文档标识：　　编号：

版　　本：V1.0　　密级：公开

技术文件

集成开发环境（IDE）

V1.0

产品说明书

北京麟卓信息科技有限公司

二〇二五年四月

目 录

1 软件介绍

1.1软件简介

麟卓通用集成开发环境（IDE）由北京麟卓信息科技有限公司研发，致力于从软件开发调试、性能分析两个方面降低软件调试分析难度，提升软件开发效率和程序鲁棒性的一款软件。软件支持 VScode 1.54.3 版以上的安装、启动、卸载、更新的管理和维护。包含串行/并行程序的轻量级调试和分析、基于事件的智能历史调试、GDB 堆栈反向调试与历史调试、面向生产环境的历史调试、图形化历史调试、基于快照的内存性能分析、面向共享资源的资源争用分析和基于处理器性能监控部件（PMU）的性能分析等功能。

1.2使用对象

该软件支持与 VSCode 集成，主要面向使用 VSCode 开发的用户。

1.3使用范围

支持 x86、ARM、MIPS64、LoongArch、SW64 等平台的 Linux 兼容。

1.4架构

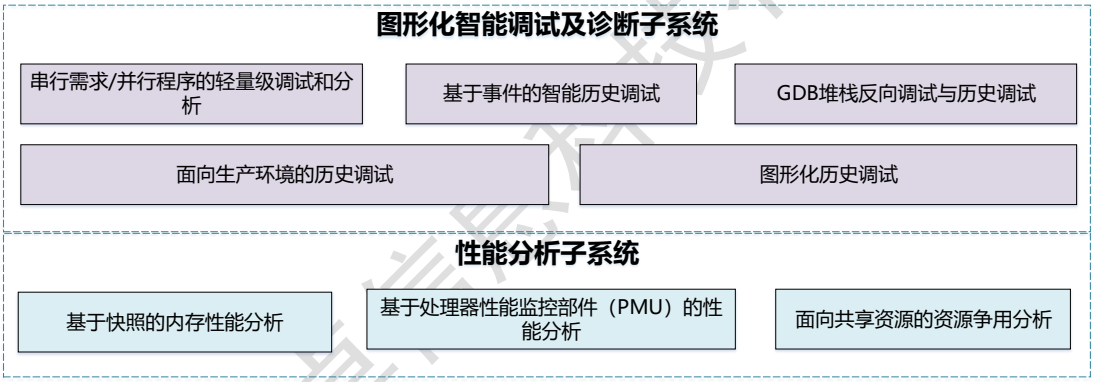


图 1 麟卓通用 IDE 架构

1.5运行环境

■ 建议硬件环境：

- 处理器：支持多种国产处理器（飞腾、鲲鹏、海光、兆芯等）以及 X86 处理器
- 内存：大于 4G
- 硬盘容量：大于 500G

■ 建议软件环境：

- Linux 操作系统：支持 Kylin、UOS、中科方德等多种国产操作系统，支持 Ubuntu、CentOS 等多种主流 Linux 操作系统

■ 建议网络配置：

- 带宽：大于 10M

1.6编程语言及程序量

编程语言：JAVASCRIPT 和 TypeScript

程序量　：33 万行

2　安装/卸载

2.1安装

集成开发环境 deb 包安装。

2.2卸载

■找到相关的插件，点击右下角设置进行卸载，如卸载【resource-analysis】；在扩展里，找到【resource-analysis】；如图 2

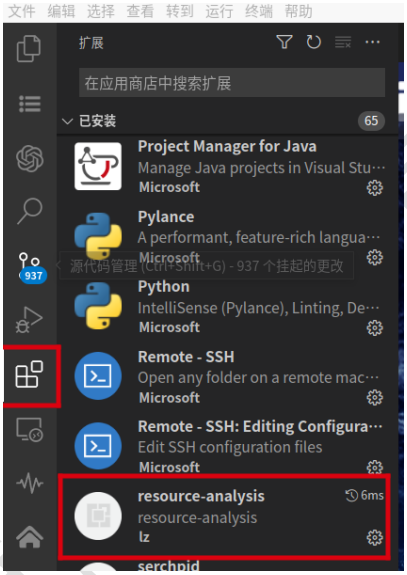


图 2 找到插件

■点击右下角设置按钮，选择【卸载】，进行卸载，如图 3

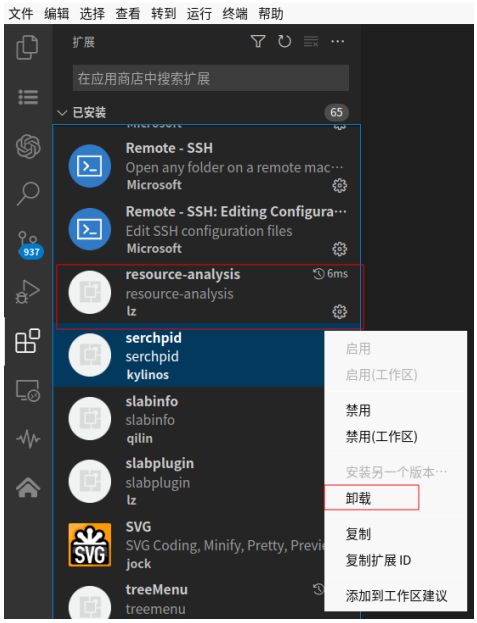


图 3 卸载插件

3 麟卓通用 IDE 的启用/关闭

3.1启动

通过左侧【性能分析】菜单栏进入性能分析功能菜单。图形化智能调试诊断功能调试时自动启动。

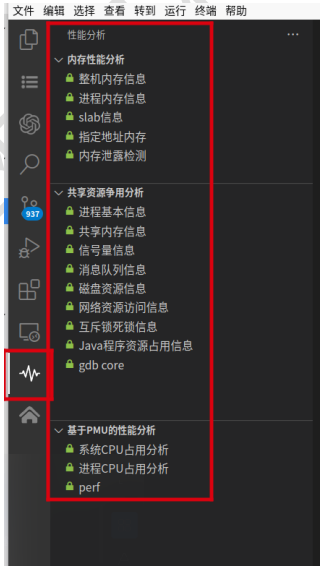


图 4 启动

3.2关闭

如点击 x 号关闭。如图 5

图 5 关闭

4 基础编辑调试功能

4.1多语言源代码编辑

麟卓 IDE 首先是一个编辑器，它包含了高效源代码编辑所需的功能，提供了对包括 C、C++、Fortran、Python、Java、JavaScript、C#在内的多种主流编程语言的全面支持。

以 C/C++语言为例，可帮助你编写代码、理解代码并在源文件中导航。如果找不到引用的头文件，IDE 会在引用它的每个 #include 指令下方显示一个波浪线。

图 6 源代码编辑功能

4.2多选（多光标）

功能介绍：IDE 支持多个光标，以便快速同时编辑。

操作步骤：

可以使用【Alt+单击】添加辅助光标。每个光标都根据其所在的上下文独立运行。

可以使用 【Ctrl+Alt+Down】或【Ctrl+Alt+Up】，在下方或上方插入光标。

```
.global-message-list.transition {
    -webkit-transition: top 200ms linear;
    -ms-transition:     top 200ms linear;
    -moz-transition:    top 200ms linear;
    -khtml-transition:  top 200ms linear;
    -o-transition:      top 200ms linear;
    transition:         top 200ms linear;
}
```

图 7 多光标操作

4.3查找和替换

功能介绍：IDE 提供快速查找文本并在当前打开的文件中替换功能。

操作步骤：

1、按【Ctrl+F】在编辑器中打开查找输入框，搜索结果将在编辑器、概览标尺和小地图中突出显示。

如果当前打开的文件中有多个匹配的结果，则可以按 Enter 和 Shift+Enter 在查找输入框聚焦时导航到下一个或上一个结果。

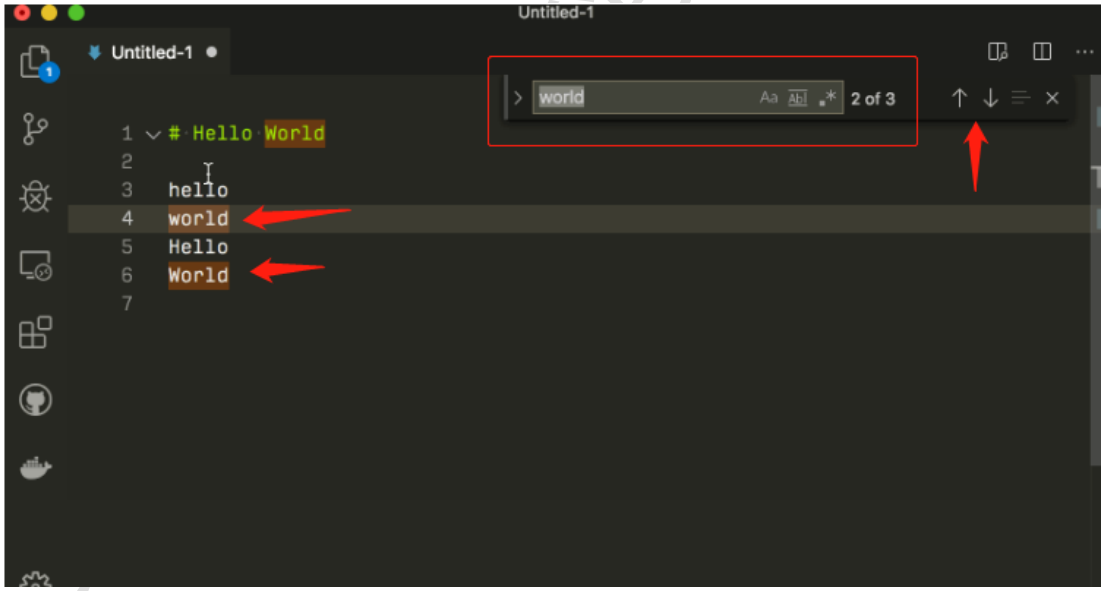


图 8 文件内查找

2、点击下拉箭头打开替换输入框，IDE 支持在搜索和替换时更改正则表达式匹配组的大小写。

支持单个替换与全部替换

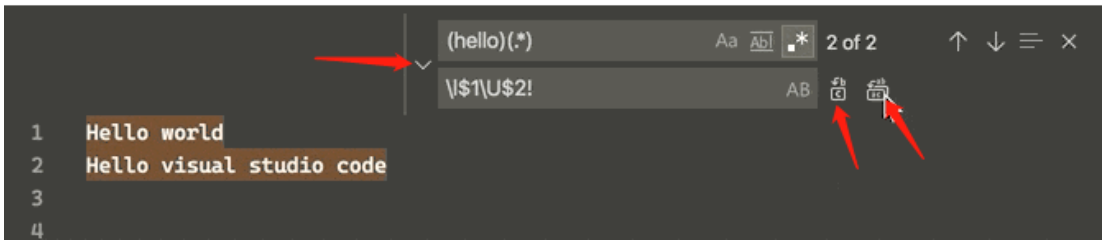


图 9 文件内替换

4. 4跨文件搜索替换

功能介绍：允许快速搜索当前打开的文件夹中的所有文件，并支持替换操作。

操作步骤：

1、按 Ctrl+Shift+F 并输入搜索词。搜索结果被分组到包含搜索词的文件中，并指示每个文件中的命中数及其位置。展开文件以查看该文件中所有命中的预览。然后单击其中一个命中以在编辑器中查看它。

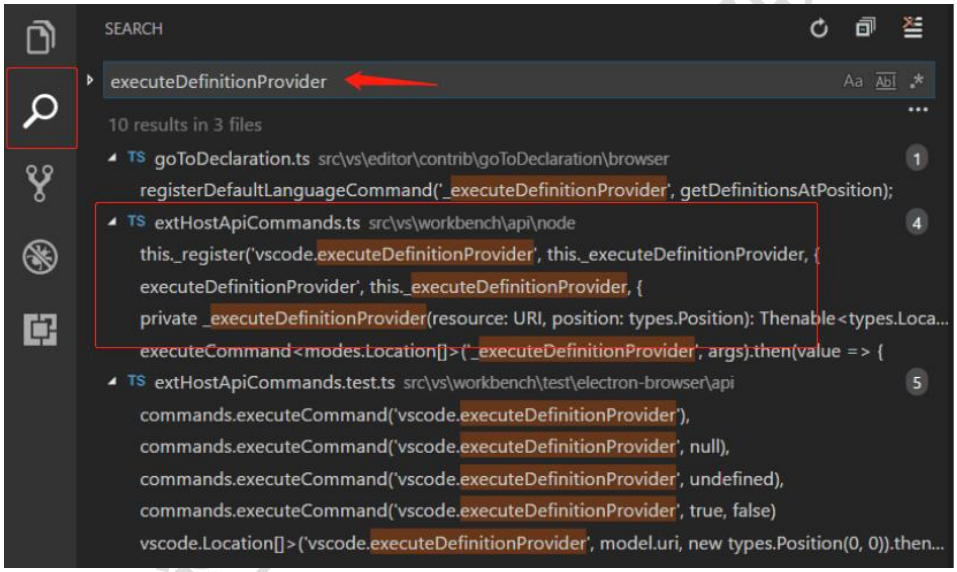


图 10 跨文件搜索

2、可以进行跨文件替换，点击搜索输入框左侧箭头，可以展开替换输入框。在替换输入框中输入文本时，可以看到对应更改的差异显示。

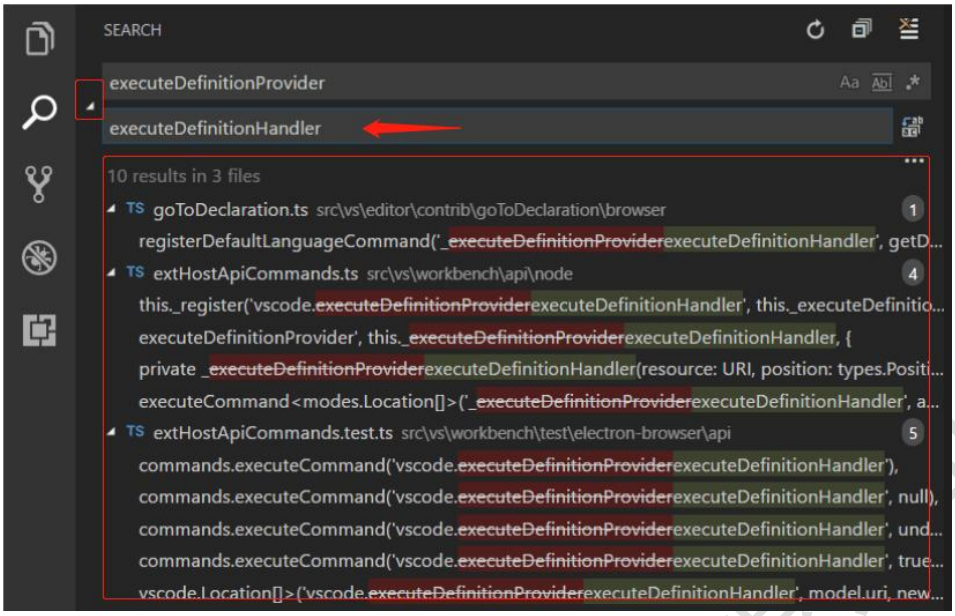


图 11 跨文件替换

4.5智能补全

功能介绍：提供基于语言语义和源代码分析的智能代码补全。在输入代码时将弹出智能补全提示。

操作步骤：

键入字符时，会筛选成员列表（变量、方法等），显示包含输入字符的成员。按 Tab 或回车键将插入所选成员。

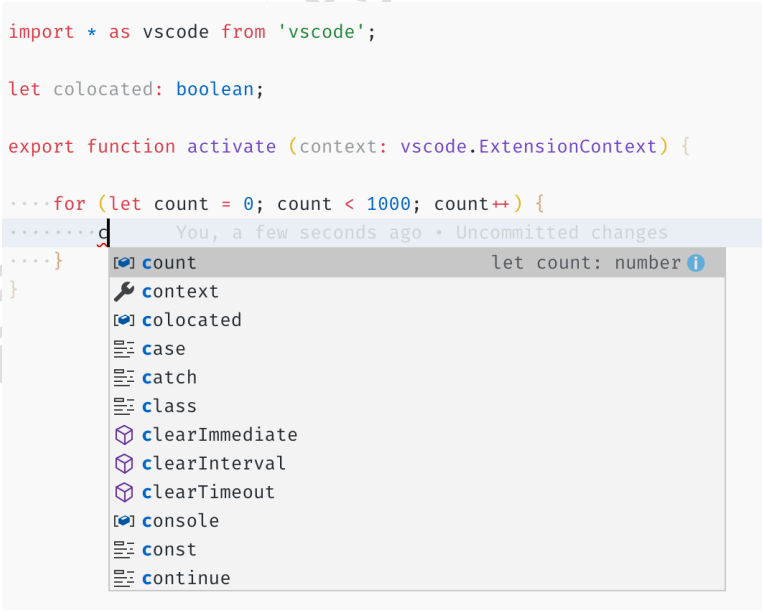


图 12 智能补全提示

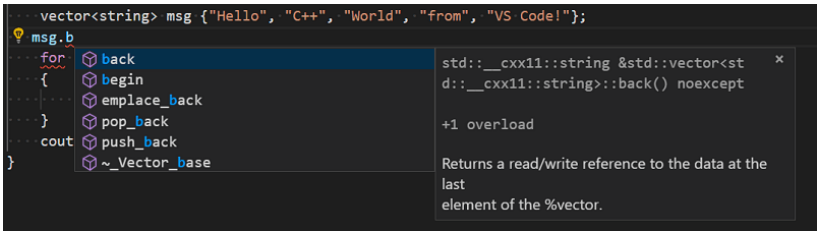


图 13 智能补全提示 2

4.6语法高亮

功能介绍：支持多语言语法高亮。

操作步骤：

语法高亮功能默认启动。

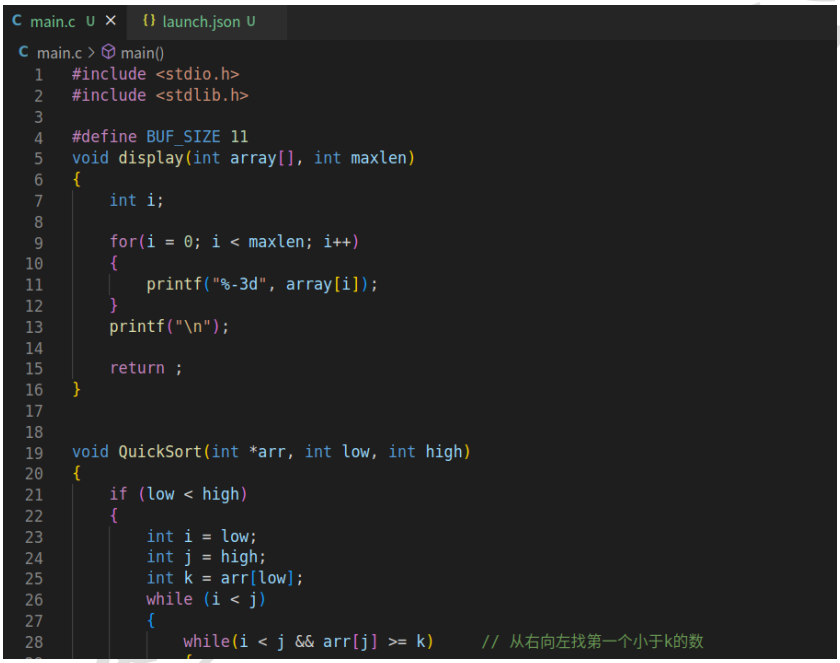


图 14 语法高亮

4.7格式化文档

功能介绍：提供源代码格式化功能。

操作步骤：

可以使用【Shift+Alt+F】格式化当前文件，也可以在右键单击上下文菜单中使用格式选择（Ctrl + K Ctrl + F）仅设置当前所选内容的格式。

4.8转到定义

功能介绍：鼠标悬停在符号上，将显示声明的预览，并根据字符转到定义处。

操作步骤：

可以使用 Ctrl+单击跳转到定义，也可以使用 Ctrl+Alt+单击将定义打开到一侧。

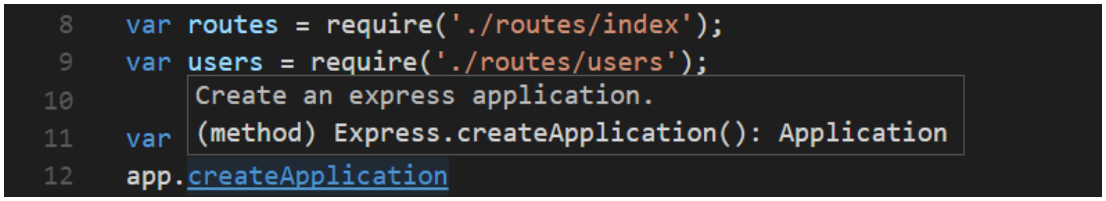


图 15 转到定义

4.9编译配置

功能介绍：IDE 提供编译配置功能进行编译任务配置。

操作步骤：

以 C/C++编译配置为例：

1、创建一个 c 语言代码文件，文件名为 main.c。

示例代码：

```
#include<stdio.h>

int main(){

    int a=4;

    printf("Hello World!\n");

}
```

2、点击顶部【终端】下的【配置任务】，进行编译任务配置。

图 16 配置新任务

3、在 tasks 中配置名为【build】的编译任务。其中 label 为任务名称，type 为任务类型，command 与 args 为编译命令与编译参数。

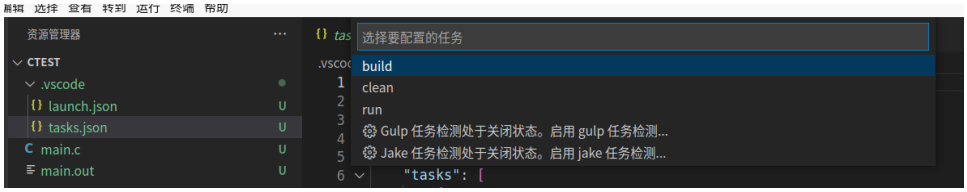


图 17 选择要配置的任务

```
{
    "version": "2.0.0",
    "options": {
        "cwd": "${workspaceFolder}"
    },
    "tasks": [
        {
            "label": "build",
            "type": "shell",
            "linux": {
                "command": "/usr/bin/gcc",
                "args": [
                    "-O0",
                    "-g3",
                    "-Wall",
                    "*.c",
                    "-o",
                    "main.out"
                ]
            }
        },
        {
            "label": "run",
            "type": "shell",
            "linux": {
                "command": "${workspaceFolder}/main.out",
                "args": []
            },
            "dependsOn": "build"
        },
        {
            "label": "clean",
            "type": "shell",
            "linux": {
                "command": "rm",
                "args": [
                    "main.out"
                ]
            }
        }
    ]
}
```

图 18 配置编译任务

4、配置完成后，通过【终端】下的【运行任务】，选择刚配置的【build】任务，即可进行编译。配置完成后的任务也可以在其他任务中通过任务名调用。

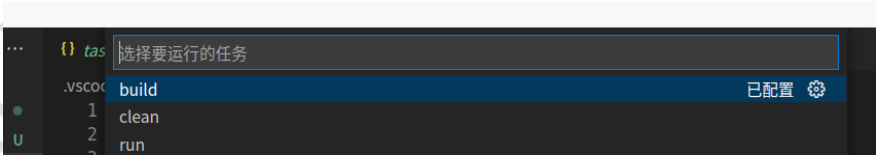


图 19 运行编译任务

4.10调试配置

功能介绍：IDE 提供调试配置功能，配置完成后实现一键调试。

操作步骤：

以 4.9 章节 main.c 程序为例

1、点击左侧【运行与调试】或使用【ctrl+shift+D】打开调试页面。

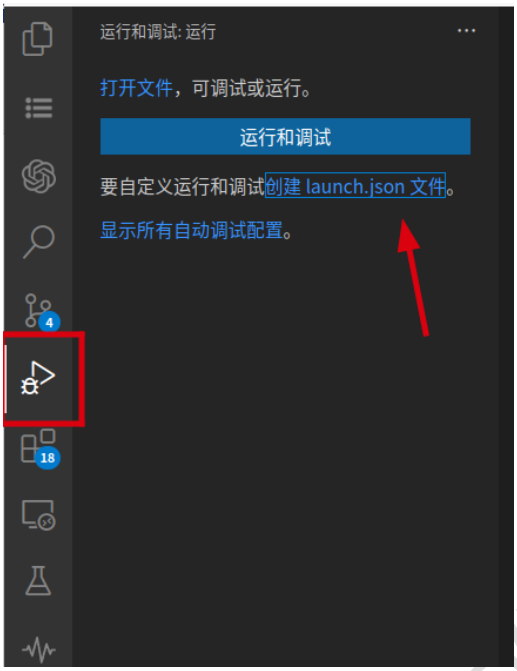


图 20 打开调试界面

2、点击【创建 launch.json】按钮，选择调试器类型，创建调试配置文件。

图 21 选择调试器

3、在配置文件中，设置调试属性。

name：配置名称。

type：配置类型。

request：请求配置类型。

target：编译后的可执行文件路径。

cwd：项目根目录。

preLaunchTask：调试回话开始前要运行的任务。（可以将 4.9 章节的编译任务配置进来，表示调试进行前，先执行编译任务。）

gdbpath：配置调试器的路径。

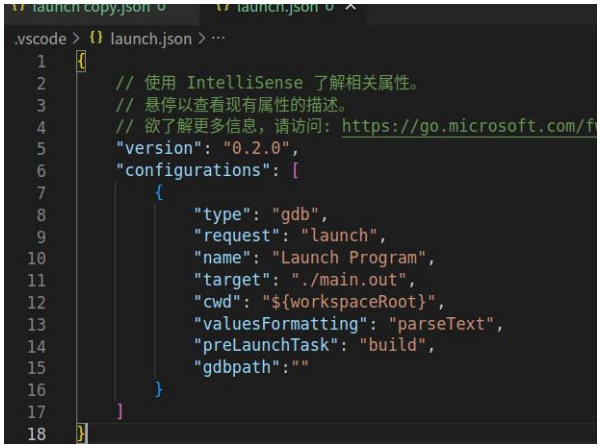


图 22 配置调试内容

4、配置完成后，在源代码左侧可通过鼠标点击，增加断点。

图 23 增加断点

5、之后点击调试按钮可进行调试。

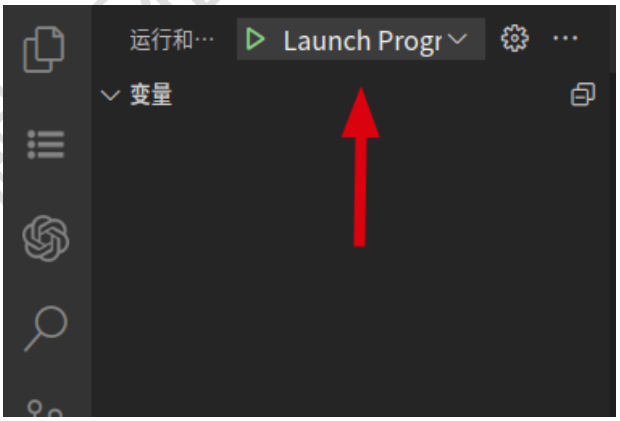


图 24 开始调试

6、进入断点停止后，可以查看断点信息，可以通过调试控制台控制调试步骤。

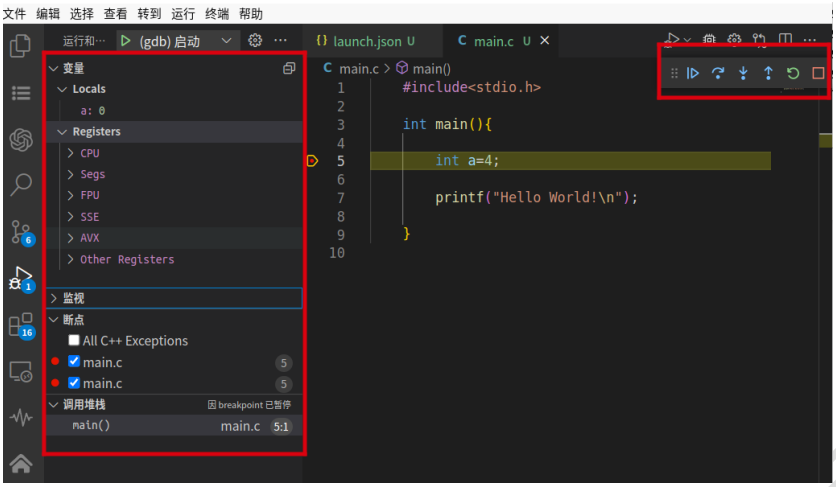


图 25 调试结果显示

4.11 代码版本管理

软件支持基于 git 的代码版本管理。

5 图形化智能调试及诊断

5.1反向调试

功能介绍：在调试过程中，有时跳过了关键节点，希望反向查看这些节点；为了避免重启调试，支持反向回看所有停顿的节点及其所有调试信息（线程、调用栈、变量、寄存器）。

操作步骤：

1、打开调试代码。

打开一个程序代码（可通过点击页面顶部右上角 打开示例程序 按钮，快速打开一个示例程序），之后打上断点，左侧选择 Native Debug 插件进行调试。

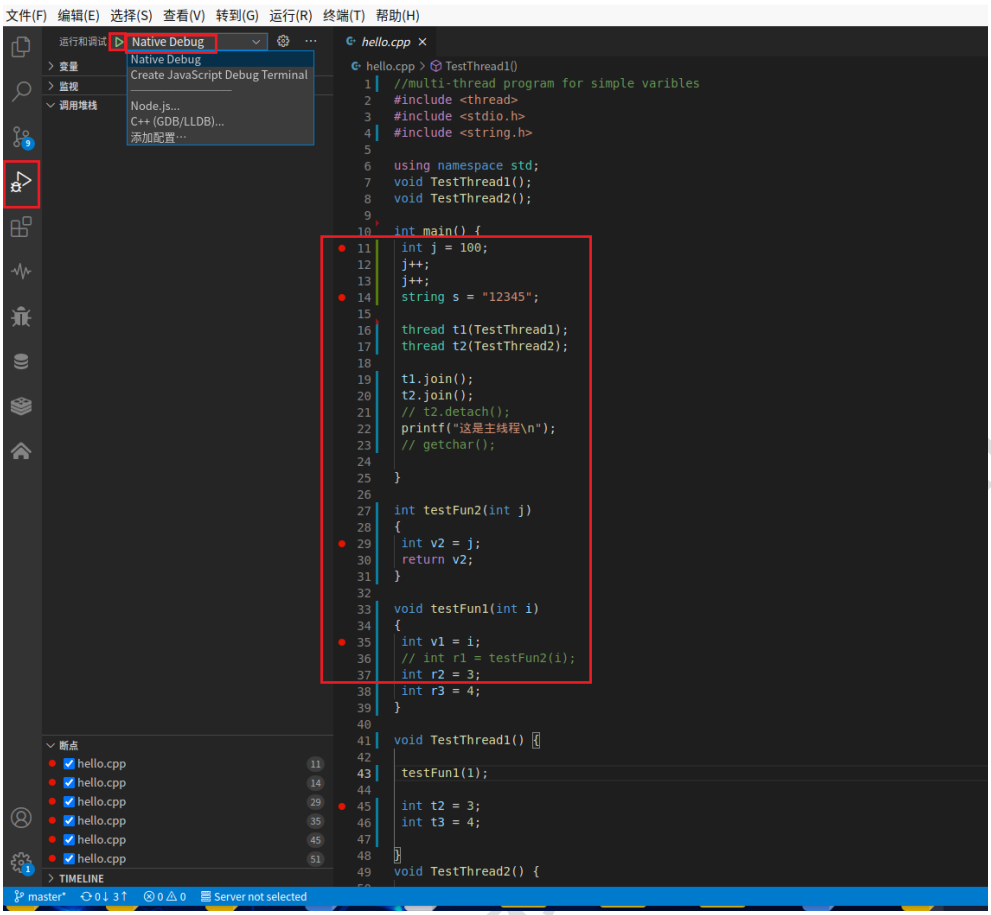


图 26 反向调试进入方式

2、使用反向调试按钮进行反向调试

开始调试后，调试控制栏右侧新增四个按钮，分别为【(反向调试)第一条】、【(反向调试)上一条】、【(反向调试)下一条】、【(反向调试)最后一条】。

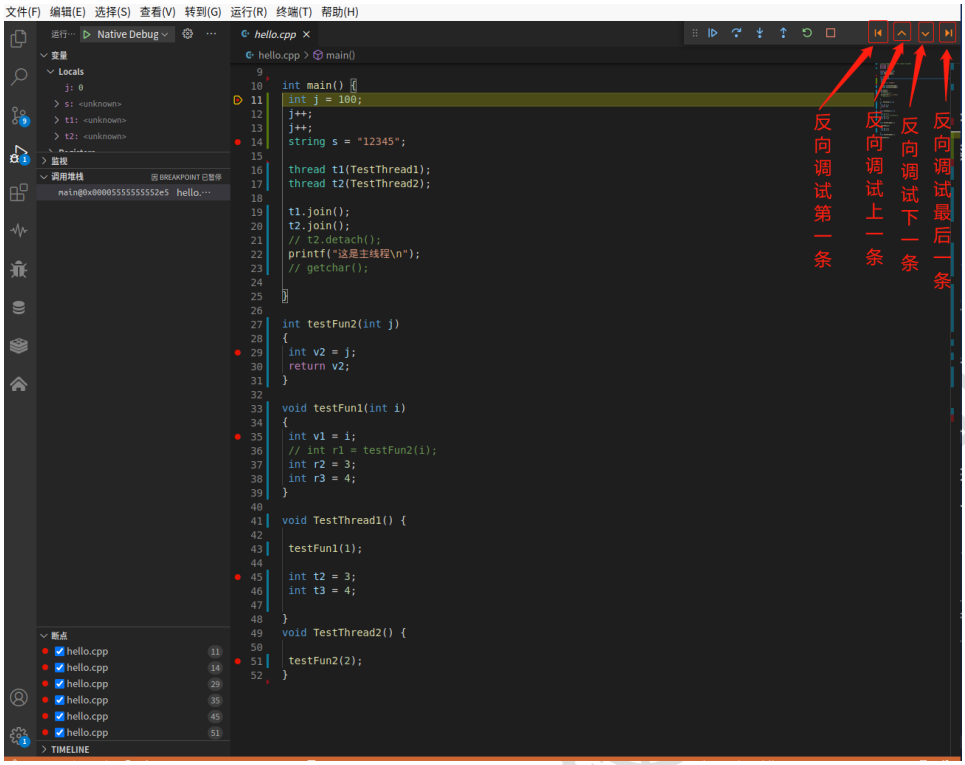


图 27 反向调试控制

在进行正向调试时，可以通过点击反向调试的四个按钮，来进行代码的反向调试。反向调试的高亮为暗红色，左侧可查看变量等信息。

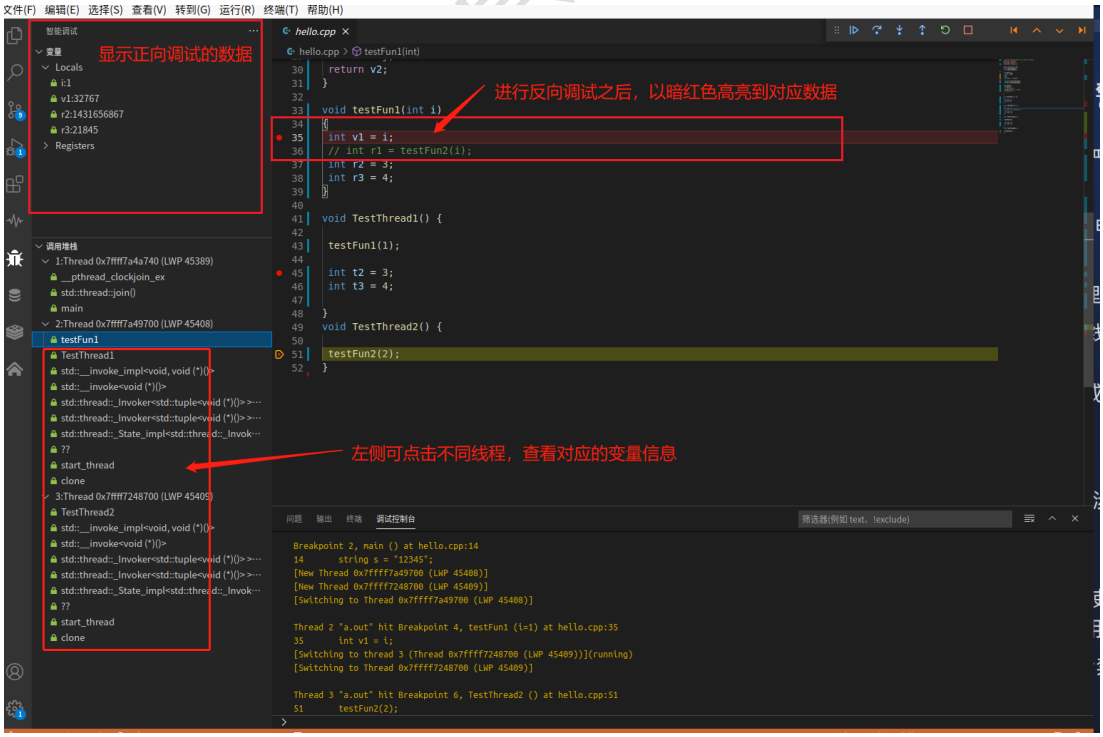


图 28 反向调试信息展示

5.2堆栈历史调试

功能介绍：在程序调试过程中记录调试和程序堆栈信息，并在调试结束后支持以图形界面的方式重新播放，或查询分析记录的程序堆栈和调试信息。

与【反向调试】的差别有两点：

(1)一是该功能需要等待调试结束后，而反向调试是在调试结束前进行。

(2)二是该功能支持多种复杂查询，而反向调试不支持查询，仅支持单步回看。

操作步骤：

1、打开调试代码。

打开一个程序代码（可通过点击页面顶部右上角 打开示例程序 按钮，快速打开一个示例程序），之后打上断点，左侧选择 Native Debug 插件进行调试。

2、进入历史调试页面。

调试结束后，会弹出调试历史页面，页面中展示刚刚调试的历史数据，默认选中为第一条。

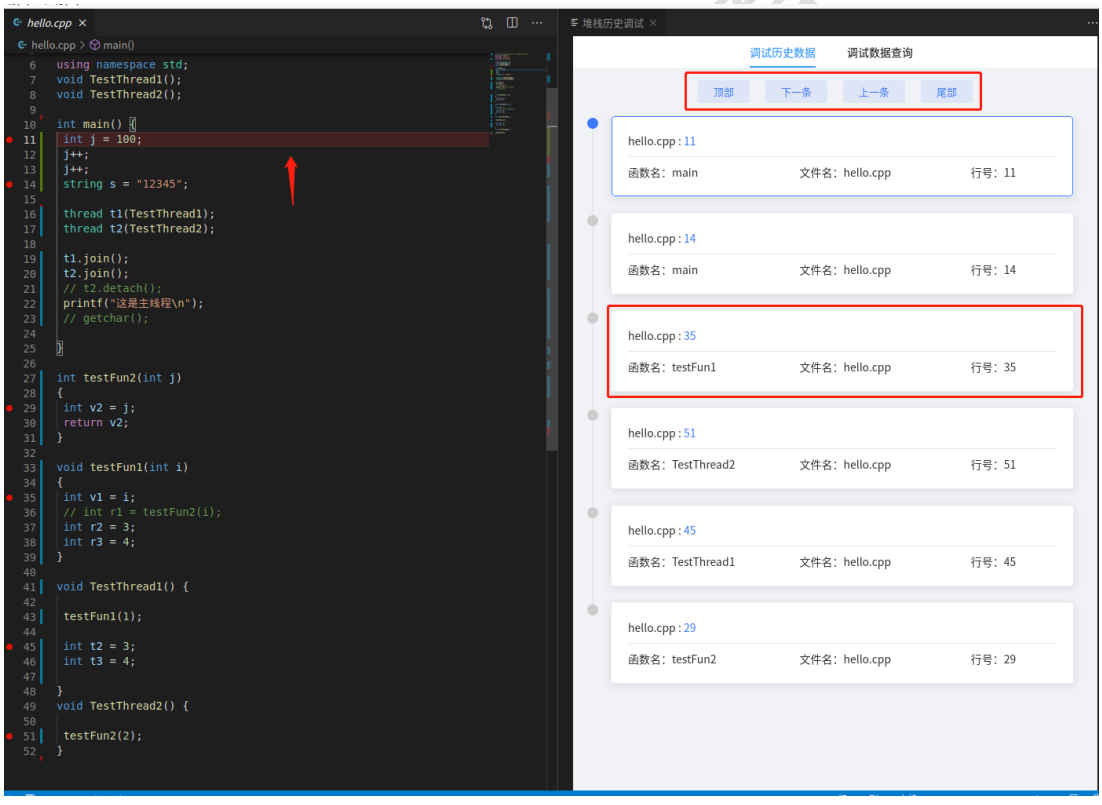


图 29 历史调试页面

(1)页面中提供四个按钮，分别为【顶部】、【下一条】、【上一条】、【尾部】。点击对应按钮，移动当前选中位置，可进行历史调试数据的查看。

(2)也可以直接点击某条数据，进行查看。

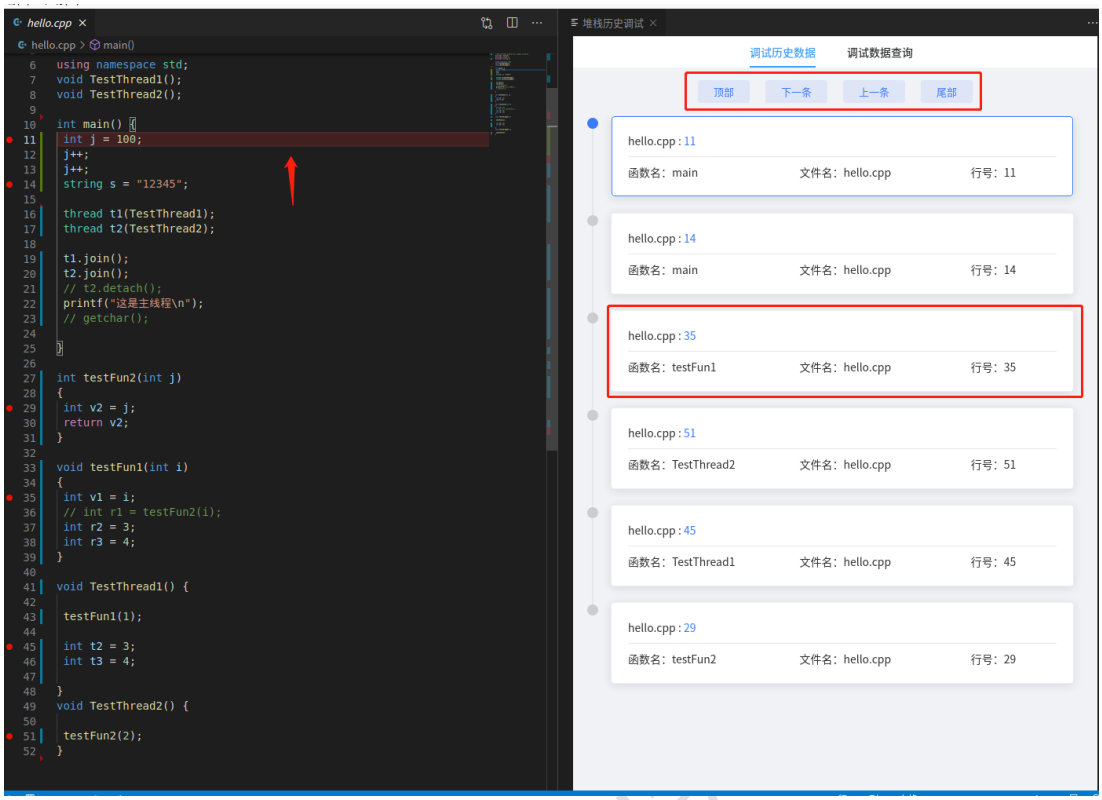


图 30 历史调试控制及信息

(3) 打开【调试数据查询】页面，可进行调试数据的查询。

(4) 【调试数据查询】页面，可通过选中上方【函数名】、【参数】、【行号】、【变量】、【寄存器】选项，进行筛选。筛选之后，列表中出现符合筛选条件的数据。

(5) 【调试数据查询】页面，可通过勾选列表中的数据后，点击【查询】按钮，进行查询。查询后，【调试历史数据】页面显示出查询后的数据。

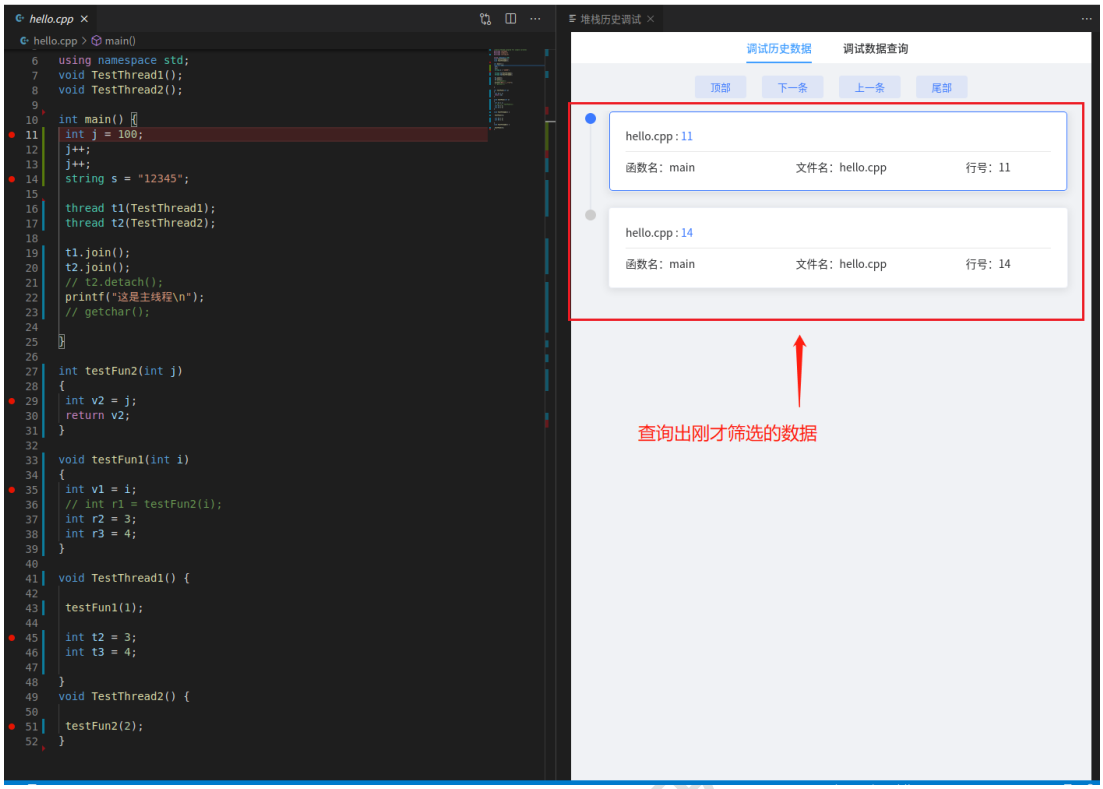


图 31 历史调数据展示

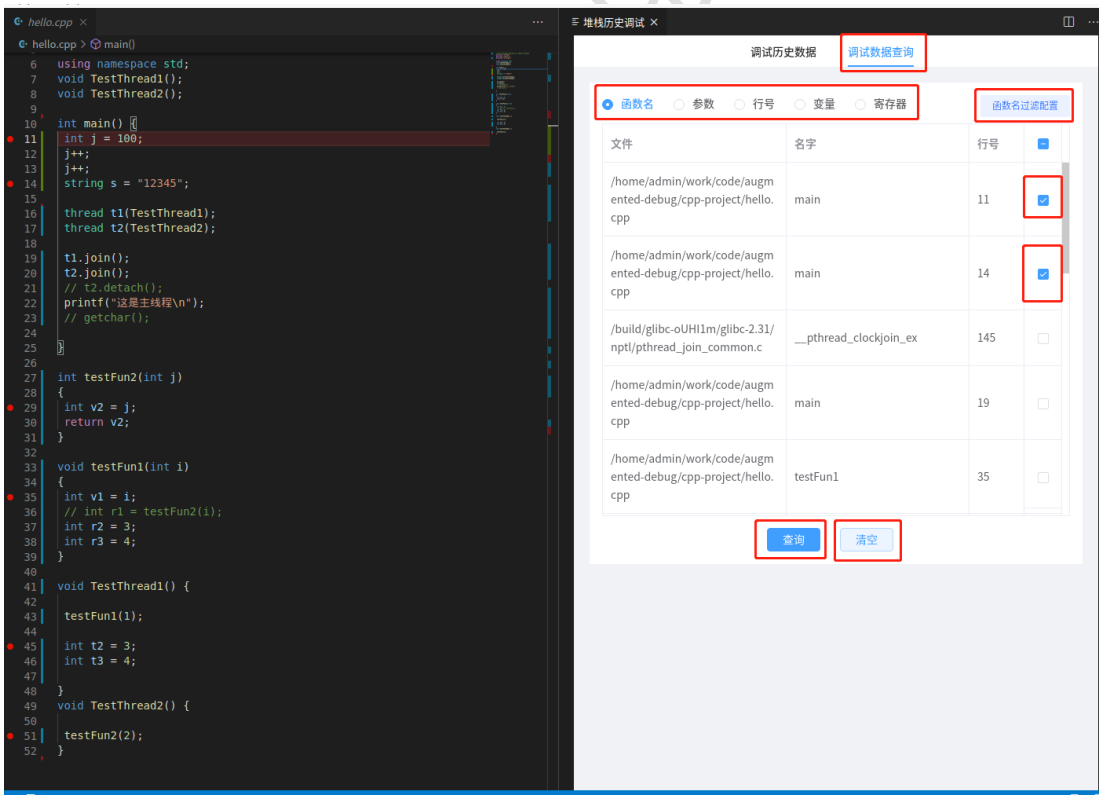


图 32 历史调试数据筛选

(6) 【调试数据查询】页面，根据【函数名】筛选时，可点击【函数名过滤配置】按钮，来设置【函数名过滤的规则】。其他选项有相应按钮与此类似。

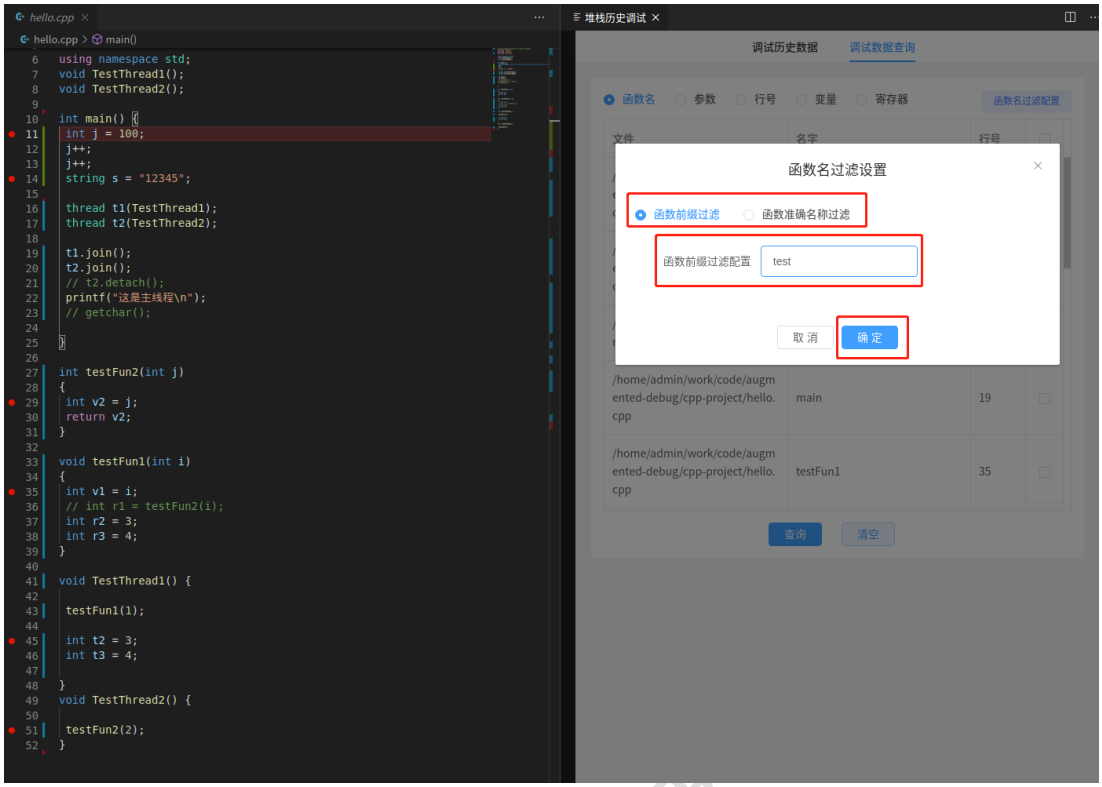


图 33 调试数据查询

(7) 过滤后，【调试数据查询】页面展示出符合过滤规则的数据。

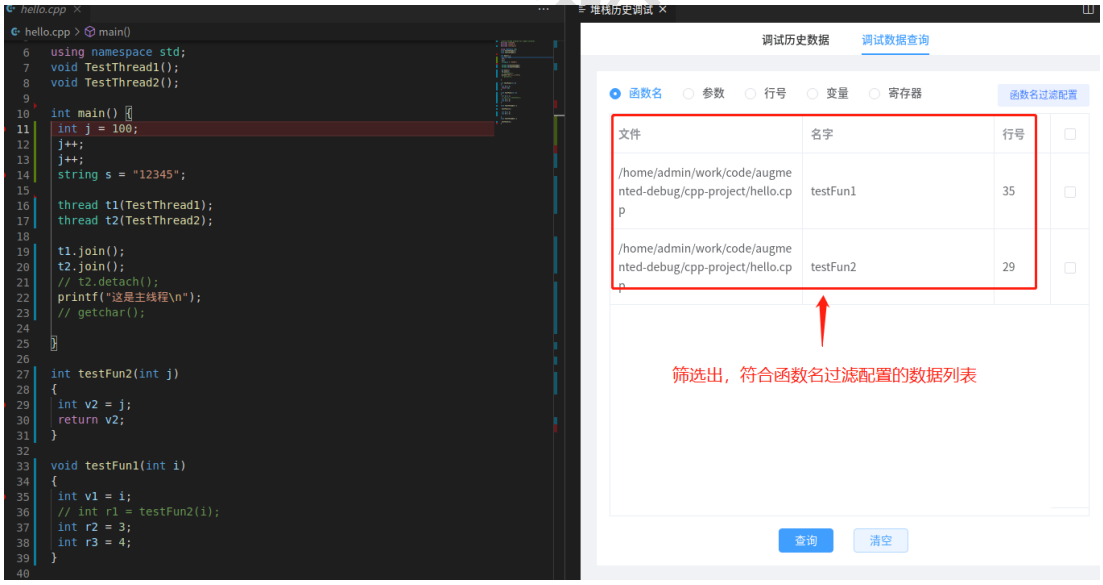


图 34 调试数据过滤

5.3coredump 文件分析

功能介绍：程序运行错误后，分析 coredump 文件找到错误所在。

操作步骤：（以下面示例程序为例）

1、新建示例程序 dumpTest.c。

dumpTest.c 示例代码

```
#include "stdio.h"
#include "stdlib.h"
void dumpCrash()
{
	char *pStr = "test_content";
	free(pStr);
}
int main()
{
	dumpCrash();
	return 0;
}
```

2、打开 core 文件设置。

运行命令：ulimit -c unlimited

3、编译示例程序。

生成名为 dumpDebug 的可执行程序。

运行编译命令：gcc -g -o dumpDebug dumpTest.c

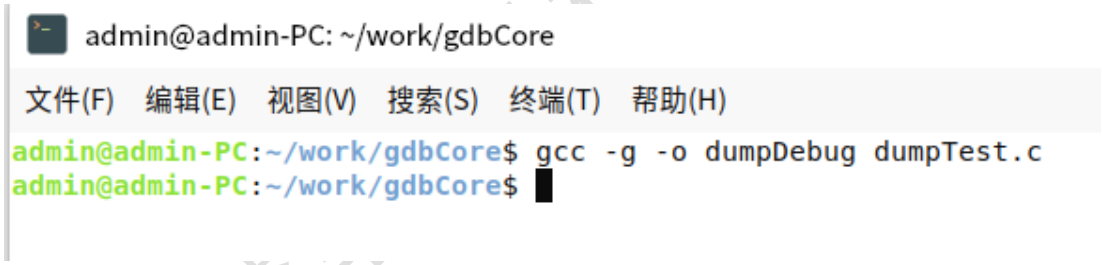


图 35 编译示例程序

4、可执行程序执行，程序错误并生成 core 文件。

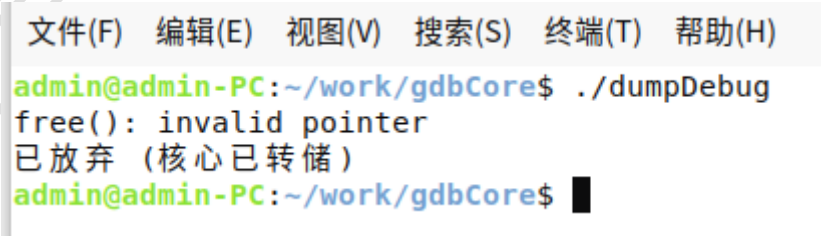


图 36 执行示例程序

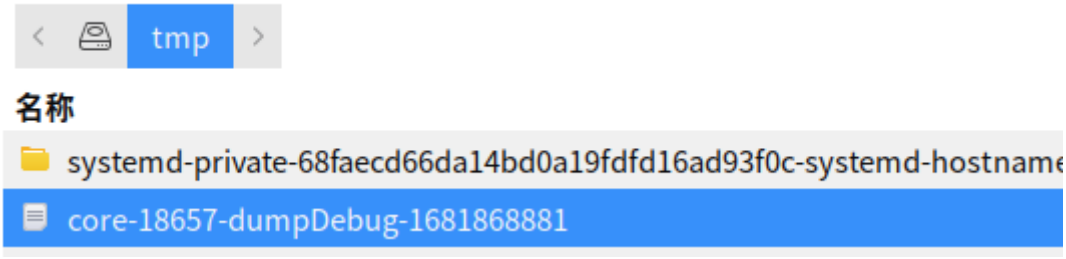


图 37 生成的 core 文件

5、打开gdbCore功能。

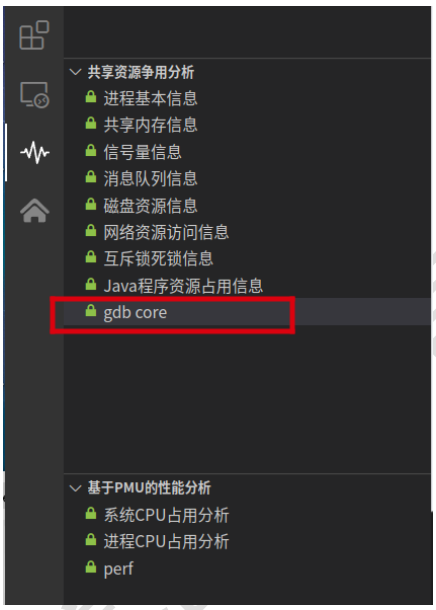


图 38 gdbCore 功能打开

6、填入可执行文件与Core文件，点击确定。

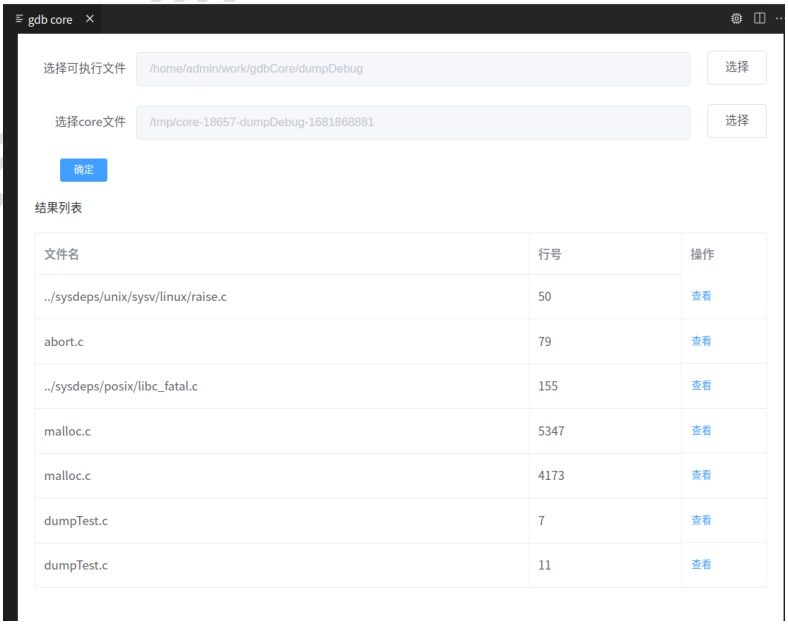


图 39 填入对应文件进行分析

7、点击查看，可转到对应代码行。

图 40 查看代码行

5.4 互斥锁死锁信息

进入方法：打开侧边栏通过侧边栏【互斥锁死锁信息】进入。

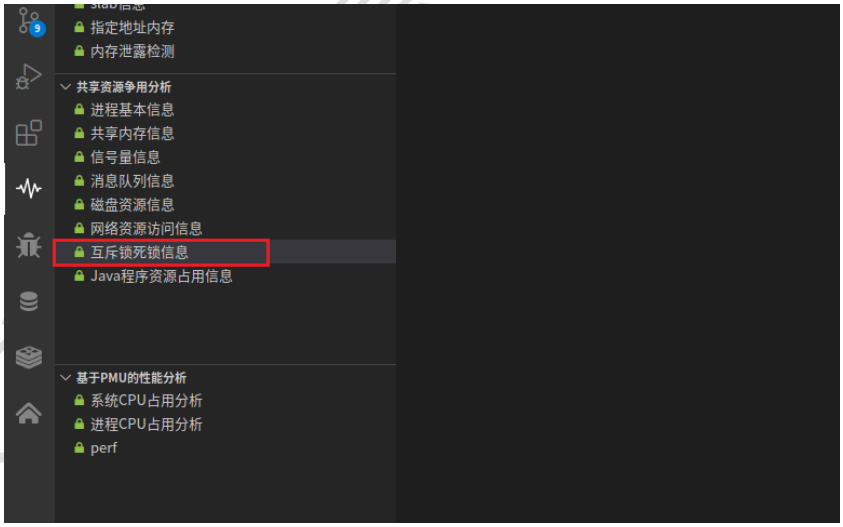


图 41 互斥锁死锁信息功能进入

5.4.1 C/C++语言程序互斥锁、死锁检测信息

功能介绍：进行 C/C++语言程序的死锁检测、互斥锁检测，显示死锁检测信息、互斥锁信息。

操作步骤：

1、开启一个 C/C++语言互斥锁/死锁进程。

以下面的测试代码为例。

(1)将该测试代码放入名为 sample_deadlock.cpp 的文件中。

测试代码（文件名：sample_deadlock.cpp）。

```
1  #include<thread>
2  #include <mutex>
3  #include <iostream>
4
5  using std::cout;
6
7  std::mutex mutex1;
8  std::mutex mutex2;
9  std::mutex mutex3;
10
11          void FuncA() {
12              std::lock_guard<std::mutex> guard1(mutex1);
13              std::this_thread::sleep_for(std::chrono::seconds(1));
14              std::lock_guard<std::mutex> guard2(mutex2);
15              std::this_thread::sleep_for(std::chrono::seconds(1));
16          }
17
18          void FuncB() {
19              std::lock_guard<std::mutex> guard2(mutex2);
20              std::this_thread::sleep_for(std::chrono::seconds(1));
21              std::lock_guard<std::mutex> guard3(mutex3);
22              std::this_thread::sleep_for(std::chrono::seconds(1));
23          }
24
25          void FuncC() {
26              std::lock_guard<std::mutex> guard3(mutex3);
27              std::this_thread::sleep_for(std::chrono::seconds(1));
28              std::lock_guard<std::mutex> guard1(mutex1);
29              std::this_thread::sleep_for(std::chrono::seconds(1));
30          }
31
```

```
32      int main() {
33          std::thread A(FuncA);
34          std::thread B(FuncB);
35          std::thread C(FuncC);
36
37          std::this_thread::sleep_for(std::chrono::seconds(5));
38
39          if (A.joinable()) {
40              A.join();
41          }
42          if (B.joinable()) {
43              B.join();
44          }
45          if (C.joinable()) {
46              C.join();
47          }
48          cout << "hello\n";
49          return 0;
50      }
51
52
```

(2)打开该文件目录终端，执行“g++ sample_deadlock.cpp -lpthread”命令进行编译，编译后，获得一个名为“a.out”的文件。

图 42 C++示例程序编译

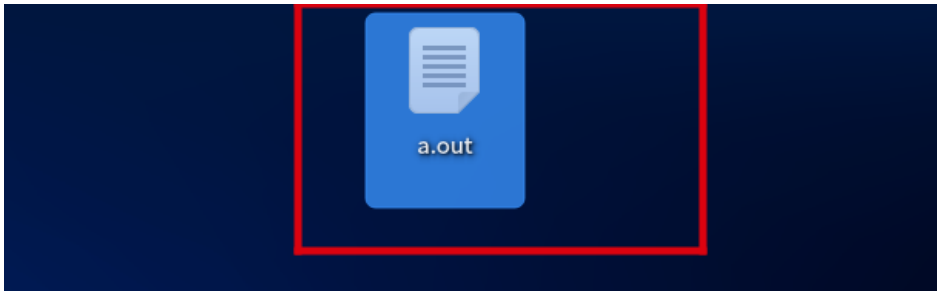


图 43 a.out 文件

(3)在终端接着执行“./a.out”命令，运行该示例程序。

图 44 运行示例程序

2、获取互斥锁信息。

在【C/C++语言程序互斥锁信息】页面下，点击【选择进程】按钮，选择一个 C/C++进程，或直接输入一个 C/C++进程的进程号 pid，然后点击【获取信息】按钮进行搜索。

图 45 互斥锁信息查询

搜索后，可查询出线程信息与互斥锁信息。

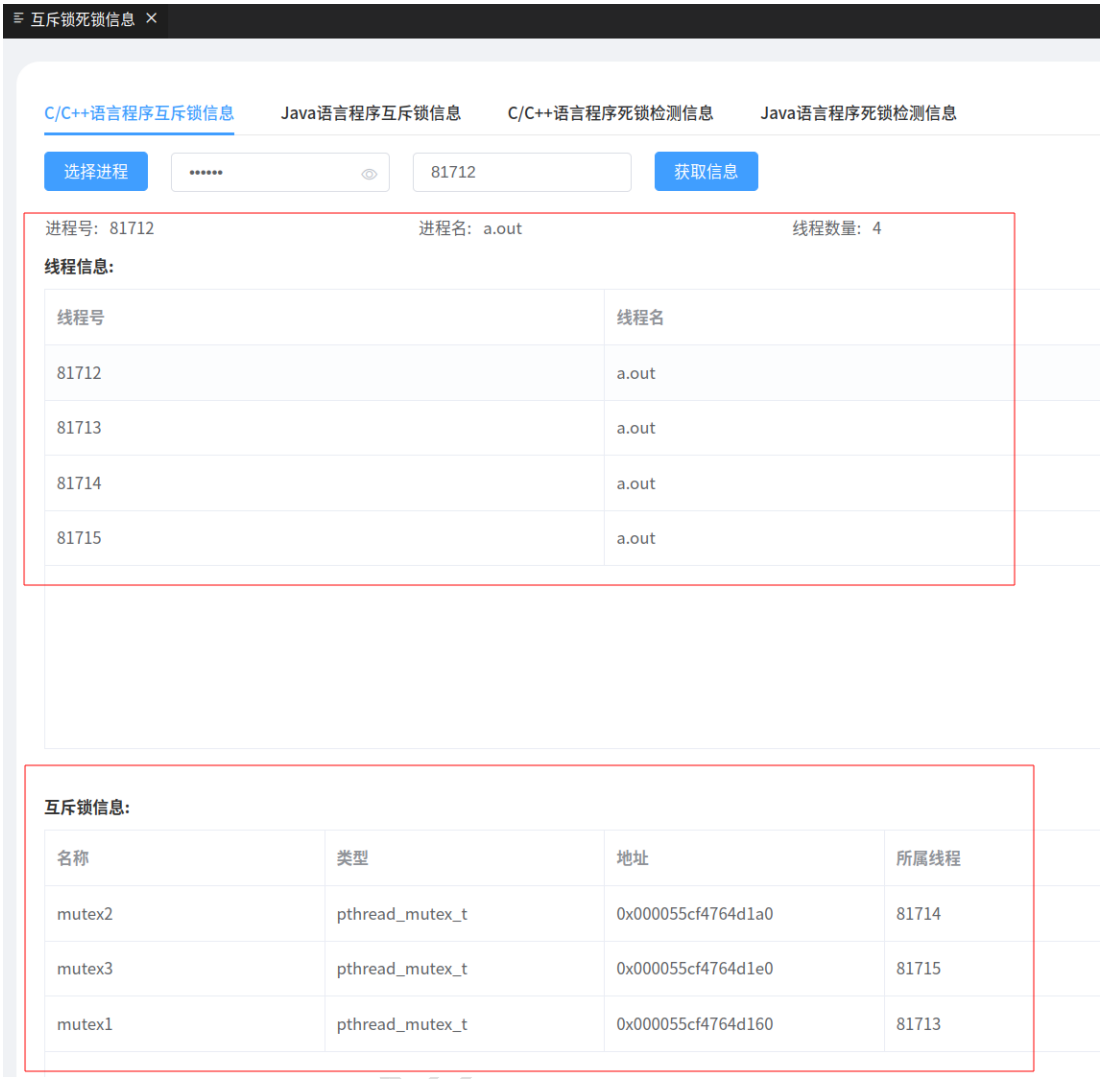


图 46 互斥锁信息查看

3、获取死锁信息。

在【C/C++语言程序死锁检测信息】页面下，点击【选择进程】按钮，选择一个 C/C++进程，或直接输入一个 C/C++进程的进程号 pid，然后点击【获取信息】按钮进行搜索。

图 47 C++示例程序死锁信息搜索

搜索后，可查询出死锁检测信息。

图 48 C++示例程序死锁信息查看

图 49 C++示例程序死锁信息查看 2

5.4.2 Java 语言程序互斥锁、死锁检测信息

功能介绍：进行 Java 语言程序的死锁检测、互斥锁检测，显示死锁检测信息、互斥锁信息。

操作步骤：

1、开启一个 Java 语言互斥锁/死锁进程。

以下面的测试代码为例。

(1)将该测试代码放入名为 DeadThread.java 的文件中。

测试代码（文件名：DeadThread.java）。

```
1
2
3  public class DeadThread implements Runnable {
4
5       public String username;
6       public Object lock1 = new Object();
7       public Object lock2 = new Object();
8
9       @Override
10              public void run() {
11                  // TODO Auto-generated method stub
12                  if (username.equals("a")) {
13                      synchronized (lock1) {
14                          try {
15                              System.out.println("username = " + username);
16                              System.out.println(Thread.currentThread().getName());
17                              Thread.sleep(3000);
18                          } catch (Exception e) {
19                              // TODO: handle exception
25                      }
26                  }
27                  if (username.equals("b")) {
28                      synchronized (lock2) {
29                          try {
30                              System.out.println("username = " + username);
```

```
31                                  System.out.println(Thread.currentThread().getName());
32                                  Thread.sleep(3000);
33
34                              } catch (Exception e) {
35                                  // TODO: handle exception
36                                  e.printStackTrace();
37                              }
38                              synchronized (lock1) {
39                                  System.out.println("按 lock2->lock1 顺序执行代码");
40                              }
41                          }
42
43                  }
44          }
45
46          public void setFlag(String username) {
47              this.username = username;
48          }
49
50          public static void main(String[] args) {
51
52              DeadThread dt1 = new DeadThread();
53              dt1.setFlag("a");
54              Thread t1 = new Thread(dt1);
55              t1.start();
56
57              try {
58                  Thread.sleep(2000);
59              } catch (InterruptedException e) {
60                  e.printStackTrace();
61              }
62
63              dt1.setFlag("b");
64              Thread t2 = new Thread(dt1);
65
```

```
66                  t2.start();
67              }
68          }
```

(2)打开该文件目录终端，执行“javac DeadThread.java”命令进行编译，编译后，获得一个名为“DeadThread.class”的文件。

图 50 java 示例程序编译

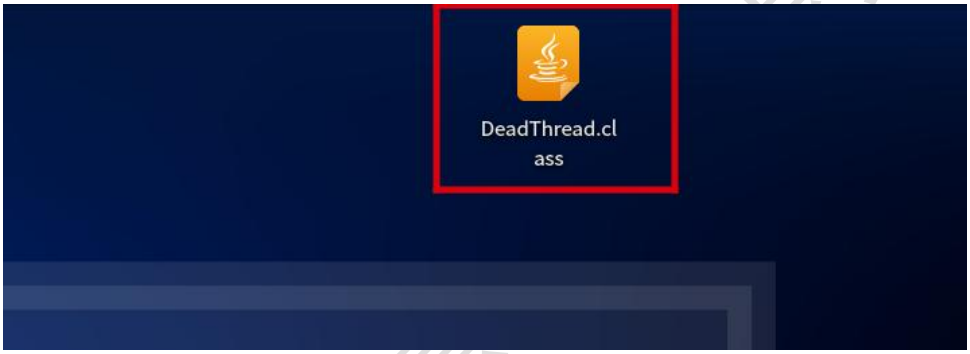


图 51 java 示例程序编译结果

(3)在终端接着执行“java DeadThread”命令，运行该示例程序。

图 52 java 示例程序运行

2、获取互斥锁信息。

在【Java 语言程序互斥锁信息】页面下，点击【选择进程】按钮，选择一个 Java 进程，或直接输入一个 Java 进程的进程号 pid，然后点击【获取信息】按钮进行搜索。

图 53 java 互斥锁信息搜索

搜索后，可查询出线程信息与互斥锁信息。

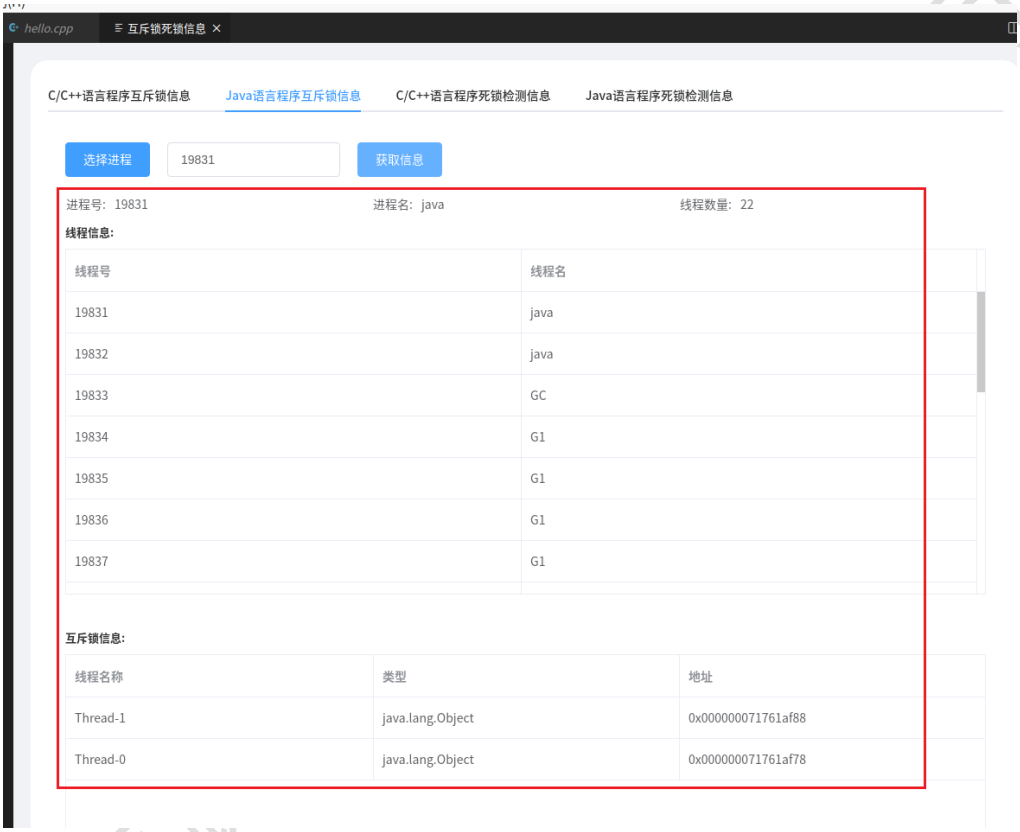


图 54 java 互斥锁信息展示

3、获取死锁信息。

在【Java 语言程序死锁检测信息】页面下，点击【选择进程】按钮，选择一个 Java 进程，或直接输入一个 Java 进程的进程号 pid，然后点击【获取信息】按钮进行搜索。

图 55 java 死锁信息搜索

搜索后，可查询出死锁检测信息。

图 56 java 死锁信息展示

5.5 内存泄露检测

功能介绍：对运行中的 C/C++代码进行内存泄露的监控。

操作步骤：

1、打开可执行文件。

(1)点击内存性能分析下的【内存泄露检测】按钮。

图 57 内存泄露检测功能进入

(2)点击选择可执行文件。

图 58 内存泄露检测功能---选择可执行文件

(3)打开文件管理器，选择编译好的 c 文件的可执行文件，点【open】按钮进行确定,可以对内存泄露等问题进行检测。

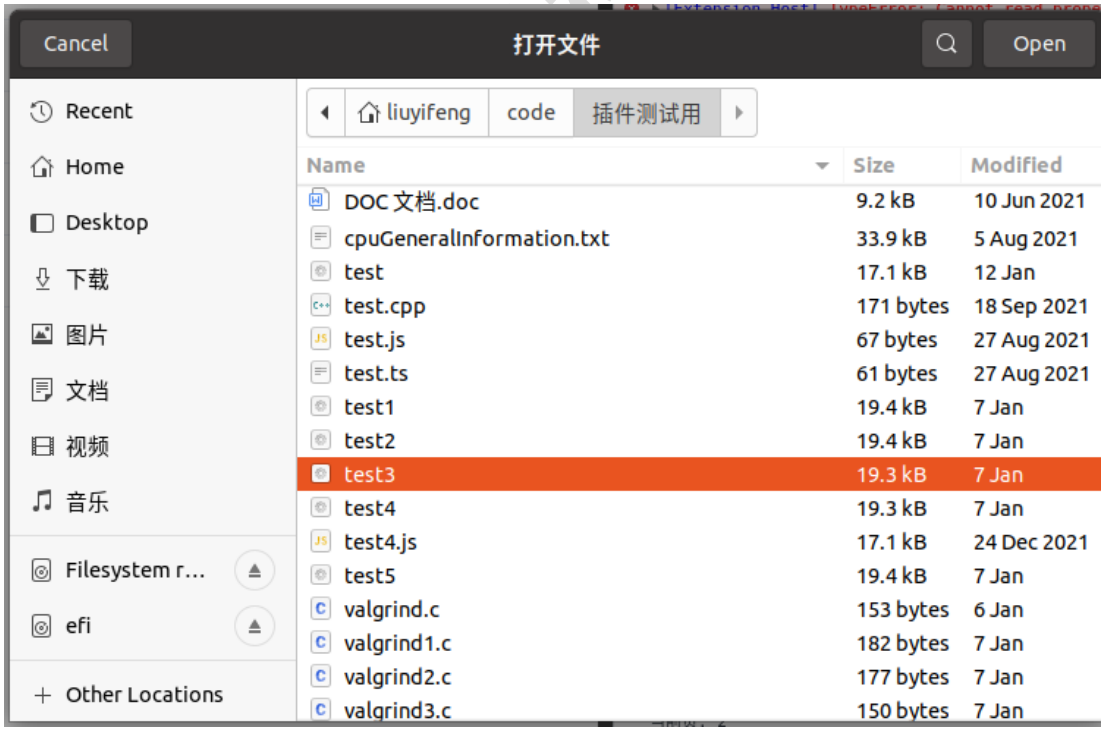


图 59 内存泄露检测功能---选择可执行文件

2、内存泄露检测测试代码及结果图

(1)准备工作

1)通过 bash 命令运行提供的 valgrindCode.run 文件

2)打开内存泄露插件

(2)数组越界/内存未释放

选择下拉列表数组越界，点击打开样例工程按钮。

会自动调用执行数组越界的可执行文件形成编译结果，并打开源码所在文件夹显示源码信息。

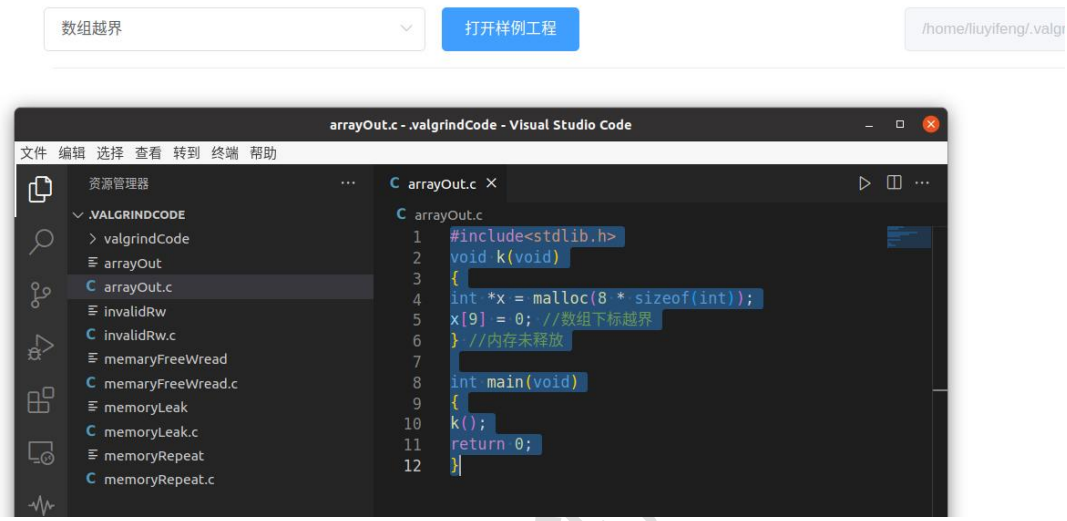


图 60 数组越界样例工程打开

1)示范代码

```
#include<stdlib.h>
void k(void)
{
    int *x = malloc(8 * sizeof(int));
    x[9] = 0; //数组下标越界
} //内存未释放
int main(void)
{
    k();
    return 0;
}
```

2)编译命令:gcc -Wall 原文件名 -g -o 别名

3)编译结果

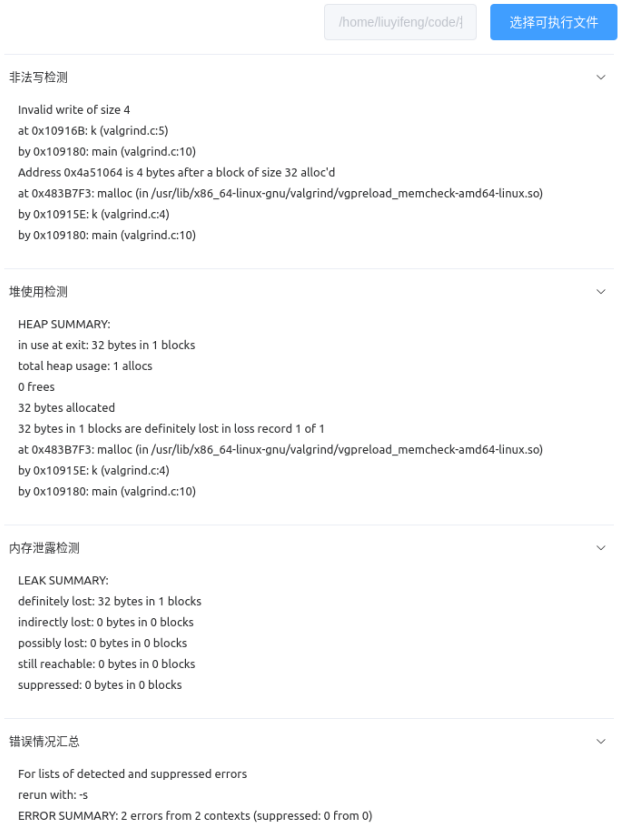


图 61 数组越界/内存未释放编译结果

(3)内存释放后读写

选择下拉列表内存释放后读写，点击打开样例工程按钮

会自动调用执行内存释放后读写的可执行文件形成编译结果，并打开源码所在文件夹显示源码信息。

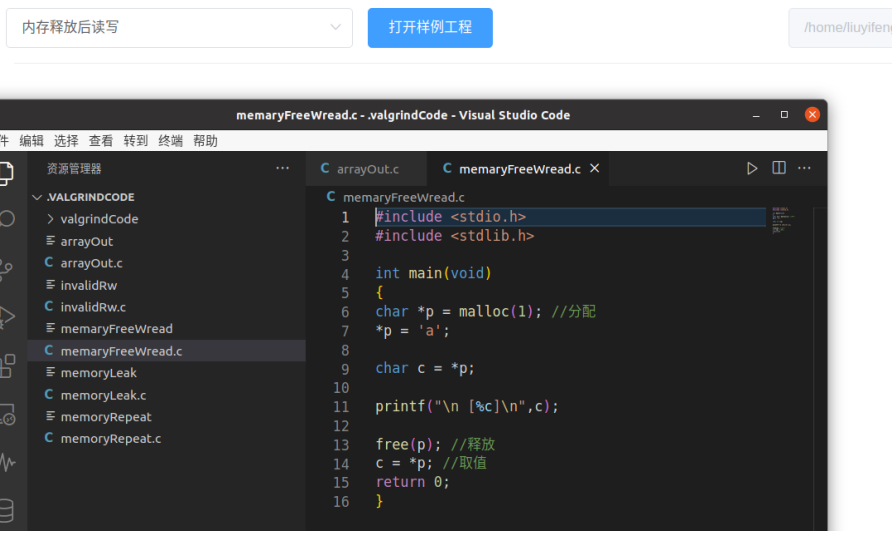


图 62 内存释放后读写示例代码打开

1)示范代码

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

```
#include <stdlib.h>
int main(void)
{
char *p = malloc(1); //分配
*p = 'a';
char c = *p;
printf("\n [%c]\n",c);
free(p); //释放
c = *p; //取值
return 0;
}
```

2)编译命令:gcc -Wall 原文件名 -g -o 别名

3)编译结果

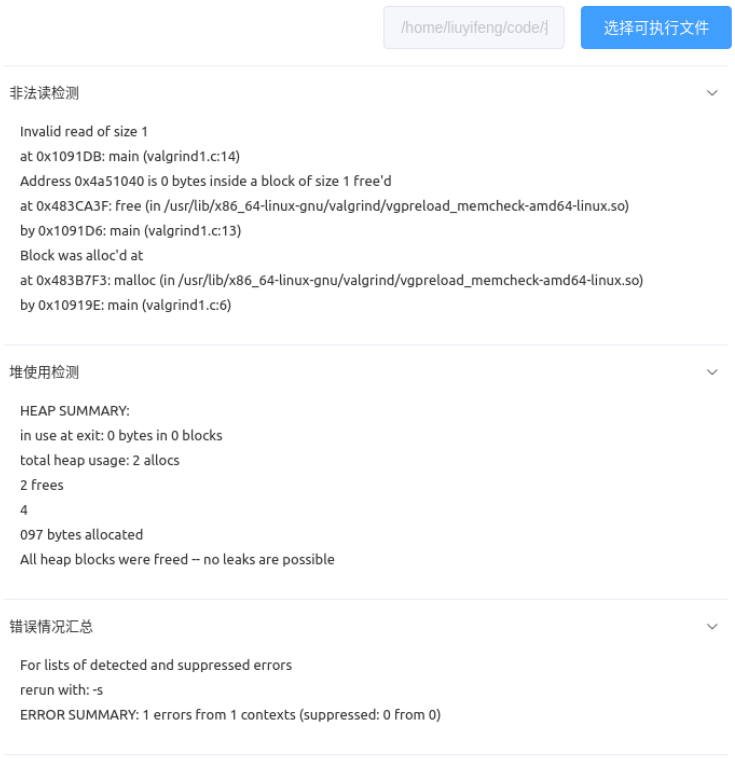


图 63 内存释放后读写编译结果

(4)无效读写

选择下拉列表无效读写，点击打开样例工程按钮。

会自动调用执行无效读写的可执行文件形成编译结果，并打开源码所在文件夹显示源码信息。

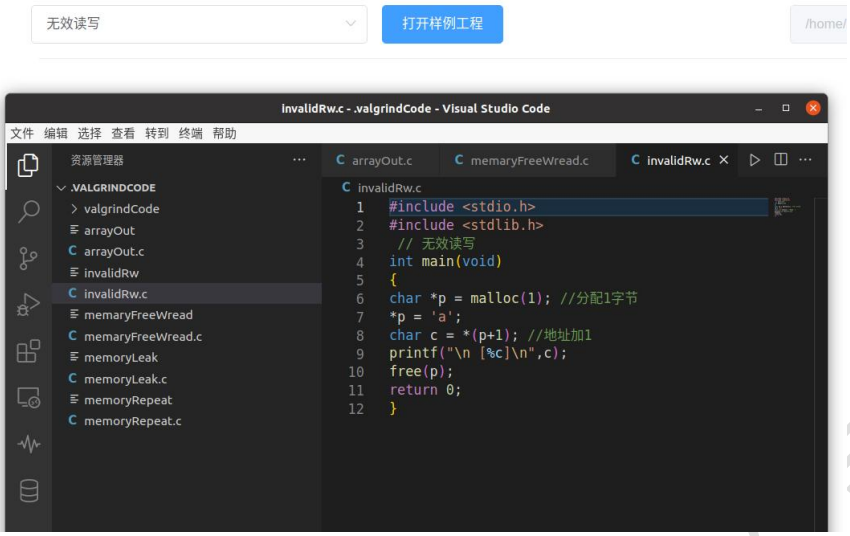


图 64 无效读写示例代码打开

1)示范代码

```
#include <stdio.h>
#include <stdlib.h>
 // 无效读写
int main(void)
{
char *p = malloc(1); //分配 1 字节
*p = 'a';
char c = *(p+1); //地址加 1
printf("\n [%c]\n",c);
free(p);
return 0;
}
```

2)编译命令:gcc -Wall 原文件名 -g -o 别名

3)编译结果

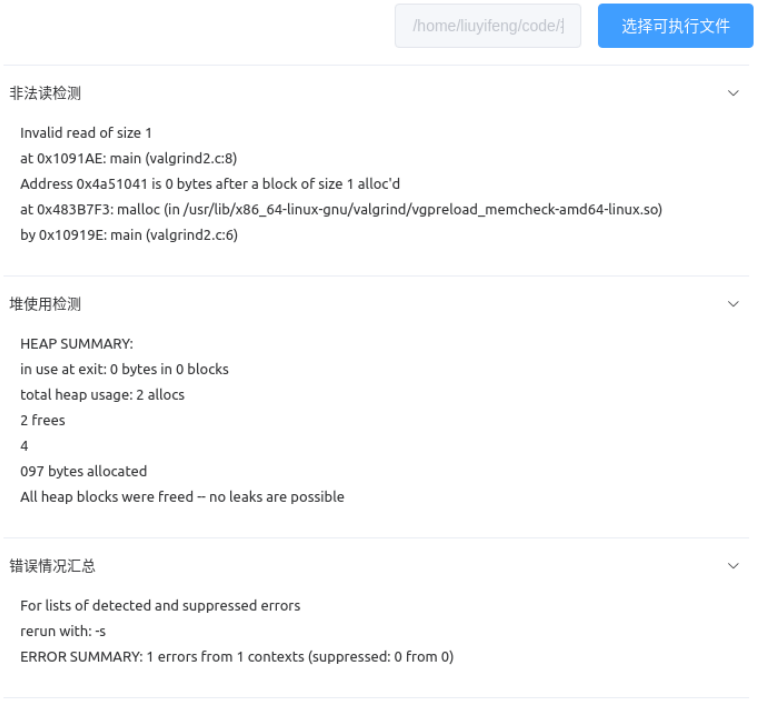


图 65 无效读写编译结果

(5)内存泄露

选择下拉列表内存泄露，点击打开样例工程按钮。

会自动调用执行内存泄露的可执行文件形成编译结果，并打开源码所在文件夹显示源码信息。

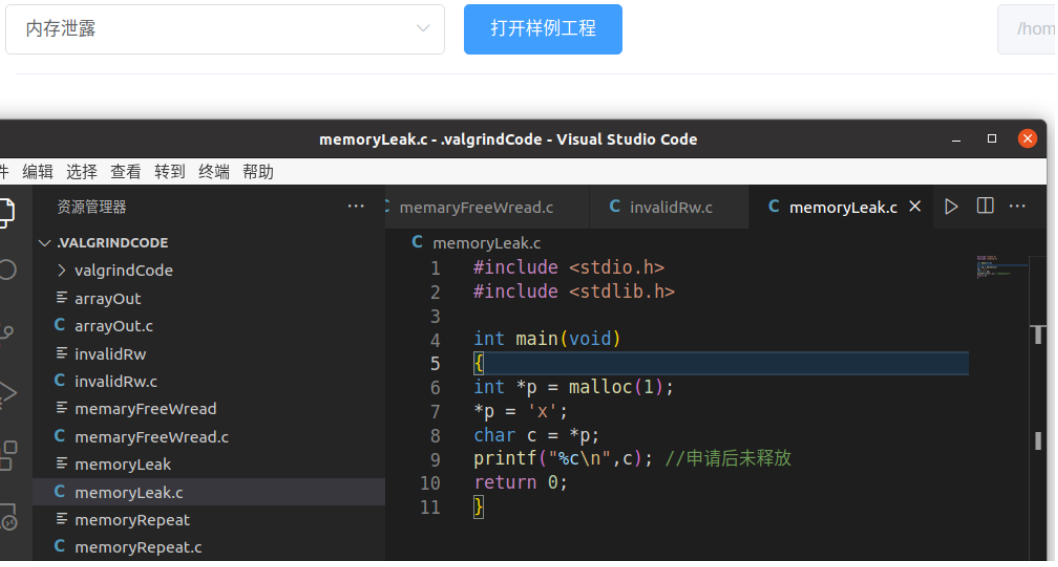


图 66 内存泄露示例代码打开

1)示范代码

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
```

```
{
int *p = malloc(1);
*p = 'x';
char c = *p;
printf("%c\n",c); //申请后未释放
return 0;
}
```

2)编译命令:gcc -Wall 原文件名 -g -o 别名

3)编译结果

图 67 内存泄露编译结果

(6)内存多次释放

选择下拉列表内存多次释放，点击打开样例工程按钮。

会自动调用执行内存多次释放的可执行文件形成编译结果，并打开源码所在文件夹显示源码信息。

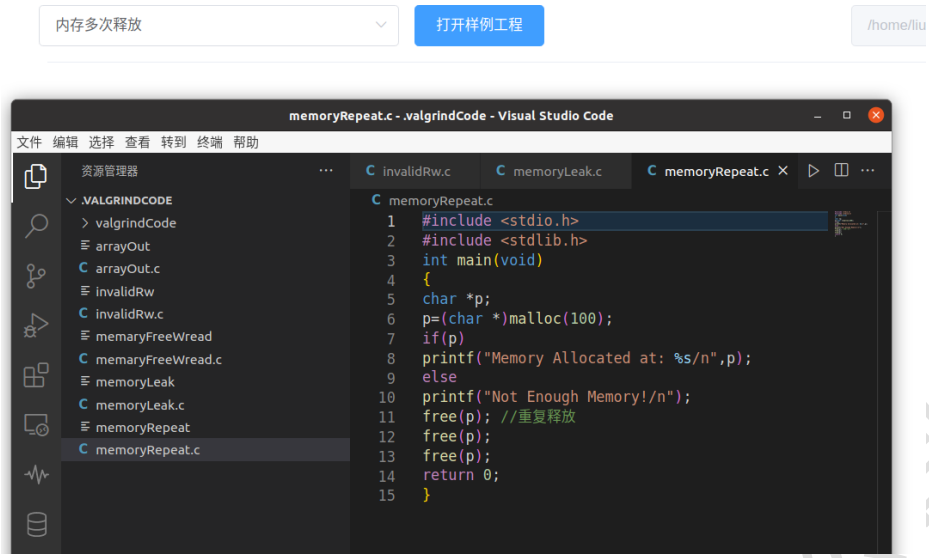


图 68 内存多次释放示例代码打开

1)示范代码

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
char *p;
p=(char *)malloc(100);
if(p)
printf("Memory Allocated at: %s/n",p);
else
printf("Not Enough Memory!/n");
free(p); //重复释放
free(p);
free(p);
return 0;
}
```

2)编译命令:gcc -Wall 原文件名 -g -o 别名

3)编译结果

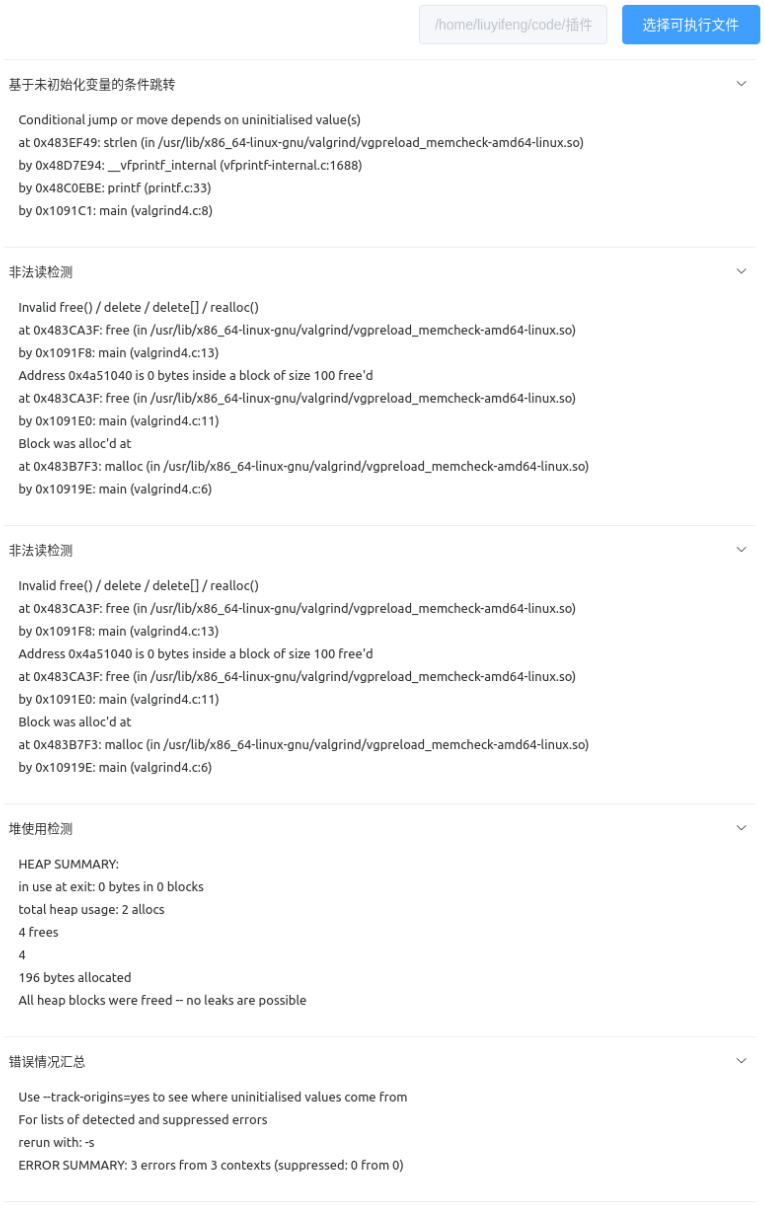


图 69 内存多次释放示例代码编译结果

3、用户打开自己的源码文件

(1)点击查看原文件按钮，在弹出的文件资源管理器中选择原文件所在文件夹，点击打开。

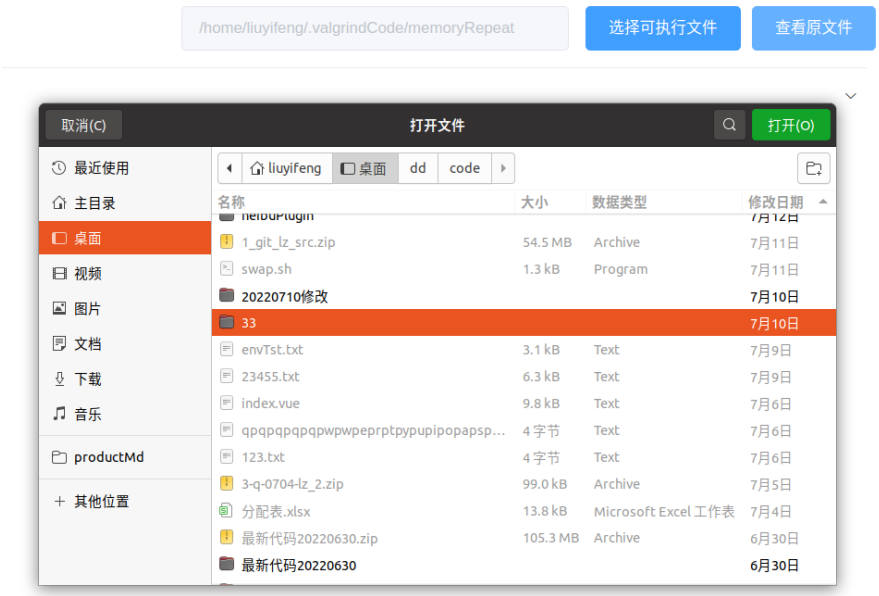


图 70 打开源码文件

(2)另起一个编辑器打开源码所在文件夹，帮助查看原文件信息。

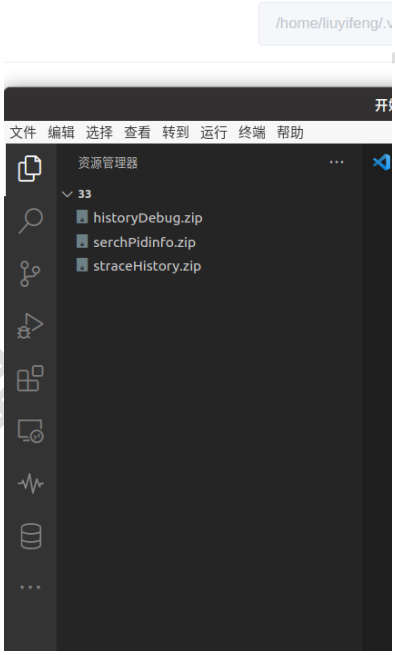


图 71 另起窗口打开文件夹

5.6 性能火焰图

功能介绍：对系统或进程相关的性能进行监控显示。支持监听 C/C++代码并生成火焰图。

操作步骤：

1、插件的进入。

选择基于 PMU 的性能分析下的【perf】选项，进入该插件。

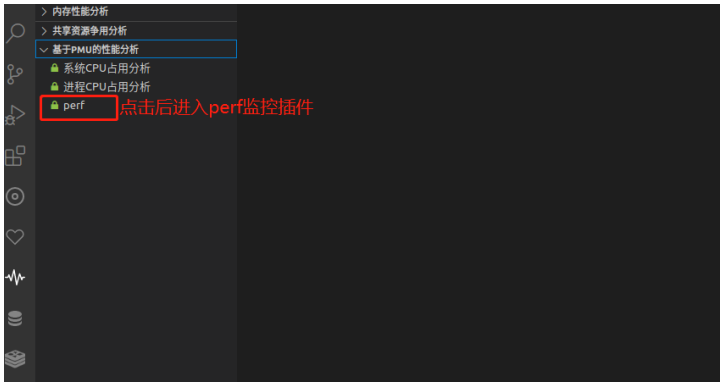


图 72 perf 监控功能进入

2、系统性能信息。

点击【系统性能信息】，出现加载页面，等待六秒后自动刷新显示系统性能信息。

系统性能信息　进程系统性能信息　svg生成页面

overHead	pid	comm	dso
1.81%	101522	code	code
1.81%	80276	vmx-vcpu-5	[kernel]
1.80%	3971	VizCompositorTh	code
1.54%	80273	vmx-vcpu-2	[kernel]
1.36%	80278	vmx-vcpu-7	[kernel]
0.84%	102231	code	code
0.82%	102295	perf	perf
0.67%	80277	vmx-vcpu-6	[kernel]
0.60%	89296	code	code
0.55%	80274	vmx-vcpu-3	[kernel]
0.48%	2173	2173:gnome-shell	gnome-shell
0.42%	2173	2173:gnome-shell	gnome-shell
0.41%	2019	Xorg	[kernel]

图 73 系统性能信息展示

3、查看进程号。

点击【查看进程号】按钮。

图 74 系统性能信息查询

4、进程号列表操作。

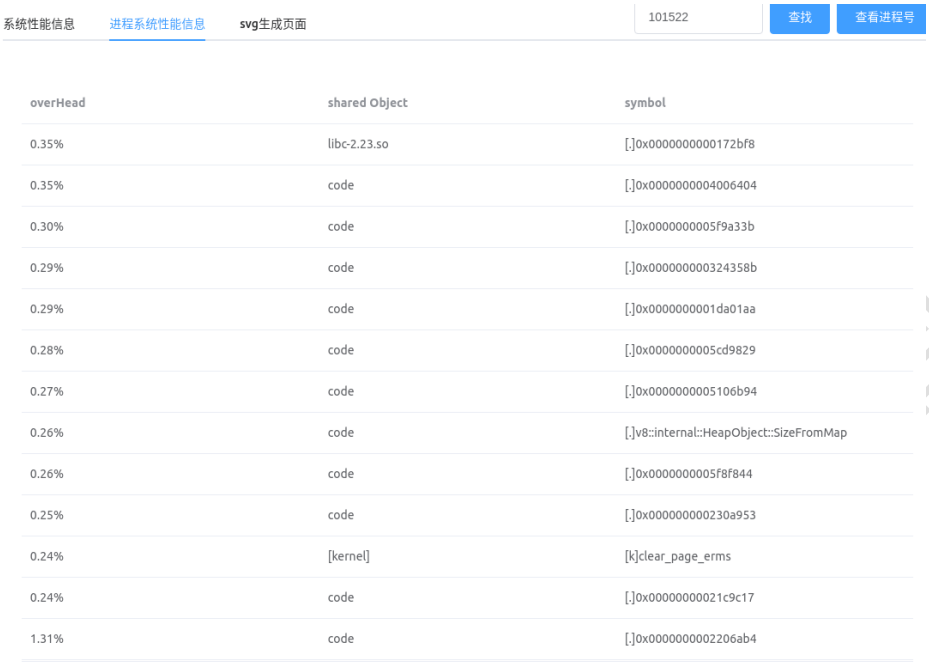
刷新按钮：点击【刷新】按钮，对进程号列表进行刷新。

搜索按钮：在文本框输入关键词，点击【搜索】按钮，模糊匹配命令中的单词。

选择按钮：点击【选择】按钮将进程号返回到页面。

5、进程号列表操作。

输入进程号，点击【确定】按钮后，等待六秒后自动刷新显示该进程性能信息。



overHead	shared Object	symbol
0.35%	libc-2.23.so	[.]0x0000000000172bf8
0.35%	code	[.]0x0000000004006404
0.30%	code	[.]0x0000000005f9a33b
0.29%	code	[.]0x000000000324358b
0.29%	code	[.]0x0000000001da01aa
0.28%	code	[.]0x0000000005cd9829
0.27%	code	[.]0x0000000005106b94
0.26%	code	[.]v8::internal::HeapObject::SizeFromMap
0.26%	code	[.]0x0000000005f8f844
0.25%	code	[.]0x000000000230a953
0.24%	[kernel]	[k]clear_page_erms
0.24%	code	[.]0x00000000021c9c17
1.31%	code	[.]0x0000000002206ab4

图 75 进程系统性能信息展示

6、程序性能火焰图。

火焰图生成测试代码。

```
1. #include <pthread.h>
2. func_d()
3. {
4.         int i;
5.         for(i=0;i<50000;i++);
6. }
7. func_a()
8. {svg 生成页面
9. //        int i;
10. //        for(i=0;i<100000;i++);
11.         func_d();
12. }
13. func_b()
14. {
15.         int i;
16.         for(i=0;i<200000;i++);
```

```
17. }
18. func_c()
19. {
20.         int i;
21.         for(i=0;i<300000;i++);
22. }
23. void* thread_fun(void* param)
24. {
25.         while(1) {
26.                 int i;
27.                 for(i=0;i<100000;i++);
28.                 func_a();
29.                 func_b();
30.                 func_c();
31.         }
32. }
33. int main(void)
34. {
35.         pthread_t tid1，tid2;
36.         int ret;
37.         ret=pthread_create(&tid1，NULL，thread_fun，NULL);
38.         if(ret==-1){
39.                 return -1;
40.         }
41.         ret=pthread_create(&tid2，NULL，thread_fun，NULL);
42.         if(ret==-1){
43.                 return -1;
44.         }
45.         if(pthread_join(tid1，NULL)!=0){
46.         }
47.         if(pthread_join(tid2，NULL)!=0){
48.         }
49.         return 0;
50. }
```

7、编译样例程序。

编译命令：gcc flame.c -pthread

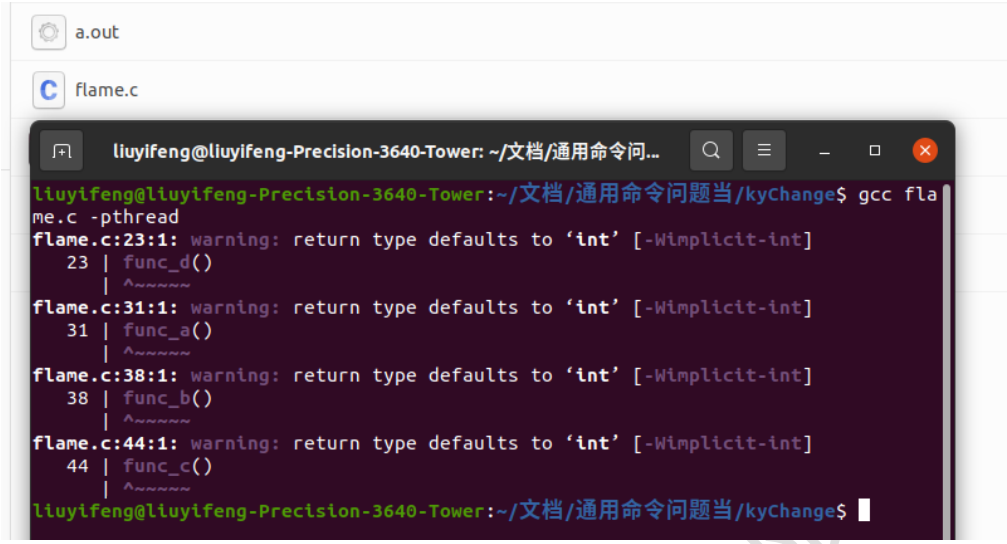


图 76 样例程序编译

8、火焰图获取中。

输入可执行程序的进程号，点击确定后，等待检测时间后显示火焰图(获取中)。

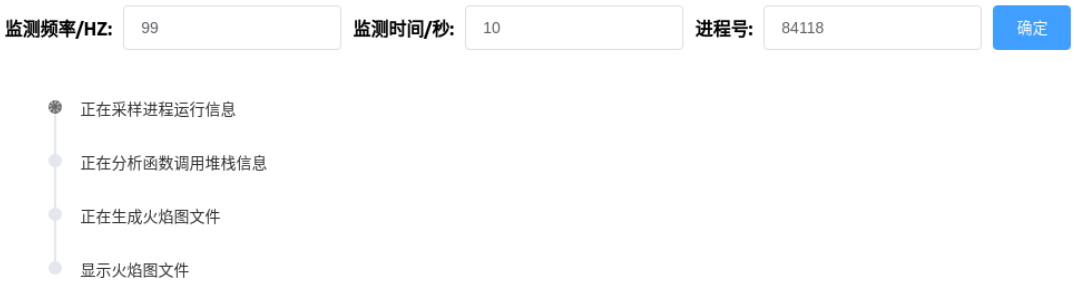


图 77 火焰图获取中

9、获取后页面样式。

图 78 火焰图获取进度

10、获取后显示火焰图。

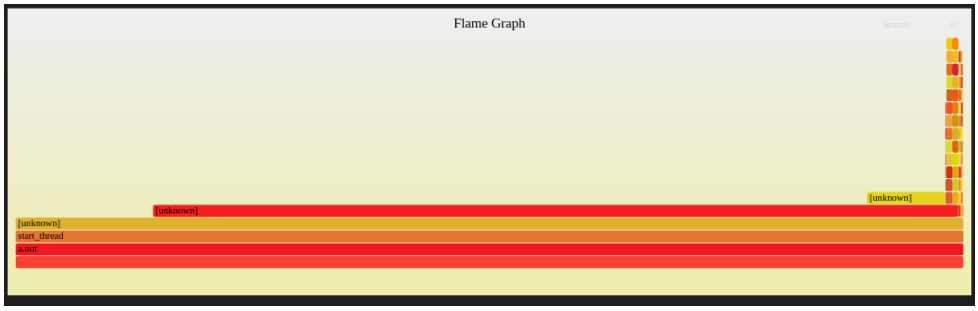


图 79 生成的火焰图

11、自动跳转到函数位置。

(1)选择要跳转函数的文件，并双击火焰图的函数名部分，然后点击 serch 按钮，将获取到的函数名放到文本框内。

图 80 函数位置跳转

(2)点击确定按钮，跳转到对应函数名位置。

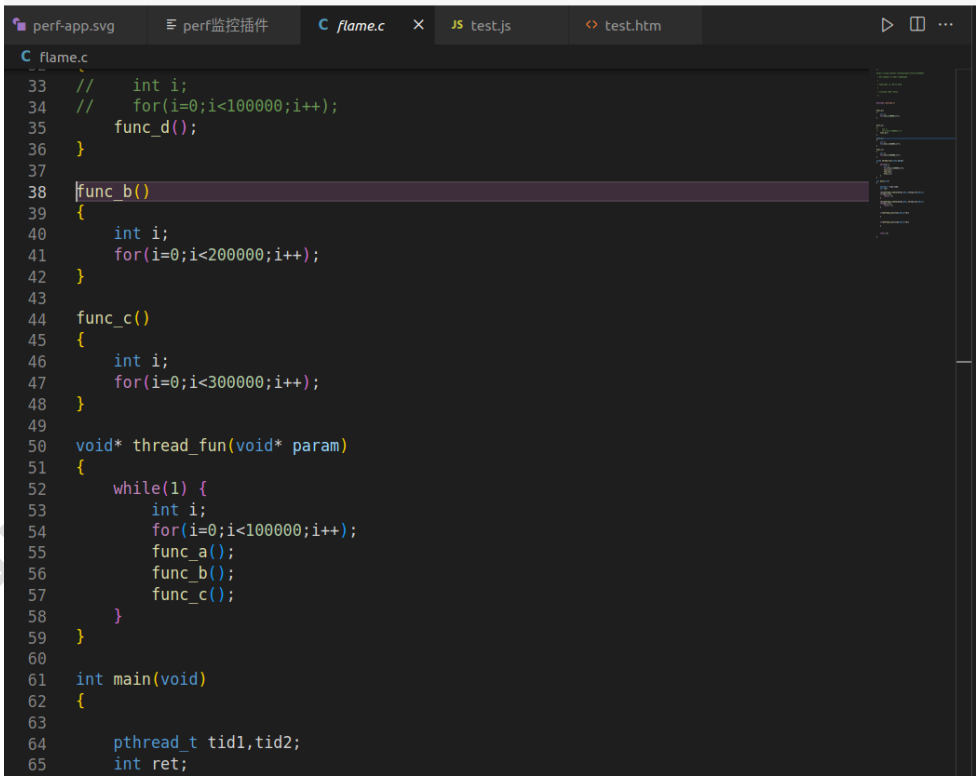


图 81 函数位置跳转 2

6　性能分析

6. 1共享资源争用分析

6.1.1　进程基本信息

功能介绍：分析显示某个进程的 CPU 占用情况、内存使用情况、进程状态、线程树、系统调用等信息。

操作步骤：

1、打开侧边栏通过侧边栏【进程基本信息】进入进程基本信息。

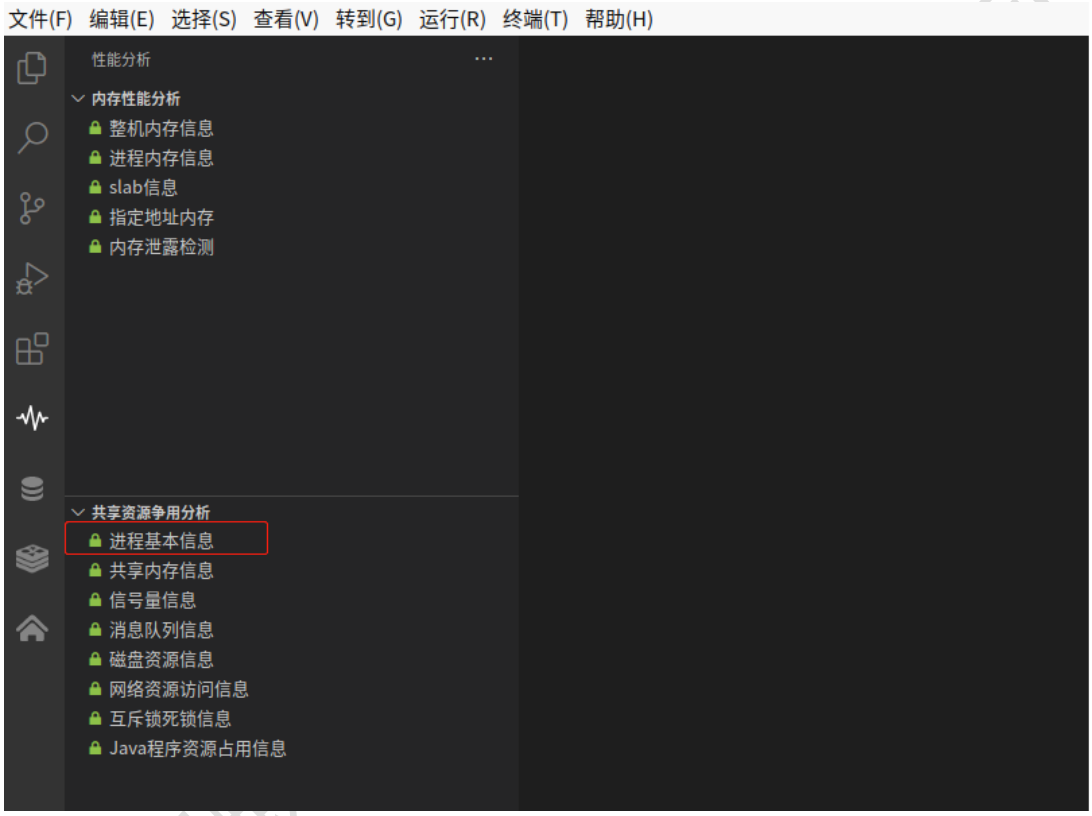


图 82 进程基本信息功能进入

2、进程基本信息的查看。

(1)点击【选择进程】按钮，或直接输入进程号 pid，然后点击【获取信息】按钮进行搜索。

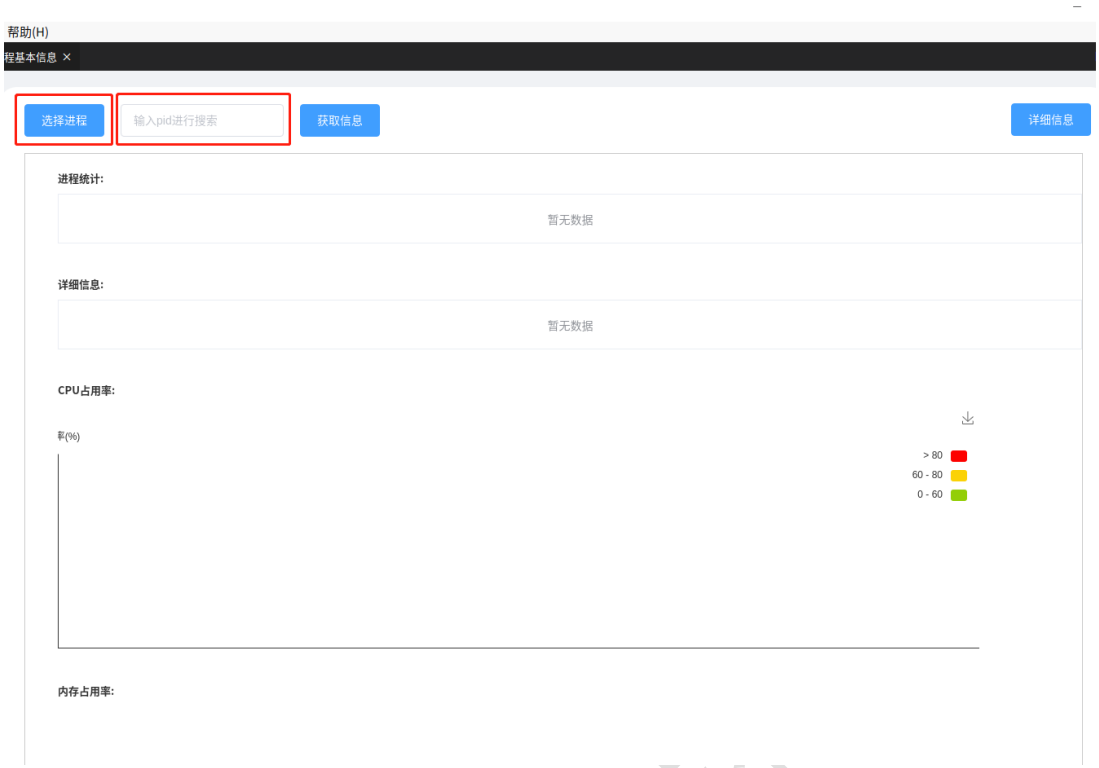


图 83 进程基本信息—选择进程

(2)点击【选择进程】后，可在进程列表中查看进程，可以输入命令进行筛选。

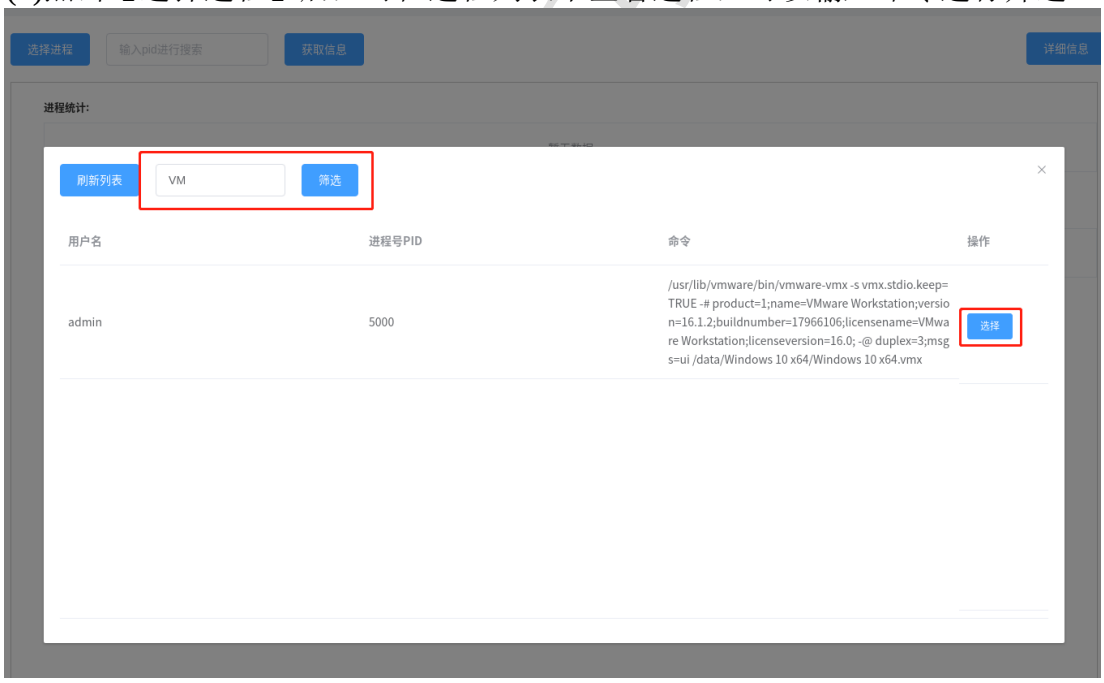


图 84 进程列表查询

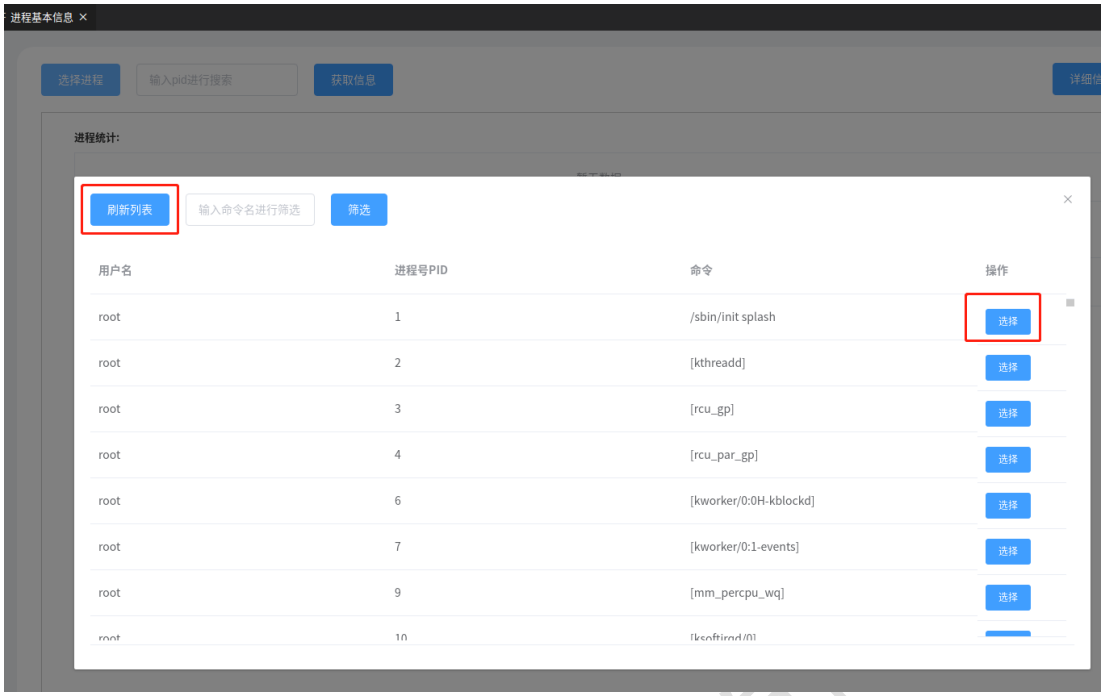


图 85 进程列表刷新与选择

(3)选择进程后，所选进程进程号 pid 会自动填入，点击【获取信息】进行搜索。

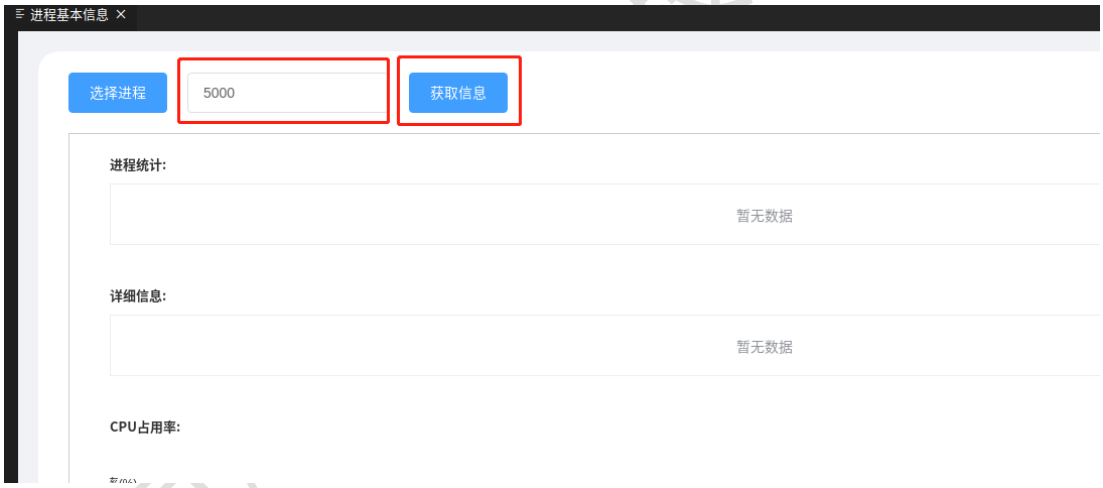


图 86 进程基本信息搜索

(4)搜索后，可查看【进程统计信息】，【详细信息】，【CPU 占用率视图】，【内存占用率视图】。

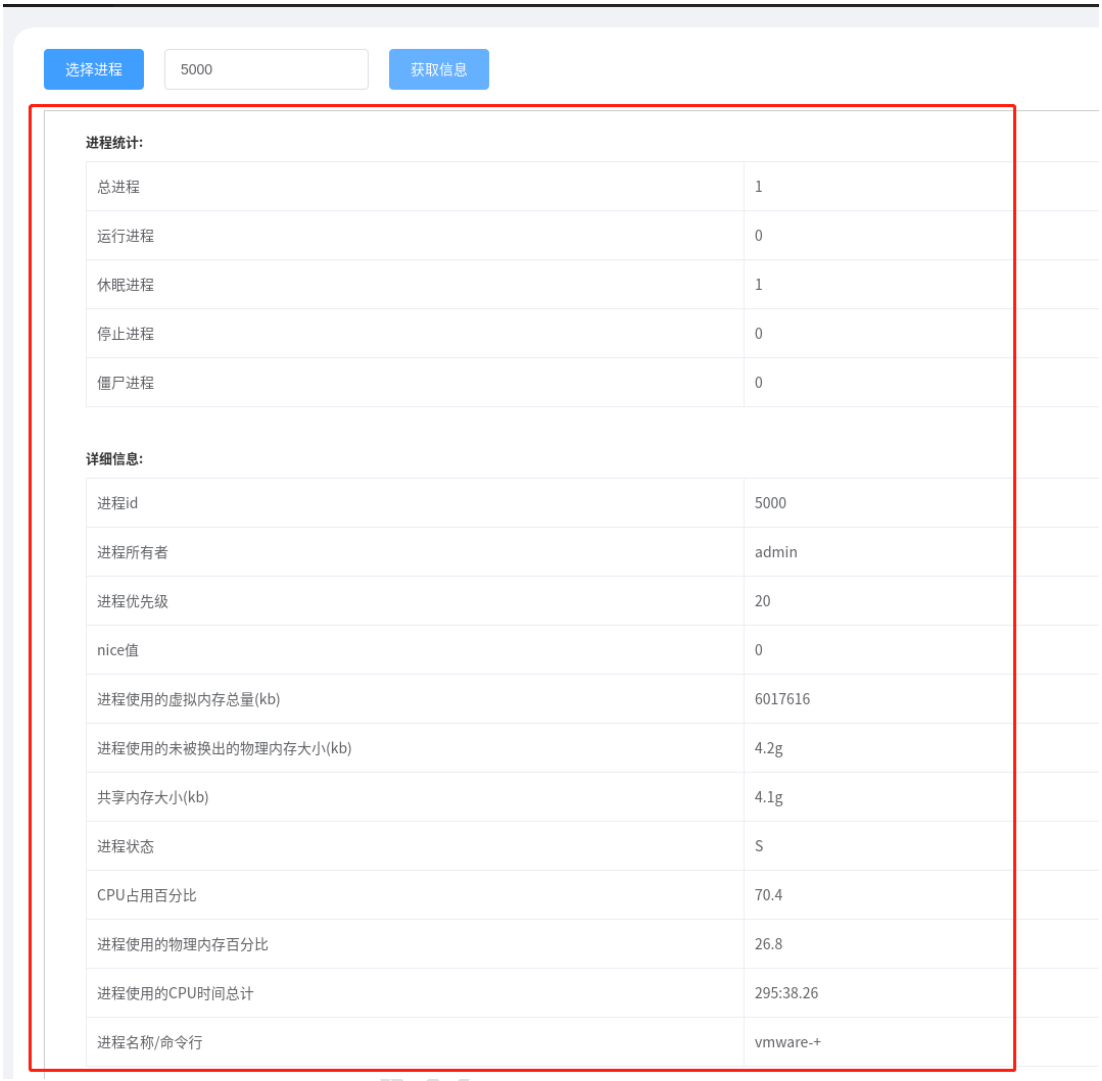


图 87 进程基本信息查看

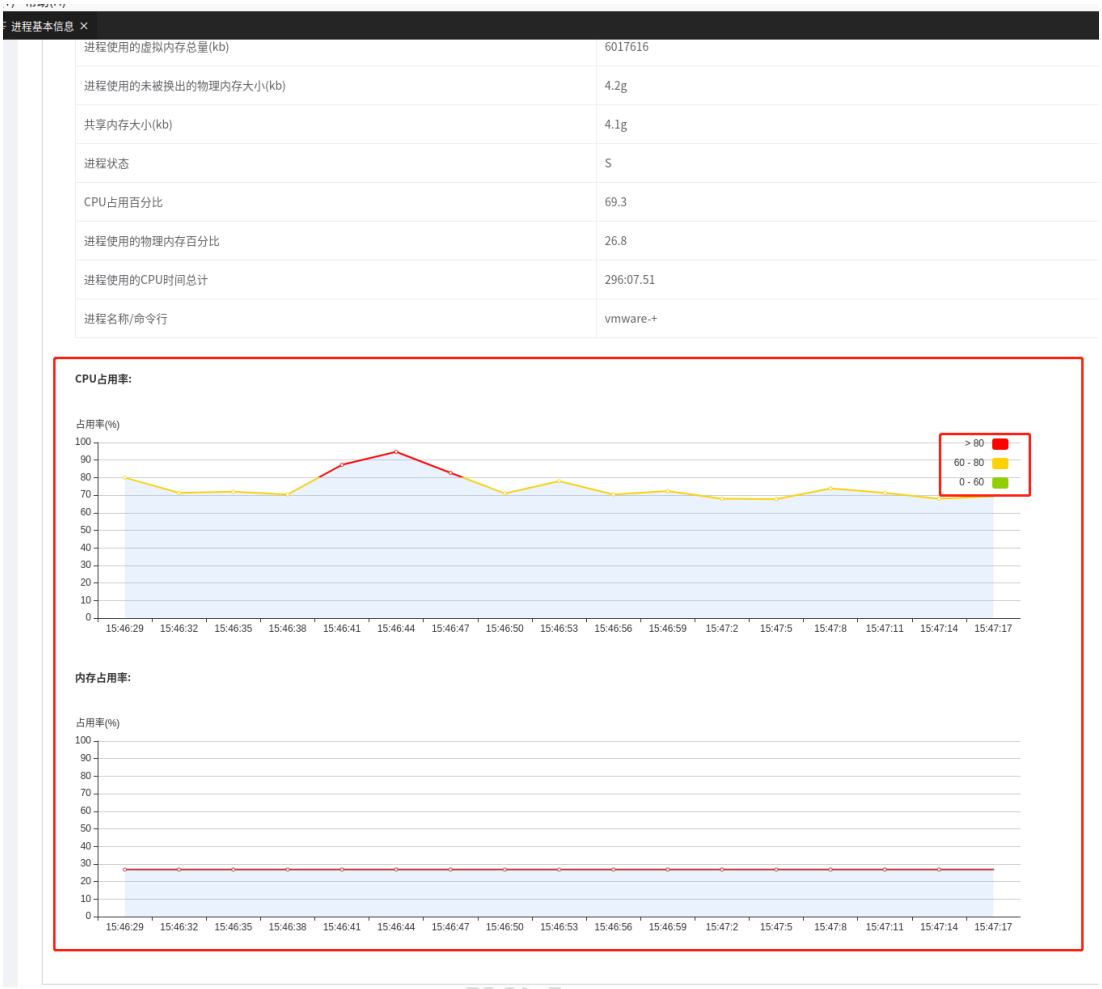


图 88 进程基本信息查看 2

3、进程详细信息的查看。

(1)点击【详细信息】按钮，可进入进程详细信息页面。

图 89 进程详细信息进入

(2)在【进程状态信息】页面下，选择进程或输入进程号 pid 后，点击【获取信息】按钮进行查询。

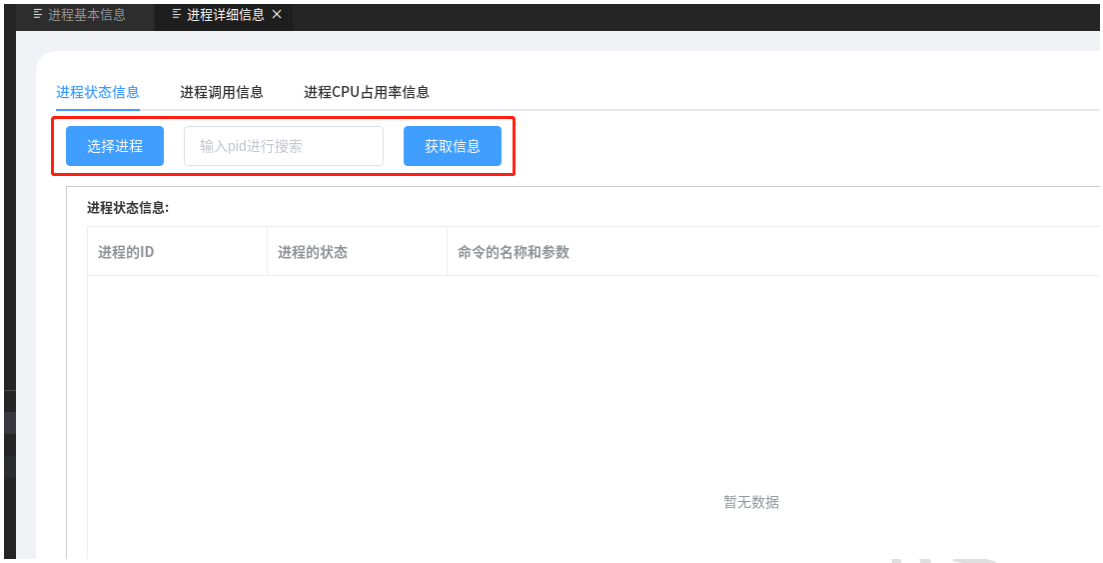


图 90 进程详细信息搜索

(3)可查看【进程状态信息】、【线程树信息】、【各线程详细信息】。

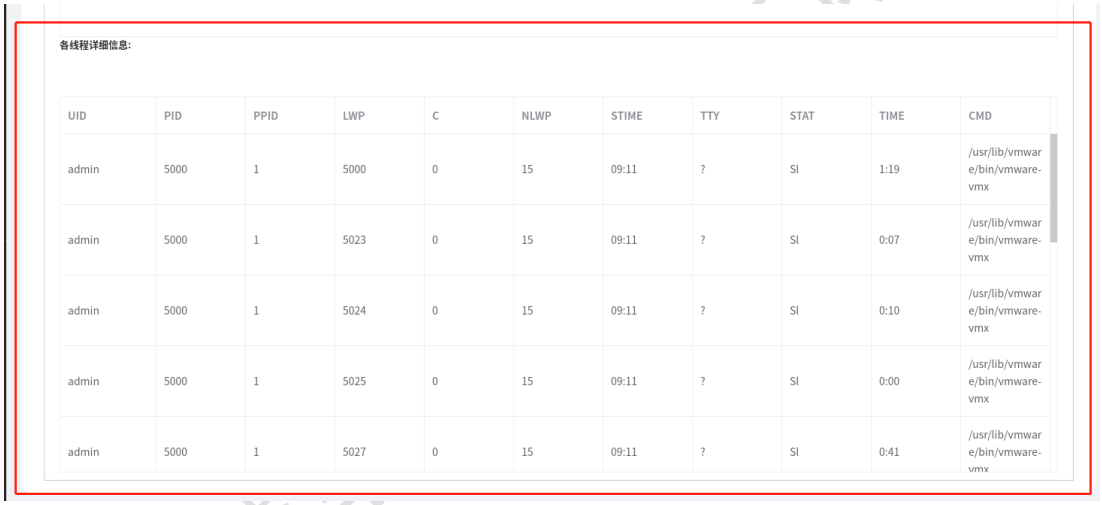


图 91 进程详细信息查看

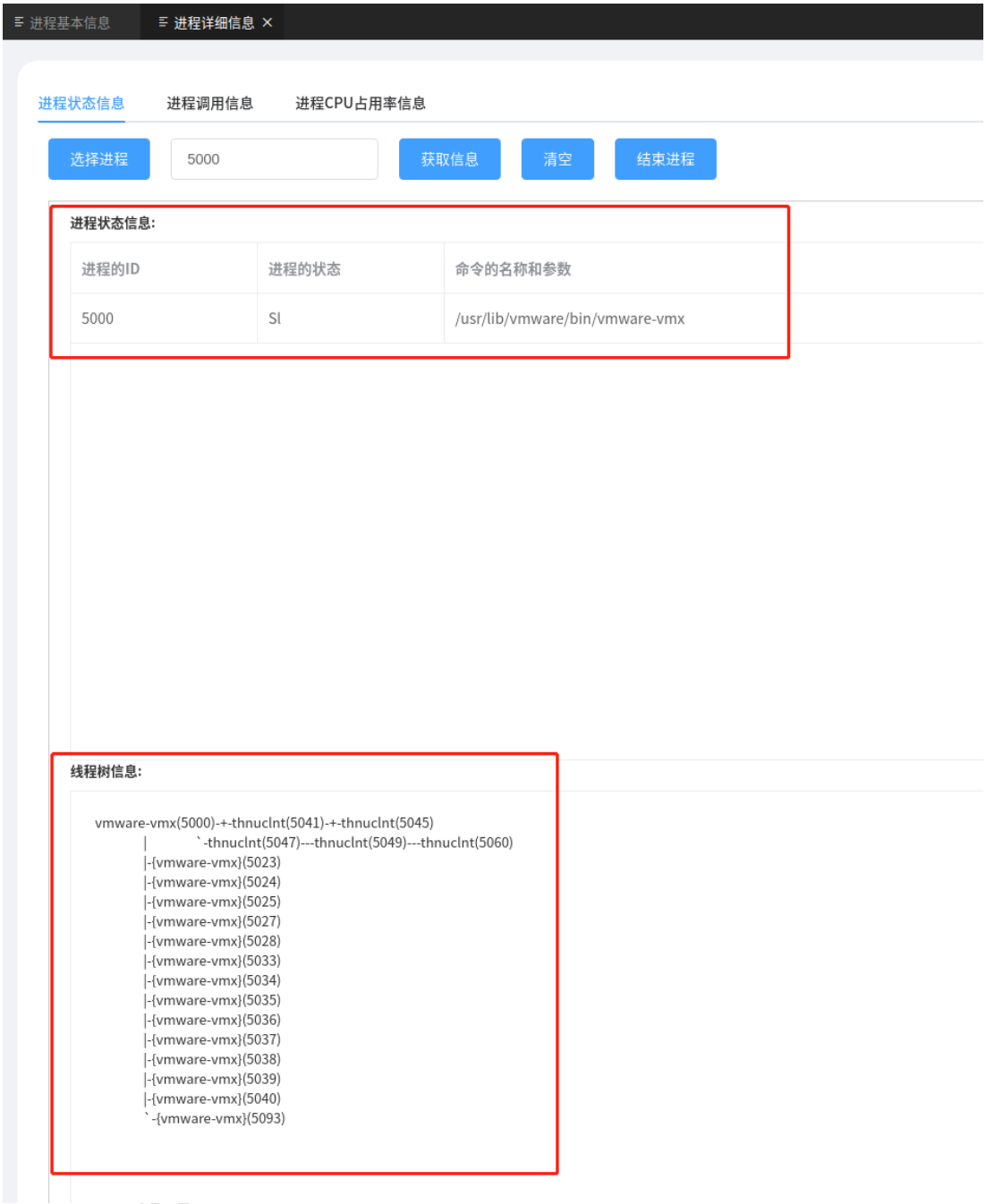


图 92 进程状态信息查看

(4)在【进程调用信息】页面下，选择进程或输入进程号 pid 后，点击【获取信息】按钮进行查询，可查询进程调用信息；点击【停止】按钮，可停止进程调用信息的获取。

图 93 进程调用信息查询

(5)在【进程 CPU 占用率信息】页面下，选择进程或输入进程号 pid 后，点击【获取信息】按钮进行查询，可实时查询 CPU 占用率信息、CPU 占用率详情信息。

(6)CPU 占用率 0-60%展示为绿色，60%-80%展示为黄色，超过 80%展示为红色。

图 94 进程 CPU 占用率信息查询

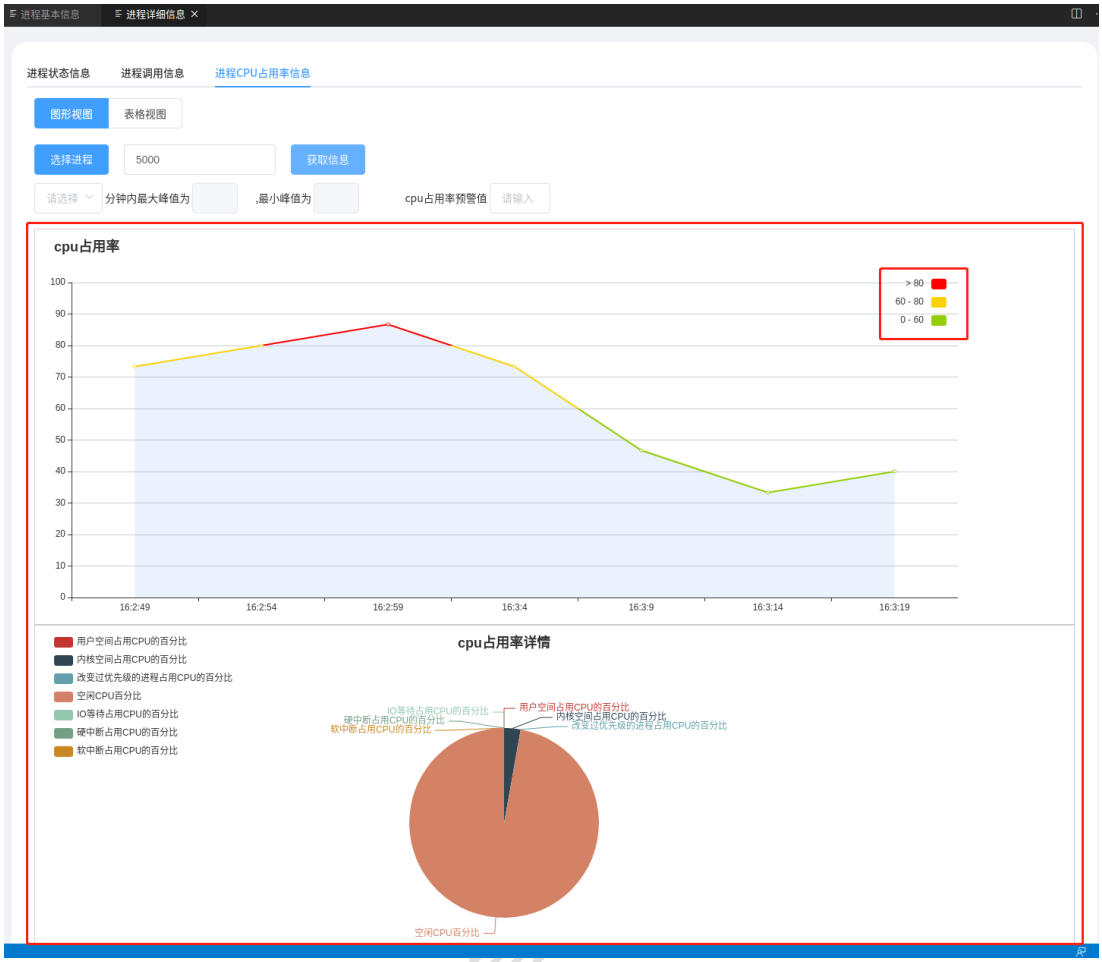


图 95 进程 CPU 占用率查看

(7)选择【刷新频率】，可修改刷新频率。

(8)选择【峰值时间】可显示所选时间内 CPU 占用率最大峰值与最小峰值。

图 96 进程 CPU 占用率信息过滤

(9)输入【CPU 占用率预警值】信息，当 CPU 占用率超出预警值，则进行弹窗预警。

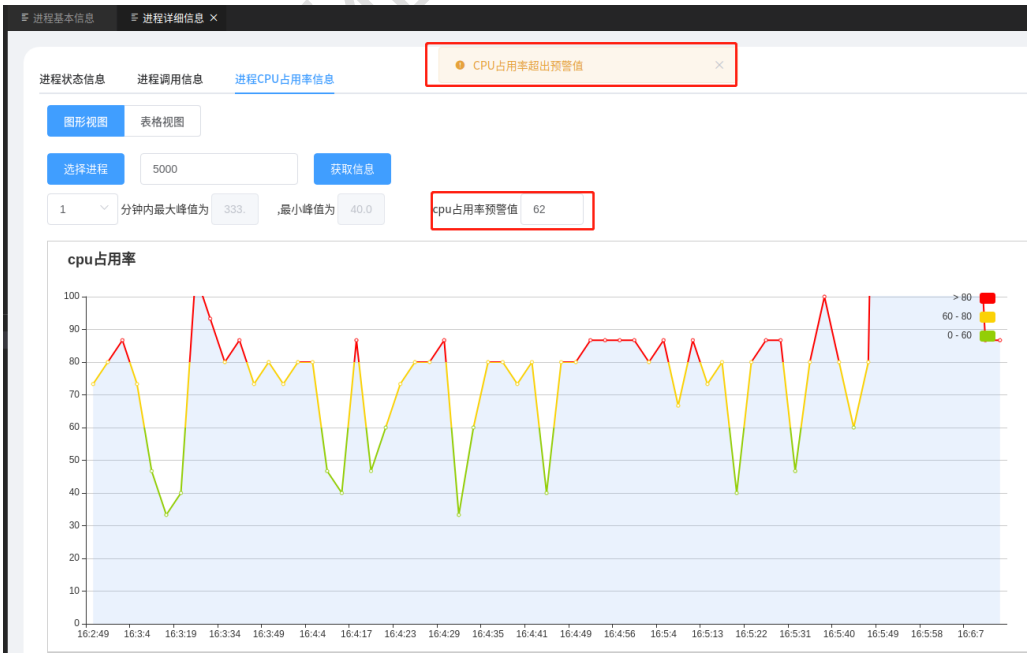


图 97 进程 CPU 占用率预警

6.1.2 共享内存信息

功能介绍：分析显示共享内存资源的相关信息。

操作步骤：

1、打开侧边栏通过侧边栏【共享内存信息】进入共享内存信息。

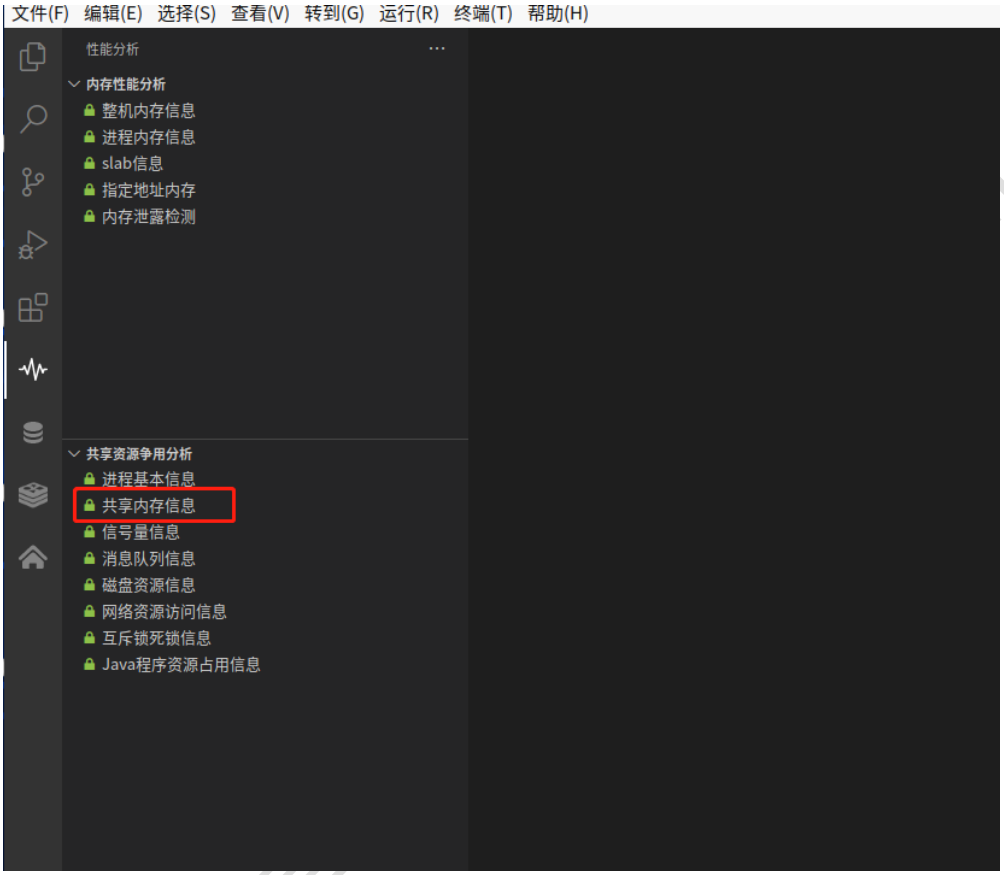


图 98 共享内存信息功能进入

2、在【共享内存详细信息】页面下，可查看共享内存详细信息。

在【共享内存统计信息】页面下，可查看共享内存统计信息

图 99 共享内存统计信息显示

6.1.3 信号量、消息队列信息

功能介绍：分析显示进程信号量、消息队列的相关信息。

操作步骤：

1、打开侧边栏通过侧边栏【信号量信息】【消息队列信息】进入对应页面。

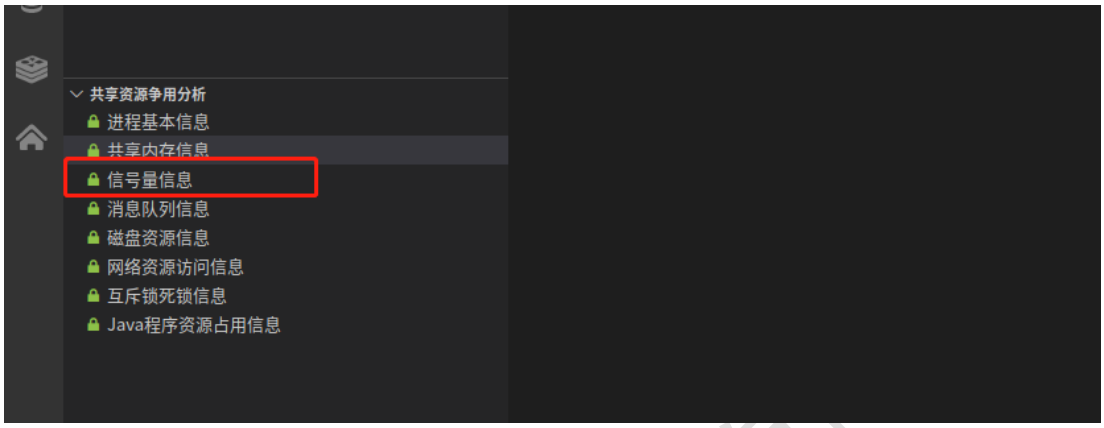


图 100 信号量信息功能进入

2、在【信号量信息】【消息队列信息】页面下，可查看信号量信息与消息队列信息。

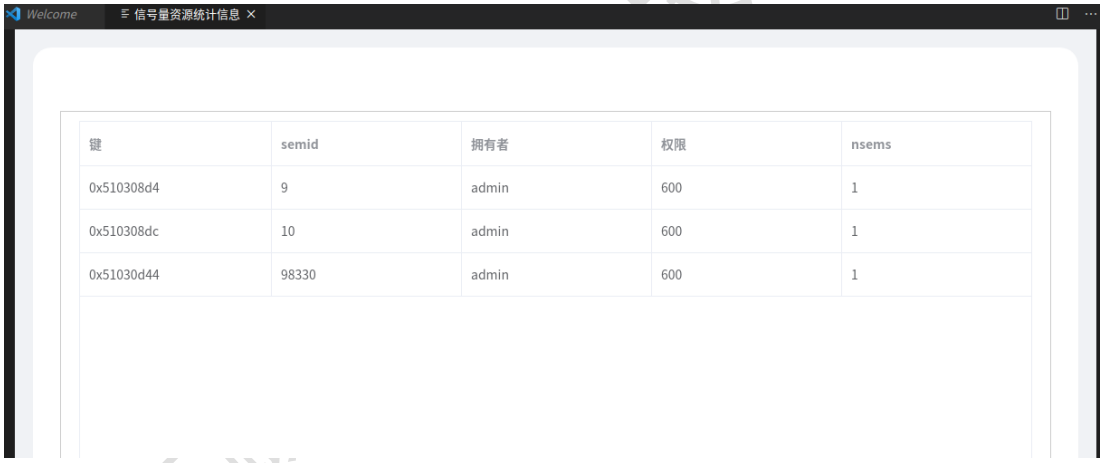


图 101 信号量信息查看

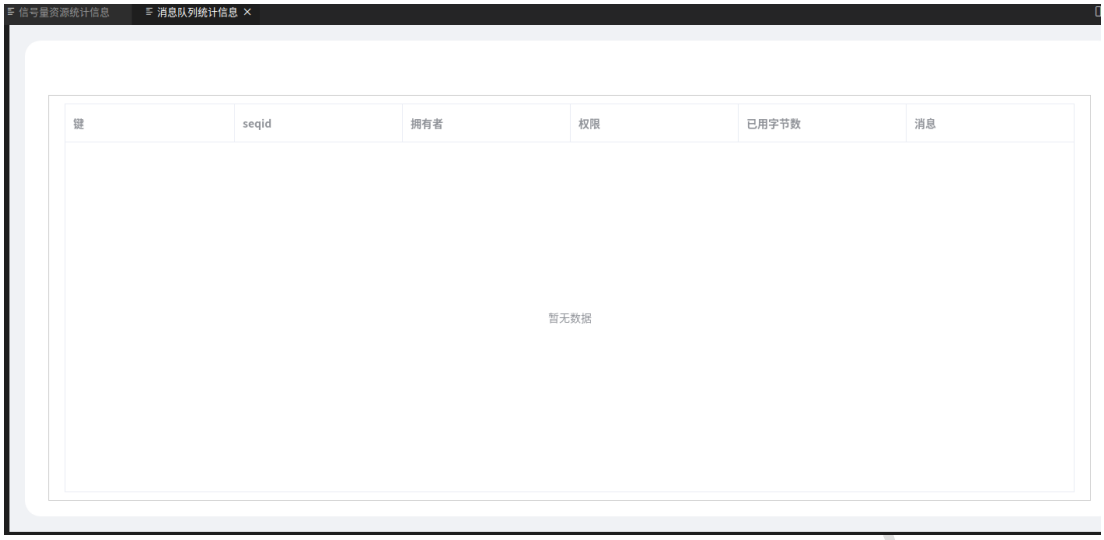


图 102 消息队列信息查看

6.1.4 磁盘资源信息

功能介绍：分析显示进程磁盘资源的相关信息。

操作步骤：

1、打开侧边栏通过侧边栏【磁盘资源信息】进入磁盘资源信息。

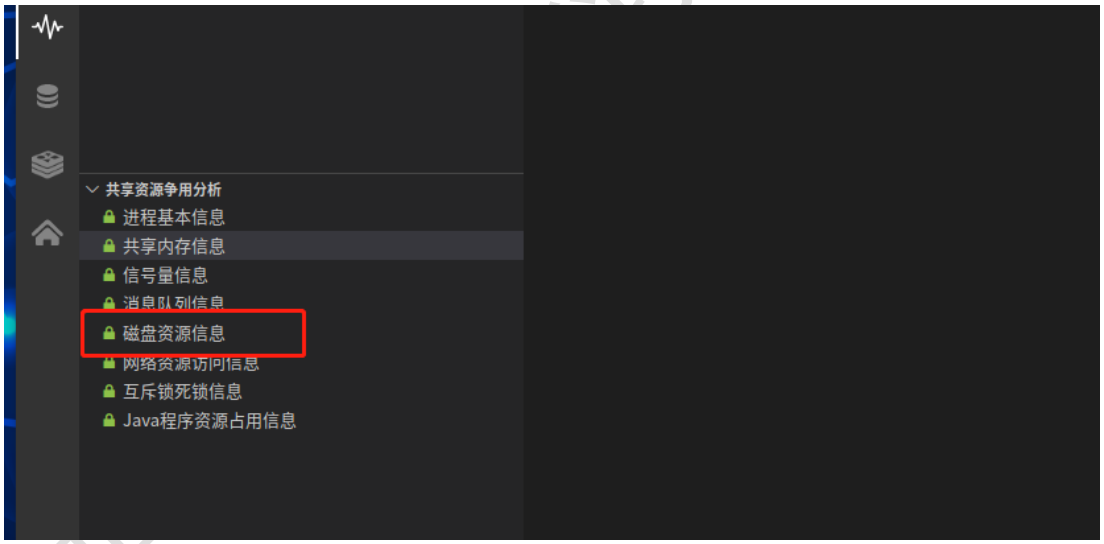


图 103 磁盘资源信息功能进入

2、获取磁盘信息。

点击【选择进程】按钮，或直接输入进程号 pid，输入电脑用户密码，然后点击【获取信息】按钮进行搜索。

图 104 磁盘资源信息搜索

搜索后，可查看进程【IO 信息】与【磁盘信息】。

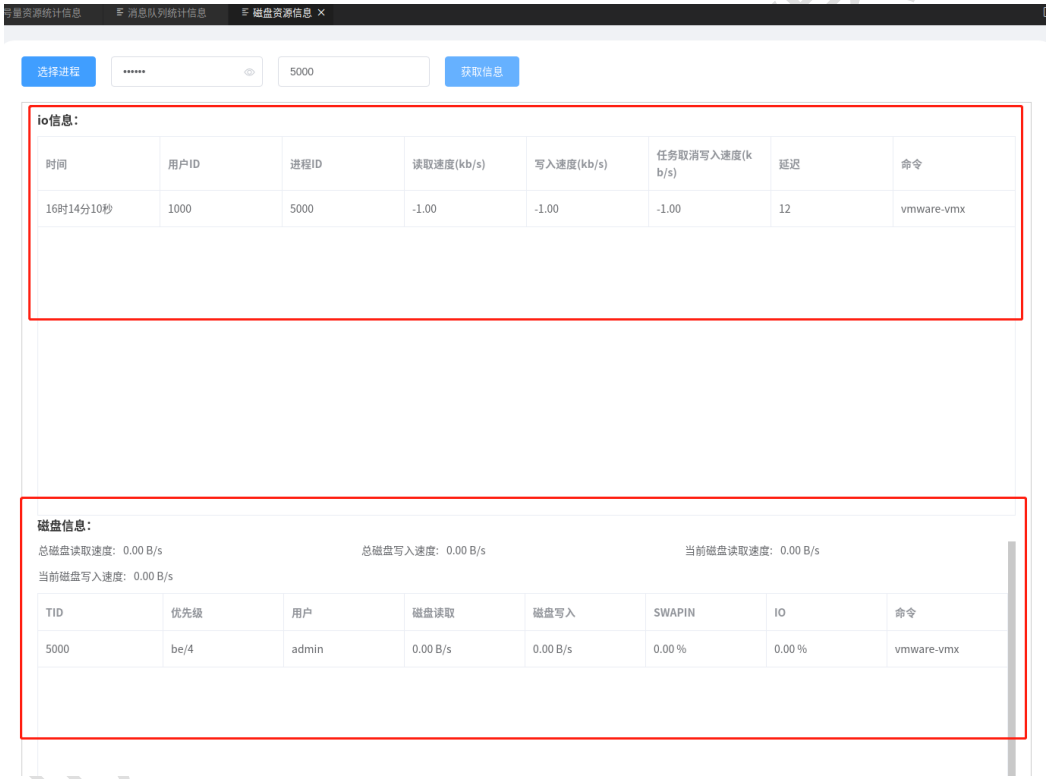


图 105 IO 信息与磁盘信息查看

6.1.5 网络资源访问信息

功能介绍：分析显示进程网络资源访问的相关信息。

操作步骤：

1、打开侧边栏通过侧边栏【网络资源访问信息】进入网络资源访问信息。

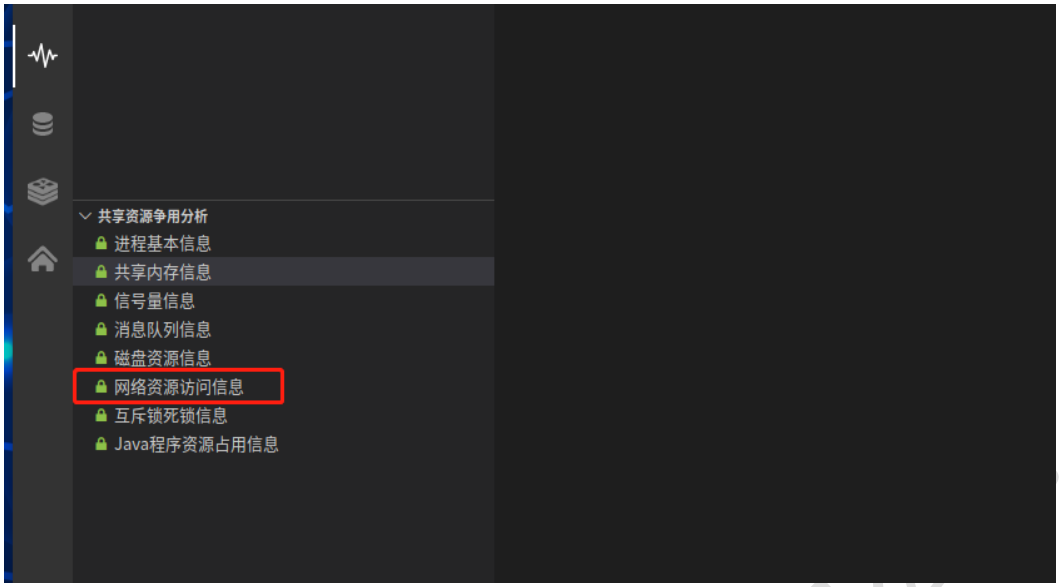


图 106 网络资源访问信息功能进入

2、查看网络资源访问信息。

(1)在【unix 信息】页面下，可查看 unix 信息。

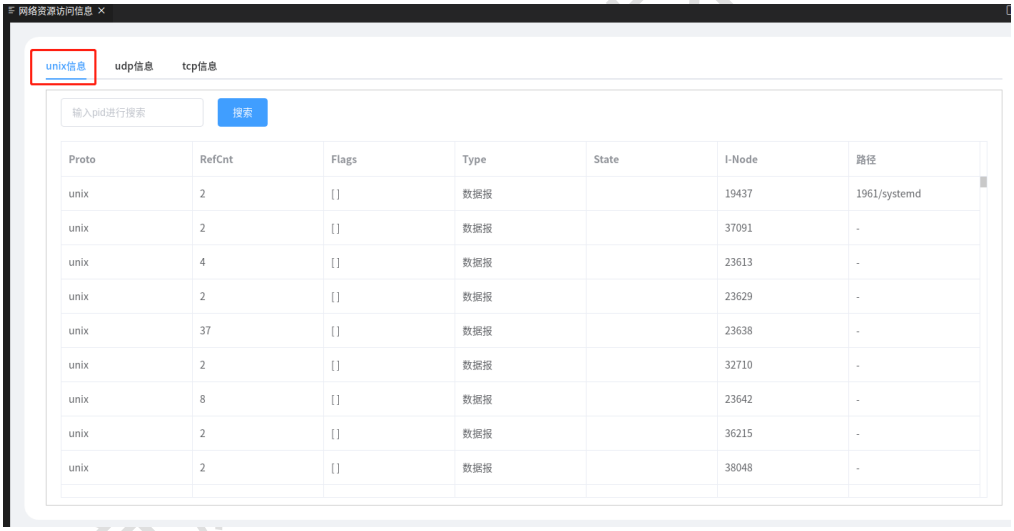


图 107 unix 信息监控

(2)在【udp 信息】页面下，可查看【所有 udp 端口信息】【udp 端口统计信息】。

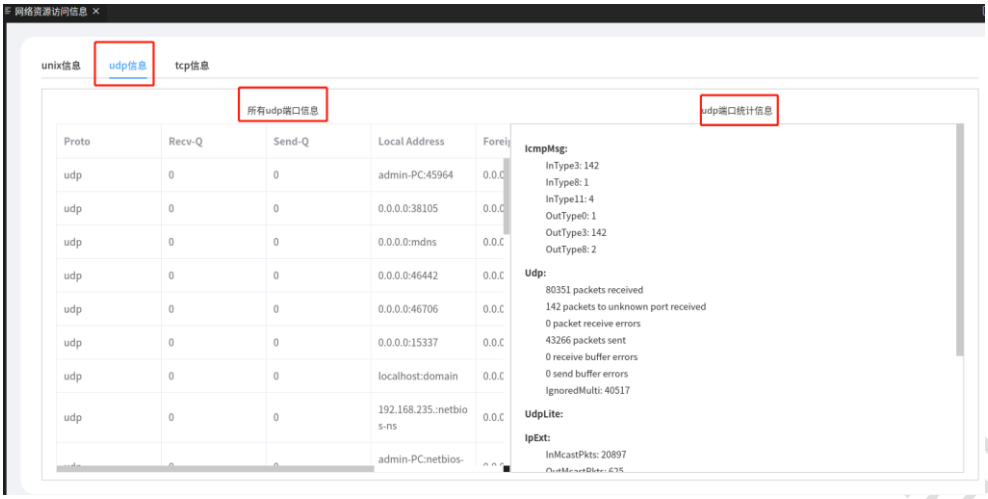


图 108 udp 信息查看

(3)在【tcp 信息】页面下，可查看【所有 tcp 端口信息】【tcp 端口统计信息】。

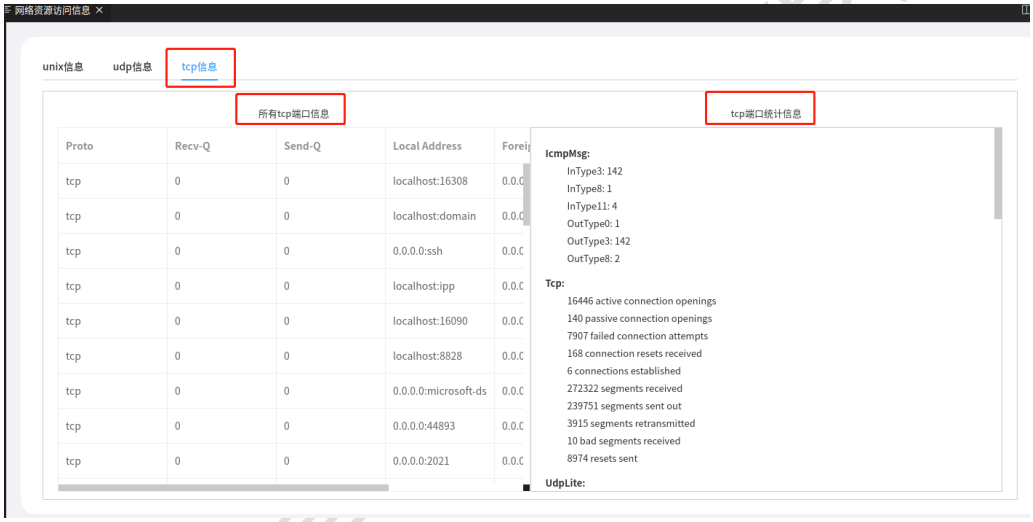


图 109 tcp 信息查看

6.1.6 Java 程序资源占用信息

功能介绍：实时显示某个 Java 进程的垃圾回收信息、堆栈信息。

操作步骤：

1、打开侧边栏通过点击侧边栏【Java 程序资源占用信息】进入 Java 程序资源占用信息。

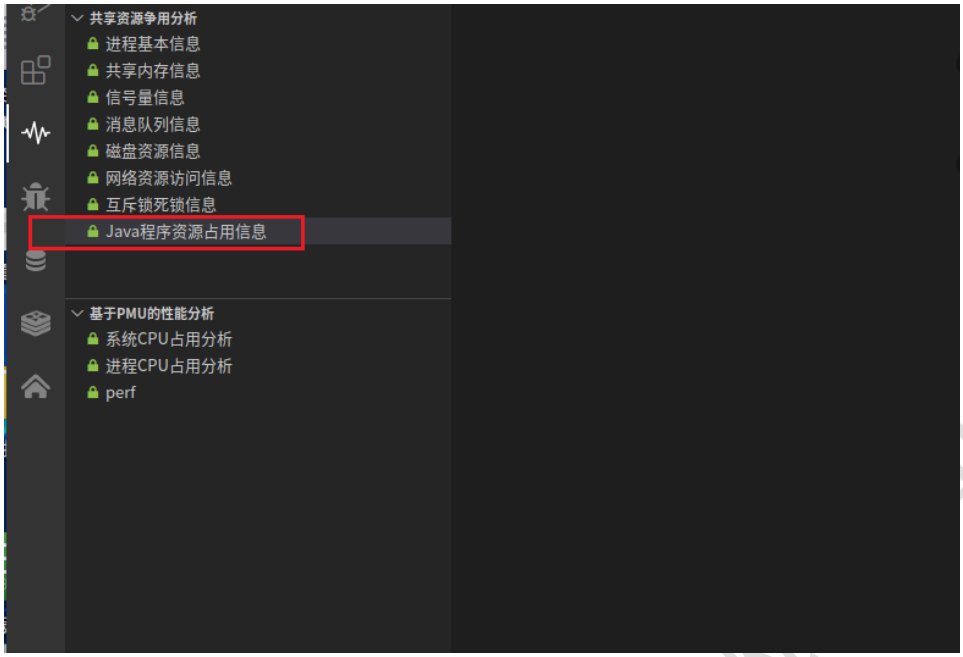


图 110 java 程序资源占用信息功能进入

2、Java 程序垃圾回收信息查看。

(1)在【Java 垃圾回收信息显示】页面下，通过点击【选择进程】按钮，选择一个 Java 进程，或直接输入一个 Java 进程的进程号 pid，然后点击【获取信息】按钮进行搜索。

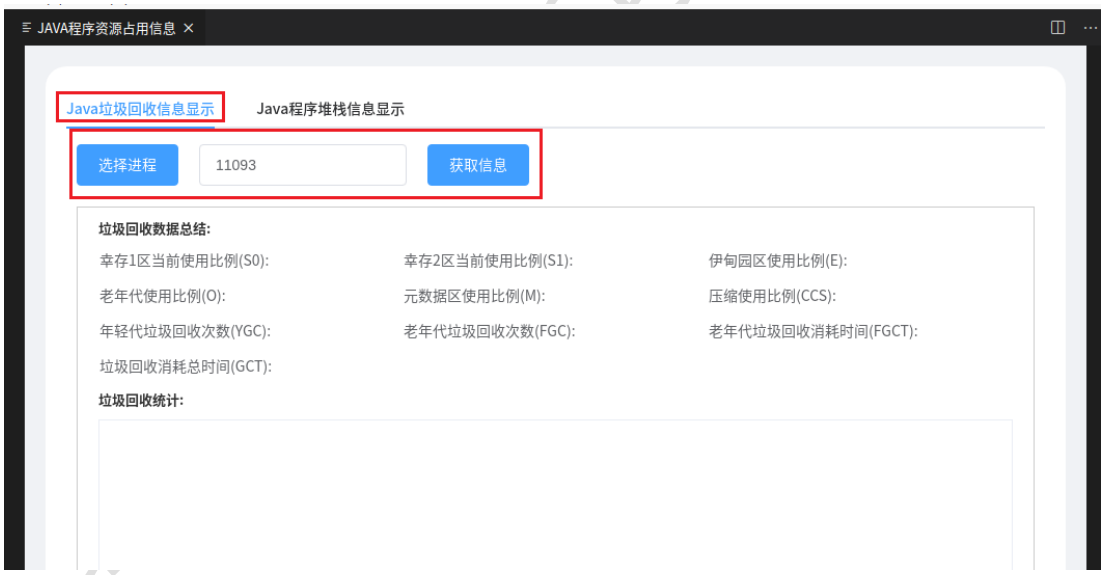


图 111 java 垃圾回收信息搜索

(2)搜索后，可查询出搜索的 Java 进程的垃圾回收等信息。

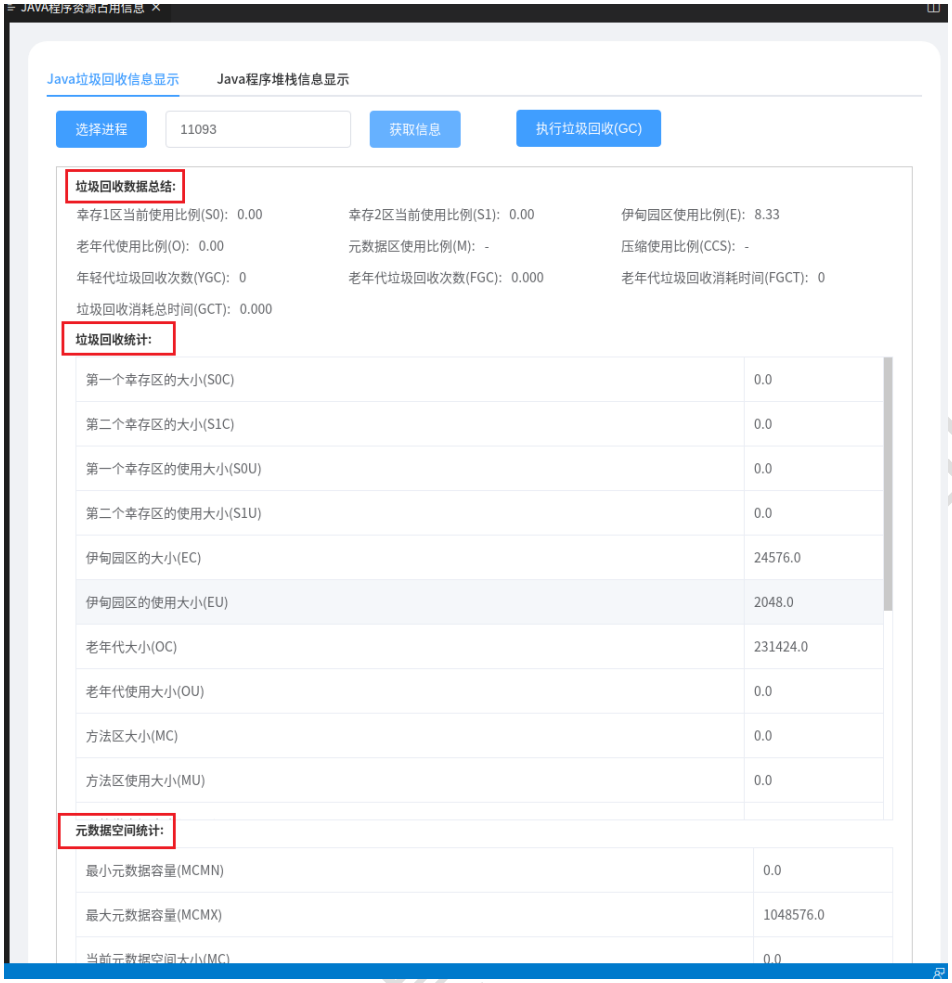


图 112 java 程序垃圾回收信息展示

3、Java 程序堆栈信息查看。

(1)在【Java 程序堆栈信息显示】页面下，通过点击【选择进程】按钮，选择一个 Java 进程，或直接输入一个 Java 进程的进程号 pid，然后点击【获取信息】按钮进行搜索。

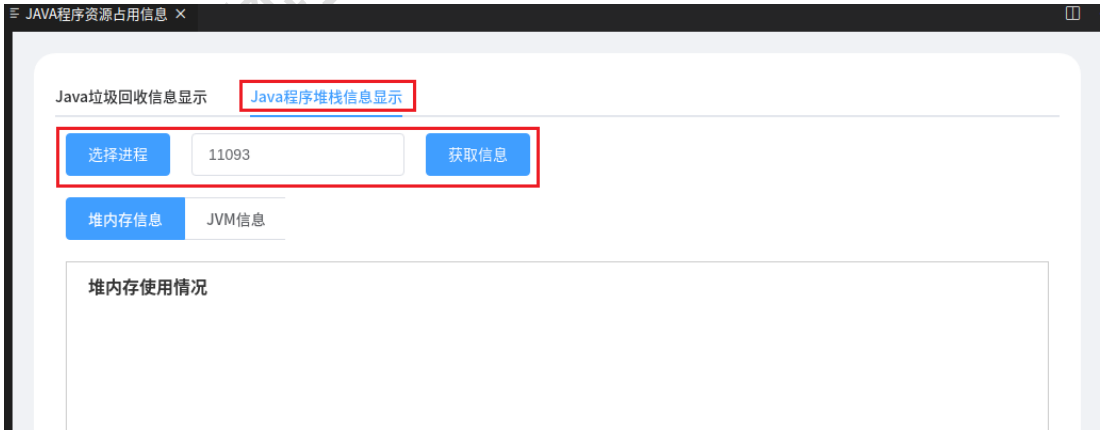


图 113 java 程序堆栈信息搜索

(2)搜索后，可查询出搜索的 Java 进程的【堆内存信息】、【JVM 信息】。

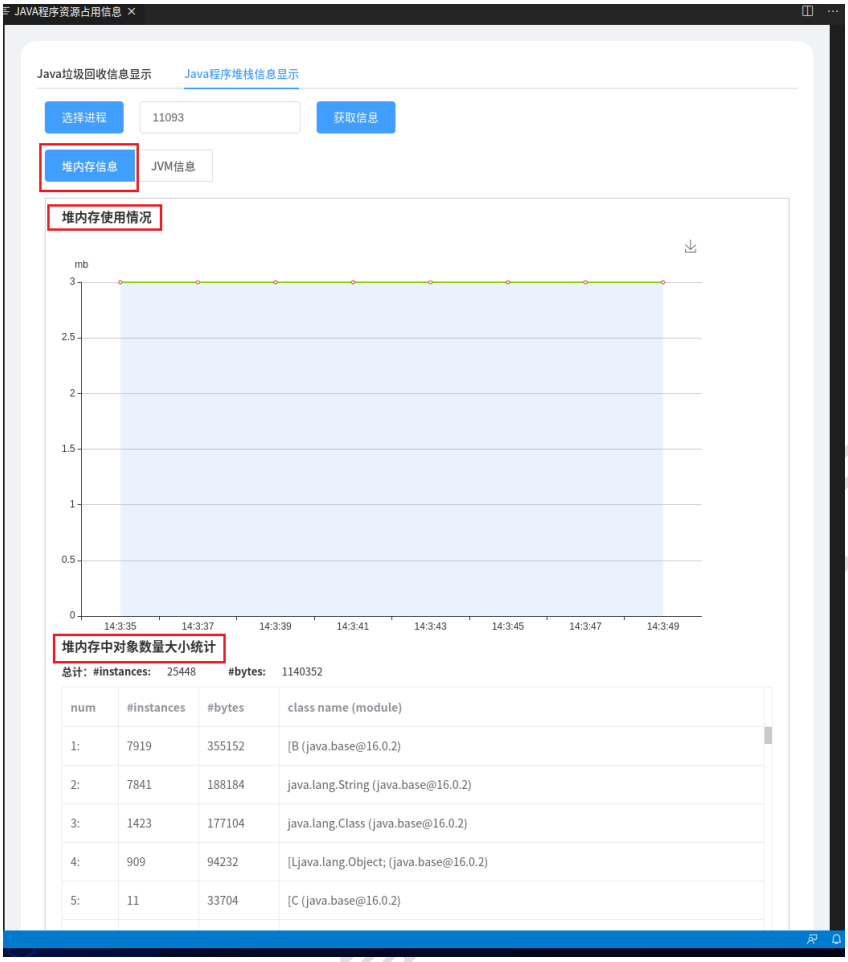


图 114 java 程序堆栈信息展示

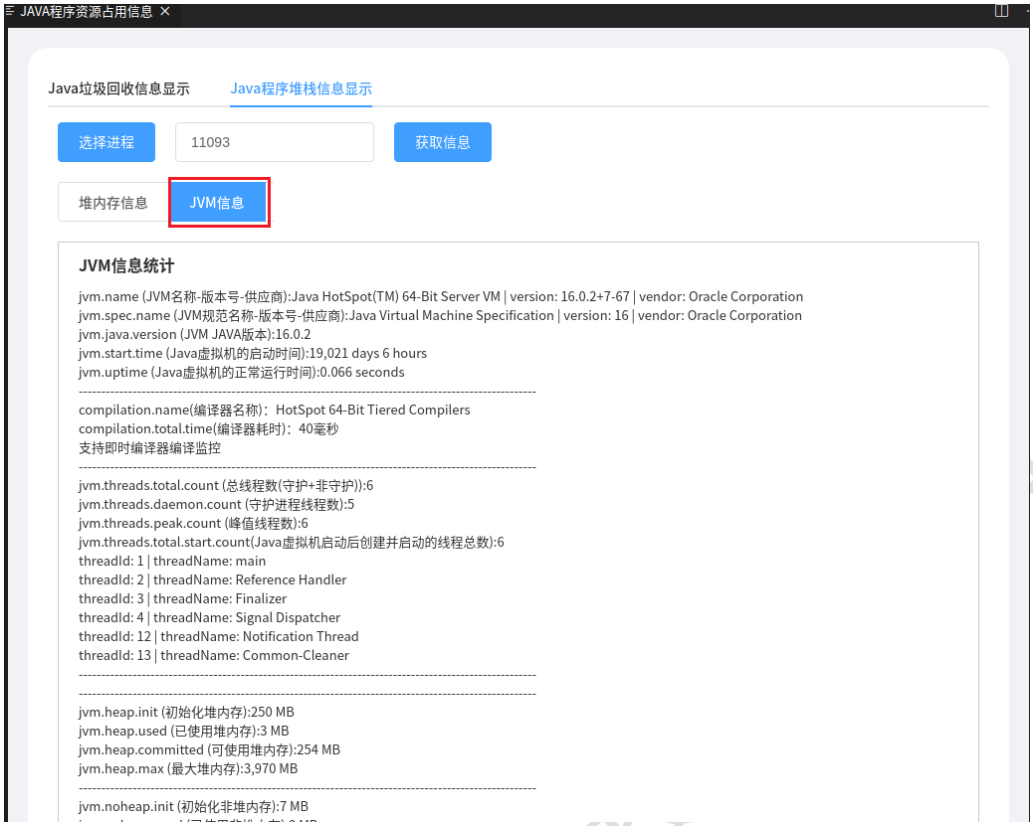


图 115 java 程序堆栈信息展示 2

6.2内存性能分析

6.2.1 整机内存信息

功能介绍：实现整机内存信息进行实时监控功能，并可以对当前的整机内存信息导出，在以后需要查看时将需要查看的信息导入进行查看。并可以对进程统计信息进行实时监控。

操作步骤：

1、进入方式。

打开侧边栏，通过侧边栏红框内内容进入整机内存信息插件。

图 116 整机内存信息功能进入

2、查看系统内存信息。

选择整机内存信息，点击【系统内存监控】，进入系统内存监控页，查看系统内存信

息。

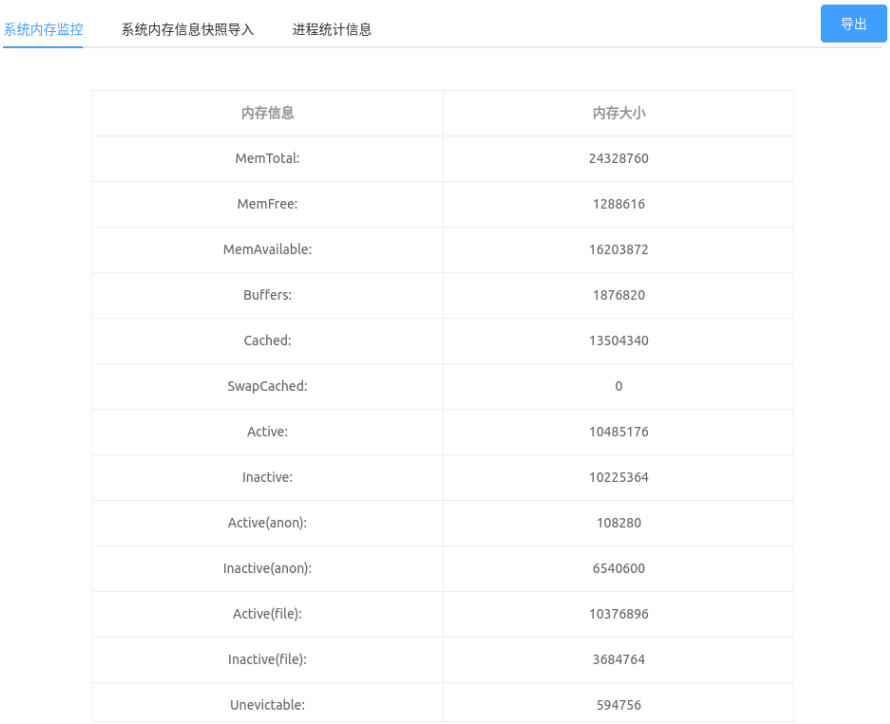


图 117 系统内存信息查看

3、系统内存信息快照导出。

(1)点击【导出】按钮。

图 118 系统内存信息快照导出

(2)弹出资源管理器，选择导出路径名为系统内存信息-时间戳的 txt 文档。

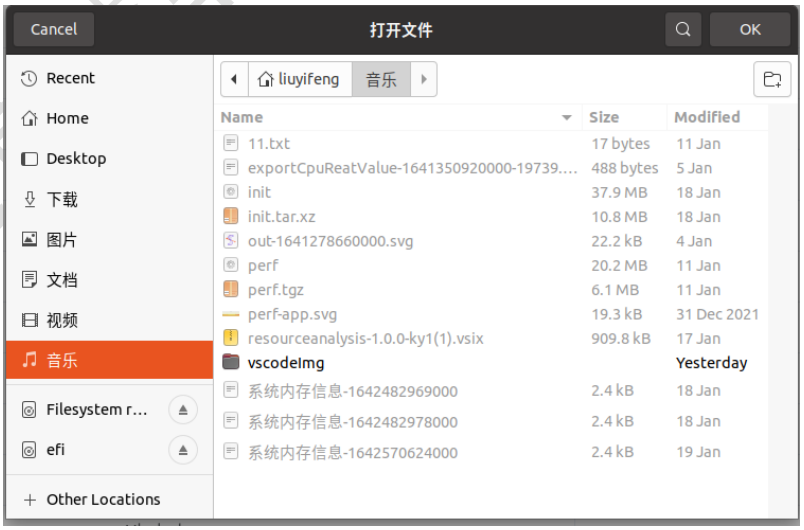


图 119 导出路径选择

(3)导出的 txt 文档内容如下。

图 120 导出内容

6.2.2 进程内存信息

功能介绍：支持通过进程号搜索对该进程占用的内存信息进行实时监控，并可以查看它的不同时间内的对比信息和内存地址信息。

操作步骤：

1、进程内容监控。

(1)点击内存性能分析下的进程内存信息选项，进入进程内存信息插件。

图 121 进程内存信息功能进入

(2)点击【选择进程】按钮。

图 122 进程内存监控选择进程

(3)弹出进程号相关列表，可进行相关操作。

刷新按钮：点击【刷新列表】按钮，对进程号列表进行刷新。

搜索按钮：在文本框输入关键词，点击【搜索】按钮，模糊匹配命令中的单词。

选择按钮：点击【选择】按钮将选择的进程号返回到页面。

(4)选择或输入进程号，并输入 sudo 密码，然后点击【查找进程内存】选项，以折线图显示该进程内存信息。

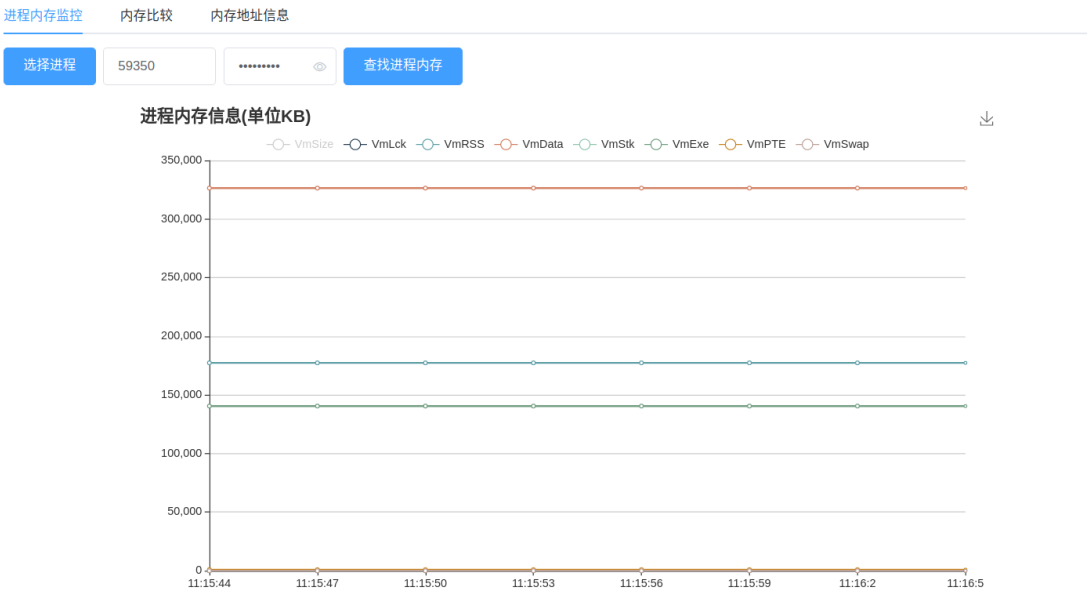


图 123 进程内存信息

2、内存比较信息。

点击【内存比较】，查看内存比较信息

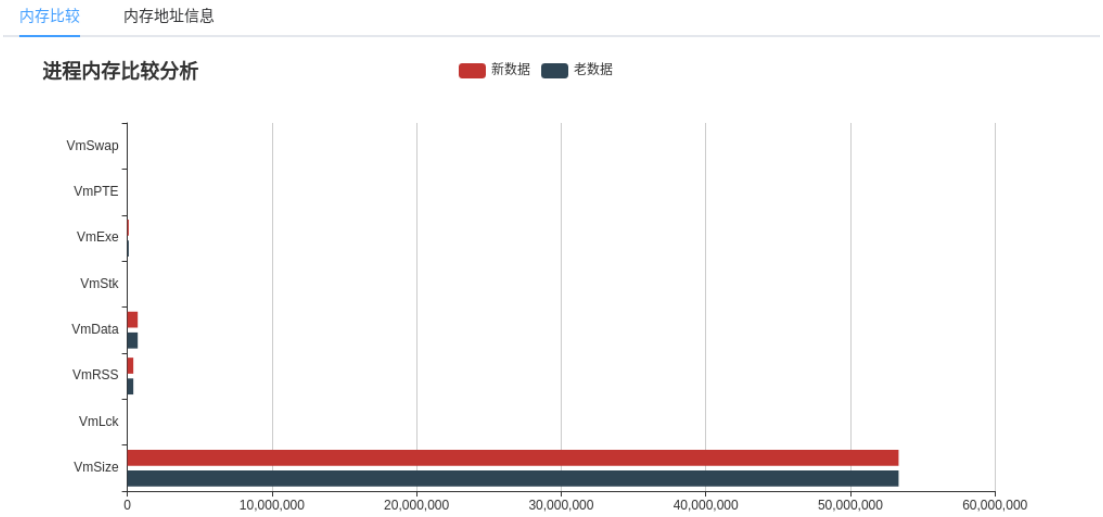


图 124 内存比较信息

3、内存地址信息。

点击【内存地址信息】，以树形图形式查看内存地址信息，可以对地址进行展开显示。

图 125 内存地址信息

6.2.3 slab 信息监控

功能介绍：对 slab 信息进行监控，并支持手动刷新以获取相关信息。

操作步骤：

1、进入 slab 信息插件。

点击内存性能分析下的【slab 信息】选项，进入 slab 信息插件。

图 126 slab 信息监控功能进入

2、刷新当前 slab 信息。

进入 slab 信息插件，输入密码，点击【获取 slab 信息】按钮，手动刷新查看当前 slab 相关信息。

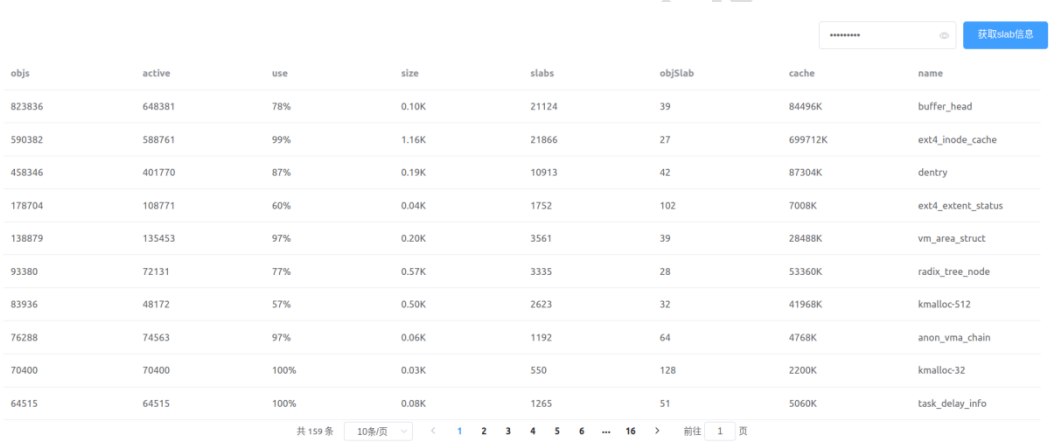


图 127 slab 信息展示

6.2.4 指定地址内存

功能介绍：通过进程内存地址和进程号，查看该内存地址的进制信息。

操作步骤：

1、进入方式。

点击内存性能分析下的指定地址内存选项，进入指定地址内存插件。

图 128 指定地址内存功能进入

2、内存地址的获取。

(1)运行可执行文件， 通过 ps -ef |grep 可执行文件名获取进程号。

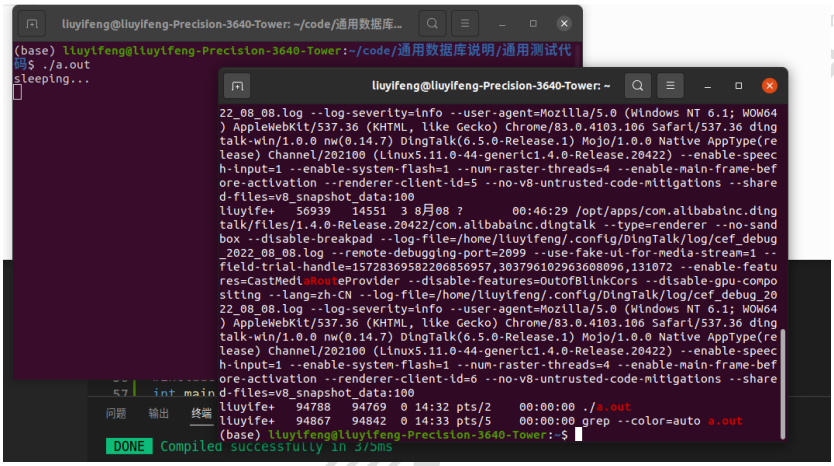


图 129 指定地址内存---运行可执行文件

(2)内存地址的获取：

1)运行命令 sudo gdb -p pid。

2)进入调试状态运行 p &glob 命令然后按 q 退出。

图 130 指定地址内存---内存地址获取

3、插件的使用。

将获取的内存地址和进程号输入到相应位置并选择字节数等信息点击确定按钮获取相关信息。

图 131 内存地址相关信息展示

6.3基于 PMU 的性能分析

6.3.1 系统 CPU 占用分析

功能介绍：对系统 CPU 占用率进行分析，并支持显示 CPU 架构信息。

操作步骤：

1、系统 CPU 占用分析。

选择基于 PMU 的性能分析下的【系统 CPU 占用分析】选项，进入该插件。

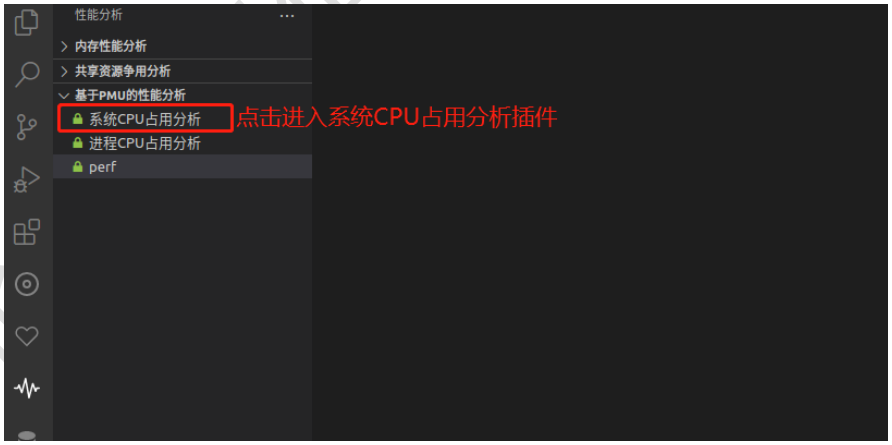


图 132 系统 CPU 占用分析功能进入

2、CPU 架构信息。

选择【系统 CPU 占用分析】，显示 CPU 架构信息。

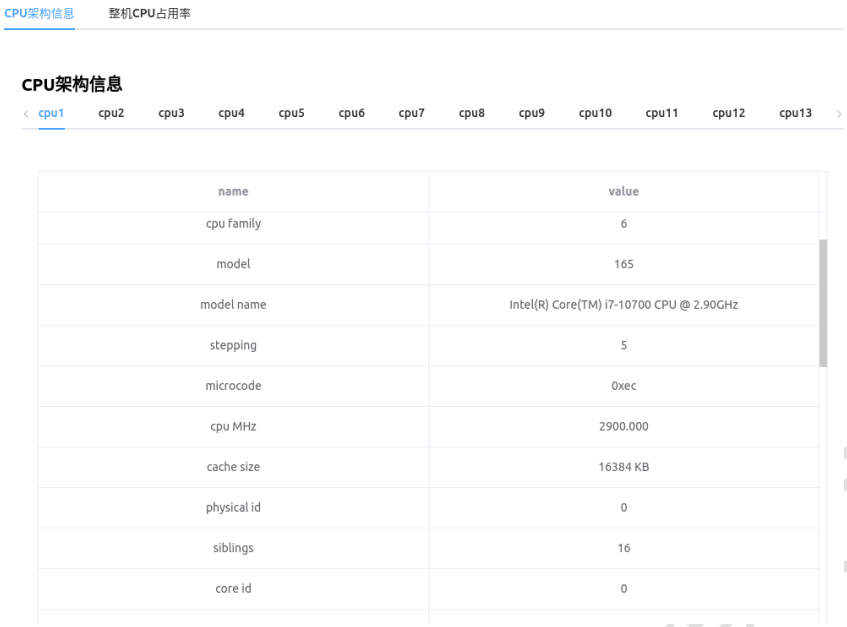


图 133 系统 CPU 占用分析

3、选择【CPU 占用率】选项，进入该页面每三秒钟，刷新整机当前 CPU 占用率。

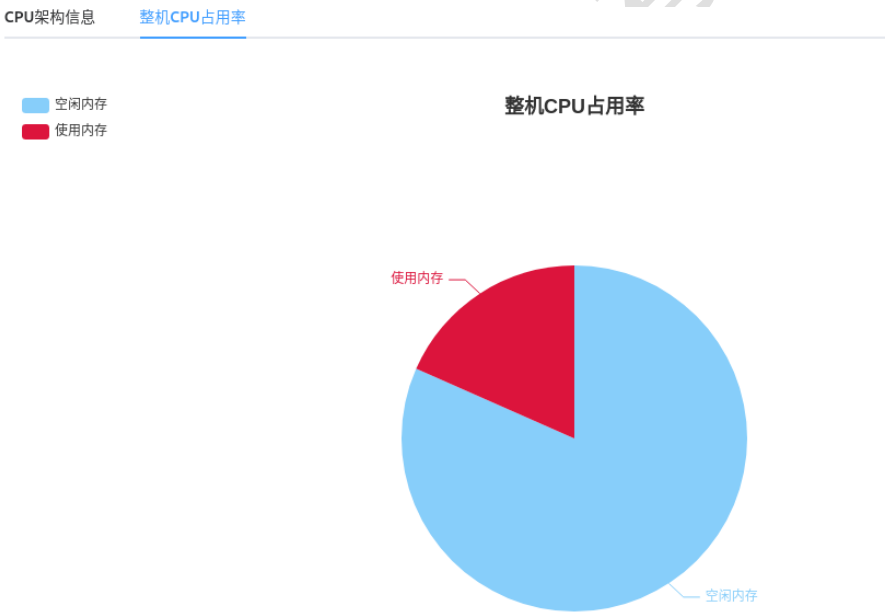


图 134 整机 CPU 占用率

6.3.2 进程 CPU 占用分析

功能介绍：对不同进程 CPU 占用情况进行实时监控。

操作步骤：

1、CPU 性能分析。

选择基于 PMU 的性能分析下的【进程 CPU 占用分析】选项，进入该插件。

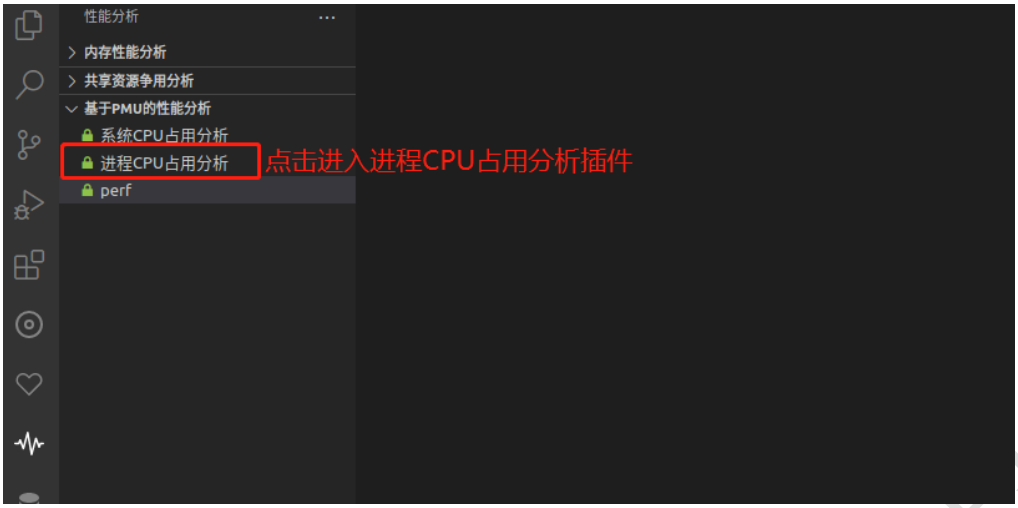


图 135 进程 CPU 占用分析功能进入

2、折线图显示进程 CPU 信息。

(1) 点击【选择进程】按钮。

图 136 进程 CPU 占用分析功能查询

(2)弹出进程号相关列表，并可进行相关操作。

刷新按钮：点击【刷新列表】按钮，对进程号列表进行刷新。

搜索按钮：在文本框输入关键词，点击【搜索】按钮模糊匹配命令中的单词。

选择按钮：点击【选择】按钮，将进程号返回到页面。

(3) 输入进程号，点击【确定】按钮，以折线图形式每三秒刷新一次该进程 CPU 占用率。

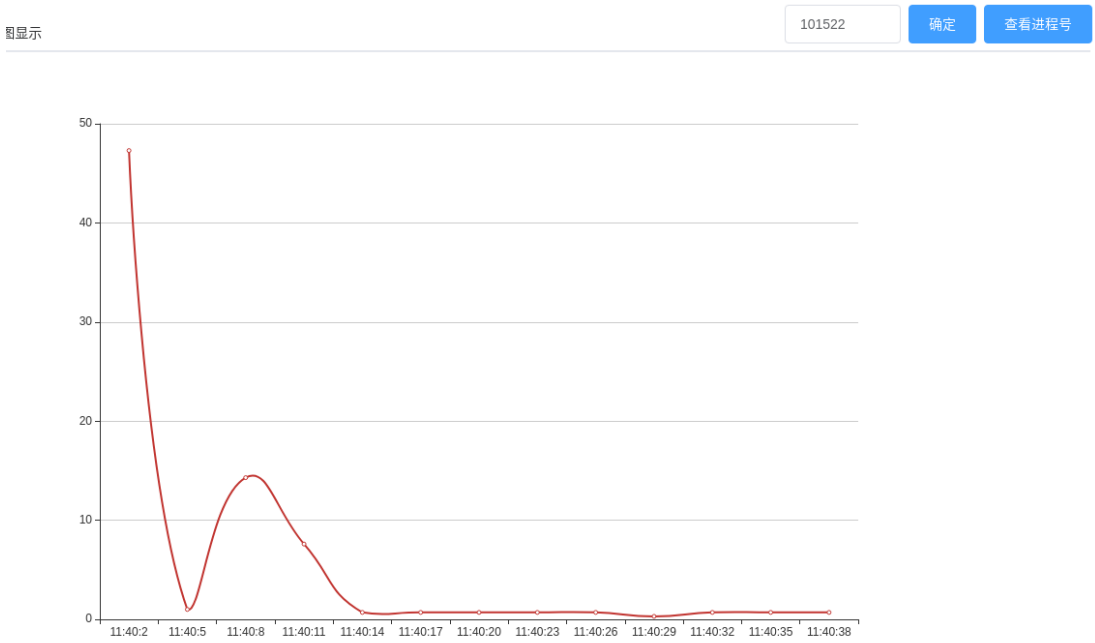


图 137 进程 CPU 占用信息

3、饼状图显示进程 CPU 信息

点击以饼状图显示，会切换到饼状图显示页面，并保持每三秒钟刷新一次的频率。

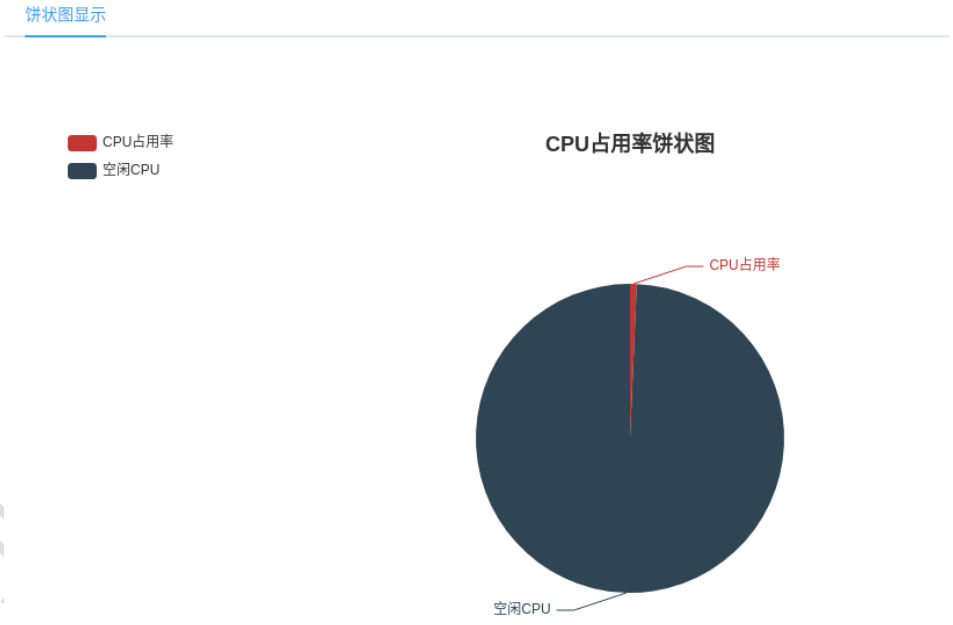


图 138 进程 CPU 占用率饼图