六十二、ReentrantReadWriteLock读写状态的设计？

读写锁对于同步状态的实现是在一个整形变量上通过“按位切割使用”：将变量切割成两部分，高16位表示读，低16位表示写。

写锁的获取与释放：

调用的独占式同步状态的获取与释放，因此真实的实现就是Sync的 tryAcquire和 tryRelease。

读锁的获取与释放：

类似于写锁，读锁的lock和unlock的实际实现对应Sync的 tryAcquireShared 和 tryReleaseShared 方法。

```
如果已经有线程获取了写锁，则其他任何线程如果尝试获取读写锁都会失败。
如果已经有线程获取了读锁，其他线程尝试获取读锁时，需要判断阻塞队列的第一个阻塞线程尝试获取的锁的类型，如果是读锁，
    则可以非公平的竞争读锁。如果是写锁，则加入阻塞队列，在两次尝试获取读锁失败后被挂起。
区别在读锁只有尝试写锁的时候才放到等待队列，而写锁是只要非当前线程尝试加锁（无论写锁还是读锁）都会放到等待队列。
```

六十三、Netty模型

Netty拥有两个NIO线程池，分别是bossGroup和workerGroup，前者处理新建连接请求，然后将新建立的连接轮询交给workerGroup中的其中一个NioEventLoop来处理，后续该连接上的读写操作都是由同一个NioEventLoop来处理。

BossGroup 和 WorkerGroup 类型都是 NioEventLoopGroup，这个组中可以含有多个 NioEventLoop

每个Boss NioEventLoop 循环执行的步骤有3步

1、轮询accept 事件 ，2、处理accept 事件 ，与client建立连接 , 生成NioScocketChannel , 并将其注册到某个worker NIOEventLoop 上的 selector，

3、处理任务队列的任务 ， 即 runAllTasks；

每个 Worker NIOEventLoop 循环执行的步骤 ：

1、轮询read, write 事件 处理i/o事件， 即read , write 事件，2、在对应NioScocketChannel 处理，

3、处理任务队列的任务 ， 即 runAllTasks ；

每个Worker NIOEventLoop 处理业务时，会使用pipeline(管道)执行的, pipeline 中包含了 channel , 即通过pipeline 可以获取到对应通道, 管道中维护了很多的 处理器。

六十四、tcp粘包、解决办法？

一次接收到了客户端发送过来的一个完整的数据包

一次接收到了客户端发送过来的 N 个数据包，由于每个包的长度不定，无法将各个数据包拆开

一次接收到了一个或者 N 个数据包 + 下一个数据包的一部分，还是很悲剧，无法将数据包拆开

一次收到了半个数据包，下一次接收数据的时候收到了剩下的一部分 + 下个数据包的一部分，更悲剧，头大了

```
1使用标准的应用层协议（比如：http、https）来封装要传输的不定长的数据包
2在每条数据的尾部添加特殊字符，如果遇到特殊字符，代表当条数据接收完毕了
    有缺陷：效率低，需要一个字节一个字节接收，接收一个字节判断一次，判断是不是那个特殊字符串
3在发送数据块之前，在数据块最前边添加一个固定大小的数据头，这时候数据由两部分组成：数据头 + 数据块
    数据头：存储当前数据包的总字节数，接收端先接收数据头，然后在根据数据头接收对应大小的字节
    数据块：当前数据包的内容
```

六十五、JAVA8新特性

1.Lambda 表达式

Lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。

使用Lambda 表达式可以使代码变的更加简洁紧凑。

2.Java 8 方法引用

方法引用通过方法的名字来指向一个方法。

方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

方法引用使用一对冒号 :: 。

3.Java 8 新增了接口的默认方法。

4.Stream

Java 8 API添加了一个新的抽象称为流Stream，可以让你以一种声明的方式处理数据。

将要处理的元素集合看作一种流，流在管道中传输，并且可以在管道的节点上进行处理，比如filter，sorted，映射map，聚合collect等

5.Optional 类

Optional 类是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。

Optional 是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。

Optional 类的引入很好的解决空指针异常。

六十六、Java 浅拷贝和深拷贝的理解和实现方式

浅拷贝的实现方式主要有三种：

一、通过拷贝构造方法实现浅拷贝：

Person p1=new Person(a,"被拷贝");

Person p2=new Person(p1);

相当于复制出一个新引用

二、通过重写clone()方法进行浅拷贝

通过super.clone()调用Object类中的原clone方法（被protected修饰，无法直接用），直接返回克隆出的obj（引用）

深拷贝的实现方法主要有两种：

一、通过重写clone方法来实现深拷贝

每一层的每个对象都进行浅拷贝=深拷贝

二、通过对象序列化实现深拷贝

将对象序列化为字节序列后，默认会将该对象的整个对象图进行序列化，再通过反序列即可完美地实现深拷贝。

六十七、RabbitMQ常用的五种模型?

第一种：简单模式 Simple

一个队列丶一个消费者。

第二种：工作模式 Work

一个队列丶多个消费者。

第三种：广播模式 (fanout)

多个消费者，每一个消费这都有自己的队列，每个队列都绑定到交换机

交换机把消息发送给绑定过的所有队列；队列的消费者都能拿到消息。fanout类型转发消息是最快的

第四种：路由模式 Routing(direct)

消息的路由键（rounting key）如果和binding key一致，交换机根据路由key发送给对应队列。是完全匹配，一对一的模式。

第五种：主题Topic模式

Topics模式和direct路由模式类似，交换器通过模式匹配分配消息的消息的路由键（rounting key），分发到对应的队列上。

符号(通配符)：#表示匹配0个或者多个词、*表示匹配一个词

六十八、如何保证消息队列MQ数据不丢失？

对于生产者：

生产者（Producer）通过网络发送消息给 Broker，当 Broker 收到之后，将会返回确认响应信息给 Producer。所以生产者只要接收到返回的确认响应，就代表消息在生产阶段未丢失。

对于MQ：

Q:rabbitMQ成功接收到了消息并保存在内存中, 但是在仓储服务没有拿走此消息之前, rabbitMQ宕机了. 怎么办?

A:此问题需要考虑消息持久化(durable机制), 通过设置队列的durable参数为true, 则当rabbitMQ重启之后, 会恢复之前的队列。

它的工作原理是rabbitMQ会把队列的相关信息持久化到磁盘.

对于消费者：

Q:接收到一条订单消息之后, 并对此条消息没有处理完之前,突然宕机了?

A: rabbitMQ默认操作是当消费者成功接收到消息之后,rabbitMQ则会自动的在队列中将此条消息删除. 这种操作称为自动ACK(自动回复).

核心痛点就在于autoAck这个参数. 需要将此参数设置为false. 当此参数设置为false. 那么当消费者接收到这个消息之后,消息队列也不会马上删除这条消息.

对于我们开发人员要做的就是只有执行成功之后才会向消息对返回一条确认消息,当消息队列接收到这条消息之后才删除消息

Q:基于ack机制,结合高并发场景会出现什么问题?

A: 每一个channel都会存在若干的unack消息(未确认消息).

比方说, rabbitMQ正在发送的消息 、 消费者实例接收到消息之后但没有处理完 、 执行了ack但是因为ack是异步的也不会马上变为ack信息 、 开始批量ack延迟时间会更长.

此时如果rabbitMQ无限制的过多过快的向消费者实例发送消息,就会导致庞大的unack消息积压在消费者实例的内存中,如果继续保持发与积压的状态,最终会导致消费者实例的oom!!.

此时需要考虑消费者实例的处理能力以及如何解决unack消息积压的问题.可以考虑提高消费者的消费能力。

同时rabbitMQ基于 prefetch count(预抓取总数)控制每一个channel的unack消息的数量。 当一个channel中的unack消息超过阈值之后, rabbitMQ则会停止向这个消费者实例投递消息,

等待unack消息总数小于阈值 或者 将消息转发给其他的消费者实例.这个阈值过大可能导致系统雪崩, 过小导致系统吞吐量过低,响应速度低.

Q：当前队列节点挂了，存储节点硬盘又坏了，消息丢了，怎么办？

消息补偿机制。消息补偿机制需要建立在业务数据库和MQ数据库的基础之上，当我们发送消息时，需要同时将消息数据保存在数据库中, 两者的状态必须记录。

然后通过业务数据库和MQ数据库的对比检查消费是否成功，不成功，进行消息补偿措施，重新发送消息处理

六十九、消息积压你是怎么处理的？

设计系统时，要保证消费端的消费性能要高于生产者的发送性能，这样系统才能健康持续运行。

消息积压产生的原因有2种：第一是消息生产速度过快，另一种是消息消费速度变慢。可以通过查看消息队列内置的监控数据，确定是生产端还是消费端的原因。

如果是消费端的原因，就进行消费端性能优化：

首先可以优化消费业务逻辑，尽量减少冗余。还可以增加消费端的并发数，也就是扩容Consumer实例，也必须同步扩容主题中的队列数量，确保Consumer的实例数量和队列数量相等。

如果是有大促或者抢购导致消息突增，短时间内不可能通过优化代码来解决，唯一的方法是通过增加消费者实例数来提升总体的消费能力。

如果短时间内没有足够的服务器资源进行扩容，那么可以将系统降级，关闭一些不重要的业务，减少生产者生产的消息数量，让系统尽可能正常运转，服务一些重要的业务。

如果通过监控发现生产消息的速度和消费消息的速度都没有什么异常，那就需要检查一下消费端，是不是消费失败导致了一条消息反复消息。

如果监控到消费变慢了，需要检查一下消费实例，分析一下是什么原因导致消费变慢。可以检查一下日志看是否有大量的消费错误，还可以通过打印堆栈信息，看一下消费线程是不是卡在什么地方了，比如发生死锁或者等待资源。

七十、RabbitMQ、RocketMQ、Kafka，都有可能会出现消息重复消费的问题，如何解决？

消费端处理消息的业务逻辑保持幂等性。幂等性，通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，不能出错。

> 比如，你拿到这个消息做数据库的`insert`操作。那就容易了，给这个消息做一个唯一主键，那么就算出现重复消费的情况，就会导致主键冲突，避免数据库出现脏数据。
> 再比如，你拿到这个消息做`redis`的`set`的操作，那就容易了，不用解决，因为你无论`set`几次结果都是一样的，`set`操作本来就算幂等操作。
> 如果上面两种情况还不行，上大招。准备一个第三方介质,来做消费记录。以`redis`为例，给消息分配一个全局`id`，只要消费过该消息，将`<id,message>`以`K-V`形式写入`redis`。那消费者开始消费前，先去`redis`中查询有没消费记录即可。