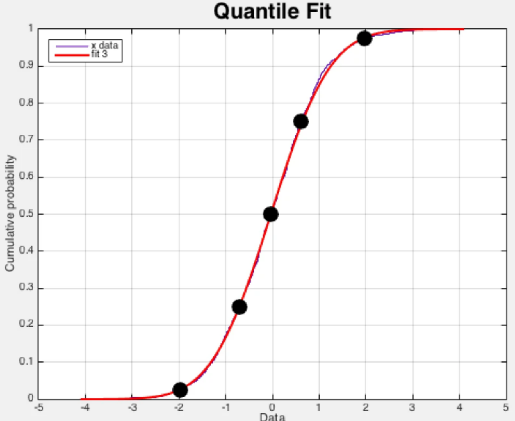
Summary指标说明

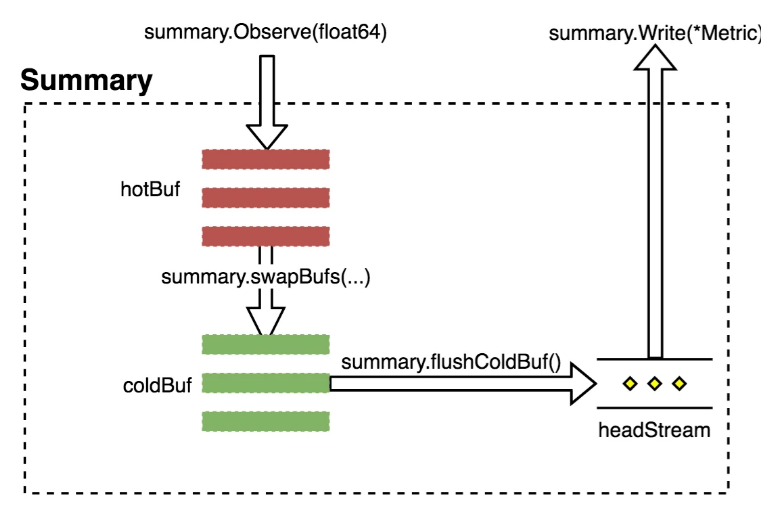
Summary 是标准数据结构中最复杂的一个，用来收集服从正态分布的采样数据，主要三种作用：

* 在客户端对于**一段时间内**（默认是10分钟）的每个采样点进行统计，并形成分位图；
* 统计所采样点总和（sum）；
* 统计所有采样点数量(count)

假设对某个Summary类型采样指标，设置分位列表（quantile）如下：[0.025 0.25 0.50 0.75 0.975]。在指定时间内采样数据形成的分布图可能如下，从采样结果统计出0.5分位点的采样数据为0，而0.975分位点的采样值为2。



关于quantile计算是一个资源消耗比较大（expensive）的操作。



根据社区官方实现，每Observer一个采样数据，都需要将其计算插入到排序后的采样点队列合适位置，并且还需要排除已经过期的采样点，并且由于指标特殊性，每次只能处理一组采样数据。因此为了避免等待阻塞流程，golang\_client每次Observe到一个Summary采样点，都会起个独立goroutine处理数据。在goroutine内先将采样数据加hotBuff队列。待hotBuff队列满或者队列到期，在将数据交换（swapBufs)到coldBuf。最后将coldBuf里的数据批量插入（flushColdBuf）到headStrem的计算队列（已经计算后的分布队列），再次计算插入后新的分布队列。因为headStream计算消耗非常大，为了解决Observe和headStrem计算的同步性问题，flushColdBuf处也增加了goroutine来解耦，因此实际上有两层goroutine。为了解决goroutine的并发冲突，也相应设置了两层锁结构，因此每次Observe或者flushCodBuf都必须先获取相应的锁，整体效率不高。

并且，在headStream的计算逻辑里，需要根据新merge值的不同，不断调整队列位置，压缩队列，存在频繁内存数据搬移（memmov）操作，这种资源消也非常高大的操作，因此计算quantile的代价是非常高的。在大规模Summary采样场景下，如果headStream处理不及时，会导致排队竞争锁的goroutine数量不断累计，内存消耗也会不断增加。

**参考资料：**

Histogram和Summary对比：https://prometheus.io/docs/practices/histograms/