

电话面试

1 JUC

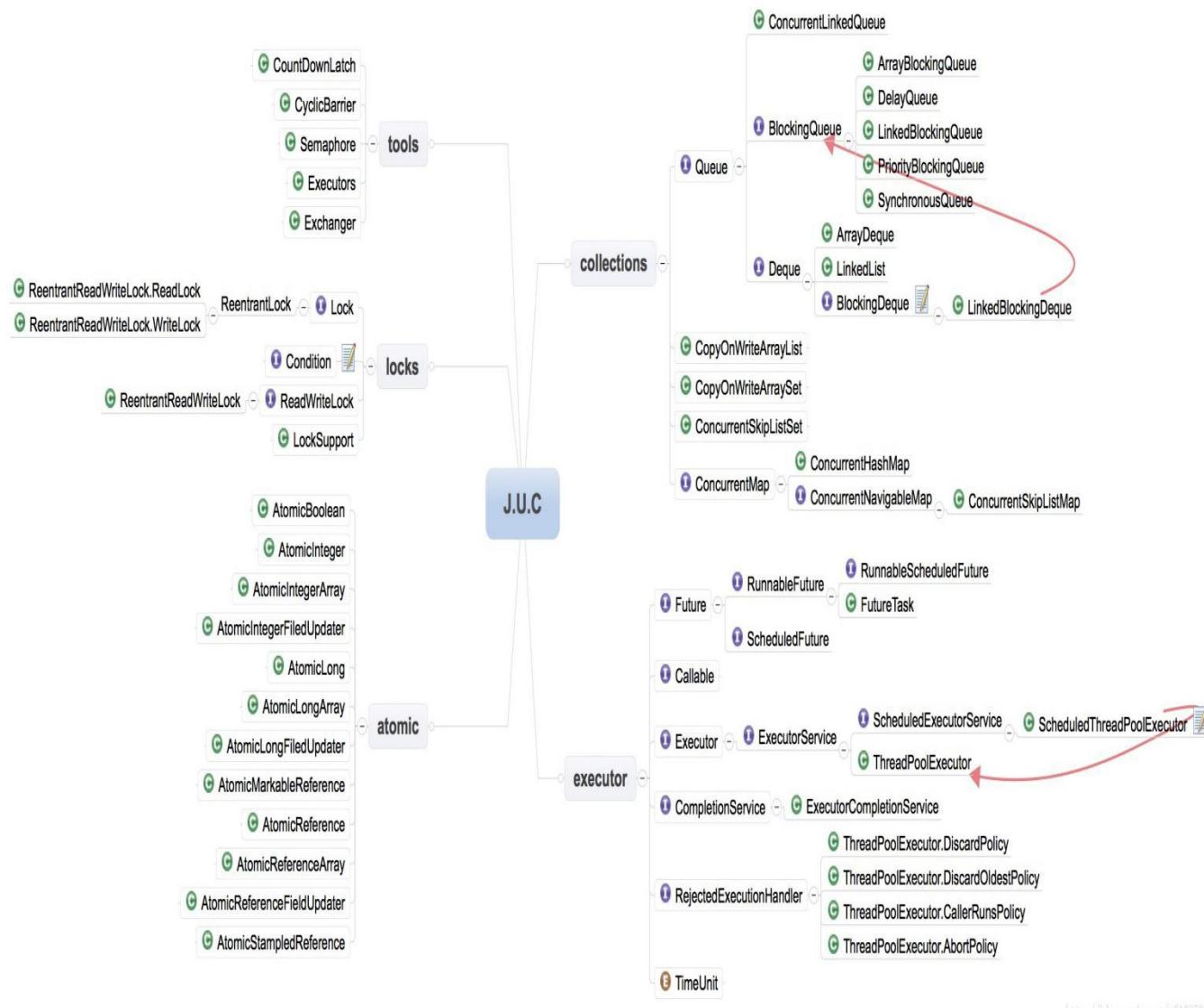
J.U.C 包下的并发类并发工具类

显示锁

原子变量类

并发集合

Executor 线程执行器



<http://blog.csdn.net/ufl08532>

2 多线程的东西同步机制

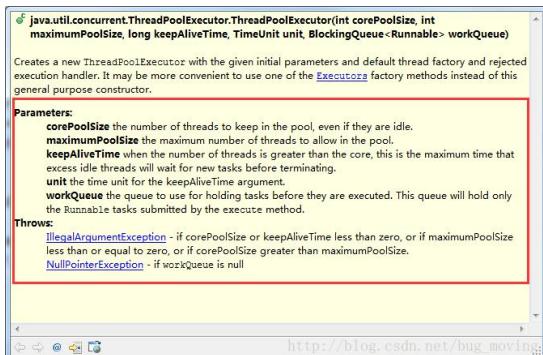
1. 多线程有什么用？

- 1 发挥多核 CPU 的优势，充分“压榨”CPU 的性能
- 2 防止阻塞。
- 3 便于建模，比如一个大的任务 A，单线程完成要考虑很多，如果分成很多小任务，多线程来完成就简单很多了。

2. 创建线程的方式

一般有三种：

- 1) 继承 Thread 类
- 2) 实现 Runnable 接口
- 3) 实现 Callable 接口
- 4) 线程池 Executor ThreadPoolExecutor



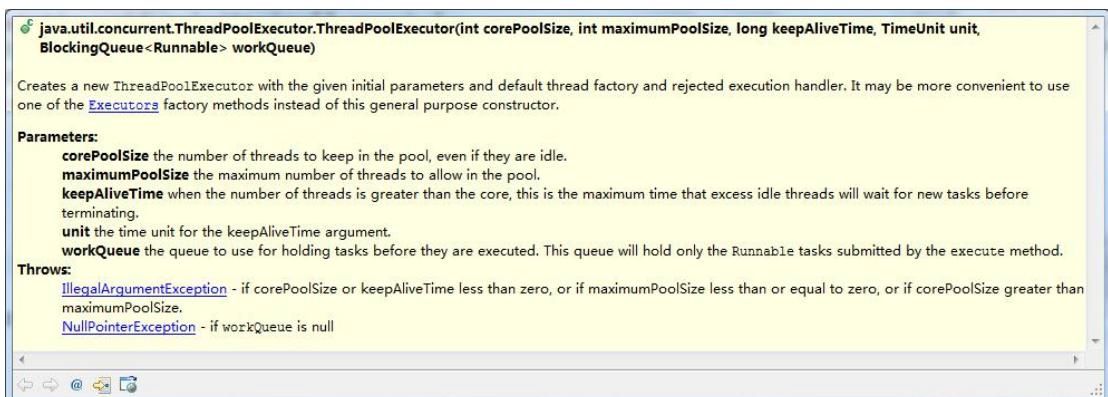
一般来说我都是通过实现 **Runnable** 接口创建线程，也觉得这种方式好一些，毕竟 Java 一直提倡面向接口编程。

3. 为什么要使用 线程池

- 1) 避免频繁地创建和销毁线程，达到线程对象的重用。创建和销毁线程的开销还是很大的)
- 2) 使用线程池还可以根据项目灵活地**控制并发的数目**。

线程池和阻塞队列

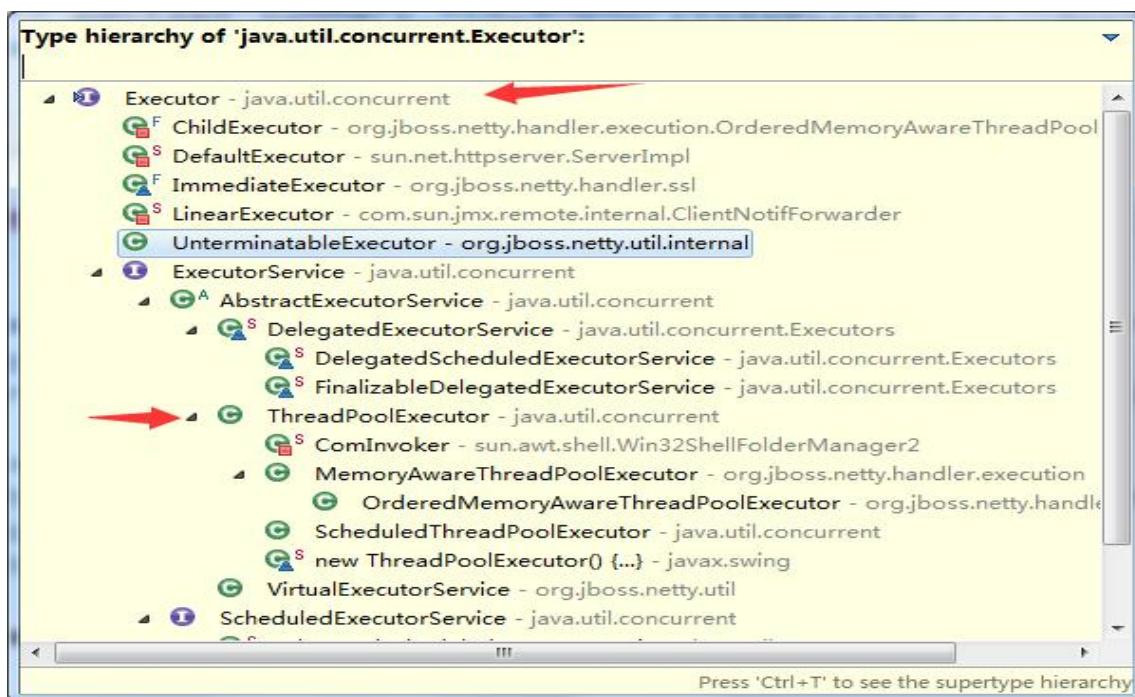
http://blog.csdn.net/bug_moving/article/details/56003645



```

* 二、线程池的体系结构：
*   java.util.concurrent.Executor : 负责线程的使用与调度的根接口
*     |--**ExecutorService 子接口：线程池的主要接口
*       |--ThreadPoolExecutor 线程池的实现类
*       |--ScheduledExecutorService 子接口：负责线程的调度
*         |--ScheduledThreadPoolExecutor : 继承 ThreadPoolExecutor，实现 ScheduledExecutor
*
* 三、工具类： Executors
* ExecutorService newFixedThreadPool() : 创建固定大小的线程池
* ExecutorService newCachedThreadPool() : 缓存线程池，线程池的数量不固定，可以根据需求自动的更改数量。
* ExecutorService newSingleThreadExecutor() : 创建单个线程池。线程池中只有一个线程
*
* ScheduledExecutorService newScheduledThreadPool() : 创建固定大小的线程，可以延迟或定时的执行任务。
*/

```



4. start()方法和 run()方法的区别

start()是 Thread 里面的方法，调用 start()方法会立刻启动线程，然后会自动的调用 run()方法，但是调用 start()方法会立即返回，不会阻塞调用线程，从而真正实现多线程。

1) start()方法来启动线程，真正实现了多线程运行。无需等待 run 方法体代码执行完毕，可以直接继续执行下面的代码；这时此线程是处于就绪状态，并没有运行。然后通过此 Thread 类调用方法 run()来完成其运行操作的，方法 run()称为线程体，它包含了要执行的这个线程的内容，Run 方法运行结束，此线程终止。然后 CPU 再调度其它线程。

2) run () 方法当作普通方法的方式调用。程序还是要顺序执行，要等待 run 方法体执行完毕后，才可继续执行下面的代码；程序中只有主线程——这一个线程，其程序执行路径还是只有一条，这样就没有达到多线程的目的。下图，第一个是 132，第二个是 123

```

@org.junit.Test
public void array(){
    System.out.println("thread1");
    Thread my = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("thread3");
        }
    });
    my.run();
    System.out.println("thread2");
}

@org.junit.Test
public void tttt(){
    System.out.println("thread1");
    Thread my = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                TimeUnit.MICROSECONDS.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("thread3");
        }
    });
    my.start();
    System.out.println("thread2");
}

```

5. Runnable 接口和 Callable 接口 (concurrent) 的区别

- 1) Runnable 接口里面的 `run()`方法是没有返回值的，所以实现 Runnable 接口一般是执行不带返回值的任务。
- 2) Callable 接口里面的 `call()`方法是有返回值的，返回值是一个泛型，和 Future、FutureTask 配合可以用来获取异步执行的结果。

有没有返回值也是这两个接口的主要区别，Callable 接口的功能更加强大，使用 Callable+Future/FutureTask 可以获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务，非常有用。

6. volatile 关键字的作用

通过 CPU 锁（总线锁，缓存锁），汇编指令会加 LOCK#，MESI 缓存一致性协议，提供了可 视性机制。volatile 关键字的作用主要：

使用 volatile 关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到 volatile 变量，一定是最新的数据。声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。在访问 volatile 变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此 volatile 变量是一种比 synchronized 关键字更轻量级的同步机制。

volatile 的一个重要作用就是和 CAS 结合，保证了原子性与可见性，详细的可以参见 java.util.concurrent.atomic 包下的类，比如 AtomicInteger。

内存屏障防止指令重排，比如 `a = 1 loadload b = 2`，编译器会先 load a

Volatile 可以避免编译时的内存乱序访问，没有办法解决运行时的内存乱序问题。

7. 什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true，否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的值 A，只要某次 CAS 操作失败，永远都不可能成功。

```
public final boolean compareAndSet(int expect,int update)
```

如果当前值 == 预期值，则以原子方式将当前值 设置为给定的更新值。

参数：

expect - 预期值

update - 新值

返回：

如果成功，则返回 true。返回 False 指示实际值与预期值不相等。

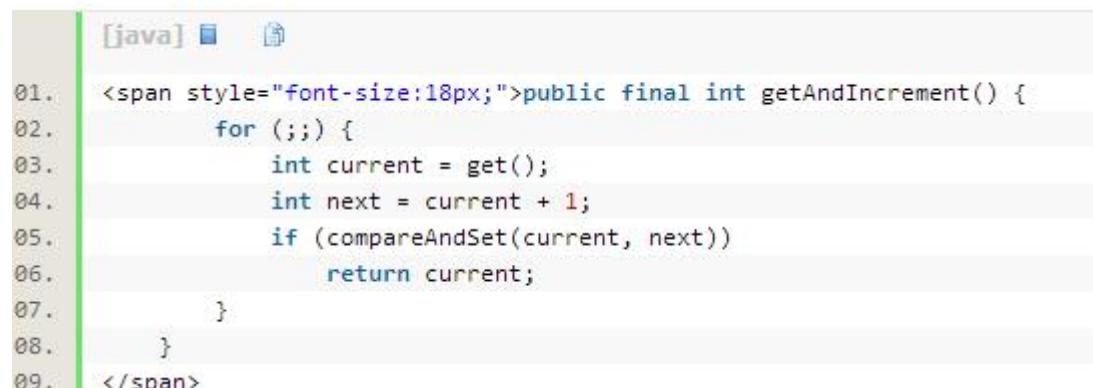
该函数 只有两个参数，可操作的确实三个值，即 value,expect, update. 他使用了 由硬件保证其原子性的指令 CAS (compare and swap)。

比如原子类：AtomicInteger 就是 volatile+CAS 实现的。getAndIncrease

8 atomicinteger addandget 和 getandadd

AtomicInteger，一个提供原子操作的 Integer 的类。在 Java 语言中，**++i 和 i++操作并不是线程安全的**。在使用的时候，不可避免的会用到 synchronized 关键字。而 AtomicInteger 则通过一种线程安全的加减操作接口。

AtomicInteger **并没有使用 Synchronized 关键字实现原子性**，几乎所有的数据更新都用到了 compareAndSet(int expect, int update)这个方法。那么就不难看出 AtomicInteger 这个类的**最核心的函数就是 compareAndSet(int expect, int update)**。



```
01. <span style="font-size:18px;">public final int getAndIncrement() {  
02.     for (;;) {  
03.         int current = get();  
04.         int next = current + 1;  
05.         if (compareAndSet(current, next))  
06.             return current;  
07.     }  
08. }  
09. </span>
```

单看这段 代码 很难保证原子性， 因为根本没有更新 value 的操作

重点在于 compareAndSet() 函数

compareAndSet 函数保证了 比较，赋值这两步操作可以通过一个原子操作完成。

然后看整个函数， 所有代码被放到了一个循环里面， 如果 compareAndSet () 执行失败，则说明 在 int current = get(); 后，其他线程对 value 进行了更新， 于是就循环一次，重新获取当前值，**直到 compareAndSet () 执行成功为止**。

这里需要注意的是 AtomicInteger 所利用的是基于冲突检测的乐观并发策略（CAS 自旋锁）。所以这种乐观在线程数目非常多的情况下，失败的概率会指数型增加。

9. 如何在两个线程之间共享数据

通过线程之间共享对象就可以了，然后通过 `wait/notify/notifyAll`、`await/signal/signallAll` 进行唤起和等待，这里需要注意的是要保证多线程环境下的数据安全性。比方说阻塞队列 `BlockingQueue` 就是为线程之间共享数据而设计的。

10. sleep 方法和 wait 方法有什么区别

相同点： `sleep` 方法和 `wait` 方法都可以用来放弃 CPU 一定的时间。

不同点：在于如果线程持有某个对象的锁，`sleep` 方法不会放弃这个对象的锁； `wait` 方法会放弃这个对象的锁。

11. 生产者消费者模型的作用是什么

- 1) 解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约。
- 2) 通过平衡生产者的生产能力 和 消费者的消费能力来提升整个系统的运行效率,这是生产者消费者模型最重要的作用。

12. ThreadLocal 有什么用

简单说 `ThreadLocal` 就是一种以 `空间换时间` 的做法，在每个 `Thread` 里面维护了一个以开地址法实现的 `ThreadLocal.ThreadLocalMap`，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了。

当使用 `ThreadLocal` 维护变量时，`ThreadLocal` 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。该作用也是 `ThreadLocal` 所表达的含义，就是线程私有的变量。

其实现的思路很简单：在 `ThreadLocal` 类中有一个虚拟的 `Map`（这里之所以说是一个虚拟的 `Map` 是因为这个 `Map` 本身不存在，只是对于每个线程 `Thread` 对象里面有一个 `ThreadLocalMap` 对象），`ThreadLocalMap` 用于存储每一个线程的变量副本。虚拟 `Map` 中元素的键为线程对象，而值对应线程的变量副本 `ThreadLocalMap` 中的值。

<http://blog.csdn.net/u010853261/article/details/55105173>

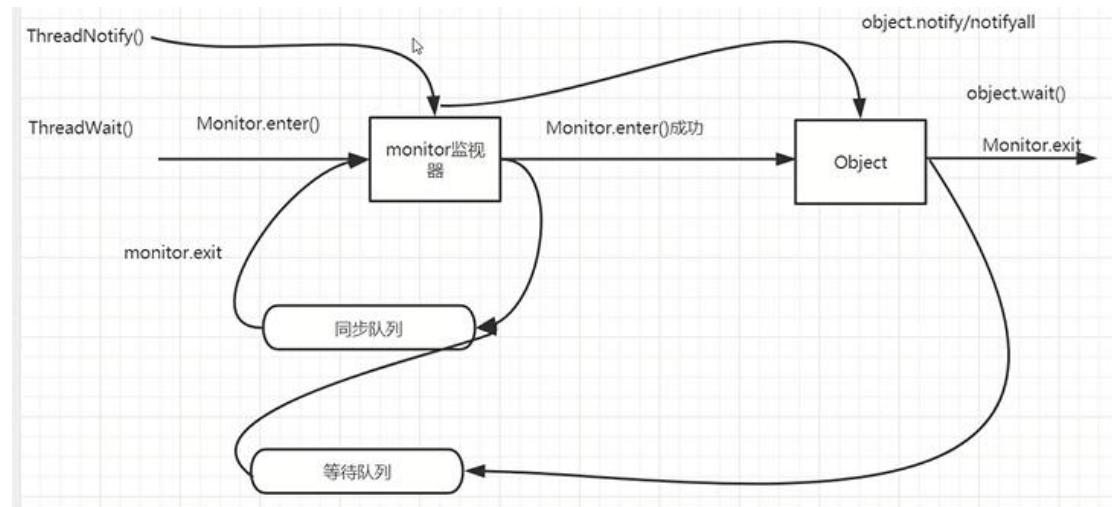
13. 为什么 `wait()` 方法和 `notify()`/`notifyAll()` 方法要在同步块中被调用

这是 JDK 强制的，`wait()` 方法和 `notify()`/`notifyAll()` 方法在调用前都必须先获得对象的锁。

14. wait()方法和 notify()/notifyAll()方法在放弃对象锁时有什么区别

wait()方法立即释放对象锁，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象的锁。

notify 是随意唤醒一个线程。



yield()应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用 yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证 yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。一个线程从 wait()状态醒来是不是一定被 notify()了？

答案是不一定。当在 obj 对象上调用 wait 操作的时候，就会释放当前持有的锁，并将线程加入到 obj 所属的条件队列，而后阻塞，直到有其它线程在该 obj 上调用了 notify 操作或阻塞线程被中断或 wait 超时。

15. CyclicBarrier 和 CountDownLatch 的区别

concurrent

这两个类都是 J.U.C 并发包下面非常有用的类：

1) CountDownLatch(闭锁)相当于一扇门：在闭锁到达结束状态之前（闭锁的结束状态也就是闭锁的计数器减到零），这扇门一直是关闭的，并且没有任何线程可以通过；当闭锁到达结束状态时，这扇门会打开并允许所有的等待线程(在闭锁上调用了 await()方法的线程)通过。并且当闭锁到达结束状态后不可逆转，这扇门会一直保持打开的状态。应用场景：确保某些活动直到其他活动都结束了才继续执行。

***所有线程执行到一个地方都调用了 await 方法，latch.countDown();把所有等待的线程放过去，没办法回头，因为并不是这几个人线程 await 的，而是在主线程里面去 latch.await();

2) CyclicBarrier 栅栏，它允许一组线程互相等待，直到都到达某个公共屏障点(common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不断地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环的

barrier。CyclicBarrier 就象它名字的意思一样，可看成是个障碍，所有的线程必须到齐后才能一起通过这个障碍。

```
class Runner implements Runnable {
    // 一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)
    private CyclicBarrier barrier; ←
    private String name;

    public Runner(CyclicBarrier barrier, String name) {
        super();
        this.barrier = barrier;
        this.name = name;
    }
    @Override
    public void run() {
        try {
            Thread.sleep(1000 * (new Random()).nextInt(8));
            System.out.println(name + " 准备好了...");
            // barrier 的 await 方法，在所有参与者都已经在此 barrier 上调用 await
            barrier.await(); ←
        } catch (InterruptedException e) {
    }
```

Await 里面有 doawait 方法，然后方法里加重入锁还有一个计数的 count

```
private int doawait(boolean timed, long nanos)
    throws InterruptedException, BrokenBarrierException,
           TimeoutException {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        final Generation g = generation;

        if (g.broken)
            throw new BrokenBarrierException();

        if (Thread.interrupted()) {
            breakBarrier();
            throw new InterruptedException();
        }

        int index = --count;
        if (index == 0) { // tripped
    }
```

***这个之所以可以循环利用，是因为是每个线程自己去 await，然后通过 count 来保证多少个线程互相等待，下次还是可以那么多线程互相等待。

16. 怎么检测一个线程是否持有对象监视器

有方法可以判断某个线程是否持有对象的锁：Thread 类提供了一个 `holdsLock(Object obj)` 方法，当且仅当对象 obj 的监视器被某条线程持有的时候才会返回 true，注意这是一个 static 方法，这意味着“某条线程”指的是当前线程。

17. synchronized 和 ReentrantLock（重入锁）的区别

（重入性）所以在 Java 内部，同一线程在调用自己类中其他 synchronized 方法/块或调用父类的 synchronized 方法/块都不会阻碍该线程的执行，就是说同一线程对同一个对象锁是可重入的，而且同一个线程可以获取同一把锁多次，也就是可以多次重入。

`synchronized` 是关键字由 JVM 底层实现，`ReentrantLock` 是类，这是二者的**本质区别**。既然 `ReentrantLock` 是类，那么它就提供了比 `synchronized` 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，`ReentrantLock` 比 `synchronized` 的扩展性体现在几点上：

- (1) `ReentrantLock` 可以对获取锁的等待时间进行设置，这样就避免了死锁
- (2) `ReentrantLock` 可以获取各种锁的信息
- (3) `ReentrantLock` 可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。`ReentrantLock` 底层调用的是 `Unsafe` 的 `park` 方法加锁，`synchronized` 操作的是对象头中 `mark word`。

* Synchronized Lock 内部实现

`synchronized` 给出的答案是在软件层面依赖 JVM，而 Lock 给出的方案是在硬件层面依赖特殊的 CPU 指令，大家可能会进一步追问：JVM 底层又是如何实现 `synchronized` 的？

`synchronized` 的底层实现主要依靠 Lock-Free 的队列 基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。

<http://www.open-open.com/lib/view/open1352431526366.html>

大湿的回答：可以先说重量级锁，也就是用系统提供的 `mutex` 来实现，竞争失败的就放到等待队列里，由 `mutex` 来搞唤醒的操作。这个等待队列不知道是不是无锁的。然后轻量级锁，就是 `cas` 修改对象头状态，标记这个对象被我这个线程锁住了，解锁的时候再改回来，要是发生竞争，自旋几次后，轻量级锁升级为重量级锁。最后是偏向锁，也是 `cas` 修改对象头状态，但是解锁了也不改回来，要是发生竞争就比较麻烦，得遍历相关的栈帧，如果没有持有锁，就重偏向给自己。要是正在持有锁，就得膨胀成轻量级锁还是重量级锁。

* synchronized

```
public synchronized void test();  
descriptor: ()V  
flags: ACC_PUBLIC, ACC_SYNCHRONIZED  
Code:  
stack=0, locals=1, args_size=1  
0: return  
LineNumberTable:  
line 12: 0  
LocalVariableTable:  
Start Length Slot Name Signature  
0 1 0 this Lcom/gupaoedu/michael/ThreadDemo;
```

```

public void demo1();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=2, locals=3, args_size=1
  0: aload_0
  1: dup
  2: astore_1
  3: monitorenter
  4: aload_1
  5: monitorexit
  6: goto      14
  9: astore_2
 10: aload_1
 11: monitorexit
 12: aload_2
 13: athrow
 14: return
Exception table:
  from   to target type

```

- A. 无论 `synchronized` 关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果 `synchronized` 作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。
- B. 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。
- C. 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

Jdk1.6 以后，锁的优化：无锁->偏向锁（默认）->轻量级锁、自旋锁->重量级锁

锁的膨胀模型，以及锁的优化原理，为什么要这样设计

对象头、实例数据、对齐填充

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁态	对象的hashCode		分代年龄	0	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量(重量级锁)的指针				10
GC标记	空				11
偏向锁	线程ID	Epoch	分代年龄	1	01

```

class markOopDesc: public oopDesc {
private:
    // Conversion
    uintptr_t value() const { return (uintptr_t) this; }

public:
    // Constants
    enum { age_bits      = 4,
           lock_bits     = 2,
           biased_lock_bits = 1,
           max_hash_bits = BitsPerWord - age_bits - lock_bits - biased_lock_bits,
           hash_bits     = max_hash_bits > 31 ? 31 : max_hash_bits,
           cms_bits      = LP64_ONLY(1) NOT_LP64(0),
           epoch_bits    = 2
    };

    // The biased locking code currently requires that the age bits be
    // contiguous to the lock bits.
    enum { lock_shift      = 0,
           biased_lock_shift = lock_bits,
           age_shift         = lock_bits + biased_lock_bits,
           cms_shift         = age_shift + age_bits,
           hash_shift        = cms_shift + cms_bits,
           epoch_shift       = hash_shift
    };

    enum { lock_mask      = right_n_bits(lock_bits),
           lock_mask_in_place = lock_mask << lock_shift,
           biased_lock_mask   = right_n_bits(lock_bits + biased_lock_bits),
           biased_lock_mask_in_place= biased_lock_mask << lock_shift,
           biased_lock_bit_in_place = 1 << biased_lock_shift,
           age_mask           = right_n_bits(age_bits),
           age_mask_in_place = age_mask << age_shift,
           epoch_mask         = right_n_bits(epoch_bits),
           epoch_mask_in_place = epoch_mask << epoch_shift
    };
}

//%note monitor_1
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitoerenter(JavaThread* thread, BasicObjectLock* elem))
#endif ASSERT
    thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
    if (PrintBiasedLockingStatistics) {
        Atomic::inc(BiasedLocking::slow_path_entry_count_addr());
    }
    Handle h_obj(thread, elem->obj());
    assert(Universe::heap()->is_in_reserved_or_null(h_obj()),
           "must be NULL or an object");
    if (UseBiasedLocking) {
        // Retry fast entry if bias is revoked to avoid unnecessary inflation
        ObjectSynchronizer::fast_enter(h_obj, elem->lock(), true, CHECK);
    } else {
        ObjectSynchronizer::slow_enter(h_obj, elem->lock(), CHECK);
    }
    assert(Universe::heap()->is_in_reserved_or_null(elem->obj()),
           "must be NULL or an object");
#endif ASSERT
    thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
IRT_END

```

加锁之后会执行 monitoerenter 这个指令，可以看到如果偏向锁，走 fast_enter，否则走 slow_enter 也就是轻量级锁的逻辑

```

void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
    markOop mark = obj->mark();
    assert(!mark->has_bias_pattern(), "should not see bias pattern here");

    if (mark->is_neutral()) { // ←
        // Anticipate successful CAS -- the ST of the displaced mark must
        // be visible <= the ST performed by the CAS.
        lock->set_displaced_header(mark);
        if (mark == (markOop) Atomic::cmpxchg_ptr(lock, obj()->mark_addr(), mark)) {
            TEVENT (slow_enter: release stacklock);
            return;
        }
        // Fall through to inflate() ...
    } else
    if (mark->has_locker() && THREAD->is_lock_owned((address)mark->locker())) {
        assert(lock != mark->locker(), "must not re-lock the same lock");
        assert(lock != (BasicLock*)obj->mark(), "don't relock with same BasicLock");
        lock->set_displaced_header(NULL);
        return;
    }

#ifndef 0
    // The following optimization isn't particularly useful.
    if (mark->has_monitor() && mark->monitor()->is_entered(THREAD)) {
        lock->set_displaced_header(NULL);
        return;
    }
#endif

    // The object header will never be displaced to this lock,
    // so it does not matter what the value is, except that it
    // must be non-zero to avoid looking like a re-entrant lock,
    // and must not look locked either.
    lock->set_displaced_header(markOopDesc::unused_mark());
    ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
}

```

通过自旋获得轻量级锁，如果获取不到，会走 inflate 也就是锁膨胀，走重量级锁的逻辑

UsingBiasedLock 设置偏向锁

轻量级锁

通过 CAS 在对象头设 MarkWord 赋值到锁记录中，如果设置成功获得轻量级锁（现象是线程 A 拿到对象锁，执行完释放，线程 B 拿到对象锁，类似串行执行），如果不成功存在竞争，获取重量级锁。

偏向锁

Donglee: 在很多情况下，锁不仅不会存在竞争，而且很多时候是由同一个线程执行的。线程 A 执行，获取锁之后，会记录到对象头，下次再进来的时候，会比较，如果还是那个锁，就不用重复获得。如果一个同步代码库实际上只有一个线程在运行，每次都要获取释放锁很费时。

与 Concurrent 包下的 Lock 的区别和联系

Lock 能够实现 synchronized 的所有功能，同时，能够实现长时间请求不到锁时自动放弃、通过构造方法实现公平锁、出现异常时 synchronized 会由 JVM 自动释放，而 Lock 必须手动释放，因此我们需要把 unLock()方法放在 finally{}语句块中

锁的优化策略

①读写分离②分段加锁③减少锁持有的时间④多个线程尽量以相同的顺序去获取资源等，这些都不是绝对原则，都要根据情况，比如不能将锁的粒度过于细化，不然可能会出现线程的加锁和释放次数过多，反而效率不如一次加一把大锁。这部分跟面试官谈了很久

18. 隐式(Synchronized)锁跟显式(java.util.Lock) 锁

显式和隐式锁都能实现对共享资源的控制，两者在内存同步上是同样的机制，但是显式锁提供了更灵活更强大的接口

1. `synchronized` 对多个锁只能按照获得锁的顺序的反序释放(先获得后释放)，显式锁可以按照需要释放锁，无此约束

2. 显式锁提供可中断的获取锁的方法，`lockInterruptibly`

3. 显式提供尝试获得锁方法

4. 提供精度更细的等待与唤醒(利用 Condition)

特别注意显式锁的 `xx.lock()` 方法只是获取了 `xx` 对象表达的锁，并不是获取了 `xx` 内置的隐式锁，这个要注意区分，与 `synchronized(xx)` 是两回事

19. 什么是乐观锁和悲观锁

其实这两种锁就是对线程安全的最悲观和最乐观的假设。

(1) **乐观锁**：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

(2) **悲观锁**：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 `synchronized`，不管三七二十一，直接上了锁就操作资源了。

• 如果对读的响应度要求非常高，比如证券交易系统，那么适合用乐观锁，因为悲观锁会

阻塞读

• 如果读远多于写，那么也适合用乐观锁，因为用悲观锁会导致大量读被少量的写阻塞

• 如果写操作频繁并且冲突比例很高，那么适合用悲观写独占锁

20. ReadWriteLock 是什么？

首先明确一下，不是说 `ReentrantLock` 不好，只是 `ReentrantLock` 某些时候有局限。如果使用 `ReentrantLock`，可能本身是为了防止线程 A 在写数据、线程 B 在读数据造成的数据不一致，但这样，如果线程 C 在读数据、线程 D 也在读数据，读数据是不会改变数据的，没有必要

加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁 `ReadWriteLock`。`ReadWriteLock` 是一个读写锁接口，`ReentrantReadWriteLock` 是 `ReadWriteLock` 接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

写写/读写 需要“互斥”

* 读读 不需要互斥

21. Java 编程写一个会导致死锁的程序

真正理解什么是死锁，这个问题其实不难，几个步骤：

(1) 两个线程里面分别持有两个 `Object` 对象：`lock1` 和 `lock2`。这两个 `lock` 作为同步代码块的锁；

(2) 线程 1 的 `run()` 方法中同步代码块先获取 `lock1` 的对象锁，`Thread.sleep(xxx)`，时间不需要太多，50 毫秒差不多了，然后接着获取 `lock2` 的对象锁。这么做主要是为了防止线程 1 启动一下子就连续获得了 `lock1` 和 `lock2` 两个对象的对象锁

(3) 线程 2 的 `run()` 方法中同步代码块先获取 `lock2` 的对象锁，接着获取 `lock1` 的对象锁，当然这时 `lock1` 的对象锁已经被线程 1 锁持有，线程 2 肯定是要等待线程 1 释放 `lock1` 的对象锁的

这样，线程 1 “睡觉”睡完，线程 2 已经获取了 `lock2` 的对象锁了，线程 1 此时尝试获取 `lock2` 的对象锁，便被阻塞，此时一个死锁就形成了。

22. 怎么唤醒一个阻塞的线程

如果线程是因为调用了 `wait()`、`sleep()` 或者 `join()` 方法而导致的阻塞，可以中断线程，并且通过抛出 `InterruptedException` 来唤醒它；如果线程遇到了 IO 阻塞，无能为力，因为 IO 是操作系统实现的，Java 代码并没有办法直接接触到操作系统。

23. 不可变对象对多线程有什么帮助

不可变对象保证了对象的内存可见性，而且不可变对象天生就是线程安全的。

24. 什么是多线程的上下文切换

多线程的上下文切换是指 CPU 控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取 CPU 执行权的线程的过程，这其中涉及到运行环境的保存与切换，存在时间开销。

25. 如果你提交任务时，线程池队列已满，这时会发生什么

- 1) 如果使用的是无界队列，比如 `LinkedBlockingQueue`，没关系，继续添加任务到阻塞队列中等待执行，因为 `LinkedBlockingQueue` 可以近乎认为是一个无穷大的队列，可以无限存放任务；
- 2) 如果使用的是有界队列。比方说 `ArrayBlockingQueue` 的话，任务首先会被添加到 `ArrayBlockingQueue` 中，`ArrayBlockingQueue` 满了，则会使用拒绝策略 `RejectedExecutionHandler` 处理满了的任务，默认是 `AbortPolicy`。

26. Java 中用到的线程调度算法是什么

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

27. `Thread.sleep(0)` 的作用是什么

简单点说就是**主动让出一次 CPU**：

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 `Thread.sleep(0)` 手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

28. 什么是自旋

很多 `synchronized` 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 `synchronized` 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 `synchronized` 的边界做忙循环，这就是**自旋**。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

29. 什么是 Java 内存模型

(1) Java 内存模型将**内存分为了主内存和工作内存**。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次 Java 线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去

- (2) 定义了几个原子操作，用于操作主内存和工作内存中的变量
- (3) 定义了 `volatile` 变量的使用规则
- (4) **happens-before**，即先行发生原则，定义了操作 A 必然先行发生于操作 B 的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁 `unlock` 的动作一定先行发生于后面对于同一个锁进行锁定 `lock` 的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的 `happens-before` 规则，则这段代码一定是线程非安全的

30. 什么是线程安全

就我的理解来说：一段代码在单线程和多线程并发下永远都能获得一样的结果，那么就是线程安全的。

有值得一提的地方，就是线程安全也是有几个级别的：

(1) 不可变(对象)

像 String、Integer、Long 这些，都是 final 类型的类，还有 final 类型的简单变量。任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用。

(2) 绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java 中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java 中也有，比方说 CopyOnWriteArrayList、CopyOnWriteArraySet

(3) 相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像 Vector 这种，add、remove 方法都是原子操作，不会被打断，但也仅限于此，如果有线程在遍历某个 Vector、有个线程同时在 add 这个 Vector，99%的情况下都会出 ConcurrentModificationException，也就是 fail-fast 机制。

(4) 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，没有加任何保证线程安全的措施。

31. FutureTask 是什么

这个其实前面有提到过，FutureTask 表示一个异步运算的任务。FutureTask 里面可以传入一个 Callable 的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于 FutureTasksh 实现 RunnableFuture，然后这个继承 Runnable 接口的实现类，所以 FutureTask 也可以放入线程池中。

```
public interface RunnableFuture<V> extends Runnable, Future<V>
    /**
     * ...
     */
```

32. 单例模式的线程安全性

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：某个类的实例在多线程环境下只会被创建一次出来。单例模式有很多种的写法，我总结一下：

(1) 饿汉式单例模式的写法：线程安全

(2) 懒汉式单例模式的写法：非线程安全

(3) 双检锁(DCL)单例模式的写法：线程安全

33. Semaphore 有什么作用 concurrent

常用作并发数控制器。**Semaphore** 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore 有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n=1，相当于变成了一个 synchronized 了。

34. Hashtable 的 size() 方法中明明只有一条语句“`return count`”，为什么还要做同步？

这是我之前的一个困惑，不知道大家有没有想过这个问题。某个方法中如果有两条语句，并且都在操作同一个类变量，那么在多线程环境下不加锁，势必会引发线程安全问题，这很好理解，但是 size() 方法明明只有一条语句，为什么还要加锁？

关于这个问题，在慢慢地工作、学习中，有了理解，主要原因有两点：

(1) 同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程 A 在执行 Hashtable 的 put 方法添加数据，线程 B 则可以正常调用 size() 方法读取 Hashtable 中当前元素的个数，那读取到的值可能不是最新的，可能线程 A 添加了完了数据，但是没有对 size++，线程 B 就已经读取 size 了，那么对于线程 B 来说读取到的 size 一定是不准确的。而给 size() 方法加了同步之后，意味着线程 B 调用 size() 方法只有在线程 A 调用 put 方法完毕之后才可以调用，这样就保证了线程安全性

(2) CPU 执行代码，执行的不是 Java 代码，这点很关键，一定得记住。Java 代码最终是被翻译成汇编代码执行的，汇编代码才是真正可以和硬件电路交互的代码。即使你看到 Java 代码只有一行，甚至你看到 Java 代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个。一句“`return count`”假设被翻译成了三句汇编语句执行，完全可能执行完第一句，线程就切换了。

35. 线程类的构造方法、静态块是被哪个线程调用的？

这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被 new 这个线程类所在的线程所调用的，而 run 方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设 Thread2 中 new 了 Thread1，main 函数中 new 了 Thread2，那么：

- (1) Thread2 的构造方法、静态块是 main 线程调用的，Thread2 的 run() 方法是 Thread2 自己调用的
- (2) Thread1 的构造方法、静态块是 Thread2 调用的，Thread1 的 run() 方法是 Thread1 自己调用的

36. 同步方法和同步块，哪个是更好的选择？

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：**同步的范围越小越好。**

借着这一条，我额外提一点，虽说同步的范围越少越好，但是在 Java 虚拟机中还是存在着一种叫做锁粗化的优化方法，这种方法就是把同步范围变大。这是有用的，比方说 **StringBuffer**，它是一个线程安全的类，自然最常用的 **append()** 方法是一个同步方法，我们写代码的时候会**反复 append 字符串，这意味着要进行反复的加锁->解锁，这对性能不利**，因为这意味着 Java 虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此 Java 虚拟机会将多次 **append** 方法调用的代码进行一个**锁粗化**的操作，将多次的 **append** 的操作扩展到 **append** 方法的头尾，变成一个大的同步块，这样就减少了加锁 -> 解锁的次数，有效地提升了代码执行的效率。

37. 什么是 AQS? (*)

简单说一下 AQS，AQS 全称为 **AbstractQueuedSynchronizer**，翻译过来应该是抽象队列同步器。如果说 **java.util.concurrent** 的基础是 CAS 的话，那么 AQS 就是整个 **Java 并发包的核心了**，**ReentrantLock**、**CountDownLatch**、**Semaphore** 等等都用到了它。AQS 实际上以双向队列的形式连接所有的 **Entry**，比方说 **ReentrantLock**，所有等待的线程都被放在一个 **Entry** 中并连成双向队列，前面一个线程使用 **ReentrantLock** 好了，则双向队列实际上的第一个 **Entry** 开始运行。

AQS 定义了对双向队列所有的操作，而只开放了 **tryLock** 和 **tryRelease** 方法给开发者使用，开发者可以根据自己的实现重写 **tryLock** 和 **tryRelease** 方法，以实现自己的并发功能。

38. 高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

(1) 高并发、任务执行时间短的业务，**线程池线程数可以设置为 CPU 核数+1**，减少线程上下文的切换

(2) 并发不高、任务执行时间长的业务要区分开看：

a) 假如是业务时间长集中在 **IO 操作上**，也就是 **IO 密集型的任务**，因为 **IO 操作并不占用 CPU**，所以不要让所有的 **CPU 闲下来**，可以加大线程池中的线程数目，让 **CPU 处理更多的业务**

b) 假如是业务时间长集中在 **计算操作上**，也就是 **计算密集型任务**，这个就没办法了，和 (1) 一样吧，**线程池中的线程数设置得少一些**，减少线程上下文的切换

(3) 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考 (2)。最后，业务执行时间长的问题，也可能需要分析一下，看看能

不能使用中间件对任务进行拆分和解耦。

39. wait 方法是否必须用在 while 循环体中

是，《Effective Java》第二版中文版第 69 条 244 页位置对这一点说了一页，我看着一知半解。我能理解的一点是：对于从 `wait` 中被 `notify` 的进程来说，**它在被 `notify` 之后还需要重新检查是否符合执行条件，如果不符，就必须再次被 `wait`，如果符合才能往下执行**。所以：`wait` 方法应该使用循环模式来调用。按照上面的生产者和消费者问题来说：**错误情况一**：如果有两个生产者 A 和 B，一个消费者 C。当存储空间满了之后，生产者 A 和 B 都被 `wait`，进入等待唤醒队列。当消费者 C 取走了一个数据后，如果调用了 `notifyAll()`，注意，此处是调用 `notifyAll()`，则生产者线程 A 和 B 都将被唤醒，如果此时 A 和 B 中的 `wait` 不在 `while` 循环中而是在 `if` 中，则 A 和 B 就不会再次判断是否符合执行条件，都将直接执行 `wait()` 之后的程序，那么如果 A 放入了一个数据至存储空间，则此时存储空间已经满了；但是 B 还是会继续往存储空间里放数据，错误便产生了。**错误情况二**：如果有两个生产者 A 和 B，一个消费者 C。当存储空间满了之后，生产者 A 和 B 都被 `wait`，进入等待唤醒队列。当消费者 C 取走了一个数据后，如果调用了 `notify()`，则 A 和 B 中的一个将被唤醒，假设 A 被唤醒，则 A 向存储空间放入了一个数据，至此空间就满了。A 执行了 `notify()` 之后，如果唤醒了 B，那么 B 不会再次判断是否符合执行条件，将直接执行 `wait()` 之后的程序，这样就导致向已经满了的数据存储区中再次放入数据。错误产生。

40. Runnable 和 Thread 的区别

在程序开发中只要是多线程肯定永远以实现 `Runnable` 接口为主，因为实现 `Runnable` 接口相比继承 `Thread` 类有如下好处：

- **避免单继承的局限，一个类可以继承多个接口。**
- **适合于资源的共享**

以卖票程序为例，通过 `Thread` 类完成要卖出 30 张票。虽然现在程序中有三个线程，但是一共卖了 10 张票，也就是说使用 `Runnable` 实现多线程可以达到资源共享目的。

41. Servlet 是线程安全的吗？

`Servlet` 不是线程安全的。

要解释为什么 `Servlet` 为什么不是线程安全的，需要了解 `Servlet` 容器（即 `Tomcat`）使如何响应 `HTTP` 请求的。

当 `Tomcat` 接收到 `Client` 的 `HTTP` 请求时，`Tomcat` 从线程池中取出一个线程，之后找到该请求对应的 `Servlet` 对象。如果该 `Servlet` 还未被请求过，那么将进行 `Servlet` 初始化并调用 `Servlet` 并调用 `service()` 方法。否则，直接调用 `service()` 方法。要注意的是**每一个 `Servlet` 对象再 `Tomcat` 容器中只有一个实例对象，即是单例模式。如果多个 `HTTP` 请求请求的是同一个 `Servlet`，那么着两个 `HTTP` 请求对应的线程将并发调用 `Servlet` 的 `service()` 方法。**

这时候，如果在 `Servlet` 中定义了实例变量或静态变量，那么可能会发生线程安全问题（因

为所有的线程都可能使用这些变量）。

42. condition 实现控制多线程执行顺序 Condition-线程通信更高效的方式

在 Condition 中, 用 `await()` 替换 `wait()`, 用 `signal()` 替换 `notify()`, 用 `signalAll()` 替换 `notifyAll()`, 传统线程的通信方式, Condition 都可以实现, 这里注意, Condition 是被绑定到 Lock 上的, 要创建一个 Lock 的 Condition 必须用 `newCondition()` 方法。

43. 多线程程序设计的 8 个规则

规则一：找到真正不相关的计算任务

规则二：尽可能地在最高层进行并行化，一个是自底向上，另一个是自顶向下

规则三：尽早针对众核趋势做好可伸缩性的规划

规则四：尽可能利用已有的线程安全库

规则五：使用合适的多线程模型

规则六：永远不要假设具体的执行顺序

规则七：尽可能使用线程本地存储或者对特定的数据加锁

规则八：敢于更换更易并行化的算法

44. 一个线程如果出现了运行时异常会怎么样

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：**如果这个线程持有某个对象的监视器，那么这个对象锁会被立即释放**

45. 如果一个线程出现异常，在线程外怎么捕获

1 实现 `UncaughtExceptionHandler` 接口来捕获抛出的异常，接口里有 `uncaughtException` 方法，还要实现 `ThreadFactory` 来创建线程的时候把自定义的异常处理器赋给线程

2 在 `Thread` 类中设置一个静态域 `setDefaultUncaughtExceptionHandler`（自定义异常处理器）
http://blog.csdn.net/bug_moving/article/details/60128564

<http://blog.csdn.net/u013256816/article/details/50417822>

这个由 JVM 调用的处理线程外的异常，先获得处理线程外异常的处理器，就是一个接口，我们自己要实现那个接口 `UncaughtExceptionHandler`，然后调用我们自己实现接口时重写的方法 `uncaughtException`

```

3 /**
4  * Dispatch an uncaught exception to the handler. This method
5  * intended to be called only by the JVM.
6 */
7 private void dispatchUncaughtException(Throwable e) {
8     getUncaughtExceptionHandler().uncaughtException(this, e);
9 }

```

46. interrupt

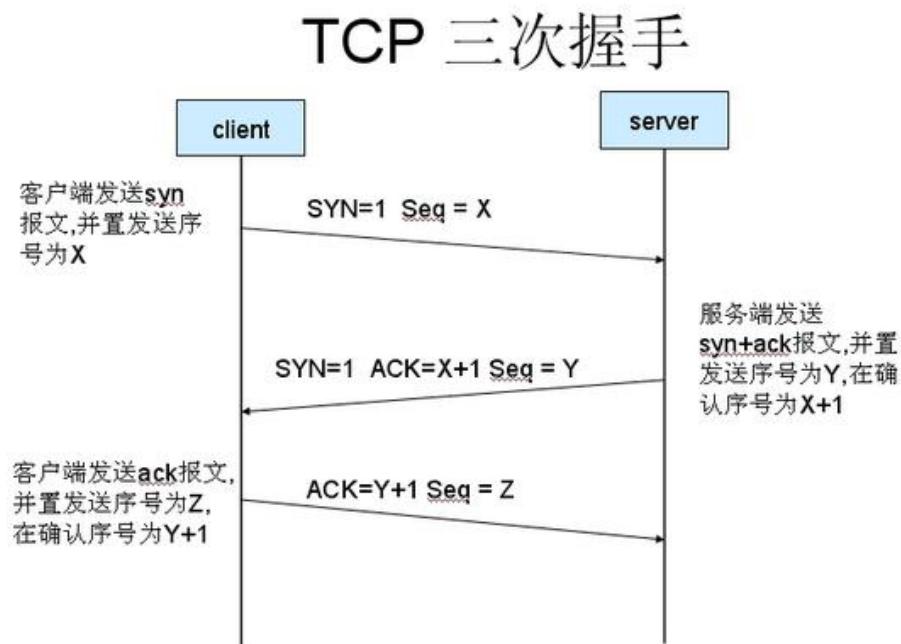
不要以为它是中断某个线程！它只是线程发送一个中断信号，让线程在无限等待时（如死锁时）能抛出抛出，从而结束线程，但是如果你吃掉了这个异常，那么这个线程还是不会中断的！

3 计算机网络

1. TCP 的三次握手、四次挥手的过程

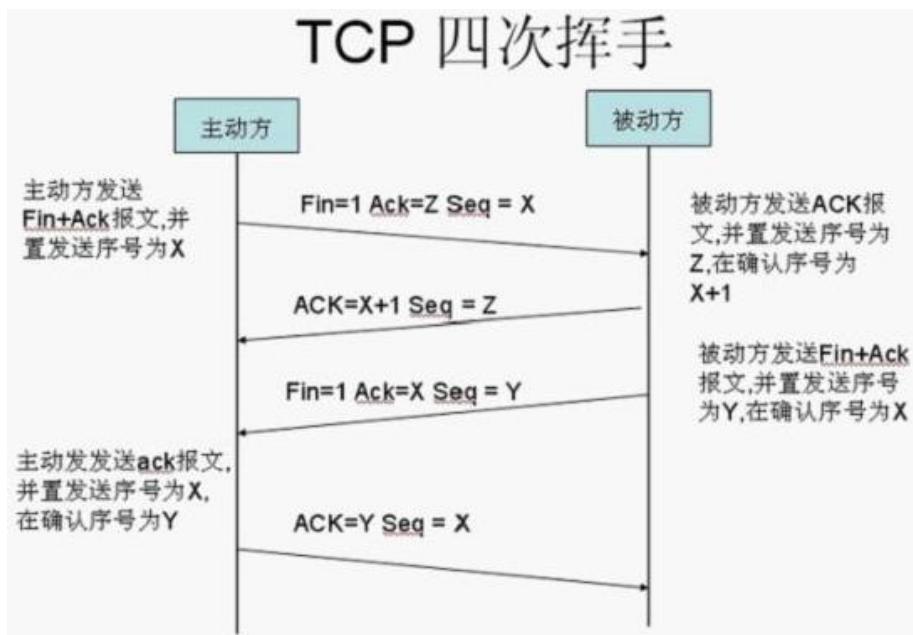
连接的三次握手

- 1) 客户端发送 SYN 信号，表示告诉服务器，我要建立连接
- 2) 服务器返回 ACK=1 和 SYN=1 给客户端，表示我已经接收到了客户端的请求，并接收了请求。
- 3) 客户端收到来自服务端响应后，知道服务端已经接受连接请求，再次发送 ACK 给服务器，确认服务端的 SYN，其实也就是一个确认的再确认。



终止的四次挥手：客户端和服务端均可主动发起挥手操作终止连接

- 1) 主动方 A 发送一个 FIN(终止信号)给被动方 B, 表示要终结主动方 A 到被动方 B 的连接;
 - 2) 被动方 B 收到了 FIN 信号, 返回 ACK 信号给主动方, 表示从主动方到被动方的连接关闭了, 也就是主动方不能再发送数据给被动方。
- 被动方 B 在发送完数据后, 给主动方 A 发送一个 FIN 信号, 请求要终结被动方 B 到主动方 A 的连接;
- 3) 主动方 A 收到了 FIN 信号, 返回 ACK 信号给被动方 B, 表示从被动方 B 到主动方 A 的连接关闭了, 也就是被动方不能再发送数据给主动方。



【问题 1】为什么连接的时候是三次握手，关闭的时候却是四次握手？

答：因为当 Server 端收到 Client 端的 SYN 连接请求报文后，可以直接发送 SYN+ACK 报文。其中 ACK 报文是用来应答的，SYN 报文是用来同步的。但是关闭连接时，当 Server 端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉 Client 端，“你发的 FIN 报文我收到了”。只有等到我 Server 端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送。故需要四步握手。

【问题 2】为什么 TIME_WAIT 状态需要经过 2MSL(最大报文段生存时间)才能返回到 CLOSE 状态？

答：虽然按道理，四个报文都发送完毕，我们可以直接进入 CLOSE 状态了，但是我们必须假象网络是不可靠的，有可能最后一个 ACK 丢失。所以 TIME_WAIT 状态就是用来重发可能丢失的 ACK 报文。

2. TCP 传输为什么是可靠的？（美团）

- 1) 将数据截取为合理的长度：TCP 将应用数据分割成认为最适合发送的数据块。UDP 完全不同，在 UDP 中应用程序产生的数据报长度将保持不变。

- 2) **超时重发:** TCP 发出数据流之后会等待接收方发出响应确认信号, 如果发送方不能及时收到确认信号会重新再次发送数据报。
- 3) **对于收到的请求, 给出确认响应:** 当 TCP 收到发自 TCP 连接另一端的数据, 它将发送一个确认。这个确认不是立即发送, 通常将推迟几分之一秒。
- 4) **TCP 会校验数据报, 如果检测出包有错, 就丢弃报文段, 不给出响应。**
- 5) **对失效数据进行重排序然后才交给应用层:** 既然 TCP 报文段作为 IP 数据报来传输, 而 IP 数据报的到达可能会失序, 因此 TCP 报文段的到达也可能会失序。如果必要, TCP 将对收到的数据进行重新排序, 将收到的数据以正确的顺序交给应用层。
- 6) **对于重复数据, 能够丢弃重复数据:** 既然 IP 数据报会发生重复, TCP 的接收端必须丢弃重复的数据。
- 7) **TCP 还能提供流量控制, 防止较快主机导致较慢主机的缓冲区溢出:** TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

TCP 使用的流量控制协议是 可变大小的滑动窗口协议。

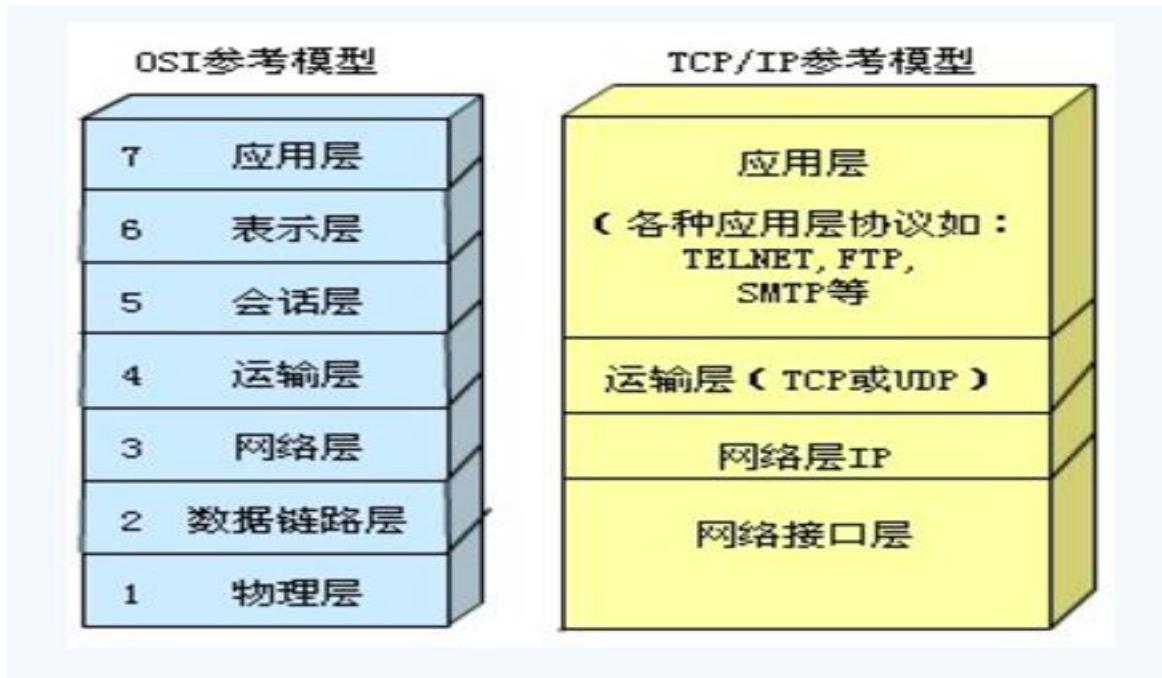
3. 为什么 TCP 建立连接是三次, 而断开连接要四次? (美团)

- 1) 连接是三次的原因?
为了防止失效的连接请求报文段突然又传送到主机 B, 因而产生错误。
- 2) 断开时四次的原因?
这是由于 TCP 的半关闭造成的, 由于 TCP 是全双工的, 表示可以同时在两个方向上既可以接收数据也可以发送数据。所以我们关闭连接必须在每个方向上面单独关闭, 这个单方向的关闭就叫半关闭。所以挥手时其实质就是 2 次断开连接, 所以是四次。

4. TCP 和 UDP 的区别

- 1) TCP 和 UDP 都是**传输层**的协议。
- 2) **TCP: 面向连接、传输可靠**(保证数据正确性, 保证数据顺序)、用于**一次传输大量数据**(流模式)、速度慢, 建立连接需要开销较多(时间, 系统资源)。比如 FTP、SMTP 之类
- 3) **UDP: 面向非连接、传输不可靠、用于一次传输少量数据**(数据包模式)、速度快。比如 QQ。

5. TCP/IP 的分层模型以及网络层和运输层的主要作用：（美团）



应用层：负责处理特定的应用程序；

传输层：负责提供端到端的通信(不同应用程序到不同应用程序，因为限制了端口号)；

网络层(IP 层)：主要是路由转发；

6 TCP/IP 的分层模型是哪几层？http 协议对应的哪一层？传输层的上面 一层是什么？（美团）

1) TCP/IP 分为四层：

2) http 属于应用层；

应用层：FTP、Http、WWW、SMTP 等等

传输层：TCP/IP

网络层 IP

网络接口层。

3) 传输层上面是应用层。

网际层协议：包括：IP 协议、ICMP 协议、ARP 协议、RARP 协议。

传输层协议：TCP 协议、UDP 协议。

应用层协议：FTP、Telnet、SMTP、HTTP、RIP、NFS、DNS。

7. http 中 GET 和 POST 的区别

1. get 是从服务器上获取数据，post 是向服务器传送数据。

2. get 的参数键值对可以在 URL 中可以看到。Post 是将表单内键值对放置在 html 的 header 中，用户看不到这些值。

3. get 传送的数据量较小，不能大于 2KB。post 传送的数据量较大，一般被默认为不受限制。

不同的浏览器(发起 http 请求)和服务器(接受 http 请求)就是不同的运输公司。虽然理论上，你可以在车顶上无限的堆货物(url 中无限加参数)。但是运输公司可不傻，装货和卸货也是有很大成本的，他们会限制单次运输量来控制风险，数据量太大对浏览器和服务器都是很大负担。(大多数)浏览器通常都会限制 url 长度在 2K 个字节，而(大多数)服务器最多处理 64K 大小的 url。超过的部分，恕不处理。如果你用 GET 服务，在 request body 偷偷藏了数据，不同服务器的处理方式也是不同的，有些服务器会帮你卸货，读出数据，有些服务器直接忽略，所以，虽然 GET 可以带 request body，也不能保证一定能被接收到哦。

对于 GET 方式的请求，浏览器会把 http header 和 data 一并发出去，服务器响应 200(返回数据)；

而对于 POST，浏览器先发送 header，服务器响应 100 continue，浏览器再发送 data，服务器响应 200 ok(返回数据)。

8 http 和 https 的区别

1) https 协议需要申请证书，一般免费证书较少，因而需要一定费用。

2) http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

3) http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

4) http 的连接很简单且是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

9. Cookie 和 session 的区别

Cookie 和 session 都是保存会话的技术：

1) cookie 保存在客户端，session 保存在服务器端；

2) cookie 可以跟踪会话，也可以保存用户喜好或者保存用户名密码；session 用来跟踪会话。

- 3) cookie 不是很安全，别人可以分析存放在本地的 cookie 并进行 cookie 欺骗，考虑到安全应当使用 session。
- 4) session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，考虑到减轻服务器性能方面，应当使用 cookie。
- 5) 这个地方可以扩展到分布式 session 服务器。

10. http 状态码的 3xx, 4xx, 5xx 分别指什么？

3xx: 重定向

常见状态代码、状态描述、说明：

200 OK //客户端请求成功

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，eg：输入了错误的 URL

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

11. 服务端 session 是如何标识已登录的用户的？

(百度)

http 是无状态的会话，需要基于 http 协议支持会话状态的机制。这时候引入 session，在会话开始时，分配一个唯一的会话标识 (sessionId)，通过 cookie 把这个标识告诉浏览器，以后每次请求的时候，浏览器的 cookie 都会带上这个会话 ID 来告诉 web 服务器请求是属于哪个会话的。这也是为什么禁用了 cookie 之后，每次操作都需要先登录。在 web 服务器上各个会话有独立的存储，保存不同会话的信息。

对于上万个用户已经登录了，服务端是怎么分辨每个用户的？

- ① 服务器在响应头内加上“Set-Cookie:XXXXXXXXXXXXXX”(相当于一个唯一的 ID 符)，此信息是服务器随机生成的，放在服务器内存里，不会重复，这就是 sessionId。
- ② 当浏览器得到这个 sessionId 会将它放在自己的进程内存里，然后你继续发请求给这个网站的时候，浏览器就会把这个 sessionId 放在请求头里发送给该服务器了，这样服务器得到 sessionId 后再和自己内存里存放的 sessionId 对比锁定客户端，从而区分不同客户端，完成会话。
- ③ 关闭浏览器结束进程，则这个 sessionId 将消失，如果用户又打开浏览器想继续这次会话的时候，就会因为发送的请求中没有这个 sessionId，而使服务器无法辨别请求身份。

12. 从浏览器打开百度，这个过程发生了什么？ (百度、头条)

- (1) 浏览器查询缓存，如果缓存存在跳到第 11 步；
- (2) 浏览器本地的 host 文件查看是否存在对应 IP 地址
- (3) DNS 域名解析，返回 IP 地址给浏览器
- (4) 浏览器打开对服务器的 TCP 连接
- (5) 浏览器通过 TCP 连接发送 HTTP 请求；
- (6) CDN 缓存静态文件，是否存在，存在就直接返回；
- (7) 经过负载均衡服务器然后到达应用服务器（这里如果是面向服务的架构，可能存在服务调用）；
- (8) 浏览器检查 HTTP 响应是否为一个重定向（3xx 结果状态码），一个验证请（401），错误（4xx 5xx）等等，这些都是不同响应的正常处理（2xx）
- (9) 如果响应可缓存，将存入缓存
- (10) 浏览器解码响应（例如：如果它是 gziped 压缩），浏览器决定如何处理这些响应（例如，它是 HTML 页面，一张图片，一段音乐）
- (11) 浏览器展现响应，对未知类型还会弹出下载对话框（现在一般不会弹出了，用户对浏览器设置而定）

note: 这个题目其实是一个很好引导面试官的题目，在我们说完大致过程之后，我们可以选择自己熟悉地方引。比如

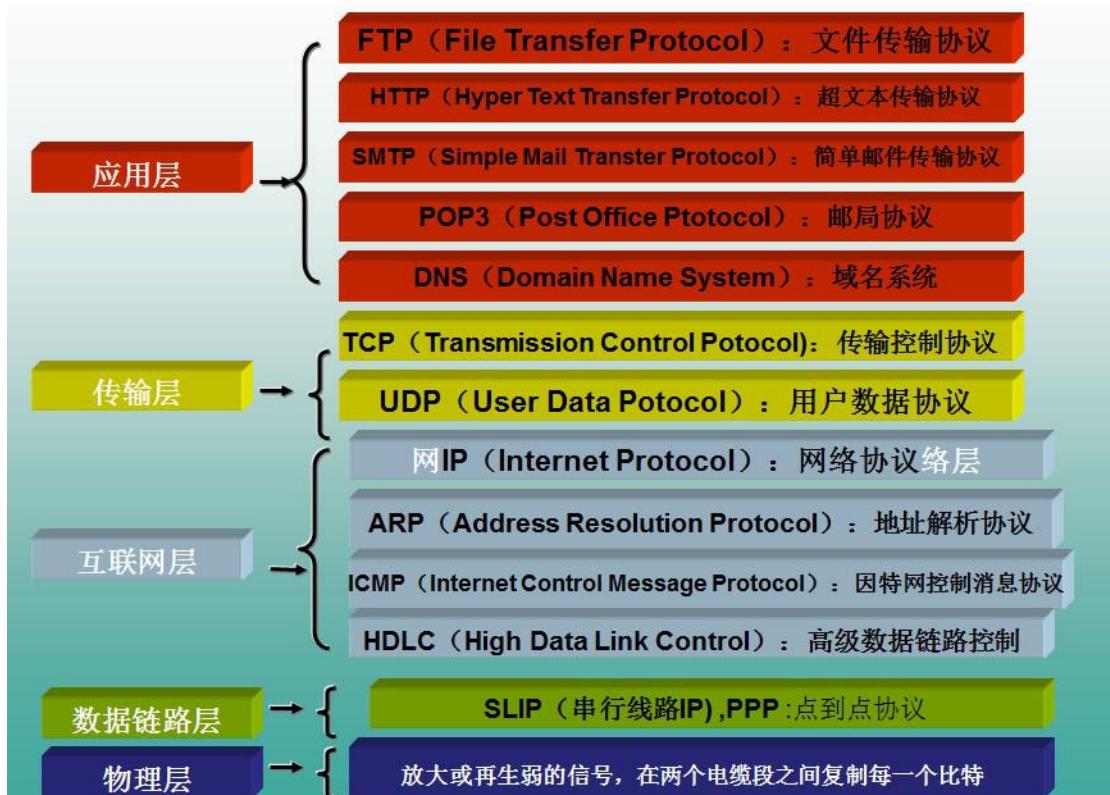
- 1) 关于负载均衡，引导负载均衡的类型以及实现负载均衡的算法；
- 2) 比如 session 定位问题，引出 session 绑定、session 复制、session 服务器集群；
- 3) 比如关于缓存我们可以引出二八定律、一致性 hash 问题。

13. http 协议概述

HTTP 请求包括三部分：**请求行(Request Line)**，**头部(Headers)** 和**实体内容(Body)**。其中，请求行由请求方法(method)，请求网址 Request-URI 和协议 (Protocol) 构成，而消息头包括多个属性，实体内容（数据体）则可以被认为是附加在请求之后的文本或二进制文件，只有请求方式为 post 的时候，实体内容才会有数据（即请求参数）。

请求头和请求体之间用什么**分割**，当时回答是两个回车换行，后来查了一下原来是一个**回车换行**。





14. 假如服务端知道 客户端 IP (假设全局 IP) 问 服务端能否向 客户端发消息。

答案是不能，客户端接受消息是通过从服务器拉数据的形式来获取数据而不是服务器向客户端取数据。客户端几乎也是不会监听端口接受服务器发来的消息。

15. 计算机相互之间的通信

互联网的关键技术就是 TCP/IP 协议。两台计算机之间的通信是通过 TCP/IP 协议在因特网上进行的。实际上这个是两个协议：

TCP : Transmission Control Protocol 传输控制协议和 IP: Internet Protocol 网际协议。

IP: 计算机之间的通信

IP 协议是计算机用来相互识别的通信的一种机制，每台计算机都有一个 IP.用来在 internet 上标识这台计算机。 IP 负责在因特网上发送和接收数据包。通过 IP，消息（或者其他数据）被分割为小的独立的包，并通过因特网在计算机之间传送。IP 负责将每个包路由至它的目的地。

IP 协议仅仅是允许计算机相互发消息，但它并不检查消息是否以发送的次序到达而且没有损坏（只检查关键的头数据）。为了提供消息检验功能，直接在 IP 协议上设计了传输控制协议 TCP.

TCP：应用程序之间的通信

TCP 确保数据包以正确的次序到达，并且尝试确认数据包的内容没有改变。TCP 在 IP 地址之

上引端口（port），它允许计算机通过网络提供各种服务。一些端口号为不同的服务保留，而且这些端口号是众所周知。

服务或者守护进程：在提供服务的机器上，有程序监听特定端口上的通信流。例如大多数电子邮件通信流出现在端口 25 上，用于 www 的 HTTP 通信流出现在 80 端口上。

当应用程序希望通过 TCP 与另一个应用程序通信时，它会发送一个通信请求。这个请求必须被送到一个确切的地址。在双方“握手”之后，TCP 将在两个应用程序之间建立一个全双工 (full-duplex) 的通信，占用两个计算机之间整个的通信线路。TCP 用于从应用程序到网络的数据传输控制。TCP 负责在数据传送之前将它们分割为 IP 包，然后在它们到达的时候将它们重组。

TCP/IP 就是 TCP 和 IP 两个协议在一起协同工作，有上下层次的关系。

TCP 负责应用软件（比如你的浏览器）和网络软件之间的通信。IP 负责计算机之间的通信。

TCP 负责将数据分割并装入 IP 包，IP 负责将包发送至接受者，传输过程要经 IP 路由器负责根据通信量、网络中的错误或者其他参数来进行正确地寻址，然后在它们到达的时候重新组合它们。

16. http 的工作过程

一次 HTTP 操作称为一个事务，其工作整个过程如下：

1) 地址解析，

如用客户端浏览器请求这个页面：<http://localhost.com:8080/index.htm>

从中分解出协议名、主机名、端口、对象路径等部分，对于我们的这个地址，解析得到的结果如下：

协议名：http

主机名：localhost.com

端口：8080

对象路径：/index.htm

在这一步，需要域名系统 DNS 解析域名 localhost.com, 得主机的 IP 地址。

2) 封装 HTTP 请求数据包

把以上部分结合本机自己的信息，封装成一个 HTTP 请求数据包

3) 封装成 TCP 包，建立 TCP 连接（TCP 的三次握手）

在 HTTP 工作开始之前，客户机（Web 浏览器）首先要通过网络与服务器建立连接，该连接是通过 TCP 来完成的，该协议与 IP 协议共同构建 Internet，即著名的 TCP/IP 协议族，因此 Internet 又被称作是 TCP/IP 网络。HTTP 是比 TCP 更高层次的应用层协议，根据规则，只有低层协议建立之后才能，才能进行更层协议的连接，因此，首先要建立 TCP 连接，一般 TCP 连接的端口号是 80。这里是 8080 端口

4) 客户机发送请求命令

建立连接后，客户机发送一个请求给服务器，请求方式的格式为：统一资源标识符（URL）、协议版本号，后边是 MIME 信息包括请求修饰符、客户机信息和可内容。

5) 服务器响应

服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是 MIME 信息包括服务器信息、实体信息和可能的内容。

实体消息是服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以 Content-Type 应答头信息所描述的格式发送用户所请求的实际数据

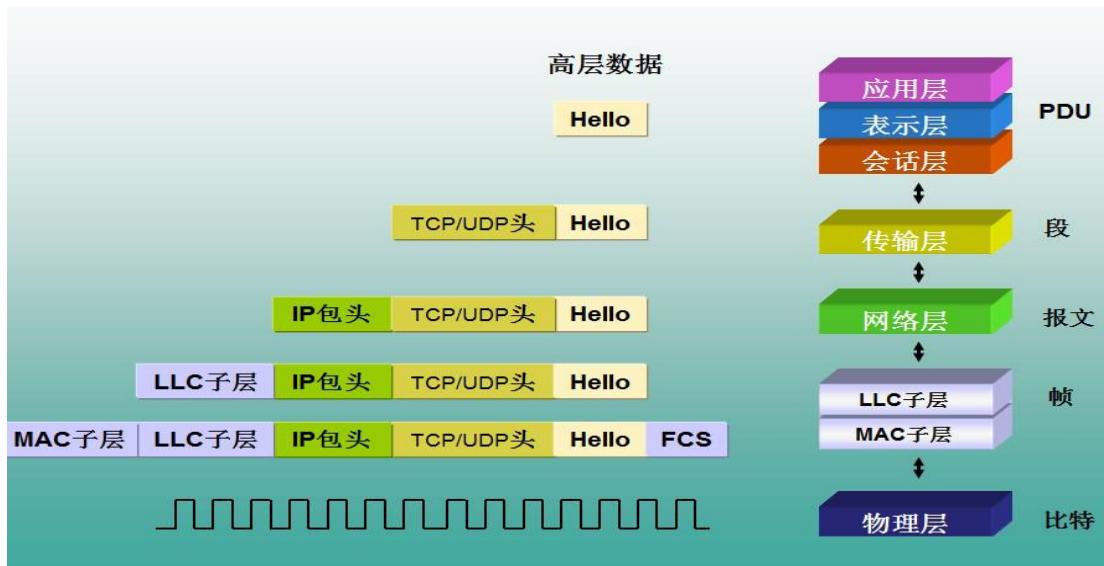
6) 服务器关闭 TCP 连接

一般情况下，一旦 Web 服务器向浏览器发送了请求数据，它就要关闭 TCP 连接，然后如果浏览器或者服务器在其头信息加入了这行代码

Connection:keep-alive

TCP 连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

17. http 协议栈中各层数据流



18. 如果有万级别的 session，怎么提高效率？

专门的 session 服务器(redis 或则 memcache 实现)，这里就有 session 在分布式环境下的问题，最后引出一致性 hash 问题。

19. session 是存储在什么地方，以什么形式存储的？

session 变量保存在[网页服务器](#)中，你不能直接修改，当然，调用程序中的 `setAttribute()`方法当然可以了。`cookie` 存储的可不是具体的数据，要不岂不是太不安全了，谁都可以修改 `session` 变量了，网站也毫无安全性可言。实际，在 `cookie` 中，存储的是一个 `sessionId`，它标示了一个服务器中的 `session` 变量，通过这种方式，服务器就知道你到底是那个 `session` 了。顺便说一句，如果客户端不支持 `cookie`,`session` 也是可以实现的，在服务器端通过 `urlEncoder`，可以实现 `sessionId` 的传递。所以，记住[客户端只存储 session 标识](#)，实际内容在网页服务器中。以键值对的方式存储的

20. TCP IP 协议栈

(1) 网络接口层

TCP/IP 协议模型的基层，负责数据帧的发送和接收。对应 OSI 模型中的物理层和数据链路层，是 TCP/IP 的最底层，不过通常在描述 TCP/IP 模型时还是会划分具体为物理层(PHY)和数据链路层(MAC)。

(2) 网络层

通过互联协议将数据包封装成互联网数据包，并运行必要的路由算法。这里有 4 种互联协议。

- (a)网际协议 IP：负责在主机和网络之间的路径寻址和数据包路由。
- (b)地址解析协议 ARP：获得同一物理网络中的主机硬件地址。
- (c)网际控制消息协议 ICMP：发送消息，并报告有关数据包的传送错误。
- (d)互联组管理协议 IGMP：用来实现本地多路广播路由器报告。

(3) 传输层 提供可靠传输 端到端

传输协议在主机之间提供通信会话。传输协议的选择根据数据传输方式而定。主要有以下 2 种传输协议：

- (a)传输控制协议 TCP：为应用程序提供可靠的通信连接，适用于要求得到响应的应用程序。
- (b)用户数据包协议 UDP：提供无连接通信，且不对传输包进行可靠性确认。

(4) 应用层

应用程序通过这一层访问网络，主要包括常见的 FTP、HTTP、DNS 和 TELNET 协议。

TCP/IP 协议模型对数据的封装



21. 哪些协议是基于 TCP，哪些协议是基于 UDP 的啊

TCP: HTTP,FTP,SMTP,TELNET,POP3,Finger>NNTP,IMAP4,

UDP: BOOTP,DHCP,NTP,TFTP,SNMP
DNS 可以基于 TCP, 也可以基于 UDP~~.

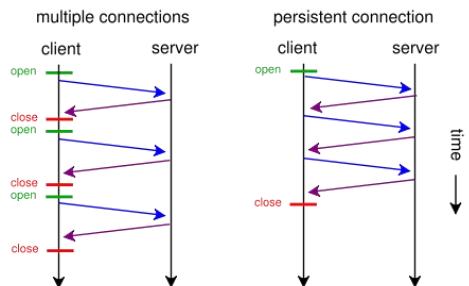
22. 浏览器缓存

浏览器缓存就是当你打开一个网页，浏览器会自动下载副本到你电脑上，就相当于自动帮你另存为网页到某个地方而已。一般 html，后者 request 是 get 请求，而 post 一般不缓存。当然客户端缓存是否需要是可以在服务端代码上控制的。那就是响应头。响应头告诉缓存器不要保留缓存，缓存器就不会缓存相应内容；如果请求信息是需要认证或者安全加密的，相应内容也不会被缓存；缓存控制头 Cache-Control public 什么的参数

23. 服务器是怎么判断用户已经登录，比如 A 登录了，去访问 B 的博客

24. http 长连接 启用 Keep-Alive 的优点

http 1.0 中默认是关闭的；http 1.1 中默认启用 Keep-Alive，如果加入"Connection: close "，才关闭。启用 Keep-Alive 模式肯定更高效，性能更高。因为避免了建立/释放连接的开销。



两种方式判断请求完了

① 使用消息首部字段 Content-Length

Content-Length 表示实体内容长度，客户端（服务器）可以根据这个值来判断数据是否接收完成。

② 使用消息首部字段 Transfer-Encoding

如果是动态页面等时，服务器是不可能预先知道内容大小，这时就可以使用 Transfer-Encoding: chunk 模式来传输数据了。即如果要一边产生数据，一边发给客户端，服务器就需要使用"Transfer-Encoding: chunked"这样的方式来代替 Content-Length。

chunk 编码将数据分成一块一块的发生。Chunked 编码将使用若干个 Chunk 串联而成，由一个标明长度为 0 的 chunk 标示结束。每个 Chunk 分为头部和正文两部分，头部内容指定正文的字符总数（十六进制的数字）和数量单位（一般不写），正文部分就是指定长度的实际内容，两部分之间用回车换行(CRLF)隔开。在最后一个长度为 0 的 Chunk 中的内容是称为 footer 的内容，是一些附加的 Header 信息（通常可以直接忽略）。

TCP 的 keep alive 是检查当前 TCP 连接是否活着；HTTP 的 Keep-alive 是要让一个 TCP 连接活久点。

讲到这里可以提到 **Http** 的流水线技术，在一个 **TCP** 连接内，多个 **HTTP** 请求可以并行，下一个 **HTTP** 请求在上一个 **HTTP** 请求的应答完成之前就发起。

25 HTTP 协议缓存机制

缓存分为服务端侧（server side，比如 Nginx、Apache）和客户端侧（client side，比如 web browser）。

服务端缓存又分为 **代理服务器缓存** 和 **反向代理服务器缓存**（也叫网关缓存，比如 Nginx 反向代理、Squid 等），其实广泛使用的 **CDN** 也是一种服务端缓存，目的都是让用户的请求走“捷径”，并且都是缓存图片、文件等静态资源。

客户端侧缓存一般指的是**浏览器缓存**，目的就是加速各种静态资源的访问，想想现在的大型网站，随便一个页面都是一两百个请求，每天 pv 都是亿级别，如果没有缓存，用户体验会急剧下降、同时服务器压力和网络带宽都面临严重的考验。

浏览器缓存控制机制有两种：HTML Meta 标签 vs. HTTP 头信息

```
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
```

Expires 策略：Expires 是 Web 服务器响应消息头字段，在响应 http 请求时告诉浏览器在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。不过 Expires 是 HTTP 1.0 的东西，现在默认浏览器均默认使用 HTTP 1.1，所以它的作用基本忽略。Expires 的一个缺点就是，返回的到期时间是服务器端的时间，这样存在一个问题，如果客户端的时间与服务器的时间相差很大（比如时钟不同步，或者跨时区），那么误差就很大，所以在 HTTP 1.1 版开始，使用 Cache-Control: max-age=秒替代。

Cache-control 策略（重点关注）：Cache-Control 与 Expires 的作用一致，都是指明当前资源的有效期，控制浏览器是否直接从浏览器缓存取数据还是重新发请求到服务器取数据。只不过 Cache-Control 的选择更多，设置更细致，如果同时设置的话，其优先级高于 Expires。

<https://my.oschina.net/leejun2005/blog/369148>

26 tcp/ip 和 http 的联系与区别

术语 **TCP/IP** 代表传输控制协议/网际协议，指的是一系列协议。“IP”代表网际协议，TCP 和 UDP 使用该协议从一个网络传送数据包到另一个网络。把 **IP** 想像成一种高速公路，它允许其它协议在上面行驶并找到到其它电脑的出口。**TCP 和 UDP** 是高速公路上的“卡车”，它们携带的货物就是像 **HTTP**，文件传输协议 **FTP** 这样的协议等。

27 TCP 协议的流量控制、拥塞控制和差错控制

<http://blog.csdn.net/tspangle/article/details/45072919>

流量控制 **滑动窗口** 拥塞控制 **慢启动**（指数增大），**拥塞避免**（加法增大），**拥塞检测**（除 2 减少，或叫做乘法减少）和差错控制 **校验和，确认，超时重传**（两种情况）：重传定时器时间到，或者 发端收到重复的三个 ACK（快重传）

28 加密和签名方案

1 我要通过支付宝转账给你，我输入你的 id，申请转账，被黑客拦截，把我的请求报文改成了他的 id，我就转账给他了。安全网络通信过程中，需要**防止报文被篡改**

2 还是我用支付宝在商城买东西，商城肯定要给反馈，也就是如果有黑客黑了商城给了假的反馈，这种也是很危险。安全网络通信过程中，**需要客户端和服务端双方确认对方的身份，即交易完成后，不可抵赖**

方案一 对称加密签名机制。举例：MD5 加密签名，签名串=md5(原文&密钥)。最终的报文=**原文&签名串**。如果黑客截取报文，并篡改原文，那么服务端进行验签的时候，将不会通过。因为原文变化了，算出的签名串会改变，那么黑客需要重新计算出签名串，只要知道签名算法（包含加密算法），原文，密钥，前 2 个肯定是会暴露的，无法保密，而客户端是 app，密钥也是暴露的，所以签名串会被重新计算出来，因此黑客将成功篡改转账报文。

方案二 对称加密签名，动态密钥。签名算法（包含加密算法），原文，密钥三者只要保证其中一个不被黑客截取，将无法算出签名串，也就无法篡改报文。那么我们可以动态生成签名的密钥，并用 rsa 公钥对其进行加密（此处 rsa 私钥在服务端，不会泄密，因为签名密钥不会被解密），然后传至服务端。

方案三 报文加密（对称加非对称）1.对称加密：3des。签名串=md5(原文&密钥 1)。最终报文=3des 密钥 2&签名串

方案四 rsa 签名+https，报文加密是必须的，那么我们用 https 加密，其原理同非对称加密+对称加密，场景一方案：客户端产生一对公私钥 pubKey_c, priKey_c；服务端产生一对公私钥 pubKey_s, priKey_s。客户端与服务端置换公钥。最终持有情况如下：

客户端：pubkey_s, priKey_c

服务端：pubKey_c, priKey_s

客户端发送报文：

签名串=rsapriKey_c

最终报文=https(报文原文+签名串)；

28 怎么理解 http 无状态协议

4 Java 基础

1 Object 类有哪些方法 hashCode equals 有哪些关系

protected Object clone()创建并返回此对象的一个副本。

boolean equals(Object obj)指示其他某个对象是否与此对象“相等”。

protected void finalize()当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

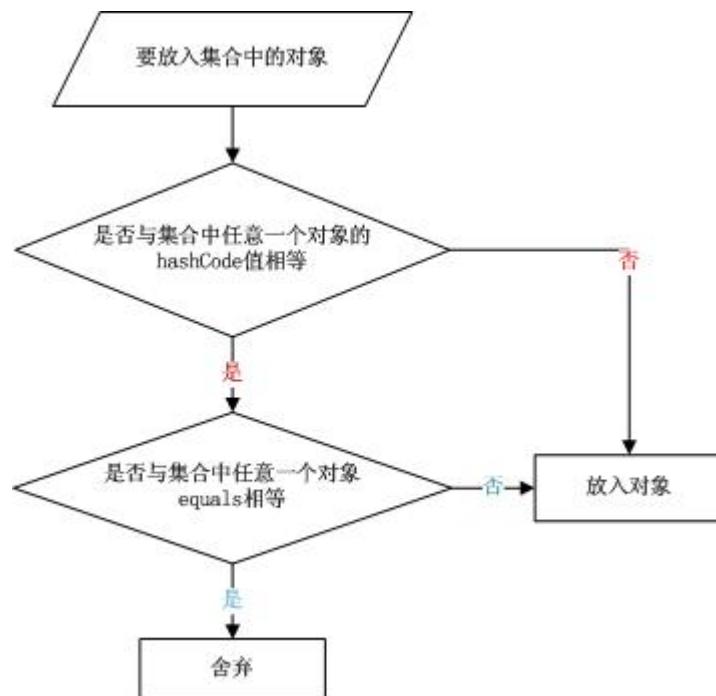
Class<?> getClass()返回此 Object 的运行时类。

int hashCode()返回该对象的哈希码值。

`void notify()` 唤醒在此对象监视器上等待的单个线程。
`void notifyAll()` 唤醒在此对象监视器上等待的所有线程。
`String toString()` 返回该对象的字符串表示。
`void wait()` 在其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法前，导致当前线程等待。
`void wait(long timeout)` 在其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法，或者超过指定的时间量前，导致当前线程等待。
`void wait(long timeout, int nanos)` 在其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

hashCode 与 equals 关系

- 1、`equals` 方法用于比较对象的内容是否相等（覆盖以后）
- 2、`hashcode` 方法只有在集合中用到
- 3、当覆盖了 `equals` 方法时，比较对象是否相等将通过覆盖后的 `equals` 方法进行比较（判断对象的内容是否相等）。
- 4、将对象放入到集合中时，首先判断要放入对象的 `hashcode` 值与集合中的任意一个元素的 `hashcode` 值是否相等，如果不相等直接将该对象放入集合中。如果 `hashcode` 值相等，然后再通过 `equals` 方法判断要放入对象与集合中的任意一个对象是否相等，如果 `equals` 判断不相等，直接将该元素放入到集合中，否则不放入。



2 事务的隔离级别

声明式事务：（注解， XML）

编程式事务：（代码中嵌入）

隔离级别：

- 1、`Serializable`: 最严格的级别，事务串行执行，资源消耗最大；

2、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。

3、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。

4、Read Uncommitted：保证了读取过程中不会读取到非法数据。

隔离级别在于处理多事务的并发问题。我们知道并行可以提高数据库的吞吐量和效率，但是并不是所有的并发事务都可以并发运行，这需要查看数据库教材的可串行化条件判断了。

事务的隔离级别的话，和数据库硬件相关

3 java 序列化的过程

1、什么是序列化和反序列化

Serialization（序列化）是一种将对象以一连串的字节描述的过程；反序列化 deserialization 是一种将这些字节重建成一个对象的过程。

2、什么情况下需要序列化

- a) 当你想把的内存中的对象保存到一个文件中或者数据库中时候；
- b) 当你想用套接字在网络上传送对象的时候；
- c) 当你想通过 RMI 传输对象的时候；

3、如何实现序列化

将需要序列化的类实现 Serializable 接口就可以了，Serializable 接口中没有任何方法，可以理解为一个标记，即表明这个类可以序列化。

4、序列化和反序列化例子

如果我们想要序列化一个对象，首先要创建某些 OutputStream(如 FileOutputStream、ByteArrayOutputStream 等)，然后将这些 OutputStream 封装在一个 ObjectOutputStream 中。这时候，只需要调用 writeObject()方法就可以将对象序列化，并将其发送给 OutputStream(记住：对象的序列化是基于字节的，不能使用 Reader 和 Writer 等基于字符的层次结构)。而反序列的过程（即将一个序列还原成为一个对象），需要将一个 InputStream(如 FileInputStream、ByteArrayInputStream 等)封装在 ObjectInputStream 内，然后调用 readObject()即可。会在此项目的工作空间生成一个 my.out 文件。序列化后的内容稍后补齐，先看反序列化后输出如下：

```
name=SheepMu
```

```
age=24
```

序列化会忽略静态变量，即序列化不保存静态变量的状态。静态成员属于类级别的，所以不能序列化。即 序列化的是对象的状态不是类的状态。这里的不能序列化的意思，是序列化信息中不包含这个静态成员域。最上面添加了 static 后之所以还是输出 24 是因为该值是 JVM 加载该类时分配的值。

总结：

- a) 当一个父类实现序列化，子类自动实现序列化，不需要显式实现 Serializable 接口；
- b) 当一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化；

c) static,transient 后的变量不能被序列化;

什么时候使用序列化:

一: 对象序列化可以实现分布式对象。主要应用例如: RMI 要利用对象序列化运行远程主机上的服务, 就像在本地机上运行对象时一样。

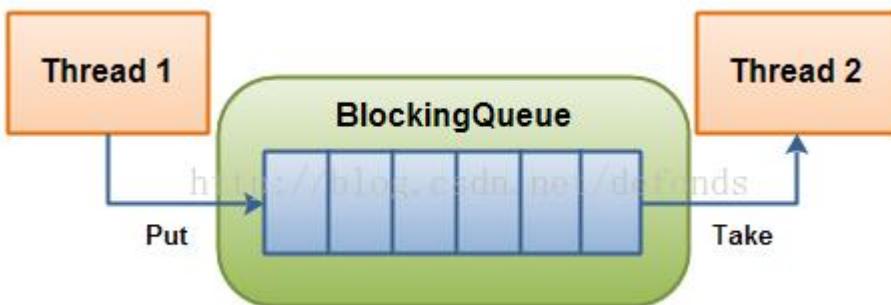
二: java 对象序列化不仅保留一个对象的数据, 而且递归保存对象引用的每个对象的数据。可以将整个对象层次写入字节流中, 可以保存在文件中或在网络连接上传递。利用对象序列化可以进行对象的"深复制", 即复制对象本身及引用的对象本身。序列化一个对象可能得到整个对象序列。

4 Java concurrent 包 BlockingQueue

Java 5 添加了一个新的包到 Java 平台, `java.util.concurrent` 包。这个包包含有一系列能够让 Java 的并发编程变得更加简单轻松的类。在这个包被添加以前, 你需要自己去动手实现自己的相关工具类。

阻塞队列 BlockingQueue

一个线程往里边放, 另外一个线程从里边取的一个 **BlockingQueue**。



一个线程将会持续生产新对象并将其插入到队列之中, 直到队列达到它所能容纳的临界点。也就是说, 它是有限的。如果该阻塞队列到达了其临界点, 负责生产的线程将会在往里边插入新对象时发生阻塞。它会一直处于阻塞之中, 直到负责消费的线程从队列中拿走一个对象。负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话, 这个消费线程将会处于阻塞之中, 直到一个生产线程把一个对象丢进队列。

数组阻塞队列 ArrayBlockingQueue。ArrayBlockingQueue 类实现了 `BlockingQueue` 接口。ArrayBlockingQueue 是一个有界的阻塞队列, 其内部实现是将对象放到一个数组里。

延迟队列 DelayQueue 链阻塞队列 LinkedBlockingQueue 具有优先级的阻塞队列

PriorityBlockingQueue 同步队列 SynchronousQueue 阻塞双端队列 BlockingDeque

并发 Map(映射) ConcurrentHashMap

`ConcurrentHashMap` 和 `java.util.HashTable` 类很相似, 但 `ConcurrentHashMap` 能够提供比 `HashTable` 更好的并发性能。在你从中读取对象的时候 `ConcurrentHashMap` 并不会把整个 `Map` 锁住。此外, 在你向其中写入对象的时候, `ConcurrentHashMap` 也不会锁住整个 `Map`。它的内部只是把 `Map` 中正在被写入的部分进行锁定。

另外一个不同点是, 在被遍历的时候, 即使是 `ConcurrentHashMap` 被改动, 它也不会抛 `ConcurrentModificationException`。尽管 `Iterator` 的设计不是为多个线程的同时使用。

- 1.Callable<V>
 - 2.Semaphore
 - 3.ReentrantLock 与 Condition
 - 4.BlockingQueue
 - 6.CountDownLatch
 - 7.CyclicBarrier
- 闭锁和栅栏见多线程的东西同步机制的 5

5 String StringBuffer StringBuilder 区别，平时用到了哪些

1. 在执行速度方面的比较: **StringBuilder > StringBuffer**
2. StringBuffer 与 StringBuilder, 他们是字符串变量, 是可改变的对象, 每当我们用它们对字符串做操作时, 实际上是在一个对象上操作的, 不像 String 一样创建一些对象进行操作, 所以速度就快了。
3. **StringBuilder: 线程非安全的**
StringBuffer: 线程安全的

当我们在字符串缓冲区被多个线程使用时, JVM 不能保证 StringBuilder 的操作是安全的, 虽然他的速度最快, 但是可以保证 StringBuffer 是可以正确操作的。当然大多数情况下就是我们是在单线程下进行的操作, 所以大多数情况下是建议用 StringBuilder 而不用 StringBuffer 的, 就是速度的原因。

对于三者使用的总结:

1. 如果要操作少量的数据用 = String
2. 单线程操作字符串缓冲区 下操作大量数据 = StringBuilder
3. 多线程操作字符串缓冲区 下操作大量数据 = StringBuffer

• 4. String、StringBuffer、StringBuilder 区别 ? ↪

- 1) 在 JDK 源码中, String 类是被修饰成 final 的, 所以 String 类是不可变的。我们对于 String 对象的操作都会返回一个新的 String 对象。所以对 String 进行字符串的拼接的时候会创建大量的 String 实例对象, 这是一个非常耗时的操作, 所以 String 直接拼接字符串效率很低。 ↪
- 2) StringBuffer 和 StringBuilder 是可变的。他们两个都是继承于 AbstractStringBuilder, 内部是基于 char[] 字符数组实现的, 其对于字符串的拼接等操作是基于数组, 所以效率很高。此外 StringBuilder 是非线程安全的, StringBuffer 是线程安全的。StringBuffer 内部的所有方法都是使用 synchronized 锁实现的同步方法, 所以在多线程环境下使用 StringBuffer。StringBuilder 的方法没有使用任何同步机制, 所以是非线程安全的。注意在单线程环境下 StringBuilder 效率比 StringBuffer 高, 因为没有加锁释放锁的时间。 ↪

6 final 关键字 & 参数传递特点

final

1.final 类表示该类不可继承,

2.final 方法表示该方法不可被覆盖

3.final 域表示该字段一被初始化就不能再改变(必须确保在构造器执行之后 final 域均被设置,且不可改变)

final 用于基本类型和不可变类型,对象类型不可变的只是引用没有意义

4.final 参数 表示在作用域里只能读取不能赋值该 final 变量

参数传递

Java 传递是"值传递":

1.基础类型+布尔 是把变量 copy 了一份传给函数,对原变量无影响

2.对象类型 是把改引用 copy 了一份(新旧引用指向同一对象),

a.在函数内部对新引用的赋值操作不会影响原引用的指向以及指向的对象

b.在函数内部对新引用进行对象改变属性操作,不会影响原引用的指向,但是会影响原引用指向的对象

基础类型和对象类型传递的都是值,只是一个值本身,一个是引用

一个 final 的 ArrayList 对象,能对其增加或删除对象么

但是如果你声明为 final

```
final ArrayList lista = new ArrayList();
```

```
ArrayList listb = new ArrayList();
```

lista = listb; 语法不合法, 此时 lista 不能指向其他区域了

像 add, delete 方法照常使用

7 Tomcat, Apache, JBoss 的区别?

1、Apache 是 Http 服务器, Tomcat 是 web 服务器, JBoss 是应用服务器。

2、Apache 解析静态的 html 文件; Tomcat 可解析 jsp 动态页面、也可充当 servlet 容器。

8 字节 kb MB

1GB=1024MB=1024*1024KB=1024*1024*1024B

1MB=1024KB=1024*1024B

1KB=1024B

1Byte=1B=8bit(八位二进制数)

字节 (Byte):通常将可表示常用英文字符 8 位二进制称为一字节。

一个英文字母 (不分大小写) 占一个字节的空间

一个中文汉字占两个字节的空间.

符号: 英文标点占一个字节, 中文标点占两个字节.

9 AtomicInteger 底层是怎么实现的

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。两个问题:

(1) CAS 算法仍然可能会出现冲突, 例如 A、B 两个线程, A 已经进入写内存但未完成, 此

时 A 读取到的副本且读取成功，AB 两个线程同时进入写内存操作，必然会造成冲突。 CAS 算法本质并非完全无锁，而是把获得锁和释放锁推迟至 CPU 原语实现，相当于尽可能的缩小了锁的范围；直接互斥地实现系统状态的改变，它的使用基本思想是 copy-on-write——在修改完对象的副本之后再用 c

10 try 里面有 return finally 会执行吗

```
public static int i=0;

public static int testtest(){
    try {
        i=10;
        System.out.println("try");
        return i;
    }finally {
        i=100;
        System.out.println("finally: " + i);
    }
}

@org.junit.Test
public void test1(){
    System.out.println("test:"+ testtest());
    System.out.println(i);
}
```

```
/Library/Java/JavaVirtualMachines/jc
try
finally: 100
test:10
100
```

11 equals() 跟 == 有什么区别

==操作比较的是两个变量的值是否相等,对于引用型变量表示的是两个变量在堆中存储的地址是否相同,即栈中的内容是否相同。equals 操作表示的两个变量是否是对同一个对象的引用,即堆中的内容是否相同。

==比较的是 2 个对象的地址,而 equals 比较的是 2 个对象的内容。显然,当 equals 为 true 时, ==不一定为 true。

12 websocket

```
var id = messages + 1;
<script type="text/javascript">
    var webSocket = new WebSocket(
        'ws://localhost:8080/WebSocketTest/websocket');

    webSocket.onerror = function(event) {
        onError(event)
    };

    webSocket.onopen = function(event) {
        onOpen(event)
    };

    webSocket.onmessage = function(event) {
        onMessage(event)
    };

    function onMessage(event) {
        document.getElementById('messages').innerHTML += '<br />' +
            event.data;
    }

    function onOpen(event) {
        document.getElementById('messages').innerHTML = 'Connection established';
    }

    function onError(event) {
        alert(event.data);
    }

    function start() {
        webSocket.send('hello');
        return false;
    }
</script>
```

```

0
1 @ServerEndpoint("/websocket")
2 public class WebSocketTest {
3
4     @OnMessage
5     public void onMessage(String message, Session session) throws IOException, InterruptedException {
6
7         // Print the client message for testing purposes
8         System.out.println("Received: " + message);
9
10        // Send the first message to the client
11        session.getBasicRemote().sendText("This is the first server message");
12
13        // Send 3 messages to the client every 5 seconds
14        int sentMessages = 0;
15        while (sentMessages < 3) {
16            Thread.sleep(5000);
17            session.getBasicRemote().sendText("This is an intermediate server message. Count: " + sentMessages);
18            sentMessages++;
19        }
20
21        // Send a final message to the client
22        session.getBasicRemote().sendText("This is the last server message");
23    }
24
25    @OnOpen
26    public void onOpen() {
27        System.out.println("Client connected");
28    }
29
30    @OnClose
31    public void onClose() {
32        System.out.println("Connection closed");
33    }
34}

```

13 静态内部类和非静态内部类的区别？

- 1) java 允许我们在一个类里面定义静态内部类（nested class），把 nested class 封闭起来的类叫外部类。在 java 中，我们不能用 static 修饰顶级类（top level class）。只有内部类可以为 static。
- 2) 静态内部类和非静态内部类之间到底有什么不同呢？
 - (1) 静态内部类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。
 - (2) 静态内部类不能访问外部类的非静态成员，他只能访问外部类的静态成员。
非静态内部类能够访问外部类的静态和非静态成员。
 - (3) 一个非静态内部类不能脱离外部类实体被创建，必须通过外部类的引用创建实例。
一个非静态内部类可以访问外部类的数据和方法，因为他就在外部类里面。

14 Java 的反射作用原理与作用：

- 1) 原理：在程序运行的时候能够获取自身的的 Class 信息，比如属性和方法，且能够调用它的任意的方法和属性，这就为 Java 提供了动态的特性。
- 2) Java 反射的作用：
 1. 在运行时判断任意一个对象所属的类；
 2. 在运行时构造任意一个类的对象；

3. 在运行时判断任意一个类所具有的成员变量和方法;
4. 在运行时调用任意一个对象的方法;
5. 生成动态代理。

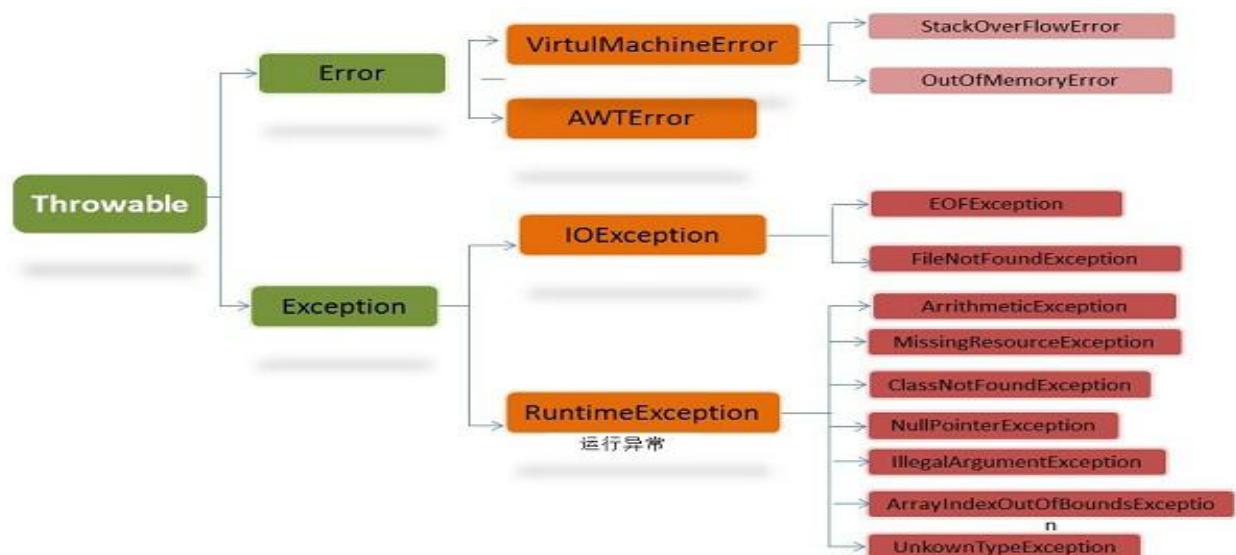
15 抽象类与接口的区别

- 1) 抽象类与接口都不能实例化;
- 2) 抽象类要被子类继承，接口要被类实现;
- 3) 接口只能声明方法，抽象类既可以声明方法也可以实现方法;
- 4) 接口里面的变量只能是公共静态常量，抽象类里面可以定义变量;
- 5) 抽象类必须被子类实现所有抽象方法，不然子类也是抽象类;
- 6) 接口可以多继承接口。但类只能单继承。

16 Static 关键字的作用

静态变量、静态方法、静态代码快、内部静态类。

17 Java 中的异常



18 正则表达式贪婪与非贪婪模式

贪婪匹配:正则表达式一般趋向于最大长度匹配，结果就是匹配到: abcaxc(ab*c)。

非贪婪匹配: 就是匹配到结果就好，就少的匹配字符。结果就是匹配到: abc(ab*c)。

默认是贪婪模式；在量词后面直接加上一个问号？就是非贪婪模式。

量词: {m,n}: m 到 n 个; *: 任意多个; +: 一个到多个; ?: 0 或一个

19 Java 静态分派与动态分派

我的理解就是动态的实现多态。父类引用指向子类对象的时候

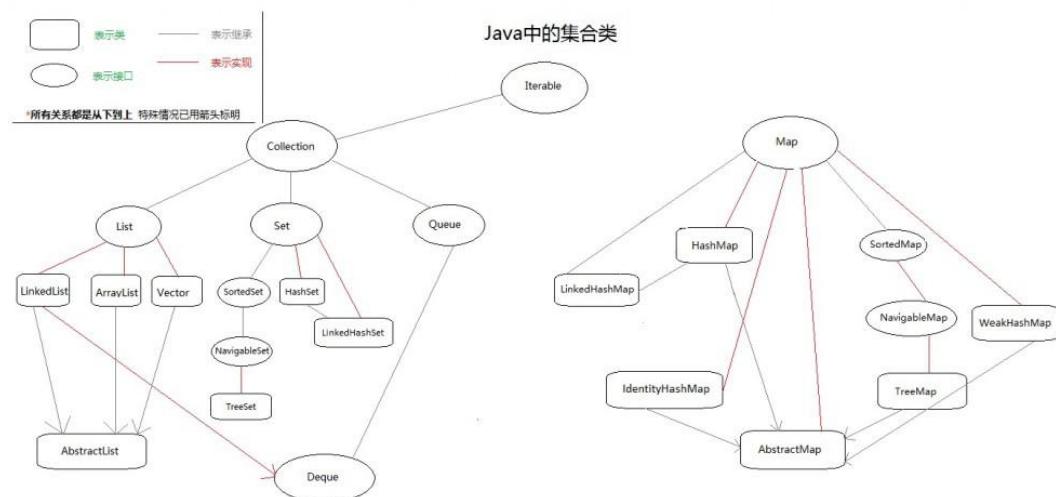
Human man=new Man();

我们把“Human”称为变量的**静态类型**，后面的“Man”称为变量的**实际类型**，静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，**变量**本身的静态类型不会被改变，并且最终的静态类型在编译器可知；而实际类型变化的结果在运行期才确定，编译器在编译期并不知道一个对象的实际类型是什么。

编译器在**重载**时是通过**参数的静态类型**而不是实际类型作为判定的依据。并且静态类型在编译期可知，因此，编译阶段，**Javac** 编译器会根据参数的静态类型决定使用哪个重载版本。所有依赖静态类型来定位方法执行版本的分派动作称为**静态分派**。静态分派的典型应用就是**方法重载**。

<http://blog.csdn.net/sunxianghuang/article/details/52280002>

5 Java 集合类



1 Vector, ArrayList, LinkedList 的区别是什么？

- 1、**Vector**、**ArrayList** 是类似**数组**的形式存储在内存中，**LinkedList** 则以**链表**的形式进行存储。
- 2、**List** 中的元素有序、允许有重复的元素，**Set** 中的元素无序、不允许有重复元素。
- 3、**Vector** 线程同步，**ArrayList**、**LinkedList** 线程不同步。
- 4、**LinkedList** 适合指定位置插入、删除操作，不适合查找；**ArrayList**、**Vector** 适合查找，不适合指定位置的插入、删除操作。
- 5、**ArrayList** 在元素填满容器时会**自动扩充容器大小的 50%**，而 **Vector** 则是 **100%**，因此 **ArrayList** 更节省空间。

2 hashmap 与 hashtable 的区别是什么

Hashtable 继承自 **Dictionary** 而 **HashMap** 继承自 **AbstractMap**

Hashtable 的 put 方法（线程安全的）

注意 1 方法是同步的

注意 2 方法不允许 `value==null`, 直接判断抛异常

注意 3 方法调用了 `key` 的 `hashCode` 方法, 如果 `key==null`, 会抛出空指针异常

HashMap 的 put 方法（线程非安全的）

注意 1 方法是非同步的

注意 2 方法允许 `key==null`

注意 3 方法并没有对 `value` 进行任何调用, 所以允许为 `null`

补充:

Hashtable 有一个 `contains` 方法, 容易引起误会, 所以在 **HashMap** 里面已经去掉了
当然, 2 个类都用 `containsKey` 和 `containsValue` 方法。

HashMap 是 **Hashtable** 的轻量级实现（非线程安全的实现）, 他们都完成了 `Map` 接口,

主要区别在于 **HashMap** 允许空(`null`)键值(`key`), 由于**非线程安全**, 效率上可能高于 **Hashtable**。

继承和实现方式不同

HashMap 继承于 **AbstractMap**, 实现了 **Map**、**Cloneable**、
java.io.Serializable 接口。

Hashtable 继承于 **Dictionary**, 实现了 **Map**、**Cloneable**、
java.io.Serializable 接口。

线程安全不同

Hashtable 的几乎所有函数都是同步的, 即它是线程安全的, 支持多线程。
而 **HashMap** 的函数则是非同步的, 它不是线程安全的。若要在多线程中使用
对 **null** 值的处理不同

HashMap 的 `key`、`value` 都可以为 `null`。

Hashtable 的 `key`、`value` 都不可以为 `null`。

支持的遍历种类不同

HashMap 只支持 **Iterator**(迭代器)遍历。

而 **Hashtable** 支持 **Iterator**(迭代器)和 **Enumeration**(枚举器)两种方式
遍历。

通过 **Iterator** 迭代器遍历时, 遍历的顺序不同

HashMap 是“从前向后”的遍历数组; 再对数组具体某一项对应的链表, 从表
头开始进行遍历。

Hashtable 是“从后往前”的遍历数组；再对数组具体某一项对应的链表，从表头开始进行遍历。

容量的初始值 和 增加方式都不一样

HashMap 默认的容量大小是 **16**；增加容量时，每次将容量变为“原始容量 **x2**”。

Hashtable 默认的容量大小是 **11**；增加容量时，每次将容量变为“原始容量 **x2 + 1**”

添加 **key-value** 时的 **hash** 值算法不同

HashMap 添加元素时，是使用自定义的哈希算法。

Hashtable 没有自定义哈希算法，而直接采用的 **key** 的 **hashCode()**。

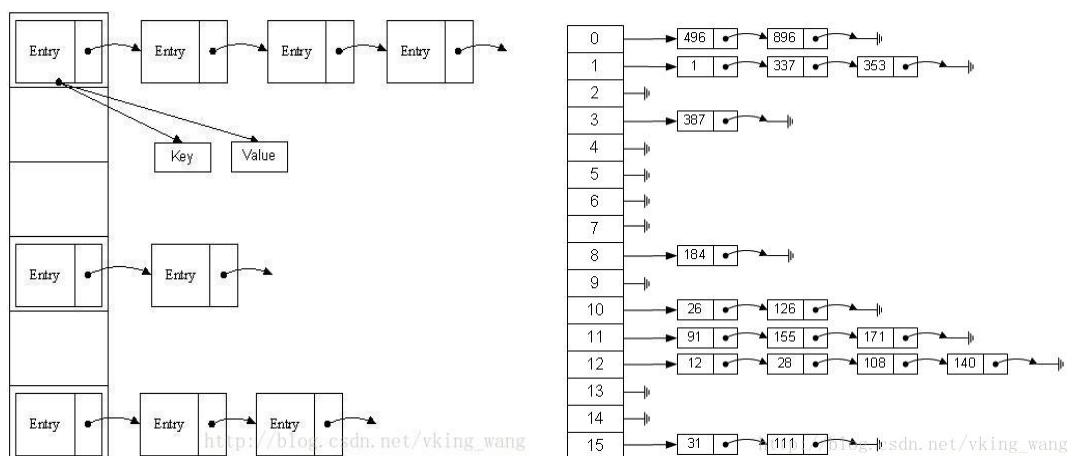
部分 **API** 不同

Hashtable 支持 **contains(Object value)** 方法，而且重写了 **toString()** 方法；

而 **HashMap** 不支持 **contains(Object value)** 方法，没有重写 **toString()** 方法。在 **Entry** 里面写的

3 HashMap 的原理

HashMap 的数据结构就是 **数组加链表**（有图就是拉链法实现）



解决 hash 冲突的办法

1. 开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）
2. 再哈希法
3. 链地址法
4. 建立一个公共溢出区

HashMap 是无序的，有序的是 LinkedHashMap（在内部增加了一个链表，用以存放元素的顺序）和 TreeMap（实现了 SortedMap 接口，这就意味着可以对元素进行排序）

HashMap 的默认的容量是 16，装填因子是 0.75，1.8 之后加了 treeify_threshold = 8 当链表的长度大于 8 时就转红黑树

理论上来讲当然是 O(1)，但是实际上还有很多时间开销的，比如 hash 碰撞，另外 hash 的计算也要耗费 CPU 时间。所以一般我们认为它的时间复杂度是常数级的。

问：这个可以直接去看 HashMap 的源码，“java 怎样哈希一个任意的类的实例呢？”简单地说就是通过类的 equals 和 hashCode 方法。

答：如果很好地实现了 equals 和 hashCode 方法，HashMap 能保证以 O(1) 的复杂度查找我自己的类的实例吗？以及，如果没有重写 equals 和 hashCode 方法，会影响效率吗？

（不考虑深判等）

问：你不写的话，就不能用自定义的类做 key，HashMap 不能保证 O(1) 的复杂度，它是根据 hashCode 计算出一个对象在桶的位置，一般情况下，同一类不同对象能保证 hashCode 值不同就可以了。

答：理想的情况下在较好实现了 hashCode 后是复杂度是 O(1)，理想情况是所有键值对在桶中均匀分布，很显然不太现实吧

4 HashMap 源码

<http://blog.csdn.net/caimengyuan/article/details/61204542> 关于 HashMap 的一些按位与计算的问题

put 方法有返回值，返回的是键值对应关系的值，集合中原来对键值没有指定关系，返回 null，有则返回原来的值。

Hash 是把 32 位的高 16 位和低 16 位都利用起来，进行异或运算

```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    modCount++;
    addEntry(hash, key, value, i);
    return null;
}
```

```

    static int hash(int h) {
        // This function ensures that hashCode's that differ only by
        // constant multiples at each bit position have a bounded
        // number of collisions (approximately 8 at default load factor).
        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
    }

    /**
     * Returns index for hash code h.
     */
    static int indexFor(int h, int length) {
        return h & (length-1);
    }
}

```

所以他的长度要为 16，然后扩容为 16 的 2 的整数倍次方，愿意就在这个 indexFor 这里是求余。求余%这种就有些浪费时间，如果是用位运算求余就很快。

什么时候 ReHash

在介绍 HashMap 的内部实现机制时提到了两个参数，CAPACITY 和 FACTOR，CAPACITY 是 table 数组的容量，FACTOR 则是为了最大程度避免哈希冲突，提高 HashMap 效率而设置的一个影响因子，将其乘以 CAPACITY 就得到了一个阈值 threshold，当 HashMap 的容量达到 threshold 时就需要进行扩容，这个时候就要进行 ReHash 操作了，可以看到下面 addEntry 函数的实现，当 size 达到 threshold 时会调用 resize 函数进行扩容。在扩容的过程中需要进行 ReHash 操作，而这是非常耗时的，在实际中应该尽量避免。

如果是光杆司令的话，会把原来的结点置空然后和新的 newCap-1 做按位与运算得新位置
如果是个红黑树，对红黑树进行分割

如果下面是链表，`e.hash & oldCap` 与 1 比较，也就是第五位为 1 的时候移到后面，如果第五位为 0 的继续放在原位，其实就是一人分一半

```

else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
         oldCap >= DEFAULT_INITIAL_CAPACITY)
    newThr = oldThr << 1; // double threshold
}

void addEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    if (size++ >= threshold)
        resize(2 * table.length);
}

```

```

void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;           扩容因子
    threshold = (int)(newCapacity * loadFactor);
}   新的扩容长度

/**
 * Transfers all entries from current table to newTable.
 */
void transfer(Entry[] newTable) {
    Entry[] src = table;
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}

```

5 TreeMap 的实现 TreeSet、红黑树

TreeMap 的实现就是**红黑树数据结构**，也就说是一棵自平衡的排序二叉树，这样就可以保证当需要快速检索指定节点。

LinkedHashMap 是基于元素进入集合的顺序或者被访问的先后顺序排序，TreeMap 则是基于元素的固有顺序 (由 Comparator 或者 Comparable 确定)。

LinkedHashMap 是根据元素增加或者访问的先后顺序进行排序，而 TreeMap 则根据元素的 Key 进行排序

TreeSet 和 TreeMap 的关系 TreeSet 的 ① 号、② 号构造器的都是新建一 TreeMap

```

public TreeSet() {                                作为实际存储 Set 元素的容器，而另外
    this(new TreeMap<E,Object>());             一个 TreeSet 会使用红黑树
}

```

2 个构造器则分别依赖于 ① 号和 ② 号构造器，由此可见，**TreeSet** 底层实际使用的存储容器就是 **TreeMap**。

Java 实现的红黑树

上面的性质 3 中指定红黑树的每个叶子节点都是空节点，而且非叶子节点都是黑色。但 Java 实现的红黑树将使用 `null` 来代表空节点，因此遍历红黑树时将看不到黑色的叶子节点，反而看到每个叶子节点都是红色的。

- 性质 1：每个节点要么是红色，要么是黑色。
- 性质 2：根节点永远是黑色的。
- 性质 3：所有的叶节点都是空节点（即 `null`），并且是黑色的。
- 性质 4：每个红色节点的两个子节点都是黑色。（从每个叶子到根的路径上不会有两个连续的红色节点）
- 性质 5：从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点。

6 TreeMap 源码

如果是一棵空树，就从 `root` 根节点开始

```
public V put(K key, V value) {
    Entry<K,V> t = root;
    if (t == null) {
        // TBD:
        // 5045147: (coll) Adding null to an empty TreeSet should
        // throw NullPointerException
        //
        // compare(key, key); // type check
        root = new Entry<K,V>(key, value, null);
        size = 1;
        modCount++;
        return null;
    }
}
```

如果 `comparator` 不为空就执行分割比较器和可比较的路径，这个哪里来，构造方法

```
/\
public TreeMap() {           \
    comparator = null;       \
}                           \
public TreeMap(Comparator<? super K> comparator) { \
    this.comparator = comparator; \
}
```

```

        ,
int cmp;
Entry<K,V> parent;
// split comparator and comparable paths
Comparator<? super K> cpr = comparator;
if (cpr != null) {
    do {
        parent = t;
        cmp = cpr.compare(key, t.key);
        if (cmp < 0)
            t = t.left;
        else if (cmp > 0)
            t = t.right;
        else
            return t.setValue(value);
    } while (t != null);
}

```

如果 comparator 为空，就执行下面的，显然 key 是不能为空的

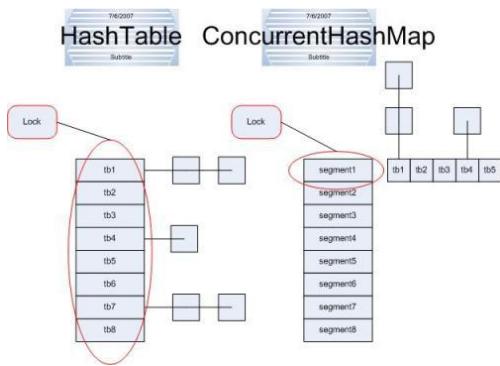
```

else {
    if (key == null)
        throw new NullPointerException();
Comparable<? super K> k = (Comparable<? super K>) key;
do {
    parent = t;
    cmp = k.compareTo(t.key);
    if (cmp < 0)
        t = t.left;
    else if (cmp > 0)
        t = t.right;
    else
        return t.setValue(value);
} while (t != null);
}
Entry<K,V> e = new Entry<K,V>(key, value, parent);
if (cmp < 0)
    parent.left = e;
else
    parent.right = e;
fixAfterInsertion(e);
size++;
modCount++;
return null;

```

7 HashMap 和 ConcurrentHashMap 的区别

- 1) 线程安全的 Map,
- 2) 基于分段锁实现，默认是 16 级分段，这里面的每个 Segment 都相当于一个 HashTable
- 3) 每个 Segment 只允许一个线程写，但是不限制读。



两个 hash 过程，第一次找到所在的桶，并将桶锁定，第二次执行写操作。

而读操作不加锁

1) ConcurrentHashMap 是基于分段锁实现的，具体可以理解成把一个大的 Map 拆分成 N 个小的 HashTable（默认是 16 个），根据 key.hashCode() 来决定把 key 放到哪个 HashTable。在 ConcurrentHashMap 中就是把 Map 分成了 N 个 Segment，put 和 get 的时候，都是先根据 key.hashCode() 算出放在哪个 Segment 中，然后对对应的 Segment 加锁，这样就不会影响其余 Segment 的并发访问。这样效率就提升了 N 倍，默认提升了 16 倍。其实也就是同时允许 16 个线程分别对 16 个 Segment 操作，只有写操作才需要锁住 Segment，读线程基本不受限制。

2) 基本上 ConcurrentHashMap 是 HashMap 和 HashTable 的结合。那么为什么 ConcurrentHashMap 里面的 get() 操作，也就是读操作不用加锁呢，除非读到的值是空的才会加锁重读？

原因是它的 get 方法里将要使用的共享变量都定义成 volatile，如用于统计当前 Segement 大小的 count 字段和用于存储值的 HashEntry 的 value。定义成 volatile 的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在 get 操作里只需要读不需要写共享变量 count 和 value，所以可以不用加锁。之所以不会读到过期的值，是根据 java 内存模型的 happen before 原则，对 volatile 字段的写入操作优先于读操作，即使两个线程同时修改和获取 volatile 变量，get 操作也能拿到最新的值，这是用 volatile 替换锁的经典应用场景。

3) ConcurrentHashMap 不允许 key 或则 value 为空。

Concurrent 包还提供了很多 collection 的实现

- ConcurrentHashMap 同步容器类是 Java 5 增加的一个线程安全的哈希表。对与多线程的操作，介于 HashMap 与 Hashtable 之间。内部采用“锁分段”机制替代 Hashtable 的独占锁。进而提高性能。
- 此包还提供了设计用于多线程上下文中的 Collection 实现： ConcurrentHashMap、ConcurrentSkipListMap、ConcurrentSkipListSet、CopyOnWriteArrayList 和 CopyOnWriteArraySet。当期望许多线程访问一个给定 collection 时，ConcurrentHashMap 通常优于同步的 HashMap，ConcurrentSkipListMap 通常优于同步的 TreeMap。当期望的读数和遍历远远大于列表的更新数时，CopyOnWriteArrayList 优于同步的 ArrayList。

8 ConcurrentHashMap 源码

<http://www.cnblogs.com/yydcut/p/3959815.html>

put

```

    /*
public V put(K key, V value) {
    if (value == null)
        throw new NullPointerException();
    int hash = hash(key.hashCode());
    return segmentFor(hash).put(key, hash, value, false);
}

/**
 * Returns the segment that should be used for key with given hash
 * @param hash the hash code for the key
 * @return the segment
 */
final Segment<K,V> segmentFor(int hash) {
    return segments[(hash >>> segmentShift) & segmentMask];
}

*/
static final class Segment<K,V> extends ReentrantLock implements Serializable {
    /*

```

Segment 继承了 ReentrantLock，表明每个 segment 都可以当做一个锁。（ReentrantLock 前文已经提到，不了解的话就把当做 synchronized 的替代者吧）这样对每个 segment 中的数据需要同步操作的话都是使用每个 segment 容器对象自身的锁来实现。只有对全局需要改变时锁定的是所有的 segment。

```

V put(K key, int hash, V value, boolean onlyIfAbsent) {
    lock();
    try {
        int c = count;
        if (c++ > threshold) // ensure capacity
            rehash();
        HashEntry<K,V>[] tab = table;
        int index = hash & (tab.length - 1);
        HashEntry<K,V> first = tab[index];
        HashEntry<K,V> e = first;
        while (e != null && (e.hash != hash || !key.equals(e.key)))
            e = e.next;

        V oldValue;
        if (e != null) {
            oldValue = e.value;
            if (!onlyIfAbsent)
                e.value = value;
        }
        else {
            oldValue = null;
            ++modCount;
            tab[index] = new HashEntry<K,V>(key, hash, first, value);
            count = c; // write-volatile
        }
    }
}

```

Get

```

    ...
public V get(Object key) {
    int hash = hash(key.hashCode());
    return segmentFor(hash).get(key, hash);
}

V get(Object key, int hash) {
    if (count != 0) { // read-volatile ←
        HashEntry<K,V> e = getFirst(hash);
        while (e != null) {
            if (e.hash == hash && key.equals(e.key)) {
                V v = e.value;
                if (v != null)          如果V为空，那么就可能是另外一个线程创建了但
                                         没来得及设置值，所以去同步获得
                    return v;           readValueUnderLock(e); // recheck
                }
                e = e.next;
            }
        }
    }
    return null;
}

```

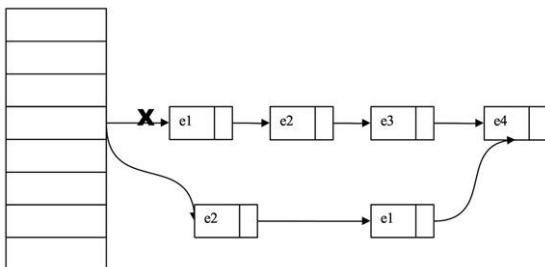
这里总是认为是乐观的，当判断 count 不为 0 之后，乐观的想象不会有线程来增删改，舍弃了一致性。第一步，先判断一下 `count != 0`; count 变量表示 segment 中存在 entry 的个数。如果为 0 就不用找了。count 变量的定义：`transient volatile int count;` 它使用了 volatile 来修改。对 volatile 域的写入操作 happens-before 于每一个后续对同一个域的读写操作。所以，每次判断 count 变量的时候，即使恰好其他线程改变了 segment 也会体现出来。

第二步，获取到要该 key 所在 segment 中的索引地址，如果该地址有相同的 hash 对象，顺着链表一直比较下去找到该 entry。当找到 entry 的时候，先做了一次比较：if(v != null)如果 v 为空，说明另外一个线程还没创建完。

Remove

假设我们的链表元素是：e1->e2->e3->e4 我们要删除 e3 这个 entry，因为 HashEntry 中 next 的不可变（都是 final 的），所以我们无法直接把 e2 的 next 指向 e4，而是将要删除的节点之前的节点复制一份，形成新的链表。

```
/*
 * static final class HashEntry<K,V> {
 *     final K key;
 *     final int hash;
 *     volatile V value;
 *     final HashEntry<K,V> next;
 *
 *     HashEntry(K key, int hash, HashEntry<K,V> next, V value) {
 *         this.key = key;
 *         this.hash = hash;
 *         this.next = next;
 *         this.value = value;
 *     }
 * }
```



这个让我想到了 CopyOnWriteArrayList。CopyOnWrite 容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对 CopyOnWrite 容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以 CopyOnWrite 容器也是一种读写分离的思想，读和写不同的容器。

9 为什么要用 HashTable

根据 key 查 value 比较快，线程安全

10 Collections.sort

```
/*
 * public static <T extends Comparable<? super T>> void sort(List<T>
 * Object[] a = list.toArray();
 * Arrays.sort(a);
 * ListIterator<T> i = list.listIterator();
 * for (int j=0; j<a.length; j++) {
 *     i.next();
 *     i.set((T)a[j]);
 * }
 * }
```

用的是 Arrays.sort，这个是归并排序

```
/*
 * public static void sort(Object[] a) {
 *     Object[] aux = (Object[])a.clone();
 *     mergeSort(aux, a, 0, a.length, 0);
 * }
```

11 JDK1.8 之后 HashMap 引入了红黑树

HashMap 在 JDK 1.8 中新增的操作：桶的树形化 treeifyBin()

当桶中元素个数超过这个值时（默认为 8，这个值必须为 8，要不然频繁转换效率也不高），需要使用红黑树节点替换链表节。

11 ArrayList 源码

ArrayList 初始化，有两个构造方法，如果给参数就是初始化的能力，如果没给默认是 10

```
/*
public ArrayList(int initialCapacity) {
super();
if (initialCapacity < 0)
    throw new IllegalArgumentException("Illegal Capacity: "+
                                       initialCapacity);
this.elementData = new Object[initialCapacity];
}

/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {
this(10);
}

ArrayList add
/*
public boolean add(E e) {
ensureCapacity(size + 1); // Increments modCount!!
elementData[size++] = e;
return true;
}

public void add(int index, E element) {
if (index > size || index < 0)
    throw new IndexOutOfBoundsException(
        "Index: "+index+", Size: "+size);

ensureCapacity(size+1); // Increments modCount!!
System.arraycopy(elementData, index, elementData, index + 1,
                 size - index);
elementData[index] = element;
size++;
}
```

在确定大小扩容的时候是 $old * 3$ 的值再除以 $2 + 1$

10->16->25->38->58->88->...

```
public void ensureCapacity(int minCapacity) {  
    modCount++;  
    int oldCapacity = elementData.length;  
    if (minCapacity > oldCapacity) {  
        Object oldData[] = elementData;  
        int newCapacity = (oldCapacity * 3)/2 + 1;  
        if (newCapacity < minCapacity)  
            newCapacity = minCapacity;  
        // minCapacity is usually close to size, so this is a win:  
        elementData = Arrays.copyOf(elementData, newCapacity);  
    }  
}
```

12 ArrayList 和 LinkedList 区别

可以这样说：当操作是在一列数据的后面添加数据而不是在前面或中间，并且需要随机地访问其中的元素时，使用 ArrayList 会提供比较好的性能；当你的操作是在一列数据的前面或中间添加或删除数据，并且按照顺序访问其中的元素时，就应该使用 LinkedList 了。

13 Java 中 List、Set、Map 区别

List, Set 都是继承自 Collection 接口；map 自己就是接口， hashmap 就是实现 map 接口

List 特点：元素有放入顺序，元素可重复；

Set 特点：元素无放入顺序，元素不可重复（注意：元素虽然无放入顺序，但是元素在 set 中的位置是有该元素的HashCode 决定的，其位置其实是固定的）；

Map 特点：元素按键值对存储，无放入顺序

6 设计模式

1 设计模式分类

创建型：工厂方法模式，抽象工厂模式，生成器模式，原型模式，单例模式

行为型：责任链，命令，解释器，迭代器，中介者，备忘录，观察者，状态，策略，模板方法，访问者

结构型：适配器，组合，代理，享元，外观，桥接，装饰

2 设计模式的应用

比如 JDK 中 clone 用了原型模式，IO 的管道各种 stream 用了装饰模式，遍历的 Iterator 用了迭代器模式。Spring 中 bean 使用了工程模式，BeanFactory 肯定用到了模板模式，然后 aop 使用了动态代理模式，创建 bean 使用了单例模式，servlet 也是单例模式

3 单例的线程安全的程序

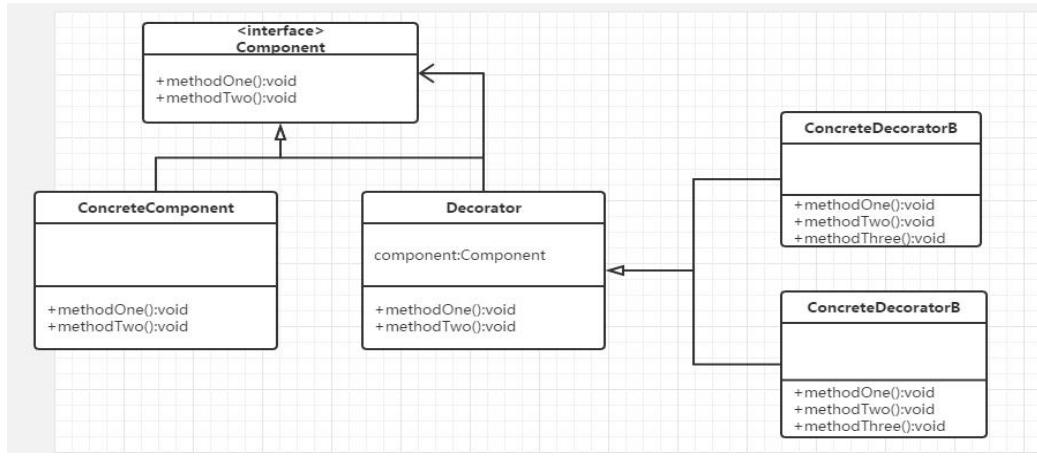
1. 饿汉式单例 饿汉式单例是指在方法调用前，实例就已经创建好了，所以肯定是线程安全的

```
3 /**
4 * Created by louyuting on 17/1/22.
5 * 饿汉式单例模式
6 */
7 public class Singleton1 {
8     //单例
9     private static Singleton1 instance = new Singleton1();
10
11    //构造器私有,不开放
12    private Singleton1() {
13
14    }
15
16    public static Singleton1 getInstance() {
17        return instance;
18    }
19}
```

线程安全的单例-synchronized 方法

```
2
3 import java.util.concurrent.TimeUnit;
4
5 /**
6 * Created by louyuting on 17/1/22.
7 * synchronized 同步方法实现线程安全
8 */
9 public class Singleton3 {
10    //单例
11    private static Singleton3 instance;
12
13    //构造器私有,不开放
14    private Singleton3() {
15
16    }
17
18    public synchronized static Singleton3 getInstance() {
19        if(instance == null) {
20            //创建实例前一些耗时操作
21            try {
22                TimeUnit.MILLISECONDS.sleep(300);
23            } catch (InterruptedException e) {
24                e.printStackTrace();
25            }
26            instance = new Singleton3();
27        }
28        return instance;
29    }
30}
```

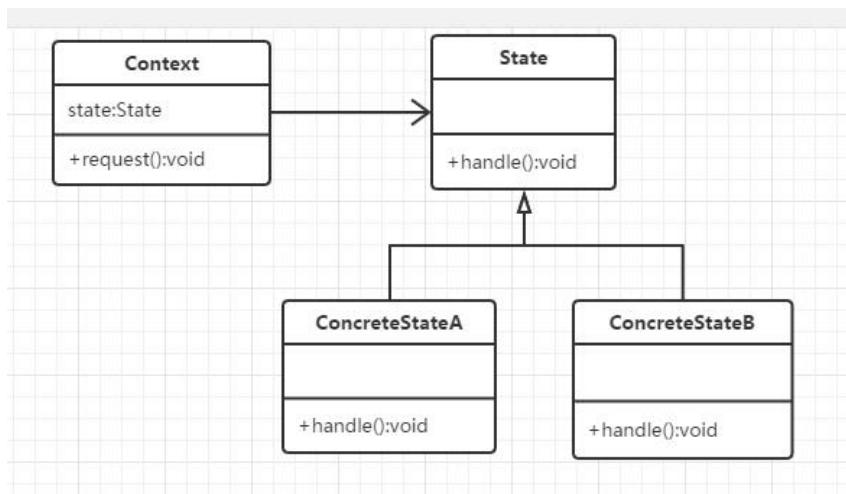
4 装饰模式



动态地给对象添加一些额外的职责。就功能来说装饰模式相比生成子类更为灵活。

比如我现在在玩网易游戏，然后我本来攻击距离只有 3 米，我吃了个苹果，攻击距离变 6 米，我吃了个火龙果，攻击距离变 12 米。

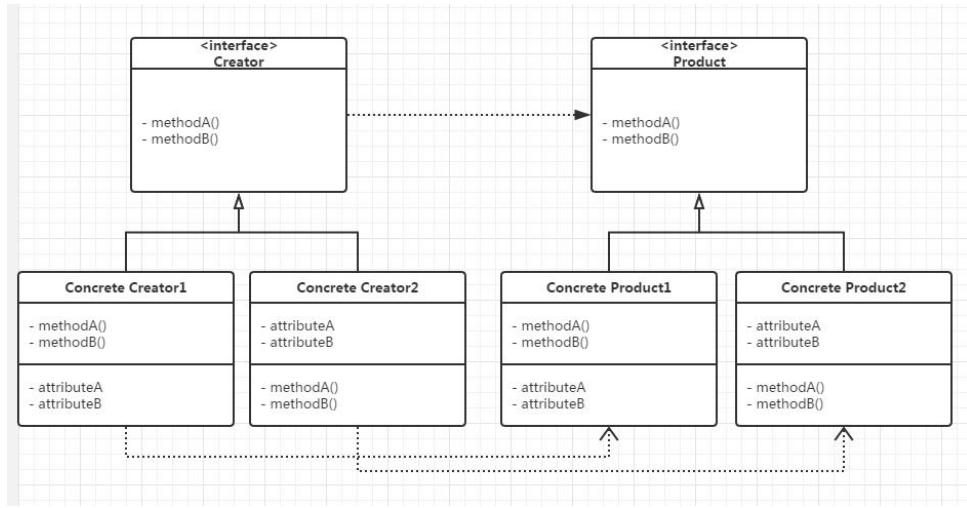
5 状态模式



允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。

比如我现在在玩网易的枪战游戏，我手枪有 5 发子弹，这是一个状态，打了一发，4 发也是一个状态

6 工厂模式



工厂模式的优点

使用工厂模式可以让用户的代码和某个特定类的子类的代码解耦。

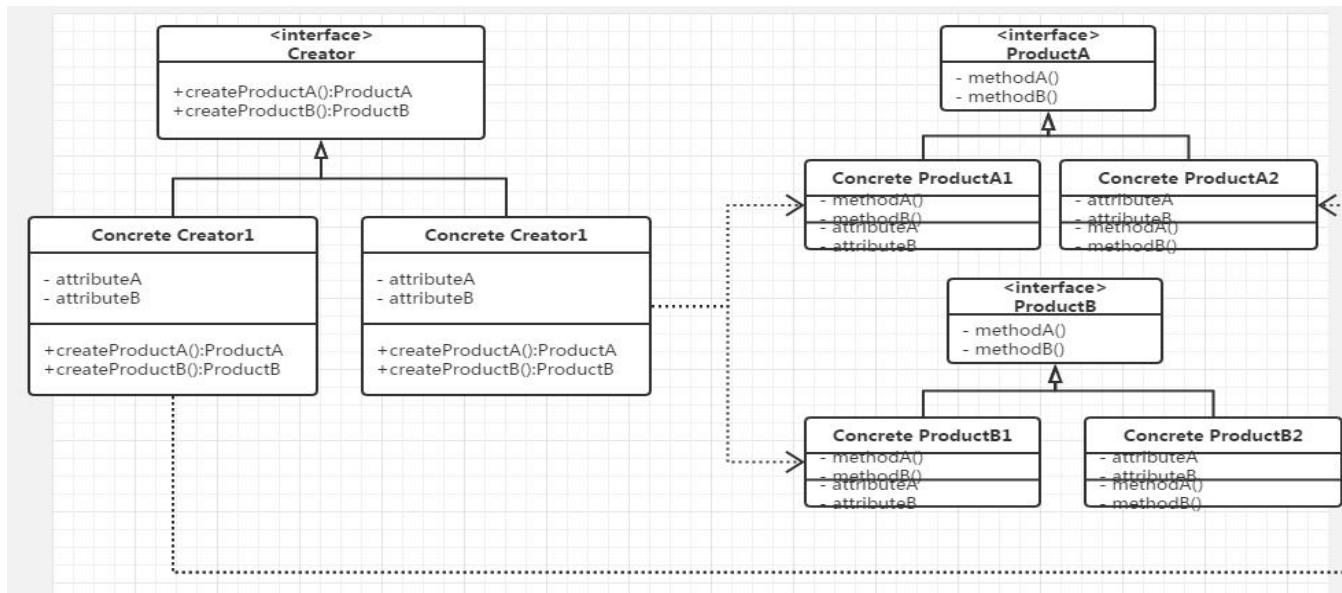
工厂方法的使用用户不必知道它所使用的对象是怎么被创建的，只需要知道该对象有哪些方法即可。

适用工厂模式的情景

用户需要一个类的子类的实例，但不希望与该类的子类形成耦合。

用户需要一个类的子类的实例，但用户不知道该类有哪些子类可用。

7 抽象工厂模式



抽象工厂模式的优点

抽象工厂模式可以为用户创建一系列相关的对象，使用户和创建这些对象的类脱耦。

使用抽象工厂模式可以方便的为用户配置一系列对象。用户使用不同的具体工厂就能得到一组相关的对象，同时也能避免用户混到不同系列的对象中。

在抽象工厂模式中，可以随时增加“具体工厂”为用户提供一组相关的对象。

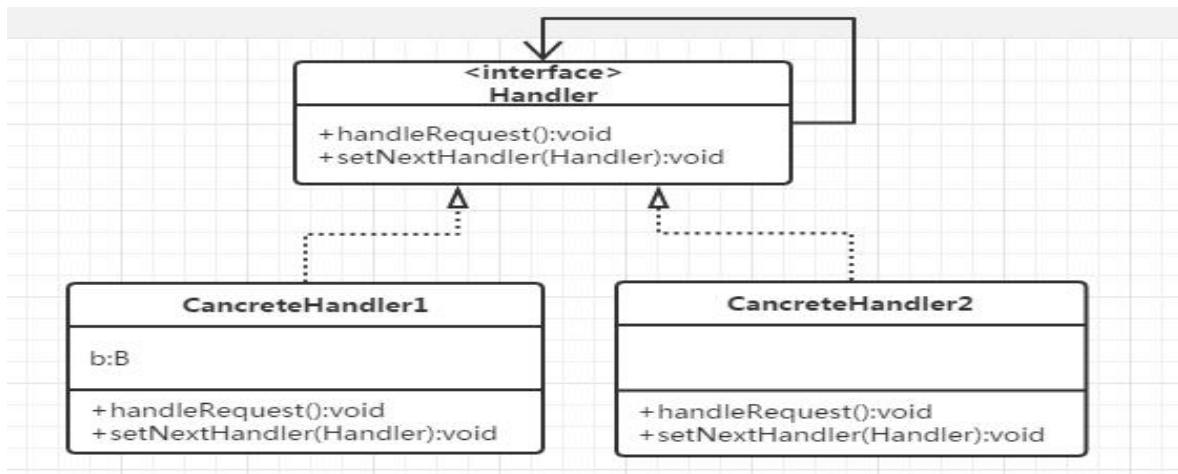
适用抽象工厂模式的情景

系统需要为用户提供多个对象，但不希望用户直接使用 new 运算符实例化这些对象，即希望用户和创建对象的类脱耦。

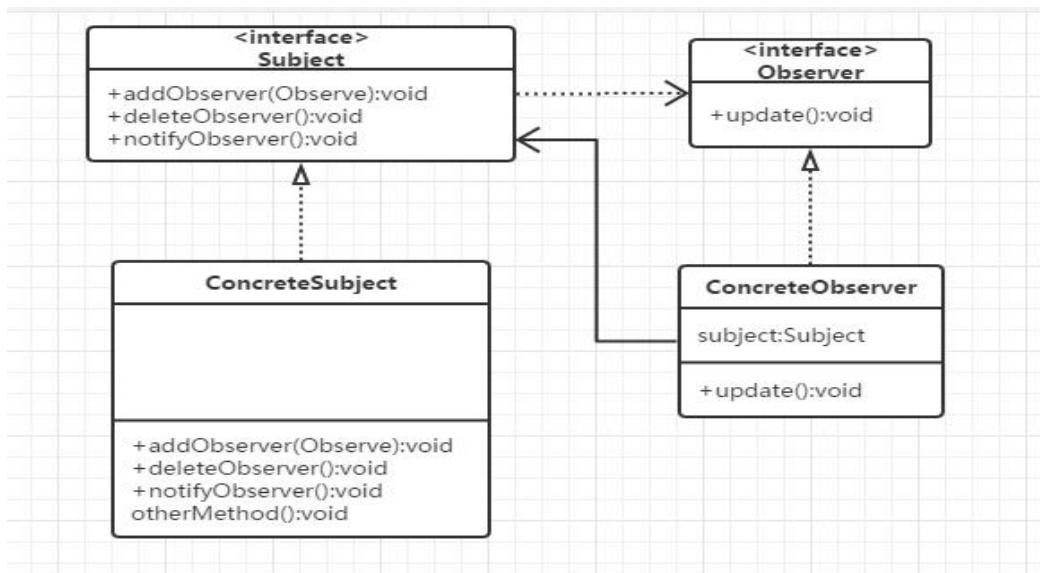
系统需要为用户提供多个相关的对象，以便用户联合使用它们，但又不希望用户来决定这些对象是如何关联的。

系统需要为用户提供一系列对象，但只需要用户知道这些对象有哪些方法可用，不需要用户知道这些对象的创建过程。

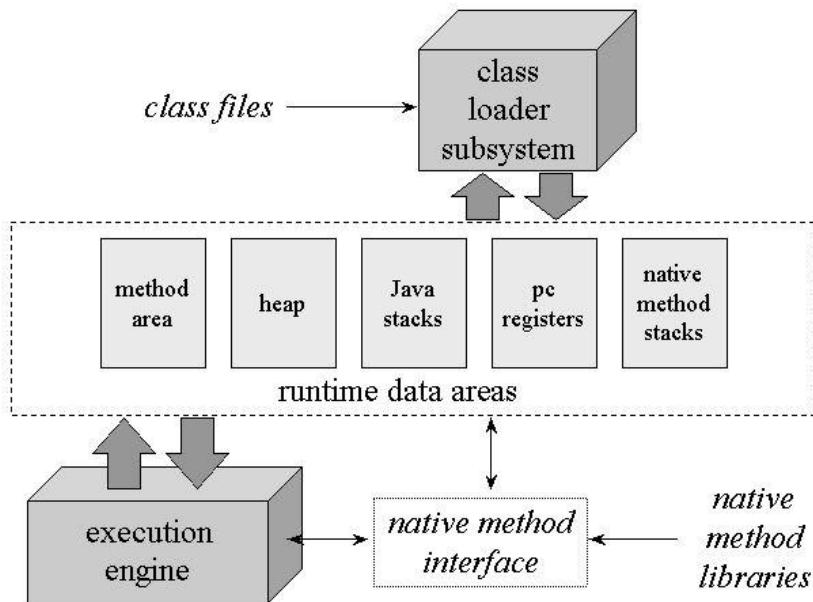
8 责任链模式



9 观察者模式



7 JVM



Baidu 百度

1. JVM 运行时的数据区域 (JVM 的内存模型) :

(1) 程序计数器--线程私有

(2) Java 虚拟机栈--线程私有

每个 Java 方法执行时都会创建一个 **栈帧** 用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个 Java 方法从调用到执行完成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。

(3) 本地方法栈 (也叫 native 栈, 线程私有)

1. 虚拟机栈为虚拟机执行 Java 方法服务;
2. 本地方法栈则为虚拟机用到的 Native 方法服务。

(4) Java 堆 (线程共享区域-虚拟机启动时创建) --也是 GC 管理的地方

(5) 方法区 (线程共享区-虚拟机启动时创建)

用于存储以被虚拟机加载的**类信息、常量、静态变量、即时编译器编译后的代码**等数据。

2. 对象创建与定位访问:

主要通过 new 关键字来实现:

首先判断有没有类 **加载**, 没有加载过就先加载类-->

为新生对象 **分配内存**-->

内存初始化-->

对象的必要设置。

对象内存布局: 对象头、实例数据、对齐填充。

访问定位：通过**虚拟机栈上的 reference** 数据来操作堆上面具体对象。
Hotspot 使用的是直接指针。reference 直接指向 Java 堆上的对象指针。

3. GC—怎么判断对象是否可回收：

- (1) **引用计数算法**：引用一次加 1，失效就减 1。但是**不能解决互相引用的情况。**
- (2) **可达性分析算法**：通过一系列称为“**GC Roots**”的对象作为起始点，从这些节点向下搜索，搜索所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链(**即从 GC Roots 到对象不可达**)时，则证明此对象是不可用的。

4. 判断对象是否可回收的可达性分析算法中，GC Roots 有哪些？

- 1) **虚拟机栈中引用的对象**；
- 2) **方法区中静态属性**引用的对象
- 3) **方法区中常量**引用的对象
- 4) **本地方法栈中 JNI** (即 **native 方法**) 引用的对象

5. Java 中引用类型有哪几种？

强引用、软引用、弱引用、虚引用。引用强度依次减弱。

6. 新生代、老年代、持久代分别指什么？

新生代：主要用来**存放新生的对象**。 **老年代**：主要用来存放**应用程序中生命周期长的对象**。**经历了 N (可配置) 次垃圾回收还健在**
持久代：主要存放的是 **Java 类的类信息**，与垃圾收集要收集的 Java 对象关系不大。

7. GC—有哪些垃圾回收算法？各有什么优缺点？各自应用场景？

- (1) **标记-清除算法**：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。

- A) 缺点：标记和清除效率不高；会存在大量的不连续内存碎片
- (2) 复制算法---新生代才采用的算法：将可用内存分为两块，每次只用其中一块，当这一块内存用完了，就将还活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉。
 - A) 缺点：内存缩小为原来的一半，内存的损失太大。
 - B) 解决方法：将内存分为较大的 Eden 区域和两个较小的 Survivor 区域，每次回收将 Eden 中和 Survivor 中还存活的对象移动到另一块空的 Survivor 区域中。
- (3) 标记整理算法---老年代常采用算法：过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。
- (4) 分代收集算法：在 Java 中把堆分为新生代和老年代，然后根据各块的特点采用最适当的收集算法，其实也就是上面几种算法的结合使用。

8. JVM 中使用了哪些技术来加快内存分配

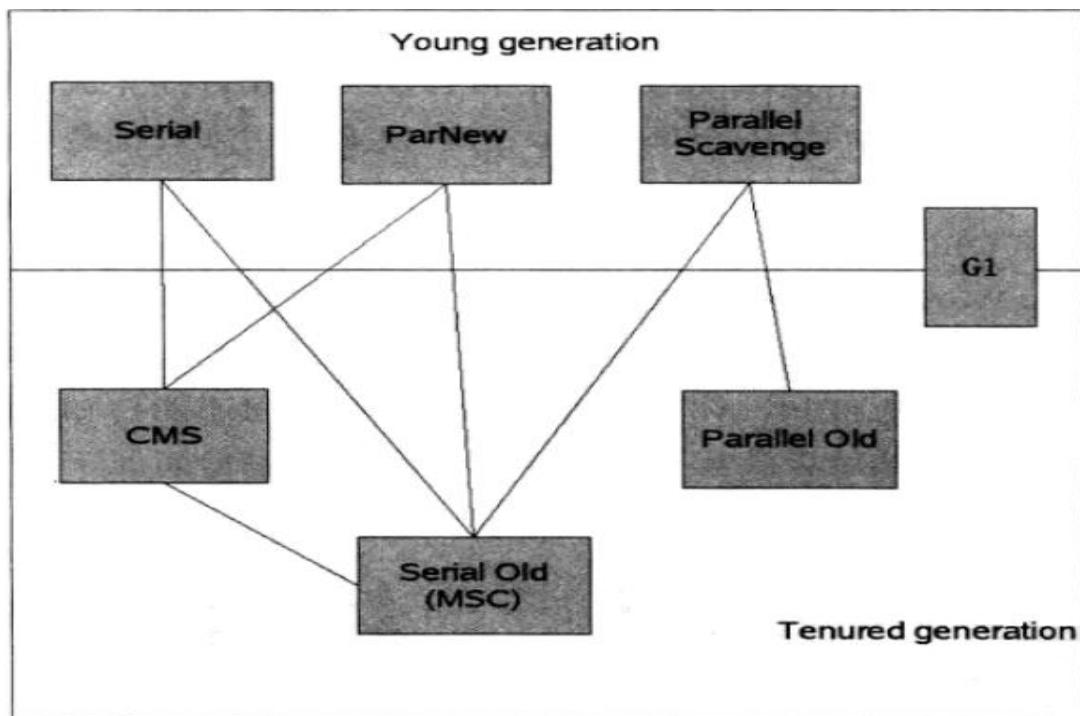
- (1) 指针碰撞：跟踪在 Eden 上新创建的对象。当有新对象创建，只需要判断新创建对象的大小是否满足剩余的 Eden 空间。如果新对象满足要求，则其会被分配到 Eden 空间，同样位于 Eden 的最上面。所以当有新对象创建时，只需要判断此新对象的大小即可，因此具有更快的内存分配速度
- (2) TBAL：在多线程环境下进行内存分配难免要进行加锁，但是我们可以对每个线程分配一个小片空间，这个空间是线程私有的，就可以实现在不加锁情况下的并发分配内存。

9. 怎么理解 Stop the world?

- 1) 可达性分析对执行时间的敏感点的一个体现就是 GC 停顿上面，可达性分析工作必须在一个能确保一致性的快照中进行 - 这里的一致性是指在整个分析期间整个执行系统看起来就像被冻结在某个时间点上，不可以出现分析过程中对象引用关系还在不断变化的情况，该点不满足就无法得到保证。这一点是导致 GC 进行时必须停顿所有 Java 执行线程(sun 称为 “Stop the world”) 的其中一个主要原因。
- 2) 安全点：在 HotSpot 虚拟机中，借助于 OopMap 这种数据结构的协助下，可以快速且准确的完成 GC Roots 的枚举。但是不可能为每条指令都生成 OopMap，这样空间成本就非常高。所以 HotSpot 中只是在“特定的位置”记录了这些信息，这些位置称为安全点，即程序执行时并非在所有地方都停顿下来开始 GC，只有到达安全点时才能暂停。

10. Hotspot 中的垃圾收集器

Hotspot 中常见的垃圾收集器如下图所示：



(1) Serial 收集器—新生代

这个收集器是一个采用[复制算法的单线程的收集器](#), 单线程一方面意味着它只会使用一个CPU或一条线程去完成垃圾收集工作, 另一方面也意味着它进行垃圾收集时必须暂停其他线程的所有工作, 直到它收集结束为止。

(2) ParNew 收集器—新生代

ParNew 收集器其实就是 Serial 收集器的多线程版本, 除了使用多条线程进行垃圾收集之外, 其余行为都与 Serial 收集器完全一样, 包括使用的也是[复制算法](#)。

(3) Parallel 收集器—新生代

Parallel 收集器也是一个新生代收集器, 也是用[复制算法的收集器](#), 也是并行的多线程收集器, 但是它的特点是它的关注点和其他收集器不同。介绍这个收集器主要还是介绍吞吐量的概念。CMS 等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间, 而 Parallel 收集器的目标则是打到一个可控制的吞吐量。所谓吞吐量的意思就是 CPU 用于运行用户代码时间与 CPU 总消耗时间的比值, 即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间), 虚拟机总运行 100 分钟, 垃圾收集 1 分钟, 那吞吐量就是 99%。另外, Parallel 收集器是虚拟机运行在 Server 模式下的默认垃圾收集器。

(4) Serial Old 收集器—老年代

Serial 收集器的老年代版本, 同样是一个单线程收集器, 使用“[标记-整理算法](#)”,

(5) Parallel Old 收集器—老年代

Parallel 收集器的老年代版本, 使用多线程和“[标记-整理](#)”算法。

(6) CMS 收集器—老年代

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为为目标的老年代收集器。

1).初始标记阶段

暂停所有的其他线程，并记录下直接与 root 相连的对象。

2).并发标记阶段

同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以算法里会跟踪记录这些发生引用更新的地方。

3).最终确认标记阶段

将上一阶段做了指针更新的区域和 root 合并为一个伪 root 集合，并对其做 tracing。从而可以保证真正可达的对象一定被标记了。但同时也会产生一部分被标记为可达，但其实已经是不可达的区域，由于已经没有了到达这个区域的路径，所以并没有办法将它的标志位置为 0，则造成了一个暂时的内存泄漏，但这部分空间会在下一次收集阶段被清扫掉。

4).并发清扫阶段

开启用户线程，同时 GC 线程开始对为标记的区域做清扫。这个过程要注意不要清扫了刚被用户线程分配的对象。一个小 trick 就是在这个阶段，将所有新分配的对象置为可达的。

<http://blog.csdn.net/hqq2023623/article/details/50993165>

11. Minor GC 和 Full GC 有什么区别？

- 1) **新生代 GC (Minor GC)** :指发生在新生代的垃圾收集动作，因为大多数 Java 对象存活率都不高，所以 Minor GC 非常频繁，一般回收速度也比较快
- 2) **老年代 GC(Full GC)** :指发生在老年代的垃圾收集动作，出现了 Full GC，经常会伴随至少一次的 Minor GC (但并不是绝对的)。Full GC 的速度一般要比 Minor GC 慢上 10 倍以上

12. 类加载的流程？

加载：类的权限定名导入 Class 文件

—》**连接**----

1 准备

①进行内存分配的仅包括类变量（被 static 修饰的变量）

②初始值是指数据类型的零值，例如：

public static int value = 123;那么变量 value 在准备阶段过后的值是 0，而不是 123，因为这时候尚未开始执行任何 java 方法。再看： public static final int value = 123;final 修饰的字段，子啊准备阶段虚拟机就会根据赋值，value 的值为 123.

2 验证 （1）文件格式验证（2）元数据验证（3）字节码验证（4）符号引用验证，比如字段和方法的访问性(private、protected、public、default)是否可被当前类访问。

3 解析

- » **初始化**: 静态变量的初始化、静态代码块的执行。使用 new 关键字实例化对象
- » **使用**
- » **卸载 unload**

13. JVM 有哪些类加载器？每种类加载器分别加载哪里的 class 代码？

- 1) **启动类加载器**---- 加载位置 : \$JAVA_HOME/lib/rt.jar 里所有的 class 或则被 -Xbootclasspath 参数指定的路径中。
- 2) **扩展类加载器**---- 加载位置 : 加载 JAVA_HOME/lib/ext 目录下的或者被 java.ext.dirs 系统变量指定所指定的路径中所有类库。
- 3) **应用程序类加载器**---- 加载位置 : classpath 环境变量中指定的 jar 包及目录中 class;
- 4) **自定义加载器**---- 自定义的类加载器，手动加载，只需要继承自 ClassLoader 然后实现 loadClass()方法。

14. 类加载的双亲委派模型



双亲委派模型的工作过程是：

- 1) 如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，向上传递。
- 2) 所有的加载请求最终都会传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

如果问能不能自己写 system 类，写是可以写，但是不会被加载。而 System 类是 Bootstrap 加载器加载的，就算自己重写，也总是使用 Java 系统提供的 System，自己写的 System 类根本没有机会得到加载。除非我们自定义个加载器，放在特定目录下去加载，让其他的系统加载器无法加载。

15. Java 的内存模型

Java 内存模型定义了一种多线程访问 Java 内存的规范：

- (1) **Java 内存模型将内存分为了主内存和工作内存。**类的状态是存储在主内存中的，每次 Java 线程用到这些主内存中的变量的时候，都会读一次主内存中的变量，并将这些变量在自己的线程的工作内存中保存一份拷贝，当运行自己线程的代码时候，操作的都是自己工作内存中的变量副本。在线程代码执行完毕之后，会将最新的值更新到主内存中去。
- (2) **定义了 8 个原子操作，用于操作主内存和工作内存中的变量的交互。**
- (3) 定义了 **volatile 变量**，保证了变量的可视性。
- (4) **原子性、可见性、有序性**
- (5) **happens-before：针对变量的可见性制定的一些通用规则**，比如定义了操作 A 必然先行发生于操作 B 的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁 **unlock** 的动作一定先行发生于后面对于同一个锁进行锁定 **lock** 的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的 happens-before 规则，则这段代码一定是线程非安全的。

16 GC 的优化方案？ JVM 优化

对 Java 栈——栈上分配优化的小结：

必须是小对象（一般几十个 bytes），且必须是在没有逃逸的情况下，如果 JVM 使用了逃逸分析优化，则该小对象可以直接分配在栈上，因为栈的空间不大（一般也就到 1m 封顶了），更没有堆大。直接分配在栈上，方法调用完毕，Java 栈帧就立即被移除，故内存可以自动回收，减轻 GC 压力。大对象（栈的空间不允许）或者逃逸的对象无法在栈上分配（即使启动了 JVM 的逃逸分析优化，且因为 Java 栈是线程私有的，不共享，局部对象变量被其他线程或者方法引用了肯定不能在栈分配内存）

基本的原则就是尽可能地减少垃圾和减少 GC 过程中的开销。其中需要注意，JVM 进行次 GC 的频率很高，但因为 Minor GC 占用时间极短，所以对系统产生的影响不大。更值得关注的是 Full GC 的触发条，具体措施包括以下几个方面：

(1)不要显式调用 System.gc()

调用 **System.gc()** 也仅仅是一个请求(建议)。JVM 接受这个消息后，并不是立即做垃圾回收，而只是对几个垃圾回收算法做了加权，使垃圾回收操作容易发生，或提早发生，或回收较多而已。但即便这样，很多情况下它会触发 Full GC，也即增加了间歇性停顿的次数。

(2)尽量减少临时对象的使用

临时对象在跳出函数调用后，会成为垃圾，少用临时变量就相当于减少了垃圾的产生，也就减少了 Full GC 的概率。

(3)对象不用时最好显式置为 Null

一般而言，为 Null 的对象都会被作为垃圾处理，所以将不用的对象显式地设为 Null，有利于 GC 收集器判定垃圾，从而提高了 GC 的效率。

(4)尽量使用 StringBuffer, 而不用 String 来累加字符串

由于 String 是常量，累加 String 对象时，并非在一个 String 对象中扩增，而是重新创建新的 String 对象，如 Str5=Str1+Str2+Str3+Str4, 这条语句执行过程中会产生多个垃圾对象，因为对次作“+”操作时都必须创建新的 String 对象，但这些过渡对象对系统来说是没有实际意义

的，只会增加更多的垃圾。避免这种情况可以改用 `StringBuffer` 来累加字符串，因 `StringBuffer` 是可变长的，它在原有基础上进行扩增，不会产生中间对象。

(5)能用基本类型如 `Int,Long`,就不用 `Integer,Long` 对象

基本类型变量占用的内存资源比相应用对象占用的少得多，如果没有必要，最好使用基本变量。

(6)尽量少用静态对象变量

静态变量属于全局变量，不会被 GC 回收，它们会一直占用内存。

(7)分散对象创建或删除的时间

集中在短时间内大量创建新对象，特别是大对象，会导致突然需要大量内存，JVM 在面临这种情况时，只能进行 `Full GC`，以回收内存或整合内存碎片，从而增加主 GC 的频率。集中删除对象，道理也是一样的。它使得突然出现了大量的垃圾对象，空闲空间必然减少，从而大大增加了下一次创建新对象时强制主 GC 的机会。

17 Java 即使有了 GC 也会出现的内存泄漏情况？举例说明。

1、静态集合类像 `HashMap`、`Vector` 等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，所有的对象 `Object` 也不能被释放，因为他们也将一直被 `Vector` 等应用着。

```
Static Vector v = new Vector();
for (int i = 1; i<100; i++)
{
    Object o = new Object();
    v.add(o);
    o = null;
}
```

在这个例子中，代码栈中存在 `Vector` 对象的引用 `v` 和 `Object` 对象的引用 `o`。在 For 循环中，我们不断的生成新的对象，然后将其添加到 `Vector` 对象中，之后将 `o` 引用置空。问题是当 `o` 引用被置空后，如果发生 GC，我们创建的 `Object` 对象是否能够被 GC 回收呢？答案是否定的。因为，`GC 在跟踪代码栈中的引用时，会发现 v 引用，而继续往下跟踪，就会发现 v 引用指向的内存空间中又存在指向 Object 对象的引用。也就是说尽管 o 引用已经被置空，但是 Object 对象仍然存在其他的引用，是可以被访问到的，所以 GC 无法将其释放掉`。如果在此循环之后，`Object` 对象对程序已经没有任何作用，那么我们就认为此 Java 程序发生了内存泄漏。

2.各种连接，数据库连接，网络连接，IO 连接等没有显示调用 `close` 关闭，不被 GC 回收导致内存泄露。

3.监听器的使用，在释放对象的同时没有相应删除监听器的时候也可能导致内存泄露。

18 JVM 什么时候会触发 Full GC

1、`System.gc()`方法的调用

2、`老年代代空间不足`，老年代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行 `Full GC` 后空间仍然不足，则抛出如下错误：

`Java.lang.OutOfMemoryError: Java heap space`

3、**永生区空间不足** Permanet Generation 中存放的为一些 class 的信息、常量、静态变量等数据，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation 可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果不足也会抛出异常。

4、**CMS GC 时出现 promotion failed 和 concurrent mode failure** 对于采用 CMS 进行老年代 GC 的程序而言，尤其要注意 GC 日志中是否有 promotion failed 和 concurrent mode failure 两种状况，当这两种状况出现时可能会触发 Full GC。

5、统计得到的 **Minor GC 晋升到旧生代的平均大小大于老年代的剩余空间** 这是一个较为复杂的触发情况，Hotspot 为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行 Minor GC 时，做了一个判断，如果之前统计所得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发 Full GC。

6、**堆中分配很大的对象** 所谓大对象，是指需要大量连续内存空间的 java 对象，例如很长的数组，此种对象会直接进入老年代，而老年代虽然有很大的剩余空间，但是无法找到足够大的连续空间来分配给当前对象，此种情况就会触发 JVM 进行 Full GC。

19 Java 相较于 PHP、C#、Ruby 等一样很优秀的编程语言的优势是什么？

(1) 体系结构中立，跨平台性能优越。Java 程序依赖于 JVM 运行，javac 编译器编译 Java 程序为平台通用的字节码文件 (.class)，再由 JVM 与不同操作系统匹配，装载字节码并解释（也有可能是编译，会在第三个问题中说到）为机器指令执行。

(2) 安全性优越。通过 JVM 与宿主环境隔离，且 Java 的语法也一定程度上保障了安全，如废弃指针操作、自动内存管理、异常处理机制等。

(3) 多线程。防止单线程阻塞导致程序崩溃，分发任务，提高执行效率。

(4) 分布式。支持分布式，提高应用系统性能。

(5) 丰富的第三方开源组件。Spring、Struts、Hibernate、Mybatis、Quartz 等等等等。

20 字节码是什么？.class 字节码文件是什么？

(1) 字节码是包含 Java 内部指令集、符号集以及一些辅助信息的能够被 JVM 识别并解释运行的符号序列。字节码内部不包含任何分隔符区分段落，且不同长度数据都会构造为 n 个 8 位字节单位表示。

(2) .class 里存放的就是 Java 程序编译后的字节码，包含了类版本信息、字段、方法、接口等描述信息以及常量池表，一组 8 位字节单位的字节流组成了一个字节码文件。

21 JVM 一些问题，怎么确定服务器端一个死循环

服务器和客户端进行通信是通过 socket，首先产生一个 Socket 实例，通过这个例子中，服务器调用 accept 这个方法来接收从客户端发送的信息，但在时间上产生 Socket 实例必须初始化一个端口。负责接收客户端请求！

客户端必须向服务器发送一个消息产生一个 Socket 实例初始化必须指定服务器的 IP 地址，

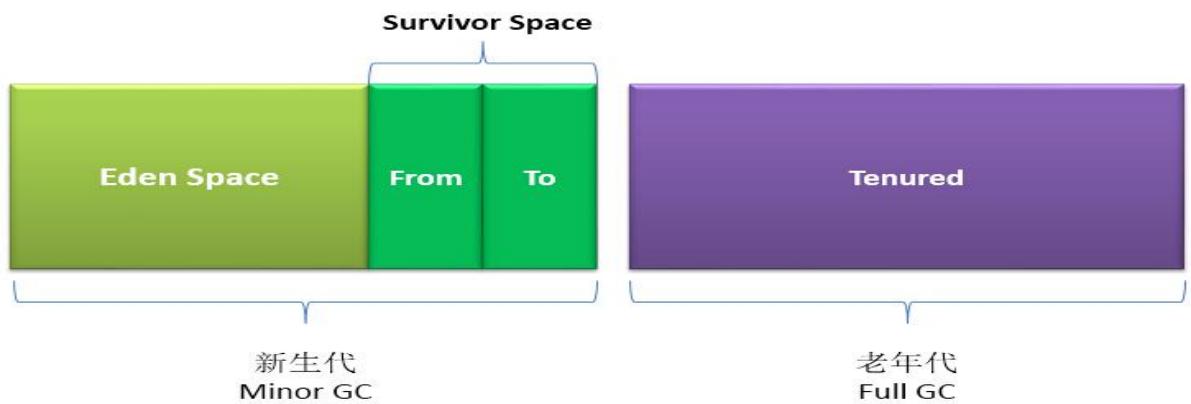
并指定该服务所收取的端口号，以便客户端可以找到你想要接收服务器，发现在可发遍的地方。

22 Java 堆分块（现有内存放不下）



上图中，刻画了 Java 程序运行时的堆空间,可以简述成如下 2 条

- 1.JVM 中堆空间可以分成三个大区，新生代、老年代、永久代
- 2.新生代可以划分为三个区，Eden 区，两个幸存区



新生代GC (Minor GC)：发生在新生代的垃圾收集动作，比较频繁，速度快

老年代GC (Full GC)：经常会伴随至少一次Minor GC。要比Minor GC慢得多
<http://blog.csdn.net/u010723709/article/details/47356507>

23 OOM 异常解析

堆内存的 OOM 异常

a) 如何产生？

堆内存用于存储实例对象，当我们不断创建对象，并且对象都有引用指向(GC Roots 到对象之间有可达路径)，那么垃圾回收机制就不会清理这些对象，当对象多到挤满堆内存的上限

后，就产生 OOM 异常。

b) 模拟堆内存 OOM 异常

PS: 在 eclipse 的 Arguments 中可以设置 VM arguments，这就是 JVM 的一些参数。

–Xms: 设置堆的最小值

–Xmx: 设置堆的最大值

```
public class A(){  
    public static void main(String[] args){  
        while(true){  
            new Person();  
        }  
    }  
}
```

//运行结果中出现: java.lang.OutOfMemoryError:Java heap space//说明是在堆内存中发生了 OOM 异常。

c) 如何解决？

使用内存映像分析工具： Eclipse Memory Analyzer 对 dump 出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，即要搞明白是内存泄漏还是内存溢出。

PS: 内存泄漏导致的 OOM: new 出来的很多对象已经不需要了，但仍然有引用指向，所以垃圾回收机制无法回收。

PS: 内存溢出： new 出来的对象都是需要的，但堆内存太小装不下了。

如果是内存泄漏，通过工具查看泄漏对象到 GC Roots 的引用链。找到泄漏对象是通过怎样的路径与 GC Roots 发生关联，然后导致垃圾回收机制无法自动回收的。

如果不存在内存泄漏，也就是所有的对象都必须存在，**这时候就调大堆内存。**

JVM 栈和本地方法栈的 OOM 异常

a) StackOverflowError

当线程请求的栈深度大于虚拟机所允许的最大栈深度，就会抛出这个异常。

b) OutOfMemoryError

当虚拟机要扩展栈时无法申请到足够空间的内存，就会抛出这个异常。

PS: 这两种异常其实是对同一个问题的两种描述。在单一线程下，不论是栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是 StackOverflowError。通过测试发现，如果给每个线程的 JVM 栈分配的内存越大，大的栈帧在这个 JVM 栈中也能装得下，理应 StackOverflowError 会减少，但事实却恰恰相反：当每个线程的 JVM 栈越大，那么所能创建的线程数就越少，稍微建立几个线程可能就会把有限的内存资源耗尽。

运行时常量池的 OOM 异常

我们通过 String 类的 intern() 方法向方法区中的常量池添加内容。

intern 方法的作用是：当常量池中已经有这个 String 类型所对应的字符串的话，就返回这个字符串的引用；如果常量池中没有这个字符串的话就将这个字符串添加到常量池中，再返回这个字符串的引用。

方法区的 OOM 异常

a) 如何产生？

方法区中存放的是 Class 的相关信息，如：类名、访问修饰符、常量池、字段描述、方法描述等。

如果产生大量的类就有可能将方法区填满，从而产生方法区的 OOM 异常。

b) 注意点

方法区的 OOM 异常是非常常见的，特别是在一些动态生成大量 Class 的应用中(JSP)，需要特别注意类的回收。

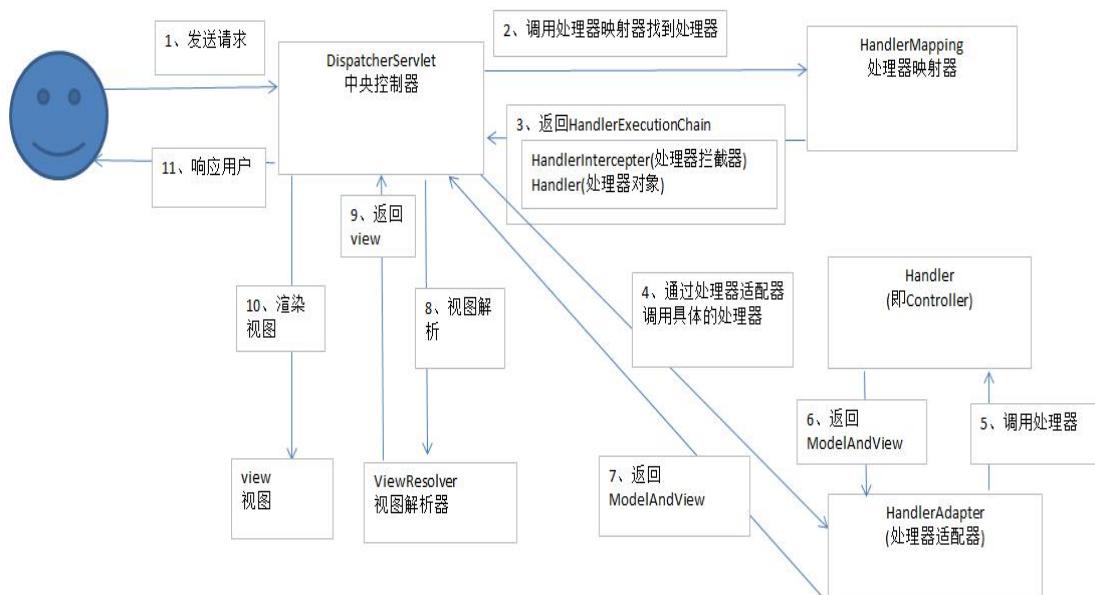
本机直接内存的 OOM 异常

8 框架

1 SpringMVC 框架

1 SpringMVC 的整个流程

SpringMVC 的架构其实也很简单，它通过一个共通的入门 DispatcherServlet 来接收所有的 request，接下来根据 request 要求的页面通过 handler 转送给 Controller 处理，处理结果返回 View 给用户，因此对 SpringMVC 来说，其核心为 DispatcherServlet。



web.xml 配置了前端控制器 DispatcherServlet，第一步：用户发起 request 请求，请求至 DispatcherServlet 前端控制器，第二步：DispatcherServlet 前端控制器请求 HandlerMapping 处理器映射器查找 Handler，第三步：HandlerMapping 处理器映射器，根据 url 及一些配置规则（xml 配置、注解配置）查找 Handler，将 Handler 返回给 DispatcherServlet 前端控制器。第四步：DispatcherServlet 前端控制器调用适配器执行 Handler，有了适配器通过适配器去扩展对不同 Handler 执行方式（比如：原始 servlet 开发，注解开发），第五步：适配器执行 Handler，Handler 是后端控制器，当成模型。第六步：Handler 执行完成返回 ModelAndView。ModelAndView：springmvc 的一个对象，对 Model 和 view 进行封装。第七步：适配器将 ModelAndView 返回给 DispatcherServlet，第八步：DispatcherServlet 调用视图解析器进行视图解析，解析后生成 view 视图解析器根据逻辑视图名解析出真正的视图。View：springmvc 视图封装对象，提供了很多 view，jsp、freemarker、pdf、excel，第九步：ViewResolver 视图解析器给前端控制器返回 view，第十步：DispatcherServlet 调用 view 的渲染视图的方法，将模型数据填充到 request 域。第十一步：DispatcherServlet 向用户响应结果(jsp 页面、json 数据。 . . .)

DispatcherServlet: 前端控制器，由 springmvc 提供
HandlerMappting: 处理器映射器，由 springmvc 提供
HandlerAdapter: 处理器适配器，由 springmvc 提供
Handler: 处理器，需要程序员开发
ViewResolver: 视图解析器，由 springmvc 提供
View: 真正视图页面需要由程序编写

2 SpringMVC 的配置

前端控制器

```
<!-- 前端控制器 -->
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 加载springmvc配置 -->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <!-- 配置文件的地址
            如果不配置contextConfigLocation,
            默认查找的配置文件名称classpath下的: servlet名称"-serlvet.xml"即: springmvc-serlvet.xml
        -->
        <param-value>classpath:springmvc.xml</param-value>
    </init-param>

    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <!--
            可以配置 / ，此工程所有请求全部由springmvc解析，此种方式可以实现 RESTful 方式，需要特殊处理对静态文件的解析不能由springmvc
            可以配置 *.do 或 *.action，所有请求的 url 扩展名为 .do 或 .action 由springmvc 解析，此种方法常用
            不可以 /*，如果配置 /*，返回 jsp 也由springmvc 解析，这是不对的。
        -->
        <url-pattern>*.action</url-pattern>
    </servlet-mapping>
```

SpringMVC.xml

在 springmvc.xml 中配置 springmvc 架构三大组件（处理器映射器、适配器、视图解析器）

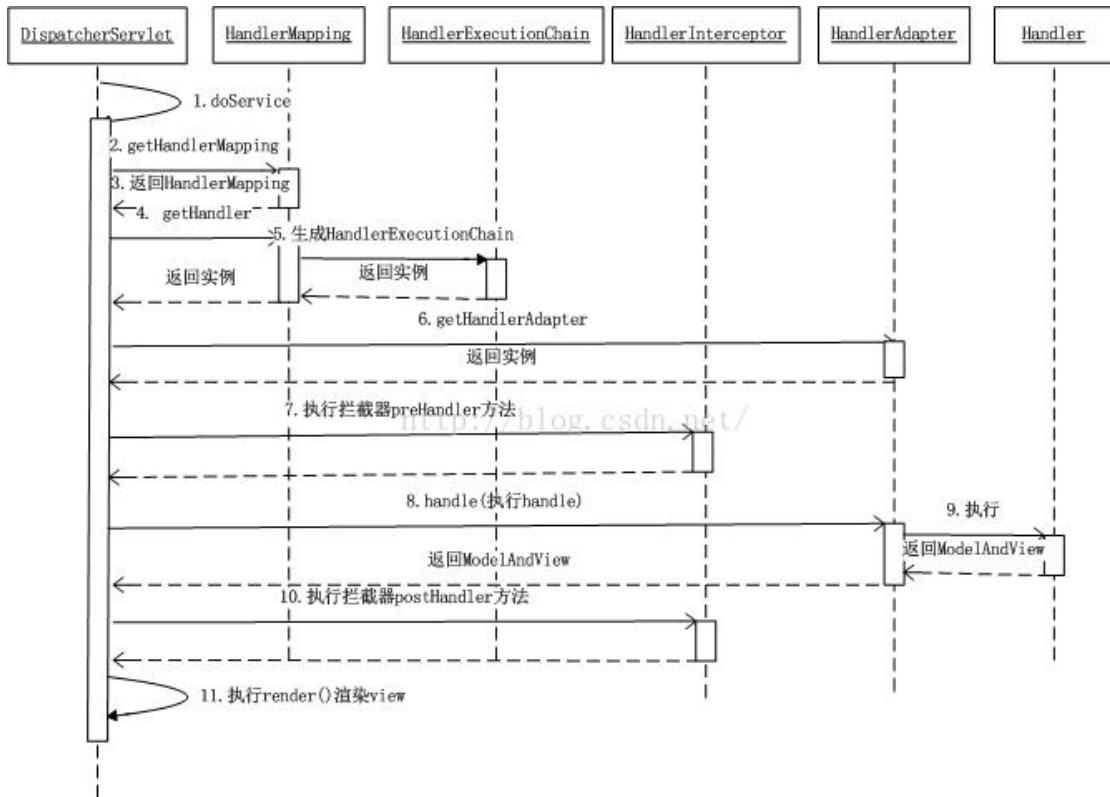
3 SpringMVC 的转发机制（转发和重定向）时序图

重定向和转发有一个重要的不同：当使用转发时，JSP 容器将使用一个内部的方法来调用目标页面，新的页面继续处理同一个请求，而浏览器将不会知道这个过程。与之相反，重定向方式的含义是第一个页面通知浏览器发送一个新的页面请求。因为，当你使用重定向时，浏览器中**所显示的 URL 会变成新页面的 URL**，而当使用转发时，该 URL 会保持不变。重定向的速度比转发慢，因为浏览器还得发出一个新的请求。同时，**由于重定向方式产生了一个新的请求，所以经过一次重定向后，request 内的对象将无法使用。**

不要仅仅为了把变量传到下一个页面而使用 session 作用域，那会无故增大变量的作用域，转发也许可以帮助你解决这个问题。

重定向：以前的 request 中存放的变量全部失效，并进入一个新的 request 作用域。

转发：以前的 request 中存放的变量不会失效，就像把两个页面拼到了一起。



4 SpringMVC 的优势

- 1、上手快，有点 web 经验的，基本上搭建好框架就能写代码了
- 2、轻量，没有 struct 那么复杂，笨重，使用注解大量简化配置，而且良好的支持 rest，而且 structs 自从那次漏洞之后，损失了不少名誉
- 3、Spring，SpringMVC 完美结合，是一家人嘛，甚至结合 spring data（SpringBoot）甚至可以一站式开发

2 Spring 框架（依赖注入的源码）

<http://ifeve.com/spring-interview-questions-and-answers/>

1 Spring 中 IOC 用到了 java 中的那些特性

IoC 的思想最核心的地方在于，资源不由使用资源的双方管理，而由不使用资源的第三方管理，这可以带来很多好处。**第一，资源集中管理，实现资源的可配置和易管理。第二，降低了使用资源双方的依赖程度，也就是我们说的耦合度。**

所谓的**依赖注入**，则是，甲方开放接口，在它需要的时候，能够讲乙方传递进来(注入)

所谓的**控制反转**，甲乙双方不相互依赖，交易活动的进行不依赖于甲乙任何一方，整个活动的进行由第三方负责管理。责任被反转了。

IoC 最原初的目的就是充分利用 OO 的**多态性**，使得通过配置文件而不是在代码里硬编码（hardcode）的方式来实例化对象和装配对象图，这样就有了为不同的客户场景服务的灵活性（不同的客户通过配置文件使用不同的子类）。IoC 本质上和插件化代码的思路很接近。让我们看看 Spring 到底是怎么依赖注入的吧，其实依赖注入的思想也很简单，它是**通过反**

射机制实现的，在实例化一个类时，它通过反射调用类中 set 方法将事先保存在 HashMap 中的类属性注入到类中。

2 Spring AOP 是用什么方法具体实现的

AOP 剖解开封装的对象内部，并将**影响了多个类并且与具体业务无关的公共行为(比如安全, 日志, 事务等)**封装成一个独立的模块（称为切面）然后注入到目标对象（具体业务逻辑）中去。它又能把这些剖开的切面复原，不留痕迹的融入核心业务逻辑中。这样重用都能够带来极大的方便。AOP 技术实现，通过动态代理技术或者是在程序编译期间进行静态的“织入”方式。几个基本术语：1、join point（连接点）：是程序执行中的一个精确执行点，例如类中的一个方法。它是一个抽象的概念，在实现 AOP 时，并不需要去定义一个 join point。2、point cut（切入点）：本质上是一个捕获连接点的结构。在 AOP 中，可以定义一个 point cut，来捕获相关方法的调用。3、advice（通知）：是 point cut 的执行代码，是执行“方面”的具体逻辑。4、aspect（方面）：point cut 和 advice 结合起来就是 aspect，它类似于 OOP 中定义的一个类，但它代表的更多是对象间横向的关系。5、introduce（引入）：为对象引入附加的方法或属性，从而达到修改对象结构的目的。有的 OP 工具又将其称为 mixin。

AOP 的源码中用到了两种动态代理来实现拦截切入功能：**jdk 动态代理和 cglie 动态代理**。**jdk 动态代理**是由 Java 内部的反射机制来实现的（效率高），**cglie 动态代理**底层则是借助 asm 来实现的。**jdk 动态代理**的应用前提，必须是目标类基于统一的接口。

CGLIB (CODE GENERLIZE LIBRARY)代理是**针对类**实现代理，主要是对指定的类**生成一个子类**，覆盖其中的所有方法，所以该类或方法不能声明称 final 的。

如果目标对象没有实现接口，则默认会采用 CGLIB 代理；

如果目标对象实现了接口，可以强制使用 CGLIB 实现代理（添加 CGLIB 库，并在 spring 配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）。

3 Spring 配置事务 事务启动

```
<!-- 配置事务管理器 -->
<!-- 开启事务注解驱动 --> |
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="datasource"></property>
</bean>

<!-- 配置事务属性 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="add*" propagation="REQUIRED" />
    <tx:method name="insert*" propagation="REQUIRED" />
    <tx:method name="delete*" propagation="REQUIRED" />
    <tx:method name="update*" propagation="REQUIRED" />
    <tx:method name="get*" read-only="true" />
    <tx:method name="select*" read-only="true" />
    <tx:method name="*" propagation="REQUIRED" />
  </tx:attributes>
</tx:advice>

<!-- 配置事务切入点，以及把事务切入点和事务属性关联起来 -->
<aop:config>
  <!-- 对 com.ang.elearning.service.impl 包下的所有类的所有方法执行事务 -->
  <aop:pointcut expression="execution(* com.ang.elearning.service.impl.*.*(..))" id="pointcut" />
  <aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut" />
</aop:config>
```

(事务处理必须抛出异常，只有这样 Spring 才帮助事务回滚)

注意@Transactional 只能被应用到 public 方法上，对于其它非 public 的方法，如果标记了 @Transactional 也不会报错，但方法没有事务功能。

①除了在带有切入点，通知和增强器的 Bean 配置文件中声明事务外，Spring 还允许简单地用 @Transactional 注解来标注事务方法。

②为了将方法定义为支持事务处理的，可以为方法添加 @Transactional 注解。根据 Spring AOP 基于代理机制，只能标注公有方法。

③可以在方法或者类级别上添加 @Transactional 注解。当把这个注解应用到类上时，这个类中的所有公共方法都会被定义成支持事务处理的。

④在 Bean 配置文件中只需要启用 <tx:annotation-driven> 元素，并为之指定事务管理器就可以了。

⑤如果事务处理器的名称是 transactionManager，就可以在<tx:annotation-driven> 元素中省略 transaction-manager 属性。这个元素会自动检测该名称的事务处理器。

```
 /**
 * 事务处理必须抛出异常，Spring才会帮助事务回滚
 * @param users
 */

@Transactional
@Override
public void insertUser(List<User> users) {
    // TODO Auto-generated method stub
    for (int i = 0; i < users.size(); i++) {
        if(i<1){
            this.userDao.insert(users.get(i));
        }
        else {
            throw new RuntimeException();
        }
    }
}
```

```
[java] view plain copy print ? C
01. @Test
02.     public void testTransaction(){
03.         List<User> users = new ArrayList<User>();
04.         for(int i= ;i< ;i++){
05.             User user = new User();
06.             user.setAge(i);
07.             user.setPassword(i+"111111");
08.             user.setUserName("测试"+i);
09.             users.add(user);
10.         }
11.         this.userService.insertUser(users);
12.     }
```

注意：此时进行 JUnit 测试会发现出现错误，这是因为方法中抛出了这个异常。实质上确实进行了事务管理，数据没有插入，此时表示配置成功了；反之，如果去掉注解，那么前两条数据会插入成功，然后后面会抛出异常。

4 spring 支持五种事务隔离设置

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 会出现不可重复读、幻读问题（锁定正在读取的行）

REPEATABLE_READ 会出幻读（锁定所读取的所有行）

SERIALIZABLE 保证所有的情况不会发生（锁表）

5 spring 中一共定义了六种事务传播属性

传播属性	描述
REQUIRED	如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行
REQUIRED_NEW	当前的方法必须启动新事务，并在它自己的事务内运行。如果有事务正在运行，应该将它挂起
SUPPORTS	如果有事务在运行，当前的方法就在这个事务内运行，否则它可以不运行在事务中。
NOT_SUPPORTED	当前的方法不应该运行在事务中。如果有运行的事务，将它挂起
MANDATORY	当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常
NEVER	当前的方法不应该运行在事务中。如果有运行的事务，就抛出异常
NESTED	如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行。否则，就启动一个新的事务，并在它自己的事务内运行。

前六个策略类似于 EJB CMT，第七个（PROPAGATION_NESTED）是 Spring 所提供的一个特殊变量。

它要求事务管理器或者使用 JDBC 3.0 Savepoint API 提供嵌套事务行为（如 Spring 的 DataSourceTransactionManager）

6 @Autowired @resource

1、@Autowired 与 @Resource 都可以用来装配 bean. 都可以写在字段上,或写在 setter 方法上。

2、@Autowired 默认按类型装配（这个注解是属业 spring 的），默认情况下必须要求依赖对象必须存在，如果要允许 null 值，可以设置它的 required 属性为 false，如：
@Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier 注解进行使用，如下：

Java 代码

```
@Autowired()@Qualifier("baseDao")  
private BaseDao baseDao;
```

3、@Resource（这个注解属于 J2EE 的），默认按照名称进行装配，名称可以通过 name 属性进行指定，如果没有指定 name 属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在 setter 方法上默认取属性名进行装配。当找不到与名称匹配的 bean 时才按照类型进行装配。但是需要注意的是，如果 name 属性一旦指定，就只会按照名称进行装配。

Java 代码

```
@Resource(name="baseDao")  
private BaseDao baseDao;
```

我喜欢用 @Resource 注解在字段上，且这个注解是属于 J2EE 的，减少了与 spring 的耦合。

最重要的这样代码看起就比较优雅。

7 Spring 三种 bean 注入方式

Spring 中依赖注入有三种注入方式：

- 一、构造器注入； xml 里面有 `constructor-arg` 这个标签
- 二、设值注入（`setter` 方式注入）；
- 三、`Field` 方式注入（注解方式注入）。

8 Spring bean 单例/多例

`bean id="borrowDao":` 标识该 bean 的名称，通过 `factory.getBean("id")` 来获得实例，

`Singleton`：默认为 `true`，即单实例模式，每次 `getBean("id")` 时获取的都是同一个实例，如果设置为 `false`，即原型模式，则每次获取的是新创建的实例。

3 MyBatis 框架

MyBatis 是支持普通 SQL 查询，存储过程和高级映射的优秀持久层框架。几乎消除了所有 JDBC 代码和参数的手工设置。以及结果集的检索。使用简单的 XML 或注解用于配置和原始映射。

- 1> 加载配置 ①通过配置文件 ②Java 中的注解
- 2> SQL 解析
- 3> SQL 执行
- 4> 结果集映射：可以将结果集转换成 `HashMap`, `JavaBean` 或基本数据类型
`xml: <insert></insert> <delete></delete> id parameterType`
`@select`

`DB -> Entity -> Mapper.xml -> mapper.java -> service.java -> controller.java -> jsp`

1 mybatis 一级缓存与二级缓存

mybatis 一级缓存是一个 `SqlSession` 级别，`sqlsession` 只能访问自己的一级缓存的数据，

二级缓存是跨 `sqlSession`，是 `mapper` 级别的缓存，对于 `mapper` 级别的缓存不同的 `sqlsession` 是可以共享的。

2 MyBatis 和 Hibernate 的比较

- 1、**开发上手难度** 只要会写 sql，很快就会用 mybatis
- 2、**SQL 优化方面** Mybatis 的 SQL 会更灵活、可控性更好、更优化。
- 3、**移植性**，MyBatis 项目中所有的 SQL 语句都是依赖所用的数据库的，所以不同数据库类型的移植性不好。Hibernate 与具体数据库的关联只需在 XML 文件中配置即可
- 4、**JDBC** Hibernate 进行了封装，而 Mybatis 是**原生的 JDBC 的**，运行速度上的优势。
- 5、**功能、特性丰富程度** Hibernate 提供了诸多功能和特性。要全掌握很难。

Mybatis 自身功能很有限，但 Mybatis 支持 plugin，可以使用开源的 plugin 来扩展功能。

6、**动态 SQL** Mybatis mapper xml 支持动态 SQL Hibernate 不支持

实际项目关于 Hibernate 和 Mybatis 的选型：

1、数据量：有以下情况最好选用 Mybatis

如果有超过千万级别的表

如果有单次业务大批量数据提交的需求（百万条及以上的），这个尤其不建议用 Hibernate

如果有单次业务大批量读取需求（百万条及以上的）(注， hibernate 多表查询比较费劲，用不好很容易造成性能问题)

2、表关联复杂度

如果主要业务表的关联表超过 20 个（大概值），不建议使用 hibernate

3、人员

如果开发成员多数不是多年使用 hibernate 的情况，建议使用 mybatis

4、数据库对于项目的重要程度

如果项目要求对于数据库可控性好，可深度调优，用 mybatis

3 MyBatis 中批量插入

```
<select id="selectProduct" resultMap="Map">
    SELECT *
    FROM PRODUCT
    WHERE PRODUCTNO IN
        <foreach item="productNo" index="index" collection="参数的类型 List 或 array">
            #{productNo}
        </foreach>
</select>
```

4 MyBatis 构建 sql 时动态传入表名以及字段名

mybatis 中一个属性：**statementType**。这个属性的作用是告诉 mybatis 我们写的这个 sql 到底是预编译（PRESTATEMENT）还是非预编译（STATEMENT）的。如果是预编译的，那么系统在初始化时就会读取这段 sql 代码，将指定的实体类中的字段替换了类似#{ }这样的语句，就是形成了类似这样的语句：“select * from tableName where code=?” 这个时候你在系统运行时再想向这句 sql 中替换 tableName 或者 code，结果可想而知。如果是非预编译呢，结果刚好相反，他会在系统运行时才会去生成这样类似的语句。此时就可以去替换这些动态的字段或者表名之类。这样在结合之前所讲的返回类型的设置，我们的问题就解决了

```

1. <select id="queryMetaList" resultType="Map" statementType="STATEMENT">
2.     select * from ${tableName} t where
3.     <foreach item="item" index="index" collection="field" open=" "
4.         separator="and" close=" ">
5.         <choose>
6.             <when test="item.fieldType == 'DATE' and item.dateQueryFlag == 0">
7.                 ${item.fieldCode} between
8.                 to_date('${item.fieldValue}', 'yyyy-mm-dd
9.                 hh24:mi:ss')
10.            </when>
11.            <when test="item.fieldType == 'DATE' and item.dateQueryFlag == 1">
12.                to_date('${item.fieldValue}', 'yyyy-mm-dd
13.                hh24:mi:ss')
14.            </when>
15.            <when test="item.fieldItemCode != null and item.fieldItemCode != ''">
16.                ${item.fieldCode} =
17.                '${item.fieldItemCode}'
18.            </when>
19.            <otherwise>
20.                ${item.fieldCode} =
21.                '${item.fieldValue}'
22.            </otherwise>
23.        </choose>
24.    </foreach>
25. </select>

```

对了，漏了一句，如果是非预编译的话，最好使用\${}而不是#{}

4 Shiro 框架

1. 配置

```

<!-- shiro start -->
    <!-- 1. 配置 SecurityManager -->
    <!-- 2. 配置 CacheManager -->
        <!-- 2.1 需要加入 ehcache 的 jar 包及配置文件 -->
    <!-- 3. 配置 Realm -->
        <!-- 3.1 直接配置继承了 org.apache.shiro.realm.AuthorizingRealm 的 bean -->
    <!-- 4. 配置 LifecycleBeanPostProcessor，可以自定义地来调用配置在 Spring IOC 容器中 shiro
        bean 的生命周期方法 -->
    <!-- 5. 使能够在 IOC 容器中使用 shiro 的注解，但必须在配置了 LifecycleBeanPostProcessor
        之后才可以使用 -->
    <!-- 6. 配置 ShiroFilter -->
        <!-- 6.1 id 必须和 web.xml 中配置的 DelegatingFilterProxy 的<filter-name>一致。 如果不
            一致，会抛出 NoSuchBeanDefinitionException 异常，因为 shiro 会在 IOC 容器中查找名称和
            <filter-name> 值一致的 filter bean -->
    <!-- 7. 配置使用自定义认证器，可以实现多 Realm 认证，并且可以根据登录类型指定使用特
        定的 Realm -->

```

```

<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager" />
    <property name="LoginUrl" value="/Login.jsp" />
    <property name="unauthorizedUrl" value="/Login.jsp" />
    <!-- 配置哪些页面需要受保护, 以及访问这些页面需要的权限 -->
    <property name="filterChainDefinitions">
        <value>
            <!-- 第一次匹配优先的原则 -->
            /login.jsp = anon
            /user/login = anon
            /admin/login = anon
            /teacher/login = anon

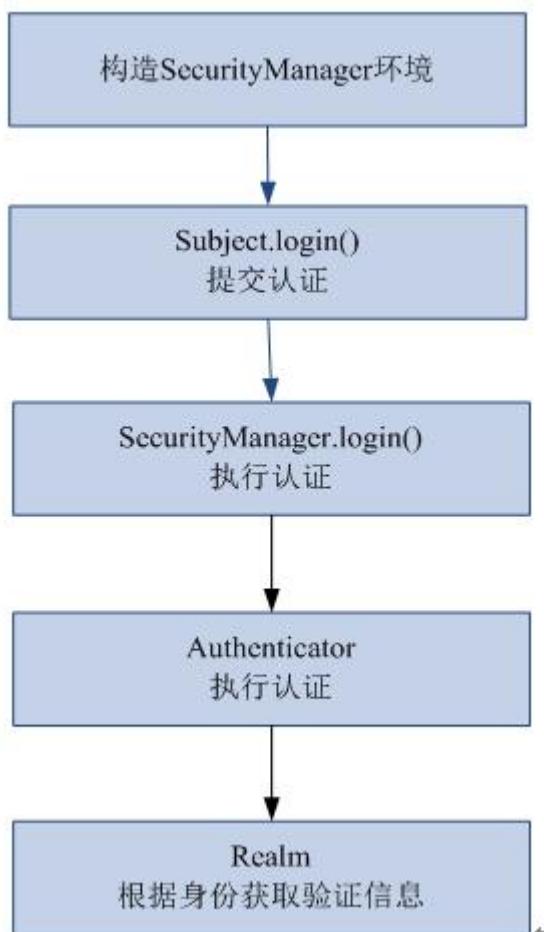
            <!-- 测试: 只有admin角色可以访问test.jsp -->
            /test.jsp = roles[admin]

            /logout = logout

            /** = authc
        </value>
    </property>
</bean>

```

2. 认证流程



- 1 创建 securityManager 工厂，通过 ini 文件创建 securityManager 工厂
- 2 创建 securityManager 3 用 SecurityUtils 将 securityManager 设置当前的运行环境中
- 3 SecurityUtils 里面创建一个 subject，在认证提交前要准备 token（令牌）执行认证提交 subject.login(token); 这个时候，其实是把 login 交给了 securityManager，Subject subject =

securityManager.login(this, token); securityManager 又交给了 authenticate, authenticate 又用自己类的对象 this.authenticator.authenticate(token); 这个类初始化的时候得到了 realm, this.authenticator = new ModularRealmAuthenticator(); 然后就是通过这个 realm 去作验证。具体用的时候 ModularRealmAuthenticator 的 doAuthenticate 的方法。

自定义 realm, 然后配置文件配置 realm, 在 controller 里面读取配置文件, 就可以自动调用 realm 的授权与认证的方法。

用于认证 doGetAuthenticationInfo 用于授权的 doGetAuthorizationInfo

3. 多 realm 认证

与 spring 结合之后, 在 spring 配置文件配置 securityManager, 并制定 authenticator 是自定义的 authenticator, 自定的 authenticator 继承自 ModularRealmAuthenticator。

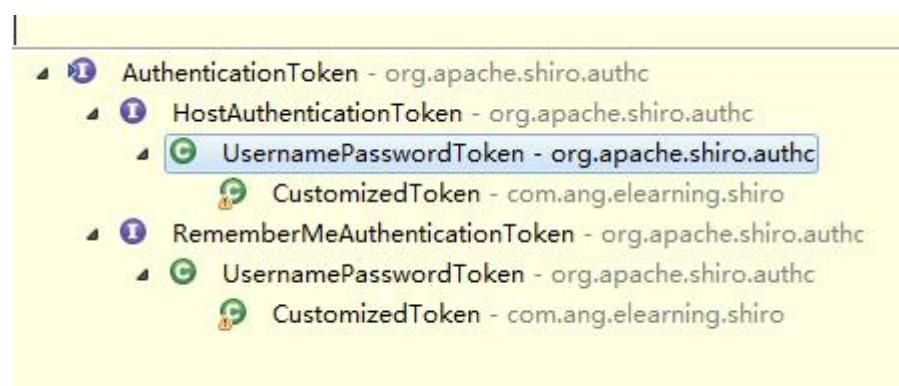
管理员和用户在两张表里面, shiro 默认是所有的参与认证, 都是交给 ModularRealmAuthenticator 的 doAuthenticate 方法, 一次性都丢进去了

```
protected AuthenticationInfo doAuthenticate(AuthenticationToken authenticationToken) throws
    assertRealmsConfigured();
    Collection<Realm> realms = getRealms();
    if (realms.size() == 1) {
        return doSingleRealmAuthentication(realms.iterator().next(), authenticationToken);
    } else {
        return doMultiRealmAuthentication(realms, authenticationToken);
    }
}
```

我写了个实现类, 继承它, 重写 doAuthenticate 方法

```
public class CustomizedModularRealmAuthenticator extends ModularRealmAuthenticator {
    @Override
    protected AuthenticationInfo doAuthenticate(AuthenticationToken authenticationToken)
        throws AuthenticationException {
        // 判断getRealms()是否返回为空
        assertRealmsConfigured();
        // 强制转换回自定义的CustomizedToken
        CustomizedToken customizedToken = (CustomizedToken) authenticationToken;
        // 登录类型
        String loginType = customizedToken.getLoginType();
        // 所有realm
        Collection<Realm> realms = getRealms();
        // 登录类型对应的所有realm
        Collection<Realm> typeRealms = new ArrayList<>();
        for (Realm realm : realms) {
            if (realm.getName().contains(loginType))
                typeRealms.add(realm);
        }
        // 判断是单realm还是多realm
        if (typeRealms.size() == 1)
            return doSingleRealmAuthentication(typeRealms.iterator().next(), customizedToken);
        else
            return doMultiRealmAuthentication(typeRealms, customizedToken);
    }
}
```

而且那个 token 也需要自定义一下, 因为要取出我们的 loginType



```

5 public class CustomizedToken extends UsernamePasswordToken {
6
7     //登录类型，判断是普通用户登录，教师登录还是管理员登录
8     private String loginType;
9
10    public CustomizedToken(final String username, final String password, String loginType) {
11        super(username, password);
12        this.loginType = loginType;
13    }
14
15    public String getLoginType() {
16        return loginType;
17    }
18
19    public void setLoginType(String loginType) {
20        this.loginType = loginType;
21    }

```

4. 授权

什么是权限管理？

权限管理是系统的安全范畴，要求必须是合法的用户才可以访问系统（用户认证），且必须具有该 资源的访问权限才可以访问该 资源（授权）。

认证：对用户合法身份的校验，要求必须是合法的用户才可以访问系统。

授权：访问控制，必须具有该 资源的访问权限才可以访问该 资源。

权限模型：标准权限数据模型包括：用户、角色、权限（包括资源和权限）、用户角色关系、角色权限关系。

权限分配：通过 UI 界面方便给用户分配权限，对上边权限模型进行增、删、改、查操作。

权限控制：

基于角色的权限控制：根据角色判断是否有操作权限，因为角色的变化 性较高，如果角色修改需要修改控制代码，系统可扩展性不强。

基于资源的权限控制：根据资源权限判断是否有操作权限，因为资源较为固定，如果角色修改或角色中权限修改不需要修改控制代码，使用此方法系统可维护性很强。建议使用。

5. 三种授权方法

Shiro 支持三种方式的授权：

■ 编程式：通过写 if/else 授权代码块完成：

```

Subject subject = SecurityUtils.getSubject();
if(subject.hasRole("admin")){
    //有权限
} else {
    //无权限
}

```

■ 注解式：通过在执行的 Java 方法上放置相应的注解完成：

```

@RequiresRoles("admin")
public void hello() {
    //有权限
}

```

■ JSP/GSP 标签：在 JSP/GSP 页面通过相应的标签完成：

```

<shiro:hasRole name="admin">
<!— 有权限 —>

```

</shiro:hasRole>

5 redis 框架

名称	修改时间
001--string.txt	15
002--list.txt	15
003--hashset.txt	15
004--set.txt	15
005--sortedset.txt	15
006--key操作.txt	15
007--事务.txt	15
008--主从复制.txt	15
009--持久化.txt	15
010--虚拟内存.txt	15
011--服务器管理.txt	15
012--管线.txt	15
013--内存优化.txt	15

1.redis 基本操作

Set sadd key member[member...] 对 key 的 set 增加值，返回增加元素的个数，重复只加一次
smembers key 获取 set 中的所有 member

sismember key member 判断值是否是 set 的 member。是返回 1，否则，返回 0

scard key 返回 set 的 member 个数，如果 set 不存在，返回 0

spop key 移除并返回一个随机 member

srem key member [member ...] 移除一个或多个 member

smove source destination member 将 source 中的 member 移动到 destination

List lpush key value[value] 把元素插入表头。按照从左到右的次序插。返回插入元素的个数
lpushx key value 插入表头元素，当且仅当列表 key 存在时，才能插。返回列表中元素的个数
rpush key [value] 将值插入到队列的队尾。从左到右依次添加。返回列表中元素个数

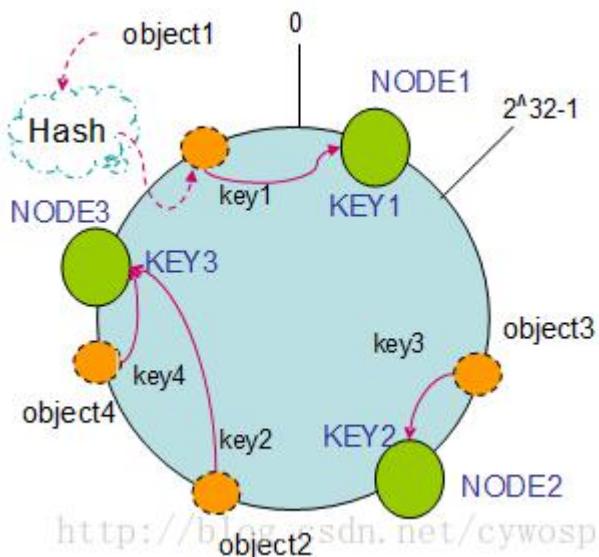
lindex key index 通过索引 index 获取列表的元素。index>=0 时，0 表头，1 第二个元素，依次类推；index<0 时，-1，表尾，-2 倒数第二个元素，依次类推

lrange key start stop 获取指定开始和结束范围的一些列元素。0：表头，-1：表尾。如果 stop 指定的元素在 start 的左边，返回空列表

lrem key count value 移除等于 value 的元素，当 count>0 时，从表头开始查找，移除 count 个；当 count=0 时，从表头开始查找，移除所有等于 value 的；当 count<0 时，从表尾开始查找，移除|count| 个。

ltrim key start stop 保留指定区域的元素，其他元素全部删除

2.Jedis 分片连接池(分布式)



从运行结果中可以看到，不同的 key 被分配到不同的 Redis-Server 上去了。

总结：客户端 jedis 的一致性哈希进行分片原理：初始化 ShardedJedisPool 的时候，会将上面程序中的 jdsInfoList 数据进行一个算法技术，主要计算依据为 list 中的 index 位置来计算

存在扩容问题和单点故障问题（主库宕机之后经过虚拟 ip 找到从库，从库改了之后要和主库换一下 slaveofnoone，才能保证修复期的一致性）

3.Redis 实现消息队列

TaskProducer

```
try {
    Thread.sleep(random.nextInt(600) + 600);
    // 模拟生成一个任务
    UUID taskid = UUID.randomUUID();
    // 将任务插入任务队列: task-queue
    jedis.lpush("task-queue", taskid.toString());
    System.out.println(Thread.currentThread().getName() + "插入了一个新的任务: " + taskid);
    System.out.println("----" + jedis.llen("task-queue") + "----");
} catch (Exception e) {
    e.printStackTrace();
}
```

TaskConsumer

```

// 从任务队列"task-queue"中获取一个任务，并将该任务放入暂存队列"tmp-queue"
String taskid = jedis.rpoplpush("task-queue", "tmp-queue");
System.out.println("+++" + jedis.llen("tmp-queue") + "+++");
// 处理任务----纯属业务逻辑，模拟一下：睡觉
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

// 模拟成功和失败的偶然现象
if (random.nextInt(13) % 7 == 0) { // 模拟失败的情况，概率为2/13
    // 将本次处理失败的任务从暂存队列"tmp-queue"中，弹回任务队列"task-queue"
    jedis.rpoplpush("tmp-queue", "task-queue");
    System.out.println(taskid + "处理失败，被弹回任务队列");
} else { // 模拟成功的情况
    // 将本次任务从暂存队列"tmp-queue"中清除
    jedis.rpop("tmp-queue");
    System.out.println(taskid + "处理成功，被清除");
}

```

6 Hadoop 框架

Hadoop 分布式系统基础框架

Hadoop 实现了一个分布式文件系统 HDFS 它的核心 1> HDFS 提供存储 2> MapReduce 提供计算 采用"分而治之"的思想，把对大规模数据集的操作，分发给一个主节点管理下的各个分节点共同完成，然后通过整合各个节点的中间结果，得到最终结果。简单地说，MapReduce 就是"任务的分解与结果的汇总"。

在 Hadoop 中，用于执行 MapReduce 任务的机器角色有两个：一个是 JobTracker；另一个是 TaskTracker，JobTracker 是用于调度工作的，TaskTracker 是用于执行工作的。一个 Hadoop 集群中只有一台 JobTracker。

<http://www.cnblogs.com/xia520pi/archive/2012/05/16/2504205.html>

9 分布式、高并发分布式问题

1 1 个亿的数据要我选出最大的 10 个数问时间空间最省的情况用什么算法，然后时间复杂度是多少

由于(1)输入的大量数据；(2)只要前 K 个，对整个输入数据的保存和排序是相当的不可取的。可以利用数据结构的**最小堆**来处理该问题。

最小堆如图所示，对于每个非叶子节点的数值，一定不大于孩子节点的数值。这样可用含有 K 个节点的最小堆来保存 K 个目前的最大值(当然根节点是其中的最小数值)。每次有数据输

入的时候可以先与根节点比较。若不大于根节点，则舍弃；否则用新数值替换根节点数值。并进行最小堆的调整。

因为要前 K 个嘛，如果是最大堆，只能知道最上面的是最大的，而无法判断最小的节点具体在哪儿，就无法一下子知道下一个数是不是前 K 大，如果是最小堆，上面的元素就是目前前 K 的，如果比它大就说明这个不是前 K 大，换掉重新调整最小堆就行，如果比目前前 K 还小，直接剔除就行。

我只回答了会考虑到两个问题，一是数据存储问题（比如一个服务器挂了，怎么保证数据不受损失），另外一个是大量用户访问时候，服务器承受问题。具体怎么解决，用什么访问，我没有给出相关的解决方案，读到此文的希望查一下

2 实现淘宝秒杀

所有的秒杀请求都会入到同一个[请求队列](#)（这个请求队列实现方式有多种），当收到的请求数等于被秒杀的对象数，之后所有的请求都会被拒绝（直接返回），之前的请求都会得到相应的回复。

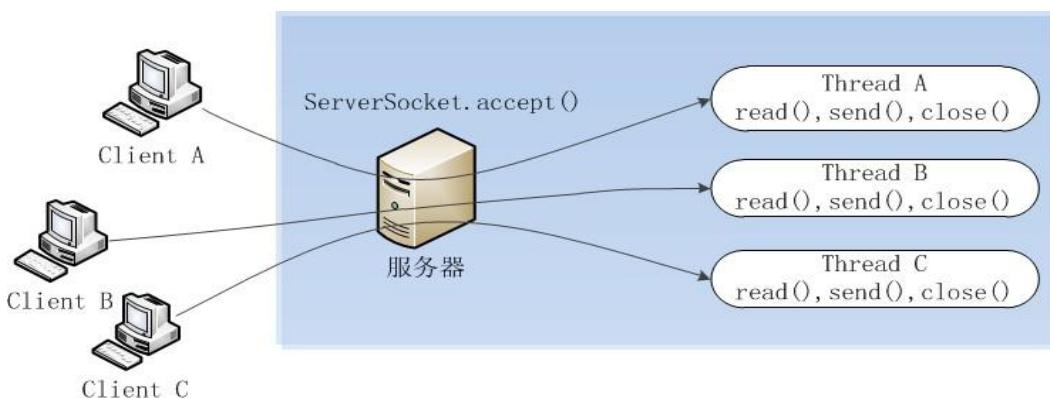
比如请求队列可以用 redis 实现。

3 Java NIO

1 java NIO 和阻塞 I/O 的区别

1. 阻塞 I/O 通信模型

假如现在你对阻塞 I/O 已有了一定了解，我们知道阻塞 I/O 在调用 `InputStream.read()` 方法时是阻塞的，它会一直等到数据到来时（或超时）才会返回；同样，在调用 `ServerSocket.accept()` 方法时，也会一直阻塞到有客户端连接才会返回，每个客户端连接过来后，服务端都会启动一个线程去处理该客户端的请求。阻塞 I/O 的通信模型示意图如下：



如果你细细分析，一定会发现阻塞 I/O 存在一些缺点。根据阻塞 I/O 通信模型，我总结了它的[两点缺点](#)：

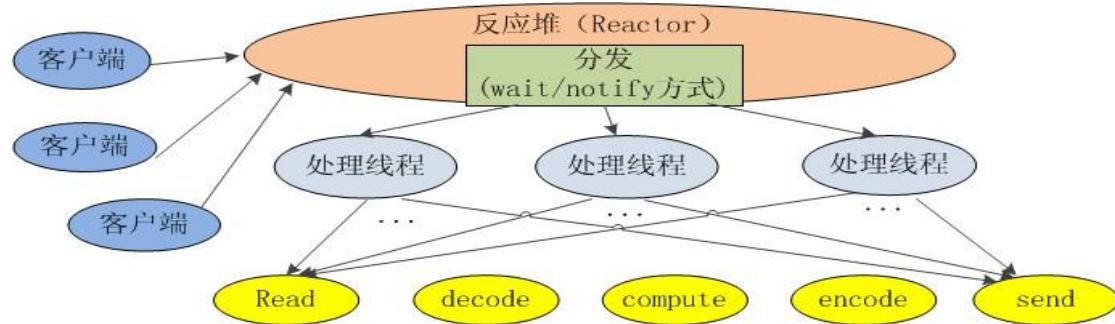
1. 当客户端多时，会创建大量的处理线程。且每个线程都要占用栈空间和一些 CPU 时间
2. 阻塞可能带来频繁的上下文切换，且大部分上下文切换可能是无意义的。

在这种情况下非阻塞式 I/O 就有了它的应用前景。

2. [java NIO 原理及通信模型](#)

Java NIO 是在 jdk1.4 开始使用的，它既可以说成“新 I/O”，也可以说成非阻塞式 I/O。下面是 **java NIO 的工作原理**：

1. 由一个专门的线程来处理所有的 IO 事件，并负责分发。
 2. 事件驱动机制：事件到的时候触发，而不是同步的去监视事件。
 3. 线程通讯：线程之间通过 `wait,notify` 等方式通讯。保证每次上下文切换都是有意义的。
- 减少无谓的线程切换。阅读过一些资料之后，下面贴出我理解的 java NIO 的工作原理图：



2 IO 与 NIO 的区别 2

IO 是面向流的，而 NIO 是面向块的（缓冲区）。面向缓冲区的方式中，一次性可以获取或者写入一整块数据，而不需要一个字节一个字节的从流中读取。面向块的方式处理数据的速度会比流方式更快。

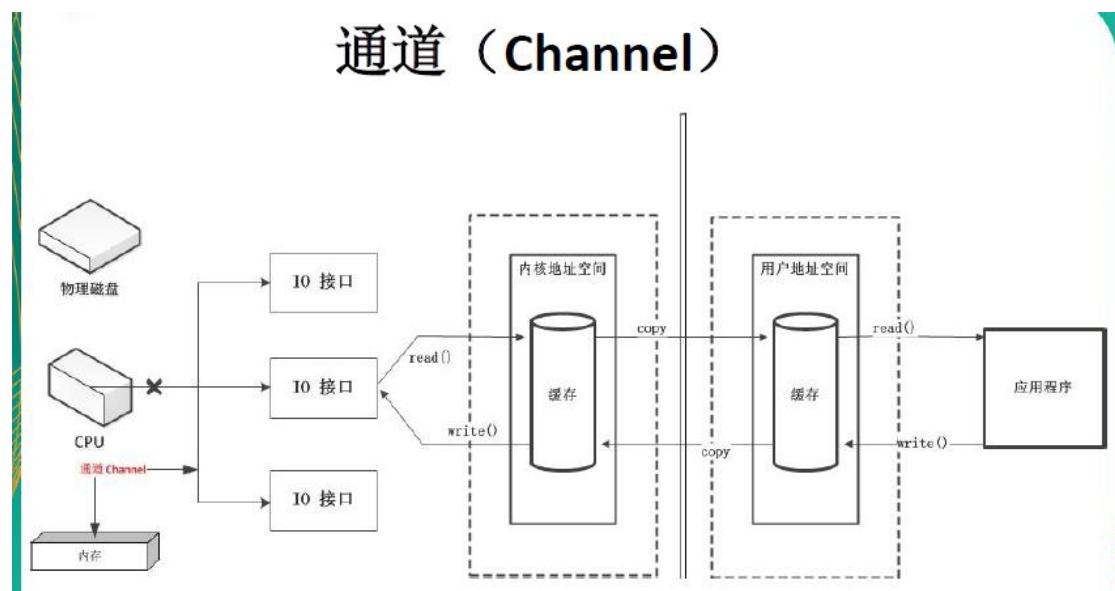
3 NIO 基础 通道（完全独立的处理器，专门用于 IO）

通道 **Channel** 与缓冲器 **Buffer**

Buffer 是一个保存数据的地方，包括刚刚写入的数据，以及被读取的数据，主要用来追踪系统的读写进程。

Channel 与流模式比较类似，但是，永远无法将数据直接写入到 Channel 或者从 Channel 中读取数据。需要通过 Buffer 与 Channel 交互。

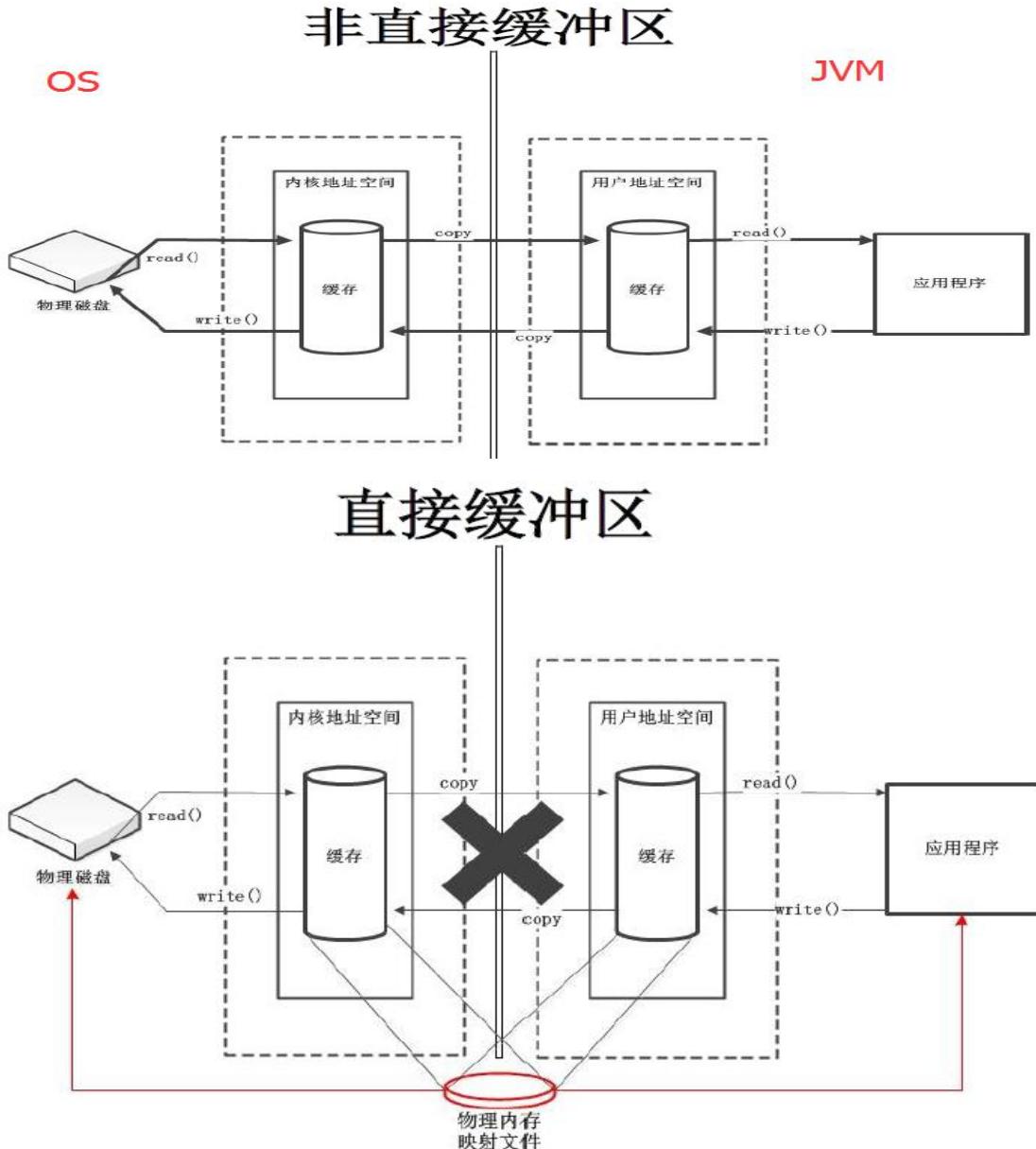
简而言之，Channel 负责传输，Buffer 负责存储



4 NIO 的缓冲区

非直接缓冲区：通过 `allocate()` 方法分配缓冲区，将缓冲区建立在 JVM 的内存中

直接缓冲区：通过 `allocateDirect()` 方法分配直接缓冲区，将缓冲区建立在物理内存中。可以提高效率，但是耗费资源空间大，而且一旦写入了就不受 JVM 控制。



下图就是使用直接缓冲区完成文件的复制(内存映射文件)

调用 channel 的 map 去建立物理内存映射文件，只有 ByteBuffer 支持

```
3 //使用直接缓冲区完成文件的复制(内存映射文件)
4 @Test
5 public void test2() throws IOException{//2127-1902-1777
6     long start = System.currentTimeMillis();
7
8     FileChannel inChannel = FileChannel.open(Paths.get("d:/1.mkv"), StandardOpenOption.READ);
9     FileChannel outChannel = FileChannel.open(Paths.get("d:/2.mkv"), StandardOpenOption.WRITE,
10
11     //内存映射文件
12     MappedByteBuffer inMappedBuf = inChannel.map(MapMode.READ_ONLY, 0, inChannel.size());
13     MappedByteBuffer outMappedBuf = outChannel.map(MapMode.READ_WRITE, 0, inChannel.size());
14
15     //直接对缓冲区进行数据的读写操作
16     byte[] dst = new byte[inMappedBuf.limit()];
17     inMappedBuf.get(dst);
18     outMappedBuf.put(dst);
19
20     inChannel.close();
21     outChannel.close();
22
23     long end = System.currentTimeMillis();
24     System.out.println("耗费时间为: " + (end - start));
25 }
```

5 NIO 的读与写

读取

第一步，获取通道

```
FileInputStream inputStream = new FileInputStream("read.txt");FileChannel inputChannel =
inputStream.getChannel();
```

第二步，创建缓冲器

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

第三步，将 channel 中的数据读取到 buffer 中。相当于写入 buffer

```
int readBytes = inputChannel.read(buffer);
```

写入

将数据写入到 buffer 中，然后再将 buffer 中的数据写入到 channel 中

```
FileOutputStream outputStream = new FileOutputStream("output.txt");
```

```
FileChannel outputChannel = inputStream.getChannel();
```

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

```
buffer.put(new String("message").getBytes());
```

```
buffer.flip();
```

```
outputChannel.write(buffer);
```

6 NIO 完成网络通信的三大核心

```
@Test
public void send() throws IOException{
    DatagramChannel dc = DatagramChannel.open();
    dc.configureBlocking(false);
    ByteBuffer buf = ByteBuffer.allocate(1024);
    Scanner scan = new Scanner(System.in);

    while(scan.hasNext()){
        String str = scan.next();
        buf.put((new Date().toString() + ":" + str).getBytes());
        buf.flip();
        dc.send(buf, new InetSocketAddress("127.0.0.1", 9898));
        buf.clear();
    }

    dc.close();
}

public void receive() throws IOException{
    DatagramChannel dc = DatagramChannel.open();
    dc.configureBlocking(false);
    dc.bind(new InetSocketAddress(9898));
    Selector selector = Selector.open();
    dc.register(selector, SelectionKey.OP_READ);
    while(selector.select() > 0){
        Iterator<SelectionKey> it = selector.selectedKeys().iterator();
        while(it.hasNext()){
            SelectionKey sk = it.next();
            if(sk.isReadable()){
                ByteBuffer buf = ByteBuffer.allocate(1024);
                dc.receive(buf);
                buf.flip();
                System.out.println(new String(buf.array(), 0, buf.limit()));
                buf.clear();
            }
        }
        it.remove();
    }
}
```

4 大型网站性能逐步优化过程:

- (1) 应用和数据的分离:也就是专门的应用服务器、数据库服务器、文件服务器
 - (2) 使用缓存, 专门的缓存服务器(集群) 改善数据访问性能。注意这里的数据访问二八定律。
 - (3) 应用服务器集群, 改善并发处理请求能力。
 - (4) 数据库的读写分离改善数据库访问压力(主从热备功能)
 - (5) CDN 加速和方向代理提高访问速度。(都是基于缓存)
1. CDN 部署在网络提供商的机房, 使用用户请求网站服务时可以从距离自己最近的网络提供商机房获取数据)
 2. 反向代理则部署在网站的机房中心, 当用户请求到达机房中心后, 首先访问的服务器是反向代理服务器, 如果反向代理服务器中缓存着用户请求的资源, 就将其直接返回给用

户，就不用经过应用服务器。

不管是 CDN 还是反向代理的目的都是为了尽早的把数据返回给用户，一方面加快用户访问速度，也能减轻后端服务器的负载压力。

(6) 分布式数据库服务器

(7) 业务拆分

(8) 分布式服务，使用消息队列。

记住：首先是使用业务手段解决业务问题，再用技术手段解决。

5 高性能

(1) 前端优化：

1. 使用浏览器缓存、静态资源使用独立的域名(减少 cookie 传输)、JS 压缩
2. CDN 加速：使用 CDN 缓存的一般都是一些静态资源，比如：图片，CSS，Script 脚本，静态网页等等。这些文件访问频率极高，将其缓存在 CDN 可极大的改善网页的打开速度。也就是实现所谓的动静（资源）分离
3. 反向代理：配置缓存功能加快 web 请求，当用户第一次请求静态内容时，静态内容就被缓存在反向代理服务器上。这样当其他用户访问时就可以直接从反向代理服务器直接返回，加快 web 请求的响应速度，减轻后端服务器压力。

(2) 应用服务器优化：缓存、集群、异步。

1. 缓存：分布式缓存存在一致性 Hash 问题，
2. 集群：负载均衡：负载均衡算法
3. 异步：消息队列（存在延迟），优点：提高可用性、加快访问速度、消除峰值

6 高可用：服务器硬件故障时服务依旧可用

session 集群： session 定位？

高可用数据：

7 CAP 理论（一致性、可用性、分区容错性）

A 是必须要保证的，所以对于 CP 基本是放弃的； 对于分布式系统一般是保证 AP 放弃强一致性 C（对于分布式系统来说 P 是基本要求）

Consistency(一致性)，数据一致更新，所有数据变动都是同步的

Availability(可用性)，好的响应性能

Partition tolerance(分区容错性) 可靠性

定理：任何分布式系统只可同时满足二点，没法三者兼顾。

忠告：架构师不要将精力浪费在如何设计能满足三者的完美分布式系统，而是应该进行取舍。

8 BASE 理论

BASE 模型反 ACID 模型，完全不同 ACID 模型，牺牲高一致性，获得可用性或可靠性：

Basically Available 基本可用。支持分区失败(e.g. sharding 碎片划分数据库)

Soft state 软状态 状态可以有一段时间不同步，异步。

Eventually consistent 最终一致，最终数据是一致的就可以了，而不是时时高一致。

9 伸缩性：应用服务器集群

- (1) http 重定向负载均衡：利用 Http 的重定向功能，客户端重定向到实际应用服务器
- (2) DNS 域名解析负载均衡：DNS 负载均衡服务器解析请求 并返回实际的 IP 地址，然后再次访问实际应用服务器；
常规用法是：DNS 域名解析作为第一级负载均衡，获得的一组服务器是同样提供负载均衡的内网服务器，然后内网的负载均衡服务器再在内网进行第二级负载均衡，分发到实际 Web 应用服务器。
- (3) 反向代理负载均衡：反向代理服务器除了缓存资源外还可以提供负载均衡。
- (4) IP 负载均衡：在网络层修改请求的目标地址进行负载均衡。
- (5) 数据链路层负载均衡：在通信协议的数据链路层修改 Mac 地址进行负载均衡。

10 路由算法

- (1) 余数 Hash 算法：扩容时存在大量的命中失效
- (2) 一致性 Hash 算法：使用 Hash 环均匀分布； 为了解决一致性 hash 算法带来的负载不均衡问题，通过加入加一层虚拟层来解决。

11 负载均衡算法：

- (1)轮询法（Round Robin）
- (2)随机法（Random）
- (3)源地址 Hash 法（Hash）
- (4)加权轮询法（Weight Round Robin）
- (5)加权随机法（Weight Random）
- (6)最小连接数法（Least Connections）

12 常用的 hash 算法有哪些？

- 1) 余数 Hash 算法
- 2) 一致性 Hash 算法

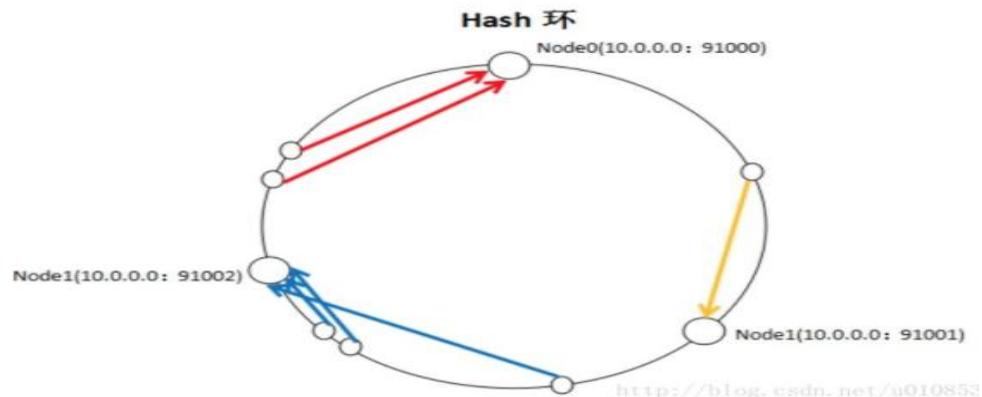
13 解决 hash 冲突的方法有哪些：

- 1) 开放定址法
- 2) 再哈希法

- 3) 链地址法
- 4) 建立一个公共的溢出区。

14 什么是一致性 hash

一致性 hash 是一种分布式 hash 算法。分布式集群中，对机器的添加删除，或者机器故障后自动脱离集群这些操作是分布式集群管理最基本的功能。如果采用常用的余数 hash 算法会导致大部分的原有数据失效，这是非常致命的。一致性 hash 算法，利用一个 hash 环的数据结构来实现，将集群中的不同机器通过计算其 hash 值均匀的分布在 hash 环上。当一个任意的 key 计算出其 hash 值之后，只需要在环上寻找顺时针下最近的结点。如果对于集群中任意一个机器的删除或增加，只会使很小的一部分的数据失效，大部分数据还是有效的。当机器的增加或则删除导致的负载不均衡，我们可以通过增加一个虚拟层来解决。



15 怎么理解分布式锁？（淘宝秒杀为例）

分布式锁主要是解决**数据的最终一致性**。先来看看分布式锁应该是怎样的？

- 1) 分布式环境下，一个方法同一时间只能被一个机器上一个线程执行；
 - 2) 可重入锁(避免死锁)
 - 3) 阻塞锁；
 - 4) 高可用的获取锁和释放锁的性能
- 简单点说：分布式锁就是控制分布式系统中同步访问共享资源的方式。（比如淘宝秒杀）

1. 基于数据库实现：

- a) 创建一张锁表，通过操作数据表中的数据实现，表的结构是这样的：

```
CREATE TABLE `methodLock` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `method_name` varchar(64) NOT NULL DEFAULT '' COMMENT '锁定的方法名',
  `desc` varchar(1024) NOT NULL DEFAULT '' COMMENT '备注信息',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '保存数据时间',
  PRIMARY KEY (`id`),
  unique key `uidx_method_name` (`method_name`) USING BTREE
);
```

表中的 method_name 做了唯一性约束，当有多个请求同时提交到数据库的时候，数据库会保证只有一个操作可以成功，操作成功的线程获得了锁。

- b) 当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候

就删除这条记录。

b) 注意这里的记录必须以该资源作为唯一标识(主键)

该方法有一些缺点：

- a) 依赖数据库的高可用高性能，数据库挂掉就不可用了。
- b) 没有设置失效时间，一旦解锁失败会导致锁永远不被释放。
- c) 锁只能是非阻塞的，因为 `insert` 语句操作失败就会报错直接返回。
- d) 不可重入。

缺点解决办法：

- a) 数据库可以配置主从同步；
- b) 数据库设置定时任务，定时清理失效记录；
- c) 非阻塞？可以在方法前加 `while` 循环直到成功；

数据库的另外一种实现方案：排它锁

`select For update`

在查询语句后面增加 **for update**，数据库会在查询过程中给数据库表增加排他锁。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。我们可以认为获得排它锁的线程即可获得分布式锁。

2. 基于缓存：比如 redis

先来看看 redis 的一些基本命令：

顺便提一句 redis 本身就是单线程的。

`# SETNX key value`

如果 `key` 不存在，就设置 `key` 对应字符串 `value`。在这种情况下，该命令和 `SET` 一样。

当 `key` 已经存在时，就不做任何操作。`SETNX` 是“`SET if Not exists`”。

`# expire KEY seconds`

设置 `key` 的过期时间。如果 `key` 已过期，将会被自动删除。

`# del KEY`

删除 `key`

- a) 怎么实现加锁：`key` 是于商品 ID 相关的字符串来唯一标识，`value` 不重要，只要这个唯一的 `key-value` 存在，就表示这个商品已经上锁。

- b) 怎么释放锁：`redis` 里面删除相应的键值对

- c) 阻塞实现，如果已经上锁就阻塞，不过可以在客户端使用自旋锁轮询。

设置过期时间解决异常问题。

16 如何避免用户多次点击造成的多次请求

1> 定义标志位:

点击触发请求后，标志位为 `false` 量；请求（或者包括请求后具体的业务流程处理）后，标志位为 `true` 量。通过标志位来判断用户点击是否具备应有的响应。

2> 卸载及重载绑定事件:

点击触发请求后，卸载点击事件；请求（或者包括请求后具体的业务流程处理）后，重新载入绑定事件。

3> 替换（移除）按钮 DOM

点击触发请求后，将按钮 DOM 对象替换掉（或者将之移除），自然而然此时不在具备点击事件；请求（或者包括请求后具体的业务流程处理）后，给新的按钮 DOM 定义点击事件。

10 数据库（优化*）

1 sql 复习

内链接

两张表如果有外键关系可以使用内链接，因为通过内链接，每一条只能返回单条记录

只要在关联表中查询出一条记录就可以使用内链接

外链接

只有部分记录可以从关联表查询到，主查询表想要显示所有记录，只能和关联表通过外链接

左外链接 右外链接

`left join left` 左边是主查询表，`left` 右边是关联查询表

`right join right` 右边是主查询表，`right` 左边是关联查询表

子查询

通过关联表的主键关联查询绝对没有问题，也可以不通过关联表的主键查询（但是子查询关联查询的结果集只能有一条记录）

在朱表上作子查询，主表记录会全部显示

嵌套表查询

将嵌套的结果集当成一个类，嵌套子查询中列不能重复的

嵌套表的层级有限制

举例：

例子：

a表	id	name	b表	id	job	parent_id
	1	张三		1	23	1
	2	李四		2	34	2
	3	王武		3	34	4

a.id 同 parent_id 存在关系

```

1) 内连接
select a.* , b.* from a inner join b on a.id=b.parent_id
结果是
1 张三          1    23    1
2 李四          2    34    2

2) 左连接
select a.* , b.* from a left join b on a.id=b.parent_id
结果是
1 张三          1    23    1
2 李四          2    34    2
3 王武          null

3) 右连接
select a.* , b.* from a right join b on a.id=b.parent_id
结果是
1 张三          1    23    1
2 李四          2    34    2
null           3    34    4

4) 完全连接
select a.* , b.* from a full join b on a.id=b.parent_id
结果是
1 张三          1    23    1
2 李四          2    34    2
null           3    34    4
3 王武          null

```

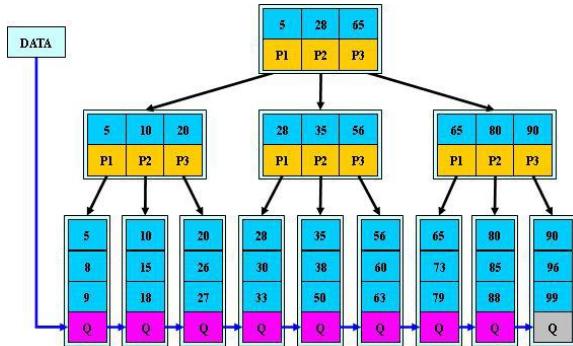
2 事物的特性

事务四大特性(简称 ACID)

- 1、**原子性(Atomicity)**: 事务中的全部操作在数据库中是不可分割的，要么全部完成，要么均不执行。
- 2、**一致性(Consistency)**: 几个并行执行的事务，其执行结果必须与按某一顺序串行执行的结果相一致。
- 3、**隔离性(Isolation)**: 事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须是透明的。
- 4、**持久性(Durability)**: 对于任意已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障。

3 mysql 的索引

B+Tree



B+树是 B-树的变体，也是一种多路搜索树：

1. 其定义基本与 B-树同，除了：
 2. 非叶子结点的子树指针与关键字个数相同；
 3. 非叶子结点的子树指针 $P[i]$ ，指向关键字值属于 $[K[i], K[i+1])$ 的子树（B-树是开区间）；
 5. 为所有叶子结点增加一个链指针；
 6. 所有关键字都在叶子结点出现；
- B+的搜索与 B-树也基本相同，区别是 B+树只有达到叶子结点才命中（B-树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；
- B+的特性：有了这些特性才会选择 B+树作索引
1. 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；
 2. 不可能在非叶子结点命中；
 3. 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；
 4. 更适合文件索引系统；

MySQL 中普遍使用 B+Tree 做索引，但在实现上又根据聚簇索引和非聚簇索引而不同。

非聚簇索索

MyISAM 上，非聚簇索引就是指 B+Tree 的叶子节点上的 data，并不是数据本身，而是数据存放的地址。主索引和辅助索引没啥区别，只是主索引中的 key 一定得是唯一的。非聚簇索引比聚簇索引多了一次读取数据的 IO 操作，所以查找性能上会差。

聚簇索引

Innodb 存储引擎中，所谓聚簇索引，就是指主索引文件和数据文件为同一份文件。在该索引实现方式中 B+Tree 的叶子节点上的 data 就是数据本身，key 为主键，如果是一般索引的话，data 便会指向对应的主索引。

在 B+Tree 的每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的 B+Tree。做这个优化的目的是为了提高区间访问的性能，例如图 4 中如果要查询 key 为从 18 到 49 的所有数据记录，当找到 18 后，只需顺着节点和指针顺序遍历就可以一次性访问到所有数据节点，极大提升了区间查询效率。

在列上加索引时事有条件的：

- 1、经常被查询的列

- 2、order by 子句中使用的列
- 3、是外键或者主键的列
- 4、列是唯一的列
- 5、两个或多个列经常同时出现在 where 子句中或者连接条件中

MySQL 的几种索引

1. 普通索引

这是最基本的索引，它没有任何限制，比如上文中为 title 字段创建的索引就是一个普通索引，MyISAM 中默认的 BTREE 类型的索引，也是我们大多数情况下用到的索引。

```
01 -直接创建索引
02 CREATE INDEX index_name ON table(column(length))
03 -修改表结构的方式添加索引
04 ALTER TABLE table_name ADD INDEX index_name ON (column(length))
05 -创建表的时候同时创建索引
06 CREATE TABLE `table` (
07 `id` int(11) NOT NULL AUTO_INCREMENT ,
08 `title` char(255) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL ,
09 `content` text CHARACTER SET utf8 COLLATE utf8_general_ci NULL ,
10 `time` int(10) NULL DEFAULT NULL ,
11 PRIMARY KEY (`id`),
12 INDEX index_name (title(length))
13 )
14 -删除索引
15 DROP INDEX index_name ON table
```

2. 唯一索引

与普通索引类似，不同的就是：**索引列的值必须唯一，但允许有空值（注意和主键不同）**。如果是**组合索引**，则列值的组合必须唯一，创建方法和普通索引类似。

```
01 -创建唯一索引
02 CREATE UNIQUE INDEX indexName ON table(column(length))
03 -修改表结构
04 ALTER TABLE table_name ADD UNIQUE indexName ON (column(length))
05 -创建表的时候直接指定
06 CREATE TABLE `table` (
07 `id` int(11) NOT NULL AUTO_INCREMENT ,
08 `title` char(255) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL ,
09 `content` text CHARACTER SET utf8 COLLATE utf8_general_ci NULL ,
10 `time` int(10) NULL DEFAULT NULL ,
11 PRIMARY KEY (`id`),
12 UNIQUE indexName (title(length))
13 );
```

3. 全文索引（FULLTEXT）

MySQL 从 3.23.23 版开始支持全文索引和全文检索，FULLTEXT 索引仅可用于 MyISAM 表；他们可以从 CHAR、VARCHAR 或 TEXT 列中作为 CREATE TABLE 语句的一部分被创建，或是随

后使用 ALTER TABLE 或 CREATE INDEX 被添加。///对于较大的数据集，将你的资料输入一个没有 FULLTEXT 索引的表中，然后创建索引，其速度比把资料输入现有 FULLTEXT 索引的速度更为快。不过切记对于大容量的数据表，生成全文索引是一个非常消耗时间非常消耗硬盘空间的做法。

```
01 -创建表的适合添加全文索引
02 CREATE TABLE `table` (
03   `id` int(11) NOT NULL AUTO_INCREMENT ,
04   `title` char(255) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL ,
05   `content` text CHARACTER SET utf8 COLLATE utf8_general_ci NULL ,
06   `time` int(10) NULL DEFAULT NULL ,
07   PRIMARY KEY (`id`),
08   FULLTEXT (content)
09 );
10 -修改表结构添加全文索引
11 ALTER TABLE article ADD FULLTEXT index_content(content)
12 -直接创建索引
13 CREATE FULLTEXT INDEX index_content ON article(content)
```

4. 单列索引、多列索引

多个单列索引与单个多列索引的查询效果不同，因为执行查询时，MySQL 只能使用一个索引，会从多个索引中选择一个限制最为严格的索引。

5. 组合索引（最左前缀）

平时用的 SQL 查询语句一般都有比较多的限制条件，所以为了进一步榨取 MySQL 的效率，就要考虑建立组合索引。例如上表中针对 title 和 time 建立一个组合索引：ALTER TABLE article ADD INDEX index_time (title(50),time(10))。建立这样的组合索引，其实是相当于分别建立了下面两组组合索引：

-title,time

-title

为什么没有 time 这样的组合索引呢？这是因为 MySQL 组合索引“最左前缀”的结果。简单的理解就是只从最左面的开始组合。并不是只要包含这两列的查询都会用到该组合索引，如下面的几个 SQL 所示：

```
1 -使用到上面的索引
2 SELECT * FROM article WHERE title='测试' AND time=1234567890;
3 SELECT * FROM article WHERE utitle='测试';
4 -不使用上面的索引
5 SELECT * FROM article WHERE time=1234567890;
```

elasticsearch 的倒排索引

elasticsearch 的倒排索引

一个倒排索引包含一系列不同的单词，这些单词出现在任何一个文档，对于每个单词，对应着所有它出现的文档。

比如说，我们有 2 个文档，每个文档有一个 `content` 字段。

内容如下：

“ The quick brown fox jumped over the lazy dog”

“ Quick brown foxes leap over lazy dogs in summer”

为了创建倒排索引，

我们首先对每个字段进行分词，我们称之为 `terms` 或者 `tokens`，创建了一些列有序列表，然后列举了每个单词所出现的文档，结果如下：

Term	Doc_1	Doc_2
Quick		X
The	X	
brown	X	X
dog	X	
dogs		X
fox	X	
foxes		X
in		X
jumped	X	
lazy	X	X
leap		X
over	X	X
quick	X	
summer		X
the	X	

现在，如果我们想搜索 "quick brown"，我们只需要找到每个单词出现的文档。

Term	Doc_1	Doc_2
brown	X	X
quick	X	
Total	2	1

两个文档都匹配，但是第一个文档有更高的匹配度，

如果我们采用一个简单的相似算法，我们可以说，第一个文档比第 2 个文档有更高的匹配度。也更相关。

MySQL 索引的优化

上面都在说使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行 `INSERT`、`UPDATE` 和 `DELETE`。因为更新表时，MySQL 不仅要保存数据，还要保存一下索引文件。建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快。索引只是提高效率的一个因素，如果你的 MySQL 有大

数据量的表，就需要花时间研究建立最优秀的索引，或优化查询语句。下面是一些总结以及收藏的 MySQL 索引的注意事项和优化方法。

1. 何时使用聚集索引或非聚集索引？

聚集索引是指数据库表行中数据的物理顺序与键值的逻辑（索引）顺序相同。

动作描述	使用聚集索引	使用非聚集索引
列经常被分组排序	使用	使用
返回某范围内的数据	使用	不使用
一个或极少不同值	不使用	不使用
小数目的不同值	使用	不使用
大数目的不同值	不使用	使用
频繁更新的列	不使用	使用
外键列	使用	使用
主键列	使用	使用
频繁修改索引列	不使用	使用

事实上，我们可以通过前面聚集索引和非聚集索引的定义的例子来理解上表。如：返回某范围内的数据一项。比如您的某个表有一个时间列，恰好您把聚合索引建立在了该列，这时您查询 2004 年 1 月 1 日至 2004 年 10 月 1 日之间的全部数据时，这个速度就将是很快的，因为您的这本字典正文是按日期进行排序的，聚类索引只需要找到要检索的所有数据中的开头和结尾数据即可；而不像非聚集索引，必须先查到目录中查到每一项数据对应的页码，然后再根据页码查到具体内容。其实这个具体用法我不是很理解，只能等待后期的项目开发中慢慢学学了。

2. 索引不会包含有 **NULL** 值的列

只要列中包含有 **NULL** 值都将不会被包含在索引中，复合索引中只要有一列含有 **NULL** 值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为 **NULL**。

3. 使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个 CHAR(255) 的列，如果在前 10 个或 20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。**短索引**不仅可以提高查询速度而且可以节省磁盘空间和 I/O 操作。

4. 索引列排序

MySQL 查询只使用一个索引，因此如果 `where` 子句中已经使用了索引的话，那么 `order by` 中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

5. like 语句操作

一般情况下不鼓励使用 `like` 操作，如果非使用不可，如何使用也是一个问题。`like "%aaa%"` 不会使用索引而 `like "aaa%"` 可以使用索引。

6. 不要在列上进行运算

例如：`select * from users where YEAR(adddate)<2007`，将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成：`select * from users where adddate<'2007-01-01'`。关于这一点可以围观：[一个单引号引发的 MySQL 性能损失](#)。

很明显，**不使用单引号没有用上主索引**，并进行了全表扫描，使用单引号就能使用上索引了。

最后总结一下，MySQL 只对一下操作符才使用索引：`<`, `<=`, `=`, `>`, `>=`, `between`, `in`, 以及某些时候的 `like`(不以通配符%或_开头的情形)。而理论上每张表里面最多可创建 16 个索引，不过除非是数据量真的很多，否则过多的使用索引也不是那么好玩的，比如我刚才针对 `text` 类型的字段创建索引的时候，系统差点就卡死了。

4 数据库事务的隔离级别

数据库事务的隔离级别有 4 个，由低到高依次为 `Read uncommitted`、`Read committed`、`Repeatable read`、`Serializable`（可串行化的），这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

√: 可能出现 ×: 不会出现

	脏读	不可重复读	幻读
Read uncommitted (大多数默认)	√	√	√
Read committed	×	√	√
Repeatable read (MySQL 默认)	×	×	√
Serializable	×	×	×

5 数据库回滚

rollback 回滚的意思。就是数据库里做修改后（update ,insert , delete)未 commit 之前 使用 rollback 可以恢复数据到修改之前。

6 explain 命令 如何查看 mysql 使用索引来执行 select 语句

对 mysql explain 讲的比较清楚的

在 explain 的帮助下，您就知道什么时候该给表添加索引，以使用索引来查找记录从而让 select 运行更快。

如果由于不恰当使用索引而引起一些问题的话，可以运行 analyze table 来更新该表的统计信息，例如键的基数，它能帮您在优化方面做出更好的选择。

explain 返回了一行记录，它包括了 select 语句中用到的各个表的信息。这些表在结果中按照 mysql 即将执行的查询中读取的顺序列出来。mysql 用一次扫描多次连接（single-sweep,multi-join）的方法来解决连接。这意味着 mysql 从第一个表中读取一条记录，然后在第二个表中查找到对应的记录，然后在第三个表中查找，依次类推。当所有的表都扫描完了，它输出选择的字段并且回溯所有的表，直到找不到为止，因为有的表中可能有多条匹配的记录下一条记录将从该表读取，再从下一个表开始继续处理。

在 mysql version 4.1 中，explain 输出的结果格式改变了，使得它更适合例如 union 语句、子查询以及派生表的结构。更令人注意的是，它新增了 2 个字段： id 和 select_type。当你使用早于 mysql4.1 的版本就看不到这些字段了。

7 show profiles 查询 mysql 语句运行时间

默认是 OFF 的状态

```
mysql> show variables like "%pro%";  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| profiling | OFF |  
| profiling_history_size | 15 |  
| protocol_version | 10 |  
| slave_compressed_protocol | OFF |  
+-----+-----+  
4 rows in set (0.00 sec)  
  
mysql> set profiling = 1;  
Query OK, 0 rows affected (0.00 sec)
```

show profiles; 即可查看所有 sql 的总的执行时间。

```
mysql> show profiles;  
+-----+-----+-----+  
| Query_ID | Duration | Query |  
+-----+-----+-----+  
| 1 | 0.00070100 | show variables like "%pro%" |  
| 2 | 0.00017700 | select 1 |  
| 3 | 0.00038000 | select 1 from seven_user |  
| 4 | 0.00036700 | select uid from seven_user |  
| 5 | 0.00011900 | select * from seven_user |  
+-----+-----+-----+  
5 rows in set (0.00 sec)
```

show profile for query 1 即可查看第 1 个 sql 语句的执行的各个操作的耗时详情。

```
mysql> show profile for query 1;
+-----+-----+
| Status          | Duration |
+-----+-----+
| starting        | 0.000051 |
| Opening tables  | 0.000041 |
| System lock     | 0.000010 |
| Table lock      | 0.000012 |
| init            | 0.000012 |
| optimizing      | 0.000010 |
| statistics      | 0.000013 |
| preparing        | 0.000014 |
| executing       | 0.000419 |
| Sending data    | 0.000024 |
| end             | 0.000011 |
| query end       | 0.000009 |
| freeing items   | 0.000026 |
| removing tmp table | 0.000020 |
| closing tables  | 0.000010 |
| logging slow query | 0.000009 |
| cleaning up     | 0.000010 |
+-----+-----+
17 rows in set (0.00 sec)
```

show profile cpu, block io, memory, swaps, context switches, source for query 6; 可以查看出一条 SQL 语句执行的各种资源消耗情况，比如 CPU, IO 等

show profile all for query 6 查看第 6 条语句的所有执行信息。

测试完毕后，关闭参数：

```
mysql> set profiling=0
```

8 大规模的 delete 或 insert 操作是否会对引起表锁定，如何解决

分批次 delete 或 insert

9 怎么在 SQL 层面做一些优化

sql 层面优化：尽量不要用*, in 和 not in 等

数据库层面优化：走索引，建分区表。

SQL 优化基本就是一些常识性的东西，比如不能用 SELECT *，少用 DISTINCT、GROUP BY 之类的命令。

SQL 优化步骤

1. 使用 pt-query-digest 分析慢日志，拿到调用频率高并且执行时间长的语句。
2. 获取语句相关表的信息，以及 explain 等相关信息，一般收集如下信息
 - | 执行计划 : explain query;
 - | 详细执行计划 : explain extended query; show warnings;
 - | 相关表信息 : show create table test1;
 - | 语句执行开销 : show profile for query num;
 - | 数据量 : select count(0) from table;

- | 相关表的索引信息 : show index for table;
- | my.cnf 查询相关参数如各类 buffer 等

3. 进行初步分析，诊断 SQL 语句，**查看在 SQL 层面能否带来提升**，一般应对手段如下

- 条件上拉、下推
- 子查询合并、展开
- 添加索引、优化索引
- distinct、group by、orderby 尽量利用索引减少文件排序
- 子查询尽量转换为 join，最好消除
- 尽量减少聚合函数，将复杂查询转换为多表链接
- 尽量优化等价谓词，尽可能利用索引
- join 的一些优化，连接消除、去除中间表等

4. SQL 无优化空间考虑从业务逻辑层面入手。常用手段如下

- | 增加冗余字段
- | 拆分表（水平/竖直）
- | 使用分布式
- | 将大事务拆分为多个小事务
- | 精简数据库的设计，减少使用存储过程、触发器等

5. 对硬件进行优化，升级服务器硬件。

3、4 步骤可与调整系统参数相结合，具体使用方法视具体业务而定。

10 MySQL 性能优化——易实现的 MySQL 优化方案 汇总

一、索引优化

- 1、合理使用索引，在经常查询而不经常增删改操作的字段加索引，一个表上的索引不应该超过 6 个。
- 2、`Order by` 与 `group by` 后应直接使用字段，而且字段应该是索引字段。
- 3、索引字段长度应较短而长度固定。
- 4、索引字段重复不能过多。

5、Hash 索引与 BTREE 索引区别 (MyISAM 与 InnoDB 不支持 Hash 索引)

(1)、BTREE 索引使用多路搜索树的**数据结构**，可以减少定位的中间过程；综合效率较高，默认使用的索引。

(2)、Hash 索引使用 Hash **算法**构建索引；精确的等值查询一次定位，效率极高，但特别不适合范围查询；使用 Hash 的复合索引是把复合索引键共同计算 hash 值，故不能单独使用。

6、会导致引擎放弃使用索引，改为进行全表的几种情况，都要在开发中尽量避免出现！

(1)、`where` 子句中使用 `like` 关键字时，前置百分号会导致索引失效（起始字符不确定都会失效）。如：`select id from test where name like "%吉坤"。`

(2)、`where` 子句中使用 `is null` 或 `is not null` 时，因为 `null` 值会被自动从索引中排除，索引一般不会建立在有空值的列上。

(3)、`where` 子句中使用 `or` 关键字时，`or` 左右字段如果存在一个没有索引，有索引字段也会失效；而且即使都有索引，因为二者的索引存储顺序并不一致，效率还不如顺序全表扫描，这时引擎有可能放弃使用索引，所以要慎用 `or`。

(4)、where 子句中使用 in 或 not in 关键字时，会导致全表扫描，能使用 exists 或 between and 替代就不使用 in。

(5)、where 子句中使用!=操作符时，将放弃使用索引，因为范围不确定，使用索引效率不高，会被引擎自动改为全表扫描；

(6)、where 子句中应尽量避免对索引字段操作（表达式操作或函数操作），比如 select id from test where num/2 = 100 应改为 num = 200。

(7)、在使用复合索引时，查询时必须使用到索引的第一个字段，否则索引失效；并且应尽量让字段顺序与索引顺序一致。

(8)、查询时必须使用正确的数据类型。数据库包含了自动了类型转换，比如纯数字赋值给字符串字段时可以被自动转换，但如果查询时不加引号查询，会导致引擎忽略索引。

二、表结构优化

1、设计符合第三范式的表结构。

2、尽量使用数字型字段，提高数据比对效率。

3、对定长、MD5 哈希码、长度较短的字段使用 char 类型，提高效率；对边长而且可能较长字段使用 varchar 类型，节约内存。

4、适当的进行水平分割与垂直分割，比如当表列数过多时，就将一部分列移出到另一张表中。

关于水平分割与垂直分割表详解：

水平分割表：一种是当多个过程频繁访问数据表的不同行时，水平分割表，并消除新表中的冗余数据列；若个别过程要访问整个数据，则要用连接*作，这也无妨分割表；典型案例是

电话话单按月分割存放。另一种是当主要过程要重复访问部分行时，最好将被重复访问的这些行单独形成子集表（冗余储存），这在不考虑磁盘空间开销时显得十分重要；但在分割表以后，增加了维护难度，要用触发器立即更新、或存储过程或应用代码批量更新，这也会增加额外的磁盘 I/O 开销。

水平分割会给应用增加复杂度，它通常在查询时需要多个表名，查询所有数据需要 union 操作。在许多数据库应用中，这种复杂性会超过它带来的优点，因为只要索引关键字不大，则在索引用于查询时，表中增加两到三倍数据量，查询时也就增加读一个索引层的磁盘次数。

垂直分割表（不破坏第三范式）：一种是当多个过程频繁访问表的不同列时，可将表垂直分成几个表，减少磁盘 I/O（每行的数据列少，每页存的数据行就多，相应占用的页就少），更新时不必考虑锁，没有冗余数据。缺点是要在插入或删除数据时要考虑数据的完整性，用存储过程维护。另一种是当主要过程反复访问部分列时，最好将这部分被频繁访问的列数据单独存为一个子集表（冗余储存），这在不考虑磁盘空间开销时显得十分重要；但这增加了重叠列的维护难度，要用触发器立即更新、或存储过程或应用代码批量更新，这也会增加额外的磁盘 I/O 开销。垂直分割表可以达到最大化利用 Cache 的目的。

垂直分割可以使得数据行变小（因为列少了，一行数据就变小），一个数据页就能存放更多的数据，在查询时就会减少 I/O 次数。其缺点是需要管理冗余列，查询所有数据需要 join 操作。

三、临时表优化——临时表常常用于排序或分组，所以 Order By 与 Group By 后的字段尽量使用索引

临时表可以根据实际需求使用，但要尽力避免磁盘临时表的生成。

1、常见的会产生内存临时表的情况

- 1、UNION 查询。
- 2、子查询（所以我们一般用 join 代替子查询）。
- 3、join 查询中，如果 order by 与 group by 如果使用的不都是第一张表上的字段，就会产生临时表。
- 4、order by 中使用 distinct 函数。

2、常见的会产生磁盘临时表的情况

- 1、数据表中包含 BLOB/TEXT 列。
- 2、Group by、distinct、union 查询中包含超过 512 字节的列。

四、其他优化

- 1、不使用 Select *，只查询需要的字段。
- 2、在只查询一条字段时，limit 1。
- 3、避免大事务操作，提高并发能力。
- 4、在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON，在结束时设置 SET NOCOUNT OFF。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE_IN_PROC 消息。
- 5、尽量少使用游标。
- 6、多去关注慢查询，总有我们提前考虑不到的问题，出现了就去解决它！

附慢查询开启方式：

在 MySQL 安装目录下，找到 my.ini 配置文件，在 mysqld 下加上如下配置：

```
log-slow-queries = D:/Mysql/mysql-5.6.27-winx64/slowquery.log  
long_query_time = 2
```

```
[mysql]
# 设置mysql客户端默认字符集
default-character-set=utf8
[mysqld]
#设置3306端口
port = 3306
# 设置mysql的安装目录
basedir=D:\Mysql\mysql-5.6.27-winx64
# 设置mysql数据库的数据的存放目录
datadir=D:\Mysql\mysql-5.6.27-winx64\data
# 允许最大连接数
max_connections=200
# 服务端使用的字符集默认为8比特编码的latin1字符集
character-set-server=utf8
# 创建新表时将使用的默认存储引擎
default-storage-engine=INNODB
skip-grant-tables
character-set-server=utf8
# 慢查询设置
log-slow-queries = D:/Mysql/mysql-5.6.27-winx64/slowquery.log
long_query_time = 2
```

11 三范式

第一：列满足原子性，即求每一列都不允许再次拆分（**列不可分**）

如：表中若有“地址”列。则地址还可以拆分为：国家、省份、城市等这些列，就说明地址这列还可拆分，则不满足第三范式

第二：满足第一的基础上，除主键以外每一列都依赖于主键（**不能有部分依赖**）

如：一张表是描述学员的；有学员编号，学校名称；其中学校名称和学员编号没有依赖关系。应把学校名称放在学校表中

第三：满足第一第二的基础上，除主键以外的列都直接依赖主键。（**不能有传递依赖**）

如：城镇表依赖市表，市表依赖省表，则可以推断出城镇依赖省表；现在城镇表和省表之间就是间接相关

12 select for update

MySQL SELECT ... FOR UPDATE 的 Row Lock 与 Table Lock

上面介绍过 SELECT ... FOR UPDATE 的用法，不过锁定(Lock)的数据是判别就得要注意一下了。由于 InnoDB 预设是 Row-Level Lock，所以只有「明确」的指定主键，MySQL 才会执行 Row lock (只锁住被选取的数据)，否则 MySQL 将会执行 Table Lock (将整个数据表单给锁住)。

举个例子：

假设有个表单 products ，里面有 id 跟 name 二个栏位，id 是主键。

例 1：(明确指定主键，并且有此数据，row lock)

```
SELECT * FROM products WHERE id='3' FOR UPDATE;
```

例 2: (明确指定主键, 若查无此数据, 无 lock)

```
SELECT * FROM products WHERE id=-1 FOR UPDATE;
```

例 2: (无主键, table lock)

```
SELECT * FROM products WHERE name='Mouse' FOR UPDATE;
```

例 3: (主键不明确, table lock)

```
SELECT * FROM products WHERE id<>'3' FOR UPDATE;
```

例 4: (主键不明确, table lock)

```
SELECT * FROM products WHERE id LIKE '3' FOR UPDATE;
```

乐观锁和悲观锁策略

悲观锁: 在读取数据时锁住那几行, 其他对这几行的更新需要等到悲观锁结束时才能继续。

乐观锁: 读取数据时不锁, 更新时检查是否数据已经被更新过, 如果是则取消当前更新, 一般在悲观锁的等待时间过长而不能接受时我们才会选择乐观锁。

13 MySQL 创建用户与授权及撤销用户权限方法

一. 创建用户

命令:CREATE USER 'username'@'host' IDENTIFIED BY 'password';

二. 授权

命令:GRANT privileges ON databasename.tablename TO 'username'@'host'

三. 设置与更改用户密码

命令:SET PASSWORD FOR 'username'@'host' = PASSWORD('newpassword');如果是

当前登陆用户用 SET PASSWORD = PASSWORD("newpassword");

四. 撤销用户权限

命令: REVOKE privilege ON databasename.tablename FROM 'username'@'host';

五. 删除用户

命令: DROP USER 'username'@'host';

14 prepareStatement 与 Statement 的区别

1. 区别：（用法）

```
stmt=conn.createStatement();
resultSet rs=stmt.executeQuery(sql);
```

上面是 statement 的用法

=====

下面是 PreparedStatement 的用法

```
ptmt=conn.PreparedStatement(sql);
resultSet rs=ptmt.executeQuery();
```

=====

Statement 是 PreparedStatement 的父类

还有就是 sql 放置的位置不同。

在开发中一般用 PreparedStatement

```
stmt.executeUpdate("insert      into      tb_name      (col1,col2,col2,col4)      values
('"+var1+"','"+var2+"','"+var3+"','"+var4+"')");
```

```
perstmt = con.prepareStatement("insert into tb_name (col1,col2,col2,col4) values (?,?,?,?,?)");
perstmt.setString(1,var1);
perstmt.setString(2,var2);
perstmt.setString(3,var3);
perstmt.setString(4,var4);
perstmt.executeUpdate();
```

二. PreparedStatement 尽最大可能提高性能.

当然并不是所以预编译语句都一定会被缓存,数据库本身会用一种策略,比如使用频度等因素来决定什么时候不再缓存已有的预编译结果.以保存有更多的空间存储新的预编译语句.

三. 最重要的一点是极大地提高了安全性.

15 数据库的索引是如何实现的，主键索引和联合索引数据结构有什么区别

主键是表中的一个或多个字段,它的值用于唯一地标识表中的某一条记录.且不能为空; 索引是对数据库表中一列或多列的值进行排序的一种结构, 只有当经常查询索引列中的数据时, 才需要在表上创建索引, 使用索引可快速访问数据库表中的特定信息。 索引占用磁盘空间, 并且降低添加、删除和更新行的速度。当然索引也有好处就是查询速度快, 它利还是大于弊的所以请慎重使用索引。 比如: 一个学生表 (t_stu) 有 1000 条数据, 给它 id 列建个主键和索引, 你想查询 id=1000; 的这条信息, 如果没有索引, 它就一条一条的比对查找, 系统运行 1000 次才找到, 要是创建了索引, 你查询 id=1000 的这条信息, 系统只运行一次就找到了。

索引是 b+树 主键索引可能是聚集索引 多列索引的话还是一棵树
只是如果第一列相同的话, 就根据第二列的值来排序

`where` 后面的条件 顺序影响执行效率吗

如果 `where` 后面的条件次序跟多列索引的次序不同，就不能充分利用多列索因或者比方说有两个单列索引，一个表有 `a,b,c` 这三列，`b` 列和 `c` 列有单列索引，然后其中 `b=1` 的记录有 100 条，`c=1` 的记录有 1000000 条，但是 `b=1&&c=1` 的记录只有一条。现在我们 `select * from table where b=1 and c = 1`，用哪个索引，就比较讲究。

16 数据库连接池

1 dbcp

`dbcp` 可能是使用最多的开源连接池，原因大概是因为配置方便，而且很多开源和 `tomcat` 应用例子都是使用的这个连接池吧。这个连接池可以设置最大和最小连接，连接等待时间等，基本功能都有。使用评价：在具体项目应用中，发现此连接池的持续运行的稳定性还是可以，不过速度稍慢，在大并发量的压力下稳定性有所下降，此外不提供连接池监控。

2 c3p0

`c3p0` 是另外一个开源的连接池，在业界也是比较有名的，这个连接池可以设置最大和最小连接，连接等待时间等，基本功能都有。使用评价：在具体项目应用中，发现此连接池的持续运行的稳定性相当不错，在大并发量的压力下稳定性也有一定保证，此外不提供连接池监控。

3 proxool

`proxool` 这个连接池可能用到的人比较少，但也有一定知名度，这个连接池可以设置最大和最小连接，连接等待时间等，基本功能都有。使用评价：在具体项目应用中，发现此连接池的持续运行的稳定性有一定问题，有一个需要长时间跑批的任务场景任务，同样的代码

17 MySQL 中的锁概念

MySQL 中不同的存储引擎支持不同的锁机制。比如 `MyISAM` 和 `MEMORY` 存储引擎采用的表级锁，`BDB` 采用的是页面锁，也支持表级锁，`InnoDB` 存储引擎既支持行级锁，也支持表级锁，默认情况下采用行级锁。

Mysql3 中锁特性如下：

表级锁：开销小，加锁块；不会出现死锁，锁定粒度大，发生锁冲突的概率最高，并发度最低。

行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发性也最高。

页面锁：开销和加锁界于表锁和行锁之间，会出现死锁；锁定粒度界与表锁和行锁之间，并发一般。

18 B+树在数据库中的应用

主要是在所有的叶子结点中增加了指向下一个叶子节点的指针，因此 `InnoDB` 建议为大部分表使用默认自增的主键作为主索引。

为什么使用 B+树？言简意赅，就是因为：

1.B+树空间利用率更高，可减少 I/O 次数

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘 I/O 消耗。而因为 B+树的内部节点只是作为索引使用，而不像 B-树那样每个节点都需要存储硬盘指针。

也就是说：B+树中每个非叶节点没有指向某个关键字具体信息的指针，所以每一个节点可以存放更多的关键字数量，即一次性读入内存所需要查找的关键字也就越多，减少了 I/O 操作。

2. 增删文件（节点）时，效率更高

因为 B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。

3. B+树的查询效率更加稳定

因为 B+树的每次查询过程中，都需要遍历从根节点到叶子节点的某条路径。所有关键字的查询路径长度相同，导致每一次查询的效率相当。

19 如何优化 MySQL

我当时是按以下四条依次回答的，他们四条从效果上第一条影响最大，后面越来越小。

- ① SQL 语句及索引的优化
- ② 数据库表结构的优化
- ③ 系统配置的优化
- ④ 硬件的优化

1、选取最适用的字段属性

字段设置尽可能的小，比如存手机号，邮箱这种固定长度的，就没必要很大的开销。字段不要设置为 NULL，第一在程序中就不需要判断空，第二如果为空不能使用索引。如省份性别这样的信息我们使用枚举类型，因为 MySQL 把枚举当数值型数据处理，比文本快很多。

2、使用连接（JOIN）来代替子查询(Sub-Queries)

举个例子我们要取出客户基本信息表中没有订单记录的用户来

```
SELECT * FROM customerinfo WHERE CustomerID NOT in (SELECT CustomerID FROM salesinfo )
SELECT * FROM customerinfo LEFT JOIN salesinfo ON customerinfo.CustomerID=salesinfo.CustomerID WHERE salesinfo.CustomerID IS NULL
```

连接（JOIN）之所以更有效率一些，是因为 MySQL 不需要在内存中创建临时表来完成这个逻辑上的需要两个步骤的查询工作。

3、使用联合(UNION)来代替手动创建的临时表

使用 UNION 来创建查询的时候，我们只需要用 UNION 作为关键字把多个 SELECT 语句连接起来就可以了，要注意的是所有 SELECT 语句中的字段数目要想同。下面的例子就演示了一个使用 UNION 的查询。

```
SELECT Name, Phone FROM client UNION SELECT Name, BirthDate FROM author
```

```
UNION
```

```
SELECT Name, Supplier FROM product
```

4、事务

事务的 ACID 性质很好，事务的另一个重要作用是当多个用户同时使用相同的数据源时，它可以利用锁定数据库的方法来为用户提供一种安全的访问方式，这样可以保证用户的操作不被其它的用户所干扰。

7、使用索引

一般说来，索引应建立在那些将用于 JOIN, WHERE 判断和 ORDER BY 排序的字段上。尽量不要对数据库中某个含有大量重复的值的字段建立索引。比如枚举类型有可能会出现大量重复。

8、优化的查询语句

例如，在一个 DATE 类型的字段上使用 YEAR() 函数时，将会使索引不能发挥应有的作用。所以，下面的两个查询虽然返回的结果一样，但后者要比前者快得多。

```
SELECT * FROM order WHERE YEAR(OrderDate)<2001;
```

```
SELECT * FROM order WHERE OrderDate<"2001-01-01";
```

同样的情形也会发生在对数值型字段进行计算的时候：

```
SELECT * FROM inventory WHERE Amount/7<24;
```

```
SELECT * FROM inventory WHERE Amount<24*7;
```

20 什么情况下设置了索引但无法使用

- ① 以“%”开头的 LIKE 语句，模糊匹配
- ② OR 语句前后没有同时使用索引
- ③ 数据类型出现隐式转化（如 varchar 不加单引号的话可能会自动转换为 int 型）

21 SQL 语句的优化

order by 要怎么处理

alter 尽量将多次合并为一次

insert 和 delete 也需要合并

22 InnoDB 索引和 MyISAM 索引的区别 (*)

MyISAM 与 InnoDB 的区别是什么？

1、存储结构

MyISAM：每个 MyISAM 在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm 文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。

InnoDB: 所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB 表的大小只受限于操作系统文件的大小，一般为 2GB。

2、存储空间

MyISAM: 可被压缩，存储空间较小。支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。

InnoDB: 需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

3、可移植性、备份及恢复

MyISAM: 数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。

InnoDB: 免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十 G 的时候就相对痛苦了。

4、事务支持

MyISAM: 强调的是性能，每次查询具有原子性，其执行速度比 InnoDB 类型更快，但是不提供事务支持。

InnoDB: 提供事务支持事务，外部键等高级数据库功能。具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。

5、AUTO_INCREMENT

MyISAM: 可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。

InnoDB: InnoDB 中必须包含只有该字段的索引。引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

6、表锁差异

MyISAM: 只支持表级锁，用户在操作 myisam 表时，select, update, delete, insert 语句都会给表自动加锁，如果加锁以后的表满足 insert 并发的情况下，可以在表的尾部插入新的数据。

InnoDB: 支持事务和行级锁，是 innodb 的最大特色。行锁大幅度提高了多用户并发操作的新能。但是 InnoDB 的行锁，只是在 WHERE 的主键是有效的，非主键的 WHERE 都会锁全表的。

7、全文索引

MyISAM: 支持 FULLTEXT 类型的全文索引

InnoDB: 不支持 FULLTEXT 类型的全文索引，但是 innodb 可以使用 sphinx 插件支持全文索引，并且效果更好。

8、表主键

MyISAM: 允许没有任何索引和主键的表存在，索引都是保存行的地址。

InnoDB: 如果没有设定主键或者非空唯一索引，就会自动生成一个 6 字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。

9、表的具体行数

MyISAM: 保存有表的总行数，如果 `select count(*) from table;` 会直接取出该值。

InnoDB: 没有保存表的总行数，如果使用 `select count(*) from table;` 就会遍历整个表，消耗相当大，但是在加了 `where` 条件后，`myisam` 和 `innodb` 处理的方式都一样。

10、CURD 操作

MyISAM: 如果执行大量的 SELECT，MyISAM 是更好的选择。

InnoDB: 如果你的数据执行大量的 INSERT 或 UPDATE，出于性能方面的考虑，应该使用 InnoDB 表。DELETE 从性能上 InnoDB 更优，但 `DELETE FROM table` 时，InnoDB 不会重新建立表，而是一行一行的删除，在 innodb 上如果要清空保存有大量数据的表，最好使用 `truncate table` 这个命令。

11、外键

MyISAM: 不支持

InnoDB: 支持

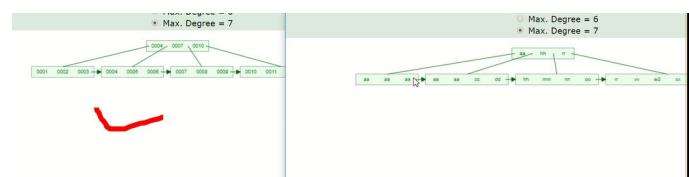
通过上述的分析，基本上可以考虑使用 InnoDB 来替代 MyISAM 引擎了，原因是 InnoDB 自身很多良好的特点，比如**事务支持、存储过程、视图、行级锁定**等等，在并发很多的情况下，相信 InnoDB 的表现肯定要比 MyISAM 强很多。另外，任何一种表都不是万能的，只用恰当的针对业务类型来选择合适的表类型，才能最大的发挥 MySQL 的性能优势。如果不是很复杂的 Web 应用，非关键应用，**做很多 count 计算，插入也不是很频繁，select 比较频繁**还是可以继续考虑 MyISAM 的。

23 数据库设计时，字段可以选择默认值或者 null，为何选择默认值

`null` 值在有些存储引擎中按最大字节数存储，占空间，`null` 不能被索引，影响性能；

24 数据库设计时，为什么要用自增 id 做为主键

插入维度：B+树插入的时候很有顺序，在最后插入，申请的空间也几乎是连续的



查找维度：字符串比较肯定比数值慢

11 操作系统 Linux

Linux

1 进程间通信的方式有哪几种？管道符“|”将两个命令隔开

管道 利用 Linux 所提供的管道符 “|” 将两个命令隔开，管道符左边命令的输出就会作为管道符右边命令的输入。

1) 管道 2) 有名管道 3) 信号量 4) 消息队列 5) 共享内存；6) 套接字 socket

2 Linux 下如何进行进程调度的？

主要有三种调度策略：

1. 分时调度策略
2. 实时调度策略—FIFO
3. 实时调度策略—时间片轮转。

3 Linux 交换空间 物理内存和虚拟内存

我们知道，直接从物理内存读写数据要比从硬盘读写数据要快的多，因此，我们希望所有数据的读取和写入都在内存完成，而内存是有限的，这样就引出了物理内存与虚拟内存的概念。

物理内存就是系统硬件提供的内存大小，是真正的内存，相对于物理内存，在 linux 下还有一个虚拟内存的概念，虚拟内存就是为了满足物理内存的不足而提出的策略，它是利用磁盘空间虚拟出的一块逻辑内存，用作虚拟内存的磁盘空间被称为交换空间（Swap Space）。

作为物理内存的扩展，linux 会在物理内存不足时，使用交换分区的虚拟内存，更详细的说，就是内核会将暂时不用的内存块信息写到交换空间，这样以来，物理内存得到了释放，这块内存就可以用于其它目的，当需要用到原始的内容时，这些信息会被重新从交换空间读入物理内存。

Linux 的内存管理采取的是分页存取机制，为了保证物理内存能得到充分的利用，内核会在适当的时候将物理内存中不经常使用的数据块自动交换到虚拟内存中，而将经常使用的信息保留到物理内存。

4 内存的监控

top 命令能够实时显示系统中各个进程的资源占用状况，类似于 Windows 下的资源管理器。监控内存最常使用的命令有 free、top 等，下面是某个系统 top 的输出：

top - 16:58:19 up 111 days, 1:54, 2 users, load average: 1.29, 1.09, 1.07												
Tasks: 158 total, 3 running, 155 sleeping, 0 stopped, 0 zombie												
%Cpu(s): 50.2 us, 0.0 sy, 0.0 ni, 49.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st												
KiB Mem: 4046856 total, 2400324 used, 1646532 free, 263456 buffers												
KiB Swap: 0 total, 0 used, 0 free. 1190568 cached Mem												
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND	
27746	root	20	0	233824	7440	1068	S	100.1	0.2	10165:29	apache	
8	root	20	0	0	0	0	R	0.3	0.0	44:39.14	rcuos/0	
1	root	20	0	33624	2908	1492	S	0.0	0.1	0:04.49	init	
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd	
3	root	20	0	0	0	0	S	0.0	0.0	0:00.77	ksoftirqd/0	
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H	
7	root	20	0	0	0	0	S	0.0	0.0	33:39.13	rcu_sched	
9	root	20	0	0	0	0	S	0.0	0.0	31:40.85	rcuos/1	
10	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/2	
11	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/3	
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/4	
13	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/5	
14	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/6	
15	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/7	
16	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/8	
17	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/9	
18	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/10	
19	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/11	
20	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/12	
21	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/13	
22	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuos/14	

每个选项的含义：

第一行：任务队列信息

当前时间，系统运行时间，登录用户数，系统负载，即任务队列的平均长度。三个数值分别为 1 分钟、5 分钟、15 分钟前到现在的平均值。

第二、三行为进程和 CPU 的信息

当有多个 CPU 时，这些内容可能会超过两行

第二行：Task

第三行：用户空间占用 CPU 百分比

第四行 Mem：代表物理内存使用情况

第五行：(buffers/cached)：代表磁盘缓存使用状态 Swap 表示交换空间内存使用状态

free 命令输出的内存状态，可以通过两个角度来查看：一个是从内核的角度来看，一个是从应用层的角度来看的

```
root@iZwz9bruzaghvihvjjbrsklZ:~# free
              total        used        free      shared  buffers   cached
Mem:       4046856     2399828     1647028          536    263456   1190568
-/+ buffers/cache:  945804   3101052
Swap:          0          0          0
root@iZwz9bruzaghvihvjjbrsklZ:~#
```

5 Linux 常用命令

- 1) 文件目录相关：ls、cd、pwd、mkdir、rmdir、mv、cp、touch、cat、
 - 2) 文件查找类型的：which、find
 - 3) 压缩和解压：tar、unzip、
 - 4) 文件权限：chmod
 - 5) 进程：ps、kill
 - 6) 网络：ssh、ping、ipconfig、netstat、scp
- 7) **Linux** 获取上一条命令的返回值。一般情况下，命令执行正确，返回 0，命令异常，返回其他值。
- 命令：echo \$?
- 8) **Windows** 获取上一条命令的返回值。一般情况下，命令执行正确，返回 0，命令异常，

返回其他值。

命令: echo %ERRORLEVEL%

9)

chmod u+rwx file 为 file 添加或取消所属用户的权限

(u 代表所属用户 o 代表其他用户 g 代表所属组的成员用户 a 表示所有人)

chmod 567 file 用数字也可以表示权限

5 → 101 → r-x

6 Linux 下查看网络端口状态

netstat 命令: 用于显示与 IP、TCP、UDP 和 ICMP 协议相关的统计数据，一般用于检验本机各端口的网络连接情况。netstat 是在内核中访问网络及相关信息的程序，它能提供 TCP 连接，TCP 和 UDP 监听，进程内存管理的相关报告。**LISTEN** 和 **ESTABLISHED**

7 Linux shell 定期删除七天之前日志文件

1. 删除文件命令:

```
find 对应目录 -mtime +天数 -name "文件名" -exec rm -rf {} \;
```

实例命令:

```
find /opt/soft/log/ -mtime +30 -name "*.log" -exec rm -rf {} \;
```

说明:

将 /opt/soft/log/ 目录下所有 30 天前带 ".log" 的文件删除。具体参数说明如下:

find: linux 的查找命令，用户查找指定条件的文件；

/opt/soft/log/: 想要进行清理的任意目录；

-mtime: 标准语句写法；

+30: 查找 30 天前的文件，这里用数字代表天数；

"*.log": 希望查找的数据类型，".jpg" 表示查找扩展名为 jpg 的所有文件，"*" 表示查找所有文件，这个可以灵活运用，举一反三；

-exec: 固定写法；

rm -rf: 强制删除文件，包括目录；

{ } \; : 固定写法，一对大括号+空格+\+；

2. 计划任务: Linux crontab 定时执行任务 命令格式与详细例子

若嫌每次手动执行语句太麻烦，可以将这小语句写到一个可执行 shell 脚本文件中，再设置 cron 调度执行，那就可以让系统自动去清理相关文件。

```
* * * * * command
```

分 时 日 月 周 命令

第 1 列表示分钟 1~59 每分钟用*或者 */1 表示

第 2 列表示小时 1~23 (0 表示 0 点)

第 3 列表示日期 1~31

第 4 列表示月份 1~12

第 5 列标识号星期 0~6 (0 表示星期天)

第 6 列要运行的命令

crontab 文件的一些例子:

30 21 * * * /usr/local/etc/rc.d/lighttpd restart

上面的例子表示每晚的 21:30 重启 apache。

45 4 1,10,22 * * /usr/local/etc/rc.d/lighttpd restart

上面的例子表示每月 1、10、22 日的 4 : 45 重启 apache。

10 1 * * 6,0 /usr/local/etc/rc.d/lighttpd restart

上面的例子表示每周六、周日的 1 : 10 重启 apache。

0,30 18-23 * * * /usr/local/etc/rc.d/lighttpd restart

上面的例子表示在每天 18 : 00 至 23 : 00 之间每隔 30 分钟重启 apache。

0 23 * * 6 /usr/local/etc/rc.d/lighttpd restart

上面的例子表示每星期六的 11 : 00 pm 重启 apache。

0 */1 * * * /usr/local/etc/rc.d/lighttpd restart

每一小时重启 apache

0 23-7/1 * * * /usr/local/etc/rc.d/lighttpd restart

晚上 11 点到早上 7 点之间，每隔一小时重启 apache

0 11 4 * mon-wed /usr/local/etc/rc.d/lighttpd restart

每月的 4 号与每周一到周三的 11 点重启 apache

0 4 1 jan * /usr/local/etc/rc.d/lighttpd restart

一月一号的 4 点重启 apache

<http://www.cnblogs.com/peida/archive/2013/03/25/2980121.html>

3. 查找 Linux 系统中的占用磁盘空间最大的前 10 个文件或文件夹

- du : 计算出单个文件或者文件夹的磁盘空间占用.
- sort : 对文件行或者标准输出行记录排序后输出.
- head : 输出文件内容的前面部分.

```
1 # du -a /var | sort -n -r | head -n 10
```

4.grep 在文件中查找指定的内容，并输出行号

```
root@iZwz9bruzaqhhivjjbrsklZ:/usr/local/redis# head -20 redis.conf | nl -ba | tail -10 | grep -iE 'bytes'
 11  # 1k => 1000 bytes
 12  # 1kb => 1024 bytes
 13  # 1m => 1000000 bytes
 14  # 1mb => 1024*1024 bytes
 15  # 1g => 1000000000 bytes
 16  # 1gb => 1024*1024*1024 bytes
 17  #
 18  # units are case insensitive so 1GB 1Gb 1gB are all the same.
 19
 20 ##### INCLUDES #####
root@iZwz9bruzaqhhivjjbrsklZ:/usr/local/redis# head -20 redis.conf | nl -ba | tail -10 | grep -iE 'bytes'
 11  # 1k => 1000 bytes
 12  # 1kb => 1024 bytes
 13  # 1m => 1000000 bytes
 14  # 1mb => 1024*1024 bytes
 15  # 1g => 1000000000 bytes
 16  # 1gb => 1024*1024*1024 bytes
root@iZwz9bruzaqhhivjjbrsklZ:/usr/local/redis#
```

```
root@iZwz9bruzaqhhivjjbrsklZ:/usr/local/redis# grep -n bytes redis.conf
11:# 1k => 1000 bytes
12:# 1kb => 1024 bytes
13:# 1m => 1000000 bytes
14:# 1mb => 1024*1024 bytes
15:# 1g => 1000000000 bytes
16:# 1gb => 1024*1024*1024 bytes
519:# Don't use more memory than the specified amount of bytes.
542:# maxmemory <bytes>
692:# Redis will try to read more data from the AOF file but not enough bytes
965:# HyperLogLog sparse representation bytes limit. The limit includes the
966:# 16 bytes header. When an HyperLogLog using the sparse representation crosses
977:hll-sparse-max-bytes 3000
1017:# So for instance if the hard limit is 32 mega bytes and the soft limit is
1018:# 16 mega bytes / 10 seconds, the client will get disconnected immediately
1019:# if the size of the output buffers reach 32 mega bytes, but will also get
1020:# disconnected if the client reaches 16 mega bytes and continuously overcomes
root@iZwz9bruzaqhhivjjbrsklZ:/usr/local/redis#
```

5.Head tail awk 获取指定行号的内容

```
root@iZwz9bruzaqhhivjjbrsklZ:/usr/local/redis# head -n 20 redis.conf | tail -n 5
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1gB are all the same.

##### INCLUDES #####
root@iZwz9bruzaqhhivjjbrsklZ:/usr/local/redis# awk 'NR>=15&&NR<=20' redis.conf
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1gB are all the same.

##### INCLUDES #####
root@iZwz9bruzaqhhivjjbrsklZ:/usr/local/redis#
```

显示行号

```
root@iZwz9bruzaqhhivjjbrsk1Z:/usr/local/redis# cat -n redis.conf | awk 'NR>=10&&NR<=15'
10  #
11  # 1k => 1000 bytes
12  # 1kb => 1024 bytes
13  # 1m => 1000000 bytes
14  # 1mb => 1024*1024 bytes
15  # 1g => 1000000000 bytes
root@iZwz9bruzaqhhivjjbrsk1Z:/usr/local/redis#
```

8 Linux 内核主要有几种内核锁

主要有自旋锁和信号量

自旋锁是防止多处理器并发而引入的一种锁，在内核中大量应用于中断处理

9 Linux 用户模式和内核模式

用户模式是受限模式，对内存和硬件的访问必须通过系统调用实现

内核模式是一种高特权模式，能直接访问内存和硬件

10 Linux 内核下 tomcat 优化

我们姑且把上面四种 Connector 按照顺序命名为 NIO, HTTP, POOL, NIOP。假设地址是 <http://tomcat1/test.jsp>。我们依次对四种 Connector 进行测试，测试的客户端在另外一台机器上用 ab 命令来完成，测试命令为： ab -c 900 -n 2000 http://tomcat1/test.jsp，最终的测试结果如下表所示(单位:平均每秒处理的请求数):

NIO	HTTP	POOL	NIOP
281	65	208	365
666	66	110	398
692	65	66	263
256	63	94	459
440	67	145	363

由这五组数据不难看出，**HTTP 的性能是很稳定，但是也是最差的**，而这种方式就是 Tomcat 的默认配置。**NIO 方式波动很大，但没有低于 280 的**，NIOP 是在 NIO 的基础上加入线程池，可能是程序处理更复杂了，因此性能不见得比 NIO 强；而 POOL 方式则波动很大，测试期间和 HTTP 方式一样，不时有停滞。连接器的传输方式： protocol="org.apache.coyote.http11.Http11NioProtocol"

11 Linux 系统调用

- 1 它为用户空间提供了一种**统一的硬件的抽象接口**。比如当需要读些文件的时候，应用程序就可以不去管磁盘类型和介质，甚至不用去管文件所在的文件系统到底是哪种类型。
- 2 系统调用**保证了系统的稳定和安全**。作为硬件设备和应用程序之间的中间人，内核可以基于权限和其他一些规则对需要进行的访问进行裁决。举例来说，这样可以避免应用程序不正确地使用硬件设备，窃取其他进程的资源，或做出其他什么危害系统的事情。
- 3 每个进程都运行在虚拟系统中，而在用户空间和系统的其余部分提供这样一层公共接口，也是出于这种考虑。如果应用程序可以随意访问硬件而内核又对此一无所知的话，几乎就无法实现多任务和虚拟内存，当然也不可能实现良好的稳定性和安全性。在 Linux 中，系统调用是**用户空间访问内核的唯一手段**；除异常和中断外，它们是内核唯一的合法入口。

12 Linux 下查看 IO，定位 IO 瓶颈的一些办法

iowait≠IO 负载，要看真实的 IO 负载情况，一般使用 iostat -x 命令：

```
$ iostat -x 1
```

```
avg-cpu: %user %nice %system %iowait %steal %idle
      0.04  0.00  0.04  4.99  0.00  94.92
```

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrrq-sz avgqu-sz await svctm %util
sda      0.00  81.00 104.00 4.00 13760.00 680.00 133.70 2.08 19.29 9.25 99.90
```

这里重点指标是 **svctm** 和 **util** 这两列，svctm 指的是“平均每次设备 I/O 操作的服务时间 (毫秒)”，而 util 指的是“一秒中 I/O 操作的利用率，或者说一秒中有多少时间 I/O 队列是非空的。”

从上面发现 util 已经接近 100%，所以这台机器已经到了 I/O 瓶颈

top 里的 wa%指的是从整体来看，CPU 等待 I/O 的耗时占比，也就是说，CPU 可能拿出一部分时间来等待 I/O 完成 (iowait)，但从磁盘的角度看，磁盘的利用率已经满了 (util%)，这种情况下，CPU 使用率可能不高，但是系统整体 QPS 已经上不去了，如果加大流量，会导致单次 I/O 耗时的继续增加 (因为 I/O 请求都堵在队列里了)，从而影响系统整体的处理性能。

确认了 I/O 负载过高后，可以使用 iotop 工具具体查看 I/O 负载主要是落在哪个进程上了

那如何**规避 I/O 负载过高**的问题呢？具体问题具体分析：

1. 如果你的服务器用来做日志分析，要避免多个 crontab 交叠执行导致多进程随机 I/O(参考：随机 I/O vs 顺序 I/O)，避免定期的压缩、解压大日志 (这种任务会造成某段时间的 I/O 抖动)。
2. 如果是前端应用服务器，要避免程序频繁打本地日志、或者异常日志等。
3. 如果是存储服务 (mysql、nosql)，尽量将服务部署在单独的节点上，不要和其它服务共用，甚至服务本身做读写分离以降低读写压力；调优一些 buffer 参数以降低 I/O 写的频率等等。另外还可以参考 LevelDB 这种将随机 I/O 变顺序 I/O 的经典方式。

操作系统

1 操作系统寻址方式

立即数寻址

寄存器寻址

 直接寻址

 间接寻址

 相对寻址

 基址变址寻址

 相对基址变址寻址

2 外部碎片和内部碎片

外部碎片：由于大量信息先后写入，置换，删除而形成的空间碎片；

6间仓库，装了 1-5，然后清空了 4，剩下 4 和 6 是空的，现在需要连续的两间仓库来存储信息，是存不了的，造成了碎片。

内部碎片：由于存储信息的容量与最小存储空间单位不完全相符而造成的空间碎片。

6间仓库，管理仓库的最小空间单位是间。现在存了 2.5 间，3号仓库只存了一半，那么 3 号仓库剩下的半间就不能用了，因为管理仓库的最小空间单位是间，造成了碎片。

3 操作系统什么情况下会死锁

一般都是存在资源竞争：比如获取一个资源必须获取两把锁 A 和 B。线程甲已经获取了锁

A, 线程乙已经获取到了锁 B, 但是线程甲要获取锁 B, 线程乙要获取锁 A, 这时候互相等待, 就会导致死锁。

4 进程和线程

进程是资源分配的基本单位, 所有与该进程相关的资源, 都被记录在进程控制块 PCB 中, 以表示该进程拥有这些资料并正在使用它们。

线程是轻量级的进程, 是独立运行和独立调度的基本单位, 是一个基本的 CPU 执行单元。它与资源分配无关, 线程本身不拥有系统资源, 但是它可与同属一个进程的其他线程共享进程所拥有的全部资源。通常在一个进程中可以包含若干个线程。

进程和线程的区别可以归纳为 4 点:

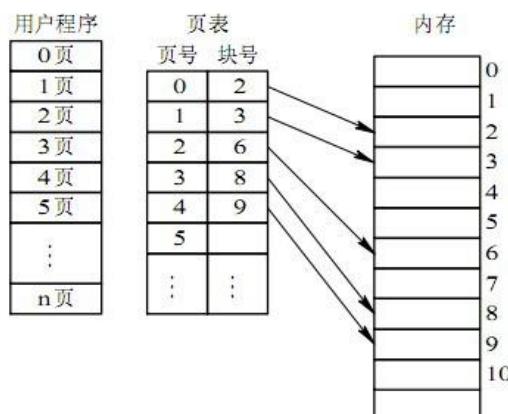
1. 地址空间和其他资源: 进程间相互独立, 同一进程间的各线程共享;
2. 通信: 进程间通信 IPC, 线程间可以直接读写进程数据段进行通信;
3. 调度和切换: 线程上下文切换比进程上下文切换快得多;
4. 在多线程操作系统中, 进程不是一个可执行的实体。 (?)

5 常用进程调度算法

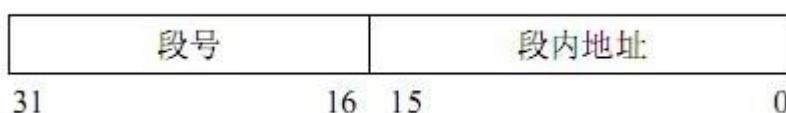
1. 先来先服务;
2. 短进程优先; (从就绪队列中选出一个估计时间最短的进程, 分配 CPU, 使它立即执行并执行到完成, 或者发生过某时间而被阻塞时再重新调度)
3. 高优先权优先调度; (抢占式, 非抢占式)
4. 高响应比优先; (优先权= (等待时间+要求服务时间) /要求服务时间)
5. 时间片轮转法。

6 操作系统内存管理

页式管理地址



段式管理地址



<http://blog.csdn.net/hguisu/article/details/5713164>

7 Windows 进程间各种通信方式浅谈

1 文件映射

应用程序有三种方法来使多个进程共享一个文件映射对象。

(1)继承：第一个进程建立文件映射对象，它的子进程继承该对象的句柄。

(2)命名文件映射：第一个进程在建立文件映射对象时可以给该对象指定一个名字(可与文件名不同)。第二个进程可通过这个名字打开此文件映射对象。

另外，第一个进程也可以通过一些其它 IPC 机制(有名管道、邮件槽等)把名字传给第二个进程。

(3)句柄复制：第一个进程建立文件映射对象，然后通过其它 IPC 机制(有名管道、邮件槽等)把对象句柄传递给第二个进程。

2 共享内存

Win32 API 中共享内存(Shared Memory)实际就是文件映射的一种特殊情况。进程在创建文件映射对象时用 0xFFFFFFFF 来代替 文件句柄(HANDLE)，由于共享内存是用 文件映射实现的，所以它也有较好的安全性，也只能运行于同一计算机上的进程之间。

3 匿名管道

管道(Pipe)是一种具有两个端点的通信通道：有一端句柄的进程可以和有另一端句柄的进程通信。管道可以是单向——一端是只读的，另一端是只写的；也可以是双向的一管道的两端点既可读也可写。匿名管道(Anonymous Pipe)是在父进程和子进程之间，或同一父进程的两个子进程之间传输数据的无名字的单向管道。通常由父进程创建管道

4 剪切板 5 动态链接库 DLL 6 远程过程调用

8 大端模式和小端模式

Big-Endian 和 Little-Endian 的定义如下：

1) Little-Endian 就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

2) Big-Endian 就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

举一个例子，比如数字 0x12 34 56 78 在内存中的表示形式为：

1)大端模式：符号位的判定固定为第一个字节，容易判断正负。

低地址----->高地址

0x12 | 0x34 | 0x56 | 0x78

2)小端模式：强制转换数据不需要调整字节内容，1、2、4 字节的存储方式一样

低地址 -----> 高地址

0x78 | 0x56 | 0x34 | 0x12

可见，大端模式和字符串的存储模式类似。我们常用的 X86 结构是小端模式

http://blog.csdn.net/ce123_zhouwei/article/details/6971544

12 数据结构

1 各种排序（一张表）

排序算法	平均时间复杂度	最坏复杂度	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n \log n)$	$O(n^2)$	$O(\log n)$	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(1)$	稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定
希尔排序	$O(n \log n)$	$O(n^{\frac{3}{2}})$	$O(1)$	不稳定
基数排序	$O(\log_R B)$	$O(\log_R B)$	$O(n)$	稳定
二叉树排序	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	稳定

2 冒泡排序

- 1) **定义:** 以从小到大排序为例，每一轮排序就找出未排序序列中最大值放在最后。
- 1) **实现 1:** 以最常规的思路，两层遍历。第一层表示需要 n 轮排序过程，第二层表示第 i 轮时候将最大值排序在 $a[n-i]$ 位置。但是
- 2) **优化 1:** 如果序列已经有序了，那么就可以避免不必要的再比较。所以利用一个标志位 `flag` 表示某一轮循环是否发生了交换，如果没有发生交换就直接结束循环，表示已经排序完成。
- 4) **优化 2:** 记录排序过程中发生交换的尾边界，每次循环只需要遍历到尾边界就 OK。

上面的优化过程的核心都是：减少比较的次数。

具体的代码可以参考：<http://blog.csdn.net/u010853261/article/details/54891710>

3 直接插入排序

- 1) 定义：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中的适当位置，直到全部记录插入完成为止。
- 2) 实现 1：
 - a) 初始时， $a[0]$ 自成 1 个有序区，无序区为 $a[1..n-1]$ 。令 $i=1$
 - b) 将 $a[i]$ 并入当前的有序区 $a[0..i-1]$ 中形成 $a[0..i]$ 的有序区间。
 - C) $i++$ 并重复第二步直到 $i=n-1$ ，排序完成。
- 3) **优化 1:** 从后往前比较，边比较边移动。即每次 $a[i]$ 先和前面一个数据 $a[i-1]$ 比较，如果 $a[i] > a[i-1]$ 说明 $a[0..i]$ 也是有序的，无须调整。否则就令 $j=i-1, temp=a[i]$ 。然后一边将数据 $a[j]$ 向后移动一边向前搜索，当有数据 $a[j]$ 小于 $temp$ 时就找到了插入位置。

4) 优化 2: 将前面的数据移动变为数据交换。如果 $a[j]$ 前一个数据 $a[j-1] > a[j]$, 就交换

$a[j]$ 和 $a[j-1]$, 直到 $a[j-1] \leq a[j]$ 。这样也可以实现将一个新数据新并入到有序区间。

上面的时间复杂度均为 $O(n^2)$, 不过后面的优化能够改善一点点性能。

实现代码: <http://blog.csdn.net/u010853261/article/details/54891737>

4 快速排序

快排就是先一个定位函数，定位函数是找到一个数（可以是随机，一般用第一个数），然后把比这个数小的都放在左边，把比这个数大的都放在右边，然后快排递归调用这个定位函数即可。

快速排序是非常重要的，这在笔试经常会用到：

1) 定义：基于分治的思想。首先在数组中选择一个基准点并把基准点放于序列的开头（该基准点的选取是优化快排的关键点），然后分别从数组的两端扫描数组，设两个指示标志（ lo 指向起始位置， hi 指向末尾），首先从后半部分开始，如果发现有元素比该基准点的值小，就交换 lo 和 hi 位置的值，然后从前半部分开始扫描，发现有元素大于基准点的值，就交换 lo 和 hi 位置的值，如此往复循环，直到 $lo \geq hi$ ，然后把基准点的值放到 hi 这个位置，一次排序就完成了。之后再采用递归的方式分别对前半部分和后半部分排序，当前半部分和后半部分均有序时该数组自然也就有序了。

2) 一轮快速排序的过程：

```
1  /**
2  * 一次快速排序
3  * @param array 数组
4  * @param lo 数组的前下标
5  * @param hi 数组的后下标
6  * @return key的下标index, 也就是分片的间隔点
7  */
8  public static int partition(int []array,int lo,int hi){
9      /** 固定的切分方式 */
10     int key=array[lo];//选取了基准点
11     while(lo<hi){
12         //从后半部分向前扫描
13         while(array[hi]>=key&&hi>lo){
14             hi--;
15         }
16         array[lo]=array[hi];
17         //从前半部分向后扫描
18         while(array[lo]<=key&&hi>lo){
19             lo++;
20         }
21         array[hi]=array[lo];
22     }
23     array[hi]=key;//最后把基准存入
24     return hi;
25 }
```

下面值得一说的是新一轮快速排序我们能得到什么？

A) 举个例子，对于 4、3、7、6、1 这个序列，以 4 为基准，一轮快速排序之后的序列为 1、3、4、6、7。对于一轮快速排序之后的序列有这样的规律，对于基准 4

前面的序列 b 数据都比 4 小，对于 4 后面的序列 c 都比 4 大。

- B) 上面的一轮快排的规律我们就能有所应用，比如对于一些求序列中最小的 k 个数。这时候序列 b 中数据都比基准小，所以当序列 b 数据量小于 k-1 时，说明序列 b 数据都是满足条件的，我们继续在序列 c 中再次执行快排，找结果。

3) 实现快排排序：

```
1  /**
2  * 快速排序
3  * @param array
4  * @param lo
5  * @param hi
6  */
7  public static void quickSort(int[] array,int lo ,int hi){
8      if(lo>=hi){
9          return ;
10     }
11     //进行第一轮排序获取分割点
12     int index=partition(array,lo,hi);
13     //排序前半部分
14     quickSort(array, lo, index - 1);
15     //排序后半部分
16     quickSort(array,index+1,hi);
17 }
```

4) 优化：对于快排的优化，一般都是基准点选取的优化。

基准点一般有三种选择：固定、随机、三数取样。下面给出三数取中的实现：

三数取中的思想也就是：比如对于 a、b、c，先比较 a 和 c 以及 b 和 c 保证 c 是最大的，然后再比较 a 和 b，选出中间值。

```
1  //三数取中
2  //下面的两步保证了array[hi]是最大的;
3  int mid=lo+(hi-lo)/2;
4  if(array[mid]>array[hi]){
5      swap(array[mid],array[hi]);
6  }
7  if(array[lo]>array[hi]){
8      swap(array[lo],array[hi]);
9  }
10 //接下来只用比较array[lo]和array[mid]，让较小的在array[lo]位置就OK。
11 if(array[mid]>array[lo]){
12     swap(array[mid],array[lo]);
13 }
14
15 int key=array[lo];
```

5) 时间复杂度和空间复杂度

对于快排来说平均时间复杂度是 $O(n \log n)$ ，但是当序列本身已经有序的情况下时间复杂度最糟糕是 $O(n^2)$ 。并且快速排序是不稳定的。

5 归并排序

- 1) 定义：归并算法是采用分治法（Divide and Conquer）的一个非常典型的应用，归并排序将两个已排序的表合并成一个表。
在这里实现归并排序其实就是两个过程：将一个序列先递归拆分序列，再归并。

- 2) 首先思考归并两个有序序列的实现，其实这个实现比较简单，时间复杂度也是 $O(n)$ ：这里归并数组 $a[first, mid]$ 和 $a[mid+1, last]$ 这两个数组。这里需要借助一个辅助数组。

```
/*
 * 将a[first, mid] 和 a[mid+1, last] 合并
 * @param a
 * @param first
 * @param mid
 * @param last
 * @param temp
 */
private static void mergeArray(int a[], int first, int mid, int last, int temp[]){
    int i = first, j=mid+1;//设置两个数组的起始边界
    int m=mid, n=last;//设置两个数组的结束边界

    int k=0;

    while (i <= m && j<=n){
        if(a[i] <= a[j]){
            temp[k++] = a[i++];
        }else {
            temp[k++] = a[j++];
        }
    }

    while (i<=m){
        temp[k++] = a[i++];
    }

    while (j <= n){
        temp[k++] = a[j++];
    }

    for(i=0; i<k; i++){
        a[first+i] = temp[i];
    }
}
```

- 3) 再看递归的拆分：

```

/**
 * 二路归并 使用递归解决.
 * @param a
 * @param first 数组的起始下标
 * @param last 数组的结束下标
 * @param temp 辅助数组
 */
public static void mergeSort(int[] a, int first, int last, int[] temp){
    if(first < last) {
        int mid = (first + last)/2;

        mergeSort(a, first, mid, temp); //左边有序
        mergeSort(a, mid+1, last, temp); //右边有序

        mergeArray(a, first, mid, last, temp); //再将两个有序序列合并.
    }
}

```

- 4) 前面的二路归并是基于数组实现的，但是如果我们要**基于单链表实现**呢？我们还是从两个角度考虑：拆分和合并。
 - a) 合并：对于两个有序单链表的合并并返回合并之后的单链表头结点，这个的是比较简单的，和数组的实现类似。
 - b) 拆分：对于拆分，可能就没有数组那么方便了，因为我们不能直接通过首尾索引值取平均数，但是链表也是可以实现的，我们可以使用双指针法，对于两个指针 p1 和 p2，分别往后移动，p1 移动一次，p2 移动两次，当 p2 遍历到尾结点时，p1 就指向中间结点实现了拆分。

具 体 的 实 现 就 不 给 出 代 码 了 ， 参 考 博 客 :

<http://blog.csdn.net/u010853261/article/details/54884650>

6 二分查找

- 1) 定义：二分查找的前提是**待查找的序列有序**。每次取中间位置的值与待查关键字比较，如果中间位置的值比待查关键字大，则在前半部分循环这个查找的过程，如果中间位置的值比待查关键字小，则在后半部分循环这个查找的过程。直到查找到了为止，否则序列中没有待查的关键字。

- 1) 非递归实现：

```


/**
 * 非递归方式查找
 * @param array
 * @param a
 * @return 查找到的索引值
 */
public static int binarySearch1(int[] array, int a){
    int lo=0, hi=array.length-1;

    int mid;//中间索引

    while (lo <= hi){
        mid = (lo+hi)/2;

        if(array[mid] == a){
            return mid;
        } else if(array[mid] < a){
            lo = mid+1;
        }else{
            hi = mid-1;
        }
    }
    return -1;
}


```

3) 递归实现:

```


1 /**
2  * 递归方式查找
3  * @param array
4  * @param a
5  * @return
6 */
7 public static int binarySearch2(int[] array, int a, int lo, int hi){

8     if(lo <= hi){
9         int mid = (lo+hi)/2 ;

10        if( a == array[mid] ){
11            return mid;
12        } else if(a > array[mid]){
13            return binarySearch2(array, a, mid+1, hi);
14        } else {
15            return binarySearch2(array, a, lo, mid-1);
16        }
17    }
18    return -1;
21 }


```

7 什么是二叉平衡树，如何插入节点，删除节点，说出关键步骤

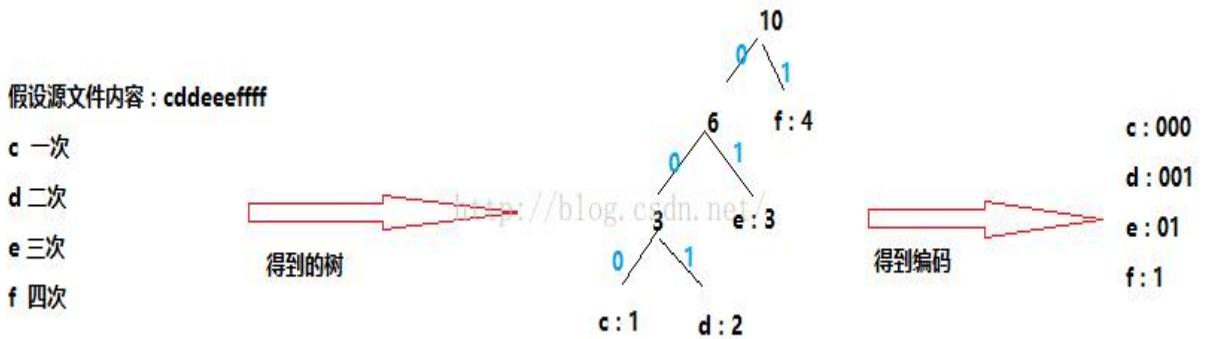
- 1) 二叉平衡树：或则是一颗空树 或则它的任意一个节点的左右两个子树的高度差的绝对值不超过 1，且左右子树都是二叉平衡树。
- 2) 插入：和二叉查找树类似，只是可能会破坏树的平衡性，然后进行相应的旋转操作。
- 3) 删除：（1）叶节点直接删除。 （2）非叶节点，要处理失衡情况。

8 桶排序

桶排序的基本思想是将一个数据表分割成许多 buckets，然后每个 bucket 各自排序，或用不同的排序算法，或者递归的使用 bucket sort 算法。也是典型的 divide-and-conquer 分而治之的策略。它是一个分布式的排序，介于 MSD 基数排序和 LSD 基数排序之间。

9 Huffman 树

Huffman 树，又称为最优二叉树，是加权路径长度最短的二叉树



13 简历

1 项目

决策树

两种决策树学习的生成算法：ID3 和 C4.5

信息熵的概念：一条信息的信息量大小和它的不确定性有直接的关系。我们对一样东西越是一无所知，想要了解它就需要越多的信息。

ID3 算法使用的就是基于信息增益的选择属性方法。信息增益选择方法有一个很大的缺陷，它总是会倾向于选择属性值多的属性，如果我们在上面的数据记录中加一个姓名属性，假设 14 条记录中的每个人姓名不同，那么信息增益就会选择姓名作为最佳属性，因为按姓名分裂后，每个组只包含一条记录，而每个记录只属于一类（要么购买电脑要么不购买），因此纯度最高，以姓名作为测试分裂的结点下面有 14 个分支。但是这样的分类没有意义，它没有任何泛化能力。

C4.5 使用增益率（gain ratio）的信息增益扩充，试图克服这个偏倚。C4.5 选择具有最大增益率的属性，ID3 选择最大信息获取量的属性，其余没啥差别。

预剪枝：当决策树在生成时当达到该指标时就停止生长，比如小于一定的信息获取量或是一定的深度，就停止生长。

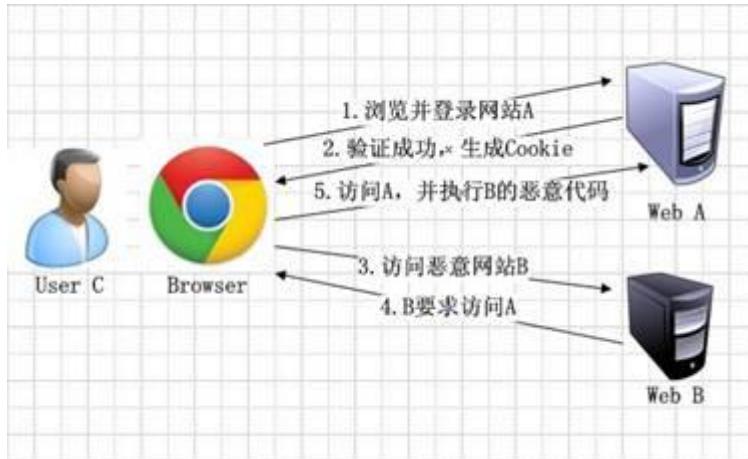
后剪枝：当决策树生成完后，再进行剪枝操作。优点是克服了“视界局限”效应，但是计算量代价较大。

1 紫领网

<http://www.open-open.com/jsoup/dom-navigation.htm>

Jsoup.parse 检查 <script><form> <a href>

CSRF 漏洞 跨站点伪造请求,CSRF 攻击原理比较简单, 如图 1 所示。其中 Web A 为存在 CSRF 漏洞的网站, Web B 为攻击者构建的恶意网站, User C 为 Web A 网站的合法用户。



http://www.h3c.com.cn/About_H3C/Company_Publication/IP_Lh/2012/04/Home/Catalog/201208/751467_30008_0.htm CSRF 漏洞

#{}是经过预编译的, 是安全的; \${}是未经过预编译的, 仅仅是取变量的值, 是非安全的, 存在 SQL 注入。

git stash: 备份当前的工作区的内容, 从最近的一次提交中读取相关内容, 让工作区保证和上次提交的内容一致。同时, 将当前的工作区内容保存到 Git 栈中。

git stash pop: 从 Git 栈中读取最近一次保存的内容, 恢复工作区的相关内容。由于可能存在多个 Stash 的内容, 所以用栈来管理, pop 会从最近的一个 stash 中读取内容并恢复。

git stash list: 显示 Git 栈内的所有备份, 可以利用这个列表来决定从那个地方恢复。

git stash clear: 清空 Git 栈。此时使用 gitg 等图形化工具会发现, 原来 stash 的哪些节点都消失了。

Web.xml 配置

```
<!-- 欢迎页 -->
<welcome-file-list>
<!-- Post 解决编码 -->
<filter>
    <filter-name>CharsetFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
```

```
</init-param>
</filter>
<filter-mapping>
    <filter-name>CharsetFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>


<context-param>
    <param-name>sysname</param-name>
    <param-value>
        紫领网-专注软件众包
    </param-value>
</context-param>
<listener>
    <listener-class>
        com.ziling.www.web.SystemInit
    </listener-class>
</listener>

<!-- 过滤器的配置--&gt;
<!-- 后台管理系统路由转换,只过滤一个地址,也就是/admin 这个地址,然后直接重定向到后台的登录界面 --&gt;
&lt!-- &lt;filter&gt;
    &lt;filter-name&gt;adminFilter&lt;/filter-name&gt;
    &lt;filter-class&gt;com.ziling.www.web.filter.AdminChangeRouteFilter&lt;/filter-class&gt;
&lt;/filter&gt;
&lt;filter-mapping&gt;
    &lt;filter-name&gt;adminFilter&lt;/filter-name&gt;
    &lt;url-pattern&gt;/admin&lt;/url-pattern&gt;
&lt;/filter-mapping&gt;--&gt;

<!-- 配置缓存过滤器, --&gt;
&lt;filter&gt;
    &lt;filter-name&gt;cacheFilter&lt;/filter-name&gt;
    &lt;filter-class&gt;com.ziling.www.web.filter.CacheFilter&lt;/filter-class&gt;
&lt;/filter&gt;
&lt;filter-mapping&gt;
    &lt;filter-name&gt;cacheFilter&lt;/filter-name&gt;
    &lt;url-pattern&gt;/static/*&lt;/url-pattern&gt;
&lt;/filter-mapping&gt;</pre>
```

```
<!-- 静态资源过滤器 -->
<filter>
    <filter-name>staticResourceFilter</filter-name>
    <filter-class>com.ziling.www.web.filter.StaticResourceFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>staticResourceFilter</filter-name>
    <url-pattern>/static/*</url-pattern>
</filter-mapping>

<!-- Struts2 的配置 -->
<filter>
    <filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <!-- 拦截所有的就是 /* -->
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- Spring 配置文件的注入和启动 Spring -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:spring/applicationContext.xml,
        classpath:spring/service_system.xml
        <!--WEB-INF/classes/spring/applicationContext.xml,
        /WEB-INF/classes/spring/applicationContext-shiro.xml,
        /WEB-INF/classes/spring/service_system.xml-->
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>

<!-- shiro 的 filter -->
```

```

<!-- shiro 过滤器，DelegatingFilterProxy 通过代理模式将 spring 容器中的 bean 和 filter 关联起来 -->
<!-- <filter>

<filter-name>shiroFilter</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
设置 true 由 servlet 容器控制 filter 的生命周期
<init-param>
    <param-name>targetFilterLifecycle</param-name>
    <param-value>true</param-value>
</init-param>
设置 spring 容器 filter 的 bean id, 如果不设置则找与 filter-name 一致的 bean
<init-param>
    <param-name>targetFilterLifecycle</param-name>
    <param-value>shiroFilter</param-value>
</init-param>
</filter>
<filter-mapping>
    <filter-name>shiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping> -->

```

2 卫生监督接口项目

Web.xml 配置 XFireServlet

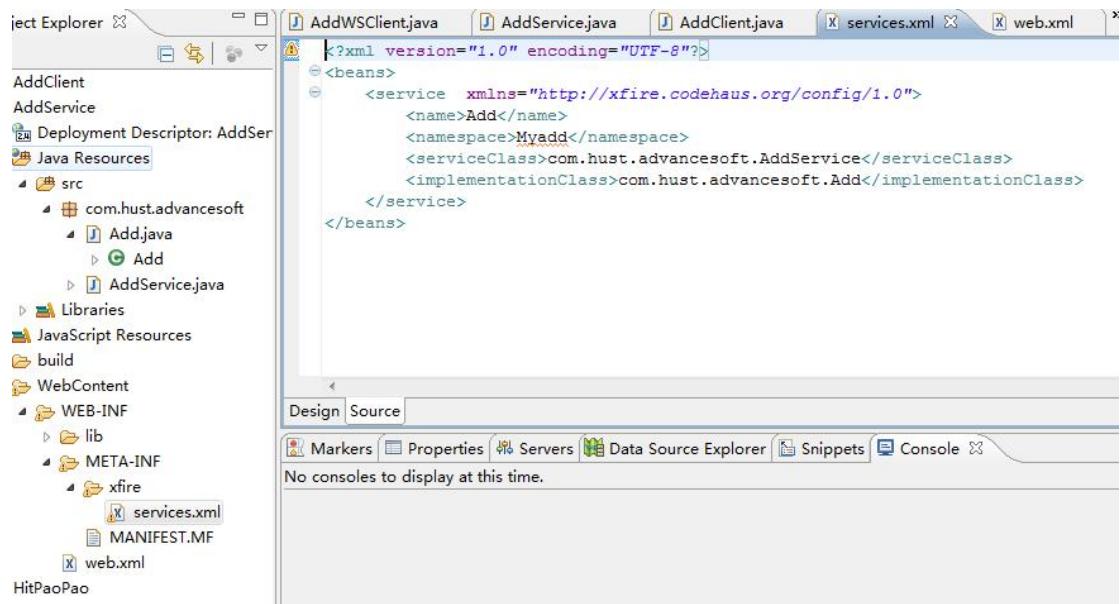
```

<servlet>
    <servlet-name>XFireServlet</servlet-name>
    <servlet-class>org.codehaus.xfire.transport.http.XFireConfigurableServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>XFireServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>

```

配置 services.xml，把接口和实现类都配置上去



客户端去获取 XFire 实例，远程过程调用

```

public static void main(String[] args) throws Exception {
    // TODO Auto-generated method stub
    AddClient bank = new AddClient();
    int s = bank.callWebService(2, 3);
    System.out.println(s);
}

public int callWebService(int a, int b) throws Exception {
    // Create a metadata of the service 创建一个service的元数据
    Service serviceModel = new ObjectServiceFactory()
        .create(AddService.class);

    // Create a proxy for the deployed service 为XFire获得一个代理工厂那个对象
    XFire xfire = XFireFactory.newInstance().getXFire();
    XFireProxyFactory factory = new XFireProxyFactory(xfire);

    // 得到一个服务的本地代理
    String serviceUrl = "http://localhost:8080/AddService/services/Add";

    AddService client = null;

    try {
        client = (AddService) factory.create(serviceModel, serviceUrl);
    } catch (Exception e) {
        System.out.println("WsClient.callWebService():Exception:"
            + e.toString());
    }

    // invoke the service 调用服务返回状态结果
    int serviceResponse = 0;
    try {
        serviceResponse = client.add(a, b);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return serviceResponse;
}

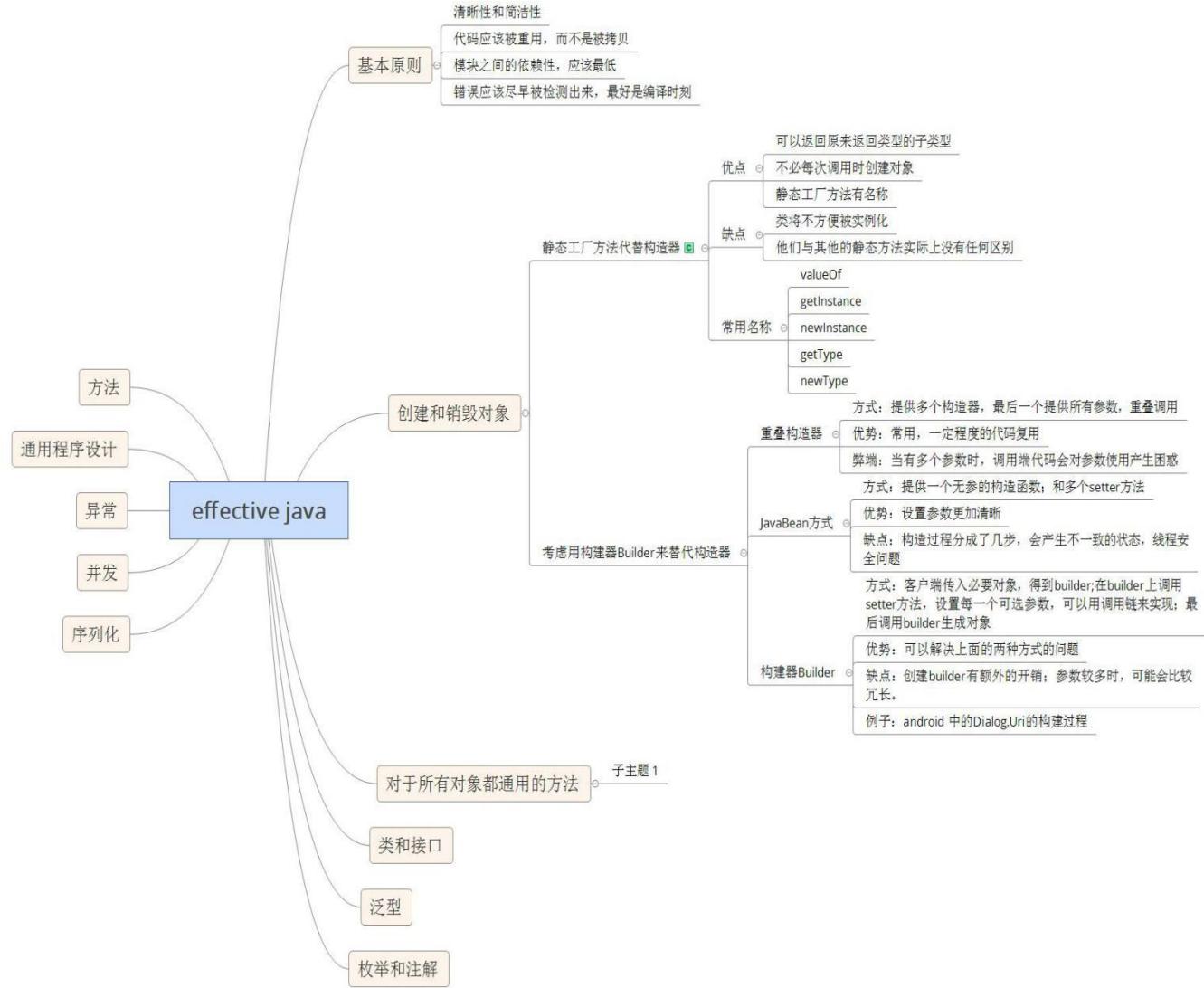
```

1 Java 编程思想

2 JavaCore

3 EffectJava

Wait 一定要在循环中使用，因为有“虚假唤醒”，唤醒的时候需要再次判断满足条件不



4 JVM

5 爬虫

Java 是基于 HttpClient 库，但是如果运维分析日志发现都是 IP(1.1.1.1)这个用户，并且 useragent 还是 JavaClient1.6，基于这两点判断非人类后直接在 Nginx 服务器上封杀。这个时候就得变换下策略：

1. useragent 模仿百度("Baiduspider...")

```
$ip = '220.181.108.91'; // 百度蜘蛛  
//伪造百度蜘蛛 IP  
//伪造百度蜘蛛头部
```

2. IP 每爬半个小时就换一个 IP 代理。

ADSL 拨号上网使用动态 IP 地址，每一次拨号得到的 IP 都不一样，所以我们可以通过程序来自动进行重新拨号以获得新的 IP 地址，以达到突破反爬虫封锁的目的。假设有 10 个线程在跑，大家都正常的跑，跑着跑着达到限制了，WEB 服务器提示你“非常抱歉，来自您 ip 的请求异常频繁”，于是大家争先恐后（几乎是同时）请求拨号，这个时候同步的作用就显示出来了，只会有一个线程能拨号，在他结束之前其他线程都在等，等他拨号成功之后，其他线程会被唤醒并返回。

算法描述：

- 1、假设我现在有 N 个线程在抓取网页，发现被封锁之后就依次排队请求锁，假设线程 1 抢先得到锁，并设置 flag=true 开始拨号，注意，注意：线程 1 设置 isDialing = true 后其他线程才可能获得锁。
- 3、其他线程（2-N）依次获得锁，发现 isDialing = true，于是 wait。注意：获得锁并判断一个布尔值，跟后面的拨号操作比起来，时间可以忽略。
- 4、线程 1 拨号完毕 isDialing = false。注意：这个时候可以断定，其他所有线程必定是处于 wait 状态等待唤醒。
- 5、线程 1 唤醒其他线程，其他线程和线程 1 返回开始抓取网页。
- 6、抓了一会儿之后，又会被封锁，于是回到步骤 1。

运维也发现了对应的变化，于是在 Nginx 上设置了一个频率限制，每分钟超过 120 次请求的再屏蔽 IP。同时考虑到百度家的爬虫有可能会被误伤，想想市场部门每月几十万的投放，于是写了个脚本，通过 hostname 检查下这个 ip 是不是真的百度家的，对这些 ip 设置一个白名单。

写爬虫的发现了新的限制后，想着我也不急着要这些数据，留给服务器慢慢爬吧，于是修改了代码，随机 1-3 秒爬一次，爬 10 次休息 10 秒，每天只在 8-12, 18-20 点爬，隔几天还休息一下。

运维看着新的日志头都大了，再设定规则不小心会误伤真实用户，于是准备换了一个思路，当 3 个小时的总请求超过 50 次的时候弹出一个验证码弹框，没有准确正确输入的话就把 IP 记录进黑名单。

14 算法题

剑指 offer

8 旋转数组的最小数字

用二分查找，最开始比较第一个数字和最后一个数字，如果最后一个数字比第一个数字大，那就是有序的，没有旋转过。如果是旋转过，如果中间这个数比开始那个数大，那么最小的肯定在他的右边，如果小，肯定在左边，但是有特殊情况，如果是特殊情况，我们必须按照遍历那种去算。

16 反转链表

```
public ListNode reverseList(ListNode head) {  
    if (head == null || head.next == null)  
        return head;  
    ListNode temp = head.next;  
    ListNode reverseHead = reverseList(head.next);  
    temp.next = head;  
    head.next = null;  
    return reverseHead;  
}  
  
public ListNode reverseList2(ListNode pHead) {  
    if (pHead == null || pHead.next == null)  
        return pHead;  
    ListNode pReverseHead = null;  
    ListNode pNode = pHead;  
    ListNode pPrev = null;  
  
    while (pNode != null) {  
        ListNode pNext = pNode.next;  
        if (pNext == null) {  
            pReverseHead = pNode;  
        }  
        pNode.next = pPrev;  
  
        pPrev = pNode; |  
        pNode = pNext;  
    }  
    return pReverseHead;  
}
```

36 数组中的逆序对

先把数组分割成子数组，先统计出子数组内部的逆序对的数目，然后统计出两个相邻子数组之间的逆序对的数目。在统计逆序对的过程中，还需要对数组进行排序。其实这个过程就是归并排序。

1 两个无序数组，求出共同的

构建哈希表插入操作的过程中，如果元素已经插入过，即其哈希地址有值，则该元素必为两数组的交集，打印输出即可。（前提数组中的元素不重复）

2 判断平衡二叉树

求深度递归，但是时间复杂度高了，要求一次遍历（后续遍历）

3 股价 贪心算法

http://blog.csdn.net/qg_26437925/article/details/52780895

1 买卖一次

从前往后，用当前价格减去此前的最低价格，就是在当前点卖出股票能获得的最高利润。

```
[cpp] 
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        //遍历，目标找到一对依次出现的最低点和最高点，使得满足达到最大利润

        if(prices.size() <= 1)
            return 0;

        int low = prices[0];
        int maxprofit = 0;

        for(int i = 1; i < prices.size(); i++)
        {
            int profit = prices[i] - low;
            if(maxprofit < profit) maxprofit = profit;
            if(prices[i] < low) low = prices[i];
        }
        return maxprofit;
    }
};
```

2 买卖多次

```
[cpp] 
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        //贪心算法 每次获得局部最优
        if(prices.size() == 0) return 0;

        int maxprofit = 0;
        for(int i = 1; i < prices.size(); i++)
        {
            if(prices[i] > prices[i-1])
            {
                maxprofit += prices[i] - prices[i-1];
            }
        }

        return maxprofit;
    }
};
```

4 最大子序列

```

public int maxSubSum(int[] a) {
    if (a == null || a.length == 0)
        return 0;

    int maxSum = 0, thisSum = 0;
    for (int i = 0; i < a.length; i++) {
        thisSum += a[i];
        if (thisSum > maxSum)
            maxSum = thisSum;
        if (thisSum < 0)
            thisSum = 0;
    }
    return maxSum;
}

```

5 一个排好序的数组，找出两数之和为 m 的所有组合

前后同时往中间加就行

6 自然数序列，找出任意连续之和等于 n 的所有子序列

连续的数与 n 比较，如果比 n 小就把后面的数往上加，如果比 n 大就把前面的数减下去。

7 给定 2, 3, 5 面值的硬币若干，给出一个数字，计算拼凑这个数字最少用到硬币的个数

动态规划

先与最大的求余嘛，得到的商记录下来，然后把这些记录加起来就是硬币的个数

8 二叉树 BFS（广度优先） DFS（深度）

BFS 相当于层次遍历，用栈 DFS 相当于中序遍历，用堆

```

// 层次遍历
public void levelOrder(TreeNode T) {
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.add(T);
    TreeNode p = null;
    while (!queue.isEmpty()) {
        p = queue.poll();
        visit(p);
        if (p.left != null)
            queue.add(p);
        if (p.right != null)
            queue.add(p);
    }
}

// 中序遍历，递归求解
public void inOrder(TreeNode T) {
    if (T != null) {
        inOrder(T.left);
        visit(T);
        inOrder(T.right);
    }
}

// 中序遍历，非递归求解
public void inOrder2(TreeNode T) {
    Stack<TreeNode> stack = new Stack<TreeNode>();
    TreeNode p = T;
    while (p != null || !stack.isEmpty()) {
        if (p != null) {
            stack.push(p);
            p = p.left;
        } else {
            p = stack.pop();
            visit(p);
            p = p.right;
        }
    }
}

```

9 旋转数组找到指定数

二分查找，根据中间个数比目标的数大小，如果比目标数大，那么目标数在左边，如果比目标数小，此时目标数又比最右边的数小，就在中间数的右边，如果比最右边的数大，那肯定

在左边。

10 对单链表排序（快排、归并）

根据普通快排的思路,选择1个点为中心点,保证中心点左边比中心点小,中心点右边比中心点大即可.

单链表的实现为:

- 1.使第一个节点为中心点.
- 2.创建2个指针(p,q),p指向头结点,q指向p的下一个节点.
- 3.q开始遍历,如果发现q的值比中心点的值小,则此时p=p->next,并且执行当前p的值和q的值交换,q遍历到链表尾即可.
- 4.把头结点的值和p的值执行交换.此时p节点为中心点,并且完成1轮快排
- 5.使用递归的方法即可完成排序

```
public void quickSort(ListNode begin, ListNode end) {  
    if (begin == null || end == null || begin == end)  
        return;  
  
    ListNode first = begin;  
    ListNode second = begin.next;  
  
    int nMidValue = begin.val;  
  
    while (second != end.next && second != null) {  
        if (second.val < nMidValue) {  
            first = first.next;  
            int temp = first.val;  
            first.val = second.val;  
            second.val = temp;  
        }  
  
        second = second.next;  
    }  
  
    int temp = begin.val;  
    begin.val = first.val;  
    first.val = temp;  
  
    quickSort(begin, first);  
    quickSort(first.next, end);  
}
```

11 实现内存复制

12 字符串转换整数

```

public int myAtoi1(String str) {
    if (str == null) {
        return 0;
    }
    str = str.trim();
    if (str.length() == 0) {
        return 0;
    }
    int sign = 1;
    int index = 0;
    if (str.charAt(index) == '+') {
        index++;
    } else if (str.charAt(index) == '-') {
        sign = -1;
        index++;
    }
    long num = 0;
    for (; index < str.length(); index++) {
        if (str.charAt(index) < '0' || str.charAt(index) > '9')
            break;
        num = num * 10 + (str.charAt(index) - '0');
        if (num > Integer.MAX_VALUE) {
            break;
        }
    }
    if (num * sign >= Integer.MAX_VALUE) {
        return Integer.MAX_VALUE;
    }
    if (num * sign <= Integer.MIN_VALUE) {
        return Integer.MIN_VALUE;
    }
    return (int) num * sign;
}

```

13 排序电话号码（桶排序， bit 位）

1 基数排序 2 用桶排序，分到不同的桶可以用 treeset 维护

14 实现 MIN 栈加辅助栈，但是要优化辅助栈

15 有 N 个长度不一样的数组，所有数组中的元素都是从小到大有序排列的，请从大到小打印这 N 个数组整体的前 K 个数

利用大根堆 1.建立个数为 n 的堆（每个数组的最大数） 2.重复以下步骤 k 次 3.取堆顶 4.插入堆顶所在的下一个数（保存为类或者是联合体那种，有 value m n 三个元素） 5.如果数组到达了边界 插入负无穷。

16 数组长度是 $3n+1$, 有 n 个数出现了三次, 有一个数出现了一次, 找到一次的

```
public int singleNumberIII(int[] nums) {
    int ans = 0;
    for(int i = 0; i < 32; i++) {
        int sum = 0;
        for(int j = 0; j < nums.length; j++) {
            if((nums[j] >> i) & 1) == 1) {
                sum++;
                sum %= 3;
            }
        }
        if(sum != 0) {
            ans |= sum << i;
        }
    }
    return ans;
}

public int singleNumberII(int[] a) {
    int ones = 0, twos = 0;
    for (int i = 0; i < a.length; i++) {
        ones = (ones ^ a[i]) & ~twos;
        twos = (twos ^ a[i]) & ~ones;
    }
    return ones;
}
```

17 一排楼房，高矮不同，找到一个区间是面积最大，面积的高度就是区间里最矮的楼房的高度

先找到最矮的那个楼房，算出整个面积。然后递归左边找到最矮的，算出左边这个部分的整个面积，右边找到最矮的也算出这个部分的整个面积。

最优解：用两个栈来 A 存当前能计算的最小值和 B 存下标

比如：int[] a = {2 1 8 9 3 1 4}

首先 A 压入 2, B 压入 0, 因为 $a[1] < A.peek()$, 不压了，去计算面积， $j=B.pop(); S=A.pop()*(i-j)$; 跟 max 比较，max 小的就赋值给 max，循环条件是 A 非空并且现在 $a[i] < A.peek()$ ，不然这个 A.peek 还没到计算的时候。其实整个的意思就是找到一个高度，然后让他最矮的情况下尽可能的长度变长，才能取得最大面积。

```
public int largestRectangleArea1(int[] height) {
    if (height == null || height.length == 0)
        return 0;

    Stack<Integer> stHeight = new Stack<Integer>();
    Stack<Integer> stIndex = new Stack<Integer>();
    int max = 0;
```

```

        for (int i = 0; i < height.length; i++) {
            if (stHeight.isEmpty() || height[i] > stHeight.peek()) {
                stHeight.push(height[i]);
                stIndex.push(i);
            } else if (height[i] < stHeight.peek()) {
                int lastIndex = 0;
                while (!stHeight.isEmpty() && height[i] < stHeight.peek()) {
                    lastIndex = stIndex.pop();
                    int area = stHeight.pop() * (i - lastIndex);
                    if (max < area) {
                        max = area;
                    }
                }
                stHeight.push(height[i]);
                stIndex.push(lastIndex);
            }
        }
        while (!stHeight.isEmpty()) {
            int area = stHeight.pop() * (height.length - stIndex.pop());
            max = max < area ? area : max;
        }
    }
    return max;
}

```

15 开放性问题 海量数据

<http://www.educity.cn/wenda/389464.html>

1 给定 a、b 两个文件，各存放 50 亿个 url，每个 url 各占 64 字节，内存限制是 4G，找出 a，b 共同的 url

用 bitmap

方案 1：可以估计每个文件的大小为 $50G \times 64=320G$ ，远远大于内存限制的 4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

s 遍历文件 a，对每个 url 求取 $\text{hash(url)} \% 1000$ ，然后根据所取得的值将 url 分别存储到 1000

个小文件（记为 a_0, a_1, \dots, a_{999} ）中。这样每个小文件的大约为 300M。

s 遍历文件 b，采取和 a 相同的方式将 url 分别存储到 1000 各小文件（记为 b_0, b_1, \dots, b_{999} ）。

这样处理后，所有可能相同的 url 都在对应的小文件 ($a_0 \text{ vs } b_0, a_1 \text{ vs } b_1, \dots, a_0 \text{ vs } b_{999}$) 中，

不对应的小文件不可能有相同的 url。然后我们只要求出 1000 对小文件中相同的 url 即可。

s 求每对小文件中相同的 url 时，可以把其中一个小文件的 url 存储到 hash_set 中。然后遍历另一个小文件的每个 url，看其是否在刚才构建的 hash_set 中，如果是，那么就是共同的 url，存到文件里面就可以了。

方案 2：如果允许有一定的错误率，可以使用 Bloom filter，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用 Bloom filter 映射为这 340 亿 bit，然后挨个读取另外一个文件

的 url，检查是否与 Bloom filter，如果是，那么该 url 应该是共同的 url（注意会有一定的错误率）

2. 有 10 个文件，每个文件 1G，每个文件的每一行存放的都是用户的 query，每个文件的 query 都可能重复。要求你按照 query 的频度排序。

方案 1：

s 顺序读取 10 个文件，按照 $\text{hash}(\text{query}) \% 10$ 的结果将 query 写入到另外 10 个文件（记为 a_0, a_1, \dots, a_9 ）中。这样新生成的文件每个的大小大约也 1G（假设 hash 函数是随机的）。

s 找一台内存 2G 左右的机器，依次对 a_0, a_1, \dots, a_9 用 $\text{hash_map}(\text{query}, \text{query_count})$ 来统计每个 query 出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的 query 和对应的 query_count 输出到文件中。这样得到了 10 个排好序的文件（记为 b_0, b_1, \dots, b_{10} ）。

s 对 b_0, b_1, \dots, b_{10} 这 10 个文件进行归并排序（内排序与外排序相结合）。

方案 2：

一般 query 的总量是有限的，只是重复的次数比较多而已，可能对于所有的 query，一次性就可以加入到内存了。这样，我们就可以采用 trie 树/hash_map 等直接来统计每个 query 出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

方案 3：

与方案 1 类似，但在做完 hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如 MapReduce），最后再进行合并。

3. 有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 字节，内存限制大小是 1M。返回频数最高的 100 个词。

方案 1：顺序读文件中，对于每个词 x ，取 $\text{hash}(x) \% 5000$ ，然后按照该值存到 5000 个小文件（记为 $x_0, x_1, \dots, x_{4999}$ ）中。这样每个文件大概是 200k 左右。如果其中的有的文件超过了 1M 大小，还可以按照类似的方法继续往下分，知道分解得到的小文件的大小都不超过 1M。对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用 trie 树/hash_map 等），并取出出现频率最大的 100 个词（可以用含 100 个结点的最小堆），并把 100 词及相应的频率存入文件，这样又得到了 5000 个文件。下一步就是把这 5000 个文件进行归并（类似与

归并排序) 的过程了。

4. 海量日志数据，提取出某日访问百度次数最多的那个 IP。

方案 1：首先是这一天，并且是访问百度的日志中的 IP 取出来，逐个写入到一个大文件中。

注意到 IP 是 32 位的，最多有 2^{32} 个 IP。同样可以采用映射的方法，比如模 1000，把整个大文件映射为 1000 个小文件，再找出每个小文中出现频率最大的 IP（可以采用 hash_map 进行频率统计，然后再找出频率最大的几个）及相应的频率。然后再在这 1000 个最大的 IP 中，找出那个频率最大的 IP，即为所求。

5. 在 2.5 亿个整数中找出不重复的整数，内存不足以容纳这 2.5 亿个整数。

方案 1：采用 2-Bitmap（每个数分配 2bit，00 表示不存在，01 表示出现一次，10 表示多次，11 无意义）进行，共需内存 $2^{32} \times 2\text{bit} = 1\text{GB}$ 内存，还可以接受。然后扫描这 2.5 亿个整数，查看 Bitmap 中相对应位，如果是 00 变 01，01 变 10，10 保持不变。扫描完事后，查看 bitmap，把对应位是 01 的整数输出即可。

方案 2：也可采用上题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

6. 海量数据分布在 100 台电脑中，想个办法高校统计出这批数据的 TOP10。

方案 1：

s 在每台电脑上求出 TOP10，可以采用包含 10 个元素的堆完成（TOP10 小，用最大堆，TOP10 大，用最小堆）。比如求 TOP10 大，我们首先取前 10 个元素调整成最小堆，如果发现，然后扫描后面的数据，并与堆顶元素比较，如果比堆顶元素大，那么用该元素替换堆顶，然后再调整为最小堆。最后堆中的元素就是 TOP10 大。

s 求出每台电脑上的 TOP10 后，然后把这 100 台电脑上的 TOP10 组合起来，共 1000 个数据，再利用上面类似的方法求出 TOP10 就可以了。

7. 怎么在海量数据中找出重复次数最多的一个？

方案 1：先做 hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的

题）。

8. 上千万或上亿数据（有重复），统计其中出现次数最多的钱 N 个数据。

方案 1：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用 `hash_map`/搜索二叉树/红黑树等来进行统计次数。然后就是取出前 N 个出现次数最多的数据了，可以用第 6 题提到的堆机制完成。

9. 1000 万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

方案 1：这题用 `trie` 树（字典树）比较合适，`hash_map` 也应该能行。

10. 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前 10 个词，请给出思想，给出时间复杂度分析。

方案 1：这题是考虑时间效率。用 `trie` 树（字典树）统计每个词出现的次数，时间复杂度是 $O(n*le)$ (le 表示单词的平均长度)。然后是找出出现最频繁的前 10 个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是 $O(n*lg10)$ 。所以总的时间复杂度，是 $O(n*le)$ 与 $O(n*lg10)$ 中较大的哪一个。

11. 一个文本文件，找出前 10 个经常出现的词，但这次文件比较长，说是上亿行或十亿行，总之无法一次读入内存，问最优解。

方案 1：首先根据用 `hash 并求模`，将文件分解为多个小文件，对于单个文件利用上题的方法求出每个文件中 10 个最常出现的词。然后再进行归并处理，找出最终的 10 个最常出现的词。