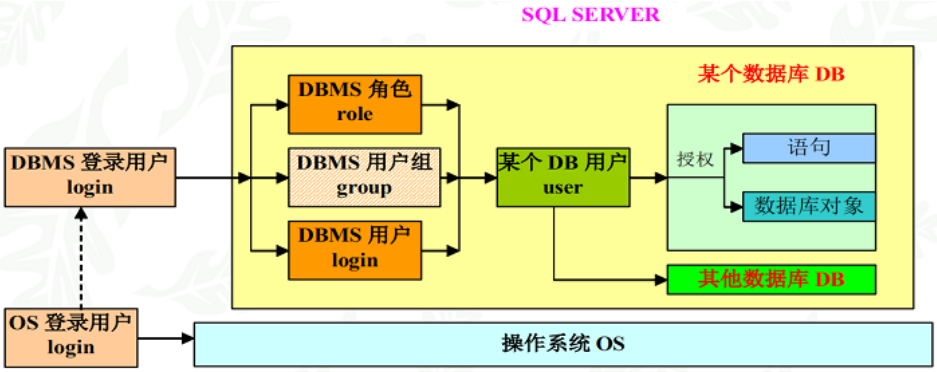
数据库的保护

笔记本： database_theory
创建时间： 2021/6/26 19:14 **更新时间：** 2021/7/2 15:48
作者： 134exetj717

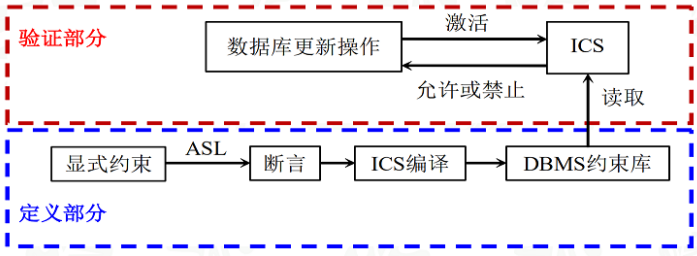
既然谈论到数据库的保护，首先我们需要知道，

- 数据库破坏的类型有哪些?
    - 非法用户：未经授权恶意访问
    - 非法数据：不符合规定的数据
    - 各种故障：硬件故障、用户失误等
    - 多用户的并发访问：访问同一个内存时产生的冲突
- 既然有破坏，那我们应该从哪几个方面去保护呢?
    - 利用权限机制：只允许具有合法权限的用户存取允许的数据
    - 利用完整性约束防止非法数据进入数据库
    - 提供故障恢复能力
    - 提供并发控制机制：保证多用户并发访问的顺利进行
- 由数据库的概述我们知道，DBMS是建立在操作系统以上的，因而在安全问题上，两者有什么关系呢?
    - DBMS应该有自己的安全体系
    - 操作系统对DBMS的基础保护
    - 操作系统用户不一定是DBMS用户：因为两者各有自己的安全体系，因此操作系统的用户想访问DBMS，必须由DBA将其设置成DBMS用户。
- 既然要做到保护，那么数据库安全性的目标是什么呢?
    - 私密性：不能对未授权的用户公开
    - 完整性：只有授权用户才允许修改数据
    - 可用性：授权用户不能被拒绝访问
- 既然数据库有自己的安全体系，那么在访问控制的过程中，又有哪些方法呢?
    - 自主式访问控制（DAC）：通过授权和撤权方式实现
    - 强制访问控制（MAC）：不由某个用户改变-->客体（Object，如：各种DB对象）和主体（Subject，如：用户、计算机、进程等）分别具有相应的安全级别(Security Class和Clearance)，如：绝密（Top-secret，TS）、机密（ Secret，S）、秘密（ Confidential，C）和无密级（Unclassified，U），主体只有在满足一定规则如（”下读”，主体的安全级别必须高于所读客体的安全级别；“上写”，主体的安全级别必须低于所写入的客体的安全级别）才能访问某个客体。）
- 我们可能好奇，自主式访问控制如何实现授权与撤权呢?
    - DAC的权限分为两种，分别是：
        - 角色权限：通过给角色授权，并为用户分配角色，则用户的权限为其角色权限之和。（角色权限由DBA授予）
        - 数据库对象权限：不同的数据库对象，可提供给用户不同的操作。对数据库的操作即为其权限。（该权限由DBA或该对象的拥有者（owner）授予用户）
- 经过以上的学习，我们已经了解了数据库具体是怎么进行安全保护，以及展开访问控制的。但是，我们却没有一个实例来很好的解释，接下来，将会用sqlserver作为实例演示安全体系的设置，那么，安全体系是怎样的呢?



- 安全体系由三级组成：分别是DBMS或数据库服务器级、数据库级、语句与对象级。
- 其安全策略为：层层递进。要先访问数据库服务器，然后访问数据库，最后访问某个命令语句。其中，每次访问都要拥有相应的权限。
- 数据库服务器的登录用户为 sa，即系统管理员（system administrator，SA）。
- sqlserver 角色定义：
  - 服务器级的系统角色
  - 数据库级系统角色：public角色只具备访问数据库的权限，其中 dbo 是数据库所有者。

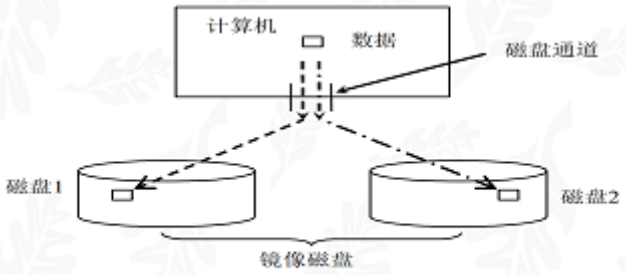
- 对于数据库而言，以上内容都是基于预防非法用户的安全保护系统，那么针对非法数据，我们又该如何应对呢?
- 应对方案是：数据库的完整性约束，简称数据库完整性。分为以下两种，
  - 静态完整性约束：静态约束，关于数据库正确状态的约束。比如，“员工工资只能在1万元以内”。
    - 隐式约束：隐含于数据模型中的完整性约束。比如，主键只能有一个，且不能为空值
    - 固有约束：数据模型固有的约束
    - 显示约束：根据具体应用需求显式地定义或说明
      - 过程化定义：利用过程（或函数）来定义和验证显示约束。比如，“员工工资不能超过上司工资”，定义一个过程函数，若超过了，则回退事务
      - 断言定义：指数据库状态必须满足的约束条件。为定义约束，提供了断言定义语言（ASL），这样，开发人员就能用断言形式编写数据库的显示约束集合。此外，还会提供完整性控制子系统（ICS），负责约束集合的编译，并且在编译结束后放入约束库中。以后每一次检验，都直接从库中读取数据就好。其原理类似于杀毒软件。如下图：



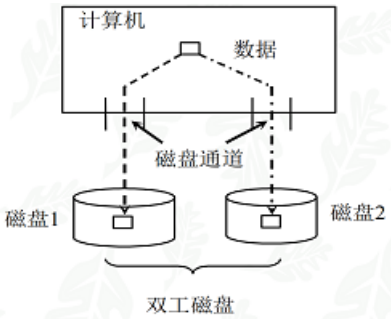
      - 触发器：由操作触发的特殊过程。
  - 动态完整性约束（变迁约束）：动态约束，指数据库状态变化过程的约束。简而言之，数据库从一个正确状态向另一个正确状态的转化过程中必须遵循的约束条件。比如，“员工工资只能涨”
    - 过程化定义、触发器。这两种方式下，开发人员均可以得到改变前后的数据，以便于比较决定是否允许这种数据状态的改变。

数据库完整性约束		定义方式		SQL-92推荐情况	SYBASE和MS SQL SERVER支持情况
静态约束	固有约束	勿需定义设计时满足即可		属性原子性	属性原子性
	隐式约束	数据库定义子语言DDL	表本身的完整性约束	域限制、主键限制、唯一限制和检查限制	缺省、规则、主键限制、唯一限制、检查限制
			表间的约束	外键限制、断言	外键限制、触发器
	显式约束	过程化定义		推荐	存储过程、函数
		断言		推荐	不支持
		触发器		无	支持
动态约束		过程化定义		推荐	存储过程、函数
		触发器		无	支持

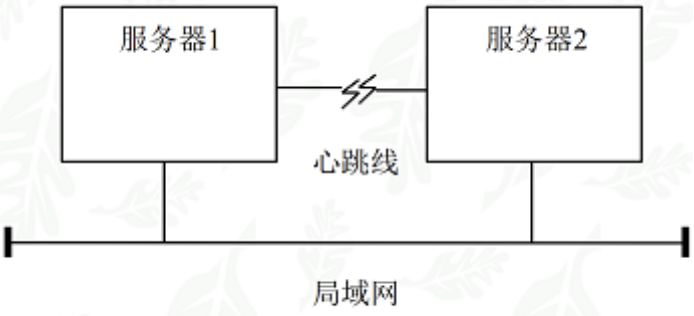
- 基于以上两点，应对非法用户和非法数据，此时，我们又该如何应对故障情况呢？
  - 首先，我们引入“事务”的概念：所有步骤做完，才算做完的工作。若做完，则提交完成；若没做完，则撤销或回退。事务是最小的执行单位。
  - 事务控制分为两类：
    - 隐式的事务控制：默认情况下，DBMS将一个数据库的操作语句当作一个事务来执行
    - 显示的事务控制：涉及多步操作的、由多个操作语句构成的事务，就需要人为地、显式地将这些操作语句组合成一个“事务”
  - 为什么需要显示的事务控制呢？
    - 比如，银行的转账和余额，表面上看这是两件事情，但转念一想，如果转账失败了，回退，但是余额已经增加，是不是不会回退呢？因此，我们必须要撤销转账的同时，撤销余额的增加。
  - 并发控制和故障恢复的基础是什么？---------->事务
  - 如果要应对故障情况，那我们必须遵守以下事务的准则（ACID）：
    - 原子性：事务中所有操作要么全部成功执行，要么都不执行——>不可分割
    - 一致性：事务执行前后，数据库从一个一致状态转到另一个一致状态——>数据始终一致
    - 隔离性：事务间互不干扰——>（主要针对 并发控制）
    - 持久性：对数据库的更新应是持久的——>（针对故障恢复）
- 应对故障情况的方法有很多，故障恢复导论里是怎么讲的呢？
  - 单纯以后备副本为基础的恢复技术：周期性地把数据库的内容储备到磁带上。
  - 以后备副本和日记为基础的恢复技术：前像（BI）和后像（AI），在系统正常运行时记下它们的变化情况。
  - 基于多副本的恢复技术：
    - DBMS是建立在操作系统之上的，同时操作系统提供有该级的可靠性技术，包括镜像磁盘、双工磁盘、镜像服务器、群集系统。



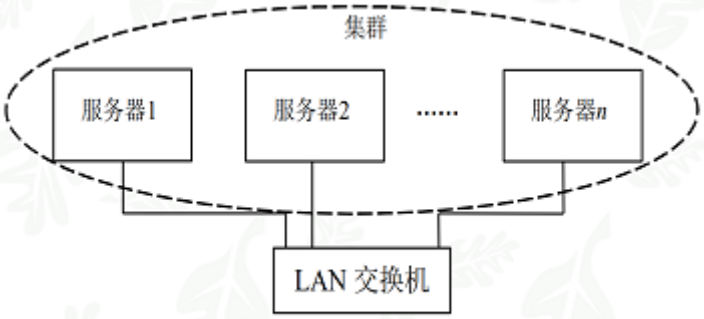
磁盘镜像



磁盘双工



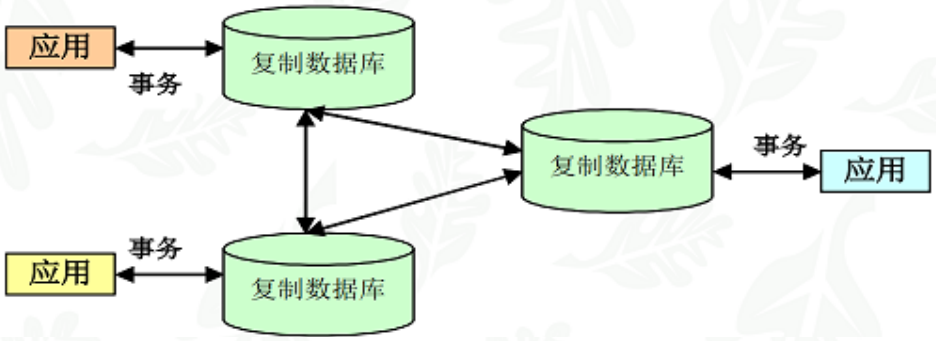
服务器镜像



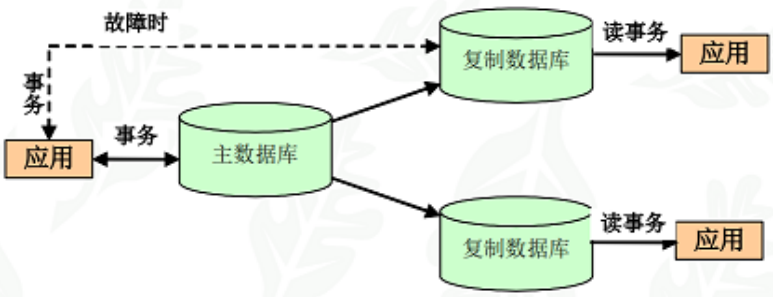
集群服务器

- DBMS也提供了该级的可靠技术，

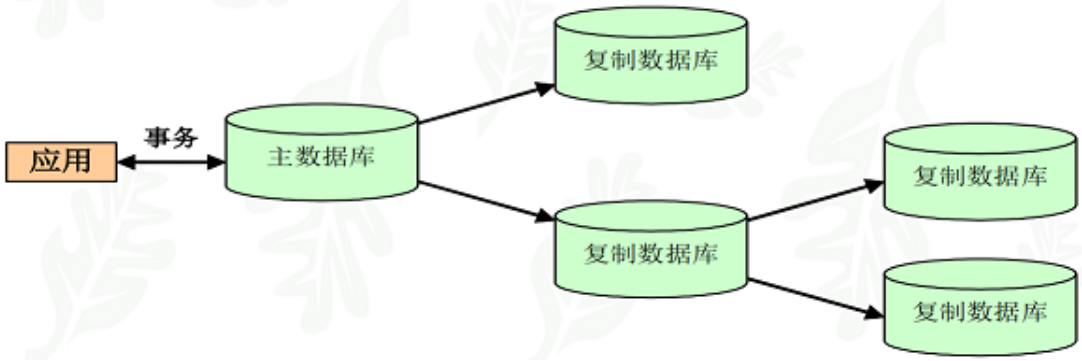
对等复制：是理想的复制方式。各场地DB地位平等，可互相复制数据。用户可在任一场地读取/更新公共数据，DBMS应将更新数据复制到所有副本。



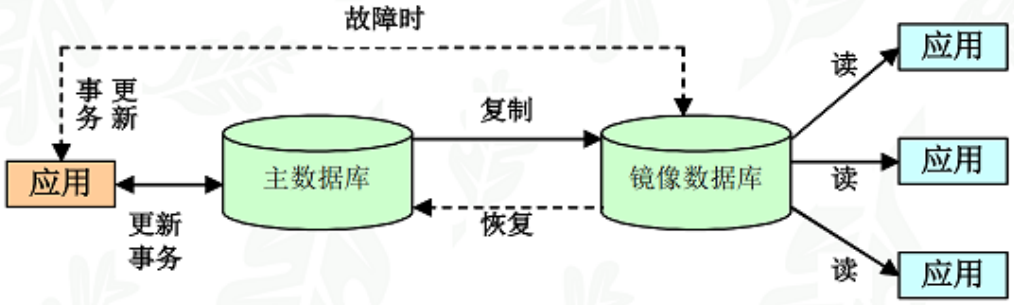
主从复制：数据只能从主DB复制到从DB。更新数据也只能在主场地进行，从场地供用户读数据。当主场地出现故障时，更新数据的应用可转到某一从场地。



级联复制：从主场地复制来的数据又从该场地复制到其他场地。级联复制可平衡当前各种数据需求对网络流量的压力。



数据库/日志镜像：DBMS为避免磁盘介质故障，提供日志与数据库镜像，由DBMS根据DBA要求，自动将整个DB或其中关键数据复制到另一个磁盘的数据库上。当主DB更新时，DBMS自动将更新后的数据复制到镜像数据库。出现介质故障时，可由镜像数据库提供应用服务，并利用它恢复主数据库。其实现通过数据复制进行。



- 刚才讲完了数据库中操作系统和DBMS本身提供的故障恢复体系，那么在导论的第二点中日志又是什么呢？
  - 日志的基本内容
    - 活动事务表（ATL，active transaction list）。记录正在执行、但尚未提交的事务的标识符（TID，transaction identifier）有。
    - 提交事务表（CTL，committed transaction list）。记录已提交事务的标识符
    - 前像：事务更新的数据物理块更新前的映像
    - 后像：事务更新的数据物理块更新后的映像
- 既然知道了日志是用来进行数据恢复的，那么故障发生，数据库恢复后又该如何处理日志呢？
  - 不保留已提交事务的前像：当再次故障时，对已提交的事务只用后像重做。
  - 有选择地保留后像：如果磁盘不出故障，后像可不用保留，因为磁盘故障概率小
  - 合并后像：对具有相同物理块号上数据的多次更新，只需保留最近的后像即可。
- 有了日志，我们也需要正确使用才行，在更新事务的过程中，我们又该如何做呢？
  - 首先需要知道，按执行时是否会更新数据库中的数据，事务可分为：更新类事务、只读事务
  - 提交更新事务应遵守的规则：先记后写规则（WAL）是提交规则的补充。后者是，AI在事务提交前写入日记；前者是，BI在AI前写入日志。
- 正确操作更新事务的过程中，我们可能会遇到突发故障，此时又该怎么办呢？
  - AI在事务提交前完全写入数据库

事务执行状态	DBMS应做的工作
开始状态	步骤1：TID→ATL /* 将事务TID记入日志的ATL */
事务内的操作执行状态	无
提交前	步骤2：BI→Log /* 将BI写入日志，先记后写规则的要求 */
	步骤3：AI→DB /* 将AI写入DB，提交前后像完全写入DB，本方案的要求 */
提交状态	步骤4：TID→CTL /* 提交更新事务 */
提交后	步骤5：从ATL删除TID

ATL	CTL	事务所处状态	恢复措施
有	无（未提交）	步骤1已完成，但步骤4尚未完成	① 若BI已➔Log,则undo；否则勿需undo； ② 从ALT删除TID。
有	有（已提交）	步骤4已完成	从ALT删除TID
无	有（已提交）	步骤5已完成	勿需处理

- 后像在事务提交后才写入数据库：

事务执行状态	DBMS应做的工作
开始状态	步骤1：TID→ATL /* 将事务的TID记入日志的ATL */
事务内的操作执行状态	无
提交前	步骤2：AI→Log /* 遵循提交规则的要求 */
提交状态	步骤3：TID→CTL /* 提交更新事务 */
提交后	步骤4：AI→DB /* 本方案的要求 */
	步骤5：从ATL删除TID

ATL	CTL	事务所处状态	恢复措施
有	无	步骤1已完成，但步骤3未完成	从ALT删除TID。
有	有	步骤3已完成，步骤5未完成	①redo； ②从ALT删除TID。
无	有	步骤5已完成	勿需处理

- 后像在事务提交前后写入数据库

事务执行状态	DBMS应做的工作
开始状态	步骤1：TID→ATL /* 将事务的TID记入日志的ATL */
事务内的操作执行状态	无
提交前	步骤2：AI，BI→Log /* 遵循提交规则和先记后写规则的要求 */
	步骤3：AI→DB /* 部分AI写入DB，本方案要求 */
提交状态	步骤4：TID→CTL /* 提交更新事务 */
提交后	步骤5：AI→DB /* 将剩余AI写入DB，本方案要求*/
	步骤6：从ATL删除TID

ATL	CTL	事务所处状态	恢复措施
有	无	步骤1已完成，但步骤4尚未完成	①若BI已➔Log,则undo；否则勿需undo； ②从ALT删除TID。
有	有	步骤4已完成，但步骤6未完成	①redo； ②从ALT删除TID。
无	有	步骤6已完成	勿需处理

- 综上所述，我们目前解决的都是一些静态数据，换句话来说，就是这些数据对象只有一个，但是有些数据的对象分别为起源对象、目标对象，比如消息的处理，那么此时，我们应该怎么做呢？

    - 答案很简单，我们设置一个消息管理器负责发送消息，要求在事务提交以前，不能对外发送事务内的任何消息。
- 那么消息具体如何处理呢?
    - 方法：事务执行时，把消息发送给消息管理器（MM），然后MM负责为每个事务建立一个消息队列，将消息暂时存储起来。
        - 若此时遇到故障，消息队列会被舍弃
    - 消息管理器对消息的发送处理：MM采用“发送——确认”方式。
- 事务的故障恢复固然有一定的方法，但每种方法不是万能的，我们必须清楚故障的类型，然后采取对应措施去应对，那么，故障类型有哪些呢?
    - 事务故障
        - 产生原因：事务无法执行而中止、用户主动撤销事务、因系统调度出错而终止。
        - 同步点，又称提交点：标志着事务的正常结束，数据库又处于一致性的状态
        - 恢复方法：1.从后向前扫描日志，找到故障事务；2.MM丢弃该事务的消息队列；3.撤销该事务已做的所有更新操作；4.从ATL中删除TID，释放该事务所占的资源
    - 介质故障
        - 产生原因：磁盘介质故障，意味着日志和物理数据都被破坏
        - 恢复方法：1.修复或更换磁盘系统，并重新启动系统；2.装入最近的数据库后备副本；3.装入有关的日志副本，重做已提交的事务。
    - 系统故障
        - 产生原因：掉电、软硬件故障
        - 检查点：定期设置
        - 恢复方法：1.重启操作系统和DBMS；2.从前完后扫描日志将TID记入重做队列中；3.反向扫描日志，对每个要重做的事务进行前像撤销；4.正向扫描日志，对每个事务用后像重做
- 对于数据库的保护，除了故障恢复以外，还有对并发的保护，那么如何展开呢?
- 首先，我们需要了解一些概念
    - 串行执行：DBMS按顺序一次执行一个事务——控制事务串行执行的调度，串行调度
    - 并发执行：DBMS同时执行多个事务——控制事务并发执行的调度，并发调度
    - 调度：被按要求重新组合交由DBMS执行的一串有序操作集
- 并发的目的是什么呢?
    - 提高系统资源利用率
    - 改善短事务的响应时间
- 并发可能引起的问题：
    - 丢失更新，覆盖未提交的数据：由于两个（或多个）事务对同一数据并发地写入引起，称为“写-写冲突”
    - 读脏数据，读未提交的数据：后一个事务读了前一个事务写了但未提交的数据，称为“写-读冲突”
    - 读值不可复现，不可重复读：“读-写冲突”，其中的“幻影问题”，基于此
- 如何判断并发是否可行呢?
    - 前驱图：有向图，若无回路，则代表可串行化（可行）
- 那么，并发具体是怎么实现的呢?
- 基于锁的并发控制协议
    - 互斥并发控制协议：互斥的方式下访问数据
    - 加锁协议：基于锁的并发控制协议，目前商用RDBMS广泛采用的并发控制方法
    - X锁协议，又称排他锁或独占锁：其他事务必须等待X锁解锁以后才可访问
        - 级联回退现象：一事务还未结束，就把它已获得的锁释放，其中又因某种原因，该事务需要回退，为避免其他事物读到脏数据，要求读了该事务更新接连回退的现象。
        - 加锁协议补丁：为避免级联回退，要求不管是写操作锁还是读操作锁，都应保持到事务结束才释放
    - （S、X）加锁协议：S锁，共享锁，又称读操作锁；读操作时要加S锁，写时加X锁
        - 并发度：允许访问同一数据的用户数。因此（S、X）锁提高了读数据的并发度。
        - 活锁：不断有事务申请S锁，导致该数据对象一直被S锁占据，则X锁的申请迟迟得不到获准的现象。

        - 先来先服务：为避免活锁，当多个事务请求对同一数据对象的加锁时，DBMS按请求加锁的先后顺序，一旦该事务身上的锁被释放，那么就优先批准申请队列上的第一个事务获得加锁。
    - （S、U、X）加锁协议：U锁，更新锁或修改修；数据对象加了U锁后，仍允许其他事务访问。目的是保证“读”时需更新的数据对象仍可被其他事务访问。
- 在刚刚的加锁协议中，我们一直都没有声明加锁的数据对象的大小，那么，多粒度加锁协议是什么呢？
    - 加锁的粒度：数据对象的大小
    - 表锁：用于锁表的锁
    - 行锁：用于锁行的锁
    - 页级锁
- 多粒度加锁协议又如何实现呢？
    - 锁冲突检测问题
        - 显示加锁：系统应事务的要求，直接对该数据对象的加锁，有时简称显式锁
        - 隐式加锁：该数据对象本身并没有被加锁，但由于祖先被加锁了，故这个数据对象被隐式加了锁，有时简称隐式锁
    - 简化锁冲突检测的方法：传统的检测方法是检测所有祖先和所有子孙是否被加锁，但是太过于麻烦，因此引入了3种意向锁。
    - 意向共享锁：一个事务要给一个数据对象加S锁，首相必须对其祖先加上IS锁。这样一来，若祖先上有IS锁，表示其子孙有S锁。
    - 意向排他锁：同理加上IX锁
    - 共享意向排他锁：SIX锁，等价于同时拥有S锁和IX锁。为什么要这样呢？比如要读整张表，并修改其中的个别行。就需要整体S，和子孙X
    - 多粒度加锁协议的相容矩阵：

加锁申请 \ 数据对象的锁状态	NL	IS	IX	S	SIX	U	X
IS	Y	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	N	N	N	N
S	Y	Y	N	Y	N	Y	N
SIX	Y	Y	N	N	N	N	N
U	Y	Y	N	Y	N	N	N
X	Y	N	N	N	N	N	N

多粒度锁相容矩阵

    - 多粒度加锁/解锁顺序
        - 加锁时，要对一个数据对象加锁，必须对这个数据对象的所有祖先，加上相应的意向锁，即自上而下
        - 解锁时，要先对该数据对象的所有子孙进行解锁，即自下而上
- 在讲解并发度的过程中，对于加锁问题，我们容易造成死锁现象，那么，我们应该如何预防、检测死锁呢？
    - 死锁是什么？
        - 如果一个事务申请锁而未获准，则需要等待其他事务释放锁，从而形成事务间的等待关系，此时，事务出现循环等待的现象。
    - 死锁预防的基本思路
        - 只允许事务间单向等待。
    - 时间戳是什么？
        - 事务的优先级：即每个事务开始执行时，给定一个时间戳，时间戳越前，事务优先级越高
    - 预防死锁的策略有哪些呢？

----------描述策略之前的假定：设TB已持有某数据对象的锁，现在TA申请同一数据对象的锁。----------

- 等待——死亡策略：若TA优先级>TB，则允许TA等待；否则令TA撤销（回退）
- 击伤——等待策略：若TA优先级<TB，则允许TA等待；否则令TB终止或撤销

第一种：

```
if ts(TA) < ts(TB)
        TA waits;              /*  wait  */
else
{
        rollback  TA;          /* die  */
        restart  TA with the same ts(TA);
}
```

第二种：

```
if ts(TA)>ts(TB)
      TA waits;              /*  wait  */
else
{
      rollback  TB;          /* wound  */
      restart  TB with the same ts(TB);
}
```

- 检测死锁的策略又有哪些呢？——>超时法、等待图
- 超时法：如果一个事务等待锁的时间太长，超过事先设定的时限，则主观认定其处于循环等待中而回退
  - 难点在于超时时间的设置。可能误判
- 等待图：有向图，若有回路，说明死锁发生。（和“并发的可串行化”中的前驱图类似）
  - 等待图的构建与维护：锁请求时，增加一条边；锁获准时，减少一条边。
  - 由于构建与维护都需要检查回路，因此开销大

- 当死锁出现时，我们又该如何处理呢？
  - 循环等待的事务中，选择一个事务作为牺牲者
  - 回退牺牲的事务，释放其所获得的锁和占用的资源
  - 将释放的锁让给等待的其他事务
- 那么选择牺牲者的标准是什么呢？
  - 最迟交付的事务
  - 获得锁最少的
  - 回退代价最小的