哈尔滨工业大学 国家示范性软件学院

第五章
微服务架构

杨大易

2021/12/14

本章内容

1. 微服务基本理论
2. Docker
3. 调度工具

1.1 什么是微服务

- 为什么微服务？
  - 传统的IT行业软件大多都是各种独立系统的堆砌，存在扩展性差，可靠性不高，维护成本高等问题。
  - 后来引入了SOA服务化，但是 SOA 早期均使用了总线模式，这种模式是与某种技术栈强绑定的，如J2EE。导致很多企业的遗留系统对接难度高，切换时间长，成本高，新系统稳定性收敛也需要较长时间。
  - 最终 SOA 看起来很美，但却成为了企业级奢侈品，中小公司都望而生畏。

1.1 什么是微服务

- 微服务最早由Martin Fowler与James Lewis于2014年共同提出。

第一代：单体架构

- 紧耦合，
- 系统复杂、错综交互，动一发而牵全身
- 重复制造各种轮子：OS、DB、Middleware
- 完全封闭的架构

第二代：SOA架构

- 松耦合
- 通常通过ESB进行系统集成
- 有状态
- 大团队：100~200人
- TTM: 1年、半年、月
- 集中式、计划内停机扩容

第三代：微服务架构

- 解耦
- 小团队：2 Pizza Team
- TTM: 按天、周进行升级发布
- DevOps: CI, CD, 全自动化
- 可扩展性：自动弹性伸缩
- 高可用：升级、扩容不中断业务

1.1 什么是微服务

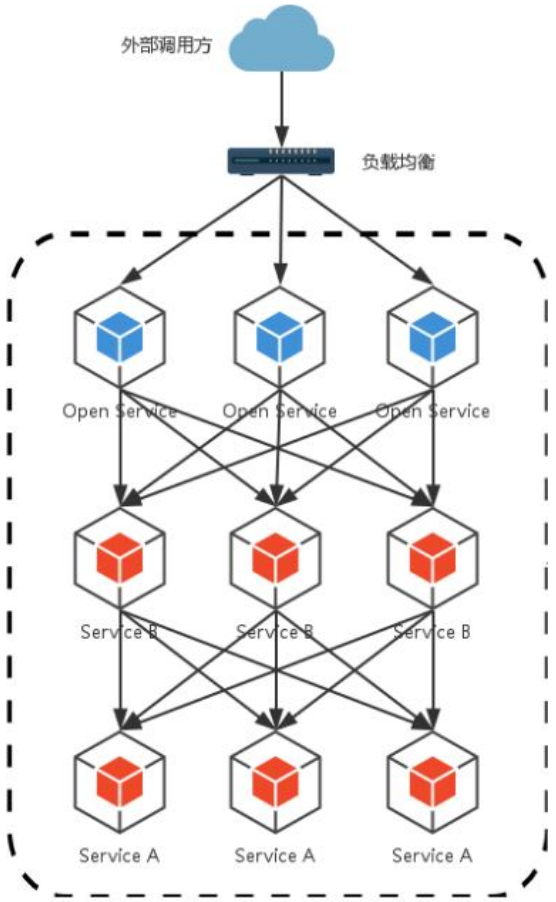
- 微服务是一种架构设计模式。在微服务架构中，业务逻辑被拆分成一系列小而松散耦合的分布式组件，共同构成了较大的应用。每个组件都被称为微服务。
- 每个微服务都在整体架构中执行着单独的任务，或负责单独的功能。
- 每个微服务可能被一个或多个其他微服务调用，以执行较大应用需要完成的具体任务。

1.1 什么是微服务

- 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，它们可以用不同的语言编写，并使用不同的数据存储技术。

1.1 什么是微服务

- 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
  - 一支小团队能维护几个微服务
  - 可替换的



1.1 什么是微服务

- 通过分解巨大单体应用为多个服务的方法解决了系统复杂性问题。在功能不变的情况下，应用被分解为多个可管理的分支或服务。每个服务都有一个用RPC或者消息驱动API定义清楚的边界。
- 微服务架构使得每个服务都可以有专门开发团队来开发，开发者可以自由选择开发技术，提供 API 服务。
- 微服务架构模式使得每个微服务独立部署，开发者不再需要协调其它服务部署对本服务的影响。
- 微服务架构模式使得每个服务独立扩展，可以根据每个服务的规模来部署满足需求的实例。

1.1 什么是微服务

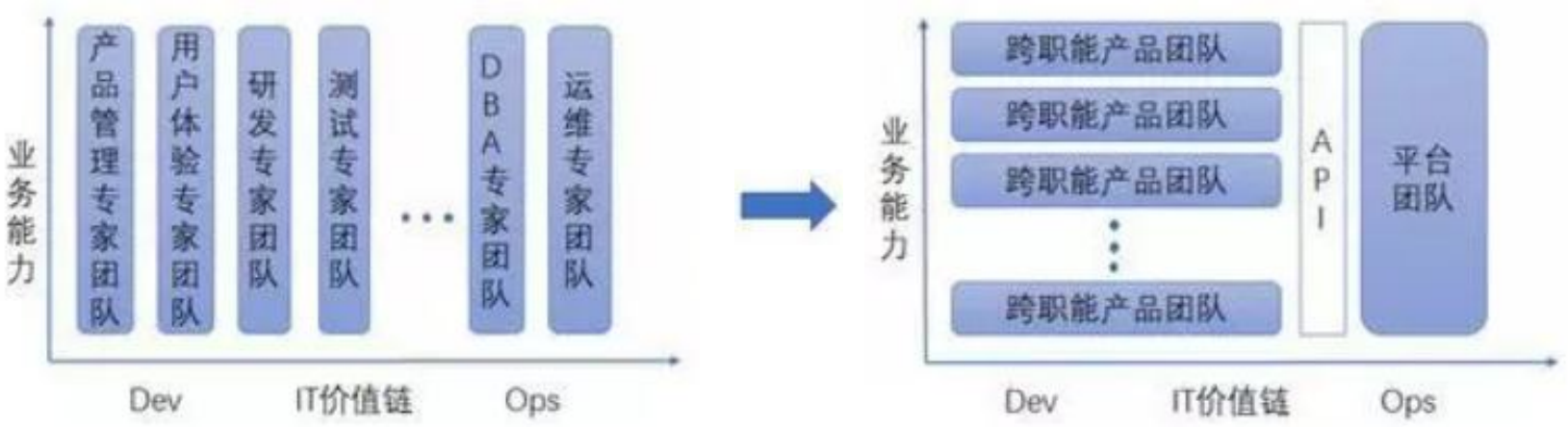
- 微服务强调了服务大小，实际开发者可能过度强调小，或者并不明确粒度的标准。所以应该秉持一个原则，即微服务的目的是有效的拆分应用，实现敏捷开发和部署。
- 微服务应用是分布式系统，必然带来固有的复杂性。开发者需要在 **RPC** 或者消息传递之间选择并完成进程间通讯机制。此外，他们必须写代码来处理消息传递中速度过慢或者不可用等局部失效问题。
- 分区的数据库架构也会为微服务带来挑战。同时更新多个业务主体的事务很普遍。在微服务架构应用中，需要更新不同服务所使用的不同的数据库。

1.1 什么是微服务

- 测试一个基于微服务架构的应用也是很复杂的任务，如面向多业务协同的单元测试，以及集成测试的规划。
- 微服务的自治管理较为复杂，一个单体应用只需要在复杂均衡器后面部署各自的服务器，但微服务中每个服务应用实例是需要配置诸如数据库和消息中间件等基础服务。每个服务都有多个实例，这就形成大量需要配置、部署、扩展和监控的部分。
- 微服务还需要一个服务发现机制，以用来发现与它通讯服务的地址（包括服务器地址和端口）。

1.1 什么是微服务

- 微服务组织架构
  - 从传统职能型到跨职能产品型



1.1 什么是微服务

- 微服务架构与中台战略
  - 将传统前后台体系中的后台进行细分，强化业务和技术中台，把前端应用变得更小更灵活。

1.2 微服务与SOA

- SOA：是一种设计方法，服务之间通过相互依赖最终提供一系列的功能。一个服务通常以独立的形式存在与操作系统进程中。各个服务之间通过网络调用。
- 微服务架构：其实和 SOA 架构类似，微服务是在 SOA 上做的升华，微服务架构强调的一个重点是“业务需要彻底的组件化和服务化”，原有的单个业务系统会拆分为多个可以独立开发、设计、运行的小应用。这些小应用之间通过服务完成交互和集成。
- 微服务架构 = 80%的SOA服务架构思想 + 100%的组件化架构思想 + 80%的领域建模思想

1.2 微服务与SOA

- **SOA的关注点**
  - 系统集成：从系统角度，解决企业信息系统间的通信问题，把原先散乱、无规划的系统间网状结构，梳理成有规律的、可治理的系统间星形结构，并引入合理的中间件以及规范标准进行管理，如 ESB、技术规范、服务管理规范等。核心目标-有序化。
  - 系统的服务化：从功能的角度，把业务逻辑抽象成可复用、可组装的服务，通过服务的编排实现业务的快速重构。把遗存固有的系统功能转变为通用的业务服务，实现业务逻辑的快速复用。核心目标-复用化。
  - 业务的服务化：从企业的角度，把企业职能抽象成可复用、可组装的服务。把原先职能化的企业架构转变为服务化的企业架构，进一步提升企业的对外服务能力。核心目标-高效化。

1.2 微服务与SOA

- 微服务的关注点
  - 去中心化
    - 每个微服务有自己私有的数据库持久化业务数据。
    - 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。
    - 数据的去中心化，进一步降低了微服务之间的耦合度，不同服务可采用不同数据库技术（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。

1.2 微服务与SOA

- 服务组件化
  - 开发者不再需要协调其它服务部署对本服务的影响。
  - 按业务能力来划分服务和开发团队开发者可以自由选择开发技术，提供 API 服务。
- 基础设施自动化（DevOps、自动化部署）
  - 传统的Java EE部署架构，通过展现层打包WARs，业务层划分到JARs最后部署为EAR一个大包，而微服务把应用拆分成为单个独立微服务，应用Docker技术，不依赖任何服务器和数据模型。
  - 每个服务运行在自己的进程中，通过轻量的通讯机制联系，这些服务基于业务能力构建，能实现集中化管理。

1.3 微服务设计原则

1. 围绕业务概念建模

- 限界上下文（**Boundary Context**）：一个由显式边界限定的特定职责。每个上下文都有明确的接口，该接口定义了它会暴露哪些模型给其他的上下文。
- 围绕业务的限界上下文定义的接口，比围绕技术概念定义的接口更加稳定。
- 松耦合和高内聚：松耦合的服务应该尽可能少地知道与之协作的那些服务的信息；高内聚则是要求把相关的行为聚集在一起，把不相关的行为放在别处。

1.3 微服务设计原则

2. 接受自动化文化

- 微服务引入了很多复杂性，其中的关键部分是，我们不得不管理大量的服务。引入持续集成、持续交付这些元素势在必行。自动化测试也必不可少。
- 通过调用统一的命令，以相同的方式把系统部署到各个环境是一个很有用的实践，也是采用持续交付对每次提交后的产品质量进行快速反馈的一个关键部分。
- 通过把配置都存到版本控制中，我们可以自动化重建服务，甚至重建整个环境。

1.3 微服务设计原则

3. 隐藏内部实现细节

- 为了使一个服务独立于其他服务，最大化独立演化的能力，隐藏实现细节至关重要。
- 服务应该隐藏它们的数据库，以免陷入数据库耦合。
- 对于服务间的交互，应尽量选用与技术无关的 API，从而让你能在不同的服务内自由选择不同的技术栈。

1.3 微服务设计原则

4. 让一切去中心化

- 类似企业服务总线或服务编配系统的方案，会导致业务逻辑的中心化，应该避免使用它们。使用协同来代替编排，在服务限界内能保持服务的内聚性。
- 构建微服务不仅要从技术的角度去思考，更得从团队的角度入手。为了最大化微服务能带来的自治性，我们需要持续寻找机会，给拥有服务的团队委派决策和控制权。

1.3 微服务设计原则

5. 可独立部署

- 应始终努力确保微服务可独立部署。这会使得无论微服务本身，还是团队，都越来越具有自治性。
- 蓝/绿部署：部署两份软件，但只有一个接受真正的请求。在新版本部署之后，首先对其运行一些测试，等测试没有问题后，再切换生产负荷到新版本上。
- 金丝雀发布：通过将部分生产流量引流到新部署的系统，来验证系统是否按预期执行。新旧版本共存的时间更长，而且经常会调整流量。

1.3 微服务设计原则

6. 隔离失败

- 超时：应给所有的跨进程调用设置超时，并选择一个默认的超时时间。当超时发生后，记录到日志里看看发生了什么，并相应地调整它们。
- 断路器：当对一个下游资源的请求发生一定数量的失败后，应采取一种“断路”机制，对接下来继续涌入的请求采取快速失败的做法，以免整个系统崩溃。
- 舱壁：把自己从故障中隔离开的一种方式。

1.3 微服务设计原则

7. 高度可观察

- 通过注入合成事务到系统中，模拟真实用户的行为，从而使用语义监控来查看系统是否运行正常。
- 聚合日志和数据，这样当遇到问题时，就可以深入分析原因。

1.3 微服务设计原则

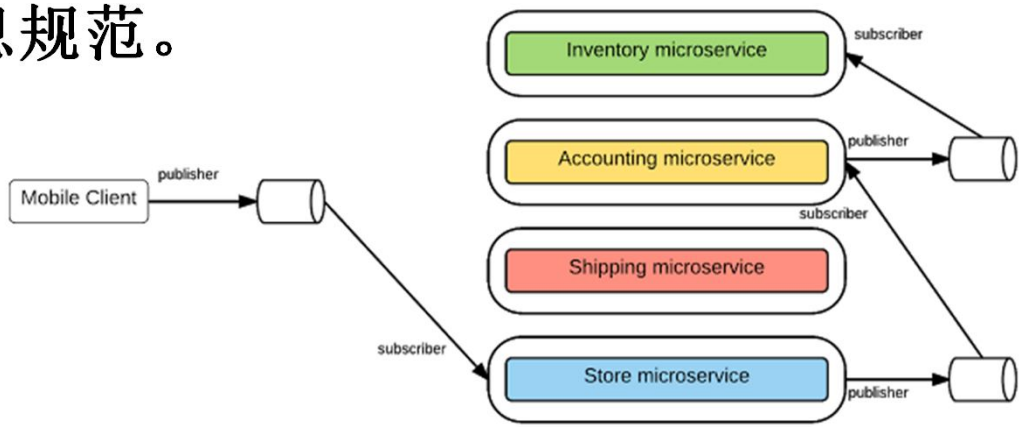
- 微服务架构案例

1.4 微服务架构的消息机制

- 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，这时尽量不采用点对点的集成方式。

1.4 微服务架构的消息机制

- 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。消费者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。



1.4 微服务架构的消息机制

- 同步消息 – REST、Thrift
  - 同步消息指客户端需要保持等待直到服务器返回应答。REST是微服务中默认的同步消息方式，它提供了基于HTTP协议和资源API风格的简单消息格式，多数微服务都采用这种方式。
  - Thrift机制采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中创建高效的、无缝的服务，传输数据采用二进制格式，相对XML和JSON体积更小。对于高并发、大数据量和多语言的环境更有优势。

1.4 微服务架构的消息机制

- 异步消息 – AMQP、STOMP、MQTT
  - 异步消息指客户端不需要一直等待服务应答，有应到后会得到通知。一般采用AMQP、STOMP、MQTT。
  - AMQP：一个提供统一消息服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。
  - STOMP：一个简单的可互操作的协议, 被用于通过中间服务器在客户端之间进行异步消息传递。
  - MQTT：IBM开发的一个即时通讯协议，可能成为物联网的重要组成部分。

1.4 微服务架构的消息机制

- 消息格式 – JSON、XML、Thrift等
  - 消息格式是微服务中另外一个很重要的因素。SOA的web服务一般采用文本消息，基于复杂的消息格式SOAP和消息定义（xsd）。微服务采用简单的文本协议JSON和XML，基于HTTP的资源API风格。
  - 如果需要二进制，通过用到Thrift、ProtoBuf、Avro。

1.4 微服务架构的消息机制

- 服务定义接口 – Swagger、RAML、Thrift IDL
  - 如果把功能实现为服务并发布，需要定义一套约定。单体架构中，SOA采用WSDL，WSDL过于复杂并且和SOAP紧耦合，不适合微服务。
  - REST设计的微服务，通常采用Swagger、RAML、YAML定义约定。
  - 对于不是基于REST设计的微服务，比如Thrift，通常采用IDL（Interface Definition Languages），比如Thrift IDL。

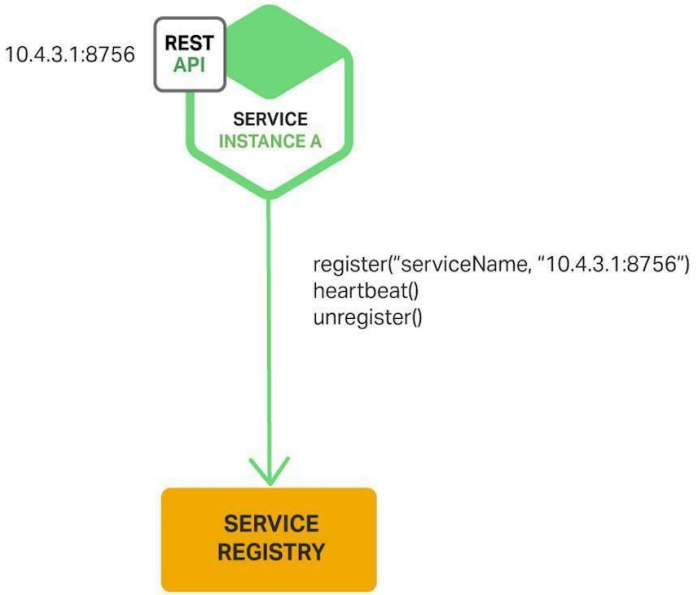
1.5 微服务架构的服务集成

- 微服务框架的服务注册
  - 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。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。
    - 管理服务实例注册的其它组件，即第三方注册模式。

1.5 微服务架构的服务集成

自注册方式

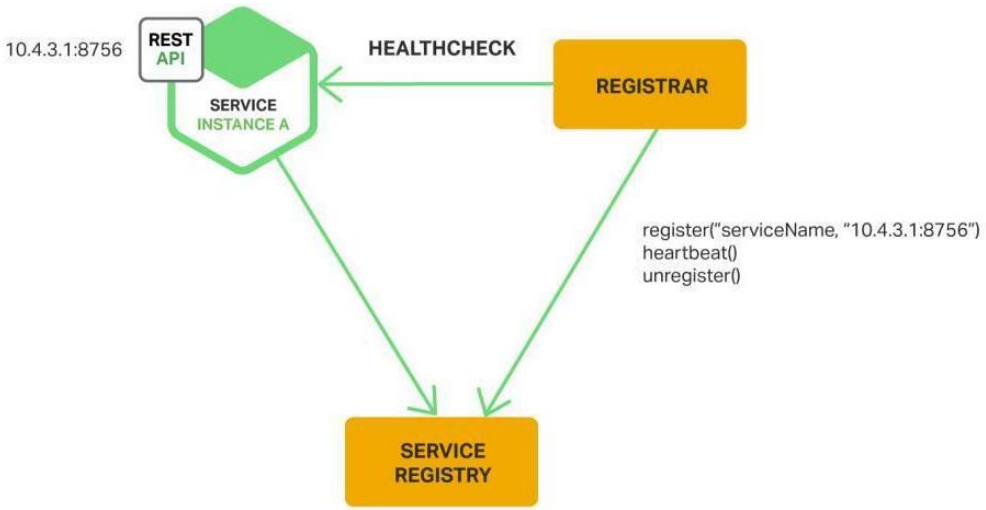
- 使用自注册模式时，服务实例负责在服务注册表中注册和注销。另外，如果需要的话，一个服务实例也要发送心跳来保证注册信息不会过时。
- 自注册模式相对简单，无需其它系统组件。然而它的主要缺点是把服务实例和服务注册表耦合，必须在每个编程语言和框架内实现注册代码。



1.5 微服务架构的服务集成

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。一旦侦测到新的服务实例，会向注册表注册此服务。



1.5 微服务架构的服务集成

- 第三方注册模式中，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。

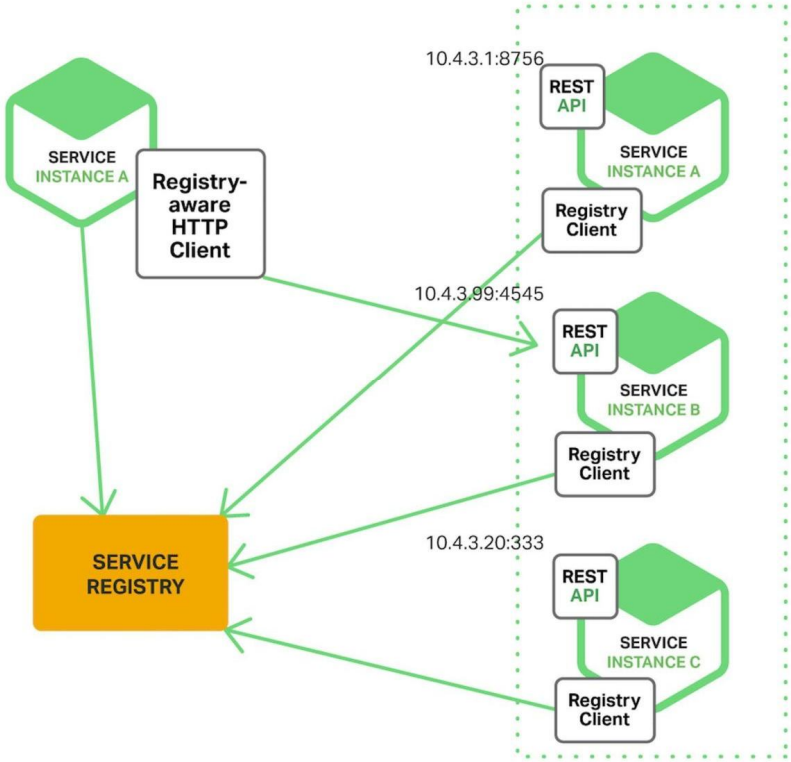
1.5 微服务架构的服务集成

- SOA体系下，服务之间通过企业服务总线ESB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。

1.5 微服务架构的服务集成

微服务架构的服务发现

- 客户端发现模式
- 客户端决定相应服务实例的网络位置，并对请求实现负载均衡。客户端查询服务注册表，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。



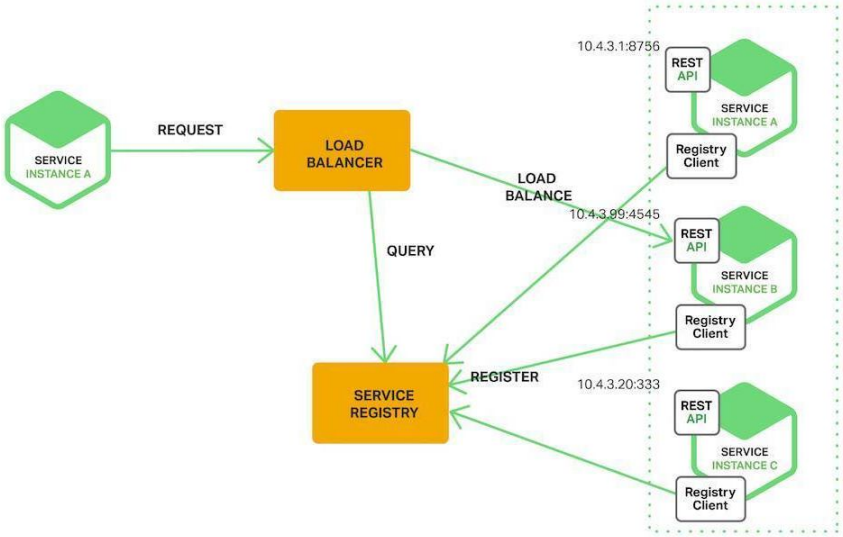
1.5 微服务架构的服务集成

- 客户端发现模式
- 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。
- 客户端发现模式的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。

1.5 微服务架构的服务集成

微服务架构的服务发现

- 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。



1.5 微服务架构的服务集成

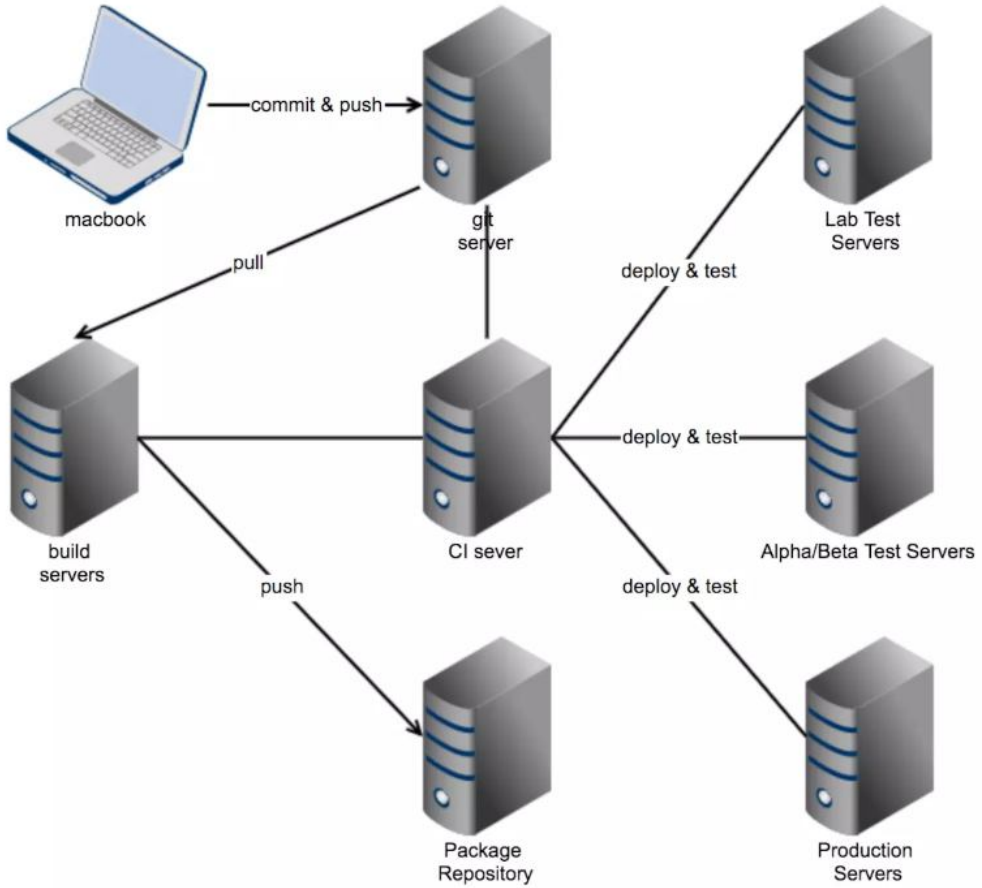
- 服务端发现模式
- **AWS Elastic Load Balancer(ELB)是服务端发现路由的例子，ELB 通常均衡来自互联网的外部流量，也可用来负载均衡 VPC（ Virtual private cloud）的内部流量。客户端使用 DNS 通过 ELB 发出请求（HTTP 或 TCP），ELB 在已注册的 EC2 实例或 ECS 容器之间负载均衡。**
- 服务端发现模式最大的优点是客户端无需关注发现的细节，只需要简单地向负载均衡器发送请求，这减少了编程语言框架需要完成的发现逻辑。并且有些部署环境免费提供这一功能。
- 服务端发现模式的缺点是除非负载均衡器由部署环境提供，否则会成为一个需要配置和管理的高可用系统组件。

1.6 微服务与持续集成

- 持续集成（CI）是一种开发实践，是指开发人员每天多次将代码整合到共享的代码库中，然后通过自动化构建验证每次提交，使团队能够及早发现问题。
- 持续续交付（CD）是指通过自动化在比较短的一个个迭代中快速发布软件的做法，允许团队更频繁地交付可以工作的软件产品。 其重点——持续整合，内置测试，持续监控和分析反馈都指向软件行业的整体趋势：提高应变能力。

1.6 微服务与持续集成

- 持续集成环境

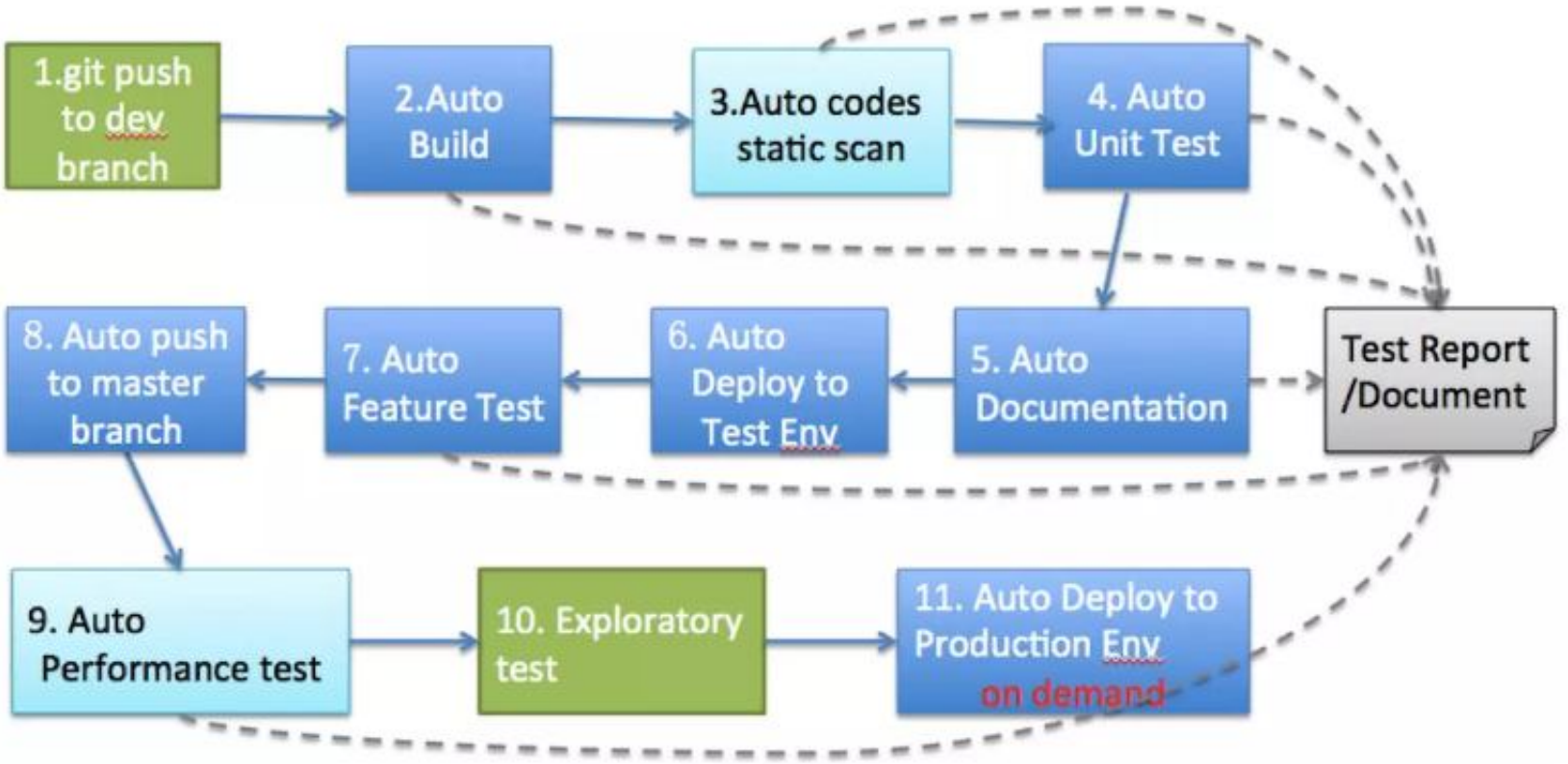


1.6 微服务与持续集成

- 持续集成的最佳实践
  - 维护单一代码库
  - 自动化构建
  - 使构建能够自我测试
  - 保持构建速度
  - 在生产环境相同的产品环境中测试
  - 任何人都要以轻松获得最新的可执行文件
  - 每个人都可以看到发生了什么
  - 自动部署

1.6 微服务与持续集成

- 持续交付流程



1.6 微服务与持续集成

- 持续交付流程
  1. 提交代码：在本地构建并通过单元测试之后，可以将代码提交到代码仓库的开发分支中。
  2. 自动构建：包括编译，链接，打包，在一些大型项目中这个速度往往需要几十分钟以上。
  3. 自动代码检查：各种语言都有各自的代码静态检查工具，检查出严重的错误则立即终止构建流水线。
  4. 自动执行单元测试：确保这次提交不会造成其他模块的单元测试失败，任何一条单元测试不通过，都会终止构建流水线。

1.6 微服务与持续集成

5. 自动生成文档：利用自动文档生成工具，分析代码并生成 API 和帮助文档。
6. 自动部署在测试环境：利用部署脚本将软件部署到测试环境，注入预设的配置变量并启动服务。
7. 自动执行功能测试：在测试环境中，自动运行所有需要依赖外部环境的 API 和 E2E(端到端)的测试。
8. 将代码合并入主分支：由产品负责人决定是否合并到主分支。
9. 自动运行性能测试：生成性能测试和剖析报告，包括吞吐量、最耗时模块、以及对系统资源消耗的各种指标。
10. 探索性测试：手工进行一些探索性测试。

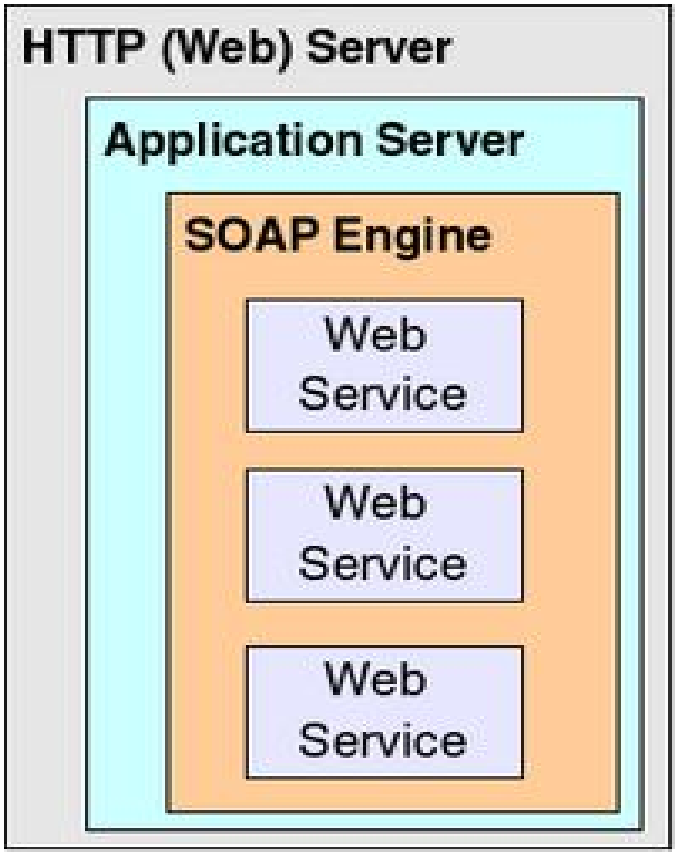
本章内容

1. 微服务基本理论
2. **Docker**
3. 调度工具

2.1 服务器运行环境

HTTP Server

- 通常叫做Web服务器“Web Server”
- 用于处理HTTP消息，处理HTML文件
- WEB服务器与客户端打交道，它要处理的主要信息有：session、request、response、HTML、JS、CS等
- 常用的Http Server有：Apache HTTP Server、IIS



2.1 服务器运行环境

- **Application server**
  - 为不同的应用提供“生存环境”
  - 允许不同的用户进行应用请求连接
  - 用于处理非常规性WEB页面（JSP文件），他动态生成WEB页面，生成的WEB页面在发送给客户端
  - 常用的Application Server有： Weblogic、Tomcat、Jboss
  - 在服务计算中，服务处理引擎如，SOAP engine，就运行在应用服务器中

2.1 服务器运行环境

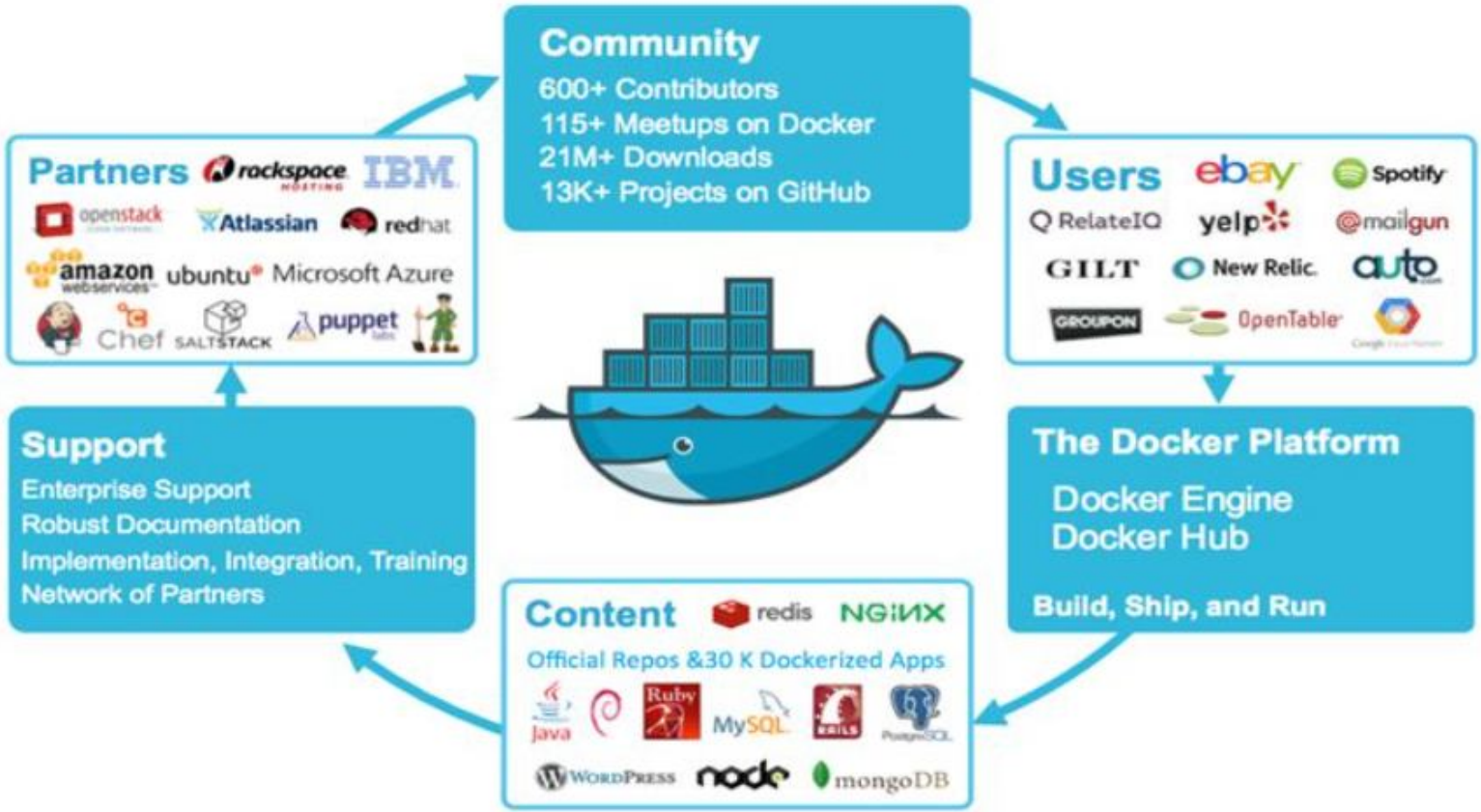
- 服务执行引擎（以SOAP Engine为例）
  - 负责处理SOAP的消息请求与消息回复
  - 为每一个服务生成一个服务桩以处理对应服务请求
  - 负责对SOAP数据包进行封装和解析
  - 常用的Http Server有：XFire，Apache Axis等

- Web Service
  - Web Service的实现并不管SOAP请求包的解析，以及SOAP回复包的生成

2.2 什么是Docker

- 环境配置的难题：从各种OS到各种中间件到各种APP，产品开发者需要关心的东西太多，且难于管理。
- 云计算时代的到来：开发者将应用转移到 云端，解决了硬件管理的问题，然而中间件相关的问题依然存在 。
- 虚拟化手段的变化
  - 云时代采用标配硬件来降低成本，采用虚拟化手段来满足用户按需使用的需求以及保证可用性和隔离性。
  - 然而无论是KVM还是Xen，在Docker看来，都在浪费资源，因为用户需要的是高效运行环境而非OS，GuestOS既浪费资源又难于管理，更加轻量级的LXC更加灵活和快速。

2.2 什么是Docker

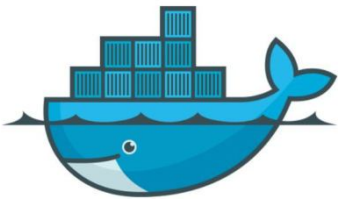


2.2 什么是Docker

- Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的Linux机器上，也可以实现虚拟化，容器是完全使用沙箱机制，相互之间不会有任何接口。
- Docker 是 PaaS 提供商 dotCloud 开源的一个基于 LXC（Linux Container）的高级容器引擎，源代码托管在 Github 上，基于go语言并遵从Apache2.0协议开源。

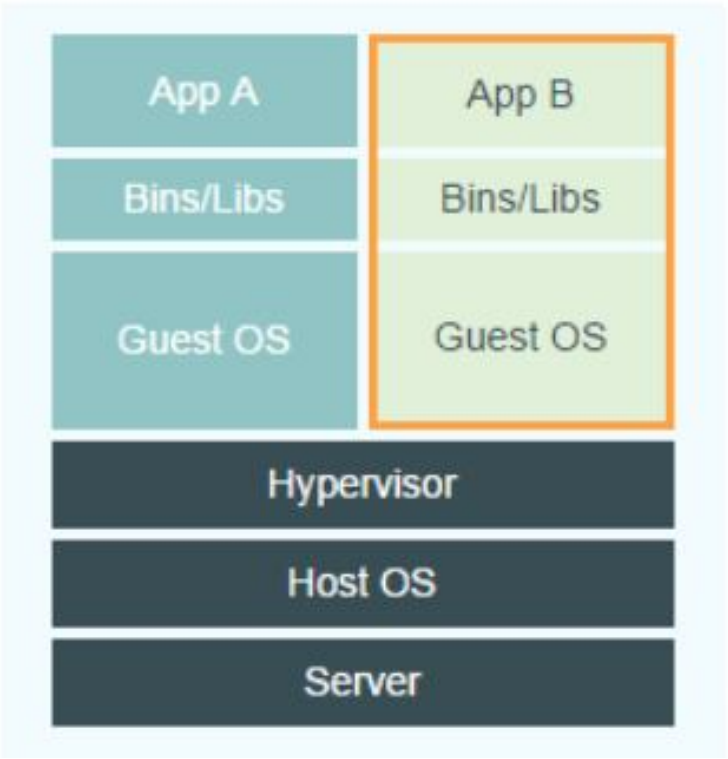
2.2 什么是Docker

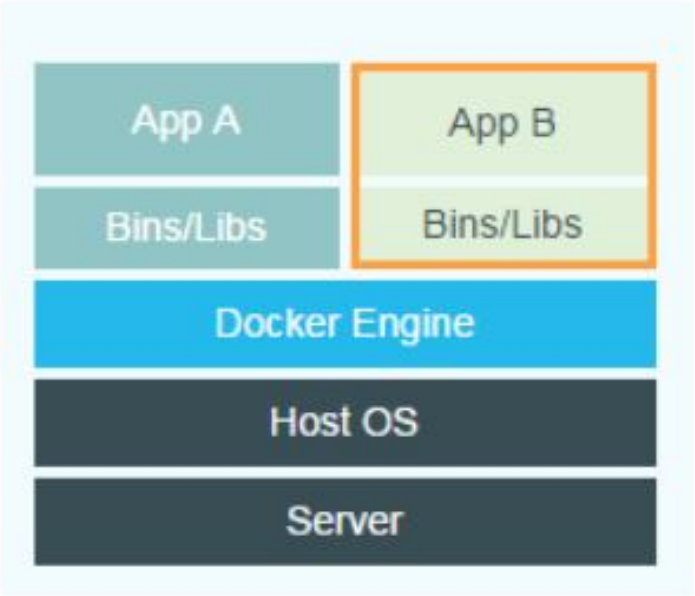
- **LXC（LinuX Containers）**
  - 一种操作系统层虚拟化技术，将应用软件系统打包成一个软件容器，内含应用软件本身的代码以及所需的操作系统核心和库，创造出应用程序的独立沙箱运行环境。
- **LXC的移动性**
  - **LXC在 linux 2.6 的 kernel 里就已经存在，但其设计之初并非为云计算考虑的，缺少标准化的描述手段和容器的可迁移性，构建出的环境难于迁移和标准化管理。**
  - **Docker 就在这个问题上做出实质性的革新。**



2.2 什么是Docker

- Docker与VM





2.2 什么是Docker

- 虚拟机的Guest OS即为虚拟机安装的操作系统，是一个完整操作系统内核；虚拟机的Hypervisor层可以理解为硬件虚拟化平台，它在Host OS是以内核态的驱动存在的。
- 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，
- 在IO设备虚拟化上，Docker的镜像管理有多种方案，比如利用Aufs文件系统或者Device Mapper实现Docker的文件管理，各种实现方案的效率略有不同。

2.2 什么是Docker

- Docker利用的是宿主机的内核，而不需要Guest OS。因此，当新建一个容器时，Docker不需要和虚拟机一样重新加载一个操作系统内核。
- 引导、加载操作系统内核是一个比较费时费资源的过程，新建一个虚拟机需要加载Guest OS，这个新建过程是分钟级别的。Docker由于直接利用宿主机的操作系统，则省略了这个过程，因此新建一个容器只需要几秒钟。
- 现代操作系统是复杂的系统，在一台物理机上新增加一个操作系统的资源开销是比较大的，因此，Docker对比虚拟机在资源消耗上也占有比较大的优势。

2.3 Docker的特点

- 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团队实现快速交付。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。
  - 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。可以快速扩容或方便的下线应用和服务，这种速度趋近实时。

2.3 Docker的特点

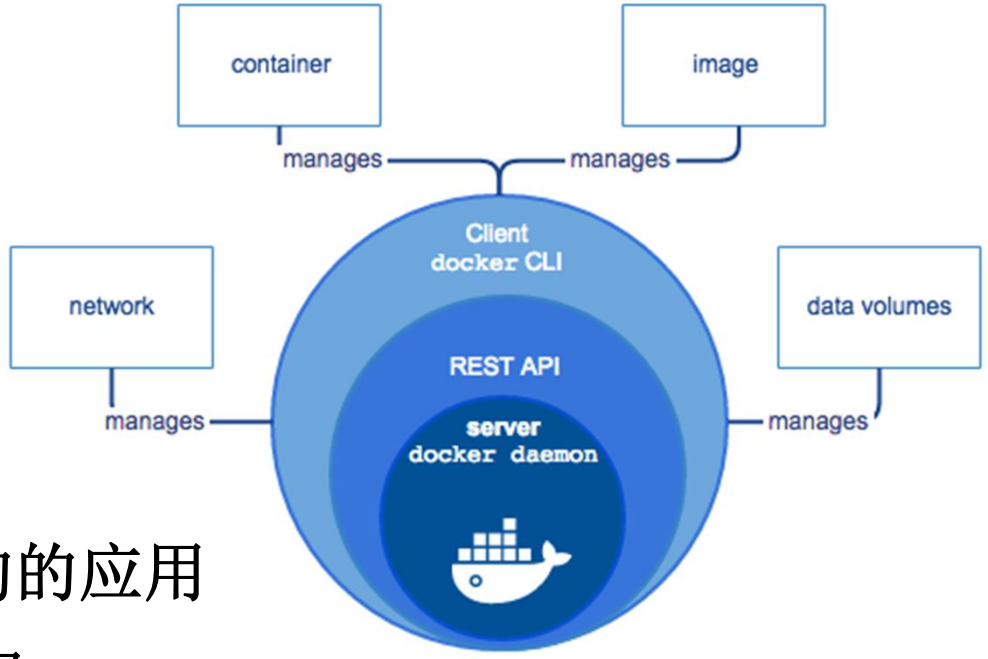
- 更高的资源利用率
  - Docker 对系统资源的利用率很高，一台主机上可以同时运行数千个 Docker 容器。
  - 容器除了运行其中的应用外，基本不消耗额外的系统资源，使得应用的性能很高，同时系统的开销尽量小。
- 更简单的管理
  - 使用 Docker，只需要小小的修改，就可以替代以往大量的更新工作。所有的修改都以增量的方式被分发和更新，从而实现自动化并且高效的管理。

2.3 Docker的特点

- 相对于虚拟机，Docker还存在着以下几个缺点：
  - 资源隔离不如虚拟机，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，存在一定的安全风险。
  - Docker还在版本的快速更新中，细节调整比较大。一些核心模块依赖于高版本内核，存在版本兼容问题。

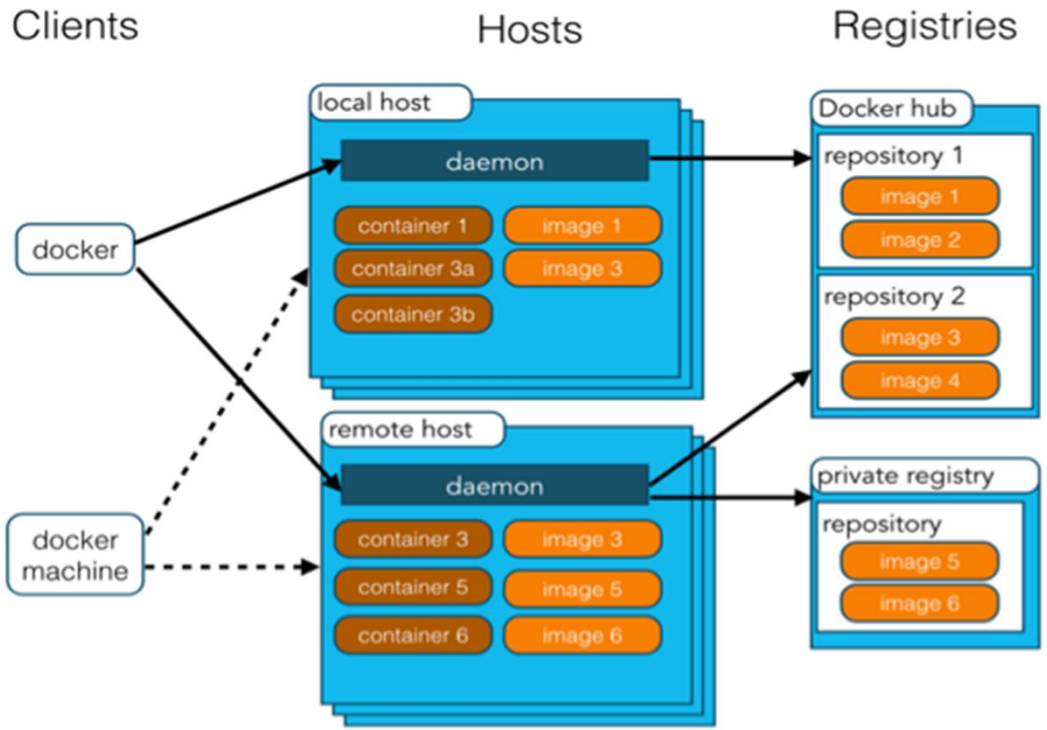
2.4 Docker架构

Docker引擎



- Docker引擎是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，为用户提供统一操作界面

2.4 Docker架构



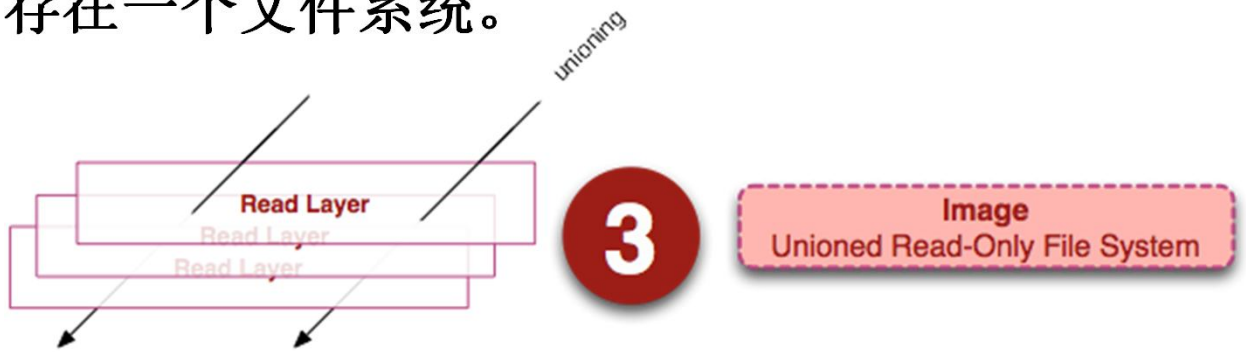
- Docker使用C/S架构，Client 通过接口与Server进程通信实现容器的构建、运行和发布。Client和Server可以运行在同一台主机，也可以通过跨主机实现远程通信。

2.4 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。

2.4 Docker架构

- 镜像就是一堆只读层（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。



2.4 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下载。国内的公开仓库包括时速云 、网易云等，可以提供更稳定快速的访问。
  - 用户也可以在本地网络内创建一个私有仓库。

2.4 Docker架构

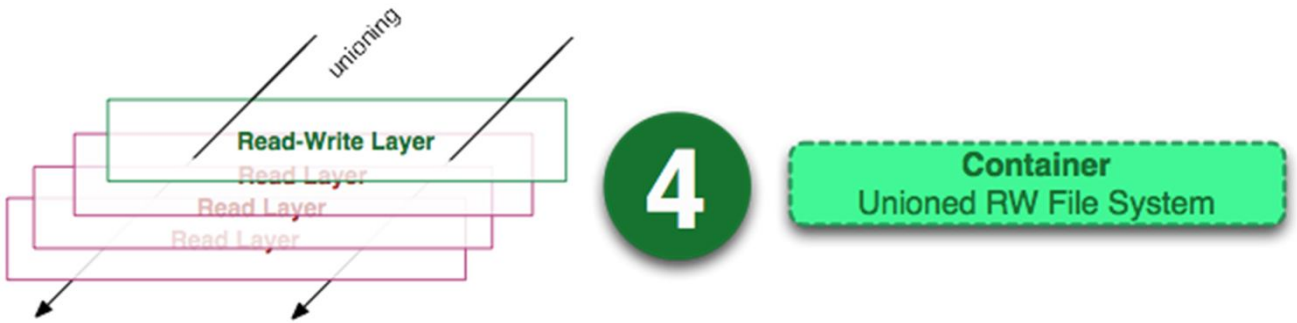
- 仓库（Repository）
  - 当用户创建了自己的镜像之后就可以使用 push 命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时候，只需要从仓库上 pull 下来就可以了。
  - Docker 仓库的概念跟 Git 类似，注册服务器可以理解为 GitHub 这样的托管服务。

2.4 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。
  - 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。

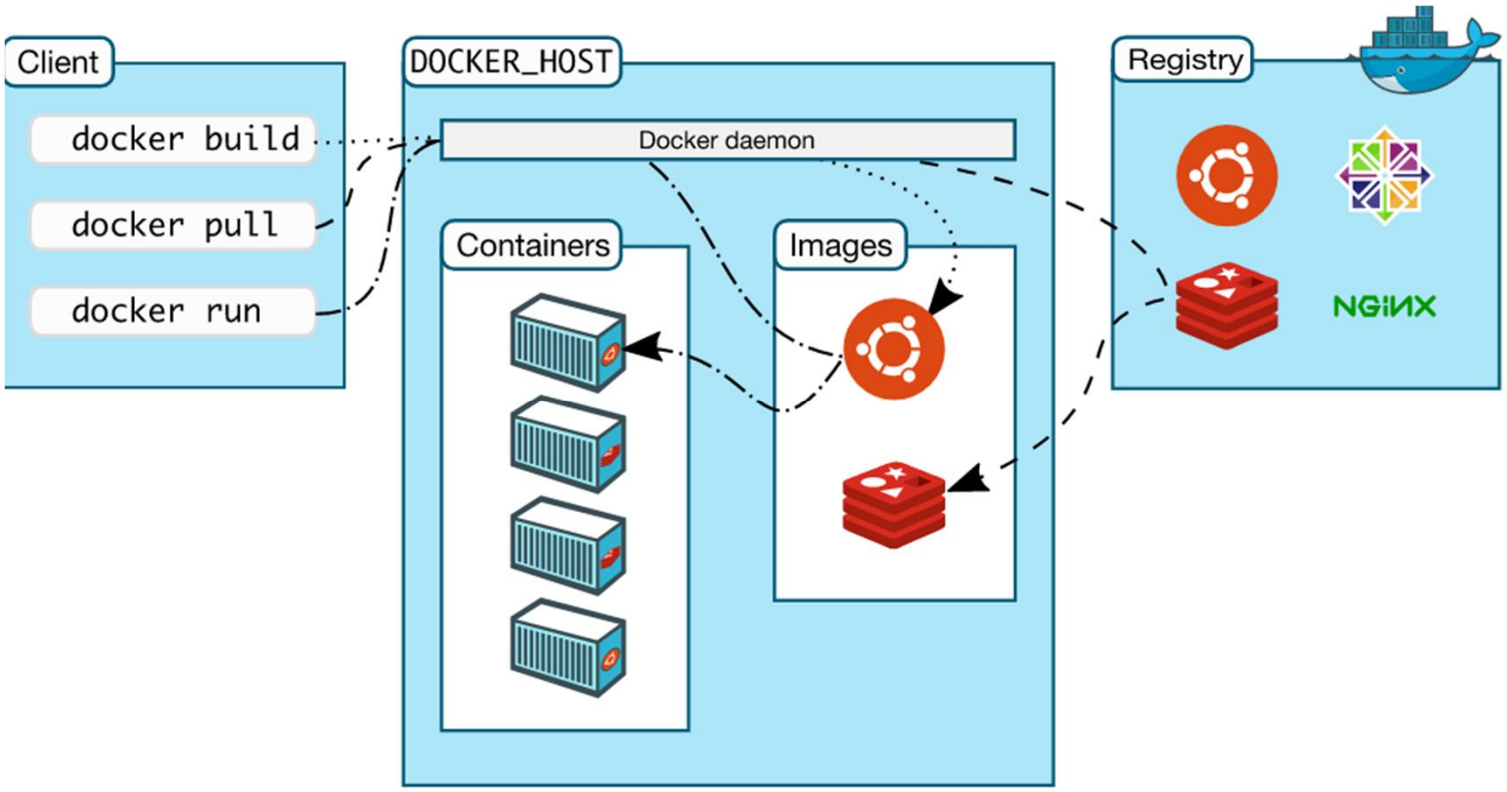
2.4 Docker架构

- 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。
- 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。



2.4 Docker架构

- Docker客户端与Docker服务器进行交互，Docker服务端负责构建、运行和分发 Docker 镜像。



本章内容

1. 微服务基本理论
2. Docker
3. 调度工具

3.1 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。
- 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。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，能够扩展容器来解决问题。

3.1 Docker调度工具

- 主流容器调度工具
  - Docker开发了Swarm，已被整合进 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。
- 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。

3.2 Kubernetes

Kubernetes指令示例

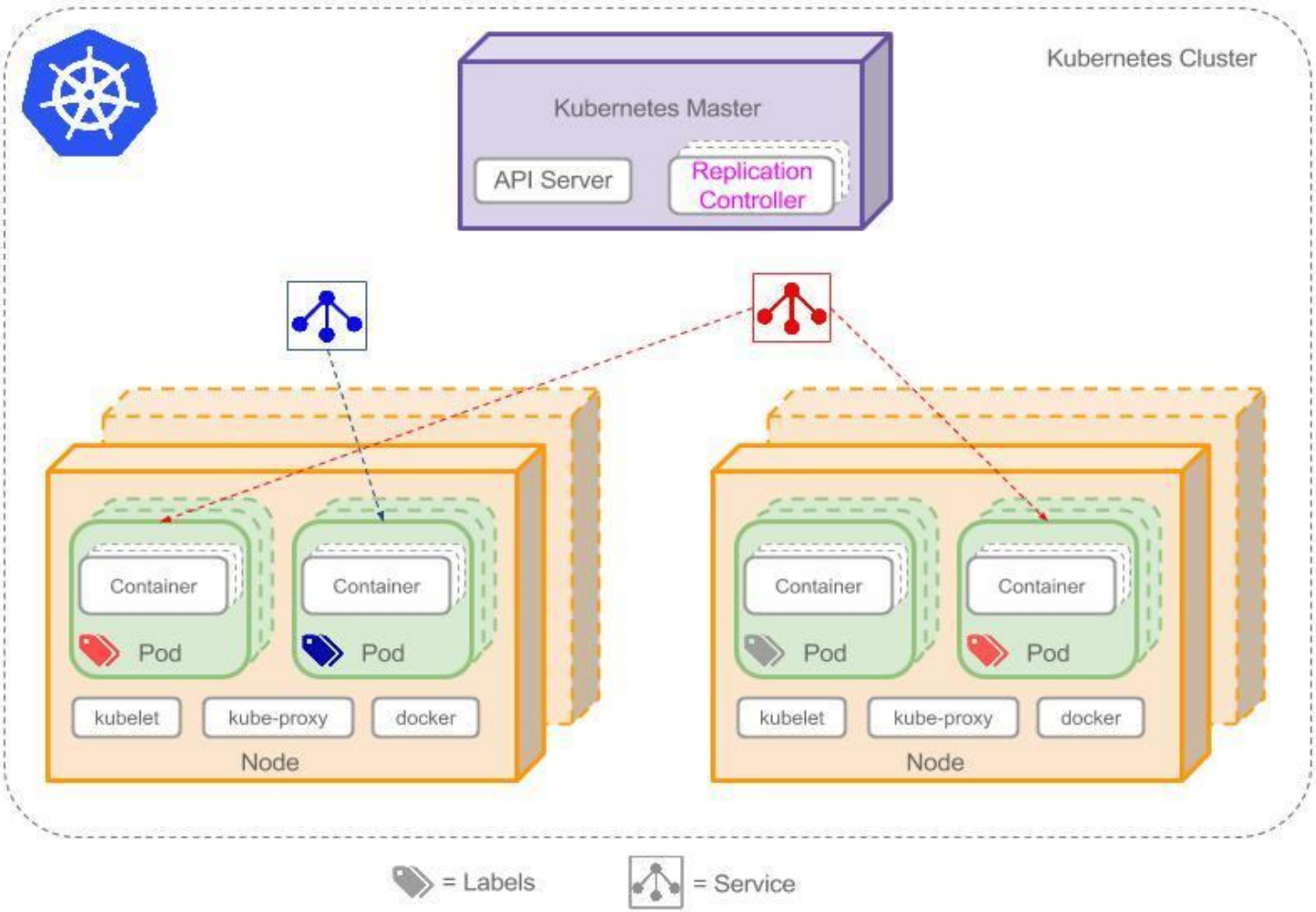
- 创建一个 Nginx Pod

```
[root@node1 ~]# kubectl run nginx-demo --image=nginx:1.14-alpine --port=80 --replicas=1
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in a future version. Use kubectl
deployment.apps/nginx-demo created
```

- 集群管理（扩容和缩减）

```
[root@node1 ~]# kubectl scale deployment nginx-demo --replicas=3
deployment.extensions/nginx-demo scaled
[root@node1 ~]# kubectl get deployment nginx-demo -o wide
NAME         READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES              SELECTOR
nginx-demo   3/3     3            3           61m   nginx-demo   nginx:1.14-alpine   run=nginx-demo
```

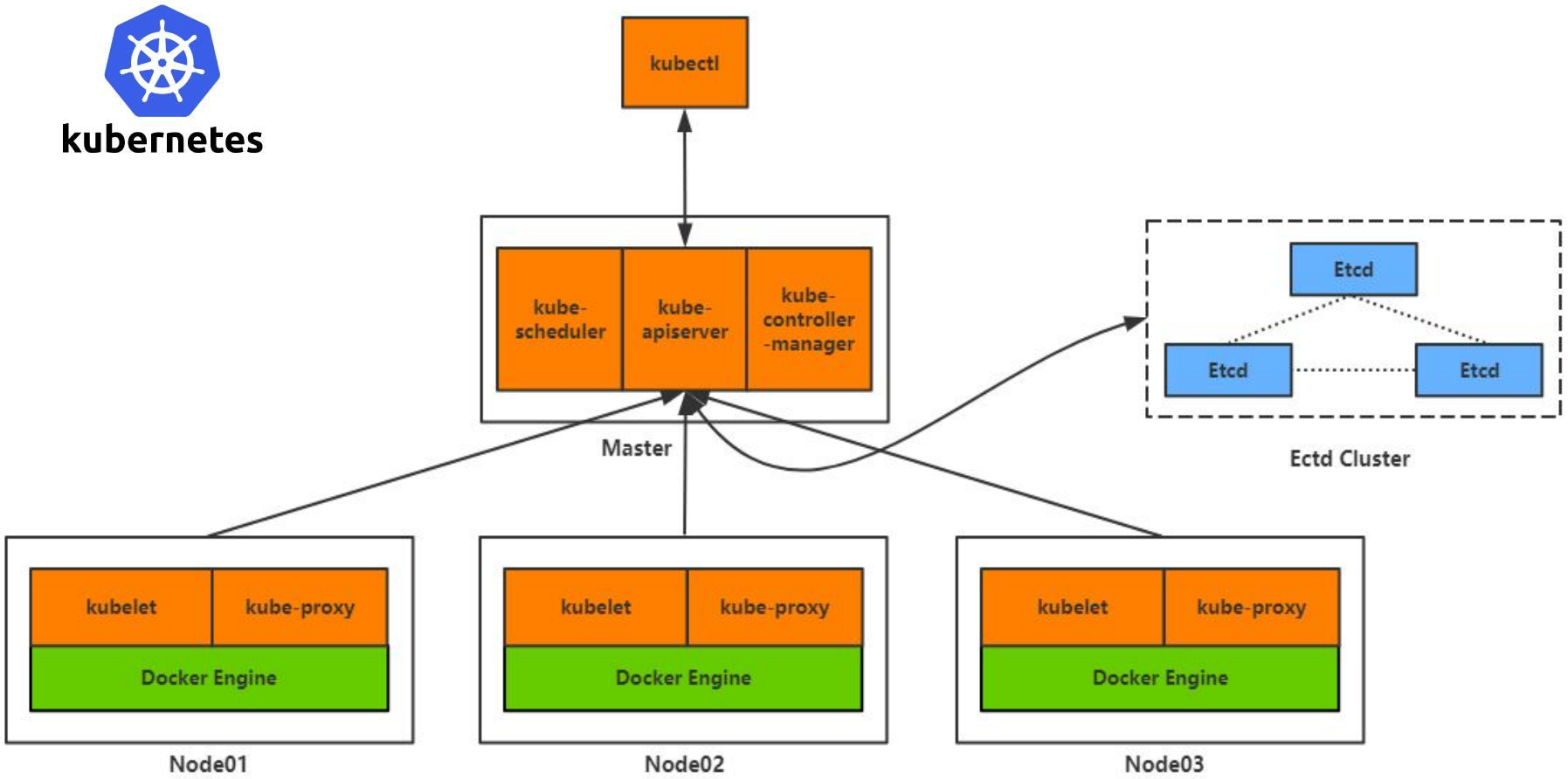
3.2 Kubernetes



3.2 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，让在一个相同 **Pod** 中的容器共享一个 **IP** 并使用 **localhost** 端口，允许所有 **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**，这些控制器对一个服务的中运行的容器进行管理合监控。

3.2 Kubernetes



3.2 Kubernetes

- Kubernetes使用label和pod的概念来将容器划分为逻辑单元。Pods是同地协作容器的集合，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调度器的任务是寻找那些PodSpec.NodeName为空的pods，然后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。

3.2 Kubernetes

- 谓词是强制性的规则，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，直到有机器能满足条件。
  - **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：有足够空间来满足pod和链接数据卷**
  - **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**

3.2 Kubernetes

- 优先级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。
  - **Priority**：寻找最佳节点
  - **LeastRequestdPriority**：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最优
  - **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。

哈尔滨工业大学 国家示范性软件学院

谢谢！