微服务
微服务定义与优缺点
RPC协议(远程过程调用协议)
负载均衡 + API Gateway
服务发现与注册
服务注册:
服务发现:
服务通信(进程间的通信)
服务通信类型:
三种IPC技术:
微服务部署策略
Monolithic 重构
策略一:不要大规模重写代码
重构方法:
重构ps:

微服务

微服务定义与优缺点

定义:

每个服务运行在其独立的进程中，服务于服务间采用轻量级的通信机制互相沟通（通常是基于 HTTP 的 RESTful API）

简单来说就是将一个单体服务拆分成多个微型服务，从而实现对单体服务的解耦。每个微型服务之间进行通信。

微服务的特点：

- 单一职责
- 自治
- 逻辑清晰
- 简化部署
- 可扩展
- ...

缺点:

- **复杂度高：**

  微服务间通过REST、RPC等形式交互，相对于Monolithic模式下的API形式，需要考虑被调用方故障、过载、消息丢失等各种异常情况，代码逻辑更加复杂。

- **运维复杂:**

在采用微服务架构时，系统由多个独立运行的微服务构成，需要一个设计良好的监控系统对各个微服务的运行状态进行监控。运维人员需要对系统有细致的了解才对够更好的运维系统。

- **影响性能**

  相对于Monolithic架构，微服务的间通过REST、RPC等形式进行交互，通信的时延会受到较大的影响。

RPC协议(远程过程调用协议)

定义:

RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。

GRPC:

- Google开源的一款框架，支持HTTP/2，支持多语言调用；
- 通过Proto3进行编译数据，从而实现各个语言的互通；
- 支持四种调用模式，引入了流的概念；
- 性能方面有点令人担忧；

Thrift:

Thrift是一种接口描述语言和二进制通讯协议，它被用来定义和创建跨语言的服务。它被当作一个远程过程调用（RPC）框架来使用，是由Facebook为“大规模跨语言服务开发”而开发的。它通过一个代码生成引擎联合了一个软件栈，来创建不同程度的、无缝的跨平台高效服务

负载均衡 + API Gateway

- **Grpc-Gateway:**

  go 的一套网关，可以直接将现有的微服务的ip按照rest格式对外开放。当然这个只能选择GRPC。

- **自己轮:**

  利用nginx做负载均衡和反向代理，并且继续使用tornado搭建web api 的网关。

服务发现与注册

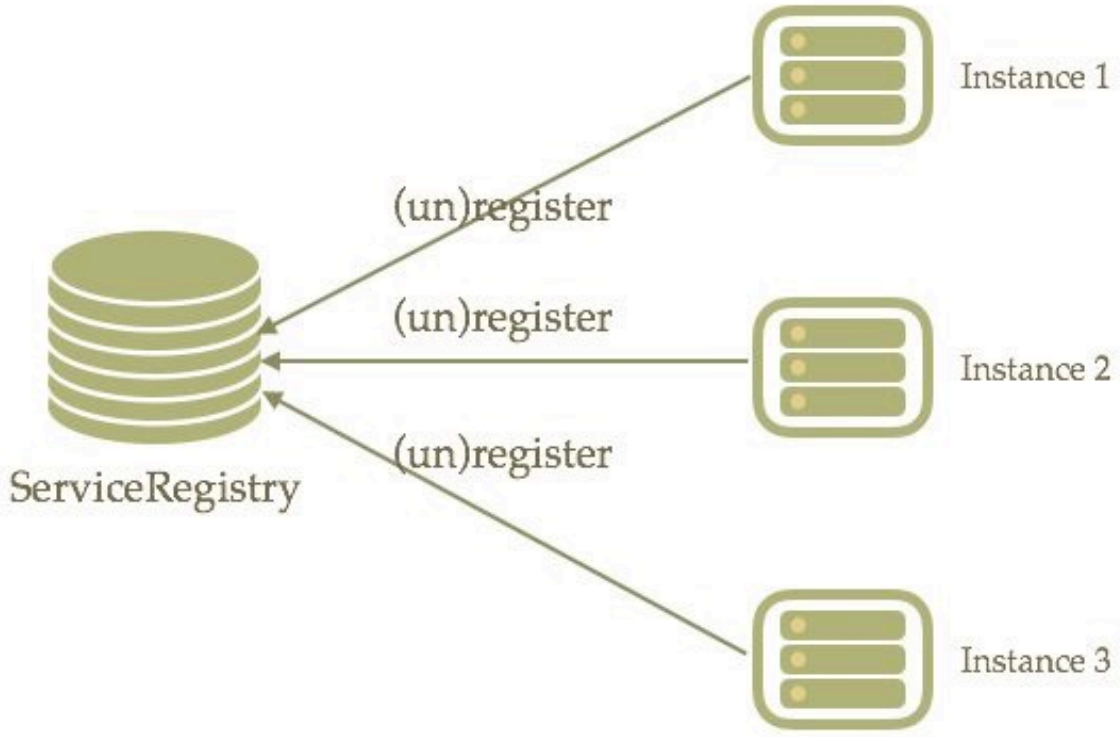
服务注册:

服务要被使用，就需要对外提供服务的位置信息，这个位置信息通常是一个IP地址+端口。在服务只有单个实例并且地址不会动态变化的情况下，服务的位置在使用端可以通过配置文件甚至代码等方式固定死。但在位置信息会动态发生变化的情况下，服务实例就需要将这个地址注册到一个注册中心。

ServiceRegistry: 所有实例在自己可以对外提供服务后，将位置注册到一个ServiceRegistry服务。这个服务具有**固定的位置或域名**，负责保存所有服务实例的位置信息。

服务注册的两种方式:

- **Self Registration:** 由每个服务实例自己实现服务的注册和取消注册的代码;



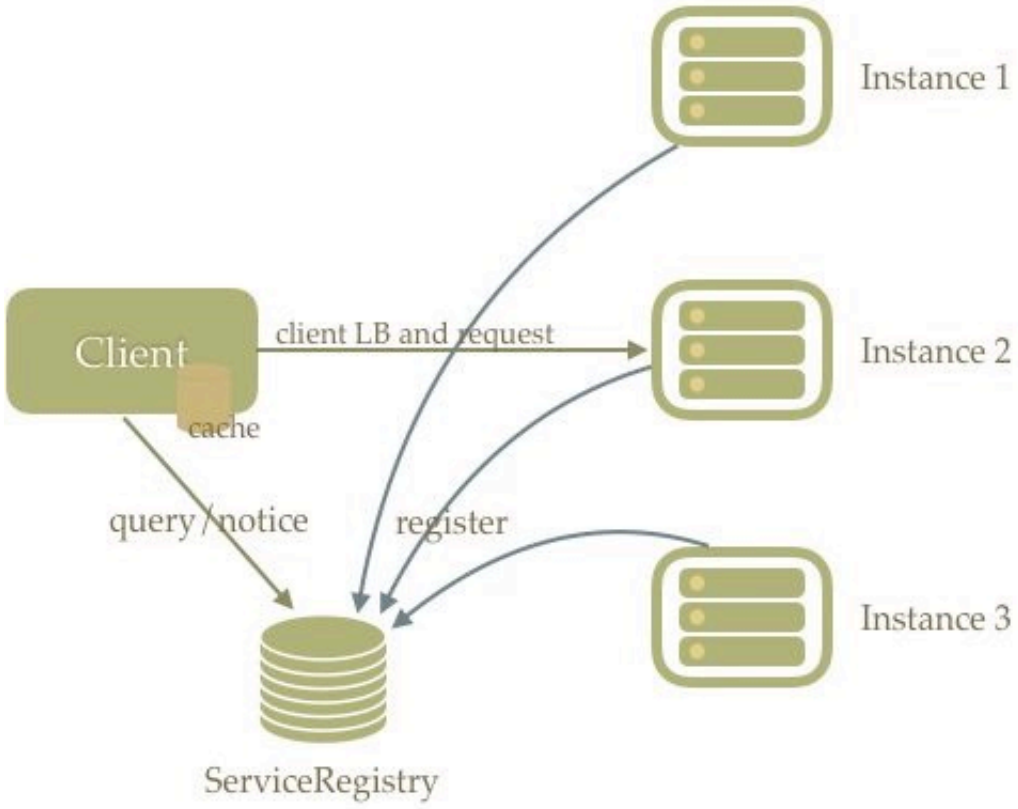
  优点: 服务实例能够更好的掌握注册时机，仅在真正可提供服务时才注册到ServiceRegistry;

  缺点:

  - 所有服务都需要实现注册和取消注册代码，实现复杂且与ServiceRegistry有耦合；
  - 服务的实例很容易在不能提供服务时忘记取消注册；
- **3rd Party Registration:** 由第三方的Registrar完成服务的注册和取消注册。

服务发现:

客户端要使用服务必须通过服务发现技术获取服务的位置信息



服务发现的两种方式:

- **Client-Side Discovery:** 服务实例的发现由Client进行，发现的方式可以是主动到ServiceRegistry查询，也可以由ServiceRegistry通知到Client。在使用Client-Side Discovery时，Client会发现服务的所有实例，并根据LB策略选择一个实例发起请求。
- **Server-Side Discovery:** 在Client和所有的服务实例间增加LoadBalance，Client只需要访问LoadBalance，由LoadBalance负责服务的发现和负载均衡。
- **方案对比:**
  - 不论是Client-Side还是Server-Side的服务发现，执行发现的组件（Client/LoadBalance）通常都需要引入本地缓存，并通过核查保证与ServiceRegistry的一致性。引入缓存可以避免对ServiceRegistry的频繁交互，能够提升性能。
  - Client-Side Discovery相对于Server-Side Discovery有更少的跳数，性能更优。但所有类型的客户端都需要实现服务发现与LB算法，客户端的复杂度高，且与ServiceRegistry耦合。
  - Server-Side Discovery设计中，客户端只需要看到LoadBalance，复杂度低；如果是基于公有云提供服务，则公有云提供商通常会提供现成的服务端LoadBalance。但相对Client-Side Discovery增加了一跳，对性能有一定影响；同时LoadBalance的开发、部署、运维带来了额外的复杂度；

服务通信(进程间的通信)

在单体服务中，最简单的组件间通信方式是语言级别的函数调用，但在微服务架构模式下，不同的微服务部署在不同的进程中，就必须引入Inter-Process Communication（IPC）。不同的微服务具备不同的特点，需要为服务的客户端提供不同的通信方式（**通信方式可以共存**）。

ps: 应当减少服务与服务之间的通信，因为通信必不可免的会进行IO操作，多的通信会导致性能的急剧下降。

IPC通信: RPC、REST、Message

服务通信类型:

	一对一 (LoadBalance)	一对多
同步	请求/响应式	无
异步	通知式	订阅式/发布式
异步	请求/异步响应式	发布/异步响应式

一对一：

一对一是最简单的通信模式，适用于以下几种场景：

1. 服务端仅有一个实例对外提供服务
2. LoadBalance对客户端屏蔽了服务端的实例个数
3. 由客户端实现了LB策略并选择一个服务端实例发起请求

一对多：

一对多适用于以下几种场景：

1. 服务端多个实例可以同时处理相同的请求
2. 没有LoadBalance，但又希望对客户端屏蔽多实例的选择

一对一/一对多以及同步/异步两个维度在组合后，存在以下几种通信方式:

- 请求/响应式：客户端在发出请求后，阻塞式的等待服务端的响应，在响应返回之前，客户端的线程会阻塞；
- 请求/异步响应式：客户端在发出请求后，在服务端的响应返回之前客户端不需要阻塞，可以继续处理其它任务，并在响应返回后通过回调等方式处理；
- 通知式：客户端向服务端发出通知，但并不需要服务端返回响应；
- 订阅/发布式：是一种一对我的通知，此方式常见于各类消息队列技术中，请求端发布请求后，一个或多个订阅了此请求的服务端收到并处理此请求；
- 订阅/异步响应式：与订阅/发布类似，但请求端会收到服务端的响应消息；

对于大部分服务，可能提供一种类型的通信方式即可；但有一些服务可能同时提供多种通信类型。

三种IPC技术:

RPC：

RPC是一种请求/响应式的通信技术。它使得我们可以像调用本地函数一样调用一个远程服务。

缺点：

- RPC自身的优点也带来了它的问题。因为RPC形式上和本地调用完全相同，导致开发人员很容易忽略了它与本地调用的差异，从而引入一些问题。事实上，在使用RPC时必须和使用其它远程通信技术一样考虑通信异常，比如超时、丢包等情况。

- 因为RPC在客户端和服务端需要增加序列化/反序列化的动作，因此对性能会产生一定影响。
- 客户端与服务端的强耦合，如果服务端对服务进行了修改，客户端必须配套修改

常见的RPC技术包括：Apache Thrift、Java RMI、CORBA。

REST：

这里说的REST指的是基于HTTP(s)的RESTful风格的通信技术，它也是一种请求/响应式通信技术，常用于作为RPC的替代方案。

因为它是基于HTTP的，因此它具有以下**优点**：

- 简单，易于开发/测试
- 天生适合于Web类的服务
- 有大量的支撑工具和技术
- 客户端和服务端解耦

REST存在以下**缺点**：

- 通信性能相对较低，不适用于低时延场景
- 需要客户端和服务端自己实现序列化/反序列化

Message (有需要的话在研究，暂时列出)：

消息是一种异步的通信方式，请求方发送一条消息到响应方，响应方如果需要的话回复另一条消息给请求方，可以通过消息方式实现前面的通知式、请求/异步响应式、订阅/发布式、发布/异步响应式的通信。

除了异步通信方式本身的优点，在使用消息方式通信还有以下优点：

- 客户端与服务端解耦。对于使用消息队列通信的双方互相不需要了解对方的位置，只需要使用正确的消息标识（如Topic）
- 可用性高。消息队列通常提供持久化能力，因此请求端在发送请求时并不要求服务端 同时可用，而只要在一定时间内能够处理即可

但同时消息系统具有以下缺点：

- 需要部署单独的消息服务，带来了额外的资源需求，以及部署、运维的复杂性

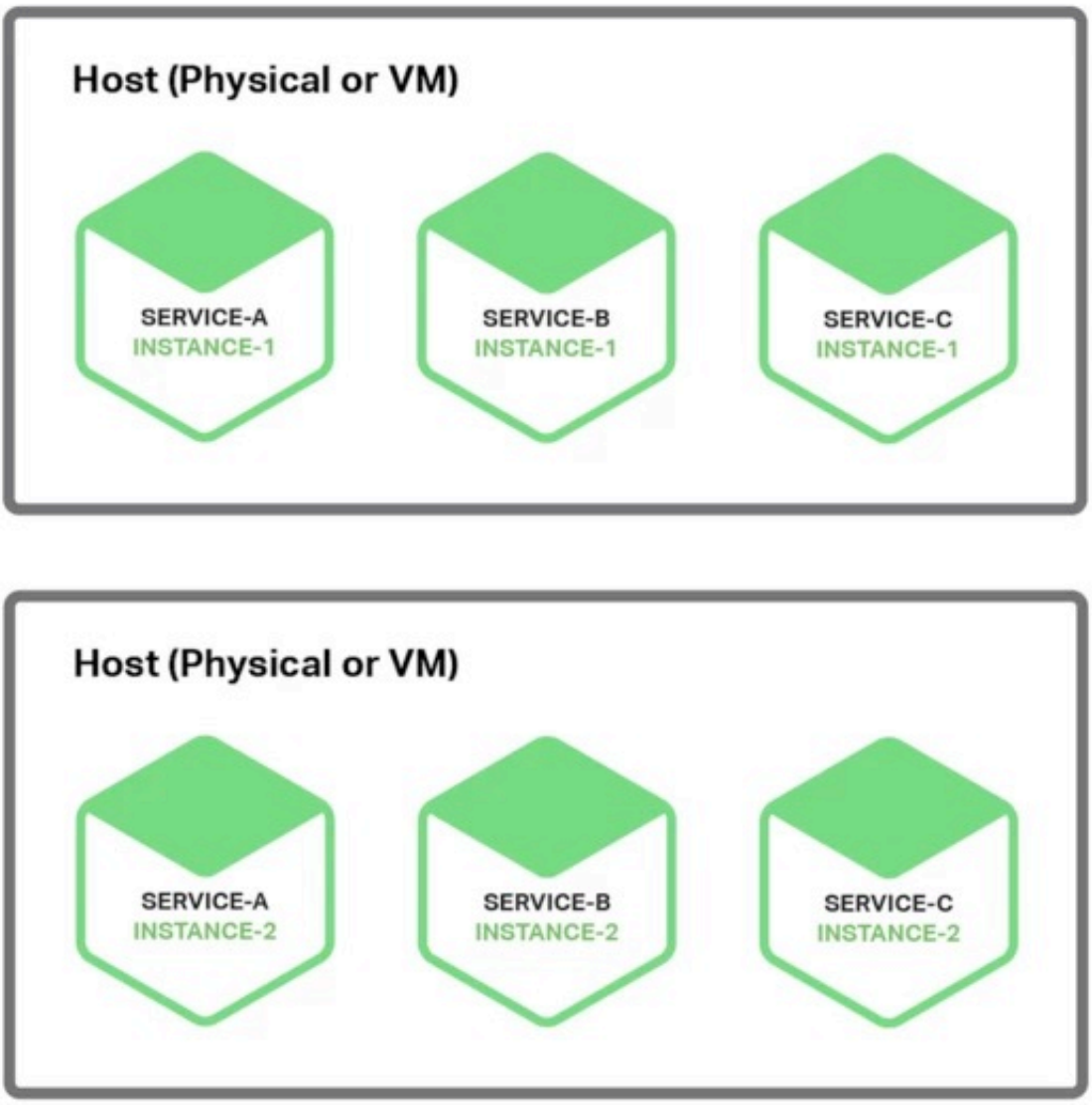
当前有很多开源的消息系统可以使用，比如RabbitMQ、RocketMQ、ActiveMQ等等

微服务部署策略

单主机多服务实例模式:

使用这种模式，需要提供若干台物理或者虚拟机，每台机器上运行多个服务实例。很多情况下，这是传统的应用部署方法。每个服务实例运行一个或者多个主机的well-known端口，主机可以看做宠物。

下图展示的是这种架构：

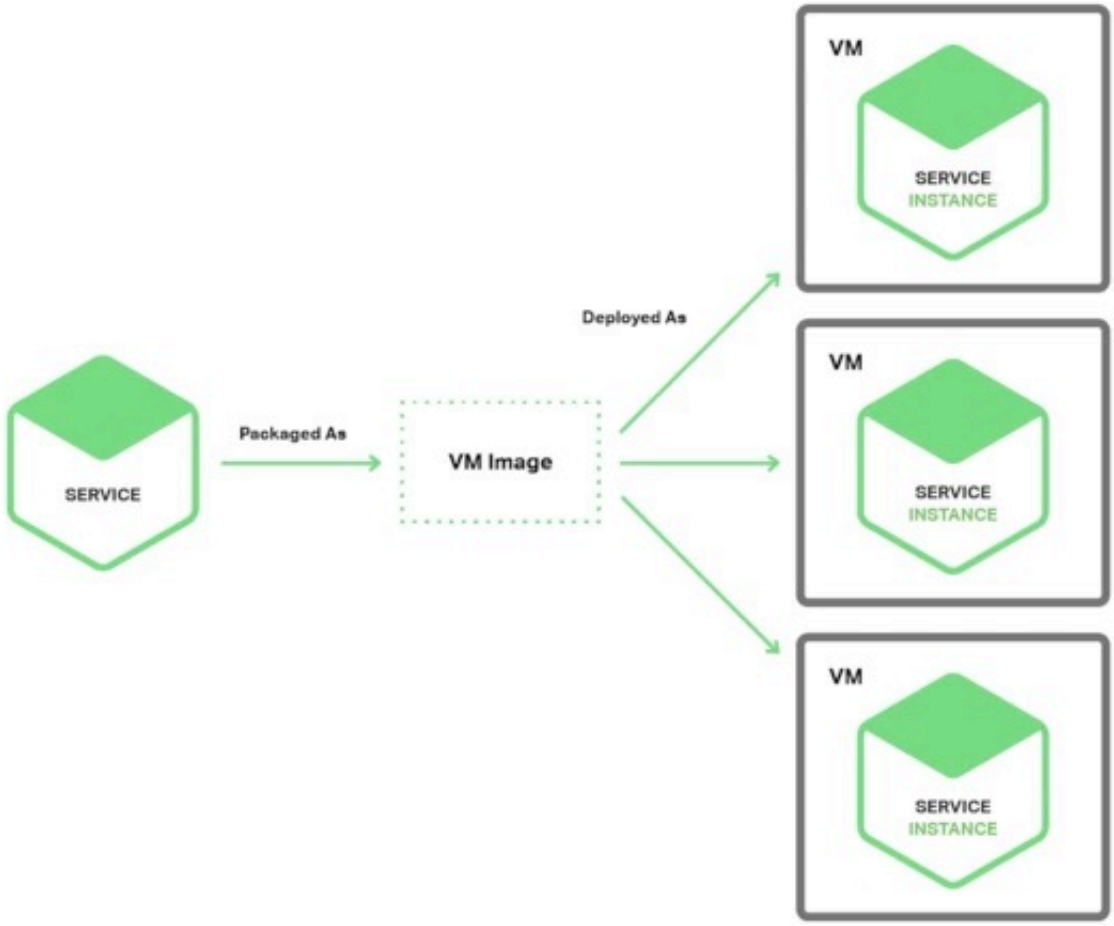


单主机单服务实例模式:

当使用这种模式，每个主机上服务实例都是各自独立的。有两种不同实现模式：单虚拟机单实例和单容器单实例。

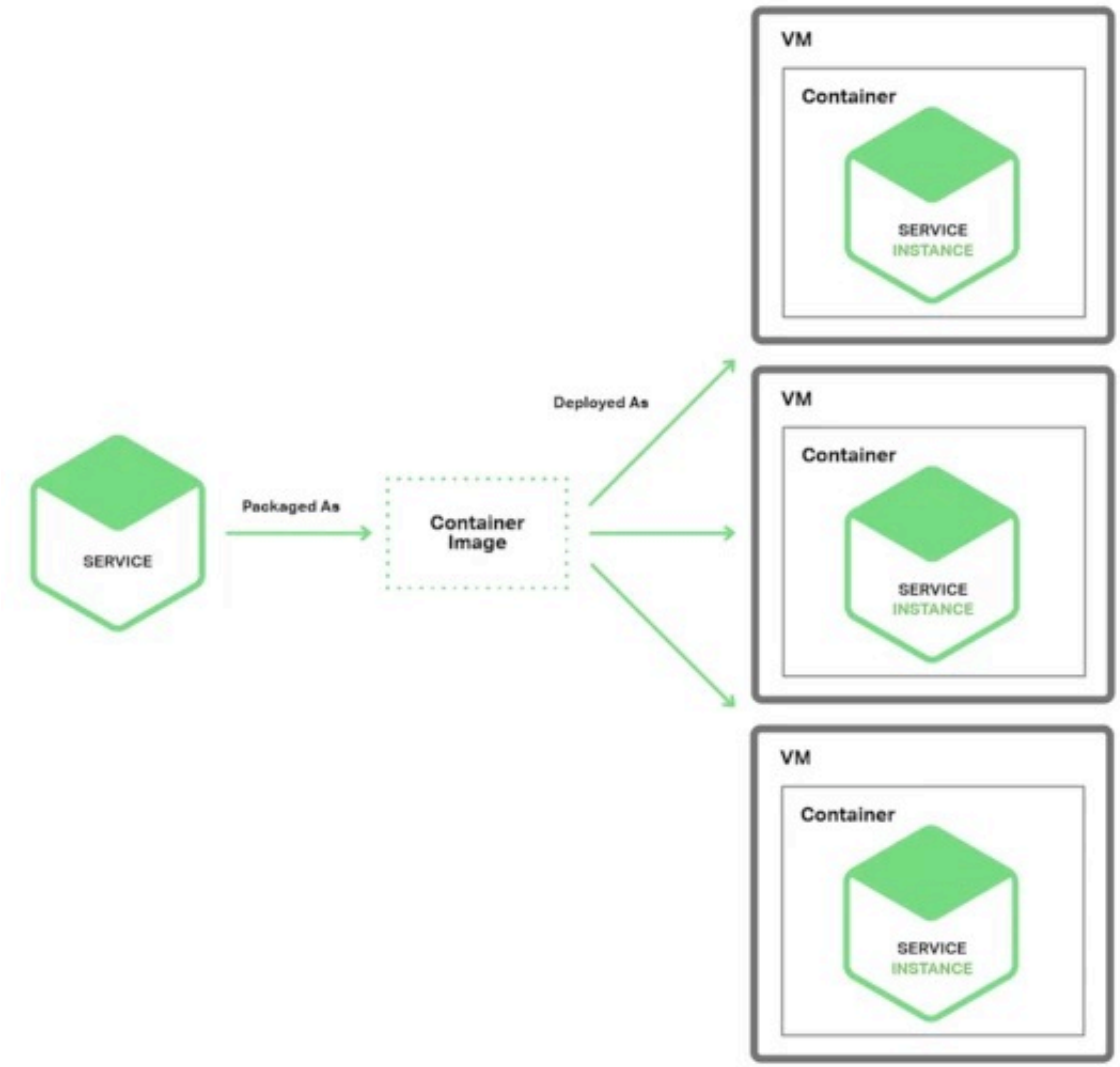
- **单虚拟机单实例模式:** 一般将服务打包成虚拟机映像

  下图展示了此架构:



- **单容器单服务实例模式:** 每个服务实例都运行在各自容器中。容器是运行在操作系统层面的虚拟化机制。一个容器包含若干运行在沙箱中的进程。从进程角度来看，他们有各自的命名空间和根文件系统；可以限制容器的内存和CPU资源。某些容器还具有I/O限制.

  下图展示了这种模式:



使用这种模式需要将服务打包成容器映像。一个容器映像是一个运行包含服务所需库和应用的文件系统 。某些容器映像由完整的linux根文件系统组成，其它则是轻量级的。例如，为了部署Java服务，需要创建包含Java运行库的容器映像，也许还要包含Apache Tomcat server，以及编译过的Java应用。

一旦将服务打包成容器映像，就需要启动若干容器。一般在一个物理机或者虚拟机上运行多个容器，可能需要集群管理系统，例如k8s或者Marathon，来管理容器。集群管理系统将主机作为资源池，根据每个容器对资源的需求，决定将容器调度到那个主机上。

单容器单服务实例模式也是优缺点都有。容器的优点跟虚机很相似，服务实例之间完全独立，可以很容易监控每个容器消耗的资源。跟虚机相似，容器使用隔离技术部署服务。容器管理API也可以作为管理服务的API。

然而，跟虚机不一样，容器是一个轻量级技术。容器映像创建起来很快，例如，在笔记本电脑上，将Spring Boot 应用打包成容器映像只需要5秒钟。因为不需要操作系统启动机制，容器启动也很快。当容器启动时，后台服务就启动了。

使用容器也有一些缺点。尽管容器架构发展迅速，但是还是不如虚机架构成熟。而且由于容器之间共享host OS内核因此并不像虚机那么安全。

另外，容器技术将会对管理容器映像提出许多客制化需求，除非使用如Google Container Engine或者Amazon EC2 Container Service (ECS)，否则用户将同时需要管理容器架构以及虚机架构。

第三，容器经常被部署在按照虚机收费的架构上，很显然，客户也会增加部署费用来应对负载的增长。

有趣的是，容器和虚机之间的区别越来越模糊。如前所述，Boxfuse虚机启动创建都很快，Clear Container技术面向创建轻量级虚机。unikernel公司的技术也引起大家关注，Docker最近收购了Unikernel公司。

(以上部署介绍 来自 ctrl+c、ctrl+v 部署大概也是需要选用 Docker+Nginx 的。具体内容，还需要大佬们的研究)

Docker:

坐等运维大佬来填充。(😏😏😏)

Monolithic 重构

策略一:不要大规模重写代码

- 变化频繁程度:

  优先重构变化频繁的功能，且尽量避免向Monolithic部分再增加新功能，新功能的代码应该尽量添加在改造后的微服务中，从而避免后面改造时带来的重复工作。

- **对周边的依赖情况:**

  可以选择一个与周边依赖较少的部分进行重构，可以避免我们在重构中对其它功能产生太大的影响，同时在重构过程中逐渐积累经验，以应对后续其它功能的重构。

- **资源占用情况:**

  对于资源占用较多，或是资源比较敏感（比如希望能够独占节点资源）的功能，重构为微服务后，可以针对它们定义独立的部署策略，比如采用不同的VM规格，或是独立进行扩容等。

重构方法:

1、根据领域抽取出微服务

一个微服务应该是仅负责单一职责的高内聚、低耦合的软件功能，为了微服务的合理划分，我们需要采用DDD（Domain Driven Design）方法，这要求重构时对系统功能以及每一个功能的Bounded Context有比较好的了解。因为我们是对一个已有系统进行重构，因此此时我们应该比较容易划分出合理的微服务。

2、疏理依赖关系

在提出微服务之前，需要将被划分出来的微服务与其它模块的依赖关系疏理清楚，包括接口和数据。

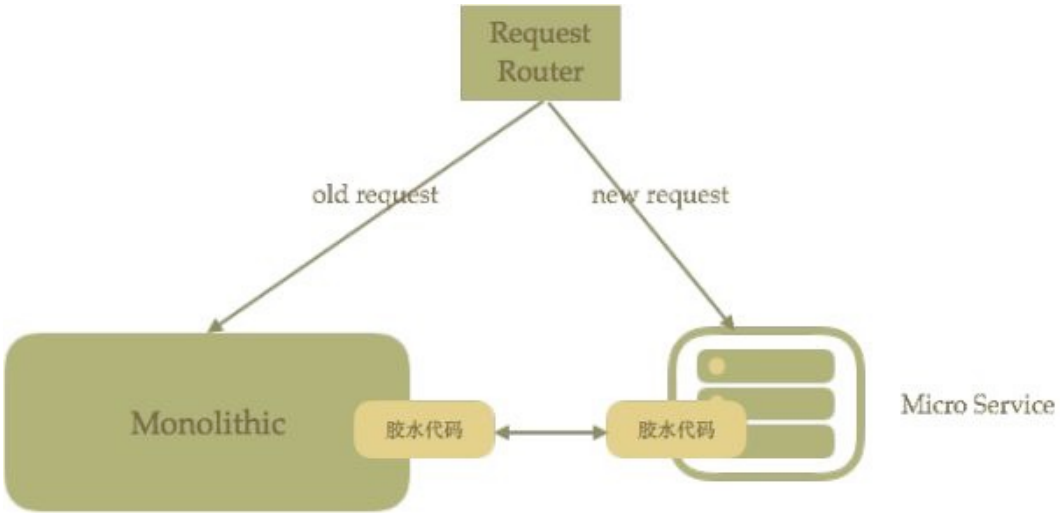
3、解决数据耦合

模块间的接口从API改造成微服务的接口并不复杂，但如果存在数据的耦合就会比较麻烦。在Monolithic形式下，多个模块可能使用了相同的全局变量、访问了相同的数据库和数据库表。

如果数据耦合不存在事务性要求，那么可以简单的原来读写表的操作重构成接口的形式，比如POST/GET的REST形式接口。但对于存在事务性要求的操作，比如购买操作需要同时修改订单表和余额表，就需要引入其它方案来解决，比如通过2PC技术，或Event-Driven的数据处理技术等。这部分将在另一篇博客中介绍。

4、通过Requet Router对请求进行分流；

在解决了上面的问题后，我们就可以构建一个全新的微服务了。此时系统中会存在一个原来的Monolithic系统，我们需要对外部屏蔽系统的不同访问地址，同时新的微服务与原有的系统必然存在交互。



对外提供一个类似API Gateway的Request Router，负责将未改造部分的请求路由到Monolithic部分，而已经完成改造部分的请求路由到新的微服务。微服务和Monolithic会存在一些交互，这部分可以通过添加胶水代码的方式解决。胶水代码可能包括三种方式：

- 直接通过IPC请求
- 直接读取数据
- 同步数据并自己维护

重构ps:

上面的都是扯淡，先本着不重要的接口来进行拆分！！！