并发处理:  
  
1.加synchronized锁单线程处理、缺点：

**1.处理速度也会很慢**

**2.只适合单点的情况**  
3.**无法做到细粒度控制**

2.redis分布式锁：  
  
**1.可以支撑每秒10多万的并发，**

**2.支持分布式，**

**3.可以更细粒的控制代码（多台机器上多个线程对一个数据进行操作的互斥）**  
  
SETNX key value  
  
将key设置值为value,如果key不存在，这种情况下等同于SET命令，当key存在时，什么也不做  
  
GETSET key value  
  
自动将key对应到value并且返回原来key和对应的value,如果key存在但是对应的value不是字符串，就返回错误

**还有Redis分布式锁其实是非公平的。**

使用Zookeeper可以解决公平锁问题，客户端在ZK中创建的临时节点是有序的，每次锁被释放时，ZK可以通知最小节点来获取锁，保证了公平。

**还有想到的可以用一个redis消息队列去存储（IO多路复用）应该也是可以的。**

**下面介绍一下，可以监听客户端连接，以及有公平锁的ZooKeeper：**

我自己写了一个redis分布式锁的模块。 但是带来了一个问题是，如果**客户端自己因为内存泄露被系统内核给oom干掉了**。

在分布式的架构下，**一堆的节点去获取锁是徒劳的**，只能等我们先前redis的**TTL自动消逝**….当然我自己也扩展了一个追加时间戳的方式，来判断他的进程在不在，但是可能会遇到的问题是，他因为hbase的堵塞，会消耗不少的时间…我不能及时的，实时的推送我的任务时间戳….

我想大家就算是在正常的情况下，也是会遇到任务堵塞的情况，又因为你的程序不健全，没有做好timeout超时的释放机制，单纯的用expire做ttl的控制不是那么的合理。 如果应用在我们项目下，会遇到任务正在干着，**但是因为某个原因堵塞了，但是自动解锁的时间马上就要到了**…  这算是个悲催的场景了….

其实我们可以**监听客户端的链接的状态**….   我们这里可以采用zookeeper来解决这类的问题，**zookeeper会自动长连接的，如果客户端中断了连接，**不管是你主动还是被动，我都会在我的注册列表里卖弄，干掉你，并且解锁

自己也懒得写了，直接搜了个开源的**模块**  https://github.com/tinyogre/zklock 。

#创建锁

z = zklock.Lock('nima')

try:

#查询是否已经锁定了

if z.acquire():

print "zklocktest: Lock acquired"

time.sleep(20)

# 解锁

z.release()

except:

z.release()

with zklock.ScopedLock("scoped\_lock\_test", block=False) as z:

if z.acquired:

print "Locked!"

time.sleep(20)

else:

print "Could not obtain lock!"

print "zklocktest: Exiting"

zk节点(znode)可以分为如下四类：

PERSISTENT：持续的，相比于EPHEMERAL，不会随着client session的close/expire而消失  
PERSISTENT\_SEQUENTIAL：顺序的，会自动在节点名后面添加一个自增计数，格式为%010d  
EPHEMERAL：临时节点，生命周期依赖于client session，对应session close/expire后其znode也会消失，临时节点不能有子节点

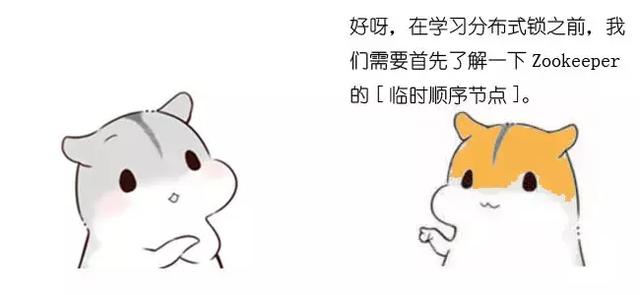
(我们这里采用的就是 临时节点，因为他有个session的概念….当客户端出现异常后，其他的客户端可以watch监听到的)  
EPHEMERAL\_SEQUENTIAL  
该方法可能触发如下异常：  
NodeExistsError：当要创建的节点已经存在时  
NoNodeError：当makepath为False且祖先节点不存在时  
NoChildrenForEphemeralsError：父节点为临时节点，在一个临时节点下面创建子节点会报该异常  
ZookeeperError：节点值太大，zk默认节点值限制为1M  
ZookeeperError：服务器返回一个非0状态码

**最后，借这个机会，简单介绍一下Zk。**

在之前的文章中，介绍了如何使用redis实现分布式锁。那么，如何用Zookeeper来实现分布式锁呢？

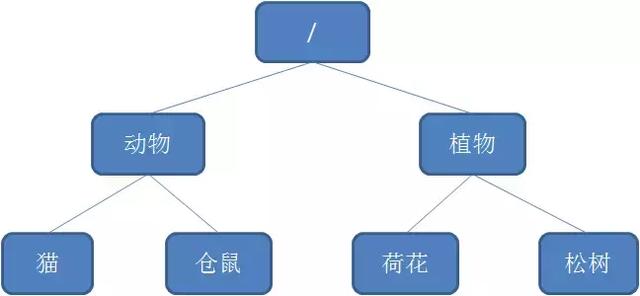
这一次我们会为大家详细讲述。





**什么是临时顺序节点？**

让我们来回顾一下Zookeeper节点的概念：



Zookeeper的数据存储结构就像一棵树，这棵树由节点组成，这种节点叫做Znode。

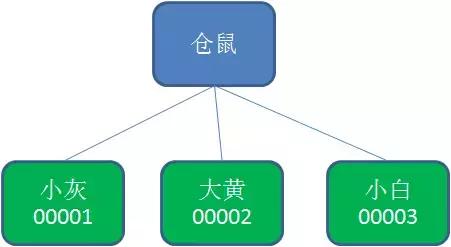
Znode分为四种类型：

**1.持久节点 （PERSISTENT）**

默认的节点类型。创建节点的客户端与zookeeper断开连接后，该节点依旧存在 。

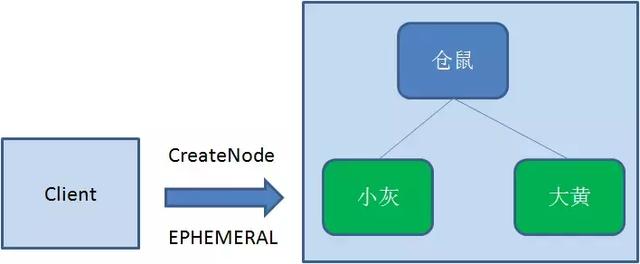
**2.持久节点顺序节点（PERSISTENT\_SEQUENTIAL）**

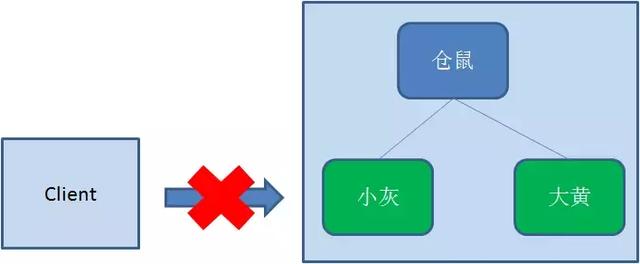
所谓顺序节点，就是在创建节点时，Zookeeper根据创建的时间顺序给该节点名称进行编号：

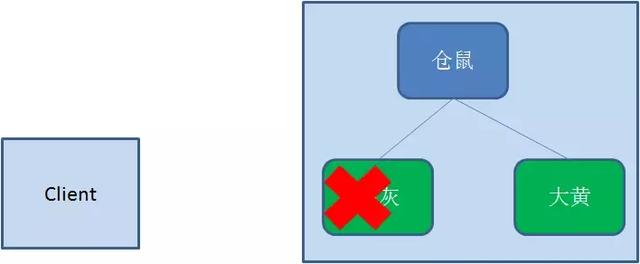


**3.临时节点（EPHEMERAL）**

和持久节点相反，当创建节点的客户端与zookeeper断开连接后，临时节点会被删除：







**4.临时顺序节点（EPHEMERAL\_SEQUENTIAL）**

顾名思义，临时顺序节点结合和临时节点和顺序节点的特点：在创建节点时，Zookeeper根据创建的时间顺序给该节点名称进行编号；当创建节点的客户端与zookeeper断开连接后，临时节点会被删除。



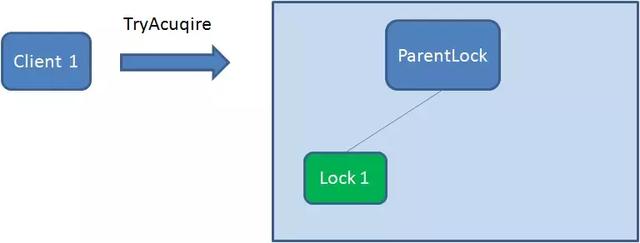


**Zookeeper分布式锁的原理**

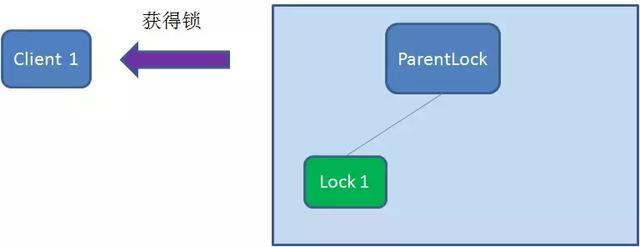
Zookeeper分布式锁恰恰应用了临时顺序节点。具体如何实现呢？让我们来看一看详细步骤：

**获取锁**

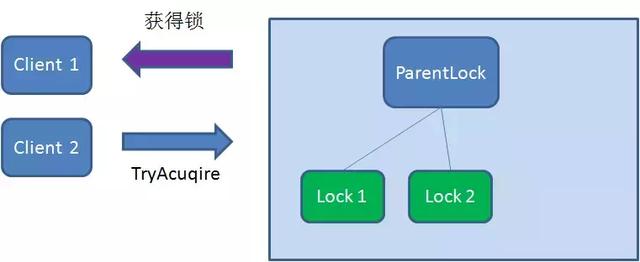
首先，在Zookeeper当中创建一个持久节点ParentLock。当第一个客户端想要获得锁时，需要在ParentLock这个节点下面创建一个**临时顺序节点**Lock1。



之后，Client1查找ParentLock下面所有的临时顺序节点并排序，判断自己所创建的节点Lock1是不是顺序最靠前的一个。如果是第一个节点，则成功获得锁。

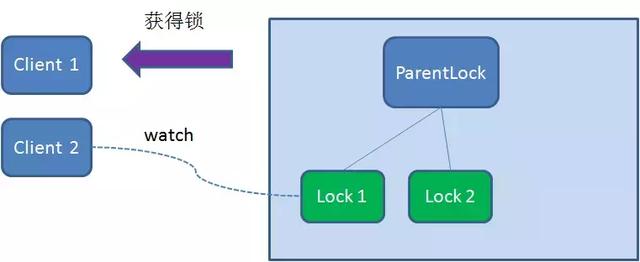


这时候，如果再有一个客户端 Client2 前来获取锁，则在ParentLock下载再创建一个临时顺序节点Lock2。

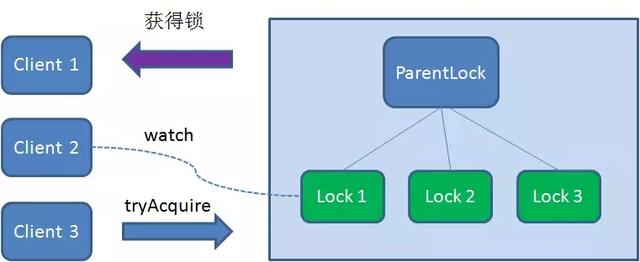


Client2查找ParentLock下面所有的临时顺序节点并排序，判断自己所创建的节点Lock2是不是顺序最靠前的一个，结果发现节点Lock2并不是最小的。

于是，Client2向排序仅比它靠前的节点Lock1注册**Watcher**，用于监听Lock1节点是否存在。这意味着Client2抢锁失败，进入了等待状态。

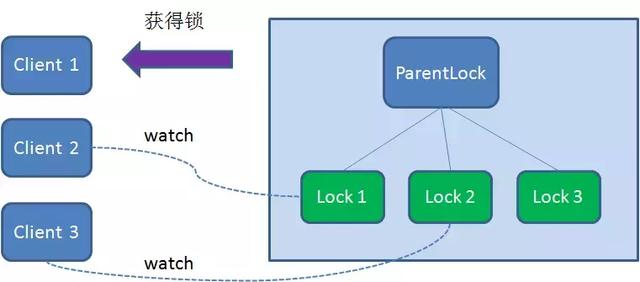


这时候，如果又有一个客户端Client3前来获取锁，则在ParentLock下载再创建一个临时顺序节点Lock3。

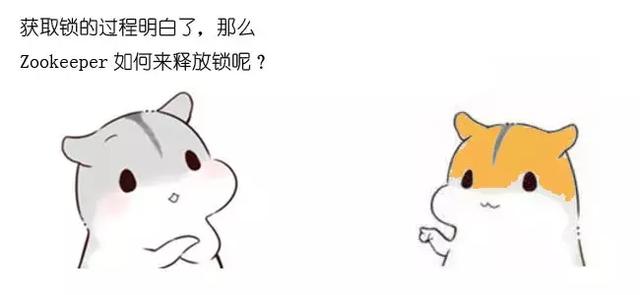


Client3查找ParentLock下面所有的临时顺序节点并排序，判断自己所创建的节点Lock3是不是顺序最靠前的一个，结果同样发现节点Lock3并不是最小的。

于是，Client3向排序仅比它靠前的节点**Lock2**注册Watcher，用于监听Lock2节点是否存在。这意味着Client3同样抢锁失败，进入了等待状态。



这样一来，Client1得到了锁，Client2监听了Lock1，Client3监听了Lock2。这恰恰形成了一个等待队列，很像是Java当中ReentrantLock所依赖的**AQS**（AbstractQueuedSynchronizer）。



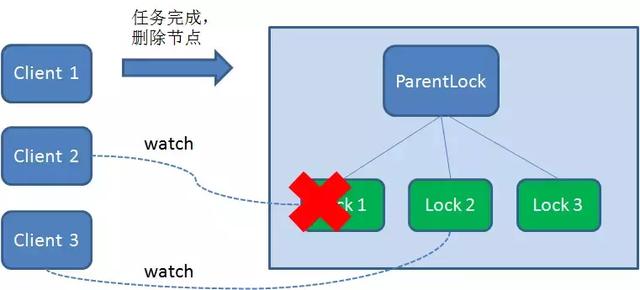


**释放锁**

释放锁分为两种情况：

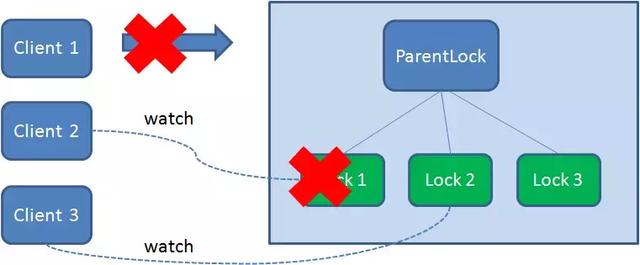
**1.任务完成，客户端显示释放**

当任务完成时，Client1会显示调用删除节点Lock1的指令。

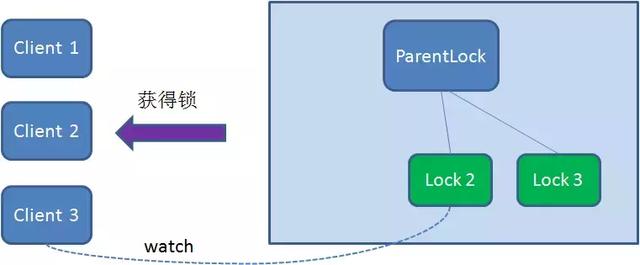


**2.任务执行过程中，客户端崩溃**

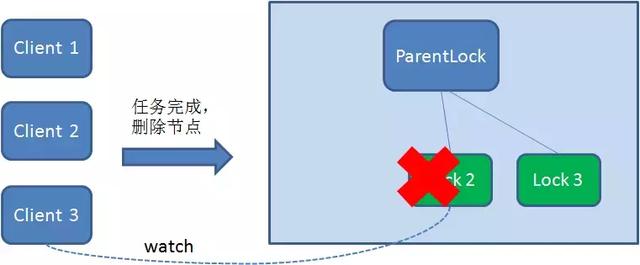
获得锁的Client1在任务执行过程中，如果Duang的一声崩溃，则会断开与Zookeeper服务端的链接。根据临时节点的特性，相关联的节点Lock1会随之自动删除。



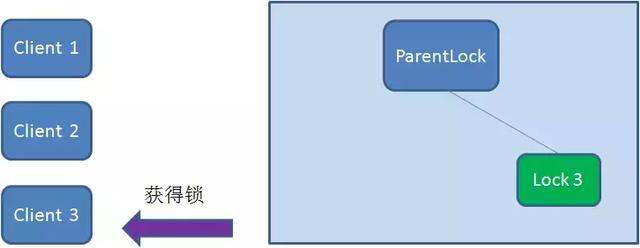
由于Client2一直监听着Lock1的存在状态，当Lock1节点被删除，Client2会立刻收到通知。这时候Client2会再次查询ParentLock下面的所有节点，确认自己创建的节点Lock2是不是目前最小的节点。如果是最小，则Client2顺理成章获得了锁。



同理，如果Client2也因为任务完成或者节点崩溃而删除了节点Lock2，那么Client3就会接到通知。



最终，Client3成功得到了锁。

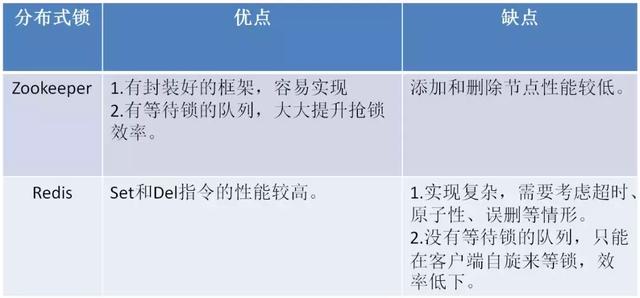






**Zookeeper和Redis分布式锁的比较**

下面的表格总结了Zookeeper和Redis分布式锁的优缺点：



有人说Zookeeper实现的分布式锁支持可重入，Redis实现的分布式锁不支持可重入，这是**错误的观点**。两者都可以在客户端实现可重入逻辑。