Contents

Proj158-支持 Rust 语言的源代码级内核调试工具

cyber$_r ose$

July 31, 2024

Abstract

code-debug 是一款同时支持 Rust 语言和 C 语言操作系统内核开发的源代码级调试工具，该工具基于 QEMU 和 GDB，支持跨内核态和用户态的源代码跟踪调试；基于 eBPF 支持开发板上跨内核态和用户态的性能分析检测；基于 VScode 构建了远程开发环境，支持断点调试与性能检测的功能结合。

1 完善调试器

1.1 解决由调试器自动设置的断点不会在 VSCode 里面显示出来的问题

在过去，由于 VSCode 没有提供“在 VSCode 中设置断点”的 API，我们的插件无法模拟用户设置断点的操作。我们如果想要自动地设置断点，只能从 Debug Adapter 入手，让 Debug Adapter 知道断点设置了，然后再告诉 GDB，但是 VSCode 是根本不知道这个断点存在的，因此不会显示出来。现在，VSCode 在某个更新中增加了“在 VSCode 中设置断点”的 API，我们的插件可以利用这个 API 来模拟用户设置断点的操作，这样 VSCode 知道了断点的存在，断点就可以了显示出来了。

1.2 完善边界断点

去年的工作将边界断点独立于断点组，若内核的出入口断点均在内核的符号表里，在用户态运行时内核的符号表已经卸载，无法触发边界断点回到内核态。我们的解决方法是直接设置边界断点，所有地方都用完整的文件路径来解决断点组名字和断点文件名的对应的问题。在实际实现时将边界断点包含在断点组属性中，支持动态设置和取消（但是逻辑上仍然独立于断点组）。我们之前把 isBorder 这个属性交给断点组管理模块去管理。因此，在 Debug Adapter 层面是先设置断点，再给断点添加“边界”属性（在 VSCode 的层面则不是）。

```
//this will go through setBreakPointsRequest in mibase.ts
vscode.debug.addBreakpoints([breakpoint]);
vscode.debug.activeDebugSession?.customRequest('setBreakpointAsBorder',args[0]);
```

由于 Debug Adapter 会对这些断点做很多的操作，把断点组和 Debug Adapter 本身的断点管理功能合为一体不是好事，怕会造成更多麻烦。之前在断点组里面直接存 SetBreakpointArguments 的策略没有问题，因为断点组管理模块的作用就是在合适的时机进行断点设置，而非存储某个断点。而且 SetBreakpointArguments 里面已经包含了断点所需要的所有信息。只不过我们要继承

SetBreakpointArguments，添加一个 isBorder 属性。然后通过 customRequest 对这个 isBorder 属性做更改，非常麻烦。

我们现在不把边界的信息并存储在断点的数据结构里，而是存在断点组的数据结构里，这样就不用去查找到某个断点的数据结构，再将它改为“边界”。这样做还有一个好处，无需改动原有的断点数据结构（因为边界的信息不再存储在断点的数据结构中，而是存在断点组的属性中）。再增加一个“去除本地址空间的边界断点”功能，就同时实现了边界断点的更改。断点组切换的代码除了完全清空所有断点组信息（removeallclibreakpoint）的情况外，断点组本身是不会被删除的（断点组里面的断点可能会被删除）。因此我们把边界的信息附加在断点组上，做到了和之前代码的兼容，因此代码量小，现有的断点组切换的代码完全不需要更改。

```
// 每个断点组只能有一个边界断点。因此，如果在同一个断点组中设置两次边界断点，
//新的边界断点会替换旧的
const setBreakpointAsBorderCmd = vscode.commands.registerCommand
('code-debug.setBreakpointAsBorder', (...args) => {
const uri = args[0].uri;
const fullpath = args[0].uri.fsPath; // fsPath 提供了适用于操作系统的路径格式
const lineNumber = args[0].lineNumber;
// 我们将行索引设置为 0，因为目前不需要处理行内的位置
let breakpoint = new vscode.SourceBreakpoint(new vscode.Location
                    (uri,new vscode.Position(lineNumber,0)),true);
//这将会通过 mibase.ts 中的 setBreakPointsRequest
vscode.debug.addBreakpoints([breakpoint]);
vscode.debug.activeDebugSession?.customRequest('setBorder',new Border(fullpath,lineNumber));
});
```

此时设置边界的逻辑变得更加简单——接收一个包含边界信息的参数，并调用 updateBorder 方法更新断点组的边界。

```
//customRequest=======
            case 'setBorder':
            // args have border type
            this.breakpointGroups.updateBorder(args as Border);
            break;
```

更新边界断点:

根据给定文件路径的边界断点，更新/创建对应的断点组。首先通过 eval 执行函数 filePathToBreakpointGroupNames 获取与文件路径关联的断点组名称列表，然后遍历这些组名，检查每个组名是否存在于当前的断点组列表中。如果找到匹配的断点组，则更新其边界属性；如果未找到，则创建一个新的断点组并将其添加到列表中。确保每个文件路径的边界断点都能正确地归属到相应的断点组中。

```
public updateBorder(border: Border) {
      const result =
      eval(this.debugSession.filePathToBreakpointGroupNames)(border.filepath);
```

```
        const groupNamesOfBorder: string[] = result;
        for (const groupNameOfBorder of groupNamesOfBorder) {
            let groupExists = false;
            for (const group of this.groups) {
                if (group.name === groupNameOfBorder) {
                    groupExists = true;
                    // 注意：这里假设每个组只有一个 border，如果需要支持多个边界断点，应更改为 grou
                    group.border = border;
                }
            }

            // 如果没有找到匹配的断点组，则创建一个新的断点组并添加到列表中
            if (groupExists === false) {
                this.groups.push(new BreakpointGroup(
                    groupNameOfBorder,
                    [],
                    new HookBreakpoints([]),
                    border // 注意：这里传入单个边界断点，如果需要支持多个边界断点，应更改为 [borde
                ));
            }
        }
    }
```

除此之外我们加了一个把边界断点改回普通断点的功能：

```
public disableBorder(border: Border) {
    const groupNamesOfBorder:string[] =
                    eval(this.debugSession.filePathToBreakpointGroupNames)(border.filepath);
    for(const groupNameOfBorder of groupNamesOfBorder){
        let groupExists = false;
        for(const group of this.groups){
            if(group.name === groupNameOfBorder){
                groupExists = true;
                group.border = undefined;
            }
        }
        if(groupExists === false){
            //do nothing
        }
    }
}
```

如果没给边界的话就不会切换断点组，就在当前断点组一直运行下去。

1.3 将断点组功能改造为状态机

由于之前代码之前散落在各处，没有可读性，而且许多代码实现起来很是复杂，我们决定将之前的代码重构，用状态机的形式来更清晰的描述行为和状态变化。我们构造的状态机只管理 extension.ts，任何 mibase.ts 的操作都要提到 extension.ts 来完成。因为我们的调试器本质上是模拟用户操作，而用户操作的相关逻辑就是在 extension.ts 里实现的。

我们的做法是：维持原有的断点组，将状态机作为断点组上层的东西。首先，调试器启动并初始化状态机，监听并处理各种事件，如程序停止、断点触发等。当事件发生时，stopEvent 方法被调用，并触发状态机中定义的动作。接着，OSStateTransition 方法根据当前状态和事件确定新状态和应执行的动作。doAction 方法执行这些动作，可能导致状态机状态的改变或执行其他调试器操作。如果到达特定的边界或条件，状态机将更新其状态，并可能触发新的动作。完成后，状态机继续监听和响应事件，直到调试会话结束。

1.3.1 使用 enum 总结所有我们要实现的功能

状态机通过定义不同的状态和事件来管理调试器的行为。OSStates 枚举列出了所有可能的状态，如内核态和用户态，以及它们之间的单步执行状态。OSEvents 枚举定义了触发状态转换的事件，例如程序停止、到达内核或用户态边界等。DebuggerActions 枚举定义了状态机可以执行的各种动作，如检查是否到达特定的内存地址区间、开始连续单步执行、获取下一个断点组名称等。

```
enum OSStates { //定义了内核、用户态以及单步执行状态。
        kernel,
        kernel_single_step_to_user,
        user,
        user_single_step_to_kernel,
}

enum OSEvents { //定义了调试过程中可能遇到的事件，如停止、到达内核/用户边界等。
        STOPPED,
        AT_KERNEL,
        AT_KERNEL_TO_USER_BORDER,
        AT_USER,
        AT_USER_TO_KERNEL_BORDER,
}

enum DebuggerActions { //定义了状态机响应事件时可能执行的动作。
        check_if_kernel_yet,
        check_if_user_yet,
        check_if_kernel_to_user_border_yet,
        check_if_user_to_kernel_border_yet,
        get_next_breakpoint_group_name,
        start_consecutive_single_steps,
        switch_breakpoint_group,
```

```
}
```

状态机里面将“自动单步”这些自动化操作表示为”actions”（类型为 Actions[]，类似于 iOS 的快捷指令）。

1.3.2 添加“钩子断点”

钩子断点允许在调试过程中的特定点执行额外的逻辑。HookBreakpoint 类存储了断点和与之关联的行为（behavior）。当调试器在该断点停止时，将执行定义的行为。

1.3.3 改变状态触发事件

- 钩子断点

  状态机通过监听调试器事件（如停止事件）来触发状态改变。stopEvent 方法处理停止事件，并根据当前状态和事件触发状态机中定义的动作。之前为了提供“停下”的信号，在四五个地方（断点触发，单步结束.....）执行后发送“停下”的事件。但是我们发现，stopEvent 确实会在每次 OS 停下来时被创建，只处理一种停下来的情况，不包括因为断点而停下来的情况。所以只要在 stopevent 一个地方设 stop 的“钩子断点”即可。

  ```
  // 获取下一进程名
  class HookBreakpoint{
      breakpoint:Breakpoint;
      behavior:FunctionString;
      constructor(breakpoint:Breakpoint, behavior:FunctionString){
          this.breakpoint = breakpoint;
          this.behavior = behavior;
      }
  }
  ```

  停下时 STOPPED 事件发生，根据状态机，会触发

  ```
  try_get_next_breakpoint_group_name action
  ```

  这个 action 的作用是，判断当前是否停在了 HookBreakpoint.breakpoint 上了。如果是的话，调用 behavior 做信息收集的工作，把结果返回即可。利用 Function 构造函数，让用户可以把函数传进来：

  ```
  //JSON: {"function":{"arguments":"a,b,c","body":"return a*b+c;"}}
  var f = new Function(function.arguments, function.body);
  ```

  这要求用户掌握我们 mibase.ts 里面的 API。但是这种设计的可扩展性很强，比较通用，以后要想完善断点组切换功能的话，只需要定义新的 behavior 和 environment 就可以。

- 状态改变具体逻辑如下：状态机通过监听调试器事件(如停止事件)来触发状态改变。stopEvent 方法处理停止事件，并根据当前状态和事件触发状态机中定义的动作。状态机的初始状态是”kernel”。当停下时，改变状态。

```
    /**
     * 处理调试器停止事件
     * @param {MINode} info - 包含停止事件信息的 MINode 对象
     */
    stopEvent(info: MINode) {
        if (!this.started) {
            this.crashed = true;
        }
        if (!this.quit) {
            const event = new StoppedEvent("exception", parseInt(info.record("thread-id
            (event as DebugProtocol.StoppedEvent).body.allThreadsStopped =
                info.record("stopped-threads") == "all";
            this.sendEvent(event);

            // 检查操作系统调试是否就绪
            if (this.OSDebugReady) {
                this.recentStopThreadID = parseInt(info.record("thread-id"));
                this.OSStateTransition(new OSEvent(OSEvents.STOPPED));
            }
        }
    }
```

this.OSStateTransition 方法做如下的事:

1. 通过 stateTransition 查询新状态和应做的 actions 2. 更新状态 this.OSState, 执行 actions

```
  public OSStateTransition(event: OSEvent){
            let actions:Action[];
            [this.OSState, actions] = stateTransition(this.OSStateMachine, this.OSState, ev
            // go through the actions to determine
            // what should be done
            actions.forEach(action => {this.doAction(action);});
        }
```

根据状态机，kernel 状态时触发 STOPPED event . 这种情况不会改变状态（或者说状态改变到 kernel 本身），但是会触发

```
check_if_kernel_to_user_border_yet action
```

```
  [OSEvents.STOPPED]: {
                        target: OSStates.kernel,
                        actions: [
                            { type: DebuggerActions.try_get_next_breakpoint_group_name }, /
                            { type: DebuggerActions.check_if_kernel_to_user_border_yet }, /
```

```
                ]
            },
```

我们这个状态机的特别之处在于：一些 action 会导致 OSState 的变化。特殊的情况全部出现在 doAction 方法中：在这个 doAction 方法中，每个 else if 分支都对应一个动作类型。这些动作包括检查当前程序计数器（PC）的值以确定是否到达特定的内存地址区间、开始连续单步执行、获取下一个断点组名称、以及在不同层级的断点组之间进行切换。每个分支都以显示一条信息开始，然后根据动作类型执行相应的逻辑。例如，在检查是否到达内核态或用户态的分支中，首先获取程序计数器的值，然后根据地址区间判断当前是否处于期望的状态，并相应地触发状态机事件或执行单步指令。在尝试获取下一个断点组名称的分支中，遍历当前断点组的钩子断点，如果当前位置与钩子断点匹配，则执行钩子行为并更新下一个断点组名称。在断点组切换的分支中，根据当前状态更新当前断点组和下一个断点组的名称，以便在不同层级的断点组之间进行切换。

```
// 检查是否到达内核态
        public doAction(action: Action) {
            if (action.type === DebuggerActions.check_if_kernel_yet) {
                this.showInformationMessage('doing action: check_if_kernel_yet');
                this.miDebugger.getSomeRegisters([this.program_counter_id]).then(v => {
                    const addr = parseInt(v[0].valueStr, 16);
                    if (this.isKernelAddr(addr)) {
                        this.showInformationMessage('arrived at kernel. current addr:' + ad
                        this.OSStateTransition(new OSEvent(OSEvents.AT_KERNEL));
                    } else {
                        this.miDebugger.stepInstruction();
                    }
                });
            }

            else if (action.type === DebuggerActions.check_if_user_yet) {
                this.showInformationMessage('doing action: check_if_user_yet');
                this.miDebugger.getSomeRegisters([this.program_counter_id]).then(v => {
                    const addr = parseInt(v[0].valueStr, 16);
                    if (this.isUserAddr(addr)) {
                        this.showInformationMessage('arrived at user. current addr:' + addr
                        this.OSStateTransition(new OSEvent(OSEvents.AT_USER));
                    } else {
                        this.miDebugger.stepInstruction();
                    }
                });
            }

    // 检查是否到达从内核态到用户态的边界
```

```
        else if (action.type === DebuggerActions.check_if_kernel_to_user_border_yet) {
            this.showInformationMessage('doing action: check_if_kernel_to_user_border_y
            const kernelToUserBorderFile = this.breakpointGroups.getCurrentBreakpointGr
            const kernelToUserBorderLine = this.breakpointGroups.getCurrentBreakpointGr
            this.miDebugger.getStack(0, 1, this.recentStopThreadID).then(v => {
                let filepath = v[0].file;
                let lineNumber = v[0].line;
                if (filepath === kernelToUserBorderFile && lineNumber === kernelToUserB
                    this.OSStateTransition(new OSEvent(OSEvents.AT_KERNEL_TO_USER_BORDE
                }
            });
        }
```

检查是否到达从用户态到内核态的边界与检查内核到用户边界类似，此处省略。

```
    // 开始连续单步执行
        else if (action.type === DebuggerActions.start_consecutive_single_steps) {
            this.showInformationMessage("doing action: start_consecutive_single_steps")
            this.miDebugger.stepInstruction();
        }

    // 尝试获取下一个断点组名称
        else if (action.type === DebuggerActions.try_get_next_breakpoint_group_name) {
            this.showInformationMessage('doing action: try_get_next_breakpoint_group_na
            // 获取当前栈帧信息
            this.miDebugger.getStack(0, 1, this.recentStopThreadID).then(v => {
                let filepath = v[0].file;
                let lineNumber = v[0].line;
                // 遍历当前断点组的钩子断点
                for (const hook of this.breakpointGroups.getCurrentBreakpointGroup().ho
                    if (filepath === hook.breakpoint.file && lineNumber === hook.breakp
                        // 执行钩子行为，并更新下一个断点组名称
                        eval(hook.behavior)().then((hookResult: string) => {
                            this.breakpointGroups.setNextBreakpointGroup(hookResult);
                            this.showInformationMessage('finished action: try_get_next_
                        });
                    }
                }
            });
        }
    // 从高层断点组切换到低层断点组
        else if (action.type === DebuggerActions.high_level_switch_breakpoint_group_to_
            // 获取当前高层断点组名称，并更新为低层断点组
```

```
                const high_level_breakpoint_group_name = this.breakpointGroups.getCurrentBr
                this.breakpointGroups.updateCurrentBreakpointGroup(this.breakpointGroups.ge
                // 设置下一个断点组为高层断点组，以便在低层执行完成后返回
                this.breakpointGroups.setNextBreakpointGroup(high_level_breakpoint_group_na
            }
        }
```

从低层断点组切换到高层断点组与高层到低层切换类似，此处省略。

在构造了状态机 OSStateMachine 和状态机的转换方法 OSStateTransition 之后，我们不再需要手动更新特权级和 “特权级改变过” 的 flag 了。我们只需要获取 PC 寄存器，判断它的区间，如果是所需区间，激活 “到达 xx 特权级” 的事件即可。

获取数据过去是在断点触发的时候判断这个断点是不是边界，现在由于状态机的表述方式不同，改为需要判断 “我在哪里” 时，发送一个单独的 GDB 命令来获得这个信息。在 google 上找到 gdb 命令（where），查找文档得到对应的 MI 命令（-stack-list-frames），查找代码发现插件已经有了一个得封装很好的实现了（getStack），因此不需要像之前那样用硬编码的方式从 MINode 里获取数据了。这个 stack 的 stack frame 指的是函数的调用栈，和页帧无关。我们只要取最上一层的 frame 0 即可。

```
    /**
     * 检查是否到达从内核态到用户态的边界
     * @description 此动作用于确定程序执行是否到达了从内核态到用户态的边界点。
     * 如果到达边界，将触发状态机转换事件，以便进行相应的处理。
     */
    else if (action.type === DebuggerActions.check_if_kernel_to_user_border_yet) {
        // 显示当前执行的动作信息
        this.showInformationMessage('doing action: check_if_kernel_to_user_border_yet');
        let filepath: string = "";
        let lineNumber: number = -1;
        const kernelToUserBorderFile = this.breakpointGroups.getCurrentBreakpointGroup().bor
        const kernelToUserBorderLine = this.breakpointGroups.getCurrentBreakpointGroup().bor

        // 获取当前线程的调用栈信息
        // 这里使用 miDebugger.getStack 获取栈顶信息，参数 0 表示从栈顶开始获取，1 表示只获取一
        // this.recentStopThreadID 是最近一次停止的线程 ID
        // 如果要进行多核调试，可能需要修改第三个参数以适应不同线程的情况
        this.miDebugger.getStack(0, 1, this.recentStopThreadID).then(v => {
            // 从获取的栈帧信息中提取文件路径和行号
            filepath = v[0].file;
            lineNumber = v[0].line;

            // 检查当前位置是否与边界文件路径和行号匹配
            // 如果匹配，则表示已经到达内核态到用户态的边界
```

```
                if (filepath === kernelToUserBorderFile && lineNumber === kernelToUserBorderLine
                    // 触发到达内核到用户边界的事件，状态机将根据此事件进行状态转换
                    this.OSStateTransition(new OSEvent(OSEvents.AT_KERNEL_TO_USER_BORDER));
                }
            });
        }
```

由于执行环境的隔离性，底层可以知道接下来要去哪个顶层的断点组，但是反过来不行。我们的解决办法是：底层转到高层时，就预先假设高层会回到这个底层。在高层运行期间不去获取接下来要回到哪个底层，因为根本获取不到。

所以，

`switch_breakpoint_group`

变为

`high_level_switch_breakpoint_group_to_low_level`

和

`low_level_switch_breakpoint_group_to_high_level`

。内核转换到用户态时用

`low_level_switch_breakpoint_group_to_high_level`

，包含了指定 nextbreakpointgroup 为内核的行为；而用户态切换回内核时用的

`high_level_switch_breakpoint_group_to_low_level`

则没有：

```
    else if(action.type === DebuggerActions.high_level_switch_breakpoint_group_to_low_level){//
                const high_level_breakpoint_group_name = this.breakpointGroups.getCurrentBreakpo
                this.breakpointGroups.updateCurrentBreakpointGroup(this.breakpointGroups.getNext
                this.breakpointGroups.setNextBreakpointGroup(high_level_breakpoint_group_name);/
            }
```

内核转换到用户态时用

`low_level_switch_breakpoint_group_to_high_level`

，而用户态切换回内核时用

`high_level_switch_breakpoint_group_to_low_level`

：状态机 OSStateMachine 来管理不同调试状态下的行为和转换。每个状态都对应一组事件处理器，当特定事件发生时，将触发预定义的动作，并可能转换到新的状态。这些状态和事件共同控制着调试器的行为，使得调试过程可以根据程序的执行状态进行适当的响应和调整。

```
    // 定义状态机
export const OSStateMachine: OSStateMachine = {
    initial: OSStates.kernel,
    states: {
        [OSStates.kernel]: {
            on: {
                [OSEvents.STOPPED]: {
                    target: OSStates.kernel,
                    actions: [
                        { type: DebuggerActions.try_get_next_breakpoint_group_name },
                        // 检查是否到达内核态到用户态的边界，如果是，
                        //则触发 AT_KERNEL_TO_USER_BORDER 事件
                        { type: DebuggerActions.check_if_kernel_to_user_border_yet },
                    ],
                },
                // 当内核态到达内核到用户边界时
                [OSEvents.AT_KERNEL_TO_USER_BORDER]: {
                    // 目标状态变为单步执行到用户态（kernel_single_step_to_user）
                    target: OSStates.kernel_single_step_to_user,
                    actions: [
                        // 开始连续单步执行
                        { type: DebuggerActions.start_consecutive_single_steps },
                    ],
                },
            },
        },
        // 单步执行到用户态（kernel_single_step_to_user）相关的行为
        [OSStates.kernel_single_step_to_user]: {
            on: {
                [OSEvents.STOPPED]: {
                    target: OSStates.kernel_single_step_to_user,
                    actions: [
                        // 检查是否到达用户态，如果是，则触发 AT_USER 事件
                        // 如果没有到达用户态，继续单步执行
                        { type: DebuggerActions.check_if_user_yet },
                    ],
                },
                [OSEvents.AT_USER]: {
                    target: OSStates.user,
                    actions: [
                        // 从低层断点组切换到高层断点组
```

```
                    // 包括边界断点在内的断点组，同时切换调试符号文件
                // 断点组改变后，设置下一个断点组为内核的断点组
                    { type: DebuggerActions.low_level_switch_breakpoint_group_to_high_level
                ],
            },
        },
    },
```

用户态（user）相关的行为和单步执行到内核态（user$_s ingle_s tep_t o_k ernel$

由于状态机比较简陋，这种实现还是有不完美的地方：

一些行为没有放到状态机里面（但是全部都在 doAction 方法里）依赖 recentStopThreadID. 这个数据是在状态机之外的 StopEvent 方法里更新的。后续可以继续改进

- 符号表文件的切换符号表文件和断点组在 ebpf(rCore-Tutorial-v3) 里是一对一的，但是其他 OS 就不能保证了。而且，ebpf 的内核和用户符号表有时可以共存，有时不行，不知道在其他 OS 上是什么样子，因此符号表文件随着断点组切换而切换的逻辑作为用户提交代码。之前的做法是添加新符号表-移除旧断点-打新断点根本没有去除符号表。之前内核的符号表是不删去的，这样其实不完善。

  由于断点是依赖符号表的，合理的顺序应该是移除旧断点-移除旧符号表-添加新符号表-添加新断点。如果是这个顺序的话，符号表切换的逻辑就得放在断点组切换的函数里面，不能单列一个函数了。因此我们不能把整个符号表切换的逻辑抽离出来作为用户自定义代码。我们只能将断点组 => 符号表文件路径的映射作为自定义代码。

  进行了以下修改：之前 addDebugSymbol 和 removeDebugSymbol 是直接在 mibase.ts 里调用 sendCliCommand，这次放到 mi2.ts 里：

```
/**
 * 添加符号文件到调试器
 * @param {string} filepath - 符号文件的路径
 * @returns {Thenable<any>} - 表示异步操作的 Promise 对象
 */
addSymbolFile(filepath: string): Thenable<any> {
    // 如果启用了跟踪日志，记录函数调用
    if (trace) this.log("stderr", "addSymbolFile");

    // 返回一个新的 Promise 对象
    return new Promise((resolve, reject) => {
        // 存储将要执行的 Promise 对象数组
        const promises: Thenable<void | MINode>[] = [];

        // 将添加符号文件的命令添加到数组中
        promises.push(
            // 发送 "add-symbol-file" 命令到调试器
```

```
                this.sendCliCommand("add-symbol-file " + filepath).then((result) => {
                    // 如果命令执行成功，并且结果类别为 "done"，则解析 Promise
                    if (result.resultRecords.resultClass == "done") resolve(true);
                    // 否则，拒绝 Promise
                    else resolve(false);
                })
            );

            // 使用 Promise.all 等待所有命令执行完成
            // 所有命令成功完成时调用 resolve，如果任何一个命令失败则调用 reject
            Promise.all(promises).then(resolve, reject);
        });
    }

    /**
     * 从调试器中移除符号文件
     * @param {string} filepath - 符号文件的路径
     * @returns {Thenable<any>} - 表示异步操作的 Promise 对象
     */
    removeSymbolFile(filepath: string): Thenable<any> {
        if (trace) this.log("stderr", "removeSymbolFile");
        return new Promise((resolve, reject) => {
            const promises: Thenable<void | MINode>[] = [];

            // 将移除符号文件的命令添加到数组中
            promises.push(
                // 发送 "remove-symbol-file" 命令到调试器
                this.sendCliCommand("remove-symbol-file " + filepath).then((result) => {
                    // 如果命令执行成功，并且结果类别为 "done"，则解析 Promise
                    if (result.resultRecords.resultClass == "done") resolve(true);
                    else resolve(false);
                })
            );

            // 使用 Promise.all 等待所有命令执行完成
            // 所有命令成功完成时调用 resolve，如果任何一个命令失败则调用 reject
            Promise.all(promises).then(resolve, reject);
        });
    }
```

断点组切换时，符号表也切换：

缓存旧空间的断点，令 GDB 清除旧断点组的断点，卸载旧断点组的符号表文件，加载新断

点组的符号表文件，加载新断点组的断点

```
public updateCurrentBreakpointGroup(updateTo: string) {
        let newIndex = -1;
        for (let i = 0; i < this.groups.length; i++) {
            if (this.groups[i].name === updateTo) {
            newIndex = i;
            }
        }
        if (newIndex === -1) {
            this.groups.push(new BreakpointGroup(updateTo, [], new HookBreakpoints([]),
            newIndex = this.groups.length - 1;
        }
        let oldIndex = -1;
        for (let j = 0; j < this.groups.length; j++) {
            if (this.groups[j].name ===
            this.getCurrentBreakpointGroupName()) {
            oldIndex = j;
            }
        }
        if (oldIndex === -1) {
            this.groups.push(new BreakpointGroup(this.getCurrentBreakpointGroupName(),
            oldIndex = this.groups.length - 1;
        }
        this.groups[oldIndex].setBreakpointsArguments.forEach((e) => {
            this.debugSession.miDebugger.clearBreakPoints(e.source.path);
});
    this.debugSession.miDebugger.removeSymbolFile(eval(this.debugSession.breakpointGrou
    this.debugSession.miDebugger.addSymbolFile(eval(this.debugSession.breakpointGroupNa

    this.groups[newIndex].setBreakpointsArguments.forEach((args) => {
        this.debugSession.miDebugger.clearBreakPoints(args.source.path).then(
        () => {
                    let path = args.source.path;
                    if (this.debugSession.isSSH) {
                    // convert local path to ssh path
                    path =
                    this.debugSession.sourceFileMap.toRemotePath(path);
                    }
                    const all = args.breakpoints.map((brk) => {
                        return this.debugSession.miDebugger.addBreakPoint({
                            file: path,
```

```
                                line: brk.line,
                                condition: brk.condition,
                                countCondition: brk.hitCondition,
                            });
                            });
                        },
                    (msg) => {
                        //TODO
                    }
                );
                });
                this.currentBreakpointGroupName = this.groups[newIndex].name;
                this.debugSession.showInformationMessage("breakpoint group changed
                to " + updateTo);
            }
```

不应该有一个单独的 setCurrentBreakpointGroupName() 函数，因为更改 currentGroupName 的同时也需要更改断点组本身，而这正是 updateCurrentBreakpointGroup() 函数的职责所在。需要特别注意的是边界可能会返回 undefined。

```
    public getCurrentBreakpointGroupName():string {
        return this.currentBreakpointGroupName;
    }
    // notice it can return undefined
    public getBreakpointGroupByName(groupName:string){
        for (const k of this.groups){
            if (k.name === groupName){
                return k;
            }
        }
        return;
}
    // notice it can return undefined
    public getCurrentBreakpointGroup():BreakpointGroup{
        const groupName = this.getCurrentBreakpointGroupName();
        for (const k of this.groups){
            if (k.name === groupName){
                return k;
            }
        }
            return;
    }
    public getNextBreakpointGroup(){
```

```
        return this.nextBreakpointGroup;
    }
    public setNextBreakpointGroup(groupName:string){
        this.nextBreakpointGroup = groupName;
    }
    public getAllBreakpointGroups():readonly BreakpointGroup[]{
        return this.groups;
    }
```

将断点信息保存到断点组中，但不是让 GDB 立即设置这些断点。即在保存断点信息时，断点不会立即被 GDB 应用或激活，而是先进行存储以供后续处理。

```
    public saveBreakpointsToBreakpointGroup(args:
    DebugProtocol.SetBreakpointsArguments, groupName: string) {
            let found = -1;
            for (let i = 0; i < this.groups.length; i++) {
                if (this.groups[i].name === groupName) {
                    found = i;
                }
            }
            if (found === -1) {
                this.groups.push(new BreakpointGroup(groupName, [], new
                HookBreakpoints([]), undefined));
                found = this.groups.length - 1;
            }
            let alreadyThere = -1;
            for (let i = 0; i <
            this.groups[found].setBreakpointsArguments.length; i++) {
                if (this.groups[found].setBreakpointsArguments[i].source.path
                === args.source.path) {
                    this.groups[found].setBreakpointsArguments[i] = args;
                    alreadyThere = i;
                }
            }
            if (alreadyThere === -1) {
                this.groups[found].setBreakpointsArguments.push(args);
            }
    }
```

- 文件路径在实现 filePathToBreakpointGroupNames 和 breakpointGroupNameToDebugFilePath 的时候，我们发现 filePathToBreakpointGroupNames 在使用后是需要做失败处理的，因为用户态程序运行到用户库的时候，由于所有应用程序都共享同一份用户库代码，这种情况下，一个断点会同时属于多个断点组。比如 rCore-Tutorial-v3（ebpf）里 user/src/syscall.rs 里

的断点就属于所有用户态程序的断点组，这时，应返回所有用户程序断点组。

breakpointGroupNameToDebugFilePath 不会有这个问题。

filePathToBreakpointGroupNames 函数：

```
"filePathToBreakpointGroupNames": {
        "type": "object",
        "description": "user-submitted js code used to turn a filepath
        into breakpoint group name",
        "default": {
                "isAsync": false,
                "functionArguments": "filePathStr",
                functionBody": "     if (filePathStr.includes('os/src'))
                {       return ['kernel'];    }
                else if (filePathStr.includes('user/src/bin'))
                {       return [filePathStr];    }
                else if (!filePathStr.includes('user/src/bin') &&
                filePathStr.includes('user/src'))
                {       return ['${workspaceFolder}/user/src/bin/adder_atomic.rs',
                '${workspaceFolder}/user/src/bin/adder_mutex_blocking.rs',
                '${workspaceFolder}/user/src/bin/adder_mutex_spin.rs',
                '${workspaceFolder}/user/src/bin/adder_peterson_spin.rs',
                '${workspaceFolder}/user/src/bin/adder_peterson_yield.rs',
                '${workspaceFolder}/user/src/bin/adder.rs',
                '${workspaceFolder}/user/src/bin/adder_simple_spin.rs',
                '${workspaceFolder}/user/src/bin/adder_simple_yield.rs',
                '${workspaceFolder}/user/src/bin/barrier_condvar.rs',
                '${workspaceFolder}/user/src/bin/barrier_fail.rs',
                '${workspaceFolder}/user/src/bin/cat.rs',
                '${workspaceFolder}/user/src/bin/cmdline_args.rs',
                '${workspaceFolder}/user/src/bin/condsync_condvar.rs',
                '${workspaceFolder}/user/src/bin/condsync_sem.rs',
                '${workspaceFolder}/user/src/bin/count_lines.rs', '${workspaceFolder}/u
                }
                },
```

注意，如果确实找不到断点空间（比如一些在 GDB 里面开头是/rust 的库文件），就把它归为内核的断点。把用到 filePathToSpaceNames 的地方都改成能够处理一个断点对应多个断点组的情况了。如果删除这种“一对多”的断点的话也不用担心，因为并不存在删除断点的动作，而是 args 参数传来那个文件对应的所有断点，然后那一个文件对应的所有断点都被清掉，全部重新设置一遍。断点组管理模块也是这样。

```
// 用于设置某一个文件的所有断点
protected override setBreakPointsRequest(response:
```

```
DebugProtocol.SetBreakpointsResponse, args:
DebugProtocol.SetBreakpointsArguments): void {
            let path = args.source.path;
            if (this.isSSH) {
                path = this.sourceFileMap.toRemotePath(path);
^^I^^I}
            //先清空该文件内的断点，再重新设置所有断点
            this.miDebugger.clearBreakPoints(path).then(() => {
                let spaceNames:string[] = this.filePathToSpaceNames(path);
                let currentSpaceName = this.addressSpaces.getCurrentSpaceName();
                //保存这些断点信息到断点所属的断点组（可能不止一个）里
                for(let spaceName in spaceNames){
                    this.addressSpaces.saveBreakpointsToSpace(args, spaceName);
                }
                //注意，此时断点组管理模块里已经有完整的断点相关的信息了
                let flag = false;
                for(let spaceName in spaceNames){
                    if(spaceName===currentSpaceName) { flag = true; }
                }
                //如果这些断点所属的断点组和当前断点组没有交集，比如还在内核态时就设置用户态的
                if(flag===true) return;

                //反之，如果这些断点所属的断点组中有一个就是当前断点组，那么就通知 GDB 立即设置
                const all = args.breakpoints.map(brk => {
                    return this.miDebugger.addBreakPoint({ file: path, line:
                    brk.line, condition: brk.condition, countCondition:
                    brk.hitCondition });
                });
                //令 GDB 设置断点
                Promise.all(all).then(brkpoints => {
                    const finalBrks: DebugProtocol.Breakpoint[] = [];
                    brkpoints.forEach(brkp => {
                        // 目前返回的所有断点都标记为已验证，这会导致在损坏的 lldb 上出现已验证
                        if (brkp[0])
                            finalBrks.push(new DebugAdapter.Breakpoint(true, brkp[1].line))
                    });
                    response.body = {
                        breakpoints: finalBrks,
                    };
                        this.sendResponse(response);
                    },
```

```
                    (msg) => {
                        this.sendErrorResponse(response, 9, msg.toString());
                    }
                );
            },
            (msg) => {
                this.sendErrorResponse(response, 9, msg.toString());
            }
        );
    }
```

1.4 添加 showInformationMessage 函数，代替 mibase.ts 中无法使用的 console.log

使用 sendEvent() 方法将构造的事件对象发送给调试客户端，以展示信息提示给用户。

```
public showInformationMessage(info:string){
        this.sendEvent({
            event: "showInformationMessage",
            body: info,
        } as DebugProtocol.Event);
}
```

1.5 有的情况 continue 不能跳转到断点

之前在内核出口边界设用户态程序开头位置的断点，然后直接 continue 就可以跳转到这个断点。这是因为 rCore-Tutorial-v3 用了跳板页（详见 https://scpointer.github.io/rcore2oscomp/docs/lab2/gdb.html）。在没有跳板页，且是双页表的 OS 的情况下，这个策略不会起作用。

由于今年新实现了单步步进功能，我们可以通过不断的自动的单步（step instruction）每单步一次就查看内存地址来确定是否到达新的特权级。

1.6 提升 Debug Console 输出内容的可读性

我们在 mi2.ts 中定义了多个处理函数，用于接收和处理来自调试器（如 GDB）的标准输出（stdout）和标准错误（stderr）。

stdout 和 stderr 函数将接收到的数据追加到 buffer 和 errbuf 字符串中。这允许函数按行处理输出，而不是字符一个接一个地处理。当缓冲区中的字符串遇到换行符 () 时，将其按行分割并逐行处理。

onOutput 和 onOutputStderr 函数分别处理标准输出和标准错误，这允许对不同类型的输出进行定制化处理。

onOutput 函数检查每一行输出，判断它是否可能是调试器的输出（使用 couldBeOutput 函数）。对于调试器的输出，尝试解析为机器可读的格式（使用 parseMI 函数）。使用 log 和 logNoNewLine 函数将信息记录到控制台。log 函数在记录信息时会在末尾添加换行符，而 logNoNewLine

则不会。如果解析后的输出包含错误记录（parsed.resultRecords.resultClass == ”error”），则将错误信息记录到标准错误输出。

onOutput 函数还处理异步事件（如程序停止、断点命中等）。这些事件被转换为相应的动作，如发出自定义事件。如果启用了调试输出（this.debugOutput 为 true），则将解析后的调试信息以美化的 JSON 格式输出到控制台。

onOutputPartial 函数处理那些可能不完整的输出行，这有助于提升输出的实时性和可读性。对于没有 token 的 MI 节点，使用 originallyNoTokenMINodes 数组来存储，并在达到一定数量时进行裁剪，这有助于管理内存并防止潜在的内存泄漏。对于特定的停止原因（如断点命中、信号接收等），提供清晰的控制台消息，这有助于开发者快速理解程序的状态。使用 emit 函数发出自定义事件，这允许其他监听器响应这些事件并采取行动，如更新 UI 或状态指示器。

相关代码实现如下:

```
onOutput(str: string) {
 const lines = str.split('\n');
 lines.forEach(line => {
     // 判断当前行是否可能是调试器的输出
     if (couldBeOutput(line)) {
         // 如果当前行不是通过 gdbMatch 正则表达式匹配的特定格式，则记录为标准输出
         if (!gdbMatch.exec(line)) this.log("stdout", line);
     } else {
         // 解析当前行为 GDB 机器接口 (MI) 格式
         const parsed = parseMI(line);
         console.log("parsed:" + JSON.stringify(parsed));

         let handled = false; // 标记当前行是否已经被处理
         // 如果解析结果包含 token
         if(parsed.token !== undefined){
             // 如果存在对应的处理函数，则调用该函数并删除该 token 的记录
             if (this.handlers[parsed.token]) {
                 this.handlers[parsed.token](parsed);
                 delete this.handlers[parsed.token];
                 handled = true;
             }
             this.tokenCount = this.tokenCount + 1;
             parsed.token = this.tokenCount;
         }
         else{
             // 如果解析结果不包含 token，则分配一个 token
             parsed.token = this.tokenCount + 1;
             // 存储原始没有 token 的 MI 节点
             this.originallyNoTokenMINodes.push(parsed);
             // 如果存储的节点超过 100 个，则移除前面的 90 个
```

```
        if (this.originallyNoTokenMINodes.length >= 100) {
            this.originallyNoTokenMINodes.splice(0, 90);
            const rest = this.originallyNoTokenMINodes.splice(89);
            this.originallyNoTokenMINodes = rest;
        }
    }
    // 如果启用了调试输出，则记录美化后的 JSON 输出和原始解析结果
    if (this.debugOutput) {
        this.log("stdout", "GDB -> App: " + prettyPrintJSON(parsed));
        console.log("onoutput:" + JSON.stringify(parsed));
    }
    // 如果解析结果包含错误记录，则记录为标准错误输出
    if (!handled && parsed.resultRecords && parsed.resultRecords.resultClass == "error")
        this.log("stderr", parsed.result("msg") || line);
    }
    // 处理异步事件记录
    if (parsed.outOfBandRecord) {
        parsed.outOfBandRecord.forEach((record) => {
            // 根据记录类型进行相应处理
            if (record.isStream) {
                this.log(record.type, record.content);
            } else {
                // 处理不同类型的异步事件
                if (record.type == "exec") {
                    this.emit("exec-async-output", parsed);
                    // 发出不同的事件，表示程序正在运行或已停止
                    if (record.asyncClass == "running") this.emit("running", parsed);
                    else if (record.asyncClass == "stopped") {
                        // 根据停止原因发出不同的事件
                        const reason = parsed.record("reason");
                        // ... (处理不同的停止原因)
                    }
                } else if (record.type == "notify") {
                    // 处理通知类型的异步事件
                    if (record.asyncClass == "thread-created") {
                        this.emit("thread-created", parsed);
                    } else if (record.asyncClass == "thread-exited") {
                        this.emit("thread-exited", parsed);
                    }
                }
            }
```

```
                });
                handled = true;
            }
            // 如果当前行没有被任何上述条件处理，则记录为未处理的输出
            if (!handled) this.log("log", "Unhandled: " + JSON.stringify(parsed));
        }
    });
}
```

通过这些方法，代码确保了调试控制台输出的可读性和有用性，使得开发者能够更容易地理解调试过程中发生的事情，这对于调试复杂的应用程序或在开发过程中解决问题至关重要。

1.7 修改插件本身的编译配置文件 tsconfig.json

使得编译本插件的时候忽略文档文件夹和根文件夹下 60m 的 “演示视频.mp4”，从而极大减小编译出的插件二进制包的大小

1.8 修改 launch.json 文件

用户可以里提交自定义代码 launch.json 支持 *workspacefolder*
修改后的文件

2 新增功能

2.1 增加通过 SSH 进行 OS 调试的功能

我们为了进一步实现通过 SSH 进行远程操作系统调试的功能，通过以下步骤来集成 SSH 功能：

首先，初始化调试会话，initializeRequest 方法设置了调试会话支持的各种功能，例如支持条件断点、函数断点、内存读写等。这些功能通过修改 response.body 的不同属性来指定。接着，用 launchRequest 方法启动调试。然后，使用提供的 GDB 路径和其他参数（如调试器参数和环境变量）创建 MI2 类的实例。根据 args.pathSubstitutions 设置源文件的路径替换规则以便调试器可以正确地定位到原始源文件。配置好调试会话后，就开始处理 ssh 配置。如果提供了 SSH 参数 (args.ssh)，则进入 SSH 配置分支：

相关代码实现如下：

```
protected override launchRequest(response: DebugProtocol.LaunchResponse, args:
LaunchRequestArguments): void {
    // 使用提供的 gdbpath 或默认的 "gdb" 作为调试器命令
    const dbgCommand = args.gdbpath || "gdb";

    if (this.checkCommand(dbgCommand)) {
        // 如果调试器命令无效，发送错误响应并终止
        this.sendErrorResponse(response, 104, `Configured debugger ${dbgCommand}
```

```
            not found.`);
            return;
        }

        // 创建 MI2 实例，它是与 GDB 交互的接口
        this.miDebugger = new MI2(
            dbgCommand,
            ["-q", "--interpreter=mi2"],
            args.debugger_args,
            args.env
        );

        // 设置源文件路径替换规则
        this.setPathSubstitutions(args.pathSubstitutions);
```

初始化调试器，设置调试会话的状态标志:

```
this.initDebugger();
this.quit = false;
this.attached = false;
this.initialRunCommand = RunCommand.RUN;
this.isSSH = false;
this.started = false;
this.crashed = false;
this.setValuesFormattingMode(args.valuesFormatting);
this.miDebugger.printCalls = !!args.printCalls;
this.miDebugger.debugOutput = !!args.showDevDebugOutput;
this.stopAtEntry = args.stopAtEntry;
```

处理 SSH 配置，如果提供了 SSH 参数，设置默认的 SSH 参数值，如端口、X11 端口等:

```
 if (args.ssh !== undefined)
    if (args.ssh.forwardX11 === undefined)
        args.ssh.forwardX11 = true;
    if (args.ssh.port === undefined)
        args.ssh.port = 22;
    if (args.ssh.x11port === undefined)
        args.ssh.x11port = 6000;
    if (args.ssh.x11host === undefined)
        args.ssh.x11host = "localhost";
    if (args.ssh.remotex11screen === undefined)
        args.ssh.remotex11screen = 0;
```

```
        // 标记为 SSH 调试会话
        this.isSSH = true;
```

设置源文件映射:

```
    this.setSourceFileMap(args.ssh.sourceFileMap, args.ssh.cwd, args.cwd);
```

通过 SSH 连接到远程主机并启动 GDB（如果没有提供 SSH 参数，表示在本地启动 GDB）:

```
        this.miDebugger.ssh(args.ssh, args.ssh.cwd, args.target, args.arguments, args.terminal,
            // 如果 SSH 成功，发送成功的响应
            this.sendResponse(response);
        }, err => {
            // 如果 SSH 失败，发送错误响应
            this.sendErrorResponse(response, 105, `Failed to SSH: ${err.toString()}`);
        });
    } else {
        // 如果没有提供 SSH 参数，表示在本地启动 GDB
        this.miDebugger.load(args.cwd, args.target, args.arguments, args.terminal, args.autorun
            this.sendResponse(response);
        }, err => {
            this.sendErrorResponse(response, 103, `Failed to load MI Debugger: ${err.toString()}
        });
    }
}
```

在上述代码中，我们首先检查是否提供了 SSH 参数。如果提供了，我们将这些参数用于设置 SSH 连接，包括端口和 X11 端口等。然后，我们设置一个标志 this.isSSH 来表明我们正在使用 SSH。之后，我们设置了源文件映射，这有助于在调试过程中正确地映射和识别文件路径。最关键的是调用 this.miDebugger.ssh 方法，它负责根据提供的参数启动 SSH 连接，并在远程主机上运行 GDB。如果 SSH 连接成功建立，我们将发送一个成功的响应；如果失败，则发送一个错误响应。通过这种方式，开发者可以在本地机器上使用 VS Code 进行远程调试，就像在本地机器上调试一样方便。

以下代码是 ssh 的方法的具体实现，这个方法的目的是建立一个 SSH 连接到远程主机，并在远程主机上执行一系列命令，通常用于启动和控制远程调试会话。

```
// 定义一个 ssh 方法，用于通过 SSH 连接到远程主机并执行命令。
ssh(args: SSHArguments, cwd: string, target: string, procArgs: string, separateConsole: string,
    return new Promise((resolve, reject) => {
        // 标记已通过 SSH 连接。
        this.isSSH = true;
        // 初始 SSH 连接状态为未就绪。
        this.sshReady = false;
        // 创建一个新的 SSH 客户端实例。
```

```
        this.sshConn = new Client();

        // 如果提供了单独的控制台参数，发出警告，因为 SSH 不支持终端模拟器的输出。
        if (separateConsole !== undefined)
            this.log("stderr", "WARNING: Output to terminal emulators are not supported over SSH

        // 如果需要 X11 转发，设置相关事件处理。
        if (args.forwardX11) {
            this.sshConn.on("x11", (info, accept, reject) => {
                const xserversock = new net.Socket();
                // 处理本地 X11 服务器连接错误。
                xserversock.on("error", (err) => {
                    this.log("stderr", "Could not connect to local X11 server! Did you enable it
                });
                // 处理成功连接到本地 X11 服务器的事件。
                xserversock.on("connect", () => {
                    const xclientsock = accept();
                    // 创建 X11 转发数据的管道。
                    xclientsock.pipe(xserversock).pipe(xclientsock);
                });
                // 连接到指定的本地 X11 端口和主机。
                xserversock.connect(args.x11port, args.x11host);
            });
        }

        // 设置 SSH 连接参数，包括主机、端口和用户名。
        const connectionArgs: any = {
            host: args.host,
            port: args.port,
            username: args.user
        };

        // 根据认证方式设置连接参数，使用密钥或密码。
        if (args.useAgent) {
            // 如果使用 SSH 代理，设置环境变量中的 SSH 认证套接字。
            connectionArgs.agent = process.env.SSH_AUTH_SOCK;
        } else if (args.keyfile) {
            // 如果使用私钥文件，检查文件是否存在并读取内容。
            if (fs.existsSync(args.keyfile))
                connectionArgs.privateKey = fs.readFileSync(args.keyfile);
            else {
```

```
            // 如果私钥文件不存在，记录错误并拒绝 Promise。
            this.log("stderr", "SSH key file does not exist!");
            this.emit("quit");
            reject();
            return;
        }
    } else {
        // 如果不使用密钥文件，则使用密码认证。
        connectionArgs.password = args.password;
    }

    // 当 SSH 客户端准备就绪时，执行回调函数。
    this.sshConn.on("ready", () => {
        // 记录正在通过 SSH 运行的应用程序。
        this.log("stdout", "Running " + this.application + " over ssh...");
        // 设置执行命令时的参数，如 X11 转发参数。
        const execArgs: ExecOptions = {};
        if (args.forwardX11) {
            execArgs.x11 = {
                single: false,
                screen: args.remotex11screen
            };
        }
        // 构造要通过 SSH 执行的命令。
        let sshCMD = this.application + " " + this.preargs.concat(this.extraargs || []).join
        if (args.bootstrap) sshCMD = args.bootstrap + " && " + sshCMD;
        // 执行命令，并处理结果。
        this.sshConn.exec(sshCMD, execArgs, (err, stream) => {
            if (err) {
                // 如果执行命令时出错，记录错误并拒绝 Promise。
                this.log("stderr", "Could not run " + this.application + "(" + sshCMD + ") c
                if (err === undefined) {
                    err = new Error("<reason unknown>");
                }
                this.log("stderr", err.toString());
                this.emit("quit");
                reject();
                return;
            }
            // 如果命令执行成功，设置 SSH 连接为就绪状态。
            this.sshReady = true;
```

```
                // 保存执行命令的流，以便后续操作。
                this.stream = stream;
                // 设置数据和错误输出的处理函数。
                stream.on("data", this.stdout.bind(this));
                stream.stderr.on("data", this.stderr.bind(this));
                // 设置命令执行退出的处理函数。
                stream.on("exit", () => {
                    this.emit("quit");
                    this.sshConn.end();
                });
                // 发送初始化命令，如更改工作目录。
                const promises = this.initCommands(target, cwd, attach);
                promises.push(this.sendCommand("environment-cd \"" + escape(cwd) + "\""));
                // 如果需要附加到进程，发送相应的命令。
                if (attach) {
                    promises.push(this.sendCommand("target-attach " + target));
                } else if (procArgs && procArgs.length)
                    promises.push(this.sendCommand("exec-arguments " + procArgs));
                // 发送自动运行命令。
                promises.push(...autorun.map(value => { return this.sendUserInput(value); }));
                // 等待所有初始化命令完成。
                Promise.all(promises).then(() => {
                    this.emit("debug-ready");
                    resolve(undefined);
                }, reject);
            });
        }).on("error", (err) => {
            // 如果 SSH 连接出错，记录错误并拒绝 Promise。
            this.log("stderr", "Error running " + this.application + " over ssh!");
            if (err === undefined) {
                err = new Error("<reason unknown>");
            }
            this.log("stderr", err.toString());
            this.emit("quit");
            reject();
        }).connect(connectionArgs);
    });
}
```

以上所用到的 SSH 配置是通过 args 参数传递给 ssh 方法的，args 是一个 SSHArguments 类型的对象，包含了建立 SSH 连接所需的所有参数和配置（在 backend.ts 文件中）。

2.2 通过右键菜单添加/取消边界断点

- 根据我们现在的状态机，边界断点应该包含在断点组里面，所以像之前那样发一个 customRequest，让 GDB 直接设断点（不经过 mibase.ts 的 setBreakPointsRequest，因此不会保存到断点组里面）就不合适了。
- GoToKernel 等几个按钮的功能要么是不必要的，要么就是已经通过新的状态机自动化地实现了。
- GDB 设断点会有一个断点偏移的问题。例如用户将断点设置在 12 行，GDB 可能会把断点改到 15 行进行设置。之前把边界断点信息都放在配置文件里面时，就需要反复尝试来找到断点不会偏移的行。如果改为用右键菜单设置边界断点，肯定是用户先设断点，断点偏移，然后用户再将偏移后的那个断点设为边界断点，不会出现上述问题。
- 调试器的用户会反复改动 os 代码，因此边界断点的行号会一直改变。不仅要改配置文件还要考虑断点偏移，比较麻烦。如果开始 debug 再用鼠标点反而更自然。断点组机制的实现也会更自然。例如，每个用户程序的出口断点可能不一样（比如，一些是 rust 程序，一些是 C 的），用户可以选择刚开始 debug 的时候并不指定所有边界断点，而是运行到了用户态再添加用户断点。比静态的配置文件要好的多。

基于以上原因，我们移除了移除 GoToKernel 等几个按钮，添加了一个右键菜单，用户在某个断点上面右键单击即可将这个断点变成边界断点。这样边界断点除了通过配置文件添加，也可以通过右键菜单添加或者取消。

2.3 实现单步步进

这次更新后实现了之前没有的单步步进的功能，包括逐步执行、单条指令级别的调试以及跳出当前函数的调试操作。通过这些方法，实现逐步分析和理解代码的执行过程，从而更快地定位和解决问题。

```
/**
 * 执行程序的单步操作，可选择正向或反向。
 * @param {boolean} [reverse=false] 是否执行反向单步。
 * @returns {Thenable<boolean>} 程序是否仍在运行的 Promise。
 */
step(reverse: boolean = false): Thenable<boolean> {
    if (trace) this.log("stderr", "step");
    return new Promise((resolve, reject) => {
        this.sendCommand("exec-step" + (reverse ? " --reverse" : "")).then((info) => {
            resolve(info.resultRecords.resultClass == "running");
        }, reject);
    });
}

/**
```

```
     * 执行单条指令的单步操作，可选择正向或反向。
     * @param {boolean} [reverse=false] 是否执行反向单步。
     * @returns {Thenable<boolean>} 程序是否仍在运行的 Promise。
     */
    stepInstruction(reverse: boolean = false): Thenable<boolean> {
        if (trace) this.log("stderr", "stepInstruction");
        return new Promise((resolve, reject) => {
            this.sendCommand("exec-step-instruction" + (reverse ? " --reverse" : "")).then((info
                resolve(info.resultRecords.resultClass == "running");
            }, reject);
        });
    }

    /**
     * 执行步出操作，从当前函数返回到调用者。
     * @param {boolean} [reverse=false] 是否执行反向步出。
     * @returns {Thenable<boolean>} 程序是否仍在运行的 Promise。
     */
    stepOut(reverse: boolean = false): Thenable<boolean> {
        if (trace) this.log("stderr", "stepOut");
        return new Promise((resolve, reject) => {
            this.sendCommand("exec-finish" + (reverse ? " --reverse" : "")).then((info) => {
                resolve(info.resultRecords.resultClass == "running");
            }, reject);
        });
    }
```

2.4 自动安装脚本

由于调试器及 ebpf 所需工具和库非常多，而且依赖关系非常复杂，为了减少人为错误、提高可移植性、简化复杂构建过程，我们决定编写一个自动化安装脚本来提高效率。考虑到跨平台兼容性和系统资源的利用，我们选择用 shell 语言来进行编写，并进行输出提示。以安装 QEMU 为例，检测是否安装 QEMU，如果没有则下载最新版本，配置并编译 QEMU，编译完成后返回上一级目录。提示用户每个步骤的状态，并将这些信息记录到 output1.txt 文件中，用以提示用户。

```
# 如果未安装 QEMU，则下载最新版本
    echo -e "${YELLOW}QEMU is not installed. Downloading the latest version...${RESET}" | tee -a
    if git clone https://github.com/chenzhiy2001/qemu-system-riscv64; then
        echo -e "${YELLOW}下载完成${RESET}"| tee -a output1.txt
        # 编译安装并配置 RISC-V 支持
        cd qemu-system-riscv64
        echo -e "${YELLOW}编译 qemu.....${RESET}"| tee -a output1.txt
        ./configure --target-list=riscv64-softmmu,riscv64-linux-user  # 如果要支持图形界面，可添加
```

```
        make -j$(nproc)
        cd ..
        echo -e "${YELLOW}编译完成.${RESET}" | tee -a output1.txt
    else
        echo -e "${YELLOW}Error: Failed to clone qemu-system-riscv64.${RESET}"| tee -a output1.t
        exit 1
    fi
fi
```

自动安装脚本经过测试和完善，可以正确安装调试所需要的所有工具和库（在网络良好的情况下）。

2.5 调试调试器

初步编写配置文件后发现只能从内核态转换到用户态，不能从用户态回到内核态，排查原因无果后我们决定调试调试器来进一步排查原因。

- 调试器的构成及调试

  code-debug 插件分为两部分，扩展和调试适配器，这两部分是由两个进程来控制。所以如果调试的话应该是启动两个调试配置，一个是 launch extension，另一个是 server。

  - launch extension

    调试 extension 的部分，更具体地说是 extension.ts 文件，用它调试就会启动一个新窗口（扩展开发宿主）

  - server

    调试调试适配器的部分，即除了 extension.ts 文件的其他文件，这部分的调试需要进行一个配置（code-debug sever 的调试配置），在 code-debug 中的 launch.json 已经配置好了，里面有一个 4711 的端口号，启动这个配置以后，会监听这个端口号。在我们要调试的项目中，添加一个”debugServer”: 4711, 的配置，使两者可以传递信息。

- 具体调试步骤：在 code-debug 中找到调试的界面，选择 launch extension，然后按 F5，启动一个新窗口，在新窗口中选择要调试的项目打开，然后不要动。回到 code-debug 的调试界面，点击调试和运行的下拉框，选择 code-debug sever，点击绿色的开始调试按钮，就会发现多了一个调试配置，如下图。

  接下来，回到新窗口，按 F5，按照正常的调试流程进行操作，会触发在文件中设置的断点（推荐的断点位置 mibase.ts 中的 handleBreakpoint () 函数中）。

- 关于断点设置：尽量不要在 extension.ts 里面设置断点，会卡在那里，最后终止程序。

- 关于 console.log 函数的输出：对于上面两个启动配置，会有两个调试控制台，在不同文件中的输出会在不同的调试控制台中。

3 适配 xv6（C 语言）

xv6-riscv 是一个小型的 Unix 第六版操作系统实现，包含了基本的操作系统功能，如进程管理、内存管理、文件系统、设备驱动和系统调用。xv6-riscv 采用单内核结构，所有的操作系统服务都在内核模式下运行。内核代码包括内存管理、进程管理、文件系统、驱动程序和系统调用接口等部分。

3.1 更新 package.json

由于之前的调试器是仅 rust 语言可见的，我们修改了 package.json 文件，让它能够适配所有语言。

3.2 编写 launch.json

3.2.1 xv6 的 qemu 启动参数

一开始我们沿用了了 ebpf 的部分参数，发现会导致启动不了，最后阅读了官方文档，找到了推荐的启动参数。

```
"qemuPath": "qemu-system-riscv64",
            "qemuArgs": [
                "-machine", "virt", "-bios", "none",
                "-kernel", "${workspaceFolder}/kernel/kernel",
                "-m", "128M", "-smp", "2", "-nographic",
                "-global", "virtio-mmio.force-legacy=false",
                "-drive", "file=${workspaceFolder}/fs.img,if=none,format=raw,id=x0",
                "-device", "virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0",

                "-s", "-S"
            ],
```

3.2.2 获取断点组名称及路径

经过调试排查，我们发现不能从用户态回到内核态的原因之一是调试器没有成功读取用户态的符号表。xv6 的用户文件经过编译后为+

```
    "filePathToBreakpointGroupNames": {
                "isAsync": false,
                "functionArguments": "filePathStr",
                "functionBody": "if (filePathStr.includes('kernel')) { return ['kernel']; } else
            },
            "breakpointGroupNameToDebugFilePath":{
                "isAsync": false,
                "functionArguments": "groupName",
```

```
            "functionBody": "if (groupName === 'kernel') {        return '${workspaceFolder}
        }
```

3.2.3 xv6 内核态和用户态转换的边界

不能从用户态回到内核态还有一个原因是用户态的边界未被正确设置。

- kernel/syscall.c 是负责处理已经进到内核之后的 syscall 处理流程。我们需要的是用户态的 syscall 接口，在 usys.S 中。
- 因为 usys.S 文件中有多个 ecall，也就是说 ** 用户态有多个边界断点 **（因为 xv6 在用户态没有一个专门的 syscall() 处理函数，而是每个 syscall 的调用单独处理）。我们的调试器一开始是基于 ebpf 写的，所以用户和内核的边界都只有一个，添加新的边界断点时旧的会被替换掉。所以需要将边界改成数组，并 ** 修改调试器的边界代码及相关处理函数 **。

修改边界定义：

```
export class Border  {
^^Ifilepath:string;
^^Iline:number;
^^Iconstructor(filepath:string, line:number){
^^I^^Ithis.filepath = filepath;
^^I^^Ithis.line = line;
^^I}
}
class BreakpointGroup {
^^Iname: string;
^^IsetBreakpointsArguments: DebugProtocol.SetBreakpointsArguments[];
^^Iborders?:Border[]; // can be a border or undefined
^^Ihooks:HookBreakpoints; //cannot be `undefined`. It should at least an empty array `[]`.
^^Iconstructor(name: string, setBreakpointsArguments: DebugProtocol.SetBreakpointsArguments[
^^I^^Iconsole.log(name);
^^I^^Ithis.name = name;
^^I^^Ithis.setBreakpointsArguments = setBreakpointsArguments;
^^I^^Ithis.hooks = hooks;
^^I^^Ithis.borders = borders;
^^I}
}
```

修改更新边界的逻辑：通过 eval 执行 filePathToBreakpointGroupNames 函数来获取与边界断点文件路径对应的断点组名称列表，然后遍历这些组名，检查每个组名是否存在于当前断点组列表中。如果找到匹配的断点组，则将边界断点添加到该组的 borders 属性中；如果未找到，则创建一个新的断点组，并将边界断点添加到新的断点组中。

```
public updateBorder(border: Border) {
^^I^^Iconst result = eval(this.debugSession.filePathToBreakpointGroupNames)(border.filepath);
```

```
^^I^^Iconst groupNamesOfBorder:string[] = result;
^^I^^Ifor(const groupNameOfBorder of groupNamesOfBorder){
^^I^^I^^Ilet groupExists = false;
^^I^^I^^Ifor(const group of this.groups){
^^I^^I^^I^^Iif(group.name === groupNameOfBorder){
^^I^^I^^I^^I^^IgroupExists = true;
^^I^^I^^I^^I^^Igroup.borders.push(border);
^^I^^I^^I^^I}
^^I^^I^^I}
^^I^^I^^Iif(groupExists === false){
^^I^^I^^I^^Ithis.groups.push(new BreakpointGroup(groupNameOfBorder, [], new HookBreakpoints([]),
^^I^^I^^I}
^^I^^I}
^^I}
```

修改相关函数：

通过调用 miDebugger.getStack 方法获取当前执行堆栈的文件路径和行号，接着遍历边界断点列表，比较文件路径和行号是否匹配。如果匹配，则触发操作系统状态转换事件。

```
else if(action.type === DebuggerActions.check_if_user_to_kernel_border_yet){
^^I^^I^^Ithis.showInformationMessage('doing action: check_if_user_to_kernel_border_yet');
^^I^^I^^Ilet filepath:string = "";
^^I^^I^^Ilet lineNumber:number = -1;
^^I^^I^^Iconst userToKernelBorders = this.breakpointGroups.getCurrentBreakpointGroup().borders;
^^I^^I^^Iconst userToKernelBorderFile = this.breakpointGroups.getCurrentBreakpointGroup().border
^^I^^I^^Iconst userToKernelBorderLine = this.breakpointGroups.getCurrentBreakpointGroup().border
^^I^^I^^Ithis.miDebugger.getStack(0, 1, this.recentStopThreadID).then(v=>{
^^I^^I^^I^^Ifilepath = v[0].file;
^^I^^I^^I^^IlineNumber = v[0].line;

^^I^^I^^I^^I if (userToKernelBorders) {
^^I^^I^^I^^I^^Ifor (const border of userToKernelBorders) {
^^I^^I^^I^^I^^Iif (filepath === border.filepath && lineNumber === border.line) {
^^I^^I^^I^^I^^I this.OSStateTransition(new OSEvent(OSEvents.AT_USER_TO_KERNEL_BORDER));
^^I^^I^^I^^I^^I break;
^^I^^I^^I^^I }
^^I^^I^^I^^I}
^^I^^I^^I }
^^I^^I^^I });
^^I^^I^^I
^^I^^I}
else if(action.type === DebuggerActions.check_if_kernel_to_user_border_yet){
^^I^^I^^I//··· 此处与上面函数逻辑相同，省略
```

```
^^I^^I^^I^^I
}
```

在 launch.json 里面只指定边界断点，没有指定边界断点所属的断点组。边界断点所属的断点组是由调试器自己去判定的。所以当触发了多个断点组中的一个，

调试器就会判定这个边界断点属于某某断点组，然后进行断点组切换的流程。

3.2.4 钩子断点

完成以上修改之后，用户态和内核态已经可以正常切换了。但是经过几次切换后，调试器会自己中断。是因为我们之前将钩子断点设在了第 6 行，此时获取下一进程名时返回为空。

```
// sysfile.c
1 sys_exec(void)
2 {
3   char path[MAXPATH], *argv[MAXARG];
4   int i;
5   uint64 uargv, uarg;
6   argaddr(1, &uargv);
7   if(argstr(0, path, MAXPATH) < 0) {
8     return -1;
9   }
10 }
```

修改后将钩子断点设置在了 “‘ int ret = exec(path, argv);“‘，可以正常返回下一进程名，至此，调试器可以正确调试 xv6 和 ebpf。

3.2.5 正确的配置文件

经过测试的 [配置文件](https://gitlab.eduxiji.net/T202410011992734/project2210132-235708/-/blob/master/installation