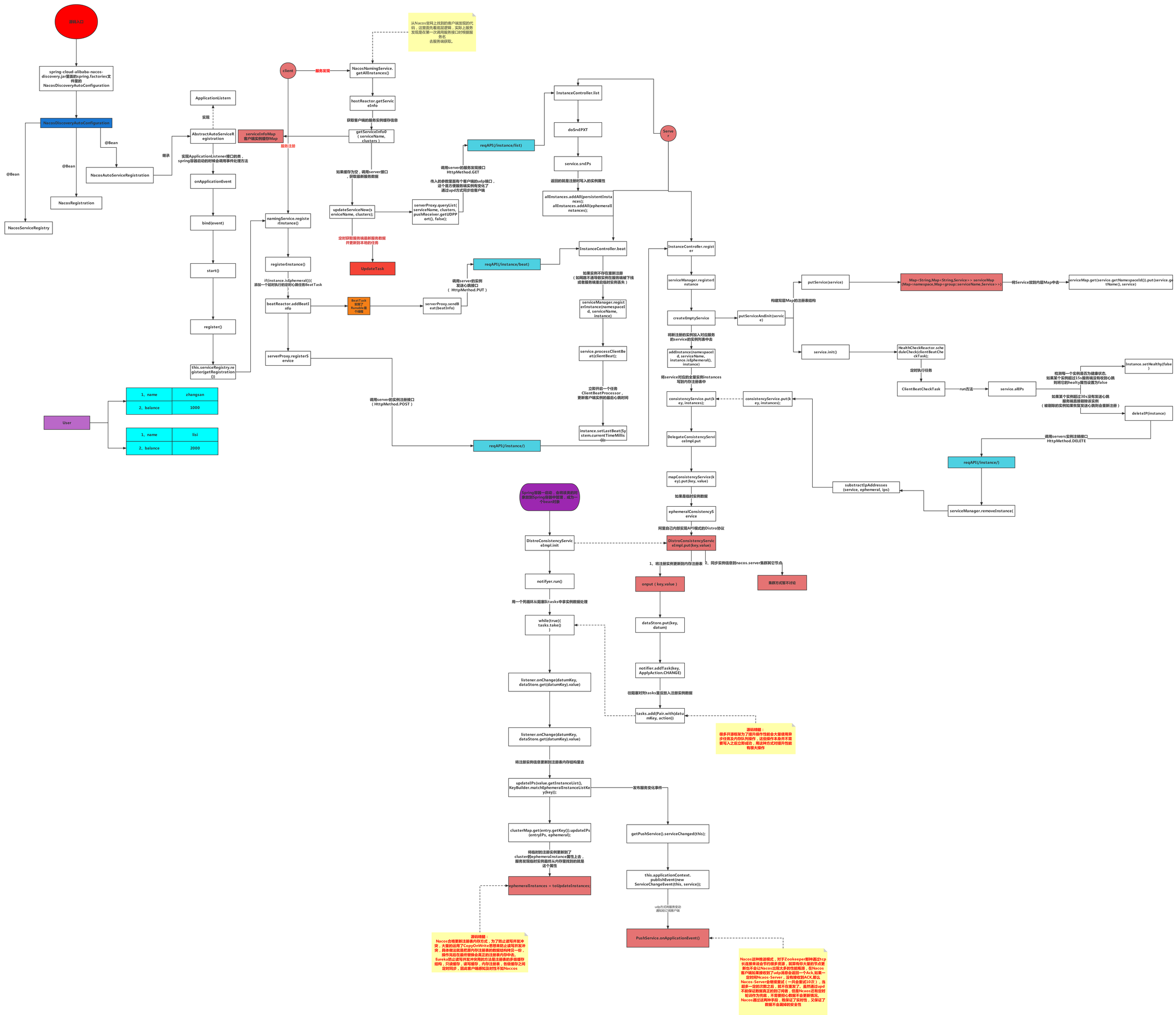
源码入口
spring-cloud-alibaba-nacos-discovery.jar里面的spring.factories文件里的NacosDiscoveryAutoConfiguration
NacosDiscoveryAutoConfiguration
@Bean
@Bean
@Bean
NacosAutoServiceRegistration
NacosRegistration
NacosServiceRegistry
继承
实现ApplicationListener接口的类，spring容器启动的时候会调用事件处理方法
AbstractAutoServiceRegistration
实现
ApplicationListern
onApplicationEvent
bind(event)
start()
register()
this.serviceRegistry.register(getRegistration())
namingService.registerInstance()
registerInstance()
If(instance.isEphemeral()){
添加一个延时执行的定时心跳任务BeatTask
}
beatReactor.addBeatInfo
BeatTask
实现了Runable接口线程
serverProxy.registerService
client
服务发现
NacosNamingService.getAllInstances()
从Nacos官网上找到的客户端发现的代码，这里面先看底层逻辑，实际上服务发现是在第一次调用服务接口时根据服务名
去服务端获取。
hostReactor.getServiceInfo
获取客户端的服务实例缓存信息
getServiceInfo0（serviceName, clusters）
serviceInfoMap
客户端实例缓存Map
服务注册
如果缓存为空，调用server接口，获取最新服务数据
updateServiceNow(serviceName, clusters);
serverProxy.queryList(serviceName, clusters, pushReceiver.getUDPPort(), false);
定时获取服务端最新服务数据并更新到本地的任务
UpdateTask
调用server的实例发送心跳接口（HttpMethod.PUT）
serverProxy.sendBeat(beatInfo)
reqAPI(/instance/beat)
InstanceController.beat
如果实例不存在重新注册（如网络不通导致实例在服务端被下线或者服务端重启临时实例丢失）
serviceManager.registerInstance(namespaceId, serviceName, instance)
service.processClientBeat(clientBeat);
立即开启一个任务ClientBeatProcessor，更新客户端实例的最后心跳时间
instance.setLastBeat(System.currentTimeMillis());
调用server的实例注册接口（HttpMethod.POST）
reqAPI(/instance/)
InstanceController.list
doSrvIPXT
service.srvIPs
返回的就是注册时写入的实例属性
allInstances.addAll(persistentInstances);
allInstances.addAll(ephemeralInstances);
reqAPI(/instance/list)
调用server的服务发现接口HttpMethod.GET
传入的参数里面有个客户端的udp端口，这个是方便服务端实例有变化了通过upd方式同步给客户端
Server
InstanceController.register
serviceManager.registerInstance
createEmptyService
putServiceAndInit(service)
putService(service)
Map<String,Map<String,Service>> serviceMap (Map<namespace,Map<group::serviceName,Service>>)
将Service放到内层Map中去
serviceMap.get(service.getNamespaceId()).put(service.getName(), service)
构建双层Map的注册表结构
service.init()
HealthCheckReactor.scheduleCheck(clientBeatCheckTask);
定时执行任务
ClientBeatCheckTask
run方法
service.allIPs
检测每一个实例是否为健康状态，如果某个实例超过15s服务端没有收到心跳则将它的healty属性设置为false
instance.setHealthy(false)
如果某个实例超过30s没有发送心跳服务端直接剔除该实例（被剔除的实例如果恢复发送心跳则会重新注册）
deleteIP(instance)
调用servers实例注销接口HttpMethod.DELETE
reqAPI(/instance/)
serviceManager.removeInstance(
substractIpAddresses(service, ephemeral, ips)
consistencyService.put(key, instances);
将新注册的实例加入对应服务的service的实例列表中去
addInstance(namespaceId, serviceName, instance.isEphemeral(), instance)
将service对应的全量实例instances写到内存注册表中
consistencyService.put(key, instances);
DelegateConsistencyServiceImpl.put
mapConsistencyService(key).put(key, value)
如果是临时实例数据
ephemeralConsistencyService
阿里自己内部实现API模式的Distro协议
DistroConsistencyServiceImpl.put(key,value)
1、将注册实例更新到内存注册表
2、同步实例信息到nacos.server集群其它节点
onput（key,value）
集群方式暂不讨论
dataStore.put(key, datum)
notifier.addTask(key, ApplyAction.CHANGE)
往阻塞对列tasks里反放入注册实例数据
tasks.add(Pair.with(datumKey, action))
源码精髓：
很多开源框架为了提升操作性能会大量使用异步任务及内存队列操作，这些操作本身并不需要写入之后立即成功，用这种方式对提升性能有很大操作
User
1、name
zhangsan
2、balance
1000
1、name
lisi
2、balance
2000
Spring容器一启动，会将该类的对象放到Spring容器中管理，成为一个bean对象
DistroConsistencyServiceImpl.init
notifyer.run()
用一个死循环从阻塞队tasks中拿实例数据处理
while(true){
tasks.take()
}
listener.onChange(datumKey, dataStore.get(datumKey).value)
listener.onChange(datumKey, dataStore.get(datumKey).value)
将注册实例信息更新到注册表内存结构里去
updateIPs(value.getInstanceList(), KeyBuilder.matchEphemeralInstanceListKey(key));
发布服务变化事件
clusterMap.get(entry.getKey()).updateIPs(entryIPs, ephemeral);
将临时的注册实例更新到了cluster的ephemeralInstance属性上去，服务发现临时实例是从内存里找到的就是这个属性
ephemeralInstances = toUpdateInstances;
源码精髓：
Nacos合格更新注册表内存方式，为了防止读写并发冲突，大量的运用了CopyOnWrite思想来防止读写并发冲突，具体做法就是把原内存注册表的数据结构拷贝一份，操作完成在最终替换会真正的注册表内存中去。
Eureka防止读写并发冲突用的方法是注册表的多级缓存结构，只读缓存，读写缓存，内存注册表，数据提存之间定时同步，因此客户端感知及时性不如Naccos
getPushService().serviceChanged(this);
this.applicationContext.publishEvent(new ServiceChangeEvent(this, service));
udp方式向服务变动通知给订阅客户端
PushService.onApplicationEvent()
Nacos这种推送模式，对于Zookeeper那种通过tcp长连接来维会节约很多资源，就算有很大量的节点更新也不会让Nacos出现太多的性能瓶颈，在Nacos客户端如果接收到了udp消息会返回一个Ack,如果一定时间Ncaos-Server，没有接收到ACK,那么Nacos-Server会继续重试（一共会重试10次），当超过一定的次数之后，就不在重发了，虽然通过upd不能保证数据真正的到订阅者，但是Ncaos还有定时轮训作为兜底，不需要担心数据不会更新情况。
Nacos通过这两种手段，既保证了实时性，又保证了数据不会漏掉的安全性