# 背景

为了保证其他模块对自己没有影响，那么最好是与其他模块没有任何关系，即去依赖。如果需要依赖，则弱依赖。

弱依赖有需要被依赖方的返回结果和不依赖返回结果两种。需要结果就要请求后回调，不需要就直接异步化。另外要做好超时和重试、蓄洪、限流、熔断、降级。

如果只能强依赖，人家死了，那就我们报错，但是我们不死。这也需要设置合理超时和重试、蓄洪、限流、熔断、降级。人家又复活了，我们也要立即恢复。

单机系统无法满足需求，采用集群，单机房存在问题，采用多机房，整个地区断网，采用多地区，即异地多活。

涉及集群和跨区问题，就需要考虑策略问题：负载均衡，主从切换，优先策略。

**如何保障系统的高可用？**

我们都知道，单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量在系统设计的过程中避免单点。方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了服务会受影响；如果有冗余备份，挂了还有其他backup能够顶上。

保证系统高可用，架构设计的核心准则是：冗余。

有了冗余之后，还不够，每次出现故障需要人工介入恢复势必会增加系统的不可服务实践。所以，又往往是通过“自动故障转移”来实现系统的高可用。

# 概述

主要涉及故障检测、故障隔离、故障备份和故障恢复。

## 集群

LB（负载均衡）：LVS/Nginx（http/upstream，stream/upstream）

HA（高可用）：SPoF（Single Point Of Failure）

高可用集群（High Availability Cluster，简称HA Cluster），是指以减少服务中断时间为目的的服务器集群技术。它通过保护用户的业务程序对外不间断提供的服务，把因软件、硬件、人为造成的故障对业务的影响降低到最小程度。

HPC

所谓高可用集群，即当前服务器出现故障时，可以将该服务器中的服务、资源、IP等转移到另外一台服务器上，从而满足业务的持续性；这两台或多台服务器构成了服务器高可用集群。

对于客户端来说，集群就像是一台服务器，因为集群运行的是同一种服务，即使其中有的服务器宕机或无法通信时，也不会对业务造成影响。

## 特征

首先高可用架构应该具备如下特征:

数据库对前端业务透明，业务不会因为数据库故障产生中断。

非主节点的数据应该和主节点的数据实时或者最终保持一致。

当业务因高可用机制发生数据库切换时，切换前后的数据库内容应当一致，不会因为数据缺失或者数据不一致而影响业务。

### 主从切换

高可用最主要的就是出现故障时，能够做到业务的无缝迁移，即主备切换。

当其中一台机器的服务宕机后，对于服务调用者来说，能够迅速的切换到其他可用服务，当服务升级为主服务，这种切换速度应当控制在秒级别（几秒钟）。

当宕机的服务恢复之后，自动变为从服务，主从服务角色切换。主从切换一定是要付出代价的，所以当主服务恢复之后，也就不再替换现有的主服务。

### 负载均衡

当服务的请求量比较高的时候，一台服务不能满足需求，这时候需要多台机器提供同样的服务，将所有请求分发到不同机器上。

高可用架构中应该具有丰富的负载均衡策略和易调节负载的方式。甚至可以自动化智能调节，例如由于机器性能的原因，响应时间可能不一样，这时候可以向性能差的机器少一点分发量，保证各个机器响应时间的均衡。

### 易横向拓展

当用户越来越多，已有服务不能承载更多的用户的时候，便需要对服务进行拓展，扩展的方式最好是不触动原有的服务，对于服务的调用者是透明的。

## 衡量指标

由系统可靠性（Availability）和可维护性（maintainabilit）来度量。

计算方式：HA=MTTF（平均无故障事件）/（MTTF+MTTR（平均修复事件））\*100%

99% 全年服务中断时间不超过4天

99.9% 全年服务中断时间不超过10个小时

99.99% 全年服务中断时间不超过1个小时

99.999% 全年服务中断时间不超过6分钟

## 容灾恢复

在灾难恢复方面，目前业界公认有三个目标值得努力。一是恢复时间，企业能忍受多长时间没有 IT，处于停业状态；二是网络多长时间能够恢复；三是业务层面的恢复。整个恢复过程中，最关键的衡量指标有两个：一个是 RTO，另一个是 RPO。

### RTO

RTO (Recovery Time Objective，复原时间目标)是企业可容许服务中断的时间长度。它是指灾难发生后，从 IT 系统当机导致业务停顿之时开始，到 IT 系统恢复至可以支持各部门运作、恢复运营之时，此两点之间的时间段称为 RTO。

比如说灾难发生后半天内便需要恢复，RTO值就是十二小时。

### RPO

RPO (Recovery Point Objective，复原点目标)是指从系统和应用数据而言，要实现能够恢复至可以支持各部门业务运作，系统及生产数据应恢复到怎样的更新程度。这种更新程度可以是上一周的备份数据，也可以是上一次交易的实时数据。

### 选择标准

根据以上两个简单的原则，企业不但可以对现有的数据系统作出，也可以按照既定的RTO及RPO要求，选购最适合的灾备方案。RTO及RPO与方案售价有着密切的关系，然而完美的方案当然是RTO及RPO皆为零，表示当灾难发生后，系统立即恢复，而且完全没有数据丢失，可是其造价是非常昂贵的，而且也不一定有这个必要。因此，最佳方案必需在RTO，RPO，维护及价钱多方面，都能达致平衡。尤其是中小企业，在资源紧拙的情况，应先好好了解对RTO及RPO的要求，然后再看看价钱，那就比较容易找到，适合企业的方案了。

在考虑采用哪个指标之前，IT 人首先要弄清楚一个基本概念，企业的容灾系统预防的是什么灾害，是多少年一遇的，能忍受多少损失，需要算出一个大概的成本，当然不一定很精确。其次，无论企业容灾系统是采用冷备、热备、温备、还是磁盘备份，几分钟恢复业务和几天恢复业务效果是完全不一样的。企业需要明确对恢复时间的容忍底限是多少。再从灾备本身的意义来讲，无论采用哪种衡量指标，最终目的是要能够很好地检验灾备系统的实用性能，否则就失去建立灾备的意义了。而灾备最核心的作用就是确保灾难发生后业务能够连续运行，交易中的数据完整保存，丢失越少越好。因此业务层面的恢复，企业要有一个底限。参考世界范围内一系列灾难恢复经验，国家之间的差别非常大。比如在美国，政府是第一位的，警察局对数据的恢复要求特别高。而在中国，无论什么性质，银行始终是排在第一位的。

作为银行，除开展自身业务之外，更多数据来自上下级银行间的财务汇兑与结算。站在管理者的位置上，一旦灾难发生，最重要的是在尽可能短的时间内排除障碍，恢复业务，保证系统做到连续运行。因此，从这个角度出发，银行容许系统停滞的时间应当越短越好。选择 RTO 刚好合适。但是，RTO 对成本要求太高，与回报似乎不成正比。企业资金不可能无限制地投入到一个灾备系统中。对于银行证券这样的联机交易事故处理非常紧密的金融机构而言，可能每一笔、每一单、每一分钱都很重要，所以都需要恢复。RPO 显然更为合适。许多时候进行选择并不意味着非此即彼，这与现实婚姻中一夫一妻的限制还是有差别的。RTO 和 RPO 对银行来讲都很重要。RTO 越短、RPO 越新，银行面临的损失就越小，但这也意味着系统开发成本将会急剧上升。许多时候，最佳的容灾解决方案却不一定是效益最好的。反之亦是。如何去平衡这中间的关系，不仅是门学问，更像是艺术。

# 集群故障

## 宕机

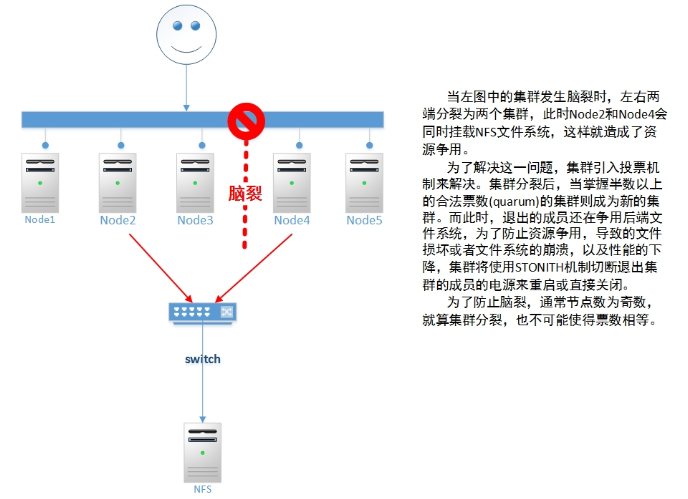
一般来说，“运行环境”是排名第一的宕机类别，大约35%的事件属于这一类。运行环境可以看作是支持数据库服务器运行的系统和资源集合，包括操作系统、硬盘以及网络等。性能问题紧随其后，占大约35%，然后是复制，占20%，最后剩下的10%包含各种类型的数据丢失和损坏，以及其他问题。

以下问题需要注意：

1. 在运行环境的问题中，最普遍的问题时磁盘空间耗尽；
2. 在性能问题中，最普遍的宕机原因却是是运行很糟糕的SQL，但也不一定都是这个原因，比如也有很多问题时由于服务器bug或错误的行为导致的；
3. 糟糕的Schema和索引设计是第二大影响性能的问题；
4. 复制问题通常是由于主备数据不一致导致；
5. 数据丢失问题通常由于DROP TABEL的误操作导致，并总是伴随着缺少可用备份的问题。

## 脑裂

### 概述



脑裂：因某种特殊原因造成集群分裂成两个小集群，而这两个小集群互相不能正常通信，此时，就会发生闹裂（Brain Split）现象。

脑裂是因为集群分裂导致的，集群中有节点因为处理器忙或者其他原因暂时停止响应时，与其他节点间的心跳出现故障，但这些节点还处于active状态，其他节点可能误认为该节点"已死"，从而争夺共享资源（如共享存储）的访问权，分裂为两部分独立节点。

**脑裂后果：**这时两个节点开始争抢共享资源，结果会导致系统混乱，数据损坏。

### 解决

**脑裂解决：**投票和参考节点的方法也能一定程度上解决脑裂的问题，但完全解决还需要资源隔离（Fencing）。

#### 投票

当有节点（一个或多个）和另外节点互相接收不到对方心跳信息时，如何决定哪一部分节点是正常运行的，而哪一部分是出现故障需要隔离的呢（避免集群脑裂）？

这时候通过法定票数（quorum）决定，即当有节点故障时，节点间投票决定哪个节点是有问题的，票数大于半数为合法。

**票数：**

每个节点可以设置票数，即决定节点在集群内是否合法（正常）的权限值，这个是可以有多有少的，例如有些节点的性能较好或有其他优势，可以设置较多的票数。

**法定票数（quorum）：**

当一个节点能和另一个节点保持心跳信息，该节点就获取得了另一个节点的票数，该节点获得的所有票数就是法定票数。

关于“投票”，有必要知道著名的Pasox算法和Zookeeper：

**Paxos算法：**

Paxos算法解决的是保证集群中每个节点执行相同的操作序列，可以保证分布式群集中的数据一致性。

例如，通过投票来对写操作进行全局编号，同一时刻，只有一个写操作被批准，同时并发的写操作要去争取选票，只有获得过半数选票的写操作才会被 批准（所以永远只会有一个写操作得到批准）；而其他的写操作竞争失败只好再发起一轮投票，就这样，在日复一日年复一年的投票中，所有写操作都被严格编号排序。

编号严格递增，当一个节点接受了一个编号为100的写操作，之后又接受到编号为99的写操作（因为网络延迟等很多不可预见原因），它马上能意识到自己数据不一致了，自动停止对外服务并重启同步过程。

任何一个节点挂掉都不会影响整个集群的数据一致性（总2n+1台，除非挂掉大于n台）。

**Zookeeper：**

Zookeeper是 Hadoop 大数据生态的一个独立组件，是 Google 的 Chubby一个开源的实现，可以说是Paxos算法（类似）的实现。

Zookeeper主要提供分布式协调服务，分布式应用程序可以基于它实现服务注册（高可用），同步服务，配置维护和命名服务等。

 Zookeeper真正提供的是类似我们普通电脑上的文件系统目录的功能，不过可以原子的进行增/删/改/查操作；具体要实现什么分布式协调服务，需要自己写程序来操作Zookeeper上的“目录”。

Zookeeper为什么可以作为分布式系统提供协调服务？

最主要的是Zookeeper本身运行的是一个由多个Zookeeper节点组成的稳定的高可用集群。

Zookeeper集群的高可用性和各节点“目录”数据的一致性正是基于 类似 Paxos算法实现的投票机制来保证的。

所以 Zookeeper集群节点数量最好也是单数（2n+1），当集群脑裂分区时，分区节点数量不超过一半的（<n+1），会自动停止对外服务。

比如：

 5台ZK节点组成ZK集群，当分成2台和3台的两个分区时，2台的分区会自动停止对外服务，3台的分区会继续提供服务。

另外，如果用6台节点组成ZK集群，当分成3台和3台的两个分区时，这两个分区都自动停止对外服务，所以，容错率和5台节点组成的集群的是一样的，更应该用单数（2n+1）节点数量组成集群。

#### 参考节点

比较特殊的是只有两个节点的集群，或两边票数相等

这时候可以借助另外的参考节点，如ping网关（可以是一个节点），可以和测试点ping通，但不可以和对方通，说明对方节点有问题，或本节点有问题；还有就是通过仲裁设备，如仲裁磁盘，每个节点都间隔一定时间不停往磁盘写数据，若监测到对方不再写入的时候，可能对方节点出故障。

 但最好还是使得组成集群的节点数量为单数台（2n+1），当集群分区脑裂时，节点数量小于一半（>n+1）的分区自动停止对外提供服务。

#### 资源隔离（Fencing）

当不能确定某个节点的状态时，通过fencing把对方干掉，确保共享资源被完全释放，前提是必须要有可靠的fence设备。

STONITH(Shoot The Other Node in the Head，“爆头”)，这种方式直接操作电源开关，当一个节点发生故障时，另一个节点如果能侦测到，就会通过网络发出命令，控制故障节点的电源开关，通过暂时断电，而又上电的方式使故障节点被重启动或者直接断电，这种方式需要硬件支持。

如果备份节点在某一时刻不能收到主节点的心跳信息时，那么如果此时备份节点立刻抢占资源时，而此时主节点正好在执行写操作，备份节点一旦也执行相应的写操作，会导致文件系统错乱或者服务器崩溃，因此在抢占资源的时候可以使用资源隔离机制来防止此类事件发生。而我们常常使用stonithd（即爆头）来使主节点不在抢占资源。

其中资源隔离包括：

节点级别：

STONITH（shoot the other node in the head，爆头。硬件方式），直接控制故障节点的电源，绝对彻底。

资源级别：

例如：FC SAN switch（软件方式）可以实现在存储资源级别拒绝某节点的访问。

## 资源争用

资源争用：当一个集群中因特殊情况分裂成两个小集群，且这两个集群都不能通信时，这时可能会造成资源争用的情况；分裂情况发生后，如果没有及时的决策，那么可能会因为两个小集群同时使用一个文件系统，而造成后端共享存储中文件损坏，甚至造成整个文件系统的崩溃。显然，这种情况是不允许发生的。

## 资源隔离

资源隔离：主要为了解决资源争用的问题。资源隔离分为节点级别隔离和资源级别隔离。

所谓节点级别隔离指当集群发生分裂时，即发生脑裂现象后，通过STONITH机制将资源隔离，并通过仲裁机制将分裂的票数不足的集群退出集群。STONITH指通过硬件设备，使得退出的主机重启或关机，或者通过交换机阻断退出的集群向外通信和资源通信的能力。

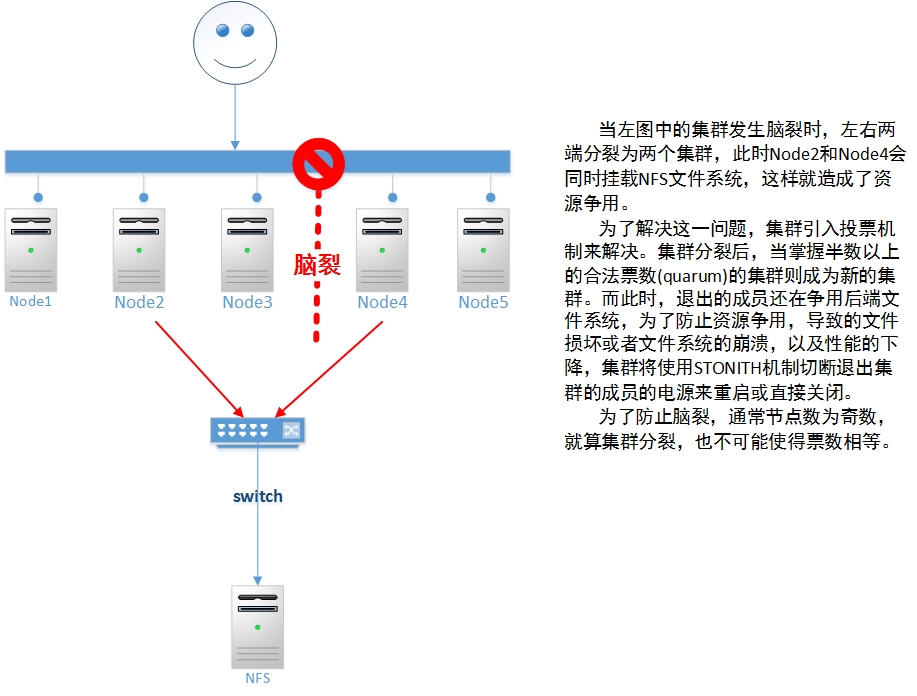
**资源隔离的解决方案：**

1、当集群分裂成两个小集群时会发生资源争用的情况，为避免争用后端存储系统而造成灾难性的系统崩溃，集群系统引入了投票机制，只有拥有半数以上合法票数的集群才能存活，否则就推出集群系统。

2、当集群为偶数时，如果分裂，两边可能都掌握相等的票数；因此，集群系统不应该为偶数，如果是偶数则需要一个额外的ping节点参与投票。

3、票数不足的集群退出集群服务后，为了保证它不会争用资源需要STONITH机制来进行资源隔离。

所以，为了防止脑裂，集群节点数一般为奇数，就算集群分裂，也不可能使得两个集群的票数相等。



# 高可用集群实现方式

实现高可用集群有三种方式：

## 主从方式（非对称）

这种方式组建的高可用集群通常包含2个节点和一个或多个服务器，其中一台作为主节点（active），另一台作为备份节点（standy）。备份节点随时都在检测主节点的健康状况，当主节点发生故障时，服务会自动切换到备份节点上以保证服务正常运行。

这种方式下的高可用集群其中的备份节点平时不会启动服务，只有发生故障时才会有用，因此感觉比较浪费。

## 对称方式

这种方式一般包含2个节点和一个或多个服务，其中每一个节点都运行着不同的服务且相互作为备份，两个节点互相检测对方的健康状况，这样当其中一个节点发生故障时，该节点上的服务会自动切换到另一个节点上去。这样可以保证服务正常运行。

## 多机方式

这种集群包含多个节点和多个服务。每一个节点都可能运行和不运行服务，每台服务器都监视着几个指定的服务，当其中的一个节点发生故障时，会自动切换到这组服务器中的一个节点上去。

# 高可用集群实现原理

高可用集群主要实现自动侦测(Auto-Detect)故障、自动切换/故障转移(FailOver)和自动恢复(FailBack)。

简单来说就是，用高可用集群软件实现故障检查和故障转移（故障/备份主机切换）的自动化，当然像负载均衡、DNS分发也可提供高可性。

## 实现高可用性

### 提升平均失效时间（MTBF）

### 降低平均恢复时间（MTTR）

提升系统可用性的解决方案之降低MTTR：

手段：冗余redundant

active/passive：主备

active/active：双主

active🡪HEARTBEAT🡪passive

active🡨🡪HEARTBEAT🡨🡪active

## 避免单点失效

单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量在系统设计的过程中避免单点。

方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了服务会受影响；如果有冗余备份，挂了还有其他backup能够顶上。

保证系统高可用，架构设计的核心准则是：冗余。

有了冗余之后，还不够，每次出现故障需要人工介入恢复势必会增加系统的不可服务实践。所以，又往往是通过“自动故障转移”来实现系统的高可用。

**典型互联网架构中，通过冗余+自动故障转移来保证系统的高可用特性。**

### 共享存储或磁盘复制

### MySQL同步复制

### 基于复制的冗余

## 自动侦测

自动侦测阶段由主机上的软件通过冗余侦测线，经由复杂的监听程序，逻辑判断，来相互侦测对方运行的情况。

常用的方法是：集群各节点间通过心跳信息判断节点是否出现故障。

## 故障转移和故障恢复

**自动切换**

自动切换阶段某一主机如果确认对方故障，则正常主机除继续进行原来的任务，还将依据各种容错备援模式接管预先设定的备援作业程序，并进行后续的程序及服务。

通俗地说，即当A无法为客户服务时，系统能够自动地切换，使B能够及时地顶上继续为客户提供服务，且客户感觉不到这个为他提供服务的对象已经更换。

通过上面判断节点故障后，将高可用集群资源（如VIP、httpd等，下面详见）从该不具备法定票数的集群节点转移到故障转移域（Failover Domain，可以接收故障资源转移的节点）。

**自动恢复**

自动恢复阶段在正常主机代替故障主机工作后，故障主机可离线进行修复工作。在故障主机修复后，透过冗余通讯线与原正常主机连线，自动切换回修复完成的主机上。

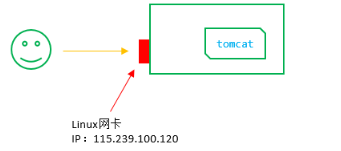
### 提升备库或切换角色

### 浮动IP/虚拟IP地址或IP接管

#### 背景

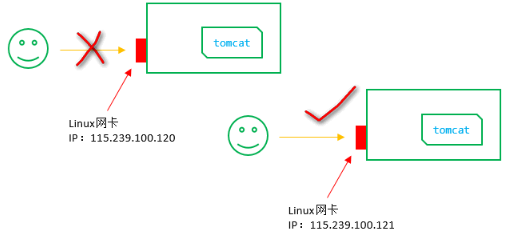
现在有一个场景，在一台Linux上部署一个web应用，应用跑在tomcat里面，linux网卡上的ip是115.239.100.120

大致就是如下的部署关系：



用户通过输入IP地址就能访问这个应用了，但是天有不测风云，有时候因为一些原因，服务会挂掉。于是开发人员就想了一个办法，在另外一个Linux上，部署同样的一个应用，这样这个服务挂了，另外一个顶上，于是架构就变成了下面。

这个样子：



如果用户访问100.120访问不了，就访问100.121就行了，但是这样引入了一个问题，用户需要记住两个IP，很不方便！

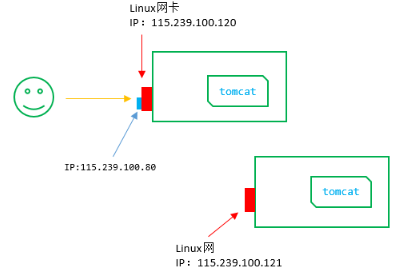
所以这个时候就需要浮动IP了。

#### 概述

虚拟IP地址（浮动IP）就是一个未分配给真实主机的IP，也就是说对外提供数据库服务的主机除了有一个真实的IP外还有一个虚拟IP，使用跟着两个IP中的任意一个都可以连接到这台主机，所有项目中的数据库连接一项配置的都是这个虚拟IP，当服务器发生故障无法对外提供服务时，动态将这个虚拟IP切换到备用主机。

上面我们搭建了两个一模一样的服务，但是用户需要记住两个IP，实在是太麻烦了，于是我们可以采用另外一个策略，

我们给其中一个机器在添加一个IP，让用户访问这个IP， 一个网卡是可以 添加多个IP的。



例如上面，我们给100.120主机添加一个100.80的IP，用户访问115.239.100.80这个IP，如果这台机器的服务挂了，我们就将

这个ip转移到100.121这个机器上，这样对于用户而言，他们始终访问的就只是一个IP而已。

在上面的场景中：115.239.100.80这个ip就是浮动IP，他会随着主机服务挂掉而转移到另外一个能够提供相同服务的主机上，

对于用户而言，他们始终觉得服务是正常的。一般而言，在高可用集群中使用的比较多，例如LVS集群等。

其实一个高可用集群一般而言都有浮动IP的，如果没有，机器宕机了还怎么玩？

#### 配置

利用单个网卡可以绑定多个IP地址的技术。

1、主服务器配置浮动IP

在主服务器上拷贝eth0位eth0:1，并做修改。

cp /etc/sysconfig/network-scripts/ifcfg-eth0 /etc/sysconfig/network-scripts/ifcfg-eth0:1

编辑文件ifcfg-eth0:1：

vim /etc/sysconfig/network-scripts/ifcfg-eth0:1

修改DEVICE为eth0:1，NM\_CONTROLLED设置为no，IPADDR改为浮动IP地址172.24.8.80，去掉网关信息，修改完如下：

DEVICE="eth0:1"

BOOTPROTO="static"

ONBOOT="yes"

NM\_CONTROLLED="no"

TYPE="Ethernet"

IPADDR="172.24.8.80"

NETMASK="255.255.255.0"

然后启动该网卡，验证是否生效：

ifup eth0:1

查看是否有浮动IP地址：

ip addr

接着访问浮动IP地址172.24.8.80，能访问成功，即和主服务器172.24.8.55访问的一样，则说明配置成功。

2、从服务器配置浮动IP

配置步骤和主服务器完全一样，eth0:1的IP地址同样设置成浮动IP172.24.8.80，在启动网卡测试之前，需在主服务器关闭浮动IP：

ifdown eth0:1

然后在从服务器启动eth0:1

ifup eth0:1

测试步骤也和主服务器一样。

### 中间件

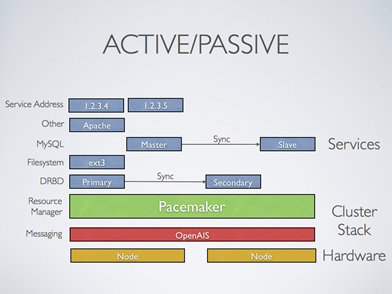
mycat 实现mysql高可用的中间价。

# 工作模型

## 主备模型

主备模型（Active/Passive，主从方式（非对称）），一个活动主节点，另一个不活动作为备用节点，当主节点故障，转移到备节点，这时备节点就成为了主节点。备节点完全冗余，造成一定浪费。

mysql、DRBD主从节点间还要进行同步：



## 双主模型

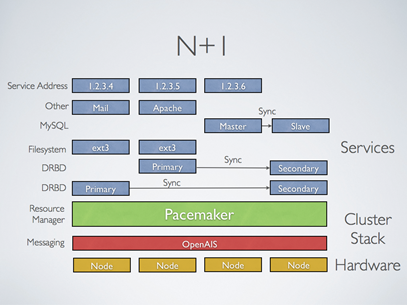
双主模型（对称方式），两个节点都是活动的，两个节点运行两个不同的服务，也互为备用节点。也可以提供同一个服务，比如ipvs，前端基于DNS轮询。这种模型可以使用比较均衡的主机配置，不会造成浪费。

## 多机模式

### N+1

多机模式（N+1）,N个活动主节点N个服务，一个备用节点。这需要额外的备用节点必须能够代替任何主节点，当任何主节点故障时，备节点能够负责它的角色对外提供相应的服务。

如下图，最后一个备用节点可以作为前两台主节点的DRBD和第三台主节点的MYSQL提供备用功能：



### N+M

N个活动主节点，M个备用节点。像上面的N+1模型，一个备用节点可能无法提供足够的备用冗余能力，备用节点的数量M是成本和可靠性要求之间的折衷。

也有一种说法：N-M: N个节点M个服务， N>M， 活动节点为N， 备用节点为N-M。

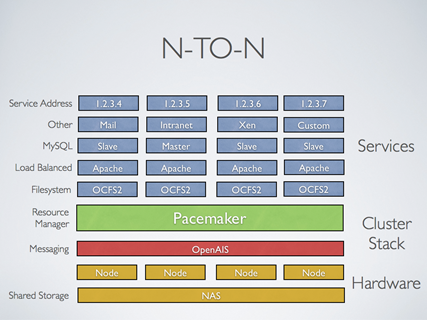
### N-to-1

这和N+1一样，也是N个活动主节点，一个备用节点；不同是的备用节点成为主节点只是暂时的，当原来故障的节点修复后，必须回转才能正常工作。

### N-to-N

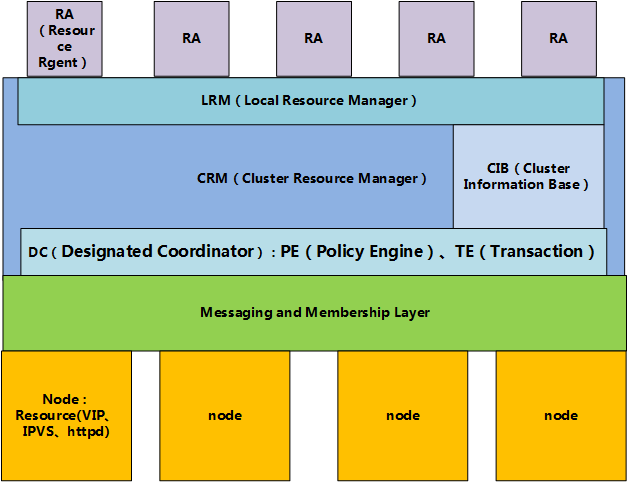
N个主节点N个备用节点。这是A/A双主和N + M模型的组合，N节点都有服务，如果一个坏了，剩下的每个节点都可以作为替代提供服务。

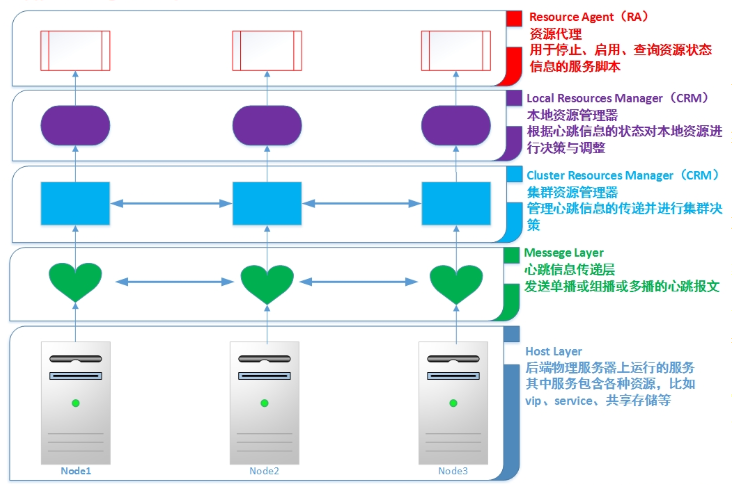
如下图，当共享存储是可用的，每一个节点都可能会被用于故障切换。起搏器甚至可以运行服务的多个副本，以分散工作量。



# 高可用集群架构及组件

高可用集群架构层次：





## 节点主机层

这一层主要是正在运行在物理主机上的服务，高可用集群相关的软件运行在各主机上，集群资源也是在各主机上。

## Messaging and Membership Layer

Messaging Layer：可以理解为信息层，主要的作用是传递当前节点的心跳信息，并告知给对方，这样对方就知道其他节点是否在线。如果不在线，则可以实现资源转移，这样另一台节点就可以充当主节点，并正常提供服务。传递心跳信息一般使用一根心跳线连接，该线接口可以使用串行接口也可以是以太网接口来连接。每一个节点上都包含信息层。

可以提供该组件的软件有：

1、heartbeat

heartbeat有三个版本即heartbeat v1、heartbeat v2和heartbeat v3

heartbeat v1是比较老的版本，heartbeat v2是目前稳定的版本，在做实验的时候使用该版本。

2、corosync(openAIS的子项目)

3、keepalive

4、cman

Heartbeat是比较常用的软件，Keepalived配置相对比较简单，而ultramonkey好像不怎么常用，Corosync比heartbeat功能还要强大，功能更加丰富。

信息传递层，传递集群信息的一种机制，通过监听UDP 694号端口，可通过单播、组播、广播的方式，实时快速传递信息，传递的内容为高可用集群的集群事务，例如：心跳信息，资源事务信息等等，只负责传递信息，不负责信息的计算和比较。

成员关系（Membership）层，这层最重要的作用是主节点（DC）通过Cluster Consensus Menbership Service（CCM或者CCS）这种服务由Messaging层提供的信息，来产生一个完整的成员关系。这层主要实现承上启下的作用，承上，将下层产生的信息生产成员关系图传递给上层以通知各个节点的工作状态；启下，将上层对于隔离某一设备予以具体实施。

## CRM（Cluster Resource Manager）

集群资源管理器层，它主要是用来提供那些不具有高可用的服务提供高可用性的。它需要借助Messaging Layer来实现工作，因此工作在Messaging Layer上层。

资源管理器的主要工作是收集messaging Layer传递的节点信息，并负责信息的计算和比较，并做出相应的动作，如服务的启动、停止和资源转移、资源的定义和资源分配。

在每一个节点上都包含一个CRM，且每个CRM都维护这一个CIB（Cluster Information Base，集群信息库），只有在主节点上的CIB是可以修改的，其他节点上的CIB都是从主节点那里复制而来的。

CRM会推选出一个用于计算和比较的节点，叫DC（Designated coordinator）指定协调节点，计算由PE（Policy Engine）策略引擎实现，计算出结果后的动作控制由TE（Transition Engine）事务引擎实现。

在每个节点上都有一个LRM（local resource manager）本地资源管理器，是CRM的一个子功能，接收TE传递过来的事务，在节点上采取相应动作，如运行RA脚本等。

## LRM

LRM：Local Resource Messager，叫做本地资源管理器，它是CRM的一个子组件，用来获取某个资源的状态，并且管理本地资源的。例如：当检测到对方没有心跳信息时，则会启动本地相应服务。

## DC

DC：可以理解为事务协调员，这个是当多个节点之间彼此收不到对方的心跳信息时，这样各个节点都会认为对方发生故障了，于是就会产尘分裂状况（分组）。并且都运行着相关服务，因此就会发生资源争夺的状况。因此，事务协调员在这种情况下应运而生。事务协调员会根据每个组的法定票数来决定哪些节点启动服务，哪些节点停止服务。 例如高可用集群有3个节点，其中2个节点可以正常传递心跳信息，与另一个节点不能相互传递心跳信息，因此，这样3个节点就被分成了2组，其中每一个组都会推选一个DC，用来收集每个组中集群的事务信息，并形成CIB，且同步到每一个集群节点上。同时DC还会统计每个组的法定票数（quorum），当该组的法定票数大于二分之一时，则表示启动该组节点上的服务；否则停止该节点上的服务。对于某些性能比较强的节点来说，它可以投多张票，因此每个节点的法定票数并不是只有一票，需要根据服务器的性能来确定。DC一般位于主节点上。

## PE和TE

PE和TE也是DC的子组件，其中：

PE（Policy Engine）：策略引擎，来定义资源转移的一整套转移方式，但只是做策略者，并不亲自来参加资源转移的过程，而是让TE来执行自己的策略。

TE（Transition Engine）： 就是来执行PE做出的策略的并且只有DC上才运行PE和TE。

## stonithd组件

STONITH(Shoot The Other Node in the Head，”爆头“)， 这种方式直接操作电源开关，当一个节点发生故障时，另 一个节点如果能侦测到，就会通过网络发出命令，控制故障节点的电源开关，通过暂时断电，而又上电的方式使故障节点被重启动或者直接断电， 这种方式需要硬件支持。

如果备份节点在某一时刻不能收到主节点的心跳信息时，那么如果此时备份节点立刻抢占资源时，而此时主节点正好在执行写操作，备份节点一旦也执行相应的写操作，会导致文件系统错乱或者服务器崩溃，因此在抢占资源的时候可以使用资源隔离机制来防止此类事件发生。而我们常常使用stonithd（即爆头）来使主节点不在抢占资源。

其中资源隔离包括：

1、节点级别

使用stonithd设备来实现

2、资源级别

例如：使用FC SAN switch可以实现在存储资源级别拒绝某节点的访问

## 共享存储

对于某些服务如http、mysql等服务，需要将某些数据共享，这样当使用不同的节点来访问存储设备时，都可以返回正确的信息。如果不使用存储设备，假设http服务为例，当某个客户想访问某个图片时，如果这个图片只放在某个指定的服务器上时，一旦该服务器挂了，http服务就会切换到另一台设备上去，而另一台设备上面没有该图片，那么该用户此时就不能访问该图片了，当然这种情况是我们不想看到了。为了解决这类事件发生，可以使用共享存储设备，将相关的数据放在共享设备上，这样无论那一台服务器挂了，都不会影响用户的访问。

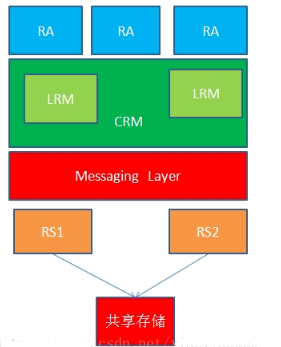
常用的共享存储设备有如下三种：

DAS:Direct Attached Storage,直接附加存储

NAS：Network Attached Storage，网络附加存储

SAN：Storage Area Network，存储区域网络

因此，一个高可用集群服务的组件架构大概是这样子的：



## 资源

其实资源就是启动一个服务需要的子项目。例如启动一个httpd服务，需要ip，也需要服务脚本、还需要文件系统（用来存储数据的），这些我们都可以统称为资源。因此，实现一个高可用集群一般需要ip、服务（脚本）和文件系统（存储数据），当然有些高可用集群不需要存储设备的。

资源也是有类型的，可以分为这样几类：

1、primitive：可以理解为主资源，有时候看到的会是native，都是一个意思，该资源只在主节点上有。（当然备份节点一旦将资源夺过来了，也就成了主节点，因此，主节点是相对来说的）

2、group：组资源，将多个资源绑定在一个同一个组上面且运行在同一个节点上。

3、clone：是将primitive资源克隆n份且运行在每一个节点上

4、master/slave：也是将primitive克隆2份、其中master和slave节点各运行一份，且只能在这2个节点上运行。

对于某些集群服务来说，启动相关的资源是有先后顺序的。例如启动一个mysql集群服务，首先应该先挂载共享存储设备，否则即时mysql服务启动起来了，用户也访问不了数据。因此，一般说来，我们需要将资源进行约束。资源约束有如下几类：

1、位置约束（location）：资源对节点的倾向程度，通常可以使用一个分数（score）来定义，当score为正值时，表示资源倾向与此节点；负值表示资源倾向逃离于此节点。也可以将score定义为-inf(负无穷大)和inf（正无穷大）。例如：有三个节点rs1、rs2、rs3当rs1是主节点且发生故障时，则比较rs2和rs3的score值，谁是正值，则资源将会转移到哪个节点上去。

2、排列约束（colocation）：用来定义资源是否可以在一起，通常也是使用一个score来定义的。当score是正值表示资源可以在一起；否则表示不可以在一起。通过定义资源类型为group也可以来将所有资源绑定在一起。

3、顺序约束（order）：用来定义资源启动和停止的顺序。例如，首先应该先挂载共享存储，在启动httpd或mysqld服务才行吧。

**资源粘性：**用来定义资源是否倾向留在该节点。通常使用score来定义，当score为正数表示乐意留在当前节点，负数表示不乐意留在当前节点。

当某个高可用集群即包含资源粘性又包含位置约束，一旦该节点发生故障后，资源就会转移到另一个节点上去。但是当之前的节点恢复正常时，需要比较所有的资源粘性之和与所有位置约束之和谁大谁小，这样资源才会留在大的一方。

**资源转移**

将有故障节点的VIP设置到另一个节点上去，并在另一个节点启用相应的服务，挂载相应的存储设备等等都可以叫做资源转移。

## RA（Resource Rgent）

资源代理层，简单的说就是能够集群资源进行管理的脚本，如启动start，停止stop、重启restart和查询状态信息status等操作的脚本。LRM本地资源管理器负责运行。

资源代理分为：

1、Legacy heartbeat（heatbeat v1版本的资源管理）；

2、LSB（Linux Standard Base），主要是/etc/init.d/\*目录下的脚本start/stop/restart/status；

3、OCF（Open Cluster Famework），比LSB更专业，更加通用，除了上面的四种操作，还包含monitor、validate-all等集群操作。

4、STONITH：实现节点隔离

# 高可用集群软件

## Messaging Layer集群信息层软件

1、heartbeat (v1, v2)

2、heartbeat v3

可以拆分为：heartbeat, pacemaker, cluster-glue

3、corosync

从OpenAIS分离的项目。

4、cman

5、keepalived

一般用于两个节点的集群。

6、ultramokey

## CRM集群资源管理器软件

1、Haresource

heartbeat v1 v2包含，使用文本配置接口haresources

2、crm

heartbeat v2包含，可以使用crmsh或者heartbeat-gui来进行配置

3、pacemaker

heartbeat v3分离出来的项目，配置接口：CLI：crm、pcs和GUI：hawk(WEB-GUI)、LCMC、pacemaker-mgmt、pcs

4、rgmanager

Cman包含，使用rgmanager(resource group manager)实现管理, 具有Failover Domain故障转移域这一特性，也可以使用RHCS（Redhat Cluster Suite）套件来进行管理：Conga的全生命周期接口，Conga（luci/ricci）先安装后，可用其安装高可用软件，再进行配置。

## 常用组合

heartbeat v2+haresource(或crm) (说明：一般常用于CentOS 5.X)

heartbeat v3+pacemaker (说明：一般常用于CentOS 6.X)

corosync+pacemaker (说明：现在最常用的组合)

cman + rgmanager (说明：红帽集群套件中的组件，还包括gfs2,clvm)

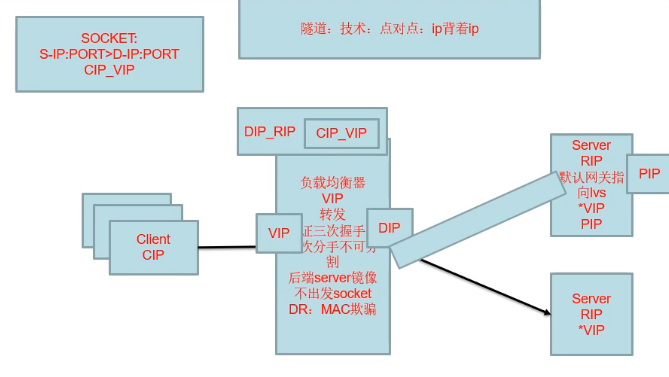
keepalived+lvs (说明：常用于lvs的高可用)

## Zookeeper

## LVS

**四层（传输层**，位于内核）：快，**应对更大的流量**。

四层：TUN/VPN



## haproxy

## Keepalive

### 概述

Keepalive是集群管理中保证集群高可用（HA：High Available）的服务软件。

1. 需要心跳机制探测后端RS（Resource Server）是否提供服务
2. 探测down，需要从LVS中删除该RS
3. 探测发送从down到up，需要从LVS中再次添加RS
4. LVS DR，需要主备（HA）

**术语：**

虚拟路由器：Virtual Router

虚拟路由器标识：VRID（0~255），唯一标识虚拟路由器

物理路由器：master：主设备backup

备用设备priority：优先级

VIP：Virtual IP

VMAC：Virtual MAC

### 原理

vrrp协议：Virtual Router Redundancy Protocol

IP漂移

## Heartbeat

Keepalived使用的vrrp协议方式，虚拟路由冗余协议 (Virtual Router Redundancy Protocol，简称VRRP)；Heartbeat是基于主机或网络的服务的高可用方式。

keepalived的目的是模拟路由器的双机，heartbeat的目的是用户service的双机。

lvs的高可用建议用keepavlived，业务的高可用建议heartbeat。

**说明：**

1）Keepalived使用更简单：从安装、配置、使用、维护等角度上对比，Keepalived都比Heartbeat要简单得多，尤其是Heartbeat2.1.4后拆分成3个子项目，安装、配置、使用都比较复杂，尤其是出问题的时候，都不知道具体是哪个子系统出问题了；而Keepalived只有1个安装文件、1个配置文件，配置文件也简单很多；

2）**Heartbeat功能更强大**：Heartbeat虽然复杂，但功能更强大，配套工具更全，适合做大型集群管理，而Keepalived主要用于集群倒换，基本没有管理功能；

3）协议不同：Keepalived使用VRRP协议进行通信和选举，Heartbeat使用心跳进行通信和选举；Heartbeat除了走网络外，还可以通过串口通信，貌似更可靠；

4）使用方式基本类似：如果要基于两者设计高可用方案，最终都要根据业务需要写自定义的脚本，Keepalived的脚本没有任何约束，随便怎么写都可以；Heartbeat的脚本有约束，即要支持service start/stop/restart这种方式，而且Heartbeart提供了很多默认脚本，简单的绑定ip，启动apache等操作都已经有了。

使用建议：**优先使用Keepalived，当Keepalived不够用的时候才选择Heartbeat**。

## Nginx

**七层（应用层）**：慢于四层，看得懂客户端的请求资源。

**上述LVS和Nginx详见《负载均衡》。**

# 高可用方案

## 客户端实现的高可用方案

以memcache 为例，客户端同时与好几个服务保持连接，按照一定的规则去调用服务，当服务挂掉之后，重新调整规则。当然，如果服务器不做主从备份的话，可能会造成部分数据丢失。

## 服务之间通信实现高可用

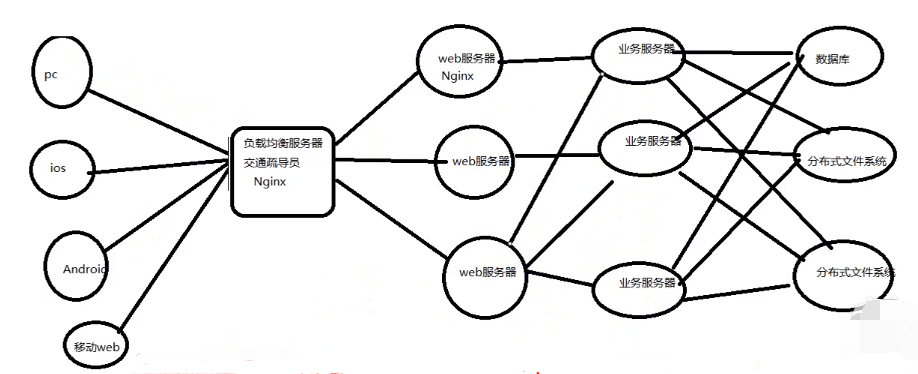
这种经典的案例就是redis了，各个redis之间保持通信，当主服务挂掉之后从服务就会升为主服务。对于客户端来说几乎是透明的。

## 通过中间件实现高可用

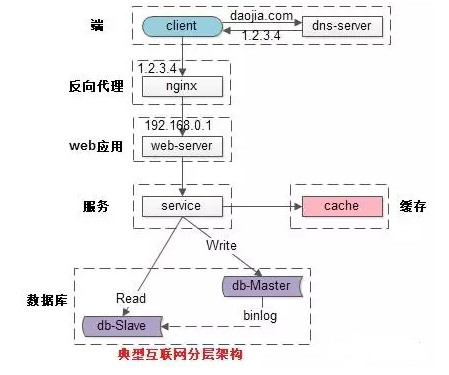
mycat实现mysql高可用的中间价。

早期版本redis不支持集群，那时候redis的高可用也是基于中间件来做的。

# 互联网分层



典型互联网架构中，通过冗余+自动故障转移来保证系统的高可用特性。



常见互联网分布式架构如上，分为：

（1）客户端层：典型调用方是浏览器browser或者手机应用APP

（2）反向代理层：系统入口，反向代理

（3）站点应用层：实现核心应用逻辑，返回html或者json

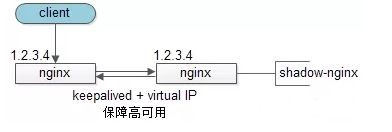
（4）服务层：如果实现了服务化，就有这一层

（5）数据-缓存层：缓存加速访问存储

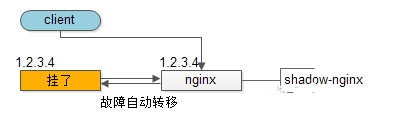
（6）数据-数据库层：数据库固化数据存储

整个系统的高可用，又是通过每一层的冗余+自动故障转移来综合实现的。

## 客户端层->反向代理层

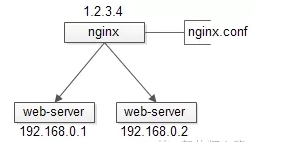


客户端层到反向代理层的高可用，是通过反向代理层的冗余来实现的。以nginx为例：有两台nginx，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测，相同virtual IP（浮动IP）提供服务。

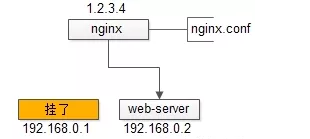


自动故障转移：当nginx挂了的时候，keepalived能够探测到，会自动的进行故障转移，将流量自动迁移到shadow-nginx，由于使用的是相同的virtual IP，这个切换过程对调用方是透明的。

## 反向代理层->站点层

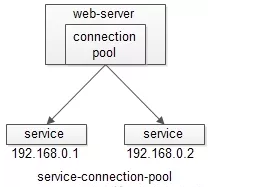


反向代理层到站点层的高可用，是通过站点层的冗余来实现的。假设反向代理层是nginx，nginx.conf里能够配置多个web后端，并且nginx能够探测到多个后端的存活性。

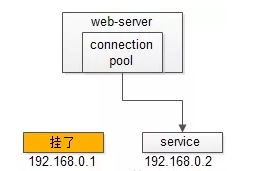


自动故障转移：当web-server挂了的时候，nginx能够探测到，会自动的进行故障转移，将流量自动迁移到其他的web-server，整个过程由nginx自动完成，对调用方是透明的。

## 站点层->服务层

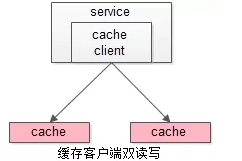


站点层到服务层的高可用，是通过服务层的冗余来实现的。“服务连接池”会建立与下游服务多个连接，每次请求会“随机”选取连接来访问下游服务。



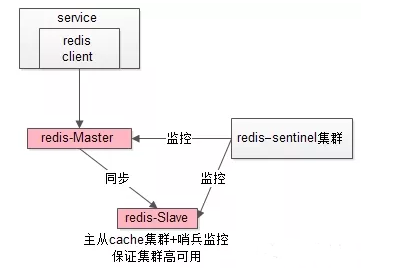
自动故障转移：当service挂了的时候，service-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的service，整个过程由连接池自动完成，对调用方是透明的（所以说RPC-client中的服务连接池是很重要的基础组件）。

## 服务层>缓存层



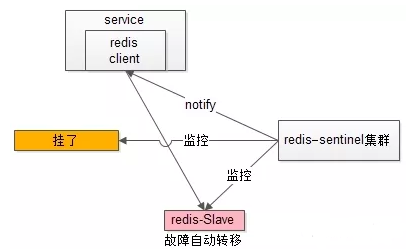
服务层到缓存层的高可用，是通过缓存数据的冗余来实现的。

缓存层的数据冗余又有几种方式：第一种是利用客户端的封装，service对cache进行双读或者双写。



缓存层也可以通过支持主从同步的缓存集群来解决缓存层的高可用问题。

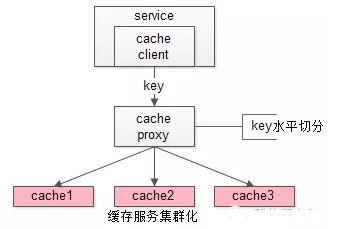
以redis为例，redis天然支持主从同步，redis官方也有sentinel哨兵机制，来做redis的存活性检测。



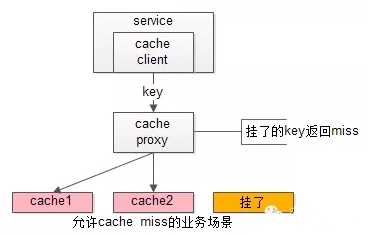
自动故障转移：当redis主挂了的时候，sentinel能够探测到，会通知调用方访问新的redis，整个过程由sentinel和redis集群配合完成，对调用方是透明的。

注意：业务对缓存并不一定有“高可用”要求，更多的对缓存的使用场景，是用来“加速数据访问”：把一部分数据放到缓存里，如果缓存挂了或者缓存没有命中，是可以去后端的数据库中再取数据的。

这类允许“cache miss”的业务场景，缓存架构的建议是：



将kv缓存封装成服务集群，上游设置一个代理（代理可以用集群冗余的方式保证高可用），代理的后端根据缓存访问的key水平切分成若干个实例，每个实例的访问并不做高可用。

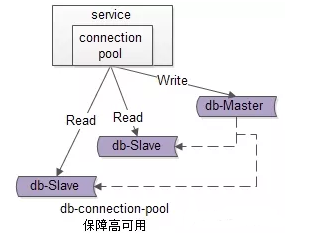


缓存实例挂了屏蔽：当有水平切分的实例挂掉时，代理层直接返回cache miss，此时缓存挂掉对调用方也是透明的。key水平切分实例减少，不建议做re-hash，这样容易引发缓存数据的不一致。

## 服务层>数据库层

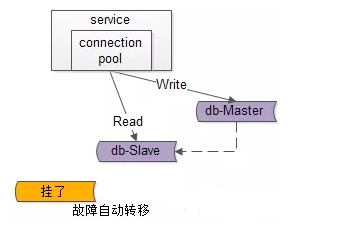
大部分互联网技术，数据库层都用了“主从同步，读写分离”架构，所以数据库层的高可用，又分为“读库高可用”与“写库高可用”两类。

## 服务层>数据库层“读”



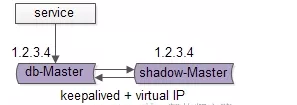
服务层到数据库读的高可用，是通过读库的冗余来实现的。

既然冗余了读库，一般来说就至少有2个从库，“数据库连接池”会建立与读库多个连接，每次请求会路由到这些读库。



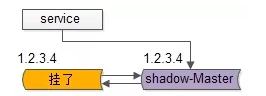
自动故障转移：当读库挂了的时候，db-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的读库，整个过程由连接池自动完成，对调用方是透明的（所以说DAO中的数据库连接池是很重要的基础组件）。

## 服务层>数据库层“写”



服务层到数据库写的高可用，是通过写库的冗余来实现的。

以mysql为例，可以设置两个mysql双主同步，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测，相同virtual IP提供服务。



自动故障转移：当写库挂了的时候，keepalived能够探测到，会自动的进行故障转移，将流量自动迁移到shadow-db-master，由于使用的是相同的virtual IP，这个切换过程对调用方是透明的。

## 总结

高可用HA（High Availability）是分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。

方法论上，高可用是通过冗余+自动故障转移来实现的。

整个互联网分层系统架构的高可用，又是通过每一层的冗余+自动故障转移来综合实现的，具体的：

（1）【客户端层】到【反向代理层】的高可用，是通过反向代理层的冗余实现的，常见实践是keepalived + virtual IP自动故障转移

（2）【反向代理层】到【站点层】的高可用，是通过站点层的冗余实现的，常见实践是nginx与web-server之间的存活性探测与自动故障转移

（3）【站点层】到【服务层】的高可用，是通过服务层的冗余实现的，常见实践是通过service-connection-pool来保证自动故障转移

（4）【服务层】到【缓存层】的高可用，是通过缓存数据的冗余实现的，常见实践是缓存客户端双读双写，或者利用缓存集群的主从数据同步与sentinel保活与自动故障转移；更多的业务场景，对缓存没有高可用要求，可以使用缓存服务化来对调用方屏蔽底层复杂性

（5）【服务层】到【数据库“读”】的高可用，是通过读库的冗余实现的，常见实践是通过db-connection-pool来保证自动故障转移

（6）【服务层】到【数据库“写”】的高可用，是通过写库的冗余实现的，常见实践是keepalived + virtual IP自动故障转移

# 数据库高可用

## 架构

目前MySQL高可用方案有很多，几种典型的高可用架构选型有:

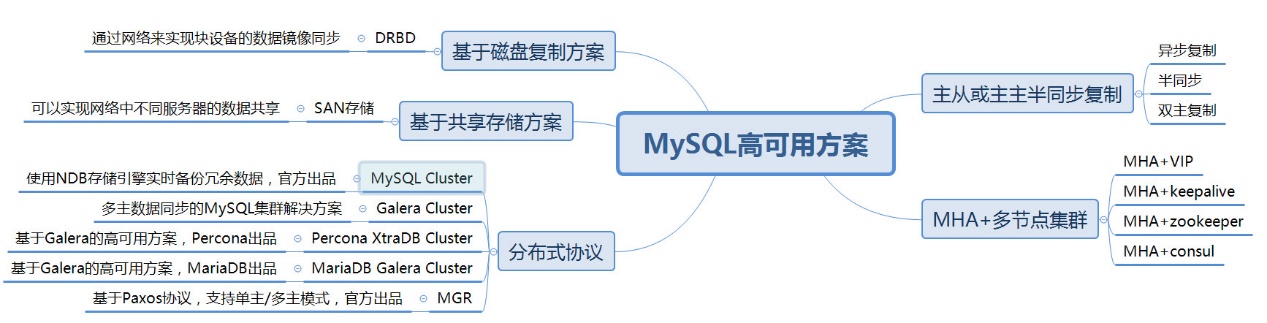
1、主从或主主半同步复制：通过依赖MySQL本身的复制，Master制作一个或多个热副本，在Master故障时，将服务切换到热副本从而达到高可用的效果。

2、MHA+多节点集群：基于MHA的集群方案，通常和其他第三方方案组合实现

3、分布式协议：基于分布式协议的高可用方案，常见的有Galera Cluster，PXC和MGR

4、基于共享存储方案：如SAN存储，这种方案可以实现网络中不同服务器的数据共享，共享存储能够为数据库服务器和存储解耦。

5、基于磁盘复制方案：如DRBD，DRDB是一个以linux内核模块方式实现的块级别同步复制技术。它通过网卡将主服务器的每个块复制到另外一个服务器块设备上，并在主设备提交块之前记录下来。类似共享存储解决方案。



## 选型

目前MySQL高可用方案有很多，几种典型的高可用架构选型有:

注：两个主流的架构：MMM架构和MHA架构。

### 主从或主主半同步复制

通过依赖MySQL本身的复制，Master制作一个或多个热副本，在Master故障时，将服务切换到热副本从而达到高可用的效果。

### MMM

MMM（Master-Master replication manager for MySQL）是一套支持**双主故障切换和双主日常管理**的脚本程序。MMM采用Perl语言开发，主要用来监控和管理MySQL Master-Master（双主）复制，虽然叫做双主复制，但是业务上同一时刻只允许对一个主进行写操作，另一台备选主上提供部分读服务，以加速在主主切换时备选主的预热。可以说MMM这套脚本程序一方面实现了故障切换的功能，另一方面其内部附加的工具脚本也可以实现多个slave的read负载均衡。

由于MMM无法完全地保证数据一致性，所以MMM适用于对数据一致性要求不是很高，但是又想最大程度的保证业务可用性的场景。对于那些对数据的一致性一致性要求很高的业务，非常不建议采用MMM这种高可用架构。

假设存在Master1（db1），Master2（db2），Slave（db3）,db1、db2、db3之间为一主两从的复制关系，一旦发生db2、db3延时于db1时，这个时刻db1 MySQL宕机，db3将会等待数据追上db1后，再重新指向新的主db2，进行change master to db2操作，在db1宕机的过程中，一旦db2落后于db1，这时发生切换，db2变成可写状态，数据的一致性将无法保证（即slave节点db3追上原来主节点db1，然后指向新的主节点db2，此时数据不一致）。

### MHA+多节点集群

基于MHA的集群方案，通常和其他第三方方案组合实现。

无论是高可用计算架构，还是高可用存储架构，其本质的设计目的都是为了解决部分服务器故障的场景下，如何保证系统能够继续提供服务。但在一些极端场景下，有可能所有服务器都出现故障。例如，典型的有机房断电、机房火灾、地震、水灾……这些极端情况会导致某个系统所有服务器都故障，或者业务整体瘫痪，而且即使有其他地区的备份，把备份业务系统全部恢复到能够正常提供业务，花费的时间也比较长，可能是半小时，也可能是12小时。因为备份系统平时不对外提供服务，可能会存在很多隐藏的问题没有发现。如果业务期望达到即使在此类灾难性故障的情况下，业务也不受影响，或者在几分钟内就能够很快恢复，那么就需要设计异地多活架构。

#### MHA原理

MHA（Master High Availability）是一套优秀的作为MySQL高可用环境下故障切换和主从提升的高可用软件。在MySQL故障切换过程中，MHA能做到在0~30秒之内自动完成数据库的故障切换操作，并且在进行故障切换过程中，MHA能在最大程度上保证数据的一致性，以达到真正意义上的高可用。

它由两部分组成：MHA Manager（管理节点）和MHA Node（数据节点）。MHA Manager可以单独部署在一台独立的机器上管理多个master-slave集群，也可以部署在一台slave上。MHA Node运行在每台MySQL服务器上，MHA Manager会定时探测集群中的master节点，当master出现故障时，它可以自动将**最新数据**的slave提升为新的master（这样就避免数据不一致），然后将所有其他的slave重新指向新的master。整个故障转移过程对应用程序是完全透明的。

在MHA自动故障切换过程中，MHA试图从宕机的主服务器上保存二进制日志，最大程度地保证数据不丢失，但这并不总是可行的。例如，主服务器硬件故障或无法通过ssh访问，MHA无法保存二进制日志，只进行故障转移而丢失了最新数据。使用MySQL5.5的**半同步复制**，可以大大降低数据丢失的风险。如果只有一个slave已经收到了最新的二进制日志，MHA可以将最新的二进制日志应用于其他所有的slave服务器上，因此它们彼此保持一致性。

目前MHA主要支持一主多从的架构，要搭建MHA，要求一个复制集群中必须最少有三台数据库服务器，一主二从，即一台充当master，一台充当备用master，另一台充当slave。MHA的工作原理：

1. 从宕机崩溃的master保存二进制日志事件（binlog events）；
2. 识别含有最新更新的slave；
3. 应用差异的中继日志（relay log）到其他slave；
4. 应用从master保存的二进制日志事件（binlog events）；
5. 提升一个slave为新master；
6. 使其他的slave连接新的master进行复制。

#### 应用场景

顾名思义，异地多活架构的关键点就是异地、多活，其中异地就是指地理位置上不同的地方，类似于“不要把鸡蛋都放在同一篮子里”；多活就是指不同地理位置上的系统都能够提供业务服务，这里的“活”是活动、活跃的意思。判断一个系统是否符合异地多活，需要满足两个标准：

正常情况下，用户无论访问哪一个地点的业务系统，都能够得到正确的业务服务。

某个地方业务异常的时候，用户访问其他地方正常的业务系统，能够得到正确的业务服务。

与“活”对应的是字是“备”，备是备份，正常情况下对外是不提供服务的，如果需要提供服务，则需要大量的人工干预和操作，花费大量的时间才能让“备”变成“活”。

单纯从异地多活的描述来看，异地多活很强大，能够保证在灾难的情况下业务都不受影响。那是不是意味着不管什么业务，我们都要去实现异地多活架构呢？其实不然，因为实现异地多活架构不是没有代价的，相反其代价很高，具体表现为：

系统复杂度会发生质的变化，需要设计复杂的异地多活架构。

成本会上升，毕竟要多在一个或者多个机房搭建独立的一套业务系统。

因此，异地多活虽然功能很强大，但也不是每个业务不管三七二十一都要上异地多活。例如，常见的新闻网站、企业内部的IT系统、游戏、博客站点等，如果无法承受异地多活带来的复杂度和成本，是可以不做异地多活的，只需要做异地备份即可。因为这类业务系统即使中断，对用户的影响并不会很大，例如，A新闻网站看不了，用户换个新闻网站即可。而共享单车、滴滴出行、支付宝、微信这类业务，就需要做异地多活了，这类业务系统中断后，对用户的影响很大。例如，支付宝用不了，就没法买东西了；滴滴用不了，用户就打不到车了。

当然，如果业务规模很大，能够做异地多活的情况下还是尽量。首先，这样能够在异常的场景下给用户提供更好的体验；其次，业务规模很大肯定会伴随衍生的收入，例如广告收入，异地多活能够减少异常场景带来的收入损失。同样以新闻网站为例，虽然从业务的角度来看，新闻类网站对用户影响不大，反正用户也可以从其他地方看到基本相同的新闻，甚至用户几个小时不看新闻也没什么问题。但是从网站本身来看，几个小时不可访问肯定会影响用户对网站的口碑；其次几个小时不可访问，网站上的广告收入损失也会很大。

#### 架构模式

根据地理位置上的距离来划分，异地多活架构可以分为同城异区、跨城异地、跨国异地。接下来我详细解释一下每一种架构的细节与优缺点。

##### 同城异区

同城异区指的是将业务部署在同一个城市不同区的多个机房。例如，在北京部署两个机房，一个机房在海淀区，一个在通州区，然后将两个机房用专用的高速网络连接在一起。

如果我们考虑一些极端场景（例如，美加大停电、新奥尔良水灾），同城异区似乎没什么作用，那为何我们还要设计同城异区这种架构呢？答案就在于“同城”。

同城的两个机房，距离上一般大约就是几十千米，通过搭建高速的网络，同城异区的两个机房能够实现和同一个机房内几乎一样的网络传输速度。这就意味着虽然是两个不同地理位置上的机房，但逻辑上我们可以将它们看作同一个机房，这样的设计大大降低了复杂度，减少了异地多活的设计和实现复杂度及成本。

那如果采用了同城异区架构，一旦发生新奥尔良水灾这种灾难怎么办呢？很遗憾，答案是无能为力。但我们需要考虑的是，这种极端灾难发生概率是比较低的，可能几年或者十几年才发生一次。其次，除了这类灾难，机房火灾、机房停电、机房空调故障这类问题发生的概率更高，而且破坏力一样很大。而这些故障场景，同城异区架构都可以很好地解决。因此，结合复杂度、成本、故障发生概率来综合考虑，同城异区是应对机房级别故障的最优架构。

##### 跨城异地

跨城异地指的是业务部署在不同城市的多个机房，而且距离最好要远一些。例如，将业务部署在北京和广州两个机房，而不是将业务部署在广州和深圳的两个机房。

为何跨城异地要强调距离要远呢？前面我在介绍同城异区的架构时提到同城异区不能解决新奥尔良水灾这种问题，而两个城市离得太近又无法应对如美加大停电这种问题，跨城异地其实就是为了解决这两类问题的，因此需要在距离上比较远，才能有效应对这类极端灾难事件。

跨城异地虽然能够有效应对极端灾难事件，但“距离较远”这点并不只是一个距离数字上的变化，而是量变引起了质变，导致了跨城异地的架构复杂度大大上升。距离增加带来的最主要问题是两个机房的网络传输速度会降低，这不是以人的意志为转移的，而是物理定律决定的，即光速真空传播大约是每秒30万千米，在光纤中传输的速度大约是每秒20万千米，再加上传输中的各种网络设备的处理，实际还远远达不到理论上的速度。

除了距离上的限制，中间传输各种不可控的因素也非常多。例如，挖掘机把光纤挖断、中美海底电缆被拖船扯断、骨干网故障等，这些线路很多是第三方维护，针对故障我们根本无能为力也无法预知。例如，广州机房到北京机房，正常情况下RTT大约是50毫秒左右，遇到网络波动之类的情况，RTT可能飙升到500毫秒甚至1秒，更不用说经常发生的线路丢包问题，那延迟可能就是几秒几十秒了。

以上描述的问题，虽然同城异区理论上也会遇到，但由于同城异区距离较短，中间经过的线路和设备较少，问题发生的概率会低很多。而且同城异区距离短，即使是搭建多条互联通道，成本也不会太高，而跨城异区距离太远，搭建或者使用多通道的成本会高不少。

跨城异地距离较远带来的网络传输延迟问题，给异地多活架构设计带来了复杂性，如果要做到真正意义上的多活，业务系统需要考虑部署在不同地点的两个机房，在数据短时间不一致的情况下，还能够正常提供业务。这就引入了一个看似矛盾的地方：数据不一致业务肯定不会正常，但跨城异地肯定会导致数据不一致。

如何解决这个问题呢？重点还是在“数据”上，即根据数据的特性来做不同的架构。如果是强一致性要求的数据，例如银行存款余额、支付宝余额等，这类数据实际上是无法做到跨城异地多活的。我们来看一个假设的例子，假如我们做一个互联网金融的业务，用户余额支持跨城异地多活，我们的系统分别部署在广州和北京，那么如果挖掘机挖断光缆后，会出现如下场景：

用户A余额有10000元钱，北京和广州机房都是这个数据。

用户A向用户B转了5000元钱，这个操作是在广州机房完成的，完成后用户A在广州机房的余额是5000元。

由于广州和北京机房网络被挖掘机挖断，广州机房无法将余额变动通知北京机房，此时北京机房用户A的余额还是10000元。

用户A到北京机房又发起转账，此时他看到自己的余额还有10000元，于是向用户C转账10000元，转账完成后用户A的余额变为0。

用户A到广州机房一看，余额怎么还有5000元？于是赶紧又发起转账，转账5000元给用户D；此时广州机房用户A的余额也变为0了。

最终，本来余额10000元的用户A，却转了20000元出去给其他用户。

对于以上这种假设场景，虽然普通用户很难这样自如地操作，但如果真的这么做，被黑客发现后，后果不堪设想。正因为如此，支付宝等金融相关的系统，对余额这类数据，一般不会做跨城异地的多活架构，而只能采用同城异区这种架构。

而对数据一致性要求不那么高，或者数据不怎么改变，或者即使数据丢失影响也不大的业务，跨城异地多活就能够派上用场了。例如，用户登录（数据不一致时用户重新登录即可）、新闻类网站（一天内的新闻数据变化较少）、微博类网站（丢失用户发布的微博或者评论影响不大），这些业务采用跨城异地多活，能够很好地应对极端灾难的场景。

##### 跨国异地

跨国异地指的是业务部署在不同国家的多个机房。相比跨城异地，跨国异地的距离就更远了，因此数据同步的延时会更长，正常情况下可能就有几秒钟了。这种程度的延迟已经无法满足异地多活标准的第一条：“正常情况下，用户无论访问哪一个地点的业务系统，都能够得到正确的业务服务”。例如，假设有一个微博类网站，分别在中国的上海和美国的纽约都建了机房，用户A在上海机房发表了一篇微博，此时如果他的一个关注者B用户访问到美国的机房，很可能无法看到用户A刚刚发表的微博。虽然跨城异地也会有此类同步延时问题，但正常情况下几十毫秒的延时对用户来说基本无感知的；而延时达到几秒钟就感觉比较明显了。

因此，跨国异地的“多活”，和跨城异地的“多活”，实际的含义并不完全一致。跨国异地多活的主要应用场景一般有这几种情况：

为不同地区用户提供服务

例如，亚马逊中国是为中国用户服务的，而亚马逊美国是为美国用户服务的，亚马逊中国的用户如果访问美国亚马逊，是无法用亚马逊中国的账号登录美国亚马逊的。

只读类业务做多活

例如，谷歌的搜索业务，由于用户搜索资料时，这些资料都已经存在于谷歌的搜索引擎上面，无论是访问英国谷歌，还是访问美国谷歌，搜索结果基本相同，并且对用户来说，也不需要搜索到最新的实时资料，跨国异地的几秒钟网络延迟，对搜索结果是没有什么影响的。

#### 跨城架构设计技巧

每个架构的关键点：

同城异区

关键在于搭建高速网络将两个机房连接起来，达到近似一个本地机房的效果。架构设计上可以将两个机房当作本地机房来设计，无须额外考虑。

跨城异地

关键在于数据不一致的情况下，业务不受影响或者影响很小，这从逻辑的角度上来说其实是矛盾的，架构设计的主要目的就是为了解决这个矛盾。

跨国异地

主要是面向不同地区用户提供业务，或者提供只读业务，对架构设计要求不高。

基于这个分析，跨城异地多活是架构设计复杂度最高的一种，接下来将介绍跨城异地多活架构设计的一些技巧

##### 技巧1：保证核心业务的异地多活

“异地多活”是为了保证业务的高可用，但很多架构师在考虑这个“业务”时，会不自觉地陷入一个思维误区：我要保证所有业务都能“异地多活”！

假设我们需要做一个“用户子系统”，这个子系统负责“注册”“登录”“用户信息”三个业务。为了支持海量用户，我们设计了一个“用户分区”的架构，即正常情况下用户属于某个主分区，每个分区都有其他数据的备份，用户用邮箱或者手机号注册，路由层拿到邮箱或者手机号后，通过Hash计算属于哪个中心，然后请求对应的业务中心。基本的架构如下：

这样一个系统，如果3个业务要同时实现异地多活，会发现这些难以解决的问题：

注册问题

A中心注册了用户，数据还未同步到B中心，此时A中心宕机，为了支持注册业务多活，可以挑选B中心让用户去重新注册。看起来很容易就支持多活了，但仔细思考一下会发现这样做会有问题：一个手机号只能注册一个账号，A中心的数据没有同步过来，B中心无法判断这个手机号是否重复，如果B中心让用户注册，后来A中心恢复了，发现数据有冲突，怎么解决？实际上是无法解决的，因为同一个手机号注册的账号不能以后一次注册为准；而如果B中心不支持本来属于A中心的业务进行注册，注册业务的多活又成了空谈。

如果我们修改业务规则，允许一个手机号注册多个账号不就可以了吗？

这样做是不可行的，类似一个手机号只能注册一个账号这种规则，是核心业务规则，修改核心业务规则的代价非常大，几乎所有的业务都要重新设计，为了架构设计去改变业务规则（而且是这么核心的业务规则）是得不偿失的。

用户信息问题

用户信息的修改和注册有类似的问题，即A、B两个中心在异常的情况下都修改了用户信息，如何处理冲突？

由于用户信息并没有账号那么关键，一种简单的处理方式是按照时间合并，即最后修改的生效。业务逻辑上没问题，但实际操作也有一个很关键的“坑”：怎么保证多个中心所有机器时间绝对一致？在异地多中心的网络下，这个是无法保证的，即使有时间同步也无法完全保证，只要两个中心的时间误差超过1秒，数据就可能出现混乱，即先修改的反而生效。

还有一种方式是生成全局唯一递增ID，这个方案的成本很高，因为这个全局唯一递增ID的系统本身又要考虑异地多活，同样涉及数据一致性和冲突的问题。

综合上面的简单分析可以发现，如果“注册”“登录”“用户信息”全部都要支持异地多活，实际上是挺难的，有的问题甚至是无解的。那这种情况下我们应该如何考虑“异地多活”的架构设计呢？答案其实很简单：优先实现核心业务的异地多活架构！

对于这个模拟案例来说，“登录”才是最核心的业务，“注册”和“用户信息”虽然也是主要业务，但并不一定要实现异地多活，主要原因在于业务影响不同。对于一个日活1000万的业务来说，每天注册用户可能是几万，修改用户信息的可能还不到1万，但登录用户是1000万，很明显我们应该保证登录的异地多活。

对于新用户来说，注册不了的影响并不明显，因为他还没有真正开始使用业务。用户信息修改也类似，暂时修改不了用户信息，对于其业务不会有很大影响。而如果有几百万用户登录不了，就相当于几百万用户无法使用业务，对业务的影响就非常大了：公司的客服热线很快就被打爆，微博、微信上到处都在传业务宕机，论坛里面到处是抱怨的用户，那就是互联网大事件了！

而登录实现“异地多活”恰恰是最简单的，因为每个中心都有所有用户的账号和密码信息，用户在哪个中心都可以登录。用户在A中心登录，A中心宕机后，用户到B中心重新登录即可。

如果某个用户在A中心修改了密码，此时数据还没有同步到B中心，用户到B中心登录是无法登录的，这个怎么处理？这个问题其实就涉及另外一个设计技巧了，稍后再谈。

##### 技巧2：保证核心数据最终一致性

异地多活本质上是通过异地的数据冗余，来保证在极端异常的情况下业务也能够正常提供给用户，因此数据同步是异地多活架构设计的核心。但大部分架构师在考虑数据同步方案时，会不知不觉地陷入完美主义误区：我要所有数据都实时同步！

数据冗余是要将数据从A地同步到B地，从业务的角度来看是越快越好，最好和本地机房一样的速度最好。但让人头疼的问题正在这里：异地多活理论上就不可能很快，因为这是物理定律决定的。

因此异地多活架构面临一个无法彻底解决的矛盾：业务上要求数据快速同步，物理上正好做不到数据快速同步，因此所有数据都实时同步，实际上是一个无法达到的目标。

既然是无法彻底解决的矛盾，那就只能想办法尽量减少影响。有几种方法可以参考：

尽量减少异地多活机房的距离，搭建高速网络

和同城异区架构类似，但搭建跨城异地的高速网络成本远远超过同城异区的高速网络，成本巨大，一般只有巨头公司才能承担。

尽量减少数据同步，只同步核心业务相关的数据

简单来说就是不重要的数据不同步，同步后没用的数据不同步，只同步核心业务相关的数据。

以前面的“用户子系统”为例，用户登录所产生的token或者session信息，数据量很大，但其实并不需要同步到其他业务中心，因为这些数据丢失后重新登录就可以再次获取了。

这时你可能会想到：这些数据丢失后要求用户重新登录，影响用户体验！

确实如此，毕竟需要用户重新输入账户和密码信息，或者至少要弹出登录界面让用户点击一次，但相比为了同步所有数据带来的代价，这个影响完全可以接受。为什么这么说呢，还是卖个关子我会在后面分析。

保证最终一致性，不保证实时一致性

最终一致性就是BASE理论，即业务不依赖数据同步的实时性，只要数据最终能一致即可。例如，A机房注册了一个用户，业务上不要求能够在50毫秒内就同步到所有机房，正常情况下要求5分钟同步到所有机房即可，异常情况下甚至可以允许1小时或者1天后能够一致。

最终一致性在具体实现时，还需要根据不同的数据特征，进行差异化的处理，以满足业务需要。例如，对“账号”信息来说，如果在A机房新注册的用户5分钟内正好跑到B机房了，此时B机房还没有这个用户的信息，为了保证业务的正确，B机房就需要根据路由规则到A机房请求数据。

而对“用户信息”来说，5分钟后同步也没有问题，也不需要采取其他措施来弥补，但还是会影响用户体验，即用户看到了旧的用户信息，这个问题怎么解决呢？好像又是一个解决不了的问题，在最后会给出答案。

##### 技巧3：采用多种手段同步数据

数据同步是异地多活架构设计的核心，幸运的是基本上存储系统本身都会有同步的功能。例如，MySQL的主备复制、Redis的Cluster功能、Elasticsearch的集群功能。这些系统本身的同步功能已经比较强大，能够直接拿来就用，但这也无形中将我们引入了一个思维误区：只使用存储系统的同步功能！

既然说存储系统本身就有同步功能，而且同步功能还很强大，为何说只使用存储系统是一个思维误区呢？因为虽然绝大部分场景下，存储系统本身的同步功能基本上也够用了，但在某些比较极端的情况下，存储系统本身的同步功能可能难以满足业务需求。

以MySQL为例，MySQL 5.1版本的复制是单线程的复制，在网络抖动或者大量数据同步时，经常发生延迟较长的问题，短则延迟十几秒，长则可能达到十几分钟。而且即使我们通过监控的手段知道了MySQL同步时延较长，也难以采取什么措施，只能干等。

Redis又是另外一个问题，Redis 3.0之前没有Cluster功能，只有主从复制功能，而为了设计上的简单，Redis 2.8之前的版本，主从复制有一个比较大的隐患：从机宕机或者和主机断开连接都需要重新连接主机，重新连接主机都会触发全量的主从复制。这时主机会生成内存快照，主机依然可以对外提供服务，但是作为读的从机，就无法提供对外服务了，如果数据量大，恢复的时间会相当长。

综合上面的案例可以看出，存储系统本身自带的同步功能，在某些场景下是无法满足业务需要的。尤其是异地多机房这种部署，各种各样的异常情况都可能出现，当我们只考虑存储系统本身的同步功能时，就会发现无法做到真正的异地多活。

解决的方案就是拓开思路，避免只使用存储系统的同步功能，可以将多种手段配合存储系统的同步来使用，甚至可以不采用存储系统的同步方案，改用自己的同步方案。

还是以前面的“用户子系统”为例，我们可以采用如下几种方式同步数据：

消息队列方式

对于账号数据，由于账号只会创建，不会修改和删除（假设我们不提供删除功能），我们可以将账号数据通过消息队列同步到其他业务中心。

二次读取方式

某些情况下可能出现消息队列同步也延迟了，用户在A中心注册，然后访问B中心的业务，此时B中心本地拿不到用户的账号数据。为了解决这个问题，B中心在读取本地数据失败时，可以根据路由规则，再去A中心访问一次（这就是所谓的二次读取，第一次读取本地，本地失败后第二次读取对端），这样就能够解决异常情况下同步延迟的问题。

存储系统同步方式

对于密码数据，由于用户改密码频率较低，而且用户不可能在1秒内连续改多次密码，所以通过数据库的同步机制将数据复制到其他业务中心即可，用户信息数据和密码类似。

回源读取方式

对于登录的session数据，由于数据量很大，我们可以不同步数据；但当用户在A中心登录后，然后又在B中心登录，B中心拿到用户上传的session id后，根据路由判断session属于A中心，直接去A中心请求session数据即可；反之亦然，A中心也可以到B中心去获取session数据。

重新生成数据方式

对于“回源读取”场景，如果异常情况下，A中心宕机了，B中心请求session数据失败，此时就只能登录失败，让用户重新在B中心登录，生成新的session数据。

注意：以上方案仅仅是示意，实际的设计方案要比这个复杂一些，还有很多细节要考虑。

综合上述的各种措施，最后“用户子系统”同步方式整体如下：

##### 技巧4：只保证绝大部分用户的异地多活

前面在给出每个思维误区对应的解决方案时，留下了几个小尾巴：某些场景下我们无法保证100%的业务可用性，总是会有一定的损失。例如，密码不同步导致无法登录、用户信息不同步导致用户看到旧的信息等，这个问题怎么解决呢？

其实这个问题涉及异地多活架构设计中一个典型的思维误区：我要保证业务100%可用！但极端情况下就是会丢一部分数据，就是会有一部分数据不能同步，有没有什么巧妙能做到100%可用呢？

很遗憾，答案是没有！异地多活也无法保证100%的业务可用，这是由物理规律决定的，光速和网络的传播速度、硬盘的读写速度、极端异常情况的不可控等，都是无法100%解决的。所以针对这个思维误区，我的答案是“忍”！也就是说我们要忍受这一小部分用户或者业务上的损失，否则本来想为了保证最后的0.01%的用户的可用性，做一个完美方案，结果却发现99.99%的用户都保证不了了。

对于某些实时强一致性的业务，实际上受影响的用户会更多，甚至可能达到1/3的用户。以银行转账这个业务为例，假设小明在北京XX银行开了账号，如果小明要转账，一定要北京的银行业务中心才可用，否则就不允许小明自己转账。如果不这样的话，假设在北京和上海两个业务中心实现了实时转账的异地多活，某些异常情况下就可能出现小明只有1万元存款，他在北京转给了张三1万元，然后又到上海转给了李四1万元，两次转账都成功了。这种漏洞如果被人利用，后果不堪设想。

当然，针对银行转账这个业务，虽然无法做到“实时转账”的异地多活，但可以通过特殊的业务手段让转账业务也能实现异地多活。例如，转账业务除了“实时转账”外，还提供“转账申请”业务，即小明在上海业务中心提交转账请求，但上海的业务中心并不立即转账，而是记录这个转账请求，然后后台异步发起真正的转账操作，如果此时北京业务中心不可用，转账请求就可以继续等待重试；假设等待2个小时后北京业务中心恢复了，此时上海业务中心去请求转账，发现余额不够，这个转账请求就失败了。小明再登录上来就会看到转账申请失败，原因是“余额不足”。

不过需要注意的是“转账申请”的这种方式虽然有助于实现异地多活，但其实还是牺牲了用户体验的，对于小明来说，本来一次操作的事情，需要分为两次：一次提交转账申请，另外一次是要确认是否转账成功。

虽然我们无法做到100%可用性，但并不意味着我们什么都不能做，为了让用户心里更好受一些，我们可以采取一些措施进行安抚或者补偿，例如：

挂公告

说明现在有问题和基本的问题原因，如果不明确原因或者不方便说出原因，可以发布“技术哥哥正在紧急处理”这类比较轻松和有趣的公告。

事后对用户进行补偿

例如，送一些业务上可用的代金券、小礼包等，减少用户的抱怨。

补充体验

对于为了做异地多活而带来的体验损失，可以想一些方法减少或者规避。以“转账申请”为例，为了让用户不用确认转账申请是否成功，我们可以在转账成功或者失败后直接给用户发个短信，告诉他转账结果，这样用户就不用时不时地登录系统来确认转账是否成功了。

核心思想

异地多活设计的理念可以总结为一句话：采用多种手段，保证绝大部分用户的核心业务异地多活！

跨城架构设计步骤

第1步：业务分级

按照一定的标准将业务进行分级，挑选出核心的业务，只为核心业务设计异地多活，降低方案整体复杂度和实现成本。

常见的分级标准有下面几种：

访问量大的业务

以用户管理系统为例，业务包括登录、注册、用户信息管理，其中登录的访问量肯定是最大的。

核心业务

以QQ为例，QQ的主场景是聊天，QQ空间虽然也是重要业务，但和聊天相比，重要性就会低一些，如果要从聊天和QQ空间两个业务里面挑选一个做异地多活，那明显聊天要更重要（当然，此类公司如腾讯，应该是两个都实现了异地多活的）。

产生大量收入的业务

同样以QQ为例，聊天可能很难为腾讯带来收益，因为聊天没法插入广告；而QQ空间反而可能带来更多收益，因为QQ空间可以插入很多广告，因此如果从收入的角度来看，QQ空间做异地多活的优先级反而高于QQ聊天了。

以我们一直在举例的用户管理系统为例，“登录”业务符合“访问量大的业务”和“核心业务”这两条标准，因此我们将登录业务作为核心业务。

第2步：数据分类

挑选出核心业务后，需要对核心业务相关的数据进一步分析，目的在于识别所有的数据及数据特征，这些数据特征会影响后面的方案设计。

常见的数据特征分析维度有：

数据量

这里的数据量包括总的数据量和新增、修改、删除的量。对异地多活架构来说，新增、修改、删除的数据就是可能要同步的数据，数据量越大，同步延迟的几率越高，同步方案需要考虑相应的解决方案。

唯一性

唯一性指数据是否要求多个异地机房产生的同类数据必须保证唯一。例如用户ID，如果两个机房的两个不同用户注册后生成了一样的用户ID，这样业务上就出错了。

数据的唯一性影响业务的多活设计，如果数据不需要唯一，那就说明两个地方都产生同类数据是可能的；如果数据要求必须唯一，要么只能一个中心点产生数据，要么需要设计一个数据唯一生成的算法。

实时性

实时性指如果在A机房修改了数据，要求多长时间必须同步到B机房，实时性要求越高，对同步的要求越高，方案越复杂。

可丢失性

可丢失性指数据是否可以丢失。例如，写入A机房的数据还没有同步到B机房，此时A机房机器宕机会导致数据丢失，那这部分丢失的数据是否对业务会产生重大影响。

例如，登录过程中产生的session数据就是可丢失的，因为用户只要重新登录就可以生成新的session；而用户ID数据是不可丢失的，丢失后用户就会失去所有和用户ID相关的数据，例如用户的好友、用户的钱等。

可恢复性

可恢复性指数据丢失后，是否可以通过某种手段进行恢复，如果数据可以恢复，至少说明对业务的影响不会那么大，这样可以相应地降低异地多活架构设计的复杂度。

例如，用户的微博丢失后，用户重新发一篇一模一样的微博，这个就是可恢复的；或者用户密码丢失，用户可以通过找回密码来重新设置一个新密码，这也算是可以恢复的；而用户账号如果丢失，用户无法登录系统，系统也无法通过其他途径来恢复这个账号，这就是不可恢复的数据。

我们同样以用户管理系统的登录业务为例，简单分析如下表所示。

第3步：数据同步

确定数据的特点后，我们可以根据不同的数据设计不同的同步方案。常见的数据同步方案有：

存储系统同步

这是最常用也是最简单的同步方式。例如，使用MySQL的数据主从数据同步、主主数据同步。

这类数据同步的优点是使用简单，因为几乎主流的存储系统都会有自己的同步方案；缺点是这类同步方案都是通用的，无法针对业务数据特点做定制化的控制。例如，无论需要同步的数据量有多大，MySQL都只有一个同步通道。因为要保证事务性，一旦数据量比较大，或者网络有延迟，则同步延迟就会比较严重。

消息队列同步

采用独立消息队列进行数据同步，常见的消息队列有Kafka、ActiveMQ、RocketMQ等。

消息队列同步适合无事务性或者无时序性要求的数据。例如，用户账号，两个用户先后注册了账号A和B，如果同步时先把B同步到异地机房，再同步A到异地机房，业务上是没有问题的。而如果是用户密码，用户先改了密码为m，然后改了密码为n，同步时必须先保证同步m到异地机房，再同步n到异地机房；如果反过来，同步后用户的密码就不对了。因此，对于新注册的用户账号，我们可以采用消息队列同步了；而对于用户密码，就不能采用消息队列同步了（也可以使用加一个时间数据块的方法实现，但过于复杂）。

重复生成

数据不同步到异地机房，每个机房都可以生成数据，这个方案适合于可以重复生成的数据。例如，登录产生的cookie、session数据、缓存数据等。

我们同样以用户管理系统的登录业务为例，针对不同的数据特点设计不同的同步方案，如下表所示。

第4步：异常处理

无论数据同步方案如何设计，一旦出现极端异常的情况，总是会有部分数据出现异常的。例如，同步延迟、数据丢失、数据不一致等。异常处理就是假设在出现这些问题时，系统将采取什么措施来应对。异常处理主要有以下几个目的：

问题发生时，避免少量数据异常导致整体业务不可用。

问题恢复后，将异常的数据进行修正。

对用户进行安抚，弥补用户损失。

常见的异常处理措施有这几类：

1. 多通道同步

多通道同步的含义是采取多种方式来进行数据同步，其中某条通道故障的情况下，系统可以通过其他方式来进行同步，这种方式可以应对同步通道处故障的情况。

以用户管理系统中的用户账号数据为例，我们的设计方案一开始挑选了消息队列的方式进行同步，考虑异常情况下，消息队列同步通道可能中断，也可能延迟很严重；为了保证新注册账号能够快速同步到异地机房，我们再增加一种MySQL同步这种方式作为备份。这样针对用户账号数据同步，系统就有两种同步方式：MySQL主从同步和消息队列同步。除非两个通道同时故障，否则用户账号数据在其中一个通道异常的情况下，能够通过另外一个通道继续同步到异地机房，如下图所示。

多通道同步设计的方案关键点有：

一般情况下，采取两通道即可，采取更多通道理论上能够降低风险，但付出的成本也会增加很多。

数据库同步通道和消息队列同步通道不能采用相同的网络连接，否则一旦网络故障，两个通道都同时故障；可以一个走公网连接，一个走内网连接。

需要数据是可以重复覆盖的，即无论哪个通道先到哪个通道后到，最终结果是一样的。例如，新建账号数据就符合这个标准，而密码数据则不符合这个标准。

2. 同步和访问结合

这里的访问指异地机房通过系统的接口来进行数据访问。例如业务部署在异地两个机房A和B，B机房的业务系统通过接口来访问A机房的系统获取账号信息，如下图所示。

同步和访问结合方案的设计关键点有：

接口访问通道和数据库同步通道不能采用相同的网络连接，不能让数据库同步和接口访问都走同一条网络通道，可以采用接口访问走公网连接，数据库同步走内网连接这种方式。

数据有路由规则，可以根据数据来推断应该访问哪个机房的接口来读取数据。例如，有3个机房A、B、C，B机房拿到一个不属于B机房的数据后，需要根据路由规则判断是访问A机房接口，还是访问C机房接口。

由于有同步通道，优先读取本地数据，本地数据无法读取到再通过接口去访问，这样可以大大降低跨机房的异地接口访问数量，适合于实时性要求非常高的数据。

3. 日志记录

日志记录主要用于用户故障恢复后对数据进行恢复，其主要方式是每个关键操作前后都记录相关一条日志，然后将日志保存在一个独立的地方，当故障恢复后，拿出日志跟数据进行对比，对数据进行修复。

为了应对不同级别的故障，日志保存的要求也不一样，常见的日志保存方式有：

服务器上保存日志，数据库中保存数据，这种方式可以应对单台数据库服务器故障或者宕机的情况。

本地独立系统保存日志，这种方式可以应对某业务服务器和数据库同时宕机的情况。例如，服务器和数据库部署在同一个机架，或者同一个电源线路上，就会出现服务器和数据库同时宕机的情况。

日志异地保存，这种方式可以应对机房宕机的情况。

上面不同的日志保存方式，应对的故障越严重，方案本身的复杂度和成本就会越高，实际选择时需要综合考虑成本和收益情况。

4. 用户补偿

无论采用什么样的异常处理措施，都只能最大限度地降低受到影响的范围和程度，无法完全做到没有任何影响。例如，双同步通道有可能同时出现故障、日志记录方案本身日志也可能丢失。因此，无论多么完美的方案，故障的场景下总是可能有一小部分用户业务上出问题，系统无法弥补这部分用户的损失。但我们可以采用人工的方式对用户进行补偿，弥补用户损失，培养用户的忠诚度。简单来说，系统的方案是为了保证99.99%的用户在故障的场景下业务不受影响，人工的补偿是为了弥补0.01%的用户的损失。

常见的补偿措施有送用户代金券、礼包、礼品、红包等，有时为了赢得用户口碑，付出的成本可能还会比较大，但综合最终的收益来看还是很值得的。例如暴雪《炉石传说》2017年回档故障，暴雪给每个用户大约价值人民币200元的补偿，结果玩家都求暴雪再来一次回档，形象地说明了玩家对暴雪补偿的充分认可。

### 分布式协议

基于分布式协议的高可用方案，常见的有Galera Cluster,PXC和MGR

### 基于共享存储方案

如SAN存储，这种方案可以实现网络中不同服务器的数据共享，共享存储能够为数据库服务器和存储解耦。

### 基于磁盘复制方案

如DRBD，DRDB是一个以linux内核模块方式实现的块级别同步复制技术。它通过网卡将主服务器的每个块复制到另外一个服务器块设备上，并在主设备提交块之前记录下来。类似共享存储解决方案。

## 总结

MySQL高可用方案的建议，这些也是基于一些高可用的实践所做的总结。

1) 行业内多活的设计目标不是多写，需要先实现跨机房的高可用容灾和计划内的机房间数据切换

2) 引入consul的域名管理，解决VIP方案带来的一些潜在瓶颈（域名的业务属性，实现单机多实例，读写分离的域名配置）

3) 对于consul的整体定位不局限于集群环境，在单实例，集群，分布式中间件方向都可以采用，consul是作为一种通用的基础域名服务。

4) 同机房高可用方案的落地，需要和应用方对接程序端对域名的支持情况，在不同语言的客户端侧会有一些配置的差异。

5) 在已有高可用方案MHA基础上平滑过渡和改进，在后续新业务尝试引入MGR的方案

6) consul 业务API的开发，对数据库层面的业务可持续性访问（服务注销，服务发现）做一些补充和定制，保证consul服务的技术可控

7) 异机房高可用实现应用无缝切换，计划内切换，会有业务中断/延时，保证可控的基础上，应用端无须修改连接配置，需要测试DNS的域名转发等策略，计划外切换，需要做确认才可完成。

8) 对已有的分布式方案，可以采用MGR,中间件，consul的组合方案，实现读写分布式扩展。

# 优秀解决方案

目前常用的高可用方案，自底层而上包括：

1. 数据库层面——主从复制，MySQL采用这种方式

金融领域很少采用，因为这种模式将主节点和从节点以及主从节点之间的网络环境紧紧地绑在一起，主节点的稳定性将不再由他们自己决定，而要同时看从节点和网络环境：一旦从节点或者网络环境抖动，主节点的性能就会受到直接影响。如果主节点和从节点之间是跨机房设置跨城市部署，发生这种概率的机会更大，影响也会更加显著。从某种程度上讲，和单节点模式相比，这种模式下主节点的稳定性不但没有增加，反而是降低了。

由于“主从复制”模式中缺少第三方仲裁者的角色，当主从节点之间的心跳信号异常时，从节点无法靠自己判断到底是主点故障了，还是主从之间网络故障了。此时，如果从节点认为是主节点故障而将自己自动切换为主节点，就极容易导致“双主”、“脑裂（brain split）”的局面，对用户来说这是绝对无法接受的结果。所以，数据库“主从复制”技术从来不会提供“从节点自动切换为主节点”的功能，一定要由人来确认主节点确实故障了，并手动发起从节点的切主动作，这就大大增加了系统恢复的时间（RTO）。

1. 底层硬件层面

在主机层面用HACMP技术以应对主机故障，或者在存储层面采取复制技术（比如FlashCopy）未提交数据冗余等，会使灾难切换方案变得很复杂，并且会有相对较长的故障恢复时间（RTO），所以通常不是数据库用户的首选。

1. 备份软件

支持异种数据库之间相互复制数据的产品，比如IBM CDC和Oracle Golden Gate（OGG）。这些产品的特点是比较灵活，可以支持异种数据库之间的数据复制，也可以指定只复制数据库中的部分对象（比如只复制指定几张数据表的数据）。但这些产品的缺点也很明显：首先相对于数据库主从复制来说，时延较大，通常会达到秒级以上，而且往往做不到数据库层面100%的完全复制。因此，这种方式通常作为不同数据库产品之间做数据“准”实时同步的手段，而不会作为数据库产品实现高可用及容灾的手段。

1. 集群层面

分布式多副本数据一致性技术，通常是基于Paxos协议或者Raft协议来实现。这种技术会将数据保存在多份副本上，每一次对数据的修改操作都会强同步到多数副本上，在保证了数据冗余的同时，不再像“主从复制”技术那样依赖某个数据节点的稳定性，从而消除了传统主从复制技术下从节点给主节点带来的危险。同时，在主节点故障的情况下，其余节点会自动选举出新的主节点以实现高可用（个别从节点故障则完全不影响服务），整个过程非常快速且完全无需人为干预。因此这种技术不仅能保证RPO=0，而且大大减小了RTO，相比传统主从复制技术，可以提供更强大的高可用能力。

此外，为了抵御机房级灾难和城市级灾难，可以将多份副本分散部署在多个机房里甚至多个城市里，以避免机房级灾难或者城市级灾难损失多数派副本。这样就具备了机房级和城市级容灾的能力，进一步加强了高可用的能力。

## 腾讯

## 阿里



# 应用

## 组建高可用

采用去中心化设计，整个集群无单点。

### 计算节点高可用

计算节点是无状态节点，一般要求为每个业务至少配置2个计算节点。

为降低计算节点异常对业务的影响，可以采用的措施：

1. 监控节点健康状况；
2. 在某些节点发生故障时，实现故障的接管；
3. 异常事务清理

### 数据节点高可用

为了提升数据服务的可靠性和可用性，安全组Group一般由多个数据节点组成。其中一个数据节点为主节点，提供读写服务，其他节点为备节点，提供读服务，在一个安全组中可以设置多个副本。

通过数据节点主备间通过数据库的复制技术来进行主备机之间的数据同步，主机至多个备机之间为星型复制模式，即主机直接向多个备机进行数据同步，具体方式目前支持半同步和异步两种。

为了权衡高可用和高性能，一般有以下几种策略：

**最大性能策略**

底层的数据节点提供最大的写性能，副本间采用异步赋值，即一旦日志数据写到主节点，事务即可提交。

**最大保护策略**

日志数据必须同时写出到主用数据节点和至少一个备用数据节点，事务才被提交。

**最大可用策略**

### 全局事务管理节点高可用

全局事务管理需要支持双机热备部署，主备双活，实现故障时秒级切换。

### 管理节点高可用

管理节点采用双机冷备的模式，可以实现秒级切换。

## 容灾

### 衡量指标

在容灾恢复方面，目前业界公认有三个目标值。

1. 恢复时间：企业能够忍受多长时间没有IT，处于停业状态；
2. 网络多长时间能够恢复；
3. 业务层面的恢复。

整个恢复过程中，最关键的衡量指标有两个：一个是RTO，一个是RPO。

所谓RTO，Recovery Time Object，它是指灾难发生后，从IT系统当前导致业务停顿开始，到IT系统恢复至可以支持各部门运作、恢复运营之时，此两点之间的时间段称为RTO。

所谓RPO，Recovery Point Object，是指从系统和应用数据而言，要实现能够恢复至可以支持各部门业务运作，系统及生产数据应恢复到怎么样的更新程度。这种更新程度可以是上一周的备份数据，也可以是上一次交易的实时数据。

### 同城双活

### 异地多活