这里我没列出 pin 代码的实现，你只需要知道，pin 方法会将此 goro上，避免查找元素期间被其它的 P 执行。固定的好处就是查找元素期间直接得到跟这个 P 相关的 local。有一点需要注意的是，pin 方法在执行的时候，如果跟这个 P 相关的 local 还没有创建，或者运行时 P 的数量被修改了的话，就会新创建 local。

Put 方法

我们来看看 Put 方法的具体实现原理。

复制代码

```
func (p *Pool) Put(x interface{}) {
    if x == nil { // nil值直接丢弃
        return
    }
    l, _ := p.pin()
    if l.private == nil { // 如果本地private没有值，直接设置这个值即可
        l.private = x
        x = nil
    }
    if x != nil { // 否则加入到本地队列中
        l.shared.pushHead(x)
    }
    runtime_procUnpin()
}
```

Put 的逻辑相对简单，优先设置本地 private，如果 private 字段已经有值了，那么就把此元素 push 到本地队列中。

sync.Pool 的坑

到这里，我们就掌握了 sync.Pool 的使用方法和实现原理，接下来，我要再和你聊聊容易踩的两个坑，分别是内存泄漏和内存浪费。

内存泄漏

这节课刚开始的时候，我讲到，可以使用 sync.Pool 做 buffer 池，但是，如果用刚刚的那种方式做 buffer 池的话，可能会有内存泄漏的风险。为啥这么说呢？我们来分析一下。

取出来的 bytes.Buffer 在使用的时候，我们可以往这个元素中增加大量的 byte 数据，这会导致底层的 byte slice 的容量可能会变得很大。这个时候，即使 Reset 再放回到池子中，