系统分析与建模第21组第二次作业

企业ERP系统设计与实施

梁沚诺 马纯华 姜苑彤 伍华明 余曼 龙俊帆

一、引言

1.1 编写目的

随着市场竞争的加剧，企业需要在信息化快速发展的大环境下驶入发展快车道，ERP系统就是帮助企业对接信息化时代的一个强有力的工具。本文档对“企业ERP系统”进行项目背景调研，分析市面上现有的ERP系统情况，并对即将开发实现的系统进行业务需求分析。

1.2 文档读者

项目经理、项目管理人员、系统分析人员、数据库设计人员、程序开发人员等。

1.3 项目背景

ERP（企业资源计划）是建立在信息技术基础上，为企业决策层及员工提供决策运行手段的管理平台，是集信息技术与先进的管理思想于一体的企业资源管理系统，已成为现代企业的运行模式，重视企业间三流（物流、信息流与资金流）合一。ERP的技术特点主要是基于网络和软件技术，采用关系型数据库、客户/服务器分布式结构（C/S结构）、浏览/服务器结构（B/S结构）等，便于维护、升级、扩充，支持多会计单位企业，具有集成工作流引擎，加强用户自定义的灵活性和可配置性能的特点。

在当今市场大环境下，企业生存和发展的环境在不断发生着变化。企业要保持良好的发展态势，就必须主动参与竞争并赢得竞争；而要赢得竞争，就要求企业全面提升管理水平。企业要立足企业化经营发展需求，推进业财融合和财务管理转型建设，优化供应链和人力资源管理流程，提高管理成熟度等相关需求，就要开展实施企业ERP系统建设工作。

目前提供ERP系统的主要企业及其优势、业务如表1-1所示。

企业	优势	主要业务
Oracle	**模块化：**可根据需要采用不同的业务模块。 **全面：**覆盖范围广，可满足不同业务的需求。 **强大：**在端到端的业务流程中提供高级分析、AI 和自动化功能。 **统一：**提供统一的数据、流程、用户体验和云服务水平。	财务、项目管理、采购、风险及合规管理、企业绩效管理（EPM）、供应链和制造、业务总分析
Infor	基于解决方案内置的行业特定功能和用户角色，只需将较少的时间用于基础设施管理。所有解决方案均通过一个具备世界级云安全的可扩展平台提供。	企业资源管理、供应链管理、人力解决方案、服务与销售、高级分析
SAP	SAP S/4HANA Cloud 是一款基于人工智能和分析技术的模块化 ERP 云软件，功能齐全，满足各种业务需求。企业可采用能充分发挥内存计算和人工智能优势的下一代ERP，替代具有局限性的传统管理技术。	资产管理、财务、制造、研发和程序设计、销售、服务、寻源和采购、供应链

金蝶	面向不同规模企业的全业务场景解决方案，提供丰富、安全、稳定的云产品及服务	财务、供应链、税务、制造、人力

表1-1 ERP企业优势与主要业务

1.4 项目意义

新常态经济发展背景之下，企业无法只依靠其内部的资源实现其市场竞争力的提升，而是需要着重考虑其供应链资源，通过合理安排供应商与生产工厂，供应商、生产工厂与客户之间关系来满足企业生产基本需求，并以此为发展的基础来不断提高自身的市场竞争实力，得以在变化多端的市场中占有一席之地。

现阶段，企业之间的市场竞争逐渐转变为多方面实力的竞争，其中就包括企业供应链实力，而ERP系统的出现与发展促进企业在市场竞争中的地位得到了有效的提高。在企业的实际运用中，ERP系统通过对企业的数据管理库进行管理，促进了企业领导动态掌握企业运营状况这一目标的实现，进而有利于站在全局角度且企业制定发展策略。

其次，ERP系统所具有的计算机特征，能够精确计算企业的各项资源，避免了人工计算产生误差而不利于企业人资、物力、财务及信息等资源的有效整合。

另外，ERP系统还具备实时监测功能，通过监测成本过程来严格控制企业生产成本的支出，并对超预算部分进行预警，为企业及时更改生产计划提供便利。

与此同时，该系统还能提供科学合理的快速采取改善措施，有效降低企业投入资金的浪费。

最后，ERP系统支持供应链管理，企业即将与其合作伙伴通过计算机与网络系统进行业务方面的联系，由于计算机和网络技术代替了传统的电话、传真等通信工具，使得各企业之间的通信效率与质量大幅度提升，极大程度上促进了企业供应链系统的完善。

二、需求描述

2.1 功能分析

根据项目背景，并结合实际情况，本ERP系统主要实现企业内部人员（销售员、采购员、财务会计、库存管理员和人事管理员）的信息管理功能。根据不同功能之间的联系与区别，我们将ERP系统抽象出客户管理、销售管理、采购管理、财务管理、人力管理和库存管理六大模块，并在模块内部划分出详细的功能。

2.2 系统模型

2.2.1 客户管理

客户管理模块能够帮助企业降低经营风险和提升客户的满意度：销售部门在制定、签约合同前能够从资金实力、信用程度等多方面了解客户信息，做出全面的评估，帮助企业防患于未然；销售主管全面掌控销售和跟进记录客户的业务流程，及时发现问题并调整，从而提升客户满意度，增加复购率。该模块的功能结构如图2-1所示。

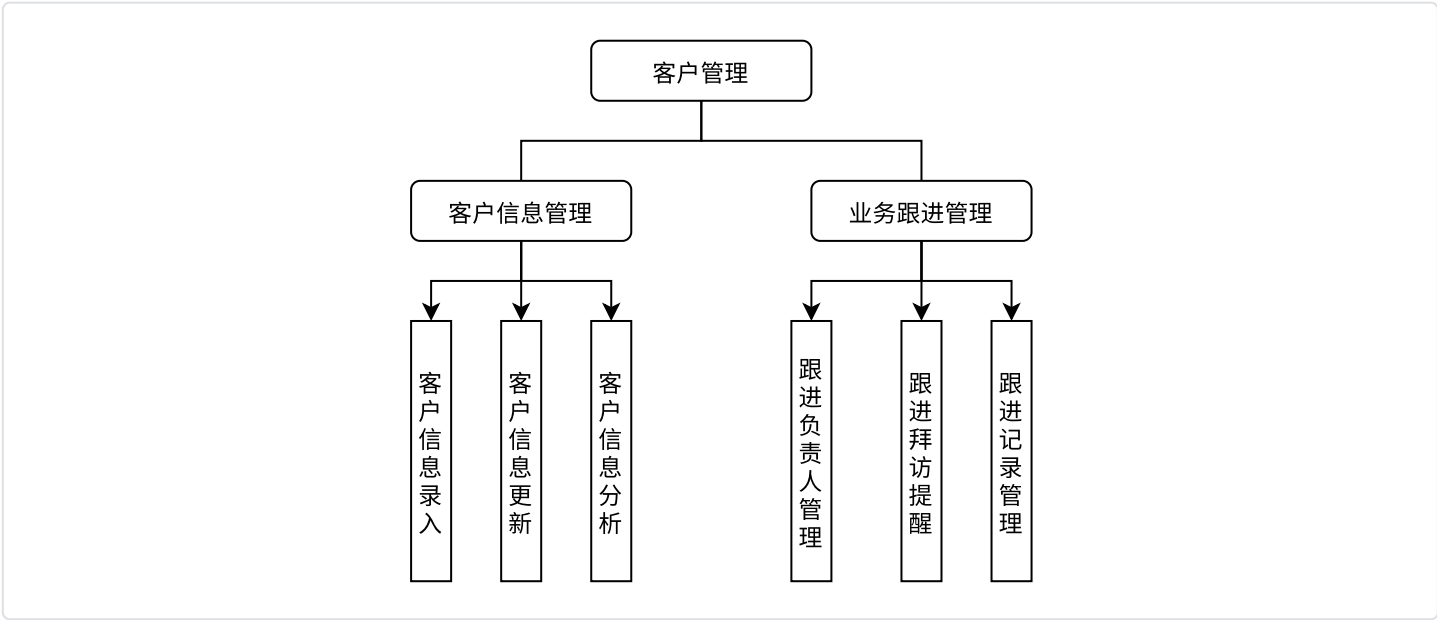

图2-1 客户管理模块

1. **客户信息管理**

客户信息管理包括对客户信息的收集、更新和分析。销售部通过不同信息渠道，确认新客户，对客户的基本信息等情况进行收集调查并录入信息系统；在后续的业务合作中，销售部根据交易数据等适时调整更新客户信息；系统根据客户等级、类别、额度，定期导出分析报表，便于企业掌握整体客户信用情况。具体情况如表2-1所示。

客户资料字段	简要说明
客户基本资料	名称，地址，联系方式，纳税人登记号等
信用记录	授权额度及信用记录，应收账款，往来资金等
订单	订单日程安排，进行状态
合伙人数据	客户的合作伙伴信息（售达方、收票方、付款方、送达方）
冻结/解冻客户	

表2-1 客户数据库的主要涵盖资料

2. **业务跟进管理**

业务跟进管理主要包括客户跟进负责人管理、跟进提醒和跟进记录管理。销售部门的工作人员能够在后台查询客户是否有人跟进，如果有人跟进，只能查看到客户基本信息和负责人，无法查看该客户的详细情况；如果没有人员跟进，可以进行申请，经销售主管审批后可正式成为负责人。这样可最大程度避免销售抢单或撞单，客户也无需担心遭到多个销售的“电话骚扰”。系统会定期提醒销售员对负责的客户进行跟进拜访，在每次跟进拜访之后销售员需要在系统上记录跟进信息。

2.2.2 销售管理

销售管理模块的功能结构如图2-2所示。

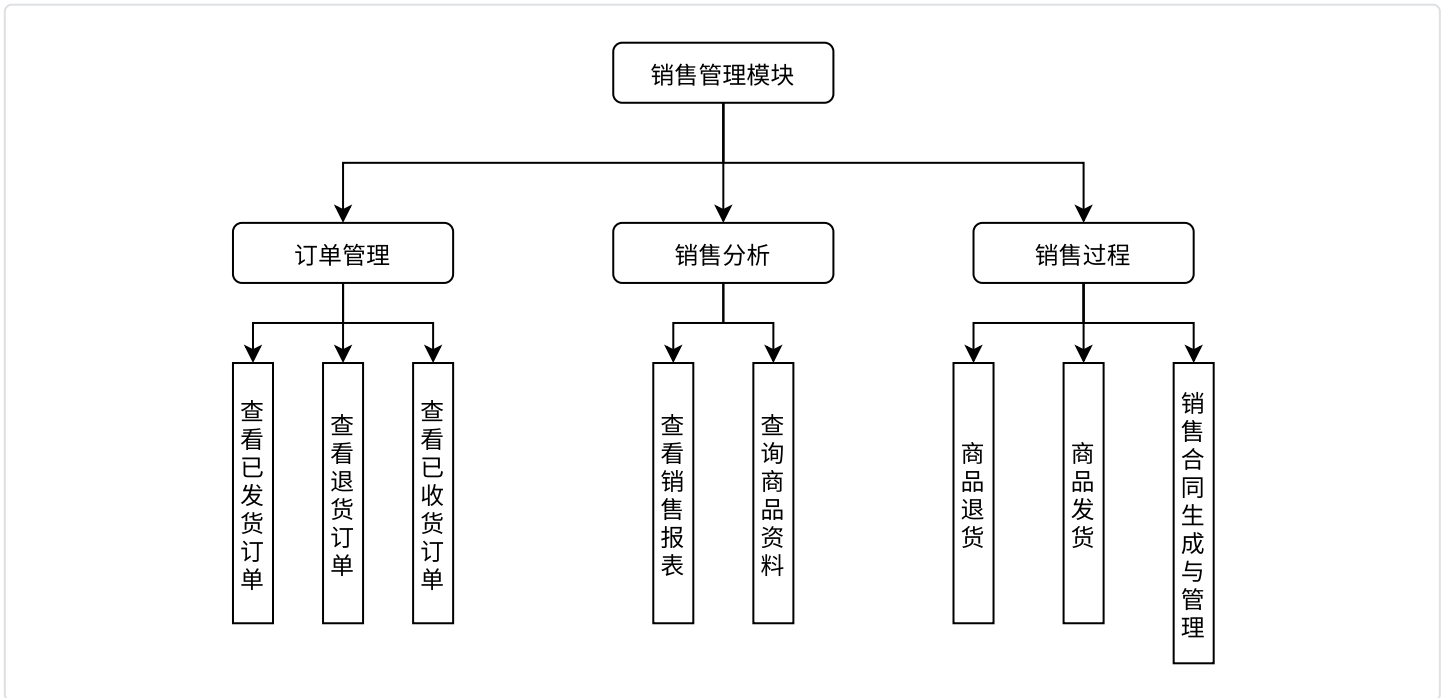

图2-2 销售管理模块

1. **订单管理**

   在订单管理功能应用中，汇总了现有以及历史的订单信息，可以查看已发货/退货/已收货的订单基本信息。

2. **销售分析**

   a. 查询商品信息：在商品签订合同前，销售员可以通过上传的商品信息（货号、款式、供应商等关键字段）实现筛选查询，快速找到需要的商品信息详情内容。

   b. 查看销售报表：在销售订单完成后，系统即根据数据，生成销售业绩报表。销售员可查看包括年度/月度（合同数、总销售额、成功交易单数、退货单数）等业务信息。

3. 销售过程

   a. 销售合同生成与管理：销售员根据销售业务填写销售合同，并根据需要修改和提交合同内容。

   b. 商品退货：退款给客户，生成退货单，修改销售订单状态，商品重新入库。

   c. 商品发货：销售员根据合同进行商品发货，设置销售订单状态，商品出库，生成发货单。

2.2.3 采购管理

采购管理模块的功能结构如图2-3所示。

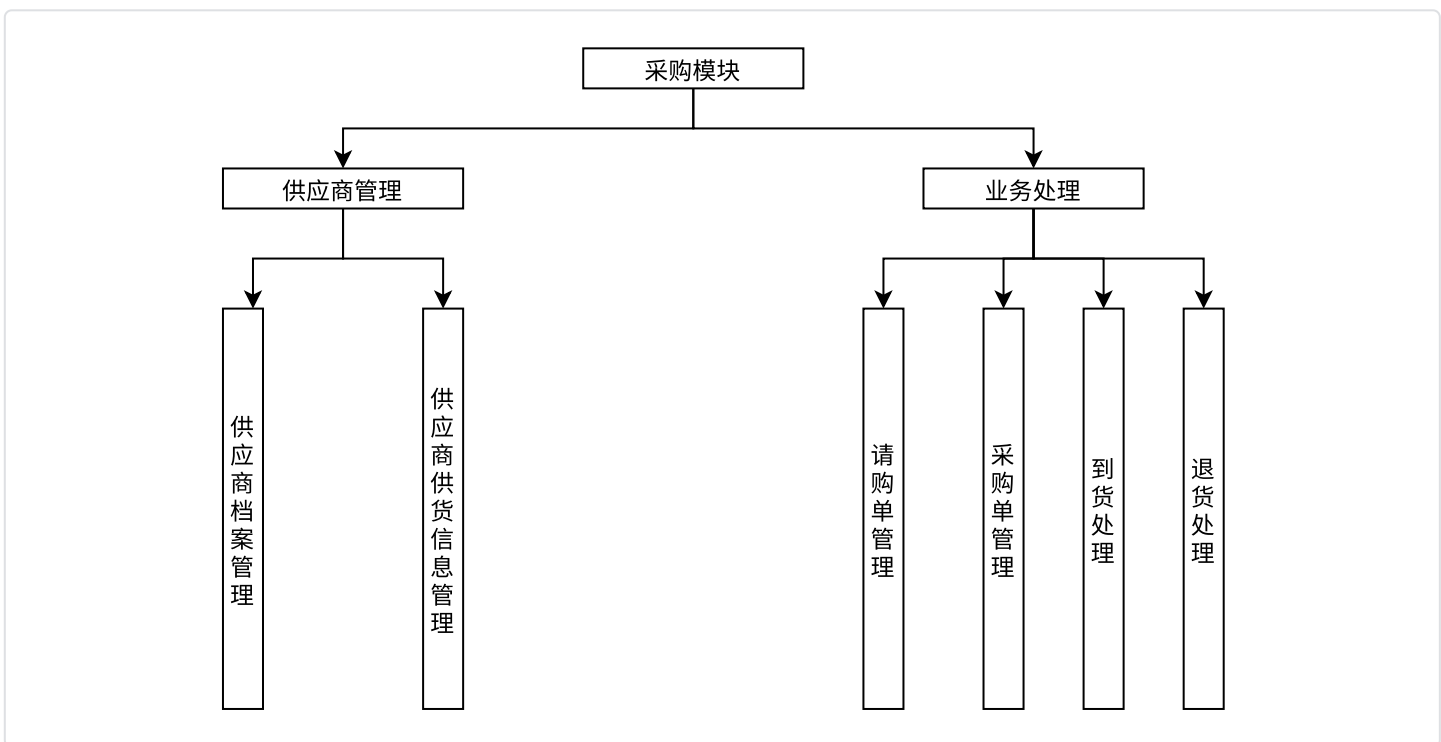

图2-3 采购管理模块

1. **供应商档案**

    供应商档案是采购管理中最重要的基础信息之一，主要记录供应商常见的一些固有 属性以及应收、 应付科目信息， 需要录入编码、 名称、 地址、 状态、 行业、 联系人、 法人代表、 开户银行、 银行账号、 营业执照、 税务登记号、 供应商分类等基本资料， 这些基本资料方便企业掌握供应商的基本信息。 供应商编码可以录入或根据编码规则生成， 但必须保证唯一性。 在采购付款时， 需要用到开户银行、 银行账号等信息， 一个供应商可以设置多个开户银行与多个银行账号。

2. **供应商供货信息**

    供应商供货信息管理主要用于设置供应商与物料之间的对应关系， 即某供应商可以供应哪些物料，本功能需要录入供应商代码、 物料代码、 最小采购批量、 配额比例等数据。 供应商代码、 物料代码分别设置为供应商及其可供应的物料； 最小采购批量设置为该物料针对基本计量单位的最小采购批量。

    供应商供货信息一方面用于处理供应商与物料的对应关系， 另一方面应用于采购管理。 在录入采购订单、 采购入库单、 采购发票等单据时， 如果设置了供应商供货信息， 在录入单据时就可以快速选择供应商及其物料， 节省输入时间。

3. **采购申请**

    采购申请单明细数据包括物料代码、 物料名称、 规格型号、 辅助属性、 计量单位、 数量、用途、供应商、 需求日期、 建议采购日期等数据， 一张采购申请单可以录入多条明细数据。

4. **采购订单处理**

    采购订单是购销双方共同签署的、 以此确认采购活动的标志文件， 相当于采购合同。 通过采购订单的管理， 可以帮助企业进行采购业务的事前预测、 事中控制与事后监督。 采购订单主要内容包括采购货物的品种、 采购数量、 到货时间、 供货商、 到货地点、 运输方式、 价格、 运费、检验方式、 付款方式等。 采购订单应根据采购合同、 采购申请、 供应商档案和存货价格文件等进行填写。 当与供应商 签订采购订单时， 可以将采购订单输入计算机， 并报采购主管确认。 生成采购订单是物资采购业务的起点， 是采购管理系统的核心业务之一， 包括订单的新增、 修改、 删除、 确认等。 经确认后的订单不能修改、 删除。 采购订单生效后可以指导供应商供货及订货方收货

5. **到货**

    采购收货处理是采购订货和采购入库的中间环节。 在采购物料到达企业后， 一般由采购业务员根据供应商的通知或送货单， 确认对方所送的货物、 数量、 价格等信息， 并将确认结果以收料通知单的形式传递到仓库作为企业收货的依据。

    采购收货处理需要填写收料通知单， 收料通知单可以手工录入， 或通过订单确认、 采购发票关联等多途径自动生成。 收料通知单是采购订单的重要执行单据， 其不仅要处理与采购订单直接关联的执行情况， 还要处理采购入库单与采购订单间接关联的执行情况， 起到承上启下的作用

6. **退货**

    当存在质量不合格、 价格不正确、 与采购订单或合同的相关条款不相符等情况时，企业可以对收货的物料进行退货处理。 退货处理需要填写退料通知单， 该单据是处理需要退回物料给供货商的业务单据， 是收料通知单的反向操作单据， 相当于红字的收料通知单。

    退料通知单可以作为 红字采购入库单的来源单据， 执行采购退货操作。 退料通知单可以通过手工录入、 收料通知单关联等方式生成。 对于企业管理来说， 退料通知单是可选单据， 企业可以根据自身业务需要决定是否选用退料通知单来管理采购退货。 如果不采 用退料通知单， 可以直接使用红字采购入库单确认出库及更新库存。 如果退货时已经付款， 还需 要对有关付款与应付单据进行处理

2.2.4 财务管理

财务管理模块的功能结构如图2-4所示。

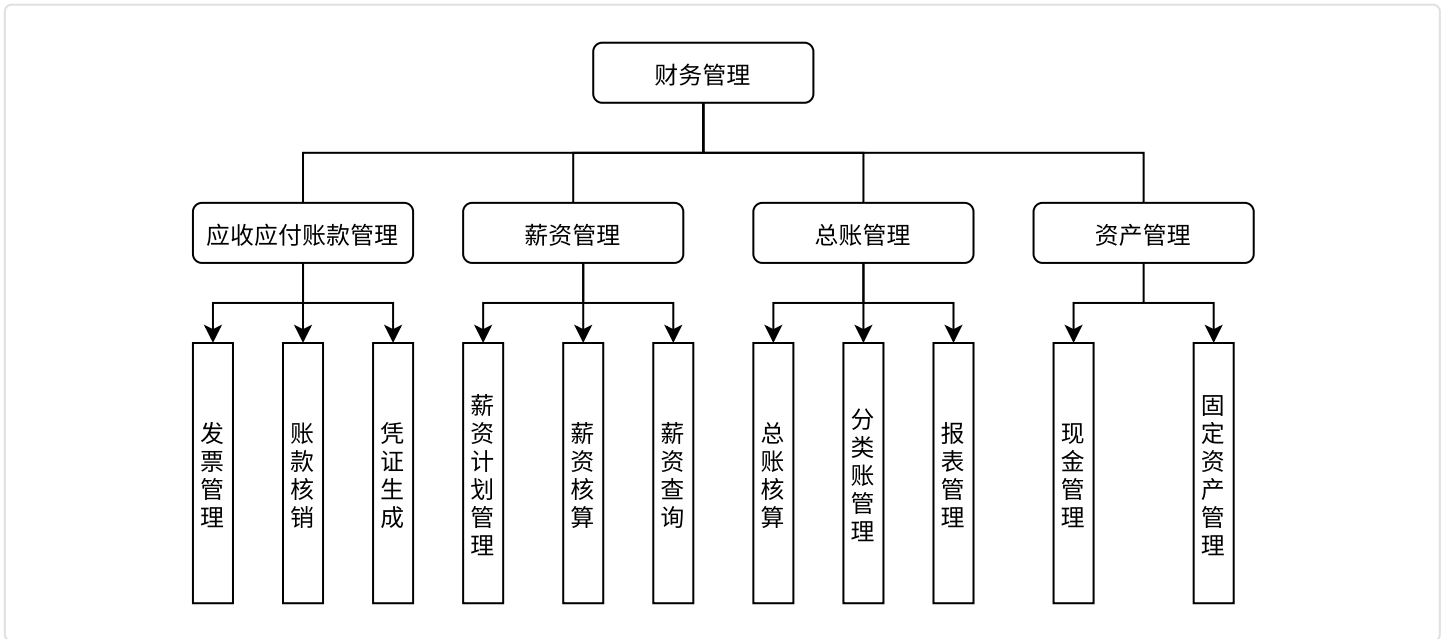

图2-4 财务管理模块

1. **应收应付账款管理**

应收应付账款管理模块对企业运营过程中的应收账款和应付账款进行管理。应收账款是企业因销售商品、物资或供应劳务等业务应向客户收取的账款。应收账管理模块的功能包括销售发票管理、应收账款核算以及收款凭证生成。应收账款管理流程如下：（1）销售管理员提交销售发票申请，（2）财务会计审核销售发票申请，批准申请后系统生成销售发票，（3）销售管理员查询销售发票，发送给相应的客户，催要应收账款（4）仓库管理员提交出库单，（5）财务会计提交收款单，（6）财务会计核销收款单与应收账款，核销通过后系统自动生成收款凭证，并传入总账系统。

应付账款是指在企业经营运作下，由于购买原材料，设备或接受服务等，应向供货单位或劳务单位所支付的款项。应付账款管理模块的功能包括采购发票管理、应付账款核算以及付款凭证生成。应付账款管理流程如下：（1）采购管理员提交采购发票申请，（2）财务会计审核采购发票申请，批准申请后系统生成采购发票，（3）采购管理员查询采购发票，进行采购，（4）仓库管理员提交入库单，（5）财务会计提交付款单，（6）财务会计核销付款单与应付账款，核销通过后系统自动生成付款凭证，并传入总账系统。

2. **总账管理**

总账对企业的各类明细账进行汇总，先根据各个会计科目的内容设置成总分类账，再将总分类账的数据整合成财务报表。总账管理模块的功能主要包括凭证管理和报表管理。各业务部门的财务数据由系统生成凭证后自动录入总账管理模块，财务会计可以查询和审核各分类账，审核完成后系统自动生成总账和财务报表。企业的各级人员被授予不同的查看权限，可以查看权限范围内的总账和报表，从而掌握实际经营情况并作出相应决策。

3. **薪资管理**

薪资管理模块分为薪资计划制定、薪资核算和薪资查询三部分。薪资核算指的是由财务会计核算员工的薪资，薪资查询则是为企业以及员工提供了解薪资情况的渠道。根据前两部分的数据，薪资计划制定子模块可以辅助企业根据实际情况和工资结构，制定工资计划、工资预算和工资标准等。

4. **资产管理**

企业资产可分为固定资产以及现金，固定资产是指企业在经营中所使用的比较持久的各项资产，现金是企业日常生产活动中的可支配流动资金。资产管理模块能自动实时生成现金流量表和利用内部银行结构对企业的资金使用情况进行量化考核，帮助企业合理分配与运用资产。

2.2.5 人事管理

人事管理模块的功能结构如图2-5所示。

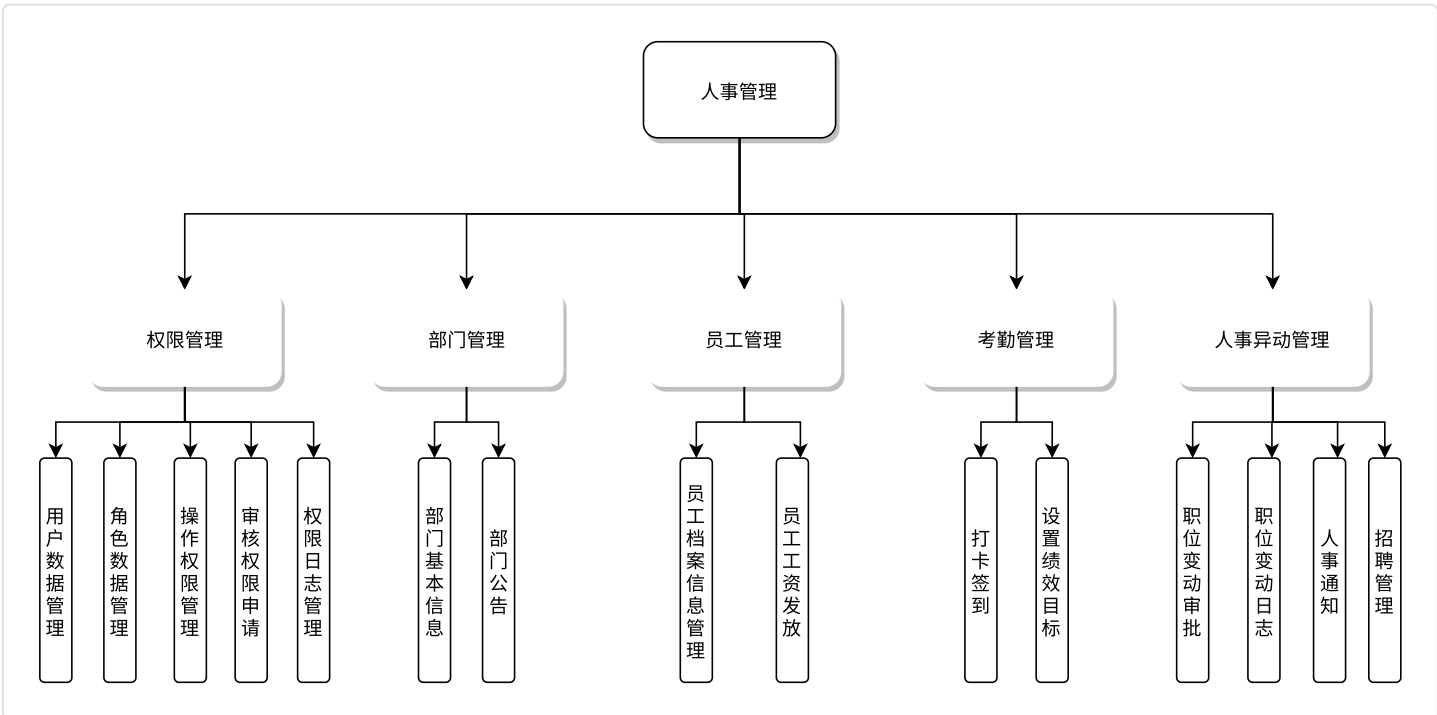

图2-5 人力管理模块

1. **权限管理**
   a. 用户数据管理：对系统使用用户的基本信息进行管理与维护。
   b. 角色数据管理：对系统中的角色以及用户的角色进行管理，可以通过添加角色组并且定义员工角色，分组管理用户的操作访问权限。
   c. 操作权限管理：操作权限管理指的是员工可以操作该系统功能的程度管理。普通员工仅可以对自身基础信息进行查看和修改，如修改密码等；而部门管理员则根据部门的区别拥有例如添加员工，删除员工，发布部门公告等操作权限。系统管理员可以单独定义用户的操作权限，也可以通过添加进角色组来管理权限。
   d. 审核权限申请：部门管理员如果没有操作某个模块的权限，可以进行申请，系统管理员可以根据其业务描述决定是否同意本次操作的申请。
   e. 权限修改日志：每次权限修改都会留下日志，权限管理员可以查看和删除权限修改日志。
2. **部门管理**
   a. 部门基本信息：本部分中，部门管理员可以对部门的基本信息进行查询和修改，该部门员工可以查询自己所属部门信息（一般无法获得增加和删除的权限，往往是最高级的权限管理员进行处理）
   b. 部门公告：部门管理员可以发布部门的公告和删除公告，部门员工登录账号则可以查看到本部门的通知和公告
3. **员工管理**
   a. 员工档案信息管理：主要存放员工的基本信息，包括合同、入职时间、签约时间等，部门管理员可以对员工相应的信息进行查看与修改，删除与增加操作需要得到授权。
   b. 员工基本权限管理：部门管理员可以为每个员工可以操作系统的内容权限进行管理，而不需要通过系统操作员。
   c. 员工工资发放：部门管理员可以根据绩效从财务系统导出员工工资信息，对员工进行工资发放。
4. **考勤管理**
   a. 打卡签到：普通员工可以每日进行打卡签到（时间段内），各部门经理可以根据签到情况，决定是否发放全勤奖
   b. 设置绩效目标：部门管理员可以根据企业实际需要设定月季绩，年考奖等（可根据员工业绩，项目评价，客户评语等）
5. **人事异动管理**

a. 职位变动审批：每一位员工的晋升、降职、调动信息，包括相应的薪资变动的情况都会经过部门管理员的审批，审批完成后才执行相应的操作

b. 职位变动日志：系统自动录入每一位员工的晋升、降职、调动信息，包括相应的薪资变动的情况

c. 查看职位变动日志：部门管理员可以查看每一位员工的晋升、降职、调动信息，包括相应的薪资变动的情况

d. 人事通知：当有人事异动出现时，部门理员可撰写部门通知告知当事人

2.2.6 库存管理

库存管理系统能够实现包括基础出/入库管理、形态转化出/入库、库存盘点功能，并且提供库存明细报表，方便商家和企业实时查看当前库存情况，该模块的功能结构如图2-6所示。

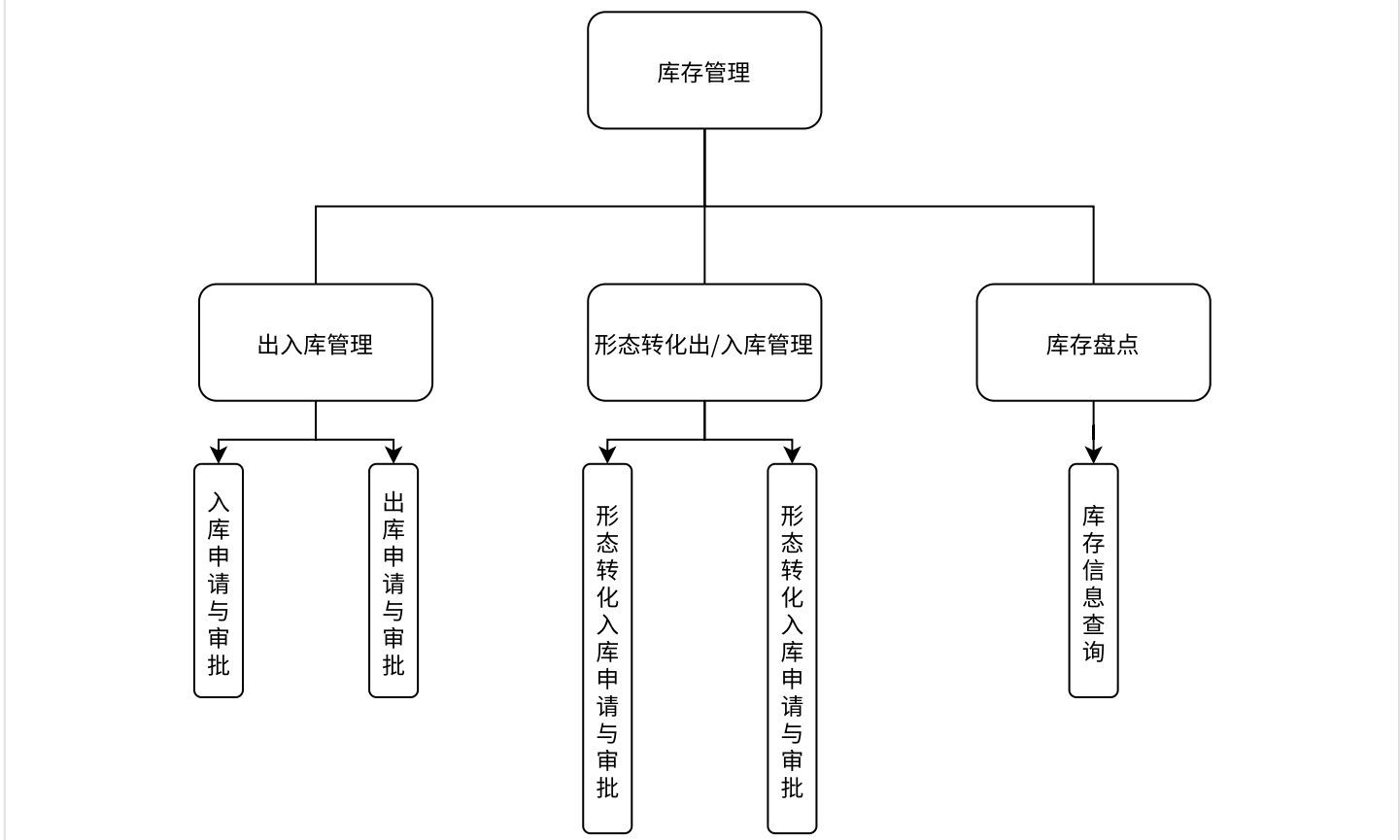

图2-5 库存管理模块

1. **出/入库管理**

   出/入库管理是指产品或物料从仓库里出去/进入仓库，包含入库申请与审批和出库申请与审批两个子模块，主要载体为出/入库信息的录入。其中出/入库信息包括出/入库单号生成、出/入库目标仓库、出/入库日期和商品明细等。审核人员根据填写的信息以及实际情况进行审核，若审核通过后，系统自动生成出/入库记录，并对库存信息进行自动更新。

2. **形态转化出/入库管理**

   形态转化出/入库是指产品或物料在存储过程中，由于环境或本身原因，其形态发生变化，由一种形态转化为另一形态，从而引起货物规格和成本的变化，在库存管理中需对此进行管理记录。与出/入库管理类似，形态转化出/入库包含形态转化入库申请与审批和形态转化出库申请与审批两个子模块，主要载体为出/入库信息的录入，经由审核人员通过后，系统自动生成出入库记录，并对库存信息进行自动更新。

3. **库存盘点**

   根据所有出/入库记录可以生成库存记录，库存盘点包含库存信息查询这一模块。系统通过在线表单详实地记录商品信息表和仓库信息表，方便商品出入库时直接调用相关信息数据。库存记录有助于对库存进行盘点。

三、业务用例

3.1 业务用例

根据需求分析，我们可以得到ERP系统的业务用例图，如图3-1所示。

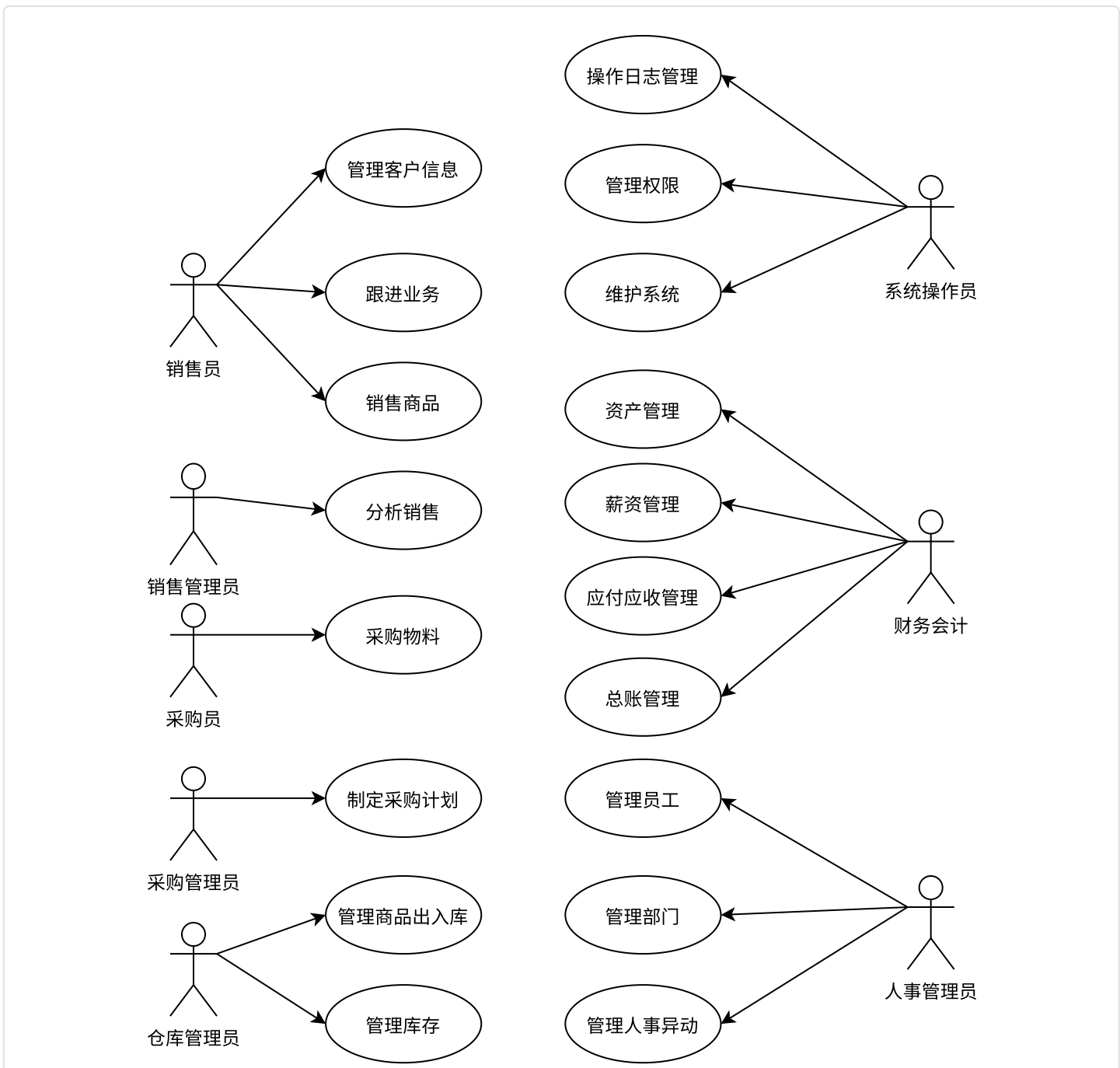

图3-1 业务用例图

作为销售员，通过客户管理模块与销售管理模块与客户取得联系，根据销售信息委托运输方进行运输；采购员通过采购管理模块与供应商取得联系，根据采购信息委托运输方进行运输；财务会计通过财务管理模块对企业的出入账、薪资、资产等财务信息进行管理；库存管理员通过库存管理模块对库存的出入库等信息进行记录与审批；部门管理员通过人力资源管理模块，总揽企业的人员权限、部门考勤、人事异动等。

3.2 业务流程

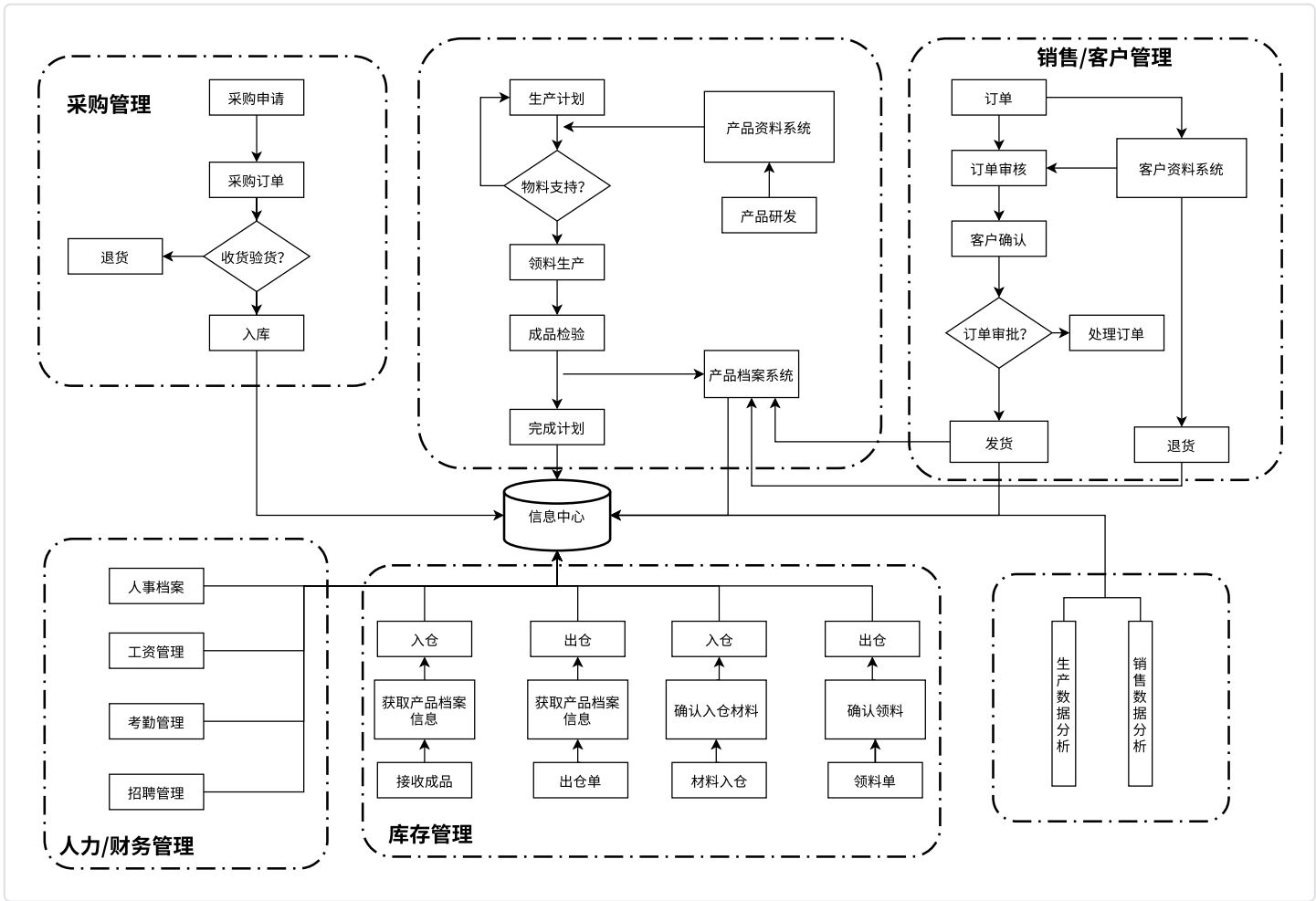

图3-2 业务流程图

根据业务用例图，我们可以细分得到业务流程图，如图3-2所示。根据系统之间的业务逻辑联系与区别，ERP系统的业务流程总体由采购管理、销售/客户管理，人力/财务管理和库存管理组成，辅以生产关系和数据分析两个子模块。

3.3 逻辑分析

ERP系统的所有业务流程由信息中心统筹，其各个模块的业务逻辑如下：

采购管理：由采购员提出采购申请并提交采购订单，如果收获验货成功则入库，否则退货。

销售/客户管理：由销售员提交订单并进行审核，并由客户进行确认。订单经审批通过后发货并录入产品档案系统，否则退货。

人力/财务管理：部门管理员可以进行人事档案管理、考勤管理和招聘管理；财务会计可以进行工资管理。

库存管理：以入库为例，由库存管理员接受成品并审核获取产品档案信息，最后将物料入仓；同理出仓单经由库存管理员获取产品档案信息后进行出仓处理。

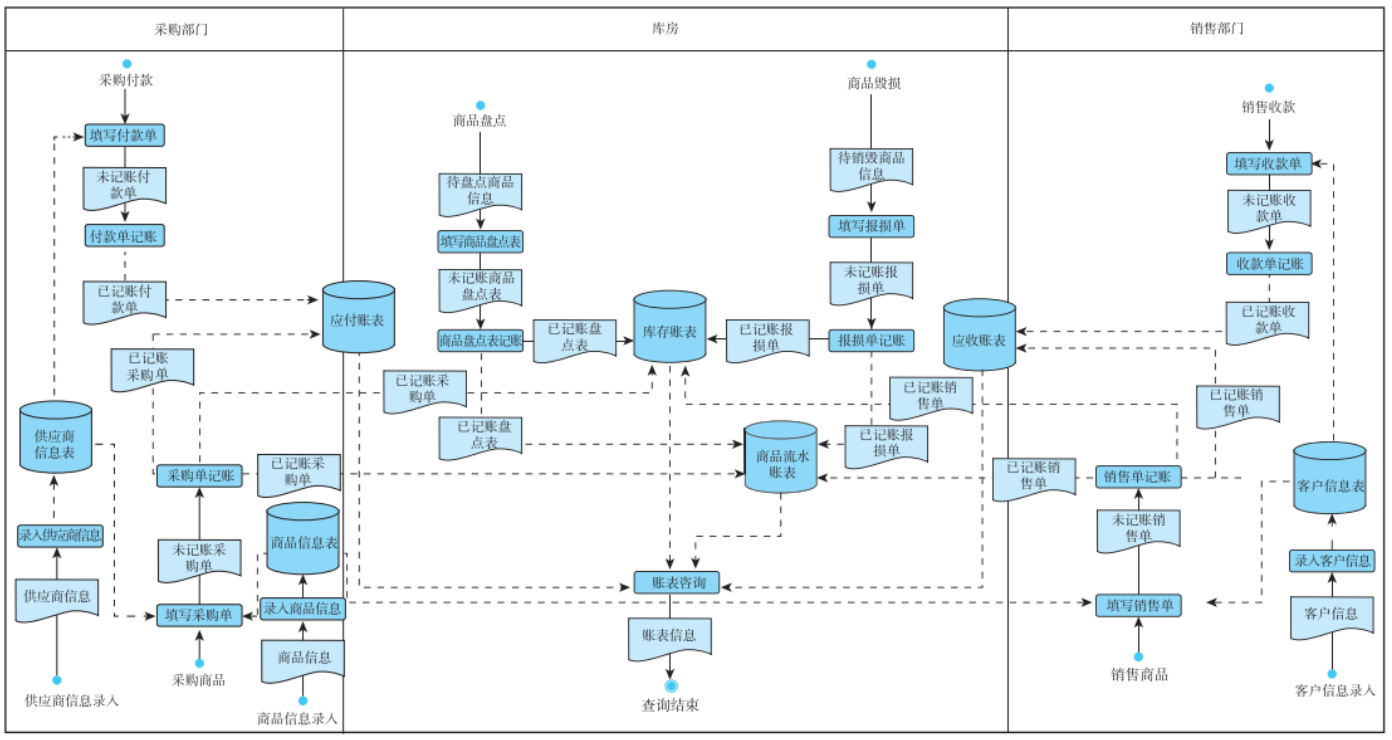

图3-3 库存管理泳道图

以库存管理为例进行逻辑分析，其泳道图如图3-3所示，

1. 采购与应付业务流程分析

采购部门根据需要填写商品采购单，并经过审核后进行记账处理，生成已记账商品采购单。 如果采购单数据存在错误，则根据是否已记账分两种情况进行处理 :在未记账情况下，可以直接 对采购单进行修改;若已记账，则通过采购单冲销模块进行处理，生成红字采购单。在对采购单 据的记账过程中，将更新商品库存表、流水账表、应付账表等账表数据。

对于付款业务，针对相应的供应商，填写付款单，并经过审核后进行记账处理。同样，如果 付款单数据存在错误，则根据是否已记账分两种情况进行处理 :在未记账情况下，直接修改付款 单信息 ;若已记账，则通过付款单冲销模块进行处理，生成红字付款单。在对付款单的记账过程 中，将更新应付账表数据。

2. 销售与应收业务流程分析

销售部门根据客户需求填写商品销售单，并经过审核后进行记账处理，生成已记账商品销售 单。如果销售单数据存在错误，则根据是否已记账分两种情况进行处理 :在未记账情况下，可以 直接对销售单进行修改 ;若已记账，则通过销售单冲销模块进行处理，生成红字销售单。在对商 品销售单据的记账过程中，将更新商品库存表、流水账表、应收账表等账表数据。

对于收款业务，针对相应的销售单据，填写收款单，并经过审核后进行记账处理。如果收款 单数据存在错误，则根据是否已记账分两种情况进行处理 :在未记账情况下，直接修改收款单信 息 ;若已记账，则通过收款单冲销模块进行处理，生成红字收款单。在对收款单的记账过程中， 将更新应收账表数据。

3. 报损业务流程分析

库房管理人员对需要报损的商品录入商品报损单，并经过审核后进行记账处理后生成已记账 报损单。如果报损单数据存在错误，则根据是否已记账分两种情况进行处理 :在未记账情况下， 直接修改报损单;若已记账，则通过报损单冲销模块进行冲销处理，生成红字报损单。在对商品 报损单据的记账过程中，将更新商品库存表、流水账表等账表数据。

4. 盘存业务流程分析

企业的存货品种多、收发频繁，在日常存货收发、保管过程中，由于计量错误、检验疏忽、 管理不善、自然损耗、核算错误以及偷窃、贪污等原因，有时会发生存货的盘盈、盘亏和毁损现 象，从而造成存货账实不相符。

库房部门根据库存结存表和实际盘存表，填写未记账库存盘点表，并经过审核后进行记账处 理，生成已记账盘点表，同时更新商品库存表、流水账表，调整库存的账面数据，从而达到账实 相符的要

求。

四、领域关键问题及模型

4.1 领域关键问题

ERP 系统不仅仅是企业资源计划系统，更是一种现代化管理思想的体现。不同企业的具体需求会有所不同，业务系统的最终运行方式也可能千差万别。但是，ERP 是一种管理思想，企业都基于这种思想实现资源的管理，不同的业务系统之间必然存在共性。

首先从整体上考虑进销存的过程：企业从供应商处采购需要的商品，企业向供应商支付费用；企业将商品存放在仓库中，随时可以查看库存量；将商品出售给客户，客户向企业支付费用。

库存盘点是清点商品库存的过程，一个简单的库存盘点流程是这样的：第一，库管员对仓库内的商品进行清点，生成库存盘点单，记录商品数量；第二，库管员查询商品库存，对比系统库存与实际库存是否一致; 第三，库管员向仓库经理汇报；第四，仓库经理核准后，库管员进行商品库存修正。

从以上简单的库存盘点业务活动流程中可以看出,库存盘点活动的概念集C=<员工、库管员、仓库经理、单据、库存盘点单，商品、类别、单位、颜色、仓库、.....>. 其中库管员和仓库经理是员工，库存盘点单是单据，表示一种继承关系is-a；库存记录某一仓库内某种商品的数量，库存与仓库、商品都是属于belong-to关系。

因此不难得出以下的领域模型图，如图4-1所示。

4.2 领域模型

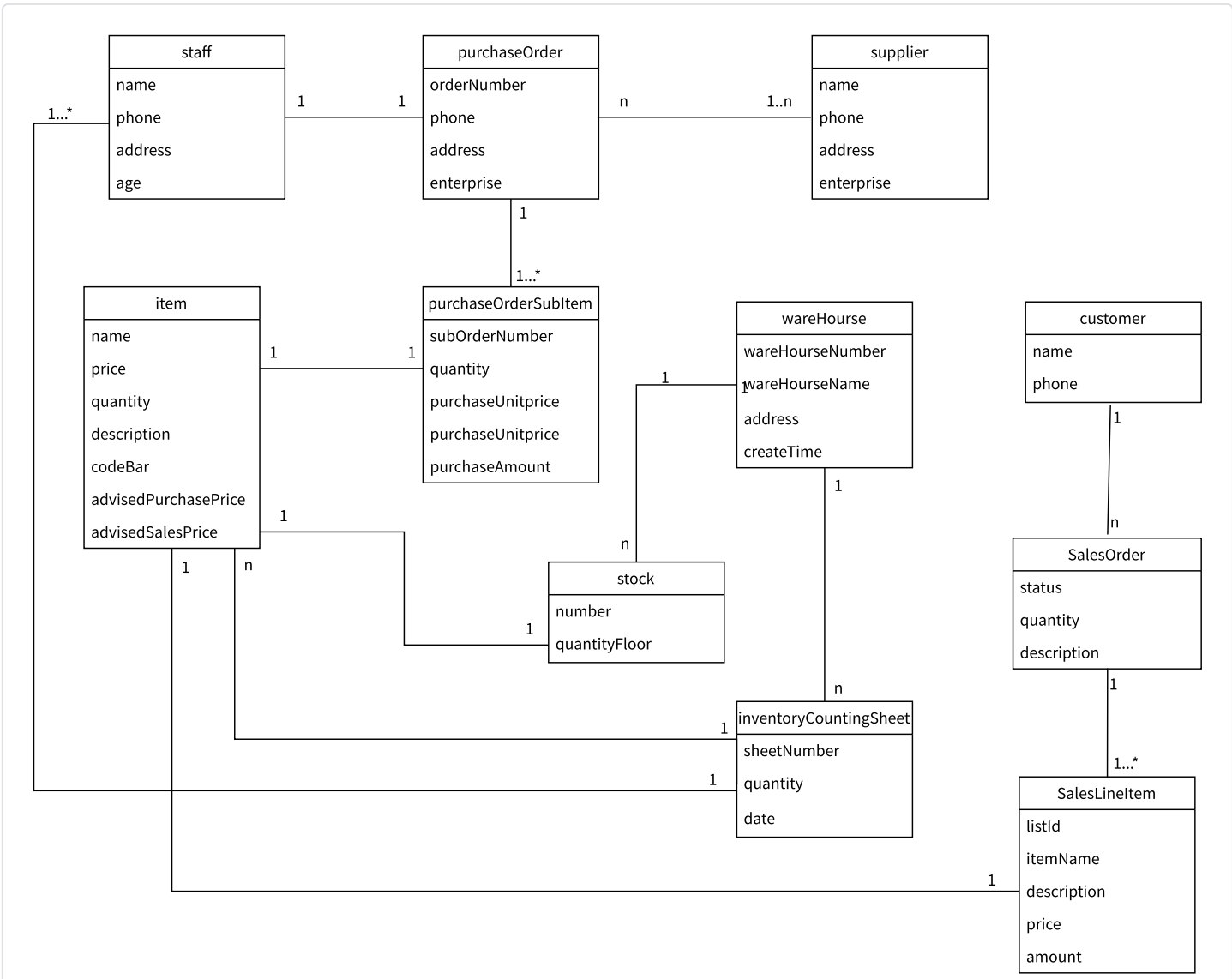

图4-1 领域模型

五、关键技术路线和业务模型

5.1 关键技术路线

根据上述分析，我们可以制定关键技术路线流程图，如图5-1所示。

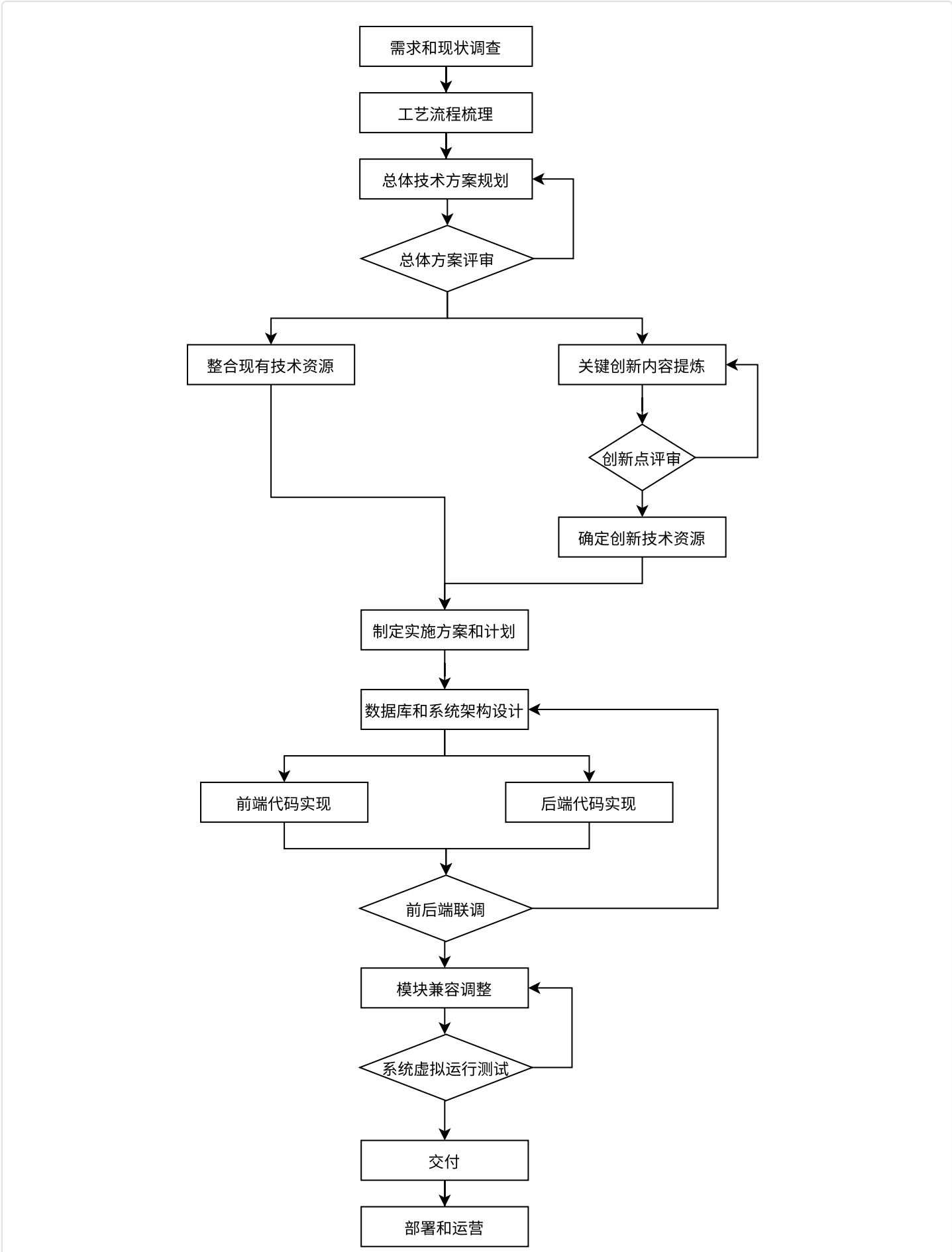

图5-1 关键技术路线流程图

首先必须进行需求和现状调查，并根据结果进行工艺流程梳理，从而提炼出总体技术方案规划。如果总体方案评审不通过，则重新规划技术方案；如果通过，则一方面整合现有技术资源，另一方面提炼关键创新内容，并进行创新点评审，评审通过后确定创新技术资源。在制定实施方案和计划后，进行数据库和系统架构设计，前后端分离进行代码实现。前后端通过API文档对接和联调，经过模块兼容调整并通过系统虚拟运行测试后，交付、部署和运营。

5.2 关键业务模型

根据上述需求分析，本系统仓库管理的主要算法业务分为两种情况，其一是直接根据货物库存数量和参考值的大小关系对货物库存情况进行判断，其二是根据某货物与其依赖货物的库存比例和参考比例的大小关系进行判断。

5.2.1 业务模型描述

1. 数量条件

针对单个货物的数量进行判断时，参考值可以由有权限的用户进行自定义（警戒值），也可以根据往年同期的库存数据自动生成，可以根据不同情况取该货物往年同季度、同月份、同周的库存均值或期间最小值。首先提取货物的 ID 值和其本期库存数据，根据 ID 值和日期查询参考值，比较本期库存数据和参考值，若本期库存较低，则警示“库存可能不足”，反之警示“库存可能溢出”。该算法模型如图5-4所示。

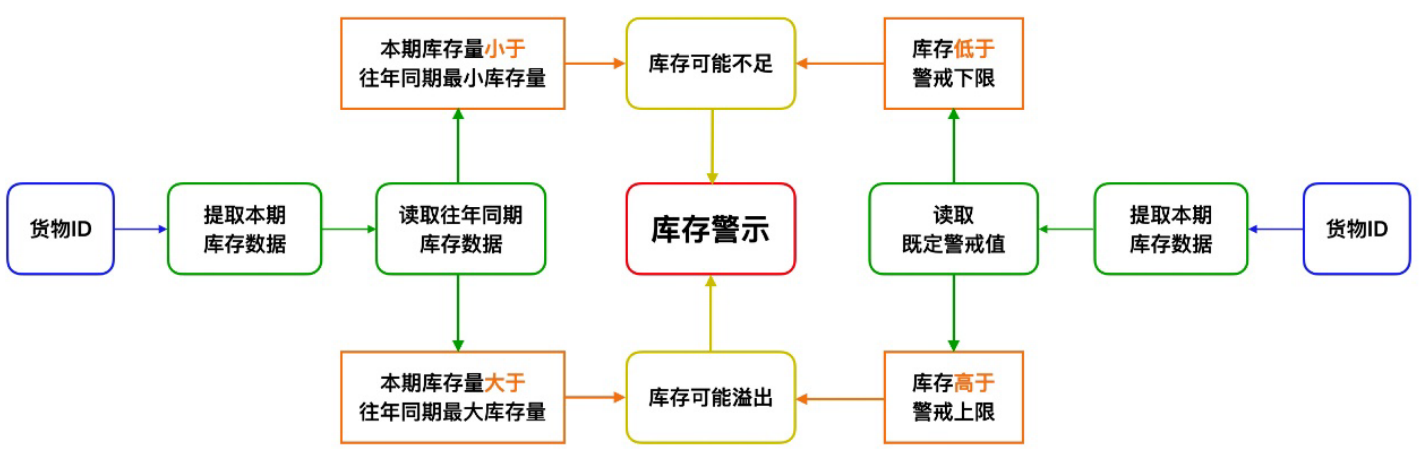

图5-4 数量条件算法模型

2. 比例条件

针对有关联的多个货物的比例关系进行判断时，参考比例可以由有权限的用户进行自定义（警戒库存比例），也可以根据往年同期的库存数据自动生成，可以根据不同情况取目标货物及其关联货物往年同季度、同月份、同周的库存均值的比例。该算法模型如图5-5所示。

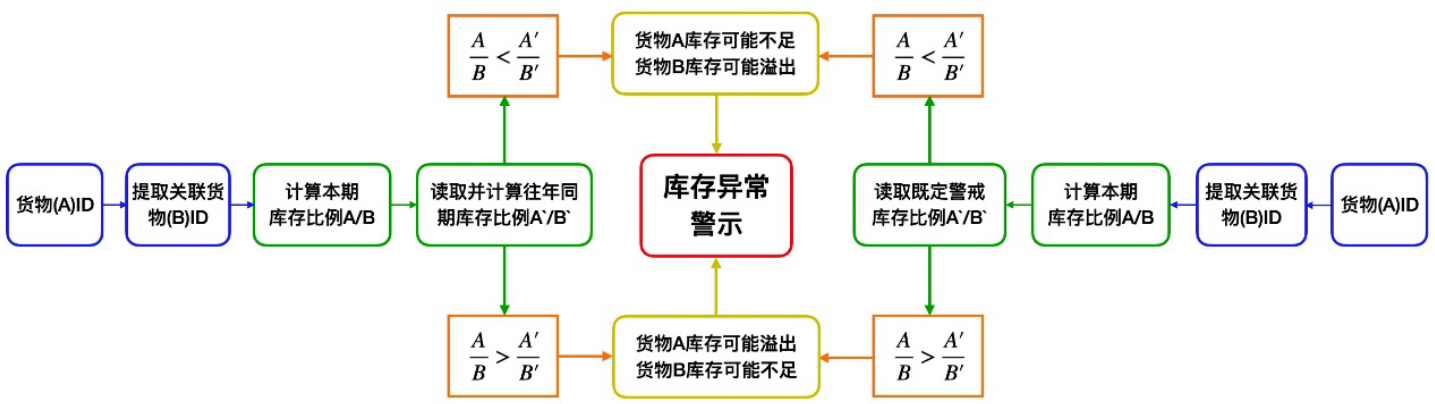

图5-5 比例条件算法模型

首先提取目标货物的 ID 值和其本期库存数据，根据 ID 值查询其关联货物的ID，根据关联货物的 ID 和日期查询计算参考比例，比较本期库存数据比例和参考比例，若本期库存比例较低，则警示“目标货物库存可能不足/关联货物库存可能溢出”，反之警示“目标货物库存可能溢出/关联货物库存可能不足”。

3. 存货计价

存货计价指企业在经营期末按照财务会计制度规定对期末存货价格的计算。一个企业在生产、经营过程中，要不断地购进和耗用物资，到期又都必然有未经售出或耗用的商品、材料等存货，一般包括有原材料、在产品、半成品、产成品或副产品等。这些存货价格的计算正确与否，与企业损益计算有着十分密切的联系。在出货时，需要给出存货的价格，需要根据这个价格来计算相关的利润以及存

货的估价。仓库中对应的商品可能是不同批次采购进来的，其单价是不一的，所以出货时的价格会受计算策略影响。

先进先出法：

假定先购入的材料先发出，并根据这种假设的成本流转次序对发出的材料进行计价。

移动平均法：

在每次存货收货后，立即根据存库库存总数量和总成本，计算出新的平均单价成本。移动平均法的计算公式如下

$$移动平均单价 = \frac{当前存结金额 + 本次收入金额}{当前存结数量 + 本次存结数量}$$

全月平均法：

全月平均法一般在期末按每种存货的结存金额与结存数量，计算出存货的全月平均单价。全月平均移动法的计算公式如下：

$$全月平均单价 = \frac{期初结存金额 + 本月收入金额}{期初结存数量 + 本月收入数量}$$

个别计价法：

个别计价法对发出存货成本进行个别计价，适用于对成本较敏感的企业。适用于数量少，但是价格特别昂贵的货物。

5.2.2 算法模型实现

1. 伪代码

   a. 数量条件

```
start
    库存 := 获取货物库存(id)
    if (库存 == null) then
        return 错误
    endif

    if (存在既定数量阈值) then
        阈值 := 获取(既定数量阈值)
        if (库存阈值 > 阈值上限) then
            return 库存可能溢出
        else if (库存阈值 < 阈值下限) then
            return 库存可能不足
        else
            return 库存阈值正常
        endif
    else then
        阈值 := 获取(往年既定数量阈值)
        if (库存阈值 > 阈值上限) then
            return 库存可能溢出
        else if (库存阈值 < 阈值下限) then
            return 库存可能不足
        else
            return 库存阈值正常
        endif
    endif
end
```

   b. 比例条件

```
start
    库存 := 获取货物库存(id)
    if (库存 == null) then
        return 错误
    endif

    关联库存 := 获取获取库存(related_id)
    if (关联库存 == null) then
        return 错误
    endif

    库存阈值 := 库存 / 关联库存
    if (关联库存存在既定比例阈值) then
        阈值 := 获取(既定比例阈值)
        if (库存阈值 > 阈值上限) then
            return 库存可能溢出
        else if (库存阈值 < 阈值下限) then
            return 库存可能不足
        else
            return 库存阈值正常
        endif
    else then
        阈值 := 获取(往年既定比例阈值)
        if (库存阈值 > 阈值上限) then
            return 库存可能溢出
        else if (库存阈值 < 阈值下限) then
            return 库存可能不足
        else
            return 库存阈值正常
        endif
    endif
end
```

c. 入库价格处理

```
start
    switch stategy do
        case 先进先出法
            货物价格列表:=仓库[入库货物编号]
            新货物价格列表:=货品价格列表.append({入库数量，入库单价})
            仓库[入库货物编号]:=新货物价格列表
            break;
        case 移动平均法
            旧货物价格项:=仓库[入库货物编号]
            旧货物价格:=旧货物价格项.价格
            旧货物数量:=旧货物价格项.数量
            新货物价格:=（旧货物价格*旧货物数量 + 入库货物价格*入库货物数量）
                /（旧货物数量+入库货物数量）
            新货物数量:=旧货物数量+入库货物数量
            仓库[入库货物编号]:={新货物价格，新货物数量}
            break;
        case  全月平均法
            本月该货物项 := 仓库[入库货物编号].本月
            货物金额 := 本月该货物项.金额 + 入库货物数量*入库货物价格
            货物数量 := 本月该货物项.数量 + 入库货物数量
            本月该货物项.金额 = 货物金额
            本月该货物项.数量 = 货物数量
```

```
            仓库[入库货物编号].本月:=本月该货物项
            // 这里只是将数量和金额计算到本月的金额中，等到月末的时候通过调度算法，执行
            // 本月的数据计算入期末金额中，而每次出货用的都是期末金额 与本月的数据无关
            break;
        case 个别计价法:
            仓库[入库货物编号].数量={入库货物数量，入库货物价格}
            // 这里是将每个货物都当成一种独立的商品
            break;
end
```

d. 出库价格

```
start
    switch stategy do
        case 先进先出法
            货物价格列表:=仓库[出库货物编号]
            while (准备出库数量 < 需要出库数量) {
                最早入库的货物项 = 货物价格列表.poll()
                if (最新入库的货物项.数量 + 准备出库数量 > 需要出库数量) {
                    // 这批货物已经能满足出库的需求了
                    重新入队货物项.数量 = 最新入库的货物项.数量 -
                        (需要出库数量 - 准备出库数量)
                    重新入队货物项.价格 = 最新入库的货物项.价格
                    货物价格表.从队头插入(重新入队货物项)

                    出库金额 += （需要出库数量 - 准备出库数量）*最新入库的货物项.价格
                    准备出库数量 = 需要出库数量
                } else {
                    出库金额 += 最新入库的货物项.价格 * 最新入库的货物项.数量
                    准备出库数量 += 最新入库的货物项.数量
                }
            }
            仓库[入库货物编号]:=货物价格列表
            出库价格 := 出库金额/准备出库数量
            break;
        case 移动平均法
            货物价格项:=仓库[入库货物编号]

            货物数量:=货物价格项.数量 - 需要出库数量

            仓库[入库货物编号]:={货物数量，货物价格项.价格}
            出库价格 := 货物价格项.价格
            break;
        case  全月平均法
            货物项 := 仓库[入库货物编号].期末结存

            货物结存数量:=货物项.数量 - 需要出库数量

            仓库[入库货物编号].期末结存:={货物结存数量，货物价格项.价格}
            出库价格 := 货物项.价格
            break;
        case 个别计价法:
            货物项 := 仓库[入库货物编号]
            货物数量:=货物项.数量 - 需要出库数量

            仓库[入库货物编号]:={货物数量，货物项.价格}
            // 这里是将每个货物都当成一种独立的商品
```

```
            出库价格 := 货物项.价格
            break;
end
```

2. Go语言实现

   a. 数量条件和比例条件

```go
import "fmt"

// Unit 货物单位的数据类型
type Unit interface {
   int | uint | float64 | float32
}

// Threshold 阈值
type Threshold[T Unit] struct {
   Min T
   Max T
}

// Product 货物的数据结构
type Product struct {
   ID                   uint
   Count                uint
   CurrentThreshold     *Threshold[uint]
   HistoryThreshold     map[uint]*Threshold[uint]
   CurrentThresholdRatio *Threshold[float64]
   HistoryThresholdRatio map[uint]*Threshold[float64]
}

// Storage 库存的数据结构
type Storage struct {
   Products []*Product
}

func (s *Storage) getProduct(id uint) *Product {
   for _, product := range s.Products {
      if product.ID == id {
         return product
      }
   }
   return nil
}

// CheckThreshold 数量条件
func (s *Storage) CheckThreshold(id uint, month uint) (string, error) {
   product := s.getProduct(id)
   if product == nil {
      return "error", fmt.Errorf("仓库中查无id为%d的货物", id)
   }

   if product.CurrentThreshold != nil {
      if product.Count > product.CurrentThreshold.Max {
         return "库存可能溢出! ", nil
      }
      if product.Count < product.CurrentThreshold.Min {
         return "库存可能不足! ", nil
```

```
51          }
52          return "", nil
53      }
54
55      if product.HistoryThreshold == nil || len(product.HistoryThreshold) == 0 {
56          return "error", fmt.Errorf("仓库中id为%d的货物往年同期库存数据为空", id)
57      }
58      historyThreshold, ok := product.HistoryThreshold[month]
59      if !ok {
60          return "error", fmt.Errorf("仓库中id为%d的货物往年%d月库存数据为空", id, mo
61      }
62
63      if product.Count > historyThreshold.Max {
64          return "库存可能溢出！", nil
65      }
66      if product.Count < historyThreshold.Min {
67          return "库存可能不足！", nil
68      }
69
70      return "", nil
71  }
72
73  // CheckThresholdRatio 比例条件
74  func (s *Storage) CheckThresholdRatio(id, relatedID uint, month uint) (string
75      product := s.getProduct(id)
76      if product == nil {
77          return "error", fmt.Errorf("仓库中查无id为%d的货物", id)
78      }
79
80      relatedProduct := s.getProduct(relatedID)
81      if relatedProduct == nil {
82          return "error", fmt.Errorf("仓库中查无id为%d的关联货物", id)
83      }
84
85      if relatedProduct.Count == 0 {
86          return fmt.Sprintf("id为%d的关联货物库存为0，已不足！", relatedID), nil
87      }
88
89      currentRatio := float64(product.Count) / float64(relatedProduct.Count)
90      if relatedProduct.CurrentThresholdRatio != nil {
91          if currentRatio > product.CurrentThresholdRatio.Max {
92              return "库存可能溢出！", nil
93          }
94          if currentRatio < product.CurrentThresholdRatio.Min {
95              return "库存可能不足！", nil
96          }
97          return "", nil
98      }
99
100     if relatedProduct.HistoryThresholdRatio == nil || len(relatedProduct.Histo
101         return "error", fmt.Errorf("仓库中id为%d的关联货物往年同期库存比例数据为空",
102     }
103     historyThresholdRatio, ok := product.HistoryThresholdRatio[month]
104     if !ok {
105         return "error", fmt.Errorf("仓库中id为%d的关联货物往年%d月库存比例数据为空",
106     }
107
108     if currentRatio > historyThresholdRatio.Max {
109         return "库存可能溢出！", nil
```

```
    }
    if currentRatio < historyThresholdRatio.Min {
        return "库存可能不足! ", nil
    }

    return "", nil
}
```

b. 数量条件和比例条件测试代码

```
package main

import (
    "fmt"
    "testing"
)

func TestStorage_CheckThreshold(t *testing.T) {
    storage := new(Storage)
    storage.Init()

    msg, err := storage.CheckThreshold(2, 10)
    fmt.Println("***************")
    fmt.Println(msg)
    fmt.Println(err)
}

func TestStorage_CheckThresholdRatio(t *testing.T) {
    storage := new(Storage)
    storage.Init()

    msg, err := storage.CheckThresholdRatio(1, 2, 10)
    fmt.Println("***************")
    fmt.Println(msg)
    fmt.Println(err)
}
```

c. 存货价格算法

```
package main

import (
    "sort"
    "time"
)

type ValuationStrategy interface {
    init()
    InWarehouse(productId uint, productNumber uint, productPrice float64)
    ExWarehouse(productId uint, productNumber uint) (productPrice float64, err
}

// FIFOStrategy 先进先出
type FIFOStrategy struct {
    warehouse map[uint]ITEMS
```

```
}

// MovingAveragePrice 平均移价法
type MovingAveragePrice struct {
   warehouse map[uint]*ITEM
}

// MovingPriceEndOfMonth 月末移价法
type MovingPriceEndOfMonth struct {
   warehouseCurMonth map[uint]*ITEM
   warehouseSum      map[uint]*ITEM
}

// SpecificPrice 个别移价法
type SpecificPrice struct {
   warehouse map[uint]*ITEM
}

type ITEM struct {
   Priority int64
   Number   uint
   Price    float64
}

type ITEMS []*ITEM

func (e ITEMS) Len() int {
   return len(e)
}
func (e ITEMS) Swap(i, j int) {
   e[i], e[j] = e[j], e[i]
}
func (e ITEMS) Less(i, j int) bool {
   return e[i].Priority > e[j].Priority
}
func (f *FIFOStrategy) init() {
   m := make(map[uint]ITEMS)
   f.warehouse = m
}

func (f *FIFOStrategy) InWarehouse(productId uint, productNumber uint, produc
   var productList ITEMS
   productList = f.warehouse[productId]
   if productList == nil {
      productList = ITEMS{}
   }

   productList = append(productList, &ITEM{
      Priority: time.Now().Unix(),
      Number:   productNumber,
      Price:    productPrice,
   })
   f.warehouse[productId] = productList
}

func (f *FIFOStrategy) ExWarehouse(productId uint, productNumber uint) (produ
   productList := f.warehouse[productId]
   sort.Sort(productList)
   newList := ITEMS{}
```

```
76     flag := false
77     curNumber := uint(0)
78     exWarehousePriceSum := float64(0)
79     for _, item := range productList {
80        if flag {
81           newList = append(newList, &ITEM{
82              Priority: item.Priority,
83              Number:   item.Number,
84              Price:    item.Price,
85           })
86        } else {
87           if curNumber+item.Number >= productNumber {
88              newList = append(newList, &ITEM{
89                 Priority: item.Priority,
90                 Number:   item.Number + curNumber - productNumber,
91                 Price:    item.Price,
92              })
93              flag = true
94              exWarehousePriceSum += item.Price * float64(productNumber-curNumb
95              curNumber = productNumber
96           } else {
97              exWarehousePriceSum += item.Price * float64(item.Number)
98              curNumber += item.Number
99           }
100       }
101    }
102    return exWarehousePriceSum / float64(productNumber), err
103 }
104 func (f *MovingAveragePrice) init() {
105    m := make(map[uint]*ITEM)
106    f.warehouse = m
107 }
108
109 func (f *MovingAveragePrice) InWarehouse(productId uint, productNumber uint,
110    product := f.warehouse[productId]
111    if product == nil {
112       product = &ITEM{}
113    }
114
115    price := (product.Price*float64(product.Number) + float64(productNumber)*p
116    number := product.Number + productNumber
117    newProduct := &ITEM{
118       Priority: 0,
119       Number:   number,
120       Price:    price,
121    }
122    f.warehouse[productId] = newProduct
123 }
124
125 func (f *MovingAveragePrice) ExWarehouse(productId uint, productNumber uint)
126    product := f.warehouse[productId]
127    exWarehousePrice := product.Price
128    product.Number -= productNumber
129    f.warehouse[productId] = product
130    return exWarehousePrice, err
131 }
132
133 func (f *MovingPriceEndOfMonth) init() {
134    m1 := make(map[uint]*ITEM)
```

```
135     m2 := make(map[uint]*ITEM)
136     f.warehouseCurMonth = m1
137     f.warehouseSum = m2
138 }
139
140 func (f *MovingPriceEndOfMonth) InWarehouse(productId uint, productNumber uin
141     product := f.warehouseCurMonth[productId]
142     if product == nil {
143         product = &ITEM{}
144     }
145
146     price := product.Price + float64(productNumber)*productPrice
147     number := product.Number + productNumber
148     newProduct := &ITEM{
149         Priority: 0,
150         Number:   number,
151         Price:    price,
152     }
153     f.warehouseCurMonth[productId] = newProduct
154 }
155
156 func (f *MovingPriceEndOfMonth) ExWarehouse(productId uint, productNumber uin
157     product := f.warehouseSum[productId]
158     exWarehousePrice := product.Price
159     product.Number -= productNumber
160     f.warehouseSum[productId] = product
161     return exWarehousePrice, err
162 }
163
164 func (f *MovingPriceEndOfMonth) Settlement(productId uint) {
165     productSum := f.warehouseSum[productId]
166     if productSum == nil {
167         productSum = &ITEM{}
168     }
169     productCur := f.warehouseCurMonth[productId]
170     if productCur == nil {
171         productCur = &ITEM{}
172     }
173     price := (productSum.Price*float64(productSum.Number) + productCur.Price)
174     number := productCur.Number + productSum.Number
175
176     f.warehouseSum[productId] = &ITEM{
177         Priority: 0,
178         Number:   number,
179         Price:    price,
180     }
181     f.warehouseCurMonth[productId] = nil
182     return
183 }
184
185 func (f *SpecificPrice) init() {
186     m := make(map[uint]*ITEM)
187     f.warehouse = m
188 }
189
190 func (f *SpecificPrice) InWarehouse(productId uint, productNumber uint, produ
191
192     price := productPrice
193     number := productNumber
```

```
194     newProduct := &ITEM{
195        Priority: 0,
196        Number:   number,
197        Price:    price,
198     }
199     f.warehouse[productId] = newProduct
200 }
201
202 func (f *SpecificPrice) ExWarehouse(productId uint, productNumber uint) (prod
203     product := f.warehouse[productId]
204     exWarehousePrice := product.Price
205     product.Number -= productNumber
206     f.warehouse[productId] = product
207     return exWarehousePrice, err
208 }
```

d. 存货价格算法测试代码

```
 1 package main
 2
 3 import (
 4     "fmt"
 5     "testing"
 6 )
 7
 8 func TestFIFOStrategy_ExWarehouse(t *testing.T) {
 9     f := &FIFOStrategy{}
10
11     var v ValuationStrategy
12     v = f
13     v.init()
14     v.InWarehouse(1, 1, 1)
15     v.InWarehouse(1, 2, 2)
16     price, err := v.ExWarehouse(1, 2)
17     fmt.Println("***************")
18     fmt.Println(price)
19     fmt.Println(err)
20 }
21
22 func TestMovingAveragePrice_ExWarehouse(t *testing.T) {
23     f := &MovingAveragePrice{}
24     var v ValuationStrategy
25     v = f
26     v.init()
27     v.InWarehouse(1, 1, 1)
28     v.InWarehouse(1, 2, 2)
29     price, err := v.ExWarehouse(1, 2)
30     fmt.Println("***************")
31     fmt.Println(price)
32     fmt.Println(err)
33 }
34
35 func TestMovingPriceEndOfMonth_ExWarehouse(t *testing.T) {
36     f := &MovingPriceEndOfMonth{}
37     var v ValuationStrategy
38     v = f
39     v.init()
```

```
    v.InWarehouse(1, 1, 1)
    v.InWarehouse(1, 2, 2)
    f.Settlement(1)
    v.InWarehouse(1, 3, 1)
    price, err := v.ExWarehouse(1, 2)
    fmt.Println("***************")
    fmt.Println(price)
    fmt.Println(err)
}

func TestSpecificPrice_ExWarehouse(t *testing.T) {
    f := &SpecificPrice{}
    var v ValuationStrategy
    v = f
    v.init()
    v.InWarehouse(1, 1, 1)
    v.InWarehouse(2, 2, 2)
    price, err := v.ExWarehouse(2, 1)
    fmt.Println("***************")
    fmt.Println(price)
    fmt.Println(err)
}
```

5.3 并发分析及控制策略

数据库是共享资源，通常有许多个事务同时在运行。当多个事务并发地存取数据库时，就会产生同时读取或修改同一数据的情况。若对并发操作不加控制就可能会存取和存储不正确的数据，破坏数据库的一致性。所以数据库管理系统必须提供并发控制机制。数据库为了提高资源利用率和事务执行效率、降低响应时间， 允许事务并发执行。但是多个事务同时操作同一对象，必然存在冲突，事务的中 间状态可能暴露给其它事务，导致一些事务依据其它事务中间状态，把错误的值写到数据库里。需要提供一种机制，保证事务执行不受并发事务的影响，让用户感觉，当前仿佛只有自己发起的事务在执行。

对于仓储管理来说，主要是过多的用户同时访问仓库物料信息，而库存信息管理员同时也对物料信息进行读写操作时，存在同时对同一数据进行读写操作的可能性，导致用户读取数据错误的问题。其次，在仓库员工数量增多后，也会增加对某些物料数据同时读写的可能性，这就需要系统对并发操作进行合理的控制管理。

在解决数据库的事务并发访问问题时，虽然将事务串行化可以保证数据在多事务并发处理下不存在数据不一致的问题，但串行执行使得数据库的处理性能大幅度地下降，常常是我们接受不了的。一般来说，我们常常结合事务隔离级别和其它并发机制来保证事务的并发，以此来兼顾事务并发的效率与安全性。

5.3.1 两阶段锁协议（2PL，Two-phase locking）

既然要保证操作按正确的顺序执行，最容易想到的方法就是加锁保护访问对象。数据库系统的锁管理器模块，专门负责给访问对象加锁和释放锁，保证只有持有锁的事务，才能操作相应的对象。锁可以分为两类：S-Lock 和 X-Lock，SLock 是读请求使用的共享锁，X-Lock 是写请求使用的排他锁。它们的兼容性如下：操作同一个对象，只有两个读请求相互兼容，可以同时执行，读写和写写操作都会因为锁冲突而串行执行。

2PL 是数据库最常见的基于锁的并发控制协议，它包含两个阶段：

阶段一：Growing，事务向锁管理器请求它需要的所有锁；

阶段二：Shrinking，事务释放 Growing 阶段获取的锁，不允许再请求新锁。

虽然 2PL 已经在并发控制上起到一定作用，但还是避免不了一些问题。例如：A 事务在提交前就释放锁，此时 B 事务再拿到相同的锁，读取到的是 A 事务提交前的数据。所以一般情况下数据库使用加强版的 SS2PL，在阶段二 Shrinking 时，必须在事务结束后才释放所有的锁，杜绝了为提交的数据被读取到的可能性。

为了提高系统的性能，可以增加意向锁。锁的对象可以是行、页面、表。层次越高的锁，可以有效减少对资源的占用，显著减少锁检查的次数，但会严重限制并发。层次越低的锁，有利于并发执行，但在事务请求对象多的情况下，需要大量的锁检查。

数据库系统为了解决高层次锁限制并发的问题，引入了意向锁的概念：

a. Intention-Shared (IS)：表明其内部一个或多个对象被 S-Lock 保护，例如某表加 IS，表中至少一行被 S-Lock 保护；

b. Intention-Exclusive (IX)：表明其内部一个或多个对象被 X-Lock 保护。例如某表加 IX，表中至少一行被 X-Lock 保护；

c. Shared+Intention-Exclusive (SIX)：表明内部至少一个对象被 XLock 保护，并且自身被 S-Lock 保护。例如某个操作要全表扫，并更改表中几行，可以给表加 SIX。

若一个事务需要对表进行操作，请求表的 X 锁，那么看到表持有IX 就直接等待了，而不用逐个检查表内的行是否持有行锁，有效减少了检查开销。若有别的读写事务过来，而表加的是 IX 而非 X，此时并不会阻止对行的读写请求（先在表上加 IX，再去记录上加 S/X），事务如果没有涉及已经加了 X 锁的行，则可以正常执行，增大了系统的并发度。

5.3.2 基于 Timing Order（T/O）的并发控制

为每个事务分配时间戳，并以此决定事务执行顺序。当 A 事务的时间戳小于 B 事务时，数据库系统要保证 A 事务先于 B 事务执行。

给每个可能被访问到的数据项维护两个值，W-time 和 R-time ，W-time(d) 维护所有成功执行 write(d) 的事务的时间戳最大值，R-time(d)维护所有成功执行 read(d) 的事务的时间戳最大值。

设 Time(d) 表示事务 T 的时间戳，当事务 T 试图执行 read(d) 时：

a. 若 time(T) < W-time(d)，说明 T 需要读的值已经被覆盖，因此操作失败，事务 T 回滚；

b. 若 time(T) ⩾ W-time(d)，则可以执行 read(d)，R-time(d) 被设置为time(T) 和 R-time(d) 的最大值。

当事务 T 试图执行 write(d) 时：

a. 若 time(T) < R-time(d)，则事务 T 执行 write(d) 会导致其它某些事务read(d) 读取的值不正确，因此操作失败，事务 T 回滚；

b. 若 time(T) < W-time(d)，则事务 T 试图写入的值已过时，因此操作失败，事务 T 回滚；

c. 其它情况下，可以执行 write(d) ，W-time(d) 被设置为 time(T)。

六、活动图

1. 入库业务流程：系统用户能够主动查询进货单并进行进货入库，也可以处理退货申请使货物入库。

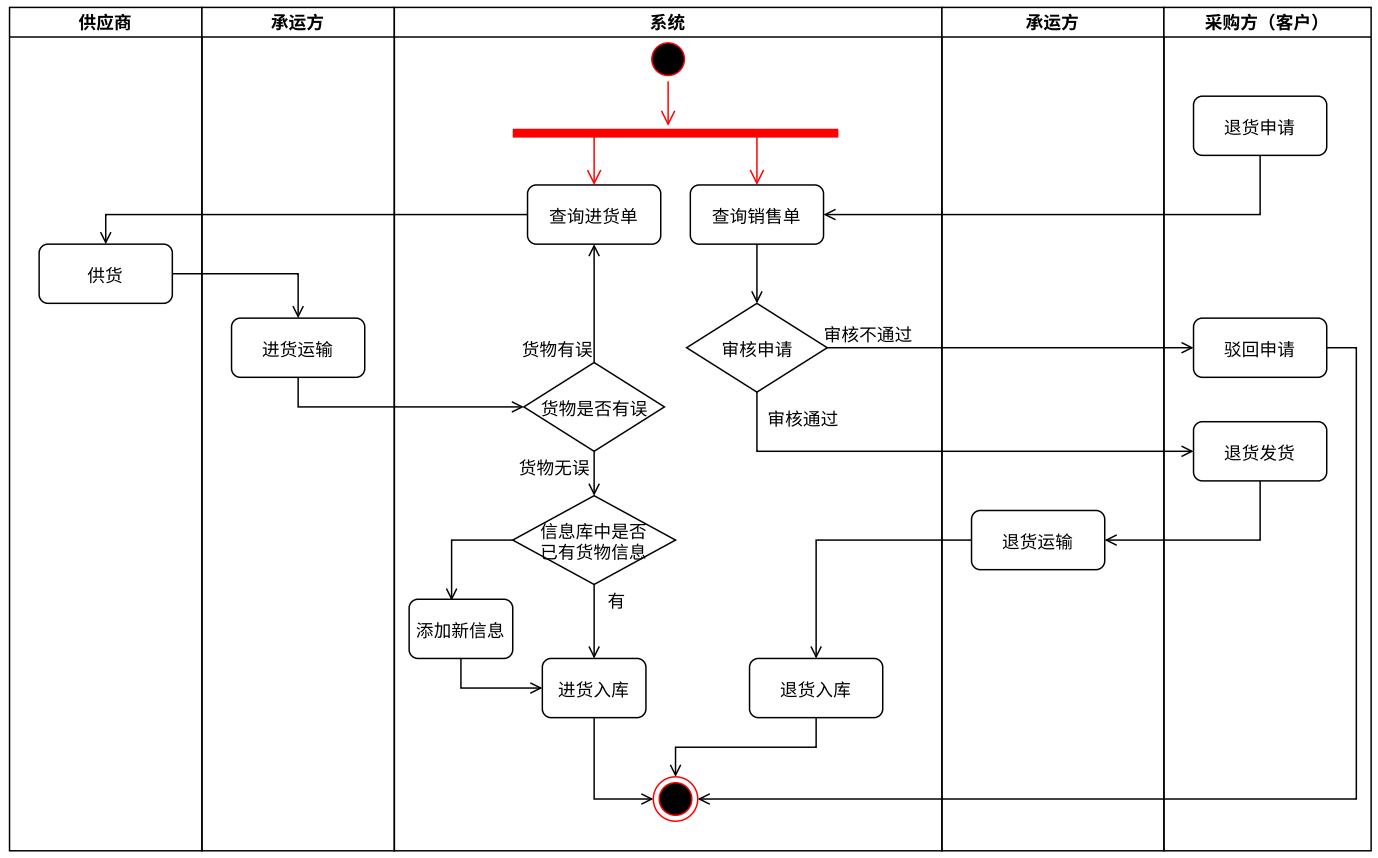

图6-1 入库业务活动图

2. 出库业务流程：系统可以处理客户采购单，进行货物销售和出库。

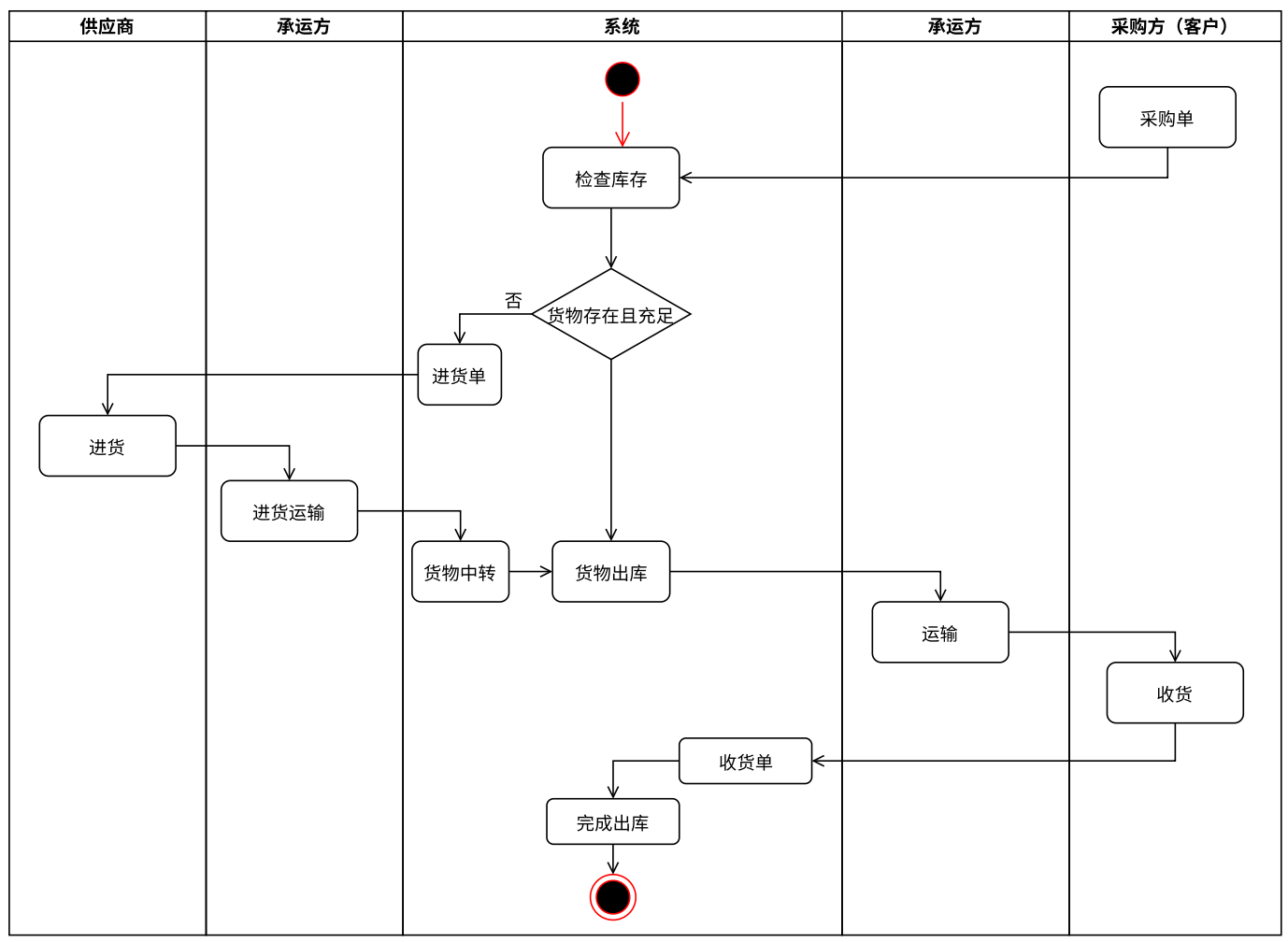

图6-2 出库业务活动图

3. 库存预警业务流程：系统将通过三种方式进行预警判断，在出入库业务结束后、主动选择货物进行分析、定时自动分析。

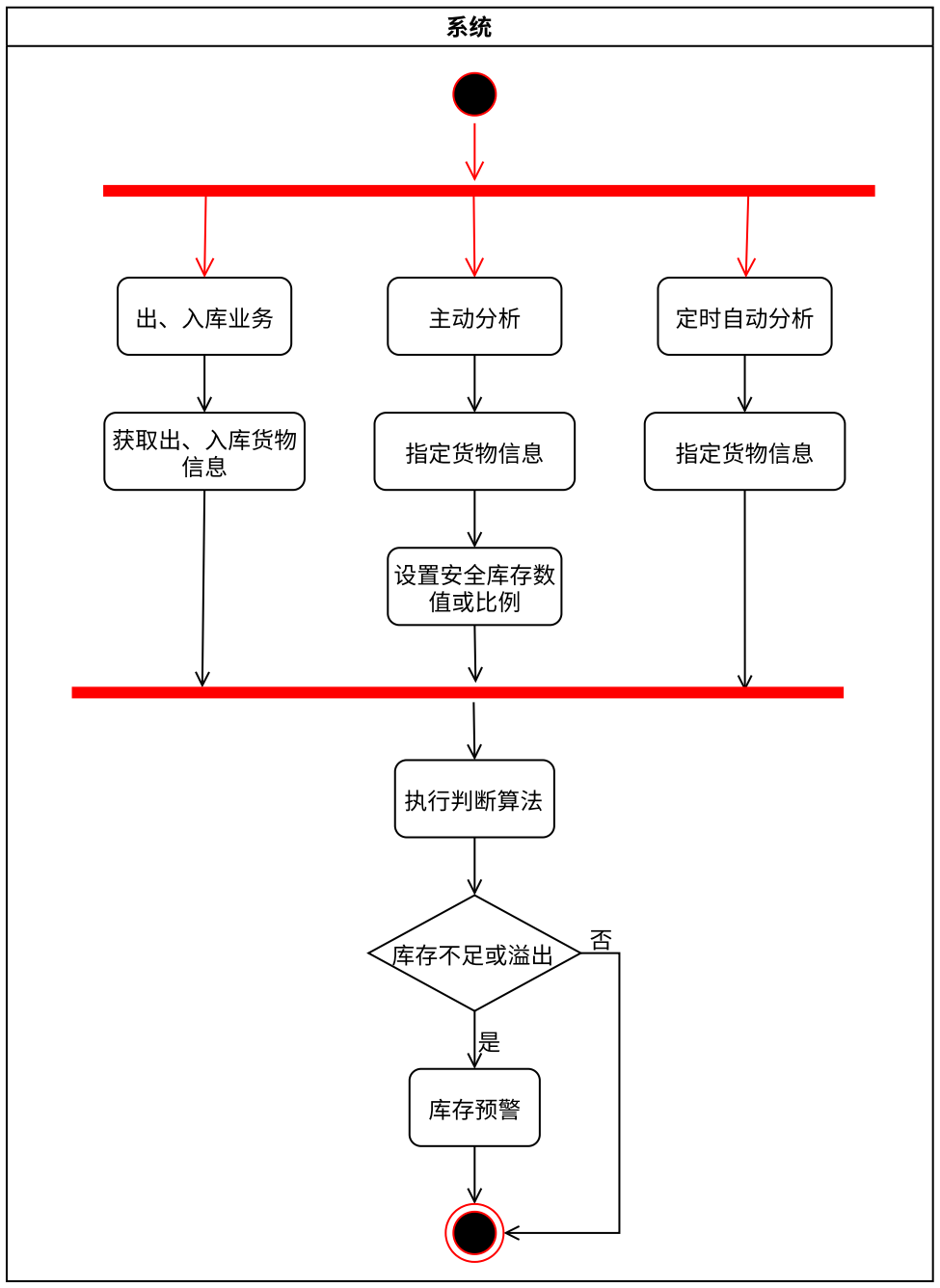

图6-3 库存预警业务活动图

4. 生成库存报表业务流程：在进行会影响到库存信息的业务后，系统会自动生成业务报表。

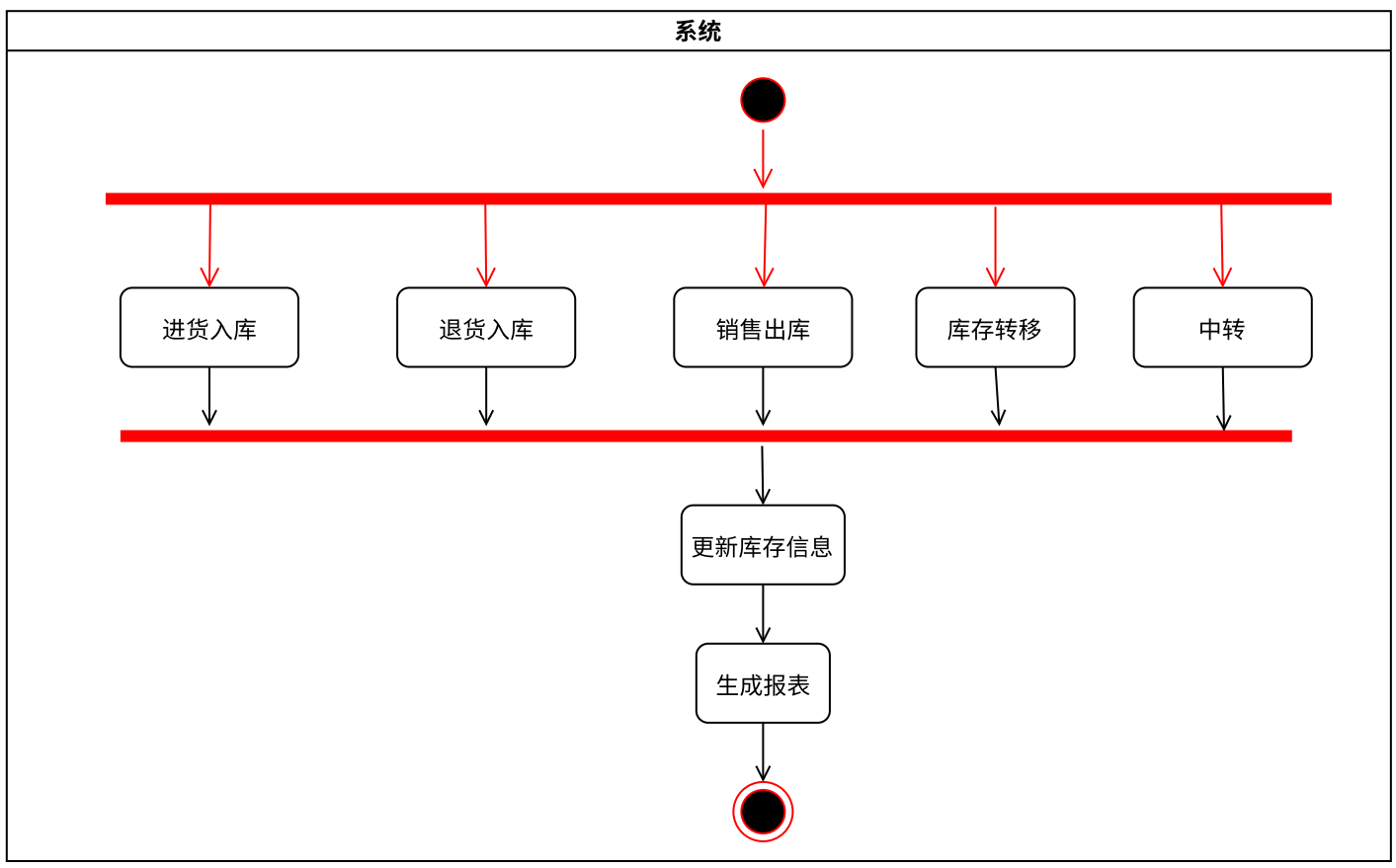

图6-4 生成库存报表业务活动图

5. 仓库内移位业务流程：系统可以生成库存移位申请，由库房管理员审核。

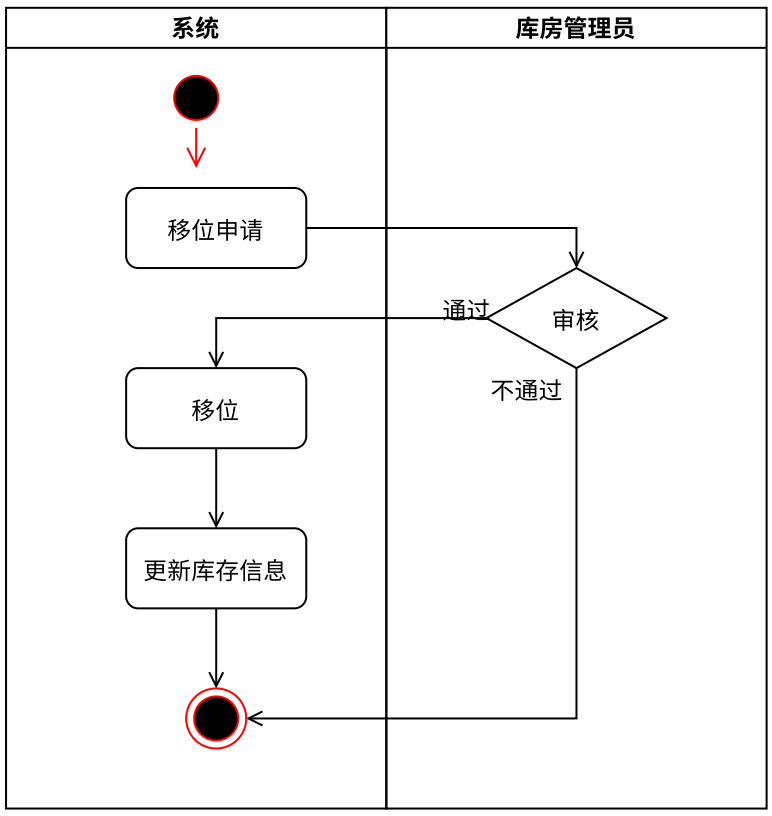

图6-5 仓库内移位业务活动图

6. 信息录入业务流程：在进行合作方信息登记时，先从信息数据库中查询已有的合作方记录核实该合作方的信息是否已存在，对不存在的合作方信息（即新合作方信息）进行登记。

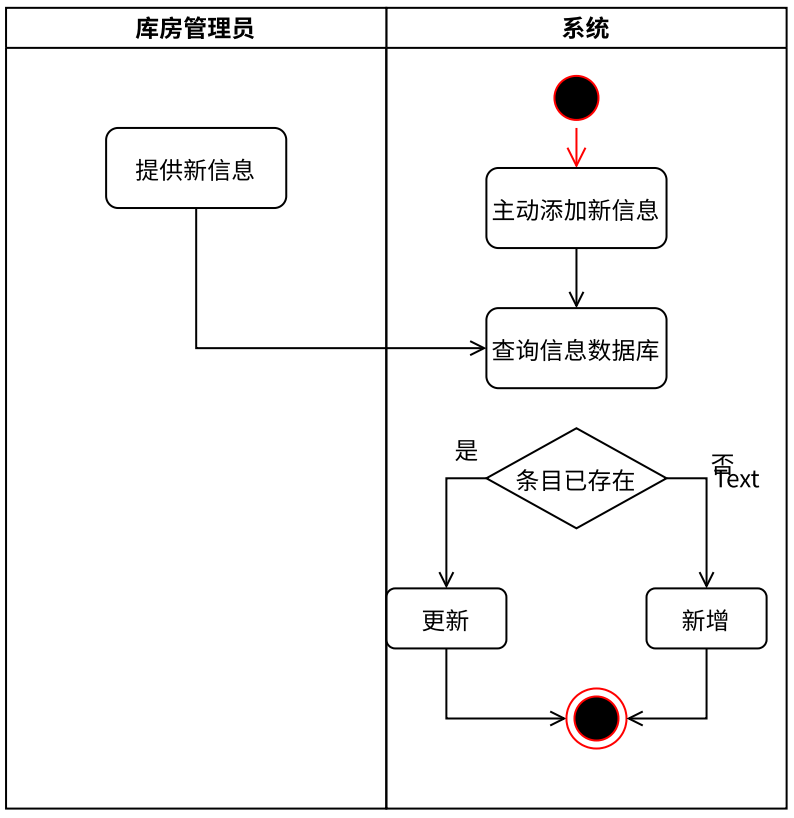

图6-6 信息录入业务活动图

七、多架构图

7.1 软件架构

本系统采用浏览器/服务器（Browser/Server，下文简称为B/S）的架构模式，前端运用Model-View-ViewModel（下文简称为MVVM）的模式，后端运用三层架构。

B/S架构是Web兴起后的一种网络结构模式，是对客户端/服务器（Client/Server，下文简称C/S）架构的一种改进。用户只要有浏览器、网络就能访问系统，进行业务处理。相比C/S架构，采用B/S架构的应用开发简单、分布性强、易于维护、业务拓展便利。

三层架构模式将后端的开发过程划分为三层：表现层、业务逻辑层和数据访问层。表现层直接和前端打交道，负责接收前端的Ajax请求和返回JSON数据给前端；业务逻辑层处理表现层转发过来的前端请求，从数据访问层中获取数据返回到表现层；数据访问层也叫持久层，直接对数据库进行增删改查操作，并将获得的数据返回业务逻辑层。

MVVM模式是对Model-View-Controller（简称MVC）模式的改进。MVVM的核心思想是双向数据绑定。当用户操作界面（View）时，视图模型（View-Model）能够感知到变化，然后通知模型（Model）做出相应的变化；反之，当Model发生变化时，视图模型也能够感知到变化，让用户界面响应更新。

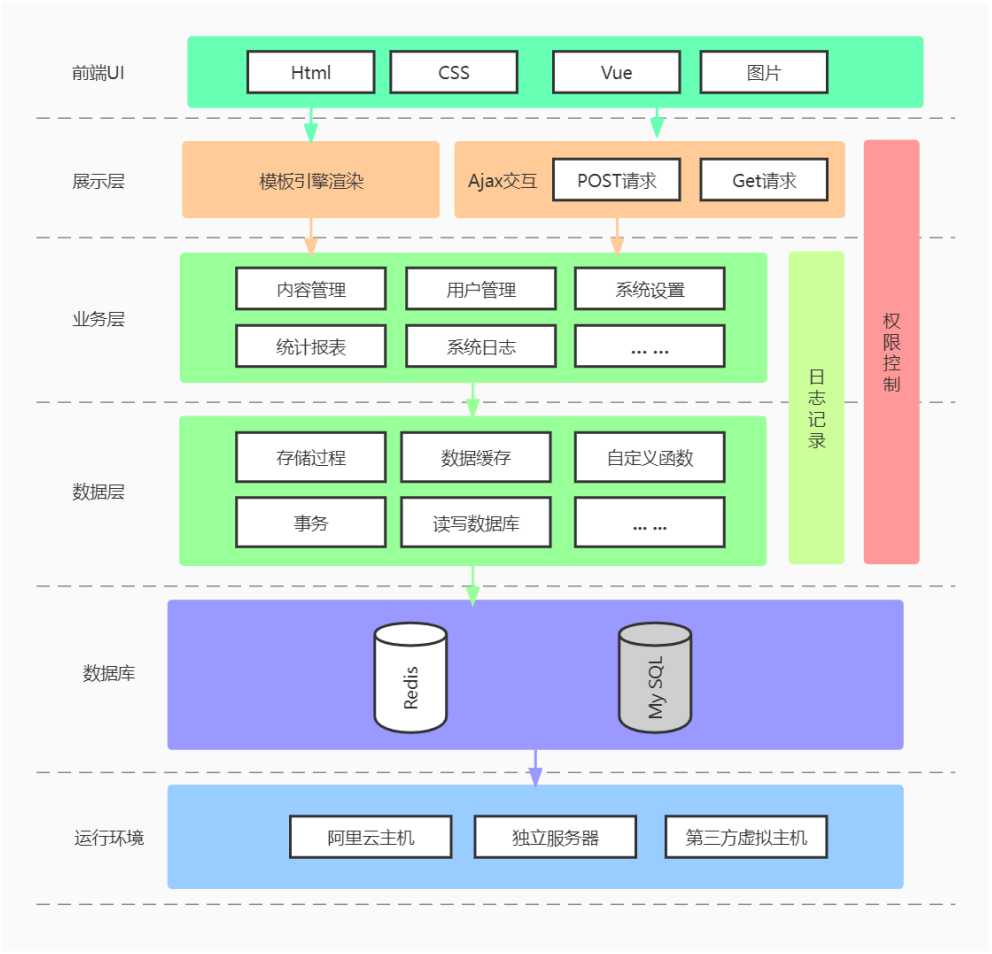

图7-1 软件架构图

7.2 逻辑视图

从逻辑视角出发描述ERP系统的架构，主要关注ERP系统的结构、模块性和基本构件等方面。在逻辑视图中，采用UML包展示了ERP系统的最重要的三个层：表示层、业务层和持久层。

表示层展示的是用户与ERP系统直接交互的界面，但是它的职责并不仅仅是显示界面，而是需要完成三件事情：（1）从界面中取得数据跟后台服务器交互，（2）跟后台交互后进行数据绑定，（3）将绑定的数据呈现在页面中。一般来说，这一层的设计会采用MVC的模式，M称为模型也就是实体类，用于数据的封装和数据的传输；V为视图也就是页面组件，用于数据的展示；C为控制也就是流程事件，用于流程的控制。

业务层包括了ERP系统的主要业务。它是表示层和持久层之间沟通的桥梁，主要负责数据的传递和处理。在日常的代码开发中一般对应着逻辑Service层，对于一些复杂的逻辑判断和涉及到数据库的数据验证都需要在这一层做出处理，同时根据传入的值返回用户想得到的值，或者处理相关的操作。

持久层也称为数据访问层，包含ERP系统涉及的核心数据。这一层其实就是跟数据库直接打交道的层，它通过连接数据库，根据传入的值对数据库进行增删改查。

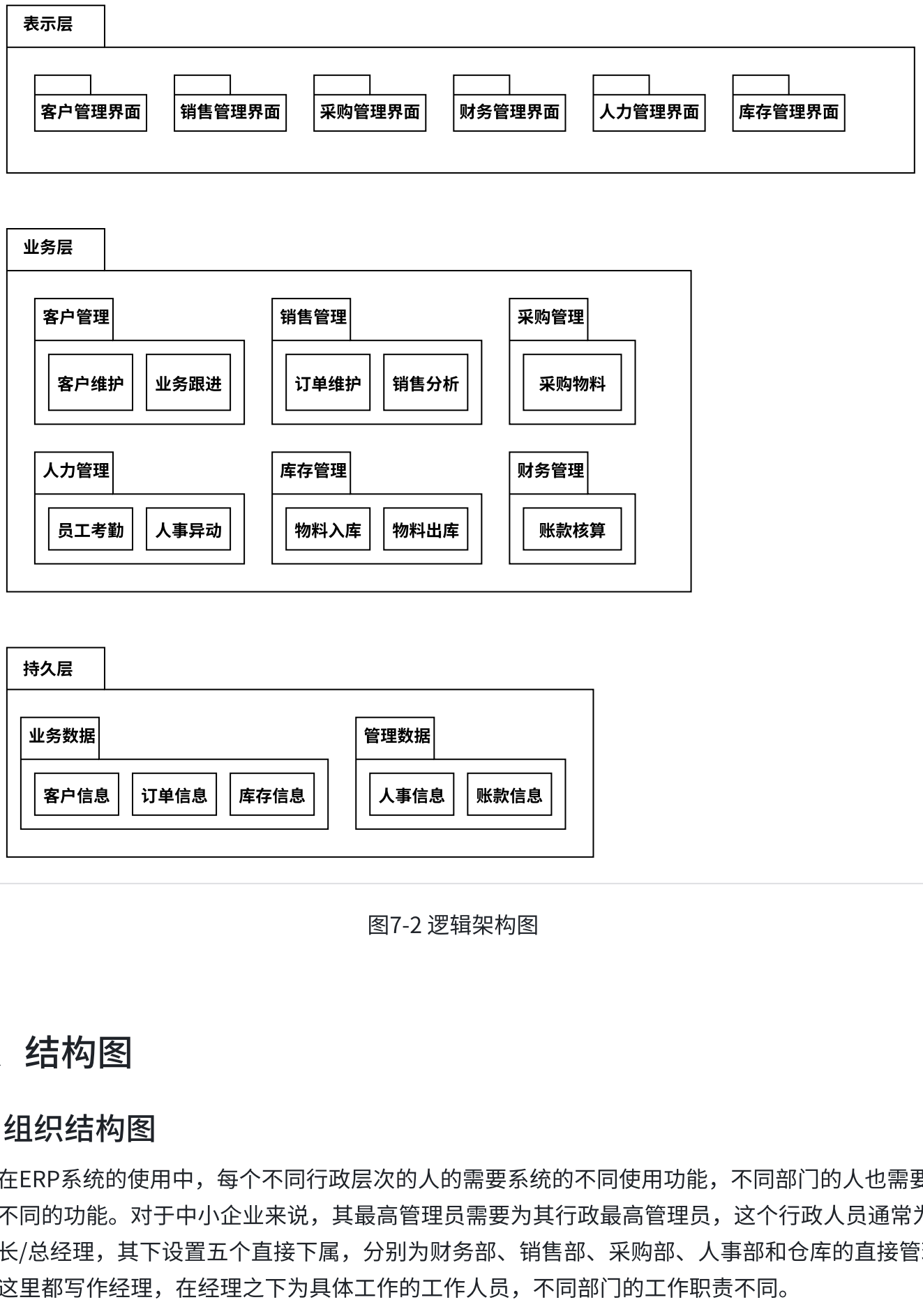

图7-2 逻辑架构图

八、结构图

8.1 组织结构图

在ERP系统的使用中，每个不同行政层次的人的需要系统的不同使用功能，不同部门的人也需要使用到不同的功能。对于中小企业来说，其最高管理员需要为其行政最高管理员，这个行政人员通常为董事长/总经理，其下设置五个直接下属，分别为财务部、销售部、采购部、人事部和仓库的直接管理者，这里都写作经理，在经理之下为具体工作的工作人员，不同部门的工作职责不同。

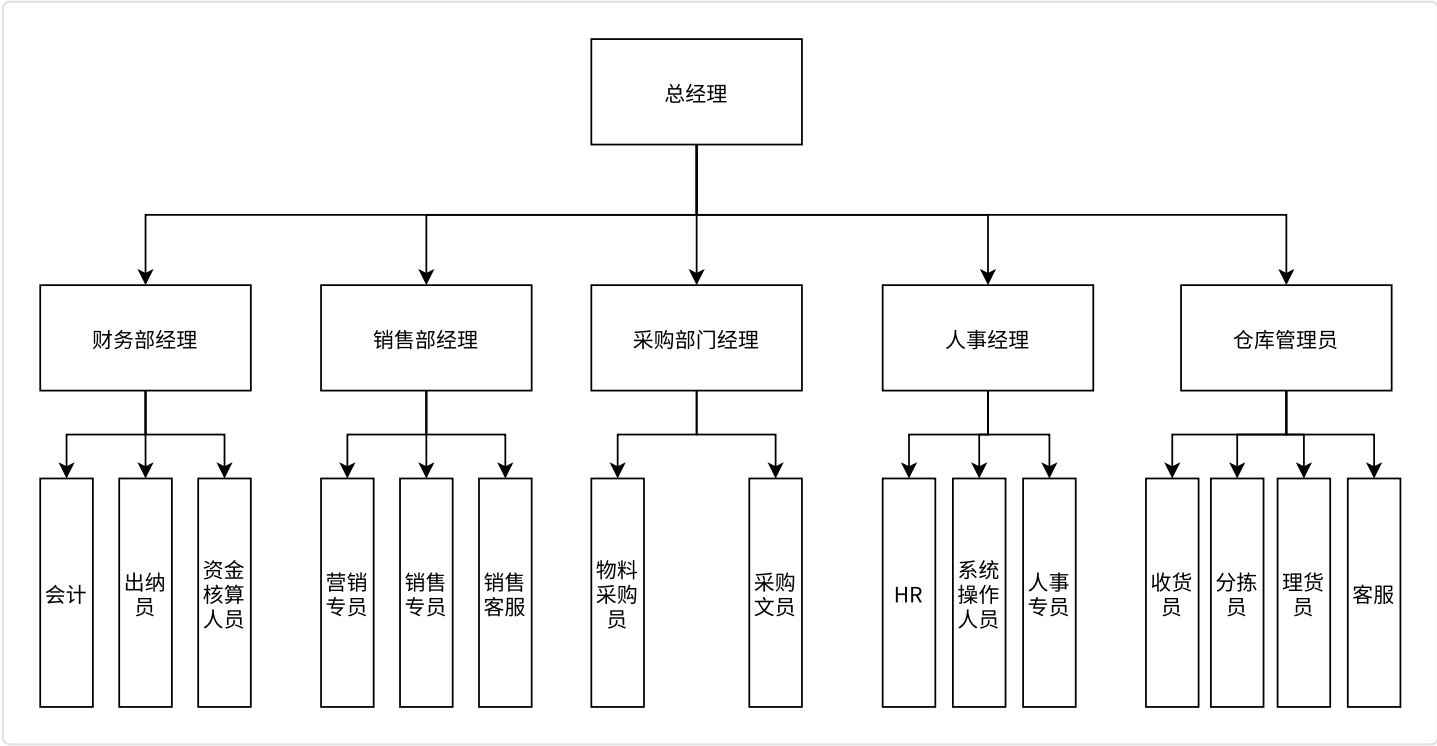

图8-1 组织结构图

8.2 角色结构图

在ERP系统开发中，角色组织是一个比较重要的模块，在企业组织中往往会大量使用到角色，角色顾名思义就是将不同的用户进行划分和归纳。因为在系统架构中，系统资源往往会与角色进行关联，这样不同的角色可以访问系统中不同的资源，而人又与角色绑定，从而达到给角色授权资源的同时，不同角色下面的人又可以进行不同资源的访问控制。

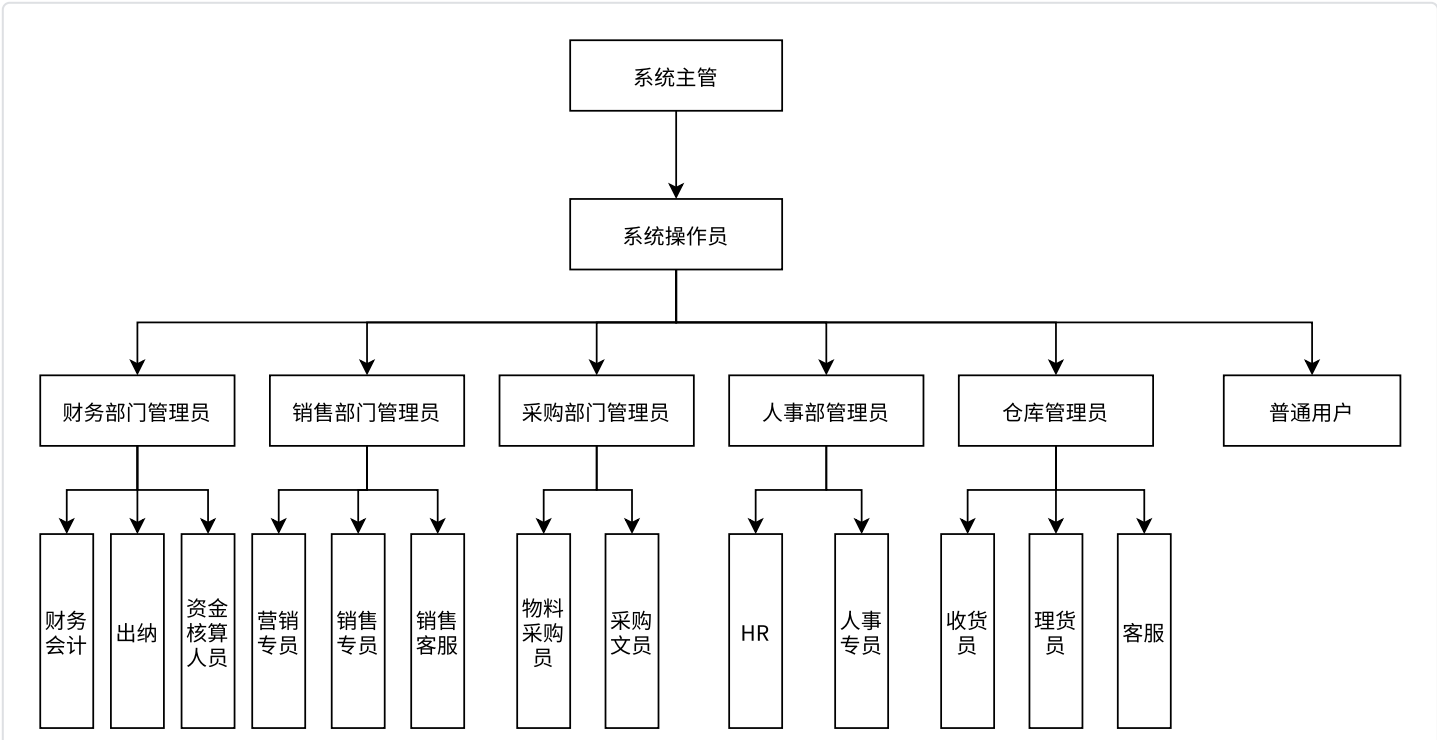

图8-2 角色结构图

拥有特定角色的用户可以看成是具有相同操作权限的一个组合，而普通用户则只具有最简单的权限，如果需要其他权限需要由操作员亲自赋予，是一种比较可以被自由赋予权限和职责的角色。