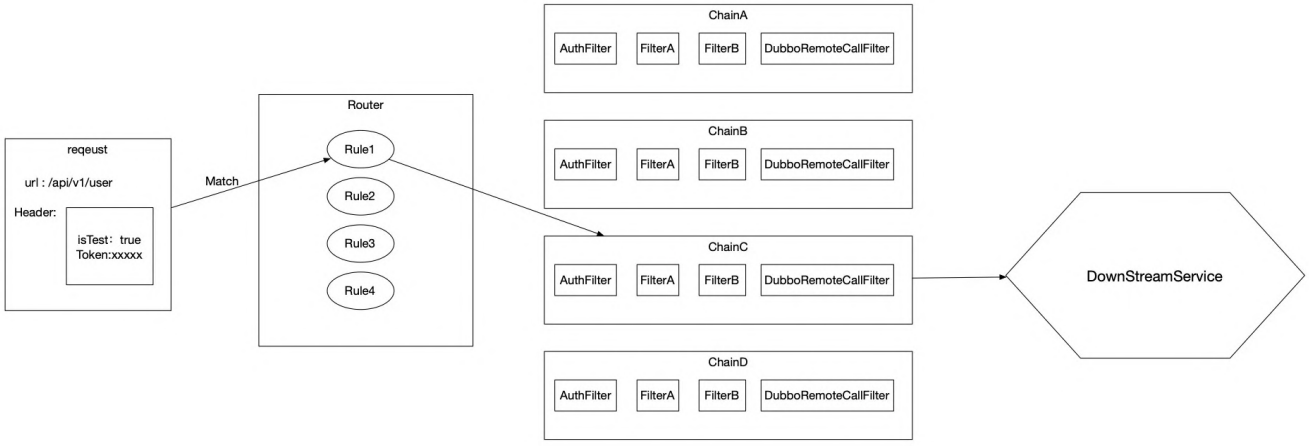
Pixiu路由原理简介

简介



网关的核心之一是路由逻辑，决定一个请求需要经过怎样的加工，被转发到哪个下行服务。
其中 80% 的路由需求表达都以 URL 为基础。需要描述清楚具有某个特征的 URL 或者 URL 集合对应怎样的一系列下游处理策略。

例如，'/test/**' 开头的 URL 路由到测试环境集群，'/release/user/**' 开头的 URL 会被路由到正式环境的 user 服务集群。

同时网关作为所有请求的入口，每一毫秒的延时都会做用在全量的业务下，在 mesh 场景下，延时还会随着调用链路的加深，被倍数放大。按照生产环境业务相应 <=7 毫秒的标准来看，规则匹配的性能要求也是十分苛刻的。一定不能随着规则数目的增加而性能退化。

使用介绍

仅从使用方的角度阐述 pixiu 的配置文件如何描述 URL 相关的路由规则。（下面，我们介绍一下如何配置 URL 路由规则）

如下是一份 pixiu 的 api 配置文件，这份配置文件会被解析后生成一份对应的内存模型，作为 pixiu 路由相关配置的初始状态。之后由 RDS 协议修改解析后得到的内存模型，实现路由逻辑动态生效的效果。RDS 协议（RDS：xDS 协议下描述路由规则的部分）相关内容是后话不详细阐述。我们把注意力聚焦到resource部分。

resource 下 path 部分就是上文阐述的，URL 相关的路由描述。意思是满足 path 描述特征的 URL 会被成功匹配。

JSON

```yaml
name: server
description: server sample
resources:
  - path: '/api/v1/test-dubbo/user/name/:name'
    type: restful
    description: user
    methods:
      - httpVerb: GET
        enable: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
          uri:
            - name: name
              required: true
        integrationRequest:
          requestType: dubbo
          mappingParams:
            - name: uri.name
              mapTo: 0
              mapType: "string"
          applicationName: "UserProvider"
          interface: "com.dubbogo.server.UserService"
          method: "GetUserByName"
          group: "test"
          version: 1.0.0
          clusterName: "test_dubbo"


```

被匹配后的请求会被转化成 dubbo 协议转发到 test_dubbo 集群调用 com.dubbogo.server.UserService 下的 GetUserByName 服务。
我们继续聚焦到如下范围：

```JSON
path: '/api/v1/test-dubbo/user/name/:name'
```

为了描述清楚一个 URL 或者一组 URL，路由引擎需要拥有以下能力：

1. URL 可以包含变量，'/api/v1/test-dubbo/user/name/:name' 代表 URL 用“/”分割后，第六个部分的值作为变量 name 的值，向下游 filter 传递供filter使用。
2. 需要有通配符
   a. * 代表一个层级任意字符的通配 '/api/*/test-dubbo/user/name/:name' 这样的一个 path 描述代表了可能并不关心具体的版本，不论什么版本下的 URL 只要匹配都使用相同的逻辑加工数据并转发。
   b. ** 代表多个层级的通配，从这个层级以后，子层级也可以是任意字符，**只可能存在于 URL 的末尾，不然会有二义性。'/api/v1/**' 这样的 path 表达了所有 V1 版本下的 URL 都采用相同的逻辑。

为了正确的使用 pixiu 您可能还需要了解如下内容。

优先级

并非是独创的，类似 java 下的 spring 以及其他框架统一具有的优先级逻辑：

1. 通配的优先级低于特指 。 '/api/v1/**' 低于 '/api/v1/test-dubbo/user/name/:name' 的优先级，假设有两个 resource 分别采用了如上两个path 配置，request 为 '/api/v1/test-dubbo/user/name/yqxu' 的请求到达pixiu 后应该生效哪个 resource？按照通配低于特指的原则，'/api/v1/test-dubbo/user/name/:name' 这条规则会生效。
2. 深度更深的，优先级更高 。 '/api/v1/**' 对比 /api/v1/test-dubbo/**' ，如果请求同时满足如上两个描述， '/api/v1/test-dubbo/**' 深度更深，会生效。
3. 通配符之间 '/*' 优先级高于 '/**'
4. 变量等同于通配。

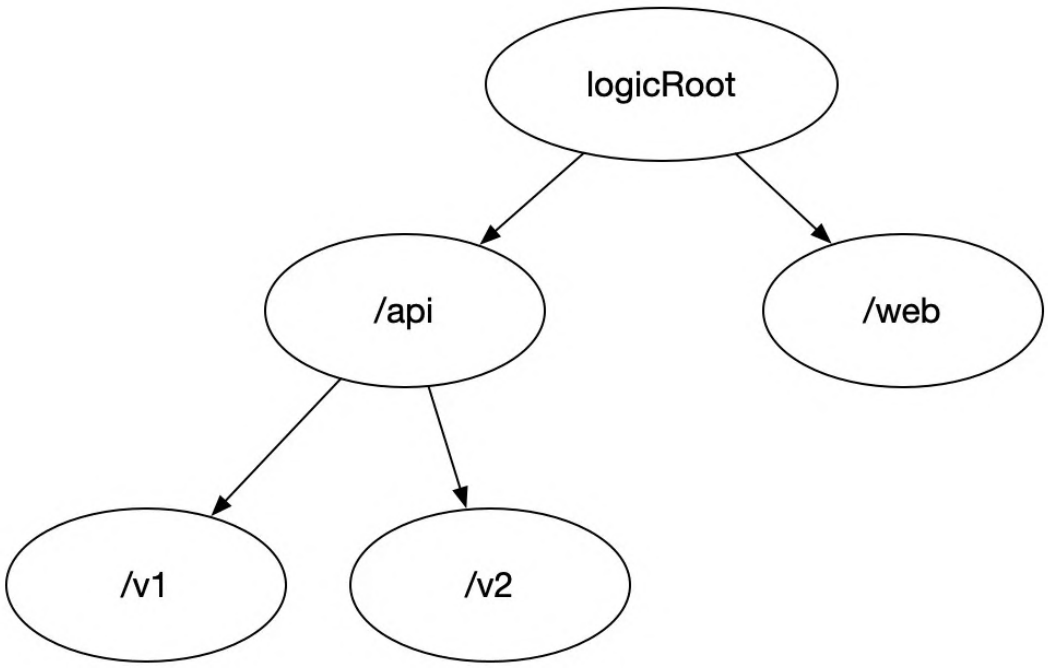
冲突处理

优先级规则只是冲突解决策略的一种，才同时匹配多个url描述时，优先级更高的那一种将会生效，然而优先级策略并不能涵盖所有的情况。
如果强行配置两条 resource path 完全相同，但是转发到不同的下游服务，这时候就会冲突。pixiu 下应对冲突的方案是 failfast，在 pixiu 初始化阶段，发现配置文件中有冲突的两项规则，则启动失败，让开发者今早发现问题并处理。

原理介绍

技术选型之初，以及确定使用pixiu后为了处理一些突发情况，以及应付一些pixiu自身可能存在的bug，开发者需要对pixiu 的路由原理有更深刻的了解。
下面，我们将详细介绍路由引擎的相关原理和实现，供感兴趣的同学了解。
相信阅读这部分内容的同学一定会有人下意识联想到字典树这个结构。使用字典树这个结构能实现存量规则数无关的匹配性能优化。

一个存放字符串作为node的字典树，具有表达url 的能力。



如上图描述等价于URL集合 '/api/v1' ，'/api/v2' ，'/web'

维护一个标准字典树有几个关键的操作

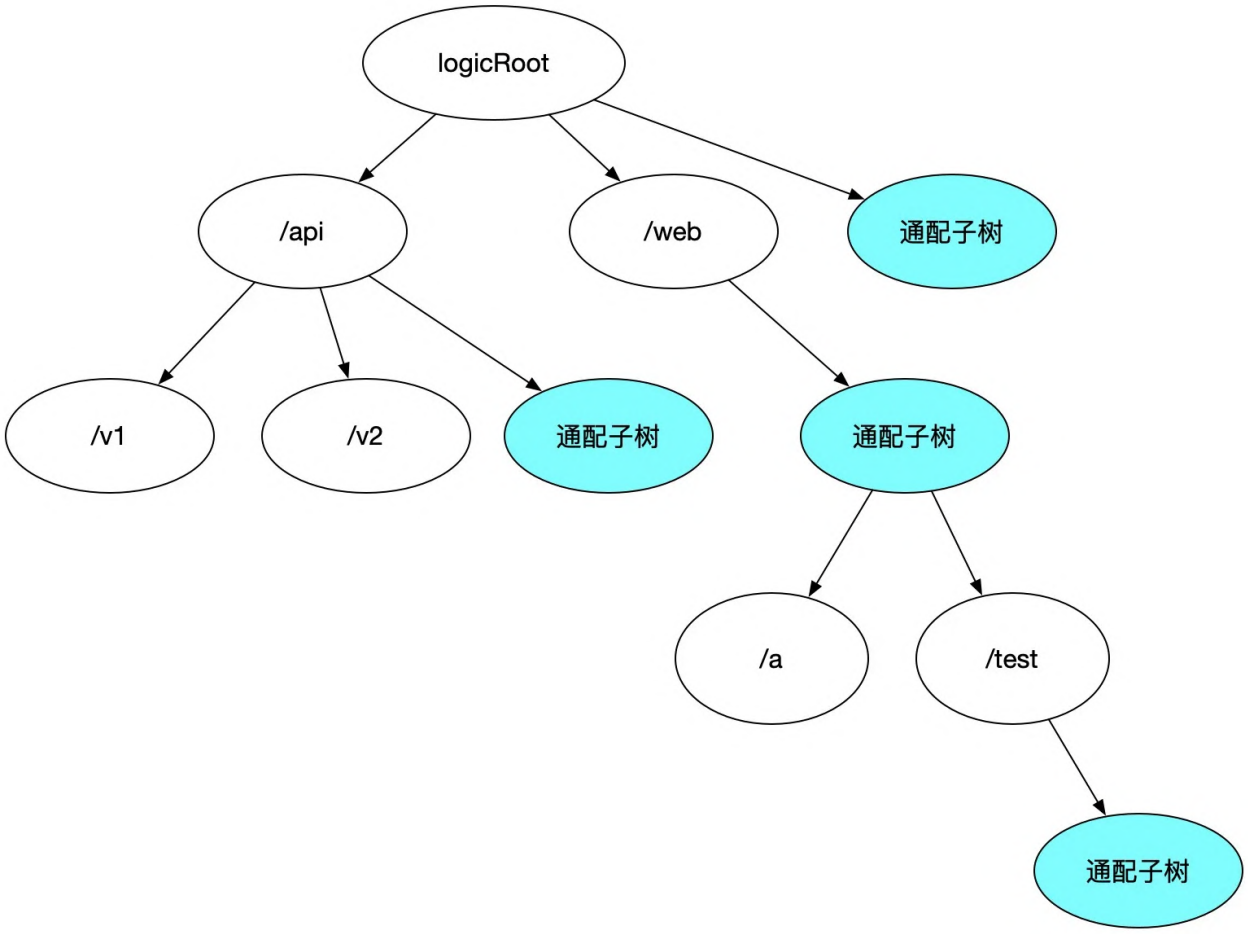
1. 字典树指定节点的查找（find）：从root 开始遍历字典书，'/api/v2' 称之为路径，在当前层级寻找指定路径，如果存在就继续在子树下完成剩下的路径匹配。'/api/v2' 先从 logic root 找到 '/api' ，并在 '/api' 的子树下继续查找剩下的路径 '/v2' 。
2. 字典树节点的添加（add）： 尝试查找指定节点，如果指定节点不存在则新建一个节点。假设一个空树状态下添加 '/api/v1' ，因为是空树那么logic root 下查找 '/api' 一定不存在，则在 root 下创建 '/api' ，继续在创建的 '/api' 节点下查找 '/v1' 因为 '/api' 是新建的 v1 一定也不存在，则继续创建v1
3. 字典树url匹配（match）： 在这个最简单的版本下，匹配逻辑与指定节点的查找逻辑没有区别。

还有一些不涉及递归或者复用上面逻辑递归操作的简单操作

4. 修改字典树节点（modify）： 通过 find 逻辑找到指定节点，调用 set 方法或者直接赋值的方式修改节点内容。
5. 删除字典树节点（delete）：通过 modify 逻辑修改 isdeleted 标为 true，并把节点内容 modify 为空。节点本身的内存不释放。
6. 重建字典树（rebuild）： 遍历所有节点，添加到新树，如果 isdeleted为 true 则不添加到新树，通过rebuild 操作创建副本。

由上可知，标准字典树结构距离通用的路由引擎底层数据结构能力还有一定差距，缺乏统配描述能力，缺乏变量表达的能力，下面我们来看一下如何进行改进。

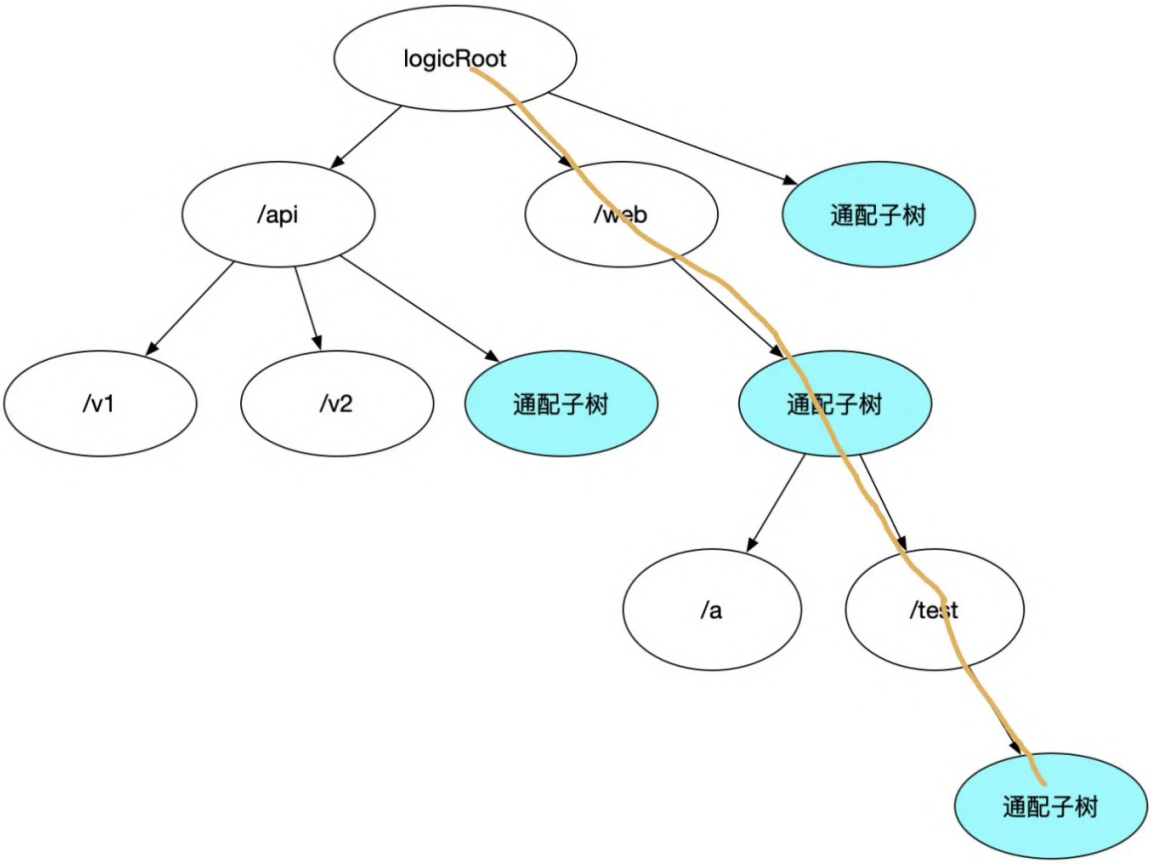
添加 描述统配逻辑的子树，作为子树中默认存在的一部分



现在我们的变种字典树多了变量表达能力

'/web/:appname/test/*' 这样的url 在图中应该怎么表达?

没错就是这个路径



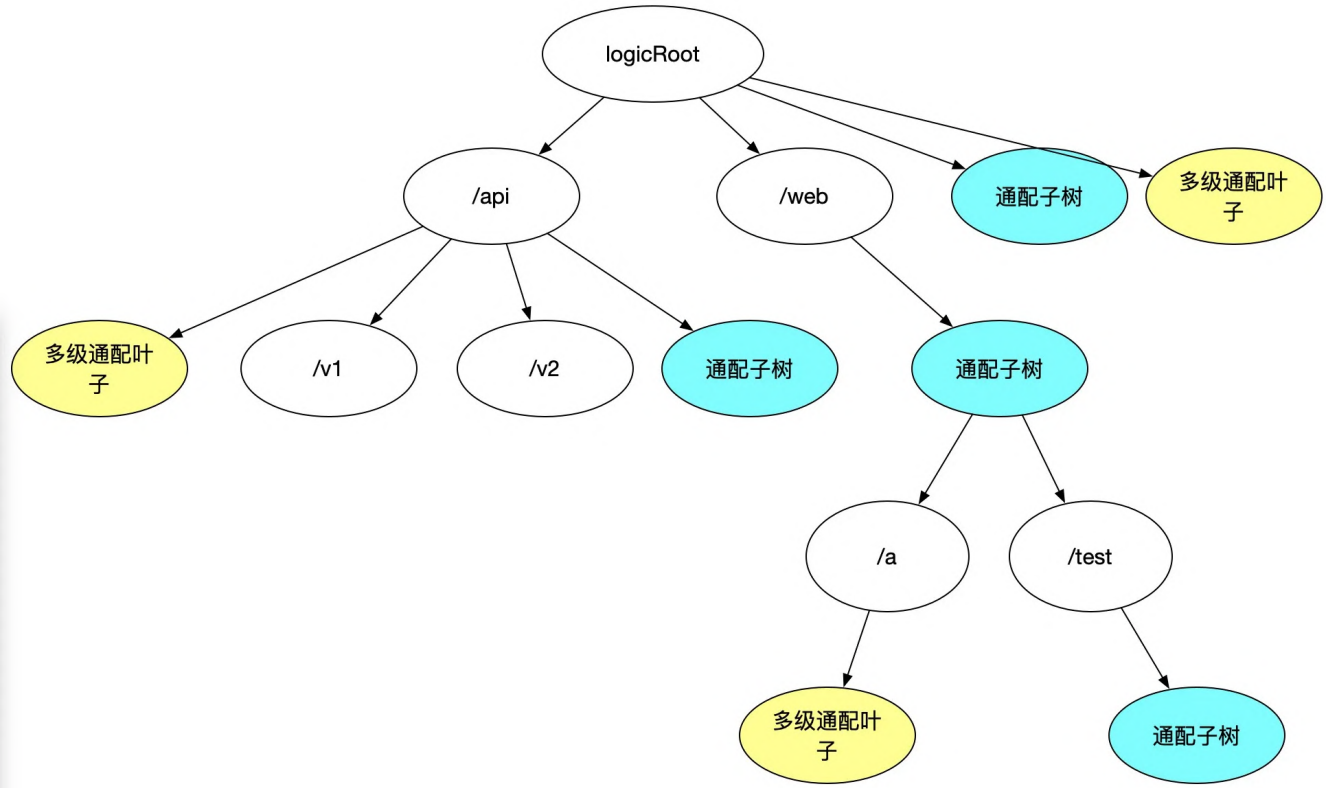
继续分析字典树几个关键的操作是否需要做变化?

1. 字典树指定节点的查找 ：
    a. 如果不改动使用前一版本逻辑在 '/*' 节点处理之前都不会有问题： 从root 开始遍历字典书，'/api/v2/*' 称之为路径，在当前层级寻找指定路径，如果存在就继续在子树下完成剩下的路径匹配。 /api/v2 先从 logic root 找到 '/api' ，并在 '/api' 的子树下继续查找剩下的路径 '/v2' 。
    b. 这版本我们加上对 '/*' 节点的处理：'/v2' 后是 '/*' ，'/*' 对应单级通配节点，继续递归查找 '/v2' 节点下一级通配节点是否为空。如果 path 是 '/api/v2/*/test2' 这样的路径则继续在统配子树下完成递归过程。
2. 字典树节点的添加 ：
    a. 在添加 '/*' 节点之前，所有逻辑上一版本就足够处理：尝试查找指定节点，如果指定节点不存在则新建一个节点。假设一个空树状态下添加 '/api/v1/*' ，因为是空树那么 logic root 下查找 '/api' 一定不存在，则在 root 下创建 '/api' ，继续在创建的 '/api' 节点下查找 '/v1' 因为 '/api' 是新建的 v1 一定也不存在，则继续创建 v1。
    b. 这版本加上 '/*' 的特殊处理 ： '/v1' 新建后，查看通配子树，通配子树不存在，则为V1 节点添

加内容为空的单级通配子树并在子树中继续递归。

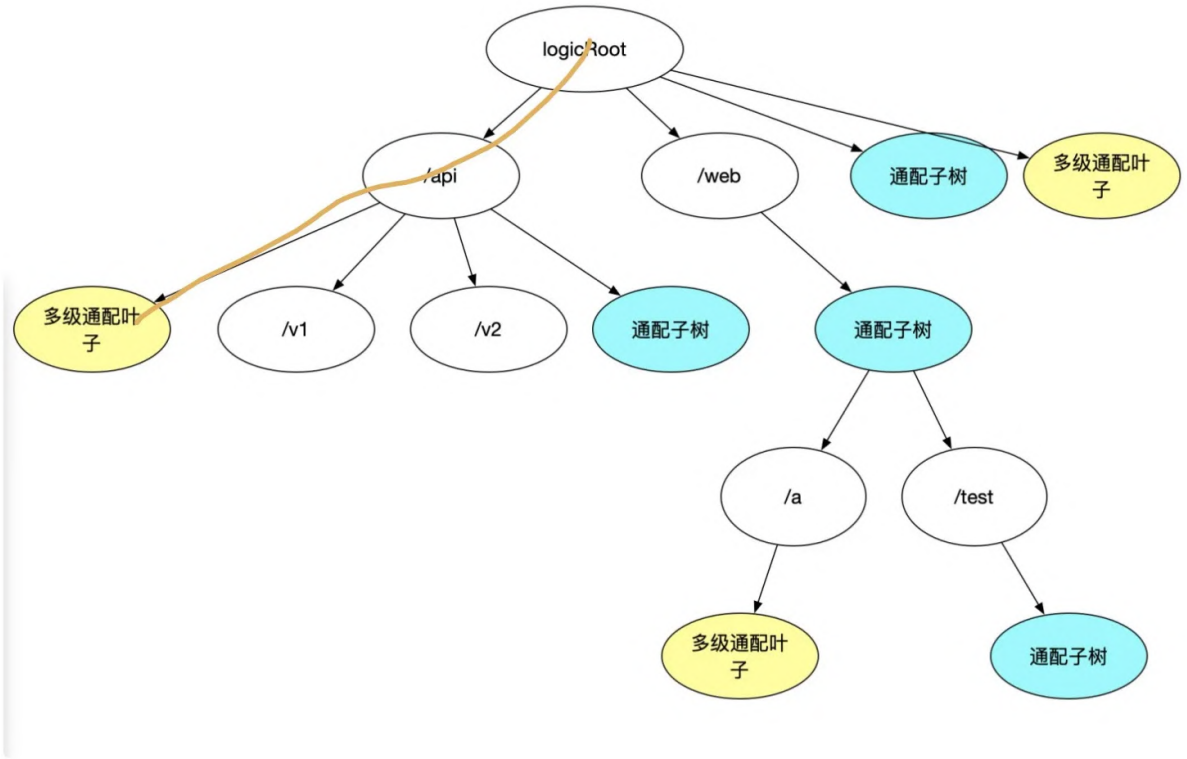
3. 字典树url匹配：在这个版本下，对比查找逻辑需要增加回朔逻辑。
    a. 在遇到通配节点前逻辑与find 依旧相同 ： 从root 开始遍历字典书，'/api/v2/*' 称之为路径，在当前层级寻找指定路径，如果存在就继续在子树下完成剩下的路径匹配。 '/api/v2' 先从 logic root 找到 '/api' ，并在 '/api' 的子树下继续查找剩下的路径 '/v2' 。
    b. 在处理统配节点的时候会与 find 逻辑有所不同：'/v2' 下普通子树无匹配节点，回朔到通配子树，查看是否能匹配，这个例子中 '/v2' 下无通配子树，查询不到节点 。值得注意的是回朔逻辑的先后顺序，是先找普通子树再回朔到通配子树还是先查找通配子树再回朔到普通子树是取决于优先级规则的，按照需求必须是先查找普通子树。

但是我们目前还是缺乏 '/**' 这种通配的表达能力代表了多级通配，可以分析需求得到结论，这种通配符，一定不存在子树，是一种特殊的叶子结点，仅用于 match 逻辑回朔时做特殊判断。继续加点特殊 node 后演化为：



好了至此，需求都能满足了。

'/api/**' 等价路径为：



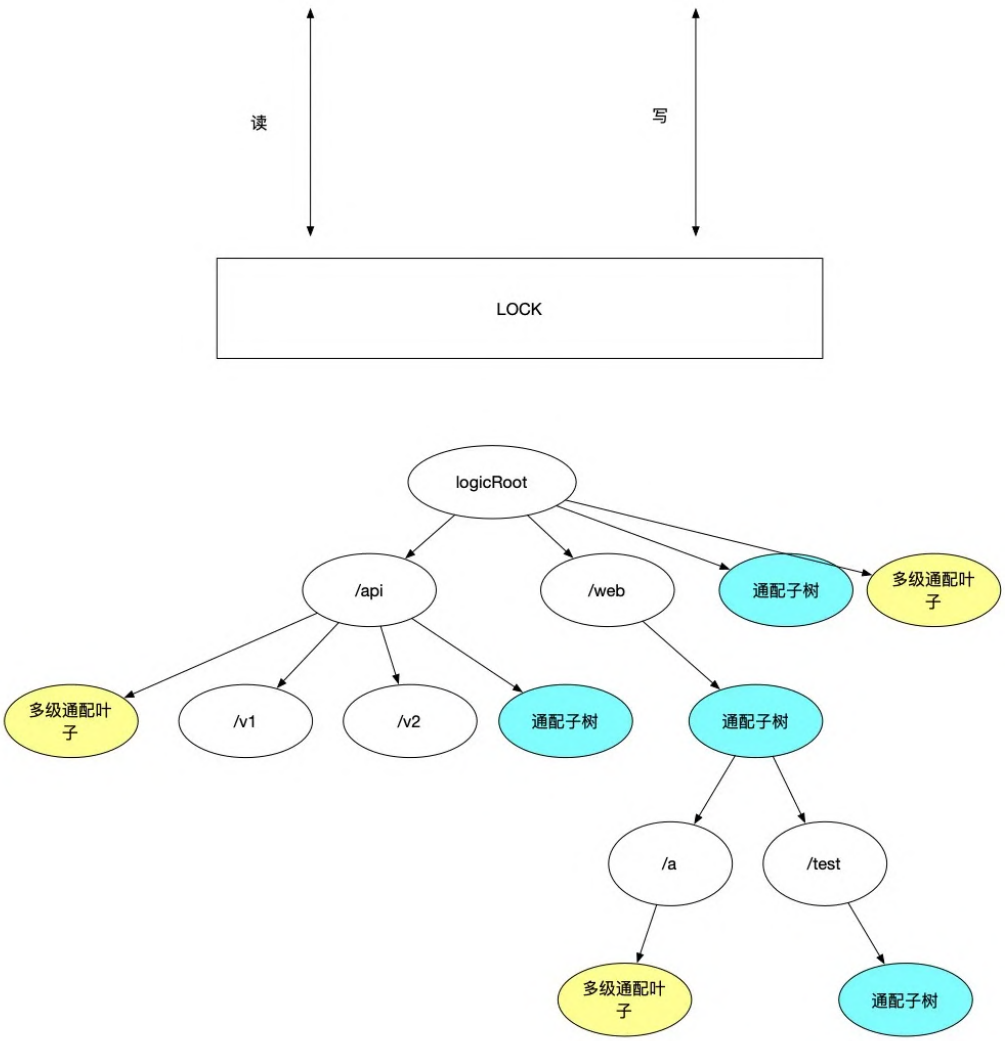
其他逻辑大同小异，match 逻辑回朔再多一级判断，如果一级通配子树也匹配不到结果，则再看一下多级通配子树是否为空（其实留一个标位就可以，为了统一模型好理解，还是用一个子树去描述）

到目前这个版本所有上文提到的能力已经都能有效支撑，回头分析一下时间复杂读。
url 被 '/' 分割出一个一个的段，容易理解在匹配一个url 过程中复杂度是 O(n) n= url 段数。与树中存有的规则数量无关。再分析 n 的范围，n 其实不是一个可以无限大的数字，一部分浏览器甚至约束 url 长度必须小于 2000，按照一个单词长度为 5 来计算，可以大概估计段数最多会在 400 左右，n 如果可以被视为一个常数，那么复杂读可以看作是 O（1）。

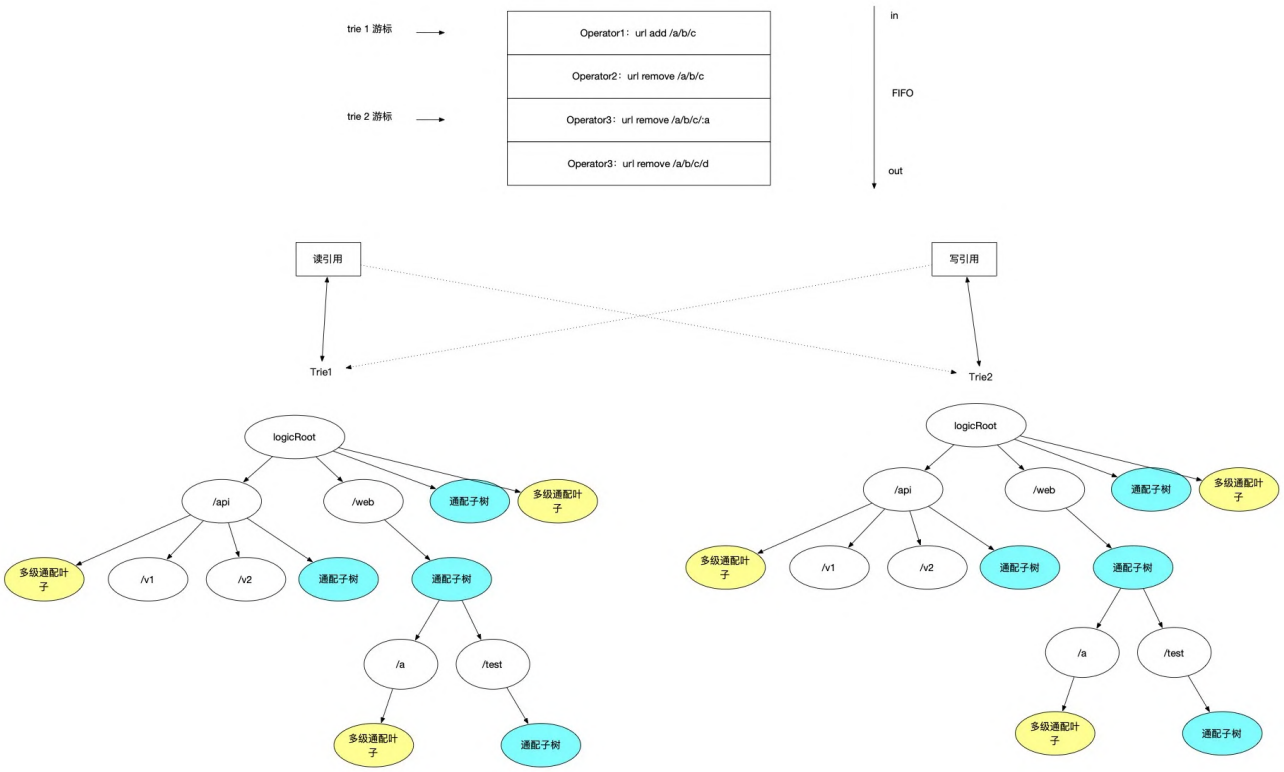
稍微解释一下find 和 match 有什么不同，为什么需要两种查找节点的方法。看下这个例子 :假设树中已经add 了 '/api/v1/:name/add' 这个 path，那么
find（"/api/v1/:name/add"），find（"/api/v1/*/add"）两个调用应该能够拿到结果，在add 的过程中用于冲突判断。
假设有请求进来url 为 '/api/v1/:name/add' 那么match（"/api/v1/:name/list"）也应该能 match 到结果且变量name 为 :name。
再假设有请求进来 url 为 '/api/v1/yq/add' 那么match（"/api/v1/yq/list"）也应该能 match 到结果且变量name 为 yq 。find（"/api/v1/yq/add" ） 则不会匹配到结果。

后续改进

目前实现在读树和写树之前，竞争一把全局锁，竞争失败后自旋直到竞争成功，然后完成读写。
解释一下为什么读都需要上锁，因为代码中大量运用了go 的 map 结构。这个结构只要并发读写直接会报如下错误：concurrent map read and map write
目前实现如下



引入 command 队列，所有对 trie 的用户写操作先入列，同时做读写分离，分为读树和写树，维护一个线程负责追 log 把 command 写入到写树，读树因为只读，没有写入线程操作读树所以可以不加锁。写树因为只有一条线程向树内写入，没有竞争问题，也可以不加锁。（写入操作并不会很频繁单线程完全能负荷）
定义一个配置延迟生效的时间，比如3s
每3秒，读树和写树角色切换，每个 trie 分别维护一个 command 队列的游标，游标代表当前这个 trie，追 log 追到了哪条记录，写入线程每3s 切换写游标引用。



如上图，最上面部分是一个先进先出的 command 队列，追 log 线程从这个队列中读取用户写操作，这个队列维护了两个游标 index1，index2，index1 代表了trie1 追 log 追到了 index1 的位置，index2 代表了 trie2 追 log 追到了 index2 的位置。追 log 线程同一时间内只会使用一个引用进行写操作，每次写完树对应的 index 游标下移一格，另一个 trie 引用将被用于读操作，一切读请求将从读引用对应的树中读取。因为追的是同一份 log ，最终一致性是能保证的。

切换逻辑：

1. 先使追 log 线程空转（不挂起，避免上下文切换，因为马上要恢复）
2. 保证两个树都没有写入线程操作
3. 切换读引用到另一个树
4. 切换写引用到另一个树
5. 恢复追 log 线程

pr：

https://github.com/apache/dubbo-go-pixiu/pull/262

pkg/common/router/trie/trie.go:26