深入学习Etcd

概述

etcd 是一个应用在分布式环境下的 key/value 存储服务。利用 etcd 的特性，应用程序可以在集群中共享信息、配置或作服务发现，etcd 会在集群的各个节点中复制这些数据并保证这些数据始终正确。随着CoreOS和Kubernetes等项目在开源社区日益火热，它们项目中都用到的etcd组件作为一个高可用强一致性的服务发现存储仓库，渐渐为开发人员所关注。

zookeeper是最多与etcd进行比较的工具，ZooKeeper有如下缺点：

- 复杂。ZooKeeper的部署维护复杂，管理员需要掌握一系列的知识和技能；而Paxos强一致性算法也是素来以复杂难懂而闻名于世；另外，ZooKeeper的使用也比较复杂，需要安装客户端，官方只提供了Java和C两种语言的接口。
- Java编写。这里不是对Java有偏见，而是Java本身就偏向于重型应用，它会引入大量的依赖。而运维人员则普遍希望保持强一致、高可用的机器集群尽可能简单，维护起来也不易出错。
- 发展缓慢。Apache基金会项目特有的“Apache Way”在开源界饱受争议，其中一大原因就是由于基金会庞大的结构以及松散的管理导致项目发展缓慢。

而etcd作为一个后起之秀，其优点也很明显。

- 简单。使用Go语言编写部署简单；使用HTTP作为接口使用简单；使用Raft算法保证强一致性让用户易于理解。 数据持久化。etcd默认数据一更新就进行持久化。
- 安全。etcd支持SSL客户端安全认证。

集群搭建

了解基本概念后，我们可以尝试地搭建一个简单的测试集群，直观感受下etcd的功能。

由于方便大家快速部署，下面的例子我只用同一台服务器进行三个成员节点的开启，生产环境不建议这样，生产环境为了容灾，建议一个节点部署在一台服务器上。

我们的测试环境我们IP就先以127.0.0.1表示。

环境准备说明：

运行linux环境：

centos 7

etcd版本

v2.3.6

测试集群服务名称

NAME	ADDRESS	HOSTNAME	PEER PORT	CLIENT PORT
cd0	127.0.0.1	cd0.example.com	2380	2379

cd1 | 127.0.0.1 | cd1.example.com | 2480 | 2479

cd2 | 127.0.0.1 | cd2.example.com | 2580 | 2579

约定以上的名称后，我们可以开始部署我们的etcd测试集群。etcd目前支持配置文件和命令行的实例启动方法，下面的例子采用命令行的启动方式。

我们启动第一个成员节点：

```
$ etcd --name cd0 --initial-advertise-peer-urls http://127.0.0.1:2380 \
  --listen-peer-urls http://127.0.0.1:2380 \
  --listen-client-urls http://192.168.139.134:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://192.168.139.134:2379,http://127.0.0.1:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster cd0=http://127.0.0.1:2380,cd1=http://127.0.0.1:2480,cd2=http://127.0.0.1:2580 \
  --initial-cluster-state new
```

Sh

然后启动第二个成员节点

```
$ etcd --name cd1 --initial-advertise-peer-urls http://127.0.0.1:2480 \
  --listen-peer-urls http://127.0.0.1:2480 \
  --listen-client-urls http://192.168.139.134:2479,http://127.0.0.1:2479 \
```

```
  --advertise-client-urls http://192.168.139.134:2479,http://127.0.0.1:2479 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster cd0=http://127.0.0.1:2380,cd1=http://127.0.0.1:2480,cd2=http://127.0.0.1:2580 \
  --initial-cluster-state new
```
Sh

我们再启动第三个成员节点

```
$ etcd --name cd2 --initial-advertise-peer-urls http://127.0.0.1:2580 \
  --listen-peer-urls http://127.0.0.1:2580 \
  --listen-client-urls http://192.168.139.134:2579,http://127.0.0.1:2579 \
  --advertise-client-urls http://192.168.139.134:2579,http://127.0.0.1:2579 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster cd0=http://127.0.0.1:2380,cd1=http://127.0.0.1:2480,cd2=http://127.0.0.1:2580 \
  --initial-cluster-state new
```
Sh

注：上面的命令中 listen-client-urls、advertise-client-urls 我们分别设置了两个ip地址，其中第一个暴露当前服务器的ip地址出去，主要的目的让“etcd的服务发现客户端”能连接上我们的etcd集群；

当三个节点都启动后，我们的测试集群其实就基本搭建完成了。接下来我们通过一些命令查看环境是否正常运行。

集群常用API

- 查看集群成员的两种方式：
  - API：curl http://127.0.0.1:2379/v2/members
  - 命令：./etcdctl -peers 127.0.0.1:2379 member list
- 查看raft选举出的leader信息：
  - API：curl http://127.0.0.1:2379/v2/stats/leader

正常情况下，第一个命令，终端会输出集群环境中成员的信息：


```
{"members":[{"id":"98f0c6bf64240842","name":"cd2","peerURLs":["http://127.0.0.1:2580"],"clientURLs":
["http://127.0.0.1:2579"]},{"id":"bf9071f4639c75cc","name":"cd0","peerURLs":["http://127.0.0.1:2380"],"clientURLs":
["http://127.0.0.1:2379"]},{"id":"e3ba87c3b4858ef1","name":"cd1","peerURLs":["http://127.0.0.1:2480"],"clientURLs":
["http://127.0.0.1:2479"]}]}
```

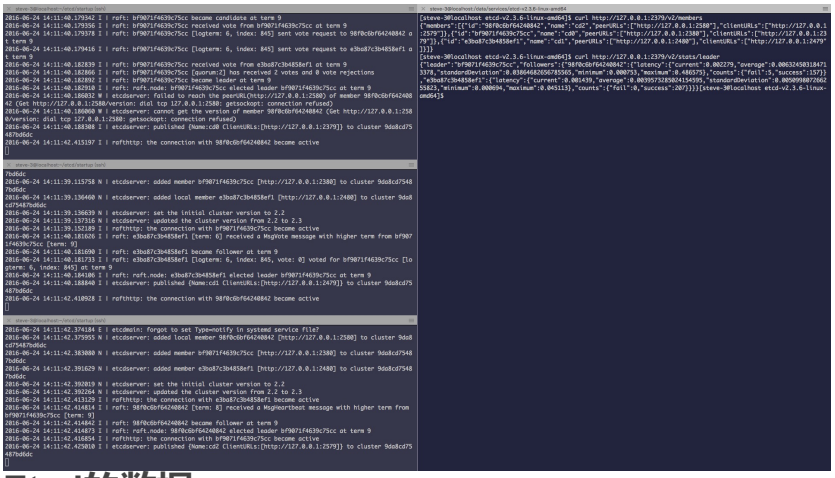
Sh

第二个名称，终端会输出leader节点的信息


```
{"leader":"bf9071f4639c75cc","followers":{"98f0c6bf64240842":{"latency":
{"current":0.001508,"average":0.003802833109017467,"standardDeviation":0.036522391987331815,"minimum":0.00063,"maximum":2.
{"fail":5,"success":7430}},"e3ba87c3b4858ef1":{"latency":
{"current":0.001107,"average":0.0043588797546012 37,"standardDeviation":0.03430699645209604,"minimum":0.000638,"maximum":3.
{"fail":0,"success":8965}}}}
```

Sh

至此，我们的etcd测试集群正常搭建完成。见下图：



Etcd的数据

etcd为我们提供了KV的服务目录存储，所以我们可以通过命令设置value

```
  $ ./etcdctl set /mysite lihaoquan.xyz
```
Sh

获取value的方式同样简单：

```
$ ./etcdctl get /mysite
```

```
> lihaoquan.xyz
```

Sh

这个例子中我们设置了一个key为 `/mysite` ， value为 `lihaoquan.xyz`的数据。

etcd的数据默认会存放在我们的命令工作目录中，我们会发现数据所在的目录，会被分为两个文件夹中：

`snap`: 存放快照数据,etcd防止WAL文件过多而设置的快照，存储etcd数据状态。

`wal`: 存放预写式日志,最大的作用是记录了整个数据变化的全部历程。在etcd中，所有数据的修改在提交前，都要先写入到WAL中。使用WAL进行数据的存储使得etcd拥有两个重要功能。

- 故障快速恢复： 当你的数据遭到破坏时，就可以通过执行所有WAL中记录的修改操作，快速从最原始的数据恢复到数据损坏前的状态。
- 数据回滚（undo）/重做（redo）：因为所有的修改操作都被记录在WAL中，需要回滚或重做，只需要方向或正向执行日志中的操作即可

既然有了WAL实时存储了所有的变更，为什么还需要snapshot呢？随着使用量的增加，WAL存储的数据会暴增，为了防止磁盘很快就爆满，etcd默认每10000条记录做一次snapshot，经过snapshot以后的WAL文件就可以删除。而通过API可以查询的历史etcd操作默认为1000条。

首次启动时，etcd会把启动的配置信息存储到data-dir参数指定的数据目录中。配置信息包括本地节点的ID、集群ID和初始时集群信息。用户需要避免etcd从一个过期的数据目录中重新启动，因为使用过期的数据目录启动的节点会与集群中的其他节点产生不一致（如：之前已经记录并同意Leader节点存储某个信息，重启后又向Leader节点申请这个信息）。所以，为了最大化集群的安全性，一旦有任何数据损坏或丢失的可能性，你就应该把这个节点从集群中移除，然后加入一个不带数据目录的新节点。

节点管理

上文, 我们搭建了一个应用开发测试的etcd”伪集群”，集群中包含了三个工作节点，它们分别是：

cd0 | 127.0.0.1 | cd0.example.com | 2380 | 2379

cd1 | 127.0.0.1 | cd1.example.com | 2480 | 2479

cd2 | 127.0.0.1 | cd2.example.com | 2580 | 2579

在我们日常的开发中，对etcd集群中添加，删除节点是一种很常见的操作。下面我们演示如何往我们目前的节点中添加新的节点和删除它。

添加节点

我们先约定好这个新的节点信息为：

cd3 | 127.0.0.1 | cd3.example.com | 2180 | 2179

添加前，我们先查看目前的集群信息：

$ ./etcdctl -peers 127.0.0.1:2379 member list

终端输出：

bf9071f4639c75cc: name=cd0 peerURLs=http://127.0.0.1:2380 clientURLs=http://127.0.0.1:2379 isLeader=true

98f0c6bf64240842: name=cd2 peerURLs=http://127.0.0.1:2580 clientURLs=http://127.0.0.1:2579 isLeader=false

e3ba87c3b4858ef1: name=cd1 peerURLs=http://127.0.0.1:2480 clientURLs=http://127.0.0.1:2479 isLeader=false

好，我们尝试往集群中添加约定的新节点

$ ./etcdctl member add cd3 http://127.0.0.1:2180

终端输出：

Added member named cd3 with ID ca3e75fab48dd93c to cluster

ETCD_NAME="cd3"

ETCD_INITIAL_CLUSTER="cd2=http://127.0.0.1:2580,cd3=http://127.0.0.1:2180,cd0=http://127.0.0.1:2380,cd1=http://127.0.0.1:2

ETCD_INITIAL_CLUSTER_STATE="existing"

我们再查看下集群当前的信息：

$ ./etcdctl -peers 127.0.0.1:2379 member list

终端输出：

98f0c6bf64240842: name=cd2 peerURLs=http://127.0.0.1:2580 clientURLs=http://127.0.0.1:2579 isLeader=false

bf9071f4639c75cc: name=cd0 peerURLs=http://127.0.0.1:2380 clientURLs=http://127.0.0.1:2379 isLeader=true

e3ba87c3b4858ef1: name=cd1 peerURLs=http://127.0.0.1:2480 clientURLs=http://127.0.0.1:2479 isLeader=false

ca3e75fab48dd93c[unstarted]: peerURLs=http://127.0.0.1:2680

从终端的信息中，我们发现我们打算添加的节点处于【unstarted】状态，原因很简单，我们添加后，并没有启动它。

启动节点

我们启动第四个成员节点(记得带上刚刚的创建后获得的环境变量)：

$ export ETCD_NAME="cd3"

$ export

ETCD_INITIAL_CLUSTER="cd2=http://127.0.0.1:2580,cd3=http://127.0.0.1:2180,cd0=http://127.0.0.1:2380,cd1=http://127.0.0.1:24

$ export ETCD_INITIAL_CLUSTER_STATE=existing

然后启动名称为cd3的成员

```
./etcd --name cd3 --listen-client-urls http://127.0.0.1:2179 \
  --advertise-client-urls http://127.0.0.1:2179 \
  --listen-peer-urls http://127.0.0.1:2180 \
  --initial-advertise-peer-urls http://127.0.0.1:2180 \
  --initial-cluster-state existing \
  --initial-cluster cd2=http://127.0.0.1:2580,cd0=http://127.0.0.1:2380,cd3=http://127.0.0.1:2179,cd1=http://127.0.0.1:2480 \
  --initial-cluster-token etcd-cluster-1
```

删除节点

如果我们需要删除刚刚的节点，可以执行以下命令：

$ ./etcdctl -peers 127.0.0.1:2379 member remove ca3e75fab48dd93c

再次查看集群信息：

$ ./etcdctl -peers 127.0.0.1:2379 member list

终端输出：

98f0c6bf64240842: name=cd2 peerURLs=http://127.0.0.1:2580 clientURLs=http://127.0.0.1:2579 isLeader=false

bf9071f4639c75cc: name=cd0 peerURLs=http://127.0.0.1:2380 clientURLs=http://127.0.0.1:2379 isLeader=true

e3ba87c3b4858ef1: name=cd1 peerURLs=http://127.0.0.1:2480 clientURLs=http://127.0.0.1:2479 isLeader=false

这时，节点已经成功删除了

对于异常节点的替换，其实也同理，无非也是删除旧节点，新增新节点，然后启动新节点，etcd会自动同步数据到新的结点。

服务发现

etcd：从应用场景到实现原理的全方位解读这篇文章，作者对服务发现的定义为：

服务发现要解决的也是分布式系统中最常见的问题之一，即在同一个分布式集群中的进程或服务，要如何才能找到对方并建立连接。本质上来说，服务发现就是想要了解集群中是否有进程在监听udp或tcp端口，并且通过名字就可以查找和连接。要解决服务发现的问题，需要有下面三大支柱，缺一不可。

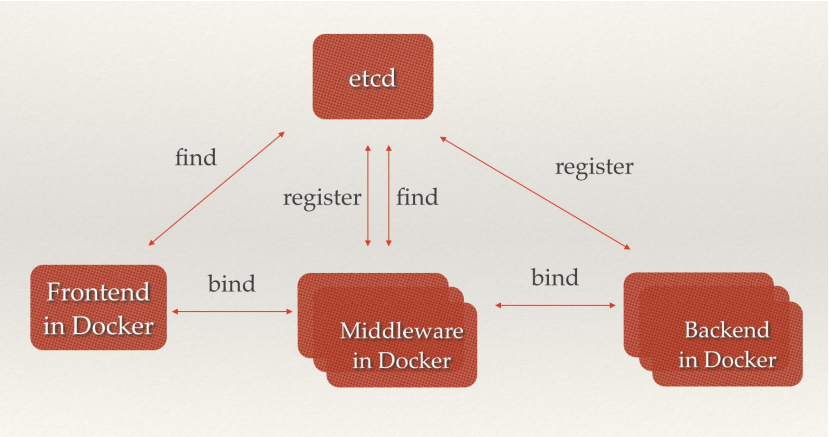
1. 一个强一致性、高可用的服务存储目录。基于Raft算法的etcd天生就是这样一个强一致性高可用的服务存储目录。
2. 一种注册服务和监控服务健康状态的机制。用户可以在etcd中注册服务，并且对注册的服务设置key TTL，定时保持服务的心跳以达到监控健康状态的效果。
3. 一种查找和连接服务的机制。通过在etcd指定的主题下注册的服务也能在对应的主题下查找到。为了确保连接，我们可以在每个服务机器上都部署一个Proxy模式的etcd，这样就可以确保能访问etcd集群的服务都能互相连接。

etcd服务发现使用场景

我们来看服务发现对应的具体场景：

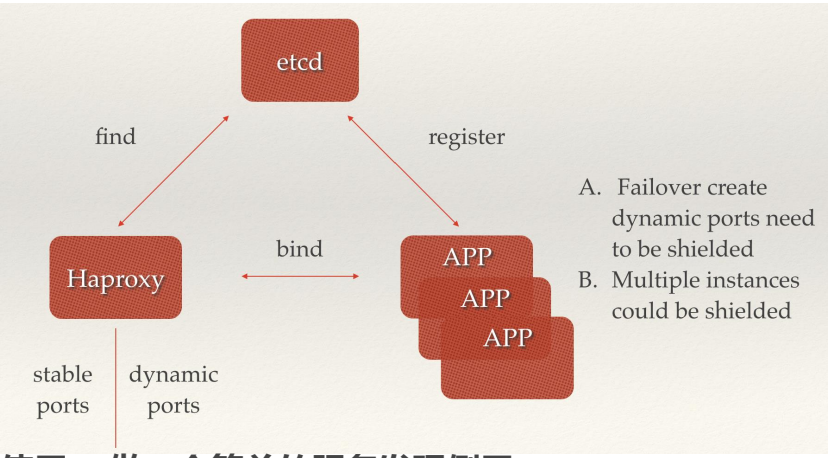
- 微服务协同工作架构中，服务动态添加。随着Docker容器的流行，多种微服务共同协作，构成一个相对功能强大的架构的案例越来越多。透明化的动态添加这些服务的需求也日益强烈。通过服务发现机制，在etcd中注册某个服务名字的目录，在该目录下存储可用的服务节点的IP。在使用服务的过程中，只要从服务目录下查找可用的服务节点去使用即可。

微服务协同架构如下图



- paaS平台中应用多实例与实例故障重启透明化。PaaS平台中的应用一般都有多个实例，通过域名，不仅可以透明的对这多个实例进行访问，而且还可以做到负载均衡。但是应用的某个实例随时都有可能故障重启，这时就需要动态的配置域名解析（路由）中的信息。通过etcd的服务发现功能就可以轻松解决这个动态配置的问题。

云平台多实例透明化如下图



使用go做一个简单的服务发现例子

编写master:

```
package etcd_service_discovery

import (
    "fmt"
    "github.com/coreos/etcd/client"
    "golang.org/x/net/context"
    "log"
    "strings"
    "sync"
    "time"
)

var kRoot = "service"

type Master struct {
    sync.RWMutex
    kapi   client.KeysAPI
    key    string
    nodes  map[string]string
```

```
    active bool
}

func NewMaster(serviceName string, endpoints []string) (*Master, error) {
    cfg := client.Config{
        Endpoints:               endpoints,
        HeaderTimeoutPerRequest: time.Second * 2,
    }
    c, err := client.New(cfg)
    if err != nil {
        return nil, err
    }
    master := &Master{
        kapi:   client.NewKeysAPI(c),
        key:    fmt.Sprintf("%s/%s/", kRoot, serviceName),
        nodes:  make(map[string]string),
        active: true,
    }
    master.fetch()

    /// `fetch` Timer may work well too?
    go master.watch()

    return master, err
}

func (m *Master) GetNodesStrictly() map[string]string {
    //log.Println("strictly active ->", m.active)
    if !m.active {
        return nil
    }
    return m.GetNodes()
}

func (m *Master) GetNodes() map[string]string {
    m.RLock()
    defer m.RUnlock()
    return m.nodes
}

func (m *Master) addNode(node, extInfo string) {
    m.Lock()
    defer m.Unlock()
    node = strings.TrimLeft(node, m.key)
    m.nodes[node] = extInfo
}

func (m *Master) delNode(node string) {
    m.Lock()
    defer m.Unlock()
    node = strings.TrimLeft(node, m.key)
    delete(m.nodes, node)
}

func (m *Master) watch() {
    watcher := m.kapi.Watcher(m.key, &client.WatcherOptions{
        Recursive: true,
    })
    for {
        resp, err := watcher.Next(context.Background())
        if err != nil {
```

```
            log.Println(err)
            m.active = false
            continue
        }
        m.active = true
        //log.Println("loop active ->", m.active)
        switch resp.Action {
        case "set", "update":
            m.addNode(resp.Node.Key, resp.Node.Value)
            break
        case "expire", "delete":
            m.delNode(resp.Node.Key)
            break
        default:
            log.Println("watchme!!!", "resp ->", resp)
        }
    }
}

func (m *Master) fetch() error {
    resp, err := m.kapi.Get(context.Background(), m.key, nil)
    if err != nil {
        return err
    }
    if resp.Node.Dir {
        for _, v := range resp.Node.Nodes {
            m.addNode(v.Key, v.Value)
        }
    }
    return err
}
```

Go

编写worker:

```
package etcd_service_discovery

import (
    "fmt"
    "time"

    "github.com/coreos/etcd/client"
    "golang.org/x/net/context"
)

var kHeartBeatInterval = time.Second * 2
var kTTL = time.Second * 5

type Worker struct {
    kapi     client.KeysAPI
    key      string
    extInfo string
    active   bool
    stop     bool
}

func NewWorker(serviceName string, node string, extInfo string, endpoints []string) (*Worker, error) {
    cfg := client.Config{
        Endpoints:               endpoints,
        HeaderTimeoutPerRequest: time.Second * 2,
    }
    c, err := client.New(cfg)
```

```
    if err != nil {
        return nil, err
    }
    worker := &Worker{
        kapi:    client.NewKeysAPI(c),
        key:     fmt.Sprintf("%s/%s/%s", kRoot, serviceName, node),
        extInfo: extInfo,
        active:  false,
        stop:    false,
    }
    return worker, nil
}

func (w *Worker) Register() {
    w.heartbeat()
    go w.heartbeatPeriod()
}

func (w *Worker) Unregister() {
    w.stop = true
    /// no need to wait result
}

func (w *Worker) IsActive() bool {
    return w.active
}

func (w *Worker) IsStop() bool {
    return w.stop
}

func (w *Worker) heartbeatPeriod() {
    for !w.stop {
        w.heartbeat()
        time.Sleep(kHeartBeatInterval)
    }
}

func (w *Worker) heartbeat() error {
    _, err := w.kapi.Set(context.Background(), w.key, w.extInfo, &client.SetOptions{
        TTL: kTTL,
    })
    w.active = err == nil
    return err
}
```

Go

编写测试程序：

master端：

```
package main

import (
    "log"
    "time"

    sd "your_project_path/etcd_service_discovery"
)

func main() {
    m, err := sd.NewMaster("sd-test", []string{
        "http://192.168.139.134:2379",
```

```
        "http://192.168.139.134:2479",
        "http://192.168.139.134:2579",
    })
    if err != nil {
        log.Fatal(err)
    }
    for {
        log.Println("all ->", m.GetNodes())
        log.Println("all(strictly) ->", m.GetNodesStrictly())
        time.Sleep(time.Second * 2)
    }
}
```

Go

worker端:

```
package main

import (
    "flag"
    "fmt"
    sd "your_project_path/etcd_service_discovery"
    "log"
    "time"
)

func main() {
    name := flag.String("name", fmt.Sprintf("%d", time.Now().Unix()), "des")
    extInfo := "lhq-demo..."

    flag.Parse()
    w, err := sd.NewWorker("sd-test", *name, extInfo, []string{
        "http://192.168.139.134:2379",
        "http://192.168.139.134:2479",
        "http://192.168.139.134:2579",
    })
    if err != nil {
        log.Fatal(err)
    }
    w.Register()
    log.Println("name ->", *name, "extInfo ->", extInfo)

    go func() {
        time.Sleep(time.Second * 20)
        w.Unregister()
    }()

    for {
        log.Println("isActive ->", w.IsActive())
        log.Println("isStop ->", w.IsStop())
        time.Sleep(time.Second * 2)
        //服务退出
        if w.IsStop() {
            return
        }
    }
}
```

Go

总结

日常开发集群管理功能中，如果要设计可以动态调整集群大小. 那么首先就要支持服务发现, 就是说当一个新的节点启动时,可以将自己的信息注册给master, 让master把它加入到集群里, 关闭之后也可以把自己从集群中删除。etcd提供了很好的服务注册与发现的基础功，我们采用etcd来做服务发现时，可以把精力用于服务本身的业务处理上。

服务故障恢复

在使用etcd集群的过程中，有时会出现少量主机故障，这时我们需要对集群进行维护。然而，在现实情况下，还可能遇到由于严重的设备 或网络的故障，导致超过半数的节点无法正常工作。

在etcd集群无法提供正常的服务，我们需要用到一些备份和数据恢复的手段。etcd背后的raft，保证了集群的数据的一致性与稳定性。所以我们对etcd的恢复，更多的是恢复etcd的节点服务，并还原用户数据。

首先，从剩余的正常节点中选择一个正常的成员节点， 使用 `etcdctl backup` 命令备份etcd数据。

```
$ ./etcdctl backup --data-dir /var/lib/etcd -backup-dir /tmp/etcd_backup
$ tar -zcxf backup.etcd.tar.gz /tmp/etcd_backup
```

这个命令会将节点中的用户数据全部写入到指定的备份目录中，但是节点ID,集群ID等信息将会丢失， 并在恢复到目的节点时被重新。这样主要是防止原先的节点意外重新加入新的节点集群而导致数据混乱。

然后将Etcd数据恢复到新的集群的任意一个节点上， 使用 `--force-new-cluster` 参数启动Etcd服务。这个参数会重置集群ID和集群的所有成员信息，其中节点的监听地址会被重置为localhost:2379, 表示集群中只有一个节点。

```
$ tar -zxvf backup.etcd.tar.gz -C /var/lib/etcd
$ etcd --data-dir=/var/lib/etcd --force-new-cluster ...
```

启动完成单节点的etcd,可以先对数据的完整性进行验证， 确认无误后再通过Etcd API修改节点的监听地址，让它监听节点的外部IP地址，为增加其他节点做准备。例如：

用etcd命令找到当前节点的ID。

```
$ etcdctl member list

98f0c6bf64240842: name=cd-2 peerURLs=http://127.0.0.1:2580 clientURLs=http://127.0.0.1:2579
```

由于etcdctl不具备修改成员节点参数的功能， 下面的操作要使用API来完成。

```
$ curl http://127.0.0.1:2579/v2/members/98f0c6bf64240842 -XPUT \
 -H "Content-Type:application/json" -d '{"peerURLs":["http://127.0.0.1:2580"]}'
```

注意，在Etcd文档中， 建议首先将集群恢复到一个临时的目录中，从临时目录启动etcd，验证新的数据正确完整后，停止etcd，在将数据恢复到正常的目录中。

最后，在完成第一个成员节点的启动后，可以通过集群扩展的方法使用 `etcdctl member add` 命令添加其他成员节点进来。

Docker部署

最近Coreos发布了Etcd3, 为了尝试Etcd3的新特性，我们可以在测试的服务器搭建一个基于v3的集群，为了让我们的devops更加容器化，下面我举例一下如何通过Docker非常快速地搭建Etcd3的集群。

我使用的etcd的镜像来自 https://quay.io/repository/coreos/etcd ，想搭建其它版本的同学可以选择不同的tag来获取不同的etcd版本。

一、拉取镜像

本例子中，我使用的是3.0.4版本的Etcd

```
$ sudo docker pull quay.io/coreos/etcd:v3.0.4
```

命令执行完毕后，我们看下是否拉取成功

```
$ sudo docker images


REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
quay.io/coreos/etcd   v3.0.4              76436bdf2956        5 days ago          43.31 MB
```

OK， 镜像已经有了，我们就可以基于镜像构建我们的集群容器

二、构建容器

下面是我的一个本地脚本

```
sudo docker run -d -p 4001:4001 -p 2380:2380 -p 2379:2379 --net=host --name etcd0 quay.io/coreos/etcd:v3.0.4
/usr/local/bin/etcd \
 -name etcd0 \
 -advertise-client-urls http://localhost:2379,http://localhost:4001 \
 -listen-client-urls http://localhost:2379,http://localhost:4001 \
 -initial-advertise-peer-urls http://localhost:2380 \
 -listen-peer-urls http://localhost:2380 \
 -initial-cluster-token etcd-cluster-1 \
 -initial-cluster etcd0=http://localhost:2380,etcd1=http://localhost:2480,etcd2=http://localhost:2580

sudo docker run -d -p 4101:4101 -p 2480:2480 -p 2479:2479 --net=host --name etcd1 quay.io/coreos/etcd:v3.0.4
/usr/local/bin/etcd \
 -name etcd1 \
 -advertise-client-urls http://localhost:2479,http://localhost:4101 \
 -listen-client-urls http://localhost:2479,http://localhost:4101 \
 -initial-advertise-peer-urls http://localhost:2480 \
 -listen-peer-urls http://localhost:2480 \
 -initial-cluster-token etcd-cluster-1 \
 -initial-cluster etcd0=http://localhost:2380,etcd1=http://localhost:2480,etcd2=http://localhost:2580

sudo docker run -d -p 4201:4201 -p 2580:2580 -p 2579:2579 --net=host --name etcd2 quay.io/coreos/etcd:v3.0.4
/usr/local/bin/etcd \
 -name etcd2 \
 -advertise-client-urls http://localhost:2579,http://localhost:4201 \
 -listen-client-urls http://localhost:2579,http://localhost:4201 \
 -initial-advertise-peer-urls http://localhost:2580 \
 -listen-peer-urls http://localhost:2580 \
 -initial-cluster-token etcd-cluster-1 \
 -initial-cluster etcd0=http://localhost:2380,etcd1=http://localhost:2480,etcd2=http://localhost:2580
```

分别执行后，我们的三节点集群就可以构建起来了。十分的方便快捷

我们在终端检查下集群是否运行正常：

```
$ curl http://localhost:2379/v2/members

{"members":[{"id":"744924ed0b37b9b1","name":"etcd2","peerURLs":["http://localhost:2580"],"clientURLs":
["http://localhost:2579","http://localhost:4201"]},{"id":"77fb14b13d7590f7","name":"etcd0","peerURLs":
["http://localhost:2380"],"clientURLs":["http://localhost:2379","http://localhost:4001"]},
{"id":"e1110983ed7857fe","name":"etcd1","peerURLs":["http://localhost:2480"],"clientURLs":
["http://localhost:2479","http://localhost:4101"]}]}

$ curl http://localhost:2379/v2/leader

{"leader":"77fb14b13d7590f7","followers":{"744924ed0b37b9b1":{"latency":
{"current":0.001336,"average":0.0032470802752293582,"standardDeviation":0.003517680948114851,"minimum":0.000564,"maxi
{"fail":10,"success":872}},"e1110983ed7857fe":{"latency":
{"current":0.000998,"average":0.0040166746987951715,"standardDeviation":0.00940817295440312,"minimum":0.000553,"maxin
{"fail":0,"success":913}}}}
```

如果，需要做得更灵活，可以把上述的脚本进行改进：

```
# For each machine
ETCD_VERSION=v3.0.0
TOKEN=my-etcd-token
```

```
CLUSTER_STATE=new
NAME_1=etcd-node-0
NAME_2=etcd-node-1
NAME_3=etcd-node-2
HOST_1=192.168.139.140
HOST_2=192.168.139.140
HOST_3=192.168.139.140
CLUSTER=${NAME_1}=http://${HOST_1}:2380,${NAME_2}=http://${HOST_2}:2380,${NAME_3}=http://${HOST_3}:2380

# For node 1
THIS_NAME=${NAME_1}
THIS_IP=${HOST_1}
sudo docker run --net=host --name etcd quay.io/coreos/etcd:${ETCD_VERSION} \
    /usr/local/bin/etcd \
    --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THIS_IP}:2380 \
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP}:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}

# For node 2
THIS_NAME=${NAME_2}
THIS_IP=${HOST_2}
sudo docker run --net=host --name etcd quay.io/coreos/etcd:${ETCD_VERSION} \
    /usr/local/bin/etcd \
    --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THIS_IP}:2380 \
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP}:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}

# For node 3
THIS_NAME=${NAME_3}
THIS_IP=${HOST_3}
sudo docker run --net=host --name etcd quay.io/coreos/etcd:${ETCD_VERSION} \
    /usr/local/bin/etcd \
    --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THIS_IP}:2380 \
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP}:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}
```

以上就是Etcd3在docker的搭建方法，测试集群搭建后，我们就可以基于它调试我们的应用程序的功能了，具体的方式就不在这里列出了