首先，在实例化 worker 的函数 ：

```
func newWorker(config *params.ChainConfig, eng
    worker := &worker{
        config:              config,
```

最底层有 4 个 异步的协程【请注意这四个协程】：

```
    go worker.mainLoop()
    go worker.newWorkLoop(recommit)
    go worker.resultLoop()
    go worker.taskLoop()
```

其中 newWorkLoop 是一个一直监听 startCh 中是否有挖矿信号 (startCh 的信号有 start 函数放置进去的)：

```
for {
    select {
    case <-w.startCh:
        commit( noempty: false, commitInterruptNewHead)

    case <-w.chainHeadCh:
        commit( noempty: false, commitInterruptNewHead)
```

如果有 就提交一个挖矿作业 方法 commit（）

我们又可以看到 commit 里面其实是 构造了一个 挖矿的请求 实体而已，并且再把 请求实体交由 w.newWorkCh 通道【请记住这个通道】；

```
commit := func(noempty bool, s int32) {
    if interrupt != nil {
        atomic.StoreInt32(interrupt, s)
    }
    interrupt = new(int32)
    w.newWorkCh <- &newWorkReq{interrupt: interrupt, noempty: noempty}
    timer.Reset(recommit)
    atomic.StoreInt32(&w.newTxs, val: 0)
}
```

下面我们再来看 newWorker 初始化函数中的四个协程之一的 worker.mainLoop()

```
for {
    select {
    case req := <-w.newWorkCh:
        w.commitNewWork(req.interrupt, req.noempty)
```

我们可以看到它里面会一直监听着 w.newWorkCh【注意这个就是我们上一步中往里头放 挖矿请求的通道】

发现有请求过来 则再次提交一个叫做 w.commitNewWork(req.interrupt, req.noempty) 挖矿作业提交，我们再跟进去看，发现他无非就是做各种各样的 block 预处理工作，到方法的末尾 交付给了 w.commit(uncles, w.fullTaskHook, true, tstart)

```
        }
    }
    w.commit(uncles, w.fullTaskHook, update: true, tstart)
}
```

我们再跟进去 看看里面是做了什么

发现 w.commit(uncles, w.fullTaskHook, true, tstart) 里面是先对 收据数据先做一些处理然后把构造好的 收据 receipts 状态 state 及预先打包好的 block (里头有些内容需要真正挖矿来回填的，比如 随机数，出块时间等等) 构造成一个任务实体 task 并提交给 w.taskCh 通道 【注意了 重头戏来了】

```
}
if w.isRunning() {
    if interval != nil {
        interval()
    }
    select {
    case w.taskCh <- &task{receipts: receipts, state: s, block: block, createdAt: time.Now()}:
        w.unconfirmed.Shift(block.NumberU64() - 1)
```


我们再回去看 newWorker 函数中四个协程的 worker.taskLoop()

```go
// push them to consensus engine.
func (w *worker) taskLoop() {
    var (
        stopCh chan struct{}
        prev   common.Hash
    )

    // interrupt aborts the in-flight sealing task.
    interrupt := func() {
        if stopCh != nil {
            close(stopCh)
            stopCh = nil
        }
    }
    for {
        select {
        case task := <-w.taskCh:
            if w.newTaskHook != nil {
                w.newTaskHook(task)
            }
            // Reject duplicate sealing work due to resubmitting.
            if task.block.HashNoNonce() == prev {
                continue
            }
            interrupt()
            stopCh = make(chan struct{})
            prev = task.block.HashNoNonce()
            go w.seal(task, stopCh)
        case <-w.exitCh:
            interrupt()
```

在里头一直监听这个 taskCh 通道过来的 task 实体，一有就启动 seal() 函数把这些信息 交由底层的共识层的实现去 做真正的挖矿

我们跟进 go w.seal(task, stopCh) 里头看看做了什么

```go
// seal pushes a sealing task to consensus engine and submits the result.
func (w *worker) seal(t *task, stop <-chan struct{}) {
    var (
        err error
        res *task
    )

    if w.skipSealHook != nil && w.skipSealHook(t) {
        return
    }

    if t.block, err = w.engine.Seal(w.chain, t.block, stop); t.block != nil {
        log.Info( msg: "Successfully sealed new block", ctx: "number", t.block.Number(),
            "elapsed", common.PrettyDuration(time.Since(t.createdAt)))
        res = t
    } else {
        if err != nil {
            log.Warn( msg: "Block sealing failed", ctx: "err", err)
        }
        res = nil
    }
    select {
    case w.resultCh <- res:
    case <-w.exitCh:
    }
}
```

发现把之前预先构造好的 block 等都交由底层共识去做了，最终把挖出来的 block 通过 w.resultCh 通道返回，【这一步和以前用 Agent 挖矿的逻辑的底层一致】

最后处理结果是那四个协程中的 go worker.resultLoop() 来处理的，so 本次改动去除了 Agent 这个代理类，全部动作都交由 newWorker 函数中的四个协程去实现，且相互之间通过 对应的通道来通信！完美