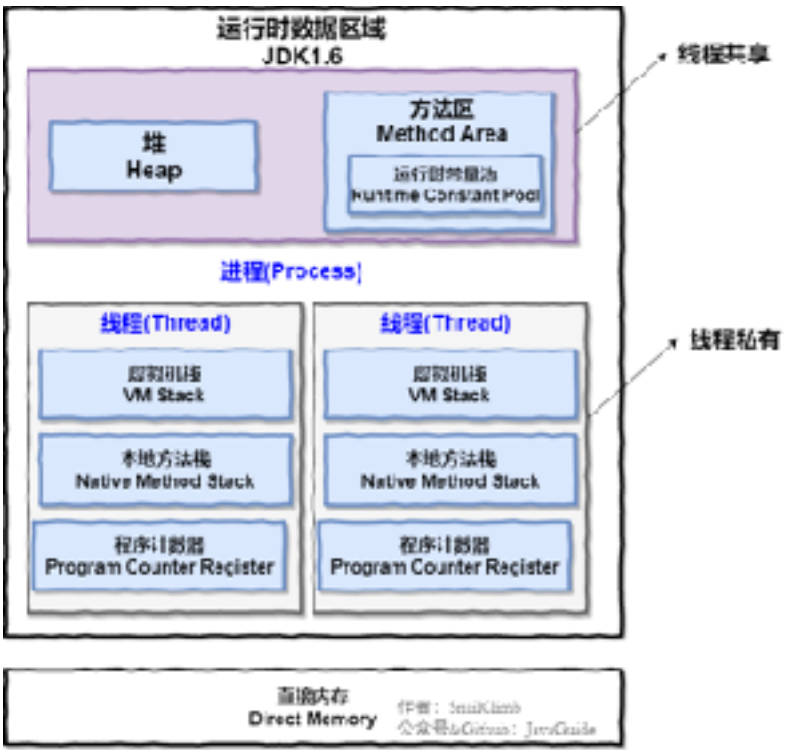
17年进入华为，试用期过后主要做后端，spring cloud + k8s+ spark，由于后期spring cloud用的少了，所以不太熟，spring这方面还可以，spark调优

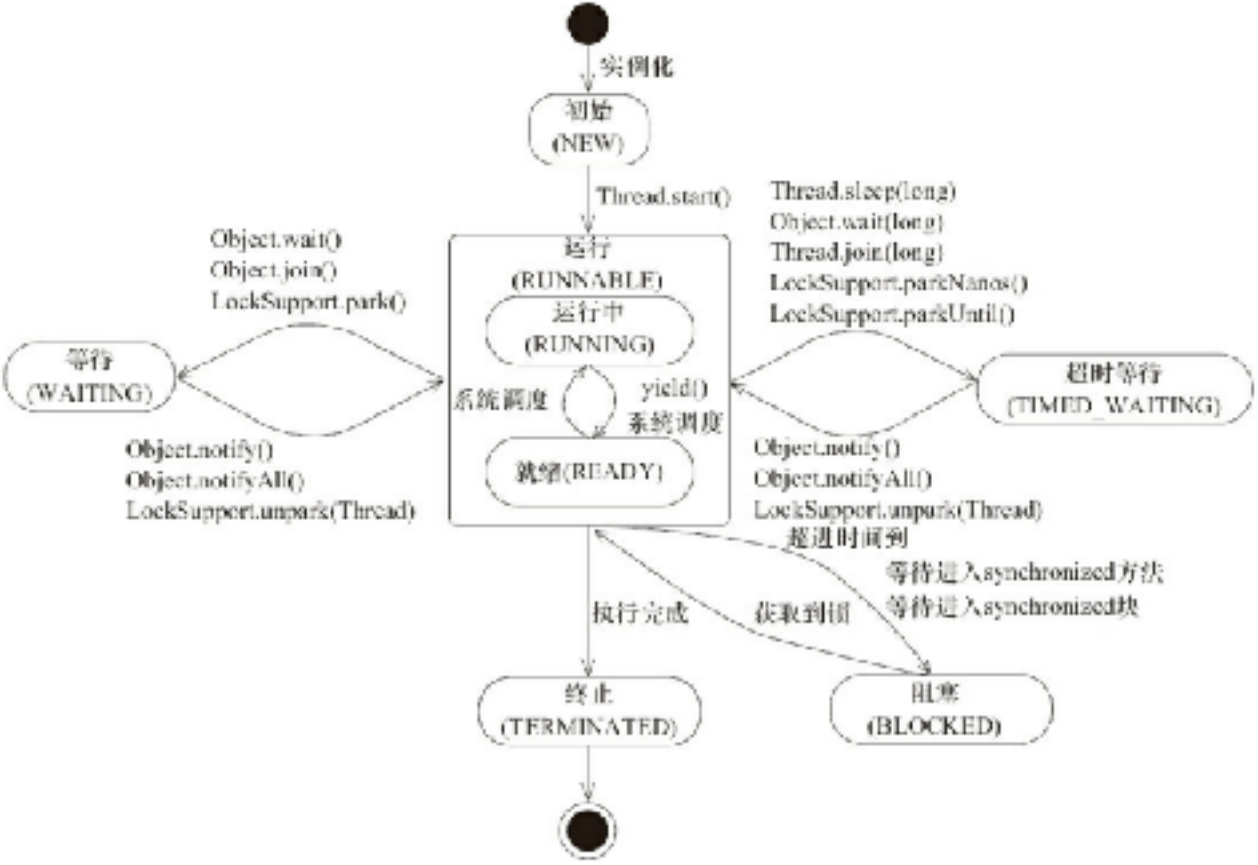
1.基础平台性的 在西安成都两地，上海这边以实现业务为主，整体水平一般
2.对华为比较认可，但部门是以业务为导向，不重视技术，部门领导的也都是业务出生，技术人员上升慢。
3.公司发展时间比较长，会遇到不少历史遗留的问题，

一.java方面

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程
线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享进程的堆和方法区资源，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。



状态名称	说　明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕



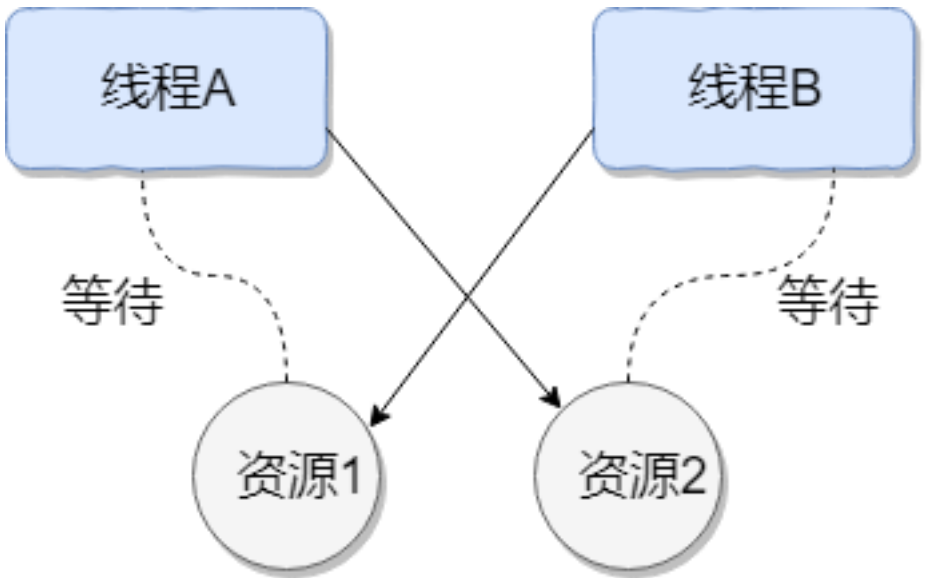
线程创建之后它将处于 NEW（新建） 状态，调用 start() 方法后开始运行，线程这时候处于 READY（可运行） 状态。可运行状态的线程获得了 cpu 时间片（timeslice）后就处于 RUNNING（运行）状态。

当线程执行 wait()方法之后，线程进入 WAITING（等待）状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 TIME_WAITING(超时等待) 状态相当于在等待状态的基础上增加了超时限制，比如通过 sleep（long millis）方法或 wait（long millis）方法可以将 Java 线程置于 TIMED WAITING 状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 BLOCKED（阻塞） 状态。线程在执行 Runnable 的run()方法之后将会进入到 TERMINATED（终止） 状态。

死锁

多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



如何避免线程死锁?
a)破坏互斥条件
这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。
b)破坏请求与保持条件
一次性申请所有的资源。
c)破坏不剥夺条件
占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
d)破坏循环等待条件
靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件

为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法?
调用 start 方法方可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。

synchronized用法
修饰实例方法
修饰静态方法
修饰代码块

单例模式

```
public class Singleton {

    private volatile static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
        //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            //类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

谈谈 synchronized和ReentrantLock 的区别
两者都是可重入锁
synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API
ReentrantLock 比 synchronized 增加了一些高级功能
①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）

volatile
每次使用它都到主存中进行读取。

说白了， **volatile** 关键字的主要作用就是保证变量的可见性然后还有一个作用是防止指令重排序。

•volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。

•volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。

线程池

最好通过ThreadPoolExecutor 创建

•降低资源消耗。 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

•提高响应速度。 当任务到达时，任务可以不需要的等到线程创建就能立即执行。

•提高线程的可管理性。 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控

AQS

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中

AQS定义两种资源共享方式

•Exclusive（独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：

公平锁：按照线程在队列中的排队顺序，先到者先拿到锁

非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的

•Share（共享）：多个线程可同时执行

悲观锁与乐观锁

乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以**一般多写的场景下用悲观锁就比较合适**

乐观锁一般会使用版本号机制或CAS算法实现。

堆

此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。

Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC 堆（Garbage Collected Heap）**.从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和

老年代：再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。**进一步划分的目的是更好地回收内存，或者更快地分配内存**

1)BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
2)NIO (New I/O): NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。NIO中的N可以理解为Non-blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。 NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
3)AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

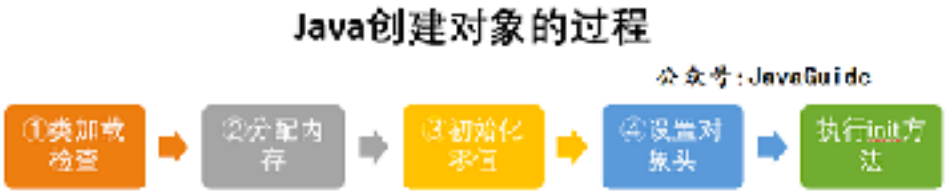
HashMap
所谓 “拉链法” 就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。

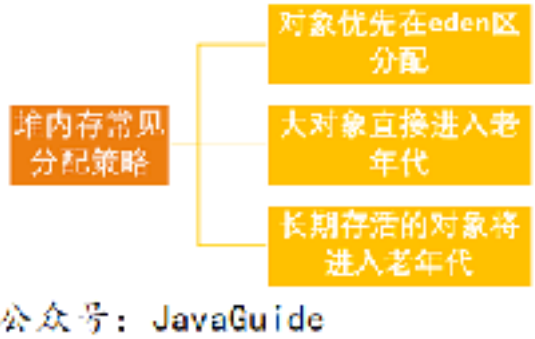
HashMap 的长度为什么是2的幂次方
为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648到2147483647，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数∂法是“ (n - 1) & hash”。（n代表数组长度）。这也就解释了 HashMap 的长度为什么是2的幂次方。
这个算法应该如何设计呢?
我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是2的 n 次方；）。” 并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

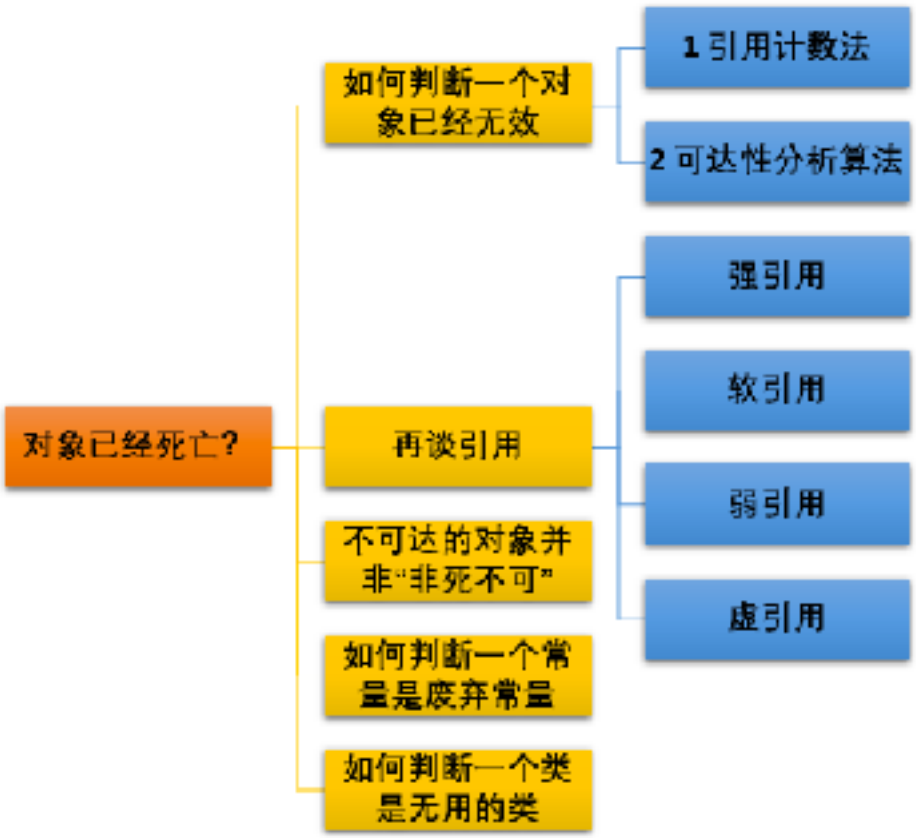
```
    static final int tableSizeFor(int cap) {
        int n = cap - 1;
        n |= n >>> 1;
        n |= n >>> 2;
        n |= n >>> 4;
```

```
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

对象创建 五步







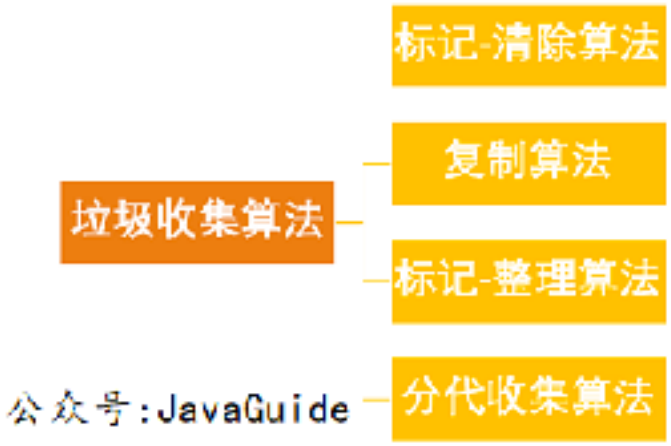
“无用的类”：

•该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。

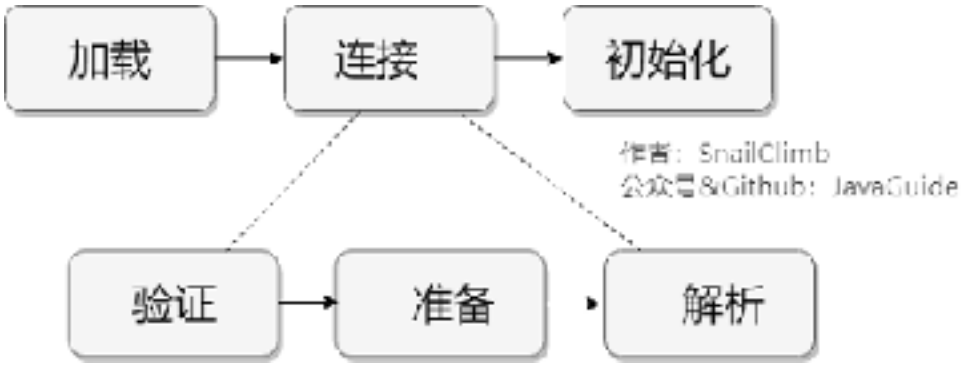
•加载该类的 ClassLoader 已经被回收。

•该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法

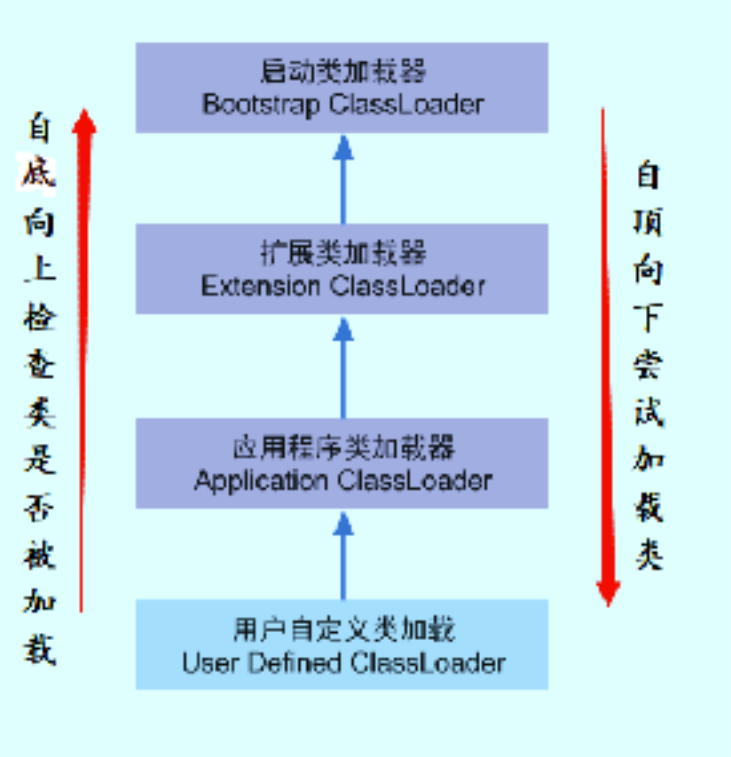
垃圾回收算法



类加载过程



双亲委托模型



双亲委派模型的好处

双亲委派模型保证了Java程序的稳定运行，可以避免类的重复加载（JVM 区分不同类的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类），也保证了 Java 的核心 API 不被篡改。如果没有使用双亲委派模型，而是每个类加载器加载自己的话就会出现一些问题，比如我们编写一个称为 `java.lang.Object` 类的话，那么程序运行的时候，系统就会出现多个不同的 `Object` 类

红黑树

1.每个节点非红即黑；

2.根节点总是黑色的；

3.每个叶子节点都是黑色的空节点（NIL节点）；

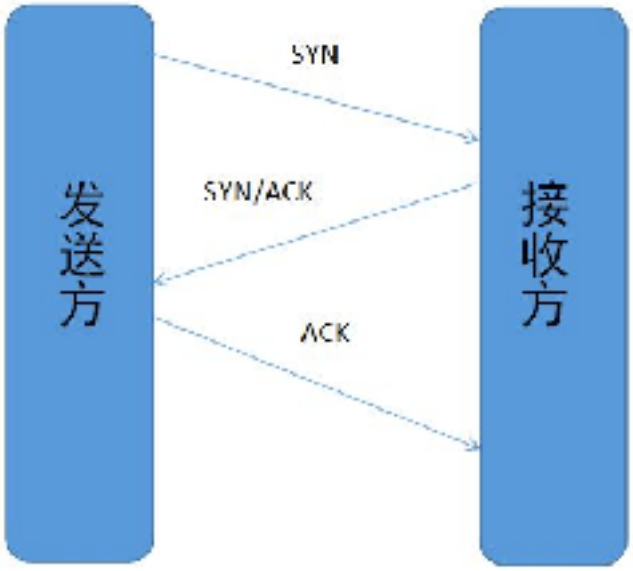
4.如果节点是红色的，则它的子节点必须是黑色的（反之不一定）；

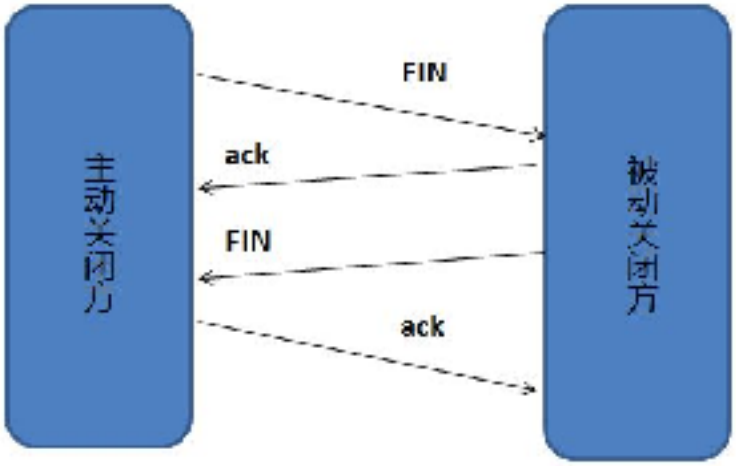
5.从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）。

TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树

简单来说红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。详细了解可以查看 漫画：什么是红黑树？（也介绍到了二叉查找树，非常推荐

二.网络





在浏览器中输入url地址 ->> 显示主页的过程(面试常客)

总体来说分为以下几个过程:

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

Cookie的作用是什么?和Session有什么区别?

Cookie 数据保存在客户端(浏览器端)，Session 数据保存在服务器端。

Cookie 存储在客户端中，而Session存储在服务器上，相对来说 Session 安全性更高。如果使用 Cookie 的一些敏感信息不要写入 Cookie 中，最好能将 Cookie 信息加密然后使用到的时候再去服务器端解密

HTTP 和 HTTPS 的区别?

1.端口： HTTP的URL由“http://”起始且默认使用端口80，而HTTPS的URL由“https://”起始且默认使用端口443。

2.安全性和资源消耗： HTTP协议运行在TCP之上，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。HTTPS是运行在SSL/TLS之上的HTTP协议，SSL/TLS 运行在TCP之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。所以说，HTTP 安全性没有 HTTPS高，但是 HTTPS 比HTTP耗费更多服务器资源。

三.数据库

索引

MySQL索引使用的数据结构主要有**BTree索引** 和 **哈希索引**

事务?

事务是逻辑上的一组操作，要么都执行，要么都不执行

并发事务带来哪些问题?

·脏读（Dirty read）：当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。

·丢失修改（Lost to modify）：指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。

·不可重复读（Unrepeatableread）：指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。

·幻读（Phantom read）：幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读

不可重复读的重点是修改比如多次读取一条记录发现其中某些列的值被修改，幻读的重点在于新增或者删除比如多次读取一条记录发现记录增多或减少了

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

大表优化

1.限定数据的范围
2. 读/写分离
3. 垂直分区
4. 水平分区

索引
是将无序的数据变成有序(相对) 所以能变快

redis
redis 的数据是存在内存中的，所以读写速度非常快，

四. spring + spring cloud

Spring 是一种轻量级开发框架，旨在提高开发人员的开发效率以及系统的可维护性

spring 官网列出的 Spring 的 6 个特征:

- **核心技术**： 依赖注入(DI)，AOP，事件(events)，资源，i18n，验证，数据绑定，类型转换，SpEL。
- **测试**： 模拟对象，TestContext框架，Spring MVC 测试，WebTestClient。
- **数据访问**： 事务，DAO支持，JDBC，ORM，编组XML。
- **Web支持**: Spring MVC和Spring WebFlux Web框架。
- **集成**： 远程处理，JMS，JCA，JMX，电子邮件，任务，调度，缓存。
- **语言**： Kotlin，Groovy，动态语言。

IOC AOP

IoC（Inverse of Control:控制反转）是一种设计思想，就是 将原本在程序中手动创建对象的控制权，交由Spring框架来管理。

IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个Map（key，value）,Map 中存放的是各种对象

IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，**却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来**，便于**减少系统的重复代码**，**降低模块间的耦合度**，并**有利于未来的可拓展性和可维护性**。

JDK Proxy Cglib

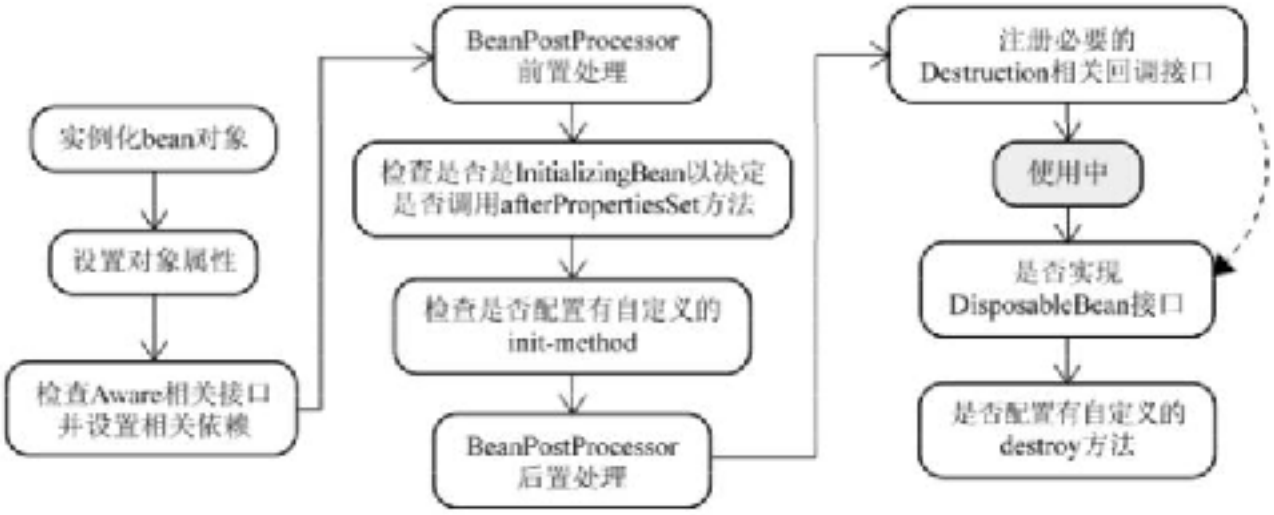
切面

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强

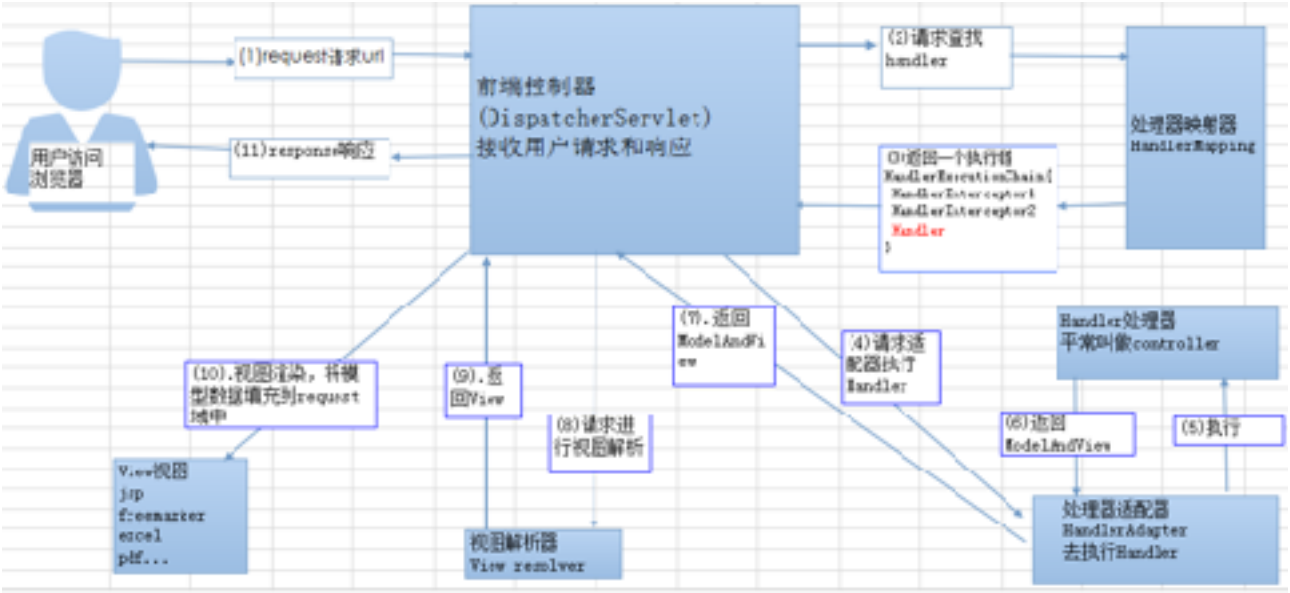
bean作用域

- singleton : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- prototype : 每次请求都会创建一个新的 bean 实例。
- request : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效
- session : 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。
- global-session： 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

Bean生命周期



•Bean 容器找到配置文件中 Spring Bean 的定义。
•Bean 容器利用 Java Reflection API 创建一个Bean的实例。
•如果涉及到一些属性值 利用 `set()`方法设置一些属性值。
•如果 Bean 实现了 `BeanNameAware` 接口，调用 `setBeanName()`方法，传入Bean的名字。
•如果 Bean 实现了 BeanClassLoaderAware 接口，调用 setBeanClassLoader()方法，传入 ClassLoader对象的实例。
•如果Bean实现了 BeanFactoryAware 接口，调用 setBeanClassLoader()方法，传入 ClassLoade r对象的实例。
•与上面的类似，如果实现了其他 *.Aware接口，就调用相应的方法。
•如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行postProcessBeforeInitialization() 方法
•如果Bean实现了InitializingBean接口，执行afterPropertiesSet()方法。
•如果 Bean 在配置文件中的定义包含 init-method 属性，执行指定的方法。
•如果有和加载这个 Bean的 Spring 容器相关的 BeanPostProcessor 对象，执行postProcessAfterInitialization() 方法
•当要销毁 Bean 的时候，如果 Bean 实现了 DisposableBean 接口，执行 destroy() 方法。
•当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的方法。



spring mvc

流程说明（重要）：

1.客户端（浏览器）发送请求，直接请求到 DispatcherServlet。

2.DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。

3.解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。

4.HandlerAdapter 会根据 Handler来调用真正的处理器开处理请求，并处理相应的业务逻辑。

5.处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。

6.ViewResolver 会根据逻辑 View 查找实际的 View。

7.DispaterServlet 把返回的 Model 传给 View（视图渲染）。

8.把 View 返回给请求者（浏览器）

spring 设计模式

·工厂设计模式：Spring使用工厂模式通过 `BeanFactory`、`ApplicationContext` 创建 bean 对象。

·代理设计模式：Spring AOP 功能的实现。

·单例设计模式：Spring 中的 Bean 默认都是单例的。

·模板方法模式：Spring 中 `jdbcTemplate`、`hibernateTemplate` 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。

·包装器设计模式：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。

·观察者模式: Spring 事件驱动模型就是观察者模式很经典的一个应用。

·适配器模式:Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配`Controller`

@Component 和 @Bean 的区别是什么

1.作用对象不同: `@Component` 注解作用于类，而`@Bean`注解作用于方法。

`2.@Component`通常是通过类路径扫描来自动侦测以及自动装配到Spring容器中（我们可以使用 `@ComponentScan` 注解定义要扫描的路径从中找出标识了需要装配的类自动装配到 Spring 的 bean 容器中）。`@Bean` 注解通常是我们在标有该注解的方法中定义产生这个 bean,`@Bean`告诉了Spring这是某个类的示例，当我需要用它的时候还给我。

`3.@Bean` 注解比 `Component` 注解的自定义性更强，而且很多地方我们只能通过 `@Bean` 注解来注册bean。比如当我们引用第三方库中的类需要装配到 `Spring`容器时，则只能通过 `@Bean`来实现

Spring 事务中哪几种事务传播行为?

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

四.spark 方面

开发调优

原则0：cache和persist

原则一：避免创建重复的RDD

原则二：尽可能复用同一个RDD

原则三：对多次使用的RDD进行持久化

原则四：尽量避免使用shuffle类算子

a)尽量用相同的 partitioner

broadcast variables 如果一个数据集小到能够塞进一个 executor 的内存中，那么它就可以在 driver 中写入到一个 hash table中，然后 broadcast 到所有的 executor 中。然后 map transformation 可以引用这个 hash table 作查询

b)是减少 shuffle 的次数以及被 shuffle 的文件的大小

c)避免使用shuffle类算子

原则五：

a)使用map-side预聚合的shuffle操作　aggregateByKey reduceByKe >groupByKey

b)mapValues

针对k,v结构的rdd，mapValues直接对value进行操作，不对Key造成影响，可以减少不必要的分区操作。

原则六：使用高性能的算子

a)使用reduceByKey/aggregateByKey替代groupByKey

b)使用mapPartitions替代普通map

c)使用foreachPartitions替代foreach

d)使用filter之后进行coalesce操作

e)使用repartitionAndSortWithinPartitions替代repartition与sort类操作.原因 shuffle与sort两个操作同时进行，比先shuffle再sort来说，性能可能是要高

f)repartition和coalesce　repartition会将所有的数据进行一次shuffle，然后重新分区

当进行联合的规约操作时，避免使用 groupByKey。举个例子，rdd.groupByKey().mapValues(_ .sum) 与 rdd.reduceByKey(_ + _) 执行的结果是一样的，但是前者需要把全部的数据通过网络传递一遍，而后者只需要根据每个 key 局部的 partition 累积结果，在 shuffle 的之后把局部的累积值相加后得到结果

当输入和输入的类型不一致时，避免使用 reduceByKey。举个例子，我们需要实现为每一个key查找所有不相同的 string。一个方法是利用 map 把每个元素的转换成一个 Set，再使用 reduceByKey 将这些 Set 合并起来

避免 flatMap-join-groupBy 的模式。当有两个已经按照key分组的数据集，你希望将两个数据集合并，并且保持分组，这种情况可以使用 cogroup。这样可以避免对group进行打包解包的开销

原则七：broadcast

a)实现map-side join

在需要join操作时，将较小的那份数据转化为普通的集合（数组）进行广播，然后在大数据集中使用小数据进行相应的查询操作，就可以实现map-side join的功能，避免了join操作的shuffle过程。在我之前的文章中对此用法有详细说明和过程图解。

b)使用较大的外部变量

原则八：使用Kryo优化序列化性能
原则九：优化数据结构
字符串替代对象，使用原始类型（比如Int、Long）替代字符串，使用数组替代集合类型

资源调优

a)因此Executor的内存主要分为三块：第一块是让task执行我们自己编写的代码时使用，默认是占Executor总内存的20%；第二块是让 task通过shuffle过程拉取了上一个stage的task的输出后，进行聚合等操作时使用，默认也是占Executor总内存的20%；第三块是 让RDD持久化时使用，默认占Executor总内存的60%。

b)调试资源分配 按需调整 executor core memory storage_memory_fraction
Spark（以及YARN） 需要关心的两项主要的资源是 CPU 和 内存
应用的master，是一个非 executor 的容器，它拥有特殊的从 YARN 请求资源的能力，它自己本身所占的资源也需要被计算在内。在 yarn-client 模式下，它默认请求 1024MB 和 1个core。在 yarn-cluster 模式中，应用的 master 运行 driver，所以使用参数 --driver-memory 和 --driver-cores 配置它的资源常常很有用。
在 executor 执行的时候配置过大的 memory 经常会导致过长的GC延时，64G是推荐的一个 executor 内存大小的上限。
我们注意到 HDFS client 在大量并发线程是时性能问题。大概的估计是每个 executor 中最多5个并行的 task 就可以占满的写入带宽。
在运行微型 executor 时（比如只有一个core而且只有够执行一个task的内存）扔掉在一个JVM上同时运行多个task的好处。比如 broadcast 变量需要为每个 executor 复制一遍，这么多小\executor会导致更多的数据拷贝

c)调试并发 rdd.partitions().size() 不停的将 partition 的个数从上次实验的 partition 个数乘以1.5，直到性能不再提升为止。

1）数据问题
key本身分布不均匀(包括大量的key为空)
key的设置不合理

三种情况:
1）null（空值）或是一些无意义的信息()之类的,大多是这个原因引起。 filter
2）无效数据，大量重复的测试数据或是对结果影响不大的有效数据。 filter
3）有效数据，业务导致的正常数据分布。
隔离执行，将异常的key过滤出来单独处理，最后与正常数据的处理结果进行union操作。
对key先添加随机值，进行操作后，去掉随机值，再进行一次操作。
使用reduceByKey 代替 groupByKey
使用map join。

2）spark使用问题
shuffle时的并发度不够
计算方式有误

数据倾斜的解决方案

解决方案一：使用Hive ETL预处理数据 (不懂)
解决方案二：过滤少数导致倾斜的key ，
解决方案三：提高shuffle操作的并行度
解决方案四：两阶段聚合（局部聚合+全局聚合）
方案适用场景：对RDD执行reduceByKey等聚合类shuffle算子或者在Spark SQL中使用group by语句进行分组聚合时，比较适用这种方案
这个方案的核心实现思路就是进行两阶段聚合。第一次是局部聚合，先给每个key都打上一个随机数，
然后将各个key的前缀给去掉，就会变成(hello,2)(hello,2)，再次进行全局聚
解决方案五：将reduce join转为map join
解决方案六：采样倾斜key并分拆join操作
方案适用场景：两个RDD/Hive表进行join的时候，如果数据量都比较大，无法采用“解决方案五”，那么 此时可以看一下两个RDD/Hive表中的key分布情况。如果出现数据倾斜，是因为其中某一个RDD/Hive表中的少数几个key的数据量过大，而另 一个RDD/Hive表中的所有key都分布比较均匀，那么采用这个解决方案是比较合适的。
解决方案七：使用随机前缀和扩容RDD进行join
将原先一样的key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同 key”分散到多个task中去处理，而不是让一个task处理大量的相同key。该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾 斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；而这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。

.Master挂掉,standby重启也失效
增加Master的内存占用，在Worker节点spark-env.sh
减少保存在Worker内存中的Driver,Executor信息

.shuffle FetchFailedException
这种问题一般发生在有大量shuffle操作的时候,task不断的failed,然后又重执行，一直循环下去，直到application失败。
一般遇到这种问题提高executor内存即可,同时增加每个executor的cpu,这样不会减少task并行度。

.Executor&Task Lost 不同环境的io时间，测试环境正常，现网环境崩溃
spark.network.timeout

a)数据倾斜
解决方法:
发现数据倾斜的时候，不要急于提高executor的资源，修改参数或是修改程序，首先要检查数据本身，是否存在异常数据
原因
常见于各种shuffle操作，例如reduceByKey,groupByKey,join等操作

后果：时间慢，同一个task OOM

b)任务倾斜
性能监控,开启spark的推测机制

.OOM
driver OOM
一般是使用了collect操作将所有executor的数据聚合到driver导致。尽量不要使用collect操作即可。

executor OOM
可以按下面的内存优化的方法增加code使用内存空间

task not serializable
a)将所有调用到的外部变量直接放入到以上所说的这些算子中，这种情况最好使用foreachPartition减少创建变量的消耗。
b)将需要使用的外部变量包括sparkConf,SparkContext,都用 @transent进行注解，表示这些变量不需要被序列化
c)将外部变量放到某个class中对类进行序列化。

五. k8s +docker
https://blog.csdn.net/huakai_sun/article/details/82378856

Kubernetes是一个开源容器管理工具，负责容器部署，容器扩缩容以及负载平衡。作为Google的创意之作，它提供了出色的社区，并与所有云提供商合作。因此，我们可以说Kubernetes不是一个容器化平台，而是一个多容器管理解决方案。

.什么是Kubernetes
Kubernetes是一个开源容器管理工具，负责容器部署，容器扩缩容以及负载平衡.Kubernetes不是一个容器化平台，而是一个多容器管理解决方案
.Kubernetes与Docker有什么关系?
因此，我们说Docker构建容器，这些容器通过Kubernetes相互通信。

自动化容器部署和复制。
实时弹性收缩容器规模。
容器编排成组，并提供容器间的负载均衡。
调度：容器在哪个机器上运行。
组成：
kubectl:客户端命令行工具，作为整个系统的操作入口。
kube-apiserver:以REST API服务形式提供接口，作为整个系统的控制入口。
kube-controller-manager:执行整个系统的后台任务，包括节点状态状况、Pod个数、Pods和Service的关联等。
kube-scheduler:负责节点资源管理，接收来自kube-apiserver创建Pods任务，并分配到某个节点。

五. k8s +docker
https://blog.csdn.net/huakai_sun/article/details/82378856

Kubernetes是一个开源容器管理工具，负责容器部署，容器扩缩容以及负载平衡。作为Google的创意之作，它提供了出色的社区，并与所有云提供商合作。因此，我们可以说Kubernetes不是一个容器化平台，而是一个多容器管理解决方案。

.什么是Kubernetes
Kubernetes是一个开源容器管理工具，负责容器部署，容器扩缩容以及负载平衡.Kubernetes不是一个容器化平台，而是一个多容器管理解决方案
.Kubernetes与Docker有什么关系?
因此，我们说Docker构建容器，这些容器通过Kubernetes相互通信。

自动化容器部署和复制。
实时弹性收缩容器规模。
容器编排成组，并提供容器间的负载均衡。
调度：容器在哪个机器上运行。
组成：
kubectl:客户端命令行工具，作为整个系统的操作入口。
kube-apiserver:以REST API服务形式提供接口，作为整个系统的控制入口。
kube-controller-manager:执行整个系统的后台任务，包括节点状态状况、Pod个数、Pods和Service的关联等。
kube-scheduler:负责节点资源管理，接收来自kube-apiserver创建Pods任务，并分配到某个节点。