第一章 绪论

1.1 数据库系统概论

1.1.1 数据库的四个概念

1. 数据（ *data* ）
   描述事物的符号记录称为***数据***。
   数据的含义称为数据的语义，数据及其语义是分不开的。
2. 数据库（ *database，DB* ）
   数据库是长期存储在计算机内、有组织的、可共享的大量数据的集合。
   基本特点： *永久存储、有组织、可共享*
3. 数据库管理系统（ *DataBase Management System，DBMS* ）
   数据库管理系统是计算机的基础软件
   主要功能：
   - 数据定义功能：**DDL：数据定义语言**
   - 数据组织、存储和管理
   - 数据操纵功能：**DML：数据操纵语言（增删改查）**
   - 数据库的事务管理和运行管理
   - 数据库的建立和维护功能
   - 其他功能
4. 数据库系统（ *DataBase System，DBS* ）
   数据库管理系统是由数据库、数据库管理系统（及其应用开发工具）、应用程序和数据库管理员（ *DBA* ）组成的存储、管理、处理和维护数据的系统。

1.1.2 数据管理技术的产生和发展

- 人工管理
- 文件系统
- 数据库系统

1.1.3 数据库系统的特点

1. **数据结构化：**
   数据库系统实现整体数据的结构化，这是数据库的主要特征之一，也是数据库系统和文件系统的本质区别。

2. **数据的共享性高、冗余度低且易扩充：**
   数据共享可以大大减少数据冗余，节约存储空间。还能避免数据之间的不相容性与不一致性。
3. **数据独立性高：**
   物理独立性：用户的应用程序与数据库中的物理存储是相互独立的。
   逻辑独立性：用户的应用程序与数据库的逻辑结构是相互独立的。
4. **数据由数据库管理系统统一管理和控制**
   - 数据的安全性保护
   - 数据的完整性检查
   - 并发控制
   - 数据库恢复

1.2 数据模型

1. 概念模型

> - **实体：** 客观存在并可互相区别的事务称为实体
> - **属性：** 实体所具有的某一特性称为属性
> - **码：** 唯一标识实体的属性集称为码
> - **实体型：** 用实体名及其属性名集合来抽象和刻画同类实体，称为实体型，例如 ***学生（学号，姓名，性别，出生年月，所在院系，入学时间）***
> - **实体集：** 同一类型实体的集合称为实体集，例如 ***全体学生***
> - **联系：** 实体之间的联系通常是指不同实体集之间的联系，有 ***一对一，一对多和多对多*** 等 多种类型

> 概念模型的一种表示方式：实体-联系方法（Entity-Relationship approach）,该方法用 **E-R图** 来描述现实世界的概念模型， **E-R方法** 也称为 **E-R模型**

> 数据模型通常由**数据结构，数据操作和数据的完整性约束条件**三部分组成

2. 逻辑模型
   - 层次模型
   - 网状模型
   - **关系模型**
   - 面向对象数据模型
   - 对象关系数据模型
   - 半结构化数据模型

> 关系模型的数据结构
>
> - 关系：一个关系对应通常说的一张表

- 元组：表中的一行
- 属性：表中的一列
- **码：** 码键 （表中的某个属性组，可以唯一确定一个元组）
- 域：域是一组具体数据类型的值的集合。属性的取值范围来自与某个域
- 分量：元组中的一个属性值
- 关系模式： ***关系名（属性1，属性2，……，属性n）***

1.3 数据库系统的结构

三级模式结构

1. 模式

模式也称为**逻辑模式**，是数据库中全体数据的逻辑结构和特征的描述，是所有用户的公共数据视图

2. 外模式

外模式也成为**子模式或用户模式**，数据库用户（包括应用程序员和最终用户）能够看见和使用的局部数据的逻辑结构和特征的描述，是数据库用户的数据视图，是与某一应用有关的数据的逻辑表示

3. 内模式

内模式也称**存储模式**，一个数据库只有一个内模式。它是数据物理结构和存储方式的描述，是数据在数据库内部的组织方式。

第二章 关系数据库

2.1 关系数据结构及形式化定义

2.1.1 关系

1. **候选码：** 某一属性组的值能唯一标识一个元组，而其子集不能，则称该属性组为候选码
2. **主码：** 若一个关系有多个候选码，则选定其中一个为主码
3. **主属性：** 候选码的诸属性成为主属性，其他为 *非主属性或非码属性*

> 在最简单情况下，候选码只包含一个属性

> 在最极端情况下，关系模式的所有属性是这个关系的候选码，称为**全码**

2.1.2 关系模式

> 关系的描述称为关系模式，形象地表示为 *R(U,D,DOM,F)* 【五元组】

> 其中 *R* 为关系名， *U* 为组成该关系的属性名集合， *D* 为 *U* 中属性所来自的域，*DOM* 为属性向域的映像集合， *F* 为属性间数据的依赖关系集合。

2.2 关系操作

> **关系模式中常用的关系操作：** 查询（query），插入（insert），删除（delete），修改（update）

> **查询操作：** 选择，投影，并，差，笛卡尔积（5种基本操作）

> **关系数据语言：**
>
> > 关系代数语言（ISBL）
> >
> > 关系演算语言
> >
> > > 元组关系演算语言（ALPHA,QUEL）
> > >
> > > 域关系演算语言（QBE）
> >
> > 具有关系代数和关系演算双重特点的语言（SQL）

2.3 关系的完整性

> 关系模型中有三类完整性约束：**实体完整性、参照完整性、用户定义的完整性**

2.3.1 实体完整性

> 若属性（指一个或一组属性）*A* 是基本关系 *R* 的主属性，则 *A* 不能取空值

2.3.2 参照完整性

> 设 *F* 是基本关系 *R* 的一个或一组属性，但不是关系 *R* 的码，*K S* 是基本关系 *S* 的主码。如果 *F* 与 *K S* 相对应，则称 *F* 是 *R* 的外码，并称基本关系 *R* 为参照关系，基本关系 *S* 为被参照关系或目标

> 关系。

2.3.3 用户定义的完整性

> 针对某一具体关系数据库的约束条件

2.4 关系代数

集合运算符：

‘∪’：并

‘-’：差

‘∩’：交

‘×’：笛卡尔积

专门的关系运算符：

选择： σ

投影： π

连接：∞ 或者 θ

> 等值连接：θ为“=”的连接运算
>
> 自然连接：
>
> 非等值连接：
>
>  连接
>
> 除：÷

第三章 关系数据库标准语言SQL

> DQL（数据查询语言）: 查询语句，凡是select语句都是DQL。
> DML（数据操作语言）： insert delete update，对表当中的数据进行增删改。
> DDL（数据定义语言）： create drop alter，对表结构的增删改。

> TCL（事务控制语言）： commit提交事务，rollback回滚事务。(TCL中的T是Transaction)
> DCL（数据控制语言）: grant授权、revoke撤销权限等。

3.1 数据定义

```
create table 表名(
        字段名1 数据类型,
        字段名2 数据类型,
        字段名3 数据类型,
        ....
);
```

3.2 数据查询

```
select          5
        ...
from            1
        ...
where           2
        ...
group by        3
        ...
having          4
        ...
order by        6
        ...
limit           7
        ...;
```

```
1. SELECT Sno,Sname FROM SC WHERE 课程号='002';

2. SELECT Sname FROM Student WHERE Sdept='信息';

3. SELECT Sno FROM SC;

4. SELECT Sname,Sage,Sdept FROM Student WHERE Sage NOT BETWEEN 21 AND 25;

5. SELECT Sname,Sage FROM Student WHERE Sage<23;

6. SELECT MAX(Grade) FROM SC WHERE Cno='003';

7. SELECT Student,Sno,Sname,Cname,Grade
FROM Student,SC,Course
WHERE SC.Sno=Student.Sno AND SC.Cno=Course.Cno AND Sdept='计算机';

8. SELECT Student.Sno,Sname,Sdept,Grade
FROM Student,SC,Course
WHERE Student.Sno=SC.Sno AND Course.Cno=SC.Cno AND Grade>90 AND
Cname='数据库';

9. SELECT Sno,Sname,Sdept
FROM Student
WHERE Sno NOT IN
(SELECT Sno FROM SC WHERE Cno='001');

10. SELECT Sno, Sname, Sdept
FROM Student
WHERE Sno IN
(SELECT X1.Sno FROM SC X1, SC X2
WHERE X1.Sno=X2.Sno AND X1.Cno='005' AND X2.Cno='008');

11. SELECT Sno,AVG(Grade)
FROM SC
GROUP BY Sno
```

```
HAVING AVG(Grade)>80;

12. SELECT Sname,Cname,Grade
FROM Student,Course,SC
WHERE Student.Sno=SC.Sno AND Course.Cno = SC.Cno;

13. SELECT Sname
FROM Student
WHERE Sdept IN
(SELECT Sdept
FROM Student
WHERE Sname='刘邦');

14. SELECT COUNT(*) FROM SC WHERE Cno='006';

15. SELECT Sno,Sname
FROM Student
WHERE Sno NOT IN
(SELECT Sno FROM SC WHERE Cno='007');

16. SELECT Sno,Sname
FROMS Student
WHERE Sno IN
(SELECT X1.Sno FROM SC X1, SC X2
WHERE X1.Sno=X2.Sno AND X1.Cno='008' AND X2.Cno='009');
```

3.3 数据更新

3.3.1 插入数据

> 语法格式：
>
> insert into 表名(字段名1,字段名2,字段名3,....)
>
> values(值1,值2,值3,....)
>
> 要求：字段的数量和值的数量相同，并且数据类型要对应相同。

3.3.2 修改数据

语法格式：
`update 表名 set 字段名1=值1,字段名2=值2... where 条件;`
注意：没有条件整张表数据全部更新。

3.3.3 删除数据

语法格式：
`delete from 表名 where 条件;`
注意：没有条件全部删除。

3.4 空值的处理

空值：NULL

判断空值：IS NULL 或者 IS NOT NULL

> 空值与另一个值（包括空值）的算数运算为空值；
>
> 空值与另一个值（包括空值）的比较运算为 UNKNOWN
>
> 有了 UNKNOWN 后，传统的逻辑运算中的二值逻辑（ TRUE,FALSE ）就拓展成了三值逻辑

3.5 视图

3.5.1 什么是视图?

站在不同的角度去看到数据。（同一张表的数据，通过不同的角度去看待）。

3.5.2 怎么创建视图？怎么删除视图?

```
create view myview as select empno,ename from emp;
drop view myview;
```

注意：只有DQL语句才能以视图对象的方式创建出来。

3.5.3 对视图进行增删改查，会影响到原表数据。

> （通过视图影响原表数据的，不是直接操作的原表）
>
> 可以对视图进行CRUD操作。

3.5.4 面向视图操作

```
mysql> select * from myview;
+-------+--------+
| empno | ename  |
+-------+--------+
|  7369 | SMITH  |
|  7499 | ALLEN  |
|  7521 | WARD   |
|  7566 | JONES  |
|  7654 | MARTIN |
|  7698 | BLAKE  |
|  7782 | CLARK  |
|  7788 | SCOTT  |
|  7839 | KING   |
|  7844 | TURNER |
|  7876 | ADAMS  |
|  7900 | JAMES  |
|  7902 | FORD   |
|  7934 | MILLER |
+-------+--------+
```

```
create table emp_bak as select * from emp;
create view myview1 as select empno,ename,sal from emp_bak;
update myview1 set ename='hehe',sal=1 where empno = 7369; // 通过视图修改原表数据。
delete from myview1 where empno = 7369; // 通过视图删除原表数据。
```

3.5.5 视图的作用?

视图可以隐藏表的实现细节。保密级别较高的系统，数据库只对外提供相关的视图，java程序员只对视图对象进行CRUD。

1. 视图能够简化用户操作
2. 视图使用户能以多种角度看待同一数据
3. 视图对重构数据库提供了一定程度的逻辑独立性
4. 视图能够对机密数据提供安全保护
5. 适当利用视图可以更清晰地表达查询

第四章 数据库安全性

4.1 数据库安全性概述

> 数据库的安全性是指保护数据库以防止不合法使用所造成的数据泄露、更改或破坏

4.2 存取控制

> 存取控制机制主要包括 **定义用户权限** 和 **合法权限检查** 两部分

> **定义用户权限** 和 **合法权限检查** 机制一起组成了数据库管理系统的存取控制系统

> 1. **自主存取控制：**
>   用户对于不同的数据库对象有不同的数据库权限，不同的用户对同一对象也有不同的权限，而且用户还可将其拥有的存取权限转授给其他用户。

GRANT 例如：

```
Create user ‘testuser’@’localhost’ identified by ‘testpwd’;

Grant select,update on *.* to ‘testuser’@’localhost’;

Flush privileges;
```

REVOKE 例如：

```
revoke insert on book.* from 'rose'@'localhost';

flush privileges;

show grants for 'rose'@'localhost';
```

> 2. **强制存取控制：**
>   每一个数据库对象被标以一定的密级，每一个用户也被授予某一个级别的许可证。对于任意一个对象，只有具有合法许可证的用户才可以存取。

在强制存取控制中，DBMS所管理的全部实体被分为 **主体** 和 **客体** 两大类。

主体是系统中的活动实体，既包括DBMS所管理的实际用户，也包括代表用户的各进程

客体是系统中的被动实体，是受主体操纵的，包括文件、基本表、索引、视图等。

对于主体和客体，DBMS为他们每个实例值指派一个敏感度标记。

敏感度标记：
绝密（Top Secret , TS）、机密(Secret , S)、可信（Confidential , C）、公开（Public , P）

主体的敏感度标记称为许可证级别，客体的敏感度标记称为密级。

仅当主体的许可证级别**大于等于**客体的密级时，该主体才能**读取**相应的客体。

仅当主体的许可证级别**小于等于**客体的密级时，该主体才能**写**相应的客体。

> 强制存取控制是对数据本身进行密级标记，无论数据如何复制，标记和数据是一个不可分割的整体，只有符合密级标记要求的用户才可以操纵数据，从而提供了更高级别的安全性。

4.3 视图机制

> 可以为不同的用户定义不同的视图，把数据对象限制在一定范围内。也就是说，通过视图机制把要保密的数据对无权存取的用户隐藏起来，从而自动对数据提供一定程度的安全保护。

4.4 审计

> 审计功能把用户对数据库的所有操作自动记录下来放入审计日志中。审计员可以利用审计日志监控数据库中的各种行为，重现导致数据库现有状况的一系列事件，找出非法存取数据的人、时间和内容等。

> 数据库安全审计系统提供了一种事后检查的安全机制。安全审查机制将特定用户或者特定对象相关的操作记录到系统审计日志中，作为后续对操作的查询分析和追踪的依据。通过审计机制，可以约束用户可能的恶意操作。

第五章 数据库完整性

> 数据库的完整性是指数据的正确性和相容性。

5.1 实体完整性

(1) 检查主码值是否唯一，如果不唯一则拒绝插入或修改
(2) 检查主码的各个属性是否为空，只要有一个空就拒绝插入或修改

怎么给一张表添加主键约束呢?

```
drop table if exists t_user;
create table t_user(
        id int primary key,  // 列级约束
        username varchar(255),
        email varchar(255)
);
insert into t_user(id,username,email) values(1,'zs','zs@123.com');
insert into t_user(id,username,email) values(2,'ls','ls@123.com');
insert into t_user(id,username,email) values(3,'ww','ww@123.com');
select * from t_user;
+----+----------+------------+
| id | username | email      |
+----+----------+------------+
|  1 | zs       | zs@123.com |
|  2 | ls       | ls@123.com |
|  3 | ww       | ww@123.com |
+----+----------+------------+

insert into t_user(id,username,email) values(1,'jack','jack@123.com');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'

insert into t_user(username,email) values('jack','jack@123.com');
ERROR 1364 (HY000): Field 'id' doesn't have a default value
```

根据以上的测试得出：id是主键，因为添加了主键约束，主键字段中的数据不能为NULL，也不能重复。
主键的特点：不能为NULL，也不能重复。

主键相关的术语

- 主键约束 : primary key
- 主键字段 : id字段添加primary key之后，id叫做主键字段
- 主键值 : id字段中的每一个值都是主键值。

主键有什么作用

- 表的设计三范式中有要求，第一范式就要求任何一张表都应该有主键。
- 主键的作用：主键值是这行记录在这张表当中的唯一标识。（就像一个人的身份证号码一样。）

主键的分类

1. 根据主键字段的字段数量来划分：
    - 单一主键（推荐的，常用的。）
    - 复合主键(多个字段联合起来添加一个主键约束)（复合主键不建议使用，因为复合主键违背三范式。）
2. 根据主键性质来划分：

- 自然主键：主键值最好就是一个和业务没有任何关系的自然数。（这种方式是推荐的）
- 业务主键：主键值和系统的业务挂钩，例如：拿着银行卡的卡号做主键，拿着身份证号码作为主键。（不推荐用）
  最好不要拿着和业务挂钩的字段作为主键。因为以后的业务一旦发生改变的时候，主键值可能也需要随着发生变化，但有的时候没有办法变化，因为变化可能会导致主键值重复。

一张表的主键约束只能有1个。（必须记住）***

使用表级约束方式定义主键：

```
drop table if exists t_user;
create table t_user(
        id int,
        username varchar(255),
        primary key(id)
);
insert into t_user(id,username) values(1,'zs');
insert into t_user(id,username) values(2,'ls');
insert into t_user(id,username) values(3,'ws');
insert into t_user(id,username) values(4,'cs');
select * from t_user;

insert into t_user(id,username) values(4,'cx');
ERROR 1062 (23000): Duplicate entry '4' for key 'PRIMARY'
```

以下内容是演示以下复合主键，不需要掌握：

```
        drop table if exists t_user;
        create table t_user(
                id int,
                username varchar(255),
                password varchar(255),
                primary key(id,username)
        );
        insert .......
```

MySQL提供主键值自增：（非常重要。）

```
drop table if exists t_user;
create table t_user(
        id int primary key auto_increment, // id字段自动维护一个自增的数字，从1开始，以1递增
        username varchar(255)
);
insert into t_user(username) values('a');
insert into t_user(username) values('b');
insert into t_user(username) values('c');
insert into t_user(username) values('d');
insert into t_user(username) values('e');
insert into t_user(username) values('f');
select * from t_user;
```

5.2 参照完整性

关于外键约束的相关术语：

- 外键约束: foreign key
- 外键字段：添加有外键约束的字段
- 外键值：外键字段中的每一个值。
    - 业务背景：
      请设计数据库表，用来维护学生和班级的信息？

第一种方案：一张表存储所有数据

no(pk)	name	classno	classna
1	zs1	101	
2	zs2	101	
3	zs3	102	
4	zs4	102	
5	zs5	102	

缺点：冗余。【不推荐】

第二种方案：两张表（班级表和学生表）

t_class 班级表

cno(pk)	cname
101	北京大兴区经济技术开发区亦庄二中高三1班
102	北京大兴区经济技术开发区亦庄二中高三2班

t_student 学生表

sno(pk)	sname	classno(该字段添加外键约束fk)
1	zs1	101
2	zs2	101
3	zs3	102
4	zs4	102
5	zs5	102

- 将以上表的建表语句写出来：
  t_student中的classno字段引用t_class表中的cno字段，此时t_student表叫做子表。t_class表叫做父表。
  顺序要求：
  删除数据的时候，先删除子表，再删除父表。
  添加数据的时候，先添加父表，在添加子表。
  创建表的时候，先创建父表，再创建子表。
  删除表的时候，先删除子表，在删除父表。

```
drop table if exists t_student;
drop table if exists t_class;

create table t_class(
        cno int,
        cname varchar(255),
        primary key(cno)
);

create table t_student(
        sno int,
        sname varchar(255),
        classno int,
        primary key(sno),
        foreign key(classno) references t_class(cno)
);

insert into t_class values(101,'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx');
insert into t_class values(102,'yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy');

insert into t_student values(1,'zs1',101);
insert into t_student values(2,'zs2',101);
insert into t_student values(3,'zs3',102);
insert into t_student values(4,'zs4',102);
insert into t_student values(5,'zs5',102);
insert into t_student values(6,'zs6',102);
select * from t_class;
select * from t_student;

insert into t_student values(7,'lisi',103);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`
```

* 外键值可以为NULL？
  外键可以为NULL。

* 外键字段引用其他表的某个字段的时候，被引用的字段必须是主键吗？
  注意：被引用的字段不一定是主键，但至少具有unique约束。

5.3 用户定义的完整性

在CREATE TABLE 中定义属性的同时，可以根据应用要求定义属性上的约束条件，即属性值限制，包括:

- 列值非空（ NOT NULL ）
- 列值唯一（ UNIQUE ）

- 检查列值是否满足一个条件表达式（ CHECK 短语 ）

5.4 完整性约束命名子句

1. 完整性约束命名子句
   CONSTRAINT <完整性约束条件名><完整性约束条件>

<完整性约束条件名>包括：

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK短语

```
CREATE TABLE Student(
        Sno NUMERIC(6)
                CONSTRAINT C1 CHECK(Sno BETWEEN 900 AND 999),
        Sname CHAR(20)
                CONSTRAINT C2 NOT NULL,
        Sage NUMERIC(3)
                CONSTRAINT C3 CHECK(Sage<30),
        Ssex CHAR(2)
                CONSTRAINT C4 CHECK(Ssex IN('男','女')),
        CONSTRAINT StudentKey PRIMARY KEY(Sno)
);
```

2. 修改表中的完整性限制

```
ALTER TABLE Student
        DROP CONSTRAINT C1;

ALTER TABLE Student
        ADD CONSTRAINT C1 CHECK(Sno BETWEEN 900 AND 999);
```

5.5 断言

通过声明性断言来指定更具一般性的约束。任何对断言中所涉及关系的操作都会触发DBMS对断言的检查，任何使断言为假的操作都会被拒绝执行。

1. 创建断言的语句格式
   CREATE ASSERTION <断言名><CHECK 子句>

每个断言都被赋予一个名字，<CHECK 子句>中的约束条件与WHERE子句的条件表达式类似。

2. 删除断言的语句格式
   DROP ASSERTION

5.6 触发器

定义：触发器时用户定义在关系表上的一类由事件驱动的特殊过程，又叫做事件-条件-动作规则

```
CREATE TRIGGER <触发器名> < BEFORE | AFTER >
<INSERT | UPDATE | DELETE > ON <表名>
FOR EACH ROW <触发器主体>
```

例如：

```
Create table account(acct_id int ,amount decimal(10,2));
Create trigger ins_num after insert on account
for each row
set @sum=@sum+new.amount;
set @sum=0;
insert into account values(1,1.00),(2,2.00);
select @sum;
```

第六章 关系数据理论

R(U,F) 【三元组】

关系名 *R* 是符号化的元组语义

U 为一组属性

F 是属性组 *U* 上的一组函数依赖

数据依赖是一个关系内部属性与属性之间的一组约束关系。通过属性间值的相等与否体现出来的数据间相关联系。

数据依赖的类型有：函数依赖，多值依赖

一个好的模式应当不会发生插入异常、删除异常和更新异常，数据冗余应尽可能少。

1. **函数依赖：**

> 非平凡的函数依赖
>
> 平凡的函数依赖
>
> 完全函数依赖
>
> 部分函数依赖

2. **码：**

> 设 K 为 $R<U,F>$ 中的属性或属性组合，若 $K \rightarrow U$,则 K 为 R 的 **候选码**

> 如果 U 函数依赖于 K ，即 $K \rightarrow U$ ，则称 K 为 **超码**

> ***候选码是一类特殊的超码，候选码的任意真子集一定不是超码***

> 若候选码多于
> 一个，则选定其中的一个为 **主码**

> 包含在任何一个候选码中的属性称为主属性；不包含在任何候选码中的属性称为非主属性或非码属性。

> 最简单的情况，单个属性是码；最极端的情况，整个属性组是码，称为全码

3. **范式：**

> 关系数据库中的关系是要满足一定要求的，满足不同程度要求的为不同范式。

> 满足最低要求的叫第一范式，简称1NF；在第一范式中满足进一步要求的为第二范式，其余依次类推。

> 一个低一级范式的关系模式通过模式分解可以转换为若干个高一级范式的关系模式的集合，这种过程就叫规范化。

第一范式：任何一张表都应该有主键，并且每一个字段原子性不可再分。
1NF：每一个分量必须是不可分的数据项

第二范式：建立在第一范式的基础之上，所有非主键字段完全依赖主键，不能产生部分依赖。
2NF：每一个非主属性完全依赖于任何一个候选码

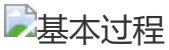
第三范式：建立在第二范式的基础之上，所有非主键字段直接依赖主键，不能产生传递依赖。
3NF：每一个非主属性既不传递依赖于码，也不部分依赖于码

第七章 数据库设计

7.1 数据库设计概述

> 数据库设计的一般定义：
>
> 数据库设计是指对于一个给定的应用环境，构造设计优化的数据库逻辑模式和物理结构，并据此建立数据库及其应用系统，使之能够有效地存储和管理数据，满足各种用户地应用需求，包括信息管理要求和数据操作要求。

> 数据库设计的特点：
>
> > 数据库建设的基本规律——三分技术，七分管理，十二分基础数据
> >
> > 将数据库结构设计和数据处理设计密切结合
>
> 基本步骤：
>
> 1. 需求分析阶段
> 2. 概念结构设计阶段（E-R图）
> 3. 逻辑结构设计阶段（通过E-R图确定主键、外键，形成表）【外模式】
> 4. 物理结构设计阶段（建立索引）【内模式】
> 5. 数据库实施阶段
> 6. 数据库运行和维护阶段

 基本过程

7.2 需求分析

7.3 概念结构设计

7.3.1 E-R模型

- 实体型：矩形，矩形框内写明实体名
- 联系：菱形，菱形框内写明联系名，并用无向边分别与有关实体型连接起来，同时在无向边旁标上联系的类型（1:1,1:n或m:n）

> 联系可以具有属性

- 属性：椭圆形，并用无向边将与其相应的实体型连接起来

> 把参与联系的实体型的数目称为联系的度
>
> 两个实体型之间的联系度为2，也称为二元联系
>
> N个实体型之间的联系度为N，也称为N元联系

7.3.2 概念结构设计

1. 实体与属性的划分原则
    - 对需求分析阶段收集到的数据进行分类、组织
    - 确定实体、实体的属性、实体之间的联系类型
2. E-R图的集成
    - 设计各个子系统的分E-R图
    - 消除冲突，进行集成
    - 设计基本E-R图

划分实体与属性的两条准则：

- 作为属性，不能再具有需要描述的性质。属性必须是不可分的数据项，不能包含其他属性。
- 属性不能与其他实体有联系。E-R图中所表示的联系是实体与实体之间的联系。

E-R图的集成一般需要分两步：

1. 合并。解决各分E-R图之间的冲突，将分E-R图合并，生成初步E-R图
    - 冲突：子系统的E-R图之间的冲突主要有三类：
        1. 属性冲突
        2. 命名冲突
        3. 结构冲突
2. 修改和重构。消除不必要的冗余生成基本E-R图
    - 冗余的数据是指：可有基本数据导出的数据
    - 冗余的联系是指：可有其他联系导出的联系
    - 冗余带来的问题：破坏数据库的完整性，数据库维护困难，应当予以消除

7.4 逻辑结构设计

任务：把概念结构设计阶段设计好的 **基本E-R图** 转换为与选用的DBMS产品所支持的逻辑结构。

目前主要使用关系模型，关系模型的逻辑结构是一组关系模式的集合。

7.4.1 E-R图向关系模型的转换

转换内容：

- 将E-R图转换为关系模型
- 将实体型、实体的属性和实体型之间的联系转化为关系模式

1. 实体型的转换：一个实体型转换为一个关系模式
    - 关系模式的属性：实体的属性
    - 关系模式的码：实体的码
2. 实体型间的1：1联系
    - 可以转换为一个独立的关系模式
        1. 关系模式的属性：与该联系相连的各实体的码以及联系本身的属性
        2. 关系模式的候选码：每个实体的码均是该关系模式的候选码
    - 可以与相连的任意一端对应的关系模式合并
        1. 关系模式的属性：
        > 与某一端关系模式合并，则在该关系模式的属性中加入另一端关系模式的码和联系的属性

        2. 合并后关系模式的码：不变
3. 实体型间的1:n联系
    - 转换为一个独立的关系模式
        1. 关系模式的属性:与该联系相连的各实体的码+联系本身的属性
        2. 关系模式的码:n端的实体的码
    - 与n端对应的关系模式合并
        1. 合并后关系模式的属性:在n端关系模式中+1端关系的码+联系本身的属性
        2. 合并后关系模式的码:不变
        > 可以减少系统模式中的关系个数，一般情况下更倾向于采用这种方法

4. 实体型间的m:n联系
    - 一个m:n联系转换为一个关系模式
        1. 关系的属性：与该联系相连的各实体的码以及联系本身的属性
        2. 关系的码：各实体码的组合
5. 三个或三个以上实体间的一个多元联系
    - 转换为一个关系模式
        1. 关系模式的属性:与该多元联系相连的各实体的码+联系本身的属性
        2. 关系模式的码:各实体码的组合
6. 具有相同码的关系模式可合并
    - 目的:减少系统中的关系个数
    - 合并方法:
        1. 将其中一个关系模式的全部属性加入到另一个关系模式中
        2. 然后去掉其中的同义属性(可能同名也可能不同名)

3. 适当调整属性的次序

7.4.2 数据模型的优化

7.4.3 设计用户子模式

第八章 数据库编程

8.1 嵌入式SQL

8.1.1 用游标的SQL语句

```
#include <iostream>
using namespace std;

EXEC SQL BEGIN DECLARE SECTION;
char HSno[9];
char HSname[20];
char HSex[2];
int HSage;
char Hdept[20];
EXEC SQL END DECLARE SECTION;/*主变量说明*/

long SQLCODE;
EXEC SQL INCLUDE sqlca;/*定义SQL通信区*/

int main(){
        cout<<"Please input the sno:"<<endl;
        cin>>HSno;

        EXEC SQL CONNECT TO bjpowernode@localhost:3306 AS CONN1
        USER "root" USING "root";

        EXEC SQL SELECT Sno,Sname,Ssex,Sage,Sdept
        INTO:HSno,
        :HSname,
        :HSex,
        :HSage,
        :Hdept
        FROM Student WHERE Sno=:HSno;
        /*
        *增加了一个INTO子句，用于保存查询结果
        *前面加“:”标识的变量称为主变量，也就是主语言的变量
        */

    if(sqlca.sqlcode==0){//SQLCA中的SQLCODE==0表示操作成功
                cout<<Sno<<","<<Sname<<","<<Ssex<<","<<Sage<<","<<Sdept;
                cout<<Hno<<","<<HSname<<","<<HSex<<","<<HSage<<","<<Hdept;
        }
        EXEC SQL DISCONNECT CONN1;

        return 0;
}
```

游标：

- 游标是数据库系统为用户开设的一个数据缓冲区，存放SQL语句的执行结果
- 每个游标区都有一个名字，也可以理解为该数据区的指针
- 可以用SQL语句逐一从游标中（指针所指示的位置获取记录），并赋给主变量，并由主语言进一步处理

必须使用游标的SQL语句

- 查询结果为多条记录的SELECT语句
- CURRENT形式的UPDATE、DELETE语句

1. 查询结果为多条记录的SELECT语句

> 使用游标的步骤
>
> 1. 申明游标
>
> ```
> EXEC SQL DECLARE <游标名> CURSOR
>         FOR <SELECT语句>;
> ```
>
> 说明性语句，DBMS并不执行SELECT语句，而是申请一个数据空间，用于存放未来执行SELECT的结构数据集
>
> 2. 打开游标
>
> ```
> EXEC SQL OPEN <游标名>;
> ```
>
> - 打开游标实际上是执行相应的SELECT语句，把查询结果取到缓冲区中
> - 这是游标处于活动状态，指针指向查询结果集中的第一条记录
>
> 3. 推进游标指针并取当前记录
>
> ```
> EXEC SQL FETCH <游标名>
>         INTO <主变量>[<指示变量>]
>                 [,<主变量>[<指示变量>]]...;
> ```
>
> 指定方向推动游标指针，同时将缓冲区中的当前记录取出来送至主变量供主语言进一步处理
>
> 4. 关闭游标
>
> ```
> EXEC SQL CLOSE <游标名>;
> ```
>
> - 关闭游标，释放结果集占用的缓冲区及其他资源
> - 游标被关闭后，就不再和原来的查询结果集相联系
> - 被关闭的游标可以再次被打开，与新的查询结果相联系

2. CURRENT形式的UPDATE、DELETE语句

> - 如果只想修改或删除其中某个记录

- 用带游标的SELECT语句查出所有满足条件的记录。从中进一步找出要修改或删除的记录
- 用CURRENT形式的UPDATE语句和DELETE语句修改或删除
- UPDATE语句和DELETE语句中要用子句

  `WHERE CURRENT OF <游标名>`

  表示修改或删除的是最近一次取出的记录，即游标指针指向的记录

8.1.2 不用游标的SQL语句

1. 查询结果为单条记录的SELECT语句
2. 非CURRENT形式的增删改语句

8.1.3 动态SQL

在编译阶段无法获得完整的SQL语句，需要在程序执行时才能够确定的SQL语句，称为“动态嵌入式SQL”。

1. 使用SQL语句主变量

- 程序主变量包含的内容是SQL语句本身的内容，而不是原来保存数据的输入和输出变量
- SQL语句主变量在程序执行期间可以赋值为不同的SQL语句，然后执行

2. 动态参数
   动态参数是SQL语句中的可变元素，使用参数符号（?）表示该位置的数据在运行时设定。通过PROPARE语句准备主变量和执行语句EXECUTE绑定数据或主变量来完成。

- 声明SQL语句主变量
- 准备SQL语句（PREPARE）
- 执行准备好的语句（EXECUTE）

8.2 过程化SQL

常量变量的定义

流程控制

- 条件控制

```
IF condition THEN
        ...;
END IF;

IF condition THEN
        ...;
ELSE
        ...;
END IF;
```

- 循环控制

```
LOOP
        ...;
END LOOP;

WHILE conditon LOOP
        ...;
END LOOP;
```

- 错误处理

8.3 存储过程和函数

```
delimiter //
create procedure proc2(in cid char(18), out num int)
begin
delete from reader_info where card_id = cid;
select count(card_id) into num from reader_info;
end//
delimiter;
select * from reader_info;
call procedure proc2('210210199901011111', @num);
select @num;
```

```
delimiter //
create function show_level(cid char(18))
        returns varchar(10)
begin
        declare lev varchar(10);
        declare money decimal(7,3);
        select balance into money from reader_info where card_id=cid;
        if money>=500 then
        set lev='金牌会员';
        elseif money>=300 then
        set lev='高级会员';
        elseif money>=200 then
        set lev='普通会员';
        else
        set lev='非会员，余额不足';
        end if;
        return lev;
end//
delimiter ;
select show_level(card_id),balance from reader_info;
```

第九章 关系查询处理和查询优化

9.1 关系数据库系统的查询处理

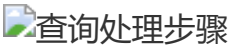
9.1.1 查询处理步骤

1. 查询分析：词法、语法分析
2. 查询检查
    - 有效性检查：语句中的数据库对象，如关系名、属性名是否存在和有效
    - 视图转换：对视图操作，采用视图消解法转换成对基本表的操作
    - 安全性检查：根据数据字典中的用户权限对用户的存取权限进行检查
    - 完整性初步检查：根据数据字典中存储的完整性约束定义，对句子进行检查

    > 检查通过后，将SQL查询语句转换成内部表示，一般用 **查询树**（语法树）

3. 查询优化：
    - 分类
        - 代数优化：指针对关系代数表达式的优化
        - 物理优化：指存取路径和底层操作算法的选择
    - 选择依据
        - 基于规则

        - 基于代价
        - 基于语义
4. 查询执行
    - 依据优化器得到的执行策略生成查询执行计划
    - 代码生成器生成执行查询计划的代码
    - 两种执行方法
        - 自顶向下
        - 自底向上

 查询处理步骤

9.1.2 实现查询操作的算法实例

1. 选择操作的实现
    - 简单的全盘扫描算法(table scan)【适合小表】
    - 索引扫描算法(index scan)【B+树，HASH】
2. 连接操作的实现：查询处理中最常用也最耗时的操作之一
    - 嵌套循环算法：最简单可行的算法
    - 排序-合并算法：等值连接常用算法
    - 索引连接算法
    - hash join 算法：划分阶段（创建阶段）+试探阶段（连接阶段）

9.2 关系数据库系统的查询优化

9.2.1 查询优化概述

1. 优点
    - 是DBMS实现的关键技术又是优点所在
    - 减轻了用户对于系统底层选择存取路径的负担
    - 用户可关注查询的正确表达上，无需考虑查询的执行效率
2. 总目标
    - 选择有效的策略
    - 求得给定关系表达式的值
    - 使得查询代价最小（实际上是较小）

9.2.2 一个实例

……

9.3 代数优化

9.3.1 关系代数表达式等价变换规则

- 等价的含义：指用相同的关系代替两个表达式中相应的关系所得到的结果是相同的
- 什么样的变换一定是等价的。（避免去证明等价）
- 两个关系表达式E1和E2是等价的，可记为E1=E2

1. 连接、笛卡尔积交换律
2. 连接、笛卡尔积结合律
3. 投影的串接定律
4. 选择的串接定律
5. 选择与投影操作的交换律
6. 选择与笛卡尔积的交换律
7. 选择与并的分配律
8. 选择与差的分配律
9. 选择对自然连接的分配律
10. 投影与笛卡尔积的分配律
11. 投影与并的分配律

9.3.2 查询树的启发式优化

- 什么样的变换一定是“好”的，即执行代价更小

1. 选择优化尽可能先做
2. 投影和选择同时进行
3. 投影同其前后的双目运算结合起来
4. 把某些选择同在它前面要执行的笛卡尔积结合起来成为一个连接运算
5. 找出公共子表达式

9.4 物理优化

物理优化就是要选择高效合理的操作算法或存取路径，求得更好的查询计划

9.4.1 基于启发式规则的存取路径选择优化

1. 选择操作的启发式规则
2. 连接操作的启发式规则

9.4.2 基于代价估算的优化

1. 统计信息
2. 代价估算实例

第十章 数据库恢复技术

10.1 事务的基本概念

1. 事务
   - 事务是用户定义的一个数据库操作序列，这些操作要么全做，要么全不做，是一个不可分割的工作单位。
   - 事务和程序的区别：
     - 在关系数据库中，一个事务可以是一条SQL语句，一组SQL语句或者整个程序
     - 一个程序通常包含多个事务
   - 事务是恢复和并发控制的基本单位
2. 事务的ACID特征
   - 原子性（**Atomicity**）
     - 事务是数据库的逻辑工作单位：事务中包括的诸操作要么都做，要么都不做
   - 一致性（**Consistency**）：事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态
     1. 一致性状态：数据库中只包含成功事务提交的结果
     2. 不一致状态：
        - 数据库系统运行中发生故障，有些事务尚未完成就被迫中断
        - 这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这是数据库就处于一种不正确的状态
   - 隔离性（**Isolation**）：一个事务的执行不能被其他事务干扰
     - 一个事务内部的操作及使用的数据对其他并发事务是隔离的
     - 并发执行的各个事务之间不能互相干扰
   - 持续性（永久性）（**Durability**）
     - 一个事务一旦提交，它对数据库中数据的改变就应该是永久性的
     - 接下来的其他操作或故障不应该对其执行结果有任何影响

> 保证事务ACID特性是事务管理的重要任务
>
> 破坏事务ACID特性的因素
>
> 1. 多个事务并行运行时，不同事务的操作交叉执行

> - DBMS必须保证多个事务的交叉运行不影响这些事务的隔离性
> 2. 事务在运行过程中被强行停止
> - DBMS必须保证被强行终止的事务对数据库和其他事务没有任何影响

10.2 数据库恢复概述

- 故障时不可避免的
  - 计算机硬件故障
  - 软件的错误
  - 操作员的失误
  - 恶意的破坏
- 故障的影响
  - 运行事务非正常中断，影响数据库中数据的正确性
  - 破坏数据库，全部或部分丢失数据
- 数据库的恢复
  - DBMS必须具有把数据库从错误状态恢复到某一已知的正确状态（一致状态或完整状态）的功能，这就是数据库的恢复管理系统对故障的对策
- 恢复子系统是DBMS的一个重要组成部分
- 恢复技术是衡量系统优劣的重要指标

10.3 故障的类型

1. 事务内部的故障

- 有的是可以通过事务程序本身发现的
- 有的是非预期的，不能由事务程序处理的
  - 运算溢出
  - 并发事务产生死锁而被选中撤销该事务
  - 违反了某些完整性限制而被终止等等
- 事务故障意味着
  - 事务没有达到预期的终点（COMMIT或显式的ROLLBACK）
  - 数据库可能处于不正确状态
- 事务故障的恢复：**事务撤销（UNDO）**
  - 强行回滚（ROLLBACK）该事务
  - 撤销该事务已经作出的任何对数据库的修改，使得该事务好像根本没有启动一样

2. 系统故障：称为**软故障**，指造成系统停止运转的任何事件，使得系统要重新启动

- 特定类型的硬件错误（CPU故障）
- 操作系统故障
- DBMS代码错误
- 系统断电

> 系统故障的影响：
>
> - 整个系统的正常运行突然被破坏
> - 所有正在运行的事务都非正常终止
> - 内存中数据库缓冲区的信息全部丢失
> - 不破坏数据库

> 发生系统故障时，一些尚未完成的事务的结果可能已送入物理数据库，造成数据库可能处于不正确状态
>
> - 恢复策略：系统重新启动时，恢复程序让所有非正常终止的事务回滚，强行撤销（UNDO）所有未完成事务

> 发生系统故障时，有些已完成的事务可能一部分甚至全部留在缓冲区，尚未写回到磁盘上的物理数据库中，系统故障使得这些事务对修改部分或全部丢失
>
> - 恢复策略：系统重新启动时，恢复程序需要重做
>
> > 系统故障的恢复需要做两件事情：
> >
> > 1. 撤销所有未完成的事务
> > 2. 重做所有已提交的事务

3. 介质故障：称为**硬故障**，指外存故障

- 磁盘损坏
- 磁头碰撞
- 瞬时强磁场干扰

> 介质故障破坏数据库或部分数据库，并影响正在存取这部分数据的所有事务
> 介质故障比前两类故障的可能性小得多，但破坏性大得多

4. 计算机病毒
    - 一种人为的故障或破坏，时一些恶作剧者研制的一种计算机程序
    - 可以繁殖、传播，造成对计算机系统包括数据库的危害

- 计算机病毒已成为计算机系统的主要威胁，自然也是数据库系统的主要威胁
- 数据库一旦被破坏仍要用恢复技术把数据库加以恢复

各类故障对数据库的影响有两种可能性

- 一是数据库本身被破坏
- 二是数据库没有被破坏，但数据可能不正确，这是由于事务的运行被非正常终止造成的。
- 恢复操作的基本原理：**冗余**
  - 利用存储在系统别处的**冗余数据**来重建数据库中已被破坏或不正确的那部分数据
- 恢复的实现技术：复杂
  - 一个大型数据库产品，恢复子系统的代码要占全部代码的10%以上

10.4 恢复的实现技术

恢复机制涉及的关键问题

1. 如何建立冗余数据
   - 数据转储（Backup）
   - 登记日志文件（Logging）
2. 如何利用这些冗余数据实施数据库恢复

10.4.1 数据转储

1. 什么是数据转储：指数据库管理员定期地将整个数据库复制到磁带、磁盘或其他存储介质上保存起来的过程。
   - 备用的数据文本成为**后备副本**或**后援副本**
   - 数据库遭到破坏后可以将后备副本重新装入
   - 重装后备副本只能将数据库恢复到转储时的状态
   - 要想恢复到故障发生时的状态，必须重新运行自转储以后的所有更新事务
2. 转储方法
   - 静态转储与动态转储
   - 海量转储与增量转储
   - 转储方法小结
   1. 静态转储
      - 在系统中无事务运行时进行的转储操作
      - 转储开始时数据库处于一致性状态
      - 转储期间不允许对数据库的任何存取、修改活动
      - 得到一个数据一致性的副本
      - 优点：简单
      - 缺点：降低了数据库的可用性
        - 转储必须等待正运行的用户事务结束
        - 新的事务必须等转储结束

2. 动态转储
    - 转储操作与用户事务并发执行
    - 转储期间允许对数据库进行存取或修改
    - 优点
        - 不用等待正在运行的用户事务结束
        - 不会影响新事务的运行
    - 缺点
        - 不能保证副本中的数据正确有效
    - 利用动态存储得到的副本进行故障恢复
        - 需要把动态转储期间的各事务对数据库的修改活动登记下来，建立日志文件
        - 后备副本加上日志文件就能把数据库恢复到某一时刻的正确状态
3. 海量转储与增量转储
    - 海量转储：每次转储全部数据库
    - 增量转储：只转储上次转储后更新过的数据
    - 海量转储与增量转储比较
        - 从恢复角度看，使用海量转储得到的后备副本进行恢复往往更方便
        - 如果数据库很大，事务处理又十分频繁，则增量转储方式更实用有效

10.4.2 登记日志文件

1. 日志文件的格式和内容

- 什么是日志文件：
    - 日志文件是用来记录事务对数据库的更新操作的文件
- 日志文件的格式
    - 以记录为单位的日志文件
        - 各个事务的开始、结束标记

> T1 BEGIN TRANSACTION
> T1 COMMIT
> T1 ROLLBACK

        - 各个事务的所有更新操作
            - 事务标识（标明是哪个事务）
            - 操作类型（插入、修改或删除）
            - 操作对象（记录ID、Block NO.）
            - 更新前数据的旧值（对插入操作而言，此项为空值）
            - 更新后数据的新值（对删除操作而言，此项为空值）

> T1 U AA 18 20
> T1 I TU 1

> T1 D TV 20

- 以数据块为单位的日志文件:
  - 每条日志记录内容
    - 事务标识
    - 被更新的数据块

2. 日志文件的作用
   - 用途
     - 进行事务故障恢复
     - 进行系统故障恢复
     - 协助后备副本进行介质故障恢复
   - 具体作用
     - 事务故障恢复和系统故障恢复必须用日志文件
     - 在动态转储方式中必须建立日志文件，后备副本和日志文件结合起来才能有效地恢复数据库
     - 在静态转储方式中，也可以建立日志文件
       - 故障恢复时重新装入后备副本把数据库恢复到转储时的正确状态
       - 利用日志文件，重做已完成事务，撤销未完成的事务
       - 不必重新运行那些已完成的事务程序就可把数据库恢复到故障前某一时刻的正确状态
3. 登记日志文件

- 为保证数据库时可恢复的，登记日志文件时必须遵循两条原则
  - 登记的次序严格按并发事务执行的时间次序
  - 必须先写日志文件，后写数据库
    - 写日志文件操作：把表示这个修改的日志记录写到日志文件中
    - 写数据库操作：把对数据的修改写到数据库中

10.5 恢复策略

10.5.1 事务故障的恢复

- 事务故障：事务在运行至正常终止点前被终止
- 恢复方法：由恢复子系统利用日志文件撤销此事务已对数据库进行的修改
- 事务故障的恢复由系统自动完成，对用户是透明的，不需要用户干预

（1）反向扫描日志文件（从最后向前扫描日志文件），查找该事务的更新操作
（2）对该事务的更新操作执行逆操作

（3）继续反向扫描日志文件，查找该事务的其他更新操作，并做同样处理
（4）如此处理下去，直至读到此事务的开始标记，事务故障恢复就完成了

10.5.2 系统故障的恢复

- 系统故障造成数据库不一致状态的原因
  - 未完成事务对数据库的更新可能已写入数据库
  - 已提交事务对数据库的更新可能还留在缓冲区没来得及写入数据库
- 恢复方法
  - UNDO 故障发生时未完成的事务
  - REDO 已完成的事务
- 系统故障的恢复由系统在重新启动时自动完成，不需要用户干预

（1）正向扫描日志文件（从头扫描日志）

- 重做（REDO）队列：在故障发生前已经提交的事务
  - 这些事务既有BEGIN TRANSACTION记录，也有COMMIT记录
- 撤销(UNDO)队列：故障发生时尚未完成的事务
  - 这些事务只有BEGIN TRANSACTION记录，无相应的COMMIT记录

（2）对撤销（UNDO）队列事务进行撤销（UNDO）处理

- 反向扫描日志文件，对每个撤销事务的更新操作执行逆操作
- 将日志记录中“更新前的值”写入数据库

（3）对重做（REDO）队列事务进行重做（REDO）处理

- 正向扫描日志文件，对每个重做事务重新执行登记的操作
- 将日志记录中“更新后的值”写入数据库

10.5.3 介质故障的恢复

1. 重装数据库
2. 重做已完成的事务

- 恢复步骤
  - （1）装入最新的后备数据库副本（离故障发生时刻最近的转储副本），使数据库恢复到最近一次转储时时一致性状态。
    - 对于静态转储的数据库副本，装入后数据库即处于一致性状态
    - 对于动态转储的数据库副本，还须装入转储时刻的日志文件副本，利用恢复系统故障的方法（即REDO+UNDO），才能将数据库恢复到一致性状态

- （2）装入有关的日志文件副本（转储结束时刻的日志文件副本），重做已完成的事务
  - 首先扫描日志文件，找出故障发生时已提交的事务的标识，将其记入重做队列
  - 然后正向扫描日志文件，对重做队列中的所有事务进行重做处理。将日志记录中“更新后的值”写入数据库

10.5.4 小结

- 事务故障的恢复（**UNDO**）
- 系统故障的恢复（**UNDO+REDO**）
- 介质故障的恢复（**重装后援副本+REDO**）

10.6 具有检查点的恢复技术

10.7 数据库镜像

10.8 小结

> 保证数据一致性是对数据库的最基本的要求。
>
> 事务是数据库的逻辑工作单位，只要DBMS能够保证系统中一切事务的ACID特性，即事务的原子性、一致性、隔离性和持续性，也就保证了数据库处于一致状态。
>
> 为了保证事务的ACID，DBMS必须对事务故障、系统故障和介质故障进行恢复。
>
> 数据转储和登记日志文件是恢复中最经常使用的技术。
>
> 恢复的基本原理就是利用存储在后备副本、日志文件和数据库镜像中的冗余数据来重建数据库。

第十一章 并发控制

事务可以一个一个地串行执行，即每个时刻只有一个事务运行

在单处理机系统中，事务的并行执行实际上是这些并行事务的并行操作轮流交叉运行

11.1 并发控制概述

- 事务是并发控制的基本单位

- 并发控制机制的任务
  - 对并发操作进行正确调度
  - 保证事务的隔离性
- 并发操作带来的数据不一致性
  - 丢失修改
  - 不可重复读
  - 读“脏”数据
- 并发控制就是要用正确的方式调度并发操作，是一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致性
- 并发控制的主要技术
  - 封锁
  - 时间戳
  - 乐观控制法
  - 多版本并发控制

11.2 封锁

- 封锁就是事务T在对某个数据对象（例如表、记录等）操作之前，先向系统发出请求，对其加锁
- 加锁后事务T就对该数据对象有了一定的控制，在事务T释放它的锁之前，其他的事务不能更新此数据对象
- 封锁是实现并发控制的一个非常重要的技术
- **基本封锁类型：**
  - 排他锁（X锁），事务T对数据对象A加上X锁，只允许T读取和修改A，其他事务不能对A加任何形式的锁
  - 共享锁（S锁），事务T对数据对象A加上S锁，T可以读取A但不能修改A，其他事务只能对T加S锁，而不能加X锁

11.3 封锁协议

- 什么是封锁协议
  - 在运用X锁和S锁对数据对象加锁时，需要约定一些规则，这些规则称为封锁协议
    - 何时申请X锁或S锁
    - 持锁时间
    - 何时释放
  - 对封锁方式规定不同的规则，就形成了各种不同的封锁协议，在不同程度傻瓜保证并发执行的正确调度
- 三级封锁协议

1. 一级封锁协议
    - 事务T在修改数据R之前必须对其加X锁，直到事务结束才释放
        - 正常结束（COMMIT）
        - 非正常结束（ROLLBACK）
    - 一级封锁协议可防止丢失修改，并保证事务T是可恢复的
    - 不能保证可重复读，不读“脏”数据
2. 二级封锁协议
    - 一级封锁协议加上事务T在读取数据R之前必须对其加S锁，读完后即可释放S锁
    - 二级封锁协议可以防止丢失修改和读“脏”数据
    - 不能保证可重复读
3. 三级封锁协议
    - 一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，直至事务结束才释放。
    - 三级封锁协议可防止丢失修改、读脏数据和不可重复读

- 不同的封锁协议使事务达到的一致性级别不同
    - 封锁协议级别越高，一致性程度越高

11.4 活锁和死锁

- 封锁技术可以自由有效地解决并行操作地一致性问题，但也带来一些新的问题
    - 死锁
    - 活锁