面试题

以下内容都是基础知识仅供铺垫参考，每个点的涉及范围都很广，而且JAVA知识面几道题不可能完全覆盖，面试过程中绝对不会只问一个点，还需自己下功夫学好，把知识学全。

java基础8题

1.string截取

- split
  正则表达式切割,在所以的都完了之后返回整个数组
- tokennizer
  分组返回
- substring
  空间换时间,根据下标
- charAT
  根据原始的String由char数组组成,所以最高效

2.spring事务

- 事务隔离机制
  事务分三个部分 DataSource、TransactionManager和代理机制;
  五个种方式:
  1每个bean单独代理;2共享基类;3拦截器(beanid拦截);4tx标签配置的拦截器;5全注解
  传播,
    - PROPAGATION_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
    - PROPAGATION_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。
    - PROPAGATION_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。
    - PROPAGATION_REQUIRES_NEW--新建事务，如果当前存在事务，把当前事务挂起。
    - PROPAGATION_NOT_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
    - PROPAGATION_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

- 事务隔离级别
    - DEFAULT 使用数据库设置的隔离级别（ 默认 ），由 DBA 默认的设置来决定隔离级别 .
    - READ_UNCOMMITTED 会出现脏读、不可重复读、幻读（ 隔离级别最低，并发性能高 ）
    - READ_COMMITTED 大多数主流数据库的默认事务等级，保证了一个事务不会读到另一个并行事务已修改但未提交的数据，避免了“脏读取”。该级别适用于大多数系统;会出现不可重复读、幻读问题（锁定正在读取的行）
    - REPEATABLE_READ 保证了一个事务不会修改已经由另一个事务读取但未提交（回滚）的数据。避免了“脏读取”和“不可重复读取”的情况，但是带来了更多的性能损失; 会出幻读（锁定所读取的所有行）
    - SERIALIZABLE 最严格的级别，事务串行执行，资源消耗最大;保证所有的情况不会发生（锁表）

3.hashMap 循环链表形成

- 循环链表形成
  由于非线程安全在扩容的时候，多个线程操作引起：
  hashMap在达到扩容因子的时候会扩容（length*0.75）；
  扩大一倍，扩容时会重新计算在数组中得位置，
  当第一个线程开始扩容的时候，读取了hash值对应链表顺序读取复制到新的hash位置的链头；原来的顺序是A、B，复制过去后的顺序就是B、A，对线程一来说A的next是null，B的next是A；
  这时第二个线程进来，读取的可能就是A的next是B，这样就造成了环形链表死循环。

- 避免
  安全线程ConcurrentHashMap。

- 其他问题
  由于不是安全线程，不仅仅是会环形链表，还有可能会造成数据覆盖丢失。

4.Synchronized 与Lock

- ReentrantLock和Synchronized
  ReentrantLock 拥有Synchronized相同的并发性和内存语义，此外还多了 锁投票，定时锁等候和中断锁等候
  线程A和B都要获取对象O的锁定，假设A获取了对象O锁，B将等待A释放对O的锁定，
  如果使用 synchronized ，如果A不释放，B将一直等下去，不能被中断
  如果 使用ReentrantLock，如果A不释放，可以使B在等待了足够长的时间以后，中断等待，而干别的事情

- ReentrantLock
  ReentrantLock获取锁定与三种方式：
  a) lock(), 如果获取了锁立即返回，如果别的线程持有锁，当前线程则一直处于休眠状态，直到获取锁
  b) tryLock(), 如果获取了锁立即返回true，如果别的线程正持有锁，立即返回false；
  c)tryLock(long timeout,TimeUnit unit)， 如果获取了锁定立即返回true，如果别的线程正持有锁，会等待参数给定的时间，在等待的过程中，如果获取了锁定，就返回true，如果等待超时，返回false；
  d) lockInterruptibly:如果获取了锁定立即返回，如果没有获取锁定，当前线程处于休眠状态，直到或者锁定，或者当前线程被别的线程中断

- 区别

  - synchronized：
    在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化synchronized，另外可读性非常好，不管用没用过5.0多线程包的程序员都能理解。
  - ReentrantLock:
    ReentrantLock提供了多样化的同步，比如有时间限制的同步，可以被Interrupt的同步（synchronized的同步是不能Interrupt的）等。在资源竞争不激烈的情形下，性能稍微比synchronized差点点。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍。而ReentrantLock确还能维持常态。

- 参考博客

```
https://blog.csdn.net/u012403290/article/details/64910926?locationNum=11&fps=1
```

5.ThreadLocal,以及死锁分析

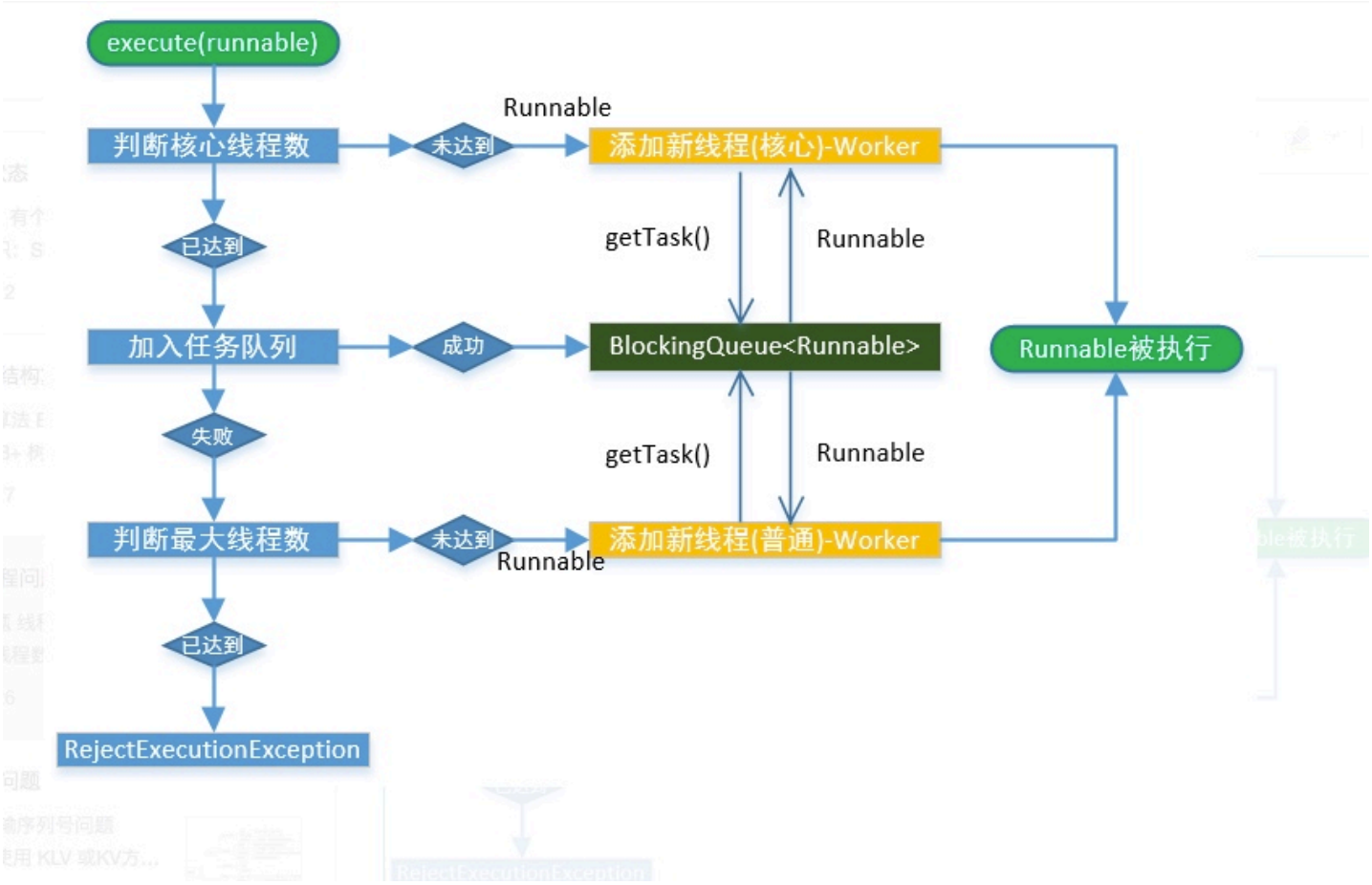
hreadLocal为每个线程维护一个本地变量。
采用空间换时间，它用于线程间的数据隔离，为每一个使用该变量的线程提供一个副本，每个线程都可以独立地改变自己的副本，而不会和其他线程的副本冲突。
ThreadLocal类中维护一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值为对应线程的变量副本。

- 参考博客

```
https://www.cnblogs.com/xzwblog/p/7227509.html
```

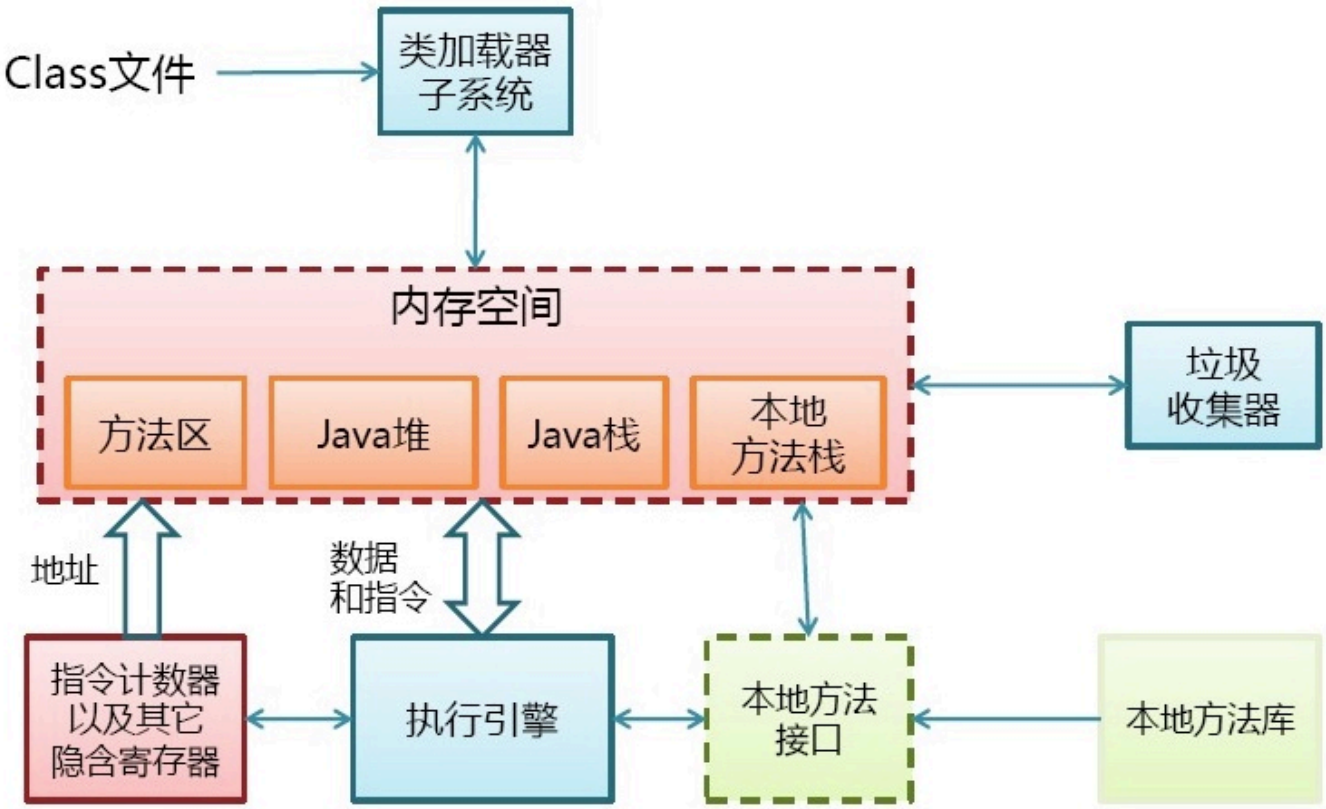
6.线程池工作原理

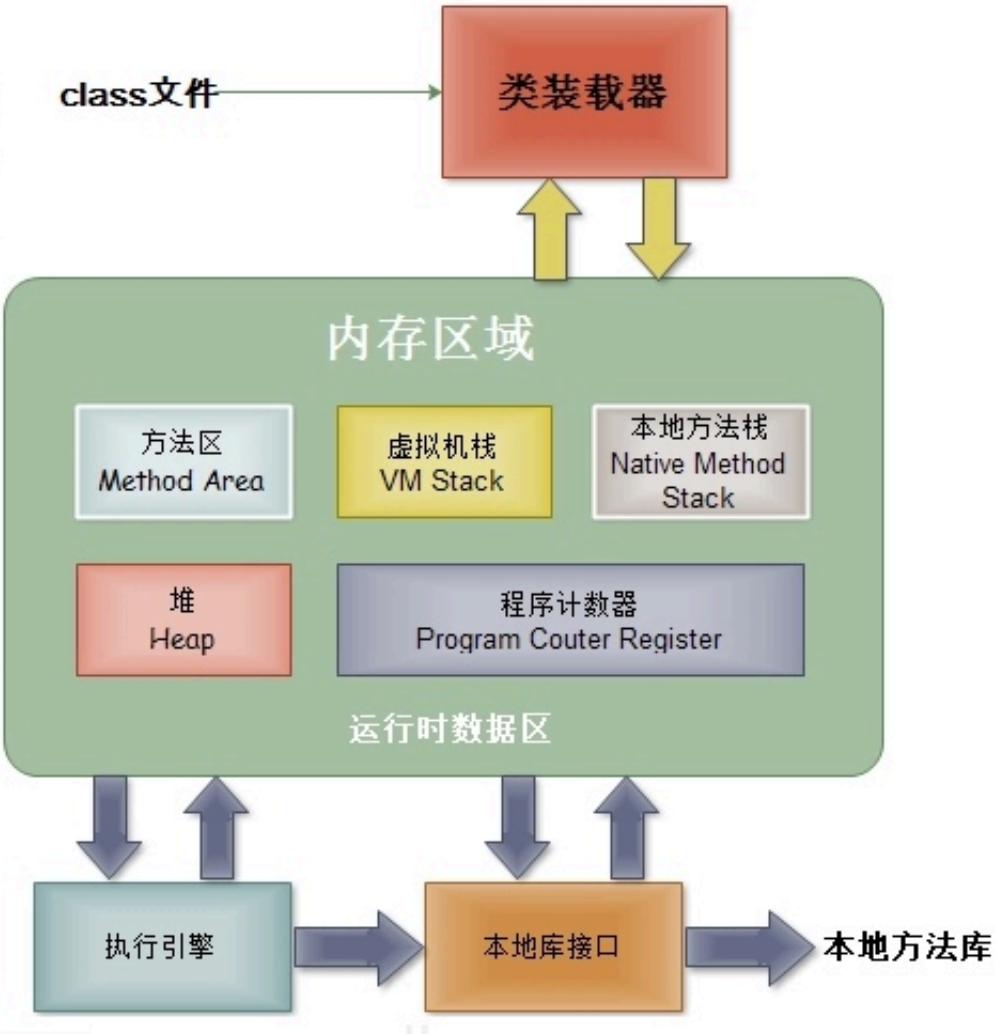
ThreadPool 工作原理
任务先去核心线程池,如果核心线程池没有则存入到Q中;
Q满了去启新的线程(启新线程时会全局锁),不能启的话就返回错误



需要多了解些，参数配置和使用结合。

7.jvm内存模型（JDK7）


Class文件
类加载器
子系统
内存空间
方法区
Java堆
Java栈
本地
方法栈
垃圾
收集器
地址
数据
和指令
指令计数器
以及其它
隐含寄存器
执行引擎
本地方法
接口
本地方法库

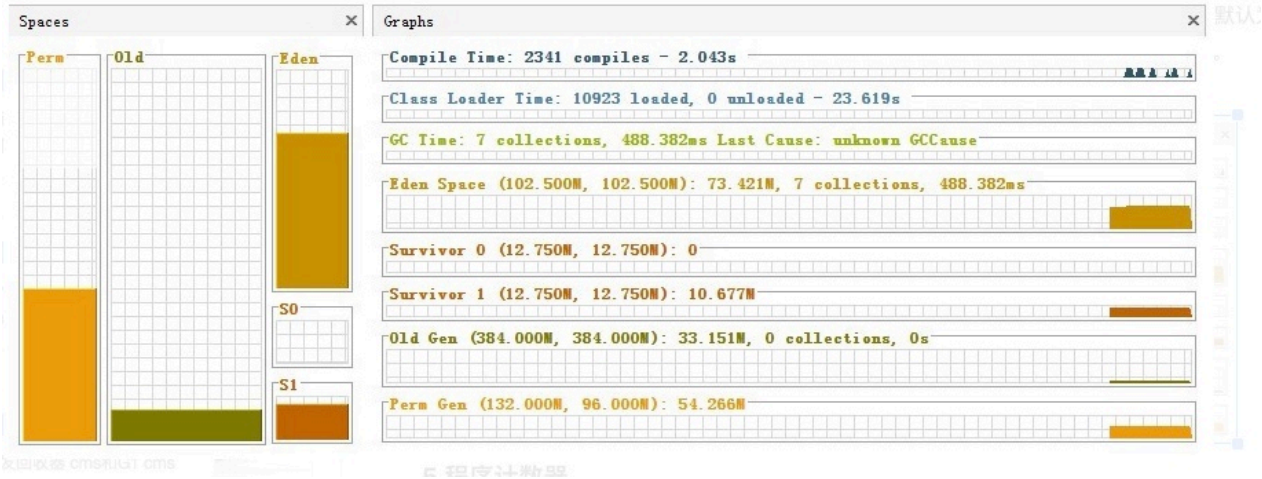

class文件
类装载器
内存区域
方法区
Method Area
虚拟机栈
VM Stack
本地方法栈
Native Method
Stack
堆
Heap
程序计数器
Program Couter Register
运行时数据区
执行引擎
本地库接口
本地方法库

- 方法区
  也称"永久代”、“非堆”， 它用于存储虚拟机加载的类信息、常量、静态变量、是各个线程共享的内存区域。默认最小值为16MB，最大值为64MB，可以通过-XX:PermSize 和 -XX:MaxPermSize 参数限制方法区的大小。
  运行时常量池：是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译器生成的各种符号引用，这部分内容将在类加载后放到方法区的运行时常量池中。
- 虚拟机栈
  描述的是java 方法执行的内存模型：每个方法被执行的时候 都会创建一个“栈帧”用于存储局部变量表(包括参数)、操作栈、方法出口等信息。每个方法被调用到执行完的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。声明周期与线程相同，是线程私有的。
  局部变量表存放了编译器可知的各种基本数据类型(boolean、byte、char、short、int、float、long、double)、对象引用(引用指针，并非对象本身)，其中64位长度的long和double类型的数据会占用2个局部变量的空间，其余数据类型只占1个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在栈帧中分配多大的局部变量是完全确定的，在运行期间栈帧不会改变局部变量表的大小空间。
- 本地方法栈
  与虚拟机栈基本类似，区别在于虚拟机栈为虚拟机执行的java方法服务，而本地方法栈则是为Native方法服务。

- 堆
  也叫做java 堆、GC堆是java虚拟机所管理的内存中最大的一块内存区域，也是被各个线程共享的内存区域，在JVM启动时创建。该内存区域存放了对象实例及数组(所有new的对象)。其大小通过-Xms(最小值)和-Xmx(最大值)参数设置，-Xms为JVM启动时申请的最小内存，默认为操作系统物理内存的1/64但小于1G，-Xmx为JVM可申请的最大内存，默认为物理内存的1/4但小于1G，默认当空余堆内存小于40%时，JVM会增大Heap到-Xmx指定的大小，可通过-XX:MinHeapFreeRation=来指定这个比列；当空余堆内存大于70%时，JVM会减小heap的大小到-Xms指定的大小，可通过XX:MaxHeapFreeRation=来指定这个比列，对于运行系统，为避免在运行时频繁调整Heap的大小，通常-Xms与-Xmx的值设成一样。
  由于现在收集器都是采用分代收集算法，堆被划分为新生代和老年代。新生代主要存储新创建的对象和尚未进入老年代的对象。老年代存储经过多次新生代GC(Minor GC)任然存活的对象。

  - 新生代：
    程序新创建的对象都是从新生代分配内存，新生代由Eden Space和两块相同大小的Survivor Space(通常又称S0和S1或From和To)构成，可通过-Xmn参数来指定新生代的大小，也可以通过-XX:SurvivorRation来调整Eden Space及Survivor Space的大小。
  - 老年代：

用于存放经过多次新生代GC任然存活的对象，例如缓存对象，新建的对象也有可能直接进入老年代，主要有两种情况：①.大对象，可通过启动参数设置–XX:PretenureSizeThreshold=1024(单位为字节，默认为0)来代表超过多大时就不在新生代分配，而是直接在老年代分配。②.大的数组对象，切数组中无引用外部对象。
老年代所占的内存大小为–Xmx对应的值减去–Xmn对应的值。



- 程序计数器
  是最小的一块内存区域，它的作用是当前线程所执行的字节码的行号指示器，在虚拟机的模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、异常处理、线程恢复等基础功能都需要依赖计数器完成。

8.并发回收器 cms和G1

- cms
  初始标记,并发标记,重新标记,回收
  停顿低,并发标记最耗时但是可以与用户线程同时进行;
  虽然不会停顿但是会造成和用户线程同时使用cpu,拖慢用户线程.可以设置成与用户交替使用cpu;
  无法清除浮动的垃圾,标记到清除之间产生的垃圾;
  空间碎片太多,会造成实际空间不大的情况下没有连续空间而Full GC;可以设置整理碎片空间,但是会有停顿;

- G1
  初始标记,并发标记,最终标记,回收
  通过并发和用户线程共存;
  弱化了分带的概念,标记中有复制过程,减少碎片空间;
  清理的停顿时间可预测

数据库3题

9.MySQL中myisam与innodb的区别

innodb支持事务，myisam不支持；

	InnoDB	MyISAM
事务	支持	不支持
存储结构	一体	分文件
锁	行表锁	表锁
全文检索	不支持	支持
外键	支持	不支持

- 应用场景
  日常使用大多数innodb，适用于insert、update多的情况，并且事务和锁能很好的支持业务使用；myisam用于查询多的，需要提供高速检索。
  常见案例，主用innodb，从是myisam；具体使用根据场景而定

10.索引的构成和原理

- 结构
  mysql索引使用的B-Tree和衍生种类
  MyISAM是B-Tree,innodb 是B+Tree
  MyISAM是表锁,innodb是行锁
  与B-Tree相比，B+Tree有以下不同点：
  每个结点的指针上限为2d而不是2d+1。
  内结点不存储data，只存储key；叶子结点不存储指针。

- 如何知道sql 有没有用到索引
  explant sql 打印出sql 是否使用索引,和索引命中

- 不推荐使用索引的情况
  数据唯一性查询, 频繁修改,查询条件少的情况;
  where中有IS NULL /IS NOT NULL/ like ‘%输入符%’ <> 等不推荐使用索引

11.事务和锁

锁表事务问题
锁类型
for update，lock in share mode，next-key locks，mvcc

forupdate时候，id为主键，RR策略时候，锁住了的条件符合的行，但是如果条件找不到任何列，锁住的是整个表，（主键，唯一索引，非唯一索引，（insert，update对于gab锁不通）

可重复读和提交读本来就是矛盾的，如果可重复读了看不到其他事务的提交；如果可提交读，两次读取的就不一样，不是重复读。

默认的可重读中，加入了next-key locks。 可重读并不保证避免幻读。

mysql 可重读为了防止幻读，加入了for update，lock in share mode;普通读的select存在幻读。

数据隔离级别

隔离级别	脏读	不可重复读	幻读
读未提交（Read uncommitted）	V	V	V
读已提交（Read committed）	x	V	V
可重复读（Repeatable read）	x	x	V
可串行化（Serializable）	x	x	x

数据隔离级别
Read Uncommitted（读取未提交内容）
未提交的事务读取,很少用
Read Committed（读取提交内容）
提交的内容读取,并发读取时如果有事务提交,内容不一致
Repeatable Read（可重读）
确保并发时一致,但是会出现幻读
Serializable（可串行化）
添加并发控制,解决幻读问题;加的共享锁,最高级别可能导致大量的超时现象和锁竞争
幻读 （Phantom Read）:读取时有新的内容加入.

中间件2题

12.redis原理

- redis为什么快
  使用单线程操作,由于是直接对内存操作,单线程不需要考虑上下文切换,锁竞争等.

极限在cpu的缓存读取,cpu速度特别快,所以单线程就够了,影响redis速度的,多是网络消耗,持久化过程;
所以redis适合部署在少核缓存快速 CPU的cpu机器上.

- 持久化方式
  master 是无阻塞的,slave是阻塞的,slave在同步时不能响应客户端查询;
  可以配置master不持久化,slave持久化,但是有延迟;
  redis有自己的虚拟内存,因为Linux的虚拟内存page太大(已经弃用)
  Snapshotting（快照）默认方式；
  父子进程操作,父进程接受client操作,子进程持久化;有写入的话父进程直接写入临时区域;
  临死区域完事后替换掉原来的快照文件. 此时就需要两倍的相同的内存.
  每次是做全部的数据完整写入硬盘,不是增量很大影响内存.
  Append-only file（缩写aof）；
  虚拟内存:2.4 后已经放弃
  diskstore:放弃了虚拟内存方式后

- redis cluster
  每个master维护了一个位序列,用bit记录自己是否拥有槽位;
  还维护了一个槽位到机器节点映射,16384数组实现,槽位是下标,value是节点

  - 集群添加过程
    各个服务之间meet握手(没有自动发现);
    分配角色, 默认是都是master;
    槽位指派,迁移槽位内容信息
    涉及到的主要的数据结构
    clusterState：集群状态
    nodes：所有结点
    migrating_slots_to：迁出中的槽
    importing_slots_from：导入中的槽
    slots_to_keys：槽中包含的所有Key，用于迁移Slot时获得其包含的Key
    slots：Slot所属的结点，用于处理请求时判断Key所在Slot是否自己负责
    clusterNode：结点信息
    slots：结点负责的所有Slot，用于发送Gossip消息通知其他结点自己负责的Slot。通过位图方式保存节省空间，16384/8恰好是2048字节，所以槽总数16384不是随意定的。
    clusterLink：与其他结点通信的连接

13.MQ

- ActiveMQ/ApolloMQ

    - 优点：老牌的消息队列，使用Java语言编写。对JMS支持最好，采用多线程并发，资源消耗比较大。如果你的主语言是Java，可以重点考虑。
    - 缺点：由于历史悠久，历史包袱较多，版本更新很缓慢。集群模式需要依赖Zookeeper实现。最新架构的产品被命名为Apollo，号称下一代ActiveMQ，目前案例较少。

- RocketMQ/Kafka
    - 优点：专为海量消息传递打造，主张使用拉模式，天然的集群、HA、负载均衡支持。话说还是那句话，适合不适合看你有没有那么大的量。
    - 缺点：所谓鱼和熊掌不可兼得，放弃了一些消息中间件的灵活性，使用的场景较窄，需关注你的业务模式是否契合，否则山寨变相使用很别扭。除此之外，RocketMQ没有.NET下的客户端可用。RocketMQ身出名门，但使用者不多，生态较小，毕竟消息量能达到这种体量的公司不多，你也可以直接去购买阿里云的消息服务。Kafka生态完善，其代码是用Scala语言写成，可靠性比RocketMQ低一些。

- RabbitMQ
    - 优点：生态丰富，使用者众，有很多人在前面踩坑。AMQP协议的领导实现，支持多种场景。淘宝的MySQL集群内部有使用它进行通讯，OpenStack开源云平台的通信组件，最先在金融行业得到运用。
    - 缺点：Erlang代码你Hold得住不？虽然Erlang是天然集群化的，但RabbitMQ在高可用方面做起来还不是特别得心应手，别相信广告。

- 参考文章

```
https://blog.csdn.net/qq_30764991/article/details/80516961
```

分布式4题

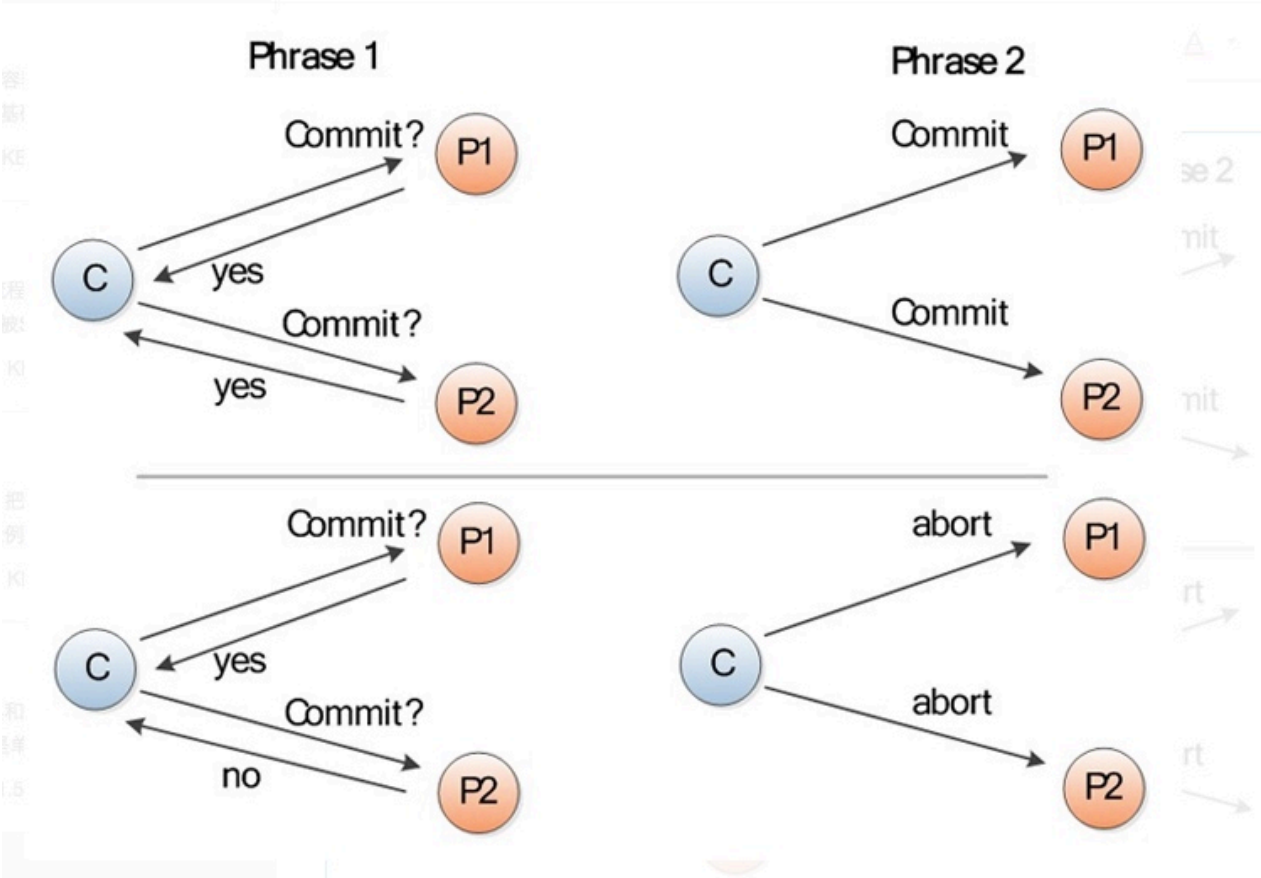
14.分布式一致性事务

- 基本概念
    - CAP
      数据一致性consistency，数据可用性Availability，分区耐受性partition tolerance。
      强一致，用户一致，最终一致。
    - BASE
      Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）

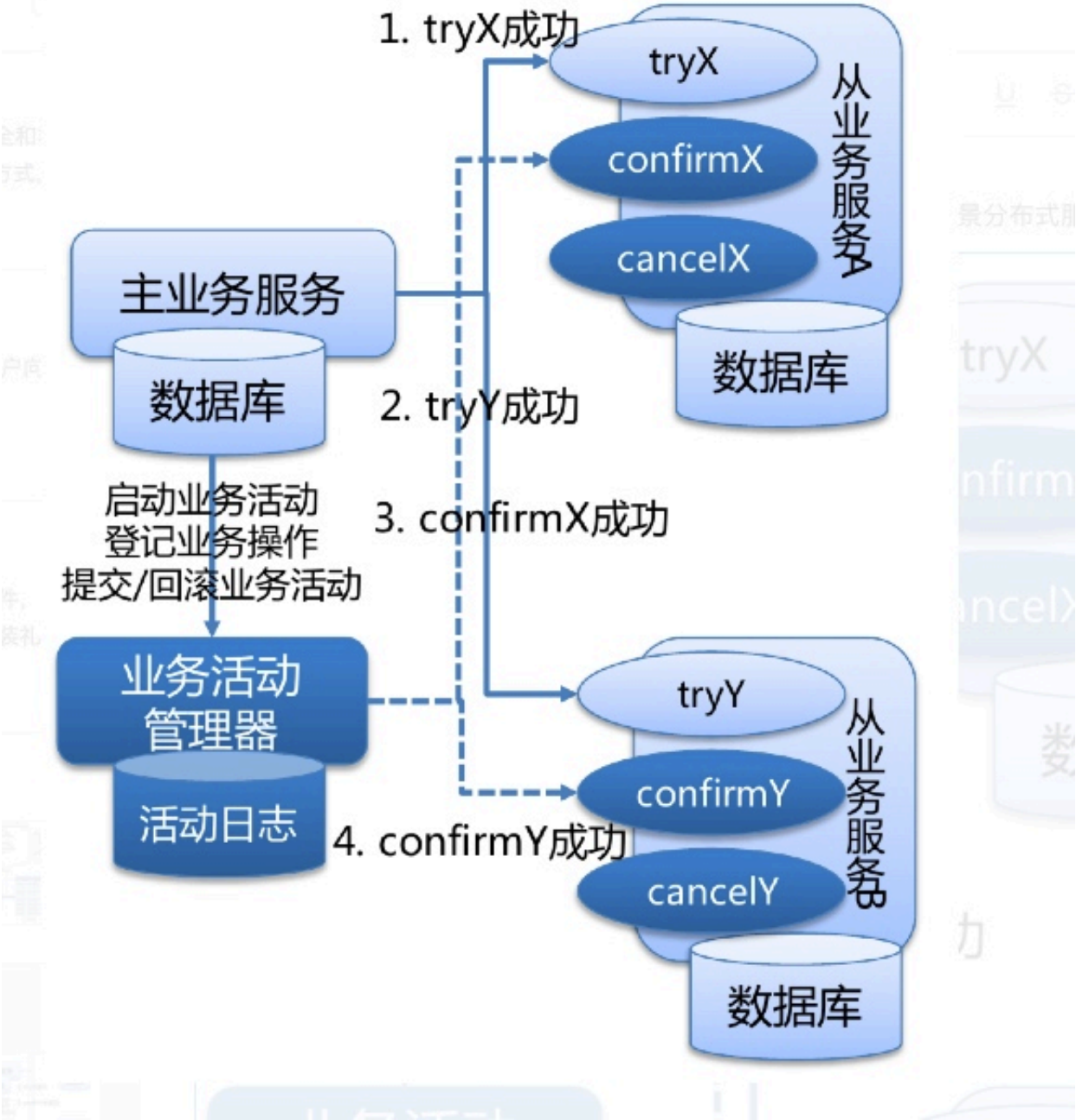
- ACID
  原子性(Atomicity)，一致性(Consistency)，隔离性(Isolation)，持久性(Durability).

- 实现理论
  分布式事务两种处理方式

  - 2PC
    分两个阶段提交,提交过程中会锁住过多的资源



  - TCC
    基于补偿型事务的AP系统的一种实现, 具有最终一致性,使用场景分布式服务系统



- 具体实现
  多阶段提交就可能用到分布式锁：redis，zk都可以实现分布式锁来保证事务的完整性；
  服务的事务补偿，就是各自服务负责自己的补偿，通讯可以用远程调用也可以用MQ消息，大多数情况用MQ，MQ的异步性可以保证服务之间独立处理能力。

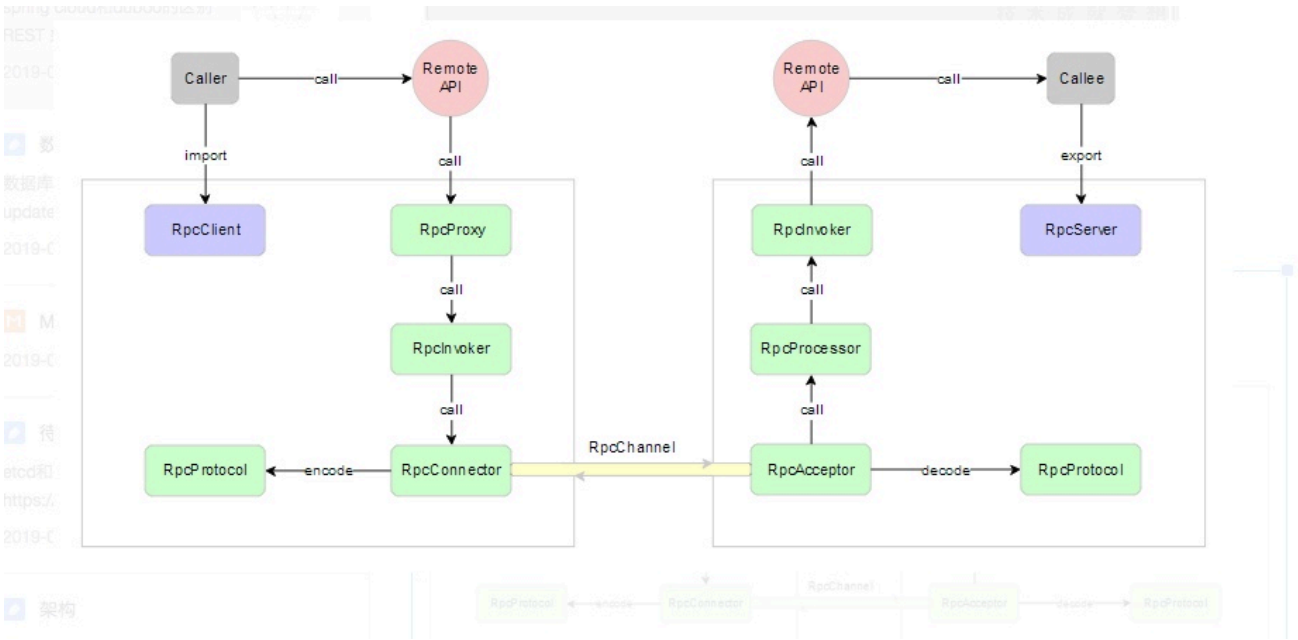
15.RPC和Restful

- 区别
  rpc例如thrift 二进制，4层传输节省宽带，无法穿越防火墙
  rpc使用上客户端和请求端要求一致，请求问题减少，而且不需要关心网络问题。
  restful不需要客户端，直接通过url操作资源
  项目使用中Restful和RPC定义一定边界。

- Restful

一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件.
直接通过url调用，同样的url通过请求方式区别操作类型
post put get delete

- RPC实现



RpcServer
负责导出（export）远程接口
RpcClient
负责导入（import）远程接口的代理实现
RpcProxy
远程接口的代理实现
RpcInvoker
客户方实现：负责编码调用信息和发送调用请求到服务方并等待调用结果返回
服务方实现：负责调用服务端接口的具体实现并返回调用结果
RpcProtocol
负责协议编/解码
RpcConnector
负责维持客户方和服务方的连接通道和发送数据到服务方
RpcAcceptor
负责接收客户方请求并返回请求结果
RpcProcessor
负责在服务方控制调用过程，包括管理调用线程池、超时时间等
RpcChannel
数据传输通道

- duboo
duboo底层是RPC实现，用zk做注册服务中心

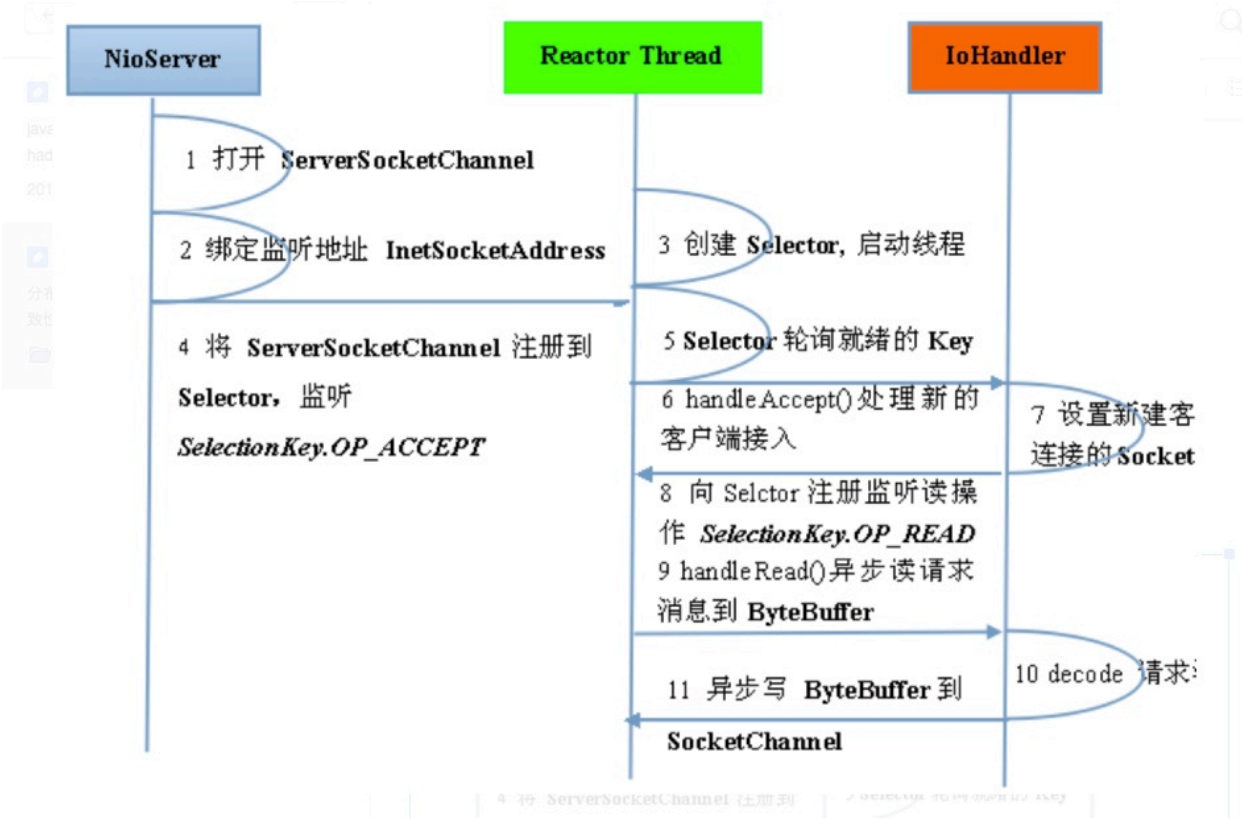
最近dubbo 3.0 启动，最好能看看。

dubbo的十层

1. 服务接口层（Service）：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。
    1. 配置层（Config）：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。
    2. 服务代理层（Proxy）：服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。
    3. 服务注册层（Registry）：封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。
    4. 集群层（Cluster）：封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。
    5. 监控层（Monitor）：RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory、Monitor和MonitorService。
    6. 远程调用层（Protocol）：封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。
    7. 信息交换层（Exchange）：封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。
    8. 网络传输层（Transport）：抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel、Transporter、Client、Server和Codec。
    9. 数据序列化层（Serialize）：可复用的一些工具，扩展接口为Serialization、ObjectInput、ObjectOutput和ThreadPool。

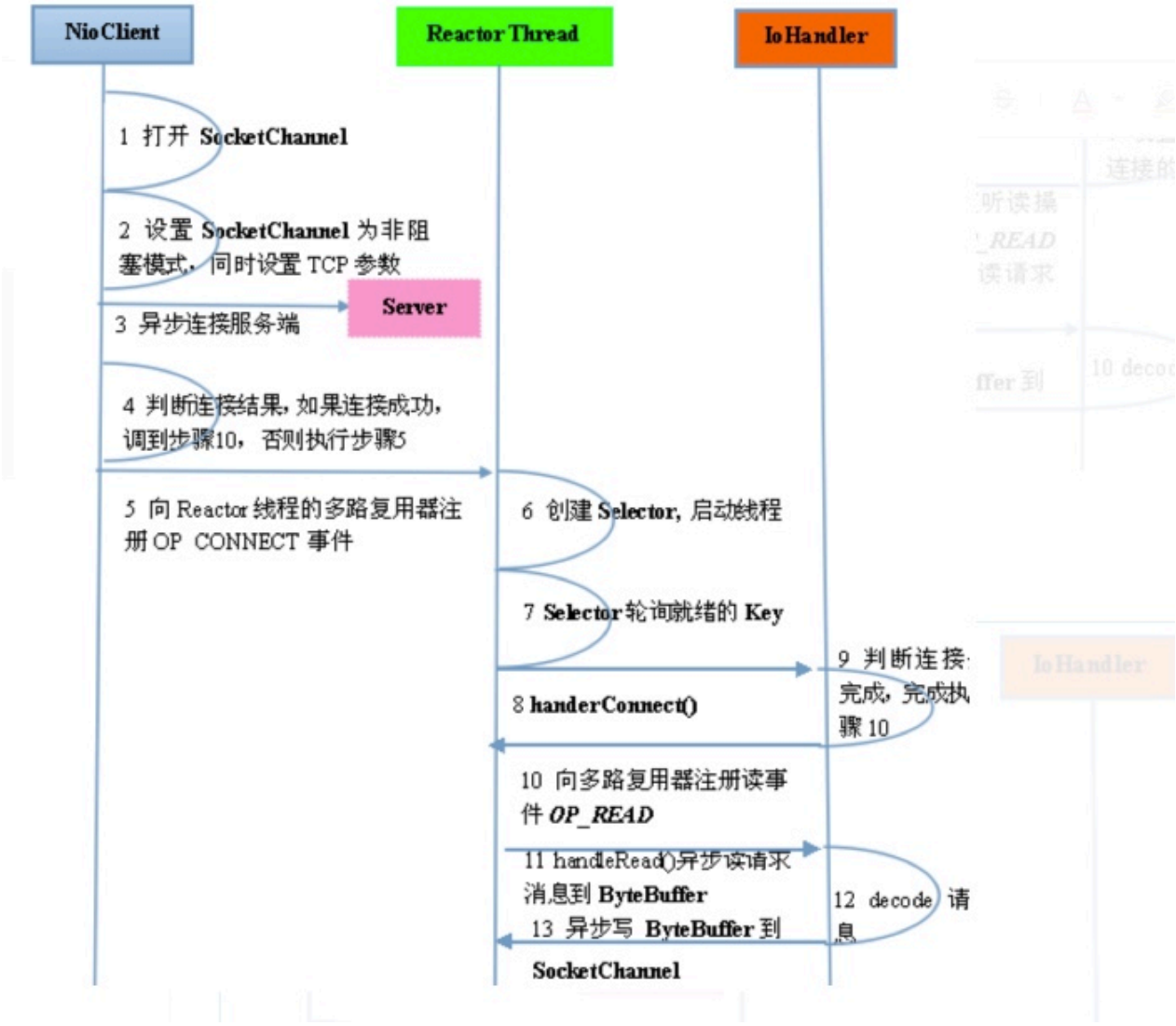
16.netty原理

- 处理流程
    - 服务端



打开socketchannel,监听地址,
创建selector,selector指定到socketchannl
handle 处理业务逻辑
去取handle 连接,
向selector异步读取消息 bytebuffer, decode ,
异步返回写入到socketchannel

- 客户端



打开socketchannel,设置参数,连接server,创建selector,启动线程,轮询

- 零拷贝
  - 介绍
    减少了从核心内存到用户内存的操作
  - DIRECT BUFFERS 直接使用堆外内存,不进行缓存区的二次拷贝;
  - transferTo 直接将缓冲区的内容推向buffer;
  - 组合buffer可以聚合多个ByteBuffer;
  - 内存池ByteBuf
    提供了多个内存管理策略，mmap，sendfile 等
- Reactor
  - 单线程模型
    nio采用的非阻塞,单路reacter理论上够用,但是一旦连接增多,出现过多的timeout,timeout再重试,会出现大量的积压;
    单线程处理不了过多的链路,无法满足海量消息处理;

- 多线程模式
  由线程Acceptor 监听服务,类似jdk线程池方式 一个队列后边多个线程,线程对请求做处理;

- 主从模式
  Acceptor线程池仅仅只用于客户端的登陆、握手和安全认证，一旦链路建立成功，就将链路注册到后端subReactor线程池的IO线程上，由IO线程负责后续的IO操作;
  Acceptor把建立的socketchannel注册到线程池.

17.zk原理

- 一致性算法
  Paxos和ZAB原理
  zk保证一致性的复制过程算法是ZAB,leader选举叫Paxos

  - zk如何保持一致性
    1个leader多个follower,client可以连接到任意机器leader或者follower工作,leader挂了后选举新的leader
    集群中超过半数的存活机器才可以工作(6台只能有2台坏掉,3台工作没有超过半数)
  - 一致性基本保障
    1:全局串行化操作，两个安全属性:
    全序,a在b之前执行,
    因果顺序 ,如果a导致了b发送,并且一起发送则a一直b前
    2:保证同一客户端的指令被FIFO(先进先出)执行
    当follower接受到更新请求后会先交给leader,由leader去保存,leader串行话(类似二阶段提交)去更新follower 超过半数则成功;
    读请求如果到follower获取不到回去leader找.

- ZAB

  - 角色
    Leader，Follower，Observer
  - 模式
    恢复模式，广播模式，同步模式
  - 数据
    zxid、epoch 与 xid

- 选举机制

- 参考文章

https://blog.csdn.net/liuj2511981/article/details/42460069

算法3题

18.求平方根

二分法、牛顿迭代法

最下边是最优解，动态规划

```
public static void main(String[] args) {
        long start = System.currentTimeMillis();
        double target=9876543212345d;
        double result =sqrt(target);
        System.out.println("sqrt耗时: "+(System.currentTimeMillis()-start)+"
,result:"+result);

        start=System.currentTimeMillis();
        result =SqrtByBisection(target, 0);
        System.out.println("SqrtByBisection耗时: "+(System.currentTimeMillis
()-start)+",result:"+result);

        start=System.currentTimeMillis();
        result = SqrtByNewton(target, 0);
        System.out.println("SqrtByNewton耗时: "+(System.currentTimeMillis()-
start)+",result:"+result);

    }

    //牛顿迭代法
    public static double SqrtByNewton(double target,double eps){
        double Xa=target,Xb;
        do {
            Xb = Xa;
            Xa = (Xa+target/Xa)/2;
        } while (fabsf(Xa,Xb)>eps);
        return Xa;
    }

    //二分法  精度是指两次mid值的差值
    public static double SqrtByBisection(double target,double eps){
        double min=1,max=target;
        double mid =(min+max)/2;
```

```
        double anMid;
        do {
            if(mid*mid>target){
                max=mid;
            }else{
                min=mid;
            }
            anMid=mid;
            mid=(max+min)/2;
        } while (fabsf(anMid,mid)>eps);
        return mid;
    }


    public static double sqrt(double d){
        double a = 0.1;
        double x1,x2=0;
        while (a*a<=d) {
            a+=0.1;
        }
         x1=a;
        for (int i = 0; i < 10; i++) {
            x2=d;
            x2/=x1;
            x2+=x1;
            x2/=2;
            x1=x2;
        }
        return x2;
    }

    public static double fabsf(double a,double b){
        if(a>b){
            return a-b;
        }else
            return b-a;
    }
```

19.判断循环链表

最优的时间复杂度，两个指针，一个快一个慢，如果遇到了就是环形。

```
public boolean isLoop(Node head){
        Node slow = head;
```

```
        Node fast = head;
        while(fast!=null && fast.next!=null)
        {
            slow = slow.next;
            fast = fast.next.next;
            if(slow==fast)
                return true;
        }

        return false;
    }
```

20.能否快速找出一个数组中的两个数字的和

最优解法，先排好序，然后从两个中位数开始相加，如果小了大的一边的数字往大的方向移动，小了小的数组往小的方向移动。时间复杂度是O（n）

```
int getSumNum(int[] arr,int Sum),    //arr为数组，Sum为和
{
    int i,j;
    for(i = 0, j = n-1; i < j ; )
    {
        if(arr[i] + arr[j] == Sum)
            return ( i , j );
        else if(arr[i] + arr[j] < Sum)
            i++;
        else
            j--;
    }
    return ( -1 , -1 );
}
```