Chapter 5 微服务与应用技术

§5.1 微服务架构

§5.2 Serverless与FaaS

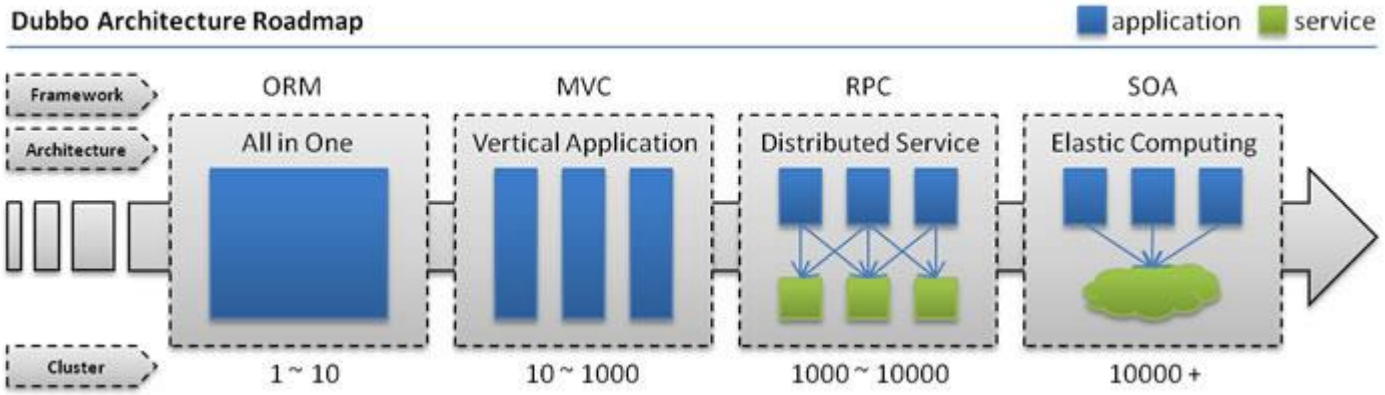
§5.3 云应用开发

§5.1 微服务

微服务架构 (Microservices Architecture)

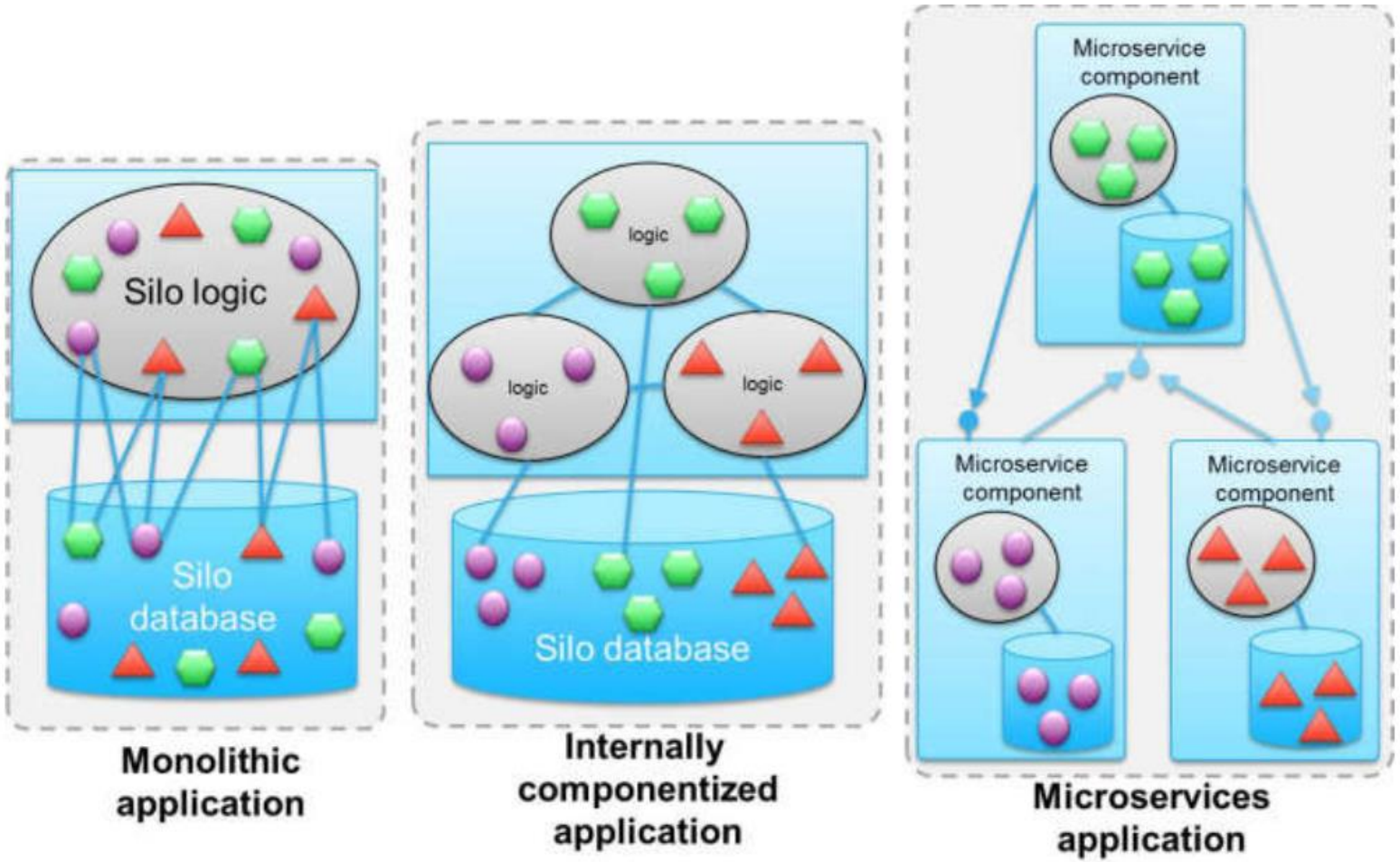
- 是一种软件架构风格
- 通过将传统的单块架构(Monolithic Architecture)的应用拆分成一组细粒度的微服务
- 每个微服务运行在独立的进程中，可以单独部署，采用不同的编程语言，通过语言无关的轻量级的通信机制进行相互调用，实现软件系统的松耦合。

互联网应用架构

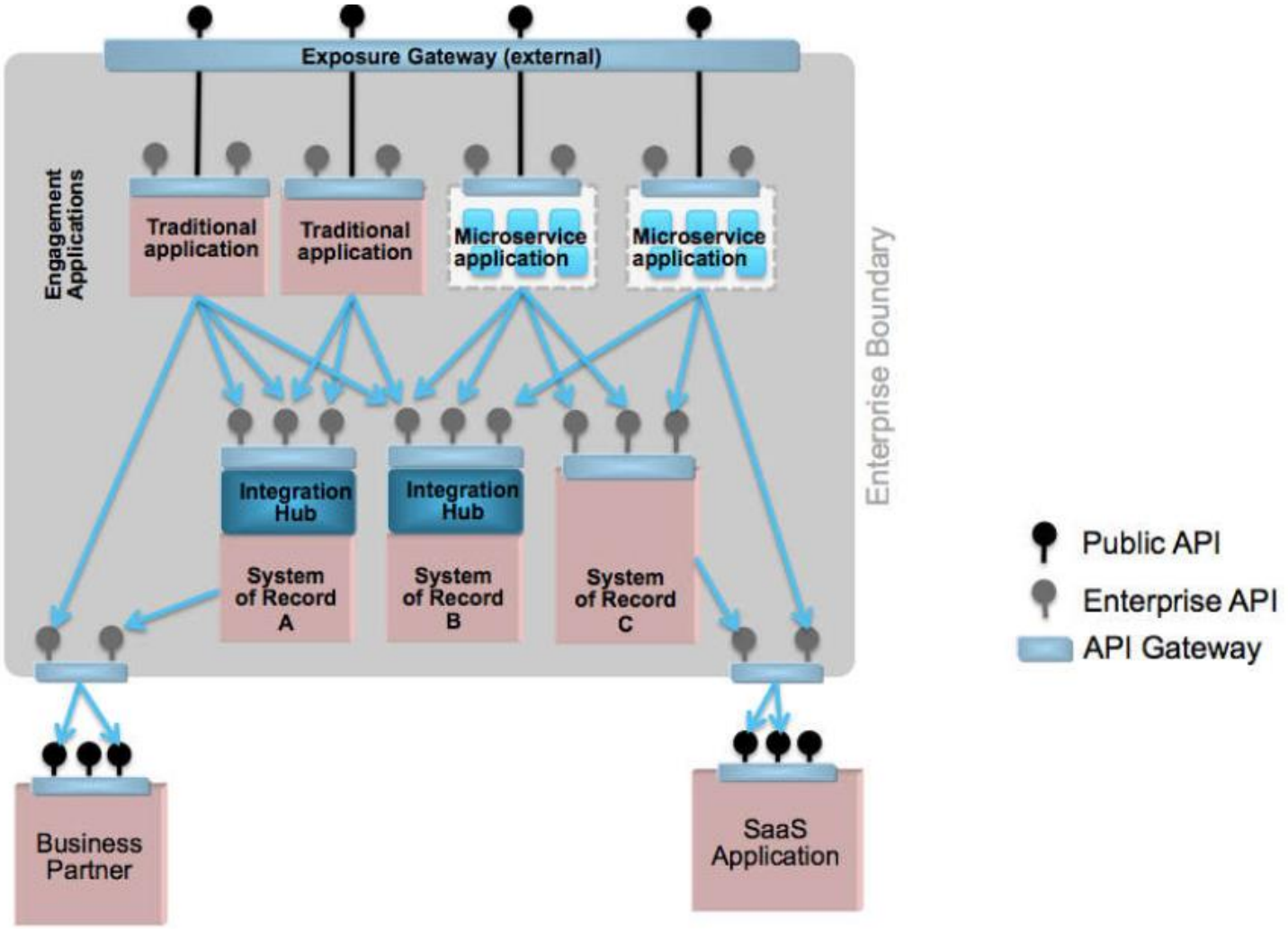


- 单一应用架构：所有功能部署在一起。
- 垂直应用架构：将应用拆分成互不相干的几个应用。
- 分布式服务架构：核心业务抽取出来，作为独立服务，形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。
  - RPC：用于提高业务复用及整合
- 弹性服务架构：服务越来越多，小服务资源浪费问题显现，需要增加调度中心实时管理集群容量。
  - SOA：资源调度和治理

微服务架构

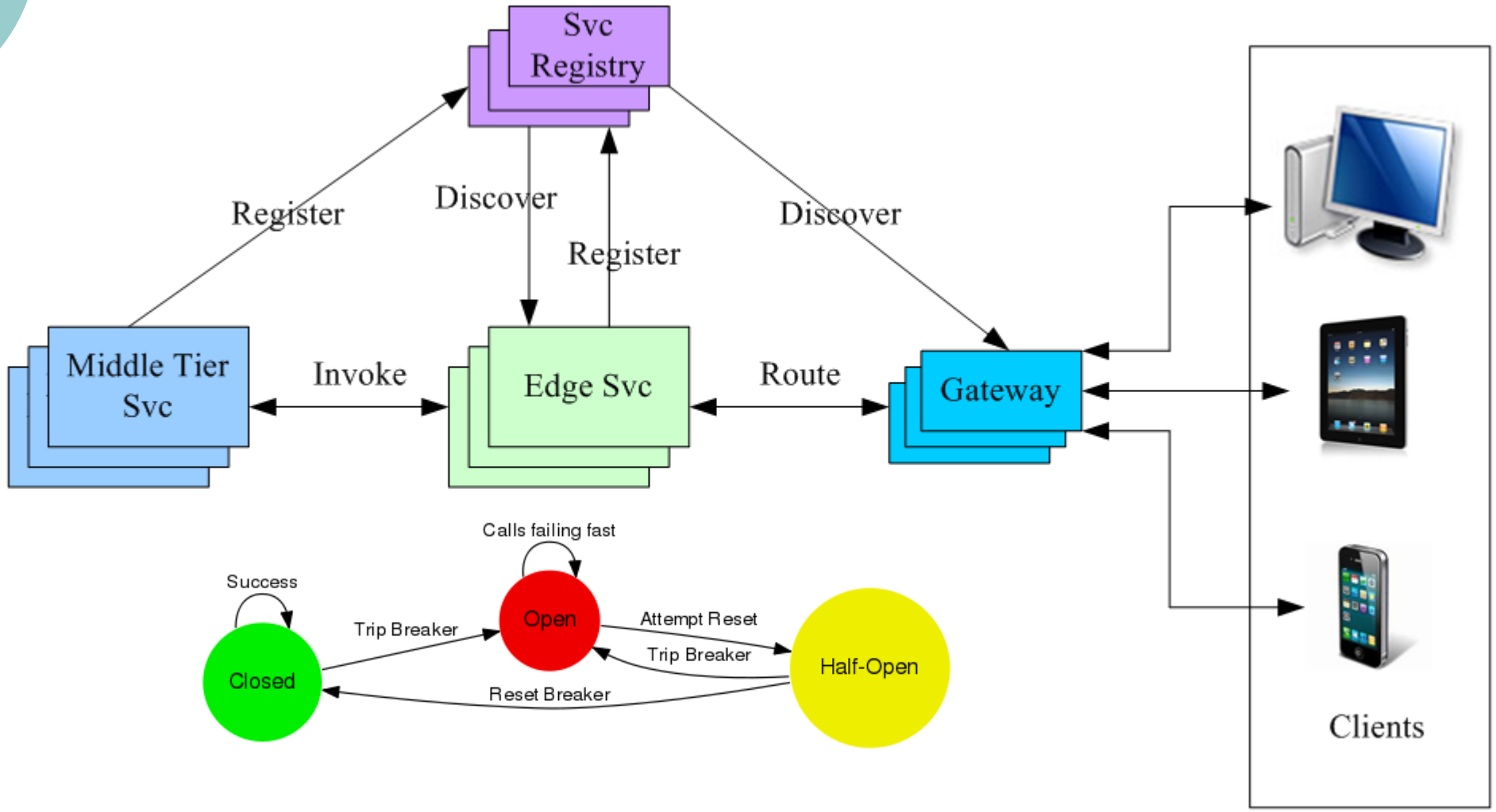

Silo logic
Silo database
Monolithic application
logic
logic
logic
Silo database
Internally componentized application
Microservice component
Microservice component
Microservice component
Microservices application

微服务、SOA 和 API


Exposure Gateway (external)
Engagement Applications
Traditional application
Traditional application
Microservice application
Microservice application
Enterprise Boundary
Integration Hub
Integration Hub
System of Record A
System of Record B
System of Record C
Business Partner
SaaS Application
Public API
Enterprise API
API Gateway

微服务基础框架及组件

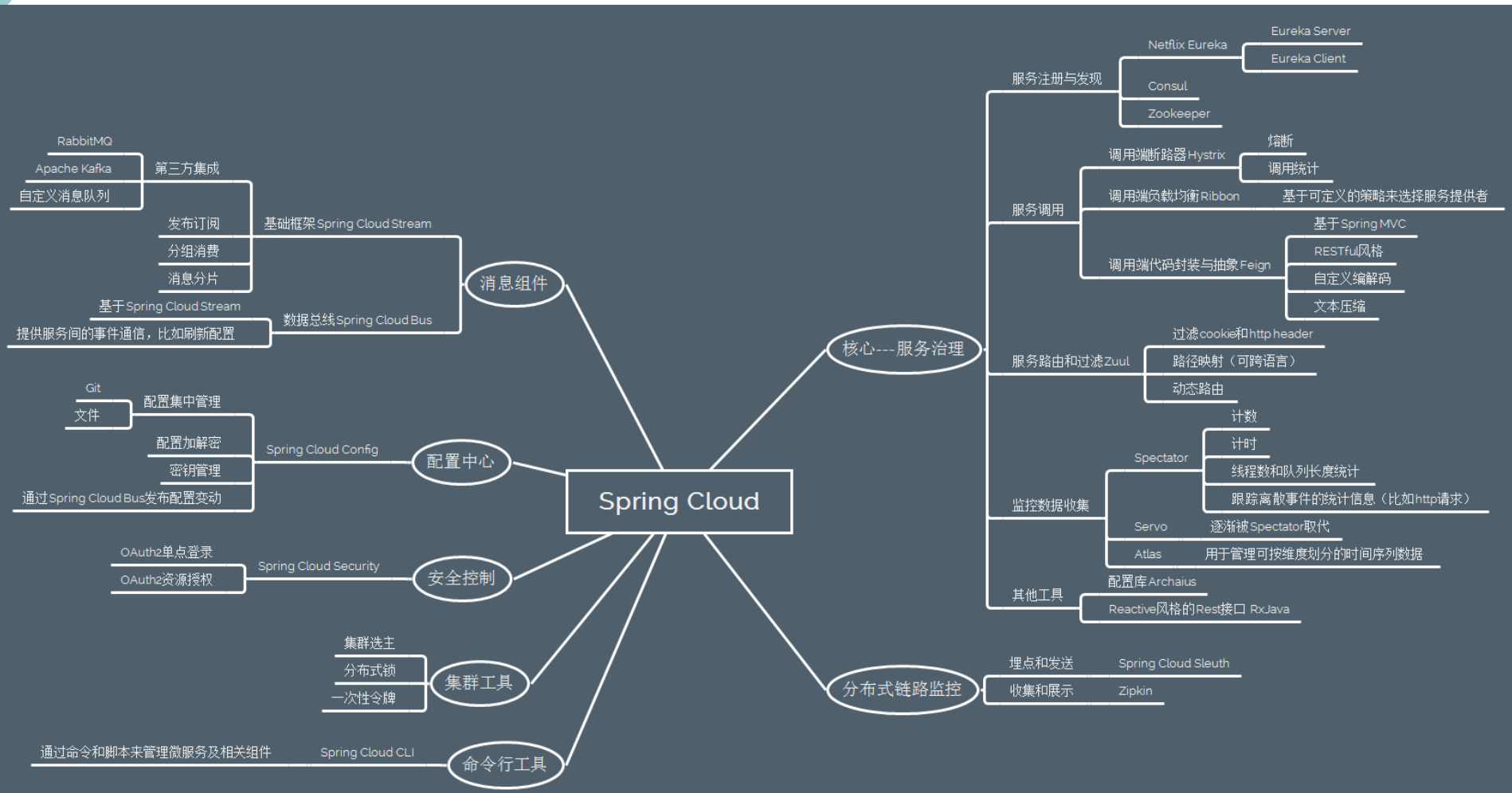
- 注册中心、服务网关、断路器



Dubbo vs Spring Cloud

- Dubbo是一个分布式服务框架，是阿里巴巴服务化治理的核心框架，并被广泛应用于阿里巴巴集团的各成员站点，在国内有较大影响力。
- Spring Cloud是一个基于Spring Boot实现的云应用开发工具，是Pivotal提供的用于简化分布式系统构建的工具集。
- 相比于Dubbo只实现了服务治理，Spring Cloud下面有17个子项目，覆盖了微服务架构下的方方面面。

Spring Cloud



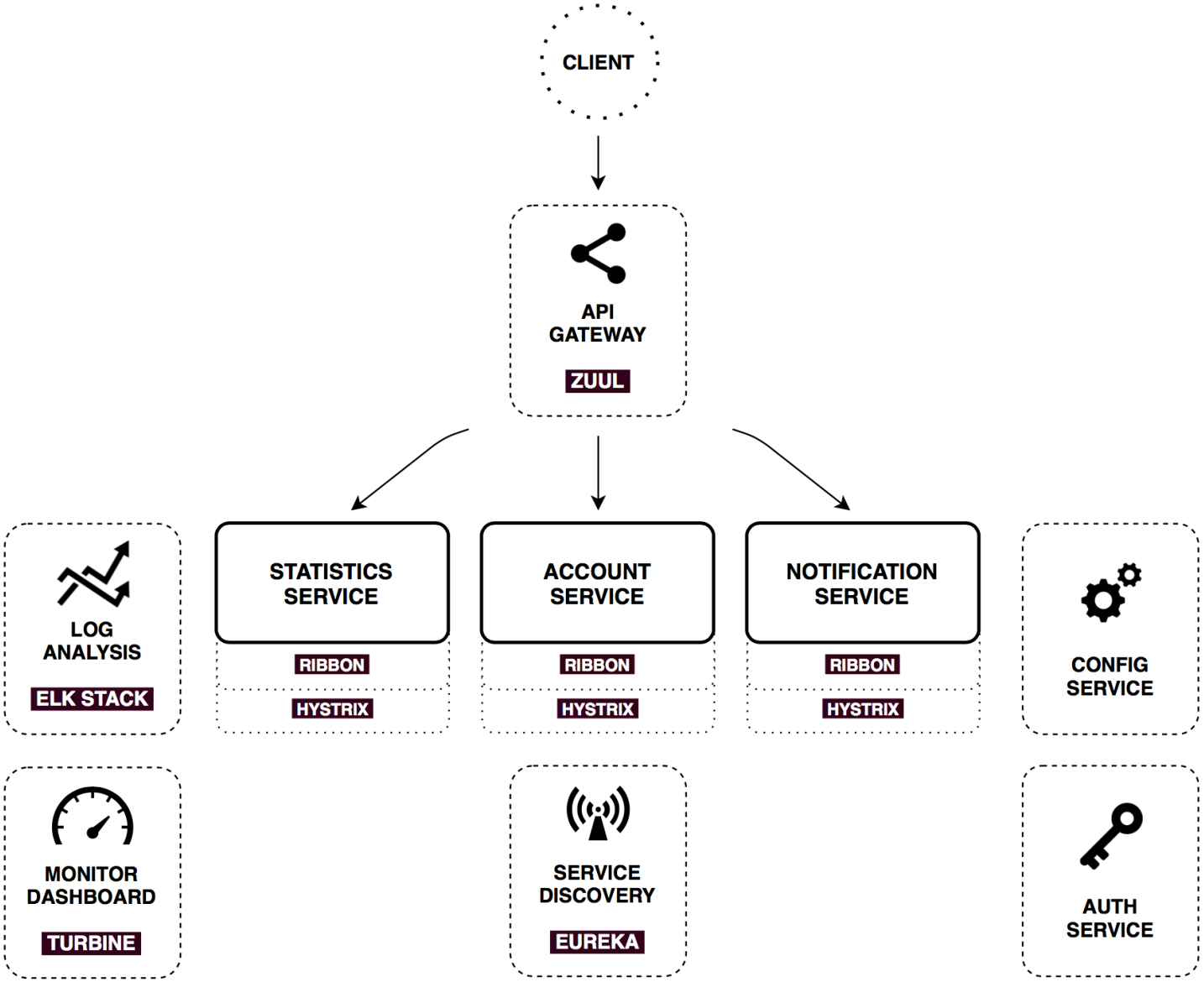
架构完整度对比

	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud Netflix Zuul
断路器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

微服务与Docker

- 虽然微服务架构带来了诸多优势，但构建、部署、维护分布式的微服务系统并不容易。
- 而容器所提供的轻量级、面向应用的虚拟化运行环境为微服务提供了理想的载体。
  - 将各个微服务以及架构组件构建为Docker镜像。
- 同样，基于容器技术的云服务将极大的简化容器化微服务创建、集成、部署、运维的整个流程，从而推动微服务在云端的大规模实践。

Spring Cloud与Docker



Kubernetes与Docker

- Kubernetes
  - Docker生态圈中重要一员
  - Google大规模容器管理技术的开源版本
  - 一套自动化容器管理平台提供应用部署、维护、 扩展机制等功能，用于容器的部署、自动化调度和集群管理。
- Kubernetes 有以下的特性：
  - 容器的自动化部署，自动化扩展或者缩容
  - 容器成组，对外提供服务，支持负载均衡
  - 服务的健康检查，自动重启
  - 以集群的方式运行、管理跨机器的容器
  - 解决Docker跨机器容器之间的通讯问题

Dubbo

Apache Dubbo

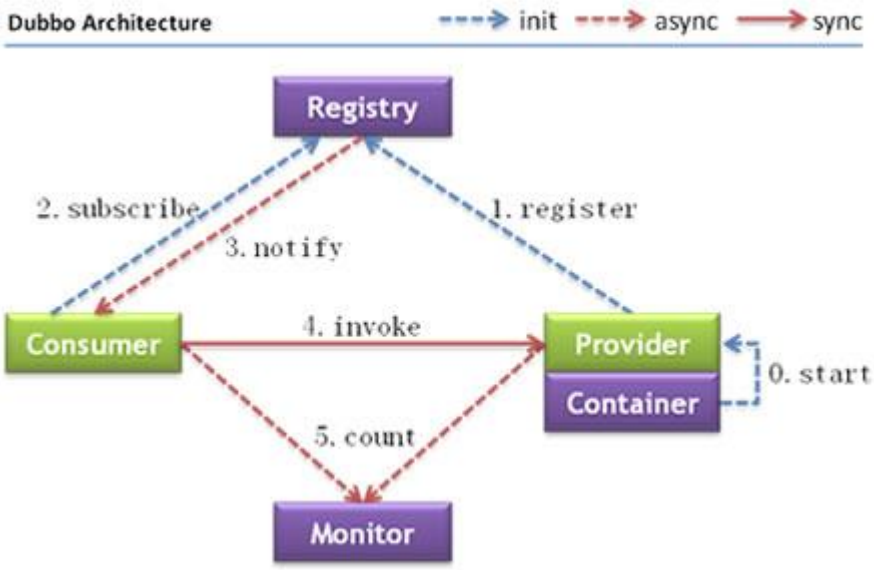
高性能、轻量级的Java RPC框架

三大核心能力：

- 面向接口的远程方法调用
  - 就像调用本地方法一样调用远程方法
  - 只需简单配置，没有任何API入侵
- 服务自动注册和发现
  - 注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者
- 智能容错和负载均均衡

Dubbo架构

节点角色说明



节点	角色说明
Provider	暴露服务的服务提供方
Consumer	调用远程服务的服务消费方
Registry	服务注册与发现的注册中心
Monitor	统计服务的调用次数和调用时间的监控中心
Container	服务运行容器

Dubbo架构

调用关系说明

1. 服务容器负责启动、加载、运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

Dubbo使用

- Dubbo采用全Spring配置方式，只需用Spring加载Dubbo的配置即可。
- 同时也提供了可编程配置方式。

编程配置：

```
// 服务实现
XxxService xxxService = new XxxServiceImpl();

// 当前应用配置
ApplicationConfig application = new ApplicationConfig();
application.setName("xxx");

// 连接注册中心配置
RegistryConfig registry = new RegistryConfig();
registry.setAddress("10.20.130.230:9090");
registry.setUsername("aaa");
registry.setPassword("bbb");

// 服务提供者协议配置
ProviderConfig provider = new ProviderConfig();
provider.setProtocol("dubbo");
provider.setPort(12345);
provider.setThreads(200);

// 服务提供者暴露服务配置
ServiceConfig service = new ServiceConfig();
service.setApplication(application);
service.setRegistry(registry); // 多个注册中心可以用setRegistries()
service.setProvider(provider); // 多个提供者可以用setProviders()
service.setInterfaceClass(XxxService.class);
service.setRef(xxxService);
service.setVersion("1.0.0");
service.export(); // 触发服务注册
```

Schema配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:dubbo="http://repo.alibaba-inc.com/schema/dubbo"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://repo.alibaba-inc.com/schema/dubbo
http://repo.alibaba-inc.com/schema/dubbo/dubbo-component-2.0.xsd">
    <!--服务实现 -->
    <bean id="xxxService" class="com.alibaba.xxx.XxxServiceImpl" />

    <!--当前应用配置 -->
    <dubbo:application name="morgan"  />

    <!-- 连接注册中心配置 -->
    <dubbo:registry address="10.20.130.230:9090" username="admin"
    password="hello1234" />

    <!-- 服务提供者协议配置 -->
    <dubbo:provider protocol="dubbo" port="12345" threads="200" />

    <!-- 服务提供者暴露服务配置 -->
    <dubbo:service interface="com.alibaba.xxx.XxxService"
    version="1.0.0" ref="xxxService" />
</beans>
```

Dubbo-服务提供者

1、定义服务接口

DemoService.java

```java
package org.apache.dubbo.demo;

public interface DemoService {
    String sayHello(String name);
}
```

2、在服务提供方实现接口

DemoServiceImpl.java

```java
package org.apache.dubbo.demo.provider;

import org.apache.dubbo.demo.DemoService;

public class DemoServiceImpl implements DemoService {
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

Dubbo-服务提供者

3、通过 Spring 配置声明暴露服务

provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans        http://w

    <!-- 提供方应用信息，用于计算依赖关系 -->
    <dubbo:application name="hello-world-app"  />

    <!-- 使用multicast广播注册中心暴露服务地址 -->
    <dubbo:registry address="multicast://224.5.6.7:1234" />

    <!-- 用dubbo协议在20880端口暴露服务 -->
    <dubbo:protocol name="dubbo" port="20880" />

    <!-- 声明需要暴露的服务接口 -->
    <dubbo:service interface="org.apache.dubbo.demo.DemoService" ref="demoService"

    <!-- 和本地bean一样实现服务 -->
    <bean id="demoService" class="org.apache.dubbo.demo.provider.DemoServiceImpl" /
</beans>
```

Dubbo-服务提供者

4、加载 Spring 配置

Provider.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Provider {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext
        context.start();
        System.in.read(); // 按任意键退出
    }
}
```

Dubbo-服务消费者

- 通过Spring配置引用远程服务

consumer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans        http://w

    <!-- 消费方应用名，用于计算依赖关系，不是匹配条件，不要与提供方一样 -->
    <dubbo:application name="consumer-of-helloworld-app"  />

    <!-- 使用multicast广播注册中心暴露发现服务地址 -->
    <dubbo:registry address="multicast://224.5.6.7:1234" />

    <!-- 生成远程服务代理，可以和本地bean一样使用demoService -->
    <dubbo:reference id="demoService" interface="org.apache.dubbo.demo.DemoService"
</beans>
```

Dubbo-服务消费者

- 加载Spring配置，并调用远程服务

Consumer.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.apache.dubbo.demo.DemoService;

public class Consumer {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext
        context.start();
        DemoService demoService = (DemoService)context.getBean("demoService"); // 获
        String hello = demoService.sayHello("world"); // 执行远程方法
        System.out.println( hello ); // 显示调用结果
    }
}
```

Dubbo-注册中心

- 使用注册中心时，通常使用zookeeper作为注册中心。
- 只需修改相应的配置文件.
- 在 provider 和 consumer 中增加 zookeeper 客户端 jar 包依赖：

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.3.3</version>
</dependency>
```

- 提供者provider.xml和消费者consumer.xml中加入

```
<dubbo:registry address="zookeeper://10.20.153.10:2181" />
```

§5.2 Serverless

什么是Serverless？

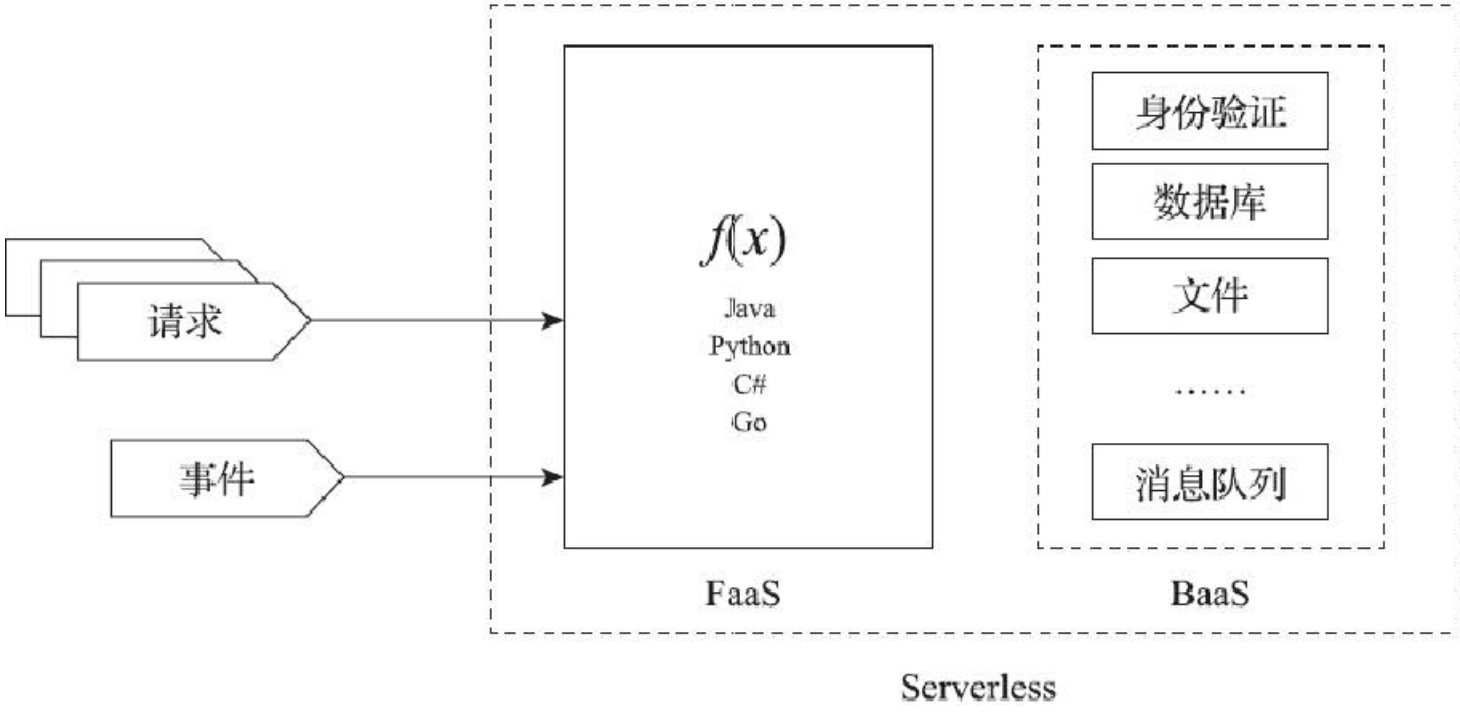
- Serverless架构即“无服务器”架构。
- Serverless不是具体的一个编程框架、类库或者工具。
- Serverless是一种软件系统架构思想和方法，它的核心思想是用户无须关注支撑应用服务运行的底层主机。
- Serverless架构下，软件的运行环境不再是具体的一台或多台服务器，而是支持Serverless的云计算平台。
- 服务器对于用户而言是透明的，不再是用户所操心的资源对象。

Serverless框架和平台

- Serverless公有云服务：
  - AWS Lambda
  - Azures Functions
- 私有云Serverless框架：
  - OpenWhisk
  - Kubeless
  - Fission
  - OpenFaaS
  - Fn

Serverless的核心技术

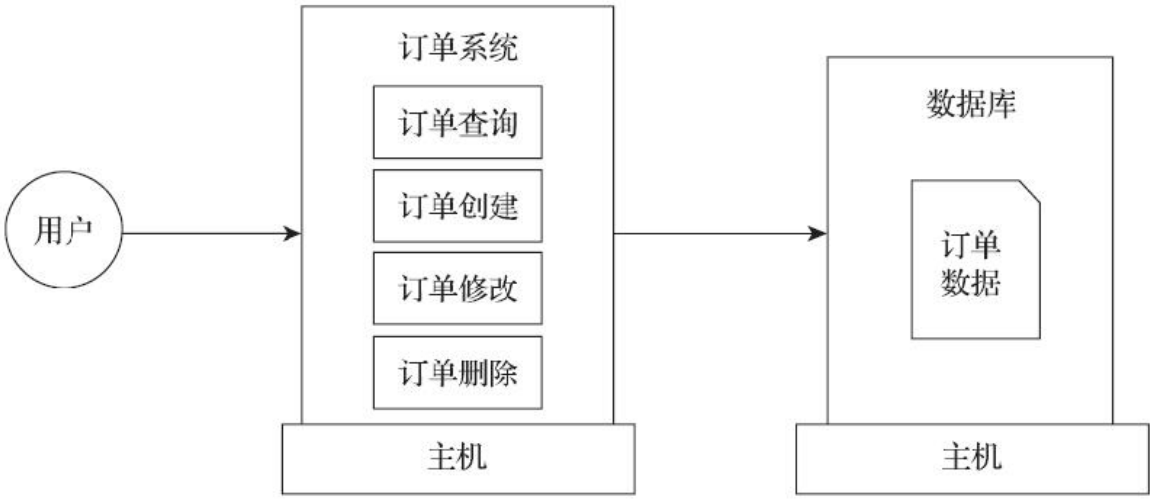
- Serverless需要两个方面的支持：
  - 函数即服务（Function as a Service, FaaS）
  - 后台即服务（Backend as a Service, BaaS）



Serverless vs. 传统框架

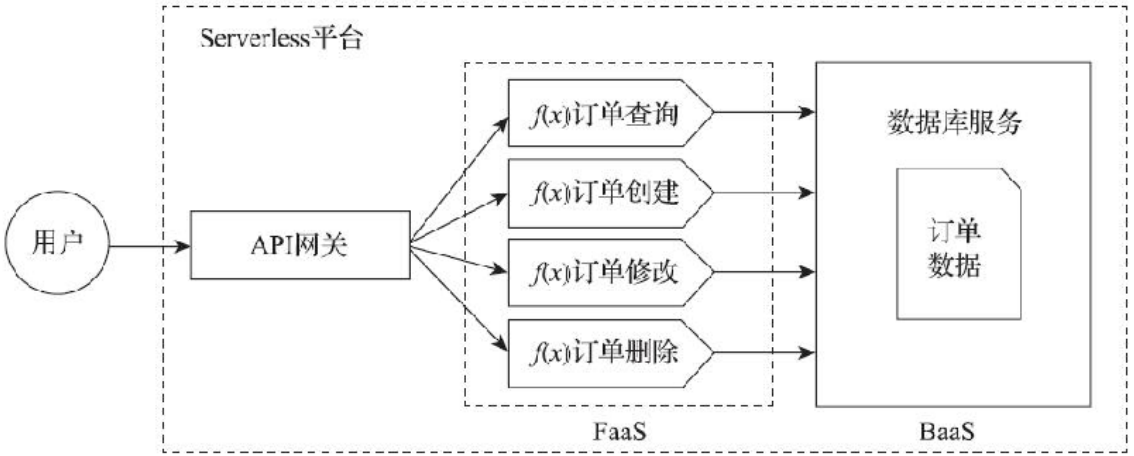
传统应用框架

- 应用部署在主机之上
- 所有的逻辑都集中在同一个部署交付件中



Serverless应用框架

- 应用部署在Serverless平台上
- 由平台提供计算资源
- 应用逻辑分散成了函数，通过API网关对函数逻辑进行统一的管理。



Serverless的技术特点

- 按需加载
- 事件驱动
- 状态非本地持久化
- 非会话保持（无法保证由同一个实例处理同一客户的请求）
- 动弹性伸缩
- 应用函数化
- 依赖服务化

Serverless的局限性

- 控制力。用户对底层的计算资源没有控制力。
- 可移植性。目前Serverless应用的实现很大程度上依赖于Serverless平台及该平台上的FaaS和BaaS服务。
- 安全性。
- 性能。应用的首次加载和重新加载将产生一定的时延。
- 执行时长。由于按需加载，不是长时间持续部署，大部分Serverless平台对FaaS函数的执行时长存在限制。
- 技术成熟度。目前Serverless相关平台、工具和框架还处在不断变化和演进的阶段。

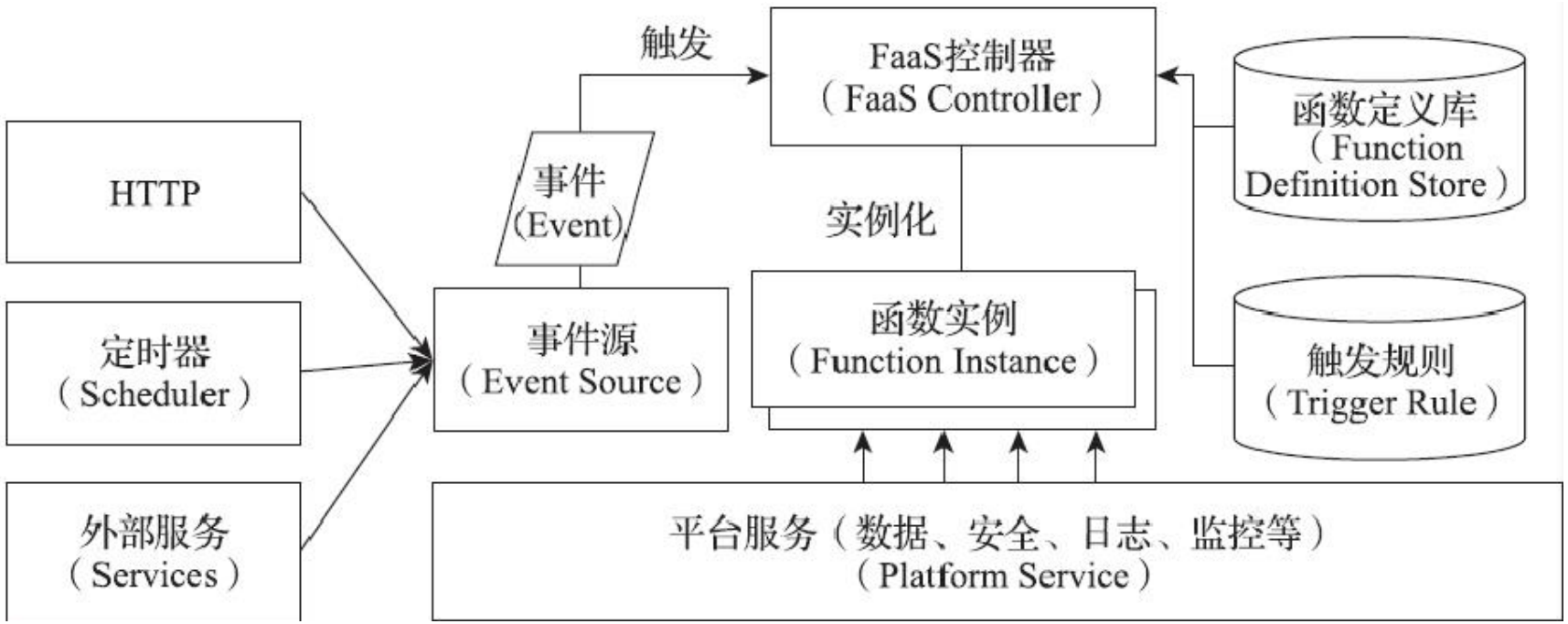
Serverless vs. 微服务

- 相似性：
  - Serverless和微服务两种架构都强调功能的解构。
  - 都强调最小的成员单位专注于做一件事情。
- 差异性：
  - Serverless的最小成员单位是函数，微服务架构最小成员单位是微服务。
  - Serverless强调的是“减负”，将服务器移出用户的管理范围。
  - 微服务强调的是化整为零，提供应用架构的灵活度。

FaaS

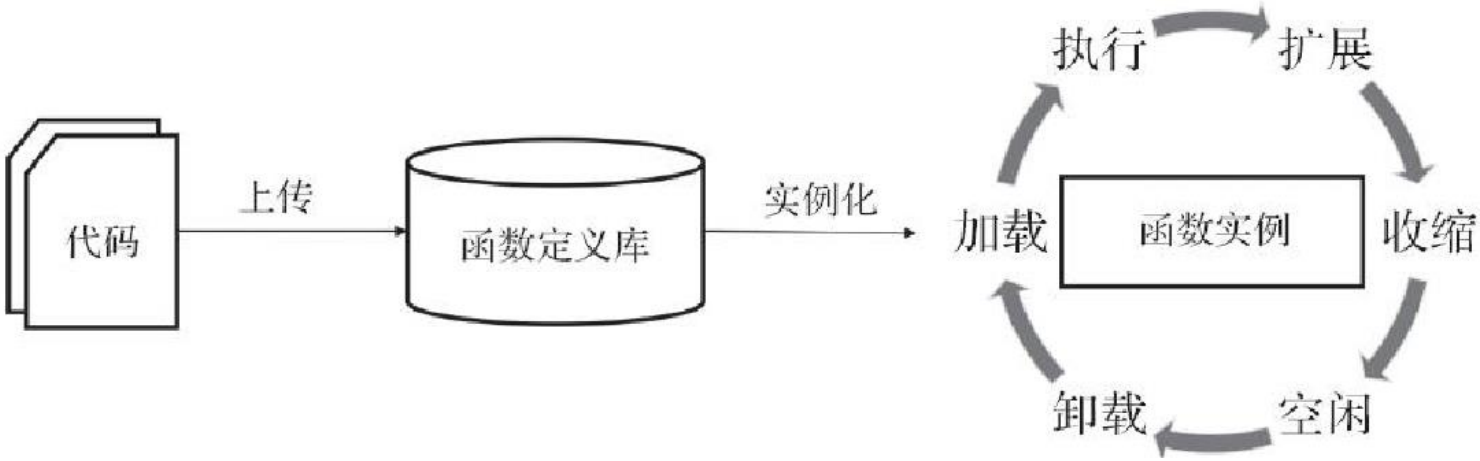
- FaaS是当前Serverless实现的技术基础
- 应用程序的粒度：Function
- 特点：
  - 抽象了底层计算资源
  - 按使用量付费
  - 自动弹性拓展
  - 事件驱动

FaaS架构



- 函数定义
- 函数实例，在运行状态的应用函数实例
- 控制器，负责应用函数的加载、执行等流程的管理
- 事件，事件驱动架构中的事件。
- 事件源，事件来源，如数据库插入新记录、消息队列收到了新消息。
- 触发规则，定义事件与函数的关系及触发的规则。
- 平台服务，支撑应用运行的各类底层服务，如计算资源、数据存储等。

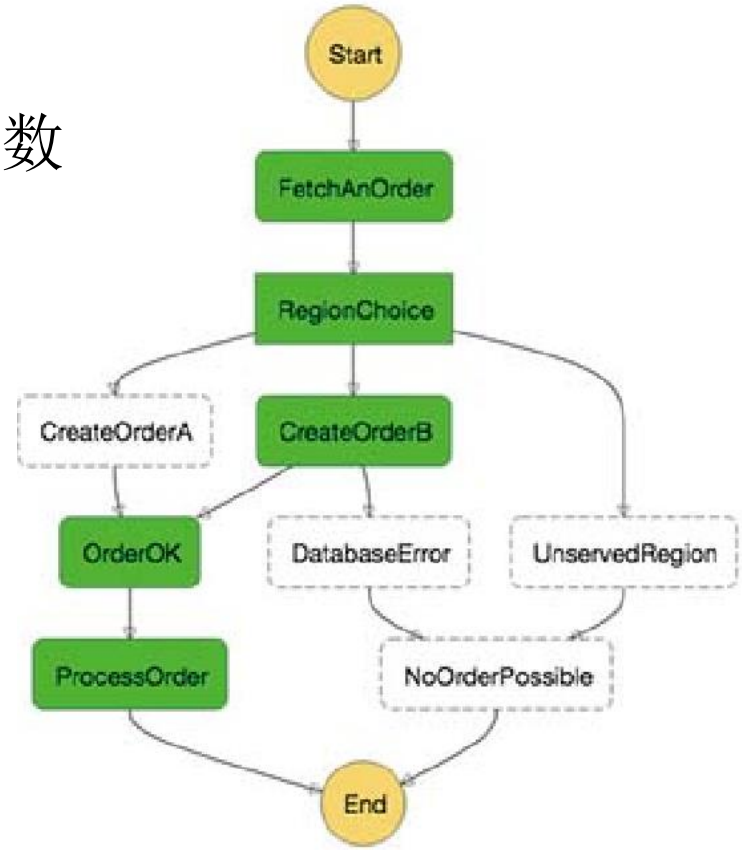
函数的生命周期



- 用户根据所选定的FaaS平台的规范进行函数应用的开发。
- 编写好的函数将上传至FaaS平台。平台将负责编译和构建这些函数，并将构建的输出保存。
- 用户设置函数被触发的规则，将事件源与特定版本的函数进行关联。
- 当事件到达且满足触发规则时，平台将会部署、编译构建后的函数并执行。
- 平台将监控函数执行的状态，根据请求量的大小，平台负责对函数实例进行扩容和缩容。

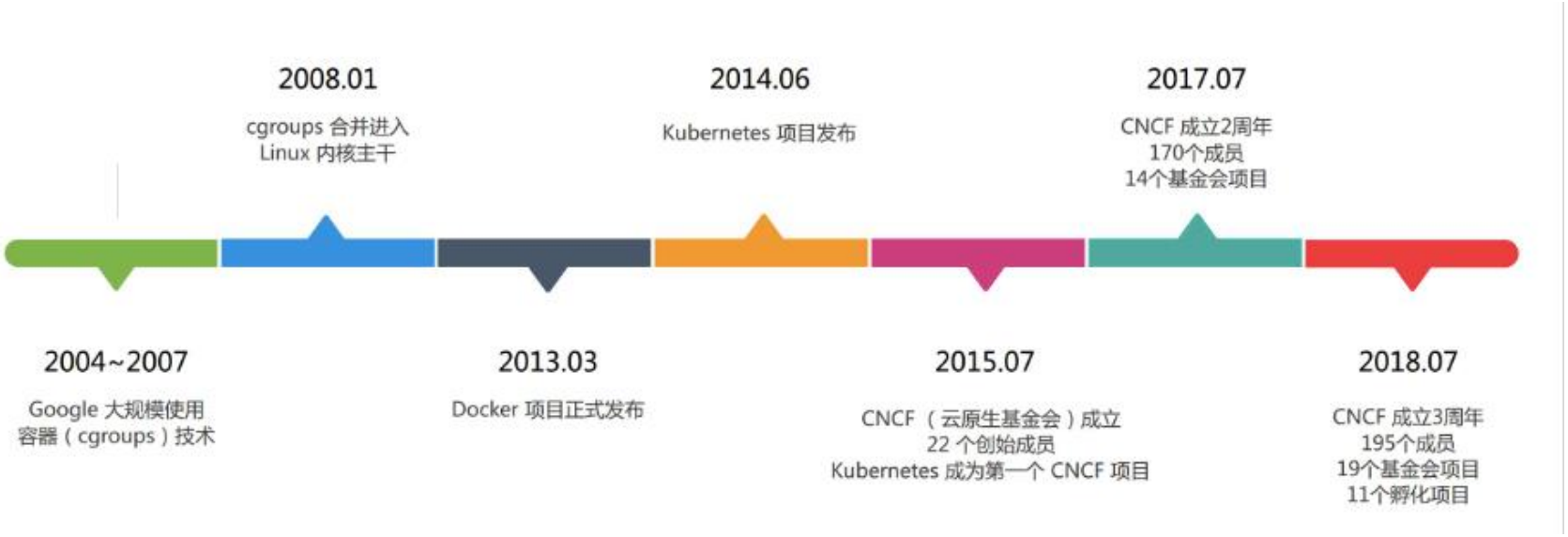
函数工作流

- 当使用场景涉及多个函数，需要共同协作完成
  - 需要处理执行顺序、错误重试、异常捕获以状态传递等细节
  - 需要用户进行逻辑编排。
- 目前一些FaaS开始实现针对FaaS函数流程变化的服务和工具
  - 如AWS Step Functions和Fission Workflows.



§5.3 云应用开发

- Cloud Native云原生
- 指充分利用云平台的各种功能和服务所设计的应用程序
- 软件从诞生起就生在云上、长在云上的、全新的软件开发、发布和运维模式



云原生技术范畴

云应用定义与开发流程

1.应用定义与镜像制作
2.CI/CD
3.消息和 Streaming
4.数据库

云应用编排与管理

1.应用编排与调度
2.服务发现与治理
3.远程调用
4.API 网关
5.Service Mesh

监控与可观测性

1.监控
2.日志
3.Tracing
4.混沌工程

云原生底层技术

1.容器运行时
2.云原生存储技术
3.云原生网络技术

云原生工具集

1.流程自动化与配置管理
2.容器镜像仓库
3.云原生安全技术
4.云端密码管理

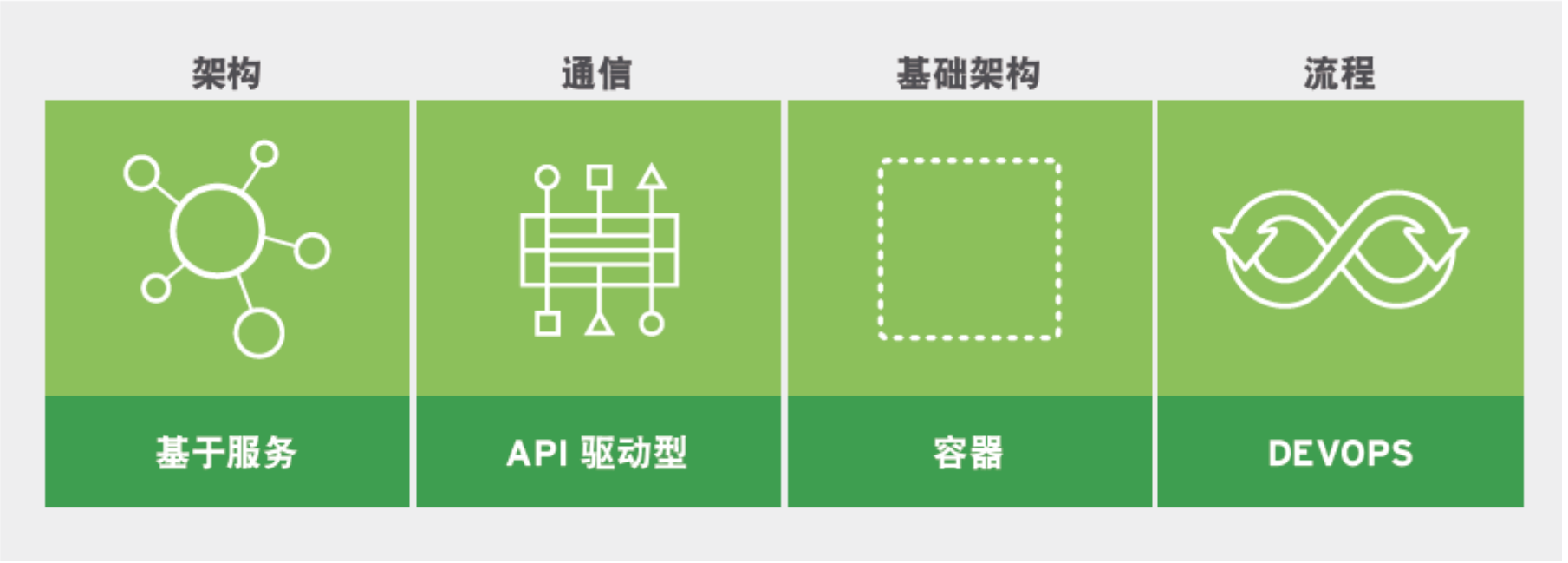
Serverless

1.FaaS
2.BaaS
3.Serverless 计费

云原生vs.传统应用

	传统	云原生
重点关注	使用寿命和稳定性	上市速度
开发方法	瀑布式半敏捷型开发	敏捷开发、DevOps
团队	相互独立的开发、运维、质量保证和安全团队	协作式DevOps团队
交付周期	长	短且持续
应用架构	紧密耦合 单体式	松散耦合 基于服务 基于应用接口（API）的通信
基础架构	以服务器为中心 适用于企业内部 依赖于基础架构 纵向扩展 针对峰值容量预先进行置备	以容器为中心 适用于企业内部和云环境 可跨基础架构进行移植 横向扩展 按需提供容量

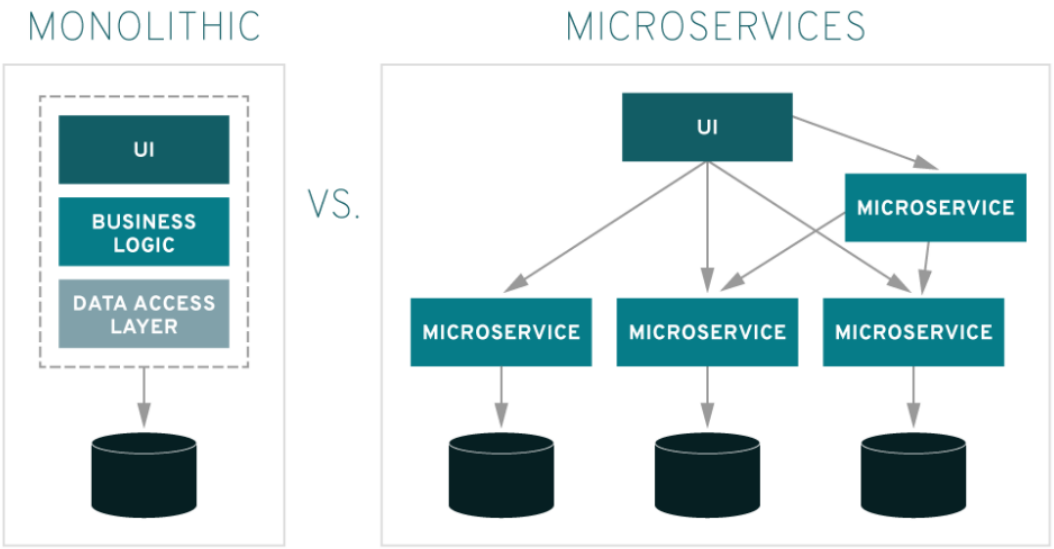
云原生应用开发和部署的四大原则



基于服务的架构

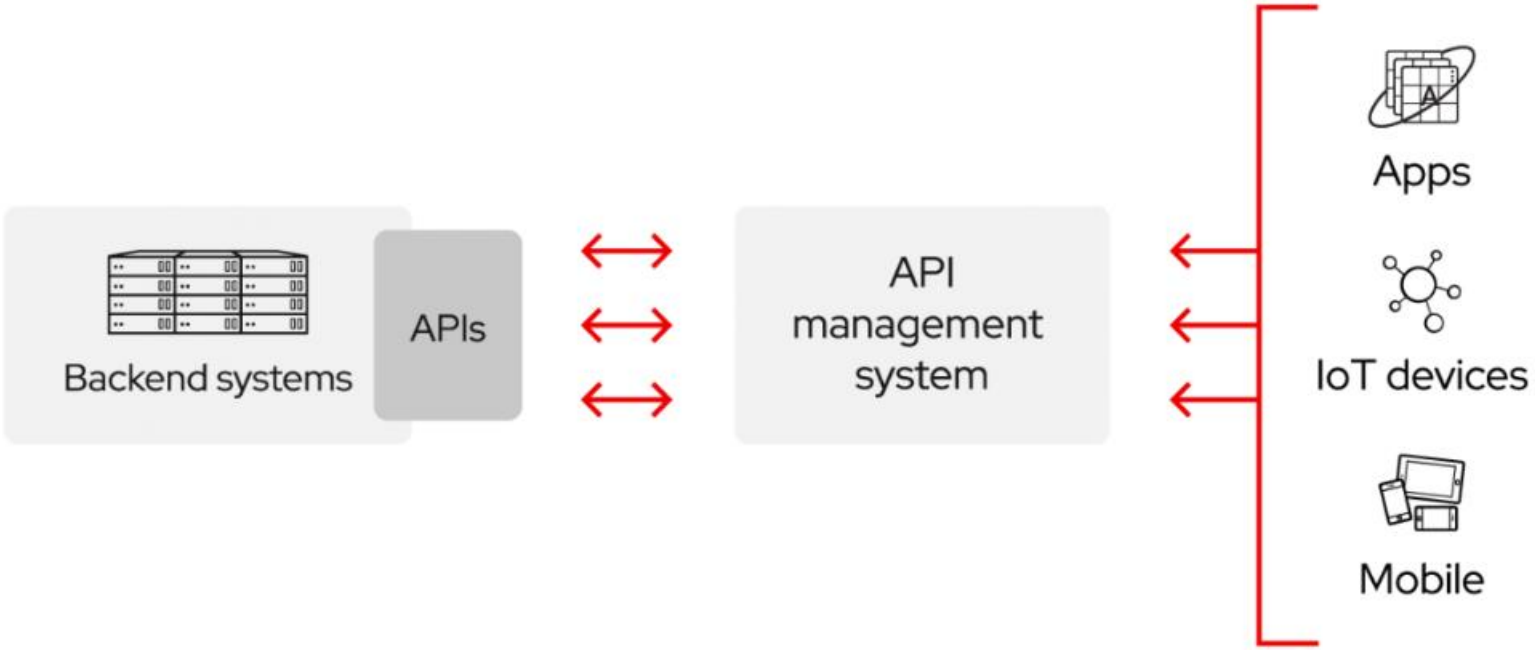
微服务优势：

- 可并行开发多个微服务，缩短开发时间。
- 高度可扩展，跨多个服务器和基础架构进行部署
- 高弹性，独立的服务不会彼此影响
- 易于部署，应用模块块且小巧
- 更加开放，使用多语言API，自由选用合适的语言和应用

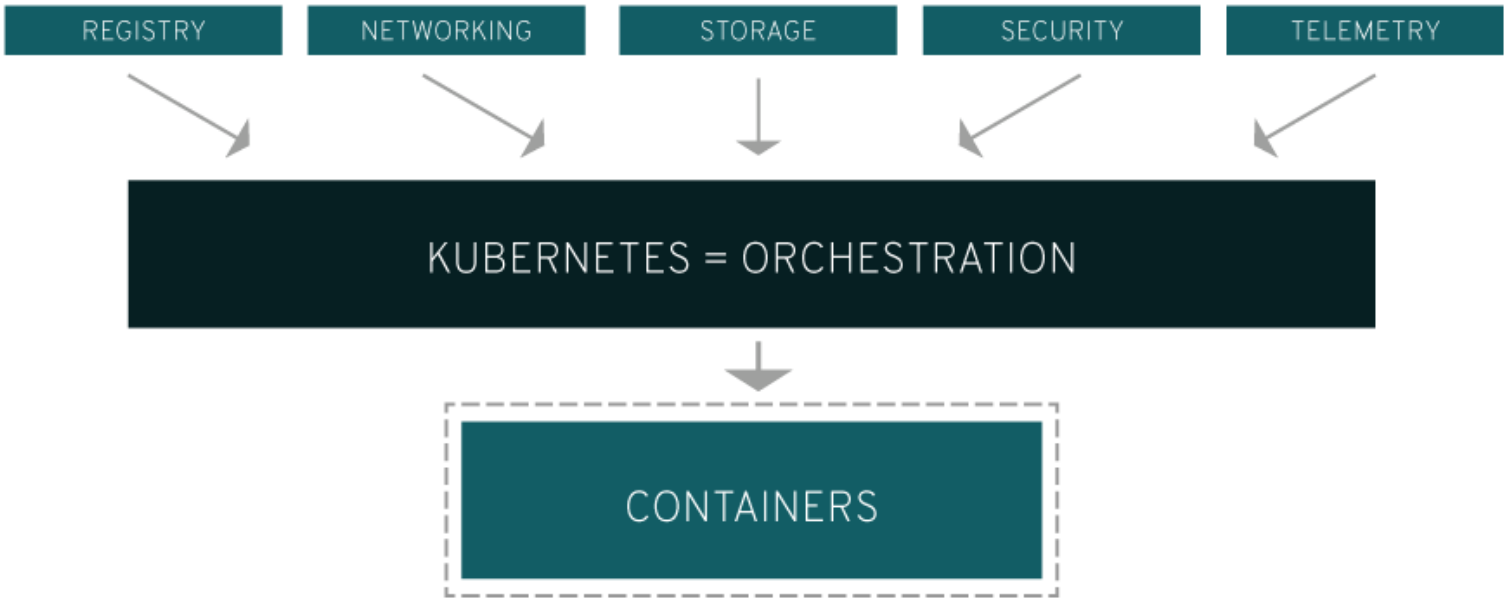


基于API的通信

- 应用编程接口 (API) 由一组工具、定义和协议组合而成
- 服务通过与技术无关的轻量级 API 来提供
- 这些 API 可以降低与部署、可扩展性和维护相关的复杂性和费用

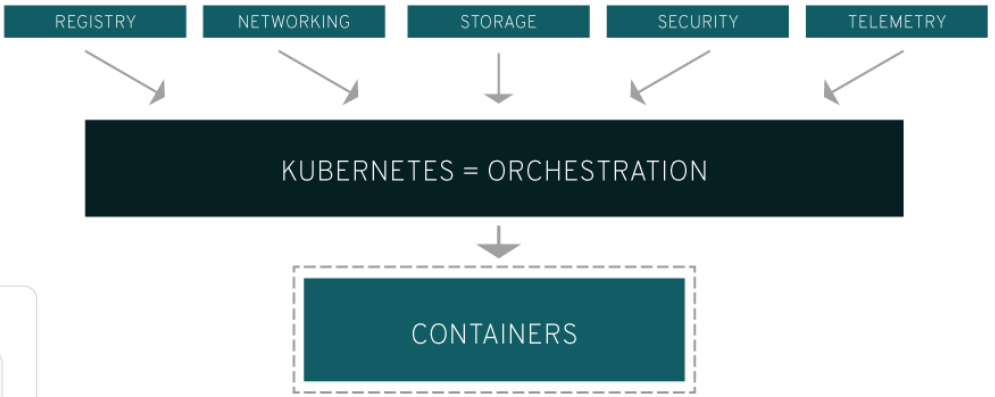


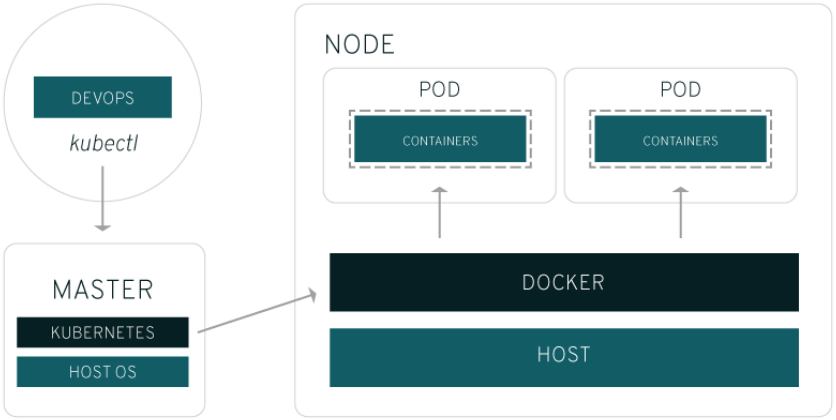
§ 3.2.3基于容器的基础架构



基于容器的基础架构

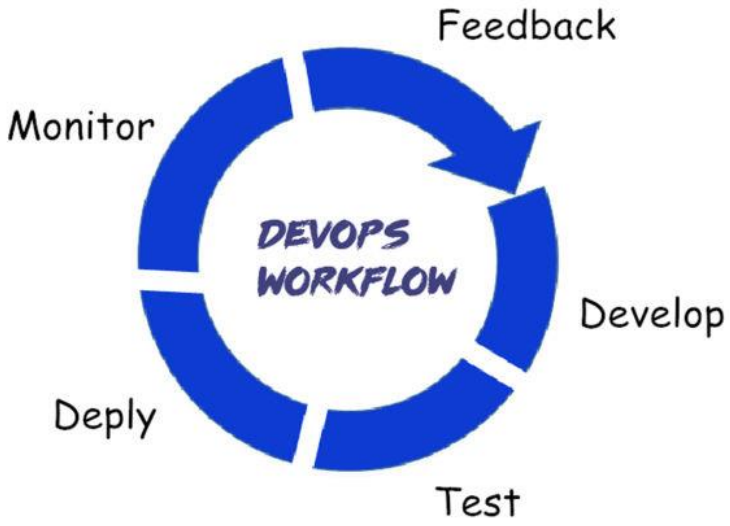
- 容器可以让开发人员全心投入应用开发
- 而运维团队则可专注于基础架构维护
  - 容器编排：在整个企业内管理这些部署的方式





基于DevOps的流程管理

- DevOps 是一个完整的面向IT运维的流程
  - 运维和开发工程师共同参与
  - 设计-开发-生产
- 以 IT 自动化以及持续集成（CI）、持续部署（CD）为基础
- 优化程式开发、测试、系统运维等所有环节



基于K8S的云应用开发实践

1、创建、运行及共享容器镜像

- 创建一个应用服务

代码清单 2.2　一个简单的 Node.js 应用：app.js

```
const http = require('http');
const os = require('os');

console.log("Kubia server starting...");

var handler = function(request, response) {
  console.log("Received request from " + request.connection.remoteAddress);
  response.writeHead(200);
  response.end("You've hit " + os.hostname() + "\n");
};

var www = http.createServer(handler);
www.listen(8080);
```

- 为镜像创建Dockerfile

代码清单 2.3　构建应用容器镜像的 Dockerfile

```
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]
```

基于K8S的云应用开发实践

- 构建容器镜像

```
$ docker build -t kubia .
```

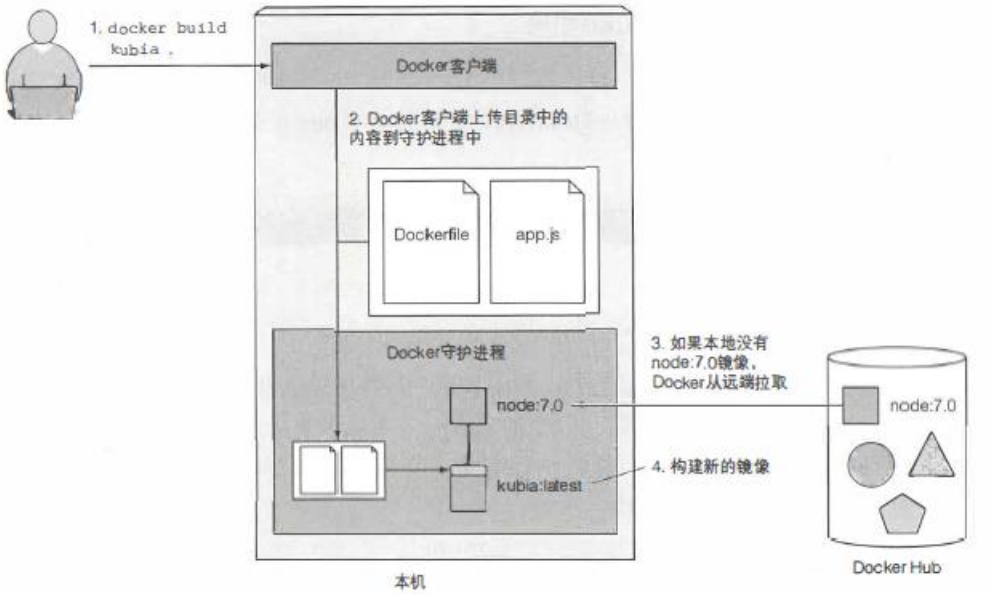


图 2.2 基于 Dockerfile 构建一个新的容器镜像

- 运行、测试容器镜像

```
$ docker run --name kubia-container -p 8080:8080 -d kubia
```

基于K8S的云应用开发实践

2、容器化部署于Kubernetes平台

- 搭建K8S集群
- 部署应用

  最简单的方式是使用kubectl run命令（也可以通过Deployment进行部署）

```
$ kubectl run kubia --image=luksa/kubia --port=8080 --generator=run/v1
replicationcontroller "kubia" created
```

- 创建服务对象

  告知Kubernetes对外暴露之前创建的服务

```
$ kubectl expose rc kubia --type=LoadBalancer --name kubia-http
service "kubia-http" exposed
```

- 水平伸缩应用

```
$ kubectl scale rc kubia --replicas=3
replicationcontroller "kubia" scaled
```

基于K8S的云应用开发实践

- 使用Deployment声明式地升级应用

代码清单 9.7 Deployment 定义：kubia-deployment-v1.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: kubia
spec:
  replicas: 3
  template:
    metadata:
      name: kubia
      labels:
        app: kubia
    spec:
      containers:
      - image: luksa/kubia:v1
        name: nodejs
```

Deployment 属于 apps API 组，版本为 v1beta1

需要将原有 kind 从 ReplicationController 修改为 Deployment

Deployment 的名称中不再需要包含版本号

```
$ kubectl create -f kubia-deployment-v1.yaml --record
deployment "kubia" created
```

- 触发滚动升级

```
$ kubectl set image deployment kubia nodejs=luksa/kubia:v2
deployment "kubia" image updated
```

基于K8S的云应用开发实践

- 通过负载均衡器将服务暴露出来

代码清单 5.12 Load Badancer 类型的服务：kubia-svc-loadbalancer.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-loadbalancer
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```

该服务从 Kubernetes 集群的基础架构获取负载平衡器

kubectl apply –f kubia-svc-loadbalancer.yaml

- 查看服务并测试

```
$ kubectl get svc kubia-loadbalancer
NAME                 CLUSTER-IP       EXTERNAL-IP       PORT(S)          AGE
kubia-loadbalancer   10.111.241.153   130.211.53.173    80:32143/TCP     1m

$ curl http://130.211.53.173
You've hit kubia-xueq1
```

本章小结

- 云应用技术纷繁复杂
- 基本原则
  - 微服务、容器、API、DevOps
- Serverless新兴技术
- 云应用开发框架、平台繁多
  - Spring cloud、Dubbo
  - K8S

谢谢！