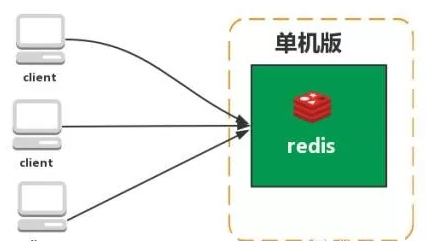
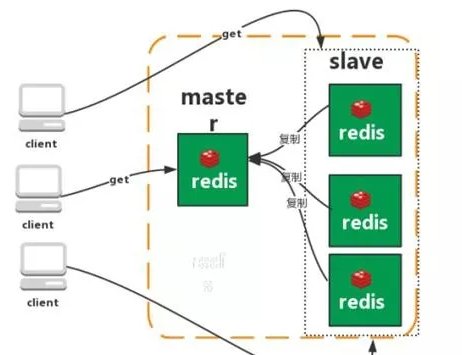
1. **消息队列**
2. **RabbitMQ**
3. 基本工作模式
4. **发布订阅模式**：每个消费者监听自己的队列。生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收到消息。
5. **路由器模式：**每个消费者监听自己的队列，并且设置routingkey。生产者将消息发给交换机，由交换机根据routingkey来转发消息到指定的队列。
6. **统配符模式：**每个消费者监听自己的队列，并且设置带统配符的routingkey。生产者将消息发给broker，由交换机根据routingkey来转发消息到指定的队列。
7. **交换机模式：**
   1. **direct**：消息中的路由键（RoutingKey）如果和 Bingding 中的 bindingKey 完全匹配，交换器就将消息发到对应的队列中。是基于完全匹配、单播的模式。
   2. **fanout**：把所有发送到fanout交换器的消息路由到所有绑定该交换器的队列中，fanout 类型转发消息是最快的。
   3. **topic：**通过模式匹配的方式对消息进行路由，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。
8. **避免消息不被重复消费**
   1. RabbitMQ通过发送一个ACK确认消息。
   2. 业务上产生一个唯一key作为主键存入或更改数据保证了幂等性。
   3. 消息产生用缓存存消息，执行消费业务查缓存，没有则被消费了。
9. **保证消息的可靠传输**
   1. **生产者丢失**：
      1. 事务机制：开启事务，然后发送消息，如果发送过程中出现什么异常，事务就会回滚，如果发送成功则提交事务。缺点是生产者发送消息会同步阻塞等待发送结果是成功还是失败，导致生产者发送消息的吞吐量降下降。
      2. 确认机制：生产者将信道设置为confirm模式，生产消息时产生一个唯一id，信道收到消息将给生产者返回该id。如果rabbitMQ没能处理该消息，也通知生产者，生产者将进行重试。
   2. **消息队列丢失**：
      1. 开启磁盘持久化，配合确认机制，在数据成功存入磁盘给生产者返回ACK。
   3. **消费者丢失：**
      1. 自动确认：消费者收到消息就返回ACK。(消费者异常，会出现消息丢失)
      2. 手动确认：正常消费之后手动发送ACK。
10. **保证消息有序性**
    1. 单线程消费者：将需要保证顺序的消息放到同一个队列，由同一个消费者去进行消费。
    2. 多线程消费者：在上述基础上加重试机制，业务上保证消息顺序消费。
11. **部署模式**
    1. **普通集群模式**：在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。我们创建的 queue，只会放在其中一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。消费的时候，如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。
    2. **镜像集群模式**：RabbitMQ 真正的高可用模式。镜像集群模式下，队列的元数据和消息会存在于多个实例上，每次写消息到 queue 时，会自动将消息同步到各个实例的 queue ，也就是说每个 RabbitMQ 节点都有这个 queue 的完整镜像，包含 queue 的全部数据。任何一个机器宕机了，其它机器节点还包含了这个 queue 的完整数据，其他 consumer 都可以到其它节点上去消费数据。
12. **死信队列DLX:**
    1. DLX也是一个正常的Exchange，和一般的Exchange没有任何区别。能在任何的队列上被指定，实际上就是设置某个队列的属性。当这个队列出现死信（dead message，就是没有任何消费者消费）的时候，RabbitMQ就会自动将这条消息重新发布到Exchange上去，进而被路由到另一个队列。可以监听这个队列中的消息作相应的处理。(1.消息被拒绝并且不重回队列。2：TTL过期。3：队列达到最大长度)
13. **缓存Redis**
    1. **数据类型**
       1. 基本数据类型：**String,hashmap,list,set,zset**
       2. 特殊数据类型：**geospatial,hyperloglog,bitmap**
    2. **持久化机制**
       1. **RDB(Redis DataBase缩写快照)**:按照一定的时间将内存的数据以快照的形式保存到硬盘中，对应产生的数据文件为dump.rdb。通过配置文件中的save参数来定义快照的周期。启动效率高
       2. **AOF(Append Only File持久化)**:则是将Redis执行的每次写命令记录到单独的日志文件中，当重启Redis会重新将持久化的日志中文件恢复数据。优先使用AOF。
    3. **Redis扩容**
       1. 增加Redis服务器的数量，在客户端对存储的key进行hash运算，存入不同的Redis服务器中，读取时，也进行相同的hash运算，找到对应的Redis服务器。
       2. 使用Redis代理twemproxy自动进行hash。
       3. 对Redis的访问分为写和读，写的往主的写，读的从备的读，遇到的情况恰好是读和写是两个服务，做到读写分离通过改下配置信息就可以很简单的做到。
    4. **Redis过期策略(redis采用2+3)**
       1. **定时过期**：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。
       2. **惰性过期**：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
       3. **定期过期**：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。(expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。
    5. **Redis淘汰机制（内存不足处理）**
       1. 基于数据集
          1. volatile-lru：已设置过期时间数据集中挑选最少被使用的数据进行淘汰。
          2. volatile-ttl：从已设置过期时间的数据集中挑选离过期时间最近的数据进行淘汰。
          3. Volatile-random:从已设置过期时间的数据集中随机选取数据进行淘汰。
       2. 基于键空间
          1. Allkeys-lru:在键空间中移除最少使用的的key。
          2. Allkeys-randow:在键空间中随机淘汰数据。
          3. No-eviction:内存不足的时候不允许新写入数据，一旦写入就会报错。
    6. **缓存雪崩，缓存穿透，缓存击穿**
       1. **缓存雪崩**
          1. 原因：在短时间内大量的缓存失效导致直接访问数据库，数据库的请求量陡增。
          2. 解决：缓存的有效时间随机设置或者不设置过期时间。
          3. 缓存穿透：
          4. 原因：请求大量不存在的key，导致请求都没有经过缓存
          5. 解决方案：布隆过滤器（可以判断一定不存在，有可能误判）
          6. 缓存击穿
          7. 原因：区别于缓存雪崩是多个key，缓存击穿是针对某个热点key失效，并且这个key的value构建需要一定时间（复杂计算，多次IO等），导致有大量的线程来构建缓存，增加后端的负载。
          8. 解决方案：
          9. 分级缓存：一级缓存时间短于二级缓存，当一级缓存拿不到数据。一级缓存加锁，读取到数据的时候更新到一二级缓存，这时候所有的请求落到二级缓存，其他线程走二级缓存。（可能在二级缓存读到脏数据）
          10. 互斥锁：只有缓存没有并且拿到锁才能访问数据库，如果缓存没有没有拿到锁，那么等待并且重新调用方法重试。
    7. **Redis架构模式**
       1. 单机版

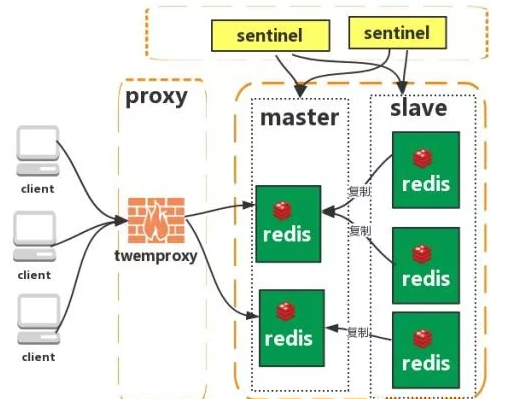


* + 1. 主从复制



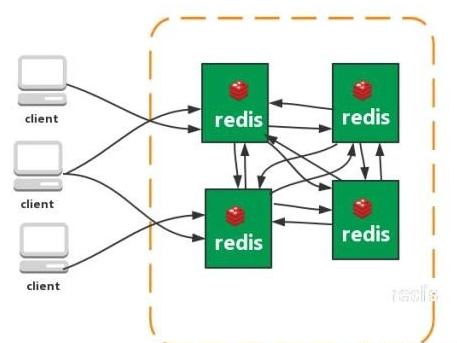
降低了master的读压力，没有解决写压力

* + 1. 集群代理模式



通过代理扩展了redis的容量，解决了读压力

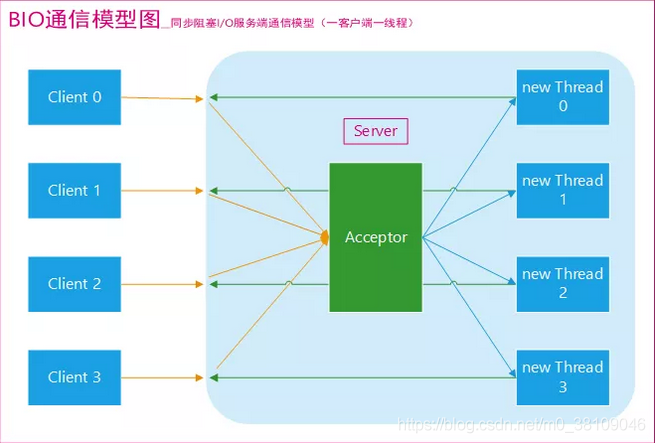
* + 1. 集群直连型



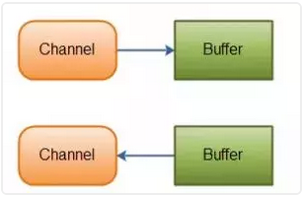
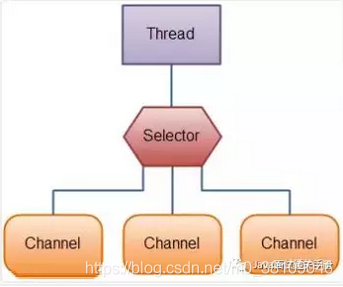
没有中心结构，没有代理，数据采用slot存储（hash），节点数据共享，可以动态扩展，各个节点可以自己做数据备份，数据通过异步复制不保证数据的强一致性。

* 1. **Redisson分布式锁**
     1. **加锁原理：**
        1. 通过hash存储客户端id,锁key的生存时间
        2. 如果你要加锁的那个锁 key 不存在的话，你就进行加锁。
     2. **锁互斥机制**
        1. 尝试获取锁，返回 null 则说明加锁成功，返回一个数值，则说明已经存在该锁，ttl 为锁的剩余存活时间。
        2. 如果此时客户端 2 进程获取锁失败，那么使用客户端 2 的线程 id（其实本质上就是进程 id）通过 Redis 的 channel 订阅锁释放的事件。如果等待的过程中一直未等到锁的释放事件通知，当超过最大等待时间则获取锁失败，返回 false。如果等到了锁的释放事件的通知，则开始进入一个不断重试获取锁的循环。
        3. 循环中每次都先试着获取锁，并得到已存在的锁的剩余存活时间。如果在重试中拿到了锁，则直接返回。如果锁当前还是被占用的，那么等待释放锁的消息，具体实现使用了 JDK 的信号量 Semaphore 来阻塞线程，当锁释放并发布释放锁的消息后，信号量的 release() 方法会被调用，此时被信号量阻塞的等待队列中的一个线程就可以继续尝试获取锁了。
        4. 以上过程存在一个细节，这里有必要说明一下，也是分布式锁的一个关键点：当锁正在被占用时，等待获取锁的进程并不是通过一个 while(true) 死循环去获取锁，而是利用了 Redis 的发布订阅机制,通过 await 方法阻塞等待锁的进程，有效的解决了无效的锁申请浪费资源的问题
     3. **锁的续期机制**
        1. Redisson 提供了一个续期机制， 只要客户端 一旦加锁成功，就会启动一个 Watch Dog如果你想开启 Watch Dog 机制必须使用默认的加锁时间为 30s。如果你自己自定义时间，超过这个时间，锁就会自定释放，并不会延长Watch Dog 机制其实就是一个后台定时任务线程，获取锁成功之后，会将持有锁的线程放入到一个 RedissonLock.EXPIRATION\_RENEWAL\_MAP里面，然后每隔 10 秒 （internalLockLeaseTime / 3） 检查一下，如果客户端 1 还持有锁 key（判断客户端是否还持有 key，其实就是遍历 EXPIRATION\_RENEWAL\_MAP 里面线程 id 然后根据线程 id 去 Redis 中查，如果存在就会延长 key 的时间），那么就会不断的延长锁 key 的生存时间。
  2. **Redis发布订阅机制**
     + 1. 原理：Redis 使用了 channel (频道)作为两者的中介 —— 发布者将信息直接发布给 channel ，而 channel 负责将信息发送给适当的订阅者，发布者和订阅者之间没有相互关系，也不知道对方的存在。
       2. 实现：Redis将所有的订阅关系都保存在服务器的名为pubsub\_channels的字典表内，字典表的键为频道名，值为链表，表中保存客户端信息。在 pubsub\_channels字典里找到对应的频道，依次遍历将效发送给所有的订阅者。
       3. 缺点：不支持复杂的协议，不做消息持久化，没有传输保障

1. **数据库**
   1. **范式**
      1. **第一范式：**属性（对应于表中的字段）不能再被分割，也就是这个字段只能是一个值，不能再分为多个其他的字段了。
      2. **第二范式**：NF 在 1NF 的基础之上，消除了非主属性对于码的部分函数依赖。
      3. **第三范式：**3NF 在 2NF 的基础之上，消除了非主属性对于码的传递函数依赖 。
   2. **存储过程和函数区别**
      1. 函数有一个返回值，存储过程可以没有或者多个
      2. 函数可以放在查询语句中直接调用，存储过程必须单独调用
      3. 存储过程难以扩展和调试没有移植性还会消耗数据库资源
   3. **数据库引擎（MyISAM、InnoDB）**
      1. **行级锁**：MyISAM 只有表级锁(table-level locking)，而 InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁。
      2. **事务**：MyISAM不支持事务，InnoDB支持事务并且具有提交和回滚的能力。
      3. **外键支持**：MyISAM不支持，InnoDB支持。
      4. **数据库异常崩溃的安全恢复**：使用 InnoDB 的数据库在异常崩溃后，数据库重新启动的时候会保证数据库恢复到崩溃前的状态。这个恢复的过程依赖于 redo log。
      5. ***MVCC***多版本并发控制：InnoDB支持，MyISAM不支持。
   4. **事务隔离级别**
      1. 读未提交：可以读取还未提交的数据，会产生脏读
      2. 读提交：只能读取已提交的数据，会产生不可重复读
      3. 可重复读：开始读事务，不允许再修改数据，会产生幻读
      4. 序列化：事务串行化，影响性能
   5. **MVCC多版本并发控制**
      1. 隐藏字段：DB\_TRX\_ID（最后一次更新该条数据的事务id）,DB\_ROLL\_PTR(回滚指针指向该行的undolog),DB\_ROW\_ID（没有主键没有索引默认产生的索引）。
      2. **Readview:**
         1. m\_low\_limit\_id(目前出现过的最大的事务 ID+1，即下一个将被分配的事务 ID)。
         2. m\_up\_limit\_id：活跃事务列表 m\_ids 中最小的事务 ID，如果 m\_ids 为空，则 m\_up\_limit\_id 为 m\_low\_limit\_id。小于这个 ID 的数据版本均可见。
         3. m\_ids：Read View 创建时其他未提交的活跃事务 ID 列表。创建 Read View时，将当前未提交事务 ID 记录下来，后续即使它们修改了记录行的值，对于当前事务也是不可见的。m\_ids 不包括当前事务自己和已提交的事务（正在内存中）。
         4. m\_creator\_trx\_id：创建该 Read View 的事务 ID
      3. **Undo-log:**
      4. **数据可见性算法：**
         1. 当前事务id<m\_up\_limit\_id||id=m\_creator\_trx\_id,可见
         2. 当前事务id>=m\_low\_limit\_id,不可见
         3. M\_ids为空，可见
         4. 其他在ids中用二分查找判断当前事务id在不在ids中，若不在则可以访问，否则不可访问。
         5. [数据库常见面试题](https://blog.csdn.net/qq_22222499/article/details/79060495)
2. **框架**
   1. Spring
      1. 单例bean的线程安全
         1. 尽量不要定义可变成员变量
         2. 实在需要定义可以用ThreadLocal来保存变量
      2. @Component和@Bean
         1. @Component用于类@Bean用于方法（注入该方法的返回类，配合@Cofiguration）
         2. 如果某个类是外部引入，没法使用@ComPonent那么就可以使用@Bean实例化，再在使用的地方@Autowired注入
      3. Spring MVC原理
         1. 客户端发送请求到服务端DispatcherServlet
         2. DispatcherServlet通过请求信息调用HandlerMapping，将请求解析到相应的handler
         3. Handler接收到请求，通过handlerAdapter来验证请求合法性并且开始通过handler处理请求
         4. 处理器处理完逻辑之后返回model给DispatcherServlet
         5. DispatcherServlet将返回推给客户端，客户端进行view的渲染
      4. Spring框架中用到的设计模式
         1. 单例模式：bean
         2. 工厂模式：beanfactory
         3. 代理模式：SpringAOP功能实现
         4. 模板模式：jdbcTemplate hibernateTemplate
         5. 适配器模式：spring MVC handlerAdapter
      5. 事务传播方式
         1. PROPAGATION\_REQUIRED:支持当前事务，如果没有事务则会创建一个新的事务
         2. PROPAGATION\_SUPPORT:支持当前事务，如果没有事务则会以非事务方式运行
         3. PROPAFAATION\_MANDATORY:支持当前事务，如果没有事务就会报错
         4. PROPAGATION\_REQUIREDS\_NEW:创建一个新的事务，并且把当前事务挂起
         5. PROPAGAYION\_NOT\_SUPPORTED：以非事务的方式运行，并且把当前事务挂起
         6. PROPGATION\_NEVER:以非事务方式运行，如果存在事务就抛出异常
         7. PROPAGATION\_NESTED:类似于REQUIRED,有事务就嵌套在当前事务里面
      6. Springboot自动装配
      7. POST和GET请求的优缺点
         1. GET请求速度更快，但是浏览器对URL长度限制
         2. GET请求会缓存返回数据
         3. GET请求数据暴露在URL中，安全性低于POST
         4. GET请求只支持SACII字符
         5. POST请求在三次握手之后，服务器会返回100 continue，浏览器这时候才开始发送数据。而get在三次握手最后一次会直接发送数据，这也是为什么GET请求速度更快。
      8. Dao接口可以重载条件
         1. 仅有一个无参方法和一个有参方法
         2. 多个有参方法，参数数目相同，并且使用相通的@Param
   2. Netty
      1. [BIO NIO和AIO区别](https://blog.csdn.net/m0_38109046/article/details/89449305)
         1. BIO:同步阻塞。默认情况下服务端为每个请求单独建立一个请求，如果服务端没有足够的线程创建，那么请求被拒绝或者服务器宕机。



* + - 1. NIO:同步非阻塞。当一个连接建立不需要立刻建立一个线程，而是被注册到多路复用器中，多路复用器进行轮训，发现连接上有请求，才为之开启一个线程进行。*HTTP/1.1出现后，有了Http长连接，这样除了超时和指明特定关闭的http header外，这个链接是一直打开的状态的，这样在NIO处理中可以进一步的进化，在后端资源中可以实现资源池或者队列，当请求来的话，开启的线程把请求和请求数据传送给后端资源池或者队列里面就返回，并且在全局的地方保持住这个现场(哪个连接的哪个请求等)，这样前面的线程还是可以去接受其他的请求，而后端的应用的处理只需要执行队列里面的就可以了，这样请求处理和后端应用是异步的.当后端处理完，到全局地方得到现场，产生响应，这个就实现了异步处理。*

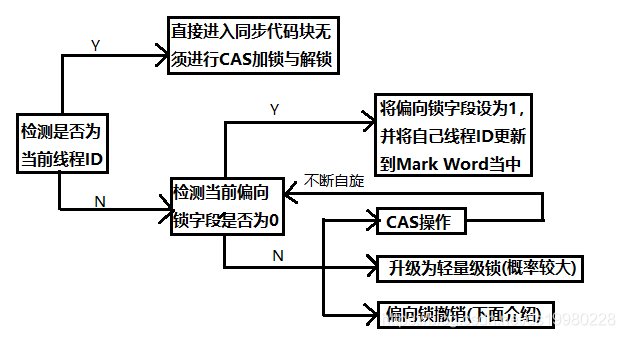


* + - 1. AIO:异步非阻塞。 与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。

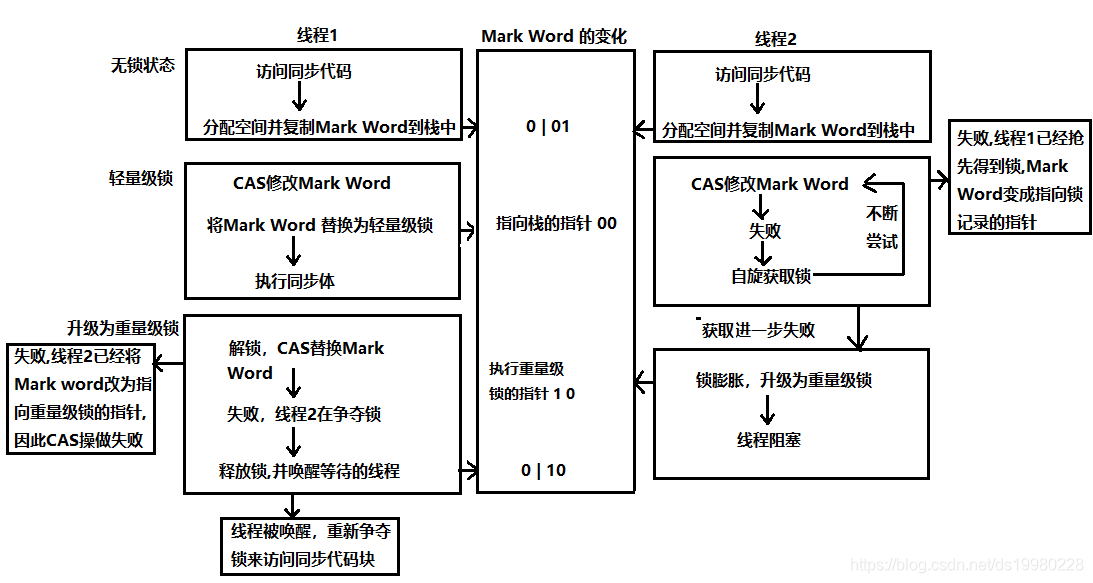
1. **设计模式**
   1. **单例模式（双重校验单例模式）**
   2. **工厂和抽象工厂模式**
   3. **策略模式**
   4. **代理和动态代理模式**
   5. **适配器模式**
2. **Zookeeper**
   1. **Zookeeper特点**
      1. **顺序一致性**
      2. **原子性**
      3. **单一系统映像**
      4. **可靠性**
   2. **PAXOS算法和raft算法**
   3. **ZK实现分布式锁**
      1. **对要申请锁的资源申请一个永久节点**
      2. **客户端通过上述节点下申请临时节点来获取锁**
      3. **当可客户端生成的节点id是最小的则表示申请到了该锁资源，否则则在最小的临时节点上设置一个wacher**
      4. **当锁释放时，临时节点删除，其他节点监听到节点删除事件，并再次创建临时节点。**
3. **分布式**
   1. **CAP理论：一致性，可用性，分区容错性（Zookeeper CP,Eureka AP）**
   2. **BASE理论：基本可用（允许牺牲部分性能或者功能），软状态（允许不一致的中间态），最终一致性（一定时间达到一致）**
4. **Springboot**
   1. **核心功能/优点**
      1. **独立运行的spring项目：可以以jar包形式独立运行**
      2. **内嵌servlet容器：内嵌tomcat、jetty或者undertow，无需以war包形式部署项目**
      3. **提供starter简化maven配置：提供一系列start pom连简化maven的依赖加载**
      4. **自动装配spring:springboot根据在类路径的jar包，为jar包中的类自动配置bean**
      5. **准生产的应用监控：提供基于http ssh telnet对运行中的项目进行监控**
      6. **无代码生产和xml配置：不是借助于代码生成实现，二十通过注解来实现**
   2. **Springboot四大组件**
      1. **Springboot Strarter：封装好我们所需要的依赖，避免我们自己添加导致的jar包冲突或者缺少包的情况。帮组我们自动注入bean，不需要我们手动配置。**
      2. **Springboot Autoconfigure:存在于starter，每个starter都依赖autoconfigure。autoconfigure内容是配置Bean实例到Spring容器的实际代码实现包，然后提供给starter依赖。**
      3. **Springboot CLI:一个命令行使用Springboot客户端的工具**
      4. **Springboot actuator:springboot监控插件，本身提供了很多接口可以获取当前项目的各项运行状态指标。**
   3. **SpringBoot自动装配**
      1. **@SpringbootApplication包含@SpringbootConfiguration（支持JavaConfig的方式来进行配置），@EnableAutoConfiguration，@ConmponentScan(没有指定扫描包，默认扫描该类同级类或者同级包下面所有类)**
      2. **@EnableAutoConfiguration（@AutoConfigurationPackage[扫描外部引入的bean],@Import(AutoConfigurationImportSelector.class)[Spring启动的时候会扫描所有jar路径下的META-INF/spring.factories，将其文件包装成Properties对象从Properties对象获取到key值为EnableAutoConfiguration的数据，然后添加到容器里边。]）**
   4. **什么是javaConfig**

**提供一种纯java的方式来配置pring ioc容器，减少或者消除了xml配置。**

* 1. **[Java 锁升级](https://blog.csdn.net/ds19980228/article/details/84189273)**
     1. **偏向锁->轻量级锁：判断当前锁对象对象头的偏向锁进程id,如果偏向锁进程id为当前线程的id,那么无需CAS加锁和解锁过程直接获取锁。否则判断Epoch字段是否为0，如果为01那么将epoch设为1并且对象头的锁id设为该进程id并获得锁。否则不断CAS操作进行自旋获得锁，直到epoch字段大于阈值40锁升级为轻量级锁，原先获得锁进程撤销偏向锁。**



* + 1. **轻量级锁->重量级锁：线程在执行同步代码块之前，JVM先在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头的mark word字段直接复制到此空间中。然后线程尝试使用CAS将对象头的mark word替换为指向锁记录的指针（指当前线程），如果成功表示获取到轻量级锁。如果失败，表示其他线程竞争轻量级锁，当前线程便使用自旋来不断尝试，最后膨胀为重量级锁。**



* + 1. **三种锁应用场景**
       1. **重量级锁会阻塞，唤醒请求加锁的线程。针对的是多个线程同一个时刻竞争同一把锁的情况，JVM采用自适应自旋，来避免线程在面对非常小的同步块时，仍会被阻塞以及唤醒。**
       2. **轻量级锁采用CAS操作，将锁对象的标记字段替换为指向线程的指针，存储着锁对象原本的标记字段。针对的是多个线程在不同时间段申请同一把锁的情况。**
       3. **偏向锁只会在第一次请求时采用CAS操作，在锁对象的mark word字段中记录下当前线程ID，此后运行中持有偏向锁的线程不再有加锁过程。针对的是锁仅会被同一线程持有**