# 疑问问题

## 1.如何保证资源池线程安全

## 2.如何让资源池能够应付高并发场景

# 客户线程

## 代码步骤分析

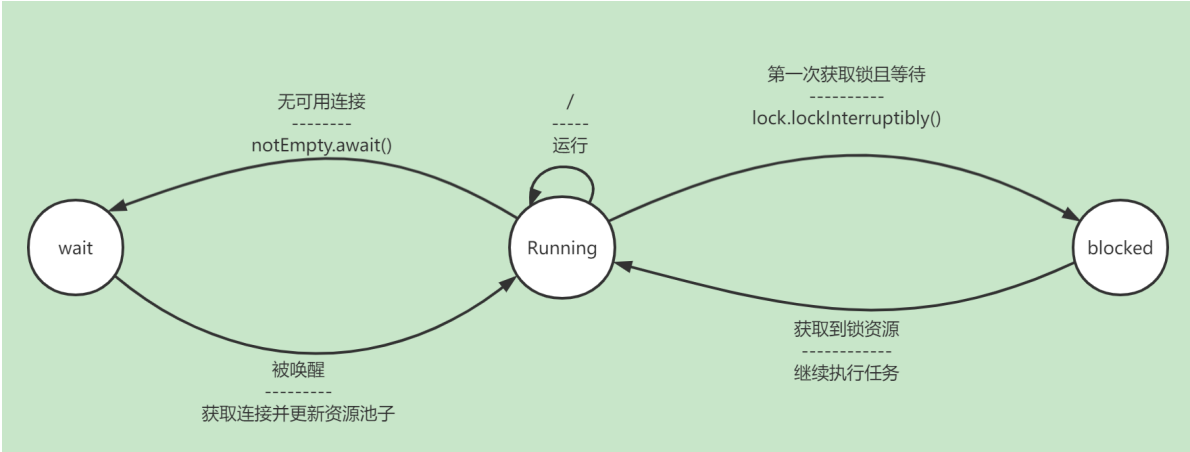
分析完创建连接线程CreateThread的主要步骤和FSM后，现在来分析客户线程获取连接时的情况。主要时分析客户线程调用getConnectionInternal方法和takeLast方法(pollLast方法情况大致相同)。

```
private DruidPooledConnection getConnectionInternal(long maxWait) throws
SQLException {
    try {
        lock.lockInterruptibly(); // 第一次获取锁
    } catch (InterruptedException e) {
        connectErrorCountUpdater.incrementAndGet(this);
        throw new SQLException("interrupt", e);
    }

    if (maxWait > 0) {
        holder = pollLast(nanos);
    } else {
        holder = takeLast(); // 分析这个方法
    }
    /*
        这里takeLast方法中的代码
         while (poolingCount == 0) {
            emptySignal(); // send signal to CreateThread create connection
            try {
                notEmpty.await(); // signal by recycle or creator
                                  // 让出锁资源，等待通知有可用连接
            } finally {
                notEmptyWaitThreadCount--;
            }
         }
         // 被唤醒之后继续执行任务，更新资源池资源。
    */
    // 回到getConnectionInternal方法
    // 完成剩下更新资源池资源的任务，最后释放锁
}
```

## FSM

以下是客户线程在getConnectionInternal的FSM：




## 阶段总结

在利用锁来保证线程安全的时候，Druid源码中有个特征让我比较有感悟的是，作者时候很清楚临界区的范围，仅仅在资源池真正要被修改的时候，才会进行上锁。这与《Java并发编程实战》中提到的“找出清晰的任务边界，以提供一种自然的并行工作结构来提升并发性”。

在createThread中，在创建物理数据库连接的时候，就会把锁资源让渡出来。当创建完成后，才会选择获取锁再把新的连接放入到池子中。

在客户线程中，除了实际获取连接时要获取锁，其他部分的代码（当然排除要更新敏感资源的部分）都是可以多线程执行的，以此来提高并发性。

到目前了解完客户线程和创建线程各自的执行步骤，接下来的目标是了解这两个线程的通信模型。