华南理工大学
South China University of Technology

微服务与容器

华南理工大学 软件学院

fengtang@scut.edu.cn

群　号:743040565

加群密码：240827

第一章 课程简介

考核

实验（40%）

课程设计：分组项目 （60%）

课程设计需提交材料：

1.课程设计文档（不少于10页）

2.源码

3.团队分工及成员评分（团队最多6人）

必须微服务架构，不限制采用何种微服务框架，推荐Spring Cloud Netflix，Spring Cloud Alibaba，Dubbo

第一章 课程简介

课程使用的工具及技术

IDEA
Spring Cloud

前置技术

JAVA编程基础
spring框架开发基础

第一章 课程简介

1.微服务概述
2.docker技术
3.微服务架构设计
4.服务通信
5.服务治理
6.典型微服务框架
7.Spring Cloud服务注册
8.Spring Cloud服务调用与负载均衡
9.Spring Cloud熔断器
10.Spring Cloud网关
11.Spring Cloud Config集中配置
12.springcloud分布式链路追踪
微服务项目案例

第一章 微服务概述

1.1 从传统单体架构到面向服务架构

1.2 从面向服务架构到微服务

1.3 微服务简介

20年软件架构演化


单体
单个软件
独立安装
一台机器
SOA
分布式
面向服务
SOAP
跨平台
多服务
Web服务
Dubbo
微服务
Spring Cloud
Docker
K8S
云计算
云原生

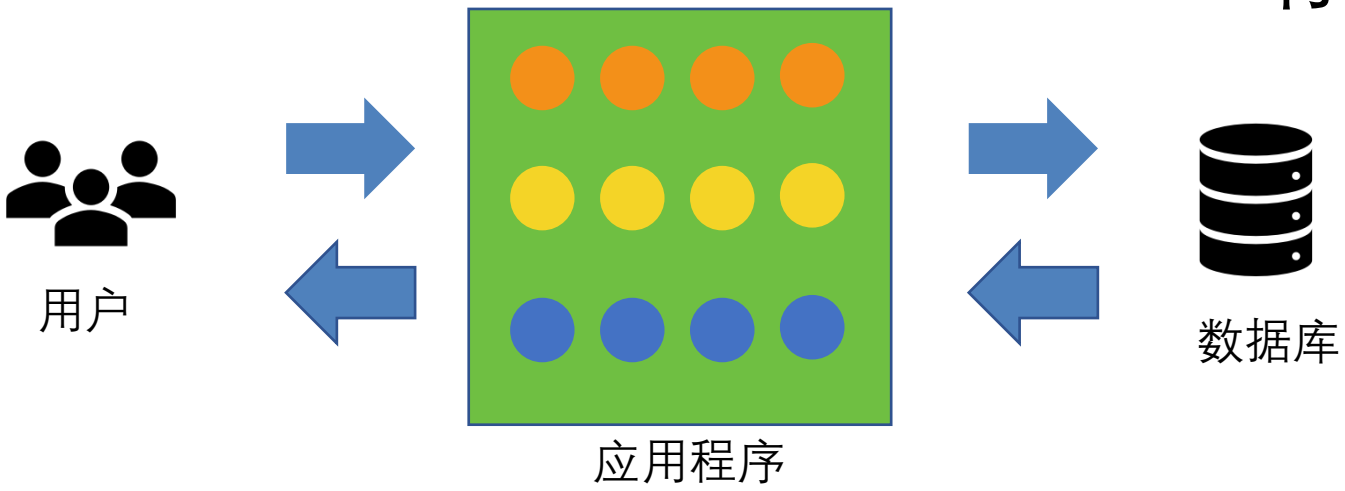
1.1 从传统单体架构到面向服务架构

- 1.1.1 JEE架构
- 1.1.2 SSH架构
- 1.1.3 面向服务架构

1.1.1 JEE架构

- 所有的程序安装在一个电脑上古时代
- 两层的C/S架构
- 三层的MVC架构

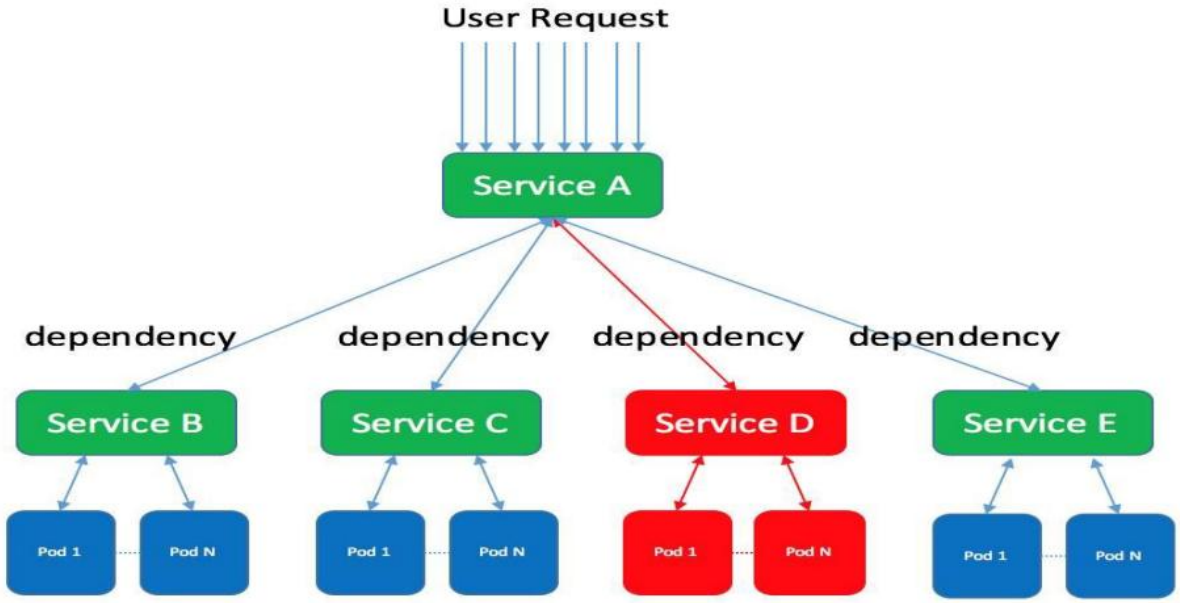
单体式架构



1.1.3 面向服务架构

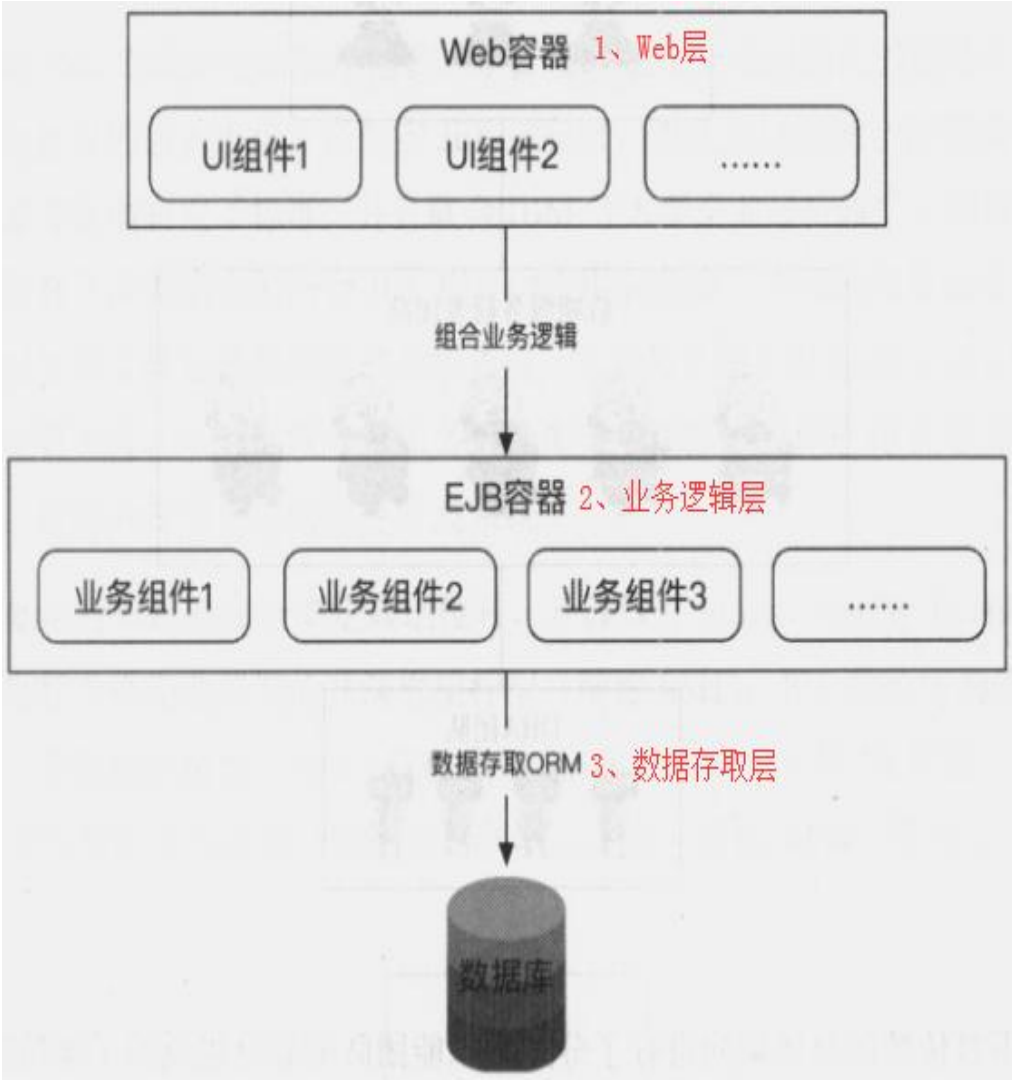
- 简单分布式系统RPC调用
- 面向服务架构SOA
- 微服务架构（Microservice Architect）

多体式架构



JEE架构

- 将软件架构分为三个层级：
- Web层：负责与用户交互或者对外提供接口。
- 业务逻辑层：为了实现业务逻辑而设计的流程处理和计算处理模块。
- 数据存取层：将业务逻辑层处理的结果持久化以待后续查询，并维护领域模型中对象的生命周期。（**ORM:Object Relational Mapping**）



JEE架构

- 以面向对象的Java编程语言为基础，作为企业级软件开发的运行时和开发平台。
- 这一时期，架构上把传统的单体系统进行分层，架构已经在一定程度上进行了逻辑上的拆分。
- 每个层次的多个业务逻辑实现仍在统一项目中，企业只能采用业务逻辑的隔离性解耦。
- 尽管JEE支持Web容器和EJB容器的分离部署。这一时期，大多数企业项目仍部署在同一应用服务器上，并运行在同一JVM进程。

1.1 从传统单体架构到面向服务架构

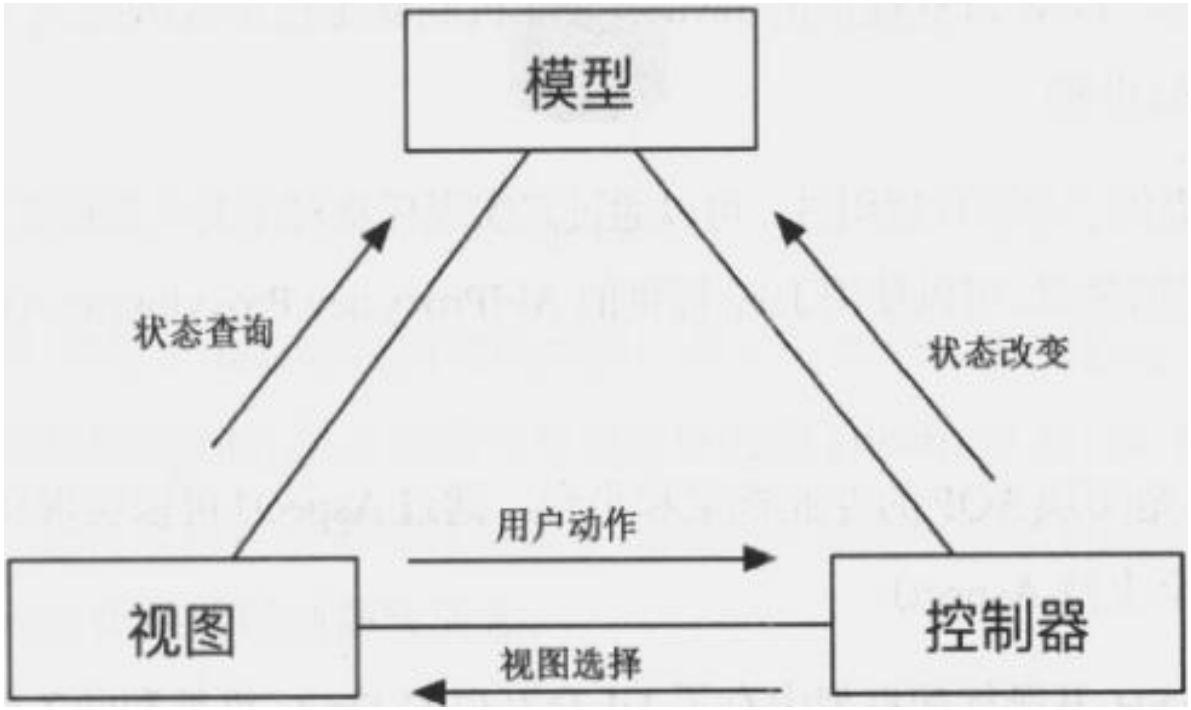
1.1.1 JEE架构

► 1.1.2 SSH架构

1.1.3 面向服务架构

SSH架构

- 开源软件Struts、Spring、Hibernate，简称SSH。
- Struts将用户交互层划分为视图、模型、控制器三大块。（MVC: Model-View-Controller）



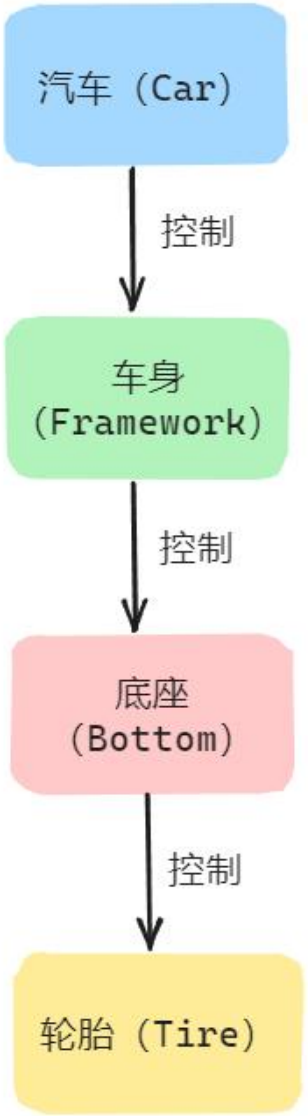
SSH结构

- ORM框架：面向对象领域的模型与关系数据库的纽带框架
- Hibernate框架：配置对象与数据库关系表之间的映射关系，对对象进行持久化和查询。

SSH架构

- Spring框架核心思想：IOC、AOP。
- IOC：控制翻转，将传统EJB基于容器开发改造为Java组件开发，运行时统一由Spring容器管理和串联。开发业务逻辑时，每个业务逻辑的服务组件相互独立，不依赖Spring但获得Spring的单元测试支持。
- AOP：面向切面编程，将面向对象方法无法抽象的业务逻辑，使用切面来表达。

1.1.2 SSH架构



1.1.2 SSH架构

```
package old;
/**
 * 传统开发方式, 耦合性问题
 */
// 汽车类
public class Car {
    // 车身
    private Framework framework;

    public Car() {
        framework = new Framework();
    }

    public void init() {
        System.out.println("do car");
        // 汽车的组建依赖于车身
        framework.init();
    }
}
```

```
package old;

// 车身类
public class Framework {
    private Bottom bottom;

    public Framework() {
        bottom = new Bottom();
    }

    public void init() {
        System.out.println("do
framework");
        // 车身的组建依赖于底盘
        bottom.init();
    }
}
```

```
package old;

// 底盘类
public class Bottom {
    private Tire tire;

    public Bottom() {
        tire = new Tire();
    }

    public void init() {
        System.out.println("do bottom");
        // 底盘的组建依赖于轮胎
        tire.init();
    }
}
```

```
package old;

// 轮胎类
public class Tire {
    private int size = 17; // 轮胎的尺寸
    public void init() {
        System.out.println("size -> " + size);
    }
}
```

```
package old;
public class Test {
    public static void main(String[] args) {
        Car car = new Car();
        car.init();
    }
}
```

增加轮胎大小的参数

```
package old;
/**
 * 传统开发方式, 耦合性问题
 */
// 汽车类
public class Car {
   // 车身
   private Framework framework;

 public Car(int size) {
    framework = new Framework(size);
   }
   public void init() {
      System.out.println("do car");
      // 汽车的组建依赖于车身
      framework.init();
   }
}
```

```
package old;

// 车身类
public class Framework {
   private Bottom bottom;

   public Framework(int size) {
      bottom = new Bottom(size);
   }

   public void init() {
      System.out.println("do
framework");
      // 车身的组建依赖于底盘
      bottom.init();
   }
}
```

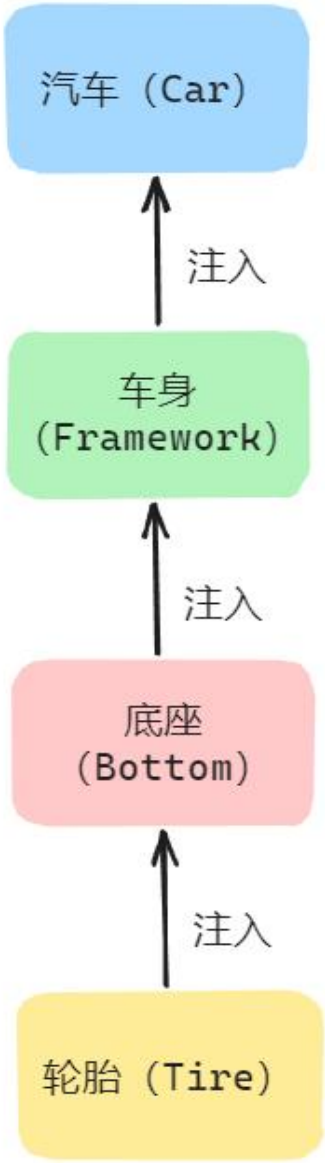
1.1.2 SSH架构

```
package old;

// 底盘类
public class Bottom {
    private Tire tire;

    public Bottom((int size) {
        tire = new Tire(size);
    }

    public void init() {
        System.out.println("do bottom");
        // 底盘的组建依赖于轮胎
        tire.init();
    }
}
```

```
package old;

// 轮胎类
public class Tire {
    private int size = 17; // 轮胎的尺寸

    public Tire(int size) {
        this.size = size;
    }

    public void init() {
        System.out.println("size -> " + size);
    }
}
```

```
package old;
public class Test {
    public static void main(String[] args) {
        Car car = new Car(20);
        car.init();
    }
```

1.1.2 SSH架构



1.1.2 SSH架构

```
package old;
/**
 * 传统开发方式, 耦合性问题
 */
// 汽车类
public class Car {
   // 汽车的组建依赖于车身的组建
   private Framework framework;

   public Car(Framework framework)
{
      this.framework = franmework;
   }
   public void init() {
      System.out.println("do car...");
      franmework.init();
   }
}
```

```
package old;

   // 车身的组建依赖于底盘
   private Bottom bottom;

   public Framework(Bottom bottom)
{
      this.bottom = bottom;
   }
   public void init() {
      System.out.println("do
franmework");
      bottom.init();
   }
}
```

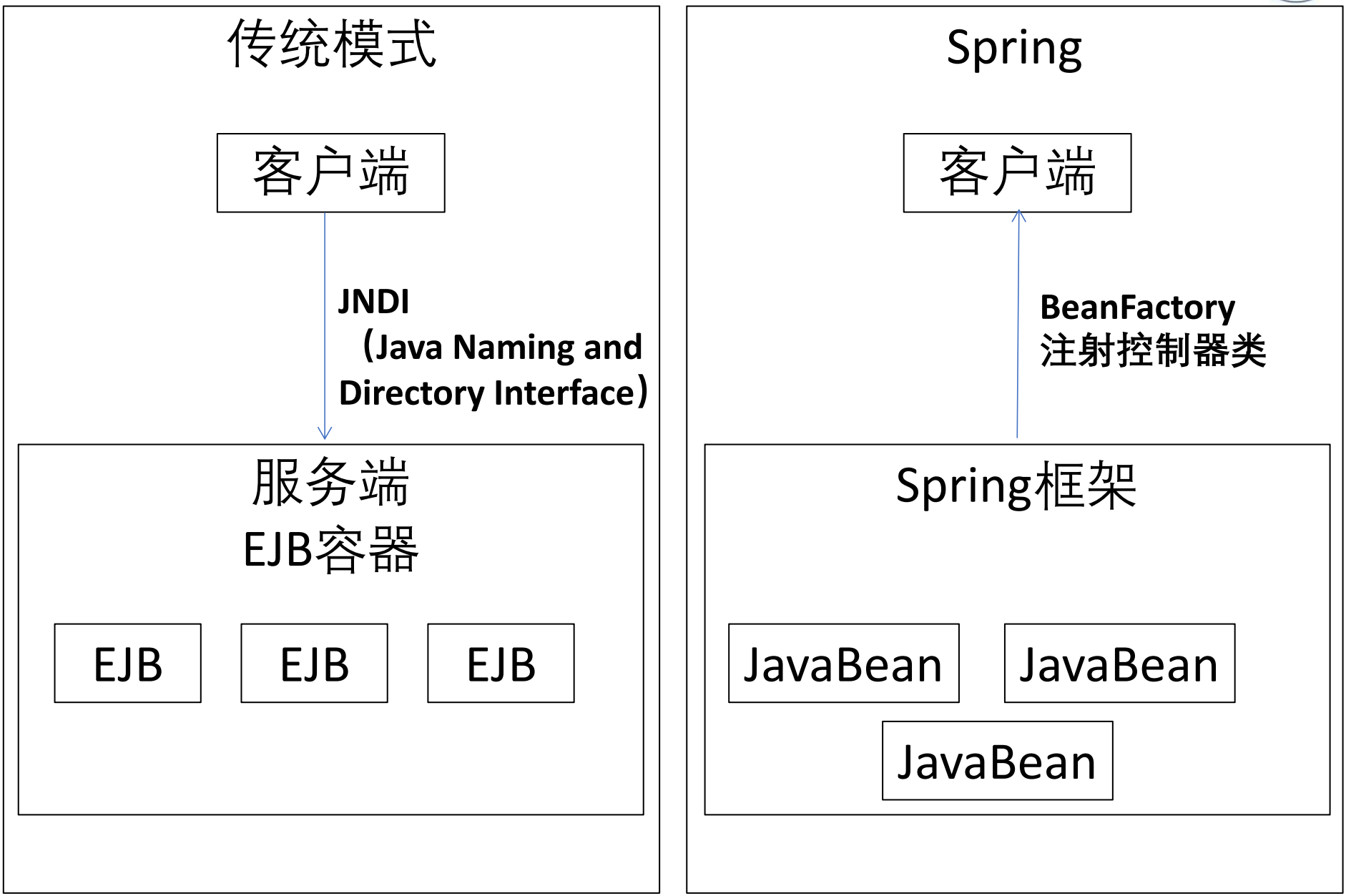
1.1.2 SSH架构

```
package old;
   // 底盘的组建依赖于轮胎
   private Tire tire;
   public Bottom(Tire tire) {
      this.tire = tire;    }
   public void init() {
      System.out.println("do bottom...");
      tire.init();  }  }
```

```
package ioc;
public class Tire {
   private int size = 17;
   private String color = "黑色";
   public Tire(int size, String color) {
      this.size = size;
      this.color = color;   }
   public void init() {
      System.out.println("size -> " + size);
      System.out.println("color->" +
color);   }   }
```

```
/** * 模拟 IoC 容器 */
public class IOCDemo {
   // 这里的内容就相当于是 IoC 容器做的事情
   // 对象的生命周期控制权就翻转给 IoC 容器了,
不再由程序员控制
   private Tire tire;
   private Bottom bottom;
   private Framework framework;
   public Car car;
   public IOCDemo() {
      tire = new Tire(17, "红色");
      bottom = new Bottom(tire);
      framework = new Framework(bottom);
       car = new Car(framework);   }   }
/** * 用户操作 */
public class Test {
   public static void main(String[] args) {
      // 直接使用, 创建就交给IoC了
      IOCDemo ioc = new IOCDemo();
      Car car = ioc.car;
      car.init();   } }
```

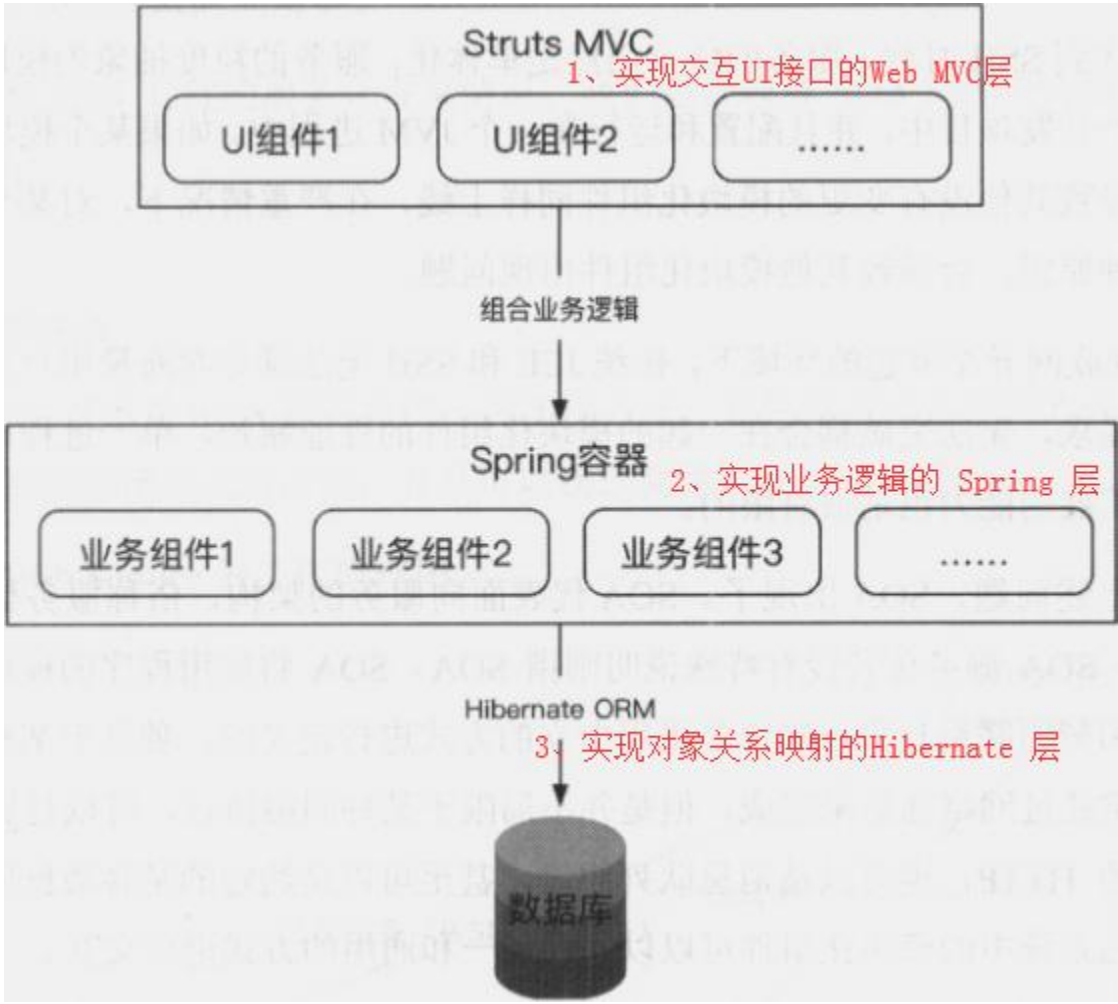
1.1.2 SSH架构


传统模式
客户端
JNDI
（Java Naming and Directory Interface）
服务端
EJB容器
EJB
EJB
EJB
Spring
客户端
BeanFactory
注射控制器类
Spring框架
JavaBean
JavaBean
JavaBean

SSH架构

UI交互的MVC层，业务逻辑的Spring层，对象关系映射的Hibernate层。

层级更简单、轻量，并支持单元测试，极大提高开发效率。



1.1 从传统单体架构到面向服务架构

1.1.1 JEE架构

1.1.2 SSH架构

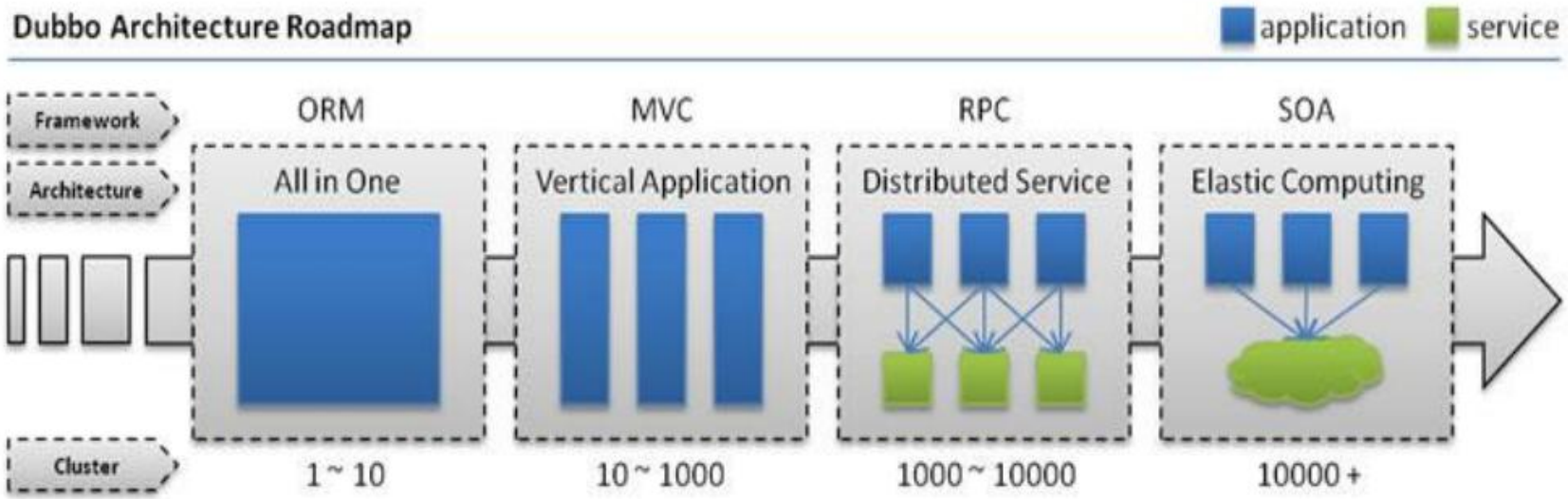
► 1.1.3 面向服务架构

面向服务架构

- JEE到SSH为止，服务特点仍然是单体化，服务的力度抽象为模块化组件，所有组件耦合在一个开发项目中，并配置和运行在同一JVM进程中。
- 单个模块化组件的升级上线会牵连到其他模块化组件，并可能触发问题。
- 另外，JEE和SSH无法满足对海量用户发起的高并发请求进行处理的需求，无法突破耦合在一起的模块化组件的性能瓶颈，水平扩展能力也十分有限。

1.1.3 面向服务架构

技术为业务而生，架构也为业务而出现。随着业务的发展、用户量的增长，系统数量增多，调用依赖关系也变得复杂，为了确保系统高可用、高并发的要求，系统的架构也从单体时代慢慢迁移至服务SOA时代，根据不同服务对系统资源的要求不同，我们可以更合理的配置系统资源，使系统资源利用率最大化。



1.1.3 面向服务架构

- 单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架\(ORM\) 是关键。

- 垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的 Web框架\(MVC\) 是关键。

- 分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐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。此时，用于提高业务复用及整合的 分布式服务框架\(RPC\) 是关键。

1.1.3 面向服务架构

- 流动计算架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的 资源调度和治理中心\(SOA\) 是关键。

平台随着业务的发展从 All in One 环境就可以满足业务需求（以Java来说，可能只是一两个war包就解决了）；发展到需要拆分多个应用，并且采用MVC的方式分离前后端，加快开发效率；再发展到服务越来越多，不得不将一些核心或共用的服务拆分出来，提供实时流动监控计算等，其实发展到此阶段，如果服务拆分的足够精细，并且独立运行，这个时候至少可以理解为SOA（Service-Oriented Architecture）架构了。

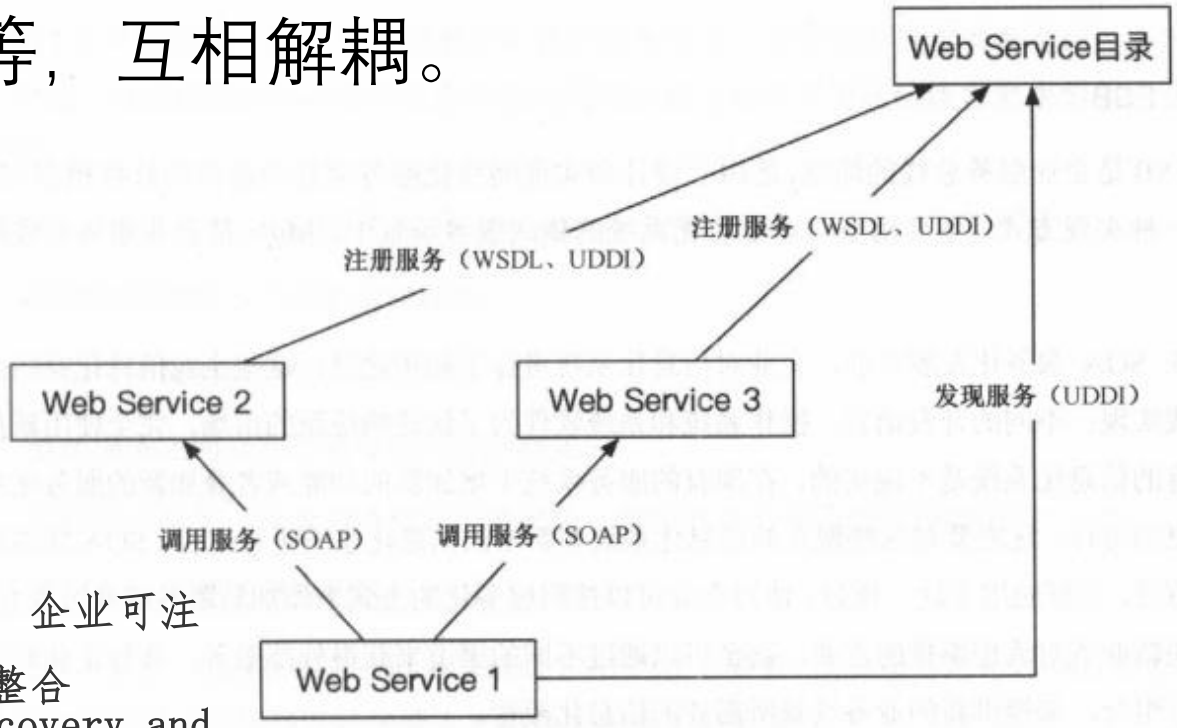
面向服务架构

- SOA：面向服务架构，SOA将应用程序的模块化组件通过定义明确的接口和契约联系起来。
- 接口是采用中立的方式进行定义，独立于语言，硬件和操作系统，通常通过网络通信完成，但也不局限于某种网络协议。
- 使得各种各样的系统中的模块化组件可以用一种统一和通用的方式进行交互。
- 对比JEE和SSH，SOA将模块化组件从单一进程中进一步拆分，形成独立的对外提供服务的网络化组件，每个网络化组件通过某种网络协议对外提供服务。

SOA实现方式： Web service

运行在不同机器和操作系统上的服务可以互相发现和调用，并通过某种协议交换数据。

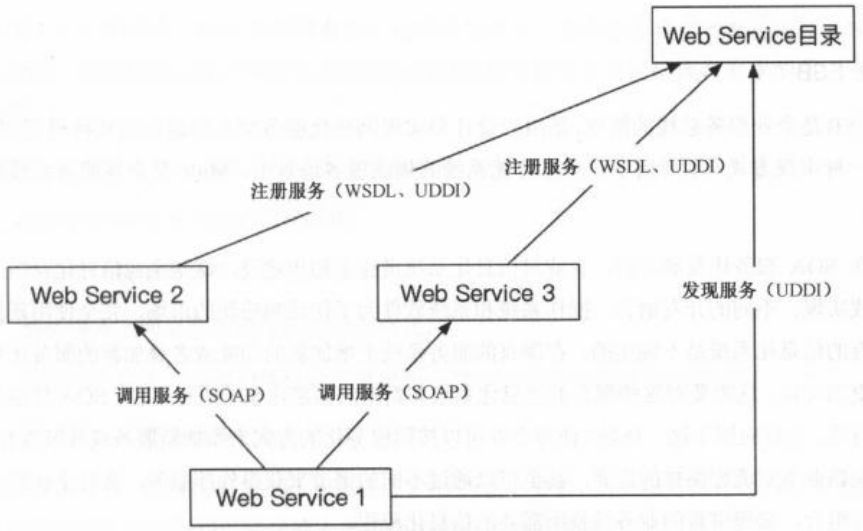
服务之间互相对等，互相解耦。



- UDDI 是一种目录服务，通过它，企业可注册并搜索 Web services。
- UDDI 指通用的描述、发现以及整合（Universal Description, Discovery and Integration）。
- UDDI 是一种由 WSDL 描述的网络服务接口目录。
- UDDI 经由 SOAP 进行通讯。

SOA实现方式： Web service

- 通过WSDL定义的服务发现接口进行访问，并通过SOAP协议进行通信。SOAP协议是一种在HTTP和HTTPS上传输XML数据来实现的协议，但每个服务都要依赖中心化Web Service目录来发现现存的服务。



- 服务提供者Web Service2和Web Service3通过UDDI协议将服务注册到Web Service目录服务中。
- 服务消费者Web Service1通过UDDI协议从Web Service目录中查询服务，并获得服务的WSDL服务描述文件。
- 服务消费者Web Service1通过WSDL语言远程调用和消费Web Service2和Web Service3提供的服务。

SOA实现方式：ESB

- ESB全称：企业服务总线，是SOA的另一种实现方式，主要用于企业信息化系统的集成服务场景中。
- ESB没有中心化的服务节点，每个服务提供者都是通过总线的模式插入系统，总线根据流程的编排负责将服务的输出进行转换并发送给流程要求的下一个服务。

ESB功能与职责

- 监控和控制服务之间的消息路由。
- 控制可插拔的服务化的功能和版本。
- 解析服务之间交互和通信的内容和格式。
- 通过组合服务、资源和消息处理器来统一编排业务需要的信息处理流程。
- 通过冗余来提供服务的备份能力。

第一章 微服务概述

1.1 从传统单体架构到面向服务架构

1.2 从面向服务架构到微服务

1.3 微服务架构概述

1.2 从面向服务架构到微服务

- 1.2.1 微服务架构的产生背景
- 1.2.2 微服务架构与传统单体架构的对比
- 1.2.3 微服务架构与SOA服务化的对比

从面向服务架构到微服务

- Web Service问题：
- 依赖中心化的服务发现机制。
- 使用SOAP通信协议，通常使用XML格式来序列化通信数据，XML格式的数据冗余太大，协议太重。
- 服务化管理和治理设施不完善。

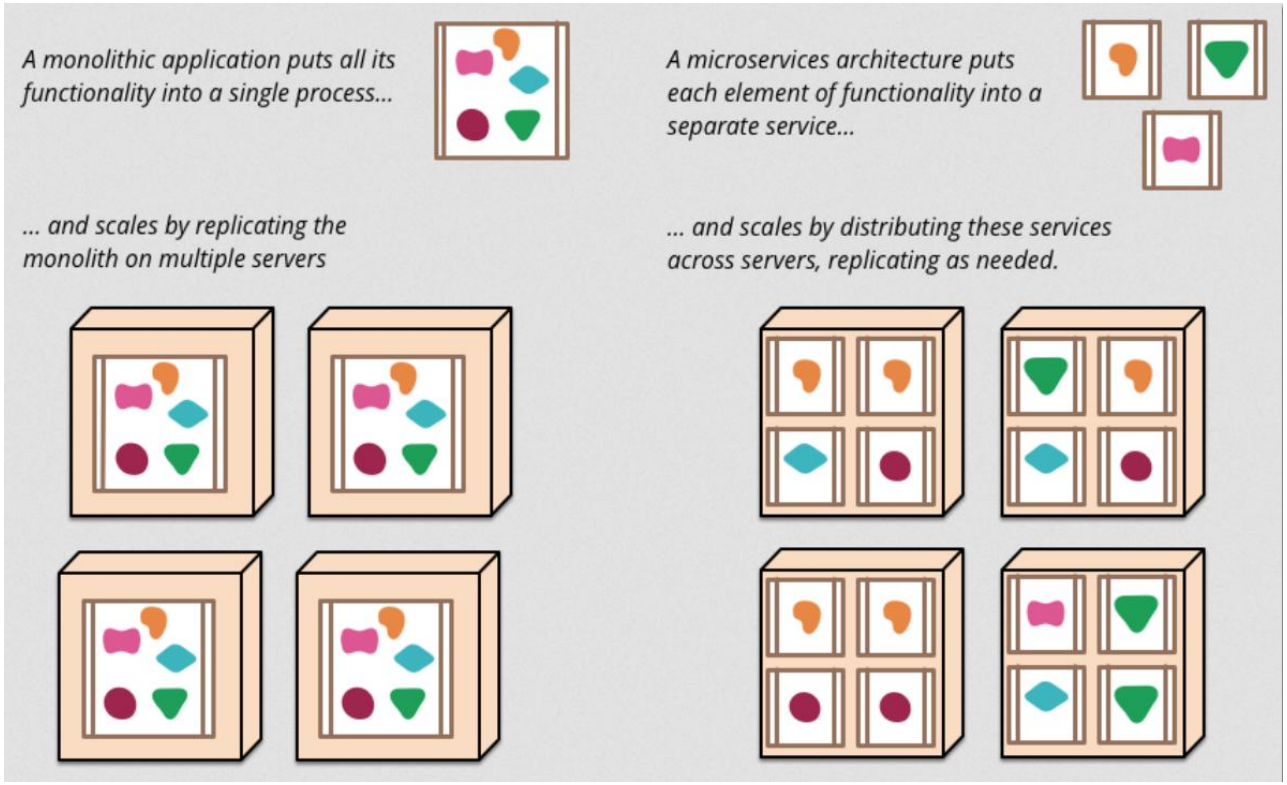
从面向服务架构到微服务

- ESB问题：
- ESB虽然是SOA实现的一种方式，却更多体现系统集成的便利性，通过统一服务总线组合在一起，并提供组合的业务流程服务。
- 组合在ESB上的服务本身可能是一个过重的整体服务或者是一个传统的JEE服务。
- ESB通过总线来隐藏系统内部的复杂性，但是系统内部的复杂性仍然存在。
- 对于总线本身的中心化的管理模型，系统变更影响的范围经常会随之扩大。

1.2.1 微服务架构的产生

什么是微服务架构

微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中各个微服务可独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表一个小的业务能力。



整体式架构

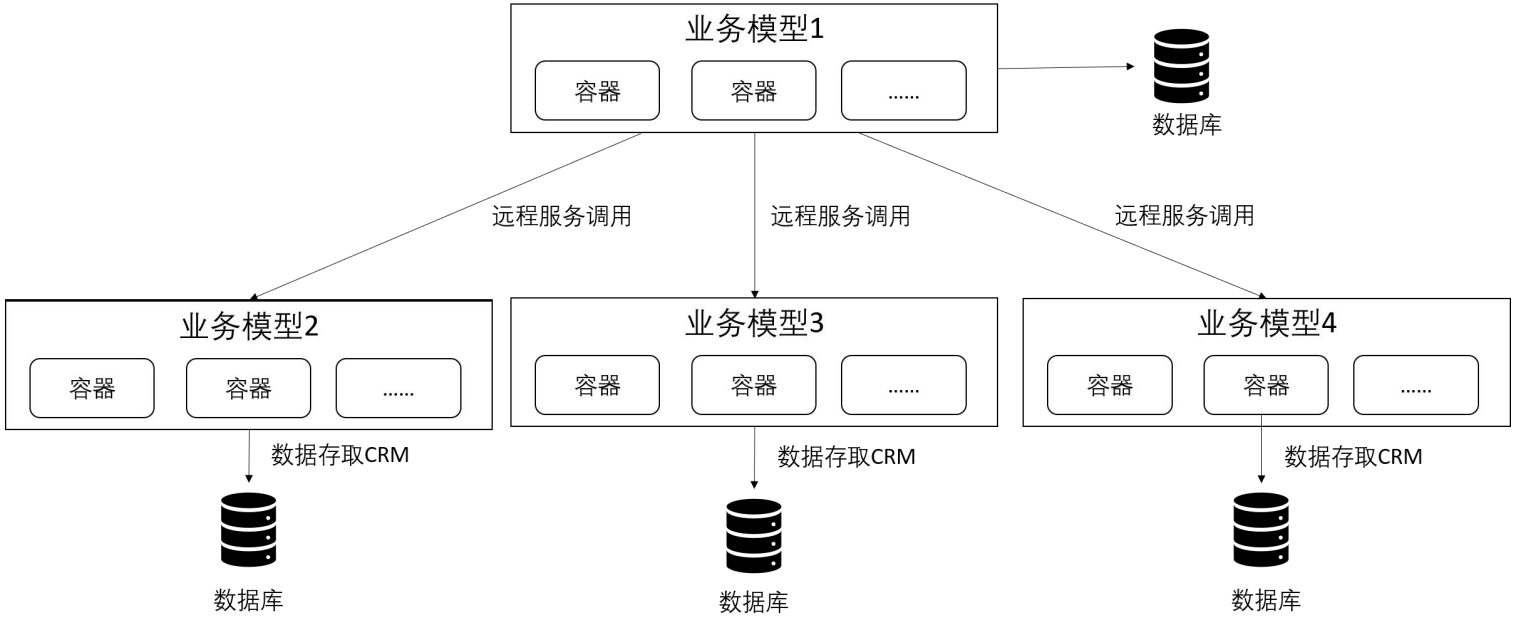
随着云化及应用功能变更越来越频繁，整体式应用在快速响应市场上显得越来越力不从心。

微服务架构

独立可部署及升级的微服务有助于大大提高系统变更的敏捷性

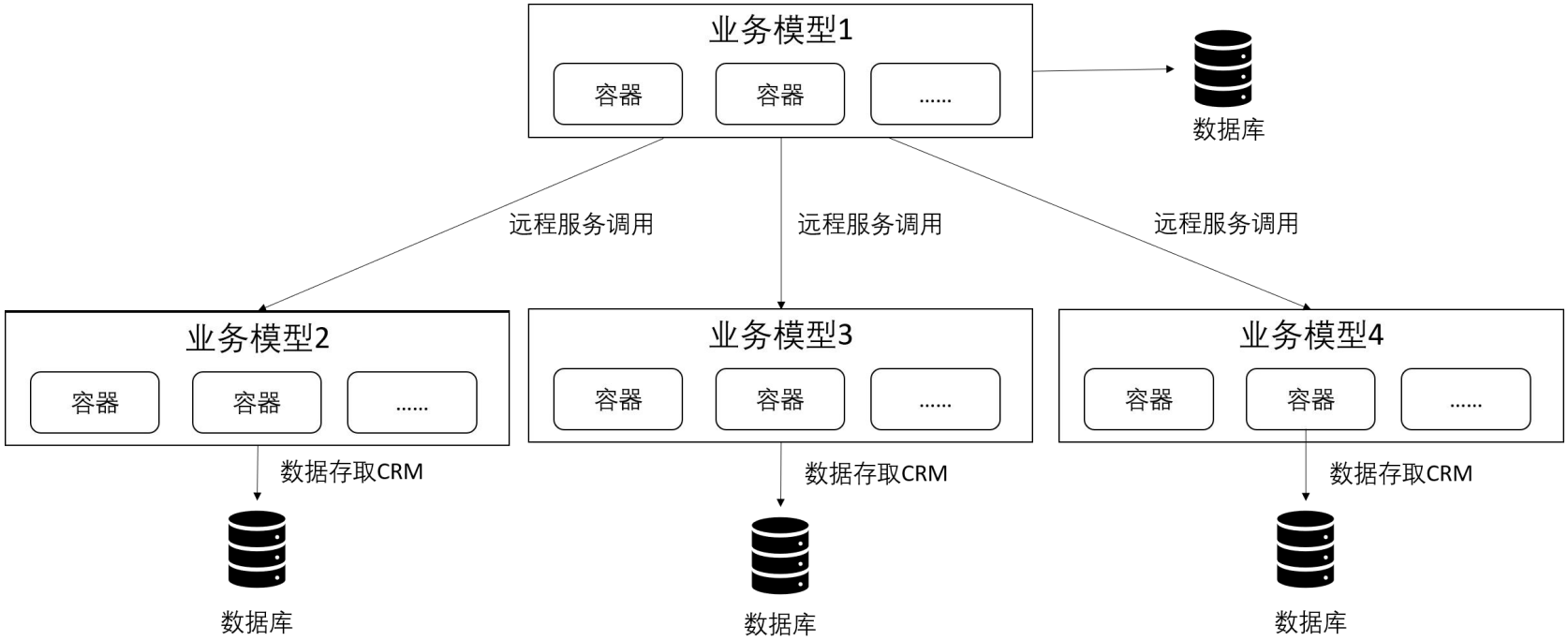
微服务架构

- 微服务把每一个职责单一的功能放在一个独立服务中。
- 每个服务运行在一个单独的进程中。
- 每个服务有多个实例在运行，每个实例可以运行在容器化平台内，达到平滑伸缩的效果。



1.2.1 微服务架构的产生

- 每个服务都有自己的数据存储，实际上，每个服务应该都有自己独享的数据库、缓存、消息队列等资源。
- 每个服务应该都有自己的运营平台，以及独享的运营人员，这包括技术运维和业务运营人员；每个服务都高度自治，内部的变化对外透明
- 每个服务都可根据性能需求独立地进行水平伸缩。



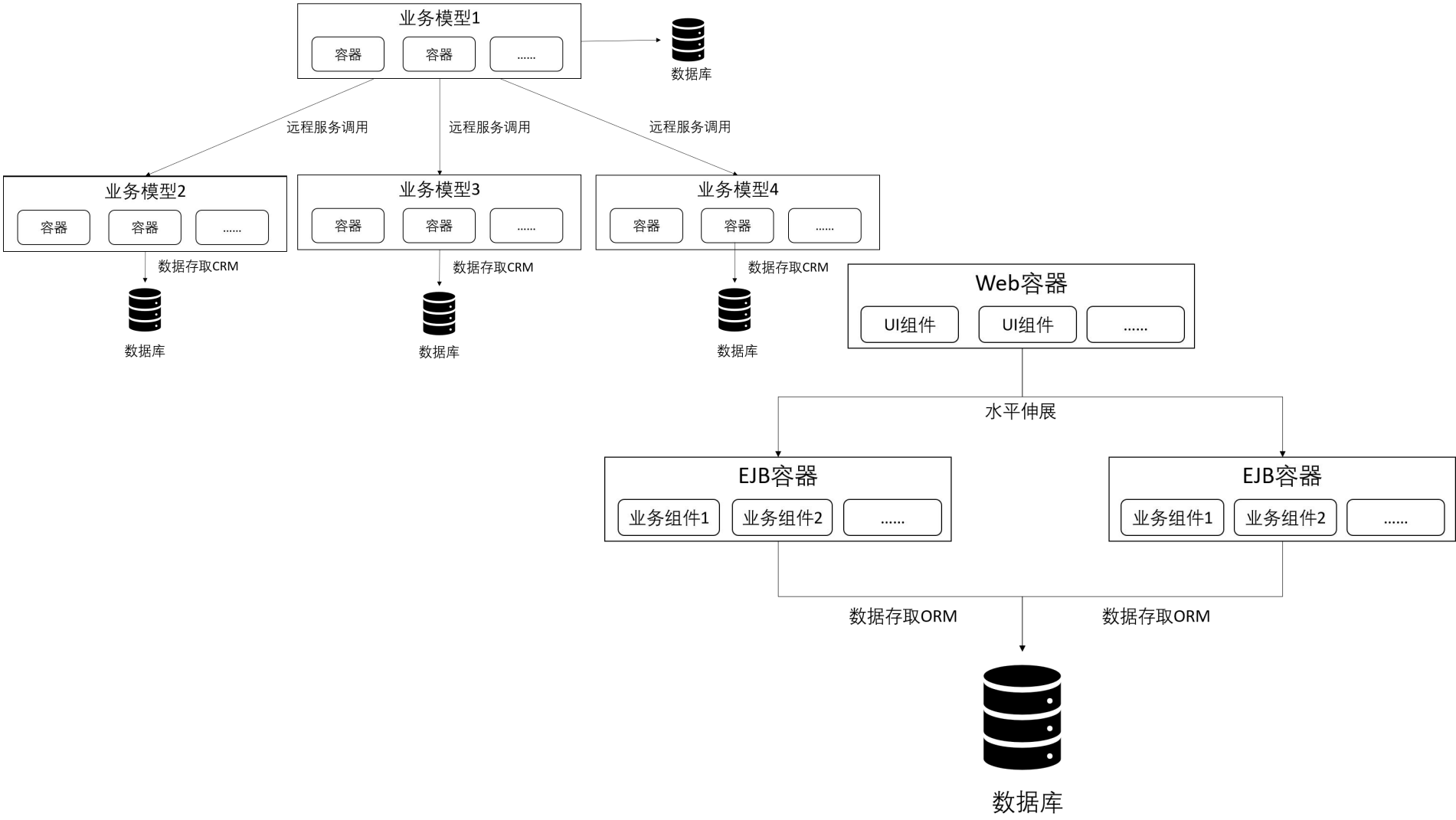
1.2 从面向服务架构到微服务

1.2.1 微服务架构的产生

► 1.2.2 微服务架构与传统单体架构的对比

1.2.3 微服务架构与SOA服务化的对比

微服务架构与传统单体架构的对比



微服务架构与传统单体架构的对比

- 传统单体架构将所有模块化组件混合后运行在同一JVM进程中。
- 可对包含多个模块化组件的整体JVM进程进行水平扩展，而无法对某个模块化组件进行水平话扩展。
- 某个模块化组件发生变化时，需要所有的模块化组件进行编译、打包和上线。
- 久而久之，模块间的依赖将会不清晰、互相耦合、互相依赖。

1.2 从面向服务架构到微服务

1.2.1 微服务架构的产生

1.2.2 微服务架构与传统单体架构的对比

► 1.2.3 微服务架构与SOA服务化的对比

微服务架构与SOA服务化的对比

- 微服务架构与SOA架构并不冲突，它们一脉相承。微服务架构是SOA架构相应特定历史时期的使用场景的延续，是SOA升华并进行落地的一种实现方式。
- SOA的服务理念在微服务架构中仍然有效，微服务在SOA的基础上进行了演进和叠加，形成了适合现代化应用场景的一个方法论。

微服务架构与SOA服务化的对比

- 目的不同：
- SOA服务化设计的范围更广一些，强调不同的异构服务之间的协作和契约，并强调有效集成，业务流程编排，历史应用集成等。
- 微服务使用一系列的细粒度服务来实现整体的业务流程，目的是有效地拆分应用，实现敏捷开发和部署，缩小变更和迭代的影响范围，并达到单一微服务更容易水平扩展到目的。

微服务架构与SOA服务化的对比

- 部署方式不同：
- 微服务将完整的应用拆分为多个细粒度的服务，通常使用敏捷扩容，缩容的Docker技术实现自动化的容器管理，每个微服务运行在单一的进程内，微服务中的部署互相独立，互不影响。
- SOA通常将多个业务服务通过组件化模块方式打包在一个War包里，然后统一部署在一个应用服务器上。

微服务架构与SOA服务化的对比

- 服务粒度不同：
- 微服务提倡将服务拆分为更细的粒度，通过多个服务组合来实现业务流程的处理，拆分到职责单一，甚至小到无法再进行拆分。
- SOA对粒度没有要求，在实践中服务通常是粗粒度的，强调接口契约的规范化，内部实现可以更粗粒度。

1.2.1 微服务架构的产生

功能	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、专注于功能交叉的团队
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

讨论

几种架构模式中哪种是最优的？
IOE 微服务

现在我们的系统处于在哪个阶段？有哪些问题？

微服务适合的应用场景

微服务：快速响应需求变化

船小好调头

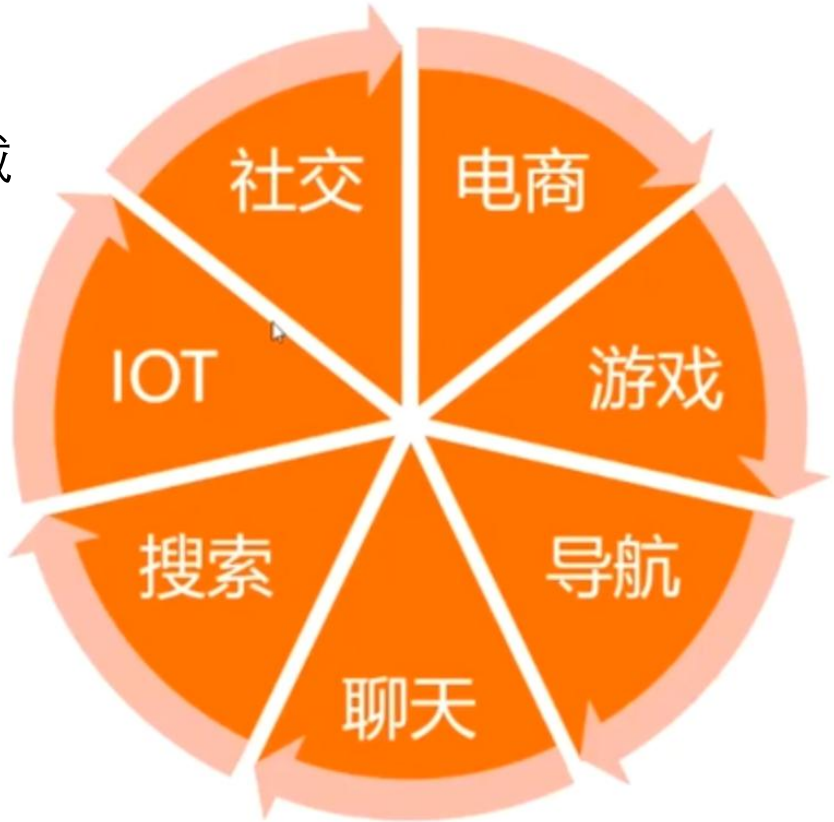
敏捷开发、敏捷运维DevOps

1.3.1 微服务架构的优缺点

微服务典型场景

微服务架构广泛应用：移动互联网的时代因素

较为适合微服务的领域



微服务架构开发：需要比单体和SOA更多的开发人员，面临更多新的问题

第一章 微服务概述

1.1 从传统单体架构到面向服务架构

1.2 从面向服务架构到微服务

1.3 微服务架构简介

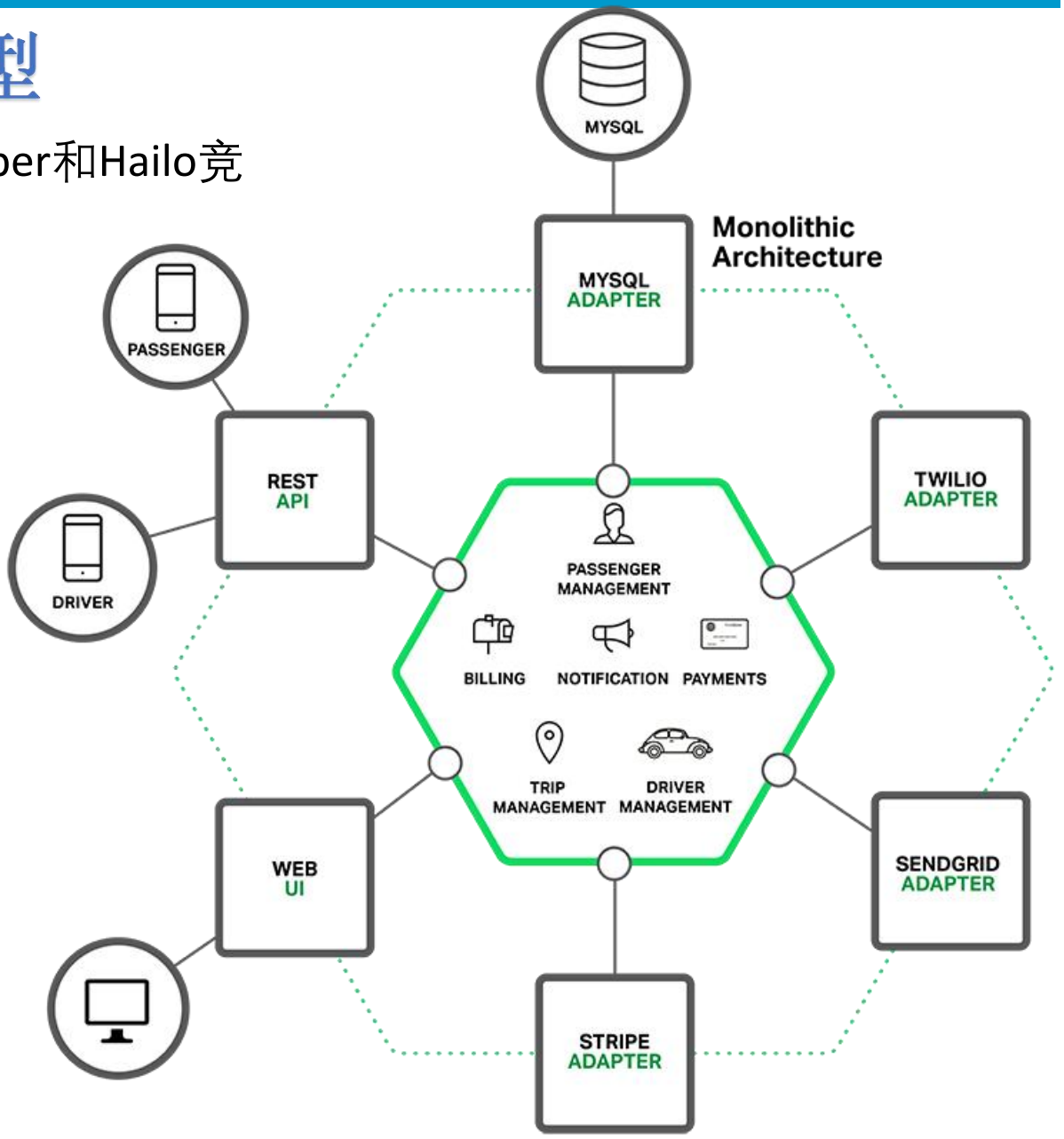
1.3 微服务架构简介

- 1.3.1 微服务架构的特点及优缺点
- 1.3.2 微服务架构企业
- 1.3.3 微服务架构改造案例

构建单体应用模型

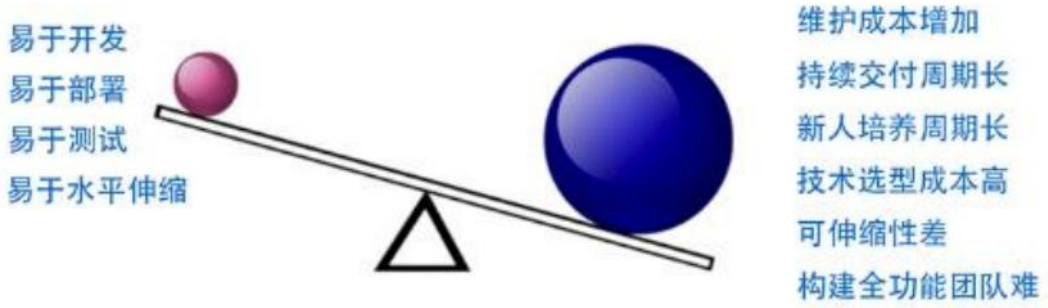
假设你正准备研发一款与Uber和Hailo竞争的出租车调度软件。

- 应用核心是业务逻辑，由定义服务、域对象和事件的模块完成
- 尽管也是模块化逻辑，但是最终它还是会打包并部署为单体式应用
- 许多Java应用会被打包为WAR格式，部署在Tomcat或者Jetty上



1.2.3 微服务架构与SOA服务化的对比

开发单体式应用的优势



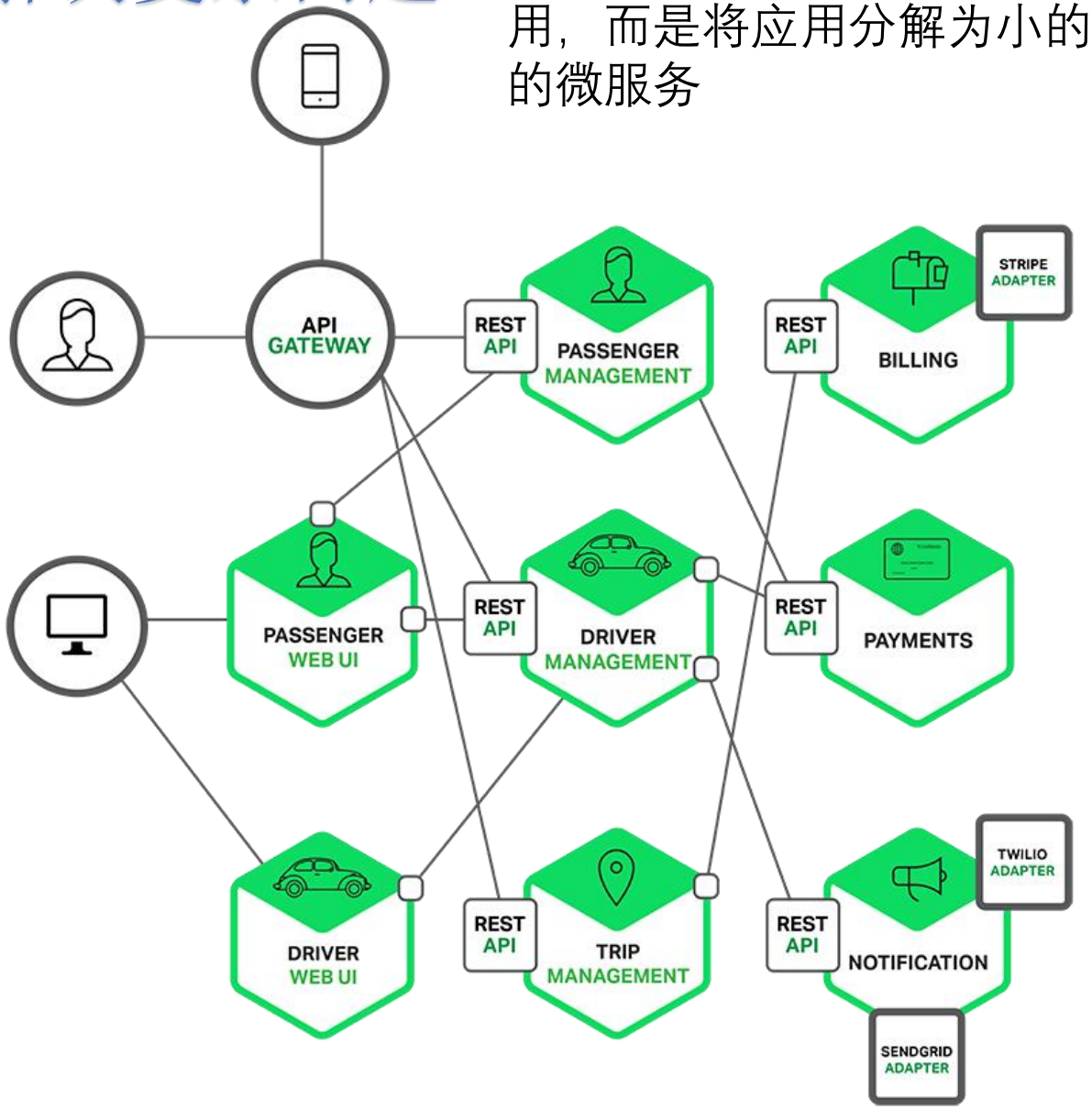
- ✓ 易于开发:现有的大部分工具、应用服务器、框架都是这类单块架构应用程序，为人所熟知。IDE都能够有效加载并配置整个应用程序的依赖，方便开发人员开发、运行、调试等。
- ✓ 易于测试：所有的功能都运行在一个进程中。
- ✓ 易于部署：由于所有的功能最终都会达成一个包，因此只需复制该软件包到服务器相应的位置即可。
- ✓ 易于水平伸缩：运行在一个进程中，因此单块架构的水平伸缩，更确切的理解其实是克隆。

开发单体式应用的不足：走向单体地狱

- 庞大、复杂的单体，敏捷开发和交付的任何一次尝试都将原地徘徊；
- 应用程序越大，启动时间越长；
- 高耦合度；
- 复杂的单体应用本身就是持续部署的障碍；
- 当不同模块存在资源需求冲突时，单体应用可能难以扩展；
- 单体应用的另一个问题是可靠性；
- 单体应用使得采用新框架和语言变得非常困难；
- 低并发性。

微服务-解决复杂问题

其思路不是开发一个巨大的单体式的应用，而是将应用分解为小的，互相连接的微服务

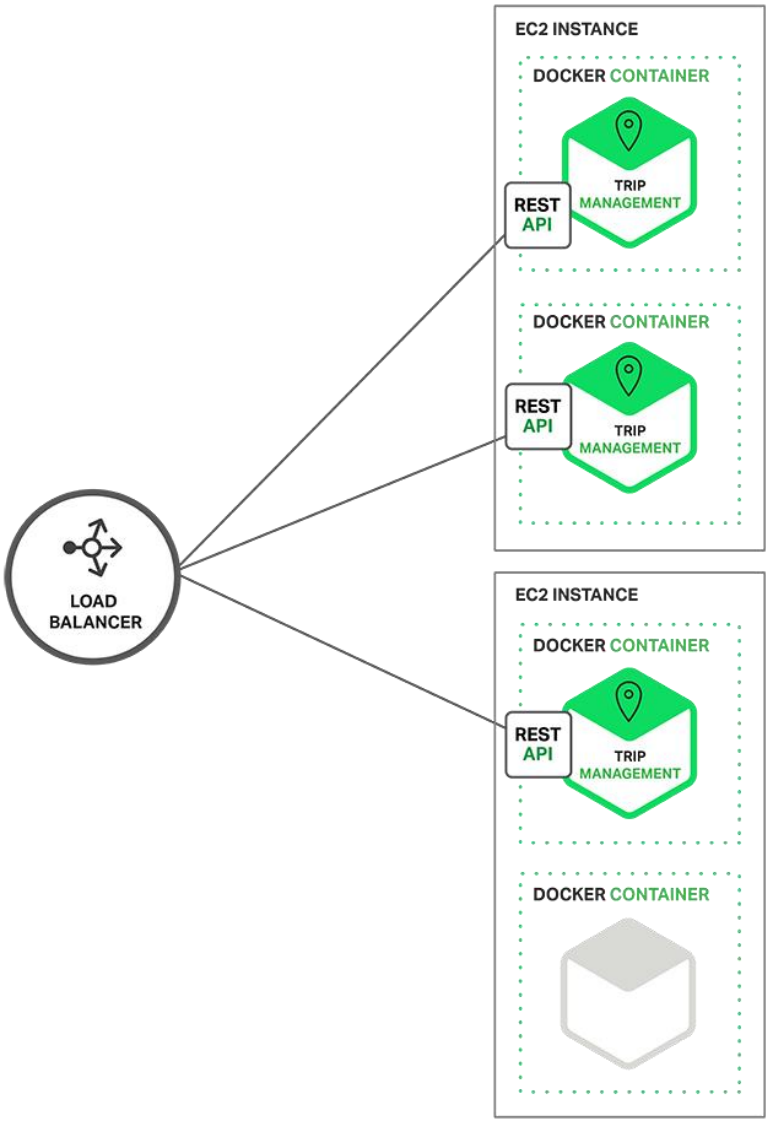


1.2.3 微服务架构与SOA服务化的对比

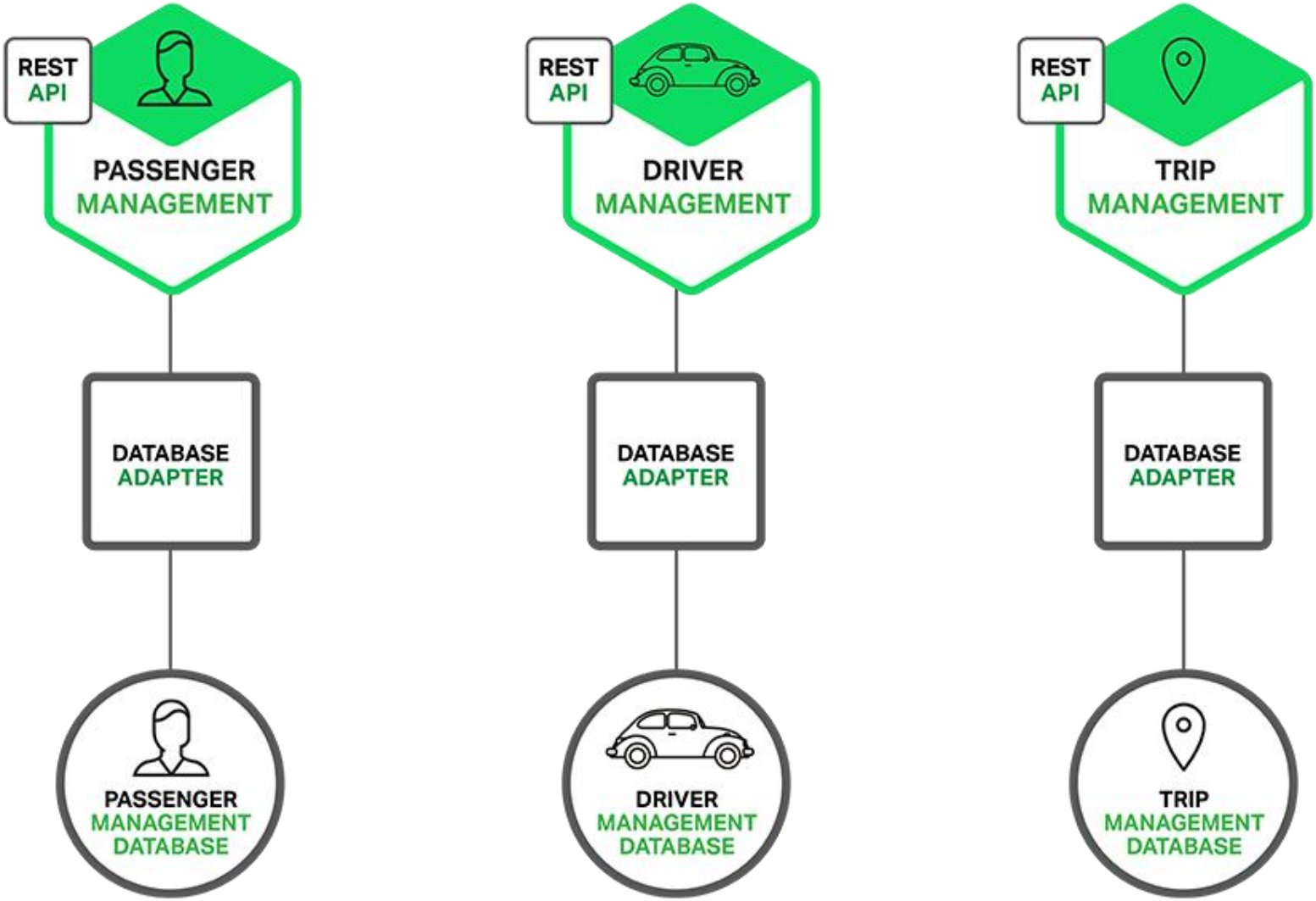
微服务架构——处理复杂事物

- ✓ 一个微服务一般完成某个特定的功能，比如下单管理、客户管理等等
- ✓ 每一个微服务都是微型六角形应用，都有自己的业务逻辑和适配器，有些还会提供API给其他服务
- ✓ 运行时，每个实例可能是一个云VM或者是Docker容器
- ✓ 微服务架构每个服务都有自己的数据库

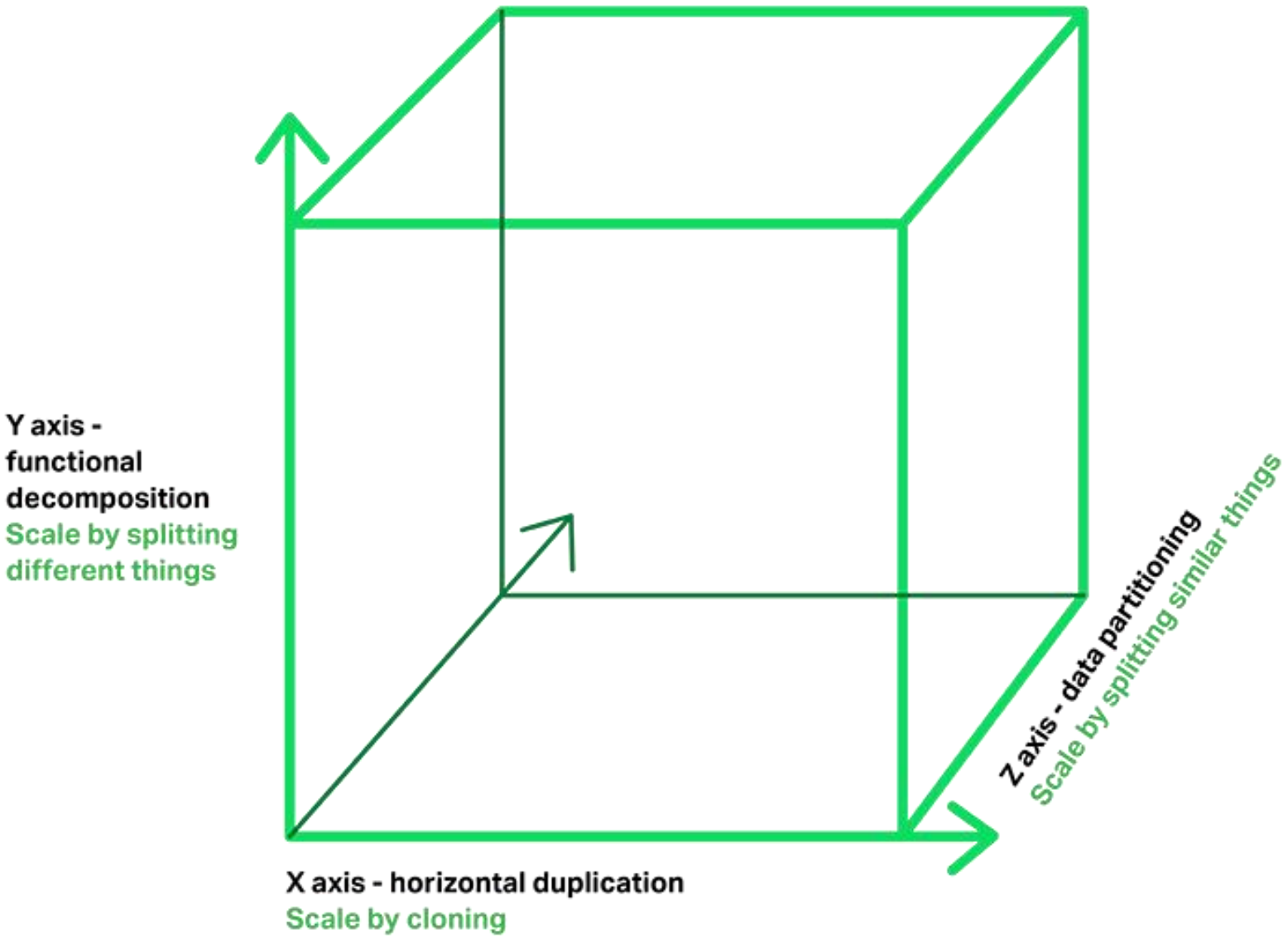
使用 Docker 部署服务



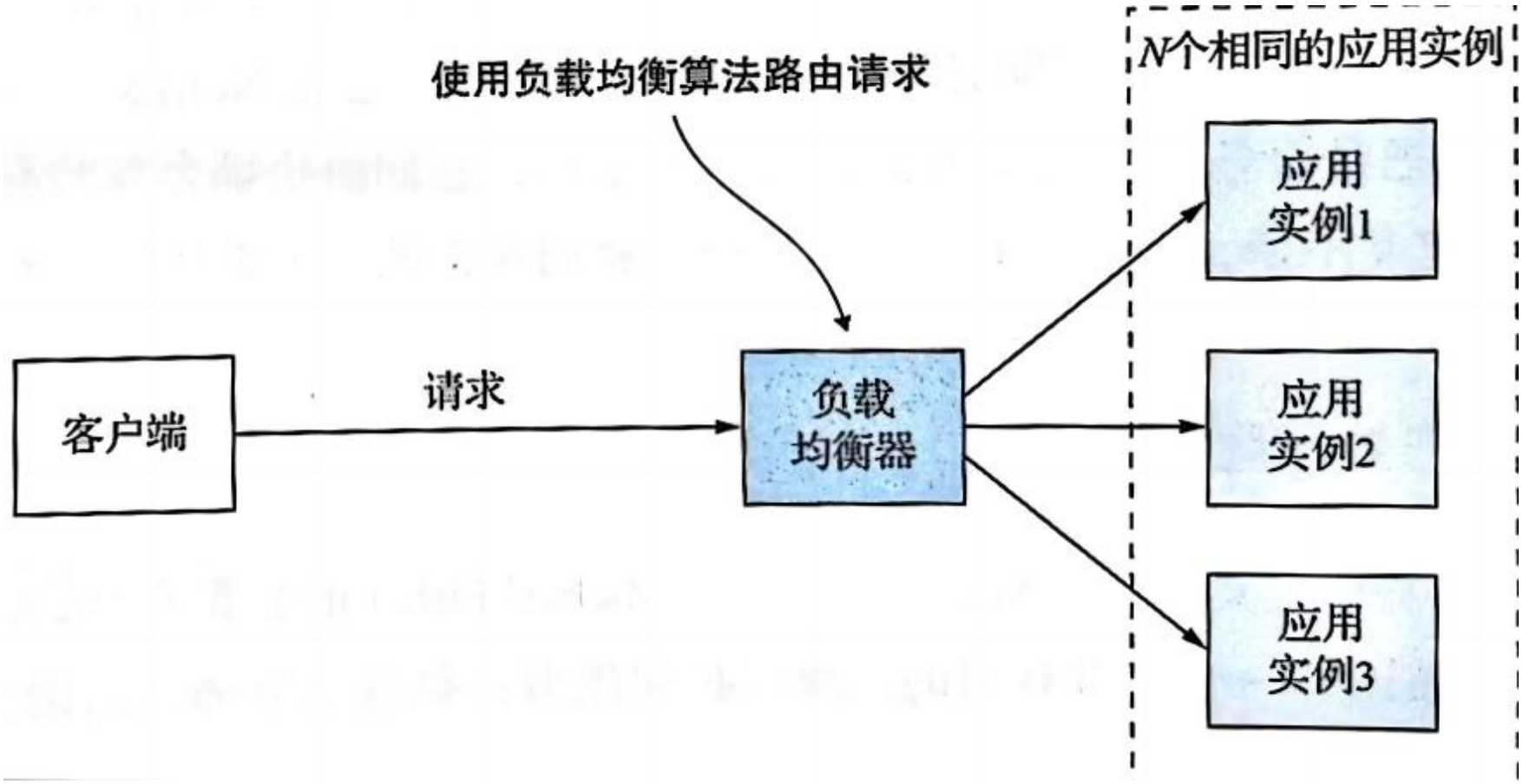
数据库架构示例应用程序



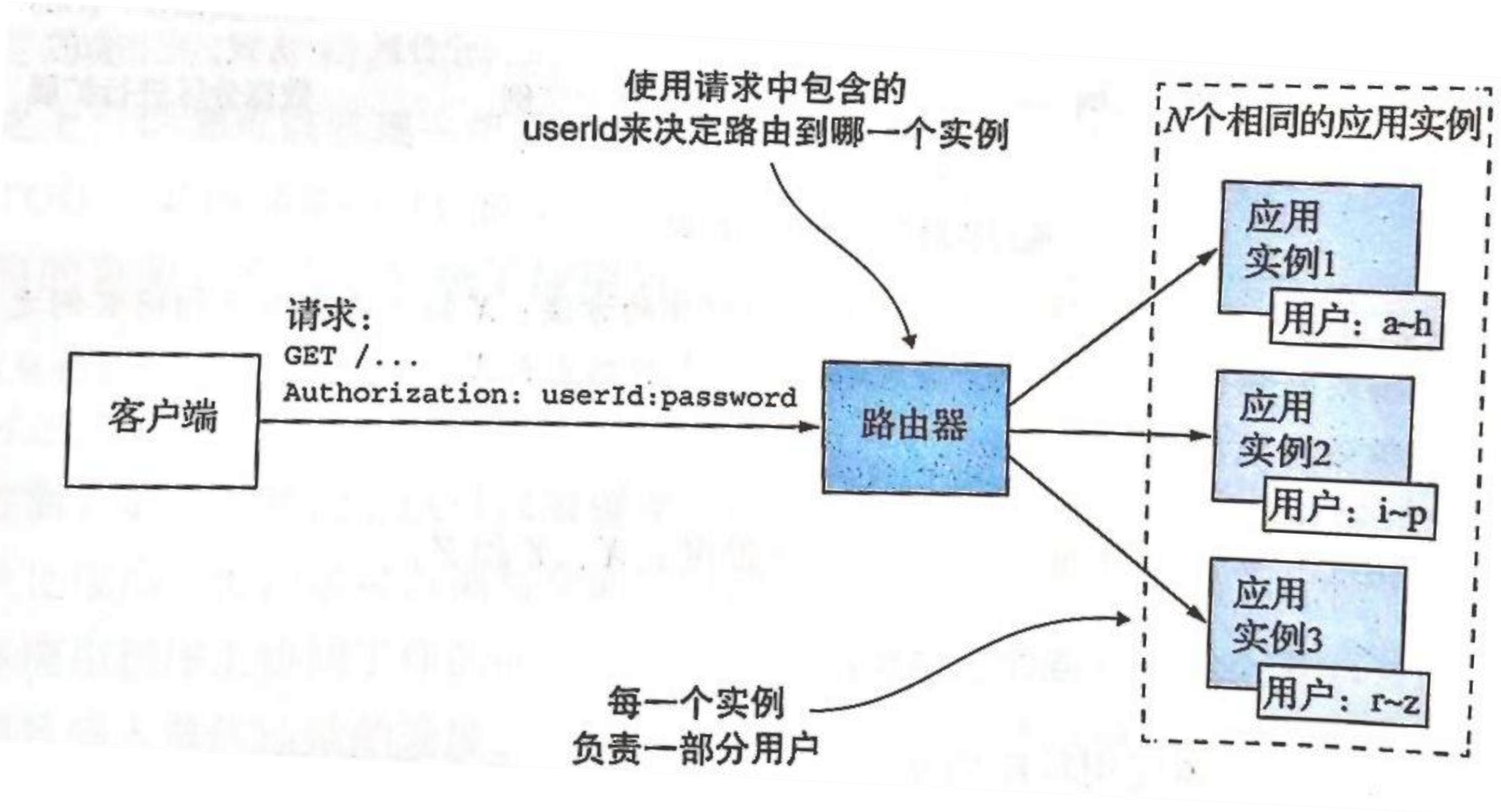
扩展立方体

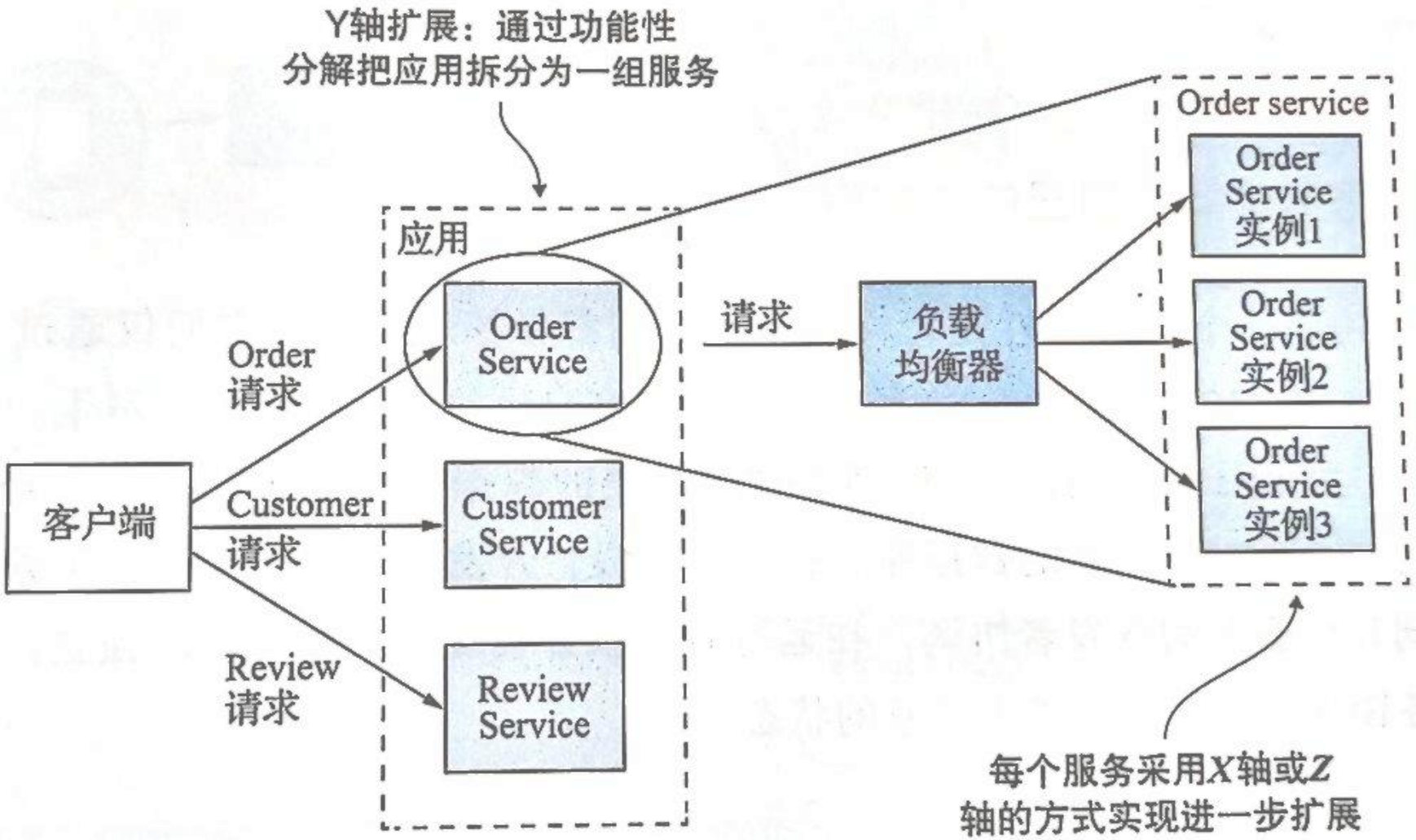

Y axis -
functional
decomposition
Scale by splitting
different things
X axis - horizontal duplication
Scale by cloning
Z axis - data partitioning
Scale by splitting similar things

x轴扩展：在多个实例之间实现请求的负载均衡。



Z轴扩展：根据请求的属性路由请求。




Y轴扩展：通过功能性
分解把应用拆分为一组服务
应用
Order Service
Customer Service
Review Service
客户端
Order
请求
Customer
请求
Review
请求
请求
负载
均衡器
Order service
Order Service 实例1
Order Service 实例2
Order Service 实例3
每个服务采用X轴或Z
轴的方式实现进一步扩展

1.3.1 微服务架构的优缺点

微服务架构的优点

第一，它解决了复杂问题。它把可能会变得庞大的单体应用程序分解成一套服务。虽然功能数量不变，但是应用程序已经被分解成可管理的块或者服务。每个服务都有一个明确定义边界的方式，如远程过程调用（RPC）驱动或消息驱动 API。微服务架构模式强制一定程度的模块化，实际上，使用单体代码来实现是极其困难的。因此，使用微服务架构模式，个体服务能被更快地开发，并更容易理解与维护。

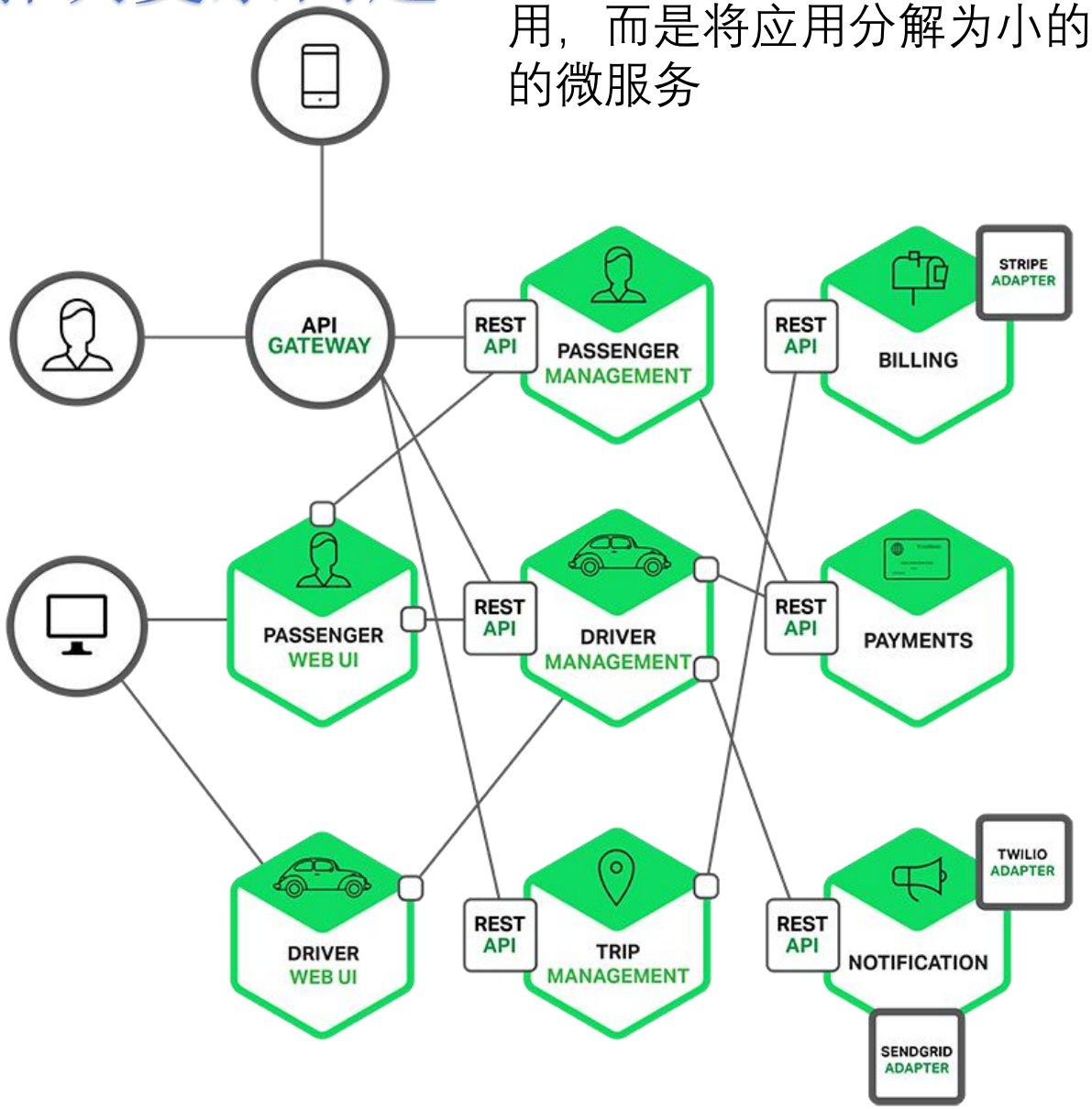
第二，这种架构使得每个服务都可以由一个团队独立专注开发。开发者可以自由选择任何符合服务 API 契约的技术。当然，更多的组织是希望通过技术选型限制来避免完全混乱的状态。然而，这种自由意味着开发人员不再有可能在这种自由的新项目开始时使用过时的技术。当编写一个新服务时，他们可以选择当前的技术。此外，由于服务较小，使用当前技术重写旧服务将变得更加可行。

第三，微服务架构模式可以实现每个微服务独立部署。开发人员根本不需要去协调部署本地变更到服务。这些变更一经测试即可立即部署。比如，UI 团队可以执行 A|B 测试，并快速迭代 UI 变更。微服务架构模式使得持续部署成为可能。

最后，微服务架构模式使得每个服务能够独立扩展。您可以仅部署满足每个服务的容量和可用性约束的实例数目。此外，您可以使用与服务资源要求最匹配的硬件。例如，您可以在 EC2 Compute Optimized 实例上部署一个 CPU 密集型图像处理服务，并且在 EC2 Memory-optimized 实例上部署一个内存数据库服务。

微服务-解决复杂问题

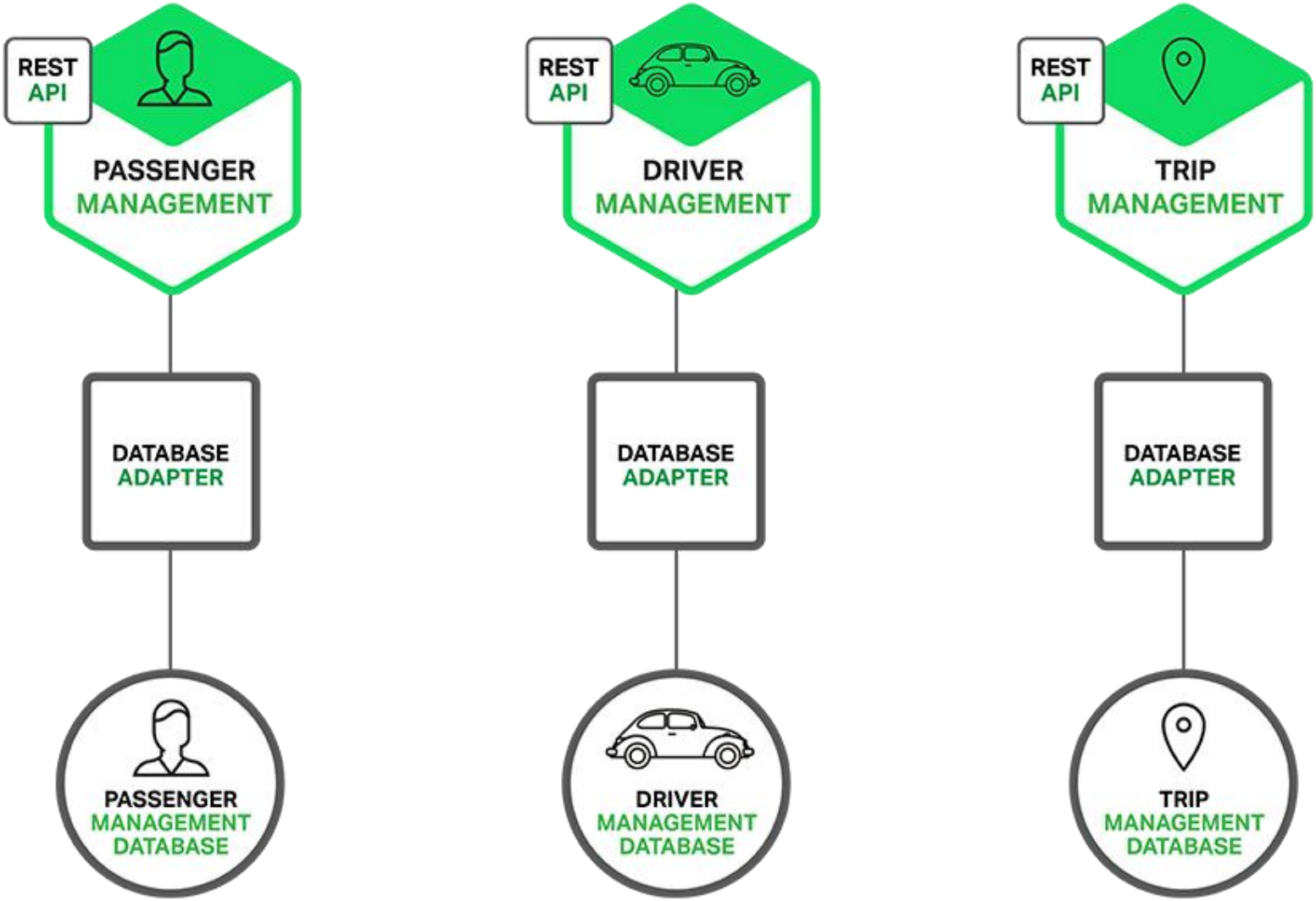
其思路不是开发一个巨大的单体式的应用，而是将应用分解为小的，互相连接的微服务



微服务架构缺点 微服务不是银弹

- 首先，微服务的边界、大小很难界定（什么时候该拆分，什么时候该合并）
- 第二，微服务应用是分布式系统，由此会带来固有的复杂性。开发者需要在RPC或者消息传递之间选择并完成进程间通信机制
- 第三，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或者缓存。使用最终一致性、分布式事务等等
- 第四，测试一个基于微服务架构的应用也是很复杂的任务
- 每个服务都有多个运行时实例。还有更多的移动部件需要配置、部署、扩展和监控
- 最后，微服务架构模式应用的改变将会波及多个服务。需要了解所有服务使用者。

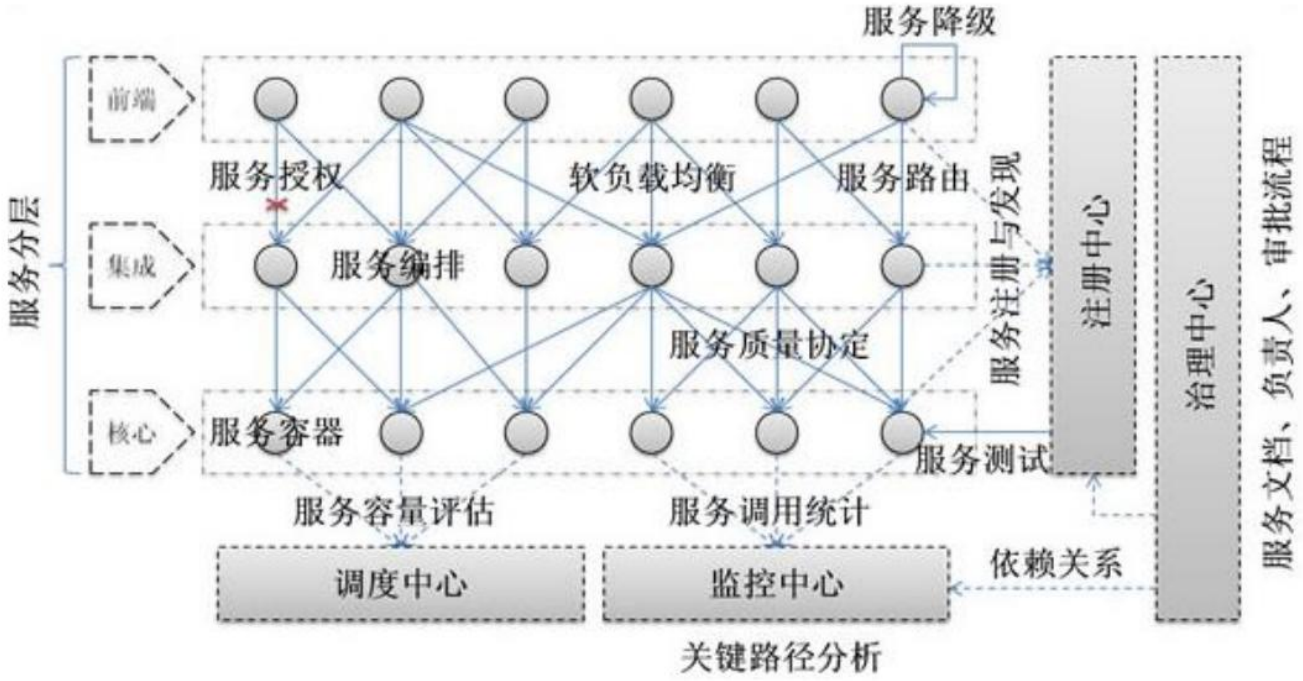
数据库架构示例应用程序



1.3.1 微服务架构的优缺点

服务治理-解决服务膨胀之后的问题

服务间依赖关系变得错综复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完成地描述应用的架构关系



常见的微服务组件及概念

- **服务注册**，服务提供方将自己调用地址注册到服务注册中心，让服务调用方能够方便找到自己。
- **服务发现**，服务调用方从服务注册中心找到自己需要调用的服务的地址。
- **负载均衡**，服务提供方一般以多实例的形式提供服务，负载均衡功能能够让服务调用方连接到合适的服务节点。并且，节点选择的工作对服务调用方来说是透明的。
- **服务网关**，服务网关是服务调用的唯一入口，可以在这个组件是实现用户鉴权、动态路由、灰度发布、A/B测试、负载限流等功能。
- **配置中心**，将本地化的配置信息(properties, xml, yaml等)注册到配置中心，实现程序包在开发、测试、生产环境的无差别性，方便程序包的迁移。
- **API管理**，以方便的形式编写及更新API文档，并以方便的形式供调用者查看和测试。

常见的微服务组件及概念

- **调用链**，记录完成一个业务逻辑时调用到的微服务，并将服务间串行或并行的调用关系展示出来。在系统出错时，可以方便地找到出错点。

- **支撑平台**，系统微服务化后，系统变得更加碎片化，系统的部署、运维、监控等都比单体架构更加复杂，那么，就需要将大部分的工作自动化。现在，可以通过Docker等工具来中和这些微服务架构带来的弊端。

1.3 微服务架构简介

1.3.1 微服务架构的优缺点

▶ 1.3.2 微服务架构企业

1.3.3 微服务架构改造案例

微服务典型企业

1. Google
2. Amazon
3. 阿里巴巴
4. 腾讯
5. 蚂蚁金服
6. 今日头条
7. 高德地图
8. 新浪微博
9. 京东
10. 拼多多
11. 滴滴打车
12. 360
13. 网易
14. 陌陌
15. 微信
16. 哈罗单车

1. Netflix（每天20亿）
2. Uber
3. Facebook
4. Twitter
5. Jelastic
6. Nirmata
7. nearForm
8. Riot Games
9. SoundCloud
10. Uber
11. Joined Node
12. Bilibili

1.3.2 微服务架构企业

- Netflix
- 阿里巴巴
- 蚂蚁金服
- 腾讯
- 京东
- 中国平安
- 亚马逊
- IBM
- 滴滴打车
- 哈罗出行

- 今日头条
- 美团
- 小米
- 饿了吗
- 拼多多
- 携程
- 趣头条
- 网易
- 陌陌
- 百度

1.3 微服务架构概述

1.3.1 微服务架构的优缺点

1.3.2 微服务架构企业

▶ 1.3.3 微服务架构改造案例

1.3.3 微服务架构改造案例

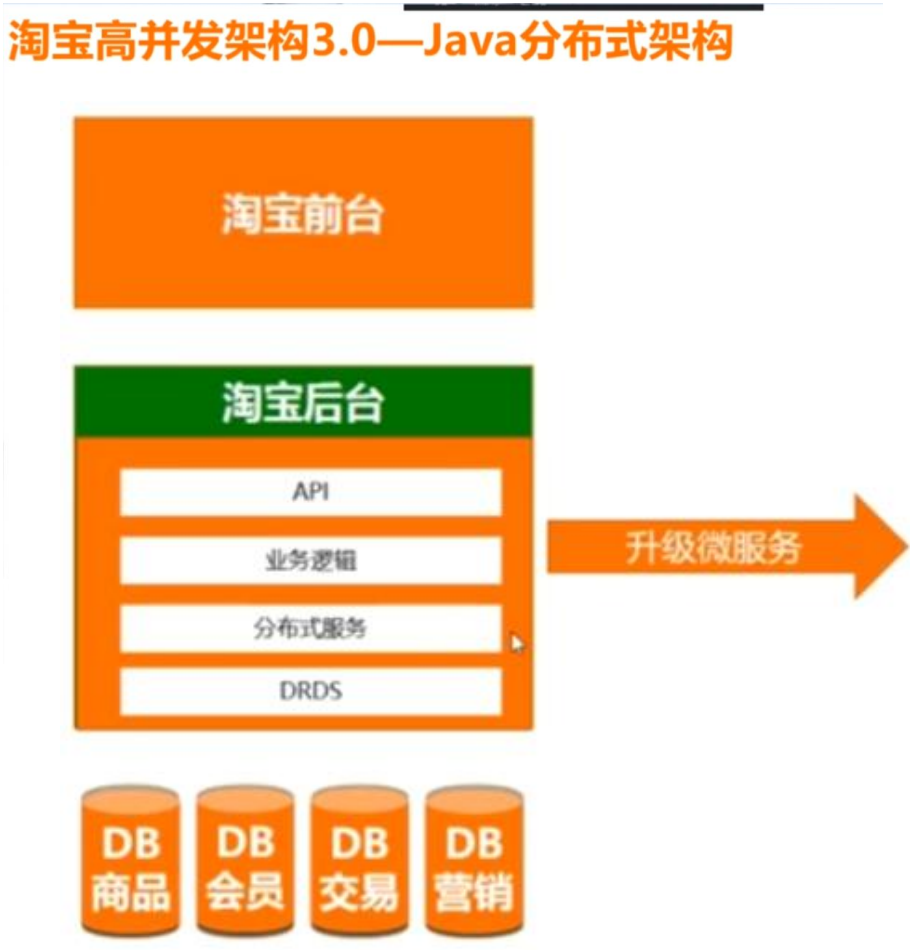
淘宝高并发架构1.0—PHP+MySQL

1.0:使用IOE架构,
满足安全性/稳定性/高并发

2.0:使用Java的开源,
廉价的方法解决高并发问题

转java

1.3.3 微服务架构改造案例



过渡主要原因：淘宝业务增加

时代背景：移动互联网时代

1.3.3 微服务架构改造案例