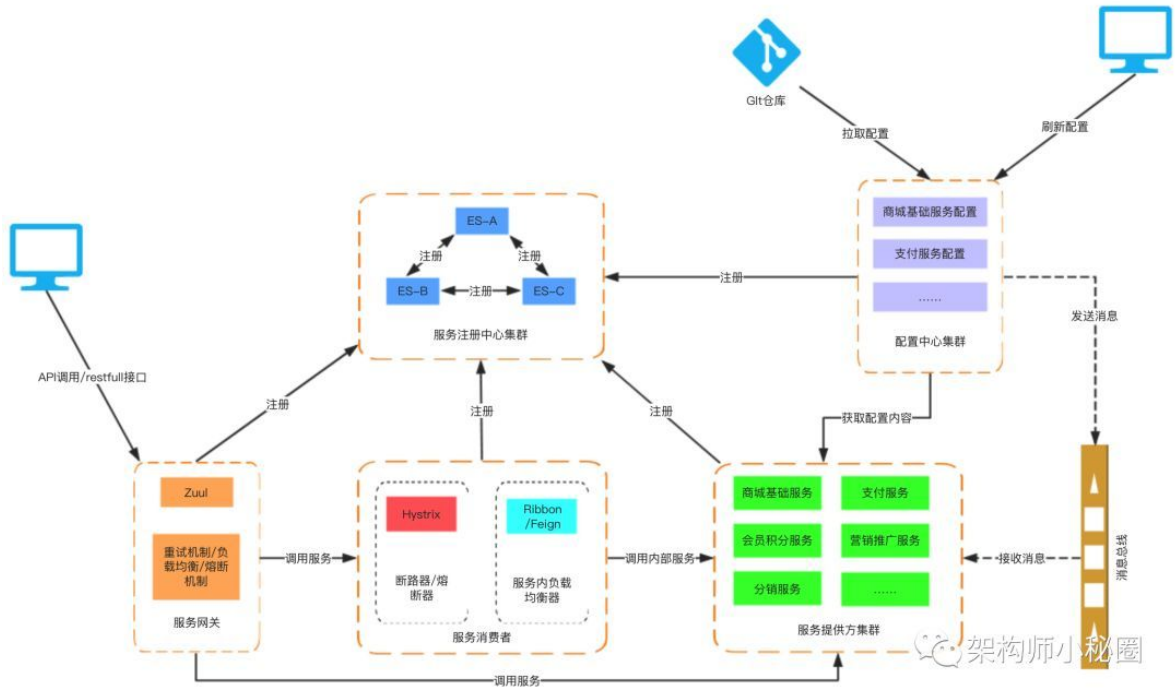
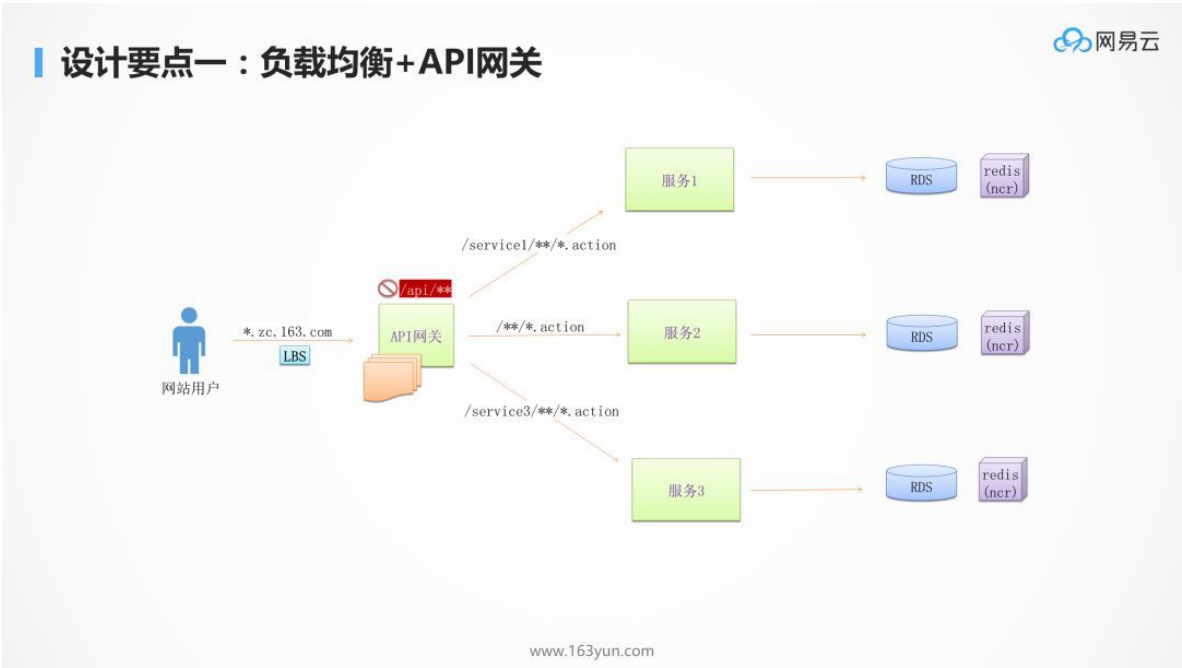
微服务化的十个设计要点

微服务生态

微服务有哪些要点呢？先看下图是 SpringCloud 的整个生态。



设计要点一：API 网关。



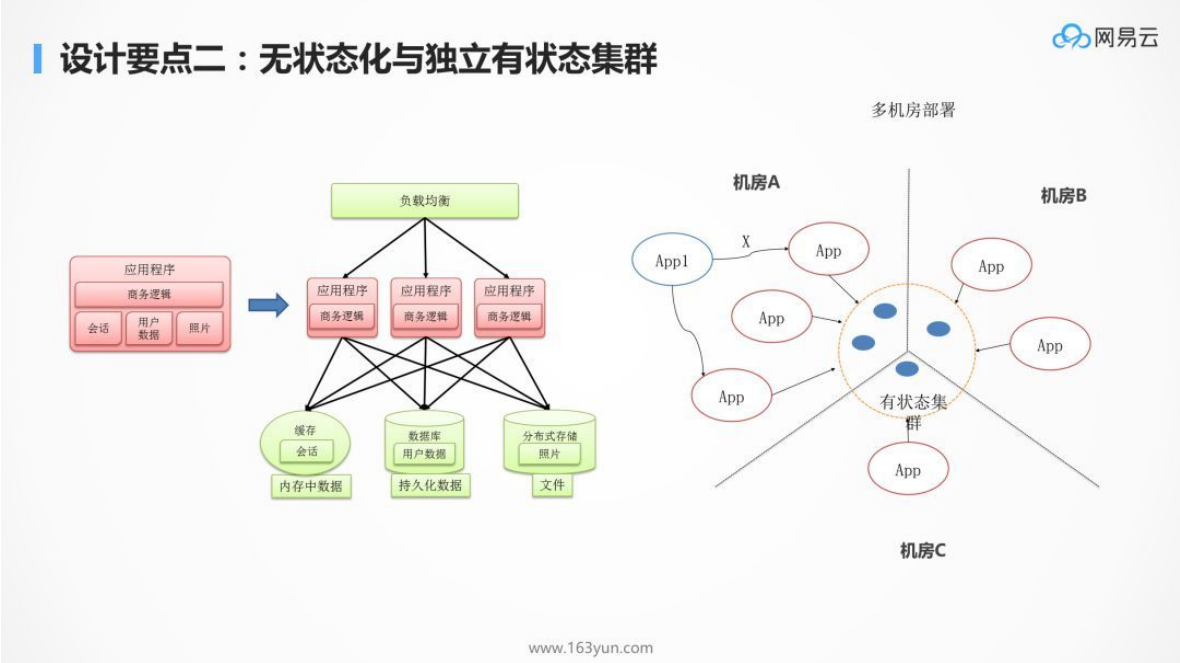
在实施微服务的过程中，不免要面临服务的聚合与拆分，当后端服务的拆分相对比较频繁的时候，作为手机 App 来讲，往往需要一个统一的入口，将不同的请求路由到不同的服务，无论后面如何拆分与聚合，对于手机端来讲都是透明的。

有了 API 网关以后，简单的数据聚合可以在网关层完成，这样就不用在手机 App 端完成，从而手机 App 耗电量较小，用户体验较好。

有了统一的 API 网关，还可以进行统一的认证和鉴权，尽管服务之间的相互调用比较复杂，接口也会比较多，API 网关往往只暴露必须的对外接口，并且对接口进行统一的认证和鉴权，使得内部的服务相互访问的时候，不用再进行认证和鉴权，效率会比较高。

有了统一的 API 网关，可以在这一层设定一定的策略，进行 A/B 测试，蓝绿发布，预发环境导流等等。API 网关往往是无状态的，可以横向扩展，从而不会成为性能瓶颈。

设计要点二：无状态化，区分有状态的和无状态的应用。



影响应用迁移和横向扩展的重要因素就是应用的状态，无状态服务，是要把这个状态往外移，将 Session 数据，文件数据，结构化数据保存在后端统一的存储中，从而应用仅仅包含商务逻辑。

状态是不可避免的，例如 ZooKeeper, DB，Cache 等，把这些所有有状态的东西收敛在一个非常集中的集群里面。

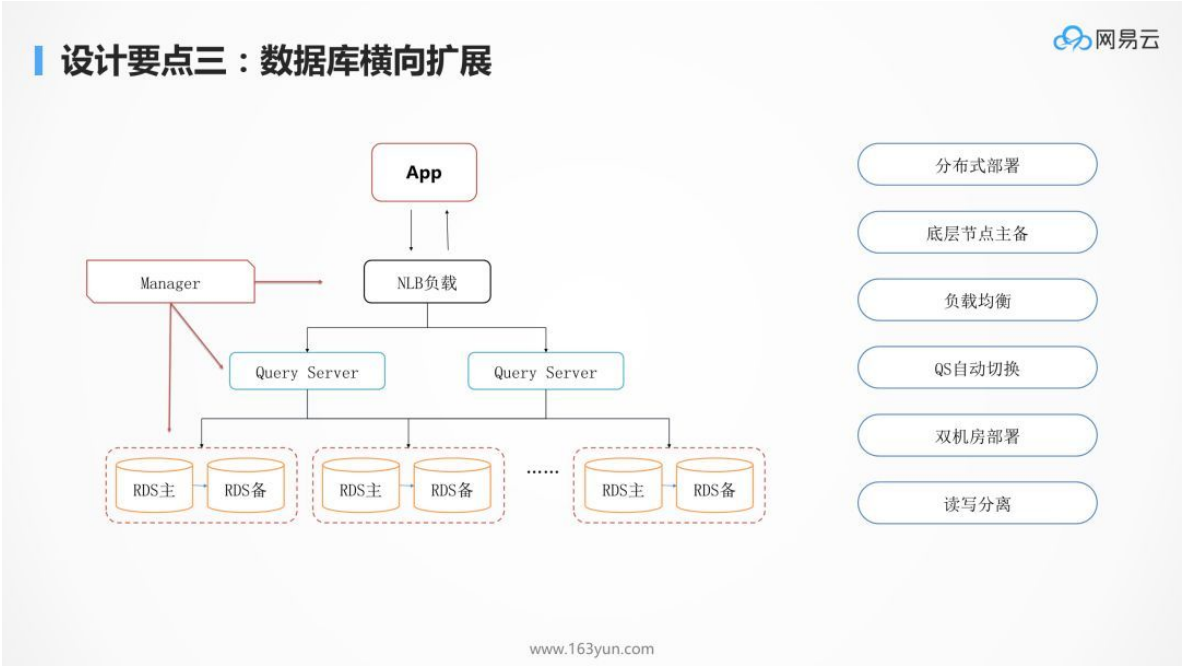
整个业务就分两部分，一个是无状态的部分，一个是有状态的部分。

无状态的部分能实现两点，一是跨机房随意地部署，也即迁移性，一是弹性伸缩，很容易地进行扩容。

有状态的部分，如 DB，Cache，ZooKeeper 有自己的高可用机制，要利用到他们自己高可用的机制来实现这个状态的集群。

虽说无状态化，但是当前处理的数据，还是会在内存里面的，当前的进程挂掉数据，肯定也是有一部分丢失的，为了实现这一点，服务要有重试的机制，接口要有幂等的机制，通过服务发现机制，重新调用一次后端服务的另一个实例就可以了。

设计要点三：数据库的横向扩展。



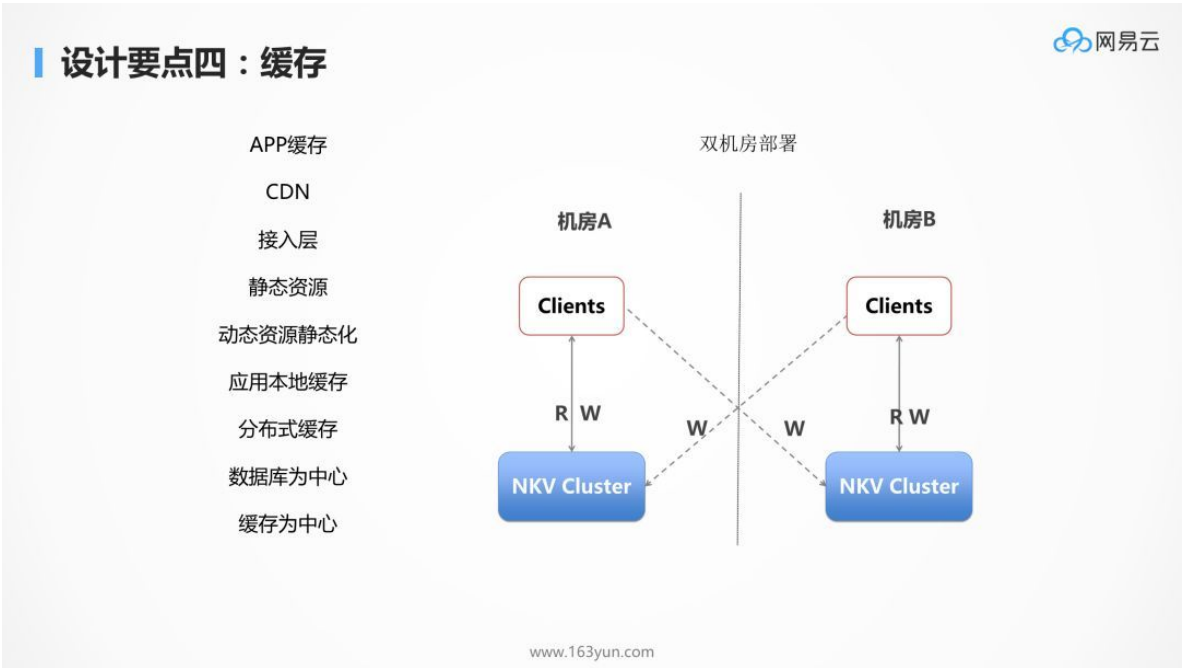
数据库是保存状态，是最重要的也是最容易出现瓶颈的。有了分布式数据库可以使数据库的性能可以随着节点增加线性地增加。

分布式数据库最最下面是 RDS，是主备的，通过 MySql 的内核开发能力，我们能够实现主备切换数据零丢失，所以数据落在这个 RDS 里面，是非常放心的，哪怕是挂了一个节点，切换完了以后，你的数据也是不会丢的。

再往上就是横向怎么承载大的吞吐量的问题，上面有一个负载均衡 NLB，用 LVS，HAProxy，Keepalived，下面接了一层 Query Server。Query Server 是可以根据监控数据进行横向扩展的，如果出现了故障，可以随时进行替换的修复，对于业务层是没有任何感知的。

另外一个就是双机房的部署，DDB 开发了一个数据运河 NDC 的组件，可以使得不同的 DDB 之间在不同的机房里面进行同步，这时候不但在一个数据中心里面是分布式的，在多个数据中心里面也会有一个类似双活的一个备份，高可用性有非常好的保证。

设计要点四：缓存



在高并发场景下缓存是非常重要的。要有层次的缓存，使得数据尽量靠近用户。数据越靠近用户能承载的并发量也越大，响应时间越短。

在手机客户端 App 上就应该有一层缓存，不是所有的数据都每时每刻从后端拿，而是只拿重要的，关键的，时常变化的数据。

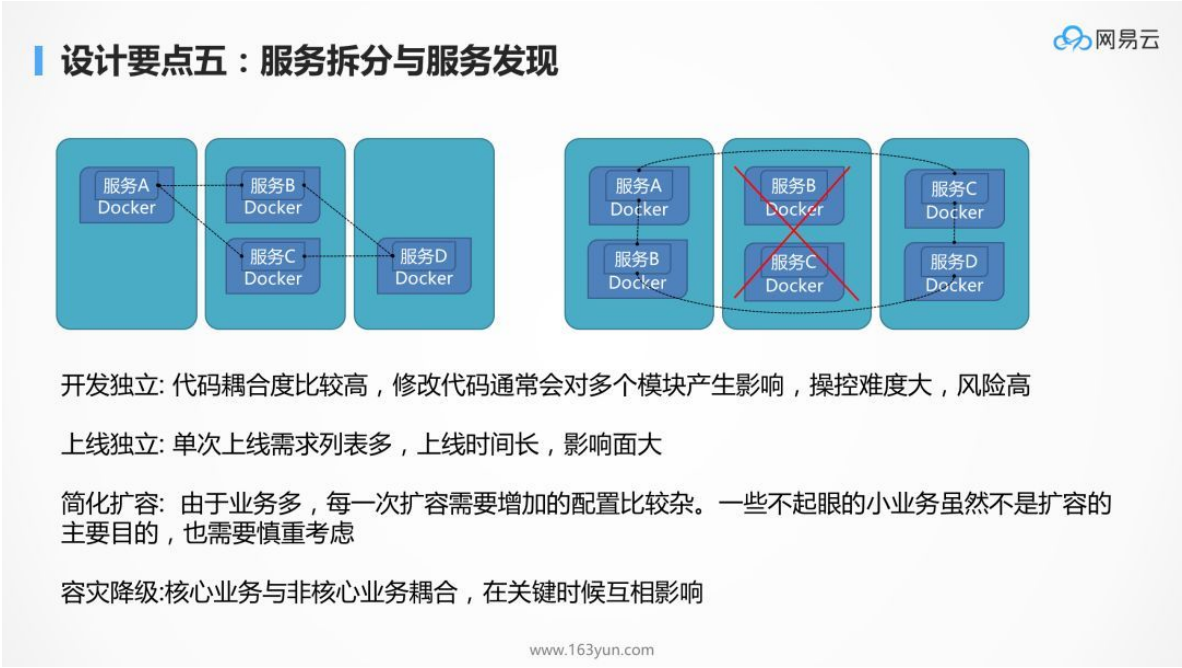
尤其对于静态数据，可以过一段时间去取一次，而且也没必要到数据中心去取，可以通过 CDN，将数据缓存在距离客户端最近的节点上，进行就近下载。

有时候 CDN 里面没有，还是要回到数据中心去下载，称为回源，在数据中心的最外层，我们称为接入层，可以设置一层缓存，将大部分的请求拦截，从而不会对后台的数据库造成压力。

如果是动态数据，还是需要访问应用，通过应用中的商务逻辑生成，或者去数据库读取，为了减轻数据库的压力，应用可以使用本地的缓存，也可以使用分布式缓存，如 Memcached 或者 Redis，使得大部分请求读取缓存即可，不必访问数据库。

当然动态数据还可以做一定的静态化，也即降级成静态数据，从而减少后端的压力。

设计要点五：服务拆分和服务发现



当系统扛不住，应用变化快的时候，往往要考虑将比较大的服务拆分为一系列小的服务。

这样第一个好处就是**开发比较独立**，当非常多的人在维护同一个代码仓库的时候，往往对代码的修改就会相互影响，常常会出现我没改什么测试就不通过了，而且代码提交的时候，经常会出现冲突，需要进行代码合并，大大降低了开发的效率。

另一个好处就是**上线独立**，物流模块对接了一家新的快递公司，需要连同下单一起上线，这是非常不合理的行为，我没改还要我重启，我没改还让我发布，我没改还要我开会，都是应该拆分的时机。

另外再就是**高并发时段的扩容**，往往只有最关键的下单和支付流程是核心，只要将关键的交易链路进行扩容即可，如果这时候附带很多其他的服务，扩容即是不经济的，也是很有风险的。

再就是**容灾和降级**，在大促的时候，可能需要牺牲一部分的边角功能，但是如果所有的代码耦合在一起，很难将边角的部分功能进行降级。

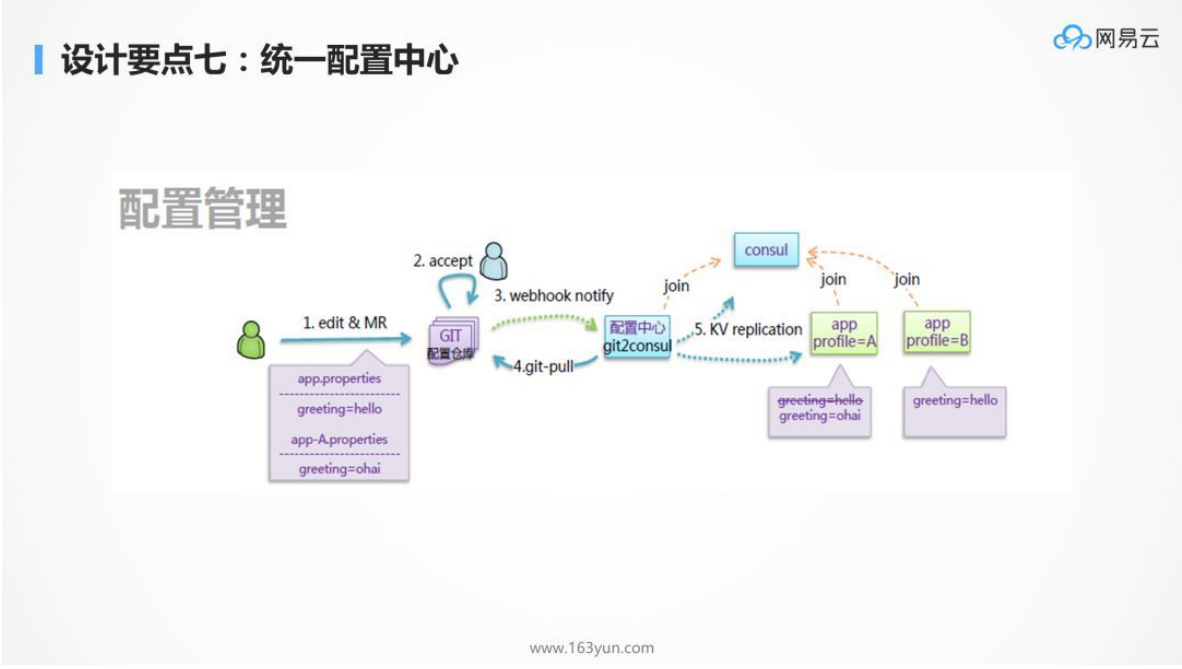
当然拆分完毕以后，应用之间的关系就更加复杂了，因而需要服务发现的机制，来管理应用相互的关系，实现自动的修复，自动的关联，自动的负载均衡，自动的容错切换。

设计要点六：服务编排与弹性伸缩

当服务拆分了，进程就会非常的多，因而需要服务编排来管理服务之间的依赖关系，以及将服务的部署代码化，也就是我们常说的基础设施即代码。这样对于服务的发布，更新，回滚，扩容，缩容，都可以通过修改编排文件来实现，从而增加了可追溯性，易管理性，和自动化的能力。

既然编排文件也可以用代码仓库进行管理，就可以实现一百个服务中，更新其中五个服务，只要修改编排文件中的五个服务的配置就可以，当编排文件提交的时候，代码仓库自动触发自动部署升级脚本，从而更新线上的环境，当发现新的环境有问题时，当然希望将这五个服务原子性地回滚，如果没有编排文件，需要人工记录这次升级了哪五个服务。有了编排文件，只要在代码仓库里面 revert，就回滚到上一个版本了。所有的操作在代码仓库里都是可以看到的。

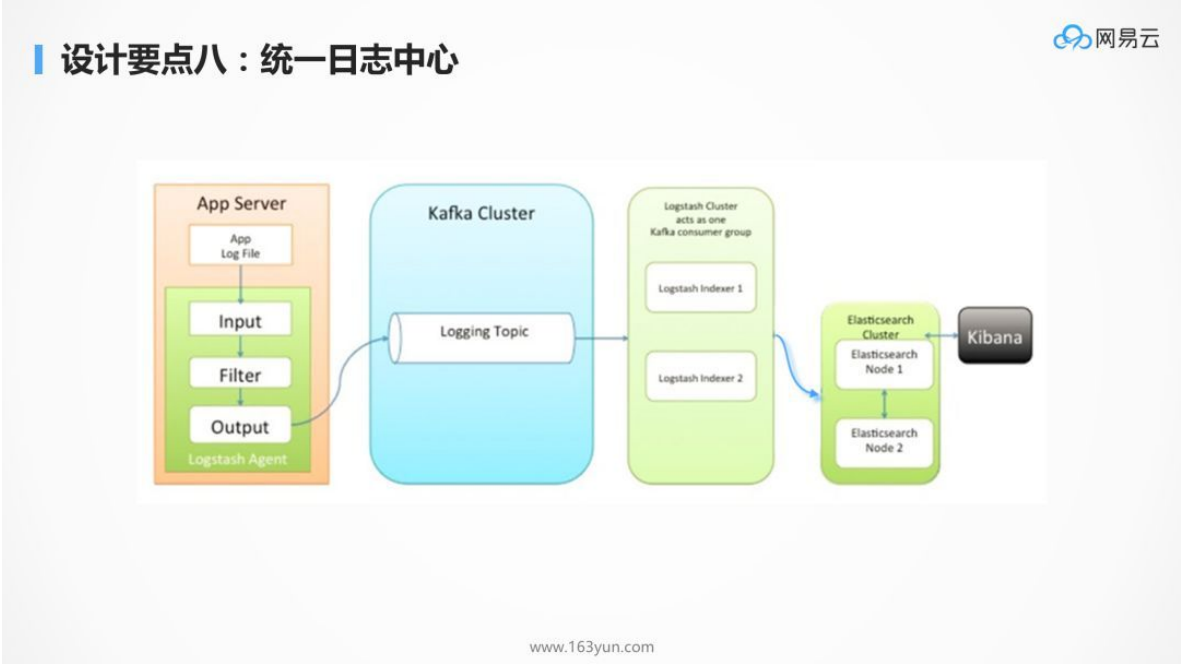
设计要点七：统一配置中心



服务拆分以后，服务的数量非常多，如果所有的配置都以配置文件的方式放在应用本地的话，非常难以管理，可以想象当有几百上千个进程中有一个配置出现了问题，是很难将它找出来的，因而需要有统一的配置中心，来管理所有的配置，进行统一的配置下发。

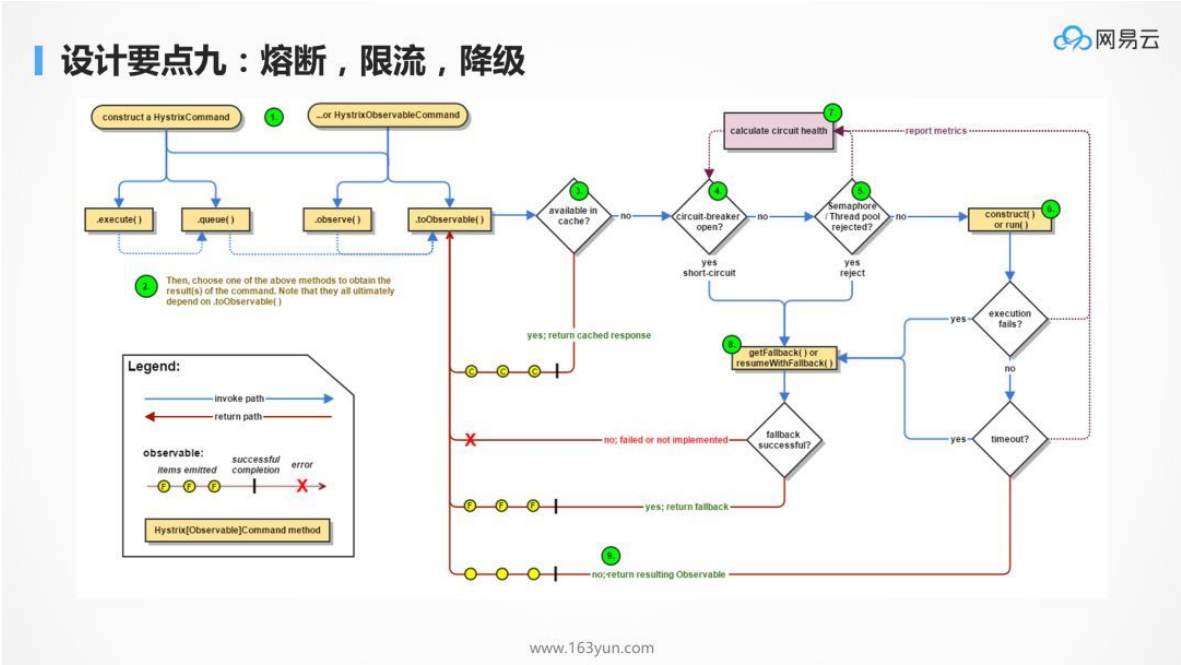
在微服务中，配置往往分为几类，一类是几乎不变的配置，这种配置可以直接打在容器镜像里面，第二类是启动时就会确定的配置，这种配置往往通过环境变量，在容器启动的时候传进去，第三类就是统一的配置，需要通过配置中心进行下发，例如在大促的情况下，有些功能需要降级，哪些功能可以降级，哪些功能不能降级，都可以在配置文件中统一配置。

设计要点八：统一的日志中心



同样是进程数目非常多的时候，很难对成千上百个容器，一个一个登录进去查看日志，所以需要统一的日志中心来收集日志，为了使收集到的日志容易分析，对于日志的规范，需要有一定的要求，当所有的服务都遵守统一的日志规范的时候，在日志中心就可以对一个交易流程进行统一的追溯。例如在最后的日志搜索引擎中，搜索交易号，就能够看到在哪个过程出现了错误或者异常。

设计要点九：熔断，限流，降级



服务要有熔断，限流，降级的能力，当一个服务调用另一个服务，出现超时的时候，应及时返回，而非阻塞在那个地方，从而影响其他用户的交易，可以返回默认的托底数据。

当一个服务发现被调用的服务，因为过于繁忙，线程池满，连接池满，或者总是出错，则应该及时熔断，防止因为下一个服务的错误或繁忙，导致本服务的不正常，从而逐渐往前传导，导致整个应用的雪崩。

当发现整个系统的确负载过高的时候，可以选择降级某些功能或某些调用，保证最重要的交易流程的通过，以及最重要的资源全部用于保证最核心的流程。

还有一种手段就是限流，当既设置了熔断策略，又设置了降级策略，通过全链路的压力测试，应该能够知道整个系统的支撑能力，因而就需要制定限流策略，保证系统在测试过的支撑能力范围内进行服务，超出支撑能力范围的，可拒绝服务。当你下单的时候，系统弹出对话框说“系统忙，请重试”，并不代表系统挂了，而是说明系统是正常工作的，只不过限流策略起到了作用。

设计要点十：全方位的监控

当系统非常复杂的时候，要有统一的监控，主要有两个方面，一个是是否健康，一个是性能瓶颈在哪里。当系统出现异常的时候，监控系统可以配合告警系统，及时地发现，通知，干预，从而保障系统的顺利运行。

当压力测试的时候，往往会遭遇瓶颈，也需要有全方位的监控来找出瓶颈点，同时能够保留现场，从而可以追溯和分析，进行全方位的优化。