# 文 后端的轮子（三）--- 缓存

后端 缓存设计 吴yh坚 1 天前发布

今天这一篇没想到会这么长，后面有一段是写网络模型的，和缓存本身的关系不大，只是写到那里就想到了这个问题，多写了一些，那一段是我自己的理解，肯定有不对的地方，欢迎讨论拍砖。

## 前言

前面花了一篇文章说数据库这个轮子，其实说得还很浅很浅的，真正的数据库比这复杂不少，今天我们继续轮子系列，今天说说缓存系统吧。

缓存是后端使用得最多的东西了，因为 性能 是后端开发一个重要的特征，所以缓存就应运而生了，而且现在缓存已经到了泛滥的程度了，我几乎没见过没有缓存的后端，一遇到性能问题，首先想到的不是看代码，而是加缓存，我也是醉了，好了，不扯这些，这些和今天的文章无关，今天我们来专门讲讲缓存吧。

## 缓存和KVDB

缓存和KVDB两个东西经常一起出现，两者在使用上没有明显的界限，当一个KVDB速度够快，性能够强劲，那么就可以当缓存来用了，我们使用 `Redis` 来做缓存，实际上就是把一个KVDB来当缓存用。但一般情况下，KVDB能提供更多的数据结构，所以象 `Redis` 这样的KVDB中有很多实用的数据结构，比如List啊，hashtable啊之类的，而且KVDB一般都提供持久化的存储，而像 `memcached` 这样的纯缓存一般不提供持久化存储功能，而且数据结构也比较简单，仅仅提供key和value都是字符串的形式。

现在KVDB的代表 `Redis` 性能已经越来越强劲了，虽然它是个单线程的服务，但目前基本上能用 `memcached` 的都可以用 `Redis` 代替，而且 `Redis` 因为支持更多的数据结构，所以扩展性更好。现在很多情况下所说的缓存，实际上都是指的是 `Redis` 缓存。

## 缓存的类型

我们这里抛开 `Redis`，来单独说说缓存，所谓缓存，实际上是为了给数据提供一个更快的访问方式，这个更快一般是相对于最终数据而言的，最终数据可以是 `KVDB` 中的数据，也可以是 文件数据，还可以是其他机器上的 数据库数据，只要比这些个数据访问的快的，都可以叫缓存，那么一般缓存分成一下几种。

- 数据库型缓存，比如最终数据是其他机器上的 `MySql数据库中`，那么我们做一个 `KVDB的数据库`，查询的时候按照key查询，总比数据库要快点吧，那这个 `KVDB` 就是个缓存。
- 文件型的缓存，进一步说，远程的 `KVDB` 还是有网络延迟，慢了点，这时候我在本地做一个文件缓存，这个文件的访问速度比远程数据库要快吧，那这个文件也是个缓存。
- 内存型的缓存，再进一步，本地文件还是嫌慢了，那么我们在做一个内存缓存，把最热的数据存到内容中，那这个内存的访问速度一定比文件要快，那这个内存块也是个缓存。

> 所以说，只要比最终数据访问得快的数据结构，就是一个缓存系统。

为了一般性，我们这里所说的缓存轮子，将会说一个内存型的缓存，只提供简单字符串类型的key和value的操作。

## 如何设计一个缓存

设计一个独立的内存型的缓存系统，首先先要确定缓存最关心的东西，那就是 性能，所有的需要考虑的东西都是围绕 性能 两个字来进行设计的，所以 最重要 的部分，设计一个缓存需要考虑以下三个方面， 底层数据结构， 内存管理， 网络模型。

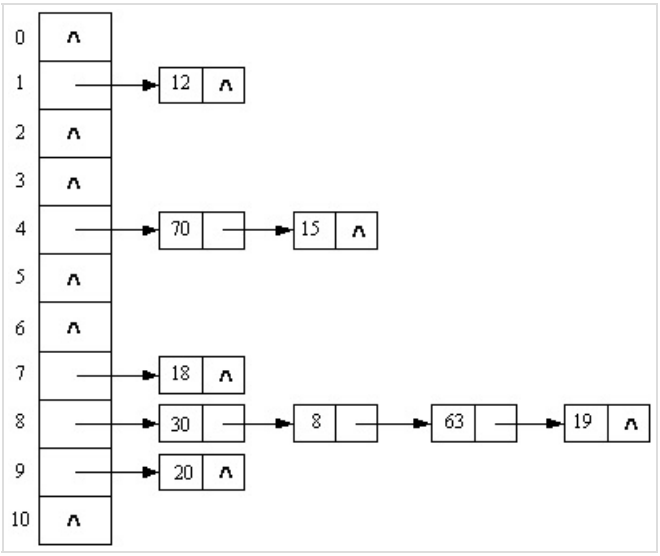
## 底层数据结构

又看到 数据结构 这个词了，我们所说的所有轮子，都跑不掉 数据结构 这个东西，对于缓存来说，一般都是KV形式的数据结构，所以底层一般会使用 树 或者 哈希表 来保存数据，而缓存对性能的要求更高，所以一般使用哈希表来保存数据，所以底层的数据结构就是哈希表了。

# 哈希表

## 哈希函数和类型

选择哈希表，是因为他的O(1)的查询复杂度，这是一个很重要的性能指标，但如果哈希函数没有选择好，产生了大量的哈希碰撞，那性能就会急剧降低，所以对于哈希函数的选择也是一个需要考虑的问题，比较流行的哈希函数有很多，特别的，如果是自用型的缓存，哈希函数可以根据业务场景再来调整，保证哈希的均匀，从而让查询复杂度更加接近O(1)。
哈希表的实现方式有很多中，最最基础的就是**数组+链表**的形式了，也叫开链哈希，数组长度就是哈希的桶的长度，链表用来解决冲突，插入数据的时候如果哈希碰撞了，把具体节点挂在该节点后面的链表上，查询数据时候有冲突，就继续线性查询这个节点下的链表。

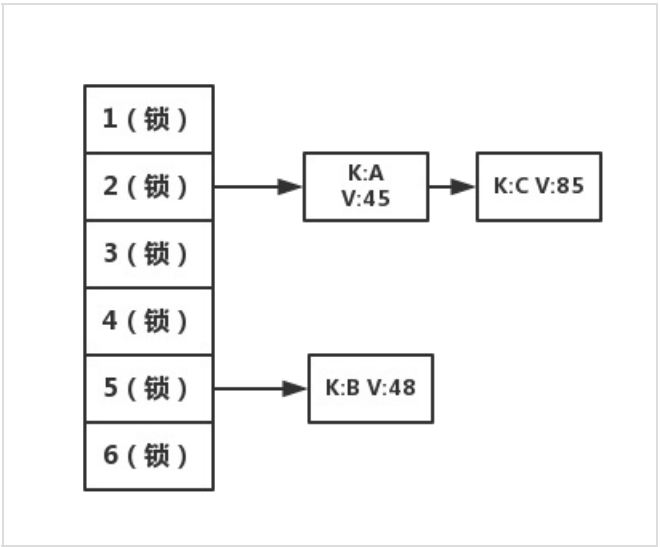



还有一种叫闭链哈希，闭链哈希实际是一个循环数组，数组长度就是桶的长度，插入数据的时候有冲突的话，移动到该节点的下一个，直到没有冲突为止，如果移动到了末尾的话，转到数组的头部，查找数据的时候类似。

我们这里说的哈希表都是第一种开链哈希表。

## 锁

由于缓存不仅仅有读，还有写操作，如果在多线程的场景下，势必会产生加锁的操作，如果设计这个锁也是需要考虑的，如果写操作不是很多的情况下，那么整个哈希表加读写锁就行了，但如果写操作也比较频繁，那么可以为一批哈希槽或者每一个哈希槽加一把锁，这样的话，可以把锁等待的时间延迟给降下来，具体还是要看场景，我实现的时候是给每个槽加了一个读写锁，这样更耗费内存，但是性能好一些。这种类型的哈希表的数据结构长成下面这个图的样子。



每一个槽配了一把读写锁，每次写的时候都对单个槽进行加锁操作，这样的坏处就是需要维护巨多无比的锁，容易造成浪费，在实际中我们可以根据实测的结果，给一批槽加一把锁，这样也可以把锁资源空下来，并且也能达到比较好的并发效果。

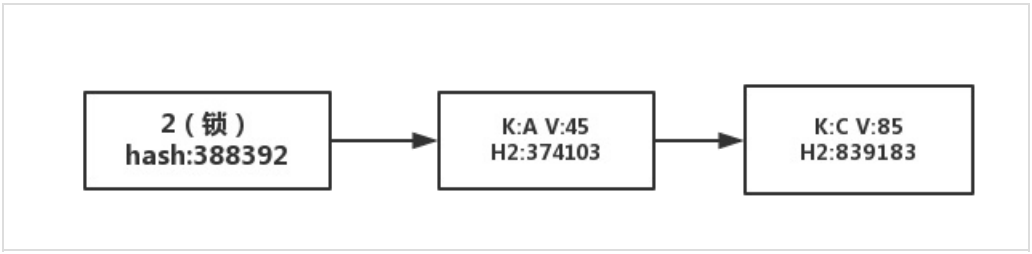
> 关于锁设计，再多说一下，多线程（或者多协程）的情况下，加锁是一种常规处理方式，现在的X86架构，支持一种 `CAS` 的无锁操作模式，是在CPU层面实现的对变量的多线程同步技术，golang中有个 `atomic` 包，简单的封装了这个功能，但是这个操作在进行变量更新的时候一般要在一个循环中来实现，不停的尝试直到成功为止，虽然说减少了锁的操作，但代码看起来没那么清晰，而且如果出了问题，调试起来也没有锁那么清晰，并且虽然是CPU级别的支持，但是还是有问题的，就是线程切换的时候还是会造成不可预知的错误，这里就不展开了，感兴趣的可以自己去搜索一下 `CAS无锁操作` ，并且在一般的缓存中还是读多写少，通过把锁扩展到槽级别基本上性能不会出现很大的损耗，当然，如果你对性能有着极致的追求，可以考虑 `CAS` 方案，但是也要注意坑哦。

## 字符串和整数

最后，我们看看对于单个的具体的哈希槽，在发生哈希碰撞的时候一个槽下面可能挂了很多节点。

当进行读操作的时候，如果哈希到这个槽下面来了，我们需要比较每个节点的key和查询串的值，只有相等的情况才是我们需要的。比较每个key的值是进行了一次字符串的比较，效率是比较低的，这里继续出现一个用空间换时间的方法，就是我们在插入节点的时候给每个节点的key生成两个哈希值，第一个哈希值用来进行槽的选择，第二个哈希值保存在节点内，查询的时候不进行key的字符串比较而比较第二个哈希值，由于哈希值是整数的，所以比较效率比直接比较字符串要快多了。用这样的方式，在写入数据和查询数据的时候需要进行两次哈希计算，并且还需要有个单独的空间来存储第二个哈希值，但是查询的时候可以节省字符串比较的时间。

> 对于两个字符串的比较，平均时间复杂度是O(n/2)吧，而对于两个整数的比较，一个异或操作就搞定了，谁快就不用说了吧，这个槽变成这样了。




## 重哈希(reHash)

对于不断增长的数据而言，重哈希是一个必不可少的过程，所谓重哈希，就是当你的桶使用到一定程度以后碰撞的概率就变很大了，这时候就需要把桶加大了。

把桶加大，必然需要进行一次重新哈希的过程，这个过程的处理办法也有一些技巧。

- 直接重新哈希，这是最简单的，就是把所有的key重新哈希一遍放到新的桶中，简单粗暴，但是缺点也很明显，就是当key很多的时候非常耗时间和资源，在这段时间中，服务是不可用的。
- 
- 逐步迁移的重哈希，因为桶的变化，重新哈希是无法避免的，这时候我们主要要考虑的是让服务尽可能的保持可用，那么除了直接哈希，还有一种策略上的优化，简单的描述就是申请两个桶A和B，B的桶数量大于A
    - 初始化申请的时候可以并不实际申请内存空间，首先用第一个桶A进行数据存储，当第一个桶A使用到一定比例，比如80%的时候，开始进行哈希迁移。
    - 新来的写操作先哈希到桶B，然后在哈希到桶A上，把A桶上命中的节点上的所有数据重新哈希到B上，然后在A上打一个标记，表示这个节点失效了。
    - 新来的读操作，先哈希到A进行命中，如果A的这个节点标记为失效，再哈希到B上读取正确数据，如果A节点没有标记失效，那么把这个节点下的所有数据重新哈希到B上，并在A上打一个标记，表示这个节点失效了。
    - 直到所有的A的数据都迁移完成，把A和B交换一下，并且把A的桶数据增加，作为下一次迁移使用。

  这样，当进行重新哈希的时候，多了几次哈希运算，性能损失了一些，但是服务始终是可用的。

# 内存管理

还有一个方面就是内存了，因为有写操作，那么就需要申请内存，如果写操作比较多的话，再加上有删除操作的话，那就会不停的申请释放内存。内存的申请和释放也是比较损耗性能的，所以一般会用自己的内存池来进行内存的分配。关于内存池这一块，有很多内存池的实现方式，这里就不详细说

了。

其实我觉得这种对性能有强要求的服务，不太适合使用带GC的语言进行编写，最直接的还是用C这种系统语言来编写，特别是涉及到内存池这种比较靠底层的东西，有GC实际上是很麻烦的事情，在Golang中，因为是自带GC的，如果需要进一步榨干系统的性能，那么这么底层的东西要用CGO来实现了，把GC丢一边，所有的内存都自己管。

## 网络模型

除了在底层数据结构层的性能损耗外，网络模型的选择也是很重要的，选择性能尽可能高的网络模型也能极大的提升性能，比如`memcached`就用了`libevent`这个事件模型，总的来说就是先通过一个主线程监听端口，接收到网络文件描述符以后，然后通过`master-worker`这种结构将网络的套接字分发到各个worker线程的独立队列中，各个线程利用`libevent`模型对队列中的套接字进行读写。

而`Redis`的网络模型更简明一点，但也是基于epoll的IO多路复用，感兴趣的朋友可以自己去看看Redis的源码，他相当于是一个稍微简化版本的`libevent`模型。

> 关于`libevent`和`libev`这两个模型（其实差不太多），我们可以专门写一篇文章分析一下源码，他们的源码都不多，正好我也看过，可以写一篇文章，当然网上这类文章也很多。

对于网络模型，实际上现代的高级语言已经基本上封装到http这个层次了，比如golang这种现代语言，http的包都可以直接用，并且并发性能也挺好的，但是对于一个缓存系统，如果配合一个http的模型，就显得太重了，http底下的TCP模型就可以很好的解决问题，对于这一块，我们可以用个简单的模型：

- 根据CPU的核心数启动相应数量的协程，每个协程配合一个channel
- 启动一个协程负责接收tcp连接，accpet链接以后通过channel交给相应的协程处理

这段代码虽然使用了channel这个东西，但也算是一个比较标准的多线程编程方式了，在多线程的世界中也可以用循环链表来表示这个channel，用代码来体现大概就是这样子的。

```
func GetConnection() error {
    tcpAddr, _ := net.ResolveTCPAddr("tcp", ":26719")
    listener, _ := net.ListenTCP("tcp", tcpAddr)
    //启动处理协程
    for i := 0; i < cpuCores; i++ {
        go Process(i)
    }
    i:=0
    for {
        conn, _ := listener.AcceptTCP()
        select {
        case processChan[i] <- conn:  //通过管道交给协程处理
                i++
                 if i==cpuCores{ i = 0 }
        default:
        }
    }
}
```

除了上面那种模型，还有一种比较奔放的模型，也是比较golang的写法了。

- 启动一个协程负责接收tcp连接，accpet链接以后go出一个协程进行处理
  代码实现就是下面这个样子

```
func GetConnection() error {
    tcpAddr, _ := net.ResolveTCPAddr("tcp", ":26719")
    listener, _ := net.ListenTCP("tcp", tcpAddr)
    for {
        conn, _ := listener.AcceptTCP()
        go Process(conn) //处理实际连接
    }
}
```

## 关于golang的协程分析

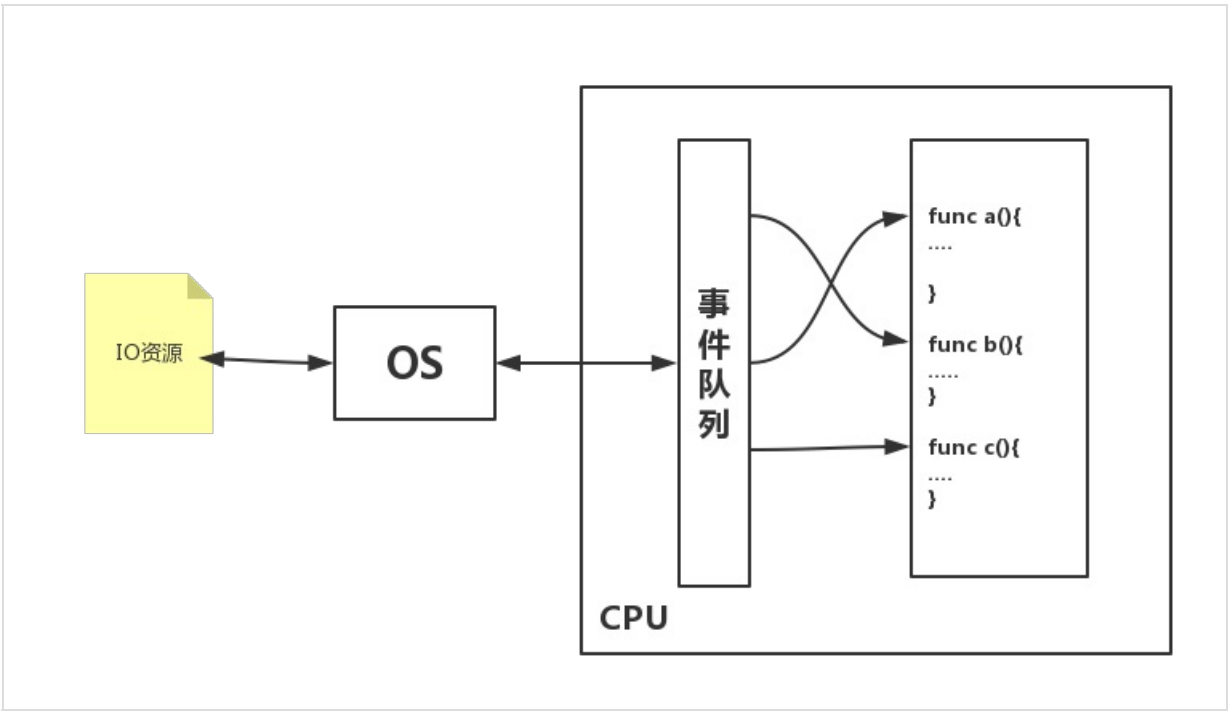
这种奔放的协程使用方式到底行不行？这两种到底哪个比较好？这里我们不讨论这两种模型哪种好，我们看看golang的协程吧。

golang的协程网上资料很多，关于协程的详细设计可以找到不少文章，总的来说就是这是一个轻量级的线程，有多轻呢？轻到它其实就是一个代码段加上一个自己的栈，光这个还不够，线程也可以说是一个代码片段加上一个自己的栈，只是协程的栈比线程的小而已，除了这个以为，协程主要的轻表现在它不通过操作系统调度，他是通过`代码`内部进行`显示调度`的。

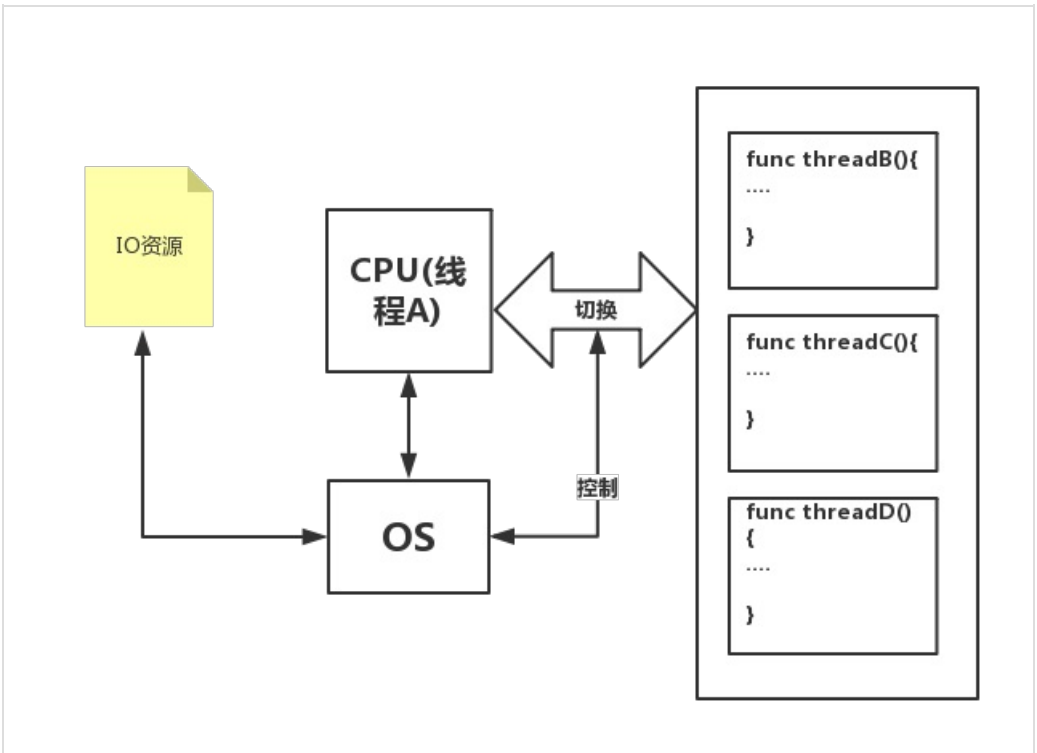
**什么叫`代码内部`进行`显示调度`呢？**

我们先看看目前流行的两个事件模型【同时也是处理网络连接的网络模型】，一种是`nodejs`为代表的IO模型，大堆回调函数，一种是传统的多线程模型。

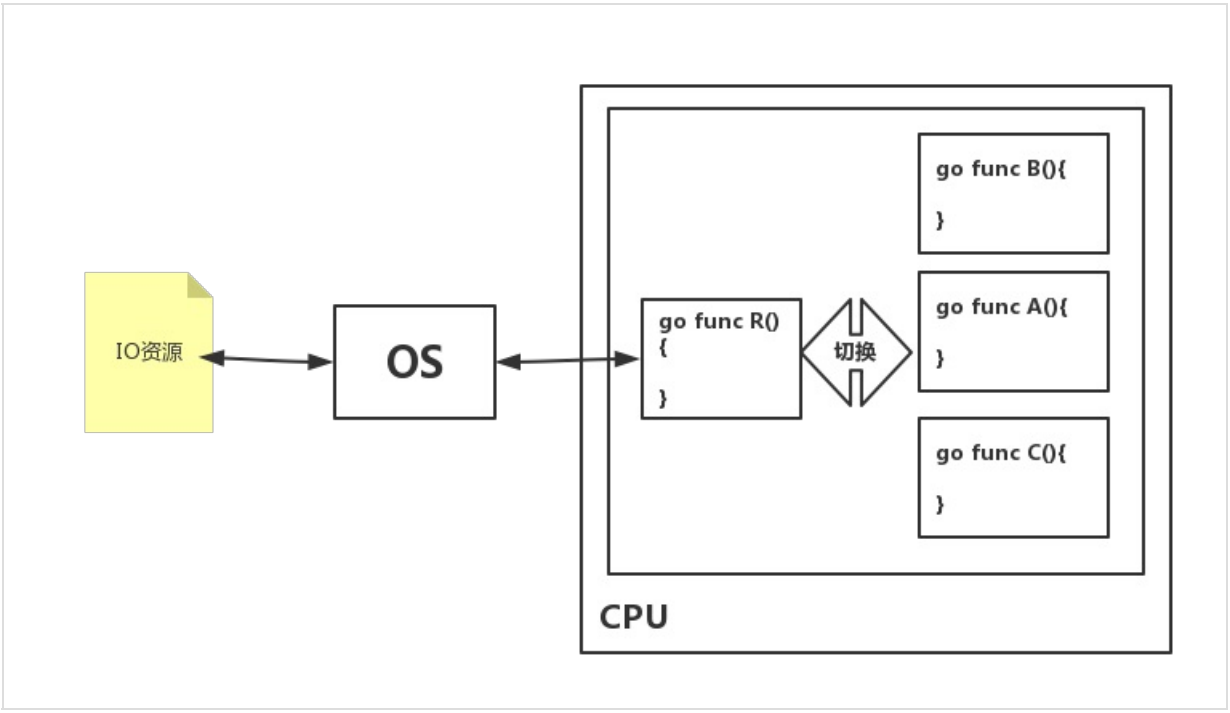
先看回调模型，像下图一样，左边是IO队列，右边是代码片段，每个IO事件对应一个回调函数，接收到IO事件以后进行相应的函数处理，这样的好处就是CPU利用率极高，不用切换资源，坏处就是代码被扯乱了，变成无序的了，对编程的要求比较高。




再看看线程模型，线程模型线程的切换实际上是操作系统来进行的，线程模型如下图，右边是线程的代码，左边的舞台就是CPU了，当在CPU上跳舞的线程进行系统调用（比如读取文件）或者每隔一段时间（时钟中断，不了解的请自行看计算机体系结构），操作系统就会把当前在CPU上玩耍的线程换下来，换个新的上去，这样的调度方式是操作系统来进行的，不需要线程本身参与，线程什么时候运行完全有操作系统说了算，线程切换也是操作系统说了算，好处就是编程的时候比较正常，缺点就是进行线程的切换是要耗费资源的，而且新开线程也需要资源。



有什么办法把这两种模型结合起来呢？我觉得golang的协程就是干了这个事情，golang的协程模型如下，这个图是我自己画的，主要是为了说明协程把上面两个模型结合，实际的协程模型长得不是这样子的啊。我们看到只剩下一个线程（或者进程）在CPU上跑了，上面的任务都跑到线程内部变成一个一个的代码片段了，执行哪个片段由线程内部决定，当遇到系统调用的时候，不用切换进程，而是直接切换执行其他的代码片段，这样的话，和线程模型比少了线程切换的开销，并且还能和回调模型一样，当IO操作的时候运行其他代码片段，最重要的是没有回调函数了，在开发人员看来和线程一样了。




好了，关键问题来了，怎么调度的呢？上面的回调模型和线程模型调度的时候都是操作系统来完成的，这里就显示出`代码内部`的`显示调用`了，就是说这些代码片段运行的时候，运行一段时间后，通过调用一个函数，主动放弃CPU，这些个代码片段就像下面这样

```
func running(){
    //计算一些东西
    stop() //调用stop主动不跑了，请把我正在运行的这个地址和我的栈记录下来，下次运行的时候继续在这里跑
    //剩下的代码
}
```

这就是代码内部的显示调用了，所谓`协程`嘛，就是需要运行的`代码`来`协助`进行`任务`的调度，怎么协助呢？就是显示的调用一个函数来进行现场保存和切换代码。

很多人说我在写golang的时候没看到这玩意啊，恩，为了让你编程容易，这东西隐藏到系统调用中去了，就是说你在协程中只要进行了系统调用(比如打印系统，读取文件，操作网络)，那么在调用类似fmt.Println的时候就调用了这个调度函数来切换协程了，当然，你也可以在你的代码中主动放弃CPU使用权，只要调用`runtime.Gosched()`就行了。

关于这部分，你可以试试下面的代码，在单核CPU的情况下，第二个函数是永远执行不了的，如果是按照多线程的思想，第二个函数是可以执行的，这就是因为第一个函数没有系统调用没有IO操作，所以一直把持着CPU不放弃，这也是协程编程需要注意的地方。

```
func main() {
    runtime.GOMAXPROCS(1)
    var workResultLock sync.WaitGroup
    go func() {
        fmt.Println("我开始跑了哦。。。")
        i := 1
        for {
            i++
        }
    }()
    workResultLock.Add(1)
    time.Sleep(time.Second * 2)
    go func() {
        fmt.Println("我还有机会吗？？？？")
    }()
    workResultLock.Add(1)
    workResultLock.Wait()
}
```

好了，协程的原理说了一下，我们再看看到底什么模型比较合适呢？我们看到，golang的协程这么设计出来，首先建立协程的消耗很少，并且在多IO操

作的时候比线程是要占优势的，因为在IO操作的时候，只是像回调一样换了一段代码来执行，没有线程的切换。这也是为什么用golang来写服务器代码比较合适的原因，因为服务端的代码基本上都少不了IO操作，网络读写是IO，数据库读写是IO，这样用golang既可以保持原来的多线程编程的连贯思维，又可以尽可能的使用事件模型的优势，减少线程切换。

恩，现在回到我们的缓存，虽然我们在对缓存读写的时候没有IO操作，但是网络读写还是IO操作，而且对于缓存的操作本身理论上并不耗费多少时间（就是几个哈希操作），所以IO时间占比还是比较大的，所以这种情况下我觉得使用奔放的协程模式是可以的，但也别太奔放了，最好限制一下协程的数量。

但对于有些系统，比如搜索系统，广告系统这种服务，每次都有个在线排序的过程，这是个非常大计算量的任务，基本上一次请求80%的时间都耗费在排序这种计算上了，IO反而不是瓶颈，这种情况下，多线程模型和golang这种协程模型差别就不是很大了，这时候8核CPU只启动8个协程和启80个协程，效率的差别就不大了。

## 总结

代码没准备好，忙死了。不过放心不会太监的。而且好久没更新了，今天先把文章写了吧，本来没准备写golang协程这一块的，后来写着写着就有这一段了。

如果你觉得不错，欢迎转发给更多人看到，也欢迎关注我的公众号，主要聊聊搜索，推荐，广告技术，还有瞎扯。。文章会在这里首先发出来：）扫描或者搜索微信号XJJ267或者搜索中文西加加语言就行

1 天前发布　更多

3 推荐　收藏

### 你可能感兴趣的文章

【分享】Web应用的缓存设计模式　2 收藏，508 浏览

略谈服务端缓存设计　4 收藏，681 浏览

缓存架构的理论分析　12 收藏，1.8k 浏览



### 讨论区

使用评论询问更多信息或提出修改意见，请不要在评论里回答问题

提交评论

评论支持部分 Markdown 语法：`**bold**` `_italic_` `[link](http://example.com)` `> 引用` `` `code` `` `- 列表`。
同时，被你 @ 的用户也会收到通知

本文隶属于专栏

## 吴说

一个老程序员说。。。。。

吴yh坚
作者

关注专栏

### 系列文章

后端的轮子（一） 14 收藏， 650 浏览

后端的轮子（二）--- 数据库 5 收藏， 391 浏览

分享扩散：