etcd 的安装与使用

版权说明

对比特项目感兴趣，可以联系这个微信。

代码 & 板书链接

https://gitee.com/bitedu-tech/cpp-chatsystem

介绍

Etcd 是一个 golang 编写的分布式、高可用的一致性键值存储系统，用于配置共享和服务发现等。它使用 Raft 一致性算法来保持集群数据的一致性，且客户端通过长连接 watch 功能，能够及时收到数据变化通知，相较于 Zookeeper 框架更加轻量化。以下是关于 etcd 的安装与使用方法的详细介绍。

安装 Etcd

首先，需要在你的系统中安装 Etcd。Etcd 是一个分布式键值存储，通常用于服务发现和配置管理。以下是在 Linux 系统上安装 Etcd 的基本步骤：

1. 安装 Etcd：

```
Bash
sudo apt-get install etcd
```

1. 启动 Etcd 服务：

```
Bash
sudo systemctl start etcd
```

1. 设置 Etcd 开机自启：

```
Bash
sudo systemctl enable etcd
```

节点配置

如果是单节点集群其实就可以不用进行配置，默认 etcd 的集群节点通信端口为 2380，客户端访问端口为 2379.

若需要修改，则可以配置：/etc/default/etcd

```
Bash

#节点名称，默认为 "default"
ETCD_NAME="etcd1"
#数据目录，默认为 "${name}.etcd"
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
```

```
#用于客户端连接的 URL。
ETCD_LISTEN_CLIENT_URLS="http://192.168.65.132:2379,http://127.0.0
.1:2379"
#用于客户端访问的公开，也就是提供服务的 URL
ETCD_ADVERTISE_CLIENT_URLS="http://192.168.65.132:2379,http://127.
0.0.1:2379"
#用于集群节点间通信的 URL。
ETCD_LISTEN_PEER_URLS="http://192.168.65.132:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://192.168.65.132:2380"
#心跳间隔时间-毫秒
ETCD_HEARTBEAT_INTERVAL=100
#选举超时时间-毫秒
ETCD_ELECTION_TIMEOUT=1000

#以下为集群配置，若无集群则需要注销
#初始集群状态和配置--集群中所有节点
#ETCD_INITIAL_CLUSTER="etcd1=http://192.168.65.132:2380,etcd2=http
://192.168.65.132:2381,etcd3=http://192.168.65.132:2382"
#初始集群令牌-集群的 ID
#ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
#ETCD_INITIAL_CLUSTER_STATE="new"


#以下为安全配置，如果要求 SSL 连接 etcd 的话，把下面的配置启用，并修改文件
路径
#ETCD_CERT_FILE="/etc/ssl/client.pem"
#ETCD_KEY_FILE="/etc/ssl/client-key.pem"
#ETCD_CLIENT_CERT_AUTH="true"
#ETCD_TRUSTED_CA_FILE="/etc/ssl/ca.pem"
#ETCD_AUTO_TLS="true"
#ETCD_PEER_CERT_FILE="/etc/ssl/member.pem"
#ETCD_PEER_KEY_FILE="/etc/ssl/member-key.pem"
#ETCD_PEER_CLIENT_CERT_AUTH="false"
#ETCD_PEER_TRUSTED_CA_FILE="/etc/ssl/ca.pem"
#ETCD_PEER_AUTO_TLS="true"
```

单节点运行示例

```
Bash
etcd --name etcd1 --initial-advertise-peer-urls
http://192.168.65.132:2380 \
    --listen-peer-urls http://192.168.65.132:2380 \
    --listen-client-urls http://192.168.65.132:2379 \
```

```
    --advertise-client-urls http://192.168.65.132:2379 \
    --initial-cluster-token etcd-cluster \
    --initial-cluster
etcd1=http://192.168.65.132:2380,etcd2=http://192.168.65.132:2381,
etcd3=http://192.168.65.132:2382 \
    --initial-cluster-state new &> nohup1.out &
```

运行验证

```
C++
etcdctl put mykey "this is awesome"
```

如果出现报错:

```
Bash
dev@bite:~/workspace$ etcdctl put mykey "this is awesome"
No help topic for 'put'
```

则 `sudo vi /etc/profile` 在末尾声明环境变量 `ETCDCTL_API=3` 以确定 etcd 版本。

```
Bash
export ETCDCTL_API=3
```

完毕后，加载配置文件，并重新执行测试指令

```
Bash
dev@bite:~/workspace$ source /etc/profile
dev@bite:~/workspace$ etcdctl put mykey "this is awesome"
OK
dev@bite:~/workspace$ etcdctl get mykey
mykey
this is awesome
dev@bite:~/workspace$ etcdctl del mykey
```

搭建服务注册发现中心

使用 Etcd 作为服务注册发现中心，你需要定义服务的注册和发现逻辑。这通常涉及到以下几个操作:

1. **服务注册**：服务启动时，向 Etcd 注册自己的地址和端口。
2. **服务发现**：客户端通过 Etcd 获取服务的地址和端口，用于远程调用。
3. **健康检查**：服务定期向 Etcd 发送心跳，以维持其注册信息的有效性。

etcd 采用 golang 编写，v3 版本通信采用 grpc API，即(HTTP2+protobuf);

官方只维护了 go 语言版本的 client 库, 因此需要找到 C/C++ 非官方的 client 开发库:

etcd-cpp-apiv3

etcd-cpp-apiv3 是一个 etcd 的 C++版本客户端 API。它依赖于 mipsasm, boost, protobuf, gRPC, cpprestsdk 等库。

etcd-cpp-apiv3 的 GitHub 地址是：https://github.com/etcd-cpp-apiv3/etcd-cpp-apiv3

依赖安装:

```
C++
sudo apt-get install libboost-all-dev libssl-dev
sudo apt-get install libprotobuf-dev protobuf-compiler-grpc
sudo apt-get install libgrpc-dev libgrpc++-dev
sudo apt-get install libcpprest-dev
```

api 框架安装

```
Bash
git clone https://github.com/etcd-cpp-apiv3/etcd-cpp-apiv3.git
cd etcd-cpp-apiv3
mkdir build && cd build

cmake .. -DCMAKE_INSTALL_PREFIX=/usr
make -j$(nproc) && sudo make install
```

客户端类与接口介绍：

```
C++
//pplx::task 并行库异步结果对象
//阻塞方式   get(): 阻塞直到任务执行完成，并获取任务结果
//非阻塞方式 wait(): 等待任务到达终止状态，然后返回任务状态

namespace etcd {
class Value {
    bool is_dir(); //判断是否是一个目录
    std::string const& key() //键值对的 key 值
    std::string const& as_string()//键值对的 val 值

    int64_t lease() //用于创建租约的响应中，返回租约 ID
}
```

```
//etcd 会监控所管理的数据的变化，一旦数据产生变化会通知客户端
//在通知客户端的时候，会返回改变前的数据和改变后的数据
class Event {
    enum class EventType {
        PUT, //键值对新增或数据发生改变
        DELETE_,//键值对被删除
        INVALID,
    };
    enum EventType event_type()
    const Value& kv()
    const Value& prev_kv()
}
class Response {
    bool is_ok()
    std::string const& error_message()
    Value const& value()//当前的数值 或者 一个请求的处理结果
    Value const& prev_value()//之前的数值
    Value const& value(int index)//
    std::vector<Event> const& events();//触发的事件
}

class KeepAlive {
    KeepAlive(Client const& client, int ttl, int64_t lease_id =
0);
    //返回租约 ID
    int64_t Lease();
    //停止保活动作
    void Cancel();
}

class Client {
    // etcd_url:  "http://127.0.0.1:2379"
    Client(std::string const& etcd_url,
        std::string const& load_balancer = "round_robin");
    //Put a new key-value pair 新增一个键值对
    pplx::task<Response> put(std::string const& key,
        std::string const& value);
    //新增带有租约的键值对 （一定时间后，如果没有续租，数据自动删除）
    pplx::task<Response> put(std::string const& key,
        std::string const& value,
        const int64_t leaseId);
    //获取一个指定 key 目录下的数据列表
    pplx::task<Response> ls(std::string const& key);
```

```
    //创建并获取一个存活 ttl 时间的租约
    pplx::task<Response> leasegrant(int ttl);
    //获取一个租约保活对象，其参数 ttl 表示租约有效时间
    pplx::task<std::shared_ptr<KeepAlive>> leasekeepalive(int
ttl);
    //撤销一个指定的租约
    pplx::task<Response> leaserevoke(int64_t lease_id);
    //数据锁
    pplx::task<Response> lock(std::string const& key);
}

class Watcher {
    Watcher(Client const& client,
        std::string const& key, //要监控的键值对 key
        std::function<void(Response)> callback, //发生改变后的回调
        bool recursive = false); //是否递归监控目录下的所有数据改变
    Watcher(std::string const& address,
        std::string const& key,
        std::function<void(Response)> callback,
        bool recursive = false);
    //阻塞等待，直到监控任务被停止
    bool Wait();
    bool Cancel();
}
```

使用样例：

registry.cc

```
C++
#include <etcd/Client.hpp>
#include <etcd/Response.hpp>
#include <etcd/KeepAlive.hpp>
#include <thread>

int main() {
    std::string registry_host = "http://127.0.0.1:2379";
    //为了防止多主机注册相同服务时，信息覆盖，
    //因此每个主机在服务名后加入自己的实例名称，相当于各有各的服务-主机
键值对
    std::string service_key = "/service/user/instance";
    std::string service_host = "112.23.23.120:9090";
    etcd::Client etcd(registry_host);
```

```
    // 对客户端创建一个指定时长的租约，若租约到期则创建的键值将被撤销（）
    // 通过租约进行保活探测，若服务提供端掉线，则租约到期后 etcd 会给服务发现者发送服务下线通知。
    // etcd::Response resp = etcd.leasegrant(3).get();  //设置一个 3s 的租约
    // auto lease_id = resp.value().lease(); // 获取租约 ID
    //创建客户端的同时，创建一个保活的 3s 租约保活对象--也就是一旦断开连接，3s 后租约失效
    std::shared_ptr<etcd::KeepAlive> keepalive = etcd.leasekeepalive(3).get();
    auto lease_id = keepalive->Lease();
    auto resp_task = etcd.put(service_key, service_host, lease_id); //注册服务
    auto resp = resp_task.get();
    if (resp.is_ok() == false) {
        std::cout << resp.error_message() << std::endl;
        return -1;
    }
    std::cout << "添加数据成功！" << std::endl;
    getchar();//回车后撤销租约
    // //租约的撤销
    etcd.leaserevoke(lease_id);


    return 0;
}
```

discovery.cc

```
C++
#include <etcd/Client.hpp>
#include <etcd/Watcher.hpp>

void watcherCallback(etcd::Response const& resp) {
    if (resp.error_code()) {
        std::cout << "Watcher Error:" << resp.error_code();
        std::cout << "-" << resp.error_message() << std::endl;
    } else {
            if (ev.event_type() == etcd::Event::EventType::PUT) {
                //如果是新增值，则通过当前值来查看新增的是什么值
                std::cout << "服务" << ev.kv().key() << "新增主机：";
```

```
                std::cout << ev.kv().as_string() << std::endl;
            }else if (ev.event_type() ==
etcd::Event::EventType::DELETE_) {
                //如果是值被删除，则需要通过发生事件之前的值来了解哪个
值被删除了
                std::cout << "服务" << ev.kv().key() << "下线主机：
";
                std::cout << ev.prev_kv().as_string() <<
std::endl;
            }
        }
    }

}

int main() {
    std::string registry_host = "http://127.0.0.1:2379";
    std::string service_key = "/service/user/instance";
    etcd::Client etcd(registry_host);
    //初次先用 ls 获取所有能够提供指定服务的实例信息
    etcd::Response resp = etcd.ls(service_key).get();
    if (resp.is_ok()){
        for (int i = 0; i < resp.keys().size(); i++) {
            std::cout << resp.key(i) << "=" <<
resp.value(i).as_string() << std::endl;
        }
    }else
        std::cout << "Get Service Error:" << resp.error_code();
        std::cout << "-" << resp.error_message() << std::endl;
    //获取之后，然后定义监控对象，监控目录内容变化，通过目录变化来感知服
务的上线与下线
    etcd::Watcher watcher(registry_host, service_key,
watcherCallback, true);
    getchar();
    watcher.Cancel();

    return 0;
}
```

Makefile

```
Makefile
all: registry discoverer
registry: registry.cc
```

```
    g++ -std=c++17 $^ -o $@ -letcd-cpp-api -lcpprest
discoverer: discoverer.cc
    g++ -std=c++17 $^ -o $@ -letcd-cpp-api -lcpprest
clean:
    rm -rf registry discoverer
```

以上的内容只是一个非常简单的样例，在真正的服务发现实现中，服务发现端，通常需要维护一张服务信息表，根据监控到的服务上线下线事件，对表中数据进行操作。

封装服务发现与注册功能：

在服务的注册与发现中，主要基于 etcd 所提供的可以设置有效时间的键值对存储来实现。

服务注册

主要是在 etcd 服务器上存储一个租期 ns 的保活键值对，表示所能提供指定服务的节点主机，比如 /service/user/instance-1 的 key，且对应的 val 为提供服务的主机节点地址：

<key, val> -- < /service/user/instance-1, 127.0.0.1:9000>

- /service 是主目录，其下会有不同服务的键值对存储
- /user 是服务名称，表示该键值对是一个用户服务的节点
- /instance-1 是节点实例名称，提供用户服务可能会有很多节点，每个节点都应该有自己独立且唯一的实例名称

当这个键值对注册之后，服务发现方可以基于目录进行键值对的发现。

且一旦注册节点退出，保活失败，则 3s 后租约失效，键值对被删除，etcd 会通知发现方数据的失效，进而实现服务下线通知的功能。

服务发现

服务发现分为两个过程：

- 刚启动客户端的时候，进行 ls 目录浏览，进行/service 路径下所有键值对的获取
- 对关心的服务进行 watcher 观测，一旦数值发生变化（新增/删除），收到通知进行节点的管理

如果 ls 的路径为/service，则会获取到 /service/user, /service/firend, ...等其路径下的所有能够提供服务的实例节点数据。

如果 ls 的路径为 /service/user， 则会获取到 /service/user/instancd-1, /service/user/instance-2,....等所有提供用户服务的实例节点数据。

客户端可以将发现的所有<实例 - 地址>管理起来，以便于进行节点的管理：

- 收到新增数据通知，则向本地管理添加新增的节点地址 -- 服务上线
- 收到删除数据通知，则从本地管理删除对应的节点地址 -- 服务下线

因为管理了所有的能够提供服务的节点主机的地址，因此当需要进行 rpc 调用的时候，则根据服务名称，获取一个能够提供服务的主机节点地址进行访问就可以了，而这里的获取策略，我们采用 RR 轮转策略。

封装思想：

将 etcd 的操作全部封装起来，也不需要管理数据，只需要向外四个基础操作接口：

- 进行服务注册，也就是向 etcd 添加 <服务-主机地址>的数据
- 进行服务发现，获取当前所有能提供服务的信息
- 设置服务上线的处理回调接口
- 设置服务下线的处理回调接口

这样封装之后，外部的 rpc 调用模块，可以先获取所有的当前服务信息，建立通信连接进行 rpc 调用，也能在有新服务上线的时候新增连接，以及下线的时候移除连接。