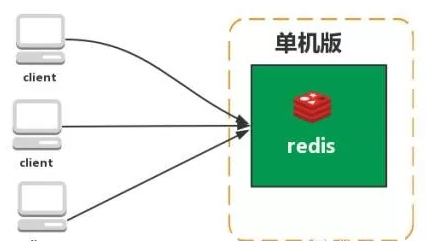
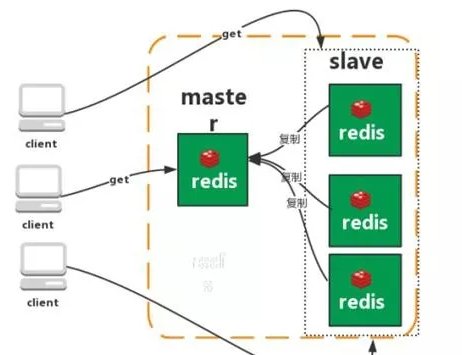
1. **消息队列**
2. **RabbitMQ**
3. 基本工作模式
4. **发布订阅模式**：每个消费者监听自己的队列。生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收到消息。
5. **路由器模式：**每个消费者监听自己的队列，并且设置routingkey。生产者将消息发给交换机，由交换机根据routingkey来转发消息到指定的队列。
6. **统配符模式：**每个消费者监听自己的队列，并且设置带统配符的routingkey。生产者将消息发给broker，由交换机根据routingkey来转发消息到指定的队列。
7. **交换机模式：**
   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。(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。
       2. **缓存穿透：**
          1. 原因：请求大量不存在的key，导致请求都没有经过缓存
          2. 解决方案：布隆过滤器（可以判断一定不存在，有可能误判）
       3. **缓存击穿**
          1. 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。
          2. 解决方案：
             1. 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）
             2. 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

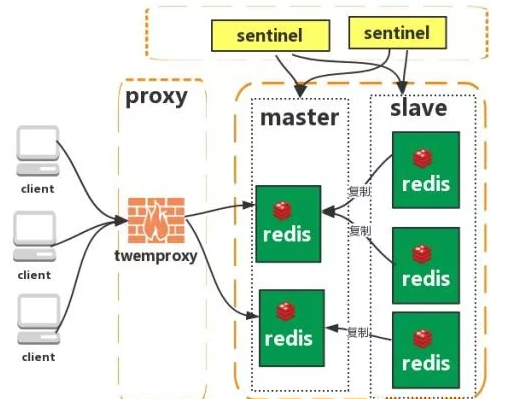


* + 1. 主从复制



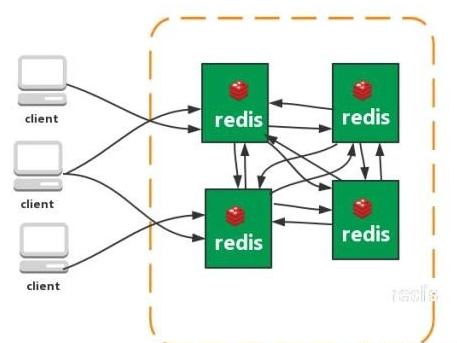
降低了master的读压力，没有解决写压力

* + 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。