ETCD 项目解析

技术原理

etcd 通过 Raft 协议实现分布式一致性，结合 WAL 和 MVCC 保证数据持久化与高效查询，利用租约和 Watch 机制支持动态协调，最终构建了一个高可靠、强一致的分布式存储系统。其设计体现了分布式系统的经典理论（如 CAP、Paxos/Raft）与现代工程实践（如 gRPC、TLS）的深度融合。

1. 分布式一致性算法（Raft 协议）

etcd 使用 Raft 分布式一致性算法来保证集群中所有节点数据的一致性。Raft 能够确保即使在网络分区或节点故障的情况下，所有节点都能就数据的状态达成一致。这意味着数据不会因为某个节点的故障而丢失或损坏。

2. 高可用性（HA）设计

- **多副本机制**：数据在多个节点（通常 3 或 5 个）间复制，容忍节点故障。
- **强一致性模型**：遵循 **Linearizable Consistency**，保证所有客户端看到一致的读写顺序。
- **租约（Lease）机制**：通过租约管理资源生命周期，避免死锁或资源泄漏（如 Kubernetes 的 Pod 状态管理）。
- **快照（Snapshot）**：定期生成数据快照，用于快速恢复或新节点加入时的数据同步。

3. 数据存储与持久化

- **预写式日志（WAL, Write-Ahead Log）**：所有操作先写入不可变的 WAL 文件，确保崩溃恢复时数据不丢失。
- **键值存储引擎**：
  - 使用 **B+ 树**（早期版本）或 **BoltDB**（基于 B+ 树的嵌入式数据库）管理键值对。
  - 支持多版本并发控制（MVCC），每个修改生成新的版本号（Revision），实现快照隔离。

- **数据压缩与碎片整理**：定期压缩旧版本数据，减少存储占用。

4. 分布式事务

- **事务原子性**：支持多键操作的原子性提交（如 `TXN` 命令）。
- **条件操作**：基于键的当前值或版本号实现条件更新（Compare-and-Swap, Compare-and-Delete）。

5. 性能优化

- **批处理（Batching）**：将多个请求合并为一个批次，减少网络开销。
- **线性读与串行读**：
  - **Linearizable Read**：通过 Leader 节点保证强一致性读。
  - **Serializable Read**：允许从 Follower 节点读取旧数据，提高吞吐量。
- **Watch 机制**：客户端可以监听键的变化，通过长轮询或 gRPC 流实现实时通知。

6. 安全性

- **TLS 加密**：支持客户端与服务器间通信的加密（mTLS）。
- **基于角色的访问控制（RBAC）**：通过用户（User）和角色（Role）管理权限。
- **审计日志**：记录所有操作，用于安全分析和合规性检查。

7. 客户端交互协议

- **gRPC 通信**：基于 HTTP/2 和 Protocol Buffers，实现高效的客户端-服务器通信。
- **API 设计**：提供简洁的键值操作 API（如 `Put`、`Get`、`Delete`、`Watch`）。

8. 监控与诊断

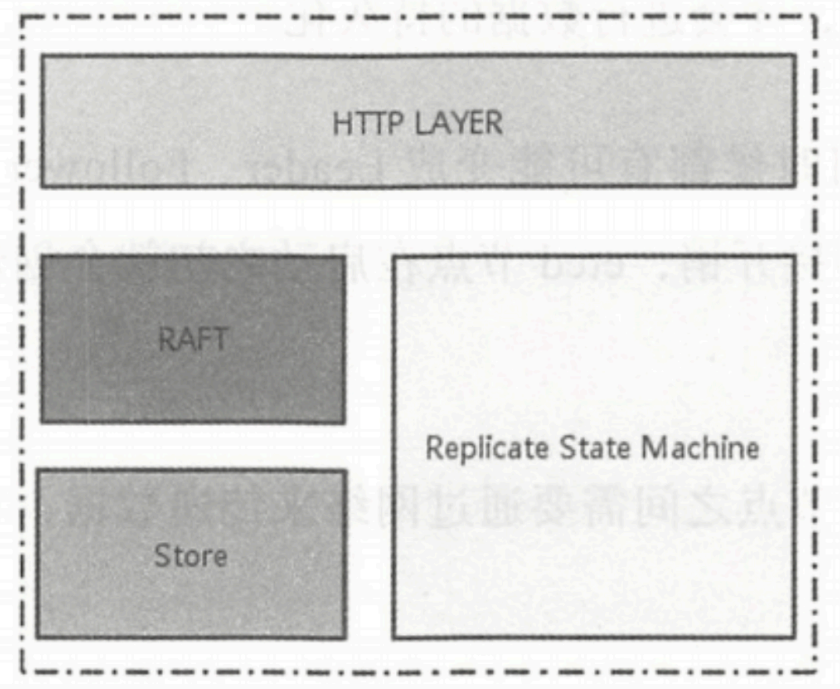
- **Prometheus 指标**：暴露集群健康状态、请求延迟、存储用量等指标。
- **日志分级**：支持 Debug、Info、Warn、Error 等多级日志，便于故障排查。

工程实现

项目结构

etcd 的核心模块包括 lease、mvcc、raft 和 etcdserver。

etcd server 模块组成



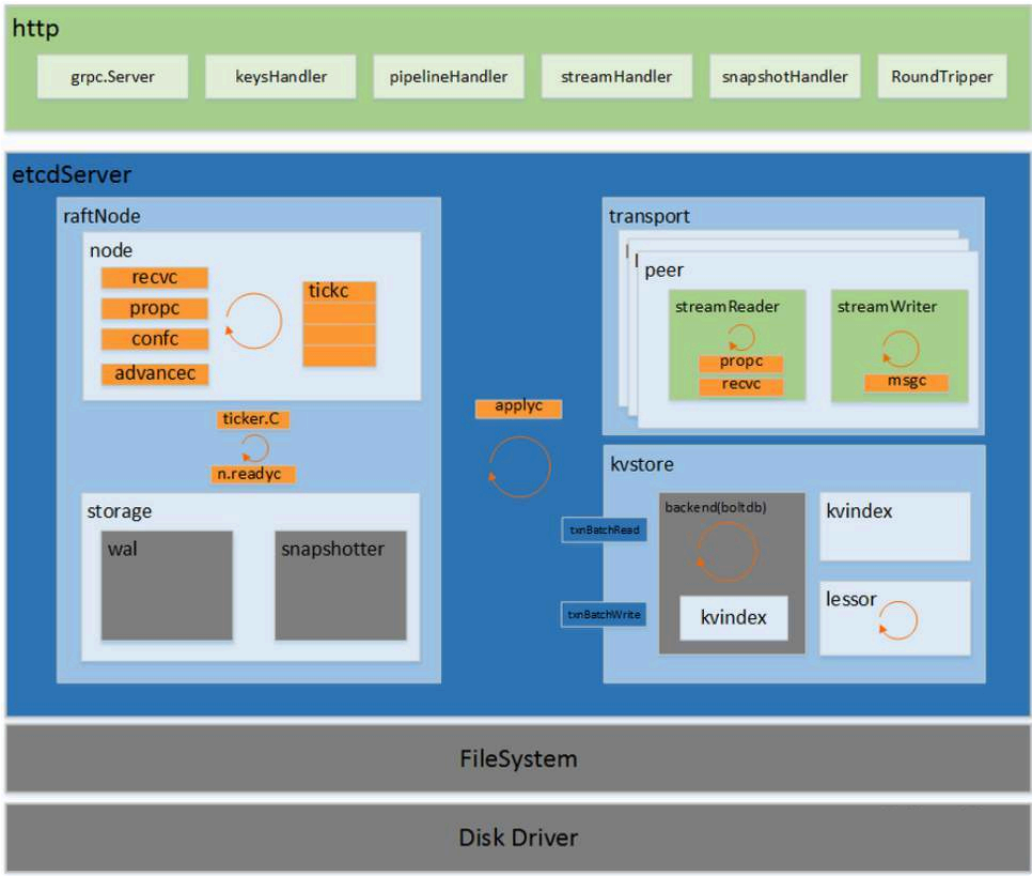
网络层：提供网络数据读写功能，监听服务端口，完成集群节点之间数据通信，收发客户端数据。

Raft 模块：Raft 强一致性算法的具体实现。

存储模块：涉及 KV 存储、WAL 文件、Snapshot 管理等，用户处理 etcd 支持的各类功能的事务，包括数据索引节点状态变更、监控与反馈、事件处理与执行 ，是 etcd 对用户提供的大多数 API 功能的具体实现。

复制状态机：这是一个抽象的模块，状态机的数据维护在内存中，定期持久化到磁盘，每次写请求都会持久化到 WAL 文件，并根据写请求的内容修改状态机数据。

整体架构



从大体上可以将其划分为以下 5 层：

1. **客户端层**：包含 clientv3 和 etcdctl 等客户端，通过 RESTful 风格的 API 简化 etcd 的使用。客户端层具有负载均衡和节点间故障转移等特性，提高 etcd 服务端的高可用性。
2. **API 接口层**：提供客户端访问服务端的通信协议和接口定义，以及服务端节点之间相互通信的协议。Etcd v3 使用 gRPC 作为消息传输协议，而 etcd v2 默认使用 HTTP/1.x 协议。对于不支持 gRPC 的客户端语言，etcd 提供 JSON 的 grpc-gateway，将 HTTP/JSON 请求转换为 gRPC 的 Protocol Buffer 格式消息。
3. **etcd Raft 层**：负责 Leader 选举和日志复制等功能，与本节点的 etcd Server 通信，并与其他 etcd 节点交互，实现分布式一致性数据同步。RaftLog 管理 raft 协议中单个节点的日志，存储在内存中。RaftLog 中包含 unstable 和 storage 两种结构体，分别用于存储不稳定的数据（尚未 commit）和已 commit 的数据。Raft 库负责与集群中的其他 etcd Server 进行交互，实现分布式一致性。
4. **逻辑层**：包括鉴权、租约、KVServer、MVCC 和 Compactor 压缩等核心功能特性。
5. **etcd 存储层**：实现了快照和预写式日志 WAL（Write Ahead Log）。Etcd v3 版本使用 BoltDB 来持久化存储集群元数据和用户写入的数据。

模块交互

客户端请求的处理流程涉及以下模块：

1. 客户端
2. API 接口层
3. etcd Server
4. etcd raft 算法库

请求首先到达 etcd Server，经过 KVServer 拦截，实现日志、Metrics 监控、请求校验等功能。Etcd Server 中的 raft 模块与 etcd-raft 库进行通信。applierV3 模块封装了 etcd v3 版本的数据存储，WAL 用于写入数据日志，保存任期号、投票信息、已提交索引、提案内容等，etcd 根据 WAL 中的内容在启动时恢复，实现集群的数据一致性。

请求流程

客户端与 etcd 集群的交互包括以下步骤：

1. 客户端通过负载均衡算法选择一个 etcd 节点，发起 gRPC 调用。
2. etcd Server 收到客户端请求。
3. 经过 gRPC 拦截、Quota 校验（校验 etcd db 文件大小是否超过配额）。
4. KVServer 模块将请求发送给本模块中的 raft，发起一个提案。
5. Raft 将数据封装成 raft 日志的形式提交给 raft 模块，首先保存到 raftLog 的 unstable 存储部分。
6. Raft 模块通过 raft 协议与集群中其他 etcd 节点进行交互。

在 raft 协议中，写入数据的 etcd 节点必须是 Leader 节点。如果客户端提交数据到非 Leader 节点，该节点需要将请求转发到 etcd Leader 节点处理。

etcd 代码目录结构

etcd 的源代码目录结构遵循功能模块化设计，各核心组件分布在不同的目录中。以下为 etcd 的主要目录及其功能解析：

核心模块目录

```
├── 主要目录
│   ├── api
│   │   └── 包含 etcd 的 API 定义，例如 gRPC API 的接口定义。
│   ├── client
│   │   └── 包含 etcd 的客户端代码，用于与 etcd 服务器进行交互。
│   ├── etcdctl
│   │   └── 命令行工具 etcdctl 的代码。
│   ├── etcdutl
│   │   └── 包含实用工具的代码，如快照管理和碎片整理工具。
│   ├── hack
│   │   └── 包含开发过程中使用的脚本和工具，例如 CI/CD 脚本。
│   ├── pkg
│   │   └── 包含 etcd 的核心包和模块，例如 Raft 算法的实现。
│   ├── scripts
│   │   └── 包含用于构建、测试和部署的脚本。
│   ├── security
│   │   └── 包含与安全相关的代码和工具。
│   ├── server
│   │   └── 包含 etcd 服务器的核心代码，例如集群管理、Raft 日志等。
│   ├── tests
│   │   └── 包含 etcd 的测试代码，例如单元测试和集成测试。
│   ├── tools
│   │   └── 包含开发和维护过程中使用的工具代码。
```

根目录其他重要文件

```
├── 根目录其他重要文件
│   ├── README.md：项目的介绍文档，包含 etcd 的简介、特性、使用方法等。
│   ├── CONTRIBUTING.md
│   ├── LICENSE：项目的许可证文件，etcd 采用 Apache 2.0 许可证。
│   ├── etcd.conf.yml.sample：etcd 配置文件的示例。
│   ├── go.mod 和 go.sum：Go 项目的依赖管理文件。
```

服务启动流程及调用的文件

etcd 的服务启动流程从 `main.go` 开始，逐步调用多个包和文件来完成服务器的初始化和启动。以下是详细的启动流程及涉及的主要文件：

1. `main.go`:

- 位于 `/ersisted/ethome/coder/pcd/server/main.go`。
- 这是一个简单的包装器，导入了 `go.etcd.io/etcd/server/v3/etcdmain` 包，并在 `main` 函数中调用了 `etcdmain.Main(os.Args)`。

2. `etcdmain/Main`：
   - 位于 `go.etcd.io/etcd/etcdmain` 包中。
   - `Main` 函数是 etcd 的实际入口点，负责解析命令行参数、配置日志、设置信号处理等。
   - 主要调用以下函数：
     - `initEtcd`：初始化 etcd 配置。
     - `run`：根据命令行参数执行不同的子命令（如 `start`、`version` 等）。
3. `pkg/transport`：
   - 处理 gRPC 和 HTTP 传输层的配置和初始化。
   - 文件如 `transport.go`、`grpc.go` 等。
4. `server/etcdserver`：
   - 实现了 etcd 服务器的核心逻辑，包括数据存储、Raft 协议、集群管理等。
   - 文件如 `server.go`、`raft.go`、`store.go` 等。
   - `NewServer` 函数用于创建一个新的 etcd 服务器实例。
5. `server/embed`：
   - 提供了一个嵌入式 etcd 服务器的接口，允许外部程序轻松集成 etcd。
   - 文件如 `embed.go`、`config.go` 等。
   - `NewEmbedEtcd` 函数用于创建一个嵌入式 etcd 实例。
6. `client/pkg/v3`：
   - 提供客户端 API，用于与 etcd 服务器进行交互。
   - 文件如 `clientv3.go`、`kv.go`、`watch.go` 等。
7. `wal` 和 `mvcc`：
   - `wal` 包负责写前日志（Write-Ahead Logging），确保数据持久性和恢复能力。
   - `mvcc` 包实现了多版本并发控制（Multi-Version Concurrency Control），支持高效的读写操作。
   - 文件如 `wal/wal.go`、`mvcc/kvstore.go` 等。
8. `raft`：
   - 实现了 Raft 共识算法，确保分布式系统的高可用性和一致性。
   - 文件如 `raft/raft.go`、`raft/node.go` 等。

产品设计

核心设计理念

etcd 的设计遵循 CAP 定理，更倾向于 **CP（一致性 & 分区容忍性）**，保证数据的强一致性，在网络分区的情况下，etcd 会选择保持数据的一致性，而不是保证所有请求都能被立即处理，使数据仍然可靠且不丢失。

- **强一致性（Strong Consistency）**：使用 Raft 共识算法，保证任意时刻所有存活节点的数据是一致的。
- **分区容错性（Fault Tolerance）**：etcd 在面临网络分区时，会优先保证数据的一致性。这意味着在网络恢复之前，客户端可能无法访问某些功能，但系统内部的数据状态将保持一致。

简单性

etcd 设计时优先考虑简单性，体现在以下几个方面：

- **RESTful API：**支持 RESTful 风格的 HTTP+JSON API，方便使用。
- **gRPC 支持：**增加了对 gRPC 的支持，同时也提供 rest gateway 进行转化。
- **Go 语言：**使用 Go 语言编写，跨平台、部署和维护简单。

安全性

- **TLS 支持**：etcd 支持 TLS 客户端安全认证，确保数据在传输过程中的安全性。
- **权限管理**：通过权限验证机制，确保只有授权用户才能访问和修改数据，提高系统的安全性。

数据持久化

默认数据一更新就落盘持久化，数据持久化存储使用 WAL (write ahead log），预写式日志. WAL 记录了数据变化的全过程，在 etcd 中所有数据在提交之前都要先写入 WAL 中。etcd Snapshot （快照）文件则存储了某一时刻 etcd 的所有数据，默认设置为每 10 000 条记录做一次快照，经过快照后 WAL 文件即可删除。