软件工程实践指南 · 在线书城示例

软件开工前到底该画哪些图?

系统讲清需求、业务、架构、数据、交互与部署设计

面向刚开始学习软件开发、软件工程，以及希望系统整理设计方法的软件开发工程师。

系统上下文　用例与流程　状态与领域　架构与组件　ER 与时序　生产部署

案例：简化版在线书城

版本：2026-07

CONTENTS

目录

- 一、先认识在线书城的业务范围
- 二、软件实施前的“画图地图”
- 三、系统上下文图：先确定系统边界
- 四、用例图：说明谁能完成什么目标
- 五、业务流程图：说明业务怎样从开始走到结束
- 六、状态机图：描述订单的完整生命周期
- 七、领域模型与 UML 类图：描述业务对象及其关系
- 八、系统架构图：使用 C4 容器图描述高层结构
- 九、组件图：继续放大后端内部模块
- 十、ER 图：设计关系型数据库的数据结构
- 十一、时序图：描述一个具体场景的运行时交互
- 十二、部署架构图：说明软件最终运行在哪里
- 十三、这些图怎样共同描述同一个系统
- 十四、怎样把图画好，而不是只把图画出来
- 十五、不同项目规模应画到什么程度
- 十六、还有哪些图可能在前期出现
- 十七、在线书城从立项到开工的推荐顺序
- 十八、开工前最终检查表
- 参考资料

> 阅读建议：先浏览“画图地图”，再按项目阶段选择相应章节；无需机械地为所有项目画满全部图形。

> 在一个软件项目真正进入编码阶段之前，团队通常需要依次回答六类问题：系统边界在哪里、用户要完成什么目标、业务怎样运行、系统怎样拆分、数据怎样组织、运行时怎样交互并最终部署。不同的图回答不同的问题，没有一张“万能架构图”能够替代全部设计工作。

本文采用一个统一而简单的“在线书城”案例，系统讲解软件实施前常见的十类图：

1. 系统上下文图
2. 用例图
3. 业务流程图与 BPMN
4. 状态机图
5. 领域模型与 UML 类图
6. 系统架构图，也就是 C4 容器图
7. 组件图
8. ER 图
9. 时序图
10. 部署架构图

你将看到每一种图在什么阶段使用、它解决什么问题、图中的形状和连线分别表示什么，以及怎样避免常见的画图错误。

> **先建立一个关键认识：**UML、BPMN、C4 是建模语言或建模方法；Mermaid、PlantUML、draw.io、Visio 是绘图工具。前者决定“用什么语义表达”，后者决定“用什么软件把图画出来”。UML 由 OMG 维护，用于可视化、描述和记录软件与系统；BPMN 也是 OMG 标准，专门描述业务过程；C4 则通过上下文、容器、组件和代码等缩放层级描述软件架构。[1] [2] [3]

一、先认识在线书城的业务范围

本文只设计一个简化版本，功能有意控制在初学者容易理解的范围内。

角色或外部系统	主要能力
顾客	浏览和搜索图书、维护购物车、提交订单、支付、查询订单、申请退款

角色或外部系统	主要能力
管理员	管理图书、维护库存、查看订单、确认发货
第三方支付平台	创建支付单、完成支付、通知支付结果、执行退款
短信或邮件平台	发送下单、支付成功和发货通知

暂不考虑优惠券、多仓库、拆单、组合支付、多币种、会员等级、复杂促销和跨境税务。

系统中会反复出现这些核心业务概念：

```
顾客 Customer
图书 Book
购物车 ShoppingCart
购物车项 CartItem
订单 Order
订单项 OrderItem
支付 Payment
库存 Inventory
```

这些名称应该在需求文档、业务流程、类图、ER 图、API 和代码中尽量保持一致。统一的业务词汇本身就是设计成果。

二、软件实施前的“画图地图”

下表给出一个可直接应用到项目中的顺序。它不是严格的瀑布流程：后续设计发现问题时，需要回头修改前面的图。

设计阶段	需要回答的问题	优先使用的图	主要读者
项目立项与范围澄清	系统服务谁？边界在哪里？依赖哪些外部系统？	系统上下文图	产品、业务、架构、开发、测试
功能需求分析	不同角色能使用哪些能力？	用例图	产品、业务、开发、测试
业务分析	下单、退款、发货怎样从开始走到结束？	业务流程图、BPMN	业务、产品、开发、测试
业务规则设计	订单有哪些状态？什么事件可以改变状态？	状态机图	产品、开发、测试
领域分析	有哪些核心业务对象？对象之间是什么关系？	领域模型、类图	架构、开发、测试

设计阶段	需要回答的问题	优先使用的图	主要读者
高层架构设计	系统内部有哪些应用和数据存储？	C4 容器图	架构、开发、运维
模块设计	后端内部由哪些模块组成？依赖方向是什么？	组件图	架构、开发
数据库设计	有哪些表、字段、主键、外键和基数约束？	ER 图	后端、DBA、测试、数据工程师
详细设计	一个具体请求经过哪些组件？顺序和异常怎样处理？	时序图	开发、测试、架构
上线设计	系统运行在哪里？实例、网络和基础设施怎样组织？	部署图	架构、运维、SRE、安全

最重要的选择原则：不要为了“文档齐全”而机械地画图。每张图都必须回答一个具体问题，帮助团队消除歧义、发现风险或做出决策。C4 官方也明确指出，并非所有项目都需要画满全部层级；上下文图和容器图对多数团队已经很有价值，组件图只在确实增加信息时绘制。[3] [6]

三、系统上下文图：先确定系统边界

3.1 它回答什么问题

系统上下文图回答：

> 我们正在建设的系统是什么？谁使用它？它与哪些外部系统直接交互？团队负责到哪里为止？

C4 Model 将系统上下文图视为很好的起点：把目标系统放在中央，周围放用户和直接交互的外部系统。这个层级关注人员、角色和软件系统，而不是数据库、框架、协议或代码细节。[4]

图 1：在线书城系统上下文图。中央是目标系统，周围是用户与外部依赖。

3.2 图中的元素怎样读

人员或角色

“顾客”和“管理员”表示系统外部的角色，不是某个具体自然人，也不是数据库中的 `user` 表。一名员工可以同时承担多个角色。

目标软件系统

中央的“在线书城”是本次开发的目标系统，也叫 **System of Interest**。系统框中最好同时写出：

```
名称
类型
一句话职责
```

只写“在线书城”信息不足；写成“为顾客提供浏览、购物、支付和订单查询，并为管理员提供图书、库存和订单管理”更有价值。

外部软件系统

支付平台和通知平台位于系统边界之外。“外部”不等于“部署在公网”，而是表示当前团队不拥有其完整实现和生命周期。公司内部统一认证平台也可能是外部系统。

连线和箭头

连线表示元素之间存在交互；箭头表示主要方向。箭头上应写业务含义：

```
创建支付单
通知支付结果
发送业务通知
浏览图书和提交订单
```

不要只写“调用”“使用”“连接”。

3.3 什么时候画

它通常是最早完成的图之一，适合项目启动、需求范围确认、跨团队协作和外部依赖识别。它应该能让非技术干系人看懂。

3.4 常见错误

- 把 Spring Boot、MySQL、Redis、Kubernetes 放进上下文图，混入了内部技术细节。
- 漏画外部身份认证、支付、通知、物流等关键依赖。
- 没有画系统边界，读者不知道哪些能力归当前团队负责。
- 关系线没有标签，读者只能猜测系统之间交换什么。

四、用例图：说明谁能完成什么目标

4.1 它回答什么问题

用例图回答：

> 系统有哪些外部参与者？每个参与者希望通过系统完成哪些业务目标？

它适合在需求早期对齐功能范围。传统 UML 中，参与者常画成小人，用例画成椭圆，系统范围画成一个大矩形。[8]

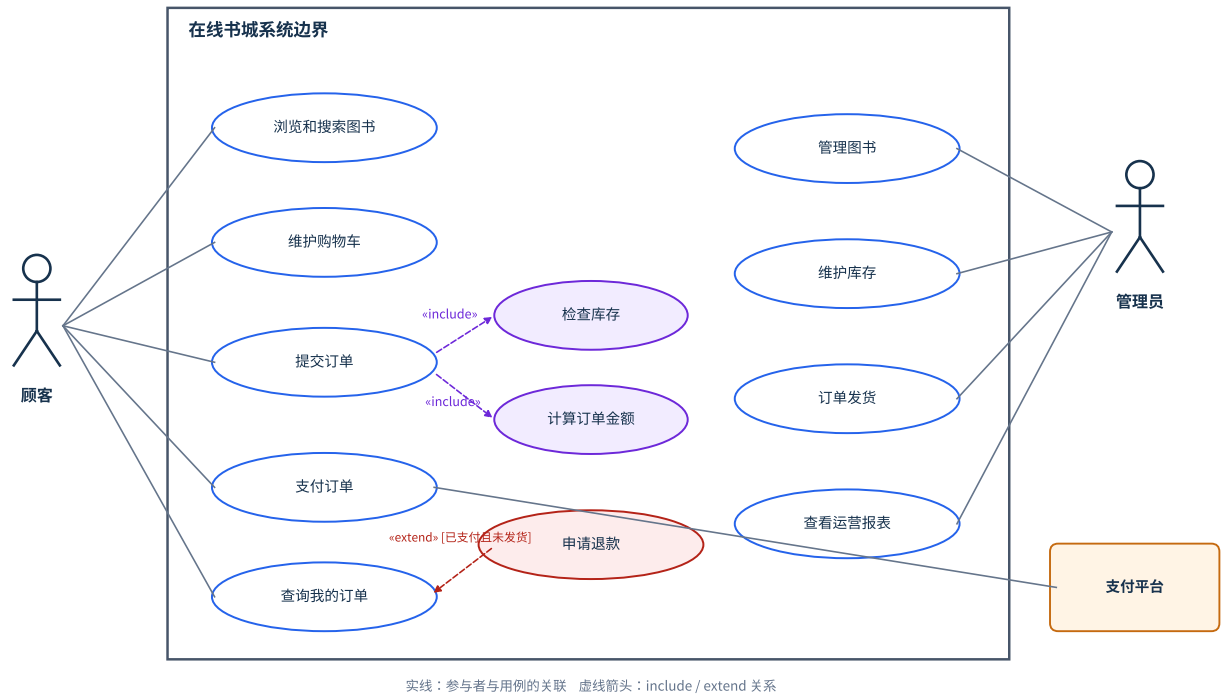


图 2：在线书城用例图。用例表达外部参与者希望通过系统完成的目标。

4.2 主要符号

元素	常见画法	含义
Actor 参与者	小人或标注了角色的框	系统外部的人员角色、外部系统或设备
Use Case 用例	椭圆	参与者希望通过系统完成的目标
System Boundary 系统边界	包围用例的大矩形	当前系统负责提供的能力范围
Association 关联	参与者与用例之间的实线	该参与者会参与该用例
`«include»`	指向被包含用例的虚线箭头	基础用例每次都必须执行的复用行为
`«extend»`	指向基础用例的虚线箭头	在特定条件下附加的可选行为
Generalization 泛化	空心三角箭头	参与者或用例之间的通用与特化关系

4.3 用例名称怎样写

推荐“动词 + 名词”：

```
浏览图书
提交订单
支付订单
维护库存
申请退款
```

不推荐：

```
订单功能
用户模块
支付页面
数据库操作
```

用例表达的是可感知的业务目标，不是菜单、页面、Controller 或数据库操作。

4.4 `include` 和 `extend` 怎样区分

`include`：必然执行

“提交订单”每次都必须“检查库存”和“计算订单金额”，因此可以表达为：

```
提交订单 «include» 检查库存
提交订单 «include» 计算订单金额
```

`extend`：满足条件时附加

“申请退款”只有在订单已支付且符合退款条件时才出现，因此可视为对“管理我的订单”的条件扩展。

不要把 `include` 和 `extend` 当成普通函数调用关系。它们服务于需求表达，而不是代码调用设计。

4.5 用例图不能回答什么

用例图不会告诉你：

- 用户先做什么、后做什么；
- 下单有哪些判断和异常路径；
- 前端调用了哪些接口；
- 哪些表会被读写；
- 订单状态如何变化。

这些问题分别交给流程图、时序图、ER 图和状态机图。

五、业务流程图：说明业务怎样从开始走到结束

5.1 它回答什么问题

业务流程图回答：

> 一项业务依次经过哪些步骤？在哪里出现判断？正常路径和异常路径分别走向哪里？

下面以“顾客提交订单并完成支付”为例。

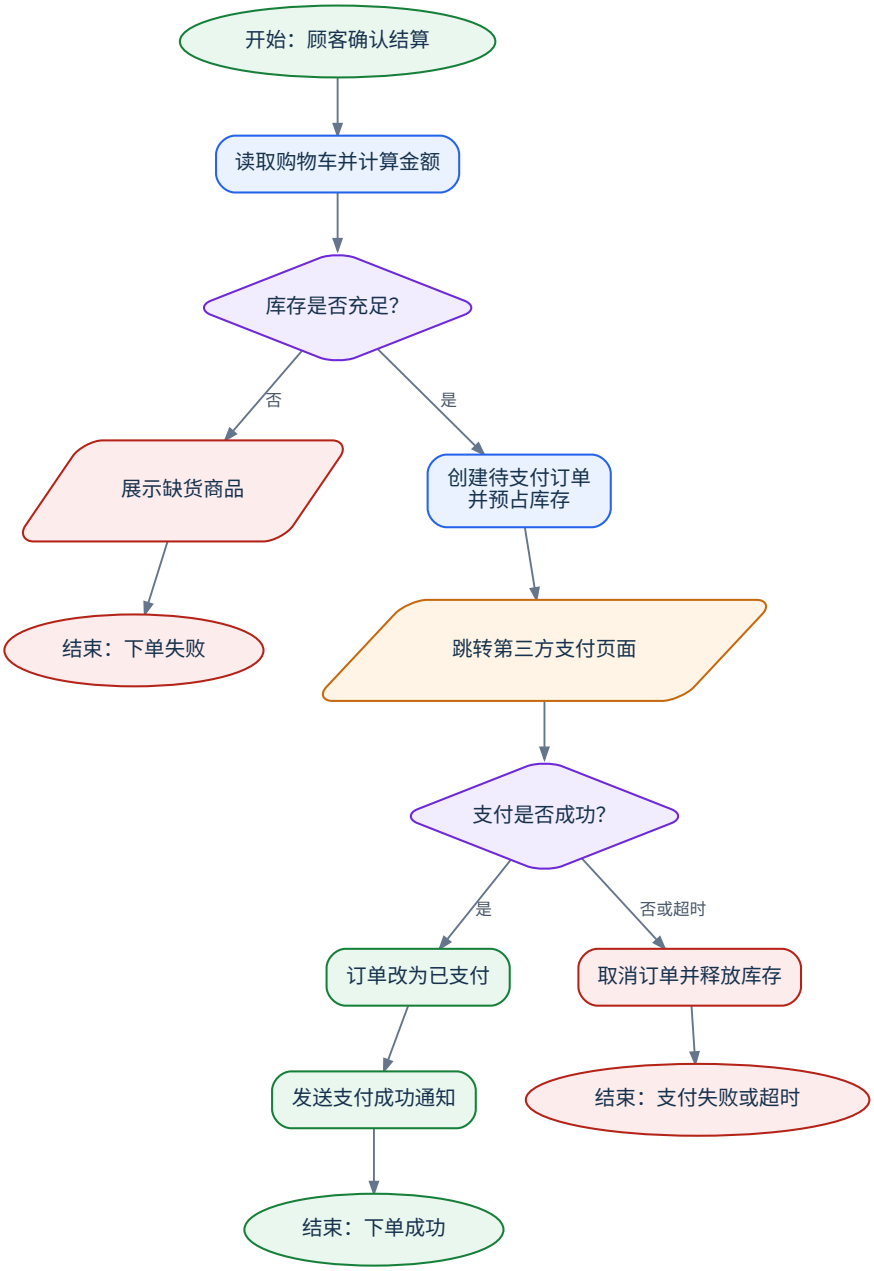


图 3：在线书城下单流程。流程图强调步骤、判断、分支和结束结果。

5.2 普通流程图的常见形状

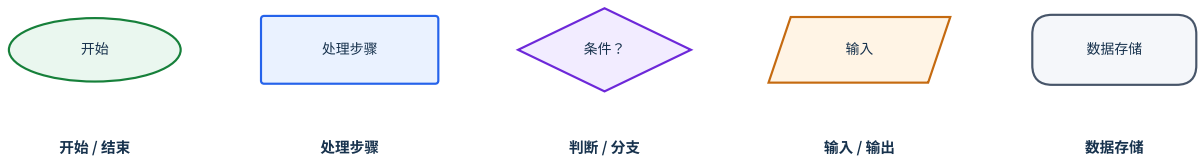


图 4：流程图常见符号。不同团队可能有细微差异，正式文档应提供图例。

Mermaid 等绘图工具提供矩形、圆角节点、菱形、平行四边形、圆柱体等形状，用来增强流程语义。[11]

开始和结束：椭圆或圆角终止符

一个完整流程应有明确的触发条件和业务结果。

不够好的写法：

```
开始
结束
```

更好的写法：

```
开始：顾客点击“确认结算”
结束：下单成功
结束：库存不足
结束：支付失败或超时
```

处理步骤：长方形

表示人员或系统执行一个动作：

```
读取购物车
检查库存
创建订单
释放库存
发送通知
```

推荐使用动宾结构，不要只写“订单”“支付”“处理”。

判断：菱形

表示流程根据条件进入不同分支。菱形中应写成可回答的问题：

```
库存是否充足？
支付是否成功？
订单是否允许退款？
```

每条出边必须写条件，例如“是 / 否”“成功 / 失败”，不能让读者猜。

输入或输出：平行四边形

常用于表达用户输入、页面展示、文件输入输出等：

```
展示缺货商品
跳转支付页面
读取上传文件
```

数据存储：圆柱体

表示数据库、文件存储或其他持久化位置。注意：流程图中不必把每次数据库读写都画出来；只有当数据存取本身对理解业务有价值时才画。

箭头

箭头表示流程执行方向。最好保持统一的阅读方向，例如从上到下。大量交叉和回折通常意味着需要拆分为多张图。

5.3 普通流程图与 BPMN 的区别

普通流程图适合快速表达简单业务。BPMN 更适合多参与方、人工任务、系统任务、消息、定时器、异常事件和并行流程。OMG 将 BPMN 定义为业务过程的标准图形化记法：既要让业务人员理解，也要足够精确地支撑技术实现。[2]

BPMN 元素	典型图形	含义
Start Event	细边圆	流程开始
Intermediate Event	双边圆	流程中发生的消息、定时、异常等事件
End Event	粗边圆	流程结束
Task / Activity	圆角矩形	人或系统完成的一项工作
Exclusive Gateway	带 `X` 或空心菱形	多个分支只选择一个
Parallel Gateway	带 `+` 的菱形	同时启动或汇合多个分支
Inclusive Gateway	带 `O` 的菱形	选择一个或多个分支
Sequence Flow	实线箭头	同一参与者流程内的执行顺序

BPMN 元素	典型图形	含义
Message Flow	虚线箭头	不同参与者之间传递消息
Pool	大泳池	独立参与者、组织或系统
Lane	Pool 内的分区	部门、角色或系统职责

菱形不一定等于 if/else。在 BPMN 中，菱形内部的标记决定语义。支付成功后同时“发送通知、增加积分、更新统计”属于并行网关，不是互斥判断。

六、状态机图：描述订单的完整生命周期

6.1 它回答什么问题

状态机图回答：

> 一个业务对象有哪些稳定状态？发生什么事件时可以从一个状态进入另一个状态？转换需要满足什么条件，又会执行什么动作？

订单、支付、退款、审批、工单和任务调度通常都需要状态机。

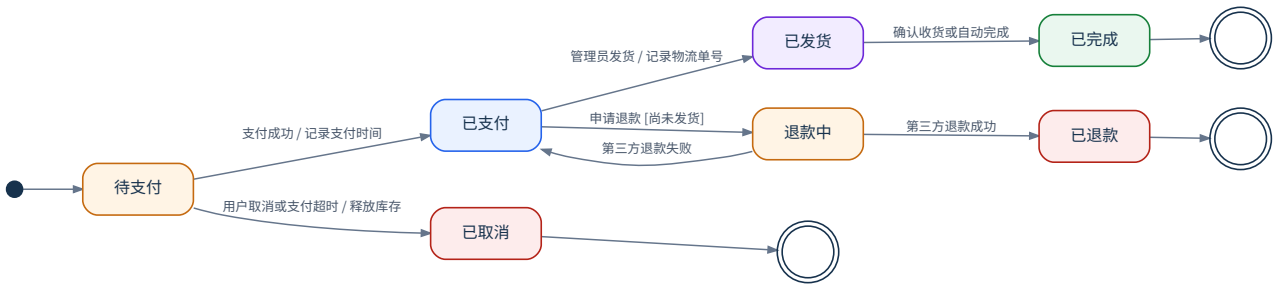


图 5：订单状态机。箭头表示由事件触发的状态转换，而不是普通业务步骤。

6.2 主要符号

状态

圆角矩形表示对象所处的稳定状态：

```
待支付
已支付
已发货
退款中
已退款
```

“支付”“发货”“取消订单”更像动作或事件，不适合作为状态名称。

初始状态与终止状态

实心黑点表示生命周期起点；双圆或终止节点表示生命周期结束。一个状态机可以存在多个终态，例如“已完成”“已取消”“已退款”。状态图中的转换通常用有方向的边表示，并可在边上附加文字；特殊起点和终点也有独立记法。[12]

状态转换

完整的转换标签通常写成：

```
事件［守卫条件］/ 执行动作
```

例如：

```
申请退款［尚未发货］/ 创建退款单
```

含义是：顾客触发“申请退款”；只有“尚未发货”时才允许转换；转换过程中创建退款单；订单进入“退款中”。

守卫条件

方括号中的条件决定转换是否合法。守卫条件必须可以被系统确定地判断，避免“情况正常”“用户合理”等模糊表达。

6.3 状态机图与流程图的区别

维度	业务流程图	状态机图
关注对象	一件业务从开始到结束的执行步骤	某一个业务对象的生命周期
核心问题	下一步做什么	当前处于什么状态，允许怎样变化
典型节点	活动、判断、输入输出	状态、事件、守卫条件、转换
在线书城示例	下单并支付的完整流程	一张订单从待支付到完成、取消或退款

6.4 画完之后必须检查

- 每个状态允许接收哪些事件?
- 非法转换怎样处理?
- 同一事件重复到达时是否幂等?
- 状态转换和数据库事务是否一致?
- 失败后停留在原状态，还是进入专门的失败状态?
- 终态之后是否允许修改?
- 谁有权限触发转换?

支付平台可能重复发送成功回调，因此“待支付 → 已支付”必须支持幂等。这个结论会继续影响唯一索引、事务和时序图。

七、领域模型与 UML 类图：描述业务对象及其关系

7.1 它回答什么问题

领域模型或类图回答：

> 业务中有哪些核心对象？对象具有什么属性和行为？对象之间是关联、组合、继承还是依赖?

UML 类图是静态结构图，通常展示类、属性、操作以及类之间的关系。[13]

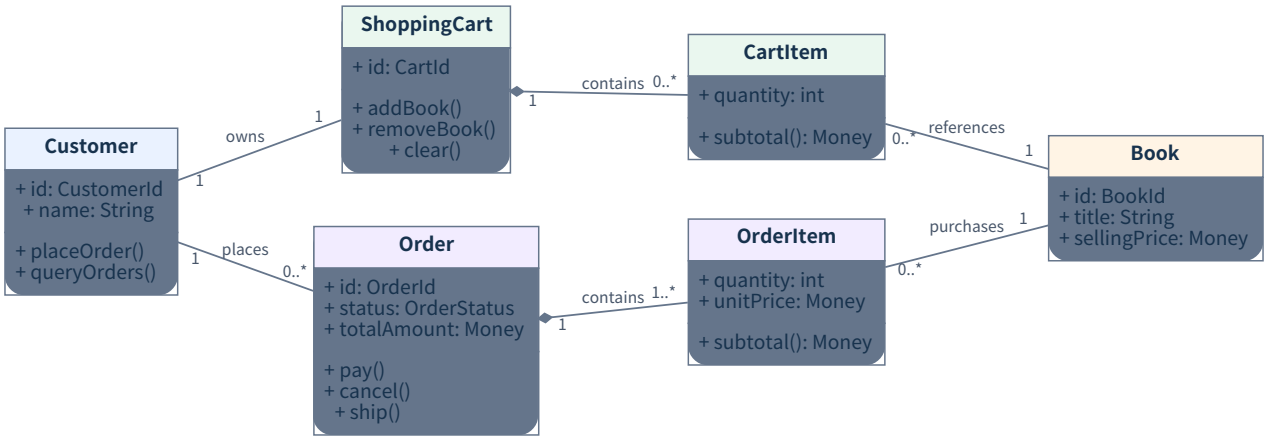
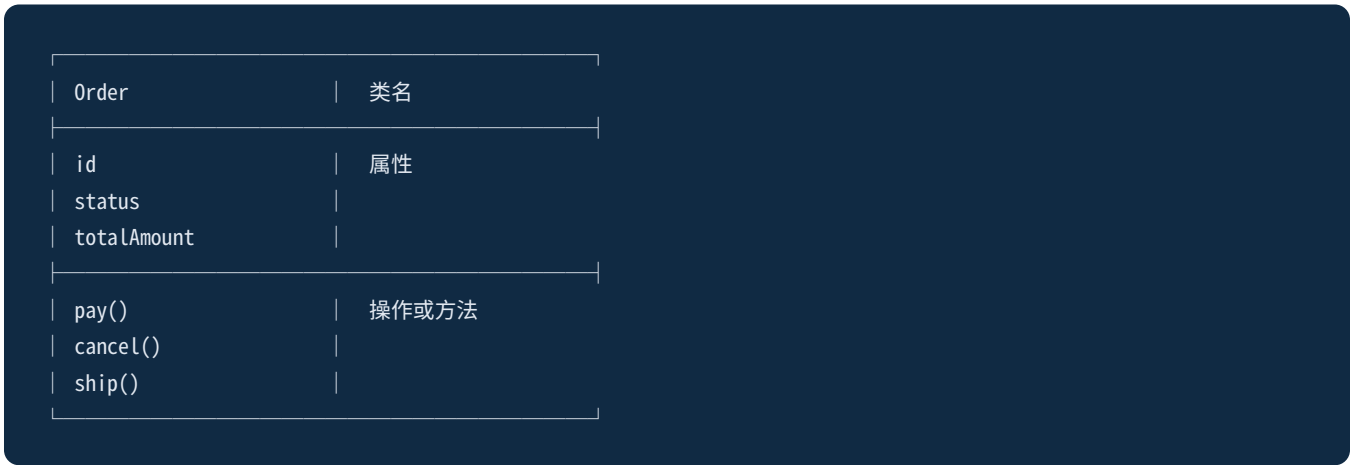


图 6：简化的在线书城领域类图。它描述业务对象，而不是直接复制数据库表。

7.2 类框的三个区域



概念级领域模型只保留最重要的业务属性和行为。不要一开始就塞入所有 getter、setter、DTO 字段和框架注解。

7.3 可见性符号

符号	含义
`+`	public
`-`	private
`#`	protected
`~`	package / internal

这些符号适合代码级类图；概念级领域模型可以省略。Mermaid 类图文档也使用这组可见性标记。[13]

7.4 类之间的关系

Association：关联

普通实线表示对象之间存在结构性关系，例如 `Customer` 创建 `Order`，`OrderItem` 引用 `Book`。

Multiplicity：多重性

标记	含义
`1`	恰好一个
`0..1`	零个或一个
`0..*`	零个或多个

标记	含义
`1..*`	一个或多个
`2..5`	两个到五个

例如：

```
Customer 1 ---- 0..* Order
```

表示一个顾客可以没有订单，也可以有多个订单；每张订单属于一个顾客。

Composition：组合

实心菱形表示强生命周期从属：

```
Order ◆---- OrderItem
```

订单项作为订单的一部分存在。删除订单后，订单项通常也失去独立意义。

Aggregation：聚合

空心菱形表示较弱的整体与部分关系。它在实际项目中经常被滥用；无法清楚解释生命周期关系时，使用普通关联通常更安全。

Inheritance：继承

空心三角箭头表示泛化或继承：

```
User <|-- Customer
User <|-- Administrator
```

Dependency：依赖

虚线箭头表示临时使用或依赖，例如 `OrderService` 调用 `PaymentGateway`，但不一定长期持有其对象。

7.5 类图不等于 ER 图

对比维度	类图 / 领域模型	ER 图
关注对象	业务对象或软件类	数据实体或关系表
主要内容	属性、行为、对象关系	字段、主键、外键、基数
是否展示方法	可以	通常不展示

对比维度	类图 / 领域模型	ER 图
与数据库是否一一对应	不一定	通常接近数据库结构
主要用途	领域建模、面向对象设计	关系型数据库设计

一个 `Order` 领域对象可能落到 `book_order`、`order_item`、`payment`、`order_status_history` 多张表中。

八、系统架构图：使用 C4 容器图描述高层结构

“系统架构图”没有唯一标准含义。团队经常把服务图、网络图、部署图和模块图都叫架构图，结果不同读者看到的是不同抽象层级。

C4 Model 用明确的缩放层级减少这种歧义：

```
System Context: 系统与外部世界
Container: 系统内部的应用和数据存储
Component: 某个容器内部的组件
Code: 组件内部的代码结构
Deployment: 软件实例怎样部署到基础设施
```

8.1 容器图回答什么问题

> 系统内部由哪些可运行的应用和数据存储组成？它们分别承担什么职责？使用哪些主要技术？怎样通信？

C4 中的 **Container** 是应用或数据存储，例如单页前端、后端应用、移动应用、数据库 Schema、对象存储等，并不专指 Docker Container。[5]

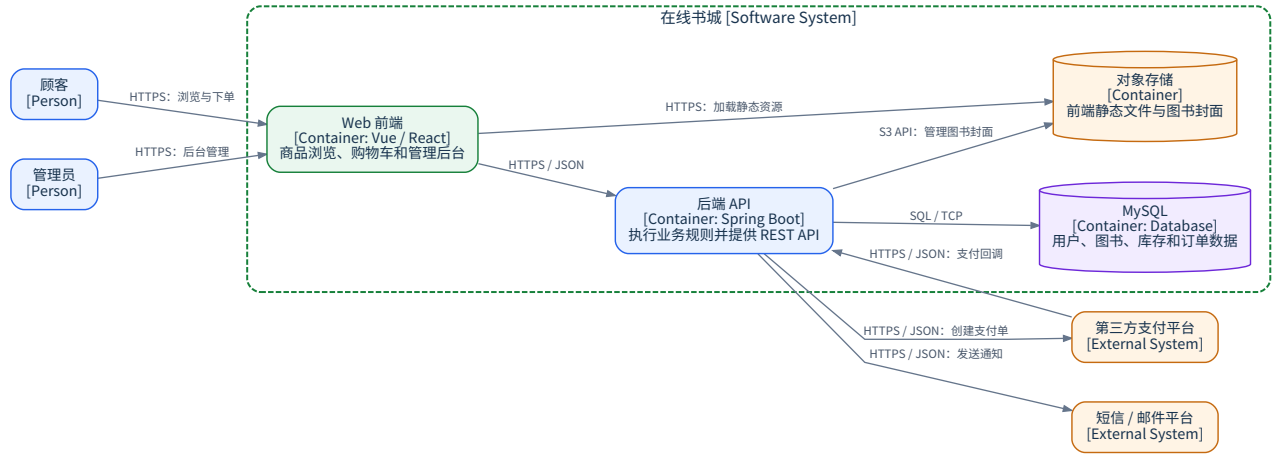


图 7：在线书城容器图。它表达逻辑应用、数据存储、职责和通信方式。

8.2 每个容器应该写什么

建议至少包含：

```
名称
类型
主要技术
一句职责描述
```

例如：

```
后端 API
[Container: Spring Boot]
执行业务规则并提供 REST API
```

仅写“Backend”通常信息不足。

8.3 关系线上写什么

最好同时写出：

```
交互目的 + 协议或技术
```

例如：

```
HTTPS / JSON：创建支付单
SQL / TCP：读写订单数据
S3 API：管理图书封面
```

8.4 为什么不画负载均衡和三个应用实例

容器图描述逻辑结构：“系统中有一个后端 API 应用”。负载均衡、集群、实例数量、故障切换属于特定环境的部署信息，应该放在部署图中。C4 官方也明确建议把这些内容放入环境对应的部署图。[5]

九、组件图：继续放大后端内部模块

9.1 它回答什么问题

组件图回答：

> 某一个容器内部由哪些主要组件或模块组成？每个组件承担什么职责？依赖方向怎样？

C4　组件图的范围是单个容器。官方建议只有在确实增加价值时绘制，因为组件结构很容易随代码变化而过时。[6]

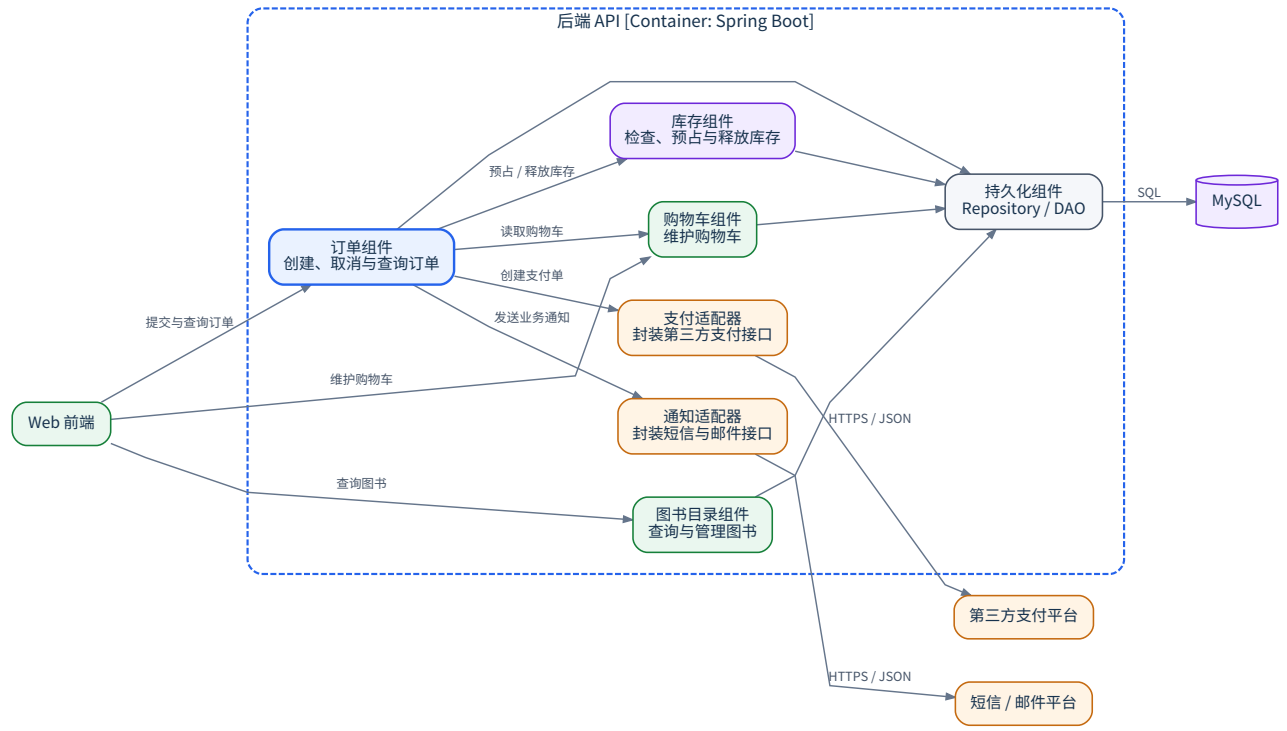


图 8：后端 API 的组件图。图中的组件仍运行在同一个 Spring Boot 进程中。

9.2 组件不是任意一个 Java 类

“订单组件”可能对应一个模块或包边界：

```
order/
 ├── application/
 ├── domain/
 ├── infrastructure/
 └── interfaces/
```

一个有意义的组件应该具有：清晰职责、明确接口、内部高内聚和可识别的依赖方向。

9.3 组件不等于微服务

图中的订单组件、库存组件和购物车组件仍可运行在同一个 Spring Boot 进程中。只有当它们具备独立部署、独立运行和独立生命周期时，才更适合建模为不同容器或不同软件系统。

9.4 什么时候不需要画

小型 CRUD 项目中，如果容器图和代码目录已经足够清楚，组件图可能没有新增信息。不要为了层级完整而复制一张更细、更难维护的图。

十、ER 图：设计关系型数据库的数据结构

10.1 它回答什么问题

ER 图回答：

> 有哪些数据实体或数据表？包含哪些字段？主键和外键是什么？实体之间存在什么关系？每种关系允许关联多少条记录？

ER 模型描述特定领域中彼此关联的实体。实际项目常用 Crow's Foot，也就是“乌鸦脚”记法表达基数。[14]

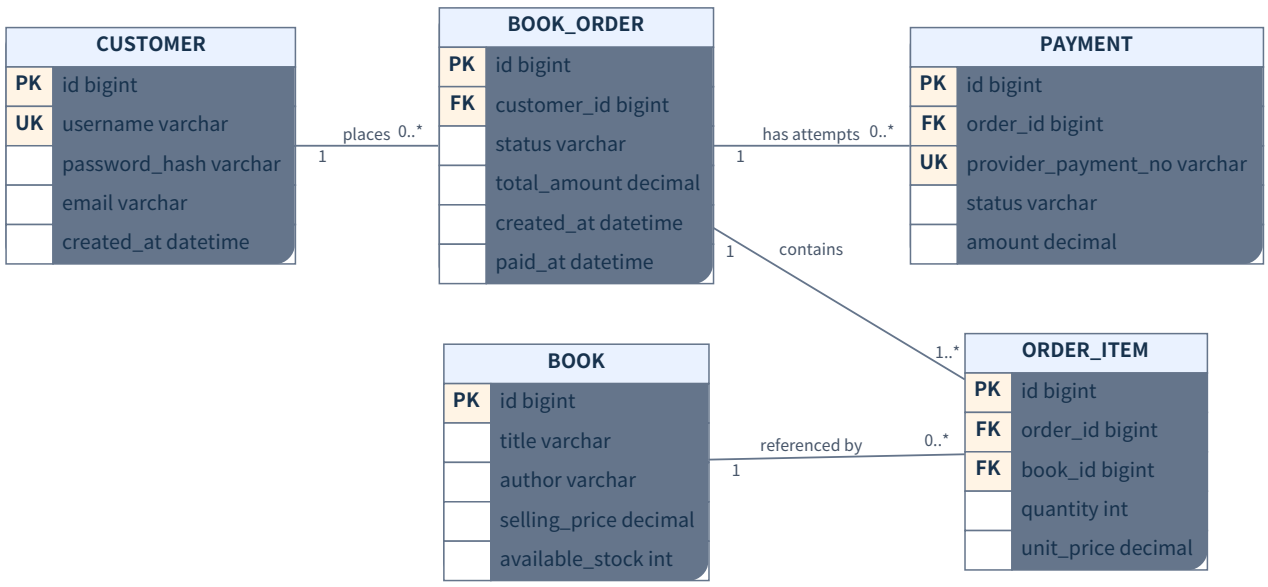


图 9：在线书城 ER 图。字段、键和连线共同表达数据库结构与约束。

10.2 实体、字段和键

矩形或表格框表示实体或数据表：

```
CUSTOMER
BOOK
BOOK_ORDER
ORDER_ITEM
PAYMENT
```

其中使用 `BOOK_ORDER` 而不是 `ORDER`，是为了避免与某些 SQL 语境中的关键字混淆。

标记	含义	示例
`PK`	Primary Key，主键，唯一标识一条记录	`BOOK_ORDER.id`
`FK`	Foreign Key，外键，引用另一张表的键	`BOOK_ORDER.customer_id -> CUSTOMER.id`
`UK`	Unique Key，唯一约束	`PAYMENT.provider_payment_no`

10.3 连线上的 `1` 和 `N` 是什么

最简化的写法：

```
CUSTOMER 1 ---- N BOOK_ORDER
```

表示一个顾客可以对应多张订单，每张订单属于一个顾客。`N` 不是固定数字，而是“多个”。

但 `1:N` 只表达最大数量，无法说明最小数量。例如顾客是否必须至少有一张订单？因此更精确的图会同时表达最小和最大基数。

10.4 常见基数

记法	含义	在线书城示例
`1` 或 `1..1`	恰好一个	每张订单必须属于一个顾客
`0..1`	零个或一个	用户可能还没有扩展资料
`0..*`	零个或多个	顾客可以没有订单，也可以有很多订单
`1..*`	一个或多个	已创建订单必须至少包含一个订单项

Crow’s Foot 记法中，圆圈通常表示“零”，竖线表示“一”，分叉的乌鸦脚表示“多个”。Mermaid ER 文档将 `零或一、恰好一、零或多、一或多` 分别定义为四类基数组合。[14]

10.5 怎样完整读取一条关系

```
CUSTOMER 1 ---- 0..* BOOK_ORDER : places
```

要从两个方向读：

- 一个顾客可以创建零张或多张订单；
- 每张订单必须且只能属于一个顾客。

再看：

```
BOOK_ORDER 1 ---- 1..* ORDER_ITEM : contains
```

表示一张订单必须包含一个或多个订单项，每个订单项只属于一张订单。

10.6 多对多怎样落到关系型数据库

订单和图书在业务上是多对多：

```
一张订单包含多本图书
一本图书可以出现在多张订单中
```

关系型数据库通常通过中间表拆分：

```
BOOK_ORDER 1 ---- N ORDER_ITEM
BOOK       1 ---- N ORDER_ITEM
```

`ORDER_ITEM` 还保存关系自身的属性：

```
quantity
unit_price
```

`unit_price` 应保存购买时的价格快照。否则图书后续调价可能破坏历史订单金额。

10.7 ER 连线不是调用箭头

ER 图中的线表示数据关系和基数约束，不是运行时调用，也不表示数据从一张表“流向”另一张表。

10.8 逻辑 ER 图与物理 ER 图

类型	重点
逻辑 ER 图	顾客、订单、图书、支付等业务实体和关系
物理 ER 图	真实表名、字段类型、主外键、可空性、唯一索引和组合索引

项目早期可先画逻辑模型，数据库技术与约束确定后再细化为物理模型。

十一、时序图：描述一个具体场景的运行时交互

11.1 它回答什么问题

时序图回答：

> 在某个具体场景中，参与者和系统组件按照什么时间顺序发送消息？调用和返回是什么？成功、失败、循环和并行路径怎样展开？

时序图是一种交互图，时间通常从上向下推进。[15]

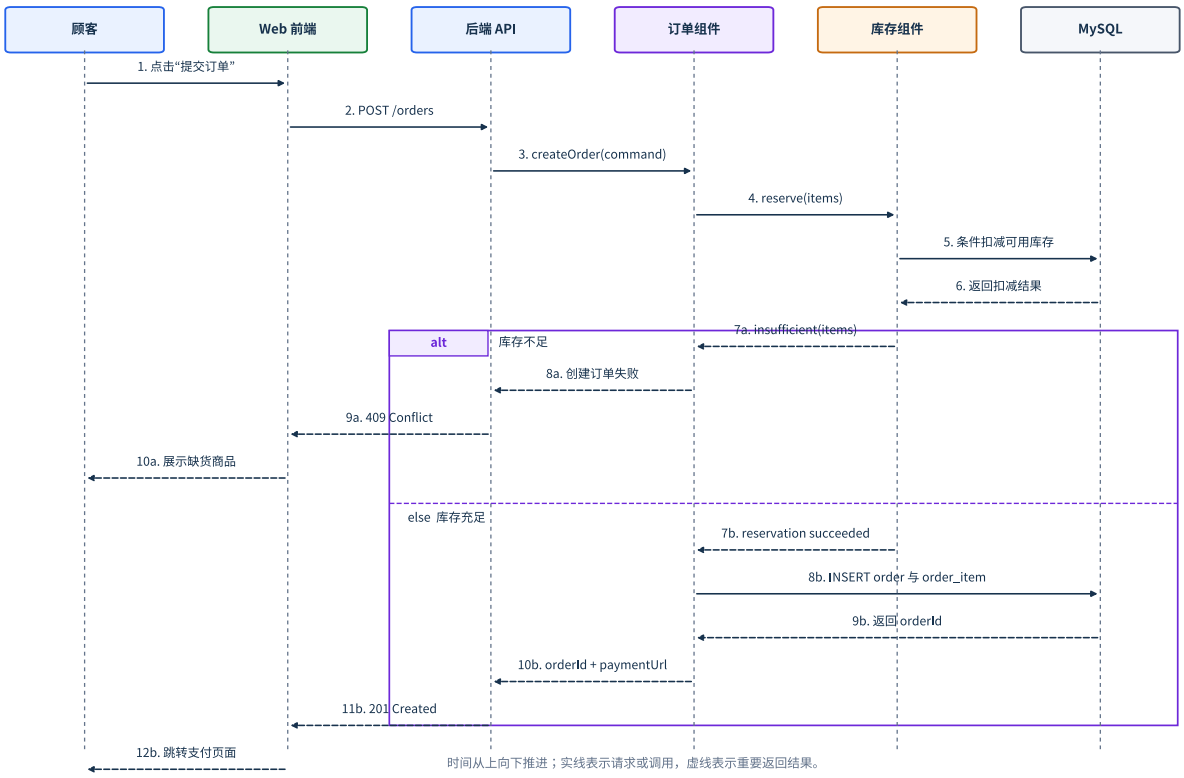


图 10：提交订单时序图。它把流程图中的“创建订单”展开为组件之间的具体交互。

11.2 主要元素

Participant：参与者

顶部的框表示交互参与者，可以是：

```
人
前端应用
后端服务
内部组件
数据库
消息队列
外部系统
```

Lifeline：生命线

从参与者向下延伸的虚线表示该参与者在当前场景中的存在时间。

Message：消息

横向箭头表示请求、方法调用、命令、事件或返回结果。消息名称必须有业务或技术含义：

```
POST /orders
createOrder(command)
reserve(items)
返回 orderId
```

不要全部写成“调用”“处理”“返回”。

同步调用、异步消息和返回

常见约定：

- 实线实心箭头：同步请求或方法调用；调用方通常等待结果。
- 开放箭头：异步消息；发送方通常不等待完整业务处理结束。
- 虚线箭头：重要返回结果。

具体箭头形状应在团队图例中统一。Mermaid 时序图也区分实线、虚线、箭头和异步开放箭头。[15]

Activation：激活条

生命线上的窄矩形表示参与者正在执行操作。高层图中可以省略；需要分析嵌套调用或持锁时间时再加入。

11.3 组合片段

片段	含义	在线书城示例
`alt`	互斥分支，相当于 if / else	库存不足 / 库存充足
`opt`	可选路径，没有 else	用户填写订单备注
`loop`	循环	逐项校验购物车中的图书

片段	含义	在线书城示例
`par`	并行	支付成功后并行通知、积分和统计

这些结构也是 Mermaid 时序图支持的标准表达。[15]

11.4 为什么支付回调应单独画图

顾客跳转支付平台后，支付结果可能几秒或几分钟后通过另一个 HTTP 请求回调在线书城。这不是“提交订单”请求中的同步返回。

单独画支付回调时序图，会暴露以下设计问题：

- 回调可能重复发送，接口必须幂等；
- 第三方支付号必须有唯一约束；
- 更新支付记录和订单状态需要清晰的事务边界；
- 通知发送失败不应回滚已确认的支付；
- 回调验签失败必须拒绝处理；
- 订单已经取消时收到成功回调，需要补偿策略。

时序图的价值不是把代码调用“抄一遍”，而是提前发现分布式交互中的超时、重试、并发、幂等和一致性风险。

十二、部署架构图：说明软件最终运行在哪里

12.1 它回答什么问题

部署图回答：

> 系统部署到哪些基础设施节点？每个应用有几个实例？网络、负载均衡、数据库和缓存怎样连接？这个图描述的是开发、测试还是生产环境？

C4 部署图用于说明某个明确环境中，软件系统或容器的实例怎样部署到物理或虚拟基础设施节点。部署节点可以表示设备、虚拟机、PaaS、容器平台、数据库服务器等，并且可以嵌套。[7]

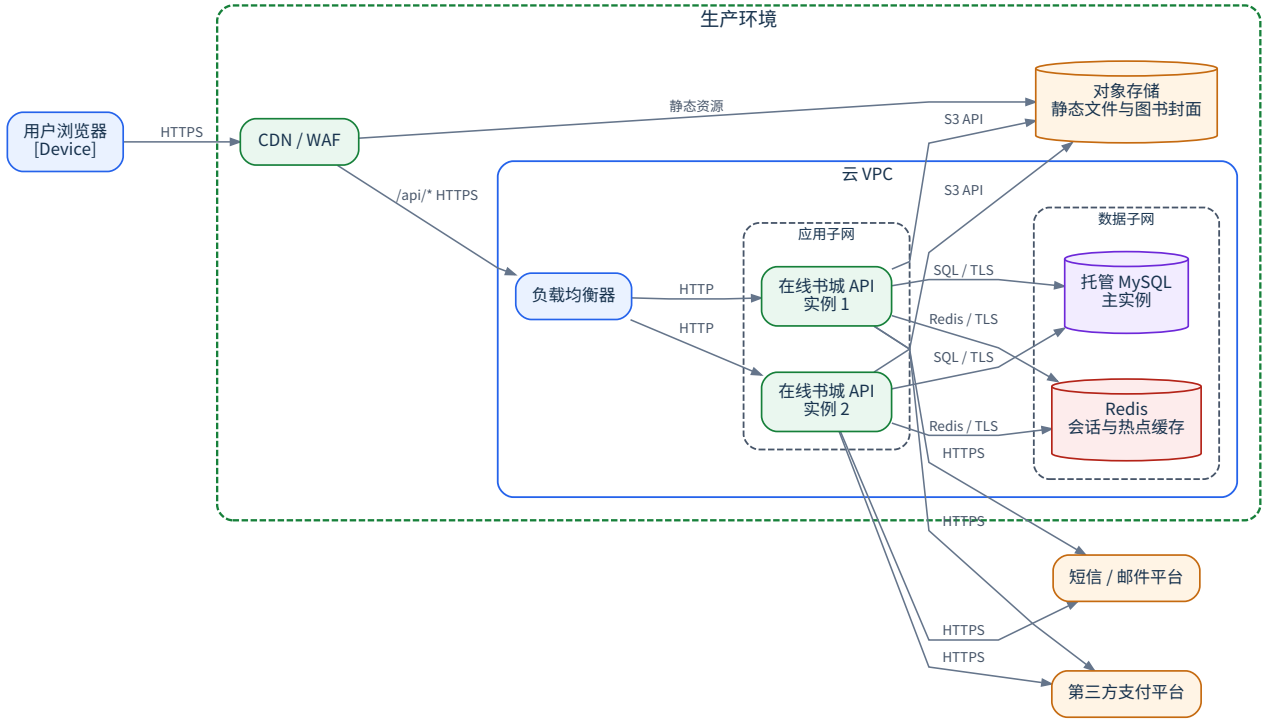


图 11：在线书城生产环境部署图。它表达实例数量、网络边界和基础设施关系。

12.2 部署节点

节点表示软件运行的位置：

```
用户设备
云 VPC
子网
虚拟机
Kubernetes 集群或 Pod
Docker 容器
托管数据库
对象存储
```

节点可以嵌套，从而表达“生产环境 → VPC → 应用子网 → API 实例”的层级。

12.3 软件实例

容器图中的逻辑元素是：

```
后端 API
```

部署图中则可能出现：

```
在线书城 API 实例 1
在线书城 API 实例 2
```

这说明生产环境运行两个相同容器的实例。

12.4 通信路径

部署图的连线应重点说明:

- 协议与端口;
- 是否加密;
- 公网还是内网;
- 防火墙和安全组;
- 服务发现与负载均衡;
- 故障切换和跨可用区;
- 访问第三方系统的出口路径。

12.5 容器图与部署图的区别

维度	容器图	部署图
关注点	逻辑应用、数据存储和职责	物理或虚拟运行环境
是否表达实例数量	通常不表达	表达
是否表达网络区域	通常不表达	经常表达
是否绑定环境	通常不绑定	绑定开发、测试、生产等环境
主要读者	架构、开发、运维	架构、运维、SRE、安全

十三、这些图怎样共同描述同一个系统

不同的图不是互相竞争，而是从不同视角观察同一个在线书城。

图	核心视角	在线书城中的一句话
系统上下文图	外部世界与系统边界	顾客、管理员、支付平台如何与在线书城交互
用例图	角色目标	顾客能浏览、下单、支付和查询订单

图	核心视角	在线书城中的一句话
业务流程图	业务执行路径	下单经过库存检查、订单创建和支付判断
状态机图	单个对象的生命周期	订单从待支付走向完成、取消或退款
类图	业务对象结构	顾客、购物车、订单、订单项和图书如何关联
容器图	高层逻辑架构	Web 前端、后端 API、MySQL 和对象存储如何协作
组件图	容器内部模块	后端由订单、库存、购物车和适配器等组件组成
ER 图	关系型数据结构	订单表通过外键关联顾客、订单项和支付记录
时序图	运行时交互顺序	提交订单时前端、API、订单、库存和数据库怎样调用
部署图	运行环境	两个 API 实例如何通过负载均衡访问 MySQL 和 Redis

13.1 典型的上下游关系

```
系统上下文图
    ↓ 确认边界和外部依赖
用例图 + 业务流程图
    ↓ 确认功能与业务规则
状态机图 + 领域模型
    ↓ 明确对象、状态和约束
容器图 + 组件图
    ↓ 确定技术结构和职责边界
ER 图 + 时序图
    ↓ 形成可实施的数据与交互设计
部署图
    ↓ 形成上线与运维方案
```

13.2 图与图之间应互相校验

- 上下文图出现支付平台，容器图和支付时序图中也应有对应外部依赖。
- 状态机中存在“退款中”，数据库状态字段和退款流程必须能表达它。
- ER 图要求订单至少一个订单项，创建订单时序图必须在同一业务操作中保证该约束。
- 容器图只有一个后端 API，部署图可以有多个 API 实例，但不应无缘无故出现另一个未定义的业务服务。
- 用例图中存在“订单发货”，流程、权限、状态转换和接口设计都要有落点。

矛盾往往不是文档问题，而是在提示真实设计尚未统一。

十四、怎样把图画好，而不是只把图画出来

14.1 一张图只回答一个主要问题

不要把以下内容塞到同一张图：

```
顾客
业务流程
Java 类
数据库字段
Kubernetes Pod
服务器 IP
```

它们属于不同抽象层级，应分别放入上下文图、流程图、类图、ER 图和部署图。

14.2 标题写清“系统 + 图类型 + 场景或环境”

不好的标题：

```
架构图
流程图
系统图
```

更好的标题：

```
在线书城 - 系统上下文图
在线书城 - 提交订单时序图
在线书城 - 订单状态机图
在线书城 - 生产环境部署图
在线书城后端 API - 组件图
```

14.3 每条关系线都要能读成一句话

例如：

```
顾客 浏览和购买图书 在线书城
在线书城 创建支付单 支付平台
订单组件 预占库存 库存组件
订单 属于 顾客
```

如果一条线无法读成一句明确的话，它很可能缺少标签、方向或基数。

14.4 保持同一抽象层级

容器图中适合放：

```
Web 前端
后端 API
MySQL
对象存储
```

下面这种组合通常不合适：

```
Web 前端
OrderController
MySQL 集群
顾客
Kubernetes Pod
```

因为它同时混入容器、代码类、人员和基础设施。

14.5 正常路径和异常路径都要设计

在线书城至少要考虑：

```
库存不足
支付失败
支付超时
支付回调重复
数据库写入失败
退款失败
通知发送失败
```

不需要全部塞进一张图，但关键异常必须有明确设计。

14.6 颜色只能辅助语义

图在黑白打印、深浅色主题和色觉障碍环境中仍应能依靠形状、边界、文字和线型被理解。只要使用自定义颜色或线型，就提供图例。

14.7 图应与代码一起版本管理

Mermaid 和 PlantUML 采用文本描述生成图，适合与 Markdown、ADR 和代码一起存入 Git。PlantUML 官方支持用文本定义用例图、时序图、类图、组件图、部署图和状态图；Mermaid 也提供流程、时序、类、状态和 ER 等图形语法。[9] [10]

推荐目录：

```
docs/
├── architecture/
│   ├── context.md
│   ├── containers.md
│   ├── components.md
│   └── deployment-production.md
├── business/
│   ├── order-flow.md
│   └── order-state.md
├── database/
│   └── erd.md
└── sequences/
    ├── create-order.md
    └── payment-callback.md
```

影响架构、状态或表结构的代码变更，应在同一个 Pull Request 中更新相应图。

十五、不同项目规模应画到什么程度

15.1 小型 CRUD 管理系统

优先准备：

1. 系统上下文图；
2. 核心业务流程图；
3. ER 图；
4. 简单容器图；
5. 上线前的部署图。

没有复杂生命周期和跨系统调用时，不必机械地画大量时序图和状态机图。

15.2 包含交易的普通 Web 系统

类似本文在线书城，建议至少有：

1. 一张系统上下文图；
2. 一张核心用例图；
3. 两到三张核心业务流程图；
4. 一张订单状态机图；
5. 一张容器图；
6. 一张核心 ER 图；
7. 两张关键时序图，例如“提交订单”和“支付回调”；
8. 一张生产环境部署图。

组件图在后端模块增多、职责开始模糊时再补充。

15.3 微服务或大量外部集成的系统

还应重点补充：

- 服务边界和数据所有权；
- API 与异步事件契约；
- 消息队列、Topic 和消费者关系；
- 分布式事务、补偿和最终一致性；
- 超时、重试、熔断和降级；
- 多环境和多区域部署；
- 网络信任边界与威胁模型；
- 可观测性链路和故障定位路径。

服务数量增加不等于把所有服务塞进一张巨大图。应按业务域、场景或系统边界拆分。

十六、还有哪些图可能在前期出现

本文十类图构成主干，但不同项目还可能需要：

图或模型	适用场景
UI 线框图 Wireframe	确认页面结构、控件和信息层级
页面导航图	展示页面之间的跳转关系

图或模型	适用场景
用户旅程图	分析用户完整体验、情绪和痛点
数据流图 DFD	分析数据从哪里来、经过什么处理、存到哪里
威胁模型图	标记信任边界、敏感数据和攻击面
网络拓扑图	描述子网、防火墙、网关、VPN 和网络设备
事件风暴 Event Storming	发现领域事件、命令、聚合和业务规则
决策表	表达大量条件组合及其动作
API 依赖图	展示接口提供者、消费者和版本关系
数据血缘图	描述数据来源、加工过程和去向
WBS / 甘特图	项目任务拆分、依赖和排期

是否需要它们，仍取决于项目风险，而不是模板要求。

十七、在线书城从立项到开工的推荐顺序

第一步：系统上下文图

确认目标系统、用户、支付和通知等外部依赖，形成清晰责任边界。

第二步：用例图

确认顾客和管理员分别需要哪些能力，哪些需求本期不做。

第三步：核心业务流程

至少覆盖提交订单、支付、发货和退款，并画出关键异常路径。

第四步：订单状态机

明确待支付、已支付、已发货、已完成、已取消、退款中和已退款，以及所有合法转换。

第五步：领域模型

识别 `Customer`、`Book`、`ShoppingCart`、`Order`、`OrderItem`、`Payment` 和 `Inventory`，建立统一业务语言。

第六步：容器架构图

确定 Web 前端、后端 API、MySQL、对象存储及外部集成方式。

第七步：必要时画组件图

确定订单、库存、购物车、持久化和外部适配器的职责边界。

第八步：ER 图

确定真实表、字段、主外键、唯一约束、基数和索引策略。

第九步：关键时序图

检查事务边界、库存并发、支付幂等、超时、重试和异常处理。

第十步：部署图

确定生产环境实例、网络、数据库、缓存、负载均衡和外部访问路径。

十八、开工前最终检查表

检查问题	主要依据
系统边界和外部依赖是否清楚？	系统上下文图
角色与功能范围是否清楚？	用例图
核心业务及异常路径是否清楚？	业务流程图 / BPMN
订单状态和转换规则是否完整？	状态机图
核心业务概念是否统一？	领域模型 / 类图
系统高层结构是否清楚？	容器图
后端模块职责是否清楚？	组件图
数据表、键和基数是否清楚？	ER 图
关键请求调用顺序和失败策略是否清楚？	时序图
生产环境如何运行是否清楚？	部署图
图与需求、接口、代码和数据库是否一致？	跨图审查与评审记录

检查问题	主要依据
图的负责人和更新机制是否明确?	文档治理规则

最终目标不是“图画得多”，而是让团队对以下问题形成一致理解：
我们要做什么，不做什么；业务规则是什么；系统怎样拆分；数据怎样保存；关键流程怎样执行；软件最终怎样运行。

对于大多数普通 Web 项目，最值得优先维护的组合是：**系统上下文图 + 核心业务流程图 + 状态机图 + 容器图 + ER 图 + 关键时序图 + 生产部署图**。

参考资料

本文优先采用标准组织或项目官方英文资料；链接访问日期为 2026-07-18。

1. Object Management Group, “Unified Modeling Language (UML)”. https://www.omg.org/uml/
2. Object Management Group, “Business Process Model and Notation 2.0.2”. https://www.omg.org/spec/BPMN/2.0.2/About-BPMN
3. C4 Model, “Diagrams”. https://c4model.com/diagrams
4. C4 Model, “System Context Diagram”. https://c4model.com/diagrams/system-context
5. C4 Model, “Container Diagram”. https://c4model.com/diagrams/container
6. C4 Model, “Component Diagram”. https://c4model.com/diagrams/component
7. C4 Model, “Deployment Diagram”. https://c4model.com/diagrams/deployment
8. PlantUML, “Use Case Diagram”. https://plantuml.com/use-case-diagram
9. PlantUML, “Supported UML Diagrams”. https://plantuml.com/
10. Mermaid, “Diagram Syntax”. https://mermaid.js.org/intro/syntax-reference.html
11. Mermaid, “Flowcharts Syntax”. https://mermaid.js.org/syntax/flowchart.html
12. Mermaid, “State Diagrams”. https://mermaid.js.org/syntax/stateDiagram.html
13. Mermaid, “Class Diagrams”. https://mermaid.js.org/syntax/classDiagram.html
14. Mermaid, “Entity Relationship Diagrams”. https://mermaid.js.org/syntax/entityRelationshipDiagram.html
15. Mermaid, “Sequence Diagrams”. https://mermaid.js.org/syntax/sequenceDiagram.html