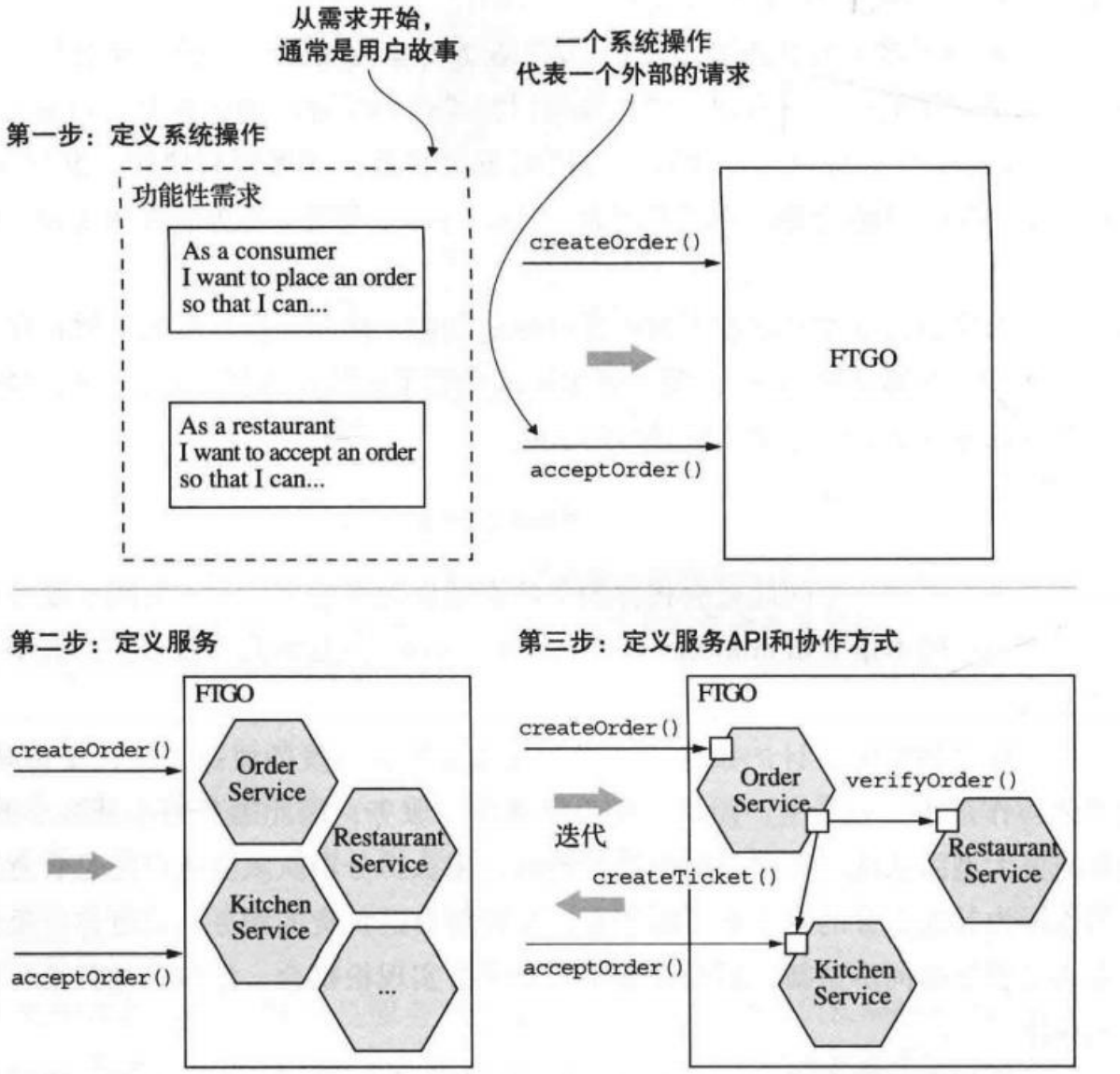
# 第四章 服务设计原则

# 为应用程序定义微服务架构

**定义应用程序架构的三步式流程**

系统操作（system operation)是应用程序必须处理的请求的一种抽象描述。



# 为应用程序定义微服务架构

## 第一步：定义系统操作

- 将需求提炼为各种关键请求
- 不是根据特定的进程间通信技术来描述请求，而是使用更抽象的系统操作的概念
- 系统操作是更新数据的命令，也是检索数据的查询
- 每个命令的行为都是根据抽象领域模型定义的，抽象领域模型也是从需求中派生的
- 系统操作是描述服务之间协作方式的架构场景

# 第二步：确定如何分解服务

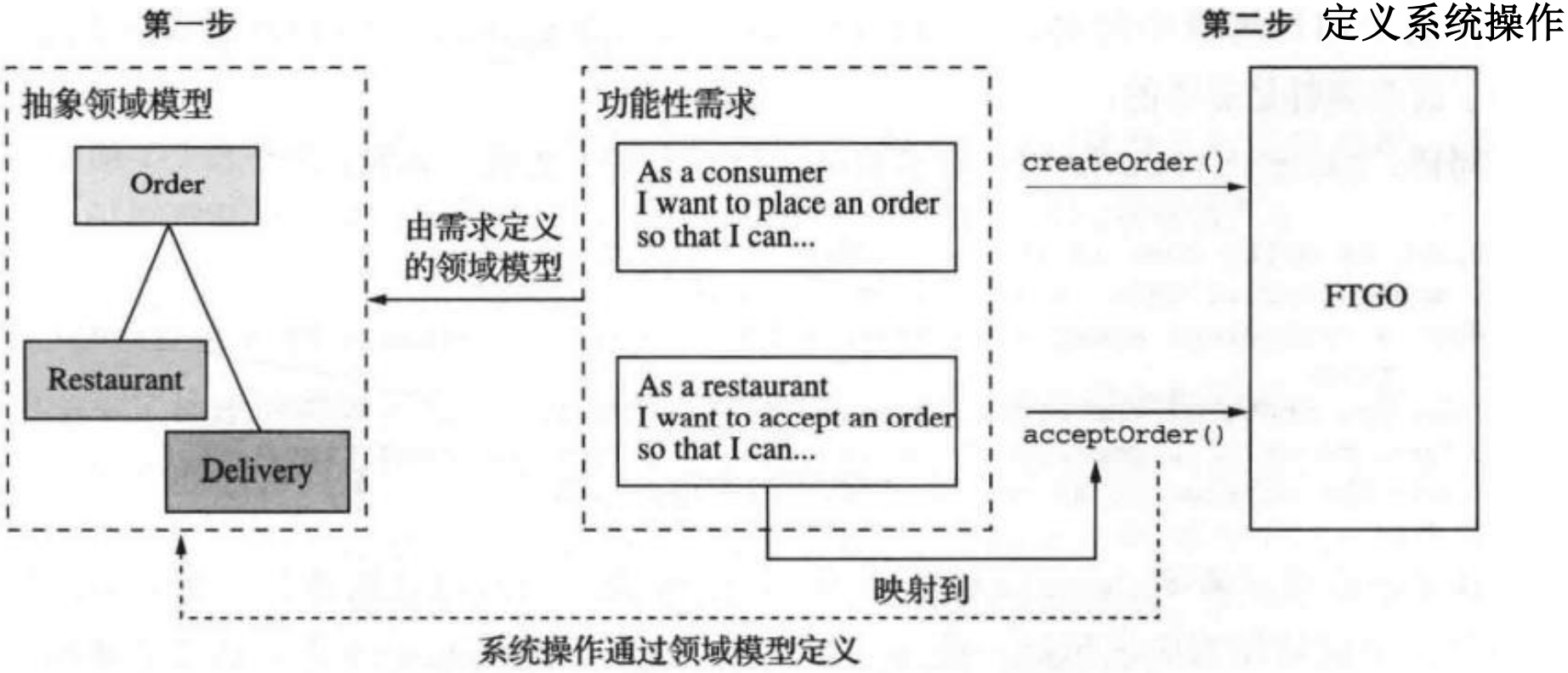
- 源于业务架构学派的策略，定义了与业务能力相对应的服务
- 围绕领域驱动设计的子域来分解和设计服务
- 最终结果都是围绕业务概念而非技术概念分解和设计服务

## 第三步：确定每个服务的API

- 将标识的每个系统操作分配给服务，服务可以完全独立地实现操作，或者与其他服务协作
- 确定服务的协作方式，需要服务来支持其他操作
- 确定哪种进程间通信机制来实现每个服务的API

# 识别系统操作

- 起点是应用程序的需求，包括用户故事及其相关的用户场景
- 两步式流程识别和定义系统操作



# 识别系统操作

- 领域模型主要源自用户故事中提及的名词，系统操作主要来自用户故事中提及的动词
- 可以使用名为事件风暴的技术定义领域模型

事件风暴三步骤：

- 头脑风暴
- 识别事件触发器
- 识别聚合

- 每个系统操作的行为都是根据它对一个或多个领域对象的影响以及它们之间的关系来描述的
- 系统操作可以创建、更新或删除领域对象，以及创建或破坏它们之间的关系

## 识别系统操作——创建抽象领域模型

- 应用程序本身并不需要一个领域模型，每个服务都有它自己的领域模型
- 抽象的领域模型有助于在开始阶段提供帮助，它定义了描述系统操作行为的一些词语
- 采用一些标准的技术，例如通过与领域专家沟通后，分析用户故事和场景中频繁出现的名词

# 识别系统操作——创建抽象领域模型

- 例如Place Order用户故事，分解为多个用户场景

```
Given a consumer
  And a restaurant
  And a delivery address/time that can be served by that restaurant
  And an order total that meets the restaurant's order minimum
When the consumer places an order for the restaurant
Then consumer's credit card is authorized
  And an order is created in the PENDING_ACCEPTANCE state
  And the order is associated with the consumer
  And the order is associated with the restaurant
```

在这个用户场景中的名词，如Consumer、Order、Restaurant和CreditCard，暗示了这些类都是需要的

# 识别系统操作——创建抽象领域模型

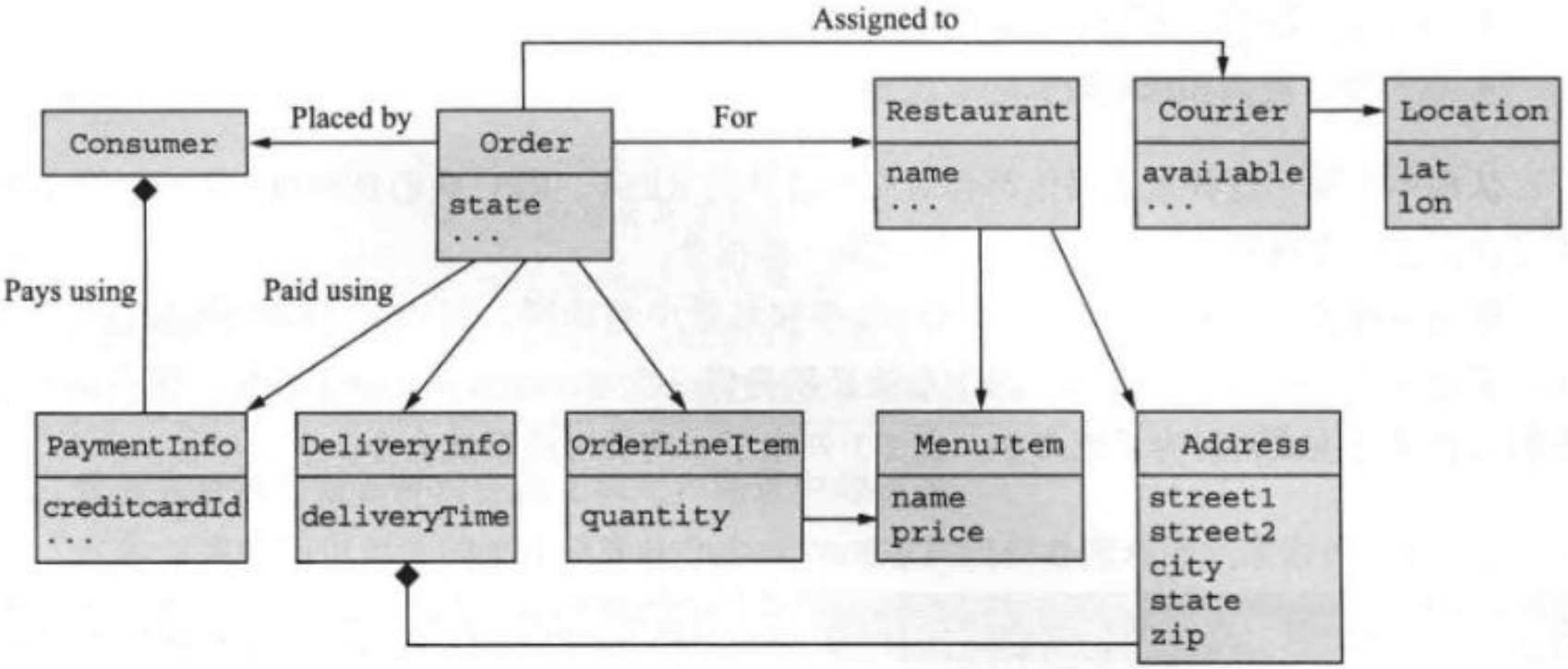
- 例如Accept Order用户故事，分解为多个用户场景

```
Given an order that is in the PENDING_ACCEPTANCE state
  and a courier that is available to deliver the order
When a restaurant accepts an order with a promise to prepare by a particular
     time
Then the state of the order is changed to ACCEPTED
  And the order's promiseByTime is updated to the promised time
  And the courier is assigned to deliver the order
```

这个场景暗示需要Courier类和Delivery类

# 识别系统操作——创建抽象领域模型

- 在经过几次迭代分析之后，结果显示就是这个领域模型应该包括一些类，如MenuItem和Address等，下图显示了核心类的类图



# 识别系统操作——创建抽象领域模型

- 每个类的作用如下：

- Consumer：下订单的用户。
- Order：用户下的订单，它用来描述订单并跟踪状态。
- OrderLineItem:Order中的一个和条目。
- DeliveryInfo：送餐的事件和地址。
- Restaurant：为用户准备生产订单的餐馆，同时也要发起送货。
- MenuItem：餐馆菜单上的一个条目。
- Courier：送餐员负责把订单送到用户手里。可跟踪送餐员的可用性和他们的位置。
- Address：Consumer或Restaurant的地址。
- Location：Courier当前的位置，用经纬度表示。

# 识别系统操作——定义系统操作

- 当定义了抽象的领域模型之后，接下来就要识别系统必须处理的各种请求
- 使用抽象的词汇来描述跟系统操作有关的请求更为合理
- 系统操作的类型
  - 命令型：创建、更新或删除数据的系统操作
  - 查询型：查询和读取数据的系统操作
- 这些系统操作都会对应到具体的REST、RPC或消息端口

# 识别系统操作——定义系统操作

- 识别系统指令的切入点是分析用户故事和场景中的动词
- 例如Place Order用户故事，系统必须提供一个Create Order操作
- 很多用户故事都会直接对应或映射为系统命令

| 操作者 | 用户故事 | 命　令 | 描　述 |
| --- | --- | --- | --- |
| Consumer | Create Order | createOrder() | 创建一个订单 |
| Restaurant | Accept Order | acceptOrder() | 表示餐馆接受了订单，并承诺在规定的时间准备完毕 |
| Restaurant | Order Ready for Pickup | noteOrderReadyForPickup() | 表示订单已经准备完毕，可以送餐 |
| Courier | Update Location | noteUpdatedLocation() | 更新送餐员的当前位置 |
| Courier | Delivery picked up | noteDeliveryPickedUp() | 表示送餐员已经取餐 |
| Courier | Delivery delivered | noteDeliveryDelivered() | 表示送餐员已经送餐 |

# 识别系统操作——定义系统操作

- 命令规范定义了命令对应参数、返回值和领域模型类的行为
- 行为规范中包括前置条件和后置条件
- 以下就是createOrder()系统操作的规范

| | |
|---|---|
| 操作 | createOrder(consumer id, payment method, delivery address, delivery time,restaurant id, order line items) |
| 返回 | orderId, … |
| 前置条件 | ▪消费者存在并可以下订单<br>▪条目对应餐馆的菜单项<br>▪送餐地址和时间在餐馆的服务范围内 |
| 后置条件 | ▪消费者的信用卡预授权完成<br>▪订单被创建并设置为 PENDING_ACCEPTANCE 状态 |

# 识别系统操作——定义系统操作

- 以下就是acceptOrder()系统操作的规范

| 操作 | acceptOrder(restaurantId, orderId, readyByTime) |
| --- | --- |
| 返回 | / |
| 前置条件 | ▪order.status 是 PENDING_ACCEPTANCE<br>▪有可用的送餐员完成这个订单的送餐任务 |
| 后置条件 | ▪order.satus 更改为 ACCEPTED<br>▪order.readyByTime 更改为 readyByTime<br>▪有送餐员被分配执行当前订单的送餐任务 |

# 识别系统操作——定义系统操作

- 多数与系统操作相关的架构元素是命令，查询虽然仅仅是简单的获取数据，也同样重要
- 查询为用户决策提供了用户界面
- 当消费者下单时往往是如下所示过程：
  1. 用户输入送餐地址和期望的送餐时间
  2. 系统显示当前可用的餐馆
  3. 用户选择餐馆
  4. 系统显示餐馆的菜单
  5. 用户点餐并结账
  6. 系统创建订单

# 识别系统操作——定义操作系统

- 下订单的用户场景包含了以下的查询型操作：

- findAvailableRestaurant(deliverAddress,deliverTime):获取所有能够送餐到用户地址并满足送餐事件要求的餐馆。
- findRestaurantMenu(id)：返回餐馆信息和这家餐馆的菜单项。

- findAvailableRestaurants()在架构层面尤其重要的一个，包含了地理位置等信息的复杂查询
- 执行这个查询时，客户多数都是“在线急等”的状态，耽误不得

# 识别系统操作

- 抽象的领域模型和系统操作能够回答这个应用“做什么”这一问题

- 系统操作被定义后，下一步就是完成应用服务的识别
  - 根据业务能力进行服务拆分
  - 根据子域进行服务拆分

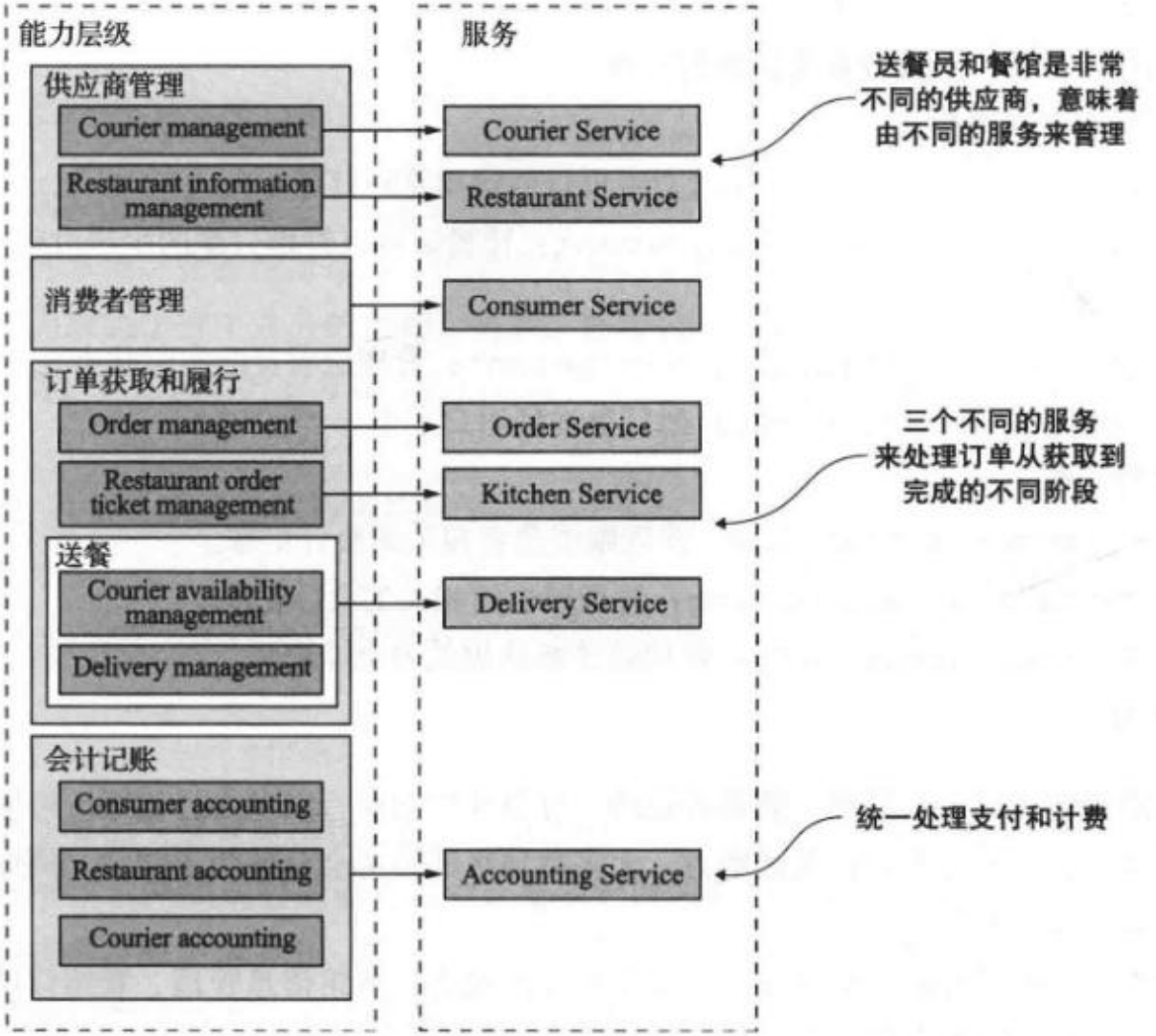
# 根据业务能力进行服务拆分

- 业务能力逐一列出

- 供应商管理：
  - Courier management：送餐员相关信息管理；
  - Restaurant information mangament：餐馆菜单和其他信息管理，例如营业地址和时间。
- 消费者管理：消费者有关信息的管理。
- 订单获取和履行：
  - Order management：让消费者可以创建和管理订单。
  - Restaurant order management：让餐馆可以管理订单的生产过程。
  - 送餐。
  - Courier availability management：管理送餐员的实时状态。
  - Delivery management：把订单送到用户手中。
- 会计记账：
  - Consumer accounting：管理跟消费者相关的会计记账。
  - Restaurant accounting：管理跟餐馆相关的会计记账。
  - Courier accounting: 管理跟送餐员相关的会计记账。
- 其他。

# 从业务能力到服务

- 一旦确定了业务能力，就可以为每个能力或相关能力组定义服务



# 根据子域进行服务拆分

- DDD有两个重要的概念：

  **子域**

- 领域驱动为每一个子域定义单独的领域模型。
- 子域是领域的一部分，领域是DDD中用来描述应用程序问题域的一个术语。

  **限界上下文**

- DDD把领域模型的边界称为限界上下文
- 限界上下文包括实现这个模型的代码集合
- 每一个限界上下文对应一个或一组服务

# 根据子域进行服务拆分

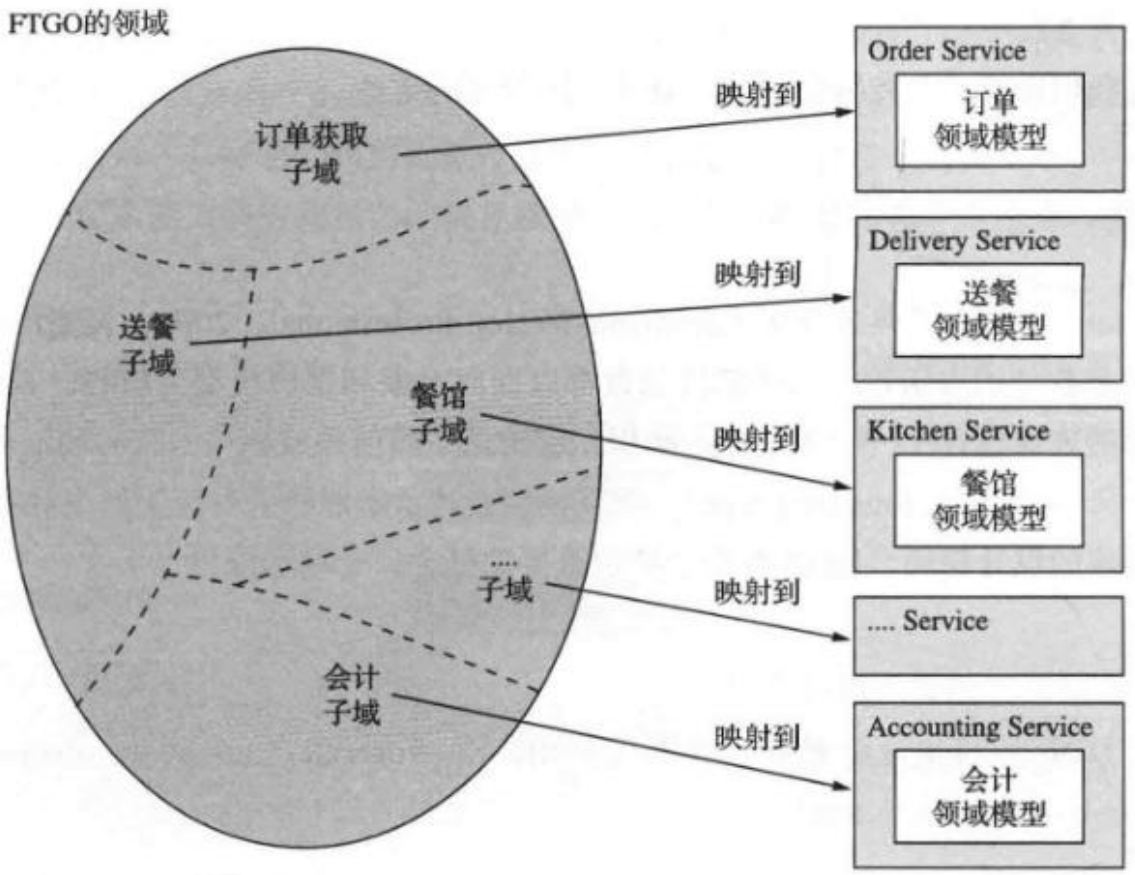
**传统的架构建模建立一个单独的模型**

- 会有适用于整个应用全局的业务实体定义
- 挑战在于让整个团队都对全局单一的建模和术语定义达成一致是非常困难的
- 单一的业务实体可能过于复杂，超出需求
- 造成混乱，因为组织内有些团队可能针对不同的概念使用相同的术语，但有些团队针对同一概念使用不同术语

**DDD通过定义多个领域模型来避免这些问题，每个领域模型都有明确的范围**

# 根据子域进行服务拆分

可以通过DDD的方式定义子域，并把子域对应为每一个服务



# 根据子域进行服务拆分

- DDD和微服务架构是天生一对
- 子域和限界上下文的概念可以很好的跟微服务架构中的服务进行匹配
- 微服务中的自治化团队负责服务开发的概念也跟DDD中每个领域模型都由一个独立团队负责开发的概念吻合
- 子域用于它自己的领域模型这个概念，为消除上帝类和优化服务拆分提供了好办法

# 服务划分的指导原则

- **无绝对统一的标准**
  - 不同的业务场景会演化出不同的服务结构
  - 具有不同工程能力、不同组织架构形式下的团队，也会演化出不同的服务结构
  - 没有哪种服务结构是绝对正确的，需要综合考虑

# 服务划分的指导原则

- **通用原则**
  - 单一职责原则
  - 闭包原则

# 单一职责原则(SRP)

- **改变一个类应该只有一个理由**

**——Robert C. Martin**

- 微服务通常功能单一，只负责处理一件事
- 呈现“高内聚、低耦合”的特征
- 高内聚体现在服务处理的业务逻辑单一，将不相关的业务逻辑隔离，需要修改某个行为时影响较小
- 低耦合体现在减少服务之间的不必要依赖，降低服务间的协作成本

# 闭包原则(CCP)

- **在包中包含的所有类应该是对同类的变化的一个集合，也就是说，如果对包做出修改，需要调整的类都在这个包之内**

**——Robert C. Martin**

- 把根据同样原因进行变化的服务放在一个组件内
- 控制服务的数量，当需求变化时，变更和部署也更加容易
- 理想情况下，一个变更只会影响一个团队和一个服务
- 解决分布式单体模式的法宝

# 拆分单体应用为服务的难点

- 主要障碍有：
  - 网络延迟
  - 同步进程间通信导致可用性降低
  - 在服务之间维持数据一致性
  - 获取一致的数据视图
  - 上帝类阻碍了拆分

## 网络延迟

- 是分布式系统中一直存在的问题
- 服务的特定拆分会导致两个服务之间的大量往返调用

## 解决方案

- 可以通过实施批处理API在一次往返中获取多个对象，从而将延迟减少到可接受的数量

- 把多个相关的服务组合在一起，用编程语言的函数调用替换昂贵的进程间通信

## 同步进程间通信导致可用性降低

- 如何处理进程间通信而不降低系统的可用性

## 解决方案

- 实现createOrder()操作最常见的方式是让Order Service使用REST同步调用其他服务，但是REST会降低Order Service的可用性。

- 异步通信来消除这类同步调用产生的紧耦合并提升可用性

# 在服务之间维持数据一致性

- 如何在某些系统操作需要更新多个服务中的数据时，仍旧维护服务之间的数据一致性

## 解决方案

- 传统的解决方案是使用基于两阶段提交的分布式事务管理机制
- 使用Saga处理事务管理，Saga是使用异步消息来协调一系列本地事务。Saga的限制是它们最终是一致的

## 获取一致的数据视图

- 无法跨多个数据库获得真正一致的数据视图

## 解决方案

- 在单体应用程序中，ACID事务的属性保证查询将返回数据库的一致视图

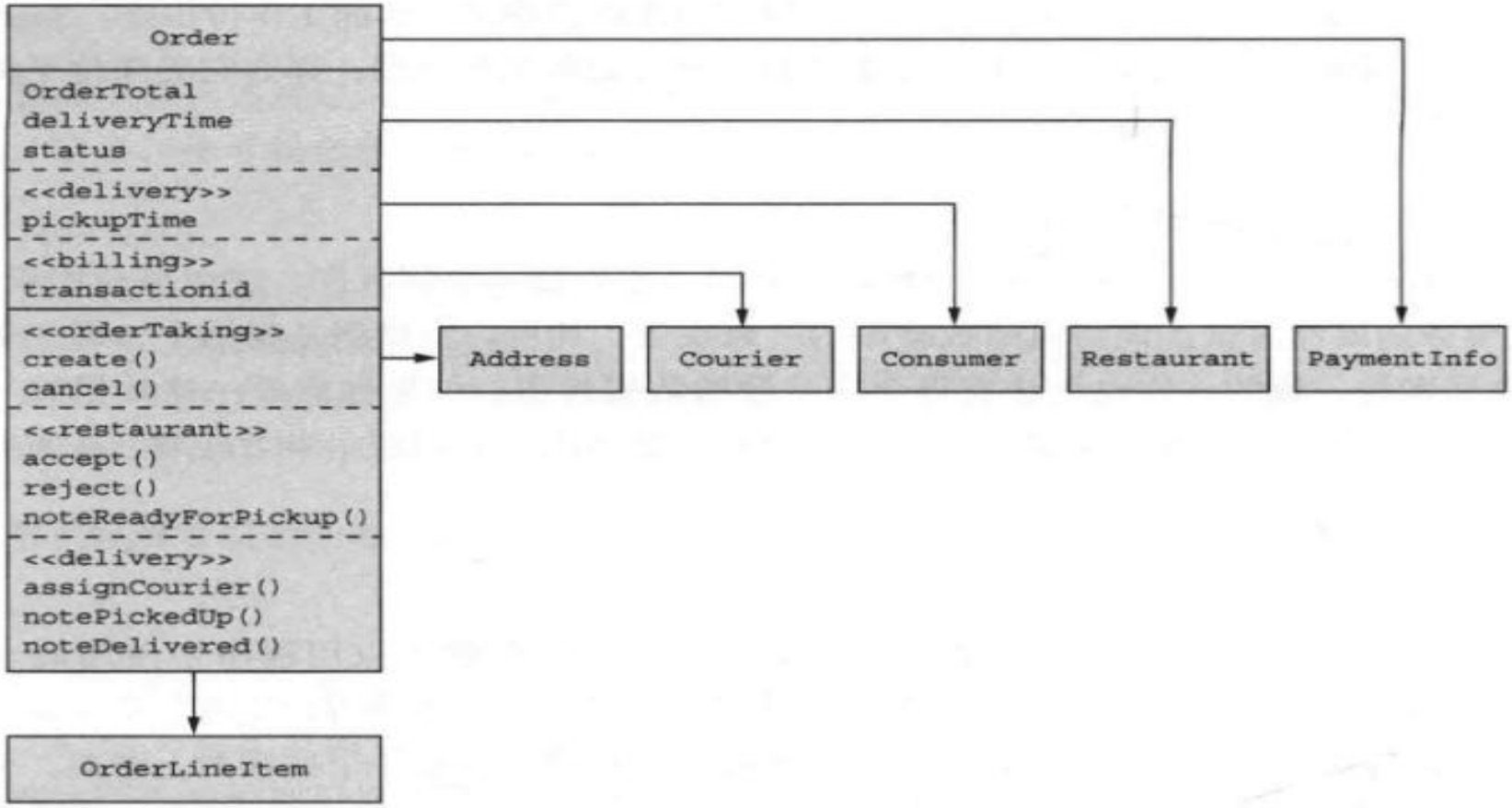
- 在微服务架构中，即使每个服务的数据库是一致的，也无法获得全局一致的数据视图，如果需要一致数据视图，必须驻留在单个服务中

## 上帝类阻碍了拆分

- 上帝类是在整个应用程序中使用的全局类
- 通常为应用程序的许多不同方面实现业务逻辑，有大量字段映射到具有许多列的数据库表
- 大多数应用程序至少有一个这样的上帝类，每个上帝类代表一个对领域至关重要的概念
- 上帝类将应用程序的许多不同方面的状态和行为捆绑在一起，所以将使用它的任何业务逻辑拆分为服务往往都是一个不可逾越的障碍

# 上帝类阻碍了拆分

- Order类就是上帝类的一个例子，系统大多数部分都涉及订单
- 如果应用程序具有单个领域模型，则Order类将是一个非常大的类，将具有与许多不同部分相对应的状态和行为



# 上帝类阻碍了拆分

- Order类具有与订单处理、餐馆订单管理、送餐和付款相对应的字段及方法
- Order类具有复杂的状态模型
- Order类的存在使得将代码分割成服务变得极其困难

# 上帝类阻碍了拆分

**第一种方案：**

- 将Order类打包到库中并创建一个中央Order数据库，处理订单的所有服务都使用此库并访问数据库。

- 导致紧耦合，对Order模式的任何更改都要求其他团队同步更新和重新编译他们的代码

# 上帝类阻碍了拆分

**第二种方案：**

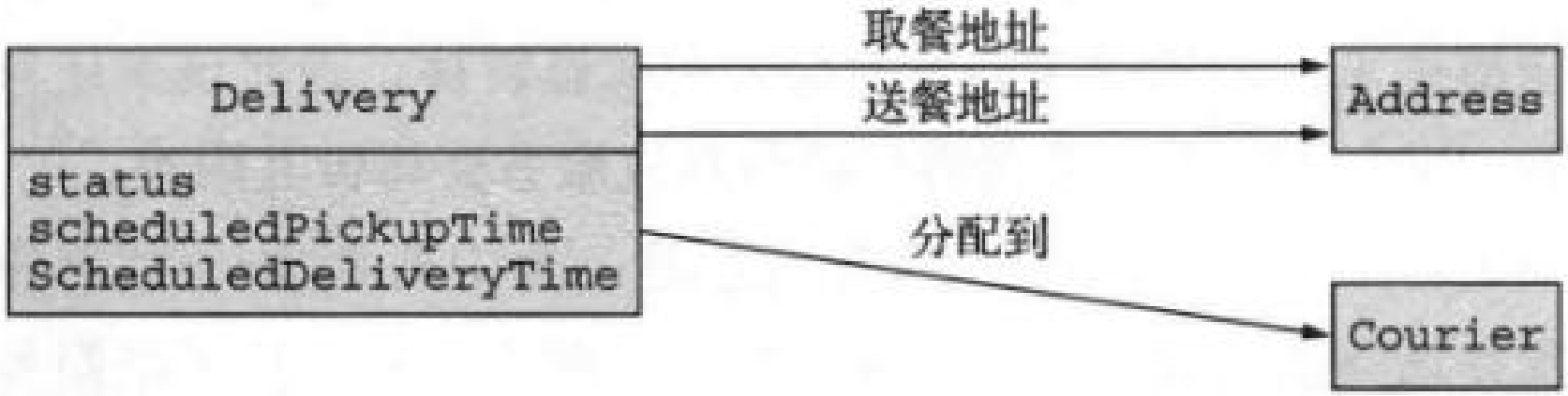
- 将Order数据库封装在Order Service中，该服务由其他服务调用以检索和更新订单

- Order Service将成为一个纯数据服务，成为包含很少或没有业务逻辑的贫血领域模型

# 上帝类阻碍了拆分

**第三种方案：**

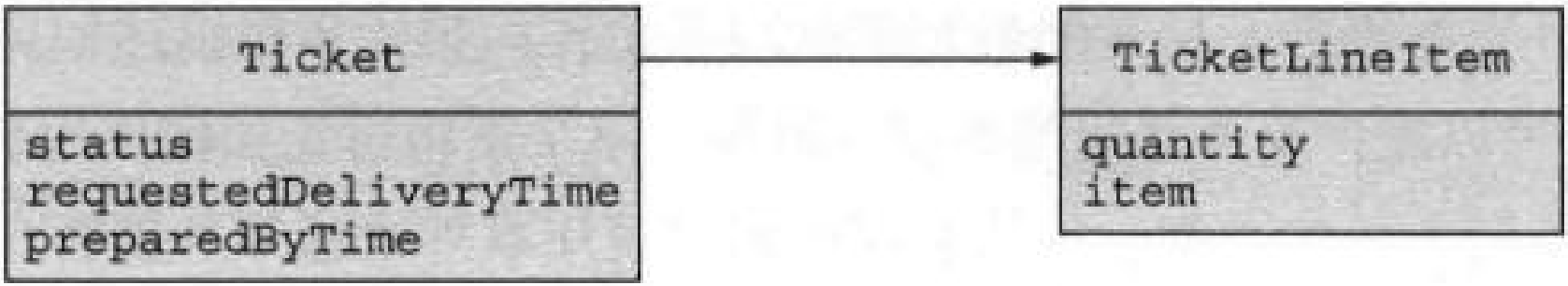
- 应用DDD将每个服务视为具有自己的领域模型的单独子域，应用程序中与订单有关的每个服务都有自己的领域模型及其对应的Order类版本
- Delivery Service是多领域模型的例子



# 上帝类阻碍了拆分

**第三种方案：**

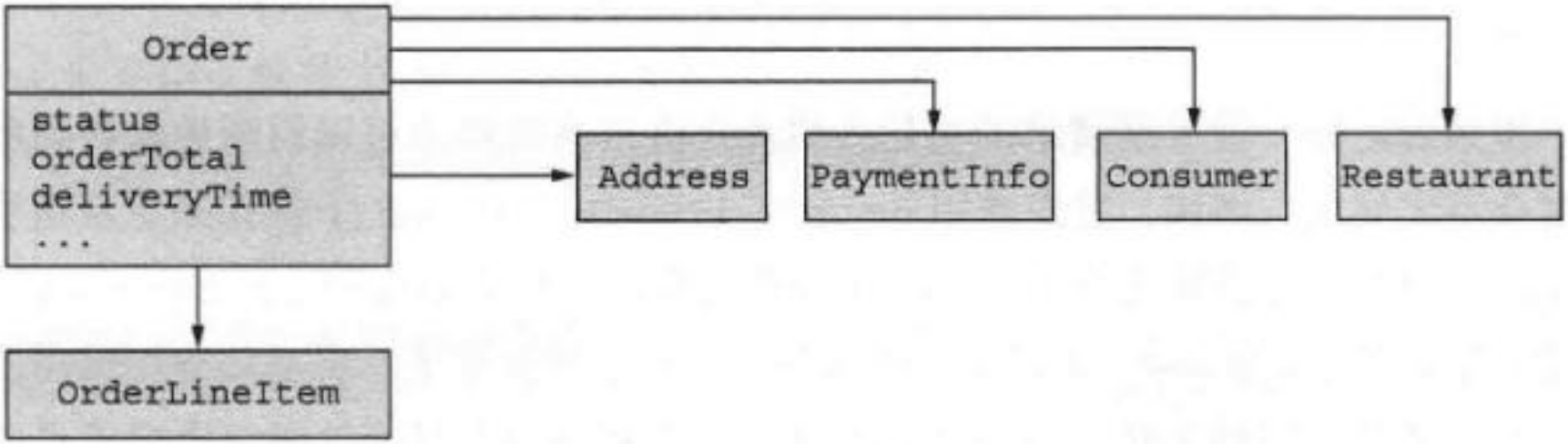
- Kitchen Service有一个更简单的订单视图，它的Order版本就是一个Kitchen



# 上帝类阻碍了拆分

**第三种方案：**

- Order Service具有最复杂的订单视图，但是比原始版本的Order上帝类简单



# 拆分上帝类带来的挑战

- 每个领域模型中的Order类表示同一Order业务实体的不同方面
- 应用程序必须维持不同服务中这些不同对象之间的一致性
- 拥有多个领域模型还会影响用户体验，应用程序必须在用户体验(即其自己的领域模型)与每个服务的领域模型之间进行转换

# 定义服务API

- 定义服务API就是定义服务的操作和事件

- 存在服务API操作有以下两个原因：
  1. 某些操作对应于系统操作，它们由外部客户端调用，也可能由其他服务调用
  2. 存在一些其他操作用以支持服务之间的协作，这些操作仅由其他服务调用

- 服务通过对外发布事件，使其能够与其他服务协作

- 定义服务API分为两步：

1）首先将每个系统操作映射到服务（即确定哪个服务是请求的初始入口点）

2）之后确定服务是否需要与其它服务协作以实现系统操作

## 1）把系统操作分配给服务

- 确定哪个服务是请求的初始入口点
- 许多系统操作可以清晰的映射到服务，但有时映射会不太明显

| 服　务 | 操　作 |
| --- | --- |
| Consumer Service | createConsumer() |
| Order Service | createOrder() |
| Restaurant Service | findAvailableRestaurants() |
| Kitchen Service | ▪ acceptOrder()<br>▪ noteOrderReadyForPickup() |
| Delivery Service | ▪ noteUpdatedLocation()<br>▪ noteDeliveryPickedUp()<br>▪ noteDeliveryDelivered() |

## 2）确定支持服务协作所需要的API

- 某些系统操作完全由单个服务处理
- 其他系统操作跨越多个服务，处理这些请求之一所需的数据可能分散在多个服务周围
- 例如为了实现createOrder()操作，Order Service必须调用以下服务以验证其前置条件并使后置条件成立：

■ Consumer Service：验证消费者是否可以下订单并获取其付款信息。

■ Restaurant Service：验证订单行项目，验证送货地址和时间是否在餐厅的服务区域内，验证订单最低要求，并获得订单行项目的价格。

■ Kitchen Service：创建Ticket(后厨工单)。

■ Accounting Service： 授权消费者的信用卡。

- 为了完整定义服务API，需要分析每个系统操作并确定所需的协作

| 服　务 | 操　作 | 协 作 者 |
| --- | --- | --- |
| Consumer Service | verifyConsumerDetails() | — |
| Order Service | createOrder() | ▪ Consumer Service verifyConsumerDetails()<br>▪ Restaurant Service verifyOrderDetails()<br>▪ Kitchen Service createTicket()<br>▪ Accounting Service authorizeCard() |
| Restaurant Service | ▪ findAvailableRestaurants()<br>▪ verifyOrderDetails() | — |
| Kitchen Service | ▪ createTicket()<br>▪ acceptOrder()<br>▪ noteOrderReadyForPickup() | ▪ Delivery Service scheduleDelivery() |
| Delivery Service | ▪ scheduleDelivery()<br>▪ noteUpdatedLocation()<br>▪ noteDeliveryPickedUp()<br>▪ noteDeliveryDelivered() | — |
| Accounting Service | ▪ authorizeCard() | — |