注意，在模板中我们直接访问 plusOne 即可，不用访问 plusOne.value，这和我们前面分析的 ref 对象在模板中不需要访问.value 属性的原理是一样的，因为计算属性对象也拥有__v_isRef 属性。

在组件渲染阶段，会访问 plusOne，也就触发了 plusOne 对象的 getter 函数：

```
get value() {
  // 对 value 属性设置 getter
  const self = toRaw(this)
  // 依赖收集
  trackRefValue(self)
  if (self._dirty) {
    // 只有数据为“脏”的时候才会重新计算
    self._dirty = false
    self._value = self.effect.run()
  }
  return self._value
}
```

首先会执行 trackRefValue，对计算属性本身做依赖收集，这个时候 activeEffect 是组件副作用渲染函数对应的 effect 对象。

然后会判断 dirty 属性，由于_dirty 默认是 true，所以这个时候会把_dirty 设置为 false，接着执行计算属性内部 effect 对象的 run 函数，并进一步执行 computed getter，也就是 count.value + 1。因为访问了 count 的值，且 count 也是一个响应式对象，所以也会触发 count 对象的依赖收集过程。

请注意，由于是在 effect.run 函数执行的时候访问 count，所以这个时候的 activeEffect 指向计算属性内部的 effect 对象。因此要特别注意，这是两个依赖收集过程：对于 plusOne 来说，它收集的依赖是组件副作用渲染函数对应的 effect 对象；对于 count 来说，它收集的依赖是计算属性 plusOne 内部的 effect 对象。

当我们点击按钮的时候，会执行 plus 函数。函数内部通过 count.value++修改 count 的值，并派发通知。由于 count 收集的依赖是 plusOne 内部的 effect 对象，所以会通知 effect 对象。但是请注意，这里并不会直接调用 effect.run 函数，而是会执行 effect.scheduler 函数。我们来回顾一下 triggerEffects 函数的实现：

```
function triggerEffects(dep, debuggerEventExtraInfo) {
  for (const effect of isArray(dep) ? dep : [...dep]) {
    if (effect !== activeEffect || effect.allowRecurse) {
      if ((process.env.NODE_ENV !== 'production') && effect.onTrigger) {
        effect.onTrigger(extend({ effect }, debuggerEventExtraInfo))
      }
      if (effect.scheduler) {
```