1.MetaData

文件数据都储存在"块"中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做inode，中文译名为"索引节点"。

1.1什么是元数据

任何文件系统中的数据分为数据和元数据。数据是指普通文件中的实际数据，而元数据指用来描述一个文件的特征的系统数据，诸如访问权限、文件拥有者以及文件数据块的分布信息(inode...)等等。在集群文件系统中，分布信息包括文件在磁盘上的位置以及磁盘在集群中的位置。用户需要操作一个文件必须首先得到它的元数据，才能定位到文件的位置并且得到文件的内容或相关属性。

1.2 元数据管理方式

元数据管理有两种方式。集中式管理和分布式管理。集中式管理是指在系统中有一个节点专门司职元数据管理，所有元数据都存储在该节点的存储设备上。所有客户端对文件的请求前，都要先对该元数据管理器请求元数据。分布式管理是指将元数据存放在系统的任意节点并且能动态的迁移。对元数据管理的职责也分布到各个不同的节点上。大多数集群文件系统都采用集中式的元数据管理。因为集中式管理实现简单，一致性维护容易，在一定的操作频繁度内可以提供较满意的性能。缺点是单一失效点问题，若该服务器失效，整个系统将无法正常工作。而且，当对元数据的操作过于频繁时，集中的元数据管理成为整个系统的性能瓶颈。

分布式元数据管理的好处是解决了集中式管理的单一失效点问题， 而且性能不会随着操作频繁而出现瓶颈。其缺点是，实现复杂，一致性维护复杂，对性能有一定影响。

2. NewSQL NoSQL

NewSQL 是对各种新的可扩展/高性能数据库的简称，这类数据库不仅具有[NoSQL](https://baike.baidu.com/item/NoSQL)对海量数据的存储管理能力，还保持了传统[数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93)支持[**ACID**](https://baike.baidu.com/item/ACID)**（原子性、一致性、隔离性和持久性**）和[SQL](https://baike.baidu.com/item/SQL)等特性.

**NewSQL**是指这样**一类新式的关系型数据库管理系统**，针对OLTP（读-写）工作负载，追求提供和NoSQL系统相同的扩展性能，且仍然保持ACID和SQL等特性（scalable and ACID and (relational and/or sql -access)）。H-Store的重要意义在于，它是第一个实现的新型并行数据库管理系统，称为NewSQL。这种数据库系统提供了NoSQL系统所具有的高吞吐量和高可用性的，而且并没有放弃传统DBMS的事务处理的功能。**这样的系统能够在多台计算机上进行**[**并行计算**](https://baike.baidu.com/item/%E5%B9%B6%E8%A1%8C%E8%AE%A1%E7%AE%97)**，以提高吞吐量，而不是运行在一个超强大，超昂贵的单节点计算机上。H-Store的商用版本是**[**VoltDB**](https://baike.baidu.com/item/VoltDB)**。**这类数据库采用不同于传统数据库的新型架构，工作在一个分布式集群的节点上，其中每个节点拥有一个数据子集。 SQL查询被分成查询片段发送给自己所在的数据的节点上执行。这些数据库可以通过添加额外的节点来线性扩展。现有的这类数据库有： Google Spanner, VoltDB。

NoSQL(NoSQL = Not Only SQL )，意即“不仅仅是[SQL](https://baike.baidu.com/item/SQL)”, NoSQL，泛指非关系型的数据库。

随着互联网[web2.0](https://baike.baidu.com/item/web2.0)网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的[SNS](https://baike.baidu.com/item/SNS/10242)类型的web2.0纯[动态网](https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E7%BD%91)站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题。

NoSQL分为**键值(**[*Key-Value*](https://baike.baidu.com/item/Key-Value)**)存储**[**数据库**](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93)**、列存储数据库（**Cassandra, HBase**）、文档型数据库（**该类型的数据模型是版本化的文档，半结构化的文档以特定的格式存储，比如JSON。MongoDb）**、图形(Graph)数据库**

相对于事务严格的ACID特性，NoSQL数据库保证的是BASE特性。BASE是最终一致性和软事务。

3.namespace

一、为什么使用命名空间

一个大型的工程往往是由若干个人独立完成的，不同的人分别完成不同的部分，最后再组合成一个完整的程序。由于各个头文件是由不同的人设计的，有可能在不同的头文件中用了相同的名字来命名所定义的类或函数，这样在程序中就会出现名字冲突。不仅如此，有可能我们自己定义的名字会与C++库中的名字发生冲突。

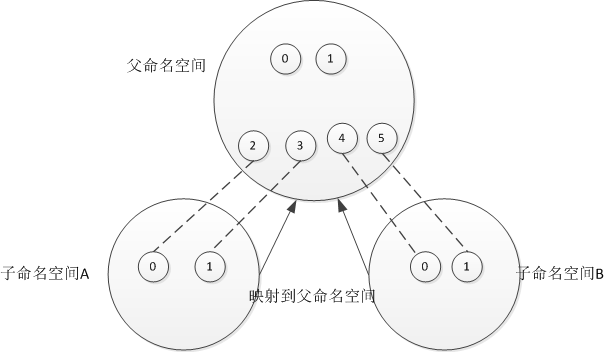
名字冲突就是在同一个作用域中有两个或多个同名的实体，为了解决命名冲突 ，C++中引入了命名空间，所谓命名空间就是一个可以由用户自己定义的作用域，在不同的作用域中可以定义相同名字的变量，互不干扰，系统能够区分它们。

二、什么是命名空间

命名空间是程序设计者命名的内存区域，程序设计者根据需指定一些有名字的空间域，把一些全局实体分别存放到各个命名空间中，从而与其他全局实体分隔开。

通俗的说，每个名字空间都是一个名字空间域，存放在名字空间域中的全局实体只在本空间域内有效。名字空间对全局实体加以域的限制，从而合理的解决命名冲突。

namespace，命名空间，从名字上看，应该是类似于包含许多名字的空间，打个比方，三年一班的小明和三年二班的小明，虽说他们名字是一样的，但是所在班级不一样，那么，在全年级排行榜上面，即使出现两个名字一样的小明，也会通过各自的学号来区分。对于学校来说，每个班级就相当于是一个命名空间，这个空间的名称是班级号。班级号用于描述逻辑上的学生分组信息，至于什么学生分配到1班，什么学生分配到2班，那就由学校层面来统一调度。大致应该就是这么个意思，恩。



命名空间建立系统的不同视图， 对于每一个命名空间，从用户看起来，应该像一台单独的Linux计算机一样，有自己的init进程(PID为0)，其他进程的PID依次递增，A和B空间都有PID为0的init进程，子容器的进程映射到父容器的进程上，父容器可以知道每一个子容器的运行状态，而子容器与子容器之间是隔离的。

4. stateless servers 无状态服务器

对服务器程序来说，究竟是有状态服务，还是无状态服务，其判断依旧是指两个来自相同发起者的请求在服务器端是否具备上下文关系。如果是**状态化请求**，那么服务器端一般都要保存请求的相关信息，每个请求可以默认地使用以前的请求信息。而对于**无状态请求**，服务器端所能够处理的过程必须全部来自于请求所携带的信息，以及其他服务器端自身所保存的、并且可以被所有请求所使用的公共信息。  
        无状态的服务器程序，最著名的就是WEB服务器。每次HTTP请求和以前都没有啥关系，只是获取目标URI。得到目标内容之后，这次连接就被杀死，没有任何痕迹。在后来的发展进程中，逐渐在无状态化的过程中，加入状态化的信息，比如COOKIE。服务端在响应客户端的请求的时候，会向客户端推送一个COOKIE，这个COOKIE记录服务端上面的一些信息。客户端在后续的请求中，可以携带这个COOKIE，服务端可以根据这个COOKIE判断这个请求的上下文关系。COOKIE的存在，是无状态化向状态化的一个过渡手段，他通过外部扩展手段，COOKIE来维护上下文关系。  
        状态化的服务器有更广阔的应用范围，比如MSN、网络游戏等服务器。他在服务端维护每个连接的状态信息，服务端在接收到每个连接的发送的请求时，可以从本地存储的信息来重现上下文关系。这样，客户端可以很容易使用缺省的信息，服务端也可以很容易地进行状态管理。比如说，当一个用户登录后，服务端可以根据用户名获取他的生日等先前的注册信息；而且在后续的处理中，服务端也很容易找到这个用户的历史信息。  
        状态化服务器在功能实现方面具有更加强大的优势，但由于他需要维护大量的信息和状态，在性能方面要稍逊于无状态服务器。无状态服务器在处理简单服务方面有优势，但复杂功能方面有很多弊端，比如，用无状态服务器来实现即时通讯服务器，将会是场恶梦。

5.zookeeper

ZooKeeper是一个[分布式](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F)的，开放源码的[分布式应用程序](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F)协调服务，是[Google](https://baike.baidu.com/item/Google)的Chubby一个[开源](https://baike.baidu.com/item/%E5%BC%80%E6%BA%90)的实现，是Hadoop和Hbase的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等

使用Zookeeper可以保证总服务器自动感知有多少提供搜索引擎的服务器并向这些服务器发出搜索请求，当总服务器宕机时自动启用备用的总服务器

6.checkpoint

checkpoint是一个内部事件，这个事件激活以后会触发数据库写进程([DBWR](https://baike.baidu.com/item/DBWR/1274116))将数据缓冲(**DATA**BUFFER CACHE)中的[脏数据](https://baike.baidu.com/item/%E8%84%8F%E6%95%B0%E6%8D%AE/631511)块写出到数据文件中。

在[数据库系统](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B3%BB%E7%BB%9F)中，写日志和写数据文件是数据库中IO消耗最大的两种操作，在这两种操作中写数据文件属于分散写，写日志文件是顺序写，因此为了保证数据库的性能，通常数据库都是保证在提交([commit](https://baike.baidu.com/item/commit/9214278))完成之前要先保证日志都被写入到日志文件中，而脏数据块则保存在数据缓存(buffer cache)中再不定期的分批写入到数据文件中。也就是说日志写入和提交操作是同步的，而数据写入和提交操作是不同步的。这样就存在一个问题，当一个数据库崩溃的时候并不能保证缓存里面的[脏数据](https://baike.baidu.com/item/%E8%84%8F%E6%95%B0%E6%8D%AE/631511)全部写入到数据文件中，这样在实例启动的时候就要使用日志文件进行恢复操作，将[数据库恢复](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E6%81%A2%E5%A4%8D/3949044)到崩溃之前的状态，保证数据的一致性。检查点是这个过程中的重要机制，通过它来确定，恢复时哪些重做日志应该被扫描并应用于恢复。

checkpoint主要2个作用：

保证数据库的一致性，这是指将脏数据写入到硬盘，保证内存和硬盘上的数据是一样的;

缩短实例恢复的时间，实例恢复要把实例异常关闭前没有写出到硬盘的[脏数据](https://baike.baidu.com/item/%E8%84%8F%E6%95%B0%E6%8D%AE/631511)通过日志进行恢复。如果脏块过多，实例恢复的时间也会很长，检查点的发生可以减少脏块的数量，从而提高实例恢复的时间。

通俗的说checkpoint就像word的自动保存一样

HDFS的checkpoint

Checkpoint（检查点）：因为数据库系统或者像HDFS这样的分布式文件系统，对文件数据的修改不是直接写回到磁盘的，很多操作是先缓存到内存的Buffer中，当遇到一个检查点Checkpoint时，系统会强制将内存中的数据写回磁盘，当然此时才会记录日志，从而产生持久的修改状态。

对于这两者的区别，个人认为主要是Snapshot是对数据的备份，而Checkpoint只是一个将数据修改持久化的机制。

**Checkpoint执行过程**

Chekpoint主要干的事情是，将Namenode中的edits和fsimage文件拷贝到Second Namenode上，然后将edits中的操作与fsimage文件merge以后形成一个新的fsimage，这样不仅完成了对现有Namenode数据的备份，而且还产生了持久化操作的fsimage。最后一步，Second Namenode需要把merge后的fsimage文件upload到Namenode上面，完成Namenode中fsimage的更新。

以上提到的文件都可以在hadoop系统的data目录下找到。

7. Quorom 机制，是一种分布式系统中常用的，用来保证数据冗余和最终一致性的投票算法，其主要数学思想来源于[鸽巢原理](http://zh.wikipedia.org/wiki/%E9%B8%BD%E5%B7%A2%E5%8E%9F%E7%90%86)。

8. redo undo

重做日志：每当有操作执行前，将数据真正更改时，先前相关操作写入重做日志。这样当断电，或者一些意外，导致后续任务无法完成时，系统恢复后，可以继续完成这些更改  
  
撤消日志：当一些更改在执行一半时，发生意外，而无法完成，则可以根据撤消日志恢复到更改之前的壮态

redo--> undo-->datafile

insert一条记录时, 表跟undo的信息都会放进 redo 中, 在commit 或之前, redo 的信息会放进硬盘上. 故障时, redo 便可恢复那些已经commit 了的数据.

redo->每次操作都先记录到redo日志中，当出现实例故障（像断电），导致数据未能更新到数据文件，则数据库重启时须redo，重新把数据更新到数据文件

undo->记录更改前的一份copy，但你系统rollback时，把这份copy重新覆盖到原来的数据

redo->记录所有操作，用于恢复（redo records all the database transaction used for recovery）

undo->记录所有的前印象，用于回滚（undo is used to store uncommited data infor used for rollback）

redo->已递交的事务,实例恢复时要写到数据文件去的

undo->未递交的事务.

redo的原因是：每次commit时，将数据的修改立即写到online redo中，但是并不一定同时将该数据的修改写到数据文件中。因为该数据已经提交，但是只存在联机日志文件中，所以在恢复时需要将数据从联机日志文件中找 出来，重新应用一下，使已经更改数据在数据文件中也改过来！

undo的原因是：在oracle正常运行时，为了提高效率，加入用户还没有commit,但是空闲内存不多时，会由DBWR进程将脏块写入到数据 文件中，以便腾出宝贵的内存供其它进程使用。这就是需要UNDO的原因。因为还没有发出commit语句，但是oracle的dbwr进程已经将没有提交 的数据写到数据文件中去了。

undo 也是也是datafile， 可能dirty buffer 没有写回到磁盘里面去。

只有先redo apply 成功了，才能保证undo datafile 里面的东西都是正确的，然后才能rollback

做undo的目的是使系统恢复到系统崩溃前(关机前)的状态,再进行redo是保证系统的一致性.

不做undo,系统就不会知道之前的状态,redo就无从谈起

所以instance crash recovery 的时候总是先rollforward， 再rollback

9. **可串行化（Serializability）**

**可串行化**是一个调度，即多个事务之间的执行方式；而多个事务之间的执行有个先后顺序，如果事务之间没有共同的操作对象（读或写操作），则事务之间的执行顺序前后置换是没有关系的；但是如果事物间存在共同的操作对象，则事务间先后执行的顺序则需要区分；对于存在共同操作对象的多个并发执行的事务，如果其执行结果“等价”于某个“**串行化调度**”，则这个调度才是“**可串行化的调度**”。满足“**可串行化的调度**”则具有了**可串行化（Serializability）**属性。所以，**可串行化（Serializability）**属性保证的是**多个事务并发时的执行顺序要对数据的一致性没有影响**。

10行锁

行级锁定的优点：

· 当在许多线程中访问不同的行时只存在少量锁定冲突。

· 回滚时只有少量的更改。

· 可以长时间锁定单一的行。

行级锁定的缺点：

· 比页级或表级锁定占用更多的内存。

· 当在表的大部分中使用时，比页级或表级锁定速度慢，因为你必须获取更多的锁。

· 如果你在大部分数据上经常进行GROUP BY操作或者必须经常扫描整个表，比其它锁定明显慢很多。

· 用高级别锁定，通过支持不同的类型锁定，你也可以很容易地调节应用程序，因为其锁成本小于行级锁定。

11.Under replicated blocks:指的是副本数少于指定副本数的block数量

Blocks with corrupt replicas: 指的是存在损坏副本的block的数据

12.pseudo-randomly 伪随机

假如，有10个事件，每个发生的概率都是10%。进行随机触发10次

对于正常随意而言，会出现某个事情出现多次的情况。但是伪随机，在事情触发前设定好，就是这个十个事件各发生一次，只不过顺序不同而已。

现实例子

现在MP3的随机列表就是用的伪随机，把要播放的歌曲打乱顺序，生成一个随机列表而已，每个歌曲都播放一次。真实随机的话，会有出现某首歌多放次的情况，歌曲基数越多，重放的概率越大。

对于MP3的这个伪随机而言，一方面是消除了重放的概率（一首歌听两次也就算了，连续听两次是不是觉得怪怪的），还有一方面，可以往回倒一首听。而真随机的话，过去就过去了，刚才那首歌就很难再找了。

13 循环死锁

循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

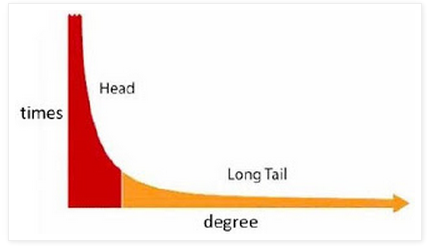
（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

14 lock upgrades

Consider the following situation:

* Thread 1 acquires a read lock
* Thread 2 acquires a read lock
* Thread 1 tries to acquire a write lock, and is blocked on thread 2’s read lock
* Thread 2 tries to acquire a write lock, and is blocked on thread 1’s read lock.

15. 重尾分布



重尾分布（Heavy-tailed distribution）是一种[概率分布](https://baike.baidu.com/item/%E6%A6%82%E7%8E%87%E5%88%86%E5%B8%83/828907)模型，它的尾部比[指数分布](https://baike.baidu.com/item/%E6%8C%87%E6%95%B0%E5%88%86%E5%B8%83/776702)还要厚。在许多情况下，右边尾部的部分比较受到重视，

16. [**乐观并发控制与悲观并发控制的区别**](http://www.cnblogs.com/Fandyx/archive/2012/07/14/2591153.html)

**悲观并发控制**

**一个锁定系统，可以阻止用户以影响其他用户的方式修改数据。如果用户执行的操作导致应用了某个锁，只有这个锁的所有者释放该锁，其他用户才能执行与该锁冲突的操作。这种方法之所以称为悲观并发控制，是因为它主要用于数据争用激烈的环境中，以及发生并发冲突时用锁保护数据的成本低于回滚事务的成本的环境中。**

**悲观锁的问题是减少了并发的程序。**

**在乐观锁和悲观锁之间进行选择的标准是：冲突的频率与严重性。如果冲突很少，或者冲突的后果不会很严重，那么通常情况下应该选择乐观锁，因为它能得到更好的并发性，而且更容易实现。但是，如果冲突的结果对于用户来说痛苦的，那么就需要使用悲观策略。**

**乐观并发控制**

**在乐观并发控制中，用户读取数据时不锁定数据。当一个用户更新数据时，系统将进行检查，查看该用户读取数据后其他用户是否又更改了该数据。如果其他用户更新了数据，将产生一个错误。一般情况下，收到错误信息的用户将回滚事务并重新开始。这种方法之所以称为乐观并发控制，是由于它主要在以下环境中使用：数据争用不大且偶尔回滚事务的成本低于读取数据时锁定数据的成本。**

**具体的区别与实例说明如下：**

**悲观并发控制：假设A和B需要在SCC（Source Code Control）上修改同一个文件，那么在A锁定这个文件并修改的过程中，B无法修改这个文件，他只能等待A解锁文件后，他才能修改。由此可见，悲观并发控制是强调控制在前，确保整个过程不会出现文件版本的冲突。这样做会使得系统效率损耗在加锁机制上，尤其是加锁机制需要用到低速的外部存储（比如FileLocking）时，然而这样做就降低了事务的并发性，尤其是事务之间本来就不存在冲突的情况下。例如在A修改数据的时候，B只能等待。**

**由此可见，悲观并发控制通过使用显式的加锁机制或者时间戳，对每一个事务进行增量同步校验。如果加锁机制的成本较高的话，悲观并发控制就会出现一些弊端。首先就是效率问题，尤其是使用低效率的外部存储系统实现加锁机制时，这样的问题会更加突出。其次，在不会出现冲突的事务处理（例如只读型事务）中，使用加锁机制就显得没有必要了，这样做只能增加系统负载。再次，这种方式降低了系统的并发性。**

**乐观并发控制：同样假设A和B需要在SCC上修改同一个文件，他们都将这个文件获取到自己的机器上，A修改完以后，就把文件上传到SCC上了，此时B也修改完了，当他也打算将文件上传时，系统会告知B，已经有人上传了，并出现一个错误。剩下的问题只能由B手动解决，例如B可以在SCC上将文件中更改的内容再次复制一遍。乐观并发控制使得系统效率损耗在事务的后期处理中，比如B必须手动的去修改他已经修改过的东西，然而这种控制方式在极少出现冲突的多事务处理中显得十分高效。**

**17.序列化**

* 序列化： 将数据结构或对象转换成二进制串的过程

序列化 (Serialization)将对象的状态信息转换为可以存储或传输的形式的过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。

18. phantom and fuzzy reads

我来列举一下事务执行过程中，会出现的几种情况，：

Dirty reads

脏读，其他事务读取到了另外的事务更改的却没有提交的数据。

Nonrepeatable (fuzzy) reads 不可重复读

因为其他事务更改了数据，一个事务中两次查询这一行数据的时候出现的结果不同。

Phantom reads 幻读（虚读）

因为其他事务更改了数据，一个事务中两次查询的结果集不同。

a) FUZZY READ: A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data.

b) PHANTOM READ: A transaction re-executes a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

**18 read-commited**

1)脏读

　　脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

　　当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户A向用户B转账100元，对应SQL命令如下

update account set money=money+100 where name=’B’; (此时A通知B)

update account set money=money - 100 where name=’A’;

　　当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

2)不可重复读

　　不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

　　例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

　　不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

　　在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据A和B依次查询就可能不同，A和B就可能打起来了……

3)虚读(幻读)

　　幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

　　幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

　　现在来看看MySQL数据库为我们提供的四种隔离级别：

　　① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

　　② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

　　③ Read committed (读已提交)：可避免脏读的发生。

　　④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。像ino类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。在MySQL数据库中默认的隔离级别为Repeatable read (可重复读)。

|  |  |  |  |
| --- | --- | --- | --- |
| **Isolation Level** | **Dirty Read** | **Nonrepeatable Read** | **Phantom Read** |
| **Read uncommitted** | Possible | Possible | Possible |
| **Read committed** | Not possible | Possible | Possible |
| **Repeatable read** | Not possible | Not possible | Possible |
| **Serializable** | Not possible | Not possible | Not possible |