2020年11月15日

监控日志系列---- Filebeat原理

Filebeat 是使用 Golang 实现的轻量型日志采集器，也是 Elasticsearch stack 里面的一员。本质上是一个 agent，可以安装在各个节点上，根据配置读取对应位置的日志，并上报到相应的地方去。

filebeat源码归属于beats项目，而beats项目的设计初衷是为了采集各类的数据，所以beats抽象出了一个libbeat库，基于libbeat我们可以快速的开发实现一个采集的工具，除了filebeat，还有像metricbeat、packetbeat等官方的项目也是在beats工程中。libbeat已经实现了内存缓存队列memqueue、几种output日志发送客户端，数据的过滤处理processor,配置解析、日志打印、事件处理和发送等通用功能，而filebeat只需要实现日志文件的读取等和日志相关的逻辑即可。

beats

对于任一种beats来说，主要逻辑都包含两个部分：

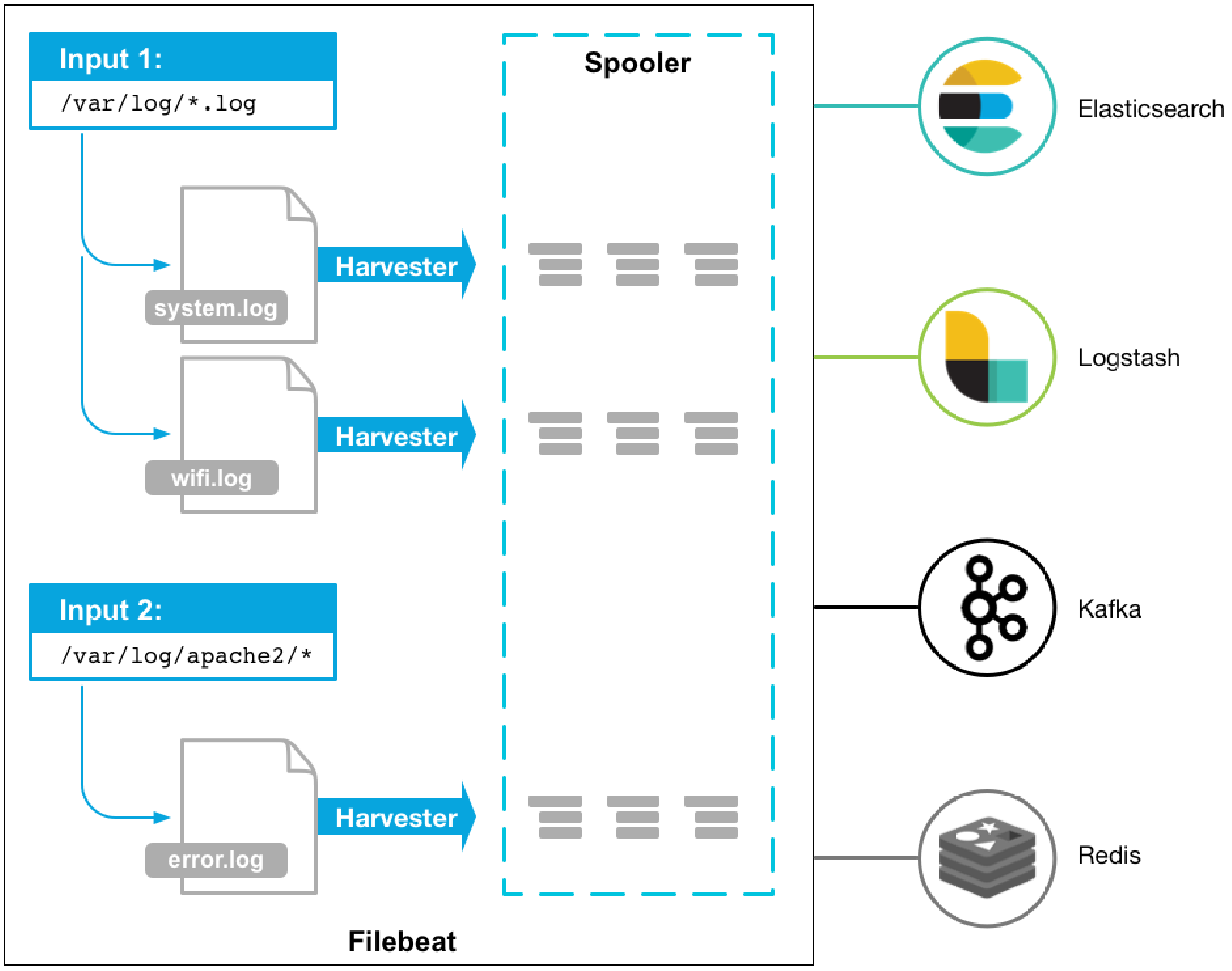
- 收集数据并转换成事件
- 发送事件到指定的输出

其中第二点已由libbeat实现，因此各个beats实际只需要关心如何收集数据并生成事件后发送给libbeat的Publisher。

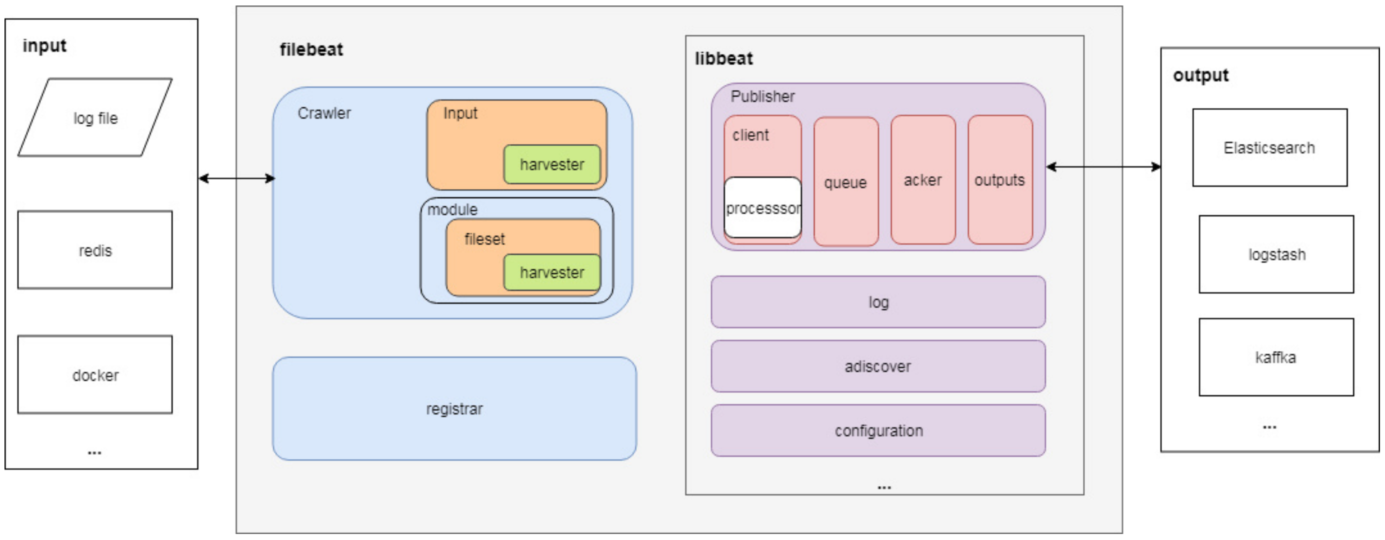
filebeat整体架构

架构图

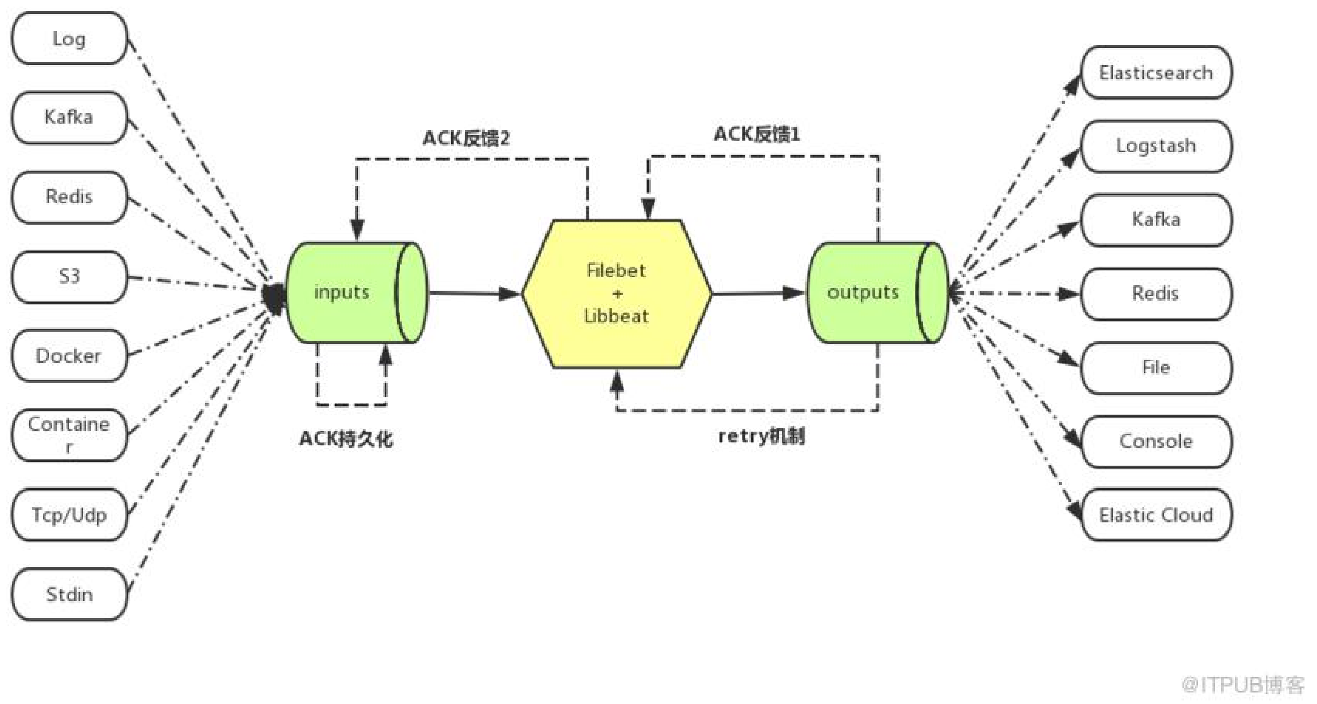
下图是 Filebeat 官方提供的架构图：



下图是看代码的一些模块组合



其实我个人觉得这一幅图是最形象的说明了filebeat的功能



模块

除了图中提到的各个模块，整个 filebeat 主要包含以下重要模块：

1.filebeat主要模块

```
Crawler: 负责管理和启动各个Input,管理所有Input收集数据并发送事件到libbeat的Publisher
Input: 负责管理和解析输入源的信息，以及为每个文件启动 Harvester。可由配置文件指定输入源信息。
    Harvester: 负责读取一个文件的数据,对应一个输入源， 是收集数据的实际工作者。配置中，一个具体的Input可以包含多个输入源（Harvester）
module: 简化了一些常见程序日志（比如nginx日志）收集、解析、可视化（kibana dashboard）配置项
    fileset: module下具体的一种Input定义（比如nginx包括access和error log），包含：
        1）输入配置；
        2）es ingest node pipeline定义；
        3）事件字段定义；
        4）示例kibana dashboard
Registrar：接收libbeat反馈回来的ACK, 作相应的持久化，管理记录每个文件处理状态，包括偏移量、文件名等信息。当 Filebeat 启动时，会从 Registrar 恢复文件处理状态。
```

2.libbeat主要模块

```
Pipeline（publisher）: 负责管理缓存、Harvester 的信息写入以及 Output 的消费等，是 Filebeat 最核心的组件。
    client: 提供Publish接口让filebeat将事件发送到Publisher。在发送到队列之前，内部会先调用processors（包括input 内部的processors和全局processors）进行处理。
    processor: 事件处理器，可对事件按照配置中的条件进行各种处理（比如删除事件、保留指定字段，过滤添加字段，多行合并等）。配置项
    queue: 事件队列，有memqueue（基于内存）和spool（基于磁盘文件）两种实现。配置项
    outputs: 事件的输出端，比如ES、Logstash、kafka等。配置项
    acker: 事件确认回调，在事件发送成功后进行回调
autodiscover：用于自动发现容器并将其作为输入源
```

filebeat 的整个生命周期，几个组件共同协作，完成了日志从采集到上报的整个过程。

基本原理（源码解析）

文件目录组织

```
├── autodiscover          # 包含filebeat的autodiscover适配器（adapter），当autodiscover发现新容器时创建对应类型的输入
├── beater                # 包含与libbeat库交互相关的文件
├── channel               # 包含filebeat输出到pipeline相关的文件
├── config                # 包含filebeat配置结构和解析函数
├── crawler               # 包含Crawler结构和相关函数
├── fileset               # 包含module和fileset相关的结构
├── harvester             # 包含Harvester接口定义、Reader接口及实现等
├── input                 # 包含所有输入类型的实现（比如：log, stdin, syslog）
├── inputsource           # 在syslog输入类型中用于读取tcp或udp syslog
├── module                # 包含各module和fileset配置
├── modules.d             # 包含各module对应的日志路径配置文件，用于修改默认路径
├── processor             # 用于从容器日志的事件字段source中提取容器id
├── prospector            # 包含旧版本的输入结构Prospector，现已被Input取代
├── registrar             # 包含Registrar结构和方法
└── util                  # 包含beat事件和文件状态的通用结构Data
└── ...
```

在这些目录中还有一些重要的文件

/beater：包含与libbeat库交互相关的文件：

```
acker.go: 包含在libbeat设置的ack回调函数，事件成功发送后被调用
channels.go: 包含在ack回调函数中被调用的记录者（logger），包括：
    registrarLogger: 将已确认事件写入registrar运行队列
    finishedLogger: 统计已确认事件数量
filebeat.go: 包含实现了beater接口的filebeat结构，接口函数包括：
    New：创建了filebeat实例
    Run：运行filebeat
    Stop: 停止filebeat运行
signalwait.go：基于channel实现的等待函数，在filebeat中用于：
    等待fileebat结束
    等待确认事件被写入registry文件
```

/channel：filebeat输出（到pipeline）相关的文件

```
factory.go: 包含OutletFactory，用于创建输出器Outleter对象
interface.go: 定义输出接口Outleter
outlet.go: 实现Outleter，封装了libbeat的pipeline client，其在harvester中被调用用于将事件发送给pipeline
util.go: 定义ack回调的参数结构data，包含beat事件和文件状态
```

/input：包含Input接口及各种输入类型的Input和Harvester实现

```
Input：对应配置中的一个Input项，同个Input下可包含多个输入源（比如文件）
Harvester：每个输入源对应一个Harvester，负责实际收集数据、并发送事件到pipeline
```

/harvester：包含Harvester接口定义、Reader接口及实现等

```
forwarder.go: Forwarder结构（包含outlet）定义，用于转发事件
harvester.go: Harvester接口定义，具体实现则在/input目录下
registry.go: Registry结构，用于在Input中管理多个Harvester（输入源）的启动和停止
source.go: Source接口定义，表示输入源。目前仅有Pipe一种实现（包含os.File），用在log、stdin和docker输入类型中。btw，这三种输入类型都是用的log input的实现。
/reader目录: Reader接口定义和各种Reader实现
```

重要数据结构

beats通用事件结构(libbeat/beat/event.go):

```
type Event struct {
    Timestamp time.Time     // 收集日志时记录的时间戳，对应es文档中的@timestamp字段
    Meta      common.MapStr // meta信息，outpus可选的将其作为事件字段输出。比如输出为es且指定了pipeline时，其pipeline id就被包含在此字段中
    Fields    common.MapStr // 默认输出字段定义在field.yml，其他字段可以在通过fields配置项指定
    Private   interface{} // for beats private use
}
```

Crawler(filebeat/crawler/crawler.go):

```
// Crawler 负责抓取日志并发送到libbeat pipeline
type Crawler struct {
    inputs          map[uint64]*input.Runner // 包含所有输入的runner
    inputConfigs    []*common.Config
    out             channel.Factory
    wg              sync.WaitGroup
    InputsFactory   cfgfile.RunnerFactory
    ModulesFactory  cfgfile.RunnerFactory
    modulesReloader *cfgfile.Reloader
    inputReloader   *cfgfile.Reloader
    once            bool
    beatVersion     string
    beatDone        chan struct{}
}
```

log类型Input(filebeat/input/log/input.go)

```
// Input contains the input and its config
type Input struct {
    cfg           *common.Config
    config        config
    states        *file.States
    harvesters    *harvester.Registry   // 包含Input所有Harvester
    outlet        channel.Outleter      // Input共享的Publisher client
    stateOutlet   channel.Outleter
    done          chan struct{}
    numHarvesters atomic.Uint32
    meta          map[string]string
}
```

log类型Harvester(filebeat/input/log/harvester.go):

```
type Harvester struct {
    id     uuid.UUID
    config config
    source harvester.Source // the source being watched

    // shutdown handling
    done     chan struct{}
    stopOnce sync.Once
    stopWg   *sync.WaitGroup
    stopLock sync.Mutex

    // internal harvester state
    state  file.State
    states *file.States
    log    *Log

    // file reader pipeline
    reader          reader.Reader
    encodingFactory encoding.EncodingFactory
    encoding        encoding.Encoding

    // event/state publishing
    outletFactory OutletFactory
    publishState  func(*util.Data) bool

    onTerminate func()
}
```

Registrar(filebeat/registrar/registrar.go):

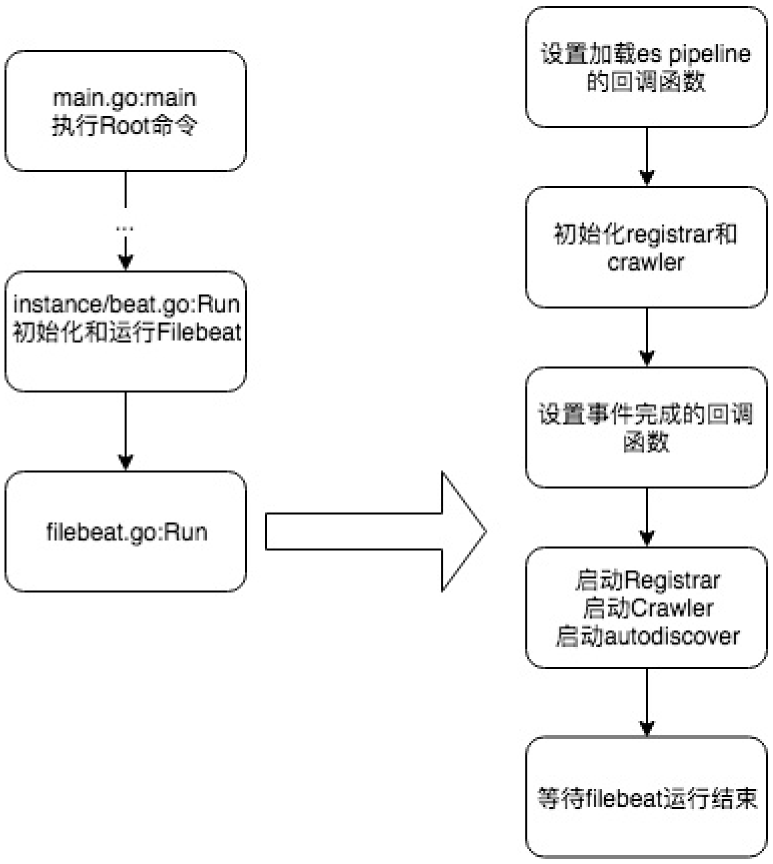
```
type Registrar struct {
    Channel      chan []file.State
    out          successLogger
    done         chan struct{}
    registryFile string      // Path to the Registry File
    fileMode     os.FileMode // Permissions to apply on the Registry File
    wg           sync.WaitGroup

    states               *file.States // Map with all file paths inside and the corresponding
 state
    gcRequired           bool         // gcRequired is set if registry state needs to be gc'ed
before the next write
    gcEnabled            bool         // gcEnabled indictes the registry contains some state t
hat can be gc'ed in the future
    flushTimeout         time.Duration
    bufferedStateUpdates int
}
```

libbeat Pipeline(libbeat/publisher/pipeline/pipeline.go)

```
type Pipeline struct {
    beatInfo beat.Info

    logger *logp.Logger
    queue  queue.Queue
    output *outputController

    observer observer

    eventer pipelineEventer

    // wait close support
    waitCloseMode    WaitCloseMode
    waitCloseTimeout time.Duration
    waitCloser       *waitCloser

    // pipeline ack
    ackMode    pipelineACKMode
    ackActive  atomic.Bool
    ackDone    chan struct{}
    ackBuilder ackBuilder // pipelineEventsACK
    eventSema  *sema

    processors pipelineProcessors
}
```

启动

filebeat启动流程图



每个 beat 的构建是独立的。从 filebeat 的入口文件filebeat/main.go可以看到，它向libbeat传递了名字、版本和构造函数来构造自身。跟着走到libbeat/beater/beater.go，我们可以看到程序的启动时的主要工作都是在这里完成的，包括命令行参数的处理、通用配置项的解析，以及最为重要的：调用象征一个beat的生命周期的若干方法

我们来看filebeat的启动过程。

1、执行root命令

在filebeat/main.go文件中，main函数调用了cmd.RootCmd.Execute()，而RootCmd则是在cmd/root.go中被init函数初始化，其中就注册了filebeat.go:New函数以创建实现了beater接口的filebeat实例

对于任意一个beats来说，都需要有：

- 实现Beater接口的具体Beater（如Filebeat）；
- 创建该具体Beater的(New)函数。

beater接口定义（beat/beat.go）：

```
type Beater interface {
    // The main event loop. This method should block until signalled to stop by an
    // invocation of the Stop() method.
    Run(b *Beat) error

    // Stop is invoked to signal that the Run method should finish its execution.
    // It will be invoked at most once.
    Stop()
}
```

2、初始化和运行Filebeat

- 创建libbeat/cmd/instance/beat.go:Beat结构
- 执行(*Beat).launch方法
    - (*Beat).Init() 初始化Beat：加载beats公共config
    - (*Beat).createBeater
    - registerTemplateLoading: 当输出为es时，注册加载es模板的回调函数
    - pipeline.Load: 创建Pipeline：包含队列、事件处理器、输出等
    - setupMetrics: 安装监控
    - filebeat.New: 解析配置(其中输入配置包括配置文件中的Input和module Input)等
- loadDashboards 加载kibana dashboard
- (*Filebeat).Run: 运行filebeat

3、Filebeat运行

- 设置加载es pipeline的回调函数
- 初始化registrar和crawler

- 设置事件完成的回调函数
- 启动Registrar、启动Crawler、启动Autodiscover
- 等待filebeat运行结束

我们再重代码看一下这个启动过程

```
main.go

    package main
    import (
        "os"
        "github.com/elastic/beats/filebeat/cmd"
    )
    func main() {
        if err := cmd.RootCmd.Execute(); err != nil {
            os.Exit(1)
        }
    }
```

进入到filebeat/cmd执行

```
package cmd

import (
    "flag"

    "github.com/spf13/pflag"

    "github.com/elastic/beats/filebeat/beater"

    cmd "github.com/elastic/beats/libbeat/cmd"
)

// Name of this beat
var Name = "filebeat"

// RootCmd to handle beats cli
var RootCmd *cmd.BeatsRootCmd

func init() {
    var runFlags = pflag.NewFlagSet(Name, pflag.ExitOnError)
    runFlags.AddGoFlag(flag.CommandLine.Lookup("once"))
    runFlags.AddGoFlag(flag.CommandLine.Lookup("modules"))

    RootCmd = cmd.GenRootCmdWithRunFlags(Name, "", beater.New, runFlags)
    RootCmd.PersistentFlags().AddGoFlag(flag.CommandLine.Lookup("M"))
    RootCmd.TestCmd.Flags().AddGoFlag(flag.CommandLine.Lookup("modules"))
    RootCmd.SetupCmd.Flags().AddGoFlag(flag.CommandLine.Lookup("modules"))
    RootCmd.AddCommand(cmd.GenModulesCmd(Name, "", buildModulesManager))
}
```

RootCmd 在这一句初始化

```
RootCmd = cmd.GenRootCmdWithRunFlags(Name, "", beater.New, runFlags)
```

beater.New跟进去看到是filebeat.go，这个函数会在后面进行调用，来创建filebeat结构体，传递filebeat相关的配置。

```
func New(b beat.Beat, rawConfig common.Config) (beat.Beater, error) {...}
```

现在进入GenRootCmdWithRunFlags方法，一路跟进去到GenRootCmdWithSettings，真正的初始化是在这个方法里面。

忽略前面的一段初始化值方法，看到RunCmd的初始化在：

```
rootCmd.RunCmd = genRunCmd(settings, beatCreator, runFlags)
```

进入getRunCmd，看到执行代码

```
err := instance.Run(settings, beatCreator)
```

跟到\elastic\beats\libbeat\cmd\instance\beat.go的Run方法

```
b, err := NewBeat(name, idxPrefix, version)
```

这里新建了beat结构体，同时将filebeat的New方法也传递了进来，就是参数beatCreator，我们可以看到在beat通过launch函数创建了filebeat结构体类型的beater

```
return b.launch(settings, bt)--->beater, err := b.createBeater(bt)--->beater, err := bt(&b.Bea
t, sub)
```

进入launch后，还做了很多的事情

1、还初始化了配置

```
err := b.InitWithSettings(settings)
```

2、pipeline的初始化

```
pipeline, err := pipeline.Load(b.Info,
        pipeline.Monitors{
            Metrics:   reg,
            Telemetry: monitoring.GetNamespace("state").GetRegistry(),
            Logger:    logp.L().Named("publisher"),
        },
        b.Config.Pipeline,
        b.processing,
        b.makeOutputFactory(b.Config.Output),
    )
```

在launch的末尾，还调用了beater启动方法，也就是filebeat的run函数

```
return beater.Run(&b.Beat)
```

因为启动的是filebeat，我们到filebeat.go的Run方法

```
func (fb *Filebeat) Run(b *beat.Beat) error {
	var err error
	config := fb.config

	if !fb.moduleRegistry.Empty() {
		err = fb.loadModulesPipelines(b)
		if err != nil {
			return err
		}
	}

	waitFinished := newSignalWait()
	waitEvents := newSignalWait()

	// count active events for waiting on shutdown
	wgEvents := &eventCounter{
		count: monitoring.NewInt(nil, "filebeat.events.active"),
		added: monitoring.NewUint(nil, "filebeat.events.added"),
		done:  monitoring.NewUint(nil, "filebeat.events.done"),
	}
	finishedLogger := newFinishedLogger(wgEvents)

	// Setup registrar to persist state
	registrar, err := registrar.New(config.RegistryFile, config.RegistryFilePermissions, co
nfig.RegistryFlush, finishedLogger)
	if err != nil {
		logp.Err("Could not init registrar: %v", err)
		return err
	}

	// Make sure all events that were published in
	registrarChannel := newRegistrarLogger(registrar)

	err = b.Publisher.SetACKHandler(beat.PipelineACKHandler{
		ACKEvents: newEventACKer(finishedLogger, registrarChannel).ackEvents,
	})
	if err != nil {
		logp.Err("Failed to install the registry with the publisher pipeline: %v", err)
		return err
	}

	outDone := make(chan struct{}) // outDone closes down all active pipeline connections
	crawler, err := crawler.New(
		channel.NewOutletFactory(outDone, wgEvents).Create,
		config.Inputs,
		b.Info.Version,
		fb.done,
		*once)
	if err != nil {
		logp.Err("Could not init crawler: %v", err)
		return err
	}

	// The order of starting and stopping is important. Stopping is inverted to the startin
g order.
	// The current order is: registrar, publisher, spooler, crawler
	// That means, crawler is stopped first.
	// Start the registrar
	err = registrar.Start()
	if err != nil {
		return fmt.Errorf("Could not start registrar: %v", err)
	}

	// Stopping registrar will write last state
	defer registrar.Stop()

	// Stopping publisher (might potentially drop items)
	defer func() {
		// Closes first the registrar logger to make sure not more events arrive at the
 registrar
		// registrarChannel must be closed first to potentially unblock (pretty unlikel
y) the publisher
		registrarChannel.Close()
		close(outDone) // finally close all active connections to publisher pipeline
	}()

	// Wait for all events to be processed or timeout
	defer waitEvents.Wait()

	// Create a ES connection factory for dynamic modules pipeline loading
	var pipelineLoaderFactory fileset.PipelineLoaderFactory
	if b.Config.Output.Name() == "elasticsearch" {
		pipelineLoaderFactory = newPipelineLoaderFactory(b.Config.Output.Config())
	} else {
		logp.Warn(pipelinesWarning)
	}

	if config.OverwritePipelines {
		logp.Debug("modules", "Existing Ingest pipelines will be updated")
	}

	err = crawler.Start(b.Publisher, registrar, config.ConfigInput, config.ConfigModules, p
ipelineLoaderFactory, config.OverwritePipelines)
	if err != nil {
		crawler.Stop()
		return err
	}

	// If run once, add crawler completion check as alternative to done signal
	if *once {
		runOnce := func() {
			logp.Info("Running filebeat once. Waiting for completion ...")
			crawler.WaitForCompletion()
			logp.Info("All data collection completed. Shutting down.")
		}
		waitFinished.Add(runOnce)
	}

	// Register reloadable list of inputs and modules
	inputs := cfgfile.NewRunnerList(management.DebugK, crawler.InputsFactory, b.Publisher)
	reload.Register.MustRegisterList("filebeat.inputs", inputs)

	modules := cfgfile.NewRunnerList(management.DebugK, crawler.ModulesFactory, b.Publisher
)
```

```
	reload.Register.MustRegisterList("filebeat.modules", modules)

	var adiscover *autodiscover.Autodiscover
	if fb.config.Autodiscover != nil {
		adapter := fbautodiscover.NewAutodiscoverAdapter(crawler.InputsFactory, crawler.
ModulesFactory)
		adiscover, err = autodiscover.NewAutodiscover("filebeat", b.Publisher, adapter,
 config.Autodiscover)
		if err != nil {
			return err
		}
	}
	adiscover.Start()

	// Add done channel to wait for shutdown signal
	waitFinished.AddChan(fb.done)
	waitFinished.Wait()

	// Stop reloadable lists, autodiscover -> Stop crawler -> stop inputs -> stop harvester
s
	// Note: waiting for crawlers to stop here in order to install wgEvents.Wait
	//       after all events have been enqueued for publishing. Otherwise wgEvents.Wait
	//       or publisher might panic due to concurrent updates.
	inputs.Stop()
	modules.Stop()
	adiscover.Stop()
	crawler.Stop()

	timeout := fb.config.ShutdownTimeout
	// Checks if on shutdown it should wait for all events to be published
	waitPublished := fb.config.ShutdownTimeout > 0 || *once
	if waitPublished {
		// Wait for registrar to finish writing registry
		waitEvents.Add(withLog(wgEvents.Wait,
			"Continue shutdown: All enqueued events being published."))
		// Wait for either timeout or all events having been ACKed by outputs.
		if fb.config.ShutdownTimeout > 0 {
			logp.Info("Shutdown output timer started. Waiting for max %v.", timeout)
			waitEvents.Add(withLog(waitDuration(timeout),
				"Continue shutdown: Time out waiting for events being published."
))
		} else {
			waitEvents.AddChan(fb.done)
		}
	}

	return nil
}
```

构造了registrar和crawler，用于监控文件状态变更和数据采集。然后

```
err = crawler.Start(b.Publisher, registrar, config.ConfigInput, config.ConfigModules, pipeline
LoaderFactory, config.OverwritePipelines)
```

crawler开始启动采集数据

```
for _, inputConfig := range c.inputConfigs {
	err := c.startInput(pipeline, inputConfig, r.GetStates())
	if err != nil {
		return err
	}
}
```

crawler的Start方法里面根据每个配置的输入调用一次startInput

```
func (c *Crawler) startInput(
	pipeline beat.Pipeline,
	config *common.Config,
	states []file.State,
) error {
	if !config.Enabled() {
		return nil
	}

	connector := channel.ConnectTo(pipeline, c.out)
	p, err := input.New(config, connector, c.beatDone, states, nil)
	if err != nil {
		return fmt.Errorf("Error in initing input: %s", err)
	}
	p.Once = c.once

	if _, ok := c.inputs[p.ID]; ok {
		return fmt.Errorf("Input with same ID already exists: %d", p.ID)
	}

	c.inputs[p.ID] = p

	p.Start()

	return nil
}
```

根据配置的input，构造log/input

```
func (p *Input) Run() {
	logp.Debug("input", "Start next scan")

	// TailFiles is like ignore_older = 1ns and only on startup
	if p.config.TailFiles {
		ignoreOlder := p.config.IgnoreOlder

		// Overwrite ignore_older for the first scan
		p.config.IgnoreOlder = 1
		defer func() {
			// Reset ignore_older after first run
			p.config.IgnoreOlder = ignoreOlder
			// Disable tail_files after the first run
			p.config.TailFiles = false
		}()
	}
	p.scan()

	// It is important that a first scan is run before cleanup to make sure all new states
 are read first
	if p.config.CleanInactive > 0 || p.config.CleanRemoved {
		beforeCount := p.states.Count()
		cleanedStates, pendingClean := p.states.Cleanup()
		logp.Debug("input", "input states cleaned up. Before: %d, After: %d, Pending: %
d",
			beforeCount, beforeCount-cleanedStates, pendingClean)
	}

	// Marking removed files to be cleaned up. Cleanup happens after next scan to make sure
all states are updated first
	if p.config.CleanRemoved {
		for _, state := range p.states.GetStates() {
			// os.Stat will return an error in case the file does not exist
			stat, err := os.Stat(state.Source)
			if err != nil {
				if os.IsNotExist(err) {
					p.removeState(state)
					logp.Debug("input", "Remove state for file as file removed:
%s", state.Source)
				} else {
					logp.Err("input state for %s was not removed: %s", state.So
urce, err)
				}
			} else {
				// Check if existing source on disk and state are the same. Remove
if not the case.
				newState := file.NewState(stat, state.Source, p.config.Type, p.met
a)
				if !newState.FileStateOS.IsSame(state.FileStateOS) {
					p.removeState(state)
					logp.Debug("input", "Remove state for file as file removed
 or renamed: %s", state.Source)
				}
			}
		}
	}
}
```

input开始根据配置的输入路径扫描所有符合的文件，并启动harvester

```
func (p *Input) scan() {
	var sortInfos []FileSortInfo
	var files []string

	paths := p.getFiles()

	var err error

	if p.config.ScanSort != "" {
		sortInfos, err = getSortedFiles(p.config.ScanOrder, p.config.ScanSort, getSortIn
fos(paths))
		if err != nil {
			logp.Err("Failed to sort files during scan due to error %s", err)
		}
	}

	if sortInfos == nil {
		files = getKeys(paths)
	}

	for i := 0; i < len(paths); i++ {

		var path string
		var info os.FileInfo

		if sortInfos == nil {
			path = files[i]
			info = paths[path]
		} else {
			path = sortInfos[i].path
			info = sortInfos[i].info
		}

		select {
		case <-p.done:
			logp.Info("Scan aborted because input stopped.")
			return
		default:
		}

		newState, err := getFileState(path, info, p)
		if err != nil {
			logp.Err("Skipping file %s due to error %s", path, err)
		}

		// Load last state
		lastState := p.states.FindPrevious(newState)

		// Ignores all files which fall under ignore_older
		if p.isIgnoreOlder(newState) {
			err := p.handleIgnoreOlder(lastState, newState)
			if err != nil {
				logp.Err("Updating ignore_older state error: %s", err)
			}
			continue
		}
```

```
			// Decides if previous state exists
			if lastState.IsEmpty() {
				logp.Debug("input", "Start harvester for new file: %s", newState.Source)
				err := p.startHarvester(newState, 0)
				if err == errHarvesterLimit {
					logp.Debug("input", harvesterErrMsg, newState.Source, err)
					continue
				}
				if err != nil {
					logp.Err(harvesterErrMsg, newState.Source, err)
				}
			} else {
				p.harvestExistingFile(newState, lastState)
			}
		}
}
```

在harvest的Run看到一个死循环读取message，预处理之后交由forwarder发送到目标输出

```
message, err := h.reader.Next()
h.sendEvent(data, forwarder)
```

至此，整个filebeat的启动到发送数据就理完了

配置文件解析

在libbeat中实现了通用的配置文件解析，在启动的过程中，在每次createbeater时候就会进行config。

调用 cfgfile.Load方法解析到cfg对象，进入load方法

```
func Load(path string, beatOverrides *common.Config) (*common.Config, error) {
	var config *common.Config
	var err error

	cfgpath := GetPathConfig()

	if path == "" {
		list := []string{}
		for _, cfg := range configfiles.List() {
			if !filepath.IsAbs(cfg) {
				list = append(list, filepath.Join(cfgpath, cfg))
			} else {
				list = append(list, cfg)
			}
		}
		config, err = common.LoadFiles(list...)
	} else {
		if !filepath.IsAbs(path) {
			path = filepath.Join(cfgpath, path)
		}
		config, err = common.LoadFile(path)
	}
	if err != nil {
		return nil, err
	}

	if beatOverrides != nil {
		config, err = common.MergeConfigs(
			defaults,
			beatOverrides,
			config,
			overwrites,
		)
		if err != nil {
			return nil, err
		}
	} else {
		config, err = common.MergeConfigs(
			defaults,
			config,
			overwrites,
		)
	}

	config.PrintDebugf("Complete configuration loaded:")
	return config, nil
}
```

如果不输入配置文件，使用configfiles定义文件

```
configfiles = common.StringArrFlag(nil, "c", "beat.yml", "Configuration file, relative to pat
h.config")
```

如果输入配置文件进入else分支

```
config, err = common.LoadFile(path)
```

根据配置文件构造config对象，使用的是yaml解析库。

```
c, err := yaml.NewConfigWithFile(path, configOpts...)
```

pipeline初始化

pipeline的初始化是在libbeat的创建对于的filebeat 的结构体的时候进行的在func (b *Beat) createBeater(bt beat.Creator) (beat.Beater, error) {}函数中

```
pipeline, err := pipeline.Load(b.Info,
    pipeline.Monitors{
        Metrics:   reg,
        Telemetry: monitoring.GetNamespace("state").GetRegistry(),
        Logger:    logp.L().Named("publisher"),
    },
    b.Config.Pipeline,
    b.processing,
    b.makeOutputFactory(b.Config.Output),
)
```

我们来看看load函数

```go
// Load uses a Config object to create a new complete Pipeline instance with
// configured queue and outputs.
func Load(
    beatInfo beat.Info,
    monitors Monitors,
    config Config,
    processors processing.Supporter,
    makeOutput func(outputs.Observer) (string, outputs.Group, error),
) (*Pipeline, error) {
    log := monitors.Logger
    if log == nil {
        log = logp.L()
    }

    if publishDisabled {
        log.Info("Dry run mode. All output types except the file based one are disabled.")
    }

    name := beatInfo.Name
    settings := Settings{
        WaitClose:     0,
        WaitCloseMode: NoWaitOnClose,
        Processors:    processors,
    }

    queueBuilder, err := createQueueBuilder(config.Queue, monitors)
    if err != nil {
        return nil, err
    }

    out, err := loadOutput(monitors, makeOutput)
    if err != nil {
        return nil, err
    }

    p, err := New(beatInfo, monitors, queueBuilder, out, settings)
    if err != nil {
        return nil, err
    }

    log.Infof("Beat name: %s", name)
    return p, err
}
```

主要是初始化queue，output，并创建对应的pipeline。

1、queue

```go
queueBuilder, err := createQueueBuilder(config.Queue, monitors)
    if err != nil {
        return nil, err
    }
```

进入createQueueBuilder

```
func createQueueBuilder(
    config common.ConfigNamespace,
    monitors Monitors,
) (func(queue.Eventer) (queue.Queue, error), error) {
    queueType := defaultQueueType
    if b := config.Name(); b != "" {
        queueType = b
    }

    queueFactory := queue.FindFactory(queueType)
    if queueFactory == nil {
        return nil, fmt.Errorf("'%v' is no valid queue type", queueType)
    }

    queueConfig := config.Config()
    if queueConfig == nil {
        queueConfig = common.NewConfig()
    }

    if monitors.Telemetry != nil {
        queueReg := monitors.Telemetry.NewRegistry("queue")
        monitoring.NewString(queueReg, "name").Set(queueType)
    }

    return func(eventer queue.Eventer) (queue.Queue, error) {
        return queueFactory(eventer, monitors.Logger, queueConfig)
    }, nil
}
```

根据queueType（有默认类型mem）找到创建的方法，一般mem就是

```
func init() {
    queue.RegisterType("mem", create)
}
```

看一下create函数

```
func create(eventer queue.Eventer, logger *logp.Logger, cfg *common.Config) (queue.Queue, erro
r) {
    config := defaultConfig
    if err := cfg.Unpack(&config); err != nil {
        return nil, err
    }

    if logger == nil {
        logger = logp.L()
    }

    return NewBroker(logger, Settings{
        Eventer:        eventer,
        Events:         config.Events,
        FlushMinEvents: config.FlushMinEvents,
        FlushTimeout:   config.FlushTimeout,
    }), nil
}
```

创建了一个broker

```go
// NewBroker creates a new broker based in-memory queue holding up to sz number of events.
// If waitOnClose is set to true, the broker will block on Close, until all internal
// workers handling incoming messages and ACKs have been shut down.
func NewBroker(
    logger logger,
    settings Settings,
) *Broker {
    // define internal channel size for producer/client requests
    // to the broker
    chanSize := 20

    var (
        sz           = settings.Events
        minEvents    = settings.FlushMinEvents
        flushTimeout = settings.FlushTimeout
    )

    if minEvents < 1 {
        minEvents = 1
    }
    if minEvents > 1 && flushTimeout <= 0 {
        minEvents = 1
        flushTimeout = 0
    }
    if minEvents > sz {
        minEvents = sz
    }

    if logger == nil {
        logger = logp.NewLogger("memqueue")
    }

    b := &Broker{
        done:   make(chan struct{}),
        logger: logger,

        // broker API channels
        events:    make(chan pushRequest, chanSize),
        requests:  make(chan getRequest),
        pubCancel: make(chan producerCancelRequest, 5),

        // internal broker and ACK handler channels
        acks:          make(chan int),
        scheduledACKs: make(chan chanList),

        waitOnClose: settings.WaitOnClose,

        eventer: settings.Eventer,
    }

    var eventLoop interface {
        run()
        processACK(chanList, int)
    }

    if minEvents > 1 {
        eventLoop = newBufferingEventLoop(b, sz, minEvents, flushTimeout)
    } else {
        eventLoop = newDirectEventLoop(b, sz)
    }

    b.bufSize = sz
    ack := newACKLoop(b, eventLoop.processACK)

    b.wg.Add(2)
    go func() {
        defer b.wg.Done()
        eventLoop.run()
    }()
    go func() {
        defer b.wg.Done()
        ack.run()
    }()

    return b
}
```

broker就是我们的queue，同时创建了一个eventLoop（根据是否有缓存创建不同的结构体，根据配置min_event是否大于1创建BufferingEventLoop或者DirectEventLoop，一般默认都是BufferingEventLoop，即带缓冲的队列。）和ack，调用他们的run函数进行监听

这边特别说明一下eventLoop的new，我们看带缓存的newBufferingEventLoop

```
func newBufferingEventLoop(b *Broker, size int, minEvents int, flushTimeout time.Duration) *bu
fferingEventLoop {
    l := &bufferingEventLoop{
        broker:       b,
        maxEvents:    size,
        minEvents:    minEvents,
        flushTimeout: flushTimeout,

        events:    b.events,
        get:       nil,
        pubCancel: b.pubCancel,
        acks:      b.acks,
    }
    l.buf = newBatchBuffer(l.minEvents)

    l.timer = time.NewTimer(flushTimeout)
    if !l.timer.Stop() {
        <-l.timer.C
    }

    return l
}
```

把broker的值很多都赋给了bufferingEventLoop，不知道为什么这么做。

```
go func() {
    defer b.wg.Done()
    eventLoop.run()
}()
go func() {
    defer b.wg.Done()
    ack.run()
}()
```

我们看一下有缓存的事件处理

```
func (l *bufferingEventLoop) run() {
    var (
        broker = l.broker
    )

    for {
        select {
        case <-broker.done:
            return

        case req := <-l.events: // producer pushing new event
            l.handleInsert(&req)

        case req := <-l.pubCancel: // producer cancelling active events
            l.handleCancel(&req)

        case req := <-l.get: // consumer asking for next batch
            l.handleConsumer(&req)

        case l.schedACKS <- l.pendingACKs:
            l.schedACKS = nil
            l.pendingACKs = chanList{}

        case count := <-l.acks:
            l.handleACK(count)

        case <-l.idleC:
            l.idleC = nil
            l.timer.Stop()
            if l.buf.length() > 0 {
                l.flushBuffer()
            }
        }
    }
}
```

到这里就可以监听队列中的事件了，BufferingEventLoop是一个实现了Broker、带有各种channel的结构，主要用于将日志发送至consumer消费。 BufferingEventLoop的run方法中，同样是一个无限循环，这里可以认为是一个日志事件的调度中心。

再来看看ack的调度

```go
func (l *ackLoop) run() {
    var (
        // log = l.broker.logger

        // Buffer up acked event counter in acked. If acked > 0, acks will be set to
        // the broker.acks channel for sending the ACKs while potentially receiving
        // new batches from the broker event loop.
        // This concurrent bidirectionally communication pattern requiring 'select'
        // ensures we can not have any deadlock between the event loop and the ack
        // loop, as the ack loop will not block on any channel
        acked int
        acks  chan int
    )

    for {
        select {
        case <-l.broker.done:
            // TODO: handle pending ACKs?
            // TODO: panic on pending batches?
            return

        case acks <- acked:
            acks, acked = nil, 0

        case lst := <-l.broker.scheduledACKs:
            count, events := lst.count()
            l.lst.concat(&lst)

            // log.Debug("ACK List:")
            // for current := l.lst.head; current != nil; current = current.next {
            //  log.Debugf("  ack entry(seq=%v, start=%v, count=%v",
            //      current.seq, current.start, current.count)
            // }

            l.batchesSched += uint64(count)
            l.totalSched += uint64(events)

        case <-l.sig:
            acked += l.handleBatchSig()
            if acked > 0 {
                acks = l.broker.acks
            }
        }

        // log.Debug("ackloop INFO")
        // log.Debug("ackloop:   total events scheduled = ", l.totalSched)
        // log.Debug("ackloop:   total events ack = ", l.totalACK)
        // log.Debug("ackloop:   total batches scheduled = ", l.batchesSched)
        // log.Debug("ackloop:   total batches ack = ", l.batchesACKed)

        l.sig = l.lst.channel()
        // if l.sig == nil {
        //  log.Debug("ackloop: no ack scheduled")
        // } else {
        //  log.Debug("ackloop: schedule ack: ", l.lst.head.seq)
        // }
    }
}
```

如果有处理信号就发送给regestry进行记录，关于registry在下面详细说明(/post/monitor/log/collect/filebeat/filebeat-principle/#registry和ack-机制)。

2、output

```
out, err := loadOutput(monitors, makeOutput)
if err != nil {
    return nil, err
}
```

进入loadOutput

```
func loadOutput(
    monitors Monitors,
    makeOutput OutputFactory,
) (outputs.Group, error) {
    log := monitors.Logger
    if log == nil {
        log = logp.L()
    }

    if publishDisabled {
        return outputs.Group{}, nil
    }

    if makeOutput == nil {
        return outputs.Group{}, nil
    }

    var (
        metrics  *monitoring.Registry
        outStats outputs.Observer
    )
    if monitors.Metrics != nil {
        metrics = monitors.Metrics.GetRegistry("output")
        if metrics != nil {
            metrics.Clear()
        } else {
            metrics = monitors.Metrics.NewRegistry("output")
        }
        outStats = outputs.NewStats(metrics)
    }

    outName, out, err := makeOutput(outStats)
    if err != nil {
        return outputs.Fail(err)
    }

    if metrics != nil {
        monitoring.NewString(metrics, "type").Set(outName)
    }
    if monitors.Telemetry != nil {
        telemetry := monitors.Telemetry.GetRegistry("output")
        if telemetry != nil {
            telemetry.Clear()
        } else {
            telemetry = monitors.Telemetry.NewRegistry("output")
        }
        monitoring.NewString(telemetry, "name").Set(outName)
    }

    return out, nil
}
```

这边是根据load传进来的makeOutput函数来进行创建的，我们看一下load这个参数。

```
func (b *Beat) makeOutputFactory(
    cfg common.ConfigNamespace,
) func(outputs.Observer) (string, outputs.Group, error) {
    return func(outStats outputs.Observer) (string, outputs.Group, error) {
        out, err := b.createOutput(outStats, cfg)
        return cfg.Name(), out, err
    }
}
```

创建createOutput

```
func (b *Beat) createOutput(stats outputs.Observer, cfg common.ConfigNamespace) (outputs.Group
, error) {
    if !cfg.IsSet() {
        return outputs.Group{}, nil
    }

    return outputs.Load(b.IdxSupporter, b.Info, stats, cfg.Name(), cfg.Config())
}
```

再看load

```
// Load creates and configures a output Group using a configuration object..
func Load(
    im IndexManager,
    info beat.Info,
    stats Observer,
    name string,
    config *common.Config,
) (Group, error) {
    factory := FindFactory(name)
    if factory == nil {
        return Group{}, fmt.Errorf("output type %v undefined", name)
    }

    if stats == nil {
        stats = NewNilObserver()
    }
    return factory(im, info, stats, config)
}
```

可见根据配置文件的配置的output的类型进行创建，比如我们用kafka做为output，我们看一下创建函数

```
func init() {
    sarama.Logger = kafkaLogger{}

    outputs.RegisterType("kafka", makeKafka)
}
```

就是makeKafka

```
func makeKafka(
    _ outputs.IndexManager,
    beat beat.Info,
    observer outputs.Observer,
    cfg *common.Config,
) (outputs.Group, error) {
    debugf("initialize kafka output")

    config, err := readConfig(cfg)
    if err != nil {
        return outputs.Fail(err)
    }

    topic, err := outil.BuildSelectorFromConfig(cfg, outil.Settings{
        Key:              "topic",
        MultiKey:         "topics",
        EnableSingleOnly: true,
        FailEmpty:        true,
    })
    if err != nil {
        return outputs.Fail(err)
    }

    libCfg, err := newSaramaConfig(config)
    if err != nil {
        return outputs.Fail(err)
    }

    hosts, err := outputs.ReadHostList(cfg)
    if err != nil {
        return outputs.Fail(err)
    }

    codec, err := codec.CreateEncoder(beat, config.Codec)
    if err != nil {
        return outputs.Fail(err)
    }

    client, err := newKafkaClient(observer, hosts, beat.IndexPrefix, config.Key, topic, codec,
libCfg)
    if err != nil {
        return outputs.Fail(err)
    }

    retry := 0
    if config.MaxRetries < 0 {
        retry = -1
    }
    return outputs.Success(config.BulkMaxSize, retry, client)
}
```

直接创建了kafka的client给发送的时候使用。

最后利用上面的两个构建函数来创建我们的pipeline

```
p, err := New(beatInfo, monitors, queueBuilder, out, settings)
if err != nil {
    return nil, err
}
```

其实上面函数有的调用是在这个new中进行的

```
// New create a new Pipeline instance from a queue instance and a set of outputs.
// The new pipeline will take ownership of queue and outputs. On Close, the
// queue and outputs will be closed.
func New(
    beat beat.Info,
    monitors Monitors,
    queueFactory queueFactory,
    out outputs.Group,
    settings Settings,
) (*Pipeline, error) {
    var err error

    if monitors.Logger == nil {
        monitors.Logger = logp.NewLogger("publish")
    }

    p := &Pipeline{
        beatInfo:         beat,
        monitors:         monitors,
        observer:         nilObserver,
        waitCloseMode:    settings.WaitCloseMode,
        waitCloseTimeout: settings.WaitClose,
        processors:       settings.Processors,
    }
    p.ackBuilder = &pipelineEmptyACK{p}
    p.ackActive = atomic.MakeBool(true)

    if monitors.Metrics != nil {
        p.observer = newMetricsObserver(monitors.Metrics)
    }
    p.eventer.observer = p.observer
    p.eventer.modifyable = true

    if settings.WaitCloseMode == WaitOnPipelineClose && settings.WaitClose > 0 {
        p.waitCloser = &waitCloser{}

        // waitCloser decrements counter on queue ACK (not per client)
        p.eventer.waitClose = p.waitCloser
    }

    p.queue, err = queueFactory(&p.eventer)
    if err != nil {
        return nil, err
    }

    if count := p.queue.BufferConfig().Events; count > 0 {
        p.eventSema = newSema(count)
    }

    maxEvents := p.queue.BufferConfig().Events
    if maxEvents <= 0 {
        // Maximum number of events until acker starts blocking.
        // Only active if pipeline can drop events.
        maxEvents = 64000
    }
    p.eventSema = newSema(maxEvents)
    p.output = newOutputController(beat, monitors, p.observer, p.queue)
    p.output.Set(out)

    return p, nil
}
```

创建一个output的控制器outputController

```
func newOutputController(
    beat beat.Info,
    monitors Monitors,
    observer outputObserver,
    b queue.Queue,
) *outputController {
    c := &outputController{
        beat:     beat,
        monitors: monitors,
        observer: observer,
        queue:    b,
    }

    ctx := &batchContext{}
    c.consumer = newEventConsumer(monitors.Logger, b, ctx)
    c.retryer = newRetryer(monitors.Logger, observer, nil, c.consumer)
    ctx.observer = observer
    ctx.retryer = c.retryer

    c.consumer.sigContinue()

    return c
}
```

同时初始化了eventConsumer和retryer。

```
func newEventConsumer(
    log *logp.Logger,
    queue queue.Queue,
    ctx *batchContext,
) *eventConsumer {
    c := &eventConsumer{
        logger: log,
        done:   make(chan struct{}),
        sig:    make(chan consumerSignal, 3),
        out:    nil,

        queue:    queue,
        consumer: queue.Consumer(),
        ctx:      ctx,
    }

    c.pause.Store(true)
    go c.loop(c.consumer)
    return c
}
```

在eventConsumer中启动了一个监听程序

```go
func (c *eventConsumer) loop(consumer queue.Consumer) {
    log := c.logger

    log.Debug("start pipeline event consumer")

    var (
        out    workQueue
        batch  *Batch
        paused = true
    )

    handleSignal := func(sig consumerSignal) {
        switch sig.tag {
        case sigConsumerCheck:

        case sigConsumerUpdateOutput:
            c.out = sig.out

        case sigConsumerUpdateInput:
            consumer = sig.consumer
        }

        paused = c.paused()
        if !paused && c.out != nil && batch != nil {
            out = c.out.workQueue
        } else {
            out = nil
        }
    }

    for {
        if !paused && c.out != nil && consumer != nil && batch == nil {
            out = c.out.workQueue
            queueBatch, err := consumer.Get(c.out.batchSize)
            if err != nil {
                out = nil
                consumer = nil
                continue
            }
            if queueBatch != nil {
                batch = newBatch(c.ctx, queueBatch, c.out.timeToLive)
            }

            paused = c.paused()
            if paused || batch == nil {
                out = nil
            }
        }

        select {
        case sig := <-c.sig:
            handleSignal(sig)
            continue
        default:
        }

        select {
```

```
        case <-c.done:
            log.Debug("stop pipeline event consumer")
            return
        case sig := <-c.sig:
            handleSignal(sig)
        case out <- batch:
            batch = nil
        }
    }
}
```

用于消费队列中的事件event，并将其构建成Batch，放到处理队列中。

```
func newBatch(ctx *batchContext, original queue.Batch, ttl int) *Batch {
    if original == nil {
        panic("empty batch")
    }

    b := batchPool.Get().(*Batch)
    *b = Batch{
        original: original,
        ctx:      ctx,
        ttl:      ttl,
        events:   original.Events(),
    }
    return b
}
```

再来看看retryer

```
func newRetryer(
    log *logp.Logger,
    observer outputObserver,
    out workQueue,
    c *eventConsumer,
) *retryer {
    r := &retryer{
        logger:     log,
        observer:   observer,
        done:       make(chan struct{}),
        sig:        make(chan retryerSignal, 3),
        in:         retryQueue(make(chan batchEvent, 3)),
        out:        out,
        consumer:   c,
        doneWaiter: sync.WaitGroup{},
    }
    r.doneWaiter.Add(1)
    go r.loop()
    return r
}
```

同样启动了监听程序，用于重试。

```
func (r *retryer) loop() {
    defer r.doneWaiter.Done()
    var (
        out             workQueue
        consumerBlocked bool

        active     *Batch
        activeSize int
        buffer     []*Batch
        numOutputs int

        log = r.logger
    )

    for {
        select {
        case <-r.done:
            return
        case evt := <-r.in:
            var (
                countFailed  int
                countDropped int
                batch        = evt.batch
                countRetry   = len(batch.events)
            )

            if evt.tag == retryBatch {
                countFailed = len(batch.events)
                r.observer.eventsFailed(countFailed)

                decBatch(batch)

                countRetry = len(batch.events)
                countDropped = countFailed - countRetry
                r.observer.eventsDropped(countDropped)
            }

            if len(batch.events) == 0 {
                log.Info("Drop batch")
                batch.Drop()
            } else {
                out = r.out
                buffer = append(buffer, batch)
                out = r.out
                active = buffer[0]
                activeSize = len(active.events)
                if !consumerBlocked {
                    consumerBlocked = blockConsumer(numOutputs, len(buffer))
                    if consumerBlocked {
                        log.Info("retryer: send wait signal to consumer")
                        r.consumer.sigWait()
                        log.Info("  done")
                    }
                }
            }

        case out <- active:
            r.observer.eventsRetry(activeSize)

            buffer = buffer[1:]
            active, activeSize = nil, 0

            if len(buffer) == 0 {
                out = nil
            } else {
                active = buffer[0]
                activeSize = len(active.events)
            }

            if consumerBlocked {
                consumerBlocked = blockConsumer(numOutputs, len(buffer))
                if !consumerBlocked {
                    log.Info("retryer: send unwait-signal to consumer")
                    r.consumer.sigUnWait()
                    log.Info("  done")
                }
            }

        case sig := <-r.sig:
            switch sig.tag {
            case sigRetryerUpdateOutput:
                r.out = sig.channel
            case sigRetryerOutputAdded:
                numOutputs++
            case sigRetryerOutputRemoved:
                numOutputs--
            }
        }
    }
}
```

然后对out进行了设置

```
func (c *outputController) Set(outGrp outputs.Group) {
    // create new outputGroup with shared work queue
    clients := outGrp.Clients
    queue := makeWorkQueue()
    worker := make([]outputWorker, len(clients))
    for i, client := range clients {
        worker[i] = makeClientWorker(c.observer, queue, client)
    }
    grp := &outputGroup{
        workQueue:  queue,
        outputs:    worker,
        timeToLive: outGrp.Retry + 1,
        batchSize:  outGrp.BatchSize,
    }

    // update consumer and retryer
    c.consumer.sigPause()
    if c.out != nil {
        for range c.out.outputs {
            c.retryer.sigOutputRemoved()
        }
    }
    c.retryer.updOutput(queue)
    for range clients {
        c.retryer.sigOutputAdded()
    }
    c.consumer.updOutput(grp)

    // close old group, so events are send to new workQueue via retryer
    if c.out != nil {
        for _, w := range c.out.outputs {
            w.Close()
        }
    }

    c.out = grp

    // restart consumer (potentially blocked by retryer)
    c.consumer.sigContinue()

    c.observer.updateOutputGroup()
}
```

这边就是对上面创建的kafka的每个client创建一个监控程序makeClientWorker

```
func makeClientWorker(observer outputObserver, qu workQueue, client outputs.Client) outputWork
er {
    if nc, ok := client.(outputs.NetworkClient); ok {
        c := &netClientWorker{observer: observer, qu: qu, client: nc}
        go c.run()
        return c
    }
    c := &clientWorker{observer: observer, qu: qu, client: client}
    go c.run()
    return c
}
```

然后就用于监控workQueue的数据，有数据就通过client的push发送到kafka，到这边pipeline的初始化也就结束了。

日志收集

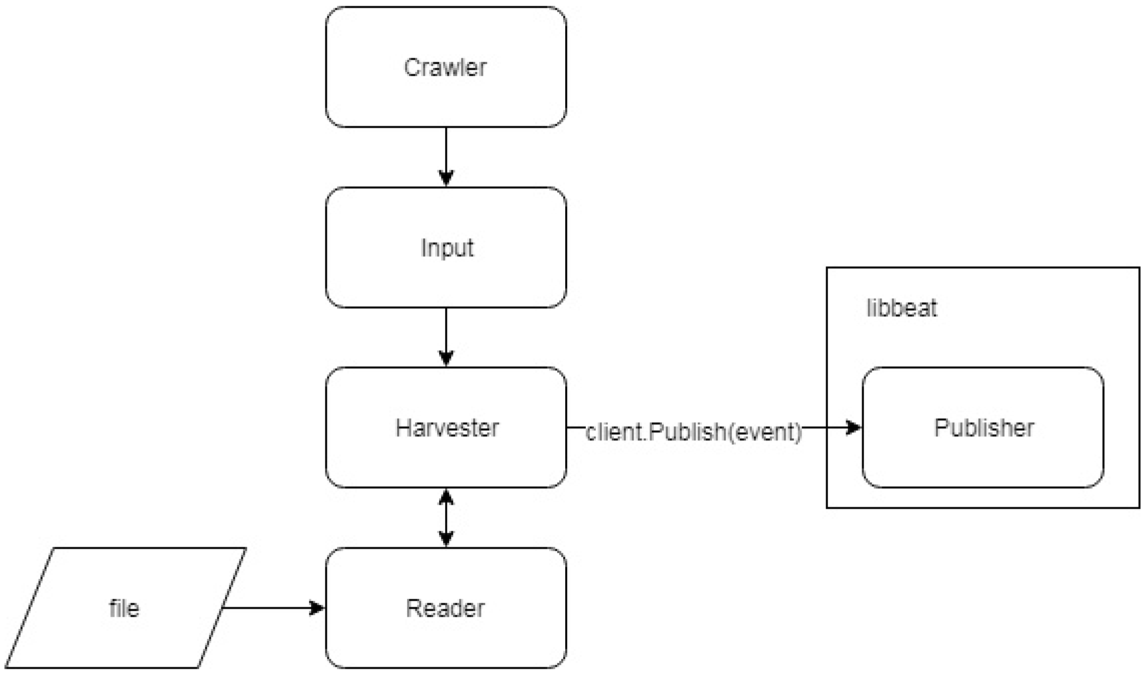
Filebeat 不仅支持普通文本日志的作为输入源，还内置支持了 redis 的慢查询日志、stdin、tcp 和 udp 等作为输入源。

本文只分析下普通文本日志的处理方式，对于普通文本日志，可以按照以下配置方式，指定 log 的输入源信息。

```
filebeat.inputs:
- type: log
  enabled: true
  paths:
    - /var/log/*.log
```

其中 Input 也可以指定多个, 每个 Input 下的 Log 也可以指定多个。

从收集日志、到发送事件到publisher，其数据流如下图所示：



filebeat 启动时会开启 Crawler，filebeat抽象出一个Crawler的结构体，对于配置中的每条 Input，Crawler 都会启动一个 Input 进行处理，代码如下所示：

```
func (c *Crawler) Start(...){
    ...
    for _, inputConfig := range c.inputConfigs {
        err := c.startInput(pipeline, inputConfig, r.GetStates())
        if err != nil {
            return err
        }
    }
    ...
}
```

然后就是创建input，比如我们采集的是log类型的，就是调用log的NewInput来创建，并且启动，定时进行扫描

```
p, err := input.New(config, connector, c.beatDone, states, nil)
```

会根据采集日志的类型来进行注册调用

```go
// NewInput instantiates a new Log
func NewInput(
    cfg *common.Config,
    outlet channel.Connector,
    context input.Context,
) (input.Input, error) {
    cleanupNeeded := true
    cleanupIfNeeded := func(f func() error) {
        if cleanupNeeded {
            f()
        }
    }

    // Note: underlying output.
    //  The input and harvester do have different requirements
    //  on the timings the outlets must be closed/unblocked.
    //  The outlet generated here is the underlying outlet, only closed
    //  once all workers have been shut down.
    //  For state updates and events, separate sub-outlets will be used.
    out, err := outlet.ConnectWith(cfg, beat.ClientConfig{
        Processing: beat.ProcessingConfig{
            DynamicFields: context.DynamicFields,
        },
    })
    if err != nil {
        return nil, err
    }
    defer cleanupIfNeeded(out.Close)

    // stateOut will only be unblocked if the beat is shut down.
    // otherwise it can block on a full publisher pipeline, so state updates
    // can be forwarded correctly to the registrar.
    stateOut := channel.CloseOnSignal(channel.SubOutlet(out), context.BeatDone)
    defer cleanupIfNeeded(stateOut.Close)

    meta := context.Meta
    if len(meta) == 0 {
        meta = nil
    }

    p := &Input{
        config:      defaultConfig,
        cfg:         cfg,
        harvesters:  harvester.NewRegistry(),
        outlet:      out,
        stateOutlet: stateOut,
        states:      file.NewStates(),
        done:        context.Done,
        meta:        meta,
    }

    if err := cfg.Unpack(&p.config); err != nil {
        return nil, err
    }
    if err := p.config.resolveRecursiveGlobs(); err != nil {
        return nil, fmt.Errorf("Failed to resolve recursive globs in config: %v", err)
    }
    if err := p.config.normalizeGlobPatterns(); err != nil {
        return nil, fmt.Errorf("Failed to normalize globs patterns: %v", err)
    }

    // Create empty harvester to check if configs are fine
    // TODO: Do config validation instead
    _, err = p.createHarvester(file.State{}, nil)
    if err != nil {
        return nil, err
    }

    if len(p.config.Paths) == 0 {
        return nil, fmt.Errorf("each input must have at least one path defined")
    }

    err = p.loadStates(context.States)
    if err != nil {
        return nil, err
    }

    logp.Info("Configured paths: %v", p.config.Paths)

    cleanupNeeded = false
    go p.stopWhenDone()

    return p, nil
}
```

最后会调用这个结构体的run函数进行扫描，主要是调用

```
p.scan()

// Scan starts a scanGlob for each provided path/glob
func (p *Input) scan() {
    var sortInfos []FileSortInfo
    var files []string

    paths := p.getFiles()

    var err error

    if p.config.ScanSort != "" {
        sortInfos, err = getSortedFiles(p.config.ScanOrder, p.config.ScanSort, getSortInfos(pa
ths))
        if err != nil {
            logp.Err("Failed to sort files during scan due to error %s", err)
        }
    }

    if sortInfos == nil {
        files = getKeys(paths)
    }

    for i := 0; i < len(paths); i++ {

        var path string
        var info os.FileInfo

        if sortInfos == nil {
            path = files[i]
            info = paths[path]
        } else {
            path = sortInfos[i].path
            info = sortInfos[i].info
        }

        select {
        case <-p.done:
            logp.Info("Scan aborted because input stopped.")
            return
        default:
        }

        newState, err := getFileState(path, info, p)
        if err != nil {
            logp.Err("Skipping file %s due to error %s", path, err)
        }



        // Load last state
        lastState := p.states.FindPrevious(newState)

        // Ignores all files which fall under ignore_older
        if p.isIgnoreOlder(newState) {
            logp.Debug("input","ignore")
            err := p.handleIgnoreOlder(lastState, newState)
            if err != nil {
                logp.Err("Updating ignore_older state error: %s", err)
            }
            //close(p.done)
            continue
        }

        // Decides if previous state exists
        if lastState.IsEmpty() {
            logp.Debug("input", "Start harvester for new file: %s", newState.Source)
            err := p.startHarvester(newState, 0)
            if err == errHarvesterLimit {
                logp.Debug("input", harvesterErrMsg, newState.Source, err)
                continue
            }
            if err != nil {
                logp.Err(harvesterErrMsg, newState.Source, err)
            }
        } else {
            p.harvestExistingFile(newState, lastState)
        }
    }
}
```

进行扫描过滤。由于指定的 paths 可以配置多个，而且可以是 Glob 类型，因此 Filebeat 将会匹配到多个配置文件。

根据配置获取匹配的日志文件，需要注意的是，这里的匹配方式并非正则，而是采用linux glob的规则，和正则还是有一些区别。

```
matches, err := filepath.Glob(path)
```

获取到了所有匹配的日志文件之后，会经过一些复杂的过滤，例如如果配置了exclude_files则会忽略这类文件，同时还会查询文件的状态，如果文件的最近一次修改时间大于ignore_older的配置，也会不去采集该文件。

还会对文件进行处理，获取每个文件的状态，构建新的state结构，

```
newState, err := getFileState(path, info, p)
if err != nil {
    logp.Err("Skipping file %s due to error %s", path, err)
}

func getFileState(path string, info os.FileInfo, p *Input) (file.State, error) {
    var err error
    var absolutePath string
    absolutePath, err = filepath.Abs(path)
    if err != nil {
        return file.State{}, fmt.Errorf("could not fetch abs path for file %s: %s", absolutePa
th, err)
    }
    logp.Debug("input", "Check file for harvesting: %s", absolutePath)
    // Create new state for comparison
    newState := file.NewState(info, absolutePath, p.config.Type, p.meta, p.cfg.GetField("broke
rlist"), p.cfg.GetField("topic"))
    return newState, nil
}
```

同时在已经存在的状态中进行对比，如果获取到对于的状态就不重新启动协程进行采集，如果获取一个新的状态就开启新的协程进行采集。

```
// Load last state
lastState := p.states.FindPrevious(newState)
```

Input对象创建时会从registry读取文件状态(主要是offset)， 对于每个匹配到的文件，都会开启一个 Harvester 进行逐行读取，每个 Harvester 都工作在自己的的 goroutine 中。

```
err := p.startHarvester(newState, 0)
if err == errHarvesterLimit {
    logp.Debug("input", harvesterErrMsg, newState.Source, err)
    continue
}
if err != nil {
    logp.Err(harvesterErrMsg, newState.Source, err)
}
```

我们来看看startHarvester

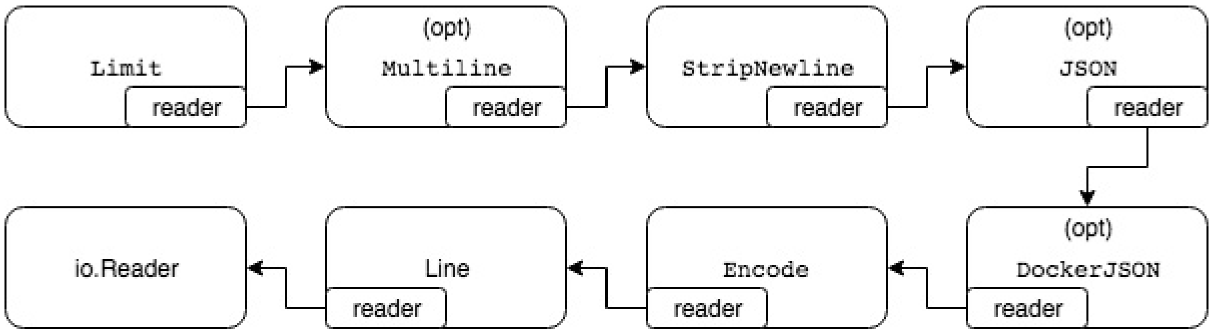
```
// startHarvester starts a new harvester with the given offset
// In case the HarvesterLimit is reached, an error is returned
func (p *Input) startHarvester(state file.State, offset int64) error {
    if p.numHarvesters.Inc() > p.config.HarvesterLimit && p.config.HarvesterLimit > 0 {
        p.numHarvesters.Dec()
        harvesterSkipped.Add(1)
        return errHarvesterLimit
    }
    // Set state to "not" finished to indicate that a harvester is running
    state.Finished = false
    state.Offset = offset

    // Create harvester with state
    h, err := p.createHarvester(state, func() { p.numHarvesters.Dec() })
    if err != nil {
        p.numHarvesters.Dec()
        return err
    }

    err = h.Setup()
    if err != nil {
        p.numHarvesters.Dec()
        return fmt.Errorf("error setting up harvester: %s", err)
    }

    // Update state before staring harvester
    // This makes sure the states is set to Finished: false
    // This is synchronous state update as part of the scan
    h.SendStateUpdate()

    if err = p.harvesters.Start(h); err != nil {
        p.numHarvesters.Dec()
    }
    return err
}
```

首先创建了Harvester

```
// createHarvester creates a new harvester instance from the given state
func (p *Input) createHarvester(state file.State, onTerminate func()) (*Harvester, error) {
    // Each wraps the outlet, for closing the outlet individually
    h, err := NewHarvester(
        p.cfg,
        state,
        p.states,
        func(state file.State) bool {
            return p.stateOutlet.OnEvent(beat.Event{Private: state})
        },
        subOutletWrap(p.outlet),
    )
    if err == nil {
        h.onTerminate = onTerminate
    }
    return h, err
}
```

harvester启动时会通过Setup方法创建一系列reader形成读处理链

关于log类型的reader处理链，如下图所示：



opt表示根据配置决定是否创建该reader

Reader包括：

- Line: 包含os.File，用于从指定offset开始读取日志行。虽然位于处理链的最内部，但其Next函数中实际的处理逻辑（读文件行）却是最新被执行的。
- Encode: 包含Line Reader，将其读取到的行生成Message结构后返回
- JSON, DockerJSON: 将json形式的日志内容decode成字段
- StripNewLine：去除日志行尾部的空白符
- Multiline: 用于读取多行日志
- Limit: 限制单行日志字节数

除了Line Reader外，这些reader都实现了Reader接口：

```
type Reader interface {
    Next() (Message, error)
}
```

Reader通过内部包含Reader对象的方式，使Reader形成一个处理链，其实这就是设计模式中的责任链模式。

各Reader的Next方法的通用形式像是这样：Next方法调用内部Reader对象的Next方法获取Message，然后处理后返回。

```
func (r *SomeReader) Next() (Message, error) {
    message, err := r.reader.Next()
    if err != nil {
        return message, err
    }

    // do some processing...

    return message, nil
}
```

其实Harvester 的工作流程非常简单，harvester从registry记录的文件位置开始读取，就是逐行读取文件，并更新该文件暂时在 Input 中的文件偏移量（注意，并不是 Registrar 中的偏移量），读取完成（读到文件的EOF末尾），组装成事件（beat.Event）后发给Publisher。主要是调用了Harvester的run方法，部分如下：

```
for {
    select {
    case <-h.done:
        return nil
    default:
    }

    message, err := h.reader.Next()
    if err != nil {
        switch err {
        case ErrFileTruncate:
            logp.Info("File was truncated. Begin reading file from offset 0: %s", h.state.Sour
ce)
            h.state.Offset = 0
            filesTruncated.Add(1)
        case ErrRemoved:
            logp.Info("File was removed: %s. Closing because close_removed is enabled.", h.sta
te.Source)
        case ErrRenamed:
            logp.Info("File was renamed: %s. Closing because close_renamed is enabled.", h.sta
te.Source)
        case ErrClosed:
            logp.Info("Reader was closed: %s. Closing.", h.state.Source)
        case io.EOF:
            logp.Info("End of file reached: %s. Closing because close_eof is enabled.", h.stat
e.Source)
        case ErrInactive:
            logp.Info("File is inactive: %s. Closing because close_inactive of %v reached.",
 h.state.Source, h.config.CloseInactive)
        case reader.ErrLineUnparsable:
            logp.Info("Skipping unparsable line in file: %v", h.state.Source)
            //line unparsable, go to next line
            continue
        default:
            logp.Err("Read line error: %v; File: %v", err, h.state.Source)
        }
        return nil
    }

    // Get copy of state to work on
    // This is important in case sending is not successful so on shutdown
    // the old offset is reported
    state := h.getState()
    startingOffset := state.Offset
    state.Offset += int64(message.Bytes)

    // Stop harvester in case of an error
    if !h.onMessage(forwarder, state, message, startingOffset) {
        return nil
    }

    // Update state of harvester as successfully sent
    h.state = state
}
```

可以看到，reader.Next()方法会不停的读取日志，如果没有返回异常，则发送日志数据到缓存队列中。

返回的异常有几种类型，除了读取到EOF外，还会有例如文件一段时间不活跃等情况发生会使harvester goroutine退出，不再采集该文件，并关闭文件句柄。 filebeat为了防止占据过多的采集日志文件的文件句柄，默认的close_inactive参数为5min，如果日志文件5min内没有被修改，上面代码会进入ErrInactive的case，之后该harvester goroutine会被关闭。 这种场景下还需要注意的是，如果某个文件日志采集中被移除了，但是由于此时被filebeat保持着文件句柄，文件占据的磁盘空间会被保留直到harvester goroutine结束。

不同的harvester goroutine采集到的日志数据都会发送至一个全局的队列queue中，queue的实现有两种：基于内存和基于磁盘的队列，目前基于磁盘的队列还是处于alpha阶段，filebeat默认启用的是基于内存的缓存队列。 每当队列中的数据缓存到一定的大小或者超过了定时的时间（默认1s)，会被注册的client从队列中消费，发送至配置的后端。目前可以设置的client有kafka、elasticsearch、redis等。

同时，我们需要考虑到，日志型的数据其实是在不断增长和变化的：

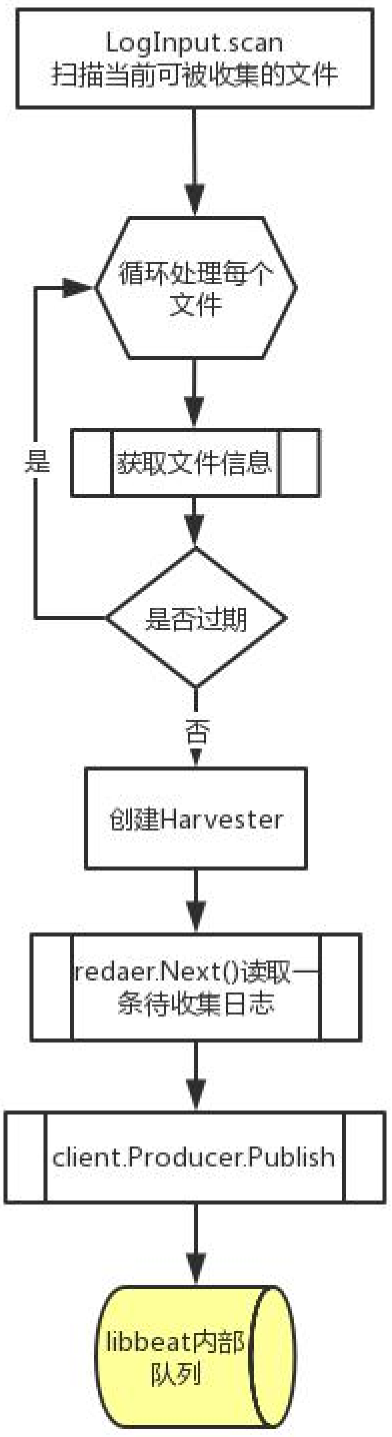
```
会有新的日志在不断产生
可能一个日志文件对应的 Harvester 退出后，又再次有了内容更新。
```

为了解决这两个情况，filebeat 采用了 Input 定时扫描的方式。代码如下，可以看出，Input 扫描的频率是由用户指定的 scan_frequency 配置来决定的 (默认 10s 扫描一次)。

```
func (p *Runner) Run() {
    p.input.Run()

    if p.Once {
        return
    }

    for {
        select {
        case <-p.done:
            logp.Info("input ticker stopped")
            return
        case <-time.After(p.config.ScanFrequency): // 定时扫描
            logp.Debug("input", "Run input")
            p.input.Run()
        }
    }
}
```

此外，如果用户启动时指定了 –once 选项，则扫描只会进行一次，就退出了。

使用一个简单的流程图可以这样表示



处理文件重命名，删除，截断

- 获取文件信息时会获取文件的device id + indoe作为文件的唯一标识;
- 文件收集进度会被持久化，这样当创建Harvester时，首先会对文件作openFile, 以 device id + inode为key在持久化文件中查看当前文件是否被收集过，收集到了什么位置，然后断点续传
- 在读取过程中，如果文件被截断，认为文件已经被同名覆盖，将从头开始读取文件

- 如果文件被删除，因为原文件已被打开，不影响继续收集，但如果设置了CloseRemoved，则不会再继续收集
- 如果文件被重命名，因为原文件已被打开，不影响继续收集，但如果设置了CloseRenamed，则不会再继续收集

pipeline调度

至此，我们可以清楚的知道，Filebeat 是如何采集日志文件。而日志采集过程，Harvest 会将数据写到 Pipeline 中。我们接下来看下数据是如何写入到 Pipeline 中的。

Haveseter 会将数据写入缓存中，而另一方面 Output 会从缓存将数据读走。整个生产消费的过程都是由 Pipeline 进行调度的，而整个调度过程也非常复杂。

不同的harvester goroutine采集到的日志数据都会发送至一个全局的队列queue中，Filebeat 的缓存queue目前分为 memqueue 和 spool。memqueue 顾名思义就是内存缓存，spool 则是将数据缓存到磁盘中。本文将基于 memqueue 讲解整个调度过程。

在下面的pipeline的写入和消费中，在client.go在(*client) publish方法中我们可以看到，事件是通过调用c.producer.Publish(pubEvent)被实际发送的，而producer则通过具体Queue的Producer方法生成。

队列对象被包含在pipeline.go:Pipeline结构中，其接口的定义如下：

```
type Queue interface {
    io.Closer
    BufferConfig() BufferConfig
    Producer(cfg ProducerConfig) Producer
    Consumer() Consumer
}
```

主要的，Producer方法生成Producer对象，用于向队列中push事件；Consumer方法生成Consumer对象，用于从队列中取出事件。Producer和Consumer接口定义如下：

```
type Producer interface {
    Publish(event publisher.Event) bool
    TryPublish(event publisher.Event) bool
    Cancel() int
}

type Consumer interface {
    Get(sz int) (Batch, error)
    Close() error
}
```

在配置中没有指定队列配置时，默认使用了memqueue作为队列实现，下面我们来看看memqueue及其对应producer和consumer定义：

Broker结构(memqueue在代码中实际对应的结构名是Broker)：

```
type Broker struct {
    done chan struct{}

    logger logger

    bufSize int
    // buf         brokerBuffer
    // minEvents   int
    // idleTimeout time.Duration

    // api channels
    events    chan pushRequest
    requests  chan getRequest
    pubCancel chan producerCancelRequest

    // internal channels
    acks          chan int
    scheduledACKs chan chanList

    eventer queue.Eventer

    // wait group for worker shutdown
    wg          sync.WaitGroup
    waitOnClose bool
}
```

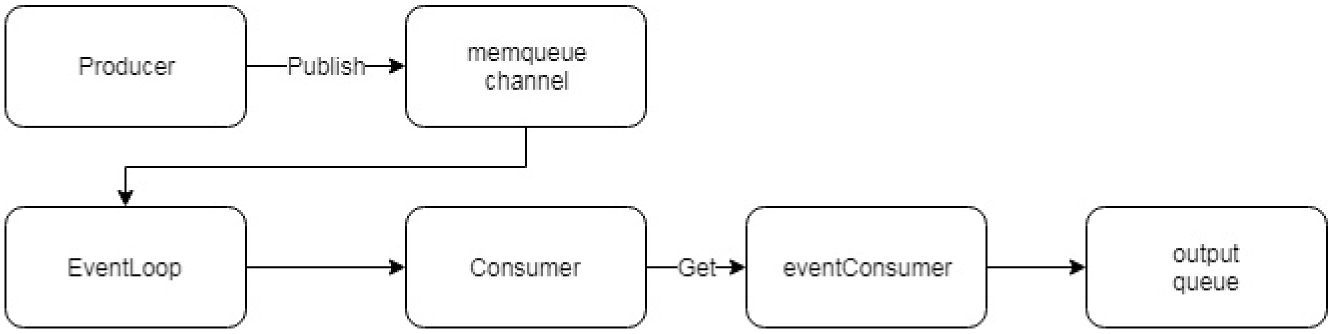
根据是否需要ack分为forgetfullProducer和ackProducer两种producer：

```
type forgetfullProducer struct {
    broker    *Broker
    openState openState
}

type ackProducer struct {
    broker    *Broker
    cancel    bool
    seq       uint32
    state     produceState
    openState openState
}
```

consumer结构:

```
type consumer struct {
    broker *Broker
    resp   chan getResponse

    done   chan struct{}
    closed atomic.Bool
}
```

queue、producer、consumer三者关系的运作方式如下图所示：



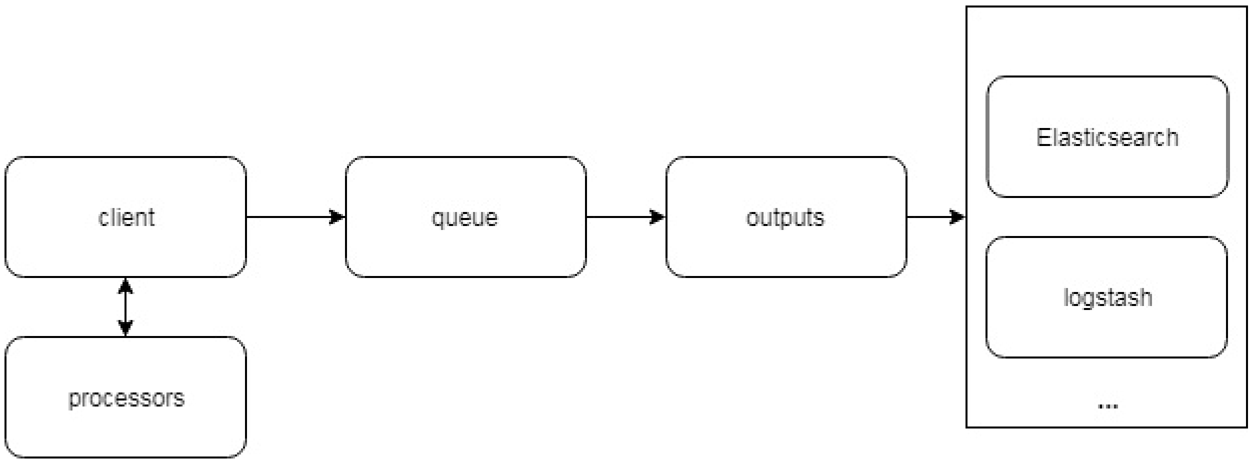
- Producer通过Publish或TryPublish事件放入Broker的队列，即结构中的channel对象evetns
- Broker的主事件循环EventLoop将（请求）事件从events channel取出，放入自身结构体对象ringBuffer中。主事件循环有两种类型：
    - 直接（不带buffer）事件循环结构directEventLoop：收到事件后尽可能快的转发；
    - 带buffer事件循环结构bufferingEventLoop：当buffer满或刷新超时时转发。具体使用哪一种取决于memqueue配置项flush.min_events，大于1时使用后者，否则使用前者。
- eventConsumer调用Consumer的Get方法获取事件：
    - 首先将获取事件请求（包括请求事件数和用于存放其响应事件的channel resp）放入Broker的请求队列requests中，等待主事件循环EventLoop处理后将事件放入resp；
    - 获取resp的事件，组装成batch结构后返回
- eventConsumer将事件放入output对应队列中

这部分关于事件在队列中各种channel间的流转，笔者认为是比较消耗性能的，但不清楚设计者这样设计的考量是什么。

另外值得思考的是，在多个go routine使用队列交互的场景下，libbeat中都使用了go语言channel作为其底层的队列，它是否可以完全替代加锁队列的使用呢？

Pipeline 的写入

在Crawler收集日志并转换成事件后，我们继续发送数据，其就会通过调用Publisher对应client的Publish接口将事件送到Publisher，后续的处理流程也都将由libbeat完成，事件的流转如下图所示：



我们首先看一下事件处理器processor

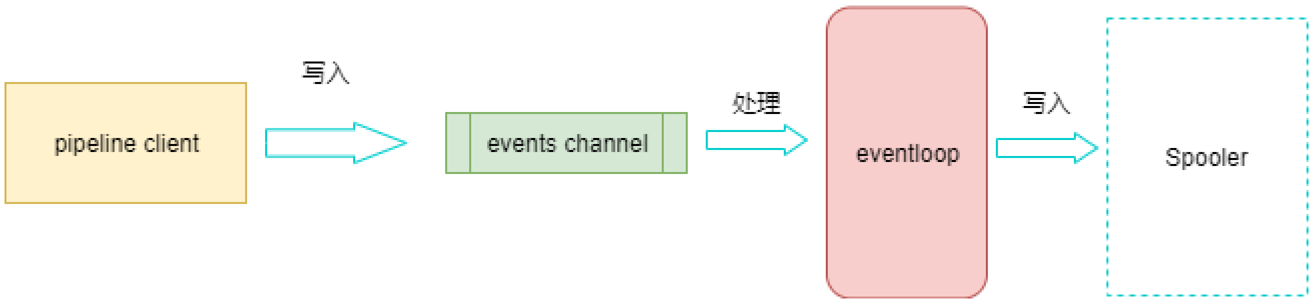
在harvester调用client.Publish接口时，其内部会使用配置中定义的processors对事件进行处理，然后才将事件发送到Publisher队列。

processor包含两种：在Input内定义作为局部（Input独享）的processor，其只对该Input产生的事件生效；在顶层配置中定义作为全局processor，其对全部事件生效。其对应的代码实现方式是：filebeat在使用libbeat pipeline的ConnectWith接口创建client时（factory.go中(*OutletFactory)Create函数），会将Input内部的定义processor作为参数传递给ConnectWith接口。而在ConnectWith实现中，会将参数中的processor和全局processor（在创建pipeline时生成）合并。从这里读者也可以发现，实际上每个Input都独享一个client，其包含一些Input自身的配置定义逻辑。

任一Processor都实现了Processor接口：Run函数包含处理逻辑，String返回Processor名。

```
type Processor interface {
    Run(event *beat.Event) (*beat.Event, error)
    String() string
}
```

我们再看下 Haveseter 是如何将数据写入缓存中的，如下图所示：



Harvester 通过 pipeline 提供的 pipelineClient 将数据写入到 pipeline 中，Haveseter 会将读到的数据会包装成一个 Event 结构体，再递交给 pipeline。

在 Filebeat 的实现中，pipelineClient 并不直接操作缓存，而是将 event 先写入一个 events channel 中。

同时，有一个 eventloop 组件，会监听 events channel 的事件到来，等 event 到达时，eventloop 会将其放入缓存中。

当缓存满的时候，eventloop 直接移除对该 channel 的监听。

每次 event ACK 或者取消后，缓存不再满了，则 eventloop 会重新监听 events channel。

```go
// onMessage processes a new message read from the reader.
// This results in a state update and possibly an event would be send.
// A state update first updates the in memory state held by the prospector,
// and finally sends the file.State indirectly to the registrar.
// The events Private field is used to forward the file state update.
//
// onMessage returns 'false' if it was interrupted in the process of sending the event.
// This normally signals a harvester shutdown.
func (h *Harvester) onMessage(
	forwarder *harvester.Forwarder,
	state file.State,
	message reader.Message,
	messageOffset int64,
) bool {
	if h.source.HasState() {
		h.states.Update(state)
	}

	text := string(message.Content)
	if message.IsEmpty() || !h.shouldExportLine(text) {
		// No data or event is filtered out -> send empty event with state update
		// only. The call can fail on filebeat shutdown.
		// The event will be filtered out, but forwarded to the registry as is.
		err := forwarder.Send(beat.Event{Private: state})
		return err == nil
	}

	fields := common.MapStr{
		"log": common.MapStr{
			"offset": messageOffset, // Offset here is the offset before the starting char.
			"file": common.MapStr{
				"path": state.Source,
			},
		},
	}
	fields.DeepUpdate(message.Fields)

	// Check if json fields exist
	var jsonFields common.MapStr
	if f, ok := fields["json"]; ok {
		jsonFields = f.(common.MapStr)
	}

	var meta common.MapStr
	timestamp := message.Ts
	if h.config.JSON != nil && len(jsonFields) > 0 {
		id, ts := readjson.MergeJSONFields(fields, jsonFields, &text, *h.config.JSON)
		if !ts.IsZero() {
			// there was a `@timestamp` key in the event, so overwrite
			// the resulting timestamp
			timestamp = ts
		}

		if id != "" {
			meta = common.MapStr{
				"id": id,
			}
```

```
        }
    } else if &text != nil {
        if fields == nil {
            fields = common.MapStr{}
        }
        fields["message"] = text
    }

    err := forwarder.Send(beat.Event{
        Timestamp: timestamp,
        Fields:    fields,
        Meta:      meta,
        Private:   state,
    })
    return err == nil
}
```

将数据包装成event直接通过send方法将数据发出去

```
// Send updates the input state and sends the event to the spooler
// All state updates done by the input itself are synchronous to make sure no states are overw
ritten
func (f *Forwarder) Send(event beat.Event) error {
    ok := f.Outlet.OnEvent(event)
    if !ok {
        logp.Info("Input outlet closed")
        return errors.New("input outlet closed")
    }

    return nil
}
```

调用Outlet.OnEvent发送data

点进去发现是一个接口

```
type Outlet interface {
       OnEvent(data *util.Data) bool
}
```

经过调试观察，elastic\beats\filebeat\channel\outlet.go实现了这个接口

outlet在Harvester的run一开始就创建了

```
outlet := channel.CloseOnSignal(h.outletFactory(), h.done)
forwarder := harvester.NewForwarder(outlet)
```

所以调用的OnEvent

```
func (o *outlet) OnEvent(event beat.Event) bool {
    if !o.isOpen.Load() {
        return false
    }

    if o.wg != nil {
        o.wg.Add(1)
    }

    o.client.Publish(event)

    // Note: race condition on shutdown:
    //  The underlying beat.Client is asynchronous. Without proper ACK
    //  handler we can not tell if the event made it 'through' or the client
    //  close has been completed before sending. In either case,
    //  we report 'false' here, indicating the event eventually being dropped.
    //  Returning false here, prevents the harvester from updating the state
    //  to the most recently published events. Therefore, on shutdown the harvester
    //  might report an old/outdated state update to the registry, overwriting the
    //  most recently
    //  published offset in the registry on shutdown.
    return o.isOpen.Load()
}
```

通过client.Publish发送数据，client也是一个接口

```
type Client interface {
       Publish(Event)
       PublishAll([]Event)
       Close() error
}
```

调试之后，client使用的是elastic\beats\libbeat\publisher\pipeline\client.go的client对象

我们来看一下这个client是通过Harvester的参数outletFactory来初始化的，我们来看一下NewHarvester初始化的时候也就是在createHarvester的时候传递的参数

```
// createHarvester creates a new harvester instance from the given state
func (p *Input) createHarvester(state file.State, onTerminate func()) (*Harvester, error) {
    // Each wraps the outlet, for closing the outlet individually
    h, err := NewHarvester(
        p.cfg,
        state,
        p.states,
        func(state file.State) bool {
            return p.stateOutlet.OnEvent(beat.Event{Private: state})
        },
        subOutletWrap(p.outlet),
    )
    if err == nil {
        h.onTerminate = onTerminate
    }
    return h, err
}
```

就是subOutletWrap中的参数p.outlet那就要看以下Input初始化的的时候NewInput传递的参数

```
// NewInput instantiates a new Log
func NewInput(
    cfg *common.Config,
    outlet channel.Connector,
    context input.Context,
) (input.Input, error) {
    cleanupNeeded := true
    cleanupIfNeeded := func(f func() error) {
        if cleanupNeeded {
            f()
        }
    }

    // Note: underlying output.
    //  The input and harvester do have different requirements
    //  on the timings the outlets must be closed/unblocked.
    //  The outlet generated here is the underlying outlet, only closed
    //  once all workers have been shut down.
    //  For state updates and events, separate sub-outlets will be used.
    out, err := outlet.ConnectWith(cfg, beat.ClientConfig{
        Processing: beat.ProcessingConfig{
            DynamicFields: context.DynamicFields,
        },
    })
    if err != nil {
        return nil, err
    }
    defer cleanupIfNeeded(out.Close)

    // stateOut will only be unblocked if the beat is shut down.
    // otherwise it can block on a full publisher pipeline, so state updates
    // can be forwarded correctly to the registrar.
    stateOut := channel.CloseOnSignal(channel.SubOutlet(out), context.BeatDone)
    defer cleanupIfNeeded(stateOut.Close)

    meta := context.Meta
    if len(meta) == 0 {
        meta = nil
    }

    p := &Input{
        config:      defaultConfig,
        cfg:         cfg,
        harvesters:  harvester.NewRegistry(),
        outlet:      out,
        stateOutlet: stateOut,
        states:      file.NewStates(),
        done:        context.Done,
        meta:        meta,
    }

    if err := cfg.Unpack(&p.config); err != nil {
        return nil, err
    }
    if err := p.config.resolveRecursiveGlobs(); err != nil {
        return nil, fmt.Errorf("Failed to resolve recursive globs in config: %v", err)
    }
    if err := p.config.normalizeGlobPatterns(); err != nil {
        return nil, fmt.Errorf("Failed to normalize globs patterns: %v", err)
    }

    // Create empty harvester to check if configs are fine
    // TODO: Do config validation instead
    _, err = p.createHarvester(file.State{}, nil)
    if err != nil {
        return nil, err
    }

    if len(p.config.Paths) == 0 {
        return nil, fmt.Errorf("each input must have at least one path defined")
    }

    err = p.loadStates(context.States)
    if err != nil {
        return nil, err
    }

    logp.Info("Configured paths: %v", p.config.Paths)

    cleanupNeeded = false
    go p.stopWhenDone()

    return p, nil
}
```

主要看outlet，这个是在Crawler的startInput的时候进行初始化

```
func (c *Crawler) startInput(
    pipeline beat.Pipeline,
    config *common.Config,
    states []file.State,
) error {
    if !config.Enabled() {
        return nil
    }

    connector := c.out(pipeline)
    p, err := input.New(config, connector, c.beatDone, states, nil)
    if err != nil {
        return fmt.Errorf("Error while initializing input: %s", err)
    }
    p.Once = c.once

    if _, ok := c.inputs[p.ID]; ok {
        return fmt.Errorf("Input with same ID already exists: %d", p.ID)
    }

    c.inputs[p.ID] = p

    p.Start()

    return nil
}
```

看out就是crawler创建new的时候传递的值

```
crawler, err := crawler.New(
        channel.NewOutletFactory(outDone, wgEvents, b.Info).Create,
        config.Inputs,
        b.Info.Version,
        fb.done,
        *once)
```

就是create返回的pipelineConnector结构体

```
func (f *OutletFactory) Create(p beat.Pipeline) Connector {
    return &pipelineConnector{parent: f, pipeline: p}
}
```

看pipelineConnector的ConnectWith函数

```
func (c *pipelineConnector) ConnectWith(cfg *common.Config, clientCfg beat.ClientConfig) (Outl
eter, error) {
    config := inputOutletConfig{}
    if err := cfg.Unpack(&config); err != nil {
        return nil, err
    }

    procs, err := processorsForConfig(c.parent.beatInfo, config, clientCfg)
    if err != nil {
        return nil, err
    }

    setOptional := func(to common.MapStr, key string, value string) {
        if value != "" {
            to.Put(key, value)
        }
    }

    meta := clientCfg.Processing.Meta.Clone()
    fields := clientCfg.Processing.Fields.Clone()

    serviceType := config.ServiceType
    if serviceType == "" {
        serviceType = config.Module
    }

    setOptional(meta, "pipeline", config.Pipeline)
    setOptional(fields, "fileset.name", config.Fileset)
    setOptional(fields, "service.type", serviceType)
    setOptional(fields, "input.type", config.Type)
    if config.Module != "" {
        event := common.MapStr{"module": config.Module}
        if config.Fileset != "" {
            event["dataset"] = config.Module + "." + config.Fileset
        }
        fields["event"] = event
    }

    mode := clientCfg.PublishMode
    if mode == beat.DefaultGuarantees {
        mode = beat.GuaranteedSend
    }

    // connect with updated configuration
    clientCfg.PublishMode = mode
    clientCfg.Processing.EventMetadata = config.EventMetadata
    clientCfg.Processing.Meta = meta
    clientCfg.Processing.Fields = fields
    clientCfg.Processing.Processor = procs
    clientCfg.Processing.KeepNull = config.KeepNull
    client, err := c.pipeline.ConnectWith(clientCfg)
    if err != nil {
        return nil, err
    }

    outlet := newOutlet(client, c.parent.wgEvents)
    if c.parent.done != nil {
        return CloseOnSignal(outlet, c.parent.done), nil
    }
    return outlet, nil
}
```

这边获取到了pipeline的客户端client

```go
// ConnectWith create a new Client for publishing events to the pipeline.
// The client behavior on close and ACK handling can be configured by setting
// the appropriate fields in the passed ClientConfig.
// If not set otherwise the defaut publish mode is OutputChooses.
func (p *Pipeline) ConnectWith(cfg beat.ClientConfig) (beat.Client, error) {
    var (
        canDrop      bool
        dropOnCancel bool
        eventFlags   publisher.EventFlags
    )

    err := validateClientConfig(&cfg)
    if err != nil {
        return nil, err
    }

    p.eventer.mutex.Lock()
    p.eventer.modifyable = false
    p.eventer.mutex.Unlock()

    switch cfg.PublishMode {
    case beat.GuaranteedSend:
        eventFlags = publisher.GuaranteedSend
        dropOnCancel = true
    case beat.DropIfFull:
        canDrop = true
    }

    waitClose := cfg.WaitClose
    reportEvents := p.waitCloser != nil

    switch p.waitCloseMode {
    case NoWaitOnClose:

    case WaitOnClientClose:
        if waitClose <= 0 {
            waitClose = p.waitCloseTimeout
        }
    }

    processors, err := p.createEventProcessing(cfg.Processing, publishDisabled)
    if err != nil {
        return nil, err
    }

    client := &client{
        pipeline:     p,
        closeRef:     cfg.CloseRef,
        done:         make(chan struct{}),
        isOpen:       atomic.MakeBool(true),
        eventer:      cfg.Events,
        processors:   processors,
        eventFlags:   eventFlags,
        canDrop:      canDrop,
        reportEvents: reportEvents,
    }
    acker := p.makeACKer(processors != nil, &cfg, waitClose, client.unlink)
    producerCfg := queue.ProducerConfig{
        // Cancel events from queue if acker is configured
        // and no pipeline-wide ACK handler is registered.
        DropOnCancel: dropOnCancel && acker != nil && p.eventer.cb == nil,
    }

    if reportEvents || cfg.Events != nil {
        producerCfg.OnDrop = func(event beat.Event) {
            if cfg.Events != nil {
                cfg.Events.DroppedOnPublish(event)
            }
            if reportEvents {
                p.waitCloser.dec(1)
            }
        }
    }

    if acker != nil {
        producerCfg.ACK = acker.ackEvents
    } else {
        acker = newCloseACKer(nilACKer, client.unlink)
    }

    client.acker = acker
    client.producer = p.queue.Producer(producerCfg)

    p.observer.clientConnected()

    if client.closeRef != nil {
        p.registerSignalPropagation(client)
    }

    return client, nil
}
```

调用的就是client的Publish函数来发送数据，publish方法即发送日志的方法，如果需要在发送前改造日志格式，可在这里添加代码，如下面的解析日志代码。

```go
func (c *client) Publish(e beat.Event) {
    c.mutex.Lock()
    defer c.mutex.Unlock()

    c.publish(e)
}

func (c *client) publish(e beat.Event) {
    var (
        event   = &e
        publish = true
        log     = c.pipeline.monitors.Logger
    )

    c.onNewEvent()

    if !c.isOpen.Load() {
        // client is closing down -> report event as dropped and return
        c.onDroppedOnPublish(e)
        return
    }

    if c.processors != nil {
        var err error

        event, err = c.processors.Run(event)
        publish = event != nil
        if err != nil {
            // TODO: introduce dead-letter queue?

            log.Errorf("Failed to publish event: %v", err)
        }
    }

    if event != nil {
        e = *event
    }

    open := c.acker.addEvent(e, publish)
    if !open {
        // client is closing down -> report event as dropped and return
        c.onDroppedOnPublish(e)
        return
    }

    if !publish {
        c.onFilteredOut(e)
        return
    }

    e = *event
    pubEvent := publisher.Event{
        Content: e,
        Flags:   c.eventFlags,
    }

    if c.reportEvents {
        c.pipeline.waitCloser.inc()
    }

    var published bool
    if c.canDrop {
        published = c.producer.TryPublish(pubEvent)
    } else {
        published = c.producer.Publish(pubEvent)
    }

    if published {
        c.onPublished()
    } else {
        c.onDroppedOnPublish(e)
        if c.reportEvents {
            c.pipeline.waitCloser.dec(1)
        }
    }
}
```

在上面创建clinet的时候，创建了队列的生产者，也就是之前broker的Producer

```
func (b *Broker) Producer(cfg queue.ProducerConfig) queue.Producer {
    return newProducer(b, cfg.ACK, cfg.OnDrop, cfg.DropOnCancel)
}

func newProducer(b *Broker, cb ackHandler, dropCB func(beat.Event), dropOnCancel bool) queue.P
roducer {
    openState := openState{
        log:    b.logger,
        isOpen: atomic.MakeBool(true),
        done:   make(chan struct{}),
        events: b.events,
    }

    if cb != nil {
        p := &ackProducer{broker: b, seq: 1, cancel: dropOnCancel, openState: openState}
        p.state.cb = cb
        p.state.dropCB = dropCB
        return p
    }
    return &forgetfulProducer{broker: b, openState: openState}
}
```

也就是forgetfulProducer结构体，调用这个的Publish函数来发送数据

```
func (p *forgetfulProducer) Publish(event publisher.Event) bool {
    return p.openState.publish(p.makeRequest(event))
}

func (st *openState) publish(req pushRequest) bool {
    select {
    case st.events <- req:
        return true
    case <-st.done:
        st.events = nil
        return false
    }
}
```

将数据放到了forgetfulProducer的openState的events中。到此数据就算发送到pipeline中了。

上文在pipeline的初始化的时候，queue初始化一般默认都是BufferingEventLoop，即带缓冲的队列。BufferingEventLoop是一个实现了Broker、带有各种channel的结构，主要用于将日志发送至consumer消费。 BufferingEventLoop的run方法中，同样是一个无限循环，这里可以认为是一个日志事件的调度中心。

```
for {
        select {
        case <-broker.done:
            return
        case req := <-l.events: // producer pushing new event
            l.handleInsert(&req)
        case req := <-l.get: // consumer asking for next batch
            l.handleConsumer(&req)
        case count := <-l.acks:
            l.handleACK(count)
        case <-l.idleC:
            l.idleC = nil
            l.timer.Stop()
            if l.buf.length() > 0 {
                l.flushBuffer()
            }
        }
    }
```

上文中harvester goroutine每次读取到日志数据之后，最终会被发送至bufferingEventLoop中的events chan pushRequest 的channel中，然后触发上面req := <-l.events的case，handleInsert方法会把数据添加至bufferingEventLoop的buf中，buf即memqueue实际缓存日志数据的队列，如果buf长度超过配置的最大值或者bufferingEventLoop中的timer定时器（默认1S）触发了case <-l.idleC，均会调用flushBuffer()方法。 flushBuffer()又会触发req := <-l.get的case，然后运行handleConsumer方法，该方法中最重要的是这一句代码：

```
req.resp <- getResponse{ackChan, events}
```

这里获取到了consumer消费者的response channel，然后发送数据给这个channel。真正到这，才会触发consumer对memqueue的消费。所以，其实memqueue并非一直不停的在被consumer消费，而是在memqueue通知consumer的时候才被消费，我们可以理解为一种脉冲式的发送

简单的来说就是，每当队列中的数据缓存到一定的大小或者超过了定时的时间（默认1s)，会被注册的client从队列中消费，发送至配置的后端。

以上是 Pipeline 的写入过程，此时 event 已被写入到了缓存中。

但是 Output 是如何从缓存中拿到 event 数据的?

Pipeline 的消费过程

在上文已经提到过，filebeat初始化的时候，就已经创建了一个eventConsumer并在loop无限循环方法里试图从Broker中其实也就是上面的resp中获取日志数据。

```
for {
        if !paused && c.out != nil && consumer != nil && batch == nil {
            out = c.out.workQueue
            queueBatch, err := consumer.Get(c.out.batchSize)
            ...
            batch = newBatch(c.ctx, queueBatch, c.out.timeToLive)
        }
        ...
        select {
        case <-c.done:
            return
        case sig := <-c.sig:
            handleSignal(sig)
        case out <- batch:
            batch = nil
        }
    }
```

上面consumer.Get就是消费者consumer从Broker中获取日志数据，然后发送至out的channel中被output client发送，我们看一下Get方法里的核心代码：

```
select {
    case c.broker.requests <- getRequest{sz: sz, resp: c.resp}:
    case <-c.done:
        return nil, io.EOF
    }

    // if request has been send, we do have to wait for a response
    resp := <-c.resp
    return &batch{
        consumer: c,
        events:   resp.buf,
        ack:      resp.ack,
        state:    batchActive,
    }, nil
```

getRequest的结构如下：

```
type getRequest struct {
    sz   int              // request sz events from the broker
    resp chan getResponse // channel to send response to
}
```

getResponse的结构：

```
type getResponse struct {
    ack *ackChan
    buf []publisher.Event
}
```

getResponse里包含了日志的数据，而getRequest包含了一个发送至消费者的channel。 在上文bufferingEventLoop缓冲队列的handleConsumer方法里接收到的参数为getRequest，里面包含了consumer请求的getResponse channel。 如果handleConsumer不发送数据，consumer.Get方法会一直阻塞在select中，直到flushBuffer，consumer的getResponse channel才会接收到日志数据。

我们来看看bufferingEventLoop的调度中心

```
func (l *bufferingEventLoop) run() {
    var (
        broker = l.broker
    )

    for {
        select {
        case <-broker.done:
            return

        case req := <-l.events: // producer pushing new event
            l.handleInsert(&req)

        case req := <-l.pubCancel: // producer cancelling active events
            l.handleCancel(&req)

        case req := <-l.get: // consumer asking for next batch
            l.handleConsumer(&req)

        case l.schedACKS <- l.pendingACKs:
            l.schedACKS = nil
            l.pendingACKs = chanList{}

        case count := <-l.acks:
            l.handleACK(count)

        case <-l.idleC:
            l.idleC = nil
            l.timer.Stop()
            if l.buf.length() > 0 {
                l.flushBuffer()
            }
        }
    }
}
```

当c.broker.requests <- getRequest{sz: sz, resp: c.resp}时候，l.get会得到信息，为什么l.get会得到信息，因为l.get = l.broker.requests

```
func (l *bufferingEventLoop) flushBuffer() {
    l.buf.flushed = true

    if l.buf.length() == 0 {
        panic("flushing empty buffer")
    }

    l.flushList.add(l.buf)
    l.get = l.broker.requests
}
```

再来看看handleConsumer如何处理信息的

```
func (l *bufferingEventLoop) handleConsumer(req *getRequest) {
    buf := l.flushList.head
    if buf == nil {
        panic("get from non-flushed buffers")
    }

    count := buf.length()
    if count == 0 {
        panic("empty buffer in flush list")
    }

    if sz := req.sz; sz > 0 {
        if sz < count {
            count = sz
        }
    }

    if count == 0 {
        panic("empty batch returned")
    }

    events := buf.events[:count]
    clients := buf.clients[:count]
    ackChan := newACKChan(l.ackSeq, 0, count, clients)
    l.ackSeq++

    req.resp <- getResponse{ackChan, events}
    l.pendingACKs.append(ackChan)
    l.schedACKS = l.broker.scheduledACKs

    buf.events = buf.events[count:]
    buf.clients = buf.clients[count:]
    if buf.length() == 0 {
        l.advanceFlushList()
    }
}
```

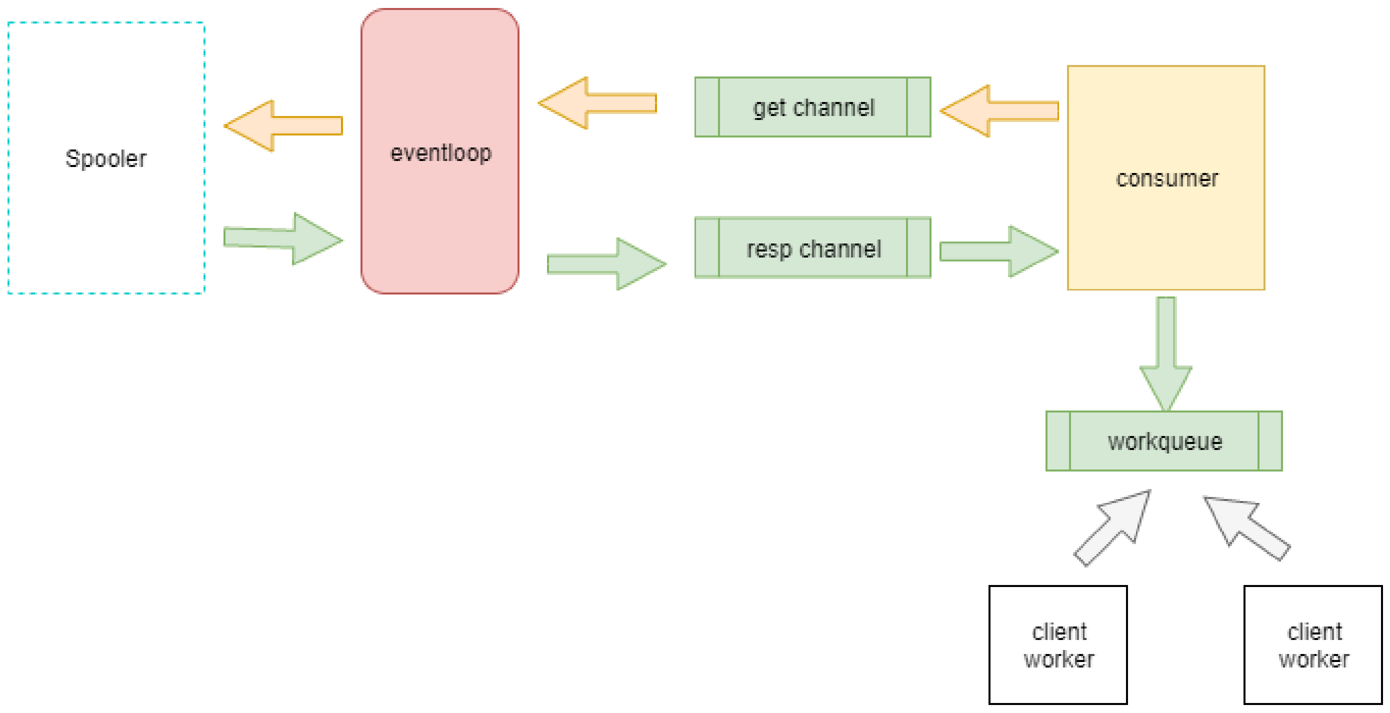
处理req，并且将数据发送给req的resp，在发送的时候c.broker.requests <- getRequest{sz: sz, resp: c.resp}:就将c.resp赋值给了req的resp。所以可以获得返回值getResponse，组装成batch发送出去，其实就是放到type workQueue chan *Batch这个barch类型的channel中。

看一下getResponse

```
type getResponse struct {
    ack *ackChan
    buf []publisher.Event
}
```

可见ack就是channel，buf就是发送的日志。

整个消费的过程非常复杂，数据会在多个 channel 之间传递流转，如下图所示：



首先再介绍两个角色:

```
consumer:  pipeline 在创建的时候，会同时创建一个 consumer。consumer 负责从缓存中取数据
client worker: 负责接收 consumer 传来的数据，并调用 Output 的 Publish 函数进行上报。
```

与 producer 类似，consumer 也不直接操作缓存，而是会向 get channel 中写入消费请求。

consumer 本身是个后台 loop 的过程，这个消费请求会不断进行。

eventloop 监听 get channel, 拿到之后会从缓存中取数据。并将数据写入到 resp channel 中。

consumer 从 resp channel 中拿到 event 数据后，又会将其写入到 workQueue。

workQueue 也是个 channel。client worker 会监听该 channel 上的数据到来，将数据交给 Output client 进行 Publish 上报。

```
type workQueue chan *Batch
```

而且，Output 收到的是 Batch Events，即会一次收到一批 Events。BatchSize 由各个 Output 自行决定。

至此，消息已经递交给了 Output 组件。

output

Filebeat 并不依赖于 Elasticsearch，可以单独存在。我们可以单独使用 Filebeat 进行日志的上报和搜集。filebeat 内置了常用的 Output 组件, 例如 kafka、Elasticsearch、redis 等。出于调试考虑，也可以输出到 console 和 file。我们可以利用现有的 Output 组件，将日志进行上报。

当然，我们也可以自定义 Output 组件，让 Filebeat 将日志转发到我们想要的地方。

在上文提到过，在pipeline初始化的时候，就会设置output的clinet，会创建一个clientWorker或者netClientWorker（可重连，默认就是这个），clientWorker的run方法中，会不停的从consumer发送的channel（就是上面的workQueue）里读取日志数据，然后调用client.Publish批量发送日志。

```
func (w *netClientWorker) run() {
    for !w.closed.Load() {
        reconnectAttempts := 0

        // start initial connect loop from first batch, but return
        // batch to pipeline for other outputs to catch up while we're trying to connect
        for batch := range w.qu {
            batch.Cancelled()

            if w.closed.Load() {
                logp.Info("Closed connection to %v", w.client)
                return
            }

            if reconnectAttempts > 0 {
                logp.Info("Attempting to reconnect to %v with %d reconnect attempt(s)", w.clie
nt, reconnectAttempts)
            } else {
                logp.Info("Connecting to %v", w.client)
            }

            err := w.client.Connect()
            if err != nil {
                logp.Err("Failed to connect to %v: %v", w.client, err)
                reconnectAttempts++
                continue
            }

            logp.Info("Connection to %v established", w.client)
            reconnectAttempts = 0
            break
        }

        // send loop
        for batch := range w.qu {
            if w.closed.Load() {
                if batch != nil {
                    batch.Cancelled()
                }
                return
            }

            err := w.client.Publish(batch)
            if err != nil {
                logp.Err("Failed to publish events: %v", err)
                // on error return to connect loop
                break
            }
        }
    }
}
```

libbeats库中包含了kafka、elasticsearch、logstash等几种client，它们均实现了client接口：

```
type Client interface {
    Close() error
    Publish(publisher.Batch) error
    String() string
}
```

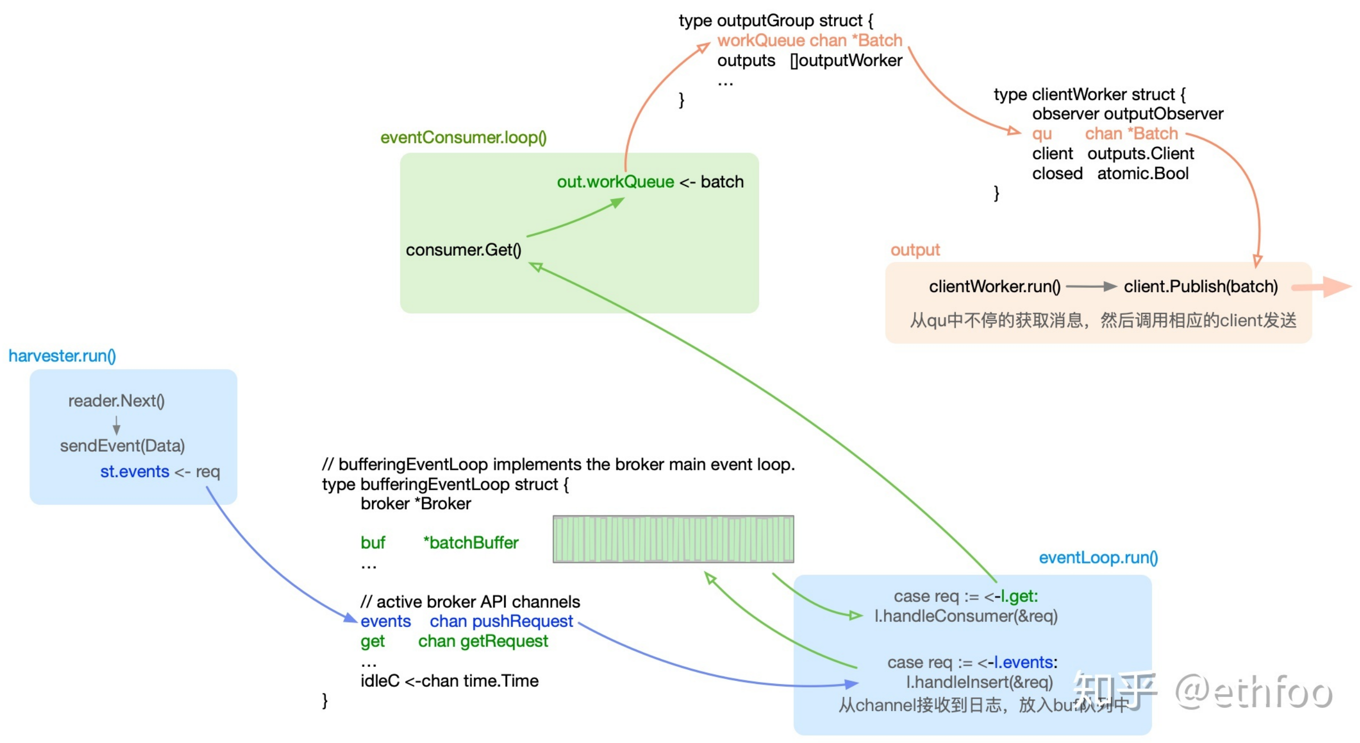
当然最重要的是实现Publish接口，然后将日志发送出去。比如我们看一下kafka的Publish接口

```
func (c *client) Publish(batch publisher.Batch) error {
    events := batch.Events()
    c.observer.NewBatch(len(events))

    ref := &msgRef{
        client: c,
        count:  int32(len(events)),
        total:  len(events),
        failed: nil,
        batch:  batch,
    }

    ch := c.producer.Input()
    for i := range events {
        d := &events[i]
        msg, err := c.getEventMessage(d)
        if err != nil {
            logp.Err("Dropping event: %v", err)
            ref.done()
            c.observer.Dropped(1)
            continue
        }

        msg.ref = ref
        msg.initProducerMessage()
        ch <- &msg.msg
    }

    return nil
}
```

就是基本的kafka客户端的使用方法，到此为止，数据也就发送的kakfa了。

总结

其实在pipleline调度的时候就说明了queue的生产消费的关系，数据在各个channel中进行传输，整个日志数据流转的过程还是表复杂的，在各个channel中进行流转，如下图



```
type outputGroup struct {
    workQueue chan *Batch
    outputs   []outputWorker
    ...
}
```

```
type clientWorker struct {
    observer outputObserver
    qu      chan *Batch
    client  outputs.Client
    closed  atomic.Bool
}
```

```
// bufferingEventLoop implements the broker main event loop.
type bufferingEventLoop struct {
    broker *Broker

    buf     *batchBuffer
    ...

    // active broker API channels
    events  chan pushRequest
    get     chan getRequest
    ...
    idleC <-chan time.Time
}
```

registry和Ack 机制

Filebeat 的可靠性很强，可以保证日志 At least once 的上报，同时也考虑了日志搜集中的各类问题，例如日志断点续读、文件名更改、日志 Truncated 等。

filebeat 之所以可以保证日志可以 at least once 的上报，就是基于其 Ack 机制。

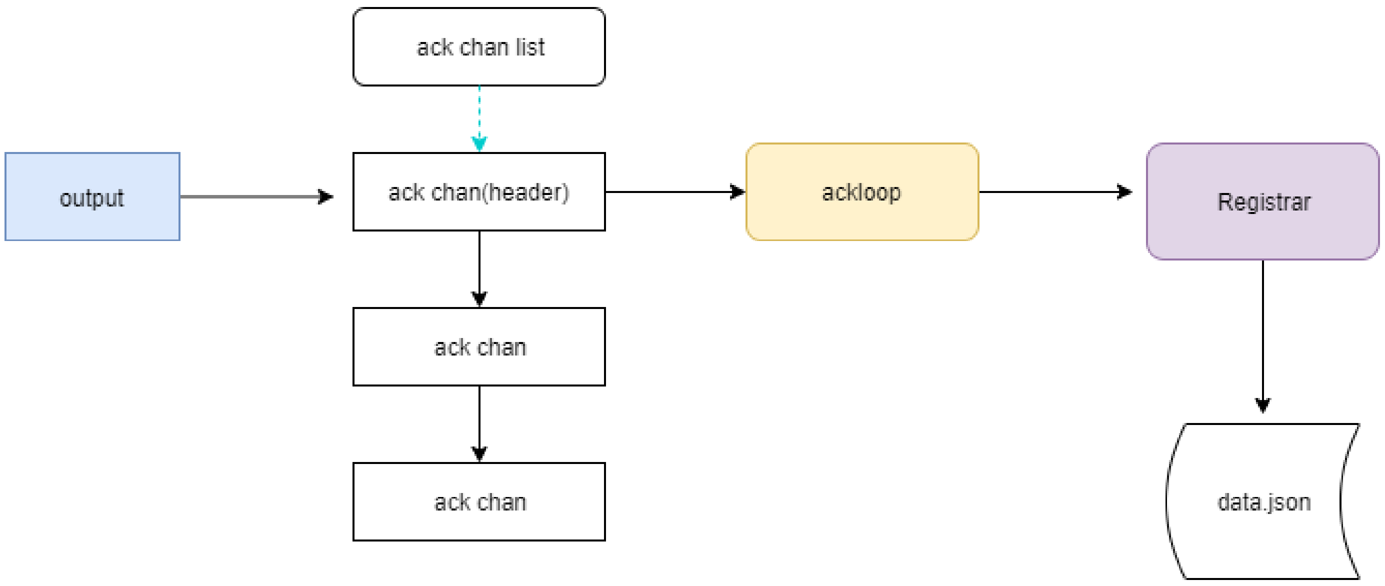
简单来说，Ack 机制就是，当 Output Publish 成功之后会调用 ACK，最终 Registrar 会收到 ACK，并修改偏移量。

而且, Registrar 只会在 Output 调用 batch 的相关信号时，才改变文件偏移量。其中 Batch 对外提供了这些信号:

```
type Batch interface {
    Events() []Event

    // signals
    ACK()
    Drop()
    Retry()
    RetryEvents(events []Event)
    Cancelled()
    CancelledEvents(events []Event)
}
```

Output 在 Publish 之后，无论失败，必须调用这些函数中的其中一个。

以下是 Output Publish 成功后调用 Ack 的流程:



可以看到其中起核心作用的组件是 Ackloop。AckLoop 中有一个 ackChanList，其中每一个 ackChan，对应于转发给 Output 的一个 Batch。 每次新建一个 Batch，同时会建立一个 ackChan，该 ackChan 会被 append 到 ackChanList 中。

而 AckLoop 每次只监听处于 ackChanList 最头部的 ackChan。

当 Batch 被 Output 调用 Ack 后，AckLoop 会收到对应 ackChan 上的事件，并将其最终转发给 Registrar。同时，ackChanList 将会 pop 头部的 ackChan，继续监听接下来的 Ack 事件。

由于 FileBeat 是 At least once 的上报，但并不保证 Exactly once, 因此一条数据可能会被上报多次，所以接收端需要自行进行去重过滤。

上面状态的修改，主要是filebeat维护了一个registry文件在本地的磁盘，该registry文件维护了所有已经采集的日志文件的状态。 实际上，每当日志数据发送至后端成功后，会返回ack事件。filebeat启动了一个独立的registry协程负责监听该事件，接收到ack事件后会将日志文件的State状态更新至registry文件中，State中的Offset表示读取到的文件偏移量，所以filebeat会保证Offset记录之前的日志数据肯定被后端的日志存储接收到。

> pipeline初始化的ack

首先是pipeline初始化只有一次，在这个时候只是简单的初始化了pipeline的ack相关信息，这边也创建的一个queue，原始是使用一个queue，这边初始化broker的时候会创建ack.run()来监听，后来改造多kafka发送后这一条queue 是不用的，而且每次连接kafka的时候创建一个新queue的时候，会都会创建一个ack.run()来监听，流程是一样的，改造可以看支持多kafka的发送 (/post/monitor/log/collect/filebeat/filebeat-principle/#支持多kafka的发送)。

```
p.ackBuilder = &pipelineEmptyACK{p}
p.ackActive = atomic.MakeBool(true)
```

然后就是在创建queue的时候，默认是使用mem的queue，会创建ack。

```
ack := newACKLoop(b, eventLoop.processACK)

b.wg.Add(2)
go func() {
    defer b.wg.Done()
    eventLoop.run()
}()
go func() {
    defer b.wg.Done()
    ack.run()
}()
```

在创建newBufferingEventLoop队列的同时，会newACKLoop并且调用相应结构体的run函数

```
func newACKLoop(b *Broker, processACK func(chanList, int)) *ackLoop {
    l := &ackLoop{broker: b}
    l.processACK = processACK
    return l
}
```

看一下

```
type ackLoop struct {
    broker *Broker
    sig    chan batchAckMsg
    lst    chanList

    totalACK   uint64
    totalSched uint64

    batchesSched uint64
    batchesACKed uint64

    processACK func(chanList, int)
}
```

再看一下run函数

```
func (l *ackLoop) run() {
    var (
        // log = l.broker.logger

        // Buffer up acked event counter in acked. If acked > 0, acks will be set to
        // the broker.acks channel for sending the ACKs while potentially receiving
        // new batches from the broker event loop.
        // This concurrent bidirectionally communication pattern requiring 'select'
        // ensures we can not have any deadlock between the event loop and the ack
        // loop, as the ack loop will not block on any channel
        acked int
        acks  chan int
    )

    for {
        select {
        case <-l.broker.done:
            // TODO: handle pending ACKs?
            // TODO: panic on pending batches?
            return

        case acks <- acked:
            acks, acked = nil, 0

        case lst := <-l.broker.scheduledACKs:
            count, events := lst.count()
            l.lst.concat(&lst)

            // log.Debug("ACK List:")
            // for current := l.lst.head; current != nil; current = current.next {
            //  log.Debugf("  ack entry(seq=%v, start=%v, count=%v",
            //      current.seq, current.start, current.count)
            // }

            l.batchesSched += uint64(count)
            l.totalSched += uint64(events)

        case <-l.sig:
            acked += l.handleBatchSig()
            if acked > 0 {
                acks = l.broker.acks
            }
        }

        // log.Debug("ackloop INFO")
        // log.Debug("ackloop:   total events scheduled = ", l.totalSched)
        // log.Debug("ackloop:   total events ack = ", l.totalACK)
        // log.Debug("ackloop:   total batches scheduled = ", l.batchesSched)
        // log.Debug("ackloop:   total batches ack = ", l.batchesACKed)

        l.sig = l.lst.channel()
        // if l.sig == nil {
        //  log.Debug("ackloop: no ack scheduled")
        // } else {
        //  log.Debug("ackloop: schedule ack: ", l.lst.head.seq)
        // }
    }
}
```

这边其实就是对ack信号的调度处理中心。

> registrar初始化

然后就是启动filebeat的时候可能是要初始化registrar

```
// Setup registrar to persist state
registrar, err := registrar.New(config.Registry, finishedLogger)
if err != nil {
    logp.Err("Could not init registrar: %v", err)
    return err
}
```

config.Registry就是registry文件的配置信息

```
type Registry struct {
    Path         string        `config:"path"`
    Permissions  os.FileMode   `config:"file_permissions"`
    FlushTimeout time.Duration `config:"flush"`
    MigrateFile  string        `config:"migrate_file"`
}
```

构建方法很简单，就是对文件的一些描述赋值给了这个结构体

```
// New creates a new Registrar instance, updating the registry file on
// `file.State` updates. New fails if the file can not be opened or created.
func New(cfg config.Registry, out successLogger) (*Registrar, error) {
    home := paths.Resolve(paths.Data, cfg.Path)
    migrateFile := cfg.MigrateFile
    if migrateFile != "" {
        migrateFile = paths.Resolve(paths.Data, migrateFile)
    }

    err := ensureCurrent(home, migrateFile, cfg.Permissions)
    if err != nil {
        return nil, err
    }

    dataFile := filepath.Join(home, "filebeat", "data.json")
    r := &Registrar{
        registryFile: dataFile,
        fileMode:     cfg.Permissions,
        done:         make(chan struct{}),
        states:       file.NewStates(),
        Channel:      make(chan []file.State, 1),
        flushTimeout: cfg.FlushTimeout,
        out:          out,
        wg:           sync.WaitGroup{},
    }
    return r, r.Init()
}
```

然后返回Registrar结构体

```
type Registrar struct {
    Channel      chan []file.State
    out          successLogger
    done         chan struct{}
    registryFile string      // Path to the Registry File
    fileMode     os.FileMode // Permissions to apply on the Registry File
    wg           sync.WaitGroup

    states               *file.States // Map with all file paths inside and the corresponding
 state
    gcRequired           bool         // gcRequired is set if registry state needs to be gc'ed
before the next write
    gcEnabled            bool         // gcEnabled indicates the registry contains some state
 that can be gc'ed in the future
    flushTimeout         time.Duration
    bufferedStateUpdates int
}
```

还进行了初始化，主要是对文件进行了一些检查。然后就是启动：

```
// Start the registrar
err = registrar.Start()
if err != nil {
    return fmt.Errorf("Could not start registrar: %v", err)
}

func (r *Registrar) Start() error {
    // Load the previous log file locations now, for use in input
    err := r.loadStates()
    if err != nil {
        return fmt.Errorf("Error loading state: %v", err)
    }

    r.wg.Add(1)
    go r.Run()

    return nil
}
```

首先就是加载了目前存在的文件的状态来赋值给结构体

```
// loadStates fetches the previous reading state from the configure RegistryFile file
// The default file is `registry` in the data path.
func (r *Registrar) loadStates() error {
    f, err := os.Open(r.registryFile)
    if err != nil {
        return err
    }

    defer f.Close()

    logp.Info("Loading registrar data from %s", r.registryFile)

    states, err := readStatesFrom(f)
    if err != nil {
        return err
    }
    r.states.SetStates(states)
    logp.Info("States Loaded from registrar: %+v", len(states))

    return nil
}
```

然后开始监听来更新文件

```
func (r *Registrar) Run() {
    logp.Debug("registrar", "Starting Registrar")
    // Writes registry on shutdown
    defer func() {
        r.writeRegistry()
        r.wg.Done()
    }()

    var (
        timer  *time.Timer
        flushC <-chan time.Time
    )

    for {
        select {
        case <-r.done:
            logp.Info("Ending Registrar")
            return
        case <-flushC:
            flushC = nil
            timer.Stop()
            r.flushRegistry()
        case states := <-r.Channel:
            r.onEvents(states)
            if r.flushTimeout <= 0 {
                r.flushRegistry()
            } else if flushC == nil {
                timer = time.NewTimer(r.flushTimeout)
                flushC = timer.C
            }
        }
    }
}
```

当接受到Registrar中channel的发来的文件状态，就更新结构体的值，如果到时间了就将内存中的值刷新到本地文件中，如果没有就定一个timeout时间后刷新到本地文件中。

我们可以简单的看一下这个channel

```
Channel:      make(chan []file.State, 1),
```

就是一个文件状态的channel，关于文件状态的结构体如下

```
// State is used to communicate the reading state of a file
type State struct {
    Id          string            `json:"-"` // local unique id to make comparison more effici
ent
    Finished    bool              `json:"-"` // harvester state
    Fileinfo    os.FileInfo       `json:"-"` // the file info
    Source      string            `json:"source"`
    Offset      int64             `json:"offset"`
    Timestamp   time.Time         `json:"timestamp"`
    TTL         time.Duration     `json:"ttl"`
    Type        string            `json:"type"`
    Meta        map[string]string `json:"meta"`
    FileStateOS file.StateOS
}
```

记录在registry文件中的数据大致如下所示：

```
[{"source":"/tmp/aa.log","offset":48,"timestamp":"2019-07-03T13:54:01.298995+08:00","ttl":-1,
"type":"log","meta":null,"FileStateOS":{"inode":7048952,"device":16777220}}]
```

由于文件可能会被改名或移动，filebeat会根据inode和设备号来标志每个日志文件。

到这边registrar启动也结束了，下面就是监控registrar中channel的数据，在启动的时候还做了一件事情，那就是把channel设置到pipeline中去。

在构建registrar的时候，通过registrar中channel构建一个结构体registrarLogger

```
type registrarLogger struct {
    done chan struct{}
    ch   chan<- []file.State
}
```

这个就是用来交互的结构体,这个结构体中的channel获取的文件状态就是给上面的监听程序进行处理。

```
// Make sure all events that were published in
registrarChannel := newRegistrarLogger(registrar)

func newRegistrarLogger(reg *registrar.Registrar) *registrarLogger {
    return &registrarLogger{
        done: make(chan struct{}),
        ch:   reg.Channel,
    }
}
```

通过这个registrarLogger结构体，做了如下的调用

```
err = b.Publisher.SetACKHandler(beat.PipelineACKHandler{
    ACKEvents: newEventACKer(finishedLogger, registrarChannel).ackEvents,
})
if err != nil {
    logp.Err("Failed to install the registry with the publisher pipeline: %v", err)
    return err
}
```

首先看一下newEventACKer(finishedLogger, registrarChannel)这个结构体的ackEvents函数

```
func newEventACKer(stateless statelessLogger, stateful statefulLogger) *eventACKer {
    return &eventACKer{stateless: stateless, stateful: stateful, log: logp.NewLogger("acker")}
}

func (a *eventACKer) ackEvents(data []interface{}) {
    stateless := 0
    states := make([]file.State, 0, len(data))
    for _, datum := range data {
        if datum == nil {
            stateless++
            continue
        }

        st, ok := datum.(file.State)
        if !ok {
            stateless++
            continue
        }

        states = append(states, st)
    }

    if len(states) > 0 {
        a.log.Debugw("stateful ack", "count", len(states))
        a.stateful.Published(states)
    }

    if stateless > 0 {
        a.log.Debugw("stateless ack", "count", stateless)
        a.stateless.Published(stateless)
    }
}
```

可见是一个eventACKer结构体的函数赋值给了beat.PipelineACKHandler的成员函数，我们再来看一下beat.PipelineACKHandler

```
// PipelineACKHandler configures some pipeline-wide event ACK handler.
type PipelineACKHandler struct {
    // ACKCount reports the number of published events recently acknowledged
    // by the pipeline.
    ACKCount func(int)

    // ACKEvents reports the events recently acknowledged by the pipeline.
    // Only the events 'Private' field will be reported.
    ACKEvents func([]interface{})

    // ACKLastEvent reports the last ACKed event per pipeline client.
    // Only the events 'Private' field will be reported.
    ACKLastEvents func([]interface{})
}
```

就是一个这样的结构体作为参数，最后我们来看一下SetACKHandler这个函数的调用，首先b的就是libbeat中创建的beat，其中的Publisher就是对应的初始化的Pipeline，看一下Pipeline的SetACKHandler方法

```go
// SetACKHandler sets a global ACK handler on all events published to the pipeline.
// SetACKHandler must be called before any connection is made.
func (p *Pipeline) SetACKHandler(handler beat.PipelineACKHandler) error {
    p.eventer.mutex.Lock()
    defer p.eventer.mutex.Unlock()

    if !p.eventer.modifyable {
        return errors.New("can not set ack handler on already active pipeline")
    }

    // TODO: check only one type being configured

    cb, err := newPipelineEventCB(handler)
    if err != nil {
        return err
    }

    if cb == nil {
        p.ackBuilder = &pipelineEmptyACK{p}
        p.eventer.cb = nil
        return nil
    }

    p.eventer.cb = cb
    if cb.mode == countACKMode {
        p.ackBuilder = &pipelineCountACK{
            pipeline: p,
            cb:       cb.onCounts,
        }
    } else {
        p.ackBuilder = &pipelineEventsACK{
            pipeline: p,
            cb:       cb.onEvents,
        }
    }

    return nil
}
```

newPipelineEventCB是根据传递的不同函数，创建不同mode的pipelineEventCB结构体，启动goroutine来work。我们这边传递的是ACKEvents，设置mode为eventsACKMode

```go
func newPipelineEventCB(handler beat.PipelineACKHandler) (*pipelineEventCB, error) {
    mode := noACKMode
    if handler.ACKCount != nil {
        mode = countACKMode
    }
    if handler.ACKEvents != nil {
        if mode != noACKMode {
            return nil, errors.New("only one callback can be set")
        }
        mode = eventsACKMode
    }
    if handler.ACKLastEvents != nil {
        if mode != noACKMode {
            return nil, errors.New("only one callback can be set")
        }
        mode = lastEventsACKMode
    }

    // yay, no work
    if mode == noACKMode {
        return nil, nil
    }

    cb := &pipelineEventCB{
        acks:          make(chan int),
        mode:          mode,
        handler:       handler,
        events:        make(chan eventsDataMsg),
        droppedEvents: make(chan eventsDataMsg),
    }
    go cb.worker()
    return cb, nil
}
```

再来看看worker工作协程

```
func (p *pipelineEventCB) worker() {
    defer close(p.acks)
    defer close(p.events)
    defer close(p.droppedEvents)

    for {
        select {
        case count := <-p.acks:
            exit := p.collect(count)
            if exit {
                return
            }

            // short circuit dropped events, but have client block until all events
            // have been processed by pipeline ack handler
        case msg := <-p.droppedEvents:
            p.reportEventsData(msg.data, msg.total)
            if msg.sig != nil {
                close(msg.sig)
            }

        case <-p.done:
            return
        }
    }
}
```

同时对于不同的mode对p.ackBuilder进行了重新构建，因为是代码设置mode为eventsACKMode

```
p.ackBuilder = &pipelineEventsACK{
    pipeline: p,
    cb:       cb.onEvents,
}

type pipelineEventsACK struct {
    pipeline *Pipeline
    cb       func([]interface{}, int)
}
```

到这里启动就结束了。

> input初始化的ack

crawler启动后构建新的input构建的时候，需要获取到pipeline的client，在使用ConnectWith进行构建的时候，会构建client的acker，第一次参数是processors != nil，影响后的结构体的创建，一般是true

```
acker := p.makeACKer(processors != nil, &cfg, waitClose, client.unlink)
```

我们来看一下makeACKer这个函数

```
func (p *Pipeline) makeACKer(
    canDrop bool,
    cfg *beat.ClientConfig,
    waitClose time.Duration,
    afterClose func(),
) acker {
    var (
        bld   = p.ackBuilder
        acker acker
    )

    sema := p.eventSema
    switch {
    case cfg.ACKCount != nil:
        acker = bld.createCountACKer(canDrop, sema, cfg.ACKCount)
    case cfg.ACKEvents != nil:
        acker = bld.createEventACKer(canDrop, sema, cfg.ACKEvents)
    case cfg.ACKLastEvent != nil:
        cb := lastEventACK(cfg.ACKLastEvent)
        acker = bld.createEventACKer(canDrop, sema, cb)
    default:
        if waitClose <= 0 {
            acker = bld.createPipelineACKer(canDrop, sema)
        } else {
            acker = bld.createCountACKer(canDrop, sema, func(_ int) {})
        }
    }

    if waitClose <= 0 {
        return newCloseACKer(acker, afterClose)
    }
    return newWaitACK(acker, waitClose, afterClose)
}
```

需要使用p.ackBuilder的create函数，我们在上面SetACKHandler的时候构建了p.ackBuilder，根据cfg配置调用，默认调用

```
acker = bld.createEventACKer(canDrop, sema, cfg.ACKEvents)
```

也就是调用

```
func (b *pipelineEventsACK) createEventACKer(canDrop bool, sema *sema, fn func([]interface{}))
acker {
    return buildClientEventACK(b.pipeline, canDrop, sema, func(guard *clientACKer) func([]inte
rface{}, int) {
        return func(data []interface{}, acked int) {
            b.cb(data, acked)
            if guard.Active() {
                fn(data)
            }
        }
    })
}
```

调用函数buildClientEventACK

```
func buildClientEventACK(
    pipeline *Pipeline,
    canDrop bool,
    sema *sema,
    mk func(*clientACKer) func([]interface{}, int),
) acker {
    guard := &clientACKer{}
    guard.lift(newEventACK(pipeline, canDrop, sema, mk(guard)))
    return guard
}
```

看一下返回值clientACKer结构体，其成员acker的赋值就是eventDataACK。

```
func newEventACK(
    pipeline *Pipeline,
    canDrop bool,
    sema *sema,
    fn func([]interface{}, int),
) *eventDataACK {
    a := &eventDataACK{pipeline: pipeline, fn: fn}
    a.acker = makeCountACK(pipeline, canDrop, sema, a.onACK)

    return a
}
```

创建一个eventDataACK的结构体，fn就是mk(guard)就是

```
func(data []interface{}, acked int) {
    b.cb(data, acked)
    if guard.Active() {
        fn(data)
    }
```

然后调用makeCountACK来赋值给eventDataACK的acker

```
func makeCountACK(pipeline *Pipeline, canDrop bool, sema *sema, fn func(int, int)) acker {
    if canDrop {
        return newBoundGapCountACK(pipeline, sema, fn)
    }
    return newCountACK(pipeline, fn)
}
```

canDrop之前说过了，就是ture，所以创建newBoundGapCountACK，将eventDataACK的onACK当参数传递进来。

```
func (a *eventDataACK) onACK(total, acked int) {
    n := total

    a.mutex.Lock()
    data := a.data[:n]
    a.data = a.data[n:]
    a.mutex.Unlock()

    if len(data) > 0 && a.pipeline.ackActive.Load() {
        a.fn(data, acked)
    }
}
```

继续newBoundGapCountACK

```
func newBoundGapCountACK(
    pipeline *Pipeline,
    sema *sema,
    fn func(total, acked int),
) *boundGapCountACK {
    a := &boundGapCountACK{active: true, sema: sema, fn: fn}
    a.acker.init(pipeline, a.onACK)
    return a
}
```

创建了一个boundGapCountACK，调用初始化函数，将这个结构体的onACK传进去就是fn

```
func (a *gapCountACK) init(pipeline *Pipeline, fn func(int, int)) {
    *a = gapCountACK{
        pipeline: pipeline,
        fn:       fn,
        done:     make(chan struct{}),
        drop:     make(chan struct{}),
        acks:     make(chan int, 1),
    }

    init := &gapInfo{}
    a.lst.head = init
    a.lst.tail = init

    go a.ackLoop()
}
```

然后就是

```
func (a *gapCountACK) ackLoop() {
    // close channels, as no more events should be ACKed:
    // - once pipeline is closed
    // - all events of the closed client have been acked/processed by the pipeline

    acks, drop := a.acks, a.drop
    closing := false

    for {
        select {
        case <-a.done:
            closing = true
            a.done = nil
            if a.events.Load() == 0 {
                // stop worker, if all events accounted for have been ACKed.
                // If new events are added after this acker won't handle them, which may
                // result in duplicates
                return
            }

        case <-a.pipeline.ackDone:
            return

        case n := <-acks:
            empty := a.handleACK(n)
            if empty && closing && a.events.Load() == 0 {
                // stop worker, if and only if all events accounted for have been ACKed
                return
            }

        case <-drop:
            // TODO: accumulate multiple drop events + flush count with timer
            a.events.Sub(1)
            a.fn(1, 0)
        }
    }
}
```

这边启动了一个acker的监听，然后使用这个clientACKer结构体又构建了一个新的结构体

```
func newWaitACK(acker acker, timeout time.Duration, afterClose func()) *waitACK {
    return &waitACK{
        acker:      acker,
        signalAll:  make(chan struct{}, 1),
        signalDone: make(chan struct{}),
        waitClose:  timeout,
        active:     atomic.MakeBool(true),
        afterClose: afterClose,
    }
}
```

到这里连接创建就结束了，创建的acker就是waitACK

```
client.acker = acker
```

所以pipeline的client的acker就是waitACK。

> publish数据的ack

下面是有数据的时候将数据发送到pipeline，调用的client的publish函数，在发送数据的时候调用了addEvent。

```
open := c.acker.addEvent(e, publish)
```

c.acker也就是上面waitACK的addEvent函数，e就是对应发送的事件

```
func (a *waitACK) addEvent(event beat.Event, published bool) bool {
    if published {
        a.events.Inc()
    }
    return a.acker.addEvent(event, published)
}
```

调用的是结构体成员acker的addEvent，也就是eventDataACK的addEvent。

```
func (a *eventDataACK) addEvent(event beat.Event, published bool) bool {
    a.mutex.Lock()
    active := a.pipeline.ackActive.Load()
    if active {
        a.data = append(a.data, event.Private)
    }
    a.mutex.Unlock()

    if active {
        return a.acker.addEvent(event, published)
    }
    return false
}
```

到这边就是将数据传输到了data中，同时调用了其对应的acker的addEvent函数。

```
func newEventACK(
    pipeline *Pipeline,
    canDrop bool,
    sema *sema,
    fn func([]interface{}, int),
) *eventDataACK {
    a := &eventDataACK{pipeline: pipeline, fn: fn}
    a.acker = makeCountACK(pipeline, canDrop, sema, a.onACK)

    return a
}
```

我们看到a.acker = makeCountACK(pipeline, canDrop, sema, a.onACK)，再看

```
func makeCountACK(pipeline *Pipeline, canDrop bool, sema *sema, fn func(int, int)) acker {
    if canDrop {
        return newBoundGapCountACK(pipeline, sema, fn)
    }
    return newCountACK(pipeline, fn)
}
```

canDrop上面说明过了，所以是newBoundGapCountACK

```
func newBoundGapCountACK(
    pipeline *Pipeline,
    sema *sema,
    fn func(total, acked int),
) *boundGapCountACK {
    a := &boundGapCountACK{active: true, sema: sema, fn: fn}
    a.acker.init(pipeline, a.onACK)
    return a
}
```

所以返回的结构体是boundGapCountACK，调用的也是这个结构体的addEvent

```
func (a *boundGapCountACK) addEvent(event beat.Event, published bool) bool {
    a.sema.inc()
    return a.acker.addEvent(event, published)
}
```

还有acker.addEvent，再看boundGapCountACK的acker

```
type boundGapCountACK struct {
    active bool
    fn     func(total, acked int)

    acker gapCountACK
    sema  *sema
}
```

是一个gapCountACK的结构体，调用初始化a.acker.init(pipeline, a.onACK)来赋值

```
func (a *gapCountACK) init(pipeline *Pipeline, fn func(int, int)) {
    *a = gapCountACK{
        pipeline: pipeline,
        fn:       fn,
        done:     make(chan struct{}),
        drop:     make(chan struct{}),
        acks:     make(chan int, 1),
    }

    init := &gapInfo{}
    a.lst.head = init
    a.lst.tail = init

    go a.ackLoop()
}
```

同时启动了一个监听程序

```
func (a *gapCountACK) ackLoop() {
    // close channels, as no more events should be ACKed:
    // - once pipeline is closed
    // - all events of the closed client have been acked/processed by the pipeline

    acks, drop := a.acks, a.drop
    closing := false

    for {
        select {
        case <-a.done:
            closing = true
            a.done = nil
            if a.events.Load() == 0 {
                // stop worker, if all events accounted for have been ACKed.
                // If new events are added after this acker won't handle them, which may
                // result in duplicates
                return
            }

        case <-a.pipeline.ackDone:
            return

        case n := <-acks:
            empty := a.handleACK(n)
            if empty && closing && a.events.Load() == 0 {
                // stop worker, if and only if all events accounted for have been ACKed
                return
            }

        case <-drop:
            // TODO: accumulate multiple drop events + flush count with timer
            a.events.Sub(1)
            a.fn(1, 0)
        }
    }
}
```

当drop获取到信号的时候，就会调用fn也就是boundGapCountACK的onACK函数

```
func (a *boundGapCountACK) onACK(total, acked int) {
    a.sema.release(total)
    a.fn(total, acked)
}
```

这边会调用到fn也就是eventDataACK的onACK

```
func (a *eventDataACK) onACK(total, acked int) {
    n := total

    a.mutex.Lock()
    data := a.data[:n]
    a.data = a.data[n:]
    a.mutex.Unlock()

    if len(data) > 0 && a.pipeline.ackActive.Load() {
        a.fn(data, acked)
    }
}
```

这边会调用fn也就是我们创建的ackBuilder的cb成员也就是我们的ackBuilder结构的cb函数，也就是我们的onEvents

```
func (p *pipelineEventCB) onEvents(data []interface{}, acked int) {
    p.pushMsg(eventsDataMsg{data: data, total: len(data), acked: acked})
}

func (p *pipelineEventCB) onCounts(total, acked int) {
    p.pushMsg(eventsDataMsg{total: total, acked: acked})
}

func (p *pipelineEventCB) pushMsg(msg eventsDataMsg) {
    if msg.acked == 0 {
        p.droppedEvents <- msg
    } else {
        msg.sig = make(chan struct{})
        p.events <- msg
        <-msg.sig
    }
}
```

获取到了file的具体数据。到这边就是继续监听，我们先看addEvent，published肯定是true，正常都是有事件的publish = event != nil

```
func (a *gapCountACK) addEvent(_ beat.Event, published bool) bool {
    // if gapList is empty and event is being dropped, forward drop event to ack
    // loop worker:

    a.events.Inc()
    if !published {
        a.addDropEvent()
    } else {
        a.addPublishedEvent()
    }

    return true
}
```

所以看addPublishedEvent，只是给结构体成员send加一

```
func (a *gapCountACK) addPublishedEvent() {
    // event is publisher -> add a new gap list entry if gap is present in current
    // gapInfo

    a.lst.Lock()

    current := a.lst.tail
    current.Lock()

    if current.dropped > 0 {
        tmp := &gapInfo{}
        a.lst.tail.next = tmp
        a.lst.tail = tmp

        current.Unlock()
        tmp.Lock()
        current = tmp
    }

    a.lst.Unlock()

    current.send++
    current.Unlock()
}
```

到这边调用的addevent就结束了，下面就是等待output的publish后的返回调用。上面已经有四个相关ack的监听，一个queue消费的监听，一个registry监听，一个是pipeline的监听，一个gapCountACK的ackLoop。

> publish后回调ack

然后就是output的publish的时候进行回调了，我们使用的是kafka，kafka在connect的时候会新建两个协程，来监听发送的情况，如下

```
func (c *client) Connect() error {
    c.mux.Lock()
    defer c.mux.Unlock()

    debugf("connect: %v", c.hosts)

    // try to connect
    producer, err := sarama.NewAsyncProducer(c.hosts, &c.config)
    if err != nil {
        logp.Err("Kafka connect fails with: %v", err)
        return err
    }

    c.producer = producer

    c.wg.Add(2)
    go c.successWorker(producer.Successes())
    go c.errorWorker(producer.Errors())

    return nil
}
```

我们一共可以看到两个处理方式，一个成功一个失败，producer.Successes()和producer.Errors()为这个producer的成功和错误返回channel。

```
func (c *client) successWorker(ch <-chan *sarama.ProducerMessage) {
    defer c.wg.Done()
    defer debugf("Stop kafka ack worker")

    for libMsg := range ch {
        msg := libMsg.Metadata.(*message)
        msg.ref.done()
    }
}

func (c *client) errorWorker(ch <-chan *sarama.ProducerError) {
    defer c.wg.Done()
    defer debugf("Stop kafka error handler")

    for errMsg := range ch {
        msg := errMsg.Msg.Metadata.(*message)
        msg.ref.fail(msg, errMsg.Err)
    }
}
```

我们看一下成功的响应，失败也是一样的，只不过多了一个错误处理，有兴趣可以自己看一下。

```
func (r *msgRef) done() {
    r.dec()
}

func (r *msgRef) dec() {
    i := atomic.AddInt32(&r.count, -1)
    if i > 0 {
        return
    }

    debugf("finished kafka batch")
    stats := r.client.observer

    err := r.err
    if err != nil {
        failed := len(r.failed)
        success := r.total - failed
        r.batch.RetryEvents(r.failed)

        stats.Failed(failed)
        if success > 0 {
            stats.Acked(success)
        }

        debugf("Kafka publish failed with: %v", err)
    } else {
        r.batch.ACK()
        stats.Acked(r.total)
    }
}
```

在else中也就是接受到成功发送信号后调用了batch.ACK()。我们来看一下batch，首先是msg的类型转化

```
msg := errMsg.Msg.Metadata.(*message)
```

转化为我们定义的kafka的message的结构体message

```
type message struct {
    msg sarama.ProducerMessage

    topic string
    key   []byte
    value []byte
    ref   *msgRef
    ts    time.Time

    hash      uint32
    partition int32

    data publisher.Event
}
```

在kafka的client使用publish的时候初始化了ref，给batch赋值

```
ref := &msgRef{
        client: c,
        count:  int32(len(events)),
        total:  len(events),
        failed: nil,
        batch:  batch,
    }
```

这个batch就是重netClientWorker的qu workQueue中获取的，看一下这个channel是bantch类型

```
type workQueue chan *Batch
```

这个batch就是我们需要找的结构，发送kafka成功后就是调用这个结构他的ACK函数

```
type Batch struct {
    original queue.Batch
    ctx      *batchContext
    ttl      int
    events   []publisher.Event
}
```

我们来看一下Batch的ACK函数

```
func (b *Batch) ACK() {
    b.ctx.observer.outBatchACKed(len(b.events))
    b.original.ACK()
    releaseBatch(b)
}
```

这边继续调用ACK，我们需要看一下b.original赋值，赋值都会调用newBatch函数

```
func newBatch(ctx *batchContext, original queue.Batch, ttl int) *Batch {
    if original == nil {
        panic("empty batch")
    }

    b := batchPool.Get().(*Batch)
    *b = Batch{
        original: original,
        ctx:      ctx,
        ttl:      ttl,
        events:   original.Events(),
    }
    return b
}
```

只在eventConsumer消费的时候调用了newBatch，通过get方法获取的queueBatch给了他

```
queueBatch, err := consumer.Get(c.out.batchSize)
if err != nil {
    out = nil
    consumer = nil
    continue
}
if queueBatch != nil {
    batch = newBatch(c.ctx, queueBatch, c.out.timeToLive)
}
```

首先consumer是基于mem的，看一下get方法

```
func (c *consumer) Get(sz int) (queue.Batch, error) {
    // log := c.broker.logger

    if c.closed.Load() {
        return nil, io.EOF
    }

    select {
    case c.broker.requests <- getRequest{sz: sz, resp: c.resp}:
    case <-c.done:
        return nil, io.EOF
    }

    // if request has been send, we do have to wait for a response
    resp := <-c.resp
    return &batch{
        consumer: c,
        events:   resp.buf,
        ack:      resp.ack,
        state:    batchActive,
    }, nil
}
```

可以看到返回使用的是结构体batch如下，我们简单看一下需要先向队列请求channel发送getRequest结构体，等待resp的返回来创建下面的结构体。具体的处理逻辑可以查看pipeline的消费 (/post/monitor/log/collect/filebeat/filebeat-principle/#pipeline-的消费过程)。

```
type batch struct {
    consumer     *consumer
    events       []publisher.Event
    clientStates []clientState
    ack          *ackChan
    state        ackState
}
```

看一下batch的ACK函数

```
func (b *batch) ACK() {
    if b.state != batchActive {
        switch b.state {
        case batchACK:
            panic("Can not acknowledge already acknowledged batch")
        default:
            panic("inactive batch")
        }
    }

    b.report()
}

func (b *batch) report() {
    b.ack.ch <- batchAckMsg{}
}
```

就是给batch的ack也就是getResponse的ack的ch发送了一个信号batchAckMsg{}。这个ch接收到信号，牵涉到一个完整的消费的调度过程。

我们先看一下正常的消费调度，在上面说过，首先在有数据发送到queue的时候，consumer会获取这个数据

```
for {
        if !paused && c.out != nil && consumer != nil && batch == nil {
            out = c.out.workQueue
            queueBatch, err := consumer.Get(c.out.batchSize)
            ...
            batch = newBatch(c.ctx, queueBatch, c.out.timeToLive)
        }
        ...
        select {
        case <-c.done:
            return
        case sig := <-c.sig:
            handleSignal(sig)
        case out <- batch:
            batch = nil
        }
    }
```

上面consumer.Get就是消费者consumer从Broker中获取日志数据，然后发送至out的channel中被output client发送，我们看一下Get方法里的核心代码：

```
func (c *consumer) Get(sz int) (queue.Batch, error) {
    // log := c.broker.logger

    if c.closed.Load() {
        return nil, io.EOF
    }

    select {
    case c.broker.requests <- getRequest{sz: sz, resp: c.resp}:
    case <-c.done:
        return nil, io.EOF
    }

    // if request has been send, we do have to wait for a response
    resp := <-c.resp
    return &batch{
        consumer: c,
        events:   resp.buf,
        ack:      resp.ack,
        state:    batchActive,
    }, nil
}
```

当c.broker.requests <- getRequest{sz: sz, resp: c.resp}时候,我们需要看一下bufferingEventLoop的调度中心的响应。

```
func (l *bufferingEventLoop) run() {
    var (
        broker = l.broker
    )

    for {
        select {
        case <-broker.done:
            return

        case req := <-l.events: // producer pushing new event
            l.handleInsert(&req)

        case req := <-l.pubCancel: // producer cancelling active events
            l.handleCancel(&req)

        case req := <-l.get: // consumer asking for next batch
            l.handleConsumer(&req)

        case l.schedACKS <- l.pendingACKs:
            l.schedACKS = nil
            l.pendingACKs = chanList{}

        case count := <-l.acks:
            l.handleACK(count)

        case <-l.idleC:
            l.idleC = nil
            l.timer.Stop()
            if l.buf.length() > 0 {
                l.flushBuffer()
            }
        }
    }
}
```

l.get会得到信息，为什么l.get会得到信息，因为l.get = l.broker.requests

```
func (l *bufferingEventLoop) flushBuffer() {
    l.buf.flushed = true

    if l.buf.length() == 0 {
        panic("flushing empty buffer")
    }

    l.flushList.add(l.buf)
    l.get = l.broker.requests
}
```

l.get得到信息后handleConsumer如何处理信息的

```
func (l *bufferingEventLoop) handleConsumer(req *getRequest) {
    buf := l.flushList.head
    if buf == nil {
        panic("get from non-flushed buffers")
    }

    count := buf.length()
    if count == 0 {
        panic("empty buffer in flush list")
    }

    if sz := req.sz; sz > 0 {
        if sz < count {
            count = sz
        }
    }

    if count == 0 {
        panic("empty batch returned")
    }

    events := buf.events[:count]
    clients := buf.clients[:count]
    ackChan := newACKChan(l.ackSeq, 0, count, clients)
    l.ackSeq++

    req.resp <- getResponse{ackChan, events}
    l.pendingACKs.append(ackChan)
    l.schedACKS = l.broker.scheduledACKs

    buf.events = buf.events[count:]
    buf.clients = buf.clients[count:]
    if buf.length() == 0 {
        l.advanceFlushList()
    }
}
```

我们可以看到获取到数据封装getResponse发送给output，我们这边不看这个数据具体发送到workqueue，而是关心的是ack。

先看ack构建的结构体

```
func newACKChan(seq uint, start, count int, states []clientState) *ackChan {
    ch := ackChanPool.Get().(*ackChan)
    ch.next = nil
    ch.seq = seq
    ch.start = start
    ch.count = count
    ch.states = states
    return ch
}
```

然后将这个ackChan新增到chanlist的链表中。

```
l.pendingACKs.append(ackChan)
```

将l.schedACKS = l.broker.scheduledACKs，一开始调度的时候l.schedACKS是阻塞的，l.pendingACKs不能写入到l.schedACKS，但是这边进行赋值后就是将l.pendingACKs写入到l.broker.scheduledACKs，这个在初始化的时候是有缓存的。就直接写入了，然后bufferingEventLoop调度中心将这个数据清空，bufferingEventLoop的ack的调度获取到这个chanenl

```go
func (l *ackLoop) run() {
    var (
        // log = l.broker.logger

        // Buffer up acked event counter in acked. If acked > 0, acks will be set to
        // the broker.acks channel for sending the ACKs while potentially receiving
        // new batches from the broker event loop.
        // This concurrent bidirectionally communication pattern requiring 'select'
        // ensures we can not have any deadlock between the event loop and the ack
        // loop, as the ack loop will not block on any channel
        acked int
        acks  chan int
    )

    for {
        select {
        case <-l.broker.done:
            // TODO: handle pending ACKs?
            // TODO: panic on pending batches?
            return

        case acks <- acked:
            acks, acked = nil, 0

        case lst := <-l.broker.scheduledACKs:
            count, events := lst.count()
            l.lst.concat(&lst)

            // log.Debug("ACK List:")
            // for current := l.lst.head; current != nil; current = current.next {
            //  log.Debugf("  ack entry(seq=%v, start=%v, count=%v",
            //      current.seq, current.start, current.count)
            // }

            l.batchesSched += uint64(count)
            l.totalSched += uint64(events)

        case <-l.sig:
            acked += l.handleBatchSig()
            if acked > 0 {
                acks = l.broker.acks
            }
        }

        // log.Debug("ackloop INFO")
        // log.Debug("ackloop:   total events scheduled = ", l.totalSched)
        // log.Debug("ackloop:   total events ack = ", l.totalACK)
        // log.Debug("ackloop:   total batches scheduled = ", l.batchesSched)
        // log.Debug("ackloop:   total batches ack = ", l.batchesACKed)

        l.sig = l.lst.channel()
        // if l.sig == nil {
        //  log.Debug("ackloop: no ack scheduled")
        // } else {
        //  log.Debug("ackloop: schedule ack: ", l.lst.head.seq)
        // }
    }
}
```

lst := <-l.broker.scheduledACKs就是获取到那个请求是新建的ackchan的channel，也就是获取到了batch回调的ack的函数的信号。也就是<-l.sig获取到了信号，调用handleBatchSig

```
// handleBatchSig collects and handles a batch ACK/Cancel signal. handleBatchSig
// is run by the ackLoop.
func (l *ackLoop) handleBatchSig() int {
    lst := l.collectAcked()

    count := 0
    for current := lst.front(); current != nil; current = current.next {
        count += current.count
    }

    if count > 0 {
        if e := l.broker.eventer; e != nil {
            e.OnACK(count)
        }

        // report acks to waiting clients
        l.processACK(lst, count)
    }

    for !lst.empty() {
        releaseACKChan(lst.pop())
    }

    // return final ACK to EventLoop, in order to clean up internal buffer
    l.broker.logger.Debug("ackloop: return ack to broker loop:", count)

    l.totalACK += uint64(count)
    l.broker.logger.Debug("ackloop:  done send ack")
    return count
}
```

l.broker.eventer我们可以追溯一下，broker就是创建queue的是newACKLoop的时候传递的，broker也是在这个时候初始化的

```
b := &Broker{
        done:   make(chan struct{}),
        logger: logger,

        // broker API channels
        events:    make(chan pushRequest, chanSize),
        requests:  make(chan getRequest),
        pubCancel: make(chan producerCancelRequest, 5),

        // internal broker and ACK handler channels
        acks:          make(chan int),
        scheduledACKs: make(chan chanList),

        waitOnClose: settings.WaitOnClose,

        eventer: settings.Eventer,
    }
```

eventer是create的时候传递的，create回传的时候是create的方法，真正调用是在pipeline初始化的时候new新建pipeline的时候

```
p.queue, err = queueFactory(&p.eventer)
```

所以说l.broker.eventer就是p.eventer也就是结构体pipelineEventer

```
type pipelineEventer struct {
    mutex      sync.Mutex
    modifyable bool

    observer  queueObserver
    waitClose *waitCloser
    cb        *pipelineEventCB
}
```

我们在pipeline中设置registry的时候p.eventer.cb就是我们创建的pipelineEventCB：p.eventer.cb = cb

到这边可以看出e.OnACK(count)就是调用pipelineEventer的成员函数OnACK

```
func (e *pipelineEventer) OnACK(n int) {
    e.observer.queueACKed(n)

    if wc := e.waitClose; wc != nil {
        wc.dec(n)
    }
    if e.cb != nil {
        e.cb.reportQueueACK(n)
    }
}
```

这个时候就调用我们最初用的pipelineEventCB的reportQueueACK函数，就是将acked发送到了p.acks <- acked中，这个时候pipelineEventCB的监听程序监听到acks信号。

收到ack的channel信息，调用collect函数

```go
func (p *pipelineEventCB) collect(count int) (exit bool) {
    var (
        signalers []chan struct{}
        data      []interface{}
        acked     int
        total     int
    )

    for acked < count {
        var msg eventsDataMsg
        select {
        case msg = <-p.events:
        case msg = <-p.droppedEvents:
        case <-p.done:
            exit = true
            return
        }

        if msg.sig != nil {
            signalers = append(signalers, msg.sig)
        }
        total += msg.total
        acked += msg.acked

        if count-acked < 0 {
            panic("ack count mismatch")
        }

        switch p.mode {
        case eventsACKMode:
            data = append(data, msg.data...)

        case lastEventsACKMode:
            if L := len(msg.data); L > 0 {
                data = append(data, msg.data[L-1])
            }
        }
    }

    // signal clients we processed all active ACKs, as reported by queue
    for _, sig := range signalers {
        close(sig)
    }
    p.reportEventsData(data, total)
    return
}
```

重pipelineEventCB中的events和droppedEvents中读取数据信息，然后进行上报reportEventsData

```
func (p *pipelineEventCB) reportEventsData(data []interface{}, total int) {
    // report ACK back to the beat
    switch p.mode {
    case countACKMode:
        p.handler.ACKCount(total)
    case eventsACKMode:
        p.handler.ACKEvents(data)
    case lastEventsACKMode:
        p.handler.ACKLastEvents(data)
    }
}
```

到这边就调用到一开始的ACKEvents函数，对数据进行处理，其实这些数据就是[]file.State文件信息。在创建pipelineEventCB的时候，也就是在pipeline使用set函数的时候，我们这边传递的是ACKEvents，所以调用的是newEventACKer(finishedLogger, registrarChannel)这个结构体的ackEvents函数。

```
func newEventACKer(stateless statelessLogger, stateful statefulLogger) *eventACKer {
    return &eventACKer{stateless: stateless, stateful: stateful, log: logp.NewLogger("acker")}
}

func (a *eventACKer) ackEvents(data []interface{}) {
    stateless := 0
    states := make([]file.State, 0, len(data))
    for _, datum := range data {
        if datum == nil {
            stateless++
            continue
        }

        st, ok := datum.(file.State)
        if !ok {
            stateless++
            continue
        }

        states = append(states, st)
    }

    if len(states) > 0 {
        a.log.Debugw("stateful ack", "count", len(states))
        a.stateful.Published(states)
    }

    if stateless > 0 {
        a.log.Debugw("stateless ack", "count", stateless)
        a.stateless.Published(stateless)
    }
}
```

调用的这个registrarLogger的Published来完成文件状态的推送

```
func (l *registrarLogger) Published(states []file.State) {
    select {
    case <-l.done:
        // set ch to nil, so no more events will be send after channel close signal
        // has been processed the first time.
        // Note: nil channels will block, so only done channel will be actively
        //       report 'closed'.
        l.ch = nil
    case l.ch <- states:
    }
}
```

> 文件持久化

registrarLogger的channel获取的信息是如何处理的？其实是Registrar的channel接受到了信息，在一开始Registrar就启动了监听channel。

```
for {
    select {
    case <-r.done:
        logp.Info("Ending Registrar")
        return
    case <-flushC:
        flushC = nil
        timer.Stop()
        r.flushRegistry()
    case states := <-r.Channel:
        r.onEvents(states)
        if r.flushTimeout <= 0 {
            r.flushRegistry()
        } else if flushC == nil {
            timer = time.NewTimer(r.flushTimeout)
            flushC = timer.C
        }
    }
```

接收到文件状态后调用onEvents

```
// onEvents processes events received from the publisher pipeline
func (r *Registrar) onEvents(states []file.State) {
    r.processEventStates(states)
    r.bufferedStateUpdates += len(states)

    // check if we need to enable state cleanup
    if !r.gcEnabled {
        for i := range states {
            if states[i].TTL >= 0 || states[i].Finished {
                r.gcEnabled = true
                break
            }
        }
    }

    logp.Debug("registrar", "Registrar state updates processed. Count: %v", len(states))

    // new set of events received -> mark state registry ready for next
    // cleanup phase in case gc'able events are stored in the registry.
    r.gcRequired = r.gcEnabled
}
```

通过processEventStates来处理

```
// processEventStates gets the states from the events and writes them to the registrar state
func (r *Registrar) processEventStates(states []file.State) {
    logp.Debug("registrar", "Processing %d events", len(states))

    ts := time.Now()
    for i := range states {
        r.states.UpdateWithTs(states[i], ts)
        statesUpdate.Add(1)
    }
}
```

然后就是states的更新UpdateWithTs

```
// UpdateWithTs updates a state, assigning the given timestamp.
// If previous state didn't exist, new one is created
func (s *States) UpdateWithTs(newState State, ts time.Time) {
    s.Lock()
    defer s.Unlock()

    id := newState.ID()
    index := s.findPrevious(id)
    newState.Timestamp = ts

    if index >= 0 {
        s.states[index] = newState
    } else {
        // No existing state found, add new one
        s.idx[id] = len(s.states)
        s.states = append(s.states, newState)
        logp.Debug("input", "New state added for %s", newState.Source)
    }
}
```

到这边states的状态就发生变化，再来看看states的初始化操作，其实就是在Registrar的New的时候调用了NewStates进行了初始化

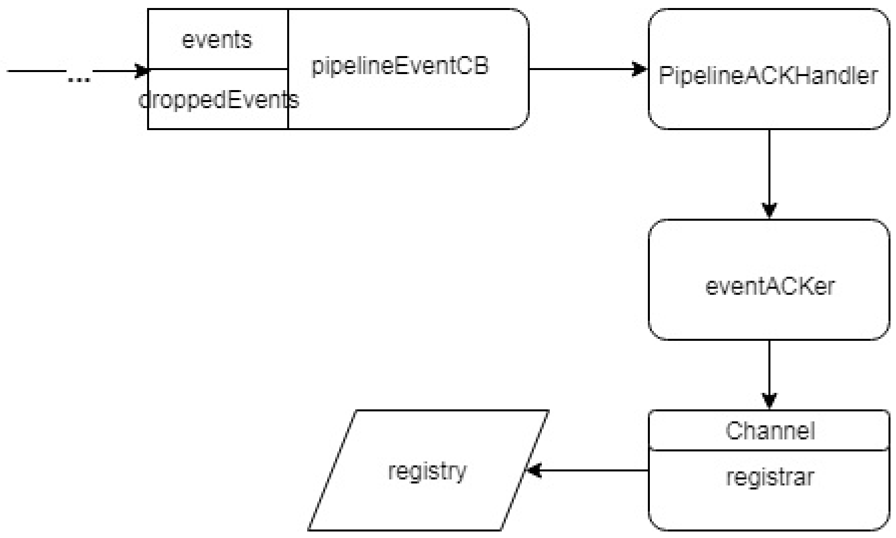
```
r := &Registrar{
    registryFile: dataFile,
    fileMode:     cfg.Permissions,
    done:         make(chan struct{}),
    states:       file.NewStates(),
    Channel:      make(chan []file.State, 1),
    flushTimeout: cfg.FlushTimeout,
    out:          out,
    wg:           sync.WaitGroup{},
}
```

在Registrar的for循环的时候，定时会对状态进行写文件操作，调用flushRegistry的writeRegistry来完成文件的持久化

```go
func (r *Registrar) flushRegistry() {
    if err := r.writeRegistry(); err != nil {
        logp.Err("Writing of registry returned error: %v. Continuing...", err)
    }

    if r.out != nil {
        r.out.Published(r.bufferedStateUpdates)
    }
    r.bufferedStateUpdates = 0
}

// writeRegistry writes the new json registry file to disk.
func (r *Registrar) writeRegistry() error {
    // First clean up states
    r.gcStates()
    states := r.states.GetStates()
    statesCurrent.Set(int64(len(states)))

    registryWrites.Inc()

    tempfile, err := writeTmpFile(r.registryFile, r.fileMode, states)
    if err != nil {
        registryFails.Inc()
        return err
    }

    err = helper.SafeFileRotate(r.registryFile, tempfile)
    if err != nil {
        registryFails.Inc()
        return err
    }

    logp.Debug("registrar", "Registry file updated. %d states written.", len(states))
    registrySuccess.Inc()

    return nil
}

func writeTmpFile(baseName string, perm os.FileMode, states []file.State) (string, error) {
    logp.Debug("registrar", "Write registry file: %s (%v)", baseName, len(states))

    tempfile := baseName + ".new"
    f, err := os.OpenFile(tempfile, os.O_RDWR|os.O_CREATE|os.O_TRUNC|os.O_SYNC, perm)
    if err != nil {
        logp.Err("Failed to create tempfile (%s) for writing: %s", tempfile, err)
        return "", err
    }

    defer f.Close()

    encoder := json.NewEncoder(f)

    if err := encoder.Encode(states); err != nil {
        logp.Err("Error when encoding the states: %s", err)
        return "", err
    }
    // Commit the changes to storage to avoid corrupt registry files
    if err = f.Sync(); err != nil {
        logp.Err("Error when syncing new registry file contents: %s", err)
        return "", err
    }

    return tempfile, nil
}
```

总结

对以上的过程最一个简单的总结



特殊情况

1.如果filebeat异常重启，每次采集harvester启动的时候都会读取registry文件，从上次记录的状态继续采集，确保不会从头开始重复发送所有的日志文件。 当然，如果日志发送过程中，还没来得及返回ack，filebeat就挂掉，registry文件肯定不会更新至最新的状态，那么下次采集的时候，这部分的日志就会重复发送，所以这意味着filebeat只能保证at least once，无法保证不重复发送。 还有一个比较异常的情况是，linux下如果老文件被移除，新文件马上创建，很有可能它们有相同的inode，而由于filebeat根据inode来标志文件记录采集的偏移，会导致registry里记录的其实是被移除的文件State状态，这样新的文件采集却从老的文件Offset开始，从而会遗漏日志数据。 为了尽量避免inode被复用的情况，同时防止registry文件随着时间增长越来越大，建议使用clean_inactive和clean_remove配置将长时间未更新或者被删除的文件State从registry中移除。

2.在harvester读取日志中，会更新registry的状态处理一些异常场景。例如，如果一个日志文件被清空，filebeat会在下一次Reader.Next方法中返回ErrFileTruncate异常，将inode标志文件的Offset置为0，结束这次harvester，重新启动新的harvester，虽然文件不变，但是registry中的Offset为0，采集会从头开始。

3.如果使用容器部署filebeat，需要将registry文件挂载到宿主机上，否则容器重启后registry文件丢失，会使filebeat从头开始重复采集日志文件。

日志重复

Filebeat对于收集到的数据（即event）的传输保证的是”at least once”，而不是”exactly once”，也就是Filebeat传输的数据是有可能有重复的。这里我们讨论一下可能产生重复数据的一些场景，我大概将其分为两类。

第一类：Filebeat重传导致数据重复。重传是因为Filebeat要保证数据至少发送一次，进而避免数据丢失。具体来说就是每条event发送到output后都要等待ack，只有收到ack了才会认为数据发送成功，然后将状态记录到registry。当然实际操作的时候为了高效是批量发送，批量确认的。而造成重传的场景（也就是没有收到ack）非常多，而且很多都不可避免，比如后端不可达、网络传输失败、程序突然挂掉等等。

第二类：配置不当或操作不当导致文件重复收集。Filebeat感知文件有没有被收集过靠的是registry文件里面记录的状态，如果一个文件已经被收集过了，但因为各种原因它的状态从registry文件中被移除了，而恰巧这个文件还在收集范围内，那就会再收集一次。

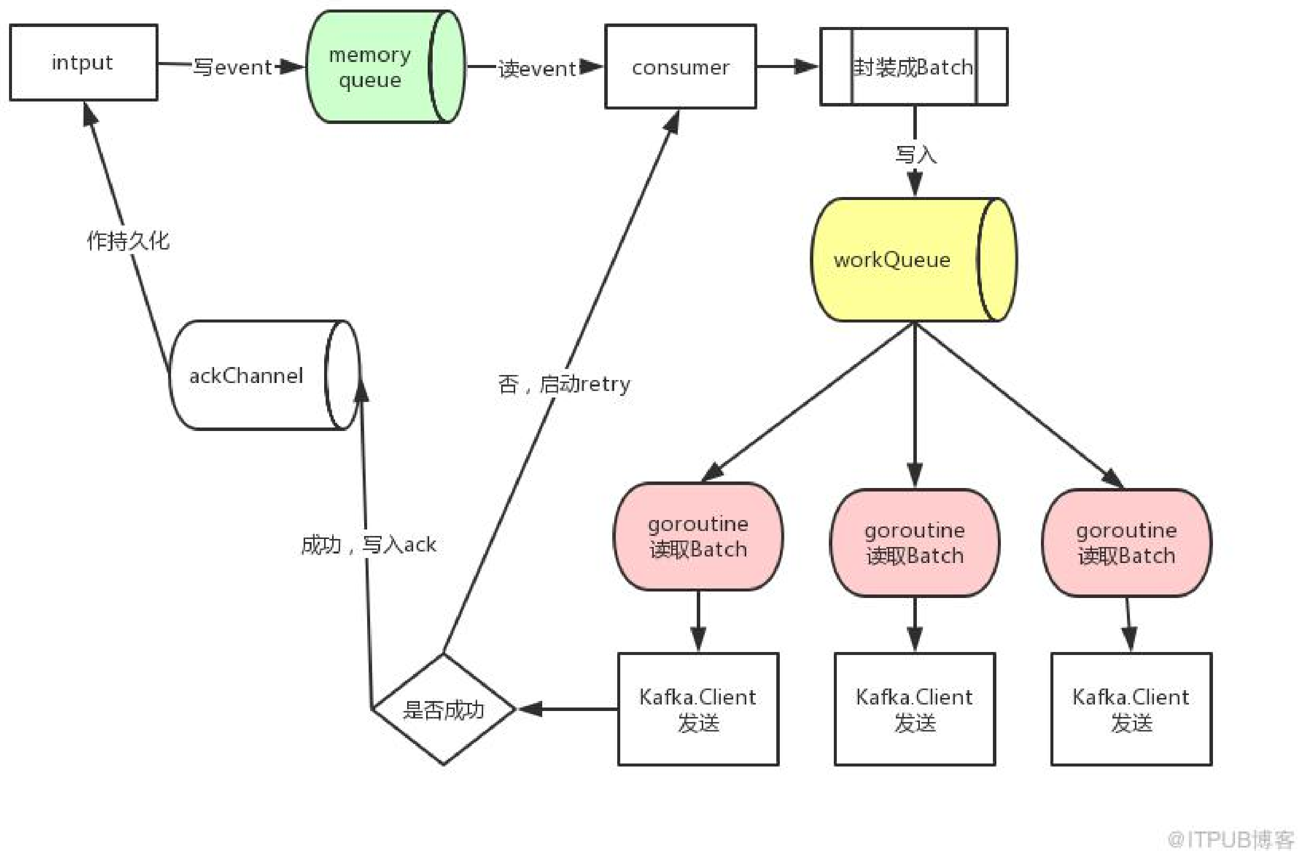
对于第一类产生的数据重复一般不可避免，而第二类可以避免，但总的来说，Filebeat提供的是at least once的机制，所以我们在使用时要明白数据是可能重复的。如果业务上不能接受数据重复，那就要在Filebeat之后的流程中去重。

数据丢失

1. inode重用的问题
2. 如果一个文件达到了限制（比如大小），不是重新创建一个新的文件写，而是将这个文件truncate掉继续复用（当然实际中这种场景好像比较少，但也并非没有），Filebeat下次来检查这个文件是否有变动的时候，这个文件的大小如果大于之前记录的offset，也会发生上面的情况。这个问题在github上面是有issue的，但目前还没有解决，官方回复是Filebeat的整个机制在重构中。
3. 还有一些其它情况，比如文件数太多，Filebeat的处理能力有限，在还没来得及处理的时候这些文件就被删掉了（比如rotate给老化掉了）也会造成数据丢失。还有就是后端不可用，所以Filebeat还在重试，但源文件被删了，那数据也就丢了。因为Filebeat的重试并非一直发送已经收集到内存里面的event，必要的时候会重新从源文件读，比如程序重启。这些情况的话，只要不限制Filebeat的收集能力，同时保证后端的可用性，网络的可用性，一般问题不大。

总结

其实重数据发送到内存队列中这一套完整的功能就是由libbeat完成的，正常流程如下



日志采集状态监控

我们之前讲到 Registrar 会记录每个文件的状态，当 Filebeat 启动时，会从 Registrar 恢复文件处理状态。

其实在 filebeat 运行过程中，Input 组件也记录了文件状态。不一样的是，Registrar 是持久化存储，而 Input 中的文件状态仅表示当前文件的读取偏移量，且修改时不会同步到磁盘中。

每次，Filebeat 刚启动时，Input 都会载入 Registrar 中记录的文件状态，作为初始状态。Input 中的状态有两个非常重要：

```
offset: 代表文件当前读取的 offset，从 Registrar 中初始化。Harvest 读取文件后，会同时修改 offse
t。
finished: 代表该文件对应的 Harvester 是否已经结束，Harvester 开始时置为 false，结束时置为 true
。
```

对于每次定时扫描到的文件，概括来说，会有三种大的情况：

```
Input 找不到该文件状态的记录，说明是新增文件，则开启一个 Harvester，从头开始解析该文件
如果可以找到文件状态，且 finished 等于 false。这个说明已经有了一个 Harvester 在处理了，这种情况
直接忽略就好了。
如果可以找到文件状态，且 finished 等于 true。说明之前有 Harvester 处理过，但已经处理结束了。
```

对于这种第三种情况，我们需要考虑到一些异常情况，Filebeat 是这么处理的：

```
如果 offset 大于当前文件大小：说明文件被 Truncate 过，此时按做一个新文件处理，直接从头开始解析
该文件
如果 offset 小于当前文件大小，说明文件内容有新增，则从上次 offset 处继续读即可。
```

对于第二种情况，Filebeat 似乎有一个逻辑上的问题: 如果文件被 Truncate 过，后来又新增了数据，且文件大小也比之前 offset 大，那么 Filebeat 是检查不出来这个问题的。

句柄保持

Filebeat 甚至可以处理文件名修改的问题。即使一个日志的文件名被修改过，Filebeat 重启后，也能找到该文件，从上次读过的地方继续读。

这是因为 Filebeat 除了在 Registrar 存储了文件名，还存储了文件的唯一标识。对于 Linux 来说，这个文件的唯一标识就是该文件的 inode ID + device ID。

重载

重载是在Crawler中启动的，首先是新建的InputsFactory的结构体

```
c.InputsFactory = input.NewRunnerFactory(c.out, r, c.beatDone)

// RunnerFactory is a factory for registrars
type RunnerFactory struct {
    outlet    channel.Factory
    registrar *registrar.Registrar
    beatDone  chan struct{}
}

// NewRunnerFactory instantiates a new RunnerFactory
func NewRunnerFactory(outlet channel.Factory, registrar *registrar.Registrar, beatDone chan st
ruct{}) *RunnerFactory {
    return &RunnerFactory{
        outlet:    outlet,
        registrar: registrar,
        beatDone:  beatDone,
    }
}
```

然后如果配置了重载，就会新建Reloader结构体

```
// Reloader is used to register and reload modules
type Reloader struct {
    pipeline beat.Pipeline
    config   DynamicConfig
    path     string
    done     chan struct{}
    wg       sync.WaitGroup
}
```

新建的过程如下

```
if configInputs.Enabled() {
    c.inputReloader = cfgfile.NewReloader(pipeline, configInputs)
    if err := c.inputReloader.Check(c.InputsFactory); err != nil {
        return err
    }

    go func() {
        c.inputReloader.Run(c.InputsFactory)
    }()
}
```

启动reloader的run方法并且将RunnerFactory作为参数传递进去

```
// Run runs the reloader
func (rl *Reloader) Run(runnerFactory RunnerFactory) {
    logp.Info("Config reloader started")

    list := NewRunnerList("reload", runnerFactory, rl.pipeline)

    rl.wg.Add(1)
    defer rl.wg.Done()

    // Stop all running modules when method finishes
    defer list.Stop()

    gw := NewGlobWatcher(rl.path)

    // If reloading is disable, config files should be loaded immediately
    if !rl.config.Reload.Enabled {
        rl.config.Reload.Period = 0
    }

    overwriteUpdate := true

    for {
        select {
        case <-rl.done:
            logp.Info("Dynamic config reloader stopped")
            return

        case <-time.After(rl.config.Reload.Period):
            debugf("Scan for new config files")
            configReloads.Add(1)

            //扫描所有的配置文件
            files, updated, err := gw.Scan()
            if err != nil {
                // In most cases of error, updated == false, so will continue
                // to next iteration below
                logp.Err("Error fetching new config files: %v", err)
            }

            // no file changes
            if !updated && !overwriteUpdate {
                overwriteUpdate = false
                continue
            }

            // Load all config objects 加载所有配置文件
            configs, _ := rl.loadConfigs(files)

            debugf("Number of module configs found: %v", len(configs))

            //启动加载程序
            if err := list.Reload(configs); err != nil {
                // Make sure the next run also updates because some runners were not properly
 loaded
                overwriteUpdate = true
            }
        }
        // Path loading is enabled but not reloading. Loads files only once and then stops.
        if !rl.config.Reload.Enabled {
            logp.Info("Loading of config files completed.")
            select {
            case <-rl.done:
                logp.Info("Dynamic config reloader stopped")
                return
            }
        }
    }
}
```

可见有一个定时的循环程序来获取所有的配置文件，交给RunnerList的reload的来加载

```
// Reload the list of runners to match the given state
func (r *RunnerList) Reload(configs []*reload.ConfigWithMeta) error {
    r.mutex.Lock()
    defer r.mutex.Unlock()

    var errs multierror.Errors

    startList := map[uint64]*reload.ConfigWithMeta{}
    //获取正在运行的runner到stopList
    stopList := r.copyRunnerList()

    r.logger.Debugf("Starting reload procedure, current runners: %d", len(stopList))

    // diff current & desired state, create action lists
    for _, config := range configs {
        hash, err := HashConfig(config.Config)
        if err != nil {
            r.logger.Errorf("Unable to hash given config: %s", err)
            errs = append(errs, errors.Wrap(err, "Unable to hash given config"))
            continue
        }

        //如果配置文件还在，就重stopList中删除，继续采集，剩下的在stopList中的下面会停止采集
        if _, ok := stopList[hash]; ok {
            delete(stopList, hash)
        } else {
            //如果不在stopList中，说明是新的文件，如果不是重复的就加入到startList中，下来开始采
集
            if _,ok := r.runners[hash]; !ok{
                startList[hash] = config
            }
        }
    }

    r.logger.Debugf("Start list: %d, Stop list: %d", len(startList), len(stopList))

    // Stop removed runners
    for hash, runner := range stopList {
        r.logger.Debugf("Stopping runner: %s", runner)
        delete(r.runners, hash)
        go runner.Stop()
    }

    // Start new runners
    for hash, config := range startList {
        // Pass a copy of the config to the factory, this way if the factory modifies it,
        // that doesn't affect the hash of the original one.
        c, _ := common.NewConfigFrom(config.Config)
        runner, err := r.factory.Create(r.pipeline, c, config.Meta)
        if err != nil {
            r.logger.Errorf("Error creating runner from config: %s", err)
            errs = append(errs, errors.Wrap(err, "Error creating runner from config"))
            continue
        }

        r.logger.Debugf("Starting runner: %s", runner)
        r.runners[hash] = runner
        runner.Start()
    }

    return errs.Err()
}
```

这边的create就是上面传进来的InputsFactory的结构体的成员函数

```
// Create creates a input based on a config
func (r *RunnerFactory) Create(
    pipeline beat.Pipeline,
    c *common.Config,
    meta *common.MapStrPointer,
) (cfgfile.Runner, error) {
    connector := r.outlet(pipeline)
    p, err := New(c, connector, r.beatDone, r.registrar.GetStates(), meta)
    if err != nil {
        // In case of error with loading state, input is still returned
        return p, err
    }

    return p, nil
}
```

看看new函数

```
// New instantiates a new Runner
func New(
    conf *common.Config,
    connector channel.Connector,
    beatDone chan struct{},
    states []file.State,
    dynFields *common.MapStrPointer,
) (*Runner, error) {
    input := &Runner{
        config:   defaultConfig,
        wg:       &sync.WaitGroup{},
        done:     make(chan struct{}),
        Once:     false,
        beatDone: beatDone,
    }

    var err error
    if err = conf.Unpack(&input.config); err != nil {
        return nil, err
    }

    var h map[string]interface{}
    conf.Unpack(&h)
    input.ID, err = hashstructure.Hash(h, nil)
    if err != nil {
        return nil, err
    }

    var f Factory
    f, err = GetFactory(input.config.Type)
    if err != nil {
        return input, err
    }

    context := Context{
        States:        states,
        Done:          input.done,
        BeatDone:      input.beatDone,
        DynamicFields: dynFields,
        Meta:          nil,
    }
    var ipt Input
    ipt, err = f(conf, connector, context)
    if err != nil {
        return input, err
    }
    input.input = ipt

    return input, nil
}
```

可以看见就是新建流程中的新建runner，下面调用

```
runner.Start()
```

就是正常的采集流程，到这边重载也就结束了。

基本使用与特性

filebeat自动reload更新

目前filebeat支持reload input配置，module配置，但reload的机制只有定时更新。

在配置中打开reload.enable之后，还可以配置reload.period表示自动reload配置的时间间隔。

filebeat在启动时，会创建一个专门用于reload的协程。对于每个正在运行的harvester，filebeat会将其加入一个全局的Runner列表，每次到了定时的间隔后，会触发一次配置文件的diff判断，如果是需要停止的加入stopRunner列表，然后逐个关闭，新的则加入startRunner列表，启动新的Runner。

filebeat对kubernetes的支持

filebeat官方文档提供了在kubernetes下基于daemonset的部署方式，最主要的一个配置如下所示：

```
- type: docker
      containers.ids:
      - "*"
      processors:
        - add_kubernetes_metadata:
            in_cluster: true
```

即设置输入input为docker类型。由于所有的容器的标准输出日志默认都在节点的/var/lib/docker/containers//*-json.log路径，所以本质上采集的是这类日志文件。

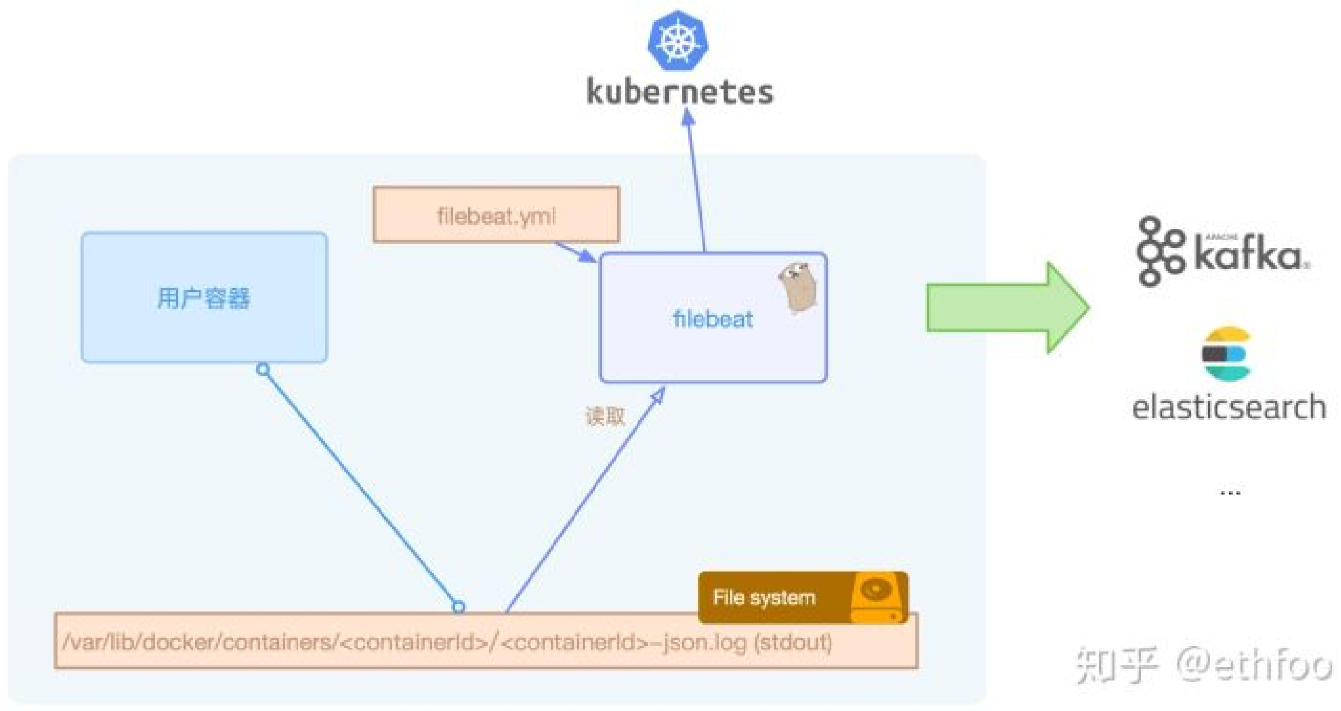
和传统的部署方式有所区别的是，如果服务部署在kubernetes上，我们查看和检索日志的维度不能仅仅局限于节点和服务，还需要有podName，containerName等，所以每条日志我们都需要打标增加kubernetes的元信息才发送至后端。

filebeat会在配置中增加了add_kubernetes_metadata的processor的情况下，启动监听kubernetes的watch服务，监听所有kubernetes pod的变更，然后将归属本节点的pod最新的事件同步至本地的缓存中。

节点上一旦发生容器的销毁创建，/var/lib/docker/containers/下会有目录的变动，filebeat根据路径提取出containerId，再根据containerId从本地的缓存中找到pod信息，从而可以获取到podName、label等数据，并加到日志的元信息fields中。

filebeat还有一个beta版的功能autodiscover，autodiscover的目的是把分散到不同节点上的filebeat配置文件集中管理。目前也支持kubernetes作为provider，本质上还是监听kubernetes事件然后采集docker的标准输出文件。

大致架构如下所示：



但是在实际生产环境使用中，仅采集容器的标准输出日志还是远远不够，我们往往还需要采集容器挂载出来的自定义日志目录，还需要控制每个服务的日志采集方式以及更多的定制化功能。

性能分析与调优

虽然beats系列主打轻量级，虽然用golang写的filebeat的内存占用确实比较基于jvm的logstash等好太多，但是事实告诉我们其实没那么简单。

正常启动filebeat，一般确实只会占用3、40MB内存，但是在轻舟容器云上偶发性的我们也会发现某些节点上的filebeat容器内存占用超过配置的pod limit限制（一般设置为200MB），并且不停的触发的OOM。

究其原因，一般容器化环境中，特别是裸机上运行的容器个数可能会比较多，导致创建大量的harvester去采集日志。如果没有很好的配置filebeat，会有较大概率导致内存急剧上升。 当然，filebeat内存占据较大的部分还是memqueue，所有采集到的日志都会先发送至memqueue聚集，再通过output发送出去。每条日志的数据在filebeat中都被组装为event结构，filebeat默认配置的memqueue缓存的event个数为4096，可通过queue.mem.events设置。默认最大的一条日志的event大小限制为10MB，可通过max_bytes设置。4096 * 10MB = 40GB，可以想象，极端场景下，filebeat至少占据40GB的内存。特别是配置了multiline多行模式的情况下，如果multiline配置有误，单个event误采集为上千条日志的数据，很可能导致memqueue占据了大量内存，致使内存爆炸。

所以，合理的配置日志文件的匹配规则，限制单行日志大小，根据实际情况配置memqueue缓存的个数，才能在实际使用中规避filebeat的内存占用过大的问题。

有些文章说filebeat内存消耗很少,不会超过100M, 这简直是不负责任的胡说,假如带着这样的认识把filebeat部署到生产服务器上就等着哭吧.

那怎么样才能避免以上内存灾难呢?

1. 每个日志生产环境生产的日志大小,爆发量都不一样, 要根据自己的日志特点设定合适的event值;什么叫合适,至少能避免内存>200MB的灾难;
2. 在不知道日志实际情况(单条大小,爆发量), 务必把event设置上,建议128或者256;
3. 合理的配置日志文件的匹配规则，是否因为通配符的原因，造成同时监控数量巨大的文件，这种情况应该避免用通配符监控无用的文件。
4. 规范日志，限制单行日志大小，是否文件的单行内容巨大，确定是否需要改造文件内容，或者将其过滤
5. 限制cpu

上面的一系列操作可以做如下配置

```
max_procs: 2
queue:
  mem:
    events: 512
    flush.min_events: 256
```

限制cpu为2core，内存最大为512*10M～=5G

限制cpu的配置

```
max_procs，限制filebeat的进程数量，其实是内核数，建议手动设为1
```

限制内存的配置

```
queue.mem.events消息队列的大小，默认值是4096，这个参数在6.0以前的版本是spool-size，通过命令行，在启动时进行配置
max_message_bytes 单条消息的大小, 默认值是10M
```

filebeat最大的可能占用的内存是max_message_bytes * queue.mem.events = 40G，考虑到这个queue是用于存储encode过的数据，raw数据也是要存储的，所以，在没有对内存进行限制的情况下，最大的内存占用情况是可以达到超过80G。

内存使用过多的情况

> 非常频繁的rotate日志

对于实时大量产生内容的文件，比如日志，常用的做法往往是将日志文件进行rotate，根据策略的不同，每隔一段时间或者达到固定大小之后，将日志rotate。 这样，在文件目录下可能会产生大量的日志文件。 如果我们使用通配符的方式，去监控该目录，则filebeat会启动大量的harvester实例

去采集文件。但是，请记住，我这里不是说这样一定会产生内存泄漏，只是在这里观测到了内存泄漏而已，不是说这是造成内存泄漏的原因。

当filebeat运行了几个月之后，占用了超过10个G的内存。

> 因为multiline导致内存占用过多

multiline.pattern: ‘^[[:space:]]+|^Caused by:|^.+Exception:|^\d+\serror，比如这个配置，认为空格或者制表符开头的line是上一行的附加内容，需要作为多行模式，存储到同一个event当中。当你监控的文件刚巧在文件的每一行带有一个空格时，会错误的匹配多行，造成filebeat解析过后，单条event的行数达到了上千行，大小达到了10M，并且在这过程中使用的是正则表达式，每一条event的处理都会极大的消耗内存。因为大多数的filebeat output是需应答的，buffer这些event必然会大量的消耗内存。

解读日志中的监控数据

其实filebeat的日志，已经包含了很多参数用于实时观测filebeat的资源使用情况，（下面是6.0版本的，6.5版本之后，整个日志格式变了，从kv格式变成了json对象格式）

里面的参数主要分成三个部分：

```
beat.*，包含memstats.gc_next，memstats.memory_alloc，memstats.memory_total，这个是所有beat组件都有的指标，是filebeat继承来的，主要是内存相关的，我们这里特别关注memstats.memory_alloc，alloc的越多，占用内存越大
filebeat.*，这部分是filebeat特有的指标，通过event相关的指标，我们知道吞吐，通过harvester，我们知道正在监控多少个文件，未消费event堆积的越多，havester创建的越多，消耗内存越大
libbeat.*，也是beats组件通用的指标，包含outputs和pipeline等信息。这里要主要当outputs发生阻塞的时候，会直接影响queue里面event的消费，造成内存堆积
registrar，filebeat将监控文件的状态放在registry文件里面，当监控文件非常多的时候，比如10万个，而且没有合理的设置close_inactive参数，这个文件能达到100M，载入内存后，直接占用内存
```

在6.5之后都是json，但也是kv结构，可以对应查看。

如何对filebeat进行扩展开发

一般情况下filebeat可满足大部分的日志采集需求，但是仍然避免不了一些特殊的场景需要我们对filebeat进行定制化开发，当然filebeat本身的设计也提供了良好的扩展性。 beats目前只提供了像elasticsearch、kafka、logstash等几类output客户端，如果我们想要filebeat直接发送至其他后端，需要定制化开发自己的output。同样，如果需要对日志做过滤处理或者增加元信息，也可以自制processor插件。 无论是增加output还是写个processor，filebeat提供的大体思路基本相同。一般来讲有3种方式：

1.直接fork filebeat，在现有的源码上开发。output或者processor都提供了类似Run、Stop等的接口，只需要实现该类接口，然后在init方法中注册相应的插件初始化方法即可。当然，由于golang中init方法是在import包时才被调用，所以需要在初始化filebeat的代码中手动import。

2.filebeat还提供了基于golang plugin的插件机制，需要把自研的插件编译成.so共享链接库，然后在filebeat启动参数中通过-plugin指定库所在路径。不过实际上一方面golang plugin还不够成熟稳定，一方面自研的插件依然需要依赖相同版本的libbeat库，而且还需要相同的golang版本编译，坑可能更多，不太推荐。

monitor (/tags/monitor/) filebeat (/tags/filebeat/) log (/tags/log/)
collect (/tags/collect/)

相关文章

- 监控metrics系列---- Prometheus Principle (/post/monitor/metrics/prometheus/prometheus-principle/) (2021年01月01日)
- 监控系列---- log (/post/monitor/log/log-scheme/) (2020年08月13日)
- 监控日志系列---- Filebeat (/post/monitor/log/collect/filebeat/filebeat/) (2020年07月08日)
- 监控metrics系列---- Infrastructure监控方案 (/post/monitor/metrics/prometheus/monitor-scheme/infrastructure-base/) (2020年06月13日)
- 监控metrics系列---- K8s监控方案 (/post/monitor/metrics/prometheus/monitor-scheme/k8s-base/) (2020年05月12日)
- 监控日志系列---- loki (/post/monitor/log/loki/loki/) (2020年01月18日)
- 监控metrics系列---- Prometheus Grok_exporter (/post/monitor/metrics/prometheus/exporter/log/grok_exporter/) (2020年01月10日)
- 监控metrics系列---- Prometheus mtail (/post/monitor/metrics/prometheus/exporter/log/mtail/) (2020年01月10日)
- 监控trace系列---- jaeger (/post/monitor/trace/jaeger/) (2019年08月13日)
- 监控trace系列---- zipkin (/post/monitor/trace/zipkin/) (2019年08月13日)