[SQL Server 中WITH (NOLOCK)浅析](https://www.cnblogs.com/kerrycode/p/3946268.html)

2014-08-30 11:58  [潇湘隐者](https://www.cnblogs.com/kerrycode/)  阅读(135900)  评论(36)  [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=3946268)  [收藏](javascript:void(0))

**概念介绍**

开发人员喜欢在SQL脚本中使用WITH(NOLOCK), WITH(NOLOCK)其实是表提示（table\_hint）中的一种。它等同于 READUNCOMMITTED 。 具体的功能作用如下所示（摘自MSDN）：

   1： 指定允许脏读。不发布共享锁来阻止其他事务修改当前事务读取的数据，其他事务设置的排他锁不会阻碍当前事务读取锁定数据。允许脏读可能产生较多的并发操作，但其代价是读取以后会被其他事务回滚的数据修改。这可能会使您的事务出错，向用户显示从未提交过的数据，或者导致用户两次看到记录（或根本看不到记录）。有关脏读、不可重复读和幻读的详细信息，请参阅[并发影响](http://msdn.microsoft.com/zh-cn/library/ms190805(v=sql.105).aspx)。

   2： READUNCOMMITTED 和 NOLOCK 提示仅适用于数据锁。所有查询（包括那些带有 READUNCOMMITTED 和 NOLOCK 提示的查询）都会在编译和执行过程中获取 Sch-S（架构稳定性）锁。因此，当并发事务持有表的 Sch-M（架构修改）锁时，将阻塞查询。例如，数据定义语言 (DDL) 操作在修改表的架构信息之前获取 Sch-M 锁。所有并发查询（包括那些使用 READUNCOMMITTED 或 NOLOCK 提示运行的查询）都会在尝试获取 Sch-S 锁时被阻塞。相反，持有 Sch-S 锁的查询将阻塞尝试获取 Sch-M 锁的并发事务。有关锁行为的详细信息，请参阅[锁兼容性（数据库引擎）](http://msdn.microsoft.com/zh-cn/library/ms186396(v=sql.105).aspx)。

   3:  不能为通过插入、更新或删除操作修改过的表指定 READUNCOMMITTED 和 NOLOCK。SQL Server 查询优化器忽略 FROM 子句中应用于 UPDATE 或 DELETE 语句的目标表的 READUNCOMMITTED 和 NOLOCK 提示。

**功能与缺陷**

    使用WIHT(NOLOCK)有利也有弊，所以在决定使用之前，你一定需要了解清楚WITH(NOLOCK)的功能和缺陷，看其是否适合你的业务需求,不要觉得它能提升性能，稀里糊涂的就使用它。

    1：使用WITH(NOLOCK)时查询不受其它排他锁阻塞

    打开会话窗口1，执行下面脚本，不提交也不回滚事务，模拟事务真在执行过程当中

BEGIN TRAN

UPDATE TEST SET NAME='Timmy' WHERE OBJECT\_ID =1;

--ROLLBACK

   打开会话窗口2，执行下面脚本，你会发现执行结果一直查询不出来（其实才两条记录）。当前会话被阻塞了

SELECT \* FROM TEST;

    打开会话窗口3，执行下面脚本，查看阻塞情况，你会发现在会话2被会话1给阻塞了，会话2的等待类型为LCK\_M\_S：“当某任务正在等待获取共享锁时出现”

SELECT wt.blocking\_session\_id AS BlockingSessesionId

,sp.program\_name AS ProgramName

,COALESCE(sp.LOGINAME, sp.nt\_username) AS HostName

,ec1.client\_net\_address AS ClientIpAddress

,db.name AS DatabaseName

,wt.wait\_type AS WaitType

,ec1.connect\_time AS BlockingStartTime

,wt.WAIT\_DURATION\_MS/1000 AS WaitDuration

,ec1.session\_id AS BlockedSessionId

,h1.TEXT AS BlockedSQLText

,h2.TEXT AS BlockingSQLText

FROM sys.dm\_tran\_locks AS tl

INNER JOIN sys.databases db

ON db.database\_id = tl.resource\_database\_id

INNER JOIN sys.dm\_os\_waiting\_tasks AS wt

ON tl.lock\_owner\_address = wt.resource\_address

INNER JOIN sys.dm\_exec\_connections ec1

ON ec1.session\_id = tl.request\_session\_id

INNER JOIN sys.dm\_exec\_connections ec2

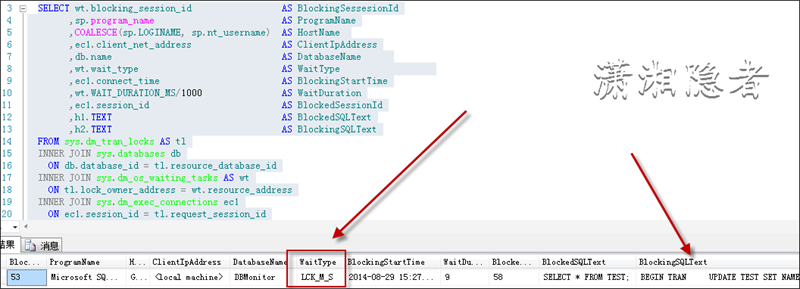
ON ec2.session\_id = wt.blocking\_session\_id

LEFT OUTER JOIN master.dbo.sysprocesses sp

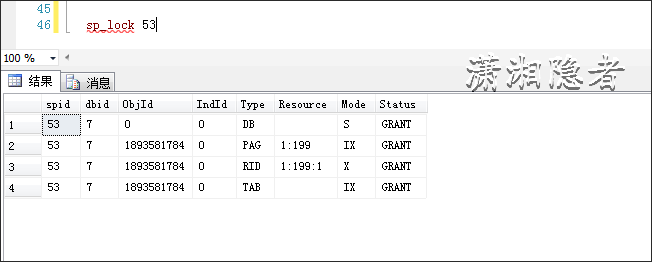
ON SP.spid = wt.blocking\_session\_id

CROSS APPLY sys.dm\_exec\_sql\_text(ec1.most\_recent\_sql\_handle) AS h1

CROSS APPLY sys.dm\_exec\_sql\_text(ec2.most\_recent\_sql\_handle) AS h2

[](https://images0.cnblogs.com/blog/73542/201408/301204003457660.png)

此时查看会话1（会话1的会话ID为53，执行脚本1前，可以用SELECT  @@spid查看会话ID）的锁信息情况，你会发现表TEST(ObjId=1893581784)持有的锁信息如下所示

[](https://images0.cnblogs.com/blog/73542/201408/301204032351645.png)

打开会话窗口4，执行下面脚本.你会发现查询结果很快就出来，会话4并不会被会话1阻塞。

    SELECT \* FROM TEST WITH(NOLOCK)

从上面模拟的这个小例子可以看出，正是由于加上WITH(NOLOCK)提示后，会话1中事务设置的排他锁不会阻碍当前事务读取锁定数据，所以会话4不会被阻塞，从而提升并发时查询性能。

2：WITH(NOLOCK) 不发布共享锁来阻止其他事务修改当前事务读取的数据，这个就不举例子了。

本质上WITH(NOLOCK)是通过减少锁和不受排它锁影响来减少阻塞，从而提高并发时的性能。所谓凡事有利也有弊，WITH(NOLOCK)在提升性能的同时，也会产生脏读现象。

如下所示，表TEST有两条记录，我准备更新OBJECT\_ID=1的记录，此时事务既没有提交也没有回滚

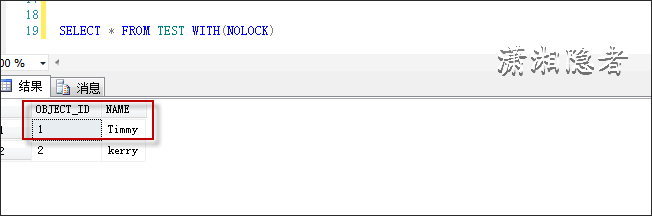
[](https://images0.cnblogs.com/blog/73542/201408/301204049076101.png)

BEGIN TRAN

UPDATE TEST SET NAME='Timmy' WHERE OBJECT\_ID =1;

--ROLLBACK

此时另外一个会话使用WITH(NOLOCK)查到的记录为未提交的记录值

[](https://images0.cnblogs.com/blog/73542/201408/301204060011901.png)

假如由于某种原因，该事务回滚了，那么我们读取到的OBJECT\_ID=1的记录就是一条脏数据。

脏读又称无效数据的读出，是指在数据库访问中，[事务](http://baike.baidu.com/view/121511.htm)T1将某一值修改，然后事务T2读取该值，此后T1因为某种原因撤销对该值的修改，这就导致了T2所读取到的数据是无效的。

**WITH(NOLOCK)使用场景**

什么时候可以使用WITH(NOLOCK)? 什么时候不能使用WITH(NOLOCK),这个要视你系统业务情况，综合考虑性能情况与业务要求来决定是否使用WITH(NOLOCK), 例如涉及到金融或会计成本之类的系统，出现脏读那是要产生严重问题的。关键业务系统也要慎重考虑。大体来说一般有下面一些场景可以使用WITH(NOLOCK)

   1: 基础数据表，这些表的数据很少变更。

   2：历史数据表，这些表的数据很少变更。

   3：业务允许脏读情况出现涉及的表。

   4：数据量超大的表，出于性能考虑，而允许脏读。

另外一点就是不要滥用WITH(NOLOCK),我发现有个奇怪现象，很多开发知道WITH(NOLOCK),但是有不了解脏读，习惯性的使用WITH(NOLOCK)。

**WITH(NOLOCK)与 NOLOCK区别**

为了搞清楚WITH(NOLOCK)与NOLOCK的区别，我查了大量的资料，我们先看看下面三个SQL语句有啥区别

    SELECT \* FROM TEST NOLOCK

    SELECT \* FROM TEST (NOLOCK);

    SELECT \* FROM TEST WITH(NOLOCK);

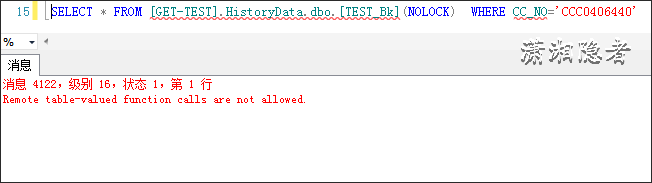
上面的问题概括起来也就是说NOLOCK、(NOLOCK)、 WITH(NOLOCK)的区别：

1： NOLOCK这样的写法，其实NOLOCK其实只是别名的作用，而没有任何实质作用。所以不要粗心将(NOLOCK)写成NOLOCK

2：（NOLOCK）与WITH(NOLOCK)其实功能上是一样的。(NOLOCK)只是WITH(NOLOCK)的别名,但是在SQL Server 2008及以后版本中，(NOLOCK)不推荐使用了，"不借助 WITH 关键字指定表提示”的写法已经过时了。 具体参见MSDN <http://msdn.microsoft.com/zh-cn/library/ms143729>%28SQL.100%29.aspx

    2.1  至于网上说WITH(NOLOCK)在SQL SERVER 2000不生效，我验证后发现完全是个谬论。

    2.2  在使用链接服务器的SQL当中，(NOLOCK)不会生效，WITH(NOLOCK)才会生效。如下所示

[](https://images0.cnblogs.com/blog/73542/201408/301204068133785.png)

    消息 4122，级别 16，状态 1，第 1 行

    Remote table-valued function calls are not allowed.

3.语法上有些许出入，如下所示

这种语法会报错

SELECT  \* FROM   sys.indexes  WITH(NOLOCK) AS i

-Msg 156, Level 15, State 1, Line 1

-Incorrect syntax near the keyword 'AS'.

这种语法正常

SELECT  \* FROM   sys.indexes  (NOLOCK) AS i

可以全部改写为下面语法

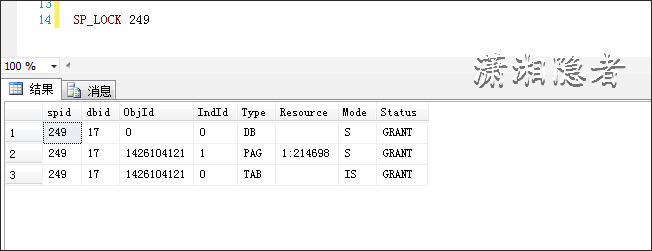
SELECT  \* FROM   sys.indexes   i WITH(NOLOCK)

SELECT  \* FROM   sys.indexes   i (NOLOCK)

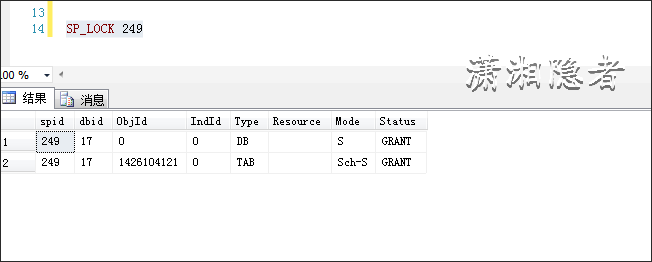
**WITH(NOLOCK)会不会产生锁**

    很多人误以为使用了WITH(NOLOCK)后，数据库库不会产生任何锁。实质上，使用了WITH(NOLOCK)后，数据库依然对该表对象生成Sch-S（架构稳定性）锁以及DB类型的共享锁， 如下所示，可以在一个会话中查询一个大表，然后在另外一个会话中查看锁信息(也可以使用SQL Profile查看会话锁信息)

    不使用WTIH(NOLOCK)

[](https://images0.cnblogs.com/blog/73542/201408/301204078603599.png)

  使用WITH(NOLOCK)

[](https://images0.cnblogs.com/blog/73542/201408/301204088769183.png)

  从上可以看出使用WITH(NOLOCK)后，数据库并不是不生成相关锁。  对比可以发现使用WITH(NOLOCK)后，数据库只会生成DB类型的共享锁、以及TAB类型的架构稳定性锁.

另外，使用WITH(NOLOCK)并不是说就不会被其它会话阻塞，依然可能会产生Schema Change Blocking

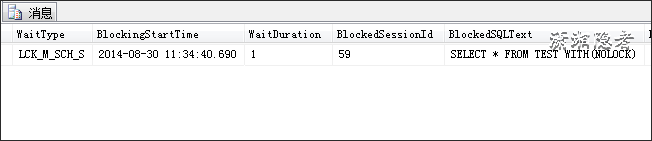
会话1：执行下面SQL语句，暂时不提交，模拟事务正在执行

BEGIN TRAN

ALTER TABLE TEST ADD Grade VARCHAR(10) ;

会话2：执行下面语句，你会发现会话被阻塞，截图如下所示。

SELECT \* FROM TEST WITH(NOLOCK)

[](https://images0.cnblogs.com/blog/73542/201408/301158081574024.png)

IIS当前请求队列查询：

进入CMD , 切换目录至： c:\windows\system32\inetsrv

常用命令： appcmd list request

异常时查看堆积的队列，判断当前请求堆积是由于web服务器或数据库效率问题导致。

若由于页面处理效率问题，则视情况回收IIS资源。

或者如果是由于数据库造成，则检查数据库服务器是否死锁。

sp\_who\_lock

或

SELECT TOP 200

[request\_id],[start\_time] AS '开始时间',[status] AS '状态',

[command] AS '命令',dest.[text] AS 'sql语句', DB\_NAME([database\_id]) AS '数据库名',

[blocking\_session\_id] AS '正在阻塞其他会话的会话ID',[wait\_type] AS '等待资源类型',

[wait\_time] AS '等待时间',[wait\_resource] AS '等待的资源',[reads] AS '物理读次数',

[writes] AS '写次数',[logical\_reads] AS '逻辑读次数',[row\_count] AS '返回结果行数'

FROM sys.[dm\_exec\_requests] AS der

CROSS APPLY

sys.[dm\_exec\_sql\_text](der.[sql\_handle]) AS dest

WHERE [session\_id]>50

ORDER BY [cpu\_time] DESC

SELECT TOP 200

[request\_id],[start\_time] AS '开始时间',[status] AS '状态',

[command] AS '命令',

REPLACE(REPLACE(dest.[text],'SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED',''),'SET NOCOUNT ON','') AS 'sql语句',

DB\_NAME([database\_id]) AS '数据库名',

[blocking\_session\_id] AS '正在阻塞其他会话的会话ID',[wait\_type] AS '等待资源类型',

[wait\_time] AS '等待时间',[wait\_resource] AS '等待的资源',[reads] AS '物理读次数',

[writes] AS '写次数',[logical\_reads] AS '逻辑读次数',[row\_count] AS '返回结果行数'

FROM sys.[dm\_exec\_requests] AS der

CROSS APPLY

sys.[dm\_exec\_sql\_text](der.[sql\_handle]) AS dest

WHERE [session\_id]>50

ORDER BY [cpu\_time] DESC

SELECT \* FROM master..sysdatabases WHERE name NOT IN ( 'master', 'model', 'msdb', 'tempdb', 'northwind','pubs' )

SELECT blocking\_session\_id, wait\_duration\_ms, session\_id FROM sys.dm\_os\_waiting\_tasks ORDER BY blocking\_session\_id

dbcc inputbuffer(166)

/\*Breaking “PAGE: 6:3:70133” down, we’ve got:

database\_id=6

data\_file\_id = 3

page\_number = 70133\*/

--PAGE: 7:1:30932824

DBCC TRACEON (3604);

GO

/\* DBCC PAGE (DatabaseName, FileNumber, PageNumber, DumpStyle)\*/

DBCC PAGE ('tlsoft',1,30932824,2);

GO

SELECT

sc.name as schema\_name,

so.name as object\_name,

si.name as index\_name

FROM sys.objects as so

JOIN sys.indexes as si on

so.object\_id=si.object\_id

JOIN sys.schemas AS sc on

so.schema\_id=sc.schema\_id

WHERE

so.object\_id = 2028234676

and si.index\_id = 117;

SELECT

sys.fn\_PhysLocFormatter (%%physloc%%),

\*

FROM YX\_T\_Ypdmb (NOLOCK)

WHERE sys.fn\_PhysLocFormatter (%%physloc%%) like '(1:31975334%'

RID查找

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; SET NOCOUNT ON;select '' e, b.rq,b.djh,m.lydjid, jg.sphh,b.chdm ,ch.chmc,

m.hpl gc\_hpl,kh.khmc g\_khmc, e.hpl g\_hpl,e.jcy g\_jcy,

e.id as g\_bgid,e.flowid g\_flowid,gc.khmc as gcmc,

jg.zdrq as cc\_jyrq, 100000+(jg.id%100000) as cc\_bgbh,

jg.hpl cc\_hpl,jg.kcsl as sjkcsl,case when jg.shbs=1 then '终审' else f.currentnodename end currentnodename,

f.currentusername,jg.id as cc\_bgid,931 as cc\_flowid,jg.skhpl --into #myzb

from dbo.cl\_v\_jhdjmxb b with(nolock)

inner join cl\_t\_chdmb ch with(nolock) on ch.tzid=1 and ch.chdm=b.chdm

inner join wl\_t\_dddjpmmx md with(nolock) on

md.mxid=b.mxid-- AND md.id=b.id ----连上ID后RID查找会不一样,

inner join cl\_t\_wljgb jg with(nolock) on jg.djlx=3905 and jg.bjdjh=b.djh and jg.bjrq=b.rq and jg.ph=md.ph-- and jg.zdrq>='2021-09-01' and jg.zdrq<dateadd(day,1,'2021-09-23')

left join (

SELECT a.djh,a.rq,a.type,b.ph,b.lydjid,b.hpl from t\_fabric a with(nolock)

INNER join t\_weight\_memo b with(nolock) on a.serviceuuid=b.serviceuuid

) m on m.djh=b.djh and m.rq=b.rq and m.type='gc' and m.ph=md.ph

inner join dbo.yx\_t\_khb kh on kh.khid=b.khid

left join yf\_t\_bjdlb e with(nolock) on e.id=b.jyid --or e.id=b.kcid-- 母体内补单关联出原报告210820

inner join yx\_t\_khb gc with(nolock) on gc.khid=jg.tzid

left join dbo.fl\_t\_flowrelation f with(nolock) on f.flowid=931 and f.dxid=jg.id

where b.djlx=2222 -- and ch.chmc like '%16042026-3%'

AND DATEDIFF(DAY,jg.zdrq,'2021-09-01')<=0

--and jg.zdrq>='2021-09-01' and jg.zdrq<dateadd(day,1,'2021-09-23')

and case when 'all'='all' then 1 when 'all'='db' and f.currentuserid=12742 then 1 else 0 end =1

and case when 1=1 then 1 when 1<>1 and gc.khid=1 then 1 else 0 end =1

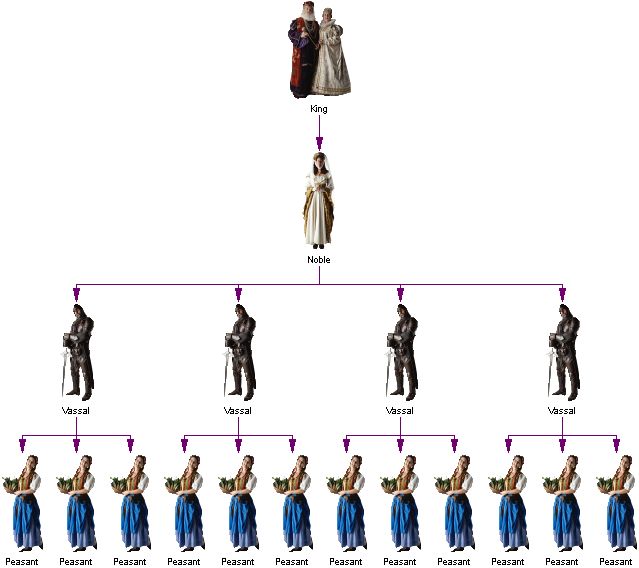
# [纸上谈兵: 堆 (heap)](https://www.cnblogs.com/vamei/archive/2013/03/20/2966612.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

堆(heap)又被为优先队列(priority queue)。尽管名为优先队列，但堆并不是队列。回忆一下，在[队列](http://www.cnblogs.com/vamei/archive/2013/03/15/2961729.html)中，我们可以进行的限定操作是dequeue和enqueue。dequeue是按照进入队列的先后顺序来取出元素。而在堆中，我们不是按照元素进入队列的先后顺序取出元素的，而是按照元素的优先级取出元素。

这就好像候机的时候，无论谁先到达候机厅，总是头等舱的乘客先登机，然后是商务舱的乘客，最后是经济舱的乘客。每个乘客都有头等舱、商务舱、经济舱三种个键值(key)中的一个。头等舱->商务舱->经济舱依次享有从高到低的优先级。

再比如，封建社会的等级制度，也是一个堆。在这个堆中，国王、贵族、骑士和农民是从高到低的优先级。



封建等级

Linux内核中的调度器(scheduler)会按照各个进程的优先级来安排CPU执行哪一个进程。计算机中通常有多个进程，每个进程有不同的优先级(该优先级的计算会综合多个因素，比如进程所需要耗费的时间，进程已经等待的时间，用户的优先级，用户设定的进程优先程度等等)。内核会找到优先级最高的进程，并执行。如果有优先级更高的进程被提交，那么调度器会转而安排该进程运行。优先级比较低的进程则会等待。“堆”是实现调度器的理想数据结构。

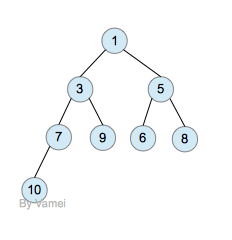
(Linux中可以使用nice命令来影响进程的优先级)

### 

### 堆的实现

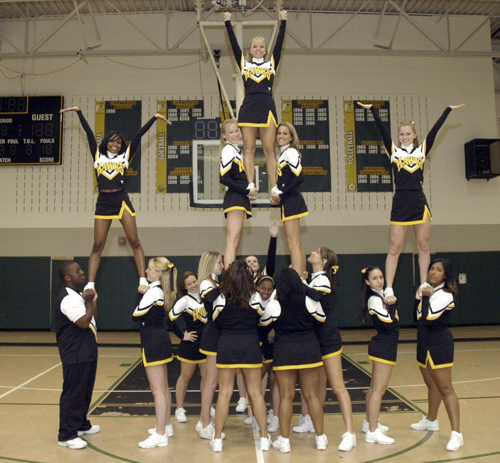
堆的一个经典的实现是完全二叉树(complete binary tree)。这样实现的堆成为二叉堆(binary heap)。

完全二叉树是增加了限定条件的二叉树。假设一个二叉树的深度为n。为了满足完全二叉树的要求，该二叉树的前n-1层必须填满，第n层也必须按照从左到右的顺序被填满，比如下图:



为了实现堆的操作，我们额外增加一个要求: 任意节点的优先级不小于它的子节点。如果在上图中，设定小的元素值享有高的优先级，那么上图就符合该要求。

这类似于“叠罗汉”。叠罗汉最重要的一点，就是让体重大的参与者站在最下面，让体重小的参与者站在上面 (体重小，优先级高)。为了让“堆”稳固，我们每次只允许最上面的参与者退出堆。也就是，每次取出的优先级最高的元素。

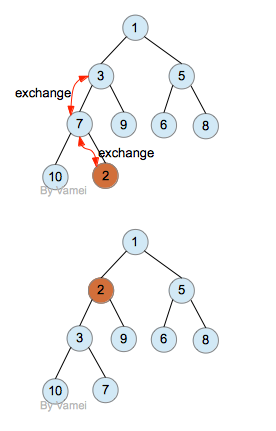


三个“叠罗汉”堆

我已经在[排序算法简介及其C实现](http://www.cnblogs.com/vamei/archive/2013/03/12/2948847.html)中实际使用了堆。堆的主要操作是插入和删除最小元素(元素值本身为优先级键值，小元素享有高优先级)。在插入或者删除操作之后，我们必须保持该实现应有的性质: 1. 完全二叉树 2. 每个节点值都小于或等于它的子节点。

在插入操作的时候，会破坏上述堆的性质，所以需要进行名为percolate\_up的操作，以进行恢复。新插入的节点new放在完全二叉树最后的位置，再和父节点比较。如果new节点比父节点小，那么交换两者。交换之后，继续和新的父节点比较…… 直到new节点不比父节点小，或者new节点成为根节点。这样得到的树，就恢复了堆的性质。

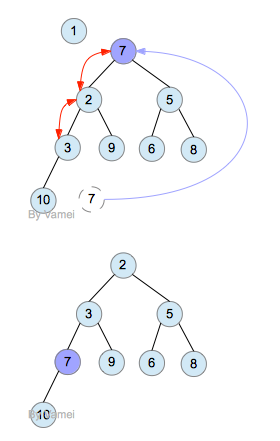
我们插入节点2:



插入

删除操作只能删除根节点。根节点删除后，我们会有两个子树，我们需要基于它们重构堆。进行percolate\_down的操作: 让最后一个节点last成为新的节点，从而构成一个新的二叉树。再将last节点不断的和子节点比较。如果last节点比两个子节点中小的那一个大，则和该子节点交换。直到last节点不大于任一子节点都小，或者last节点成为叶节点。

删除根节点1。如图:



删除根节点

下面是代码。与我们在[二叉搜索树](http://www.cnblogs.com/vamei/archive/2013/03/17/2962290.html)中使用表不同，我们这里使用数组来表示完全二叉树。数组下标为0的元素不用于储存节点，而用于记录完全二叉树中元素的总数。

[复制代码](javascript:void(0);)

/\* By Vamei

Use an big array to implement heap

DECLARE: int heap[MAXSIZE] in calling function

heap[0] : total nodes in the heap

for a node i, its children are i\*2 and i\*2+1 (if exists)

its parent is i/2 \*/

void insert(int new, int heap[])

{

int childIdx, parentIdx;

heap[0] = heap[0] + 1;

heap[heap[0]] = new;

/\* recover heap property \*/

percolate\_up(heap);

}

static void percolate\_up(int heap[]) {

int lightIdx, parentIdx;

lightIdx = heap[0];

parentIdx = lightIdx/2;

/\* lightIdx is root? && swap? \*/

while((parentIdx > 0) && (heap[lightIdx] < heap[parentIdx])) {

/\* swap \*/

swap(heap + lightIdx, heap + parentIdx);

lightIdx = parentIdx;

parentIdx = lightIdx/2;

}

}

int delete\_min(int heap[])

{

int min;

if (heap[0] < 1) {

/\* delete element from an empty heap \*/

printf("Error: delete\_min from an empty heap.");

exit(1);

}

/\* delete root

move the last leaf to the root \*/

min = heap[1];

swap(heap + 1, heap + heap[0]);

heap[0] -= 1;

/\* recover heap property \*/

percolate\_down(heap);

return min;

}

static void percolate\_down(int heap[]) {

int heavyIdx;

int childIdx1, childIdx2, minIdx;

int sign; /\* state variable, 1: swap; 0: no swap \*/

heavyIdx = 1;

do {

sign = 0;

childIdx1 = heavyIdx\*2;

childIdx2 = childIdx1 + 1;

if (childIdx1 > heap[0]) {

/\* both children are null \*/

break;

}

else if (childIdx2 > heap[0]) {

/\* right children is null \*/

minIdx = childIdx1;

}

else {

minIdx = (heap[childIdx1] < heap[childIdx2]) ?

childIdx1 : childIdx2;

}

if (heap[heavyIdx] > heap[minIdx]) {

/\* swap with child \*/

swap(heap + heavyIdx, heap + minIdx);

heavyIdx = minIdx;

sign = 1;

}

} while(sign == 1);

}

[复制代码](javascript:void(0);)

你可以尝试一下构建自己的main函数，测试相关的操作。

### 总结

堆，优先级

插入元素，删除最大优先级元素