浅析下开源监控解决方案Prometheus的存储机制

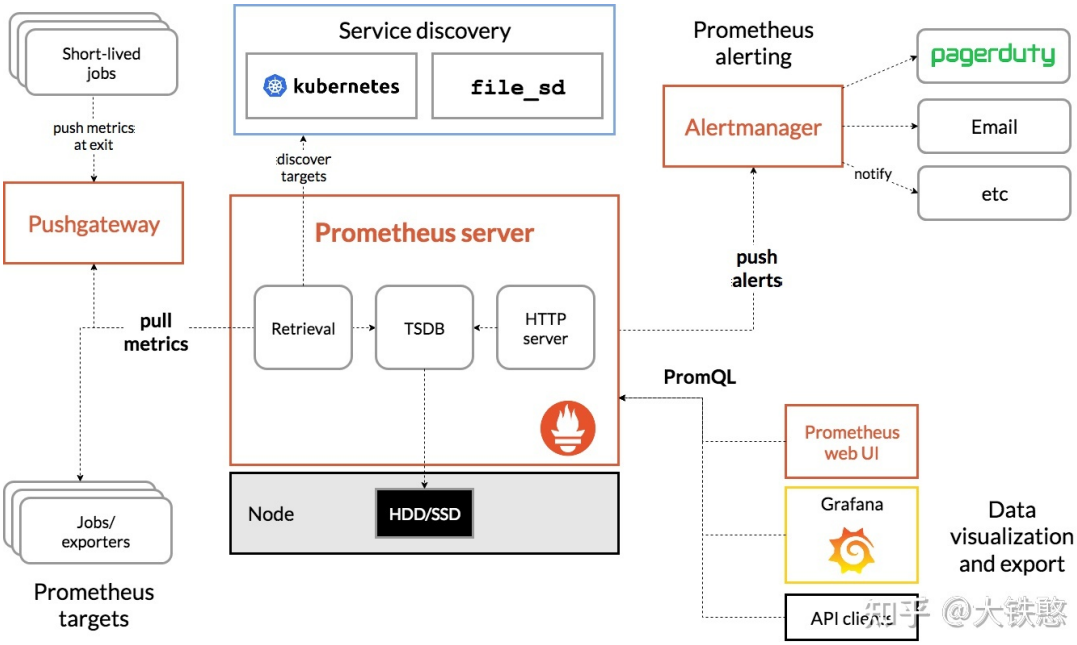
已关注

5 人赞同了该文章

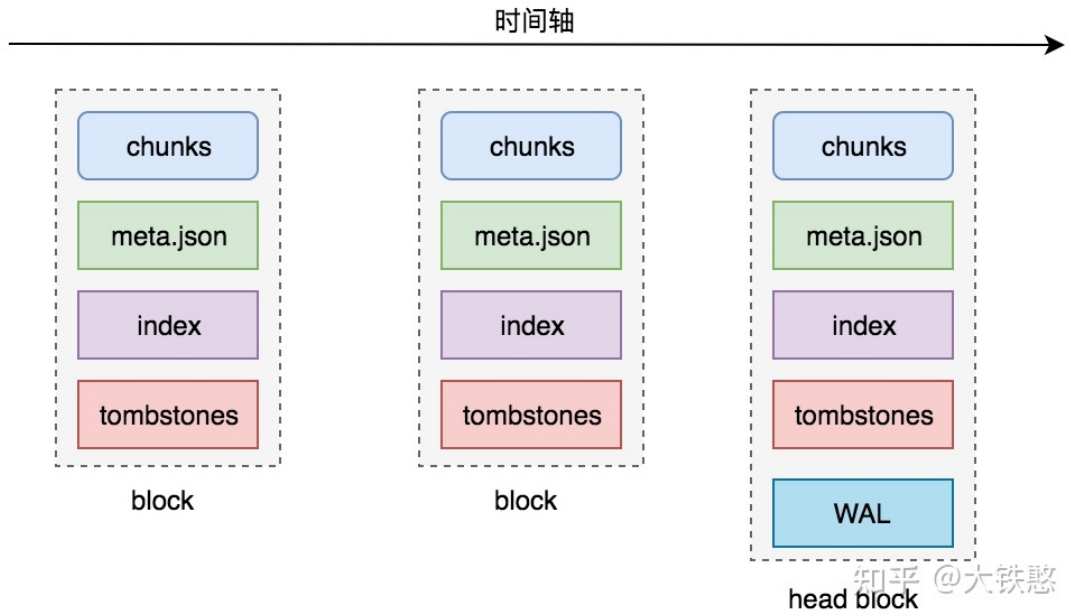
Prometheus是一个开源的监控和报警系统，依赖少，功能齐全，于2016年加入CNCF，广泛用于 Kubernetes集群的监控系统中，2018年8月成为继K8S之后第二个毕业的项目。基于Prometheus可以很轻松且快速的构建一套包含监控指标的抓取、存储、查询以及告警的完整监控系统。单个的Prometheus实例就能实现每秒上百万的采样，同时支持对于采集数据的快速查询。

整体架构

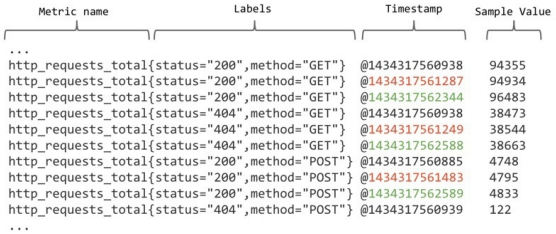
Prometheus的整体架构及其一些生态系统组件如下图所示，Prometheus的优秀特性的实现都离不开一个其内部设计优良的时序数据库TSDB的支持。这个TSDB整体上是一个LSM结构，包括WAL、memtable、文件等部分。



Prometheus本地存储TSDB的核心设计包括block和WAL两个，其中，block包含chunk，index，meta.json，tombstones等。Prometheus使用时间作为一级索引，将数据分为多个block，新创建的block的时间范围默认为2小时。同时会通过compaction异步将这些block以步长为3的方式进行合并，即将2小时的block合并为6小时、6小时的block合并成18小时这样的大block。



另外，Prometheus的写入模型，是单值模型， 如下图左边所示。Prometheus为了方便做字典映射（将metric + labels映射为一个整型id，以减少日志记录的大小）和构建倒排索引，Prometheus将metric做为一个特殊的标签，如下图右边所示。



WAL机制

日志编码

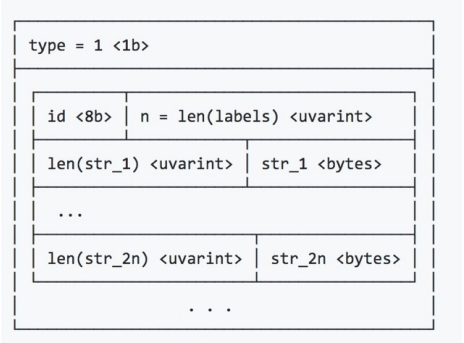
Prometheus的日志的类型有时间线记录（series record），样本记录（sample record）、墓碑记录（tombstone record）。

前面提到，Prometheus的写入模型由labels，timestamp，value等3个部分组成，其中labels唯一确定一条时间线。当一条时间线写入多个点时，相同的labels会出现多次，而且原始的labels占用的字节数通常要比 timestamp + value的占用的字节数多，如果原封不动的将上面的原始数据点（labels，timestamp，value）3元组写入WAL中，不仅磁盘IO开销大，而且因labels也会造成磁盘空间的极大浪费。

因此Prometheus为了减少WAL的磁盘IO，并提高吞吐，在WAL之前对写入的数据点做了预处理，将labels在内存中转换为一个整型seriesId，这个seriesId占用的大小会比较小，并将labels到id的映射写入WAL一次，后续写数据点到WAL中时，就只需要写（seriesId，timestamp，value）三元组了，而不是原先的（labels，timestamp，value）三元组如下图所示。 具体来说，就是对于每次写入，先根据labels在内存中查找是否存在已经映射好的seriesId，若不存在，则采用自增的方式创建一个seriesId，然后labels + seriesId 作为时间线记录（series record）只写入WAL一次，后续不在写入了。若存在，则直接将seriesId + value作为样本记录（sample record）即可。

时间线记录（series record）编码，时间线记录比较简单，开头记录类型，然后是映射出来的Id，最后是labels。[源码传送门]

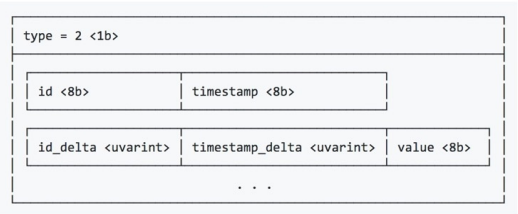
Series records




```go
// Series appends the encoded series to b and returns the resulting slice.
func (e *RecordEncoder) Series(series []RefSeries, b []byte) []byte {
        buf := encoding.Encbuf{B: b}
        buf.PutByte(byte(RecordSeries))

        for _, s := range series {
                buf.PutBE64(s.Ref)
                buf.PutUvarint(len(s.Labels))

                for _, l := range s.Labels {
                        buf.PutUvarintStr(l.Name)
                        buf.PutUvarintStr(l.Value)
                }
        }
        return buf.Get()
}
```

样本记录（sample record）编码，值得一提的是，在样本编码中，Prometheus使用了delta-encoding的方式编码了时间戳部分。[源码传送门]

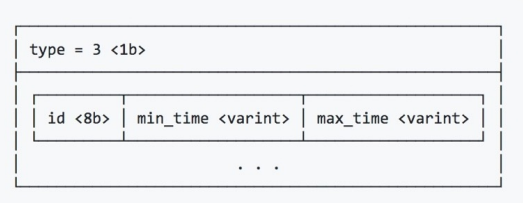
Sample records




```go
// Samples appends the encoded samples to b and returns the resulting slice.
func (e *RecordEncoder) Samples(samples []RefSample, b []byte) []byte {
        buf := encoding.Encbuf{B: b}
        buf.PutByte(byte(RecordSamples))

        if len(samples) == 0 {
                return buf.Get()
        }

        // Store base timestamp and base reference number of first sample.
        // All samples encode their timestamp and ref as delta to those.
        first := samples[0]

        buf.PutBE64(first.Ref)
        buf.PutBE64int64(first.T)

        for _, s := range samples {
                buf.PutVarint64(int64(s.Ref) - int64(first.Ref))
                buf.PutVarint64(s.T - first.T)
                buf.PutBE64(math.Float64bits(s.V))
        }
        return buf.Get()
}
```


墓碑记录（tombstone record）编码，墓碑记录比较简单，只记录了被删除的时间线的时间线id和删除的时间范围。

Tombstone records




```go
// Tombstones appends the encoded tombstones to b and returns the resulting slice.
func (e *RecordEncoder) Tombstones(tstones []Stone, b []byte) []byte {
        buf := encoding.Encbuf{B: b}
        buf.PutByte(byte(RecordTombstones))

        for _, s := range tstones {
                for _, iv := range s.intervals {
                        buf.PutBE64(s.ref)
                        buf.PutVarint64(iv.Mint)
                        buf.PutVarint64(iv.Maxt)
                }
        }
        return buf.Get()
}
```


日志刷盘

Prometheus的日志刷盘方式默认使用32KB页对齐的方式刷盘的，这个直接从LevelDB/RocksDB借过来的（传送门），每个日志的文件的大小默认为128MB，写满后，就滚动一个新的文件继续写入。日志文件的编号以000000、000001、000002的方式自增生成。

WAL page

The type flag has the following states:

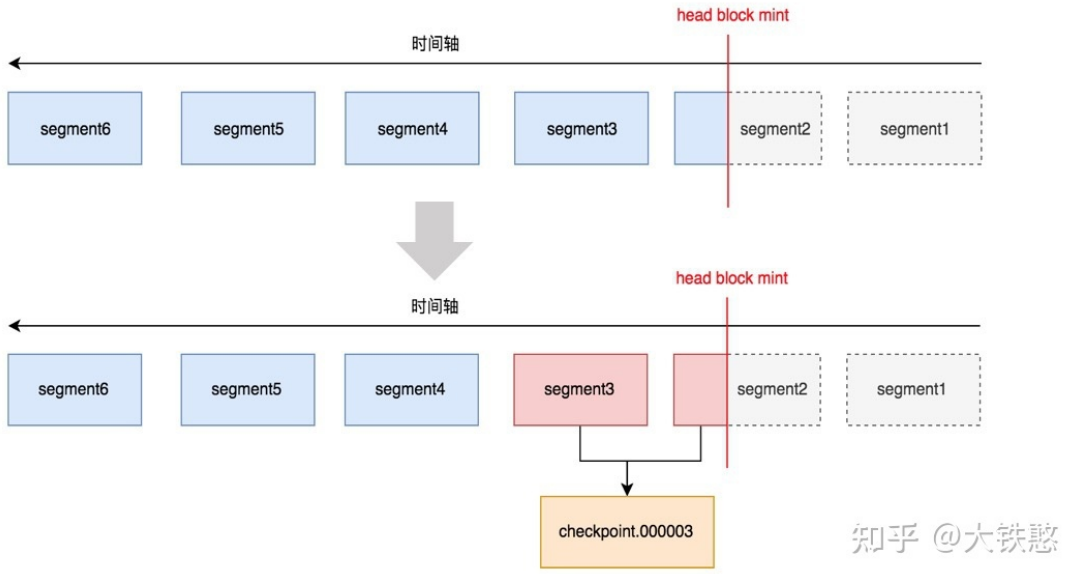
- `0` : rest of page will be empty
- `1` : a full record encoded in a single fragment
- `2` : first fragment of a record
- `3` : middle fragment of a record
- `4` : final fragment of a record

The write ahead log operates in segments that are numbered and sequential, e.g. 000000, 000001, 000002, etc., and are limited to 128MB by default. A segment is written to in pages of 32KB. Only the last page of the most recent segment may be partial.

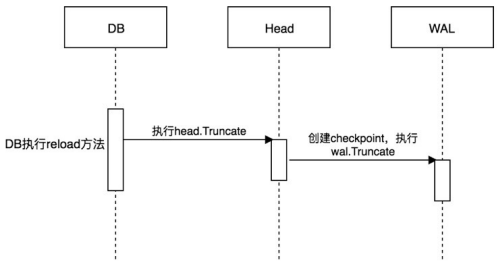
A WAL record is an opaque byte slice that gets split up into sub-records should it exceed the remaining space of the current page. Records are never split across segment boundaries. If a single record exceeds the default segment size, a segment with a larger size will be created.

日志删除

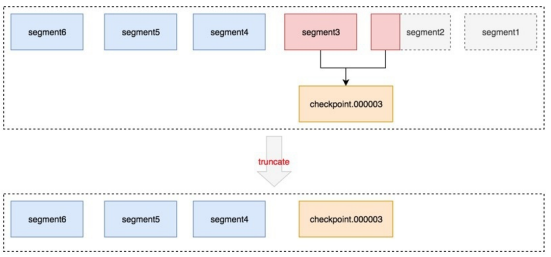
日志删除的时机有两个，一个是重启的时候，另一个是当内存中刷盘生成一个新的block时。前面提到，prometheus在写WAL时，将原先的写入拆成了时间线记录和样本记录，时间线记录只写入了一次，这个就导致了如果时间线记录关联的样本记录没有持久化到文件中时，保存这个时间线记录的日志文件是不能直接删除的。因此prometheus在删除日志文件时，需要根据内存中时间线映射关系和最小时间戳head block mint读取已有的segment日志文件的记录的record，然后将过滤后保留下来的记录生成checkpoint, 即一个或多个新的segment文件， 保存在chenkpoint.*文件中，然后删除已经读取过的segmet日志文件。如下图所示。生成checkpoint时默认不处理最新的3个segment文件，即不读取最新生成的3个日志文件。



另外，当生成checkpoint时，record中的data字段会根据情况使用snappy进行压缩，压缩标记记录在record的type字段中。日志删除流程和执行操作，如下图所示。



DB何时执行reload方法:
1.当启动时在Open方法中调用;
2.当compaction结束时调用;



Truncate流程：
1.修改Head的最小时间戳和最小合法时间戳;
2.调用head.gc()清理掉内存中过期的数据;
3.生成新的checkpoint;
4.删除过期的WAL;
5.删除过期的checkpoint;

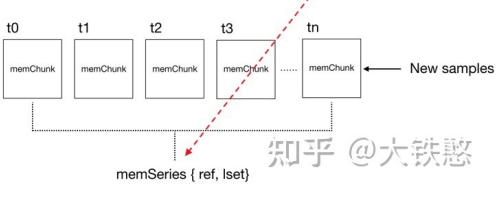
内存结构

Prometheus的内存结构核心是head这个数据结构，这个结构主要以下内容：

- minTime: 已经写入的数据点的最小时间戳；
- maxTime：已经写入的数据点的最大时间戳；
- minValidTime: 合法最小时间戳, 写入数据点的时间戳小于这个时间戳，Prometheus将直接丢弃，因为promtheus使用的拉模式采集的指标，时间戳是由exporter生成的，一般都在当前写入窗口期内。
- wal: 写日志的组件；
- lastSeriesID: 当前最新已经分配的Id的最大值；
- stripeSeries: 数据点存储结构
- symbols：符号表，存储labels到id的映射关系；
- postings：内存倒排索引；

```
// Head handles reads and writes of time series data within a time window.
type Head struct {
        // Keep all 64bit atomically accessed variables at the top of this st
        // See https://golang.org/pkg/sync/atomic/#pkg-note-BUG for more infc
        chunkRange       int64
        numSeries        uint64
        minTime, maxTime int64 // Current min and max of the samples includec
        minValidTime     int64 // Mint allowed to be added to the head. It sh
        lastSeriesID     uint64

        wal        *wal.WAL
        // All series addressable by their ID or hash.
        series *stripeSeries

        symbols map[string]struct{}
        values  map[string]stringset // label names to possible values

        deleted    map[uint64]int // Deleted series, and what WAL segment the

        postings *index.MemPostings // postings lists for terms
}
```

```
const stripSize = 1 << 14

// 为表达直观，已将Prometheus原生数据结构简化
type stripeSeries struct {
        series  [stripeSize]map[uint64]*memSeries
        hashes  [stripeSize]map[uint64][]*memSeries
        locks   [stripeSize]sync.RWMutex
}
```



其中，最核心的数据结构是stripeSeries中的memSeries，从上图中可以看出，一个memSeries主要由三部分组成：

- lset：用以识别这个series的label集合；
- ref：每接收到一个新的时间序列（即它的label集合与已有的时间序列都不同）Prometheus就会用一个唯一的整数标识它，如果有ref，我们就能轻易找到相应的series；
- * memChunks：每一个memChunk是一个时间段内该时间序列所有sample的集合。如果我们想要读取[tx, ty]（t1 < tx < t2, t2 < ty < t3）时间范围内该时间序列的数据，只需要对[t1, t3]范围内的两个memChunk的sample数据进行裁剪即可，从而提高了查询的效率。每当采集到新的sample，Prometheus就会用Gorilla算法将它压缩至最新的memChunk中；

当内存中的时间积赞到3.5个小时后，Prometheus会将前面最老的2个小时的数据进行刷盘。

磁盘文件

磁盘文件目录

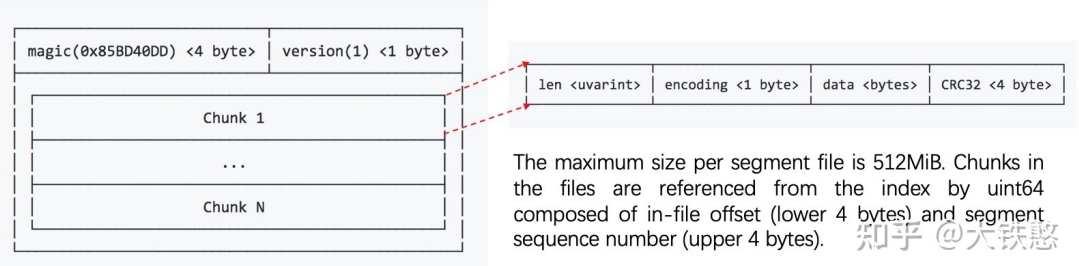
prometheus在磁盘的上文件最外层有一系列的block和一个wal目录组成，每个block下面由chunks目录，tombstones文件和index文件，以及meta.json文件组成。如下图所示。

```
├── 01BKGV7JBM69T2G1BGBGM6KB12
│   └── meta.json
├── 01BKGTZQ1SYQJTR4PB43C8PD98
│   ├── chunks  # 保存压缩后的时序数据，每个chunks大小为512M，超过会生成新的chunks
│   │   └── 000001
│   ├── tombstones # 数据进行软删除,因为TSDB删除数据会删除整个目录；可以用来删除一部分数据
│   ├── index # chunks中的偏移位置
│   └── meta.json #  记录block块元信息，比如 样本的起始时间、chunks数量和数据量大小等
├── 01BKGTZQ1HHWHV8FBJXW1Y3W0K
│   └── meta.json
├── 01BKGV7JC0RY8A6MACW02A2PJD
│   ├── chunks
│   │   |── 000001
│   │   └── 000002
│   ├── tombstones
│   ├── index
│   └── meta.json
└── wal
    ├── 00000002
    └── checkpoint.000001
        ├── 000001
        └── 000002
```

形如，“01BKGTZQ1HHWV8FBJXW1Y3W0K”的名称是block的标识，是一个长度为16字节字节数组的16进制表示，是一个ULID。ULID的总长度是128位（16字节），其中，前48位（6字节）为时间戳，后80位（10字节）为随机数。

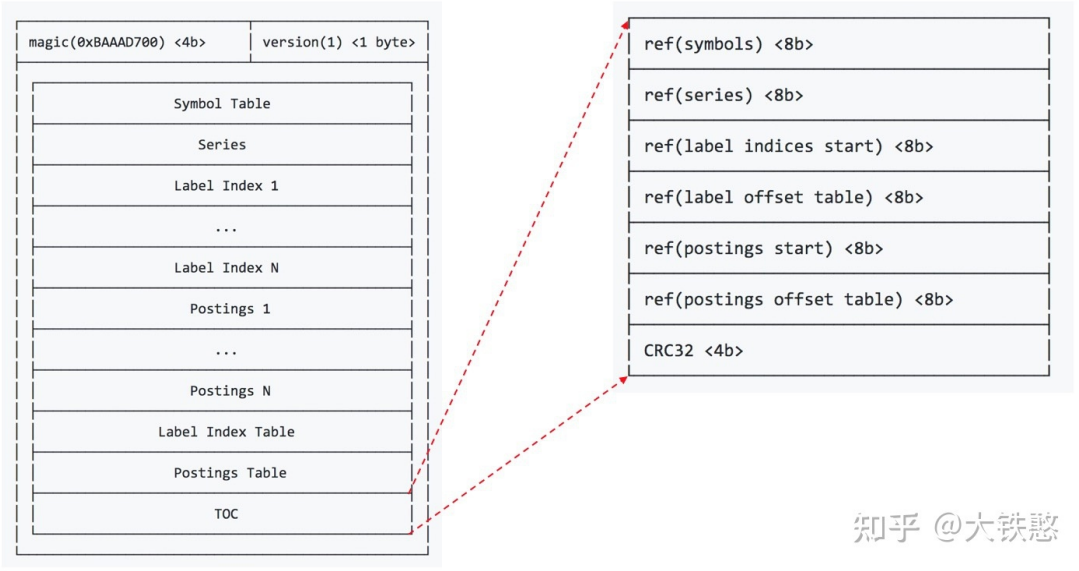
chunk文件

chunk文件的文件格式相对来说比较简单，只是一系列按照时间区间进行切分的压缩块chunk，每个chunk块在文件中的索引通过一个64位的无符号数来索引，高32位为chunk文件的文件名fileId，低32位为chunk块在文件内的偏移量offset。每个chunk文件的最大大小为512MB。



index文件

Prometheus的文件最复杂或者包含内容最多的莫过于index文件，这个index文件包含了符号表、倒排索引、chunk块的索引等信息。其文件格式整体设计如下图所示，官方文档传送门。

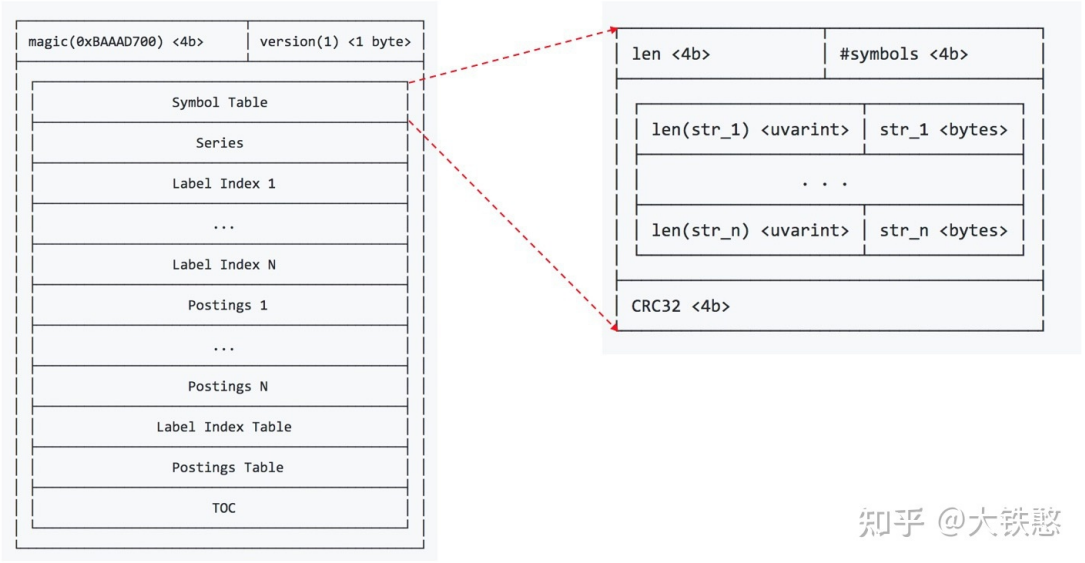


其中，TOC虽然位于文件的末尾，但是其包含了整个index文件的全局信息，它存储的内容是其余六部分的位置信息，即它们的起始位置在

index文件中的偏移量。在打开或加载文件时，TOC、Postings Table、Label Index Table会全部加载至内存中。

符号表symbol table

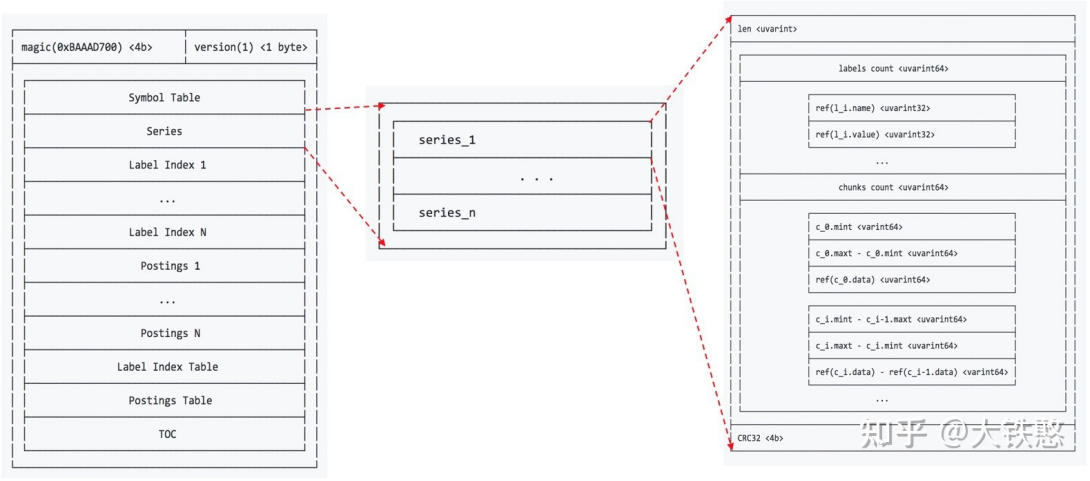
因为Labels中包含的key或者value存在大量的重复，如果直接存储原始字符串，会导致在构建时间序列索引和倒排索引时占用较大的磁盘空间。通过符号表，可以在存储时间序列series以及Label Index等信息的时候，就不需要完整存储所有的label了，只需将label的key和value用对应的字符串在symbol Table中的编号表示即可，从而大大减小了index文件的体积。符号表symbol table存储的是这个文件中包含所有的labels的key和value，即一个symbol既可以是一个label的key，也可以是它的value。在写入文件时，Prometheus为每个symbol使用自增的方式分配了一个编号，每个symbol的写入顺序是按照symbol的字典序升序写入的。需要值得注意的是，Prometheus的每个block都是独立自治，因此每个block内的index文件的符号表也是独立的，编号的分配都是在写入文件时独立分配的。



时间序列series

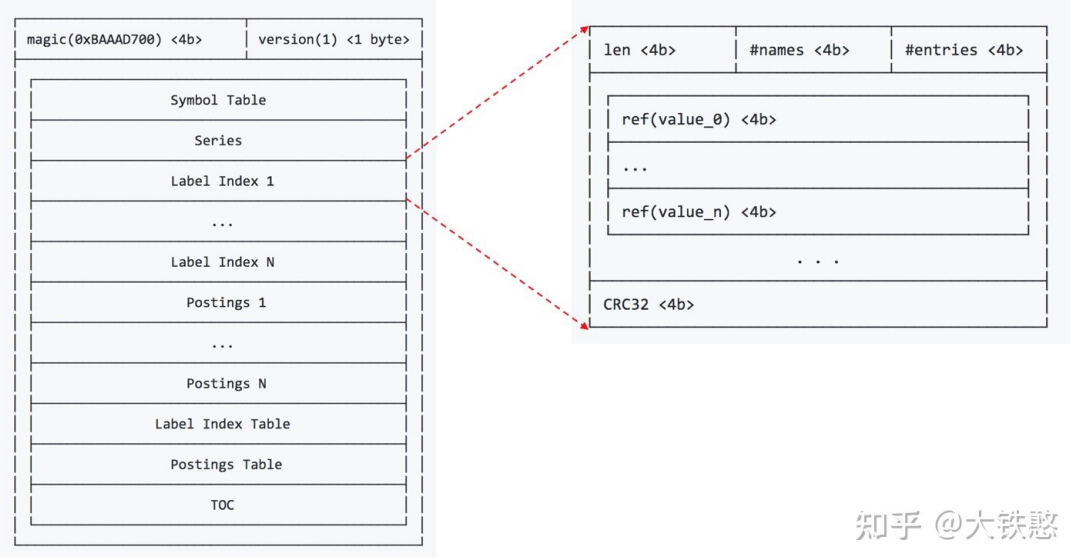
时间序列series存储是每个series包含的labels和一系列chunk块的索引信息。每个series首先存储是其包含的labels存储的是对应key和value在symbol table中的编号，然后存储的series相关的chunks的索引信息，包含每个chunk的时间窗口[mint, maxt]，以及该chunk在chunks子目录下的文件位置索引。其中对于每个chunk的时间范围，使用了变长

无符号整型和delta-encoding的方式进行的编码。时间序列series的写入顺序是按照series包含的labels的字典序升序写入的。另外，在写入时，会为每个series分配一个整型的ID，用于构建series的倒排索引。值得一提的是，prometheus为了避免在series中额外存储每个series分配好的ID，没有使用自增的方式生成series的ID，而是让每个series在文件中的offset都是按照16个字节对齐的，如果一个series的写入的大小无法对齐16个字节，将使用填0的方式对齐16个字节，这样就可以使用每个series在文件偏移量offset来隐式分配ID，即`ID = offset / 16`。



标签索引lable index

标签索引存储的是每个label的key和其关联的所有value，如下图所示。



例如，对于一个有着四个不同的value的key，其存储格式如下所示：

For instance, a single label name with 4 different values will be encoded as:

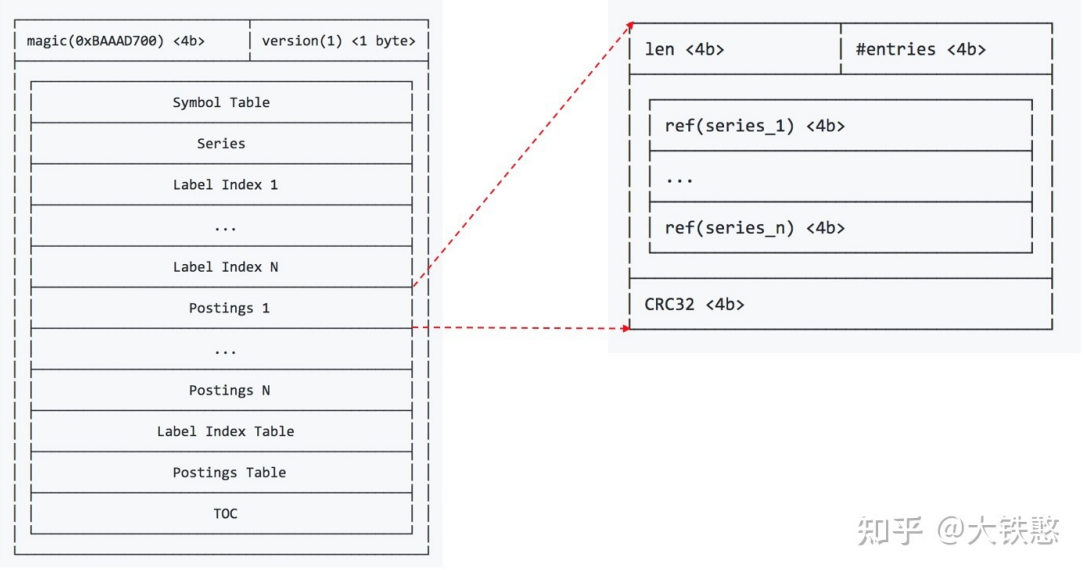
24	1	4	ref(value_0)	ref(value_1)	ref(value_2)	ref(value_3)	CRC32

The sequence of label index sections is finalized by a label offset table containing label offset entries that points to the beginning of each label index section for a given label name.

其中，每个label index条目可以存储多个key和它们的value的映射关系，因为可能有多个不同的key对应的value列表是一样的，但一般key的个数都为1。需要注意的是，这个条目中只存储了这些value关联的key的个数，而没有存储具体的key的信息，key的信息放在了后面的标签索引表中。

倒排索引postings

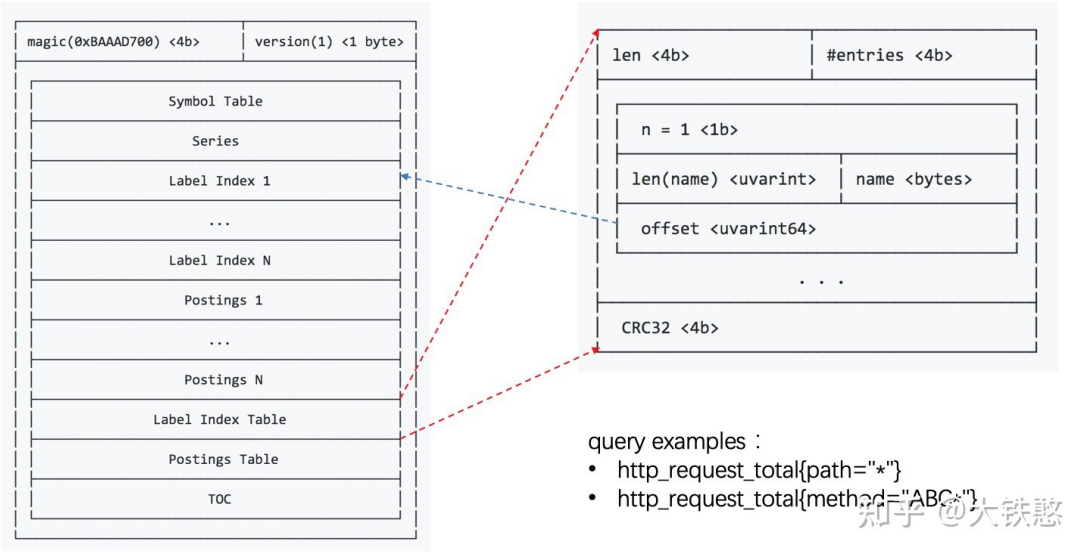
倒排索引包含的比较简单，存储的内容只是某个label关联的所有的series的ID。与标签索引Label Index相似，没有指定具体的key和value，具体的key和value存储在倒排索引表postings table中。



标签索引表label index table

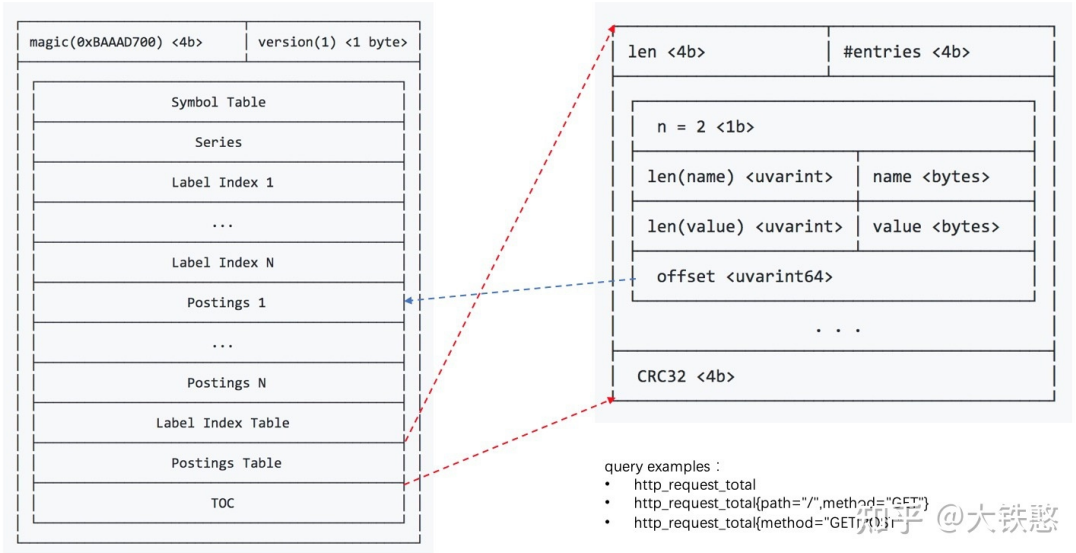
标签索引表label index table存储了所有label的key，以及这个key关联的value列表在标签索引label index中的索引。Prometheus在加载index文件文件时，会加载标签索引表label index table到内存中，但不加载标签索引lable index，可以加快文件打开速度，同时降低内存空间消耗。

这个标签索引表有什么用呢？当指定lable key但不指定查询的label value或者进行lable value模糊匹配时，就可以通过查询标签索引表查询这个key关联的所有lable value列表，然后根据label key和label value列表到倒排索引表postings table查询对应的series的ID的交集即可。比如`http_request_total{path="*"}`和`http_request_total{method="ABC*"}`等查询。



倒排索引表postings table

倒排索引表postings table存储是每个label的key和value，以及关联的series的ID列表在postings中的索引。当查询是精确查询时，可以直接查询倒排索引表和对应的倒排索引即可查询出关联的时间序列series的ID集合，然后根据ID查询时间序列series关联的chunk索引，然后根据chunk索引在chunk文件中读出chunk块，最后进行解压和过滤，即可得到每个series关联的原始数据点。



总结

Prometheus主要面向的是云原生下的APM场景，整体架构清晰明了，但由于是单机架构，导致无法存储长周期的数据，单机吞吐有限。整体上来说，虽然 Prometheus 文件格式设计比较巧妙，也还有一定的优化空间，比如倒排索引部分的series的ID列表可以使用位图或者变长整型来存储ID列表，可以进一步节省磁盘存储空间。对于Prometheus无法存储长周期的数据和单机吞吐有限的问题，开源社区提供了很多针对Prometheus的长期存储（Long-term storage）解决方案方案, 包括Thanos、VictoriaMetrics、M3、Cortex等。