分布式锁

分布式锁：“分布式锁是控制分布式系统之间同步访问共享资源的一种方式。在分布式系统中，常常需要协调他们的动作。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此干扰来保证一致性，在这种情况下，便需要使用到分布式锁。”

* 基于数据库的分布式锁：

1. 基于表主键唯一：加锁时向数据库插入以锁名为主键的数据，解锁时删除该数据。

优化：  
1.没有过期时间，如果获得锁的服务解锁失败，将导致其他服务无法获得锁。网上也有解决方法：增加过期时间字段，定期清除过期锁数据。  
2.重入性，增加获得锁服务的标记字段。

1. 基于版本号字段：为表结构增加版本号字段，每次获取数据将版本号获取到，下次修改数据时判断当前版本是否是该版本号。
2. 排他锁：

* 基于Redis实现分布式锁：

1. 加锁：key是锁的唯一标识，setnx（key，1）当一个线程setnx返回1时，说明key原本不存在，该线程成功得到了锁；当一个线程执行setnx’返回0，说明key已经存在，该线程抢锁失败
2. 解锁：得到锁的线程执行完任务，需要释放锁，以便其他线程可以进入。释放锁的最简单方式是执行del指令，
3. 锁超时：如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放，这块资源将会永远被锁住，别的线程无法访问。

致命问题：

1. setnx和expire的非原子性：线程1得到锁，还没来得及expire，挂了，锁一直在，没法访问。、

解决：新版本Redis，使用set增加了可选参数，set（key，1，30，NX）。

1. del误删：线程2得到了锁，设置的超时时间是30秒，某些原因线程过了30秒没执行完，锁过期自动释放，线程3得到了锁，随后，线程2结束。执行del命令来释放锁，，但这时候锁在3上，线程3还没执行完，所以线程2释放的其实是线程3的锁。

解决：只能在释放锁之前判断当前的锁是不是自己加的锁。加锁的时候以线程ID作为value，删除时验证key对应的value是不是自己的线程ID。这样做隐含一个问题就是：判断和释放锁是两个独立操作，不是原子性。

所以这一块可以使用Lua（？？？？？）脚本实现，这样就是原子操作了

1. 出现并发的可能性：基于第2描述的情况，线程2没干完，锁没了。我们可以让获得锁的线程开启一个守护线程，给快要过期的锁续航。

过了29秒没执行完守护线程就会执行expire指令，为这把锁续命20秒，此后每20秒执行一次。当线程2执行完任务，就会显式关掉守护线程。

另外，如果节点一忽然断电，由于线程2和守护线程在同一个进程，守护线程也会停下。这把锁到了超时的时候，不续航就会自动释放。

 Redisson 框架实现Redis分布式锁。官方推荐。

* 基于ZooKeeper的分布式锁：

有序性是ZooKeeper中一个非常重要的特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid，而读请求只会相对于更新有序，也就是读请求返回结果中会带有这个ZooKeeper最新的zxid。

ZOOkeeper节点的性质：

1. 有序节点：父节点创建子节点，指明有序，生成子节点时会根据当前子节点的数量自动添加整数序号。
2. 临时节点：客户端可以建立一个临时节点，在会话结束后或者会话超时后，ZooKeeper会自动删除该节点。
3. 事件监听：在读取数据时，我们可以同时对节点设置事件监听，当节点数据或结构变化时，ZoOKeeper会通知客户端。

分布式锁算法流程如下：

• 客户端连接zookeeper，并在/lock下创建临时的且有序的子节点，第一个客户端对应的子节点为/lock/lock-0000000000，第二个为/lock/lock-0000000001，以此类推；

• 客户端获取/lock下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听刚好在自己之前一位的子节点删除消息，获得子节点变更通知后重复此步骤直至获得锁；

• 执行业务代码；

• 完成业务流程后，删除对应的子节点释放锁。

ZooKeeper的应用：

1、分布式锁

2、服务注册和发现：阿里的分布式RPC框架和Dubbo

3、共享配置和状态信息

。。。。。。。

 Curator 这个开源框架实现了ZooKeeper分布式锁。

微服务架构

微服务架构，旨在通过将功能分解到各个离散的服务中以实现对解决方案的解耦。

概念：把一个大型的单个应用程序和服务拆分为数个甚至数十个的支持微服务，它可扩展单个组件而不是整个的应用程序堆栈，从而满足服务等级协议。

定义：围绕业务领域组件来创建应用，这些应用可独立地进行开发、管理和迭代。在分散的组件中使用云架构和平台式部署、管理和服务功能，使产品交付变得更加简单。

本质：用一些功能比较明确、业务比较精练的服务去解决更大、更实际的问题。

拆分后的服务可以采用异构的技术。比如网站的数据分析服务可以使用数据仓库作为持久化层，以便于高效的做一些统计计算；商品服务和促销服务访问频率比较大，因此加入了缓存机制等。对于数据库拆分有一些问题：比如说跨库级联的需求，通过服务查询数据颗粒度的粗细问题等。但是这些问题可以通过合理的设计来解决。总体来说，数据库拆分是一个利大于弊的。

以往单体应用，排查问题通常是看一下日志，研究错误信息和调用堆栈。而微服务架构整个应用分散成多个服务，定位故障点非常困难，并且一个服务故障可能会产生雪崩效应，导致整个系统故障。服务之间彼此调用，该怎么？？？微服务好像是创建maven工程，在服务中引入另一个服务，即可使用另一个服务的API。

微服务架构虽然解决了旧问题，也引入了新的问题:

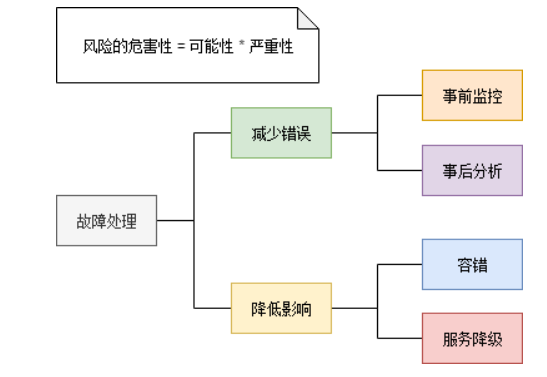
1、 微服务架构整个应用分散成多个服务，定位故障点非常困难。、

2、 稳定性下降。服务数量变多导致其中一个服务出现故障的概率增大，并且一个服务故障可能导致整个系统挂掉。事实上，在大访问量的生产场景下，故障总是会出现的。

3、 服务数量非常多，部署、管理的工作量很大。

4、 开发方面：如何保证各个服务在持续开发的情况下仍然保持协同合作。

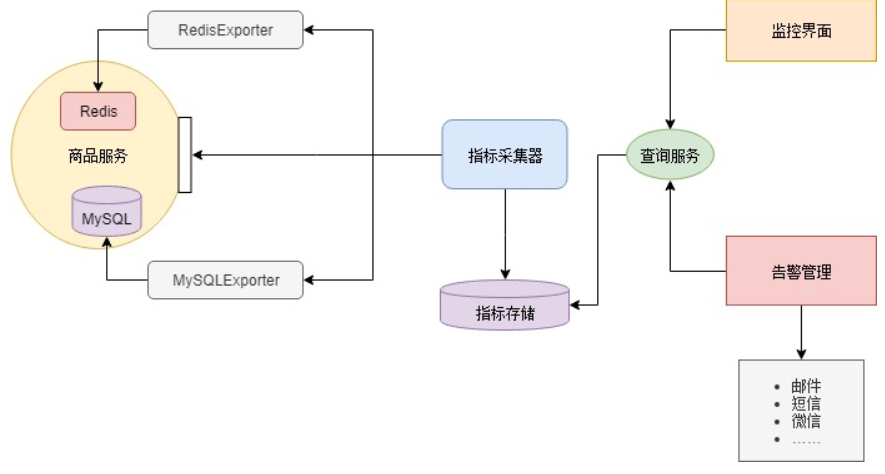
5、 测试方面：服务拆分后，几乎所有功能都会涉及多个服务。原本单个程序的测试变为服务间调用的测试。测试变得更加复杂。



**减少错误方面：**

* 监控——发现故障的征兆

微服务监控系统：



* 定位问题——链路跟踪

在微服务架构下，一个用户的请求往往涉及多个内部服务调用。为了方便定位问题，需要能够记录每个用户请求时，微服务内部产生了多少服务调用，及其调用关系。这个叫做链路跟踪。

链路跟踪只能定位到哪个服务出现问题，不能提供具体的错误信息。查找具体的错误信息的能力则需要由日志分析组件来提供。

* 分析问题——日志分析

**降低故障影响方面：**

* 服务注册于发现——动态扩容
* 熔断、服务降级和限流：