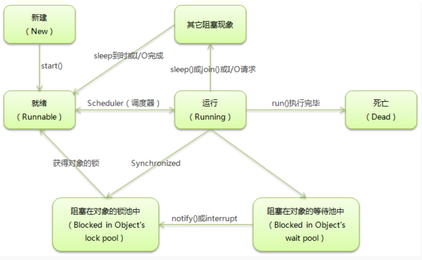
> Bugly支持ANR上报后，很多童鞋反馈看不懂上报上来的trace文件，精神哥很担忧，因为读懂trace文件是分析ANR问题的最关键的一步。
> **Trace文件是个什么鬼？**
> App的进程发生ANR时，系统让活跃的Top进程都进行了一下dump，进程中的各种Thread就都dump到这个trace文件里了，所以trace文件中包含了每一条线程的运行时状态。
> 刚好我们的美女攻城狮sunny(邹灵灵)最近收集了这块的内容，下面给大家详细介绍Thread Dump到底是个什么鬼，相信看完的童鞋，读懂trace文件就So easy了！

一、java线程的状态转换介绍



1.1新建状态（New）

用new语句创建的线程处于新建状态，此时它和其他Java对象一样，仅仅在堆区中被分配了内存。

1.2就绪状态（Runnable）

当一个线程对象创建后，其他线程调用它的start()方法，该线程就进入就绪状态，Java虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得CPU的使用权。

1.3运行状态（Running）

处于这个状态的线程占用CPU，执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。

1.4阻塞状态（Blocked）

阻塞状态是指线程因为某些原因放弃CPU，暂时停止运行。当线程处于阻塞状态时，Java虚拟机不会给线程分配CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。

阻塞状态可分为以下3种：
1）位于对象等待池中的阻塞状态（Blocked in object's wait pool）：当线程处于运行状态时，如果执行了某个对象的wait()方法，Java虚拟机就会把线程放到这个对象的等待池中，这涉及到“线程通信”的内容。
2）位于对象锁池中的阻塞状态（Blocked in object's lock pool）：当线程处于运行状态时，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其他线程占用，Java虚拟机就会把这个线程放到这个对象的锁池中，这涉及到“线程同步”的内容。
3）其他阻塞状态（Otherwise Blocked）：当前线程执行了sleep()方法，或者调用了其他线程的join()方法，或者发出了I/O请求时，就会进入这个状态。

1.5死亡状态（Dead）

当线程退出run()方法时，就进入死亡状态，该线程结束生命周期。

二、Thread Dump分析

2.1首先介绍一下Thread Dump信息的各个部分

线程info信息块：

```
1. "Timer-0" daemon prio=10tid=0xac190c00 nid=0xaef in Object.wait() [0xae77d000]
2. java.lang.Thread.State: TIMED_WAITING (on object monitor)
3. atjava.lang.Object.wait(Native Method)
4. -waiting on <0xb3885f60> (a java.util.TaskQueue) ###继续wait
5. atjava.util.TimerThread.mainLoop(Timer.java:509)
6. -locked <0xb3885f60> (a java.util.TaskQueue) ###已经locked
7. atjava.util.TimerThread.run(Timer.java:462)
```

* 线程名称：Timer-0
* 线程类型：daemon
* 优先级：10，默认是5
* **jvm线程id**：tid=0xac190c00，jvm内部线程的唯一标识（一般通过java.lang.Thread.getId()获取，通常用自增方式实现。）
* **对应系统线程id（Native Thread ID）**：nid=0xaef，和top命令查看的线程pid对应，不过一个是10进制，一个是16进制。（可以通过命令：top -H -p pid，查看该进程的所有线程信息）
* 线程状态：in Object.wait().
* 起始栈地址：[0xae77d000]
* **Java thread statck trace**：上面2~7行的信息。到目前为止这是最重要的数据，Java stack trace提供了大部分信息来精确定位问题根源。

对于thread dump信息，主要关注的是线程的状态和其执行堆栈。现在针对这两个重点部分进行讲解.

1、Java thread statck trace详解

堆栈信息应该逆向解读：程序先执行的是第7行，然后是第6行，依次类推。

```
- locked <0xb3885f60> (a java.util.ArrayList)
- waiting on <0xb3885f60> (a java.util.ArrayList)
```

也就是说对象先上锁，锁住对象0xb3885f60，然后释放该对象锁，进入waiting状态。

为啥会出现这样的情况呢？看看下面的java代码示例，就会明白：

```
synchronized(obj) {
 .........
 obj.wait();
 .........
}
```

在堆栈的第一行信息中，进一步标明了线程在代码级的状态，例如：

```
java.lang.Thread.State: TIMED_WAITING (parking)
```

解释如下：

```
|blocked|
This thread tried to enter asynchronized block, but the lock was taken by another thread. This thread isblocked until the lock gets released.
|blocked (on thin lock)|
This is the same state asblocked, but the lock in question is a thin lock.
|waiting|
This thread calledObject.wait() on an object. The thread will remain there until some otherthread sends a notification to that object.
|sleeping|
This thread calledjava.lang.Thread.sleep().
|parked|
This thread calledjava.util.concurrent.locks.LockSupport.park().
|suspended|
The thread's execution wassuspended by java.lang.Thread.suspend() or a JVMTI agent call.
```

2、线程状态详解

```
Runnable
_The thread is either running or ready to run when it gets its CPU turn._
Wait on condition
_The thread is either sleeping or waiting to be notified by another thread._
```

该状态出现在线程等待某个条件的发生或者sleep。具体是什么原因，可以结合 stacktrace来分析。

最常见的情况是线程在等待网络的读写，比如当网络数据没有准备好读时，线程处于这种等待状态，而一旦有数据准备好读之后，线程会重新激活，读取并处理数据。在Java引入New IO之前，对于每个网络连接，都有一个对应的线程来处理网络的读写操作，即使没有可读写的数据，线程仍然阻塞在读写操作上，这样有可能造成资源浪费，而且给操作系统的线程调度也带来压力。在New IO里采用了新的机制，编写的服务器程序的性能和可扩展性都得到提高。

如果发现有大量的线程都处在Wait on condition，从线程 stack看， 正等待网络读写，这可能是一个网络瓶颈的征兆。因为网络阻塞导致线程无法执行。

一种情况是网络非常忙，几乎消耗了所有的带宽，仍然有大量数据等待网络读写；另一种情况也可能是网络空闲，但由于路由等问题，导致包无法正常的到达。

所以要结合系统的一些性能观察工具来综合分析，比如netstat统计单位时间的发送包的数目，看是否很明显超过了所在网络带宽的限制；观察CPU的利用率，看系统态的CPU时间是否明显大于用户态的CPU时间；这些都指向由于网络带宽所限导致的网络瓶颈。另外一种出现 Wait on condition的常见情况是该线程在 sleep，等待 sleep的时间到了，将被唤醒。

```
Waiting for Monitor Entry and in Object.wait()
_The thread is waiting to getthe lock for an object (some other thread may be holding the lock). Thishappens if two or more threads try to execute synchronized code. Note that thelock is always for an object and not for individual methods._
```

在多线程的 JAVA程序中，实现线程之间的同步，就要说说Monitor。

Monitor是Java中用以实现线程之间的互斥与协作的主要手段，它可以看成是对象或者 Class的锁。每一个对象都有，也仅有一个monitor。每个Monitor在某个时刻，只能被一个线程拥有，该线程就是 “ActiveThread”，而其它线程都是“Waiting Thread”，分别在两个队列“ Entry Set”和“Wait Set”里面等候。在“Entry Set”中等待的线程状态是“Waiting for monitorentry”，而在“Wait Set”中等待的线程状态是“in Object.wait()”。

先看 “Entry Set”里面的线程。
我们称被synchronized保护起来的代码段为临界区。当一个线程申请进入临界区时，它就进入了 “Entry Set”队列。对应的 code就像：

```
synchronized(obj) {
 .........
}
```

这时有两种可能性：
1）该 monitor不被其它线程拥有， Entry Set里面也没有其它等待线程。本线程即成为相应类或者对象的 Monitor的 Owner，执行临界区的代码。
2）该 monitor被其它线程拥有，本线程在 Entry Set队列中等待。
在第一种情况下，线程将处于 “Runnable”的状态，而第二种情况下，线程 DUMP会显示处于 “waiting for monitor entry”。

临界区的设置，是为了保证其内部的代码执行的原子性和完整性。但是因为临界区在任何时间只允许线程串行通过，这和我们多线程的程序的初衷是相反的。如果在多线程的程序中，大量使用 synchronized，或者不适当的使用了它，会造成大量线程在临界区的入口等待，造成系统的性能大幅下降。如果在线程 DUMP中发现了这个情况，应该审查源码，改进程序。

再看“Wait Set”里面的线程。当线程获得了Monitor，进入了临界区之后，如果发现线程继续运行的条件没有满足，它则调用对象（一般就是被synchronized的对象）的wait() 方法，放弃Monitor，进入“Wait Set”队列。只有当别的线程在该对象上调用了notify() 或者notifyAll()，“Wait Set”队列中线程才得到机会去竞争，但是只有一个线程获得对象的Monitor，恢复到运行态。在“Wait Set”中的线程， DUMP中表现为： in Object.wait()。

一般，CPU很忙时，则关注runnable的线程，CPU很闲时，则关注waiting for monitor entry的线程。