**Design Inventory system**

**场景：  
你需要设计一个库存管理系统（Inventory System），用于管理商品库存和订单操作，保证库存在高并发环境下的正确性，同时考虑系统可扩展性和安全性。为我们的一些零售合作伙伴建立‘暗店’（dark stores）。这些是专门的门店，备有全品类的杂货，不对公众开放，只供 Instacart 的购物员使用。**

**核心功能要求**

1. **Add stock（增加库存）**
   * **支持补货或增加仓库库存数量。**
2. **Reserve stock（预留库存）**
   * **当用户下单时，将库存临时锁定，防止其他订单抢占（防止超卖）。**
3. **Fulfill stock（完成订单）**
   * **当订单完成支付或发货时，真正扣减库存。**
4. **Clear reservation（清理预留）**
   * **订单取消或超时未支付时，将预留库存释放回可用库存。**

**关键非功能点**

1. **并发控制 / race condition**
   * **多个用户同时下单时，需要防止库存超卖。**
   * **面试官可能希望你讨论事务、锁、乐观/悲观并发控制或 Redis 原子操作。**
2. **扩展性 / Scaling**
   * **系统需要支持高并发访问。**
   * **可考虑缓存、分库分表、异步消息队列等。**
3. **安全性**
   * **防止非法库存操作。**
   * **API 鉴权、权限控制、加密传输等。**
4. **Production Deployment Best Practices**
   * **高可用 DB、负载均衡、日志监控、备份、CI/CD 部署。**

**面试考察重点**

* **业务流程理解：Add → Reserve → Fulfill / Clear**
* **事务与并发处理：防止库存被重复预留或扣减**
* **系统可扩展性：缓存、异步处理、分库分表**
* **安全性与运维：权限控制、监控、高可用部署**

**🎯 System Design 面试开场 5 问模板**

**1️⃣ 范围 / Scope**

* **ZH: “请问这个系统只针对暗店，还是要同时支持普通门店？”**
* **EN: “Is this system focused only on dark stores, or should it also support normal retail stores?”**

**👉 明确设计范围，避免过度或不足。**

**2️⃣ 功能需求 / Functional Requirements**

* **ZH: “库存预留是否需要 TTL？也就是用户下单但不付款时，系统要不要自动释放库存？”**
* **EN: “Do reservations need a TTL, so if a customer doesn’t pay, the system automatically releases the stock?”**

**👉 确认核心业务逻辑。**

**3️⃣ 非功能需求 / Non-functional Requirements**

* **ZH: “大概的规模和并发量是多少？读写比例是怎样的，比如查询是否远多于下单？”**
* **EN: “What’s the expected scale and concurrency? And what’s the read/write ratio—for example, are reads much higher than writes?”**

**👉 决定是否需要缓存、分库分表、事件驱动。**

**4️⃣ 一致性 vs 性能 / Consistency vs Performance**

* **ZH: “对一致性的要求是什么？必须强一致，还是允许一定的最终一致性？”**
* **EN: “What’s the consistency requirement? Do we need strong consistency, or is eventual consistency acceptable?”**

**👉 决定是否要用强事务，还是可以牺牲一点一致性换性能。**

**5️⃣ 业务约束 & 安全 / Business Constraints & Security**

* **ZH: “谁可以调用这些 API？是顾客直接调用，还是都通过 order service？另外是否需要审计日志，记录每次库存变更的 who/what/when/why？”**
* **EN: “Who is allowed to call these APIs—do customers call directly, or only through the order service? And do we need full audit logs to record who/what/when/why for every inventory change?”**

**👉 确认鉴权、权限控制和合规性需求。**

**✨ 开场口播示例**

* **ZH:  
  “在开始设计之前，我想先澄清几个点。第一，这个系统只针对暗店，还是所有门店都要支持？第二，库存预留是否需要 TTL 自动清理？第三，预期的规模和并发量是多少，读写比例大概如何？第四，我们对一致性的要求是强一致还是可以最终一致？第五，谁能调用 API，是顾客还是只有订单服务？另外是否需要审计日志记录所有变更？这些信息明确了，我才能更好地做架构设计。”**
* **EN:  
  “Before diving into the design, I’d like to clarify a few things. First, is this system only for dark stores or all stores? Second, do reservations require a TTL for automatic expiry? Third, what’s the expected scale and read/write ratio? Fourth, do we need strong consistency or is eventual consistency acceptable? And fifth, who is authorized to call the APIs—customers directly or only via the order service—and do we need audit logs of all inventory changes? With these clarified, I can make better architectural trade-offs.”**

**Design: Inventory System (High-level) / 库存系统高层设计**

**0) 范围与假设 / Scope & Assumptions**

* **ZH:** 系统需要支持多门店/仓库（含暗店 Dark Store），每个商品以 SKU 唯一标识。
* **EN:** The system must support multiple stores/warehouses (including dark stores), with each product uniquely identified by a SKU.
* **ZH:** 实时可用量与防超卖是核心目标。
* **EN:** Real-time availability and preventing overselling are the core goals.
* **ZH:** 系统需要承受高并发的读写请求。
* **EN:** The system must handle high-concurrency read and write traffic.
* **ZH:** 数据库是库存的最终真相源（Source of Truth）；缓存只作为读优化。
* **EN:** The database is the source of truth for inventory; cache is only for read optimization.

**1) 核心概念与状态 / Core Concepts & States**

* **ZH:** 关键公式：available = on\_hand − reserved (必须始终 ≥ 0)
* **EN:** Key formula: available = on\_hand − reserved (must always be ≥ 0)
* **ZH:** 状态流转：
* Available --(Reserve)--> Reserved --(Fulfill)--> Shipped/Completed
* Reserved --(Clear/Expire)--> Available
* **EN:** State transitions:
* Available --(Reserve)--> Reserved --(Fulfill)--> Shipped/Completed

Reserved --(Clear/Expire)--> Available

**- CDN (Edge Cache) — *in front of the Load Balancer*  
为何 / Why: 缓存静态资源与热门分类的 JSON，降低首字节延迟与源站压力。 / Cache static assets and hot JSON responses for popular categories, reducing time-to-first-byte and lowering load on the origin servers.**

**Flow: Client → CDN → LB → API Gateway → API…**

**- WAF / Rate Limiter — *between CDN and API Gateway (or at the gateway)*  
为何 / Why: 基础防护与限流，抵御恶意流量、避免突发把后端打爆。 / Provide basic protection and throttling, defending against malicious traffic and preventing sudden spikes from overwhelming backend services.  
Flow: Client → CDN → WAF → LB → Gateway…**

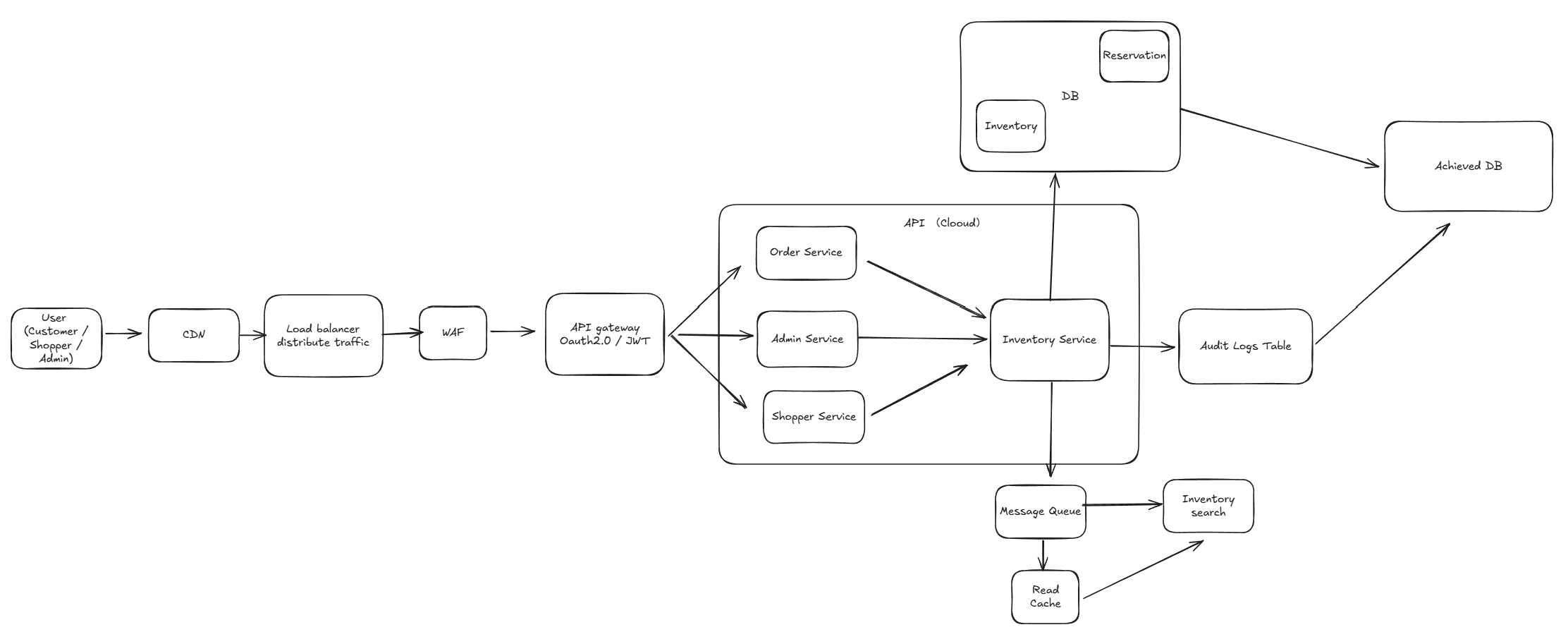
**- Object Storage (e.g., S3) + CDN — *out of band for images*  
为何 / Why: 图片放对象存储作为源站，前置 CDN 分发；数据库里只存 key/path。 / Store images in object storage as the origin, with CDN distribution in front; keep only the key or path in the database.**

**Flow: Client → CDN (images) → S3 (on miss)**

**- Observability Stack (Metrics + Tracing + Logs) — *sidecar/service level*  
为何/Why: 监控缓存命中率、P95 延迟、错误率；追踪链路（LB→GW→API→Cache/Replica/DB）。  
Flow: API/Cache/DB emit → Metrics/Tracing backend**

**- Secrets / Config Management — *platform layer (e.g., KMS + config store)*  
为何 / Why: 安全管理令牌、数据库凭据、限流阈值与开关，支持灰度与快速回滚。 / Securely manage tokens, DB credentials, and feature flags; support canary releases and fast rollback with centralized config.**

**- Health Checks & Blue/Green (or Canary) Deploy — *LB + deploy pipeline*  
为何 / Why: 平滑发布与失效转移；负载均衡器仅把流量打到健康实例。 / Ensure smooth deployments and failover; the load balancer only routes traffic to healthy instances.**



**A user request first goes through the CDN, which caches static assets like images and product pages close to the user to reduce latency. The request then passes through a Web Application Firewall (WAF), which filters malicious traffic such as SQL injection or DDoS attempts. After that, it reaches the Load Balancer, which distributes traffic to healthy nodes.**

**From there, it flows into the API Gateway, where the token is validated and the request is routed to one of our stateless API servers running in the cloud. We have two main business services: the Order Service, which handles the customer-facing order lifecycle—placement, payment, cancellation—and the Shopper Service, which handles order acceptance and fulfillment by shoppers.**

**All inventory changes go through the Inventory Service, which is the single source of truth. It manages adds, reserves, fulfillments, and clears, always persisting through the database with strong consistency. This separation keeps responsibilities clear: Order manages lifecycle, Shopper handles execution, and Inventory guarantees stock correctness.**

**On the read side, traffic is served through Redis and the Catalog/Search service for fast queries. Meanwhile, a message queue broadcasts inventory changes in real time so caches, search, reporting, and fraud detection systems can stay up to date without putting extra load on the database.**

**2) 系统组件（High level） / System Components (High level)**

**1) API Gateway**

**ZH（作用）：** 统一入口，负责请求路由、流控、鉴权与审计。对外隐藏内部服务拓扑，支持金丝雀/灰度放量与全链路追踪的入口打点。可在异常或高压时做限流与熔断，保护下游服务与数据库。  
**EN (Role):** Single entry point for routing, throttling, authentication, and auditing. It hides internal topology, supports canary/gradual rollouts, and injects tracing headers. Under stress, it applies rate limiting and circuit breaking to protect downstream services and the DB.  
**关键交互 / Key interactions:** 路由到 Order/Shopper/Inventory/Catalog；校验身份（JWT/OIDC）；打统一审计日志。

**2) Order Service**

**ZH（作用）：** 面向顾客的订单生命周期编排者：创建订单、计算价格/优惠、发起支付、处理取消/退款。它**不直接改库存**，而是调用 Inventory Service 完成 **Reserve** 和 **Clear**，确保库存与订单状态一致。  
**EN (Role):** Customer-facing orchestrator for the order lifecycle: create orders, pricing/discounts, initiate payment, handle cancel/refund. It **never edits inventory directly**; it calls Inventory Service to **Reserve** and **Clear** so order state and stock remain consistent.  
**关键交互 / Key interactions:**

* 创建订单前后查询 Catalog/Search 展示可用量；
* 下单时调用 Inventory.Reserve（含 TTL）；
* 取消/支付超时调用 Inventory.Clear；
* 支付成功后通知 Shopper Service 生成拣货任务。

**3) Shopper Service**

**ZH（作用）：** 面向拣货员（shopper）的执行层：接单、任务分配、拣货确认、异常替换（替换同类 SKU）、出库/交付。订单执行到位后调用 Inventory Service **Fulfill**，完成最终扣减。  
**EN (Role):** Shopper-facing execution layer: accept assignments, pick items, handle exceptions/substitutions, and mark out-for-delivery. After successful picking, it calls Inventory Service **Fulfill** for the final deduction.  
**关键交互 / Key interactions:**

* 从 Order/分配系统接收任务；
* 拣货确认后调用 Inventory.Fulfill；
* 异常（缺货/替换）时与 Order 协调、回滚或局部 Clear。

**4) Inventory Service**

**ZH（作用）：** 库存**唯一真相源**（Source of Truth），对外暴露四个核心能力：**Add**（补货/退货）、**Reserve**（预留，带 TTL）、**Fulfill**（实扣）、**Clear**（释放预留）。内部使用事务与条件更新/锁来防止超卖，并发布库存变更事件给下游。  
**EN (Role):** The **single source of truth** for stock. Exposes four core operations: **Add**, **Reserve** (with TTL), **Fulfill**, **Clear**. Enforces invariants with transactional conditional updates/locking to prevent overselling, and publishes change events downstream.

Inventory Service is focused on stock itself—it’s the single source of truth for adds, reserves, fulfillments, and clears. Order Service manages the order lifecycle: placement, payment, cancellation. It never changes stock directly but calls Inventory Service to do so. This separation keeps responsibilities clear and ensures inventory consistency.

**关键交互 / Key interactions:**

接收 Order/Shopper 的调用；

维护 Inventory 与 Reservation 两张核心表；

经 Event Bus 广播 InventoryUpdated/Reserved/Released/Fulfilled。

**5) DB (OLTP：Inventory & Reservation)**

**ZH（作用）：** 关系型数据库承载强一致写入：Inventory(store\_id, sku\_id, on\_hand, reserved, version) 与 Reservation(reservation\_id, status, expires\_at, idempotency\_key …)。通过条件更新/乐观锁或行级锁，保证 available = on\_hand - reserved ≥ 0 不变量。  
**EN (Role):** Relational OLTP backing strong-consistency writes for Inventory and Reservation. Uses conditional updates/optimistic versioning or row locks to enforce the invariant available = on\_hand - reserved ≥ 0.  
**关键交互 / Key interactions:** 作为最终裁决者；触发 Outbox/CDC 将变更推送到 Event Bus；支撑回放/审计与 PITR 恢复。

**6) Read Cache (Redis)**

**ZH（作用）：**  
面向读的加速层，缓存 (store\_id, sku\_id) 的 **available** 等只读数据。写入仍以 DB 为准，缓存不是“真相源”，而是加速查询的副本。缓存数据的更新主要通过 **消息队列 (MQ) 推送事件刷新**，同时配合短 TTL 兜底，避免因消息丢失导致长时间不一致。这样能极大减轻数据库的高频读压力。

**EN (Role):**  
Read-side accelerator storing availability per (store, sku) as a **read-only view**. The DB remains authoritative for writes; the cache is just a fast replica. Cache freshness is maintained via **event-driven updates from the message queue**, with short TTLs as a fallback. This offloads heavy read traffic from the DB.

**关键交互 / Key interactions:**

* **ZH:** Catalog/Search 在查询时优先读缓存；缓存由 MQ 事件驱动刷新，并用短 TTL 兜底。
* **EN:** Catalog/Search reads from cache first; cache is refreshed by MQ events and falls back to short TTLs for safety.

**7) Message Queue (Kafka 等)**

**ZH（作用）：**  
消息队列是库存事件的中转站。当库存有变化（Reserve、Fulfill、Clear、Add），**Inventory Service 更新 DB 成功后会发事件到 MQ**。其他系统（Search Service、报表、风控、缓存刷新器）订阅这些事件，收到后更新自己的缓存或索引。这样库存服务只写一次，下游都能实时同步，数据库也不会被下游读压垮。

**EN (Role):**  
The message queue is the backbone for inventory events. When inventory changes (Reserve, Fulfill, Clear, Add), the **Inventory Service publishes an event after committing the DB update**. Other systems—Search Service, reporting, fraud detection, cache updater—subscribe and refresh their own views. This way, Inventory only writes once and all downstream stays in sync without hammering the DB.

**关键交互 / Key interactions:**

* **ZH:** Inventory 更新成功 → 发事件到 MQ → 消费者刷新缓存、更新搜索索引、生成报表。
* **EN:** Inventory commit success → event published to MQ → consumers refresh cache, update search index, generate reports.
* MQ is an append-only event stream, **not a query store**. It can’t serve random reads, Use MQ to **propagate changes**, and a cache to **serve queries**. Without a cache, Search can’t do random access, must handle replay/idempotency itself, and becomes fragile.

**8) Catalog / Search Service**

**ZH（作用）：**  
顾客前台的展示服务，负责商品信息、价格和实时可用量，支持搜索和排序。它是 **读多写少** 的典型场景：查询时优先读缓存 (Redis)，未命中再回 DB。与此同时，它会 **订阅消息队列 (MQ)**，在库存有变化时，主动刷新缓存和搜索索引，保证顾客不会看到已缺货的商品。

**EN (Role):**  
The customer-facing presentation service for product data, pricing, and availability, supporting search and sort. It’s highly read-heavy: queries go to **Redis cache** first, with DB fallback on misses. It also **subscribes to the message queue (MQ)**, so whenever inventory changes, it refreshes cache and search indexes to avoid showing out-of-stock items.

**关键交互 / Key interactions:**

* **ZH:** 查询时 → 优先读 Redis；  
  库存变化 → 订阅 MQ → 刷新缓存和搜索索引。
* **EN:** For queries → read Redis first;  
  On inventory changes → consume from MQ → refresh cache and search index.

**一句话总览 / One-line overview**

* **ZH:** “写入由 Inventory+DB 强一致裁决，Order/Shopper 分别处理下单和拣货；读流量通过 Redis 与 Catalog/Search；消息队列负责把库存变更实时广播给缓存和下游系统。”
* **EN:** “All writes are committed by Inventory + DB with strong consistency; Order and Shopper handle ordering and picking; reads go through Redis and Catalog/Search; the message queue broadcasts inventory changes in real time to caches and downstream systems.”

**3) 数据模型（最小可行） / Data Model (Minimal viable)**

**Inventory（每行=一个门店×一个SKU）**

**字段 / Columns**

| **字段** | **类型** | **约束** | **作用（ZH / EN）** |
| --- | --- | --- | --- |
| store\_id | BIGINT | PK(1/2) | 门店ID / Store identifier |
| sku\_id | BIGINT | PK(2/2) | 商品SKU / Product SKU |
| on\_hand | INT | NOT NULL, DEFAULT 0 | 实物在库 / Physical stock on hand |
| reserved | INT | NOT NULL, DEFAULT 0 | 已预留量 / Reserved for orders |
| version | BIGINT | NOT NULL, DEFAULT 0 | 乐观并发版本 / Optimistic concurrency version |
| updated\_at | TIMESTAMP | NOT NULL | 最近更新时间 / Last update timestamp |

**索引与约束 / Indexes & Constraints**

* PRIMARY KEY (store\_id, sku\_id) → 复合主键确保每个门店 × SKU 唯一一行。 / Composite PK ensures one row per store × SKU.
* CHECK (on\_hand >= 0 AND reserved >= 0) → 确保数量不为负。 / Ensures on\_hand and reserved are non-negative.
* CHECK (on\_hand - reserved >= 0) → 确保可用量不为负。 / Guarantees availability (on\_hand - reserved) never goes below zero.

**乐观锁 (Optimistic Locking)**

* **ZH:** 通过 version 字段实现。更新时带上 WHERE version = ?，如果版本匹配则更新成功，并将 version 加 1。如果两个请求同时到达，只有第一个成功，第二个因为 version 不一致而失败。这样就避免了并发更新互相覆盖，从而防止超卖。
* **EN:** Implemented with a version column. Each update includes WHERE version = ?. If the version matches, the row is updated and version increments. If two requests arrive at the same time, the first succeeds and the second fails due to version mismatch. This prevents lost updates and overselling.

**悲观锁 (Pessimistic Locking)**

* **ZH:** 使用 SELECT … FOR UPDATE 在事务中锁定行。数据库会在该行上加排他锁，直到事务提交或回滚。在这期间，其他事务不能修改同一行。这样能保证并发更新是串行执行的，从而彻底避免超卖。
* **EN:** Achieved with SELECT … FOR UPDATE. This places an exclusive lock on the row within the transaction until commit or rollback. During this time, no other transaction can modify the row. This enforces serialized updates, preventing overselling.

**Reservation（预留记录）**

**字段 / Columns**

| **字段** | **类型** | **约束** | **作用（ZH / EN）** |
| --- | --- | --- | --- |
| reservation\_id | UUID | PK | 预留ID / Reservation ID |
| store\_id | BIGINT | NOT NULL | 门店ID / Store ID |
| sku\_id | BIGINT | NOT NULL | SKU |
| user\_id | BIGINT | NOT NULL | 用户ID / Customer ID |
| order\_id | BIGINT | NULL | 订单ID（下单后可回填）/ Order ID (nullable) |
| qty | INT | NOT NULL, CHECK(qty > 0) | 预留数量 / Reserved quantity, must be greater than 0. A reservation means “locking some stock.” If qty = 0, it has no effect and the record is meaningless. |
| status | SMALLINT | NOT NULL | 状态枚举 / Status enum（ACTIVE=1, RELEASED=2, FULFILLED=3, EXPIRED=4） |
| expires\_at | TIMESTAMP | NOT NULL | 过期时间 / TTL deadline, used for automatic release. |
| idempotency\_key | TEXT | UNIQUE | 幂等键 / Idempotency key, ensures retries don’t create duplicates. |
| created\_at | TIMESTAMP | NOT NULL | 创建时间 / Created at |
| updated\_at | TIMESTAMP | NOT NULL | 更新时间 / Updated at |

**索引与约束 / Indexes & Constraints**

* PRIMARY KEY (reservation\_id) → 保证每条预留唯一。 / Ensures each reservation is unique.
* FOREIGN KEY (store\_id, sku\_id) → 引用 Inventory(store\_id, sku\_id)，保持库存与预留一致性（可选，高并发下可关闭 FK，只在应用层校验）。 / References Inventory(store\_id, sku\_id) to enforce consistency between stock and reservations (optional, can be disabled under high concurrency).
* FOREIGN KEY (order\_id) → 引用 Orders(order\_id)，用于订单与预留的关联（可选）。 / References Orders(order\_id) for linking reservations to orders (optional).
* UNIQUE (idempotency\_key) → 确保 API 幂等性。 / Guarantees API idempotency.
* INDEX (store\_id, sku\_id, status) → 按 SKU 查活动预留。 / Query active reservations by SKU.
* INDEX (user\_id, status) → 查用户的购物车或订单预留。 / Query user’s active reservations.
* INDEX (status, expires\_at) → 扫描过期记录做清理。 / TTL cleanup scan on expired reservations.

**PostgreSQL 示例 DDL（带注释 / With comments）**

-- INVENTORY

CREATE TABLE inventory (

store\_id BIGINT NOT NULL,

sku\_id BIGINT NOT NULL,

on\_hand INT NOT NULL DEFAULT 0,

reserved INT NOT NULL DEFAULT 0,

version BIGINT NOT NULL DEFAULT 0,

updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),

CONSTRAINT pk\_inventory PRIMARY KEY (store\_id, sku\_id),

CONSTRAINT chk\_nonneg CHECK (on\_hand >= 0 AND reserved >= 0),

CONSTRAINT chk\_available CHECK (on\_hand - reserved >= 0)

);

COMMENT ON TABLE inventory IS '每门店×SKU一行；DB为真相源 / One row per (store, sku); DB is SoT';

COMMENT ON COLUMN inventory.version IS '乐观并发 / Optimistic concurrency';

-- RESERVATION

CREATE TABLE reservation (

reservation\_id UUID PRIMARY KEY,

store\_id BIGINT NOT NULL,

sku\_id BIGINT NOT NULL,

user\_id BIGINT NOT NULL,

order\_id BIGINT NULL,

qty INT NOT NULL,

status SMALLINT NOT NULL, -- 1=ACTIVE,2=RELEASED,3=FULFILLED,4=EXPIRED

expires\_at TIMESTAMP NOT NULL,

idempotency\_key TEXT NOT NULL UNIQUE,

created\_at TIMESTAMP NOT NULL DEFAULT NOW(),

updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),

CONSTRAINT chk\_qty\_pos CHECK (qty > 0)

-- 可按需开启外键（高并发下可只做应用校验）

-- ,CONSTRAINT fk\_res\_inventory FOREIGN KEY (store\_id, sku\_id) REFERENCES inventory(store\_id, sku\_id)

);

CREATE INDEX idx\_res\_store\_sku\_status ON reservation (store\_id, sku\_id, status);

CREATE INDEX idx\_res\_user\_status ON reservation (user\_id, status);

CREATE INDEX idx\_res\_expire ON reservation (status, expires\_at);

At the data model level, I keep it simple with just two core tables: **Inventory** and **Reservation**.  
The Inventory table has one row per store and SKU. It tracks two numbers: how many items are on hand and how many are already reserved. There’s also a version field for optimistic concurrency, and I enforce constraints directly in the database—for example, both on\_hand and reserved must be non-negative, and on\_hand minus reserved can never drop below zero. That way, even if the application has a bug, the database itself prevents overselling.

The Reservation table records each hold when a customer adds items to their basket. It has a UUID reservation ID, the store and SKU, the user ID, the quantity, and a status that goes from ACTIVE to either RELEASED, FULFILLED, or EXPIRED. There’s also an expiry time so we can automatically release holds that time out. Each reservation carries an idempotency key to make retries safe and avoid duplicates. I also add indexes for the main access patterns: by store and SKU to check active reservations, by user to show their basket, and by expiry time so a background job can efficiently clean up expired holds.

With just these two tables, we cover all the flows: **add stock** updates on\_hand, **reserve** increases reserved and creates a reservation, **fulfill** consumes the reservation and deducts from on\_hand, and **clear** releases the reservation back into available stock.

**一眼可读的对照总结 / Quick Visual Summary**

**Inventory**

* Key: (store\_id, sku\_id)
* Numbers: on\_hand, reserved, version
* Rule: on\_hand - reserved >= 0

**Reservation**

* Key: reservation\_id, idempotency\_key (unique)
* Links: store\_id, sku\_id, user\_id, order\_id?
* Lifecycle: status, expires\_at
* Query Paths: (store, sku, status), (user, status), (status, expires\_at)
* **解释 / Explanation**
  + **ZH:** Fulfill → reserved 减少，同时 on\_hand 真正扣减。
  + **EN:** Fulfill → reserved decreases, on\_hand is actually deducted.
  + **ZH:** Clear → reserved 释放回 available。
  + **EN:** Clear → reserved is released back to available.

**4) 对外 API（核心四个） / Core APIs (4 endpoints)**

**所有写操作都支持 Idempotency-Key，避免重复请求。 / All write operations require Idempotency-Key to ensure idempotency.**

**1. Add stock（增加库存）**

* **Endpoint → POST /inventory/stock**
* **Body:**
  + **store\_id → 门店 ID / Store identifier**
  + **sku\_id → 商品 SKU / Product SKU**
  + **qty\_delta → 增加数量，可为正（补货）或负（退货） / Quantity change, positive (restock) or negative (return)**
  + **reason → 原因（补货、退货等） / Reason (restock, return, etc.)**
* **Response:**
  + **on\_hand → 更新后的实际库存 / Updated on\_hand quantity**
  + **reserved → 当前已预留数量 / Current reserved quantity**
  + **available → 可用量 (on\_hand - reserved) / Available stock (on\_hand - reserved)**

**2. Reserve stock（预留库存）**

* **Endpoint → POST /reservations**
* **Body:**
  + **store\_id → 门店 ID / Store identifier**
  + **sku\_id → 商品 SKU / Product SKU**
  + **qty → 预留数量，必须 > 0 / Quantity reserved, must be > 0**
  + **user\_id → 用户 ID / Customer identifier**
  + **ttl\_sec → 预留有效期（秒），超时未支付会自动释放 / TTL in seconds, auto-release if not fulfilled**
* **Response:**
  + **reservation\_id → 预留 ID / Reservation identifier**
  + **expires\_at → 过期时间 / Expiration timestamp**
  + **available\_after → 预留完成后的剩余可用量 / Remaining available stock after reservation**
* **Error:**
  + **409 OUT\_OF\_STOCK → 可用量不足时返回 / Returned when available stock < qty**

**3. Fulfill stock（完成订单扣减库存）**

* **Endpoint → POST /orders/{order\_id}/fulfill**
* **Body:**
  + **reservations → 预留列表（reservation\_id + qty） / List of reservations (reservation\_id + qty)**
  + **operator\_id → 拣货员（shopper）ID / Operator ID (shopper)**
* **Effect:**
  + **reserved -= qty → 减少已预留量 / Decrease reserved stock**
  + **on\_hand -= qty → 扣减实际库存 / Deduct on\_hand stock**
  + **status = FULFILLED → Reservation 标记为完成 / Mark reservation as FULFILLED**
* **Response:**
  + **fulfilled → 是否成功完成 / Whether fulfillment succeeded**

**4. Clear reservation（清理预留库存）**

* **Endpoint → POST /reservations/{reservation\_id}/clear**
* **Effect:**
  + **reserved -= qty → 释放预留量 / Release reserved stock**
  + **status = RELEASED → Reservation 状态更新为已释放 / Mark reservation as RELEASED**
* **Response:**
  + **cleared → 是否成功清理 / Whether clearing succeeded**
  + **available\_after → 清理后的剩余可用量 / Available stock after clearing**
* **后台定时任务 → 扫描 expires\_at，自动清理过期的 ACTIVE Reservation，相当于自动 Clear。 / Background job scans expires\_at and clears expired ACTIVE reservations, equivalent to auto-Clear.**

**Add stock（增加库存）admin**

* **调用方 / Caller: 系统或运营人员 / System or operations staff**
* **场景 / Scenario: 仓库补货、退货入库 / Restocking or returns**

**Reserve stock（预留库存）customer**

* **调用方 / Caller: 订单服务（代表顾客） / Order Service (on behalf of customer)**
* **场景 / Scenario: 顾客下单时锁定库存 / Customer places an order to reserve stock**

**Fulfill stock（完成订单扣减库存）shopper**

* **调用方 / Caller: 拣货服务（代表 shopper） / Shopper Service (on behalf of shopper)**
* **场景 / Scenario: Shopper 拣货完成后扣减库存 / Shopper picks items and stock is deducted**

**Clear reservation（清理预留库存）admin / auto**

* **调用方 / Caller: 订单服务或后台任务 / Order Service or background job**
* **场景 / Scenario: 订单取消或支付超时释放库存 / Cancelled order or expired payment releases stock**

**6) 并发与正确性 / Concurrency & Correctness**

**ZH:**  
在并发处理上，我始终以数据库为唯一真相源。判断和更新必须放在同一个事务里完成，避免“先查后改”的竞态问题。具体做法是条件更新加乐观锁，只有当 available ≥ qty 的时候才允许增加 reserved。如果两个请求同时过来，只有一个能成功，另一个会失败。另外我还通过 Idempotency-Key 或 Reservation 的唯一约束来保证幂等性，防止重复请求造成重复扣减。总结来说，就是在同一行 (store, sku) 上做条件更新并用本地事务包裹，就能天然防止超卖。

**EN:**  
For concurrency control, the database is the single source of truth. Validation and update always happen in the same transaction to avoid race conditions like “check-then-update.” The mechanism is conditional updates with optimistic locking: we only increase reserved if available ≥ qty. If two requests come in at the same time, only one succeeds while the other fails. I also enforce idempotency using Idempotency-Keys or unique constraints on Reservations to avoid duplicate deductions. In short, conditional updates combined with local transactions on the same (store, sku) row naturally prevent overselling.

**7) 可扩展性 / Scalability**

**ZH:**  
在可扩展性方面，我会做读写分离，所有写操作都落在主库，而读请求优先走 Redis 缓存，通过短 TTL 或事件驱动来保持数据新鲜，缓存未命中再回源数据库。数据层面会做分区或分片，常见做法是按 store\_id 来拆，这样一个门店的数据集中在一个分片里，查询简单高效。如果某些门店很大或者某些 SKU 特别热门，就可以用 (store\_id, sku\_id) 的 hash 来做更细粒度的分片，把压力分散掉，避免热点。同时，我会引入事件驱动机制：一旦库存变化，就把事件发到消息队列，下游像搜索、风控、报表这些服务订阅消息后更新自己的视图，这样大家都能实时同步库存，又不会对核心数据库增加负担。最后还有降级策略，在极端情况下，比如数据库写压力太大时，可以暂时只提供读，不再接收新的预留；如果缓存失效，回源到数据库的查询要限流，这样系统可以“弯而不折”，保持核心功能可用。

**EN:**  
For scalability, I would use read/write split: all writes go to the master database, while reads are served from Redis cache with freshness maintained by short TTLs or event-driven updates; if cache misses, the request falls back to the DB. On the data layer, we use partitioning or sharding. A simple approach is by store\_id, so all data for a store stays in one shard, making queries efficient. If a store is very large or certain SKUs are extremely hot, we shard further by hash(store\_id, sku\_id) to spread load and avoid hotspots. We also make the system event-driven: whenever inventory changes, we publish an event to the message queue, and downstream systems like search, fraud detection, and reporting subscribe and update their own views. This keeps everyone in sync without hitting the core DB. Finally, we apply degradation strategies. Under heavy DB pressure, we can allow read-only availability and pause new reservations; if cache fails, we throttle DB fallbacks. This way, the system bends but doesn’t break, keeping core functionality available under stress.

**8) 安全（高层） / Security (High-level)**

* **ZH:** 鉴权：OIDC / JWT；服务间 mTLS。
* **EN:** Authentication: OIDC / JWT; mTLS for service-to-service.
* **ZH:** 授权：用户仅能操作自己的 Reservation；内部接口走 RBAC。
* **EN:** Authorization: Users can only operate their own Reservations; RBAC for internal APIs.
* **ZH:** 审计：所有库存变更落日志，记录 who/what/when/why。
* **EN:** Auditing: log all inventory changes with who/what/when/why.
* **ZH:** 敏感数据：传输/存储加密。
* **EN:** Sensitive data: encryption in transit and at rest.

**Production Deployment:**

**1. 高可用数据库 (High-availability DB)**

* **ZH:** 数据库是库存的真相源，所以必须做高可用。常见方案是 **主从复制 + 自动故障转移 (failover)**，甚至跨可用区多副本。这样一个节点宕机时，系统可以秒级切换到备用节点。
* **EN:** Since the DB is the source of truth, we must make it highly available—typically via master–replica with automatic failover, or multi-AZ replicas. This ensures the system survives node failures without losing availability.

**2. 负载均衡 (Load Balancing)**

* **ZH:** Inventory Service、API Gateway 等前端服务要有多个实例，流量通过 **负载均衡器 (LB)** 分发。这样即使某个实例挂了，请求还能自动切到健康的实例。
* **EN:** Services like Inventory or API Gateway run multiple instances behind a load balancer. If one goes down, traffic automatically shifts to healthy instances, ensuring zero-downtime.

**3. 日志与监控 (Logging & Monitoring)**

* **ZH:** 需要全链路的监控：
  + **指标监控 (Metrics)**：请求量、延迟、错误率、冲突重试率。
  + **日志 (Logs)**：每次库存变更记录 who/what/when/why。
  + **追踪 (Tracing)**：跨服务请求链路，用来排查性能瓶颈。
* **EN:** Full observability is key:
  + **Metrics** like request volume, latency, error/retry rates.
  + **Logs** for every stock change (who/what/when/why).
  + **Tracing** across services to debug bottlenecks.

**4. 备份与恢复 (Backups & Recovery)**

* ZH: 数据库必须有定期全量备份，并且要定期做 恢复演练。这样不仅能保证“有备份”，还能确认“能恢复”。在这里我们关注两个核心指标：
  + RPO (恢复点目标)：表示系统能容忍的数据丢失量，比如 RPO=5 分钟 → 最多丢最近 5 分钟的数据，所以要保证至少每 5 分钟可回放。
  + RTO (恢复时间目标)：表示系统从灾难发生到恢复可用的时间，比如 RTO=30 分钟 → 故障发生后半小时内必须恢复上线。  
    所以备份策略 + 恢复演练要确保这两个指标可以达标。
* EN: The database must take regular full backups, and we need regular recovery drills to prove not only that backups exist but also that we can restore. Two key metrics here are:
  + RPO (Recovery Point Objective): how much data we can afford to lose. For example, RPO=5 minutes means we can lose at most 5 minutes of data, so binlogs must allow replay within that window.
  + RTO (Recovery Time Objective): how quickly the system must be back online. For example, RTO=30 minutes means the system must be fully restored within half an hour.  
    Backup frequency and recovery procedures are designed to meet these targets.

**备份与恢复演练 (Backup & Recovery Drills)**

* **ZH:** 线上系统不仅要定期做数据库和关键数据的备份，还要定期做**恢复演练**。
  + 不仅验证“能备份”，更要确认“能从备份恢复”。
  + 这样一旦出现数据丢失或库损坏，能在目标 RPO/RTO 范围内恢复业务。
* **EN:** Production systems must not only take regular backups of databases and key data but also perform **recovery drills**.
  + It’s not enough to know backups exist—we must prove we can restore from them.
  + This ensures that if data loss or corruption occurs, we can recover within the agreed RPO/RTO.

**5. CI/CD 部署 (Continuous Integration & Continuous Deployment)**

* **ZH:** 部署流程需要自动化流水线：
  + **CI**：代码合并 → 自动跑单元测试、集成测试。
  + **CD**：自动部署到测试/预发/生产环境，配合蓝绿或金丝雀发布。
  + 出现问题时支持快速回滚。
* **EN:** Deployment must be automated with CI/CD:
  + **CI:** run unit/integration tests on every merge.
  + **CD:** deploy automatically to staging and production, with blue-green or canary strategies.
  + Must support fast rollback on failure.

**面试口播总结**

* **ZH:** “在生产部署时，我会确保几个关键点。首先是数据库要高可用，比如主从复制和自动故障转移；服务要跑在负载均衡器后面，多实例保证高可用。其次是全链路日志和监控，能及时发现问题。数据库要有备份并定期演练恢复，保证 RPO/RTO。最后是 CI/CD 自动化部署，结合蓝绿或金丝雀发布，出现问题可以快速回滚。”
* **EN:** “For production deployment, I’d ensure high availability on the DB with replicas and automatic failover, services behind load balancers with multiple instances, full logging and monitoring for observability, regular backups with recovery drills to meet RPO/RTO, and CI/CD automation with blue-green or canary deployments to enable safe rollouts and fast rollback.”

**6. 蓝绿发布 (Blue-Green Deployment)**

* **ZH:** 系统维护两套几乎一模一样的生产环境：**蓝环境**和**绿环境**。
  + 当前流量跑在“蓝”。
  + 新版本先部署到“绿”，测试验证没问题，再把流量切到“绿”。
  + 如果出问题，流量立刻切回“蓝”，实现快速回滚。
* **EN:** In blue-green deployment, we maintain two nearly identical environments: “blue” and “green.”
  + Traffic runs on blue.
  + We deploy the new version to green, test it, and then switch traffic over.
  + If issues arise, we can instantly roll back by switching back to blue.

**7. 金丝雀发布 (Canary Deployment)**

* **ZH:** 新版本不会一次性上线所有用户，而是先给一小部分用户（比如 1%）使用。
  + 如果指标正常，再逐步扩大比例到 10%、50%、100%。
  + 如果发现问题，只影响小部分用户，可以迅速回滚。
* **EN:** In a canary deployment, we release the new version to a small subset of users first, say 1%.
  + If metrics look good, we gradually increase to 10%, 50%, and eventually 100%.
  + If problems are detected, only a small group is affected, and we can roll back quickly.

**面试口播总结**

* **ZH:** “在上线实践上，我会用蓝绿或金丝雀发布来降低风险：蓝绿可以快速切换环境回滚，金丝雀可以先让少量用户试用再逐步放量。除此之外，还会定期做备份和恢复演练，确保不仅有备份，而且在出问题时能按 RPO/RTO 目标快速恢复。”
* **EN:** “For deployment practices, I’d use blue-green or canary deployments to reduce risk: blue-green lets us roll back instantly, while canary lets us release gradually. On top of that, I’d enforce regular backup and recovery drills, so we’re not only backing up but also verifying we can restore within our RPO/RTO targets.”

**what is RPO and RTO ?**

* **ZH:** “RPO 是能容忍丢多少数据，比如设成 5 分钟就表示最多只能丢最近 5 分钟的交易记录。RTO 是多快能恢复，比如设成 30 分钟就表示半小时内必须恢复服务。这两个指标决定了备份频率、容灾架构和演练目标。”
* **EN:** “RPO is about how much data we can afford to lose—for example, 5 minutes means at most 5 minutes of transactions are lost. RTO is about how fast we must recover—for example, 30 minutes means the system must be back online within half an hour. These two numbers drive our backup frequency, DR architecture, and recovery drills.”

**8.告警重点 / Key Alerts:**

**ZH:** available < 0（绝对不能出现）；Reservation 超时率；冲突重试率 Conflict retry rate、P99 延迟 P99 latency。

**EN:** available < 0 (must never happen); reservation expiry rate; conflict retry rate; P99 latency.

**P99 latency:** ZH: “P99 延迟就是 99% 的请求延迟都要小于某个值，只允许最慢的 1% 超过。比如 P99=200ms 就表示 99% 的请求小于 200 毫秒。它比平均值更能反映用户真实体验，尤其是尾部延迟。”

EN: “P99 latency means that 99% of requests finish faster than that threshold, and only the slowest 1% are worse. For example, P99=200ms means 99% of requests complete in under 200ms. It’s better than averages because it captures tail latency, which directly affects user experience.”

ZH: “告警上我会重点关注四个指标。第一，available 绝对不能小于 0，否则说明出现了超卖。第二，Reservation 的超时率，如果太高说明库存被无效锁定。第三，冲突重试率，如果过高说明有热点 SKU，需要调整策略。第四，P99 延迟，如果超标说明系统性能劣化，需要扩容或优化。”

EN: “For alerts, I focus on four things. First, available must never go below zero, otherwise overselling happened. Second, the reservation expiry rate—if it’s too high, stock is being locked unnecessarily. Third, the conflict retry rate—if it spikes, we may have hot SKUs and need to change strategy. And fourth, P99 latency—if it exceeds targets, performance is degrading and we need to scale or optimize.”

**Follow Up Questions**

**1) 并发控制 / Concurrency & Race Conditions**

**Q1：多人同时下单，如何防止超卖？**

**ZH:**  
为了防止超卖，我一般不会显式加锁，而是利用数据库事务和条件更新，把“检查是否有库存”和“增加预留库存”放在同一条 SQL 里完成。这样即使两个用户同时下单，也只有一个能成功，另一个会返回 409。数据库始终是唯一真相源，缓存只做读优化。如果遇到特别热点的 SKU，我会考虑用悲观锁，比如 SELECT FOR UPDATE，让并发更新串行化，进一步保证一致性。

**EN:**  
To prevent overselling, I don’t necessarily use explicit locks. Instead, I rely on atomic conditional updates inside a transaction, so checking availability and reserving stock happen in the same SQL. If two users order at the same time, only one succeeds and the other gets a 409 conflict. The database remains the single source of truth, while the cache is only used for reads. For extremely hot SKUs, I’d consider switching to pessimistic locking with SELECT FOR UPDATE to serialize concurrent updates and ensure consistency.

**Q2：乐观锁还是悲观锁？**

**ZH:**  
大多数情况下我会用乐观锁，比如通过 version 字段或条件更新来检测并发冲突。如果 version 没变，更新成功并加一；如果 version 被改过，就说明有并发冲突，需要重试。这样没有额外锁等待，吞吐量高。悲观锁则是在更新前用 SELECT FOR UPDATE 先把行锁住，其他事务必须等到我提交后才能操作。这保证了不会发生写冲突，但在高并发下可能会出现排队等待甚至死锁。一般来说，普通 SKU 我用乐观锁，热点 SKU 或多个 shopper 竞争同一 reservation 时，我会用短事务的悲观锁。如果不想阻塞，还可以配合 NOWAIT 或 SKIP LOCKED 来立即失败或跳过被锁的行。

**EN:**  
Most of the time I use optimistic locking—for example, with a version field or conditional update. If the version hasn’t changed, the update succeeds and increments the version; if it has changed, that signals a conflict and I retry. This avoids lock contention and gives higher throughput. Pessimistic locking, on the other hand, locks the row upfront with SELECT FOR UPDATE, so other transactions must wait until the current one commits. That prevents write conflicts but can lead to queuing or deadlocks under high concurrency. In practice, I’d use optimistic locking for most SKUs, and switch to short-lived pessimistic locks for hot SKUs or cases where multiple shoppers might try to consume the same reservation. To avoid blocking, I can use NOWAIT to fail fast or SKIP LOCKED to skip locked rows.

**Q3：要不要用 Redis 原子操作？ Atomic operation means the whole operation happens in one indivisible step: either fully succeeds or not at all, with no interleaving.**

* **EN:** “Redis can be used for ultra-low-latency checks with Lua scripts, but the final decision must always be in the database. Cache is for acceleration, not for correctness.”
* **ZH:** “Redis 可以用 Lua 脚本做低延迟扣减，但**最终裁决必须在数据库**。缓存只用于加速，不用于保证正确性。”

**2) 扩展性 / Scaling**

**Q1：读写都很高，怎么扩展？**

* **EN:** “I follow a CQRS approach: writes go to the Inventory DB, reads go to Redis. Updates are pushed to an event bus (Kafka) so search, recommendation, and analytics stay updated asynchronously.”
* **ZH:** “我采用 **CQRS**：写入走 Inventory DB（强一致），读取走 Redis 缓存。库存变化通过事件总线（Kafka）广播给搜索、推荐、报表等下游异步更新。”

**Q2：如何分库分表，热点怎么处理？**

* **EN:** “Partition by store\_id or hash(store\_id, sku). For very hot SKUs, we can use sharded counters or even a single-threaded queue per SKU to serialize updates.”
* **ZH:** “可以按 store\_id 或 (store\_id, sku\_id) hash 来分库分表。如果出现超级热点 SKU，可以用**分片计数**或给该 SKU 上一个**单分区串行队列**，保证顺序处理。”

**Q3：缓存一致性如何保证？**

* **EN:** “On writes, we publish an event to update or invalidate cache keys. We also use short TTLs as a safety net, and include version/timestamp to prevent stale writes overriding fresh ones.”
* **ZH:** “写操作时发布事件更新或失效缓存，缓存本身设短 TTL 兜底。同时带 version 或时间戳，防止旧数据覆盖新数据。”

**3) 安全性 / Security**

**Q1：如何防止非法库存操作？**

* **EN:** “We combine authentication, authorization, and auditing. Authentication with OIDC/JWT and mTLS; authorization with RBAC/ABAC (customers can only act on their reservations; shoppers only for their store); and auditing every stock change with who/what/when/why.”
* **ZH:** “通过**鉴权、授权、审计**三层：鉴权用 OIDC/JWT 和 mTLS；授权用 RBAC/ABAC（顾客只能操作自己的 reservation，shopper 只能操作自己的门店）；所有库存变更都记录 who/what/when/why。”

**Q2：其他安全实践？**

* **EN:** “Idempotency keys, request signing, rate limiting to prevent abuse, least-privilege access to DB/queues, and encryption for data in transit and at rest.”
* **ZH:** “使用幂等键、请求签名、防刷限流、最小权限（DB/队列凭据走密钥管理），以及传输/存储加密。”

**4) 生产部署最佳实践 / Production Deployment**

**Q1：如何保证高可用和发布安全？**

* **EN:** “Deploy with Kubernetes across multiple AZs, DB with replicas. Use blue/green or canary releases with feature flags for safe rollouts and fast rollback.”
* **ZH:** “K8s 多 AZ 部署，数据库主从；发布走**蓝绿或金丝雀**，配合特性开关，能快速回滚。”

**Q2：可观测性和 SLO？**

* **EN:** “Key SLOs: inventory query P99 < 80ms, reserve/fulfill P99 < 200ms, and available should never be negative. Monitor reservation expiry rate, retry/conflict rate, cache hit rate, and message queue lag.”
* **ZH:** “关键 SLO：查询 P99 < 80ms，预留/扣减 P99 < 200ms，**available 永不为负**。监控预留超时率、冲突重试率、缓存命中率、消息队列积压。”

**Q3：备份与容灾？**

* **EN:** “Daily full backups with binlogs for point-in-time recovery, cross-region replicas, and regular disaster recovery drills.”
* **ZH:** “每天全量备份 + binlog 实现时间点恢复，跨区域热备，定期演练容灾恢复。”

**Q4：消息投递语义？**

* **EN:** “At-least-once delivery. Consumers must be idempotent, using business keys or deduplication tables. Support replaying historical topics to rebuild downstream indexes.”
* **ZH:** “采用 **至少一次投递**；消费者要幂等（用业务键或去重表）。支持历史消息重放，用于修复下游索引或报表。”

**面试收尾总结 / Wrap-up**

* **EN:** “We centralize strong consistency in the Inventory Service with atomic conditional updates, use cache and event-driven architecture for scale, secure the APIs with auth and RBAC, and deploy with blue/green and full observability. That gives us correctness, scalability, and operational safety.”
* **ZH:** “我们把强一致集中在 Inventory Service（条件更新防超卖），通过缓存和事件驱动实现扩展性，用鉴权/RBAC 保障安全，用蓝绿发布和完备监控保证可运维性。这样系统既**正确**又**可扩展**，还能**安全上线**。”

So the system I’d design is meant to handle multiple stores, including dark stores, where each product is identified by a SKU. The core goal is to always keep track of real-time availability and prevent overselling, even under high concurrency. I treat the database as the single source of truth, while the cache is only for speeding up reads.

Conceptually, each (store, sku) has two numbers: how many items are on hand, and how many are reserved. The key invariant is that on\_hand minus reserved must always be non-negative. The state machine is simple: an item starts as Available, then can be Reserved when a customer adds it to their order; it becomes Fulfilled once the order is paid and the shopper picks it; or, if the order is canceled or times out, the reservation is Cleared and goes back to Available.

At a high level, the system is made of an API gateway in front, an Inventory Service as the authoritative writer, an Order Service that handles checkout and calls into Inventory for fulfill and clear, a Redis cache for fast reads of availability, a relational database for strong consistency, and optionally an event bus like Kafka to broadcast inventory changes to search, recommendation, or analytics. My principle is: all writes go through the Inventory Service and database; reads are served from cache, but the database always has the final say.

The data model is straightforward: an Inventory table with on\_hand, reserved, and a version field for optimistic concurrency; and a Reservation table with a UUID, user, quantity, status, and expiry time. That lets us support add stock, reserve, fulfill, and clear as simple API calls. For example, add stock increases on\_hand; reserve checks availability and increments reserved with an expiry; fulfill decreases both reserved and on\_hand; clear just releases the reserved back to available.

To prevent race conditions, I rely on conditional updates inside a single transaction—so the check and the update happen atomically in the database. That naturally prevents overselling. Every write also uses an idempotency key, so retries won’t double count. For scaling, I’d separate reads and writes: writes stay in the main database, reads go through Redis with short TTL or event-driven updates. If we see very high load, we can shard by store or hash of SKU, and use events to keep downstream systems in sync.

Finally, for production readiness I’d add authentication and RBAC to secure the APIs, audit logs for all inventory changes, blue-green or canary deployments for safe releases, backups and recovery drills, and monitoring to make sure things like “available less than zero” never happen.

**DB Schema**

So at the data model level, I keep it really simple with just two core tables: **Inventory** and **Reservation**.

The **Inventory** table has one row per store and SKU. It stores two key numbers: how many items are on hand and how many are already reserved. There’s also a version field for optimistic concurrency, and I enforce invariants at the database level—for example, both on\_hand and reserved must be non-negative, and on\_hand - reserved can never go below zero. That makes sure the system itself prevents overselling, even if there’s a bug in the application code.

Then the **Reservation** table tracks each hold when a customer adds items to their basket. It has a UUID reservation ID, the store and SKU, the user ID, quantity, a status field that goes from ACTIVE to either RELEASED, FULFILLED, or EXPIRED, and an expiry time so we can automatically clear holds that time out. Each reservation also carries an idempotency key, which ensures retries don’t create duplicates. I also add indexes to support the common queries: by store and SKU to check active reservations, by user to show their current basket, and by expiry time so a background job can efficiently release expired holds.

With just these two tables, we can implement all the key flows: add stock updates on\_hand, reserve increases reserved and creates a reservation record, fulfill consumes the reservation and deducts from on\_hand, and clear just releases the reservation back into available stock.

High level architecture + API design + DB Schema + Concurrency   
+ Scalability + Security + Deployment

Inventory audi table：

**表：inventory\_audit（库存层面的变更审计）**

* audit\_id → 主键，雪花/UUID。 / Primary key (UUID/Snowflake).
* event\_time → 事件时间（DB 侧 NOW()）。 / Event time (NOW() on DB).
* store\_id, sku\_id → 受影响的键。 / Affected keys.
* action → ‘ADD’ | ‘RESERVE’ | ‘FULFILL’ | ‘CLEAR’ | ‘EXPIRE’。 / Action enum.
* old\_on\_hand, new\_on\_hand → 变更前后 on\_hand。 / Before/after on\_hand.
* old\_reserved, new\_reserved → 变更前后 reserved。 / Before/after reserved.
* delta\_on\_hand, delta\_reserved → 差值。 / Deltas.
* operator\_id → 执行者（shopper/员工/系统）。 / Operator (shopper/staff/system).
* user\_id → 关联顾客（如适用）。 / Related customer (if any).
* order\_id, reservation\_id → 业务关联。 / Business linkage.
* reason → 文本（补货/退货/支付成功/取消等）。 / Reason text.
* request\_id / idempotency\_key → 幂等/去重追踪。 / Idempotency/correlation.
* trace\_id → 分布式追踪。 / Distributed trace id.
* source\_service → 发起服务名（inventory/order/shopper/job）。 / Source service.
* version → 触发时的行版本（便于还原时序）。 / Row version at change.
* actor\_ip / user\_agent（可选，注意隐私）。 / Optional client info (mind privacy).

设计要点：**不更新只插入**；必要字段**去标识化**，避免过多 PII。 / Insert-only; minimize PII.

**PostgreSQL DDL（示例，可直接用）：**

CREATE TABLE inventory\_audit (

audit\_id UUID PRIMARY KEY,

event\_time TIMESTAMP NOT NULL DEFAULT NOW(),

store\_id BIGINT NOT NULL,

sku\_id BIGINT NOT NULL,

action TEXT NOT NULL, -- ADD/RESERVE/FULFILL/CLEAR/EXPIRE

old\_on\_hand INT,

new\_on\_hand INT,

old\_reserved INT,

new\_reserved INT,

delta\_on\_hand INT NOT NULL DEFAULT 0,

delta\_reserved INT NOT NULL DEFAULT 0,

operator\_id TEXT,

user\_id BIGINT,

order\_id BIGINT,

reservation\_id UUID,

reason TEXT,

request\_id TEXT,

idempotency\_key TEXT,

trace\_id TEXT,

source\_service TEXT NOT NULL,

version BIGINT,

-- 便于查询的复合索引 / helpful composite indexes

INDEX idx\_audit\_store\_sku\_time (store\_id, sku\_id, event\_time),

INDEX idx\_audit\_order (order\_id),

INDEX idx\_audit\_reservation (reservation\_id),

INDEX idx\_audit\_action\_time (action, event\_time)

);