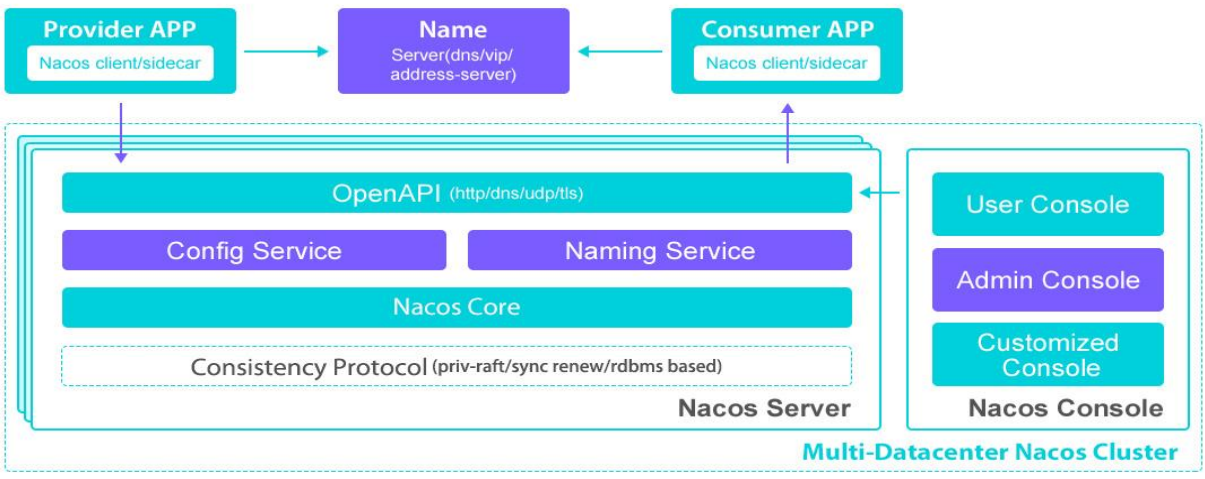
Spring Cloud – Nacos 与 Eureka 技术对比

1、背景

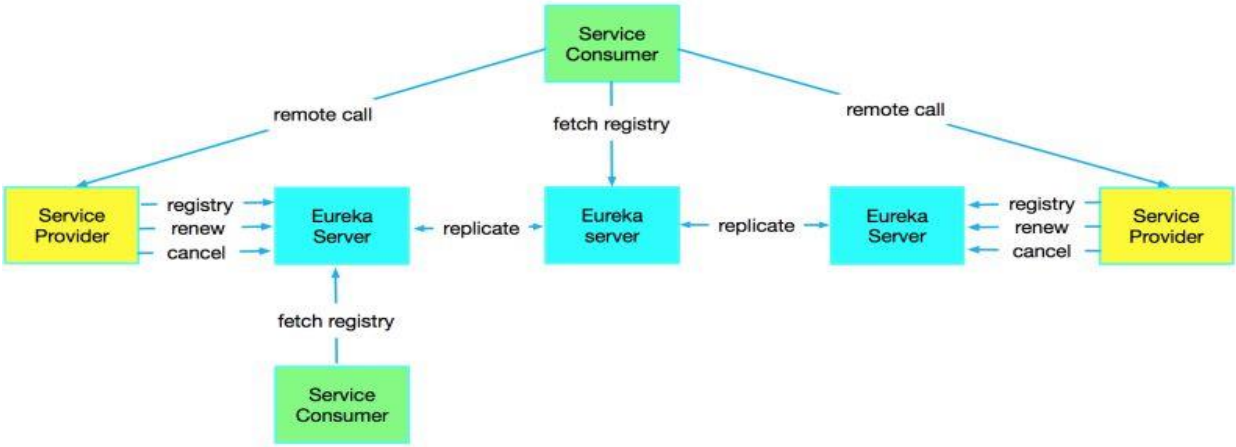
Nacos 与 Eureka 都是微服务服务注册中心，Eureka 出现于 2014 年，与 Spring cloud 生态深度结合，但于 2018 年 7 月 Netflix 公司决定停止开源 2.X 版本的 Eureka，对日后的更新与维护造成很大困难。而 Nacos 是阿里集团于 2018 年开源发布的，携带了阿里大规模生产服务的经验，提供给用户一个新的选择。

2、服务架构图

1、Spring cloud Nacos



2、Spring cloud Eureka



3、性能测试

3.1、Spring cloud Nacos

1、测试环境

指标	参数
机器	CPU 8 核，内存 16G
集群规模	三节点
Nacos 版本	服务端：Nacos2.0.0-ALPHA2，客户端：Nacos2.0.0-ALPHA2

2、启动参数

3、测试场景

以下测试场景都是服务发现重要接口： 吞吐量（TPS），响应时间（RT）

服务发现注册实例的能力

相关 API：**NamingService.registerInstance(String serviceName, Instance instance)**

施压机数量	每台线程数	平均 TPS	平均 RT	最小 RT	最大 RT	CPU 使用率
1	100	7256.32	13.14	0.39	2522.25	80%
2	50	16418.04	5.8	0.41	3906.77	90%
5	20	26784.84	3.6	0.38	1606.41	90%

服务发现查询实例的能力

相关 API：**NacosNamingService.getAllInstances(String serviceName, boolean subscribe)**

施压机数量	每台线程数	平均 TPS	平均 RT	最小 RT	最大 RT	CPU 使用率
1	100	12998.46	7.54	0.55	213.86	40%
2	50	12785.01	7.93	0.38	900.48	40%
2	100	18451.78	10.63	0.6	829.42	45%
5	20	30680.48	3.12	0.46	1138.38	50%

服务发现注销实例的能力

相关 API：**NacosNamingService.deregisterInstance(String serviceName, String ip, int port)**

施压机数量	每台线程数	平均 TPS	平均 RT	最小 RT	最大 RT	CPU 使用率
1	100	9614.96	9.88	0.41	1115.27	70%
2	50	22252.07	4.28	0.39	856.1	90% -> 60%
5	20	29393.8	2.55	0.42	741.09	90% -> 60%

4、结果分析

相较 Nacos1.X 版本，注销性能总体提升至少 2 倍，在服务端机能减半的情况下，服务实例数基本一致的情况下，TPS 仍能做到 2 倍左右的提高。 关于 CPU 由 90% 降低为 60% 的场景， 是由于随着注销的的服务和实例增多，重复注销的操作变得频繁，未命中服务和实例的操作会被快速返回且操作量小，因此 CPU 下降、TPS 相对注册略高。

5、测试结论

Nacos2 服务发现性能测试都是针对重点功能，通过对 3 节点规模集群进行压测，可以看到接口性能负载和容量，以及对比相同/类似场景下 Nacos1.X 版本的提升。

1. 压测时服务及实例容量达到百万级，集群运行持续稳定，达到预期；（该场景没有计算频繁变更导致的频繁推送内容，仅单纯计算容量上线，附带推送的真实场景将在下轮压测报告中给出）
2. 注册/注销实例 TPS 达到 26000 以上，总体较 Nacos1.X 提升至少 2 倍，接口达到预期；
3. 查询实例 TPS 能够达到 30000 以上，总体较 Nacos1.X 提升 3 倍左右，接口达到预期；

6、参考

https://nacos.io/zh-cn/docs/nacos-naming-benchmark.html

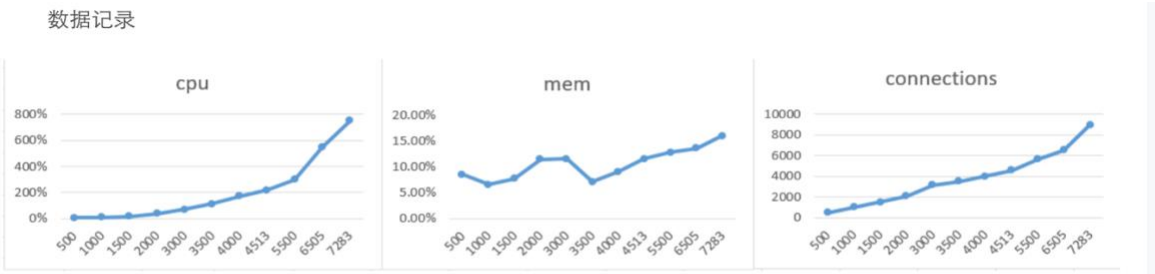
https://nacos.io/zh-cn/docs/nacos2-naming-benchmark.html

3.2、Spring cloud Eureka

1、测试环境

工具选项	描述	配置/能力
测试机器	CentOS release 5.4 (Final) 3 台	8c16g，open files：65535，max user processes：65535
Eureka 服务端	单机部署,boot 版本(Dalston.SR5)	-XX:MaxHeapSize=4g（默认）
Wireshark	Windows 平台抓包工具	抓取 HTTP、TCP 等报文内容、协议相关信息
UAV 监控	公司自研监控平台	实时监控采集应用性能指标

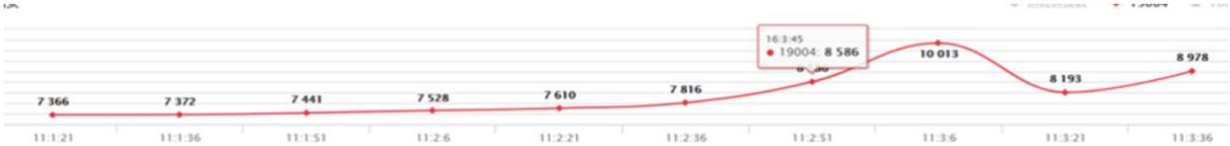
2、测试结果



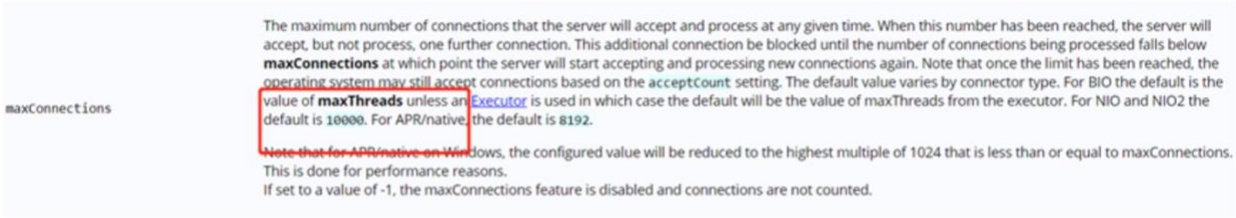
可以看出到实例注册数到==7000多==时候，连接数不稳定飙到10000左右，同时此时客户端开始大量报错超时，服务端开始拒绝连接：

```
... 2 more
2018-04-12 16:17:01.945  INFO   --- [     Thread-789] o.a.http.impl.client.DefaultHttpClient   : I/O exce
2018-04-12 16:17:01.949  INFO   --- [     Thread-789] o.a.http.impl.client.DefaultHttpClient   : Retrying
Exception in thread "Thread-551" com.sun.jersey.api.client.ClientHandlerException: java.net.SocketTimeout
        at com.sun.jersey.api.client.ClientResponse.close(ClientResponse.java:684)
        at com.gantry.test.FetchResign.run(FetchResign.java:69)
        at java.lang.Thread.run(Thread.java:748)
Caused by: java.net.SocketTimeoutException: Read timed out
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
        at java.net.SocketInputStream.read(SocketInputStream.java:171)
        at java.net.SocketInputStream.read(SocketInputStream.java:141)
        at org.apache.http.impl.io.AbstractSessionInputBuffer.fillBuffer(AbstractSessionInputBuffer.java:
        at org.apache.http.impl.io.SocketInputBuffer.fillBuffer(SocketInputBuffer.java:82)
        at org.apache.http.impl.io.AbstractSessionInputBuffer.read(AbstractSessionInputBuffer.java:209)
        at org.apache.http.impl.io.ChunkedInputStream.read(ChunkedInputStream.java:189)
        at org.apache.http.impl.io.ChunkedInputStream.read(ChunkedInputStream.java:213)
        at org.apache.http.impl.io.ChunkedInputStream.close(ChunkedInputStream.java:315)
        at org.apache.http.conn.BasicManagedEntity.streamClosed(BasicManagedEntity.java:166)
        at org.apache.http.conn.EofSensorInputStream.checkClose(EofSensorInputStream.java:228)
        at org.apache.http.conn.EofSensorInputStream.close(EofSensorInputStream.java:172)
```

此时连接数截图详细，可以清楚看到 conns 达到 10000 阀值后接着下降：



此时结果和预期猜测一样。MaxConnection=10000，且大于AcceptCount=100时，Tomcat会触发拒绝连接。



而我们使用的spring boot版本使用内嵌Tomcat版本8.5
接着改了服务端tomcat配置改成如下配置：

```
@Override
 public void customize(Connector connector) {

    Http11NioProtocol protocol = (Http11NioProtocol) connector.getProtocolHandler();
    // 设置最大连接数
    protocol.setMaxConnections(20000);// 10000
    // 设置最大线程数
    protocol.setMaxThreads(5000);// 200
    // protocol.setConnectionTimeout(1); // 20s
    protocol.setAcceptCount(1000); // 100
    // protocol.setKeepAliveTimeout(1);
    System.out.println("protocol.get:" + protocol.getMaxConnections());
 }
```

再次从新开始一轮测试，数据记录如下：

实例数	7100	8000
cpu	749%	790%
mem	17.7	18%
conn	12007	13690
Threads	1982	1997

可以看出，在修改了 tomcat 对应配置，将最大连接数调至 20000，线程数调至 5000 后，Eureka 可注册的实例数突破了 7000，连接数也突破了 10000，实例数注册到 8000 后才开始报错，看出此时 cpu 已经接近满负载，操作系统本身调度已经压到极限，于是结束了本次测试。

3、结论

Eureka Server 服务实例注册量的负载值和操作系统、应用容器本身对应的配置相关，调

整操作系统可打开最大文件句柄、进程数，调整应用容器相关最大连接数、线程数、NIO 服务器模型引入等等手段都可提高我们应用服务整体吞吐量。

4、参考

http://springcloud.cn/view/31

3.3、比对结果

上两种测试均在 8 核 16G 服务器上运行，从结果可看出在高负载和多注册应用上 Nacos 的性能更加，能容纳更多的实例，并且 Eureka 实例数过高时会导致 CPU 高负荷运载，而 Nacos 的测试 CPU 最高只处于 90%状态。

4、优点

1、 Nacos 开源方为阿里，有专门的社区和教程，维护与更新及时，并带有可视化控制界面。

2、 Nacos 同时支持 CP（一致性与分区容错性）与 AP（可用性与分区容错性），根据服务注册选择临时和永久来决定走 AP 模式还是 CP 模式，而 eureka 只支持 AP。

3、 Nacos 还支持管理台动态配置服务，效果如同 Spring cloud Eureka + Spring cloud Config。

5、功能差异

模块	Nacos	Eureka	说明
注册中心	✅	✅	服务治理基本功能，负责服务中心化注册
配置中心	✅	❌	Eureka 需要配合 Config 实现配置中心，且不提供管理界面
动态刷新	✅	❌	Eureka 需要配合 MQ 实现配置动态刷新，Nacos 采用 Netty 保持 TCP 长连接实时推送
可用区 AZ	✅	✅	对服务集群划分不同区域，实现区域隔离，并提供容灾自动切换
分组	✅	❌	Nacos 可用根据业务和环境进行分组管理
元数据	✅	✅	提供服务标签数据，例如环境或服务标识
权重	✅	❌	Nacos 默认提供权重设置功能，调整承载流量压力
健康检查	✅	✅	Nacos 支持由客户端或服务端发起的健康检查，Eureka 是由客户端发起心跳
负载均衡	✅	✅	均提供负责均衡策略，Eureka 采用 Ribion
管理界面	✅	❌	Nacos 支持对服务在线管理，Eureka 只是预览服务状态

6、安装与使用教程

https://github.com/XiaoTiJun/ExperienceSharing/blob/master/JAVA/nacos%26eureka/nacos/installNacos.md

附录（CAP 定理）📎

CAP 定理，指的是在一个分布式系统中， Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。

一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）

可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）

分区容忍性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在 C 和 A 之间做出选择。

CAP 原则的精髓就是要么 AP，要么 CP，要么 AC，但是不存在 CAP。如果在某个分布式系统中数据无副本， 那么系统必然满足强一致性条件， 因为只有独一数据，不会出现数据不一致的情况，此时 C 和 P 两要素具备，但是如果系统发生了网络分区状况或者宕机，必然导致某些数据不可以访问，此时可用性条件就不能被满足，即在此情况下获得了 CP 系统，但是 CAP 不可同时满足。因此在进行分布式架构设计时，必须做出取舍。