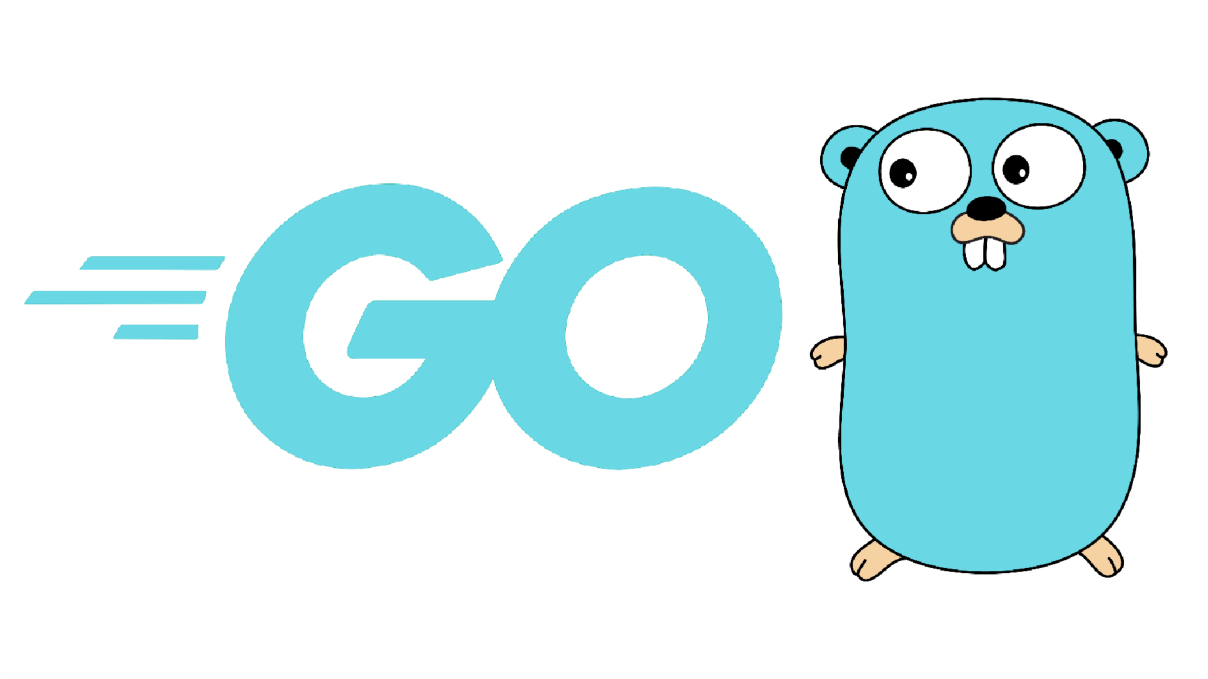
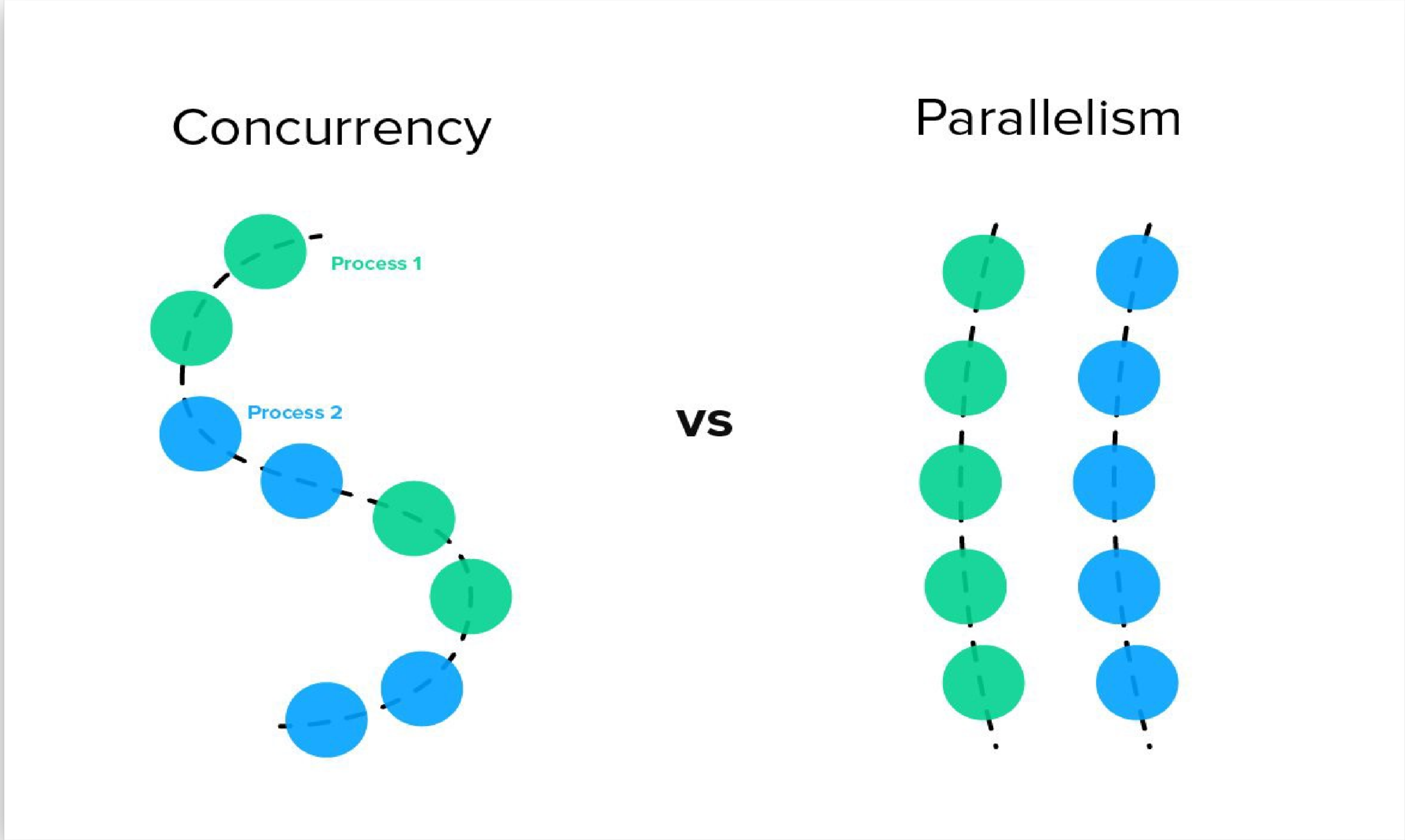
Concurrency in Go



并发的产生?

CPU 计算 vs I/O：

Operation	time
2.5GHz的CPU一个时钟周期	0.4ns
1Gbps的网络传输2KB数据时间	20μs
SSD随机读取时间	150μs
磁盘寻道+旋转时间	15ms


Concurrency
Process 1
Process 2
vs
Parallelism

并发的载体

Process

独立的地址空间，内核态上下文切换

Thread

内核态上下文切换，切换开销小一些

Coroutine

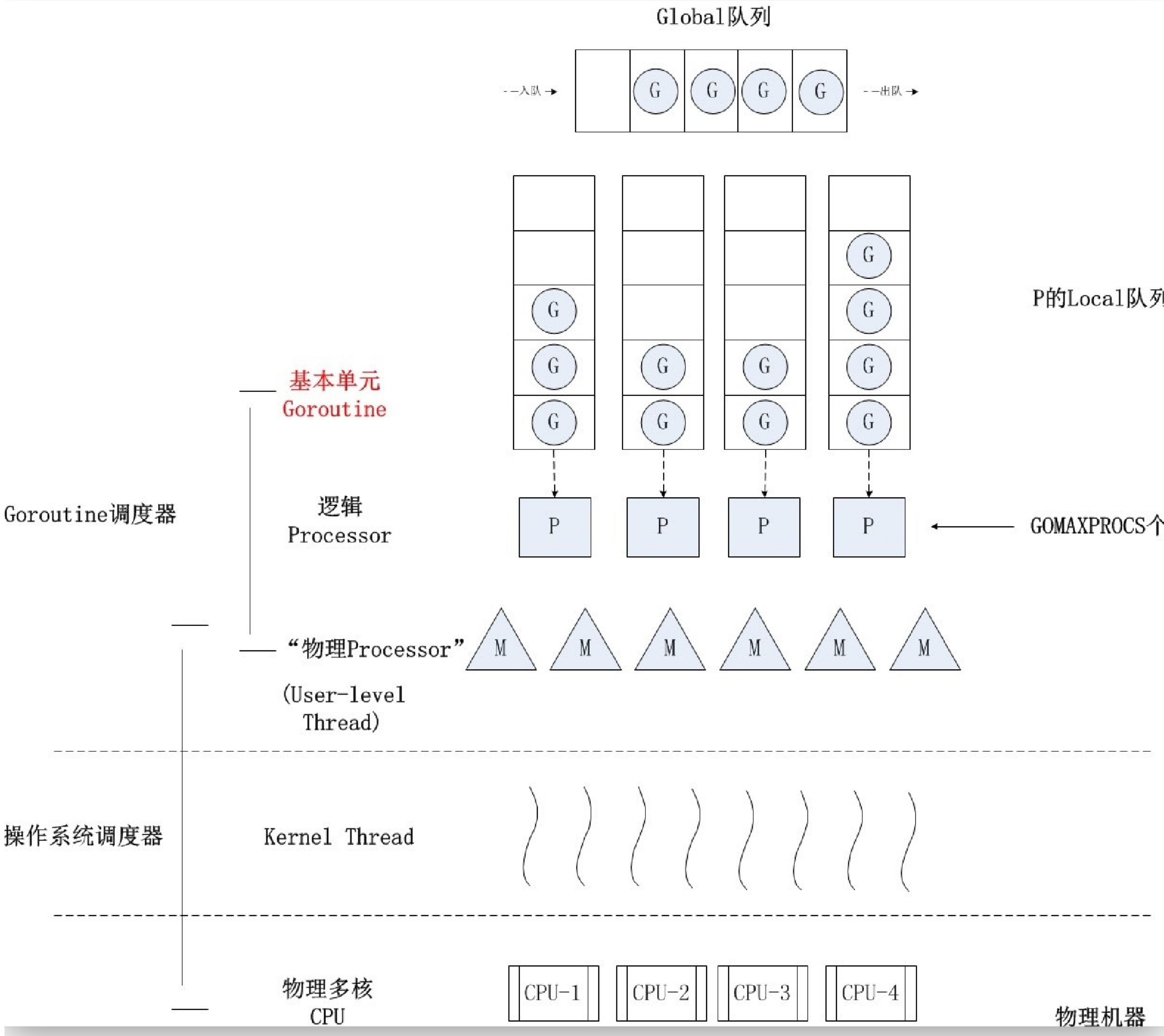
用户态上下文切换

Goroutine

栈轻量、可扩缩容（2KB ~ 1GB）

由用户态线程调度

Goroutine的调度 - GMP模型



并发从入门到放弃 - Race Condition

> 计算机中的两个进程同时试图修改一个共享内存的内容，在没有并发控制的情况下，最后的结果依赖于两个进程的执行顺序与时机。而且如果发生了并发访问冲突，则最后的结果是不正确的。(Wikipedia)

并发从入门到放弃 - （1）丧失原子性

```go
var sum int64

func main() {
	for i := 0; i < 1000; i++ {
		go func() {
			sum++
		}()
	}

	fmt.Println(sum)
}
```

并发从入门到放弃 -（2）丧失顺序性

```go
func f1() {
    fmt.Print(1)
}

func f2() {
    fmt.Print(2)
}

func f3() {
    fmt.Print(3)
}

func main() {
    go f1()
    go f2()
    go f3()
}
```

并发从入门到放弃 - (3) 丧失可见性

```go
var s string

func hello() {
	s = "hello"
}

func main() {
	go hello()
	fmt.Print(s)
}
```

原子性、顺序性、可见性都丧失了，还编个JB?

Race检查

Concurrency Primitive

- Sync - Mutex & Atomic
- CSP模型 - Channel

> CSP 是 Communicating Sequential Process 的简称，中文可以叫做通信顺序进程，是一种并发编程模型，是一个很强大的并发数据模型，是上个世纪七十年代提出的，用于描述两个独立的并发实体通过共享的通讯 channel(管道)进行通信的并发模型。相对于Actor模型，CSP中channel是第一类对象，它不关注发送消息的实体，而关注与发送消息时使用的channel。
> Do not communicate by sharing memory; instead, share memory by communicating. ——Effective Go

拯救刚才的例子 - (1) 原子性

sync/atomic

```
atomic.AddInt64(&sum, 1)
```

sync.Mutex

```
var mx sync.Mutex

mx.Lock()
sum++
mx.Unlock()
```

拯救刚才的例子 - (2) 顺序性

```go
func f1() {
    fmt.Print(1)
    ch1 <- 1
}

func f2() {
    <- ch1
    fmt.Print(2)
    ch2 <- 2
}

func f3() {
    <- ch2
    fmt.Print(3)
    ch3 <- 3
}
```

拯救刚才的例子 - (3) 可见性

```go
func hello() {
    s = "hello"
    ch <- 1
}

func main() {
    go hello()
    <- ch
    fmt.Print(s)
}
```

Understanding Real-World Concurrency Bugs in Go

Application	Shared Memory					Message		Total
	Mutex	atomic	Once	WaitGroup	Cond	chan	Misc.	
Docker	62.62%	1.06%	4.75%	1.70%	0.99%	27.87%	0.99%	1410
Kubernetes	70.34%	1.21%	6.13%	2.68%	0.96%	18.48%	0.20%	3951
etcd	45.01%	0.63%	7.18%	3.95%	0.24%	42.99%	0	2075
CockroachDB	55.90%	0.49%	3.76%	8.57%	1.48%	28.23%	1.57%	3245
gRPC-Go	61.20%	1.15%	4.20%	7.00%	1.65%	23.03%	1.78%	786
BoltDB	70.21%	2.13%	0	0	0	23.40%	4.26%	47

Table 4. Concurrency Primitive Usage. *The Mutex column includes both Mutex and RWMutex.*

Application	Behavior		Cause	
	blocking	non-blocking	shared memory	message passing
Docker	21	23	28	16
Kubernetes	17	17	20	14
etcd	21	16	18	19
CockroachDB	12	16	23	5
gRPC	11	12	12	11
BoltDB	3	2	4	1
Total	85	86	105	66

Table 5. Taxonomy. *This table shows how our studied bugs distribute across different categories and applications.*

Blocking Bugs

Application	Shared Memory			Message Passing		
	Mutex	RWMutex	Wait	Chan	Chan w/	Lib
Docker	9	0	3	5	2	2
Kubernetes	6	2	0	3	6	0
etcd	5	0	0	10	5	1
CockroachDB	4	3	0	5	0	0
gRPC	2	0	0	6	2	1
BoltDB	2	0	0	0	1	0
Total	28	5	3	29	16	4

Table 6. Blocking Bug Causes. *Wait includes both the Wait function in* `Cond` *and in* `WaitGroup`. *Chan indicates channel operations and Chan w/ means channel operations with other operations. Lib stands for Go libraries related to message passing.*

- Kubernetes

```go
    func finishReq(timeout time.Duration) r ob {
-     ch := make(chan ob)
+     ch := make(chan ob, 1)
      go func() {
        result := fn()
        ch <- result // block
      } ()
      select {
        case result = <- ch:
          return result
        case <- time.After(timeout):
          return nil
      }
    }
```

Figure 1. A blocking bug caused by channel.

- Docker#25384

```go
  var group sync.WaitGroup
  group.Add(len(pm.plugins))
  for _, p := range pm.plugins {
    go func(p *plugin) {
      defer group.Done()
    }
-   group.Wait()
  }
+ group.Wait()
```

Figure 5. A blocking bug caused by WaitGroup.

- context

```go
hctx, hcancel := context.WithCancel(ctx)
if timeout > 0 {
    //hcancel.Cancel()
    hctx, hcancel = context.WithTimeout(ctx, timeout)
}
```

```go
  func goroutine1() {
    m.Lock()
-   ch <- request //blocks
+   select {
+     case ch <- request
+     default:
+   }
    m.Unlock()
  }
```

(a) goroutine 1

```go
func goroutine2() {
  for {
    m.Lock()     //blocks
    m.Unlock()
    request <- ch
  }
}
```

(b) goroutine 2

Figure 7. A blocking bug caused by wrong usage of channel with lock.

Non-Blocking Bugs

Application	Shared Memory				Message Passing	
	traditional	anon.	waitgroup	lib	chan	lib
Docker	9	6	0	1	6	1
Kubernetes	8	3	1	0	5	0
etcd	9	0	2	2	3	0
CockroachDB	10	1	3	2	0	0
gRPC	8	1	0	1	2	0
BoltDB	2	0	0	0	0	0
Total	46	11	6	6	16	1

Table 9. Root causes of non-blocking bugs. *traditional: traditional non-blocking bugs; anonymous function: non-blocking bugs caused by anonymous function; waitgroup: misusing WaitGroup; lib: Go library; chan: misusing channel.*

- Docker

```
  for i := 17; i <= 21; i++ { // write
-     go func() { /* Create a new goroutine */
+     go func(i int) {
          apiVersion := fmt.Sprintf("v1.%d", i) // read
          ...
-     }()
+     }(i)
  }
```

Figure 8. A data race caused by anonymous function.

- etcd

```
func (p *peer) send() {
    p.mu.Lock()
    defer p.mu.Unlock()
    switch p.status {
        case idle:
+           p.wg.Add(1)
            go func() {
-               p.wg.Add(1)
                ...
                p.wg.Done()
            }()
        case stopped:
    }
}
```

(a) func1

```
func (p * peer) stop() {
    p.mu.Lock()
    p.status = stopped
    p.mu.Unlock()
    p.wg.Wait()
}
```

(b) func2

Figure 9. A non-blocking bug caused by misusing WaitGroup.

- Docker

```
- select {
-   case <- c.closed:
-   default:
+     Once.Do(func() {
        close(c.closed)
+     })
- }
```

Figure 10. A bug caused by closing a channel twice.