



申威软件调试调优技术

生态软件研发部

开发生态组 - 刘汉旭

目录 Contents

第一章

软件调试介绍

第二章

调试原理和技巧

第三章

申威调优技术

第四章

申威IDE技术

一、软件调试介绍

1

Why?

2

What?

3

How?

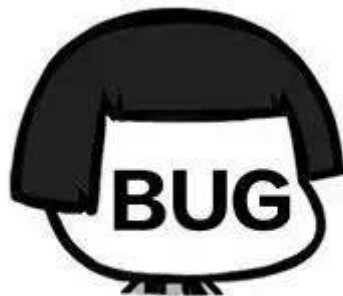
■ 软件调试的诞生背景



The code
I write



What it
does



你从我脸上看到了什么



■ 为什么要软件调试？

- 软件开发过程中，经常出现意料之外的结果
- 从写代码、测试、后期维护，bug 无处不在
- 跟踪程序执行流程，辅助源码分析



据统计，程序员20%时间写代码，80%时间调试



工作三连

查bug



改bug



写bug



👉 软件调试是贯穿程序员一生的必备开发技能

■ 软件调试技术的诞生

- 软件调试技术能够跟踪和记录CPU执行过程，把动态的瞬间“凝固”下来，以供检查和定位错误。
- 软件调试技术还可以分析代码执行逻辑。



☞ 软件是通过指令的组合来指挥硬件，既简单又复杂，是个充满神秘与挑战的世界。而软件调试是帮助人们探索和征服这个神秘世界的有力工具。



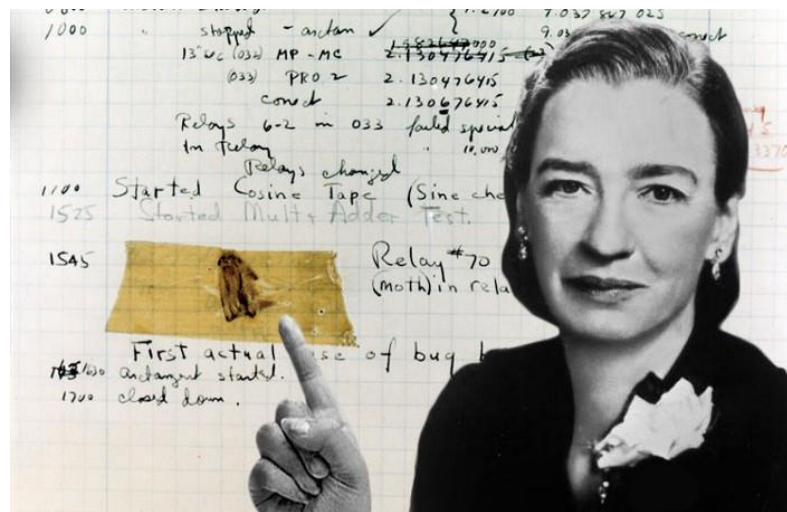
■ 软件调试标准定义

软件调试泛指重现软件缺陷问题、定位和查找问题根源，最终解决问题的过程。

(software debug, debug是在bug前面加上de, 意思是分离和去除bug)

👉 关于 bug 一词的来源

1947, Mark II 计算机运行时突发故障, Grace Hopper 检查发现有只飞蛾卡到了一个继电器中。取出飞蛾后, 计算机恢复正常。Grace Hopper 将这只飞蛾粘贴到工作手册中, 并写上 “First actual case of bug being found”



■ 软件调试与软件测试区别

➤ 目的不同 ★

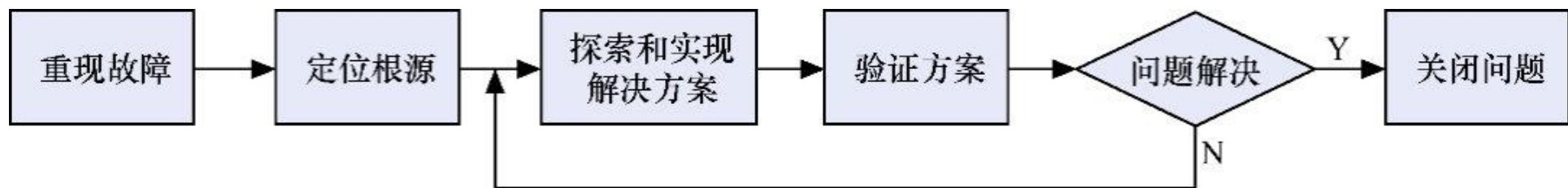
☞ 软件测试目的是发现错误，保障软件质量；（挑错）

☞ 软件调试目的是保证程序正确性，不断地定位错误和解决错误；（排错）

➤ 操作人员不同

☞ 执行软件测试的一般不是开发人员，从而使测试过程更客观有效；

☞ 执行软件调试的是开发人员，有编码就有调试；

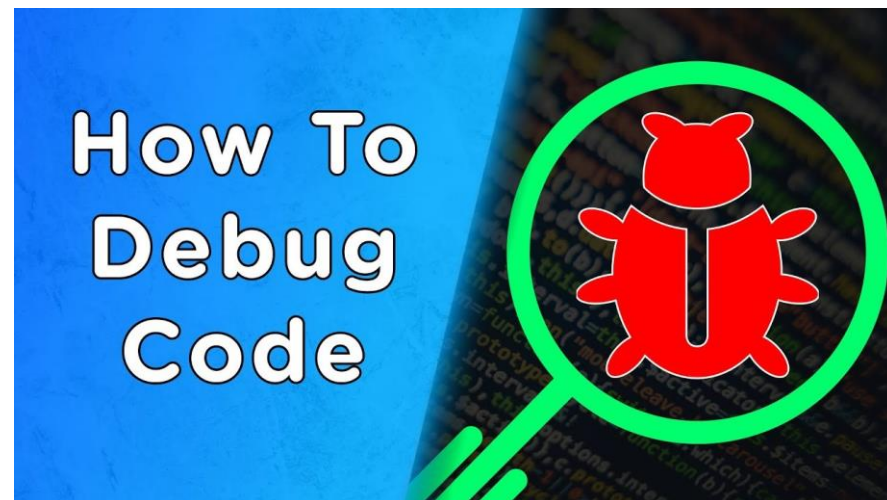


■ 如何进行软件调试？

- 硬核看代码（人工检查分析每行代码）
- 添加打印信息（printf、log日志）
- 调试器（gdb、lldb、WinDbg.....）
- IDE 调试（vs code、eclipse.....）

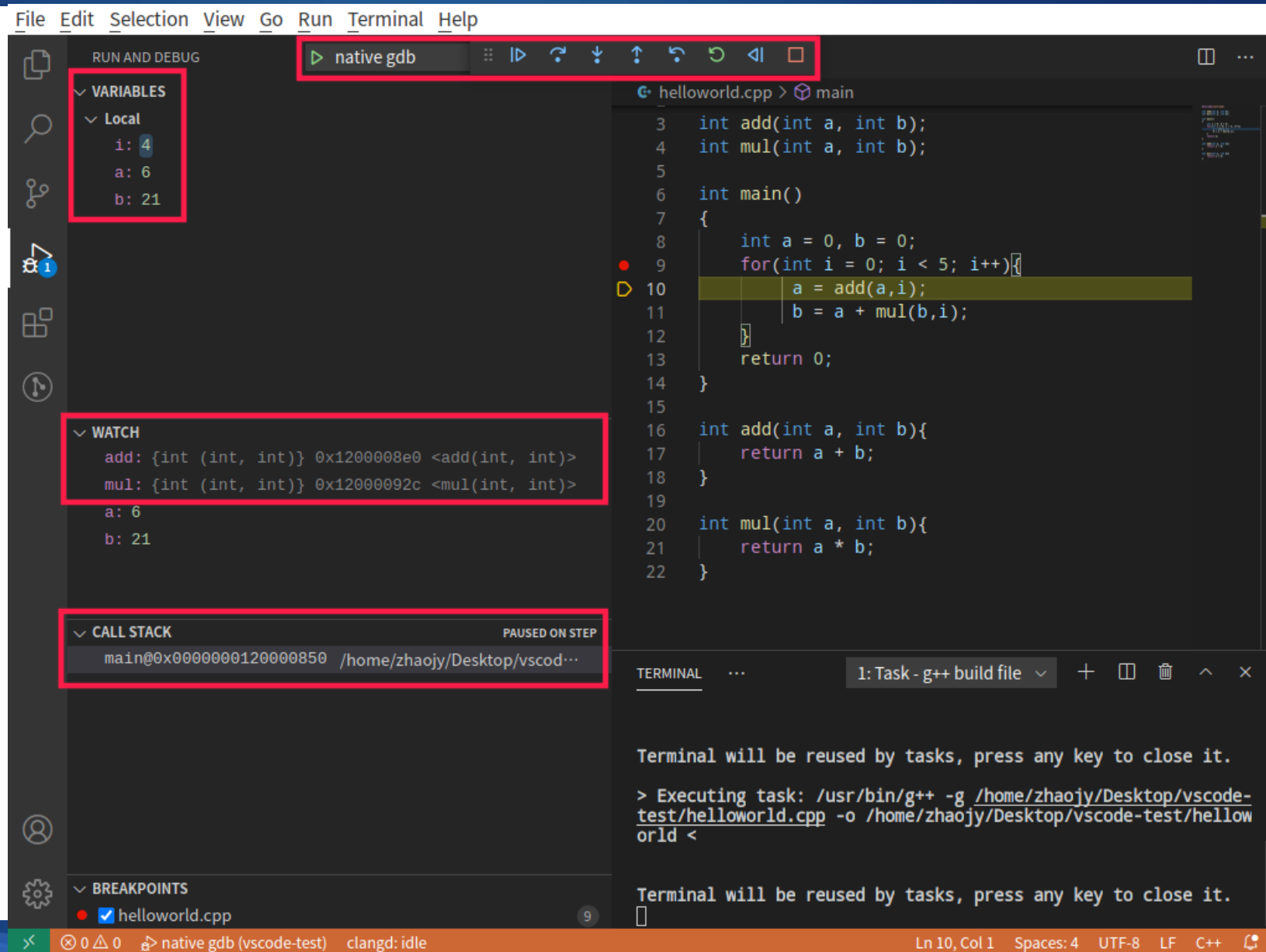
■ Linux环境下常规调试方法：

- 添加打印（printf）+调试器（gdb）



一、软件调试介绍

■ 软件调试场景图



■ 三大平台主流调试器



Linux

GDB（各大Linux发行版默认调试器）、
LLDB



Windows

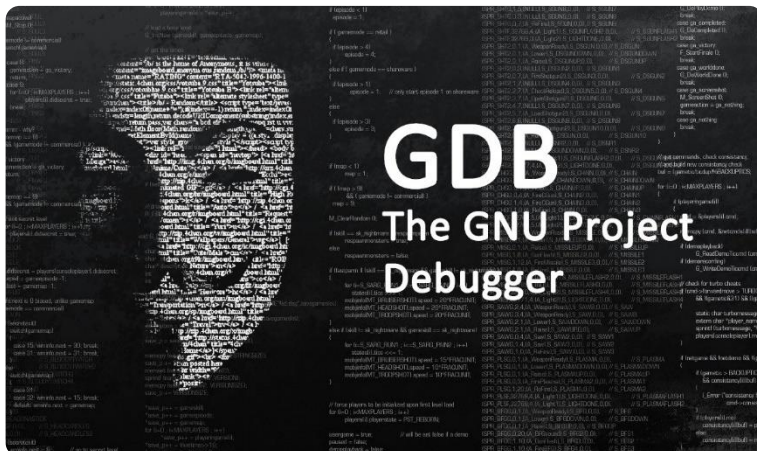
Visual Studio Debugger（Visual Studio自带的调试器）、
WinDbg



Mac OS

LLDB（Xcode默认调试器，Mac OS下开发必备调试器）、
GDB、Visual Studio Debugger

■ 两大调试器区别



GDB (GNU Debugger)

- 重量级，更成熟，基于GCC
- 复杂的C++场景，调试不友好
- gcc编译，优先推荐
- C/C++/FORTRAN/Go.....



LLDB (LLVM Debugger)

- 轻量级，插件化，更好集成LLVM
- C++支持更完善
- Clang编译，优先推荐
- C/C++/Objective C

■ 调试器主要功能

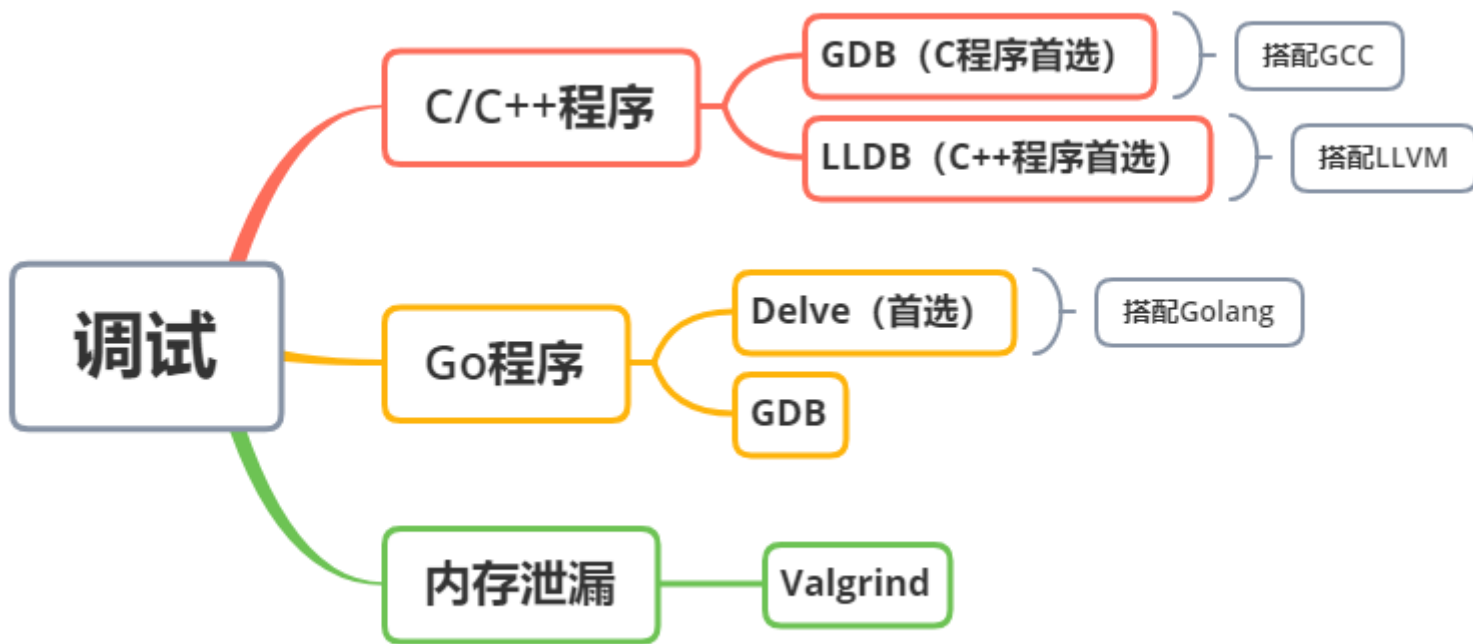
- 控制程序执行：启动、暂停、继续、跟踪、attach、多线程、多进程
- 读取程序状态：变量、传参、CPU寄存器、内存数据、函数栈帧
- 修改程序运行数据：修改变量值、寄存器值、内存地址
- 显示程序代码：显示源代码、汇编代码、机器码.....
- 远程调试：调试远程机、调试开发板.....
- GUI 图形界面：各种窗口（源码/汇编/寄存器/变量/内存/线程

.....

■ 调试器常用命令

常用命令	GDB	LLDB
启动调试器	<code>gdb [program]</code>	<code>lldb [program]</code>
运行程序	<code>(gdb) run</code>	<code>(lldb) run</code>
单步-源码级	<code>(gdb) step/next</code>	<code>(lldb) step/next</code>
单步-指令级	<code>(gdb) stepi/nexti</code>	<code>(lldb) stepi/nexti</code>
跳出当前函数	<code>(gdb) finish</code>	<code>(lldb) finish</code>
设置断点	<code>(gdb) break main</code>	<code>(lldb) breakpoint set --name main</code>
显示寄存器	<code>(gdb) info registers</code>	<code>(lldb) register read</code>
反汇编	<code>(gdb) disassemble main</code>	<code>(lldb) disassable --name main</code>
打印变量	<code>(gdb) print a</code>	<code>(lldb) frame variable a</code>
打印堆栈	<code>(gdb) bt</code>	<code>(lldb) thread backtrace (bt)</code>
显示线程	<code>(gdb) info thread</code>	<code>(lldb) thread list</code>

■ 申威现有调试工具



二、调试原理和技巧

1

系统调用ptrace

2

断点和单步原理

3

调试技巧

■ 系统调用ptrace

ptrace是由 Linux 内核提供的一个功能强大的系统调用，允许一个用户态进程检查、修改另一个进程的内存和寄存器。函数原型如下：

```
#include <sys/ptrace.h>  
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

👉 **ptrace**功能由**request** 参数确定：

⑩ 建立进程间的跟踪关系：PTTRACE_ATTACH、PTTRACE_TRACEME.....

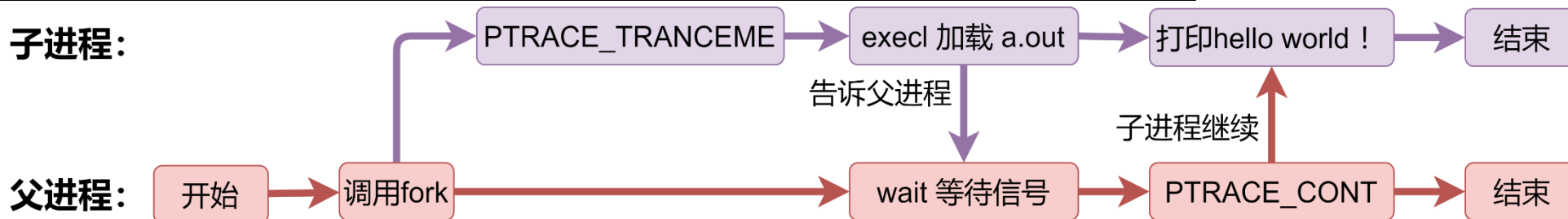
⑩ 读写进程的内存：PTTRACE_PEEKTEXT、PTTRACE_POKETEXT.....

⑩ 读写进程的寄存器：PTTRACE_GETREGSET、PTTRACE_POKEUSR

■ PTRACE TRACEME——控制程序执行

```
1 #include <stdio.h>
2 #include <sys/ptrace.h>
3 #include <sys/wait.h>
4
5 int main() {
6     pid_t child;
7     child = fork();
8     if(child == 0) { // child process
9         ptrace(PTRACE_TRACEME, 0, NULL, NULL);
10        printf("Child run a.out by execl\n");
11        execl("./a.out", "a.out", NULL); // run a.out and send signal
12    } else { // parent process
13        wait(NULL); // wait for child signal
14        ptrace(PTRACE_CONT, child, NULL, NULL); // tell child to continue
15        printf("Parent exit\n");
16    }
17    return 0;
18 }
```

```
lhx@sw6b:~/ppt-code/demo1$ ./test-fork
Child run a.out by execl
Parent exit
lhx@sw6b:~/ppt-code/demo1$ hello world !
```

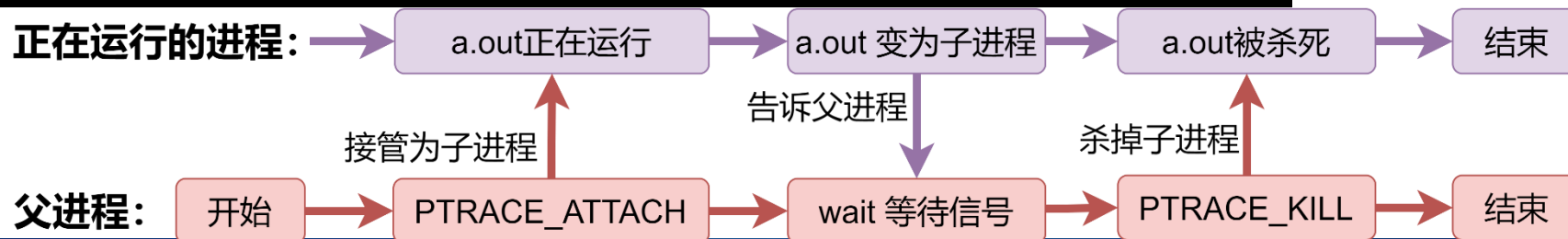


■ PTRACE ATTACH——接管正在运行的进程

```
1 #include <stdio.h>
2 #include <sys/ptrace.h>
3 #include <sys/wait.h>
4
5 int main(int argc, char** argv) {
6     pid_t child;
7     child = atoi(argv[1]);
8     printf("Try to attach child: %d\n", child);
9     int ret = ptrace(PTRACE_ATTACH, child, NULL, NULL); // attach child process
10    if (ret == 0)
11        printf("Attach child %d success !\n", child);
12    else
13        printf("Attach error !\n");
14    wait(NULL); // wait for child signal
15    ptrace(PTRACE_KILL, child, NULL, NULL); // kill child process
16    printf("Kill child success !\n");
17    return 0;
18 }
```

```
lhx@sw6b:~/ppt-code/demo2$ ./a.out
[0] Hello World ! (pid = 13811)
[1] Hello World ! (pid = 13811)
[2] Hello World ! (pid = 13811)
[3] Hello World ! (pid = 13811)
已杀死
```

```
lhx@sw6b:~/ppt-code/demo2$ ./test-attach 13811
Try to attach child: 13811
Attach child 13811 success !
Kill child success !
```



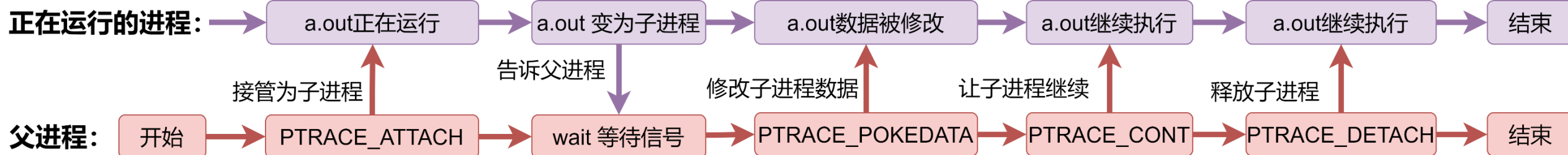
■ PTRACE_POKEDATA——修改进程的内存数据

```
1 #include <stdio.h>
2 #include <sys/ptrace.h>
3 #include <sys/wait.h>
4
5 int main(int argc, char *argv[]) {
6     char new_value = 'B';
7     long address = 0x0000000120012088; // address of variable 'A'
8     pid_t pid = atoi(argv[1]);
9     ptrace(PTRACE_ATTACH, pid, 0, 0); // attach running process by pid
10    wait(NULL);
11    ptrace(PTRACE_POKEDATA, pid, address, new_value); // modify the value at address
12    printf("Modify data succes !\n");
13    ptrace(PTRACE_CONT, pid, 0, 0);
14    ptrace(PTRACE_DETACH, pid, NULL, NULL);
15    return 0;
16 }
17
```

```
lhx@sw6b:~/ppt-code/demo3$ nm a.out | grep var
00000000120012088 G var
```

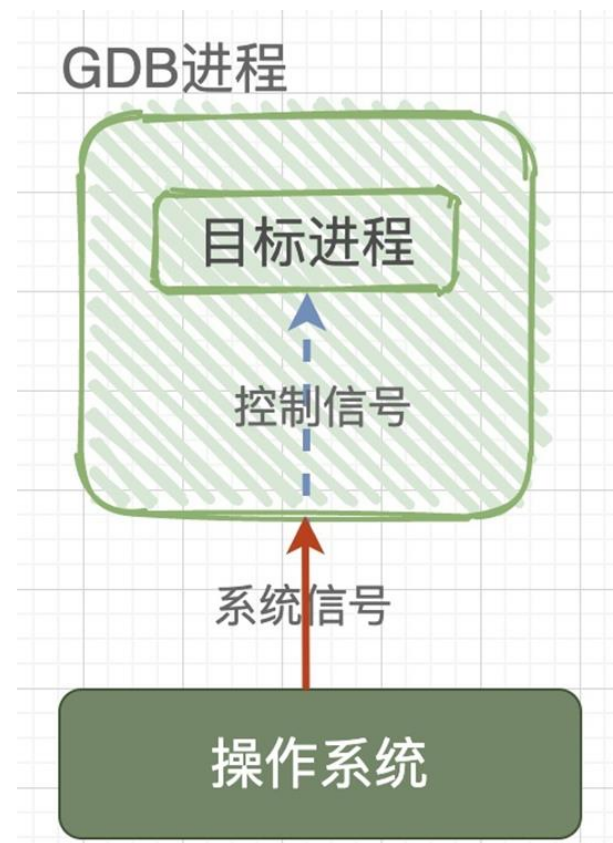
```
lhx@sw6b:~/ppt-code/demo3$ ./a.out
var = A (pid = 16915)
var = A (pid = 16915)
var = A (pid = 16915)
var = A (pid = 16915)
var = B (pid = 16915)
var = B (pid = 16915)
^C
```

```
lhx@sw6b:~/ppt-code/demo3$ ./test-memory 16915
Modify data succes !
```



■ GDB调试原理

- 程序正常运行时，操作系统与目标进程之间是直接交互的；
- gdb调试时，产生追踪者（debugger）和被追踪者（debugee）；
- gdb通过ptrace接管目标进程，操作系统发的信号，都被gdb接收；
- gdb进程通过ptrace读写目标进程的内存地址和寄存器；



■ 窥探断点实现过程

- ◆ 编译时添加-g选项后，会产生源码和汇编对应关系
- ◆ 若想在第5行设置断点，首先找到第5行对应汇编（第11条指令）
- ◆ 替换成申威断点指令，并将原来的指令保存到链表里

1	#include <stdio.h>		7	ldi	\$r1,1
2	int main() {		8	stw	\$r1,16(fp)
3	int a = 1;	→	9	ldi	\$r1,2
4	int b = 2;	→	10	stw	\$r1,20(fp)
5	int sum = a + b;	→	11	ldw	\$r2,16(fp)
6	printf("sum = %d\n", sum);		12	ldw	\$r1,20(fp)
7	return 0;		13	addw	\$r2,\$r1,\$r1
8	}				

1	#include <stdio.h>		7	ldi	\$r1,1	
2	int main() {		8	stw	\$r1,16(fp)	
3	int a = 1;	→	9	ldi	\$r1,2	
4	int b = 2;	→	10	stw	\$r1,20(fp)	
5	int sum = a + b;	→	11	sys_call 0x80	←	ldw \$r2,16(fp)
6	printf("sum = %d\n", sum);		12	ldw	\$r1,20(fp)	
7	return 0;		13	addw	\$r2,\$r1,\$r1	
8	}					

将原指令保存到断点链表里

■ 窥探断点实现过程

◆ 程序执行第11条指令后，产生中断并停止运行，此时pc指向第12条指令

◆ 查找断点链表判断是否为断点，替换回来原指令并进行pc-4操作，退一步回来

◆ 等待用户下一步操作

```
1 #include <stdio.h>
2 int main() {
3     int a = 1;
4     int b = 2;
5     int sum = a + b;
6     printf("sum = %d\n", sum);
7     return 0;
8 }
```

```
7 ldi    $r1,1
8 stw    $r1,16(fp)
9 ldi    $r1,2
10 stw    $r1,20(fp)
11 sys_call 0x80
12 ldw    $r1,20(fp)
13 addw   $r2,$r1,$r1
```

PC指针

brk

发生中断，暂停运行

```
1 #include <stdio.h>
2 int main() {
3     int a = 1;
4     int b = 2;
5     int sum = a + b;
6     printf("sum = %d\n", sum);
7     return 0;
8 }
```

```
7 ldi    $r1,1
8 stw    $r1,16(fp)
9 ldi    $r1,2
10 stw    $r1,20(fp)
11 ldw    $r2,16(fp)
12 ldw    $r1,20(fp)
13 addw   $r2,$r1,$r1
```

回退后的PC指针

从断点链表里
取回原指令

■ 在程序某行设置断点的原理 ★

- ☞ 读取程序某行对应的指令并保存起来；
- ☞ 用中断指令替换掉原来位置的指令；
- ☞ 当程序执行到该处时，会发生断点异常，停止运行；
- ☞ 从断点链表中将保存的指令替换回来，让PC倒退一步后继续运行。



■ 窥探单步实现过程

单步执行目的是执行1行源码，例如从源码第4行单步执行到第5行：

1. 调试器通过符号信息得知第5行源码对应第11行汇编
2. 调试器就控制PC指针一直执行汇编指令
3. 直到第10行执行结束，PC指向第11行时，就停止下来等待用户操作。

step

```
1 #include <stdio.h>
2 int main() {
3     int a = 1;
4     int b = 2;
5     int sum = a + b;
6     printf("sum = %d\n", sum);
7     return 0;
8 }
```

```
7 ldi    $r1,1
8 stw    $r1,16(fp)
9 ldi    $r1,2
10 stw    $r1,20(fp)
11 ldw    $r2,16(fp)
12 ldw    $r1,20(fp)
13 addw   $r2,$r1,$r1
```

最初的PC指针

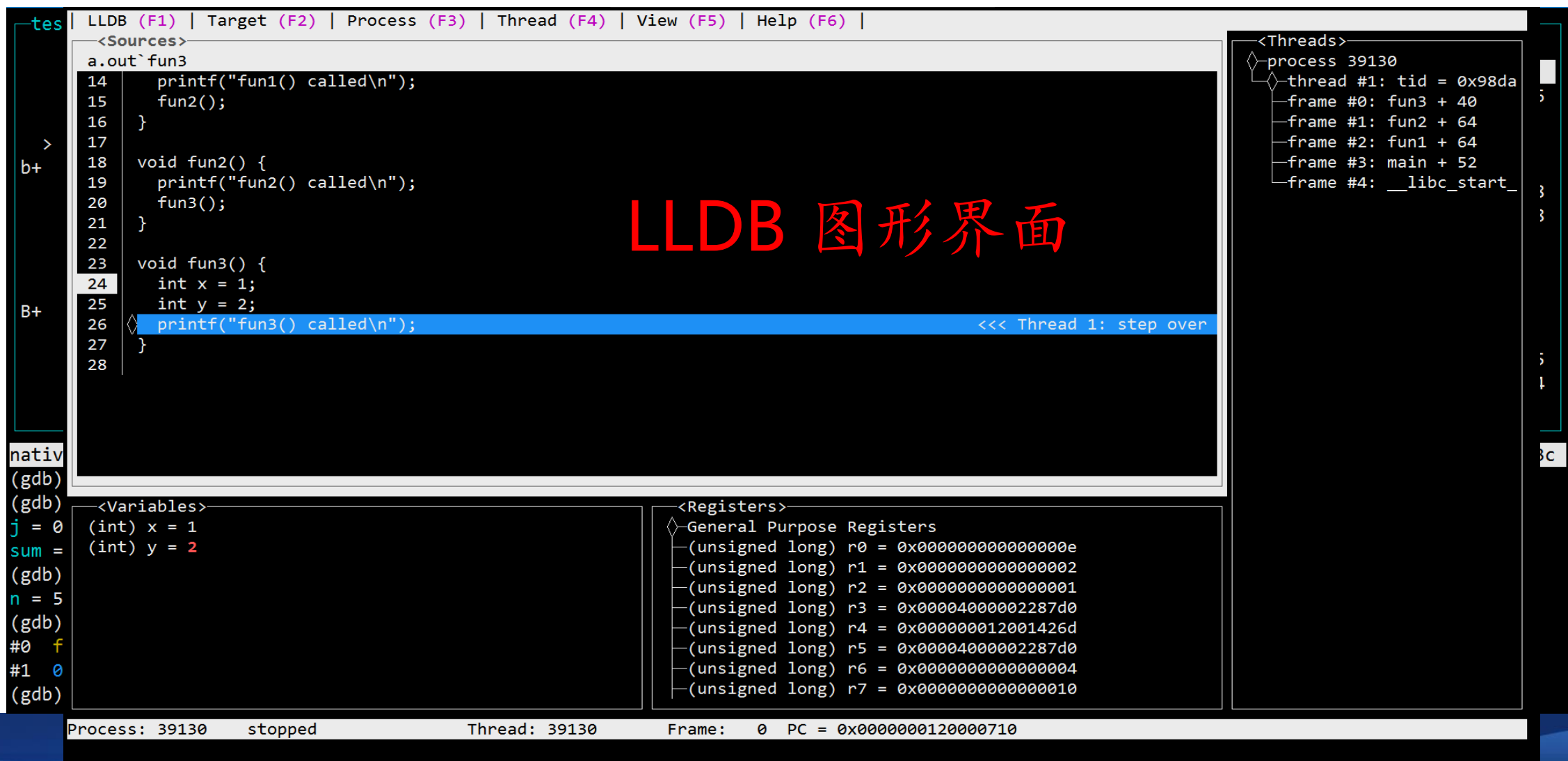
```
1 #include <stdio.h>
2 int main() {
3     int a = 1;
4     int b = 2;
5     int sum = a + b;
6     printf("sum = %d\n", sum);
7     return 0;
8 }
```

```
7 ldi    $r1,1
8 stw    $r1,16(fp)
9 ldi    $r1,2
10 stw    $r1,20(fp)
11 ldw    $r2,16(fp)
12 ldw    $r1,20(fp)
13 addw   $r2,$r1,$r1
```

单步后的PC指针

■ 调试技巧1：使用GUI调试界面

LLDB 图形界面

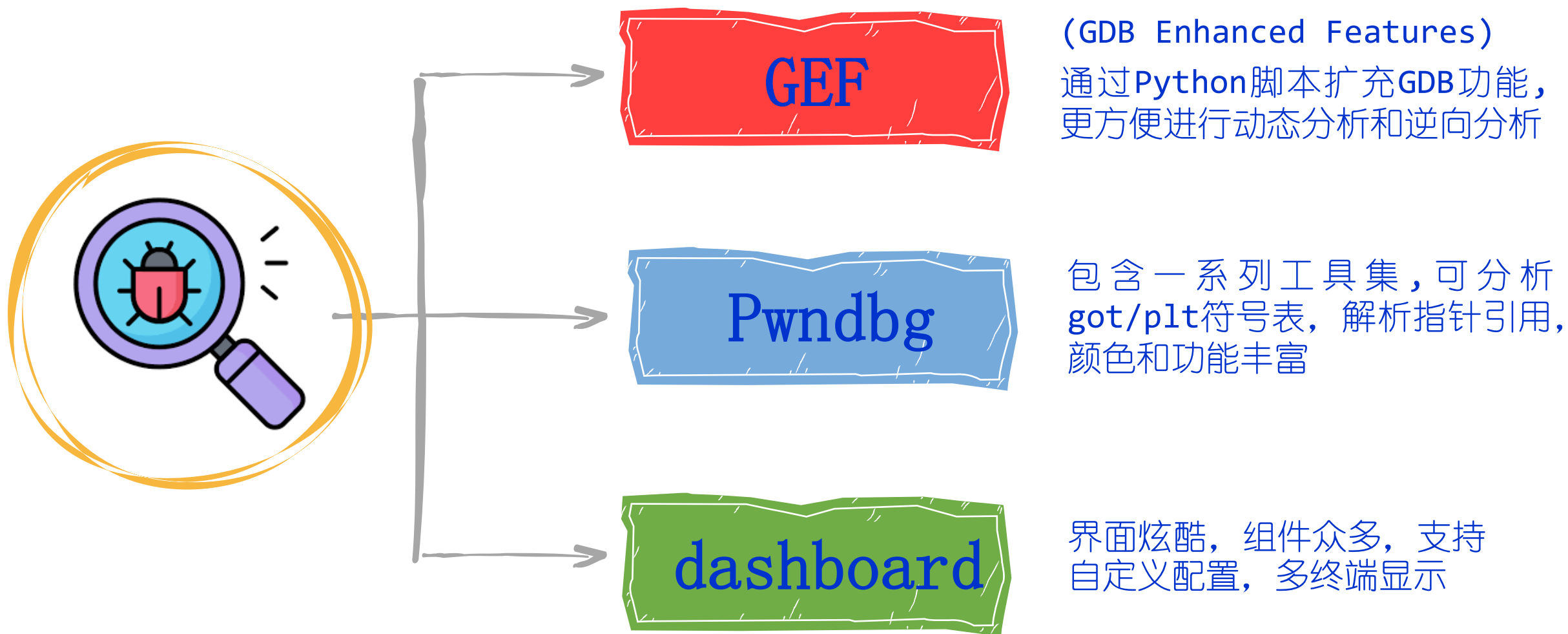


The screenshot displays the LLDB graphical user interface with the following components:

- Top Menu:** LLDB (F1) | Target (F2) | Process (F3) | Thread (F4) | View (F5) | Help (F6) |
- Left Sidebar:** Contains a tree view with items like 'tes', 'b+', and 'B+'. The 'B+' item is selected, showing a list of variables: `j = 0`, `sum =`, `n = 5`, `#0 f`, and `#1 0`.
- Source View:** Displays the source code for `a.out`fun3`. The code includes functions `fun1`, `fun2`, and `fun3`. Line 26, `printf("fun3() called\n");`, is highlighted in blue, indicating the current execution point. A status bar at the bottom of this view reads `<<< Thread 1: step over`.
- Threads View:** Shows the call stack for process 39130, thread #1 (tid = 0x98da). The stack frames are: `frame #0: fun3 + 40`, `frame #1: fun2 + 64`, `frame #2: fun1 + 64`, `frame #3: main + 52`, and `frame #4: __libc_start_`.
- Variables View:** Shows the current values of variables: `(int) x = 1` and `(int) y = 2`.
- Registers View:** Lists the general purpose registers (r0 through r7) with their current values in hexadecimal.
- Bottom Status Bar:** Displays the current state: `Process: 39130 stopped Thread: 39130 Frame: 0 PC = 0x000000120000710`.

二、调试原理和技巧

■ 调试技巧2：通过插件增强功能



二、调试原理和技巧

Breakpoint 1 at test.c:14

Source

```
9 printf("sum = %d\n", sum);
10 return sum;
11 }
12 int main()
13 {
!14 int a = 0, b = 0, i;
15 b = fun(5);
16 printf("b = %d \n",b);
17 for(i = 0; i <= 5; i++)
18 {
```

Stack

```
[0] from 0x0000000120000670 in main+24 at test.c:14
```

Threads

```
[1] id 29747 name a.out from 0x0000000120000670 in main+24 at test.c:14
```

```
pwndbg> tty /dev/pts/10
pwndbg> cwatch cnt
pwndbg> cwatch execute "ds BUF"
pwndbg> run
Starting program: /tmp/pwndbg/test

Breakpoint 1, func (arg=5) at test.c:19
19 fprintf(stdout, "%d\n", cnt);
pwndbg>
```

Assembly

```
0x000000012000065c main+4 ldi $r29,-30776($r29)
0x0000000120000660 main+8 ldi sp,-32(sp)
0x0000000120000664 main+12 stl ra,0(sp)
0x0000000120000668 main+16 stl fp,8(sp)
0x000000012000066c main+20 bis $r31,sp,fp
!0x0000000120000670 main+24 stw $r31,16(fp)
0x0000000120000674 main+28 stw $r31,24(fp)
0x0000000120000678 main+32 ldi $r16,5($r31)
0x000000012000067c main+36 ldih $r27,0($r29)
0x0000000120000680 main+40 ldl $r27,-32752($r27)
```

Variables

```
loc a = 501024, b = 0, i = 512
```

dashboard

二、调试原理和技巧



■ 调试技巧3：学会反向调试

```
test.c
19      printf("fun2() called\n");
20      fun3();
21  }
22
23  void fun3() {
B+>24      int x = 0;
25      int y = 5;
26      for(int i=0; i<5; i++)
27      {
28          x++;
29          y--;
30      }
31      printf("x = %d, y = %d\n", x, y);

process 42994 In: fun3          L24    PC: 0x120000760
(gdb) █
```

已连接 172.16.129.107:22, SSH2 xterm 58x23 17,7 1 会话 CAP NUN

```
0x12000075c <fun3+20> bis    $r31,sp,fp
B+>0x120000760 <fun3+24> stw    $r31,16(fp)
0x120000764 <fun3+28> ldi    $r1,5($r31)
0x120000768 <fun3+32> stw    $r1,20(fp)
0x12000076c <fun3+36> stw    $r31,24(fp)
0x120000770 <fun3+40> br     $r31,0x120000798 <fu
0x120000774 <fun3+44> ldw    $r1,16(fp)
0x120000778 <fun3+48> addw   $r1,0x1,$r1
0x12000077c <fun3+52> stw    $r1,16(fp)
0x120000780 <fun3+56> ldw    $r1,20(fp)
0x120000784 <fun3+60> subw   $r1,0x1,$r1
0x120000788 <fun3+64> stw    $r1,20(fp)
0x12000078c <fun3+68> ldw    $r1,24(fp)

process 44431 In: fun3          L24    PC: 0x120000760
(gdb) layout asm
(gdb) █
```

已连接 172.16.129.107:22, SSH2 xterm 58x23 18,7 1 会话 CAP NUN

三、申威调优技术

1

调优目的

2

调优原理

3

调优工具

■ 为什么要调优？

日常使用软件时，你是否有过如下体验：

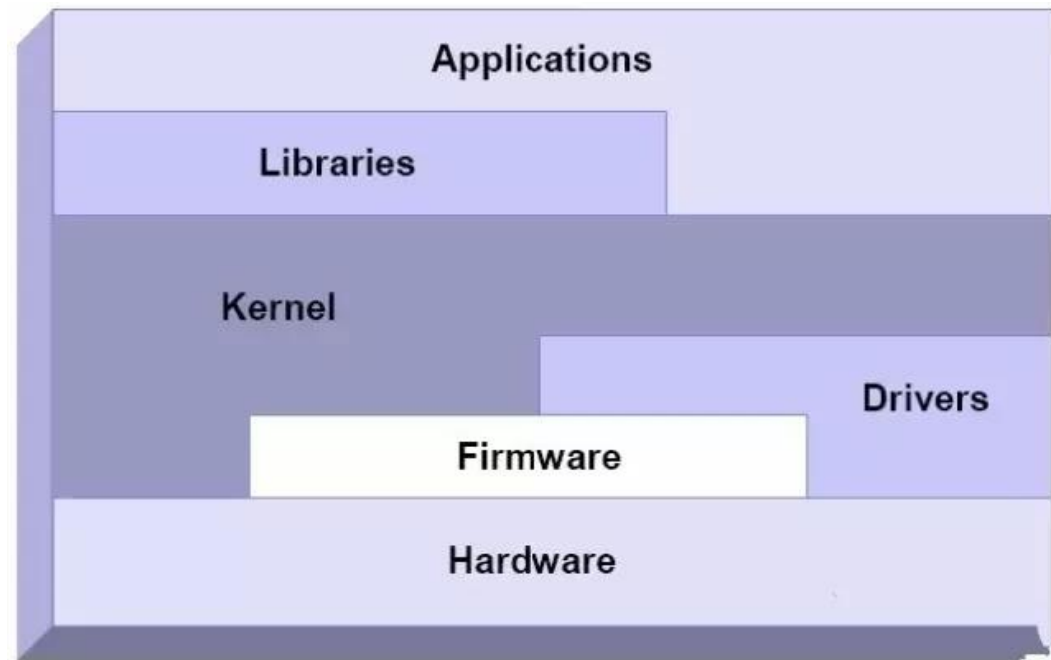
- 程序运行好慢啊，它到底在干什么？？
- 界面卡顿？鼠标没响应？
- 系统资源不足，程序崩溃？



■ 什么是性能调优？

计算机体系结构中，核心三部分：

- 硬件 (Hardware)
- 操作系统 (Kernel)
- 应用 (Applications)



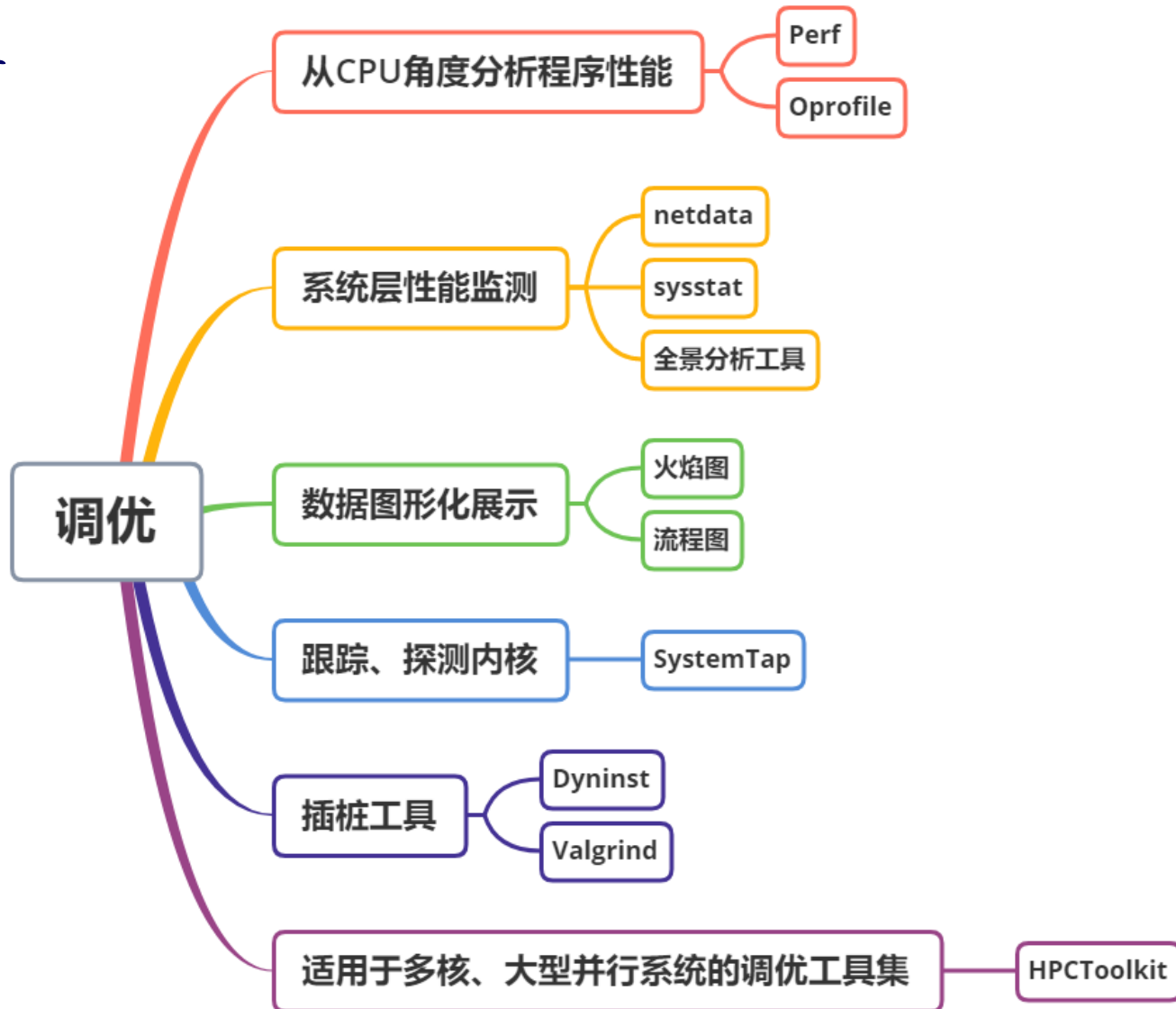
★ 性能调优就是在对硬件、操作系统和应用了解的基础上，调节三者之间的关系，实现整个系统的性能最大化，并能不断的满足现有的业务需求。

■ 调优工具的原理——Linux的可观测性支持

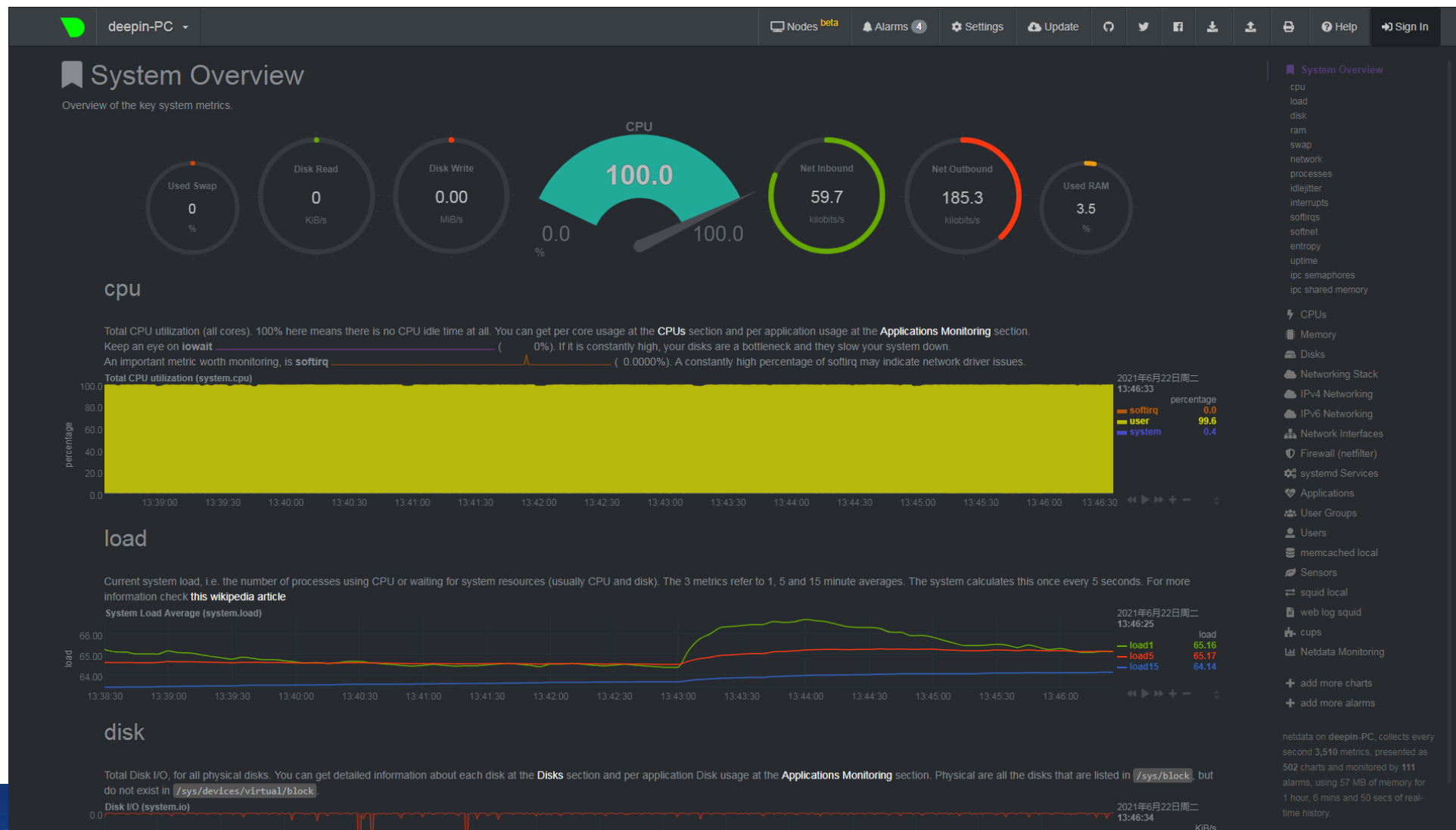
- 硬件提供了事件计数、统计功能
- 操作系统提供了可观测性支持
- 调优工具基于这些功能获取数据，
收集系统内各种信息。

类型	数据来源
进程计数器	/proc
全系统计数器	/proc, /sys
设备配置和计数器	/sys
Cgroup统计	/sys/fs/cgroup
进程跟踪	ptrace
硬件计数器(PMCs)	perf_event
网络统计	netlink
网络数据包捕获	libpcap
线程延迟指标	Delay accounting
全系统追踪	tracepoints, software events, kprobes, uprobes, perf_event

■ 申威现有调优工具



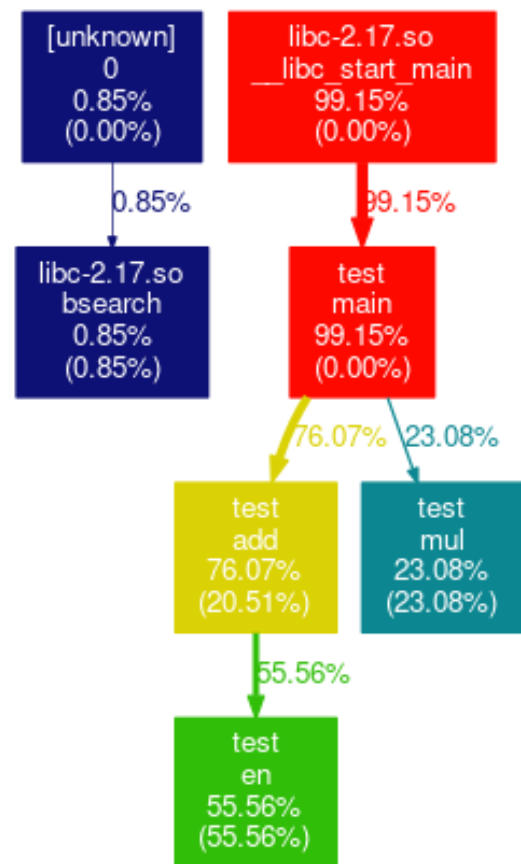
■ 调优工具——netdata展示图



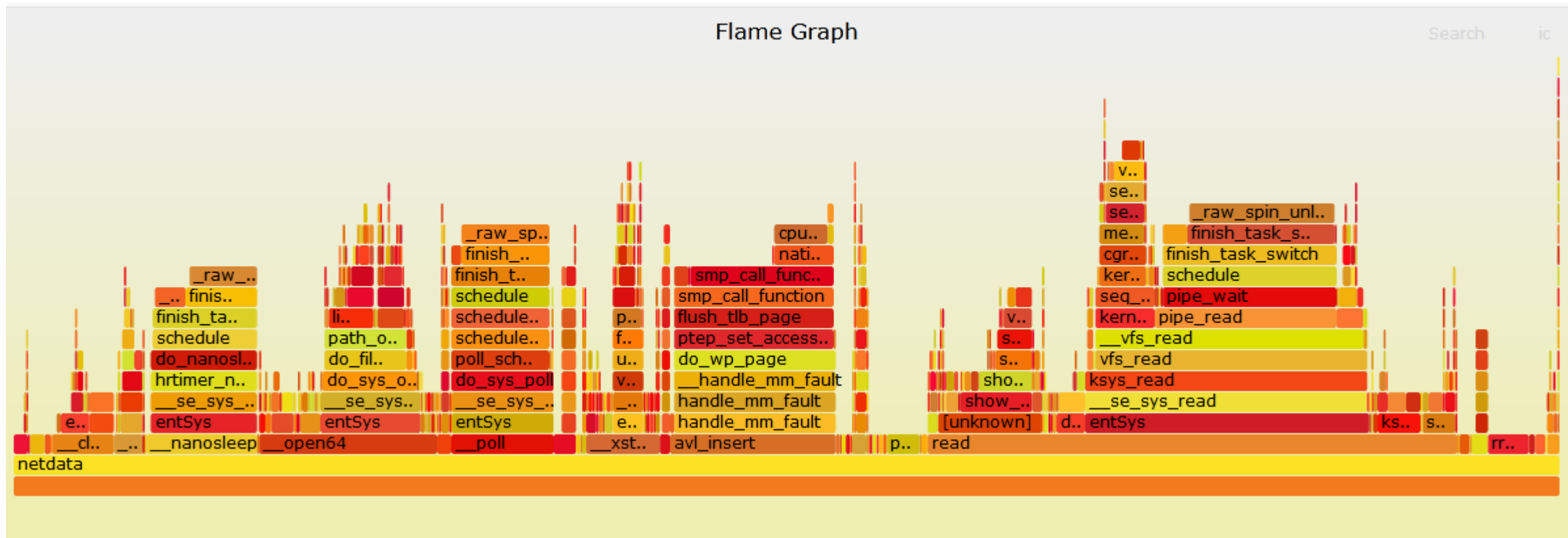
■ 调优工具——流程图

```
#include<stdio.h>
#define NUM 3921
int en(int* a,int* b){.....}
int add(int* a,int* b,int num){.....en(a,b);.....}
int mul(int* a ,int* b,int num){.....}

int main(){
.....    if(.....){
                add(a,b,NUM);
        }else{
                mul(a,b,NUM);
        }
.....}
```



■ 调优工具——火焰图



四、申威IDE技术

👉 什么是IDE?

👉 IDE的优势?

👉 申威有哪些IDE?

■ 什么是 IDE ？

集成开发环境 (Integrated Develop Environment) 是一种软件应用程序，为计算机程序员提供全面的软件开发工具。IDE通常至少由一个源代码编辑器、构建自动化工具和一个调试器组成。

IDE = 文本编辑器 + 调试器 + 构建工具

■ IDE 的优势



■ 常见的 IDE



**Visual Studio
Code**

在 Windows、macOS 和 Linux 上运行的独立源代码编辑器。JavaScript 和 Web 开发人员的最佳选择，具有大量扩展以几乎支持任何编程语言。

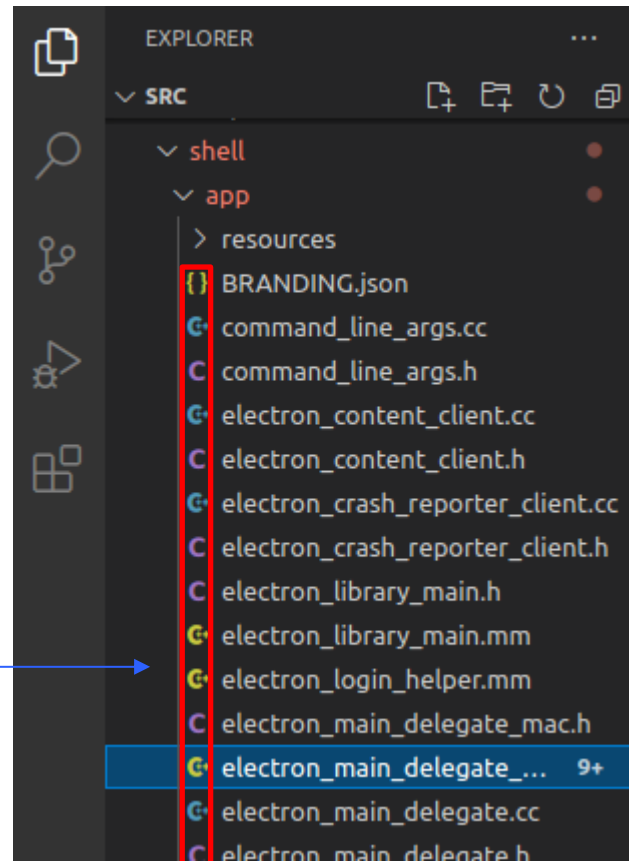
■ VSCode的优势

	主流开发环境	Visual Studio Code
开发技术	软件开发技术， 与底层依赖性强	Web开发技术， 支持跨平台开发
显著特征	完整的ide和调试器 体积较大	轻量级、多语言支持、 高度自定义的插件支持
兼容性	单平台	包括linux、windows等主流平台
开源情况	商业化非开源	开源

■ IDE 的优势1：资源管理器

👉 与终端的文件管理相比，IDE能够收起或打开目录显示内容。同时IDE也能够同时打开多个窗口。

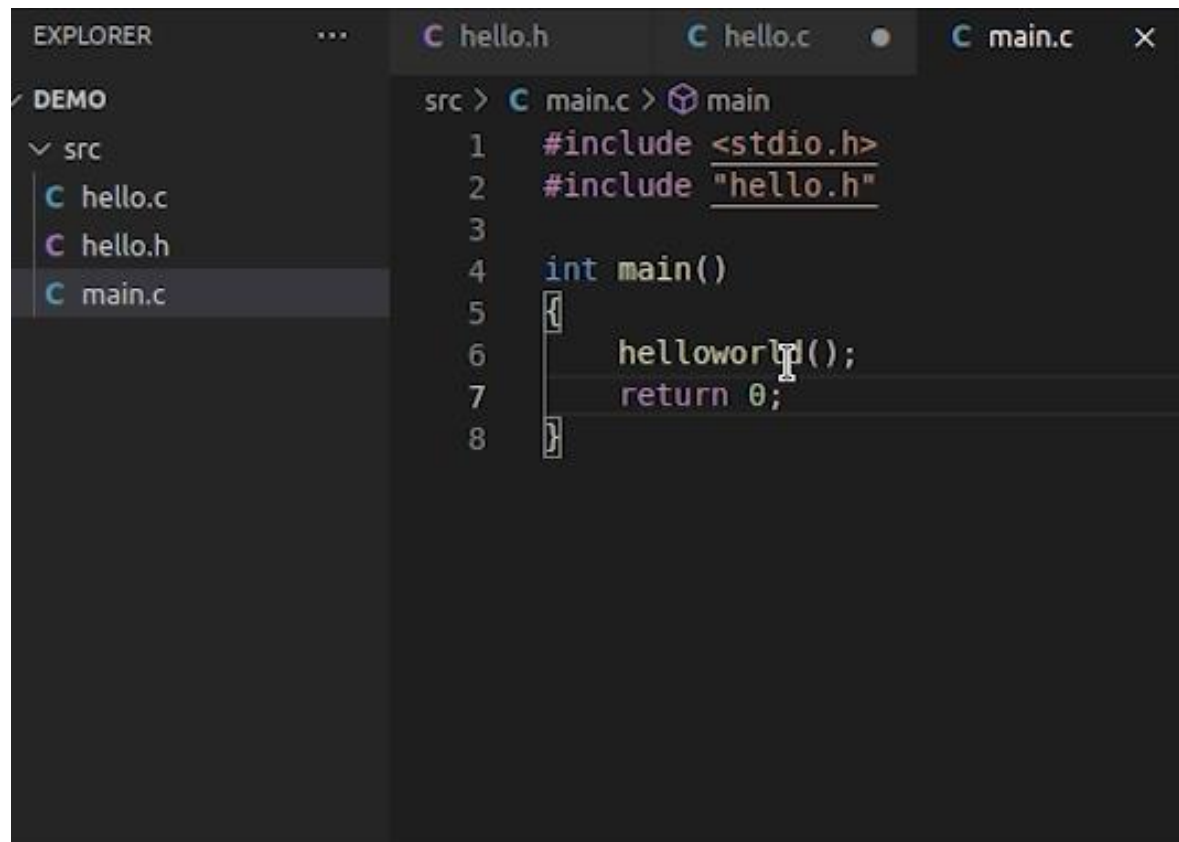
IDE能通过图标更细致地
区分目录下文件的类型



■ IDE 的优势2：语法高亮、函数跳转

👉 IDE能够更全面地展示语法高亮

并支持点击函数名跳转至函数定义处

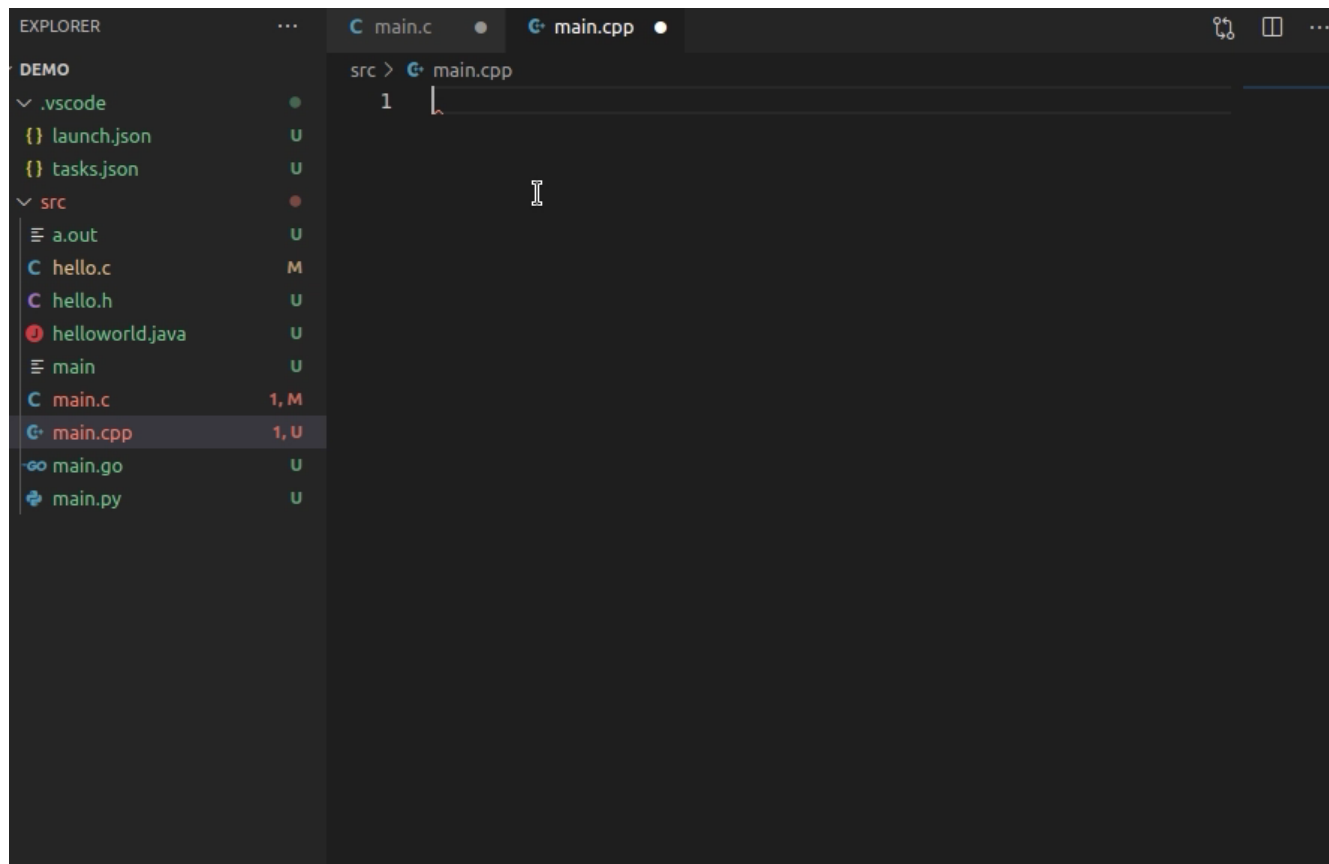


The screenshot shows an IDE interface with a dark theme. On the left, the 'EXPLORER' panel displays a project structure with a 'DEMO' folder containing a 'src' subfolder. Inside 'src', three files are listed: 'hello.c', 'hello.h', and 'main.c', with 'main.c' selected. The main editor area shows the content of 'main.c'. The code includes two preprocessor directives: `#include <stdio.h>` and `#include "hello.h"`. Below these, the `main` function is defined as `int main()`. The function body contains a call to `helloworld();` and a `return 0;` statement. The code is syntax-highlighted, with keywords in blue, strings in red, and identifiers in white. A cursor is positioned at the end of the `helloworld();` line.

```
src > C main.c > main
1  #include <stdio.h>
2  #include "hello.h"
3
4  int main()
5  {
6      helloworld();
7      return 0;
8  }
```

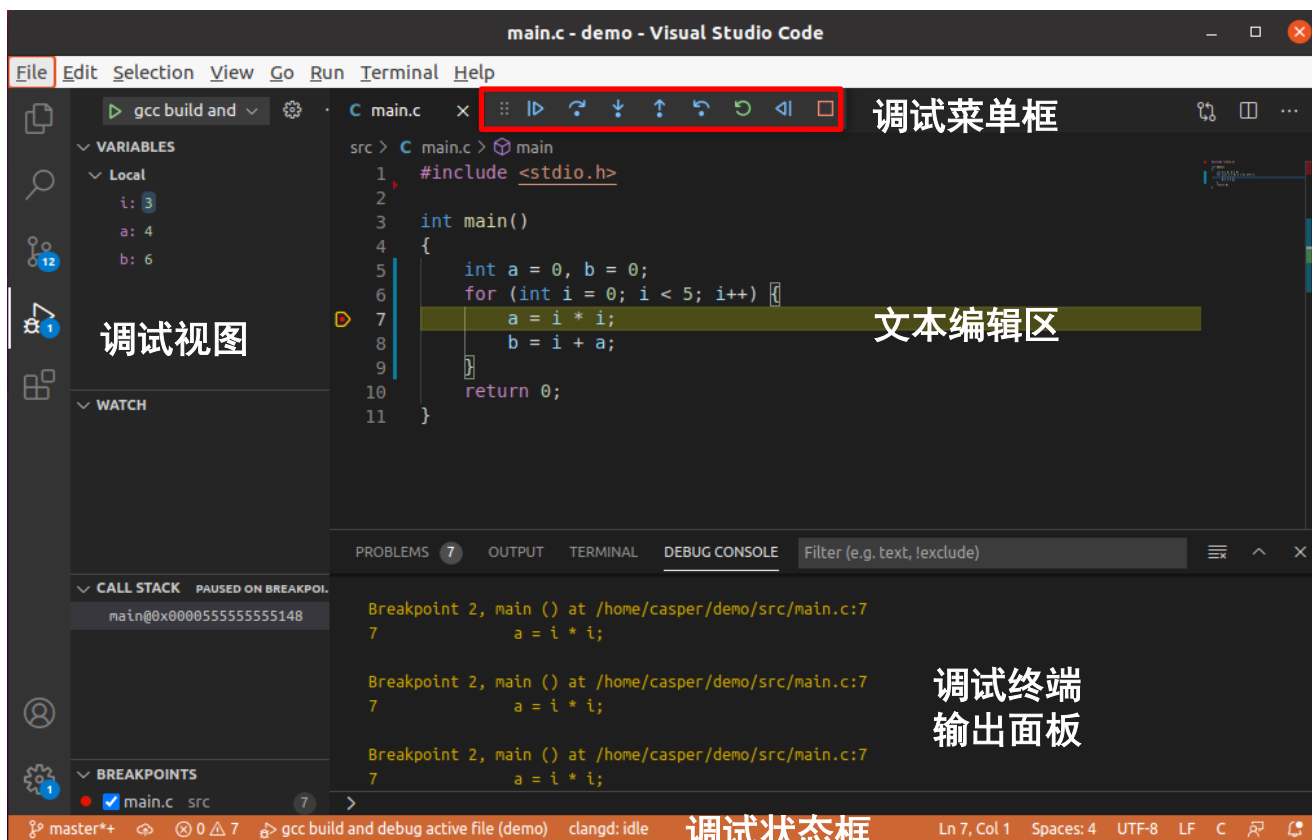
■ IDE 的优势3：智能补全

👉 IDE能够更全面地展示语法高亮



■ IDE 的优势4：调试集成

👉 运行调试功能集成与编辑器内部



■ 申威VSCode特性



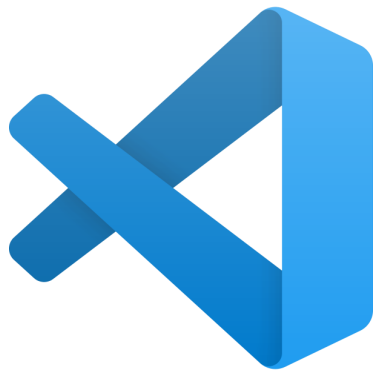
支持多种申威架构平台

支持C/C++、Java、GO等多种语言开发
可调用申威编译器和调试器

已支持多种服务于申威架构开发库

适配市面上主流插件

■ 申威 IDE 研发进展



VSCode申威适配情况	
版本支持	1.36.0、1.54.3、1.56.2
环境支持	sw421、sw1621、sw3231
系统支持	麒麟V10、UOS 20
插件支持	clangd、language support for Java、go and more...



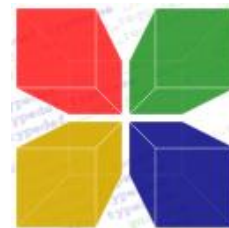
Eclipse申威适配情况	
版本支持	4.5.1、4.11.0、4.18.0
环境支持	sw421、sw1621、sw3231
系统支持	麒麟V10、UOS 20
插件支持	CDT(c++ develop tools)、goclipse

四、申威 IDE 技术



netbeans申威适配情况

版本支持	8.0.2
环境支持	sw421、sw1621、sw3231
系统支持	麒麟V10、UOS 20

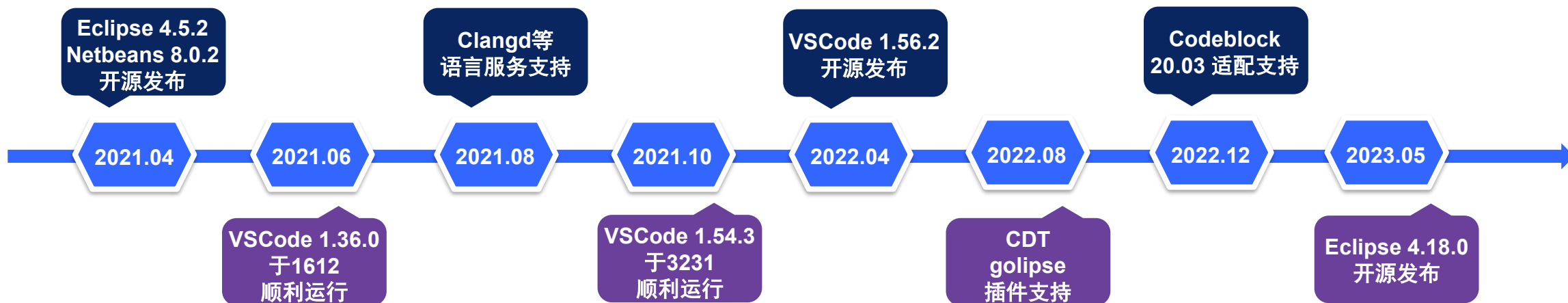


Code::Blocks

The open source, cross-platform IDE

codeblocks申威适配情况

版本支持	20.03
环境支持	sw421、sw1621、sw3231
系统支持	麒麟V10、UOS 20





谢谢各位！

生态软件开发部

开发生态组 - 刘汉旭