软件架构与中间件

涂志莹
tzy_hit@hit.edu.cn

哈尔滨工业大学

软件架构与中间件

Software Architecture and Middleware

第3章
计算层的软件架构技术

第3章 计算层的软件架构技术

3.1 计算层的软件架构技术挑战

3.2 分布式编程模型

3.3 负载均衡

3.4 消息队列

3.5 分布式服务框架

3.5 分布式服务框架

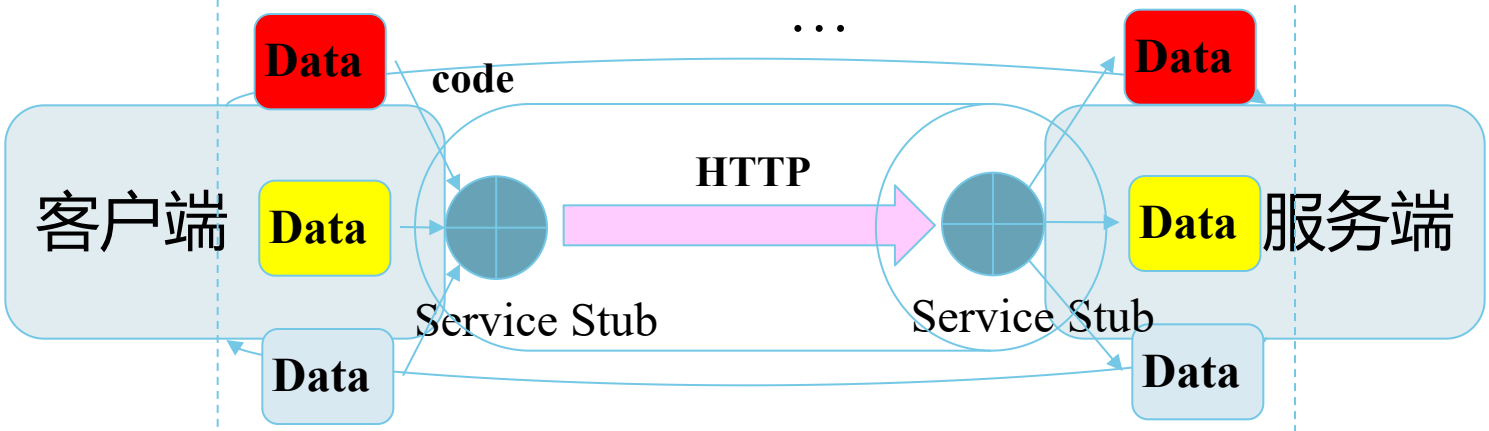
回顾Web Services请求

1. 提出Web Services请求

<?xml version="1.0"?>

<methodCall>

…



2. 传输Web Services反馈

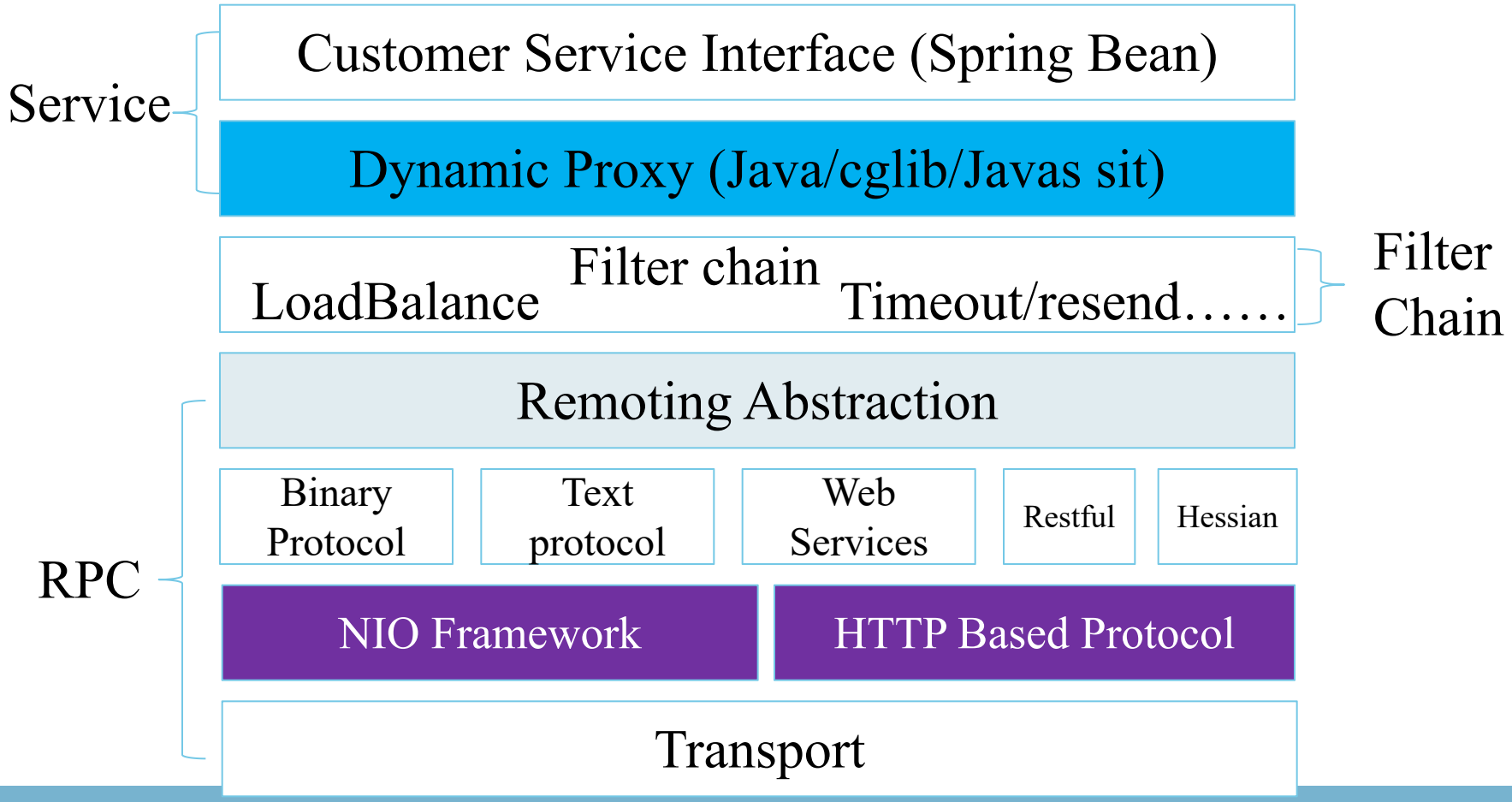
<?xml version="1.0"?>

<methodResponse>

…

分布式服务框架原理

Service层：主要包括Java动态代理，消费者使用，主要用于将服务提供者的接口封装成远程服务调用；Java反射，服务提供者使用，根据消费者请求消息中的接口名、方法名、参数列表反射调用服务提供者的接口本地实现类。再向上就是业务的服务接口定义和实现类，对于使用Spring配置化开发的就是Spring Bean，具体服务逻辑内容由业务部门来实现，平台部分负责将业务接口发布成远程服务。



分布式服务框架的主要功能

- **从功能角度看，分布式服务框架通常包括两大功能：服务治理中心和服务注册中心**
  - **服务注册中心：**负责服务的发布和通知，通常支持对等集群部署，某一个服务注册中心宕机并不会导致整个服务注册中心集群不可用。即便整个服务注册中心全部宕机，也只影响新服务的注册和发布，不影响已经发布的服务的访问。HSF使用的是基于数据库的ConfigServer，Dubbo默认使用Zookeeper，SpringCloud使用eureka。
  - **服务治理中心：**通常包含服务治理接口和服务质量Portal，架构师、测试人员和系统运维人员通过服务治理Portal对服务的运行状态、历史数据、健康度和调用关系等进行可视化的分析和维护，目标就是要持续化服务，防止服务架构腐化，保证服务高质量运行。

分布式服务框架的功能特性

- **服务订阅分布**
  - 配置化发布和引用服务：支持通过XML配置的方法发布和导入服务，降低对业务代码的侵入。
  - 服务自动发现机制：支持服务实时自动发现，由注册中心推送服务地址，消费者不需要配置服务提供者地址，服务地址透明化。
  - 服务在线注册和去注册：支持运行注册新服务，也支持运行态取消某个服务的注册。
- **服务路由**
  - 路由策略：默认提供随机路由、轮循、基于权重的路由策略。避免每个框架使用者都重复开发。
  - 粘滞连接：总是向同一个提供方发起请求，除非此提供方宕机，再切换到另一台。
  - 路由定制：支持用户自定义路由策略，扩展平台的功能。

分布式服务框架

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress ALL -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- 配置数据源 -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
        <property name="username" value="root"/>
        <property name="password" value="password"/>
    </bean>

    <!-- 配置 MyRepository -->
    <bean id="myRepository" class="com.example.repository.MyRepository">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- 配置 MyService -->
    <bean id="myService" class="com.example.service.MyService">
        <property name="myRepository" ref="myRepository"/>
    </bean>

    <!-- 配置一个组件 -->
    <bean id="myComponent" class="com.example.component.MyComponent">
        <property name="myService" ref="myService"/>
    </bean>

</beans>
```

分布式服务框架的功能特性

```
<!-- 配置网关路由 -->
<gateway:route id="route1">
    <gateway:uri>http://192.168.1.100:8080</gateway:uri> <!-- 指向特定 IP 地址和端口 -->
    <gateway:predicates>
        <gateway:predicate name="Path" args="/getPath"/> <!-- 匹配路径 -->
    </gateway:predicates>
    <gateway:filters>
        <gateway:filter name="RequestGlobalFilter"/> <!-- 可以添加全局过滤器 -->
    </gateway:filters>
</gateway:route>

<gateway:route id="route2">
    <gateway:uri>http://192.168.1.100:8080</gateway:uri> <!-- 指向特定 IP 地址和端口 -->
    <gateway:predicates>
        <gateway:predicate name="Path" args="/getRoot"/>
    </gateway:predicates>
</gateway:route>

<!-- 其他路由配置 -->
<gateway:route id="route3">
    <gateway:uri>http://192.168.1.100:8080</gateway:uri> <!-- 指向特定 IP 地址和端口 -->
    <gateway:predicates>
        <gateway:predicate name="Path" args="/getChild"/>
    </gateway:predicates>
    <gateway:filters>
        <gateway:filter name="StripPrefix" args="1"/> <!-- 去掉前缀 -->
    </gateway:filters>
</gateway:route>
```

分布式服务框架的功能特性（续）

- **集群容错**
  - Failover：失败自动切换，当出现失败，重试其他服务器，通常用于读操作；也可用于幂等性写操作。
  - Failback：失败自动恢复，后台记录失败请求，定时重发，通常用于消息通知操作。
  - Failfast：快速失败，只发起一次调用，失败立即报错，通常用于非幂等性的写操作。
- **服务调用**
  - 同步调用：消费者发起服务调用之后，同步阻塞等待服务端响应。
  - 异步调用：消费者发起服务调用之后，不阻塞立即返回，由服务端返回应答后异步通知消费者。
  - 并行调用：消费者同时对多个服务提供者批量发起服务调用请求，批量发起请求后，集中等待应用。

分布式服务框架的功能特性（续）

- **多协议**
  - 私有协议：支持二进制等私有协议，支持私有协议定制和扩展。
  - 公有协议：提供Web Service等公有协议，用于外部服务对接。
- **序列化方式**
  - 二进制类序列化：支持Thrift、Protocol buffer等二进制协议，提升序列化性能。
  - 文本类序列化：支持JSON、XML等文本类型的序列化方式，提升通用性和可读性。
- **统一配置**
  - 本地静态配置：安装部署修改一次，运行态不修改的配置，可以存放到本地配置文件中。
  - 基于配置中心的动态配置：运行态需要调整的参数，统一放到配置中心（服务注册中心），修改之后统一下发，实时生效。

补充：幂等性概念

- **分布式系统中的幂等性概念：**用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。
- **幂等性的重要性：**由于服务无状态的本质，对于业务的敏感性是很弱的。如果不支持幂等性的话，就会导致服务重复操作，对于业务数据进行违背业务逻辑的重复性操作。例如电商服务中的，超卖现象、重复转账、扣款或付款、重复增加金币、积分或优惠券。
- **幂等场景：**
  - 网络波动：因网络波动，可能会引起重复请求
  - 分布式消息消费：任务发布后，使用分布式消息服务来进行消费
  - 用户重复操作：用户在使用产品时，可能无意地触发多笔交易，甚至没有响应而有意触发多笔交易
  - 未关闭的重试机制：因开发、测试、或运维人员未检测出错误的情况下开启的重试机制（如Nginx重试、RPC通信重试或业务层重试等）

补充：数据库的幂等性

- **幂等性的影响往往作用在数据上，而不同数据库操作对于幂等性的反应也不一样：**
  - 新增类请求：不具备幂等性
  - 查询类动作：重复查询不会产生或变更新的数据，查询具有天然幂等性
  - 更新类请求：
    - 基于主键的计算式Update，不具备幂等性，即UPDATE products SET price = price-1 WHERE id = 1
    - 基于主键的非计算式Update：具备幂等性，即UPDATE products SET price = newPrice WHERE id = 1
    - 基于条件查询的更新，不一定具备幂等性（视情而定，一般不具备）
  - 删除类请求：
    - 基于主键的Delete具备幂等性
    - 一般业务层面都是逻辑删除（即update操作），而基于主键的逻辑删除操作也是具有幂等性的

补充：数据库的幂等性解决方案

- **数据库加锁法：**让关键资源的操作串行起来，但是这会引入其他的问题，包括效率问题，死锁问题等。
- **全局唯一ID法：**根据业务的操作和内容生成一个全局ID，在执行操作前先根据这个全局唯一ID是否存在，来判断这个操作是否已经执行。该方案缺点实现起来困难，同时与服务的业务解绑有一定的冲突。
- **去重表法：**在本身具有唯一标识的业务场景下是非常好的方法，利用唯一的标识号，如订单号等，判断操作是否被重复执行。
- **多版本控制法：**为每一次操作添加一次版本号，以示区别。缺点在于版本号的管理，以及通常适用于更新操作，并且往往需要配合日志来完成数据最终一致性。
- **状态机控制法：**这种方法适合在有状态机流转的情况下，比如就会订单的创建和付款，订单的付款肯定是在之前，这时我们可以通过在设计状态字段时，使用int类型，并且通过值类型的大小来做幂等，比如订单的创建为0，付款成功为100，付款失败为99。

微服务定义

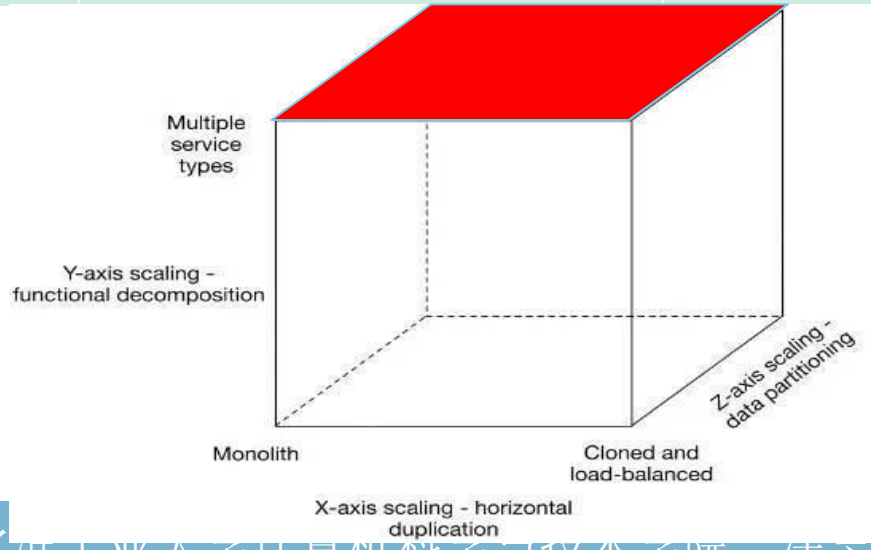
- 微服务是一种架构设计模式。在微服务架构中，业务逻辑被拆分成一系列小而松散耦合的分布式组件，共同构成了较大的应用。每个组件都被称为微服务。
- 每个微服务都在整体架构中执行着单独的任务，或负责单独的功能。
- 每个微服务可能会被一个或多个其他微服务调用，以执行较大应用需要完成的具体任务。
- 系统为任务执行——比如搜索或显示图片任务，或者其他可能需要多次执行的任务提供了统一的解决处理方式，并限制应用内不同地方生成或维护相同功能的多个版本。

微服务定义（续）

- 微服务是围绕业务功能构建的，可以通过全自动部署机制进行独立部署。这些服务的集中化管理已经是最少的，它们可以用不同的编程语言编写，并使用不同的数据存储技术。
- 微服务需要做到：
  - ✓ 负责单个功能
  - ✓ 单独部署
  - ✓ 包含一个或多个进程
  - ✓ 拥有自己的数据存储
  - ✓ 一支小团队就能维护几个微服务
  - ✓ 可替换的

微服务架构与SOA架构的区别

	SOA	微服务
组件大小	大块业务逻辑	单独任务或小块业务逻辑
耦合	通常松耦合	总是松耦合
公司架构	任何类型	小型、专注于功能交叉团队
管理	着重中央管理	着重分散管理
目标	确保应用能够交互操作	执行新功能、快速拓展开发团队

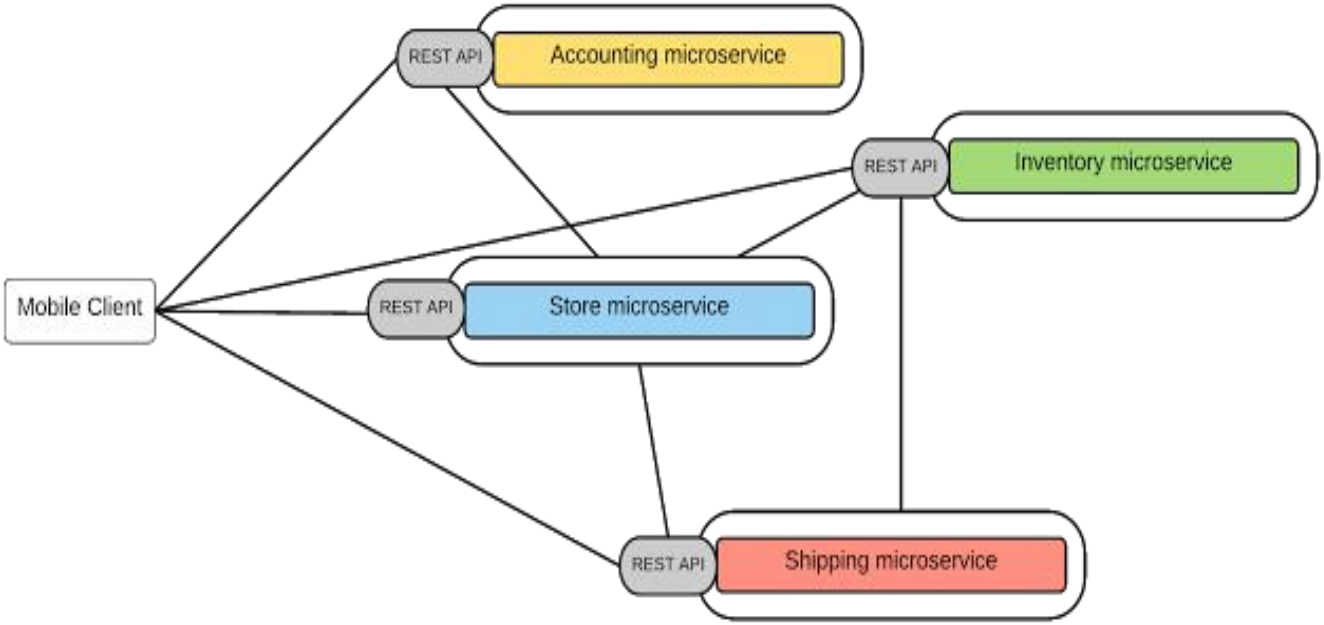


SOA服务架构的服务集成

- SOA体系下，服务之间通过企业服务总线（Enterprise Service Bus）通信，许多业务逻辑在中间层（消息的路由、转换和组织）。
- 微服务架构倾向于降低中心消息总线（类似于ESB）的依赖，将业务逻辑分布在每个具体的服务终端。
- 大部分微服务基于HTTP、JSON这样的标准协议，集成不同标准和格式变的不再重要。另外一个选择是采用轻量级的消息总线或者网关，有路由功能，没有复杂的业务逻辑。

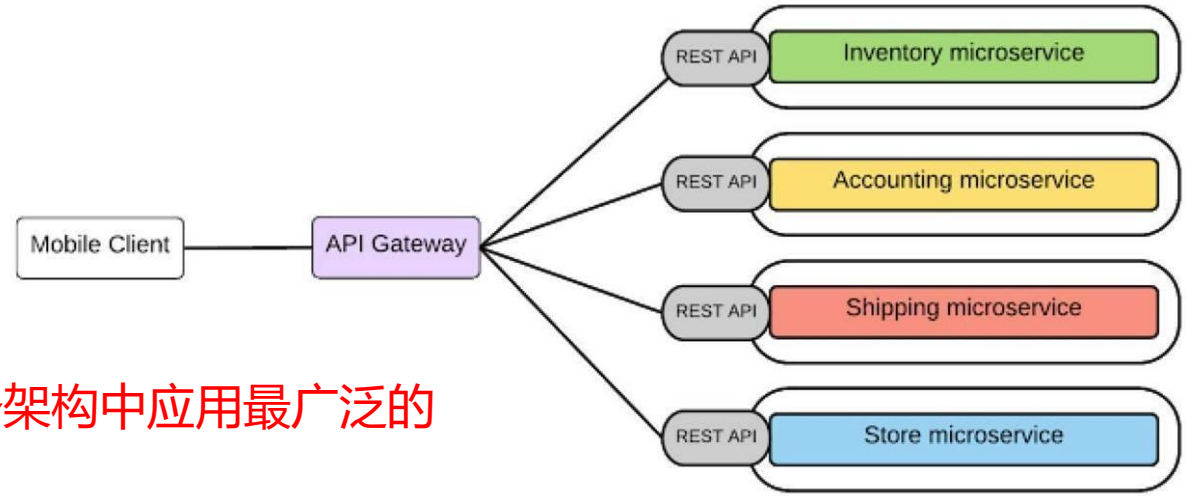
微服务架构的服务集成

- **点对点方式：**直接调用服务，每个微服务都开放REST API，并且调用其它微服务的接口。
- 在比较简单的微服务应用场景下，这种方式还可行，随着应用复杂度的提升，会变得越来越不可维护，这是尽量不采用点对点的集成方式。



微服务架构的服务集成（续）

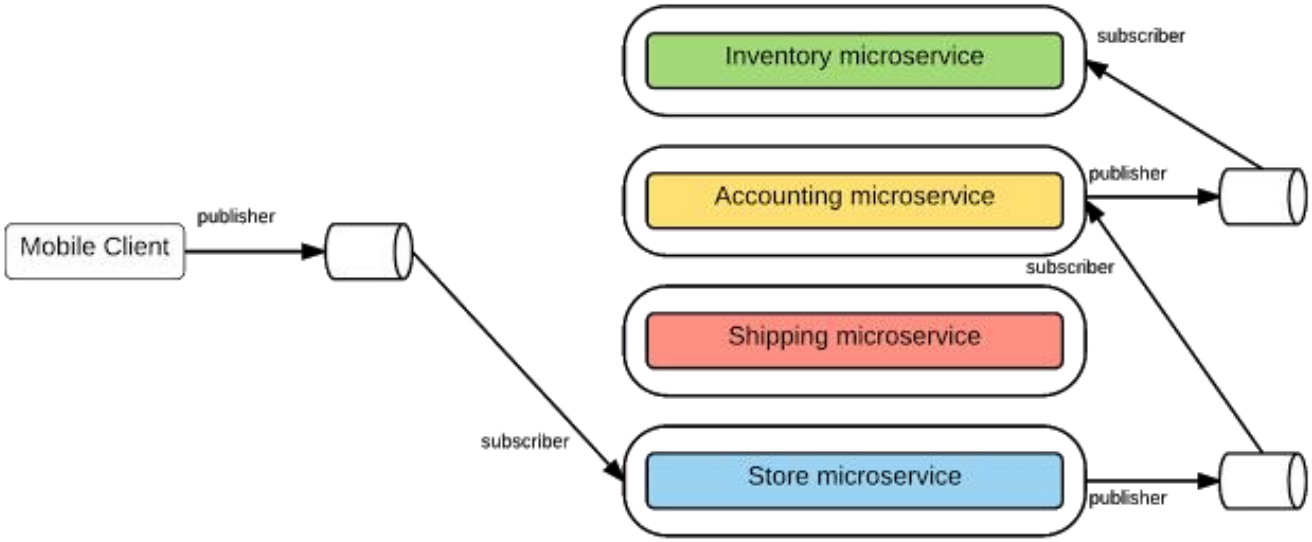
- **API-网关方式：**其核心要点是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能。通常，网关也是提供REST/HTTP的访问API。服务端通过API-GW注册和管理服务。
- **用网上电商为例，所有的业务接口通过API网关暴露，是所有客户端接口的唯一入口。微服务之间的通信也通过API网关。其优势在于：**
  - a.有能力为微服务接口提供网关层次的抽象。比如：微服务的接口可以各种各样，在网关层，可以对外暴露统一的规范接口。
  - b.轻量的消息路由、格式转换。
  - c.统一控制安全、监控、限流等非业务功能。
  - d.每个微服务会变得更加轻量，非业务功能个都在网关层统一处理，微服务只需要关注业务逻辑



目前，API网关方式应该是微服务架构中应用最广泛的设计模式。

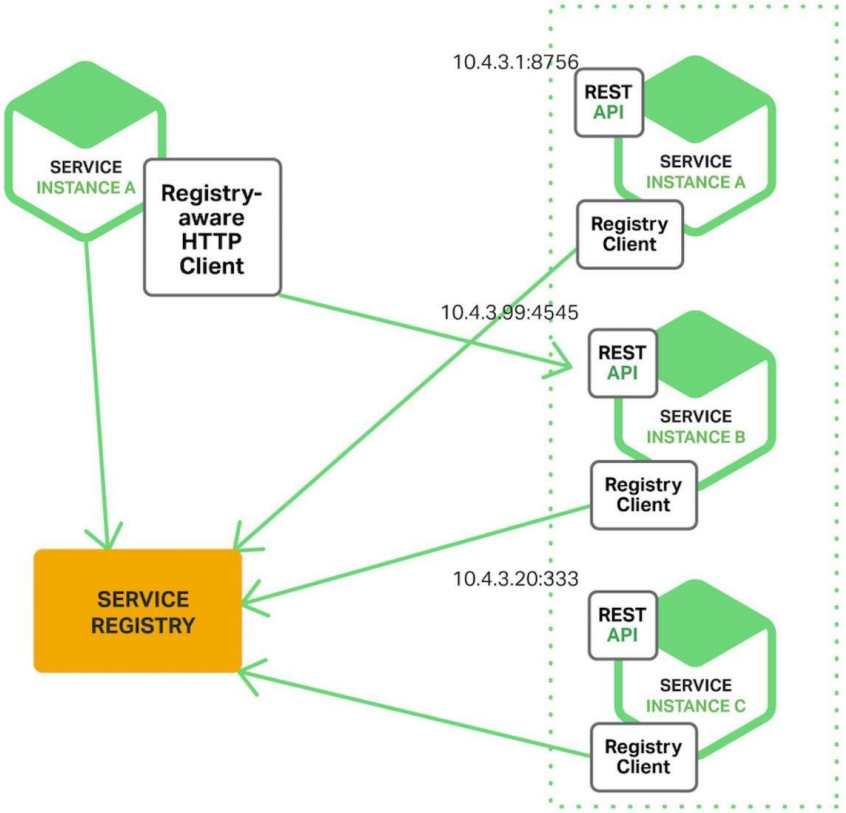
微服务架构的服务集成（续）

- **消息代理方式：**微服务也可以集成在异步的场景下，通过队列和订阅主题，实现消息的发布和订阅。一个微服务可以是消息的发布者，把消息通过异步的方式发送到队列或者订阅主题下。作为消费者的微服务可以从队列或者主题共获取消息。通过消息中间件把服务之间的直接调用解耦。
- 通常异步的生产者/消费者模式，通过AMQP、MQTT等异步消息规范。



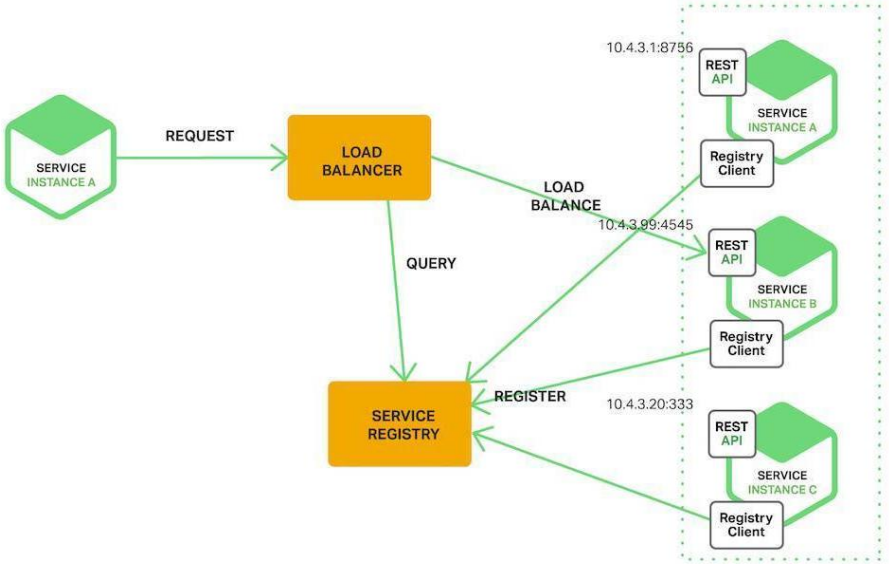
微服务架构的服务发现

- **客户端发现模式**
  - 使用客户端发现模式时，客户端决定相应服务实例的网络位置，并且对请求实现负载均衡。客户端查询服务注册表，后者是一个可用服务实例的数据库；然后使用负载均衡算法从中选择一个实例，并发出请求。
  - 客户端从服务注册服务中查询，其中是所有可用服务实例的库。客户端使用负载均衡算法从多个服务实例中选择出一个，然后发出请求。
  - 服务实例的网络位置在启动时被记录到服务注册表，等实例终止时被删除。服务实例的注册信息通常使用心跳机制来定期刷新。
  - 客户端发现模式优缺点兼有。这一模式相对直接，除了服务注册外，其它部分无需变动。此外，由于客户端知晓可用的服务实例，能针对特定应用实现智能负载均衡，比如使用哈希一致性。这种模式的一大缺点就是客户端与服务注册绑定，要针对服务端用到的每个编程语言和框架，实现客户端的服务发现逻辑。



微服务架构的服务发现（续）

- **服务端发现模式**
  - 客户端通过负载均衡器向某个服务提出请求，负载均衡器查询服务注册表，并将请求转发到可用的服务实例。如同客户端发现，服务实例在服务注册表中注册或注销。
  - AWS Elastic Load Balancer(ELB)是服务端发现路由的例子，ELB 通常均衡来自互联网的外部流量，也可用来负载均衡 VPC（Virtual private cloud）的内部流量。客户端使用 DNS 通过 ELB 发出请求（HTTP 或 TCP），ELB 在已注册的 EC2 实例或 ECS 容器之间负载均衡。这里并没有单独的服务注册表，相反，EC2 实例和 ECS 容器注册在 ELB。
  - 服务端发现模式兼具优缺点。它最大的优点是客户端无需关注发现的细节，只需要简单地向负载均衡器发送请求，这减少了编程语言框架需要完成的发现逻辑。并且，有些部署环境免费提供这一功能。这种模式也有缺点。除非负载均衡器由部署环境提供，否则会成为一个需要配置和管理的高可用系统组件。

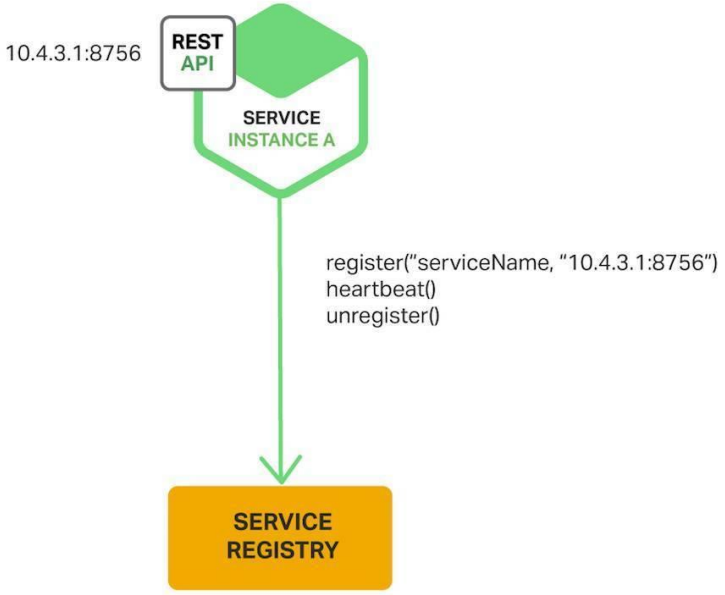


微服务架构的服务注册

- 服务注册表是服务发现的核心部分，是包含服务实例的网络地址的数据库。
- 服务注册表需要高可用而且随时更新。客户端能够缓存从服务注册表中获取的网络地址，然而，这些信息最终会过时，客户端也就无法发现服务实例。因此，服务注册表会包含若干服务端，使用复制协议保持一致性。
- 服务实例必须在注册表中注册和注销。注册和注销有两种不同的方法:
  - 服务实例自己注册，也叫自注册模式（self-registration pattern）。
  - 管理服务实例注册的其它系统组件，即第三方注册模式。

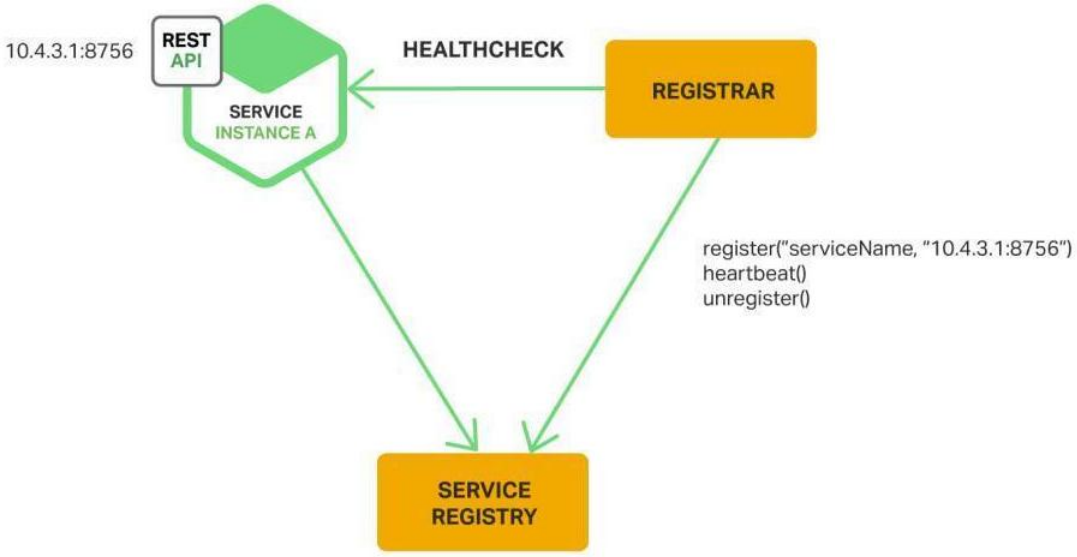
微服务架构的服务注册（续）

- **自注册方式**
  - 当使用自注册模式时，服务实例负责在服务注册表中注册和注销。另外，如果需要的话，一个服务实例也要发送心跳来保证注册信息不会过时。
  - 自注册模式优缺点兼备。它相对简单，无需其它系统组件。然而，它的主要缺点是把服务实例和服务注册表耦合，必须在每个编程语言和框架内实现注册代码。

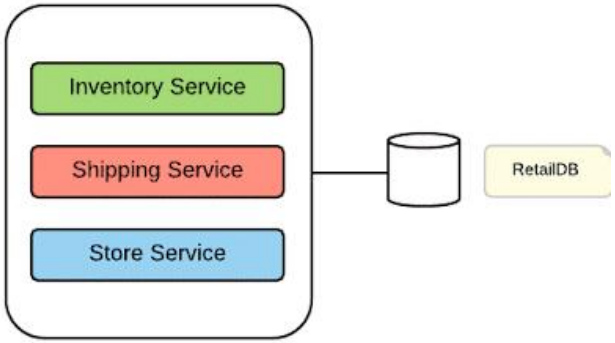


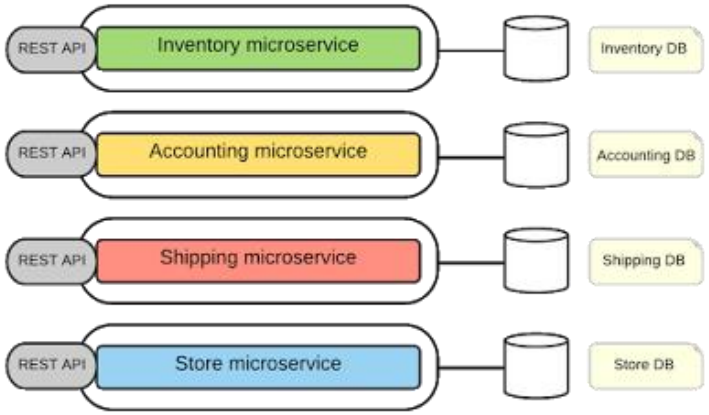
微服务架构的服务注册（续）

- **第三方注册模式**
  - 服务实例则不需要向服务注册表注册；相反，被称为服务注册器的另一个系统模块会处理。服务注册器会通过查询部署环境或订阅事件的方式来跟踪运行实例的更改。一旦侦测到有新的可用服务实例，会向注册表注册此服务。服务管理器也负责注销终止的服务实例。
  - 第三方注册模式也是优缺点兼具。服务与服务注册表解耦合，无需为每个编程语言和框架实现服务注册逻辑；相反，服务实例通过一个专有服务以中心化的方式进行管理。它的不足之处在于，除非该服务内置于部署环境，否则需要配置和管理一个高可用的系统组件。



微服务架构的数据去中心化

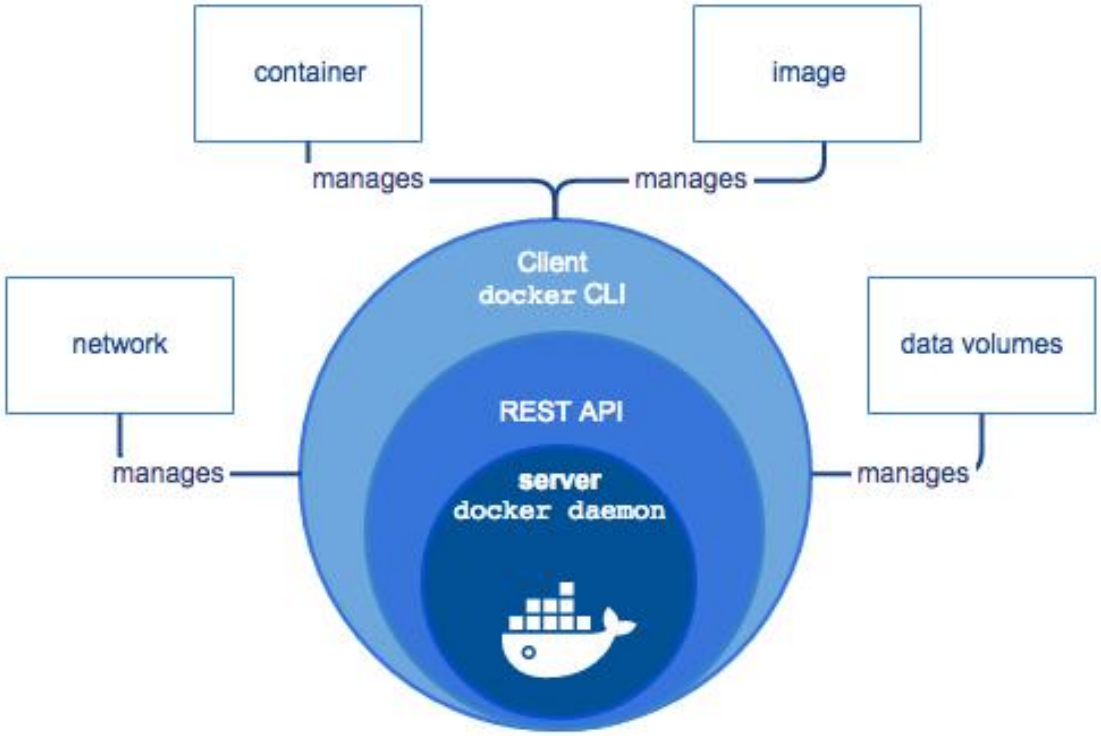




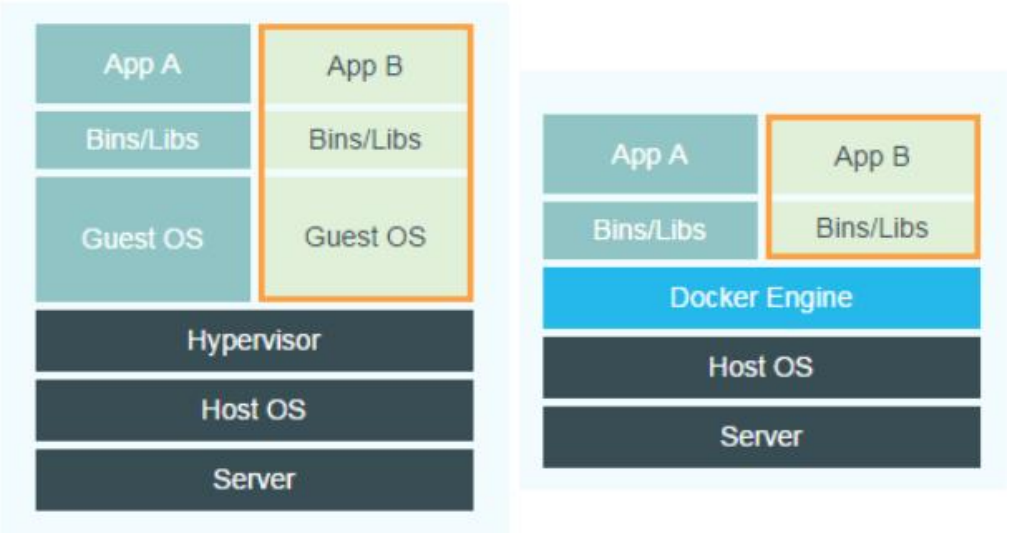
- 单体架构中，不同功能的服务模块都把数据存储在某个中心数据库中。
- 微服务方式，多个服务之间的设计相互独立，数据也应该相互独立（比如，某个微服务的数据库结构定义方式改变，可能会中断其它服务）。因此，每个微服务都应该有自己的数据库。
- **数据去中心化的核心要点：**
  - 1）每个微服务有自己私有的数据库持久化业务数据
  - 2）每个微服务只能访问自己的数据库，而不能访问其它服务的数据库
  - 3）某些业务场景下，需要在一个事务中更新多个数据库。这种情况也不能直接访问其它微服务的数据库，而是通过对于微服务进行操作。
- 数据的去中心化，进一步降低了微服务之间的耦合度，不同服务可以采用不同的数据库技术（SQL、NoSQL等）。在复杂的业务场景下，如果包含多个微服务，通常在客户端或者中间层（网关）处理。

Docker引擎

- Docker引擎是一个C/S结构的应用
- Server是一个常驻进程
- REST API 实现了Client和Server间的交互协议
- CLI 实现容器和镜像的管理，为用户提供统一的操作界面



Docker VS VM



- 虚拟机的Guest OS即为虚拟机安装的操作系统，它是一个完整操作系统内核；虚拟机的Hypervisor层可以简单理解为一个硬件虚拟化平台，它在Host OS是以内核态的驱动存在的。他们被Docker Engine层所替代
- docker有着比虚拟机更少的抽象层。由于docker不需要Hypervisor实现硬件资源虚拟化，运行在docker容器上的程序直接使用的都是实际物理机的硬件资源。因此在CPU、内存利用率上docker将会在效率上有优势，具体的效率对比在下几个小节里给出。

Docker优势

- **更快速的交付和部署**
  - Docker在整个开发周期都可以完美的辅助你实现快速交付。Docker允许开发者在装有应用和服务本地容器做开发。可以直接集成到可持续开发流程中。
- **高效的部署和扩容**
  - Docker 容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑、服务器等。 这种兼容性可以让用户把一个应用程序从一个平台直接迁移到另外一个。
  - Docker的兼容性和轻量特性可以很轻松的实现负载的动态管理。你可以快速扩容或方便的下线的你的应用和服务，这种速度趋近实时。

Docker优势

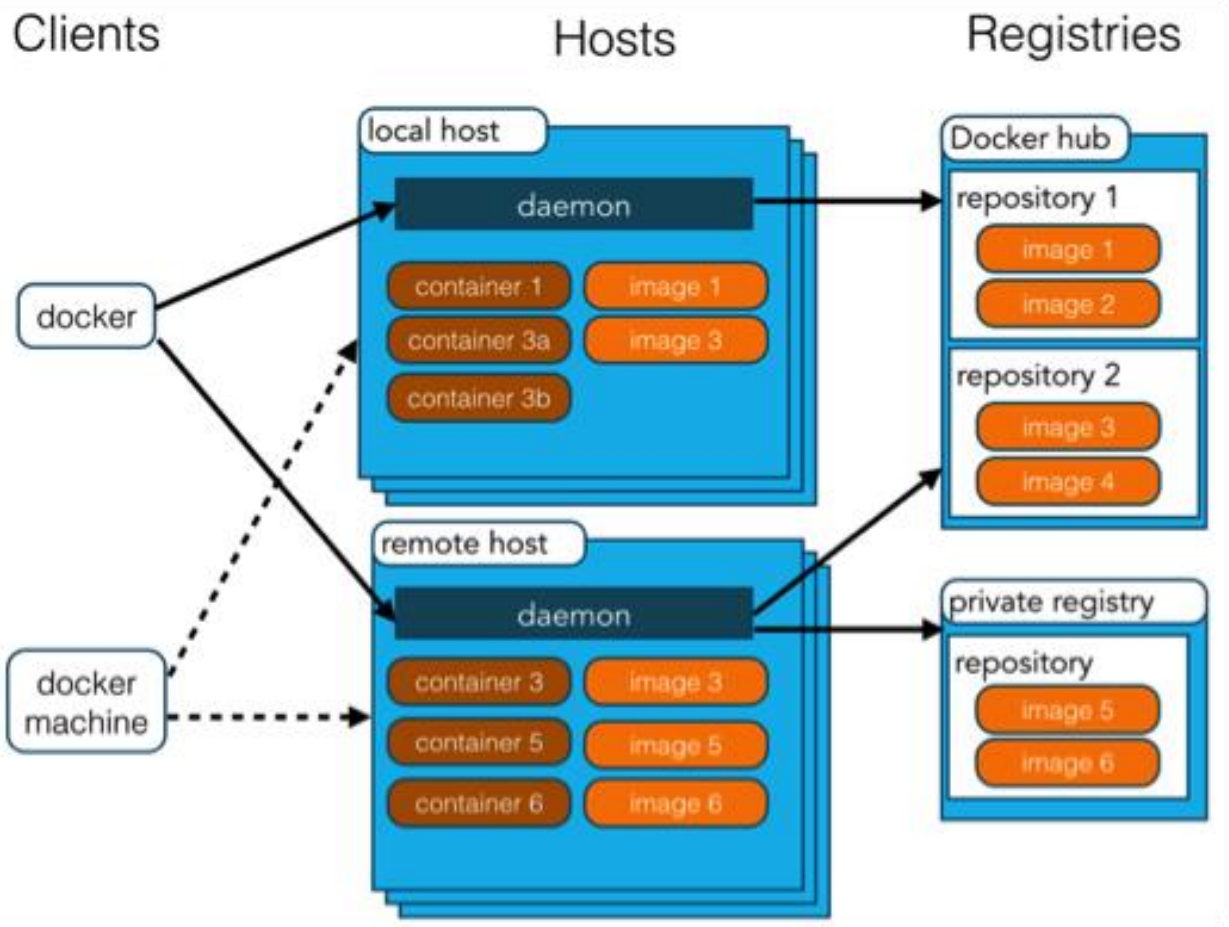
- **更高的资源利用率**
  - Docker 对系统资源的利用率很高，一台主机上可以同时运行数千个 Docker 容器。容器除了运行其中应用外，基本不消耗额外的系统资源，使得应用的性能很高，同时系统的开销尽量小。传统虚拟机方式运行 10 个不同的应用就要起 10 个虚拟机，而Docker只需要启动 10 个隔离的应用即可。
- **更简单的管理**
  - 使用 Docker，只需要小小的修改，就可以替代以往大量的更新工作。所有的修改都以增量的方式被分发和更新，从而实现自动化并且高效的管理。

Docker劣势

- **Docker也不是完美的系统。相对于虚拟机，Docker还存在着以下几个缺点：**
  - 资源隔离方面不如虚拟机，Docker是利用cgroup实现资源限制的，只能限制资源消耗的最大值，而不能隔绝其他程序占用自己的资源。
  - 安全性问题。Docker目前并不能分辨具体执行指令的用户，只要一个用户拥有执行Docker的权限，那么他就可以对Docker的容器进行所有操作，不管该容器是否是由该用户创建。比如A和B都拥有执行Docker的权限，由于Docker的server端并不会具体判断Docker cline是由哪个用户发起的，A可以删除B创建的容器，存在一定的安全风险。
  - Docker目前还在版本的快速更新中，细节功能调整比较大。一些核心模块依赖于高版本内核，存在版本兼容问题

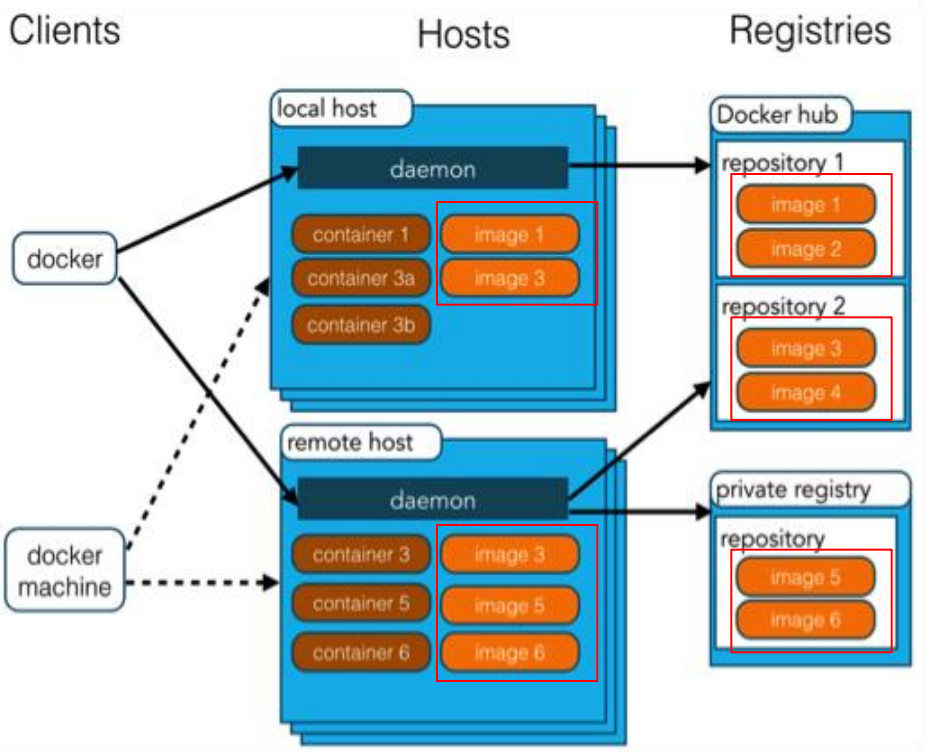
Docker架构

- Docker使用C/S架构，Client 通过接口与Server进程通信实现容器的构建，运行和发布。Client和Server可以运行在同一台集群，也可以通过跨主机实现远程通信。



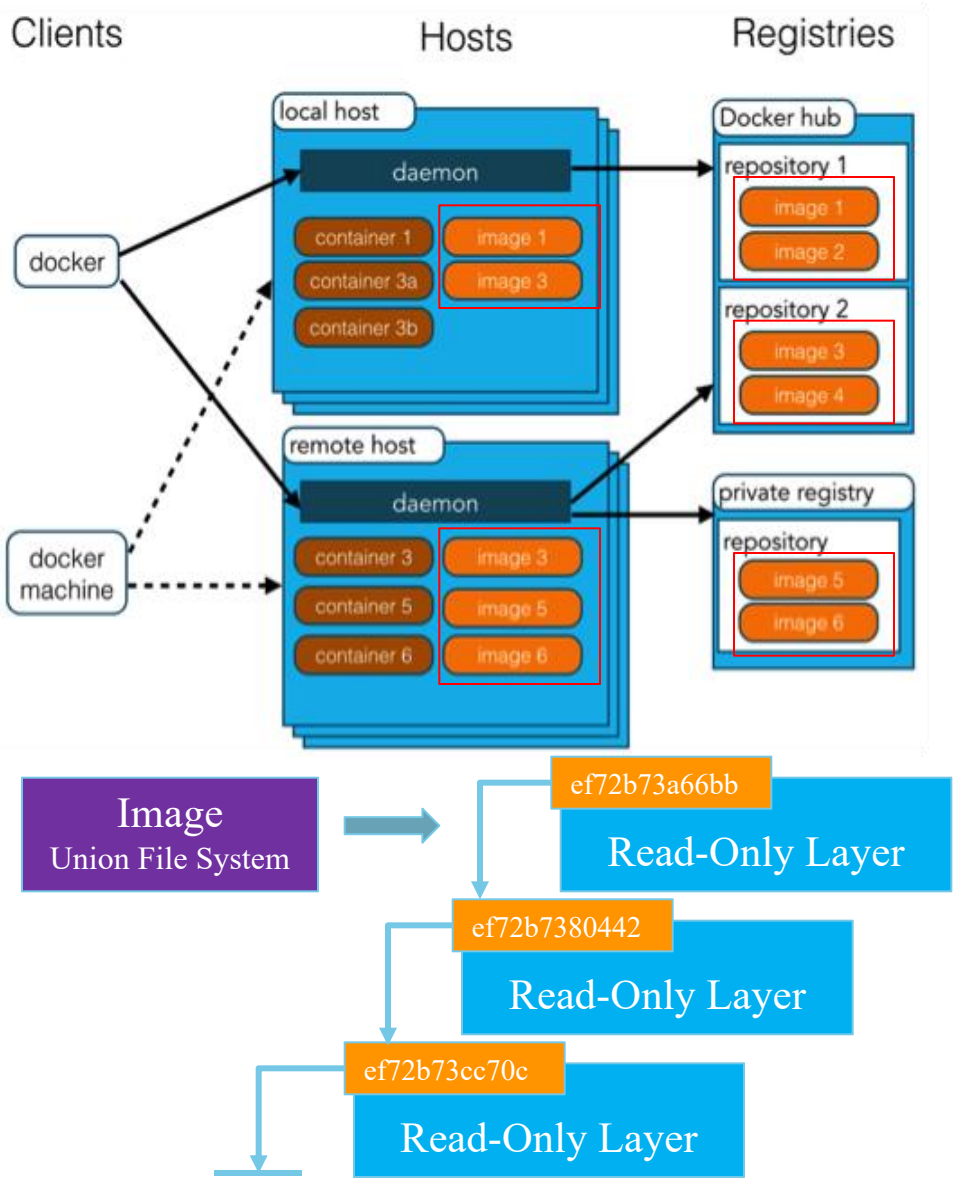
Docker架构

- **Docker 镜像（Image）：**
  - 一个只读的模板，例如：一个镜像可以包含一个完整的操作系统环境，里面仅安装了 Apache 或用户需要的其它应用程序。
  - 镜像可以用来创建 Docker 容器，一个镜像可以创建很多容器。
  - Docker 提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。



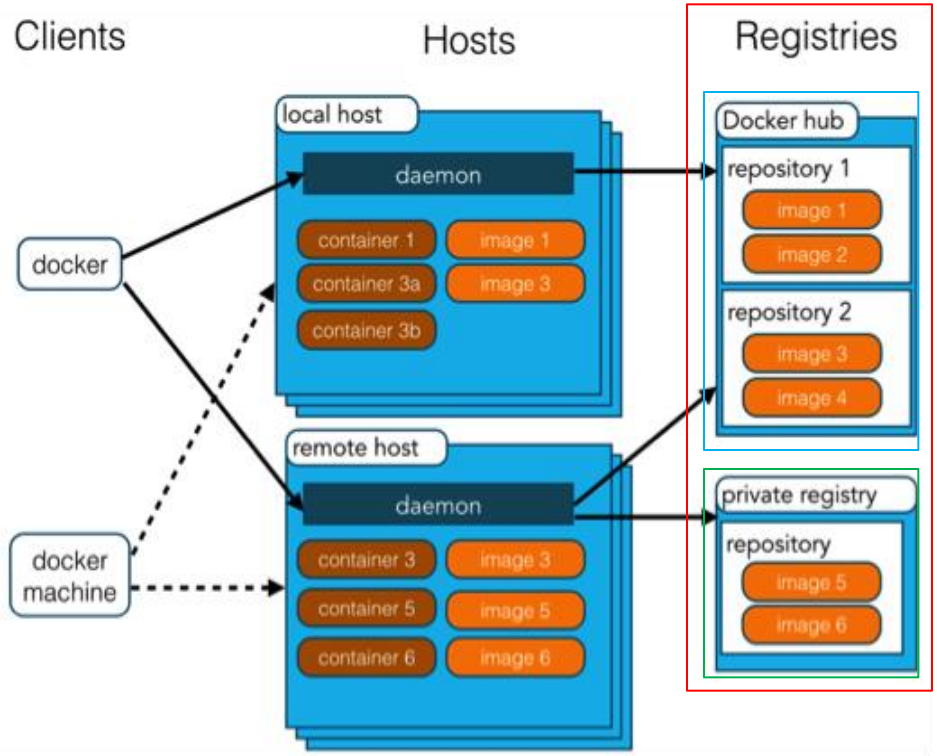
Docker架构

- **Docker 镜像（Image）：**
  - 镜像（Image）就是一堆只读层（read-only layer）的统一视角。
  - 这些只读层，它们重叠在一起。除了最下面一层，其它层都会有一个指针指向下一层。
  - 这些层是Docker内部的实现细节，并且能够在docker宿主机的文件系统上访问到。统一文件系统（Union File System）技术能够将不同的层整合成一个文件系统，为这些层提供了一个统一的视角，这样就隐藏了多层的存在，在用户的角度看来，只存在一个文件系统。



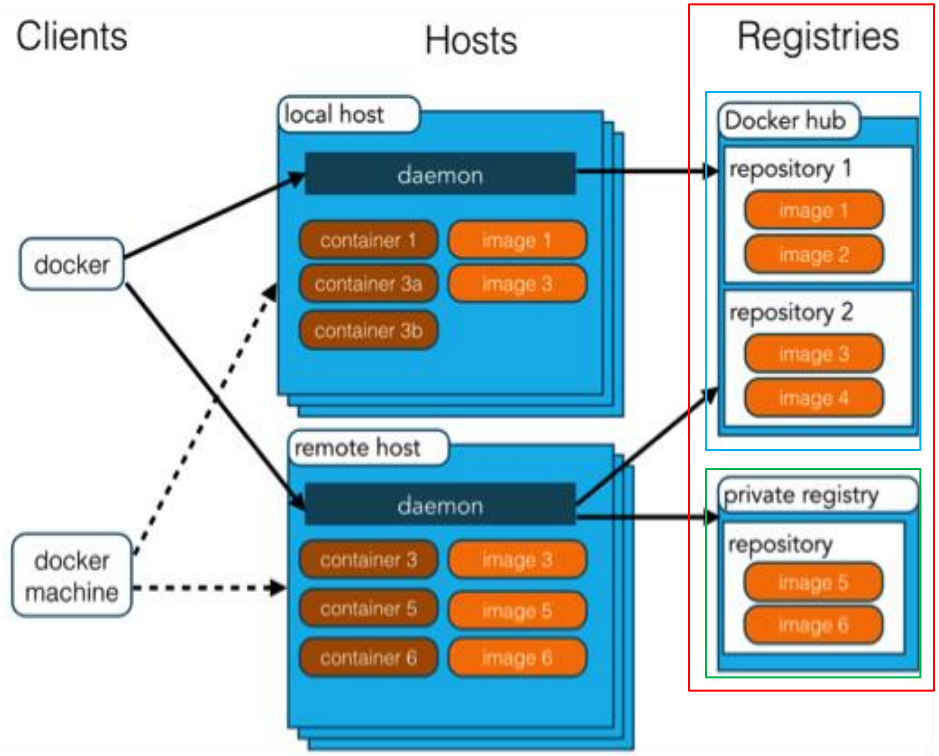
Docker架构

- **仓库（Repository）**
  - 集中存放镜像文件的场所。有时候会把仓库和仓库注册服务器（Registry）混为一谈，并不严格区分。实际上，仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。
  - 仓库分为公开仓库（Public）和私有仓库（Private）两种形式。最大的公开仓库是 Docker Hub，存放了数量庞大的镜像供用户下载。国内的公开仓库包括 时速云 、网易云 等，可以提供大陆用户更稳定快速的访问。当然，用户也可以在本地网络内创建一个私有仓库。



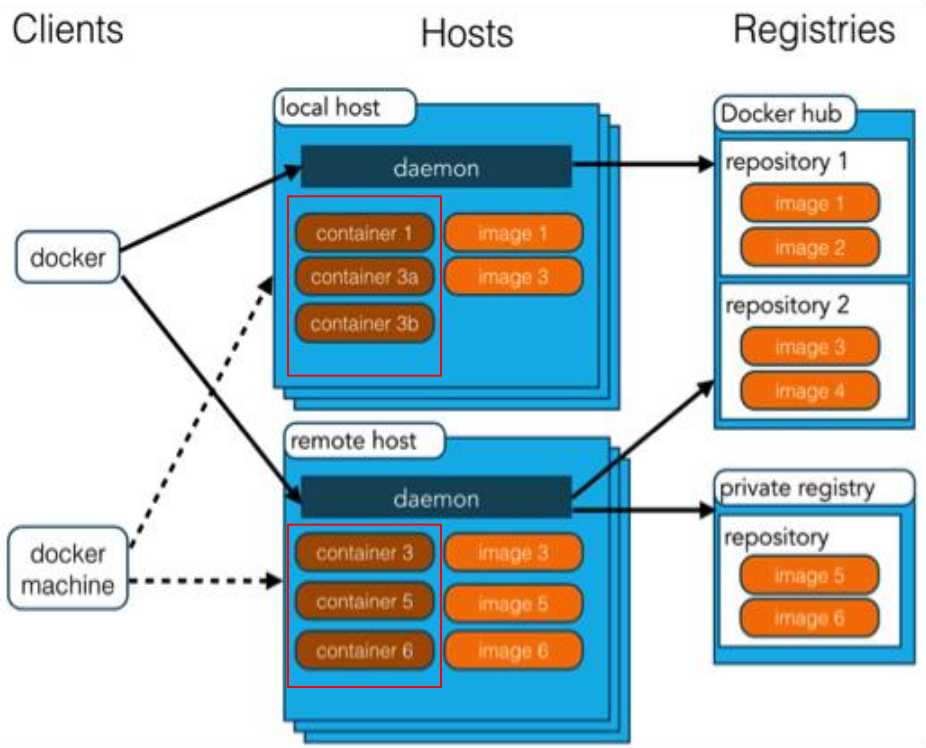
Docker架构

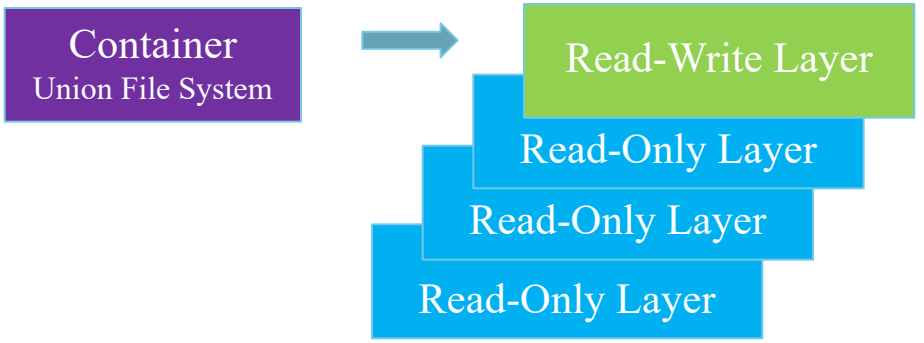
- **仓库（Repository）**
  - 当用户创建了自己的镜像之后就可以使用 push 命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时候，只需要从仓库上 pull 下来就可以了。
  - Docker 仓库的概念跟 Git 类似，注册服务器可以理解为 GitHub 这样的托管服务。



Docker架构

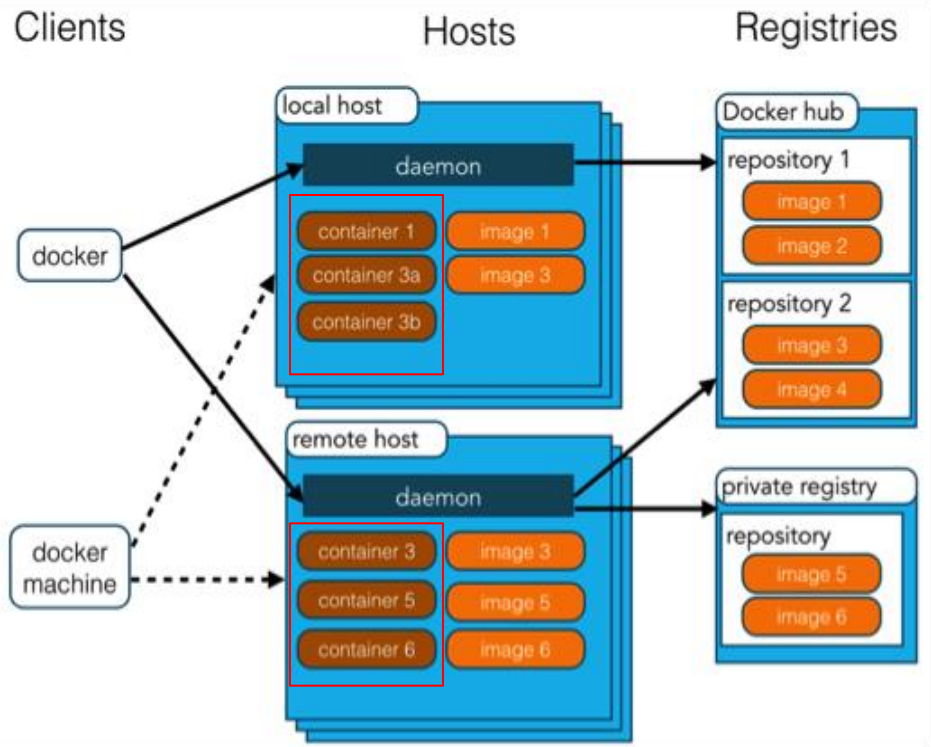
- **容器(container)**
  - Docker 利用容器（Container）来运行应用。容器是从镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。可以把容器看做是一个简易版的 Linux 环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。
  - 容器的定义和镜像几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。





Docker架构

- **容器(container)**
  - 一个运行态容器被定义为一个可读写的统一文件系统加上隔离的进程空间和包含其中的进程。
  - 文件系统隔离技术使得Docker成为了一个非常有潜力的虚拟化技术。一个容器中的进程可能会对文件进行修改、删除、创建，这些改变都将作用于可读写层。



Bash/process

RW Union File System

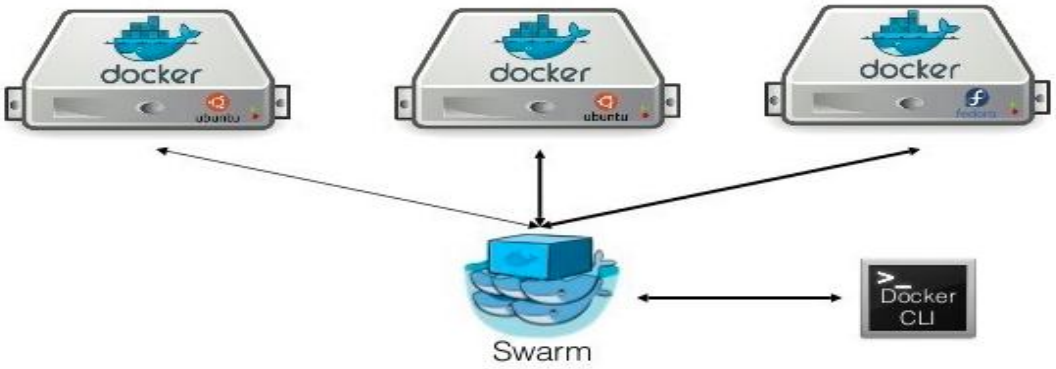
Docker调度工具

- 因为容器没有操作系统或者 hypervisor，容器没有独立运作的能力，所以，它们需要有自己的调度管理工具。它的主要任务就是负责在最合适的主机上启动容器，并且将它们关联起来。它必须能够通过自动的故障转移（fail-overs）来处理错误，并且当一个实例不足以处理/计算数据时，它能够扩展容器来解决问题。
- 下面介绍三个的主流的容器调度工具。
  - Docker开发了Swarm，已被整合进了 Docker Toolbox。
  - Apache Mesos & Mesosphere Marathon，目标建立一个高效可扩展容器调度系统
  - Google开源的Kubernetes，是一种较为成熟的容器管理器。
- 上述三种工具都能解决缺乏独立运行能力的问题，通过提供一个能跨多主机、多数据中心、多云环境运行的系统。

Docker调度工具-Swarm

- **Swarm**
  - Docker Swarm，现在作为一个 beta 版本，是一个 Docker 的集群化工具。它通过使用一个或者多个 Docker 主机来组成一个 Swarm 集群。
  - Docker的开发者将Docker Machine、Compose 和 Swarm 整合进了 Docker Toolbox 中。开发者可利用这三项工具进行配置，进而完成容器的配置、管理或者集群化容器。
  - Swarm 的设计是将容器打包到主机上，所以它能为更大的容器预留其他的主机资源。较之于随机地将容器调度到集群中的一个主机上，这种集群的组成方式能取得更加经济的伸缩性能。

Docker调度工具-Swarm



- 一个机器运行了一个Swarm的镜像（就像运行其他Docker镜像一样），它负责调度容器，在图片上鲸鱼代表这个机器。Swarm使用了和Docker标准API一致的API，这意味着在Swarm上运行一个容器和在单一主机上运行容器使用相同的命令。尽管有新的flags可用，但是开发者在使用Swarm的同时并不需要改变它的工作流程。
- Swarm由多个代理（agent）组成，把这些代理称之为节点（node）。这些节点就是主机，这些主机在启动Docker daemon的时候就会打开相应的端口，以此支持Docker远程API。这些机器会根据Swarm调度器分配给它们的任务，拉取和运行不同的镜像。
- 当启动Docker daemon时，每一个节点都能够被贴上一些标签（label），这些标签以键值对的形式存在，通过标签就能够给予每个节点对应的细节信息。当运行一个新的容器时，这些标签就能够被用来过滤集群。

Docker调度工具-Swarm

- **Swarm的调度策略**
  - random策略：随机选择节点。一般用于开发测试阶段。
  - spread策略：默认策略，swarm优先选择占用资源（如CPU、内存等）最少的节点，能保证集群中所有节点资源的均匀使用。
  - binpack策略：与spread相反，它的目的是尽可能地填满一个节点，以保证更多空余的节点。

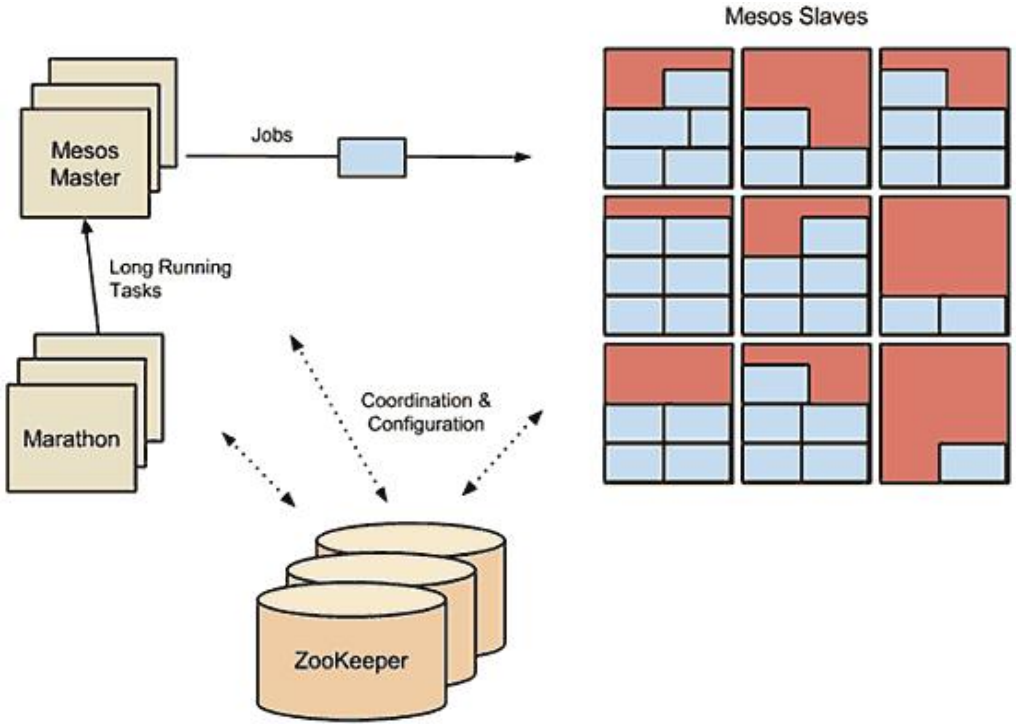
Docker调度工具-Mesos

- Apache Mesos & Mesosphere Marathon的目的就是建立一个高效可扩展的系统，并且这个系统能够支持很多各种各样的框架。
- Mesos的提出就是为了在底部添加一个轻量的资源共享层（resource-sharing layer），这个层使得各个框架能够适用一个统一的接口来访问集群资源。Mesos并不负责调度而是负责委派授权，毕竟很多框架都已经实现了复杂的调度。
- 取决于用户想要在集群上运行的作业类型，共有四种类型的框架可供使用，其中有一些支持原生的Docker。

  Marathon主要是由Mesosphere维护，并且提供了很多关于调度的功能，比如说约束（constraints），健康检查（health checks），服务发现（service discovery）和负载均衡（load balancing）。

Docker调度工具-Mesos

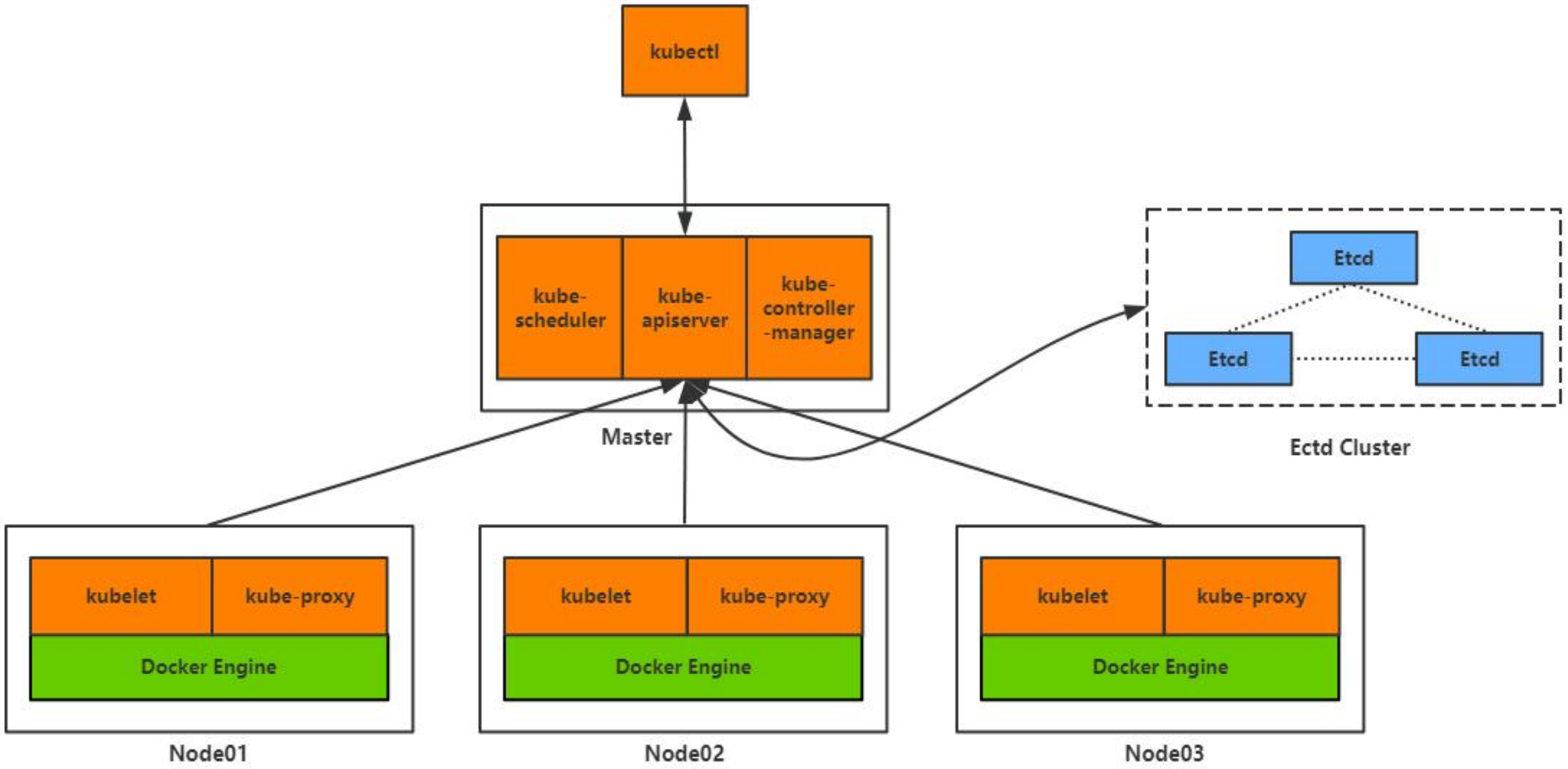
- **集群中一共出现了4个模块：**
  - ZooKeeper帮助Marathon查找Mesos master的地址，同时它具有多个实例可用，以此应付故障的发生。
  - Marathon负责启动，监控，扩展容器。
  - Mesos maser则给节点分配任务，同时如果某一个节点有空闲的CPU/RAM，它就会通知Marathon。
  - Mesos slave运行容器，并且报告当前可用的资源。



Docker调度工具-K8S

- **Kubernetes**
  - **一个开源的可以用来自动部署、伸缩和管理容器化应用的系统。Kubernetes集群包含一些基本组成部分：**
    - Kubernetes成组地部署和调度容器，这个组叫Pod，常见的Pod包含一个到五个容器，它们协作来提供一个 Service。
    - Kubernetes默认使用扁平的网络模式。通过让在一个相同 Pod 中的容器共享一个 IP 并使用 localhost 上的端口，允许所有的 Pod 彼此通讯。
    - Kubernetes 使用 Label 来搜索和更新多个对象，就好像对一个集合进行操作一样。
    - Kubernetes 会搭设一个 DSN 服务器来供集群监控新的服务，然后可以通过名字来访问它们。
    - Kubernetes 使用 Replication Controller 来实例化的 Pod。作为一个提升容错性的机制，这些控制器对一个服务的中运行的容器进行管理合监控。

Docker调度工具-K8S



Docker调度工具-K8S

- **Kubernetes**
  - Docker容器的编排系统，它使用label和pod的概念来将容器换分为逻辑单元。Pods是同地协作（co-located）容器的集合，这些容器被共同部署和调度，形成了一个服务，这是Kubernetes和其他两个框架的主要区别。相比于基于相似度的容器调度方式（就像Swarm和Mesos），这个方法简化了对集群的管理。
  - Kubernetes调度器的任务就是寻找那些PodSpec.NodeName为空的pods，然后通过对它们赋值来调度对应集群中的容器。相比于Swarm和Mesos，Kubernetes允许开发者通过定义PodSpec.NodeName来绕过调度器。调度器使用谓词（predicates）和优先级（priorites）来决定一个pod应该运行在哪一个节点上。通过使用一个新的调度策略配置可以覆盖掉这些参数的默认值。

Docker调度工具-K8S

- **Kubernetes**
  - 谓词（**Predicates**）
    - **谓词是强制性的规则，它能够用来调度集群上一个新的pod。如果没有任何机器满足该谓词，则该pod会处于挂起状态，知道有机器能够满足条件。可用的谓词如下所示：**
      - Predicate：节点的需求
      - PodFitPorts：没有任何端口冲突
      - PodFitsResurce：有足够的资源运行pod
      - NoDiskConflict：有足够的空间来满足pod和链接的数据卷
      - MatchNodeSelector：能够匹配pod中的选择器查找参数。
      - HostName：能够匹配pod中的host参数。

Docker调度工具-K8S

- **Kubernetes**
  - 优先级（**Priorities**）
    - **如果调度器发现有多个机器满足谓词的条件，那么优先级就可以用来判别哪一个才是最适合运行pod的机器。优先级是一个键值对，key表示优先级的名字，value是该优先级的权重。可用的优先级如下：**
      - Priority：寻找最佳节点
      - LeastRequestdPriority：计算pods需要的CPU和内存在当前节点可用资源的百分比，具有最小百分比的节点就是最优的。
      - BalanceResourceAllocation：拥有类似内存和CPU使用的节点。
      - ServicesSpreadingPriority：优先选择拥有不同pods的节点。
      - EqualPriority：给所有集群的节点同样的优先级，仅仅是为了做测试。

第3章

计算层的软件架构技术

Thanks for listening

涂志莹

哈尔滨工业大学计算机学院

企业与服务计算研究中心