**调优基础**

1. **区别不同的运算符**

在所有T-SQL语句在执行的时候，都会将语句分解为一些基本的结构单元，这些结构单元统称为：运算符。每一个运算符都实现一个单独的基本操作，比如：表扫描、索引查找、索引扫描、过滤等。每个运算符可以循环迭代，也可以延续子运算符，这样就可以组成查询树，即：查询计划。

每个T-SQL语句都会通过多种运算符进行组合形成不同的查询计划，并且这些查询计划对于结果的筛选都是有效的，但在执行的时候，SQL Server的查询优化器会自动为我们找到一个最优的。

每一个运算符都会有源数据的传入和结果数据的输出，源数据的输入可以来源于其它的运算符或者直接从数据源表中读取，经过本身的运算进行结果的输出。所以每一个运算符是独立的。互不关心的

每一个运算符会有三个属性影响其执行的效率

**内存消耗**

a、如果服务器上正在执行其它的类似的内存消耗巨大的查询，导致系统内存剩余不足的时候，当前的查询就得延迟进行，直接影响性能。

b、当并发量过大的的情况下，多个查询竞争有限的内存资源，服务器会适当的控制并发和减少吞吐量来维护机器性能，这时候同样也会影响性能

c、如果当前申请的到可用内存很少的情况下，SQL Server会在执行过程中和磁盘进行交换数据，通常是使用Tempdb临时库进行操作，而这个过程会很慢。更有甚者，会耗尽Tempdb上的磁盘空间以失败结束

通常比较消耗内存的运算符主要有分类、哈希连接以及哈希聚合等连接操作

**阻断运算和非阻断运算**

所谓阻断和非阻断的区别就是：运算符是否在输入数据的时候能够直接输出结果数据。

a、当一个运算符在消耗输入行的同时生成输出行，这种运算符就是非阻断式的。

比如我们经常使用的 Select Top ...操作，此操作就是输入行的同时进行输出行操作，所以此操作就是非阻断式的。

b、当一个运算符所产生的输出结果需要等待所有的数据输入的时候，这个操作运算就是阻断运算的。

比如上面我们举的例子Count(\*)操作，此操作就需要等待所有的数据行输入才能计算出，所以为阻断式运算，另外还有分组计算。

提示：并不是所有的阻断式操作就需要消耗内存，比如Count(\*)就为阻断式，但它不消耗内存，但大部分阻断式操作都会消耗内存。

在大部分的OLTP系统中，我们要尽量的使用非阻断式操作来代替阻断式操作，这样才能更好的提高相应时间，比如有时候我们用EXISTS子查询来判断，比用SELECT count(\*)>0的速度要理想的多。

**二、查看查询计划**

a、只开启执行计划，不包括详细的评估值

SET SHOWPLAN\_TEXT ON

b、开启所有的执行计划明细，包括各个属性的评估值

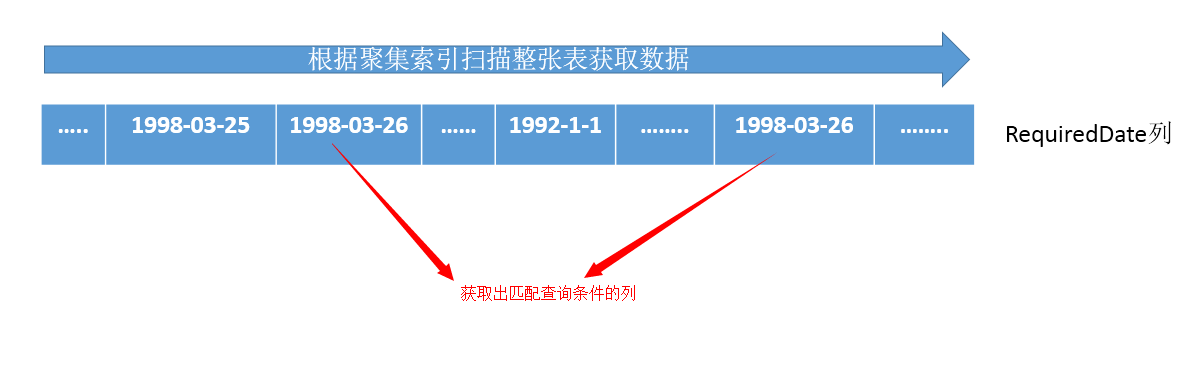
SET SHOWPLAN\_ALL ON

**三、分析查询计划**

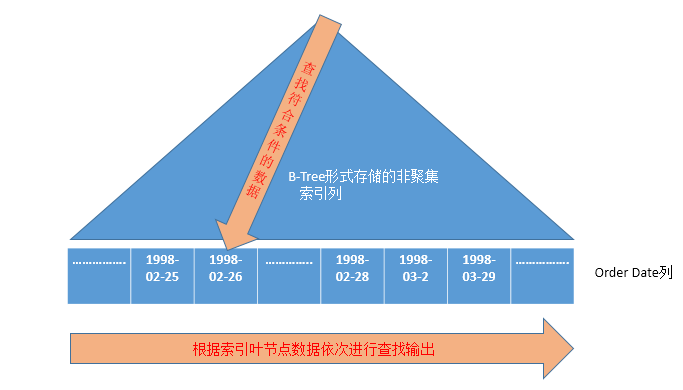
**1、扫描以及查找**

对于扫描（scan）和查找（seek）这两种方式是数据库里面从基础数据表里获取的数据的基本方式。

a、当一张表为堆表（没有任何索引）的时候或者获取的数据列不存在任何索引来供查找，此种数据的获取只能通过全表扫描过滤获取，如果存在索引项会通过索引项的扫描来获取数据，提高获取数据的速度



b、如果当前搜寻的数据行存在索引项，那么会采取索引查找（seek）进行数据检索。

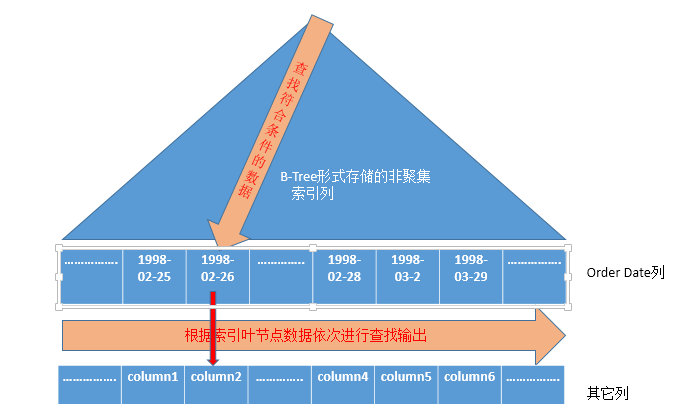


c、当所选的索引列不包含输出列的时候，也就是说要筛选出的列项不为索引所覆盖，对于这种情况又引出了另外一种查找方式.

**书签查找（Bookmark Lookup）**

其实该方式是扫描和查找之间的一个折中方式，我们知道，如果通过聚集索引扫描，则会获取所有的列，但是这涉及表中的每一行数据，影响性能，相反如果只是通过聚集索引方式进行查找，则有一些列不能获取得到，如果这些列正是我们需要的，这就是不准确的，所以，鉴于此，引入了折中的方式：书签查找（Bookmark Lookup)

简单点讲：书签查找就是通过索引页节点数据查找相关的列数据



**常用运算符总结—三种物理连接方式剖析**

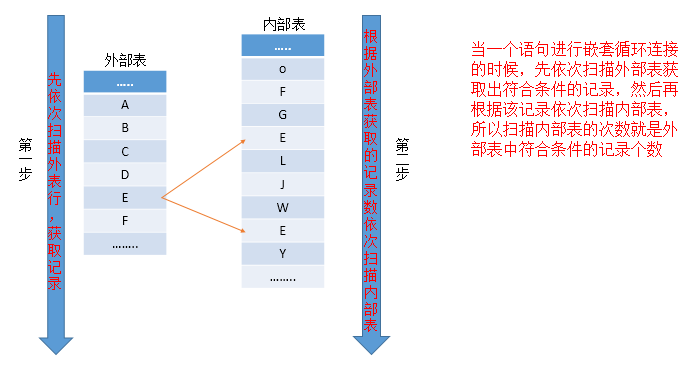
**一、数据连接**

数据连接是我们在写T-SQL语句的时候最常用的，通过两个表之间关联获取想要的数据。

SQL Server默认支持三种物理连接运算符：嵌套循环连接、合并连接以及哈希连接。三种连接各有用途，各有特点，不同的场景会数据库会为我们选择最优的连接方式

**a、嵌套循环连接（nested loops join)**

嵌套循环连接是最简单也是最基础的连接方式。两张表通过关键字进行关联，然后通过双层循环依次进行两张表的行进行关联，然后通过关键字进行筛选。



这种方法的消耗就是外表和内表的乘积，其实就是我们所称呼的笛卡尔积。所以消耗的大小是随着两张表的数据量增大而增加的，尤其是内部表，因为它是多次重复扫描的，所以我们在实践中的采取的措施就是减少每个外表或者内表的行数来减少消耗。

对于这种算法还有一种提高性能的方式，因为两张表是通过关键字进行关联的，所以在查询的时候对于底层的数据获取速度直接关乎着此算法的性能，这里优化的方式尽量使用两个表关键字为索引查询，提高查询速度。

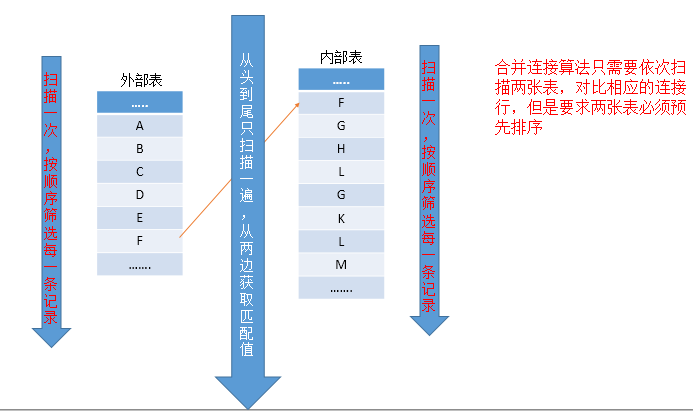
所以对嵌套循环连接连接的优化方式就是集中在这几点：**对两张表数据量的减少、连接关键字上建立索引、谓词查询条件上覆盖索引最好能减少符合谓词条件的记录数**

**b、合并连接(merge join)**

上面提到的嵌套循环连接方式存在着诸多的问题，尤其不适合两张表都是大表的情况下，因为它会产生N多次的全表扫描，很显然这种方式会严重的消耗资源。

鉴于上述原因，在数据库里又提供了另外一种连接方式：合并连接。记住这里没有说SQL Server所提供的，是因为此连接算法是市面所有的RDBMS所共同使用的一种连接算法。

合并连接是依次读取两张表的一行进行对比。如果两个行是相同的，则输出一个连接后的行并继续下一行的读取。如果行是不相同的，则舍弃两个输入中较少的那个并继续读取，一直到两个表中某一个表的行扫描结束，则执行完毕，所以该算法执行只会产生每张表一次扫描，并且不需要整张表扫描完就可以停止。



该算法要求按照两张表进行依次扫描对比，但是有两个前提条件：**1、必须预先将两张表的对应列进行排序；2、对两张表进行合并连接的条件必须存在等值连接**。

合并连接运算符总的消耗是和输入表中的行数成正比的，而且对表最多读取一次，这个和嵌套循环连接不一样。因此，合并连接对于大表的连接操作是一个比较好的选择项。

**对于合并连接可以从如下几点提高性能：**

1、两张表间的连接值内容列类型，如果两张表中的关联列都为唯一列，也就说都不存在重复值，这种关联性能是最好的，或者有一张表存在唯一列也可以，这种方式关联为一对多关联方式，这种方式也是我们最常用的，比我们经常使用的主从表关联查询；如果两张表中的关联列存在重复值，这样在两表进行关联的时候还需要借助第三张表来暂存重复的值，这第三张表叫做”worktable “是存放在Tempdb或者内存中，而这样性能就会有所影响。所以鉴于此，我们常做的优化方式有：关联连尽量采用聚集索引（唯一性）

2、我们知道采用该种算法的前提是，两张表都经过排序，所以我们在应用的时候，最好优先使用排序后的表关联。如果没有排序，也要选择的关联项为索引覆盖项，因为大表的排序是一个很耗资源的过程，我们选择索引覆盖列进行排序性能要远远好于普通列的排序。

**c、哈希连接(hash join)**

我们分析了上面的两种连接算法，两种算法各有特点，也各有自己的应用场景：嵌套循环连接适合于相对小的数据集连接，合并连接则应对与中型的数据集，但是又有它自己的缺点，比如要求必须有等值连接，并且需要预先排序等。

那对于大型的数据集合的连接数据库是怎么应对的呢？那就是哈希连接算法的应用场景了。

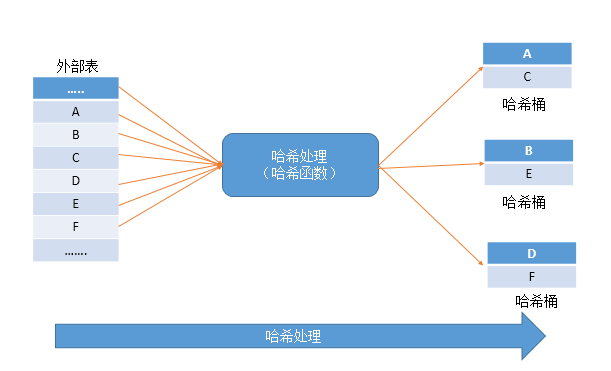
哈希连接对于大型数据集合的并行操作上都比其它方式要好很多，尤其适用于OLAP数据仓库的应用场景中。

哈希连接很多地方和合并连接类似，比如都需要至少一个等值连接，同样支持所有的外连接操作。但不同于合并连接的是，哈希连接不需要预先对输入数据集合排序，我们知道对于大表的排序操作是一个很大的消耗，所以去除排序操作，哈希操作性能无疑会提升很多。

哈希连接在执行的时候分为两个阶段：

构建阶段

在构建阶段，哈希连接从一个表中读入所有的行，将等值连接键的行机型哈希话处理，然后创建形成一个内存哈希表，而将原来列中行数据依次放入不同的哈希桶中。



探索阶段

在第一个阶段完成之后，开始进入第二个阶段探索阶段，该阶段哈希连接从第二个数据表中读入所有的行，同样也是在相同的等值连接键上进行哈希。哈希过程桶上一阶段，然后再从哈希表中探索匹配的行。

上述的过程中，在第一个阶段的构建阶段是阻塞的，也就是说在，哈希连接必须读入和处理所有的构建输入，之后才能返回行。而且这一过程是需要一块内存存储提供支持，并且利用的是哈希函数，所以相应的也会消耗CPU等。

并且上述流程过程中一般采用的是并发处理，充分利用资源，当然系统会对哈希的数量有所限制，如果数据量超大，也会发生内存溢出等问题，而对于这些问题的解决，SQL Server有它自身的处理方式。

**二、聚合操作**

聚合也是我们在写T-SQL语句的时候经常遇到的，我们来分析一下一些常用的聚合操作运算符的特性和可优化项。

**a、标量聚合**

标量聚合是一种常用的数据聚合方式，比如我们写的语句中利用的以下聚合函数：MAX()、MIN()、AVG()、COUNT()、SUM()

以上的这些数据结果项的输出基本都是通过流聚合的方式产生，并且这个运算符也被称为：标量聚合

总结出对于流聚合的一种优化方式：尽量避免排序产生，而要避免排序就需要将分组（Group by）字段在索引覆盖范围内。

b、哈希聚合

上述的流聚合的方式需要提前排序，我们知道排序是一个非常大的消耗过程，所以不适合大表的分组聚合操作，为了解决这个问题，又引入了另外一种聚合运算：哈希聚合

所谓的哈希聚合内部的方法和本篇前面提到的哈希连接机制一样。

哈希聚合不需要排序和过大的内存消耗，并且很容易并行执行计划，利用多CPU同步进行，但是有一个缺点就是：这一过程是阻塞的，也就说哈希聚合不会产生任何结果直到完整的输入。

所以在大数据表中采用哈希聚合是一个很好的应用场景

在SQL Server中并不是固定的语句就会形成特定的计划，并且生成的特定计划也不是总是最优的，这和数据库现有数据表中的内容分布、数据量、数据类型等诸多因素有关，而记录这些详细信息的就是统计信息。

所有的最优计划的选择都是基于现有统计信息来评估，如果我们的统计信息未及时更新，那么所评估出来最优的执行计划将不是最好的，有时候反而是最烂的。

**一、联合运算符**

所谓的联合运算符，其实应用最多的就两种：UNION ALL和UNION。

这两个运算符用法很简单，前者是将两个数据集结果合并，后者则是合并后进行去重操作，如果有过写T-SQL语句的码农都不会陌生

**并行运算**

**一、并行运算符**

在我们日常所写的T-SQL语句，并不是所有的最优执行计划都是一样的，其最优的执行计划的形成需要多方面的评估才可以，大部分根据SQL Server本身所形成的统计信息，然后对形成的多个执行计划进行评估，进而选出最优的执行方式。

在SQL Server根据库内容形成的统计信息进行评估的同时，还要参照当前运行的硬件资源，有时候它认为最优的方案可能当前硬件资源不支持，比如：内存限制、CPU限制、IO瓶颈等，所以执行计划的优劣还要依赖于底层硬件。

当SQL Server发现某个处理的数据集比较大，耗费资源比较多时，但此时硬件存在多颗CPU时，SQL Server会尝试使用并行的方法，把数据集拆分成若干个，若干个线程同时处理，来提高整体效率。

关于这些优化器的最重要原则的就是：尽可能的减少扫描范围，不管是表或者索引，当然走索引比表好，索引的量也是越少越好，最理想的情况是只有一条或者几条

维护数据库索引

索引管理

1. 索引创建

当我们要开始对表进行索引的创建的时候，首先明确的是，一张表内只能创建一个聚集索引，最多可以创建最多249个非聚集索引(SQL Server2005)，在SQL Server2008以后聚集索引数提升至999个，上一篇文章我们知道对于聚集索引项一般要创建上，而非聚集索引项要根据日常的T-SQL语句进行选择。

关于索引的选择是一个很考验调优能力的事情，大部分的情况下优质的索引新建全靠经验而论，有兴趣的可以点击查阅我前面的一系列关于分析查询计划的文章，掌握住里面的精髓才能有的放矢。

当然，小白级别的也可以参照如下方法尝试进行创建：

由于SQL Server有着自己的一套调优技巧，所以在我们每次运行的T-SQL语句应该怎样优化，SQL Server是了如指掌的，所以它会将缺失的索引项进行记录，用于提示使用者，尝试去建立这些索引。

**主要记录在以下几个DMV中**

sys.dm\_db\_missing\_index\_details

sys.dm\_db\_missing\_index\_groups

sys.dm\_db\_missing\_index\_group\_stats

sys.dm\_db\_missing\_index\_columns(index\_handle)

SELECT migs.group\_handle, mid.\*

FROM sys.dm\_db\_missing\_index\_group\_stats AS migs

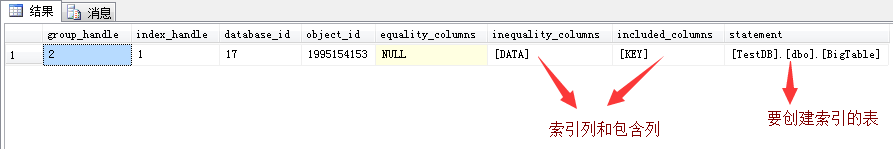
INNER JOIN sys.dm\_db\_missing\_index\_groups AS mig

ON (migs.group\_handle = mig.index\_group\_handle)

INNER JOIN sys.dm\_db\_missing\_index\_details AS mid

ON (mig.index\_handle = mid.index\_handle)

WHERE migs.group\_handle = 6



创建索引

CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index\_name

ON <object> ( column [ ASC | DESC ] [ ,...n ] )

[ INCLUDE ( column\_name [ ,...n ] ) ]

[ WHERE <filter\_predicate> ]

[ WITH ( <relational\_index\_option> [ ,...n ] ) ]

[ ON { partition\_scheme\_name ( column\_name )

| filegroup\_name

| default

}

]

[ FILESTREAM\_ON { filestream\_filegroup\_name | partition\_scheme\_name | "NULL" } ]

**数据库索引缺失查询语句**

SELECT  TableName = o.name ,   
migs\_Adv.index\_advantage ,   
s.avg\_user\_impact ,   
s.avg\_total\_user\_cost ,   
s.last\_user\_seek ,   
s.unique\_compiles ,   
d.index\_handle ,   
d.equality\_columns ,   
d.inequality\_columns ,   
d.included\_columns ,   
d.[statement]   
FROM    sys.dm\_db\_missing\_index\_group\_stats s   
INNER JOIN sys.dm\_db\_missing\_index\_groups g ON g.index\_group\_handle = s.group\_handle   
INNER JOIN sys.dm\_db\_missing\_index\_details d ON d.index\_handle = g.index\_handle   
INNER JOIN sys.objects o ON o.object\_id = d.object\_id   
INNER JOIN ( SELECT user\_seeks \* avg\_total\_user\_cost   
\* ( avg\_user\_impact \* 0.01 ) AS index\_advantage ,   
migs.\*   
FROM   sys.dm\_db\_missing\_index\_group\_stats migs   
) AS migs\_adv ON migs\_adv.group\_handle = g.index\_group\_handle   
ORDER BY migs\_adv.index\_advantage DESC ,   
s.avg\_user\_impact DESC

所以，大部分情况下，通过查看以上语句基本能确认到需要创建的索引项有哪些。

**提示：**但是，这里的DMV信息只是记录自上次SQL Server启动以后的信息项，也就是说每次重启之后这部分信息就丢失了，所以对于生产系统，建议确保运行了一段周期之后再进行查

**索引管理**

**a、索引的重建**

当我们发现索引索引覆盖范围不够或者存在大量索引锁片，影响性能的时候，我们就需要对索引进行重建。

索引范围的问题其实大部分来源于对于T-SQL语句性能的把握，也就是我们前面几篇文章中分析的需要调优的内容项。

而关于索引碎片的形成，也是源于数据库长时间的运行，大量的增删该查造成了B-Tree结构的不准确，确切的说是不能正确的提供平衡查询的性能，或者大量的数据分页造成索引碎片，进而增大了IO，影响了性能。

**索引碎片**

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT

DB\_NAME() AS DatbaseName

, SCHEMA\_NAME(o.Schema\_ID) AS SchemaName

, OBJECT\_NAME(s.[object\_id]) AS TableName

, i.name AS IndexName

, ROUND(s.avg\_fragmentation\_in\_percent,2) AS [Fragmentation %]

INTO #TempFragmentation

FROM sys.dm\_db\_index\_physical\_stats(db\_id(),null, null, null, null) s

INNER JOIN sys.indexes i ON s.[object\_id] = i.[object\_id]

AND s.index\_id = i.index\_id

INNER JOIN sys.objects o ON i.object\_id = O.object\_id

WHERE 1=2

EXEC sp\_MSForEachDB 'USE [?];

INSERT INTO #TempFragmentation

SELECT TOP 20

DB\_NAME() AS DatbaseName

, SCHEMA\_NAME(o.Schema\_ID) AS SchemaName

, OBJECT\_NAME(s.[object\_id]) AS TableName

, i.name AS IndexName

, ROUND(s.avg\_fragmentation\_in\_percent,2) AS [Fragmentation %]

FROM sys.dm\_db\_index\_physical\_stats(db\_id(),null, null, null, null) s

INNER JOIN sys.indexes i ON s.[object\_id] = i.[object\_id]

AND s.index\_id = i.index\_id

INNER JOIN sys.objects o ON i.object\_id = O.object\_id

WHERE s.database\_id = DB\_ID()

AND i.name IS NOT NULL

AND OBJECTPROPERTY(s.[object\_id], ''IsMsShipped'') = 0

ORDER BY [Fragmentation %] DESC'

SELECT top 20 \* FROM #TempFragmentation ORDER BY [Fragmentation %] DESC

DROP TABLE #TempFragmentation

维护的方式也就主要集中在以下几种：

1、重建索引

这种方式简单高效也就是我们上面分析的CREATE INDEX 命令后面加上DROP\_EXISTING方式。当然可以联机操作，操作方式参考文章前面

2、修改索引

这种方式是05版本以后才提供的，简单点将就是ALTER INDEX命令进行。其实底层的运行方式同索引重建，只不过这种方式更改的选项多一些。

3、索引重组

这种方式就是重新填充索引里面的数据，对于解决索引碎片的方式不如前面两种来的直接。不过也是一种推荐的方式，因为此方式在运行的时候，也是随时停止。

不像前面两种方式为原子性操作，并且业务阻塞。

**b、索引的禁用**

关于索引的禁用，这个功能也是SQL Server2005版本以后才出现的新功能，这个功能一般应用的不多。

因为大部分情况下将索引禁用了，还倒不如直接将索引删除掉来的直接。

但是，记住了既然SQL Server设计了它就是有它的用武之地的。

很多情况下，数据库在运行很长一段时间之后，会发生坏页的情况。而如果通过命令查找，发现损坏也处于索引项上，那么你所做的操作就是禁用这个索引（记住只能是禁用）

然后重新建立一个新索引就可以了。

在这种情况下我们可选的最快处理方式就是禁用该索引，因为一旦发生坏页的情况，该索引项是不允许删除的。

**c、索引的删除**

关于索引的删除，就不需要太多的介绍了，原因很简单，索引的存在会影响数据插入数据的速度，并且在查询的时候需要维护等多的锁，进而影响并发。

所以，一旦索引存在着一点优化的作用没有，我们就要及时的删除掉，因为百害而无一利嘛。

**查看未使用的索引**

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT

DB\_NAME() AS DatbaseName

, SCHEMA\_NAME(O.Schema\_ID) AS SchemaName

, OBJECT\_NAME(I.object\_id) AS TableName

, I.name AS IndexName

INTO #TempNeverUsedIndexes

FROM sys.indexes I INNER JOIN sys.objects O ON I.object\_id = O.object\_id

WHERE 1=2

EXEC sp\_MSForEachDB 'USE [?];

INSERT INTO #TempNeverUsedIndexes

SELECT

DB\_NAME() AS DatbaseName

, SCHEMA\_NAME(O.Schema\_ID) AS SchemaName

, OBJECT\_NAME(I.object\_id) AS TableName

, I.NAME AS IndexName

FROM sys.indexes I INNER JOIN sys.objects O ON I.object\_id = O.object\_id

LEFT OUTER JOIN sys.dm\_db\_index\_usage\_stats S ON S.object\_id = I.object\_id

AND I.index\_id = S.index\_id

AND DATABASE\_ID = DB\_ID()

WHERE OBJECTPROPERTY(O.object\_id,''IsMsShipped'') = 0

AND I.name IS NOT NULL

AND S.object\_id IS NULL'

SELECT \* FROM #TempNeverUsedIndexes

ORDER BY DatbaseName, SchemaName, TableName, IndexName

DROP TABLE #TempNeverUsedIndexes

当然，这些记录都是自动SQL Server启动以来未曾使用的索引，所以在生产系统中，一定要确保已经运行了一段周期了。

索引脚本的删除，很简单和表删除类似，直接drop掉就可以了

**查看那些经常被大量更新，但是却基本不适用的索引项**

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT

DB\_NAME() AS DatabaseName

, SCHEMA\_NAME(o.Schema\_ID) AS SchemaName

, OBJECT\_NAME(s.[object\_id]) AS TableName

, i.name AS IndexName

, s.user\_updates

, s.system\_seeks + s.system\_scans + s.system\_lookups

AS [System usage]

INTO #TempUnusedIndexes

FROM sys.dm\_db\_index\_usage\_stats s

INNER JOIN sys.indexes i ON s.[object\_id] = i.[object\_id]

AND s.index\_id = i.index\_id

INNER JOIN sys.objects o ON i.object\_id = O.object\_id

WHERE 1=2

EXEC sp\_MSForEachDB 'USE [?];

INSERT INTO #TempUnusedIndexes

SELECT TOP 20

DB\_NAME() AS DatabaseName

, SCHEMA\_NAME(o.Schema\_ID) AS SchemaName

, OBJECT\_NAME(s.[object\_id]) AS TableName

, i.name AS IndexName

, s.user\_updates

, s.system\_seeks + s.system\_scans + s.system\_lookups

AS [System usage]

FROM sys.dm\_db\_index\_usage\_stats s

INNER JOIN sys.indexes i ON s.[object\_id] = i.[object\_id]

AND s.index\_id = i.index\_id

INNER JOIN sys.objects o ON i.object\_id = O.object\_id

WHERE s.database\_id = DB\_ID()

AND OBJECTPROPERTY(s.[object\_id], ''IsMsShipped'') = 0

AND s.user\_seeks = 0

AND s.user\_scans = 0

AND s.user\_lookups = 0

AND i.name IS NOT NULL

ORDER BY s.user\_updates DESC'

SELECT TOP 20 \* FROM #TempUnusedIndexes ORDER BY [user\_updates] DESC

DROP TABLE #TempUnusedIndexes