zhuanlan.zhihu.com /p/440149077

事务的弱隔离级别之Serializability

前面的两篇文章介绍了两种弱隔离级别：

- Read Committed
- Snapshot Isolation

程序员阿sir：事务的弱隔离级别之Read Committed9 赞同 · 0 评论文章 

程序员阿sir：事务的弱隔离级别之快照隔离4 赞同 · 0 评论文章 

今天介绍一种新的隔离级别，也是**最强的隔离级别**：

- **可串行化 (Serializability)**

3. 可串行化 (Serializability)

3.1. 概念

可串行化隔离能保证所有事务无论在任何并发情况下同时运行，其运行结果保证与串行运行的结果**一致**。

可串行化隔离是一种**最强**的隔离级别。

其他的弱隔离级别都是尽量避免一些并发问题，但是都有一些问题无法避免。

但是 Serializability 可以**避免所有可能的并发问题**。

有些人可能会问：既然它这么厉害，大家都用这种隔离级别就行了，为什么还有那些弱隔离级别呢？

这里主要是因为**性能**问题。

可串行化隔离需要做严格的并发控制，所以性能肯定会大大降低。

而且不是所有应用都需要这么严格的隔离级别，但是需要更好的性能，因此会选择弱一些的隔离级别。

大多数数据库都实现了可串行化隔离，一般有三种方式：

1. 真串行执行 (Actual Serial Execution)
2. 二段锁 (Two-phase Locking, 2PL)
3. 可串行的快照隔离 (Serializable Snapshot Isolation, SSI)

下面我们来分别介绍一下这三种实现方式。

3.2. 真串行执行 (Actual Serial Execution)

最容易理解的方式就是把所有并发都完全移除，**每一个时间**只有**一个线程**在执行**唯一**的一个事务。

这样就肯定不会有并发问题了。

早期的数据库会把整个业务流程当作一个数据库事务，包括用户行为。

> 比如订机票的过程，就是一个多阶段的过程：
> 先搜索航班、点击预订、付钱、选择座位。
> 这里面的用户逻辑太多了，这就导致一个事务一直在这里卡着，其他事务也执行不了。

另外，在应用与数据库交互的时候，一般是通过**网络**发送数据库请求。

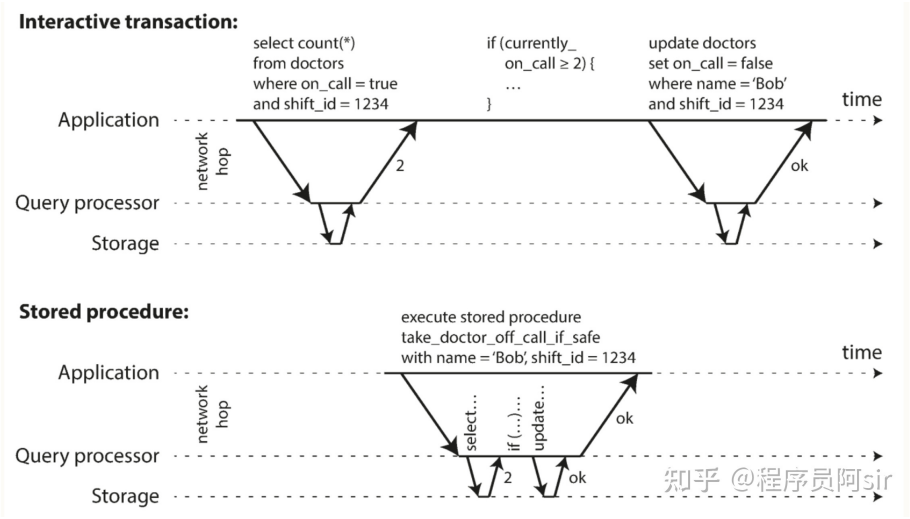
如果每个数据库操作都需要网络请求，那整个事务就把大量的**网络延迟**也包含了进来，就进一步导致了**性能**的降低。

因此，单线程的事务处理系统**不允许交互式的多阶段事务**。

应用必须把整个事务的代码**提前提交**给数据库，每次应用想要执行这段代码的时候就传入对应的参数就可以执行了。

这段代码就叫做**存储过程 (Stored Procedure)**。当应用提供了存储过程所需要的数据之后，存储过程会在内存中被执行，速度很快，没有网络延迟或者IO操作。

交互式事务和使用存储过程的事务的区别如下图示例所示。



交互式事务和使用存储过程的事务的区别例子

可以看出：
交互式事务逻辑都是在**应用端**执行的，然后每次会把数据库操作通过网络请求发送给数据库。
同时在多次数据库操作之间，应用还会处理一些业务逻辑。
而**存储过程**就是把逻辑部分**全部提前**放到了数据库端，并且暴露一些参数。
应用端每次想要执行事务的时候，会把这些**参数**传递给数据库，并让数据库执行这个**存储过程**。
这样就节省了很多网络时间，并且避免了等待用户操作的逻辑。

但是存储过程也有很多**缺点**：

1. 每个数据库要求写存储过程的语言是不同的。
比如：Orale用的是 PL/SQL， SQL Server用 T-SQL，PostgreSRL用 PL/pgSQL等等。这些语言各不相同，无法直接迁移。
2. 运行在数据库端的代码很难debug和测试，也很难管理。
因为都存在数据库端。
3. 存储过程对数据库有更高的性能要求。
因为可能很多应用会使用**同一个**数据库，并且运行**不同**的存储过程。
如果有些应用写的存储过程性能不好，会影响其他应用的存储过程。

其实现在的存储过程已经做了**很多改进**，比如:

很多现代存储过程的实现使用一些**通用的语言**，例如 VoltDB 支持使用Java 或 Groovy 来写存储过程；Datomic 使用 Java 或 Clojure；Redis 使用 Lua。这样就比较容易学习和理解了。

3.3. 二段锁 (Two-phase Locking, 2PL)

从数据库诞生以来，唯一被广泛应用的可序列化隔离算法就是**二段锁 (Two-phase Locking, 2PL)**

2PL与之前弱隔离级别的加锁实现有点类似。只不过用了更强的锁。

二段锁允许多个事务同时读一个数据库对象，只要没人写。

但是，只要有事务想要对一个对象执行写操作，就需要给这个对象加上排他锁 (Exclusive Lock):

1. 如果事务 A **正在读**一个对象，事务 B 想要**写**这个对象，那 B 必须等到 A **commit** 或者 **回滚**之后才能执行。——**这保证了 B 不会在 A 不知道的情况下修改了对象。**
2. 如果事务 A **正在写**一个对象，事务 B 想要**读**这个对象，那 B 必须等到 A **commit** 或者 **回滚**之后才能执行——**这保证了 B 不会读到旧版本的数据。**

可以看出：

在二段锁中，写者不仅会阻塞 (block)其他写者，也会阻塞其他读者。反过来，读者虽然不会阻塞其他读者，但是会阻塞其他写者。

> 值得注意的是：快照隔离中，**读操作**不能阻塞**写操作**，**写操作**也不能阻塞**读操作**。 这与二段锁是完全不同的。

二段锁的意思就是分为**两个阶段**：

1. 第一阶段：**当事务开始执行时获取锁**。
2. 第二阶段：**当事务结束时释放锁**。

如何实现二段锁？

在二段锁的实现中，锁有两种类型：

共享锁 (Shared Mode)

排他锁 (Exclusive Mode)。规则如下：

- 当一个事务想要**读**一个对象时，它必须请求这个对象的**共享锁**。其他事务也可以**同时持有**这个对象共享锁。但是如果其他事务已经持有了这个对象的**排他锁**，那当前事务必须等待直到排他锁被**释放**。
- 当一个事务想要**写**一个对象时，它必须请求这个对象的**排他锁**。没有其他事务可以同时持有这个对象的**任何锁（包括共享锁和排他锁）**。因此如果这个对象上有锁，那当前事务必须等待直到现存的锁被释放。
- 如果一个事务想要**先读后写**一个对象，它必须要将它的**共享锁**升级为**排他锁**。升级的规则和直接获取排他锁的规则是一样的。
- 当一个事务获得了一个锁之后，它必须持有这个锁一直到事务结束为止（Commit 或者 Abort）。

因为可能有很多锁被**同时占用**，所以可能出现**死锁**的情况。

这时数据库会自动检测死锁，并**主动回滚**其中的一个事务来解除死锁。

应用层则需要**重试**这个被回滚的事务。

二段锁的最大**缺点**就是**性能**问题。

使用二段锁的数据库事务的**吞吐量**和**响应时间**要明显比其他弱隔离级别的要差。

3.4. 可串行的快照隔离 (Serializable Snapshot Isolation, SSI)

SSI提供了**可串行化隔离**，性能只比快照隔离的性能差了一点，但是比二段锁和真串行执行的性能要好很多。

这个算法是2008年Michael Cahill在他的毕业论文中提出的。

SSI和前面的方法的区别在于，并发控制的思路不一样：

前两种是**悲观并发控制 (Pessimistic Concurrency Control)** ，SSI是**乐观并发控制 (Optimistic Concurrency Control)**。

详细地说：

- **【二段锁】是一种悲观并发控制机制。**

它的原则是：如果事情可能导致出错，那就加锁避免这种情况出现，进而保证事务不出现并发问题。
所以它采用加共享锁或排他锁的方式来避免并发问题。

- **【真串行执行】是一种极度悲观的并发控制机制。**

它的原则是：并发执行多个事务可能出现并发问题，那事务执行的时候把整个数据库都锁上，而且是排他锁，这样连并发都不要了肯定就不会出现并发问题了。

- **【SSI】是一种乐观的并发控制机制。**

它的原则是事情可能导致出错，但是不管它，所有事务都继续执行，SSI希望最后能得到正确的结果。所以让大家都直接执行，直到事务结束之后，SSI去检查是否结果有并发问题。如果有，那就回滚这个事务。

所以，**SSI** (可串行的快照隔离) 比起之前的**快照隔离**，**区别**在于：

SSI多了检测串行化冲突和决定哪些事务需要回滚的机制。

因篇幅原因，这些机制在这里就不一一介绍了。

总结

到这里我们用了三篇文章分别介绍了三种隔离级别：

内容参考自下面提到的这本书，如果大家对事务相关的内容感兴趣可以关注一下，也可以去查阅更多事务相关的书籍或博客，来了解更深层次的内容。

参考文献

[1] Kleppmann, Martin. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. " O'Reilly Media, Inc.", 2017.