第八章　面向对象的系统设计 II

曹东刚
caodg@pku.edu.cn

北京大学信息学院研究生课程 - 面向对象的分析与设计
http://sei.pku.edu.cn/~caodg/course/oo

内容提要

什么是控制驱动部分

控制驱动部分

是 OOD 模型的外围组成部分之一，由系统中全体主动类构成。这些主动类描述了整个系统中所有的主动对象，每个主动对象是系统中一个控制流的驱动者

控制流（control flow）：进程（process）和线程（thread）的总称
有多个控制流并发执行的系统称作并发系统（多任务系统）

为什么需要控制驱动部分

- 并发行为是现实中固有的
  - 外围设备与主机并发工作的系统
  - 有多个窗口进行人机交互的系统
  - 多用户系统
  - 多个子系统并发工作的系统
  - 单处理机上的多任务系统
  - 多处理机系统
- 多任务的设置
  - 描述问题域固有的并发行为
  - 表达实现所需的设计决策
- 隔离硬件、操作系统、网络的变化对整个系统的影响

由系统总体方案决定的实现条件

- 计算机硬件：性能、容量和 CPU 数目
- 操作系统：对并发和通讯的支持
- 网络方案：网络软硬件设施、网络拓扑结构、通讯速率、网络协议等
- 软件体系结构
- 编程语言：对进程和线程的描述能力
- 其它软件：如数据管理系统、界面支持系统、构件库等——对共享和并发访问的支持

软件体系结构

描述了构成系统的元素、这些元素之间的相互作用、指导其组合的模式以及对这些模式的约束

几种典型的软件体系结构风格

- 管道与过滤器风格（pipe and filter style）
- 面向对象风格（object-oriented style）
- 层次风格（layered style）
- 黑板风格（blackboard style）
- 进程控制风格（process control style）
- 客户-服务器风格（client-server style）

分布式系统的体系结构风格

- 主机 + 仿真终端体系结构
- 文件共享体系结构
- 客户-服务器体系结构
  - 二层客户-服务器体系结构
  - 三层客户-服务器体系结构
  - 对等式客户-服务器体系结构
  - 瘦客户-服务器体系结构
  - 胖客户-服务器体系结构
- 浏览器-服务器体系结构

系统的并发性

进程（process）概念出现之前，并发程序设计困难重重，主要原因：

- 并发行为彼此交织，理不出头绪
- 与时间有关的错误不可重现

进程概念的提出使这个问题得到根本解决

系统的并发性

进程的全称是顺序进程（sequential process），其基本思想是把并发程序分解成一些顺序执行的进程，使得：

- 每个进程内部不再包含并发行为
  所以叫做顺序进程，其设计避免了并发问题
- 多个进程之间是并发（异步）执行的
  所以能够构成并发程序

进程与线程

由于并行计算的需要，要求人为地在顺序程序内部定义和识别可并发执行的单位，线程的概念就诞生了

线程与进程的区别：

- 进程既是处理机分配单位，也是存储空间、设备等资源的分配单位（重量级的控制流）
- 线程只是处理机分配单位（轻量级的控制流）
- 一个进程可以包含多个线程，也可以是单线程的

应用系统的并发性

从网络、硬件平台的角度看：

- 分布在不同计算机上的进程之间的并发
- 在多 CPU 的计算机上运行的进程或线程之间的并发
- 在一个 CPU 上运行的多个进程或线程之间的并发

应用系统的并发性

从应用系统的需求看：

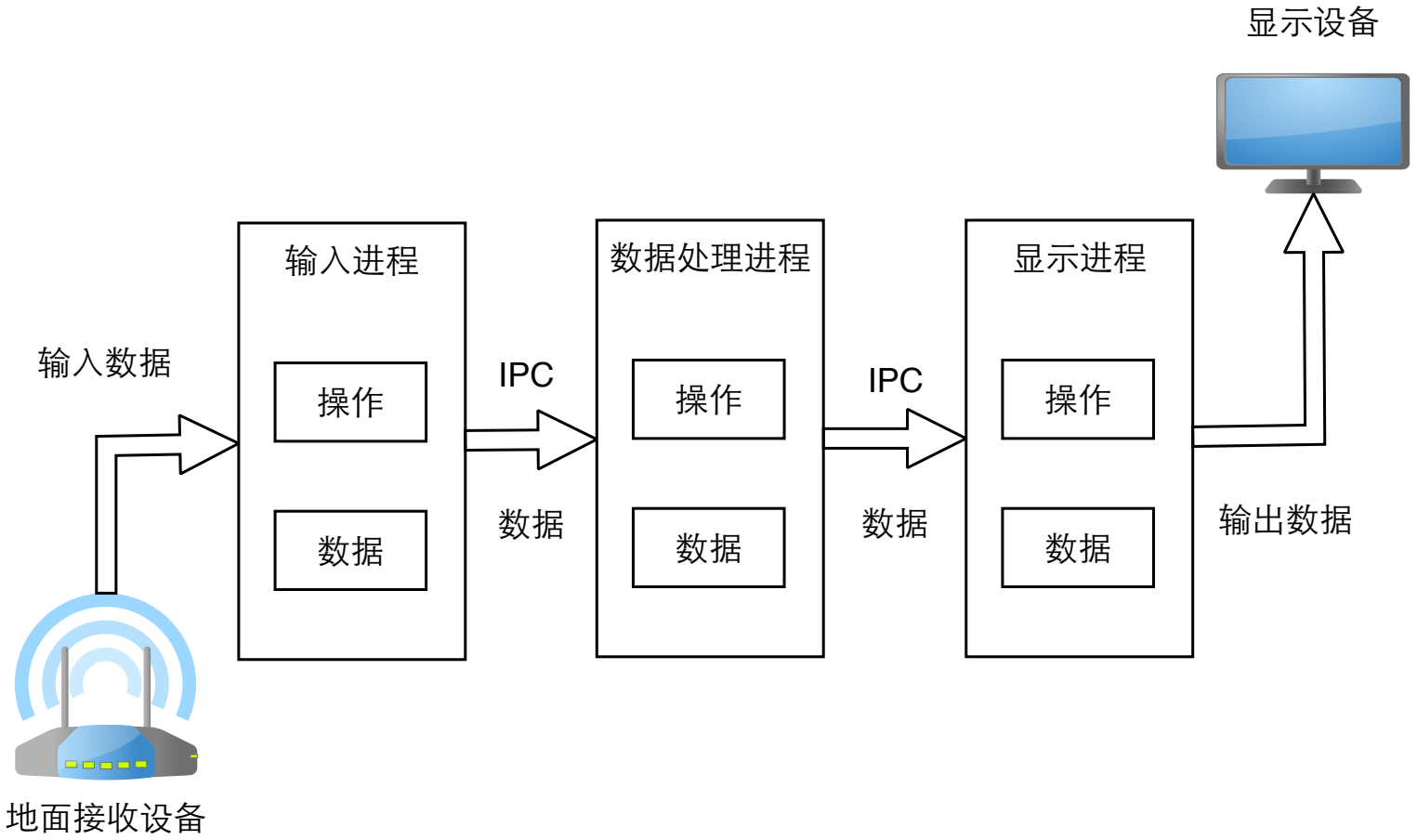
- 需要跨地域进行业务处理的系统
- 需要同时使用多台计算机或多个 CPU 进行处理的系统
- 需要同时供多个用户或操作者使用的系统
- 需要在同一时间执行多项功能的系统
- 需要与系统外部多个参与者同时进行交互的系统

处理应用系统并发的例子

问题描述:

某单位想开发一个卫星遥感信息处理系统，要求是：实时把通过地面接收设备传来的卫星遥感图片信息输入系统，经过必要的数据处理，及时将图片显示在屏幕上。

处理应用系统并发的例子



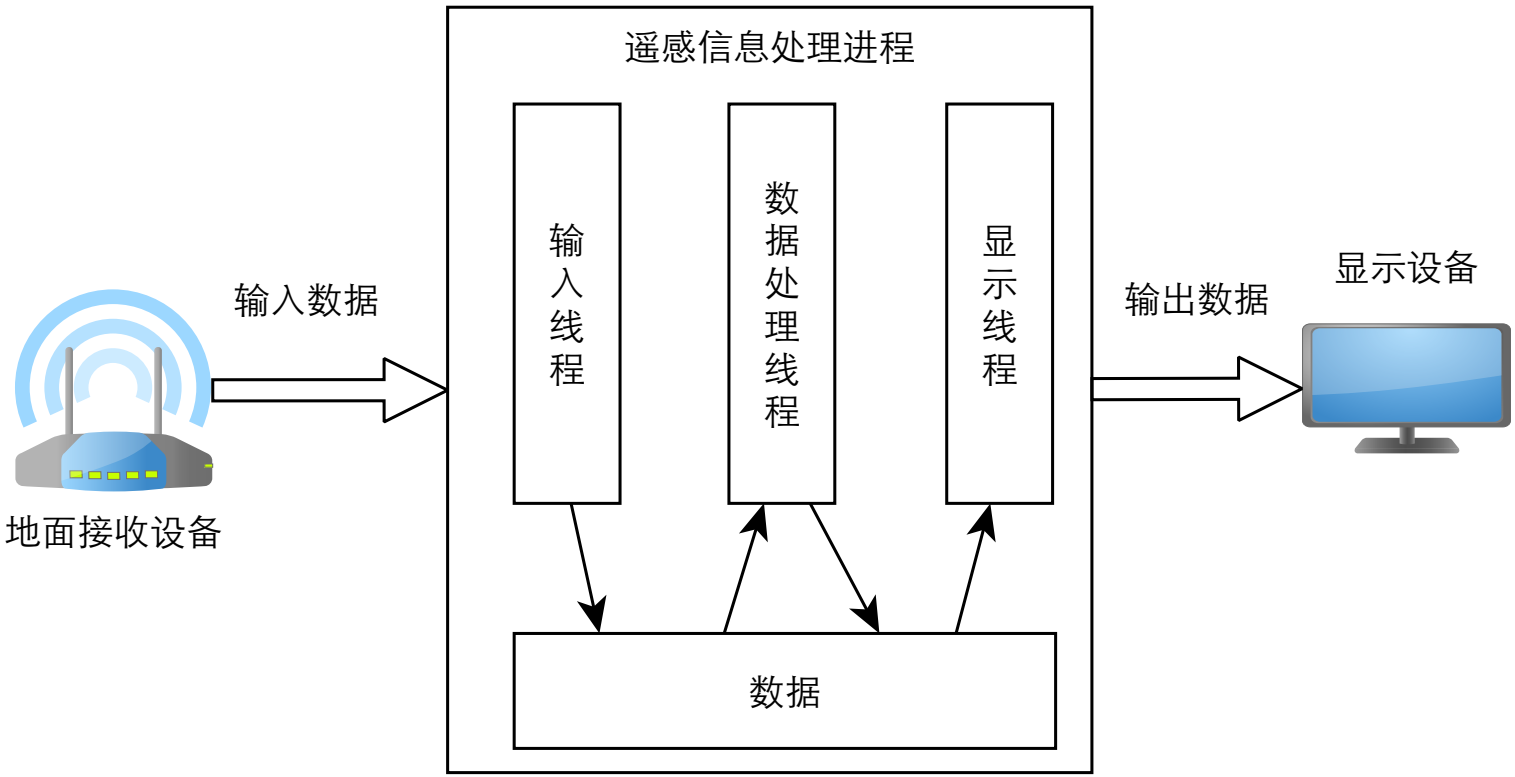
用多进程实现的遥感信息处理系统

处理应用系统并发的例子

新的需求:

针对前页10例子中多进程共享数据速度慢的问题，希望改变设计，采用多线程技术实现并发，避免控制流之间传送大量数据。

处理应用系统并发的例子



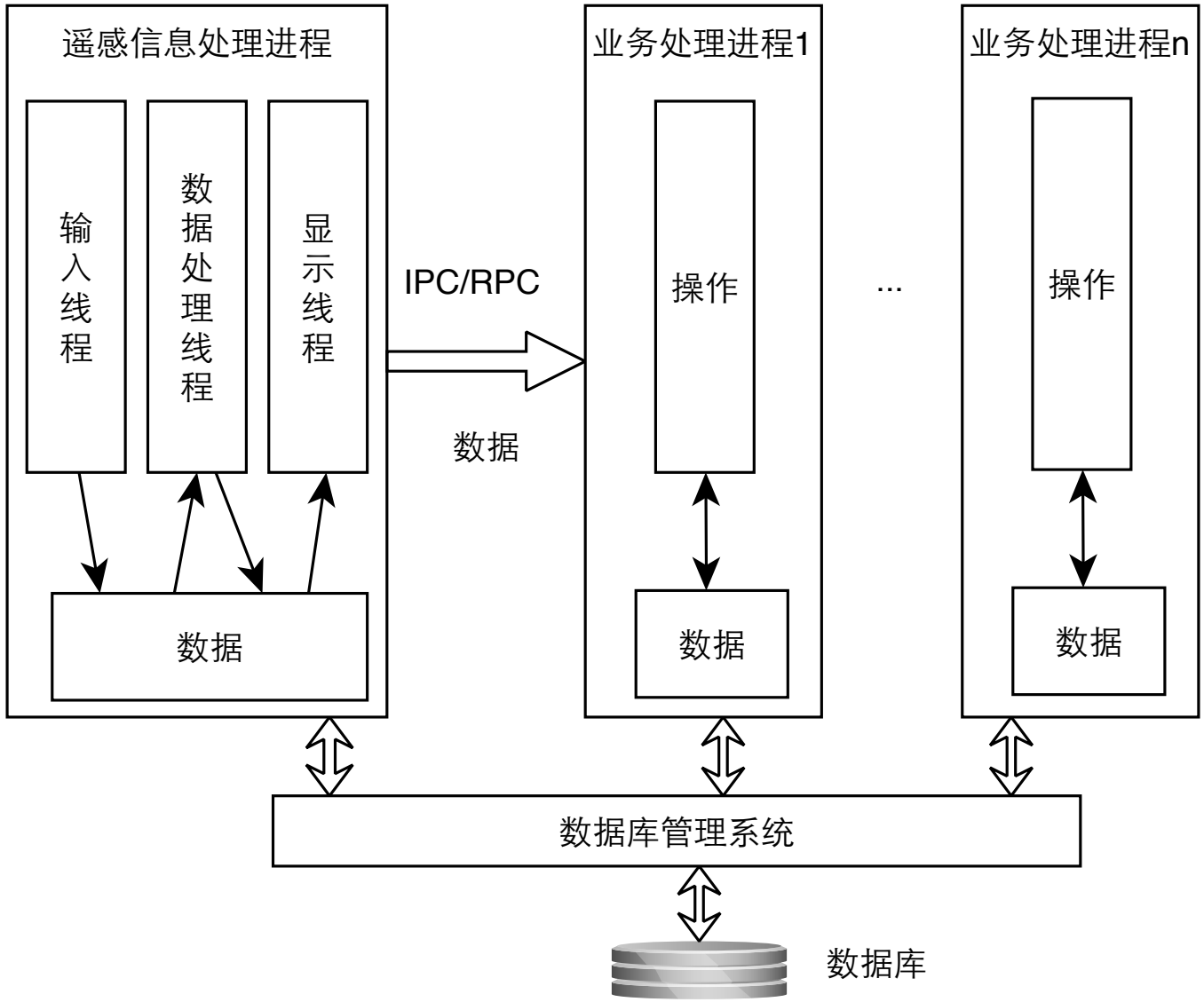
用多线程实现的遥感信息处理系统

处理应用系统并发的例子

拓展业务:

考虑面向不同应用的遥感信息处理系统，不仅需要把图片信息实时显示出来，而且需进行更多处理，如面向地理信息系统的特征信息提取等。

处理应用系统并发的例子



同时采用多线程和多进程的多应用遥感信息处理系统

讨论：进程 vs 线程

进程：重量，分布内存
线程：轻量，共享内存

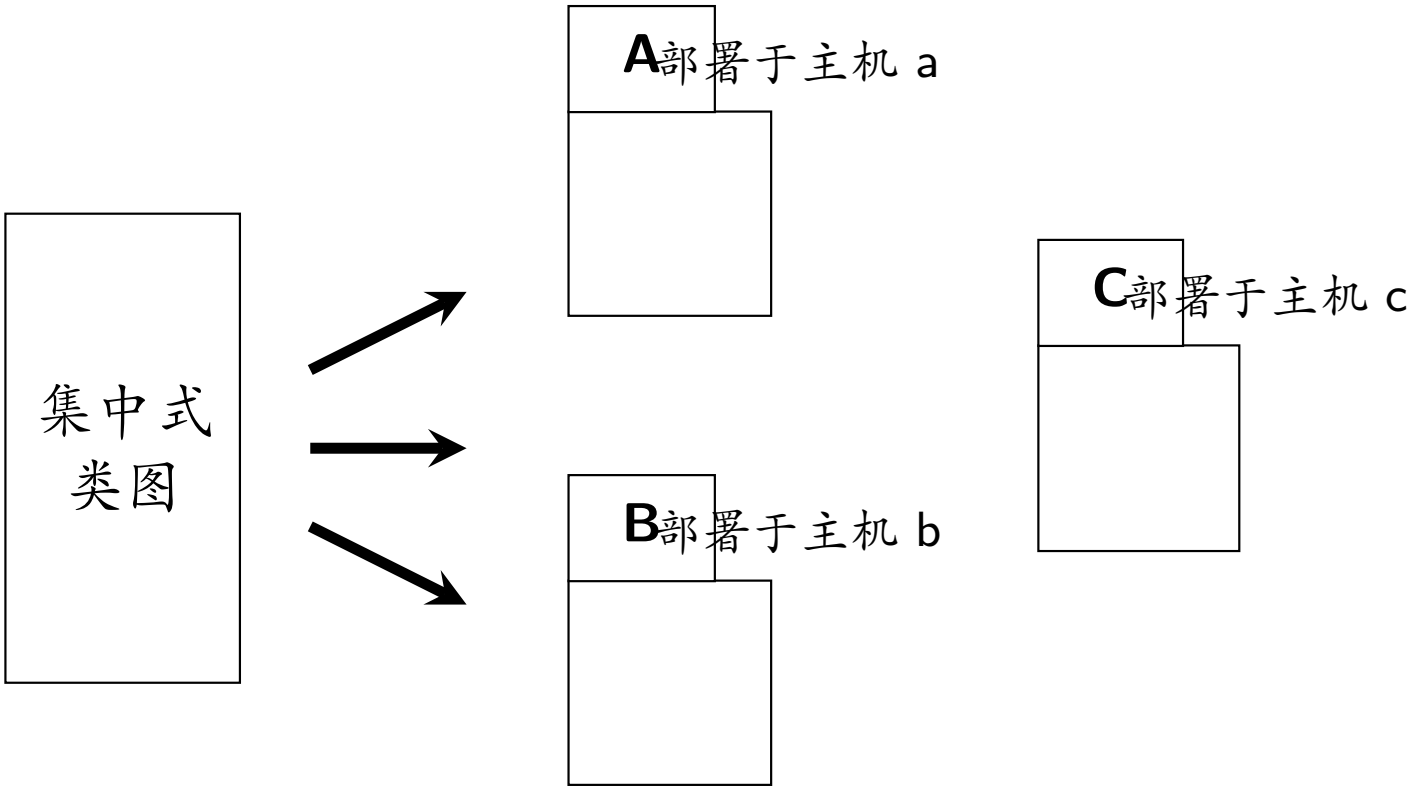
- 数据访问的成本和效率？
- 创建、销毁、切换的代价？
- 健壮性？
- 易于程序员编写并发程序？

选择软件体系结构风格

- 二层客户–服务器体系结构
  （数据）服务器–客户机
- 三层客户–服务器体系结构
  数据服务器–应用服务器–客户机
- 浏览器–服务器体系结构

确定系统分布方案

考虑分布方案之前：暂时将系统看作集中式的
确定分布方案之后：将对象分布到各个处理机上，
以每台处理机上的类作为一个包



确定系统分布方案

系统分布包括功能分布和数据分布，在面向对象的系统中都体现于对象分布

原则：减少远程传输，便于管理

决定对象分布：

- 软件体系结构
- 系统功能在哪些结点提供
- 数据在哪些结点长期存储管理，在哪些结点临时使用
- 参照用况，把合作紧密的对象尽可能分布在同一结点
- 追踪消息，把一个控制流经历的对象分布在同一结点

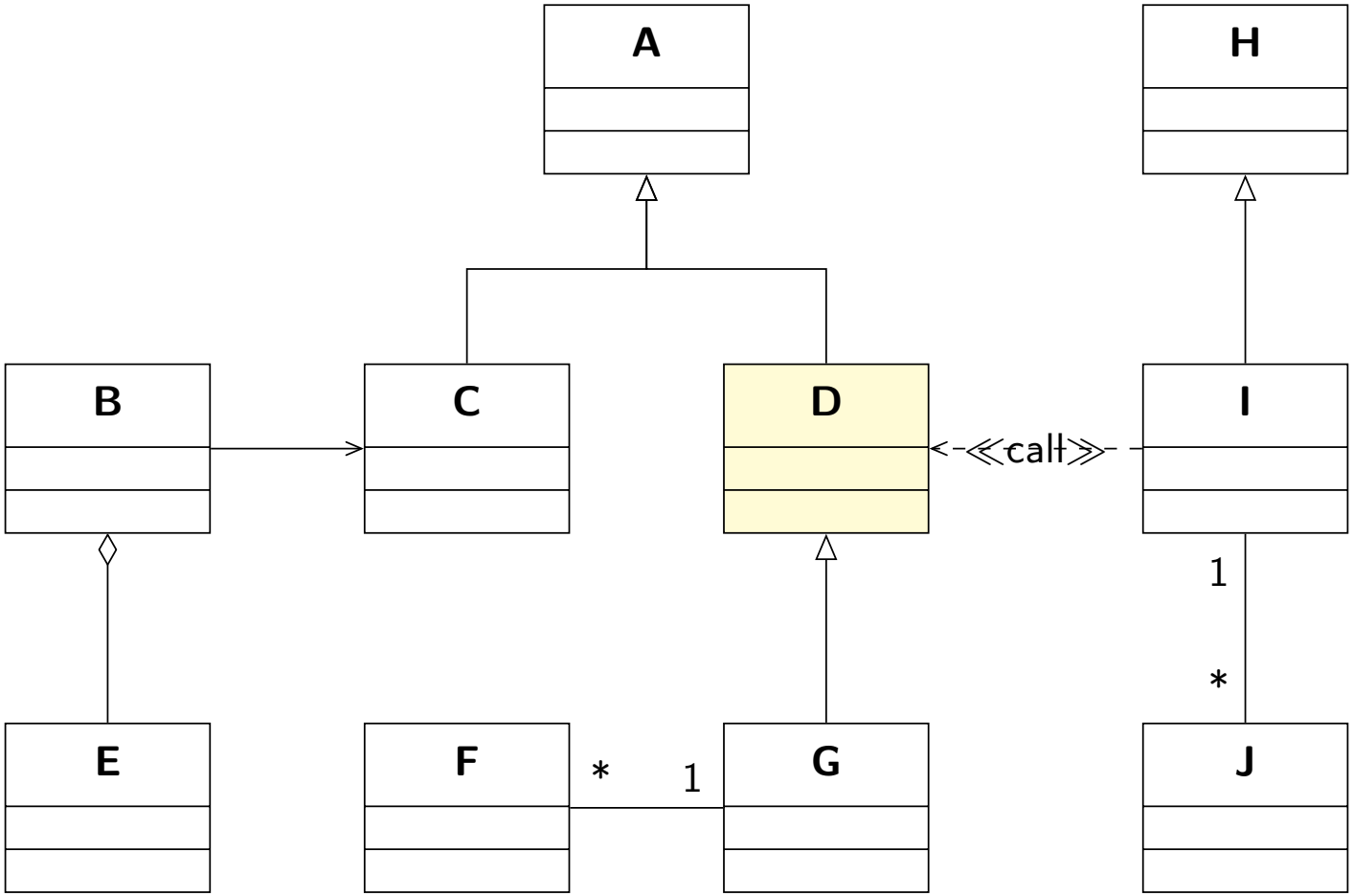
确定系统分布方案

类的分布：根据对象分布的需要
分布在每个结点上的对象，都需要相应的类来创建

策略 1 如果一个类只需要在一个结点上创建对象实例：
把这个类分布在该结点上

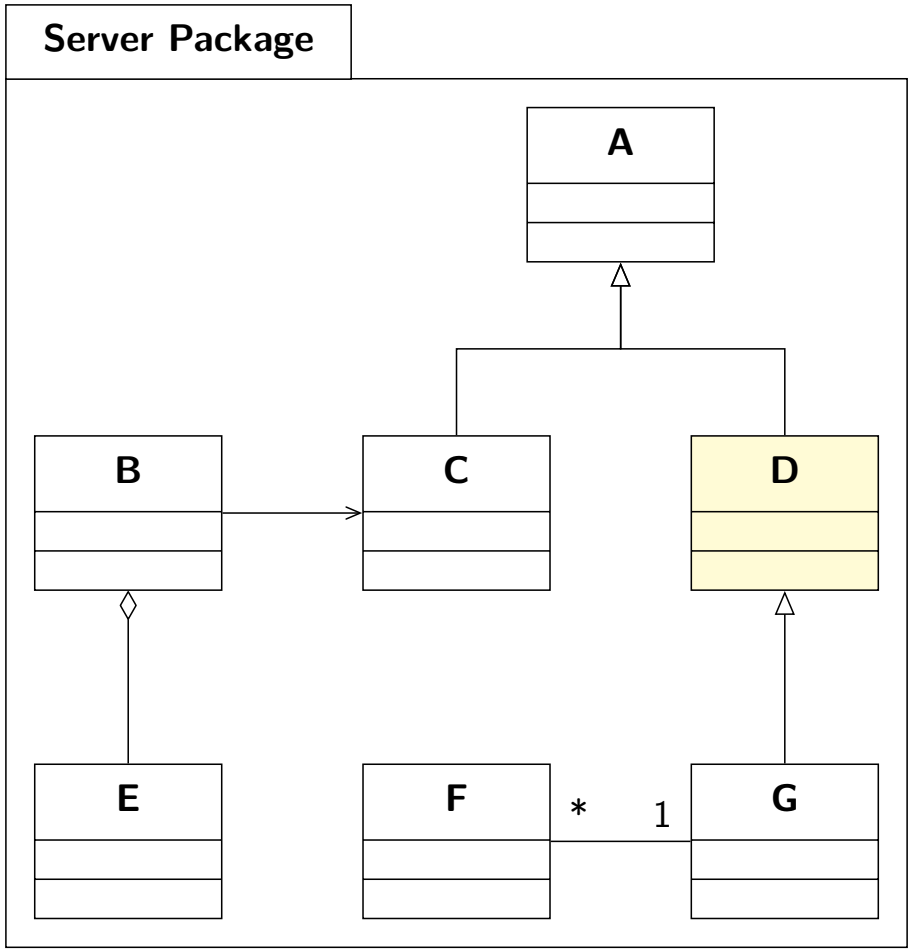
策略 2 如果一个类需要在多个结点上创建对象实例：
把这个类分布到每个需要创建其实例的结点上，
其中一个作为正本，其他作为副本

确定系统分布方案



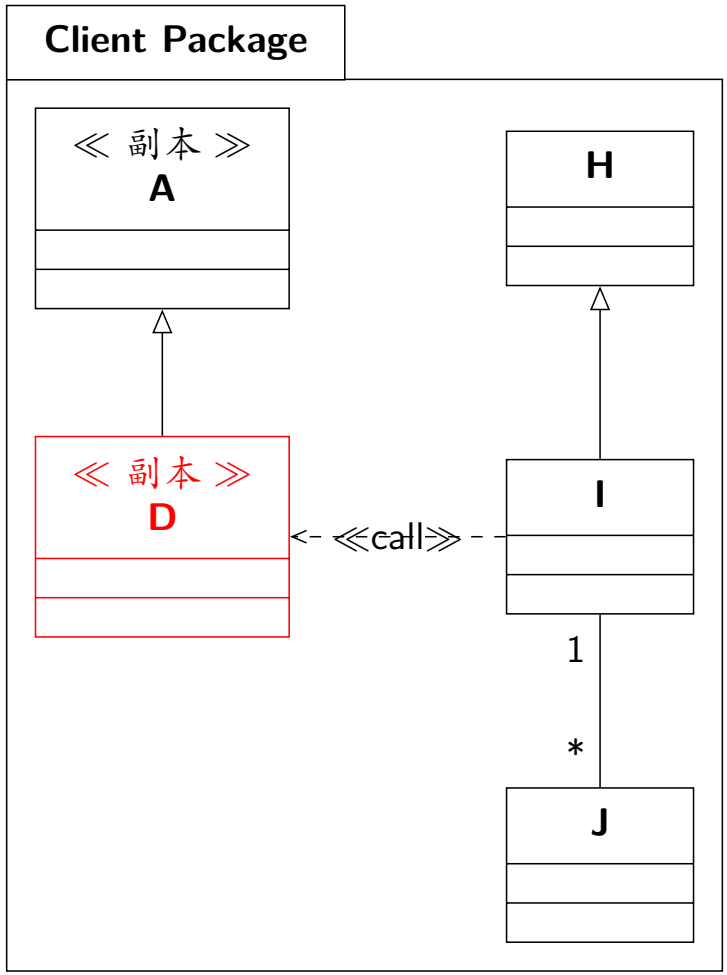
例：一个集中式类图

确定系统分布方案



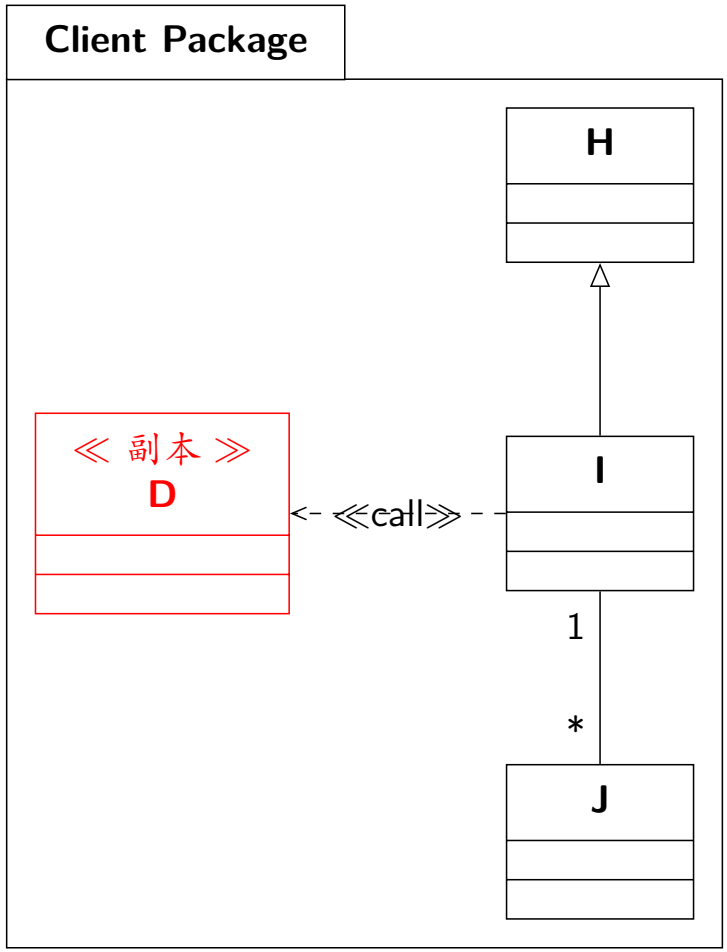
例：服务器包

确定系统分布方案



例：客户机包（完整类图）

确定系统分布方案



例：客户机包（部分类图）

识别控制流

1. 以结点为单位识别控制流
   - 不同结点上程序的并发问题已经解决
   - 考虑在每个结点上运行的程序还需要如何并发
2. 从用户需求出发认识控制流
   - 有哪些任务必须在同一台计算机上并发执行
3. 从用况认识控制流关注描述如下三类功能的用况
   - 要求与其他功能同时执行的功能
   - 用户随时要求执行的功能
   - 处理系统异常事件功能

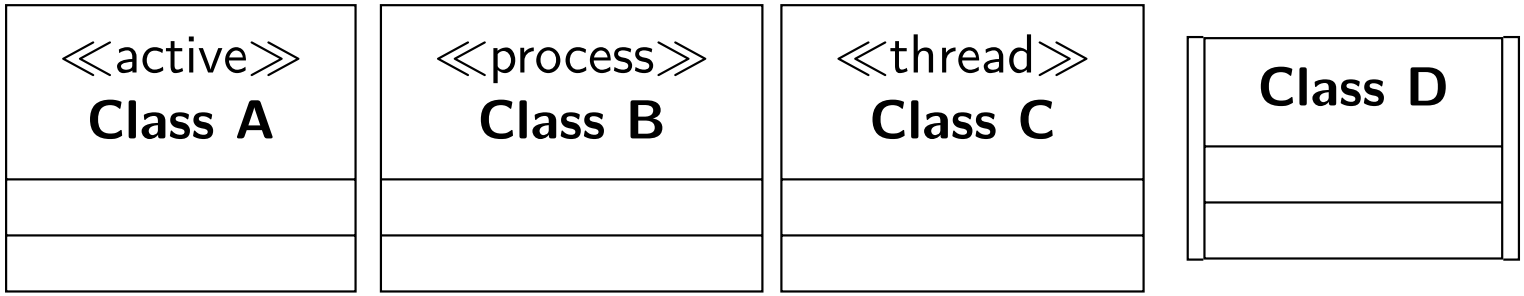
识别控制流

4. 参照 OOA 模型中的主动对象
5. 为改善性能而增设的控制流
   - 高优先级任务
   - 低优先级任务
   - 紧急任务
6. 实现并行计算的控制流（线程/进程）
7. 实现结点之间通讯的控制流（进程）
8. 对其它控制流进行协调的控制流

用主动对象表示控制流

控制流

是主动对象中一个主动操作的一次执行。其间可能要调用其他对象的操作，后者又可能调用另外一些对象的操作，这就是一个控制流的运行轨迹。



UML1 和 UML2 中的主动类表示法

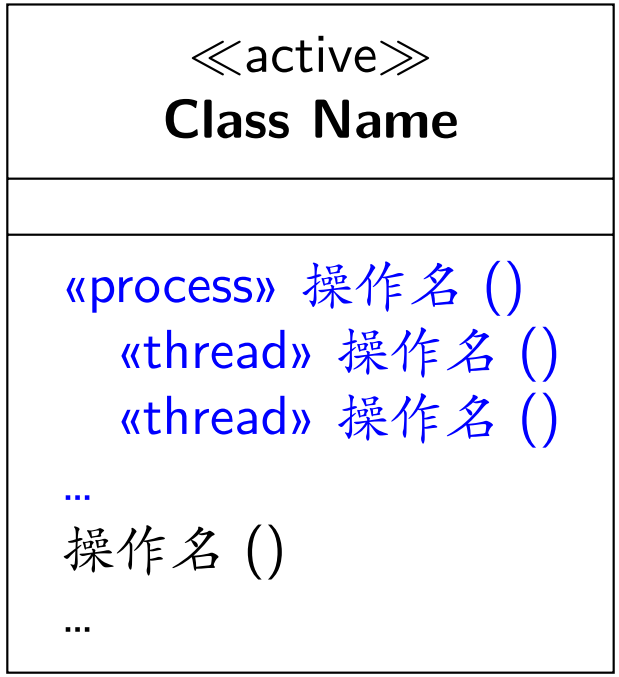
用主动对象表示控制流

问题：

一个主动类可以有多个主动操作和若干被动操作，UML 的表示法如何显式地表示哪个（哪些）操作是主动操作？

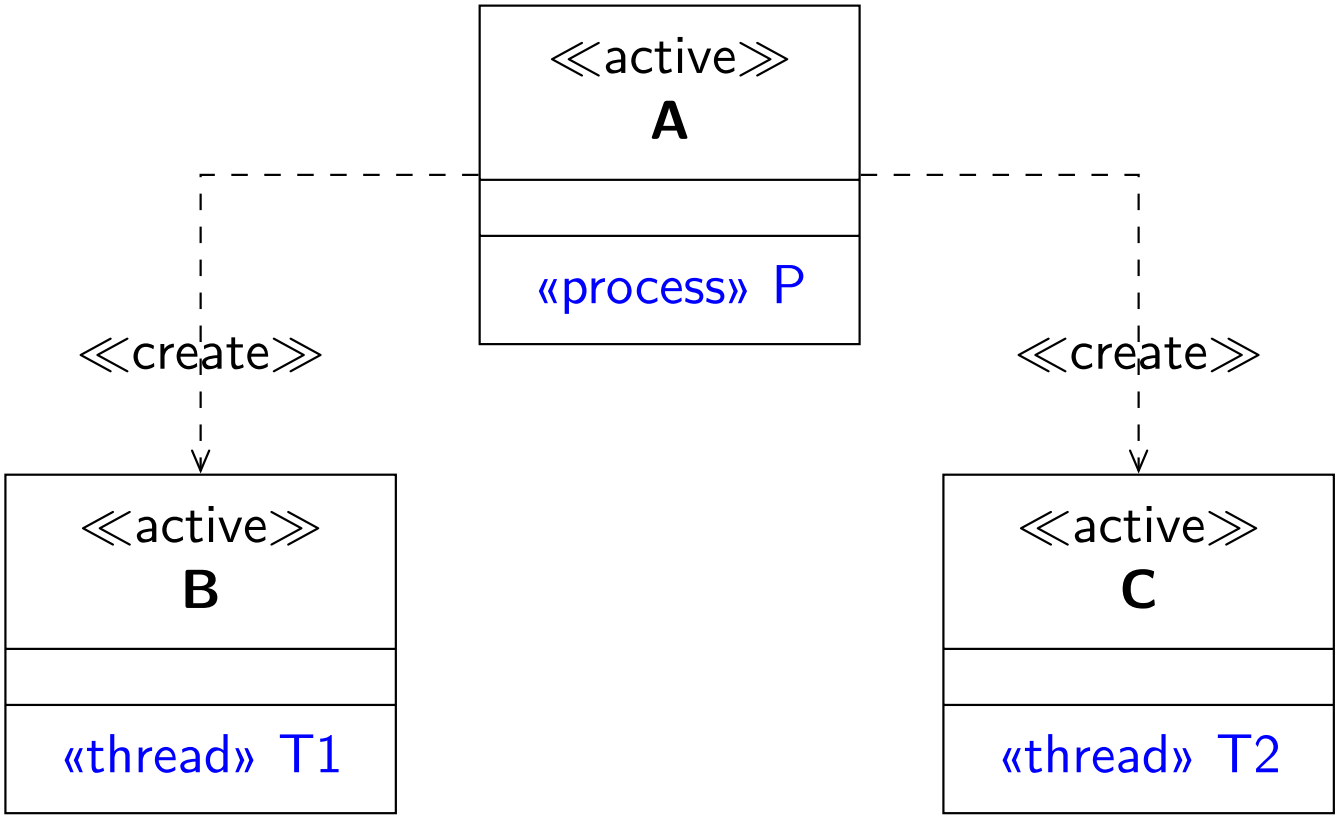
用主动对象表示控制流

用关键词表示主动操作

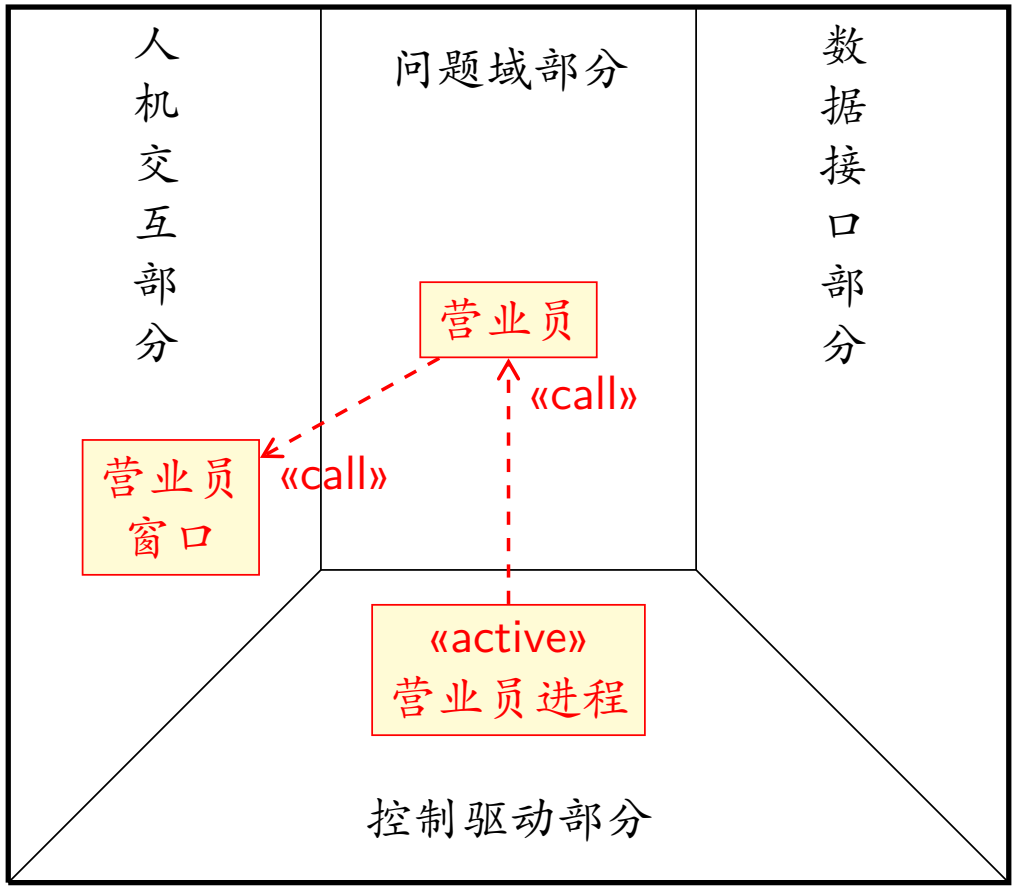


用主动对象表示控制流

显式地表示由进程创建线程

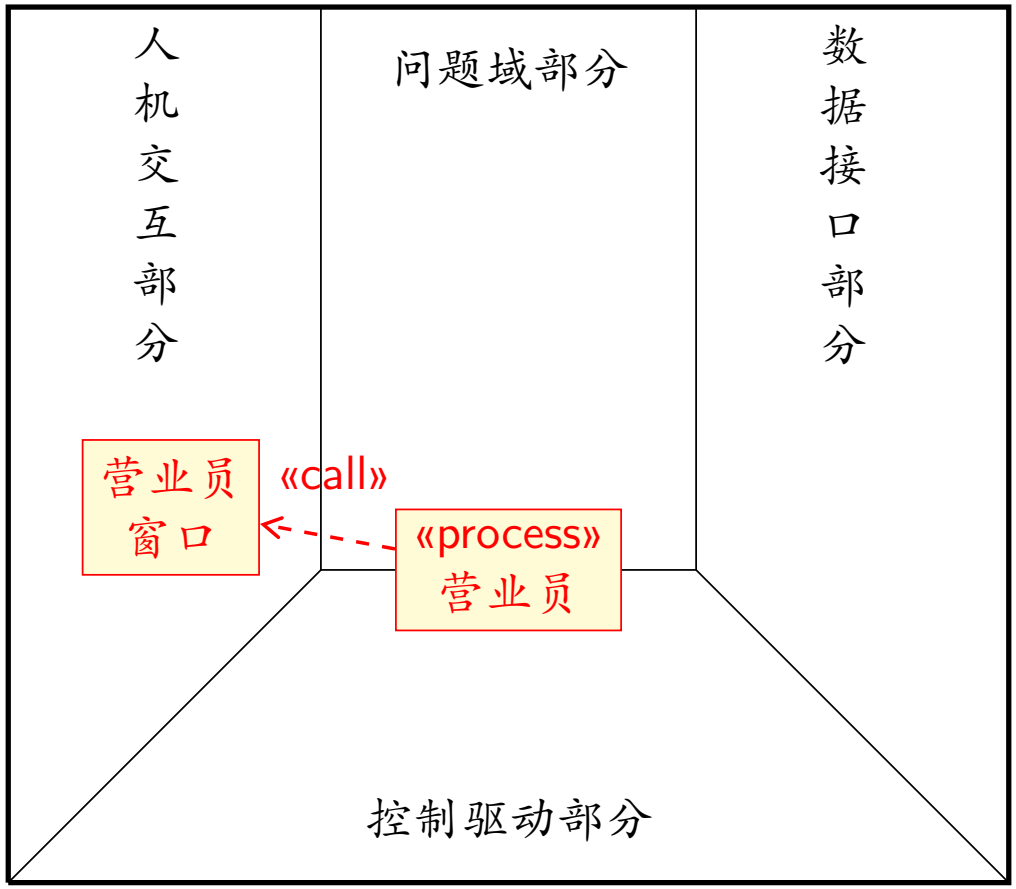


主动对象在 OOD 模型中的位置



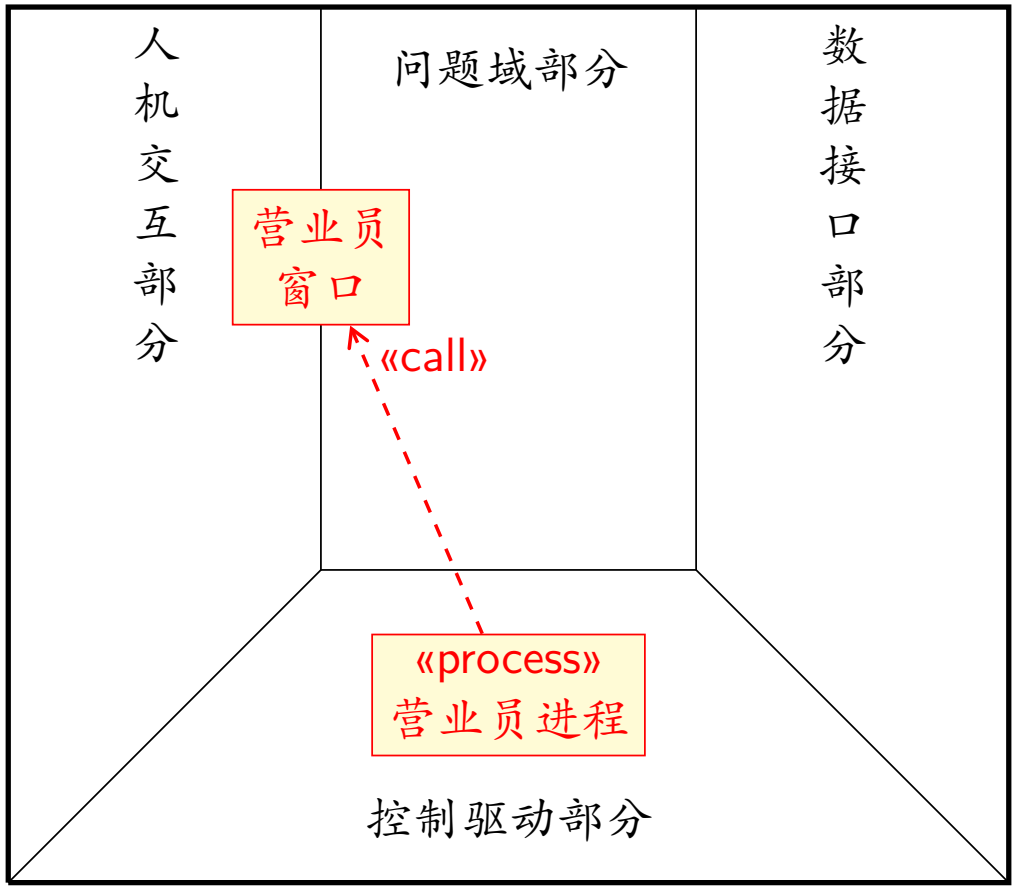
订单系统中营业员对象：无交叉方案

主动对象在 OOD 模型中的位置



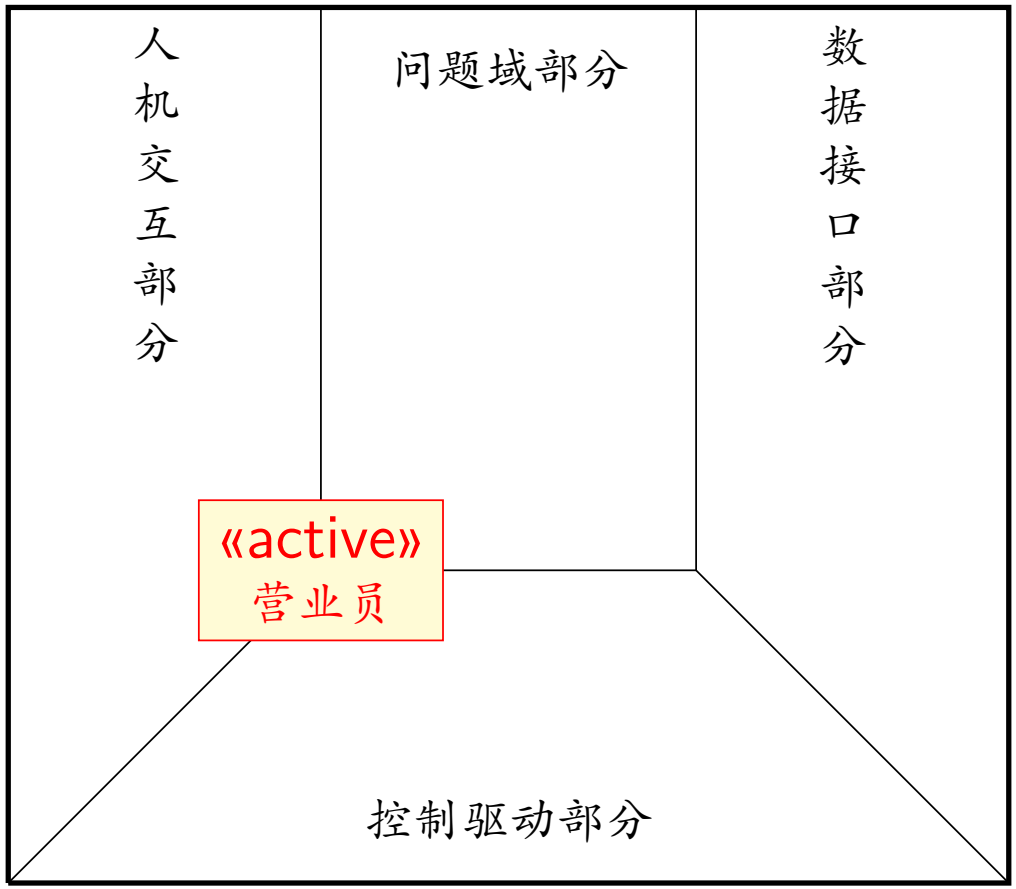
订单系统中营业员对象：问题域和控制驱动部分交叉

主动对象在 OOD 模型中的位置



订单系统中营业员对象：问题域和人机交互部分交叉

主动对象在 OOD 模型中的位置



问题域、人机交互、控制驱动部分都交叉

内容提要

什么是数据接口部分

数据接口部分

是 OOD 模型中负责与具体的数据管理系统衔接的外围组成部分，它为系统中需要长久存储的对象（持久对象）提供了在选定的数据管理系统中进行数据 存储与恢复的功能

问题范围：

- OO 系统中数据的存储表现为对象存储
- 只讨论对象在永久性存储介质上的存储
- 只须存储对象的属性部分
- 可能只有一部分对象需要长久存储

什么是数据接口部分

数据管理系统：实现数据的存储、检索、管理与维护

- 不同的数据管理系统各有不同的数据定义和操纵方式
  - 文件系统
  - 数据库：关系型、面向对象
- 针对不同的数据管理系统，需要设计一些专门处理其它对象的持久存储问题的对象，组成独立的数据接口部分
- 数据接口部分集中解决存储问题，可隔离数据管理系统对其它部分的影响

文件系统

优点:

- √ 操作系统和编程语言直接提供相关接口
- √ 廉价，容易学习和掌握，对数据类型没有限制

缺点:

- 功能贫乏、低级
- 不容易体现数据之间的关系
- 不易按地址或者记录读写
- 不能按属性进行数据检索与更新
- 缺少数据完整性支持
- 数据共享支持薄弱

数据库管理系统

数据库

长期存储在计算机内、有组织、可共享的数据集合。其中的数据按一定的数据模型组织、描述和储存，具有较小的冗余度、较高的数据独立性和易扩展性，并可为各种用户共享

- 按数据模型可分为：网状型、层次型、关系型、对象型等

数据库管理系统（DBMS）

用于建立、使用和维护数据库的软件。它对数据库进行统一的管理和控制，以保证数据库的安全性和完整性

关系数据库管理系统 RDBMS

关系模型
给定一组域 $D_1, D_2, \ldots, D_n$
其笛卡尔积 $D_1 \times D_2 \times \ldots \times D_n$ 的一个子集就是一个关系，又称二维表

数据的组织：用二维表组织各类数据
- 既可存放描述实体自身特征的数据
- 也可存放描述实体之间关系的数据
- 每一列称作一个属性，每一行称作一个元组

关系数据库管理系统 RDBMS

关系数据库术语对照

数据库专业术语	开发者的习惯术语	用户习惯术语
关系 relation	文件 file	表 table
元组 tuple	记录 record	行 row
属性 attribute	字段，域 field	列 column

面向对象数据库管理系统 OODBMS

RDBMS 不能直接、有效地组织和存储对象数据，需要对数据模式进行转换，并提供相应的接口，因此出现了面向对象数据库管理系统—OODBMS

OODBMS 的特征:

- 是面向对象的，支持对象、对象标识、对象的属性与操作、封装、继承、聚合、关联、多态等 OO 概念
- 具有数据库管理系统的功能
- 通常和面向对象程序设计语言配合良好

面向对象数据库管理系统 OODBMS

OODBMS 现状:

- 概念非常有吸引力
- 技术和产品逐渐完善
- 最大问题: 和现有主流数据库工具的兼容和集成

产品	支持语言	SQL 支持	许可证
Caché	ObjectScript, Java/.NET	子集	商业
Db4o	C#, Java	db4o-sql	GPL, 商业
ObjectDB	Java	JPA/JDO	商业
WakandaDB	JavaScript, C++	无	AGPL, 商业
ZODB	Python	无	Zope

数据管理系统的选择

非技术因素:
与项目的成本、工期、风险、宏观计划有关的问题

- 产品的成熟性，先进性，支持度
- 价格
- 开发队伍的技术背景
- 与其它系统的关系

技术因素:
考虑各种数据管理系统适应哪些情况，不适应哪些情况

数据管理系统的选择

文件系统的适应性:

优点 可存储任何类型的数据，包括具有复杂内部结构的数据和图形、图象、视频、音频等多媒体数据

缺点 操作低级；数据操纵功能贫乏；缺少数据完整性支持；缺少数据共享、故障恢复、事务处理等功能

适应 数据类型复杂，但对数据存取、数据共享、数据完整性维护、故障恢复、事务处理等功能要求不高的应用系统

不适应 数据操纵复杂、多样，数据共享及数据完整性维护要求较高的系统

数据管理系统的选择

关系数据库管理系统的适应性:

优点 对数据存取、数据共享、数据完整性维护、故障恢复、事务处理等功能提供强有力的支持

适应 对数据处理功能要求较高的应用系统，以及需大量保存和管理各类实体之间关系信息的应用系统

缺点 关系数据模型对数据模式的限制较多，当对象的内部结构较为复杂时，就不能直接地与关系数据库的数据模式相匹配，需要经过转换

不适合 图形、图象、音频、视频等多媒体数据

数据管理系统的选择

对象数据库管理系统的适应性:

- 从纯技术的角度看，对用 OO 方法开发的系统采用 OODBMS 是最合理的选择，几乎没有不适合的情况。
- 如果某些项目不适合，主要是由于非技术因素，而不是技术因素
- 各种 OODBMS 采用的对象模型多少有些差异，与用户选用的 OOA & D 方法及 OOPL 中的匹配程度不尽一致，功能也各有区别，对不同的应用系统有不同的适应性

对象存储方案和数据接口的设计策略

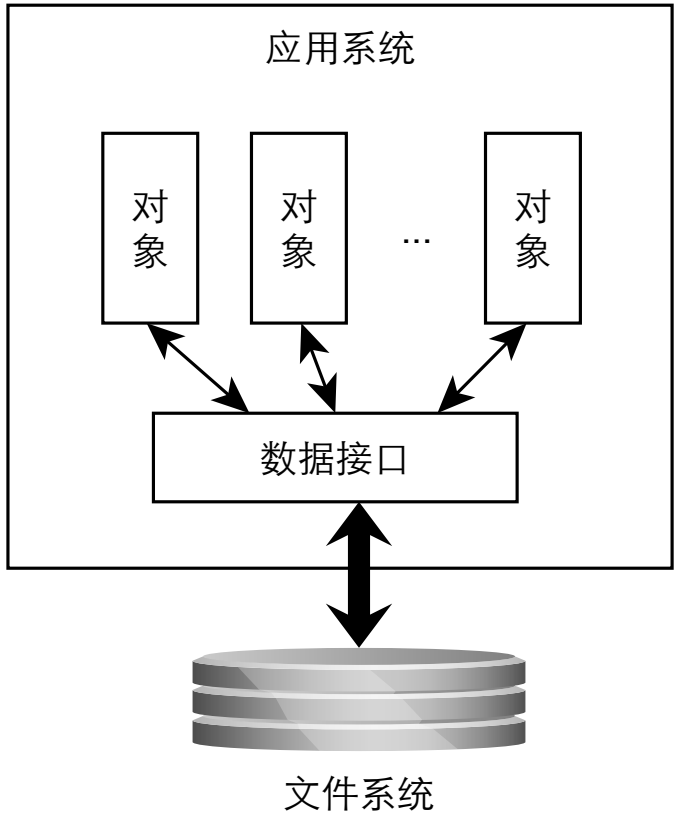
针对三种数据管理系统分别设计

- 对象存储方案 $\Longrightarrow$
  如何把对象映射到数据管理系统?
- 数据接口部分的设计策略 $\Longrightarrow$
  如何设计数据接口部分的对象类?
- 问题域部分 $\Longrightarrow$
  如何对问题域部分做必要的修改?

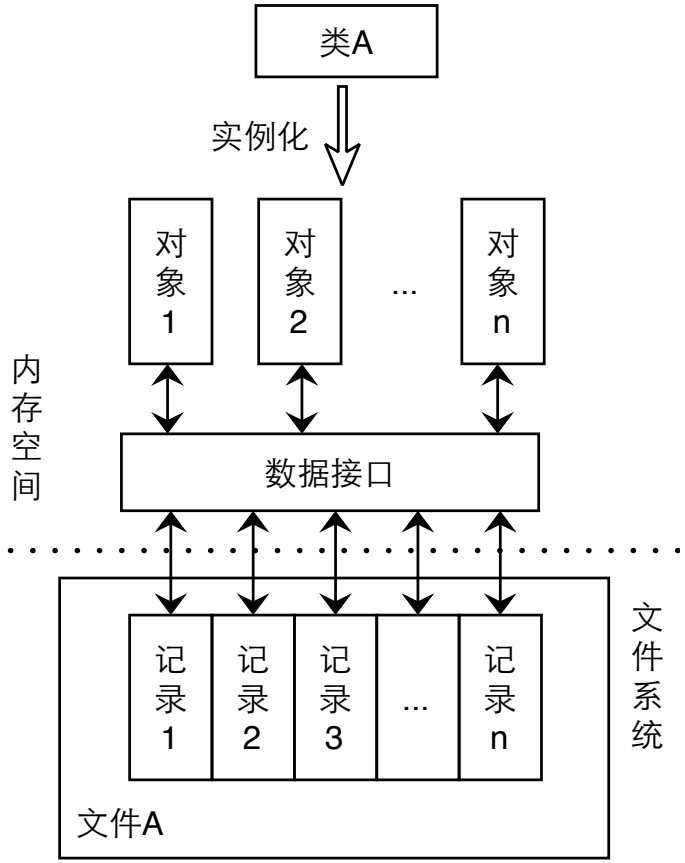
如何看待用文件存储对象

对象在内存空间和文件空间的映像

- 应用系统仍然是面向对象的
- 只是用文件系统存储对象的数据

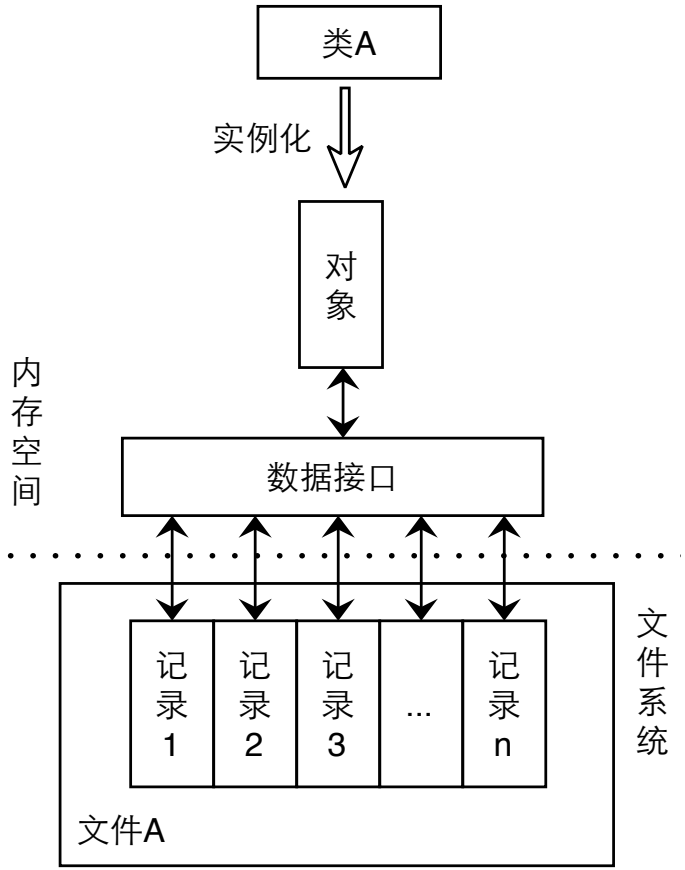


从应用对象到文件记录的不同映射方式



一一对应的映射方式

从应用对象到文件记录的不同映射方式



非一一对应的映射方式

对象在文件中的存放策略

基本策略:

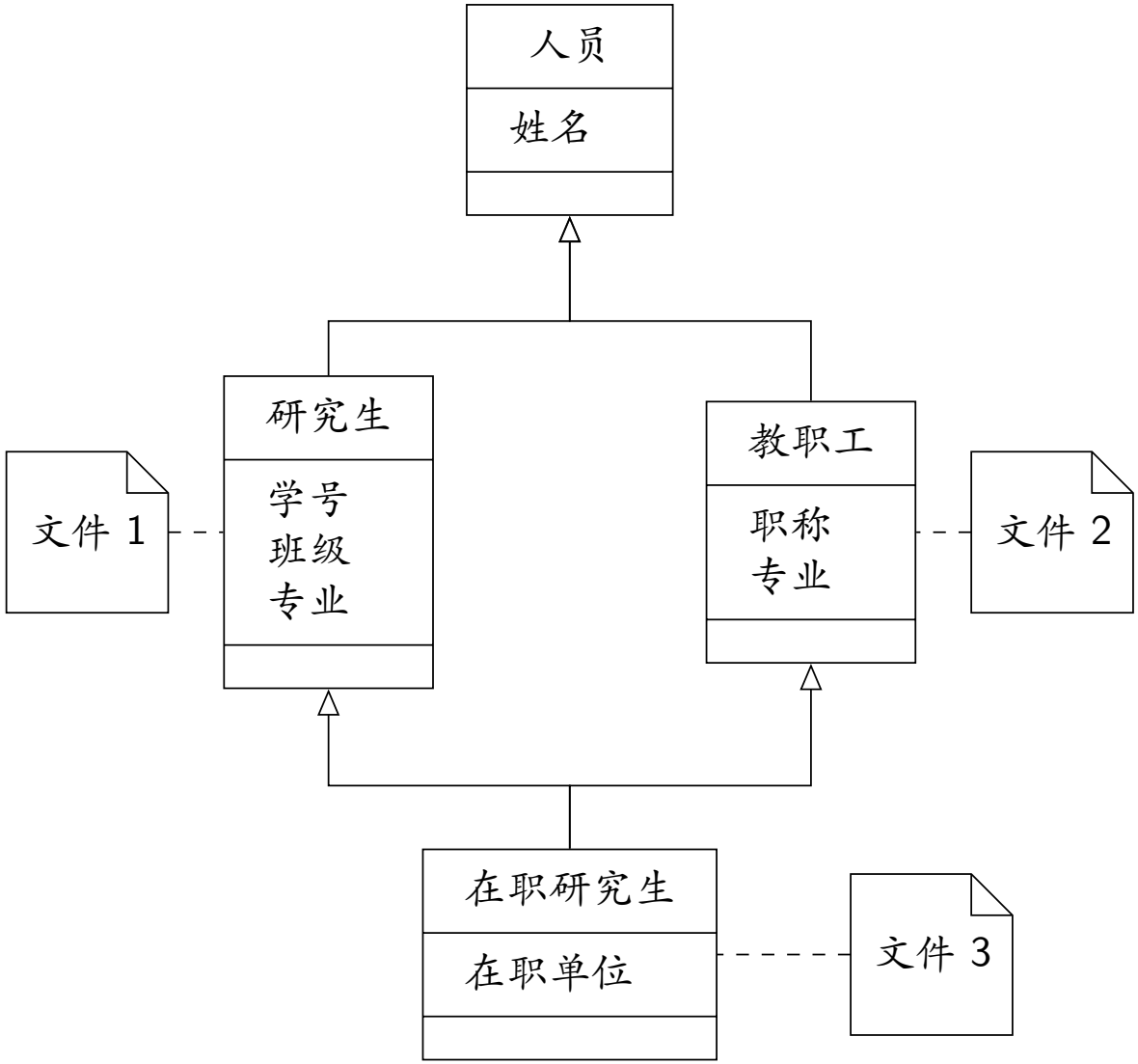
把由每个类直接定义、需要持久存储的全部对象实例存放在一个文件中；每个对象实例的全部属性作为一个存储单元，占用该文件的一个记录

一般类和特殊类:

每个类各自使用不同的文件，分别存放各自直接创建的对象实例

对象在文件中的存放策略

例:



对象在文件中的存放策略

另一种策略:

一个一般—特殊结构用一个文件，结构中各个类定义的所有对象实例都存放在一个文件中

缺点:

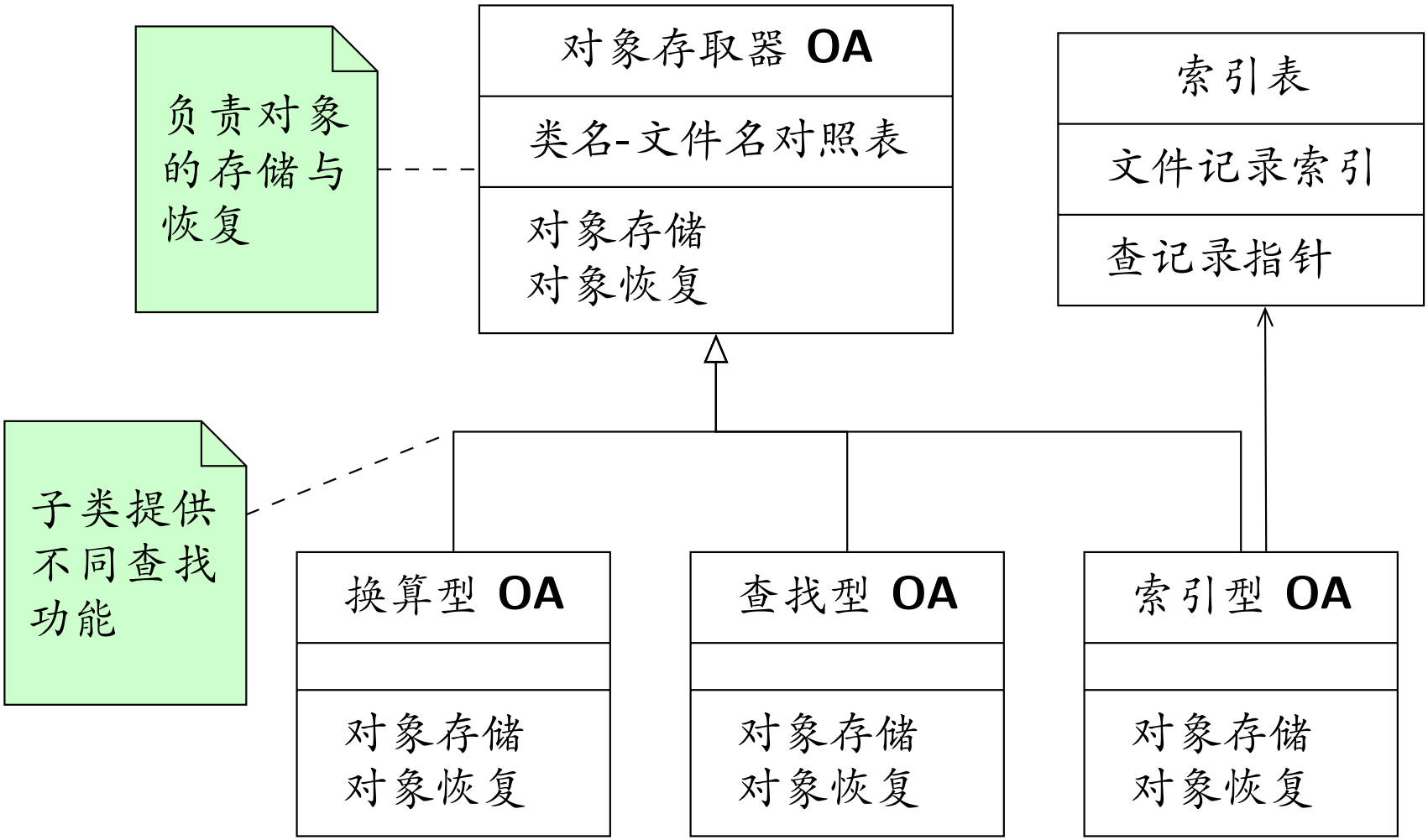
- 浪费空间
- 模糊了对象分类关系
- 使操作复杂化

对象在文件中的存放策略

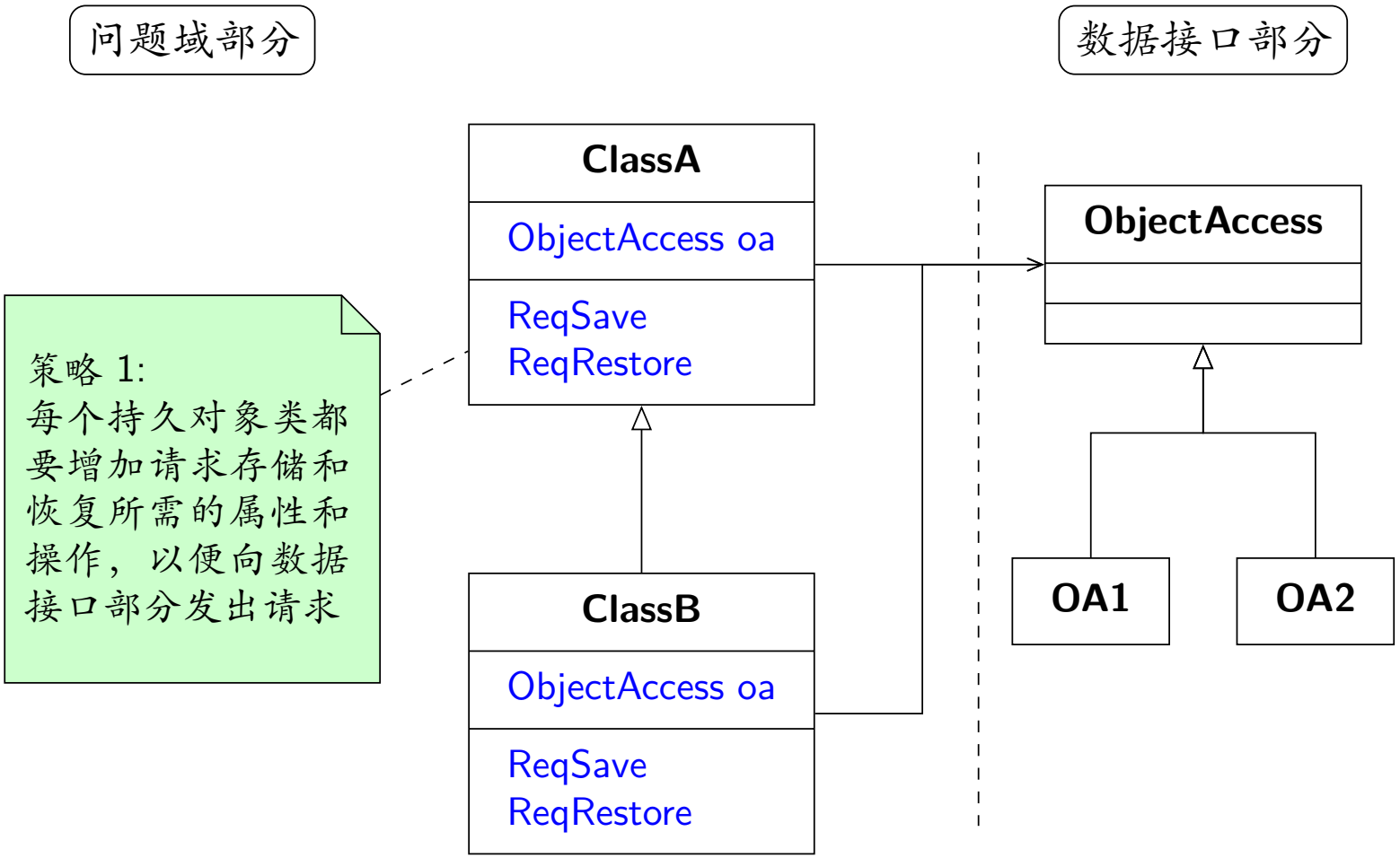
提高检索效率: 在对象和文件记录间建立有规律的映射关系

- 对象名或关键字呈线性规律
  - 按对象名或关键字的顺序形成文件记录
  - 给出对象名称或关键字，快速地计算出它的存放位置
- 对象名称或关键字可以比较和排序
  - 按关键字顺序安排记录，检索时采用折半查找法
  - 建立按对象名称或者按关键字排序的索引表，通过该表中的记录指针找到相应的记录
- 其他措施: 如散列表、倒排表、二叉排序树等等

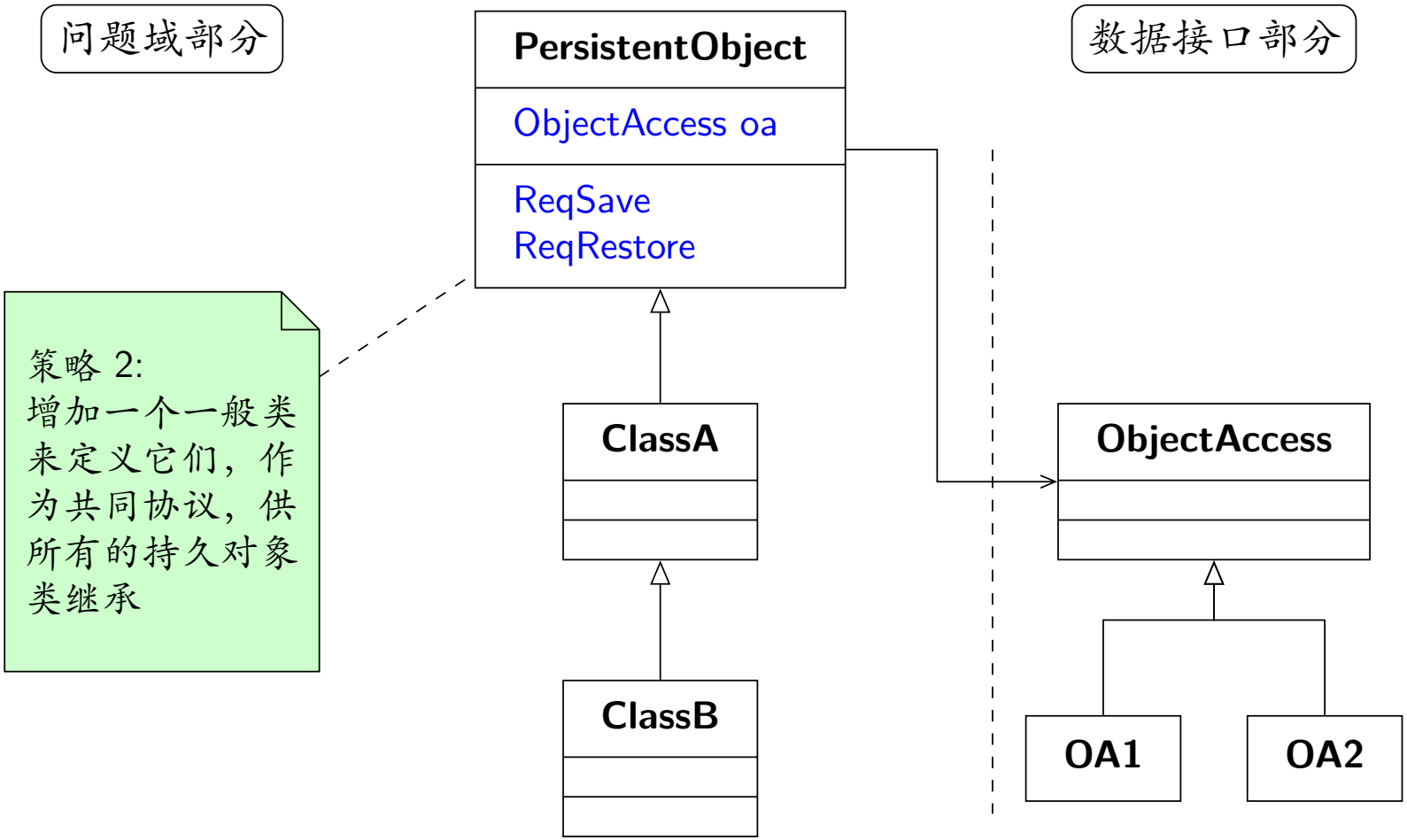
设计数据接口部分的对象类



问题域部分的修改


问题域部分
数据接口部分
ClassA
ObjectAccess oa
ReqSave
ReqRestore
ObjectAccess
策略 1:
每个持久对象类都要增加请求存储和恢复所需的属性和操作，以便向数据接口部分发出请求
ClassB
ObjectAccess oa
ReqSave
ReqRestore
OA1
OA2

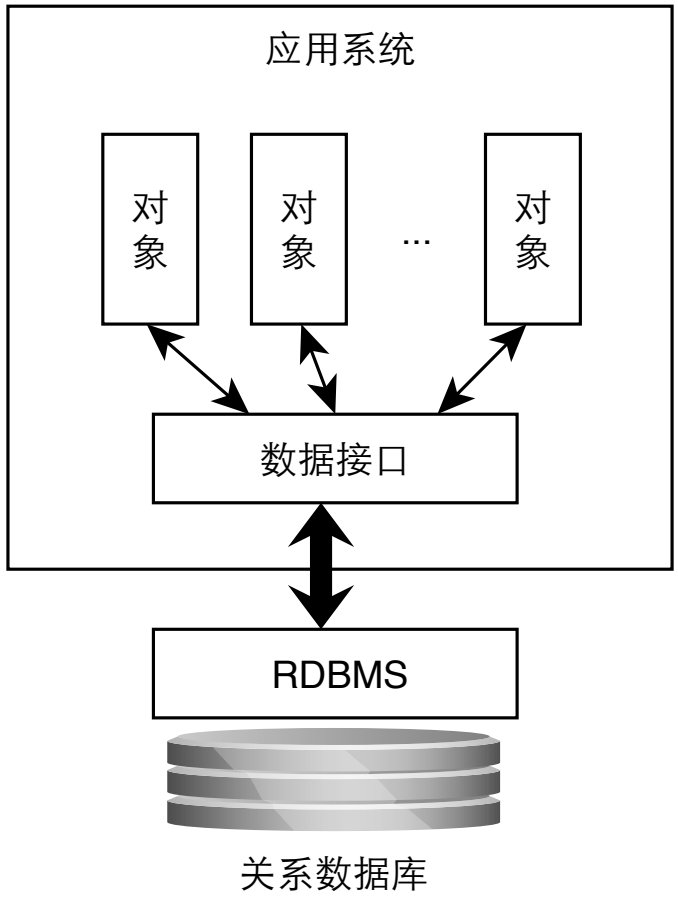
问题域部分的修改


问题域部分
PersistentObject
ObjectAccess oa
ReqSave
ReqRestore
数据接口部分
策略 2:
增加一个一般类来定义它们，作为共同协议，供所有的持久对象类继承
ClassA
ClassB
ObjectAccess
OA1
OA2

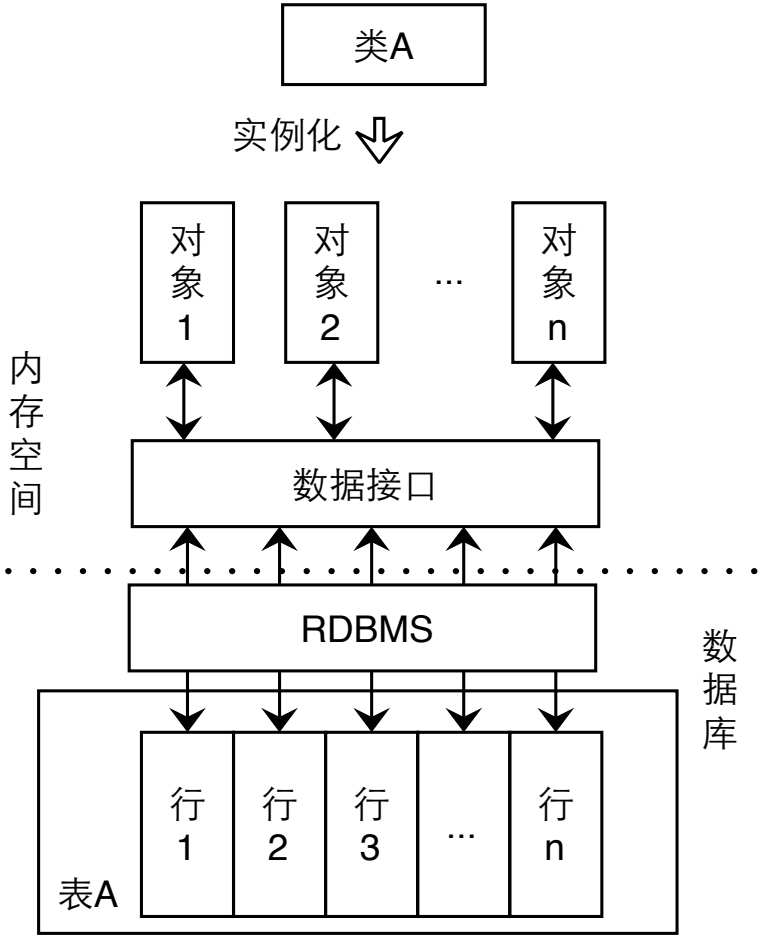
如何看待用 RDBMS 存储对象

对象在内存空间和数据库的映像

- 应用系统仍然是面向对象的
- 只是用关系数据库存储对象的数据

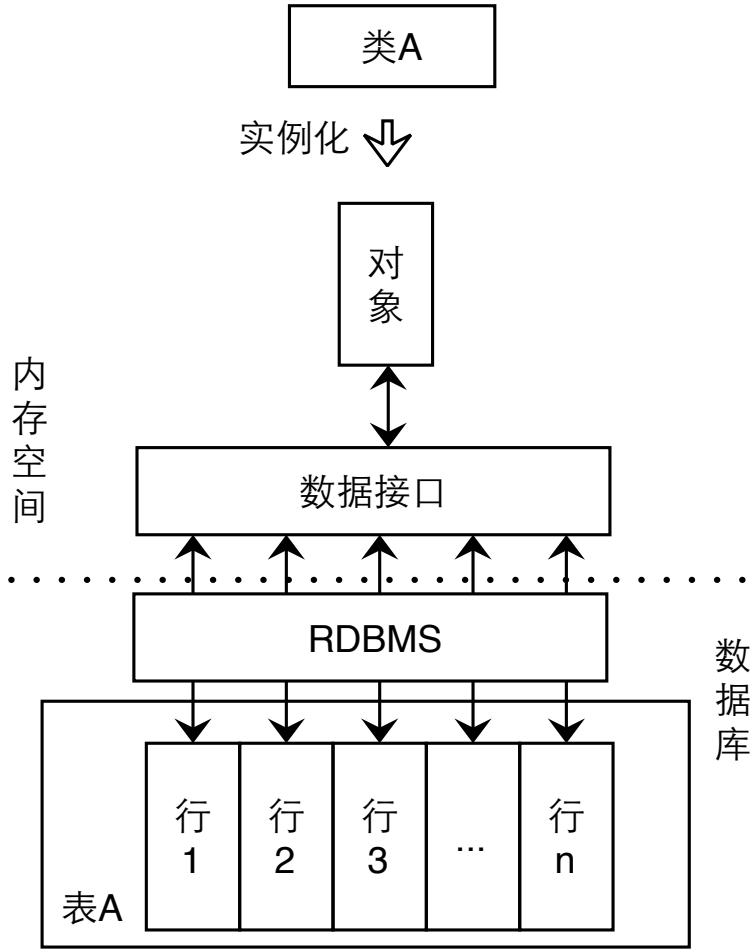


从应用对象到数据库表元组的不同映射方式



一一对应的映射方式

从应用对象到数据库表元组的不同映射方式



非一一对应的映射方式

使用 RDBMS 与文件系统的比较

1. 系统以不同方式使用数据库中的数据
   - 把数据库中已有的数据描述为本系统中的类和对象
   - 系统中的对象直接使用数据库中的普通数据，二者之间不存在直接映射关系，只是一种简单的使用关系
2. 为了满足关系数据库对规范化的要求，可能需要数据格式的转换
   - 只限于一个类的范围的情况
   - 牵涉到多个类的情况

对象在数据库中的存放策略

- 对象数据的规范化
- 修改类图
- 确定关键字
- 从类图映射到数据库表
  - 类 → 表
  - 类的属性 → 表的属性
  - 对象实例 → 行
  - 对一般-特殊结构、整体-部分结构、关联等 OO 概念的处理

对象数据的规范化

关系数据库要求存入其中的数据符合一定的规范，并且用范式衡量规范化程度的高低

第一范式（1NF）：属性是原子的，不可再分

不符合第一范式的例子：

姓名	电话	
张三	18900010002	62750114

对象数据的规范化

第二范式 (2NF)：非关键属性完全依赖整个关键字，即不能依赖关键属性的一部分，意味着一个表只描述一个事物

不符合第二范式的例子：

学号	姓名	年龄	课程	成绩	学分
1010	张三	19	英语	88	4

对象数据的规范化

第二范式 (2NF)：非关键属性完全依赖整个关键字，即不能依赖关键属性的一部分，意味着一个表只描述一个事物

不符合第二范式的例子：

学号	姓名	年龄	课程	成绩	学分
1010	张三	19	英语	88	4

修正：

学号	姓名	年龄
1010	张三	19

课程	学分
英语	4

学号	课程	成绩
1010	英语	88

对象数据的规范化

第三范式 (3NF)：不存在非关键属性对任一候选关键属性的传递依赖，即属性不依赖于其它非关键属性

不符合第三范式的例子：

学号	姓名	年龄	所在学院	学院地点	学院电话
1010	张三	19	信科	理科 2 号楼	62751760

对象数据的规范化

第三范式 (3NF)：不存在非关键属性对任一候选关键属性的传递依赖，即属性不依赖于其它非关键属性

不符合第三范式的例子：

学号	姓名	年龄	所在学院	学院地点	学院电话
1010	张三	19	信科	理科 2 号楼	62751760

修正：

学号	姓名	年龄	学院
1010	张三	19	信科

学院	地点	电话
信科	理科 2 号楼	62751760

对象数据的规范化

Boyce-Codd 范式 (BCNF)：所有属性（包括关键属性）都不传递依赖于任何候选关键字

不满足 BCNF 范式的例子:
某仓库管理系统，一个管理员只在一个仓库工作；一个仓库可以存储多种物品

仓库 ID	管理员 ID	存储物品 ID	数量
10	张三	BMWX5	10

对象数据的规范化

第四范式：是第三范式，且表中不能包含一个实体的多个互相独立的值

不满足第四范式的例子及其规范化：

学号	课程	活动
100	音乐	游泳
100	会计	游泳
100	音乐	网球
100	会计	网球
100	音乐	桥牌
100	会计	桥牌

对象数据的规范化

第四范式：是第三范式，且表中不能包含一个实体的多个互相独立的值

不满足第四范式的例子及其规范化：

学号	课程	活动
100	音乐	游泳
100	会计	游泳
100	音乐	网球
100	会计	网球
100	音乐	桥牌
100	会计	桥牌

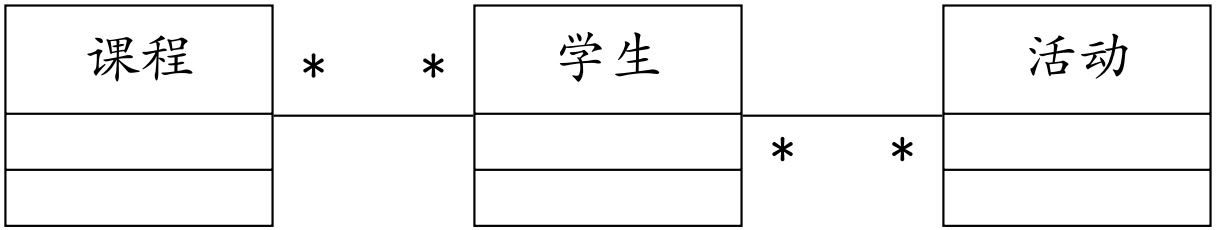
$\Rightarrow$

学号	课程
100	音乐
100	会计

学号	活动
100	游泳
100	网球
100	桥牌

对象数据的规范化

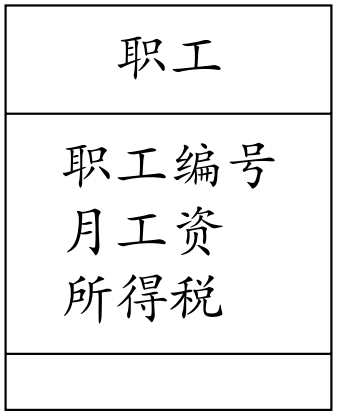
用面向对象方法得到的分类



化解多对多关联后

对象数据的规范化

面向对象虽然可以得到符合较高范式要求的数据库表，但面向对象方法并不总是能得到规范化的结果



对象数据的规范化

小结

- 未必规范化程度越高越好
  - 规范化可能影响系统的可理解性，另外增加了多表查询和连接操作
- 面向对象方法与关系数据库的规范化目标既有相违的一面，又有相符的一面
  - 对象的数据结构常常连 1NF 的要求都不能满足
  - 以对象为中心组织数据与操作，可能有助于达到第 2、3 等范式的要求

修改类图

规范化可能引发对类图的修改

1. 保持类图，对表规范化
   缺点是对象的存储与恢复必须经过数据格式的转换
2. 修改类图
   对问题域的映射可能不像规范化之前那么直接。但是这个问题并不严重 – 利大于弊
3. 确定关键字
   用较少的属性作为关键字将为含关键字的操作带来方便

修改类图

最终效果:
经过必要的规范化处理和关键字处理之后，得到一个符合数据库设计要求的类图，其中每个需要映射到数据库表的类，都满足如下条件:

- 至少满足第一范式
- 满足所期望的更高范式
- 有一组属性被确定为关键字

从类图到数据库的映射策略

- 对每个要在数据库中存储对象实例的类，都建立一个数据库表
- 类的每个属性（包括从所有祖先继承来的属性）都对应表的一个属性（列）
  - 名称、数据类型完全相同
  - 其中一组属性被确定为关键字
- 类的每个对象实例将对应表的一个元组（行）

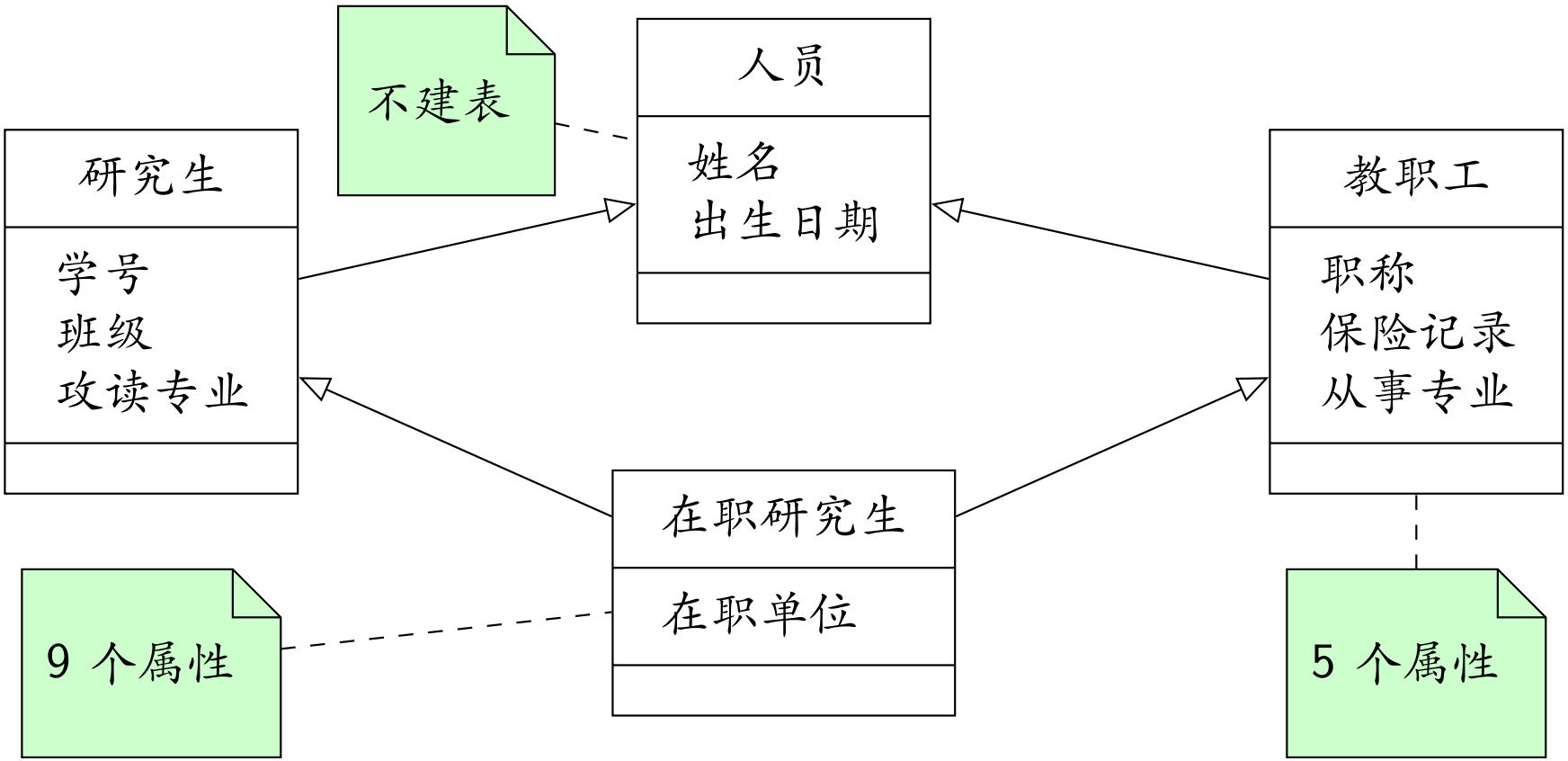
从类图到数据库映射中类图的处理

主要考虑：

- 一般–特殊结构
- 关联
- 整体–部分结构

对一般–特殊结构的处理

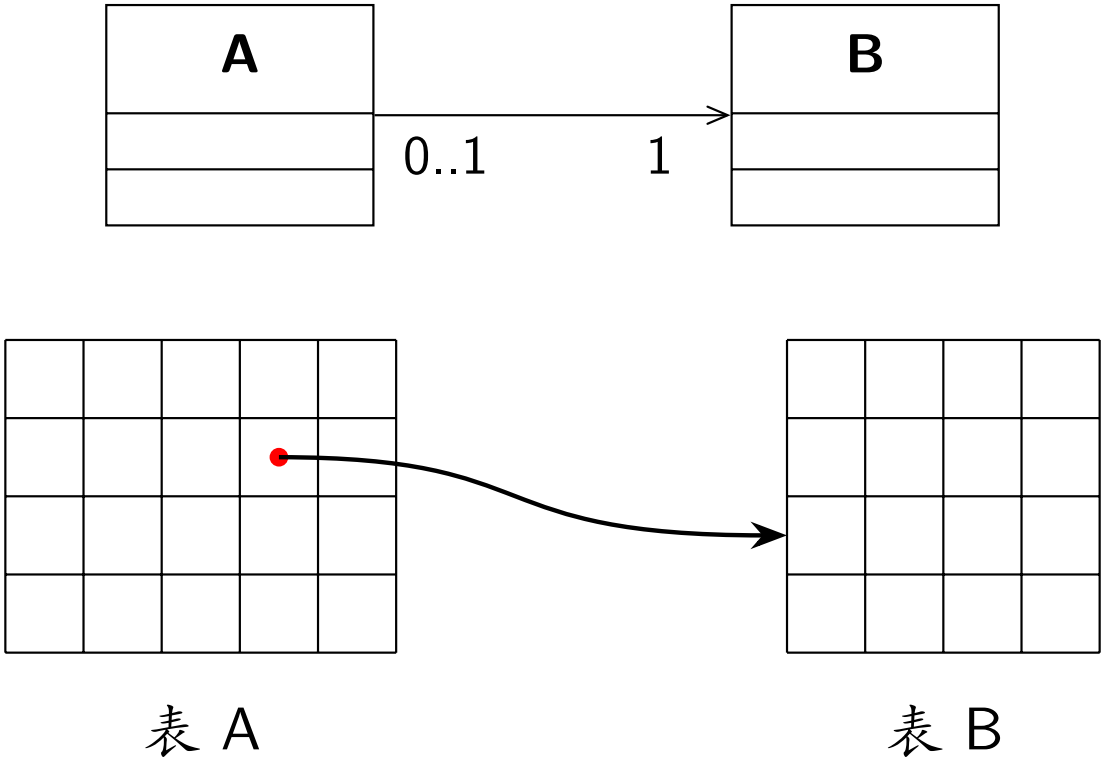
抽象类~~不对应~~数据库表
特殊类包括自己定义的和继承来的所有属性



对关联的处理

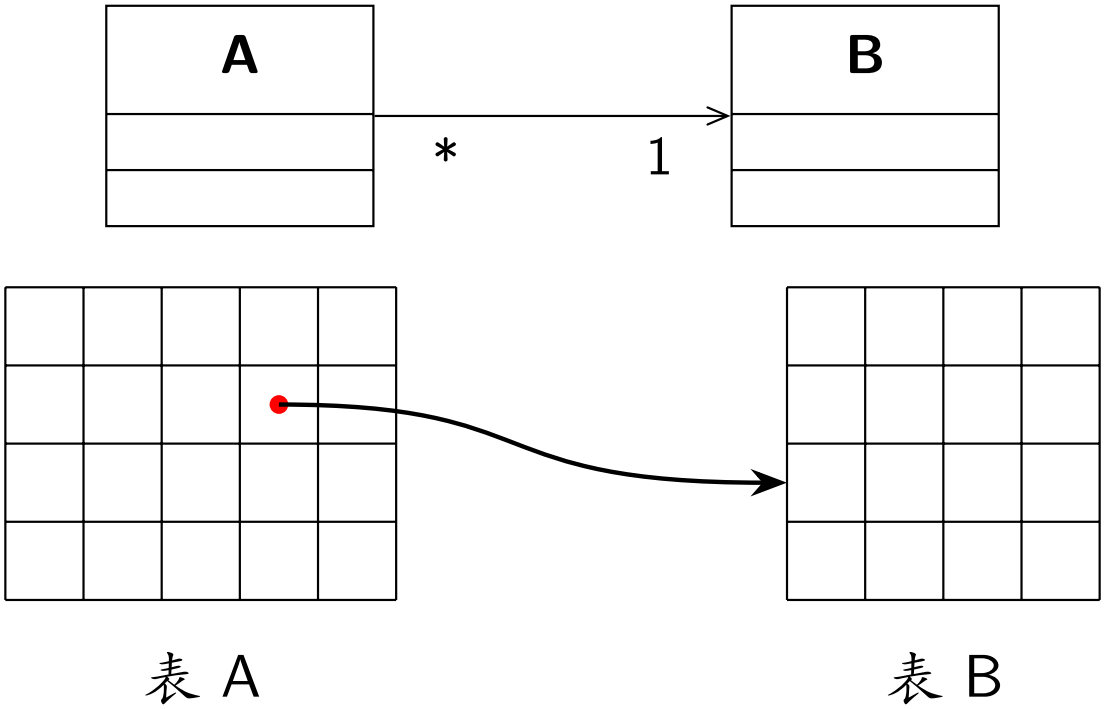
1. 在关联连接线一端的类中定义一个（或一组）属性，表明另一端类的哪个对象实例与本端的对象实例相关联
   - 该属性（属性组）应该和另一端的关键字相同
   - 如果另一端的关键字包含多个属性，本端也要定义同样的多个属性
2. 在对应的数据库表中，一个表以该属性（或属性组）作为 外键 ，另一个表以它作为主键，使前者的元组通过其属性值指向后者的元组

一对一的关联



一对多的关联

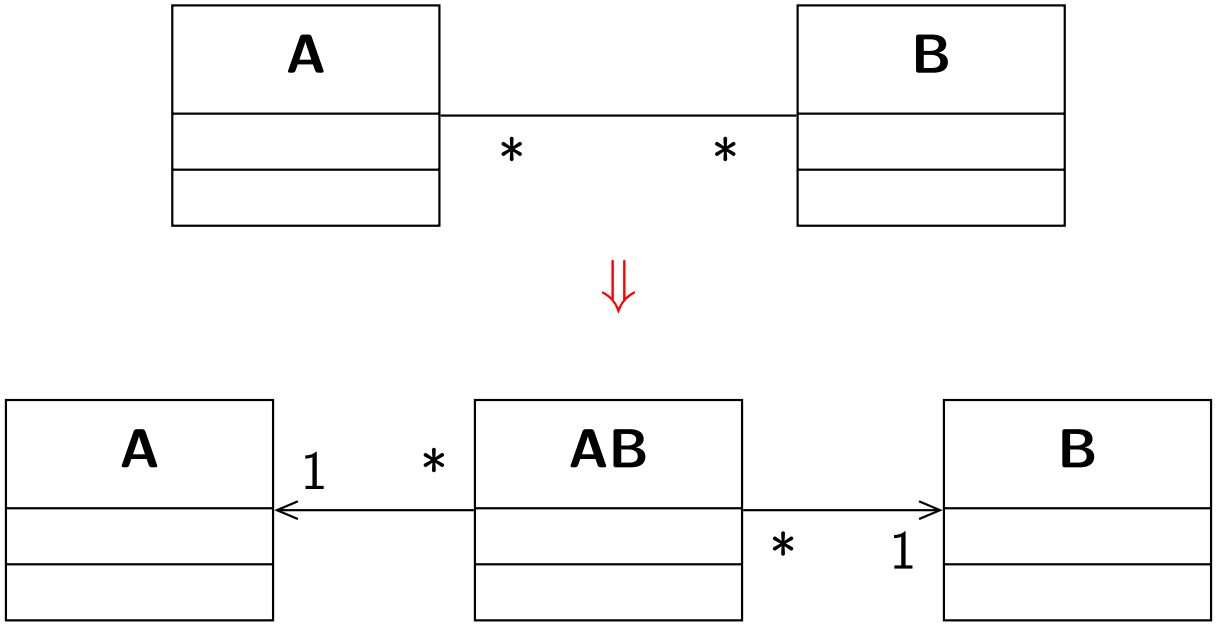
从多重性约束为 m 的一端指向多重性约束为 1 的一端



映射为数据库表后，A 表以 B 表的主键作为自己的外键

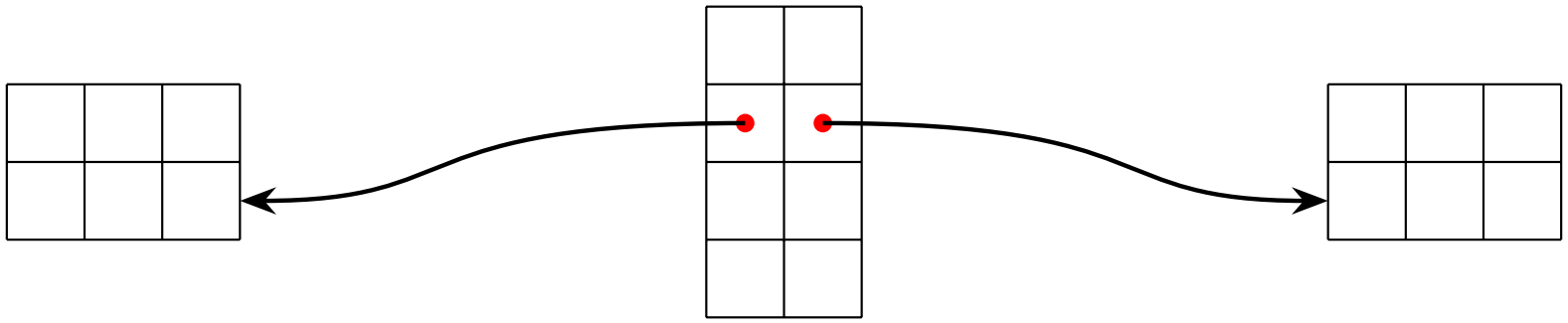
多对多的关联

先将多对多关联化为两个 1 对多的关联



多对多的关联

然后将每个类映射到一个数据库表



AB 表含有两个外键，一个是 A 的主键，一个是 B 的主键

多对多的关联

对象类转化为数据库表的几种情况：

- 表中只包含描述本类事物自身特征的属性
- 表中既包含描述本类事物自身特征的属性，也包含作为外键指向另一个表的元组的属性
- 表中只包含作为外键指向其它表的元组的属性

对整体-部分结构的处理

分为 紧密、固定的方式和 松散、灵活的方式

紧密、固定方式

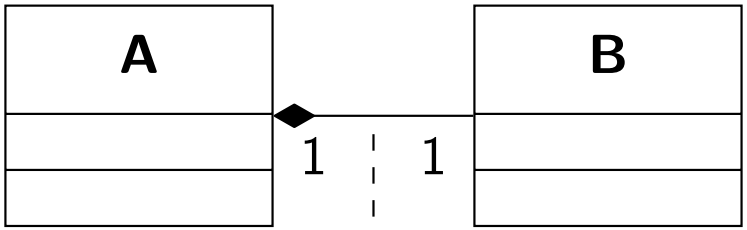
把部分对象类的属性合并到整体对象类中

松散、灵活方式

整体对象类和部分对象类分别建立一个表，通过外键表现整体-部分关系

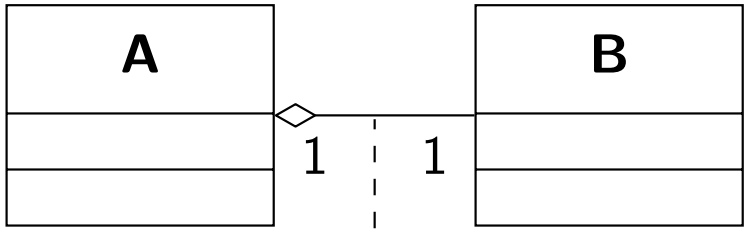
多重性约束对实现方式也有影响，紧密、固定的实现方式适合 1 对 1 的组合关系

对整体-部分结构的处理



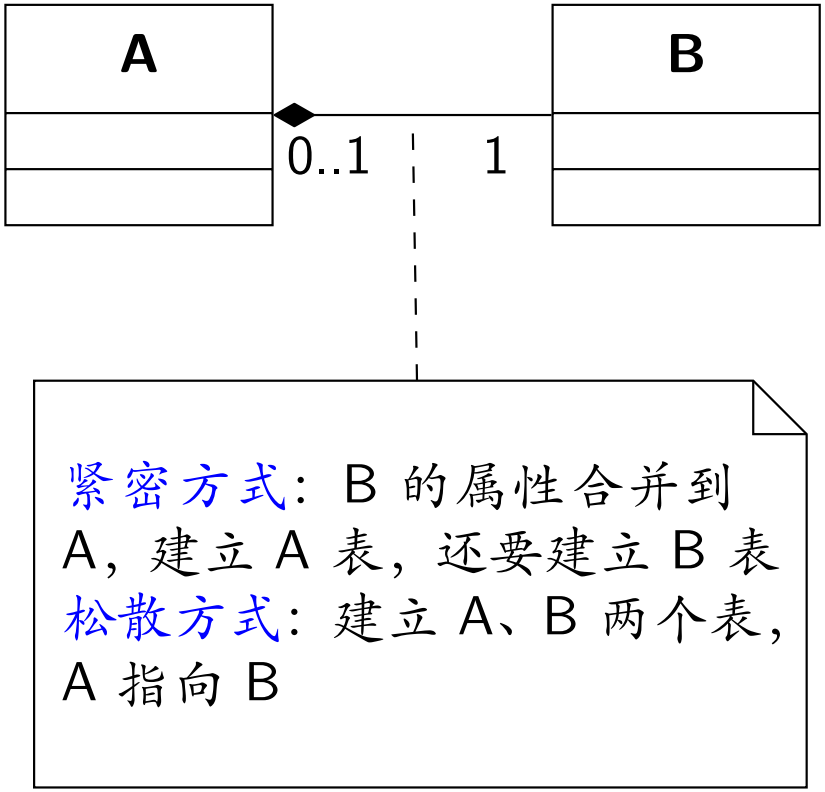
紧密方式：B 的属性合并到 A，建立 A 表

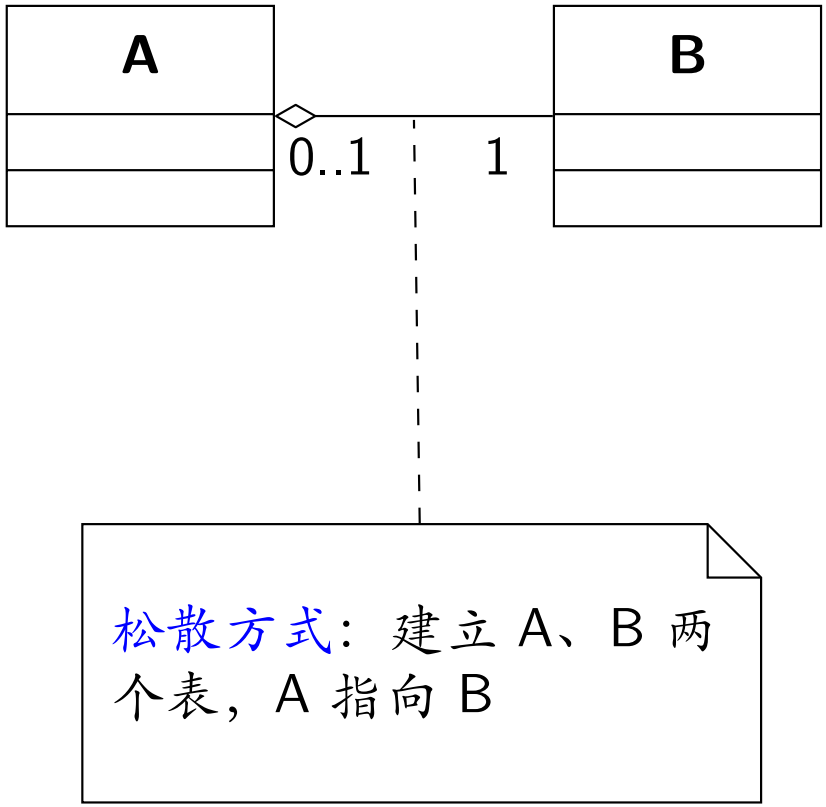
松散方式：建立 A、B 两个表，A 指向 B，或者 B 指向 A



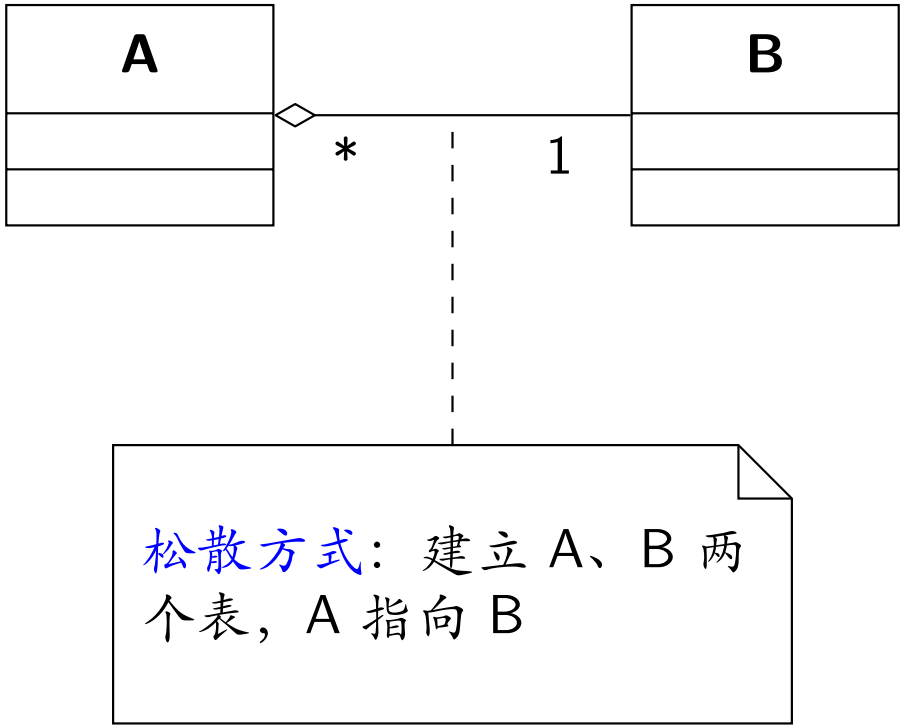
松散方式：建立 A、B 两个表，A 指向 B，或者 B 指向 A

对整体-部分结构的处理

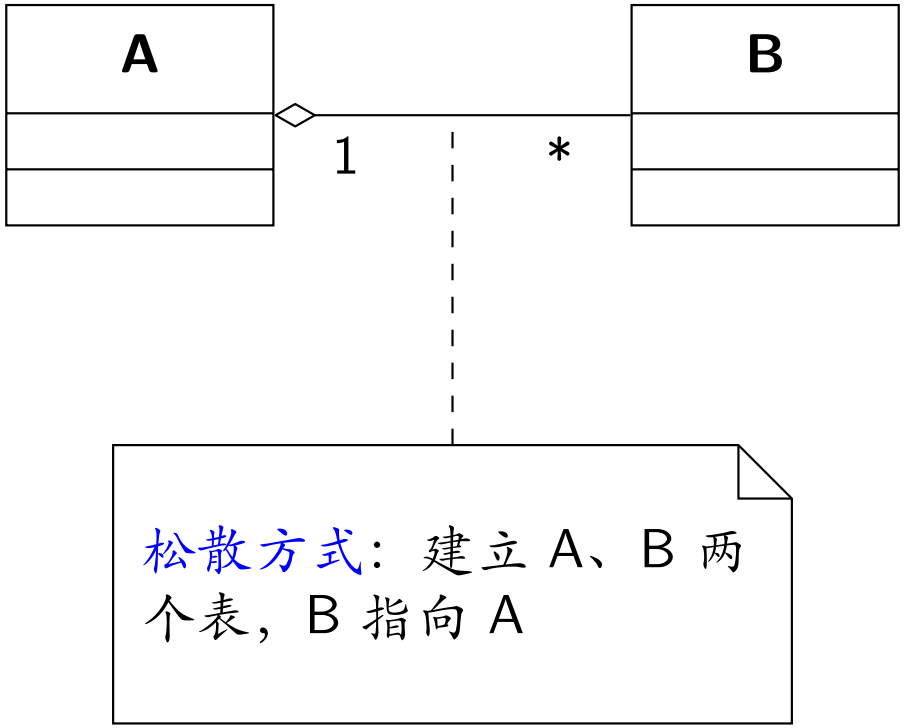

A
B
0..1
1
紧密方式：B 的属性合并到 A，建立 A 表，还要建立 B 表
松散方式：建立 A、B 两个表，A 指向 B


A
B
0..1
1
松散方式：建立 A、B 两个表，A 指向 B

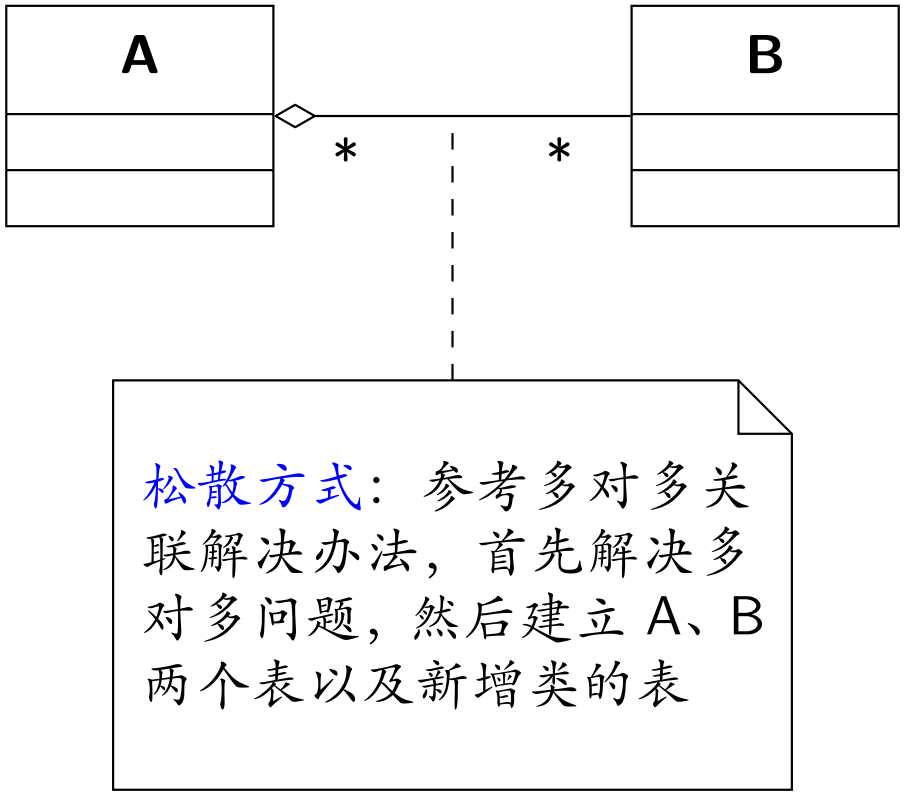
对整体-部分结构的处理



对整体-部分结构的处理

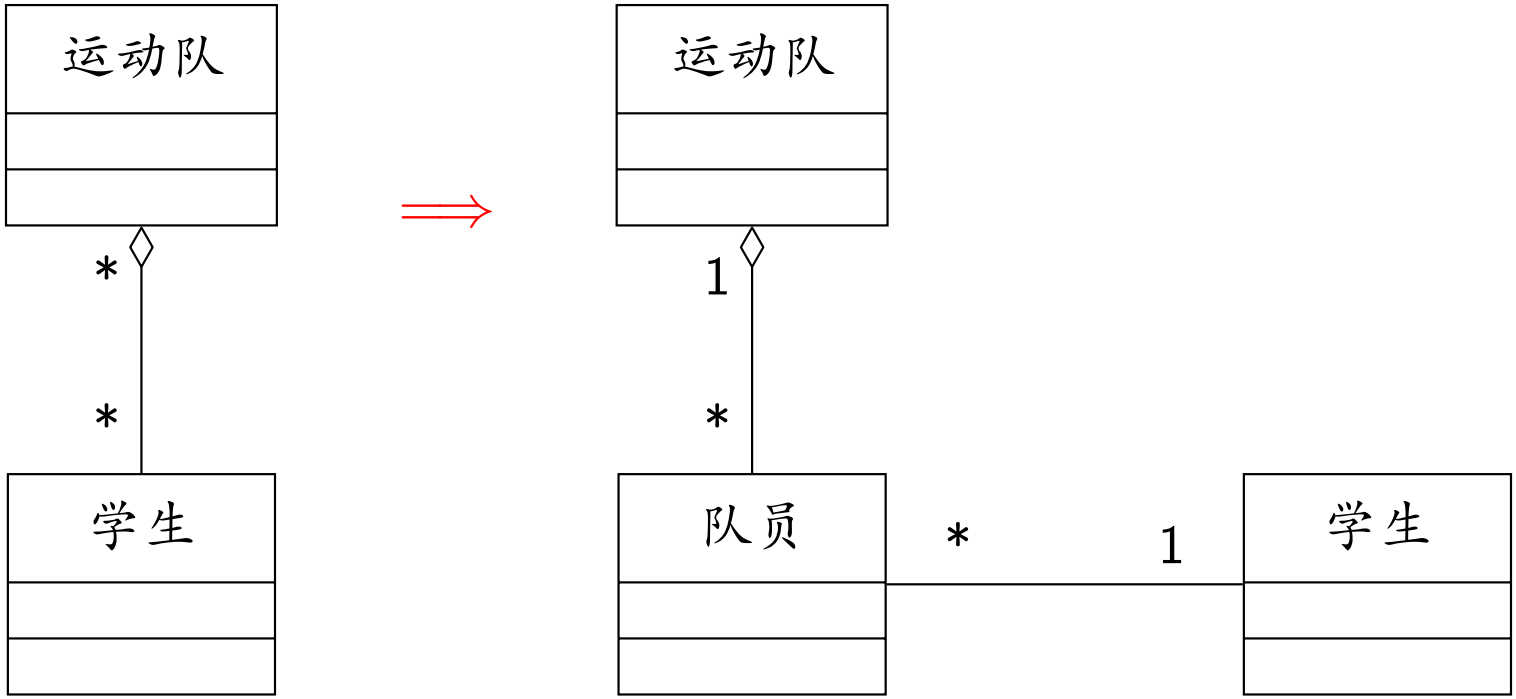


对整体-部分结构的处理



对整体-部分结构的处理

例：多对多的整体-部分结构的转化



数据接口部分类的设计

设计一个名为“对象存取器”的对象类，它提供两种操作

- “对象保存”
  将内存中一个对象保存到相应的数据库表中
- “对象恢复”
  从数据库表中找到对象所对应的元组，把它恢复成内存中的对象

数据接口部分类的设计

执行对象保存和对象恢复操作需要知道对象的下述信息：

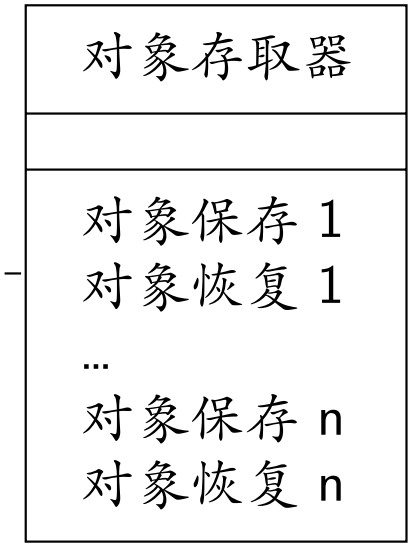
- 它在内存中是哪个对象
  为了知道从何处取得对象数据，或者把数据恢复到何处
- 它属于哪个类
  为了知道该对象应保存在哪个数据库表中
- 它的关键字
  为了知道该对象对应数据库表的哪个元组

对象存取器的设计方案: 第一种

对每个要求保存和恢复的对象类，分别设计一个“对象保存”操作和一个“对象恢复”操作，每个操作只负责一类对象的存或取

优点: 接口参数简单，只需变量名和关键字信息；操作容易实现，通常只需一个数据操纵语句

缺点: 操作个数太多，很难在问题部分采用统一的消息协议

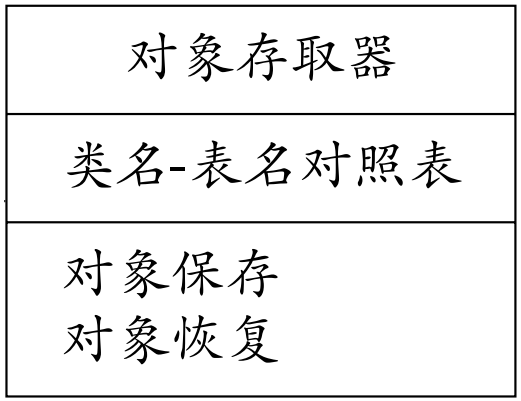


对象存取器的设计方案: 第二种

只设计一个“对象保存”操作和一个“对象恢复”操作供全系统所有要求保存和恢复的对象类共同使用

优点: 操作少，消息协议统一

缺点: 操作接口参数需增加类名信息；实现难度大，表名、关键字、对象类型等信息不能在编程时确定



问题域部分的修改

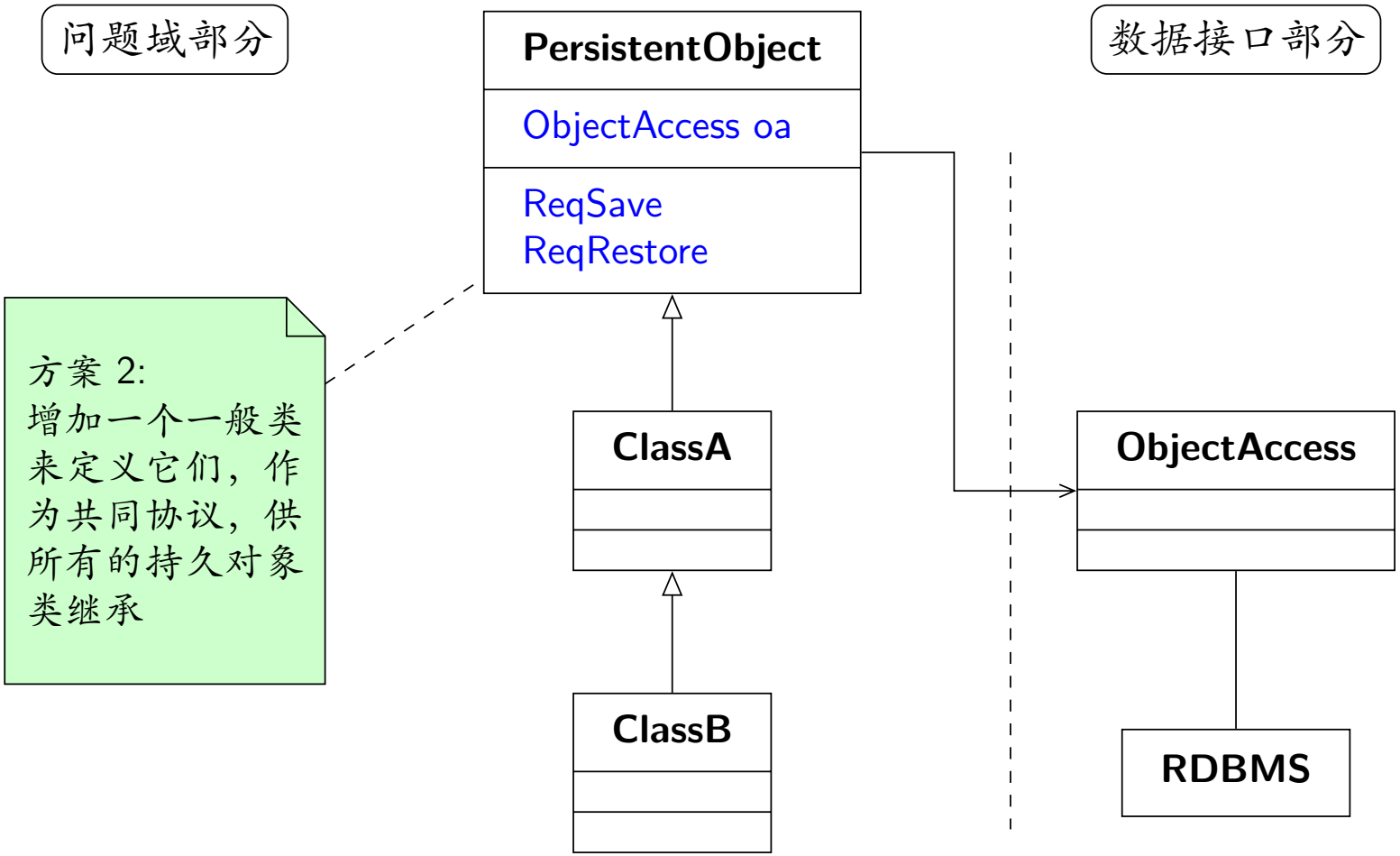
采用第一种方案时:

问题域部分每个请求保存或恢复的类，都要使用不同的操作请求语句，这些请求只能分散到各个类中

采用第二种方案时:

在问题域部分设计一个高层的类，提供统一的协议，供各个需要在数据库中存储其对象实例的类继承，可以做到和采用文件系统时的处理完全一致

问题域部分的修改


问题域部分
PersistentObject
ObjectAccess oa
ReqSave
ReqRestore
数据接口部分
方案 2:
增加一个一般类来定义它们，作为共同协议，供所有的持久对象类继承
ClassA
ClassB
ObjectAccess
RDBMS

针对 OODBMS 设计的考虑策略

- 从应用系统到数据库，从内存空间到外存空间，数据模型都是一致的。因此，几乎不要为此再做更多的设计工作
- 类图中的类一般不需要类似于规范化的改造，也不需要专门设计专门负责对象保存与恢复的对象类
- 主要考虑如何用 OODBMS 提供的数据定义语言、数据操纵语言和其它编程语言来实现 OOD 模型
- 必要时要根据语言的功能限制对类图做适当的修改

内容提要

“面向对象界一直广为流传的一项争论就是构件和正常类之间的区别为何。”
“‘什么是构件’这一问题是一个争论不休的题目。”

“要点是，构件代表可以独立购买与升级的软件片。因此，把一个系统分成若干构件既是一个技术抉择，又是一个销售抉择。”

——Fowler M and Scott K. UML Distilled 3rd edition

面向对象方法与构件技术

“基于构件”与“面向对象”并不是两种相互取代的方法或技术，它们是正交的、互补的关系

用 OO 方法开发的系统，既可以组织成构件，也可以不组织成构件

构件技术既可以用于 OO 软件开发，也可以用于非 OO 的软件开发

面向对象方法与构件技术

面向对象侧重于用什么概念来认识问题域，并把其中的事物以及其关系映射到软件系统中，是一种贯穿软件生命周期的软件方法学。

构件技术的侧重点是如何把系统组织成能够独立地进行生产、组装、复用、部署、发布、销售和升级的产品单位。

面向对象方法与构件技术

构件技术已经发展到软件生命周期的各个阶段，但它并不取代现有的分析与设计方法

与其他软件工程方法和技术相比，面向对象方法与构件技术之间的配合最为紧密、融洽。面向对象方法的抽象、继承、封装、聚合、多态等概念与原则对构件技术形成良好的支持

构件是一种比类粒度更大的系统单位。一个构件可以包括多个类，一个类不应该拆散到不同的构件。这意味着，构件的概念并不影响面向对象概念的语法和语义

OO 模型的构件化

构件化的意义——支持基于构件的软件开发

- OOA 阶段：支持分析级的软件复用
- OOD 阶段：支持设计级的软件复用，并且支持以构件为单位进行编程实现
- OOA 阶段的构件通常不是最终结果，在 OOD 阶段会有变化

OO 模型的构件化

主要工作:

把类图中的类组织成一些可以独立进行编程、发布、销售和升级的构件

基本原则:

构件的粒度不宜过小，一个构件通常可以包含多个类，除非某些类本身就已经很庞大

一个类可以在多个构件中复用，但是不把一个类拆分到多个构件中，即：把类看成一个原子的系统单位

如何将类组织为构件

考虑的因素和面向对象划分包 (package) 很接近:

- 各个类之间关系的紧密程度
- 在问题域中所对应的事物
- 所提供的功能类别
- 彼此之间通信频繁程度
- 在系统中的分布与并发情况

如何将类组织为构件

结论:

- 以包作为组织构件的基本依据
- 必要时对包进行合并或拆分
- 兼顾软件的发布、销售等因素

软件制品的组织

- 源文件制品：构件及其接口编程实现后的源文件
- 可执行文件制品：由源文件编译产生
- 数据库制品和数据文件制品：按部署的结点打包
- 模型文件制品：各种模型图及其规约
- 测试用例制品：按被测试的程序单位进行组织
- 其他制品：如产品说明书、用户手册、联机帮助文件等

系统部署过程与策略

针对不同的目标确定不同的部署方案

部署过程

1. 描述结点及通信路径，例如：

2. 配置结点的执行环境
   - 操作系统、编译系统、DBMS、界面支持系统、中间件等

系统部署过程与策略

3. 把制品部署到结点上
   包的组织策略、系统分布策略和构件组织策略，决定了各个结点上应该有哪些构件
   - 源文件制品和可执行文件制品：部署到相应的构件所在的结点上
   - 模型文件制品和测试用例制品：根据模型文件和测试用例的作用范围
   - 数据库制品和数据文件制品：根据应用范围和数据传输量较小的原则
   - 产品说明书、用户手册和联机帮助文件等：根据使用范围