |- header2.h  |- main.c |- obj/ |- bin/ |

makefile

|  |
| --- |
| Bash CC = gcc CFLAGS = -Wall -Wextra -g LDFLAGS = TARGET = bin/myprogram SRCDIRS = aes hasfh INCDIRS = aes hasfh OBJDIR = obj SOURCES := $(wildcard $(addsuffix /\*.c, $(SRCDIRS))) main.c OBJECTS := $(patsubst %.c,$(OBJDIR)/%.o,$(notdir $(SOURCES))) .PHONY: all clean all: $(TARGET) $$(TARGET): $$(OBJECTS)     $(CC) $(LDFLAGS) $^ -o $@ $$(OBJDIR)/%.o: $$(addsuffix /%.c, $(SRCDIRS)) main.c     $(CC) $(CFLAGS) -I$(addsuffix /, $(INCDIRS)) -c $< -o $@ clean:     rm -f $(OBJECTS) $(TARGET) |

3. **GCC**

GCC 使用的命令语法如下：

3.1 **输入选项**

gcc [选项] 输入的文件名

|  |  |
| --- | --- |
| 常用选项： |  |
| -o： | 小写字母“o”，指定生成的可执行文件的名字，不指定的话生成的可执行文件名为 a.out。 |
| -E： | 只进行预处理，既不编译，也不汇编。 |
| -S： | 只编译，不汇编。 |
| -c： | 编译并汇编，但不进行链接。 |
| -g： | 生成的可执行文件带调试信息，方便使用 gdb 进行调试。 |
| -Ox： | 大写字母“O”加数字，设置程序的优化等级，如“-O0”“-O1”“-O2”“-O3”，数字越大代码的优化等级越高，编译出来的程序一般会越小，但有可能会导致程序不正常运行。 |

3.2 **gcc 符号解释**

|  |  |
| --- | --- |
| $@ | --代表目标文件(target) |
| $^ | --代表所有的依赖文件(components) |
| $< | --代表第一个依赖文件(components中最左边的那个)。 |
| $? | --代表当前目标所依赖的文件列表中比当前目标文件还要新的文件。 |
| $\* | --不包括后缀名的当前依赖文件的名字 |

3.3 **编译路径指定**

|  |  |
| --- | --- |
| -I ( i 的大写)： | 指定头文件路径（相对路径或绝对路径，建议相对路径） |
| -i ： | 指定头文件名字 (一般不使用，而是直接放在\*\*.c 文件中通过#include<\*\*\*.h> 添加) |
| -L ： | 指定连接的动态库或者静态库路径（相对路径或绝对路径，建议相对路径） |
| -l (L的小写)： | 指定需要链接的库的名字（链接 libc.a : -lc 链接动态库：libc.so : -lc）。 注意：-l后面可以直接添加库名省去“lib”和“.so”或“.a”。 -l(L的小写)链接的到底是动态库还是静态库，如果链接路径下同时有 .so 和 .a 那优先链接 .so 。 |

3.4 **动态库、静态库编译**

|  |  |
| --- | --- |
| 编译动态库 | gcc -shared -o libexample.so example.c |
| 编译静态库 | gcc -c example.c  ar rcs libexample.a example.o |
| 使用动态库编译 | libabc.so(下面的编译命令不带lib和so,只添加了-l)  gcc main.c -o main -L （动态库目录） -labc |

3.5 **运行设置库路径**

|  |
| --- |
| Bash export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/the/path/you/want/set |

3.6 **查看编译文件库依赖**

|  |
| --- |
| Bash lld xxx.so 查看库依赖 |

4. **GDB工具**

4.1 **命令**

B (break) 断点当前位置, B+line 断点需要的line行

R (run)开始执行

C 执行到下个断点

N (next)单步执行

S 跳入执行

P（print） 查看变量

1. 查看某线称崩溃

|  |
| --- |
| C++ info threads |

1. 切换到线程6：

|  |
| --- |
| C++ thread 6 |

1. 查看当前线程的堆栈跟踪信息：

|  |
| --- |
| C++ bt |

4.2 **GDB多线程调试**

shell的命令：

(1)查看当前运行的进程：ps aux | grep book

(2)查看当前运行的轻量级进程：ps -aL | grep book

(3)查看主线程和子线程的关系：pstree -p 主线程id

gdb的命令：

(1)查看可切换调试的线程：info threads

(2)切换调试的线程：thread 线程id

(3)只运行当前线程：set scheduler-locking on

(4)运行全部的线程：set scheduler-locking off

(5)指定某线程执行某gdb命令：thread apply 线程id gdb\_cmd

Eg:thread apply 3 n 只让线程3执行next

(6)全部的线程执行某gdb命令：thread apply all gdb\_cmd

4.3 **GDB内存查询**

1、内存查看

x/3uh 0x54320:从地址0x54320开始，读取3个双字节（h），以十六进制方式显示（x）

3可以替换成任意正整数

u可以替换成：

d 按十进制格式显示变量 x 按十六进制格式显示变量 a 按十六进制格式显示变量 u 按十六进制格式显示无符号整型 o 按八进制格式显示变量 t 按二进制格式显示变量 c 按字符格式显示变量 f 按浮点数格式显示变量

h可以替换成：

b表示单字节，h表示双字节，w表示四字 节，g表示八字节

5. **valgrind 工具**

在树莓派上用 valgrind 这个工具可以查查有没有没释放的内存

6. **core dump工具**

1. 查看限制

|  |
| --- |
| C++ ulimit -c |

 如果显示 1024 那么说明 coredump 已经被开启。

1024 限制产生的 core 文件的大小不能超过 1024kb，可以使用参数unlimited，取消该限制

|  |
| --- |
| C++ ulimit -c unlimited |

1. 查询core dump文件路径：

|  |
| --- |
| C++ 方法1： cat /proc/sys/kernel/core\_pattern 方法2： /sbin/sysctl kernel.core\_pattern |

1. 修改core dump文件路径：

方法1：临时修改：修改/proc/sys/kernel/core\_pattern文件，但/proc目录本身是动态加载的，每次系统重启都会重新加载，因此这种方法只能作为临时修改。

/proc/sys/kernel/core\_pattern

|  |
| --- |
| C++ 例：echo ‘/var/log/%e.core.%p’ > /proc/sys/kernel/core\_pattern |

方法2：永久修改：使用sysctl -w name=value命令。

|  |
| --- |
| C++ 例：/sbin/sysctl -w kernel.core\_pattern=/var/log/%e.core.% |

 gdb+core文件