这是道哥的第008篇原创

一、前言

上篇文章我们聊了gdb的底层调试机制，明白了gdb是利用操作系统提供的系统信号来调试目标程序的。很多朋友私下留言了，看到能帮助到大家，我心里还是很开心的，其实这也是我继续输出文章的最大动力！后面我会继续把自己在项目开发中的实战经验进行总结。

由于gdb的代码相对复杂，没有办法从代码层面仔细的分析调试细节，所以这次我们选择一个小巧、开源的Lua脚本语言，深入到最底层的代码中去探究一下代码调试真正是怎么一回事。

不过请放心，虽然深入到代码最底层，但是理解难度并不大，只要C语言掌握的没问题，其他就都不是问题。
另外，这篇文章重点不是介绍代码，而是介绍实现一个调试器应该如何思考，解决问题的思路是什么。

通过阅读这篇文章，能有什么收获？

1. 如果你使用过Lua语言，那么你能够从源代码级别了解到调试库的代码逻辑。
2. 如果你对Lua不了解，可以从设计思想、实现架构上学习到一门编程语言是如何进行调试程序的。

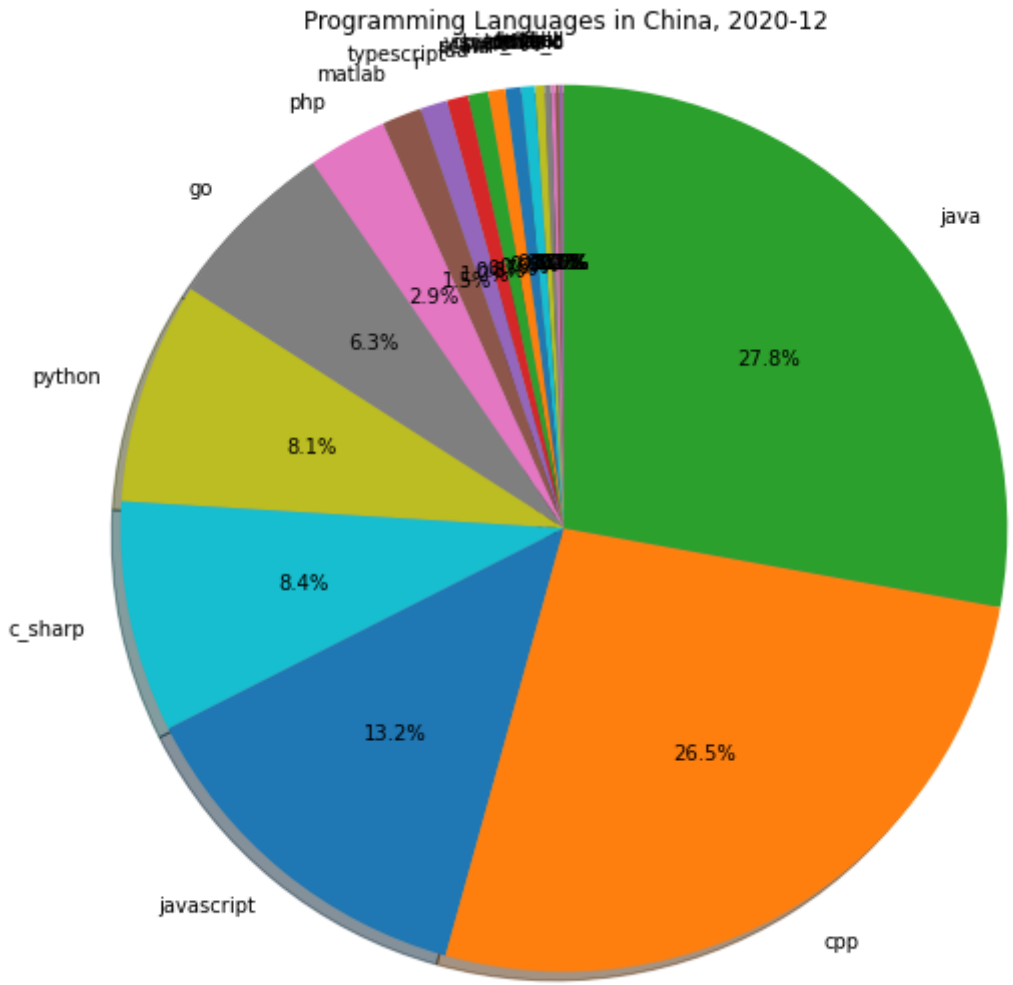
二、Lua 语言简介

1. Lua是什么鬼?

喜欢玩游戏的小伙伴可能会知道，Lua语言在游戏开发中使用的比较多。它是一个轻量、小巧的脚本语言，用标准C语言编写，源码开放。正因为这几个原因，所以我才选择它作为剖析对象。

如果对于Lua语言还是没有感觉，Python语言总应该知道吧？广告满天飞，你就把Lua想象为类似Python一样的脚本语言，只不过体积比Python要轻量的得多。

这里有1张图可以了解下，2020年12月份的编程语言市场占有率。



在上图中看不到Lua的身影，因为市场占有率太低了，大概是位于30几名。但是再看看下面这张图，从工资的角度再体会一下Lua的高贵：

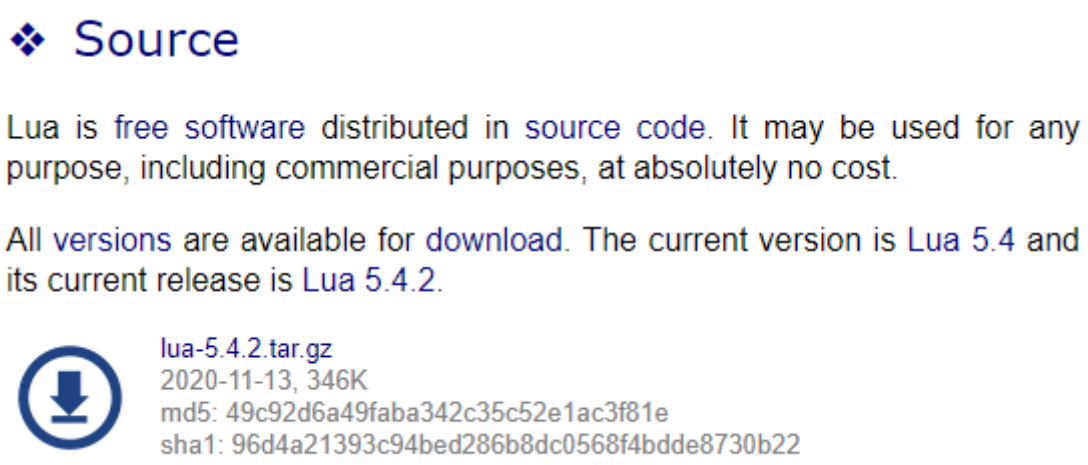
排名	编程语言	平均工资	工资中位数	最低工资(2.5%)	最高工资(97.5%)	招聘人数	百分比
1	haskell	25167	22500	20000	33000	3	0.0%
2	rust	19788	18500	5250	45000	665	0.1%
3	julia	18853	20416	9000	40000	34	0.0%
4	scala	18740	17500	6500	43006	3419	0.6%
5	go	17395	15000	5250	45000	34061	6.3%
6	lua	17048	16000	4900	41549	3942	0.7%
7	python	17024	15000	5000	45000	43539	8.0%
8	perl	16730	15500	5110	37567	2684	0.5%
9	matlab	16721	15000	5249	41548	7996	1.5%
10	swift	16583	15000	5959	40288	3085	0.6%
11	kotlin	16172	15000	4936	38547	1872	0.3%
12	ruby	15736	12500	5250	39477	1230	0.2%
13	r	15037	13500	4500	40000	5306	1.0%
14	c/c++	14905	12500	5250	40000	144142	26.5%
15	typescript	14317	12500	5250	37005	4218	0.8%
16	java	14102	12500	5250	37500	151560	27.9%
17	php	13621	12500	5000	40000	15620	2.9%
18	objective_c	13478	12500	5212	30000	151	0.0%
19	javascript	12591	12500	5000	30000	72562	13.3%
20	c#	12423	11500	5000	35000	46050	8.5%

远远的把C/C++、JAVA甩在了身后，是不是有点冲动想学一下Lua语言了？先别激动，学习任何东西，先要想明白可以用在什么地方。如果仅仅是从找工作的角度来，Lua可以不用考虑了，毕竟市场需求量比较小。

2. 为什么选择Lua语言作为研究对象?

虽然Lua语言在招聘网站中处于小众需求，但是这并不妨碍我们利用Lua来深入的学习、研究一门编程语言，Lua语言虽小，但是五脏俱全。就像我们如果想学习Linux内核的设计思想，你是愿意从最开始的版本(几千行代码)开始呢？还是愿意从当前最新的内核代码(2780万行代码，66492个文件)开始呢?

看一下当前最新版的Lua代码体积:



同样的思路，如果我们想深入研究一门编程语言，选择哪一种语言，对于我们的积极性和学习效率是非常重要的。每个人的职业生涯都很长，花一些时间沉下心来研究透一门语言，对于一个开发者来说，还是蛮有成就的，对于职业的发展是非常有好处的，你会有一览众山小的感觉!

再看一下Lua代码量与Python代码量的对比：

语言	行数
Python所有C源码	54万行
Python核心C源码不含Module	约17万行
Lua5.3所有C源码	2.4万行

从功能上来说，Lua与Python之间是没有可比性的，但是我们的目的不是学习一个编程工具，而是研究一门编程语言本身，因此选择Lua脚本语言进行学习、研究，没有错！

言归正传。

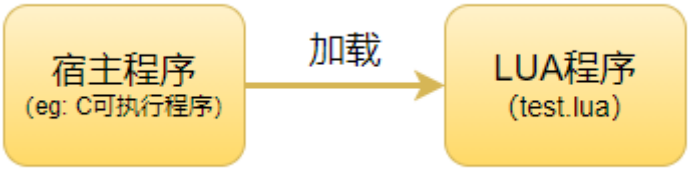
三、Lua源代码5.3.5

1. Lua程序是如何执行的?

Lua 是一门扩展式程序设计语言，被设计成支持通用过程式编程，并有相关数据描述设施。同时对面向对象编程、函数式编程和数据驱动式编程也提供了良好的支持。它作为一个强大、轻量的嵌入式脚本语言，可供任何需要的程序使用。

作为一门扩展式语言，Lua没有"main"程序的概念：它只能嵌入一个宿主程序中工作，该宿主程序被称为被嵌入程序或者简称宿主。宿主程序可以调用函数执行一小段Lua代码，可以读写Lua变量，可以注册C函数让Lua代码调用。依靠C函数，Lua可以共享相同的语法框架来定制编程语言，从而适用不同的领域。

也就是说，我们写了一个test.lua程序，是没有办法直接运行它的。而实需要一个“宿主”程序，来加载test.lua文件。



宿主程序可以是一个最简单的C程序，Lua官方提供了一个宿主程序。

我们也可以自己写一个，如下：

```
// 引入Lua头文件
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

int main(int argc, char *argv[])
{
    // 创建一个Lua虚拟机
    lua_State *L = luaL_newstate();

    // 打开LUA中的标准库
    luaL_openlibs(L);
```

```
    // 加载 test.lua 程序
    if (luaL_loadfile(L, "test.lua") || lua_pcall(L, 0, 0, 0))
    {
        printf("Error: %s \n", lua_tostring(g_lua_handle.L, -1));
        lua_close(g_lua_handle.L);
    }
    // 其他代码
}
```

2. Lua语法

在语法层面，Lua涵盖的内容还是比较全面的，它是一门动态类型语言，基本概念包括：八种基本数据类型，表是唯一的数据结构，环境与全局变量，元表及元方法，协程，闭包，错误处理，垃圾收集。具体的信息可以看一下Lua5.3参考手册。

这篇文章主要从调试器这个角度进行分析，因此我不会在这里详细的贴出很多代码细节，而只是把与调试有关的代码贴出来进行解释。

我之前在学习Lua源码时(5.3.5版本)，在代码文件中记录了很多注释，可以很好的帮助理解，主要是因为我的忘性比较好。

其实我更建议大家自己去下载源码学习，经过自己的理解、加工，印象会更深刻。在之前的工作中，由于项目需要，我对源码进行了一些优化，这部分代码就不放出来了，添加注释的源码是完完全全的Lua5.3.5版本，大概是这个样子：

```
75: /*
76: 在 Lua中短字符串是被内化的 ， 什么是内化 ？
77: 简单来说 ，  每个存放Lua字符串的变量 ， 实际上存放的只是字符串数据的引用，
78: lua中有一个全局的地方存放着当前系统中所有字符串， 每当创建一个新的字符串，
79: 首先去查找是否已经存在同样的字符串。有的话，直接将引用指向已经存在的
80: 字符串数据 ， 否则在系统内创建一个新的字符串数据，复制引用 。
81:
82: hash:  指向哈希桶的指针数组，数组中的每个元素都是一个 TString * 指针。
83: nuse:  当前哈希桶内的字符串数量
84: size : 当前哈希桶的字符串容量
85: */
86: typedef struct stringtable {
87:   TString **hash;
88:   int nuse;  /* number of elements */
89:   int size;
90: } stringtable;
```

```
Lua 把调用栈和数据栈分开保存。调用栈放在 CallInfo 结构中，
以双向链表的形式存储在 lua_State 对象里。

CallInfo 保存着正在调用的函数的运行状态。
CallInfo 是一个标准的双向链表结构，不直接被GC 管理。
这个链表表达的是一个逻辑上的栈，在运行过程中，并不是每次调用更深层次的函数
就立刻构造出一个 CallInfo 节点 。整个 CallInfo 链表会在运行中反复复用。
直到 GC 的时候才清理那些比当前调用层次更深的无用节点。
*/
typedef struct CallInfo {
  StkId func;    /* 指向正在执行的函数在数据栈上的位置,会初始化为 L1->top.
  StkId top;     /* 初始化为: L1->top + LUA_MINSTACK。 top for this functic
  struct CallInfo *previous, *next;  /* dynamic call link */
  union {
    struct {                    /* only for Lua functions */
      StkId base;               /* 调用此闭包时的第一个参数所在的数据栈位
      const Instruction *savedpc; // 指令指针，指向 Proto 中的 code 指令数
    } l;
    struct {                    /* only for C functions */
      lua_KFunction k;          /* continuation in case of yields */
      ptrdiff_t old_errfunc;
      lua_KContext ctx;         /* context info. in case of yields */
    } c;
  } u;
  ptrdiff_t extra;
  short nresults;               /* expected number of results from this fu
  unsigned short callstatus;    /* 状态标识 */
} CallInfo;
```


如果有小伙伴需要加了注释的源码，请在公众号(IOT物联网小镇)里留言给我。

四、Lua调试库相关

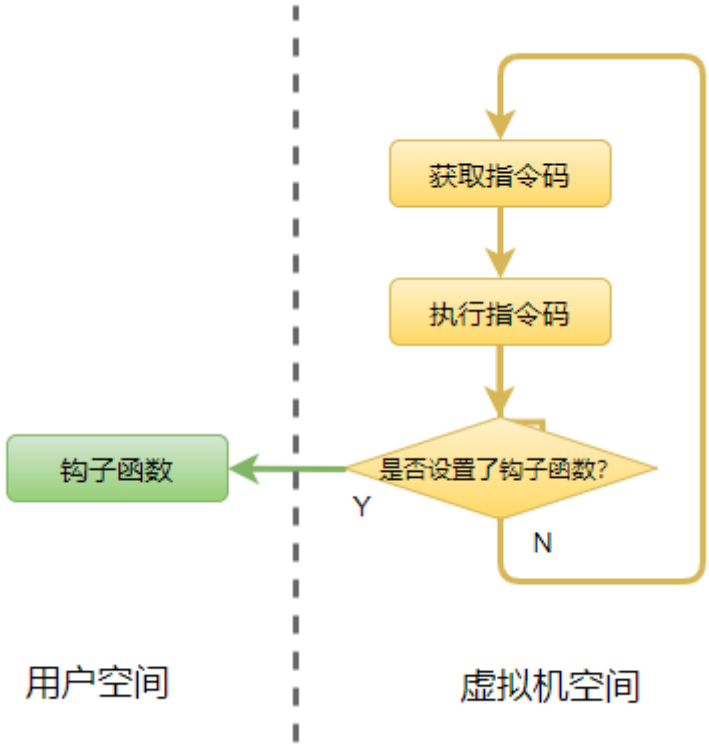
我们可以停下来稍微想一下，对一个程序进行调试，需要考虑的问题有3点：

> 1. 如何让程序暂停执行?
> 2. 如何获取程序的内部信息?
> 3. 如果修改程序的内部信息?

带着这些问题，我们来逐个击破。

1. 钩子函数(Hook)：让程序暂停执行

Lua虚拟机(也可称之为解释器)内部提供了一个接口：用户可以在应用程序中设置一个钩子函数(Hook)，虚拟机在执行指令码的时候会检查用户是否设置了钩子函数，如果设置了，就调用这个钩子函数。本质上就是设置一个回调函数，因为都是用C语言来实现的，虚拟机中只要把这个钩子函数的地址记住，然后在某些场合回调这个函数就可以了。



那么，虚拟机在哪些场合回调用户设置的钩子函数呢?

我们在设置Hook函数的时候，可以通过mask参数来设置回调策略，也就是告诉虚拟机：在什么时候来回调钩子函数。mask参数可以是下列选项的组合操作：

> 1. LUA_MASKCALL：调用一个函数时，就调用一次钩子函数。
> 2. LUA_MASKRET：从一个函数中返回时，就调用一次钩子函数。
> 3. LUA_MASKLINE：执行一行指令时，就回调一次钩子函数。
> 4. LUA_MASKCOUNT：执行指定数量的指令时，就回调一次钩子函数。

设置钩子函数的基础API原型如下：

> void lua_sethook (lua_State *L, lua_Hook f, int mask, int count);

第二个参数f需要指向我们自己定义的钩子函数，这个钩子函数原型为：

> typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);

我们也可以通过下面即将介绍的调试库中的函数来设置钩子函数，效果是一样的，因为调试库函数的内部也是调用基础函数。

> debug.sethook ([thread,] hook, mask [, count])

再来看一下虚拟机中的相关代码。
当执行完上一条指令，获取下一条指令之后，调用函数*luaG_traceexec(lua_State *L)*：

```
void luaG_traceexec (lua_State *L) {
  // 获取mask掩码
  lu_byte mask = L->hookmask;
  int counthook = (--L->hookcount == 0 && (mask & LUA_MASKCOUNT));
  if (counthook)
    resethookcount(L);
  else if (!(mask & LUA_MASKLINE))
    return;
```

```
  if (counthook)
    luaD_hook(L, LUA_HOOKCOUNT, -1);  // 按指令次数调用钩子函数
  if (mask & LUA_MASKLINE) {
    Proto *p = ci_func(ci)->p;
    int npc = pcRel(ci->u.l.savedpc, p);
    int newline = getfuncline(p, npc);
    if (npc == 0 ||
        ci->u.l.savedpc <= L->oldpc ||
        newline != getfuncline(p, pcRel(L->oldpc, p)))
      luaD_hook(L, LUA_HOOKLINE, newline); // 按行调用钩子函数
  }
}
```


可以看到，当mask掩码中包含了LUA_MASKLINE时，就调用函数luaD_hook()，如下代码：

```
void luaD_hook (lua_State *L, int event, int line) {
  lua_Hook hook = L->hook;
  if (hook && L->allowhook) {
    // 为钩子函数准备参数，其中包括了各种调试信息
    lua_Debug ar;
    ar.event = event;
    ar.currentline = line;
    ar.i_ci = ci;
    // 调用钩子函数
    (*hook)(L, &ar);
  }
}
```

只要进入了用户设置的钩子函数，那么我们就可以在这个函数中为所欲为了。

比如：获取程序内部信息，读取、修改变量的值，查看函数调用栈信息等等，这就是下面要讲解的内容。

2. Lua调试库是什么?

首先说一下Lua中的标准库。
所谓的标准库就是Lua为开发者提供的一些有用的函数，可以提高开发效率，当然我们可以选择不使用标准库，或者只使用部分标准库，这是可以裁剪的。

- 基础库
- 协程库
- 包管理库
- 字符串控制
- 基础UTF-8支持
- 表控制
- 数学函数
- 输入输出
- 操作系统库
- 调试库

这里我们只介绍一下基础库、操作系统库和调试库这3个家伙。

基础库

基础库提供了Lua核心函数，如果你不将这个库包含在你的程序中，就需要小心检查程序是否需要自己提供其中一些特性的实现，这个库一般都是需要使用的。

操作系统库

这个库提供与操作系统进行交互的功能，例如提供了函数：

> os.date
> os.time
> os.execute
> os.exit
> os.getenv

调试库

先看一下库中提供的几个重要的函数：

> debug.gethook
> debug.sethook
> debug.getinfo
> debug.getlocal
> debug.setlocal
> debug.setupvalue
> debug.traceback
> debug.getregistry

上面已经说到，Lua给用户提供了设置钩子的API函数lua_sethook，用户可以直接调用这个函数，此时传入的钩子函数的定义格式需要满足要求。

为了简化用户编程，Lua还提供了调试库来帮助用户降低编程难度。调试库其实也就是把基础API函数进行封装了一下，我们以设置钩子函数debug.sethook为例：
文件ldblib.c中，定义了调试库支持的所有函数：

```
static int db_sethook (lua_State *L) {
  lua_sethook(L1, func, mask, count);
}

static const luaL_Reg dblib[] = {
  // 其他接口函数都删掉了，只保留这一个来讲解
  {"sethook", db_sethook},
  {NULL, NULL}
};

// 这个函数用来把调试库中的函数注册到全局变量表中
LUAMOD_API int luaopen_debug (lua_State *L) {
  luaL_newlib(L, dblib);
  return 1;
}
```

可以看到，调试库的debgu.sethook()函数最终也是调用基础API函数:lua_sethook()。

在后面的调试器开发讲解中，我就是用debug库来实现一个远程调试器。

3. 获取程序内部信息

在钩子函数中，可以通过如下API函数还获取程序内部的信息了：

> int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);

在这个API函数中：

> 第二个参数用来告诉虚拟机我们想获取程序的哪些信息
> 第三个参数用来存储获取到的信息

结构体lua_Debug比较重要，成员变量如下：

```
typedef struct lua_Debug {
  int event;
  const char *name;           /* (n) */
  const char *namewhat;       /* (n) */
  const char *what;           /* (S) */
  const char *source;         /* (S) */
  int currentline;            /* (l) */
  int linedefined;            /* (S) */
  int lastlinedefined;        /* (S) */
  unsigned char nups;         /* (u) 上值的数量 */
  unsigned char nparams;      /* (u) 参数的数量 */
  char isvararg;              /* (u) */
  char istailcall;            /* (t) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* 私有部分 */
  其它域
} lua_Debug;
```

> 1. source：创建这个函数的代码块的名字。 如果 source 以 '@' 打头， 指这个函数定义在一个文件中，而 '@' 之后的部分就是文件名。
> 2. linedefined: 函数定义开始处的行号。
> 3. lastlinedefined: 函数定义结束处的行号。
> 4. currentline: 给定函数正在执行的那一行。

其他字段可以在参考手册中查询。
例如：如果想知道函数 f 是在哪一行定义的， 你可以使用下列代码：

```
lua_Debug ar;
lua_getglobal(L, "f");  /* 取得全局变量 'f' */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

同样的，也可以调用调试库debug.getinfo()来达到同样的目的。

4. 修改程序内部信息

经过上面的讲解，已经看到我们获取程序信息都是通过Lua提供的API函数，或者是利用调试库提供的接口函数来完成的。那么修改程序内部信息也同样如此。
Lua提供了下面这2个API函数来修改函数中的变量：

1. 修改当前活动记录总的局部变量的值：

> const char *lua_setlocal (lua_State *L, const lua_Debug *ar, int n);

2. 设置闭包上值的值(上值upvalue就是闭包使用了外层的那些变量)

> const char *lua_setupvalue (lua_State *L, int funcindex, int n);

同样的，也可以利用调试库中的debug.setlocal和debug.setupvalue来完成同样的功能。

5. 小结

到这里，我们就把Lua语言中与调试有关的机制和代码都理解清楚了，剩下的问题就是如何利用它提供的这些接口，来编写一个类似gdb一样的调试器。

就好比：Lua已经把材料(米、面、菜、肉、佐料)摆在我们的面前了，剩下的就需要我们把这些材料做成一桌美味佳肴。

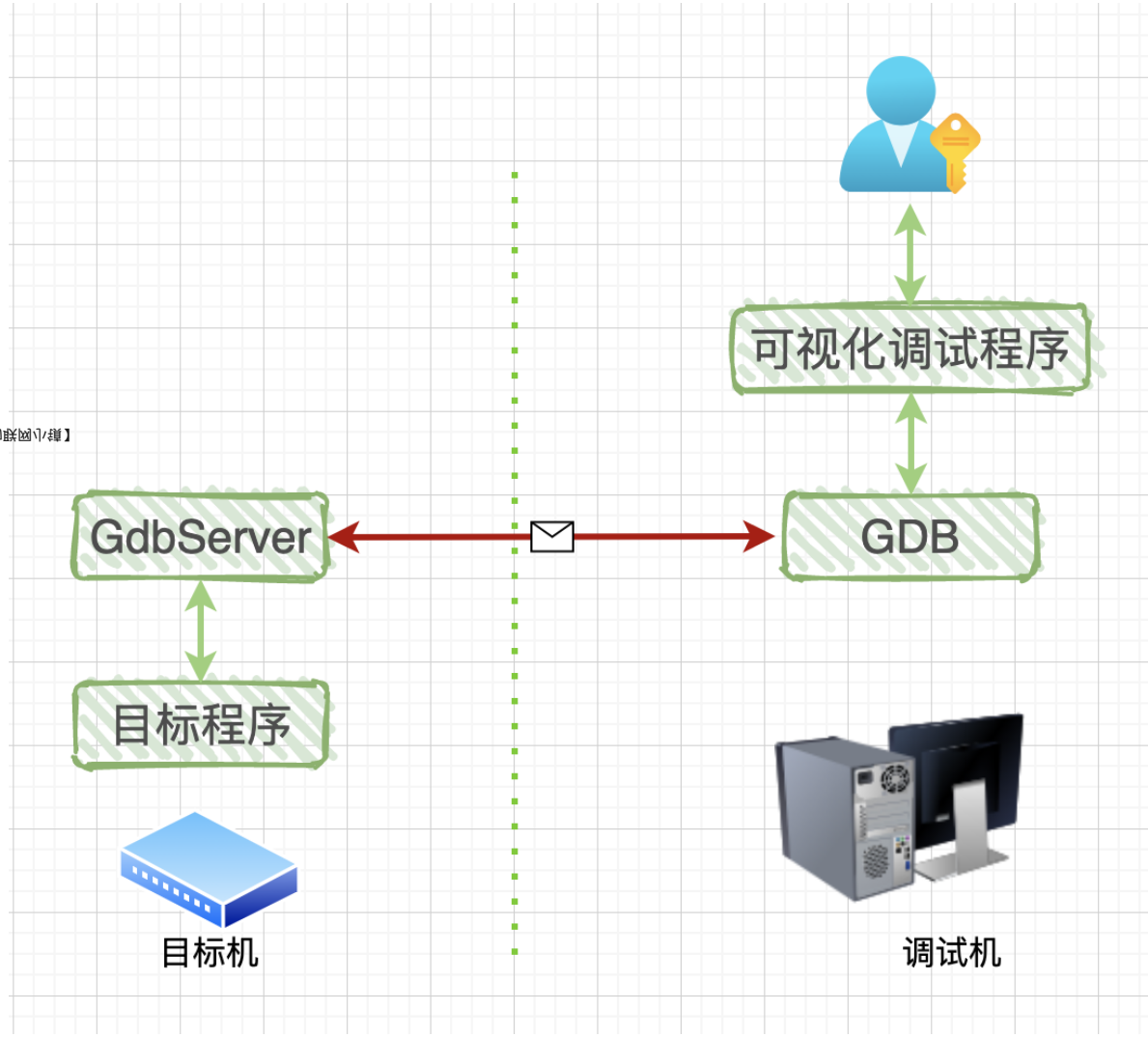
五、Lua调试器开发

1. 与gdb调试模型做类比

上一篇文章说过，gdb调试模型有两种：本地调试和远程调试。

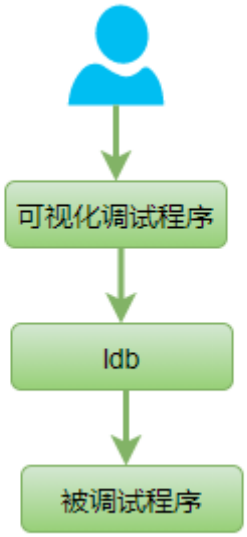
本地调试

远程调试

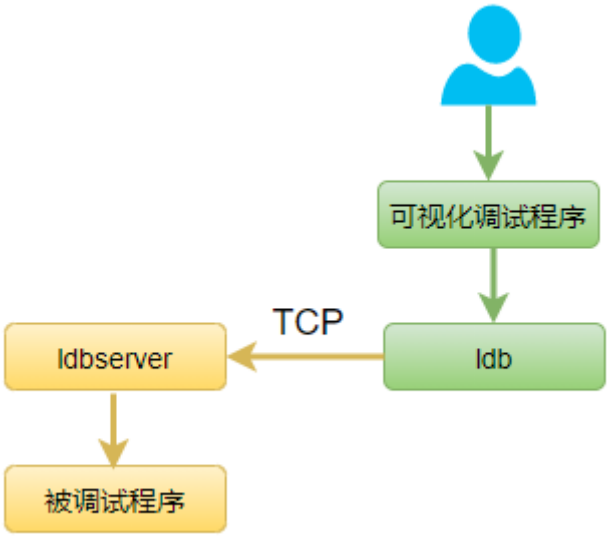


那么，我们也可以按照这个思路来实现两种调试模型，只要把其中的gdb替换成ldb，gdbserver替换成ldbserver即可。

本地调试



远程调试



这两种调试模型本质是一样的，只是调试程序和被调试程序是否运行在同一台电脑上而已。

如果是远程调试，ldbserver调用接口函数对被调试程序进行控制，然后把结果通过TCP网络传递给ldb，ldbserver就相当于一个传话筒。

至于选择实现哪一种调试模型？这个要根据实际场景的需求来决定。
我在这里实现的是远程调试，因为被调试程序是需要运行在ARM板子(下位机)中的，但是调试器是需要运行在PC电脑上(上位机)的，通过远程调试，只需要把ldbserver和被调试程序放到下位机中运行，ldb嵌入到上位机的集成开发环境(IDE)中运行就可以了。

另外，远程调试模型同样也可以全部运行在同一台PC电脑中，这个时候ldb与ldbserver之间就是在本机中进行TCP网络连接。

这里有2个内容需要补充一下：

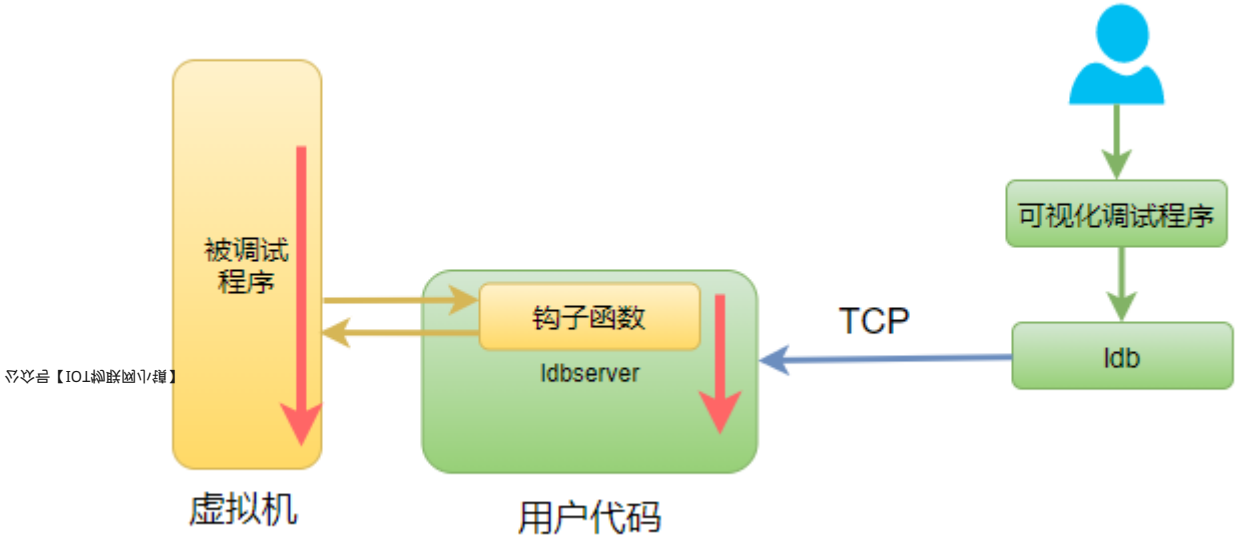
> 1. TCP链接可以直接利用第三方库luasocket。
> 2. ldb与ldbserver之间的通讯协议可以参照gdb与gdbserver之间的协议，也可以自定义。我借鉴了HTTP协议，简化了很多。

2. ldbserver如何实现

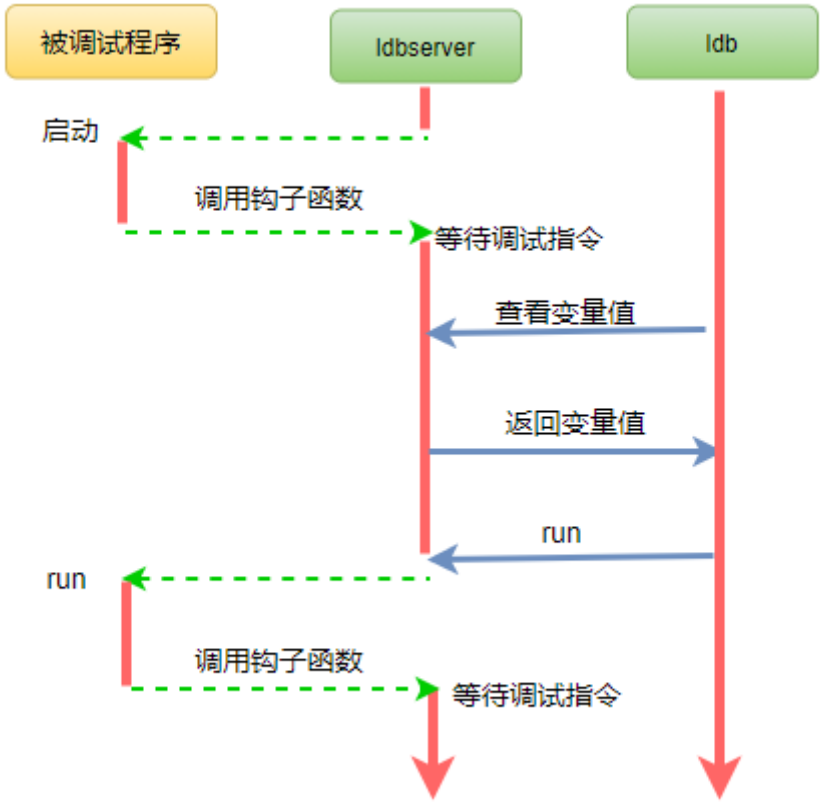
思考一个问题：被调试程序在执行时调用钩子函数，在钩子函数中我们可以做各种调试操作，但是在执行到钩子函数的最后，是需要返回到被调试程序中的下一行指令码继续执行的，我们不能打断被调试程序的执行序列。

但是，调试操作又需要通过TCP连接与上位机进行通信协议的交互，比如：设置断点、查看变量的值、查看函数信息等等。所以，被调试程序的执行与调试器ldbserver的执行是2个并发的执行序列，可以理解为2个线程在并发执行。我们需要在这2个执行序列之间进行协调，比如：

> 1. ldbserver在等待用户输入指令时(running)，被调试程序应该处于暂停状态(pending)。
> 2. ldbserver接收到用户指令后(eg: run)，自己应该暂停执行(pending)，让被调试程序继续执行(running)。



上图中，两条红色箭头表示两个执行序列。这两个执行序列并不是同时在执行的，而是交替执行，如下图所示：



那么怎么样才能让这2个执行序列交替执行呢?

如果是在C语言中，我们可以通过信号量、互斥锁等各种方法实现，但这是在Lua语言中，应该利用什么机制来实现这个功能?

柳暗花明又一村!

Lua中提供了协程机制!
下面这段话是从参考手册中摘抄过来：

> 1. Lua 支持协程，也叫协同式多线程。一个协程在 Lua 中代表了一段独立的执行线程。然而，与多线程系统中的线程的区别在于， 协程仅在显式调用一个让出（yield）函数时才挂起当前的执行。
> 2. 调用函数coroutine.create可创建一个协程。
> 3. 调用coroutine.resume函数执行一个协程。
> 4. 通过调用coroutine.yield使协程暂停执行，让出执行权。

我们可以让ldbserver运行在一个协程中，被调试程序运行在主程序中。
当虚拟机执行一条被调试程序的指令码之后，调用钩子函数，在钩子函数中通过coroutine.resume让协程运行，主程序停止。前面说到，ldbserver运行在运行在一个协程中，此时就可以在ldbserver中利用阻塞函数(例如：TCP 中的receive)，接收用户的调试指令。

假设用户发送来全速执行指令(run)，ldbserver就调用coroutine.yield让自己挂起，此时被调试程序所在的主程序就可以继续执行了。

进行到这里，基本上大功告成！剩下的就是一些代码细节问题了。

3. ldb如何实现

这部分就比较简单了，从功能上来说包括3部分内容：

> 1. 与ldbserver之间建立TCP连接。
> 2. 读取调试人员输入的指令，发送给ldbserver。
> 3. 接收ldbserver发来的信息，显示给调试人员。

可以在调试终端中手动输入、显示调试信息，也可以把ldb嵌入到一个可视化的编辑工具中，例如：

```
local function print_commands()
    print("setb <file> <line>    -- sets a breakpoin")
    print("step                  -- run one line, stepping into function")
    print("next                  -- run one line, stepping over function")
    print("goto <line>           -- goto line in a function")
    // 其他指令
end
```

六、调试指令举例

1. break指令的实现

（1）设置钩子函数

ldbserver通过调试库的debug.sethook函数，设置了一个钩子函数，调用参数是：

> debug.sethook(my_hook, "lcr")

第二个参数"lcr"的含义是：

> 'c': 每当 Lua 调用一个函数时，调用钩子。
> 'r': 每当 Lua 从一个函数内返回时，调用钩子。
> 'l': 每当 Lua 进入新的一行时，调用钩子。

也即是说：虚拟机进入一个函数、从一个函数返回、每执行一行代码，都调用一次钩子函数。注意：这里的一行指定是被调试程序中的一行Lua代码，而不是二进制文件中的一行指令码，一行Lua代码可能被会编译生成多行指令码。

这里还有一点需要注意：钩子函数虽然是定义在用户代码中，但是它是被虚拟机调用的，也就是说钩子函数是处于主程序的执行序列中。

（2）设置断点

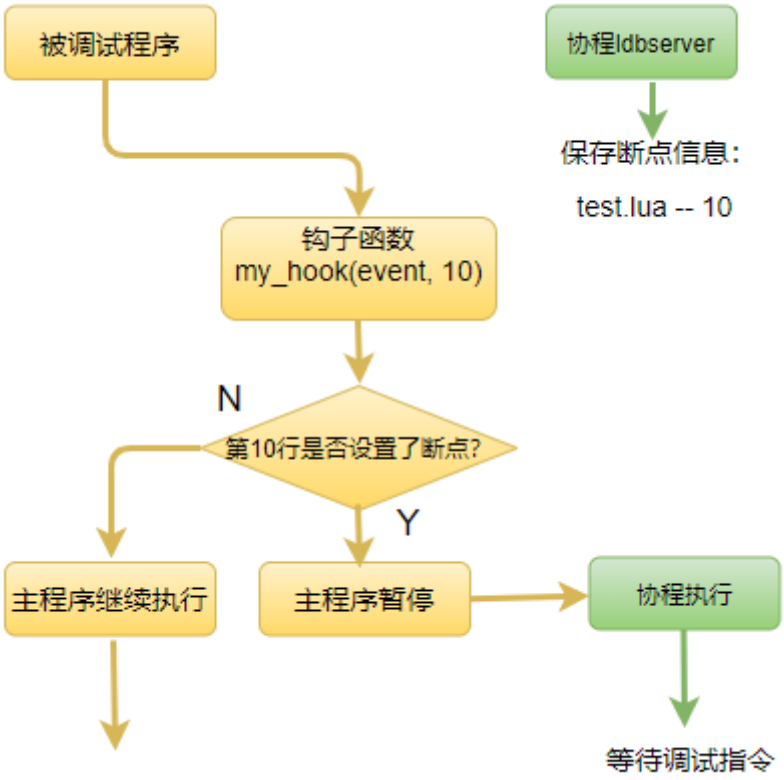
ldb向ldbserver发送设置断点的指令：setb test.lua 10，即：在test.lua文件的第10行设置一个断点，ldbserver接收到指令后，在内存中记录这个信息(文件名-行号)。

（3）捕获断点

虚拟机在调用钩子函数时，传入两个参数(注意：钩子函数是被虚拟机调用的，所以它是处于主程序的执行序列中)，

> local function my_hook(event, line)

在钩子函数中，查找这个line是否被用户设置为断点，如果是那么就通过coroutine.resume让主程序暂停，让协程中的ldbserver执行。此时，ldbserver就可以在TCP网络上继续等待ldb发来的下一个调试指令。



2. next指令的实现

next指令与step指令类似，区别在于当下一条指令是一个函数调用时:

> step指令: 进入到函数内部。
> next指令: 不进入函数内部，而是直接把这个函数执行完。

next指令的实现主要依赖于钩子函数的第一个参数event，上面在设置钩子函数的时候，告诉虚拟机在3种条件下调用钩子函数，重新贴一下：

> 'c': 每当 Lua 调用一个函数时，调用钩子
> 'r': 每当 Lua 从一个函数内返回时，调用钩子
> 'l': 每当 Lua 进入新的一行时，调用钩子

在进入钩子函数之后，event参数会告诉我们：为什么会调用钩子函数。代码如下：

```
function my_hook(event, line)
    if event == "call" then
        // 进入了一个函数
        func_level = func_level + 1
    elseif event == "return" then
        // 从一个函数返回
        func_level = func_level - 1
    else
        // 执行完一行代码
    end
```

所以就可以利用event参数来记录进入、退出函数层数，然后在钩子函数中判断：是否需要暂停主程序，把执行的机会让给协程。

3. goto指令的实现

在调试过程中，如果我们想跳过当前执行函数中的某几行，可以发送goto指令,被调试程序就从当前停止的位置直接跳转到goto指令中设置的那行代码。

目前goto指令有一个限制:

因为Lua虚拟机中的所有代码都是以函数为单位的，通过函数调用栈把所有的代码串接在一起，因此只能goto到当前函数内的指定行。

这部分功能Lua源码中并没有提供，需要扩展调试库的功能。核心步骤就是：强制把虚拟机中的PC指针设置为指定的那行Lua代码所对应的第一个指令码。

> ar->i_ci->u.l.savedpc = cl->p->code + 需要跨过的指令码

ar变量就是调试库为我们准备的：

> const lua_Debug *ar

(如果你能跟着思路看到这里，我心里时非常非常的感激，能容忍我这么唠叨这么久。到这里我想表达的内容也差不多结束了，后面两个模块如果有兴趣的话可以稍微了解一下，不是重点。)

七、其他重要的模块

这部分先空着，如果有小伙伴想要详细了解的话，请在公众号(IOT物联网小镇)中留言给我，单独整理成文档。
比较重要的内容包括：

> 1. 标准库的加载过程
> 2. 函数调用栈
> 3. 同时调试多个程序
> 4. 如何处理中断信号
> 5. 如何处理中断信号嵌套问题
> 6. 如何添加自己的库
> 7. 如何同时调试多个程序
> 8. 其他指令的实现机制：查看、修改变量，查看函数调用栈，多个被调试程序的切换等等。

八、调试操作步骤

关于实际操作步骤，用文档表达起来比较费劲，全部是黑乎乎的终端窗口。计划录一个60分钟左右的视频，把上面提到的内容都操作演示一遍，这样效果会更好一下。有兴趣的话可以在B站搜一下我的ID(道哥分享)。

内容主要包括：

> 1. 在Linux平台下：编译和调试步骤。
> 2. Windows平台下：编译和调试步骤。
> 3. 简单的图形调试界面，就是把ldb嵌入到IDE中。

【原创声明】

> 作者：道哥(公众号: IOT物联网小镇)
>
> 知乎：道哥
>
> B站：道哥分享
>
> 掘金：道哥分享
>
> CSDN：道哥分享

如果觉得文章不错，请转发、分享给您的朋友。

我会把十多年嵌入式开发中的项目实战经验进行总结、分享，相信不会让你失望的！

长按下图二维码关注，每篇文章都有干货。

推荐阅读

[1] 原来gdb的底层调试原理这么简单
[2] 生产者和消费者模式中的双缓冲技术
[3] C_C++*静态库*动态库的制作和使用
[4] 利用C可变参数和宏定义来实现自己的日志系统
[5] C与C++混合编程
[6] 拿来即用：用C+JS结构来处理JSON数据
[7] 拿来即用：分享一个检查内存泄漏的小工具