什么是 Zookeeper?

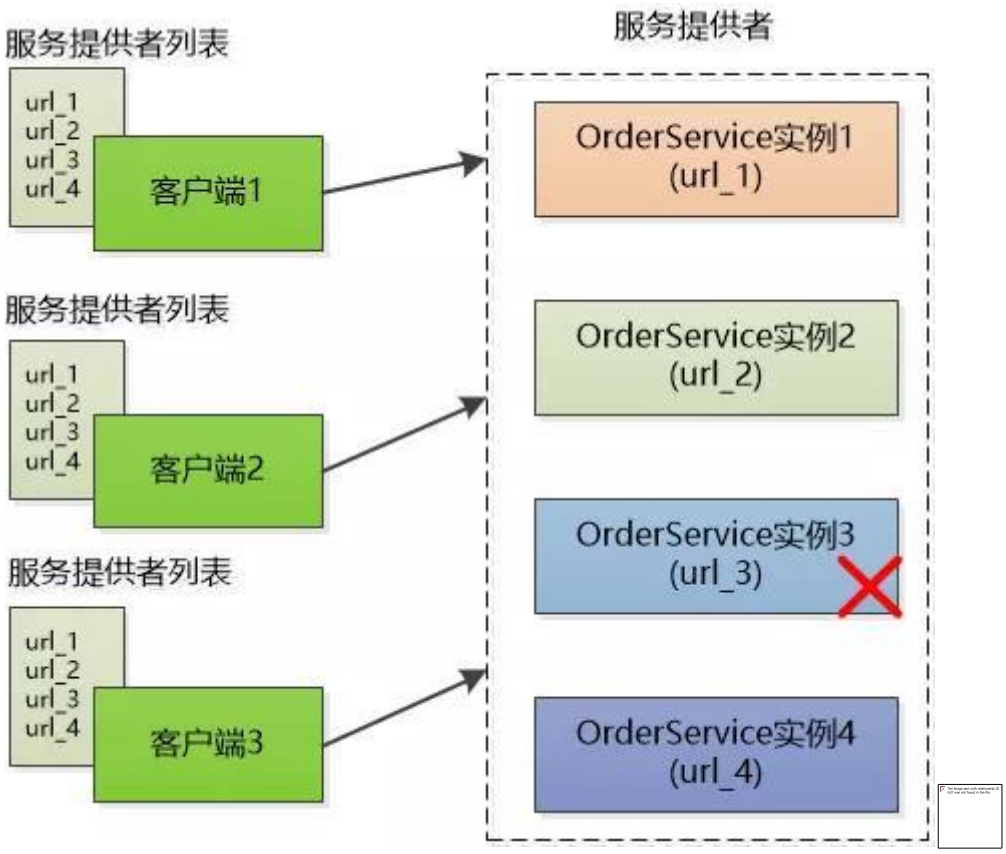
张大胖所在的公司这几年发展得相当不错，业务激增，人员也迅速扩展，转眼之间，张大胖已经成为公司的“资深”员工了，更重要的是，经过这些年的不懈努力，他终于坐上了架构师的宝座。

但是大胖很快发现，这架构师真不是好当的，技术选型、架构设计，尤其是大家搞不定的技术难点，最终都得自己扛起来。沟通、说服、妥协、甚至争吵都是家常便饭，比自己之前单纯做开发的时候难多了。

公司的 IT 系统早已经从单机转向了分布式，分布式系统带来了巨大的挑战。这周一刚上班，张大胖的邮箱里已经塞满了紧急邮件。

1 小梁的邮件

小梁的邮件里说了一个 RPC 调用的问题，本来公司的架构组开发了一个 RPC 框架让各个组去使用，但是各开发小组纷纷抱怨：这个 RPC 框架不支持动态的服务注册和发现。



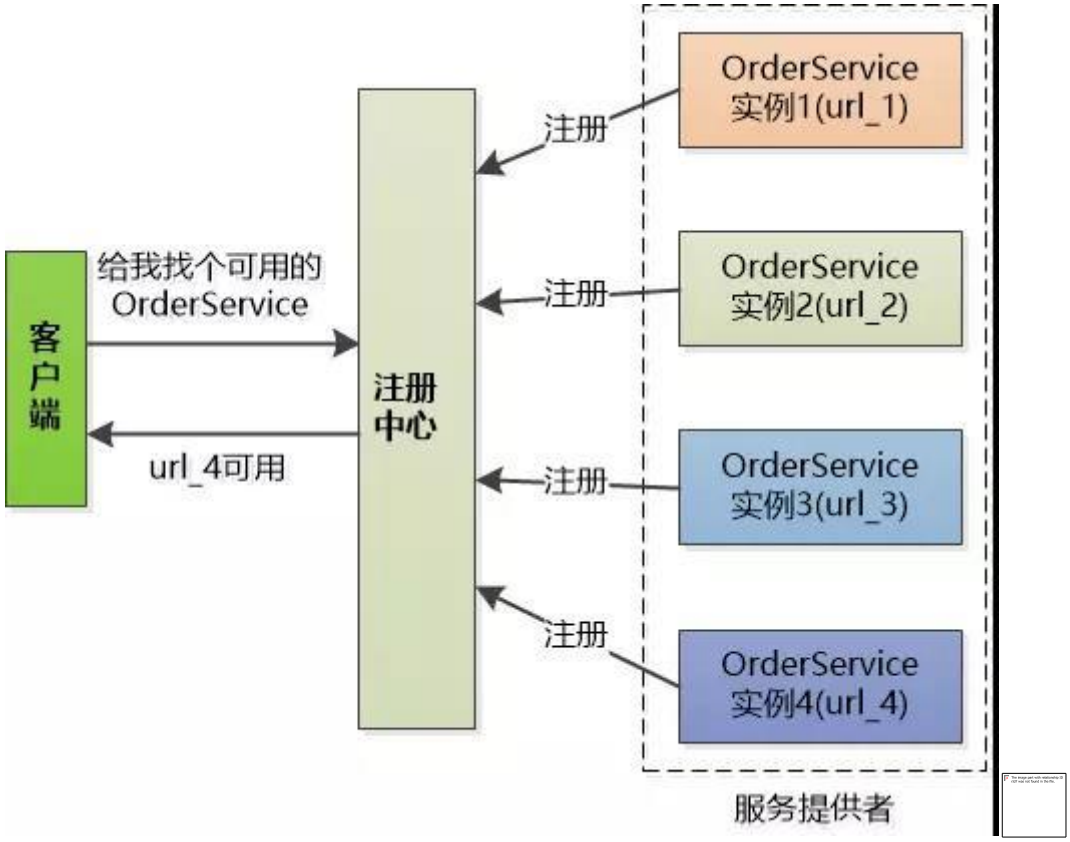
张大胖一看这个图就明白怎么回事了，为了支持高并发，*OrderService* 被部署了 *4* 份，每个客户端都保存了一份服务提供者的列表，但是这个列表是静态的（在配置文件中写死的），如果服务的提供者发生了变化，例如有些机器 *down* 了，或者又新增了 *OrderService* 的实例，客户端根本不知道，可能还在傻乎乎地尝试那些已经坏掉的实例呢！

想要得到最新的服务提供者的 *URL* 列表，必须得手工更新配置文件才行，确实很不方便。

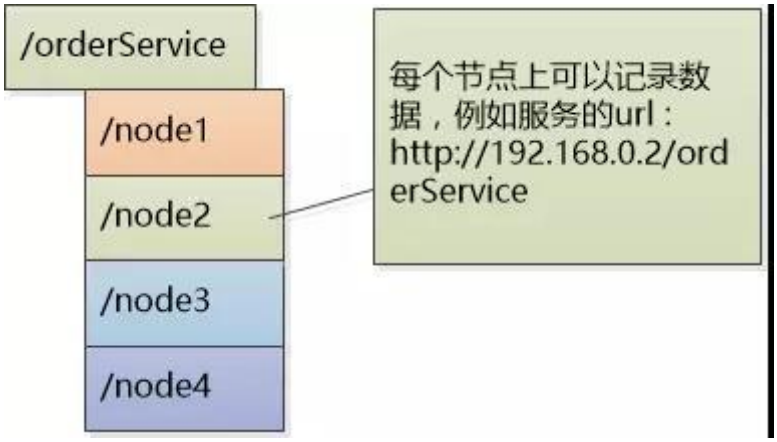
对于这样的问题，大胖马上就意识到，这就是客户端和服务提供者的紧耦合啊。

想解除这个耦合，非得增加一个中间层不可！

张大胖想到，应该有个注册中心，首先给这些服务命名（例如 *orderService*），其次那些 *OrderService* 都可以在这里注册一下，客户端就到这里来查询，只需要给出名称 *orderService*，注册中心就可以给出一个可以使用的 *url*， 再也不怕服务提供者的动态增减了。

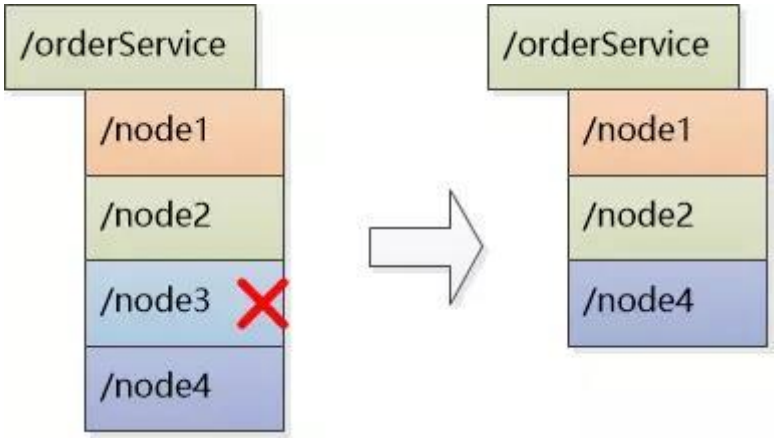


不知道是不是下意识的行为，**张大胖把这个注册中心的数据结构设计成为了一个树形结构**：



/orderService 表达了一个服务的概念， 下面的每个节点表示了一个服务的实例。例如/orderService/node2 表示的 order service 的第二个实例， 每个节点上可以记录下该实例的 url ，这样就可以查询了。

当然这个注册中心必须得能和各个服务实例通信，如果某个服务实例不幸 down 掉了， 那它在树结构中对于的节点也必须删除， 这样客户端就查询不到了。

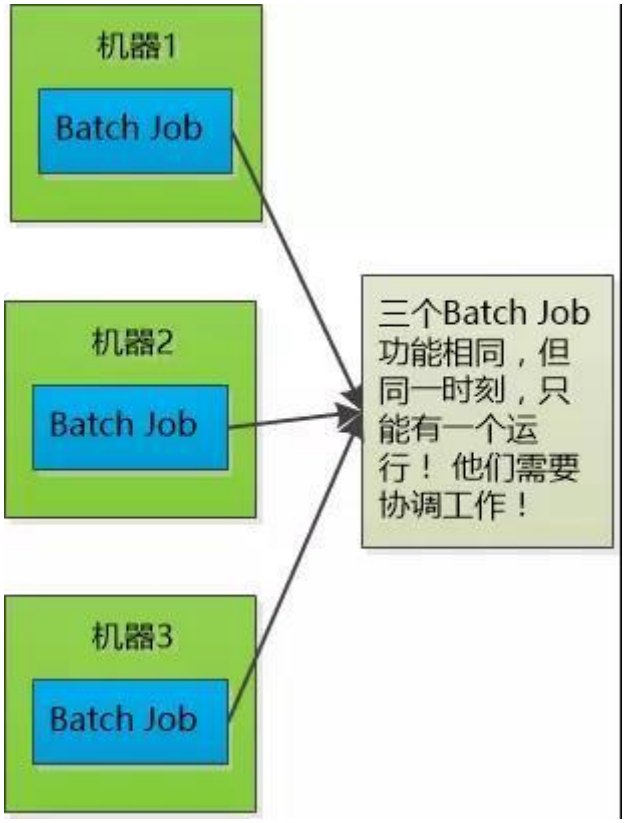


嗯，可以在注册中心和各个服务实例直接建立 Session，让各个服务实例定期地发送心跳，如果过了特定时间收不到心跳，就认为这个服务实例挂掉了，Session 过期， 把它从树形结构中删除。

张大胖把自己的想法回复了小梁， 接着看小王的邮件。

2 小王的 Master 选举

小王邮件中说的是三个 Batch Job 的协调问题，这三个 Batch Job 部署在三台机器上，但是这三个 Batch Job 同一个时刻只能有一个运行，如果其中某个不幸 down 掉，剩下的两个就需要做个选举，选出来的那个 Batch Job 需要“继承遗志”，继续工作。



其实这就是一个 Master 的选举问题，张大胖一眼就看出了本质。

只是为了选举出 Master，这三个 Batch Job 需要互通有无，互相协调才行，这就麻烦了！

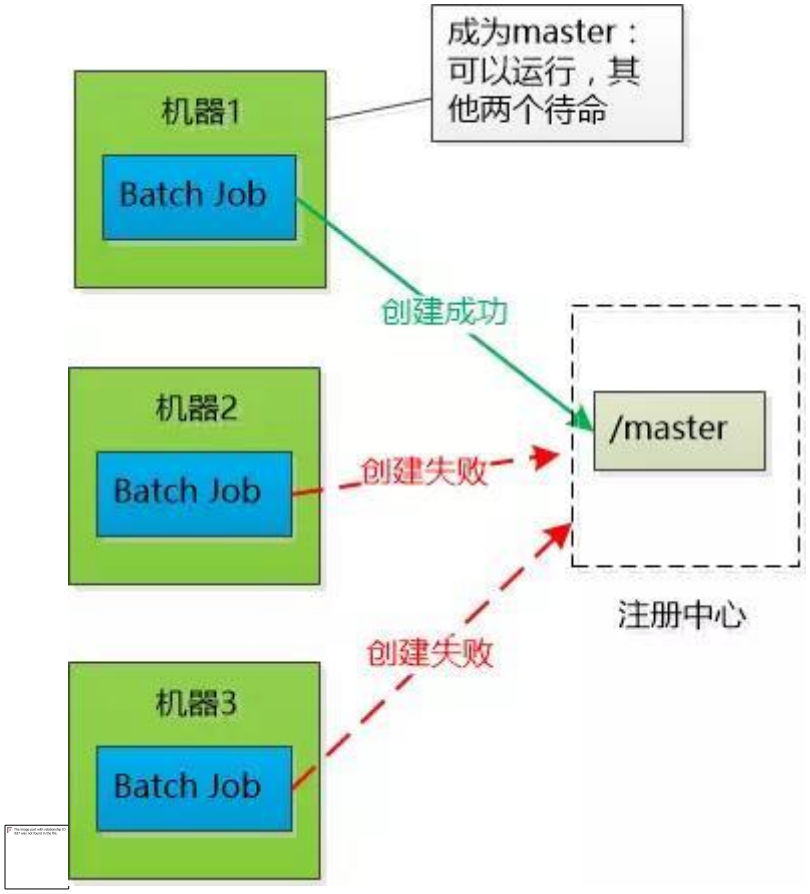
要不弄个数据库表？利用数据库表主键不能冲突的特性，让这三个 Batch Job 都向同一个表中插入同样的数据，谁先成功谁就是 Master ！

可是如果抢到 Master 的那个 Batch Job 挂掉了，别人永远就抢不到了！ 因为记录已经存在了， 别的 Batch Job 没法插入数据了！

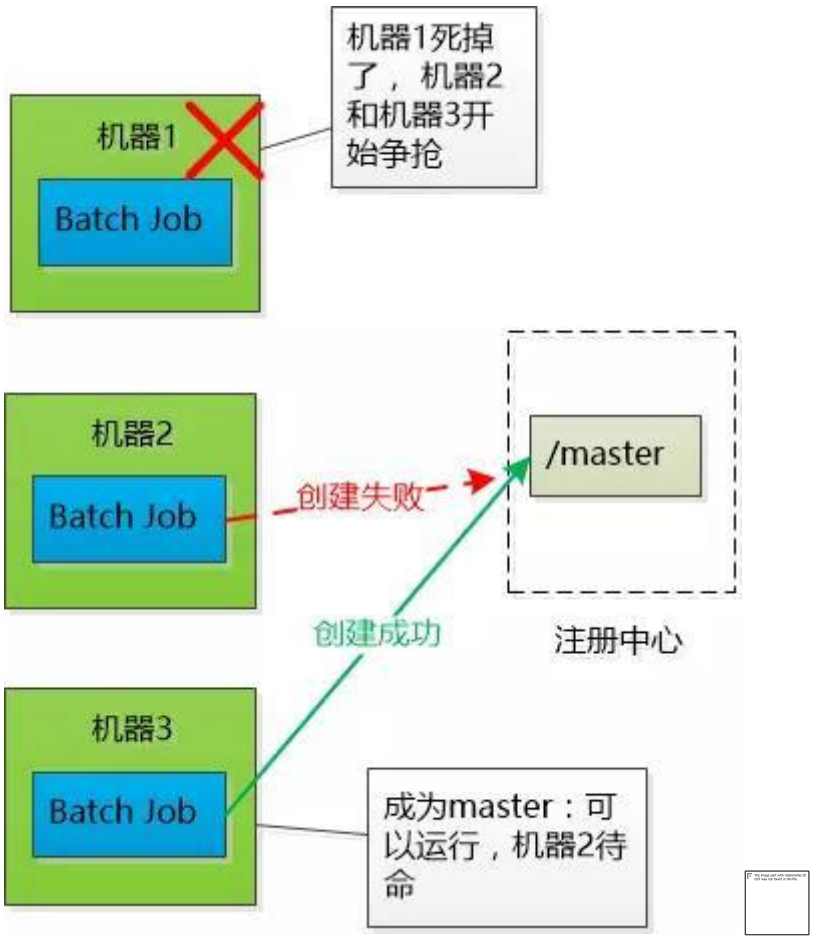
嗯, 还得加上定期更新的机制, 如果一段时间内没有更新就认为 Master 死掉了, 别的 Batch Job 可以继续抢..... 不过这么做好麻烦！

换个思路，让他们也去一个注册中心去大吼一声："我是 master!"， 谁的声音大谁是 Master 。

其实不是吼一声，三个 Batch Job 启动以后，都去注册中心争抢着去创建一个树的节点（例如/master ），谁创建成功谁就是 Master （**当然注册中心必须保证只能创建成功一次，其他请求就失败了**），其他两个 Batch Job 就对这个节点虎视眈眈地监控，如果这个节点被删除，就开始新一轮争抢，去创建那个 /master 节点。



什么时候节点会被删除呢？ 对，就是当前 Master 的机器 down 掉了 ！ 很明显，注册中心也需要和各个机器通信，看看他们是否活着。



等等，这里还有一个复杂的情况， **如果机器 1 并没有死掉，只是和注册中心长时间连接不上**，注册中心会发现 Session 超时，会把机器 1 创建的/master 删除。 让机器 2 和机器 3 去抢，如果机器 3 成为了 master，开始运行 Batch Job， 但是机器 1 并不知道自己被解除了 Master 的职务， 还在努力的运行 Batch Job，这就冲突了！

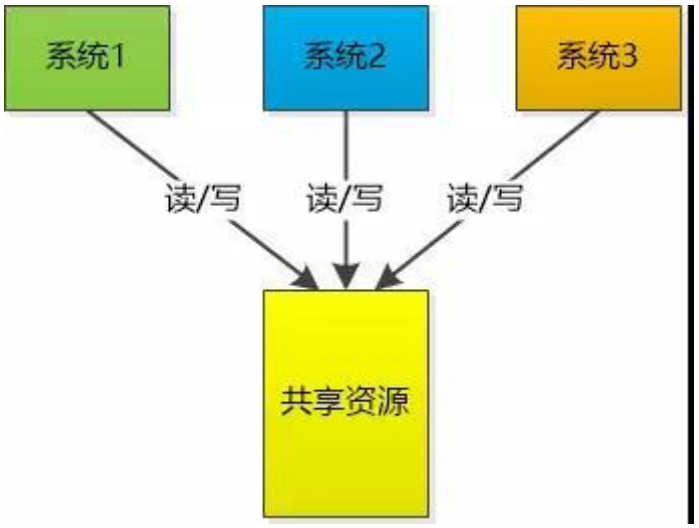
看来机器 1 必须得能感知到和注册中心的连接断开了，需要停止 Batch Job 才行，等到和注册中心再次连接上以后，才知道自己已经不是 master 了，老老实实地等下一次机会吧。

无论哪种方案，实现起来都很麻烦，这该死的分布式！

先把思路给小王回复一下吧。接着看小蔡的邮件。

3 小蔡的分布式锁

小蔡的邮件里说的问题更加麻烦，有多个不同的系统（当然是分布在不同的机器上！），要对同一个资源操作。



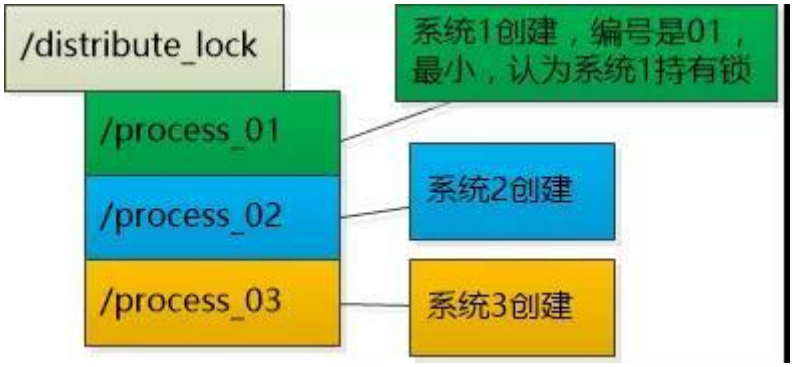
这要是在一个机器上，使用某个语言内置的锁就可以搞定，例如 Java 的 synchronized ， 但是现在是分布式啊，程序都跑在不同机器的不同进程中，synchcronized 一点用都没有了！

这是个分布式锁的问题啊！

能不能考虑下 Master 选举问题中的方式，让大家去抢？ 谁能抢先在注册中心创建一个/distribute_lock 的节点就表示抢到这个锁了，然后读写资源，读写完以后就把/distribute_lock 节点删除，大家再来抢。

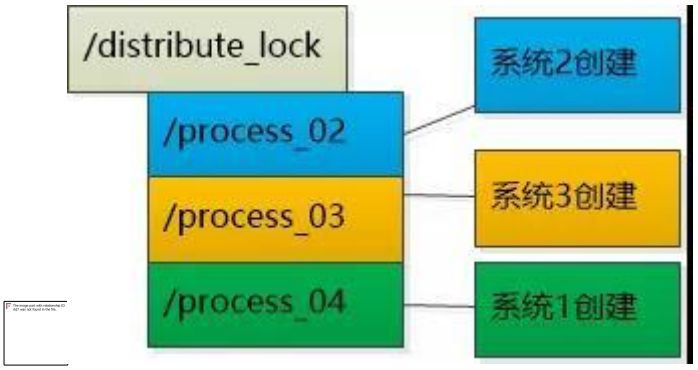
可是这样的话某个系统可能会多次抢到，不太公平。

如果让这些系统在注册中心的/distribute_lock 下都创建子节点， 然后给每个系统一个编号，会是这个样子：

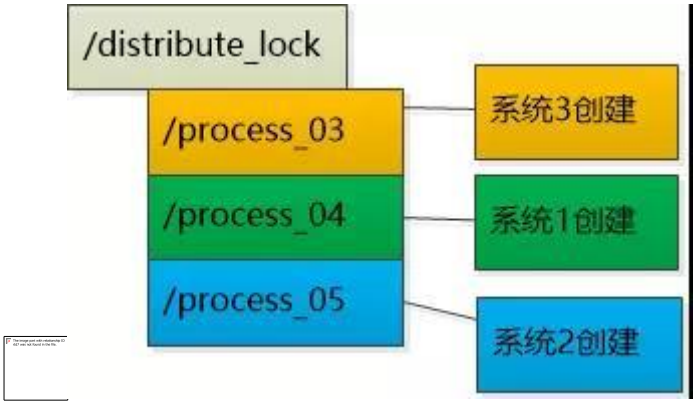


然后各个系统去检查自己的编号，谁的编号小就认为谁持有了锁， 例如系统 1。

系统 1 持有了锁，就可以对共享资源进行操作了， 操作完成以后 process_01 这个节点删除， 再创建一个新的节点（编号变成 process_04 了）：



其他系统一看，编号为 01 的删除了，再看看谁是最小的吧，是 process_02，那就认为系统 2 持有了锁，可以对共享资源操作了。 操作完成以后也要把 process_02 节点删除，创建新的节点。这时候 process_03 就是最小的了，可以持有锁了。



这样循环往复下去..... 分布式锁就可以实现了！

看看，我设计的这个集中式的树形结构很不错吧，能解决各种各样的问题！张大胖不由得意起来。

好，先把这个想法告诉小蔡，实现细节下午开个会讨论。

4、Zookeeper

正准备回复小蔡的时候，大胖突然意识到，自己漏了一个重要的点，那就是**注册中心的高可用性**，如果注册中心只有那么一台机器，一旦挂掉，整个系统就玩完了。

这个注册中心也得有多台机器来保证高可用性，那个自己颇为得意的树形结构也需要在多个机器之间同步啊，要是有机器挂掉怎么办？通信超时怎么办？树形结构的数据怎么在各个机器之间保证强一致性？

小王、小梁、小蔡的原始问题没有解决，单单是这个注册中心就要了命了。 以自己公司的技术实力，搞出一套这样的注册中心简直是 *Mission Impossible！*

大胖赶紧上网搜索，看看有没有类似的解决方案，让大胖感到万分幸运的是，果然有一个，叫做 **Zookeeper** ！

Zookeeper 所使用的树形结构和自己想象的非常类似，更重要的是，人家实现了树形结构数据在多台机器之间的可靠复制，达到了数据在多台机器之间的一致性。并且这多台机器中如果有部分挂掉了/或者由于网络原因无法连接上了， 整个系统还可以工作。

大胖赶快去看 Zookeeper 的关键概念和 API：

1. **Session** ： 表示某个客户系统（例如 Batch Job）和 ZooKeeper 之间的连接会话， Batch Job 连上 ZooKeeper 以后会周期性地发送心跳信息， 如果 Zookeepr 在特定时间内收不到心跳，就会认为这个 Batch Job 已经死掉了，Session 就会结束。

2. **znode** ： 树形结构中的每个节点叫做 znode， 按类型可以分为**永久的 znode**（除非主动删除，否则一直存在），**临时的 znode**（Session 结束就会删除）和 **顺序 znode**（就是小蔡的分布式锁中的 process_01,process_02.....）。

3. **Watch** ： 某个客户系统（例如 Batch Job）可以监控 znode， znode 节点的变化（删除，修改数据等）都可以通知 Batch Job， 这样 Batch Job 可以采取相应的动作，例如争抢着去创建节点。

嗯，这些概念和接口应该可以满足我们的要求了， 就是它了，下午召集大家开会开始学习 Zookeeper 吧。