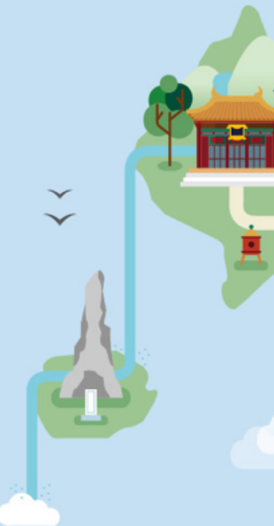
Go在证券行情系统中的应用

广发证券 刘楠



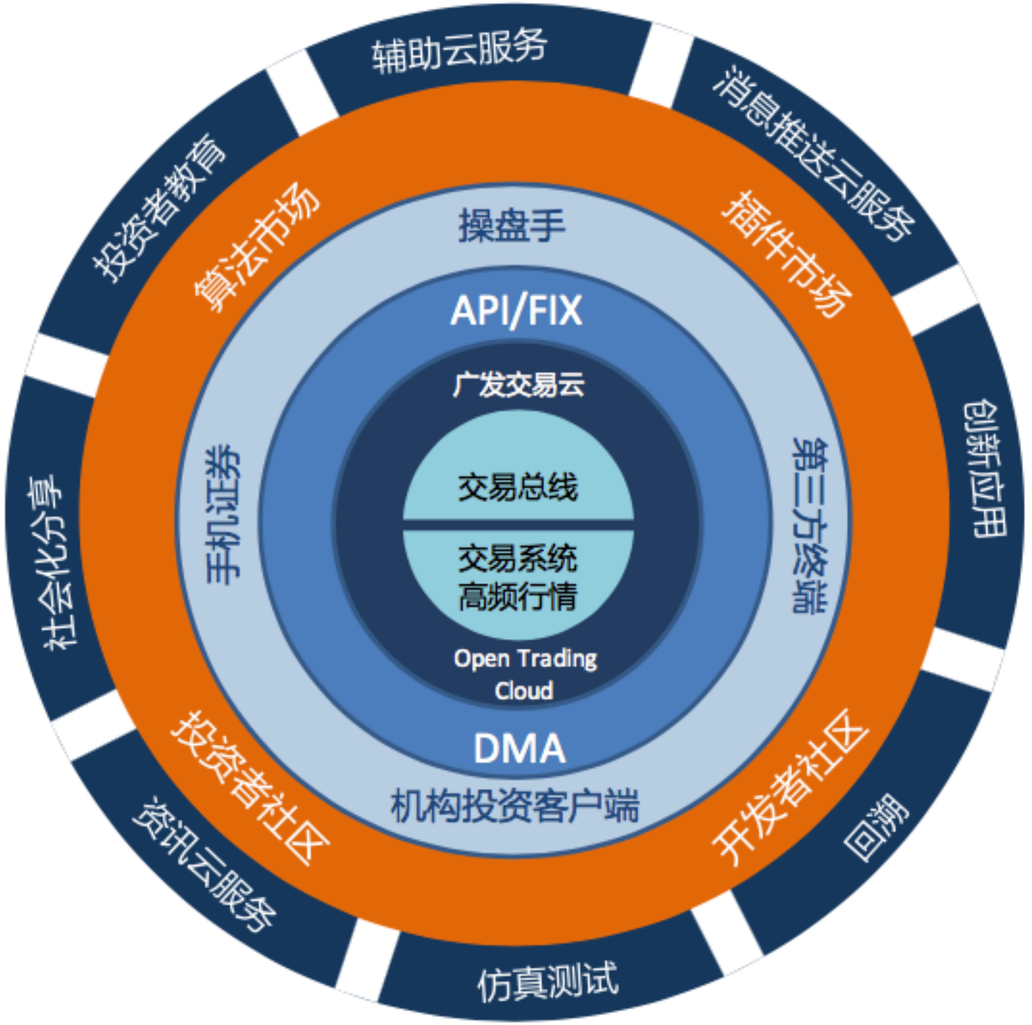
Agenda

- 证券行情系统背景介绍
- 证券行情业务特点
- 行情系统开发遇到的挑战

证券行情系统背景介绍

以**行情云**和**交易云**为核心，广发证券构建了OpenTrading交易平台、GF Quant量化分析平台、各类交易终端、开发者社区等FinTech生态系统，从理念到技术水平均走在业内前沿。

高频行情信息包括实时报价、逐笔成交、分时成交、周期K线、资金流向等信息数据，券商通过专线线路从交易所获取原始数据后通过计算生成。



证券行情系统的特点

* 超低延迟：　延迟过大会误导投资决策，导致客户流失

* 超高并发：　牛市时全民炒股刷行情数据

* 超高可靠性：数据出错可导致真金白银的损失

* 超严格监管：全面监管从严监管的时代受到各种合规约束

2015年5月29日，招商证券、东兴证券、齐鲁证券、国泰君安等证券公司信息系统发生中断或缓慢，引起各方广泛关注。——证券日报《证监部门处罚部分信息系统瘫痪券商》

行情开发遇到的挑战

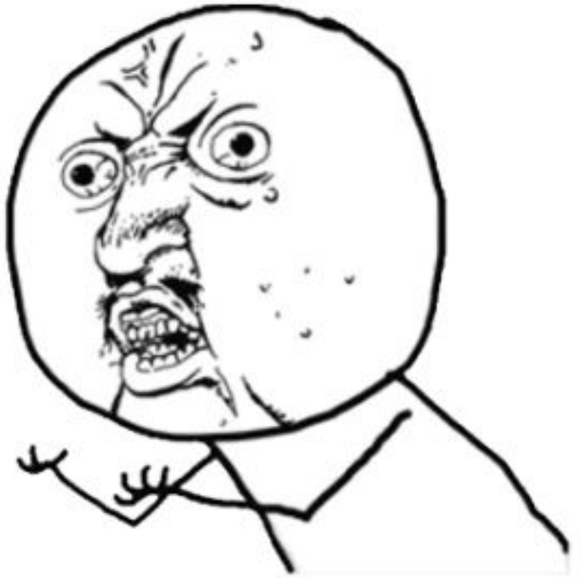
1 开发语言的选择

2 GC问题的困扰

3 面向并发的数据结构

4 融合替代方案

5 网络底层优化



1 开发语言的选择问题

C/C++: 历史悠久的高性能系统级语言，类似于AE86，1970s的设计理念，车重不足1吨，适合爱改装和造轮子的老司机

Java: 金融机构广泛使用的安全可靠系统级语言，类似于T99主战坦克，诞生于1990s，车重50吨，火力猛装甲厚行动迟缓安全性高

Golang: 为并发而生集成现代设计理念的系统级语言，类似于Tesla Model S，诞生于近几年，有AutoPilot等ADAS功能，代表业界发展方向

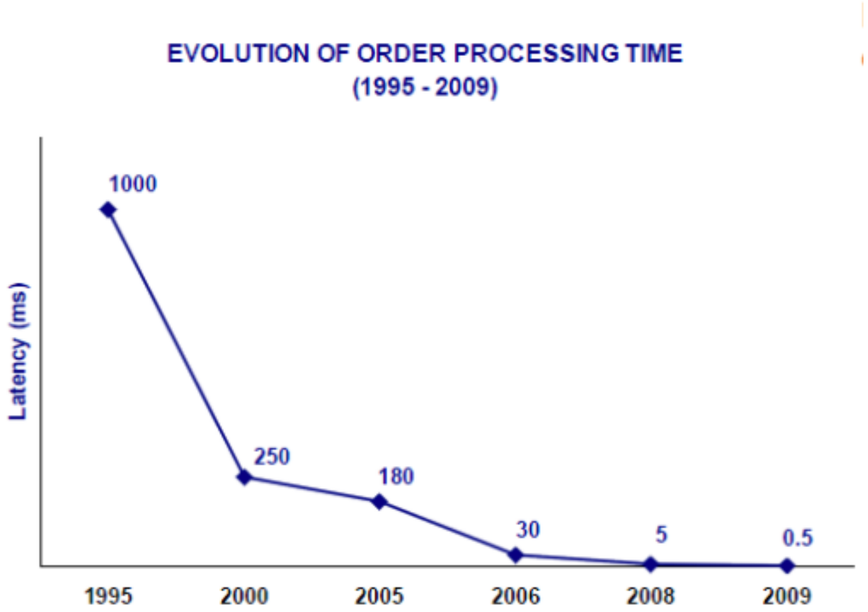
我们团队有着互联网技术基因，用前沿开源技术打造创新型 FinTech Startup

2 GC问题的困扰

海量并发和海量数据处理的系统，GC的内存对象扫描标记不仅消耗大量CPU资源，还会因为GC过程Stop The World造成毫秒级延时，拖慢行情推送速度。

行情与交易都是与时间赛跑的实时应用领域

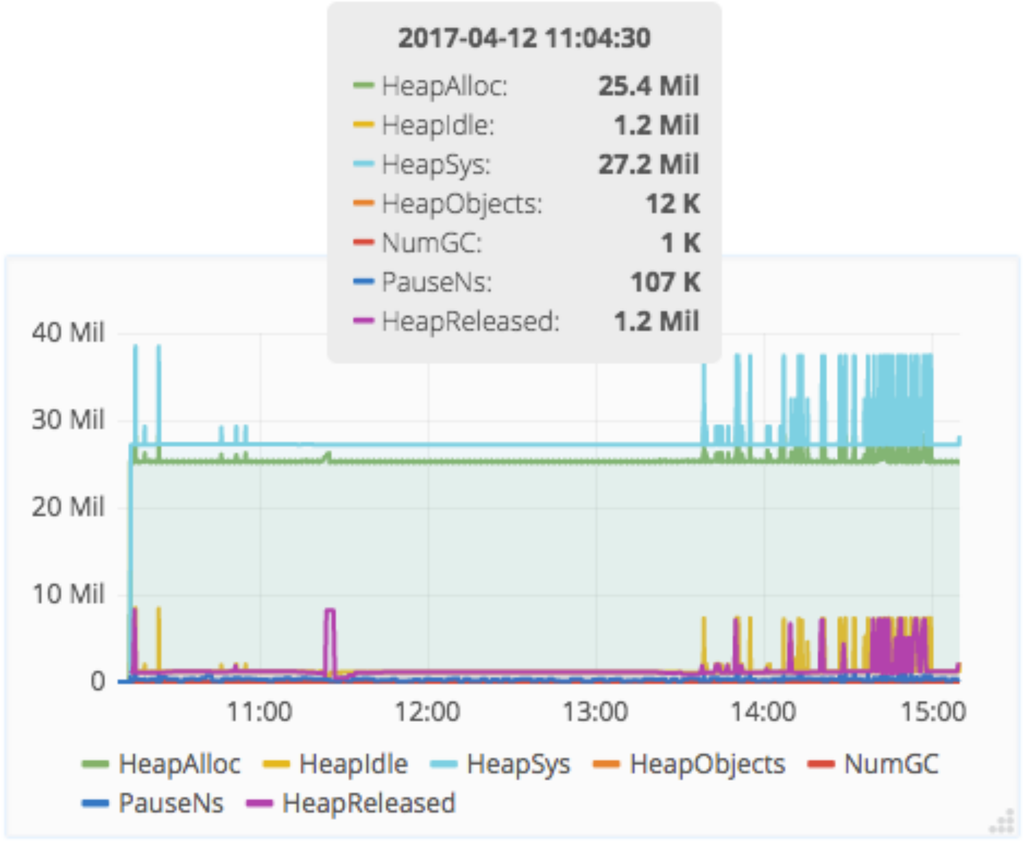
例如，为了把芝加哥期货市场和纽交所的通讯时间缩短3毫秒而花费数亿美元专门埋一条遇山开山遇河挖隧道只为了走直线的光纤，再如频繁花巨资更新通讯设备只为了几微秒的提速，乃至co-location到把机器并排放在证交所的服务器旁边。



2.1 Go在GC性能上的改进

- Go 1.8 Release Notes: 相比1.7版本GC Pause大幅减小，通常低于100微妙甚至10微秒

- Go 使用CMS(Concurrent Mark Sweep) GC算法，优点是不中断业务的情况下并行执行，将STW时间降低到最小，缺点是并行执行需要更多的同步开销降低了吞吐量，以及堆空间的增长难以预测。



2.2 GC算法考量的因素

1 并发：回收器利用多核机器并行执行

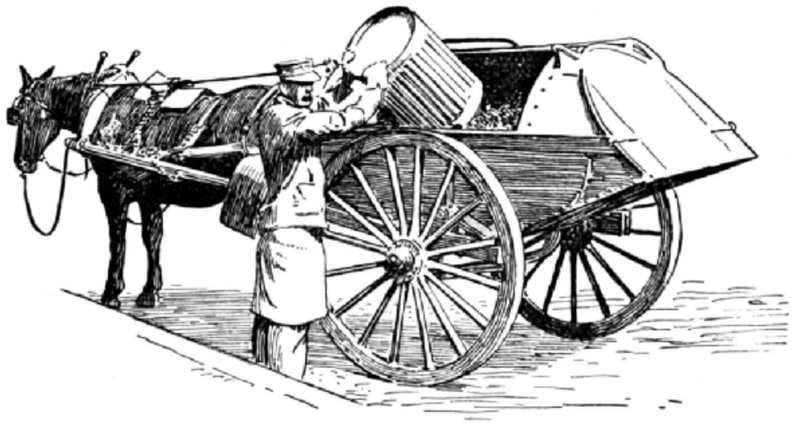
2 停顿时间：回收器会造成多长时间的停顿

3 停顿频率：回收器造成的停顿频率分布

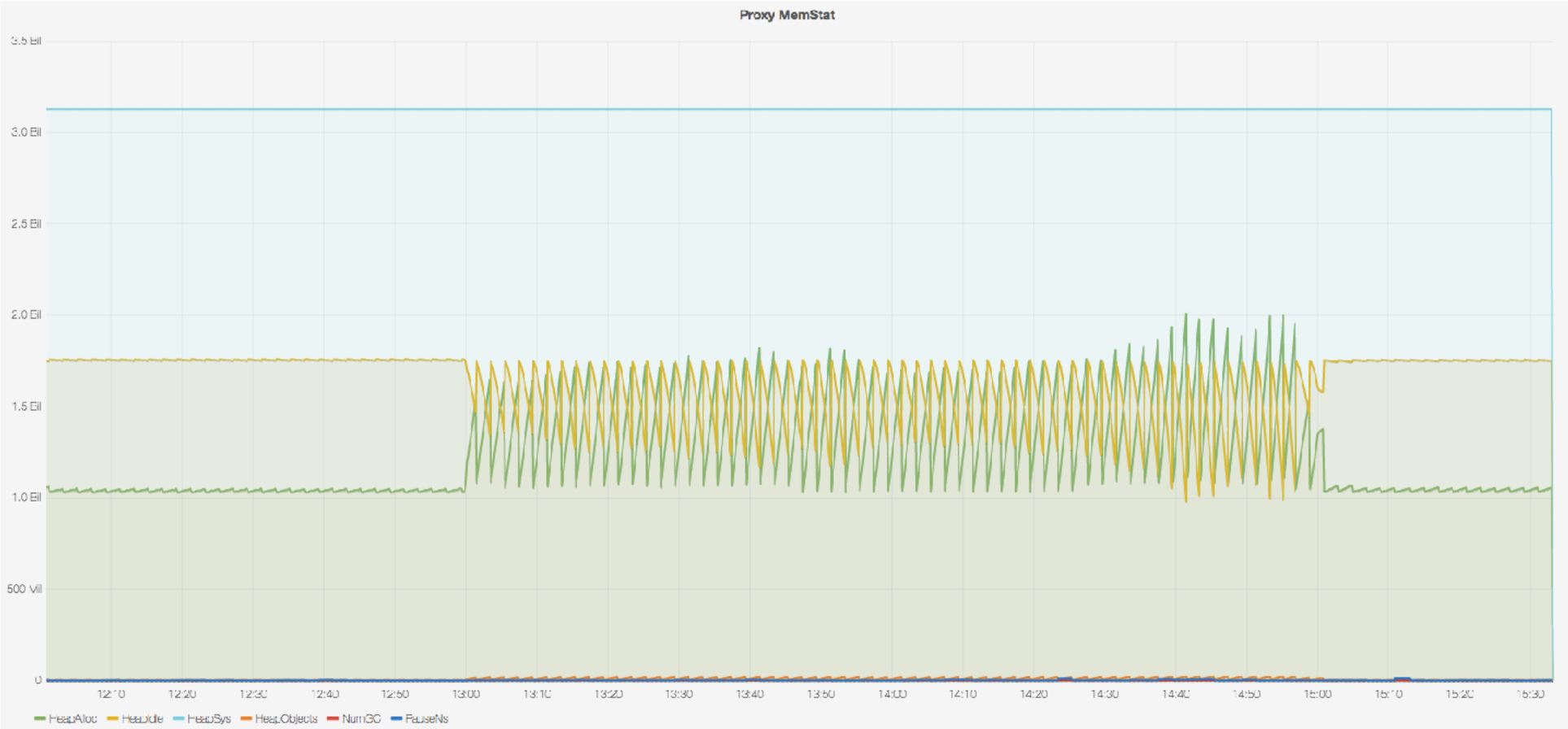
4 压缩：移动内存对象，整理内存碎片

5 堆内存开销：回收器最少需要多少额外的内存开销

6 GC吞吐量：在给定的CPU时间内，回收器可回收垃圾数量



堆空间的暴涨



无压缩？吞吐量不足？无停顿处理不及时？并发执行不可预测？

2.3 避免Goroutine的频繁创建销毁

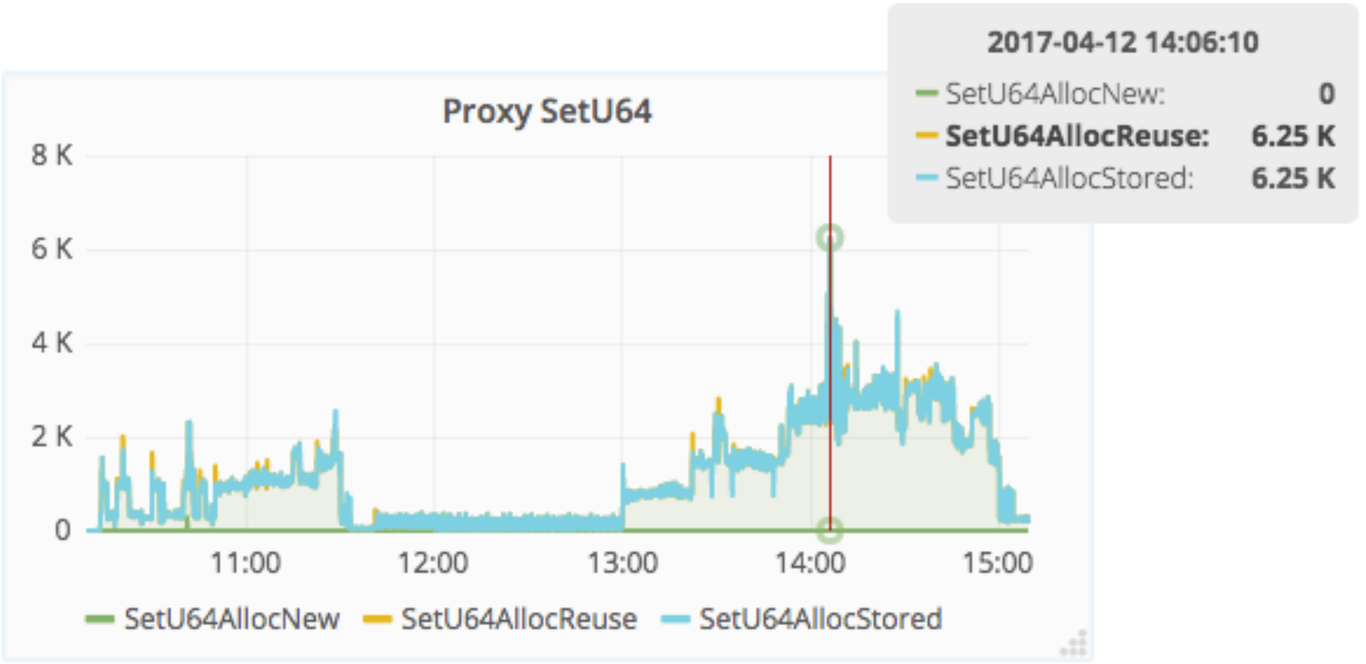
- 并发量小于1000时：每个请求分配一个Goroutine，并发模型简单易于开发，类似于Apache并发模型
- 并发量大于1000时：频繁创建的Goroutine在销毁时产生大量内存垃圾，GC过程拖慢系统响应速度，宜采用Nginx并发模型

2.4 对象缓存池的使用

- 不创建新对象才能避免GC
- 对象的创建速度和销毁速度近似平衡
- sync.Pool无法控制缓存对象数量和销毁时机
- 造自己的轮子

对象缓存池的简单实现

```
var (
        backendSetU64Chan = make(chan *SetU64, 10000)
)

func AllocSetU64() *SetU64 {
        var set *SetU64
        select {
        case set = <-backendSetU64Chan:
                stat.SetAttr("Proxy.SetU64AllocReuse", 1)
        default:
                t := make(SetU64)
                set = &t
                stat.SetAttr("Proxy.SetU64AllocNew", 1)
        }

        return set
}

func FreeSetU64(set *SetU64) {
        for key, _ := range *set {
                delete(*set, key)
        }

        select {
        case backendSetU64Chan <- set:
                stat.SetAttr("Proxy.SetU64FreeStored", 1)
        default:
                stat.SetAttr("Proxy.SetU64FreeDroped", 1)
        }
}
```

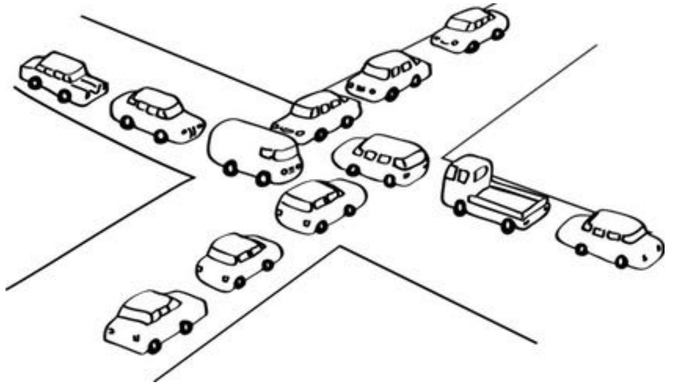


2.5 栈对象和堆对象

- 栈对象在函数返回时释放，堆对象由GC释放
- Go编译器的做法：不逃逸的对象放栈上，可能逃逸的放堆上
- 尽量使用栈对象，特别是在快速调用和返回的函数中，栈对象的分配速度比堆对象快一倍
- 长时间不返回的函数中，过多的栈对象可能增加Goroutine调度开销
- go tool compile -m 辅助分析对象的分配情况

3 面向并发的数据结构

- 多线程时代并发访问临界区资源时往往要加锁，锁的存在使得并行任务互相干扰影响性能
- 多处理器多核时代并发问题更加严重，同一内存单元读写也会互相干扰影响性能



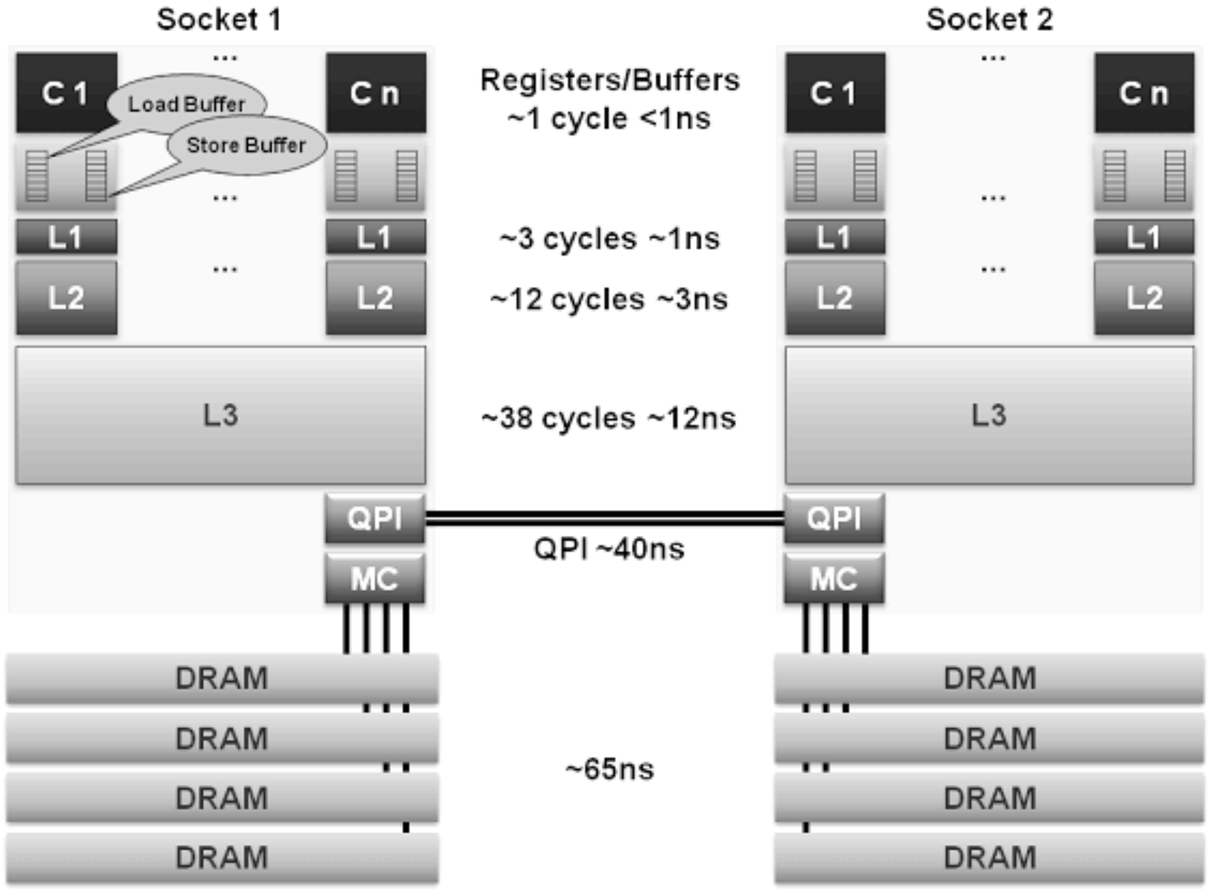
3.1内部存储器层次结构

Cache Miss的代价：

内存的延迟往往是很高的，从10到100纳秒不等。一个3.0GHz的CPU在100ns可以处理多达1200条指令。一次缓存失效约等于失去执行500 CPU指令机会。

Cache 一致性协议：

缓存段处于独占或已修改状态时才可修改，否则通过总线请求独占权，通知其他处理器失效同一缓存段的拷贝。



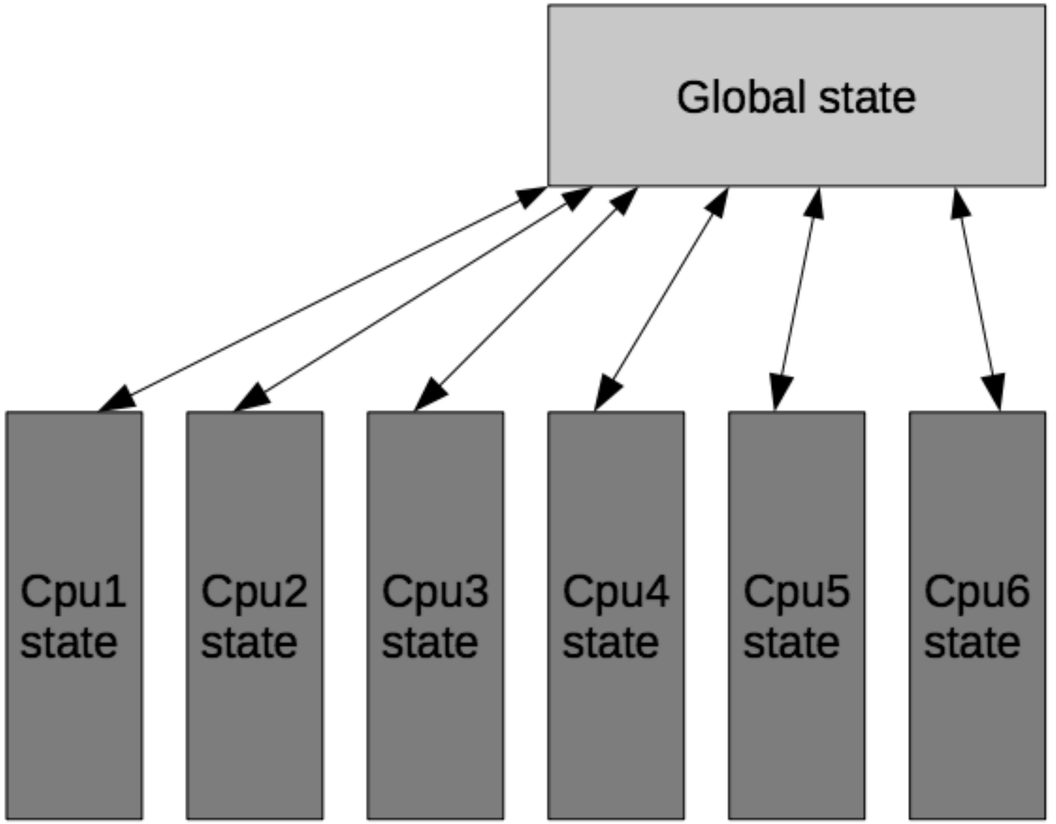
3.2 Per-CPU storage

- 统计变量之间的干扰

```
stat.SetAttr("Proxy.ManagerMsg", 1)
```

- 并行读锁之间的干扰

```
// RLock locks rw for reading.
func (rw *RWMutex) RLock() {
        if race.Enabled {
                _ = rw.w.state
                race.Disable()
        }
        if atomic.AddInt32(&rw.readerCount, 1) < 0 {
                // A writer is pending, wait for it.
                runtime_Semacquire(&rw.readerSem)
        }
        if race.Enabled {
                race.Enable()
                race.Acquire(unsafe.Pointer(&rw.readerSem))
        }
}
```

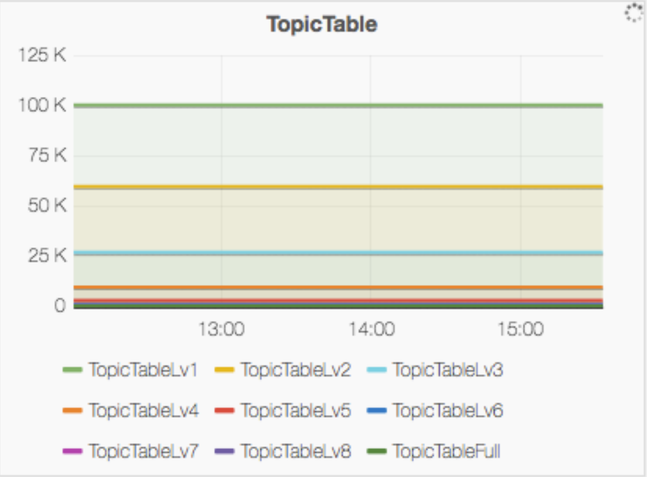


3.3 支持并发访问的Map

- 多级Hash Map是一种容量固定的数据结构，将并发访问锁的粒度细化到每个Map存储元组

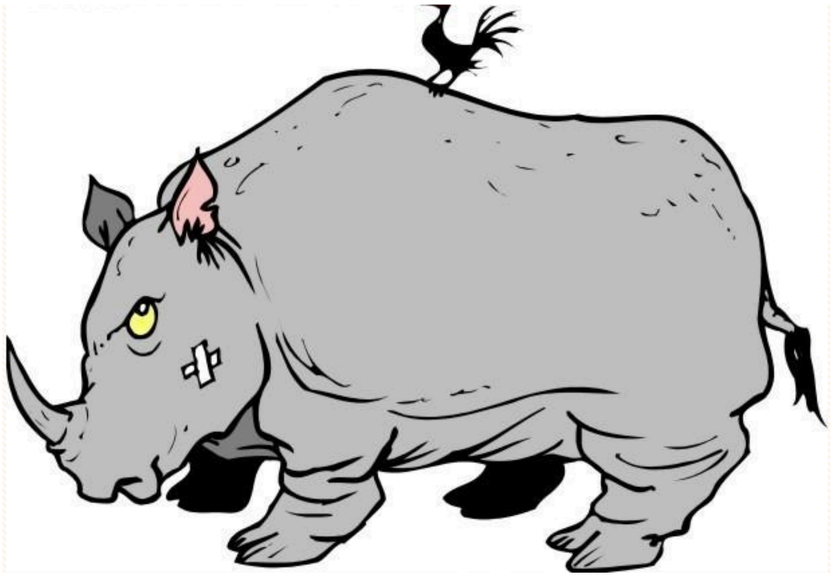
```
func (thiz *TopicTable) Get(key uint64) *TopicInfo {
        hash := key % thiz.basePrime
        target := &thiz.buffer[hash]

        target.mu.Lock()
        if target.key == key {
                return target
        }
        target.mu.Unlock()

        thiz.modList[0] = key % thiz.primeList[0]
        thiz.modList[1] = key % thiz.primeList[1]
        thiz.modList[2] = key % thiz.primeList[2]

        var i uint64
        for i = 1; i < thiz.hashTimes; i++ {
                thiz.modList[i+2] = key % thiz.primeList[i+2]
                hash = (thiz.modList[i+2]*thiz.primeList[i+1]*thiz.primeList[i]*thiz.primeList[i-1] +
                        thiz.modList[i+1]*thiz.primeList[i]*thiz.primeList[i-1] +
                        thiz.modList[i]*thiz.primeList[i-1] +
                        thiz.modList[i-1]) % thiz.basePrime

                target = &thiz.buffer[thiz.hashLen*i+hash]
                target.mu.Lock()
                if target.key == key {
                        return target
                }
                target.mu.Unlock()
        }
        return nil
}
```

	Hash 1	Hash 2	Hash 3	Hash 3	...	Hash k
Level 1						
Level 2						
Level 3						
Level 4						
Level 5						
Level 6						
...						
Level N						



4 融合替代方案

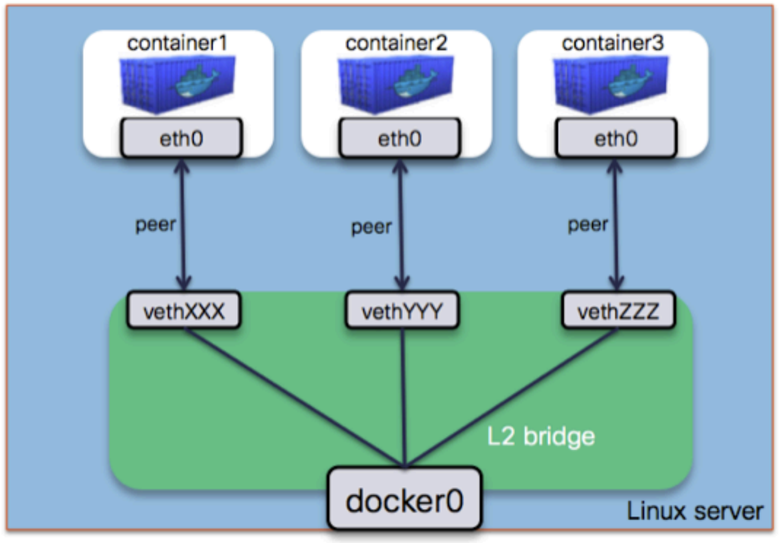
- 第三方库产生大量垃圾对象
- Goroutine难以管理10万级以上连接
- Cgo调度性能不高
- ……



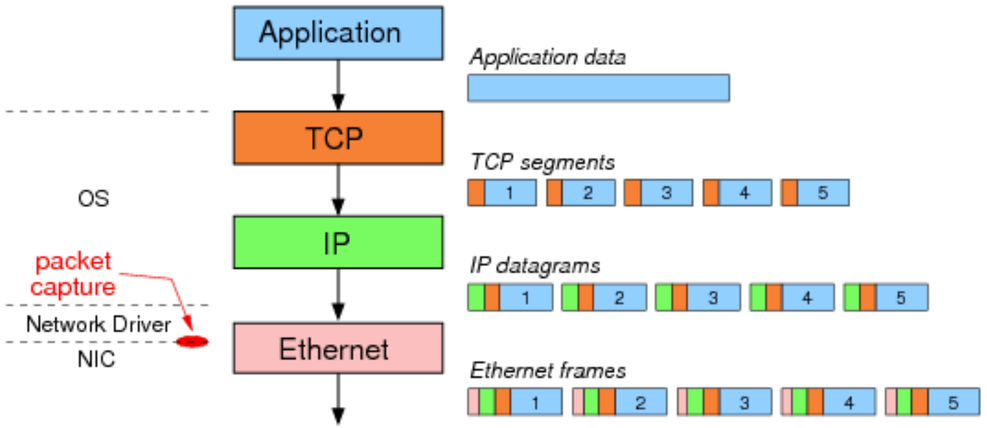
以上问题的解决方案：Cgo线程常驻内存，通过内存与Go通信

5 网络底层优化

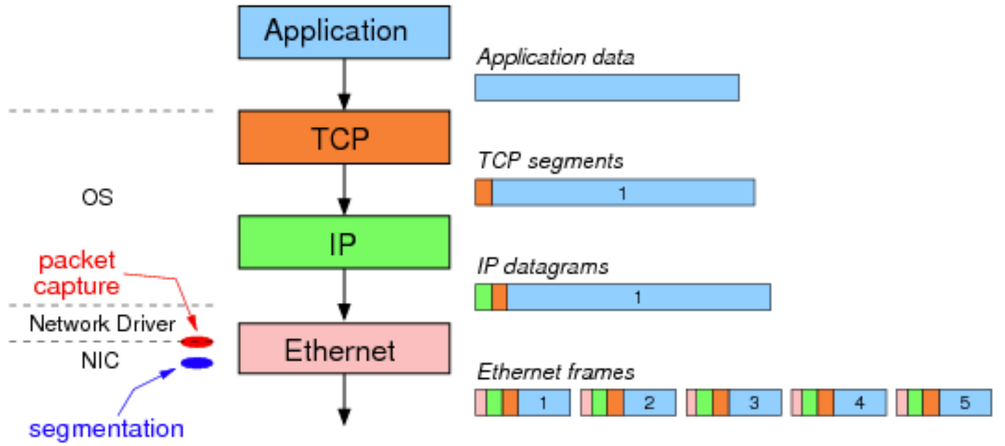
- Docker等虚拟网络可以调大MTU使用巨型帧传输数据，应用层一次系统调用可以传输更多数据



- 跨主机通信可以利用网卡硬件的分片offload和校验offload功能



开启前



开启后

大包无法正常接收的问题

- 数据长度为1493的UDP数据包校验错误

```
 23 7.916744    172.17.0.34     10.2.130.15     UDP      1537 8080 → 8080  Len=1493 [UDP CHECKSUM INCORRECT]
 24 7.916744    172.17.0.34     10.2.130.15     UDP      1537 8080 → 8080  Len=1493 [UDP CHECKSUM INCORRECT]
 25 7.916780    10.2.130.16     10.2.130.15     IPv4     1516 Fragmented IP protocol (proto=UDP 17, off=0, ID=8dc5) [Reassembled in #26]
 26 7.916792    10.2.130.16     10.2.130.15     UDP        57 8080 → 8080  Len=1493 [UDP CHECKSUM INCORRECT]
```

- Docker容器网卡信息和Docker host网卡信息

```
root@fe95266d57cb:/# ethtool -i eth0
driver: veth
version: 1.0
```

```
root@fe95266d57cb:/# ethtool -k eth0
Features for eth0:
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: off [fixed]
        tx-checksum-ip-generic: on
        tx-checksum-ipv6: off [fixed]
        tx-checksum-fcoe-crc: off [fixed]
        tx-checksum-sctp: off [fixed]
scatter-gather: on
        tx-scatter-gather: on
        tx-scatter-gather-fraglist: on
tcp-segmentation-offload: on
        tx-tcp-segmentation: on
        tx-tcp-ecn-segmentation: on
        tx-tcp6-segmentation: on
udp-fragmentation-offload: on
```

```
root@st1:~# ethtool -i eth0
driver: vmxnet3
version: 1.2.0.0-k-NAPI
```

```
root@st1:~# ethtool -k eth0
Features for eth0:
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: off [fixed]
        tx-checksum-ip-generic: on
        tx-checksum-ipv6: off [fixed]
        tx-checksum-fcoe-crc: off [fixed]
        tx-checksum-sctp: off [fixed]
scatter-gather: on
        tx-scatter-gather: on
        tx-scatter-gather-fraglist: off [fixed]
tcp-segmentation-offload: on
        tx-tcp-segmentation: on
        tx-tcp-ecn-segmentation: off [fixed]
        tx-tcp6-segmentation: on
udp-fragmentation-offload: off [fixed]
```

- 同类问题报告

https://github.com/docker/docker/issues/18776

evanj commented on 19 Dec 2015

veth devices mark all packets as "checksums known good." This is wrong if packets came from hardware. If corrupt packets are routed from "real" devices to veth devices, like Docker does for IPv6, applications can receive corrupt data. Docker's standard IPv4 configurations, which use NAT to route packets, are not affected by this.

Workaround: Docker should be disabling rx checksum offloading on veth devices inside containers to avoid this bug, for all configurations. Even if typical IPv4 configurations are not affected, since many people use Docker containers with other networking configurations, this will protect all of them.

We were affected by this bug at Twitter, so I've been searching the Internet to determine what else is affected. More Details are in this Kubernetes bug report (which is also affected).

Kubernetes bug: kubernetes/kubernetes#18898
Mesos bug: https://issues.apache.org/jira/browse/MESOS-4105
Linux kernel patch: http://thread.gmane.org/gmane.linux.kernel/2111961

drivers/net/veth.c

```
303 #define VETH_FEATURES (NETIF_F_SG | NETIF_F_FRAGLIST | NETIF_F_HW_CSUM | \
304                        NETIF_F_RXCSUM | NETIF_F_SCTP_CRC | NETIF_F_HIGHDMA | \
305                        NETIF_F_GSO_SOFTWARE | NETIF_F_GSO_ENCAP_ALL | \
306                        NETIF_F_HW_VLAN_CTAG_TX | NETIF_F_HW_VLAN_CTAG_RX | \
307                        NETIF_F_HW_VLAN_STAG_TX | NETIF_F_HW_VLAN_STAG_RX )
308
309 static void veth_setup(struct net_device *dev)
310 {
311         ether_setup(dev);
312
313         dev->priv_flags &= ~IFF_TX_SKB_SHARING;
314         dev->priv_flags |= IFF_LIVE_ADDR_CHANGE;
315         dev->priv_flags |= IFF_NO_QUEUE;
316         dev->priv_flags |= IFF_PHONY_HEADROOM;
317
318         dev->netdev_ops = &veth_netdev_ops;
319         dev->ethtool_ops = &veth_ethtool_ops;
320         dev->features |= NETIF_F_LLTX;
321         dev->features |= VETH_FEATURES;
322         dev->vlan_features = dev->features &
323                              ~(NETIF_F_HW_VLAN_CTAG_TX |
324                                NETIF_F_HW_VLAN_STAG_TX |
325                                NETIF_F_HW_VLAN_CTAG_RX |
326                                NETIF_F_HW_VLAN_STAG_RX);
327         dev->destructor = veth_dev_free;
328         dev->max_mtu = ETH_MAX_MTU;
329
330         dev->hw_features = VETH_FEATURES;
331         dev->hw_enc_features = VETH_FEATURES;
332         dev->mpls_features = NETIF_F_HW_CSUM | NETIF_F_GSO_SOFTWARE;
333 }
```

Thank you!

nandyliu@outlook.com