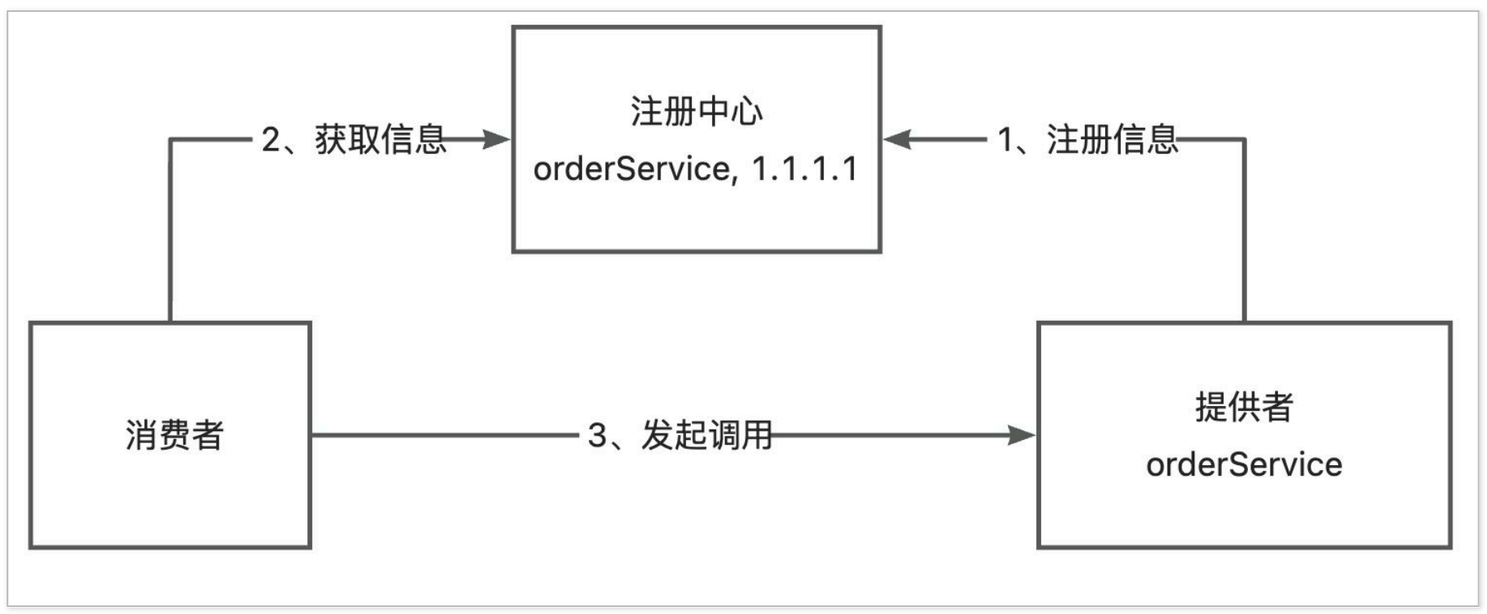
5. 注册中心基本实现 - 手写 RPC 框架项目教程 - 编程导航教程

> 仅供 [编程导航](https://www.code-nav.cn/post/1816420035119853569) 内部成员观看，请勿对外分享！一、需求分析 RPC 框架的一个核心模块是注册中心，目。

一、需求分析

RPC 框架的一个核心模块是注册中心，目的是帮助服务消费者获取到服务提供者的调用地址，而不是将调用地址硬编码到项目中。1747226499385180161_0.4393812930240619

在第一节教程中，分享过这样一张图片，注册中心的作用一目了然：



本节教程，我们先来实现一个具有基本功能的注册中心，跑通注册中心的流程，之后再优化。
1747226499385180161_0.9748205756903816

二、设计方案

注册中心核心能力

我们先明确注册中心的几个实现关键（核心能力）：1747226499385180161_0.06533856377949432

1. 数据分布式存储：集中的注册信息数据存储、读取和共享
2. 服务注册：服务提供者上报服务信息到注册中心
3. 服务发现：服务消费者从注册中心拉取服务信息 1747226499385180161_0.4192720066690607
4. 心跳检测：定期检查服务提供者的存活状态
5. 服务注销：手动剔除节点、或者自动剔除失效节点
6. 更多优化点：比如注册中心本身的容错、服务消费者缓存等。1747226499385180161_0.6376282986398707

技术选型

明确了注册中心的核心能力后，我们可以根据这些能力进行技术选型。1747226499385180161_0.7056319102609643

第一点是最重要的，我们首先需要一个能够集中存储和读取数据的中间件。此外，它还需要有数据过期、数据监听的能力，便于我们移除失效节点、更新节点列表等。

此外，对于注册中心的技术选型，我们还要考虑它的性能、高可用性、高可靠性、稳定性、数据一致性、社区的生态和活跃度等。注册中心的可用性和可靠性尤其重要，因为一旦注册中心本身都挂了，会影响到所有服务的调用。

主流的注册中心实现中间件有 ZooKeeper、Redis 等。在鱼皮的 RPC 框架教程中，会带大家使用一种更新颖的、更适合存储元信息（注册信息）的云原生中间件 Etcd，来实现注册中心。1747226499385180161_0.6818531421344562

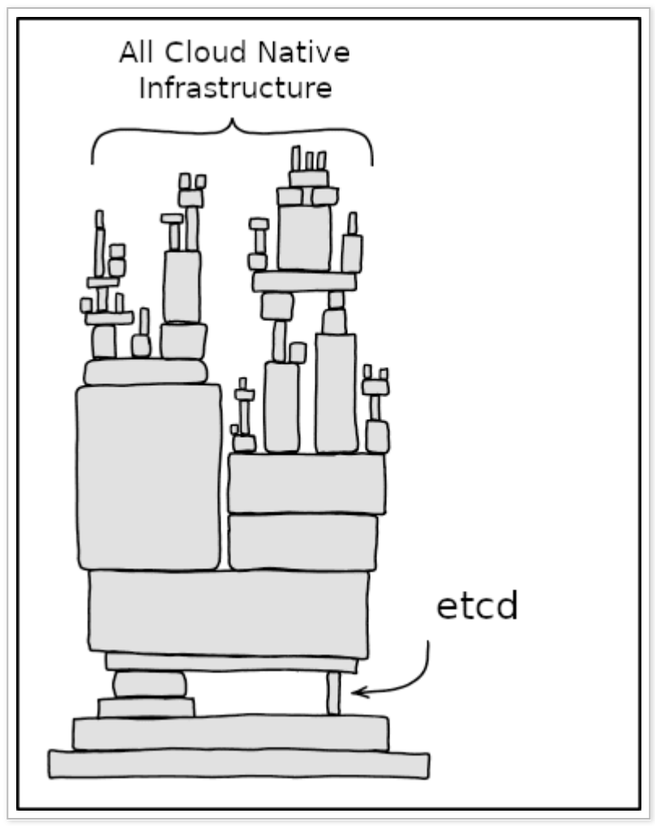
Etcd 入门

Etcd 介绍

GitHub： https://github.com/etcd-io/etcd 1747226499385180161_0.7901431100608183

Etcd 是一个 Go 语言实现的、开源的、分布式 的键值存储系统，它主要用于分布式系统中的服务发现、配置管理和分布式锁等场景。

提到 Go 语言实现，有经验的同学应该就能想到，Etcd 的性能是很高的，而且它和云原生有着密切的关系，通常被作为云原生应用的基础设施，存储一些元信息。比如经典的容器管理平台 k8s 就使用了 Etcd 来存储集群配置信息、状态信息、节点信息等。



除了性能之外，Etcd 采用 Raft 一致性算法来保证数据的一致性和可靠性，具有高可用性、强一致性、分布式特性等特点。

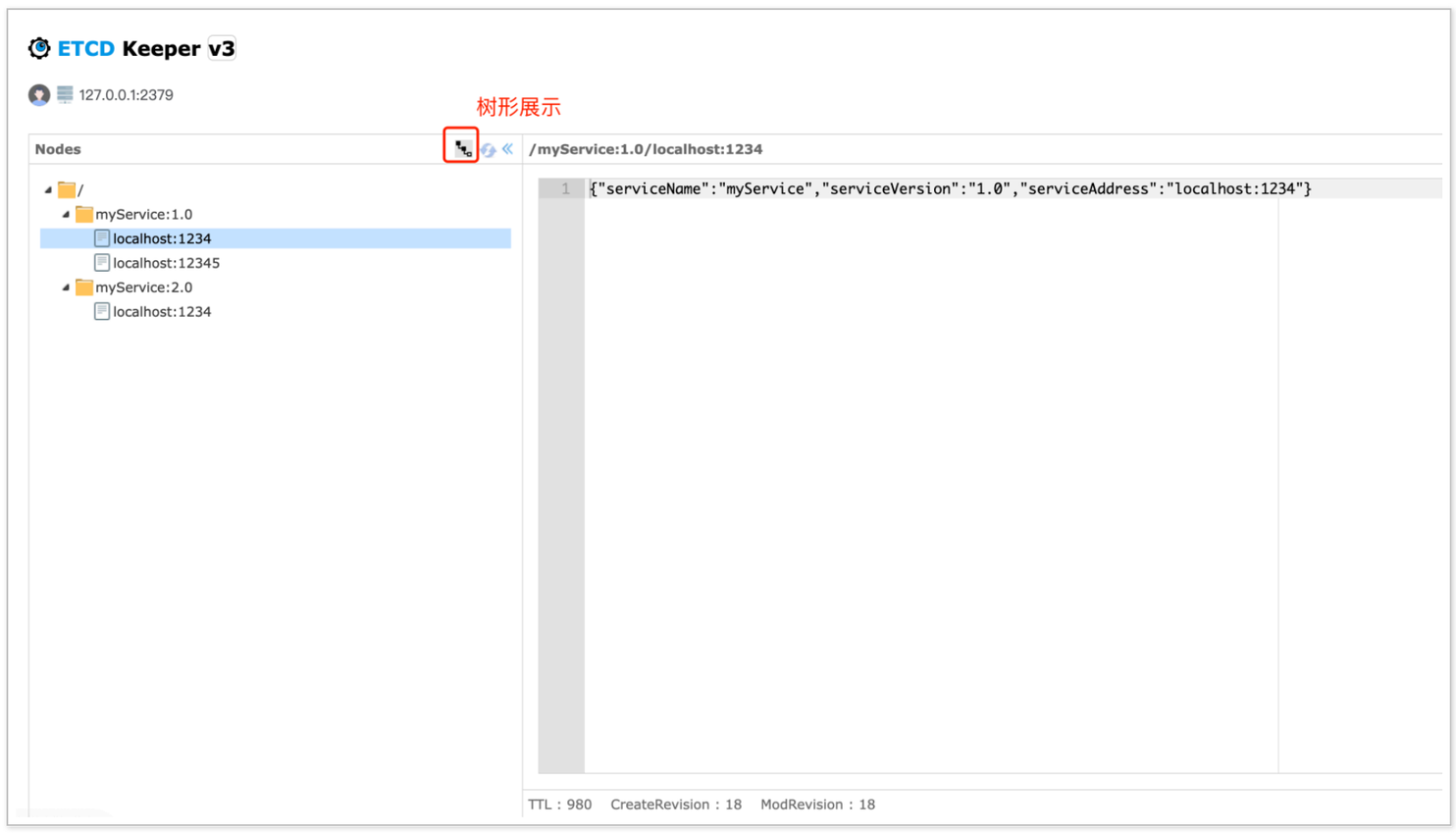
最可气的是，Etcd 还非常简单易用！提供了简单的 API、数据的过期机制、数据的监听和通知机制等，完美满足注册中心的实现诉求。

Etcd 的入门成本是极低的，只要你学过 Redis、ZooKeeper 或者对象存储中的一个，就能够很快理解 Etcd 并投入实战运用。我们学知识的一个技巧，就是把新知识和老知识进行类比和关联。1747226499385180161_0.9057574882548884

Etcd 数据结构与特性

Etcd 在其数据模型和组织结构上更接近于 ZooKeeper 和对象存储，而不是 Redis。它使用层次化的键值对来存储数据，支持类似于文件系统路径的层次结构，能够很灵活地单 key 查询、按前缀查询、按范围查询。

如下图：1747226499385180161_0.6567621853772561



Etcd 的核心数据结构包括：

1. Key（键）：Etcd 中的基本数据单元，类似于文件系统中的文件名。每个键都唯一标识一个值，并且可以包含子键，形成类似于路径的层次结构。

2. Value（值）：与键关联的数据，可以是任意类型的数据，通常是字符串形式。

1747226499385180161_0.8281299602210546

只有 key、value，是不是比 Redis 好理解多了？我们可以将数据序列化后写入 value。

Etcd 有很多核心特性，其中，应用较多的特性是：

1. Lease（租约）：用于对键值对进行 TTL 超时设置，即设置键值对的过期时间。当租约过期时，相关的键值对将被自动删除。

2. Watch（监听）：可以监视特定键的变化，当键的值发生变化时，会触发相应的通知。

1747226499385180161_0.7118341671798556

有了这些特性，我们就能够实现注册中心的服务提供者节点过期和监听了。

此外，Etcd 的一大优势就是能够保证数据的强一致性。

Etcd 如何保证数据一致性?

从表层来看，Etcd 支持事务操作，能够保证数据一致性。

从底层来看，Etcd 使用 Raft 一致性算法来保证数据的一致性。

Raft 是一种分布式一致性算法，它确保了分布式系统中的所有节点在任何时间点都能达成一致的数据视图。
1747226499385180161_0.25419987885402273

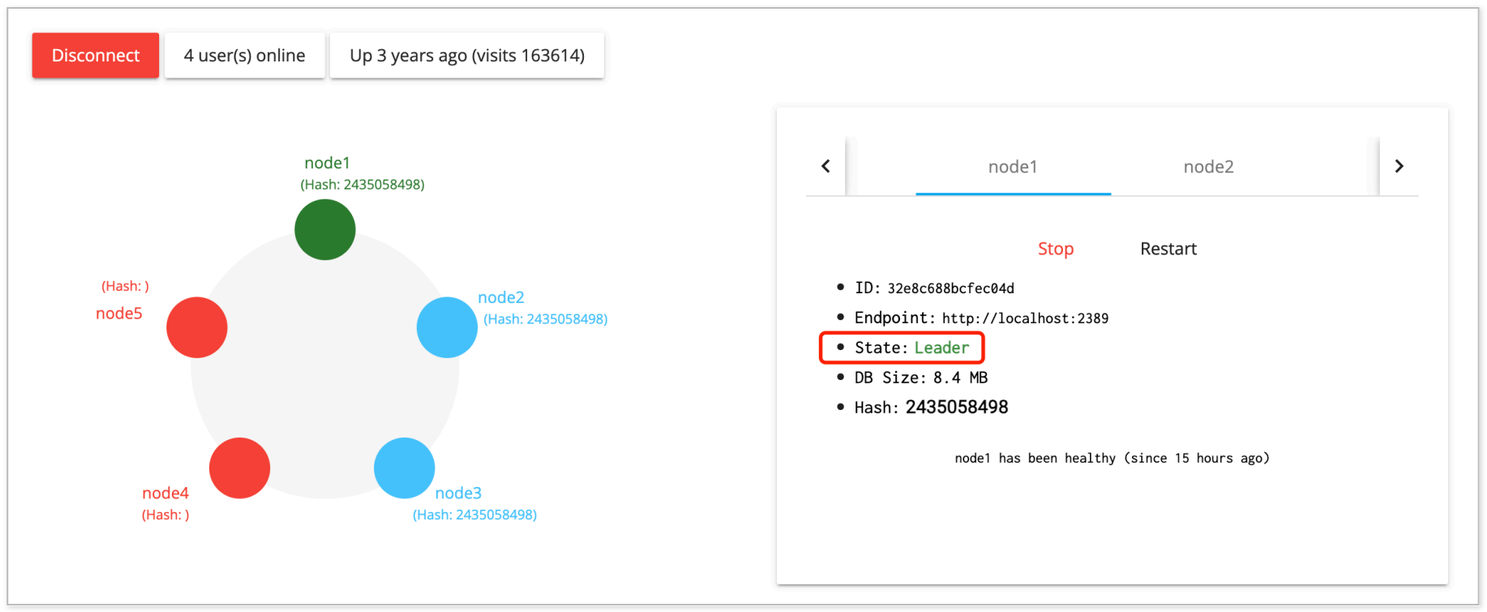
具体来说，Raft 算法通过选举机制选举出一个领导者（Leader）节点，领导者负责接收客户端的写请求，并将写操作复制到其他节点上。当客户端发送写请求时，领导者首先将写操作写入自己的日志中，并将写操作的日志条目分发给其他节点，其他节点收到日志后也将其写入自己的日志中。一旦 大多数节点（即半数以上的节点）都将该日志条目成功写入到自己的日志中，该日志条目就被视为已提交，领导者会向客户端发送成功响应。在领导者发送成功响应后，该写操作就被视为已提交，从而保证了数据的一致性。

如果领导者节点宕机或失去联系，Raft 算法会在其他节点中 选举出新的领导者，从而保证系统的可用性和一致性。新的领导者会继续接收客户端的写请求，并负责将写操作复制到其他节点上，从而保持数据的一致性。

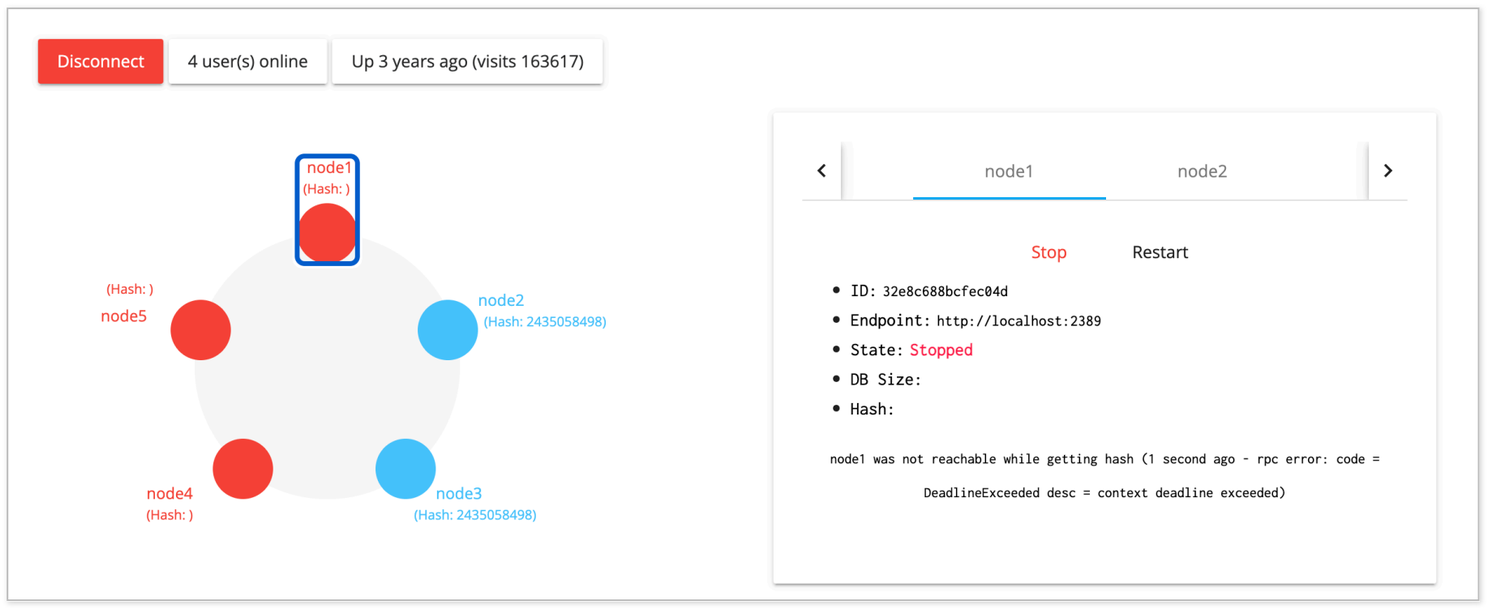
上面这段不理解也没关系，我们可以使用官方提供的 Etcd Playground 来可视化操作 Etcd，便于学习。
1747226499385180161_0.7828381356113294

Playground 地址： http://play.etcd.io/play

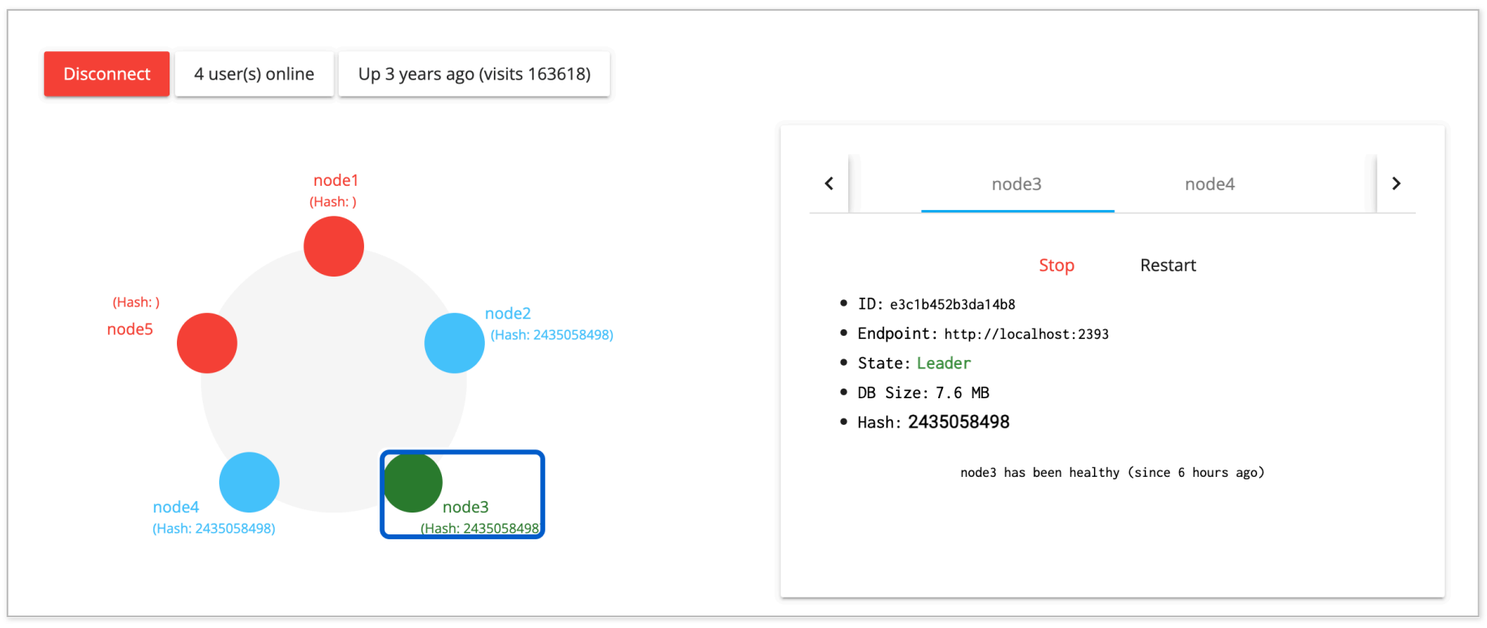
比如我们可以尝试停止主节点，其余节点为从节点：



然后会发现主节点挂掉后，并没有新的从节点成为主节点，因为还剩 2 个节点，一人一票，谁都不服谁！这种现象也称为“脑裂”。



然后我们启动 node4，会发现 node3 成为了主节点，因为 3 个节点，不会出现选举主节点时的平票情况。
1747226499385180161_0.5159652295644908

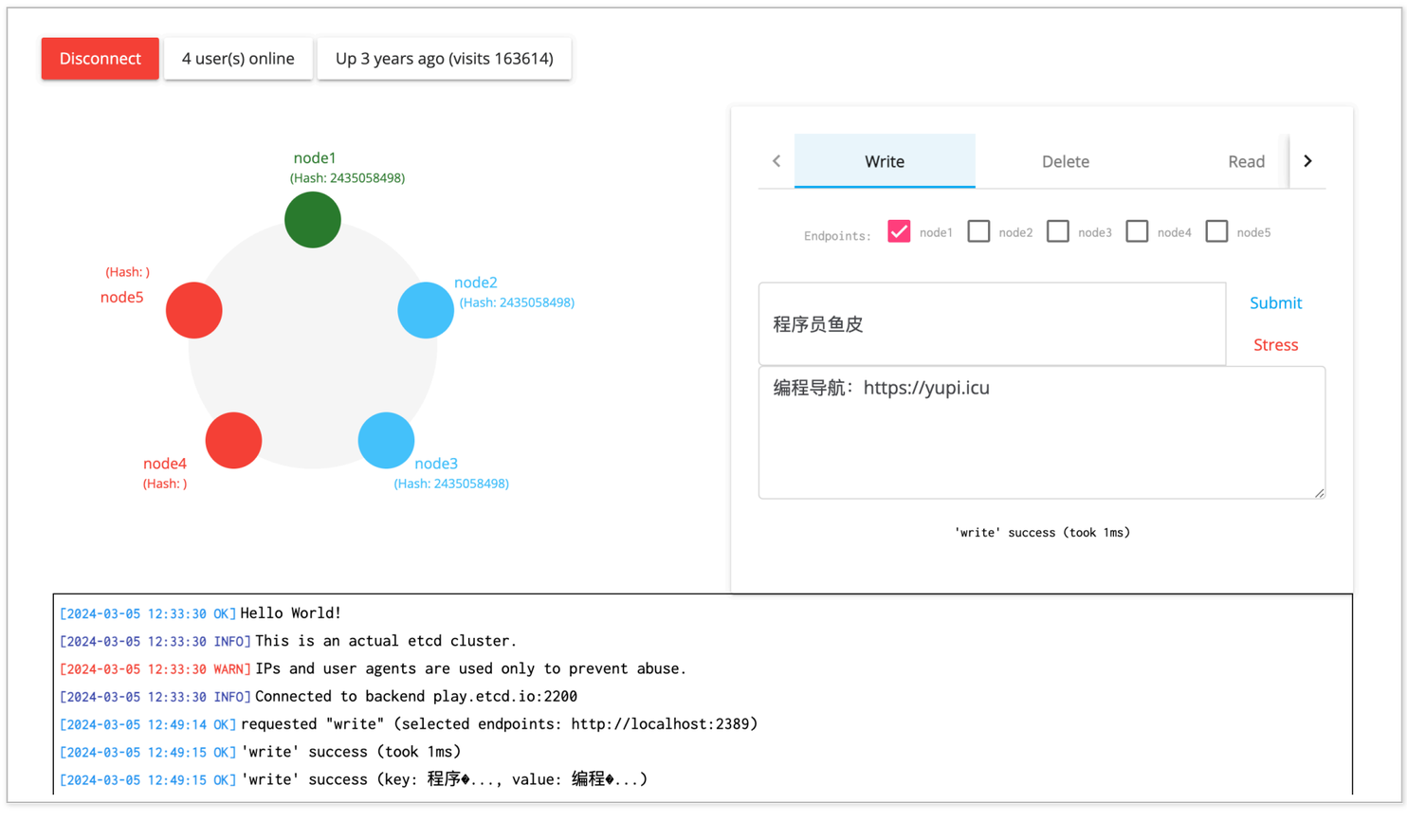


大概就是这样，初学者没必要深究背后的算法了，能理解大致流程即可。鱼皮当时专门看过华为研究 Etcd 的书籍，属实是看完就忘。

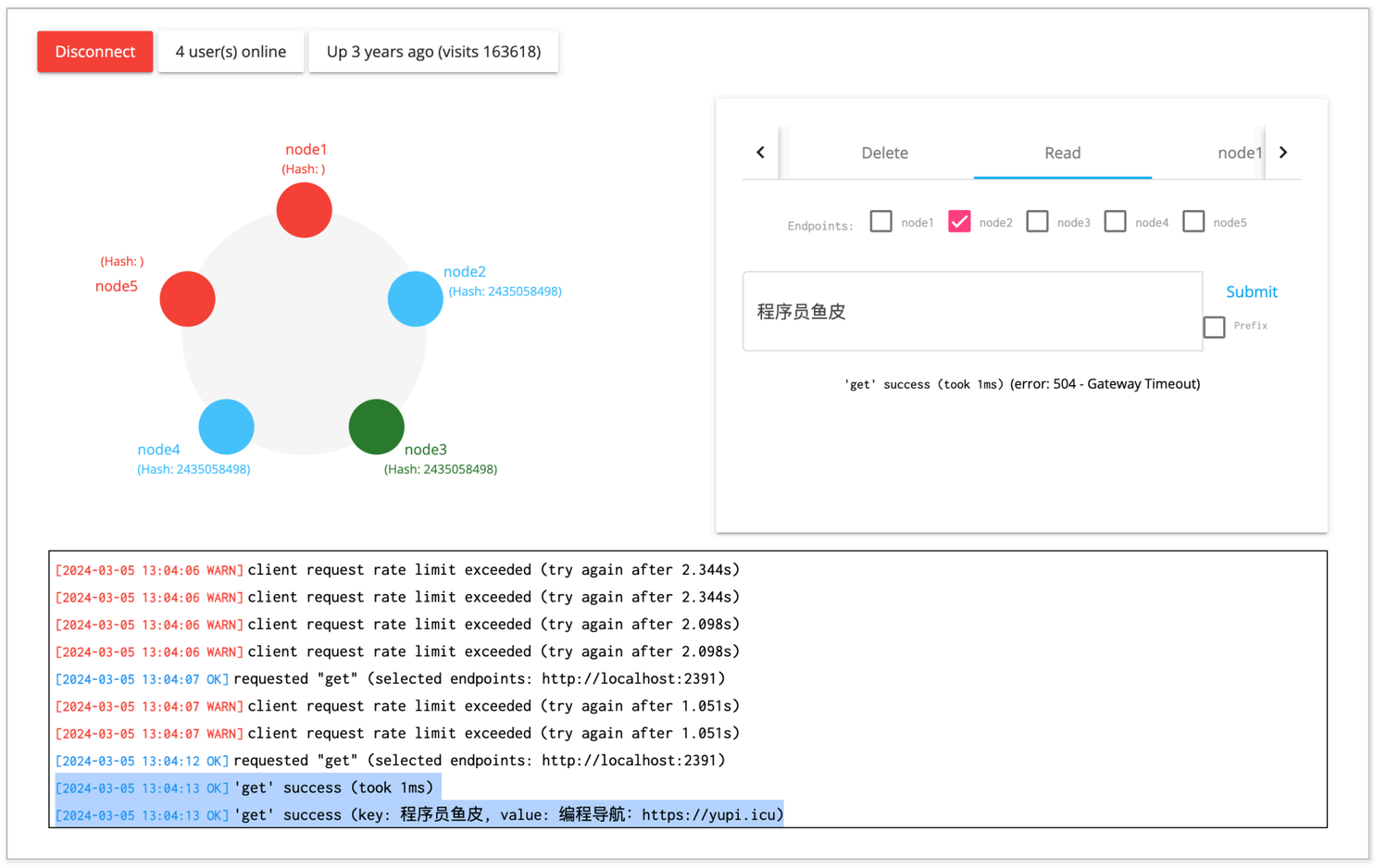
Etcd 基本操作

和所有数据存储中间件一样，基本操作无非就是：增删改查。

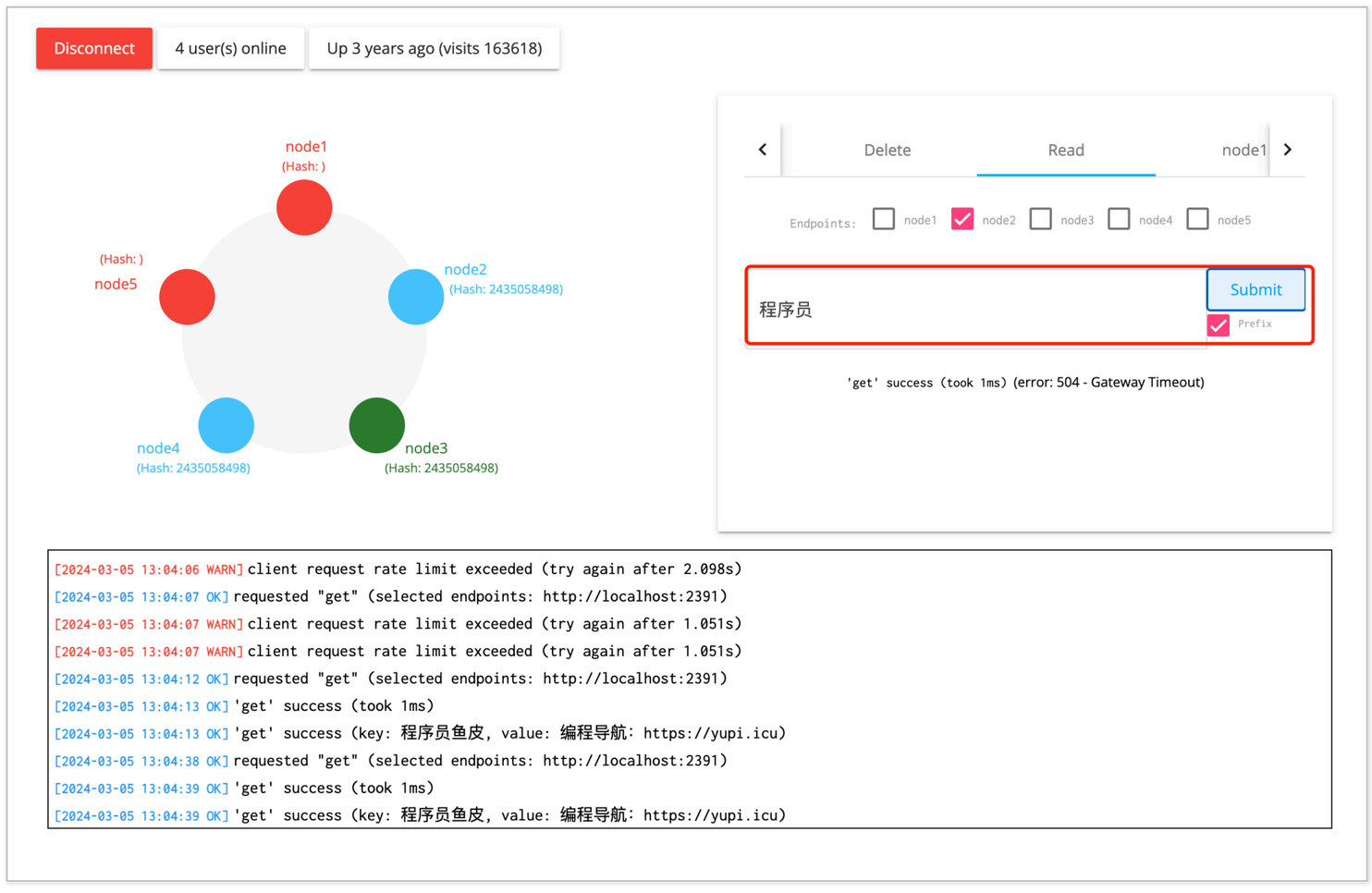
可以用可视化界面模拟操作，比如 write 写数据（更新数据）：



然后读取数据：



还支持根据前缀搜索数据：1747226499385180161_0.7440277272960969



还有一些其他操作，比如租约、监听等，后续会给大家演示。

Etcd 安装

进入 Etcd 官方的下载页： https://github.com/etcd-io/etcd/releases

也可以在这里下载： https://etcd.io/docs/v3.2/install/

找到自己操作系统的版本执行即可：1747226499385180161_0.5445503820836812

macOS (Darwin)

```
ETCD_VER=v3.5.12

# choose either URL
GOOGLE_URL=https://storage.googleapis.com/etcd
GITHUB_URL=https://github.com/etcd-io/etcd/releases/download
DOWNLOAD_URL=${GOOGLE_URL}

rm -f /tmp/etcd-${ETCD_VER}-darwin-amd64.zip
rm -rf /tmp/etcd-download-test && mkdir -p /tmp/etcd-download-test

curl -L ${DOWNLOAD_URL}/${ETCD_VER}/etcd-${ETCD_VER}-darwin-amd64.zip -o /tmp/etcd-${ETCD_VER}-darwin-amd
unzip /tmp/etcd-${ETCD_VER}-darwin-amd64.zip -d /tmp && rm -f /tmp/etcd-${ETCD_VER}-darwin-amd64.zip
mv /tmp/etcd-${ETCD_VER}-darwin-amd64/* /tmp/etcd-download-test && rm -rf mv /tmp/etcd-${ETCD_VER}-darwin-

/tmp/etcd-download-test/etcd --version
/tmp/etcd-download-test/etcdctl version
/tmp/etcd-download-test/etcdutl version
```

安装完成后，会得到 3 个脚本：

- etcd：etcd 服务本身
- etcdctl：客户端，用于操作 etcd，比如读写数据
- etcdutl：备份恢复工具 1747226499385180161_0.1919146795961959

  1747226499385180161_0.7723891138991996

执行 etcd 脚本后，可以启动 etcd 服务，服务默认占用 2379 和 2380 端口，作用分别如下：

- 2379：提供 HTTP API 服务，和 etcdctl 交互
- 2380：集群中节点间通讯

```
yupi@192 ~ % lsof -i:2379
COMMAND   PID USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
etcd    44784 yupi    9u  IPv4 0x391a4966ba65e6b9      0t0  TCP localhost:2379 (LISTEN)
etcd    44784 yupi   16u  IPv4 0x391a4966b8c766b9      0t0  TCP localhost:55298->localhost:2379 (ESTABLISHED)
etcd    44784 yupi   17u  IPv4 0x391a4966bd439e09      0t0  TCP localhost:2379->localhost:55298 (ESTABLISHED)
yupi@192 ~ % lsof -i:2380
COMMAND   PID USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
etcd    44784 yupi    5u  IPv4 0x391a4966bb5a5e09      0t0  TCP localhost:2380 (LISTEN)
```

Etcd 可视化工具

一般情况下，我们使用数据存储中间件时，一定要有一个可视化工具，能够更直观清晰地管理已经存储的数据。比如 Redis 的 Redis Desktop Manager。

同样的，Etcd 也有一些可视化工具，比如：1747226499385180161_0.16135774451844886

- etcdkeeper： https://github.com/evildecay/etcdkeeper/
- kstone： https://github.com/kstone-io/kstone/tree/master/charts

更推荐 etcdkeeper，安装成本更低，学习使用更方便。

进入项目的 GitHub，就能看到安装方式，直接按照指引下载、解压、运行脚本即可：
1747226499385180161_0.9393638505419448

Usage

- Run etcdkeeper.exe (windows version)
- Run etcdkeeper.exe -auth (If enable etcd authentication)
- Download other platform releases.

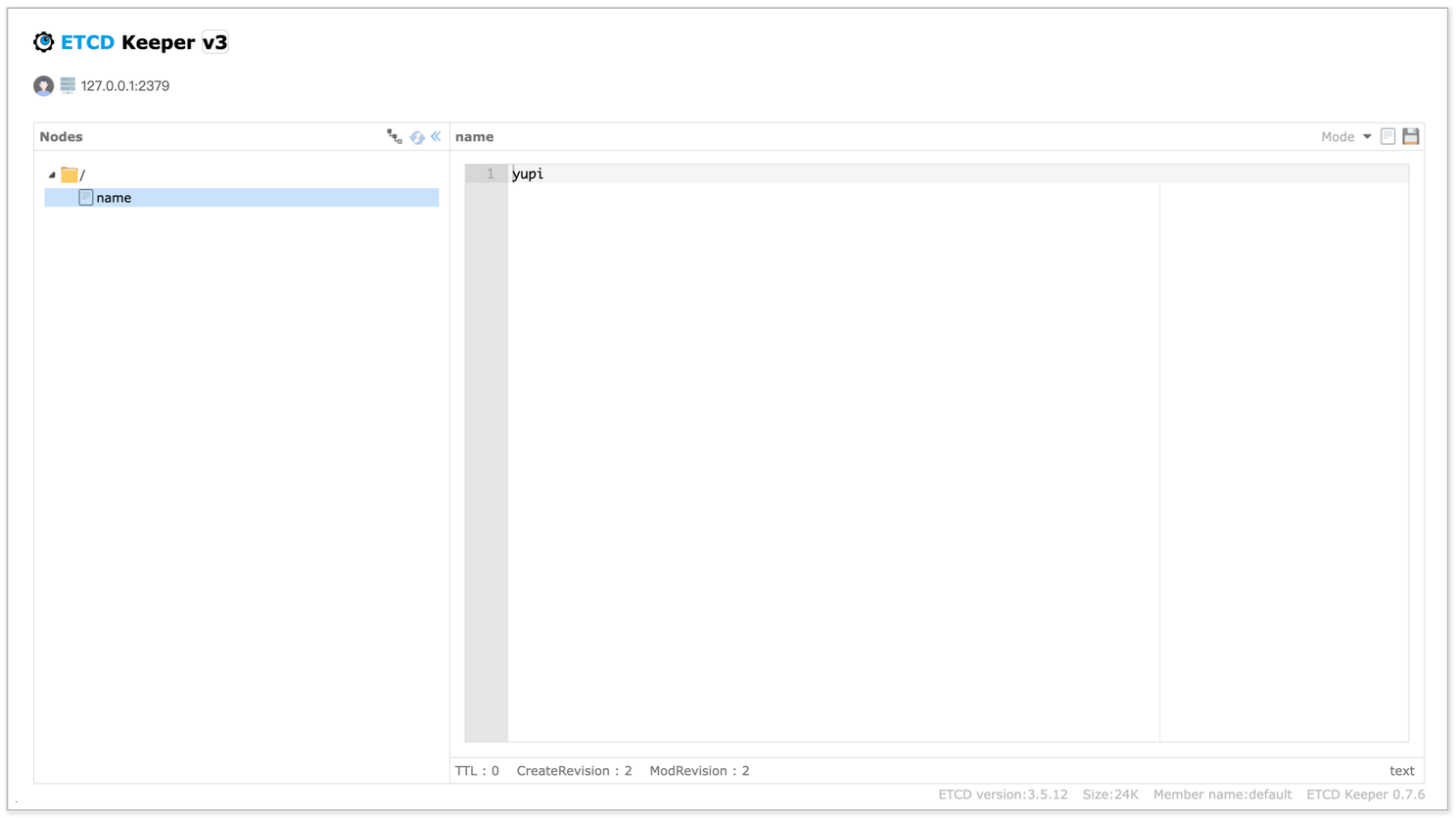
```
Usage of etcdkeeper.exe:
-h string
      host name or ip address (default: "0.0.0.0", the http server addreess, not etcd addres
-p int
      port (default 8080)
-sep string
      Separator (default "/")
-usetls
      use tls (only v3)
-cacert string
      verify certificates of TLS-enabled secure servers using this CA bundle (only v3)
-cert string
      identify secure client using this TLS certificate file (only v3)
-key string
      identify secure client using this TLS key file (only v3)
-auth bool
      use etcd auth
-timeout int
      ETCD client connect timeout
```

- Open your browser and enter the address: http://127.0.0.1:8080/etcdkeeper

安装后，执行命令，可以在指定端口启动可视化界面（默认是 8080 端口），比如鱼皮在 8081 端口启动。

```
./etcdkeeper -p 8081
```

安装后，访问本地 http://127.0.0.1:8081/etcdkeeper/，就能看到可视化页面了，如图：



Etcd Java 客户端

所谓客户端，就是操作 Etcd 的工具。

etcd 主流的 Java 客户端是 jetcd： https://github.com/etcd-io/jetcd。

注意，Java 版本必须大于 11！1747226499385180161_0.6734315703269387

用法非常简单，就像 curator 能够操作 ZooKeeper、jedis 能够操作 Redis 一样。

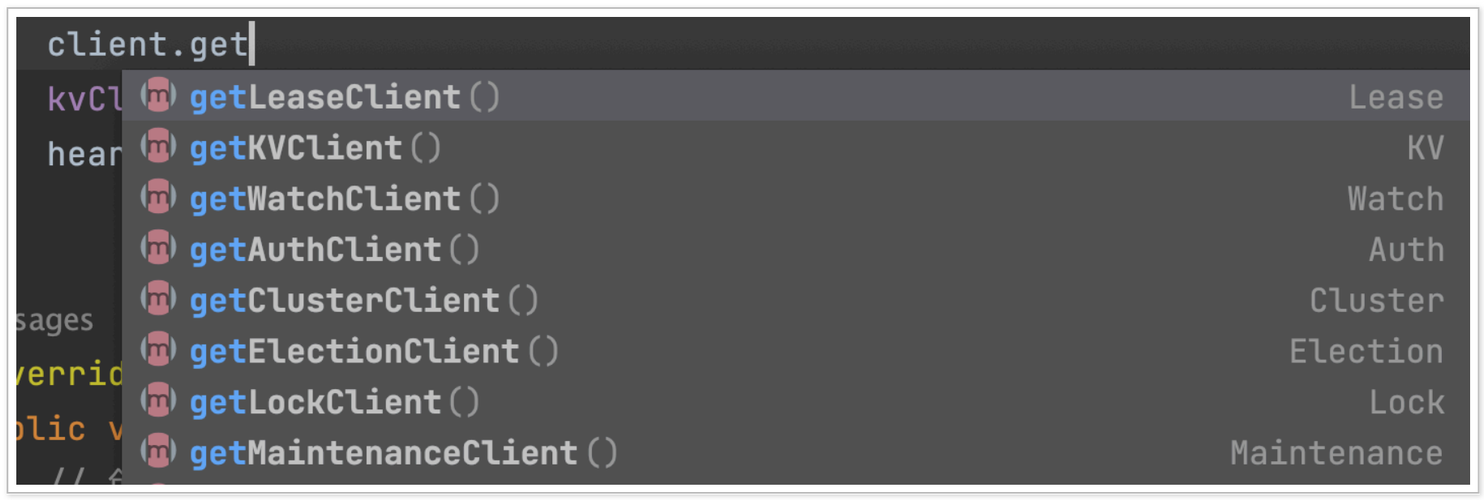
1）首先在项目中引入 jetcd：

```
<!-- https://mvnrepository.com/artifact/io.etcd/jetcd-core -->
<dependency>
    <groupId>io.etcd</groupId>
    <artifactId>jetcd-core</artifactId>
    <version>0.7.7</version>
</dependency>
```

2）按照官方文档的示例写 Demo：

```
package com.yupi.yurpc.registry;

import io.etcd.jetcd.ByteSequence;
import io.etcd.jetcd.Client;
import io.etcd.jetcd.KV;
import io.etcd.jetcd.kv.GetResponse;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class EtcdRegistry {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        // create client using endpoints
        Client client = Client.builder().endpoints("http://localhost:2379")
                .build();

        KV kvClient = client.getKVClient();
        ByteSequence key = ByteSequence.from("test_key".getBytes());
        ByteSequence value = ByteSequence.from("test_value".getBytes());

        // put the key-value
        kvClient.put(key, value).get();

        // get the CompletableFuture
        CompletableFuture<GetResponse> getFuture = kvClient.get(key);

        // get the value from CompletableFuture
        GetResponse response = getFuture.get();

        // delete the key
        kvClient.delete(key).get();
    }
}
```

在上述代码中，我们使用 KVClient 来操作 etcd 写入和读取数据。除了 KVClient 客户端外，Etcd 还提供了很多其他客户端。
1747226499385180161_0.7381995771551089



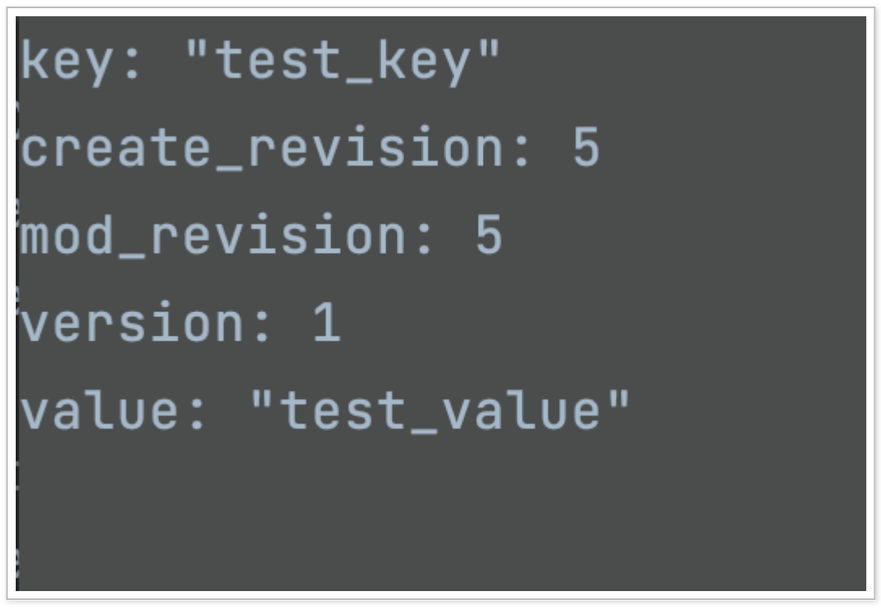
常用的客户端和作用如下，仅作了解即可：

1. kvClient：用于对 etcd 中的键值对进行操作。通过 kvClient 可以进行设置值、获取值、删除值、列出目录等操作。
2. leaseClient：用于管理 etcd 的租约机制。租约是 etcd 中的一种时间片，用于为键值对分配生存时间，并在租约到期时自动删除相关的键值对。通过 leaseClient 可以创建、获取、续约和撤销租约。
3. watchClient：用于监视 etcd 中键的变化，并在键的值发生变化时接收通知。
   1747226499385180161_0.8329040732453219
4. clusterClient：用于与 etcd 集群进行交互，包括添加、移除、列出成员、设置选举、获取集群的健康状态、获取成员列表信息等操作。
5. authClient：用于管理 etcd 的身份验证和授权。通过 authClient 可以添加、删除、列出用户、角色等身份信息，以及授予或撤销用户或角色的权限。
6. maintenanceClient：用于执行 etcd 的维护操作，如健康检查、数据库备份、成员维护、数据库快照、数据库压缩等。
   1747226499385180161_0.1495333088177091
7. lockClient：用于实现分布式锁功能，通过 lockClient 可以在 etcd 上创建、获取、释放锁，能够轻松实现并发控制。
8. electionClient：用于实现分布式选举功能，可以在 etcd 上创建选举、提交选票、监视选举结果等。

1747226499385180161_0.5937980913981593

绝大多数情况下，用前 3 个客户端就足够了。

3）使用 Debug 执行上述代码，观察 Etcd 的数据结构，如图：



发现除了 key 和 value 外，还能看到版本、创建版本、修改版本字段。这是因为 etcd 中的每个键都有一个与之关联的版本号，用于跟踪键的修改历史。当一个键的值发生变化时，其版本号也会增加。

通过使用 etcd 的 Watch API，可以监视键的变化，并在发生变化时接收通知。这种版本机制使得 etcd 在分布式系统中能够实现乐观并发控制、一致性和可靠性的数据访问。

OK，了解了 Etcd 的基础用法后，我们还要设计服务注册信息如何存储在注册中心内。

存储结构设计

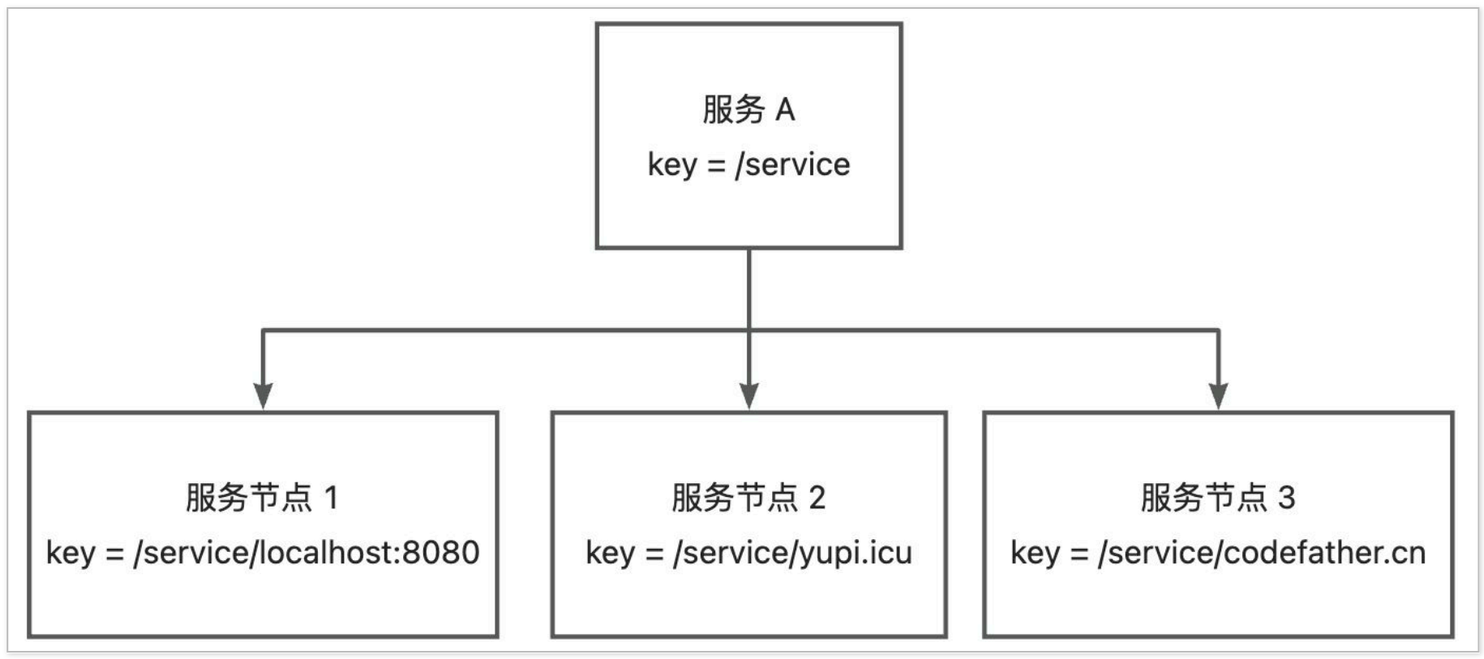
存储结构设计的几个要点：1747226499385180161_0.8943237696314603

1. key 如何设计?
2. value 如何设计?
3. key 什么时候过期？1747226499385180161_0.6919543363288998

由于一个服务可能有多个服务提供者（负载均衡），我们可以有两种结构设计：

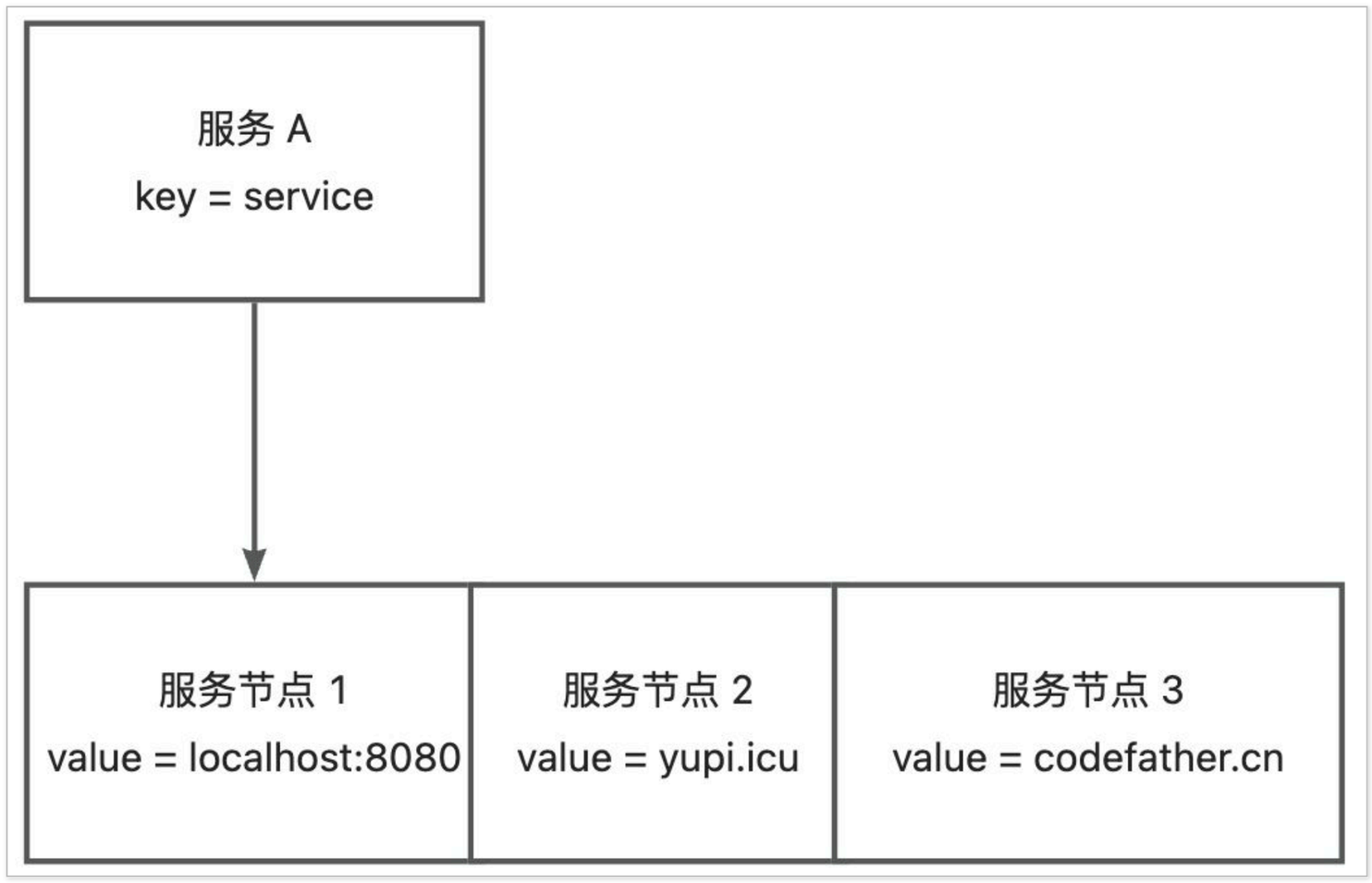
1）层级结构。将服务理解为文件夹、将服务对应的多个节点理解为文件夹下的文件，那么可以通过服务名称，用前缀查询的方式查询到某个服务的所有节点。1747226499385180161_0.541466578514098

如图，键名的规则可以是 `/业务前缀/服务名/服务节点地址` ：



2）列表结构。将所有的服务节点以列表的形式整体作为 value。1747226499385180161_0.6384537931586334

如图：



选择哪种存储结构呢？这个也会跟我们的技术选型有关。对于 ZooKeeper 和 Etcd 这种支持层级查询的中间件，用第一种结构会更清晰；对于 Redis，由于本身就支持列表数据结构，可以选择第二种结构。1747226499385180161_0.9087016290514485

最后，一定要给 key 设置过期时间，比如默认 30 秒过期，这样如果服务提供者宕机了，也可以超时后自动移除。

做好整体的方案设计后，下面我们开发实现。

三、开发实现

1、注册中心开发

1）注册信息定义。

在 model 包下新建 `ServiceMetaInfo` 类，封装服务的注册信息，包括服务名称、服务版本号、服务地址（域名和端口号）、服务分组等。1747226499385180161_0.5460395515401961

代码如下：

```
package com.yupi.yurpc.model;

/**
 * 服务元信息（注册信息）
 */
public class ServiceMetaInfo {

    /**
     * 服务名称
     */
    private String serviceName;

    /**
     * 服务版本号
     */
    private String serviceVersion = "1.0";

    /**
     * 服务域名
     */
    private String serviceHost;

    /**
     * 服务端口号
     */
    private Integer servicePort;
```

```
    /**
     * 服务分组（暂未实现）
     */
    private String serviceGroup = "default";

}
```

需要给 `ServiceMetaInfo` 增加一些工具方法，用于获取服务注册键名、获取服务注册节点键名等。

1747226499385180161_0.3494551452390757

可以把版本号和分组都放到服务键名中，就可以在查询时根据这些参数获取对应版本和分组的服务了。

代码如下：

```
/**
 * 获取服务键名
 *
 * @return
 */
public String getServiceKey() {
    // 后续可扩展服务分组
    // return String.format("%s:%s:%s", serviceName, serviceVersion, serviceGroup);
    return String.format("%s:%s", serviceName, serviceVersion);
}

/**
 * 获取服务注册节点键名
 *
 * @return
 */
public String getServiceNodeKey() {
    return String.format("%s/%s:%s", getServiceKey(), serviceHost, servicePort);
}
```

由于注册信息里包含了服务版本号字段，所以我们也可以给 RpcRequest 对象补充服务版本号字段，可以先作为预留字段，默认值为 "1.0"，后续再自行实现。

在 RpcConstant 常量类中补充默认服务版本常量：

```
package com.yupi.yurpc.constant;

/**
 * RPC 相关常量
 */
public interface RpcConstant {

    /**
     * 默认配置文件加载前缀
     */
    String DEFAULT_CONFIG_PREFIX = "rpc";

    /**
     * 默认服务版本
     */
    String DEFAULT_SERVICE_VERSION = "1.0";
}
```

在 RpcRequest 请求类中使用该常量，代码如下：

```
package com.yupi.yurpc.model;

import com.yupi.yurpc.constant.RpcConstant;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.io.Serializable;

/**
 * RPC 请求
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class RpcRequest implements Serializable {

    /**
     * 服务名称
     */
    private String serviceName;

    /**
     * 方法名称
     */
    private String methodName;

    /**
     * 服务版本
     */
    private String serviceVersion = RpcConstant.DEFAULT_SERVICE_VERSION;

    /**
     * 参数类型列表
     */
    private Class<?>[] parameterTypes;

    /**
     * 参数列表
     */
    private Object[] args;

}
```

2）注册中心配置。1747226499385180161_0.7928145044885933

在 config 包下编写注册中心配置类 `RegistryConfig` ，让用户配置连接注册中心所需的信息，比如注册中心类别、注册中心地址、用户名、密码、连接超时时间等。

代码如下：

```
package com.yupi.yurpc.config;

import lombok.Data;

/**
 * RPC 框架注册中心配置
 */
@Data
public class RegistryConfig {

    /**
     * 注册中心类别
     */
    private String registry = "etcd";

    /**
     * 注册中心地址
     */
    private String address = "http://localhost:2380";

    /**
     * 用户名
     */
    private String username;

    /**
     * 密码
     */
    private String password;

    /**
     * 超时时间（单位毫秒）
     */
    private Long timeout = 10000L;
}
```

还要为 RpcConfig 全局配置补充注册中心配置，代码如下：

```
@Data
public class RpcConfig {
    ...

    /**
     * 注册中心配置
     */
    private RegistryConfig registryConfig = new RegistryConfig();
}
```

3）注册中心接口。1747226499385180161_0.48423098421840205

遵循可扩展设计，我们先写一个注册中心接口，后续可以实现多种不同的注册中心，并且和序列化器一样，可以使用 SPI 机制动态加载。

注册中心接口代码如下，主要是提供了初始化、注册服务、注销服务、服务发现（获取服务节点列表）、服务销毁等方法。

```
package com.yupi.yurpc.registry;

import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;

import java.util.List;

/**
 * 注册中心
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public interface Registry {

    /**
     * 初始化
     *
     * @param registryConfig
     */
    void init(RegistryConfig registryConfig);

    /**
     * 注册服务（服务端）
     *
     * @param serviceMetaInfo
     */
    void register(ServiceMetaInfo serviceMetaInfo) throws Exception;

    /**
     * 注销服务（服务端）
     *
     * @param serviceMetaInfo
     */
    void unRegister(ServiceMetaInfo serviceMetaInfo);

    /**
     * 服务发现（获取某服务的所有节点，消费端）
     *
     * @param serviceKey 服务键名
     * @return
     */
    List<ServiceMetaInfo> serviceDiscovery(String serviceKey);

    /**
```

```
     * 服务销毁
     */
    void destroy();
}
```

4）Etcd 注册中心实现。

在 registry 目录下新建 `EtcdRegistry` 类，实现注册中心接口，先完成初始化方法，读取注册中心配置并初始化客户端对象。

代码如下：1747226499385180161_0.3975819147979025

```
public class EtcdRegistry implements Registry {

    private Client client;

    private KV kvClient;

    /**
     * 根节点
     */
    private static final String ETCD_ROOT_PATH = "/rpc/";

    @Override
    public void init(RegistryConfig registryConfig) {
        client = Client.builder().endpoints(registryConfig.getAddress()).connectTimeout(Duration.ofMillis(registryConfig.getTimeout())).b
        kvClient = client.getKVClient();
    }
}
```

上述代码中，我们定义 Etcd 键存储的根路径为 `/rpc/` ，为了区分不同的项目。

依次实现不同的方法，首先是服务注册，创建 key 并设置过期时间，value 为服务注册信息的 JSON 序列化。代码如下：1747226499385180161_0.20582332206273546

```
@Override
public void register(ServiceMetaInfo serviceMetaInfo) throws Exception {
    // 创建 Lease 和 KV 客户端
    Lease leaseClient = client.getLeaseClient();

    // 创建一个 30 秒的租约
    long leaseId = leaseClient.grant(30).get().getID();

    // 设置要存储的键值对
    String registerKey = ETCD_ROOT_PATH + serviceMetaInfo.getServiceNodeKey();
    ByteSequence key = ByteSequence.from(registerKey, StandardCharsets.UTF_8);
    ByteSequence value = ByteSequence.from(JSONUtil.toJsonStr(serviceMetaInfo), StandardCharsets.UTF_8);

    // 将键值对与租约关联起来，并设置过期时间
    PutOption putOption = PutOption.builder().withLeaseId(leaseId).build();
    kvClient.put(key, value, putOption).get();
}
```

然后是服务注销，删除 key：

```
public void unRegister(ServiceMetaInfo serviceMetaInfo) {
    kvClient.delete(ByteSequence.from(ETCD_ROOT_PATH + serviceMetaInfo.getServiceNodeKey(), StandardCharsets.UTF_8));
}
```

然后是服务发现，根据服务名称作为前缀，从 Etcd 获取服务下的节点列表：

```
public List<ServiceMetaInfo> serviceDiscovery(String serviceKey) {
    // 前缀搜索，结尾一定要加 '/'
    String searchPrefix = ETCD_ROOT_PATH + serviceKey + "/";

    try {
        // 前缀查询
        GetOption getOption = GetOption.builder().isPrefix(true).build();
        List<KeyValue> keyValues = kvClient.get(
                        ByteSequence.from(searchPrefix, StandardCharsets.UTF_8),
                        getOption)
                .get()
                .getKvs();
        // 解析服务信息
        return keyValues.stream()
                .map(keyValue -> {
                    String value = keyValue.getValue().toString(StandardCharsets.UTF_8);
                    return JSONUtil.toBean(value, ServiceMetaInfo.class);
                })
                .collect(Collectors.toList());
    } catch (Exception e) {
        throw new RuntimeException("获取服务列表失败", e);
    }
}
```

最后是注册中心销毁，用于项目关闭后释放资源：1747226499385180161_0.16117471093762759

```
public void destroy() {
    System.out.println("当前节点下线");
    // 释放资源
    if (kvClient != null) {
        kvClient.close();
    }
    if (client != null) {
        client.close();
    }
}
```

注册中心实现类的完整代码如下：

```
package com.yupi.yurpc.registry;

import cn.hutool.core.collection.CollUtil;
import cn.hutool.cron.CronUtil;
import cn.hutool.cron.task.Task;
import cn.hutool.json.JSONUtil;
import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;
import io.etcd.jetcd.*;
import io.etcd.jetcd.options.GetOption;
```

```
import io.etcd.jetcd.options.PutOption;

import java.nio.charset.StandardCharsets;
import java.time.Duration;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class EtcdRegistry implements Registry {

    private Client client;

    private KV kvClient;

    /**
     * 根节点
     */
    private static final String ETCD_ROOT_PATH = "/rpc/";

    @Override
    public void init(RegistryConfig registryConfig) {
        client = Client.builder().endpoints(registryConfig.getAddress()).connectTimeout(Duration.ofMillis(registryConfig.getTimeout())).b
        kvClient = client.getKVClient();
    }

    @Override
    public void register(ServiceMetaInfo serviceMetaInfo) throws Exception {
        // 创建 Lease 和 KV 客户端
        Lease leaseClient = client.getLeaseClient();

        // 创建一个 30 秒的租约
        long leaseId = leaseClient.grant(30).get().getID();

        // 设置要存储的键值对
        String registerKey = ETCD_ROOT_PATH + serviceMetaInfo.getServiceNodeKey();
        ByteSequence key = ByteSequence.from(registerKey, StandardCharsets.UTF_8);
        ByteSequence value = ByteSequence.from(JSONUtil.toJsonStr(serviceMetaInfo), StandardCharsets.UTF_8);

        // 将键值对与租约关联起来，并设置过期时间
        PutOption putOption = PutOption.builder().withLeaseId(leaseId).build();
        kvClient.put(key, value, putOption).get();
    }

    @Override
    public void unRegister(ServiceMetaInfo serviceMetaInfo) {
        kvClient.delete(ByteSequence.from(ETCD_ROOT_PATH + serviceMetaInfo.getServiceNodeKey(), StandardCharsets.UTF_8));
    }

    @Override
    public List<ServiceMetaInfo> serviceDiscovery(String serviceKey) {
        // 前缀搜索，结尾一定要加 '/'
        String searchPrefix = ETCD_ROOT_PATH + serviceKey + "/";

        try {
            // 前缀查询
            GetOption getOption = GetOption.builder().isPrefix(true).build();
            List<KeyValue> keyValues = kvClient.get(
                    ByteSequence.from(searchPrefix, StandardCharsets.UTF_8),
                    getOption)
                    .get()
                    .getKvs();
            // 解析服务信息
            return keyValues.stream()
                    .map(keyValue -> {
                        String value = keyValue.getValue().toString(StandardCharsets.UTF_8);
                        return JSONUtil.toBean(value, ServiceMetaInfo.class);
                    })
                    .collect(Collectors.toList());
        } catch (Exception e) {
            throw new RuntimeException("获取服务列表失败", e);
        }
    }

    @Override
    public void destroy() {
        System.out.println("当前节点下线");
        // 释放资源
        if (kvClient != null) {
            kvClient.close();
        }
        if (client != null) {
            client.close();
        }
    }
}
```

2、支持配置和扩展注册中心

一个成熟的 RPC 框架可能会支持多个注册中心，像序列化器一样，我们的需求是，让开发者能够填写配置来指定使用的注册中心，并且支持自定义注册中心，让框架更易用、更利于扩展。

要实现这点，开发方式和序列化器也是一样的，都可以使用工厂创建对象、使用 SPI 动态加载自定义的注册中心。
1747226499385180161_0.3260223039039032

1）注册中心常量。

在 registry 包下新建 `RegistryKeys` 类，列举所有支持的注册中心键名。

代码如下：1747226499385180161_0.15010206231391865

```
package com.yupi.yurpc.registry;

/**
 * 注册中心键名常量
 */
public interface RegistryKeys {

    String ETCD = "etcd";

    String ZOOKEEPER = "zookeeper";

}
```

2）使用工厂模式，支持根据 key 从 SPI 获取注册中心对象实例。

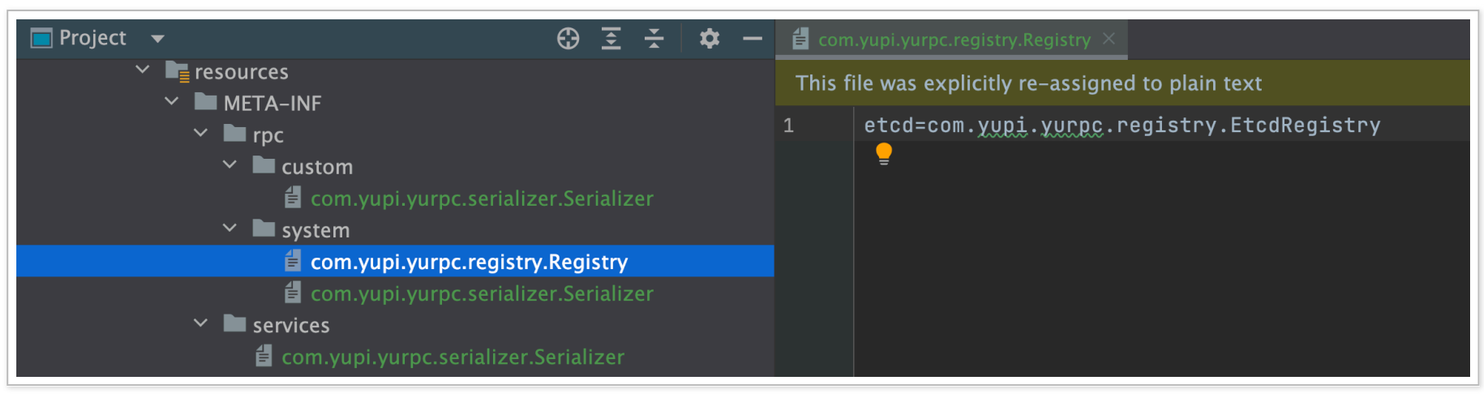
在 registry 包下新建 `RegistryFactory` 类，代码如下：1747226499385180161_0.9514603049210113

```
package com.yupi.yurpc.registry;

import com.yupi.yurpc.spi.SpiLoader;

/**
 * 注册中心工厂（用于获取注册中心对象）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class RegistryFactory {

    static {
        SpiLoader.load(Registry.class);
    }

    /**
     * 默认注册中心
     */
    private static final Registry DEFAULT_REGISTRY = new EtcdRegistry();

    /**
     * 获取实例
     *
     * @param key
     * @return
     */
    public static Registry getInstance(String key) {
        return SpiLoader.getInstance(Registry.class, key);
    }

}
```

这个类可以直接复制之前的 SerializerFactory，然后略做修改。可以发现，只要跑通了 SPI 机制，后续的开发就很简单了~

3）在 `META-INF` 的 `rpc/system` 目录下编写注册中心接口的 SPI 配置文件，文件名称为 `com.yupi.yurpc.registry.Registry` 。1747226499385180161_0.9557078039922486

如图：



代码如下：1747226499385180161_0.4228263252065858

```
etcd=com.yupi.yurpc.registry.EtcdRegistry
```

4）最后，我们需要一个位置来初始化注册中心。由于服务提供者和服务消费者都需要和注册中心建立连接，是一个 RPC 框架启动必不可少的环节，所以可以将初始化流程放在 `RpcApplication` 类中。

修改其 `init` 方法代码如下：1747226499385180161_0.22252723765808513

```
/**
 * 框架初始化，支持传入自定义配置
 *
 * @param newRpcConfig
 */
public static void init(RpcConfig newRpcConfig) {
    rpcConfig = newRpcConfig;
    log.info("rpc init, config = {}", newRpcConfig.toString());
    // 注册中心初始化
    RegistryConfig registryConfig = rpcConfig.getRegistryConfig();
    Registry registry = RegistryFactory.getInstance(registryConfig.getRegistry());
    registry.init(registryConfig);
    log.info("registry init, config = {}", registryConfig);
}
```

3、完成调用流程

下面我们要改造服务消费者调用服务的代码，跑通整个动态获取节点并调用的流程。1747226499385180161_0.8435376301566495

1）服务消费者需要先从注册中心获取节点信息，再得到调用地址并执行。

需要给 `ServiceMetaInfo` 类增加一个方法，便于获取可调用的地址，代码如下：

```
/**
 * 获取完整服务地址
 *
 * @return
 */
public String getServiceAddress() {
    if (!StrUtil.contains(serviceHost, "http")) {
        return String.format("http://%s:%s", serviceHost, servicePort);
    }
    return String.format("%s:%s", serviceHost, servicePort);
}
```

2）修改服务代理 `ServiceProxy` 类，更改调用逻辑。

修改的部分代码如下：

```
...

// 序列化
byte[] bodyBytes = serializer.serialize(rpcRequest);

// 从注册中心获取服务提供者请求地址
RpcConfig rpcConfig = RpcApplication.getRpcConfig();
Registry registry = RegistryFactory.getInstance(rpcConfig.getRegistryConfig().getRegistry());
ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
serviceMetaInfo.setServiceName(serviceName);
serviceMetaInfo.setServiceVersion(RpcConstant.DEFAULT_SERVICE_VERSION);
List<ServiceMetaInfo> serviceMetaInfoList = registry.serviceDiscovery(serviceMetaInfo.getServiceKey());
if (CollUtil.isEmpty(serviceMetaInfoList)) {
    throw new RuntimeException("暂无服务地址");
}
// 暂时先取第一个
ServiceMetaInfo selectedServiceMetaInfo = serviceMetaInfoList.get(0);

// 发送请求
try (HttpResponse httpResponse = HttpRequest.post(selectedServiceMetaInfo.getServiceAddress())
        .body(bodyBytes)
        .execute()) {
    byte[] result = httpResponse.bodyBytes();
    // 反序列化
    RpcResponse rpcResponse = serializer.deserialize(result, RpcResponse.class);
    return rpcResponse.getData();
}

...
```

注意，从注册中心获取到的服务节点地址可能是多个。上述代码中，我们为了方便，暂时先取第一个，之后会带大家对这里的代码进行优化。

`ServiceProxy` 的完整代码如下：

```
package com.yupi.yurpc.proxy;

import cn.hutool.core.collection.CollUtil;
import cn.hutool.http.HttpRequest;
import cn.hutool.http.HttpResponse;
import com.yupi.yurpc.RpcApplication;
import com.yupi.yurpc.config.RpcConfig;
import com.yupi.yurpc.constant.RpcConstant;
import com.yupi.yurpc.model.RpcRequest;
import com.yupi.yurpc.model.RpcResponse;
import com.yupi.yurpc.model.ServiceMetaInfo;
import com.yupi.yurpc.registry.Registry;
import com.yupi.yurpc.registry.RegistryFactory;
import com.yupi.yurpc.serializer.Serializer;
import com.yupi.yurpc.serializer.SerializerFactory;
import java.io.IOException;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.List;

/**
 * 服务代理（JDK 动态代理）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class ServiceProxy implements InvocationHandler {

    /**
     * 调用代理
     *
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 指定序列化器
        final Serializer serializer = SerializerFactory.getInstance(RpcApplication.getRpcConfig().getSerializer());

        // 构造请求
        String serviceName = method.getDeclaringClass().getName();
        RpcRequest rpcRequest = RpcRequest.builder()
                .serviceName(serviceName)
                .methodName(method.getName())
                .parameterTypes(method.getParameterTypes())
                .args(args)
                .build();
        try {
            // 序列化
            byte[] bodyBytes = serializer.serialize(rpcRequest);

            // 从注册中心获取服务提供者请求地址
            RpcConfig rpcConfig = RpcApplication.getRpcConfig();
            Registry registry = RegistryFactory.getInstance(rpcConfig.getRegistryConfig().getRegistry());
            ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
            serviceMetaInfo.setServiceName(serviceName);
            serviceMetaInfo.setServiceVersion(RpcConstant.DEFAULT_SERVICE_VERSION);
            List<ServiceMetaInfo> serviceMetaInfoList = registry.serviceDiscovery(serviceMetaInfo.getServiceKey());
            if (CollUtil.isEmpty(serviceMetaInfoList)) {
                throw new RuntimeException("暂无服务地址");
            }
            ServiceMetaInfo selectedServiceMetaInfo = serviceMetaInfoList.get(0);

            // 发送请求
            try (HttpResponse httpResponse = HttpRequest.post(selectedServiceMetaInfo.getServiceAddress())
                    .body(bodyBytes)
                    .execute()) {
                byte[] result = httpResponse.bodyBytes();
                // 反序列化
                RpcResponse rpcResponse = serializer.deserialize(result, RpcResponse.class);
                return rpcResponse.getData();
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```

四、测试

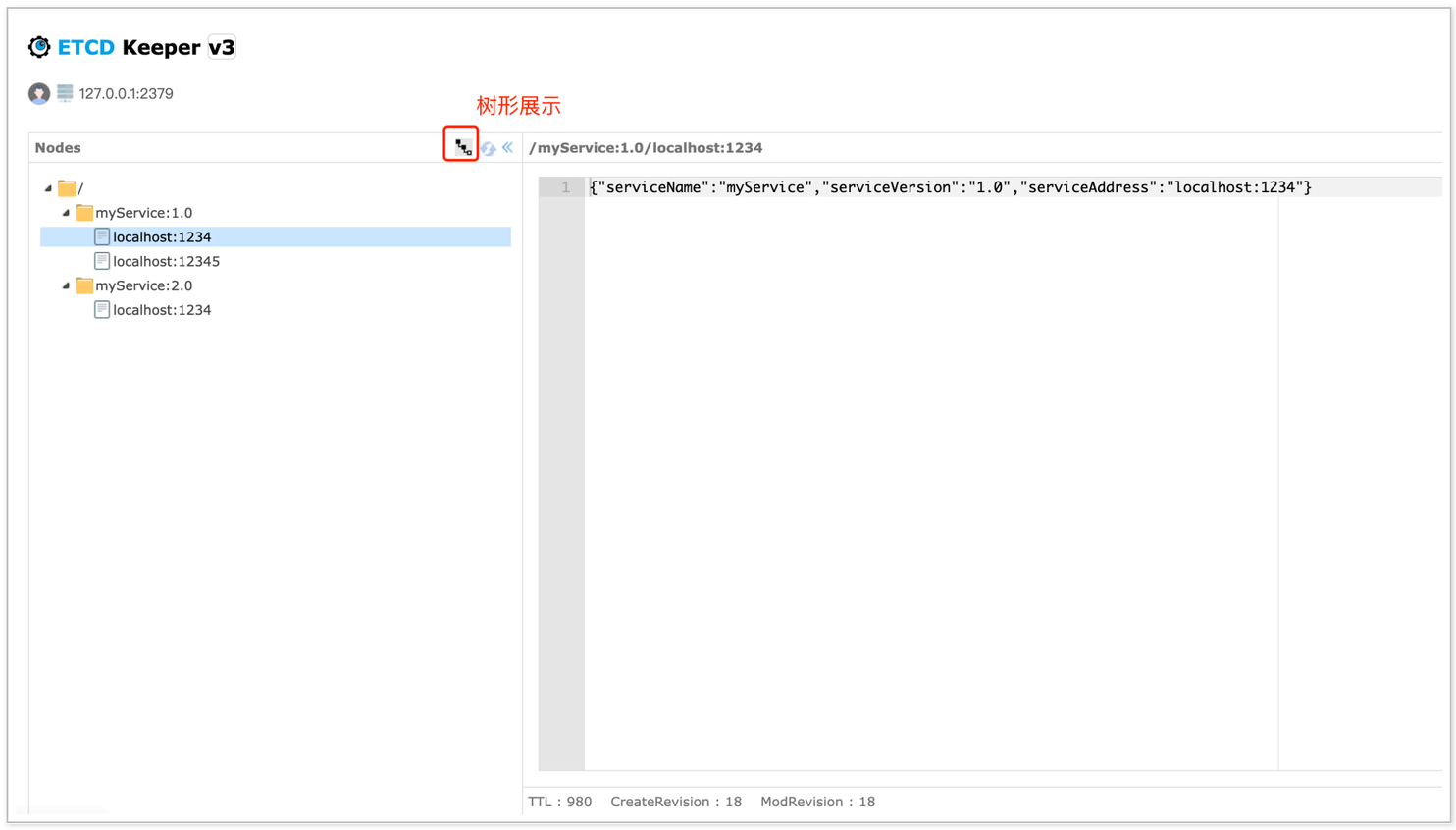
1、注册中心测试

首先验证注册中心能否正常完成服务注册、注销、服务发现。1747226499385180161_0.2278510133357412

编写单元测试类 `RegistryTest` ，代码如下：

```
package com.yupi.yurpc.registry;

import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import java.util.List;

/**
 * 注册中心测试
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">程序员鱼皮的编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class RegistryTest {

    final Registry registry = new EtcdRegistry();

    @Before
    public void init() {
        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setAddress("http://localhost:2379");
        registry.init(registryConfig);
    }

    @Test
    public void register() throws Exception {
        ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
        serviceMetaInfo.setServiceName("myService");
        serviceMetaInfo.setServiceVersion("1.0");
        serviceMetaInfo.setServiceHost("localhost");
        serviceMetaInfo.setServicePort(1234);
        registry.register(serviceMetaInfo);
        serviceMetaInfo = new ServiceMetaInfo();
        serviceMetaInfo.setServiceName("myService");
        serviceMetaInfo.setServiceVersion("1.0");
        serviceMetaInfo.setServiceHost("localhost");
        serviceMetaInfo.setServicePort(1235);
        registry.register(serviceMetaInfo);
        serviceMetaInfo = new ServiceMetaInfo();
        serviceMetaInfo.setServiceName("myService");
        serviceMetaInfo.setServiceVersion("2.0");
        serviceMetaInfo.setServiceHost("localhost");
        serviceMetaInfo.setServicePort(1234);
        registry.register(serviceMetaInfo);
    }

    @Test
    public void unRegister() {
        ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
        serviceMetaInfo.setServiceName("myService");
        serviceMetaInfo.setServiceVersion("1.0");
        serviceMetaInfo.setServiceHost("localhost");
        serviceMetaInfo.setServicePort(1234);
        registry.unRegister(serviceMetaInfo);
    }

    @Test
    public void serviceDiscovery() {
        ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
        serviceMetaInfo.setServiceName("myService");
        serviceMetaInfo.setServiceVersion("1.0");
        String serviceKey = serviceMetaInfo.getServiceKey();
        List<ServiceMetaInfo> serviceMetaInfoList = registry.serviceDiscovery(serviceKey);
        Assert.assertNotNull(serviceMetaInfoList);
    }
}
```

服务注册后，打开 EtcdKeeper 可视化界面，能够看到注册成功的服务节点信息，如图：
1747226499385180161_0.5872796158981153



可以发现 key 列表是树形展示的，因为 Etcd 是层级结构，很清晰。

2、完整流程测试

在 `example-provider` 模块下新增服务提供者示例类，需要初始化 RPC 框架并且将服务手动注册到注册中心上。

代码如下：

```
package com.yupi.example.provider;

import cn.hutool.core.net.NetUtil;
import com.yupi.example.common.service.UserService;
import com.yupi.yurpc.RpcApplication;
import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.config.RpcConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;
import com.yupi.yurpc.registry.EtcdRegistry;
import com.yupi.yurpc.registry.LocalRegistry;
import com.yupi.yurpc.registry.Registry;
import com.yupi.yurpc.registry.RegistryFactory;
import com.yupi.yurpc.server.HttpServer;
import com.yupi.yurpc.server.VertxHttpServer;

/**
 * 服务提供者示例
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class ProviderExample {

    public static void main(String[] args) {
        // RPC 框架初始化
        RpcApplication.init();

        // 注册服务
        String serviceName = UserService.class.getName();
        LocalRegistry.register(serviceName, UserServiceImpl.class);

        // 注册服务到注册中心
        RpcConfig rpcConfig = RpcApplication.getRpcConfig();
        RegistryConfig registryConfig = rpcConfig.getRegistryConfig();
        Registry registry = RegistryFactory.getInstance(registryConfig.getRegistry());
        ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
        serviceMetaInfo.setServiceName(serviceName);
        serviceMetaInfo.setServiceHost(rpcConfig.getServerHost());
        serviceMetaInfo.setServicePort(rpcConfig.getServerPort());
        try {
            registry.register(serviceMetaInfo);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        // 启动 web 服务
        HttpServer httpServer = new VertxHttpServer();
        httpServer.doStart(RpcApplication.getRpcConfig().getServerPort());
    }
}
```

服务消费者的代码不用改动，我们依然是先启动提供者、再启动消费者，验证流程能否正常跑通。

五、扩展

至此，我们完成基本的注册中心，大家可以思考下注册中心还有哪些值得去完善和优化的地方，下节教程中，鱼皮会手把手带大家扩展注册中心。1747226499385180161_0.44856344109370205

1747226499385180161_0.22477760475642872