JAVA基础

类加载器

双亲委派机制：

1. 防止重复加载（热加载时可能需要重复加载）
2. 沙箱安全机制，防止核心类被篡改

引导类加载器（Bootstrap classloader，负责加载 jre/lib下的核心类，rt.jar等，是扩展类加载器的父加载器）
扩展类加载器（extension classloader，负责加载 jre/lib/ext 下的类，1.9之后更名为PlatformClassLoader：平台类加载器，lib/modules目录下的class，是应用类加载器的父加载器）
应用类加载器（application classloader，负责加载 classpath 下的 class 字节码，主要是自己写的那些类，扩展类加载器和应用类加载器都继承ClassLoader）
tomcat打破双亲委派机制，为每一个webApp项目定义一个独立的WebAppClassLoader

volatile

- 可见性

让其他线程能够立即知道这个共享变量已经被修改了，当其他线程要读取这个变量的时候，最终会去内存中读取，而不是从自己的工作空间中读取
缓存一致性协议：
线程中的处理器会一直在总线上嗅探其内部缓存中的内存地址在其他处理器的操作情况，一旦嗅探到某处处理器打算修改其内存地址中的值，而该内存地址刚好也在自己的内部缓存中，那么处理器就会强制让自己对该缓存地址的无效。所以当该处理器要访问该数据的时候，由于发现自己缓存的数据无效了，就会去主存中访问。

```
class MyDate {
    //共享变量
    volatile int number = 0;
    public void change() {
        this.number = 60;
    }
}
public class VolatileDemo {
    public static void main(String[] args) {
        MyDate myDate = new MyDate();
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + " come in");
            try {
                TimeUnit.SECONDS.sleep(3);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //修改了number的值，当前线程的number值已经改变
            myDate.change();
            System.out.println(Thread.currentThread().getName() + " update number value=" + myDate.number);
        }, "A线程").start();
        //main线程读number，没有加volatile时，不知道number已修改，一直在循环，加上volatile后，值改变后会从主内存去读取，此时值为60，跳出循环
        while (myDate.number == 0) {
            //main 线程一直循环等待直到number值不等于0
        }
        System.out.println(Thread.currentThread().getName() + " over number value="+myDate.number);
    }
}
```

- **有序性(禁止指令重排)**

虚拟机会保证这个变量之前的代码一定会比它先执行，而之后的代码一定会比它慢执行。计算机在执行程序时，为了提高性能，编译器和处理器常常会对指令重排，一般分为编译器优化重排，指令并行的重排，内存系统的重排。

- **不保证原子性**

对任意单个volatile变量的读/写具有原子性，但**类似于i++这种复合操作不具有原子性**（基于这点，我们**通过会认为volatile不具备原子性**）。

```
public class VolatileDemo {
    //改造1: private static final Object object = new Object();
    //改造2: static AtomicInteger number = new AtomicInteger();
    volatile static int number = 0;
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    //改造1:
                    /*synchronized (object){
                        number++;
                    }*/
                    //改造2: number.getAndIncrement();
                    number++;
                }
```

```
            }, String.valueOf(i)).start();
        }
        // 等待线程计算完成后 由main线程取最终结果 10*1000 = 1万
        while (Thread.activeCount()>2){
            //mian线程和GC后台线程
            Thread.yield();
        }
        //结果并不是10000，有两种修改方法
        System.out.println(Thread.currentThread().getName()+" sum value="+number);
    }
}
```

AtomicInteger原理

自旋锁 + CAS 算法

```
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        //线程1、2同时进入到这，比如1先cas修改成功了，2的值和1修改后的值变得不一致了，cas失败，循环再次获取变量的值，AtomicInteger中的value用volatile修饰，所以获取的值就是1修改后的值，此时再次cas就会成功
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;
}
```

synchronized 和 volatile 的区别是什么?

- volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
- volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的。
- volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性。
- volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。
- volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化。

synchronized 和 Lock 有什么区别

- 首先synchronized是java内置关键字，在jvm层面，Lock是个java类；
- synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；

- synchronized会自动释放锁(a 线程执行完同步代码会释放锁 ； b 线程执行过程中发生异常会释放锁)，Lock需在finally中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；
- 用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去，而Lock锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了；
- synchronized有锁升级的过程
- synchronized的锁可重入、不可中断、非公平，而Lock锁可重入、可中断、可公平（两者皆可）；
- Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。

Lock接口
实现：ReentrantLock（ConcurrentHashMap 1.7有用到,,,,1.8使用synchronized+cas）
应用场景：
如果发现该操作已经在执行中则不再执行（有状态执行）
lock.tryLock();
如果发现该操作已经在执行，等待一个一个执行
new ReentrantLock(false);//不公平锁，允许插队
如果是公平锁，则会按照队列的顺序进行执行
如果是非公平锁，则会按优先级高的先执行
new ReentrantLock();//公平锁，一个一个来
lock.lock();
如果发现该操作已经在执行，则尝试等待一段时间，等待超时则不执行（尝试等待执行）
lock.tryLock(5,TimeUnit.SECONDS)

为什么说 Synchronized 是可重入锁

可重入性

可重入性是锁的一个基本要求，是为了解决自己锁死自己的情况。
比如下面的伪代码，一个类中的同步方法调用另一个同步方法，假如 Synchronized 不支持重入，进入 `m1` 方法时当前线程获得锁， `m1` 方法里面执行 `m2` 时当前线程又要去尝试获取锁，这时如果不支持重入，它就要等释放，把自己阻塞，导致自己锁死自己。

```
public synchronized static  void m1(){
    System.out.println("m1");
    m2();
}
private synchronized static void m2() {
    System.out.println("m2");
}
```

```
8  public static void main(String[] args) {
9      m1();
10 }
```

对 Synchronized 来说，可重入性是显而易见的，在执行 monitorenter 指令时，如果这个对象没有锁定，或者当前线程已经拥有了这个对象的锁（而不是已拥有了锁则不能继续获取），就把锁的计数器 + 1，其实本质上就通过这种方式实现了可重入性。

synchronized原理

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。
是非公平锁， 获取锁的行为并非是按照申请对象锁的先后时间分配锁的，每次对象锁被释放时，每个线程都有机会获得对象锁，这样有利于提高执行性能，但是也会造成线程饥饿现象。
Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的class对象
- 同步方法块，锁是括号里面的对象

同步代码块

synchronized关键字在经过Javac编译之后，会在同步块的前后形成monitorenter和monitorexit两个字节码指令

- 在执行monitorenter指令的时候，首先要去尝试获取对象的锁（**获取对象锁的过程，其实是获取monitor对象的所有权的过程**）。
- 如果这个对象没被锁定，或者当前线程已经持有了那个对象的锁，就把锁的计数器的值增加一。
- 而在执行monitorexit指令时会将锁计数器减一。一旦计数器的值为零，锁随即就被释放了。
- 如果获取对象锁失败，那当前线程就应当被阻塞等待，直到请求锁定的对象被持有它的线程释放为止。

同步方法

这个ACC_SYNCHRONIZED标志。代表的是当线程执行到方法后会检查是否有这个标志，如果有的话就会隐式的去调用monitorenter和monitorexit两个命令来将方法锁住。

monitor对象

获取对象锁的过程，其实是获取monitor对象的所有权的过程。哪个线程持有了monitor对象，那么哪个线程就获得了锁，获得了锁的对象可以重复的来获取monitor对象，但是同一个线程每获取一次monitor对象所有权锁计数就加一，在解锁的时候也是需要将锁计数减成0才算真的释放了锁。
monitor对象，我们其实在Java的反编译文件中并没有看到。这个对象是存放在对象头中的。

加锁过程

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁。但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级。

偏向锁

什么是偏向锁?

通过对大量数据的分析可以发现，大多数情况下锁竞争是不会发生的，往往是一个线程多次获得同一个锁，于是引入了偏向锁，偏向锁不会被刻意的释放，如果没有竞争，线程再次请求锁时可以直接获得锁。

上面在介绍对象头的时候，说到了对象头中包含的内容了，其中有一个就是偏向锁的线程ID，它代表的意思就是说，如果当一个线程获取到了锁之后，锁的标志计数器就会+1，并且把这个线程的id存储在锁住的这个对象的对象头上面。

这个过程是通过CAS来实现的，每次线程进入都是无锁的，当执行CAS成功后，直接将锁的标志计数+1（持有偏向锁的线程以后每次进入锁时不做任何操作，标志计数直接+1），这个时候其他线程再进来时，执行CAS就会失败，也就是获取锁失败。

轻量级锁

轻量级锁的性能介于偏向锁与重量级锁之间，在存在锁竞争的情况下，不需要让线程在阻塞与唤醒状态间切换

轻量级锁还是和对象头的第一部分（Mark Word）相关。

- 在代码即将进入同步块的时候，如果此同步对象没有被锁定，虚拟机首先将当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用户存储锁对象目前的Mark Word的拷贝。
- 然后JVM将使用CAS操作尝试把对象的Mark Word更新为指向Lock Record的指针。如果这个更新动作成功了，说明线程获取锁成功，并执行后面的同步操作。
- 如果这个更新动作失败了，说明锁对象已经被其他线程抢占了，那轻量级锁不在有效，必须膨胀为重量级锁。此时被锁住的对象的标志变为重量级锁的标志。

自旋锁

当轻量级锁获取失败后，就会升级为重量级锁，但是重量级锁之前也介绍了是很耗资源的，JVM开发团队注意到许多程序上，共享数据的锁定状态只会持续很短一段时间，为了这段时间去挂起和恢复线程并不值得。**所以想到了一个策略，那就是当线程请求一个已经被锁住的对象时，可以让未获取锁的线程“稍等一会”，但不放弃处理器执行时间，只需要让线程执行一个忙循环（自旋），这就是所谓**

的自旋锁。自旋锁在JDK1.4.2中引入，默认关闭，可以通过-XX:UserSpinning参数来开启，默认自旋次数是10次，用户可以自定义次数，配置参数是-XX:PreBockSpin。
无论是用户指定还是默认值的自旋次数，对JVM重所有的锁来说都是相同的。在JDK6中引入了自适应自旋，根据前一次在同一锁上的自旋时间及拥有者的状态来决定。如果上一次同一个对象自旋锁获得成功了，那么再次进行自旋时就会认为成功几率很大，那么自旋次数就会自动增加。反之如果自旋很少成功获得锁，那么以后这个自旋过程都有可能被省略掉。
这样在轻量级失败后，就会升级为自旋锁，如果自旋锁也失败了，那就只能是升级到重量级锁了。

Synchronized减重的过程，通常被称为锁膨胀或是锁升级的过程。主要步骤是：

- 先是通过偏向锁来获取锁，解决了虽然有同步但无竞争的场景下锁的消耗。
- 再是通过对象头的Mark Word来实现的轻量级锁，通过轻量级锁如果还有竞争，那么继续升级。
- 升级为自旋锁，如果达到最大自旋次数了，那么就直接升级为重量级锁，所有未获取锁的线程都阻塞等待。

可重入锁实现原理
可重入锁的原理：判断当前线程是否是持有锁的线程，如果是则无需要wait()，如果不是则等待持有锁的线程释放！

实现线程安全的懒汉式（双重检查加锁）

```
public class Singleton {
    private volatile static Singleton instance = null;
    // 私有化构造方法
    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }

        }
        return instance;
    }
}
```

Java引用类型

强引用：默认就是强引用，复制操作就是前引用，只要引用存在，对象就不会被回收
软引用：SoftReference，内存不足时会被回收
弱引用：WeakReference，只要垃圾回收执行，就会被回收，而不管是否内存不足
虚引用：PhantomReference，作用是跟踪垃圾回收器收集对象的活动

什么是线程安全

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量 的值也和预期的是一样的，就是线程安全的。

ThreadLocal

ThreadLocal不能使用原子类型，只能使用Object类型
ThreadLocal作为key存放在Thread的ThreadLocalMap中
为什么把ThreadLocal做为key，而不是Thread做为key？（所有的线程都去操作同一个Map，Map体积有可能会膨胀，导致访问性能的下降，而作为Thread的变量，只要线程销毁了，ThreadLocalMap也会被销毁）
ThreadLocal和Synchonized都用于解决多线程并发访问。但是ThreadLocal与synchronized有本质的区别。synchronized是利用锁的机制，使变量或代码块在某一时该只能被一个线程访问。而ThreadLocal为每一个线程都提供了变量的副本，使得每个线程在某一时间访问到的并不是同一个对象，这样就隔离了多个线程对数据的数据共享。而Synchronized却正好相反，它用于在多个线程间通信时能够获得数据共享。
内存泄漏问题（jvm认为对象还在引用，无法回收）
ThreadLocal内存泄露指的是：ThreadLocal被回收了，ThreadLocalMap Entry的key没有了指向，但Entry仍然有ThreadRef->Thread->ThreadLoalMap-> Entry value-> Object 这条引用一直存在导致内存泄露（如果一个ThreadLocal没有外部强引用来引用它，下一次系统GC时，这个ThreadLocal必然会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value）
get、set、remove等方法中，都会对key为null的Entry进行清除（expungeStaleEntry方法，将Entry的value清空，等下一次垃圾回收时，这些Entry将会被彻底回收）
当前线程一直在运行，并且一直不执行get、set、remove方法，这些key为null的Entry的value就会一直存在一条强引用链：Thread Ref -> Thread -> ThreadLocalMap -> Entry -> value，导致这些key为null的Entry的value永远无法回收，造成内存泄漏。

如何避免内存泄漏?
为了避免这种情况，我们可以在使用完ThreadLocal后，手动调用remove方法，以避免出现内存泄漏。
remove()方法

拿到当前线程的threadLocals属性，如果不为空，则将key为当前ThreadLocal的键值对移除，并且会调用expungeStaleEntry方法清除key为null的Entry

子线程访问父线程的ThreadLocal

```
public class Get_ThreadLocal {
    public static void main(String[] args) throws InterruptedException {
        Thread parentParent = new Thread(() -> {
            ThreadLocal<String> threadLocal = new ThreadLocal<>();
            threadLocal.set("a");
            InheritableThreadLocal<String> inheritableThreadLocal = new InheritableThreadLocal<>();
            inheritableThreadLocal.set("b");
            new Thread(() -> {
                System.out.println("threadLocal=" + threadLocal.get());
                System.out.println("inheritableThreadLocal=" + inheritableThreadLocal.get());
            }).start();
        }, "父线程");
        parentParent.start();
        //运行结果
        //threadLocal=null
        //inheritableThreadLocal=b
    }
}
```

InheritableThreadLocal：可继承的ThreadLocal

```
//线程构造函数
private Thread(ThreadGroup g, Runnable target, String name, long stackSize, AccessControlContext acc, boolean inheritThreadLocals) {
    //......
    //inheritThreadLocals默认true，设置了父线程的inheritableThreadLocals，就会把值也赋给当前线程的inheritableThreadLocals
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
            this.inheritableThreadLocals = ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    //......
}
```

InheritableThreadLocal 对ThreadLocal 的getMap()方法进行重写

createMap方法不仅创建了threadLocals，同时也将要添加的本地变量值添加到了threadLocals中。

InheritableThreadLocal类继承了ThreadLocal类，并重写了childValue、getMap、createMap方法。

其中createMap方法在被调用的时候，创建的是inheritableThreadLocal而不是threadLocals。

同理，getMap方法在当前调用者线程调用get方法的时候返回的也不是threadLocals而是inheritableThreadLocal。

乐观锁和悲观锁

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁，如synchronized
乐观锁：总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。如java.util.concurrent.atomic 使用的是CAS

CAS(compare and swap)

比较并交换：可以解决多线程并行情况下使用锁造成性能损耗的一种机制.CAS 操作包含三个操作数—内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。一个线程从主内存中得到num值，并对num进行操作，写入值的时候，线程会把第一次取到的num值和主内存中num值进行比较，如果相等，就会将改变后的num写入主内存，如果不相等，则一直循环对比，知道成功为止。会出现ABA现象，可以加版本号进行控制
应用：java.util.concurrent.atomic

CAS使用的时机

- 线程数较少、等待时间短可以采用自旋锁进行CAS尝试拿锁，较于synchronized高效。
- 线程数较大、等待时间长，不建议使用自旋锁，占用CPU较高

线程和进程有什么区别

线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

线程的状态

新建(NEW)：新创建了一个线程对象。

可运行(RUNNABLE)：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。
运行(RUNNING)：可运行状态(runnable)的线程获得了cpu 时间片（timeslice） ，执行程序代码。
阻塞(BLOCKED)：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

> (一). 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。 (二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。 (三). 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。

死亡(DEAD)：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

线程池

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时 候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短 的任务的程序的可扩展线程池）。
四种常用线程池

- newSingleThreadExexcutor：单线程数的线程池（核心线程数=最大线程数=1），阻塞队列 `LinkedBlockingQueue`
- newFixedThreadPool：固定线程数的线程池（核心线程数=最大线程数=自定义），阻塞队列 `LinkedBlockingQueue`
- newCacheThreadPool：可缓存的线程池（核心线程数=0，最大线程数=Integer.MAX_VALUE），阻塞队列 `SynchronousQueue`
- newScheduledThreadPool：支持定时或周期任务的线程池（核心线程数=自定义，最大线程数=Integer.MAX_VALUE），阻塞队列 `DelayedWorkQueue`

Executors各个方法的弊端

1）newFixedThreadPool和newSingleThreadExecutor:

主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至OOM。

2）newCachedThreadPool和newScheduledThreadPool:

> 主要问题是线程数最大数是Integer.MAX_VALUE，可能会创建数量非常多的线程，甚至OOM。

corePoolSize:核心线程数量。当线程数少于corePoolSize的时候，直接创建新的线程，尽管其他线程是空闲的。当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中。
maximunPoolSize：线程池最大线程数。只有在缓冲队列满了之后才会申请超过核心线程数的线程。当线程数量大于最大线程数且阻塞队列满了这时候就会执行一些策略来响应该线程。
workQueue：阻塞队列。存储等待执行的任务，会对线程池的运行产生很大的影响。当提交一个新的任务到线程池的时候，线程池会根据当前线程数量来选择不同的处理方式。

keepAliveTime：允许线程的空闲时间。当超过了核心线程数之外的线程在空闲时间到达之后会被销毁。
unit：keepAliveTime的时间单位。
threadFactory：线程工厂。用来创建线程，当使用默认的线程工厂创建线程的时候，会使得线程具有相同优先级，并且设置了守护性，同时也设置线程名称。
handler：拒绝策略。当workQueue满了，并且没有空闲的线程数，即线程达到最大线程数。就会有四种不同策略来处理

- 直接抛出异常（默认）
- 使用调用者所在线程执行任务
- 只要线程池没有关闭的话，丢弃阻塞队列 workQueue 中最老的一个任务，并将新任务加入
- 直接丢弃当前任务，什么也不做

核心线程数设置

Java中实现线程

在语言层面有三种方式。
java.lang.Thread 类的实例就是一个线程但是它需要调用java.lang.Runnable接口来执行，由于线程类本身就是调用的Runnable接口所以你可以继承 java.lang.Thread 类或者直接调用Runnable接口来重写run()方法实现线程。第三种 实现Callable<>接口并重写call方法。

Thread 类中的start() 和 run() 方法有什么区别

start()方法被用来启动新创建的线程，而且start()内部 调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启 动，start()方法才会启动新线程

```
1 public static void main(String[] args) throws Exception {
2     Thread thread = new Thread(()->
  System.out.println(Thread.currentThread().getName()));
3     thread.run();//输出 main
```

```
    thread.start();//输出 Thread-0
}
```

Runnable和Callable有什么不同

Runnable和Callable都代表那些要在不同的线程中执行的任务。Runnable从JDK1.0开始就有了，Callable是在 JDK1.5增加的。它们的主要区别是Callable的 call() 方法可以返回值和抛出异常，而Runnable的run()方法没有这些功能。Callable可以返回装载有计算结果的Future对象。

什么是FutureTask？Future 实现阻塞等待获取结果的原理

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

Future 实现阻塞等待获取结果的原理：AQS实现的阻塞，利用一个可见的state+cas+队列实现的

以FutureTask为例

```
public V get() throws InterruptedException, ExecutionException {
    int s = state;
    //如果线程没有完成就等待完成，然后返回结果
    if (s <= COMPLETING)
        //在内部，是通过lockSupport.park()方法去阻塞线程的
        s = awaitDone(false, 0L);
    return report(s);
}
private int awaitDone(boolean timed, long nanos)
        throws InterruptedException {
    long startTime = 0L;    // Special value 0L means not yet parked
    WaitNode q = null;
    boolean queued = false;
    for (;;) {
        int s = state;
        if (s > COMPLETING) {
            if (q != null)
                q.thread = null;
            return s;
        }
        else if (s == COMPLETING)
            // We may have already promised (via isDone) that we are done
            // so never return empty-handed or throw InterruptedException
```

```
            Thread.yield();
        else if (Thread.interrupted()) {
            removeWaiter(q);
            throw new InterruptedException();
        }
        else if (q == null) {
            if (timed && nanos <= 0L)
                return s;
            q = new WaitNode();
        }
        else if (!queued)
            queued = WAITERS.weakCompareAndSet(this, q.next = waiters, q);
        else if (timed) {
            final long parkNanos;
            if (startTime == 0L) { // first time
                startTime = System.nanoTime();
                if (startTime == 0L)
                    startTime = 1L;
                parkNanos = nanos;
            } else {
                long elapsed = System.nanoTime() - startTime;
                if (elapsed >= nanos) {
                    removeWaiter(q);
                    return state;
                }
                parkNanos = nanos - elapsed;
            }
            // nanoTime may be slow; recheck before parking
            if (state < COMPLETING)
                LockSupport.parkNanos(this, parkNanos);
        }
        else
            LockSupport.park(this);
    }
}
public void run() {
    if (state != NEW ||
        !RUNNER.compareAndSet(this, null, Thread.currentThread()))
        return;
```

```
63      try {
64          Callable<V> c = callable;
65          if (c != null && state == NEW) {
66              V result;
67              boolean ran;
68              try {
69                  result = c.call();
70                  ran = true;
71              } catch (Throwable ex) {
72                  result = null;
73                  ran = false;
74                  setException(ex);
75              }
76              if (ran)
77                  set(result);
78          }
79      } finally {
80          // runner must be non-null until state is settled to
81          // prevent concurrent calls to run()
82          runner = null;
83          // state must be re-read after nulling runner to prevent
84          // leaked interrupts
85          int s = state;
86          if (s >= INTERRUPTING)
87              handlePossibleCancellationInterrupt(s);
88      }
89  }
90  private void finishCompletion() {
91      // assert state > COMPLETING;
92      for (WaitNode q; (q = waiters) != null;) {
93          if (WAITERS.weakCompareAndSet(this, q, null)) {
94              for (;;) {
95                  Thread t = q.thread;
96                  if (t != null) {
97                      q.thread = null;
98                      LockSupport.unpark(t);
99                  }
100                 WaitNode next = q.next;
101                 if (next == null)
102                     break;
```

```
                q.next = null; // unlink to help gc
                q = next;
            }
            break;
        }
    }

    done();

    callable = null;        // to reduce footprint
}
private V report(int s) throws ExecutionException {
    Object x = outcome;
    if (s == NORMAL)
        return (V)x;
    if (s >= CANCELLED)
        throw new CancellationException();
    throw new ExecutionException((Throwable)x);
}

```

AQS

AQS，在java.util.concurrent.locks包中，AbstractQueuedSynchronizer这个类是并发包中的核心，了解其他类之前，需要先弄清楚AQS。

AQS就是一个同步器，要做的事情就相当于一个锁，所以就会有两个动作：一个是获取，一个是释放。获取释放的时候该有一个东西来记住他是被用还是没被用，这个东西就是一个状态。如果锁被获取了，也就是被用了，还有很多其他的要来获取锁，总不能给全部拒绝了，这时候就需要他们排队，这里就需要一个队列。

AQS的核心思想是：通过一个volatile修饰的int属性state代表同步状态，例如0是无锁状态，1是上锁状态。多线程竞争资源时，通过CAS的方式来修改state，例如从0修改为1，修改成功的线程即为资源竞争成功的线程，将其设为exclusiveOwnerThread，也称【工作线程】，资源竞争失败的线程会被放入一个FIFO的队列中并挂起休眠，当exclusiveOwnerThread线程释放资源后，会从队列中唤醒线程继续工作，循环往复。

notify 和 notifyAll有什么区别

这又是一个刁钻的问题，因为多线程可以等待单监控锁，Java API 的设计人员提供了一些方法当等待条件改变的时候通知它们，但是这些方法没有完全实现。notify()方法不能唤醒某个具体的线程，所以

只有一个线程在等 待的时候它才有用武之地。而notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。

为什么wait, notify 和 notifyAll这些方法不在thread类里面

一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁 就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

为什么wait和notify方法要在同步块中调用?

主要是因为Java API强制要求这样做，如果你不这么做，你的代码会抛出IllegalMonitorStateException异常。还有一个原因是为了避免wait和notify之间产生竞态条件。

如何避免死锁

Java多线程中的死锁

死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务，死锁的发生必须满足以下四个条件：

- 互斥条件：一个资源每次只能被一个进程使用。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。
- 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。

普通类和抽象类有哪些区别

- 普通类不能包含抽象方法，抽象类可以包含抽象方法。
- 抽象类不能直接实例化，普通类可以直接实例化。

接口和抽象类有什么区别

- 实现：抽象类的子类使用 extends 来继承；接口必须使用 implements 来实现接口。
- 构造函数：抽象类可以有构造函数；接口不能有。
- main 方法：抽象类可以有 main 方法，并且我们能运行它；接口不能有 main 方法。
- 实现数量：类可以实现很多个接口；但是只能继承一个抽象类。
- 接口的方法默认是 public ，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现），而抽象类可以有非抽象的方法
- 接口中除了 static 、 final 变量，不能有其他变量，而抽象类中则不一定

- 接口方法默认修饰符是 public ，抽象方法可以有 public 、 protected 和 default 这些修饰符（抽象方法就是为了被重写所以不能使用 private 关键字修饰！）。
- 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

1. 在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。

3. 在 jdk 7 或更早版本中，接口里面只能有常量变量和抽象方法。这些接口方法必须由选择实现接口的类实现。

4. jdk 8 的时候接口可以有默认方法和静态方法功能。

5. jdk 9 在接口中引入了私有方法和私有静态方法。

Files的常用方法都有哪些

- Files.exists()：检测文件路径是否存在。
- Files.createFile()：创建文件。
- Files.createDirectory()：创建文件夹。
- Files.delete()：删除一个文件或目录。
- Files.copy()：复制文件。
- Files.move()：移动文件。
- Files.size()：查看文件个数。
- Files.read()：读取文件。
- Files.write()：写入文件。

OSI 的七层模型

1. 应用层：网络服务与最终用户的一个接口。（http、ftp）
2. 表示层：数据的表示、安全、压缩。（TIFF、JPEG）
3. 会话层：建立、管理、终止会话。（smtp、rpc）
4. 传输层：定义传输数据的协议端口号，以及流控和差错校验。（tcp、udp）
5. 网络层：进行逻辑地址寻址，实现不同网络之间的路径选择。（ip）
6. 数据链路层：建立逻辑连接、进行硬件地址寻址、差错校验等功能。
7. 物理层：建立、维护、断开物理连接。

TCP、UDP

TCP：可靠有连接，提供全双工的通讯方式

UDP：无连接的不可靠服务

tcp三次握手、四次挥手

TCP的建立和释放都是由客户端发起的

握手

1. 由客户端向服务器发送的SYN数据包
2. 服务器在接收到客户端的SYN数据包后发送给客户端的回应数据包
3. 客户端收到服务器的SYN数据包后，会发送一个数据包来应答

客户端在第三次握手的时候，就可以携带应用层数据了。在发送完第三次握手数据包后，客户端就进入ESTABLISH（建立）状态。

当服务器接收到客户端的第三次握手数据包后，也会进入ESTABLISH状态。

当发送完三次握手数据包后，服务器和客户端都进入了ESTABLISH状态，也就代表双方正式建立了连接。

挥手

1. 由客户端发起断开请求，断开的是客户端向服务器写数据的方向连接，服务器还是能继续向客户端写入数据的
2. 当服务器接收到客户端的FIN数据包后，会返回确认应答包。
3. 第三次挥手是服务器向客户端释放连接发送的FIN数据包
4. 客户端向服务器发送的数据包，发送完之后，客户端进入TIME_WAIT状态

throw 和 throws 的区别

throws是用来声明一个方法可能抛出的所有异常信息，throws是将异常声明但是不处理，而是将异常往上传，谁调用我就交给谁处理。而throw则是指抛出的一个具体的异常类型

Java 中异常处理体系

1. 运行时异常：

都是RuntimeException类及其子类异常，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

2. 编译异常：

是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。

常见的编译异常： IOException（包括FileNotFoundException）、SQLException等以及用户自定义的Exception异常， 一般情况下不自定义检查异常。

常见的异常类

- NullPointerException：当应用程序试图访问空对象时，则抛出该异常。
- SQLException：提供关于数据库访问错误或其他错误信息的异常。
- IndexOutOfBoundsException：指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
- NumberFormatException：当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
- FileNotFoundException：当试图打开指定路径名表示的文件失败时，抛出此异常。
- IOException：当发生某种I/O异常时，抛出此异常。此类是失败或中断的I/O操作生成的异常的通用类。
- ClassCastException：当试图将对象强制转换为不是实例的子类时，抛出该异常。
- ArrayStoreException：试图将错误类型的对象存储到一个对象数组时抛出的异常。
- IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数。
- ArithmeticException：当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。
- NegativeArraySizeException：如果应用程序试图创建大小为负的数组，则抛出该异常。
- NoSuchMethodException：无法找到某一特定方法时，抛出该异常。
- SecurityException：由安全管理器抛出的异常，指示存在安全侵犯。
- UnsupportedOperationException：当不支持请求的操作时，抛出该异常。
- RuntimeExceptionRuntimeException：是那些可能在Java虚拟机正常运行期间抛出的异常的超类。

Java是值传递，还是引用传递

Java语言是**值传递。**Java 语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。

JVM 的内存分为堆和栈，其中**栈中存储了基本数据类型和引用数据类型实例的地址**，也就是对象地址。

而对象所占的空间是在堆中开辟的，所以传递的时候可以理解为把变量存储的对象地址给传递过去，因此引用类型也是值传递。

基本数据类型存在栈还是堆

深拷贝和浅拷贝

- 浅拷贝：仅拷贝被拷贝对象的成员变量的值，也就是基本数据类型变量的值，和引用数据类型变量的地址值，而对于引用类型变量指向的堆中的对象不会拷贝。
- 深拷贝：完全拷贝一个对象，拷贝被拷贝对象的成员变量的值，堆中的对象也会拷贝一份。

深拷贝是安全的，浅拷贝的话如果有引用类型，那么拷贝后对象，引用类型变量修改，会影响原对象。

> 浅拷贝如何实现呢？

Object类提供的clone()方法可以非常简单地实现对象的浅拷贝。

> 深拷贝如何实现呢？

- 重写克隆方法：重写克隆方法，引用类型变量单独克隆，这里可能会涉及多层递归。
- 序列化：可以先讲原对象序列化，再反序列化成拷贝对象。

final关键字

final表示不可变的意思，可用于修饰类、属性和方法：

- 被final修饰的类不可以被继承
- 被final修饰的方法不可以被重写
- 被final修饰的变量不可变，被final修饰的变量必须被显式第指定初始值，还得注意的是，这里的不可变指的是变量的引用不可变，不是引用指向的内容的不可变。

创建对象有哪几种方式

- new创建新对象
- 通过反射机制

```
Class<Test> clazz = Test.class;
Test test1 = clazz.newInstance();
Test test2 = clazz.getConstructor().newInstance();
```

- 采用clone机制
- 通过序列化机制

前两者都需要显式地调用构造方法。对于clone机制,需要注意浅拷贝和深拷贝的区别，对于序列化机制需要明确其实现原理，在Java中序列化可以通过实现Externalizable或者Serializable来实现。

什么是反射

反射主要是指程序可以访问、检测和修改它本身状态或行为的一种能力

Java反射：

在Java运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法

Java反射机制主要提供了以下功能：

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的成员变量和方法。
- 在运行时调用任意一个对象的方法。

什么是 java 序列化？什么情况下需要序列化

简单说就是为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来。虽然你可以用你自己的各种各样的方法来保存object states，但是Java给你提供一种应该比你自己好的保存对象状态的机制，那就是序列化。

什么情况下需要序列化：

a）当你想把的内存中的对象状态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；

几种序列化方式

- Java对象流列化 ： Java原生序列化方法即通过Java原生流(`InputStream` 和 `OutputStream` 之间的转化)的方式进行转化，一般是对象输出流 `ObjectOutputStream` 和对象输入流 `ObjectInputStream` 。
- Json序列化：这个可能是我们最常用的序列化方式，Json序列化的选择很多，一般会使用jackson包，通过ObjectMapper类来进行一些操作，比如将对象转化为byte数组或者将json串转化为对象。
- ProtoBuff序列化：ProtocolBuffer是一种轻便高效的结构化数据存储格式，ProtoBuff序列化对象可以很大程度上将其压缩，可以大大减少数据传输大小，提高系统性能。

switch 是否能作用在 byte/long/String上

Java5 以前 switch(expr)中，expr 只能是 byte、short、char、int。

从 Java 5 开始，Java 中引入了枚举类型， expr 也可以是 enum 类型。

从 Java 7 开始，expr还可以是字符串(String)，但是长整型(long)在目前所有的版本中都是不可以的。

String str1 = new String("abc")和String str2 = "abc" 的区别

两个语句都会去字符串常量池中检查是否已经存在 “abc” ，如果有则直接使用，如果没有则会在常量池中创建 “abc” 对象。

但是不同的是，String str1 = new String("abc") 还会通过 new String() 在堆里创建一个 "abc" 字符串对象实例。所以后者可以理解为被前者包含。

> String s = new String(″abc″)创建了几个对象？

很明显，一个或两个。如果字符串常量池已经有“abc”，则是一个；否则，两个。

当字符创常量池没有 “abc”，此时会创建如下两个对象：

- 一个是字符串字面量 "abc" 所对应的、字符串常量池中的实例
- 另一个是通过 new String() 创建并初始化的，内容与"abc"相同的实例，在堆中。

java 中操作字符串都有哪些类？它们之间有什么区别

操作字符串的类有：String、StringBuffer(方法加上了synchronized)、StringBuilder。
String 和 StringBuffer、StringBuilder 的区别在于 String 声明的是不可变的对象，每次操作都会生成新的 String 对象，然后将指针指向新的 String 对象，而 StringBuffer、StringBuilder 可以在原有对象的基础上进行操作，所以在经常改变字符串内容的情况下最好不要使用 String。
StringBuffer 和 StringBuilder 最大的区别在于，StringBuffer 是线程安全的(方法使用了synchronized关键字)，而 StringBuilder 是非线程安全的，但 StringBuilder 的性能却高于 StringBuffer，所以在单线程环境下推荐使用 StringBuilder，多线程环境下推荐使用 StringBuffer。

String不是不可变类吗？字符串拼接是如何实现的

String的确是不可变的，“+”的拼接操作，其实是会生成新的对象。
例如：

```
String a = "hello ";
String b = "world!";
String ab = a + b;
```

在**jdk1.8之前**，a和b初始化时位于字符串常量池，ab拼接后的对象位于堆中。经过拼接新生成了String对象。如果拼接多次，那么会生成多个中间对象。
在**Java8时**JDK对“+”号拼接进行了优化，上面所写的拼接方式会被优化为基于StringBuilder的append方法进行处理。Java会在编译期对“+”号进行处理。
也就是说其实上面的代码其实相当于：

```
String a = "hello ";
String b = "world!";
StringBuilder sb = new StringBuilder();
sb.append(a);
sb.append(b);
String ab = sb.toString();
```

此时，如果再笼统的回答：通过加号拼接字符串会创建多个String对象，因此性能比StringBuilder差，就是错误的了。因为本质上加号拼接的效果最终经过编译器处理之后和StringBuilder是一致的。
当然，循环里拼接还是建议用StringBuilder，为什么，因为循环一次就会创建一个新的StringBuilder对象

JDK1.8都有哪些新特性

java8接口为什么引入默认方法

假设我们正在使用某些接口，并在该接口中实现了所有抽象方法，后来又添加了新方法。
然后，除非您在每个类中实现新添加的方法，否则使用此接口的所有类将无法工作。
为了从Java8解决此问题，引入了默认方法。

Steam操作

```
Stream<Integer> stream = Stream.of(1,2,3,4,5,6);
stream.forEach(System.out::println);

```

流的中间操作
filter：过滤流中的某些元素
limit(n)：获取n个元素
skip(n)：跳过n元素，配合limit(n)可实现分页
distinct：通过流中元素的 hashCode() 和 equals() 去除重复元素
map： 接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
flatMap： 接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。
sorted()：自然排序，流中元素需实现Comparable接口
sorted(Comparator com)：定制排序，自定义Comparator排序器
allMatch： 接收一个 Predicate 函数，当流中每个元素都符合该断言时才返回true，否则返回false
noneMatch： 接收一个 Predicate 函数，当流中每个元素都不符合该断言时才返回true，否则返回false
anyMatch： 接收一个 Predicate 函数，只要流中有一个元素满足该断言则返回true，否则返回false
findFirst： 返回流中第一个元素 findAny：返回流中的任意元素 count：返回流中元素的总个数
max： 返回流中元素最大值
min： 返回流中元素最小值

泛型

Java 泛型（generics）是 JDK 5 中引入的一个新特性, 泛型提供了**编译时类型安全检测机制**，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

- ？ 表示不确定的 java 类型
- T (type) 表示具体的一个 java 类型

- K V (key value) 分别代表 java 键值中的 Key Value
- E (element) 代表 Element

泛型擦除

主要是为了向下兼容，因为JDK5之前是没有泛型的，为了让JVM保持向下兼容，就出了类型擦除这个策略。

简述 tcp 和 udp的区别

- TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接。
- TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付。
- Tcp通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。
- UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。
- 每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信。
- TCP对系统资源要求较多，UDP对系统资源要求较少。

什么是 XSS 攻击，如何避免

XSS攻击又称CSS,全称Cross Site Script （跨站脚本攻击），其原理是攻击者向有XSS漏洞的网站中输入恶意的 HTML 代码，当用户浏览该网站时，这段 HTML 代码会自动执行，从而达到攻击的目的。
XSS 攻击类似于 SQL 注入攻击，SQL注入攻击中以SQL语句作为用户输入，从而达到查询/修改/删除数据的目的，而在xss攻击中，通过插入恶意脚本，实现对用户游览器的控制，获取用户的一些信息。
XSS是 Web 程序中常见的漏洞，XSS 属于被动式且用于客户端的攻击方式。
XSS防范的总体思路是：对输入(和URL参数)进行过滤，对输出进行编码。

session 和 cookie 有什么区别

- 由于HTTP协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是Session.典型的场景比如购物车，当你点击下单按钮时，由于HTTP协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的Session，用用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。这个Session是保存在服务端的，有一个唯一标识。在服务端保存Session的方法很多，内存、数据库、文件都有。集群的时候也要考虑Session的转移，在大型的网站，一般会有专门的Session服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的，使用一些缓存服务比如Memcached之类的来放 Session。
- 思考一下服务端如何识别特定的客户？这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用 Cookie 来实现Session跟踪

的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，需要在 Cookie 里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了。有人问，如果客户端的浏览器禁用了 Cookie 怎么办？一般这种情况下，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户。

- Cookie其实还可以用在一些方便用户的场景下，设想你某次登陆过一个网站，下次登录的时候不想再次输入账号了，怎么办？这个信息可以写到Cookie里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户。这也是Cookie名称的由来，给用户的一点甜头。所以，总结一下：Session是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；Cookie是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现Session的一种方式。

有状态与无状态

有状态对象与无状态对象

1、有状态就是有数据存储功能。有状态对象(Stateful Bean)，就是有实例变量的对象，可以保存数据，是非线程安全的。在不同方法调用间不保留任何状态。

2、无状态就是一次操作，不能保存数据。无状态对象(Stateless Bean)，就是没有实例变量的对象.不能保存数据，是不变类，是线程安全的。

有状态服务与无状态服务

对服务器程序来说，究竟是有状态服务，还是无状态服务，其判断依旧是指两个来自相同发起者的请求在服务器端是否具备上下文关系。

1、有状态服务。

是指程序在执行过程中生成的中间数据。服务器端一般都要保存请求的相关信息，每个请求可以默认地使用以前的请求信息。

2、无状态服务。

是指容器在运行时，不在容器中保存任何数据，而将数据统一保存在容器外部，比如数据库中。服务器端所能够处理的过程必须全部来自于请求所携带的信息，以及其他服务器端自身所保存的、并且可以被所有请求所使用的公共信息。

final、finally、finalize 有什么区别

- final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。
- finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。

- finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用System的gc()方法的时候，由垃圾回收器调用finalize(),回收垃圾。

静态代理

（1）代理类是自己手工实现的，自己创建一个Java类，表示代理类。

（2）同时你所要代理的目标是确定的。

动态代理

动态代理：

当想要给实现了某个接口的类中的方法，加一些额外的处理。比如说加日志，加事务等。可以给这个类创建一个代理，故名思议就是创建一个新的类，这个类不仅包含原来类方法的功能，而且还在原来的基础上添加了额外处理的新类。这个代理类并不是定义好的，是动态生成的。具有解耦意义，灵活，扩展性强。

动态代理的应用：

- Spring的AOP
- 加事务
- 加权限
- 加日志

怎么实现动态代理

首先必须定义一个接口，还要有一个InvocationHandler(将实现接口的类的对象传递给它)处理类。再有一个工具类Proxy(习惯性将其称为代理类，因为调用他的newInstance()可以产生代理对象,其实他只是一个产生代理对象的工具类）。利用到InvocationHandler，拼接代理类源码，将其编译生成代理类的二进制码，利用加载器加载，并将其实例化产生代理对象，最后返回。

首先JDK Proxy提供interface列表，而cglib提供superclass供代理类继承，本质上都是一样的，就是提供这个代理类的签名，也就是对外表现为什么类型。

然后是一个方法拦截器，JDK Proxy里是InvocationHandler，而cglib里一般就是MethodInterceptor，所有被代理的方法的调用是通过它们的invoke和intercept方法进行转接的，AOP的逻辑也是在这一层实现。

```
class Handler implements InvocationHandler {
  private final Real real;

  public Handler(Real real) {
    this.real = real;
  }

  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
```

```
      throws IllegalAccessException, IllegalArgumentException,
             InvocationTargetException {
    System.out.println("=== BEFORE ===");
    Object re = method.invoke(real, args);
    System.out.println("=== AFTER ===");
    return re;
  }
}

Handler handler = new Handler(new Real());
ifc p = (ifc)Proxy.newProxyInstance(ifc.class.getClassLoader(), new Class[]
{ifc},handler);
p.add(1, 2);
```

```
public class Interceptor implements MethodInterceptor {
  @Override
  public Object intercept(Object obj,
                          Method method,
                          Object[] args,
                          MethodProxy proxy) throws Throwable {
      System.out.println("=== BEFORE ===");
      Object re = proxy.invokeSuper(obj, args);
      System.out.println("=== AFTER ===");
      return re;
  }
}
public static void main(String[] args) {
  Enhancer eh = new Enhancer();
  eh.setSuperclass(Real.class);
  eh.setCallback(new Interceptor());

  Real r = (Real)eh.create();
  int result = r.add(1, 2);
}
```

Java动态代理和cglib

区别

JDK动态代理只能对实现了接口的类生成代理，而不能针对类，java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

Cglib是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法，并覆盖其中方法的增强，因为采用的是继承，所以该类或方法最好不要生成final，对于final类或方法，是无法继承的

为什么需要cglib

cglib代理，使用cglib就是为了弥补动态代理的不足【动态代理的目标对象一定要实现接口】

Spring如何选择是用JDK还是cglib

1、当bean实现接口时，会用JDK代理模式

2、当bean没有实现接口，用cglib实现

3、可以强制使用cglib（在spring配置中加入<aop:aspectj-autoproxy proxyt-target-class="true"/>）

IO和NIO学习

BIO（Blocking IO 同步阻塞）

NIO（Non-blocking IO 同步非阻塞）mina、Netty

AIO（Asynchronous 异步非阻塞）t-io

一个经典生活的例子：

- 小明去吃同仁四季的椰子鸡，就这样在那里排队，等了一小时，然后才开始吃火锅。(BIO)
- 小红也去同仁四季的椰子鸡，她一看要等挺久的，于是去逛会商场，每次逛一下，就跑回来看看，是不是轮到她了。于是最后她既购了物，又吃上椰子鸡了。（NIO）
- 小华一样，去吃椰子鸡，由于他是高级会员，所以店长说，你去商场随便逛会吧，等下有位置，我立马打电话给你。于是小华不用干巴巴坐着等，也不用每过一会儿就跑回来看有没有等到，最后也吃上了美味的椰子鸡（AIO）

BIO中每一个连接都需要分配一个线程来执行，假如A客户端连接了服务器，但是还没有发送消息，这个时候B客户端向服务器发送连接请求，这个时候服务器是没有办法处理B客户端的连接请求的。

因为一个线程处理了一个客户端的连接后就阻塞住，并等待处理该客户端发送过来的数据。处理完该客户端的数据后才能处理其他客户端的连接请求。

那你这个是只有一个线程的时候，那我弄多个线程不就好了，来一个请求连接我弄一个线程?

那假如有一万个连接请求同时过来，那你开启一万个线程服务端不就崩了嘛

那我弄一个线程池呢，我最大线程数最多弄500呢?

那假如有500线程只请求连接，并不发送数据呢，那你这个线程池不也一样废了吗。这500个请求连接上了还没有发送数据，那么线程池的500个线程就没办法去处理别的请求，这样照样废了。

那咋办呢?

可以使用NIO同步非阻塞，这样就不需要很多线程，一个线程也能处理很多的请求连接和请求数据。NIO会将获取的请求连接放入到一个数组中，然后再遍历这个数据查看这些连接有没有数据发送过来。

但是有个问题啊，如果B和C只连接了，但是一直没有发送数据，那每次还循环判断他俩有没有发送数据的请求是不是有点多余了，能不能在我知道B和C肯定发送了数据的情况下再去遍历他呢?
可以引入Epoll，在JDK1.5开始引入了epoll通过事件响应来优化NIO，原理是客户端的每一次连接和每一次发送数据都看作是一个事件，每次发生事件会注册到服务端的一个集合中去，然后客户端只需要遍历这个集合就可以了。
那AIO有什么特点呢
AIO是异步非阻塞，他对于客户端的连接请求和发送数据请求是用不同的线程来处理的，他是通过回调来通知服务端程序去启动线程处理，适用于长连接的场景。

HashMap 和 Hashtable 有什么区别

- hashMap去掉了HashTable 的contains方法，但是加上了containsValue（）和containsKey（）方法。
- hashTable同步的，而HashMap是非同步的，效率上比hashTable要高。
- hashMap允许空键值，而hashTable不允许。

ConcurrentMap也是线程安全的，推荐使用ConcurrentMap

HashMap

如何减少hash冲突
1.**扰动函数可以减少碰撞**，原理是如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这就意味着存链表结构减小，这样取值的话就不会频繁调用equal方法，这样就能提高HashMap的性能。（扰动即Hash方法内部的算法实现，目的是让不同对象返回不同hashcode。）
2.**使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法**的话，将会减少碰撞的发生。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。为什么String, Interger这样的wrapper类适合作为键？因为String是final的，而且已经重写了equals()和hashCode()方法了。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象

为什么哈希表的容量一定要是2的整数次幂
首先，为了通过hash值确定元素在数组中存的位置，我们需要进行如下操作hash%length，当时%操作比较耗时间，所以优化为 hash & (length - 1)，当length为2的n次方时，hash & (length - 1) =hash % length；

其次，length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，因此，length取2的整数次幂，是**为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列**

实现原理

默认长度是16，负载因子是0.75，扩容的阈值=数组长度 * 负载因子

1. 负载因子越小，容易扩容，浪费空间，但查找效率高
2. 负载因子越大，不易扩容，对空间的利用更加充分，查找效率低（链表拉长）

1. jdk1.7的HashMap是用数组+链表实现的
2. jdk1.8的HashMap是用数组+链表+红黑树实现的

1.7插入链表使用的是头插法（扩容时，链表在复制的时候会反转，多线程会发生死循环）

1.8使用的尾插法（多线程在链表转树时会发生死循环）

HashMap在发生hash冲突的时候用的是链地址法，解决hash冲突并不只有这一种方法，常见的有如下四种方法

1. 开放定址法（一旦发生了冲突，就去寻找下一个空的散列地址）
2. 链地址法
3. 再哈希法（有多个不同的Hash函数，当发生冲突时，使用第二个，第三个等哈希函数计算地址，直到无冲突）
4. 公共溢出区域法（将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表）

HashMap在什么条件下扩容

jdk1.7

1. 超过扩容的阈值
2. 发生碰撞

jdk1.8

1. 超过扩容的阈值

JDK1.8发生了哪些变化

由数组+链表改为数组+链表+红黑树，当链表的长度超过8时，链表变为红黑树。

为什么要这么改?

我们知道链表的查找效率为O(n)，而红黑树的查找效率为O（logn），查找效率变高了。
为什么不直接用红黑树?
因为红黑树的查找效率虽然变高了，但是插入效率变低了，如果从一开始就用红黑树并不合适。从概率学的角度选了一个合适的临界值为8
优化了hash算法
计算元素在新数组中位置的算法发生了变化，新算法通过新增位判断oldTable[i]应该放在newTable[i]还是newTable[i+oldTable.length]
头插法改为尾插法，扩容时链表没有发生倒置（避免形成死循环）

HashMap在高并发下会发生什么问题

1. 多线程扩容，会让链表形成环，从而造成死循环
2. 多线程put可能导致元素丢失

什么情况需要重写hashcode方法

一般在需要重写equals方法时需要同时重写hashcode方法，自定义对象的equals，两个对象equals true，hashcode必须为true

ConcurrentHashMap

为什么不让插入null?
多线程下到底是插入的null还是没有值返回的null，会产生歧义

1.7

put操作

1. 首先是通过 key 定位到 Segment，之后在对应的 Segment 中进行具体的 put。
2. 虽然 HashEntry 中的 value 是用 volatile 关键词修饰的，但是并不能保证并发的原子性，所以 put 操作时仍然需要加锁处理。首先第一步的时候会尝试获取锁，如果获取失败肯定就有其他线程存在竞争，则利用 `scanAndLockForPut()` 自旋获取锁。1、尝试自旋获取锁。2、如果重试的次数达到了 `MAX_SCAN_RETRIES` 则改为阻塞锁获取，保证能获取成功。
3. 1、加锁操作；

2、遍历该 HashEntry，如果不为空则判断传入的 key 和当前遍历的 key 是否相等，相等则覆盖旧的 value。
3、为空则需要新建一个 HashEntry 并加入到 Segment 中，同时会先判断是否需要扩容。　　4、释放锁；

Get 操作

1、Key 通过 Hash 之后定位到具体的 Segment；2、再通过一次 Hash 定位到具体的元素上；3、由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是

最新值。
ConcurrentHashMap 的 get 方法是非常高效的，因为整个过程都不需要加锁。

1.8

结构改变
数组+链表改为数组+链表+红黑树

HashEntry 改为 Node

put操作
1、根据 key 计算出 hashcode，然后开始遍历 table；
2、判断是否需要初始化；
3、f 即为当前 key 定位出的 Node，如果为空表示当前位置可以写入数据，利用 CAS 尝试写入，失败则自旋保证成功。
4、如果当前位置的 hashcode == MOVED == -1,则需要进行扩容。
5、如果都不满足，则利用 synchronized 锁写入数据。
7、如果数量大于 TREEIFY_THRESHOLD 则要转换为红黑树。
get操作
根据计算出来的 hashcode 寻址，如果就在桶上那么直接返回值。 如果是红黑树那就按照树的方式获取值。 都不满足那就按照链表的方式遍历获取值。

SPI（Service Provider Interface）

SPI **将服务接口和具体的服务实现分离开来**，将服务调用方和服务实现者解耦，能够提升程序的扩展性、可维护性。修改或者替换服务实现并不需要修改调用方。
使用场景：
很多框架都使用了 Java 的 SPI 机制，比如：数据库加载驱动，日志接口，以及 dubbo 的扩展实现等等。
在 `MATE-INF/services` 下暴露对外使用的具体实现类

```
ServiceLoader.loadInstalled(c)
public static <S> ServiceLoader<S> loadInstalled(Class<S> service) {
    ClassLoader cl = ClassLoader.getPlatformClassLoader();
    return new ServiceLoader<>(Reflection.getCallerClass(), service, cl);
}
```

通过 SPI 机制能够大大地提高接口设计的灵活性，但是 SPI 机制也存在一些缺点，比如：

1. 遍历加载所有的实现类，这样效率还是相对较低的；
2. 当多个 ServiceLoader 同时 load 时，会有并发问题。

零拷贝

传统的IO模式

传统的IO模式，主要包括 read 和 write 过程：

- read：把数据从磁盘读取到内核缓冲区，再拷贝到用户缓冲区
- write：先把数据写入到 socket缓冲区，最后写入网卡设备

1. 用户空间的应用程序通过read()函数，向操作系统发起IO调用，上下文从用户态到切换到内核态，然后再通过 DMA 控制器将数据从磁盘文件中读取到内核缓冲区
2. 接着CPU将内核空间缓冲区的数据拷贝到用户空间的数据缓冲区，然后read系统调用返回，而系统调用的返回又会导致上下文从内核态切换到用户态
3. 用户空间的应用程序通过write()函数向操作系统发起IO调用，上下文再次从用户态切换到内核态；接着CPU将数据从用户缓冲区复制到内核空间的 socket 缓冲区（也是内核缓冲区，只不过是给socket使用），然后write系统调用返回，再次触发上下文切换
4. 最后异步传输socket缓冲区的数据到网卡，也就是说write系统调用的返回并不保证数据被传输到网卡

> 在传统的数据 IO 模式中，读取一个磁盘文件，并发送到远程端的服务，就共有四次用户空间与内核空间的上下文切换，四次数据复制，包括两次 CPU 数据复制，两次 DMA 数据复制。但两次 CPU 数据复制才是最消耗资源和时间的，这个过程还需要内核态和用户态之间的来回切换，而CPU资源十分宝贵，要拷贝大量的数据，还要处理大量的任务，如果能把 CPU 的这两次拷贝给去除掉，既能节省CPU资源，还可以避免内核态和用户态之间的切换。而零拷贝技术就是为了解决这个问题

> DMA（Direct Memory Access，直接内存访问）：DMA 本质上是一块主板上独立的芯片，允许外设设备直接与内存存储器进行数据传输，并且不需要CPU参与的技术

零拷贝

零拷贝指在进行数据 IO 时，数据在用户态下经历了零次 CPU 拷贝，并非不拷贝数据。通过减少数据传输过程中 内核缓冲区和用户进程缓冲区 间不必要的CPU数据拷贝 与 用户态和内核态的上下文切换次数，降低 CPU 在这两方面的开销，释放 CPU 执行其他任务，更有效的利用系统资源，提高传输效率，同时还减少了内存的占用，也提升应用程序的性能。

拷贝 一份相同的数据从一个地方移动到另外一个地方的过程，叫拷贝。

零 希望在IO读写过程中，CPU控制的数据拷贝到次数为0。

mmap + write 实现的零拷贝

在传统 IO 模式的4次内存拷贝中，与物理设备相关的2次拷贝（把磁盘数据拷贝到内存 以及 把数据从内存拷贝到网卡）是必不可少的。但与用户缓冲区相关的2次拷贝都不是必需的，如果内核在读取文件后，直接把内核缓冲区中的内容拷贝到 Socket 缓冲区，待到网卡发送完毕后，再通知进程，这样就可以减少一次 CPU 数据拷贝了。而 内存映射mmap 就是通过前面介绍的方式实现零拷贝的，它的核心就是操作系统把内核缓冲区与应用程序共享，将一段用户空间内存映射到内核空间，当映射成功后，用户对这段内存区域的修改可以直接反映到内核空间；同样地，内核空间对这段区域的修改也直接反映用户空间。正因为有这样的映射关系, 就不需要在用户态与内核态之间拷贝数据， 提高了数据传输的效率，这就是内存直接映射技术。

（1）用户应用程序通过 mmap() 向操作系统发起 IO调用，上下文从用户态切换到内核态；然后通过 DMA 将数据从磁盘中复制到内核空间缓冲区

（2）mmap 系统调用返回，上下文从内核态切换回用户态（这里不需要将数据从内核空间复制到用户空间，因为用户空间和内核空间共享了这个缓冲区）

（3）用户应用程序通过 write() 向操作系统发起 IO调用，上下文再次从用户态切换到内核态。接着 CPU 将数据从内核空间缓冲区复制到内核空间 socket 缓冲区；write 系统调用返回，导致内核空间到用户空间的上下文切换

（4）DMA 异步将 socket 缓冲区中的数据拷贝到网卡

mmap 的零拷贝 I/O 进行了4次用户空间与内核空间的上下文切换，以及3次数据拷贝；其中3次数据拷贝中包括了2次 DMA 拷贝和1次 CPU 拷贝。所以 mmap 通过内存地址映射的方式，节省了数据IO过程中的一次CPU数据拷贝以及一半的内存空间

sendfile 实现的零拷贝

只要我们的代码执行 read 或者 write 这样的系统调用，一定会发生 2 次上下文切换：首先从用户态切换到内核态，当内核执行完任务后，再切换回用户态交由进程代码执行。因此，如果想减少上下文切换次数，就一定要减少系统调用的次数，解决方案就是把 read、write 两次系统调用合并成一次，在内核中完成磁盘与网卡的数据交换。在 Linux 2.1 版本内核开始引入的 sendfile 就是通过这种方式来实现零拷贝的

（1）用户应用程序发出 sendfile 系统调用，上下文从用户态切换到内核态；然后通过 DMA 控制器将数据从磁盘中复制到内核缓冲区中

（2）然后CPU将数据从内核空间缓冲区复制到 socket 缓冲区

（3）sendfile 系统调用返回，上下文从内核态切换到用户态

（4）DMA 异步将内核空间 socket 缓冲区中的数据传递到网卡

通过 sendfile 实现的零拷贝I/O使用了2次用户空间与内核空间的上下文切换，以及3次数据的拷贝。其中3次数据拷贝中包括了2次DMA拷贝和1次CPU拷贝。那能不能将CPU拷贝的次数减少到0次呢？答

案肯定是有的，那就是 带 DMA 收集拷贝功能的 sendfile

带 DMA 收集拷贝功能的 sendfile 实现的零拷贝

Linux 2.4 版本之后，对 sendfile 做了升级优化，引入了 SG-DMA技术，其实就是对DMA拷贝加入了 scatter/gather 操作，它可以直接从内核空间缓冲区中将数据读取到网卡，无需将内核空间缓冲区的数据再复制一份到 socket 缓冲区，从而省去了一次 CPU拷贝。

1）用户应用程序发出 sendfile 系统调用，上下文从用户态切换到内核态；然后通过 DMA 控制器将数据从磁盘中复制到内核缓冲区中

（2）接下来不需要CPU将数据复制到 socket 缓冲区，而是将相应的文件描述符信息复制到 socket 缓冲区，该描述符包含了两种的信息：①内核缓冲区的内存地址、②内核缓冲区的偏移量

（3）sendfile 系统调用返回，上下文从内核态切换到用户态

（4）DMA 根据 socket 缓冲区中描述符提供的地址和偏移量直接将内核缓冲区中的数据复制到网卡

带有 DMA 收集拷贝功能的 sendfile 实现的 I/O 使用了2次用户空间与内核空间的上下文切换，以及2次数据的拷贝，而且这2次的数据拷贝都是非CPU拷贝，这样就实现了最理想的零拷贝I/O传输了，不需要任何一次的CPU拷贝，以及最少的上下文切换


需要注意的是，零拷贝有一个缺点，就是不允许进程对文件内容作一些加工再发送，比如数据压缩后再发送。

零拷贝应用场景

Java 的 NIO

1. mmap + write 的零拷贝方式

FileChannel 的 map() 方法产生的 MappedByteBuffer：FileChannel 提供了 map() 方法，该方法可以在一个打开的文件和 MappedByteBuffer 之间建立一个虚拟内存映射，MappedByteBuffer 继承于 ByteBuffer；该缓冲器的内存是一个文件的内存映射区域。map() 方法底层是通过 mmap 实现的，因此将文件内存从磁盘读取到内核缓冲区后，用户空间和内核空间共享该缓冲区。

```
1 public class MmapTest {

2     public static void main(String[] args) {
3         try {
4             FileChannel readChannel = FileChannel.open(Paths.get("./jay.txt"), StandardOpenOption.READ);
5             MappedByteBuffer data = readChannel.map(FileChannel.MapMode.READ_ONLY, 0, 1024 * 1024 * 40);
6             FileChannel writeChannel = FileChannel.open(Paths.get("./siting.txt"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);
7             //数据传输
```

```
            writeChannel.write(data);
            readChannel.close();
            writeChannel.close();
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

2. sendfile 的零拷贝方式

FileChannel 的 transferTo、transferFrom 如果操作系统底层支持的话，transferTo、transferFrom 也会使用 sendfile 零拷贝技术来实现数据的传输

```
@Override
public long transferFrom(FileChannel fileChannel, long position, long count) throws IOException {
   return fileChannel.transferTo(position, count, socketChannel);
}
```

```
public class SendFileTest {
    public static void main(String[] args) {
        try {
            FileChannel readChannel = FileChannel.open(Paths.get("./jay.txt"), StandardOpenOption.READ);
            long len = readChannel.size();
            long position = readChannel.position();

            FileChannel writeChannel = FileChannel.open(Paths.get("./siting.txt"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);
            //数据传输
            readChannel.transferTo(position, len, writeChannel);
            readChannel.close();
            writeChannel.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Netty 框架

1. 在网络通信上，Netty 的接收和发送 ByteBuffer 采用直接内存，使用堆外直接内存进行 Socket

读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接内存中（为什么拷贝？因为 JVM 会发生 GC 垃圾回收，数据的内存地址会发生变化，直接将堆内的内存地址传给内核，内存地址一旦变了就内核读不到数据了），然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

2. 在文件传输上，Netty 的通过 FileRegion 包装的 FileChannel.tranferTo 实现文件传输，它可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。
3. 在缓存操作上，Netty 提供了CompositeByteBuf 类，它可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf，避免了各个 ByteBuf 之间的拷贝。
4. 通过 wrap 操作，我们可以将byte[]数组、ByteBuf、ByteBuffer等包装成一个Netty ByteBuf对象，进而避免了拷贝操作。、
5. ByteBuf 支持 slice 操作，因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf，避免了内存的拷贝。

kafka

Kafka 的索引文件使用的是 mmap + write 方式，数据文件使用的是 sendfile 方式

ArrayList、 LinkedList 和 Vector

ArrayList底层是用数组实现的。可以认为ArrayList是一个可改变大小的数组。随着越来越多的元素被添加到ArrayList中，其规模是动态增加的。

LinkedList底层是通过双向链表实现的。所以，LinkedList和ArrayList之前的区别主要就是数组和链表的区别。

所以，LinkedList和ArrayList相比，增删的速度较快。但是查询和修改值的速度较慢。同时，LinkedList还实现了Queue接口，所以他还提供了offer(), peek(), poll()等方法。

Vector和ArrayList一样，都是通过数组实现的，但是Vector是线程安全的。和ArrayList相比，其中的很多方法都通过同步（synchronized）处理来保证线程安全。二者之间还有一个区别，就是扩容策略不一样。在List被第一次创建的时候，会有一个初始大小，随着不断向List中增加元素，当List认为容量不够的时候就会进行扩容。Vector缺省情况下自动增长原来一倍的数组长度，ArrayList增长原来的50%。

快速失败(fail-fast)和安全失败(fail-safe)

快速失败（fail—fast)

现象：在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则会抛出Concurrent Modification Exception(并发修改异常)。

原理：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变 modCount 的值。每当迭代器使用hashNext()/next()遍历

下一个元素之前，都会检测 modCount 变量是否为 expectedmodCount 值，是的话就返回遍历；否则抛出异常，终止遍历。(源码很清晰)

场景：java.util 包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）。

安全失败（fail-safe)

现象：在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则不会抛出 Concurrent Modification Exception。

原理：采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

场景：java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

```
public static void main(String[] args) throws Exception {

    // 创建集合对象
    CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>() ;
    list.add("张三") ;
    list.add("李四") ;
    list.add("王五") ;
    // 遍历
    Iterator<String> it = list.iterator();
    while(it.hasNext()) {
        String e = it.next();
        if(e.equals("王五")){
            list.remove(e) ;
        }
    }
    // 正常输出
    System.out.println(list);

    // 创建集合对象
    List<String> list2 = new ArrayList<>() ;
    list2.add("张三") ;
    list2.add("李四") ;
    list2.add("王五") ;
    // 遍历
    Iterator<String> it2 = list2.iterator();
    while(it2.hasNext()) {
        //抛异常 java.util.ConcurrentModificationException
```

```
        String e = it2.next();
        if(e.equals("王五")){
            list2.remove(e) ;
        }
    }
    //可以用removeIf
    //1.使用与list元素个数等长的 BitSet
    //2.使用过滤器，过滤出需要删除的元素，同时在 bitSet 相应的位打上1
    //3.此时知道没被删除的 元素个数 notDelNum
    //4.从0开始循环， 将list元素的基于位图为 0的元素位置 重新排列位置
    //5.将大于notDelNum的元素全部去掉
    list2.removeIf(e -> e.equals("王五"));
    // 输出
    System.out.println(list2);
}
```

List,Map,Set

存放时:

1.List以特定的索引(有顺序的存放)来存放元素,可以有重复的元素

2.Set存放元素是无序的,而且不可重复（用对象的**equals()**方法来区分元素是否重复）

3.Map保存键值对的映射,映射关系可以是一对一(键值)或者多对一,需要注意到的是：**键无序不可重复**,值可以重复

 HashSet底层就是HashMap

```
private transient HashMap<E,Object> map;
//Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
public HashSet() {
    map = new HashMap<>();
}
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

取出时:

(1)List取出元素for循环，foreach循环，Iterator迭代器迭代

(2)Set取出元素foreach循环，Iterator迭代器迭代

(3)Map取出元素需转换为Set，然后进行Iterator迭代器迭代，或转换为Entry对象进行Iterator迭代器迭代

ArrayList

底层就是Object[]
使用无参构造函数初始化 ArrayList 后，它当时的数组容量为 0
如果使用了无参构造函数来初始化 ArrayList， 只有当我们真正对数据进行添加操作 add 时，才会给数组分配一个默认的初始容量 DEFAULT_CAPACITY = 10
扩容发生在啥时候？那肯定是我们往数组中新加入一个元素但是发现数组满了的时候。
扩容后的数组长度 = 当前数组长度 + 当前数组长度 / 2。最后使用 Arrays.copyOf 方法直接把原数组中的数组 copy 过来，需要注意的是，Arrays.copyOf 方法会创建一个新数组然后再进行拷贝。

LinkedHashMap 的实现原理

LinkedHashMap 是通过哈希表和链表实现的，它通过 维护一个链表来保证对哈希表迭代时的有序性，而这个有序是指键值对插入的顺序。 另外，当向哈希表中重复插入某个键的时候，不会影响到原来的有序性。也就是说，假设你插入的键的顺序为 1 、 2 、 3 、 4 ，后来再次插入 2 ， 迭代时的顺序还是 1 、 2 、3、 4 ，而不会因为后来插入的 2 变成 1 、 3 、 4 、 2 。 (但其实我们可以改变它的规则，使它变成 1 、 3 、 4 、 2)
LinkedHashMap 的实现主要分两部分，一部分是哈希表，另外一部分是链表。哈希表部分继承了 HashMap ，拥有了 HashMap 那一套高效的操作，所以我们要看的就是 LinkedHashMap 中链表的部分，了解它是如何来维护有序性的。
LinkedHashMap 的大致实现如下图所示，当然链表和哈希表中相同的键值对都是指向同一个对象，这里把它们分开来画只是为了呈现出比较清晰的结构。

```
// Callbacks to allow LinkedHashMap post-actions
void afterNodeAccess(Node<K,V> p) { }
void afterNodeInsertion(boolean evict) { }
void afterNodeRemoval(Node<K,V> p) { }
```

三个方法表示的是在访问、插入、删除某个节点之后，进行一些处理，它们在 LinkedHashMap 都有各自的实现。 LinkedHashMap 正是通过重写这三个方法来保证链表的插入、删除的有序性。

为什么集合类没有实现Cloneable和Serializable接口

克隆(cloning)或者是序列化(serialization)的语义和含义是跟具体的实现相关的。因此，应该由集合类的具体实现来决定如何被克隆或者是序列化。集合类又是一个容器, 会装载不同的对象(各种各样的). 不可能每个对象都实现克隆,

Iterator 和 ListIterator 的区别

1. Iterator 可遍历 Set 和 List 集合； ListIterator 只能遍历 List。
2. Iterator 只能单向遍历；ListIterator 可双向遍历（向前/后遍历）。
3. ListIterator 继承自 Iterator 接口，添加新功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。

Iterator代表迭代器，是Collection框架中的一个接口；用于遍历集合元素。它允许逐个迭代集合中的每个元素，从集合中获取元素或从集合中删除元素；但无法使用Iterator修改集合中的任何元素。
ListIterator是Collection框架中的一个接口；是用于扩展Iterator接口的。使用ListIterator，可以向前和向后遍历集合的元素。还可以添加、删除或修改集合中的任何元素。简而言之，我们可以说它消除了Iterator的缺点。

Comparable接口和Comparator接口

Comparable（可比较的）接口和Comparator（比较器）接口都可以用来实现类的比较排序功能
Comparable接口会被某个类实现，用来表示该类的默认比较规则，通过这个比较规则进行做排序，不过如果此时Comparable接口的比较规则无法满足业务要求，需要再新增其他的排序规则，如果这些规则不是该类的默认比较规则，那么把它们继续加在compareTo()方法中就不符合需求了，太固定，同时编码也要尽量做到对修改关闭，对扩展开放
Comparator接口，它是一个工具类接口，可以根据业务需要，实现Comparator接口来定义不同规则的比较器，有几种业务就可以有几种比较器，这样扩展起来就很方便了。所以Comparable接口用作定义类的默认排序规则，而Comparator接口则用来扩展类的排序规则。

```
public class Student implements Comparable{
    int age;
    String name;

    public Student(int age) {
        this.age = age;
    }

    public Student(int age, String name) {
        this.age = age;
        this.name = name;
    }

    public Student() {
    }

    @Override
```

```
    public int compareTo(Object o) {
        if(o instanceof Student) {
            Student s = (Student) o;
            return this.age - ((Student) o).age;
        }else{
            throw new RuntimeException("传入的数据不一致");
        }

    }

    @Override
    public String toString() {
        return "Student{" +
                "age=" + age +
                ", name='" + name + '\'' +
                '}';
    }
}

public class TestStudent {
    public static void main(String[] args) {
        //容器
        List<Student> stu = new ArrayList<>();
        Student s1 = new Student(23,"ab");
        Student s2 = new Student(12,"bcdef");
        Student s3 = new Student(45,"cdd");
        stu.add(s1);
        stu.add(s2);
        stu.add(s3);
        Collections.sort(stu);
        for(Student s:stu){
            System.out.println(s);
        }
    }
```

```
public class Student{
    int age;
    String name;
    public Student(int age, String name) {
```

```
        this.age = age;
        this.name = name;
    }
    public Student() {
    }
    @Override
    public String toString() {
        return "Student{" +
                "age=" + age +
                ", name='" + name + '\'' +
                '}';
    }

}
public class TestStudent {
    public static void main(String[] args) {
        //容器
        List<Student> stu = new ArrayList<>();

        Student s1 = new Student(23,"aaaaa");
        Student s2 = new Student(12,"bb");
        Student s3 = new Student(45,"cddddddddd");

        stu.add(s1);
        stu.add(s2);
        stu.add(s3);

        Comparator c = new Comparator() {
            @Override
            public int compare(Object o1, Object o2) {
                if(o1 instanceof Student && o2 instanceof Student) {
                    Student s1 = (Student) o1;
                    Student s2 = (Student) o2;
                    return s1.name.length()-s2.name.length();
                }else{
                    throw new RuntimeException("传入的数据不一致");
                }
            }
        };
        Collections.sort(stu,c);
```

```
42          for(Student s:stu){
43              System.out.println(s);
44          }
45      }
46  }
```

过滤器、拦截器

拦截器(Interceptor)只对action请求起作用 即对外访问路径
而过滤器(Filter)则可以对几乎所有的请求都能起作用 包括css js等资源文件
拦截器(Interceptor)是在Servlet和Controller控制器之间执行
而过滤器(Filter)是在请求进入Tomcat容器之后 但是在请求进入Servlet之前执行

JVM

运行时数据区域

程序计数器

可以看作是当前线程所执行的字节码的行号指示器，通过改变计数器的值选取下一条需要执行的指令。
每个程序都有自己的程序计数器，互不影响，是“线程私有”的内存。
如果程序执行的是Java方法，记录的就是正在执行虚拟机字节码执行的地址，如果是本地方法，则是空，这是唯一一个没有OutOfMemoryError的区域

虚拟机栈 VM Stack

也是“线程私有”的，生命周期与线程相同，每个方法执行是都会创建一个栈帧（Stack Frame），用于存储局部变量表、操作数栈（）、动态连接（符号引用转为直接引用）、方法出口等信息，方法调用到执行完毕，就是栈帧入栈、出栈的过程。
局部变量表中存储空间以局部变量槽（Slot）表示，64为的long和double占两个槽，其余占一个。局部变量表所需的存储空间在编译时就确定了。
异常情况：如果请求的栈深度超过VM所允许的将抛出StackOverflowError；如果VM可以动态扩展，扩展时无法申请到足够内存抛出 OOM
hotspot虚拟机栈容量是不可以扩展的，所以不会由于栈扩展OOM

本地方法栈 Native Method Stack
和虚拟机栈相似，虚拟机栈是为java方法服务，本地方法栈为本地方法服务
HotSpot直接把两个合二为一了
异常情况也一样，也是“线程私有”

堆 Heap
线程所共享的一块区域，在虚拟机启动时创建，唯一的目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
是垃圾收集器管理的内存区域，也称为GC堆，Java堆既可以被实现为固定大小的，也可以是扩展的，不过主流实现都是按照扩展的（通过参数-Xmx（程序运行期间最大可占用的内存大小，超多会oom）和 -Xms（程序启动时占用内存大小）设定）

方法区 Method Area
各个线程共享的内存区域，用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存，有个别名是non-heap，目的是与堆区别开。

运行时常量池
Class文件中除了有类的版本、字段、方法、接口等描述信息，还有一项时常量池表，用于存放编译期生成的各种字面量和符号引用，这部分在加载后存放到方法区的运行时常量池 String.intern() 可以在运行期间把新的常量放入池中
（String + 实际调用的时StringBuilder的append方法）

直接内存 Direct Memory
并不是虚拟机运行时的一部分，也不是JVM规范中的内存区域。
主要时 jdk1.4之后引入的NIO，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用操作，也会导致OOM

JVM判断对象是否存活

计数法
会有循环引用问题

可达性分析
通过一系列 `GC roots` 对象作为起点搜索，不可达对象不等价于可回收对象，不可达对象变成可回收对象至少要经过两次标记过程，两次标记后仍是可回收对象，则面临回收。

GC roots 对象有

1、Java虚拟机栈中引用的对象

2、本地方法栈中JNI引用的对象

3、方法区中常量、类静态属性引用的对象

三色标记法

- 白色

对象尚未被垃圾收集器访问过（刚开始阶段所有对象都是白色（GC roots 是黑色），可达性分析结束后仍是白色，则表示不可达）

- 灰色

对象已经被垃圾收集器访问过，但这个对象至少还存在一个引用没有被扫描到

- 黑色

对象已经被垃圾收集器访问过，且这个对象所有引用都被扫描到（黑色对象不能不经过灰色对象直接指向白色对象）

三色标记法缺陷

1. 浮动垃圾

2. 漏标

解决

CMS（屏障+增量更新）

g1（写屏障+原始快照）

类的加载过程

加载（加载类文件到方法区，生成class对象）->**验证**（验证文件是否符合规范）->**准备**（为静态变量分配内存并设置变量初始值）->**解析**（Java虚拟机将常量池内的符号引用替换为直接引用的过程）->**初始化**（执行类中的Java程序代码）

gc

minor gc 、 full gc（会暂停用户线程STW，full gc时间会久点）

可达性分析，标记非垃圾

新生代：伊甸园 存活区（s0，s1）| 老年代
PermGen(永久代)空间是为长期对象保留的-大多数对象是ClassLoader加载的Class对象。除非在非常特殊的情况下(特别是当加载这些类的ClassLoader超出范围时)，否则不会对PermGen进行垃圾回收。
jdk1.8 Metaspace取代了永久代(PermGen)

调优

Arthas（阿尔萨斯） Alibaba开源的Java诊断工具
内存分配：计算业务对象的大小，每秒产生多少大小的对象，对象先伊甸园->存活区->老年代
老年代：年轻代 = 2：1
伊甸园:s0:s1 = 8：1：1
Xmn 设置年轻代大小，整个堆大小=年轻代大小 + **老年代大小** + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8
-XX:NewRatio=4:设置**年轻代**(包括Eden和两个Survivor区)与**年老代的比值**(除去持久代)。设置为4，则**年轻代与年老代所占比值为1:4**，年轻代占整个堆栈的1/5
-XX:SurvivorRatio=4:设置**年轻代中Eden区**与**Survivor区**的大小比值。设置为4，则**两个Survivor区**与**一个Eden区**的比值为2:4，**一个Survivor区**占**整个年轻代的1/6。**
-XX:MaxPermSize=256m:设置**持久代大小为256m。**
XX:MaxTenuringThreshold=0:设置**垃圾最大年龄。**如果设置为0的话，则**年轻代对象**不经过Survivor区，直接**进入年老代。** 对于**年老代比较多的应用**，可以提高效率。如果**将此值设置为一个较大值**，则年轻代对象会**在Survivor区进行多次复制**，这样可以增加对象**在年轻代的存活时间**，增加在**年轻代即被回收**的概率。
Xms 初始化Java堆的大小，超过后自动扩容到 Xmx（Java堆可以扩展到的最大值），一般设置成一样的，扩容内存抖动会影响程序稳定性
-Xss：规定了每个线程虚拟机栈及堆栈的大小，此配置将会影响此进程中并发线程数的大小

常见的几种内存溢出异常的原因及解决

堆内存溢出

java.lang.OutOfMemoryError: Java heap space
原因：

- 当堆内存不足，并且已经达到JVM设置的最大值，无法继续申请新的内存，存活的对象在堆内存中无法被回收，那么就会抛出该异常，表示堆内存溢出。
- 当一次从数据库查询大量数据，堆内存没有足够的内存可以存放大量的数据
- 大量的强引用对象在堆内存中存活，GC无法回收这些对象，新创建的对象在新生代无法进行分

配，Full GC仍然无法进行回收

解决：

- 查看当前JVM的堆内存配置是否太小，可以考虑增加堆内存大小（–Xms设置堆内存的初始值，–Xmx设置堆内存的最大值）
- 查看代码中是否有从数据库中一次加载大量数据的情况，或者代码中有大量强引用无法进行回收

栈内存溢出

java.lang.outOfMemoryError:StackOverFlow Error

原因：

- 线程请求的栈深度大于虚拟机允许的最大深度，抛出StackOverflowError
- 虚拟机在扩展栈时无法申请到足够的内存空间，抛出OutOfMemoryError

解决：

- 检查代码是否出现深度递归的情况，或者递归的终止条件没有设置
- 如果是线程的栈内存空间过小，则通过–Xss设置每个线程的栈内存空间（默认1M）

方法区和运行时常量池内存溢出

原因：

- 方法区存放的是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，内存溢出的原因可能是加载的类过多导致方法区没有足够的内存
- 如果程序中大量使用cglib或者动态代理等对目标类进行代理，那么在运行时会生成大量的代理类，如Spring、Hibernate等框架。所以生成的代理类过多导致方法区没有足够的内存

解决：

- 减少没有必要的Class加载，防止方法区内存溢出并且减少程序的编译时间
- 通过JVM参数设置方法区的大小，–XX:PermSize和–XX:MaxPermSize设置方法区的大小

本机直接内存溢出

Direct buffer memory

Java 允许应用程序通过 DirectByteBuffer 直接访问堆外内存，许多高性能程序通过 DirectByteBuffer 结合内存映射文件（Memory Mapped File）实现高速 IO

原因：

DirectByteBuffer 的默认大小为 64 MB，一旦使用超出限制，就会抛出 Direct buffer memory 错误。使用NIO则可能会出现该异常

解决：

DirectMemory的内存大小可以通过-XX:MaxDirectMemorySize指定，如果没有设置，则默认和Java堆最大值(-Xmx)一样

元空间内存溢出

Metaspace
Jdk8 之后使用元空间(metaspace)代替永久代，元空间和永久代最大的区别是元空间的内存使用的是本地内存，而永久代使用的是JVM的内存
永久代、元空间都是方法区的实现，方法区是规范
原因：
元空间中存储的是类信息、常量池、方法描述等信息，直接使用本地内存，当本地内存不足的时候，会抛出OutOfMemoryError:Metaspace异常
解决：
虽然元空间的内存不是由JVM控制，不过可以通过JVM参数来设置分配的内存空间的大小-XX:MaxMetaspaceSize配置参数

java性能调优工具

VisualVM
jps：用于查看当前用户下的java进程的pid及基本信息
jstack：查看进程的堆栈情况

```
//保存堆栈信息
jstack -l 17584 > E://17584-stacklog.txt
```

jmap：用来查看堆内存使用情况
使用jmap -heap pid查看进程堆内存使用状况，包括使用的GC算法、堆配置参数和各代中堆内存使用状况

```
//对进程ID为21711进行Dump
jmap -dump:format=b,file=/tmp/dump.dat 21711
```

System.gc()与Runtime.gc()

System.gc();就是呼叫java虚拟机的垃圾回收器运行回收内存的垃圾
每个 Java 应用程序都有一个 Runtime 类实例，使应用程序能够与其运行的环境相连接。可以通过 getRuntime 方法获取当前运行时。 Runtime.getRuntime().gc();
java.lang.System.gc()只是java.lang.Runtime.getRuntime().gc()的简写，两者的行为没有任何不同。
System.gc()是个静态方法，也就是说调用System.gc()方法，可以直接调用。而通过Runtime.gc()方法调用时，必须先通过getRuntime方法来得到Runtime实例，然后再调用gc方法，由此可见API说明文档中“方法 System.gc() 是调用此方法的一种传统而便捷的方式 ” 的说法还是很准确的” 。

如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存

不会，在下一个垃圾回收周期中，这个对象将是可被回收的。

什么是分布式垃圾回收（DGC）

DGC 叫做分布式垃圾回收。 RMI （远程方法调用）使用 DGC 来做自动垃圾回收。因为 RMI 包含了跨虚拟机的远程对象的引用，垃圾回收是很困难的。 DGC 使用引用计数算法来给远程对象提供自动内存管理。

RMI (Remote Method Invocation) 模型是一种分布式对象应用，使用 RMI 技术可以使一个 JVM 中的对象，调用另一个 JVM 中的对象方法并获取调用结果。这里的另一个 JVM 可以在同一台计算机也可以是远程计算机。因此，RMI 意味着需要一个 Server 端和一个 Client 端。

Minor GC和Major GC

堆内存在大的结构上分为：年轻代和年老代。其中年轻代又分为Eden区和Survivor区。Survivor区又分为两个相等的区域，一个是fromspace区，另外一个是tospace区。年轻代内存=Eden+其中一个Survivor区，也就是说两个Survivor区，虚拟机只使用了其中一个。

Minor GC：简单理解就是发生在年轻代的GC。

Minor GC的触发条件为：当产生一个新对象，新对象优先在Eden区分配。如果Eden区放不下这个对象，虚拟机会使用复制算法发生一次Minor GC，清除掉无用对象，同时将存活对象移动到Survivor的其中一个区(fromspace区或者tospace区)。虚拟机会给每个对象定义一个对象年龄(Age)计数器，对象在Survivor区中每“熬过”一次GC，年龄就会+1。待到年龄到达一定岁数(默认是15岁)，虚拟机就会将对象移动到年老代。如果新生对象在Eden区无法分配空间时，此时发生Minor GC。发生MinorGC，对象会从Eden区进入Survivor区，如果Survivor区放不下从Eden区过来的对象时，此时会使用分配担保机制将对象直接移动到年老代。

Major GC的触发条件：当年老代空间不够用的时候，虚拟机会使用“标记—清除”或者“标记—整理”算法清理出连续的内存空间，分配对象使用。

大家注意：**Major GC和Full GC**是不一样的，前者只清理老年代，后者会清理年轻代+老年代

Full GC触发条件

- 调用System.gc()方法时，可通过–XX:+ DisableExplicitGC 参数来禁止调用System.gc()

当方法区空间不足时

- Minor GC后存活的对象大小超过了老年代剩余空间
- Minor GC时中Survivor幸存区空间不足时，判断是否允许担保失败，不允许则触发Full GC。允许，并且每次晋升到老年代的对象平均大小>老年代最大可用连续内存空间，也会触发Full GC
- CMS GC异常，CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，会触发Full GC

当一个URL被访问时，内存申请过程 如下

A. JVM会试图为相关Java对象在Eden中初始化一块内存区域

B. 当Eden空间足够时，内存申请结束。否则到下一步

C. JVM试图释放在Eden中所有不活跃的对象(这属于1或更高级的垃圾回收), 释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区

D. Survivor区被用来作为Eden及OLD的中间交换区域，当OLD区空间足够时，Survivor区的对象会被移到Old区，否则会被保留在Survivor区
E. 当OLD区空间不够时，JVM会在OLD区进行完全的垃圾收集(0级)
F. 完全垃圾收集后，若Survivor及OLD区仍然无法存放从Eden复制过来的部分对象，导致JVM无法在Eden区为新对象创建内存区域，则出现”out of memory错误”
对象衰老的过程
young generation的内存，由一块Eden(伊甸园，有意思)和两块Survivor Space(1.4文档中称为semi-space)构成。新创建的对象的内存都分配自eden。两块Survivor Space总有会一块是空闲的，用作copying collection的目标空间。Minor collection的过程就是将eden和在用survivor space中的活对象copy到空闲survivor space中。所谓survivor，也就是大部分对象在伊甸园出生后，根本活不过一次GC。对象在young generation里经历了一定次数的minor collection后，年纪大了，就会被移到old generation中，称为tenuring。(是否仅当survivor space不足的时候才会将老对象tenuring?目前资料中没有找到描述)
剩余内存空间不足会触发GC，如eden空间不够了就要进行minor collection，old generation空间不够要进行major collection，permanent generation空间不足会引发full GC

JVM 的永久代中会发生垃圾回收么

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收（ Full GC ）。
注： Java 8 中已经移除了永久代，新加了一个叫做元数据区的 native 内存区。

垃圾收集

对象是否存活

引用计数法

引用计数法一般是：在对象中添加一个引用计数器，每当有一个地方引用他时，计数器值就加1，当引用失效时，计数器值就减1，所以任何时刻计数器值为0的对象就是不再被引用的。
缺点：需要维护一个引用计算器

可达性算法分析

目前主流的语言都是通过可达性分析（Reachability Analysis）算法来判定对象是否存活的。这个算法的基本思路是通过一系列被成为“GC Roots”的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径成为引用链（Reference Chain），如果某个对象到GC Roots间没有任何引用链相连，则证明该对象是不可能再被使用的。

垃圾收集算法

标记清除

标记-清除（Mark-Sweep）算法是最基础的GC算法，如同他的名字一样，算法分为标记和清除两个阶段，首先标记需要回收的对象，再统一回收掉标记的对象。

缺点：

（1）执行效率不稳定。若对象数量及回收对象的数量过于大，遍历标记清除的效率会很低。

（2）内存空间的碎片化问题。标记清除后会存在大量不连续的内存空间，空间碎片太多会导致以后的程序运行时如果要分配较大对象时无法找到足够连续多的内存而出发另一次GC。

标记复制算法

为了解决标记-清除算法的执行效率低的问题，Fenichel提出了一种被称为半区复制的垃圾收集算法，它将可用内存按容量划分为大小相同的两个部分，每次只使用其中的一个部分。当这一部分用完了时，就将还活着的对象复制到另一部分内存中，然后再把已使用过的内存部分一次性清理掉。

年轻代中使用的是Minor GC，这种GC算法采用的是复制算法(Copying)

a) 效率高，

缺点：需要内存容量大，比较耗内存

b) 使用在占空间比较小、刷新次数多的新生区

标记-整理算法

标记-复制算法在对象存活率较高的内存区域进行时，效率会降低。更关键的是，如果不想浪费50%的内存空间，所以就需要有额外的空间进行分配担保，以应对100%对象都存活的极端情况，所以在老年代，一般不推荐用标记-复制算法。

针对老年代的对象的存亡特征，Edward Lueders提出了另一种有针对性的标记-整理算法（Mark-Compact），其中的标记过程仍然与标记-清除算法相同，但后续的步骤不是对可回收的对象进行回收操作，而是让所有存活的对象都向内存空间的一端移动，然后直接清理掉边界以外的内存。

缺点：

（1）移动存活对象，是一种较为负重的操作，而且这种对象移动操作必须全程暂停用户应用程序才能进行。

垃圾收集器

Serial收集器

Serial收集器是一个单线程工作的收集器，它在垃圾收集时，必须暂停其他所有工作线程（Stop The World），直到它收集结束。

简单而高效（和其他收集器的单线程相比），对于内存受限的场景，它是所有收集器里面额外内存消耗最小的。

ParNew收集器

ParNew收集器实质上是Serial收集器的多线程并行版本。

ParNew 追求"低停顿时间", 与 Serial 唯一的区别就是使用了多线程进行垃圾收集, 在多 CPU 环境下性能比 Serial 会有一定的提升; 但线程切换需要额外开销, 因此在单 CPU 环境下表现不如 Serial.

CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。CMS收集器是基于标记-清除算法实现的，他的运作过程整体可以分为四个部分：

（1）初始标记

标记一下GC Roots能够关联到的对象，需要Stop The World。

（2）并发标记

从GC Roots的直接关联对象开始遍历整个对象图的过程，整个过程耗时比较长但是不会影响到用户的正常操作线程，可以和垃圾回收线程并发运行。

（3）重新标记

修正并发标记期间，因用户程序还在运行导致标记产生变动的那一部分对象的标记记录，需要Stop The World。

（4）并发清除

清除删除掉标记阶段判断的已经死亡的对象，可以并发进行。

由于在并发标记和并发清除两个耗时比较大的阶段可以和用户线程并发进行，所以CMS收集器可以有效地减少等待与停顿的时间。

缺点：

（1）由于并发进行，会减少吞吐量

（2）无法处理浮动垃圾，容易频繁导致Full GC

（3）收集结束会产生大量的空间碎片

Garbage First收集器（G1）

G1收集器是目前来说比较好的一个收集器，它开创了收集器面向局部收集的设计思路以及基于Region的内存布局格式。

它没有新生代和老年代的概念, 而是将堆划分为一块块独立的 Region. 当要进行垃圾收集时, 首先计算每个 Region 中垃圾的数量, 每次都从垃圾回收价值最大的 Region 开始回收, 因此可以获得最大的回收效率.

从整体上看, G1 是基于"标记-整理"算法实现的收集器, 从局部 (两个 Region 之间) 上看是基于"复制"算法实现的, 这意味着运行期间不会产生内存碎片.

每个 Region 中都有一个 Remembered Set, 用于记录本区域中所有对象的引用所在的区域, 进行可达性分析时, 只要在 GC Roots 中再加上 Remembered Set 即可防止对整个堆内存进行遍历.

如果不计算维护 Remembered Set 的操作, G1 收集器的工作过程可以分为一下几个步骤:

（1）初始标记: Stop The World, 仅使用一条初始标记线程对所有与 GC Roots 直接关联的对象进行标记.
（2）并发标记: 使用一条标记线程, 与用户线程并发执行. 此过程进行可达性分析, 标记出所有废弃对象, 速度很慢.
（3）最终标记: Stop The world, 使用多条线程并发执行.
（4）筛选回收: 回收废弃对象, 此时也要 Stop The world, 并使用多条筛选回收线程并发执行.

Spring

谈谈对spring框架的了解，spring有什么作用（IOC，AOP），spring的核心是什么

Spring是一个开源框架，它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。简单来说，Spring是一个分层的JavaSE/EE full-stack(一站式) 轻量级开源框架。Spring的核心是控制反转（IoC）和面向切面（AOP）
1)方便解耦，简化开发：通过Spring提供的IOC容器，我们可以将对象之间的依赖关系交由Spring进行控制，避免编码所造成的过度程序耦合。有了Spring，用户不必再为单例模式类、属性文件解析等这些很多底层的需求编写代码，可以更专注于上层的应用。
2）AOP编程的支持：通过Spring提供的AOP功能，方便进行面向切面的编程，许多不容易用传统OOP实现的功能可以通过AOP轻松应付。
3）声明式事务的支持：在Spring中，我们可以从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提供开发效率和质量

- 轻量：Spring 是轻量的，基本的版本大约2MB。
- 控制反转：Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
- 面向切面的编程(AOP)：Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
- 容器：Spring 包含并管理应用中对象的生命周期和配置。
- MVC框架：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。
- 事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
- 异常处理：Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO抛出的）转化为一致的unchecked 异常。

IOC

在Sping IoC的体系结构中BeanFactory作为最顶层的一个接口类，它定义了IoC容器的基本功能规范。并且为了区分在 Spring 内部在操作过程中对象的传递和转化过程中，对对象的数据访问做限制，

使用了多层接口ListableBeanFactory 接口表示这些 Bean 是可列表的. HierarchicalBeanFactory 表示的是这些 Bean 是有继承关系的，也就是每个Bean 有可能有父 Bean。
AutowireCapableBeanFactory 接口定义 Bean 的自动装配规则。
默认实现类是 DefaultListableBeanFactory，他实现了所有的接口.

用到的设计模式

1. **工厂模式**

Spring使用工厂模式可以通过BeanFactory或者ApplicationContext创建bean对象
BeanFactory：延迟注入（spring默认为懒加载，即使用到某个bean的时候才会注入，可以通过@Lazy(false)设置为非懒加载）。相比于ApplicationContext来说会占用更少的内存。
ApplicationContext：容器启动的时候，不管有没有用到，一次性创建所有的bean，ApplicationContext继承了BeanFactory，除了BeanFactory的功能外还有额外的更多功能，所以一般使用的更多。

2. **单例模式**

Spring 中 bean 的默认作用域就是 singleton(单例)的。

3. **策略模式**

在spring中通过ApplicationContext 来获取Resource的实例，包括urlResource，classPathResource,fileSystemResource等不同的资源类型，spring针对不同的资源类型采取不同的访问策略。ApplicationContext 将会负责选择 Resource 的实现类，也就是确定具体的资源访问策略，从而将应用程序和具体的资源访问策略分离开来。

4. **装饰者模式**

装饰者模式可以动态地给对象添加一些额外的属性或行为。相比于使用继承，装饰者模式更加灵活。简单点儿说就是当我们需要修改原有的功能，但我们又不愿直接去修改原有的代码时，设计一个Decorator套在原有代码外面。其实在 JDK 中就有很多地方用到了装饰者模式，比如 InputStream家族，InputStream 类下有 FileInputStream (读取文件)、BufferedInputStream (增加缓存,使读取文件速度大大提升)等子类都在不修改InputStream 代码的情况下扩展了它的功能。

Spring，Spring MVC，Spring Boot 的区别与联系

Spring 中的核心功能是 IOC 和 DI，适当的使用 IOC 和 DI 可以开发出一个高内聚低耦合的应用程序，而这可以使单元测试变的很方便。另外，Spring 还基于 DI 提供了很多功能模块，比方说，Spring JDBC，Spring MVC，Spring AOP, Spring ORM, Spring JMS, Spring Test，可能不是每一个模块你都需要。但是这些模块的使用都可以减少大量重复的代码，并提高单元测试效率。
Spring MVC框架提供了开发 web 应用程序的解耦方法。通过 Dispatcher Servlet、ModelAndView 和视图解析器等简单概念，可以轻松地开发 web 应用程序。基于servlet功能实现的，通过实现Servlet接口的DispatcherServlet来封装其核心功能实现，通过将请求分派给处理程序，同时带有可配置的处理程序映射、视图解析、本地语言、主题解析以及上传下载文件支持。
Spring Boot 使用Spring Framework来开发应用程序，需要进行大量的配置工作以及依赖包的管理，工作繁重而且极易出现配置错误，尤为明显的是依赖包之间的版本冲突问题。SpringBoot的**自动配置**

和**starter**机制可以快速搭建项目
1.springboot是约定大于配置，可以简化spring的配置流程；springmvc是基于servlet的mvc框架，个人感觉少了model中的映射。
2.以前web应用要使用到tomat服务器启动，而springboot内置服务器容器，通过@SpringBootApplication中注解类中main函数启动即可.

Spring Boot 启动流程

1. 准备环境，根据不同的环境创建不同的Environment
2. 准备、加载上下文，为不同的环境选择不同的Spring Context，然后加载资源，配置Bean
3. 初始化，这个阶段刷新Spring Context，启动应用
4. 最后结束流程

SpringMVC工作流程

1. 用户向服务器发送请求，请求被Spring 前端控制Servelt DispatcherServlet(中央处理器)捕获；
2. DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI）。
3. 然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象的形式返回给DispatcherServlet(中央处理器)；
4. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。（附注：如果成功获得HandlerAdapter后，此时将开始执行拦截器的preHandler(...)方法）
5. 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller）。 在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息数据转换：对请求消息进行数据转换。如String转换成Integer、Double等数据根式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中.
6. Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；
7. 根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的ViewResolver)返回给DispatcherServlet ；
8. ViewResolver 结合Model和View，来渲染视图
9. 将渲染结果返回给客户端。

Spring Boot 自动装配原理

是什么?

Spring Framework 早就实现了这个功能。Spring Boot 只是在其基础上，通过 SPI 的方式，做了进一步优化

> SpringBoot 定义了一套接口规范，这套规范规定：SpringBoot 在启动时会扫描外部引用 jar 包中的META-INF/spring.factories文件，将文件中配置的类型信息加载到 Spring 容器（此处涉及到 JVM 类加载机制与 Spring 的容器知识），并执行类中定义的各种操作。对于外部 jar 来说，只需要按照 SpringBoot 定义的标准，就能将自己的功能装置进 SpringBoot。

没有 Spring Boot 的情况下，如果我们需要引入第三方依赖，需要手动配置，非常麻烦。但是，Spring Boot 中，我们直接引入一个 starter 即可。比如你想要在项目中使用 redis 的话，直接在项目中引入对应的 starter 即可。
引入 starter 之后，我们通过少量注解和一些简单的配置就能使用第三方组件提供的功能了。
自动装配可以简单理解为：通过注解或者一些简单的配置就能在 Spring Boot 的帮助下实现某块功能。

怎么实现的?
SpringBoot 的核心注解 `@SpringBootApplication`

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
//允许在上下文中注册额外的 bean 或导入其他配置类
@SpringBootConfiguration
//启用 SpringBoot 的自动配置机制
@EnableAutoConfiguration
//扫描被@Component (@Service,@Controller)注解的 bean，注解默认会扫描启动类所在的包下所有的类 ，可以自定义不扫描某些 bean
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication{}
```

`@EnableAutoConfiguration` 只是一个简单地注解，自动装配核心功能的实现实际是通过

AutoConfigurationImportSelector类

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
//将main包下的所欲组件注册到容器中
@AutoConfigurationPackage
//@Import用来导入配置类或者一些需要前置加载的类.
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
    Class<?>[] exclude() default {};
    String[] excludeName() default {};
}
```

Spring Boot 通过@EnableAutoConfiguration开启自动装配，通过 SpringFactoriesLoader 最终加载META-INF/spring.factories中的自动配置类实现自动装配，自动配置类其实就是通过@Conditional按需加载的配置类，想要其生效必须引入spring-boot-starter-xxx包实现起步依赖

创建自己的starter

@Autowired 与@Resource的区别

@Autowired默认按类型装配（这个注解是属于spring的），如果类型有多个实现类可以用name判断，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

@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，默认取字段名进行安装名称查找，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。

@Scope注解

```
@Scope("singleton ")
```

- singleton
- prototype 每次都重新生成一个新的对象给请求方

- request 每个HTTP请求创建一个全新的RequestPrecessor对象
- session Spring容器会为每个独立的session创建
- global session 只有应用在基于porlet的web应用程序中才有意义，它映射到porlet的global范围的session，如果普通的servlet的web 应用中使用了这个scope，容器会把它作为普通的session的scope对待。

@Required 注解

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。

@Qualifier 注解

当有多个相同类型的bean却只有一个需要自动装配时，将@Qualifier 注解和@Autowire 注解结合使用以消除这种混淆，指定需要装配的确切的bean。

SpringMVC必须要返回modelView么，SpringMVC接收一个json数据时怎么处理的

不是，使用@ResponseBody把后台pojo转换json对象，返回到页面；@RequestBody接受前台json，把json数据自动封装pojo。

依赖注入

依赖注入，是IOC的一个方面，是个通常的概念，它有多种解释。这概念是说你不用创建对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，之后一个容器（IOC容器）负责把他们组装起来。

AOP

面向切面的编程，或AOP， 是一种编程技术，允许程序模块化横向切割关注点，或横切典型的责任划分，如日志和事务管理。

设计模式死磕Spring AOP系列5：设计模式在AOP中的使用 - 爱码网 (likecs.com)

1. **策略模式**
2. **模板模式**
3. **责任链模式**
4. **适配器模式**
5. **桥接模式**

通知类型

//前置通知：在方法执行前通知

@Before(value = “”)

//环绕通知：可以将要执行的方法（point.proceed()）进行包裹执行，可以在前后添加需要执行的操作
@Around(value = “”)
//后置通知：在方法正常执行完成进行通知，可以访问到方法的返回值的。
@AfterReturning(value = “”)
//异常通知：在方法出现异常时进行通知,可以访问到异常对象，且可以指定在出现特定异常时在执行通知。
@AfterThrowing(value = “”)
//方法执行后通知： 在目标方法执行后无论是否发生异常，执行通知,不能访问目标方法的执行的结果。
@After(value = “”)

> 😄 注意区分@After和@AfterReturning区别，After不论方法是否执正常结束都会通知，而AfterReturning出现异常未正常结束则不会通知。

Aspect 切面
AOP核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组API提供横切功能。比如，一个日志模块可以被称作日志的AOP切面。根据需求的不同，一个应用程序可以有若干切面。在Spring AOP中，切面通过带有@Aspect注解的类实现。

Spring AOP 和 AspectJ AOP 有什么区别
Spring AOP 基于动态代理实现，属于运行时增强。
AspectJ 则属于编译时增强，主要有3种方式：
1. 编译时织入：指的是增强的代码和源代码我们都有，直接使用 AspectJ 编译器编译就行了，编译之后生成一个新的类，他也会作为一个正常的 Java 类装载到JVM。
2. 编译后织入：指的是代码已经被编译成 class 文件或者已经打成 jar 包，这时候要增强的话，就是编译后织入，比如你依赖了第三方的类库，又想对他增强的话，就可以通过这种方式。
3. 加载时织入：指的是在 JVM 加载类的时候进行织入。
总结下来的话，就是 Spring AOP 只能在运行时织入，不需要单独编译，性能相比 AspectJ 编译织入的方式慢，而 AspectJ 只支持编译前后和类加载时织入，性能更好，功能更加强大。

Spring Bean注入的几种方式
Spring常用的依赖注入方式是：Setter方法注入、构造器注入、Filed注入(用于注解方式)；
如果有循环依赖，构造器注入会报错，创建bean时需要先创建实例，这个时候循环依赖了，创建不了实例，就出错了，改为setter注入，让可以创建实例，再通过属性注入的方式填充属性。

同类型多个Bean的注入

通过名称注入

名称注入会指定一个明确的 Bean 名称，容器不允许注册相同名称的 Bean，所以不会有任何问题。

通过类型注入

通过类型注入的时候，有时会因为多个 Bean 的类型相同而产生冲突。例如：

同一类型注册多个不同名称的 Bean

抽象类型注册多个不同实现类的 Bean

```
// BeanService的三个实现类注册Bean
@Configuration
public class AppConfig {

    @Bean
    public BeanService oneServiceImpl() {
        return new OneServiceImpl();
    }

    @Bean
    public BeanService twoServiceImpl() {
        return new TwoServiceImpl();
    }

    @Bean
    public BeanService threeServiceImpl() {
        return new ThreeServiceImpl();
    }
}

// 通过接口的类型注入会抛出异常
public class ServiceTest {

    @Autowired
    private BeanService beanService;
}
```

注册 Bean 的时候，使用 `@Primary` 指定一个 Bean 为主要的，存在冲突时默认选择主要的 Bean。

```
@Configuration
public class AppConfig {

```

```
    @Bean
    @Primary
    public BeanService oneServiceImpl() {
        return new OneServiceImpl();
    }

    // ......
}
```

注入 Bean 的时候，使用 `@Qualifier` 指定具体 Bean 的名称，通过名称注入解决冲突

```
public class ServiceTest {

    @Autowired
    @Qualifier("oneServiceImpl")
    private BeanService beanService;

    // ......
}
```

也可以直接通过**字段名称**来指定具体 Bean 的名称，来解决冲突。

```
public class ServiceTest {

    @Autowired
    private BeanService oneServiceImpl;

    // ......
}
```

注入多个 Bean

注入集合

```
public class ServiceTest {

    @Autowired
    private BeanService[] beanServiceArr;

    // ......
}
public class ServiceTest {

```

```
    @Autowired
    private List<BeanService> beanServiceList;

    // ......
}
public class ServiceTest {

    @Autowired
    private Set<BeanService> beanServiceSet;

    // ......
}
```

注入map

```
public class ServiceTest {

    @Autowired
    private Map<String, BeanService> beanServiceMap;

    // ......
}
```

注册 Bean 的时候可以使用 `@Order` 注解来指定 Bean 的权重（或顺序）。
在使用有序集合（数组或 `List`）注入的时候，会根据权重来排序。

```
@Configuration
public class AppConfig {

    @Bean
    @Order(1)
    public BeanService oneServiceImpl() {
        return new OneServiceImpl();
    }

    @Bean
    @Order(3)
    public BeanService twoServiceImpl() {
        return new TwoServiceImpl();
    }

    @Bean
```

```
    @Order(2)
    public BeanService threeServiceImpl() {
        return new ThreeServiceImpl();
    }
}
```

可以在Spring中注入一个null 和一个空字符串吗

可以

FactoryBean 和 BeanFactory有什么区别

BeanFactory 是 Bean 的工厂， ApplicationContext 的父类，IOC 容器的核心，负责生产和管理Bean对象。

FactoryBean 是 Bean，可以通过实现 FactoryBean 接口定制实例化 Bean 的逻辑，通过代理一个Bean对象，对方法前后做一些操作。

Spring中Bean的生命周期

- Spring容器 从XML 文件中读取bean的定义，并实例化bean。
- Spring根据bean的定义填充所有的属性。
- 如果bean实现了BeanNameAware 接口，Spring 传递bean 的ID 到 setBeanName方法。
- 如果Bean 实现了 BeanFactoryAware 接口， Spring传递beanfactory 给setBeanFactory 方法。

(这里可以通过实现ApplicationContextAware，获取到ApplicationContext,通过代码用ApplicationContext获取bean)

- 如果有任何与bean相关联的BeanPostProcessors，Spring会在postProcesserBeforeInitialization()方法内调用它们。
- 如果bean实现IntializingBean了，调用它的afterPropertySet方法，如果bean声明了初始化方法，调用此初始化方法。
- 如果有BeanPostProcessors 和bean 关联，这些bean的postProcessAfterInitialization() 方法将被调用。
- 如果bean实现了 DisposableBean，它将调用destroy()方法。

实例化->属性设置->初始化->销毁

内部bean

当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean，为了定义inner bean，在Spring 的 基于XML的 配置元数据中，可以在 <property/>或 <constructor-arg/> 元素内使用 <bean/> 元素，内部bean通常是匿名的，它们的Scope一般是prototype。

spring框架中的单例bean是线程安全的吗

不，Spring框架中的单例bean不是线程安全的。

Spring 如何解决循环依赖的

使用三级缓存

- singletonObjects 一级缓存，用于保存实例化、注入、初始化完成的bean实例
- earlySingletonObjects 二级缓存，用于保存实例化完成的bean实例（半成品）
- singletonFactories 三级缓存，用于保存bean创建工厂，以便于后面扩展有机会创建代理对象。

一级缓存能解决吗?

- 其实只有一级缓存并不是不能解决循环依赖，就像我们自己做的例子一样。
- 但是在 Spring 中如果像我们例子里那么处理，就会变得非常麻烦，而且也可能会出现 NPE 问题。
- 所以如图按照 Spring 中代码处理的流程，我们去分析一级缓存这样存放成品 Bean 的流程中，是不能解决循环依赖的问题的。因为 A 的成品创建依赖于 B，B的成品创建又依赖于 A，当需要补全B的属性时 A 还是没有创建完，所以会出现死循环。

二级缓存能解决吗?

- 有了二级缓存其实这个事处理起来就容易了，一个缓存用于存放成品对象，另外一个缓存用于存放半成品对象。
- A 在创建半成品对象后存放到缓存中，接下来补充 A 对象中依赖 B 的属性。
- B 继续创建，创建的半成品同样放到缓存中，在补充对象的 A 属性时，可以从半成品缓存中获取，现在 B 就是一个完整对象了，而接下来像是递归操作一样 A 也是一个完整对象了。

三级缓存解决什么

- 有了二级缓存都能解决 Spring 依赖了，怎么要有三级缓存呢。其实我们在前面分析源码时也提到过，三级缓存主要是解决 Spring AOP 的特性。AOP 本身就是对方法的增强，是 ObjectFactory<?> 类型的 lambda 表达式，而 Spring 的原则又不希望将此类类型的 Bean 前置创建，所以要存放到三级缓存中处理。
- 其实整体处理过程类似，唯独是 B 在填充属性 A 时，先查询成品缓存、再查半成品缓存，最后在看看有没有单例工程类在三级缓存中。最终获取到以后调用 getObject 方法返回代理引用或者原始引用。

通过代码获取bean的方法

- 继承ApplicationContextAware获取ApplicationContext
- 用ContextLoader.getCurrentWebApplicationContext()获取WebApplicationContext

Spring事务

Spring为事务提供了完整的支持，使用Spring来管理事务有以下好处：

- Spring为不同的编程模型如JTA、JDBC、Hibernate、JPA等，提供了统一的事务API
- 支持声明式事务，更容易地管理事务
- 相比于 JTA，Spring为编程式事务提供更加简单的API

Spring事务抽象的核心类图

`PlatformTransactionManager` 事务管理器
`getTransaction` 事务获取操作，根据事务属性定义，获取当前事务或者创建新事务
`commit` 事务提交操作，注意这里所说的提交并非直接提交事务，而是根据当前事务状态执行提交或者回滚操作
`rollback` 事务回滚操作，同样，也并非一定直接回滚事务，也有可能只是标记事务为只读，等待其他调用方执行回滚。
`TransactionDefinition` 事务属性定义
`TransactionStatus` 当前事务状态
部分Spring包含的对 `PlatformTransactionManager` 的实现类如下图所示

`AbstractPlatformTransactionManager` 抽象类实现了Spring事务的标准流程，其子类 `DataSourceTransactionManager` 是我们使用较多的JDBC单数据源事务管理器，而 `JtaTransactionManager` 是JTA（Java Transaction API）规范的实现类，另外两个则分别是JavaEE容器 `WebLogic` 和 `WebSphere` 的JTA事务管理器的具体实现。

Spring采用AOP来实现声明式事务

代理对象生成的核心类是 `AbstractAutoProxyCreator` ，实现了 `BeanPostProcessor` 接口，会在Bean初始化完成之后，通过 `postProcessAfterInitialization` 方法生成代理对象

Spring事务拦截

当Bean方法通过代理对象调用时，会触发对应的AOP增强拦截器，前面提到声明式事务是一种环绕增强，对应接口为 `MethodInterceptor` ，事务增强对该接口的实现为 `TransactionInterceptor` ，类图如下：

事务拦截器 `TransactionInterceptor` 在invoke方法中，通过调用父类 `TransactionAspectSupport` 的 `invokeWithinTransaction` 方法进行事务处理，该方法支持声明式事务和编程式事务。

声明式事务管理

使用@Transactional注解来管理事务比较简单
@Transactional 默认回滚RuntimeException

```
1 //@Transactional 默认回滚RuntimeException
```

```java
//@Transactional(rollbackFor = {Exception.class}) //出现FileNotFoundException会回滚
@Transactional //出现FileNotFoundException，不会回滚
public int insert(UserInfo record) throws IOException {
    int result = userInfoDao.insert(record);
    File file = new File("e://1.txt");
    InputStream inputStream = new FileInputStream(file);
    return result;
}
```

属性

- propagation（事务的传播行为）
- isolation （隔离级别）
- timeout
- readOnly
- rollbackFor

事务传播行为

事务的传播行为指的是，当应用程序中的服务间互相调用， 如果调用方已经创建或尚未创建事务，那么被调用的服务将如何处理事务的一种行为特征，如下图所示：

REQUIRED（必需的-默认）	如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务
SUPPORTS（支持）	如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行
MANDATORY（强制）	如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常
REQUIRES_NEW	创建一个新的事务，如果当前存在事务，则把当前事务挂起。
NOT_SUPPORTED	以非事务方式运行，如果当前存在事务，则把当前事务挂起。
NEVER	以非事务方式运行，如果当前存在事务，则抛出异常。
NESTED （嵌套）	如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 REQUIRED

事务不生效

1. Spring要求被代理方法必须是 `public` 的
2. 如果某个方法用 `final` 修饰了，那么在它的代理类中，就无法重写该方法，而添加事务功能。如

果某个方法是 `static` 的，同样无法通过动态代理，变成事务方法。

3. 未被spring管理，对象要被spring管理，需要创建bean实例。
4. 多线程调用

```
@Slf4j
@Service
public class UserService {
    @Autowired
    private UserMapper userMapper;
    @Autowired
    private RoleService roleService;
    @Transactional
    public void add(UserModel userModel) throws Exception {
        userMapper.insertUser(userModel);
        new Thread(() -> {
            roleService.doOtherThing();
        }).start();
    }
}
@Service
public class RoleService {
    @Transactional
    public void doOtherThing() {
        System.out.println("保存role表数据");
    }
}
```

看到事务方法add中，调用了事务方法doOtherThing，但是事务方法doOtherThing是在另外一个线程中调用的。

这样会导致两个方法不在同一个线程中，获取到的数据库连接不一样，从而是两个不同的事务。如果想doOtherThing方法中抛了异常，add方法也回滚是不可能的。

spring的事务是通过数据库连接来实现的。当前线程中保存了一个map，key是数据源，value是数据库连接

```
private static final ThreadLocal<Map<Object, Object>> resources =  new
NamedThreadLocal<>("Transactional resources");
```

我们说的同一个事务，其实是指同一个数据库连接，只有拥有同一个数据库连接才能同时提交和回滚。如果在不同的线程，拿到的数据库连接肯定是不一样的，所以是不同的事务。

5. 表不支持事务
6. 未开启事务

如果使用的是springboot项目，springboot通过 `DataSourceTransactionManagerAutoConfiguration` 类，已经默认开启了事务。
如果使用的还是传统的spring项目，则需要在 `applicationContext.xml` 文件中，手动配置事务相关参数。如果忘了配置，事务肯定是不会生效的

```
<!-- 配置事务管理器 -->
<bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager" id="transactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
<tx:advice id="advice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
<!-- 用切点把事务切进去 -->
<aop:config>
    <aop:pointcut expression="execution(* com.susan.*.*(..))" id="pointcut"/>
    <aop:advisor advice-ref="advice" pointcut-ref="pointcut"/>
</aop:config>
```

7. 接口中A、B两个方法，A无@Transactional标签，B有，上层通过A间接调用B，此时事务不生效。

```
//调用test方法，事务不生效，不会回滚
@Override
public int test(UserInfo record) throws IOException {
    return insert(record);
}
@Override
@Transactional(rollbackFor = {Exception.class})
public int insert(UserInfo record) throws IOException {
    int result = userInfoDao.insert(record);
    File file = new File("e://1.txt");
    InputStream inputStream = new FileInputStream(file);
    return result;
}
```

原因：
声明式事务基于Spring AOP实现,将具体业务逻辑与事务处理解耦，在 Spring 的 AOP 代理下，只有目标方法由外部调用，目标方法才由 Spring 生成的代理对象来管理，这会造成自调用问题

spring的@Transactional事务生效的一个前提是进行方法调用前经过拦截器TransactionInterceptor，也就是说只有通过TransactionInterceptor拦截器的方法才会被加入到spring事务管理中

如果是在同一个类中的方法调用，则不会被方法拦截器拦截到，因此事务不会起作用，必须将方法放入另一个类，并且该类通过spring注入。

```
//一种解决方法，自己注入自己，用注入的对象去调有事务的方法
@Lazy//解决循环依赖
@Autowired
UserInfoService userInfoService;
@Override
public int test(UserInfo record) throws IOException {
    System.out.println(this);
    return userInfoService.insert(record);
}
@Override
@Transactional(rollbackFor = {Exception.class})
public int insert(UserInfo record) throws IOException {
    System.out.println(this);
    int result = userInfoDao.insert(record);
    File file = new File("e://1.txt");
    InputStream inputStream = new FileInputStream(file);
    return result;
}
```

编程式事务管理

```
@Service
public class TransactionDemo {
    @Autowired
    private TransactionTemplate transactionTemplate;
    public void programmaticUpdate() {
        // 这里也可以使用Lambda表达式
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                updateOperation1();
                updateOperation2();
            }
        });
```

```
13      }
14  }
```

大事务问题

`@Transactional` 注解，如果被加到方法上，有个缺点就是整个方法都包含在事务当中了。
如果query方法非常多，调用层级很深，而且有部分查询方法比较耗时的话，会造成整个事务非常耗时，而从造成大事务问题。

Spring设置为单例，那么线程安全问题怎么解决

1）在Bean对象中尽量避免定义可变的成员变量；
2）在Bean对象中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在ThreadLocal中。

Spring对设计模式的应用

简单工厂

BeanFactory。Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

工厂方法

FactoryBean接口。
实现了FactoryBean接口的bean是一类叫做factory的bean。其特点是，spring会在使用getBean()调用获得该bean时，会自动调用该bean的getObject()方法，所以返回的不是factory这个bean，而是这个bean.getOjbect()方法的返回值。

单例模式

Spring依赖注入Bean实例默认是单例的。
Spring的依赖注入（包括lazy-init方式）都是发生在AbstractBeanFactory的getBean里。getBean的doGetBean方法调用getSingleton进行bean的创建。

代理模式

AOP底层，就是动态代理模式的实现。
动态代理：
在内存中构建的，不需要手动编写代理类
静态代理：
需要手工编写代理类，代理类引用被代理对象。
实现原理：

切面在应用运行的时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象创建动态的创建一个代理对象。SpringAOP就是以这种方式织入切面的。
织入：把切面应用到目标对象并创建新的代理对象的过程。

策略模式
Resource
Resource 接口主要提供了如下几个方法:

- getInputStream()：定位并打开资源，返回资源对应的输入流。每次调用都返回新的输入流。调用者必须负责关闭输入流。
- exists()：返回 Resource 所指向的资源是否存在。
- isOpen()： 返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束应该显式关闭，以防止资源泄漏。
- getDescription()：返回资源的描述信息，通常用于资源处理出错时输出该信息，通常是全限定文件名或实际 URL。
- getFile：返回资源对应的 File 对象。
- getURL：返回资源对应的 URL 对象。

最后两个方法通常无须使用，仅在通过简单方式访问无法实现时，Resource 提供传统的资源访问的功能。
Resource 接口本身没有提供访问任何底层资源的实现逻辑，针对不同的底层资源，Spring 将会提供不同的 Resource 实现类，不同的实现类负责不同的资源访问逻辑。
Spring 为 Resource 接口提供了如下实现类:

- UrlResource：访问网络资源的实现类。
- ClassPathResource：访问类加载路径里资源的实现类。
- FileSystemResource：访问文件系统里资源的实现类。
- ServletContextResource：访问相对于 ServletContext 路径里的资源的实现类.
- InputStreamResource：访问输入流资源的实现类。
- ByteArrayResource：访问字节数组资源的实现类。

这些 Resource 实现类，针对不同的的底层资源，提供了相应的资源访问逻辑，并提供便捷的包装，以利于客户端程序的资源访问。

模版方法模式
经典模板方法定义：
父类定义了骨架（调用哪些方法及顺序），某些特定方法由子类实现。
最大的好处：代码复用，减少重复代码。除了子类要实现的特定方法，其他方法及方法调用顺序都在父类中预先写好了。
所以父类模板方法中有两类方法：

- 共同的方法：所有子类都会用到的代码

- 不同的方法： 子类要覆盖的方法，分为两种：
  a. 抽象方法：父类中的是抽象方法，子类必须覆盖
  b. 钩子方法：父类中是一个空方法，子类继承了默认也是空的

注：为什么叫钩子，子类可以通过这个钩子（方法），控制父类，因为这个钩子实际是父类的方法（空方法）！

Spring模板方法模式实质：

是模板方法模式和回调模式的结合，是Template Method不需要继承的另一种实现方式。Spring几乎所有的外接扩展都采用这种模式。

具体实现：

JDBC的抽象和对Hibernate的集成，都采用了一种理念或者处理方式，那就是模板方法模式与相应的Callback接口相结合。

Spring Cloud

spring cloud 的核心组件有哪些

- Eureka：服务注册于发现。
- Feign：基于动态代理机制，根据注解和选择的机器，拼接请求 url 地址，发起请求。
- Ribbon：实现负载均衡，从一个服务的多台机器中选择一台。
- Hystrix：提供线程池，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题。
- Zuul：网关管理，由 Zuul 网关转发请求给对应的服务。

所知道的微服务技术栈

- 维度(springcloud)
- 服务开发：springboot spring springmvc
- 服务配置与管理:Netfix公司的Archaiusm ,阿里的Diamond
- 服务注册与发现:Eureka,Zookeeper
- 服务调用:Rest RPC gRpc
- 服务熔断器:Hystrix
- 服务负载均衡:Ribbon Nginx
- 消息队列:Kafka Rabbitmq activemq
- 服务配置中心管理:SpringCloudConfig
- 服务路由（API网关）Zuul
- 事件消息总线:SpringCloud Bus
- Spring Cloud Security
- 服务接口调用:Fegin

微服务的优缺点分别是什么

优点

- 每一个服务足够内聚,代码容易理解
- 开发效率提高,一个服务只做一件事
- 微服务能够被小团队单独开发
- 微服务是松耦合的,是有功能意义的服务
- 可以用不同的语言开发,面向接口编程
- 易于与第三方集成
- 微服务只是业务逻辑的代码,不会和HTML,CSS或者其他界面组合
- 可以灵活搭配,连接公共库/连接独立库

缺点

- 分布式系统的负责性
- 多服务运维难度,随着服务的增加,运维的压力也在增大
- 系统部署依赖
- 服务间通信成本
- 数据一致性
- 系统集成测试
- 性能监控

Spring Cloud Consul

提供的模式包括服务发现、分布式配置和控制总线

Spring Cloud Consul 功能：

- 服务发现：实例可以注册到 Consul 代理，客户端可以使用 Spring 管理的 bean 发现实例
- 支持 Ribbon，通过 Spring Cloud Netflix 的客户端负载均衡器
- 支持 Spring Cloud LoadBalancer – Spring Cloud 项目提供的客户端负载均衡器
- 支持 Zuul，通过 Spring Cloud Netflix 的动态路由器和过滤器
- 分布式配置：使用 Consul Key/Value 存储 (配置中心作用：随着业务的发展、微服务架构的升级，服务的数量、程序的配置日益增多（各种微服务、各种服务器地址、各种参数），传统的配置文件方式和数据库的方式已无法满足开发人员对配置管理的要求：

安全性：配置跟随源代码保存在代码库中，容易造成配置泄漏时效性：修改配置，需要重启服务才能生效局限性：无法支持动态调整：例如日志开关、功能开关因此，我们需要配置中心来统一管理配置。)

- 控制总线：使用 Consul Events 的分布式控制事件

需要启动Consul，Spring Boot项目知道加上 @EnableDiscoveryClient 注解就会尝试连接Consul，可通过 spring.cloud.consul.host 和 spring.cloud.consul.port设置Consul的地址。

consul和eureka区别

Eureka优点是注册速度很快，不管同步到其他节点时是否有问题，只要服务注册到主节点既代表注册成功。牺牲了一致性，但是保证了高可用性和最终一致性。即使当前节点因为一些问题没有注册成功，那么也会通过其他节点找到当前服务，返回元数据。和eureka相比，consul注册就稍慢一些了。上面提到了，consul是强一致性的，所以consul注册服务是先注册，然后同步到各节点，raft算法使consul在有一半以上的节点注册成功时才证明服务注册成功。这样保证了数据的一致性。但也因为这样，导致注册服务相对较慢，并且当主节点挂掉之后，重新选举时整个consul不可用。可以说是牺牲了一部分可用性换来的一致性。eureka不再更新了
Nacos 与consul基本相似，多了雪崩保护，支持cp/ap,

服务间调用（Fegin）

通过注册中心根据服务名获取到真实的请求地址，有服务直接发起请求，不经过网关、注册中心

MySQL

存储引擎

存储引擎Storage engine：MySQL中的数据、索引以及其他对象是如何存储的，是一套文件系统的实现。

- `Innodb` ： Innodb引擎提供了对数据库ACID事务的支持，有提交、回滚、崩溃恢复功能以保护用户数据。并且还提供了行级锁和外键的约束，将数据存储在聚集索引中，以减少基于主键的常见查询的I/O,它的设计的目标就是处理大数据容量的数据库系统。（.ibd文件，存的数据+索引）
- `MyISAM`（原本Mysql的默认引擎）：不提供事务的支持，也不支持行级锁和外键。.MYD存数据，.MYI存索引，占空间小，使用表级锁（.Myd数据、.Myi索引、.frm建表语句8.0取消了）
- `MEMORY` ： 所有的数据都在内存中，以便在需要快速查找非关键数据的环境中快速访问。

特征	InnoDB	MyISAM	Memory
外键	**yes**	**no**	**no**
全文搜索索引	**yes** **5.6之后**	**yes**	**no**
锁定粒度	**行**（未命中索引会退化未表级锁）	**表**	**表**
MVCC	**yes**	**no**	**no**
事务	**yes**	**no**	**no**
存储限制	**64TB**	**256TB**	**RAM**
hash索引	no 在内部使用哈希索引	no	yes

	来实现其自适应哈希索引功能		
B-tree	yes	yes	yes
集群索引	yes	no	no
数据缓存	yes	no	N/A
加密数据 通过加密功能在服务器中实现	yes 5.7之后支持静态数据加密	yes	yes
地理空间数据类型支持	yes	yes	no
地理空间索引支持	yes 5.7之后	yes	no
索引缓存	yes	yes	N/A

锁机制

- 共享锁：由读表操作加上的锁，加锁后其他用户只能获取该表或行的共享锁，不能获取排它锁，也就是说只能读不能写
- 排他锁：由写表操作加上的锁，加锁后其他用户不能获取该表或行的任何锁

锁的范围:

- 行锁:对某行记录加上锁
- 表锁: 对整个表加上锁

状态锁

状态锁包括意向共享锁和意向排它锁，把他们区分为状态锁的一个核心逻辑，是因为这两个锁都是都是描述是否可以对某一个表进行加表锁的状态。

意向锁的解释：当一个事务试图对**整个表**进行加锁（共享锁或排它锁）之前，首先需要获得对应类型的意向锁（意向共享锁或意向共享锁）

意向共享锁

当一个事务试图对**整个表**进行加共享锁之前，首先需要获得这个表的意向共享锁。

意向排他锁

当一个事务试图对**整个表**进行加排它锁之前，首先需要获得这个表的意向排它锁。

为什么我们需要意向锁?

意向锁光从概念上可能有点难理解，所以我们有必要从一个案例来分析其作用，这里首先我们先要有一个概念那就是innodb加锁的方式是基于索引，并且加锁粒度是行锁，然后我们来看下面的案例。

第一步:

事务A对user_info表执行一个SQL:update user_info set name =”张三” where id=6 加锁情况如下图;

第二步：

与此同时数据库又接收到事务B修改数据的请求：SQL: update user_info set name =”李四”；

1、因为事务B是对整个表进行修改操作，那么此SQL是需要对整个表进行加排它锁的（update加锁类型为排他锁）；

2、我们首先做的第一件事是先检查这个表有没有被别的事务锁住，只要有事务对表里的任何一行数据加了共享锁或排他锁我们就无法对整个表加锁（**排他锁不能与任何属性的锁兼容**）。

3、因为INNODB锁的机制是基于行锁，那么这个时候我们会对整个索引每个节点一个个检查，我们需要检查每个节点是否被别的事务加了共享锁或排它锁。

4、最后检查到索引ID为6的节点被事务A锁住了，最后导致事务B只能等待事务A锁的释放才能进行加锁操作。

思考：

在A事务的操作过程中，后面的每个需要对user_info加持表锁的事务都需要遍历整个索引树才能知道自己是否能够进行加锁，这种方式是不是太浪费时间和损耗数据库性能了？

所以就有了意向锁的概念：如果当事务A加锁成功之后就设置一个状态告诉后面的人，已经有人对表里的行加了一个排他锁了，你们不能对整个表加共享锁或排它锁了，那么后面需要对整个表加锁的人只需要获取这个状态就知道自己是不是可以对表加锁，避免了对整个索引树的每个节点扫描是否加锁，而这个状态就是我们的意向锁。

事务

特性

- A 原子性（一个事务对数据进行修改时，要么都成功，要么都失败）
- C 一致性（当事务回滚、提交、进行中时数据库保持一致的状态，如跨表更新时，会看到所有的旧值或所有的新值，不会是混和的）
- I 隔离性（事务在进程中是相互隔离的，互不干扰）
- D持久性（事务一旦提交后，数据的改变不受其他潜在危险的影响）

数据库如何保证ACID的?

A 通过undo log，undo log记录了这些回滚需要的信息，当事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。

C 数据如果有约束性，那么事务执行前后，数据的约束性还是存在的，并没有破坏掉。比如说转账事务的前后，资金的总数这个约束条件是不会被破坏的

如何保证：通过undo，回滚机制来保证

I 多个事务并行，一个事务所做的更改，不管有没有提交，在并发的另一个事务中都是不可见的，但最后的效果看起来多个并行的事务好像是串行一样

如何保证：通过给操作的对象加悲观锁或者乐观锁，MVCC(undo log)来保证，RC不满足隔离性，RR满足隔离性

D 一旦事务提交，其所做的修改就会永久保存到数据库中。此时即使系统崩溃，修改的数据也不会丢失。这个duration指的是mysql服务器可以重启的情况下，crash掉之后(比如说只写到pool buffer中，还没有fsync到磁盘文件中)，保证duration。而不是物理损坏，物理损坏由备份来负责的,redo log用于保证事务的持久性
如何保证：是通过redo来保证的。

隔离级别

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED 读未提交	√	√	√
READ COMMITTED 读提交的	×	√	√
REPEATABLE READ 可重复读	×	×	√
SERIALIZABLE 序列化	×	×	×

- read uncommitted

A事务修改数据会对数据行加共享锁，A事务未提交或回滚时B事务的修改会被挂起，B可以查询到A未提交的数据，如果A回滚了，B就读到了脏数据;

- read committed

A事务修改数据会对数据行加共享锁，A事务未提交或回滚时B事务的修改会被挂起，B查询数据时看不到A未提交的修改，A提交后B再查询会看到和之前不一样的数据，出现不可重复读;

- repeatable read

A事务修改数据会对数据行加共享锁，A事务未提交或回滚时B事务的修改会被挂起，B事务未结束前读到的数据都是一样的，不受A事务修改的影响，即使A事务提交了；可能出现幻读，一行数据出现在结果集中，但不在之前的结果集中。B事务查询了两次表，中间A插入了一条数据并提交了，B就查到了新添的数据，称为幻读。（MySQL中select不会看到新加的数据， select .... for update会看到，A事务提交的情况下，，，）

- serializable

提供严格的事务隔离。它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行。

```
#查询当前会话隔离级别
select @@transaction_isolation;
#查看全局隔离级别
select @@global.transaction_isolation;
#设置隔离级别
set session transaction isolation level read uncommitted;
```

当前读、快照读、MVCC

当前读

读取的是最新版本, 并且对读取的记录加锁, 阻塞其他事务同时改动相同记录，避免出现安全问题；

记录锁(Record Lock)

```
select...lock in share mode (共享读锁)
select...for update
update , delete , insert
```

实现方法：**next-key锁(行记录锁+Gap间隙锁)** https://www.cnblogs.com/wwcom123/p/10727194.html

间隙锁：只有在Read Repeatable、Serializable隔离级别才有，就是锁定范围空间的数据，假设id有3,4,5，锁定id>3的数据，是指的4，5及后面的数字都会被锁定，因为此时如果不锁定没有的数据，例如当加入了新的数据id=6，就会出现幻读，间隙锁避免了幻读。

1.对主键或唯一索引，如果当前读时，where条件全部精确命中(=或者in)，这种场景本身就不会出现幻读，所以只会加行记录锁（唯一索引不会有间隙索引）。

2.没有索引的列，当前读操作时，会加全表gap锁，生产环境要注意。

3.非唯一索引列，如果where条件部分命中(>、<、like等)或者全未命中，则会加附近Gap间隙锁。例如，某表数据如下，非唯一索引2,6,9,9,11,15。如下语句要操作非唯一索引列9的数据，gap锁将会锁定的列是(6,11]（左开右闭），该区间内无法插入数据。

快照读

单纯的select操作，不包括上述 select ... lock in share mode, select ... for update。

Read Committed隔离级别：每次select都生成一个快照读。

Read Repeatable隔离级别：开启事务后第一个select语句才是快照读的地方，而不是一开启事务就快照读。

实现方法：**undolog和多版本并发控制MVCC**

一行数据记录，主键ID是10，name='Jack'，age=10, 被update更新set为name= 'Tom'，age=23。

事务会先使用“排他锁”锁定改行，将该行当前的值复制到undo log中，然后再真正地修改当前行的值，最后填写事务的**DB_TRX_ID**，使用回滚指针**DB_ROLL_PTR**指向undo log中修改前的行**DB_ROW_ID**。

索引

一种为表的行提供快速查找能力的数据结构，通常是一个树结构

InnoDB 总是有一个代表主键的 **聚簇索引（clustered index）**,还可以有一个或多个定义在一列或多列的 **二级索引（secondary index）**，根据他们的结构，二级索引可分为 **部分索引（partial）、列索引**

（column）、复合索引（composite）

聚簇索引：主键索引的 InnoDB 术语，InnoDB 表存储基于主键列的值进行组织，以加快涉及主键列的查询和排序。

覆盖查询：通过查询能检索到所有的列，查询不是使用索引值作为指针来查找完整的表行，而是从索引结构中返回值，从而节省磁盘 I/O；给定正确的查询，任何列索引或复合索引都可以充当覆盖索引。

主键索引：可以唯一标识每一行数据，必须是不含任何null值的唯一索引

二级索引：一种表示表列子集的 InnoDB 索引，二级索引可用于满足只需要索引列中的值的查询。对于更复杂的查询，它可用于识别表中的相关行，然后使用聚集索引通过查找来检索这些行。节点存的是索引的key，叶子节点存的是主键key，根据主键去聚簇索引中找行数据（回表）

部分索引：通常是varchar值的前n个字符

列索引：单个列的索引

复合索引：包含多个列的索引

常用的索引：**B-tree、hash index、R-tree**

聚簇和非聚簇区别

聚簇索引：将数据存储和索引放在一起、并且是按照一定的顺序组织的，找到索引也就找到了数据，数据的物理存放顺序与索引顺序是一致的，一个表只能有一个聚簇索引，聚簇索引在一个文件，建表时指定了主键就会以主键为key建立B+树，叶子节点存的就是数据，避免了回行

非聚簇索引：叶子节点不存储数据，存储的是数据行地址，也就是说根据索引查找到数据行的位置再去磁盘查找数据，MyISM使用的是非聚簇索引，没有聚簇索引

InnoDB默认对主键建立聚簇索引。如果你不指定主键，InnoDB会用一个具有唯一且非空值的索引来代替。如果不存在这样的索引，InnoDB会定义一个隐藏的主键，然后对其建立聚簇索引。一般来说，InnoDB 会以聚簇索引的形式来存储实际的数据，它是其它二级索引的基础。

假设对 InnoDB 引擎上表name字段加索引，那么name索引叶子页面则只会存储主键id：

检索时，先通过name索引树找到主索引id，再通过id在主索引树的聚簇索引叶子页面取出数据.

如何减少回表

覆盖索引+延时关联(3条消息) mysql优化：覆盖索引（延迟关联）_ekb_technology的博客-CSDN博客

延时关联：通过使用覆盖索引查询返回需要的主键,再根据主键关联原表获得需要的数据

```
select * from table where xxx limit a,b;

select * from table where id in (select id from table where xxx limit a,b);
```

在覆盖索引的场景下，第一条的执行逻辑是

1. 通过索引找到(a+b)条符合查询条件的记录id
2. 再通过(a+b)个id回表查询这(a+b)条记录

3．最后按分页条件给用户返回b条记录

而第二条SQL的执行逻辑则是

1．通过索引找到(a+b)条符合查询条件的记录id

2．按分页条件取b个记录id，然后回表查询这b条记录

3．最后给用户返回b条记录

不难看出，第二条SQL在覆盖索引的场景下，减少了大量的回表执行次数，从而提高了执行效率。而在非索引覆盖的场景下，延时关联失效，两种SQL的执行速度没有多少区别。

优缺点

优点：

- 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
- 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

缺点：

- 时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，会降低增/改/删的执行效率；
- 空间方面：索引需要占物理空间。

索引类型

主键索引: 数据列不允许重复，不允许为NULL，一个表只能有一个主键。

唯一索引: 数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。

- 可以通过 ALTER TABLE table_name ADD UNIQUE (column); 创建唯一索引
- 可以通过 ALTER TABLE table_name ADD UNIQUE (column1,column2); 创建唯一组合索引

普通索引: 基本的索引类型，没有唯一性的限制，允许为NULL值。

- 可以通过ALTER TABLE table_name ADD INDEX index_name (column);创建普通索引
- 可以通过ALTER TABLE table_name ADD INDEX index_name(column1, column2, column3);创建组合索引

全文索引：是目前搜索引擎使用的一种关键技术。

- 可以通过ALTER TABLE table_name ADD FULLTEXT (column);创建全文索引

索引算法

索引算法有 BTree算法和Hash算法

BTree算法

BTree是最常用的mysql数据库索引算法，也是mysql默认的算法。因为它不仅可以被用在=,>,>=,<,<=和between这些比较操作符上，而且还可以用于like操作符，只要它的查询条件是一个不以通配符开头的常量， 例如：

```
-- 只要它的查询条件是一个不以通配符开头的常量
select * from user where name like 'jack%';
-- 如果一通配符开头，或者没有使用常量，则不会使用索引，例如：
select * from user where name like '%jack';
```

Hash算法

Hash Hash索引只能用于对等比较，例如=,<=>（相当于=）操作符。由于是一次定位数据，不像BTree索引需要从根节点到枝节点，最后才能访问到页节点这样多次IO访问，所以检索效率远高于BTree索引。

索引的创建删除

创建

建表时创建

```
CREATE TABLE user_index2 (
    id INT auto_increment PRIMARY KEY,
    first_name VARCHAR (16),
    last_name VARCHAR (16),
    id_card VARCHAR (18),
    information text,
    KEY name (first_name, last_name),
    FULLTEXT KEY (information),
    UNIQUE KEY (id_card)
);
```

alter table

```
ALTER TABLE table_name ADD INDEX index_name (column_list);
```

CREATE INDEX

```
CREATE INDEX index_name ON table_name (column_list);
```

删除

```
//alter table 表名 drop KEY 索引名
alter table user_index drop KEY name;
alter table user_index drop KEY id_card;
alter table user_index drop KEY information;
```

百万级别或以上的数据如何删除

1. 所以我们想要删除百万数据的时候可以先删除索引（此时大概耗时三分多钟）
2. 然后删除其中无用数据（此过程需要不到两分钟）
3. 删除完成后重新创建索引(此时数据较少了)创建索引也非常快，约十分钟左右。
4. 与之前的直接删除绝对是要快速很多，更别说万一删除中断,一切删除会回滚。那更是坑了。

前缀索引

语法：index(field(10))，使用字段值的前10个字符建立索引，默认是使用字段的全部内容建立索引。
前提：前缀的标识度高。比如密码就适合建立前缀索引，因为密码几乎各不相同。
实操的难度：在于前缀截取的长度。
我们可以利用select count(*)/count(distinct left(password,prefixLen));，通过从调整prefixLen的值（从1自增）查看不同前缀长度的一个平均匹配度，接近1时就可以了（表示一个密码的前prefixLen个字符几乎能确定唯一一条记录）

in 和 exists 区别

mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

1. 如果查询的两个表大小相当，那么用in和exists差别不大。
2. 如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in。
3. not in 和not exists：如果查询语句使用了not in，那么内外表都进行全表扫描，没有用到索引；而not extsts的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

exists

语法：

```
//SELECT 字段 FROM table WHERE EXISTS (subquery);
//subquery是一个受限的SELECT语句（不允许有COMPUTE子句和INTO关键字）
SELECT * FROM A WHERE EXISTS (SELECT 1 FROM B WHERE B.id = A.id);
```

EXISTS执行顺序：
1、首先执行一次外部查询，并缓存结果集，如 SELECT * FROM A
2、遍历外部查询结果集的每一行记录R，代入子查询中作为条件进行查询，如 SELECT 1 FROM B WHERE B.id = A.id
3、如果子查询有返回结果，则EXISTS子句返回TRUE，这一行R可作为外部查询的结果行，否则不能作为结果

组合索引

组合索引可以这样理解，比如（a,b,c），abc都是排好序的，在任意一段a的下面b都是排好序的，任何一段b下面c都是排好序的；

生效的规则是：从前往后依次使用生效，如果中间某个索引没有使用，那么断点前面的索引部分起作用，断点后面的索引没有起作用；

> ⚠️ 最左匹配原则遇到范围查询（>、<、between、like）就会停止匹配

```
where a=3 and b=45 and c=5 .... 这种三个索引顺序使用中间没有断点，全部发挥作用；
where a=3 and c=5... 这种情况下b就是断点，a发挥了效果，c没有效果
where b=3 and c=4... 这种情况下a就是断点，在a后面的索引都没有发挥作用，这种写法联合索引没有发挥任何效果；
where b=45 and a=3 and c=5 .... 这个跟第一个一样，全部发挥作用，abc只要用上了就行，跟写的顺序无关
```

```
(0)    select * from mytable where a=3 and b=5 and c=4;
abc三个索引都在where条件里面用到了，而且都发挥了作用
(1)    select * from mytable where  c=4 and b=6 and a=3;
这条语句列出来只想说明 mysql没有那么笨，where里面的条件顺序在查询之前会被mysql自动优化，效果跟上一句一样
(2)    select * from mytable where a=3 and c=7;
a用到索引，b没有用，所以c是没有用到索引效果的
(3)    select * from mytable where a=3 and b>7 and c=3;
a用到了，b也用到了，c没有用到，这个地方b是范围值，也算断点，只不过自身用到了索引
(4)    select * from mytable where b=3 and c=4;
因为a索引没有使用，所以这里 bc都没有用上索引效果
(5)    select * from mytable where a>4 and b=7 and c=9;
a用到了  b没有使用，c没有使用
(6)    select * from mytable where a=3 order by b;
a用到了索引，b在结果排序中也用到了索引的效果，前面说了，a下面任意一段的b是排好序的
(7)    select * from mytable where a=3 order by c;
a用到了索引，但是这个地方c没有发挥排序效果，因为中间断点了，使用 explain 可以看到 filesort
(8)    select * from mytable where b=3 order by a;
b没有用到索引，排序中a也没有发挥索引效果
```

SQL优化

查询数据超多总表数据的30%时，会不走索引，直接全表查询

1. 查询SQL尽量不要使用select *，而是select具体字段。
2. 如果知道查询结果只有一条或者只要最大/最小一条记录，建议用limit 1
3. 应尽量避免在where子句中使用or来连接条件
4. 优化limit分页，当偏移量最大的时候，查询效率就会越低，因为Mysql并非是跳过偏移量直接去取后面的数据，而是先把偏移量+要取的条数，然后再把前面偏移量这一段的数据抛弃掉再返回的。

5. 优化你的like语句，把%放前面，并不走索引，把% 放关键字后面，还是会走索引的。
6. 尽量避免在索引列上使用mysql的内置函数，会导致索引失效
7. 应尽量避免在where子句中对字段进行表达式操作，这将导致系统放弃使用索引而进行全表扫
8. Inner join 、left join、right join，优先使用Inner join，如果是left join，左边表结果尽量小
9. 应尽量避免在where子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。
10. 对查询进行优化，应考虑在where及order by涉及的列上建立索引，尽量避免全表扫描。
11. 如果插入数据过多，考虑批量插入。
12. 在适当的时候，使用覆盖索引。
13. 在查询字段很多的时候慎用distinct关键字
14. 删除冗余和重复索引
15. 如果数据量较大，优化你的修改/删除语句。一次性删除太多数据，可能会有lock wait timeout exceed的错误，所以建议分批操作。
16. where子句中考虑使用默认值代替null。并不是说使用了is null 或者 is not null 就会不走索引了，这个跟mysql版本以及查询成本都有关。

如果mysql优化器发现，走索引比不走索引成本还要高，肯定会放弃索引，这些条件 `！=，>isnull，isnotnull` 经常被认为让索引失效，其实是因为一般情况下，查询的成本高，优化器自动放弃索引的。如果把null值，换成默认值，很多时候让走索引成为可能，同时，表达意思会相对清晰一点。

17. exist&in的合理利用，主表小用exist，主表大用in
18. 尽量用union all替换 union，如果检索结果中不会有重复的记录，推荐union all 替换 union。
19. 索引不宜太多，一般5个以内。
20. 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型
21. 索引不适合建在有大量重复数据的字段上，如性别这类型数据库字段。
22. 尽量避免向客户端返回过多数据量
23. 尽可能使用varchar/nvarchar 代替 char/nchar。
24. 为了提高group by 语句的效率，可以在执行到该语句前，把不需要的记录过滤掉。
25. 如果字段类型是字符串，where时一定用引号括起来，否则索引失效。这是因为不加单引号时，是字符串跟数字的比较，它们类型不匹配，MySQL会做隐式的类型转换，把它们转换为浮点数再做比较。
26. 使用explain 分析你SQL的计划

索引失效

1. 最佳左前缀法则，如果左边的值未确定，那么无法使用此索引。
2. 计算、函数、类型转换(自动或手动：比如列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引)导致索引失效
3. 范围条件右边的列索引失效
4. 不等于(!= 或者<>)导致索引失效

5. is null可以使用索引，is not null无法使用索引
6. like以通配符%开头索引失效
7. 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。
8. OR 前后只要存在非索引的列，都会导致索引失效
9. 统一使用utf8mb4(5.5.3 版本以上支持) 兼容性更好，统一字符集可以避免由于字符集转换产生的乱码。不同的 字符集 进行比较前需要进行 转换 会造成索引失效。

MySQL 和Oracle的区别

MySQL 是轻量型数据库，并且免费，没有服务恢复数据。 Oracle 是重量型数据库，收费，Oracle公司对Oracle数据库有任何服务。

- 对事务的提交

MySQL默认是自动提交，而Oracle默认不自动提交，需要用户手动提交，需要在写commit;指令或者点击commit按钮

- 分页查询

MySQL是直接在SQL语句中写"select... from ...where...limit x, y",有limit就可以实现分页;而Oracle则是需要用到伪列ROWNUM和嵌套查询

- 对事务的支持

MySQL在innodb存储引擎的行级锁的情况下才可支持事务，而Oracle则完全支持事务

- 并发性

MySQL以表级锁为主，对资源锁定的粒度很大，如果一个session对一个表加锁时间过长，会让其他session无法更新此表中的数据。 虽然InnoDB引擎的表可以用行级锁，但这个行级锁的机制依赖于表的索引，如果表没有索引，或者sql语句没有使用索引，那么仍然使用表级锁。 Oracle使用行级锁，对资源锁定的粒度要小很多，只是锁定sql需要的资源，并且加锁是在数据库中的数据行上，不依赖与索引。所以Oracle对并发性的支持要好很多。

- 单引号的处理

MYSQL里可以用双引号包起字符串，ORACLE里只可以用单引号包起字符串。在插入和修改字符串前必须做单引号的替换：把所有出现的一个单引号替换成两个单引号。

- 空字符的处理

MYSQL的非空字段也有空的内容，ORACLE里定义了非空字段就不容许有空的内容。按MYSQL的NOT NULL来定义ORACLE表结构, 导数据的时候会产生错误。因此导数据时要对空字符进行判断，如果为NULL或空字符，需要把它改成一个空格的字符串。

- 字符串的模糊比较

MYSQL里用 字段名 like '%字符串%',ORACLE里也可以用 字段名 like '%字符串%' 但这种方法不能使用索引, 速度不快，用字符串比较函数 instr(字段名,'字符串')>0 会得到更精确的查找结果。

主从

1. master提交完事务后，写入binlog

2. slave连接到master，获取binlog
3. master创建dump线程，推送binglog到slave
4. slave启动一个IO线程读取同步过来的master的binlog，记录到relay log中继日志中
5. slave再开启一个sql线程读取relay log事件并在slave执行，完成同步
6. slave记录自己的binglog
由于mysql默认的复制方式是异步的，主库把日志发送给从库后不关心从库是否已经处理，这样会产生一个问题就是假设主库挂了，从库处理失败了，这时候从库升为主库后，日志就丢失了
源到副本复制
1、异步复制

2、半同步复制（需要副本确认收到事务，主节点才会提交）

组复制
将执行源 1 接收的事务。然后，源 1 向复制组发送一条消息，该复制组由自身、源 2 和源 3 组成。当所有三个成员达成共识时，他们将对交易进行认证。然后，源 1 将事务写入其二进制日志，提交它，并向客户端应用程序发送响应。源 2 和 3 将事务写入其中继日志，然后应用它，将其写入二进制日志，然后提交它。

日志

- 逻辑日志：可以简单理解为记录的就是sql语句 。
- 物理日志：`mysql` 数据最终是保存在数据页中的，物理日志记录的就是数据页变更 。

binlog

二进制日志包含描述数据库更改（如表创建操作或表数据更改）的“事件”。它还包含可能已进行更改的语句的事件（例如，与任何行都不匹配的 `DELETE`），除非使用基于行的日志记录。二进制日志还包含有关每个语句花费了更新数据多长时间的信息。通过追加的方式进行写入的，可以通过`max_binlog_size`参数设置每个 `binlog`文件的大小，当文件大小达到给定值之后，会生成新的文件来保存日志。
二进制日志有两个重要用途：

- 对于复制，复制源服务器上的二进制日志提供要发送到副本的数据更改的记录。源将其二进制日志中包含的信息发送到其副本，副本将重现这些事务以进行与在源上所做的相同数据更改。
- 某些数据恢复操作需要使用二进制日志。还原备份后，将重新执行二进制日志中在进行备份后记录的事件。这些事件使数据库从备份点开始保持最新状态。

二进制日志不用于不修改数据的语句，如 `SELECT` 或 `SHOW`。
mysql binlog的应用场景有很多，如主从同步、数据恢复、数据备份等等
binlog本质是一个二进制文件，那又如何解析和使用呢？现成的工具已经有很多了，如Canal、maxwell。

Canal是阿里巴巴旗下的一款开源项目，纯Java开发。基于数据库增量日志解析，提供增量数据订阅&消费。Canal分为服务端和客户端，拥有众多的衍生应用，性能稳定，功能强大。canal的工作原理很简单，就是把自己伪装成slave，假装从master复制数据。

Maxwell由zendesk开源，也是由java开发，解析出来的结果为json，可以方便的发送到kafka、rabbitmq、redis等下游应用，进行处理。

binlog刷盘时机

对于 InnoDB 存储引擎而言，只有在事务提交时才会记录binlog ，此时记录还在内存中，那么 biglog是什么时候刷到磁盘中的呢？mysql 通过 sync_binlog 参数控制 biglog 的刷盘时机，取值范围是 0-N：

- 0：不去强制要求，由系统自行判断何时写入磁盘；
- 1：每次 commit 的时候都要将 binlog 写入磁盘；
- N：每N个事务，才会将 binlog 写入磁盘。

从上面可以看出， sync_binlog 最安全的是设置是 1 ，这也是MySQL 5.7.7之后版本的默认值。但是设置一个大一些的值可以提升数据库性能，因此实际情况下也可以将值适当调大，牺牲一定的一致性来获取更好的性能。

binlog日志格式

binlog 日志有三种格式，分别为 STATMENT 、 ROW 和 MIXED 。

> 在 MySQL 5.7.7 之前，默认的格式是 STATEMENT， MySQL 5.7.7 之后，默认值是 ROW。日志格式通过 binlog-format 指定。

- STATMENT：基于SQL 语句的复制(statement-based replication, SBR)，每一条会修改数据的sql语句会记录到binlog 中 。
  - 优点：不需要记录每一行的变化，减少了 binlog 日志量，节约了 IO , 从而提高了性能；
  - 缺点：在某些情况下会导致主从数据不一致，比如执行sysdate() 、 slepp() 等 。
- ROW：基于行的复制(row-based replication, RBR)，不记录每条sql语句的上下文信息，仅需记录哪条数据被修改了 。
  - 优点：不会出现某些特定情况下的存储过程、或function、或trigger的调用和触发无法被正确复制的问题 ；
  - 缺点：会产生大量的日志，尤其是alter table 的时候会让日志暴涨
- MIXED：基于STATMENT 和 ROW 两种模式的混合复制(mixed-based replication, MBR)，一般的复制使用STATEMENT 模式保存 binlog ，对于 STATEMENT 模式无法复制的操作使用 ROW 模式保存 binlog

Relay Log

relay-log中继日志是连接master和slave的核心

中继日志与二进制日志一样，由一组编号的文件组成，其中包含描述数据库更改的事件，以及一个包含所有使用的中继日志文件名称的索引文件。中继日志文件的默认位置是数据目录。

副本服务器在下列情况下创建新的中继日志文件：

- 每次复制 I/O（接收器）线程启动时。
- 当日志被刷新时（例如，使用 FLUSH LOGS 或 mysqladmin flush-logs）。
- 当当前中继日志文件的大小变得过大时，其确定方式如下：
  - 如果 max_relay_log_size 的值大于 0，则这是最大中继日志文件大小。
  - 如果 max_relay_log_size 的值为 0，则max_binlog_size确定最大中继日志文件大小。

复制 SQL（应用程序）线程在执行了文件中的所有事件并且不再需要它后，会自动删除每个中继日志文件。没有用于删除中继日志的显式机制，因为复制 SQL 线程负责执行此操作。但是，FLUSH LOGS会轮换中继日志，这会影响复制 SQL 线程删除它们的时间。

redo log

为InnoDB存储引擎层的日志,redo log用于保证事务的持久性，即ACID中的D。redo log有两种类型，分别为物理重做日志和逻辑重做日志。 在InnoDB中redo log大多数情况下是一个物理日志，记录数据页面的物理变化（实际的数据值）。

redo log的主要功能是用于数据库崩溃时的数据恢复。

重做日志是在崩溃恢复期间使用的基于磁盘的数据结构，用于更正由未完成事务写入的数据。在正常操作期间，重做日志会对更改由 SQL 语句或低级 API 调用生成的表数据的请求进行编码。在意外关闭之前未完成数据文件更新的修改将在初始化期间和接受连接之前自动重播。有关重做日志在崩溃恢复中的作用的信息

redo log 是物理日志，记录的是“在某个数据页上做了什么修改”；**binlog** 是逻辑日志，记录的是这个语句的原始逻辑(sql语句及其反向操作)。

redo log是循环写，空间固定、会用完；checkpoint之前表示擦除完了的，即可以进行写的，擦除之前会更新到磁盘中，write pos是指写的位置，当write pos和checkpoint相遇的时候表明redo log已经满了，这个时候数据库停止进行数据库更新语句的执行，转而进行redo log日志同步到磁盘中。

binlog是追加写，是指一份写到一定大小的时候会更换下一个文件，不会覆盖以前的日志。

innodb事务日志包括**redo log**和**undo log**：

1. redo log通常是物理日志，记录的是数据页的物理修改，它用来恢复提交后的物理数据页(恢复数据页，且只能恢复到最后一次提交的位置)。
2. undo用来回滚行记录到某个版本。undo log一般是逻辑日志，根据每行记录进行记录。

为什么需要redo log

我们都知道，事务的四大特性里面有一个是 **持久性** ，具体来说就是**只要事务提交成功，那么对数据库做的修改就被永久保存下来了，不可能因为任何原因再回到原来的状态** 。

那么 mysql是如何保证一致性的呢？最简单的做法是在每次事务提交的时候，将该事务涉及修改的数据页全部刷新到磁盘中。但是这么做会有严重的性能问题，主要体现在两个方面：

1. 因为 Innodb 是以 页 为单位进行磁盘交互的，而一个事务很可能只修改一个数据页里面的几个字节，这个时候将完整的数据页刷到磁盘的话，太浪费资源了！
2. 一个事务可能涉及修改多个数据页，并且这些数据页在物理上并不连续，使用随机IO写入性能太

差！

因此 mysql 设计了 redo log， **具体来说就是只记录事务对数据页做了哪些修改**，这样就能完美地解决性能问题了(相对而言文件更小并且是顺序IO)。

redo log包括两部分：一个是内存中的日志缓冲(redo log buffer)，另一个是磁盘上的日志文件(redo logfile)。mysql每执行一条DML语句，先将记录写入redo log buffer，后续某个时间点再一次性将多个操作记录写到 redo log file。这种 **先写日志，再写磁盘**的技术就是 MySQL里经常说到的 WAL(Write-Ahead Logging) 技术。在计算机操作系统中，用户空间(user space)下的缓冲区数据一般情况下是无法直接写入磁盘的，中间必须经过操作系统内核空间(kernel space)缓冲区(OS Buffer)。因此， redo log buffer 写入 redo logfile 实际上是先写入 OS Buffer ，然后再通过系统调用 fsync() 将其刷到 redo log file

中，过程如下：

undo log

保存了事务发生之前的数据的一个版本，可以用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读

逻辑格式的日志，在执行undo的时候，仅仅是将数据从逻辑上恢复至事务之前的状态，而不是从物理页面上操作实现的，这一点是不同于redo log的

撤消日志是与单个读写事务关联的撤消日志记录的集合。撤消日志记录包含有关如何撤消事务对聚集索引记录的最新更改的信息。如果另一个事务需要将原始数据视为一致读取操作的一部分，则会从撤消日志记录中检索未修改的数据。撤消日志存在于撤消日志段中，这些日志段包含在回滚段中。回滚段驻留在撤消表空间和全局临时表空间中。

页

- 为了避免一条一条读取磁盘数据，InnoDB采取页的方式，作为磁盘和内存之间交互的基本单位。
- 一个页的大小一般是16KB。
- InnoDB为了不同的目的而设计了多种不同类型的页。比如：存放表空间头部信息的页、存放undo日志信息的页等等。
- 存放表中数据记录的页，称为索引页或者数据页。

数据库时间类型的选择 Datetime 或者 Timestamp

1.切记不要用字符串存储日期

这种存储日期的方式的优点还是有的，就是简单直白，容易上手。但是，这是不正确的做法，主要会有下面两个问题：

1. 字符串占用的空间更大！
2. 字符串存储的日期比较效率比较低（逐个字符进行比对），无法用日期相关的 API 进行计算和比

较。

2.Datetime 和 Timestamp 之间抉择 Datetime 和 Timestamp 是 MySQL 提供的两种比较相似的保存时间的数据类型。他们两者究竟该如何选择呢?

通常我们都会首选 Timestamp。 下面说一下为什么这样做!

2.1 DateTime 类型没有时区信息的 DateTime 类型是没有时区信息的（时区无关） ，DateTime 类型保存的时间都是当前会话所设置的时区对应的时间。这样就会有什么问题呢？当你的时区更换之后，比如你的服务器更换地址或者更换客户端连接时区设置的话，就会导致你从数据库中读出的时间错误。不要小看这个问题，很多系统就是因为这个问题闹出了很多笑话。

Timestamp 和时区有关。Timestamp 类型字段的值会随着服务器时区的变化而变化，自动换算成相应的时间，说简单点就是在不同时区，查询到同一个条记录此字段的值会不一样。

2.2 DateTime 类型耗费空间更大

Timestamp 只需要使用 4 个字节的存储空间，但是 DateTime 需要耗费 8 个字节的存储空间。但是，这样同样造成了一个问题，Timestamp 表示的时间范围更小。

- DateTime ： 1000–01–01 00:00:00 ~ 9999–12–31 23:59:59
- Timestamp： 1970–01–01 00:00:01.000000到 2038–01–19 03:14:07.999999(准备的来讲应该是UTC范围)；

也可以用时间戳，但是不够直观

不同数据类型占用空间

Data Type	Storage Required
TINYINT	1 byte
SMALLINT	2 bytes
MEDIUMINT	3 bytes
INT , INTEGER	4 bytes
BIGINT	8 bytes
FLOAT(*p*)	4 bytes if 0 <= **p** <= 24, 8 bytes if 25 <= **p** <= 53
FLOAT	4 bytes
DOUBLE [PRECISION] , REAL	8 bytes
DECIMAL(*M*,*D*), NUMERIC(*M*,*D*)	Varies; see following discussion
BIT(*M*)	approximately (**M**+7)/8 bytes

Data Type	Storage Required Before MySQL 5.6.4	Storage Required as of MySQL 5.6.4
YEAR	1 byte	1 byte
DATE	3 bytes	3 bytes
TIME	3 bytes	3 bytes + fractional seconds storage
DATETIME	8 bytes	5 bytes + fractional seconds storage
TIMESTAMP	4 bytes	4 bytes + fractional seconds storage

常用sql

```
#设置自增ID从1开始
alter table tablename AUTO_INCREMENT =1;
#显示用户正在运行的线程,分析sql当前的运行状态
show processlist;
#显示表结构
desc table_name
#去重
SELECT DISTINCT Store_Name FROM Store_Info;
```

存储过程

```
#创建一个存储过程，输入一个分数后返回是分数的评判等级
create procedure p1(in score int, out result varchar(10))
begin
     if score >= 85 then
        set result := '优秀';
     elseif score >= 60 then
        set result := '及格';
     eles
        set result := '不及格';
     end if;
end;
call p1(85,@result);
```

```
13
14 #将传入的200分制的分数，进行换算，换成百分制，然后返回分数
15 create procedure p2(inout score duble)
16 begin
17      set score := score * 0.5;
18 end;

19 set @score = 78;
20 call p2(@score);
21
22  #while实现1到n的累加
23 create procedure p3(in n int)
24 begin
25      declare total int default 0;
26      while n>0 do
27            set total := total + n;
28            set n := n -1;
29      end while;
30      select total;
31 end;
32
33 #loop实现1到n的奇数的累加
34 create procedure p5(in n int)
35 begin
36      declare total int default 0;
37      sum:loop
38          if n<=0 then
39             #退出指定标记的循环体
40             leave sum;
41          end if;
42          if n%2 = 1 then
43             set n := n -1;
44             #进入下一次循环
45             iterate sum;
46          end if;
47          set total := total + n;
48          set n := n -1;
49      end loop sum;
50      select total;
51 end;
```

```
52
53 #case when使用
54 create procedure p6(in month int)
55 begin
56      declare result varchar(10)
57      case
58          when month >= 1 and month <= 3 then
59               set result := '第一季度';
60          when month >=4 and month <= 6 then
61               set result := '第二季度';
62          when month >= 7 and month <= 9 then
63               set result := '第三季度';
64          when month >=10 and month <= 12 then
65               set result := '第四季度';
66          else
67               set result := '非法参数';
68     end case;
69     select result;
70 end;
71
72 call p6(6)
73 #游标的使用
74 #输入一个年龄，把tb_user表中小于这个年龄的用户的信息插入到另一个表tb_user_pro中
75 create procedure p7(int uage int)
76 begin
77      declare u_cursor cursor for select name,profession from tb_user where age <=uage;
78      declare uname varchar(10);
79      declare upro varchar(100);
80      --条件处理程序，当下面while循环时游标的数据为空时就不会报错了。
81      declare exit handler for NOT FOUND close u_cuesor;
82      drop table id exists tb_user_pro;
83      create table if not exists tb_user_pro(
84           id int primary key auto_increment,
85           name varchar(10),
86           profession varchar(100)
87      );
88      open u_cursor;
89      while true do
90           fetch u_cursor into uname,upro;
```

```
91            insert into tb_user_pro values(null,uname,upro);
92        end while;
93        close u_cursor;
94    end;
95
96
97
98
```

分库分表

垂直分库

将库按照业务拆分为多个库

垂直分表

如果表字段比较多，将不常用的、数据较大的等等做拆分

水平分表

首先根据业务场景来决定使用什么字段作为分表字段(sharding_key)，比如我们现在日订单1000万，我
们大部分的场景来源于C端，我们可以用user_id作为sharding_key，数据查询支持到最近3个月的订单，超过3个月的做归档处理，那么3个月的数据量就是9亿，可以分1024张表，那么每张表的数据大概
就在100万左右。
比如用户id为100，那我们都经过hash(100)，然后对1024取模，就可以落到对应的表上了。

分表后的ID怎么保证唯一性

1. 设定步长，比如1-1024张表我们设定1024的基础步长，这样主键落到不同的表就不会冲突了。
2. 分布式ID，自己实现一套分布式ID生成算法或者使用开源的比如雪花算法这种
3. 分表后不使用主键作为查询依据，而是每张表单独新增一个字段作为唯一主键使用，比如订单表订单号是唯一的，不管最终落在哪张表都基于订单号作为查询依据，更新也一样。

 雪花id：

snowflake的结构如下(每部分用-分开):

0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 - 000000000000

第 1 位标识位：符号标识，由于ID一般是正数，第一位一般固定 0,第一位未使用

后 41 位时间戳：毫秒级，可以存储起始时间到当前时间的时间戳的差值（取值范围是 0~2^41 -1），最多可以使用约 69 年，(1L << 41) / (1000L * 60 * 60 * 24 * 365) = 69

后 10 位机器位：可以部署1024个节点，包含 5 位数据中心ID（取值范围是0~31）（dataCenterId）和 5 位工作机器ID（取值范围是0~31）（workerId）

后 12 位序列位：毫秒内的计算。12 位的计数顺序号支持每个节点，每毫秒生成 4096 个序号

总 64 位：Long,转换成字符串后长度最多19

整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞(由datacenter和workerId作区分)

分表后非sharding_key的查询怎么处理

1. 可以做一个mapping表，比如这时候商家要查询订单列表怎么办呢？不带user_id查询的话你总不能扫全表吧？所以我们可以做一个映射关系表，保存商家和用户的关系，查询的时候先通过商家查询到用户列表，再通过user_id去查询。
2. 打宽表，一般而言，商户端对数据实时性要求并不是很高，比如查询订单列表，可以把订单表同步到离线（实时）数仓，再基于数仓去做成一张宽表，再基于其他如es提供查询服务。
3. 数据量不是很大的话，比如后台的一些查询之类的，也可以通过多线程扫表，然后再聚合结果的方式来做。或者异步的形式也是可以的。

主从的延迟怎么解决

1. 针对特定的业务场景，读写请求都强制走主库
2. 读请求走从库，如果没有数据，去主库做二次查询

MyBatis

什么是 MyBatis

MyBatis 是一个可以自定义 SQL、存储过程（statementType="CALLABLE"）和高级映射的持久层框架。

拦截器

类型

- Executor(update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)：拦截执行器的方法。
- ParameterHandler (getParameterObject, setParameters)：拦截参数的处理。
- ResultHandler (handleResultSets, handleOutputParameters)：拦截结果集的处理。
- StatementHandler (prepare, parameterize, batch, update, query)：拦截Sql语法构建的处理。

其实Executor都可以实现ParameterHandler，ResultHandler，StatementHandler拦截器的功能

执行流程

mybatis在执行过程中按照Executor => StatementHandler => ParameterHandler => ResultSetHandler。
Executor在执行过程中会创建StatementHandler，在创建StatementHandler过程中会创建ParameterHandler和ResultSetHandler。

使用

简单的只需实现 Interceptor 接口，并在类中指定想要拦截的方法签名即可

```
@Intercepts({@Signature(type= Executor.class,method = "update",args = {MappedStatement.class,Object.class})})
public class ExamplePlugin implements Interceptor {
  private Properties properties = new Properties();
  public Object intercept(Invocation invocation) throws Throwable {
    // implement pre processing if need
    Object returnObject = invocation.proceed();
    // implement post processing if need
    return returnObject;
  }
  public void setProperties(Properties properties) {
    this.properties = properties;
  }
}
```

MyBatis 的缓存

MyBatis 的缓存分为一级缓存和二级缓存,一级缓存放在 session 里面,默认就有,二级缓存放在它的命名空间里,默认是不打开的,使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置 `<cache/>`

一级缓存:

在参数和SQL完全一样的情况下，我们使用同一个SqlSession对象调用一个Mapper方法，往往只执行一次SQL，因为使用SelSession第一次查询后，MyBatis会将其放在缓存中，以后再查询的时候，如果

没有声明需要刷新，并且缓存没有超时的情况下，SqlSession都会取出当前缓存的数据，而不会再次发送SQL到数据库。
一级缓存时执行commit，close，增删改等操作，就会清空当前的一级缓存；当对SqlSession执行更新操作（update、delete、insert）后并执行commit时，不仅清空其自身的一级缓存（执行更新操作的效果），也清空二级缓存（执行commit()的效果）。
二级缓存：
二级缓存指的就是同一个namespace下的mapper，二级缓存中，也有一个map结构，这个区域就是一级缓存区域。一级缓存中的key是由sql语句、条件、statement等信息组成一个唯一值。一级缓存中的value，就是查询出的结果对象。

Mybatis 是如何进行分页的？分页插件的原理是什么

1）Mybatis 使用 RowBounds 对象进行分页，也可以直接编写 sql 实现分页，也可以使用Mybatis 的分页插件。
2）分页插件的原理：实现 Mybatis 提供的接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql。
举例：select from student，拦截 sql 后重写为：select t. from （select from student）t limit 0，10

Mybatis 动态 sql 是做什么的

1）Mybatis 动态 sql 可以让我们在 Xml 映射文件内，以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能。
2）Mybatis 提供了 9 种动态 sql 标签：
trim|where|set|foreach|if|choose|when|otherwise|bind。
3）其执行原理为，使用 OGNL 从 sql 参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态 sql 的功能。

#{}和${}的区别是什么

1）#{}是预编译处理，${}是字符串替换。
2）Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的 set 方法来赋值；
3）Mybatis 在处理${}时，就是把${}替换成变量的值。
4）使用#{}可以有效的防止 SQL 注入，提高系统安全性。

为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里

Hibernate 属于全自动 ORM 映射工具，使用 Hibernate 查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

MyBatis 与 Hibernate 有哪些不同

1）Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句，不过 mybatis 可以通过 XML 或注解方式灵活配置要运行的 sql 语句，并将java 对象和 sql 语句映射生成最终执行的 sql，最后将 sql 执行的结果再映射生成 java 对
象。
2）Mybatis 学习门槛低，简单易学，程序员直接编写原生态 sql，可严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是 mybatis 无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套 sql 映射文件，工作量大。
3）Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用 hibernate 开发可以节省很多代码，提高效率。但是Hibernate 的缺点是学习门槛高，要精通门槛更高，而且怎么设计 O/R 映射，在性能和对象模型之间如何权衡，以及怎样用好 Hibernate 需要具有很强的经验和能力才行。
总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。
mybatis：
1）入门简单，即学即用，提供数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美；2）可以进行更为细致的SQL优化，可以减少查询字段；
3）缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改；
4）二级缓存机制不佳。
hibernate：
1）功能强大，数据库无关性好，O/R映射能力强，如果你对Hibernate相当精通，而且对Hibernate进行了适当的封装，那么你的项目整个持久层代码会相当简单，需要写的代码很少，开发速度很快，非常爽；
2）有更好的二级缓存机制，可以使用第三方缓存；
3）缺点就是学习门槛不低，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何取得平衡，以及怎样用好Hibernate方面需要你的经验和能力都很强才行。

MyBatis 的好处是什么

1）MyBatis 把 sql 语句从 Java 源程序中独立出来，放在单独的 XML 文件中编写，给程序的维护带来了很大便利。
2）MyBatis 封装了底层 JDBC API 的调用细节，并能自动将结果集转换成 Java Bean 对象，大大简化了 Java 数据库编程的重复工作。
3）因为 MyBatis 需要程序员自己去编写 sql 语句，程序员可以结合数据库自身的特点灵活控制 sql 语句，因此能够实现比 Hibernate 等全自动 orm 框架更高的查询效率，能够完成复杂查询。

简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系

Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。在Xml 映射文件中，<parameterMap>标签会被解析为 ParameterMap 对象，其每个子元素会被解析为 ParameterMapping 对象。<resultMap>标签会被解析为 ResultMap 对象，其每个子元素会被解析为 ResultMapping 对象。每一个<select>、<insert>、<update>、<delete>标签均会被解析为 MappedStatement 对象，标签内的 sql 会被解析为 BoundSql 对象。

什么是 MyBatis 的接口绑定,有什么好处

接口绑定有两种实现方式

一种是通过注解绑定,就是在接口的方法上面加上@Select@Update 等注解里面包含 Sql 语句来绑定

另外一种就是通过 xml 里面写 SQL 来绑定,在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名.

> 当 Sql 语句比较简单时候,用注解绑定；当 SQL 语句比较复杂时候,用 xml 绑定,一般用xml 绑定的比较多

MyBatis 实现一对一有几种方式

有联合查询和嵌套查询

联合查询是几个表联合查询,只查询一次,通过在 resultMap 里面配置 association 节点配置一对一的类就可以完成

嵌套查询是先查一个表,根据这个表里面的结果的外键 id,去再另外一个表里面查询数据,也是通过 association 配置,但另外一个表的查询通过 select 属性配置。

结果映射

- constructor

- 用于在实例化类时，注入结果到构造方法中
  - idArg – ID 参数；标记出作为 ID 的结果可以帮助提高整体性能
  - arg – 将被注入到构造方法的一个普通结果
- id – 一个 ID 结果；标记出作为 ID 的结果可以帮助提高整体性能
- result – 注入到字段或 JavaBean 属性的普通结果
- association– 一个复杂类型的关联；许多结果将包装成这种类型
  - 嵌套结果映射 – 关联可以是 `resultMap` 元素，或是对其它结果映射的引用
- collection– 一个复杂类型的集合
  - 嵌套结果映射 – 集合可以是 `resultMap` 元素，或是对其它结果映射的引用
- discriminator– 使用结果值来决定使用哪个resultMap
  - case– 基于某些值的结果映射

- 嵌套结果映射 — case 也是一个结果映射，因此具有相同的结构和元素；或者引用其它的结果映射

Hibernate

多表联查

@ManyToOne

```
@ManyToOne(cascade = CascadeType.MERGE)
@JoinColumn(name="CASECODE", referencedColumnName="CASENO")
private CaseInfoModel caseInfo;
```

CASECODE 表示当前实体类对应数据库表中的列名称。
CASENO 表示关联表中的列名称，可以不是主键，不写的时候则默认为关联表中设置的主键列。

使用HQL语句进行多表联查

```
    String hql = select f from FileModel as f,StatusModel as s
    //前面不加select时，默认from后面两个实体都会查询出来。

    String where = Where f.statusId = s.Id and s.type = 2
```

使用HQL进行多表查询时，不能使用join on语句，sql形式hibernate不识别，会报错。

为什么要使用 hibernate

hibernate 是对 jdbc 的封装，大大简化了数据访问层的繁琐的重复性代码。
hibernate 是一个优秀的 ORM 实现，很多程度上简化了 DAO 层的编码功能。
可以很方便的进行数据库的移植工作。
提供了缓存机制，是程序执行更改的高效

hibernate 有几种查询方式

hql、原生 SQL、条件查询 Criteria。

hibernate 实体类可以被定义为 final 吗

实体类可以定义为 final 类，但这样的话就不能使用 hibernate 代理模式下的延迟关联提供性能了，所以不建议定义实体类为 final。

hibernate 是如何工作的

读取并解析配置文件。
读取并解析映射文件，创建 SessionFactory。
打开 Session。
创建事务。
进行持久化操作。
提交事务。
关闭 Session。
关闭 SessionFactory。

get()和 load()的区别

数据查询时，没有 OID 指定的对象，get() 返回 null；load() 返回一个代理对象。
load()支持延迟加载；get() 不支持延迟加载。
get：执行完get方法后，执行发送了sql语句，person对象也有了值。此时，在session关闭后，person对象依然存在，因此控制台可以打印出name的值。
load:在执行完load方法后，并未向数据库发送查询语句，此时person对象的值是一个CBLIB代理对象，这个代理对象只保存了实体对象的ID的值，如果在关闭session之前，执行person.getName()方法，程序将正常执行，正常发送查询请求，并打印出name的值。关闭session后，再次执行person.getName()方法，将出现延迟加载异常：
使用load方法的加载方式会比get方法的加载方式性能要好一些，因为load方法，得到的代理对象，只有在真正使用这个对象时，才会向数据库发送请求。如果不使用，就节省了向数据库查询的步骤，间接提升了系统性能。
get 会先查一级缓存，再查二级缓存，然后查数据库;load 会先查一级缓存，如果没有找到，就创建代理对象，等需要的时候去查询二级缓存和数据库。(这里就体现 load 的延迟加载的特性。)

hibernate 的缓存机制

hibernate 常用的缓存有一级缓存和二级缓存：
一级缓存：也叫 Session 缓存，只在 Session 作用范围内有效，不需要用户干涉，由 hibernate 自身维护，可以通过：evict(object)清除 object 的缓存；clear()清除一级缓存中的所有缓存；flush()刷出缓存；
二级缓存：应用级别的缓存，在所有 Session 中都有效，支持配置第三方的缓存，如：EhCache。

在 hibernate 中 getCurrentSession 和 openSession 的区别是什么

getCurrentSession 会绑定当前线程，而 openSession 则不会。
getCurrentSession 事务是 Spring 控制的，并且不需要手动关闭，而 openSession 需要我们自己手动开启和提交事务。

hibernate 实体类必须要有无参构造函数吗

hibernate 中每个实体类必须提供一个无参构造函数，因为 hibernate 框架要使用 reflection api，通过调用 ClassnewInstance() 来创建实体类的实例，如果没有无参的 构造函数就会抛出异常。

hibernate 中对象的三种状态

- 瞬时态(临时态、自由态):不存在持久化标识OID，尚未与 Hibernate Session 关联对象， 被认为处于瞬时态，失去引用将被 JVM 回收
- 持久态：存在持久化标识 OID，与当前 session 有关联，并且相关联的 session 没有关闭 ， 并且事务未提交
- 脱管态(离线态、游离态):存在持久化标识 OID，但没有与当前 session 关联，脱管状态 改变 hibernate 不能检测到

Redis

基本概念

Redis (Remote Dictionary Server) 是一个使用 C 语言 编写的，开源的 (BSD许可) 高性能 非关系型 (NoSQL) 的 键值对数据库。
key是统一的string
Redis 可以存储 键 和 不同类型数据结构值 之间的映射关系。键的类型只能是字符串，而值除了支持 最 基础的五种数据类型 外，还支持一些 高级数据类型：
与传统数据库不同的是 Redis 的数据是 存在内存 中的，所以 读写速度 非常 快，因此 Redis 被广泛应用于 缓存 方向，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value 数据库。另外，Redis 也经常用来做 分布式锁。
除此之外，Redis 支持事务 、持久化、LUA脚本、LRU驱动事件、多种集群方案。

为什么快

1. 完全基于内存操作
2. C语言实现，优化过的数据结构，基于几种基础的数据结构，redis做了大量的优化，性能极高
3. 使用单线程，无上下文的切换成本
4. 基于非阻塞的IO多路复用机制

Redis和MongoDB的区别

Redis主要把数据存储在内存中，其“缓存”的性质远大于其“数据存储“的性质，其中数据的增删改查也只是像变量操作一样简单；
MongoDB却是一个“存储数据”的系统，增删改查可以添加很多条件，就像SQL数据库一样灵活，这一点在面试的时候很受用。
应用场景：Redis较小数据量的性能及运算，支持事务，比较弱，mongoDB海量数据的访问效率提升，不支持事务

MongoDB是一个文档数据库，提供好的性能，领先的非关系型数据库。采用BSON存储文档数据。
BSON（）是一种类json的一种二进制形式的存储格式，简称Binary JSON.
相对于json多了date类型和二进制数组。

redis集群中的master节点个数要至少大于3个

因为master-1挂了以后，剩下的master节点（假如还有master-2、master-3）会重新选举出新的master，但是要超过一半的master节点个数都认为master-1挂了，这样才会重新选举新的master，如果是两个，挂了一个，还有一个，1>1不成立，所以要至少有3个master节点

集群节点个数要是奇数，虽然不是强制要求

奇数个master节点可以在满足选举该条件的基础上节省一个节点，比如三个master节点和四个master节点的集群相比，大家如果都挂了一个master节点都能选举新master节点，如果都挂了两个master节点都没法选举新master节点了，所以奇数的master节点更多的是从节省机器资源角度出发说的。
例如：
在9个master的架构中，如果4台master故障，通过过半机制，redis可以选举新的master。如果5台master故障无法选举新的master
在10个master的架构中，如果4台master故障，通过过半机制，redis可以选举新的master。如果5台master故障无法选举新的master
在高可用方面，9台master与10台master一致。所以通常会使用奇数。假设现在reids内存不足需要拓展，我们将master的数量加到11台，就高可用方面来说，就算其中5台master发送故障，也可以自动选举新的master。

基本数据类型

string 字符串，list 列表，hash 字典， set 集合，zset 有序列表

1. string，对应数据结构为：简单动态字符串，初始长度10，1M内扩容 length*2，超过1M，每次+1M，最大512M；
2. list，对应的数据结构为：双向链表和压缩列表；
3. hash，对应数据结构为：hash表和压缩列表；
4. set，对应数据结构为：hash表和整数数组；
5. sorted set，对应数据结构为：压缩列表和skiplist(跳表)；

跳表

上图中 level1，level2，level3 就是跳表的层级，每一个 level 层级都有一个指向下一个相同 level 层级元素的指针，比如上图我们遍历寻找元素 35 的时候就有三种方案：

- 第 1 种就是执行 level1 层级的指针，需要遍历 7 次（1–>8–>9–>12–>15–>20–>35）才能找到元

素 35。

- 第 2 种就是执行 level2 层级的指针，只需要遍历 5 次（1–>9–>12–>15–>35）就能找到元素 35。
- 第 3 种就是执行 level3 层级的元素，这时候只需要遍历 3 次（1–>12–>35）就能找到元素 35 了，大大提升了效率。

skiplist 的存储结构

跳跃表中的每个节点是一个 zskiplistNode 节点（源码 server.h 内）：

```
typedef struct zskiplistNode {
    sds ele;//元素
    double score;//分值
    struct zskiplistNode *backward;//后退指针
    struct zskiplistLevel {//层
        struct zskiplistNode *forward;//前进指针
        unsigned long span;//当前节点到下一个节点的跨度（跨越的节点数）
    } level[];
} zskiplistNode;
```

- level（层）

level 即跳跃表中的层，其是一个数组，也就是说一个节点的元素可以拥有多个层，即多个指向其他节点的指针，程序可以通过不同层级的指针来选择最快捷的路径提升访问速度。

level 是在每次创建新节点的时候根据幂次定律（power law）随机生成的一个介于 1~32 之间的数字。

- forward（前进指针）

每个层都会有一个指向链表尾部方向元素的指针，遍历元素的时候需要使用到前进指针。

- span（跨度）

跨度记录了两个节点之间的距离,需要注意的是，如果指向了 NULL 的话，则跨度为 0。

- backward（后退指针）

和前进指针不一样的是后退指针只有一个，所以每次只能后退至前一个节点（上图中没有画出后退指针）。

- ele（元素）

跳跃表中元素是一个 sds 对象（早期版本使用的是 redisObject 对象），元素必须唯一不能重复。

- score（分值）

节点的分值是一个 double 类型的浮点数，跳跃表中会将节点按照分值按照从小到大的顺序排列，不同节点的分值可以重复。

上面介绍的只是跳跃表中的一个节点，多个 zskiplistNode 节点组成了一个 zskiplist 对象：

```
typedef struct zskiplist {
    struct zskiplistNode *header, *tail;//跳跃表的头节点和尾结点指针
```

```
    unsigned long length;//跳跃表的节点数
    int level;//所有节点中最大的层数
} zskiplist;
```

到这里你可能以为有序集合就是用这个 zskiplist 来实现的，然而实际上 Redis 并没有直接使用 zskiplist 来实现，而是用 zset 对象再次进行了一层包装。

```
typedef struct zset {
    dict *dict;//字典对象
    zskiplist *zsl;//跳跃表对象
} zset;
```

所以最终，一个有序集合如果使用了 skiplist 编码，其数据结构如下图所示

上图中上面一部分中的字典中的 key 就是对应了有序集合中的元素（member），value 就对应了分值（score）。上图中下面一部分中跳跃表整数 1,8,9,12 也是对应了元素（member），最后一排的 double 型数字就是分值（score）。
也就是说字典和跳跃表中的数据都指向了我们存储的元素（两种数据结构最终指向的是同一个地址，所以数据并不会出现冗余存储），Redis 为什么要这么做呢?
为什么同时选择使用字典和跳跃表
有序集合直接使用跳跃表或者单独使用字典完全可以独自实现，但是我们想一下，如果单独使用跳跃表来实现，那么虽然可以使用跨度大的指针去遍历元素来找到我们需要的数据，但是其复杂度仍然达到了 O(logN)，而字典中获取一个元素的复杂度是 O(1)，而如果单独使用字典虽然获取元素很快，但是字典是无序的，所以如果要范围查找就需要对其进行排序，这又是一个耗时的操作，所以 Redis 综合了两种数据结构来最大程度的提升性能，这也是 Redis 设计的精妙之处。

ziplist压缩列表

ziplist 是为了节省内存而设计出来的一种数据结构。ziplist 是由一系列特殊编码组成的连续内存块的顺序型数据结构，一个 ziplist 可以包含任意多个 entry，而每一个 entry 又可以保存一个字节数组或者一个整数值。
ziplist 作为一种列表，其和普通的双端列表，如 linkedlist 的最大区别就是 ziplist 并不存储前后节点的指针，而 linkedlist 一般每个节点都会维护一个指向前置节点和一个指向后置节点的指针。那么 ziplist 不维护前后节点的指针，它又是如何寻找前后节点的呢?
ziplist 虽然不维护前后节点的指针，但是它却维护了上一个节点的长度和当前节点的长度，然后每次通过长度来计算出前后节点的位置。既然涉及到了计算，那么相对于直接存储指针的方式肯定有性能上的损耗，这就是一种典型的用时间来换取空间的做法。因为每次读取前后节点都需要经过计算才能得到前后节点的位置，所以会消耗更多的时间，而在 Redis 中，一个指针是占了 8 个字节，但是大部

分情况下，如果直接存储长度是达不到 8 个字节的，所以采用存储长度的设计方式在大部分场景下是可以节省内存空间的。

```
<zlbytes> <zltail> <zllen> <entry> <entry> ... <entry> <zlend>
```

高级数据类型

bitMap 位图，Hyperloglog，布隆过滤器，Geohash，Stream

Redis Stream

是 Redis 5.0 版本新增加的数据结构。
Redis Stream 主要用于消息队列（MQ，Message Queue），Redis 本身是有一个 Redis 发布订阅 (pub/sub) 来实现消息队列的功能，但它有个缺点就是消息无法持久化，如果出现网络断开、Redis 宕机等，消息就会被丢弃。
简单来说发布订阅 (pub/sub) 可以分发消息，但无法记录历史消息。
而 Redis Stream 提供了消息的持久化和主备复制功能，可以让任何客户端访问任何时刻的数据，并且能记住每一个客户端的访问位置，还能保证消息不丢失。
Redis Stream 的结构如下所示，它有一个消息链表，将所有加入的消息都串起来，每个消息都有一个唯一的 ID 和对应的内容：

每个 Stream 都有唯一的名称，它就是 Redis 的 key，在我们首次使用 xadd 指令追加消息时自动创建。
上图解析：

- Consumer Group

：消费组，使用 XGROUP CREATE 命令创建，一个消费组有多个消费者(Consumer)。

- last_delivered_id

：游标，每个消费组会有个游标 last_delivered_id，任意一个消费者读取了消息都会使游标 last_delivered_id 往前移动。

- pending_ids

：消费者(Consumer)的状态变量，作用是维护消费者的未确认的 id。 pending_ids 记录了当前已经被客户端读取的消息，但是还没有 ack (Acknowledge character：确认字符）。

消息队列相关命令：

- XADD

- 添加消息到末尾

- XTRIM

- 对流进行修剪，限制长度

- XDEL
- 删除消息
- XLEN
- 获取流包含的元素数量，即消息长度
- XRANGE
- 获取消息列表，会自动过滤已经删除的消息
- XREVRANGE
- 反向获取消息列表，ID 从大到小
- XREAD
- 以阻塞或非阻塞方式获取消息列表

消费者组相关命令：

- XGROUP CREATE
- 创建消费者组
- XREADGROUP GROUP
- 读取消费者组中的消息
- XACK
- 将消息标记为"已处理"
- XGROUP SETID
- 为消费者组设置新的最后递送消息ID
- XGROUP DELCONSUMER
- 删除消费者
- XGROUP DESTROY
- 删除消费者组
- XPENDING
- 显示待处理消息的相关信息
- XCLAIM
- 转移消息的归属权
- XINFO
- 查看流和消费者组的相关信息；
- XINFO GROUPS
- 打印消费者组的信息；
- XINFO STREAM
- 打印流信息

redis优缺点

优点

- 读写性能优异， Redis能读的速度是 110000 次/s，写的速度是 81000 次/s。

- 支持数据持久化，支持 AOF 和 RDB 两种持久化方式。
- 支持事务，Redis 的所有操作都是原子性的，同时 Redis 还支持对几个操作合并后的原子性执行。
- 数据结构丰富，除了支持 string 类型的 value 外还支持 hash、set、zset、list 等数据结构。
- 支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。

缺点

- 数据库 容量受到物理内存的限制，不能用作海量数据的高性能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。
- Redis 不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的 IP 才能恢复。
- 主机宕机，宕机前有部分数据未能及时同步到从机，切换 IP 后还会引入数据不一致的问题，降低了 系统的可用性。
- Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

使用缓存会出现什么问题

- 缓存雪崩问题：（同一时间内大量缓存数据失效，导致请求到数据库，数据库短期内压力巨大；可能是由于失效时间一致；给过期时间设置一个随机值）
- 缓存穿透问题：查询大量数据库不存在的数据，会绕过缓存，（1.使用**布隆过滤器**提前拦截;2.把不存在的空对象也放在缓存中，设置一个较短过期时间）
- 缓存击穿：大量的请求同时查询一个 key 时，此时这个 key 正好失效了，就会导致大量的请求都落到数据库。(保证热点数据都在缓存中，设置不过期，加锁更新，比如请求查询A，发现缓存中没有，对A这个key加锁，同时去数据库查询数据，写入缓存）

存，再返回给用户，这样后面的请求就可以从缓存中拿到数据了。）

- 缓存和数据库双写一致性问题：当更新删除时，导致数据不一致（先删缓存后操作数据库，或先操作数据库，后操作缓存------延时双删）

持久化方式

快照

Redis 快照 是最简单的 Redis 持久性模式。当满足特定条件时，它将生成数据集的时间点快照，例如，如果先前的快照是在 2 分钟前创建的，并且现在已经至少有 100 次新写入，则将创建一个新的快照。此条件可以由用户配置 Redis 实例来控制，也可以在运行时修改而无需重新启动服务器。快照作为包含整个数据集的单个 .rdb 文件生成。

可以通过SAVE或者BGSAVE来生成RDB文件

SAVE命令会阻塞redis进程，直到RDB文件生成完毕，在进程阻塞期间，redis不能处理任何命令请求，这显然是不合适的。

BGSAVE则是会fork出一个子进程，然后由子进程去负责生成RDB文件，父进程还可以继续处理命令请求，不会阻塞进程。

AOF

AOF(Append Only File - 仅追加文件) 它的工作方式非常简单：每次执行 修改内存 中数据集的写操作时，都会 记录 该操作。假设 AOF 日志记录了自 Redis 实例创建以来 所有的修改性指令序列，那么就可以通过对一个空的 Redis 实例 顺序执行所有的指令，也就是 「重放」，来恢复 Redis 当前实例的内存数据结构的状态。

1. 当AOF持久化处于激活状态，服务器执行完写命令之后，写命令将会被追加append到aof_buf缓冲区的末尾
2. 在服务器每结束一个事件循环之前，将会调用flushAppendOnlyFile函数决定是否要将aof_buf的内容保存到AOF文件中，可以通过配置appendfsync来决定。

always aof_buf内容写入并同步到AOF文件

everysec 将aof_buf中内容写入到AOF文件，如果上次同步AOF文件时间距离现在超过1秒，则再次对AOF文件进行同步

no 将aof_buf内容写入AOF文件，但是并不对AOF文件进行同步，同步时间由操作系统决定

如果不设置，默认选项将会是everysec，因为always来说虽然最安全（只会丢失一次事件循环的写命令），但是性能较差，而everysec模式只不过会可能丢失1秒钟的数据，而no模式的效率和everysec相仿，但是会丢失上次同步AOF文件之后的所有写命令数据

Redis 4.0 的混合持久化

重启 Redis 时，我们很少使用 rdb 来恢复内存状态，因为会丢失大量数据。我们通常使用 AOF 日志重放，但是重放 AOF 日志性能相对 rdb 来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。

Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——混合持久化。将 rdb 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是自持久化开始到持久化结束的这段时间发生的增量 AOF 日志，通常这部分 AOF 日志很小

redis命令

string

单个值最大是1G，可以包含任何数据，包括图片、序列化的对象

hash

set

zset

list

发布订阅

事务

常见的部署方式

单机版
很少使用。存在的问题：1、内存容量有限 2、处理能力有限 3、无法高可用。

主从模式
master 节点挂掉后，需要手动指定新的 master，可用性不高，基本不用。
1. slave发送sync命令到master
2. master收到sync之后，执行bgsave，生成RDB全量文件
3. master把slave的写命令记录到缓存
4. bgsave执行完毕之后，发送RDB文件到slave，slave执行
5. master发送缓存中的写命令到slave，slave执行

哨兵模式
master 节点挂掉后，哨兵进程会主动选举新的 master，可用性高，但是每个节点存储的数据是一样的，浪费内存空间。数据量不是很多，集群规模不是很大，需要自动容错容灾的时候使用。具备自动故障转移、集群监控、消息通知
1. 初始化sentinel，将普通的redis代码替换成sentinel专用代码
2. 初始化masters字典和服务器信息，服务器信息主要保存ip:port，并记录实例的地址和ID
3. 创建和master的两个连接，命令连接和订阅连接，并且订阅sentinel:hello频道
4. 每隔10秒向master发送info命令，获取master和它下面所有slave的当前信息
5. 当发现master有新的slave之后，sentinel和新的slave同样建立两个连接，同时每个10秒发送info命令，更新master信息

6. sentinel每隔1秒向所有服务器发送ping命令，如果某台服务器在配置的响应时间内连续返回无效回复，将会被标记为下线状态
7. 选举出领头sentinel，领头sentinel需要半数以上的sentinel同意
8. 领头sentinel从已下线的的master所有slave中挑选一个，将其转换为master
9. 让所有的slave改为从新的master复制数据
10. 将原来的master设置为新的master的从服务器，当原来master重新回复连接时，就变成了新master的从服务器

Redis cluster（集群）

主要是针对海量数据+高并发+高可用的场景，如果是海量数据，如果你的数据量很大，那么建议就用Redis cluster，所有主节点的容量总和就是Redis cluster可缓存的数据容量。

1. 节点A收到客户端的cluster meet命令
2. A根据收到的IP地址和端口号，向B发送一条meet消息
3. 节点B收到meet消息返回pong
4. A知道B收到了meet消息，返回一条ping消息，握手成功
5. 最后，节点A将会通过gossip协议把节点B的信息传播给集群中的其他节点，其他节点也将和B进行握手

Redis hash槽存储数据原理

RedisCluster采用此分区，所有的键根据哈希函数(CRC16[key]%16384)映射到0 - 16383槽内，共16384个槽位，每个节点维护部分槽及槽所映射的键值数据.根据主节点的个数,均衡划分区间.

算法:哈希函数: Hash()=CRC16[key]%16384

Redis事务机制

redis通过MULTI、EXEC、WATCH等命令来实现事务机制，事务执行过程将一系列多个命令按照顺序一次性执行，并且在执行期间，事务不会被中断，也不会去执行客户端的其他请求，直到所有命令执行完毕。事务的执行过程如下：

1. 服务端收到客户端请求，事务以MULTI开始
2. 如果客户端正处于事务状态，则会把事务放入队列同时返回给客户端QUEUED，反之则直接执行这个命令
3. 当收到客户端EXEC命令时，WATCH命令监视整个事务中的key是否有被修改，如果有则返回空回复到客户端表示失败，否则redis会遍历整个事务队列，执行队列中保存的所有命令，最后返回结果给客户端

WATCH的机制本身是一个CAS的机制，被监视的key会被保存到一个链表中，如果某个key被修改，那么REDIS_DIRTY_CAS标志将会被打开，这时服务器会拒绝执行事务

redis锁

SETNX（set if not exists）锁机制

1、如果不存在则设置，拿到锁，执行业务，最后删除锁（问题：如果业务中途退出了，没有删除锁，锁就一直在）

2、如果不存在则设置，并添加过期时间（问题：过期时间不好确定，可能业务时间超过了过期时间（锁续命），会被别的操作拿到锁，当前业务执行完，可能删的是别的操作加的锁（加唯一值，删除的时候去判断））

3、Redisson

支持多种连接方式，主从、哨兵、集群

redisson中有一个watchdog看门狗的概念，翻译过来就是看门狗，它会在你获取锁之后，每隔10秒（设置时间的1/3）帮你把key的超时时间设为30s

```
String key = "product_id";
RLock lock = redisson.getLock(key);
try {
    lock.lock();
    System.out.println("业务操作");
}finally {
    lock.unlock();
}
```

采用Redisson分布式锁的问题分析

主从同步问题

当主Redis加锁了，开始执行线程，若还未将锁通过异步同步的方式同步到从Redis节点，主节点就挂了，此时会把某一台从节点作为新的主节点，此时别的线程就可以加锁了，这样就出错了，怎么办？

1. 采用zookeeper代替Redis

由于zk集群的特点，其支持的是CP。而Redis集群支持的则是AP。

zk是如何实现CP的?

使用 zk 实现 Master 选举的原理是，集群中所有主机都向 zk 中创建相同路径下的某持久节点注册子节点列表变更 watcher 监听，并在该节点下持久相同名称的临时节点，谁创建成功谁就是 Master。

当 Master 宕机，该临时节点消失，此时会触发其他主机 watcher 回调的执行。watcher回调会重新抢注该节点下的临时节点，谁注册成功谁就是 Master。即可以实现 Master 宕机后

的自动重新选举。

区别

使用 zk 实现的分布式锁是 CP 的分布式锁。因为 zk 是 CP 的。在某客户端向 zk 集群中的某节点写入数据后，会等待超过半数的其它节点完成同步后，才会响应该客户端。

使用 Redis 实现的分布式锁是 AP 的分布式锁。因为 Redis 是 AP 的。在某客户端向 Redis集群中的某节点写入数据后，会立即响应该客户端，之后在 Redis 集群中会以异步的方式来同步数据。

对于 AP 的分布式锁，需要注意可能会出现的问题：一个客户端 a 在 Redis 集群的某节点 A 写入数据后，另一个节点 B 在还未同步时，另一个客户端 b 从 B 节点读取数据，没有发现 a 写入的数据。此时可能会出现问题。所以，如果某共享资源要求必须严格按照锁机制进行访问，那么就使用 zk 实现的 CP 锁。

2. 采用RedLock

```
@RequestMapping("/deduct_stock_redlock")
public String deductStockRedlock() {
    String lockKey = "product_001";
    //TODO 这里需要自己实例化不同redis实例的redisson客户端连接，这里只是伪代码用一个redisson客户端简化了
    RLock rLock1 = redisson.getLock(lockKey);
    RLock rLock2 = redisson.getLock(lockKey);
    RLock rLock3 = redisson.getLock(lockKey);

    // 向3个redis实例尝试加锁
    RedissonRedLock redLock = new RedissonRedLock(rLock1, rLock2, rLock3);
    boolean isLock;
    try {
        // 500ms拿不到锁，就认为获取锁失败。10000ms即10s是锁失效时间。
        isLock = redLock.tryLock(500, 10000, TimeUnit.MILLISECONDS);
        System.out.println("isLock = " + isLock);
        if (isLock) {
            //业务逻辑处理
            ...
        }
    } catch (Exception e) {

    } finally {
        // 无论如何，最后都要解锁
        redLock.unlock();
    }
```

```
26 }
```

提高并发：分段锁

由于Redisson实际上就是将并行的请求，转化为串行请求。这样就降低了并发的响应速度，为了解决这一问题，可以将锁进行分段处理：例如秒杀商品001，原本存在1000个商品，可以将其分为20段，为每段分配50个商品...

实现消息队列的几种方式

使用 List 类型实现

主要是通过 lpush、rpop 存入和读取实现消息队列的，如下图所示：

使用 ZSet 类型实现

它是利用 zadd 和 zrangebyscore 来实现存入和读取消息的

优点：同样具备持久化的功能

缺点：List 存在的问题它也同样存在，不但如此，使用 ZSet 还不能存储相同元素的值。因为它是有序集合，有序集合的存储元素值是不能重复的，但分值可以重复，也就是说当消息值重复时，只能存储一条信息在 ZSet 中。

使用发布订阅者模式实现消息队列

使用发布和订阅的类型，我们可以实现主题订阅的功能，也就是 Pattern Subscribe 的功能。因此我们可以使用一个消费者“queue*”**来订阅所有以“queue**”开头的消息队列，如下图所示：

优点：可以按照主题订阅方式

缺点： a、无法持久化保存消息，如果 Redis 服务器宕机或重启，那么所有的消息将会丢失； b、发布订阅模式是“发后既忘”的工作模式，如果有订阅者离线重连之后就不能消费之前的历史消息； c、不支持消费者确认机制，稳定性不能得到保证，例如当消费者获取到消息之后，还没来得及执行就宕机了。因为没有消费者确认机制，Redis 就会误以为消费者已经执行了，因此就不会重复发送未被正常消费的消息了，这样整体的 Redis 稳定性就被没有办法得到保障了。

使用 Stream 实现消息队列

使用 Stream 的 xadd 和 xrange 来实现消息的存入和读取了，并且 Stream 提供了 xack 手动确认消息消费的命令，用它我们就可以实现消费者确认的功能了，使用命令如下：

```
127.0.0.1:6379> xack mq group1 1580959593553-0
(integer) 1
```

消费确认增加了消息的可靠性，一般在业务处理完成之后，需要执行 ack 确认消息已经被消费完成，整个流程的执行如下图所示

其中“Group”为群组，消费者也就是接收者需要订阅到群组才能正常获取到消息。

大key

BigKey指的是redis中一些key value值很大,这些key在序列化与反序列化过程中花费的时间很大! 操作bigkey的通常比较耗时，也就意味着阻塞Redis可能性越大!占用的流量同时也会变得很大!
大白话就是bigkey实际指一个key对应的value很大,占用的空间很大!
string长度大于10K，list长度大于10240认为是big bigkeys

查找、删除

查询bigkeys(可以找到某个实例5种数据类型(String、hash、list、set、zset)的最大key)

```
redis-cli -h 127.0.0.1 -p 6379 --bigkeys
```

Redis 4.0之后的大key的发现与删除方法

Redis 4.0引入了 `memory usage` 命令和 `lazyfree` 机制，不管是对大key的发现，还是解决大key删除或者过期造成的阻塞问题都有明显的提升。
在某些业务场景下，Redis大key的问题是难以避免的，但是，memory usage命令和lazyfree机制分别提供了内存维度的抽样算法和异步删除优化功能，这些特性有助于我们在实际业务中更好的预防大key的产生和解决大key造成的阻塞。

大key如何优化

拆分

如果对象是整存整取
将对象拆分后才能多个小key-value,get不同的key或者批量获取
stringRedisTemplate.opsForValue() .multiGet(keyList)
如果对象是部分更新获取数据
可以分拆成几个key-value,也可以存储在hash中,部分更新部分存取!
如果是hash ,set,zset ,list 等元素
固定一个桶的数量，比如1000，每次存取的时候，先在本地计算field的hash值，模除1000，确定该field落在哪个key上。
newHashKey = hashKey + (hash(field) % 1000);
hset(newHashKey, field, value);
hget(newHashKey, field)
set, zset, list 也可以类似上述做法!

本地缓存

减少访问redis次数，降低危害减少访问redis次数，降低危害! 当然本地开销也会变大!

热Key

热key问题就是，突然有几十万的请求去访问redis上的某个特定key。那么，这样会造成流量过于集中，达到物理网卡上限，从而导致这台redis的服务器宕机。

那接下来这个key的请求，就会直接怼到你的数据库上，导致你的服务不可用。

热Key问题产生的原因

1、用户消费的数据远大于生产的数据（热卖商品、热点新闻、热点评论、明星直播）。

在日常工作生活中一些突发的的事件，例如：双十一期间某些热门商品的降价促销，当这其中的某一件商品被数万次点击浏览或者购买时，会形成一个较大的需求量，这种情况下就会造成热点问题。

同理，被大量刊发、浏览的热点新闻、热点评论、明星直播等，这些典型的读多写少的场景也会产生热点问题。

2、请求分片集中，超过单 Server 的性能极限。

在服务端读数据进行访问时，往往会对数据进行分片切分，此过程中会在某一主机 Server 上对相应的 Key 进行访问，当访问超过 Server 极限时，就会导致热点 Key 问题的产生。

热Key问题的危害

1、流量集中，达到物理网卡上限。

2、请求过多，缓存分片服务被打垮。

3、DB 击穿，引起业务雪崩。

怎么发现热key

方法一:**凭借业务经验，进行预估哪些是热key**

其实这个方法还是挺有可行性的。比如某商品在做秒杀，那这个商品的key就可以判断出是热key。缺点很明显，并非所有业务都能预估出哪些key是热key。

方法二:**在客户端进行收集**

这个方式就是在操作redis之前，加入一行代码进行数据统计。那么这个数据统计的方式有很多种，也可以是给外部的通讯系统发送一个通知信息。缺点就是对客户端代码造成入侵。

方法三:**在Proxy层做收集**

有些集群架构是下面这样的，Proxy可以是Twemproxy，是统一的入口。可以在Proxy层做收集上报，但是缺点很明显，并非所有的redis集群架构都有proxy。

方法四:**用redis自带命令定时扫描**

(1)**monitor**命令，该命令可以实时抓取出redis服务器接收到的命令，然后写代码统计出热key是啥。当然，也有现成的分析工具可以给你使用，比如redis-faina。但是该命令在高并发的条件下，有内存增暴增的隐患，还会降低redis的性能。

(2)**hotkeys**参数，redis 4.0.3提供了redis-cli的热点key发现功能，执行redis-cli时加上–hotkeys选项即可。但是该参数在执行的时候，如果key比较多，执行起来比较慢。

方法五:自己抓包评估

Redis客户端使用TCP协议与服务端进行交互，通信协议采用的是RESP。自己写程序监听端口，按照RESP协议规则解析数据，进行分析。
该方案无需侵入现有的 SDK 或者 Proxy 中间件，开发维护成本可控，但也存在缺点的，具体是热 key 节点的网络流量和系统负载已经比较高了，抓包可能会情况进一步恶化。
如何解决
(1)利用二级缓存
比如利用ehcache，或者一个HashMap都可以。在你发现热key以后，把热key加载到系统的JVM中。针对这种热key请求，会直接从jvm中取，而不会走到redis层。
假设此时有十万个针对同一个key的请求过来,如果没有本地缓存，这十万个请求就直接怼到同一台redis上了。
现在假设，你的应用层有50台机器，OK，你也有jvm缓存了。这十万个请求平均分散开来，每个机器有2000个请求，会从JVM中取到value值，然后返回数据。避免了十万个请求怼到同一台redis上的情形。
(2)备份热key
这个方案也很简单。不要让key走到同一台redis上不就行了。我们把这个key，在多个redis上都存一份不就好了。接下来，有热key请求进来的时候，我们就在有备份的redis上随机选取一台，进行访问取值，返回数据。
假设redis的集群数量为N，步骤如下图所示

有办法在项目运行过程中，自动发现热key，然后程序自动处理么?
(1)监控热key
(2)通知系统做处理

过期删除策略

redisDb中有两个dict对象，dict内部实现的是哈希表的结构。两个dict对象的名字一个叫dict，一个叫expires。
dict用于存放实际数据、expires用于存放过期时间数据。
当往redisDb中的dict中加入key-value数据的的时候，并且为数据设置了过期时间的时候，会将对应的key和过期时间存放到expires中，便于后期查找过期时间。

1、定时删除

当放入数据后，设置一个定时器，当定时器读秒完毕后，将对应的数据从dict中删除。
优点： 内存友好，数据一旦过期就会被删除缺点： CPU不友好，定时器耗费CPU资源，并且频繁的执行清理操作也会耗费CPU资源。
用时间换空间

2、惰性删除

当数据过期的时候，不做任何操作。当访问数据的时候，查看数据是否过期，如果过期返回null，并且将数据从内存中清除。如果没过期，就直接返回数据。

优点： CPU友好，数据等到过期并且被访问的时候，才会删除。缺点： 内存不友好，会占用大量内存。

用空间换时间

3、定期删除

定期删除是定时删除和惰性删除的折中方案。每隔一段时间对redisServer中的所有redisDb的expires依次进行随机抽取检查。（每次随机删除哪些key呢？可以提下LRU算法（Least Recently Used 最近最少使用））

redis中有一个server.hz定义了每秒钟执行定期删除的次数，每次执行的时间为250ms/server.hz。

redis中会维护一个current_db变量来标志当前检查的数据库。current_db++，当超过数据库的数量的时候，会重新从0开始。

定期检查就是执行一个循环，循环中的每轮操作会从current_db对应的数据库中随机依次取出w个key，查看其是否过期。如果过期就将其删除， 并且记录删除的key的个数。如果过期的key个数大于w * 25%，就会继续检查当前数据库，当过期的key小于w * 25%，会继续检查下一个数据库。

当执行时间超过规定的最大执行时间的时候，会退出检查。

一次检查中可以检查多个数据库，但是最多检查数量是redisServer中的数据库个数，也就是最多只能从当前位置检查一圈。

优点： 通过控制定时时间来动态的调整CPU和内存之间的状态，十分灵活。

缺点： 定期删除的定时时间十分重要，如果时间过短，就会对CPU造成很大压力。如果时间过长，就会造成过期数据挤压内存。

Redis采用的是 **惰性删除 + 定期删除** 的策略。

对过期键的处理

在主从复制模式下，从服务器的过期键删除动作由主服务器控制：

- 主服务器在删除一个过期键后，会显式地向所有从服务器发送一个DEL命令，告知从服务器删除这个过期键。
- 从服务器在执行客户端发送的读命令时，即使发现该键已过期也不会删除该键，照常返回该键的值。
- 从服务器只有接收到主服务器发送的DEL命令后，才会删除过期键。

通过由主服务器来控制从服务器统一地删除过期键，可以保证主从服务器数据的一致性，也正是因为这个原因，当一个过期键仍然存在于主服务器的数据库时，这个过期键在从服务器里的复制品也会继续存在

RDB

1、生成RDB文件在执行SAVE命令或者BGSAVE命令创建一个新的RDB文件时，程序会对数据库中的键进行检查，已过期的键不会被保存到新创建的RDB文件中

2、载入RDB文件在启动Redis服务器时，如果服务器开启了RDB功能，那么服务器将对RDB文件进行载入：

如果服务器以主服务器模式运行，那么在载入RDB文件时，程序会对文件中保存的键进行检查，未过期的键会被载入到数据库中，而过期键则会被忽略，所以过期键对载入RDB文件的主服务器不会造成影响如果服务器以从服务器模式运行，那么在载入RDB文件时，文件中保存的所有键，不论是否过期，都会被载入到数据库中。不过，因为主从服务器在进行数据同步的时候，从服务器的数据库就会被清空，所以一般来说，过期键在载入RDB文件的从服务器也不会造成影响

AOF

1、AOF文件写入当服务器以AOF持久化模式运行时，如果数据库中的某个键已经过期，但它还没有被惰性删除或者定期删除，那么AOF文件不会因为这个过期键而产生任何影响

当过期键被惰性删除或者定期删除之后，程序会向AOF文件追加一个DEL命令，来显式地记录该键已被删除

2、AOF重写和生成RDB文件时类似，在执行AOF重写的过程中，程序会对数据库中的键进行检查，已过期的键不会被保存到重写后的AOF文件中

memcached与redis区别

1.Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例片、视频等等

2.Redis 不仅仅支持简单的k/v 类型的数据，同时还提供list，set，zset，hash等数据结构的存储。而memcache 只支持简单数据类型，需要客户端自己处理复杂对象。

3.Redis只使用单核，而Memcached可以使用多核。所以平均每一个核上Redis在存储小数据时比Memcached性能更高。而在100k以上的数据中，Memcached性能要高于Redis，虽然Redis在存储大数据的性能上进行优化，但是比起Memcached，还是稍有逊色。

redis是单线程还是多线程

> Redis4.0之前是单线程运行的；Redis4.0后开始支持多线程。Redis4.0之前使用单线程的原因：1、单线程模式方便开发和调试；2、Redis内部使用了基于epoll的多路复用；3、Redis主要的性能瓶颈是内存或网络带宽。

单线程并不代表效率低,原因是Redis是基于内存的，它的瓶颈在于机器的内存、网络带宽，而不是CPU，在CPU还没达到瓶颈时机器内存可能就满了、或者带宽达到瓶颈了。因此CPU不是主要原因，那么自然就采用单线程了，况且使用多线程比较麻烦。

但是在Redis4.0的时候，已经开始支持多线程了，比如**后台删除等功能**。意思就是我们可以使用异步的方式对Redis中的数据进行删除操作，例如：

unlink key：和del key类似，删除指定的key，若key不存在则key被跳过。但是del会产生阻塞，而unlink命令会在另一个线程中回收内存，即它是非阻塞的

flushdb async：删除当前数据库的所有数据

flushall async：删除所有库中的数据

这样处理的好处是不会使Redis的主线程卡顿，会把这些操作交给后台线程来执行。

【通常情况下使用del指令可以很快的删除数据，但是当被删除的key是一个非常大的对象时，例如：删除的时包含成千上万个元素的hash集合时，那么del指令就会造成Redis主线程卡顿，因此使用惰性删除可以有效避免Redis卡顿问题。】
在Redis6.0中新增了多线程的功能来**提高IO的读写性能**，它的主要实现思路是将主线程的IO读写任务拆分给一组独立的线程去执行，这样就可以使用多个socket的读写并行化了，但Redis的命令依旧是主线程串行执行的。这个I/O threads 指的是网络I/O处理方面使用了多线程，如网络数据的读写和协议解析等，这是因为读写网络的read/write在Redis执行期间占用了大部分CPU时间，如果把这部分模块使用多线程方式实现会对性能带来极大地提升。但是Redis执行命令的核心模块还是单线程的。

但是注意：Redis6.0是默认禁用多线程的，但可以通过配置文件redis.conf中的io-threads-do-reads等于 true 来开启。但是还不够，除此之外我们还需要设置线程的数量才能正确地开启多线程的功能，同样是修改Redis的配置，例如设置 io-threads 4，表示开启4个线程。
【关于线程数的设置，官方的建议是如果为4核CPU，那么设置线程数为2或3；如果为8核CPU，那么设置线程数为6.总之线程数一定要小于机器的CPU核数，线程数并不是越大越好。】

淘汰机制

假设redis每次定期随机查询key的时候没有删掉，这些key也没有做查询的话，就会导致这些key一直保
存在redis里面无法被删除，这时候就会走到redis的内存淘汰机制。
1. volatile-lru：从已设置过期时间的key中，移出最近最少使用的key进行淘汰
2. volatile-ttl：从已设置过期时间的key中，移出将要过期的key
3. volatile-random：从已设置过期时间的key中随机选择key淘汰
4. allkeys-lru：从key中选择最近最少使用的进行淘汰
5. allkeys-random：从key中随机选择key进行淘汰
6. noeviction：当内存达到阈值的时候，新写入操作报错

RabbitMQ

消息队列的三个要求

- 消息有序执行（**有序性**）；
- 由于网络阻塞等原因，消息不会被重复执行（**幂等性**）；
- 宕机后，消息不会丢失（**可靠性**）。

消息队列的优点

1. **解耦**，系统A在代码中直接调用系统B和系统C的代码，如果将来D系统接入，系统A还需要修改代码，过于麻烦！
2. **异步**，将消息写入消息队列，非必要的业务逻辑以异步的方式运行，加快响应速度
3. **削峰**，并发量大的时候，所有的请求直接怼到数据库，造成数据库连接异常

消息队列会有什么缺点

1.系统可用性降低:你想啊，本来其他系统只要运行好好的，那你的系统就是正常的。现在你非要加个消息队列进去，那消息队列挂了，你的系统不是呵呵了。因此，系统可用性降低
2.系统复杂性增加:要多考虑很多方面的问题，比如一致性问题、如何保证消息不被重复消费，如何保证保证消息可靠传输。因此，需要考虑的东西更多，系统复杂性增大。

Kafka、ActiveMQ、RabbitMQ、RocketMQ

消息中间件的组成

- Broker：消息服务器，作为server提供消息核心服务
- Producer：消息生产者，业务的发起方，负责生产消息传输给broker
- Consumer：消息消费者，业务的处理方，负责从broker获取消息并进行业务逻辑处理
- Topic：主题，发布订阅模式下的消息统一汇集地，不同生产者向topic发送消息，由MQ服务器分发到不同的订阅者，实现消息的 广播
- Queue：队列，PTP模式下，特定生产者向特定queue发送消息，消费者订阅特定的queue完成指定消息的接收
- Message：消息体，根据不同通信协议定义的固定格式进行编码的数据包，来封装业务数据，实现消息的传输

消息中间件模式

点对点 PTP

消息生产者生产消息发送到queue中，然后消息消费者从queue中取出并且消费消息。
消息被消费以后，queue中不再存储，所以消息消费者不可能消费到已经被消费的消息。 Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。

发布/订阅

Pub/Sub发布订阅（广播）：使用topic作为通信载体
消息生产者（发布）将消息发布到topic中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到topic的消息会被所有订阅者消费。
queue实现了负载均衡，将producer生产的消息发送到消息队列中，由多个消费者消费。但一个消息只能被一个消费者接受，当没有消费者可用时，这个消息会被保存直到有一个可用的消费者。 **topic**实现了发布和订阅，当你发布一个消息，所有订阅这个topic的服务都能得到这个消息，所以从1到N个订阅者都能得到一个消息的拷贝。

常用协议

AMQP协议

AMQP即Advanced Message Queuing Protocol,一个提供统一消息服务的应用层标准高级消息队列协议,是应用层协议的一个开放标准,为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同开发语言等条件的限制。 优点：可靠、通用

MQTT协议

MQTT（Message Queuing Telemetry Transport，消息队列遥测传输）是IBM开发的一个即时通讯协议，有可能成为物联网的重要组成部分。该协议支持所有平台，几乎可以把所有联网物品和外部连接起来，被用来当做传感器和致动器（比如通过Twitter让房屋联网）的通信协议。 优点：格式简洁、占用带宽小、移动端通信、PUSH、嵌入式系统

STOMP协议

STOMP（Streaming Text Orientated Message Protocol）是流文本定向消息协议，是一种为MOM(Message Oriented Middleware，面向消息的中间件)设计的简单文本协议。STOMP提供一个可互操作的连接格式，允许客户端与任意STOMP消息代理（Broker）进行交互。 优点：命令模式（非topic\queue模式）

XMPP协议

XMPP（可扩展消息处理现场协议，Extensible Messaging and Presence Protocol）是基于可扩展标记语言（XML）的协议，多用于即时消息（IM）以及在线现场探测。适用于服务器之间的准即时操作。核心是基于XML流传输，这个协议可能最终允许因特网用户向因特网上的其他任何人发送即时消息，即使其操作系统和浏览器不同。 优点：通用公开、兼容性强、可扩展、安全性高，但XML编码格式占用带宽大

其他基于TCP/IP自定义的协议

有些特殊框架（如：redis、kafka、zeroMq等）根据自身需要未严格遵循MQ规范，而是基于TCP\IP自行封装了一套协议，通过网络socket接口进行传输，实现了MQ的功能。

JMS

即Java消息服务（Java Message Service）应用程序接口，是一个Java平台中关于面向消息中间件（MOM）的API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。Java消息服务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。

RabbitMQ 中的 broker 是指什么？cluster 又是指什么

broker 是指一个或多个 erlang node 的逻辑分组，且 node 上运行着 RabbitMQ 应用程序。cluster 是在 broker 的基础之上，增加了 node 之间共享元数据的约束。

3.RabbitMQ 概念里的 channel、exchange 和 queue 是逻辑概念，还是对应着进程实体？分别起什么作用?

queue 具有自己的 erlang 进程；exchange 内部实现为保存 binding 关系的查找表；channel 是实际进行路由工作的实体，即负责按照 routing_key 将 message 投递给 queue 。由 AMQP 协议描述

可知，channel 是真实 TCP 连接之上的虚拟连接，所有 AMQP 命令都是通过 channel 发送的，且每一个 channel 有唯一的 ID。一个 channel 只能被单独一个操作系统线程使用，故投递到特定 channel 上的 message 是有顺序的。但一个操作系统线程上允许使用多个 channel 。

vhost 是什么？起什么作用

vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

消息基于什么传输?

由于TCP连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ使用信道的方式来传输数据。信道是建立在真实的TCP连接内的虚拟连接，且每条TCP连接上的信道数量没有限制。

消息如何分发

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。

消息怎么路由

从概念上来说，消息路由必须有三部分：交换器、路由、绑定。生产者把消息发布到交换器上；绑定决定了消息如何从路由器路由到特定的队列；消息最终到达队列，并被消费者接收。
消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。通过队列路由键，可以把队列绑定到交换器上。消息到达交换器后，RabbitMQ会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）。如果能够匹配到队列，则消息会投递到相应队列中；如果不能匹配到任何队列，消息将进入 “黑洞” 。常用的交换器主要分为四种：
direct：如果路由键完全匹配，消息就被投递到相应的队列
fanout：如果交换器收到消息，将会广播到所有绑定的队列上
topic：可以使来自不同源头的消息能够到达同一个队列。 使用topic交换器时，可以使用通配符，比如： “*” 匹配特定位置的任意文本， “.” 把路由键分为了几部分， “#” 匹配所有规则等。特别注意：发往topic交换器的消息不能随意的设置选择键（routing_key），必须是由"."隔开的一系列的标识符组成。
headers:分发消息不依赖路由键，使用发送消息basicProperties对象中的headers来匹配的，将消息中的headers与该交换机中所有Binding中的参数进行匹配。
Exchange 属性说明
Virtual host:属于哪个Virtual host。（如果有多个Virtual host的有此属性，一般默认的Virtual host是"/"，Virtual host可以做最小粒度的权限控制。）

Name：名字，同一个Virtual host里面的Name不能重复。
Durability： 是否持久化 （Durable：持久化，Transient：不持久化）。
Auto delete：当最后一个绑定（队列或者exchange）被unbind之后，该exchange自动被删除。
Internal： 是否是内部专用exchange，是的话，就意味着我们不能往该exchange里面发消息。
Arguments： 参数，是AMQP协议留给AMQP实现做扩展使用的。alternate_exchange配置的时候，exchange根据路由路由不到对应的队列的时候，这时候消息被路由到指定的alternate_exchange的value值配置的exchange上。

在单 node 系统和多 node 构成的 cluster 系统中声明 queue、exchange ，以及进行 binding 会有什么不同

当你在单 node 上声明 queue 时，只要该 node 上相关元数据进行了变更，你就会得到 Queue.Declare-ok 回应；而在 cluster 上声明 queue ，则要求 cluster 上的全部 node 都要进行元数据成功更新，才会得到 Queue.Declare-ok 回应。另外，若 node 类型为 RAM node 则变更的数据仅保存在内存中，若类型为 disk node 则还要变更保存在磁盘上的数据。

死信队列&死信交换器

DLX 全称（Dead-Letter-Exchange）,称之为死信交换器，当消息变成一个死信之后，如果这个消息所在的队列存在x-dead-letter-exchange参数，那么它会被发送到x-dead-letter-exchange对应值的交换器上，这个交换器就称之为死信交换器，与这个死信交换器绑定的队列就是死信队列。

如何确保消息正确地发送至RabbitMQ

RabbitMQ使用发送方确认模式，确保消息正确地发送到RabbitMQ。发送方确认模式：将信道设置成confirm模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的ID。一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一ID）。如果RabbitMQ发生内部错误从而导致消息丢失，会发送一条nack（not acknowledged，未确认）消息。发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

如何确保消息接收方消费了消息

接收方消息确认机制：消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ才能安全地把消息从队列中删除。这里并没有用到超时机制，RabbitMQ仅通过Consumer的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ给了Consumer足够长的时间来处理消息。
下面罗列几种特殊情况：

如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要根据bizId去重）
如果消费者接收到消息却没有确认消息，连接也未断开，则RabbitMQ认为该消费者繁忙，将不会给该消费者分发更多的消息。

如何避免消息重复投递或重复消费

在消息生产时，MQ内部针对每条生产者发送的消息生成一个inner-msg-id，作为去重和幂等的依据（消息投递失败并重传），避免重复的消息进入队列；在消息消费时，要求消息体中必须要有一个bizId（对于同一业务全局唯一，如支付ID、订单ID、帖子ID等）作为去重和幂等的依据，避免同一条消息被重复消费。
这个问题针对业务场景来答分以下几点：
1.比如，你拿到这个消息做数据库的insert操作。那就容易了，给这个消息做一个唯一主键，那么就算出现重复消费的情况，就会导致主键冲突，避免数据库出现脏数据。
2.再比如，你拿到这个消息做redis的set的操作，那就容易了，不用解决，因为你无论set几次结果都是一样的，set操作本来就算幂等操作。
3.如果上面两种情况还不行，上大招。准备一个第三方介质,来做消费记录。以redis为例，给消息分配一个全局id，只要消费过该消息，将<id,message>以K-V形式写入redis。那消费者开始消费前，先去redis中查询有没消费记录即可。

如何解决丢数据的问题

1.生产者丢数据

生产者的消息没有投递到MQ中怎么办？从生产者弄丢数据这个角度来看，RabbitMQ提供**transaction**和**confirm**模式来确保生产者不丢消息。
transaction机制就是说，发送消息前，开启事物(channel.txSelect())，然后发送消息，如果发送过程中出现什么异常，事物就会回滚(channel.txRollback())，如果发送成功则提交事物(channel.txCommit())。
然而缺点就是吞吐量下降了。因此，按照博主的经验，生产上用confirm模式的居多。一旦channel进入confirm模式，所有在该信道上面发布的消息都将会被指派一个唯一的ID(从1开始)，一旦消息被投递到所有匹配的队列之后，rabbitMQ就会发送一个Ack给生产者(包含消息的唯一ID)，这就使得生产者知道消息已经正确到达目的队列了.如果rabiitMQ没能处理该消息，则会发送一个Nack消息给你，你可以进行重试操作。
2.消息队列丢数据
处理消息队列丢数据的情况，一般是开·启持久化磁盘的配置。这个持久化配置可以和confirm机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个Ack信号。这样，如果消息持久化磁盘之前，rabbitMQ阵亡了，那么生产者收不到Ack信号，生产者会自动重发。
那么如何持久化呢，这里顺便说一下吧，其实也很容易，就下面两步
①、将queue的持久化标识durable设置为true,则代表是一个持久的队列

②、发送消息的时候将deliveryMode=2
这样设置以后，rabbitMQ就算挂了，重启后也能恢复数据。在消息还没有持久化到硬盘时，可能服务已经死掉，这种情况可以通过引入mirrored-queue即镜像队列，但也不能保证消息百分百不丢失（整个集群都挂掉）

3.消费者丢数据
启用手动确认模式可以解决这个问题
①自动确认模式，消费者挂掉，待ack的消息回归到队列中。消费者抛出异常，消息会不断的被重发，直到处理成功。不会丢失消息，即便服务挂掉，没有处理完成的消息会重回队列，但是异常会让消息不断重试。
②手动确认模式，如果消费者来不及处理就死掉时，没有响应ack时会重复发送一条信息给其他消费者；如果监听程序处理异常了，且未对异常进行捕获，会一直重复接收消息，然后一直抛异常；如果对异常进行了捕获，但是没有在finally里ack，也会一直重复发送消息(重试机制)。
③不确认模式，acknowledge="none" 不使用确认机制，只要消息发送完成会立即在队列移除，无论客户端异常还是断开，只要发送完就移除，不会重发。

RabbitMQ 相关名词

ConnectionFactory（连接管理器）：应用程序与Rabbit之间建立连接的管理器，程序代码中使用。
Connection：与RabbitMQ服务器的连接
Exchange：交换机
Channel（信道）：消息推送使用的通道。与Exchange的连接
Queue（队列）：用于存储生产者的消息。
Routing Key：路由键，生产者发送消息的时候可以带上路由键发送给交换器
Binding Key：绑定键，交换机与队列之间的绑定关系
Broker：服务器端
Binding：队列和交互机的根据路由键映射的关系

Master和Slave之间是怎么同步数据

1. 在broker收到消息后，会被标记为uncommitted状态
2. 然后会把消息发送给所有的slave
3. slave在收到消息之后返回ack响应给master
4. master在收到超过半数的ack之后，把消息标记为committed
5. 发送committed消息给所有slave，slave也修改状态为committed

ElasticSearch

elasticsearch 是如何实现 master 选举的

描述一下 Elasticsearch 索引文档的过程

描述一下 Elasticsearch 搜索的过程

搜索拆解为“query then fetch” 两个阶段。

query 阶段的目的：定位到位置，但不取。

步骤拆解如下：

（1）假设一个索引数据有 5 主+1 副本 共 10 分片，一次请求会命中（主或者副本分片中）的一个。

（2）每个分片在本地进行查询，结果返回到本地有序的优先队列中。

（3）第 2）步骤的结果发送到协调节点，协调节点产生一个全局的排序列表。

fetch 阶段的目的：取数据。

路由节点获取所有文档，返回给客户端。

Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法

（1）关闭缓存 swap;

（2）堆内存设置为：Min（节点内存/2, 32GB）；

（3）设置最大文件句柄数；

（4）线程池+队列大小根据业务需要做调整；

（5）磁盘存储 raid 方式——存储有条件使用 RAID10，增加单节点性能以及避免单节点存储故障。

Elasticsearch 是如何实现 Master 选举的

（1）Elasticsearch 的选主是 ZenDiscovery 模块负责的，主要包含 Ping（节点之间通过这个 RPC 来发 现彼此）和 Unicast（单播模块包含一个主机列表以控制哪些节点需要 ping 通）这两部分；

（2）对所有可以成为 master 的节点（node.master: true）根据 nodeId 字典排序，每次选举每个节 点都把自己所知道节点排一次序，然后选出第一个（第 0 位）节点，暂且认为它是 master 节点。

（3）如果对某个节点的投票数达到一定的值（可以成为 master 节点数 n/2+1）并且该节点自己也选举自己，那这个节点就是 master。否则重新选举一直到满足上述条件。

（4）补充：master 节点的职责主要包括集群、节点和索引的管理，不负责文档级别的管理；data 节点可以关闭 http 功能

Elasticsearch 更新和删除文档的过程

（1）删除和更新也都是写操作，但是 Elasticsearch 中的文档是不可变的，因此不能被删除或者改动以展示其变更；

（2）磁盘上的每个段都有一个相应的.del 文件。当删除请求发送后，文档并没有真的被删除，而是在.del 文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在.del 文件中被标记为删除的文档将不会被写入新段。

（3）在新的文档被创建时，Elasticsearch 会为该文档指定一个版本号，当执行更新时，旧版本的文档在.del 文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉

拼写纠错是如何实现的

（1）拼写纠错是基于编辑距离来实现；编辑距离是一种标准的方法，它用来表示经过插入、删除和替换操作从一个字符串转换到另外一个字符串的最小操作步数；

（2）编辑距离的计算过程：比如要计算 batyu 和 beauty 的编辑距离，先创建一个7×8 的表（batyu 长度为 5，coffffee 长度为 6，各加 2），接着，在如下位置填入黑色数字。

设计模式

创建型模式

对象实例化的模式，创建型模式用于解耦对象的实例化过程

1. 单例模式：某个类只能有一个实例，提供一个全局的访问点。
2. 简单工厂：一个工厂类根据传入的参量决定创建出那一种产品类的实例。
3. 工厂方法：定义一个创建对象的接口，让子类决定实例化那个类。
4. 抽象工厂：创建相关或依赖对象的家族，而无需明确指定具体类。
5. 建造者模式：封装一个复杂对象的构建过程，并可以按步骤构造。
6. 原型模式：通过复制现有的实例来创建新的实例。

结构型模式

把类或对象结合在一起形成一个更大的结构。

1. 适配器模式：将一个类的方法接口转换成客户希望的另外一个接口。
2. 组合模式：将对象组合成树形结构以表示“”部分–整体“”的层次结构。
3. 装饰模式：动态的给对象添加新的功能。
4. 代理模式：为其他对象提供一个代理以便控制这个对象的访问。
5. 亨元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象。
6. 外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。
7. 桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

行为型模式

类和对象如何交互，及划分责任和算法。

1. 模板模式：定义一个算法结构，而将一些步骤延迟到子类实现。
2. 解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器。
3. 策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。
4. 状态模式：允许一个对象在其对象内部状态改变时改变它的行为。

5. 观察者模式：对象间的一对多的依赖关系。
6. 备忘录模式：在不破坏封装的前提下，保持对象的内部状态。
7. 中介者模式：用一个中介对象来封装一系列的对象交互。
8. 命令模式：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。
9. 访问者模式：在不改变数据结构的前提下，增加作用于一组对象元素的新功能。
10. 责任链模式：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。
11. 迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

Docker

什么是 Docker 容器

Docker 容器 在应用程序层创建抽象并将应用程序及其所有依赖项打包在一起。这使我们能够快速可靠地部署应用程序。容器不需要我们安装不同的操作系统。相反，它们使用底层系统的 CPU 和内存来执行任务。这意味着任何容器化应用程序都可以在任何平台上运行，而不管底层操作系统如何。我们也可以将容器视为 Docker 镜像的运行时实例。

应用场景

- Web 应用的自动化打包和发布。
- 自动化测试和持续集成、发布。
- 在服务型环境中部署和调整数据库或其他的后台应用。
- 从头编译或者扩展现有的 OpenShift 或 Cloud Foundry 平台来搭建自己的 PaaS 环境。

DockerFile

Dockerfile 是一个文本文件，其中包含我们需要运行以构建 Docker 映像的所有命令。Docker 使用 Dockerfile 中的指令自动构建镜像。我们可以 `docker build` 用来创建按顺序执行多个命令行指令的自动构建。

dockerfile 的命令摘要

```
FROM- 镜像从那里来
MAINTAINER- 镜像维护者信息
RUN- 构建镜像执行的命令，每一次RUN都会构建一层
CMD- 容器启动的命令，如果有多个则以最后一个为准，也可以为ENTRYPOINT提供参数
VOLUME- 定义数据卷，如果没有定义则使用默认
USER- 指定后续执行的用户组和用户
WORKDIR- 切换当前执行的工作目录
HEALTHCHECH- 健康检测指令
ARG- 变量属性值，但不在容器内部起作用
EXPOSE- 暴露端口
```

```
ENV- 变量属性值，容器内部也会起作用
ADD- 添加文件，如果是压缩文件也解压
COPY- 添加文件，以复制的形式
ENTRYPOINT- 容器进入时执行的命令
```

常用命令

```
//构建镜像 -t ：指定要创建的目标镜像名， .: Dockerfile 文件所在目录，可以指定Dockerfile 的绝对路径
$ docker build -t runoob/centos:6.7 .
//获取镜像
$ docker pull ubuntu
//列出镜像
$ docker images
//运行容器，-t: 在新容器内指定一个伪终端或终端，-i: 允许你对容器内的标准输入 (STDIN) 进行交互，-d: 后台运行容器，并返回容器ID；-p: 指定端口映射，格式为：主机(宿主)端口:容器端口
$ docker run -it -d --name test <image_name>
2b1b7a428627c51ab8810d541d759f072b4fc75487eed05812646b8534a2fe63
//查看正在运行的容器
$ docker ps -a
CONTAINER ID        IMAGE                  COMMAND               ...
5917eac21c36        ubuntu:15.10           "/bin/sh -c 'while t…"    ...
//查看容器内的标准输出
$ docker logs 2b1b7a428627
//启动已停止运行的容器
$ docker start b750bbbcfd88
//停止容器
$ docker stop
//删除容器
$ docker rm -f 1e560fca3906
//进入容器
$ docker attach 1e560fca3906  //退出容器会停止
$ docker exec -it 243c32535da7 /bin/bash  //如果从这个容器退出，容器不会停止
//将主机/www/runoob目录拷贝到容器96f7f14e99ab的/www目录下。
$ docker cp /www/runoob 96f7f14e99ab:/www/
//将主机/www/runoob目录拷贝到容器96f7f14e99ab中，目录重命名为www
$ docker cp /www/runoob 96f7f14e99ab:/www
//将容器96f7f14e99ab的/www目录拷贝到主机的/tmp目录中
$ docker cp  96f7f14e99ab:/www /tmp/
```

Linux

常用命令

1. `find . –name "*.xml"` 递归查找所有的xml文件
2. `ls –l | grep '.jar'` 查找当前目录中的所有jar文件
3. `ps –ef|grep tomcat` 查看所有有关 tomcat 的进程
4. `ps –ef` 显示当前所有进程
5. `ls –al` 和 `ls –la` 显示当前目录下所有文件的详细信息
6. `lsof –i :8080` 查看端口属于哪个程序
7. `du –h /opt/test` 查看/opt/test目录的磁盘使用情况
8. `df –h` 查看磁盘空间使用情况
9. `chmod u+x test.sh` 给文件拥有者增加 test.sh 的执行权限
10. `chmod u+x –R test` 给文件拥有者增加test目录及其下所有文件的执行权限
11. `docker start/stop/restart tomcat`

框架类

定时任务框架

单机

- timer

是一个定时器类，通过该类可以为指定的定时任务进行配置。TimerTask类是一个定时任务类，该类实现了Runnable接口，缺点异常未检查会中止线程

- ScheduledExecutorService

相对延迟或者周期作为定时任务调度，缺点没有绝对的日期或者时间

- spring定时框架

配置简单功能较多，如果系统使用单机的话可以优先考虑spring定时器

分布式

- Quartz

Java事实上的定时任务标准。但Quartz关注点在于定时任务而非数据，并无一套根据数据处理而定制化的流程。虽然Quartz可以基于数据库实现作业的高可用，但缺少分布式并行调度的功能

- elastic–job

当当开发的弹性分布式任务调度系统，功能丰富强大，采用zookeeper实现分布式协调，实现任务高可用以及分片，目前是版本2.15，并且可以支持云开发，这个我写了系列教程了，在码猿技术专栏公从号可以搜索阅读。

- xxl–job

是大众点评员工徐雪里于2015年发布的分布式任务调度平台，是一个轻量级分布式任务调度框架，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。

X-Job 侧重的业务实现的简单和管理的方便，学习成本简单，失败策略和路由策略丰富。推荐使用在“用户基数相对少，服务器数量在一定范围内”的情景下使用。
E-Job 关注的是数据，增加了弹性扩容和数据分片的思路，以便于更大限度的利用分布式服务器的资源。但是学习成本相对高些，推荐在“数据量庞大，且部署服务器数量较多”时使用。

分布式

CAP定理

一个分布式系统不可能同时满足一致性（C:Consistency），可用性（A: Availability）和分区容错性（P：Partition tolerance）这三个基本需求，最多只能同时满足其中的2个。

一致性

目标：
1.商品服务写入主数据库成功, 则想从数据库查询数据也成功
2.商品服务写入主数据库失败,则向从数据库查询也失败
实现：
1.写入主数据库后要数据同步到从数据库，同步有一定延迟。
2.写入主数据库后,在向从数据库同步期间要将从数据库锁定, 等待同步完成后在释放锁,以免在写新数据后,向从数据库查询到旧的数据。

可用性

目标：
1.从数据库接收到数据库查询的请求则立即能够响应数据查询结果
2.从数据库不允许出现响应超时或错误
实现：
1.写入主数据库后要将数据同步到从数据
2.由于要保证数据库的可用性,不可以将数据库中资源锁定
3.即使数据还没有同步过来,从数据库也要返回查询数据, 哪怕是旧数据,但不能返回错误和超时.

分区容错性

目标：
1.主数据库想从数据库同步数据失败不影响写操作
2.其中一个节点挂掉不会影响另一个节点对外提供服务
实现：
1.尽量使用异步取代同步操作,如使用异步方式将数据从主数据库同步到从数据库, 这样节点之间能有效的实现松耦合;
2.添加数据库节点,其中一个从节点挂掉,由其他从节点提供服务。

PRC和http

RPC架构

先说说RPC服务的基本架构吧。允许我可耻地盗一幅图哈~我们可以很清楚地看到，一个完整的RPC架构里面包含了四个核心的组件，分别是Client ,Server,Client Stub以及Server Stub，这个Stub大家可以理解为存根。分别说说这几个组件：

- 客户端（Client），服务的调用方。
- 服务端（Server），真正的服务提供者。
- 客户端存根，存放服务端的地址消息，再将客户端的请求参数打包成网络消息，然后通过网络远程发送给服务方。
- 服务端存根，接收客户端发送过来的消息，将消息解包，并调用本地的方法。

RPC是基于Socket的，即工作在会话层

RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。Dubbo支持dubbo、rmi、hessian、http、webservice、thrift、redis等多种协议，但是Dubbo官网是推荐我们使用Dubbo协议的。

1、传输协议：

RPC：可以基于HTTP协议，也可以基于TCP协议

HTTP：基于HTTP协议

从网络协议来说，Http协议与Rpc同属于应用层， 他们的底层都是tcp协议。RPC（即Remote Procedure Call，远程过程调用）和HTTP（HyperText Transfer Protocol，超文本传输协议）他们最本质的区别，就是RPC主要工作在TCP协议之上，而HTTP服务主要是工作在HTTP协议之上，我们都知道HTTP协议是在传输层协议TCP之上的，所以效率来看的话，RPC当然是要更胜一筹。

2、传输效率：

RPC：使用自定义的TCP协议，可以让请求报文体积更小，或者使用HTTP2协议，也可以很好的减小报文体积，提高传输效率

HTTP：如果是基于http1.1的协议，请求中会包含很多无用的内容，如果是基于HTTP2.0，那么简单的封装下可以作为一个RPC来使用，这时标准的RPC框架更多的是服务治理。

http协议其实是属于面向桌面浏览器的一个通信协议，对于缓存，幂等或者Cookies相关的方面做了很多的事情。但是对于服务器之间直接的交互，Rpc就能够体现出来他的优势了。自定义协议，减少数据传输：我们大概看一下http协议。请求行，请求头部，请求数据，空行。很明显对于远程调用场景，我们对于请求行的依赖不是特别的强，那么这一部分在我们应用场景下，将会成为负担，但是http协议又是固定的，我们也不可能随便修改协议的格式。所以，通过rpc协议我们可以精简请求的数据，来尽可能少的传输我们的数据。当前，rpc也可以通过http协议来进行传输。

3、性能消耗：

RPC：可以基于thrift实现高效的二进制传输

HTTP：大部分是基于json实现的，字节大小和序列化耗时都比thrift要更消耗性能

4、负载均衡：

RPC：基本自带了负载均衡策略

HTTP：需要配置Nginx、HAProxy配置

5、服务治理：（下游服务新增，重启，下线时如何不影响上游调用者）

RPC：能做到自动通知，不影响上游

HTTP：需要事先通知，如修改NGINX配置。

6、连接：

RPC：长连接

HTTP：短连接

rpc使用长连接：直接基于socket进行连接，不用每个请求都重新走三次握手的流程

前端

CSS引入的方式有哪些

(1)外联：<link>标签

(2)内联：<style>标签

(3)元素内嵌：元素的style属性

CSS选择符有哪些

标签选择符、类选择符、id选择符、组合选择符

”==”和“===”的不同

==只比较值，===比较值和数据类型

什么是盒子模型

在网页中，一个元素占有空间的大小由几个部分构成，其中包括元素的内容(content)，元素的内边距(padding)，元素的边框(border)，元素的外边距(margin)四个部分。这四个部分占有的空间中，有的部分可以显示相应的内容，而有的部分只用来分隔相邻的区域或区域。4个部分一起构成了css中元素的盒模型。

$(this) 和 this 关键字在 jQuery 中有何不同

前者是jQuery对象，使用jQuery方法和属性;后者是JavaScript对象，使用JavaScript方法和属性。

jQuery 里的 each() 是什么函数

each() 函数就像是 Java 里的一个 Iterator，它允许你遍历一个元素集合。你可以传一个函数给 each() 方法，被调用的 jQuery 对象会在其每个元素上执行传入的函数。

AJAX应用和传统Web应用有什么不同

在传统的Javascript编程中，如果想得到服务器端数据库或文件上的信息，或者发送客户端信息到服务器，需要建立一个HTML form然后GET或者POST数据到服务器端。用户需要点击”Submit”按钮来发送或者接受数据信息，然后等待服务器响应请求，页面重新加载。因为服务器每次都会返回一个新的页面， 所以传统的web应用有可能很慢而且用户交互不友好。使用AJAX技术， 就可以使Javascript通过XMLHttpRequest对象直接与服务器进行交互。通过HTTP Request， 一个web页面可以发送一个请求到web服务器并且接受web服务器返回的信息(不用重新加载页面)，展示给用户的还是同一个页面，用户感觉不到页面刷新，也看不到到Javascript后台进行的发送请求和接受响应。

AJAX的全称是Asynchronous JavaScript And XML(异步的JavaScript和XML).。AJAX是2005年由Google发起并流行起来的编程方法， AJAX不是一个新的编程语言，但是它是一个使用已有标准的新的编程技术。使用AJAX可以创建更好，更快，更用户界面友好的Web应用。AJAX技术基于Javascript和HTTP Request。

使用场景?

--》登录失败时不跳转页面，注册时提示用户名是否存在，二级联动等等使用场景

AJAX的优缺点都有什么

优点:

(1)最大的一点是页面无刷新，用户的体验非常好。

(2)使用异步方式与服务器通信，具有更加迅速的响应能力。

(3)可以把以前一些服务器负担的工作转嫁到客户端，利用客户端闲置的能力来处理，减轻服务器和带宽的负担，节约空间和宽带租用成本。并且减轻服务器的负担，ajax的原则是“按需取数据”，可以最大程度的减少冗余请求，和响应对服务器造成的负担。

(4)基于标准化的并被广泛支持的技术，不需要下载插件或者小程序。

缺点:

(1)不支持浏览器back按钮。

(2)安全问题 AJAX暴露了与服务器交互的细节。

(3)对搜索引擎的支持比较弱。

(4)破坏了程序的异常机制。

(5)不容易调试。

简述Ajax的工作原理

Ajax的核心是JavaScript对象XmlHttpRequest。该对象在Internet Explorer 5中首次引入，它是一种支持异步请求的技术。简而言之，XmlHttpRequest使您可以使用JavaScript向服务器提出请求并处理响应，而不阻塞用户。在创建Web站点时，在客户端执行屏幕更新为用户提供了很大的灵活性。

简单说一下HTML，CSS，javaScript在网页开发中的定位

HTML：超文本标记语言，定义网页的结构
CSS：层叠样式表，用来美化页面
JavaScript：主要用来验证表单，做动态交互(其中AJAX)

JS和JQuery的关系

JQuery是一个JS框架，封装了JS的属性和方法，并且增强了JS的功能，让用户使用起来更加方便，原来使用js是要处理很多兼容性的问题(注册事件)，由于Jquery封装了底层，就不用处理兼容性问题(注册事件等)。
原生的js的dom和事件绑定Ajax等操作非常麻烦，JQuery等装以后非常方便。

简单说一下css3

css3是最新版本的css，是对原来的css2的功能增强
css3中提供一些css2中实现起来比较困难或者不能实现的功能。
(1)盒子圆角边框
(2)盒子和文字的阴影
(3)渐变
(4)装换、移动、缩放、旋转等
(5)过渡、动画都可以使用动画
(6)可以使用媒体查询实现响应式网站

对于MVVM的理解

MVVM 是 Model-View-ViewModel 的缩写。
Model代表数据模型，也可以在Model中定义数据修改和操作的业务逻辑。
View 代表UI 组件，它负责将数据模型转化成UI 展现出来。
ViewModel 监听模型数据的改变和控制视图行为、处理用户交互，简单理解就是一个同步View 和 Model的对象，连接Model和View。
在MVVM架构下，View 和 Model 之间并没有直接的联系，而是通过ViewModel进行交互，Model 和 ViewModel 之间的交互是双向的， 因此View 数据的变化会同步到Model中，而Model 数据的变化也会立即反应到View 上。
ViewModel 通过双向数据绑定把 View 层和 Model 层连接了起来，而View 和 Model 之间的同步工作完全是自动的，无需人为干涉，因此开发者只需关注业务逻辑，不需要手动操作DOM, 不需要关注数据状态的同步问题，复杂的数据状态维护完全由 MVVM 来统一管理。

vue实现双向数据绑定

vue实现数据双向绑定主要是：采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty（）来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应监听回调。当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历

它的属性，用 Object.defineProperty 将它们转为 getter/setter。用户看不到 getter/setter，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。
vue的数据双向绑定 将MVVM作为数据绑定的入口，整合Observer，Compile和Watcher三者，通过Observer来监听自己的model的数据变化，通过Compile来解析编译模板指令（vue中是用来解析{{}}），最终利用watcher搭起observer和Compile之间的通信桥梁，达到数据变化 —>视图更新；视图交互变化（input）—>数据model变更双向绑定效果。

Vue组件间的参数传递

1.父组件与子组件传值
父组件传给子组件：子组件通过props方法接受数据; 子组件传给父组件：$emit方法传递参数
2.非父子组件间的数据传递，兄弟组件传值
eventBus，就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。项目比较小时，用这个比较合适。

v-show 与 v-if 区别

v-show和v-if的区别：
v-show是css切换，v-if是完整的销毁和重新创建。
使用
频繁切换时用v-show，运行时较少改变时用v-if
v-if= ‘false’ v-if是条件渲染，当false的时候不会渲染

Vue的生命周期

Vue 实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、销毁等一系列过程，称之为 Vue 的生命周期。
它的生命周期中有多个事件钩子，让我们在控制整个Vue实例的过程时更容易形成好的逻辑。
`beforeCreate` （创建前） 在数据观测和初始化事件还未开始
`created` （创建后） 完成数据观测，属性和方法的运算，初始化事件，$el属性还没有显示出来
`beforeMount` （载入前） 在挂载开始之前被调用，相关的render函数首次被调用。实例已完成以下的配置：编译模板，把data里面的数据和模板生成html。注意此时还没有挂载html到页面上。
`mounted` （载入后） 在el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的html内容替换el属性指向的DOM对象。完成模板中的html渲染到html页面中。此过程中进行ajax交互。
`beforeUpdate` （更新前） 在数据更新之前调用，发生在虚拟DOM重新渲染和打补丁之前。可以在该钩子中进一步地更改状态，不会触发附加的重渲染过程。
`updated` （更新后） 在由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。调用时，组件DOM已经更新，所以可以执行依赖于DOM的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。

`beforeDestroy` （销毁前） 在实例销毁之前调用。实例仍然完全可用。
`destroyed` （销毁后） 在实例销毁之后调用。调用后，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。

第一次页面加载会触发哪几个钩子

beforeCreate, created, beforeMount, mounted

DOM 渲染在 哪个周期中就已经完成

DOM 渲染在 mounted 中就已经完成了

绑定 class 的数组用法

对象方法 v-bind:class="{'orange': isRipe, 'green': isNotRipe}"
数组方法v-bind:class="[class1, class2]"
行内 v-bind:style="{color: color, fontSize: fontSize+'px' }"

计算属性computed和 监听watch 的区别

计算属性是自动监听依赖值的变化，从而动态返回内容
监听是一个过程，在监听的值变化时，可以触发一个回调，并做一些事情。
所以区别来源于用法，只是需要动态值，那就用计算属性；需要知道值的改变后执行业务逻辑，才用watch，用反或混用虽然可行，但都是不正确的用法。
1、computed 是一个对象时，它有哪些选项?
有get和set两个选项
2、computed 和 methods 有什么区别?
methods是一个方法，它可以接受参数，而computed不能，computed是可以缓存的，methods不会
3、computed 是否能依赖其它组件的数据?
computed可以依赖其他computed，甚至是其他组件的data
4、watch 是一个对象时，它有哪些选项?
handler deep 是否深度 immeditate 是否立即执行
总结
当有一些数据需要随着另外一些数据变化时，建议使用computed。 当有一个通用的响应数据变化的时候，要执行一些业务逻辑或异步操作的时候建议使用watcher

Vue的路由实现：hash模式 和 history模式

hash模式：在浏览器中符号“#”，#以及#后面的字符称之为hash，用window.location.hash读取；
特点：hash虽然在URL中，但不被包括在HTTP请求中；用来指导浏览器动作，对服务端安全无用，

hash不会重加载页面。 hash 模式下，仅 hash 符号之前的内容会被包含在请求中，如 http://www.xxx.com，因此对于后端来说，即使没有做到对路由的全覆盖，也不会返回 404 错误。

history模式：history采用HTML5的新特性；且提供了两个新方法：pushState（），replaceState（）可以对浏览器历史记录栈进行修改，以及popState事件的监听到状态变更。 history 模式下，前端的 URL 必须和实际向后端发起请求的 URL 一致，如 http://www.xxx.com/items/id。后端如果缺少对 /items/id 的路由处理，将返回 404 错误。

Vue与Angular以及React的区别

与React的区别

相同点：

React采用特殊的JSX语法，Vue.js在组件开发中也推崇编写.vue特殊文件格式，对文件内容都有一些约定，两者都需要编译后使用；中心思想相同：一切都是组件，组件实例之间可以嵌套；都提供合理的钩子函数，可以让开发者定制化地去处理需求；都不内置列数AJAX，Route等功能到核心包，而是以插件的方式加载；在组件开发中都支持mixins的特性。

不同点：

React采用的Virtual DOM会对渲染出来的结果做脏检查；Vue.js在模板中提供了指令，过滤器等，可以非常方便，快捷地操作Virtual DOM。

事件修饰符

- 绑定一个原生的click事件： 加native，
- 其他事件修饰符：stop prevent self
- 组合键：click.ctrl.exact 只有ctrl被按下的时候才触发

组件中 data 为什么是函数

为什么组件中的 data 必须是一个函数，然后 return 一个对象，而 new Vue 实例里，data 可以直接是一个对象？

因为组件是用来复用的，JS 里对象是引用关系，这样作用域没有隔离，而 new Vue 的实例，是不会被复用的，因此不存在引用对象的问题

对于Vue是一套渐进式框架的理解

vue.js的两个核心是什么

数据驱动和组件化

vue中 key 值的作用

使用key来给每个节点做一个唯一标识

key的作用主要是为了高效的更新虚拟DOM。另外vue中在使用相同标签名元素的过渡切换时，也会使用到key属性，其目的也是为了让vue可以区分它们，否则vue只会替换其内部属性而不会触发过渡

效果。

v-for 与 v-if 的优先级

v-for的优先级比v-if高

组件

1、vue中子组件调用父组件的方法

第一种方法是直接在子组件中通过this.$parent.event来调用父组件的方法。

第二种方法是在子组件里用$emit向父组件触发一个事件，父组件监听这个事件就行了。

第三种是父组件把方法传入子组件中，在子组件里直接调用这个方法。

2、vue中父组件调用子组件的方法

父组件利用ref属性操作子组件方法。

```
//父:
<child ref="childMethod"></child>
//子:
method: {
  test() {
     alert(1)
  }
}
//在父组件里调用test即 this.$refs.childMethod.test()
```

vue组件之间传值

(1)父组件给子组件传值:

- 父组件调用子组件的时候动态绑定属性

```
<parent :dataList='dataList'></parent>
```

- 子组件定义props接收动态绑定的属性props: ['dataList']
- 子组件使用数据

(2)子组件主动获取父子间的属性和方法:

在子组件中使用this.$parent.属性/this.$parent.方法。

(3)子组件给父组件传值:

```
一、使用ref属性
1.父组件调用子组件时绑定属性ref
<parent :ref='parent'></parent>
2.在父组件中使用this.$refs.parent.属性/this.$refs.parent.方法
```

```
二、使用$emit方法
1.子组件调用this.$emit('方法名',传值)
2.父组件通过子组件绑定的'方法名'获取传值。
```

怎么定义vue-router的动态路由？怎么获取传过来的值?

动态路由的创建，主要是使用path属性过程中，使用动态路径参数，以冒号开头，如下：

vue-router有哪几种路由守卫

全局守卫：beforeEach

后置守卫：afterEach

全局解析守卫：beforeResolve

路由独享守卫：beforeEnter

加密

对称密钥、非对称密钥

对称密钥

发送和接收数据的双方必须使用相同的/对称的密钥对明文进行加密和解密运算。最大优势是加/解密速度快，适合于对大数据量进行加密，但密钥管理困难

- DES（56位密钥）、IDEA和AES（支持128、192、256、512位密钥的加密）

非对称密钥

也叫公开密钥加密，指每个人都有一对唯一对应的密钥：公开密钥和私有密钥，公钥对外公开，私钥由个人秘密保存；用其中一把密钥来加密，就只能用另一把密钥来解密。发送数据的一方用另一方的公钥对发送的信息进行加密，然后由接受者用自己的私钥进行解密。公开密钥加密技术解决了密钥的发布和管理问题，是目前商业密码的核心。使用公开密钥技术，进行数据通信的双方可以安全地确认对方身份和公开密钥，提供通信的可鉴别性。但加密和解密速度却比对称密钥加密慢得多。

- **RSA**

  目前最有影响力的公钥加密运算，将两个大素数相乘十分容易，但想要对其乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥，即公钥，而两个大素数组合成私钥。公钥是可发布的供任何人使用，私钥则为自己所有。

  DSA

  这是一种更高级的验证方式，用作数字签名。不单单只有公钥、私钥，还有数字签名。私钥加密生成数字签名，公钥验证数据及签名，如果数据和签名不匹配则认为验证失败。数字签

名的作用就是校验数据在传输过程中不被修改，数字签名，是单向加密的升级。

(1) 使用消息摘要算法将发送数据加密生成数字摘要。

(2) 发送方用自己的私钥对摘要再加密，形成数字签名。

(3) 将原文和加密的摘要同时传给对方。

(4) 接受方用发送方的公钥对摘要解密，同时对收到的数据用消息摘要算法产生同一摘要。

(5) 将解密后的摘要和收到的数据在接收方重新加密产生的摘要相互对比，如果两者一致，则说明在传送过程中信息没有破坏和篡改。否则，则说明信息已经失去安全性和保密性。

非对称密钥加密的使用过程：
1. A要向B发送信息，A和B都要先在系统中生成一对用于加密和解密的公钥和私钥。
2. A的私钥保密，A的公钥告诉B；B的私钥保密，B的公钥告诉A。
3. A要给B发送信息时，A用B的公钥加密信息，因为A知道B的公钥。
4. A将这个消息发给B（已经用B的公钥加密消息）。
5. B收到这个消息后，B用自己的私钥解密A的消息，其他所有收到这个报文的人都无法解密，因为只有B才有B的私钥。
6. 反过来，B向A发送消息也是一样。

混合使用

将对称加密的密钥使用非对称加密的公钥进行加密，然后发送出去，接收方使用私钥进行解密得到对称加密的密钥，然后双方可以使用对称加密来进行沟通。

区别

1．密钥不同

对称加密：对称加密加密和解密使用同一个密钥

非对称加密：非对称加密加密和解密所使用的不是同一个密钥，需要两个密钥来进行加密和解密

2．安全性不同

对称加密：对称加密如果用于通过网络传输加密文件，那么不管使用任何方法将密钥告诉对方，都有可能被窃听。

非对称加密：非对称加密因为它包含有两个密钥，且仅有其中的“公钥”是可以被公开的，接收方只需要使用自己已持有的私钥进行解密，这样就可以很好的避免密钥在传输过程中产生的安全问题。

3．数字签名不同

对称加密：对称加密不可以用于数字签名和数字鉴别

非对称加密：非对称加密可以用于数字签名和数字鉴别

4．算法上区别

对称密钥算法采用的分组加密技术，即将待处理的明文按照固定长度分组，并对分组利用密钥进行数次的迭代编码，最终得到密文。解密的处理同样，在固定长度密钥控制下，以一个分组为单位进行数次迭代解码，得到明文。
非对称密钥算法采用一种特殊的数学函数，单向陷门函数（ one way trapdoor function ），即从一个方向求值是容易的，而其逆向计算却很困难，或者说是计算不可行的。加密时对明文利用公钥进行加密变换，得到密文。解密时对密文利用私钥进行解密变换，得到明文。

安全hash算法

不可逆的
将数据进行加密，发送数据时把密文和明文同时发送，接收方根据明文再次加密，得到结果和密文一直则说明数据在传输过程中没有被篡改过

> MD5、SHA

乱七八糟

负载均衡*Load Balance*

策略

轮询

平均分配，人人都有、一人一次

平滑加权轮询

三个节点，权重 weight：5，1，1
A（5）B（1）C（1）
初始动态权重 currentWeight 0，0，0

currentWeight+=weight	max(currentWeight)	result	maxCurrentWeight-=sum(weight)
5,1,1	5	A	-2,1,1
3,2,2	3	A	-4,2,2
1,3,3	3	B	1,-4,3
6,-3,4	6	A	-1,-3,4
4,-2,5	5	C	4,-2,-2
9,-1,-1	9	A	2,-1,-1

7,0,0	7	A	0,0,0
5,1,1	5	A	-2,1,1

AABACAA

最少连接数

根据实时的负载情况，进行动态负载均衡的方式。维护好活动中的连接数量，然后取最小的返回即可。

最快响应

本质是根据每个节点对过去一段时间内的响应情况来分配，响应越快分配的越多。具体的运作方式也有很多，上图的这种可以理解为，将最近一段时间的请求耗时的平均值记录下来，结合前面的「加权轮询」来处理，所以等价于2：1：3的加权轮询。

题外话：一般来说，同机房下的延迟基本没什么差异，响应时间的差异主要在服务的处理能力上。如果在跨地域（例：浙江->上海，还是浙江->北京）的一些请求处理中运用，大多数情况会使用定时「ping」的方式来获取延迟情况，因为是OSI的L3转发，数据更干净，准确性更高。

Hash法

对比

健康探测

不管是什么样的策略，难免会遇到机器故障或者程序故障的情况。所以要确保负载均衡能更好的起到效果，还需要结合一些「健康探测」机制。定时的去探测服务端是不是还能连上，响应是不是超出预期的慢。如果节点属于“不可用”的状态的话，需要将这个节点临时从待选取列表中移除，以提高可用性。一般常用的「健康探测」方式有3种。

HTTP探测

使用Get/Post的方式请求服务端的某个固定的URL，判断返回的内容是否符合预期。一般使用Http状态码、response中的内容来判断。

TCP探测

基于Tcp的三次握手机制来探测指定的IP + 端口。最佳实践可以借鉴阿里云的SLB机制，如下图。

UDP探测

可能有部分应用使用的UDP协议。在此协议下可以通过报文来进行探测指定的IP + 端口。最佳实践同样可以借鉴阿里云的SLB机制，如下图。

结果的判定方式是：在服务端没有返回任何信息的情况下，默认正常状态。否则会返回一个ICMP的报错信息。

限流 rate limit

限流场景:突发流量、恶意流量、业务本身需要

基于计数器的限流器

基于滑动窗口的限流

露桶算法

首先，我们有一个固定容量的桶，有水流进来，也有水流出去。对于流进来的水来说，我们无法预计一共有多少水会流进来，也无法预计水流的速度。但是对于流出去的水来说，这个桶可以固定水流出的速率。而且，当桶满了之后，多余的水将会溢出。当使用了漏桶算法，我们可以保证接口会以一个常速速率来处理请求

```
// 桶的容量
public int capacity = 10;
// 当前水量
public int water = 0;
//水流速度/s
public int rate = 4;
// 最后一次加水时间
public long lastTime = System.currentTimeMillis();
```

代码逻辑

```
public class LeakyDemo {
    public long timeStamp = System.currentTimeMillis(); // 当前时间
    public int capacity; // 桶的容量
    public int rate; // 水漏出的速度
    public int water; // 当前水量(当前累积请求数)
```

```
    public boolean acquire() {
        long now = System.currentTimeMillis();
        water = max(0, water - (now - timeStamp) * rate);
        // 先执行漏水，计算剩余水量
        timeStamp = now;
        if ((water + 1) < capacity) {
            // 尝试加水,并且水还未满
            water += 1;  return true;
        } else {
            // 水满，拒绝加水
            return false;
        }
    }
}
```

令牌桶算法