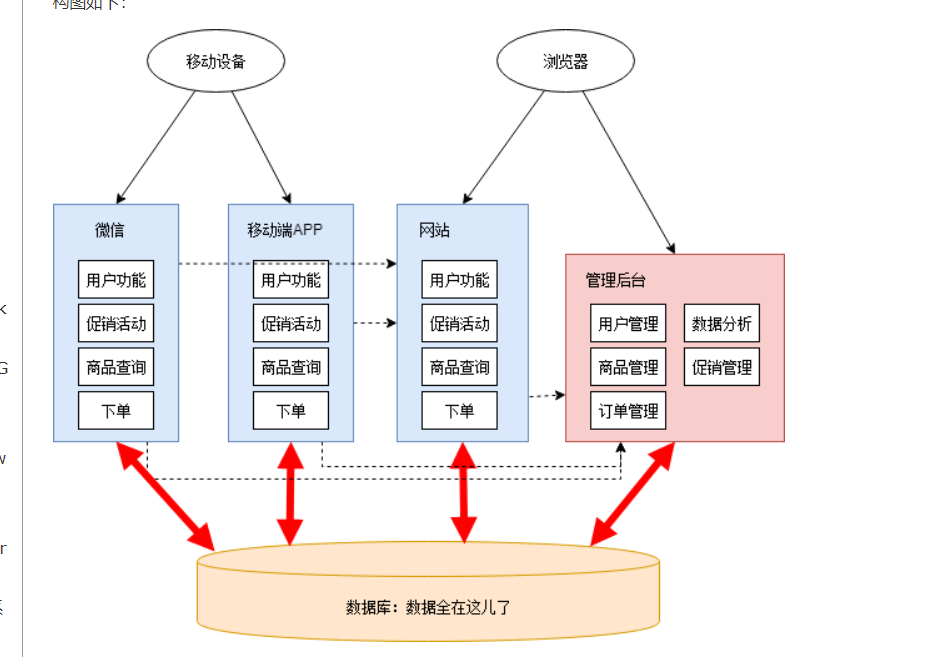
微服务架构的全局图景

以网上超市为例：



这一阶段的不合理：

1. 网站和移动端有很多相同业务逻辑的重复代码。
2. 数据有时候通过数据库共享，有时候通过接口调用传输。接口调用关系杂乱。
3. 单个应用为了给其他应用提供接口，会包含许多本来不属于它的逻辑，应用边界模糊，功能归属混乱。
4. 管理后台在一开始的设计中保障级别较低。加入数据分析和促销管理功能后出现性能瓶颈，影响其他应用。
5. 数据库多个表结构被多个应用依赖，无法重构和优化。
6. 所有应用都在一个数据库上操作，数据库出现性能瓶颈。特别是数据分析运行的时候，数据库性能急剧下降。
7. 开发测试部署维护愈发困难。即使改动一个小功能，整个应用一起发布。有时候发布会不小心带上了一些未经测试的代码，或者修改了一个功能后，另一个意想不到的地方出错了。为了减轻发布可能产生的问题的影响和线上业务停顿的影响，所有应用都要在凌晨三四点执行发布。发布后为了验证应用正常运行，还得盯到第二天白天的用户高峰期……
8. 团队出现推诿扯皮现象。关于一些公用的功能应该建设在哪个应用上的问题常常要争论很久，最后要么干脆各做各的，或者随便放个地方但是都不维护。

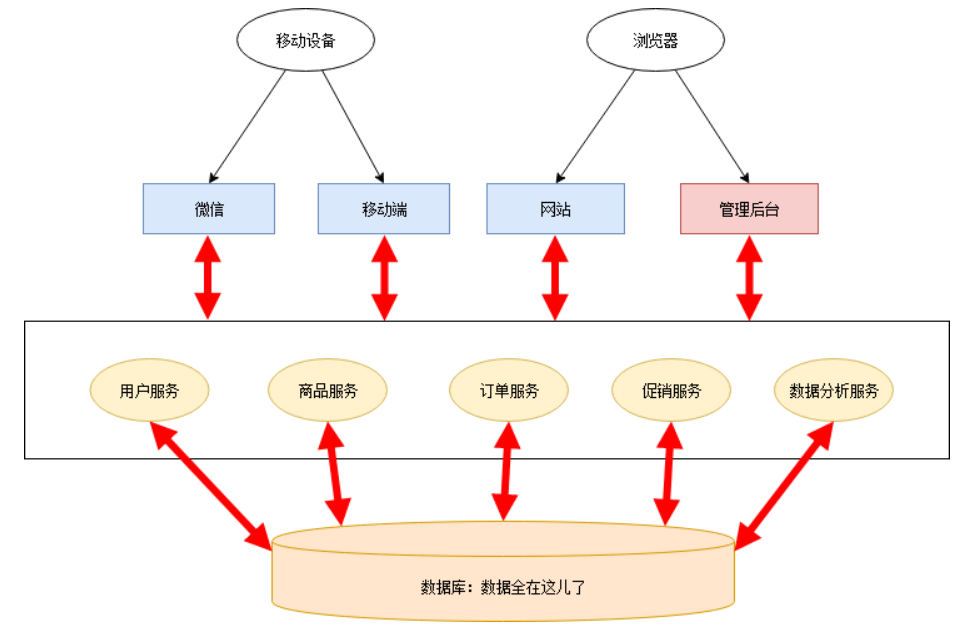
**紧迫且繁重的任务容易使人陷入局部、短浅的思维方式，从而做出妥协式的决策**

**在编程的世界中，最重要的是抽象能力。微服务改造的过程实际上是个抽象的过程。**

**整理出网上超市的业务逻辑，抽象出公用的业务能力，做成公共服务：**

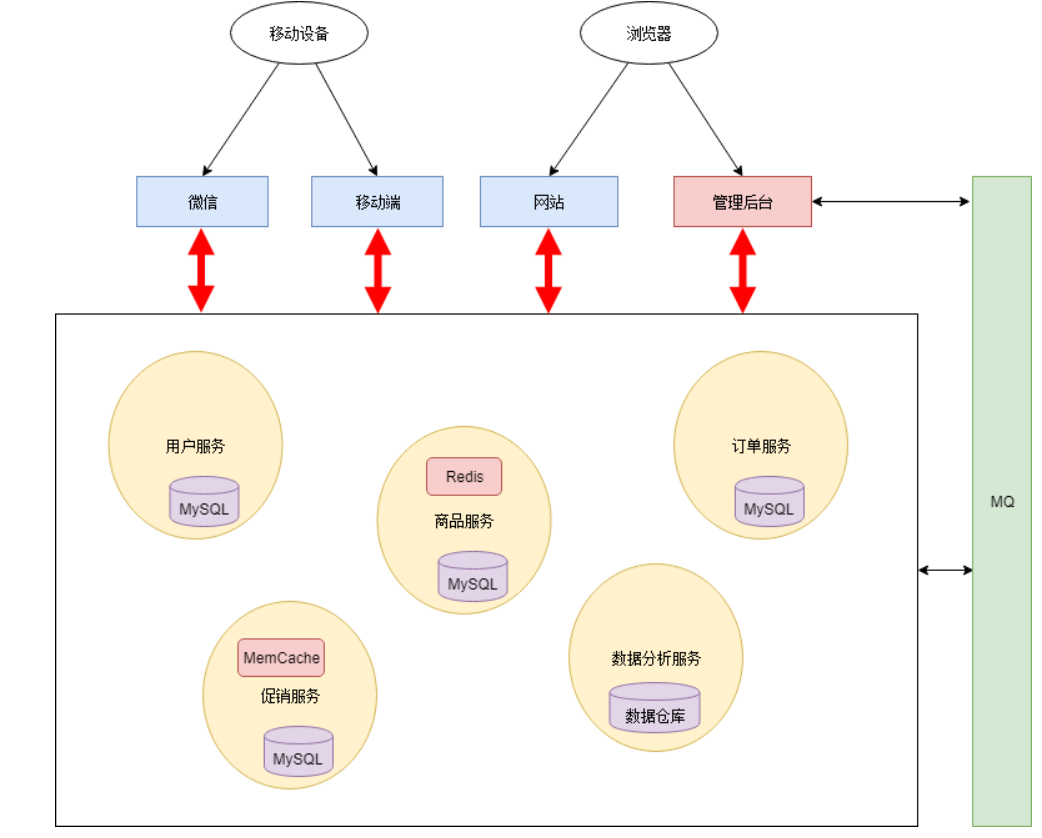
**用户服务、商品服务、促销服务、订单服务、数据分析服务。**

**各个应用后台只需从这些服务获取所需的数据，从而删除了大量冗余的代码，剩下个轻薄的控制层和前端。这一时期的架构：**

****

**但是数据库依然是共用的，关于数据库的缺点依然存在。**

**下一步：把所有持久化层相互隔离，有各个服务自己负责。另外，为了提高系统的实时性，加入了消息队列机制。**

****

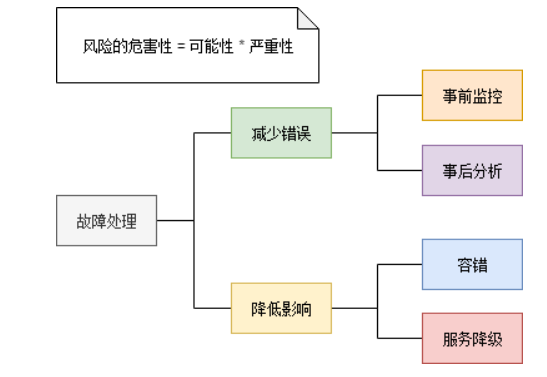
完全拆分后各个服务可以采用异构的技术。比如数据分析服务可以使用数据仓库作为持久化层，以便于高效的做一些统计计算；商品服务和促销服务访问频率比较大，因此加入了缓存机制等。

*数据库拆分也有一些问题和挑战：比如说跨库级联的需求，通过服务查询数据颗粒度的粗细问题等。但是这些问题可以通过合理的设计来解决。总体来说，数据库拆分是一个利大于弊的。*

以往单体应用，排查问题通常是看一下日志，研究错误信息和调用堆栈。而微服务架构整个应用分散成多个服务，定位故障点非常困难，并且一个服务故障可能会产生雪崩效应，导致整个系统故障。服务之间彼此调用，该怎么？？？微服务好像是创建maven工程，在服务中引入另一个服务，即可使用另一个服务的API。

微服务架构虽然解决了旧问题，也引入了新的问题:

1. 微服务架构整个应用分散成多个服务，定位故障点非常困难。、
2. 稳定性下降。服务数量变多导致其中一个服务出现故障的概率增大，并且一个服务故障可能导致整个系统挂掉。事实上，在大访问量的生产场景下，故障总是会出现的。
3. 服务数量非常多，部署、管理的工作量很大。
4. 开发方面：如何保证各个服务在持续开发的情况下仍然保持协同合作。
5. 测试方面：服务拆分后，几乎所有功能都会涉及多个服务。原本单个程序的测试变为服务间调用的测试。测试变得更加复杂。



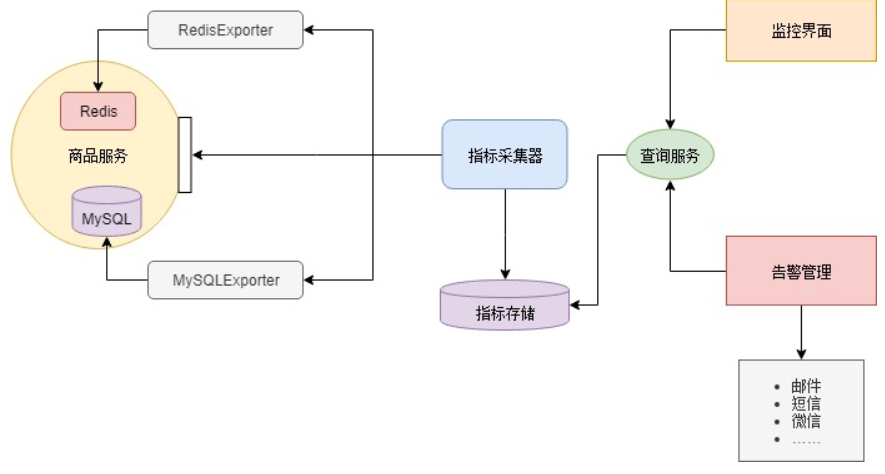
**减少错误方面：**

* 监控——发现故障的征兆

由于各个组件所需监控的指标不同，如Redis缓存一般监控占用内存值、网络流量，数据库监控连接数、磁盘空间，业务服务监控并发数、响应延迟、错误率等。所以是让各个组件提供报告自己当前状态的接口（metrics接口），这个接口输出的数据格式应该是一致的。然后部署一个指标采集器组件，定时从这些接口获取并保持组件状态，同时提供查询服务。最后还需要一个UI，从指标采集器查询各项指标，绘制监控界面或者根据阈值发出告警。

大多组件不需要自己动手开发，网上有开源组件。RedisExporter和MySQLExporter，这两个组件分别提供了Redis缓存和MySQL数据库的指标接口

最后，微服务监控系统：



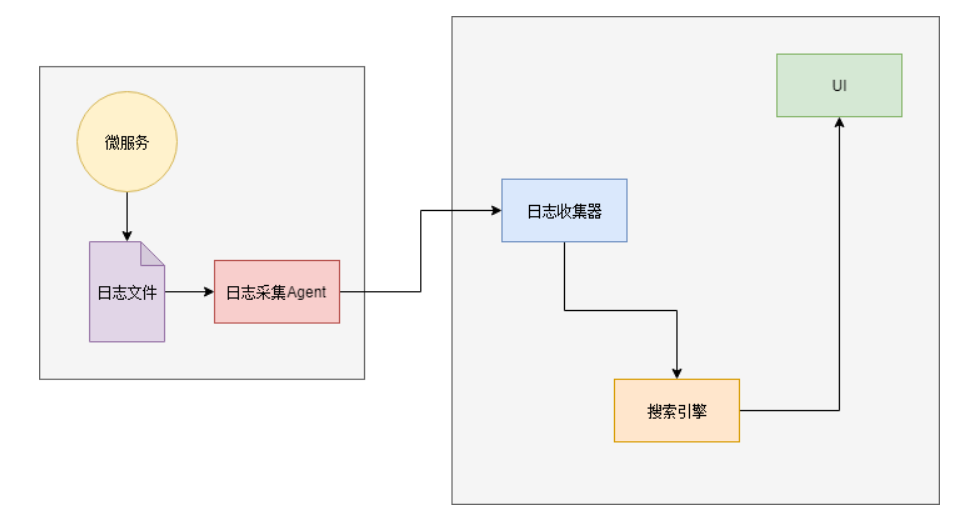
* 定位问题——链路跟踪

在微服务架构下，一个用户的请求往往涉及多个内部服务调用。为了方便定位问题，需要能够记录每个用户请求时，微服务内部产生了多少服务调用，及其调用关系。这个叫做链路跟踪。

链路跟踪只能定位到哪个服务出现问题，不能提供具体的错误信息。查找具体的错误信息的能力则需要由日志分析组件来提供。

* 分析问题——日志分析

在应用规模变大时，我们需要一个日志的“搜索引擎”。以便于能准确的找到想要的日志。另外，数据源一侧还需要收集日志的组件和展示结果的UI组件：

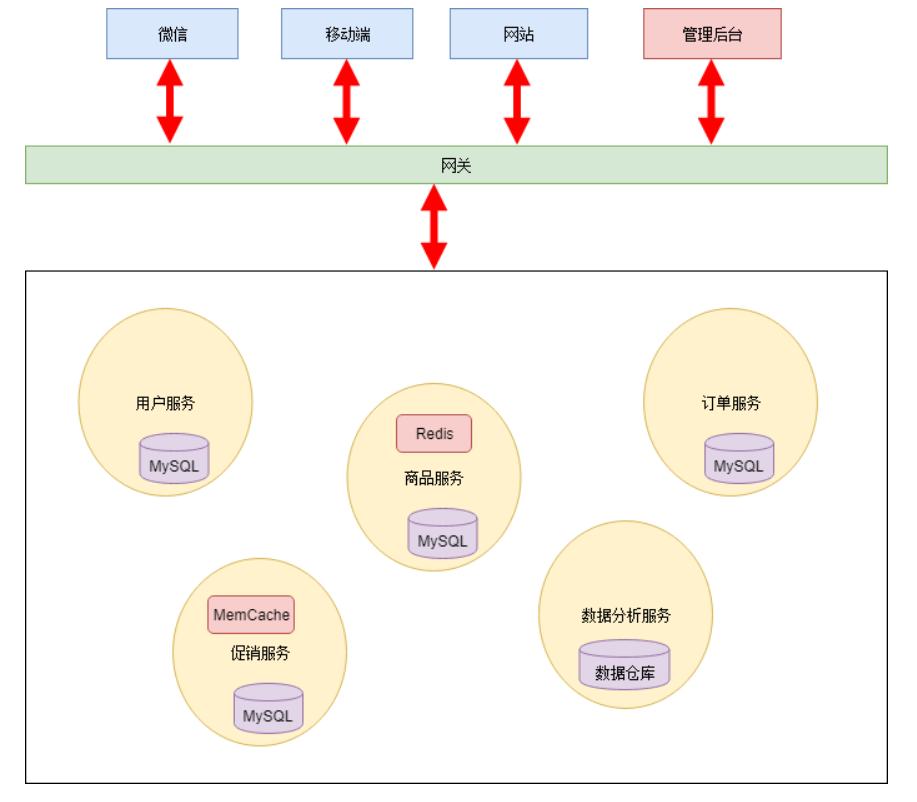


* 网关——权限控制，服务治理

拆分成微服务后，出现大量的服务和接口，使得整个调用关系乱糟糟的。开发过程，某个数据应该调用哪个服务，或者调用了不该调用的服务，本来一个只读的功能结果修改了数据。。。。。。

在调用者和被调用者之间加一层网关，每次调用时进行权限校验。另外，网关也可以作为一个提供接口文档的平台。

最粗粒度的方案是整个微服务一个网关，微服务外部通过网关访问微服务，微服务内部则直接调用；最细粒度则是所有调用，不管是微服务内部调用或者来自外部的调用，都必须通过网关。折中的方案是按照业务领域将微服务分成几个区，区内直接调用，区间通过网关调用。



粗粒度的方案

**降低故障影响方面：**

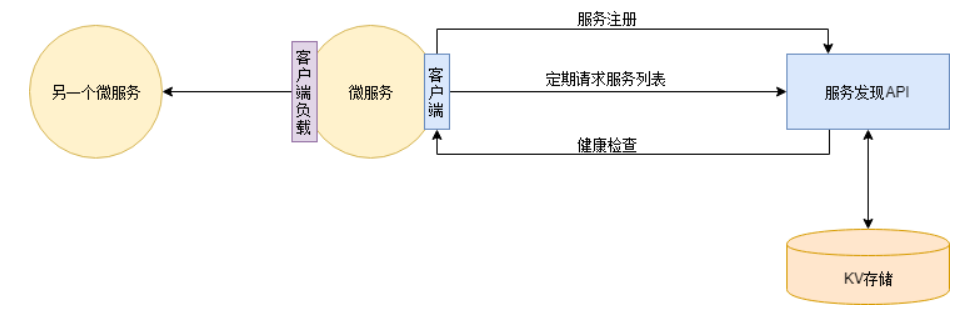
* 服务注册于发现——动态扩容

最粗暴的（也是最常用的）故障处理策略就是冗余。一般来说，一个服务都会部署多个实例，一来分担压力提高性能，二来即使一个实例挂了其他实例还能响应。

到底冗余多少个合适？

新增实例作为冗余需要：1、部署新实例 2、将新的实例注册到负载均衡或DNS上。

注册到负载均衡或DNS的操作可以通过服务自动注册与发现。首先，需要部署一个服务发现服务，它提供所有已注册服务的地址信息的服务。DNS也算是一种服务发现服务，然后各个应用服务在启动时将自己注册到服务发现服务上。并且应用服务启动后会实时（定期）从服务发现服务同步各个应用的地址列表到本地。服务发现服务也会定期检查应用服务的健康状态，去掉不健康的实例地址。这样新增实例只需要部署新实例，实例下线时直接关停服务即可，服务发现服务会自动检查实例的增减。

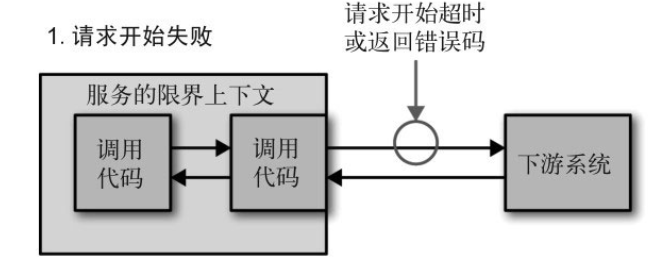


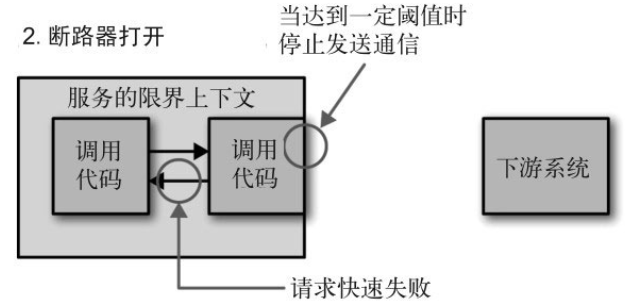
服务发现还会跟客户端负载均衡配合使用。由于应用服务已经同步服务地址列表在本地了，所以访问微服务时，可以自己决定负载策略。甚至可以在服务注册时加入一些元数据（服务版本等信息），客户端负载则根据这些元数据进行流量控制，实现A/B测试、蓝绿发布等功能。

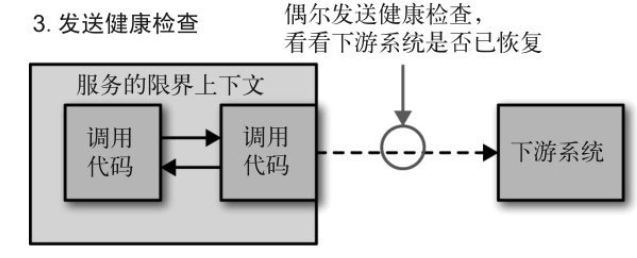
服务发现有很多组件可以选择，比如说Zookeeper （服务注册和发现）、Eureka、Consul、Etcd等，也可以基于Redis……

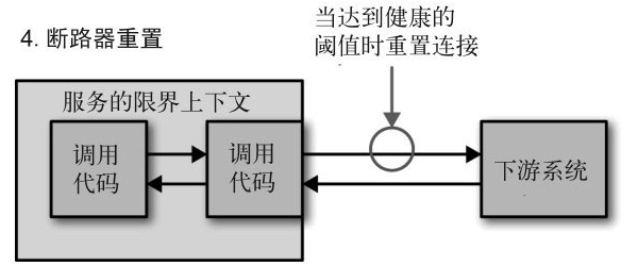
* 熔断、服务降级和限流：

1. 熔断: 一个服务因为各种原因停止响应，调用方通常会等待一段时间，然后超时或者收到错误返回。如果调用链路比较长，可能会导致请求堆积，整条链路占用大量资源一直在等待下游响应。所以当多次访问一个服务失败时，应熔断，标记该服务已停止工作，直接返回错误。直至该服务恢复正常后再重新建立连接。









1. 服务降级

当下级服务停止工作后，如果该服务并非核心业务，则上游应该降级，以保证核心业务不中断。如网上超市下单界面有一个推荐商品凑单的功能，当推荐模块挂了后，下单功能不能一起挂掉，只需要暂时关闭推荐功能即可（降级）。

1. 限流

一个服务挂掉后，上游服务或者用户一般会习惯性地重试访问。一旦服务恢复正常，很可能因为瞬间网络流量过大又立刻挂掉。限流策略有很多，最简单的比如当单位时间内请求数过多时，丢弃多余的请求。另外，也可以考虑分区限流。仅拒绝来自产生大量请求的服务的请求。例如商品服务和订单服务都需要访问促销服务，商品服务由于代码问题发起了大量请求，促销服务则只限制来自商品服务的请求，来自订单服务的请求则正常响应。

* 测试

微服务架构下，测试分为三个层次：

端到端测试：覆盖整个系统，一般在用户界面进行测试

服务测试：针对服务接口进行测试

单元测试：针对代码单元进行测试

微服务框架：

指标接口、链路跟踪注入、日志引流、服务注册发现、路由规则等组件以及熔断、限流等功能都需要在应用服务上添加一些对接代码。如果让每个应用服务自己实现是非常耗时耗力的。基于DRY（不要重复自己）的原则，将与各个组件对接的代码和另外一些公共代码抽离到框架中，所有的应用服务都统一使用框架进行开发。

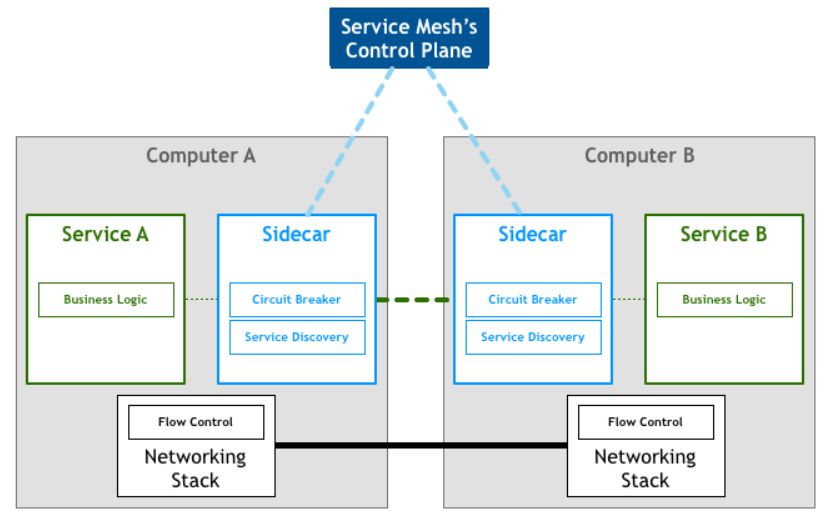
使用微服务框架可以实现很多自定义的功能。甚至可以将程序调用堆栈信息注入到链路跟踪，实现代码级别的链路跟踪。或者输出线程池、连接池的状态信息，实时监控服务底层状态。

但是框架更新成本很高。使用统一微服务框架需要完善的版本管理方法和开发管理规范。

Another way——Service Mesh

另一个抽象公共代码的方法是直接将这些代码抽象到一个反向代理组件。每个服务都额外部署这个代理组件，所有出站入站的流量都通过该组件进行处理和转发。这个组件被称为Sidecar。*Sidecar不会产生额外网络成本。Sidecar会和微服务节点部署在同一台主机上并且共用相同的虚拟网卡。所以sidecar和微服务节点的通信实际上是通过内存拷贝实现的。*

SideCar只负责网络通信，还需要组件统一管理所有的sidecar的配置。所以，在Service Mesh中，负责网络通信的部分叫数据平面，负责配置管理的部分叫控制平面，数据平面和控制平面构成了Service Mesh 的基本框架。



Service Mesh相比于微服务框架的优点在于它不侵入代码，升级和维护会更方便。但是性能问题令人诟病，即使回环网络不会产生实际的网络请求，但仍然有内存拷贝的额外成本。另外有一些集中式的流量处理也会影响性能。

**消息中间件的相关知识**

1··. 消息队列已经逐渐成为企业IT系统内部通信的核心手段。它具有低耦合、可靠投递、广播、流量控制、最终一致性等一系列功能，成为异步RPC（远程过程调用）的主要手段之一。

2. 消息中间件的组成：

2.1 Broker

消息服务器，作为server提供消息核心服务

2.2 Producer

消息生产者，业务的发起方，负责生产消息传输给broker，

2.3 Consumer

消息消费者，业务的处理方，负责从broker获取消息并进行业务逻辑处理

2.4 Topic

主题，发布订阅模式下的消息统一汇集地，不同生产者向topic发送消息,由MQ服务器分发到不同的订阅者，实现消息的广播

2.5 Queue

队列，PTP(点对点)模式下，特定生产者向特定queue发送消息，消费者订阅特定的queue完成指定消息的接收

2.6 Message

消息体，根据不同通信协议定义的固定格式进行编码的数据包，来封装业务数据，实现消息的传输

3、消息中间件模式分类：点对点 发布/订阅

4、消息中间件的优势：4.1 系统解耦

交互系统之间没有直接的调用关系，只是通过消息传输，故系统侵入性不强，耦合度低。

4.2 提高系统响应时间

例如原来的一套逻辑，完成支付可能涉及先修改订单状态、计算会员积分、通知物流配送几个逻辑才能完成；通过MQ架构设计，就可将紧急重要（需要立刻响应）的业务放到该调用方法中，响应要求不高的使用消息队列，放到MQ队列中，供消费者处理。

4.3 为大数据处理架构提供服务

通过消息作为整合，大数据的背景下，消息队列还与实时处理架构整合，为数据处理提供性能支持。

4.4 Java消息服务——JMS

Java消息服务（Java Message Service，JMS）应用程序接口是一个Java平台中关于面向消息中间件（MOM）的API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

JMS中的P2P和Pub/Sub消息模式：点对点（point to point， queue）与发布订阅（publish/subscribe，topic）最初是由JMS定义的。这两种模式主要区别或解决的问题就是发送到队列的消息能否重复消费(多订阅)。

Serverless无服务：

开发者再也不用过多考虑服务器的问题，但是并不代表完全去除服务器，而是我们依靠第三方资源服务器后端，比如使用AWS计算服务执行代码。Serverless架构分为BaaS（后端即服务）和FaaS（函数即服务）。

传统的架构模式是c/s模式的，在Serverless架构中，应用业务逻辑是基于FaaS架构形成多个相互独立的功能组件，并且以API服务的形式向外提供服务，，在FaaS中，后端的应用被拆分成为一个个函数，我们只需要编写完成函数后部署到serverless服务即可。后续我们也不用关心任何服务器的操作。那么整个流程就只需要我们一个前端工程师的角色来完成所有的开发工作，那么沟通成本降低了。

Serverless应用场景：构建比较简单的应用。

特点：细粒度的计算资源的分配，不需要预分配资源，具备真正意义上的高度扩容和弹性，按需使用，按需计费。

因此以下应用可能使用serverless架构：

1. 静态网站的管理
2. 替代WordPress
3. 个人媒体服务器
4. 物联网IoT或家庭自动框架或项目