1. **sync.Cond 使用场景**

有一个协程正在接收数据，其他协程必须等待这个协程接收完数据，才能读取到正确的数据。

上述情形下，如果单纯的使用 channel 或者互斥锁，只能有一个协程可以等待，并读取到数据，没办法通知其他协程也读取数据。

这个时候怎么办？

* 可以用一个全局变量标识第一个协程是否接收数据完毕，剩下的协程反复检查该变量的值，直到读取到数据。
* 也可创建多个 channel, 每个协程阻塞在一个 Channel 上，由接收数据的协程在数据接收完毕后，挨个通知。

然后 Go 中其实内置来一个 sync.Cond 来解决这个问题。

Golang 的 sync 包中的 Cond 实现了一种条件变量，可以使用多个 Reader 等待公共资源。

每个 Cond 都会关联一个 Lock ,当修改条件或者调用 Wait 方法，必须加锁，保护 Condition。sync.Cond 条件变量是用来协调想要共享资源的那些 goroutine, 当共享资源的状态发生变化时，可以被用来通知被互斥锁阻塞的 gorountine。

2. **sync.Cond**

type Cond struct {

noCopy noCopy

// L is held while observing or changing the condition

L Locker

notify notifyList

checker copyChecker

}

每个 Cond 都会关联一个 锁 L (互斥锁 Mutex, 或者读写锁 \* RMMutex), 当修改条件或者使用 Wait 的时候必须要加锁。

3. **sync.Cond 有哪些方法**

1. NewCond 创建实例 func NewCond(l Locker) \*Cond

NewCond 创建实例需要关联一个锁。

具体实例：

cadence := sync.NewCond(&sync.Mutex{})

1. Broadcast 广播唤醒所有 func (c \*Cond) Broadcast()

Broadcast 唤醒所有等待条件变量 c 的 goroutine，无需锁保护。

具体实例：

go func() {

for range time.Tick(1 \* time.Millisecond) {

cadence.Broadcast()

}

}()

1. Signal 唤醒一个协程 func (c \*Cond) Signal()

Signal 只唤醒任意1个等待条件变量 c 的 goroutine，无需锁保护。

1. Wait 等待 func (c \*Cond) Wait()

调用 Wait 会自动释放锁 c.L，并挂起调用者所在的 goroutine，因此当前协程会阻塞在 Wait 方法调用的地方。如果其他协程调用了 Signal 或 Broadcast 唤醒了该协程，Wait 方法结束阻塞，并重新给 c.L 加锁，并且继续执行 Wait 后面的代码

代码示例：

c.L.Lock()

for !condition() {

    c.Wait()

}

... make use of condition ...

c.L.Unlock()