**首先实现一个简单的单线程的mergesort**

第一次调用在切片的前半部分完成，而第二次调用在后半部分完成。最后，merge函数合并两半并确保数组被排序

看了一眼时间：benchmark的时间是sort.slice的两倍多一些。（失败） 。

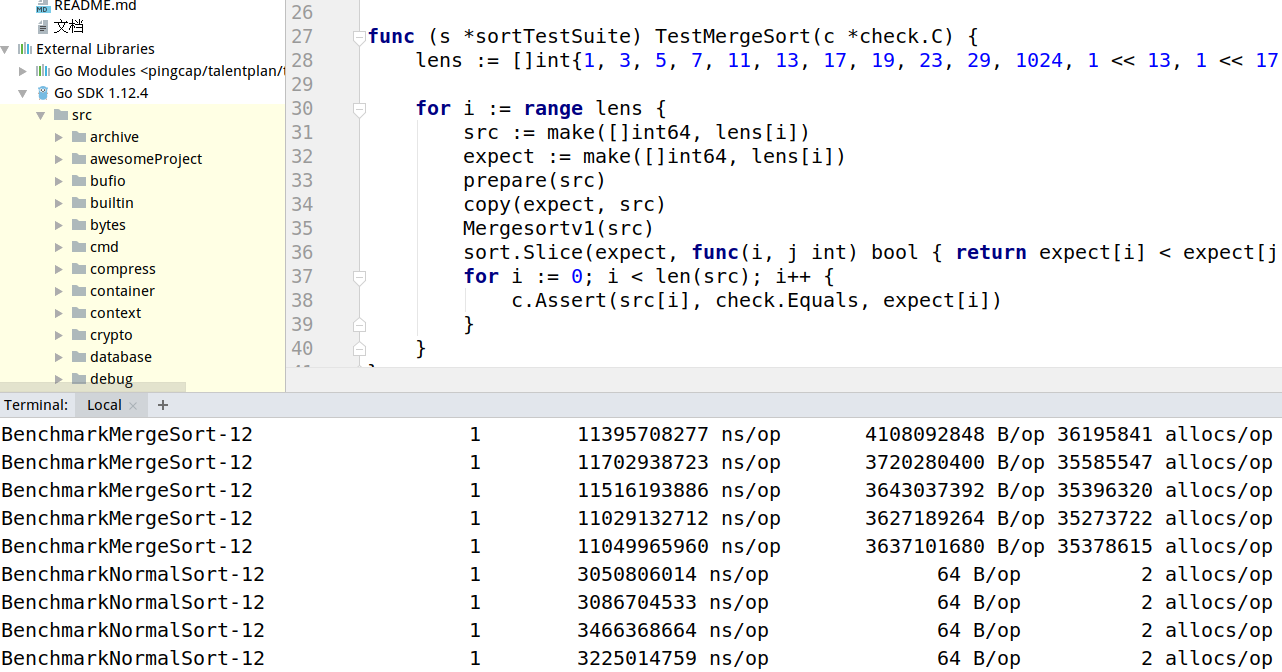
**考虑优化**

使用gouroutine并发归并排序。

**Mergesort V1版本**

不是每次mergesort调用创建一个协程，主要的想法是通过协程池处理并行性并分配工作负载。然后，每个任务都可用于池里的协程。（考虑过使用channel，但还是觉得waitgroup更好控制）。

使用WaitGroup同时使用两个goroutine,并在goroutine执行结束之后开始merge。make bench 35.391s



运行时间是normalsort的三倍。（失败）

**猜想**

应该是goroutine大量的调用导致的，那么接下来应该控制goroutine的调用。

**发现工具go trace**

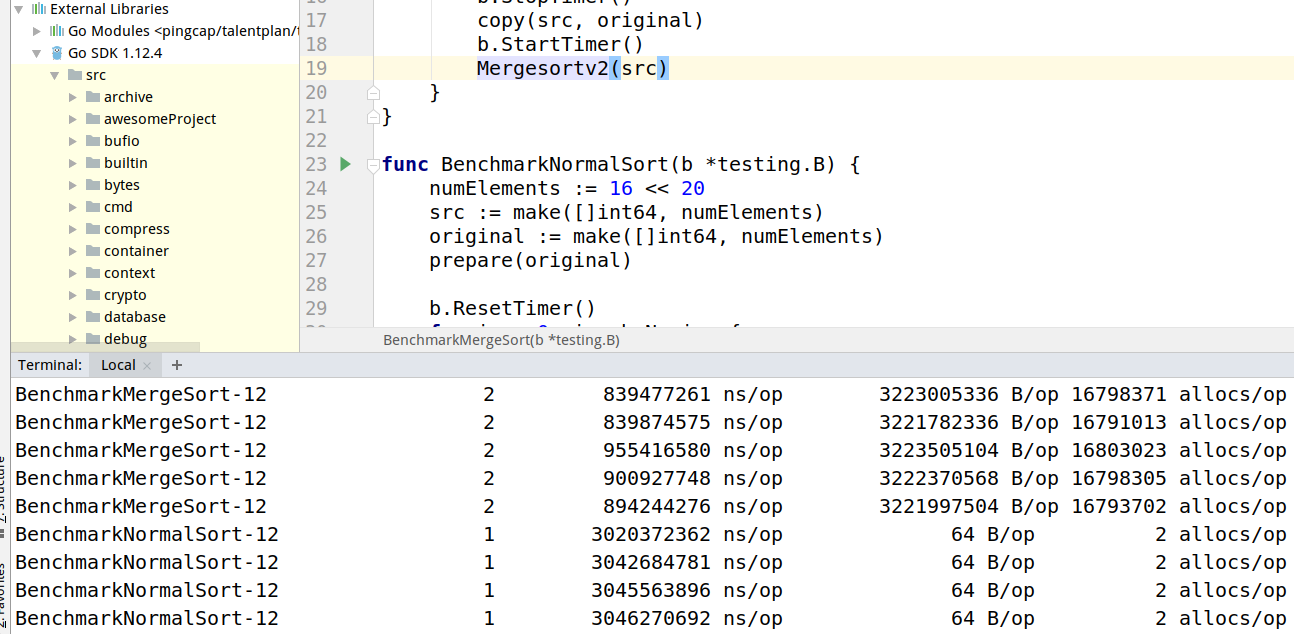
根据go trace的图发现执行期间使用了所有的cpu内核，然而中间有很多空白区域，这说明cpu并没有专注执行我的代码，空白是由大量的上下文切换引入的开销（即大量goroutine大量的gc） 。

**思考**

假如我有一个512个元素的切片，那么作用于mergesort的就有256，以此类推直到1个元素，但此时我们还要为它单独启一个goroutine,这就很浪费了，所以我们设置一个最大值max在小于max时使用循序实现（mergesort），在大于max时使用并发实现(v1)。经过多次测试max为1<<11到1<<13之间,等于2048时，表现最佳（应该是电脑配置不同导致这里使用了两台电脑做测试一台6核12线程，一台4核8线程，内存都是16G） 。

**Mergesort V2版本**

make bench 33.398s



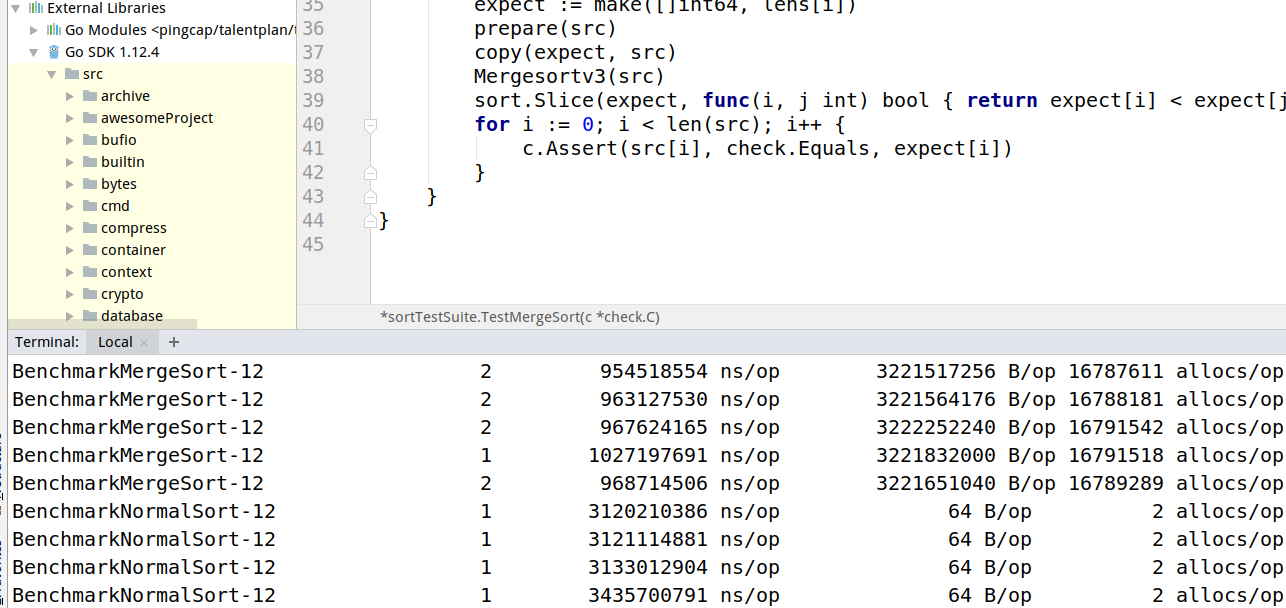
运行时间比normalSort快。（成功！）

再次查看go trace空白区域大量减少，gc暂停时间也大量减少。

**考虑再次减少gouroutine的调用**

**V3版本**

只生成一个goroutine处理调用，用主goroutine处理另外一个调用。make bench 33.840s



运行时间比normalSort快。比起V2无明显优化。查看go trace调度时间没有明显提升。但gc暂停时间快了35%（舒服了）。