influxdb源码分析 集群版Sharding过程

源码目录：

(1)client

主要是相关客户端API的封装和example

(2)cluster

集群方面的一些API,负责集群节点之间数据通讯（service），负责集群sharding（PointsWriter）

(3)cmd/influx

influx 命令行客户端

(4) cmd/influxd

influxdb 的main起始点

(9)influxql

influxql的语法解析包

(10)models

metric和point的数据结构

(11) monitor

influxdb 状态自监控目录

(12)service/admin

管理界面service

（13）service/collectd

外部插件

（14）service/continus_queirer

CQ服务

（15）service/copier

（16）service/graphite

外部插件

（17）service/httpd

http服务，对外提供读和写

（18）service/meta

meta client / server 通信结构   CP

（19）service/opentsdb

外部插件

（20）service/precreator

shard 预创器服务

（21）service/retention

提供定时清除过期数据服务

（22）service/snapshotter

快照服务

（23）service/subscriber

推送kapacitor插件服务

（24）service/udp

UDP推动服务

（25）tcp

influxdb自己实现的tcp复用技术

（26）toml

（27）tsdb

StoreDB

启动过程：

(1)cmd/influxd 中有四大命令 help backup restore run 默认的是run

(2)从run说起

```go
	case "", "run":
		cmd := run.NewCommand()

		// Tell the server the build details.
		cmd.Version = version
		cmd.Commit = commit
		cmd.Branch = branch

		if err := cmd.Run(args...); err != nil {
			return fmt.Errorf("run: %s", err)
		}

		signalCh := make(chan os.Signal, 1)
		signal.Notify(signalCh, os.Interrupt, syscall.SIGTERM)
		m.Logger.Println("Listening for signals")

		// Block until one of the signals above is received
		select {
		case <-signalCh:
			m.Logger.Println("Signal received, initializing clean shutdown...")
			go func() {
				cmd.Close()
			}()
		}
```

(3) 设定最大cpu使用核数，创建pid文件，建立build info ，解析config，确定config文件合法性

使用config 初始化server

```go
// services in the proper order.
type Server struct {
	buildInfo BuildInfo

	err     chan error
	closing chan struct{}

	BindAddress string
	Listener    net.Listener

	Logger *log.Logger

	MetaClient *meta.Client

	TSDBStore     *tsdb.Store
	QueryExecutor *influxql.QueryExecutor
	PointsWriter  *cluster.PointsWriter
	Subscriber    *subscriber.Service

	Services []Service

	// These references are required for the tcp muxer.
	ClusterService     *cluster.Service
	SnapshotterService *snapshotter.Service
	CopierService      *copier.Service

	Monitor *monitor.Monitor

	// Server reporting and registration
	reportingDisabled bool

	// Profiling
	CPUProfile string
	MemProfile string

	// httpAPIAddr is the host:port combination for the ma
	httpAPIAddr string

	// httpUseTLS specifies if we should use a TLS connect
	httpUseTLS bool

	// tcpAddr is the host:port combination for the TCP li
	tcpAddr string
```

s,err:=NewServer(config)
s.open()

(4) NewServer中干了哪些事

4.1)创建meta的目录，即使不实meta node 也需要创建

4.2）加载node， node是一个包含path 和ID的结构体
如果不存在则创建

4.3) 判断是否是meta 节点或者data 节点 都不是则返回错误

4.4) 初始化sever

新建meta.client

初始化raft端口设置

新建monitor service

设置是否上报

设置集群joinpeers

设置httpd的API端口

设置tcp复用端口

设置config

如果meta.Enabled是true 也就是如果是meta 节点

新建meta.service

```
if c.Meta.Enabled {
    s.MetaService = meta.NewService(c.Meta)
    s.MetaService.Version = s.buildInfo.Version
    s.MetaService.Node = s.Node
}
```

如果data.Enabled是true 也就是如果是数据存储节点

A) TSDBStore 初始化

```
s.TSDBStore = tsdb.NewStore(c.Data.Dir)
s.TSDBStore.EngineOptions.Config = c.Data

// Copy TSDB configuration.
s.TSDBStore.EngineOptions.EngineVersion = c.Data.Engine
```

B)ShardWriter 和handhintoff 初始化，subscriber的新建service

shardwriter 主要是进行shard write 使用的一个通用结构，是由cluster来进行实现

Hinted Handoff作为写操作的可选的一部分，主要目的是当不要求一致性的时候，提高写的高可用性

```
// Set the shard writer
s.ShardWriter = cluster.NewShardWriter(time.Duration(c.Cluster.ShardWriterTimeout),
    c.Cluster.MaxRemoteWriteConnections)

// Create the hinted handoff service
s.HintedHandoff = hh.NewService(c.HintedHandoff, s.ShardWriter, s.MetaClient)
s.HintedHandoff.Monitor = s.Monitor

// Create the Subscriber service
s.Subscriber = subscriber.NewService(c.Subscriber)
```

C) PointsWriter 初始化

```
// Initialize points writer.
s.PointsWriter = cluster.NewPointsWriter()
s.PointsWriter.WriteTimeout = time.Duration(c.Cluster.WriteTimeout)
s.PointsWriter.TSDBStore = s.TSDBStore
s.PointsWriter.ShardWriter = s.ShardWriter
s.PointsWriter.HintedHandoff = s.HintedHandoff
s.PointsWriter.Subscriber = s.Subscriber
s.PointsWriter.Node = s.Node
```

points write 是一个聚合集群shardwriter和TSDBStore、hintedHandoff的结构

如果在该台节点上写入数据，如果正好sharding到自己这台机器则直接写，如果不是则调用shardwriter 写到指定机器上

D）meta 执行器

```
// Initialize meta executor.
metaExecutor := cluster.NewMetaExecutor()
metaExecutor.MetaClient = s.MetaClient
metaExecutor.Node = s.Node
```

E) Query 执行器

```go
// Initialize query executor.
s.QueryExecutor = cluster.NewQueryExecutor()
s.QueryExecutor.MetaClient = s.MetaClient
s.QueryExecutor.TSDBStore = s.TSDBStore
s.QueryExecutor.Monitor = s.Monitor
s.QueryExecutor.PointsWriter = s.PointsWriter
s.QueryExecutor.MetaExecutor = metaExecutor
if c.Data.QueryLogEnabled {
	s.QueryExecutor.LogOutput = os.Stderr
}
```

query 执行器可以执行读和写的sql解析并执行

它包含 metaclient 包含metaExecutor 包含TSDB数据库 包含PointWriter 写工具

(5)s.Open() 打开service

5.1) 设置多路复用共享端口Listen

```go
// Open shared TCP connection.
ln, err := net.Listen("tcp", s.BindAddress)
if err != nil {
	return fmt.Errorf("listen: %s", err)
}
s.Listener = ln

// Multiplex listener.
mux := tcp.NewMux()
go mux.Serve(ln)
```

influxdb 自己实现了tcp端口复用器

5.2) 如果自己是meta node， 也就是有metaService

```go
if s.MetaService != nil {
	s.MetaService.RaftListener = mux.Listen(meta.MuxHeader)
	// Open meta service.
	if err := s.MetaService.Open(); err != nil {
		return fmt.Errorf("open meta service: %s", err)
	}
	go s.monitorErrorChan(s.MetaService.Err())
}
```

设置raft 监听端口为复用端口， 并且打开metaservice 服务

5.3) meta client的一些初始化工作

```go
// initialize MetaClient.
if err = s.initializeMetaClient(); err != nil {
	return err
}
```

5.4)如果自己是data node 则进行以下工作

A）添加各种服务

```go
if s.TSDBStore != nil {
	// Append services.
	s.appendClusterService(s.config.Cluster)
	s.appendPrecreatorService(s.config.Precreator)
	s.appendSnapshotterService()
	s.appendCopierService()
	s.appendAdminService(s.config.Admin)
	s.appendContinuousQueryService(s.config.ContinuousQuery)
	s.appendHTTPDService(s.config.HTTPD)
	s.appendCollectdService(s.config.Collectd)
	if err := s.appendOpenTSDBService(s.config.OpenTSDB); err != nil {
		return err
	}
	for _, g := range s.config.UDPs {
		s.appendUDPService(g)
	}
	s.appendRetentionPolicyService(s.config.Retention)
	for _, g := range s.config.Graphites {
		if err := s.appendGraphiteService(g); err != nil {
			return err
		}
	}
}
```

添加集群服务

添加shard 预创建服务

添加快照服务
添加copy服务
添加Admin 服务
添加CQ服务
添加 HTTPD服务
添加Collectd服务
添加TSDB服务
添加UDP服务
添加RetentionPolicy服务
添加graphite服务

B）设置metaclient 参数，和实现端口复用，集群服务，快照服务和拷贝服务都端口复用

```
}

s.QueryExecutor.Node = s.Node

s.Subscriber.MetaClient = s.MetaClient
s.ShardWriter.MetaClient = s.MetaClient
s.HintedHandoff.MetaClient = s.MetaClient
s.Subscriber.MetaClient = s.MetaClient
s.PointsWriter.MetaClient = s.MetaClient
s.Monitor.MetaClient = s.MetaClient

s.ClusterService.Listener = mux.Listen(cluster.MuxHeader)
s.SnapshotterService.Listener = mux.Listen(snapshotter.MuxHeader)
s.CopierService.Listener = mux.Listen(copier.MuxHeader)
```

C) 打开所有服务
XXX.Open()

写数据过程：

通过上面的启动过程分析，可以知道所有meta 节点和data 节点都已经启动了，对外提供服务，

这时候对外提供的读写的http服务主要由httpd 来进行服务

（1）在什么时候初始化和启动的httpd服务

```go
func (s *Server) appendHTTPDService(c httpd.Config) {
	if !c.Enabled {
		return
	}
	srv := httpd.NewService(c)
	srv.Handler.MetaClient = s.MetaClient
	srv.Handler.QueryAuthorizer = meta.NewQueryAuthorizer(s.MetaClient)
	srv.Handler.QueryExecutor = s.QueryExecutor
	srv.Handler.PointsWriter = s.PointsWriter
	srv.Handler.Version = s.buildInfo.Version

	// If a ContinuousQuerier service has been started, attach it.
	for _, srvc := range s.Services {
		if cqsrvc, ok := srvc.(continuous_querier.ContinuousQuerier); ok {
			srv.Handler.ContinuousQuerier = cqsrvc
		}
	}

	s.Services = append(s.Services, srv)
```

httpd 中有metaclient 有授权结构 、有Query执行器和写工具、和CQ服务

（2）httpd中主要处理读和写的是handler这个结构,handler中主要处理以下路由

```go
h.SetRoutes([]route{
	route{
		"query", // Satisfy CORS checks.
		"OPTIONS", "/query", true, true, h.serveOptions,
	},
	route{
		"query", // Query serving route.
		"GET", "/query", true, true, h.serveQuery,
	},
	route{
		"write", // Satisfy CORS checks.
		"OPTIONS", "/write", true, true, h.serveOptions,
	},
	route{
		"write", // Data-ingest route.
		"POST", "/write", true, true, h.serveWrite,
	},
	route{ // Ping
		"ping",
		"GET", "/ping", true, true, h.servePing,
	},
	route{ // Ping
		"ping-head",
		"HEAD", "/ping", true, true, h.servePing,
	},
	route{ // Ping w/ status
		"status",
		"GET", "/status", true, true, h.serveStatus,
	},
	route{ // Ping w/ status
		"status-head",
		"HEAD", "/status", true, true, h.serveStatus,
	},
	route{ // Tell data node to run CQs that should be run
		"process_continuous_queries",
		"POST", "/data/process_continuous_queries", false, false, h.serveProcessContinuousQueries,
	},
```

(3) 写数据服务函数 处理write serveWrite

检查是否是gzip压缩，如果是压缩则解压

检查是否是json形式写， 如果是则调用json形式写

如果是line协议写则调用line协议写

(4) Line协议写函数

解析存储精度 默认为ns

解析db 解析错误则返回错误

调用metaclient的Database函数根据dbname查找database Info

设置数据一致性等级，默认为写一个节点就成功返回

解析请求中一致性等级

解析retention policy

调用写工具PointsWriter 写数据（database,rp,一致性等级， 点数据）

```
// Write points.
if err := h.PointsWriter.WritePoints(&cluster.WritePointsRequest{
	Database:         database,
	RetentionPolicy:  r.FormValue("rp"),
	ConsistencyLevel: consistency,
	Points:           points,
```

(5) 现在整个写过程请转到PointsWriter这个工具的writePoints中

PointsWriter在上面的初始化我们已经介绍过，它是由cluster这个包中初始化的，所以这个写应该由cluster来决策，写到哪里

让我们进入Cluster包中PointsWriter的WritePoints

```
// WritePoints writes across multiple local and remote data nodes c
func (w *PointsWriter) WritePoints(p *WritePointsRequest) error {
```

终于见到庐山真面目,传入WritePointsrequest结构，然后就被写到集群中去了，具体如何写的呢，继续扒

A) 如果WritePointsrequest传入的rp为空，则复值为默认rp

```
if p.RetentionPolicy == "" {
	db, err := w.MetaClient.Database(p.Database)
	if err != nil {
		return err
	} else if db == nil {
		return influxdb.ErrDatabaseNotFound(p.Database)
	}
	p.RetentionPolicy = db.DefaultRetentionPolicy
}
```

B)根据传入的WritePointsrequest计算shardMappings

```
shardMappings, err := w.MapShards(p)
if err != nil {
	return err
}
```

#)shardmappings 是个什么东西

```go
// ShardMapping contains a mapping of a shards to a points.
type ShardMapping struct {
	Points map[uint64][]models.Point  // The points associated with a shard ID
	Shards map[uint64]*meta.ShardInfo // The shards that have been mapped, keyed by shard ID
}
```

#)shardMapping 原来是这个东西，主要包含两个map的结构体

1.第一个表示某个shardID对应的点数组的map

2. 第二个shard ID 对应的Shard的Info的map

#)ShardInfo里面包含哪些东西呢

```go
// ShardInfo represents metadata about a shard
type ShardInfo struct {
	ID     uint64
	Owners []ShardOwner
}
```

```go
// ShardOwner represents a node that owns a shard
type ShardOwner struct {
	NodeID uint64
}
```

上面的结构已经很清晰，所谓shardInfo主要由shardID 和shard所在的哪些node上

总结： 计算shardMapping 也就是将这一群点分到对应shard上和对应node 节点上

C) 接下来看看MapShard是根据WritePointsrequest如何填充shardMapping的

```go
// MapShards maps the points contained in wp to a ShardMapping.  If a point
// maps to a shard group or shard that does not currently exist, it will be
// created before returning the mapping.
func (w *PointsWriter) MapShards(wp *WritePointsRequest) (*ShardMapping, error) {
```

D）新建时间和shardGroupInfo之间的map结构

```go
// holds the start time ranges for required shard groups
timeRanges := map[time.Time]*meta.ShardGroupInfo{}
```

```go
// safely delete any associated
type ShardGroupInfo struct {
	ID        uint64
	StartTime time.Time
	EndTime   time.Time
	DeletedAt time.Time
	Shards    []ShardInfo
}
```

E) 获取RPInfo

```go
rp, err := w.MetaClient.RetentionPolicy(wp.Database, wp.RetentionPolicy)
if err != nil {
	return nil, err
}
if rp == nil {
	return nil, influxdb.ErrRetentionPolicyNotFound(wp.RetentionPolicy)
}
```

```go
// RetentionPolicyInfo represents metadata
type RetentionPolicyInfo struct {
	Name               string
	ReplicaN           int
	Duration           time.Duration
	ShardGroupDuration time.Duration
	ShardGroups        []ShardGroupInfo
	Subscriptions      []SubscriptionInfo
}
```

其中Name 表示rp的name 比如有的人起名字two_week

ReplicaN 表示保存副本几份

Duration   表示存多久

shardgroupDuration 表示根据duration 计算出来的
duration <2day shradgroupduration =1h
duration >=2day && duration<180d shradgroupduration=1d
duration>=180d shradgroupduration=7d

F)将所有的点就算分配到timeRanges中

```
for _, p := range wp.Points {
>	timeRanges[p.Time().Truncate(rp.ShardGroupDuration)] = nil
}
```

其中rp.shardGroupDuration表示每个shard周期为时间 比如大于180d 此时 rp.shardGroupDuration＝7d

Trucate的含义去尾法求近似值

比如：

```
t, _ := time.Parse("2006 Jan 02 15:04:05", "2012 Dec 07 12:15:30.918273645")

fmt.Printf("t.Truncate(%6s) = %s\n", d, t.Truncate(d).Format("15:04:05.999999999"))

t.Truncate(   1ns) = 12:15:30.918273645
t.Truncate(   1µs) = 12:15:30.918273
t.Truncate(   1ms) = 12:15:30.918
t.Truncate(    1s) = 12:15:30
t.Truncate(    2s) = 12:15:30
t.Truncate(  1m0s) = 12:15:00
t.Truncate( 10m0s) = 12:10:00
t.Truncate(1h0m0s) = 12:00:00
```

如果当前的点是2018-01-05 10:10:10 如果按照7d为最小近似则
2018-01-01～2018-01-07 ======> 2018-01-01
2018-01-08~2018-01-15 ======> 2018-01-08

简而言之就是将这些点按照rp.shardGroupDuration进行分时间段 也就是分shardGroup

G)对所有的分类完shardgroup 进行填充shardGroupInfo

```go
// holds all the shard groups and shards that are required for writes
for t := range timeRanges {
	sg, err := w.MetaClient.CreateShardGroup(wp.Database, wp.RetentionPolicy, t)
	if err != nil {
		return nil, err
	}
	timeRanges[t] = sg
}
```

根据timeRange中的key 进行创建ShardGroupInfo

H) CreateShardGroup是如何创建的ShardGroupInfo的呢传入dbname 、rp、和 t

这个需要进入meta包中寻找CreateShardGroup函数

```go
// CreateShardGroup creates a shard group on a database and policy for a given timestamp.
func (c *Client) CreateShardGroup(database, policy string, timestamp time.Time) (*ShardGroupInfo, error) {
	if sg, _ := c.data().ShardGroupByTimestamp(database, policy, timestamp); sg != nil {
		return sg, nil
	}

	cmd := &internal.CreateShardGroupCommand{
		Database:  proto.String(database),
		Policy:    proto.String(policy),
		Timestamp: proto.Int64(timestamp.UnixNano()),
	}

	if err := c.retryUntilExec(internal.Command_CreateShardGroupCommand, internal.E_CreateShardGroupCommand_
		return nil, err
	}

	rpi, err := c.RetentionPolicy(database, policy)
	if err != nil {
		return nil, err
	} else if rpi == nil {
		return nil, errors.New("retention policy deleted after shard group created")
	}

	return rpi.ShardGroupByTimestamp(timestamp), nil
}
```

#) 首先调用metaclient.data去数据中心找ShardgroupByTimeStamp 函数找这个时间段的ShardgroupInfo，找到直接返回没找到就创建

#) 构建cmd的CreateShardGroupCmd

#）调用meta client的retryuntilExec 重试直到成功函数 创建该GroupShard，应该是调用meta service raft服务来创建

#) meta Service 在meta data中如何创建的

```go
// CreateShardGroup creates a shard group on a database and policy for a given timestamp.
func (data *Data) CreateShardGroup(database, policy string, timestamp time.Time) error {
	// Ensure there are nodes in the metadata
```

```go
// Require at least one replica but no more replicas than node
replicaN := rpi.ReplicaN
if replicaN == 0 {
	replicaN = 1
} else if replicaN > len(data.DataNodes) {
	replicaN = len(data.DataNodes)
}

// Determine shard count by node count divided by replication
// This will ensure nodes will get distributed across nodes ev
// replicated the correct number of times.
shardN := len(data.DataNodes) / replicaN
```

先读区该rp中rpN 如果rpN为0则至少写一份
如果rpN的数值比data node 的节点数还多的话就写每个节点写一份
shardN＝节点数目 / rpN
例如： 如果6个节点 副本数目2分 那可以供选择shard的组数就有2组

```go
// Create the shard group.
data.MaxShardGroupID++
sgi := ShardGroupInfo{}
sgi.ID = data.MaxShardGroupID
sgi.StartTime = timestamp.Truncate(rpi.ShardGroupDuration).UTC()
sgi.EndTime = sgi.StartTime.Add(rpi.ShardGroupDuration).UTC()

// Create shards on the group.
sgi.Shards = make([]ShardInfo, shardN)
for i := range sgi.Shards {
	data.MaxShardID++
	sgi.Shards[i] = ShardInfo{ID: data.MaxShardID}
}
```

data数据中的最大ShardGroupID自增
开始时间设置为去尾近似时间，结束时间为开始时间＋每个shardGroup的时间
给shardgroupInfo中shards 创建ShardN个Shards

```
// start from a repeatably random place in the node list.
nodeIndex := int(data.Index % uint64(len(data.DataNodes)))
for i := range sgi.Shards {
>	si := &sgi.Shards[i]
>	for j := 0; j < replicaN; j++ {
>	>	nodeID := data.DataNodes[nodeIndex%len(data.DataNodes)].ID
>	>	si.Owners = append(si.Owners, ShardOwner{NodeID: nodeID})
>	>	nodeIndex++
>	}
}
```

给每个shards随机生成replicaN个owner

实例解析：

比如我有5个data Node 节点 要求每分数据写2分 那么shardGroup会创建2个shards 每个shards随机选两个owner

比如单机版的就是1个dataNode 写1份数据 shardGroup只会有1个shard

#）创建成功后调用meta client的rp函数获取rpi

#）在所有rpi中找timestamp的GroupInfo

l)将所有点按照规则填充到shardMapping中去

```
mapping := NewShardMapping()
for _, p := range wp.Points {
>	sg := timeRanges[p.Time().Truncate(rp.ShardGroupDuration)]
>	sh := sg.ShardFor(p.HashID())
>	mapping.MapPoint(&sh, p)
}
return mapping, nil
```

#）将所有点依次找到对应的shardGroupInfo

#）计算point的hashID

对p.key进行hash key为 measurement_name+tags_key1+tag_value1+tag_key2+tag_value2......

#) ShardFor

```go
// ShardFor returns the ShardInfo for a Point hash
func (sgi *ShardGroupInfo) ShardFor(hash uint64) ShardInfo {
	return sgi.Shards[hash%uint64(len(sgi.Shards))]
}
```

将每个不同seriers的key分别求余放到不同shard中，如果某个shardGroup只有一个shard，那么全部放到这个
shard中

#) 将每个点放倒对应的shardID中形成shardMapping

(6) 再将所有的点都分到对应的ShardGroup的Shard中后，形成K个shardID对应这些points ，然后依次构造K个gorotine将
这些points 写到集群中去，shardInfo中都有对应的owner node id 所以写起来就简单了

```go
// Write each shard in it's own goroutine and return as soon
// as one fails.
ch := make(chan error, len(shardMappings.Points))
for shardID, points := range shardMappings.Points {
	go func(shard *meta.ShardInfo, database, retentionPolicy string, points []models.Point) {
		ch <- w.writeToShard(shard, p.Database, p.RetentionPolicy, p.ConsistencyLevel, points)
	}(shardMappings.Shards[shardID], p.Database, p.RetentionPolicy, points)
}
```

（7）调用PointsWriter的WriteToShard函数，传入 shardInfo， database，rp，consistency，points

```go
// writeToShards writes points to a shard and ensures a write consistency level has been met
// partially succeeds, ErrPartialWrite is returned.
func (w *PointsWriter) writeToShard(shard *meta.ShardInfo, database, retentionPolicy string,
	consistency ConsistencyLevel, points []models.Point) error {
```

根据owners 和一致性等级来判定写完多少就算成功

```go
	// The required number of writes to achieve the
	required := len(shard.Owners)
	switch consistency {
	case ConsistencyLevelAny, ConsistencyLevelOne:
		required = 1
	case ConsistencyLevelQuorum:
		required = required/2 + 1
	}
```

如果为any或者one 写1个成功就返回成功
如果其他就一半＋1节点成功就返回成功

```go
for _, owner := range shard.Owners {
	go func(shardID uint64, owner meta.ShardOwner, points []models.Point) {
		if w.Node.ID == owner.NodeID {
			w.statMap.Add(statPointWriteReqLocal, int64(len(points)))

			err := w.TSDBStore.WriteToShard(shardID, points)
			// If we've written to shard that should exist on the current node, bu
			// not actually created this shard, tell it to create it and retry the
			if err == tsdb.ErrShardNotFound {
				err = w.TSDBStore.CreateShard(database, retentionPolicy, shardID)
				if err != nil {
					ch <- &AsyncWriteResult{owner, err}
					return
				}
				err = w.TSDBStore.WriteToShard(shardID, points)
			}
			ch <- &AsyncWriteResult{owner, err}
			return
		}
```

#)根据每个owner进行开gorutine写数据

#）如果onwer.ID==current.Node.ID
直接调用TSDBStore写Shard
如果不存在Shard
在TSDBStore中创建Shard
然后继续写Shard
#) 不应该写到此节点上，则调用shard Write 进行WriteShard, shardwriter 也是cluster包中工具

```go
// WriteShard writes time series points to a shard
func (w *ShardWriter) WriteShard(shardID, ownerID uint64, points []models.Point) error
	c, err := w.dial(ownerID)
	if err != nil {
		return err
	}

	conn, ok := c.(*pooledConn)
	if !ok {
		panic("wrong connection type")
	}
	defer func(conn net.Conn) {
		conn.Close() // return to pool
	}(conn)
```

调用集群通讯服务进行写Shard，cluster包中的service中有响应如下

```go
switch typ {
case writeShardRequestMessage:
	buf, err := ReadLV(conn)
	if err != nil {
		s.Logger.Printf("unable to read length-value: %s", err)
		return
	}

	s.statMap.Add(writeShardReq, 1)
	err = s.processWriteShardRequest(buf)
	if err != nil {
		s.Logger.Printf("process write shard error: %s", err)
	}
	s.writeShardResponse(conn, err)
```

```go
func (s *Service) processWriteShardRequest(buf []byte) error {
	// Build request
	var req WriteShardRequest
	if err := req.UnmarshalBinary(buf); err != nil {
		return err
	}

	points := req.Points()
	s.statMap.Add(writeShardPointsReq, int64(len(points)))
	err := s.TSDBStore.WriteToShard(req.ShardID(), points)

	// We may have received a write for a shard that we don't ha
```

(8) 最后还是归结到TSDB Write Shard，让我们直接进入TSDB的前世今生中，并如何执行Write Shard的

```go
s.TSDBStore = tsdb.NewStore(c.Data.Dir)
s.TSDBStore.EngineOptions.Config = c.Data

// Copy TSDB configuration.
s.TSDBStore.EngineOptions.EngineVersion = c.Data.Engine
```

在前面tsdb store是这么初始化的，传入data的目录和引擎的Config和引擎的版本就把TSDB跑起来了

A)继续我们的正题，上面已经调用到TSDBStore的WriteToShard函数

```go
// WriteToShard writes a list of points to a shard identified by its ID.
func (s *Store) WriteToShard(shardID uint64, points []models.Point) error {
	s.mu.RLock()
	defer s.mu.RUnlock()

	select {
	case <-s.closing:
		return ErrStoreClosed
	default:
	}

	sh, ok := s.shards[shardID]
	if !ok {
		return ErrShardNotFound
	}

	return sh.WritePoints(points)
}
```

根据shardID照到对应的Shard

B) 调用该Shard的WritePoints函数,这个shard在tsdb的store目录中

```go
// WritePoints will write the raw data points and any new metadata to the index in the shard
func (s *Shard) WritePoints(points []models.Point) error {
	s.statMap.Add(statWriteReq, 1)

	seriesToCreate, fieldsToCreate, seriesToAddShardTo, err := s.validateSeriesAndFields(points)
	if err != nil {
		return err
	}
	s.statMap.Add(statSeriesCreate, int64(len(seriesToCreate)))
	s.statMap.Add(statFieldsCreate, int64(len(fieldsToCreate)))
```

```go
// add any new series to the in-memory index
if len(seriesToCreate) > 0 {
	s.index.mu.Lock()
	for _, ss := range seriesToCreate {
		s.index.CreateSeriesIndexIfNotExists(ss.Measurement, ss.Series)
	}
	s.index.mu.Unlock()
}

if len(seriesToAddShardTo) > 0 {
	s.index.mu.Lock()
	for _, k := range seriesToAddShardTo {
		ss := s.index.series[k]
		if ss != nil {
			ss.shardIDs[s.id] = true
		}
	}
	s.index.mu.Unlock()
}

// add any new fields and keep track of what needs to be saved
measurementFieldsToSave, err := s.createFieldsAndMeasurements(fieldsToCreate)
if err != nil {
	return err
}
```

```go
// only required for the b1 and bz1 formats
if s.engine.Format() != TSM1Format {
	for _, p := range points {
		// Ignore if raw data has already been marshaled.
		if p.Data() != nil {
			continue
		}

		// This was populated earlier, don't need to validate that it's there.
		s.mu.RLock()
		mf := s.measurementFields[p.Name()]
		s.mu.RUnlock()

		// If a measurement is dropped while writes for it are in progress, this could be nil
		if mf == nil {
			return ErrFieldNotFound
		}

		data, err := mf.Codec.EncodeFields(p.Fields())
		if err != nil {
			return err
		}
		p.SetData(data)
	}
}

// Write to the engine.
if err := s.engine.WritePoints(points, measurementFieldsToSave, seriesToCreate); err != nil {
	s.statMap.Add(statWritePointsFail, 1)
	return fmt.Errorf("engine: %s", err)
}
```

结果engine.WritePoints是一个接口，我们得找到源函数

C)直接调用engine的WritePoints进行写,在研究engine的WritePoints函数之前我们了解一下这个引擎如何构造和启动的