golang调度机制解析

杨宏波

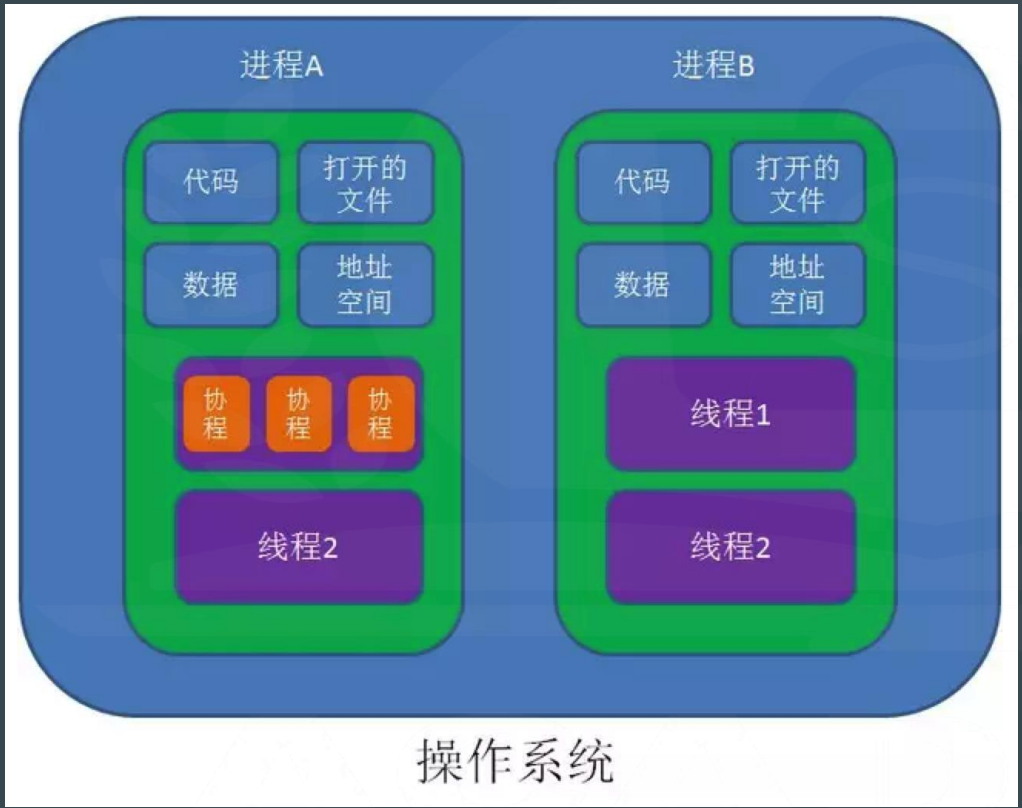
前言

并发与并行

并发：逻辑上具有处理多个同时性任务的能力。

并行：物理上同一时刻执行多个并发任务

进程、线程、协程

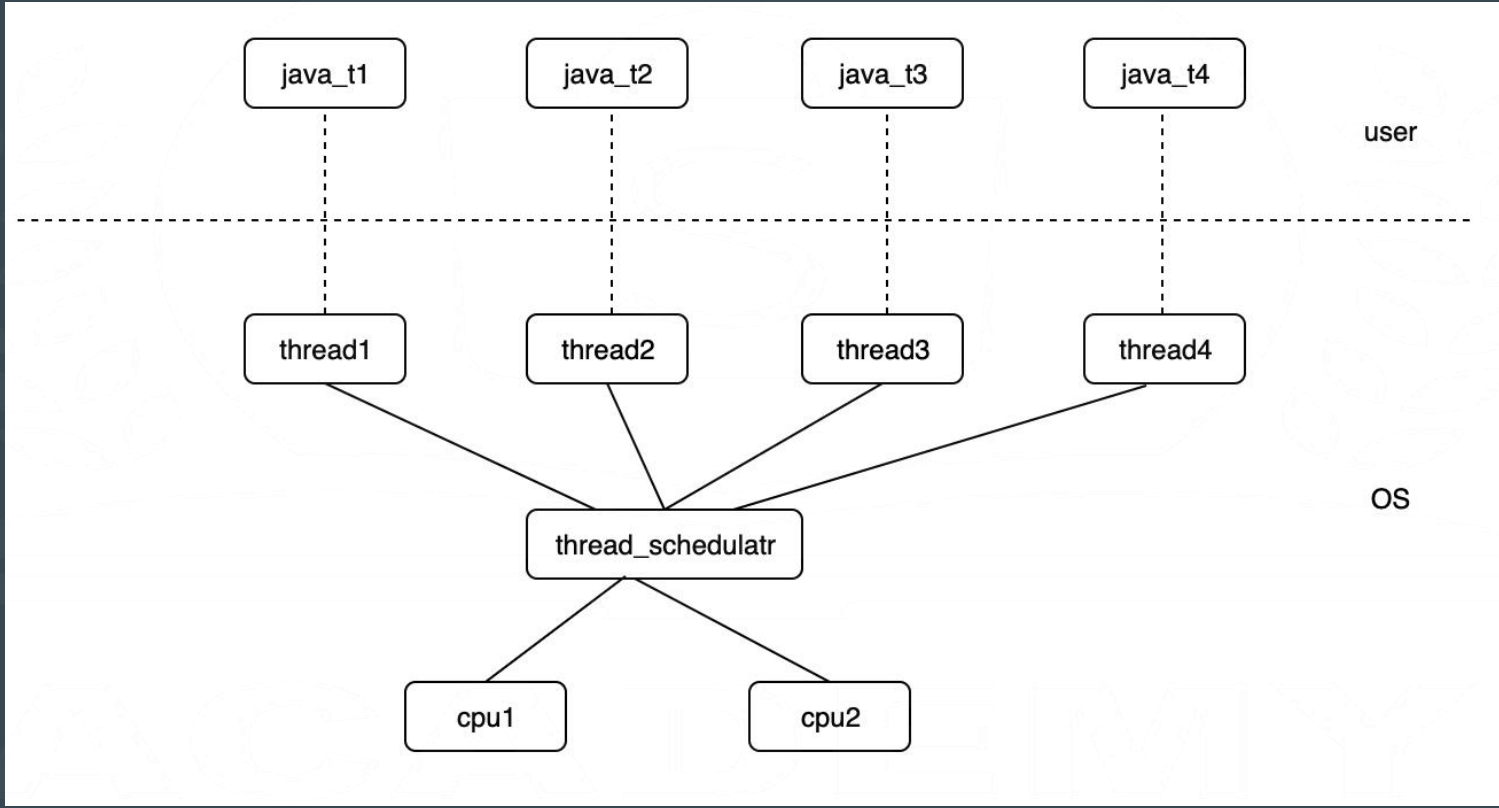


进程：系统资源分配的最小单元

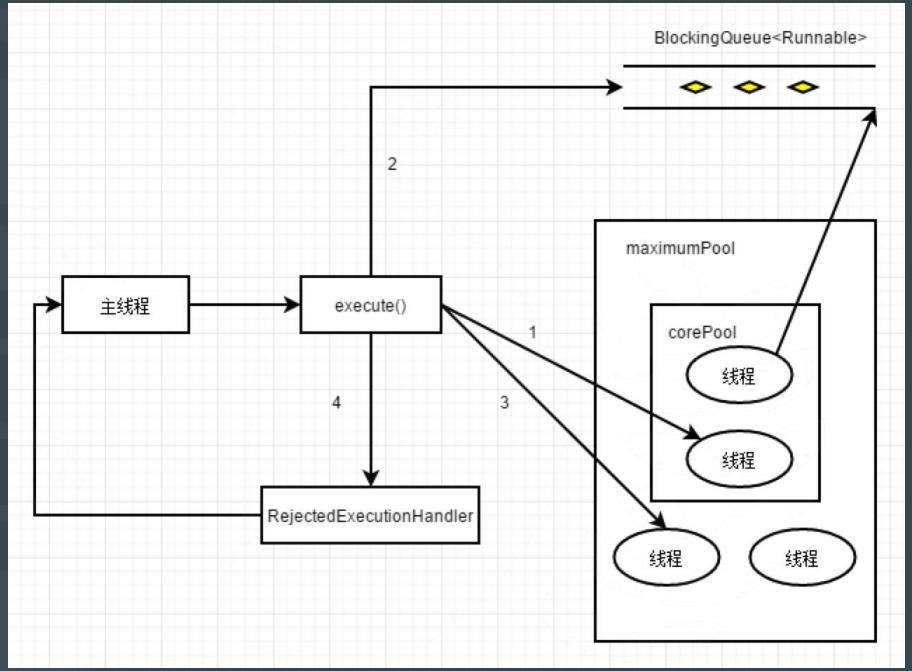
线程：系统调度执行的最小单元

协程：线程的线程

java的线程模型


java_t1
java_t2
java_t3
java_t4
user
thread1
thread2
thread3
thread4
OS
thread_schedulatr
cpu1
cpu2

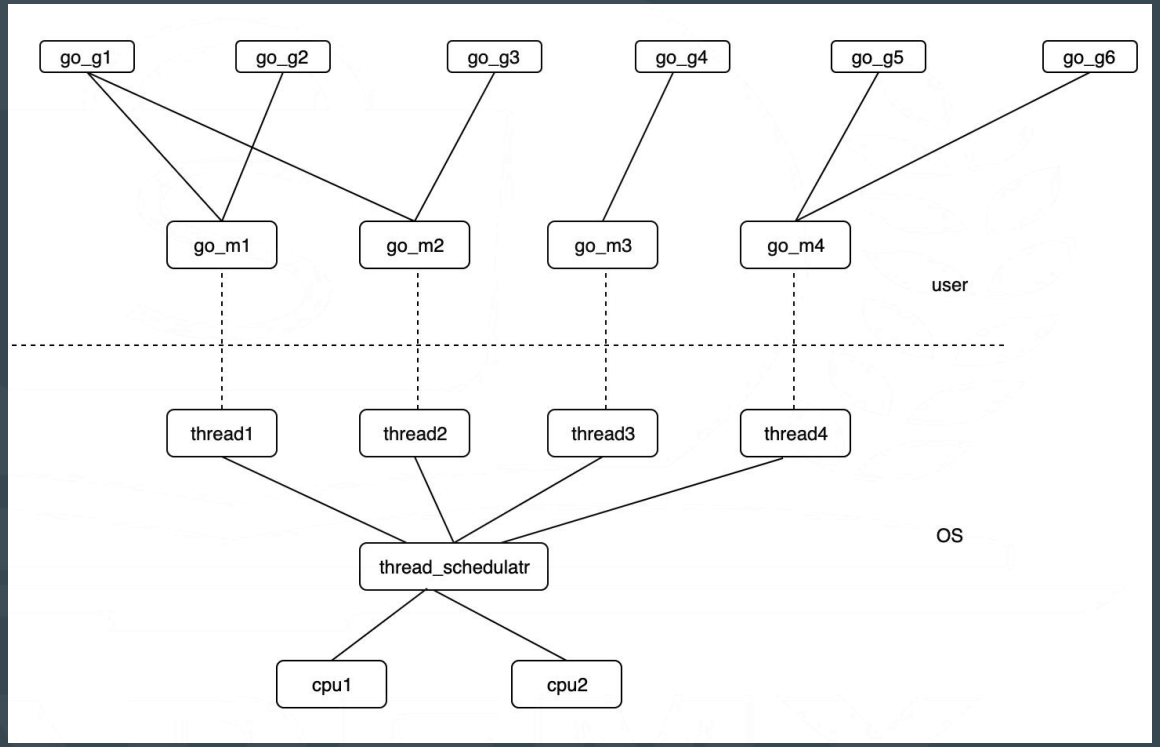
```java
public class Test {
    public static void main(String[] args) throws Exception {
        Task task1 = new Task(1);
        Task task2 = new Task(2);

//        ExecutorService normalExecutor = new ThreadPoolExecutor(2, 4, 200, TimeUnit.MILLISECONDS,
//                new ArrayBlockingQueue<Runnable>(5));
//        ExecutorService singleExecutor = Executors.newSingleThreadExecutor();
//        ExecutorService cachedExecutor = Executors.newCachedThreadPool();
        //创建线程池服务
        ExecutorService executor = Executors.newFixedThreadPool(2);
        //将任务交给线程池执行
        executor.execute(task1);
        executor.execute(task2);
        executor.shutdown();
    }
}

//可以提交给线程池执行的任务类，线程池执行任务时会执行其中的run方法
class Task implements Runnable {
    private int taskNum;

    public Task(int num) {
        this.taskNum = num;
    }

    public void run() {
        System.out.println("开始执行任务: " + taskNum);
        try {
            Thread.currentThread().sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("结束执行任务: " + taskNum);
    }
}
```



java线程池使用范式及其原理

golang的多线程范式

```go
func main() {
    for i := 0; i < 100; i++ {
        go worker()
    }
}

func worker()  {
    fmt.Println( a...: "do some work")
}
```



go的GMP调度模型

```
type g struct {
        stack       stack   //goroutine的栈，可动态调整大小
        stackguard0         uintptr //goroutine栈的阈值

        m           *m     // 当前g关联的m
    // goroutine切换时，用于保存g的上下文
        sched       gobuf
        atomicstatus   uint32 //保存g的当前状态
        goid        int64  // goroutine的ID
}
```

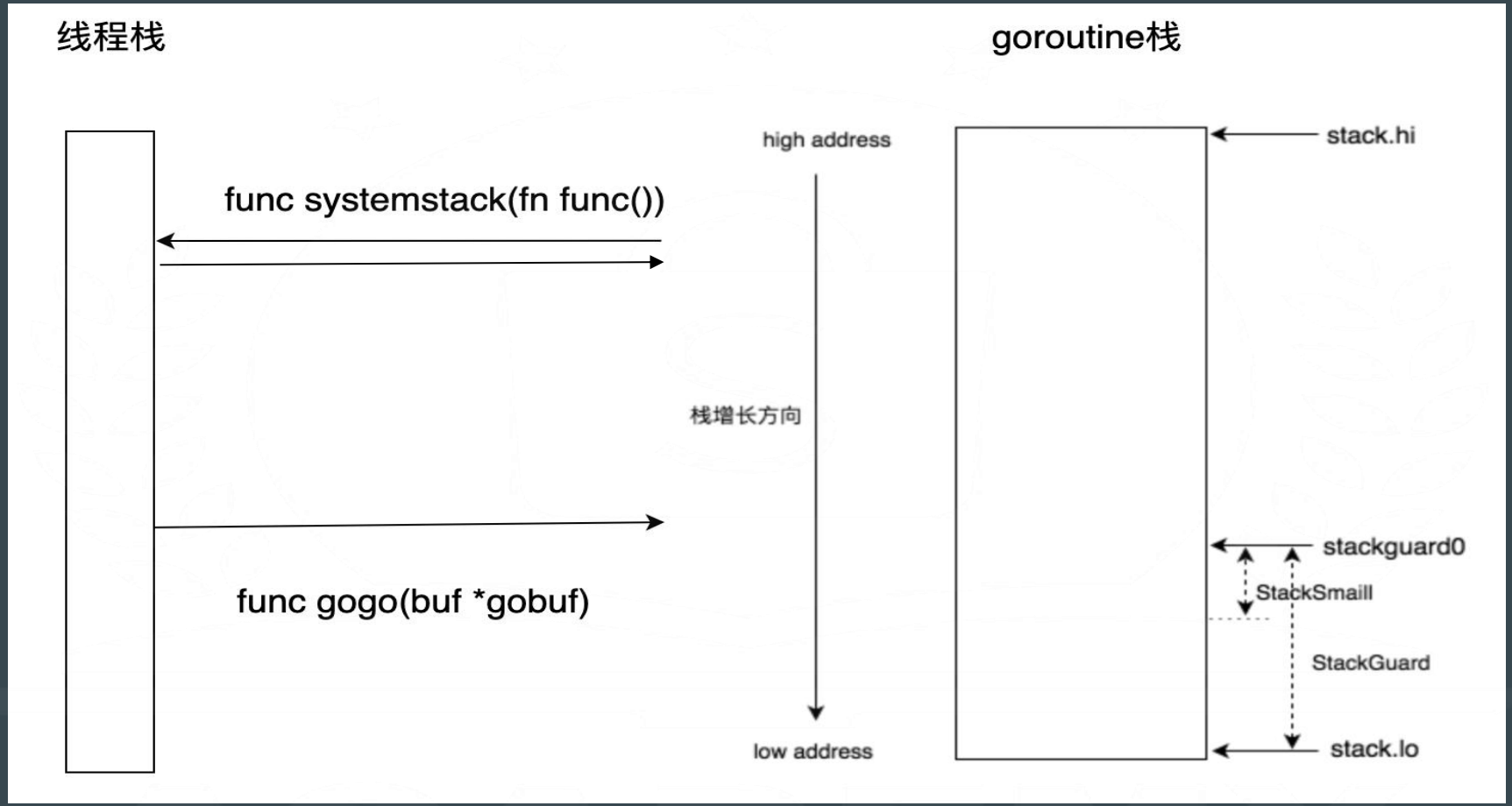
协程栈

```
type m struct {
        g0      *g     //提供系统栈

        tls         [6]uintptr // thread-local storage
        mstartfn    func()          //线程启动函数
        curg        *g       // 当前运行的goroutine
        p           puintptr  // 当前m关联的p
        //缓存一个p，优先从该处获取关联的
    nextp       puintptr p

        spinning    bool // m是否自旋

}
```

线程栈

g : 代表一个goroutine对象

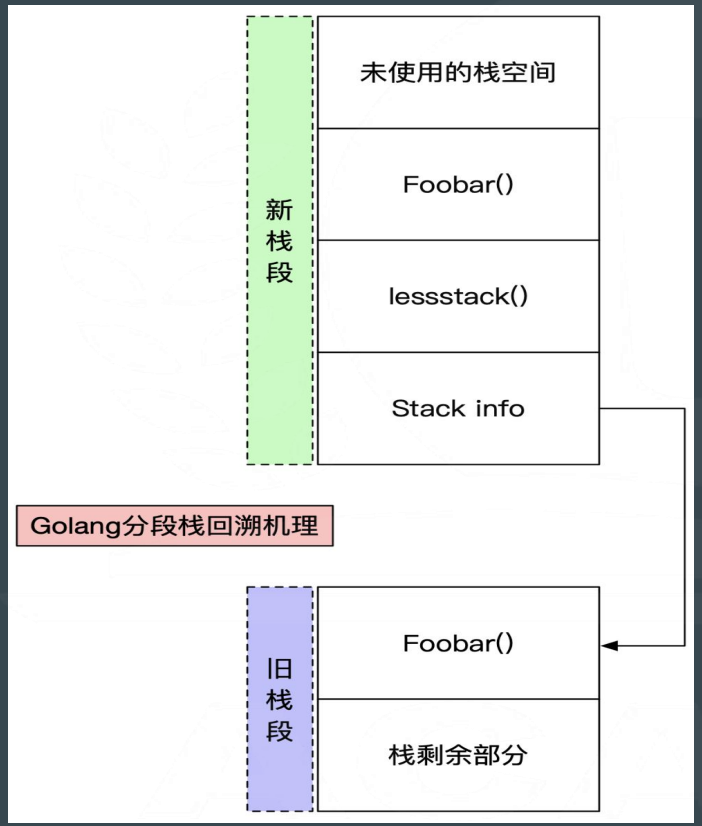
m : 代表一个线程，每一个新建的m会和一个底层的os线程对应



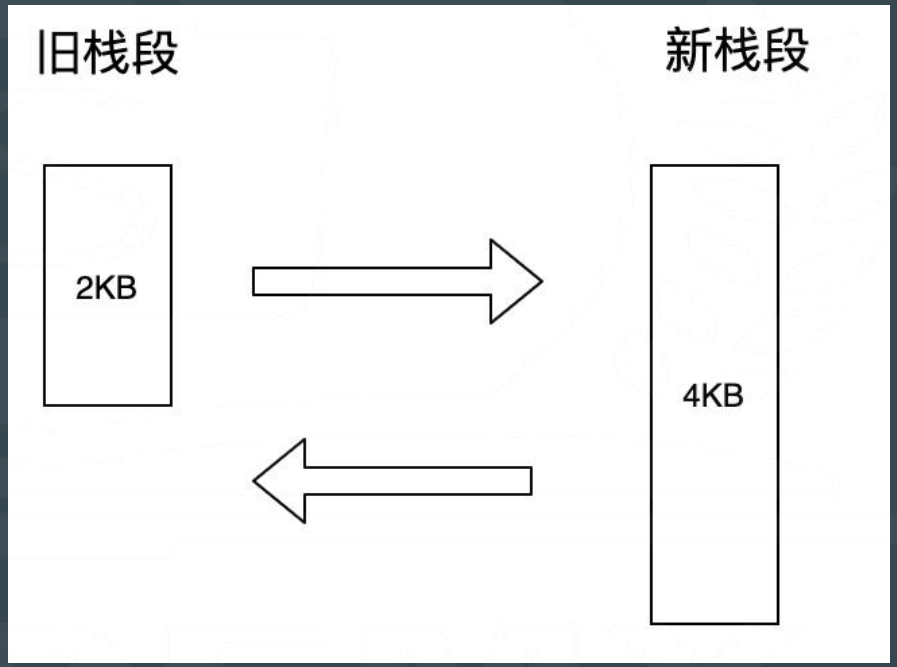
线程栈和goroutine的栈

goroutine栈的扩容和收缩

早期版本：分段栈



现在版本：连续栈

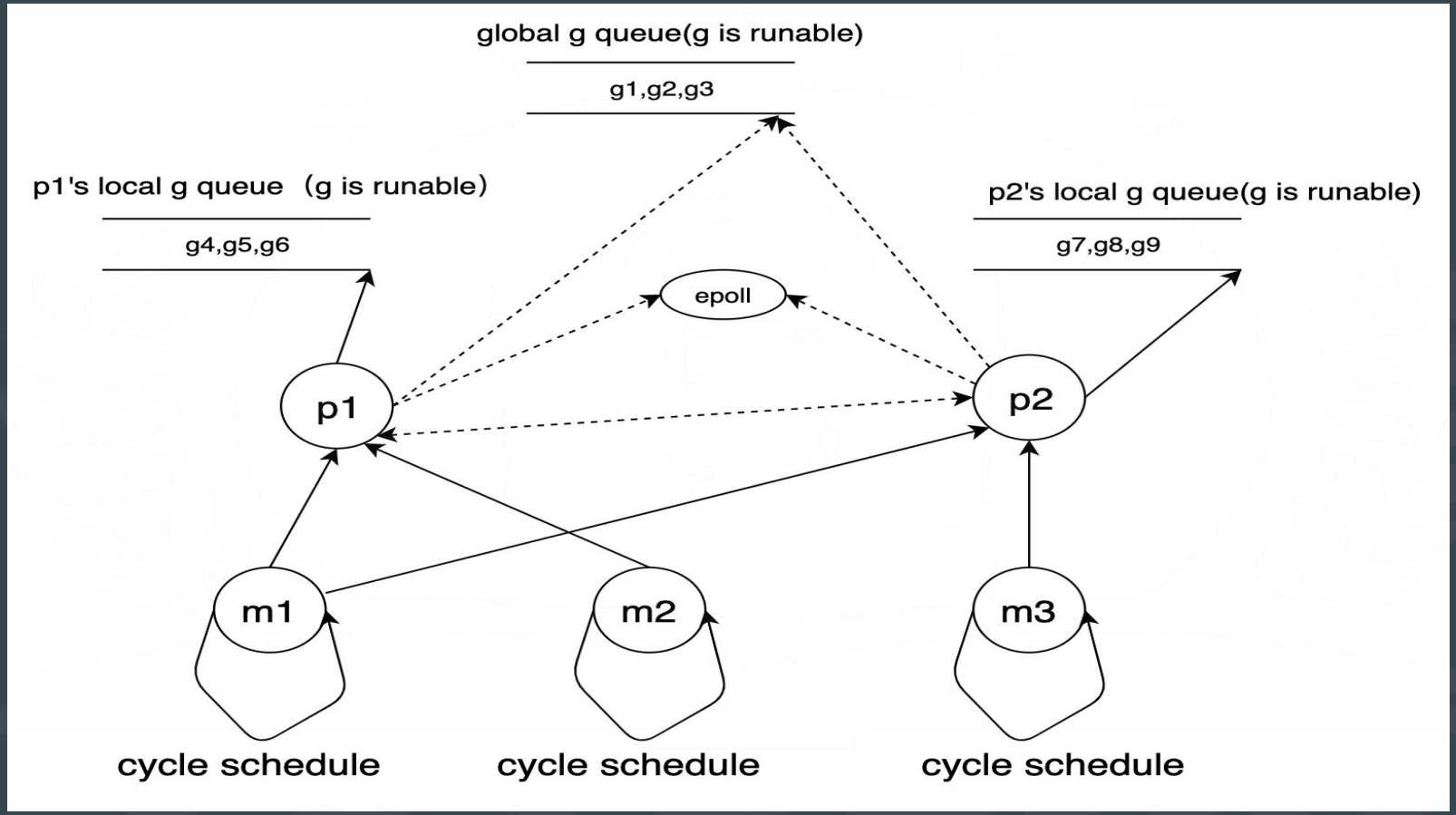


```
type p struct {
        id          int32
        status      uint32 // p的当前状态
        schedtick   uint32 // 调度次数计数器
        syscalltick uint32 // 系统调用次数计数器
        m           muintptr // 回链到关联的m

    // goroutine的ID的缓存
        goidcache    uint64
        goidcacheend uint64

        // 可运行的goroutine的队列
        runqhead uint32
        runqtail uint32
        runq     [256]guintptr

        runnext guintptr // 下一个运行的g
}
```

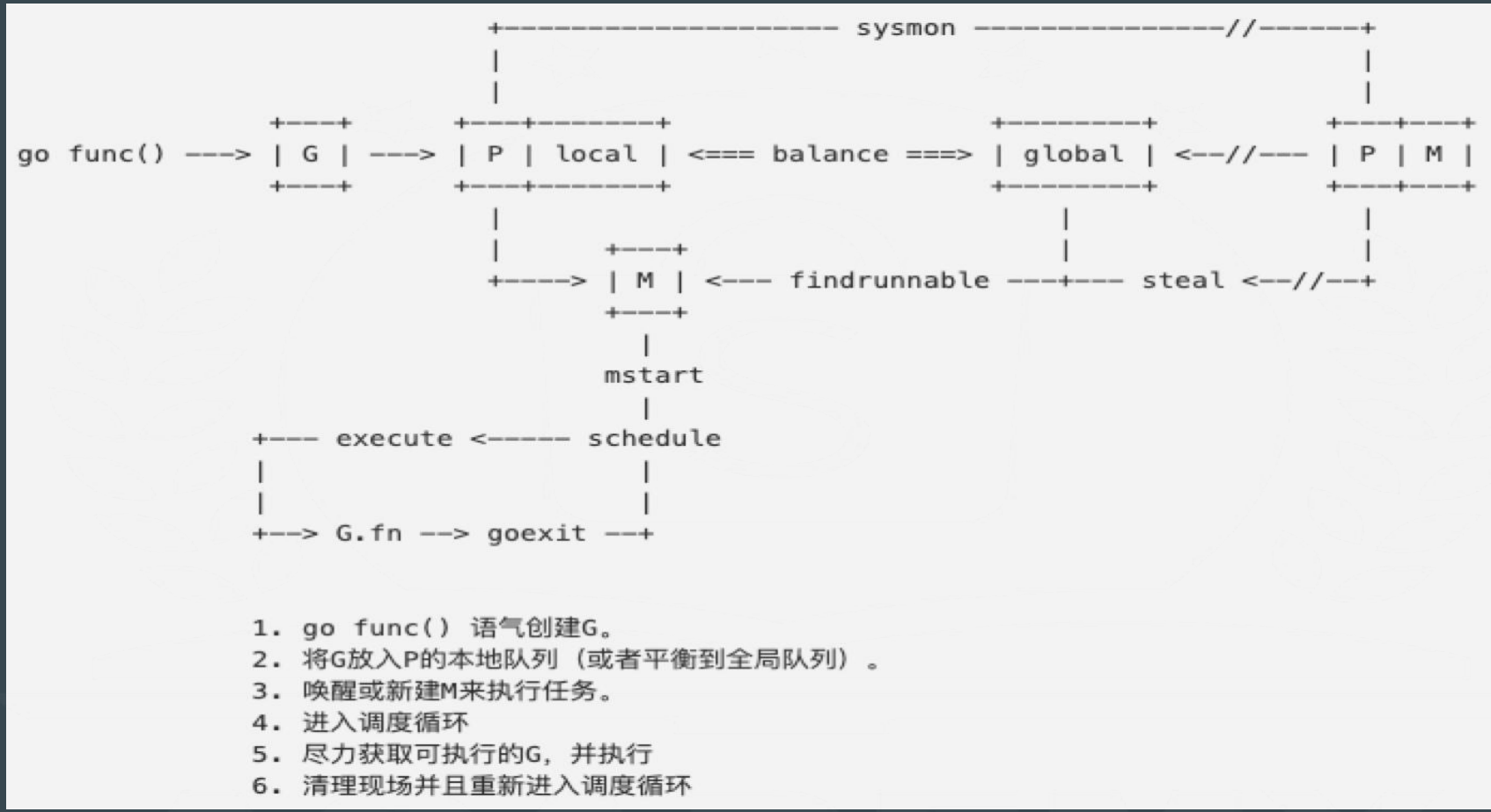
p : m和g之间的连接器，逻辑上映射到一个CPU。决定了go的实际并行能力

```
type schedt struct {
        midle muintptr          // 空闲m的链表
        nmidle int32            // 空闲m的数量
        mcount int32            //已创建m的总数

        pidle      puintptr // 空闲p的链表
    npidle     uint32 // 空闲p的数量


    // Global runnable queue.
    runq     gQueue // 全局g队列
    runqsize int32 //全局g队列长度


    // g的空闲池 (status == Gdead)
    gFree struct {
                        gList
                        n int32
    }
```

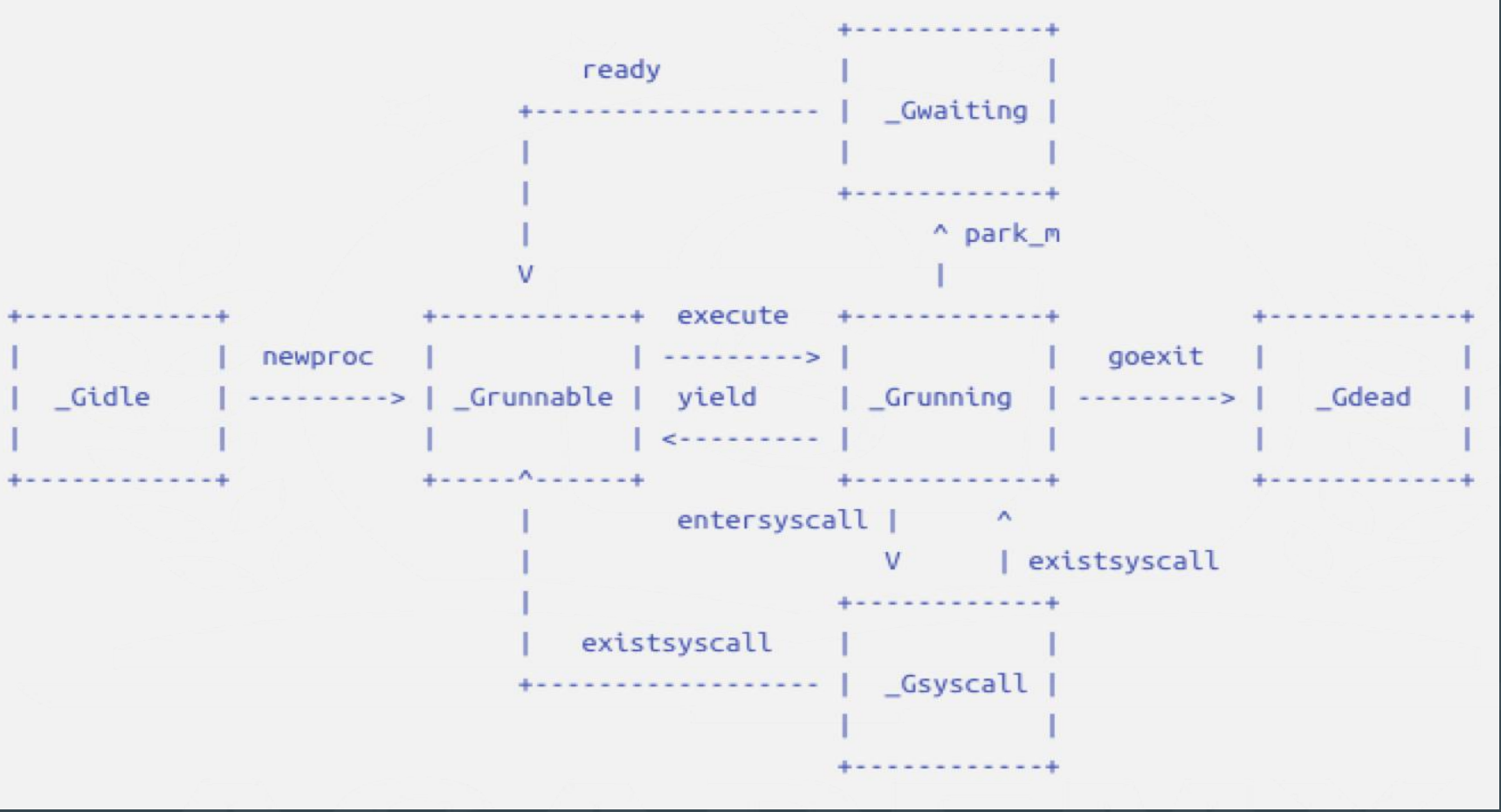
schedt : 维护着全局的gmp资源


global g queue(g is runable)
g1,g2,g3
p1's local g queue （g is runable）
g4,g5,g6
p2's local g queue(g is runable)
g7,g8,g9
epoll
p1
p2
m1
m2
m3
cycle schedule
cycle schedule
cycle schedule

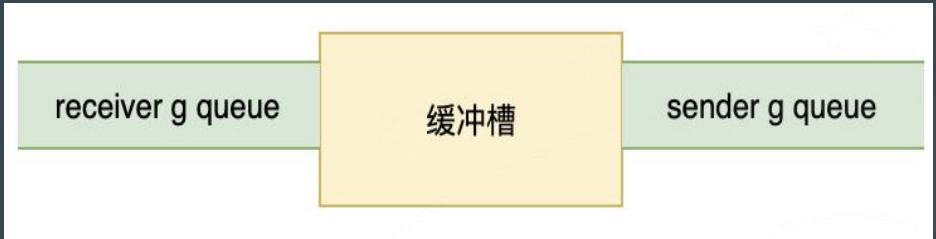
GMP模型结构图



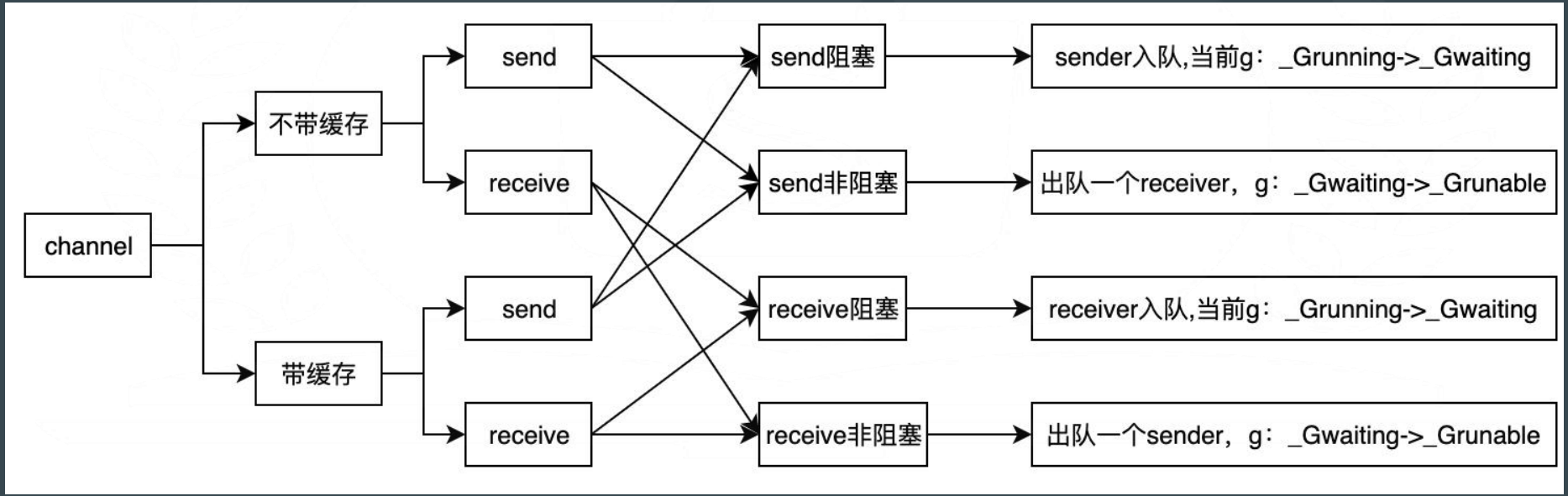
go func() 的执行流程


ready
_Gwaiting
park_m
_Gidle
newproc
_Grunnable
execute
yield
_Grunning
goexit
_Gdead
entersyscall
existsyscall
existsyscall
_Gsyscall

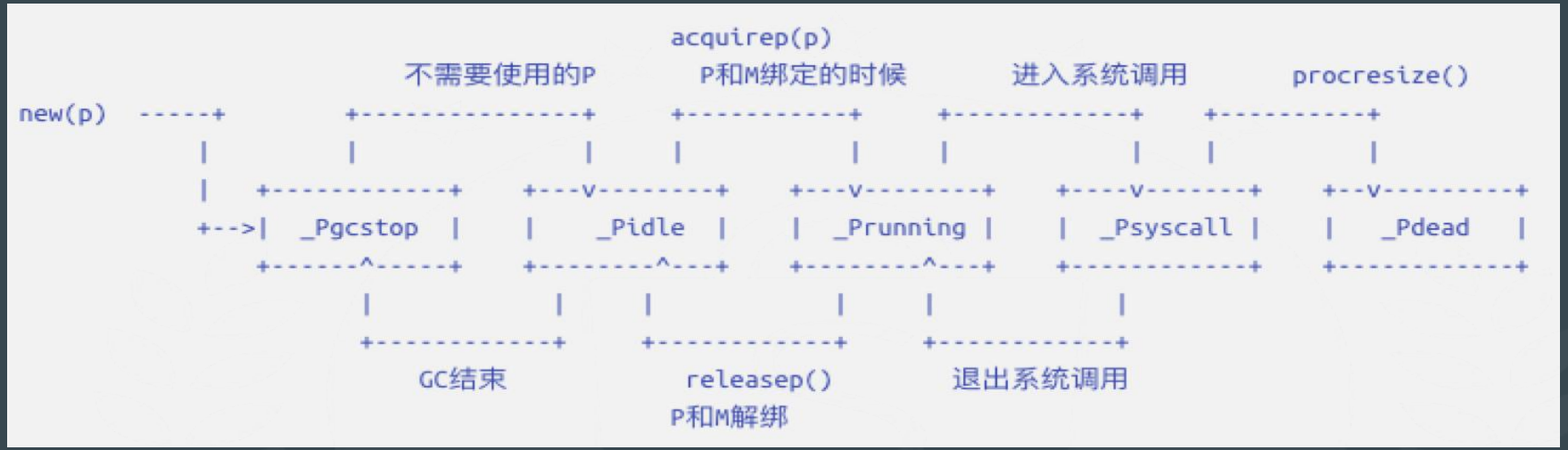
g的状态流转图



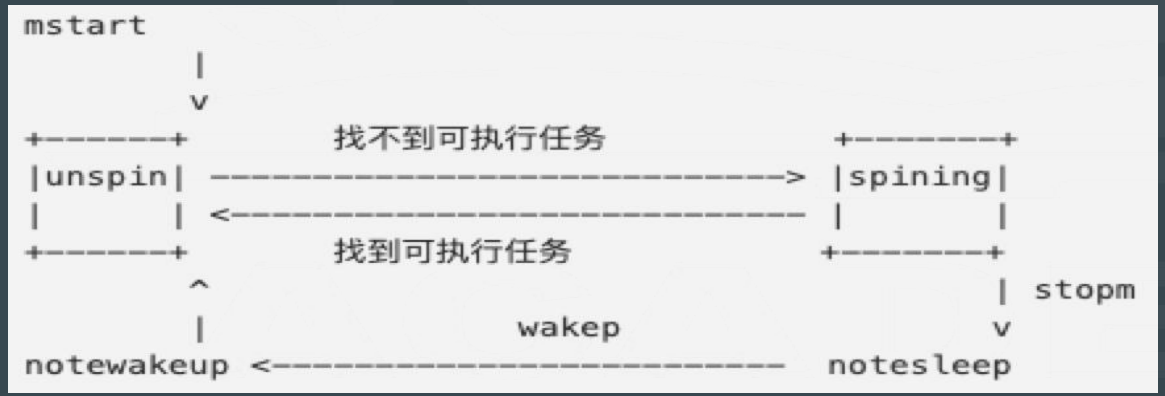
channel结构图



channel实现原理中的g状态变化



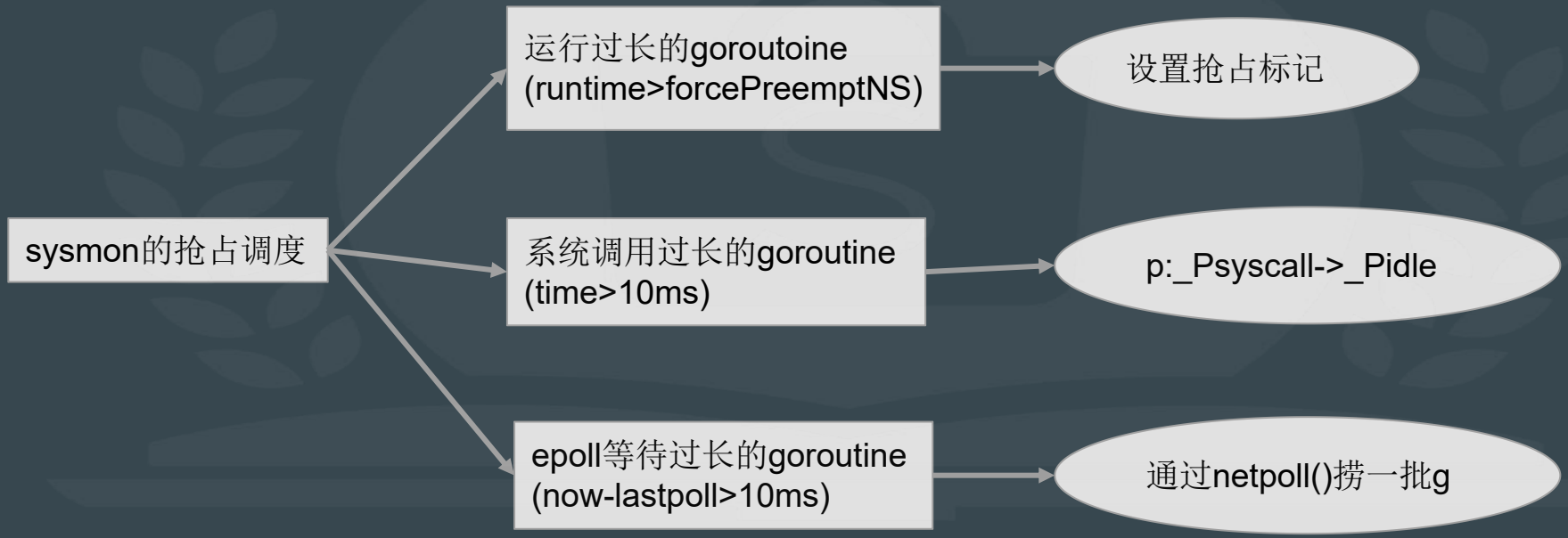
p的状态流转图



m的状态流转图

go调度中的抢占

sysmon线程的抢占式调度



注：sysmon是一个系统监控函数，运行在一个独立的线程上，会定期进行gc和抢占retake的动作

```
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err syscall.Errno)
```

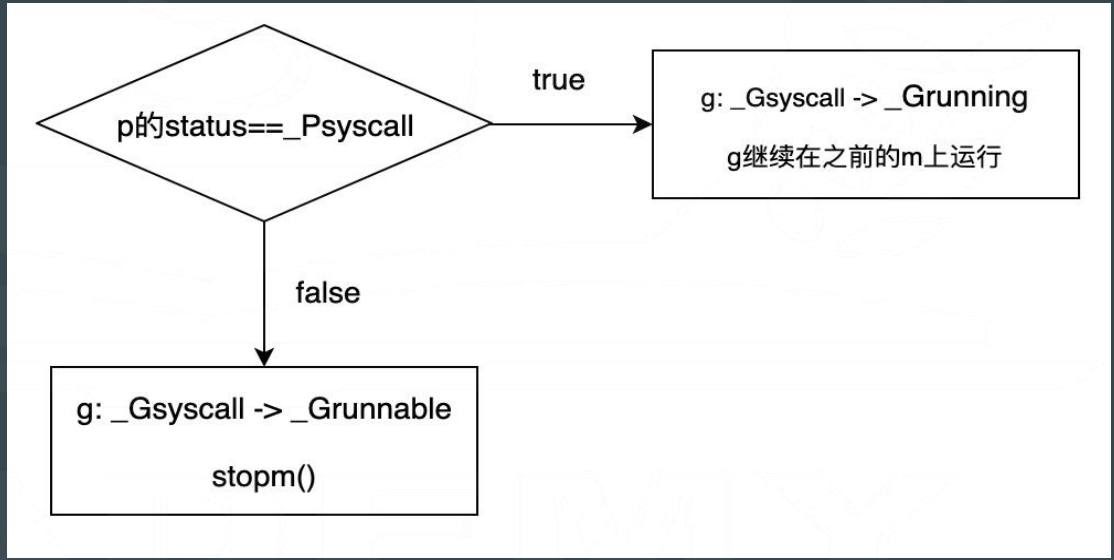
```
//Syscall
TEXT ·Syscall(SB),NOSPLIT,$0-56
    CALL    runtime·entersyscall(SB)
    MOVQ    a1+8(FP), DI
    MOVQ    a2+16(FP), SI
    MOVQ    a3+24(FP), DX
    MOVQ    $0, R10
    MOVQ    $0, R8
    MOVQ    $0, R9
    MOVQ    trap+0(FP), AX  // syscall entry
    SYSCALL
    CMPQ    AX, $0xfffffffffffff001
    JLS ok
    MOVQ    $-1, r1+32(FP)
    MOVQ    $0, r2+40(FP)
    NEGQ    AX
    MOVQ    AX, err+48(FP)
    CALL    runtime·exitsyscall(SB)
    RET
ok:
    MOVQ    AX, r1+32(FP)
    MOVQ    DX, r2+40(FP)
    MOVQ    $0, err+48(FP)
    CALL    runtime·exitsyscall(SB)
    RET
```

进入系统调用前执行的go函数

完成系统调用后执行的go函数

runtime.entersyscall():
g : _Grunning -> _Gsyscall
p : _Prunning -> _Psyscall

runtime.exitsyscall():



go中的系统调用过程

go实现的异步非阻塞模型（基于对epoll的封装）

```go
func main() {
    ln, err := net.Listen( network: "tcp",  address: ":8080")
    if err != nil {
        panic(err)
    }
    for {
        conn, err := ln.Accept()
        if err != nil {
            panic(err)
        }
        // 每个Client一个Goroutine
        go handleConnection(conn)
    }
}
func handleConnection(conn net.Conn) {
    defer conn.Close()
    var body [4]byte
    for {
        // 读取客户端消息
        _, err := conn.Read(body[:])
        if err != nil {
            break
        }
        fmt.Printf( format: "收到消息: %s\n", string(body[:]))
    }
}
```

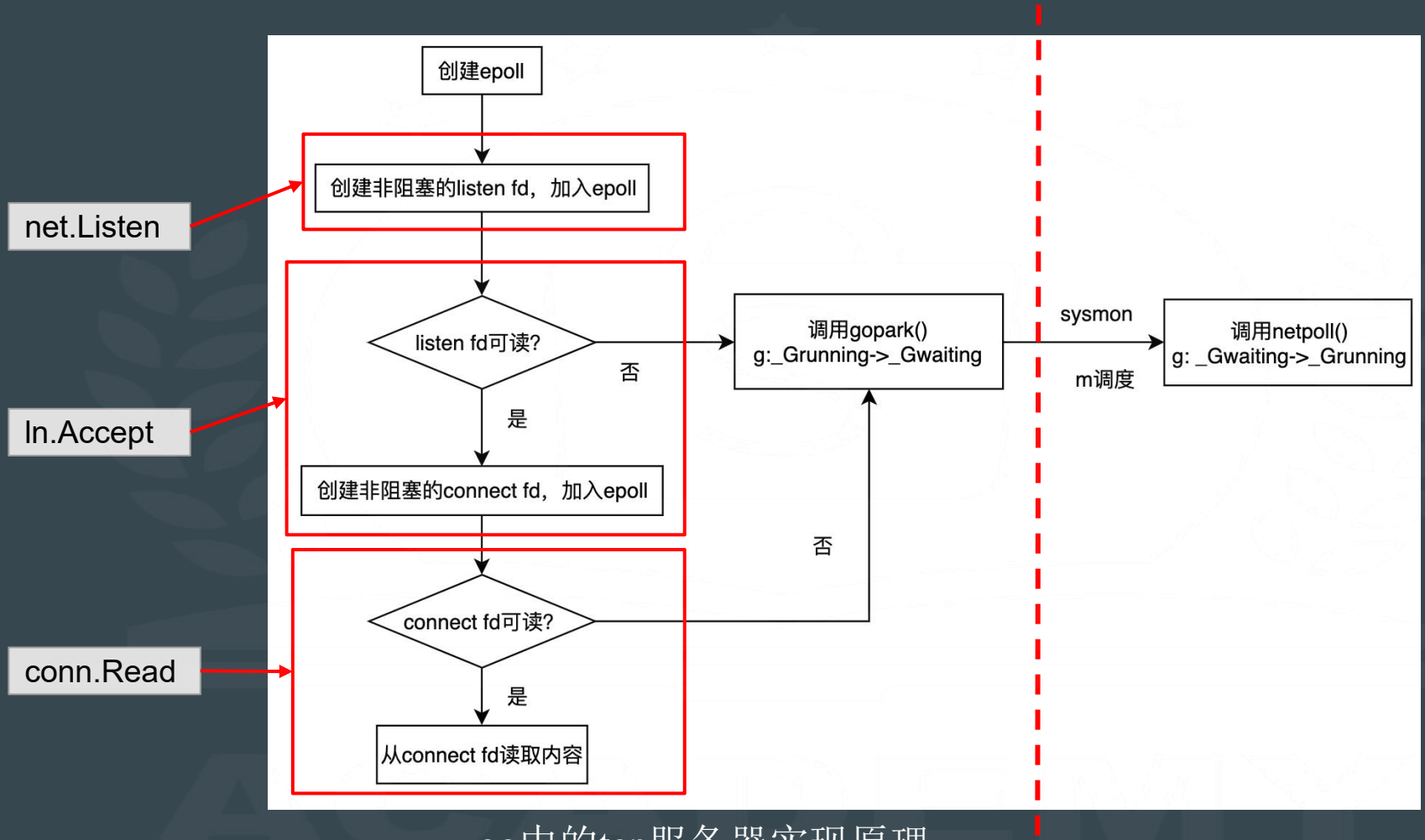
创建监听socket

创建连接socket

会阻塞吗？

在连接socket上读取字符串

go中的tcp服务器实现范式



go中的tcp服务器实现原理

总结

go的调度模型特点

1.高并发支持。通过轻量级goroutine（初始栈2KB，可动态伸缩）和go关键字提供简易的并发模型。

2.高并行支持。通过GMP的调度机制和独立的抢占式调度，保证了对CPU资源的充分利用。

核心源码

主要源文件：
runtime/asm_amd64.s：进程启动时调用的一些汇编函数，可以理解为进程的入口。要了解go的启动过程（如init函数初始化、main包中的main函数调用等）可以从这里看起
runtime/runtime2.go：包括调度模型的核心数据结构（g、m、p）的定义
runtime/proc.go：调度器逻辑代码的实现
核心函数：
runtime/proc.go的schedinit：调度系统的初始化，所有的p在这里进行分配
runtime/proc.go的mstart：调度器的启动，这里会启动一个m开始调度goroutine
runtime/proc.go的newproc：使用go关键字时调用的函数，里面会生成一个g代表一个goroutine
runtime/proc.go的sysmon：go的监控线程执行的核心函数，调度中的抢占行为在此实现

参考

https://www.cnblogs.com/sunsky303/p/9705727.html
https://reading.developerlearning.cn/reading/12-2018-08-02-goroutine-gpm/
https://zboya.github.io/post/go_scheduler/
https://blog.csdn.net/u010853261/article/details/88312904
https://jiajunhuang.com/articles/2018_03_29-goroutine_schedule.md.html

thanks