1.微服务架构的挑战：

⑴需要快速启动:你不能花一个月的时间来启动每个微服务，应该能够快速创建微服务。

⑵自动化:因为有许多更小的组件而不是一个整体，需要自动化所有的事情——构建、部署、监控等等。

⑶可见性:现在有许多更小的组件需要部署和维护。可能有100个或者1000个组件，应该能够自动监视和识别问题，需要在所有组件周围有良好的可见性。

⑷有界上下文:确定微服务的边界不是一件容易的事情。领域驱动设计(Domain Driven Design)的有界上下文是一个很好的起点。对领域的理解将在一段时间内不断发展，需要确保微服务边界的发展。

⑸配置管理:需要跨环境维护数百个组件的配置。需要一个配置管理解决方案。

⑹动态伸缩:只有当应用程序可以在云中方便地伸缩时，才能实现微服务的优势。(微服务实例数动态伸缩)

⑺容错能力:如果调用链底部的微服务失败，它会对所有其他微服务产生连锁反应。微服务应该在设计上具有容错能力。

⑻调试:当出现需要调查的问题时，可能需要查看跨不同组件的多个服务。集中式日志记录和仪表板对于简化调试问题非常重要。

⑼一致性:不可能有很多工具来解决同一个问题。虽然促进创新很重要，但是围绕用于实现/部署/监视微服务的语言、平台、技术和工具进行一些分散的治理也很重要。

2.微服务组件

⑴ Spring Cloud Config Server

Spring Cloud Config Server为分布式系统中的外部配置提供了基于HTTP资源的API。我们可以通过使用注释@EnableConfigServer来启用Spring Cloud配置服务器。

⑵Netflix Eureka Naming Server

Netflix Eureka服务器是一个发现服务器。它提供了与外部通信的REST接口。微服务出现后，将自己注册为发现客户端。Eureka服务器还有另一个软件模块，称为Eureka客户端。Eureka客户端与Eureka服务器交互以发现服务。Eureka客户端也会平衡客户端请求。

⑶Hystrix Server

Hystrix服务器充当容错健壮系统。它用于避免应用程序的完全失败。这是通过使用断路器机制实现的。如果应用程序运行没有任何问题，电路保持关闭状态。如果在应用程序中遇到错误，Hystrix服务器将打开电路。Hystrix服务器停止对呼叫服务的进一步请求。它提供了一个高度健壮的系统。

⑷Netflix Zuul API Gateway Server

Netflix Zuul服务器是一个网关服务器，所有的客户端请求都从这里通过。它充当客户机的统一接口。它还内置了一个负载均衡器，用于加载来自客户机的所有传入请求的平衡。

⑸Netflix Ribbon

Netflix Ribbon是客户端进程间通信(IPC)库。它提供了客户端平衡算法，它使用循环负载平衡：

①负载平衡

②容错

③多协议(HTTP，TCP，UDP)

④缓冲和批处理

⑹Zipkin Distributed Server

Zipkin是一个开源项目，为发送、接收和可视化跟踪提供了一种机制。您需要关注的一件事是端口号。

|  |  |
| --- | --- |
| Application | Port |
| Spring Cloud Config Server | 8888 |
| Netflix Eureka Naming Server | 8761 |
| Netflix Zuul API gateway Server | 8765 |
| Zipkin distributed Tracing Server | 9411 |

3.微服务监控

⑴监控是微服务的控制系统。由于微服务更复杂，更难以理解其性能并排除问题。考虑到软件交付的生动变化，需要监视服务。监视微服务有五个原则，如下所示：

①监视容器及其内部内容。

②提醒服务性能。

③监视弹性和多位置的服务。

④监控API。

⑤监视组织结构。

⑵微服务监控工具

Hystrix dashboard

Eureka admin dashboard

Spring boot admin dashboard

⑶微服务虚拟化

微服务虚拟化是模拟各种基于组件的应用程序(如基于云的应用程序、SOA和API驱动的体系结构)中特定组件的行为的方法。服务虚拟化还可以降低成本和节省时间。通过结合服务虚拟化，组织可以开发可以从不同位置和不同环境交付的应用程序。

4.实战：

⑴微服务A如何调用微服务B的REST接口?

①RestTemplate，如new RestTemplate().getForEntity(String url, Class<?> responseType, Map<String, ?> uriVariables)将请求url接口，参数为uriVariables，返回responseType类型数组。

②Feign代理

创建代理的接口，接口注解@FeignClient(name = serviceName， url = serviceIPAndPort)

@FeignClient(name = “serviceName”, url = “serviceIPAndPort”)

public interface BProxy{

@XXXMapping(“serviceUrl”)

public responseType methodName(parameter);

}

在再应用程序类中加上注解@EnableFeignClients(packageName)

接下来就可以通过bProxy.methodName(parameter)调用Rest接口。

⑵不管是RestTemplate还是FeignClient，接口地址(IP+Port)都被硬编码到代码中，这样就意味着，我们重启一个微服务B的实例的时候，A就无法调用B的新实例，也就无法实现客户端的负载均衡。

使用Ribbon正好解决了这个问题，可以在微服务A中配置多个微服务B的实例。

在代理接口中开启RibbonClient，即添加@RibbonClient(name = serviceName)

@FeignClient(name = “serviceName”)

@RibbonClient(name = “serviceName”)

public interface BProxy{

@XXXMapping(“serviceUrl”)

public responseType methodName(parameter);

}

在application.properties中配置微服务B的实例。

serviceName.ribbon.listOfServers=IP1:Port1,IP2+Port2,etc

⑶虽然使用Ribbon使得微服务的IP+Port从代码中剥离到配置文件中了，并且也能实现微服务的负载均衡。但是产生新的微服务的实例时，需要在调用它微服务中重新配置和启动。如何在产生新的微服务的实例的时候，不需要重新配置和重启调用它的微服务，这就是Eureka Naming Server的作用。

⑷Eurake Naming Server创建步骤

①SpringBoot的启动类加@EnableEurekaServer注解。

②将微服务与Eureka连接起来，即在微服务的配置文件中配置Eureka服务器的地址http://IP:Port/eureka。

③通过Eureka来路由Ribbon Request，去除application.properties中配置微服务的实例。