# 总结

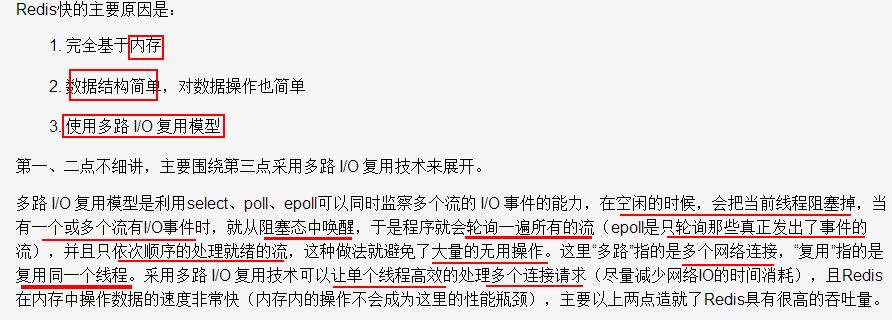
## Redis

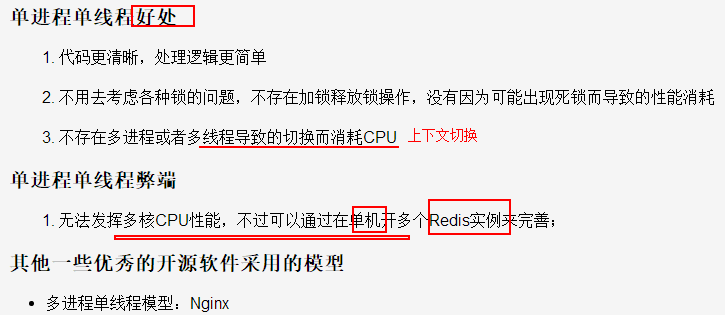
### 为什么用Redis

性能和并发（还可以说是单进程单线程的）

<https://mp.weixin.qq.com/s/mmDo7_ogqxtoxwfKpe9cYg> (java技术栈，为什么用redis)

### Redis单线程单进程





### Redis过期策略和内存淘汰机制

比如你 Redis 只能存 5G 数据，可是你写了 10G，那会删 5G 的数据。怎么删的，这个问题思考过么？

还有，你的数据已经设置了过期时间，但是时间到了，内存占用率还是比较高，有思考过原因么?

**回答：Redis 采用的是定期删除+惰性删除策略。**

**为什么不用定时删除策略**

定时删除，用一个定时器来负责监视 Key，过期则自动删除。虽然内存及时释放，但是十分消耗 CPU 资源。

在大并发请求下，CPU 要将时间应用在处理请求，而不是删除 Key，因此没有采用这一策略。

**定期删除+惰性删除是如何工作**

定期删除，Redis 默认每个 100ms 检查，是否有过期的 Key，有过期 Key 则删除。

需要说明的是，Redis 不是每个 100ms 将所有的 Key 检查一次，而是随机抽取进行检查(如果每隔 100ms，全部 Key 进行检查，Redis 岂不是卡死)。

因此，如果只采用定期删除策略，会导致很多 Key 到时间没有删除。于是，惰性删除派上用场。

也就是说在你获取某个 Key 的时候，Redis 会检查一下，这个 Key 如果设置了过期时间，那么是否过期了？如果过期了此时就会删除。

采用定期删除+惰性删除就没其他问题了么?

不是的，如果定期删除没删除 Key。然后你也没即时去请求 Key，也就是说惰性删除也没生效。这样，Redis的内存会越来越高。那么就应该采用内存淘汰机制。

在 redis.conf 中有一行配置：

# maxmemory-policy volatile-lru

该配置就是配内存淘汰策略的(什么，你没配过？好好反省一下自己)：

* **noeviction：**当内存不足以容纳新写入数据时，新写入操作会报错。应该没人用吧。
* **allkeys-lru：**当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 Key。推荐使用，目前项目在用这种。
* **allkeys-random：**当内存不足以容纳新写入数据时，在键空间中，随机移除某个 Key。应该也没人用吧，你不删最少使用 Key，去随机删。
* **volatile-lru：**当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的 Key。这种情况一般是把 Redis 既当缓存，又做持久化存储的时候才用。不推荐。
* **volatile-random：**当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个 Key。依然不推荐。
* **volatile-ttl：**当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的 Key 优先移除。不推荐。

### Redis 和数据库双写一致性问题

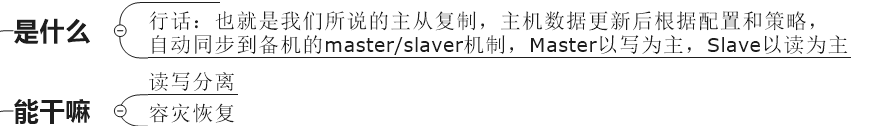
就是如果对数据有强一致性要求，不能放缓存。我们所做的一切，只能保证最终一致性。

首先，采取正确更新策略，先更新数据库，再删缓存。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

### 集群

为了分担些压力，将不同的数据放在不同的redis库中，每个Redis库搭建主从复制，在Redis3.0之前，没有将自动将数据按照一个算法取值，当值在某个区间时放在对应的库中，此时Java需要知道操作哪个库，便出现了代理中间件，比如tw…或者codis。在Redis3.0cluster出来后，即对key按某一算法取值，然后将数据放到对应的槽中。

### 主从复制



**主从复制的模式：**

1. **一主二仆模式**

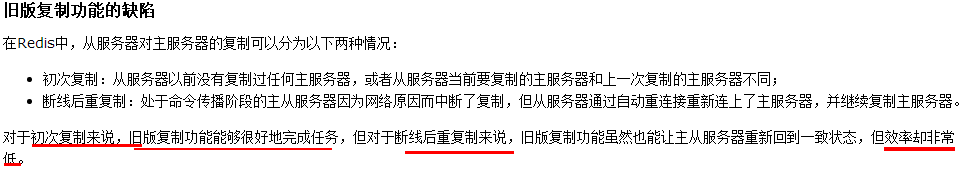
有一个主机作为master，负责写操作；两台主机作为slaver，只能负责读操作，不能有写操作。当一台从机加进来后，会向主机发送同步请求，此时主机进行备份（RDB），从机通过该备份文件进行数据同步，即会复制主机的所有数据（这个与Mysql主从复制不一样）。

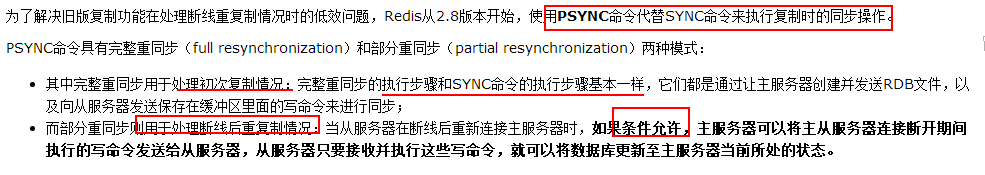
**细节：**

1. 主机宕机后，从机待命，一直等主机恢复，即从机不上位变为主机。
2. 主机恢复后，主机新增记录，从机可以复制到。
3. 从机宕机，不影响，当从机恢复后，不再是从机，需要手动再次声明为从机。
4. **从机从主机复制同步数据的原理**

从机联机后，向主机发送sync（同步）请求，主机收到请求后立即备份，发送RDB文件到从机，从机收到RDB文件后，执行文件完成复制操作。之后主机的写操作，都会立刻发送给从机，从机立即执行。

SYNC是一个如此消耗资源的命令





1. **薪火相传模式**

在复制原理中，我们发现主机不仅负责写操作，还负责将命令发送给它的每一个从机，比较消耗资源，所以，可以采用“链式操作”，即一台主机有一台从机（或者两台），然后该从机又是其它机器的主机。

这样可以分担主机的压力。对于既有主机角色又有从机角色的机器，只能负责读操作，没有写操作。

1. **哨兵模式**

一主二仆和薪火相传模式均为能解决当主机宕机，其它从机等待的问题。我们可以使用slave of no one 成为主机。但是为了自动化，引入一个哨兵，哨兵负责监控主机，会规律地发送ping命令测试主机的存活，若ping不通，此时当然一个哨兵不能下定论，因为有可能是ping的网络问题，所以可以在每个从机都安装哨兵，当有一个哨兵ping不通时，会通知其他哨兵执行ping操作，最后执行投票判决主机是否宕机。当主机宕机后，会投票选择从机上位成为主机（选取原则依次为：选取优先级靠前的；选取偏移量最大的，即数据量多的；选取runid最小的）。当从机重新成为主机后，哨兵发送slaveof命令给剩下的从机，并完成同步操作，当旧主机恢复后，哨兵发送slaveof命令，让其成为从机。

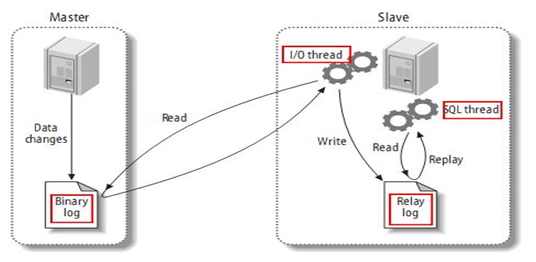
### 缓存穿透

## Mycat（mysql主从复制，分库分表）

### 主从复制

**Mysql主从复制：**

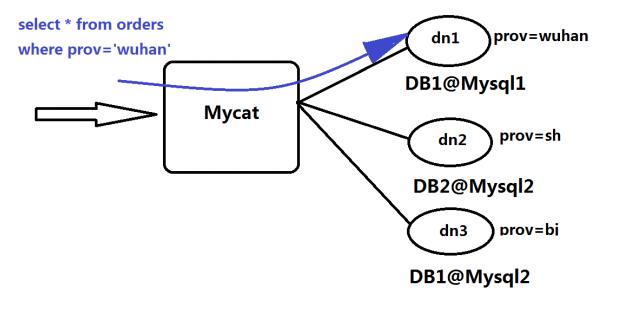
1. master将改变记录到二进制日志（binary log）。这些记录过程叫做二进制日志事件，binary log events；
2. slave将master的binary log events拷贝到它的中继日志（relay log）；
3. slave重做中继日志中的事件，将改变应用到自己的数据库中。 MySQL复制是异步的且串行化的



当实现了主从复制后，主机负责写操作，从机负责读操作，那么Java操作数据库时需要指定操作哪个库，这样很繁琐，因此引入中间件---Mycat。

Mycat屏蔽了客户端操作与数据库之间的直接关系，客户端操作时，只需要连接到Mycat即可。Mycat拦截到sql语句后，对sql做特定的分析，如分片，路由等等。然后操作真时的数据库，最后将结果返回给客户端。

配置主要是通过操作Mycat的schema.xml文件，见D:\JavaWeb\尚硅谷MySQL高级\SQL高级08&MyCat\Mycat.mmap



### 分库

同样是操作schema.xml文件。

分库原则：如果某个表访问量很大，规模很庞大，可以选择单独成库。但是当该表涉及到其他表的关联，可以使用冗余字段，冗余表等手段解决，具体分库策略还是得看业务。比如在订单表，订单详情表，订单类型表，用户表这四个表中，订单表有用户id和订单类型id，订单详情有订单id，即四表均关联，要想对这四个表分库，可以考虑将用户表为一库，与订单相关的表为一个库，因为一般查询订单时拿到用户id后才进行查询。

### 分表

类型：水平分表（按行分）；垂直分表（按列分）

考虑将大表拆分存储在不同的库中，具体拆分原则主要根据业务，拆分算法可以自定义（在rule.xml里定义）。

对于上述的订单表，用户表，订单详情表，订单类型表，现在用户表和订单相关表已经分库，考虑订单表太大，对订单表分表。

但是分表后当然要避免关联查询，所以可以针对用户id对订单表分表，这样可以保证同一个用户的所有订单在同一个表中。具体可以对用户id模二求余（则分两个表。那么模三求余就是三个表）。

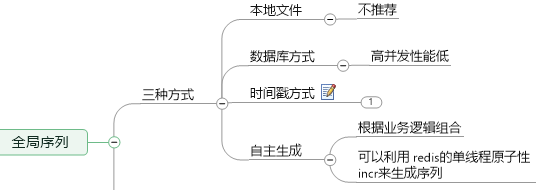
分好订单表后，订单详情表也是一个大表，因此也需要分表，但是订单详情表时关联订单表的，即订单详情表有订单id，因此不好使用某种算法针对某列来分表，Mycat提供了跨库关联，只需在schea.xml的订单表下添加：<childTable name="order\_detail" primaryKey="id" joinKey="order\_id" parentKey="id" />即可。那么对应订单表相关记录的订单详情表记录在插入时会进入该订单表的库。

E-R表的原理：

根据配置的关联id，找到主表所在的库，然后将从表插入到该库中。**疑问**：查找过程中的性能是否损耗较大？是否可以使用在订单详情表里添加冗余字段用户id，然后使用与订单表的相同策略进行分表？

最后还有一个订单类型表，这个表规模不大，与订单表有关联，所以把该表可以作为全局表。

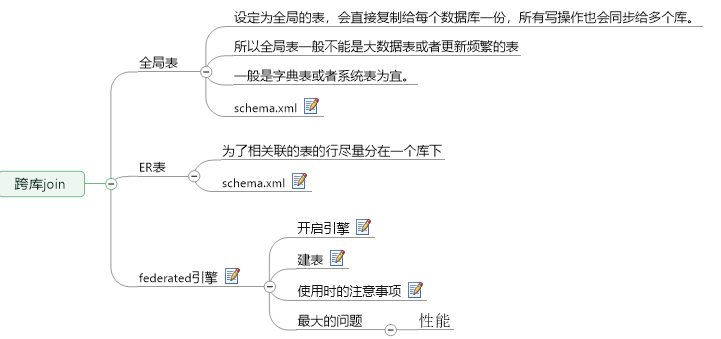
最后，为了避免不同库的相同表（分表）的主键相同，我们采用全局序列的时间戳方式来避免。



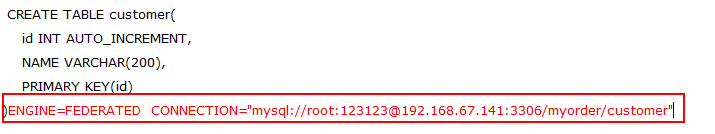
uuid的顺序不统一，即不是增加的或减少的，但时间戳是增长的，所以uuid会影响性能，不能使用。

时间戳操作语句：insert into `orders`(id,amount,customer\_id,order\_type) values(next value for MYCATSEQ\_GLOBAL,1000,103,10); （大小要是18位（BigInt类型））。

### 跨库关联表

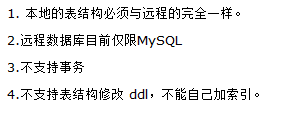


全局表和ER表（主从表）在上面已经说了，现在假如硬性要求不同库的两个表进行关联，直接将表复制多一份到另一个表有点伤，则可以通过**虚表**来解决问题。具体需要开启federated引擎，然后建表：

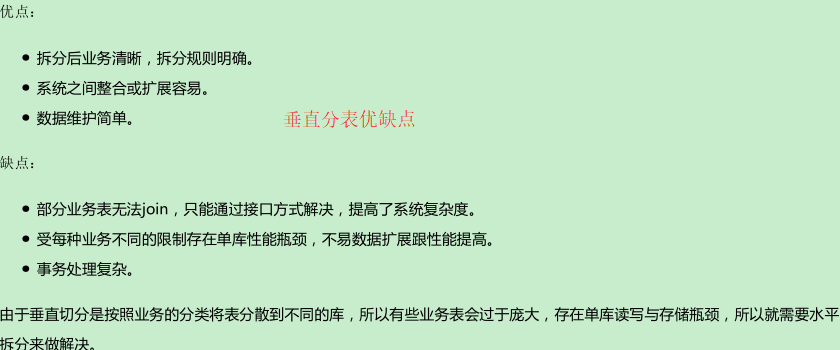


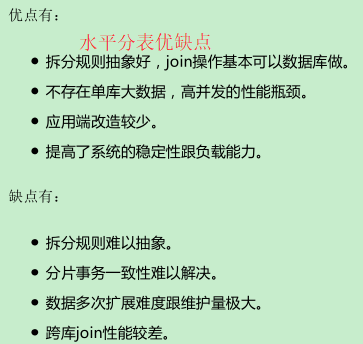
即表结构我有，但数据从远端来，但是表的结构必须一样，不能更改。

注意事项：

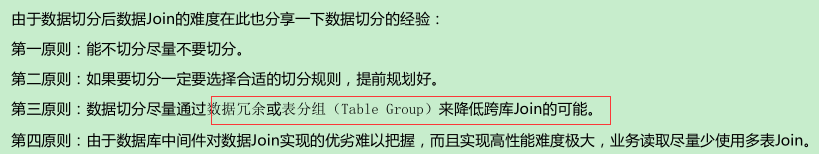


### 水平分表与垂直分表





### 数据库切分原则



## Zookeeper

### 分布式环境的各种问题

分布式的特点为：分布性，对等性（无主从之分），并发性，缺乏全局时钟，故障总是发生。

分布式的问题： 通信异常，网络分区（脑裂），三态（成功，超时，失败），节点故障。具体参考《从pasox到zookeeper》P21。

### 分布式事务

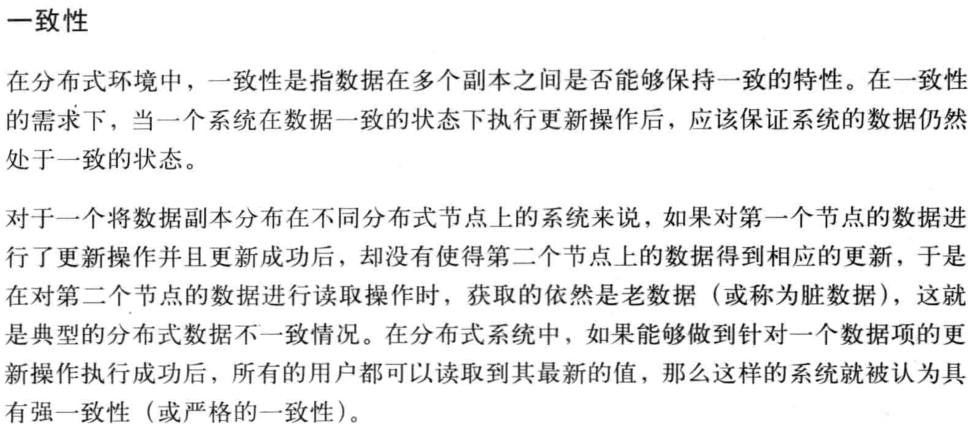
单机事务：ACID（原子性，一致性，隔离性，持久性）

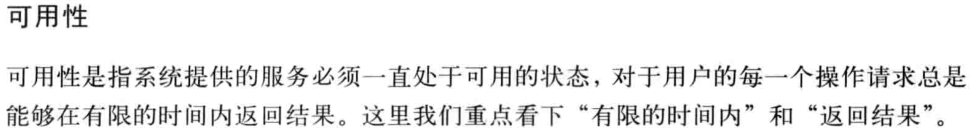
分布式事务（CAP理论）

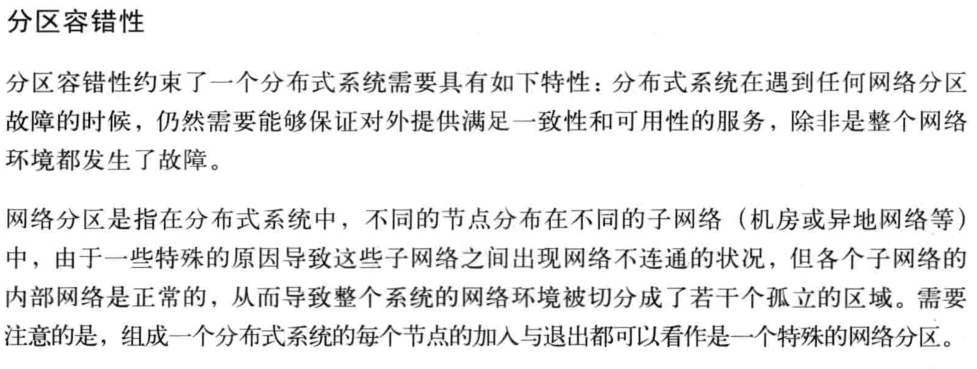


CAP理论：

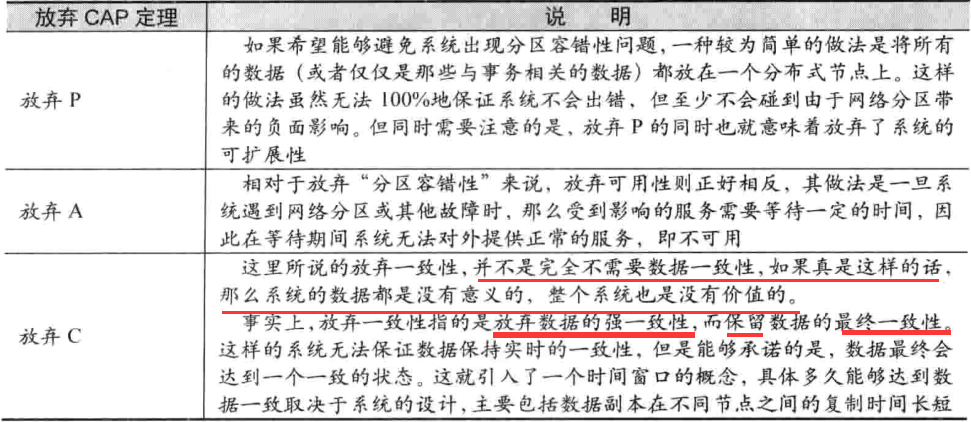
C：一致性，A：可用性，P：集群容错性；一个分布式系统，最多只能同时满足其中两项，不可能同时满足三项。







CAP理论的权衡选择:

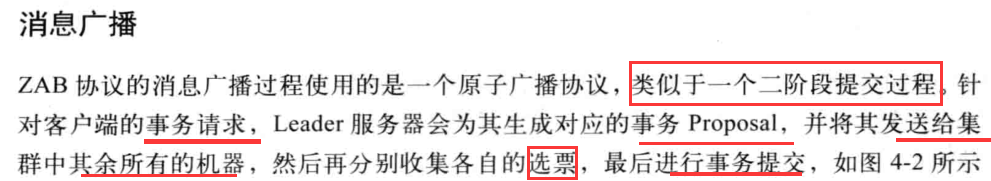


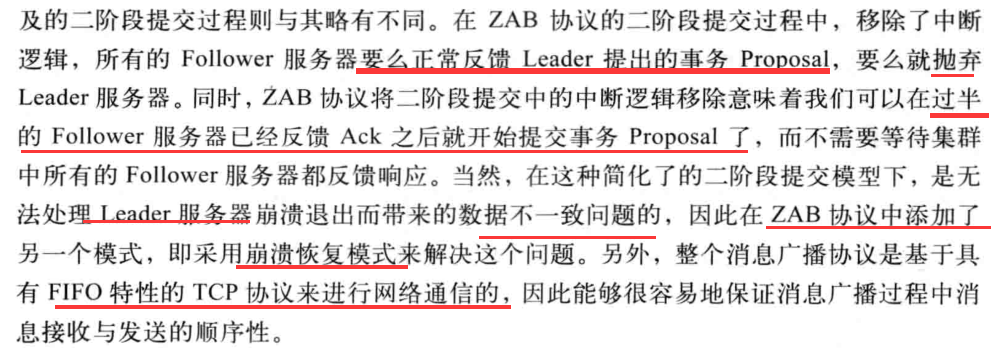
<https://www.cnblogs.com/jiangyu666/p/8522547.html>

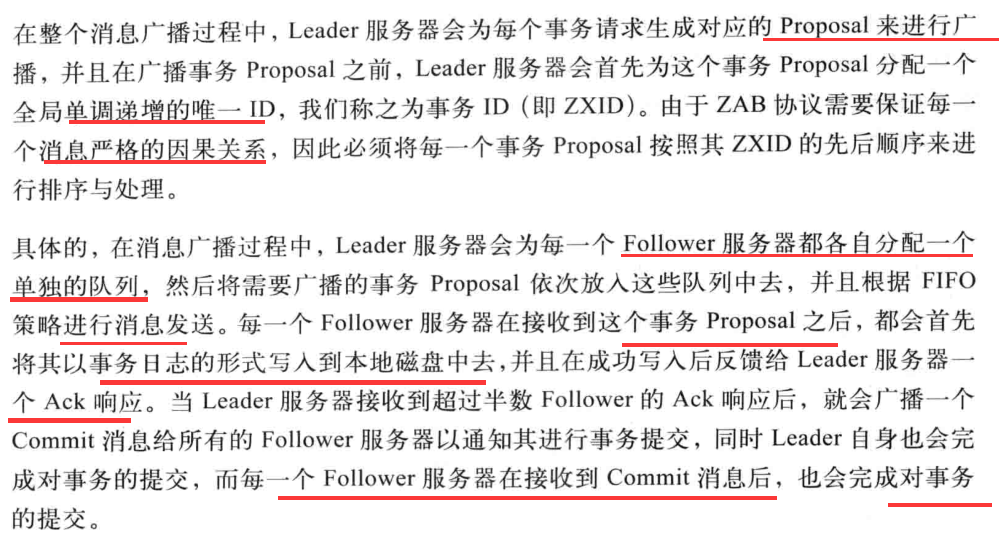
### 3、ZAB协议

ZAB协议分为**崩溃恢复**和**消息广播模式**，当集群中的leader宕机时，进入崩溃恢复模式，当选举出leader，并且集群中有过半的follower与leader完成同步，进入消息广播模式。

消息广播模式类似一个二阶段提交过程，但也有不同。

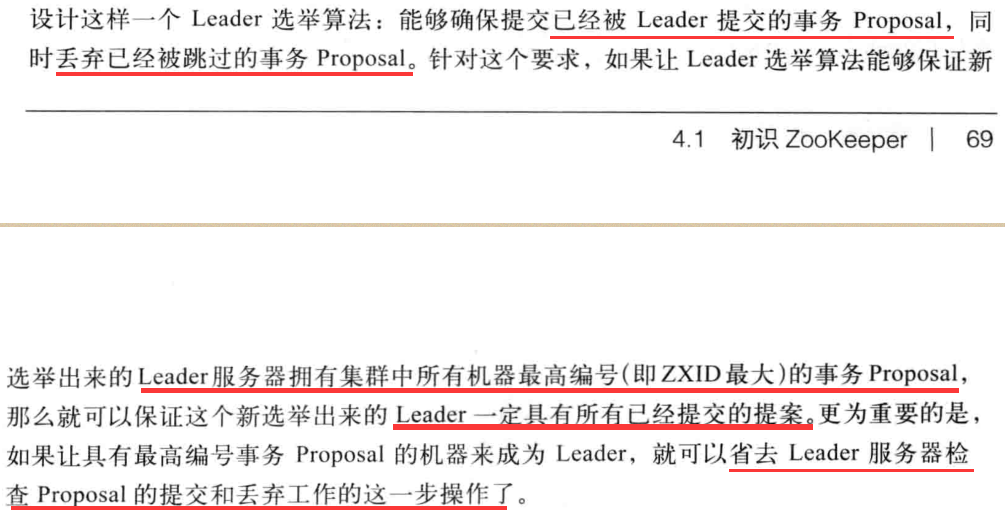


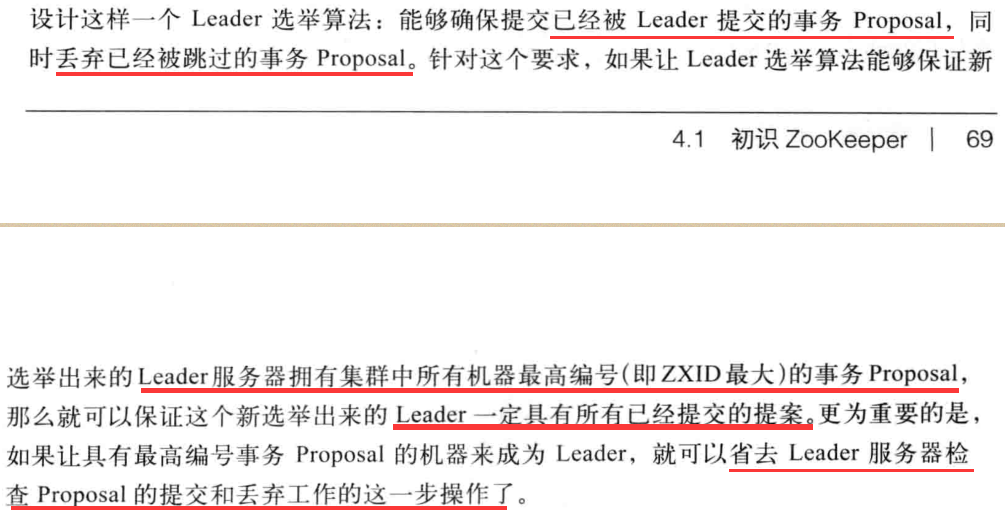




若leader失去与集群中半数以上follower的联系或者leader忽然崩溃，就会进入崩溃模式，选举新的leader。

需要完善的选举算法，保证数据一致性，zk遵循一个原则：若一个事务在一个服务器上提交了，那么必定要在其他服务器提交成功。





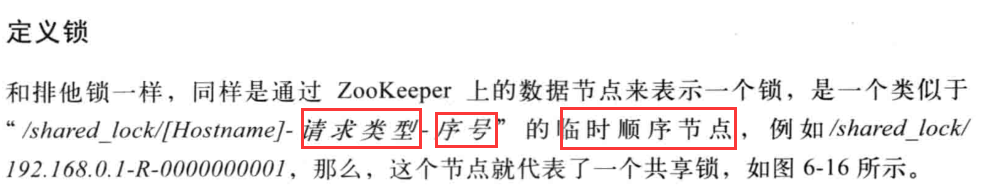
### 集群

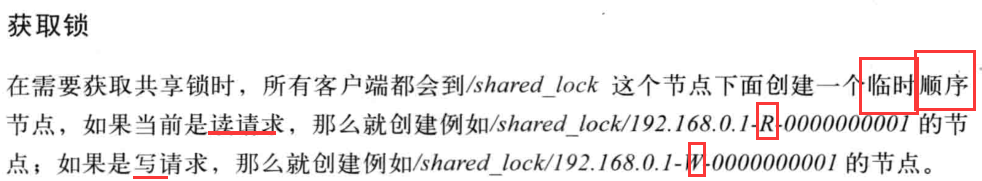
### 分布式锁

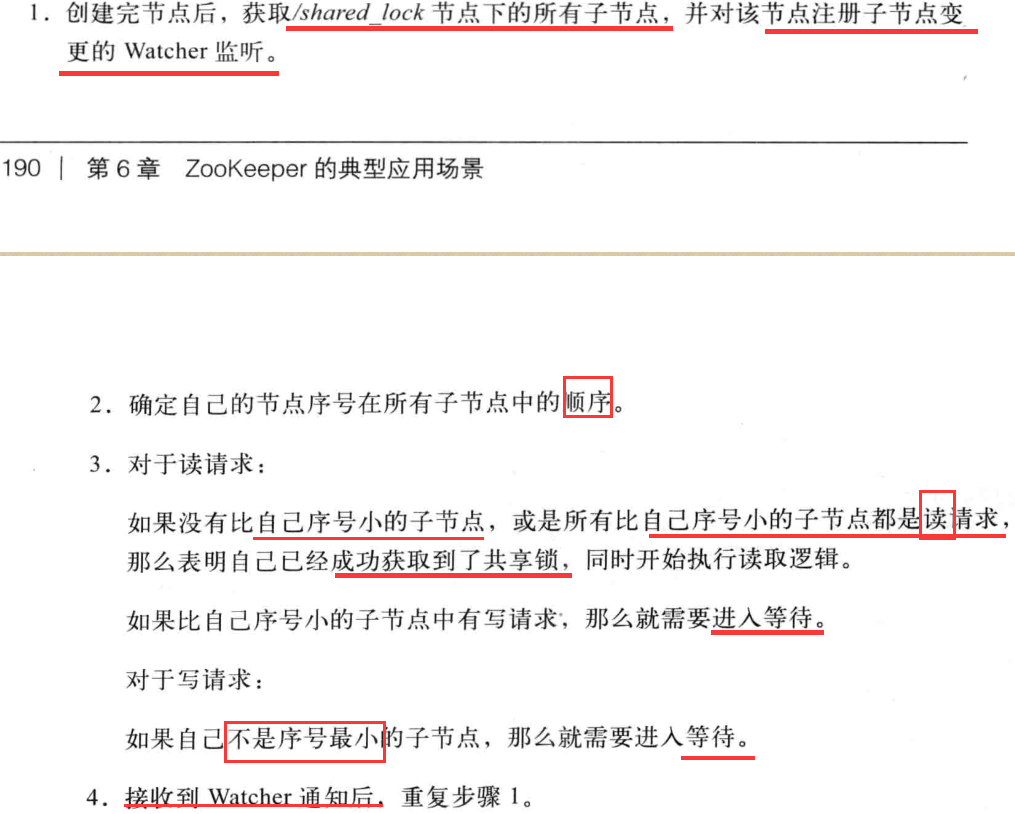
#### 排他锁

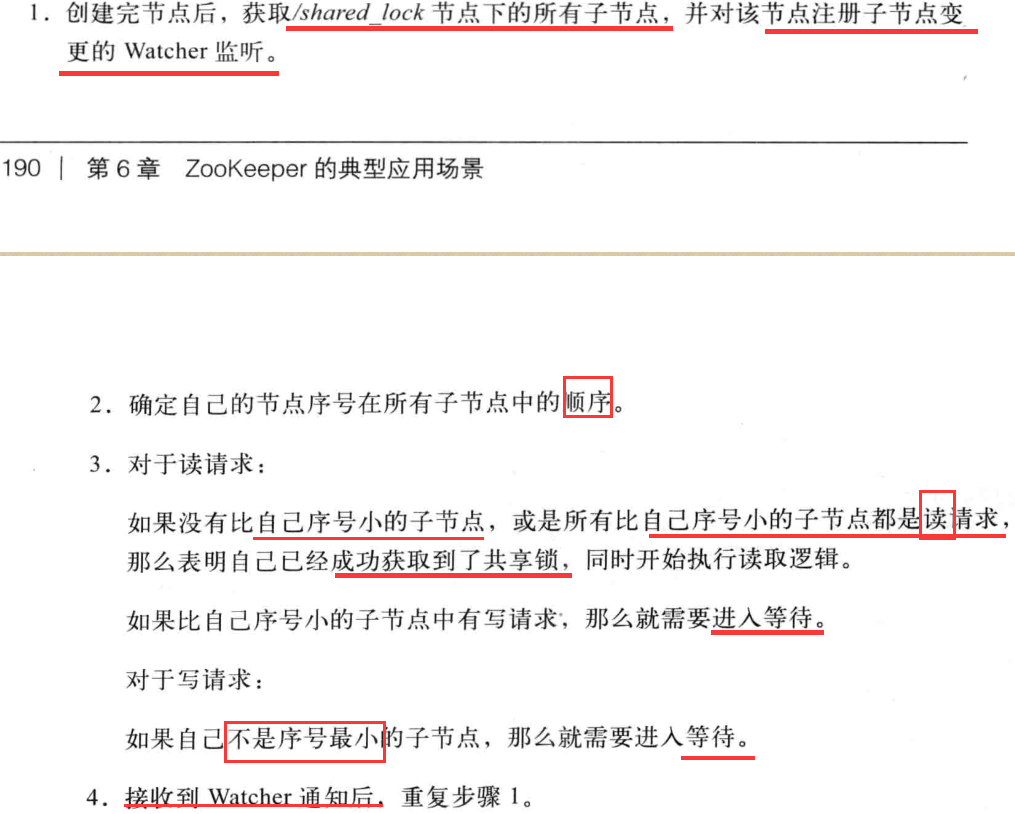
利用客户端在路径下创建一个相同的临时节点实现，当一个线程创建成功，其他线程就在该节点上注册监听，当节点释放，尝试获取创建。

#### 共享锁

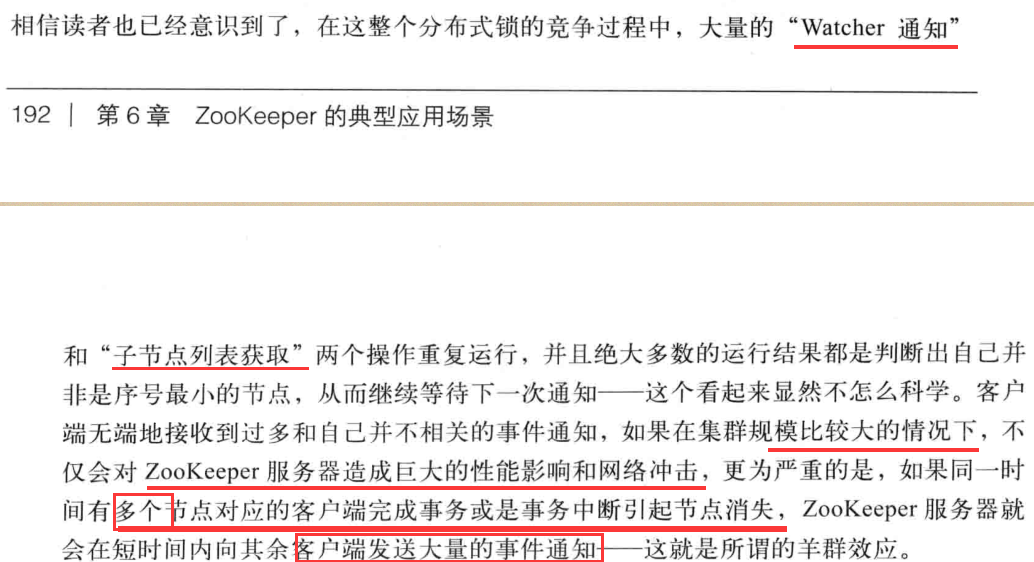


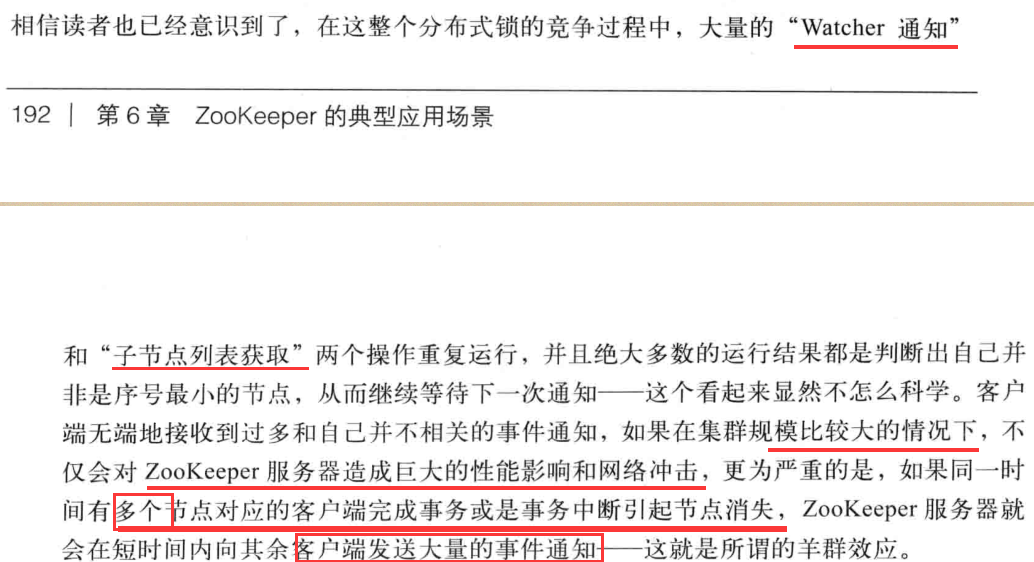


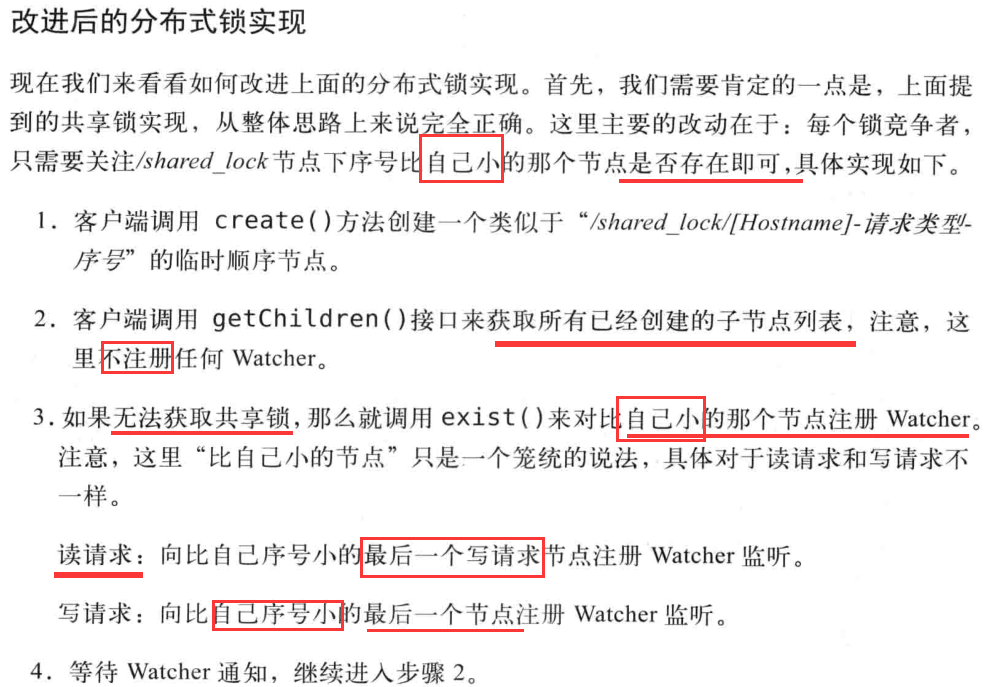


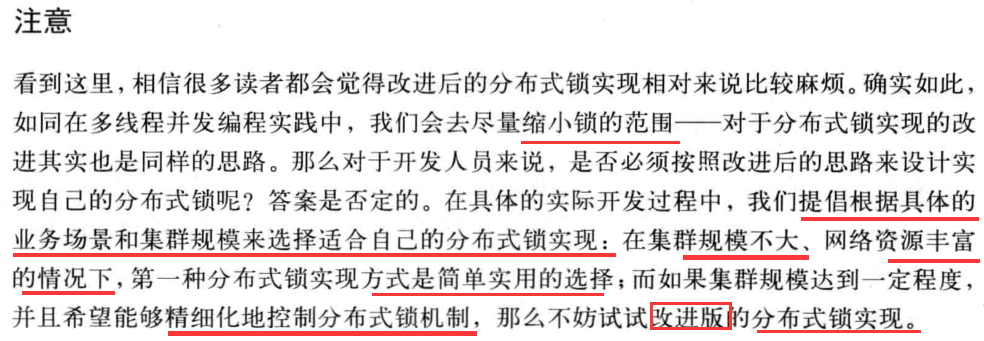


**但是：**



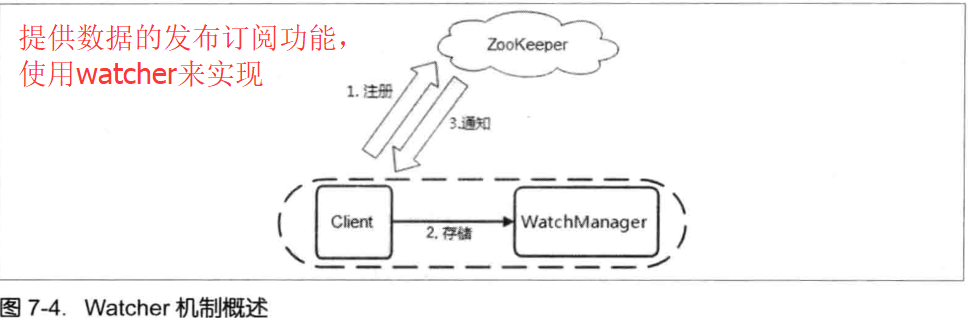


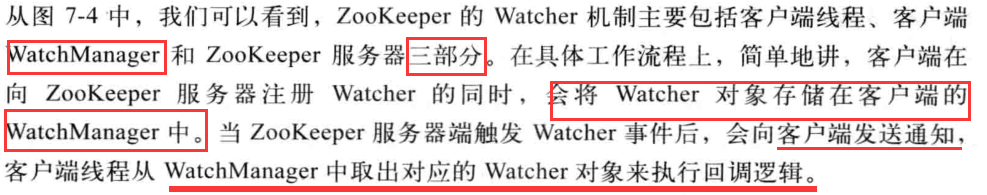




### 5、分布式队列

### 6、Watcher机制





### 7、选举机制

## ActiveMQ

### 集群

## 分布式事务

### 刚性事务，柔性事务

### ACID

### CAP、BASE理论

## Dubbo

### 服务注册与发现

### 服务远程调用与本地调用

### 通信协议

默认是Dubbo协议，是基于传输层的协议，长连接，NIO方式。

其次有WebService协议（SOAP风格），也有rest风格的。

### 通信框架

默认使用Netty实现通信。

### 线程模型

### 常见问题

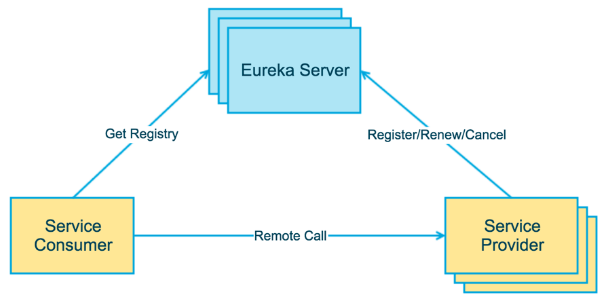
1. 服务提供者全部宕机，注册中心会删除对应的节点（因为创建的是 临时节点，与服务的生命周期关联），但此时消费者会**无限重连提供者**，知道提供者恢复。

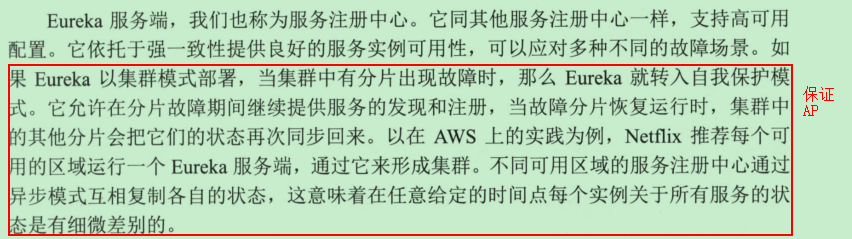
<https://blog.csdn.net/luwei42768/article/details/54847427/>

## Springcloud

### 服务注册与发现

Eureka是一个服务注册与发现中心，它基于CS架构，即存在Server端与Client端。Eureka Server作为服务注册功能的服务器，它是服务注册中心，而系统中的其他微服务，使用Eureka客户端注册到注册中心，并维持心跳。





Eureka自我保护

因为eureka客户端每个30秒会向server发送心跳续约，当server超过90秒没有收到客户端的心跳，会进入自我保护机制，即此时不再接受注册，且不会像zookeeper那样把提供者剔除，（因为可能是因为网络问题，而不是提供者宕机）。但是自我保护期间若是提供者出现问题，那么消费者有可能得到错误信息，因此，客户端应有容错机制，比如请求重试，断路由等

Eureka集群

借助服务消费者也是服务提供者的思想，把server向集群中其他server注册，完成服务列表同步，具体当一个服务注册进一台server时，该server会转发请求到其他server，完成同步，所以不保证强一致性。

#### Eureka与zookeeper比较

https://www.cnblogs.com/zgghb/p/6515062.html

Zookeeper保证的是CP,

Eureka则是AP。

Zookeeper保证CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当leader节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

Eureka保证AP

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务

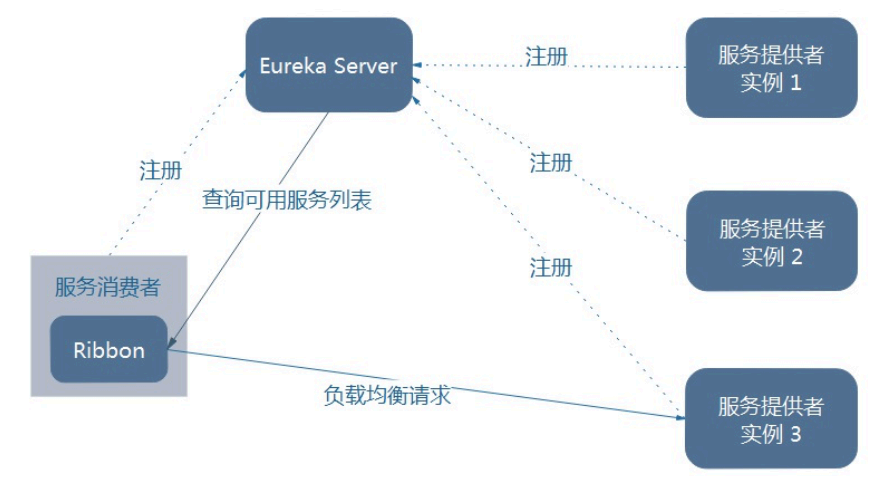
2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)

3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。

### 负载均衡（看周阳笔记）

使用ribbon是基于http+tcp的客户端负载均衡器，eureka客户端实现服务发现，ribbon实现服务消费。



Ribbon在工作时分成两步

第一步先选择 EurekaServer ,它优先选择在同一个区域内负载较少的server.

第二步再根据用户指定的策略，在从server取到的服务注册列表中选择一个地址。

### hystrix服务降级、熔断

服务熔断与服务降级

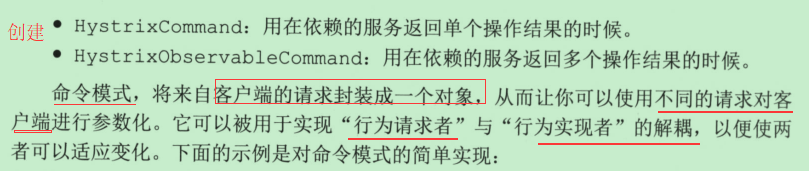
熔断机制是应对雪崩效应的一种微服务链路保护机制。

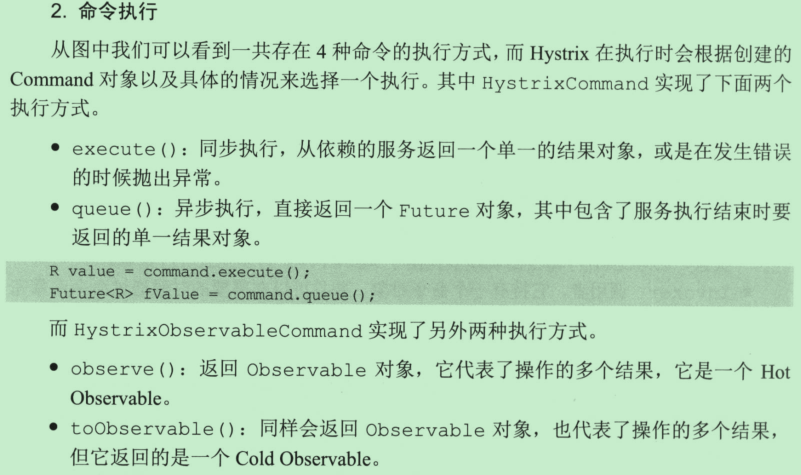
服务熔断一般是因为异常或者服务故障引起的，当某个异常条件出发，就直接熔断整个服务，而不是一直等待此服务致超时。

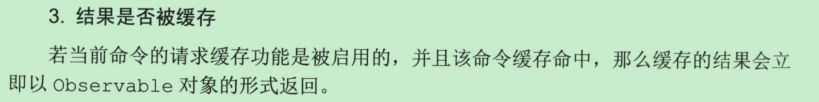
服务降级一般从系统的整体负荷考虑，即在服务熔断之后，该服务不再调用，此时客户端可以准备一个本地的fallback，返回一个缺省值，所以，服务降级是在客户端进行的，与服务端没有关系。

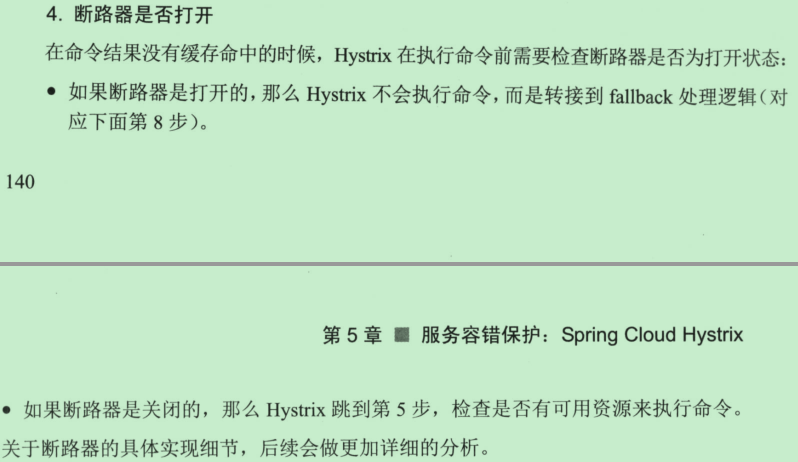
两者的共同结果都是某个服务不能提供服务，都是在系统高负载时触发。

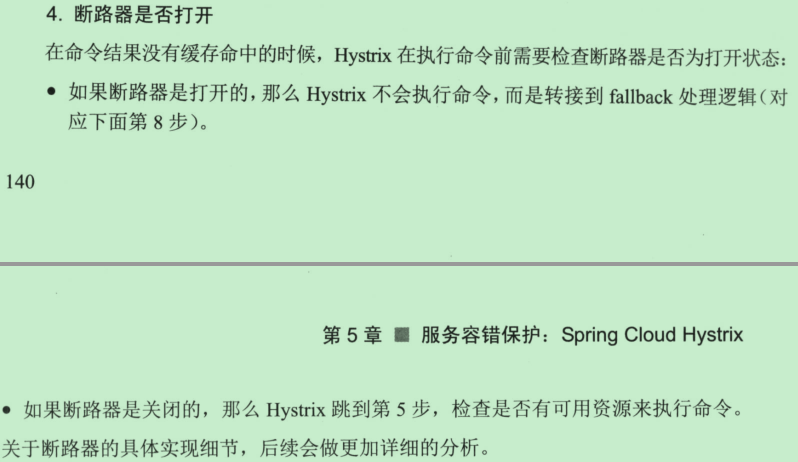
#### Hystrix与断路器

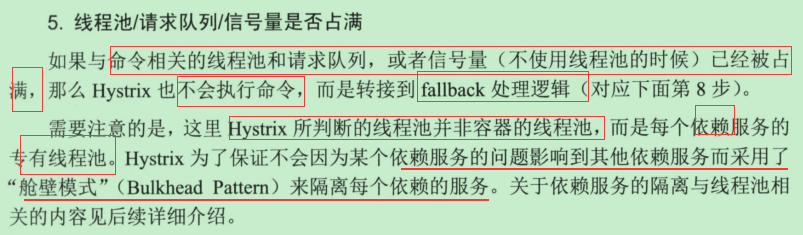


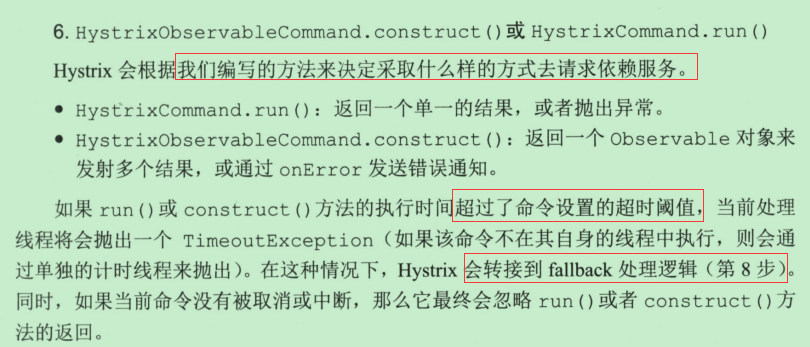


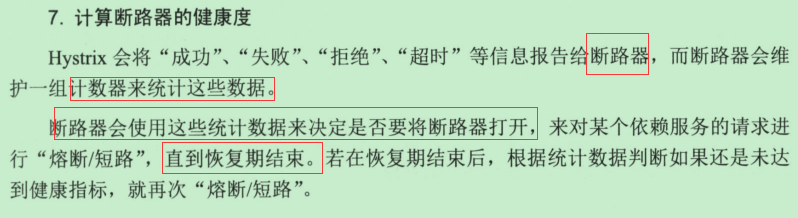


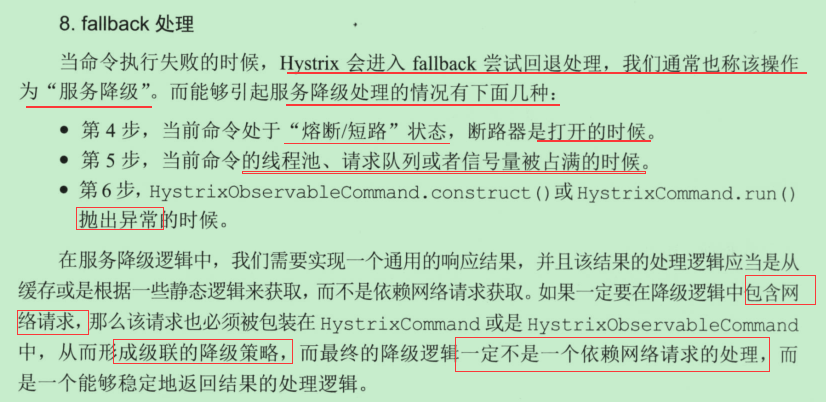


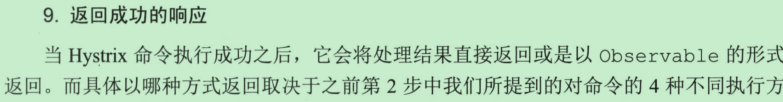


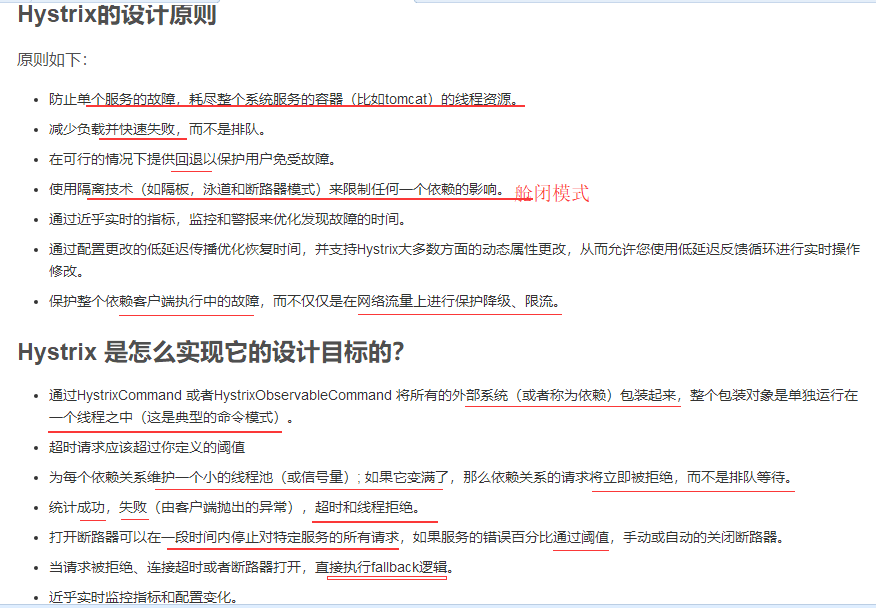


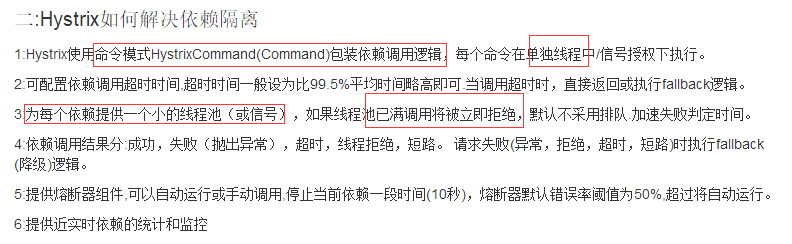












### 路由

### 分布式配置

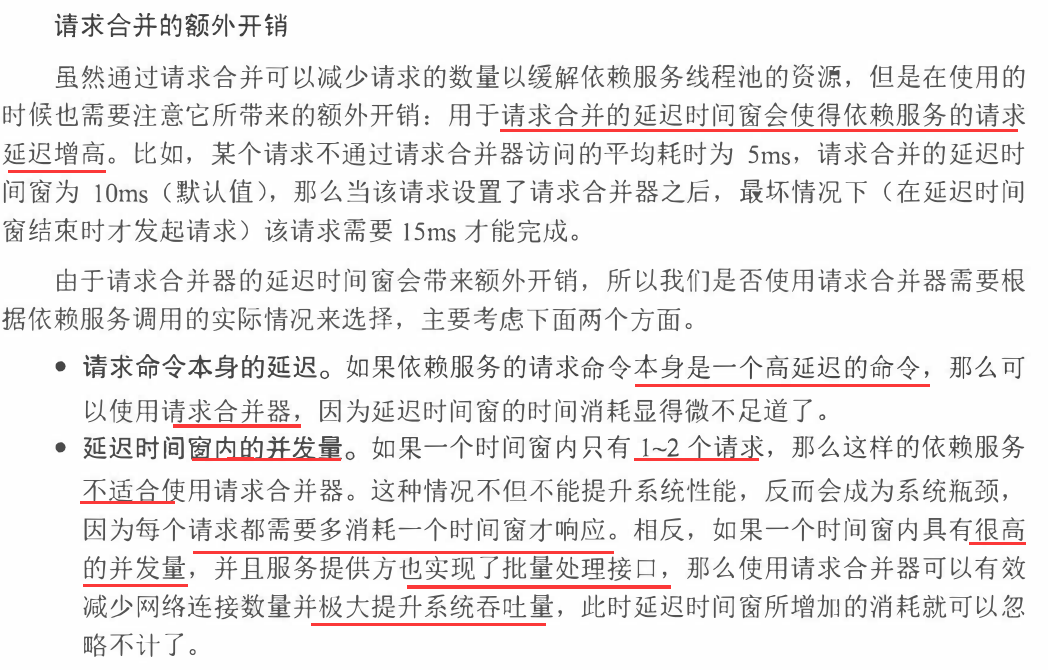
### Hystrix

#### 1、hystrix隔离策略：

Hystrix的线程池隔离模式即为每个依赖服务提供一个单独的线程池，可单独控制每个服务的并发度，当某个依赖服务出现问题不影响其他服务。较不隔离下下有9ms的延迟，若系统是低延迟的（1ms），那可以通过使用信号量来代替，信号量不可异步，无上下切换，开销小。应保证服务足够可靠。



#### 2、请求合并



## 总体内容

### 分布式，集群，SOA，微服务

<https://blog.csdn.net/xinzun/article/details/79412150>

### 什么是微服务

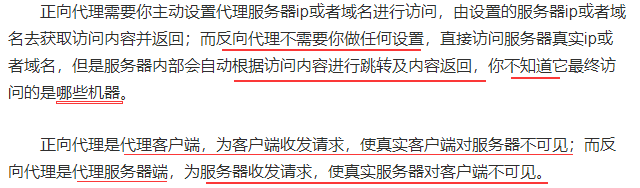
### dubbo与Springcloud比较

### zookeeper与Eureka比较

https://blog.csdn.net/paincupid/article/details/80610441

### dubbo，niginx，Ribbon，Feign负载均衡比较

### 6、正、反向代理



## 有机商城分析

### session共享问题



<https://blog.csdn.net/wojiaolinaaa/article/details/62424642>

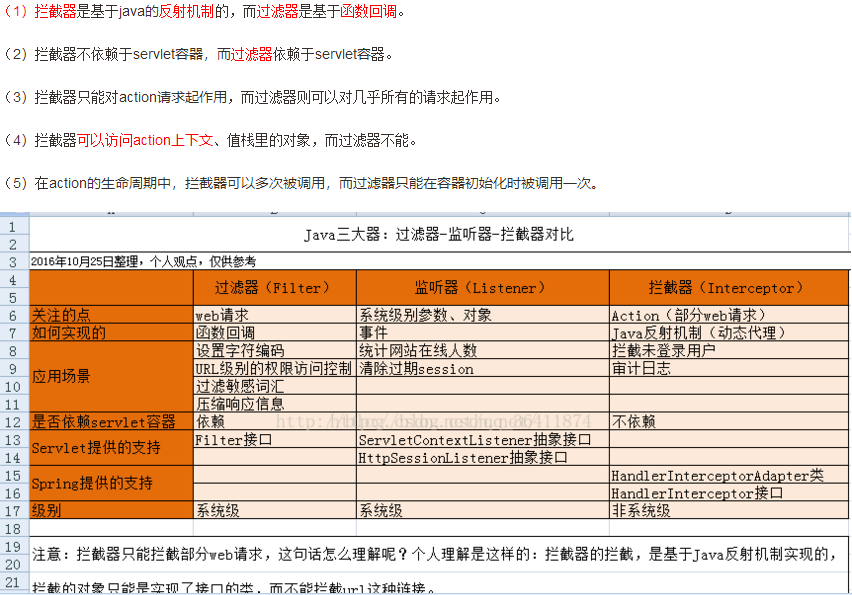
<http://feitianbenyue.iteye.com/blog/2326408>

修改源码实现url重写，即把sessionId跟在url之后。

<https://www.cnblogs.com/duanxz/p/3482089.html>

redis操作类：RedisOperationsSessionRepository

### 2、拦截器与过滤器



## Linux