第二届 eBPF开发者大会

www.ebpftravel.com

eBPF 与 Golang Profiling 阿里云的探索与实践

刘恺（千陆）

2024-04-13

个人简介

刘恺

花名千陆， 阿里云高级研发工程师, 目前负责云原生可观测团队 eBPF 基础设施。

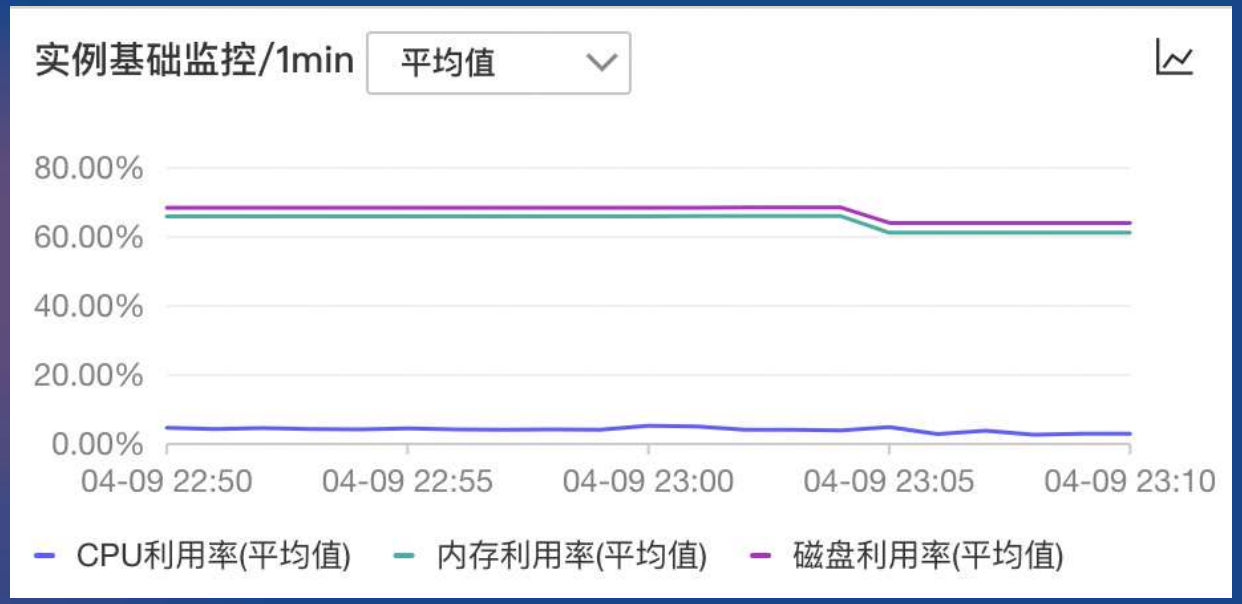
持续深耕监控及可观测领域，在可观测方案架构及落地方面有着多年实践经验与深刻见解。目前，作为eBPF探针及K8s监控的研发负责人，积极探索容器可观测及关联场景，为企业提供开箱即用的可观测产品与服务。

1. Profiling 与 Golang

2. Continuous Profiling

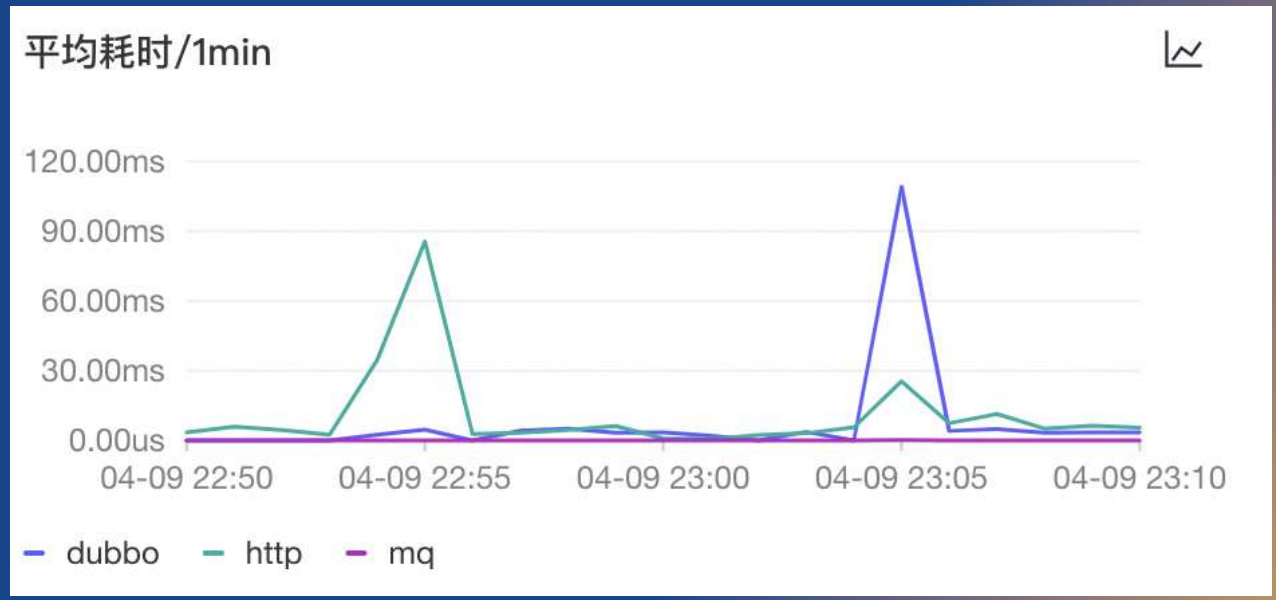
3. Golang Runtime 诊断

为什么需要 Profiling?



OPTIMAZATION

INCIDENTS



Golang Profiling 生态

Profiler	Methodology	Deployment	Traces Lib/Sys Calls?	overhead
gopprof	Visualization	Recompile & Rerun	Yes	Very High (20x*)
delve	run-time instrumentation	Profile any running process	Yes	vary
gperftools	run-time instrumentation	Recompile & Rerun	Yes	vary
eBPF	Attach probe to kernel or user elf	Profile any running process	Yes	Low

四个挑战：

- 开销较高
- 侵入性强
- 即时诊断能力弱
- 可扩展性弱

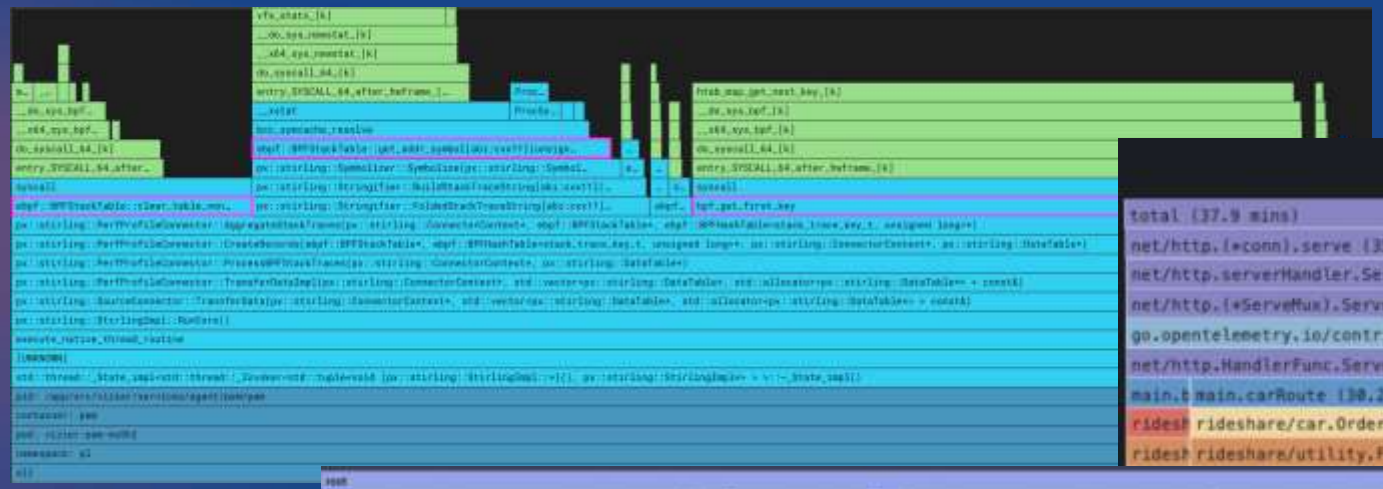
1. Profiling 与 Golang

2. Continuous Profiling

3. Golang Runtime 诊断

什么是 Continuous Profiling?

Continuous Profiling 是持续监控和记录程序性能特征的工具，能够协助开发者充分保留现场并发现性能瓶颈。



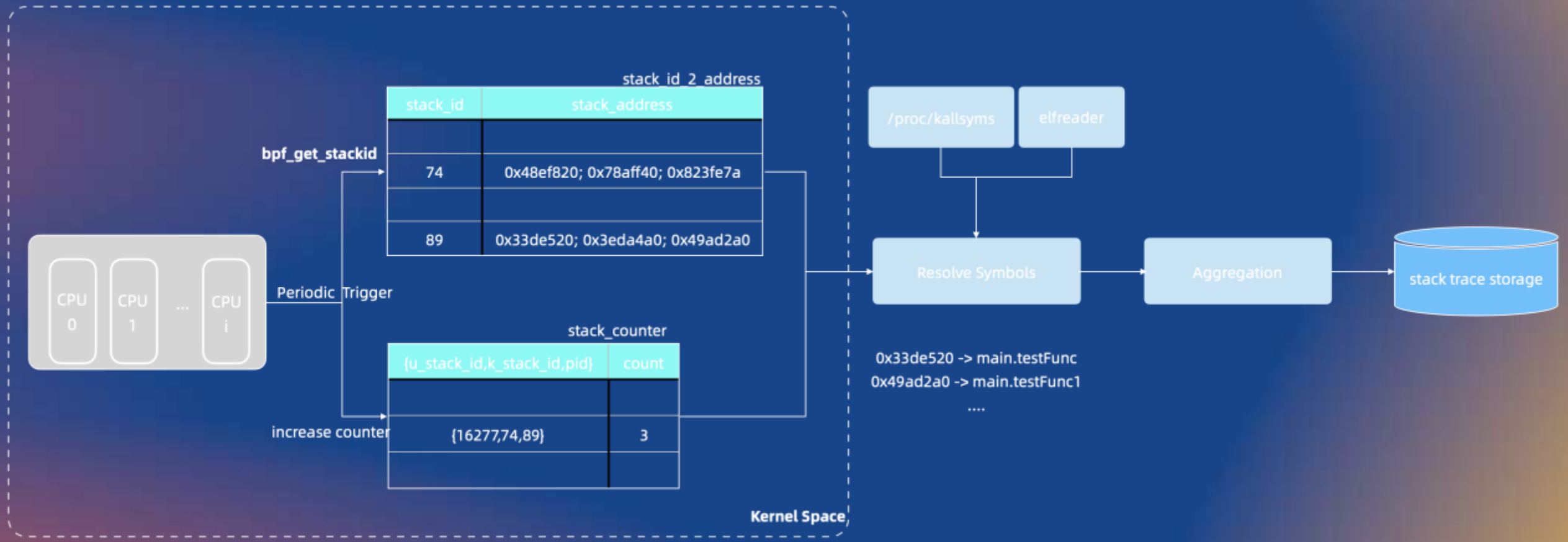
Pixie

Parca

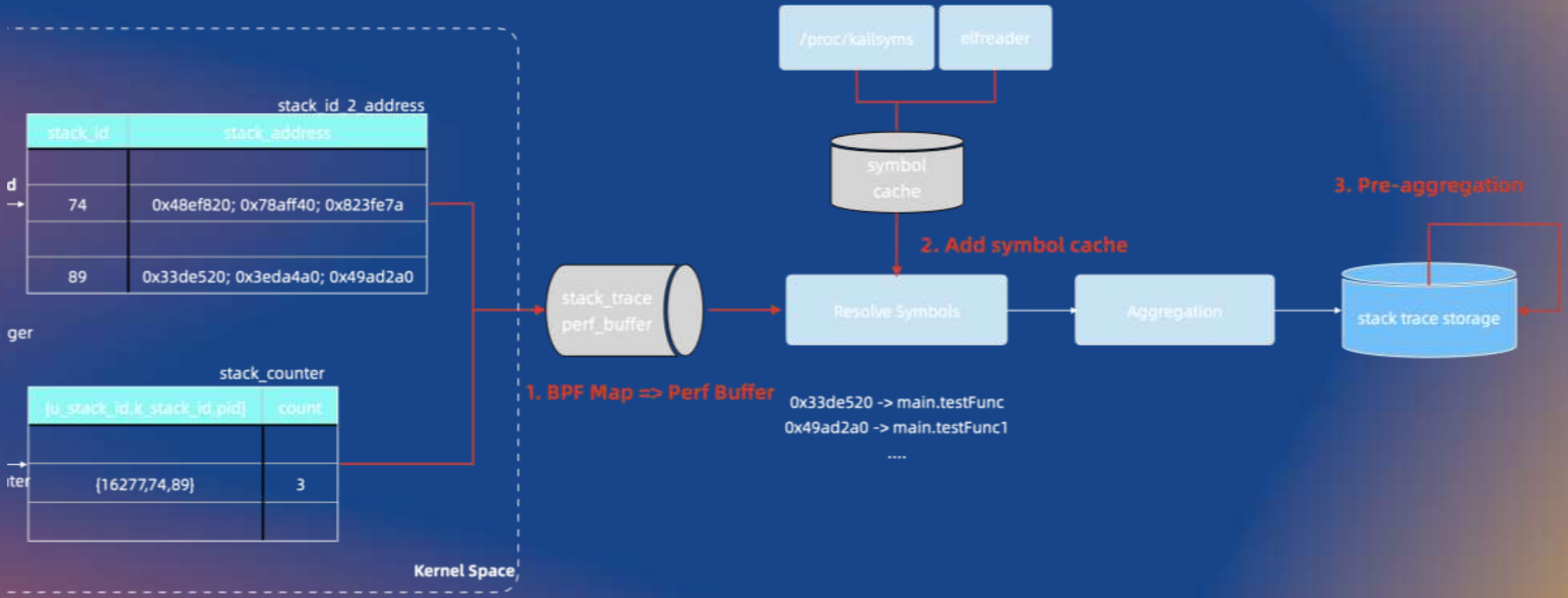
Pyroscope

CPU Continuous Profiling —— 架构

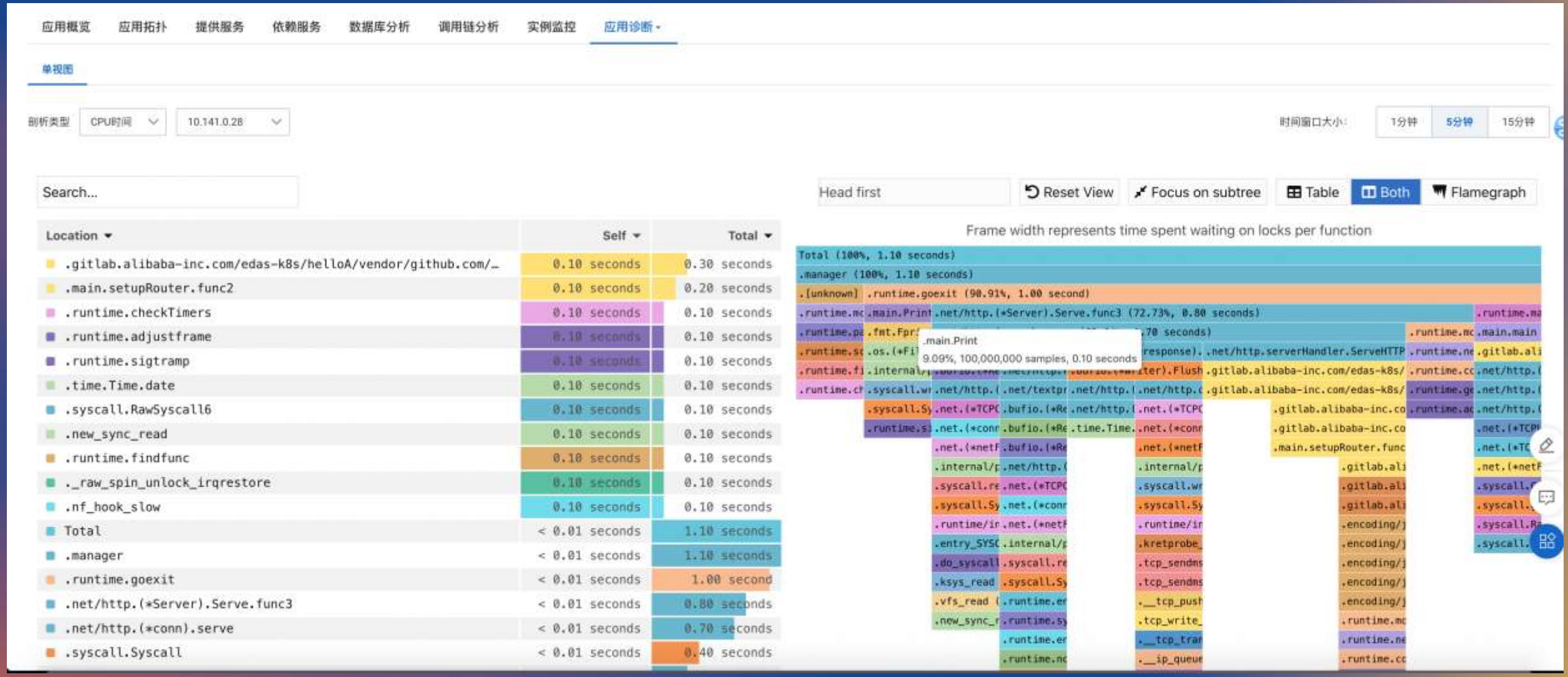
打造低开销、无侵入的工具



CPU Continuous Profiling —— 性能优化



成果展示


应用概览
应用拓扑
提供服务
依赖服务
数据库分析
调用链分析
实例监控
应用诊断
单视图
剖析类型
CPU时间
10.141.0.28
时间窗口大小:
1分钟
5分钟
15分钟
Search...
Head first
Reset View
Focus on subtree
Table
Both
Flamegraph
Location
Self
Total
.gitlab.alibaba-inc.com/edas-k8s/helloA/vendor/github.com/_ 0.10 seconds 0.30 seconds
.main.setupRouter.func2 0.10 seconds 0.20 seconds
.runtime.checkTimers 0.10 seconds 0.10 seconds
.runtime.adjustframe 0.10 seconds 0.10 seconds
.runtime.sigtramp 0.10 seconds 0.10 seconds
.time.Time.date 0.10 seconds 0.10 seconds
.syscall.RawSyscall6 0.10 seconds 0.10 seconds
.new_sync_read 0.10 seconds 0.10 seconds
.runtime.findfunc 0.10 seconds 0.10 seconds
._raw_spin_unlock_irqrestore 0.10 seconds 0.10 seconds
.nf_hook_slow 0.10 seconds 0.10 seconds
Total < 0.01 seconds 1.10 seconds
.manager < 0.01 seconds 1.10 seconds
.runtime.goexit < 0.01 seconds 1.00 second
.net/http.(*Server).Serve.func3 < 0.01 seconds 0.80 seconds
.net/http.(*conn).serve < 0.01 seconds 0.70 seconds
.syscall.Syscall < 0.01 seconds 0.40 seconds
Frame width represents time spent waiting on locks per function
Total (100%, 1.10 seconds)
.manager (100%, 1.10 seconds)
.[unknown]
.runtime.goexit (90.91%, 1.00 second)
.main.Print
9.09%, 100,000,000 samples, 0.10 seconds

1. Profiling 与 Golang

2. Continuous Profiling

3. Golang Runtime 诊断

Runtime 诊断 —— 概述

Runtime 诊断能够对运行中的应用进行 diagnose，能够帮助开发者发现预期外的异常。

	delve	eBPF
侵入性	无侵入	无侵入
环境要求	较高	较低
性能	一般	较高
扩展性	较低	较高

Runtime 诊断 —— eBPF 面临的问题

- Golang 的 ABI 规范不同于C 语言
- 不同版本编译器遵循的 ABI 规范也不同

- 识别参数类型
- 解析参数的Value
- Golang 内置了很多复杂的数据结构

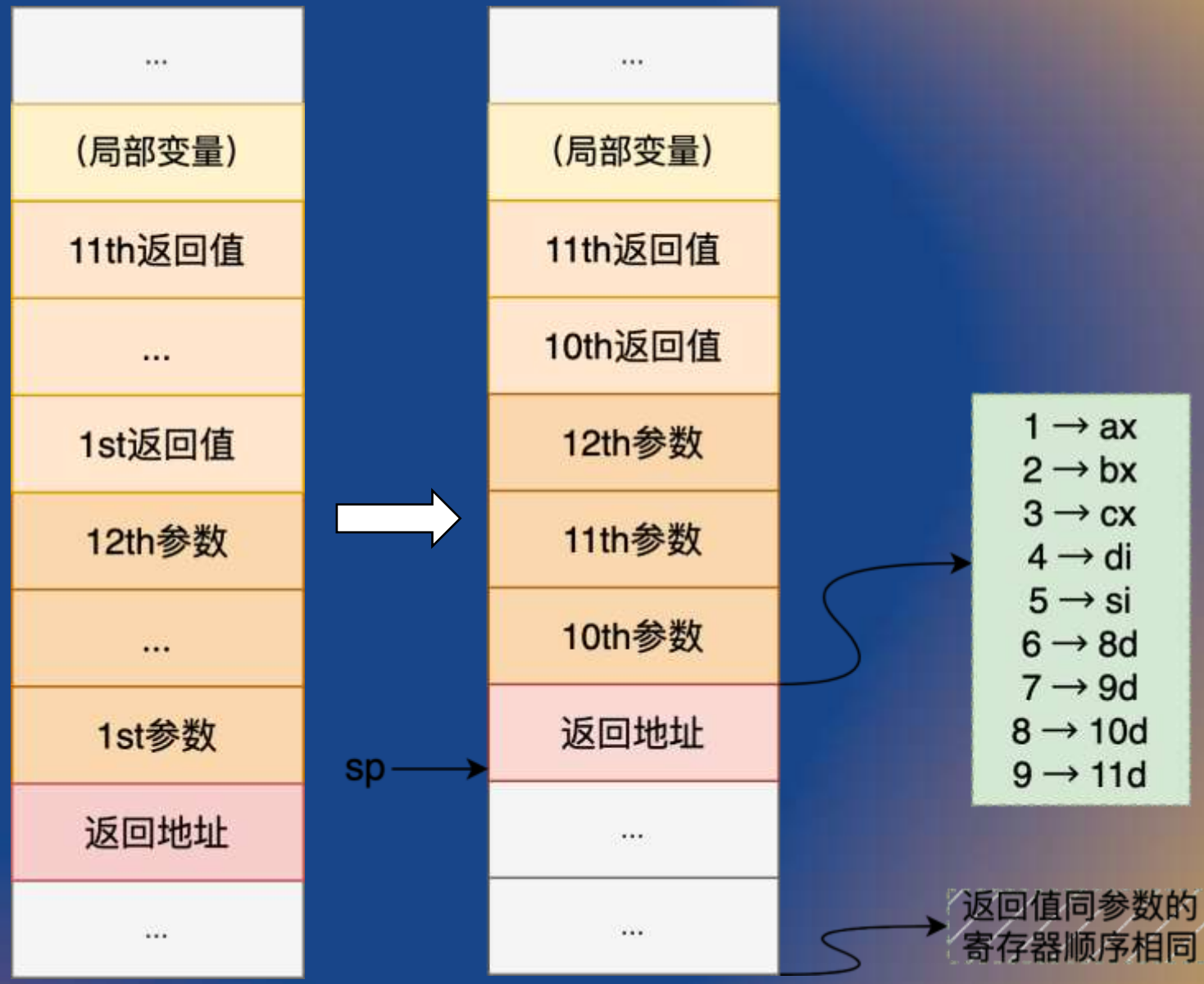
- uretprobe 的实现原理与 Golang 语言的设计理念有冲突

Golang 栈帧介绍

```
// 12 params, 11 return
func addFunc(x int, y int, z int, a, b, c int, d, e, f int, g, h, l int)
(int, int, int, int, int, int, int, int, int, int, int) {
  return 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
}

func main() {
  addFunc(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
}
```

- Go1.17以前，**所有的参数和返回值**都放在栈上
- Go1.17及以后，**前9个参数和返回值使用寄存器**，之后的参数和返回值使用栈



Runtime 诊断 —— 参数类型

```
type Test struct {
    data int32
    len int64
    ptr *int
    slice []bool
}

func addFunc(x *Test, y []int, z string, a, b, c int, d, e, f int,
g, h, l int) (int, int, int, int, int, int, int, int, int, int,
int) {
    return 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
}

func main() {
    arr := make([]int, 0, 4)
    arr = append(arr, 1)
    head := 1
    test := &Test {
        data: 628,
        len: 2002,
        ptr: &head,
        slice:  make([]bool, 0, 3),
    }

    addFunc(test, arr, "1", 4, 5, 6, 7, 8, 9, 10, 11, 12)
}
```

```
<1><4977d>: Abbrev Number: 3 (DW_TAG_subprogram)
    <4977e>   DW_AT_name        : main.addFunc
    <4978b>   DW_AT_low_pc      : 0x4585c0
    <49793>   DW_AT_high_pc     : 0x458604
    <4979b>   DW_AT_frame_base  : 1 byte block: 9c
(DW_OP_call_frame_cfa)
    <4979d>   DW_AT_decl_file   : 0x2
    <497a1>   DW_AT_external    : 1
 <2><497a2>: Abbrev Number: 18 (DW_TAG_formal_parameter)
    <497a3>   DW_AT_name        : x
    <497a5>   DW_AT_variable_parameter: 0
    <497a6>   DW_AT_decl_line   : 38
    <497a7>   DW_AT_type        : <0x4a366>
    <497ab>   DW_AT_location    : 0x6ef44 (location list)
    ......
 <2><498f7>: Abbrev Number: 0
```

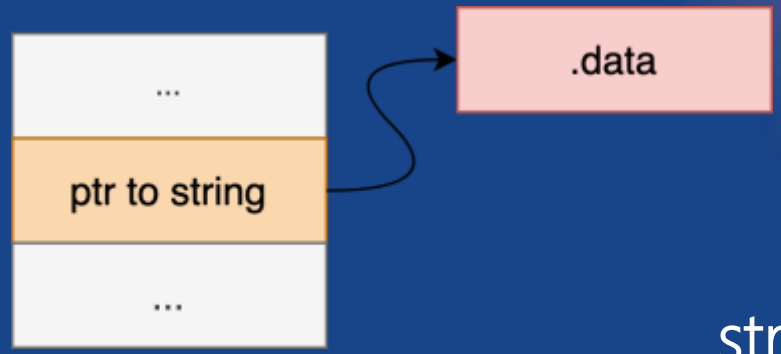
```
<1><4a366>: Abbrev Number: 35 (DW_TAG_pointer_type)
   <4a367>   DW_AT_name        : *main.Test
   <4a372>   DW_AT_type        : <0x4a3c4>
   <4a376>   Unknown AT value: 2900: 22
   <4a377>   Unknown AT value: 2904: 0x0
<1><4a37f>: Abbrev Number: 39 (DW_TAG_structure_type)
   <4a380>   DW_AT_name        : main.Test
   <4a38a>   DW_AT_byte_size   : 48
   <4a38b>   Unknown AT value: 2900: 25
   <4a38c>   Unknown AT value: 2904: 0x0
<2><4a394>: Abbrev Number: 24 (DW_TAG_member)
   <4a395>   DW_AT_name        : data
   <4a39a>   DW_AT_data_member_location: 0
   <4a39b>   DW_AT_type        : <0x4a3d3>
   <4a39f>   Unknown AT value: 2903: 0
   ....
```

Runtime 诊断 —— 参数解析

基础类型

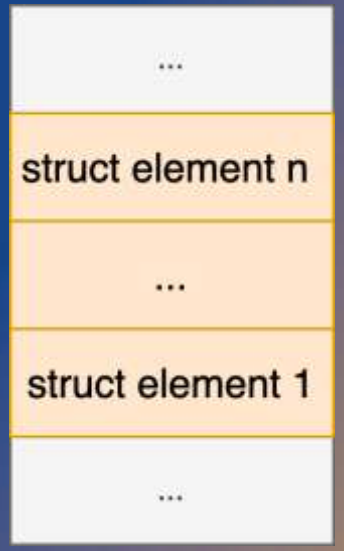
Type	64-bit		32-bit	
	Size	Align	Size	Align
bool, uint8, int8	1	1	1	1
uint16, int16	2	2	2	2
uint32, int32	4	4	4	4
uint64, int64	8	8	8	4
int, uint	8	8	4	4
float32	4	4	4	4
float64	8	8	8	4
complex64	8	4	8	4
complex128	16	8	16	4
uintptr, *T, unsafe.Pointer	8	8	4	4

string 类型

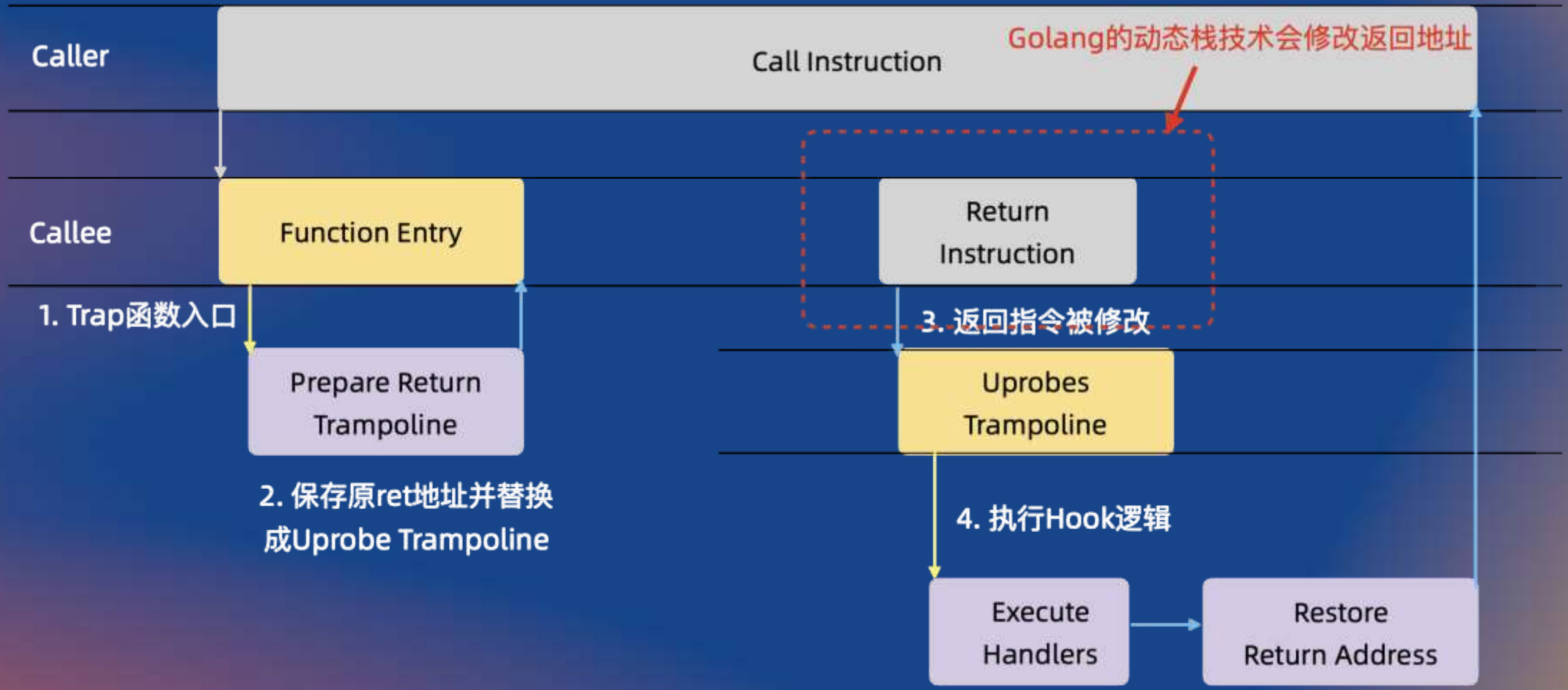


slice 类型

struct 类型



Runtime 诊断 —— uretprobe



Runtime 诊断 —— uretprobe

通过 uprobe 挂载到 return 符号地址来模拟 uretprobe 行为

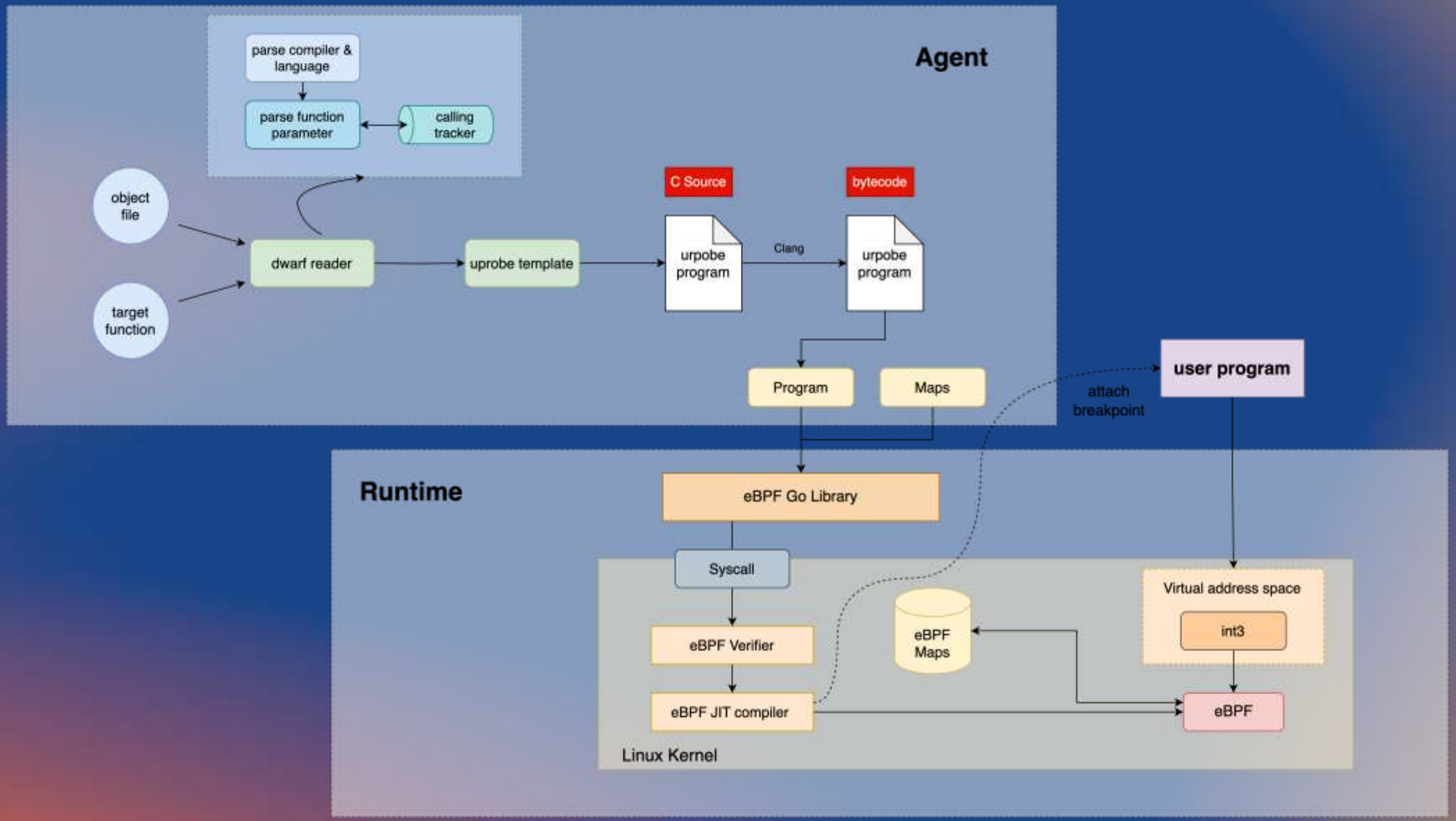
```
func uprobeTest(a, b, c int64, d float64) (int64, float64) {
    fmt.Printf( format: "uprobeTest return :%d\n", a*b*c)
    return a*b + c + b, d
}
```

```
000000000049f360 <main.uprobeTest>:
  49f360:       4c 8d 64 24 d0          lea    -0x30(%rsp),%r12
  49f365:       4d 3b 66 10             cmp    0x10(%r14),%r12
  49f369:       0f 86 21 02 00 00       jbe    49f590 <main.uprobeTest+0x230>
  49f36f:       48 81 ec b0 00 00 00    sub    $0xb0,%rsp
  49f376:       48 89 ac 24 a8 00 00    mov    %rbp,0xa8(%rsp)
  49f37d:       00
  49f37e:       48 8d ac 24 a8 00 00    lea    0xa8(%rsp),%rbp
  49f385:       00
  49f386:       48 89 84 24 b8 00 00    mov    %rax,0xb8(%rsp)
  49f38d:       00
  49f38e:       48 89 9c 24 c0 00 00    mov    %rbx,0xc0(%rsp)
... ...
  49f44a:       c3                      ret
... ...
  49f58f:       c3                      ret
... ...
```

```
SEC("uprobe/exit_uprobe_test")
int probe_ret_uprobe_test(struct pt_regs *ctx) {
    uint64_t ret0 = (uint64_t)(ctx->ax);
    uint64_t ret1 = (uint64_t)(ctx->bx);
    bpf_printk("[uretprobe] uprobe_test,
        ~r0:%ld, ~r1:%s", ret0, ret1);
}
```

```
obj->links.probe_exit_foo =
    bpf_program__attach_uprobe_opts(
            obj->progs.probe_ret_uprobe_test,
            -1,
            binary,
            0xea,
            null
);
... ...
```

Runtime 诊断 —— 架构



Runtime 诊断 —— 结果展示

```go
type Trace struct {
    a int
    b string
}

func demoTest(a int, b string, c []int, d *Trace) {
    e := a * len(b) * a
    f := a * e / len(c)
    fmt.Println(a, b, c, d, e, f)
}
```


```
{"funcName":"main.demoTest","args":[{"type":"INT","value":82},{"type":"STRING","value":"hello0"},{"type":"[]INT","value":[0,1,2,3]},{"type":"PTR","value":[{"type":"INT","value":0},{"type":"STRING","value":"trace test"}]}],"latency":12181}

{"funcName":"main.demoTest","args":[{"type":"INT","value":109},{"type":"STRING","value":"hello1"},{"type":"[]INT","value":[4,5,6,7]},{"type":"PTR","value":[{"type":"INT","value":1},{"type":"STRING","value":"trace test"}]}],"latency":9871}
```

THANKS