# SpringCloud Alibaba 学习圣经，10万字实现SpringCloud 自由

原创 40岁老架构师尼恩 技术自由圈 2023-03-15 17:24 发表于湖北

技术自由圈
疯狂创客圈（技术自由架构圈）：一个 技术狂人、技术大神、高性能 发烧友 圈子。圈内...
23篇原创内容

公众号

**40岁老架构师尼恩的掏心窝：**

现在**拿到offer超级难**，甚至连面试电话，一个都搞不到。

尼恩的技术社群中（50+），很多小伙伴凭借 “左手云原生+右手大数据 +SpringCloud Alibaba 微服务“三大绝活，拿到了offer，并且是非常优质的offer，据说**年终奖都足足18个月** ，非常令人羡慕。

问题是：“左手云原生+右手大数据 +SpringCloud Alibaba 微服务“ **内容非常多，实操的环境非常复杂，底层原理很深。**

**米饭要一口一口的吃，不能急。**在这里，尼恩从架构师视角出发，左手云原生+右手大数据 +SpringCloud Alibaba 微服务 核心原理做一个宏观的介绍。

> 由于内容确实太多， 所以写多个pdf 电子书：
>
> (1) 《 **Docker 学习圣经** 》**PDF** （V1已经完成）
>
> (2) 《 **SpringCloud Alibaba 微服务 学习圣经** 》PDF （V1已经完成）
>
> (3) 《 **K8S 学习圣经** 》**PDF** （coding.....）
>
> (4) 《 **flink + hbase 学习圣经** 》**PDF** （planning .....）

以上学习圣经，并且后续会持续升级，从V1版本一直迭代发布。就像咱们的《尼恩Java面试宝典》一样， 已经迭代到V60啦。

40岁老架构师尼恩的掏心窝：通过一系列的学习圣经，带大家穿透“左手云原生+右手大数据 +SpringCloud Alibaba 微服务“ ，实现技术 自由 ，走向颠覆人生，让大家不迷路。

本PDF 《SpringCloud Alibaba 微服务 学习圣经 》PDF的 V1版本，后面会持续迭代和升级。供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

以上学习圣经的 基础知识是 尼恩的 高并发三部曲，建议在看 学习圣经之前，一定把尼恩的 Java 高并发三部曲过一遍，切记，切记。

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，点这里获取

---

《 SpringCloud Alibaba 微服务 学习圣经 》PDF 封面

《 **SpringCloud Alibaba** 微服务 学习圣经 》目录

- merge
- rename
- remove_field：移除字段
- join
- geoip
- ruby
- urldecode
- kv
- useragent

- 数据输出
- stdout
- file
- kafka
- elasticseach

- Kibana查看应用日志
- 1 查看应用日志
- 2 如何搜索日志
- 3 如何查看指定时间的应用日志
- 4 如何定位错误日志
- 5 如何展开显示日志

- es的安全认证
- 配置 elk的ElastAlert 预警插件
- Prometheus+Grafana 检测预警
- 什么是性能可观测
- 系统监控的核心指标
- 系统性能指标
- 资源性能指标

- 什么是prometheus
- prometheus的运行原理
- prometheus主要特点

- 什么是 Grafana
- Prometheus的体系结构
- Prometheus+Grafana分层架构
- Promcthcus体系涉及的组件
- 如何收集度量值

- 指标类型
- 计数器
- 仪表盘
- 直方图
- Summary

- 指标摘要及聚合
  - 指标摘要
  - 指标聚合
- 一键安装 prometheus
  - - bridge网络管理
    - 创建库
  - docker编排文件
  - 一键安装 prometheus的脚本
  - 进入 prometheus
  - 进入 grafana
- Prometheus+Grafana监控SpringBoot项目JVM信息
  - SpringBoot项目配置JVM采集
  - Prometheus配置
- 配置grafana监控Linux系统
  - 使用 Exporter 收集指标
  - inux直接安装node_exporter
  - 使用Docker容器安装node_exporter
  - 创建一个任务定时扫描暴露的指标信息
  - 创建仪表盘grafna
  - 导入Dashboard
- 选择数据源为Prometheus
- 配置grafana监控SpringBoot应用
  - 主要步骤
  - 找jvm的 dashboard
  - JVM Quarkus 面板
- Prometheus数据模型
  - time-series 时间序列值
  - Sample样本值
  - metrics name指标名称
  - label标签
  - Notation(符号)
  - TSDB时序数据库
- 度量指标类型
  - Counter(计数器)类型
  - Gauge(计量器、测量器)
  - Histogram(柱状图、直方图)
  - Summary
  - Summary 和 Histogram 的区分
- 学习 PromQL
  - 数据模型

- PromQL 入门
- HTTP API
- 告警和通知
- 配置告警规则
- 使用 Alertmanager 发送告警通知
- 服务发现
- 为什么需要服务发现
- prometheus目前支持的服务发现类型
- 基于文件的服务发现方式
- file_sd_configs
- 基于consul 的服务发现
- 什么是基于consul的服务发现
- Prometheus配置
- 基于eureka的服务发现
- eureka 客户端暴露出 prometheus 端口
- prometheus配置文件
- 基于nacos的服务发现
- docker 编排文件
- 生产的配置文件
- 修改prometheus配置文件
- 修改springboot项目配置文件
- 全方位 Springcloud 性能调优
- Servlet 容器 优化
- Zuul配置 优化
- Feign 配置优化
- hystrix配置 优化
- ribbon 优化
- 高质量实操：SpringCloud 高并发实战案例
- 1、超高并发10Wqps秒杀实操
- 2、超高并发100Wqps车联网实操
- 3、N多其他的超高并发实操项目
- 技术自由的实现路径：
- - 实现你的 架构自由：
- 实现你的 响应式 自由：
- 实现你的 spring cloud 自由：
- 实现你的 linux 自由：
- 实现你的 网络 自由：
- 实现你的 分布式锁 自由：
- 实现你的 王者组件 自由：
- 实现你的 面试题 自由：

- 参考文献

## 当前版本V1

此书会持续迭代，最新版本，请关注公众号：技术自由圈

## 一键导入 SpringCloud 开发环境（地表最强）

*SpringCloud + docker 学习环境非常复杂，尼恩搞这个 前前后后起码* ***折腾了一周 ,应该还不止****，*

*其中，很多头疼的工作，包括linux内核升级、磁盘扩容等等， 苦不堪言。*

现在把这个环境，以虚拟机box镜像的方式，导出来直接给大家，

大家一键导入后，直接享受SpringCloud + docker 的实操，可以说，爽到不要不要的。

*以上软件和 尼恩个人的虚拟机box镜像，可以找尼恩获取。*

### 环境准备

硬件总体要求，可以参考尼恩的本地硬件情况：

**1 硬件要求。**

本文硬件总体要求如下表：

| 序号 | 硬件 | 要求 |
| --- | --- | --- |
| 1 | CPU | 至少2核 |
| 2 | 内存 | 至少16G |
| 3 | 硬盘 | 至少100G磁盘空间 |

**2 本地虚拟机环境**

| 软件 | 版本 |
| --- | --- |
| Win | win10以上 |
| virtual box | 6以上 |
| vagrant | 2以上 |

### 一键导入OR自己折腾

大家一键导入尼恩的虚拟机环境，里边zookeeper、nacos、docker、k8s等组件都已经预装，直接享受SpringCloud + docker 的实操，可以说，爽到不要不要的。

当然，如果想**自己折腾**， 也可以按照的步骤来哈。

| 工欲善其事 必先利其器 |
| --- |
| 地表最强 开发环境：vagrant+java+springcloud+redis+zookeeper镜像下载(&制作详解)(https://www.cnblogs.com/crazymakercircle/p/14194688.html) |
| 地表最强 热部署：java SpringBoot SpringCloud 热部署 热加载 热调试(https://www.cnblogs.com/crazymakercircle/p/12077373.html) |
| 地表最强 发请求工具（再见吧， PostMan ）：IDEA HTTP Client（史上最全）(https://www.cnblogs.com/crazymakercircle/p/14317222.html) |
| 地表最强 PPT 小工具：屌炸天，像写代码一样写PPT(https://www.cnblogs.com/crazymakercircle/p/14326975.html) |
| 无编程不创客，无编程不创客，一大波编程高手正在疯狂创客圈交流、学习中！找组织，GO(https://www.cnblogs.com/crazymakercircle/p/9904544.html) |

---

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，点这里获取

### 随书源码 **crazy-springcloud** 脚手架涉以及基础中间件

本PDF的随书源码仓库，就是尼恩的 《Java高并发核心编程 卷3 加强版》一书的源码

也是尼恩的一个微服务开发脚手架 crazy-springcloud，其大致的模块和功能具体如下：

```
crazymaker-server     --  根项目
│  ├─cloud-center     --  微服务的基础设施中心
│  │  ├─cloud-eureka      --  注册中心
│  │  ├─cloud-config      --  配置中心
│  │  ├─cloud-zuul        --  网关服务
│  │  ├─cloud-zipkin      -- 监控中心
│  ├─crazymaker-base  -- 公共基础依赖模块
│  │  ├─base-common     -- 普通的公共依赖，如 utils 类的公共方法
│  │  ├─base-redis      -- 公共的 redis 操作模块
│  │  ├─base-zookeeper  -- 公共的 zookeeper 操作模块
│  │  ├─base-session    -- 分布式 session 模块
│  │  ├─base-auth       -- 基于 JWT + SpringSecurity 的用户凭证与认证模块
│  │  ├─base-runtime    -- 各 provider 的运行时公共依赖，装配的一些通用 Spring IOC Bean 实例
│  ├─crazymaker-uaa   --业务模块: 用户认证与授权
│  │  ├─uaa-api        -- 用户 DTO、Constants 等
│  │  ├─uaa-client     --  用户服务的 Feign 远程客户端
│  │  ├─uaa-provider   -- 用户认证与权限的实现，包含controller 层、service层、dao层的代码实现
│  ├─crazymaker-seckill  --业务模块: 秒杀练习
│  │  ├─seckill-api        -- 秒杀 DTO、Constants 等
│  │  ├─seckill-client     -- 秒杀服务的 Feign 远程调用模块
│  │  ├─seckill-provider   -- 秒杀服务核心实现，包含controller层、service层、dao层的代码实现
│  ├─crazymaker-demo    --业务模块: 练习演示
│  │  ├─demo-api        -- 演示模块的 DTO、Constants 等
│  │  ├─demo-client     -- 演示模块的 Feign 远程调用模块
│  │  ├─demo-provider   -- 演示模块的核心实现，包含controller层、service层、dao层的代码实现
```

基于 crazy-springcloud 脚手架（其他的脚手架也类似）的微服务开发和自验证过程中，涉及到的基础中间件大致如下：

（1）ZooKeeper （虚拟机中已经预装）

ZooKeeper 是一个分布式的、开放源码的分布式协调应用程序，是大数据框架 Hadoop 和 Hbase 的重要组件。在分布式应用中，它能够高可用地提供很多保障数据一致性的基础能力：分布式锁、选主、分布式命名服务等。

在 crazy-springcloud 脚手架中，高性能分布式 ID 生成器用到了 ZooKeeper。有关其原理和使用，请参见《Netty Zookeeper Redis 高并发实战》一书。

（2）Redis （虚拟机中已经预装）

Redis 是一个高性能的缓存数据库。在高并发的场景下，Redis 可以对关系数据库起到很好的缓冲作用；在提高系统的并发能力和响应速度方面，Redis 举足轻重和至关重要。crazy-

springcloud 脚手架的分布式 Session 用到了 Redis。有关 Redis 的原理和使用，还是请参见《Netty Zookeeper Redis 高并发实战》一书。

（3）Eureka （虚拟机中已经预装）

Eureka 是 Netflix 开发的服务注册和发现框架，本身是一个 REST 服务提供者，主要用于定位运行在 AWS（Amazon 云）的中间层服务，以达到负载均衡和中间层服务故障转移的目的。SpringCloud 将它集成在其子项目 spring-cloud-netflix 中，以实现 SpringCloud 的服务注册和发现功能。

（4）SpringCloud Config （虚拟机中已经预装）

SpringCloud Config 是 SpringCloud 全家桶中最早的配置中心，虽然在生产场景中，很多的企业已经使用 Nacos 或者 Consul 整合型的配置中心替代了独立的配置中心，但是 Config 依然适用于 SpringCloud 项目，通过简单的配置即可使用。

（5） Zuul

Zuul 是 Netflix 开源网关，可以和 Eureka、Ribbon、Hystrix 等组件配合使用，SpringCloud 对 Zuul 进行了整合与增强，使用其作为微服务集群的内部网关，负责对给集群内部各个 provider 服务提供者进行 RPC 路由和请求过滤。

以上中间件的端口配置，以及部分安装和使用视频，大致如下表所示。

| 中间件 | 链接地址 |
| --- | --- |
| Linux Redis 安装（带视频） | https://www.cnblogs.com/crazymakercircle/p/11985983.html |
| Linux Zookeeper 安装（带视频） | https://www.cnblogs.com/crazymakercircle/p/12006500.html |
| ....完整的开发环境的准备工作，-> | 请去疯狂创客圈 博客园 总入口(https://www.cnblogs.com/crazymakercircle/p/9904544.html) |
| 。。。。 | |

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，点这里获取

ok，是不是很简单

> 源码请参见 《Java高并发核心编程 卷3 加强版》一书源码**虽然SpringCloud 入门很简单，但是原理很复杂， 而且掌握SpringCloud 原理，是成为核心工厂师的必备知识**

## 微服务 分布式系统的架构12次演进

在进入实操之前，咱们来点微服务 分布式系统的架构演进。

在尼恩的 《Java高并发核心编程 卷3 加强版》 一书中，对亿级流量的系统架构，做了一个内部的分析。

本文《SpringCloud Alibaba学习圣经 》与之相配合，介绍一下微服务 分布式系统的架构演进。

《Java 高并发核心编程 卷3 加强版》的 亿级流量的系统架构， 大家更加要好好看看，可以结合起来看。

## 微服务 分布式系统的几个基础概念

在介绍架构之前，为了避免部分读者对架构设计中的一些概念不了解，下面对几个微服务分布式系统中最基础的概念进行介绍。

**1）什么是分布式?**

系统中的多个模块在不同服务器上部署，即可称为分布式系统，如Tomcat和数据库分别部署在不同的服务器上，或两个相同功能的Tomcat分别部署在不同服务器上。

**2）什么是高可用?**

系统中部分节点失效时，其他节点能够接替它继续提供服务，则可认为系统具有高可用性。

**3）什么是集群?**

一个特定领域的软件部署在多台服务器上并作为一个整体提供一类服务，这个整体称为集群。

如Zookeeper中的Master和Slave分别部署在多台服务器上，共同组成一个整体提供集中配置服务。

在常见的集群中，客户端往往能够连接任意一个节点获得服务，并且当集群中一个节点掉线时，其他节点往往能够自动的接替它继续提供服务，这时候说明集群具有高可用性。

**4）什么是负载均衡?**

请求发送到系统时，通过某些方式把请求均匀分发到多个节点上，使系统中每个节点能够均匀的处理请求负载，则可认为系统是负载均衡的。

**5）什么是正向代理和反向代理?**

系统内部要访问外部网络时，统一通过一个代理服务器把请求转发出去，在外部网络看来就是代理服务器发起的访问，此时代理服务器实现的是正向代理；

当外部请求进入系统时，代理服务器把该请求转发到系统中的某台服务器上，对外部请求来说，与之交互的只有代理服务器，此时代理服务器实现的是反向代理。

简单来说，正向代理是代理服务器代替系统内部来访问外部网络的过程，反向代理是外部请求访问系统时通过代理服务器转发到内部服务器的过程。

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，点这里获取

## 架构演进1：单机架构

在网站最初时，应用数量与用户数都较少，可以把Java应用（主要是Tomcat承载）和数据库部署在同一台服务器上。

浏览器往发起请求时，首先经过DNS服务器（域名系统）把域名转换为实际IP地址10.102.4.1，浏览器转而访问该IP对应的Tomcat。

具体的架构图，如下：

**架构瓶颈：**

随着用户数的增长，Tomcat和数据库之间竞争资源，单机性能不足以支撑业务。

### 架构演进**2**：引入缓存架构

随着 吞吐量的提升， 不得不引入引入缓存架构。通过缓存能把绝大多数请求在读写数据库前拦截掉，大大降低数据库压力。

含：本地缓存和分布式缓存

在Tomcat同服务器上或同JVM中增加本地缓存，并在外部增加分布式缓存，缓存热门商品信息或热门商品的html页面等。

**涉及的技术包括：**

使用caffeine 作为本地缓存，使用Redis作为分布式缓存

**架构瓶颈：**

（1）这里会涉及缓存穿透/击穿、缓存雪崩、热点数据集中失效等问题

（2）这里会涉及缓存一致性的问题

（3）这里会涉及 hotkey的问题

有关上面问题的解决方案：

请参见尼恩的 《100Wqps三级缓存组件实操》+ 《100Wqps Caffeine 底层源码、架构实操实操》

## 架构演进3：接入层引入反向代理实现负载均衡

接下来，缓存抗住了大部分的访问请求，服务层Tomcat上还是吞吐量低，响应逐渐变慢，需要进行架构演进。

在多台服务器上分别部署Tomcat，使用反向代理软件（Nginx）把请求均匀分发到每个Tomcat中。

此处假设Tomcat最多支持100个并发，Nginx最多支持50000个并发，那么理论上Nginx把请求分发到500个Tomcat上，就能抗住50000个并发。

**其中涉及的技术包括：**Nginx、HAProxy，

两者都是工作在网络第七层的反向代理软件，主要支持http协议，还会涉及session共享、文件上传下载的问题。

### 架构演进**4**：数据库读写分离

反向代理使服务层的并发量大大增加，但并发量的增长也意味着：更多请求会穿透到数据库，数据库最终成为瓶颈。

**数据库如何高并发?**

简单的方案：把数据库划分为读库和写库，读库可以有多个，

**读库和写库之间，如何实现数据一致性?**

简单的方案：可以通过DB的同步机制，把写库的数据同步到读库，对于需要查询最新写入数据场景，可通过在缓存中多写一份，通过缓存获得最新数据。

**数据库读写分离的架构如下：**

其中涉及的技术包括：shardingjdbc ，它是数据库中间件，可通过它来组织数据库的分离读写和分库分表，客户端通过它来访问下层数据库，还会涉及数据同步，数据一致性的问题。

有关上面分库分表解决方案：

请参见尼恩的 《10Wqps 日志平台实操》， 对分库分表方案，做了非常详解的架构介绍，并且在实操维度，对其中的核心的组件分布式ID，结合雪花id源码，百度id源码，shardingjdbc id源码，做了深入骨髓的介绍。

架构演进**5**：数据库按业务分库

业务逐渐变多，不同业务之间的访问量差距较大，不同业务直接竞争数据库，相互影响性能。

随着用户数的增长，单机的写库会逐渐会达到性能瓶颈。

把不同业务的数据保存到不同的数据库中，使业务之间的资源竞争降低，对于访问量大的业务，可以部署更多的服务器来支撑。

有关上面分库分表解决方案：

请参见尼恩的 《10Wqps 日志平台实操》， 对分库分表方案，做了非常详解的架构介绍，并且在实操维度，对其中的核心的组件分布式ID，结合雪花id源码，百度id源码，shardingjdbc id源码，做了深入骨髓的介绍。

### 架构演进6：使用LVS或F5接入层负载均衡

随着吞吐量大于5W，接入层Nginx扛不住了

由于瓶颈在Nginx，因此无法LVS或F5来实现多个Nginx的负载均衡。

图中的LVS和F5是工作在网络第四层的负载均衡解决方案，区别是：

(1）LVS是软件，运行在操作系统内核态，可对TCP请求或更高层级的网络协议进行转发，因此支持的协议更丰富，并且性能也远高于Nginx，可假设单机的LVS可支持几十万个并发的请求转发；

(2）F5是一种负载均衡硬件，与LVS提供的能力类似，性能比LVS更高，但价格昂贵。

如果不是财大气粗的guoqi，推荐使用LVS。

由于LVS是单机版的软件，若LVS所在服务器宕机则会导致整个后端系统都无法访问，因此需要有备用节点。

LVS 如何高可用呢?

可使用keepalived软件模拟出虚拟IP，然后把虚拟IP绑定到多台LVS服务器上，浏览器访问虚拟IP时，会被路由器重定向到真实的LVS服务器

当主LVS服务器宕机时，keepalived软件会自动更新路由器中的路由表，把虚拟IP重定向到另外一台正常的LVS服务器，从而达到LVS服务器高可用的效果。

### 架构演进**7**：通过**DNS**轮询实现机房间的负载均衡

由于LVS也是单机的，随着并发数增长到几十万时，LVS服务器最终会达到瓶颈，此时用户数达到千万甚至上亿级别，用户分布在不同的地区，与服务器机房距离不同，导致了访问的延迟会明显不同。

此时，可以使用 DNS 进行负载均衡：在DNS服务器中可配置一个域名对应多个IP地址，每个IP地址对应到不同的机房里的虚拟IP。

当用户访问taobao时，DNS服务器会使用轮询策略或其他策略，来选择某个IP供用户访问。

此方式能实现机房间的负载均衡

至此，系统可做到机房级别的水平扩展，千万级到亿级的并发量都可通过增加机房来解决，系统入口处的请求并发量不再是问题。

问题是，光用DNS进行简单的 LVS负载均衡，是不够的。

**所以呢？大部分的大厂应用，都是采用 智能DNS + 接入层流量二次路由的模式， 具体的案例，可以来找尼恩进行交流，这里不做展开。主要是内容太多啦。**

## 架构演进8：引入NoSQL数据库和搜索引擎等技术

随着数据的丰富程度和业务的发展，检索、分析等需求越来越丰富，单单依靠数据库无法解决如此丰富的需求。

当数据库中的数据多到一定规模时，数据库就不适用于复杂的查询了，往往只能满足普通查询的场景。

对于统计报表场景，在数据量大时不一定能跑出结果，而且在跑复杂查询时会导致其他查询变慢

对于全文检索、可变数据结构等场景，数据库天生不适用，使用 elasticsearch 分布式搜索引擎解决。

如对于海量文件存储，可通过分布式文件系统hbase解决

对于全文检索场景，可通过搜索引擎如ElasticSearch解决，对于多维分析场景，可通过Kylin或Druid等方案解决。

当然，引入更多组件同时会提高系统的复杂度，不同的组件保存的数据需要同步，需要考虑一致性的问题，需要有更多的运维手段来管理这些组件等。

**接下来尼恩会讲 云原生+大数据的架构，就是介绍的这套方案。**

## 架构演进**9**：大应用拆分为微服务

引入更多组件解决了丰富的需求，业务维度能够极大扩充，随之而来的是一个应用中包含了太多的业务代码，业务的升级迭代、部署维护变得困难，效率低下。

解决方式是，进行 业务的解耦。按照业务板块来划分应用代码，使单个应用的职责更清晰，相互之间可以做到独立升级迭代。

不同的业务，可以解耦成不同的微服务。

这样的服务就是所谓的微服务，应用和服务之间通过HTTP、TCP或RPC请求等多种方式来访问公共服务，每个单独的服务都可以由单独的团队来管理。

此外，可以通过Dubbo、SpringCloud等框架实现服务治理、限流、熔断、降级等功能，提高服务的稳定性和可用性。

这时候应用之间可能会涉及到一些公共配置，可以通过分布式配置中心 Nacos来解决。

### 架构演进**10**：引入企业服务总线**ESB**对微服务进行编排

由于不同服务之间存在共用的模块，由微服务单独管理会导致相同代码存在多份，导致公共功能升级时全部应用代码都要跟着升级。

不同微服务的接口访问方式不同，微服务代码需要适配多种访问方式才能使用，此外，微服务访问微服务，微服务之间也可能相互访问，调用链将会变得非常复杂，逻辑变得混乱。

在微服务的基础上，以应用为单位，进行微服务的分组，并且引入企业服务总线ESB，对微服务进行编排，形成应用。

通过ESB统一进行访问协议转换，应用统一通过ESB来访问后端服务，服务与服务之间也通过ESB来相互调用，以此降低系统的耦合程度。

这种微服务编排为多个应用，公共服务单独抽取出来来管理，并使用企业消息总线来解除服务之间耦合问题的架构。

**接下来尼恩会讲 ESB架构，就是介绍的这套方案。**

### 架构演进11：引入容器化技术实现动态扩容和缩容

业务不断发展，应用和服务都会不断变多，应用和服务的部署变得复杂，同一台服务器上部署多个服务还要解决运行环境冲突的问题

此外，对于如大促这类需要动态扩缩容的场景，需要水平扩展服务的性能，就需要在新增的服务上准备运行环境，部署服务等，运维将变得十分困难。

目前最流行的容器化技术是Docker，最流行的容器管理服务是Kubernetes(K8S)，应用/服务可以打包为Docker镜像，通过K8S来动态分发和部署镜像。

Docker镜像可理解为一个能运行你的应用/服务的最小的操作系统，里面放着应用/服务的运行代码，运行环境根据实际的需要设置好。

把整个“操作系统”打包为一个镜像后，就可以分发到需要部署相关服务的机器上，直接启动Docker镜像就可以把服务起起来，使服务的部署和运维变得简单。

有关 Docker + Kubernetes(K8S) 的内容，请参见尼恩的电子书：

由于内容确实太多， 所以写多个pdf 电子书：

(1) **《 Docker 学习圣经 》PDF**

(2) **《 SpringCloud Alibaba 微服务 学习圣经 》**PDF

使用 Docker + Kubernetes(K8S) 后，在大促的之前，可以在现有的机器集群上划分出服务器来启动Docker镜像，增强服务的性能

大促过后就可以关闭镜像，对机器上的其他服务不造成影响。

### 架构演进12：以云平台承载系统

使用容器化技术后服务动态扩缩容问题得以解决，但是机器还是需要公司自身来管理，在非大促的时候，还是需要闲置着大量的机器资源来应对大促，机器自身成本和运维成本都极高，资源利用率低。

系统可部署到公有云上，利用公有云的海量机器资源，解决动态硬件资源的问题

在大促的时间段里，在云平台中临时申请更多的资源，**结合Docker和K8S来快速部署服务**，在大促结束后释放资源，真正做到按需付费，资源利用率大大提高，同时大大降低了运维成本。

所谓的云平台，就是把海量机器资源，通过统一的资源管理，抽象为一个资源整体

在云平台上可按需动态申请硬件资源（如CPU、内存、网络等），并且之上提供通用的操作系统，提供常用的技术组件（如Hadoop技术栈，MPP数据库等）供用户使用，甚至提供开发好的应用

用户不需要关心应用内部使用了什么技术，就能够解决需求（如音视频转码服务、邮件服务、个人博客等）。

在云平台中会涉及如下几个概念：

IaaS：基础设施即服务。对应于上面所说的机器资源统一为资源整体，可动态申请硬件资源的层面；PaaS：平台即服务。对应于上面所说的提供常用的技术组件方便系统的开发和维护；SaaS：软件即服务。对应于上面所说的提供开发好的应用或服务，按功能或性能要求付费。

至此：以上所提到的从高并发访问问题，到服务的架构和系统实施的层面都有了各自的解决方案。

### 架构演进的涉及的核心知识

通过以上架构的演进，可以看出：

（1）开发侧：重点的知识体系是 SpringCloud + Nginx 的基础架构；

有关 SpringCloud + Nginx的知识，请阅读本文《SpringCloud Alibaba学习圣经 》和与之相配合的《Java 高并发核心编程 卷3 加强版》

《Java 高并发核心编程 卷3 加强版》

（2）运维侧：重点的知识体系是 docker + k8s 的基础架构；

有关 docker + k8s 的知识，请阅读本文《docker 学习圣经》和与之相配合的《K8s 学习圣经》

**搞定这些，应对亿级流量，就具备了基础的知识底座。**

本文，聚焦 SpringCloud 的学习，主要是 SpringCloud Alibaba 的学习。

---

## SpringCloud netflix 入门

要了解 SpringCloud Alibaba ，先得了解 **SpringCloud netflix。**

为啥？SpringCloud Alibaba 仅仅是在 SpringCloud netflix 的基础上，替换了部分组件。 比如说注册中心，比如RPC组件。

**所以，咱们得从SpringCloud netflix 开始。**

SpringCloud Netflix全家桶是 Pivotal 团队提供的一整套微服务开源解决方案，包括服务注册与发现、配置中心、全链路监控、服务网关、负载均衡、断路器等组件，以上的组件主要通过对NetFilx的 NetFlix OSS 套件中的组件通过整合完成的，其中，比较重要的整合组件有:

（1）spring-cloud-netflix-Eureka 注册中心

（2）spring-cloud-netflix-hystrix RPC保护组件

（3）spring-cloud-netflix-ribbon 客户端负载均衡组件

（4）spring-cloud-netflix-zuul 内部网关组件

（6）spring-cloud-config 配置中心

SpringCloud 全家桶技术栈除了对 NetFlix OSS的开源组件做整合之外，还有整合了一些选型中立的开源组件。比如，SpringCloud Zookeeper 组件整合了 Zookeeper，提供了另一种方式的服务发现和配置管理。

SpringCloud 架构中的单体业务服务是基于 SpringBoot 应用进行启动和执行的。SpringBoot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。SpringCloud 利用 SpringBoot 是什么关系呢?

（1）首先 SpringCloud 利用 SpringBoot 开发便利性巧妙地简化了分布式系统基础设施的开发;

（2）其次 SpringBoot 专注于快速方便地开发单体微服务提供者，而 SpringCloud 解决的是各微服务提供者之间的协调治理关系;

（3）第三 SpringBoot 可以离开 SpringCloud 独立使用开发项目，但是 SpringCloud 离不开 SpringBoot，其依赖 SpringBoot 而存在。

最终，SpringCloud 将 SpringBoot 开发的一个个单体微服务整合并管理起来，为各单体微服务提供配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等基础的分布式协助能力。

### SpringCloud 开发脚手架

无论是单体应用还是分布式应用，如果从零开始开发，都会涉及很多基础性的、重复性的工作需要做，比如用户认证，比如 session 管理等等。有了开发脚手架，这块基础工作就可以省去，直接利用脚手架提供的基础模块，然后按照脚手架的规范进行业务模块的开发即可。

笔者看了开源平台的不少开源的脚手架，发现很少是可以直接拿来做业务模块开发的，或者封装的过于重量级而不好解耦，或者业务模块分包不清晰而不方便开发，所以，本着简洁和清晰的原则，笔者的发起的疯狂创客圈社群推出了自己的微服务开发脚手架 crazy-springcloud，其大致的模块和功能具体如下：

```
crazymaker-server     --  根项目
|  ├─cloud-center     --  微服务的基础设施中心
|  |  ├─cloud-eureka      --  注册中心
|  |  ├─cloud-config      --  配置中心
|  |  ├─cloud-zuul        --  网关服务
|  |  ├─cloud-zipkin      -- 监控中心
|  ├─crazymaker-base  -- 公共基础依赖模块
|  |  ├─base-common     -- 普通的公共依赖，如 utils 类的公共方法
|  |  ├─base-redis      -- 公共的 redis 操作模块
|  |  ├─base-zookeeper  -- 公共的 zookeeper 操作模块
|  |  ├─base-session    -- 分布式 session 模块
|  |  ├─base-auth       -- 基于 JWT + SpringSecurity 的用户凭证与认证模块
|  |  ├─base-runtime    -- 各 provider 的运行时公共依赖，装配的一些通用 Spring IOC Bean 实例
|  ├─crazymaker-uaa   --业务模块: 用户认证与授权
|  |  ├─uaa-api        -- 用户 DTO、Constants 等
|  |  ├─uaa-client     --  用户服务的 Feign 远程客户端
|  |  ├─uaa-provider   -- 用户认证与权限的实现，包含controller 层、service层、dao层的代码实现
|  ├─crazymaker-seckill  --业务模块: 秒杀练习
|  |  ├─seckill-api        -- 秒杀 DTO、Constants 等
|  |  ├─seckill-client     -- 秒杀服务的 Feign 远程调用模块
|  |  ├─seckill-provider   -- 秒杀服务核心实现，包含controller层、service层、dao层的代码实现
|  ├─crazymaker-demo    --业务模块: 练习演示
|  |  ├─demo-api        -- 演示模块的 DTO、Constants 等
|  |  ├─demo-client     -- 演示模块的 Feign 远程调用模块
|  |  ├─demo-provider   -- 演示模块的核心实现，包含controller层、service层、dao层的代码实现
```

在业务模块如何分包的问题上，实际上大部分企业都有自己的统一规范。crazy-springcloud 脚手架从职责清晰、方便维护、能快速导航代码的角度出发，将每一个业务模块，细分成以下三个子模块：

（1） {module}-api

此子模块定义了一些公共的 Constants 业务常量和 DTO 传输对象，该子模块既被业务模块内部依赖，也可能被依赖该业务模块的外部模块所依赖；

（2） {module}-client

此子模块定义了一些被外部模块所依赖的 Feign 远程调用客户类，该子模块是专供外部的模块，不能被内部的其他子模块所依赖；

（3） {module}-provider

此子模块是整个业务模块的核心，也是一个能够独立启动、运行的服务提供者（Application），该模块包含涉及到业务逻辑的 controller 层、service 层、dao 层的完整代码实现。

crazy-springcloud 微服务开发脚手架在以下两方面进行了弱化：

（1）在部署方面对容器的介绍进行了弱化，没有使用 Docker 容器而是使用 Shell 脚本。有多方面的原因：一是本脚手架初心是学习，使用 Shell 脚本而不是 Docker 去部署，方便大家学习 Shell 命令和脚本；二是 Java 和 Docker 其实整合得很好，学习非常容易，可以稍加配置就能做到一键发布，找点资料就可以掌握；三是部署和运维是一个专门的工作，生产环境的部署、甚至是整个自动化构建和部署的工作，实际上属于运维的专项工作，由专门的运维岗位人员去完成，而部署的核心仍然是 Shell 脚本，所以对于开发人员来说掌握 Shell 脚本才是重中之重。

（2）对监控软件的介绍进行了弱化。本书没有对链路监控、JVM性能指标、断路器监控软件的使用做专门介绍。有多方面的原因：一是监控的软件太多，如果介绍太全，篇幅又不够，介绍太少，大家又不一定用到；二是监控软件的使用大多是一些软件的操作步骤和说明，原理性的内容比较少，使用视频的形式会比文字形式知识传递的效果会更好。疯狂创客圈后续可能（但不一定）会推出一些微服务监控方面的教学视频供大家参考，请大家关注社群博客。不论如何，只要掌握了 SpringCloud 核心原理，对那些监控组件使用的掌握，对大家来说基本上都是一碟小菜。

## 启动**Eureka Server** 注册中心

Eureka 本身是 Netflix 开源的一款注册中心产品，并且 SpringCloud 提供了相应的集成封装，选择其作为注册中心的讲解实例，是出于以下的原因：

（1）Eureka 在业界的应用十分广泛（尤其是国外），整个框架也经受住了 Netflix 严酷生产环境的考验。

（2）除了 Eureka 注册中心，Netflix 的其他服务治理功能也十分强大，包括 Ribbon、Hystrix、Feign、Zuul 等组件，结合到一起组成了一套完整的服务治理框架，使得服务的调用、路由也变得异常容易。

那么，Netflix 和 SpringCloud 是什么关系呢?

Netflix 是一家互联网流媒体播放商，是美国视频巨头，访问量非常的大。也正是如此，Netflix 把整体的系统迁移到了微服务架构。并且，Netflix 就把它的几乎整个微服务治理生态中的组件，都开源贡献给了 Java 社区，叫做 Netflix OSS。

SpringCloud 是 Spring 背后的 Pivotal 公司（由 EMC 和 VMware 联合成立的公司）在 2015 年推出的开源产品，主要对 Netflix 开源组件的进一步封装，方便 Spring 开发人员构建微服务架构的应用。

SpringCloud Eureka 是 SpringCloud Netflix 微服务套件的一部分，基于 Netflix Eureka 做了二次封装，主要负责完成微服务实例的自动化注册与发现，这也是微服务架构中最为核心和基础的功能。

Eureka 所治理的每一个微服务实例，被称之为 Provider Instance (提供者实例)。每一个 Provider Instance 微服务实例包含一个 Eureka Client 客户端组件（相当于注册中心客户端组件），其主要的工作为:

（1）向 Eureka Server 完成 Provider Instance 的注册、续约和下线等操作，主要的注册信息包括服务名、机器 IP、端口号、域名等等。

（2）向 Eureka Server 获取 Provider Instance 清单，并且缓存在本地。

一般来说，Eureka Server 作为服务治理应用，会独立地部署和运行。一个 Eureka Server 注册中心应用在新建的时候，首先需要在pom.xml文件中添加上 eureka-server 依赖库。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
```

```
</dependency>
```

然后，需要在启动类中添加注解 @EnableEurekaServer，声明这个应用是一个Eureka Server，启动类的代码如下：

```
package com.crazymaker.springcloud.cloud.center.eureka;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
//在启动类中添加注解 @EnableEurekaServer
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

然后，在应用配置文件 application.yml 中，对 Eureka Server 的一些参数进行配置。一份基础的配置文件大致如下：

```
server:
  port: 7777
spring:
  application:
    name: eureka-server
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    service-url:
       #服务注册中心的配置内容，指定服务注册中心的位置
       defaultZone: ${SCAFFOLD_EUREKA_ZONE_HOSTS:http://localhost:7777/eureka/}
  instance:
hostname: ${EUREKA_ZONE_HOST:localhost}
```

```
  server:
    enable-self-preservation: true # 开启自我保护
eviction-interval-timer-in-ms: 60000 # 扫描失效服务的间隔时间（单位毫秒，默认是60*1000）即60秒
```

以上的配置文件中，包含了三类配置项：作为服务注册中心的配置项（eureka.server.*）、作为Provider 提供者的配置项（eureka.instance.*）、作为注册中心客户端组件的配置项（eureka.client.*），至于具体的原因，请参考《SpringCloud Nginx高并发核心编程》一书。

配置完成后，通过运行启动类 EurekaServerApplication 就可以启动 Eureka Server，然后通过浏览器访问 Eureka Server 的**控制台界面**（其端口为 server.port 配置项的值），大致如下图所示。

图：Eureka Server 的控制台界面

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，点这里获取

### 启动Config 配置中心

在采用分布式微服务架构的系统中，由于服务数量巨多，为了方便服务配置文件统一管理，所以需要分布式配置中心组件。如果各个服务的配置分散管理，则，上线之后配置的如何保持一致，将会是一个很头疼的问题。

所以，各个服务的配置定然需要集中管理。SpringCloud Config 配置中心是一个比较好的解决方案。使用SpringCloud Config配置中心，涉及到两个部分：

（1）config-server 服务端配置；（需要独立运行）

（2）config-client 客户端配置。（作为组件嵌入到Provider微服务提供者）

### **config-server** 服务

通过SpringCloud 构建一个 config-server 服务，大致需要三步。首先，在pom.xml中引入spring-cloud-config-server 依赖，大致如下：

```
 <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

其次，在所创建的 SpringBoot的程序主类上，添加@EnableConfigServer注解，开启Config Server 服务，代码如下：

```
@EnableConfigServer
@SpringBootApplication
public
class Application {

    public static void main(String[] args) {
        new  SpringApplicationBuilder(Application.class).web(true).run(args);
   }
}
```

第三步，设置属性文件的位置。SpringCloud Config 提供本地存储配置的方式。在bootstrap启动属性文件中，设置属性 spring.profiles.active=native，并且设置属性文件所在的位置，大致如下：

```
server:
  port: 7788  #配置中心端口
```

```yaml
spring:
  application:
    name: config-server    # 服务名称
  profiles:
    active: native    # 设置读取本地配置文件
  cloud:
    config:
      server:
        native:
          searchLocations: classpath:config/  #申明本地配置文件的存放位置
```

配置说明：

（1）spring.profiles.active=native，表示读取本地配置，而不是从git读取配置。

（2）search-locations=classpath:config/ 表示查找文件的路径，在类路径的config下。

服务端的配置规则：在配置路径下，以 {label}/{application}-{profile}.properties 的命令规范，放置对应的配置文件。上面实例，放置了以下配置文件：

分别对通用配置common、数据库配置db、缓存配置的相关属性，进行设置。Config 配置中心启动之后，使用 http:// $CONFIG-HOST$ :{CONFIG-PORT}/{application}/{profile}[/{label}] 的地址格式，可以直接访问所加载好的配置属性。

例如，访问示例中的 http://192.168.233.128:7788/crazymaker/redis/dev 地址，返回的配置信息如下图所示。

**特别说明：SpringCloud config-server 支持有多种配置方式，比如 Git，native，SVN 等。**虽然官方建议使用Git方式进行配置，这里没有重点介绍 Git方式，而是使用了本地文件的方式。有三个原因：

（1） 对于学习或者一般的开发来说，本地的文件的配置方式更简化；

（2） **生产环境建议使用 Nacos，集成注册中心和配置中心，更加方便和简单**；

## 微服务入门案例

在本书的配套源码 crazy-springcloud 脚手架中，设计三个 Provider 服务提供者：uaa-provider（用户账号与认证）、demo-provider （演示用途）、seckill-provider （秒杀服务），具体如下图所示。

图：本书的配套源码中的服务提供者

### **uaa-provider** 微服务提供者

首先，一个 Provider 服务提供者至少需要以下两个组件包依赖：SpringBoot WEB 服务组件、Eureka Client 客户端组件，大致如下：

```
<dependencies>
<!--SpringBoot WEB 服务组件 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>

    <!-- Eureka Client 客户端组件 -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>
```

SpringBoot WEB 服务组件用于提供 REST 接口服务，Eureka Client 客户端组件用于服务注册与发现。从以上的 Maven 依赖可以看出，在 SpringCloud 技术体系中，一个 Provider 服务提供者首先是一个 SpringBoot 应用，所以，在学习 SpringCloud 微服务技术之前，必须具备一些基本

的 SpringBoot 开发知识。然后，在 SpringBoot 应用的启动类上加上 @EnableDiscoveryClient 注解，用于启用 Eureka Client 客户端组件，启动类的代码如下：

```
package com.crazymaker.springcloud.user.info.start;
//...省略import
@SpringBootApplication
/*
 *  启用 Eureka Client 客户端组件
 */
@EnableEurekaClient
public class UAACloudApplication
{
    public static void main(String[] args)
    {
        SpringApplication.run(UAACloudApplication.class, args);
    }
}
```

接下来，在 Provider 模块（或者项目）的 src/main/resources 的 bootstrap 启动属性文件中（bootstrap.properties或bootstrap.yml），增加 Provider 实例相关的配置，具体如下：

```
spring:
  application:
    name: uaa-provider

server:
  port: 7702
  servlet:
      context-path: /uaa-provider
eureka:
  instance:
    instance-id: ${spring.cloud.client.ip-address}:${server.port}
    ip-address: ${spring.cloud.client.ip-address}
    prefer-ip-address: true  #访问路径优先使用 IP地址
    status-page-url-path: /${server.servlet.context-path}${management.endpoints.web.base-path}/
    health-check-url-path: /${server.servlet.context-path}${management.endpoints.web.base-path
client:
    egister-with-eureka: true  #注册到eureka服务器
```

```
    fetch-registry: true        #是否去注册中心获取其他服务
    serviceUrl:
      defaultZone: http://${EUREKA_ZONE_HOST:localhost}:7777/eureka/
```

在详细介绍上面的配置项之前，先启动一下 Provider 的启动类， 控制台的日志大致如下：

```
...com.netflix.discovery.DiscoveryClient - DiscoveryClient_UAA-PROVIDER/192.168.233.128:7702:
....
...com.netflix.discovery.DiscoveryClient - DiscoveryClient_UAA-PROVIDER/192.168. 233.128:7702
```

如果看到上面的日志，表明 Provider 实例已经启动成功。可以进一步通过 Eureka Server 检查服务是否注册成功：打开 Eureka Server 的控制台界面，可以看到 uua-provider 的一个实例已经成功注册，具体如下图所示。

图：uua-provider 实例已经在成功注册到 Eureka Server

前面讲到，SpringCloud 中一个 Provider 实例身兼两者角色：Provider 服务提供者、注册中心客户端。所以，在 Provider 的配置文件中，包含了两类配置：Provider 实例角色的相关配置、Eureka Client 客户端角色的相关配置。有关的Provider 实例角色的相关配置，请参考《SpringCloud Nginx高并发核心编程》一书。

### uaa-provider 实现一个Rest接口

以 uaa-Provider的 获取用户信息接口为例，进行介绍，

这里实现一个获取用户信息的接口 /api/user/detail/v1 ，该接口的具体的代码，在uaa-Provider模块中，如下图所示：

具体的代码如下：

```
package com.crazymaker.springcloud.user.info.controller;

import com.alibaba.fastjson.JSONObject;
import com.crazymaker.springcloud.common.dto.UserDTO;
import com.crazymaker.springcloud.common.result.RestOut;
import com.crazymaker.springcloud.user.info.service.impl.FrontUserEndSessionServiceImpl;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.Resource;


@Api(value = "用户信息、基础学习DEMO", tags = {"用户信息、基础学习DEMO"})
@RestController
@RequestMapping("/api/user" )
public class UserController
{

    @Resource
```

```
    private FrontUserEndSessionServiceImpl userService;
    /**
     * 注入全局的加密器
     */
    @Resource
    PasswordEncoder passwordEncoder;

    @GetMapping("/detail/v1" )
    @ApiOperation(value = "获取用户信息" )
    public RestOut<UserDTO> getUser(@RequestParam(value = "userId", required = true) Long user
    {
        UserDTO dto = userService.getUser(userId);
        if (null == dto)
        {
            return RestOut.error("没有找到用户" );
        }
        return RestOut.success(dto).setRespMsg("操作成功" );
    }


    @GetMapping("/passwordEncoder/v1" )
    @ApiOperation(value = "密码加密" )
    public RestOut<String> passwordEncoder(
            @RequestParam(value = "raw", required = true) String raw)
    {

//        passwordEncoder = PasswordEncoderFactories.createDelegatingPasswordEncoder();
        String encode = passwordEncoder.encode(raw);
        return RestOut.success(encode);
    }
}
```

**uaa-provider**的运行结果

获取用户信息：

### **demo-provider** 完成**RPC**远程调用

demo-provider 使用 Feign+Ribbon 进行 RPC 远程调用时，对每一个Java 远程调用接口，Feign 都会生成了一个 RPC远程调用客户端实现类，只是，该实现类对于开发者来说是透明的，开发者感觉不到这个类的存在。

需要在 Maven 的pom文件中，增加以下Feign+Ribbon 集成模块的依赖：

```
 <!--导入 SpringCloud Ribbon -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>

<!--添加Feign依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

客户端 RPC 实现类位于远程调用 Java 接口和服务提供者 Provider 之间，承担了以下职责：

（1）拼装 REST 请求：根据 Java 接口的参数，拼装目标 REST 接口的 URL；

（2）发送请求和获取结果：通过 Java HTTP 组件（如 HttpClient）调用服务提供者 Provider 的 REST 接口，并且获取 REST 响应；

（3）结果解码：解析 REST 接口的响应结果，封装成目标 POJO 对象（Java 接口的返回类型），并且返回。

> **REST**服务的本地代理接口

该接口的具体的代码，在**uaa-client模块**中，如下图所示：

具体的代码如下：

```
package com.crazymaker.springcloud.user.info.remote.client;

import com.crazymaker.springcloud.common.dto.UserDTO;
import com.crazymaker.springcloud.common.result.RestOut;
import com.crazymaker.springcloud.standard.config.FeignConfiguration;
import com.crazymaker.springcloud.user.info.remote.fallback.UserClientFallback;
import com.crazymaker.springcloud.user.info.remote.fallback.UserClientFallbackFactory;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springfra 的PDF文件，请从mework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

/**
 * Feign 客户端接口
 * @description: 用户信息 远程调用接口
 * @date 2019年7月22日
 */

@FeignClient(value = "uaa-provider",
```

```
        configuration = FeignConfiguration.class,
        fallback = UserClientFallback.class,
//        fallbackFactory = UserClientFallbackFactory.class,
        path = "/uaa-provider/api/user")
public interface UserClient
{
    /**
     * 远程调用 RPC 方法：获取用户详细信息
     * @param userId 用户 Id
     * @return 用户详细信息
     */
    @RequestMapping(value = "/detail/v1", method = RequestMethod.GET)
    RestOut<UserDTO> detail(@RequestParam(value = "userId") Long userId);
}
```

> 通过**REST**服务的本地代理接口，进行**RPC**调用

进行RPC调用的具体的代码，在**demo-provider**模块中，如下图所示：

```
package com.crazymaker.springcloud.demo.controller;

import com.alibaba.fastjson.JSONObject;
import com.alibaba.fastjson.TypeReference;
import com.crazymaker.springcloud.common.dto.UserDTO;
import com.crazymaker.springcloud.common.result.RestOut;
import com.crazymaker.springcloud.common.util.JsonUtil;
import com.crazymaker.springcloud.user.info.remote.client.UserClient;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import javax.annotation.Resource;
```

```
@RestController
@RequestMapping("/api/call/uaa/")
@Api(tags = "演示 uaa-provider 远程调用")
public class UaaRemoteCallController
{

    //注入 @FeignClient注解配置 所配置的 客户端实例
    @Resource
    UserClient userClient;

    @GetMapping("/user/detail/v2")
    @ApiOperation(value = "Feign 远程调用")
    public RestOut<JSONObject> remoteCallV2(
            @RequestParam(value = "userId") Long userId)
    {
        RestOut<UserDTO> result = userClient.detail(userId);
        JSONObject data = new JSONObject();
        data.put("uaa-data", result);
        return RestOut.success(data).setRespMsg("操作成功");
    }
}
```

demo-provider需要依赖 **uaa-client模块**

> 启动**demo-provider**

访问swagger ui：

通过**swagger**执行**RPC**操作

## SpringCloud Eureka 服务注册

Eureka 作为老牌 SpringCloud 注册中心，很多项目，仍然在使用，

另外，底层原理都是想通的，大家可以和 nacos 对比学习

Eureka 的详细介绍，请阅读 《Java高并发核心编程 卷3 加强版》

## SpringCloud Config 统一配置

SpringCloud Config 作为老牌 SpringCloud配置中心，很多项目，仍然在使用，

另外，底层原理都是想通的，大家可以和 nacos 对比学习

SpringCloud Config 的详细介绍，请阅读 《Java高并发核心编程 卷3 加强版》

## Nacos 服务注册+统一配置

### 1、Nacos 优势

> 问题，既然有了Eureka ，为啥还要用Nacos?

而 Nacos 作为微服务核心的服务注册与发现中心，让大家在 Eureka 和 Consule 之外有了新的选择，开箱即用，上手简洁，暂时也没发现有太大的坑。

#### 1.1 与eureka对比

> 1 eureka 2.0闭源码了。
>
> 2 从官网来看nacos 的注册的实例数是大于eureka的,
>
> 3 因为nacos使用的raft协议,nacos集群的一致性要远大于eureka集群.

分布式一致性协议 Raft，自 2013 年论文发表，之后就受到了技术领域的热捧，与其他的分布式一致性算法比，Raft 相对比较简单并且易于实现，这也是 Raft 能异军突起的主要因素。

> **Raft** 的数据一致性策略
>
> Raft 协议强依赖 Leader 节点来确保集群数据一致性。即 client 发送过来的数据均先到达 Leader 节点，Leader 接收到数据后，先将数据标记为 uncommitted 状态，随后 Leader 开始向所有 Follower 复制数据并等待响应，在获得集群中大于 N/2 个 Follower 的已成功接收数据完毕的响应后，Leader 将数据的状态标记为 committed，随后向 client 发送数据已接收确认，在向 client 发送出已数据接收后，再向所有 Follower 节点发送通知表明该数据状态为committed。

### 1.2 与**springcloud config** 对比

三大优势：

- springcloud config大部分场景结合git 使用，动态变更还需要依赖Spring Cloud Bus 消息总线来通过所有的客户端变化.
- springcloud config不提供可视化界面
- nacos config使用长连接更新配置, 一旦配置有变动后，通知Provider的过程非常的迅速, 从速度上秒杀springcloud原来的config几条街,

## 2、Spring Cloud Alibaba 套件

目前 Spring Cloud Alibaba 主要有三个组件：

- Nacos：一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。
- Sentinel：把流量作为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。
- AliCloud OSS: 阿里云对象存储服务（Object Storage Service，简称 OSS），是阿里云提供的海量、安全、低成本、高可靠的云存储服务。您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。

### Spring Cloud Alibaba 套件和Spring Cloud Netflix套件类比

仔细看看各组件的功能描述，Spring Cloud Alibaba 套件和Spring Cloud Netflix套件大致的对应关系：

- Nacos = Eureka/Consule + Config + Admin
- Sentinel = Hystrix + Dashboard + Turbine
- Dubbo = Ribbon + Feign
- RocketMQ = RabbitMQ
- Schedulerx = Quartz
- AliCloud OSS、AliCloud SLS 这三个应该是独有的

链路跟踪（Sleuth、Zipkin）不知道会不会在 Sentinel 里

以上只是猜测，待我从坑里爬出来之后再回来更新。也欢迎大家一起交流探讨~

这里我就先试试 Nacos。

## 3、Nacos 的架构和安装

### 3.1 Nacos 的架构

这是 Nacos 的架构图，可以看到它确实是融合了服务注册发现中心、配置中心、服务管理等功能，类似于 Eureka/Consule + Config + Admin 的合体。

另外通过官方文档发现，Nacos 除了可以和 Spring Cloud 集成，还可以和 Spring、SpringBoot 进行集成。

不过我们只关注于 Spring Cloud，别的就略过了，直接上手实战吧。

### 3.2 Nacos Server 的下载和安装

在使用 Nacos 之前，需要先下载 Nacos 并启动 Nacos Server。

> 安装的参考教程：
>
> https://www.cnblogs.com/crazymakercircle/p/11992539.html

## 4、Nacos Server 的运行

### 4.1两种模式

Nacos Server 有两种运行模式：

- standalone
- cluster

### 4.2 standalone 模式

此模式一般用于 demo 和测试，不用改任何配置，直接敲以下命令执行

```
sh bin/startup.sh -m standalone
```

Windows 的话就是

```
cmd bin/startup.cmd -m standalone
```

然后从 http://cdh1:8848/nacos/index.html 进入控制台就能看到如下界面了

默认账号和密码为：nacos nacos

### 4.3 cluster 模式

测试环境，可以先用 standalone 模式撸起来，享受 coding 的快感，但是，生产环境可以使用 cluster 模式。

**cluster** 模式需要依赖 **MySQL**，然后改两个配置文件：

```
conf/cluster.conf
conf/application.properties
```

大致如下：

1：cluster.conf，填入要运行 Nacos Server 机器的 ip

```
192.168.100.155
192.168.100.156
```

2. 修改NACOS_PATH/conf/application.properties，加入 MySQL 配置

```
db.num=1
db.url.0=jdbc:mysql://localhost:3306/nacos_config?characterEncoding=utf8&connectTimeout=1000&s
db.user=root
db.password=root
```

创建一个名为nacos_config的 database，将NACOS_PATH/conf/nacos-mysql.sql中的表结构导入刚才创建的库中，这几张表的用途就自己研究吧

### 4.4 Nacos Server 的配置数据是存在哪里呢？

> 问题来了：Nacos Server 的配置数据是存在哪里呢?

我们没有对 Nacos Server 做任何配置，那么数据只有两个位置可以存储：

- 内存
- 本地数据库

如果我们现在重启刚刚在运行的 Nacos Server，会发现刚才加的 nacos.properties 配置还在，说明不是内存存储的。

这时候我们打开NACOS_PATH/data，会发现里边有个derby-data目录，我们的配置数据现在就存储在这个库中。

> Derby 是 Java 编写的数据库，属于 Apache 的一个开源项目

如果将数据源改为我们熟悉的 MySQL 呢？当然可以。

> 注意：不支持 MySQL 8.0 版本

这里有两个坑：

> Nacos Server 的数据源是用 Derby 还是 MySQL 完全是由其运行模式决定的：

```
standalone 的话仅会使用 Derby，即使在 application.properties 里边配置 MySQL 也照样无视；
cluster 模式会自动使用 MySQL，这时候如果没有 MySQL 的配置，是会报错的。
```

官方提供的 cluster.conf 示例如下

```
#it is ip
#example
10.10.109.214
11.16.128.34
11.16.128.36
```

以上配置结束后，运行 Nacos Server 就能看到效果了。

## 5、实战1：使用Nacos作为注册中心

> 实战的工程

实战的工程的目录结构如下：

### 5.1 如何使用Nacos Client组件

首先引入 **Spring Cloud Alibaba** 的 **BOM**

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.4.RELEASE</version>
    <relativePath/>
</parent>
<properties>
    <spring-cloud.version>Finchley.SR2</spring-cloud.version>
    <spring-cloud-alibaba.version>0.2.0.RELEASE</spring-cloud-alibaba.version>
</properties>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>${spring-cloud-alibaba.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

这里版本号有坑，文档上说和 Spring Boot 2.0.x 版本兼容，但是实测 2.0.6.RELEASE 报错

```
java.lang.NoClassDefFoundError: org/springframework/core/env/EnvironmentCapable
```

### 5.2 演示的模块结构

服务注册中心和服务发现的服务端都是由 Nacos Server 来提供的，我们只需要提供 Service 向其注册就好了。

这里模拟提供两个 service：provider 和 consumer

```
alibaba
├── service-provider-demo
│   ├── pom.xml
│   └── src
└── sevice-consumer-demo
│   ├── pom.xml
│   └── src
└── pom.xml
```

### 5.3 provider 微服务

**step1：在 provider 和 consumer 的 pom 添加以下依赖：**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
```

```
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

**step2：**启动类

使用 Spring Cloud 的原生注解 @EnableDiscoveryClient 开启服务注册与发现

```
package com.crazymaker.cloud.nacos.demo.starter;

import com.crazymaker.springcloud.standard.context.SpringContextUtil;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.env.Environment;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

import java.util.List;

@EnableSwagger2
@SpringBootApplication
@EnableDiscoveryClient
@Slf4j
public class ServiceProviderApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =   SpringApplication.run(ServicePro

        Environment env = applicationContext.getEnvironment();
        String port = env.getProperty("server.port");
        String path = env.getProperty("server.servlet.context-path");
        System.out.println("\n----------------------------------\n\t" +
                "Application is running! Access URLs:\n\t" +
                "Local: \t\thttp://localhost:" + port + path+ "/index.html\n\t" +
               "swagger-ui: \thttp://localhost:" + port + path + "/swagger-ui.html\n\t" +
```

```
                "------------------------------------------------------");

    }
}
```

**step3**：服务提供者的 **Rest** 服务接口

service-provider-demo 提供 一个非常简单的 Rest 服务接口以供访问

```
package com.crazymaker.cloud.nacos.demo.controller;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/echo")
public class EchoController {
    //回显服务
    @RequestMapping(value = "/{string}", method = RequestMethod.GET)
    public String echo(@PathVariable String string) {
        return "echo: " + string;
    }
}
```

**step4**：配置文件

```
spring:
  application:
    name: service-provider-demo
  cloud:
    nacos:
      discovery:
        server-addr: ${NACOS_SERVER:cdh1:8848}
server:
  port: 18080
```

**step5**：启动之后，通过**swagger UI**访问：

### 5.4 Consumer 微服务演示RPC远程调用

在 NacosConsumerApplication 中集成 RestTemplate 和 Ribbon

```
@LoadBalanced
@Bean
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

消费者的**controller** 类

```
package com.crazymaker.cloud.nacos.demo.consumer.controller;

import com.crazymaker.cloud.nacos.demo.consumer.client.EchoClient;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.Resource;

@RestController
@RequestMapping("/echo")
@Api(tags = "服务- 消费者")
public class EchoConsumerController {


    //注入 @FeignClient 注解配置 所配置的 EchoClient 客户端Feign实例
    @Resource
    EchoClient echoClient;


    //回显服务
    @ApiOperation(value = "消费回显服务接口")
    @RequestMapping(value = "/{string}", method = RequestMethod.GET)
    public String echoRemoteEcho(@PathVariable String string) {
        return "provider echo is:" + echoClient.echo(string);
    }
}
```

消费者配置文件

```
spring:
  application:
    name: sevice-consumer-demo
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
```

```
server:
  port: 18081
```

通过**swagger UI**访问消费者：

访问远程的echo API：

### 5.5涉及到的演示地址：

服务提供者 service-provider-demo：

(http://localhost:18080/provider/swagger-ui.html#/Echo_演示)

服务消费者：

(http://localhost:18081/consumer/swagger-ui.html#/服务- 消费者/echoRemoteEchoUsingGET)

注册中心Nacos：

http://cdh1:8848/nacos/index.html#/serviceManagement?
dataId=&group=&appName=&namespace=

### 5.6 Nacos Console

这时候查看 Nacos Console 也能看到已注册的服务列表及其详情

## 6、实战2：使用Nacos作为配置中心

### 6.1 基本概念

#### 1）Profile

Java项目一般都会有多个Profile配置，用于区分开发环境，测试环境，准生产环境，生成环境等，每个环境对应一个properties文件（或是yml/yaml文件），然后通过设置spring.profiles.active 的值来决定使用哪个配置文件。

例子：

```
spring:
  application:
    name: sharding-jdbc-provider
  jpa:
    hibernate:
      ddl-auto: none
      dialect: org.hibernate.dialect.MySQL5InnoDBDialect
      show-sql: true
  profiles:
     active: sharding-db-table     # 分库分表配置文件
    #active: atomiclong-id     # 自定义主键的配置文件
    #active: replica-query     # 读写分离配置文件
```

Nacos Config的作用就把这些文件的内容都移到一个统一的配置中心，即方便维护又支持实时修改后动态刷新应用。

### 2）Data ID

当使用Nacos Config后，Profile的配置就存储到Data ID下，即一个Profile对应一个Data ID

Data ID的拼接格式：$prefix-$ {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 取 spring.profiles.active 的值，即为当前环境对应的 profile
- file-extension 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

### 3）Group

Group 默认为 DEFAULT_GROUP，可以通过 spring.cloud.nacos.config.group 来配置，当配置项太多或者有重名时，可以通过分组来方便管理

最后就和原来使用springcloud一样通过@RefreshScope 和@Value注解即可

## 6.2 通过Nacos的console 去增加配置

这回首先要在nacos中配置相关的配置，打开Nacos配置界面，依次创建2个Data ID

- nacos-config-demo-dev.yaml　　开发环境的配置
- nacos-config-demo-test.yaml　　测试环境的配置

### 1）nacos-config-demo-dev.yaml

内容如下图：

**2）nacos-config-demo-sit.yaml**

> 内容如下图：

## 6.3 使用Nacos Config Client组件

问题2：微服务Provider实例上，如何使用Nacos Config Client组件的有哪些步骤?

**1**）加载**nacos config** 的客户端依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
    <version>${nacos.version}</version>
</dependency>
```

启动类

```
package com.crazymaker.cloud.nacos.demo.consumer.starter;

import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration;
import org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;
import org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguratio
import org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration;
import org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.env.Environment;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@EnableSwagger2
@EnableDiscoveryClient
@Slf4j
@SpringBootApplication(
        scanBasePackages =
                {
                        "com.crazymaker.cloud.nacos.demo",
                        "com.crazymaker.springcloud.standard"
                },
        exclude = {SecurityAutoConfiguration.class,
                //排除db的自动配置
```

```
                DataSourceAutoConfiguration.class,
                DataSourceTransactionManagerAutoConfiguration.class,
                HibernateJpaAutoConfiguration.class,
                //排除redis的自动配置
                RedisAutoConfiguration.class,
                RedisRepositoriesAutoConfiguration.class})
//启动Feign
@EnableFeignClients(basePackages =
        {"com.crazymaker.cloud.nacos.demo.consumer.client"})
public class ConfigDomeProviderApplication
{
    public static void main(String[] args)
    {
        ConfigurableApplicationContext applicationContext = null;
        try
        {
            applicationContext = SpringApplication.run(ConfigDomeProviderApplication.class, ar
            System.out.println("Server startup done.");
        } catch (Exception e)
        {
            log.error("服务启动报错", e);
            return;
        }

        Environment env = applicationContext.getEnvironment();
        String port = env.getProperty("server.port");
        String path = env.getProperty("server.servlet.context-path");
        System.out.println("\n----------------------------------------------------------\n\t" +
                "Application is running! Access URLs:\n\t" +
                "Local: \t\thttp://localhost:" + port + path + "/index.html\n\t" +
                "swagger-ui: \thttp://localhost:" + port + path + "/swagger-ui.html\n\t" +
                "----------------------------------------------------------");

    }
}
```

控制类：

```
package com.crazymaker.cloud.nacos.demo.config.controller;
```

```
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.Resource;

@RestController
@RequestMapping("/config")
@Api(tags = "Nacos 配置中心演示")
public class ConfigGetController {


    @Value("${foo.bar:empty}")
    private String bar;


    @Value("${spring.datasource.username:empty}")
    private String dbusername;

    //获取配置的内容
    @ApiOperation(value = "获取配置的内容")
    @RequestMapping(value = "/bar", method = RequestMethod.GET)
    public String getBar() {
        return "bar is :"+bar;
    }
    //获取配置的内容
    @ApiOperation(value = "获取配置的db username")
    @RequestMapping(value = "/dbusername", method = RequestMethod.GET)
    public String getDbusername() {
        return "db username is :"+bar;
    }
}
```

### 2）bootstrap配置文件

然后是在配置文件(bootstrap.yml)中加入以下的内容：

```
spring:
  application:
    name: nacos-config-demo-provider
  profiles:
    active:  dev
  cloud:
    nacos:
      discovery:
        server-addr: ${NACOS_SERVER:cdh1:8848}
      config:
        server-addr: ${NACOS_SERVER:cdh1:8848}
        prefix: nacos-config-demo
        group: DEFAULT_GROUP
        file-extension: yaml
server:
  port: 18083
  servlet:
    context-path: /config
```

### 6.4 测试结果

启动程序，通过swagger ui访问：

```
http://localhost:18083/config/swagger-ui.html#
```

执行结果如下：

### 6.4 可以端如何与服务端的配置文件相互对应

- config.prefix 来对应主配置文件
- 使用spring.cloud.nacos.config.ext-config 选项来对应更多的文件eg：

```
spring:
  application:
    name: nacos-config-demo-provider
  profiles:
    active: dev
  cloud:
    nacos:
      discovery:
        server-addr: ${NACOS_SERVER:cdh1:8848}
      config:
        server-addr: ${NACOS_SERVER:cdh1:8848}
        prefix: nacos-config-demo
        group: DEFAULT_GROUP
        file-extension: yaml
        ext-config:
          - data-id: crazymaker-db-dev.yml
            group: DEFAULT_GROUP
```

```
        refresh: true
      - data-id: crazymaker-redis-dev.yml
        group: DEFAULT_GROUP
        refresh: true
      - data-id: crazymaker-common-dev.yml
        group: DEFAULT_GROUP
        refresh: true
      - data-id: some.properties
        group: DEFAULT_GROUP
        refresh: true
```

启动程序，发现可以获取到其他data-id的配置 ,大家可以自行配置。

### 7、配置的隔离

在实际的应用中，存在着以下几种环境隔离的要求：

1、开发环境、测试环境、准生产环境和生产环境需要隔离

2、不同项目需要隔离

3、同一项目，不同的模块需要隔离

可以通过三种方式来进行配置隔离：Nacos的服务器、namespace命名空间、group分组，在bootstrap.yml文件中可以通过配置Nacos的server-addr、namespace和group来区分不同的配置信息。

- Nacos的服务器 spring.cloud.nacos.config.server-addr
- Nacos的命名空间 spring.cloud.nacos.config.namespace，注意，这里使用命名空间的ID不是名称
- Nacos的分组 spring.cloud.nacos.config.group

### 8、nacos集群搭建

如果我们要搭建集群的话，那么肯定是不能用内嵌的数据库，不然数据无法共享。所以，集群搭建的时候我们需要将Nacos对接Mysql进行数据存储。

集群模式跟我们平时进行扩容是一样的，可以通过Nginx转发到多个节点，最前面挂一个域名即可，如下图：

### IP规划

通常如果我们只是为了体验的话，直接在本地起动3个实例就可以了，没必要真的去搞三台服务器，下面我们就以在本地的方式来搭建集群。将Nacos的解压包复制分成3份，分别是:

nacos nacos1 nacos2

进入nacos的conf目录，编辑application.properties文件，增加数据库配置

```
# 指定数据源为Mysql
spring.datasource.platform=mysql

# 数据库实例数量
db.num=1
db.url.0=jdbc:mysql://localhost:3306/nacos?characterEncoding=utf8&connectTimeout=1000&socketTi
db.user=root
db.password=123456
```

复制代码同样的步骤进入nacos1和nacos2操作一遍，唯一需要修改的就是application.properties文件中的server.port，默认nacos的server.port=8848，

我们在本地启动三个实例，那么端口肯定会冲突，所以其他2个实例的端口我们需要进行修改，比如nacos1修改成8847，nacos2修改成8846。

数据库配置信息好了后，我们需要将对应的数据库和表进行初始化，数据库脚本在conf目录下的nacos-mysql.sql中，执行即可。

最后一步需要配置一份集群节点信息，配置文件在conf目录下的cluster.conf.example文件，我们进行重命名成cluster.conf。然后编辑cluster.conf文件，增加3个节点的信息，格式为IP:PORT，三个目录都一致即可。

```
127.0.0.1:8848
127.0.0.1:8847
127.0.0.1:8846
```

启动的话直接到bin目录下，执行./startup.sh就可以了，默认就是集群模式，不需要加任何参数。

### 集群的使用

上面的集群，虽然可用， 但仍不是真正的集群， 我们一般不会这么用。nacos集群的使用一般有4种方式：

- http://ip1:port/openAPI 直连ip模式，不同的节点，则需要修改ip才可以使用。
- http://VIP:port/openAPI VIP模式高可用，客户端vip即可，VIP下面挂server真实ip，部署比较麻烦，需要部署vip（keepalive）。
- http://nacos.com:port/openAPI 域名模式，可读性好，而且换ip方便，在host文件配置本地域名即可。
- http://反向代理:port/openAPI 反向代理模式

> 这里介绍一下反向代理模式。

关于Nginx的安装和配置，本文就不进行讲解了，不会的可以自己去尝试下，反向代理模式 核心配置如下：

```
upstream nacos_server {
  server 127.0.0.1:8848;
  server 127.0.0.1:8847;
  server 127.0.0.1:8846;
}

server {
listen 8648;
   server_name localhost;
```

```
    #charset koi8-r;

    #access_log logs/host.access.log main;
    location / {
       proxy_pass http://nacos_server;
       index index.html index.htm;
     }
}
```

整体来说，nacos的集群搭建方式还是挺简单的，没什么特别要注意的，最好是能通过域名的方式来进行访问，另外数据库这块如果上生产环境，也需要考虑高可用问题，至少也得有个主从。

8648 的nginx 提供的 nacos 服务接口，可以自定义。我们访问

http://localhost:8648/nacos/#/clusterManagement?dataId=&group=&appName=&namespace=&serverId=

，就可以看到：

我们可以简单测试一下，杀掉 一个的 nacos ，看服务是否正常。后面，我们对微服务提供nacos服务的时候，只要配置这个nginx 端口就好了！！

dataId=&group=&appName=&namespace=&serverId=

，就可以看到：

我们可以简单测试一下，杀掉 一个的 nacos ，看服务是否正常。后面，我们对微服务提供nacos服务的时候，只要配置这个nginx 端口就好了！！

## Nacos 高可用架构与实操

当我们在聊高可用时，我们在聊什么?

- 系统可用性达到 99.99%
- 在分布式系统中，部分节点宕机，依旧不影响系统整体运行
- 服务端集群化部署多个节点

Nacos 高可用，则是 Nacos 为了提升系统稳定性而采取的一系列手段。

Nacos 的高可用不仅仅存在于服务端，同时也存在于客户端，以及一些与可用性相关的功能特性中，这些点组装起来，共同构成了 Nacos 的高可用。

### 客户端高可用

先统一一下语义，在微服务架构中一般会有三个角色：

- Consumer
- Provider
- Registry

以上的registry 角色是 nacos-server，而 Consumer 角色和 Provider 角色都是 nacos-client。

#### 客户端高可用的方式一：配置多个**nacos-server**

在生产环境，我们往往需要搭建 Nacos 集群，代码中，是这样配置的：

```
server:
  port: 8081
spring:
  cloud:
    nacos:
      server-addr: 127.0.0.1:8848,127.0.0.1:8848,127.0.0.1:8848
```

当其中一台Nacos server机器宕机时，为了不影响整体运行，客户端会存在重试机制。

```
package com.alibaba.nacos.client.naming.net;


/**
 * @author nkorange
 */
public class NamingProxy {

//api注册

public String reqAPI(String api, Map<String, String> params, String body, List<String> servers

params.put(CommonParams.NAMESPACE_ID, getNamespaceId());

if (CollectionUtils.isEmpty(servers) && StringUtils.isEmpty(nacosDomain)) {
    throw new NacosException(NacosException.INVALID_PARAM, "no server available");
}

NacosException exception = new NacosException();

if (servers != null && !servers.isEmpty()) {

    Random random = new Random(System.currentTimeMillis());
    int index = random.nextInt(servers.size());

    //拿到地址列表，在请求成功之前逐个尝试，直到成功为止

    for (int i = 0; i < servers.size(); i++) {
        String server = servers.get(index);
        try {
            return callServer(api, params, body, server, method);
```

```
        } catch (NacosException e) {
            exception = e;
            if (NAMING_LOGGER.isDebugEnabled()) {
                NAMING_LOGGER.debug("request {} failed.", server, e);
            }
        }
        index = (index + 1) % servers.size();
    }
}
...
```

该可用性保证存在于 nacos-client 端。

**Nacos Java Client**通用参数

| 参数名 | 含义 | 可选值 | 默认值 | 支持版本 |
| --- | --- | --- | --- | --- |
| endpoint | 连接Nacos Server指定的连接点，可以参考文档 | 域名 | 空 | >= 0.1.0 |
| endpointPort | 连接Nacos Server指定的连接点端口，可以参考文档 | 合法端口号 | 空 | >= 0.1.0 |
| namespace | 命名空间的ID | 命名空间的ID | config模块为空，naming模块为public | >= 0.8.0 |
| serverAddr | Nacos Server的地址列表，这个值的优先级比endpoint高 | ip:port,ip:port,... | 空 | >= 0.1.0 |
| JM.LOG.PATH(-D) | 客户端日志的目录 | 目录路径 | 用户根目录 | >= 0.1.0 |

客户端高可用的方式二：本地缓存文件 **Failover** 机制

注册中心发生故障最坏的一个情况是整个 Server 端宕机，如果三个Server 端都宕机了，怎么办呢?

> 这时候 Nacos 依旧有高可用机制做兜底。

本地缓存文件 **Failover** 机制

一道经典的 高可用的面试题：

> 当 springcloud 应用运行时，Nacos 注册中心宕机，会不会影响 RPC 调用。

这个题目大多数人，应该都不能回答出来.

Nacos 存在本地文件缓存机制，nacos-client 在接收到 nacos-server 的服务推送之后，会在内存中保存一份，随后会落盘存储一份快照snapshot 。有了这份快照，本地的RPC调用，还是能正常的进行。

> 关键是，这个本地文件缓存机制，默认是关闭的。

Nacos 注册中心宕机，Dubbo /springcloud 应用发生重启，会不会影响 RPC 调用。如果了解了 Nacos 的 Failover 机制，应当得到和上一题同样的回答：不会。

客户端**Naming**通用参数

| 参数名 | 含义 | 可选值 | 默认值 | 支持版本 |
| --- | --- | --- | --- | --- |
| namingLoadCacheAtStart | 启动时是否优先读取本地缓存 | true/false | false | >= 1.0.0 |
| namingClientBeatThreadCount | 客户端心跳的线程池大小 | 正整数 | 机器的CPU数的一半 | >= 1.0.0 |
| namingPollingThreadCount | 客户端定时轮询数据更新的线程池大小 | 正整数 | 机器的CPU数的一半 | >= 1.0.0 |
| com.alibaba.nacos.naming.cache.dir(-D) | 客户端缓存目录 | 目录路径 | {user.home}/nacos/naming | >= 1.0.0 |
| com.alibaba.nacos.naming.log.level(-D) | Naming客户端的日志级别 | info,error,warn等 | info | >= 1.0.0 |
| com.alibaba.nacos.client.naming.tls.enable | 是否打开HTTPS | true/false | false | |

| 参数名 | 含义 | 可选值 | 默认值 | 支持版本 |
| --- | --- | --- | --- | --- |
| (-D) | | | | |

snapshot 默认的存储路径为：{USER_HOME}/nacos/naming/ 中：

这份文件有两种价值，一是用来排查服务端是否正常推送了服务；二是当客户端加载服务时，如果无法从服务端拉取到数据，会默认从本地文件中加载。

在生产环境，推荐开启该参数，以避免注册中心宕机后，导致服务不可用，在服务注册发现场景，可用性和一致性 trade off 时，我们大多数时候会优先考虑可用性。

另外：{USER_HOME}/nacos/naming/{namespace} 下除了缓存文件之外还有一个 failover 文件夹，里面存放着和 snapshot 一致的文件夹。

这是 Nacos 的另一个 failover 机制，snapshot 是按照某个历史时刻的服务快照恢复恢复，而 failover 中的服务可以人为修改，以应对一些极端场景。

该可用性保证存在于 nacos-client 端。

## Nacos两种健康检查模式

### agent上报模式

**客户端（注册在nacos上的其它微服务实例）健康检查。**

客户端通过心跳上报方式告知服务端(nacos注册中心)健康状态；

默认心跳间隔5秒；

nacos会在超过15秒未收到心跳后将实例设置为不健康状态；

超过30秒将实例删除；

### 服务端主动检测

**服务端健康检查。**

nacos主动探知客户端健康状态，默认间隔为20秒；

健康检查失败后实例会被标记为不健康，不会被立即删除。

> 临时实例

**临时实例通过agent上报模式实现健康检查。**

Nacos 在 1.0.0版本 `instance` 级别增加了一个 `ephemeral` 字段，该字段表示注册的实例是否是临时实例还是持久化实例。

微服务注册为临时实例：

```
# 默认true
spring:
 cloud:
  nacos:
   discovery:
    ephemeral: true
```

注意：默认为临时实例，表示为临时实例。

### 注册实例支持**ephemeral**字段

如果是临时实例，则 `instance` 不会在 Nacos 服务端持久化存储，需要通过上报心跳的方式进行包活，

如果 `instance` 一段时间内没有上报心跳，则会被 Nacos 服务端摘除。

在被摘除后如果又开始上报心跳，则会重新将这个实例注册。

持久化实例则会持久化被 Nacos 服务端，此时即使注册实例的客户端进程不在，这个实例也不会从服务端删除，只会将健康状态设为不健康。

同一个服务下可以同时有临时实例和持久化实例，这意味着当这服务的所有实例进程不在时，会有部分实例从服务上摘除，剩下的实例则会保留在服务下。

使用实例的 `ephemeral` 来判断， `ephemeral` 为 `true` 对应的是服务健康检查模式中的 client 模式,为 `false` 对应的是 server 模式。

Nacos 1.0.0 之前服务的健康检查模式有三种：client、server 和none, 分别代表客户端上报、服务端探测和取消健康检查。在控制台操作的位置如下所示：

在 Nacos 1.0.0 中将把这个配置去掉，改为使用实例的 `ephemeral` 来判断， `ephemeral` 为 `true` 对应的是服务健康检查模式中的 client 模式,为 `false` 对应的是 server 模式。

### 临时实例和持久化实例区别

临时和持久化的区别主要在健康检查失败后的表现，持久化实例健康检查失败后会被标记成不健康，而临时实例会直接从列表中被删除。

这个特性比较适合那些需要应对流量突增，而弹性扩容的服务，当流量降下来后这些实例自己销毁自己就可以了，不用再去nacos里手动调用注销实例。持久化以后，可以实时看到健康状态，

便于做后续的告警、扩容等一系列处理。

### Nacos Server运行模式

Server的运行模式，是指 Nacos Server 可以运行在多种模式下，当前支持三种模式：

- AP
- CP
- MIXED

这里的运行模式，使用的是CAP理论里的C、A和P概念。

CAP原则又称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。

基于CAP理论，在分布式系统中，数据的一致性、服务的可用性和网络分区容忍性只能三者选二。一般来说分布式系统需要支持网络分区容忍性，那么就只能在C和A里选择一个作为系统支持的属性。C 的准确定义应该是所有节点在同一时间看到的数据是一致的，而A的定义是所有的请求都会收到响应。

Nacos 支持 AP 和 CP 模式的切换，这意味着 Nacos 同时支持两者一致性协议。这样，Nacos能够以一个注册中心管理这些生态的服务。不过在Nacos中，AP模式和CP模式的具体含义，还需要再说明下。

AP模式为了服务的可能性而减弱了一致性，因此AP模式下只支持注册临时实例。AP 模式是在网络分区下也能够注册实例。在AP模式下也不能编辑服务的元数据等非实例级别的数据，但是允许创建一个默认配置的服务。同时注册实例前不需要进行创建服务的操作，因为这种模式下，服务其实降级成一个简单的字符创标识，不在存储任何属性，会在注册实例的时候自动创建。

CP模式下则支持注册持久化实例，此时则是以 Raft 协议为集群运行模式，因此网络分区下不能够注册实例，在网络正常情况下，可以编辑服务器别的配置。改模式下注册实例之前必须先注册服务，如果服务不存在，则会返回错误。

MIXED 模式可能是一种比较让人迷惑的模式，这种模式的设立主要是为了能够同时支持临时实例和持久化实例的注册。这种模式下，注册实例之前必须创建服务，在服务已经存在的前提下，临时实例可以在网络分区的情况下进行注册。

### Nacos CP/AP模式设定

使用如下请求进行Server运行模式的设定：

```
curl -X PUT
'$NACOS_SERVER:8848/nacos/v1/ns/operator/switches?entry=serverMode&value=CP'
```

### Nacos CP/AP模式切换

Nacos 集群默认支持的是CAP原则中的AP原则.

但是Nacos 集群可切换为CP原则，切换命令如下：

```
curl -X PUT '$NACOS_SERVER:8848/nacos/v1/ns/operator/switches?entry=serverMode&value=CP'
```

同时微服务的bootstrap.properties 需配置如下选项指明注册为临时/永久实例 AP模式不支持数据一致性，所以只支持服务注册的临时实例，CP模式支持服务注册的永久实例，满足配置文件的一致性

```
#false为永久实例，true表示临时实例开启，注册为临时实例
spring.cloud.nacos.discovery.ephemeral=true
```

### AP/CP的配套一致性协议

介绍一致性模型之前，需要回顾 Nacos 中的两个概念：临时服务和持久化服务。

- 临时服务（Ephemeral）：临时服务健康检查失败后会从列表中删除，常用于服务注册发现场景。
- 持久化服务（Persistent）：持久化服务健康检查失败后会被标记成不健康，常用于 DNS 场景。

两种模式使用的是不同的一致性协议：

- 临时服务使用的是 Nacos 为服务注册发现场景定制化的私有协议 distro，其一致性模型是 AP；
- 而持久化服务使用的是 raft 协议，其一致性模型是 CP。

### AP模式下的distro 协议

distro 协议的工作流程如下：

- Nacos 启动时首先从其他远程节点同步全部数据。
- Nacos 每个节点是平等的都可以处理写入请求，同时把新数据同步到其他节点。
- 每个节点只负责部分数据，定时发送自己负责数据的校验值到其他节点来保持数据一致性。

如图所示，每个节点负责一部分服务的写入。

但每个节点都可以接收到写入请求，这时就存在两种情况：

- 当该节点接收到属于该节点负责的服务时，直接写入。
- 当该节点接收到不属于该节点负责的服务时，将在集群内部路由，转发给对应的节点，从而完成写入。

读取操作则不需要路由，因为集群中的各个节点会同步服务状态，每个节点都会有一份最新的服务数据。

而当节点发生宕机后，原本该节点负责的一部分服务的写入任务会转移到其他节点，从而保证 Nacos 集群整体的可用性。

一个比较复杂的情况是，节点没有宕机，但是出现了网络分区，即下图所示：

这个情况会损害可用性，客户端会表现为有时候服务存在有时候服务不存在。

综上，Nacos 的 distro 一致性协议可以保证在大多数情况下，集群中的机器宕机后依旧不损害整体的可用性。

Nacos 有两个一致性协议：distro 和 raft，distro 协议不会有脑裂问题。

> **CP模式下的raft协议**

> 此文还是聚焦于介绍nacos的高可用， raft协议，请参考尼恩的架构师视频。

### 集群内部的特殊的心跳同步服务

心跳机制一般广泛存在于分布式通信领域，用于确认存活状态。

一般心跳请求和普通请求的设计是有差异的，心跳请求一般被设计的足够精简，这样在定时探测时可以尽可能避免性能下降。

而在 Nacos 中，出于可用性的考虑，一个心跳报文包含了全部的服务信息，这样相比仅仅发送探测信息降低了吞吐量，而提升了可用性，怎么理解呢?

考虑以下的两种场景:

- nacos-server 节点全部宕机，服务数据全部丢失。nacos-server 即使恢复运作，也无法恢复出服务，而心跳包含全部内容可以在心跳期间就恢复出服务，保证可用性。
- nacos-server 出现网络分区。由于心跳可以创建服务，从而在极端网络故障下，依旧保证基础的可用性。

调用 OpenApi 依次删除各个服务:

```
curl -X "DELETE mse-xxx-p.nacos-ans.mse.aliyuncs.com:8848/nacos/v1/ns/service?serviceName=prov
```

过 5s 后刷新，服务又再次被注册了上来，符合我们对心跳注册服务的预期。

### 集群部署模式高可用

最后给大家分享的 Nacos 高可用特性来自于其部署架构。

### 节点数量

我们知道在生产集群中肯定不能以单机模式运行 Nacos。

那么第一个问题便是：我应该部署几台机器?

Nacos 有两个一致性协议：distro 和 raft，distro 协议不会有脑裂问题，所以理论来说，节点数大于等于 2 即可；raft 协议的投票选举机制则建议是 2n+1 个节点。

综合来看，选择 3 个节点是起码的，其次处于吞吐量和更吞吐量的考量，可以选择 5 个，7 个，甚至 9 个节点的集群。

### 多可用区部署

组成集群的 Nacos 节点，应该尽可能考虑两个因素：

- 各个节点之间的网络时延不能很高，否则会影响数据同步。
- 各个节点所处机房、可用区应当尽可能分散，以避免单点故障。

> 以阿里云的 ECS 为例，选择同一个 Region 的不同可用区就是一个很好的实践。

### 部署模式

生产环境，建议使用k8s部署或者阿里云的 ECS 部署。

考虑的中等公司，都会有运维团队，开发人员不需要参与。

所以，这里介绍的开发人员必须掌握的，docker模式的部署。

### 高可用**nacos**的部署架构

### 高可用**nacos**的部署实操

> 实操这块，使用视频介绍更为清晰，请参考尼恩的架构师视频。

### 总结

本文从多个角度出发，总结了一下 Nacos 是如何保障高可用的。

高可用特性绝不是靠服务端多部署几个节点就可以获得的，而是要结合客户端使用方式、服务端部署模式、使用场景综合来考虑的一件事。

特别是在服务注册发现场景，Nacos 为可用性做了非常多的努力，而这些保障，ZooKeeper 是不一定有的。在做注册中心选型时，可用性保障上，Nacos 绝对是优秀的。

---

## SpringCloud Feign 实现RPC 远程调用

SpringCloud Feign 作为老牌 RPC 远程调用 组件，很多项目，仍然在使用，

并且任然是面试的重点

另外，底层原理都是想通的，大家可以Feign 和 dubbo 对比学习

SpringCloud Feign 的详细介绍，请阅读 《Java 高并发核心编程 卷3 加强版》

---

## SpringCloud + Dubbo 实现RPC 远程调用

### 大背景：全链路异步化的大趋势来了

随着业务的发展，微服务应用的流量越来越大，使用到的资源也越来越多。

在微服务架构下，大量的应用都是 SpringCloud 分布式架构，这种架构总体上是**全链路同步模式。**

**全链路同步模式**不仅造成了资源的极大浪费，并且在流量发生激增波动的时候，受制于系统资源而无法快速的扩容。

全球后疫情时代，降本增效是大背景。如何降本增效?

可以通过技术升级，**全链路同步模式** ，升级为 **全链路异步模式。**

先回顾一下全链路同步模式架构图

**全链路同步模式**，如何升级为 **全链路异步模式**， 就是一个一个 环节的异步化。

40岁老架构师尼恩，持续深化自己的3高架构知识宇宙，当然首先要去完成一次牛逼的**全链路异步模式 微服务实操，下面是尼恩的实操过程、效果、压测数据(性能足足提升10倍多)。**

**全链路异步模式**改造 具体的内容，请参考尼恩的深度文章：全链路异步，让你的 SpringCloud 性能优化10倍+

并且，上面的文章，作为尼恩 全链路异步的架构知识，收录在《尼恩 Java 面试宝典》的架构专题中

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，点这里获取

**.....**

**尼恩说明：本文400页，10W多字，还剩下5W多字，放不下.....**

**由于公众号最多可以发布5W字**

**还有5W多字放不下....**

**更多内容，**

**请参见《SpringCloud Alibaba 微服务 学习圣经 》PDF**

**请公众号发送“领电子书”**

技术自由的实现路径：

> 实现你的 架构自由：

《吃透8图1模板，人人可以做架构》

《10Wqps评论中台，如何架构？B站是这么做的！！！》

《阿里二面：千万级、亿级数据，如何性能优化？教科书级 答案来了》

《峰值21WQps、亿级DAU，小游戏《羊了个羊》是怎么架构的？》

《100亿级订单怎么调度，来一个大厂的极品方案》

《2个大厂 100亿级 超大流量 红包 架构方案》

*..... 更多架构文章，正在添加中*

> 实现你的 响应式 自由：

《响应式圣经：10W字，实现Spring响应式编程自由》

这是老版本 《Flux、Mono、Reactor 实战（史上最全）》

> 实现你的 **spring cloud** 自由：

《Nacos (史上最全)》

《nacos高可用（图解+秒懂+史上最全）》

《sentinel （史上最全）》

《Springcloud gateway 底层原理、核心实战 (史上最全)》

《SpringCloud+Dubbo3 = 王炸！》

《分库分表 Sharding-JDBC 底层原理、核心实战（史上最全）》

《一文搞定：SpringBoot、SLF4j、Log4j、Logback、Netty之间混乱关系（史上最全）》

> 实现你的 **linux** 自由：

《Linux命令大全：2W多字，一次实现Linux自由》

> 实现你的 网络 自由：

《TCP协议详解 (史上最全)》

《网络三张表：ARP表, MAC表, 路由表，实现你的网络自由！！》

> 实现你的 分布式锁 自由：

《Redis分布式锁（图解 - 秒懂 - 史上最全）》

《Zookeeper 分布式锁 - 图解 - 秒懂》

> 实现你的 王者组件 自由：

《队列之王：Disruptor 原理、架构、源码 一文穿透》

《缓存之王：Caffeine 源码、架构、原理（史上最全，10W字 超级长文）》

《缓存之王：Caffeine 的使用（史上最全）》

《Java Agent 探针、字节码增强 ByteBuddy（史上最全）》

> 实现你的 面试题 自由：

4000页《尼恩Java面试宝典 》 40个专题

(以上文章PDF版本点这里获取)

---

参考文献

1、疯狂创客圈 JAVA 高并发 总目录 https://www.cnblogs.com/crazymakercircle/p/9904544.html

ThreadLocal（史上最全） https://www.cnblogs.com/crazymakercircle/p/14491965.html

2、3000页《尼恩 Java 面试宝典》的 35个面试专题

3、价值10W的架构师知识图谱
https://www.processon.com/view/link/60fb9421637689719d246739

4、架构师哲学 https://www.processon.com/view/link/616f801963768961e9d9aec8

5、尼恩 3高架构知识宇宙 https://www.processon.com/view/link/635097d2e0b34d40be778ab4

《Java 高并发核心编程 卷3 加强版》

《响应式圣经：10W字，实现Spring响应式编程自由》

全链路异步，让你的 SpringCloud 性能优化10倍+

---

## 硬核电子书

---

👍《尼恩Java面试宝典》（极致经典，不断升级）**全网下载超过300万次**

👍尼恩Java高并发三部曲：**全网下载超过200万次**

- 👍《Java高并发核心编程-卷1（加强版）》，不断升级
- 👍《Java高并发核心编程-卷2（加强版）》，不断升级
- 👍《Java高并发核心编程-卷3（加强版）》，不断升级

👍《顶级3高架构行业案例 + 尼恩架构笔记 》N 篇+，不断添加

👍100份简历模板

阅读原文

喜欢此内容的人还喜欢

SSRF中对gopher协议的利用

奇点bit

推荐 7 个 Vue.js 插件，也许你的项目用的上（五）

前端达人

Geek新鲜事4 - Linux kernel社区发起了抵制Rust的号召

内核及虚拟化漫谈