## 场景一：服务发现（Service Discovery）

服务发现要解决的也是分布式系统中最常见的问题之一，即在同一个分布式集群中的进程或服务，要如何才能找到对方并建立连接。本质上来说，服务发现就是想要了解集群中是否有进程在监听udp或tcp端口，并且通过名字就可以查找和连接。要解决服务发现的问题，需要有下面三大支柱，缺一不可。

1. **一个强一致性、高可用的服务存储目录。**基于Raft算法的etcd天生就是这样一个强一致性高可用的服务存储目录。
2. **一种注册服务和监控服务健康状态的机制。**用户可以在etcd中注册服务，并且对注册的服务设置key TTL，定时保持服务的心跳以达到监控健康状态的效果。
3. **一种查找和连接服务的机制。**通过在etcd指定的主题下注册的服务也能在对应的主题下查找到。为了确保连接，我们可以在每个服务机器上都部署一个Proxy模式的etcd，这样就可以确保能访问etcd集群的服务都能互相连接。

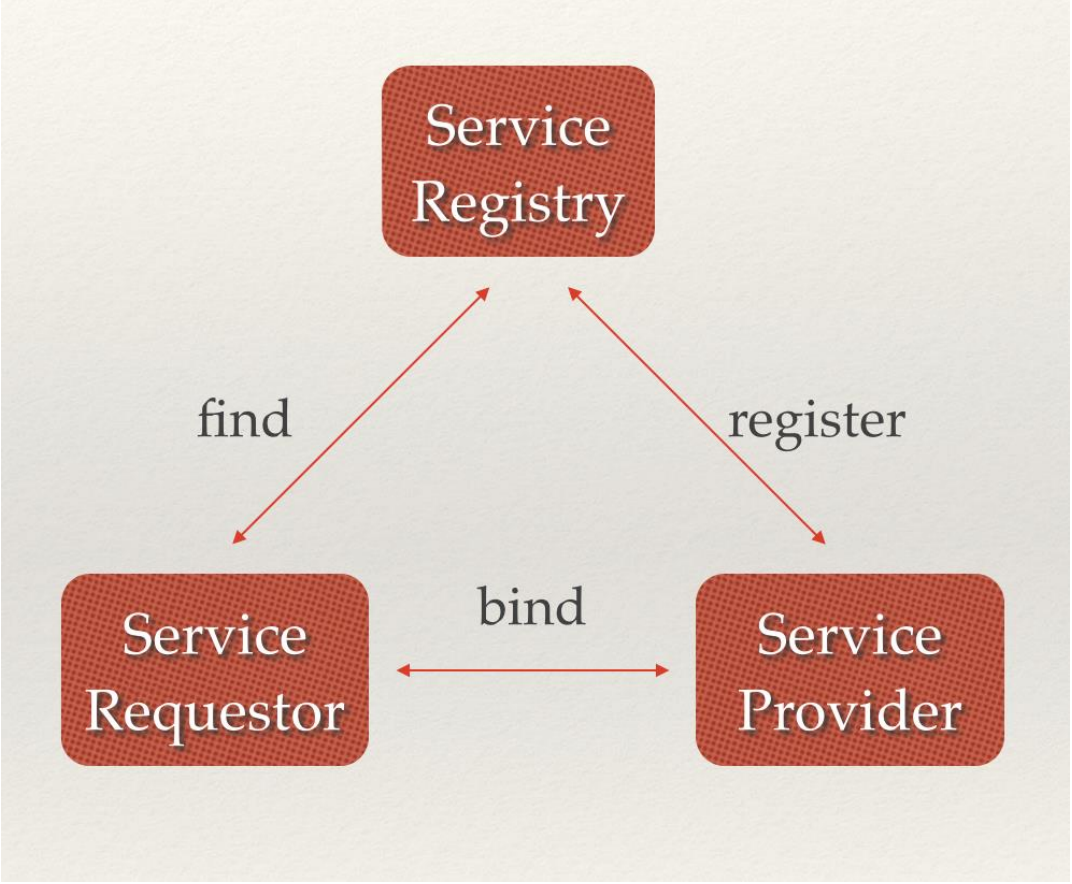


**图1 服务发现示意图**

下面我们来看服务发现对应的具体场景。

- **微服务协同工作架构中，服务动态添加。**随着Docker容器的流行，多种微服务共同协作，构成一个相对功能强大的架构的案例越来越多。透明化的动态添加这些服务的需求也日益强烈。通过服务发现机制，在etcd中注册某个服务名字的目录，在该目录下存储可用的服务节点的IP。在使用服务的过程中，只要从服务目录下查找可用的服务节点去使用即可。

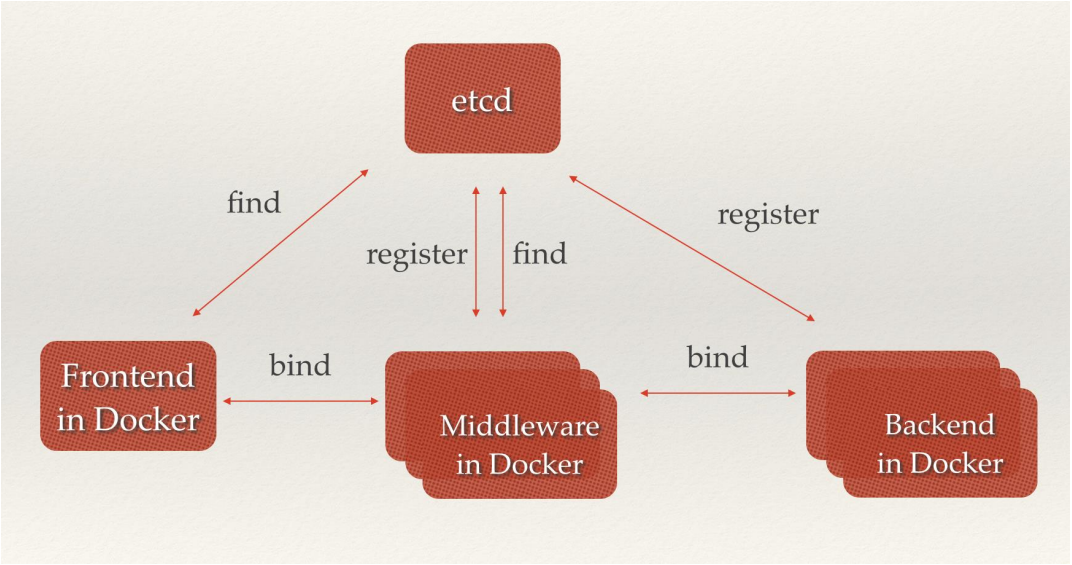


**图2 微服务协同工作**

- **PaaS平台中应用多实例与实例故障重启透明化。**PaaS平台中的应用一般都有多个实例，通过域名，不仅可以透明的对这多个实例进行访问，而且还可以做到负载均衡。但是应用的某个实例随时都有可能故障重启，这时就需要动态的配置域名解析（路由）中的信息。通过etcd的服务发现功能就可以轻松解决这个动态配置的问题。

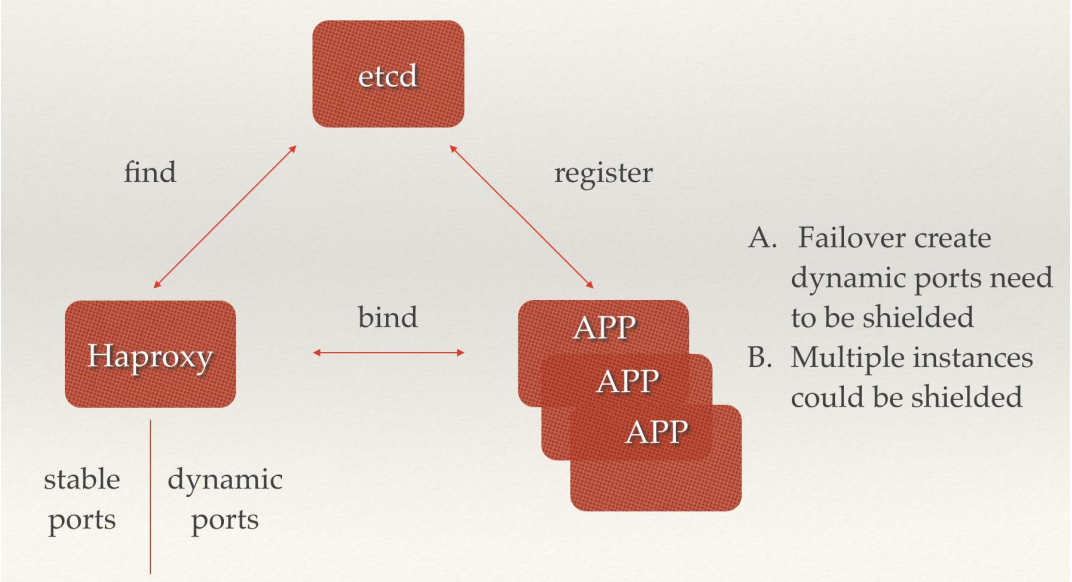


**图3 云平台多实例透明化**

## 场景二：消息发布与订阅

在分布式系统中，最适用的一种组件间通信方式就是消息发布与订阅。即构建一个配置共享中心，数据提供者在这个配置中心发布消息，而消息使用者则订阅他们关心的主题，一旦主题有消息发布，就会实时通知订阅者。通过这种方式可以做到分布式系统配置的集中式管理与动态更新。

- **应用中用到的一些配置信息放到etcd上进行集中管理。**这类场景的使用方式通常是这样：应用在启动的时候主动从etcd获取一次配置信息，同时，在etcd节点上注册一个Watcher并等待，以后每次配置有更新的时候，etcd都会实时通知订阅者，以此达到获取最新配置信息的目的。

- 分布式搜索服务中，**索引的元信息和服务器集群机器的节点状态存放在etcd中**，供各个客户端订阅使用。使用etcd的key TTL功能可以确保机器状态是实时更新的。
- **分布式日志收集系统**。这个系统的核心工作是收集分布在不同机器的日志。收集器通常是按照应用（或主题）来分配收集任务单元，因此可以在etcd上创建一个以应用（主题）命名的目录P，并将这个应用（主题相关）的所有机器ip，以子目录的形式存储到目录P上，然后设置一个etcd递归的Watcher，递归式的监控应用（主题）目录下所有信息的变动。这样就实现了机器IP（消息）变动的时候，能够实时通知到收集器调整任务分配。
- **系统中信息需要动态自动获取与人工干预修改信息请求内容的情况。**通常是暴露出接口，例如JMX接口，来获取一些运行时的信息。引入etcd之后，就不用自己实现一套方案了，只要将这些信息存放到指定的etcd目录中即可，etcd的这些目录就可以通过HTTP的接口在外部访问。

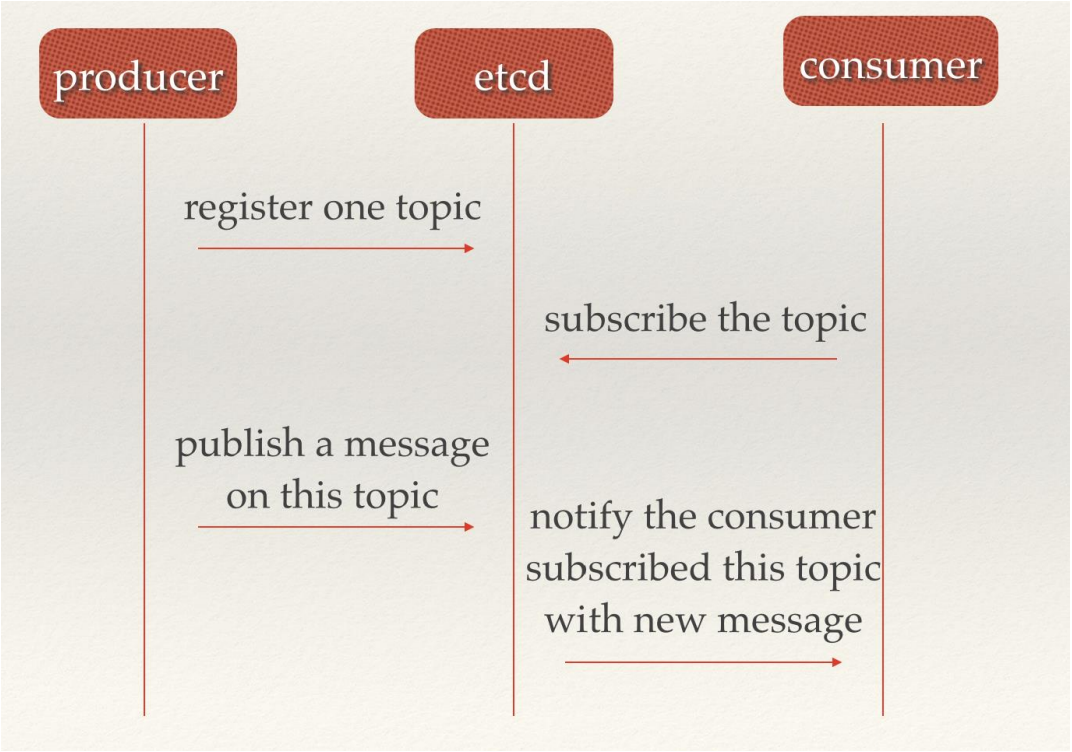


**图4 消息发布与订阅**

## 场景三：负载均衡

在场景一中也提到了负载均衡，本文所指的负载均衡均为软负载均衡。分布式系统中，为了保证服务的高可用以及数据的一致性，通常都会把数据和服务部署多份，以此达到对等服务，即使其中的某一个服务失效了，也不影响使用。由此带来的坏处是数据写入性能下降，而好处则是数据访问时的负载均衡。因为每个对等服务节点上都存有完整的数据，所以用户的访问流量就可以分流到不同的机器上。

- **etcd本身分布式架构存储的信息访问支持负载均衡**。etcd集群化以后，每个etcd的核心节点都可以处理用户的请求。所以，把数据量小但是访问频繁的消息数据直接存储到etcd中也是个不错的选择，如业务系统中常用的二级代码表（在表中存储代码，在etcd中存储代码所代表的具体含义，业务系统调用查表的过程，就需要查找表中代码的含义）。
- **利用etcd维护一个负载均衡节点表**。etcd可以监控一个集群中多个节点的状态，当有一个请求发过来后，可以轮询式的把请求转发给存活着的多个状态。类似KafkaMQ，通过ZooKeeper来维护生产者和消费者的负载均衡。同样也可以用etcd来做ZooKeeper的工作。

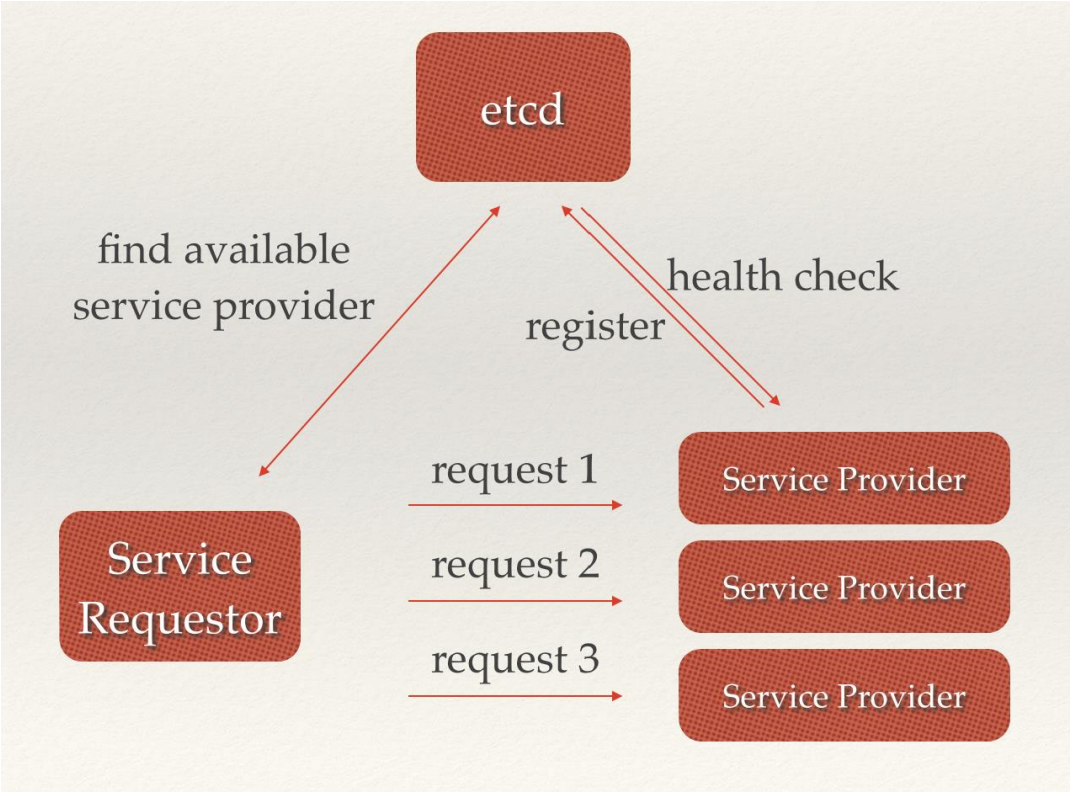


**图5 负载均衡**

## 场景四：分布式通知与协调

这里说到的分布式通知与协调，与消息发布和订阅有些相似。都用到了etcd中的Watcher机制，通过注册与异步通知机制，实现分布式环境下不同系统之间的通知与协调，从而对数据变更做到实时处理。实现方式通常是这样：不同系统都在etcd上对同一个目录进行注册，同时设置Watcher观测该目录的变化（如果对子目录的变化也有需要，可以设置递归模式），当某个系统更新了etcd的目录，那么设置了Watcher的系统就会收到通知，并作出相应处理。

- **通过etcd进行低耦合的心跳检测**。检测系统和被检测系统通过etcd上某个目录关联而非直接关联起来，这样可以大大减少系统的耦合性。
- **通过etcd完成系统调度**。某系统有控制台和推送系统两部分组成，控制台的职责是控制推送系统进行相应的推送工作。管理人员在控制台作的一些操作，实际上是修改了etcd上某些目录节点的状态，而etcd就把这些变化通知给注册了Watcher的推送系统客户端，推送系统再作出相应的推送任务。
- **通过etcd完成工作汇报**。大部分类似的任务分发系统，子任务启动后，到etcd来注册一个临时工作目录，并且定时将自己的进度进行汇报（将进度写入到这个临时目录），这样任务管理者就能够实时知道任务进度。

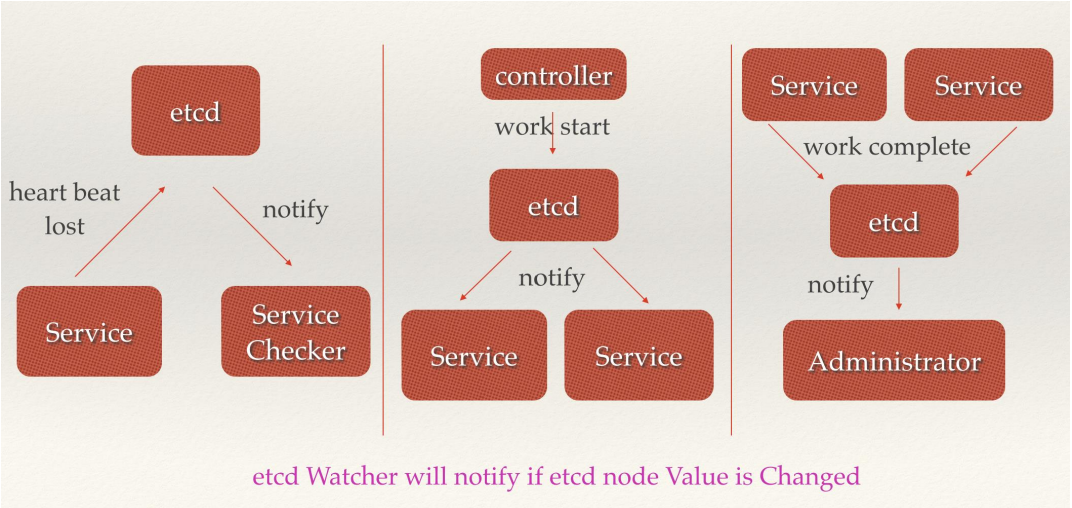


**图6 分布式协同工作**

## 场景五：分布式锁

因为etcd使用Raft算法保持了数据的强一致性，某次操作存储到集群中的值必然是全局一致的，所以很容易实现分布式锁。锁服务有两种使用方式，一是保持独占，二是控制时序。

- **保持独占即所有获取锁的用户最终只有一个可以得到。**etcd为此提供了一套实现分布式锁原子操作CAS（CompareAndSwap）的API。通过设置prevExist值，可以保证在多个节点同时去创建某个目录时，只有一个成功。而创建成功的用户就可以认为是获得了锁。
- 控制时序，即所有想要获得锁的用户都会被安排执行，但是**获得锁的顺序也是全局唯一的，同时决定了执行顺序。**etcd为此也提供了一套API（自动创建有序键），对一个目录建值时指定为POST动作，这样etcd会自动在目录下生成一个当前最大的值为键，存储这个新的值（客户端编号）。同时还可以使用API按顺序列出所有当前目录下的键值。此时这些键的值就是客户端的时序，而这些键中存储的值可以是代表客户端的编号。

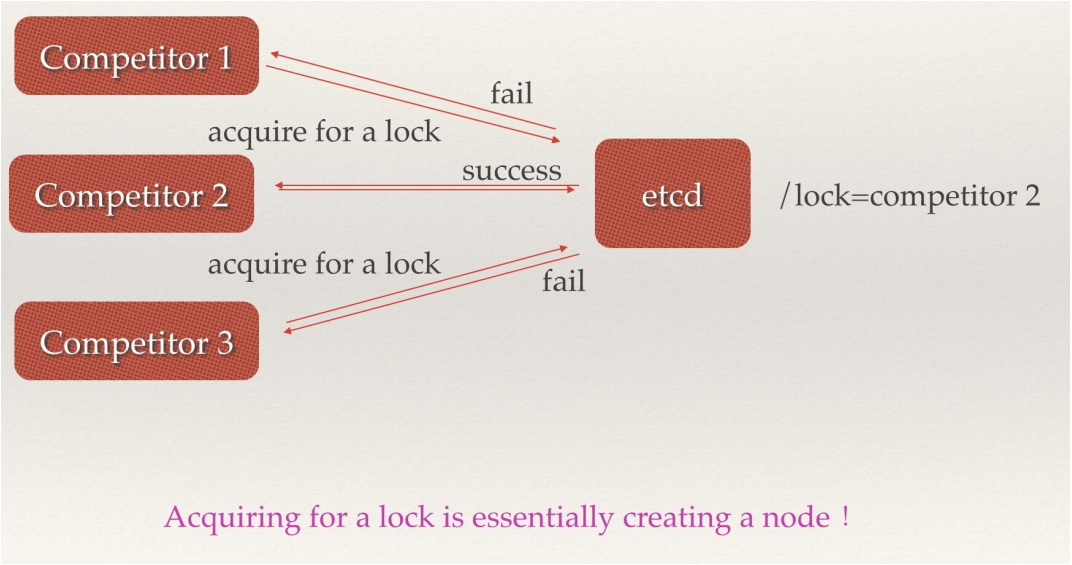


**图7 分布式锁**

## 场景六：分布式队列

分布式队列的常规用法与场景五中所描述的分布式锁的控制时序用法类似，即创建一个先进先出的队列，保证顺序。

另一种比较有意思的实现是**在保证队列达到某个条件时再统一按顺序执行**。这种方法的实现可以在/queue这个目录中另外建立一个/queue/condition节点。

- condition可以**表示队列大小**。比如一个大的任务需要很多小任务就绪的情况下才能执行，每次有一个小任务就绪，就给这个condition数字加1，直到达到大任务规定的数字，再开始执行队列里的一系列小任务，最终执行大任务。
- condition可以**表示某个任务在不在队列**。这个任务可以是所有排序任务的首个执行程序，也可以是拓扑结构中没有依赖的点。通常，必须执行这些任务后才能执行队列中的其他任务。
- condition还可以**表示其它的一类开始执行任务的通知**。可以由控制程序指定，当condition出现变化时，开始执行队列任务。

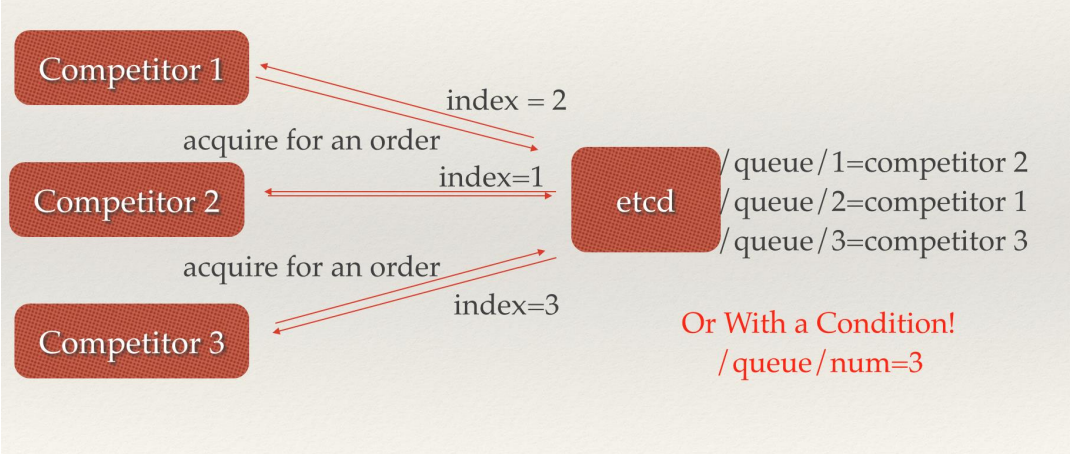


**图8 分布式队列**

## 场景七：集群监控与Leader竞选

通过etcd来进行监控实现起来非常简单并且实时性强。

1. 前面几个场景已经提到Watcher机制，当某个节点消失或有变动时，Watcher会第一时间发现并告知用户。
2. 节点可以设置TTL key，比如每隔30s发送一次心跳使代表该机器存活的节点继续存在，否则节点消失。

这样就可以第一时间检测到各节点的健康状态，以完成集群的监控要求。

另外，使用分布式锁，可以完成Leader竞选。这种场景通常是一些长时间CPU计算或者使用IO操作的机器，只需要竞选出的Leader计算或处理一次，就可以把结果复制给其他的Follower。从而避免重复劳动，节省计算资源。

这个的经典场景是**搜索系统中建立全量索引**。如果每个机器都进行一遍索引的建立，不但耗时而且建立索引的一致性不能保证。通过在etcd的CAS机制同时创建一个节点，创建成功的机器作为Leader，进行索引计算，然后把计算结果分发到其它节点。

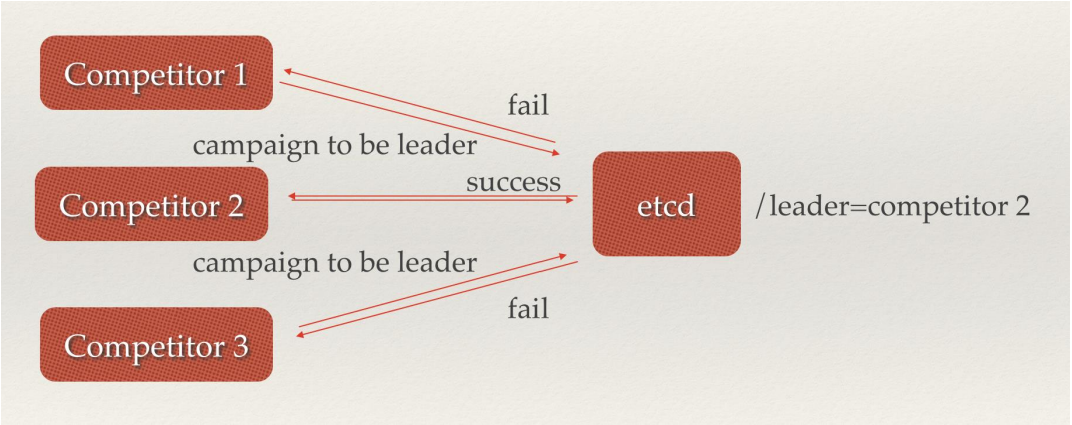
Competitor 1
fail
campaign to be leader
Competitor 2
success
etcd
/leader=competitor 2
campaign to be leader
fail
Competitor 3

**图9 Leader竞选**