基础

1. String 和 StringBuffer、StringBuilder 的区别是什么？String 为什么是不可变的?

    - 可变性
      简单的来说：String 类中使用 final 关键字字符数组保存字符串， private final char value[] ，所以 String 对象是不可变的。而StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串 char[]value 但是没有用 final 关键字修饰，所以这两种对象都是可变的。
    - 线程安全性
      String 中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共 方法。StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对 方法进行加同步锁，所以是非线程安全的。

    - 性能
      每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。 StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 StirngBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

    - 对于三者使用的总结：

        a. 操作少量的数据 = String
        b. 单线程操作字符串缓冲区下操作大量数据 = StringBuilder
        c. 多线程操作字符串缓冲区下操作大量数据 = StringBuffer

2. == 与 equals

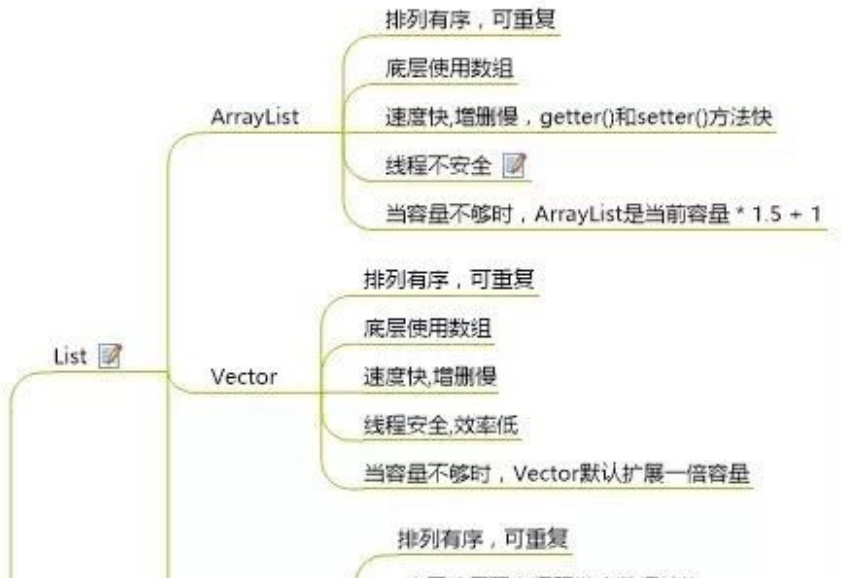
   == : 它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象。(基本数据类型==比较的是值，引用数据类型==比较的是内存地址)

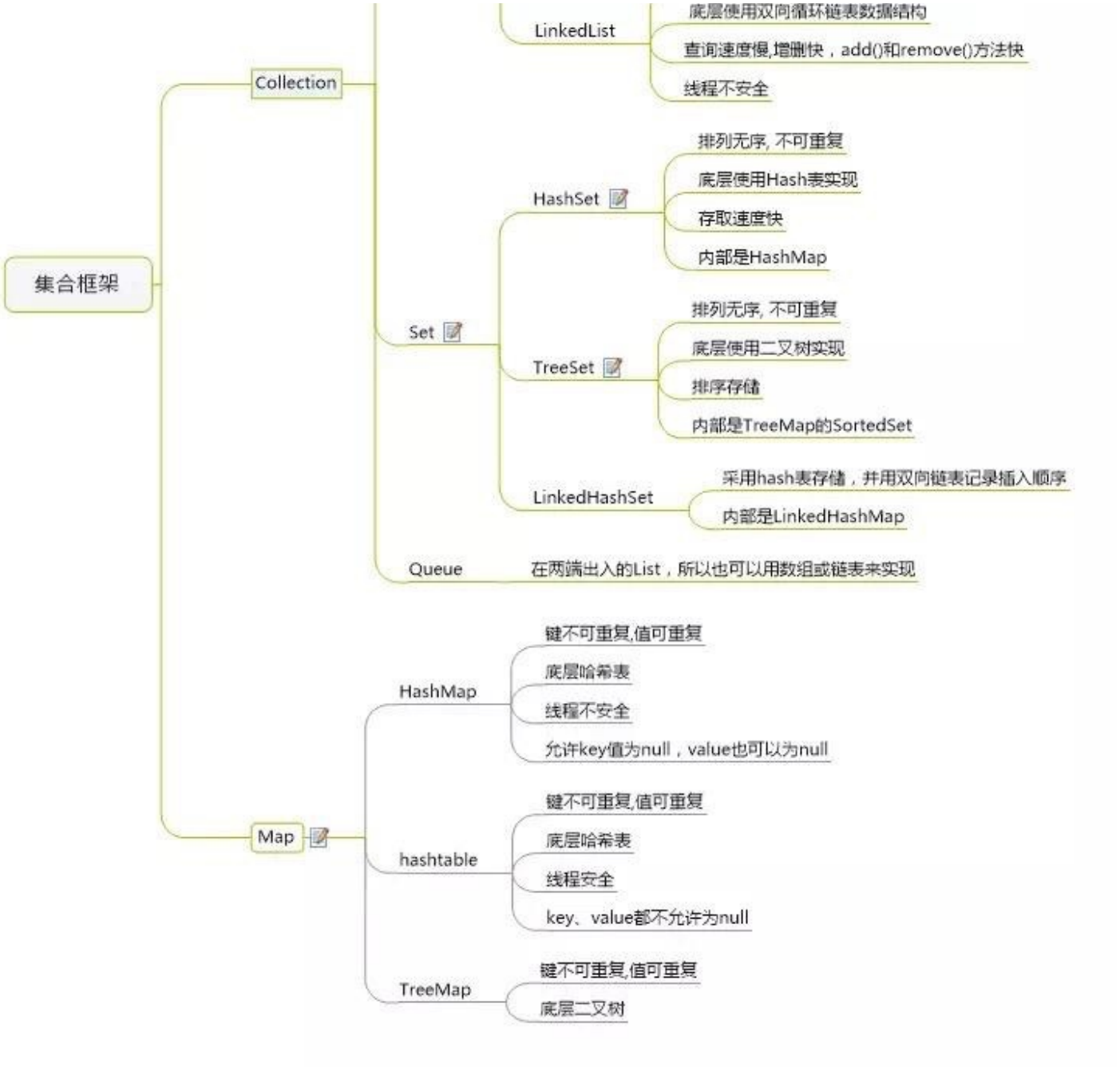
   equals() : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：
   情况1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。
   情况2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来两个对象的内容相等；若它们的内容相 等，则返回 true (即，认为这两个对象相等)。

   Integer
   Integer类中有一个静态内部类IntegerCache，在IntegerCache类中有一个Integer数组，用以缓存当数值范围为-128~127时的Integer对象

```
public static void main(String[] args) {
        Integer i1 = 127;
        Integer i2 = 127;
        System.err.println(i1 == i2);
        i1 = 128;
        i2 = 128;
        System.err.println(i1 == i2);
    }
```

集合





1. Arraylist 与 LinkedList 异同
   i. 是否保证线程安全： ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
   ii. 底层数据结构： Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向链表数据结构（JDK1.6之 前为循环链表，JDK1.7取消了循环。注意双向链表和双向循环链表的区别
   iii. 插入和删除是否受元素位置的影响： ① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素 位置的影响。 比如：执行 add(E e) 方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种 情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（ add(int index, E element) ）时 间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向

前移一位的操作。 ② LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是 近似 O（1）而数组为近似 O（n）。

iv. 是否支持快速随机访问： LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通 过元素的序号快速获取元素对象(对应于 get(int index) 方法)。

v. 内存空间占用： ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空 间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数 据）。

2. HashMap（数组+链表+红黑树）

HashMap 根据键的 hashCode 值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。 HashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null。HashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致。 如果需要满足线程安全， 可以用 Collections 的 synchronizedMap 方法使HashMap 具有线程安全的能力， 或者使用 ConcurrentHashMap。

HashMap 里面是一个数组，然后数组中每个元素是一个单向链表。capacity：当前数组容量，始终保持 2^n，可以扩容，扩容后数组大小为当前的 2 倍。
loadFactor：负载因子，默认为 0.75。
threshold：扩容的阈值，等于 capacity * loadFactor

Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链表+红黑树组成。根据 Java7 HashMap 的介绍，我们知道，查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后的话， 需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决于链表的长度，为 O(n)。为了降低这部分的开销，在 Java8 中，当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)。

3. ConcurrentHashMap和Collections.synchronizedMap(会同步整个对象)

4. ConcurrentHashMap

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，Segment 通过继承ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。
concurrencyLevel：并行级别、并发数、Segment 数，怎么翻译不重要，理解它。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在

初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。

JVM

1. JVM内存模型
    i. 程序计数器：当前线程执行的字节码的位置指示器
    ii. 虚拟机栈(VM Strack)：方法本身和操作数
    iii. Heap：保存对象实例的属性值，属性的类型和对象本身的类型标记等，并不保存对象的方法（以帧栈的形式保存在Stack中）
    iv. Method Area：有类的版本、字段、方法、接口等先关信息描述外，还有常量池（Constant Pool Table）信息
    v. 本地方法栈Native Method Stack： 与VM Strack相似，VM Strack为JVM提供执行JAVA方法的服务，Native Method Stack则为JVM提供使用native 方法的服务。
2. 对象头
    i. 对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。
    ii. 在Hotspot JVM中，32位机器下，Integer对象的大小是int的几倍?
    Integer只有一个int类型的成员变量value，所以其对象实际数据部分的大小是4个字节，markword 4字节，指针4字节，然后再在后面填充4个字节达到8字节的对齐，所以可以得出Integer对象的大小是16个字节。
3. 偏向锁就是在锁对象的对象头中有个ThreaddId字段，这个字段如果是空的，第一次获取锁的时候，就将自身的ThreadId写入到锁的ThreadId字段内，将锁头内的是否偏向锁的状态位置1.这样下次获取锁的时候，直接检查ThreadId是否和自身线程Id一致， 如果一致，则认为当前线程已经获取了锁，因此不需再次获取锁，略过了轻量级锁和重量级锁的加锁阶段。提高了效率。
4. 类加载
    i. 加载：类的装载指的是将类的.class文件中的二进制数据读入到内存中
    ii. 类的连接
        a. 验证：验证被加载后的类是否有正确的结构
        b. 准备：为类的静态变量（static filed）在方法区分配内存，并赋默认初值（0值或null值）。如static int a = 100;
            a. 静态变量a就会在准备阶段被赋默认值0。
            b. 对于一般的成员变量是在类实例化时候，随对象一起分配在堆内存中。
            c. 静态常量（static final filed）会在准备阶段赋程序设定的初值
        c. 解析：将类的二进制数据中的符号引用换为直接引用。
    iii. 类的初始化：为静态变量赋程序设定的初值。

5. 双亲委派
   引导（Bootstrap）类加载器、扩展（Extension）类加载器、系统（System）类加载器（也称应用类加载器）

   双亲委派模型的破坏者-线程上下文类加载器

6. Full GC触发条件：
   （1）调用System.gc时，系统建议执行Full GC，但是不必然执行
   （2）老年代空间不足
   （3）方法去空间不足
   （4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存
   （5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

7. JVM分代

   i. JVM把年轻代分为了三部分：1个Eden区和2个Survivor区（分别叫from和to）。默认比例为8：1
   ii. 在GC开始的时候，对象只会存在于Eden区和名为“From”的Survivor区，Survivor区“To”是空的。紧接着进行GC，Eden区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次GC后，Eden区和From区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次GC前的“From”，新的“From”就是上次GC前的“To”。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。

8. CMS+ParNew

9. CMS：标记清理，老年代回收器
   优点：GC时间短，缺点：内存碎片化严重
   CMS 处理过程：

   i. 初始标记 ,会导致swt；标记老年代中所有的GC Roots对象
   ii. 并发标记，与用户线程同时运行；负责将引用发生改变的Card标记为Dirty状态
   iii. 预清理，与用户线程同时运行；处理前一个阶段因为引用关系改变导致没有标记到的存活对象
   iv. 重新标记，会导致swt；标记整个年老代的所有的存活对象

v. 并发清除，与用户线程同时运行；清除那些没有标记的对象并且回收空间

并发模式失败(concurrent mode failure)：新生代发生垃圾回收，同时老年代又没有足够的空间容纳晋升的对象时，CMS 垃圾回收就会退化成单线程的Full GC。所有的应用线程都会被暂停，老年代中所有的无效对象都被回收

晋升失败(promotion failed)：当新生代发生垃圾回收，老年代有足够的空间可以容纳晋升的对象，但是由于空闲空间的碎片化，导致晋升失败，此时会触发单线程且带压缩动作的Full GC

10. G1

    - 相比CMS

      a. 在压缩空间方面有优势
      b. 可预期的GC停顿周期；防止高并发下应用雪崩现象
      c. G1会在Young GC中使用、而CMS只能在O区使用

    - 过程（两种GC模式都STW）

      a. Young GC：选定所有年轻代里的Region。通过控制年轻代的region个数，即年轻代内存大小，来控制young GC的时间开销
      b. Mixed GC：选定所有年轻代里的Region，外加根据global concurrent marking统计得出收集收益高的若干老年代Region。在用户指定的开销目标范围内尽可能选择收益高的老年代Region。

    - full GC

      a. 并发模式失败
      b. 晋升失败
      c. 巨型对象分配失败：如果一个对象占用的空间超过了分区容量50%以上，G1收集器就认为这是一个巨型对象

    - 优化

      a. 调大Region区域避免巨型对象分配
      b. 设置STW工作线程
      c. 设置并发标记周期的java堆占用率
      d. 去除年轻代活年轻代相关设置，否则会覆盖暂停时间目标

11. 对象分配

i. 如果确定一个对象的作用域不会逃逸出方法之外，那可以将这个对象分配在栈上
ii. JVM使用TLAB来避免多线程冲突，在给对象分配内存时，每个线程使用自己的TLAB，这样可以避免线程同步，提高了对象分配的效率。
iii. 大对象直接进入老年代
iv. 长期存活的对象将进入老年代

12. CPU占用过高问题排查

MySQL

- MySQL索引
    - 最左前缀原则
    - InnoDB聚集索引
        - 按主键构造一颗B+数，叶子节点存放整张表的行记录数据，每张表只能有一个聚集索引
        - 当定义主键后，InnoDB会利用主键来生成其聚簇索引；如果没有主键，InnoDB会选择一个非空的唯一索引来创建聚簇索引；如果这也没有，InnoDB会隐式的创建一个自增的列来作为聚簇索引。
    - 辅助索引
        - 叶子节点并不包含行记录的全部数据，叶子节点除了包含键值以外，每个叶子节点中的索引行中还包含了一个书签，辅助索引的存在并不影响数据在聚集索引中的组织，因此每张表上可以有多个辅助索引。当通过辅助索引来寻找数据时，Innodb存储引擎会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键，然后再通过主键索引来找到一个完整的行记录。
    - 为什么用B+树
        - 因为哈希表的特点就是可以快速的精确查询，但是不支持范围查询
        - 用B+树这种数据结构作为索引，可以提高查询索引时的磁盘IO效率，并且可以提高范围查询的效率，并且B+树里的元素也是有序的。
    - 一个B+树的一个节点大小为多大
        - Mysql的Innodb引擎中一页的默认大小是16k（如果操作系统中一页大小是4k，那么Mysql中1页=操作系统中4页），可以使用命令SHOW GLOBAL STATUS like

'Innodbpagesize'; 查看。一个节点为一页

- 为什么一个节点为1页

  - B+树中的一个节点存储的内容是：非叶子节点：主键+指针；叶子节点：数据
  - 那么，假设我们一行数据大小为1K，那么一页就能存16条数据，也就是一个叶子节点能存16条数据；再看非叶子节点，假设主键ID为bigint类型，那么长度为8B，指针大小在Innodb源码中为6B，一共就是14B，那么一页里就可以存储16K/14=1170个(主键+指针)，那么一颗高度为2的B+树能存储的数据为：117016=18720条，一颗高度为3的B+树能存储的数据为：11701170*16=21902400（千万级条）。所以在InnoDB中B+树高度一般为1-3层，它就能满足千万级的数据存储。在查找数据时一次页的查找代表一次IO，所以通过主键索引查询通常只需要1-3次IO操作即可查找到数据。所以也就回答了我们的问题，1页=16k这么设置是比较合适的，是适用大多数的企业的，当然这个值是可以修改的，所以也能根据业务的时间情况进行调整。

- SQL优化

  - 走索引、少join、少排序
  - join用小表驱动大表，被驱动表的join字段上建立索引
  - 用join替代子查询
  - 尽量避免 select *
  - 尽量少 or，使用 union all 或者是union
  - union 和 union all 的差异主要是前者需要将两个(或者多个)结果集合并后再进行唯一性过滤操作，这就会涉及到排序，增加大量的 CPU 运算
  - 尽量早过滤
  - 避免类型转换
  - select * from A where cc in (select cc from B) 如果表A较大,那么in比exists的效率要高
  - 多用explain

- 分库分表

  i. 分库：多数据源
  ii. 分表：垂直分表 水平分表

- 分库分表后面临的问题
  分布式事务、跨库join、全局ID问题

- 中间件 mycat

Redis

1. redis持久化 AOF与RDB
2. redis分布式锁 setnx+lua
    i. set命令要用set key value px milliseconds nx；
    ii. value要具有唯一性；
    iii. 释放锁时要验证value值，不能误解锁；
3. 定时刷新过期时间保障过期时间大于业务执行时间
4. redlock
    i. 获取当前Unix时间，以毫秒为单位。
    ii. 依次尝试从5个实例，使用相同的key和具有唯一性的value（例如UUID）获取锁。当向Redis请求获取锁时，客户端应该设置一个网络连接和响应超时时间，这个超时时间应该小于锁的失效时间。例如你的锁自动失效时间为10秒，则超时时间应该在5-50毫秒之间。这样可以避免服务器端Redis已经挂掉的情况下，客户端还在死死地等待响应结果。如果服务器端没有在规定时间内响应，客户端应该尽快尝试去另外一个Redis实例请求获取锁。
    iii. 客户端使用当前时间减去开始获取锁时间（步骤1记录的时间）就得到获取锁使用的时间。当且仅当从大多数（N/2+1，这里是3个节点）的Redis节点都取到锁，并且使用的时间小于锁失效时间时，锁才算获取成功。
    iv. 如果取到了锁，key的真正有效时间等于有效时间减去获取锁所使用的时间（步骤3计算的结果）。
    v. 如果因为某些原因，获取锁失败（没有在至少N/2+1个Redis实例取到锁或者取锁时间已经超过了有效时间），客户端应该在所有的Redis实例上进行解锁（即便某些Redis实例根本就没有加锁成功，防止某些节点获取到锁但是客户端没有得到响应而导致接下来的一段时间不能被重新获取锁）。
5. 缓存更新
    i. 先更新数据库，再删缓存
    ii. 缓存删除失败可通过添加队列或binlog解决
6. 缓存穿透 布隆过滤器或缓存空值
7. 缓存雪崩
    i. 不同的key，设置不同的过期时间
    ii. 二级缓存
8. 热点key
    i. 预热
    ii. 将热点key分散为多个子key

kafka

- 基本概念
    - 每条发布到Kafka的消息都有个类别(Topic).
    - 一个topic有多个partition，producer根据key hashcode对分区数取余
    - Producer发布消息的对象称之为主题生产者(Kafka topic producer)
    - Consumer订阅消息并处理发布的消息的种子的对象称之为主题消费者(consumers)
    - Broker已发布的消息保存在一组服务器中，称之为Kafka集群。集群中的每一个服务器都是一个代理(Broker). 消费者可以订阅一个或多个主题（topic），并从Broker拉数据，从而消费这些已发布的消息。
- 消息重复消费：增加去重表
- 消息丢失
    - acks=-1: 等待所有ISR接收到消息后再给Producer发送Response，还要保证ISR的大小大于等于2
    - 配置最少同步副本
    - 重试与错误处理
- 消费者可靠性
    - 偏移量提交：正常情况异步提交，异常情况同步提交
    - 再均衡：分区撤销前提交偏移量
    - 遇到错误重试时候：
        - 把还没处理好的消息保存到缓存里
        - 把错误写入一个独立的主题

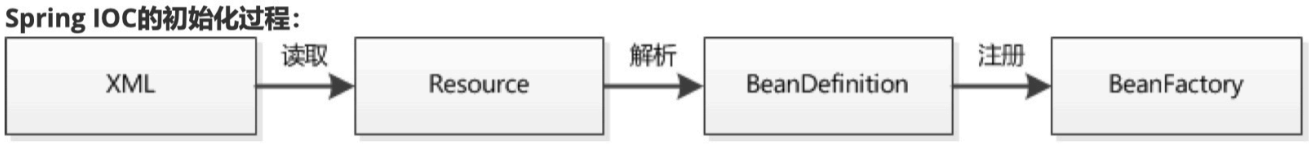
Spring

- Bean的作用域

类别	说明
singleton	在Spring IoC容器中仅存在一个Bean实例，Bean以单例方式存在，默认值
prototype	每次从容器中调用Bean时，都返回一个新的实例，即每次调用getBean()时，相当于执行new XxxBean()
request	每次HTTP请求都会创建一个新的Bean，该作用域仅适用于WebApplicationContext环境
session	同一个HTTP Session 共享一个Bean，不同Session使用不同Bean，仅适用于WebApplicationContext 环境
globalSession	一般用于Portlet应用环境，该作用域仅适用于WebApplicationContext 环境

- AOP
  AOP思想的实现是基于 代理模式 ，
  - 如果目标对象的实现类实现了接口，Spring AOP 将会采用 JDK 动态代理来生成 AOP 代理类；
  - 如果目标对象的实现类没有实现接口，Spring AOP 将会采用 CGLIB 来生成 AOP 代理类——不过这个选择过程 对开发者完全透明、开发者也无需关心。

- IOC



- Spring 事务中的事务传播行为
  支持当前事务的情况：
  - TransactionDefinition.PROPAGATION_REQUIRED： 如果当前存在事务，则加入该事务；如果当前没有事 务，则创建一个新的事务。
  - TransactionDefinition.PROPAGATION_SUPPORTS： 如果当前存在事务，则加入该事务；如果当前没有事 务，则以非事务的方式继续运行。
  - TransactionDefinition.PROPAGATION_MANDATORY： 如果当前存在事务，则加入该事务；如果当前没有 事务，则抛出异常。（mandatory：强制性）

  不支持当前事务的情况：

- TransactionDefinition.PROPAGATION_REQUIRES_NEW： 创建一个新的事务，如果当前存在事务，则把当 前事务挂起。
- TransactionDefinition.PROPAGATION_NOT_SUPPORTED： 以非事务方式运行，如果当前存在事务，则把 当前事务挂起。
- TransactionDefinition.PROPAGATION_NEVER： 以非事务方式运行，如果当前存在事务，则抛出异常。

  其他情况：

- TransactionDefinition.PROPAGATION_NESTED： 如果当前存在事务，则创建一个事务作为当前事务的嵌套 事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION_REQUIRED。

- Spring Boot 的自动配置是如何实现的?
  Spring Boot 项目的启动注解是：@SpringBootApplication，其实它就是由下面三个注解组成的：
  @Configuration
  @ComponentScan
  @EnableAutoConfiguration
  其中 @EnableAutoConfiguration 是实现自动配置的入口，该注解又通过 @Import 注解导入了AutoConfigurationImportSelector，在该类中加载 META-INF/spring.factories 的配置信息。然后筛选出以 EnableAutoConfiguration 为 key 的数据，加载到 IOC 容器中，实现自动配置功能！

Dubbo

- Dubbo启动时如果依赖的服务不可用会怎样?
  Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，默认 check="true"，可以通过 check="false" 关闭检查。

- Dubbo推荐使用什么序列化框架，你知道的还有哪些?
  推荐使用Hessian序列化，还有Duddo、FastJson、Java自带序列化。

- Dubbo默认使用的是什么通信框架，还有别的选择吗?
  Dubbo 默认使用 Netty 框架，也是推荐的选择，另外内容还集成有Mina、Grizzly。

- Dubbo有哪几种集群容错方案，默认是哪种?

集群容错方案	说明
Failover Cluster	失败自动切换，自动重试其它服务器（默认）
Failfast Cluster	快速失败，立即报错，只发起一次调用
Failsafe Cluster	失败安全，出现异常时，直接忽略
Failback Cluster	失败自动恢复，记录失败请求，定时重发
Forking Cluster	并行调用多个服务器，只要一个成功即返回
Broadcast Cluster	广播逐个调用所有提供者，任意一个报错则报错

- Dubbo有哪几种负载均衡策略，默认是哪种?

负载均衡策略	说明
Random LoadBalance	随机，按权重设置随机概率（默认）
RoundRobin LoadBalance	轮询，按公约后的权重设置轮询比率
LeastActive LoadBalance	最少活跃调用数，相同活跃数的随机
ConsistentHash LoadBalance	一致性 Hash，相同参数的请求总是发到同一提供者