

# 目录

Introduction	1.1
介绍	1.2
资料收集整理	1.2.1
安装	1.3
Linux单机部署	1.3.1
官方文档(翻译)	1.4
开发指南	1.4.1
搭建本地集群	1.4.1.1
和etcd交互	1.4.1.2
API 参考文档	1.4.1.3
API 并发参考文档	1.4.1.4
gRPC 网关	1.4.1.5
gRPC命名与发现	1.4.1.6
试验性的API和特性	1.4.1.7
系统限制	1.4.1.8
操作etcd集群	1.4.2
搭建etcd集群	1.4.2.1
运行时重配置	1.4.2.1.1
运行时重配置的设计	1.4.2.1.2
搭建etcd网关	1.4.2.2
在容器内运行etcd集群	1.4.2.3
配置	1.4.2.4
加密(TODO)	1.4.2.5
维护	1.4.2.6
理解失败	1.4.2.7
灾难恢复	1.4.2.8
性能	1.4.2.9
版本	1.4.2.10
支持平台	1.4.2.11
学习	1.4.3

---

理解数据模型	1.4.3.1
理解API	1.4.3.2
术语	1.4.3.3
API保证	1.4.3.4
API参考文档(翻译)	1.5
KV service	1.5.1
Range方法	1.5.1.1
Put方法	1.5.1.2
DeleteRange方法	1.5.1.3
Txn方法	1.5.1.4
Compact方法	1.5.1.5
Watch service	1.5.2
Watch方法	1.5.2.1
Lease service	1.5.3
LeaseGrant方法	1.5.3.1
LeaseRevoke方法	1.5.3.2
LeaseKeepAlive方法	1.5.3.3
全文标签总览	1.6

---

# Etcd3 学习笔记

Etcd3 是 CoreOS 于 2016年7月发布的 etcd 最新版本。

这份学习笔记, 记录 etcd3 的学习过程和日常使用.

## etcd3 介绍

etcd 是一个分布式一致性键值存储，用于共享配置和服务发现，专注于：

- 简单：良好定义的，面向用户的API (gRPC)
- 安全：带有可选客户端证书认证的自动 TLS
- 快速：测试验证，每秒 10000 写入
- 可靠：使用 Raft 适当分布

etcd 是 Go 编写，并使用 Raft 一致性算法来管理高可用复制日志。

# 资料收集整理

## 网站

- [etcd官网](#)
- [etcd @ github](#) : 首页有大量资料链接
- [etcd @ google group](#)

## 文档

- [etcd3 API](#): 还在TODO中...
- [etcd v3 API](#): RFC

## 工具

- [etcdctl](#): etcd的命令行工具
- [Mirror Maker](#): etcd 的多数据中心支持方案，待研究

## 介绍资料

### etcd3

- [etcd3: A new etcd](#): etcd3发布时官方介绍的blog
- [新一代etcd : etcd3](#): 上文的中文翻译版本，dockone出品，质量不错

### etcd2

- [DockOne技术分享（三十一）：细节 | 谈谈CoreOS的etcd](#)

### Raft

- [Raft一致性算法论文的中文翻译](#)

# 安装

# Linux单机部署

注：参考 [etcd releases 页面](#) 的说明

## 下载

执行下面的命令，下载(大概10M)解压即可，无需安装：

```
curl -L https://github.com/coreos/etcd/releases/download/v3.2.1/etcd-v3.2.1-linux-amd64.tar.gz
tar xzvf etcd-v3.2.1-linux-amd64.tar.gz
mv etcd-v3.2.1-linux-amd64 etcd
cd etcd
./etcd --version

etcd Version: 3.2.1
Git SHA: 61fc123
Go Version: go1.8.3
Go OS/Arch: linux/amd64
```

注：以 etcd-v3.2.1 为例，后续更新版本时可能细节有所不同。

安装目录文件列表如下：

```
$ ls
default.etcd    etcd      README-etcctl.md  READMEv2-etcctl.md
Documentation  etcctl   README.md
```

## 运行

直接运行命令 `./etcd` 就可以启动了，非常简单。

默认使用2379端口为客户端提供通讯，并使用端口2380来进行服务器间通讯。

## 配置

为了方便使用，将 etcd 加入 PATH，另外设置 ETCDCTL\_API 为3(后面解释)。

在 `/etc/profile` 中加入以下内容：

```
# etcd
export PATH=/home/sky/work/soft/etcd:$PATH
export ETCDCTL_API=3
```

然后执行 `source /etc/profile` 重新加载。

## 客户端访问

### 配置etcdctl

etcdctl 是 etcd 的客户端命令行。

特别提醒：使用 etcdctl 前，务必设置环境变量 `ETCDCTL_API=3`！

注意：如果不设置 `ETCDCTL_API=3`，则默认是的API版本是2：

```
$ etcdctl version
etcdctl version: 3.2.1
API version: 2
```

正确设置后，API版本变成3：

```
$ etcdctl version
etcdctl version: 3.2.1
API version: 3.2
```

### 使用etcdctl

通过下面的put和get命令来验证连接并操作etcd：

```
$ ./etcdctl put aaa 1
OK
$ ./etcdctl get aaa
aaa
1
```

## 总结

上面操作完成之后，就有一个可运行的简单 etcd 服务器和一个可用的 etcdctl 客户端。



# 官方文档

注：内容翻译自 <https://github.com/coreos/etcd/blob/master/Documentation/docs.md>

etcd 是一个分布式键值对存储，设计用来可靠而快速的保存关键数据并提供访问。通过分布式锁，leader选举和写屏障(write barriers)来开启可靠的分布式协同。etcd集群是为高可用，持久性数据存储和检索而准备。

## 开始

现在etcd的用户和开发者可以从[下载并构建 etcd](#)开始。在获取etcd之后，跟随[quick demo](#)来看构建和操作etcd集群的基本内容。

## 使用etcd开发

开始使用etcd作为分布式键值对存储的最简单的方式是[搭建本地集群](#)

- [搭建本地集群](#)
- 和 [etcd 交互](#)
- [gRPC etcd core 和 etcd concurrency API 参考](#)
- [经过gRPC 网关的 HTTP JSON API](#)
- [gRPC命名与发现](#)
- [客户端 和 代理 命名空间](#)
- [内嵌的 etcd](#)
- [试验性的 API 和特性](#)
- [系统限制](#)

## 操作 etcd 集群

管理员，需要为支持的开发人员创建可靠而可扩展的键值存储，应该从[多机集群 开始](#).

- [搭建etcd集群](#)
- [搭建etcd网关](#)
- [在容器内运行etcd集群](#)
- [配置](#)
- [加密\(TODO\)](#)
- [Monitoring](#)

- 维护
- 理解失败
- 灾难恢复
- 性能
- 版本
- 支持平台

## 学习

要学习更多 etcd 背后的概念和内部细节，请阅读下面的内容：

- Why etcd (TODO)
- 理解数据模型
- 理解API
- 术语
- API保证
- Internals (TODO)

## 升级和兼容性

- Migrate applications from using API v2 to API v3
- Updating v2.3 to v3.0

注：因为是直接从etcd3开始，所以这两节关于升级的内容不关心，就不翻译了 :)

# 开发指南

## 内容

开发指南的内容如下：

- 搭建本地集群
- 和 etcd 交互
- gRPC etcd core 和 etcd concurrency API 参考
- 经过gRPC 网关的 HTTP JSON API
- gRPC命名与发现
- 客户端 和 代理 命名空间
- 内嵌的 etcd
- 试验性的 API 和特性
- 系统限制

注：内容来自 <https://github.com/coreos/etcd/tree/master/Documentation/dev-guide>

# 搭建本地集群

对于测试和开发部署，最快最简单的方式是搭建本地集群。对于产品部署，参考 [集群](#) 章节。

## 本地单独集群

注： `单独集群` 指只有一台服务器的集群。

部署 `etcd` 集群作为单独集群是直截了当的。仅用一个命令启动它：

```
$ ./etcd  
...
```

启动的 `etcd` 成员在 `localhost:2379` 监听客户端请求。

通过使用 `etcdctl` 来和已经启动的集群交互：

```
# 使用 API 版本 3  
$ export ETCDCTL_API=3  
  
$ ./etcdctl put foo bar  
OK  
  
$ ./etcdctl get foo  
bar
```

## 本地多成员集群

注： `多成员集群` 指有多台服务器的集群。

提供 `Procfile` 用于简化搭建本地多成员集群。通过少量命令来启动多成员集群：

```
# install goreman program to control Profile-based applications.  
$ go get github.com/mattm/goreman  
$ goreman -f Procfile start  
...
```

注1：必须先安装 `go`，请见章节 [Go语言安装](#) 注2：这里所说的 `Procfile` 文件是来自 `etcd` 的 [gitub](#) 项目的根目录下的 `Procfile` 文件，但是需要修改一下，将里面的 `bin/etcd` 修改为 `etcd`

启动的成员各自在 `localhost:12379` , `localhost:22379` , 和 `localhost:32379` 上监听客户端请求。

注：英文原文中是 `localhost:12379` 用的是 12379 端口，但是实际上述 `Procfile` 文件中启动的是 2379 端口，如果连接时发现无法访问，请自行修改。下面的 12379 也是如此，请自行修改为 2379.

通过使用 `etcdctl` 来和已经启动的集群交互：

```
# 使用 API 版本 3
$ export ETCDCTL_API=3

$ etcdctl --write-out=table --endpoints=localhost:12379 member list
+-----+-----+-----+-----+
---+
|      ID      | STATUS | NAME   |      PEER ADDRS      |      CLIENT ADDRS
|-----+-----+-----+-----+
---+
| 8211f1d0f64f3269 | started | infra1 | http://127.0.0.1:12380 | http://127.0.0.1:1237
9 |
| 91bc3c398fb3c146 | started | infra2 | http://127.0.0.1:22380 | http://127.0.0.1:2237
9 |
| fd422379fda50e48 | started | infra3 | http://127.0.0.1:32380 | http://127.0.0.1:3237
9 |
+-----+-----+-----+-----+
---+

$ etcdctl --endpoints=localhost:12379 put foo bar
OK
```

为了体验etcd的容错性，杀掉一个成员：

```
# 杀掉 etcd2
$ goreman run stop etcd2
# 注：实测这个命令无法停止etcd，最后还是用ps命令找出pid，然后kill
# ps -ef | grep etcd | grep 127.0.0.1:22379
# kill ****

$ etcdctl --endpoints=localhost:12379 put key hello
OK

$ etcdctl --endpoints=localhost:12379 get key
hello

# 试图从被杀掉的成员获取key
$ etcdctl --endpoints=localhost:22379 get key
2016/04/18 23:07:35 grpc: Conn.resetTransport failed to create client transport: connection error: desc = "transport: dial tcp 127.0.0.1:22379: getsockopt: connection refused"; Reconnecting to "localhost:22379"
Error: grpc: timed out trying to connect

# 重启被杀掉的成员
# 注：实测这个restart命令可用
$ goreman run restart etcd2

# 从重启的成员获取key
$ etcdctl --endpoints=localhost:22379 get key
hello
```

要学习更多和etcd的交互，请阅读 [和etcd交互](#)

# 和 etcd 交互

用户通常通过设置或者获取 key 的值来和 etcd 交互。这一节描述如何使用 etcdctl 来操作，etcdctl 是一个和 etcd 服务器交互的命令行工具。这里描述的概念也适用于 gRPC API 或者客户端类库 API。

默认，为了向后兼容 etcdctl 使用 v2 API 来和 etcd 服务器通讯。为了让 etcdctl 使用 v3 API 来和 etcd 通讯，API 版本必须通过环境变量 `ETCDCTL_API` 设置为版本3。

```
export ETCDCTL_API=3
```

## 写入 key

应用通过写入 key 来储存 key 到 etcd 中。每个存储的 key 被通过 Raft 协议复制到所有 etcd 集群成员来达到一致性和可靠性。

这是设置 key `foo` 的值为 `bar` 的命令：

```
$ etcdctl put foo bar
OK
```

## 读取 key

应用可以从 etcd 集群中读取 key 的值。查询可以读取单个 key，或者某个范围的 key。

假设 etcd 集群存储有下面的 key：

```
foo = bar
foo1 = bar1
foo3 = bar3
```

这是读取 key `foo` 的值的命令：

```
$ etcdctl get foo
foo
bar
```

这是覆盖从 `foo` to `foo9` 的 key 的命令：

```
$ etcdctl get foo foo9
foo
bar
foo1
bar1
foo3
bar3
```

注：测试中发现，这个命令的有效区间是这样：`[foo foo9]`，即包含第一个参数，但是不包含第二个参数。因此如果第二个参数是`foo3`，上面的命令是不会返回key`foo3`的值的。具体可以参考API说明。

## 读取 key 过往版本的值

应用可能想读取 key 的被替代的值。例如，应用可能想通过访问 key 的过往版本来回滚到旧的配置。或者，应用可能想通过访问 key 历史记录的多个请求来得到一个覆盖多个 key 上的统一视图。

因为 etcd 集群上键值存储的每个修改都会增加 etcd 集群的全局修订版本，应用可以通过提供旧有的 etcd 版本来读取被替代的 key。

假设 etcd 集群已经有下列 key：

```
$ etcdctl put foo bar          # revision = 2
$ etcdctl put foo1 bar1        # revision = 3
$ etcdctl put foo bar_new      # revision = 4
$ etcdctl put foo1 bar1_new    # revision = 5
```

这里是访问 key 的过往版本的例子：

```
$ etcdctl get foo foo9 # 访问 key 的最新版本
foo
bar_new
foo1
bar1_new

$ etcdctl get --rev=4 foo foo9 # 访问 key 的修订版本4
foo
bar_new
foo1
bar1

$ etcdctl get --rev=3 foo foo9 # 访问 key 的修订版本3
foo
bar
foo1
bar1

$ etcdctl get --rev=2 foo foo9 # 访问 key 的修订版本2
foo
bar

$ etcdctl get --rev=1 foo foo9 # 访问 key 的修订版本1
```

## 删除 key

应用可以从 etcd 集群中删除一个 key 或者特定范围的 key。

下面是删除 key `foo` 的命令：

```
$ etcdctl del foo
1 # 删除了一个key
```

这是删除从 `foo` to `foo9` 范围的 key 的命令：

```
$ etcdctl del foo foo9
2 # 删除了两个key
```

## 观察 key 的变化

应用可以观察一个 key 或者特定范围内的 key 来监控任何更新。

这是在 key `foo` 上进行观察的命令：

```
$ etcdctl watch foo
# 在另外一个终端: etcdctl put foo bar
foo
bar
```

这是观察从 `foo` to `foo9` 范围key的命令：

```
$ etcdctl watch foo foo9
# 在另外一个终端: etcdctl put foo bar
foo
bar
# 在另外一个终端: etcdctl put foo1 bar1
foo1
bar1
```

## 观察 key 的历史改动

应用可能想观察 etcd 中 key 的历史改动。例如，应用想接收到某个 key 的所有修改。如果应用一直连接到etcd，那么 `watch` 就足够好了。但是，如果应用或者 etcd 出错，改动可能发生在出错期间，这样应用就没能实时接收到这个更新。为了保证更新被接收，应用必须能够观察到 key 的历史变动。为了做到这点，应用可以在观察时指定一个历史修订版本，就像读取 key 的过往版本一样。

假设我们完成了下列操作序列：

```
etcdctl put foo bar      # revision = 2
etcdctl put foo1 bar1    # revision = 3
etcdctl put foo bar_new  # revision = 4
etcdctl put foo1 bar1_new # revision = 5
```

这是观察历史改动的例子：

```
# 从修订版本 2 开始观察key `foo` 的改动
$ etcdctl watch --rev=2 foo
PUT
foo
bar
PUT
foo
bar_new

# 从修订版本 3 开始观察key `foo` 的改动
$ etcdctl watch --rev=3 foo
PUT
foo
bar_new
```

## 压缩修订版本

如我们提到的，etcd 保存修订版本以便应用可以读取 key 的过往版本。但是，为了避免积累无限数量的历史数据，压缩过往的修订版本就变得很重要。压缩之后，etcd 删除历史修订版本，释放资源来提供未来使用。所有修订版本在压缩修订版本之前的被替代的数据将不可访问。

这是压缩修订版本的命令：

```
$ etcdctl compact 5
compacted revision 5

# 在压缩修订版本之前的任何修订版本都不可访问
$ etcdctl get --rev=4 foo
Error: rpc error: code = 11 desc = etcdserver: mvcc: required revision has been compacted
```

## 授予租约

应用可以为 etcd 集群里面的 key 授予租约。当 key 被附加到租约时，它的生存时间被绑定到租约的生存时间，而租约的生存时间相应的被 time-to-live (TTL) 管理。租约的实际 TTL 值是不低于最小 TTL，由 etcd 集群选择。一旦租约的 TTL 到期，租约就过期并且所有附带的 key 都将被删除。

这是授予租约的命令：

```
# 授予租约，TTL为10秒
$ etcdctl lease grant 10
lease 32695410dcc0ca06 granted with TTL(10s)

# 附加key foo到租约32695410dcc0ca06
$ etcdctl put --lease=32695410dcc0ca06 foo bar
OK
```

## 撤销租约

应用通过租约 id 可以撤销租约。撤销租约将删除所有它附带的 key。

假设我们完成了下列的操作：

```
$ etcdctl lease grant 10
lease 32695410dcc0ca06 granted with TTL(10s)
$ etcdctl put --lease=32695410dcc0ca06 foo bar
OK
```

这是撤销同一个租约的命令：

```
$ etcdctl lease revoke 32695410dcc0ca06
lease 32695410dcc0ca06 revoked

$ etcdctl get foo
# 空应答，因为租约撤销导致foo被删除
```

## 维持租约

应用可以通过刷新 key 的 TTL 来维持租约，以便租约不过期。

假设我们完成了下列操作：

```
$ etcdctl lease grant 10
lease 32695410dcc0ca06 granted with TTL(10s)
```

这是维持同一个租约的命令：

```
$ etcdctl lease keep-alive 32695410dcc0ca0
lease 32695410dcc0ca0 keepalive with TTL(100)
lease 32695410dcc0ca0 keepalive with TTL(100)
lease 32695410dcc0ca0 keepalive with TTL(100)
...
...
```

注：上面的这个命令中，etcdctl 不是单次续约，而是 etcdctl 会一直不断的发送请求来维持这个租约。

## etcd API 参考

### 特别说明

这个参考文档内容实在太多，英文原文的文档是一个巨型 markdown 文件，所有内容堆在一起，长的离谱……

为了解决阅读困难的问题，我采用按照服务和方法拆分的方式，重新整理了一份 [API 参考文档](#) 并翻译，内容和原文一致，只是拆分成多节。

推荐阅读拆分的版本，请用力戳下面的链接：

[API 参考文档](#)

# etcd concurrency API Reference

This is a generated documentation. Please read the proto files for more.

## service Lock (etcdserver/api/v3lock/v3lockpb/v3lock.proto)

The lock service exposes client-side locking facilities as a gRPC interface.

Method	Request Type	Response Type	Description
Lock	LockRequest	LockResponse	Lock acquires a distributed shared lock on a given named lock. On success, it will return a unique key that exists so long as the lock is held by the caller. This key can be used in conjunction with transactions to safely ensure updates to etcd only occur while holding lock ownership. The lock is held until Unlock is called on the key or the lease associate with the owner expires.
Unlock	UnlockRequest	UnlockResponse	Unlock takes a key returned by Lock and releases the hold on lock. The next Lock caller waiting for the lock will then be woken up and given ownership of the lock.

## message LockRequest (etcdserver/api/v3lock/v3lockpb/v3lock.proto)

Field	Description	Type
name	name is the identifier for the distributed shared lock to be acquired.	bytes
lease	lease is the ID of the lease that will be attached to ownership of the lock. If the lease expires or is revoked and currently holds the lock, the lock is automatically released. Calls to Lock with the same lease will be treated as a single acquisition; locking twice with the same lease is a no-op.	int64

## message LockResponse (etcdserver/api/v3lock/v3lockpb/v3lock.proto)

Field	Description	Type
header		etcdserverpb.ResponseHeader
key	key is a key that will exist on etcd for the duration that the Lock caller owns the lock. Users should not modify this key or the lock may exhibit undefined behavior.	bytes

**message unlockRequest** ([etcdserver/api/v3lock/v3lockpb/v3lock.proto](#))

Field	Description	Type
key	key is the lock ownership key granted by Lock.	bytes

**message unlockResponse** ([etcdserver/api/v3lock/v3lockpb/v3lock.proto](#))

Field	Description	Type
header		etcdserverpb.ResponseHeader

**service Election** ([etcdserver/api/v3election/v3electionpb/v3election.proto](#))

The election service exposes client-side election facilities as a gRPC interface.

Method	Request Type	Response Type	Description
Campaign	CampaignRequest	CampaignResponse	Campaign waits to acquire leadership in an election, returning a LeaderKey representing the leadership if successful. The LeaderKey can then be used to issue new values on the election, transactionally guard API requests on leadership still being held, and resign from the election.
Proclaim	ProclaimRequest	ProclaimResponse	Proclaim updates the leader's posted value with a new value.
Leader	LeaderRequest	LeaderResponse	Leader returns the current election proclamation, if any.
Observe	LeaderRequest	LeaderResponse	Observe streams election proclamations in-order as made by the election's elected leaders.
Resign	ResignRequest	ResignResponse	Resign releases election leadership so other campaigners may acquire leadership on the election.

**message campaignRequest (etcdserver/api/v3election/v3selectionpb/v3selection.proto)**

Field	Description	Type
name	name is the election's identifier for the campaign.	bytes
lease	lease is the ID of the lease attached to leadership of the election. If the lease expires or is revoked before resigning leadership, then the leadership is transferred to the next campaigner, if any.	int64
value	value is the initial proclaimed value set when the campaigner wins the election.	bytes

**message campaignResponse (etcdserver/api/v3election/v3selectionpb/v3selection.proto)**

Field	Description	Type
header		etcdserverpb.ResponseHeader
leader	leader describes the resources used for holding leadership of the election.	LeaderKey

**message LeaderKey (etcdserver/api/v3election/v3electionpb/v3election.proto)**

Field	Description	Type
name	name is the election identifier that corresponds to the leadership key.	bytes
key	key is an opaque key representing the ownership of the election. If the key is deleted, then leadership is lost.	bytes
rev	rev is the creation revision of the key. It can be used to test for ownership of an election during transactions by testing the key's creation revision matches rev.	int64
lease	lease is the lease ID of the election leader.	int64

**message LeaderRequest (etcdserver/api/v3election/v3electionpb/v3election.proto)**

Field	Description	Type
name	name is the election identifier for the leadership information.	bytes

**message LeaderResponse (etcdserver/api/v3election/v3electionpb/v3election.proto)**

Field	Description	Type
header		etcdserverpb.ResponseHeader
kv	kv is the key-value pair representing the latest leader update.	mvccpb.KeyValue

**message ProclaimRequest (etcdserver/api/v3election/v3electionpb/v3election.proto)**

Field	Description	Type
leader	leader is the leadership hold on the election.	LeaderKey
value	value is an update meant to overwrite the leader's current value.	bytes

**message ProclaimResponse (etcdserver/api/v3election/v3electionpb/v3election.proto)**

Field	Description	Type
header		etcdserverpb.ResponseHeader

**message ResignRequest (etcdserver/api/v3election/v3electionpb/v3election.proto)**

Field	Description	Type
leader	leader is the leadership to relinquish by resignation.	LeaderKey

**message ResignResponse (etcdserver/api/v3election/v3electionpb/v3election.proto)**

Field	Description	Type
header		etcdserverpb.ResponseHeader

**message Event (mvcc/mvccpb/kv.proto)**

Field	Description	Type
type	type is the kind of event. If type is a PUT, it indicates new data has been stored to the key. If type is a DELETE, it indicates the key was deleted.	EventType
kv	kv holds the KeyValue for the event. A PUT event contains current kv pair. A PUT event with kv.Version=1 indicates the creation of a key. A DELETE/EXPIRE event contains the deleted key with its modification revision set to the revision of deletion.	KeyValue
prev_kv	prev_kv holds the key-value pair before the event happens.	KeyValue

**message KeyValue (mvcc/mvccpb/kv.proto)**

Field	Description	Type
key	key is the key in bytes. An empty key is not allowed.	bytes
create_revision	create_revision is the revision of last creation on this key.	int64
mod_revision	mod_revision is the revision of last modification on this key.	int64
version	version is the version of the key. A deletion resets the version to zero and any modification of the key increases its version.	int64
value	value is the value held by the key, in bytes.	bytes
lease	lease is the ID of the lease that attached to key. When the attached lease expires, the key will be deleted. If lease is 0, then no lease is attached to the key.	int64

# gRPC 网关

## 为什么用 **grpc-gateway**

etcd v3 使用 gRPC 作为它的消息协议。etcd 项目包括基于 gRPC 的 Go client 和命令行工具 etcdctl，通过 gRPC 和 etcd 集群通讯。对于不支持 gRPC 支持的语言，etcd 提供 JSON 的 [grpc-gateway](#)。这个网关提供 RESTful 代理，翻译 HTTP/JSON 请求为 gRPC 消息。

## 使用 **grpc-gateway**

网关接受 etcd 的 [protocol buffer](#) 的 [JSON mapping](#)。注意 `key` 和 `value` 字段被定义为 byte 数组，因此必须在 JSON 中以 base64 编码。

```
< < COMMENT
https://www.base64encode.org/
foo is 'Zm9v' in Base64
bar is 'YmFy'
COMMENT

curl -L http://localhost:2379/v3alpha/kv/put \
-X POST -d '{"key": "Zm9v", "value": "YmFy"}'

curl -L http://localhost:2379/v3alpha/kv/range \
-X POST -d '{"key": "Zm9v"}'
```

## Swagger

生成的 [Swagger](#) API 定义可以在 [rpc.swagger.json](#) 找到。

# gRPC naming and discovery

etcd provides a gRPC resolver to support an alternative name system that fetches endpoints from etcd for discovering gRPC services. The underlying mechanism is based on watching updates to keys prefixed with the service name.

## Using etcd discovery with go-grpc

The etcd client provides a gRPC resolver for resolving gRPC endpoints with an etcd backend. The resolver is initialized with an etcd client and given a target for resolution:

```
import (
    "github.com/coreos/etcd/clientv3"
    etcdnaming "github.com/coreos/etcd/clientv3/naming"

    "google.golang.org/grpc"
)

...

cli, cerr := clientv3.NewFromURL("http://localhost:2379")
r := &etcdnaming.GRPCResolver{Client: cli}
b := grpc.RoundRobin(r)
conn, gerr := grpc.Dial("my-service", grpc.WithBalancer(b))
```

## Managing service endpoints

The etcd resolver treats all keys under the prefix of the resolution target following a "/" (e.g., "my-service/") with JSON-encoded go-grpc `naming.Update` values as potential service endpoints. Endpoints are added to the service by creating new keys and removed from the service by deleting keys.

### Adding an endpoint

New endpoints can be added to the service through `etcdctl`:

```
ETCDCTL_API=3 etcdctl put my-service/1.2.3.4 '{"Addr":"1.2.3.4", "Metadata":"..."}'
```

The etcd client's `GRPCResolver.Update` method can also register new endpoints with a key matching the `Addr`:

```
r.Update(context.TODO(), "my-service", naming.Update{Op: naming.Add, Addr: "1.2.3.4",  
Metadata: "..."})
```

## Deleting an endpoint

Hosts can be deleted from the service through `etcdctl`:

```
ETCDCTL_API=3 etcdctl del my-service/1.2.3.4
```

The etcd client's `GRPCResolver.Update` method also supports deleting endpoints:

```
r.Update(context.TODO(), "my-service", naming.Update{Op: naming.Delete, Addr: "1.2.3.4"})
```

## Registering an endpoint with a lease

Registering an endpoint with a lease ensures that if the host can't maintain a keepalive heartbeat (e.g., its machine fails), it will be removed from the service:

```
lease=`ETCDCTL_API=3 etcdctl lease grant 5 | cut -f2 -d' '`  
ETCDCTL_API=3 etcdctl put --lease=$lease my-service/1.2.3.4 '{"Addr":"1.2.3.4", "Metada  
ta":"..."}'  
ETCDCTL_API=3 etcdctl lease keep-alive $lease
```

## 试验性的**API**和特性

`etcd` 项目的大部分是稳定的，但是我们依然在快速前行！我们信仰快速发布的哲学。我们想得到还在开发和完善中的特性的早期反馈。因此，有，而且将会有更多，试验性的特性和 API。我们计划基于社区的早期反馈来在后面几个发布中改进这些特性，或者如果没有人在有兴趣就中止他们。如果你正在运行一个产品系统，请不要依赖任何试验性的特性或者API。

当前试验性的 **API/特性** 是：

(暂无)

# System limits

## Request size limit

etcd is designed to handle small key value pairs typical for metadata. Larger requests will work, but may increase the latency of other requests. For the time being, etcd guarantees to support RPC requests with up to 1MB of data. In the future, the size limit may be loosened or made configurable.

## Storage size limit

The default storage size limit is 2GB, configurable with `--quota-backend-bytes` flag; supports up to 8GB.

# 操作 etcd 集群

## 内容

- 搭建 etcd 集群
- 搭建 etcd 网关
- 在容器内运行 etcd 集群
- 配置
- 加密 (TODO)
- Monitoring
- 维护
- 理解失败
- 灾难恢复
- 性能
- 版本
- 支持平台

## 额外说明

1. 这些内容相当于是 etcd3 的管理手册，针对的是 etcd3 的管理者(如运维人员)。对于运维人员，需要重点阅读这些内容。
2. 内容来自 github 官网，地址：  
<https://github.com/coreos/etcd/tree/master/Documentation/op-guide>
3. 这些文档打包在 etcd3 的发行包中(Documentent / op-guide 目录)，平时使用时可以随时阅读。

# 集群指南

注：内容翻译自 [Clustering Guide](#)

## 概述

启动 etcd 集群要求每个成员知道集群中的其他成员。在一些场景中，集群成员的 IP 地址可能无法提前知道。在这种情况下，etcd 集群可以在发现服务的帮助下启动。

一旦 etcd 集群启动并运行，通过 [运行时重配置](#) 来添加或者移除成员。为了更好的理解运行时重配置背后的设计，建议阅读 [运行时重配置的设计](#)。

这份指南将覆盖下列用于启动 etcd 集群的机制：

- [Static / 静态](#)
- [etcd Discovery / etcd 发现](#)
- [DNS Discovery / DNS 发现](#)

启动机制的每一种都将用于启动三台机器的 etcd 集群，详情如下：

名字	地址	主机
infra0	10.0.1.10	infra0.example.com
infra1	10.0.1.11	infra1.example.com
infra2	10.0.1.12	infra2.example.com

## 静态

如果在启动前我们知道集群成员，他们的地址和集群的大小，我们可以使用通过设置 `initial-cluster` 标记来离线启动配置。每个机器将得到下列环境变量或者命令行：

```
ETCD_INITIAL_CLUSTER="infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=http://10.0.1.12:2380"
ETCD_INITIAL_CLUSTER_STATE=new
```

```
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=http://10.0.1.12:2380 \
--initial-cluster-state new
```

注意：在 `initial-cluster` 中指定的 URL 是 *advertised peer URLs*，例如，他们将匹配对应节点的 `initial-advertise-peer-urls` 的值。

如果使用同样的配置启动多个集群(或者创建并部署单个集群)用于测试目的，强烈推荐每个集群给予一个唯一的 `initial-cluster-token`。这样做之后，etcd 可以为集群生成唯一的集群 ID 和成员 ID，甚至他们有完全一样的配置。这可以将 etcd 从可能让集群孵化的跨集群交互中保护起来。

etcd 在 `listen-client-urls` 上接收客户端访问。etcd 成员将 `advertise-client-urls` 指定的 URI 上通告给其他成员，代理和客户端。常见的错误是设置 `advertise-client-urls` 为 `localhost` 或者留空为默认值，如果远程客户端可以达到 etcd。

在每台机器上，使用这些标记启动 etcd：

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
--listen-peer-urls http://10.0.1.10:2380 \
--listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.10:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=h
ttp://10.0.1.12:2380 \
--initial-cluster-state new
```

```
$ etcd --name infra1 --initial-advertise-peer-urls http://10.0.1.11:2380 \
--listen-peer-urls http://10.0.1.11:2380 \
--listen-client-urls http://10.0.1.11:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.11:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=h
ttp://10.0.1.12:2380 \
--initial-cluster-state new
```

```
$ etcd --name infra2 --initial-advertise-peer-urls http://10.0.1.12:2380 \
--listen-peer-urls http://10.0.1.12:2380 \
--listen-client-urls http://10.0.1.12:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.12:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=h
ttp://10.0.1.12:2380 \
--initial-cluster-state new
```

以 `--initial-cluster` 开头的命令行参数将在 etcd 随后的运行中被忽略。可以在初始化启动进程之后随意的删除环境变量或者命令行标记。如果配置需要稍后修改(例如，添加成员到集群或者从集群中移除成员)，查看 [运行时配置指南](#)。

## TLS

etcd 支持通过 TLS 协议的加密通讯。TLS 通道可以用于加密伙伴间的内部集群通讯，也可以用于加密客户端请求。这个章节提供例子来搭建使用伙伴和客户端 TLS 的集群。详细描述 etcd 的 TLS 支持的额外信息可以在 [加密指南](#)

### 自签名证书

使用自签名证书证书(self-signed certificates)的集群同事加密请求并认证它的连接。要启动使用自签名证书的集群，每个集群成员应该有一个唯一的通过共享的集群CA证书来签名的键对(key pair) ( `member.crt` , `member.key` )，用于伙伴连接和客户端连接。证书可以通过仿照 etcd [搭建TLS](#) 的例子来生成。

在每台机器上，etcd可以使用这些标记来启动：

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
--listen-peer-urls https://10.0.1.10:2380 \
--listen-client-urls https://10.0.1.10:2379,https://127.0.0.1:2379 \
--advertise-client-urls https://10.0.1.10:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2
=https://10.0.1.12:2380 \
--initial-cluster-state new \
--client-cert-auth --trusted-ca-file=/path/to/ca-client.crt \
--cert-file=/path/to/infra0-client.crt --key-file=/path/to/infra0-client.key \
--peer-client-cert-auth --peer-trusted-ca-file=ca-peer.crt \
--peer-cert-file=/path/to/infra0-peer.crt --peer-key-file=/path/to/infra0-peer.key
```

```
$ etcd --name infra1 --initial-advertise-peer-urls https://10.0.1.11:2380 \
--listen-peer-urls https://10.0.1.11:2380 \
--listen-client-urls https://10.0.1.11:2379,https://127.0.0.1:2379 \
--advertise-client-urls https://10.0.1.11:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2
=https://10.0.1.12:2380 \
--initial-cluster-state new \
--client-cert-auth --trusted-ca-file=/path/to/ca-client.crt \
--cert-file=/path/to/infra1-client.crt --key-file=/path/to/infra1-client.key \
--peer-client-cert-auth --peer-trusted-ca-file=ca-peer.crt \
--peer-cert-file=/path/to/infra1-peer.crt --peer-key-file=/path/to/infra1-peer.key
```

```
$ etcd --name infra2 --initial-advertise-peer-urls https://10.0.1.12:2380 \
--listen-peer-urls https://10.0.1.12:2380 \
--listen-client-urls https://10.0.1.12:2379,https://127.0.0.1:2379 \
--advertise-client-urls https://10.0.1.12:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2 \
=https://10.0.1.12:2380 \
--initial-cluster-state new \
--client-cert-auth --trusted-ca-file=/path/to/ca-client.crt \
--cert-file=/path/to/infra2-client.crt --key-file=/path/to/infra2-client.key \
--peer-client-cert-auth --peer-trusted-ca-file=ca-peer.crt \
--peer-cert-file=/path/to/infra2-peer.crt --peer-key-file=/path/to/infra2-peer.key
```

## 自动证书

如果集群需要加密通讯，但是不需要认证连接，etcd 可以配置为自动生成 key。在初始化时，每个集群基于它的通告IP(advertised IP) 地址和主机名创建它自己的 key 集合。

在每台机器上，etcd 使用这些标记启动：

```
$ etcd --name infra0 --initial-advertise-peer-urls https://10.0.1.10:2380 \
--listen-peer-urls https://10.0.1.10:2380 \
--listen-client-urls https://10.0.1.10:2379,https://127.0.0.1:2379 \
--advertise-client-urls https://10.0.1.10:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2 \
=https://10.0.1.12:2380 \
--initial-cluster-state new \
--auto-tls \
--peer-auto-tls
```

```
$ etcd --name infra1 --initial-advertise-peer-urls https://10.0.1.11:2380 \
--listen-peer-urls https://10.0.1.11:2380 \
--listen-client-urls https://10.0.1.11:2379,https://127.0.0.1:2379 \
--advertise-client-urls https://10.0.1.11:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2 \
=https://10.0.1.12:2380 \
--initial-cluster-state new \
--auto-tls \
--peer-auto-tls
```

```
$ etcd --name infra2 --initial-advertise-peer-urls https://10.0.1.12:2380 \
--listen-peer-urls https://10.0.1.12:2380 \
--listen-client-urls https://10.0.1.12:2379,https://127.0.0.1:2379 \
--advertise-client-urls https://10.0.1.12:2379 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2\
=https://10.0.1.12:2380 \
--initial-cluster-state new \
--auto-tls \
--peer-auto-tls
```

## 错误案例

### 案例1

在下面的例子中，我们没有在列举的节点列表中包含我们新的主机地址。如果这是一个新的集群，节点必须添加到初始化集群成员的列表中。

```
$ etcd --name infra1 --initial-advertise-peer-urls http://10.0.1.11:2380 \
--listen-peer-urls https://10.0.1.11:2380 \
--listen-client-urls http://10.0.1.11:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.11:2379 \
--initial-cluster infra0=http://10.0.1.10:2380 \
--initial-cluster-state new
etcd: infra1 not listed in the initial cluster config
exit 1
```

### 案例2

在这个例子中，我们试图映射节点(infra0)在不同地址(127.0.0.1:2380)而不是它在集群列表(10.0.1.10:2380)中列举的地址。如果这个节点是监听多个地址，所有地址必须在 "initial-cluster" 配置指令中反映出来。

```
$ etcd --name infra0 --initial-advertise-peer-urls http://127.0.0.1:2380 \
--listen-peer-urls http://10.0.1.10:2380 \
--listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.10:2379 \
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=h
ttp://10.0.1.12:2380 \
--initial-cluster-state=new
etcd: error setting up initial cluster: infra0 has different advertised URLs in the cl
uster and advertised peer URLs list
exit 1
```

## 案例3

如果伙伴被用配置参数的不同集合配置并试图加入这个集群，etcd 将报告集群 ID 不匹配并退出。

```
$ etcd --name infra3 --initial-advertise-peer-urls http://10.0.1.13:2380 \
--listen-peer-urls http://10.0.1.13:2380 \
--listen-client-urls http://10.0.1.13:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.13:2379 \
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra3=h
ttp://10.0.1.13:2380 \
--initial-cluster-state=new
etcd: conflicting cluster ID to the target cluster (c6ab534d07e8fcc4 != bc25ea2a74fb18
b0). Exiting.
exit 1
```

## 发现

在一些案例中，集群伙伴的 IP 可能无法提前知道。当使用云提供商或者网络使用 DHCP 时比较常见。在这些情况下，相比指定静态配置，使用使用已经存在的 etcd 集群来启动一个新的。我们称这个过程为"发现"。

有两个方法可以用来做发现：

- etcd 发现服务
- DNS SRV 记录

### etcd 发现

为了更好的理解发现服务协议的设计，建议阅读发现服务项目 [文档](#)。

### 发现 URL 的存活时间

发现 URL 标识唯一的 etcd 集群。对于新的集群，总是创建发现 URL 而不是重用发现 URL。

此外，发现URL应该仅仅用于集群的初始化启动。在集群已经运行之后修改集群成员，阅读 [运行时重配置 指南](#)。

### 定制 etcd 发现服务

发现使用已有集群来启动自身。如果使用私有的 etcd 集群，可以创建像这样的 URL：

```
$ curl -X PUT https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83/_config/size -d value=3
```

通过设置 URL 的 size，创建了带有期待集群大小为3的发现 URL。

用于这个场景的 URL 将是

`https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83` 而 etcd 成员将使用

`https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83` 目录来注册，当他们启动时。

每个成员必须有指定不同的名字标记。 `Hostname` 或者 `machine-id` 是个好选择。否则发现会因为重复名字而失败

现在我们用这些用于每个成员的相关标记启动 etcd：

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
--listen-peer-urls http://10.0.1.10:2380 \
--listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.10:2379 \
--discovery https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83
```

```
$ etcd --name infra1 --initial-advertise-peer-urls http://10.0.1.11:2380 \
--listen-peer-urls http://10.0.1.11:2380 \
--listen-client-urls http://10.0.1.11:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.11:2379 \
--discovery https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83
```

```
$ etcd --name infra2 --initial-advertise-peer-urls http://10.0.1.12:2380 \
--listen-peer-urls http://10.0.1.12:2380 \
--listen-client-urls http://10.0.1.12:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.12:2379 \
--discovery https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83
```

这将导致每个成员使用定制的 etcd 发现服务注册自身并开始集群，一旦所有的机器都已经注册。

## 公共 etcd 发现服务

如果没有现成的集群可用，可以使用托管在 `discovery.etcd.io` 的公共发现服务。为了使用"new" endpoint 来创建私有发现URL，使用命令：

```
$ curl https://discovery.etcd.io/new?size=3  
https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

这将创建带有初始化预期大小为3个成员的集群。如果没有指定大小，将使用默认值3。

```
ETCD_DISCOVERY=https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

```
--discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

每个成员必须有指定不同的名字标记。`Hostname` 或者 `machine-id` 是个好选择。否则发现会因为重复名字而失败

现在我们用这些用于每个成员的相关标记启动 etcd：

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \  
--listen-peer-urls http://10.0.1.10:2380 \  
--listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \  
--advertise-client-urls http://10.0.1.10:2379 \  
--discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

```
$ etcd --name infra1 --initial-advertise-peer-urls http://10.0.1.11:2380 \  
--listen-peer-urls http://10.0.1.11:2380 \  
--listen-client-urls http://10.0.1.11:2379,http://127.0.0.1:2379 \  
--advertise-client-urls http://10.0.1.11:2379 \  
--discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

```
$ etcd --name infra2 --initial-advertise-peer-urls http://10.0.1.12:2380 \  
--listen-peer-urls http://10.0.1.12:2380 \  
--listen-client-urls http://10.0.1.12:2379,http://127.0.0.1:2379 \  
--advertise-client-urls http://10.0.1.12:2379 \  
--discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

这将导致每个成员使用定制的etcd发现服务注册自身并开始集群，一旦所有的机器都已经注册。

使用环境变量 `ETCD_DISCOVERY_PROXY` 来让 etcd 使用 HTTP 代理来连接到发现服务。

## 错误和警告案例

发现服务错误

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
--listen-peer-urls http://10.0.1.10:2380 \
--listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.10:2379 \
--discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
etcd: error: the cluster doesn't have a size configuration value in https://discovery.
etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de/_config
exit 1
```

## 警告

这是一个无害的警告，表明发现 URL 将在这台机器上被忽略。

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
--listen-peer-urls http://10.0.1.10:2380 \
--listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://10.0.1.10:2379 \
--discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
etcdserver: discovery token ignored since a cluster has already been initialized. Valid log found at /var/lib/etcd
```

## DNS 发现

DNS SRV records 可以作为发现机制使用。

-discovery-srv 标记可以用于设置 DNS domain name，在这里可以找到发现 SRV 记录。

下列 DNS SRV 记录将以列出的顺序查找：

- \_etcd-server-ssl.\_tcp.example.com
- \_etcd-server.\_tcp.example.com

如果 \_etcd-server-ssl.\_tcp.example.com 被发现则 etcd 将在 TLS 上启动尝试启动进程。

为了帮助客户端发现 etcd 集群，下列 DNS SRV 记录将以列出的顺序查找：

- \_etcd-client.\_tcp.example.com
- \_etcd-client-ssl.\_tcp.example.com

如果发现 \_etcd-client-ssl.\_tcp.example.com，客户端将在 SSL/TLS 上尝试和 etcd 集群通讯。

如果 etcd 不带定义的证书 authority 使用 TLS，发现域名(如, example.com) 必须匹配 SRV record domain (例如, infra1.example.com). 这是为了减轻仿制 SRV 记录来指向不同域名的损害; 域名可能有 PKI 之下的有效证书，但是被未知的第三方控制。

## 创建 DNS SRV 记录

```
$ dig +noall +answer SRV _etcd-server._tcp.example.com
_etcd-server._tcp.example.com. 300 IN SRV 0 0 2380 infra0.example.com.
_etcd-server._tcp.example.com. 300 IN SRV 0 0 2380 infra1.example.com.
_etcd-server._tcp.example.com. 300 IN SRV 0 0 2380 infra2.example.com.
```

```
$ dig +noall +answer SRV _etcd-client._tcp.example.com
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra0.example.com.
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra1.example.com.
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra2.example.com.
```

```
$ dig +noall +answer infra0.example.com infra1.example.com infra2.example.com
infra0.example.com. 300 IN A 10.0.1.10
infra1.example.com. 300 IN A 10.0.1.11
infra2.example.com. 300 IN A 10.0.1.12
```

## 使用 DNS 启动 etcd 集群

etcd 集群成员可以监听域名或者IP地址，启动进程将解析 DNS A 记录。

在 `--initial-advertise-peer-urls` 中解析出来的地址 必须匹配 在 `SRV` 目录中解析出来的地址中的一个。 etcd 成员读取被解析的地址来发现它是否属于 `SRV` 记录定义的集群。

```
$ etcd --name infra0 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://infra0.example.com:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://infra0.example.com:2379 \
--listen-client-urls http://infra0.example.com:2379 \
--listen-peer-urls http://infra0.example.com:2380
```

```
$ etcd --name infra1 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://infra1.example.com:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://infra1.example.com:2379 \
--listen-client-urls http://infra1.example.com:2379 \
--listen-peer-urls http://infra1.example.com:2380
```

```
$ etcd --name infra2 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://infra2.example.com:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://infra2.example.com:2379 \
--listen-client-urls http://infra2.example.com:2379 \
--listen-peer-urls http://infra2.example.com:2380
```

集群也可以使用 IP 地址而不是域名来启动：

```
$ etcd --name infra0 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://10.0.1.10:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://10.0.1.10:2379 \
--listen-client-urls http://10.0.1.10:2379 \
--listen-peer-urls http://10.0.1.10:2380
```

```
$ etcd --name infra1 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://10.0.1.11:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://10.0.1.11:2379 \
--listen-client-urls http://10.0.1.11:2379 \
--listen-peer-urls http://10.0.1.11:2380
```

```
$ etcd --name infra2 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://10.0.1.12:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://10.0.1.12:2379 \
--listen-client-urls http://10.0.1.12:2379 \
--listen-peer-urls http://10.0.1.12:2380
```

## 网关

etcd 网关是一个简单的 TCP 代码，转发网络数据到 etcd 集群。请阅读 [网关指南](#) 来获取更多信息。

## 代理

当 `--proxy` 标记被设置时，etcd 运行于 [代理模式](#)。这个代理模式仅仅支持 etcd v2 API；没有支持 v3 API 的计划。取而代之的是，对于 v3 API 支持，在 etcd 3.0 发布之后将会有一个新的有增强特性的代理。

为了搭建带有v2 API的代理的 etcd 集群，请阅读 [clustering doc in etcd 2.3 release](#).

# 运行时重配置

注：内容翻译自 [Runtime Reconfiguration](#)

etcd 自带对渐进的运行时重配置的支持，这容许用户在运行时更新集群成员。

重配置请求仅能在集群成员的大多数正常工作时可以处理。强烈推荐 在产品中集群大小总是大于2.从一个两成员集群中移除一个成员是不安全的。在移除过程中如果有失败，集群可能无法前进而需要[从重大失败中重启](#).

为了更好的理解运行时重配置后面的设计，建议阅读 [运行时重配置文档](#).

## 重配置使用案例

让我们过一下一些重配置集群的常见理由。他们中的大多数仅仅涉及添加或者移除成员的组合，这些在后面的 [集群重配置操作](#) 下解释。

### 循环或升级多台机器

如果多个集群成员因为计划的维护(硬件升级，网络停工)而需要移动，推荐一次一个的修改多个成员。

移除 leader 是安全的，但是在选举过程发生的期间有短暂的停机时间。如果集群保存超过 50MB，推荐[迁移成员的数据目录](#).

### 修改集群大小

增加集群大小可以改善[失败容忍度](#) 并提供更好的读取性能。因为客户端可以从任意成员读取，增加成员的数量可以提高整体的读取吞吐量。

减少集群大小可以改善集群的写入性能，作为交换是降低弹性。写入到集群是要复制到集群成员的大多数才能被认定已经提交。减少集群大小消减了大多数的数量，从而每次写入可以更快提交。

### 替换失败的机器

如果机器因为硬件故障，数据目录损坏，或者一些其他致命情况而失败，它应该尽快被替代。已经失败但是还没有移除的机器对法定人数有不利影响并减低对额外失败的容忍性。

为了替换机器，遵循从集群中[移除成员](#) 的建议，然后再[添加新成员](#) 替代它的未知。如果集群保存超过 50MB，推荐[迁移失败成员的数据目录](#)，如果它还可以访问。

## 从多数失败中重启集群

如果集群的多数已经丢失或者所有的节点已经修改了IP地址，则需要手工动作来安全恢复。

恢复过程中的基本步骤包括 [使用旧有数据创建新的集群](#), 强制单个成员成，并最终使用运行时配置来一次一个 [添加新的成员](#) 到这个新的集群.

## 集群重配置操作

现在我们心里有使用案例了，让我们展示每个案例中涉及到的操作。

在任何变动前，etcd 成员的简单多数(quorum) 必须可用。

对于任何其他到 etcd 的写入，这也是根本性的同样要求。

所有集群的改动一次一个的完成：

- 要更新单个成员 peerURLs，做一个更新操作
- 要替代单个成员，做一个添加然后一个删除操作
- 要将成员从3增加到5, 做两次添加操作
- 要将成员从5减少到3, 做两次删除操作

所有这些案例将使用 etcd 自带的 `etcdctl` 命令行工具。

如果不使用 `etcdctl` 修改成员，可以使用 [v2 HTTP members API](#) 或者 [v3 gRPC members API](#).

### 更新成员

## 更新 **advertise client URLs**

为了更新成员的 `advertise client URLs`，简单用更新后的 `client URL` 标记(`--advertise-client-urls`)或者环境变量来重启这个成员(`ETCD_ADVERTISE_CLIENT_URLS`)。重新后的成员将自行发布更新后的URL。错误更新的 `client URL` 将不会影响 etcd 集群的健康。

## 更新 **advertise peer URLs**

要更新成员的 `advertise peer URLs`，首先通过成员命令更新它然后再重启成员。需要额外的行为是因为更新 `peer URL` 修改了集群范围配置并能影响 etcd 集群的健康。

要更新 `peer URL`，首先，我们需要找到目标成员的ID。使用 `etcdctl` 列出所有成员：

```
$ etcdctl member list
6e3bd23ae5f1eae0: name=node2 peerURLs=http://localhost:23802 clientURLs=http://127.0.0
.1:23792
924e2e83e93f2560: name=node3 peerURLs=http://localhost:23803 clientURLs=http://127.0.0
.1:23793
a8266ecf031671f3: name=node1 peerURLs=http://localhost:23801 clientURLs=http://127.0.0
.1:23791
```

在这个例子中，让我们 `更新` `a8266ecf031671f3` 成员ID并修改它的 `peerURLs` 值为 <http://10.0.1.10:2380>。

```
$ etcdctl member update a8266ecf031671f3 http://10.0.1.10:2380
Updated member with ID a8266ecf031671f3 in cluster
```

## 删除成员

假设我们要删除的成员ID是 `a8266ecf031671f3`.

我们随后用 `remove` 命令来执行删除:

```
$ etcdctl member remove a8266ecf031671f3
Removed member a8266ecf031671f3 from cluster
```

此时目标成员将停止自身并在日志中打印出移除信息：

```
etcd: this member has been permanently removed from the cluster. Exiting.
```

可以安全的移除 `leader`，当然在新 `leader` 被选举时集群将不活动(`inactive`)。这个持续时间通常是选举超时时间加投票过程。

## 添加新成员

添加成员的过程有两个步骤：

- 通过 [HTTP members API](#) 添加新成员到集群, [gRPC members API](#), 或者 `etcdctl member add` 命令.
- 使用新的层原配置启动新成员，包括更新后的成员列表(以后成员加新成员)

使用 `etcdctl` 指定 `name` 和 `advertised peer URLs` 来添加新的成员到集群:

```
$ etcdctl member add infra3 http://10.0.1.13:2380
added member 9bf1b35fc7761a23 to cluster

ETCD_NAME="infra3"
ETCD_INITIAL_CLUSTER="infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=
=http://10.0.1.12:2380,infra3=http://10.0.1.13:2380"
ETCD_INITIAL_CLUSTER_STATE=existing
```

`etcdctl` 已经给出关于新成员的集群信息并打印出成功启动它需要的环境变量。现在用关联的标记为新的成员启动新 `etcd` 进程：

```
$ export ETCD_NAME="infra3"
$ export ETCD_INITIAL_CLUSTER="infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=
=http://10.0.1.12:2380,infra3=http://10.0.1.13:2380"
$ export ETCD_INITIAL_CLUSTER_STATE=existing
$ etcd --listen-client-urls http://10.0.1.13:2379 --advertise-client-urls http://10.0.1.13:2379 --listen-peer-urls http://10.0.1.13:2380 --initial-advertise-peer-urls http://10.0.1.13:2380 --data-dir %data_dir%
```

新成员将作为集群的一部分运行并立即开始赶上集群的其他成员。

如果添加多个成员，最佳实践是一次配置单个成员并在添加更多新成员前验证它正确启动。

如果添加新成员到一个节点的集群，在新成员启动前集群无法继续工作，因为它需要两个成员作为galosh才能在一致性上达成一致。这个行为仅仅发生在 `etcdctl member add` 影响集群和新成员成功建立连接到已有成员的时间内。

## 添加成员时的错误案例

在下面的案例中，我们没有在列举节点的列表中包含新的host。如果这是一个新的集群，节点必须添加到初始化集群成员列表中。

```
$ etcd --name infra3 \
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=h
ttp://10.0.1.12:2380 \
--initial-cluster-state existing
etcdserver: assign ids error: the member count is unequal
exit 1
```

在这个案例中，我们给出一个和我们用来加入集群的(10.0.1.13:2380)不同的地址(10.0.1.14:2380)

```
$ etcd --name infra4 \
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=h
ttp://10.0.1.12:2380,infra4=http://10.0.1.14:2380 \
--initial-cluster-state existing
etcdserver: assign ids error: unmatched member while checking PeerURLs
exit 1
```

当我们使用一个被移除成员的数据目录来启动 etcd 时，etcd 将立即退出，如果它连接到任何集群中的活动成员：

```
$ etcd
etcd: this member has been permanently removed from the cluster. Exiting.
exit 1
```

## 严格重配置检查模式 ( `-strict-reconfig-check` )

如上所述，添加新成员的最佳实践是一次配置单个成员并在添加更多新成员前验证它正确启动。这个逐步的方式非常重要，因为如果最新添加的成员没有正确配置(例如 peer URL不正确)，集群会丢失法定人数。发生法定人数丢失是因为最新加入的成员被法定人数计数，即使这个成员对其他已经存在的成员是无法访问的。同样法定人数丢失可能发生在有连接问题或者操作问题时。

为了避免这个问题，etcd 提供选项 `-strict-reconfig-check`。如果这个选项被传递给 etcd，etcd 拒绝重配置请求，如果启动的成员的数量将少于被重配置的集群的法定人数。

推荐开启这个选项。当然，为了保持兼容它被默认关闭。

# 运行时重配置的设计

注：内容翻译自 [Design of runtime reconfiguration](#)

在分布式系统中，运行时重配置是最困难和最有错误倾向的特性，尤其是基于一致性的系统如 etcd 中。

继续阅读来学习关于 etcd 的运行时重配置命令的设计和我们如何解决这些问题。

## 两阶段配置修改保持集群安全

在 etcd 中，为了安全每个运行时重配置必须通过 [两阶段](#). 例如，为了添加成员，首先通知集群新配置然后再启动新成员。

### 阶段 1 - 通知集群新配置

为了添加成员到 etcd 集群，发起一个 API 调用来请求要添加一个新成员到集群。这是添加新成员到现有集群的唯一方法。当集群同意配置修改时 API 调用返回。

### 阶段 2 - 启动新成员

为了将 etcd 成员加入已有的集群，指定正确的 `initial-cluster` 并设置 `initial-cluster-state` 为 `existing`. 当成员启动时，它会首先联系已有的集群并验证当前集群配置匹配在 `initial-cluster` 中期待的配置。当新成员成功启动时，集群就达到了期待的配置。

通过将过程拆分为两个分离的阶段，用户被强制去明确关于集群成员的修改。这实际给了用户更多灵活性并让事情容易推导。例如，如果有尝试使用和集群中现有成员相同的 ID 添加新成员，这个行为将在阶段 1 期间立即失败而不影响运行中的集群。提供类似的保存来放置错误添加新成员。如果新 etcd 成员试图在集群接受配置修改前加入集群，它将无法被集群接受。

没有围绕集群成员的明确工作流，etcd 将因意外的集群成员修改而容易受伤。例如，如果 etcd 运行在初始化系统例如 systemd，etcd 在通过成员 API 移除之后将被重启，并试图在启动时重新加入集群。这个循环在每次成员被通过 API 删除时继续，而 systemd 在失败之后被设置为重启 etcd，这是出乎意料的。

我们期待运行时重配置是极少进行的操作。我们决定让它保持明确和用户驱动以保证配置安全和保持集群总是在明确控制下平稳运行。

## 法定人数永久丢失需要新集群

如果集群永久丢失了它的成员的多数，需要从旧有的数据目录启动新的集群来恢复之前的状态。

从已有集群中强制删除失败成员来恢复是完全可能的。但是，我们决定不支持这个方法，因为它绕开了正常的一致性提交阶段，这是不安全的。如果要删除的成员并没有实际死亡或者是通过在同一个集群中的不同成员强制删除，etcd 将以分离的有同样ID的集群方式结束。这非常危险而之后难于调试/修改。

正确部署时，永久多数丢失的可能性非常低。但是它是一个足够严重的问题，值得特别小心。强烈建议阅读 [灾难恢复文档](#) 并在产品中使用 etcd 前为永久多数丢失做好准备。

## 不要为运行时重配置使用公开发现服务

公开发现服务(discovery service)仅仅应该用于启动集群。要往已有集群中添加成员，使用运行时重配置API。

发现服务设计用于在云环境下启动 etcd 集群，当所有成员的 IP 地址实现不知道时。在成功启动集群后，所有成员的 IP 地址都已知。严格说，发现服务应该不再需要。

看上去使用公开发现服务是做运行时重配置的捷径，毕竟发现服务已经有所有集群配置信息。但是，依赖公开发现服务将带来问题：

1. 它为集群的完整生命周期引入额外依赖，而不仅仅是启动时间。如果在集群和公开发现服务之间有网络问题，集群将为此受折磨。
2. 公开发现服务必须考虑在生命周期期间集群的正确运行时配置。它必须提供安全机制来避免恶劣行为，而这是很困难的。
3. 公开发现服务必须保持数以万计的集群配置。我们的公开发现服务没有为这种负载做好准备。

要有支持运行时重配置的发现服务，最佳选择是搭建一个私有的。

# etcd 网关

注：内容翻译自 [etcd gateway](#)

## etcd 网关是什么

etcd 网关是一个简单的 TCP 代理，转发网络数据到 etcd 集群。网关是无状态和透明的；它既不检查客户端请求也不干涉集群应答。

网关支持多 etcd 服务器终端。当网关启动时，它随机的选择一个 etcd 服务器终端并转发所有请求到这个终端。这个终端服务锁偶请求直到网关发现一个网络失败。如果网关检测到终端失败，它将切换到其他的终端，如果可用，从而给它的客户端隐藏失败。其他重试策略，比如带权重的轮询，可能在未来支持。

## 何时使用 etcd 网关

每个访问 etcd 的应用必须首先要有 etcd 集群客户端终端的地址。如果在同一台服务器上的多个应用访问同一个 etcd 集群，每个应用都需要知道 etcd 集群的公告的客户端终端。如果 etcd 集群被重新配置为使用不同的终端，每个应用都需要更新它的终端列表。这个大规模的重新配置是令人生厌的，而且容易出错。

etcd 网关通过以稳定本地终端的方式来解决这个问题。典型 etcd 网关配置是在每台机器上运行网关监听在本地地址，而每个 etcd 应用连接到它本地的网关。结果是仅仅网关需要更新它的终端而不是更新每个应用。

总的来说，为了自动扩散集群终端变化，etcd 网关运行在每台机器，服务于访问同一个 etcd 集群的多个应用。

## 何时不使用 etcd 网关

- 提升性能

网关不是设计用来提升 etcd 集群的性能。它不提供缓存，监听联合或者批量。etcd 团队正在开发缓存代理，设计用于提升集群可扩展性。

- 运行在集群管理系统之上

高级集群管理系统比如 Kubernetes 原生支持服务发现。应用可以访问使用 DNS 名称或者系统管理的虚拟IP地址来访问 etcd 集群。例如，kube-proxy 等价于 etcd 网关。

## 启动 etcd 网关

假定 etcd 集群有下列静态终端：

Name	Address	Hostname
infra0	10.0.1.10	infra0.example.com
infra1	10.0.1.11	infra1.example.com
infra2	10.0.1.12	infra2.example.com

用命令来启动 etcd 网关来使用这些静态终端：

```
$ etcd gateway start --endpoints=infra0.example.com,infra1.example.com,infra2.example.com
2016-08-16 11:21:18.867350 I | tcpproxy: ready to proxy client requests to [...]
```

或者，如果使用 DNS 做服务发现，考虑 DNS SRV entries:

```
$ dig +noall +answer SRV _etcd-client._tcp.example.com
_etcdb-client._tcp.example.com. 300 IN SRV 0 0 2379 infra0.example.com.
_etcdb-client._tcp.example.com. 300 IN SRV 0 0 2379 infra1.example.com.
_etcdb-client._tcp.example.com. 300 IN SRV 0 0 2379 infra2.example.com.
```

```
$ dig +noall +answer infra0.example.com infra1.example.com infra2.example.com
infra0.example.com. 300 IN A 10.0.1.10
infra1.example.com. 300 IN A 10.0.1.11
infra2.example.com. 300 IN A 10.0.1.12
```

用命令来启动 etcd 网关来从 DSN SRV entries 中获取终端：

```
$ etcd gateway --discovery-srv=example.com
2016-08-16 11:21:18.867350 I | tcpproxy: ready to proxy client requests to [...]
```

# 在容器内运行etcd集群

注： 内容翻译自 [Run etcd clusters inside containers](#)

下列指南展示如何使用 [static bootstrap process](#) 来用rkt和docker运行 etcd 。

## rkt

### 运行单节点 etcd

下列 rkt 运行命令将在端口 2379 上暴露 etcd 客户端API，而在端口 2380上暴露伙伴API。

当配置 etcd 时使用 host IP地址。

```
export NODE1=192.168.1.21
```

信任 CoreOS App Signing Key.

```
sudo rkt trust --prefix coreos.com/etcd
# gpg key fingerprint is: 18AD 5014 C99E F7E3 BA5F  6CE9 50BD D3E0 FC8A 365E
```

运行 v3.0.6 版本的 etcd or 或者指定其他发布版本。

```
sudo rkt run --net=default:IP=${NODE1} coreos.com/etcd:v3.0.6 -- -name=node1 -advertis
e-client-urls=http://${NODE1}:2379 -initial-advertise-peer-urls=http://${NODE1}:2380 -
listen-client-urls=http://0.0.0.0:2379 -listen-peer-urls=http://${NODE1}:2380 -initial
-cluster=node1=http://${NODE1}:2380
```

列出集群成员。

```
etcdctl --endpoints=http://192.168.1.21:2379 member list
```

### 运行3节点集群

使用 rkt 本地搭建 3 节点的集群，使用 `-initial-cluster` 标记.

```
export NODE1=172.16.28.21
export NODE2=172.16.28.22
export NODE3=172.16.28.23
```

```
# node 1
sudo rkt run --net=default:IP=${NODE1} coreos.com/etcd:v3.0.6 -- -name=node1 -advertise-client-urls=http:// ${NODE1}:2379 -initial-advertise-peer-urls=http:// ${NODE1}:2380 -listen-client-urls=http://0.0.0.0:2379 -listen-peer-urls=http:// ${NODE1}:2380 -initial-cluster=node1=http:// ${NODE1}:2380, node2=http:// ${NODE2}:2380, node3=http:// ${NODE3}:2380

# node 2
sudo rkt run --net=default:IP=${NODE2} coreos.com/etcd:v3.0.6 -- -name=node2 -advertise-client-urls=http:// ${NODE2}:2379 -initial-advertise-peer-urls=http:// ${NODE2}:2380 -listen-client-urls=http://0.0.0.0:2379 -listen-peer-urls=http:// ${NODE2}:2380 -initial-cluster=node1=http:// ${NODE1}:2380, node2=http:// ${NODE2}:2380, node3=http:// ${NODE3}:2380

# node 3
sudo rkt run --net=default:IP=${NODE3} coreos.com/etcd:v3.0.6 -- -name=node3 -advertise-client-urls=http:// ${NODE3}:2379 -initial-advertise-peer-urls=http:// ${NODE3}:2380 -listen-client-urls=http://0.0.0.0:2379 -listen-peer-urls=http:// ${NODE3}:2380 -initial-cluster=node1=http:// ${NODE1}:2380, node2=http:// ${NODE2}:2380, node3=http:// ${NODE3}:2380
```

检验集群健康并可以到达。

```
ETCDCTL_API=3 etcdctl --endpoints=http://172.16.28.21:2379,http://172.16.28.22:2379,http://172.16.28.23:2379 endpoint-health
```

## DNS

通过被本地解析器已知的 DNS 名称指向伙伴的产品集群必须挂载 [主机的 DNS 配置](#).

## Docker

为了暴露 etcd API 到 docker host 之外的客户端，使用容器的 host IP 地址。请见 [docker inspect](#) 来获取关于如何得到IP地址的更多细节. 或者，为 `docker run` 命令指定 `--net=host` 标记来跳过放置容器在分隔的网络栈中。

```

# 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=10.20.30.1
HOST_2=10.20.30.2
HOST_3=10.20.30.3
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}

```

为了使用 API 版本3 来运行 etcdctl :

```
docker exec etcd /bin/sh -c "export ETCDCTL_API=3 && /usr/local/bin/etcdctl put foo bar"
```

## 裸机

为了在裸机上提供 3 节点 etcd 集群，可以在 [baremetal repo](#) 中搜索有用的案例。

# 配置标记

注：内容翻译自 [Configuration flags](#)

etcd可以通过命令行标记和环境变量来配置。命令行上设置的选项优先于环境变量。

对于标记 `--my-flag` 环境变量的格式是 `ETCD_MY_FLAG`。适用于所有标记。

正式的etcd端口是2379用于客户端连接，而2380用于伙伴通讯。etcd端口可以设置为接受TLS通讯，non-TLS通讯，或者同时有TLS和non-TLS通讯。

为了在linux启动时使用自定义设置自动启动etcd，强烈推荐使用[systemd](#)单元。

## 成员标记

### **--name**

- 成员的可读性的名字。
- 默认: "default"
- 环境变量: `ETCD_NAME`
- 这个值被作为这个节点自己的入口中被引用，在`--initial-cluster`标记(例如,  
`default=http://localhost:2380`)中列出。如果使用[static bootstrapping](#),这需要匹配在标记中使用的key。当使用发现时，每个成员必须有唯一名字。`Hostname`或`machine-id`可以是一个好选择。

### **--data-dir**

- 到数据目录的路径。
- 默认: "\${name}.etcd"
- 环境变量: `ETCD_DATA_DIR`

### **--wal-dir**

- 到专用的wal目录的路径。如果这个标记被设置，etcd将写WAL文件到walDIR而不是dataDIR。这容许使用专门的硬盘，并帮助避免日志和其他IO操作之间的IO竞争。
- 默认: ""
- 环境变量: `ETCD_WAL_DIR`

### **--snapshot-count**

- 触发快照到硬盘的已提交事务的数量。
- 默认: "10000"
- 环境变量: ETCD\_SNAPSHOT\_COUNT

## --heartbeat-interval

- 心跳间隔时间 (单位 毫秒)。
- 默认: "100"
- 环境变量: ETCD\_HEARTBEAT\_INTERVAL

## --election-timeout

- 选举的超时时间(单位 毫秒)。阅读 [Documentation/tuning.md](#) 得到更多详情。
- 默认: "1000"
- 环境变量: ETCD\_ELECTION\_TIMEOUT

## --listen-peer-urls

用于监听伙伴通讯的URL列表。这个标记告诉 etcd 在特定的 scheme://IP:port 组合上从它的伙伴接收进来的请求。scheme 可是 http 或者 https。如果IP被指定为 0.0.0.0,etcd 在所有接口上监听给定端口。如果给定IP地址和端口，etcd 将监听在给定端口和接口上。多个URL可以用来指定多个地址和端口来监听。etcd将从任何列出来的地址和端口上应答请求。

- 默认: "<http://localhost:2380>"
- 环境变量: ETCD\_LISTEN\_PEER\_URLS
- 例子: "<http://10.0.0.1:2380>"
- 无效例子: "<http://example.com:2380>" (对于绑定域名无效)

## --listen-client-urls

用于监听客户端通讯的URL列表。这个标记告诉 etcd 在特定的 scheme://IP:port 组合上从客户端接收进来的请求。scheme 可是 http 或者 https。如果IP被指定为 0.0.0.0,etcd 在所有接口上监听给定端口。如果给定IP地址和端口，etcd 将监听在给定端口和接口上。多个 URL 可以用来指定多个地址和端口来监听。etcd 将从任何列出来的地址和端口上应答请求。

- 默认: "<http://localhost:2379>"
- 环境变量: ETCD\_LISTEN\_CLIENT\_URLS
- 例子: "<http://10.0.0.1:2379>"
- 无效例子: "<http://example.com:2379>" (对于绑定域名无效)

## --max-snapshots

- 保持的快照文件的最大数量 (0 表示不限制)
- 默认: 5
- 环境变量: ETCD\_MAX\_SNAPSHOTS
- 对于 windows 用户默认不限制，而且推荐手工降低到5 (或者某些安全偏好)。

## --max-wals

- 保持的 wal 文件的最大数量 (0 表示不限制)
- 默认: 5
- 环境变量: ETCD\_MAX\_WALS
- 对于 windows 用户默认不限制，而且推荐手工降低到5 (或者某些安全偏好)。

## --cors

- 逗号分割的 origin 白名单，用于 CORS (cross-origin resource sharing/跨 origin 资源共享)。
- 默认: none
- 环境变量: ETCD\_CORS

# 集群标记

`--initial` 前缀标记用于启动([static bootstrap](#), [[discovery-service bootstrap](#)]) ([clustering.md#discovery](#)) 或 [runtime reconfiguration](#)) 新成员，然后当重新启动一个已有的成员时被忽略。

`--discovery` 前缀标记在使用 [发现服务](#) 需要设置。

## --initial-advertise-peer-urls

列出这个成员的伙伴 URL 以便通告给集群的其他成员。这些地方用于在集群中通讯 etcd 数据。至少有一个必须对所有集群成员可以路由的。这些 URL 可以包含域名。

- 默认: "<http://localhost:2380>"
- 环境变量: ETCD\_INITIAL\_ADVERTISE\_PEER\_URLS
- 例子: "<http://example.com:2380>, <http://10.0.0.1:2380>"

## --initial-cluster

为启动初始化集群配置。

- 默认: "default=<http://localhost:2380>"

- 环境变量: `ETCD_INITIAL_CLUSTER`
- `key`是每个提供的节点的 `--name` 标记的值。默认为这个 `key` 使用 `default` 因为这是 `--name` 标记的默认值。

## --initial-cluster-state

初始化集群状态("new" or "existing")。在初始化静态(initial static)或者 DNS 启动 (DNS bootstrapping) 期间为所有成员设置为 `new`。如果这个选项被设置为 `existing` , etcd 将试图加入已有的集群。如果设置为错误的值，etcd 将尝试启动但安全失败。

- 默认: "new"
- 环境变量: `ETCD_INITIAL_CLUSTER_STATE`

## --initial-cluster-token

在启动期间用于 etcd 集群的初始化集群记号(cluster token)。

- 默认: "etcd-cluster"
- 环境变量: `ETCD_INITIAL_CLUSTER_TOKEN`

## --advertise-client-urls

列出这个成员的客户端URL，通告给集群中的其他成员。这些 URL 可以包含域名。

- 默认: "<http://localhost:2379>"
- 环境变量: `ETCD_ADVERTISE_CLIENT_URLS`
- 例子: "<http://example.com:2379>, <http://10.0.0.1:2379>"

小心，如果来自集群成员的通告 URL 比如 <http://localhost:2379> 正在使用 etcd 的 proxy 特性。这将导致循环，因为代理将转发请求给它自己直到它的资源(内存，文件描述符)最终耗尽。

## --discovery

用于启动集群的发现URL。

- 默认: `none`
- 环境变量: `ETCD_DISCOVERY`

## --discovery-srv

用于启动集群的 DNS srv 域名。

- 默认: `none`

- 环境变量: ETCD\_DISCOVERY\_SRV

## --discovery-fallback

当发现服务失败时的期待行为("exit" 或 "proxy"). "proxy" 仅支持 v2 API.

- 默认: "proxy"
- 环境变量: ETCD\_DISCOVERY\_FALLBACK

## --discovery-proxy

用于请求到发现服务的 HTTP 代理。

- 默认: none
- 环境变量: ETCD\_DISCOVERY\_PROXY

## --strict-reconfig-check

拒绝将导致法定人数丢失的重配置请求。

- 默认: false
- 环境变量: ETCD\_STRICT\_RECONFIG\_CHECK

## --auto-compaction-retention

自动压缩用于 mvcc 键值存储的保持力(注：应该指多版本保存)，单位小时。0 表示关闭自动压缩。

- 默认: 0
- 环境变量: ETCD\_AUTO\_COMPACTION\_RETENTION

注：对于服务注册等只保存运行时动态信息的场合，建议开启。完全没有理由损失存储空间和效率来保存之前的版本信息。推荐设置为1,每小时压缩一次。

# Proxy flags

--proxy 前缀标记配置 etcd 以 [代理模式](#) 运行. "proxy" 仅支持 v2 API.

## --proxy

代理模式设置("off", "readonly" or "on").

- 默认: "off"

- 环境变量: ETCD\_PROXY

## --proxy-failure-wait

在被重新考虑之前，终端将被视为失败状态的时间(单位 毫秒)，用于被代理的请求。

- 默认: 5000
- 环境变量: ETCD\_PROXY\_FAILURE\_WAIT

## --proxy-refresh-interval

终端刷新间隔时间(单位 毫秒)

- 默认: 30000
- 环境变量: ETCD\_PROXY\_REFRESH\_INTERVAL

## --proxy-dial-timeout

请求的拨号(dial)超时时间(单位 毫秒)，或者 0 禁用超时。

- 默认: 1000
- 环境变量 ETCD\_PROXY\_DIAL\_TIMEOUT

## --proxy-write-timeout

写操作的超时时间(单位 毫秒)，或者 0 禁用超时。

- 默认: 5000
- 环境变量: ETCD\_PROXY\_WRITE\_TIMEOUT

## --proxy-read-timeout

读操作的超时时间(单位 毫秒)，或者 0 禁用超时。

不要修改这个值，如果在使用 watch，因为 watch 将使用 long polling 请求。

- 默认: 0
- 环境变量: ETCD\_PROXY\_READ\_TIMEOUT

## 安全标记

安全标记用于帮助 [搭建安全 etcd 集群](#).

## --ca-file [弃用]

客户端服务器 TLS 证书文件的路径。`--ca-file ca.crt` 可以被 `--trusted-ca-file ca.crt --client-cert-auth` 替代，而 etcd 同样工作。

- 默认: `none`
- 环境变量: `ETCD_CA_FILE`

## --cert-file

客户端服务器 TLS 证书文件的路径。

- 默认: `none`
- 环境变量: `ETCD_CERT_FILE`

## --key-file

客户端服务器 TLS key 文件的路径。

- 默认: `none`
- 环境变量: `ETCD_KEY_FILE`

## --client-cert-auth

开启客户端证书认证。

- 默认: `false`
- 环境变量: `ETCD_CLIENT_CERT_AUTH`

## --trusted-ca-file

客户端服务器 TLS 信任证书文件的路径。

- 默认: `none`
- 环境变量: `ETCD_TRUSTED_CA_FILE`

## --auto-tls

使用生成证书的客户端 TLS。

- 默认: `false`
- 环境变量: `ETCD_AUTO_TLS`

## --peer-ca-file [弃用]

peer server TLS 证书文件的路径. `--peer-ca-file ca.crt` 可以被 `--peer-trusted-ca-file ca.crt --peer-client-cert-auth` 替代，而 etcd 同样工作.

- 默认: none
- 环境变量: ETCD\_PEER\_CA\_FILE

## --peer-cert-file

peer server TLS 证书文件的路径.

- 默认: none
- 环境变量: ETCD\_PEER\_CERT\_FILE

## --peer-key-file

peer server TLS key 文件的路径.

- 默认: none
- 环境变量: ETCD\_PEER\_KEY\_FILE

## --peer-client-cert-auth

开启 peer client 证书验证.

- 默认: false
- 环境变量: ETCD\_PEER\_CLIENT\_CERT\_AUTH

## --peer-trusted-ca-file

peer server TLS 信任证书文件路径.

- 默认: none
- 环境变量: ETCD\_PEER\_TRUSTED\_CA\_FILE

## --peer-auto-tls

使用生成证书的peer TLS。

- 默认: false
- 环境变量: ETCD\_PEER\_AUTO\_TLS

---

# 日志标记

## --debug

设置所有子包的默认日志级别为 DEBUG

- 默认: false (所有包为 INFO)
- 环境变量: ETCD\_DEBUG

## --log-package-levels

设置个人 etcd 子包为指定日志级别。例如 `etcdserver=WARNING, security=DEBUG`

- 默认: none (所有包为 INFO)
- 环境变量: ETCD\_LOG\_PACKAGE\_LEVELS

## 不安全的标记

请谨慎使用不安全标记，因为它将打破一致性协议提供的保证。

例如，它可能惊慌，如果集群中的其他成员还活着。

当使用这些标记时，遵循操作指南。

## --force-new-cluster

强制创建新的单一成员的集群。它提交配置修改来强制移除集群中的所有现有成员然后添加自身。当 [restore a backup](#) 时需要设置。

- 默认: false
- 环境变量: ETCD\_FORCE\_NEW\_CLUSTER

## 其他标记

### --version

打印版本并退出。

- 默认: false

### --config-file

从文件中装载服务器配置。

- 默认: none

## 分析标记

### --enable-pprof

通过HTTP服务器开启运行时分析数据。地址是 client URL + "/debug/pprof/"

- 默认: false

## 加密模式

| 注： 内容翻译自 [Security Model](#)

TBD.....

# 维护

注： 内容翻译自 [Maintenance](#)

## 概述

etcd 集群需要定期维护来保持可靠。基于 etcd 应用的需要，这个维护通常可以自动执行，不需要停机或者显著的降低性能。

所有 etcd 的维护是指管理被 etcd 键空间消耗的存储资源。通过存储空间的配额来控制键空间大小;如果 etcd 成员运行空间不足，将触发集群级警告，这将使得系统进入有限操作的维护模式。为了避免没有空间来写入键空间，etcd 键空间历史必须被压缩。存储空间自身可能通过碎片整理 etcd 成员来回收。最后，etcd 成员状态的定期快照备份使得恢复任何非故意的逻辑数据丢失或者操作错误导致的损坏变成可能。

## 历史压缩

因为 etcd 保持它的键空间的确切历史，这个历史应该定期压缩来避免性能下降和最终的存储空间枯竭。压缩键空间历史删除所有关于被废弃的在给定键空间修订版本之前的键的信息。这些key使用的空间随机变得可用来继续写入键空间。

键空间可以使用 etcd 的时间窗口历史保持策略自动压缩，或者使用 etcdctl 手工压缩。  
etcdctl 方法在压缩过程中提供细粒度的控制，反之自动压缩适合仅仅需要一定时间长度的键历史的应用。

etcd 可以使用带有小时时间单位的 --auto-compaction 选项来设置为自动压缩键空间：

```
# 保持一个小时的历史
$ etcd --auto-compaction-retention=1
```

etcdctl 如下发起压缩工作：

```
# 压缩到修订版本3
$ etcdctl compact 3
```

在压缩修订版本之前的修订版本变得无法访问：

```
$ etcdctl get --rev=2 somekey
Error: rpc error: code = 11 desc = etcdserver: mvcc: required revision has been compacted
```

## 反碎片化

在压缩键空间之后，后端数据库可能出现内部。内部碎片是指可以被后端使用但是依然消耗存储空间的空间。反碎片化过程释放这个存储空间到文件系统。反碎片化在每个成员上发起，因此集群范围的延迟尖峰(latency spike)可能可以避免。

通过留下间隔在后端数据库，压缩旧有修订版本会内部碎片化 `etcd`。碎片化的空间可以被 `etcd` 使用，但是对于主机文件系统不可用。

为了反碎片化 `etcd` 成员，使用 `etcdctl defrag` 命令：

```
$ etcdctl defrag
Finished defragmenting etcd member[127.0.0.1:2379]
```

## 空间配额

在 `etcd` 中空间配额确保集群以可靠方式运作。没有空间配额，`etcd` 可能会收到低性能的困扰，如果键空间增长的过度的巨大，或者可能简单的超过存储空间，导致不可预测的集群行为。如果键空间的任何成员的后端数据库超过了空间配额，`etcd` 发起集群范围的警告，让集群进入维护模式，仅接收键的读取和删除。在键空间释放足够的空间之后，警告可以被解除，而集群将恢复正常运作。

默认，`etcd` 设置适合大多数应用的保守的空间配额，但是它可以在命令行中设置，单位为字节：

```
# 设置非常小的 16MB 配额
$ etcd --quota-backend-bytes=16777216
```

空间配额可以用循环触发：

```
# 消耗空间
$ while [ 1 ]; do dd if=/dev/urandom bs=1024 count=1024 | etcdctl put key || break;
done
...
Error: rpc error: code = 8 desc = etcdserver: mvcc: database space exceeded
# 确认配额空间被超过
$ etcdctl --write-out=table endpoint status
+-----+-----+-----+-----+-----+
|      ENDPOINT      |      ID       |  VERSION   | DB SIZE | IS LEADER | RAFT TERM | RA
FT INDEX |
+-----+-----+-----+-----+-----+-----+
| 127.0.0.1:2379 | bf9071f4639c75cc | 2.3.0+git | 18 MB  | true     |          2 |
3332 |
+-----+-----+-----+-----+-----+-----+
# 确认警告已发起
$ etcdctl alarm list
memberID:13803658152347727308 alarm:NOSPACE
```

删除多读的键空间将把集群带回配额限制，因此警告能被接触：

```
# 获取当前修订版本
$ etcdctl --endpoints=:2379 endpoint status
[{"Endpoint": "127.0.0.1:2379", "Status": {"header": {"cluster_id": 8925027824743593106, "me
mber_id": 13803658152347727308, "revision": 1516, "raft_term": 2}, "version": "2.3.0+git", "db
Size": 17973248, "leader": 13803658152347727308, "raftIndex": 6359, "raftTerm": 2}}
# 压缩所有旧有修订版本
$ etcdctl compact 1516
compacted revision 1516
# 反碎片化过度空间
$ etcdctl defrag
Finished defragmenting etcd member[127.0.0.1:2379]
# 解除警告
$ etcdctl alarm disarm
memberID:13803658152347727308 alarm:NOSPACE
# 测试put被再度容许
$ etcdctl put newkey 123
OK
```

## 快照备份

在正规基础上执行 `etcd` 集群快照可以作为 `etc` 键空间的持久备份。通过获取 `etcd` 成员的候选数据库的定期快照，`etcd` 集群可以被恢复到某个有已知良好状态的时间点。

通过 `etcdctl` 获取快照：

```
$ etcdctl snapshot save backup.db
$ etcdctl --write-out=table snapshot status backup.db
+-----+-----+-----+
| HASH | REVISION | TOTAL KEYS | TOTAL SIZE |
+-----+-----+-----+
| fe01cf57 |      10 |          7 | 2.1 MB    |
+-----+-----+-----+
```

# 理解失败

注：内容翻译自 [Understand failures](#)

在机器的大量部署中失败是很常见的。当硬件或者软件故障时单台机器失败。当电力故障或者网络问题时多台机器一起失败。多种失败也可能一起发生；几乎不可能列举出所有可能的失败场景。

在这节中，我们分类失败的种类并讨论 etcd 是如何设计来容忍这些失败的。大部分用户，不是所有，可以映射一个特别的失败到一种失败。为了应对罕见或者 [不可恢复的失败](#)，总是 [备份 etcd 集群](#)。

## 少数跟随者失败

当少于一半的跟随者失败时，etcd 集群依然可以接收请求并让程序没有任何大的中断。例如，两个跟随者失败将不会影响一个五个成员的 etcd 集群运作。但是，客户端将失去到失败成员的连通性。对于读请求客户端类库应该通过自动重新连接到其他成员来对用户隐藏这些中断。运维人员应该预期其他成员上的系统负载会因为重新连接而提升。

## Leader 失败

当 leader 失败时，etcd 集群自动选举一个新的 leader。选举不会在 leader 失败之后立即发生。大约需要一个选举超时的时间来选举新的 leader，因为失败选取模型是基于超时的。

在 leader 选举的期间，集群不能处理任何写。在选举期间发送的写请求将排队等待处理知道新的 leader 被选举出来。

已经发送给旧有 leader 但是还没有提交的写请求可能会丢失。新的 leader 有能力重新写入从之前 leader 而来的任何未提交的条目。从用户的角度，在新的 leader 选举之后某些写请求可能超时。无论如何，已提交的请求从来不会丢失。

新的 leader 自动延长所有的租约(lease)。这个机制保证足浴将不会在准许的 TTL 之前过期，即使它是被旧有的 leader 准许。

## 多数失败

当集群的大多数成员失败时，etcd 集群失败并无法接收更多写请求。

etcd 集群仅在成员的大多数可用时可以从多数失败中恢复。如果成员的大多数无法回来上线，那么运维必须启动[disaster recovery/灾难恢复](#) 来恢复集群。

一旦成员的大多数可以工作， etcd 集群自动选举新的 leader 并返回到健康状态。新的 leader 自动延长是所有租约的超时时间。这个机器确保没有租约因为服务器端不可访问而过期。

## 网络分区

网络分区类似少数跟随者失败或者 leader 失败。网络分区将 etcd 集群分成两个部分；一个有大多数成员而另外一个有少数成员。多数这边变成可用集群而少数这边不可用。在 etcd 中没有 "脑裂"。

如果 leader 在多数这边，那么从多数这边的角度看失败是一个少数跟随者失败。如果 leader 在少数这边，那么它是 leader 失败。在少数这边的 leader 辞职(step down)然后多数那边选举新的 leader。

一旦网络分区清除，少数这边自动承认来自多数这边的 leader 并恢复状态。

## 启动期间失败

集群启动时仅仅当所有要求的成员都成功启动才视为成功。如果在启动期间发生任何失败，在所有成员上删除数据目录并用新的集群记号(cluster-token)或者新的发现记号(discovery token)重新启动集群。

当然，可以像恢复运行中的集群那样恢复失败的启动集群。但是，它大多数情况下总是比启动一个新的花费更多时间和资源来恢复集群，因为没有任何数据要恢复。

# 灾难恢复

注：内容翻译自 [Disaster recovery](#)

`etcd` 被设计为能承受机器失败。`etcd` 集群自动从临时失败(例如，机器重启)中恢复，而且对于一个有  $N$  个成员的集群能容许  $(N-1)/2$  的持续失败。当一个成员持续失败时，不管是因硬件失败或者磁盘损坏，它丢失到集群的访问。如果集群持续丢失超过  $(N-1)/2$  的成员，则它只能悲惨的失败，无可救药的失去法定人数(quorum)。一旦法定人数丢失，集群无法达到一致而因此无法继续接收更新。

为了从灾难失败中恢复，`etcd v3` 提供快照和修复工具来重建集群而不丢失 `v3` 键数据。要恢复 `v2` 的键，参考[v2 管理指南](#)。

## 快照键空间

恢复集群首先需要来自 `etcd` 成员的键空间的快照。快速可以是用 `etcdctl snapshot save` 命令从活动成员获取，或者是从 `etcd` 数据目录复制 `member/snap/db` 文件。例如，下列命令快照在 `$ENDPOINT` 上服务的键空间到文件 `snapshot.db`：

```
$ etcdctl --endpoints $ENDPOINT snapshot save snapshot.db
```

## 恢复集群

为了恢复集群，需要的只是一个简单的快照 "db" 文件。使用 `etcdctl snapshot restore` 的集群恢复创建新的 `etcd` 数据目录;所有成员应该使用相同的快照恢复。恢复覆盖某些快照元数据(特别是，成员ID和集群ID);成员丢失它之前的标识。这个元数据覆盖防止新的成员不经意间加入已有的集群。因此为了从快照启动集群，恢复必须启动一个新的逻辑集群。

在恢复时快照完整性的检验是可选的。如果快照是通过 `etcdctl snapshot save` 得到的，它将有一个被 `etcdctl snapshot restore` 检查过的完整性hash。如果快照是从数据目录复制而来，没有完整性hash，因此它只能通过使用 `--skip-hash-check` 来恢复。

恢复初始化新集群的新成员，带有新的集群配置，使用 `etcd` 的集群配置标记，但是保存 `etcd` 键空间的内容。继续上面的例子，下面为一个3成员的集群创建新的 `etcd` 数据目录 (`m1.etcd`，`m2.etcd`，`m3.etcd`)：

```
$ etcdctl snapshot restore snapshot.db \
--name m1 \
--initial-cluster m1=http://host1:2380,m2=http://host2:2380,m3=http://host3:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-advertise-peer-urls http://host1:2380
$ etcdctl snapshot restore snapshot.db \
--name m2 \
--initial-cluster m1=http://host1:2380,m2=http://host2:2380,m3=http://host3:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-advertise-peer-urls http://host2:2380
$ etcdctl snapshot restore snapshot.db \
--name m3 \
--initial-cluster m1=http://host1:2380,m2=http://host2:2380,m3=http://host3:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-advertise-peer-urls http://host3:2380
```

下一步, 用新的数据目录启动 `etcd` :

```
$ etcd \
--name m1 \
--listen-client-urls http://host1:2379 \
--advertise-client-urls http://host1:2379 \
--listen-peer-urls http://host1:2380 &
$ etcd \
--name m2 \
--listen-client-urls http://host2:2379 \
--advertise-client-urls http://host2:2379 \
--listen-peer-urls http://host2:2380 &
$ etcd \
--name m3 \
--listen-client-urls http://host3:2379 \
--advertise-client-urls http://host3:2379 \
--listen-peer-urls http://host3:2380 &
```

现在恢复的集群可以使用并提供来自快照的键空间服务.

# 性能

注：内容翻译自 [Performance](#)

## 理解性能

etcd 提供稳定的，持续的高性能。两个定义性能的因素：延迟(latency)和吞吐量(throughput)。延迟是完成操作的时间。吞吐量是在某个时间期间之内完成操作的总数量。当 etcd 接收并发客户端请求时，通常平均延迟随着总体吞吐量增加而增加。在通常的云环境，比如 Google Compute Engine (GCE) 标准的 n-4 或者 AWS 上相当的机器类型，一个三成员 etcd 集群在轻负载下可以在低于1毫秒内完成一个请求，并在重负载下可以每秒完成超过 30000 个请求。

etcd 使用 Raft 一致性算法来在成员之间复制请求并达成一致。一致性性能，特别是提交延迟，受限于两个物理约束：网络IO延迟和磁盘IO延迟。完成一个etcd请求的最长时间是成员之间的网络往返时延(Round Trip Time / RTT)，加需要提交数据到持久化存储的 fdatasync 时间。在一个数据中心内的 RTT 可能有数百毫秒。在美国典型的 RTT 是大概 50ms, 而在大陆之间可以慢到400ms. 旋转硬盘(注：指传统机械硬盘)的典型 fdatasync 延迟是大概 10ms。对于 SSD 硬盘，延迟通常低于 1ms. 为了提高吞吐量，etcd 将多个请求打包在一起并提交给 Raft。这个批量策略让 etcd 在重负载时获得高吞吐量。

有其他子系统影响到 etcd 的整体性能。每个序列化的 etcd 请求必须通过 etcd 的 boltdb 支持的(boltdb-backed) MVCC 存储引擎，它通常需要 10 微秒来完成。etcd 定期递增快照它最近实施的请求，将他们和之前在磁盘上的快照合并。这个过程可能导致延迟尖峰(latency spike)。虽然在SSD上这通常不是问题，在HDD上它可能加倍可观察到的延迟。而且，进行中的压缩可以影响 etcd 的性能。幸运的是，压缩通常无足轻重，因为压缩是错开的，因此它不和常规请求竞争资源。RPC 系统，gRPC，为 etcd 提供定义良好，可扩展的 API，但是它也引入了额外的延迟，尤其是本地读取。

## 评测

可以通过 etcd 自带的 [benchmark](#) CLI 工具来评测 etcd 的性能。

对于某些基线性能数字，我们考虑的是3成员 etcd 集群，带有下列硬件配置：

- Google Cloud Compute Engine
- 3 台机器， 8 vCPUs + 16GB Memory + 50GB SSD
- 1 台机器(客户端)，16 vCPUs + 30GB Memory + 50GB SSD
- Ubuntu 15.10

- etcd v3 master 分支 (commit SHA d8f325d), Go 1.6.2

使用这些配置，etcd 能近似的写入：

key 的数量	Key 的大小	Value 的大小	连接数量	客户端数量	目标 etcd 服务器	平均写入 QPS	每请求平均延迟	内存
10,000	8	256	1	1	leader only	525	2ms	35 MB
100,000	8	256	100	1000	leader only	25,000	30ms	35 MB
100,000	8	256	100	1000	all members	33,000	25ms	35 MB

注：key 和 value 的大小单位是字节 / bytes

采样命令是：

```
# 假定 IP_1 是 leader, 写入请求发到 leader
benchmark --endpoints={IP_1} --conns=1 --clients=1 \
    put --key-size=8 --sequential-keys --total=10000 --val-size=256
benchmark --endpoints={IP_1} --conns=100 --clients=1000 \
    put --key-size=8 --sequential-keys --total=100000 --val-size=256

# 写入发到所有成员
benchmark --endpoints={IP_1},{IP_2},{IP_3} --conns=100 --clients=1000 \
    put --key-size=8 --sequential-keys --total=100000 --val-size=256
```

为了保证一致性，线性化(Linearizable)读取请求要通过集群成员的法定人数来获取最新的数据。串行化(Serializable)读取请求比线性化读取要廉价一些，因为他们是通过任意单台 etcd 服务器来提供服务，而不是成员的法定人数，代价是可能提供过期数据。etcd 可以读取：

请求数量	Key 大小	Value 大小	连接数量	客户端数量	一致性	每请求平均延迟	平均读取 QPS
10,000	8	256	1	1	Linearizable	2ms	560
10,000	8	256	1	1	Serializable	0.4ms	7,500
100,000	8	256	100	1000	Linearizable	15ms	43,000
100,000	8	256	100	1000	Serializable	9ms	93,000

采样命令是：

```
# Linearizable 读取请求
benchmark --endpoints={IP_1},{IP_2},{IP_3} --conns=1 --clients=1 \
    range YOUR_KEY --consistency=l --total=10000
benchmark --endpoints={IP_1},{IP_2},{IP_3} --conns=100 --clients=1000 \
    range YOUR_KEY --consistency=l --total=100000

# Serializable 读取请求，使用每个成员然后将数字加起来
for endpoint in {IP_1} {IP_2} {IP_3}; do
    benchmark --endpoints=$endpoint --conns=1 --clients=1 \
        range YOUR_KEY --consistency=s --total=10000
done
for endpoint in {IP_1} {IP_2} {IP_3}; do
    benchmark --endpoints=$endpoint --conns=100 --clients=1000 \
        range YOUR_KEY --consistency=s --total=100000
done
```

当在新的环境中第一次搭建 etcd 集群时，我们鼓励运行评测测试来保证集群达到足够的性能；集群延迟和吞吐量对微小的环境差异会很敏感。

# 版本

注： 内容翻译自 [Versioning](#)

## 服务版本

etcd 使用 [semantic versioning](#)。新的小版本可能增加额外功能到API。

使用 `etcdctl` 获取运行中的 etcd 集群的版本：

```
ETCDCTL_API=3 etcdctl --endpoints=127.0.0.1:2379 endpoint status
```

## API 版本

在 3.0.0 发布值偶 `v3` API 应答将不会更改，但是随着时间的推移将加入新的功能。

# 支持平台

注： 内容翻译自 [Supported platforms](#)

## 当前支持

下面的表单列出了常见架构和系统的 etcd 支持状态：

架构	操作系统	状态	维护者
amd64	Darwin	实现性	etcd maintainers
amd64	Linux	稳定	etcd maintainers
amd64	Windows	实现性	
arm64	Linux	实现性	@glevand
arm	Linux	不稳定	
386	Linux	不稳定	

- etcd-维护者被列举在 <https://github.com/coreos/etcd/blob/master/MAINTAINERS>.

试验性的平台似乎在不断练习中工作，并在 etcd 中有一些平台特有代码，而没有完全遵守稳定支持策略。不稳定平台有轻度测试，那是比试验性少。为列出的架构和操作系统当前不支持，请当心！

## 支持新平台

对于 etcd 官方支持新的稳定平台，有一些要求是必须的，以保证可接受的质量：

1. 一个这个平台的 "官方" 的维护者，有清晰的动力;必须有人负责照看这个平台。
2. 搭建构建的CI; etcd 必须编译
3. 搭建用于运行单元测试的CI; etcd 必须通过简单的测试。
4. 搭建CI (TravisCI, SemaphoreCI 或 Jenkins) 用于运行集成测试; etcd 必须通过加强测试。
5. (可选) 搭建功能测试集群; etcd 集群应该能通过压力测试。

## 32-位 和其他未支持系统

由于go runtime 的 bug，etcd 在32位系统上有众所周知的问题。阅读 [Go issue](#) 和 [atomic package](#) 来获取详细信息。

为了避免不经意的运行可能不稳定的 etcd 服务器，在不稳定或者未支持架构上的 etcd 将打印警告信息并立即退出，如果环境变量 `ETCD_UNSUPPORTED_ARCH` 没有设置为目标架构。

当前仅有 **amd64** 架构被 etcd 官方支持。

# 学习

## 内容

- Why etcd (TODO)
- 理解数据模型
- 理解API
- 术语
- API保证
- Internals (TODO)

## 额外说明

1. 这些内容主要是介绍 etcd 的概念和实现细节，适合希望深入了解 etcd 的同学。
2. 内容来自 github 官网，地址：

<https://github.com/coreos/etcd/tree/master/Documentation/learning>

# 数据模型

注：内容翻译自 [data model](#)

`etcd`设计用于可靠存储不频繁更新的数据，并提供可靠的观察查询。`etcd`暴露键值对的先前版本来支持不昂贵的快速和观察历史事件("time travel queries")。对于这些使用场景，持久化，多版本，并发控制的数据模型是非常适合的。

`etcd`使用多版本持久化键值存储来存储数据。当键值对的值被新的数据替代时，持久化键值存储保存先前版本的键值对。键值存储事实上是不可变的;它的操作不会就地更新结构，替代的是总是生成一个新的更新后的结构。在修改之后，`key`的所有先前版本还是可以访问和观察的。为了防止随着时间的过去为了维护老版本导致数据存储无限增长，存储应该压缩来脱离被替代的数据的最旧的版本。

## 逻辑视图

存储的逻辑视图是一个扁平的二进制键空间。键空间有一个在`byte string`键上的语义排序的索引，因此范围查询不是太昂贵。

键空间维护多个版本。每个原子变化操作(类如，一个事务操作可能包含多个操作)在键空间上创建一个新的修订版本。先前版本持有的所有数据保持不变。`key`的旧有版本还可以通过先前修订版本访问。同样的，修订版本也是被索引的;在修订版本上搜索(带观察者)是高效的。一旦存储被压缩来恢复空间，在压缩修订版本之前的修订版本被移除。

`key`的生命周期跨越一代(spans a generation)。每个`key`可以有一代或者多代。创建`key`增加了`key`的代，如果`key`从未存在则从1开始。删除一个`key`产生一个`key`的墓碑，结束`key`的当前时代。`key`的每次改动创建`key`的一个新的版本。一旦压缩发生，任何在给定修订版本之前结束的时代将被删除，而在压缩修订版本之前设置的值，除最后一个外，都将被移除。

## 物理视图

`etcd`以持久性 [b+tree](#) 键值对的方式存储物理数据。存储的状态的每个修订版本仅仅包含和它的前一个修订版本的增量以求高效。单个修订版本可能对应到`tree`上的多个`key`。

键值对的`key`是三元数组(`major, sub, type`)。`Major`是持有`key`的存储修订版本。`Sub`区分离一个修订版本的不同`key`。`Type`是用于特别值(例如，`t`，如果值包含一个墓碑)的可选后缀。键值对的值包含对前一个修订版本的改动，例如对前一个修订版本的增量。`b+tree`以`key`的词典字节顺序排序。在修订版本增量上的延伸查找(Ranged lookups)很快。这样可以快速发送从一个特定修订版本到另一个的改变。压缩删除过期的键值对。

etcd也保持内存中的第二btree索引来加速key的范围查询。在btree索引中的key是存储暴露给用户的key。值是到持久化b+tree修改的指针。压缩删除死指针。

# etcd3 API

注：内容翻译自 [etcd3 API](#)

注意：这个文档还没有完成！

注：原文如此，的确是还没有完成 :)

## Response header

从etcd API返回的所有应答都附带有 [response header](#)。这个response header包含应答的元数据。

```
message ResponseHeader {  
    uint64 cluster_id = 1;  
    uint64 member_id = 2;  
    int64 revision = 3;  
    uint64 raft_term = 4;  
}
```

- Cluster\_ID - 生成应答的集群的ID
- Member\_ID - 生成应答的成员的ID
- Revision - 当应答生成时键值存储的修订版本
- Raft\_Term - 当应答生成时成员的 Raft term

应用可以读取 Cluster\_ID (Member\_ID) 字段来确保它正在和预期的集群(成员)通讯。

应用可以使用 `Revision/修订版本` 来获知键值存储最新的修订版本。当应用指定一个历史修订版本来实现 `time travel query` 并希望知道请求时刻最新的修订版本时有用。

应用可以使用 `Raft_Term` 来检测集群何时完成了新的leader选举。

## 键值 API

键值API用于操作etcd中的键值对存储。键值API被定义为 [gRPC服务](#)。在 [protobuf](#) 格式中键值对被定义为结构化的数据。

### 键值对

键值对是键值API可以操作的最小单元。每个键值对有一些字段：

```
message KeyValue {  
    bytes key = 1;  
    int64 create_revision = 2;  
    int64 mod_revision = 3;  
    int64 version = 4;  
    bytes value = 5;  
    int64 lease = 6;  
}
```

- **Key** - 字节数组形式的key。key不容许空。
- **Value** - 字节数组形式的value
- **Version** - key的版本。删除将重置版本为0而key的任何修改将增加它的版本。
- **Create\_Revision** - key最后一次创建的修订版本。
- **Mod\_Revision** - key最后一次修改的修订版本。
- **Lease** - 附加到key的租约的ID。如果lease为0,则表示没有租约附加到key。

# 术语

注： 内容翻译自 [Glossary](#)

这份文档定义etcd文档，命令行和源代码中使用的多个术语。

## Node / 节点

Node/节点是raft状态机的一个实例。

它有唯一标识，并内部记录其他节点的发展，如果它是leader。

## Member / 成员

Member/成员是etcd的一个实例。它承载一个node/节点，并为client/客户端提供服务。

## Cluster / 集群

Cluster/集群由多个member/成员组成。

每个成员的节点遵循 raft 一致性协议来复制日志。集群从成员中接收提案，提交他们并应用到本地存储。

## Peer / 同伴

Peer/同伴是同一个集群中的其他成员。

## Proposal / 提议

提议是一个需要完成raft协议的请求(例如写请求，配置修改请求)。

## Client / 客户端

Client/客户端是集群HTTP API的调用者。

注： 对于V3,应该包括gRPC API。

## **Machine / 机器 (弃用)**

在 2.0 版本之前，在 etcd 中的成员备选。

# KV API 保证

注： 内容翻译自 [KV API guarantees](#)

注2： 在官网文档页面中没有看到这个文档的链接，我是在git仓库的leaning目录下找到的。

etcd是一致而持久的键值存储，带有微事务(mini-transaction)。键值对存储通过KV API暴露。etcd力图为分布式系统获取最强的一致性和持久性保证。这份规范列举etcd实现的 KV API 保证。

## 考虑的 APIs

- 读 APIs
  - range
  - watch
- 写 APIs
  - put
  - delete
- 联合 (读-改-写) APIs
  - txn

## etcd 明确定义

### 操作完成

当etcd操作通过一致同意而提交时，视为操作完成，并且因此"执行过" -- 永久保存 -- 被etcd存储引擎。当客户端从etcd服务器接收到应答时，客户端得知操作已经完成。注意，如果客户端和etcd成员之间有超时或者网络中断，客户端可能不确定操作的状态。当有leader选举时，etcd也可能中止操作。在这个事件中，etcd不会发送 `abort` 应答给客户端的未完成的请求。

### revision/修订版本

修改键值存储的etcd操作被分配有一个唯一的增加的修订版本。事务操作可能多次修改键值存储，但是只分配一个修订版本。被操作修改的键值对的revision属性和操作的revision有同样的值。修订版本可以用来给键值存储做逻辑时间。有较大修订版本的键值对在有较小修订版本的键值对之后修改。两个拥有相同修订版本的键值对是被一个操作同时修改。

### 提供的保证

## 原子性

所有API请求都是原子的；操作或者完整的完成，或者完全不。对于观察请求，一个操作生成的所有事件将在一个观察应答中。观察绝不会看到一个操作的部分事件。

## 一致性

所有API调用确保 [顺序一致性](#)，分布式系统可用的最强的一致性保证。不管客户端的请求发往哪个etcd成员，客户端以同样的顺序读取相同的事件。如果两个成员完成同样数量的操作，这两个成员的状态是一致的。

对于观察操作，etcd保证所有成员对于同样的修订版本返回同样的key返回同样的值。对于范围操作，etcd有类似的线性一致性访问保证；连续访问可能落后于法定人数状态([quorum state](#))，因此后来的修订版本还不用。

这段好晦涩，原文：For range operations, etcd has a similar guarantee for linearized access; serialized access may be behind the quorum state, so that the later revision is not yet available.

和所有分布式系统一样，对etcd来说确保强一致性是不可能的。etcd不保证它会给读操作返回在任意集群成员上可以访问的“最近”(most recent)()的值

叹息一声，翻译不好，原文：etcd does not guarantee that it will return to a read the “most recent” value (as measured by a wall clock when a request is completed) available on any cluster member.

## 隔离

etcd确保串行化隔离，这是在分布式系统中可用的最高隔离级别。读操作从不查看任何中间数据。

## 持久性

任何完成的操作都是持久的。所有可访问的数据也都是持久的数据。读从来不会返回没有持久化的数据。

## 线性一致性

线性一致性（也被称为 [Atomic Consistency](#) 或者 [External Consistency](#)）是介于 [strict consistency](#)/严格一致性和 [sequential consistency](#)/顺序一致性之间的一致性级别。

对于线性一致性，假设每个操作从宽松的已同步全局时钟获取时间戳。如果并且仅当操作总是完成操作是线性化的

操作是线性的，如果并且仅当操作总是完成的好像他们是按顺序执行并且每个操作看上去以程序指定的顺序完成那样。

这句话翻译的好痛苦，还是看原文吧: Operations are linearized if and only if they always complete as though they were executed in a sequential order and each operation appears to complete in the order specified by the program.

同样的，如果操作的时间戳发生在其他操作前，这个操作在顺序上必须也发生在其他操作前。

例如，考虑客户端在时间点 $t_1$ ( $t_1$ )完成一个写操作。客户端在 $t_2$  ( $t_2 > t_1$ )发送的读请求应该收到至少在上一次在 $t_1$ 时间点完成的写之后最近的值。然后，读可能实际在 $t_3$ 完成，而返回值，在 $t_2$ 时刻开始读，可能在 $t_3$ 时刻已经不再新鲜。

对于观察操作etcd不保证线性一致性。期望用户验证观察应答的修订版本来确保正确顺序。

对于所有其他操作etcd默认保证线性一致性。线性一致性会带来开销，无论如何，因为线性化请求必须通过Raft一致性过程。为了获得读请求更低的延迟和更高的吞吐量，客户端可以配置请求的一致性模型为 `serializable` ,这可能访问不新鲜的数据(*with respect to quorum*)，但是移除了线性化访问时依赖活动一致性的性能损失。

# API 参考文档

## 前言

这份文档来自 ectd3 的 [API reference](#)。

原文内容是从 .proto 文件生成的，所有的 service 和他们的方法定义/message 定义混合在一起构成一个巨大的单页文件。

看起来非常累.....

所以干脆拆分出来，按照服务/方法分开，看起来方便。

主要内容在 message 的字段定义上，我翻译时没有复制原文中从 .proto 文件生成的表格，而是直接在 .proto 文件的 message 定义上翻译，感觉更直观一些。

注：我假定阅读这份 API 的同学对 proto3 的基本语法有初步的了解，如果没有，可以参考我的另外一份 [proto3 学习笔记](#)。

## 内容

已经整理并翻译的内容：

- [KV service](#)
- [Watch service](#)
- [Lease service](#)

# KV service

KV service 提供对键值对操作的支持。

在 `rpc.proto` 文件中 KV service 定义如下：

```
service KV {
    // 从键值存储中获取范围内的 key.
    rpc Range(RangeRequest) returns (RangeResponse) {}

    // 设置给定 key 到键值存储.
    // put 请求增加键值存储的修订版本并在事件历史中生成一个事件.
    rpc Put(PutRequest) returns (PutResponse) {}

    // 从键值存储中删除给定范围。
    // 删除请求增加键值存储的修订版本并在事件历史中为每个被删除的key生成一个删除事件.
    rpc DeleteRange(DeleteRangeRequest) returns (DeleteRangeResponse) {}

    // 在单个事务中处理多个请求。
    // 一个 txn 请求增加键值存储的修订版本并为每个完成的请求生成带有相同修订版本的事件。
    // 不容许在一个txn中多次修改同一个key.
    rpc Txn(TxnRequest) returns (TxnResponse) {}

    // 压缩在etcd键值存储中的事件历史。
    // 键值存储应该定期压缩，否则事件历史会无限制的持续增长.
    rpc Compact(CompactionRequest) returns (CompactionResponse) {}
}
```

# Range方法

Range方法从键值存储中获取范围内的key.

```
rpc Range(RangeRequest) returns (RangeResponse) {}
```

注意没有操作单个key的方法，即使是存取单个key，也是需要使用 Range 方法的。

## 消息体

请求的消息体是RangeRequest：

```

message RangeRequest {
    enum SortOrder {
        NONE = 0; // 默认, 不排序
        ASCEND = 1; // 正序, 低的值在前
        DESCEND = 2; // 倒序, 高的值在前
    }
    enum SortTarget {
        KEY = 0;
        VERSION = 1;
        CREATE = 2;
        MOD = 3;
        VALUE = 4;
    }

    // key是range的第一个key。如果 range_end 没有给定, 请求仅查找这个key
    bytes key = 1;

    // range_end 是请求范围的上限[key, range_end)
    // 如果 range_end 是 '\0', 范围是大于等于 key 的所有key。
    // 如果 range_end 比给定的key长一个bit, 那么范围请求获取所有带有前缀(给定的key)的key
    // 如果 key 和 range_end 都是'\0', 则范围查询返回所有key
    bytes range_end = 2;

    // 请求返回的key的数量限制
    int64 limit = 3;

    // 修订版本是用于范围的键值对存储的时间点。
    // 如果 revision 小于或等于零, 范围是在最新的键值对存储上。
    // 如果修订版本已经被压缩, 返回 ErrCompacted 作为应答
    int64 revision = 4;

    // 指定返回结果的排序顺序
    SortOrder sort_order = 5;

    // 用于排序的键值字段
    SortTarget sort_target = 6;

    // 设置范围请求使用串行化成员本地读(serializable member-local read)。
    // 范围请求默认是线性化的;线性化请求相比串行化请求有更高的延迟和低吞吐量,但是反映集群当前的一致性。
    // 为了更好的性能,以可能脏读为交换,串行化范围请求在本地处理,无需和集群中的其他节点达到一致。
    bool serializable = 7;

    // 设置仅返回key而不需要value
    bool keys_only = 8;

    // 设置仅仅返回范围内key的数量
    bool count_only = 9;
}

```

应答的消息体是 RangeResponse :

```

message RangeResponse {
    ResponseHeader header = 1;

    // kvs是匹配范围请求的键值对列表
    // 当请求数量时是空的
    repeated mvccpb.KeyValue kvs = 2;

    // more代表在被请求的范围内是否还有更多的key
    bool more = 3;

    // 被请求范围内key的数量
    int64 count = 4;
}

```

**mvccpb.KeyValue** 的消息体：

```

message KeyValue {
    // key是bytes格式的key。不容许key为空。
    bytes key = 1;

    // create_revision 是这个key最后创建的修订版本
    int64 create_revision = 2;

    // mod_revision 是这个key最后修改的修订版本
    int64 mod_revision = 3;

    // version 是key的版本。删除会重置版本为0,而任何key的修改会增加它的版本。
    int64 version = 4;

    // value是key持有的值，bytes格式。
    bytes value = 5;

    // lease是附加给key的租约id。
    // 当附加的租约过期时，key将被删除。
    // 如果lease为0, 则没有租约附加到key。
    int64 lease = 6;
}

```

# Put 方法

Put方法放置给定key到键值存储.

Put方法增加键值存储的修订版本并在事件历史中生成一个事件.

```
rpc Put(PutRequest) returns (PutResponse) {}
```

## 消息体

请求的消息体是 PutRequest :

```
message PutRequest {
    // byte数组形式的key，用来放置到键值对存储
    bytes key = 1;

    // byte数组形式的value，在键值对存储中和key关联
    bytes value = 2;

    // 在键值存储中和key关联的租约id。0代表没有租约。
    int64 lease = 3;

    // 如果 prev_kv 被设置，etcd获取改变之前的上一个键值对。
    // 上一个键值对将在put应答中被返回
    bool prev_kv = 4;
}
```

应答的消息体是 PutResponse :

```
message PutResponse {
    ResponseHeader header = 1;

    // 如果请求中的 prev_kv 被设置，将会返回上一个键值对
    mvccpb.KeyValue prev_kv = 2;
}
```

# DeleteRange 方法

DeleteRange 方法从键值存储中删除给定范围。

删除请求增加键值存储的修订版本并在事件历史中为每个被删除的key生成一个删除事件。

```
rpc DeleteRange(DeleteRangeRequest) returns (DeleteRangeResponse)
```

## 消息体

请求的消息体是 DeleteRangeRequest :

```
message DeleteRangeRequest {
    // key是要删除的范围的第一个key
    bytes key = 1;

    // range_end 是要删除范围[key, range_end)的最后一个key
    // 如果 range_end 没有给定，范围定义为仅包含key参数
    // 如果 range_end 是 '\0'， 范围是所有大于等于参数key的所有key。
    bytes range_end = 2;

    // 如果 prev_kv 被设置，etcd获取删除之前的上一个键值对。
    // 上一个键值对将在delete应答中被返回
    bool prev_kv = 3;
}
```

应答的消息体是 DeleteRangeResponse :

```
message DeleteRangeResponse {
    ResponseHeader header = 1;

    // 被范围删除请求删除的key的数量
    int64 deleted = 2;

    // 如果请求中的 prev_kv 被设置，将会返回上一个键值对
    repeated mvccpb.KeyValue prev_kvs = 3;
}
```

# Txn 方法

Txn 方法在单个事务中处理多个请求。

一个 txn 请求增加键值存储的修订版本并为每个完成的请求生成带有相同修订版本的事件。

不容许在一个 txn 中多次修改同一个 key。

```
rpc Txn(TxnRequest) returns (TxnResponse) {}
```

## 背景

以下内容翻译来自 proto 文件中 TxnRequest 的注释，解释了 Txn 请求的工作方式。

来自 google paxosdb 论文：

我们的实现围绕强大的我们称为 MultiOp 的原生(primitive)。所有除了循环外的其他数据库操作被实现为对 MultiOp 的单一调用。MultiOp 被原子性的应用并由三个部分组成：

1. 被称为 guard 的测试列表。在 guard 中每个测试检查数据库中的单个项(entry)。它可能检查某个值的存在或者缺失，或者和给定的值比较。在 guard 中两个不同的测试可能应用于数据库中相同或者不同的项。guard 中的所有测试被应用然后 MultiOp 返回结果。如果所有测试是 true，MultiOp 执行 t 操作(见下面的第二项)，否则它执行 f 操作(见下面的第三项)。
2. 被称为 t 操作的数据库操作列表。列表中的每个操作是插入，删除，或者查找操作，并应用到单个数据库项。列表中的两个不同操作可能应用到数据库中相同或者不同的项。如果 guard 评价为 true 这些操作将被执行
3. 被成为 f 操作的数据库操作列表。类似 t 操作，但是是在 guard 评价为 false 时执行。

## 消息体

请求的消息体是 TxnRequest：

```
message TxnRequest {  
    // compare 是断言列表，体现为条件的联合。  
    // 如果比较成功，那么成功请求将被按顺序处理，而应答将按顺序包含他们对应的应答。  
    // 如果比较失败，那么失败请求将被按顺序处理，而应答将按顺序包含他们对应的应答。  
    repeated Compare compare = 1;  
  
    // 成功请求列表，当比较评估为 true 时将被应用。  
    repeated RequestOp success = 2;  
  
    // 失败请求列表，当比较评估为 false 时将被应用。  
    repeated RequestOp failure = 3;  
}
```

应答的消息体是 **TxnResponse** :

```
message TxnResponse {  
    ResponseHeader header = 1;  
  
    // 如果比较评估为true则succeeded被设置为true，否则是false  
    bool succeeded = 2;  
  
    // 应答列表，如果 succeeded 是 true 则对应成功请求，如果 succeeded 是 false 则对应失败请求  
    repeated ResponseOp responses = 3;  
}
```

**Compare** 消息体 :

```

message Compare {
    enum CompareResult {
        EQUAL = 0;
        GREATER = 1;
        LESS = 2;
    }
    enum CompareTarget {
        VERSION = 0;
        CREATE = 1;
        MOD = 2;
        VALUE= 3;
    }
}

// result是这个比较的逻辑比较操作
CompareResult result = 1;

// target是比较要检查的键值字段
CompareTarget target = 2;

// key是用于比较操作的主题key
bytes key = 3;

oneof target_union {
    // version是给定key的版本
    int64 version = 4;

    // create_revision 是给定key的创建修订版本
    int64 create_revision = 5;

    // mod_revision 是给定key的最后修改修订版本
    int64 mod_revision = 6;

    // value 是给定key的值，以bytes的形式
    bytes value = 7;
}
}

```

RequestOp 消息体：

```

message RequestOp {
    // request 是可以被事务接受的请求类型的联合
    oneof request {
        RangeRequest request_range = 1;
        PutRequest request_put = 2;
        DeleteRangeRequest request_delete_range = 3;
    }
}

```

ResponseOp 消息体：

```
message ResponseOp {  
    // response 是事务返回的应答类型的联合  
    oneof response {  
        RangeResponse response_range = 1;  
        PutResponse response_put = 2;  
        DeleteRangeResponse response_delete_range = 3;  
    }  
}
```

# Compact 方法

Compact 方法压缩在 etcd 键值存储中的事件历史。

键值存储应该定期压缩，否则事件历史会无限制的持续增长。

```
rpc Compact(CompactionRequest) returns (CompactionResponse) {}
```

## 消息体

请求的消息体是 PutRequest：

```
message CompactionRequest {  
    // 用于比较操作的键值存储的修订版本  
    int64 revision = 1;  
  
    // physical 设置为 true 时 RPC 将会等待知道压缩物理性的应用到本地数据库，到这程度被压缩的项将完全  
    // 从后端数据库中移除。  
    bool physical = 2;  
}
```

应答的消息体是 PutResponse：

```
message CompactionResponse {  
    ResponseHeader header = 1;  
}
```

# Watch service

Watch service 提供观察键值对变化的支持。

在 rpc.proto 文件中 Watch service 定义如下：

```
service Watch {  
    // Watch 观察将要发生或者已经发生的事件。  
    // 输入和输出都是流；输入流用于创建和取消观察，而输出流发送事件。  
    // 一个观察 RPC 可以一次性在多个key范围上观察，并为多个观察流化事件。  
    // 整个事件历史可以从最后压缩修订版本开始观察。  
    rpc Watch(stream WatchRequest) returns (stream WatchResponse) {}  
}
```

只有一个 Watch 方法。

# Watch 方法

WatchService 只有一个 Watch 方法。

## 消息定义

请求的消息体是 WatchRequest :

```
message WatchRequest {  
    // request_union 要么是创建新的观察者的请求，要么是取消一个已经存在的观察者的请求  
    oneof request_union {  
        WatchCreateRequest create_request = 1;  
        WatchCancelRequest cancel_request = 2;  
    }  
}
```

创建新的观察者的请求 WatchCreateRequest :

```

message WatchCreateRequest {
    // key 是注册要观察的key
    bytes key = 1;

    // range_end 是要观察的范围 [key, range_end) 的终点。
    // 如果 range_end 没有设置，则只有参数key被观察。
    // 如果 range_end 等同于 '\0'，则大于等于参数key的所有key都将被观察
    bytes range_end = 2;

    // start_revision 是可选的开始(包括)观察的修订版本。不设置 start_revision 则表示 "现在".
    int64 start_revision = 3;

    // 设置 progress_notify ，这样如果最近没有事件，etcd 服务器将定期的发送不带任何事件的 WatchRe
    sponse 给新的观察者。
    // 当客户端希望从最近已知的修订版本开始恢复断开的观察者时有用。
    // etcd服务器将基于当前负载决定它发送通知的频率。
    bool progress_notify = 4;

    enum FilterType {
        // 过滤掉 put 事件
        NOPUT = 0;

        // 过滤掉 delete 事件
        NODELETE = 1;
    }

    // 过滤器，在服务器端发送事件给回观察者之前，过滤掉事件。
    repeated FilterType filters = 5;

    // 如果 prev_kv 被设置，被创建的观察者在事件发生前获取上一次的KV。
    // 如果上一次的KV已经被压缩，则不会返回任何东西
    bool prev_kv = 6;
}

```

取消已有观察者的 WatchCancelRequest :

```

message WatchCancelRequest {
    // watch_id 是要取消的观察者的id，这样就不再有更多事件传播过来了。
    int64 watch_id = 1;
}

```

应答的消息体 WatchResponse :

```

message WatchResponse {
    ResponseHeader header = 1;
    // watch_id 是和应答相关的观察者的ID
    int64 watch_id = 2;

    // 如果应答是用于创建观察者请求的，则 created 设置为true。
    // 客户端应该记录 watch_id 并期待从同样的流中为创建的观察者接收事件。
    // 所有发送给被创建的观察者的事件将附带同样的watch_id
    bool created = 3;

    // 如果应答是用于取消观察者请求的，则 canceled 设置为true。
    // 不再有事件发送给被取消的观察者。
    bool canceled = 4;

    // compact_revision 被设置为最小index，如果观察者试图观察被压缩的index。
    // 当在被压缩的修订版本上创建观察者或者观察者无法追上键值对存储的进展时发生。
    // 客户端应该视观察者为被取消，并不应该试图再次创建任何带有相同 start_revision 的观察者。
    int64 compact_revision = 5;

    repeated mvccpb.Event events = 11;
}

```

`mvccpb.Event` 的消息体：

```

message Event {
    enum EventType {
        PUT = 0;
        DELETE = 1;
    }

    // type是事件的类型。
    // 如果类型是 PUT，表明新的数据已经存储到key。
    // 如果类型是 DELETE， 表明key已经被删除。
    EventType type = 1;

    // kv 为事件持有KeyValue。
    // PUT 事件包含当前的kv键值对
    // DELETE/EXPIRE 事件包含被删除的key，它的修改修订版本设置为删除的修订版本
    KeyValue kv = 2;

    // prev_kv 持有在事件发生前的键值对
    KeyValue prev_kv = 3;
}

```

# Lease service

Lease service 提供租约的支持。

在 rpc.proto 文件中 Lease service 定义如下：

```
service Lease {
    // LeaseGrant 创建一个租约，当服务器在给定时间内没有接收到 keepAlive 时租约过期。
    // 如果租约过期则所有附加在租约上的key将过期并被删除。
    // 每个过期的key在事件历史中生成一个删除事件。
    rpc LeaseGrant(LeaseGrantRequest) returns (LeaseGrantResponse) {}

    // LeaseRevoke 撤销一个租约。
    // 所有附加到租约的key将过期并被删除。
    rpc LeaseRevoke(LeaseRevokeRequest) returns (LeaseRevokeResponse) {}

    // LeaseKeepAlive 通过从客户端到服务器端的流化的keep alive 请求和从服务器端到客户端的流化的keep alive 应答来维持租约。
    rpc LeaseKeepAlive(stream LeaseKeepAliveRequest) returns (stream LeaseKeepAliveResponse) {}
}
```

# LeaseGrant 方法

LeaseGrant 方法创建一个租约。

```
rpc LeaseGrant(LeaseGrantRequest) returns (LeaseGrantResponse) {}
```

## 消息体

请求的消息体是 LeaseGrantRequest：

```
message LeaseGrantRequest {
    // TTL 是建议的以秒为单位的 time-to-live
    int64 TTL = 1;

    // ID 是租约的请求ID。如果ID设置为0，则出租人(也就是etcd server)选择一个ID。
    int64 ID = 2;
}
```

应答的消息体是 LeaseGrantResponse：

```
message LeaseGrantResponse {
    ResponseHeader header = 1;
    // ID 是承认的租约的ID
    int64 ID = 2;
    // TTL 是服务器选择的以秒为单位的租约time-to-live
    int64 TTL = 3;
    string error = 4;
}
```

两个待后续继续理解的细节：

1. 请求中的TTL和应答中的TTL：貌似请求只是给个建议，而服务器可能接收或者给出自己的决定。因此处理时要考虑两者不一致的情况？什么情况服务器会不依从客户端的建议。
2. 应答中的error：什么情况下会使用？

# LeaseGrant 方法

LeaseGrant 方法取消一个租约.

```
rpc LeaseRevoke(LeaseRevokeRequest) returns (LeaseRevokeResponse) {}
```

## 消息体

请求的消息体是 LeaseRevokeRequest :

```
message LeaseRevokeRequest {  
    // ID是要取消的租约的ID。  
    // 当租约被取消时，所有关联的key将被删除  
    int64 ID = 1;  
}
```

应答的消息体是 LeaseRevokeResponse :

```
message LeaseRevokeResponse {  
    ResponseHeader header = 1;  
}
```

# LeaseKeepAlive 方法

LeaseKeepAlive 方法取消一个租约.

```
rpc LeaseKeepAlive(stream LeaseKeepAliveRequest) returns (stream LeaseKeepAliveResponse)
e) {}
```

## 消息体

请求的消息体是 LeaseKeepAliveRequest :

```
message LeaseKeepAliveRequest {
    // ID是要继续存活的租约的ID
    int64 ID = 1;
}
```

应答的消息体是 LeaseKeepAliveResponse :

```
message LeaseKeepAliveResponse {
    ResponseHeader header = 1;
    // ID是从继续存活请求中得来的租约ID
    int64 ID = 2;
    // TTL是租约新的 time-to-live
    int64 TTL = 3;
}
```

## Tags