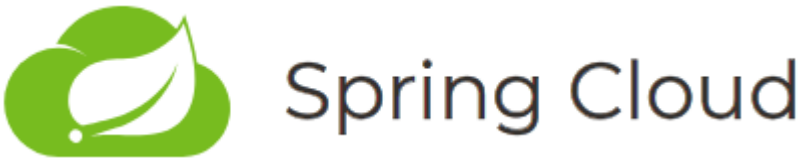


一、微服务Microservices简述

近几年来，“微服务体系结构”这个术语出现了，它描述了将软件应用程序设计为可独立部署的服务套件的特定方式。尽管这种架构风格没有确切的定义，但围绕业务能力，自动化部署，端点智能以及语言和数据的分散控制等方面存在着某些共同特征。

请参考原作者 Martin Fowler 的个人博客：https://martinfowler.com/articles/microservices.html

1、什么是微服务Microservice？（面试）

微服务的核心就是将传统的一站式应用，根据业务拆分成一个一个的服务，彻底的去耦合，每一个微服务提供单个业务功能的服务，一个服务做一件事，从技术角度看就是一种小而独立的处理过程，类似进程概念，能够自行单独启动或销毁，拥有自己独立的数据库。

- 业务拆分
- 单独进程
- 独立数据库

2、微服务的优缺点？（面试）

优点：

- 每个服务足够内聚，足够小，代码容易理解，这样能聚焦一个指定的业务功能或业务需求；
- 开发简单，开发效率提高，一个服务可能就是专一的只干一件事情；
- 微服务能够被小团队单独开发，这个小团队是2-5人的开发人员组成；
- 微服务是松耦合的，是有功能意义的服务，无论在开发阶段还是部署阶段都是独立的；
- 微服务可以使用不同的语言开发；
- 易于和第三方集成，微服务允许容易且灵活的方式集成自动部署，通过持续集成工具，如Jenkins，Hudson；
- 微服务易于被一个开发人员理解，修改和维护，这样小团队能够更关注自己的工作成果，无需通过合作才能体现价值；
- 微服务只是业务逻辑的代码，不会和HTML、CSS或其它界面组件混合；
- 每个微服务有自己独立的存储，可以有独立的数据库。也可以有统一的数据库。

缺点：

- 开发人员要处理分布式系统的复杂性；
- 多服务运维难度，随着服务的增加，运维的压力也在增大；
- 系统部署依赖；
- 服务间通信成本；
- 数据一致性；

- 系统集成测试；
- 性能监控；
- ...

3、微服务和微服务架构？（面试）

- 微服务强调的是一个一个的个体，每个个体做自己的事情。
- 微服务架构强调的是整体，把一个个的微服务组装起来，对外提供服务。

4、微服务技术栈有哪些？（面试）

- 技术栈：多种技术的集合体
- 服务开发 SpringBoot、Spring、SpringMVC
- 服务配置与管理 Netflix公司的Archaius，Alibaba的Diamond等
- 服务注册与发现 Eureka、ZeeKeeper
- 服务调用 Rest、RPC、gRPC
- 服务熔断器 Hystrix、Envoy
- 服务负载均衡 Ribbon、Nginx
- 服务接口调用 Feign
- 消息队列 Kafka、RabbitMQ、ActiveMQ
- 服务配置中心管理 SpringCloudConfig、Chef
- 服务监控 Zabbix、Nagios、Metrics、Spectator
- 服务路由(API网关) Zuul
- 全链路跟踪 Zipkin、Brave、Dapper
- 服务部署 Docker、OpenStack、Kubernetes
- 数据流操作开发包 SpringCloud Stream
- 事件消息总线 SpringCloud Bus
- ...

5、为什么选择SpringCloud作为微服务架构？（面试）

选型依据？

- 整体解决方案和框架成熟度
- 社区热度
- 可维护性
- 学习曲线

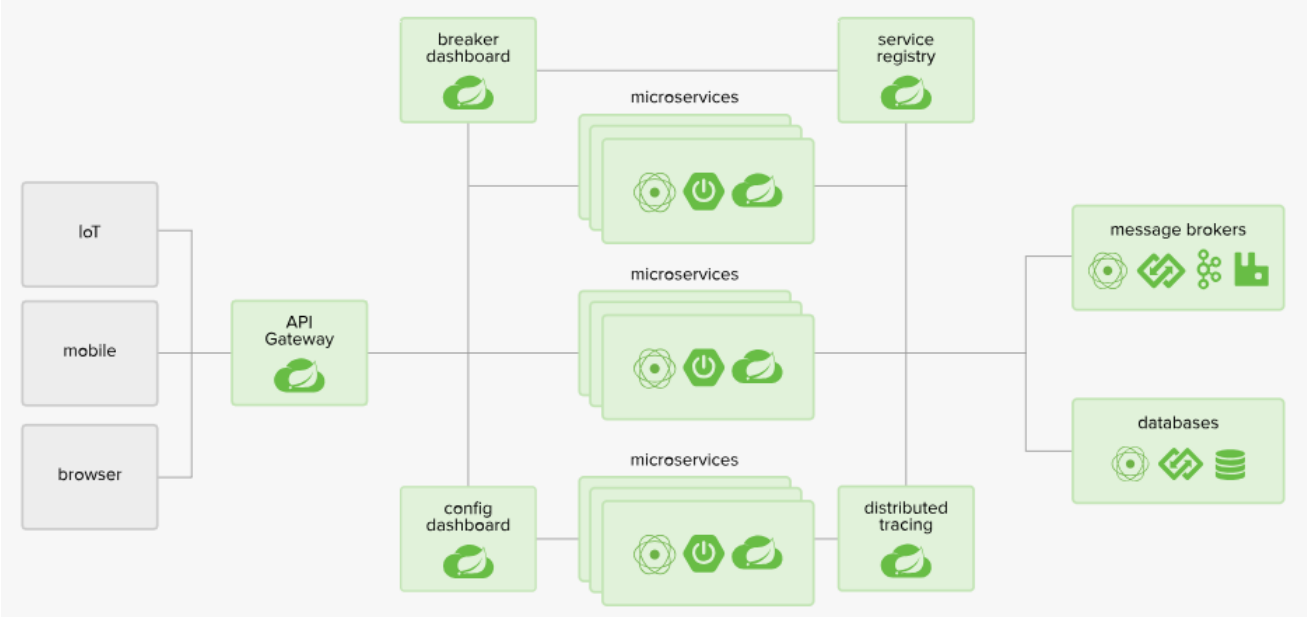
目前IT公司用的微服务架构有哪些？

- 阿里：Dubbo/HSF
- 京东：JSF
- 新浪微博：Motan
- 当当网：DubboX

6、什么是SpringCloud？（面试）

是一个基于Spring Boot的快速构建分布式系统的工具集。

基于SpringBoot提供了一套微服一站式务解决方案，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、熔断器等组件，除了基于NetFilx的开源组件做高度抽象封装之外，还有一些选型中立的开源组件。 分布式微服务架构下的一站式解决方案，是各个微服务架构落地技术的集合体，俗称微服务全家桶。



中文官网：https://springcloud.cc/

中国社区：http://www.springcloud.cn/

GitHub：https://github.com/spring-cloud

7、SpringBoot 和 SpringCloud 关系？（面试）

SpringBoot专注于快速方便的开发单个个体微服务。SpringCloud是关注全局的微服务协调治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等集成服务。 SpringBoot可以离开SpringCloud独立使用开发项目，但SpringCloud离不开SpringBoot，属于依赖关系。

8、SpringCloud 和 Dubbo 区别？（面试）

最大区别：SpringCloud抛弃了Dubbo的RPC通信，采用的是HTTP的REST方式。

严格来说，这两种方式各有优劣。虽然从一定程度上讲，REST牺牲了服务调用的性能，但也避免了原生RPC带来的问题。而且REST相比RPC更灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更加合适。

核心要素	Dubbo	Spring Cloud
服务注册中心	Zookeeper、Redis	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud Netflix Zuul
断路器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
分布式追踪系统	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream 基于Redis,Rabbit,Kafka实现的消息微服务
批量任务	无	Spring Cloud Task

Dubbo只是实现了服务治理，而Spring Cloud子项目分别覆盖了微服务架构下的众多部件，而服务治理只是其中的一个方面。Dubbo提供了各种Filter，对于上述中"无"的要素，可以通过扩展Filter来完善。

二、微服务开发实例

通过一个部门管理微服务案例来讲解SpringCloud的相关技术。

1、基本底层结构

- microservicecloud 整体父工程Project
- microservicecloud-api 公共子模块Module
- microservicecloud-provider-dept-8001 部门微服务提供者Module
- microservicecloud-consumer-dept-80 部门微服务消费者Module

实例-1、新建 microservicecloud 父工程（Maven Project）

实例-2、microservicecloud-api 公共子模块Module

1. 新建 microservicecloud-api 公共子模块（Maven Module）
2. 修改POM文件
3. 新建部门Entity（Dept），配合lombok使用
4. java -jar lombok.jar 安装Eclipse插件
5. 在Java类上使用相关注解，如@Data、@NoArgsConstructor、@Accessors(chain = true)等
6. mvn clean install 将jar安装到本地库，给其它模块使用

实例-3、microservicecloud-provider-dept-8001 部门微服务提供者Module

1. 新建 microservicecloud-provider-dept-8001 （Maven Module）
2. 修改 pom 文件
3. yml 文件

4. 工程src/main/resources目录下新建mybatis文件夹后新建mybatis.cfg.xml文件
5. MySQL创建部门数据库脚本
6. DeptDao部门接口
7. 工程src/main/resources/mybatis目录下新建mapper文件夹后再建DeptMapper.xml文件
8. DeptService部门服务接口
9. DeptServiceImpl部门服务接口实现类
10. DeptController部门微服务提供者REST
11. DeptProvider8001_App主启动类
12. 测试
13. 最终工程展示

实例-4、microservicecloud-consumer-dept-80 部门微服务消费者 Module

1. 新建 microservicecloud-consumer-dept-80 （ Maven Module ）
2. 修改 pom 文件
3. yml 文件
4. cn.ruisiyuan.springcloud.cfgbeans包下新建ConfingBean
5. cn.ruisiyuan.springcloud.controller包下新建DeptController_Consumer 部门微服务消费者REST
6. DeptConsumer80_App主启动类
7. 测试
    1. 启动DeptProvider8001_App 和 DeptConsumer80_App
    2. http://localhost/consumer/dept/get/2
    3. http://localhost/consumer/dept/list
    4. http://localhost/consumer/dept/add?dname=AI

2、Eureka 服务注册与发现

2.1、基本概念

Eureka是Netflix的一个子模块，也是核心模块之一。Eureka是基于REST的服务，用于定位服务，以实现云端中间层服务发现和故障转移。服务注册与发现对于微服务架构来说非常重要，有了服务注册与发现，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件了。功能类似于dubbo的注册中心ZooKeeper。

2.2、Eureka原理？

SpringCloud封装了Netflix公司开发的Eureka模块来实现服务注册和发现（ZooKeeper）。

Eureka采用了C-S的设计架构，Eureka Server作为服务注册功能的服务器，它是服务注册中心。而系统的其它微服务，使用Eureka 客户端连接到 Eureka Server并维持心跳连接。这样系统的维护人员就可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。SpringCloud的一些其它模块（比如Zuul）就可以通过 Eureka Server 来发现系统的其它微服务，并执行相关逻辑。

Eureka 包含两大组件：Eureka Server、Eureka Client

Eureka Server 提供服务注册服务，各个节点启动后，会在 Eureka Server中进行注册，这样 Eureka Server中的服务注册表中将会存储所有可用服务的节点信息，服务节点的信息可以在界面中直观看到。

Eureka Client是一个Java客户端用于简化Eureka Server的交互，客户端同时也具备一个内置的、使用轮询（round-robin）负载算法的负载均衡器。在应用启动后，将会向 Eureka Server 发送心跳（默认周期为30秒）。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，Eureka Server 将会从服务注册表中把这个服务节点移除（默认90秒）。

2.3、如何实现？

- Eureka Server 提供服务注册和发现。
- Service Provider服务提供方将自身服务注册到Eureka，从而使服务消费方找到。
- Service Consumer服务消费方从Eureka获取服务注册列表，从而能够消费服务。

实例-5、microservicecloud-eureka-7001 eureka服务注册中心Moduel

1. 新建 microservicecloud-eureka-7001（Maven Module）
2. POM
3. YML
4. EurekaServer7001_App主启动类，加入 @EnableEurekaServer 注解
5. 测试 http://localhost:7001

实例-6、将 microservicecloud-provider-dept-8001 部门微服务注册进 Eureka 服务中心

1. 修改 POM，加入 eureka 依赖
2. 修改 YML，加入eureka配置
3. 修改 DeptProvider8001_App 启动类，加入 @EnableEurekaClient
4. 启动测试 http://localhost:7001
    1. 启动 EurekaServer7001_App
    2. 启动 DeptProvider8001_App

实例-7、正常显示微服务的info信息

1. 主机名称：服务名修改
2. 访问信息有IP信息提示
3. 微服务info内容详细信息
    1. 修改 microservicecloud-provider-dept-8001 pom文件添加 actuator 依赖
    2. 总的父工程 microservicecloud 修改pom 添加构建build信息
    3. microservicecloud-provider-dept-8001

2.4、Eureka 自我保护机制？AP原则

默认情况下，如果Eureka Server在一定时间内没有接收到某个微服务实例的心跳，Eureka Server将会注销该实例（默认90秒）。但是当网络分区故障发生时，微服务与Eureka Server之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。

Eureka通过“自我保护模式”来解决这个问题——当Eureka Server节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。一旦进入该模式，Eureka Server就会保护服务注册表中的信息，不再删除服务注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该Eureka Server节点会自动退出自我保护模式。

综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

SpringCloud中，可以使用**eureka.server.enable-self-preservation = false**禁用自我保护模式。

实例-8、microservicecloud-provider-dept-8001 服务发现Discovery（理解即可）

对于注册进eureka里面的微服务，可以通过服务发现来获得该服务的信息。

1. 修改 microservicecloud-provider-dept-8001 工程的 DeptController
2. 修改 DeptProvider8001_App主启动类
3. 自测 http://localhost:8001/dept/discovery
4. 修改 microservicecloud-consumer-dept-80 工程的 DeptController_Consumer

实例-9、Eureka 集群配置

1. 新建 microservicecloud-eureka-7002、microservicecloud-eureka-7003
2. 按照 microservicecloud-eureka-7001 为模板粘贴 POM
3. 修改 7002、7003主启动类
4. 修改映射配置
    1. 找到 C:\Windows\System32\drivers\etc 目录下的hosts文件，添加如下信息：
    2. 127.0.0.1 eureka7001.com
    3. 127.0.0.1 eureka7002.com
    4. 127.0.0.1 eureka7003.com
5. 3台Eureka服务器的yml配置修改
6. 修改microservicecloud-provider-dept-8001 yml文件， 微服务发布到上面3台Eureka集群配置中
7. 测试
    1. http://eureka7001.com:7001/
    2. http://eureka7002.com:7002/
    3. http://eureka7003.com:7003/

2.5、作为服务注册中心 Eureka 比 Zookeeper 好在哪里？（面试）

1、Eureka 遵守AP原则，Zookeeper 遵守CP原则。

根据CAP理论，一个分布式系统不可能同时满足一致性、可用性和分区容错性，由于分区容错性是分布式 系统中必须保证的，因此我们只能在一致性和可用性之间权衡。

2、Zookeeper采用CP，节点采用主从。一旦主机down机，会在多个从中进行决策选举一个从作为主，但是 选举的时间30~120s，时间太长，在选举期间会导致集群不可用，这样就会导致整个注册中心瘫痪。

3、Eureka采用AP，所有节点平等，没有主从。如果有一个节点挂掉，会自动切换到一个可用的节点，只要 有一个Eureka节点正常运行，就能保证注册中心可用。只不过查到的信息可能不是最新的（不保证强一致性）。 除此之外Eureka还有自我保护机制。因此Eureka可以很好的应对因网络故障导致部分节点失去联系的情况， 而不会像Zookeeper那样使整个注册服务瘫痪。

2.6、CAP 原则？

RDBMS(Oracle/MySQL/SQLServer) ===> ACID

NOSQL(Redis/MongoDB/HBase) ===> CAP

1）、传统的ACID分别是什么？

A（Atomicity）原子性

一个事务的所有系列操作步骤被看成是一个动作，所有的步骤要么全部完成要么一个也不会完成，如果事务过程中任何一点失败，将要被改变的数据库记录就不会被真正被改变。

C（Consistency）一致性

数据库的约束、级联和触发机制Trigger都必须满足事务的一致性。也就是说，通过各种途径包括外键约束等任何写入数据库的数据都是有效的，不能发生表与表之间存在外键约束，但是有数据却违背这种约束性。所有改变数据库数据的动作事务必须完成，没有事务会创建一个无效数据状态，这是不同于CAP理论的一致性"consistency"。

I（Isolation）隔离性

主要用于实现并发控制, 隔离能够确保并发执行的事务能够顺序一个接一个执行，通过隔离，一个未完成事务不会影响另外一个未完成事务。

D（Durability）持久性

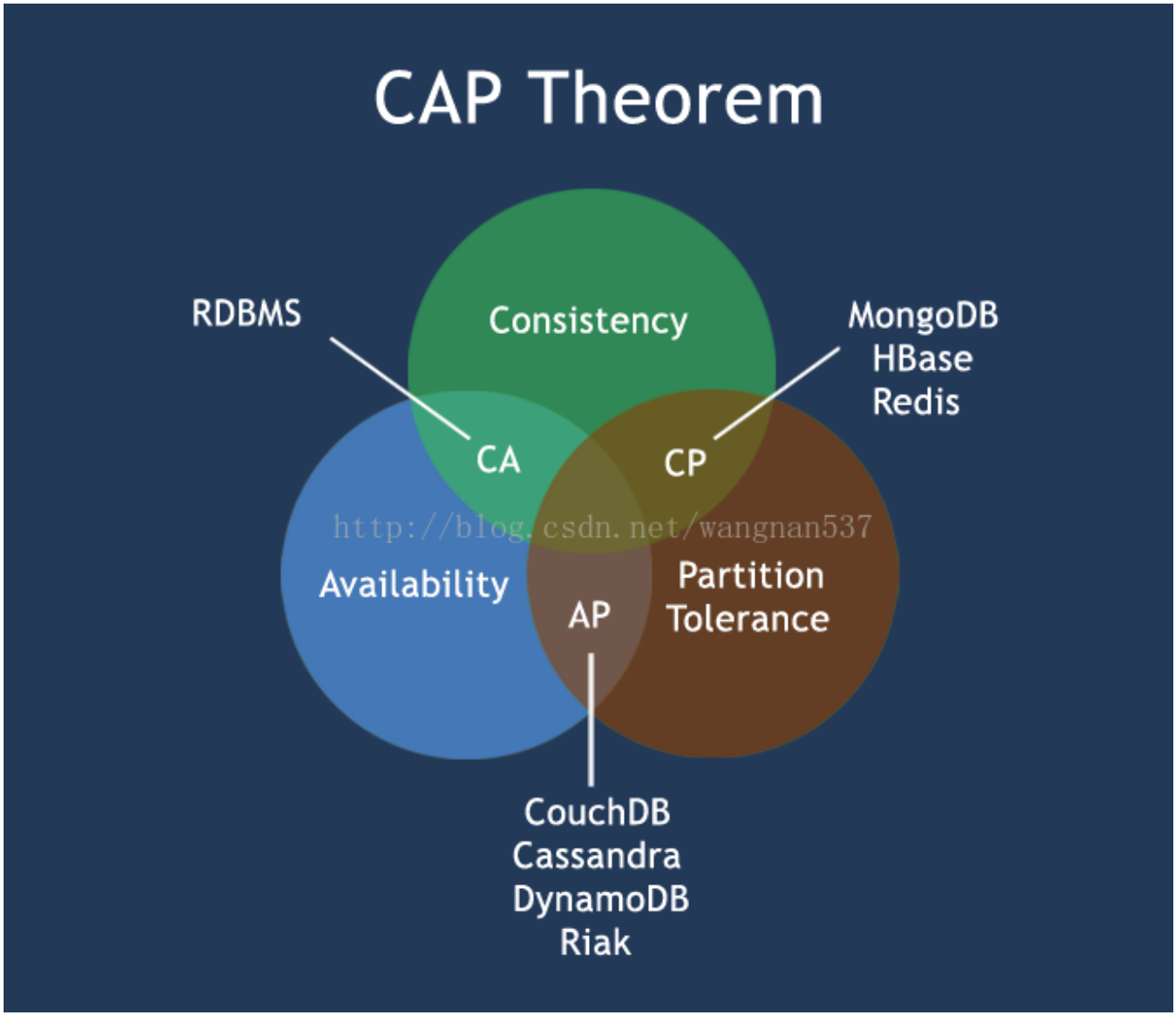
一旦一个事务被提交，它应该持久保存，不会因为和其他操作冲突而取消这个事务。很多人认为这意味着事务是持久在磁盘上，但是规范没有特别定义这点。

2）、CAP C（Consistency）强一致性

同样数据在分布式系统中所有地方都是被复制成相同。 **A（Availability）可用性**

所有在分布式系统活跃的节点都能够处理操作且能响应查询。 **P（Partition tolerance）分区容错性**

在两个复制系统之间，如果发生了计划之外的网络连接问题，对于这种情况，有一套容错性设计来保证。



CAP是分布式系统中进行平衡的理论，它是由 Eric Brewer发布在2000年。

CAP原则是NoSQL数据库的基石。

一般情况下CAP理论认为你不能同时拥有上述三种，只能同时选择两种，这是一个实践总结，当有网络分区情况下，也就是分布式系统中，你不能又要有完美一致性和100%的可用性，只能这在两者选择一个。在单机系统中，你则需要在一致性和延迟性latency之间权衡。

CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性、可用性和分区容错性这三个需求。因此根据CAP原理，将NoSQL数据库分成了满足CA原则、满足CP原则、满足AP原则三类。

- CA原则：单点集群，满足一致性、可用性的系统，通常在可扩展性上不太强大。
- CP原则：满足一致性、分区容错性的系统，通常性能不是特别高。
- AP原则：满足可用性、分区容错性的系统，通常可能对一致性要求低些。

3）、CAP的3进2？

CAP的理论就是说在分布式存储系统中，最多只能实现上面的两点。而由于当前网络硬件肯定会出现延迟丢包等问题，所以分区容错性是必须要实现的。

淘宝双11当天应该怎样选择？AP原则

在分布式数据库中CAP原理和BASE思想？

3、Ribbon负载均衡

3.1、什么是Ribbon？

SpringCloud Ribbon是基于NetFlix Ribbon实现的一套 **客户端 负载均衡 工具**。

3.2、负载均衡（Load Balance）

在微服务或分布式系统中经常用到的一种应用。负载均衡简单的说就是将用户的请求平摊的分配到多个服务上，从而达到系统的HA（High Availability高可用）。

常见的负载均衡有软件Nginx、LVS，硬件F5等。

相应的中间件，如Dubbo和SpringCloud中都给我们提供了负载均衡，SpringCloud的负载均衡算法可以自定义。

分类：

集中式LB，即在服务的消费方和提供方之间使用独立的LB设施（可以是硬件，如F5，也可以是软件，如Nginx），由该设施负责把访问请求通过某种策略转发至服务提供方。

进程内LB，将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后再从这些地址中选择出一个合适的服务器。Ribbon就是属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

实例-10、Ribbon初步配置

1. 修改 microservicecloud-consumer-dept-80 工程 POM 文件，加入Ribbon相关依赖
2. 修改 yml 文件，追加 Eureka 的服务注册地址
3. 对 ConfigBean 进行新注解 @LoadBalanced，获取RestTemplate时加入Ribbon配置
4. 主启动类 DeptConsumer80_App 添加 @EnableEurekaClient 注解
5. 修改 DeptController_Consumer 客户端访问类
6. 启动
    1. 先启动三个Eureka集群
    2. 再启动 microservicecloud-provider-dept-8001 并注册进 Eureka
    3. 启动 microservicecloud-consumer-dept-80
7. 测试
    1. http://localhost/consumer/dept/get/2
    2. http://localhost/consumer/dept/list
    3. http://localhost/consumer/dept/add?dname=BigData

实例-11、Ribbon负载均衡

1. 参考 microservicecloud-provider-dept-8001，新建两份，分别命名为8002、8003
2. 新建 8002、8003数据库，各自微服务连各自数据库
3. 修改 8002、8003 各自的 yml 文件
4. 启动3个Eureka集群，启动3个Dept微服务，并各自测试通过
    1. http://localhost:8001/dept/list
    2. http://localhost:8002/dept/list

3. http://localhost:8003/dept/list
4. 启动 microservicecloud-consumer-dept-80

5. 客户端通过Ribbon完成负载均衡，并访问上一步的Dept微服务
6. 总结：Ribbon其实就是一个软负载均衡的客户端组件，它可以和其它所需请求的客户端结合使用，
7. 和Eureka结合只是其中的一个实例。

3.3、SpringCloud提供的负载均衡算法

所有算法都实现了 IRule 接口，提供了7种算法，如轮询，随机，根据响应时间加权等。

- **RoundRobinRule（轮询）**：默认，挨个一次一次访问
- **WeightedResponseTimeRule**：根据平均响应时间计算所有服务的权重，响应时间越快服务权重越大，被选中的概率越高。刚启动时如果统计信息不足，则使用RoundRobinRule轮询策略，等统计信息足够，会切换到WeightedResponseTimeRule。
- **RandomRule（随机）**：
- **AvailabilityFilteringRule**：会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，还有并发的连接数量超过阈值的服务，然后对剩余的服务列表按照轮询策略进行访问。
- **ZoneAvoidanceRule**：
- **RetryRule**：先按照RoundRobinRule轮询策略获取服务，如果获取服务失败则在指定时间内进行重试，获取可用服务。
- **BestAvailableRule**：会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量最小的服务。

3.4、切换负载均衡算法

将一个实现了IRule接口的负载均衡算法作为一个Bean配置在Spring IOC 容器中即可进行自动切换。

实例-12、自定义负载均衡算法

1. 新建cn.ruisiyuan.myrule包，在其新建包下新建自定义Robbin规则类，（注意这个类不能建在和主启动类相同的包下，否则不起作用）。
2. 主启动类 DeptConsumer80_App 添加 @RibbonClient 注解来声明自定义Robbin规则类。

4、Feign负载均衡

4.1、什么是Feign？

Feign是一个声明式**WebService客户端**。使用Feign能让编写WebService客户端更简单，它是使用是定义一个接口，然后在上面添加注解，同时也支持JAX-RS标准的注解。Feign也支持可拔插式的编码器和解码器SpringCloud对Feign进行了封装，使其支持了SpringMVC标准注解和HttpMessageConverters。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

4.2、Feign能干什么？

Feign旨在使编写Java Http客户端变得更加容易。

前面在使用Ribbon+RestTemplate时，利用RestTemplate对http请求的封装处理，形成了一套模板化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所有多会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用。所以，Feign在此基础了做了进一步封装，由它来帮我们定义和实现依赖服务接口的定义，在Feign的实现下，我们只需要创建一个接口并使用注解的方式来配置它，即可

完成对服务提供方的接口绑定，简化了使用Springcloud Ribbon时，自己封装服务调用客户端的开发量。

Feign集成了Ribbon，利用Ribbon维护了microservicecloud-dept的服务列表信息，并且通过轮询实现了客户端的负载均衡。而与Ribbon不同的是，通过Feign只需要定义服务绑定接口且以声明式的方法，优雅而简单的实现了服务调用。

Feign通过接口的方法调用Rest服务（之前是Ribbon + RestTemplate），该请求发送给Eureka服务器（http://MICROSERVICECLOUD-DEPT/dept/list），通过Feign直接找到服务接口，由于在进行服务调用的时候融合了Ribbon技术，所以也支持负载均衡作用。

Feign：面向接口编程

Ribbon + RestTemplate：面向服务名编程

实例-13、Feign使用

1. 新建 microservicecloud-consumer-dept-feign 工程，参考 microservicecloud-consumer-dept-80
2. POM，添加对 Feign 的支持
3. 修改 microservicecloud-api 工程
4. POM
5. 新建DeptClientService接口，并新增注解 @FeignClient
6. Maven clean install
7. 修改 microservicecloud-consumer-dept-feign 工程 Controller，添加上一步新建的 DeptClientService接口
8. 修改 DeptConsumer80_Feign_App 主启动类，加入 @EnableFeignClients 注解
9. 测试

4.3、Nginx、Ribbon、Feign区别？

Nginx、Ribbon、Feign都可以实现负载均衡。

Nginx是一个反向代理、负载均衡的服务器，是在消费端和服务提供者之间的一个设施。

Ribbon是一个集成在消费端的进程内的一个软负载均衡，用来为客户端提供负载均衡功能。

Feign是用来简化消费端调用微服务，以面向接口编程的方式来调用，且内置了Ribbon负载均衡。

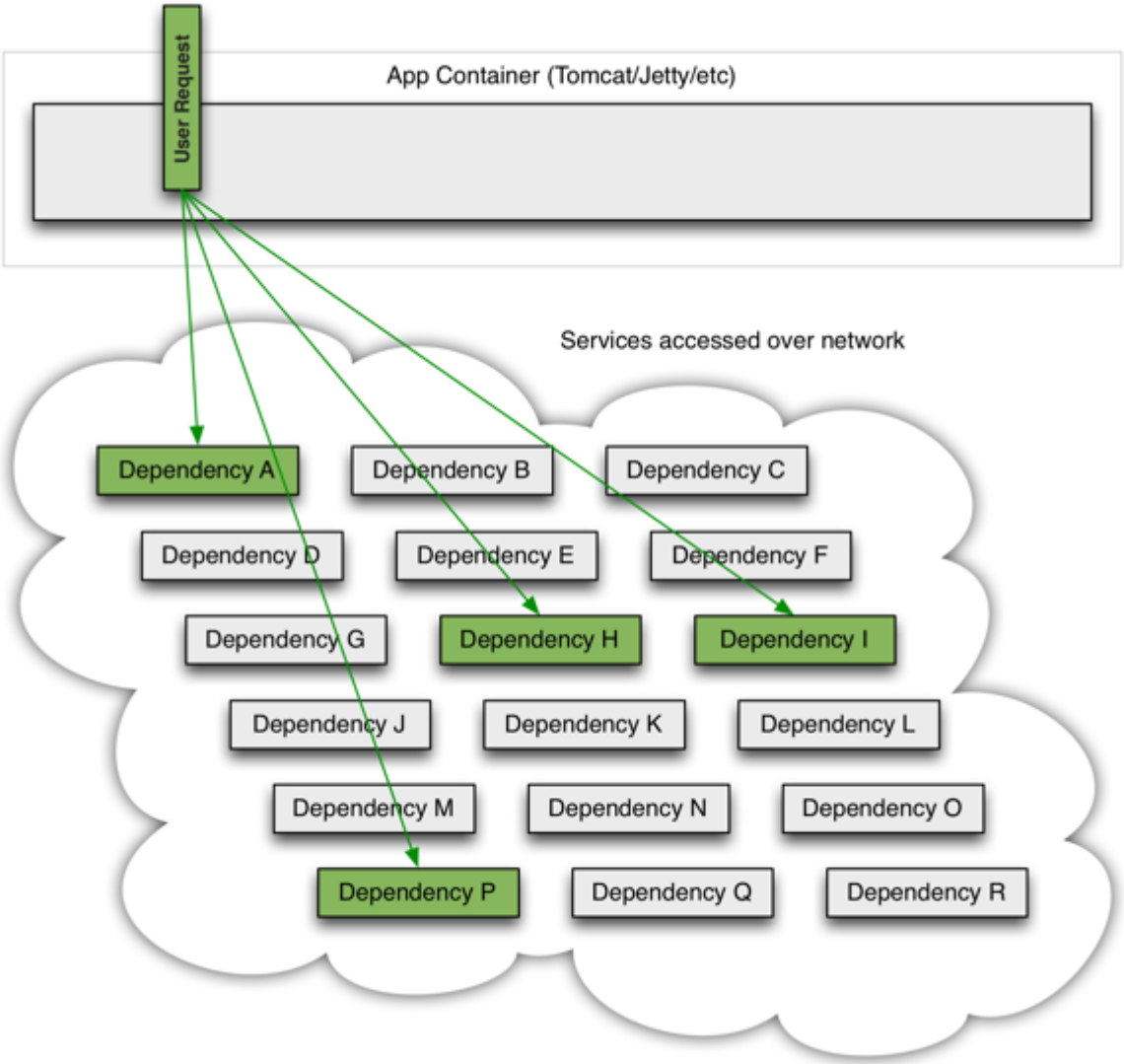
5、Hystrix断路器（熔断器）

5.1、分布式系统面临的问题？

复杂分布式系统结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免的失败。

5.2、服务雪崩？

多个微服务之间调用的时候，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其它微服务，这就是所谓的“扇出”。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“雪崩效应”（是一种因服务提供者的不可用导致服务调用者的不可用,并将不可用逐渐放大的过程）。



5.3、Hystrix是什么？

Hystrix是一个用于处理分布式系统的延迟和容错的开源库（NetFlix）。在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix能够保证在一个依赖出问题情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

断路器本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要的占用，从而避免了故障在分布式系统的蔓延，乃至雪崩。

Hystrix的设计原则：

- 资源隔离
- 熔断器
- 命令模式

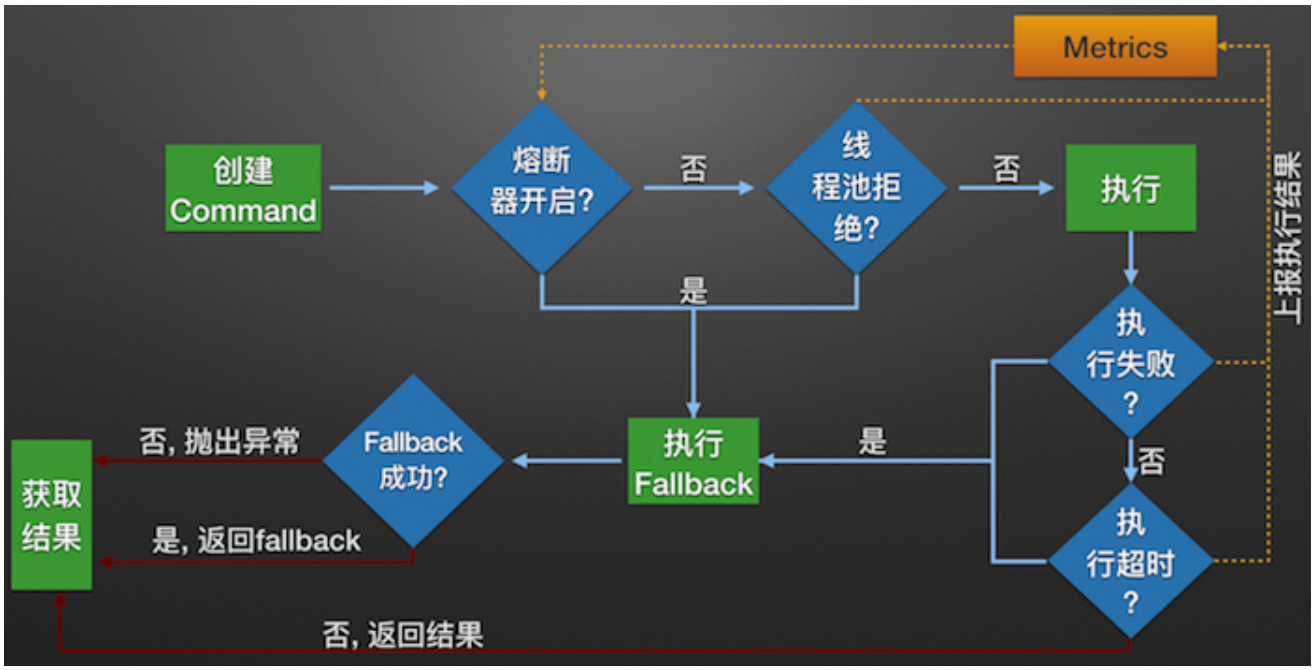
5.4、Hystrix能干什么？

服务熔断、服务降级、服务限流、服务监控等。

5.5、服务熔断？（Provider端）

熔断机制是应对雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回错误的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里，熔断机制通过Hystrix实现，Hystrix会监控微服务间的调用状况，当失败的调用到一定的阈值，缺省是5秒内20次调用失败就会启动熔断机制。熔断机制的注解是 @HystrixCommand。



实例-14、服务熔断实现

1. 新建 microservicecloud-provider-dept-hystrix-8001 工程，参考 microservicecloud-provider-dept-8001
2. POM，加入 Hystrix 依赖
3. YML，修改主机服务名
4. 修改 DeptController
5. 修改 DeptProvider8001_Hystrix_App，增加 @EnableCircuitBreaker 注解，对Hystrix熔断器支持
6. 测试 http://localhost/consumer/dept/get/100
    1. 启动3个Eureka集群
    2. 启动DeptProvider8001_Hystrix_App
    3. 启动DeptConsumer80_Feign_App

5.6、服务降级？（Consumer端）

当服务器压力剧增的情况下，根据实际业务情况及流量，对一些服务和页面有策略的不处理或换种简单的方式处理，从而释放服务器资源以保证核心交易正常运作或高效运作。

服务降级一般是从整体负荷考虑，就是当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的fallback回调，返回一个缺省值。这样做，虽然服务水平下降，但好歹可用，比直接挂掉要强。

服务降级处理是在客户端完成的，与服务端没有关系。

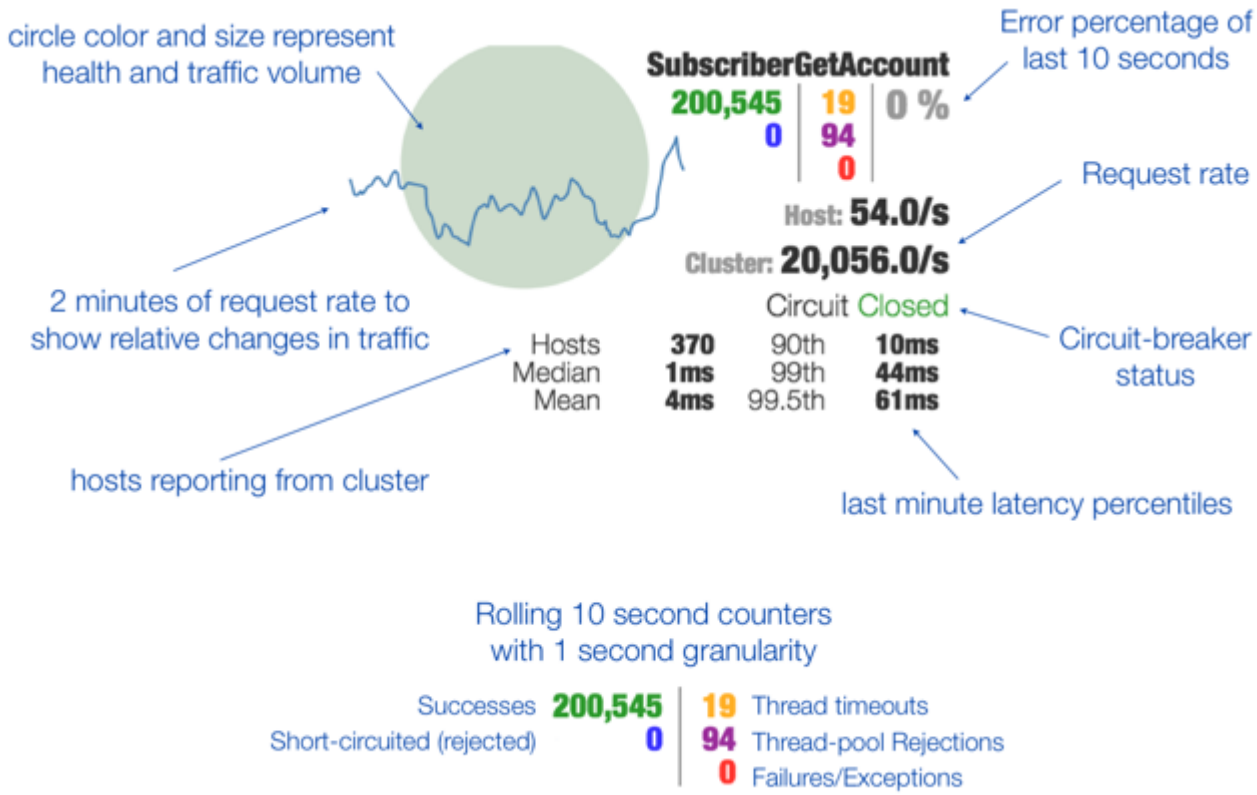
实例-15、服务降级实现

1. 修改 microservicecloud-api 工程，
    1. 根据已有的DeptClientService接口创建一个实现了FallbackFactory接口的类 DeptClientServiceFallbackFactory。
    2. 在DeptClientService的 @FeignClient 注解中添加fallbackFactory。
2. microservicecloud-api 工程 执行 mvn clean install
3. microservicecloud-consumer-dept-feig 工程修改 YML 文件
4. 测试
    1. 启动3个Eureka服务器
    2. 启动microservicecloud-provider-dept-8001
    3. 启动microservicecloud-consumer-dept-feign
    4. 正常访问测试 http://localhost/consumer/dept/get/1
    5. 故意关闭微服务 microservicecloud-provider-dept-8001

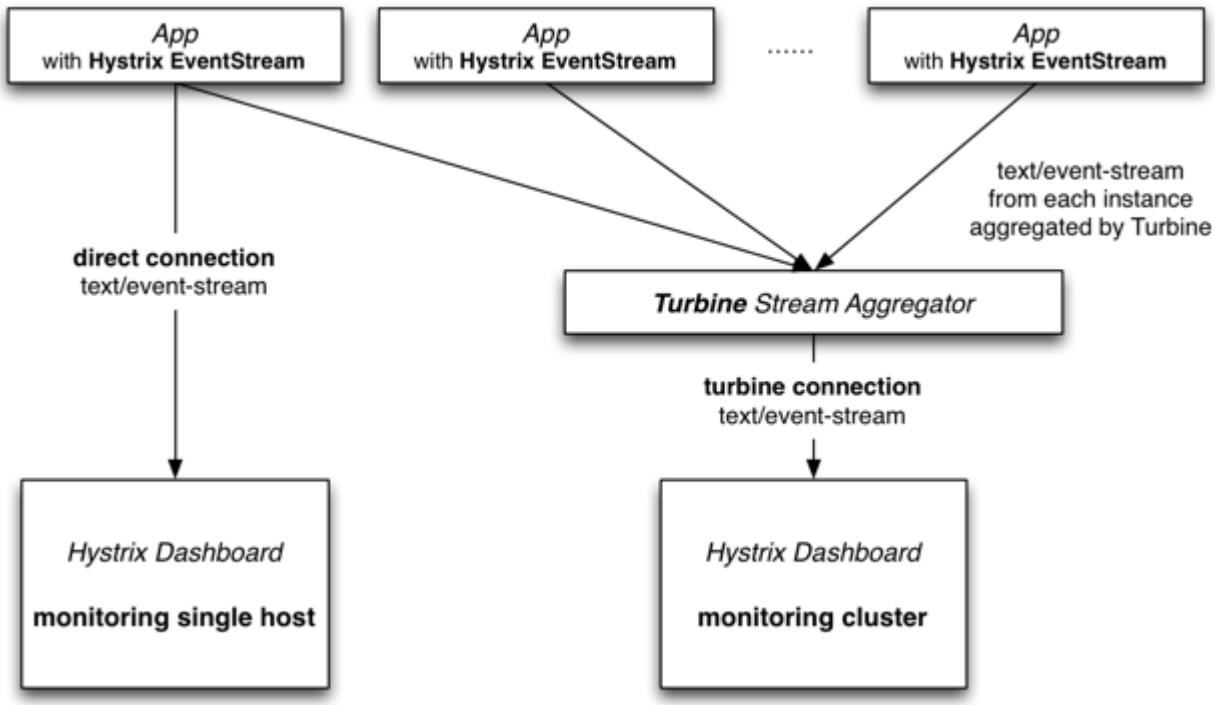
5.7、服务监控（Hystrix Dashboard）

除了隔离依赖服务的调用之外，Hystrix还提供了准实时的调用监控（Hystrix Dashboard），Hystrix会持续的记录所有通过Hystrix发起请求的执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求、多少成功、多少失败等。

NetFlix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。SpringCloud也提供了Hystrix Dashboard的整合，对监控内容转化为可视化界面。



hystrix dashboard 可以用来监控单个服务或者通过turbin来监控整个集群。



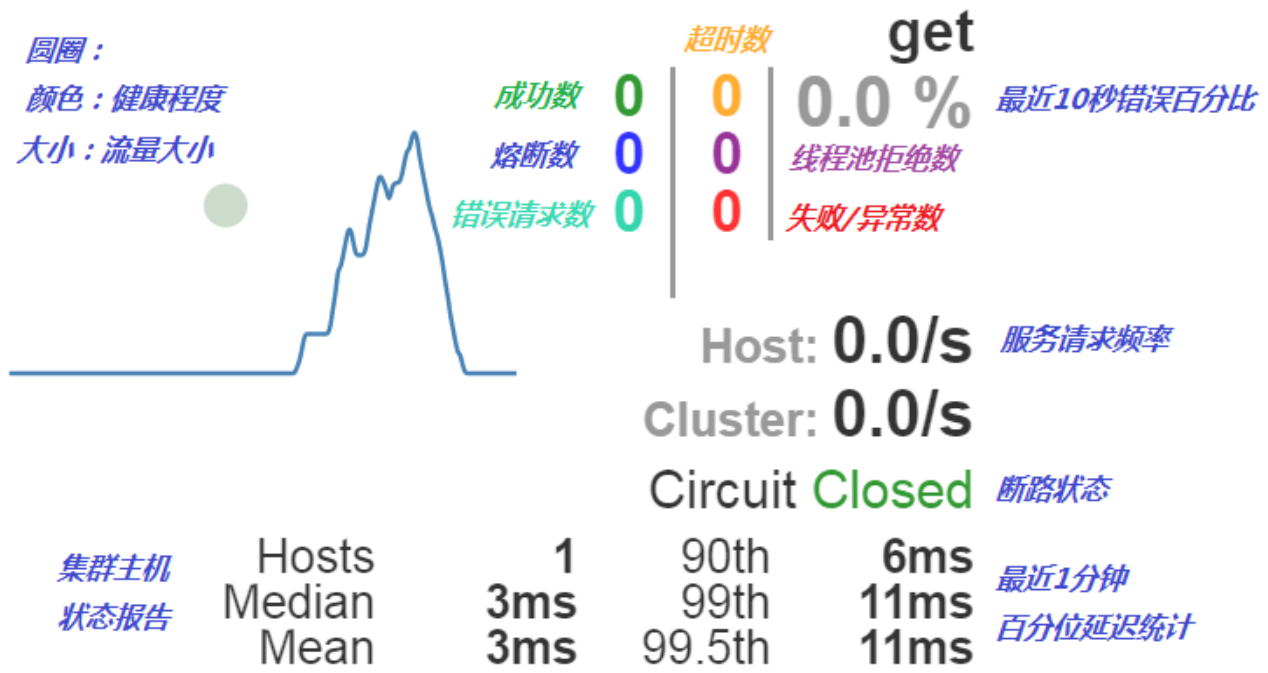
实例-16、服务监控实现

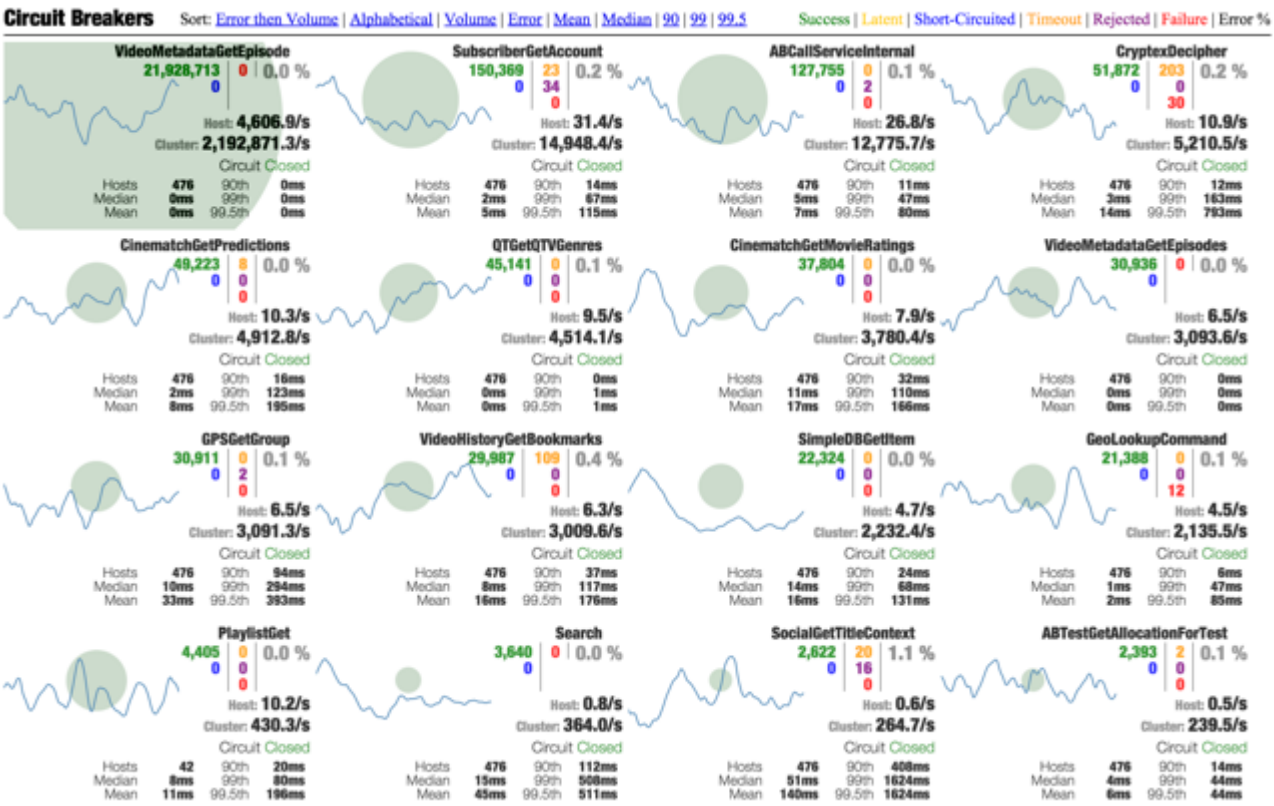
1. 新建 microservicecloud-consumer-hystrix-dashboard 工程
2. POM
3. YML
4. 新建主启动类 DeptConsumer_Dashboard_App，并添加 @EnableHystrixDas启动 3个Eureka集群
5. 启动 microservicecloud-provider-dept-hystrix-8001
6. 启动 microservicecloud-consumer-hystrix-dashboard
7. 监控测试（监控的微服务必须有熔断器 HystrixCommand，否则无法显示）
    1. http://localhost:8001/dept/1
    2. http://localhost:8001/hystrix.stream
8. 观察监控窗口
    1. http://localhost:9001/hystrix 进入 Hystrix Dashboard 启动页面。
    2. 在 Hystrix Dashboard 启动界面填写：
        监控地址： http://localhost:8001/hystrix.stream
        Delay：2000 ms（该参数用来控制服务器上轮询监控信息的延迟时间，默认为2000毫秒，可以通过配置该属性来降低客户端的网络和CPU消耗）
        Title：监控结果页面标题，默认会使用具体监控实例的URL
    3. 查看监控结果
        多次刷新 http://localhost:8001/dept/1 ， 查看Hystrix Dashboard 监控面板

7色：

1圆：实心圆，共有两层含义，它通过颜色的变化代表了实例的健康程度，健康程度从 绿色<黄色<橙色<红色 递减。 该实心圆除了颜色变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过 该实心圆的展示，就可以在大量的实例中快速的发现故障实例和高压力实例。

1线：用来记录2分钟内流量的相对变化，可以通过它来观察到流量的上升和下降趋势。





6、Zuul路由网关

6.1、Zuul是什么？

Zuul包含了对请求的路由和过滤两个最主要的功能。

其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础，而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础。

Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其它微服务的消息，也即以后的访问微服务都是通过Zuul跳转后获得。

注意：Zuul服务最终还是会注册进Eureka。

提供：代理 + 路由 + 过滤 三大功能

实例-17、路由基本配置

1. 新建 microservicecloud-zuul-gateway-9527 工程（Maven Module）
2. POM
3. YML
4. hosts 修改：127.0.0.1 gateway-9527.com
5. 主启动类 Zuul_9527_StartSpringCloud_App，添加 @EnableZuulProxy 注解
6. 启动
7. 启动 3个Eureka集群（服务注册）
8. 启动 microservicecloud-provider-dept-8001（服务提供者）
9. 启动 microservicecloud-zuul-gateway-9527（路由）
10. 测试
    1. 不用路由：http://localhost:8001/dept/1
    2. 启动路由：http://myzuul.com:9527/microservicecloud-dept/dept/2

实例-18、路由访问映射规则

1）、修改 microservicecloud-zuul-gateway-9527 功能 YML 文件，增加路由映射规则。 http://myzuul.com:9527/xinyan/mydept/dept/1

```
zuul:
  prefix: /xinyan                                  #统一前缀
  #ignored-services: microservicecloud-dept        #忽略单个真实服务名
  ignored-services: "*"                            #忽略所有真实服务名，不能再直接使用真实服务名访问
  routes:
    dept-rout:
      serviceId: microservicecloud-dept            #服务名
      path: /mydept/**                             #映射路径
```

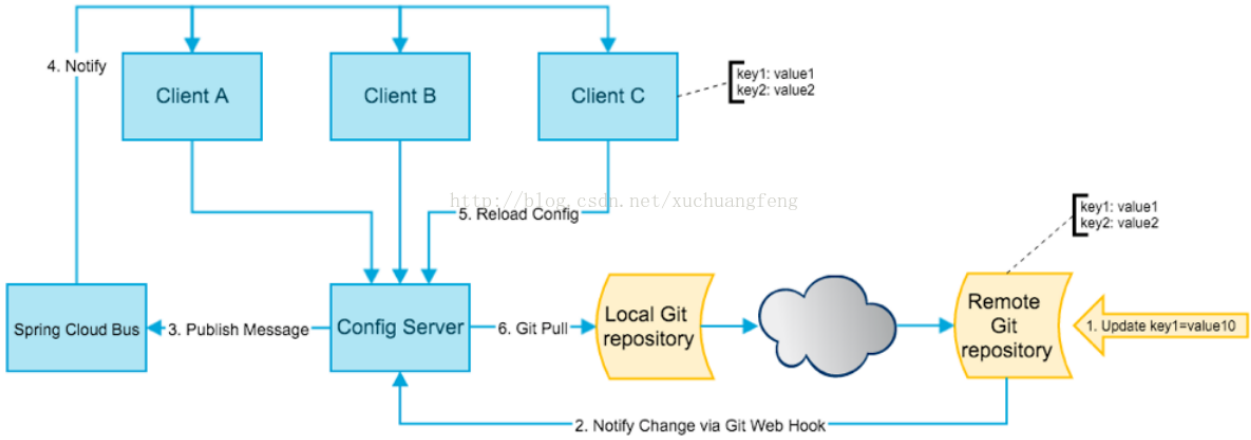
7、SpringCloud Config 分布式配置中心

分布式系统面临的问题---配置问题。

7.1、是什么？

SpringCloud Config为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。

下图是SpringCloudConfig结合SpringCloudBus实现分布式配置的工作流



SpringCloud Config分为服务端和客户端两部分。

服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密/解密信息等访问接口。

客户端则是通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取 和加载配置信息。

配置服务器默认采用git来存储配置信息，这样就有助于对环境配置进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容。

7.2、能干什么？

集中管理配置文件。

不同环境不同配置，动态化的配置更新，分环境部署，比如dev/test/prod/beta/release。 运行期间动态调整配置，不再需要在每个服务器部署的机器上编写配置文件，服务会向配置中心统一拉取配置自己的信息。当配置发生变动时，服务不需要重启即可感知到配置的变化并应用新的配置。

将配置信息以REST接口的形式暴露。

7.3、与GitHub整合配置

由于SpringCloud Config默认使用Git来存储配置文件（也有其它方式，比如支持SVN和本地文件），但最推荐的还是Git，而且使用的是http/https访问的形式。

实例-19、SpringCloud Config服务端配置

用自己的GitHub账号在GitHub上新建一个名为microservicecloud-config的新Repository

由上一步获得SSH 或 HTTPS 的git地址

本地硬盘目录上新建git仓库（C:\git\mySpringCloud），并clone

```
# SSH
$ git clone git@github.com:rsyxf/microservicecloud-config.git
# HTTPS
$ git clone https://github.com/xiefang1980/microservicecloud-config.git
```

在本地C:\git\mySpringCloud\microservicecloud-config里面新建一个application.yml

将上一步的YML文件推送到GitHub上，github 上查看上传结果

```
$ git add .
$ git commit -m "init yml"
$ git push origin master
```

新建 microservicecloud-config-3344 工程（Maven Module），它即为SpringCloud的配置中心

POM

YML

主启动类 Config_3344_StartSpringCloudApp，添加 @EnableConfigServer 注解

window下修改hosts文件，增加映射 127.0.0.1 config-3344.com

测试，通过Config微服务是否可以从GitHub上获取配置内容

1. 启动 微服务3344
2. http://config-3344.com:3344/application-dev.yml
3. http://config-3344.com:3344/application-test.yml
4. http://config-3344.com:3344/application-xxx.yml（不存在的配置）

配置读取规则

1. /{application}-{profile}.yml

   http://config-3344.com:3344/application-dev.yml

   http://config-3344.com:3344/application-test.yml

2. /{application}/{profile}[/{label}]

   http://config-3344.com:3344/application/dev/master

   http://config-3344.com:3344/application/test/master

3. /{label}/{application}-{profile}.yml

   http://config-3344.com:3344/master/application-dev.yml

http://config-3344.com:3344/master/application-test.yml

实例-20、SpringCloud Config客户端配置与测试

1. 在本地 D:\Repository\mySpringCloud\microservicecloud-config 路径下新建文件

   microservicecloud-config-client.yml

   ```
   spring:
     profiles:
       active:
       - dev
   ---
   server:
     port: 8201
   spring:
   ```

```
  profiles: dev
  application:
    name: microservicecloud-config-client
eureka:
  client:
    service-url:
      defaultZone: http://eureka-dev.com:7001/eureka
---
server:
  port: 8202
spring:
  profiles: test
  application:
    name: microservicecloud-config-client
eureka:
  client:
    service-url:
      defaultZone: http://eureka-test.com:7001/eureka
```

2. 将 microservicecloud-config-client.yml 提交到GitHub中

```
$ git add .
$ git commit -m "add file"
$ git push origin master
```

3. 新建 microservicecloud-config-client-3355 工程
4. 修改 POM
5. 新建 bootstrap.yml
    1. application.yml 是用户级的资源配置项，bootstrap.yml是系统级的（优先级更高）
6. 新建 application.yml
7. windows下修改hosts文件，增加映射 127.0.0.1 client-config.com
8. 新建 ConfigClientRest 类，验证是否能从GitHub上读取配置文件
9. 主启动类ConfigClient_3355_StartSpringCloudApp
10. 测试
    1. 启动Config配置中心3344微服务并自测
    2. http://config-3344.com:3344/application-dev.yml
    3. http://config-3344.com:3344/application-test.yml
11. 启动3355作为Client访问
    1. http://client-config.com:8201/config ：profile:dev
    2. http://client-config.com:8202/config ：profile:test

实例-21、SpringCloud Config配置实战

目前情况：

1. Config服务端配置OK且测试通过，我们可以和config+GitHub进行配置修改并获得内容。

2. 我们做一个Eureka服务 + 一个Dept访问的微服务，将两个微服务的配置统一由github获得，实现统一配置分布式管理，完成多环境的变更。

实现步骤：

1. Git配置文件本地配置
    1. 在本地D:\Repository\mySpringCloud\microservicecloud-config路径下新建文件

       microservicecloud-config-eureka-client.yml

```yaml
spring:
  profiles:
    active:
    - dev
---
server:
  port: 7001

spring:
  profiles: dev
  application:
    name: microservicecloud-config-eureka-client

eureka:
  instance:
    hostname: eureka7001.com
  client:
    register-with-eureka: false     #当前的eureka-server自己不注册进服务列表中
    fetch-registry: false           #不通过eureka获取注册信息
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka/
---
server:
  port: 7001

spring:
  profiles: test
  application:
    name: microservicecloud-config-eureka-client

eureka:
  instance:
    hostname: eureka7001.com
  client:
    register-with-eureka: false     #当前的eureka-server自己不注册进服务列表中
    fetch-registry: false           #不通过eureka获取注册信息
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka/
```

    2. 在本地D:\Repository\mySpringCloud\microservicecloud-config路径下新建文件

       microservicecloud-config-dept-client.yml

```yaml
spring:
  profiles:
    active:
    - dev
---
server:
  port: 8001
spring:
  profiles: dev
  application:
    name: microservicecloud-config-dept-client
  datasource:
     type: com.alibaba.druid.pool.DruidDataSource            # 当前数据源操作类型
     driver-class-name: com.mysql.jdbc.Driver                # mysql驱动包
     url: jdbc:mysql://localhost:3306/clouddb03?
characterEncoding=utf8&serverTimezone=UTC&useSSL=false    # 数据库名称
     username: root
     password: 123456
     dbcp2:
       min-idle: 5                                           # 数据库连接池的最小维持连接
数
       initial-size: 5                                       # 初始化连接数
       max-total: 5                                          # 最大连接数
       max-wait-millis: 200                                  # 等待连接获取的最大超时时间

mybatis:
  config-location: classpath:mybatis/mybatis.cfg.xml        # mybatis配置文件所在路径
  type-aliases-package: com.xinyan.springcloud.entities     # 所有Entity别名类所在包
  mapper-locations: classpath:mybatis/mapper/*Mapper.xml    # mapper映射文件

eureka:
  client:
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka
    instance:
      instance-id: dept-8001.com
      prefer-ip-address: true

info:
  app.name: xinyan-microservicecloud-springcloudconfig01
  company.name: www.xinyanyuan.com
  build.artifactId: $project.artifactId$
  build.version: $project.version$
---
server:
  port: 8001
spring:
  profiles: test
  application:
    name: microservicecloud-config-dept-client
  datasource:
     type: com.alibaba.druid.pool.DruidDataSource            # 当前数据源操作类型
     driver-class-name: com.mysql.jdbc.Driver                # mysql驱动包
```

```
      url: jdbc:mysql://localhost:3306/clouddb02?
characterEncoding=utf8&serverTimezone=UTC&useSSL=false    # 数据库名称
      username: root
      password: 123456
      dbcp2:
        min-idle: 5                                          # 数据库连接池的最小维持连接
数
        initial-size: 5                                      # 初始化连接数
        max-total: 5                                         # 最大连接数
        max-wait-millis: 200                                 # 等待连接获取的最大超时时间

mybatis:
  config-location: classpath:mybatis/mybatis.cfg.xml        # mybatis配置文件所在路径
  type-aliases-package: com.xinyan.springcloud.entities     # 所有Entity别名类所在包
  mapper-locations:
  - classpath:mybatis/mapper/*Mapper.xml                      # mapper映射文件

eureka:
  client:
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka
    instance:
      instance-id: dept-8001.com
      prefer-ip-address: true

info:
  app.name: xinyan-microservicecloud-springcloudconfig01
  company.name: www.xinyanyuan.cn
  build.artifactId: $project.artifactId$
  build.version: $project.version$
```

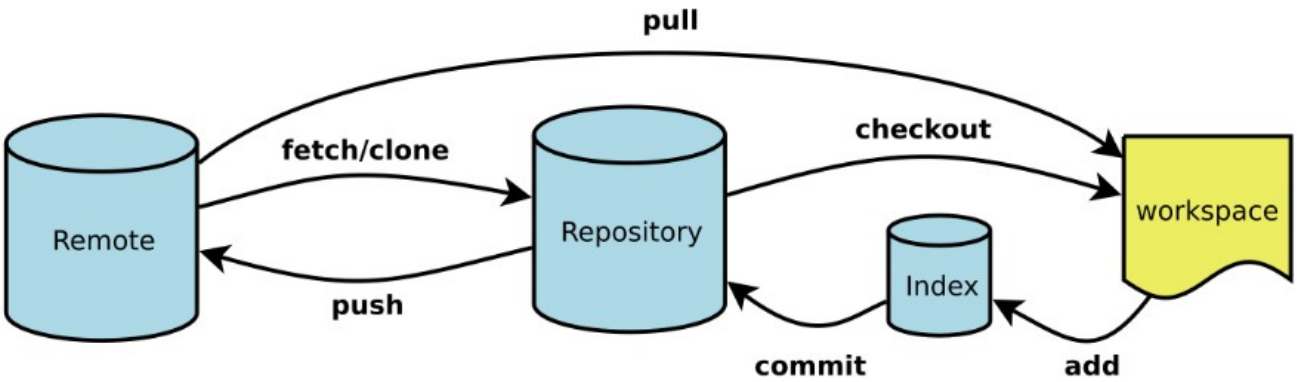
2. Config版的Eureka服务端
    1. 新建 microservicecloud-config-eureka-client-7001 工程
    2. POM
    3. bootstrap.yml，application.yml
    4. 主启动类Config_Git_EurekaServerApp
    5. 测试
        1. 先启动 microservicecloud-config-3344微服务，保证Config总配置是OK的
        2. 再启动 microservicecloud-config-eureka-client-7001 微服务
        3. http://eureka7001.com:7001 ,出现Eureka主页，表示启动成功。
3. Config版的Dept微服务
    1. 新建 microservicecloud-config-dept-client-8001 工程
    2. POM
    3. bootstrap.yml，application.yml
    4. 主启动类
    5. 测试 http://localhost:8001/dept/list

1. 启动 microservicecloud-config-3344
2. 启动 microservicecloud-config-eureka-client-7001
3. 启动 microservicecloud-config-dept-client-8001
4. 修改 microservicecloud-config-dept-client.yml 文件中dev的数据库为clouddb03，同步到github，再次访问，观看查询结果。

三、附录

1、Git 概述

Git是一款免费、开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。一般来说，日常使用Git只要记住下图6个命令，就可以了。但是熟练使用，恐怕要记住60～100个命令。



专用名词

```
Workspace：工作区
Index / Stage：暂存区
Repository：仓库区（或本地仓库）
Remote：远程仓库
```

2、Git命令手册

3、Git常用命令

3.1、新建代码库

```
# 在当前目录新建一个Git代码库
$ git init
# 新建一个目录，将其初始化为Git代码库
$ git init [project-name]
# 下载一个项目和它的整个代码历史
$ git clone [url]
```

3.2、配置

Git的配置文件为.gitconfig，它可以在用户主目录下（全局配置），也可以在项目目录下（项目配置）。

```
# 显示当前的Git配置
$ git config --list
# 编辑Git配置文件
$ git config -e [--global]
# 设置提交代码时的用户信息
$ git config [--global] user.name "[name]"
$ git config [--global] user.email "[email address]"
```

3.3、增加 / 删除文件

```
# 添加指定文件到暂存区
$ git add [file1] [file2] ...
# 添加指定目录到暂存区，包括子目录
$ git add [dir]
# 添加当前目录的所有文件到暂存区
$ git add .
# 删除工作区文件，并且将这次删除放入暂存区
$ git rm [file1] [file2] ...
# 停止追踪指定文件，但该文件会保留在工作区
$ git rm --cached [file]
# 改名文件，并且将这个改名放入暂存区
$ git mv [file-original] [file-renamed]
```

3.4、代码提交

```
# 提交暂存区到仓库区
$ git commit -m [message]
# 提交暂存区的指定文件到仓库区
$ git commit [file1] [file2] -m [message]
# 提交工作区自上次commit之后的变化，直接到仓库区
$ git commit -a
# 提交时显示所有diff信息
$ git commit -v
# 使用一次新的commit，替代上一次提交
# 如果代码没有任何新变化，则用来改写上一次commit的提交信息
$ git commit --amend -m
 [message]
# 重做上一次commit，并包括指定文件的新变化
$ git commit --amend
```

3.5、分支

```
# 列出所有本地分支
$ git branch
# 列出所有远程分支
$ git branch -r
# 列出所有本地分支和远程分支
$ git branch -a
# 新建一个分支，但依然停留在当前分支
$ git branch [branch-name]
# 新建一个分支，并切换到该分支
$ git checkout -b [branch]
# 新建一个分支，指向指定commit
$ git branch [branch] [commit]
# 新建一个分支，与指定的远程分支建立追踪关系
$ git branch --track [branch] [remote-branch]
# 切换到指定分支，并更新工作区
```

```
$ git checkout [branch-name]
# 建立追踪关系，在现有分支与指定的远程分支之间
$ git branch --set-upstream [branch] [remote-branch]
# 合并指定分支到当前分支
$ git merge [branch]
# 选择一个commit，合并进当前分支
$ git cherry-pick [commit]
# 删除分支
$ git branch -d [branch-name]
# 删除远程分支
$ git push origin --delete
$ git branch -dr
```

3.6、标签

```
# 列出所有tag
$ git tag
# 新建一个tag在当前commit
$ git tag [tag]
# 新建一个tag在指定commit
$ git tag [tag] [commit]
# 查看tag信息
$ git show [tag]
# 提交指定tag
$ git push [remote] [tag]
# 提交所有tag
$ git push [remote] --tags
# 新建一个分支，指向某个tag
$ git checkout -b [branch] [tag]
```

3.7、查看信息

```
# 显示有变更的文件
$ git status
# 显示当前分支的版本历史
$ git log
# 显示commit历史，以及每次commit发生变更的文件
$ git log --stat
# 显示某个文件的版本历史，包括文件改名
$ git log --follow [file]
$ git whatchanged [file]
# 显示指定文件相关的每一次diff
$ git log -p [file]
# 显示指定文件是什么人在什么时间修改过
$ git blame [file]
# 显示暂存区和工作区的差异
$ git diff
# 显示暂存区和上一个commit的差异
$ git diff --cached []
# 显示工作区与当前分支最新commit之间的差异
```

```
$ git diff HEAD
# 显示两次提交之间的差异
$ git diff [first-branch]...
[second-branch]
# 显示某次提交的元数据和内容变化
$ git show [commit]
# 显示某次提交发生变化的文件
$ git show --name-only [commit]
# 显示当前分支的最近几次提交
$ git reflog
```

3.8、远程同步

```
# 下载远程仓库的所有变动
$ git fetch [remote]
# 显示所有远程仓库
$ git remote -v
# 显示某个远程仓库的信息
$ git remote show [remote]
# 增加一个新的远程仓库，并命名
$ git remote add [shortname] [url]
# 取回远程仓库的变化，并与本地分支合并
$ git pull [remote] [branch]
# 上传本地指定分支到远程仓库
$ git push [remote] [branch]
# 强行推送当前分支到远程仓库，即使有冲突
$ git push [remote] --force
# 推送所有分支到远程仓库
$ git push [remote] --all
```

3.9、撤销

```
# 恢复暂存区的指定文件到工作区
$ git checkout [file]
# 恢复某个commit的指定文件到工作区
$ git checkout [commit] [file]
# 恢复上一个commit的所有文件到工作区
$ git checkout .
# 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变
$ git reset [file]
# 重置暂存区与工作区，与上一次commit保持一致
$ git reset --hard
# 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变
$ git reset [commit]
# 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致
$ git reset --hard [commit]
# 重置当前HEAD为指定commit，但保持暂存区和工作区不变
$ git reset --keep [commit]
# 新建一个commit，用来撤销指定commit
# 后者的所有变化都将被前者抵消，并且应用到当前分支
```

```
$ git revert [commit]
```

4、使用GitHub

GitHub是一个面向开源及私有软件项目的托管平台，因为只支持git 作为唯一的版本库格式进行托管，故名gitHub。

1）、注册账户

2）、设置Git全局用户名和邮箱

```
#yourname GitHub注册用户名
$ git config --global user.name "yourname"
#your@email.com GitHub注册邮箱，也可以使用其它邮箱
$ git config --global user.email "your@email.com"
```

3）、生成密钥

```
$ ssh-keygen -t rsa -C "your@email.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Administrator/.ssh/id_rsa): 直接回车
```

然后系统会自动在.ssh文件夹下生成两个文件，id_rsa和id_rsa.pub，将id_rsa.pub全部的内容复制。

4、GitHub添加 SSH key

1. 打开https://github.com/，登陆github
2. 进入Settings -> SSH and GPG keys
3. 点击 New SSH key
4. 在key中将刚刚复制的粘贴进去
5. 点击 Add SSH key

5、推荐学习链接

中文官网：https://springcloud.cc/

中国社区：http://www.springcloud.cn/

SpringCloud GitHub：https://github.com/spring-cloud

SpringCloud教程：https://github.com/dyc87112/SpringCloud-Learning

SpringBoot基础教程：https://github.com/dyc87112/SpringBoot-Learning

微服务架构专题：http://blog.didispace.com/micro-serivces-arch/

Spring Cloud与Docker微服务架构实战：https://github.com/eacdy/spring-cloud-book