Fiber优化

Assignee	Ⓛ LeoN
Status	In progress
Project	🍏 LioNet
Tags	

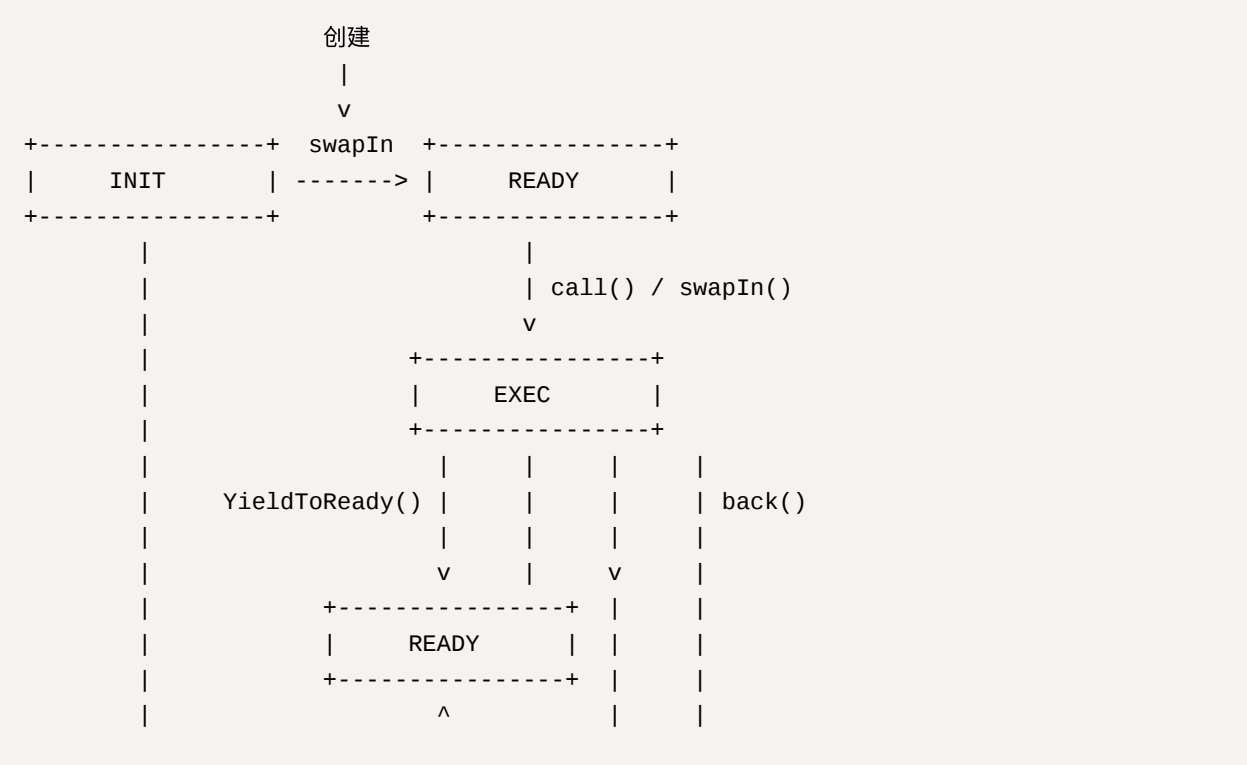
本篇主要是对sylar协程库的分析，包括设计结构、性能分析，以及优化思路。https://github.com/sylar-yin/sylar

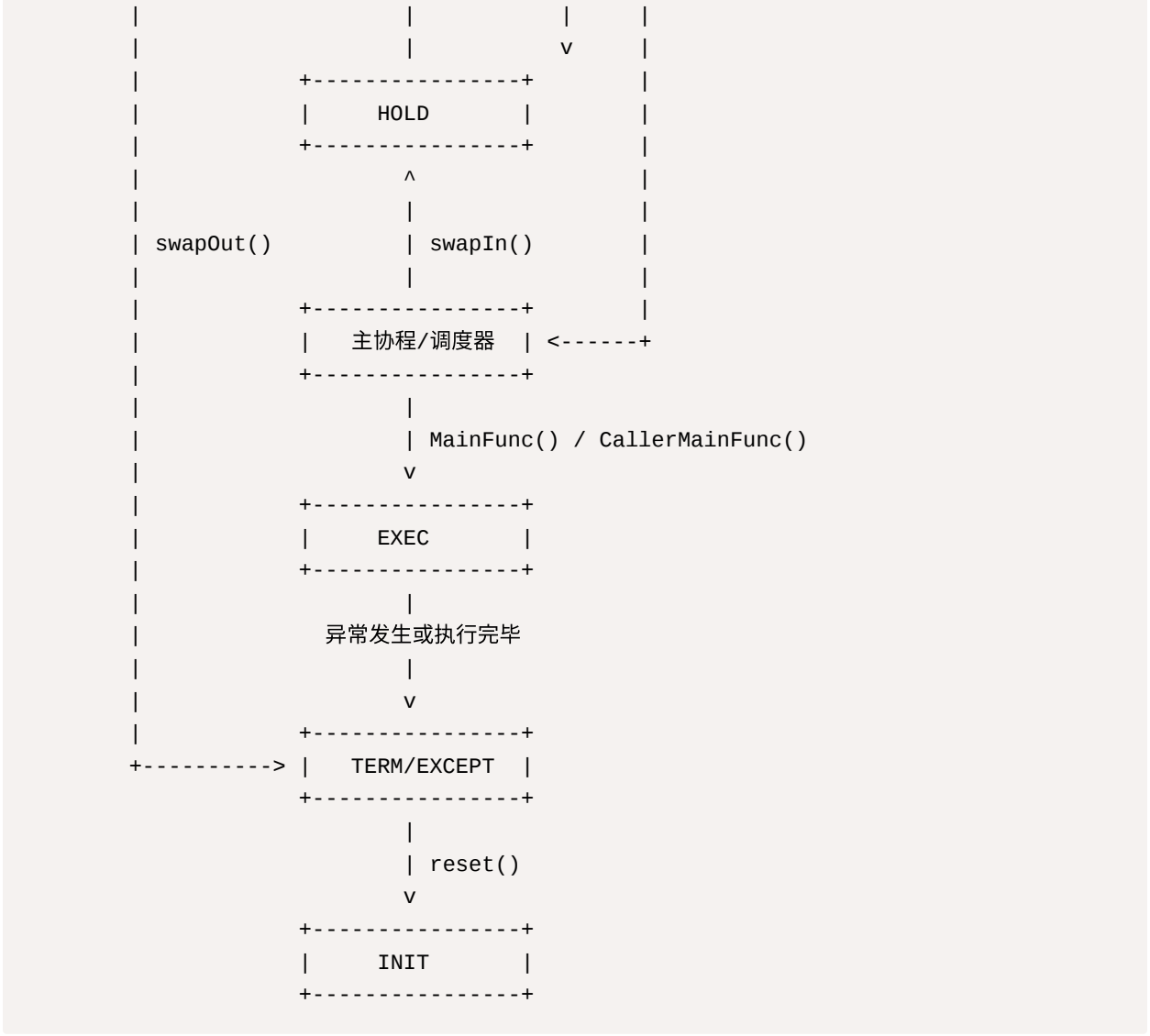
概述

Fiber类是一个基于ucontext实现的M:N对称有栈协程。

- 协程类型：M:N对称有栈协程，也支持在单线程中的N:1模式
- 实现基础：使用ucontext库
- 内存管理：使用智能指针（std::shared_ptr）进行协程对象管理
- 状态管理：包含多个协程状态（INIT, HOLD, EXEC, TERM, READY, EXCEPT）
- 栈管理：支持自定义栈大小

状态流转图





目前存在的问题

个人理解，目前代码存在一些问题。

- 在单线程中使用N:1模式时，可能会有coredump。原因是协程切出时默认回到调度器主协程，而此时其实逻辑上是不需要由调度器接收的。（假设没有初始化化调度器，会coredump）

```
2024-11-12 17:07:06     67657   name_0  0       [ERROR] [root]  /home/leon/workspace/c
swapcontext
backtrace:
    LioNet::Fiber::swapOut()
    LioNet::Fiber::YieldToHold()
    ./bin/test_fiber(+0xa7ff)
    ./bin/test_fiber(+0xd427)
    ./bin/test_fiber(+0xcc0d)
    ./bin/test_fiber(+0xc695)
```

```
    std::function<void ()>::operator()() const
    LioNet::Fiber::CallerMainFunc()
    /lib/x86_64-linux-gnu/libc.so.6(+0x5a130)
```

- 当前代码对于M:N和N:1两种模式的接口设计不清晰。当前通过use_caller参数进行控制，但是会有第一点的问题，同时使用了两组接口来控制协程的切出和切入，通用些不够好。
- 当任务需要分发时，对于用户提交的任务（协程或者函数），其封装逻辑有些冗余，不够简洁。
- 协程的状态设计其物理含义不是很明确。
- 任务池设计的比较简略，锁的开销较大
- 对于内存的使用有待优化

性能测试

这一节从用户层面分析协程的创建和切换开销。

测试环境和方法

- CPU: 64 核 3737.89 MHz
- CPU 缓存:
  - L1 数据缓存: 32 KiB (x32)
  - L1 指令缓存: 32 KiB (x32)
  - L2 统一缓存: 512 KiB (x32)
  - L3 统一缓存: 32768 KiB (x8)
- 操作系统: Linux 6.2.0-34-generic
- 编译器: GCC 11.4.0
- 测试框架: Google Benchmark

测试分为两个主要部分：协程创建 (BM_FiberCreation) 和协程切换 (BM_FiberSwitch)。每项测试都在不同的协程数量 (1000 和 3000) 和线程数 (1, 2, 4, 8, 16) 下进行。

- 协程创建测试: 测量创建指定数量协程所需的时间。
- 协程切换测试: 测量在指定数量的协程间进行 1000 次切换所需的时间。

测试结果

协程创建性能

协程数	线程数	时间 (ns)	创建速率 (items/s)	平均创建时间 (ns/fiber)	标准差 (ns)	加速比
1000	1	3557298	281.112k	3557.30	14.49	1.00
1000	2	3557123	281.126k	3557.12	15.47	1.00
1000	4	3554345	281.346k	3554.35	14.50	1.00
1000	8	3560406	280.866k	3560.41	19.14	1.00
1000	16	3559788	280.915k	3559.79	15.48	1.00

3000	1	11227702	267.203k	3742.57	46.20	1.00
3000	2	11199540	267.869k	3733.18	75.94	1.00
3000	4	11197767	267.911k	3732.59	87.57	1.00
3000	8	11206335	267.707k	3735.45	45.83	1.00
3000	16	11199011	267.881k	3733.00	75.49	1.00

注：标准差和加速比是基于三次测试结果计算得出。

协程切换性能

协程数	线程数	时间 (ns)	切换速率 (items/s)	平均切换时间 (ns/switch)	标准差 (ns)	加速比
1000	1	724088641	220.552M	724.09	5.30	1.00
1000	2	624267007	431.309M	624.27	18.51	1.16
1000	4	938064906	270.899M	938.06	22.80	0.77
1000	8	1069489288	241.442M	1069.49	30.86	0.68
1000	16	1225384502	199.437M	1225.38	31.67	0.59
3000	1	2194733991	220.315M	731.58	12.53	1.00
1000	2	1993875993	410.188M	664.63	36.89	1.10
3000	4	2985946164	259.838M	995.32	50.85	0.73
3000	8	3503675863	226.229M	1167.89	98.09	0.63
3000	16	3754937239	192.774M	1251.65	98.25	0.58

注：标准差和加速比是基于三次测试结果计算得出。

分析

协程创建性能

1. **一致性**: 创建性能在不同线程数下保持高度一致，标准差较小，表明创建过程是线程安全且高效的。
2. **可扩展性**: 从 1000 到 3000 个协程，平均创建时间仅增加约 5%，显示出良好的可扩展性。
3. **效率**: 平均每秒可创建约 267,000 到 281,000 个协程，性能表现优秀。
4. **线程影响**: 加速比接近 1.00，说明线程数对创建性能几乎没有影响。

协程切换性能

1. **线程敏感性**: 切换性能对线程数量高度敏感，2 线程配置在所有测试中表现最佳。
2. **性能峰值**: 2 线程配置下，1000 个协程可达到 431.309M 次/秒的切换率，加速比为 1.16。
3. **性能下降**: 随着线程数增加，切换性能和加速比显著下降，16 线程时加速比降至约 0.59。
4. **协程数量影响**: 3000 个协程的切换性能略低于 1000 个，但差异不大，表明良好的可扩展性。
5. **标准差**: 切换性能的标准差随线程数增加而增大，表明高线程数下性能的不稳定性增加。

性能瓶颈分析

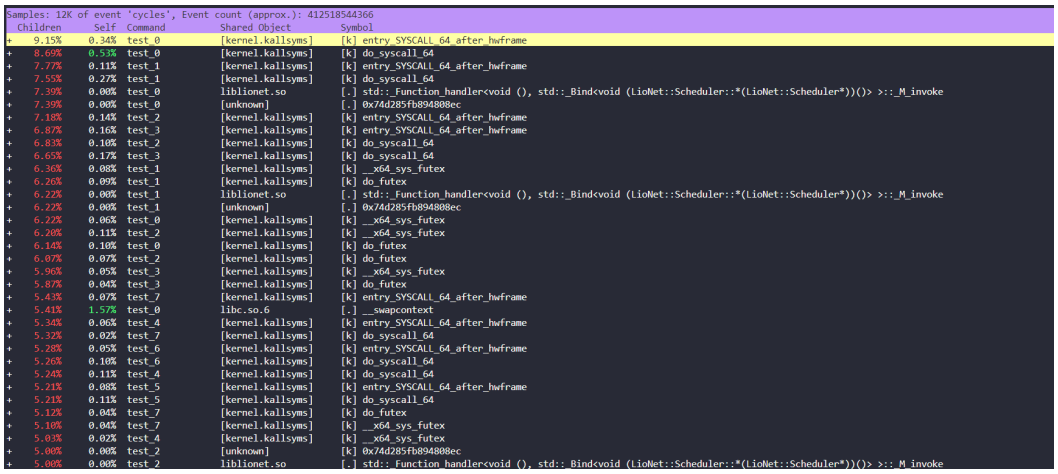
接下里使用 perf 工具进行分析，重点关注 CPU 时间分布和函数调用开销，识别主要性能瓶颈。

（其实也应该做一下内存使用分析，但是由于目前对tcmalloc和jemalloc的理解不够，同时内存优化需要结合调度模式来看，现在暂时没有好的想法，所以暂时没有做）

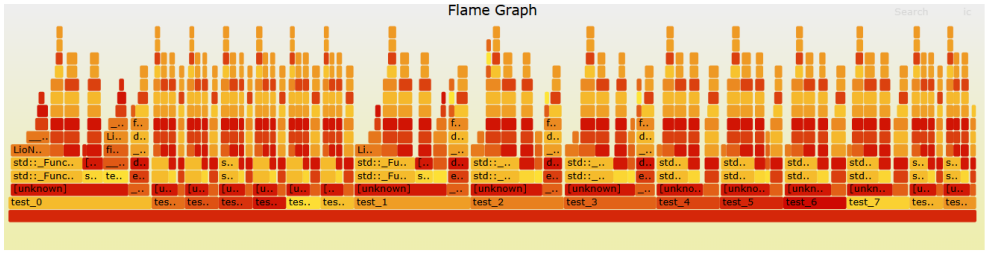
这里也记录一下相关命令。

```
# 需要开启内核的权限
# 运行程序，收集数据
perf record -F 99 -g -o perf_analysis/perf_sched.data ./bin/test_fiber_sched

# 展示数据
perf report -i perf_analysis/perf_sched.data
```



热点函数及耗时



耗时火焰图

```
# 相关命令
perf record -F 99 -g -o perf.data your_program_command
perf script -i perf.data > perf.unfold
stackcollapse-perf.pl perf.unfold > perf.folded
flamegraph.pl --minwidth 5 perf.folded > flamegraph.svg
```

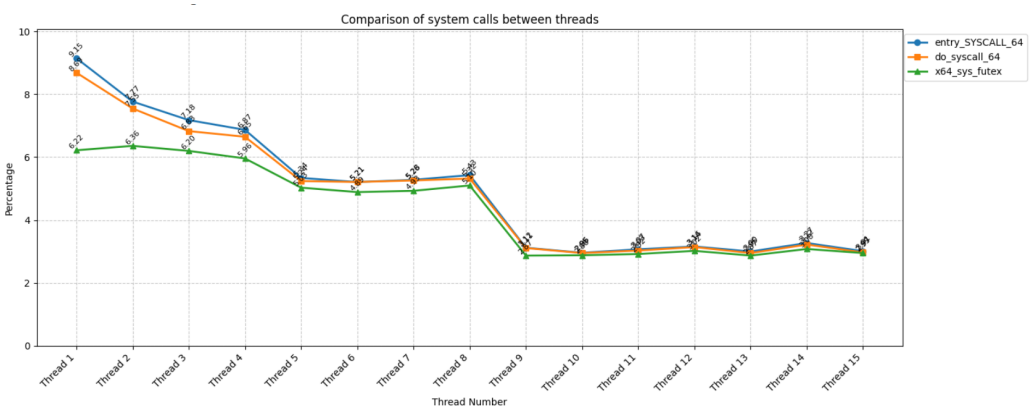
可视化分析

以下饼图展示了主要函数的 CPU 时间占用比例

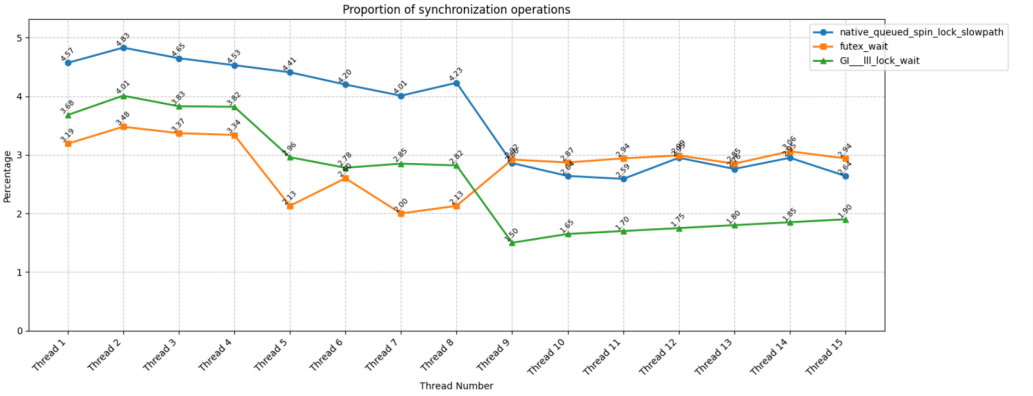
```
pie title CPU 时间分布
    "系统调用" : 35.97
    "同步操作" : 30.25
    "调度器操作" : 18.61
    "上下文切换" : 8.50
    "自旋锁操作" : 5.52
    "其他" : 1.15
```

线程间系统调用对比

以下折线比较了不同线程中主要系统调用的占用时间：



同步操作分析



详细分析

系统调用开销

1. `entry_SYSCALL_64_after_hwframe` 和 `do_syscall_64` 在所有线程中都占据了显著比例（约 15-17%），表明频繁的系统调用。
2. `__x64_sys_futex` 在大多数线程中占 5-6% 的 CPU 时间，表明大量的 futex 操作。

同步操作

1. `native_queued_spin_lock_slowpath` 在所有线程中都有较高占用（2.5-4.8%），表明存在严重的锁竞争。

2. `futex_wait` 和 `__GI___lll_lock_wait` 也占用了显著的 CPU 时间，进一步证实了同步开销大的问题。

上下文切换

1. `__swapcontext` 在 test_0 和 test_1 线程中分别占用了 5.41% 和 3.09% 的 CPU 时间，表明协程切换的开销较大。

调度器性能

`LioNet::Scheduler::run` 在 test_0 和 test_1 线程中分别占用了 4.15% 和 2.86% 的 CPU 时间，表明调度器可能存在优化空间。

自旋锁操作

1. `_raw_spin_lock` 在多个线程中出现，占用了 2-3% 的 CPU 时间，表明存在大量的自旋等待。

负载不均衡

1. 从耗时图可以明显看出线程之间负载并不均衡

性能瓶颈分析

1. **系统调用开销**: 频繁的系统调用，特别是 futex 操作，占用了大量 CPU 时间。
2. **同步开销**: 自旋锁、互斥锁和 futex 等同步机制的使用导致了显著的性能开销。
3. **上下文切换**: 协程间的切换操作仍然有较高的开销。
4. **调度器效率**: 调度器的运行占用了相当比例的 CPU 时间，存在优化空间。

这里的测试结果和前面对代码的分析是一致的。

优化计划

优化同步机制

- 实现更细粒度的锁策略，减少锁竞争。
- 使用无锁数据结构替代部分互斥锁。
- 优化自旋锁的使用，考虑使用自适应自旋策略。

改进协程切换

- 优化上下文保存和恢复的实现。
- 考虑使用汇编级别的优化来加速上下文切换。
- 实现协程亲和性，减少跨核心切换。

优化调度器

- 重构调度器的核心逻辑，减少每次调度的开销。
- 实现多任务队列和工作窃取，提高负载均衡效率。

内存优化

- 实现协程栈池，减少动态内存分配的开销。
- 使用内存预分配策略，减少运行时的内存管理操作。

- 结合协程的调度策略，使用更高级的内存分配技术