

## 11.1 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。

大部分实用的系统都要处理数据的永久存储问题

数据保存于永久性存储介质

在数据管理系统的支持下实现其存储、检索和维护

在面向对象的系统中，数据的存储表现为对象存储

问题范围：

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

## 不同的数据管理系统:

文件系统

关系型数据库管理系统

面向对象的数据库管理系统

——各有不同的数据定义方式和数据操纵方式

针对不同的数据管理系统，需要做不同设计

根据所选用的数据管理系统特点，设计一些专门处理其它对象的永久存储问题的对象

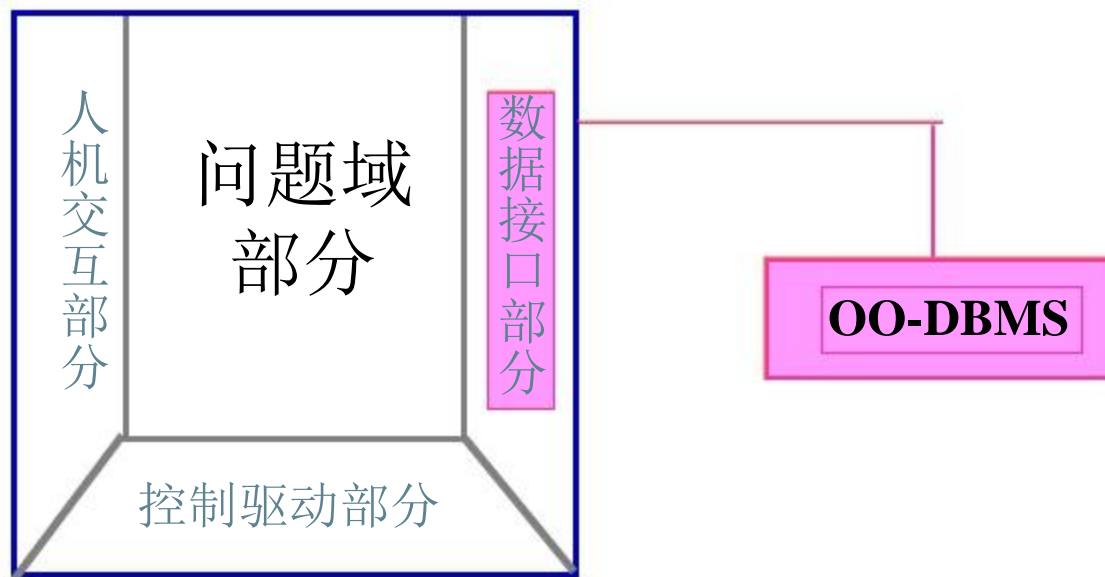
组织成一个独立的组成部分—— 数据接口部分

优点:

集中解决对象存储问题

隔离数据管理系统对其它部分的影响

选用不同的数据管理系统时，问题域部分变化较少



## 14.2 数据管理系统及其选择

数据管理系统

——实现数据存储、检索、管理与维护的系统

包括文件系统和数据库管理系统两大类

文件系统 **file system**

关系型数据库管理系统 **RDBMS**

面向对象的数据库管理系统 **OODBMS**

## (一) 文件系统

通常是操作系统的一部分

管理外存空间的文件数据

提供存储、检索、更新、共享和保护等功能

文件结构

物理结构

文件数据在存储空间的存放方法和组织关系

逻辑结构

呈现给用户的文件结构

如流式结构、记录式结构 等

文件系统提供的支持

在人机界面上进行操作的系统命令

在程序中使用的广义指令

创建、删除、打开、关闭、读、写、控制等

编程语言可以提供更方便的文件定义与使用方式

## 文件系统的优缺点

优点：

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

只能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

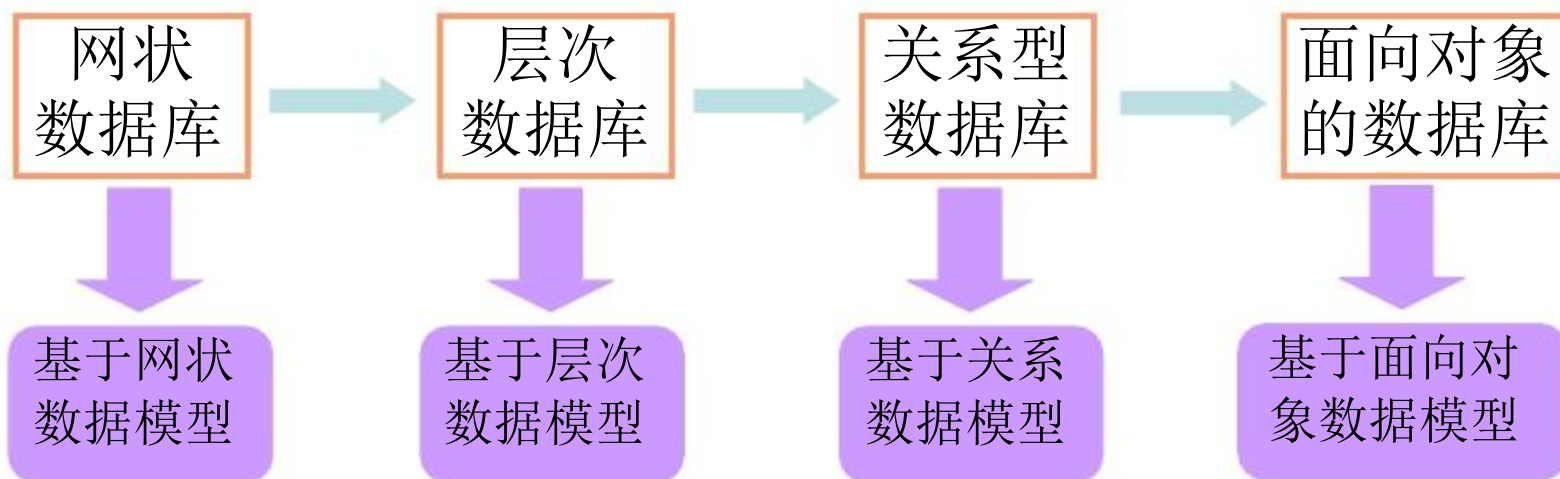
## （二）数据库管理系统

**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。

**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

**数据模型**：描述如何在数据库中定义数据及其操作，内容包括：

实体及实体间联系的数据结构描述，对数据的操作，完整性约束  
不同的数据库基于不同的数据模型



# 1、关系数据库管理系统 RDBMS

## 关系模型

给定一组域  $D_1, D_2, \dots, D_n$

其笛卡尔积  $D_1 \times D_2 \times \dots \times D_n$  的一个子集就是一个关系，  
又称二维表

基本要求：关系的每个属性必须是原子的

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联系的数据

每一列称作一个属性

每一行称作一个元组

数据的运算：

提供并、交、差等集合运算  
以及选取、投影、联结等操作

## 关系数据库术语对照

数据库专业术语	开发者的惯术语	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

## 采用对象数据模型的数据库管理系统

背景:

越来越多的系统用**OO**技术开发（采用**OO**数据模型）

**RDBMS**的局限性（采用关系数据模型）

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、类、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:

数据定义与操纵语言、完整性保障、安全机制，并发控制、故障恢复、事务运行管理、可扩充

## 三种类型的OODBMS产品

- 1) 在**OOPL**基础上增加数据库管理系统的功能  
例如: **GemStone** 和 **ObjectStore**
- 2) 对**RDBMS**进行扩充, 使之支持面向对象数据模型  
并向用户提供面向对象的应用程序接口  
例如: **Iris** 和 **POSTGRES**
- 3) “全新的”**OODBMS**  
即按照面向对象数据模型进行全新的设计  
例如: **O<sub>2</sub>** 和 **DAMOKLES**

### (三) 数据管理系统的选择

理论上**OODBMS**最适合用**OO**方法开发的系统  
实际上要权衡技术和非技术两方面的因素

#### 非技术因素

与项目的成本、工期、风险、宏观计划有关的问题

产品的成熟性和先进性

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

### 文件系统的适应性

**优点：**可储任何类型的数据，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。以类和对象的形式定义的数据都可以用文件存储

**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；

**缺点：**操作低级；数据操纵功能贫乏；缺少数据完整性支持；缺少多用户及多应用共享、故障恢复、事务处理等功能

**不适应：**数据操纵复杂、多样，数据共享及数完整性维护要求较高的系统。

## 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：**对这些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

**问题：**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

**更不适合**图形、图象、音频、视频等多媒体数据和经过压缩处理的数据

文件系统和**RDBMS**的优点和缺点形成了明显对照

但是对二者的选择却未必互相排斥，有时它们是互补的。某些应用系统可能同时采用**RDBMS**和文件系统，分别存储各自所适合的数据。

## 面向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**采用的对象模型多有差异，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

针对三种  
数据管理系统  
文件系统  
RDBMS  
OODBMS

分别讨论

对象存储方案——如何把对象映射到数据管理系统

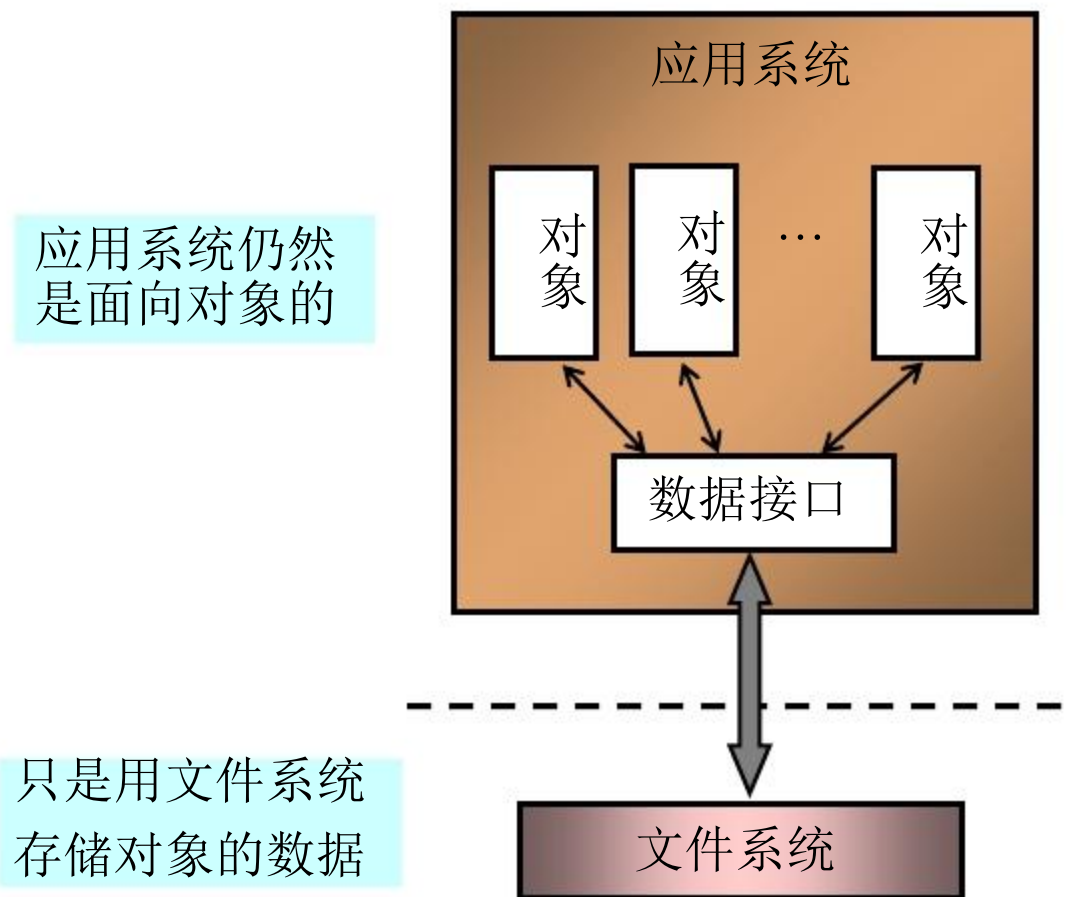
数据接口部分的设计策略——如何设计数据接口部分的对象类

如何对问题域部分做必要的修改

# (一) 针对文件系统的设计

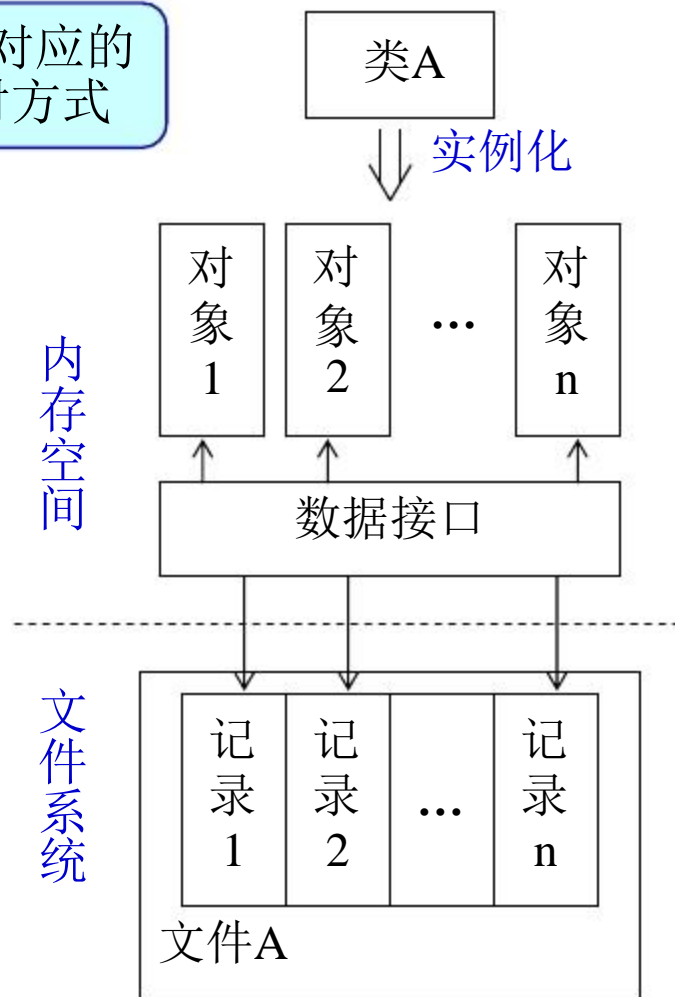
## 1、对象在内存空间和文件空间的映像

如何看待用文件系统存储对象

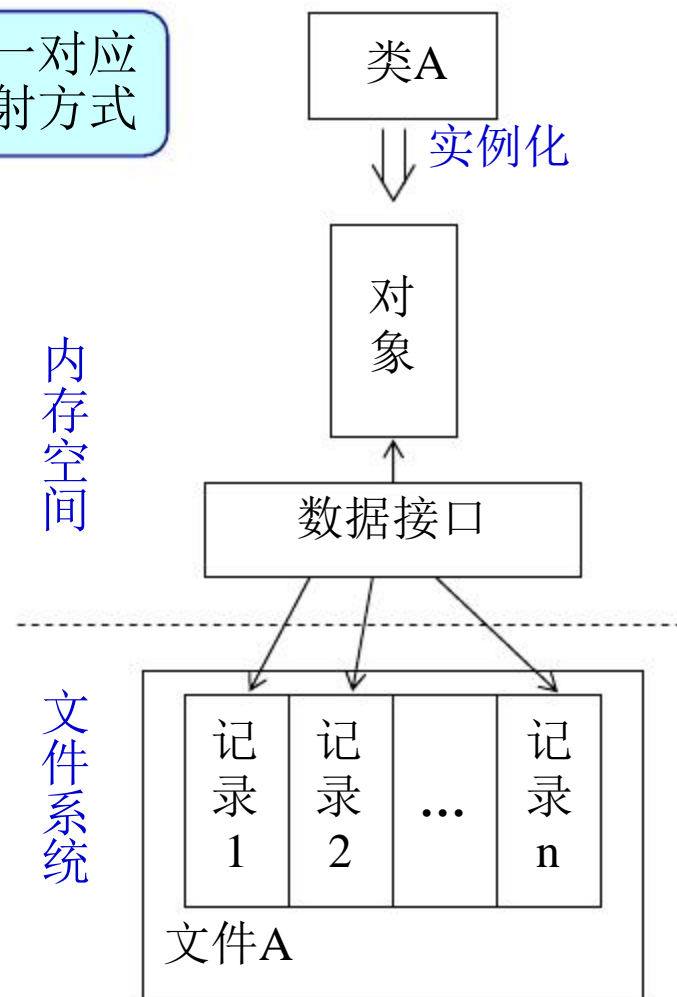


## 从应用系统的对象到文件记录的不同映射方式

一一对应的  
映射方式



非一一对应的  
映射方式



## 2、对象在文件中的存放策略

### 1) 基本策略

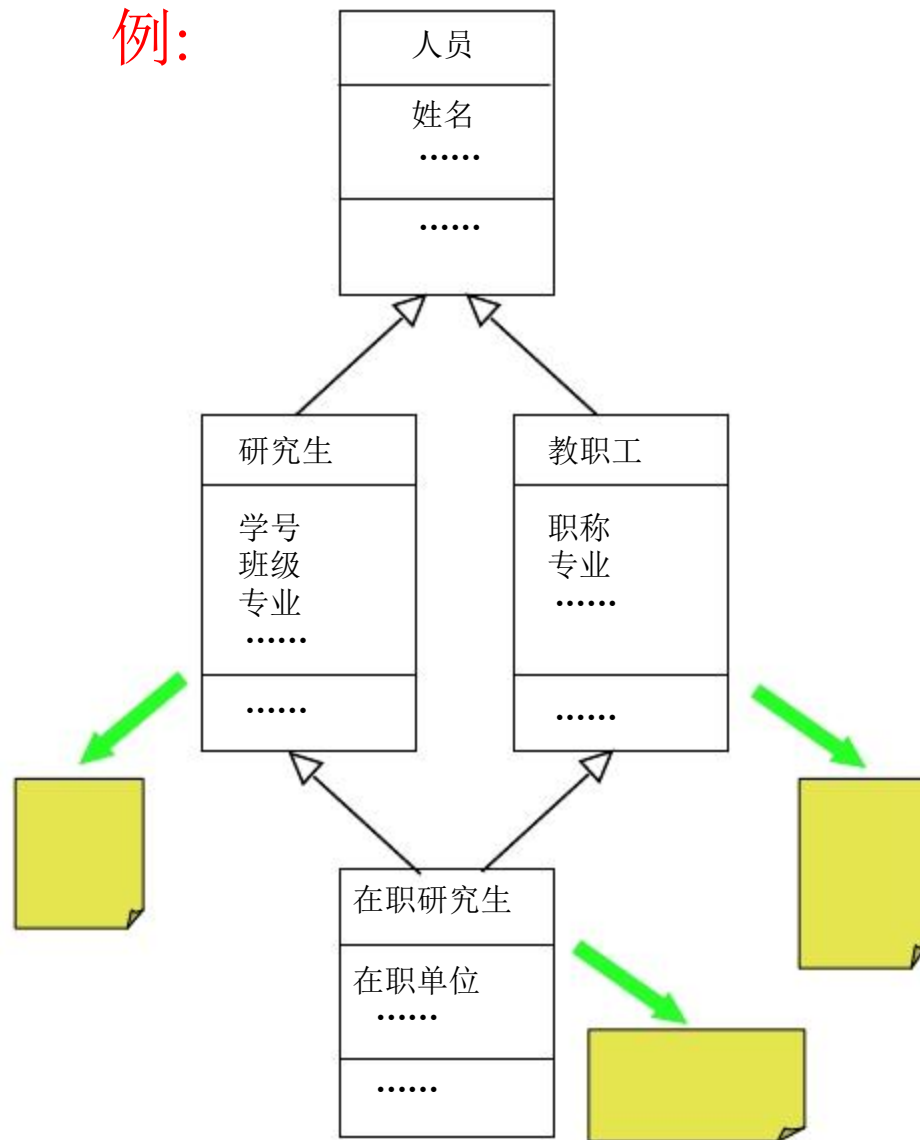
把由每个类直接定义、需要永久存储的全部对象实例存放在一个文件中；  
每个对象实例的全部属性作为一个存储单元，占用该文件的一个记录。

如何理解“由一个类直接定义的”对象实例

另一种策略——  
一个结构用一个文件

浪费空间  
模糊了对象分类关系  
使操作复杂化

例：



## 2) 提高检索效率

——在对象和文件记录之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

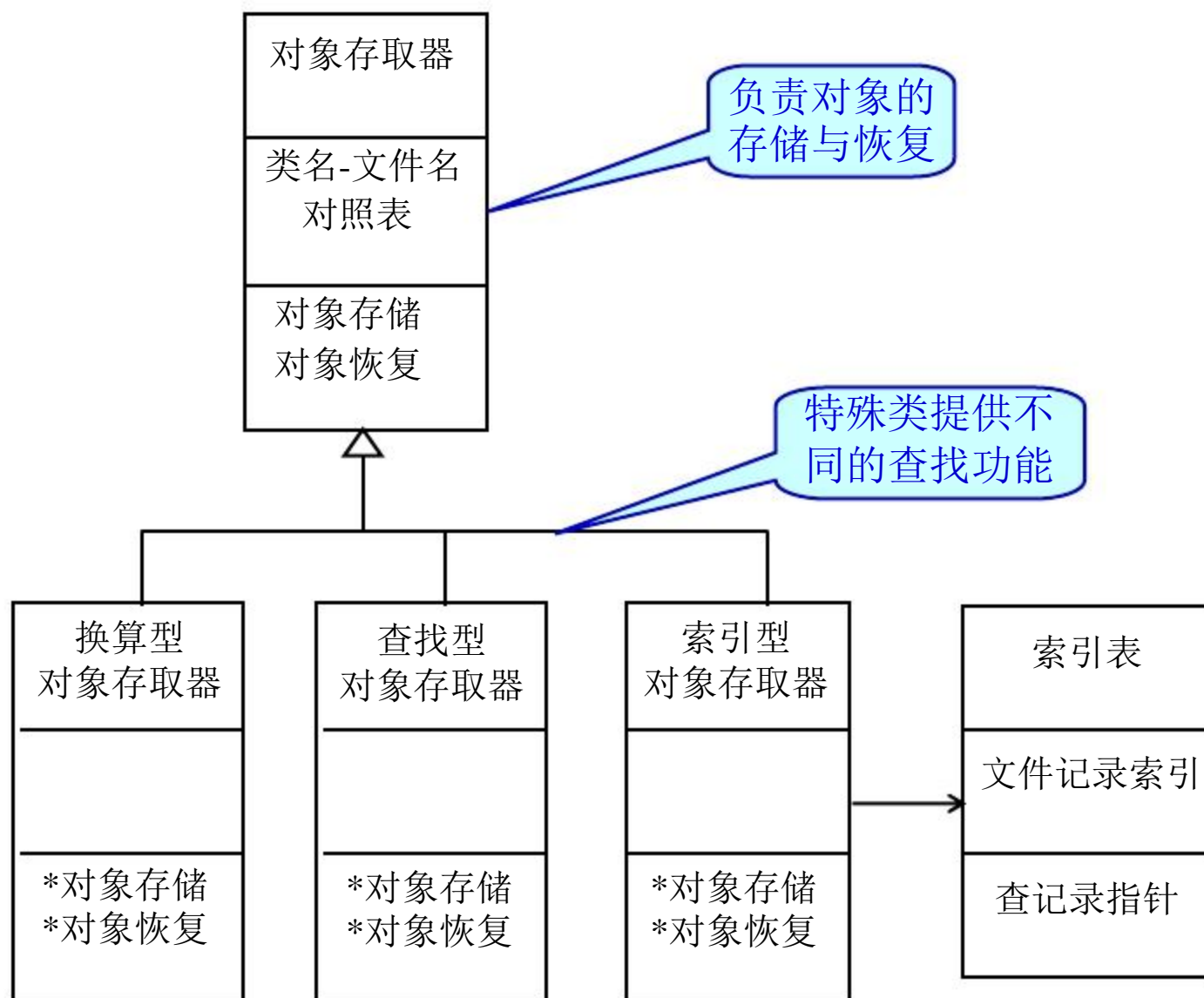
按关键字顺序安排记录，检索时采用折半查找法

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

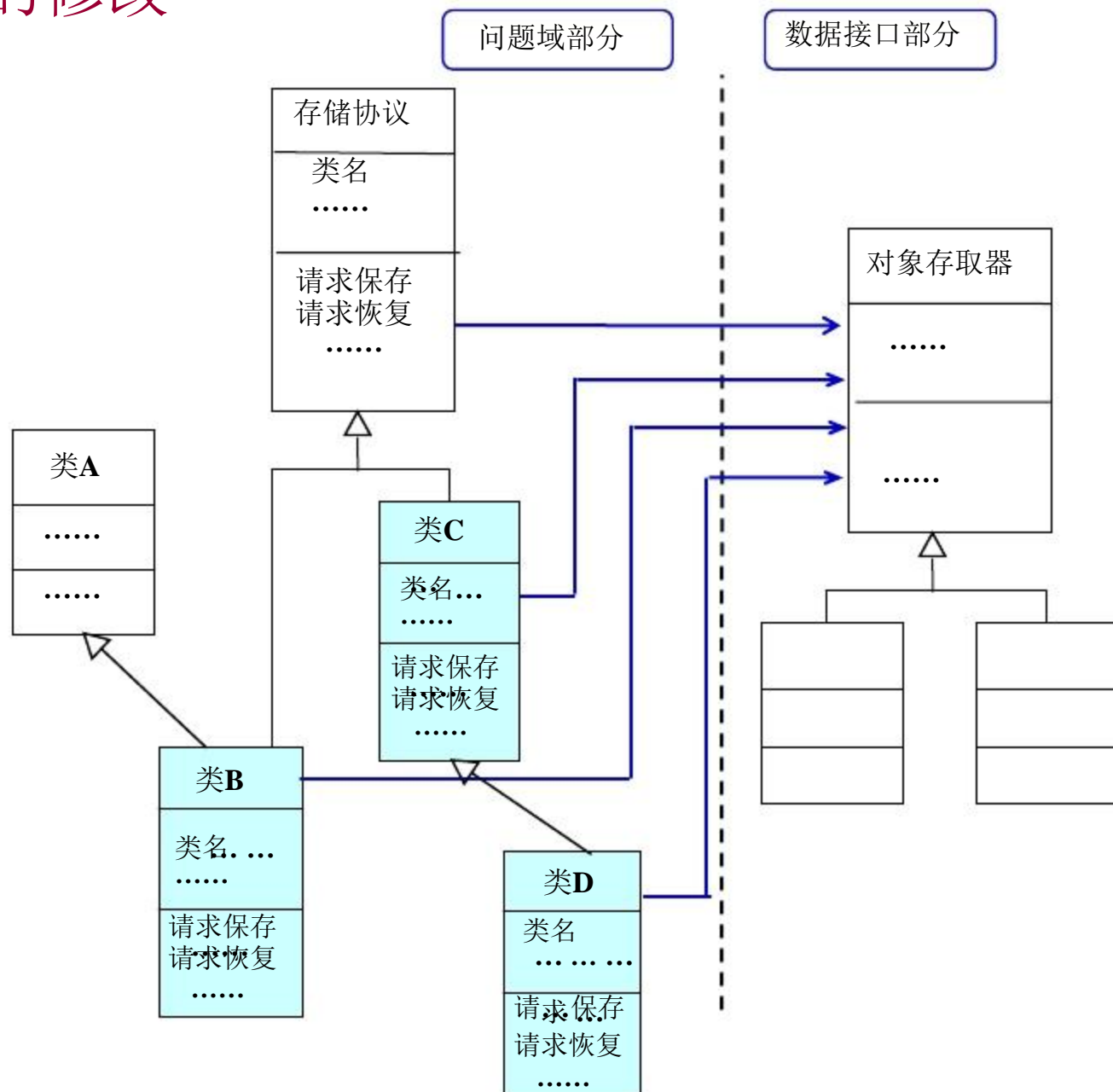
### 3、设计数据接口部分的对象类



## 4、问题域部分的修改

增加一个一般类来定义它们，作为共同协议，供所有的永久对象类继承

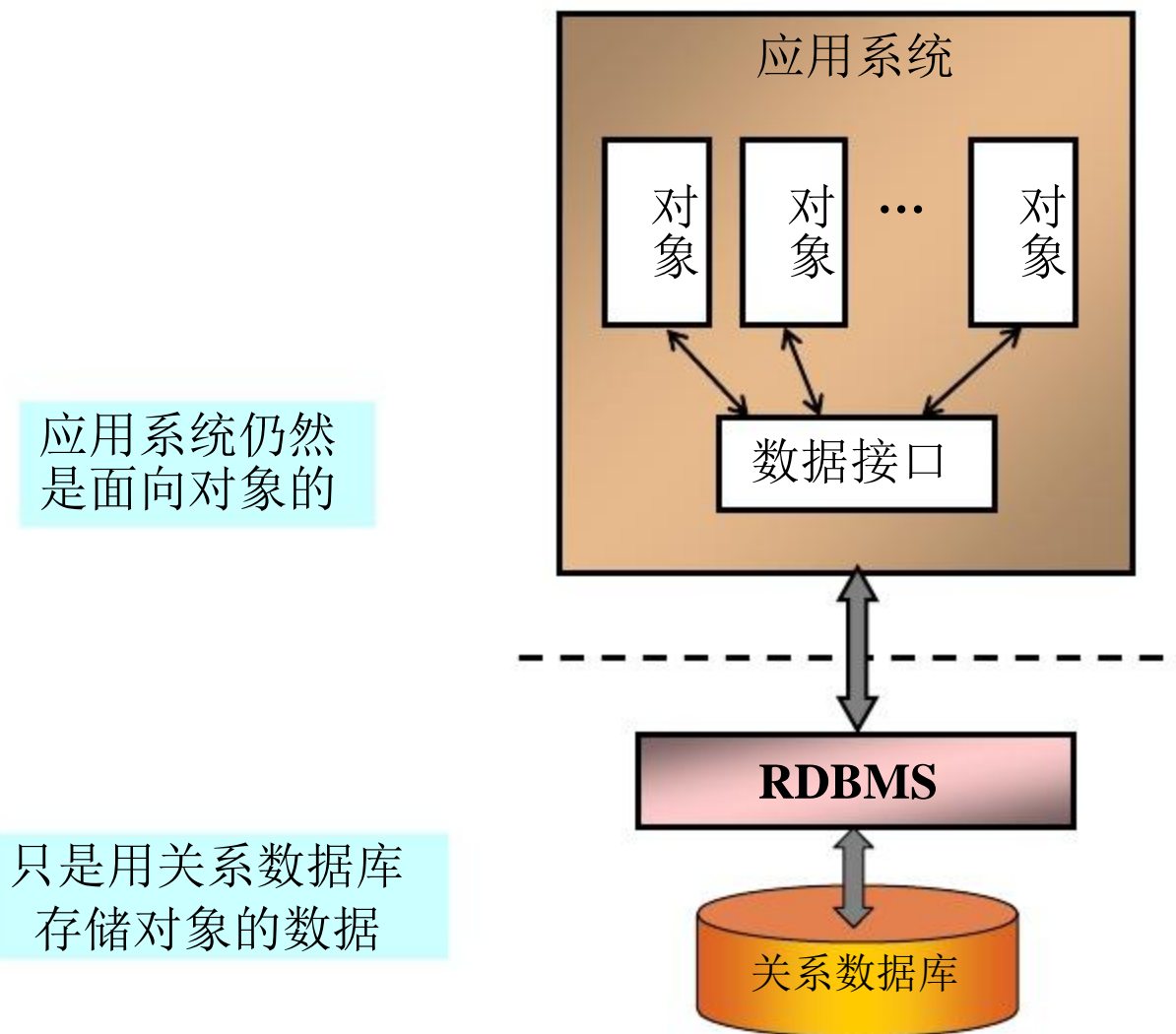
每个永久对象类都要增加请求存储和恢复所需的属性和操作，以便向数据接口部分发出请求



## （二）针对**RDBMS**的设计

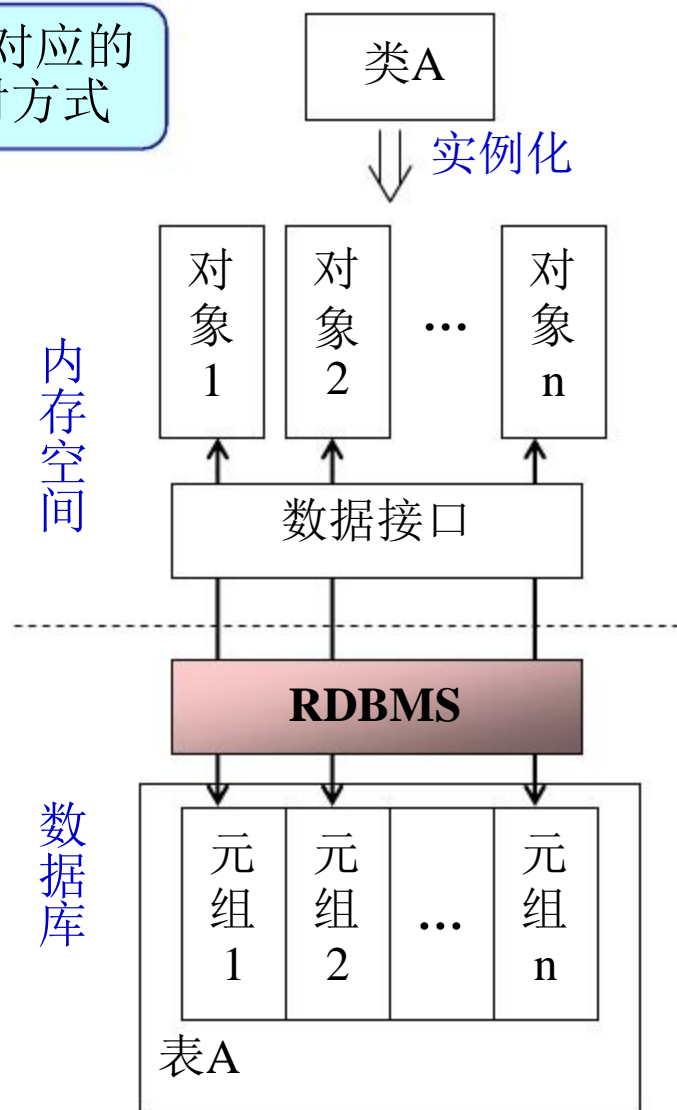
### 1、对象及其对数据库的使用

如何看待用 **RDBMS**存储对象

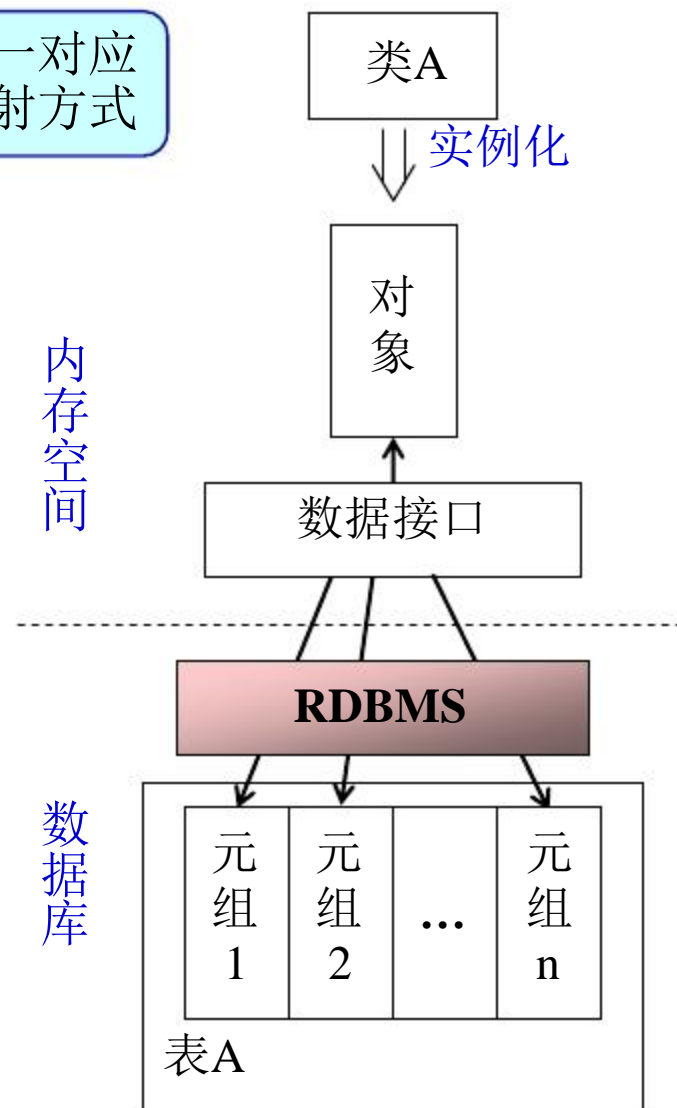


# 从应用系统的对象到数据库表元组的不同映射方式

一一对应的  
映射方式



非一一对应的  
映射方式

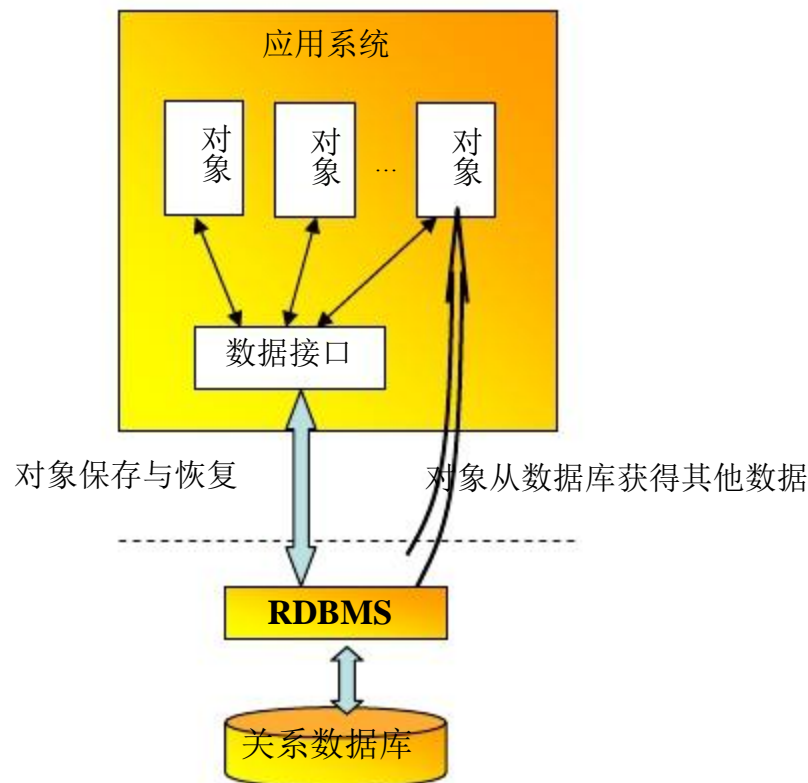


# 使用**RDBMS**和使用文件系统的不同

## (1) 系统以不同方式使用数据库中的数据

存储对象 或 使用其普通数据

——原因：数据库的共享性



## (2) 可能需要数据格式转换

原因：关系数据库对规范化的要求

## 2、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

## 1) 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）**：关系（表）的每个属性都必须是原子的。就是说，关系的每个属性都是单值的，它不再包含内部的数据结构。

是由关系数据模型决定的，是对一个关系的起码要求

**第二范式（2NF）**：如果一个关系的所有非关键字属性都只能依赖整个关键字（而不是依赖关键字的一部分属性），则该关系在第二范式中。

主要为了解决关系中的函数依赖带来的更新异常问题。也可以减少数据冗余，但更新异常问题更为重要

**第三范式（3NF）**：如果一个关系在第二范式中，而且没有传递依赖，则该关系在第三范式中。

**Boyce-Codd范式（BCNF）**：如果一个关系的每个决定因素都是候选关键字，则该关系在BCNF中。

**第四范式（4NF）**：如果一个关系在BCNF中，而且没有多值依赖，则该关系在第四范式中。

主要为了减少数据冗余

未必规范化程度越高越好

规范化的代价

——响系统的可理解性，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，恰恰有助于达到第**2NF**、**3NF**、**BCNF**和**4NF**要求的条件

例如“通信地址”属性

## 例1：一个不满足3NF的关系及其规范化

**HOUSING (SID, Building, Fee)**

**Key:SID**

**Functional Dependencies:**

**Building→Fee**

**SID→Building→Fee**

SID	Building	Fee
100	Randolph	1200
150	Ingersoll	1100
200	Randolph	1200
250	Pitkin	1100
300	Randolph	1200

**STU-HOUSING**

**(SID, Building)**

**Key:SID**

SID	Building
100	Randolph
150	Ingersoll
200	Randolph
250	Pitkin
300	Randolph

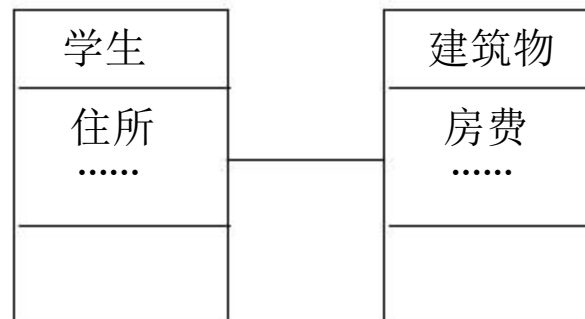
**BLDG-FEE (Building, Fee)**

**Key:Building**

Building	Fee
Randolph	1200
Ingersoll	1100
Pitkin	1100

分析原因：把描述不同事物的数据组织在一起

从面向对象的观点看应该定义“学生”和“建筑物”两个类——和规范化要求吻合



## 例2：一个不满足4NF的关系及其规范化

**STUDENT** ( **SID** , **Major** ,  
**Activity** )

**Key:** ( **SID** , **Major** , **Activity** )

<b>SID</b>	<b>Major</b>	<b>Activity</b>
100	Music	Swimming
100	Accounting	Swimming
100	Music	Tennis
100	Accounting	Tennis
150	Math	Jogging

**STU-MAJOR** ( **SID** ,  
**Major** )

**Key:** ( **SID** , **Major** )

<b>SID</b>	<b>Major</b>
100	Music
100	Accounting
150	Math

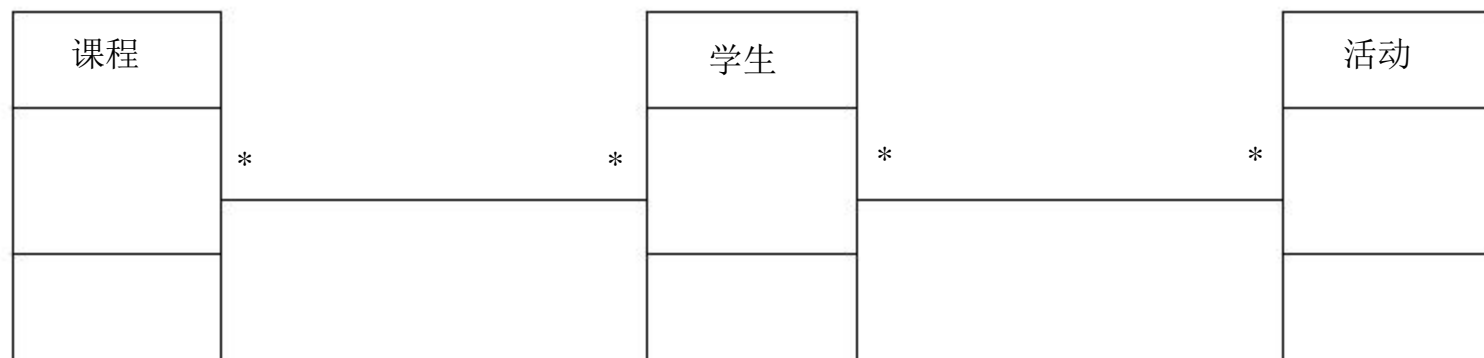
**STU-ACT** ( **SID** , **Activity** )

**Key:** ( **SID** , **Activity** )

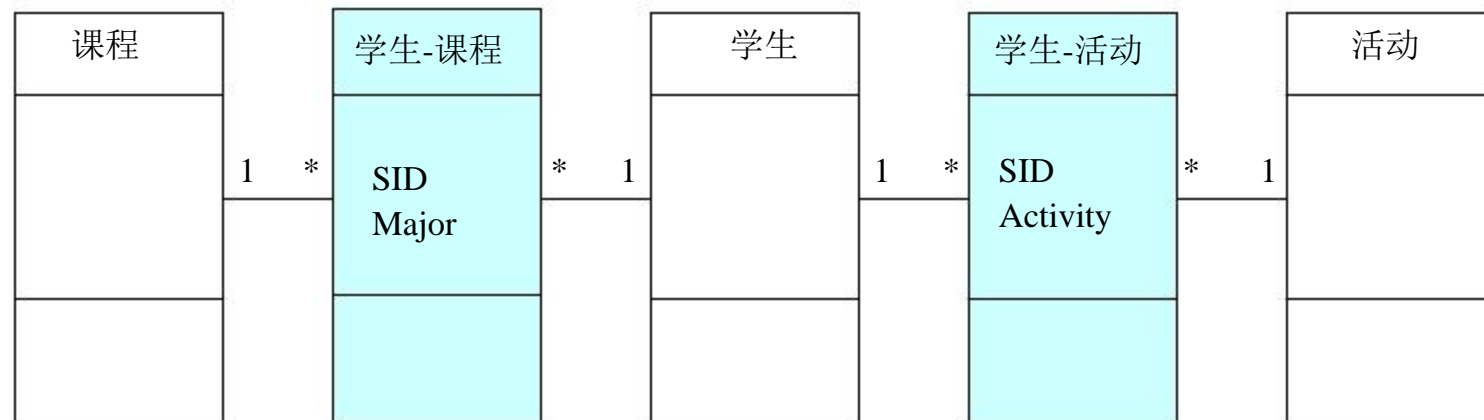
**SID**   **Activity**

100	Skiing
100	Swimming
100	Tennis
150	Jogging

## 用面向对象方法得到的分类——有三类对象



## 化解多对多关联之后的结果



为什么多出来**3**个类？

## 用面向对象方法未能避免函数依赖的例子

职工
职工编号 月工资 所得税 .....
.....

可以不通过规范化解决问题——讨论为什么

## 2) 修改类图

### 规范化的两种策略

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

### 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

采用第二种策略——按规范化的要求修改类图中的类  
作为问题域部分的设计内容之一

### 3) 确定关键字

用较少的属性作关键字，将为含关键字的操作带来方便  
常用的技巧——引入编号

#### 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

## 4) 从类图到数据库的映射

### (1) 映射策略概要

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）

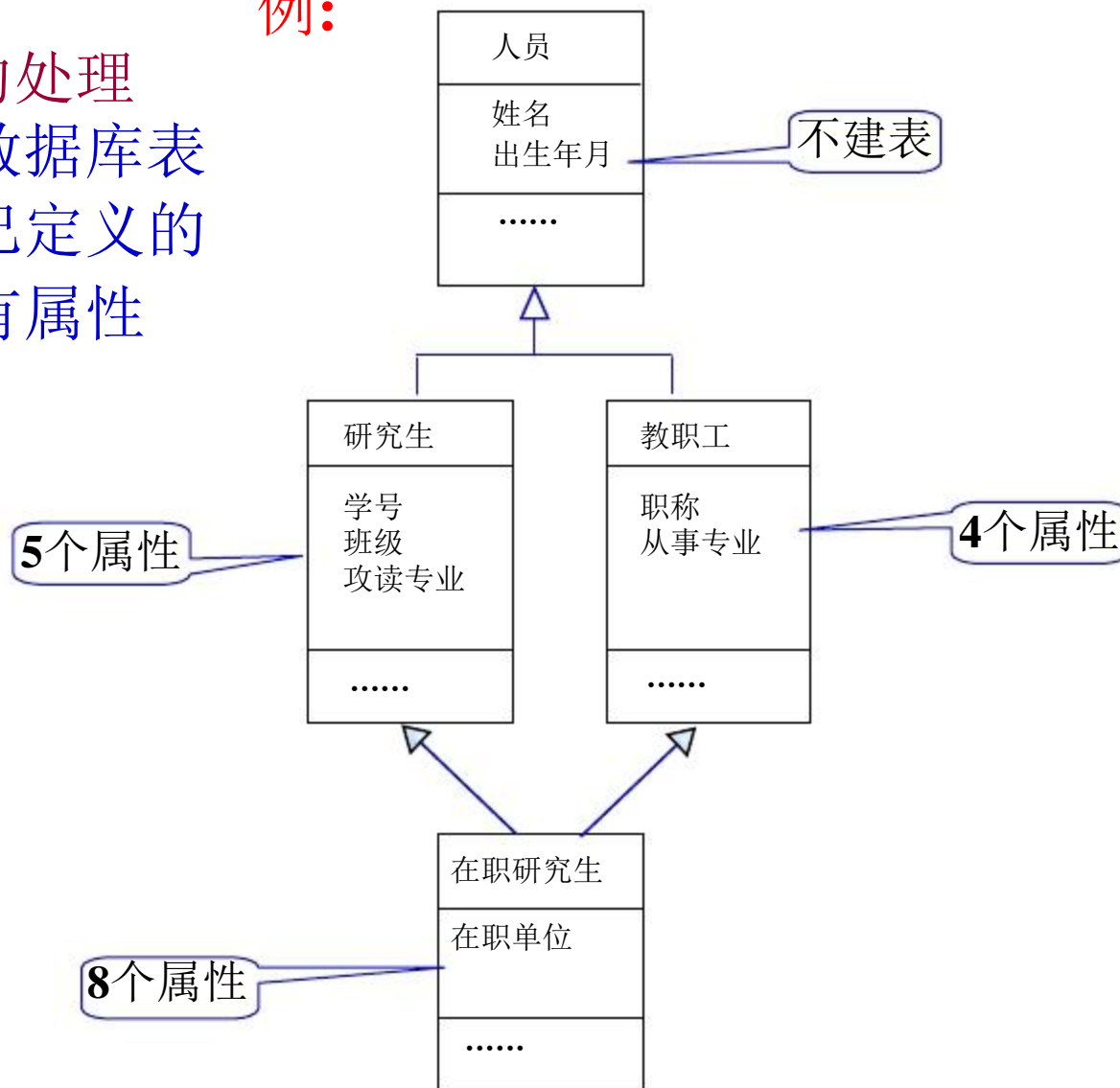
## (2) 对OO概念的处理

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

例:



## 对关联的处理

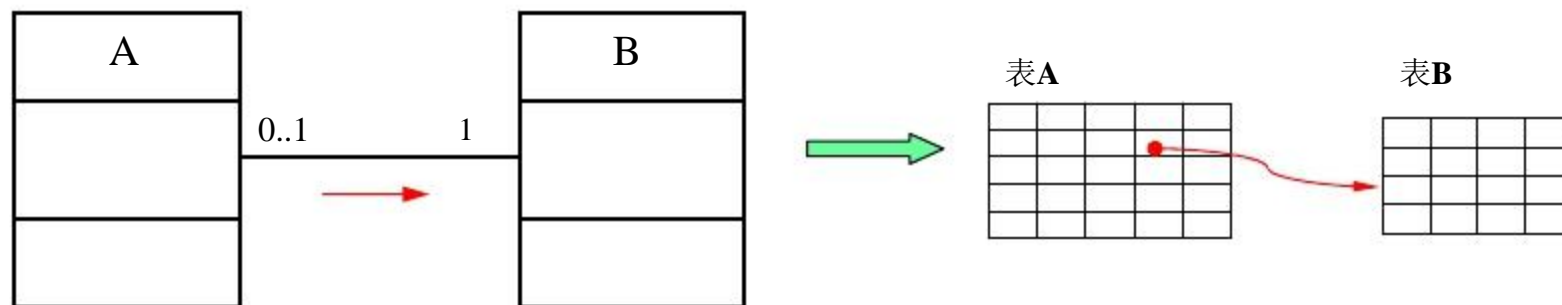
在关联连接线一端的类中定义一个（或一组）属性，表明另一端类的哪个对象实例与本端的对象实例相关联

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

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

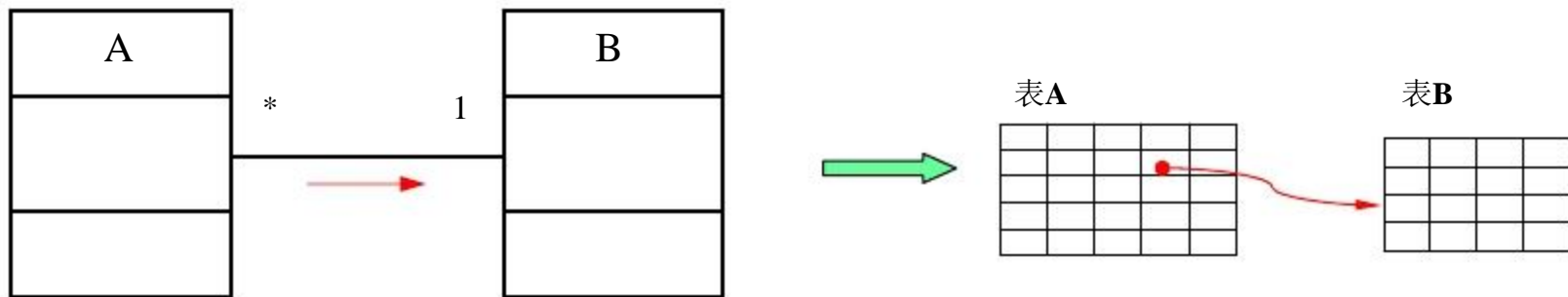
一对一的关联



从**B**端指向**A**端，则**B**表的外键对有些元组可能是空值（**NULL**）  
从**A**端指向**B**端则不存在这一问题

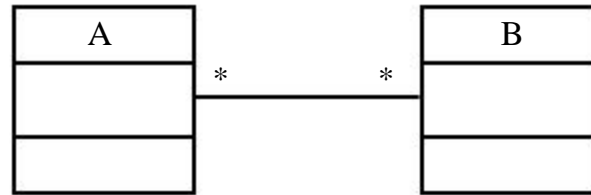
## 一对多的关联

从多重性约束为“**m**”的一端指向多重性约束为“**1**”的一端

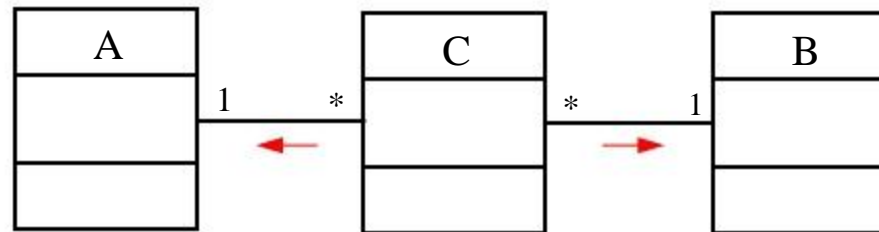


映射为数据库表后，  
**A**表以**B**表的主键作为自己的外键

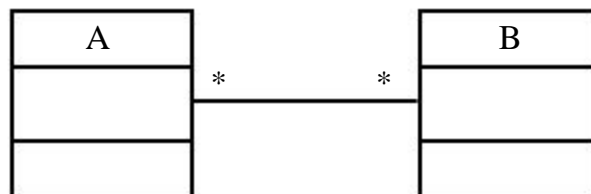
## 多对多的关联



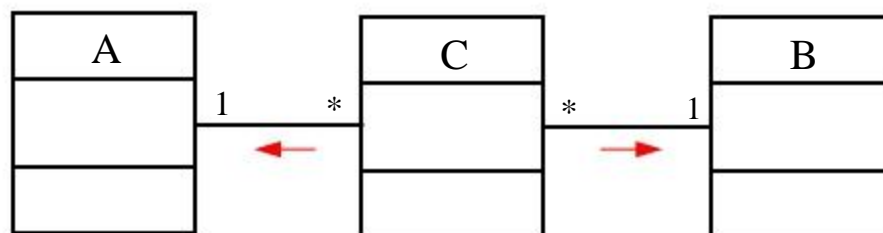
先在类图中化为两个一对多的关联



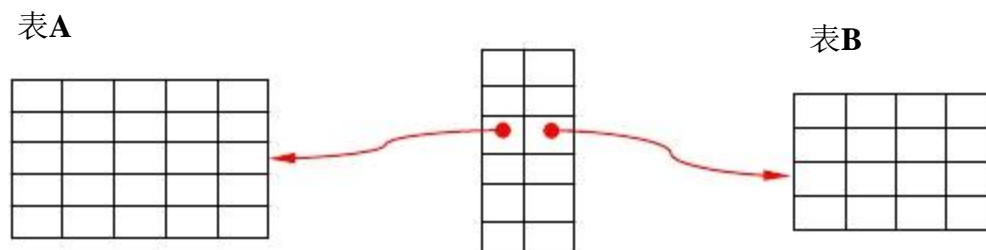
## 多对多的关联



先在类图中化为两个一对多的关联



然后将每个类映射到一个数据库表



**C表**含有两个外键，一个是**A**的主键，一个是**B**的主键

对象类转化为数据库表的三种情况：

- ①表中只包含描述本类事物自身特征的属性
- ②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
- ③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

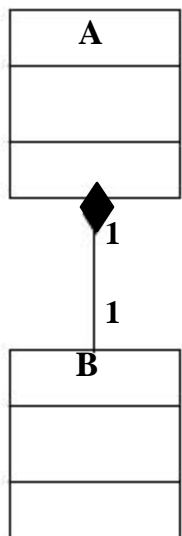
分为紧密、固定的方式 和 松散、灵活的方式  
二者的区别将通过数据库表的设计体现出来

紧密、固定方式：

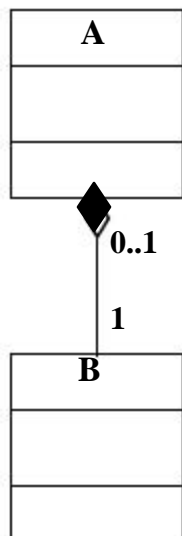
把部分对象类的属性合并到整体对象类中

松散、灵活方式：

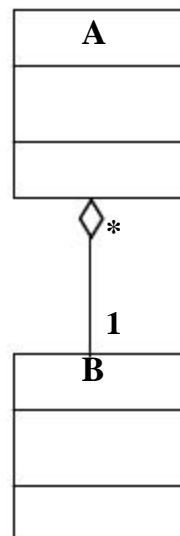
整体对象类和部分对象类分别建立一个表  
通过外键表现整体部分关系



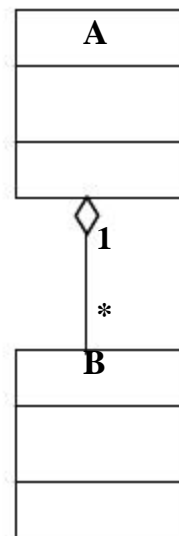
(a)



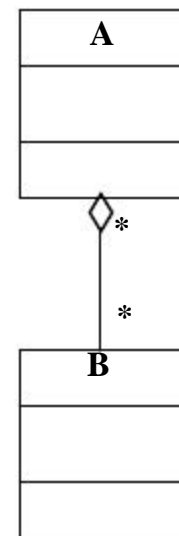
(b)



(c)



(d)



(e)

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**

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**

松散方式:

建立**A**、**B**两个表  
**A**指向**B**

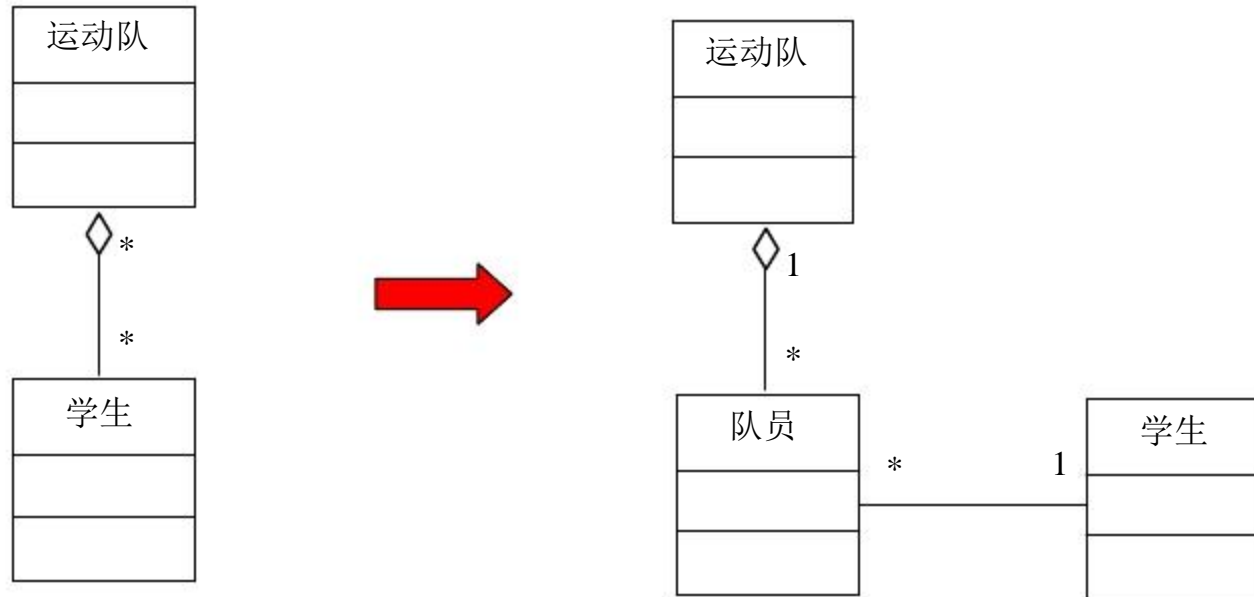
松散方式:

建立**A**、**B**两个表  
**B**指向**A**

松散方式:

参考多对多关联解决办法  
首先解决多对多问题  
然后建立**A**、**B**两个表以及新增类的表

## 例：多对多的整体-部分结构的转化



### 3、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

执行这些操作需要知道对象的下述信息：

- 它在内存中是哪个对象

从而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

从而知道该对象应保存在哪个数据库表中

- 它的关键字

从而知道该对象对应数据库表的哪个元组

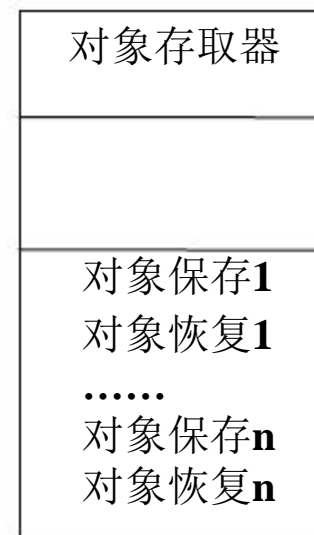
## 第一种设计方案

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  
因此下述信息是确定的：

对象存放在哪个数据库表  
关键字所包含的属性数目与名称

在操作接口中只需传递如下参数：  
内存中的一个对象变量  
用来提供或接收对象数据  
对象关键字的值



优点：

每个操作都很容易实现，通常只需要一个数据操纵语句，  
(例如静态**SQL**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

## 第二种设计方案

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

操作接口的参数应传送三项信息：  
类名

指明被存取的对象属于哪个类

对象变量

提供或接受被存取的对象数据

关键字的值

指明是哪个对象实例要求保存  
或恢复

数据类型不  
能静态确定

对象存取器
类名-表名 对照表
对象保存 对象恢复

优点：

操作少，消息协议统一

缺点：

实现难度大

——表的名称、关键字的构成、对象的类型不能在编程时确定  
需要编程语言和数据操纵语言提供较强的支持

## 4、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

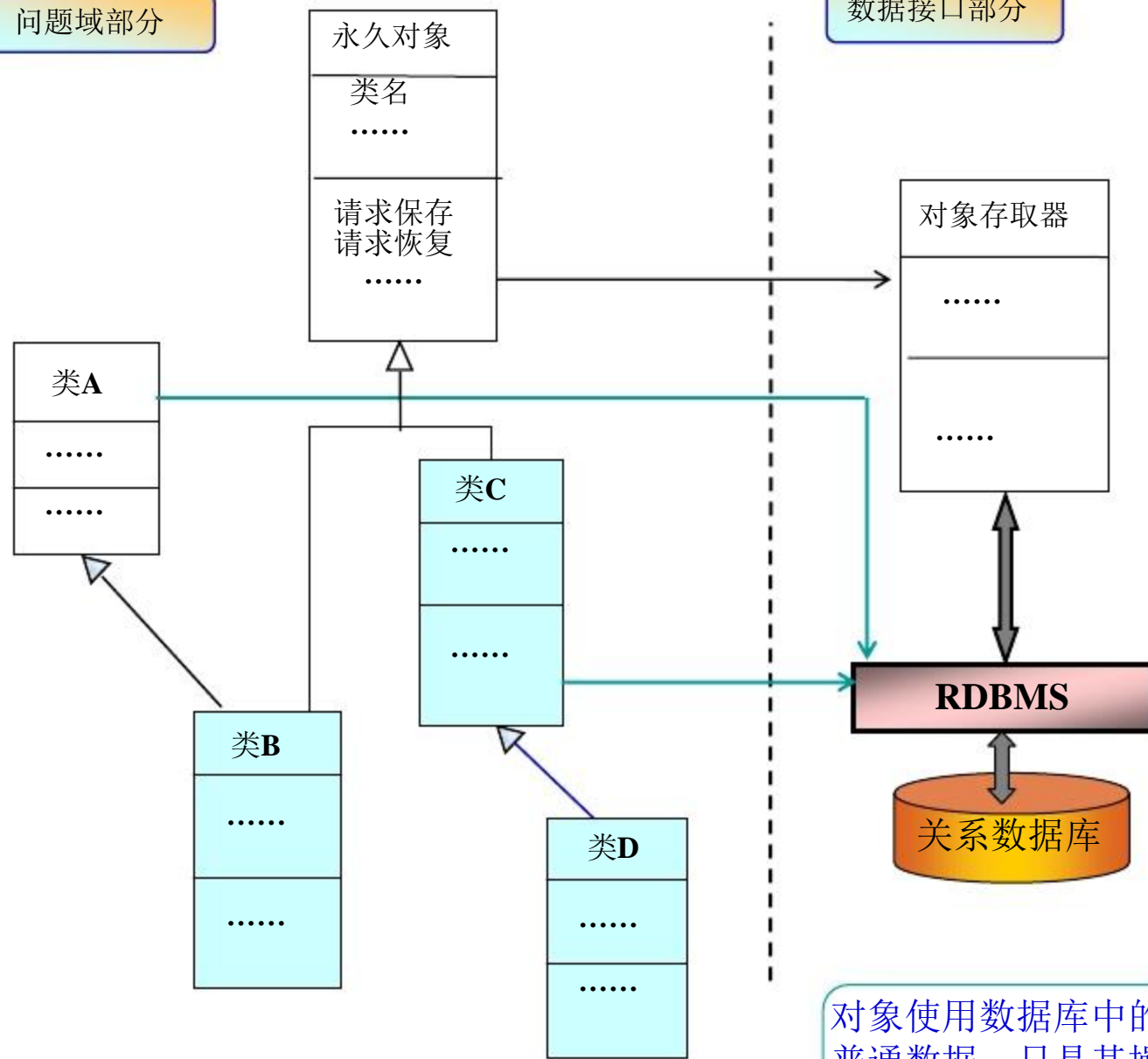
在问题域部分设计一个高层的类，提供统一的协议，供各个需要在数据库中存储其对象实例的类继承

可以做到和采用文件系统时的处理完全一致

问题域部分

数据接口部分

增加一个一般类来定义它们，作为共同协议，供所有的永久对象类继承



采用第二种方案时  
修改情况和采用文件  
系统时完全一致

对象使用数据库中的  
普通数据，只是其操  
作算法的实现问题

### (三) 针对**OODBMS**的设计

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——实现类和对象的定义和对数据库的访问

必要时要根据语言的功能限制对类图做适当的修改

## 15.1 面向对象方法与构件技术

**M. Fowler:**

“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。”

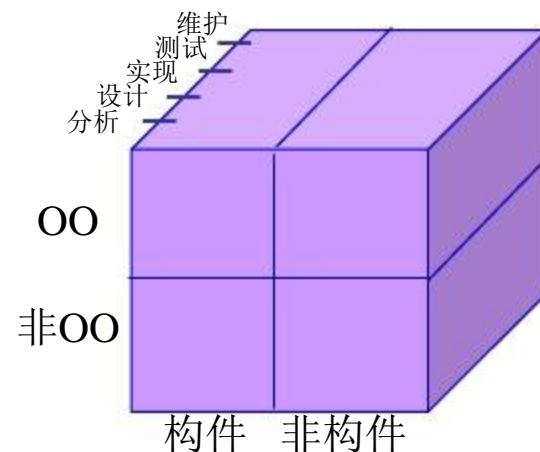
——Fowler M and Scott K. UML Distilled 3rd edition

“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面向对象的软件开发，也可以用于基于其他软件方法学的开发。

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。



构件技术已经发展到软件生命周期的各个阶段——通过领域分析识别领域构件，在系统分析、设计和实现中形成分析级、设计级和实现级的构件，在构件生产线上进行构件的分析、设计和编程。但是这一切并不取代现有的分析与设计方法，正如它并不取代任何一种编程语言一样。

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，

## 15.2 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和OOD阶段：**支持分析级和设计级的软件复用

**OOD阶段：**支持以构件为单位进行软件的编程实现

### 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：

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

这一切正是在面向对象建模中进行包的划分所考虑的因素。

## 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

## 15.3 系统部署

### 1、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

## 2、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

例如：针对系统安装、售后服务和最终用户的不同方案

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...  
标准配置，自选配置

- (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制品  
根据使用范围

# 谢谢大家