java线程面试大全

1、 什么是线程?

- 线程是操作系统能够进行运算的最小单位，他包含在实际的运作单位里面，是进程中的实际运作单位。
- 程序员可以通过它进行多处理器编程，你可以使用多线程对运算密集型任务提速。比如，如果一个线程完成一个任务要100毫秒，那么用十个线程完成改任务只需10毫秒。Java在语言层面对多线程提供了卓越的支持，它也是一个很好的卖点
- 它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。线程也有就绪、阻塞和运行三种基本状态。我们通过多线程编程，能更高效的提高系统内多个程序间并发执行的程度，从而显著提高系统资源的利用率和吞吐量。

2、进程调度算法

- 实时系统：FIFO(First Input First Output，先进先出算法)，SJF(Shortest Job First，最短作业优先算法)，SRTF(Shortest Remaining Time First，最短剩余时间优先算法）。
- 互式系统：RR(Round Robin，时间片轮转算法)，HPF(Highest Priority First，最高优先级算法)，多级队列，最短进程优先，保证调度，彩票调度，公平分享调度。

3、线程和进程有什么区别?

- 线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据
- 通信：不同进程之间通过IPC（进程间通信）接口进行通信。同一进程的线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。

- 调度和切换：线程上下文切换比进程上下文切换要快得多。

4、多线程编程的好处是什么?

- 在多线程程序中，多个线程被并发的执行以提高程序的效率，CPU不会因为某个线程需要等待资源而进入空闲状态(提高CPU的利用率)。
- 多个线程共享堆内存(heap memory)，因此创建多个线程去执行一些任务会比创建多个进程更好。举个例子，Servlets比CGI更好，是因为Servlets支持多线程而CGI不支持。

5、如何在java中实现多线程

- 在语言层面有两种方式。可以**继承**java.lang.Thread线程类，但是它需要调用java.lang.Runnable接口来执行。由于线程类本身就是调用的Runnable接口，所以你可以继承java.lang.Thread类或者直接调用Runnable接口来重写run()方法实现线程。
- 还可以实现callable接口，和实现 Runnable接口一样。
- 那么选择哪个更好?
    - 由于Java不支持类的**多重继承**，但允许调用多个接口。因此我们建议调用Runnable接口来创建线程.

6、Thread 类中的start() 和 run() 方法有什么区别?

- start()方法被用来启动新创建的线程,而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样
    - 首先，start方法内部会调用run方法。
    - start与run方法的主要区别在于当程序调用start方法一个新线程将会被创建，并且在run方法中的代码将会在新线程上运行。
    - 然而在你直接调用run方法的时候，程序并不会创建新线程，run方法内部的代码将在当前线程上运行。大多数情况下调用run方法是一个bug或者变成失误。因为调用者的初衷是调用start方法去开启一个新的线程，这个错误可以被很多静态代码覆盖工具检测出来，比如与fingbugs. 如果你想要运行需要消耗大量时间的任务，你最好使用start方法，否则在你调用run方法的时候，你的主线程将会被卡住。
    - 还有一个区别在于，一但一个线程被启动，你不能重复调用该thread对象的start方法，调用已经启动线程的start方法将会报IllegalStateException异常， 而你却可以重复调用run方法。

6、notify()和notifyAll()有什么区别?

- 两者最大的区别：
    - **notifyAll**使所有原来在该对象上等待被notify的线程统统退出wait的状态，变成等待该对象上的锁，一旦该对象被解锁，他们就会去竞争。
    - notify他只是选择一个wait状态线程进行通知，并使它获得该对象上的锁，但不惊动其他同样在等待被该对象notify的线程们，当第一个线程运行完毕以后释放对象上的锁，此时如果该对象没有再次使用notify语句，即便该对象已经空闲，其他wait状态等待的线程由于没有得到该对象的通知，继续处在wait状态，直到这个对象发出一个notify或notifyAll，它们等待的是被notify或notifyAll，而不是锁。
- notify()和notifyAll()都是Object对象用于通知处在等待该对象的线程的方法。

- void notify(): 唤醒一个正在等待该对象的线程。
- void notifyAll(): 唤醒所有正在等待该对象的线程。

7、请说出与线程同步以及线程调度相关的方法。

- wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
- sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
- notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
- notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

8、java如何实现多线程之间的通讯和协作?

- Java提供了3个非常重要的方法来巧妙地解决线程间的通信问题。这3个方法分别是：wait()、notify()和notifyAll()。
- 它们都是Object类的最终方法，因此每一个类都默认拥有它们。虽然所有的类都默认拥有这3个方法，但是只有在synchronized关键字作用的范围内，并且是同一个同步问题中搭配使用这3个方法时才有实际的意义。这些方法在Object类中声明的语法格式如下所示：
  - final void wait() throws InterruptedException
  - final void notify()
  - final void notifyAll()

9、为什么线程通信的方法wait(),notify()和notifyAll()被定义在Object类里,为什么不放在Thread类里面?

- 不把它放在Thread类里的原因，++一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得++，简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象
- Java的每个对象中都有一个锁(monitor，也可以成为监视器)并且wait()，notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用
- 在Java的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是Object类的一部分，这样Java的每一个类都有用于线程间通信的基本方法
- Java API 的设计人员提供了一些方法当等待条件改变的时候通知它们，但是这些方法没有完全实现。notify()方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地,而notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行.

10、为什么wait(),notify()和notifyAll()必须在同步方法或者同步块中被调用?

- 当一个线程需要调用对象的wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的notify()方法。同样的，当一个线程需要调用对象的notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。
- 由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。
- **主要是因为**Java API强制要求这样做，如果你不这么做，你的代码会抛出IllegalMonitorStateException异常。还有一个原因是为了避免wait和notify之间产生竞态条件。

11、进程间的通信方式

> - **管道(pipe)**：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。
> - **有名管道 (named pipe)**： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
> - **信号量(semophore)**： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
> - **消息队列(message queue)**： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
> - **信号 (sinal)**： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
> - **共享内存(shared memory)**：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。
> - **套接字(socket)**： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

12、sleep()和wait()有什么区别?

> - sleep是线程类（Thread）的方法，导致此线程暂停执行指定时间，给执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复。调用sleep不会释放对象锁。
> - wait是Object类的方法，对此对象调用wait方法导致本线程放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象发出notify方法（或notifyAll）后本线程才进入对象锁定池准备获得对象锁进入运行状态。

13、为什么Thread类的sleep()和yield()方法是静态的?

> - Thread类的sleep()和yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。
> - 它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

14、Java中CyclicBarrier 和 CountDownLatch有什么不同?

> - CyclicBarrier 和 CountDownLatch 都可以用来让一组线程等待其它线程。与 CyclicBarrier 不同的是，CountdownLatch 不能重新使用

7、为什么需要并行设计?

> - 业务需求：业务上需要多个逻辑单元，比如多个客户端要发送请求
> - 性能需求：在多核OS中，使用多线程并发执行性能会比单线程执行的性能好很多

15、并发和并行的区别：

> - 举例：
>   - 你吃饭吃到一半，电话来了，你一直到吃完了以后才去接，这就说明你不支持并发也

不支持并行。
    - 你吃饭吃到一半，电话来了，你停了下来接了电话，接完后继续吃饭，这说明你支持并发。
    - 你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行。
- 并发的关键是有处理多个任务的能力，但是不一定同时处理，而并行表示同一个时刻处理多个任务，两者的关键点就是是否同时。
    - 解释一：并行是指两个或者多个线程在同一时刻发生；而并发是指两个或多个线程在同一时间间隔发生（交替运行）
    - 解释二：并行是在不同实体上的多个事件（多个JVM），并发是在同一实体上的多个事件（一个JVM）。
    - 并行又分在一台处理器上同时处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

16、什么是Daemon(守护)线程？它有什么意义？

- 在Java中有两类线程：用户线程 (User Thread)、守护线程 (Daemon Thread)。
- 所谓后台(daemon)线程，是指在程序运行的时候在后台提供一种通用服务的线程，并且这个线程并不属于程序中不可或缺的部分。因此，当所有的非后台线程介绍时，程序也就终止了，同时会杀死进程中的所有后台线程。反过来说，只要有任何非后台线程还在运行，程序就不会终止。必须在线程启动之前调用setDaemon()方法，才能把它设置为后台线程。注意：后台进程在不执行finally子句的情况下就会终止其run()方法。
- 守护线程和用户线程的区别在于：守护线程依赖于创建它的线程，而用户线程则不依赖。举个简单的例子：如果在main线程中创建了一个守护线程，当main方法运行完毕之后，守护线程也会随着消亡。而用户线程则不会，用户线程会一直运行直到其运行完毕。在JVM中，像垃圾收集器线程就是守护线程。
- 守护线程必须在用户线程执行前调用，它是一个后台服务线程,一个守护线程创建的子线程依然是守护线程。

17、如何创建守护线程?

- 使用Thread类的setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在调用start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

18、 如何停止一个线程

- Java提供了很丰富的API但没有为停止线程提供API。JDK1.0本来有一些像++stop(), suspend() 和 resume()的控制方法但是由于潜在的死锁威胁因此在后续的JDK版本中他们被弃用了++，之后JavaAPI的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。
- ==当run()或者call()方法执行完的时候线程会自动结束,如果要手动结束一个线程，可以用volatile布尔变量来退出run()方法的循环或者是取消任务来中断线程。==
- 当不阻塞时候设置一个标志位，让代码块正常运行结束并停止线程。
- 如果发生了阻塞，用interupt()方法，Thread.interrupt（）方法不会中断一个正在运行的线程。这一方法实际上完成的是，在线程受到阻塞时抛出一个中断信号，这样线程就得以退出阻塞的状态。

19、什么是Thread Group？为什么不建议使用它？

- ThreadGroup是一个类，它的目的是提供关于线程组的信息。
- hreadGroup API比较薄弱，它并没有比Thread提供了更多的功能。它有两个主要的功能：一是获取线程组中处于活跃状态线程的列表；二是设置为线程设置未捕获异常处理器(ncaughtexceptionhandler)。但在Java1.5中Thread类也添加了setUncaughtExceptionHandler(UncaughtExceptionHandlereh)方法，所以ThreadGroup是已经过时的，不建议继续使用。

20、 什么是Java线程转储(Thread Dump)，如何得到它?

- 线程转储是一个JVM活动线程的列表，它对于分析系统瓶颈和死锁非常有用。>> - 有很多方法可以获取线程转储——使用Profiler，Kill-3命令，jstack工具等等。我更喜欢jcmd命令（jdk1.8以上）。

21、什么是FutureTask?

- 在Java并发程序中FutureTask表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成get方法将会阻塞。一个FutureTask对象可以对调用了Callable和Runnable的对象进行包装，由于FutureTask也是调用了Runnable接口所以它可以提交给Executor来执行。

22、Java中interrupted 和 isInterruptedd方法的区别?

- interrupted() :会将中断状态清除，Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。
- isInterruptedd : 不会将中断状态清除，非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。
- 简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有有可能被其它线程调用中断来改变。

23、为什么你应该在循环中检查等待条件?

- 处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在notify()方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用wait()方法效果更好的原因。

24、Java中的同步集合与并发集合有什么区别?

- 同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，
- 同步集合:在Java1.5之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性
- 并发集合: 可扩展性更高,Java5介绍了并发集合像ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性

25、java 的内存模型是什么？《Java并发编程实践》16章

- Java内存模型规定和指引Java程序在不同的内存架构、CPU和操作系统间有确定性地行为。它在多线程的情况下尤其重要。Java内存模型对一个线程所做的变动能被其它线程可

见提供了保证，它们之间是**==先行发生关系==**。这个关系定义了一些规则让程序员在并发编程时思路更清晰

- 线程内的代码能够按先后顺序执行，这被称为程序次序规则。
- 对于同一个锁，一个解锁操作一定要发生在时间上后发生的另一个锁定操作之前，也叫做管程锁定规则。
- 前一个对volatile的写操作在后一个volatile的读操作之前，也叫volatile变量规则。
- 一个线程内的任何操作必需在这个线程的start()调用之后，也叫作线程启动规则。
- 一个线程的所有操作都会在线程终止之前，线程终止规则。
- 一个对象的终结操作必需在这个对象构造完成之后，也叫对象终结规则。

26、Java中堆和栈有什么不同?

- **栈**是一块和线程紧密相关的内存区域,每个线程都有自己的栈内存，用于存储本地变量，方法参数和栈调用，一个线程中存储的变量对其它线程是不可见的。
- **堆**是所有线程共享的一片公用内存区域,对象都在堆里创建，为了提升效率线程会从堆中弄一个缓存到自己的栈，如果多个线程使用该变量就可能引发问题，这时volatile 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

27、 什么是线程池？ 为什么要使用它?

- 节省资源，提高效率，提高线程的可管理性,创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程,
- 从JDK1.5开始，JavaAPI提供了Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池).
- 常用线程池：ExecutorService 是主要的实现类
  - Executors.newSingleT hreadPool()
  - newFixedThreadPool()
  - newcachedTheadPool()
  - newScheduledThreadPool()

28、CachedThreadPool 、 FixedThreadPool、SingleThreadPool

- **newSingleThreadExecutor**:
  - 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务， 保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行
  - 适用场景：任务少 ，并且不需要并发执行
- **newCachedThreadPool** :
  - 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程.线程没有任务要执行时，便处于空闲状态，处于空闲状态的线程并不会被立即销毁（会被缓存住），只有当空闲时间超出一段时间(默认为60s)后，线程池才会销毁该线程（相当于清除过时的缓存）。新任务到达后，线程池首先会让被缓存住的线程（空闲状态）去执行任务，如果没有可用线程（无空闲线程），便会创建新的线程。

    - **适用场景**：处理任务速度>提交任务速度,耗时少的任务(避免无限新增线程)
- **newFixedThreadPool**:
    - 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
- **newScheduledThreadPool**:创建一个定长线程池，支持定时及周期性任务执行

29、ThreadPoolExecutor

- 构造方法参数说明
    - **corePoolSize**:核心线程数，默认情况下核心线程会一直存活，即使处于闲置状态也不会受存keepAliveTime限制。除非将allowCoreThreadTimeOut设置为true。
    - **maximumPoolSize**:线程池所能容纳的最大线程数。超过这个数的线程将被阻塞。当任务队列为没有设置大小的LinkedBlockingDeque时，这个值无效。
    - **keepAliveTime**:非核心线程的闲置超时时间，超过这个时间就会被回收。 unit:指定keepAliveTime的单位，如TimeUnit.SECONDS。当将allowCoreThreadTimeOut设置为true时对corePoolSize生效。
    - **workQueue**:线程池中的任务队列.常用的有三种队列，SynchronousQueue,LinkedBlockingDeque,ArrayBlockingQueue。
    - **threadFactory**:线程工厂，提供创建新线程的功能。ThreadFactory是一个接口，只有一个方法
- 原理
    - 如果当前池大小 poolSize 小于 corePoolSize，则创建新线程执行任务。
    - 如果当前池大小poolSize大于corePoolSize，且等待队列未满，则进入等待队列
    - 如果当前池大小 poolSize 大于 corePoolSize 且小于 maximumPoolSize ，且等待队列已满，则创建新线程执行任务。
- 如果当前池大小 poolSize 大于 corePoolSize 且大于 maximumPoolSize ，且等待队列已满，则调用拒绝策略来处理该任务。
    - 线程池里的每个线程执行完任务后不会立刻退出，而是会去检查下等待队列里是否还有线程任务需要执行，如果在 keepAliveTime 里等不到新的任务了，那么线程就会退出。

30、CopyOnWriteArrayList

- **CopyOnWriteArrayList** : 写时加锁，当添加一个元素的时候，将原来的容器进行copy，复制出一个新的容器，然后在新的容器里面写，写完之后再将原容器的引用指向新的容器，而读的时候是读旧容器的数据，所以可以进行并发的读，但这是一种弱一致性的策略。
- **使用场景**：CopyOnWriteArrayList适合使用在读操作远远大于写操作的场景里，比如缓存。

31、Executor拒绝策略

- Executor框架同java.util.concurrent.Executor 接口在Java 5中被引入。Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架,利用Executors框架可以非常方便的创建一个线程池.
- **AbortPolicy**: 为java线程池默认的阻塞策略，不执行此任务，而且直接抛出一个运行时异常，切记ThreadPoolExecutor.execute需要try catch，否则程序会直接退出.
- **DiscardPolic**y: 直接抛弃，任务不执行，空方法
- **DiscardOldestPolicy**:从队列里面抛弃head的一个任务，并再次execute 此task。

- **CallerRunsPolicy**:在调用execute的线程里面执行此command，会阻塞入
- **用户自定义拒绝策略**:实现RejectedExecutionHandler，并自己定义策略模式

32、如果你提交任务时，线程池队列已满。会时发会生什么?

- 事实上如果一个任务不能被调度执行那么ThreadPoolExecutor's submit()方法将会抛出一个RejectedExecutionException异常。

33、Java线程池中submit() 和 execute()方法有什么区别?

- 两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法

34、Swing是线程安全的吗？ 为什么?

- 你可以很肯定的给出回答，Swing不是线程安全的，但是你应该解释这么回答的原因即便面试官没有问你为什么。当我们说swing不是线程安全的常常提到它的组件，这些组件不能在多线程中进行修改，所有对GUI组件的更新都要在AWT线程中完成，而Swing提供了同步和异步两种回调方法来进行更新

35、Swing API中那些方法是线程安全的?

- 这个问题又提到了swing和线程安全，虽然组件不是线程安全的但是有一些方法是可以被多线程安全调用的，比如repaint(), revalidate()。 JTextComponent的setText()方法和JTextArea的insert() 和 append() 方法也是线程安全的。

36、多线程中的忙循环是什么?

- 忙循环就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了

36、如果同步块内的线程抛出异常会发生什么?

- 这个问题坑了很多Java程序员，若你能想到锁是否释放这条线索来回答还有点希望答对。无论你的同步块是正常还是异常退出的，里面的线程都会释放锁，所以对比锁接口我更喜欢同步块，因为它不用我花费精力去释放锁，该功能可以在finally block里释放锁实现。

37、单例模式的双检锁是什么?

- 这个问题在Java面试中经常被问到，但是面试官对回答此问题的满意度仅为50%。一半的人写不出双检锁还有一半的人说不出它的隐患和Java1.5是如何对它修正的。它其实是一个用来创建线程安全的单例的老方法，当单例实例第一次被创建时它试图用单个锁进行性能优化，但是由于太过于复杂在JDK1.4中它是失败的，我个人也不喜欢它。无论如何，即便你也不喜欢它但是还是要了解一下，因为它经常被问到。

38、写出3条你遵循的多线程最佳实践

- 给你的线程起个有意义的名字。
    - 这样可以方便找bug或追踪。OrderProcessor, QuoteProcessor or TradeProcessor

这种名字比 Thread-1. Thread-2 and Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。

- 避免锁定和缩小同步的范围
  - 锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。
- 多用同步类少用wait 和 notify
  - 首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。
- 多用并发集合少用同步集合
  - 这是另外一个容易遵循且受益巨大的最佳实践，并发集合比同步集合的可扩展性更好，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到map，你应该首先想到用ConcurrentHashMap

39、如何强制启动一个线程?

- 这个问题就像是如何强制进行Java垃圾回收，目前还没有觉得方法，虽然你可以使用System.gc()来进行垃圾回收，但是不保证能成功。在Java里面没有办法强制启动一个线程，它是被线程调度器控制着且Java没有公布相关的API。

40、Java中invokeAndWait 和 invokeLater有什么区别?

- 这两个方法是Swing API 提供给Java开发者用来从当前线程而不是事件派发线程更新GUI组件用的。InvokeAndWait()同步更新GUI组件，比如一个进度条，一旦进度更新了，进度条也要做出相应改变。如果进度被多个线程跟踪，那么就调用invokeAndWait()方法请求事件派发线程对组件进行相应更新。而invokeLater()方法是异步调用更新组件的。

41、如何合理的配置java线程池?

- 如CPU密集型的任务，基本线程池应该配置多大？IO密集型的任务，基本线程池应该配置多大？用有界队列好还是无界队列好？任务非常多的时候，使用什么阻塞队列能获取最好的吞吐量?
  - 配置线程池时CPU密集型任务可以少配置线程数，大概和机器的cpu核数相当，可以使得每个线程都在执行任务
  - IO密集型时，大部分线程都阻塞，故需要多配置线程数，2*cpu核数
  - 有界队列和无界队列的配置需区分业务场景，一般情况下配置有界队列，在一些可能会有爆发性增长的情况下使用无界队列。
  - 任务非常多时，使用非阻塞队列使用CAS操作替代锁可以获得好的吞吐量。

42、如何写代码来解决生产者消费者问题?

- 在现实中你解决的许多线程问题都属于生产者消费者模型，就是一个线程生产任务供其它线程进行消费，你必须知道怎么进行线程间通信来解决这个问题。比较低级的办法是用wait和notify来解决这个问题，比较赞的办法是用Semaphore 或者 BlockingQueue来实现生产者消费者模型

- 实现生产者消费者模型是多线程程序经典问题之一，它描述是有一块缓冲区作为仓库，生产者可以将产品放入仓库，消费者则可以从仓库中取走产品。两个解决方法：
    - （1）采用某种机制保护生产者和消费者之间的同步，较高的效率并且易于实现，代码的可控制性较好，常用。
    - （2）在生产者和消费者之间建立一个管道，管道缓冲区不易控制，被传输数据对象不易于封装等，实用性不强。
- 同步问题核心在于：如何保证同一资源被多个线程并发访问时的完整性。常用的同步方法是采用信号或加锁机制，保证资源在任意时刻至多被一个线程访问.三个同步方法，一个管道方法。
    - wait() / notify()方法
    - await() / signal()方法
    - BlockingQueue阻塞队列方法//JDK5.0的新增内容
        - 实现方式采用的是我们第2种await() / signal()方法,它可以在生成对象时指定容量大小。它用于阻塞操作的是put()和take()方法。
        - put()方法：类似于我们上面的生产者线程，容量达到最大时，自动阻塞。
        - take()方法：类似于我们上面的消费者线程，容量为0时，自动阻塞。
    - Semaphore方法
        - 信号量（Semaphore）维护了一个许可集。在许可可用前会阻塞每一个acquire()，然后再获取该许可,是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源 Semaphore可以用于做流量控制，特别公用资源有限的应用场景，比如数据库连接。假如有一个需求，要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发的读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有十个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。
        - 每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore 只对可用许可的号码进行计数，并采取相应的行动。
        - Semaphore 通常用于限制可以访问某些资源（物理或逻辑的）的线程数目。
        - 注意，调用acquire()时无法保持同步锁，因为这会阻止将项返回到池中。信号量封装所需的同步，以限制对池的访问，这同维持该池本身一致性所需的同步是分开的。同步令牌（notFull.acquire()）必须在互斥令牌（mutex.acquir()）前面获得，如果先得到互斥锁再发生等待，会造成死锁。
    - PipedInputStream / PipedOutputStream方法

43、什么是Callable和Future?

- Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。
- Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结果。
- java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。
- Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。
- FutureTask是Future的一个基础实现，我们可以将它同Executors使用处理异步任务。通

> 常我们不需要使用FutureTask类，单当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。阅读Java FutureTask例子，学习如何使用它。

44、多读少写的场景应该使用哪个并发容器，为什么使用它?

> - 比如你做了一个搜索引擎，搜索引擎每次搜索前需要判断搜索关键词是否在黑名单里，黑名单每天更新一次。
>   - CopyOnWriteArrayList这个容器适用于多读少写...
>   - 读写并不是在同一个对象上。在写时会大面积复制数组，所以写的性能差，在写完成后将读的引用改为执行写的对象

45、Java里的阻塞队列

> - 7个队列阻塞
>   - **ArrayBlockingQueue**： 一个由数组结构组成的有界阻塞队列。
>   - **LinkedBlockingQueue**： 一个由链表结构组成的有界阻塞队列。
>   - **PriorityBlockingQueue**： 一个支持优先级排序的无界阻塞队列。
>   - **DelayQueue**：一个使用优先级队列实现的无界阻塞队列。
>   - **SynchronousQueue**：一个不存储元素的阻塞队列。
>   - **LinkedTransferQueue**：一个由链表结构组成的无界阻塞队列。
>   - **LinkedBlockingDeque**：一个由链表结构组成的双向阻塞队列。
> - 添加元素
>   - Java中的阻塞队列接口BlockingQueue继承自Queue接口。BlockingQueue接口提供了3个添加元素方法。
>   - **add**：添加元素到队列里，添加成功返回true，由于容量满了添加失败会抛出IllegalStateException异常
>   - **offer**：添加元素到队列里，添加成功返回true，添加失败返回false
>   - **put**：添加元素到队列里，如果容量满了会阻塞直到容量不满
> - 删除方法
>   - **poll**：删除队列头部元素，如果队列为空，返回null。否则返回元素。
>   - **remove**：基于对象找到对应的元素，并删除。删除成功返回true，否则返回false
>   - **take**：删除队列头部元素，如果队列为空，一直阻塞到队列有元素并删除

46、什么是Java Timer类？如何创建一个有特定时间间隔的任务?

> - java.util.Timer是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。
> - java.util.TimerTask是一个实现了Runnable接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用Timer去安排它的执行。

47、什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)?

> - java.util.concurrent.atomic包下，可以分为四种类型的原子更新类：原子更新基本类型、原子更新数组类型、原子更新引用和原子更新属性。

    - 原子更新基本类型
        - AtomicBoolean：原子更新布尔变量
        - AtomicInteger：原子更新整型变量
        - AtomicLong：原子更新长整型变量
    - 原子更新数组
        - AtomicIntegerArray：原子更新整型数组的某个元素
        - AtomicLongArray：原子更新长整型数组的某个元素
        - AtomicReferenceArray：原子更新引用类型数组的某个元素
        - AtomicIntegerArray常用的方法有：
            - int addAndSet(int i, int delta)：以原子方式将输入值与数组中索引为i的元素相加 boolean compareAndSet(int i,
            - int expect, int update)：如果当前值等于预期值，则以原子方式更新数组中索引为i的值为update值
- 原子更新引用类型
    - AtomicReference：原子更新引用类型
    - AtomicReferenceFieldUpdater：原子更新引用类型里的字段
    - AtomicMarkableReference：原子更新带有标记位的引用类型。
- 原子更新字段类
    - AtomicIntegerFieldUpdater：原子更新整型字段
    - AtomicLongFieldUpdater：原子更新长整型字段
    - AtomicStampedReference：原子更新带有版本号的引用类型。
    - 更新字段，需要两个步骤:
        - 每次必须使用newUpdater创建一个更新器，并且需要设置想要更新的类的字段
        - 更新类的字段（属性）必须为public volatile
- 原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。
- int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。
- 为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用同步技术来做到这一点。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

48、有三个线程T1，T2，T3，怎么确保它们按顺序执行?

- 在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。你可以查看这篇文章了解更多。

49、线程作用

- 发挥多核CPU的优势 如果是单线程的程序，那么在双核CPU上就浪费了50%，在4核CPU

上就浪费了75%。多线可以充分利用CPU的。
- 防止阻塞 多条线程同时运行，一条线程的代码执行阻塞，也不会影响其它任务的执行。

50、Thread类中的yield方法有什么作用?

- Yield方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。点击这里查看更多yield方法的相关内容。

51、Runnable接口和Callable接口的区别

- Runnable接口中的run()方法的返回值是void，它只是纯粹地去执行run()方法中的代码而已；
- Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获 取异步执行的结果。

52、线程安全的级别

代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么代码就是线程安全的。线程安全也是有级别之分的：

- 不可变: 像String、Integer、Long这些，都是final类型的类，要改变除非新创建一个。
- 绝对线程安全: 不管运行时环境如何都不需要额外的同步措施。Java中有绝对线程安全的类，比如CopyOnWriteArrayList、CopyOnWriteArraySet。
- 相对线程安全: 像Vector这种，add、remove方法都是原子操作，不会被打断。如果有个线程在遍历某个Vector，同时另一个线程对其结构进行修改，会出现ConcurrentModificationException（failfast机制）。
- 线程非安全: 这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类。

53、java中的锁

1. 在Java多线程中,synchronized实现线程之间同步互斥,JDK1.5以后,Java类库中新增了Lock接口用来实现锁功能。
2. 锁为对共享数据进行保护,同一把锁保护的共享数据,任何线程访问都需要先持有该锁。一把锁一个线程,当该锁的持有线程对数据访问结束之后必须释放该锁,让其他线程持有。++锁的持有线程在锁的获得和锁的释放之间的这段时间所执行的代码被称为临界区++。
3. 锁能够保护共享数据以实现线程安全,主要作用有保障原子性、保障可见性和保障有序性。由于锁具有互斥性,因此当线程执行临界区中的代码时,其他线程无法做到干扰,临界区中的代码也就具有了不可分割的原子特性。
4. 锁具有排他性,即一个锁一次只能被一个线程持有,被称之为排他锁或互斥锁。当然,新版JDK为了性能优化，推出了读写锁,读写锁是排它锁的改进。 5.按照Java虚拟机对锁的实现方式划分,Java平台中的锁**包括==内部锁==(主要是通过synchronized实现)和==显式锁==(主要是通过Lock接口及其实现类实现)**.

54、公平锁和非公平锁:

- 锁Lock分为"公平锁"和"非公平锁":
    - 公平锁表示线程获取锁的顺序是按照线程加锁的顺序来分配的,即先来先得的FIFO先进先出顺序。

    - 非公平锁就是一种获取锁的抢占机制,是随机获得锁的,先来的不一定先得到锁,可能造成某些线程一直拿不到锁,即不公平了。

56、内部锁——众所周知的synchronized

Java平台中的任何一个对象都有唯一一个与之关联的锁,这种锁被称之为监视器(或者叫内部锁)。内部锁是一种排它锁,它能保证原子性、可见性和有序性。内部锁就由synchronized关键字实现。

- synchronized可以修饰方法或者代码块.
    - synchronized修饰方法,该方法内部的代码就属于一个临界区,该方法就属于一个同步方法。此时一个线程对该方法内部的变量的更新就保证了原子性和可见性,从而实现了线程安全.
    - synchronized修饰代码块,需要一个锁句柄(一个对象的引用或者是一个可以返回对象的表达式),此时synchronized关键字引导的代码块就是临界区;同步块的锁句柄可以写为this关键字,表示当前对象,锁句柄对应的监视器就被称之为相应同步块的引导锁。
    - 作为锁句柄的变量通常以private final修饰,防止锁句柄变量的值改变之后,导致执行同一个同步块的多个线程使用不同的锁,从而避免了竞态。
    - 注意Java虚拟机会为每一个内部锁分配一个入口集用于存放等待获得相应内部锁的线程,当内部锁的持有线程释放当前锁的时候,可能是入口集中处于BLOCKED状态的线程获得当前锁也可能是处于RUNNABLE状态的其他线程。内部锁的竞争是激烈的,也是不公平的,可能等待了长时间的线程没有获得锁,也可能是没有经过等待的线程直接就获得了锁。

55、显式的加锁和解锁——Lock接口

- 在Java5.0之前,在协调对共享对象的访问时可以使用的机制只有synchronized和volatile,在Java 5.0中:Lock接口(以及其实现类如ReentrantLock等),Lock接口中定义了一组抽象的加锁操作。不同的是,synchronized可以方便的隐式的获取锁,而Lock接口则提供了一种显式获取锁(排它锁)。

58、重入锁——ReentrantLock类

- 如果一个线程持有一个锁的时候还能继续成功的申请该锁,那么我们就称该锁是可重入的,否则我们就称该锁是非可重入的。
    - ReentrantLock是一个可重入锁,ReentrantLock类与synchronized类似,都可以实现线程之间的同步互斥。但ReentrantLock类此外还扩展了更多的功能,如嗅探锁定、多路分支通知等,在使用上也比synrhronized更加的灵活。
    - 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是一个既公平又非公平的显示锁,所以在实例化ReentrantLock类时,ReentrantLock的一个构造签名为ReentrantLock(boolean fair),传入true时是公平锁。公平锁的开销较非公平锁的开销大,因此显式锁默认使用的是非公平的调度策略。
    - 默认情况下使用内部锁,而当多数线程持有一个锁的时间相对较长或者线程申请锁的平均时间间隔相对长的情况下我们可以考虑使用显式锁。
- ReentrantLock获取锁定与三种方式

- 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
- 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；
- 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

59、synchronized和ReentrantLock对比

- 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。

60、读写锁——(Read/WriteLock):主要用于读线程持有锁的时间比较长的情景下。

- ReadWriteLock接口是对读写锁的抽象,其默认的实现类是ReentrantReadWriteLock。ReadWriteLock定义了两个方法readLock()和writeLock(),分别用于返回相应读写锁实例的读锁和写锁。这两个方法的返回值类型都是Lock。
- 读写锁是一种改进型的排它锁,读写锁允许多个线程可以同时读取(只读)共享变量
    - 读写锁是分为读锁和写锁两种角色的,读线程在访问共享变量的时候必须持有相应读写锁的读锁,而且读锁是共享的、多个线程可以共同持有的;
    - 写锁是排他的,以一个线程在持有写锁的时候,其他线程无法获得相应锁的写锁或读锁。总之,读写锁通过读写锁的分离从而提高了并发性。

61、锁的替代

- 多个线程共享同一个非线程安全对象时,我们往往采用锁来保证线程安全性,但是,锁也有其弊端,比如锁的开销和在使用锁的时候容易发生死锁等
- Java中也提供了一些对于某些情况下替代锁的同步机制解决方案,如volatile关键字、final关键字、static关键字、原子变量以及各种并发容器和框架.
- 策略模式：
    - **采用线程特有对象**: 各个不同的线程创建各自的实例,一个实例只能被一个线程访问的对象就被称之为线程的特有对象。采用线程特有对象,保障了对非线程安全对象的访问的线程安全。
    - **只读共享**:在没有额外同步的情况下,共享的只读对象可以有可以由多个线程并发访问,但是任何线程都不能修改它。共享的只读对象包括不可变对象和事实不可变对象。
    - **线程安全共享**:线程安全的对象在其内部实现同步,多个线程可以通过对象的公有接口来进行访问而不需要进一步的同步。
    - **保护对**象:被保护的对象只能通过持有特定的锁来访问。保护对象包括封装在其他线程安全对象中的对象,以及已发布的并且由某个特定锁保护的对象。 > - ==volatile关键字、ThreadLocal二者在锁的某些功能上的替代作用:如下==

62、什么是ThreadLocal?

- ThreadLocal是Java里一种特殊的变量。它是为创建代价高昂的对象获取线程安全的好方法，比如你可以用ThreadLocal让SimpleDateFormat变成线程安全的，因为那个类创建代价高昂且每次调用都需要创建不同的实例所以不值得在局部范围使用它，如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。==首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全==。线程局部变量的另一个不错的例子是ThreadLocalRandom类，它在多线程环境中减少了创建代价高昂的Random对象的个数
- ThreadLocal用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择ThreadLocal变量。
- 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。

63、Java中的volatile 变量是什么?

- volatile是一个特殊的修饰符，只有成员变量才能使用它,是java提供的一种同步手段，只不过它是轻量级的同步。在Java并发程序缺少同步类的情况下，多线程对成员变量的操作对其它线程是透明的。volatile变量可以保证下一个读取操作会在前一个写操作之后发生，就是上一题的volatile变量规则。
- 所以线程都会直接读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。
- 任何被volatile修饰的变量，都不拷贝副本到工作内存，任何 修改都及时写在主存
- 要使 volatile 变量提供理想的线程安全,必须同时满足下面两个条件:
  - 对变量的写操作不依赖于当前值。
  - 该变量没有包含在具有其他变量的不变式中.

64、什么场景下可以使用volatile替换synchronized?

- 只需要保证共享资源的可见性的时候可以使用volatile替代，synchronized保证可操作的原子性一致性和可见性。volatile适用于新值不依赖于就值的情形。
- Volatile和Synchronized四个不同点：
  - 粒度不同，前者针对变量 ，后者锁对象和类
  - syn阻塞，volatile线程不阻塞
  - syn保证三大特性，volatile不保证原子性
  - syn编译器优化，volatile不优化

65、乐观锁与悲观锁(并发编程)

- **悲观锁**：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。++传统的关系型数据库里边就用到了很多这种锁机制++，比如**行锁，表锁等，读锁，写锁**等，都是在做操作之前先上锁。再比如Java里面的同步原语++synchronized关键字的实现也是悲观锁++。

    - 在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。
    - 一个线程持有锁会导致其它所有需要此锁的线程挂起。
    - 如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险。
- **乐观锁**：
    - 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。
    - CAS是乐观锁技术:当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。
    - CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

66、当一个线程进入某个对象的一个synchronized的实例方法后，其它线程是否可进入此对象的其它方法?

- A、一个线程在访问一个对象的同步方法时，另一个线程可以同时访问这个对象的非同步方法
- B、 一个线程在访问一个对象的同步方法时，另一个线程不能同时访问这个同步方法。

67、synchronized和java.util.concurrent.locks.Lock的异同?

- Lock 和 synchronized 有一点明显的区别 —— lock 必须在 finally 块中释放。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放！这一点区别看起来可能没什么，但是实际上，它极为重要。忘记在 finally 块中释放锁，可能会在程序中留下一个定时炸弹，当有一天炸弹爆炸时，您要花费很大力气才有找到源头在哪。而使用同步，JVM 将确保锁会获得自动释放。

68、SynchronizedMap和ConcurrentHashMap有什么区别?

- java5中新增了ConcurrentMap接口和它的一个实现类ConcurrentHashMap。ConcurrentHashMap提供了和Hashtable以及SynchronizedMap中所不同的锁机制,比起synchronizedMap来，它提供了好得多的并发性
    - 多个读操作几乎总可以并发地执行，同时进行的读和写操作通常也能并发地执行，而同时进行的写操作仍然可以不时地并发进行（相关的类也提供了类似的多个读线程的并发性，但是，只允许有一个活动的写线程）
- Hashtable中采用的锁机制是一次锁住整个hash表，从而同一时刻只能由一个线程对其进行操作；而ConcurrentHashMap中则是一次锁住一个桶。ConcurrentHashMap默认将hash表分为16个桶，诸如get,put,remove等常用操作只锁当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有16个写线程执行，并发性能的提升是显而易见的。前面说到的16个线程指的是写线程，而读操作大部分时候都不需要用到锁。只有在size等操作时才需要锁住整个hash表。

- 在迭代方面，ConcurrentHashMap使用了一种不同的迭代方式。在这种迭代方式中，当iterator被创建后集合再发生改变就不再是抛出ConcurrentModificationException，取而代之的是在改变时new新的数据从而不影响原有的数据，iterator完成后再将头指针替换为新的数据，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。
- CopyOnWriteArrayList可以用于什么应用场景?
    - CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出ConcurrentModificationException。在CopyOnWriteArrayList中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

69、同步方法和同步块，哪个是更好的选择?

- 同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

70、进程死锁的四个必要条件以及解除死锁的基本策略：

- 互斥条件：线程对资源的访问是排他性的，如果一个线程对占用了某资源，那么其他线程必须处于等待状态，直到资源被释放。
- 请求和保持条件：线程T1至少已经保持了一个资源R1占用,但又提出对另一个资源R2请求，而此时，资源R2被其他线程T2占用，于是该线程T1也必须等待，但又对自己保持的资源R1不释放。
- 不可剥夺条件：是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放
- 环路等待条件：在死锁发生时，必然存在一个“进程-资源环形链”。
- 解除死锁的基本策略
    - 预防死锁：通过设置一些限制条件，去破坏产生死锁的必要条件
    - 避免死锁：在资源分配过程中，使用某种方法避免系统进入不安全的状态，从而避免发生死锁
    - 检测死锁：允许死锁的发生，但是通过系统的检测之后，采取一些措施，将死锁清除掉 -解除死锁：该方法与检测死锁配合使用

71、如何避免死锁?

- 死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务，死锁的发生必须满足以下四个条件：
    - 互斥条件：一个资源每次只能被一个进程使用。
    - 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
    - 不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。
    - 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。
- 避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。这篇教程有代码示例和避免死锁的讨论细节。

- 分析死锁，我们需要查看Java应用程序的线程转储。我们需要找出那些状态为BLOCKED的线程和他们等待的资源。每个资源都有一个唯一的id，++用这个id我们可以找出哪些线程已经拥有了它的对象锁++。
- 避免嵌套锁，只在需要的地方使用锁和避免无限期等待是避免死锁的通常办法

72、怎么检测一个线程是否拥有锁?

- 在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁

73、解释一下活锁：

- 是指线程1可以使用资源，但它很礼貌，让其他线程先使用资源，线程2也可以使用资源，但它很绅士，也让其他线程先使用资源。释放完后，双方发现资源满足需求了，又都去强占资源，但是又只拿到一部分，就这样，资源在各个线程间一直往复。这样你让我，我让你，最后两个线程都无法使用资源。

74、Java中活锁和死锁有什么区别?

- 活锁和死锁类似，不同之处在于处于活锁的线程或进程的状态是不断改变的，活锁可以认为是一种特殊的饥饿。一个现实的活锁例子是两个人在狭小的走廊碰到，两个人都试着避让对方好让彼此通过，但是因为避让的方向都一样导致最后谁都不能通过走廊。简单的说就是，活锁和死锁的主要区别是前者进程的状态可以改变但是却不能继续执行。

75、如何确保线程安全，servlet线程安全吗?

- 在Java中可以有很多方法来保证线程安全---同步、使用原子类(atomic concurrent classes)、实现并发锁、使用volatile关键字、使用不变类和线程安全类。
    - 使用java.util.concurrent.atomic包中的Atomic Wrapper类。例如AtomicInteger
    - 使用java.util.concurrent.locks包中的锁。
    - 使用线程安全集合类，请查看此文章以了解ConcurrentHashMap的使用情况以确保线程安全。
    - 使用带有变量的volatile关键字使每个线程从内存中读取数据，而不是从线程缓存中读取。
- **同步是java中最简单和最广泛使用的线程安全工具,同步是我们可以实现线程安全的工具，JVM保证同步代码一次只能由一个线程执行。java关键字synchronized用于创建同步代码，在内部它使用Object或Class上的锁来确保只有一个线程正在执行同步代码。**
    - Java同步在锁定和解锁资源时起作用，在任何线程进入同步代码之前，它必须获取对象的锁定，并且当代码执行结束时，它解锁可以被其他线程锁定的资源。同时，其他线程处于等待状态以锁定同步资源。
    - 我们可以用两种方式使用synchronized关键字，一种是使一个完整的方法同步，另一种方法是创建synchronized块。
    - 当方法同步时，它会锁定Object，如果方法是静态的，它会锁定Class，因此最好使用synchronized块来锁定需要同步的方法的唯一部分。
    - 在创建synchronized块时，我们需要提供将获取锁的资源，它可以是XYZ.class或类的任何Object字段。
    - synchronized(this) 将在进入同步块之前锁定对象。
    - 您应该使用最低级别的锁定，例如，如果类中有多个同步块，并且其中一个锁定了Object，则其他同步块也将无法由其他线程执行。当我们锁定一个Object时，它会获

取Object的所有字段的锁定。
    - Java同步提供了性能成本的数据完整性，因此只有在绝对必要时才应该使用它。
    - Java同步仅在同一个JVM中工作，因此如果您需要在多个JVM环境中锁定某些资源，它将无法工作，您可能需要考虑一些全局锁定机制。
    - Java synchronized关键字不能用于构造函数和变量。
    - 最好创建一个用于同步块的虚拟私有对象，这样它的引用就不能被任何其他代码更改。例如，如果您正在同步的Object的setter方法，则可以通过其他一些代码更改其引用，以并行执行synchronized块。
    - 我们不应该例如使用字符串不应该被用于同步的是保持在常量池中的任何对象，因为如果任何其他代码也需要在同一个String锁，它会尝试从相同的参考对象上获取锁串池和即使两个代码都不相关，它们也会相互锁定。
- servlet不是线程安全的，每个servlet都只被实例化一次，每个调用都是servlet的同一个实例，并且对类变量没有线程安全，数据量大的时候容易造成异常

76、你对线程优先级的理解是什么?

- 每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。
- 我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级,默认为5。
- 我们调用setPriority(),方法来设置优先级。

77、什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)?

- 线程调度器是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间
- 一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。
- 时间分片是指将可用的CPU时间分配给可用的Runnable线程的过程。==分配CPU时间可以基于线程优先级或者线程等待的时间==。**线程调度并不受到Java虚拟机控制**，所以由应用程序来控制它是更好的选择（++也就是说不要让你的程序依赖于线程的优先级++）。

78、你如何确保main()方法所在的线程是Java程序最后结束的线程?

- 我们可以使用Thread类的joint()方法来确保所有程序创建的线程在main()方法退出前结束。
- 关于Thread类的joint()方法:
    - join()方法的作用，是等待这个线程结束,也就是说，t.join()方法阻塞调用此方法的线程(calling thread)，直到线程t完成，此线程再继续；通常用于在main()主线程内，等待其它线程完成再结束main()主线程
    - Join方法实现是通过wait（Objecte提供的方法）。当main线程调用t.join时候，main线程会获得线程对象t的锁（wait 意味着拿到该对象的锁),调用该对象的wait(等待时间)，直到该对象唤醒main线程 ，比如退出后。这就意味着main线程调用t.join时，必须能够拿到线程t对象的锁。

79、在多线程中，什么是上下文切换(context-switching)?

- 上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

80、 Java中什么是竞态条件？ 举个例子说明。

- 当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件,竞态条件会导致程序在并发情况下出现一些bugs
- 多线程对一些资源的竞争的时候就会产生竞态条件，如果首先要执行的程序竞争失败排到后面执行了，那么整个程序就会出现一些不确定的bugs。这种bugs很难发现而且会重复出现，因为线程间的随机竞争.
- 导致竞态条件发生的代码区称作临界区。在临界区中使用适当的同步就可以避免竞态条件。
    - 临界区实现方法有两种，一种是用synchronized，一种是用Lock显式锁实现。
    - 有临界区是为了让更多的其它线程能安全够访问资源，**临界区就是修改对象状态标记的代码区**。

81、一个线程运行时发生异常会怎样?

- 如果异常没有被捕获,该线程将会停止执行。
- Thread.UncaughtExceptionHandler是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。
- 当一个未捕获异常将造成线程中断的时候JVM会使用Thread.getUncaughtExceptionHandler()来查询线程的UncaughtExceptionHandler并将线程和异常作为参数传递给handler的uncaughtException()方法进行处理。

82、如何在两个线程间共享数据?

- 如果每个线程执行的代码相同，可以使用同一个Runnable对象，这个Runnable对象中有那个共享数据，例如，卖票系统就可以这么做。
- 如果每个线程执行的代码不同，这时候需要用不同的Runnable对象，例如，设计4个线程。其中两个线程每次对j增加1，另外两个线程对j每次减1，银行存取款
- 有两种方法来解决此类问题
    - 将共享数据封装成另外一个对象，然后将这个对象逐一传递给各个Runnable对象，每个线程对共享数据的操作方法也分配到那个对象身上完成，这样容易实现针对数据进行各个操作的互斥和通信
    - 将Runnable对象作为一个类的内部类，共享数据作为这个类的成员变量，每个线程对共享数据的操作方法也封装在外部类，以便实现对数据的各个操作的同步和互斥，作为内部类的各个Runnable对象调用外部类的这些方法。
- 总结：其实多线程间的共享数据最主要的还是互斥，多个线程共享一个变量，针对变量的操作实现原子性即可

如何处理项目的高并发、大数据

HTML静态化

- 如果网站的请求量过大，我们可以将页面静态化提供访问来缓解服务器压力，能够缓解服务器压力加大以及降低数据库数据的频繁交换。适合于某些访问了过大，但是内容不经常改变的页面，如首页、新闻页等

82、文件服务器

- 文件服务器就是将文件系统单独拿出来提供专注于处理文件的存储访问系统，甚至于对个文件服务器。因为对于图片这种资源的访问存储是web服务最耗资源的地方，将文件服务

器单独部署既可以将压力转移，交给专门的系统处理，又可以分担风险，如果图片服务器出现问题，那么主服务器能够保证正常，顶多就是文件请求不到。

83、负载均衡

- 负载均衡将是大型网站解决高负荷访问和大量并发请求采用的终极解决办法。
- 负载均衡建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。其原理就是将大量工作分摊到多个操作单元上进行执行，例如Web服务器、FTP服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。
- **IP负载均衡**: IP负载均衡是基于特定的TCP/IP技术实现的负载均衡。比如NAT、DR、Turning等。是最经常使用的方式。IP负载均衡可以使用硬件设备，也可以使用软件实现。硬件设备的主要产品是F5-BIG-IP-GTM（简称F5)，软件产品主要有LVS、HAProxy、NginX。其中LVS、HAProxy可以工作在4-7层，NginX工作在7层。硬件负载均衡设备可以将核心部分做成芯片，性能和稳定性更好，而且商用产品的可管理性、文档和服务都比较好。
- 四个分类
    - **软件负载均衡**:
        - 解决方案是指在一台或多台服务器相应的操作系统上安装一个或多个附加软件来实现负载均衡，如DNS Load Balance，CheckPoint Firewall-1 ConnectControl等，它的优点是基于特定环境，配置简单，使用灵活，成本低廉，可以满足一般的负载均衡需求。
        - 软件解决方案缺点也较多，因为每台服务器上安装额外的软件运行会消耗系统不定量的资源，越是功能强大的模块，消耗得越多，所以当连接请求特别大的时候，软件本身会成为服务器工作成败的一个关键；软件可扩展性并不是很好，受到操作系统的限制；由于操作系统本身的Bug，往往会引起安全问题。
    - **硬件负载均衡**:
        - 解决方案是直接在服务器和外部网络间安装负载均衡设备，这种设备通常称之为负载均衡器，由于专门的设备完成专门的任务，独立于操作系统，整体性能得到大量提高，加上多样化的负载均衡策略，智能化的流量管理，可达到最佳的负载均衡需求。
        - 负载均衡器有多种多样的形式，除了作为独立意义上的负载均衡器外，有些负载均衡器集成在交换设备中，置于服务器与Internet链接之间，有些则以两块网络适配器将这一功能集成到PC中，一块连接到Internet上，一块连接到后端服务器群的内部网络上。
        - 一般而言，硬件负载均衡在功能、性能上优于软件方式，不过成本昂贵> - F5的全称是F5-BIG-IP-GTM，是最流行的硬件负载均衡设备，其并发能力达到百万级。F5的主要特性包括：
            - 多链路的负载均衡和冗余 :可以接入多条ISP链路，在链路之间实现负载均衡和高可用。
            - 防火墙负载均衡 :F5具有异构防火墙的负载均衡与故障自动排除能力;
            - 服务器负载均衡 :这是F5最主要的功能，F5可以配置针对所有的对外提供服务的服务器配置Virtual Server实现负载均衡、健康检查、回话保持等;
            - 高可用: F5设备自身的冗余设计能够保证99.999%的正常运行时间，双机F5的故障切换时间为毫秒级。 使用F5可以配置整个集群的链路冗余和服务

器冗余，提高可靠的健康检查机制，以保证高可用。
        - 安全性:与防火墙类似，F5采用缺省拒绝策略，可以为任何站点增加额外的安全保护，防御普通网络攻击，包括DDoS、IP欺骗、SYN攻击、teartop和land攻击、ICMP攻击等。
        - 易于管理:F5提供HTTPS、SSH、Telnet、SNMP等多种管理方式，包含详尽的实时报告和历史纪录报告。同时还提供二次开发包(i-Control)。
        - F5还提供了SSL加速、软件升级、IP地址过滤、带宽控制等辅助功能。
    - **本地负载均衡**: 是指对本地的服务器群做负载均衡
        - 本地负载均衡能有效地解决数据流量过大、网络负荷过重的问题，并且不需花费昂贵开支购置性能卓越的服务器，充分利用现有设备，避免服务器单点故障造成数据流量的损失。其有灵活多样的均衡策略把数据流量合理地分配给服务器群内的服务器共同负担。即使是再给现有服务器扩充升级，也只是简单地增加一个新的服务器到服务群中，而不需改变现有网络结构、停止现有的服务。
    - **全局负载均衡**: 是指对分别放置在不同的地理位置、有不同网络结构的服务器群间作负载均衡。
        - 全局负载均衡主要用于在一个多区域拥有自己服务器的站点，为了使全球用户只以一个IP地址或域名就能访问到离自己最近的服务器，从而获得最快的访问速度，也可用于子公司分散站点分布广的大公司通过Intranet（企业内部互联网）来达到资源统一合理分配的目的。
        - 特点：
            - （1）、实现地理位置无关性，能够远距离为用户提供完全的透明服务
            - （2）、除了能避免服务器、数据中心等的单点失效，也能避免由于ISP专线故障引起的单点失效。
            - （3）。解决网络拥塞问题，提高服务器响应速度，服务就近提供，达到更好的访问质量。

84、反向代理

- 客户端直接访问的服务器并不是直接提供服务的服务器，它从别的服务器获取资源，然后将结果返回给用户。
- 理服务器是代我们访获取资源，然后将结果返回。例如，访问外网的代理服务器。反向代理服务器是我们正常访问一台服务器的时候，服务器自己调用了别的服务器。
- 反向代理就是说，用户的请求请求到负载均衡的设备上，负载均衡设备再讲请求分发到空闲的应用服务器上处理，处理完成之后再通过负载均衡设备返回给用户，这样对于用户来说，后来的分发是不可见的。
- 代理服务器我们主动使用，是为我们服务的，不需要有自己的域名；反向代理是服务器自己使用的，我们并不知道，有自己的域名。
- 反向代理的实现
    - 需要有一个负载均衡设备来分发用户请求，将用户请求分发到空闲的服务器上
    - 服务器返回自己的服务到负载均衡设备
    - 负载均衡将服务器的服务返回用户

85、动静分离

- 所谓动静分离就是将网站静态资源（HTML，JavaScript，CSS，img等文件）与后台应用分开部署，提高用户访问静态代码的速度，降低对后台应用访问。上面的文件服务器就是动静分离的一部分。
- 动静分离的一种做法是将静态资源部署在nginx上，后台项目部署到应用服务器上，根据一定规则静态资源的请求全部请求nginx服务器，达到动静分离的目标。
- **静态资源部署至CDN上**:
    - CDN（Content Delivery Network，内容分发网络）。通过发布机制将内容同步到大量的缓存节点，并在DNS服务器上进行扩展， 找到里用户最近的缓存节点作为服务提供节点。因为很难自建大量的缓存节点，所以通常使用CDN运营商的服务。目前国内的服务商很少，而且按流量计费，价格也比较昂贵。
    - 我们的方案是直接将静态资源全部存放在CDN服务器上。因为之前项目中的JavaScript,CSS以及img文件都是存放在CDN服务器上，将HTML文件一起存放到CDN上之后，可以将静态资源统一放置在一种服务器上，便于前端进行维护；而且用户在访问静态资源时，可以很好利用CDN的优点——CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。
- **后端API提供数据**:
    - 后端应用提供API，根据前端的请求进行处理，并将处理结果通过JSON格式返回至前端。目前应用主要采用Java平台开发，因此应用服务器主要是Tomcat服务器，现在也开始有部分应用采用 node进行开发，应用服务器也开始使用node服务器。
- **前后端域名**: 动静分离因为静态资源和应用服务分别部署在不同的服务器上，因此会面临域名策略的选择
    - **不同域名**:
        - 前后端采用不同域名时，需要前后端开发时兼容跨域请求的情况，开发量相对上一种会稍多一些。解决跨域方式最常用的方式就是采用JSONP，还有一种解决方式使用CORS（HTTP访问控制）允许某些域名下的跨域请求。
        - 目前在我们的项目中JSONP方式更多，CORS因为需要浏览器支持，因此只会在APP内嵌HTML5，且需要POST方式时中使用。
        - 采用不同域名的方式优点也是非常明显的，不同域名采用两个域名服务器，不同的域名服务器根据请求的不同采用不同的负载均衡策略；而且不同域名也可以邮箱方式前端携带过多的Cookie。
    - **相同域名**： 采用相同域名下，用户请求api时可以避免跨域所带来的问题，相对开发更为快速，工作量也相对小一些。
- **优点**:
    - ==api接口服务化==：动静分离之后，后端应用更为服务化，只需要通过提供api接口即可，可以为多个功能模块甚至是多个平台的功能使用，可以有效的节省后端人力，更便于功能维护。
    - ==前后端开发并行==：前后端只需要关心接口协议即可，各自的开发相互不干扰，并行开发，并行自测，可以有效的提高开发时间，也可以有些的减少联调时间
    - ==减轻后端服务器压力，提高静态资源访问速度==：后端不用再将模板渲染为html返回给用户端，且静态服务器可以采用更为专业的技术提高静态资源的访问速度。

- **缺点**
    - 在开发中可以采用前端缓存不经常变化数据的方式来解决，只有哪些经常发生变化的

数据才每次向后端请求。。。- 开发量变大，前后端交流成本升高：后端api返回的数据，往往是有自身逻辑在内的，比如返回数据中的包含status（1-处理中，2-处理成功，3-处理失败），前端需要理解status的不同含义，对应的前端操作需要理解（如，status =1 or status = 2，不可提交）。
    - 在业务高速发展时需要慎重考虑：因为开发量变大，如果在业务开始阶段，缺乏前端又要求开发速度很快，就需要慎重考虑这种方式的实现成本对业务发展的影响。

86、数据库sql优化

- 对于相同功能的sql，如果数据库的sql没有做过优化和做过优化的sql比较起来，其处理能力完全是天壤之别，其差距可以有几倍甚至几十上百上千的速度差距、资源消耗差距。所以对于一个优秀的web应用，sql优化是必须做的。

87、缓存

- 对于缓存我想大家都不陌生，缓存可以让我们将一些有时效性的、经常访问的、不便于存储数据库等的数据，我们可以将数据存储在专门的用于缓存的应用程序中，如果有必要，还可以将缓存应用服务器单独部署，如果数据量过大，我们还可以组成缓存服务器集群，比如：cache、redis等都是比较专注于缓存数据的。
- 只所以使用缓存，是因为一是减少数据库的访问压力，二是一般专注于缓存的应用对于数据的读写较于数据库都是非常快的

88、数据库读写分离

- 读写分离是为了提供程序的性能，随着用户的增加，数据库的压力也会越来越大，对数据库或者SQL的基本优化可能达不到最终的效果，
- 读写分离简单的说是**把对数据库读和写的操作分开对应不同的数据库服务器**，这样能有效地减轻数据库压力，也能减轻io压力。**主数据库提供写操作，从数据库提供读操作**。主数据库提供写操作，从数据库提供读操作，其实在很多系统中，主要是读的操作。当主数据库进行写操作时，数据要同步到从的数据库，这样才能有效保证数据库完整性。Quest SharePlex就是比较牛的同步数据工具，听说比oracle本身的流复制还好，mysql也有自己的同步数据技术。mysql只要是通过二进制日志来复制数据。通过日志在从数据库重复主数据库的操作达到复制数据目的。这个复制比较好的就是通过异步方法，把数据同步到从数据库。

89、数据库活跃数据分离

- 所谓的活跃数据就是经常用到的数据，比如经常活跃的用户数据等。不活跃数据，比如好长时间不等路的用户数据，还有几个月前的数据等等。
- 对于这种不活跃的数据参与到查询中，会很大的拖累查询速度吗，所以讲非活跃数据单独同步到一个数据库中，这样大概率的查询只需要查活跃数据的数据库，只有活跃数据的数据库查不到时，才去查非活跃数据库。

90、批量读取和延迟修改

- 高并发情况可以将多个查询请求合并到一个。同一查询，同一返回处理，降低短时间内的数据库请求数量。高并发且频繁修改的可以暂存缓存中，然后统一进行修改。

91、数据库集群和库表散列

- 通常对于一个服务器的处理的瓶颈大多在于数据库的瓶颈，对于大数据量的请求处理，单个数据库处理能力有限，所以我们可以部署多个数据库，然后将数据库组成一个集群。
- 对于数据库集群，每个成熟的数据库都有自己的解决方案，我们按照方案进行扩展即可。
- 上面提到的数据库集群由于在架构、成本、扩张性方面都会受到所采用DB类型的限制，于是我们需要从应用程序的角度来考虑改善系统架构，库表散列是常用并且最有效的解决方案。我们在应用程序中安装业务和应用或者功能模块将数据库进行分离，不同的模块对应不同的数据库或者表，再按照一定的策略对某个页面或者功能进行更小的数据库散列，比如用户表，按照用户ID进行表散列，这样就能够低成本的提升系统的性能并且有很好的扩展性。sohu的论坛就是采用了这样的架构，将论坛的用户、设置、帖子等信息进行数据库分离，然后对帖子、用户按照板块和ID进行散列数据库和表，最终可以在配置文件中进行简单的配置便能让系统随时增加一台低成本的数据库进来补充系统性能。

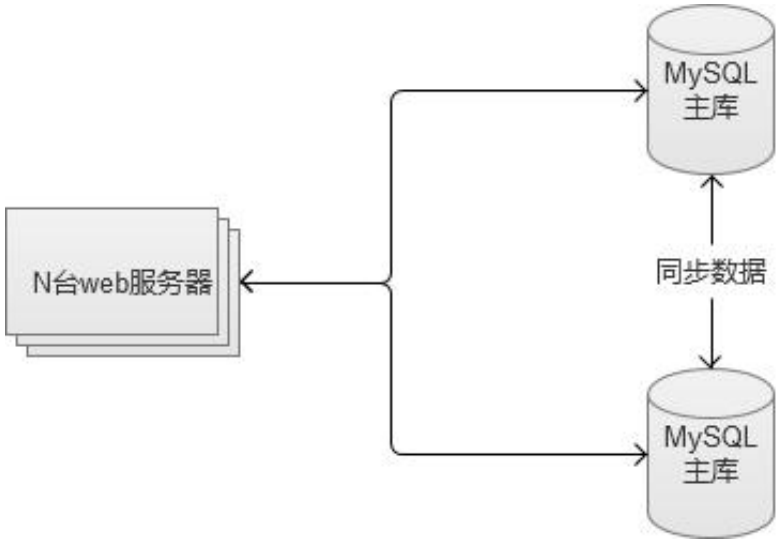
92、使用MySQL数据库内部缓存

- MySQL的缓存机制，就从先从MySQL内部开始，下面的内容将以最常见的InnoDB存储引擎为主。
- 建立恰当的索引:
    - 最简单的是建立索引，索引在表数据比较大的时候，起到快速检索数据的作用，但是成本也是有的。首先，占用了一定的磁盘空间，其中组合索引最突出，使用需要谨慎，它产生的索引甚至会比源数据更大。其次，建立索引之后的数据insert/update/delete等操作，因为需要更新原来的索引，耗时会增加。当然，实际上我们的系统从总体来说，是以select查询操作居多，因此，索引的使用仍然对系统性能有大幅提升的作用。
- 数据库连接线程池缓存
    - 如果，每一个数据库操作请求都需要创建和销毁连接的话，对数据库来说，无疑也是一种巨大的开销。为了减少这类型的开销，可以在MySQL中配置thread_cache_size来表示保留多少线程用于复用。线程不够的时候，再创建，空闲过多的时候，则销毁。
- Innodb缓存设置（innodb_buffer_pool_size）
    - innodb_buffer_pool_size这是个用来保存索引和数据的内存缓存区，如果机器是MySQL独占的机器，一般推荐为机器物理内存的80%。在取表数据的场景中，它可以减少磁盘IO。一般来说，这个值设置越大，cache命中率会越高。
- 分库/分表/分区。
    - MySQL数据库表一般承受数据量在百万级别，再往上增长，各项性能将会出现大幅度下降，因此，当我们预见数据量会超过这个量级的时候，建议进行分库/分表/分区等操作。最好的做法，是服务在搭建之初就设计为分库分表的存储模式，从根本上杜绝中后期的风险。

93、搭建MySQL数据库多台服务

- **建立MySQL主从，从库作为备份**: 在主库出故障的时候，切换到从库。不过，这种做法实际上有点浪费资源，因为从库实际上被闲着了
- MySQL读写分离，主库写，从库读。
    - 两台数据库做读写分离，主库负责写入类的操作，从库负责读的操作。并且，如果主库发生故障，仍然不影响读的操作，同时也可以将全部读写都临时切换到从库中（需要注意流量，可能会因为流量过大，把从库也拖垮）。

- 主主互备。
    - 两台MySQL之间互为彼此的从库，同时又是主库。这种方案，既做到了访问量的压力分流，同时也解决了“单点故障”问题。任何一台故障，都还有另外一套可供使用的服务。不过，这种方案，只能用在两台机器的场景。如果业务拓展还是很快的话，可以选择将业务分离，建立多个主主互备。



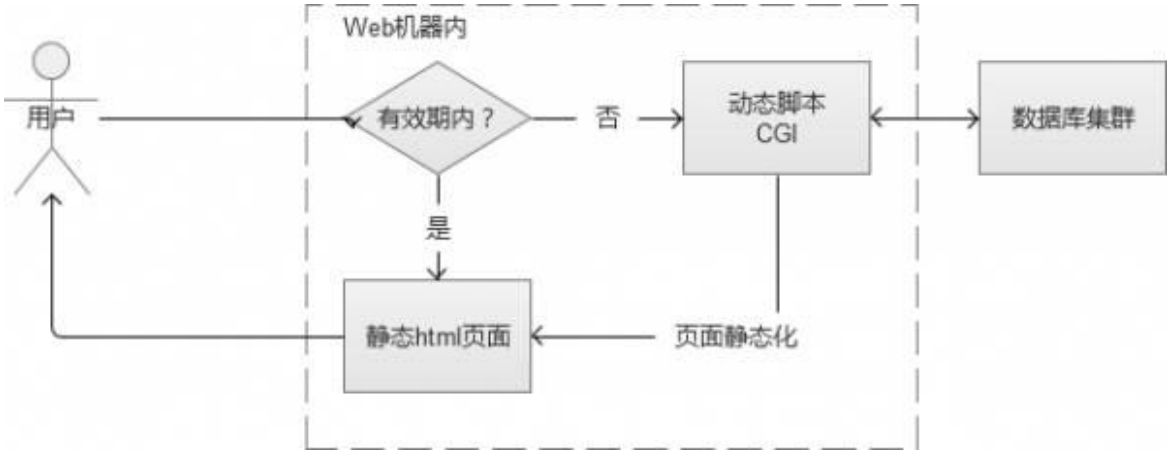
84、MySQL数据库机器之间的数据同步

- 每当我们解决一个问题，新的问题必然诞生在旧的解决方案上。当我们有多台MySQL，在业务高峰期，很可能出现两个库之间的数据有延迟的场景。并且，网络和机器负载等，也会影响数据同步的延迟。我们曾经遇到过，在日访问量接近1亿的特殊场景下，出现，从库数据需要很多天才能同步追上主库的数据。这种场景下，从库基本失去效用了。于是，解决同步问题，就是我们下一步需要关注的点。
- MySQL自带多线程同步
    - MySQL5.6开始支持主库和从库数据同步，走多线程。但是，限制也是比较明显的，只能以库为单位。MySQL数据同步是通过binlog日志，主库写入到binlog日志的操作，是具有顺序的，尤其当SQL操作中含有对于表结构的修改等操作，对于后续的SQL语句操作是有影响的。因此，从库同步数据，必须走单进程。
- 自己实现解析binlog，多线程写入。
    - 以数据库的表为单位，解析binlog多张表同时做数据同步。这样做的话，的确能够加快数据同步的效率，但是，如果表和表之间存在结构关系或者数据依赖的话，则同样存在写入顺序的问题。这种方式，可用于一些比较稳定并且相对独立的数据表。国内一线互联网公司，大部分都是通过这种方式，来加快数据同步效率。还有更为激进的做法，是直接解析binlog，忽略以表为单位，直接写入。但是这种做法，实现复杂，使用范围就更受到限制，只能用于一些场景特殊的数据库中（没有表结构变更，表和表之间没有数据依赖等特殊表）。

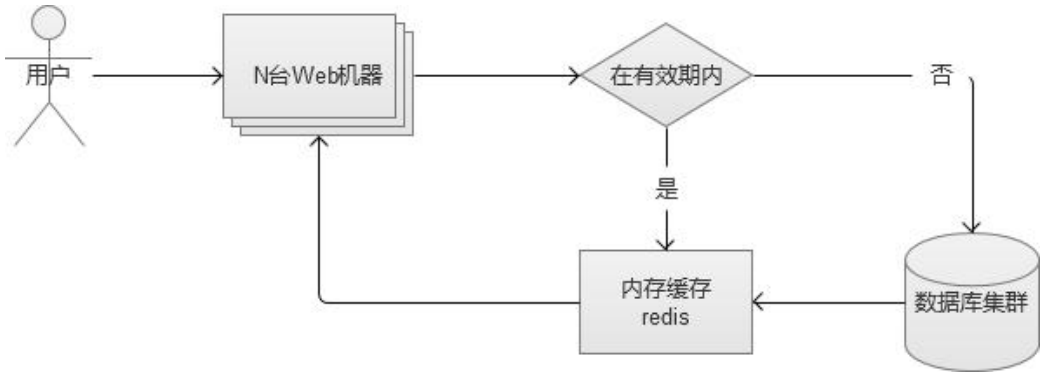
95、在Web服务器和数据库之间建立缓存

- 实际上，解决大访问量的问题，不能仅仅着眼于数据库层面。根据“二八定律”，80%的请求只关注在20%的热点数据上。因此，我们应该建立Web服务器和数据库之间的缓存机制。这种机制，可以用磁盘作为缓存，也可以用内存缓存的方式。通过它们，将大部分的热点数据查询，阻挡在数据库之前。

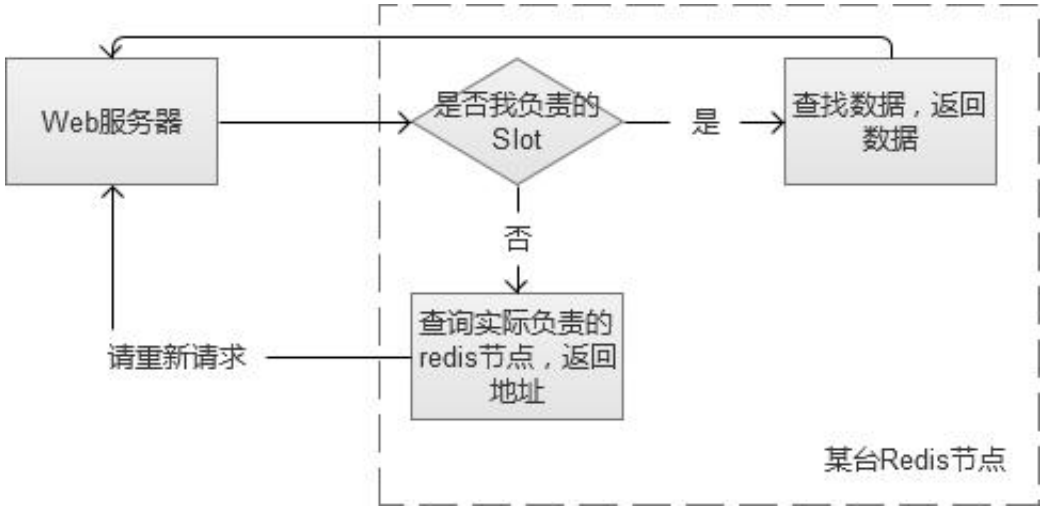
页面静态化



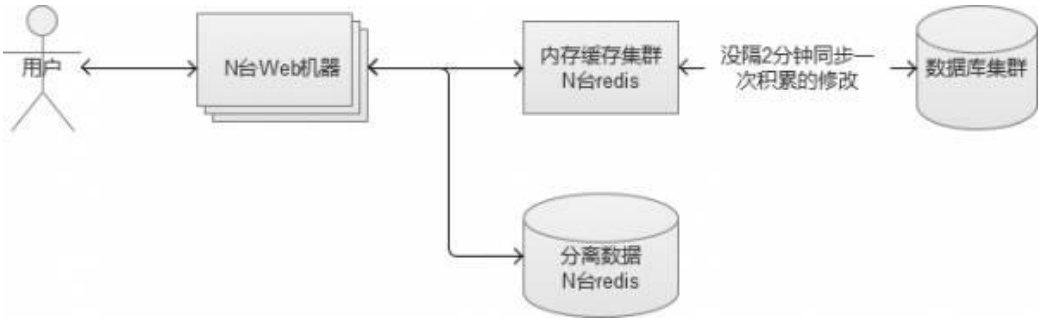
- 单台内存缓存
  - 通过页面静态化的例子中，我们可以知道将“缓存”搭建在Web机器本机是不好维护的，会带来更多问题（实际上，通过PHP的apc拓展，可通过Key/value操作Web服务器的本机内存）。因此，我们选择搭建的内存缓存服务，也必须是一个独立的服务。
  - 内存缓存的选择，主要有redis/memcache。从性能上说，两者差别不大，从功能丰富程度上说，Redis更胜一筹。



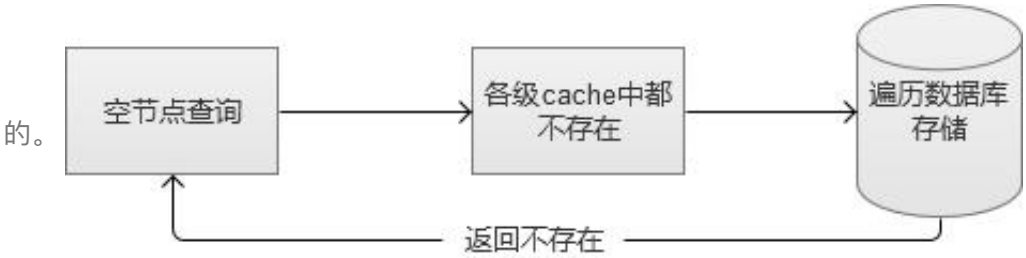
- 内存缓存集群
  - 当我们搭建单台内存缓存完毕，我们又会面临单点故障的问题，因此，我们必须将它变成一个集群。简单的做法，是给他增加一个slave作为备份机器。但是，如果请求量真的很多，我们发现cache命中率不高，需要更多的机器内存呢？因此，我们更建议将它配置成一个集群。例如，类似redis cluster。
  - Redis cluster集群内的Redis互为多组主从，同时每个节点都可以接受请求，在拓展集群的时候比较方便。客户端可以向任意一个节点发送请求，如果是它的“负责”的内容，则直接返回内容。否则，查找实际负责Redis节点，然后将地址告知客户端，客户端重新请求。
  - 除了上述通过改变系统架构的方式提升写的性能外，MySQL本身也可以通过配置参数innodb_flush_log_at_trx_commit来调整写入磁盘的策略。如果机器成本允许，从硬件层面解决问题，可以选择老一点的RAID（Redundant Arrays of independent Disks，磁盘列阵）或者比较新的SSD（Solid State Drives，固态硬盘）。



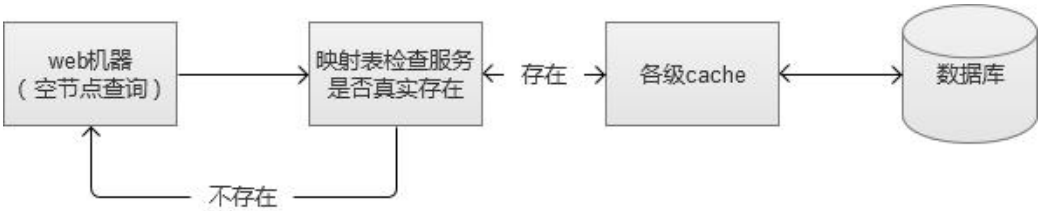
- NoSQL存储
  - 管数据库的读还是写，当流量再进一步上涨，终会达到“人力有穷时”的场景。继续加机器的成本比较高，并且不一定可以真正解决问题的时候。这个时候，部分核心数据，就可以考虑使用NoSQL的数据库。NoSQL存储，大部分都是采用key-value的方式，这里比较推荐使用上面介绍过Redis，Redis本身是一个内存cache，同时也可以当做一个存储来使用，让它直接将数据落地到磁盘。
  - 这样的话，我们就将数据库中某些被频繁读写的数据，分离出来，放在我们新搭建的Redis存储集群中，又进一步减轻原来MySQL数据库的压力，同时因为Redis本身是个内存级别的Cache，读写的性能都会大幅度提升。
  - 国内一线互联网公司，架构上采用的解决方案很多是类似于上述方案，不过，使用的cache服务却不一定是Redis，他们会有更丰富的其他选择，甚至根据自身业务特点开发出自己的NoSQL服务。



- 空节点查询问题
  - 当我们搭建完前面所说的全部服务，认为Web系统已经很强的时候。我们还是那句话，新的问题还是会来的。空节点查询，是指那些数据库中根本不存在的数据请求。例如，我请求查询一个不存在人员信息，系统会从各级缓存逐级查找，最后查到到数据库本身，然后才得出查找不到的结论，返回给前端。因为各级cache对它无效，这个请求是非常消耗系统资源的，而如果大量的空节点查询，是可以冲击到系统服务的。



  - 在我曾经的工作经历中，曾深受其害。/(ㄒoㄒ)/~~因此，为了维护Web系统的稳定性，设计适当的空节点过滤机制，非常有必要。



  - 我们当时采用的方式，就是设计一张简单的记录映射表。将存在的记录存储起来，放入到一台内存cache中，这样的话，如果还有空节点查询，则在缓存这一层就被阻挡了。

96、地理分布式部署

- **核心集中与节点分散**
  - 当一个系统和服务足够大的时候，就必须开始考虑异地部署的问题了。让你的服务，尽可能离用户更近。这个时候，异地部署就开始了。异地部署一般遵循：核心集中，节点分散。
    - 核心集中：实际部署过程中，总有一部分的数据和服务存在不可部署多套，或者部署多套成本巨大。而对于这些服务和数据，就仍然维持一套，而部署地点选择一个地域比较中心的地方，通过网络内部专线来和各个节点通讯。
    - 节点分散：将一些服务部署为多套，分布在各个城市节点，让用户请求尽可能选择近的节点访问服务。
- **节点容灾和过载保护**
  - 节点容灾是指，某个节点如果发生故障时，我们需要建立一个机制去保证服务仍然可用。毫无疑问，这里比较常见的容灾方式，是切换到附近城市节点。

    - 过载保护，指的是一个节点已经达到最大容量，无法继续接接受更多请求了，系统必须有一个保护的机制。一个服务已经满负载，还继续接受新的请求，结果很可能就是宕机，影响整个节点的服务，为了至少保障大部分用户的正常使用，过载保护是必要的。
        - 拒绝服务，检测到满负载之后，就不再接受新的连接请求。例如网游登入中的排队。
        - 分流到其他节点。这种的话，系统实现更为复杂，又涉及到负载均衡的问题。
- 避免并发
    - 如果存在并发问题，很难通过技术去解决，或者解决的代价很大，我们首先要想想是不是可以通过某些策略和业务设计来避免并发。比如通过合理的时间调度，避开共享资源的存取冲突。另外，在并行任务设计上可以通过适当的策略，保证任务与任务之间不存在共享资源。
- 串行化
    - 有的时候可以通过串行化可能产生并发问题操作，牺牲性能和扩展性，来满足对数据一致性的要求。比如分布式消息系统就没法保证消息的有序性，但可以通过变分布式消息系统为单一系统就可以保证消息的有序性了。另外，当接收方没法处理调用有序性，可以通过一个队列先把调用信息缓存起来，然后再串行地处理这些调用。