一、什么是nacos

Nacos的主要特点：

Nacos 帮助您更敏捷和容易地构建、交付和管理微服务平台。Nacos 提供了一组简单易用的特性集，帮助您快速实现动态服务发现、服务配置、服务元数据及流量管理。

1、服务注册与发现

Nacos 提供基于 DNS 和基于 RPC 的服务发现，即能被用来支持 https/http 的服务注册与发现，也支持 RPC 如 dubbo 的服务注册与发现。

2、动态配置服务

动态修改配置并实时生效，这种服务能够让我们的服务拥有更多的灵活性，不需要重启服务即可做到配置实时生效，非常适合于“配置优先”的服务开发。

动态配置服务能够以中心化、外部化和动态化的方式管理所有环境的配置。动态配置消除了配置变更时重新部署应用和服务的需要。配置中心化管理让实现无状态服务更简单，也让按需弹性扩展服务更容易。

Nacos 提供了一个简洁易用的 UI (控制台样例 Demo) 帮助您管理所有的服务和应用的配置。Nacos 还提供包括配置版本跟踪、金丝雀发布、一键回滚配置以及客户端配置更新状态跟踪在内的一系列开箱即用的配置管理特性，帮助您更安全地在生产环境中管理配置变更和降低配置变更带来的风险。

3、服务发现和服务健康监测

Nacos 支持基于 DNS 和基于 RPC 的服务发现。服务提供者使用 原生 SDK、OpenAPI、或一个独立的 Agent TODO 注册 Service 后，服务消费者可以使用 DNS TODO 或 HTTP&API 查找和发现服务。

Nacos 提供对服务的实时的健康检查，阻止向不健康的主机或服务实例发送请求。Nacos 支持传输层 (PING 或 TCP)和应用层 (如 HTTP、MySQL、用户自定义）的健康检查。 对于复杂的云环境和网络拓扑环境中（如 VPC、边缘网络等）服务的健康检查，Nacos 提供了 agent 上报模式和服务端主动检测 2 种健康检查模式。Nacos 还提供了统一的健康检查仪表盘，帮助您根据健康状态管理服务的可用性及流量。

4、动态 DNS 服务

动态 DNS 服务支持权重路由，更容易地实现中间层负载均衡、更灵活的路由策略、流量控制以及数据中心内网的简单 DNS 解析服务。动态 DNS 服务还能让您更容易地实现以 DNS 协议为基础的服务发现，以帮助您消除耦合到厂商私有服务发现 API 上的风险。

Nacos 提供了一些简单的 DNS APIs TODO 帮助您管理服务的关联域名和可用的 IP:PORT 列表.

5、服务及其元数据管理

Nacos 能让您从微服务平台建设的视角管理数据中心的所有服务及元数据，包括管理服务的描述、生命周期、服务的静态依赖分析、服务的健康状态、服务的流量管理、路由及安全策略、服务的 SLA 以及最首要的 metrics 统计数据。

二、快速开始：启动nacos服务（单机模式）

1、下载源码或者安装包

安装包地址：https://github.com/alibaba/nacos/releases

解压后进入nacos/bin目录

输入命令启动服务

linux：sh startup.sh -m standalone

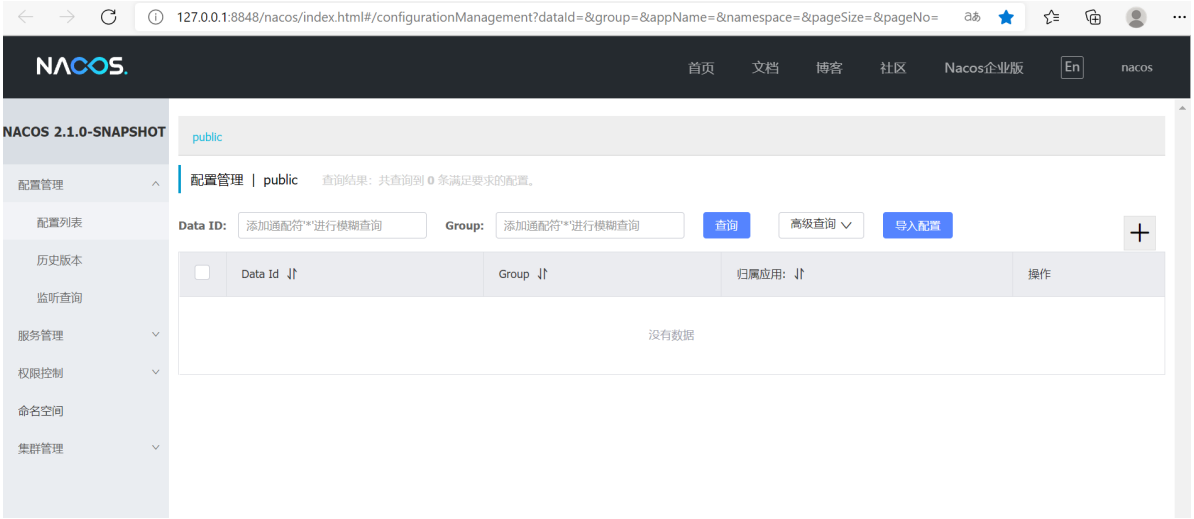
windows：cmd startup.cmd -m standlone（单机模式启动）

控制台启动下，看到"Nacos started successfully in stand alone mode.”后表示服务已启动

```
2021-10-08 14:55:47,698 INFO Creating filter chain: any request, [org.springframework.security.web.context.request.async
WebAsyncManagerIntegrationFilter@5c534b5b, org.springframework.security.web.context.SecurityContextPersistenceFilter@5e
048149, org.springframework.security.web.header.HeaderWriterFilter@4fc142ec, org.springframework.security.web.csrf.CsrfF
ilter@6ba7383d, org.springframework.security.web.authentication.logout.LogoutFilter@702b06fb, org.springframework.securi
ty.web.savedrequest.RequestCacheAwareFilter@3d5790ea, org.springframework.security.web.servletapi.SecurityContextHolderA
wareRequestFilter@57402ba1, org.springframework.security.web.authentication.AnonymousAuthenticationFilter@396639b, org.s
pringframework.security.web.session.SessionManagementFilter@29eda4f8, org.springframework.security.web.access.ExceptionT
ranslationFilter@710d89e2]

2021-10-08 14:55:48,072 INFO Initializing ExecutorService 'taskScheduler'

2021-10-08 14:55:48,104 INFO Exposing 16 endpoint(s) beneath base path '/actuator'

2021-10-08 14:55:48,278 INFO Tomcat started on port(s): 8848 (http) with context path '/nacos'

2021-10-08 14:55:48,283 INFO Nacos started successfully in stand alone mode. use embedded storage
```

2、访问nacos页面

nacos默认使用8848端口，可通过http://127.0.0.1:8848/nacos/index.html进入自带的控制台界面，默认用户名/密码是nacos/nacos



三、nacos作为配置中心和注册中心

1、添加pom依赖

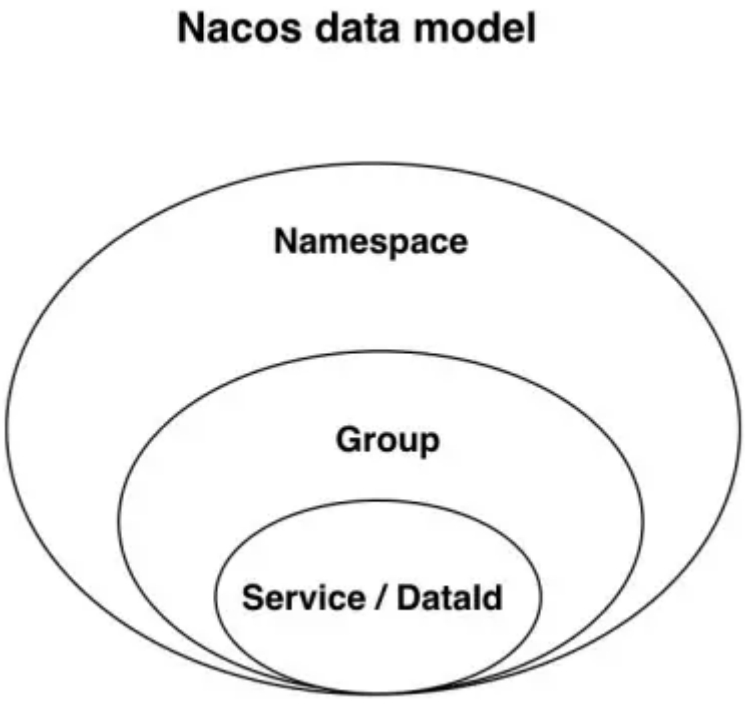
```xml
<!-- nacos config-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
<!-- SpringCloud ailibaba nacos-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2、yml文件配置

```yaml
server:
  port: 8801
spring:
  application:
    name: demoproject-service
  cloud:
    nacos:

      discovery:
        #server-addr: 10.51.23.206:1111 #服务注册中心地址
        server-addr: 10.51.23.233:8848 #服务注册中心地址
        namespace: a6bbe3fd-d57b-4b70-9fae-81b81be10817
       # group: test  #2.1.0的版本discovery 的group不好用
      config:
        #server-addr: 10.51.23.206:1111 #配置中心地址
        server-addr: 10.51.23.233:8848 #配置中心地址
        file-extension: yaml #指定yaml格式的配置
        namespace: a6bbe3fd-d57b-4b70-9fae-81b81be10817
        group: test
```

在实际开发和测试中，我们经常要切换不同的开发环境，比如从开发版切换到测试版，这时就可以使用nacos的namespace进行隔离，在团队中每个成员也可以公用一个nacos服务，通过开辟个人的namespace空间的方式，达到共享nacos资源，毕竟每个团队成员都开启单独的nacos服务，不仅耗费资源，也浪费时间。namespace 指定命名空间，如果不是public，那么必须指定命名空间的id.

注意：不同命名空间的服务不能相互调用



3、添加注解

在启动类上添加注解@EnableDiscoveryClient，也可以不加

4、配置中心只需要做三件事：1、存储配置项。2、提供接口返回配置项。3、通过接口通知应用放弃之前的环境，重新根据配置项生成一个新的环境

四、如何为nacos配置mysql数据库

1.找到nacos/conf目录下的nacos-mysql.sql脚本导入数据库

2.找到nacos/conf/application.properties 文件，解开以下

注释，并把账号和密码改成自己数据库的密码，然后重启nacos服务

`spring.datasource.platform=mysql`

`db.num=1`

`db.url.0=jdbc:mysql://127.0.0.1:3306/nacos?characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&useUnicode=true&useSSL=false&serverTimezone=UTC`

`db.user.0=nacos`

`db.password.0=nacos`

3.新建一个命名空间，然后刷新本地数据库的tenant_info表，就可以看到刚刚新建的命名空间了

4.可以通过nacos提供的api去查询实例的相关信息Open API 指南 (nacos.io)

五、通过 Spring Cloud 原生注解 `@RefreshScope` 实现配置自动更新（基于spring-boot 2.2.5，spring-cloud 2.1.0）

```
@RestController
@RequestMapping("/config")
@RefreshScope
public class ConfigController {

    @Value("${useLocalCache:false}")
    private boolean useLocalCache;

    @RequestMapping("/get")
    public boolean get() {
        return useLocalCache;
    }
}
```

1.首先通过调用 Nacos Open API 向 Nacos Server 发布配置：dataId 为`nacos-service.yaml`，内容为`useLocalCache=true`

http://127.0.0.1:8848/nacos/v1/cs/configs?dataId=nacos-service.yaml&group=prod&content=useLocalCache: true&tenant=a81ad0f0-eafd-4310-a786-8c4bd25eb141&type=yaml

2.运行 NacosConfigApplication，调用 curl http://localhost:8080/config/get，返回内容是 true

3.再次通过调用 Nacos Open API 向 Nacos Server 发布配置：dataId 为`nacos-service.yaml`，内容为useLocalCache=false

http://127.0.0.1:8848/nacos/v1/cs/configs?dataId=nacos-service.yaml&group=prod&content=useLocalCache: true&tenant=a81ad0f0-eafd-4310-a786-8c4bd25eb141&type=yaml

4.再次访问`http://localhost:8080/config/get`，此时返回内容为`false`，说明程序中的`useLocalCache`值已经被动态更新了

注意事项:

（1）若在bootstrap.yml文件里没有配置namespace和group，需要去掉namespace和group属性，不可以置空

（2）若调用Nacos Open API发布配置，spring.profiles.active指定的话，需要在boostrap.yml或者在application.yml里面配置：

```
spring:
  profiles:
    active: xxx
```

在 Nacos Spring Cloud 中，`dataId` 的完整格式如下：

```
${prefix}-${spring.profiles.active}.${file-extension}
```

- `prefix` 默认为 `spring.application.name` 的值，也可以通过配置项 `spring.cloud.nacos.config.prefix`来配置。
- `spring.profiles.active` 即为当前环境对应的 profile，详情可以参考 Spring Boot文档。 **注意：当 `spring.profiles.active` 为空时，对应的连接符 `-` 也将不存在，dataId 的拼接格式变成 `${prefix}.${file-extension}`**
- `file-exetension` 为配置内容的数据格式，可以通过配置项 `spring.cloud.nacos.config.file-extension` 来配置。目前只支持 `properties` 和 `yaml` 类型。

（3）如果文件是.yml文件格式，配置中心必须是.yaml后缀

5.RefreshScope 的实现原理

（1）当ScopedProxyMode 为TARGET_CLASS 的时候会给当前创建的bean 生成一个代理对象，会通过代理对象来访问，每次访问都会创建一个新的对象，可以看出，使用的是scope

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Scope("refresh")
@Documented
public @interface RefreshScope {

   ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET_CLASS;

}
```

（2）Object get(String name, ObjectFactory<?> objectFactory); 这个方法帮助我们来创建一个新的bean ，也就是说，@RefreshScope 在调用 刷新的时候会使用此方法来给我们创建新的对象，这样就可以通过spring 的装配机制将属性重新注入了，也就实现了所谓的动态刷新

```
public interface Scope {

    Object get(String name, ObjectFactory<?> objectFactory);

    @Nullable
    Object remove(String name);

    void registerDestructionCallback(String name, Runnable callback);

    @Nullable
    Object resolveContextualObject(String key);

    @Nullable
    String getConversationId();

}
```

（3）怎么处理老的对象，怎么创建新的对象

RefreshScope extends GenericScope, GenericScope implements Scope，通过查看代码，是GenericScope 实现了 Scope 最重要的 get(String name, ObjectFactory<?> objectFactory) 方法，在GenericScope 里面 包装了一个内部类 BeanLifecycleWrapperCache 来对加了 @RefreshScope 从而创建的对象进行缓存，使其在不刷新时获取的都是同一个对象。（这里你可以把BeanLifecycleWrapperCache 想象成为一个大Map 缓存了所有@RefreshScope 标注的对象）

知道了对象是缓存的，所以在进行动态刷新的时候，只需要清除缓存，重新创建就好了

```
// RefreshScope 内部代码
@ManagedOperation(description = "Dispose of the current instance of all beans "
      + "in this scope and force a refresh on next method execution.")
public void refreshAll() {
   super.destroy();
   this.context.publishEvent(new RefreshScopeRefreshedEvent());
}
```

```
// GenericScope 里的方法
//进行对象获取，如果没有就创建并放入缓存
@Override
public Object get(String name, ObjectFactory<?> objectFactory) {
   BeanLifecycleWrapper value = this.cache.put(name,
         new BeanLifecycleWrapper(name, objectFactory));
   this.locks.putIfAbsent(name, new ReentrantReadWriteLock());
   try {
      return value.getBean();
   }
   catch (RuntimeException e) {
      this.errors.put(name, e);
      throw e;
   }
}
//缓存了所有@RefreshScope  标注的对象
private static class BeanLifecycleWrapperCache {

   private final ScopeCache cache;

   BeanLifecycleWrapperCache(ScopeCache cache) {
      this.cache = cache;
   }

   public BeanLifecycleWrapper remove(String name) {
      return (BeanLifecycleWrapper) this.cache.remove(name);
   }

   public Collection<BeanLifecycleWrapper> clear() {
      Collection<Object> values = this.cache.clear();
      Collection<BeanLifecycleWrapper> wrappers = new
LinkedHashSet<BeanLifecycleWrapper>();
      for (Object object : values) {
         wrappers.add((BeanLifecycleWrapper) object);
      }
      return wrappers;
   }
   //进行缓存的数据清理
   @Override
    public void destroy() {
        List<Throwable> errors = new ArrayList<Throwable>();
        Collection<BeanLifecycleWrapper> wrappers = this.cache.clear();
        for (BeanLifecycleWrapper wrapper : wrappers) {
            try {
                Lock lock = this.locks.get(wrapper.getName()).writeLock();
                lock.lock();
                try {
                    wrapper.destroy();
                }
                finally {
                    lock.unlock();
                }
            }
            catch (RuntimeException e) {
                errors.add(e);
            }
        }
        if (!errors.isEmpty()) {
```

```
            throw wrapIfNecessary(errors.get(0));
        }
        this.errors.clear();
    }
```

（4）总结：需要动态刷新的类标注@RefreshScope 注解

@RefreshScope 注解标注了@Scope 注解，并默认了ScopedProxyMode.TARGET_CLASS; 属性，此属性的功能就是在创建一个代理，在每次调用的时候都用它来调用GenericScope get 方法来获取对象

如属性发生变更会调用 RefreshScope refreshAll() 进行缓存清理方法调用，并发送刷新事件通知 ->GenericScope 真正的 清理方法destroy() 实现清理缓存

在下一次使用对象的时候，会调用GenericScope get(String name, ObjectFactory<?> objectFactory) 方法尝试从缓存加载，如果没有就创建一个新的对象，并存入缓存中，此时就实现了动态刷新。

六、心跳机制

1.AbstractAutoServiceRegistration实现了ApplicationListener接口，我们知道spring的监听机制当服务器启动后会调用ApplicationListener的onApplicationEvent方法，所以，nacos客户端是在项目启动后通过onApplicationEvent方法将本服务的实例信息发送给nacos的服务端的

```
public abstract class AbstractAutoServiceRegistration<R extends Registration>
      implements AutoServiceRegistration, ApplicationContextAware,
      ApplicationListener<WebServerInitializedEvent> {
   ......省略
   @Override
   @SuppressWarnings("deprecation")
   public void onApplicationEvent(WebServerInitializedEvent event) {
      bind(event);
   ......省略
   }
```

2.服务实例发送心跳,在client这一侧是心跳的发起源，进入NacosNamingService，可以发现，只有注册服务实例的时候才会构造心跳包组装心跳包BeatInfo,注册nacos实例，默认每隔5秒发送一次心跳

```
   @Override
    public void registerInstance(String serviceName, String groupName, Instance
instance) throws NacosException {

        if (instance.isEphemeral()) {
            BeatInfo beatInfo = new BeatInfo();
            beatInfo.setServiceName(NamingUtils.getGroupedName(serviceName,
groupName));
            beatInfo.setIp(instance.getIp());
            beatInfo.setPort(instance.getPort());
            beatInfo.setCluster(instance.getClusterName());
            beatInfo.setWeight(instance.getWeight());
            beatInfo.setMetadata(instance.getMetadata());
            beatInfo.setScheduled(false);
            long instanceInterval = instance.getInstanceHeartBeatInterval();
            beatInfo.setPeriod(instanceInterval == 0 ?
DEFAULT_HEART_BEAT_INTERVAL : instanceInterval);

            beatReactor.addBeatInfo(NamingUtils.getGroupedName(serviceName,
groupName), beatInfo);
```

```
        }

        serverProxy.registerService(NamingUtils.getGroupedName(serviceName,
groupName), groupName, instance);
    }
```

3.该方法内部主要是将BeatInfo放入Map中,executorService.schedule->创建一个定时任务

```
public void addBeatInfo(String serviceName, BeatInfo beatInfo) {
    NAMING_LOGGER.info("[BEAT] adding beat: {} to beat map.", beatInfo);
     //dom2Beat 是一个concurrenthashmap, 当nacos服务关闭时， 会用到它来销毁定时任务
    dom2Beat.put(buildKey(serviceName, beatInfo.getIp(), beatInfo.getPort()),
beatInfo);
    executorService.schedule(new BeatTask(beatInfo), 0, TimeUnit.MILLISECONDS);
    MetricsMonitor.getDom2BeatSizeMonitor().set(dom2Beat.size());
}
```

4.sendBeat就是请求了nacos服务端的API->/instance/beat接口，只返回了一个心跳间隔时长，将这个返回值用于client设置定时任务间隔，表示完成了此次心跳发送任务，再创建一个定时任务，这个心跳检测就能一直不间断进行下去，直到nacos服务销毁,至此心跳检测客服端流程走完

```
class BeatTask implements Runnable {

    BeatInfo beatInfo;

    public BeatTask(BeatInfo beatInfo) {
        this.beatInfo = beatInfo;
    }

    @Override
    public void run() {
      // 用于控制心跳停止发送行为
        if (beatInfo.isStopped()) {
            return;
        }
        long result = serverProxy.sendBeat(beatInfo);
        long nextTime = result > 0 ? result : beatInfo.getPeriod();
        executorService.schedule(new BeatTask(beatInfo), nextTime,
TimeUnit.MILLISECONDS);
    }
```

服务器端总结:

当客户端调用reqApi向服务器发送Http PUT心跳请求，URI是/nacos/v1/ns/instance/beat，对应的controller就是InstanceController的beat方法。如果是参数beat信息的话，说明是第一次发起心跳，则会带有服务实例信息，因为发起心跳成功则服务端会返回下次不要带beat信息的参数，这样客户端第二次就不会携带beat信息了。如果实例不存在，并且客户端也没有发送BeatInfo包，这时会返回给客户端一个实例不存在的错误；如果Instance实例不存在，但客户端发送了BeatInfo包，这时能拿到BeatInfo里的信息，beat方法就会创建Instance对象，并调用ServiceManager类的registerInstance方法注册服务实例。

Nacos Server会开启一个定时任务来检查注册服务的健康情况，对于超过15秒没收到客户端的心跳实例会将它的 healthy属性置为false，此时当客户端不会将该实例的信息发现，如果某个服务的实例超过30秒没收到心跳，则剔除该实例，30秒后再发送心跳时就会出现Instance不存在的情况。如果剔除的实例恢复，发送心跳则会恢复。（ClientBeatCheckTask中的run方法主要做两件事心跳时间超过15秒则设置

该实例信息为不健康状况和心跳时间超过30秒则删除该实例信息）

七、基于权重的负载均衡

Nacos提供了权重配置来控制访问频率，权重越大则访问频率越高
a.Nacos控制台可以设置实例的权重值

b.同集群内的多个实例，权重越高被访问的频率越高

c.权重设置为0则完全不会被访问

eg：如把nacos-service:8801服务权重调整为0，此时nacos-service:8801服务不承担用户请求，这时做停机操作对用户是无感知操作，我们可以做版本升级，升级结束后权重调小点(如：0.01)，对小部分用户开放等没问题再把权重调大。这样操作对用户是无感知的，平滑升级非常优雅。

RestTemplate是远程调用Http的工具，支持本地负载均衡，配合Ribbon一起使用,是对java底层http的封装，使用RestTemplata用户可以不再关注底层的连接建立，并且RestTemplata不仅支持Rest规范，还可以定义返回值对象类型。

feign客户端是一个web声明式http远程调用工具，提供了接口和注解方式进行调用，是对Ribbon的封装,可以以Java接口注解的方式调用Http请求，而不用像Java中通过封装HTTP请求报文的方式直接调用。Feign通过处理注解，将请求模板化，当实际调用的时候，传入参数，根据参数再应用到请求上，进而转化成真正的请求，支持负载均衡。

首先提供两个服务端，二者提供的服务一模一样，只有端口号不一样

1.第一个服务提供者

```
@RestController
public class LoadBalanceController {

        @Value("${server.port}")
        private String port;

        @RequestMapping(value = "/test/{string}", method = RequestMethod.GET)
        public String echo(@PathVariable String string) {
            System.out.println(string);

            return "Nacos Discovery " + "server port : " + port + "==="+string;
        }

}
```

2. 第二个服务提供者

```
@RestController
public class LoadBalanceController {

        @Value("${server.port}")
        private String port;

        @RequestMapping(value = "/test/{string}", method = RequestMethod.GET)
        public String echo(@PathVariable String string) {
            System.out.println(string);

            return "Nacos Discovery " + "server port : " + port + "==="+string;
        }
```

```
}
```

3.定义一个消费者，作用是调用服务提供者：

```
@SpringBootApplication
@EnableDiscoveryClient
public class GatewayDemoApplication {
    @LoadBalanced
    //就是RestTemplate发起一个请求，这个请求被LoadBalancerInterceptor给拦截了，拦截后将请求的地址中的服务逻辑名转为具体的服务地址，然后继续执行请求
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
    public static void main(String[] args) {
        SpringApplication.run(GatewayDemoApplication.class, args);
    }
    @RestController
    public class TestController {
        @Autowired
        private RestTemplate restTemplate;

        @RequestMapping(value = "/echos", method = RequestMethod.GET)
        public String echo() {
//          System.out.println("str:"+str);
// nacos-service代表服务端的服务名，/echo/mxb接口名
            return restTemplate.getForObject("http://nacos-service/echo/mxb",
String.class);
        }
    }
}
```

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
gateway-demo	DEFAULT_GROUP	1	1	1	false	详情 | 示例代码 | 订阅者 | 删除
nacos-service	DEFAULT_GROUP	1	2	2	false	详情 | 示例代码 | 订阅者 | 删除

IP	端口	临时实例	权重	健康状态	元数据	操作
10.51.23.233	8802	true	1	true	preserved.register.source=SPRING_CLOUD	编辑 下线
10.51.23.233	8801	true	1	true	preserved.register.source=SPRING_CLOUD	编辑 下线

4.然后我们访问http://localhost:8803/echos会发现结果为：

Nacos Discovery server port : 8801===hello或者

Nacos Discovery server port : 8802===hello

注意事项：

1.Nacos默认是轮询。此时在控制台上修改服务的权重是无效的，仍然是轮询，nacos源码里是有ribbon负载均衡策略实现的，但是nacos可能没有将自己重写的负载均衡策略启用。我们将NacosRule 注册成为Bean，替换默认的Rule，在消费者页面加上如下代码，这样就可以支持在控制台上手动配置权重了。

```
        @Bean
        @Scope(value = "prototype")
        public IRule loadBalanceRule()
        {
            return new NacosRule();
        }
```

配置完权重后，我们重新调用接口，会发现权重大的访问频率高。

2.也可以直接在配置文件bootstrap.yml里修改权重：

```
cloud:
  nacos:
    discovery:
      #server-addr: 10.51.23.206:1111 #服务注册中心地址
      server-addr: 10.51.23.233:8848 #服务注册中心地址
      weight: 2
```

基于权重的负载均衡算法：

```
public class NacosRule extends AbstractLoadBalancerRule {
   private static final Logger LOGGER =
LoggerFactory.getLogger(NacosRule.class);

   @Autowired
   private NacosDiscoveryProperties nacosDiscoveryProperties;

   @Override
   public Server choose(Object key) {
      try {
         String clusterName = this.nacosDiscoveryProperties.getClusterName();
         DynamicServerListLoadBalancer loadBalancer =
(DynamicServerListLoadBalancer) getLoadBalancer();
          //服务名称
         String name = loadBalancer.getName();

         NamingService namingService = this.nacosDiscoveryProperties
               .namingServiceInstance();
         List<Instance> instances = namingService.selectInstances(name, true);
         if (CollectionUtils.isEmpty(instances)) {
            LOGGER.warn("no instance in service {}", name);
            return null;
         }

         List<Instance> instancesToChoose = instances;
         if (StringUtils.isNotBlank(clusterName)) {
            List<Instance> sameClusterInstances = instances.stream()
                  .filter(instance -> Objects.equals(clusterName,
                        instance.getClusterName()))
                  .collect(Collectors.toList());
            if (!CollectionUtils.isEmpty(sameClusterInstances)) {
               instancesToChoose = sameClusterInstances;
            }
            else {
               LOGGER.warn(
```

```
                    "A cross-cluster call occurs，name = {}, clusterName = {},
instance = {}",
                    name, clusterName, instances);
            }
         }

         Instance instance =
ExtendBalancer.getHostByRandomWeight2(instancesToChoose);

         return new NacosServer(instance);
      }
      catch (Exception e) {
         LOGGER.warn("NacosRule error", e);
         return null;
      }
   }
```

```
public static Instance getHostByRandomWeight2(List<Instance> instances) {
   return getHostByRandomWeight(instances);
}
```

```
protected static Instance getHostByRandomWeight(List<Instance> hosts) {
    NAMING_LOGGER.debug("entry randomWithWeight");
    if (hosts == null || hosts.size() == 0) {
        NAMING_LOGGER.debug("hosts == null || hosts.size() == 0");
        return null;
    }

    Chooser<String, Instance> vipChooser = new Chooser<String, Instance>
("www.taobao.com");

    NAMING_LOGGER.debug("new Chooser");

    List<Pair<Instance>> hostsWithWeight = new ArrayList<Pair<Instance>>();
    for (Instance host : hosts) {
        if (host.isHealthy()) {// 非健康节点不会被选中，组装Pair的列表，包含健康节点的权
重和Host信息
            hostsWithWeight.add(new Pair<Instance>(host, host.getWeight()));
        }
    }
    NAMING_LOGGER.debug("for (Host host : hosts)");
    //刷新需要的数据，具体包括三部分：所有健康节点权重求和、计算每个健康节点权重占比、组织递增数
组
    vipChooser.refresh(hostsWithWeight);
    NAMING_LOGGER.debug("vipChooser.refresh");
    return vipChooser.randomWithWeight();
}
```

```
public void refresh() {
    Double originWeightSum = (double) 0;
    //所有健康节点权重求和originWeightSum
    for (Pair<T> item : itemsWithWeight) {

        double weight = item.weight();
```

```
        //ignore item which weight is zero.see test_randomWithWeight_weight0 in
ChooserTest
        //weight小于等于0将会被删除
        if (weight <= 0) {
            continue;
        }

        items.add(item.item());
        // 值如果无穷大
        if (Double.isInfinite(weight)) {
            weight = 10000.0D;
        }

        // 值如果为非数字值
        if (Double.isNaN(weight)) {
            weight = 1.0D;
        }

        // 累加权重总和
        originWeightSum += weight;
    }
    // 计算每个健康节点权重占比exactWeights数组
    double[] exactWeights = new double[items.size()];
    int index = 0;
    for (Pair<T> item : itemsWithWeight) {
        double singleWeight = item.weight();
        //ignore item which weight is zero.see test_randomWithWeight_weight0 in
ChooserTest
        if (singleWeight <= 0) {
            continue;
        }
        // 每个节点权重的占比
        exactWeights[index++] = singleWeight / originWeightSum;
    }
//组织递增数组weights，每个元素值为数组前面元素之和
    weights = new double[items.size()];
    double randomRange = 0D;
    for (int i = 0; i < index; i++) {
        weights[i] = randomRange + exactWeights[i];
        randomRange += exactWeights[i];
    }

    double doublePrecisionDelta = 0.0001;

    if (index == 0 || (Math.abs(weights[index - 1] - 1) < doublePrecisionDelta))
{
        return;
    }
    throw new IllegalStateException("Cumulative Weight caculate wrong , the sum
of probabilities does not equals 1.");
}
```

```
public T randomWithWeight() {
    Ref<T> ref = this.ref;
    //产生0到1区间的随机数
    double random = ThreadLocalRandom.current().nextDouble(0, 1);
    //二分法查找数组中接近的值
```

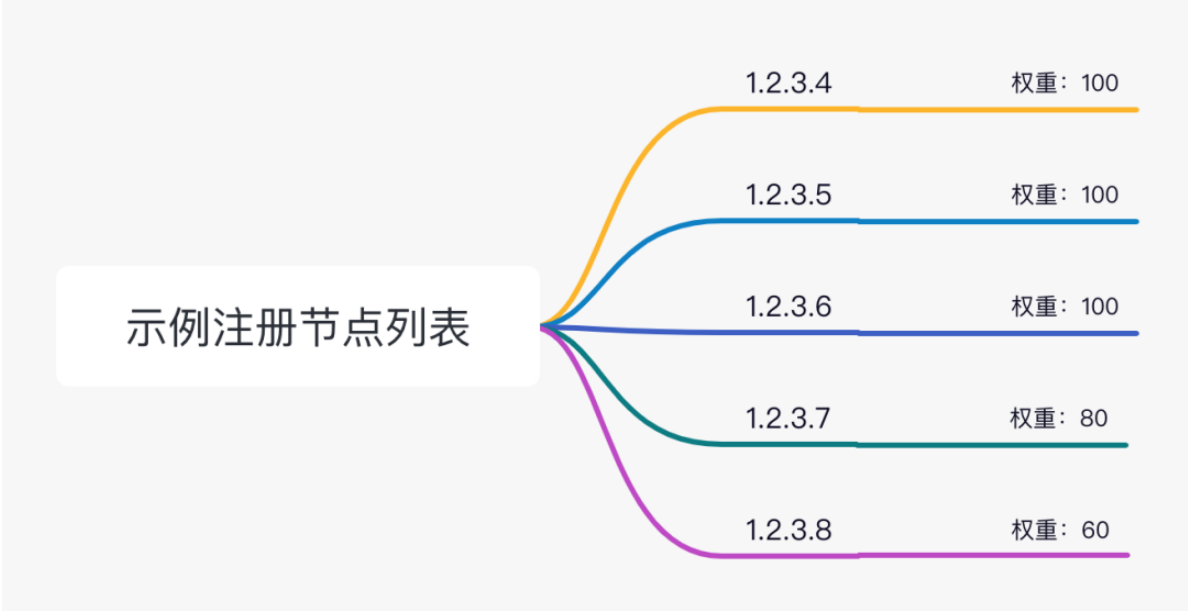
```
    int index = Arrays.binarySearch(ref.weights, random);
    //没有命中返回插入数组理想索引值
    if (index < 0) {
        index = -index - 1;
    } else {
    //命中直接返回选中节点
        return ref.items.get(index);
    }

    if (index >= 0 && index < ref.weights.length) {
        if (random < ref.weights[index]) {
            return ref.items.get(index);
        }
    }

    /* This should never happen, but it ensures we will return a correct
     * object in case there is some floating point inequality problem
     * wrt the cumulative probabilities. */
    return ref.items.get(ref.items.size() - 1);
}
```

以上就是基于权重的负载均衡算法。

通过例子看一下：

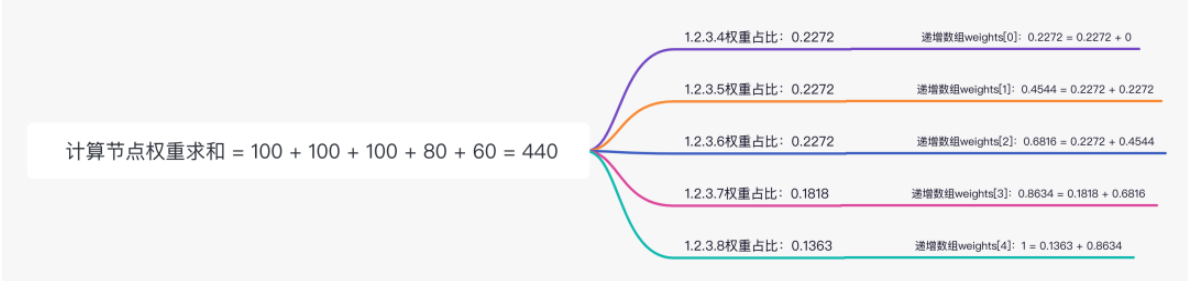
(1)节点列表

假设注册了5个节点，每个节点的权重如下：



2.组织递增数组

目的在于形成weights数组，该数组元素取值[0~1]范围，元素逐个递增，计算过程如下图示。另外注意非健康节点或者权重小于等于0的不会被选择。



3.随机算法

通过生成[0~1]范围的随机数，通过二分法查找递增数组weights[]接近的index，再从注册节点列表中返回