基于开源系统开发应用程序

– nim 移植 LoongArch

队伍名称： nimble

成员： 蔺春名，赵禹惟，李佳潼

创建日期： 2024.12.06

目　录

1 项目背景与目标

1.1 项目背景

Nim 是一个静态系统编程语言，LoongArch 是龙芯中科基于 CPU 研制和生态建设积累推出的自主指令系统架构。在先前的研究中，Nim 已经基本实现了在 LoongArch 上的迁移，其 GitHub 上的开源项目仓库如下：

https://github.com/nim-lang/Nim.git

我们的工作是基于已有的迁移上进行的。Nim 项目已有的迁移成果如下：

1. 在 *install.ini* 增添支持：https://github.com/nim-lang/Nim/pull/23672
2. 为 Nim 的符号常量 hostCPU 增添可能的项"loongarch64", etc.
   https://github.com/nim-lang/Nim/pull/19233

但是，目前缺少对 LoongArch 架构的完美支持：

我们尝试在 LoongArch 架构主机上使用 Nim 时发现，不但无法按照官方索引安装 Nim 编译器，而且即使通过源码编译安装也仍然失败。

具体详情我们小队已经报告到 Nim 的官方仓库上，详见：

https://github.com/nim-lang/Nim/issues/24118

另一方面，在后续安装成功后，我们运行了 Nim 的官方测试用例，发现在 LoongArch 架构上的一些测试用例没有正常通过。

1.2 项目目标

正如上述，在 Nim 的官方仓库中，LoongArch 的迁移工作已经有一定的进展，但是仍存在一些问题需要解决。

- 添加对 LoongArch 的安装支持
- 完善对 LoongArch 的运行支持 (通过测试用例)

通过如下链接可以访问我们在现有的迁移成果基础上的进一步开发：

https://gitlab.eduxiji.net/T202410423994345/project2608128-274097.git

这是由 git 管理的 2024 年全国大学生操作系统竞赛的龙芯相关项目 — "基于开源操作系统开发应用程序-nim 移植 LoongArch"。

2 开发环境准备

2.1 前情引入

在将 Nim 编译器移植到 LoongArch 架构的过程中，我们需要配置一个完整的开发环境，确保工具链和编译器能够支持 RISC64le 架构。LoongArch 处理器架构是龙芯系列处理器的核心架构，具有 RISC(简化指令集计算机) 的特性，并且基于 RISC64le 架构实现，即采用 64 位架构且字节序为小端排序。RISC64le 时 RISC-V 架构的一个变种，借鉴了 RISC-V 的一些理念，并加入了本地的指令集扩展。

2.2 必要工具和依赖

Nim 编译器在进行跨平台移植时，通常会通过本机编译生成目标平台的 C 代码，再通过 C 编译器进行编译，最终生成可执行文件 (.exe)。为了确保开发工作顺利进行，需要安装以下工具和依赖项：

- Nim 及其 csource 源码: 用于编译 Nim 编译器
  https://github.com/nim-lang/csources_v2
- 依赖库: GCC 等开发工具

确保以上工具的安装，推荐使用最新稳定的版本。

2.3 配置 Nim 编译环境

Nim 的编译过程是: $nim \rightarrow C \rightarrow .exe$，因为 Nim 编译器实现了自举，所以有本机编译和交叉编译两种方式，具体流程如下:

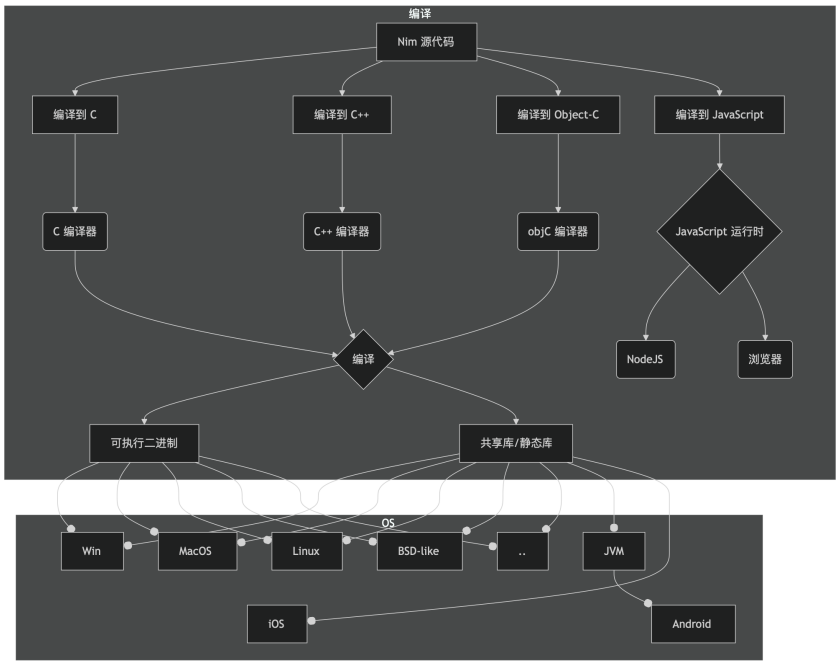

图 1: Workflow

2.3.1 本机编译

- 安装 Nim 编译器:
  https://nim-lang.org/install.html
- 通过如下指令验证是否成功安装:
  `nim --version`
- 配置 C 编译器 GCC / Clang

2.3.2 交叉编译

- 下载交叉编译工具链
  LoongArch 的开发需要编译工具链，具体步骤如下:
  1. 通过如下网址找到适用于目标架构的工具链
     http://www.loongide.com/content/article.asp?style=nodate&typeid=26&id=150
  2. 选择对应工具链版本
- 配置 Nim 编译环境
  Nim 编译器需要进行配置以支持龙芯芯片架构，以下是基于 LoongArch64 所使用工具链演示:

```
export PATH=$PATH:/opt/toolchain-loongarch64-linux-gnu-gcc8-host-x86_64-2022-07-18/bin/
export ARCH=loongarch64
export CROSS_COMPILE=loongarch64-linux-gnu-
```

之后输入如下命令进行编译 (注：这是个例子，具体情况具体分析):

```
loongarch64-linux-gnu-gcc main.c -o main
```

2.4 测试开发环境

为确保环境配置无误，运行以下测试程序：

1. 创建测试文件
   创建一个简单的 Nim 程序 hello.nim：
   ```
   echo "Hello Loongson!"
   ```
2. 编译并运行
   使用交叉工具链编译：
   ```
   wget http://example.com/toolchain.tar.gz
   tar -xzvf toolchain.tar.gz -C /opt/loongarch
   ```
3. 验证生成的 ELF 文件
   确认生成的二进制文件适用于 Loongson 架构：
   ```
   file hello
   ```
4. 运行
   在龙芯机器上测试：
   ```
   ./hello
   ```

成功运行后，开发环境配置完成。

3 移植过程

3.1 LoongArch 平台的特性与挑战

3.1.1 平台特性

- 指令集架构：LoongArch 是龙芯自主研发的指令集架构，具有 32 位和 64 位模式，支持丰富的矢量扩展指令。
- 独特寄存器集：包括通用寄存器 (GPRs)、浮点寄存器 (FPRs)、响亮寄存器 (VEC) 等，数量和布局与传统 RISC 架构不同。
- 内存管理：采用多级页表机制，支持打页、超大页和段式内存映射。
- 异常处理：异常向量表布局与传统 MIPS 架构存在类似，但偏移和一场代码分配有所不同。

3.1.2 主要挑战

- 指令兼容性：Nim 编译器需要支持生成 LoongArch 特定的汇编指令。
- 启动代码适配：LoongArch 的引导过程需要重新编写启动代码以支持新架构。
- 寄存器操作与调用约定：LoongArch 使用独特的调用约定，需要在编译器后端中适配

3.2 Nim 编译器调整

3.2.1 编译后端支持

1. 修改 Nim 编译器的 GCC 后端配置文件 (nim.cfg)，添加对 LoongArch 平台的支持 (仅交叉编译时必须)：

```
when hostCPU == "loongarch64":
 const
  gcc.exe = "loongarch64-linux-gnu-gcc"
  linker.exe = "loongarch64-linux-gnu-ld"
```

2. 为 LoongArch 平台启用特定的编译选项：

```
--cpu:loongarch64
```

3.2.2 适配调用约定

1. 根据 LoongArch 的函数调用约定，调整参数传递规则：

   - 前 8 个整形参数通过寄存器 a0 到 a7 传递，剩余参数通过堆栈传递。
   - 返回值存放在 a0 (或 a0/a1) 中。

3.2.3 特定指令支持

1. 如果需要直接生成特定 LoongArch 指令 (如原子操作指令 amadd.d)，可使用 Nim 的内联汇编支持：
   注：以下的 $0, $1, $2 实际书写时应替换为寄存器名，如 a0、a1、a2 等，且 Loongarch 汇编中寄存器需用 $ 前缀。

```
asm """
amadd.d $0, $1, $2
"""
```

3.3 LoongArch64 CPU 的启动

3.3.1 启动流程概述

LoongArch 启动代码流程与传统架构类似，主要包括以下步骤：

1. 初始化栈指针 (sp)。
2. 设置异常向量表。
3. 初始化内存和页表。
4. 跳转到内核主函数。

3.4 寄存器与内存布局设置

3.4.1 寄存器初始化

1. 全局寄存器配置

   (a) csr_crmd: CPU 当前模式寄存器，设置内核模式。

   (b) csr_asid: 地址空间标识寄存器，用于虚拟内存管理。

2. 通用寄存器初始化在初始化阶段，清零所有通用寄存器，保留 sp, gp 等必要寄存器：

```
li t0, 0
mv a0, t0
mv a1, t0
# 其他寄存器类似
```

3.4.2 内存布局设计

在 link.ld 链接脚本中定义内存段：

```
SECTIONS {
    . = 0x80000000;
.text : {
   *(.text) }
.rodata : {
   *(.rodata) }
.data : {
   *(.data)}
.bss : {
   *(.bss) }
}
```

3.5 std/posix

3.5.1 std/posix 介绍

std/posix 模块提供了与 POSIX 标准相关的功能，使得 Nim 程序能够在类 Unix 系统上进行低级操作，例如文件管理、进程控制和系统调用。

这一模块对系统的调用进行封装，使得开发者可以使用 Nim 语言的语法进行系统级编程，增强了程序的可移植性。

具体内容见：https://nim-lang.org/docs/posix.html

3.5.2 主要功能

1. 文件和目录操作：
   - 提供了创建、删除、重命名和遍历目录的功能。
   - 支持文件权限的修改以及文件属性的查询。
2. 进程管理：
   - 可以创建子进程，并支持进程的同步和通信。
   - 提供了对进程状态的监控功能。
3. 信号处理：
   - 提供了设置和捕获 POSIX 信号的能力，使程序能够对异步事件作出反应。
4. 线程支持：
   - 提供了多线程编程的接口，允许创建和管理线程。
   - 支持互斥锁和条件变量，以确保线程安全。

3.5.3 移植注意事项

1. 系统调用的兼容性：确保所有使用的 POSIX 系统调用在 LoongArch 架构上都能正确执行，特别是涉及底层操作的函数。
2. 编译器支持：检查 LoongArch 的编译器是否支持 Nim 的特性，尤其是与 POSIX 相关的扩展。
3. 性能优化：评估在 LoongArch 上执行的性能，可能需要进行特定的优化，以确保与其他架构的兼容性和性能一致性。
4. 测试用例：编写对 std/posix 功能的单元测试，确保在 LoongArch 架构上正确运行，并与预期结果一致。

4 错误信息筛选与处理

进行 Nim 的测试过程中，./koch test 默认在工作目录 (即./testresults) 下生成测试结果文件，结构为：

```
category/testName.json
category为测试类型，如“compiler”、“tools”等，
testName为测试名称，如“t001”、“t002”等。
```

4.1 测试结果解析与管理

这个过程是遍历并指定目录及其子目录下的所有 JSON 文件，逐文件解析内容，过滤出未成功的记录并对记录进行筛选 (删除重复字段) 和结构化 (转换字段类型)，最后将非空数据以 FileRecord 的形式返回。得到的 json 文件见~ */nim_testresult/*

1. 定义 RunRecord 和 FileRecord 类型，存储与测试相关的数据，包括测试状态、后段类型、命令行参数等。

```
# RunRecord
iterator filterNonSuccess*(arrNode: JsonNode): RunRecord =
 for node in arrNode:
  if "reSuccess" == node["result"].getStr():
   continue

  let target = node["target"].getStr()
  for field in DupFields:
   node.delete field

  var argvJoined = node["name"].getStr
  var argv = argvJoined.split ' '
  node["name"] = newJString argv[0]
  argv.delete 0
  var res: RunRecord

  res.status = node.popEnumKey[:TResultEnum]("result")
  res.backend = node.popEnumKey[:Backend]("target")

  res.node = node
  res.args = argv
  yield res

# FileRecord
using dir: string proc parseFile*(dir; filename: string):
Option[FileRecord] =
  ## returns none iff file is empty
  let path = dir/filename
  var fstr: FileStream
  if not fstr.openMayAfterFixJson path:
   return
  assert not fstr.isNil
  defer: fstr.close
  var res: FileRecord
  res.name = filename
  res.name.removeSuffix".json"
  for i in fstr.filterNonSuccess filename:
   res.data.add i
  result = some res
```

2. 使用 JSON 数据结构来表示测试结果，通过解析和处理 JSON 文件，实现对测试状态的分析。

4.2 状态管理

1. 定义 getAllStatus() 函数用于手机所有测试记录中出现的状态，利用集合 (HashSet) 来避免重复。

```
# getAllStatus()
proc getAllStatus*: HashSet[TResultEnum] =
 template doWith(d) = result.incl d.status
 dir.doWithNonSuccNode doWith
```

2. 定义 getStatusDiffMap() 函数通过状态分类，将测试记录分组存储在表 (Table) 中，便于后续查看和分析。

```
# getStatusDiffMap()
proc getStatusDiffMap*: Table[TResultEnum, seq[RunRecord]] =
# result = initTable[TResultEnum, seq[RunRecord]]()
template doWith(d) =
 if d.status not_in result:
  result[d.status] = @[]
 result[d.status].add d
dir.doWithNonSuccNode doWith
```

4.3 文件处理与修复

这个过程是修复和打开可能存在格式问题的 JSON 文件，并将其内容加载为文件流。

1. 定义 openMayAfterFixJson() 函数在打开文件时检查并修复文件格式，确保数据能够正确解析。

2. 其中包括读取并处理 JSON 文件，包括自动修复可能缺失的 JSON 结构 (如缺少闭合的']')。

```
let arrEnded = f.readChar() == ']'

if arrStarted and not arrEnded:
 f.truncate()
 f.write ']'
 f.flushFile()

f.setFilePos(0, fspSet)
result = true
```

4.4 输出与格式化

这个过程是将测试运行得到的数据进行格式化为 Markdown 列表。
具体内容见~ */nim_testresult/loongson_testresult.md*

1. 利用 MarkdownList 类型，生成 Markdown 格式的输出，便于在测试结果生成后以结构化的形式展示测试结果。

```
using self: MarkdownList
func `\$`*(self; nSpace=self.nSpace; appendNewLine=true): string =
 let spaces = ' '.repeat nSpace
   for line in self.lines:
      result.add spaces
      result.add self.prefix
      result.add line
      result.add '\n'
   if appendNewLine: result.add '\n'
```

2. 提供 asCode() 和 toInlineCode() 方法来处理输出格式，增强可读性和信息传递。

```
# asCode()
func asCode(s: var string; lang=""; default="*None*") =
  if s.len == 0:
     s = default
     return
  s = '\n' & "```"& lang & '\n' & s
  if s[^1] not_in {'\n', '\r'}: s.add '\n'
  s = indent(s & "```", IndentSpaceN)

# toInlineCode()
func toInlineCode(s: string): string = '`' & s & '`'
```

4.5 主程序逻辑

1. 使用 *when isMainModule* 结构来确保只有在直接运行该文件时才执行相关逻辑，这样可以方便地将代码作为模块导入而不执行主逻辑。

2. 调用 getAllStatus() 和 echoStatusDiffMap() 函数以打印出测试状态和状态对应的详细信息。

```
import parseTestamentJson/mains

when isMainModule:
   echo getAllStatus()
   echoStatusDiffMap()
```

(详细内容见：https://github.com/nimigrate/parseTestamentJson.git)

5 代码实现

5.1 问题解决

5.1.1 高级语言迁移

涉及修正内容过多，这里举几个例子，其他修改请见 $\sim /build\_nim/gitlab-eduxiji-repo/src$

- 进行对 *socketstreams.nim* 和 *thttpclient_ssl.nim* 测试时，如果采用 $--cpu:mips64el$ 架构编译会失败。具体代码如下所示：

```
discard """
matrix: "--mm:refc; --mm:orc"
  output: '''
OM
NIM
3
NIM
NIM
Hello server!
Hi there client!
'''"""
import std/socketstreams, net, streams

block UDP:
   var recvSocket = newSocket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
   var recvStream = newReadSocketStream(recvSocket)
   recvSocket.bindAddr(Port(12345), "127.0.0.1")

   var sendSocket = newSocket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
   sendSocket.connect("127.0.0.1", Port(12345))
   var sendStream = newWriteSocketStream(sendSocket)
   sendStream.write "NOM\n"
   sendStream.setPosition(1)
   echo sendStream.peekStr(2)
   sendStream.write "I"
   sendStream.setPosition(0)
   echo sendStream.readStr(3)
   echo sendStream.getPosition()
   sendStream.flush()

   echo recvStream.readLine()
   recvStream.setPosition(0)
   echo recvStream.readLine()
   recvStream.close()

block TCP:
   var server = newSocket()
   server.setSockOpt(OptReusePort, true)
   server.bindAddr(Port(12345))
   server.listen()
```

```
var
     client = newSocket()
     clientRequestStream = newWriteSocketStream(client)
     clientResponseStream = newReadSocketStream(client)
  client.connect("127.0.0.1", Port(12345))
  clientRequestStream.writeLine("Hello server!")
  clientRequestStream.flush()

  var
     incoming: Socket
     address: string
  server.acceptAddr(incoming, address)
  var
     serverRequestStream = newReadSocketStream(incoming)
     serverResponseStream = newWriteSocketStream(incoming)
  echo serverRequestStream.readLine()
  serverResponseStream.writeLine("Hi there client!")
  serverResponseStream.flush()
  serverResponseStream.close()
  serverRequestStream.close()

  echo clientResponseStream.readLine()
  clientResponseStream.close()
clientRequestStream.close()
```

分析：

MIPS 和 LoongArch64 架构在实现网络相关的系统调用时，会有不同的约定和支持的功能。包括套接字选项 SO_REUSEPORT 的实现和可用性。在 MIPS 架构上，SO_REUSEPORT 的值为 0x0200，而在 LoongArch64 架构上，其值为 15，可知该选项在 MIPS 中和在 LoongArch 上得到的结果不同。

考虑到第一版 Nim 编译器时将 Nim 代码编译成 C 代码，而 C 代码本身在处理不同的套接字选项时可能未能考虑到不同架构的实现差异。特别是如果在编译时未能正确检测到目标平台的特性，可能导致生成的代码出现问题。再者，C 代码中的某些特性 (如 SO_REUSEPORT 的使用) 可能在不同架构上没有经过充分测试，导致不兼容的情况。

5.1.2 汇编语言迁移

LoongsonArch 架构的汇编指令集在某些方面与 intel 格式相似，例如：

- 立即数无前缀：在龙芯汇编中，立即数（即直接给出的数值）通常不需要前缀。这意味着在指令中直接写出数值即可，不需要额外的符号来指明这是一个立即数。
- reg(at&t: %)：龙芯汇编中寄存器通常用 $ 符号表示，这与 AT&T 汇编语法中的% 符号相似。例如 $r0 表示通用寄存器 0，这种表示方法有助于区分寄存器和内存地址或立即数。
- do dst, src：这是龙芯指令的格式，其中 *do* 表示某种操作，*dst* 是目标操作数（通常是寄存

器或内存地址），而 *src* 是源操作数。这种格式与 Intel 汇编中的 *instruction dst, src* 类似，表示将源操作数 *src* 进行某种操作后存储到目标操作数 *dst* 中。

- do. 类型后缀：这是龙芯指令后缀，用于指定操作数的类型或大小。类似于 intel_x86 架构中，*add dword ptr*[*eax*], *ebx* 中的 *dword ptr* 指定了操作数的类型为 32 位。

- 进行对 *tasm.nim* 测试时发现龙芯不支持 *mov* 指令。具体代码如下所示：

```
proc testAsm() =
 let src = 41
 var dst = 0

 asm """
  mov %1, %0\n\t
  add $1, %0
  : "=r" (`dst`)
  : "r" (`src`)"""

 doAssert dst == 42
when defined(gcc) or defined(clang) and not defined(cpp):
 {.passc: "-std=c99".}
 testAsm()
```

运行结果如下：

```
loongson@loongson-pc:~/build_nim/Nim$ testament --nim:bin/nim run tests/compiler/tasm.nim
FAIL: tests/compiler/tasm.nim c                                    ( 0.68 sec)
Test "tests/compiler/tasm.nim" in category "compiler"
Failure: reNimcCrash
$ /home/loongson/build_nim/Nim/bin/nim c --hints:on -d:testing --nimblePath:build/deps/pkgs2   --nimCache:nimcache/tests/compiler/tasm.
nim_4a8a08f09d37b73795649038408b5f33  tests/compiler/tasm.nim
Hint: used config file '/home/loongson/build_nim/Nim/config/nim.cfg' [Conf]
Hint: used config file '/home/loongson/build_nim/Nim/config/config.nims' [Conf]
Hint: used config file '/home/loongson/build_nim/Nim/tests/config.nims' [Conf]
Hint: used config file '/home/loongson/build_nim/Nim/tests/compiler/nim.cfg' [Conf]
CC: tasm.nim
/tmp/ccBJfUvn.s: Assembler messages:
/tmp/ccBJfUvn.s:182: 致命错误: no match insn: mov       $r12,$r12
Error: execution of an external compiler program 'gcc -c  -w -fmax-errors=3 -pthread -std=c99   -I/home/loongson/build_nim/Nim/lib -I/h
ome/loongson/build_nim/Nim/tests/compiler -o /home/loongson/build_nim/Nim/nimcache/tests/compiler/tasm.nim_4a8a08f09d37b73795649038408b
5f33/@mtasm.nim.c.o /home/loongson/build_nim/Nim/nimcache/tests/compiler/tasm.nim_4a8a08f09d37b73795649038408b5f33/@mtasm.nim.c' failed
 with exit code: 1


FAILURE! total: 1 passed: 0 skipped: 0 failed: 1
```

图 2: tasm

分析：这其中的 *mov* $*r*12, $12 没有匹配，这是因为 Loongson 缺少 *mov* 指令导致。
解决方案：

```
```asm
 mov %1, %0\n\t
 add $1, %0
```
```

将这部分代码修改为

```
`addi.w %0, %1, 1`
```

5.2 其他问题

5.2.1 LoongArch 内置 Node 版本问题

- 在测试 *tnativeexc.nim* 时，发现对于同一报错，每次报错信息都不一样，具体代码如下所示：

```
discard """
 action: "run"
"""

import jiffy

# Can catch JS exceptions
try:
   asm """throw new Error('a new error');"""
except JsError as e:
   doAssert e.message == "a new error"
except:
   doAssert false

# Can distinguish different exceptions
try:
   asm """JSON.parse(';;');"""
except JsEvalError:
   doAssert false
except JsSyntaxError as se:
   doAssert se.message == "Unexpected token ';', \";;\" is not valid JSON"
except JsError as e:
   doAssert false

# Can catch parent exception
try:
   asm """throw new SyntaxError();"""
except JsError as e:
   discard
except:
   doAssert false
```

运行结果如下：

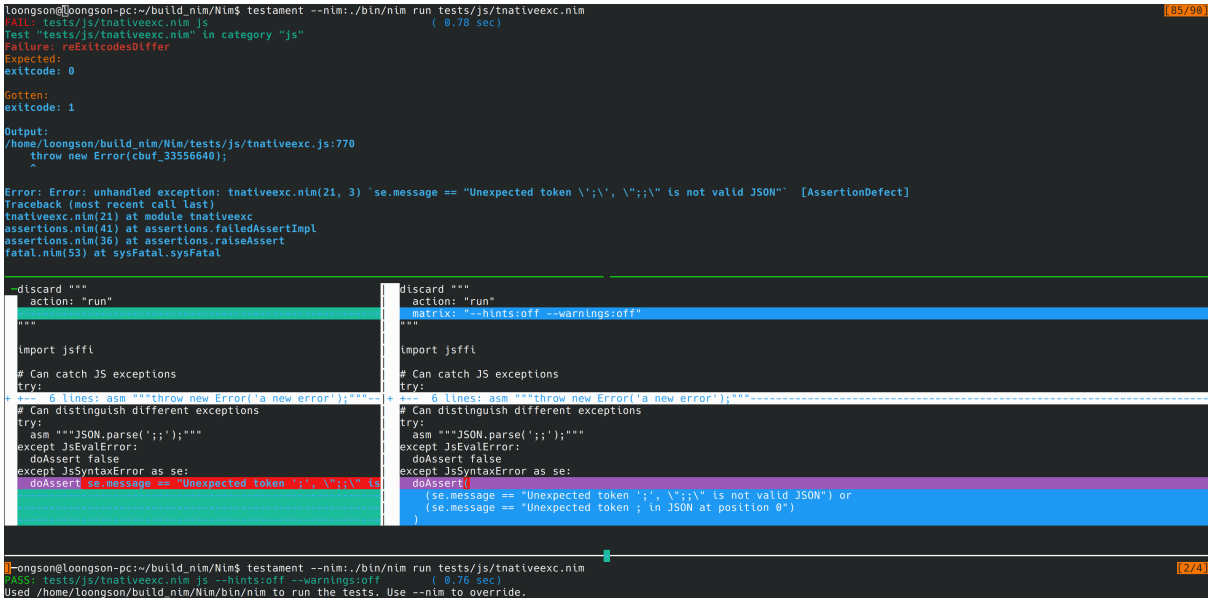

图 3: TasmDebug

分析：由此发现是因为环境导致的问题，即当前所使用 Node 版本过低，所以对于同一报错会有不同报错信息，从而导致测试失败。检查结果如下：

```
loongson@loongson-pc:~/build_nim/Nim$ node --version
v14.16.1
loongson@loongson-pc:~/build_nim/Nim$ node -e "JSON.parse(';;')"
undefined:1
;;
^

SyntaxError: Unexpected token ; in JSON at position 0
    at JSON.parse (<anonymous>)
    at [eval]:1:6
    at Script.runInThisContext (vm.js:133:18)
    at Object.runInThisContext (vm.js:310:38)
    at internal/process/execution.js:77:19
    at [eval]-wrapper:6:22
    at evalScript (internal/process/execution.js:76:60)
    at internal/main/eval_string.js:23:3
```

图 4: Node

5.2.2 命令行参数问题

- Nim 有一个命令行参数 $--asm$，用于生成 asm 代码，当在 LoongArch 下执行时，编译器会报错：

  `gcc: error: unrecognized command line option ‘-masm=intel’`

  检查后发现，Nim 源码的 *extccomp.nim* 有

  `gnuAsmListing = "-Wa,-acdl=$asmfile -g -fverbose-asm -masm=intel"`

  而 LoongArch 的汇编既不是 AT&T 格式，也不是 intel 格式，而是 MIPS 系，所以不接受 $-masm = intel$ 参数 (即 $-masm$ 用于切换生成汇编格式是 intel 格式还是 AT&T 格式)。之后发现 $-Wa$，表示给汇编器传参，具体如下：

`-Wa, acdl=$asmfile`

不会导致 LoongArch 的 GCC 生成名为 $asmfile 的汇编代码文件。因此这个问题需要之后向 GCC 提交 issue。考虑到的解决方案有：增添 $-S$ 参数，这样能使 GCC 生成汇编代码，但是会造成生成文件后缀名为 $.s$，而 Nim 期望生成文件后缀为 $.asm$。参考文档详见：https://nim-lang.org/docs/nimc.html

5.2.3 文档格式修复

1. 在查阅文档时偶然间发现了"loongarch64" 错误，具体如下：

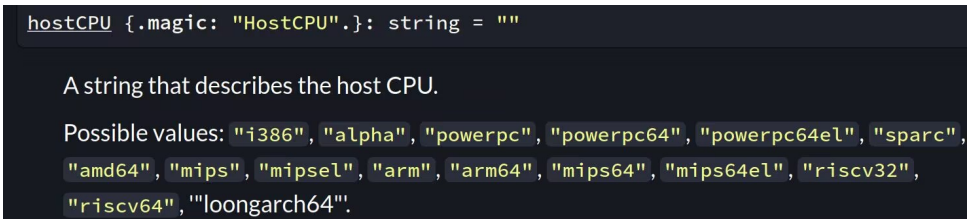

图 5: Format

经过上报官方 Nim，确认这是一个文档格式的修复。详见：https://github.com/nim-lang/Nim/pull/23621

6 性能优化与问题解决

6.1 编译优化策略

1. 启用 LoongArch 特定优化选项，如矢量化和流水线优化。
2. 利用 Nim 的内嵌 C 代码功能，直接调用高校的底层库函数。

6.2 常见问题与解决方法

1. 问题 1：工具链配置错误
   解决方案：检查 PATH 和交叉编译工具链是否正确配置。
2. 问题 2：系统调用失败
   解决方案：验证 POSIX 接口的实现是否与 LoongArch 平台兼容。

7 预期效果与未来计划

7.1 预期效果

1. 完善国产龙芯芯片的生态
2. 为软件国产化提供更优方案进行探索
3. 完成 Nim 语言编译器在国产龙芯芯片下的迁移，提高 Nim 的跨平台性
4. 推进 Nim 在软件开发中的应用

7.2 未来计划

1. 增加对更多 LoongArch 特性的支持
2. 测试 LoongArch 新版本 ABI 的支持
3. 优化 Nim 编译器生成代码的性能，使其更适合龙芯平台
4. 扩展第三方库的功能，以支持更多的 LoongArch 应用场景

8 附录

8.1 项目相关工具使用指南

1. 交叉编译工具链下载和安装：
   - 从https://www.loongide.com 获取最新版工具链。
   - 配置环境变量以使用工具链。
2. 调试工具使用：
   - GDB(GNU Debugger)
   - GCC(GNU Compiler Collection)

8.2 参考资料

- Nim 官方文档 https://nim-lang.org/documentation.html
- 龙芯中科官网 https://www.loongson.cn/system/loongarch
- Nim 官方 GitHub 仓库 https://github.com/nim-lang/Nim
- Nim 支持 LoongArch 架构官方文档 https://nim-lang.org/docs/system.html#hostCPU