ROG 技术剖析与业务落地

演讲人：陈卓钰

SPEAKER: CHEN ZHUOYU

目录 | Contents

01

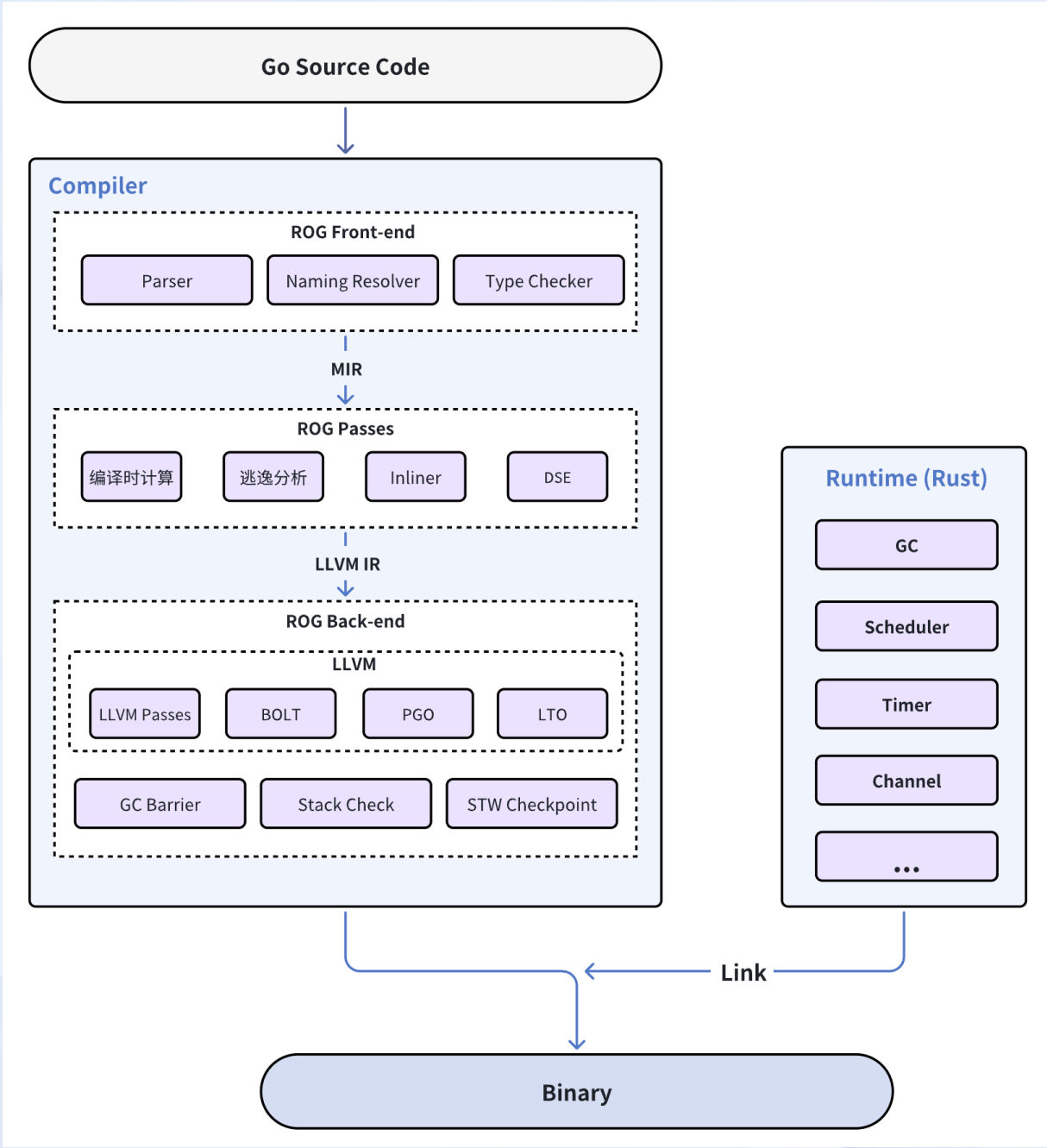
项目简介

ROG 的项目背景、功能特性，以及设计目标和边界

ROG 是什么

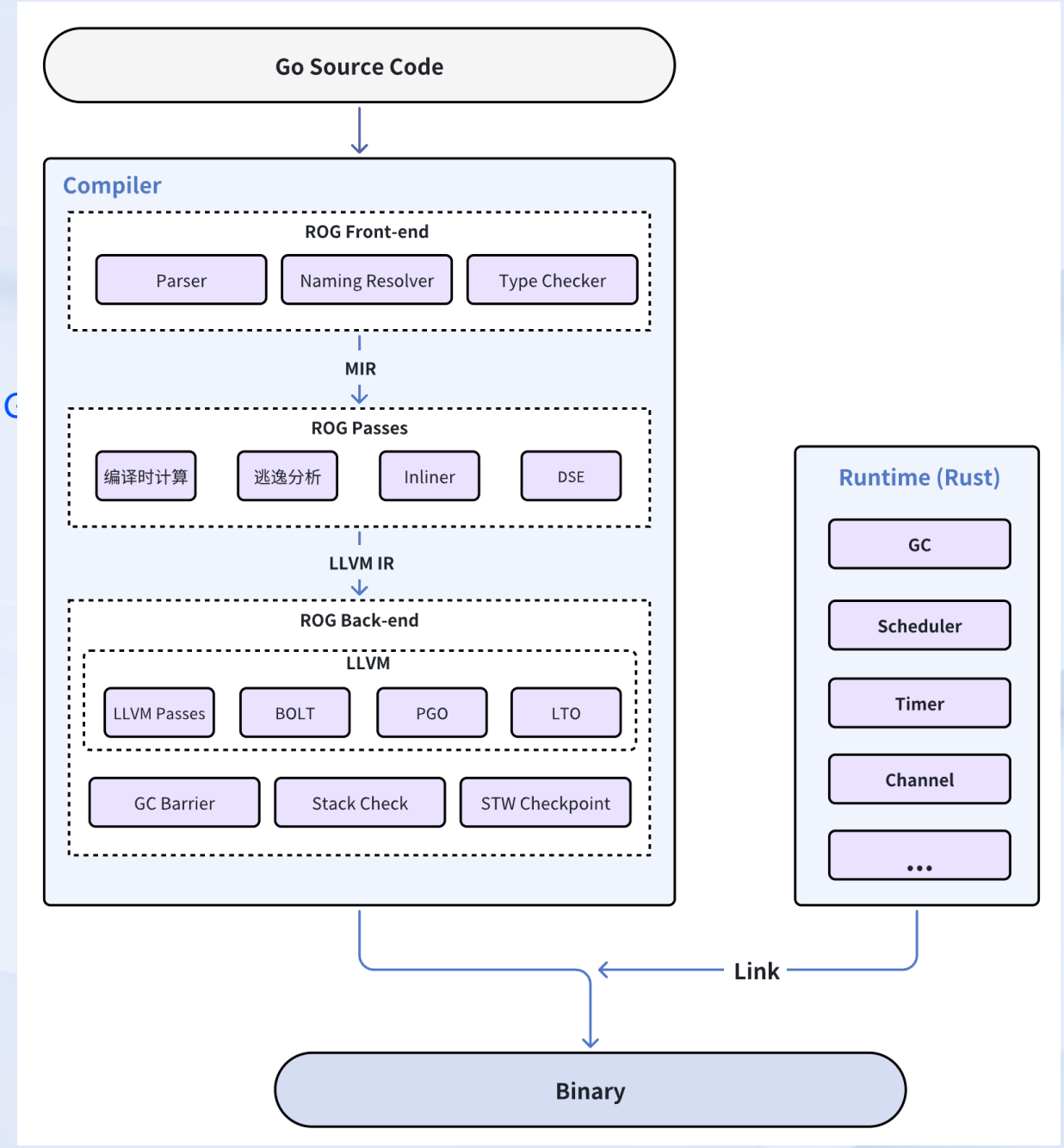
ROG 是什么

- 一个用 Rust 从头实现的 Go 编译器



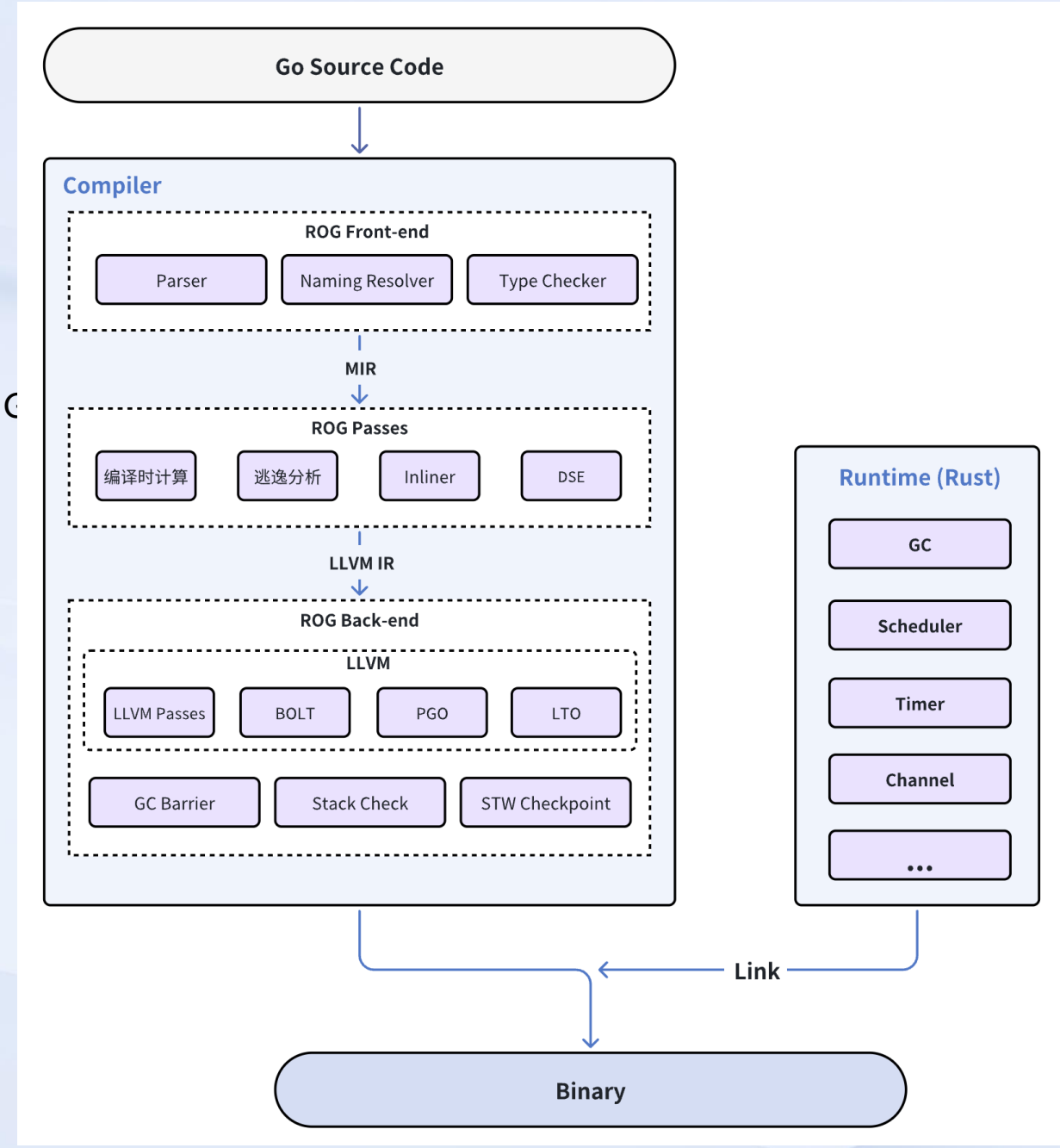
ROG 是什么

- 一个用 Rust 从头实现的 Go 编译器
- Refinement of Go / Reimplementation of Go / Rust on G



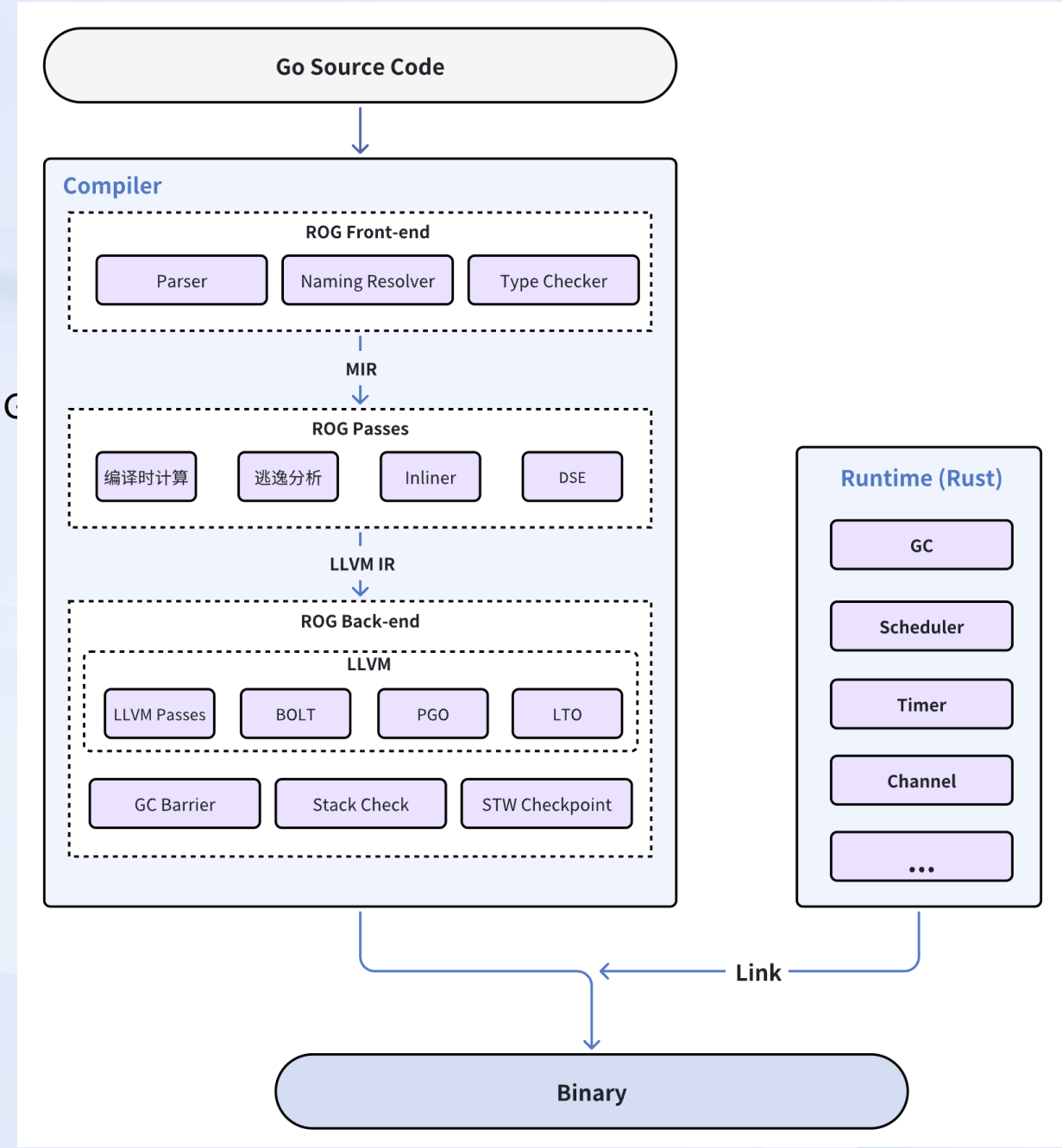
ROG 是什么

- 一个用 Rust 从头实现的 Go 编译器
- Refinement of Go / Reimplementation of Go / Rust on G
- 编译器前端从 rustc 改造而来，支持更多高级特性



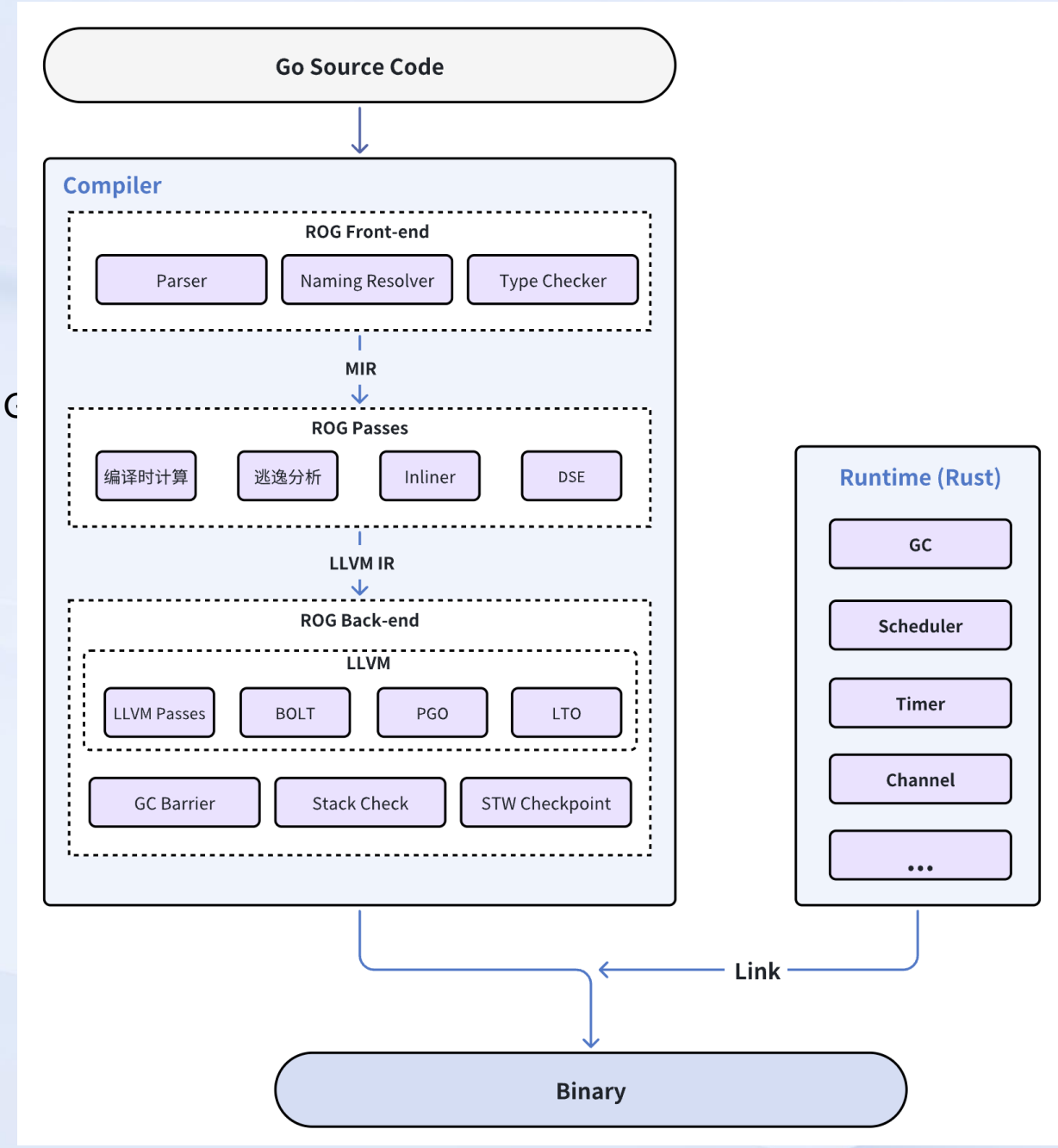
ROG 是什么

- 一个用 Rust 从头实现的 Go 编译器
- Refinement of Go / Reimplementation of Go / Rust on G
- 编译器前端从 rustc 改造而来，支持更多高级特性
- 后端基于 LLVM，是高性能的保障



ROG 是什么

- 一个用 Rust 从头实现的 Go 编译器
- Refinement of Go / Reimplementation of Go / Rust on G
- 编译器前端从 rustc 改造而来，支持更多高级特性
- 后端基于 LLVM，是高性能的保障
- Runtime 由 Rust 写成，使用顶会论文算法、无锁结构等



ROG 的背景

ROG 的背景

- 降本增效，而不增笑

ROG 的背景

- 降本增效，而不增笑
- 字节拥有海量使用 Go 进行开发的业务

ROG 的背景

- 降本增效，而不增笑
- 字节拥有海量使用 Go 进行开发的业务
- Go 的性能对于我们来说已经有点掣肘了：一味追求编译速度，放弃了大量的优化，从而牺牲了运行时性能

ROG 的背景

- 降本增效，而不增笑
- 字节拥有海量使用 Go 进行开发的业务
- Go 的性能对于我们来说已经有点掣肘了：一味追求编译速度，放弃了大量的优化，从而牺牲了运行时性能
- LLVM 是业界使用最广泛的编译器后端，能提供我们需要的各种能力

ROG 的背景

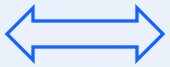
- 原版 Go 编译器缺少许多优化 —— 我们以下列 MaxArray 函数为例

```
func MaxArray(x, y []float64) {
    for i, c := range x {
        if y[i] > c {
            x[i] = y[i]
        }
    }
}
```

ROG 的背景

- 原版 Go 编译器缺少许多优化 —— 我们以下列 MaxArray 函数为例

```
12          XORL    CX, CX
13          JMP     main_MaxArray_pc25
14  main_MaxArray_pc22:
15          INCQ    CX
16  main_MaxArray_pc25:
17          CMPQ    BX, CX
18          JLE     main_MaxArray_pc58
19          MOVSD   (AX)(CX*8), X0
20          CMPQ    SI, CX
21          JLS     main_MaxArray_pc65
22          MOVSD   (DI)(CX*8), X1
23          UCOMISD X0, X1
24          JLS     main_MaxArray_pc22
25          MOVSD   X1, (AX)(CX*8)
26          JMP     main_MaxArray_pc22
27  main_MaxArray_pc58:
28          ADDQ    $16, SP
29          POPQ    BP
30          NOP
31          RET
```



```
        xorl    %eax, %eax
.LBB0_2:
        vmovupd (%rsi,%rax,8), %zmm0
        vmovupd 64(%rsi,%rax,8), %zmm1
        vmovupd 128(%rsi,%rax,8), %zmm2
        vmovupd 192(%rsi,%rax,8), %zmm3
        vmovupd 256(%rsi,%rax,8), %zmm4
        vmovupd 512(%rsi,%rax,8), %zmm5
        vmaxpd  (%rdi,%rax,8), %zmm0, %zmm0
        vmaxpd  64(%rdi,%rax,8), %zmm1, %zmm1
        vmaxpd  128(%rdi,%rax,8), %zmm2, %zmm2
        vmaxpd  192(%rdi,%rax,8), %zmm3, %zmm3
        vmovupd %zmm0, (%rdi,%rax,8)
        vmovupd %zmm1, 64(%rdi,%rax,8)
        vmovupd %zmm2, 128(%rdi,%rax,8)
        vmovupd %zmm3, 192(%rdi,%rax,8)
```

ROG 的目标与非目标

目标

- ✓ 语言特性与 Go 100% 兼容
- ✓ 高性能 runtime
- ✓ 支持 LLVM 高级特性
  - ✓ LTO
  - ✓ PGO
  - ✓ BOLT
- ✓ 可扩展

非目标

- 自举
- 编译速度快
- Plan9 汇编
- linkname 内部符号

02

同类现状

Go 编译器的现状、高性能 Go 所面临的挑战以及解决思路

Go 编译器的现状

Go 编译器的现状

- 市面上有许多第三方的 Go 编译器，但都存在各自的问题

Go 编译器的现状

- 市面上有许多第三方的 Go 编译器，但都存在各自的问题
- gccgo: 采用 GCC 作为后端，版本老旧，维护不积极，性能差

Setting up and using gccgo

Table of Contents

This document explains how to use gccgo, a compiler for the Go language. The gccgo compiler is a new frontend for GCC, the widely used GNU compiler. Although the frontend itself is under a BSD-style license, gccgo is normally used as part of GCC and is then covered by the GNU General Public License (the license covers gccgo itself as part of GCC; it does not cover code generated by gccgo).

Note that gccgo is not the `gc` compiler; see the Installing Go instructions for that compiler.

Go 编译器的现状

- 市面上有许多第三方的 Go 编译器，但都存在各自的问题
- gccgo: 采用 GCC 作为后端，版本老旧，维护不积极，性能差
- gollvm: 后端由 C++写成，前端是 gccgo 移植而来，特性支持不完整，开发不积极

Gollvm

Gollvm is an LLVM-based Go compiler. It incorporates “gofrontend” (a Go language front end written in C++ and shared with GCCGO), a bridge component (which translates from gofrontend IR to LLVM IR), and a driver that sends the resulting IR through the LLVM back end.

Gollvm is set up to be a subproject within the LLVM tools directory, similar to how things work for “clang” or “compiler-rt”: you check out a copy of the LLVM source tree, then within the LLVM tree you check out additional git repos.

Go 编译器的现状

- 市面上有许多第三方的 Go 编译器，但都存在各自的问题
- gccgo: 采用 GCC 作为后端，版本老旧，维护不积极，性能差
- gollvm: 后端由 C++写成，前端是 gccgo 移植而来，特性支持不完整，开发不积极
- TinyGo: 面向嵌入式，许多特性不支持

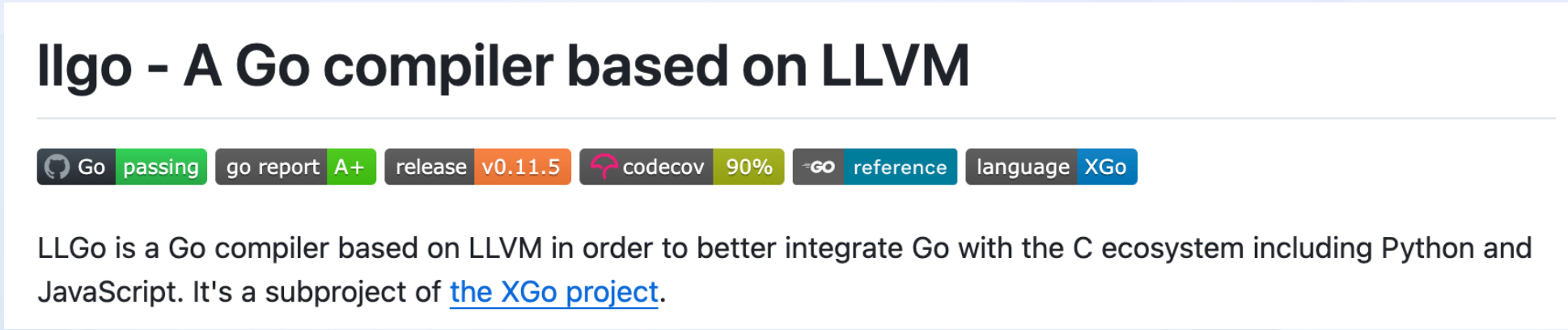
TinyGo - Go compiler for small places

TinyGo is a Go compiler intended for use in small places such as microcontrollers, WebAssembly (wasm/wasi), and command-line tools.

It reuses libraries used by the Go language tools alongside LLVM to provide an alternative way to compile programs written in the Go programming language.

Go 编译器的现状

- 市面上有许多第三方的 Go 编译器，但都存在各自的问题
- gccgo: 采用 GCC 作为后端，版本老旧，维护不积极，性能差
- gollvm: 后端由 C++写成，前端是 gccgo 移植而来，特性支持不完整，开发不积极
- TinyGo: 面向嵌入式，许多特性不支持
- llgo: 只是采用了 Go 的语法的一种胶水语言



高性能 Go 所面临的挑战

高性能 Go 所面临的挑战

其本质都是 LLVM 与 Go 的适配问题

高性能 Go 所面临的挑战

- GC Write Barrier

高性能 Go 所面临的挑战

- GC Write Barrier
  - 假设 p 和 q 都是指针，那么对于 *p = q 这样的语句是需要插入 GC 写屏障的

```
        CMPL    runtime.writeBarrier(SB), $0
        PCDATA  $0, $-2
        JEQ     command-line-arguments_StorePtr_pc36
        CALL    runtime.gcWriteBarrier2(SB)
        MOVQ    BX, (R11)
        MOVQ    (AX), CX
        MOVQ    CX, 8(R11)
command-line-arguments_StorePtr_pc36:
        MOVQ    BX, (AX)
```

高性能 Go 所面临的挑战

- GC Write Barrier
  - 假设 p 和 q 都是指针，那么对于 *p = q 这样的语句是需要插入 GC 写屏障的
  - 需要在 LLVM 里增加相关的 pass 来处理写屏障

高性能 Go 所面临的挑战

- GC Write Barrier
- 栈空间的动态管理

高性能 Go 所面临的挑战

- GC Write Barrier
- 栈空间的动态管理
  - 函数栈空间不足时需要扩栈，而栈空间富裕时可以回收多余的空间

高性能 Go 所面临的挑战

- GC Write Barrier
- 栈空间的动态管理
  - 函数栈空间不足时需要扩栈，而栈空间富裕时可以回收多余的空间
  - 通过 LLVM 里的 pass 插入 stack check prologue 来实现

```
1   store_ptr:
2           cmpq    %fs:128, %rsp
3           jbe     .LBB0_3
4   .LBB0_1:…
16  .LBB0_4:…
20  .LBB0_3:
21          leaq    -8(%rsp), %r11
22          callq   rog_morestack_abi
23          jmp     .LBB0_1
```

高性能 Go 所面临的挑战

- GC Write Barrier
- 栈空间的动态管理
- 抢占式调度

高性能 Go 所面临的挑战

- GC Write Barrier
- 栈空间的动态管理
- 抢占式调度
  - 为了避免长时间运行的任务阻塞调度器，通过 LLVM 插入 checkpoint 来实现

```
store_ptr:
        movq    rog_checkpoint_switch@GOTPCREL(%rip), %r11
        cmpl    $0, (%r11)
        jne     .LBB0_3
.LBB0_1:…
.LBB0_4:…
.LBB0_3:
        callq   rog_checkpoint_abi
        jmp     .LBB0_1
```

03

实现细节

两个关键组件：内存分配器以及 GC 的设计思路与实现

内存分配器

内存分配器

- 市面上有不少开源的分配器，但是都不满足需求

内存分配器

- 市面上有不少开源的分配器，但是都不满足需求
- jemalloc: 难以绑定元数据，难以索引对象，难以指定地址范围，不能批量释放，近期停止维护

```
jemalloc is a general purpose malloc(3) implementation that emphasizes
fragmentation avoidance and scalable concurrency support.  jemalloc first came
into use as the FreeBSD libc allocator in 2005, and since then it has found its
way into numerous applications that rely on its predictable behavior.  In 2010
jemalloc development efforts broadened to include developer support features
such as heap profiling and extensive monitoring/tuning hooks.  Modern jemalloc
releases continue to be integrated back into FreeBSD, and therefore versatility
remains critical.  Ongoing development efforts trend toward making jemalloc
among the best allocators for a broad range of demanding applications, and
eliminating/mitigating weaknesses that have practical repercussions for real
world applications.
```

内存分配器

- 市面上有不少开源的分配器，但是都不满足需求
- jemalloc: 难以绑定元数据，难以索引对象，难以指定地址范围，不能批量释放，近期停止维护
- tcmalloc / mimalloc: 无法绑定元数据，不能批量释放

TCMalloc

This repository contains the TCMalloc C++ code.

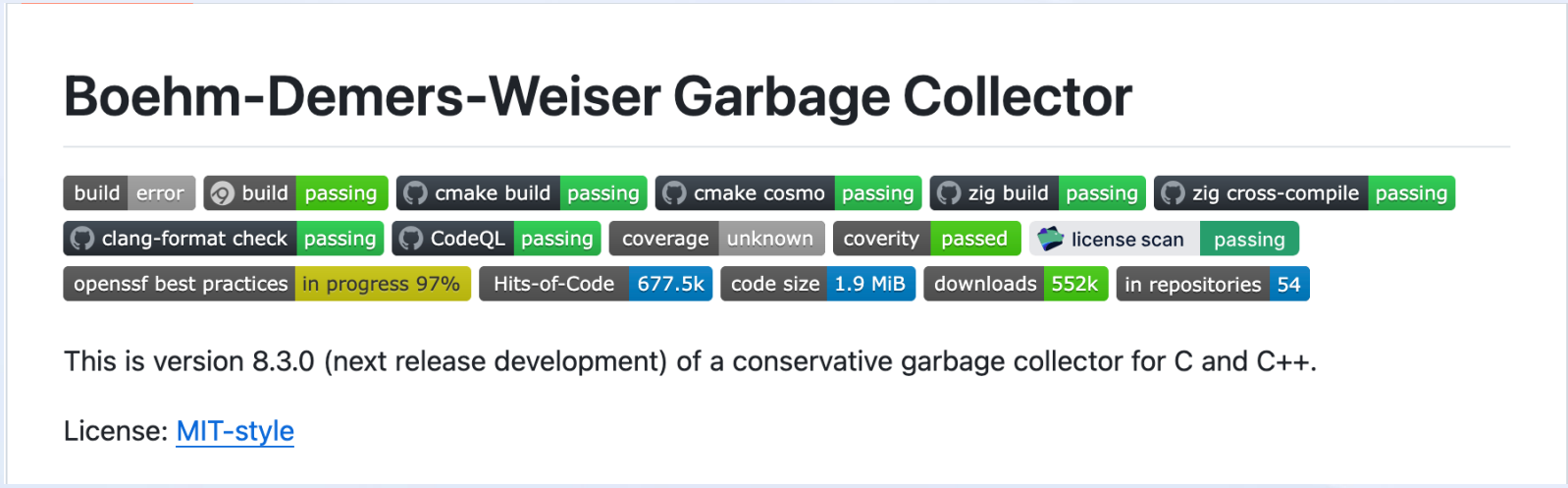
TCMalloc is Google's customized implementation of C's `malloc()` and C++'s `operator new` used for memory allocation within our C and C++ code. TCMalloc is a fast, multi-threaded malloc implementation.

mimalloc

mimalloc (pronounced "me-malloc") is a general purpose allocator with excellent performance characteristics. Initially developed by Daan Leijen for the runtime systems of the Koka and Lean languages.

内存分配器

- 市面上有不少开源的分配器，但是都不满足需求
- jemalloc: 难以绑定元数据，难以索引对象，难以指定地址范围，不能批量释放，近期停止维护
- tcmalloc / mimalloc: 无法绑定元数据，不能批量释放
- bdw-gc: 多线程性能差，协程支持困难，仅支持 STW GC



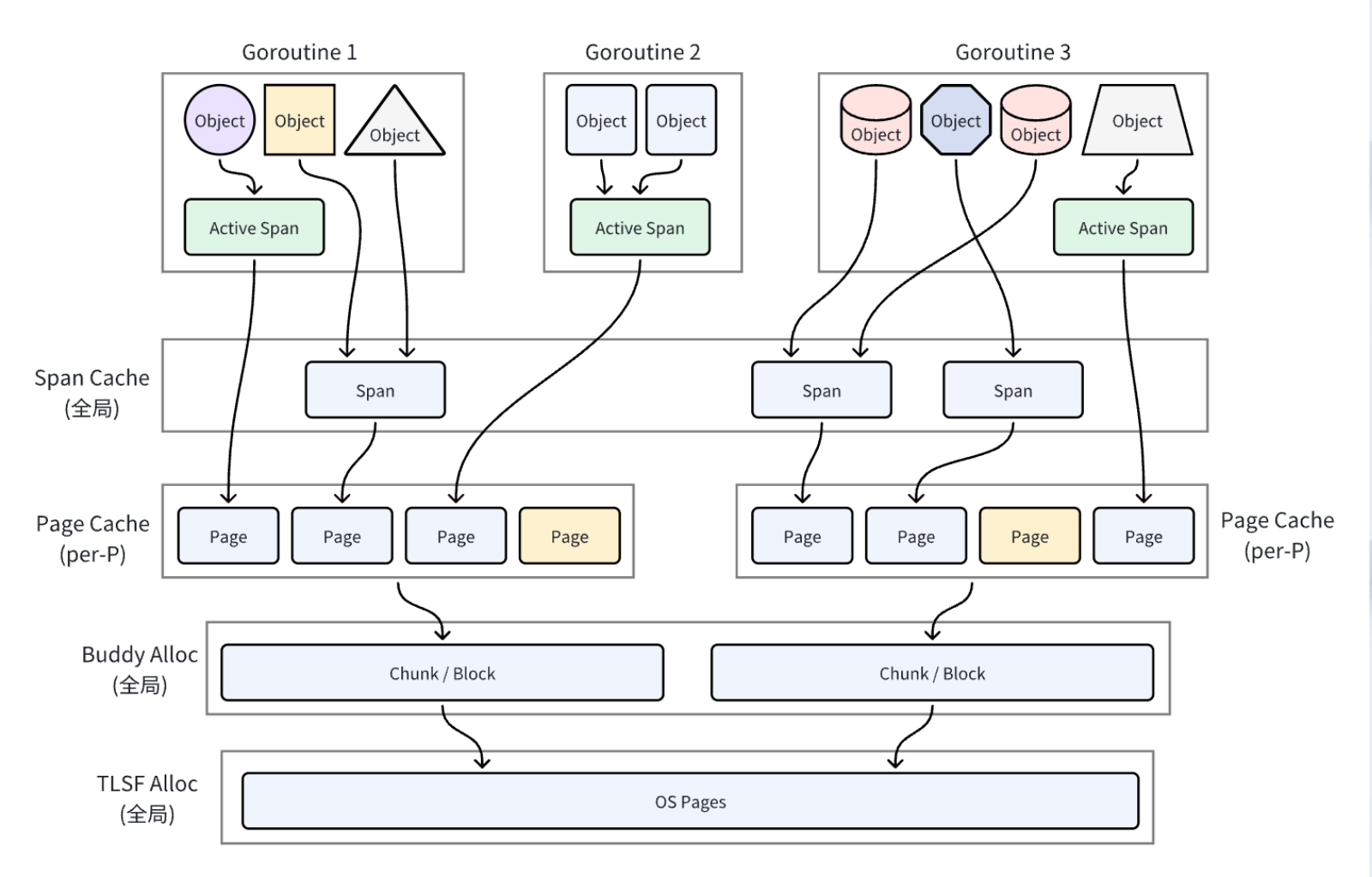
内存分配器

- 市面上有不少开源的分配器，但是都不满足需求
- jemalloc: 难以绑定元数据，难以索引对象，难以指定地址范围，不能批量释放，近期停止维护
- tcmalloc / mimalloc: 无法绑定元数据，不能批量释放
- bdw-gc: 多线程性能差，协程支持困难，仅支持 STW GC

所以 ROG 需要设计自己的内存分配器

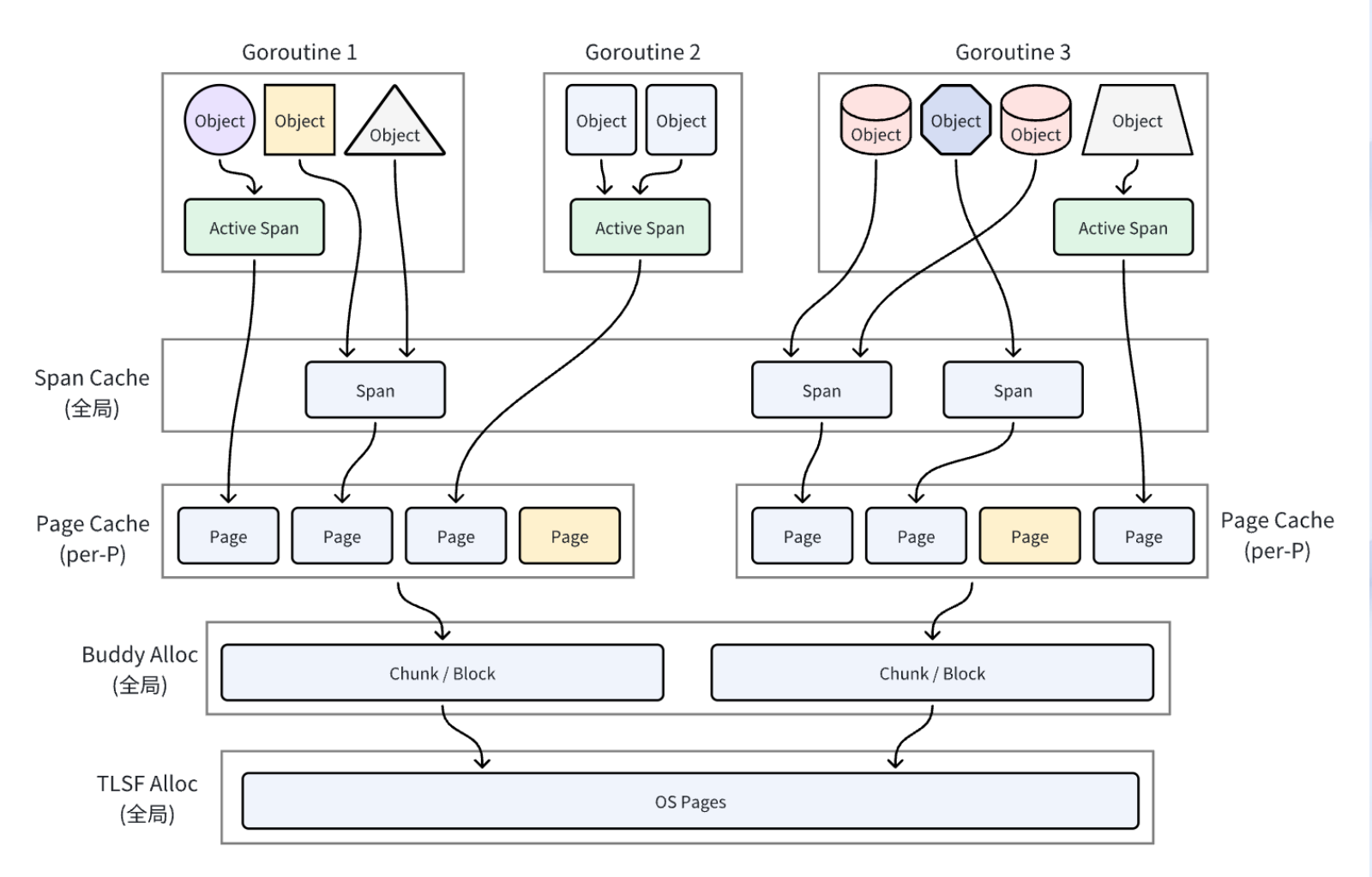
内存分配器

- 采用分级内存管理，高性能



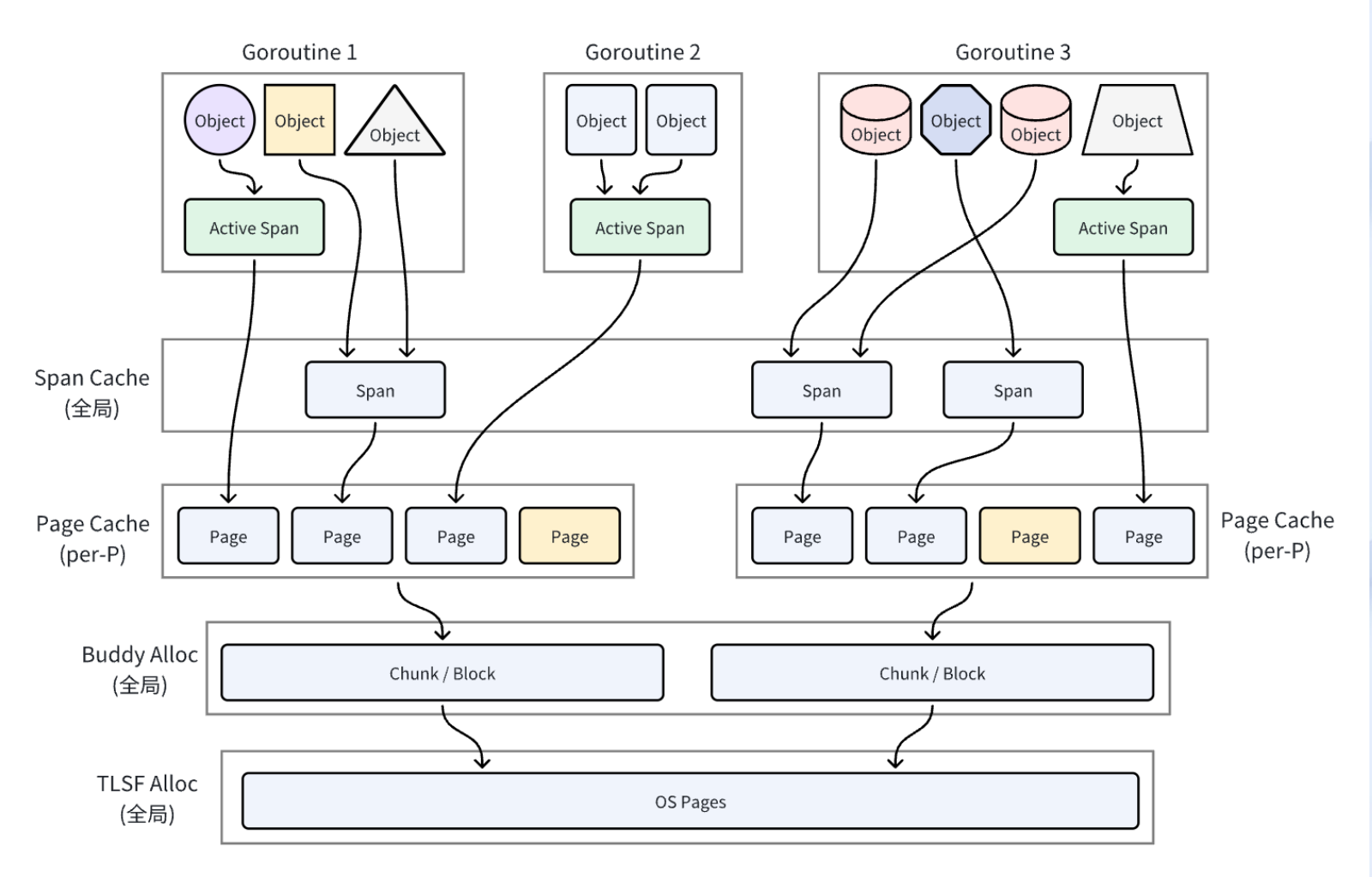
内存分配器

- 采用分级内存管理，高性能
- 最大支持 1TB 堆大小



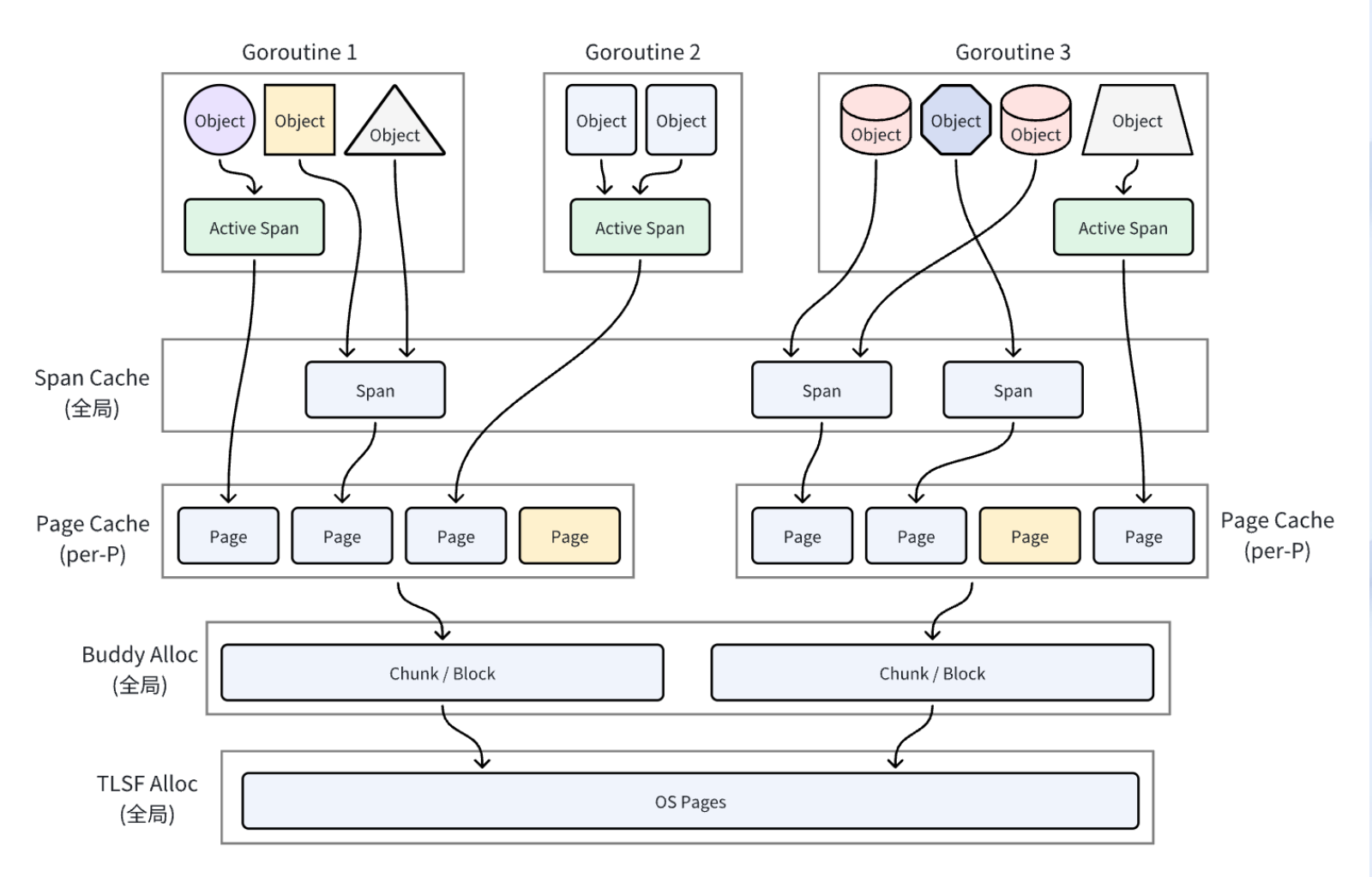
内存分配器

- 采用分级内存管理，高性能
- 最大支持 1TB 堆大小
- 支持运行时对象索引



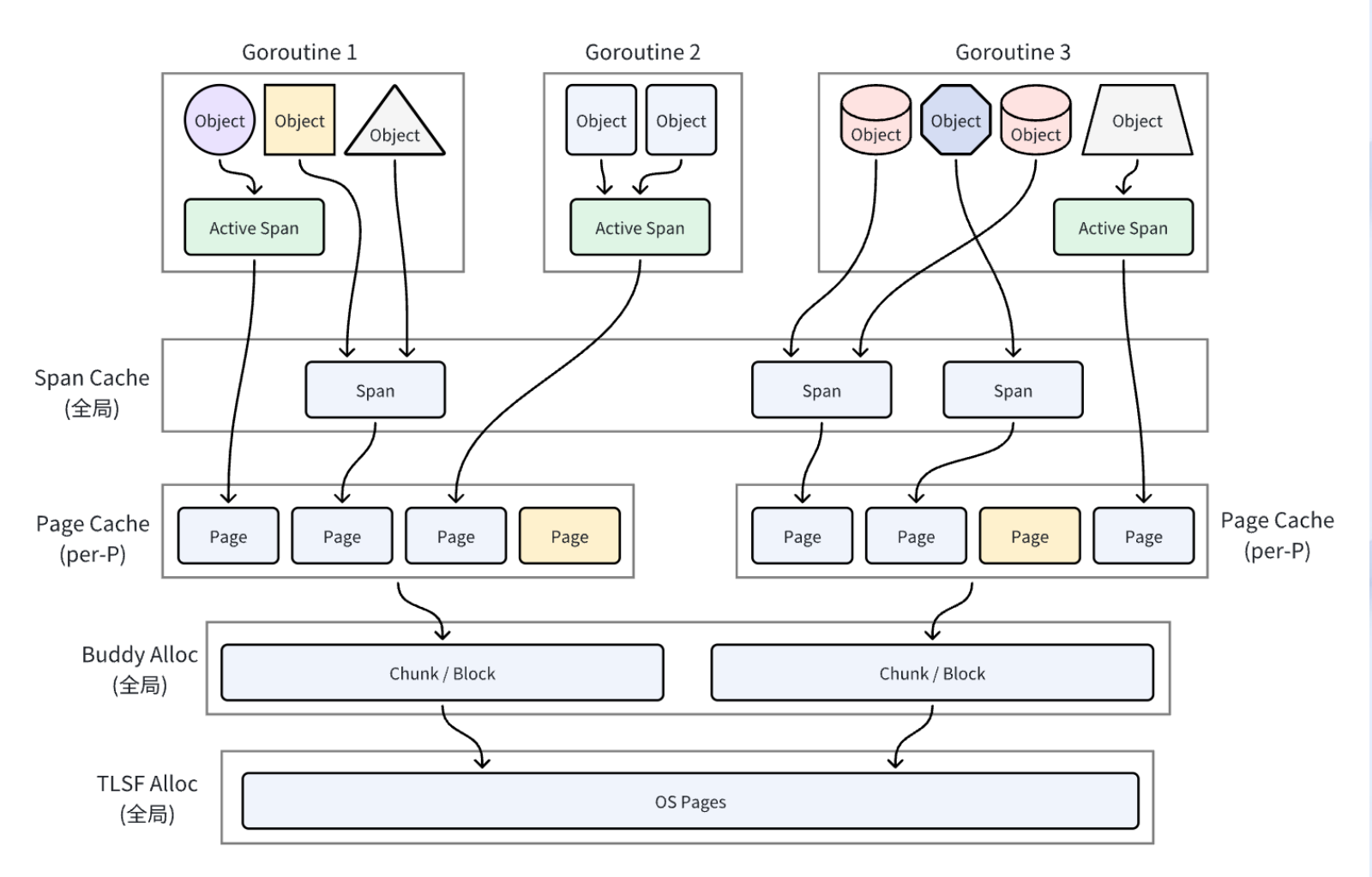
内存分配器

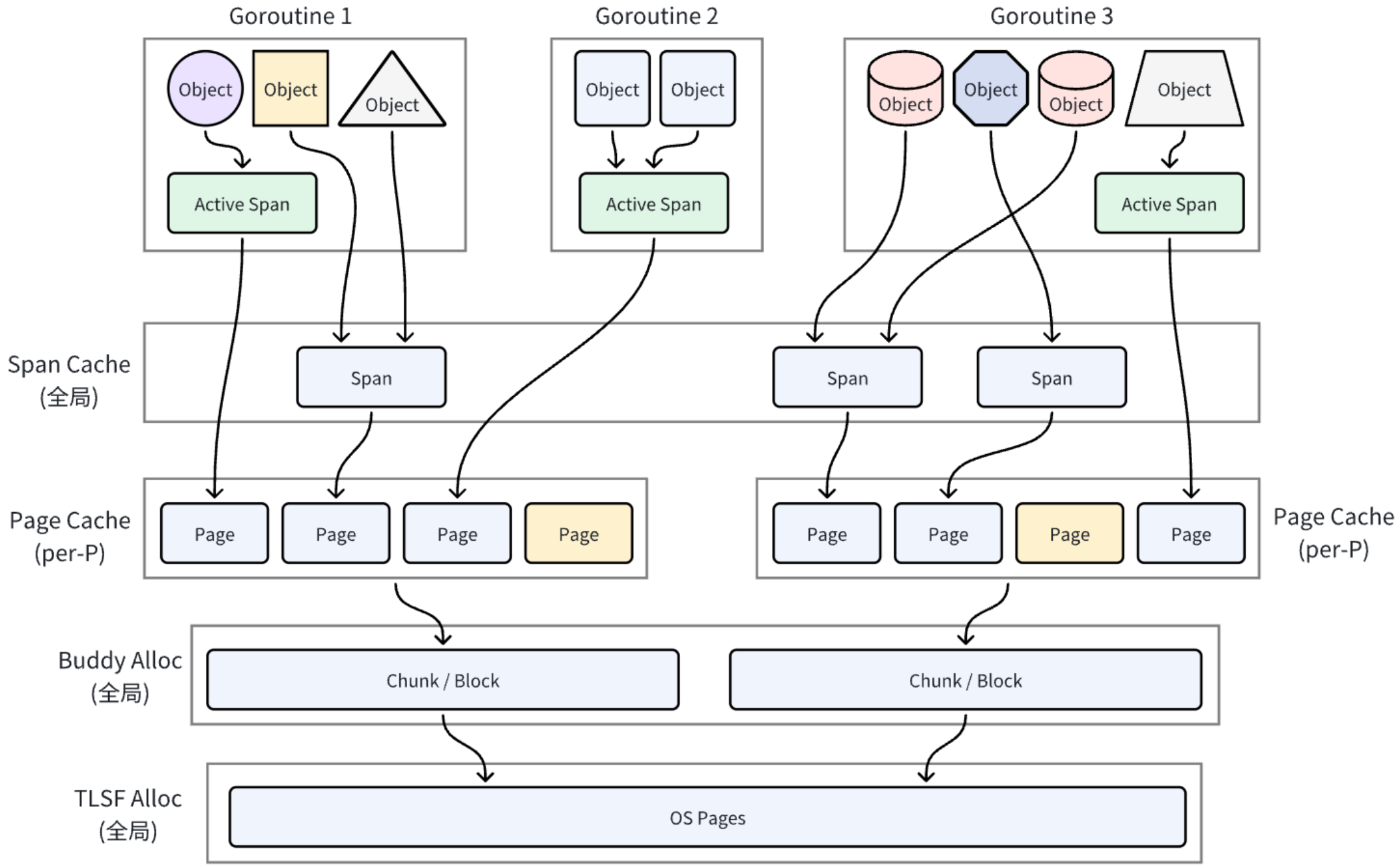
- 采用分级内存管理，高性能
- 最大支持 1TB 堆大小
- 支持运行时对象索引
- 支持批量释放（配合 GC）



内存分配器

- 采用分级内存管理，高性能
- 最大支持 1TB 堆大小
- 支持运行时对象索引
- 支持批量释放（配合 GC）
- 优秀的缓存局部性




Goroutine 1
Goroutine 2
Goroutine 3
Object
Object
Object
Object
Object
Object
Object
Object
Object
Active Span
Active Span
Active Span
Span Cache
(全局)
Span
Span
Span
Page Cache
(per-P)
Page
Page
Page
Page
Page
Page
Page
Page
Page Cache
(per-P)
Buddy Alloc
(全局)
Chunk / Block
Chunk / Block
TLSF Alloc
(全局)
OS Pages

垃圾回收器（GC）

垃圾回收器（GC）

- 支持多种 GC 触发策略

垃圾回收器（GC）

- 支持多种 GC 触发策略
- 支持多种 GC 算法（STW GC，三色 GC），目前默认为 STW GC

垃圾回收器（GC）

- 支持多种 GC 触发策略
- 支持多种 GC 算法（STW GC，三色 GC），目前默认为 STW GC

- 暂停所有的 P 和 G（Stop the world）
- 标记所有的 GC Roots（全局变量、栈）

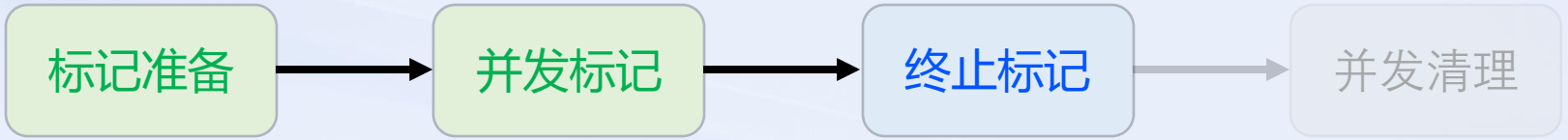
垃圾回收器（GC）

- 支持多种 GC 触发策略
- 支持多种 GC 算法（STW GC，三色 GC），目前默认为 STW GC

- 从 GC Roots 开始多线程并发标记存活对象
- 将当前对象引用的、未标记的对象加入队列
- 直到所有对象都被标记

垃圾回收器（GC）

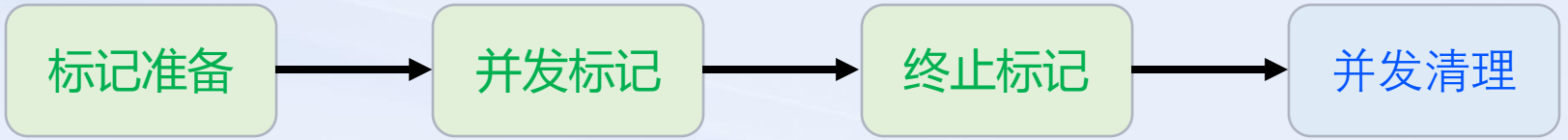
- 支持多种 GC 触发策略
- 支持多种 GC 算法（STW GC，三色 GC），目前默认为 STW GC



- 处理析构函数（Finalizer）
- 递归标记析构函数所关联的对象
- 复活关联的对象，摘除并入队相应的析构函数

垃圾回收器（GC）

- 支持多种 GC 触发策略
- 支持多种 GC 算法（STW GC，三色 GC），目前默认为 STW GC



- 清理未标记的对象
- 重启世界（Start the World）
- 调用析构函数

04 性能数据

综合压测数据以及业务落地之后产生的实际收益展示

多线程计算 π

```
[[chenzhuoyu@epyc multi-pi]$ time ./multi-pi
Calculated π: 3.1415926535897927
Calculated π (hex) : 0x400921fb54442d17
math.Pi (hex)      : 0x400921fb54442d18
Difference between math.Pi and calculated: -4.440892098500626e-16

real    0m2.912s
user    0m11.353s
sys     0m0.150s
[chenzhuoyu@epyc multi-pi]$ time ./multi-pi-go
Calculated π: 3.1415926535897927
Calculated π (hex) : 0x400921fb54442d17
math.Pi (hex)      : 0x400921fb54442d18
Difference between math.Pi and calculated: -4.440892098500626e-16

real    0m12.994s
user    0m51.501s
sys     0m0.047s
[chenzhuoyu@epyc multi-pi]$
```

Kitex Benchmark

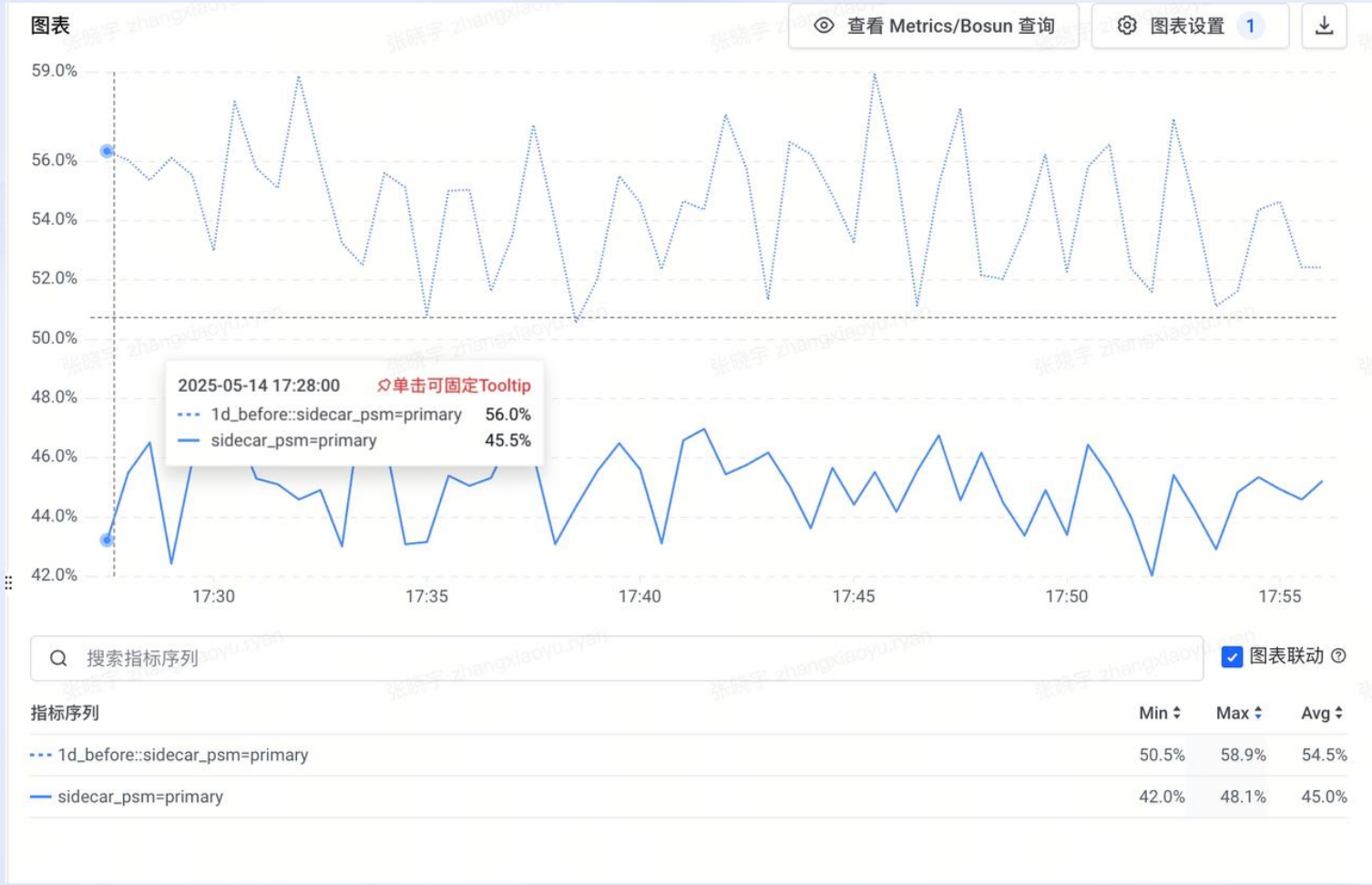
ROG_OPTS=@./config.toml taskset -c 0-3 ./kitex-benchmark-server

```
Client [kitex] running with [ ]
2025/09/17 10:57:53 warmup start
2025/09/17 10:57:54 warmup done
Info: [KITEX] start benching [2025-09-17 10:57:54.298629434 +0800 CST m=+0.353069271], concurrent: 200,
Info: [KITEX]: finish benching [2025-09-17 10:58:21.710238958 +0800 CST m=+27.764678795], took 27411 ms
Info: [KITEX]: requests total: 10000000, failed: 0
Info: [KITEX]: TPS: 364808.91, TP99: 1.45ms, TP999: 2.45ms (b=4096 Byte, c=200, qps=0, n=10000000)
```

GOMAXPROCS=4 taskset -c 0-3 ./kitex-benchmark-server-go

```
Client [kitex] running with [ ]
2025/09/17 11:00:13 warmup start
2025/09/17 11:00:14 warmup done
Info: [KITEX] start benching [2025-09-17 11:00:14.123725507 +0800 CST m=+0.342705528], concurrent: 200,
Info: [KITEX]: finish benching [2025-09-17 11:00:44.42200782 +0800 CST m=+30.640987841], took 30298 ms
Info: [KITEX]: requests total: 10000000, failed: 0
Info: [KITEX]: TPS: 330051.85, TP99: 1.80ms, TP999: 4.69ms (b=4096 Byte, c=200, qps=0, n=10000000)
```

线上服务 A 平均 %CPU: 54.5% -> 45%

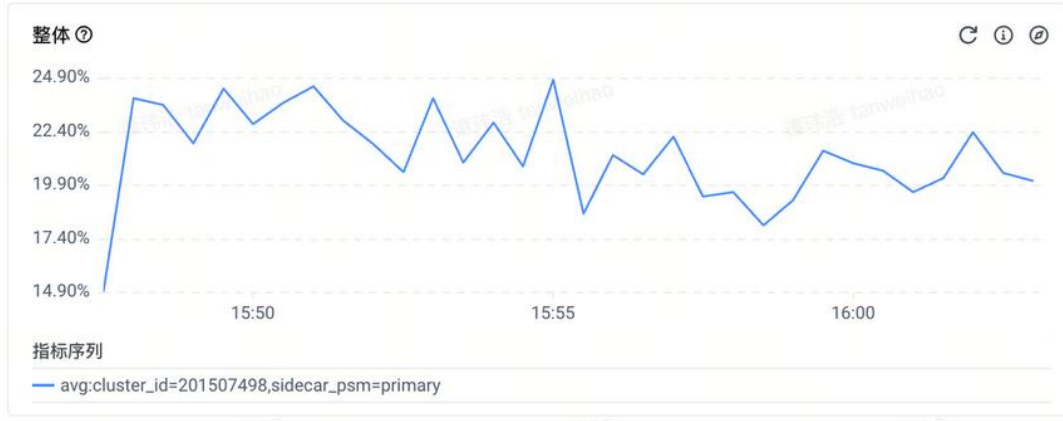


线上服务 B

平均 %CPU: ~36% -> ~22%

普通编译

ROG



问答时间

THANKS