弹力设计总图

首先，我们的服务不能是单点。所以，我们需要在我们的架构中冗余服务，也就是说有多个服务的副本。这需要使用到的具体技术有：

- 负载均衡 + 服务健康检查 – 可以使用像 Nginx 或 HAProxy 这样的技术；
- 服务发现 + 动态路由 + 服务健康检查，比如 Consul 或 Zookeeper；
- 自动化运维，Kubernetes 服务调度、伸缩和故障迁移。

然后，我们需要隔离我们的业务，要隔离我们的服务我们就需要对服务进行解耦和拆分。这需要使用到以前的相关技术。

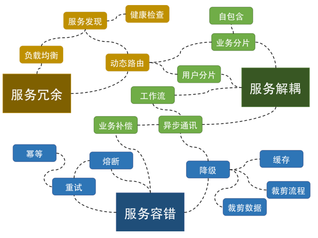
- bulkheads 模式：业务分片，用户分片，数据库拆分。
- 自包含系统：所谓自包含的系统是从单体到微服务的中间状态。其把一组密切相关的微服务给拆分出来。只需要做到没有外部依赖就行。
- 异步通讯：服务发现，事件驱动，消息队列，业务工作流。
- 自动化运维：需要一个服务调用链和性能监控的管理系统。

然后，接下来，我们就要进行和能让整个架构接受失败的相关处理设计。也就是所谓的容错设计。这会用到下面的这些技术。

- 错误方面：调用重试 + 熔断 + 服务的等性设计。
- 一致性方面：强一致性使用两阶段提交，最终一致性使用异步通讯方式。
- 流控方面：使用限流 + 降级技术。
- 自动化运维方面：网关流量调度，服务监控。

我不敢保证有上面这些技术可以解决所有的问题，但是，只要我们设计得当，绝大多数的问题应该是可以打得住的了。

下面我画一个图来表示一下。



在上面这个图上，我们可以看到，有三个大块的东西。

- 冗余服务，通过冗余服务的复本数可以消除单点故障。这需要服务发现、负载均衡、动态路由和健康检查四个功能或组件。
- 服务解耦，通过解耦可以做到把业务隔离开来，不让服务间受影响。这样就可以有更好的稳定性。在水平层面上，需要把业务或用户分片分区（业分做隔离，用户做多租户）。在垂直层面上，需要异步通讯机制。因为应用被分解成了一个一个的服务，所以在服务的编排和聚合上，需要有工作流（像 Spring 的 Stream 或 Akk 的 flow 或是 AWS 的 Simple Workflow）来把服务给串联起来。而一致性的问题又需要业务补偿机制来做反向交易。
- 服务容错，服务容错方面，需要有重试机制、重试机制会带来幂等操作。对于服务保护来说。熔断、限流，降级都是为了保护整个系统的稳定性，并在可用性和一致性方面在出错的情况下做一部分的妥协。

当然，除了这一切的架构设计外，你还需要一个成乡个自动运维的工具。否则，如果是人肉运维的话，那么在故障发生的时候，不能及时地做出运维决定，也就空有这些弹力设计了。比如：监控到服务性能不够了，就自动或半自动地开始进行限流或降级。

弹力设计开发和运维

对于运维工具来说，你至少需要两个系统：

- 一个是做 APM 这样的服务监控；
- 另一个是服务调度的系统，如：Docker + Kubernetes。

此外，如果你需要一个开发架构来让整个开发团队一在同一个标准下开发上面的这些东西，这里，Spring Cloud 就是不二之选了。

关于 Spring Cloud 和 Kubernetes，它们都是为了微服务而生，但它们没有什么可比性。因为，前者偏开发，后者偏运维。我们来看一下它们的差别。

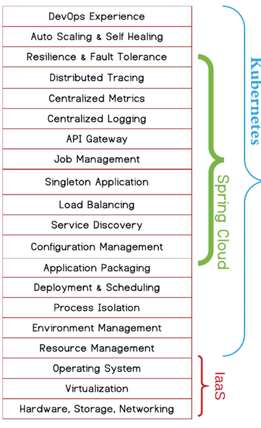
Microservices Concern	Spring Cloud & Netflix OSS	Kubernetes
Configuration Management	Config Server, Consul, Netflix Archaius	Kubernetes ConfigMap & Secrets
Service Discovery	Netflix Eureka, Hashicorp Consul	Kubernetes Service & Ingress Resources
Load Balancing	Netflix Ribbon	Kubernetes Service
API Gateway	Netflix Zuul	Kubernetes Service & Ingress Resources
Service Security	Spring Cloud Security	
Centralized Logging	ELK Stack (LogStash)	EFK Stack (Fluentd)
Centralized Metrics	Netflix Spectator & Atlas	Heapster, Prometheus, Grafana
Distributed Tracing	Spring Cloud Sleuth, Zipkin	OpenTracing, Zipkin
Resilience & Fault Tolerance	Netflix Hystrix, Turbine & Ribbon	Kubernetes Health Check & resource isolation
Auto Scaling & Self Healing		Kubernetes Health Check, Self Healing, Autoscaling
Packaging, Deployment & Scheduling	Spring Boot	Docker/Rkt, Kubernetes Scheduler & Deployment
Job Management	Spring Batch	Kubernetes Jobs & Scheduled Jobs
Singleton Application	Spring Cloud Cluster	Kubernetes Pods

（图片来自：Deploying Microservices: Spring Cloud vs Kubernetes）

从上表我们可以得知：

- Spring Cloud 有一套丰富且集成良好的 Java 库，作为应用栈的一部分解决所有运行时问题。因此，微服务本身可以通过库和运行时代理解决客户端服务发现、负载均衡、配置更新、统计跟踪等。工作模式就像单实例服务集群。（译者注：集群中 master 节点工作：当 master 挂掉后，slave 节点被选举顶替。）并且一般工作也是在 JVM 中被管理。
- Kubernetes 不是针对语言的，而是针对容器的，所以，它是以通用的方式为所有语言解决分布式计算问题。Kubernetes 提供了配置管理、服务发现、负载均衡、跟踪、统计、单实例、平台级和应用栈之外的调度工作。该应用不需要任何客户端逻辑的库或代理程序。可以用任何语言编写。

下图是微服务所需的关键技术。以及这些技术中在 Spring Cloud 和 Kubernetes 的涵盖面。



（图片来自：Deploying Microservices: Spring Cloud vs Kubernetes）

两个平台依靠相似的第三方工具，如 ELK 和 EFK stacks, tracing libraries 等。Hystrix 和 Spring Boot 等库，在两个环境中都表现良好。很多情况下，Spring Cloud 和 Kubernetes 可以形成互补，组建出更强大的解决方案（例如 KubeFlix 和 Spring Cloud Kubernetes）。

下图是在 Kubernetes 上使用 Spring Cloud 可以表现出来的整体特性。要做出一个可运维的分布式系统。除了在架构上的设计之外，还需要一整套的用来支撑分布式系统的管控系统。也就是所谓的运维系统。要做到这些，不是靠几个人几天就可以完成的。这需要我们根据自己的业务特点来规划相关的实施路径。

（图片来自：Deploying Microservices: Spring Cloud vs Kubernetes）

上面这张图中，对于所有的特性，都列举了一些相关的软件和一些设计的重点。其中红色的是运维层面的和 Spring Cloud 和 Kubernetes 不相关的，绿色的 Spring Cloud 提供的开发框架，需要作开发。蓝色的是 Kubernetes 相关的重要功能。

从今天看下来，微服务的最佳实践在未来有可能会成为 SpringCloud 和 Kubernetes 的天下了。这个让我们拭目以待。

我在本篇文章中总结了整个弹力设计、提供了一张总图、并介绍了开发运维的实践。希望对你有帮助。

也欢迎你分享一下你对弹力设计和弹力设计系列文章的感想。

文末给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。