## MYSQL面试记录

### 关于面试官提问项目上MYSQL是怎么优化的?

首先这个问题有点范，我一般会从这几个点去优化。SQL及索引优化，数据库表结构优化，系统配置，硬件

#### 第一，我会从数据库配置上的优化。

例如

数据库的最大连接数(**max\_connections**)是否设置正确。

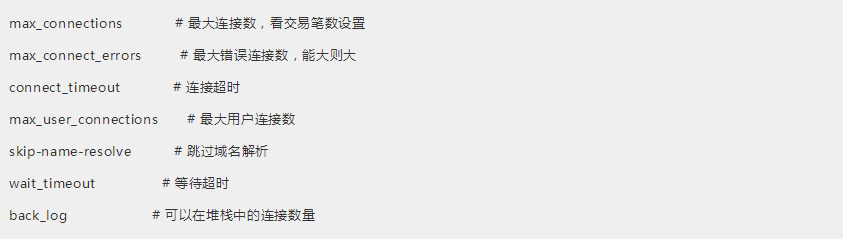
Innodb缓冲池(**innodb\_buffer\_pool\_size**)设置是否正确，一般根据你机器的内存设置。Redo

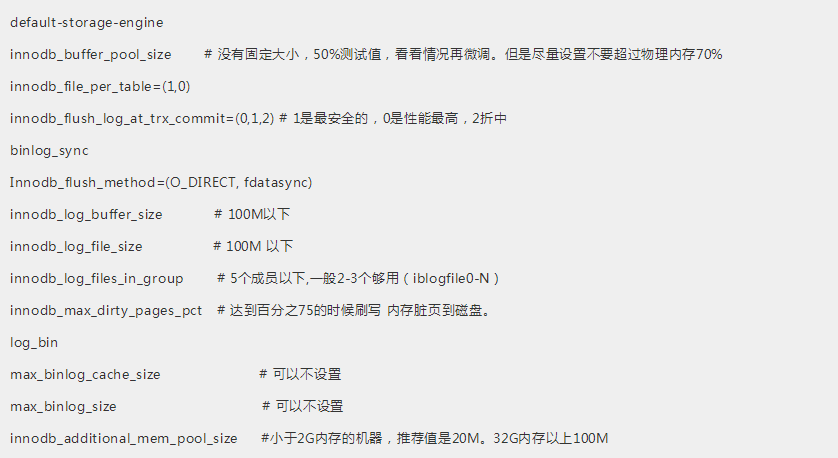
日志的大小(**innodb\_log\_file\_size**)设置是否正确，redo日志主要是用于数据库崩溃恢复用的。

并发线程数量个数(**thread\_concurrency**),用于设置Mysql的允许的并发数量

排序缓存(**sort\_buffer\_size**)







#### 我会根据需求对项目上的SQL进行优化，这个需要对不同的业务场景进行不同的分析优化。

SQL的优化主要根据执行计划命令去调优，然后根据表建的索引是否正确，语句是否正确使用索引来分析。我这里举几个以前工作时优化的例子。

1、首先我会看这个SQL上用到的表是MyIsam还是Innodb的引擎，如果是MyIsam的话，因为这个引擎会锁表，性能比较差，是否可以换成用Innodb。

2、如果是Innodb，拿单表查询来说，这条语句在符合业务需求下，是否可以直接用聚簇索引，因为Innodb本身是数据既索引，就拿常见的B+索引来说，在存储上，数据以页的形式存储，页内的数据按照主键进行排序，B+索引的叶子节点按照主键顺序存放每一条的数据，所以直接通过主键查询是最快的。如果不行，看看where后面的提交查询是列是否有建立普通索引，虽然普通索引查询时候会回表，但也比没索引快非常多。

3、还有可以看看是否可以直接用上覆盖索引，一般很多人就直接写select \*,其实很多时候没用到。

4、是否写的SQL有没有按照最左前缀匹配。

5、对一些数据可以用视图或者存储过程。

6、在对索引的创建上应该合理，例如，

索引最好只创建where和order涉及的列。这里如果order涉及的列不是索引的话，就会采用加载到排序缓存中排序，占用内存

索引列的值应该是重复率比较少的，这里的原因是重复率高还要回表操作(mysql有执行优化过，所以这里其实会判断是直接全表扫描还是走回表操作)。

#### 第三，系统层面上对MYSQL的优化

1、数据库配置读写分离，一主多从，应用程序上设置多个数据源

2、查询数据库属于IO操作，程序上可以访问数据库是进行多线程访问。

3、使用shardingjdbc进行分库分表

#### 第四，表结构优化

表中列字段的选择。主要是为了减少IO的读写。例如能用Int,就不要用BigInt。能用varchar，char就不要用Text

## Mybatis面试记录

### Mybatis用到了哪些设计模式

#### 建造者模式

mybatis用了很多建造者模式，例如SqlSessionFactoryBuilder创建SqlSessionFactory时，需要解析整个配置文件加载到Configuration对象中，所以将这个复杂的创建对象过程交给了XmlConfigBuilder。加载mapper映射文件的过程交给了XMLMapperBuilder。构建SQL语句过程交给了XMLStatementBuilder。

对于CacheBuilder这种具有可选参数的类，MyBatis还是用了经典的连点设计.

|  |
| --- |
| Cache cache = new CacheBuilder(currentNamespace)  .implementation(valueOrDefault(typeClass, PerpetualCache.class))  .addDecorator(valueOrDefault(evictionClass, LruCache.class))  .clearInterval(flushInterval)  .size(size)  .readWrite(readWrite)  .blocking(blocking)  .properties(props)  .build(); |

总之，Myabtis中的建造者模式将对象的创建和使用分离开，使用单独的建造者来创建对象

#### 装饰者模式

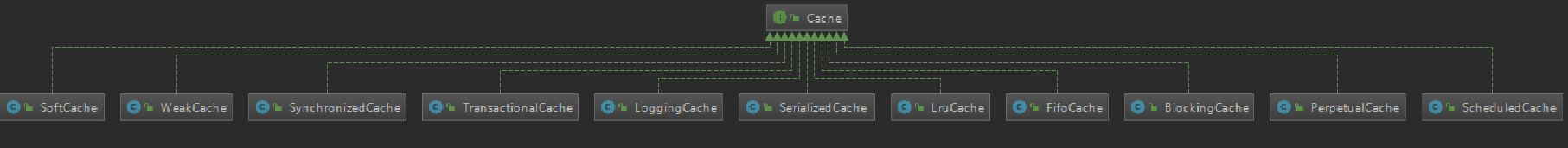
Mybatis在cache中采用了装饰器模式。

例如执行器这一块。首先默认的是SimpleExecutor执行器，这个执行器是普通的查询的执行器，然后默认我们查询的时候他是采用了CachingExecotr进行装饰后，里面的query方法经过装饰后就加了缓存的执行逻辑。





例如Cache缓存这一块，默认的也是PerpetualCache缓存，但是根据不同装饰器去装饰可以实现不同的效果，例如采用LruCache装饰后就有了LRU特性

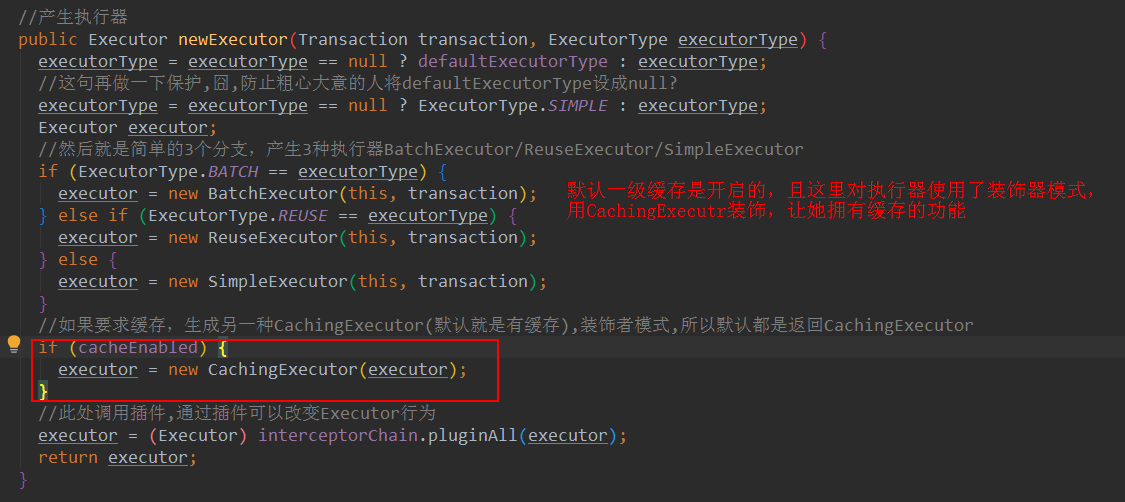


### Mybatis加载mappers文件的时候一共有几种方式

4种。 一种是通过resource，一种是mapperClass，一种是加载整个包package，一种是url。

看代码package的优先级最高。

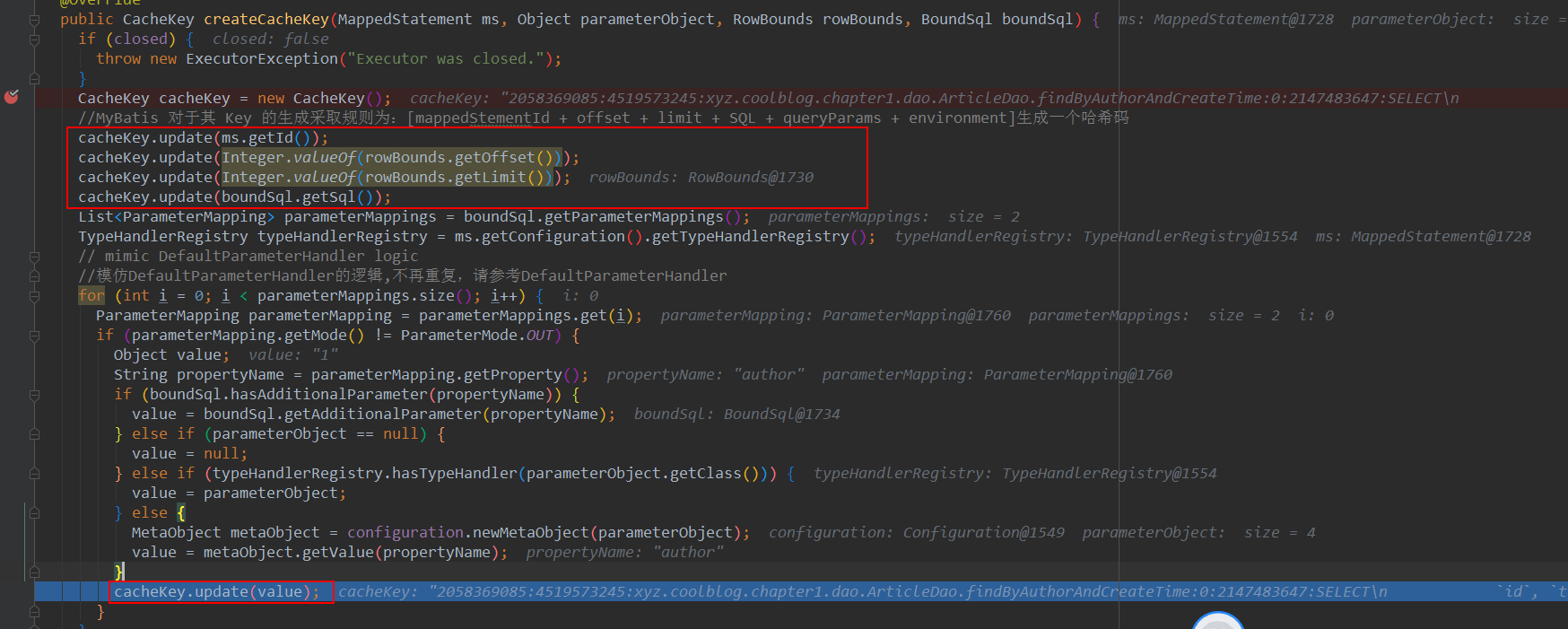
### Mybatis一级缓存默认是开启还是关闭



### Mybatis是怎么判断是同一个一级缓存的

Mybatis中有一个CacheKey对象，根据源码。一个语句的缓存key包括如下组合

1. 语句的ID，例如 xyz.coolblog.chapter1.dao.ArticleDao.findByAuthorAndCreateTime
2. 如果有分页，还包括offset,limit这两个数
3. 整条SQL语句
4. 如果语句有包括参数，则包括参数的值

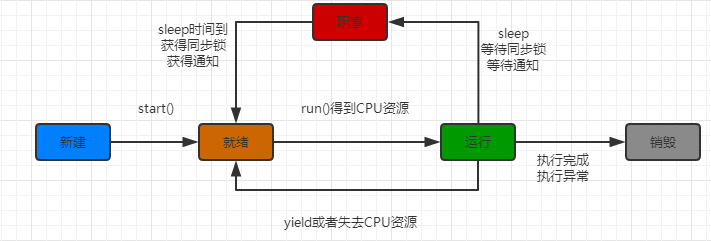


## JAVA面试记录 -- 并发

### 线程生命周期

线程的生命周期包括：

1. 新建 --- 就是刚使用new方法，new出来新线程
2. 就绪 --- 调用了线程的start方法，这时线程处于等待CPU分配资源阶段，谁先抢到谁先执行
3. 运行 --- 当就绪的线程被调度并获得CPU资源时，便进入运行状态
4. 阻塞 --- 在运行状态的时候，可能因为某些原因导致运行状态的线程变成了阻塞状态，例如sleep,wait方法造成阻塞，唤醒后不会马上进入运行状态，需要抢CPU
5. 销毁 --- 线程正常执行完毕后或线程被提前强制性的终止或出现异常导致结束



### Sleep()和Wait()的区别

1. sleep时间到自己唤醒，wait需要别人唤醒
2. Sleep不会释放对象锁，只会让出CPU时间片 -- 理解就是这个线程用了synchronized加锁了即使执行了sleep还是加锁状态，但是执行了wait就解锁了
3. Sleep是Thread的方法，wait是Obeject的方法
4. Thread.Sleep(0)的作用是“触发操作系统立刻重新进行一次CPU竞争

### 导致死锁的原因？有什么解决方案？

Xxxx

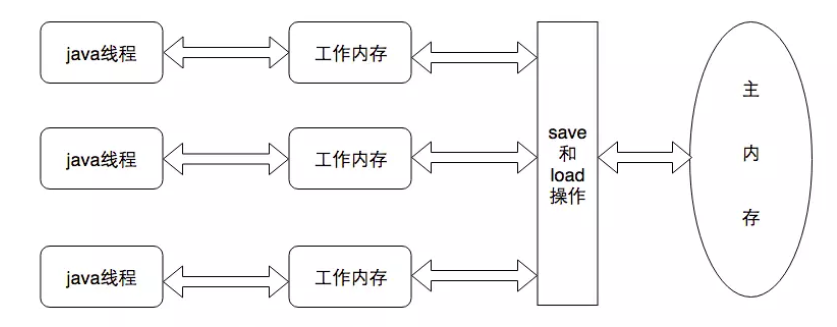
### Interrupted和isInterrupted方法的区别？

### 什么是多线程的上下文切换?

CPU处理任务的时候不是一直处理一个，而是通过给每个线程分配CPU时间片，用完了就换下一个线程，所以CPU通过时间片分配算法来循环执行多个线程的任务，那可能一个线程还没执行完时间片到了CPU就换下一个线程了，那此时就需要将原来的线程的状态，然后后面才可以加载这个线程的状态继续执行。从任务保存到再加载的过程就是一次上下文切换

### 说一说JMM(JAVA内存模型)

通俗来说，JMM主要是定义了程序中变量的访问规则。即在虚拟机中将变量存储到主内存或者将变量从主内存取出这样的底层细节



由于JVM运行程序的实体是线程,而每个线程创建时JVM都会为其创建一个工作内存(有些地方成为栈空间),工作内存是每个线程的私有数据区域,而Java内存模型中规定所有变量都存储在主内存,主内存是共享内存区域,所有线程都可访问,但线程对变量的操作(读取赋值等)必须在工作内存中进行,首先要将变量从主内存拷贝到自己的工作空间,然后对变量进行操作,操作完成再将变量写回主内存,不能直接操作主内存中的变量,各个线程中的工作内存储存着主内存中的变量副本拷贝,因此不同的线程无法访问对方的工作内存,此案成间的通讯(传值) 必须通过主内存来完成

JMM关于同步的规定:

1. 线程解锁前，必须把共享变量的值刷新回主内存
2. 线程加锁前，必须取主内存的最新值到自己的工作内存
3. 加锁和解锁是同一把锁

### 说说指令重排序是什么东西

计算机在执行时为了提供性能，编译器和处理器会对指令进行重排序，过程如下

最终执行

内存系统重排

指令并行重排

编译器重排

源代码

虽然会重排，但是JMM先天具备有序性，就是happens-before原则，反正就是说重排可以，但是必须考虑到数据的依赖性，例如编译器你要对我两条指令重排序，但不符合happends-before原则，那就不能重排。

### 谈谈你对volatile的理解

原理是：通过插入内存屏障指令禁止编译器和CPU对程序进行重排序。

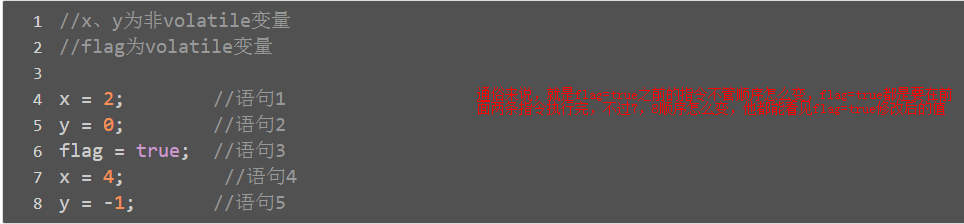
用volatile声明的变量进行读写时，JVM就向处理器发送一条Lock前缀的指令，这条指令产生两个作用:

1. Lock前缀指令会引起处理器缓存回写到系统内存，并使用缓存一致性机制来确保回写的原子性
2. 一个处理器的缓存回写到系统内存会导致其他处理器的缓存无效

Volatitle用来修饰变量从而保证了:

1. 可见性，通过JMM我们知道不同线程直接修改共享变量可能存在A线程改了W变量，同时B线程也改了W变量时，两个线程直接修改的值对于彼此是不可见的，从而导致了工作内存和主内存同步延迟的现象，而用volatile修饰能保证共享变量改的值能及时同步到主内存(**volatie保证了用这个修饰的变量会把该线程对应的本地内存中的值立即刷新到主存,JMM会直接从主存读取该变量**)
2. 禁止指令重排序 -- 有两层意思。第一，当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行。第二，在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

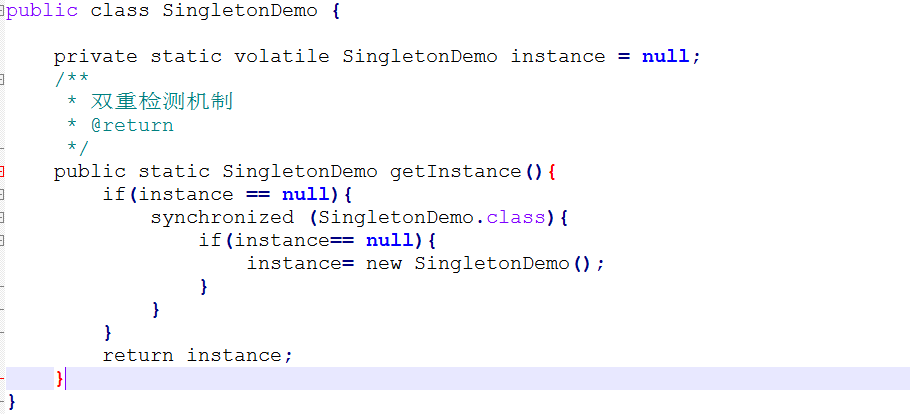
**如图，通俗来说flag是volatile的话，不管指令怎么重排序，都不可能将4，5放到6后面，也不可能，也可能将7，8放到6前面。6必须在4，5后面，在7，8前面。但是4，5直接的顺序和7，8之间的顺序没法做保证**



1. 不能保证原子性。**例如number++不能保证。如何用其他方法保证原子性? --- synchronized或者CAS(AtomicInteger)或者ReentrantLock**

### 说说哪些地方用到了volatile？

#### 单例模式的双检查锁用到了volatile -- 为什么要加volatile？



Instance = new StringtonDemo()指令的步骤可以分成如下

**memory=allocate();//1.分配对象内存空间**

**instance(memory);//2.初始化对象**

**instance=memory;//3.设置instance的指向刚分配的内存地址,此时instance!=null**

由于2，3不存在依赖关系，编译器是可以对他进行重排序的。如果发生了重排序，设置指向刚分配的内存地址，此时instance!=null，但此时对象其实初始化对象，这样就造成了线程安全问题。所以使用volatile来禁止指令的重新排序

#### AtomicInteger原子类中用到了volatile来修饰内存值



**为什么要用volatile？ -- 因为CAS获取内存值与旧的预测值进行比较，他可以保证CAS在获取内存值时都是能获取到最新的值。保证线程的值能及时更新到主存**

#### AQS中同步状态state用了volatile修饰

他也是要保证每个线程加锁解锁后修改state后能被其他线程看到。保证可见性。

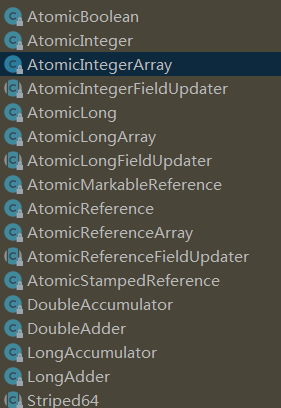
### 说一说CAS，java中哪些用到了CAS？

通俗来说就是比较并替换，核心是采用UnSafe类。这是一个都是native修饰的类

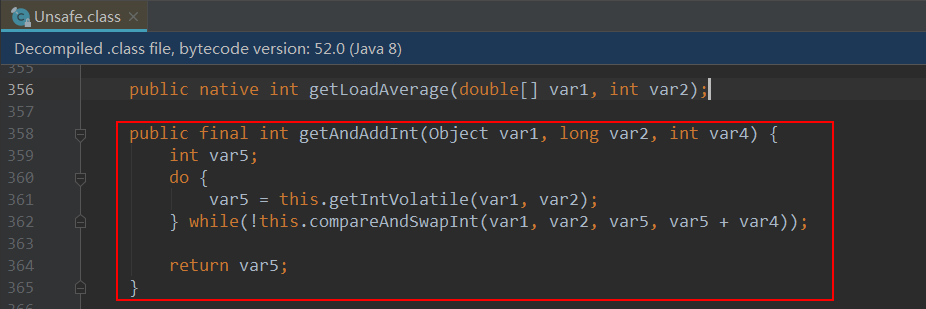
CAS使用了三个基本操作数:

1. 内存地址上的值V
2. 旧的预期值A
3. 要修改的新值B

Java中提供了原子类来实现原子性



过程分析AtomicInteger是如何通过CAS实现number++：



假设现在内存中的值V=10

线程A要将内存值改成11，则此时内存值V=10，旧的预期值A=10，要修改的值B=11

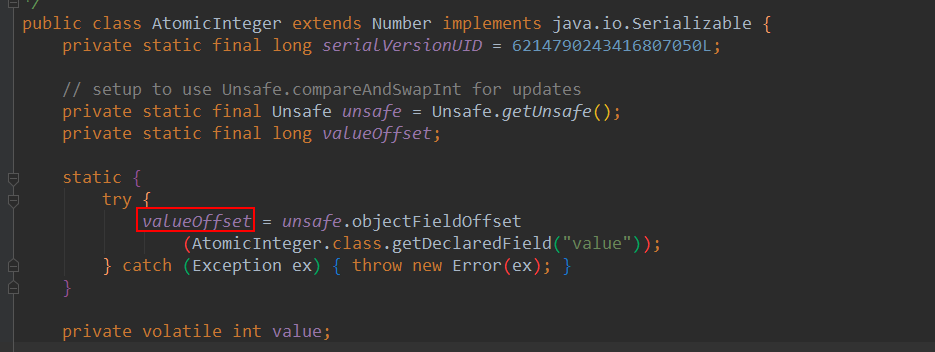
但是此时线程B抢先一步将内存值改成11，此时内存值V=11

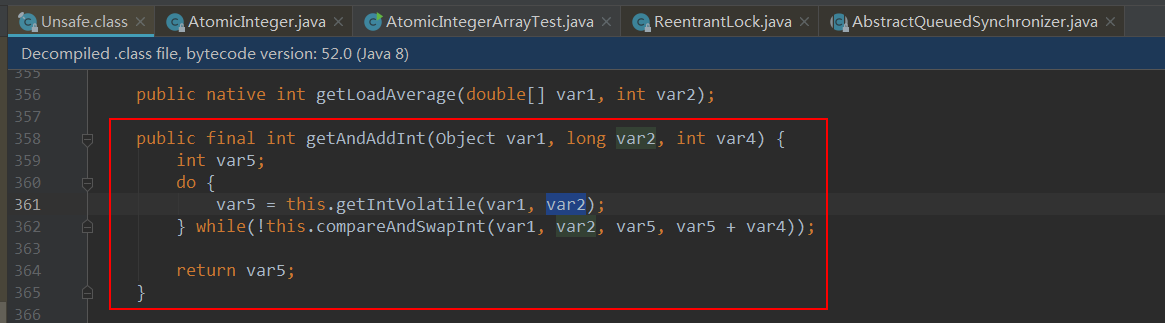
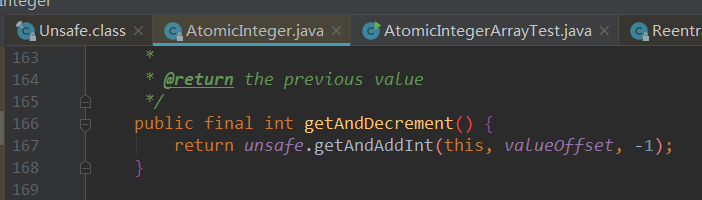
线程1此时执行时，将内存值V=11，旧的预期值A=10进行比较，发现不相等，提交失败

线程1提交失败后重新获取当前最新的内存值，并重新计算想要修改的值，这个过程称之为**自旋**

这一次比较幸运，别的线程没有抢更改内存值，线程1进行比较后发现内存值和旧的预期值相等，此时提交修改成功。

### 说说AtomicInteger是如何实现自增的？





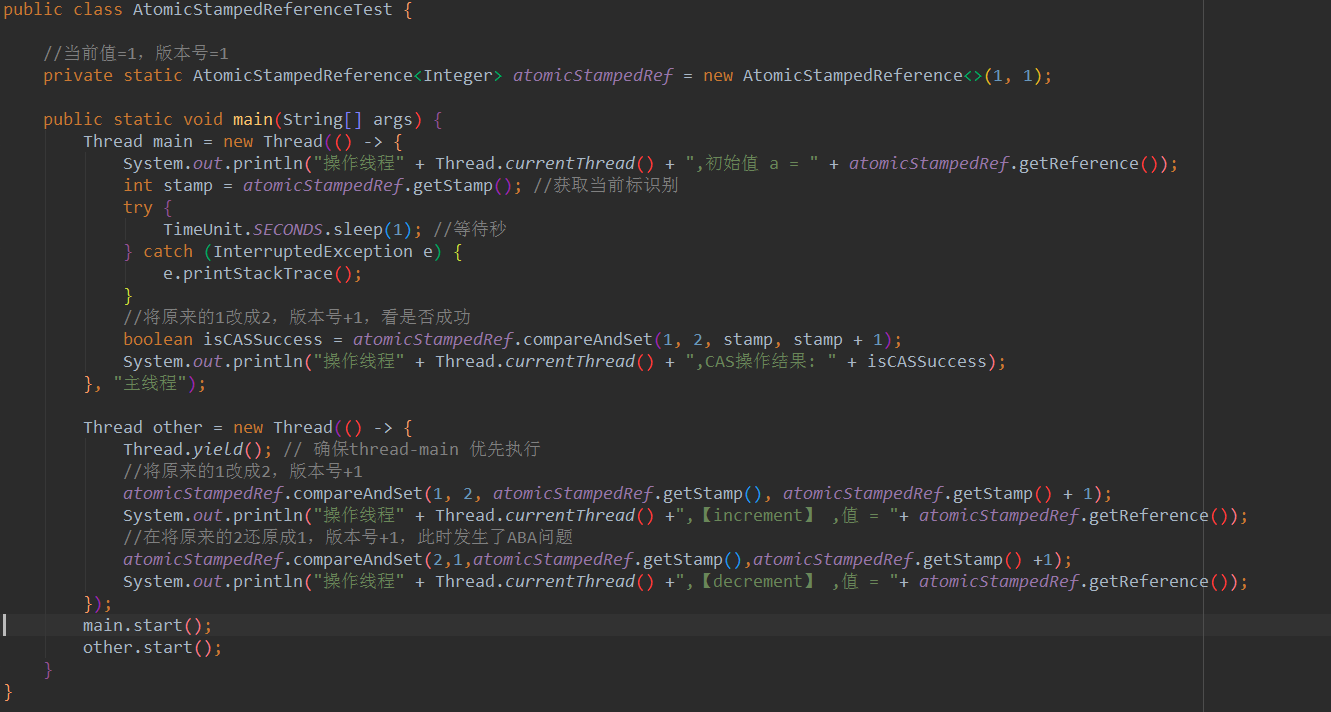
valueOffset表示当前变量值的内存地址偏移量。根据代码知道，根据内存偏移量获取到内存值，然后自旋的方式判断内存值与旧的预期值是否一致，如果一直则加上1。

### CAS的缺点有什么，怎么解决？

1. 因为采用的是CPU自旋，循环时间如果过长则浪费CPU自旋
2. 只能保证一个共享变量的原子性
3. 会引出ABA问题

ABA有什么方法可以解决

1. AtomicStampedReference 带时间戳的原子类
2. AtomicMarkableReference 带boolean类型的原子类



由于有版本号的限制，发生了ABA也不能提交成功

### 实现一个自旋锁

|  |
| --- |
| public class SpinLock {   private AtomicReference<Thread> owner = new AtomicReference<>();   private volatile int count = 0;   public static void main(String[] args) {  SpinLock lock = new SpinLock();  try{  lock.lock();  //*todo* }catch (Exception e){   }finally {  lock.unlock();  }  }   public void lock() {  Thread t = Thread.*currentThread*();  if (t == owner.get()) {  ++count;  return;  }  //自旋 判断是否是null，如果是则设置为当前线程，如果不是则自旋  while (owner.compareAndSet(null, t)) {   }  }  public void unlock() {  Thread t = Thread.*currentThread*();  if (t == owner.get()) {  if (count > 0) {  --count;  } else {  owner.set(null);  }  }  } } |

### 谈谈synchronized，如何使用？

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

1. 普通同步放，锁的是当前实例
2. 静态同步方法，锁的是当前类的Class对象
3. 同步代码块，锁的是括号里面的对象

synchronized不可以为null对象加锁。

Synchronized原理是给JAVA对象头加锁，使用监视器锁(monitor)

Synchronized是可重入的，因为底层有一个计数器

### 如果synchronized内方法发生异常，锁会不会释放?

会释放，具体要分析JAVA虚拟机内核了

### synchronized做了什么锁优化?

为了在一定程度上减少获得锁和释放锁带来的性能消耗，在 jdk6 之后便引入了“偏向锁”和“轻量级锁”，所以总共有4种锁状态，级别由低到高依次为：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态

### ThreadLocal的原理了解吗?

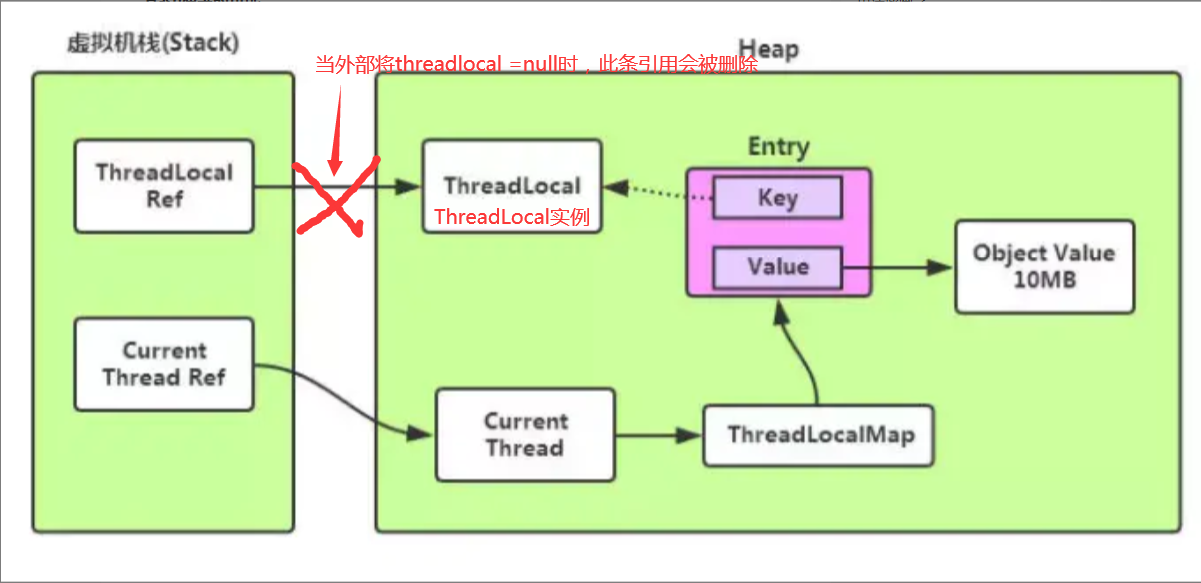
#### Thread,ThreadLocal,ThreadLocalMap之间的关系？

插入图

#### ThreadLocal会发生内存泄漏吗，会的原因，怎么解决?

##### 为什么ThreadLocal要用弱引用?

如果，堆里面存着ThreadLocal实例，当我们将ThreadLocalRef =null时，外部没有强引用ThreadLocal实例了，那就会导致ThreadLocal一直常驻内存，所以用了弱引用后，每当GC时会将ThreadLocal实例删掉，不会造成ThreadLocal的内存泄漏



##### 用了弱引用就能解决内存泄漏问题吗?

不能，虽然上述弱引用解决了ThreadLocal实例能够被回收，当Value仍然存在着内存泄漏的问题，如果一个线程的执行时间太长，value又很大，除非线程结束，不然value不会被回收

##### 如何能彻底解决内存泄漏?

1. 在一个线程内，如果value使用完后最好先调用一下threadLocal.remove()方法删掉，不要到了方法最后才删掉，可能中间有很长的执行时间会导致value常驻内存
2. 尽量不使用大内存对象

#### 为什么ThreadLocal最好定义成静态变量static？

假设ThreadLocal是某个类的成员变量，那每次这个类实例化后内存都对应了一个新的ThreadLocal实例被创建，于是同一个线程可能访问到不同的实例，没任何意义，且会造成内存的浪费。

### ArrayList为什么是线程不安全的？有什么方法可以解决?

因为ArrayList中的增删改都没有加锁或用到CAS解决，所以有线程安全问题。

具体的解决方法：

1. 使用new Vector()类 --- 这类里面的方法都是加synchronized
2. 使用Collections.synchronizedList(new ArrayList) 修饰ArrayList --- 这做了一层包装，里面也是用了synchronized
3. 使用CopyOnWriteArrayList类，写时复制类。

### CopyOnWriteArrayList如何能做到线程安全，适合什么场景，有什么缺点？

CopyOnWriteArrayList使用了一种叫**写时复制**的方法，当有新元素添加到CopyOnWriteArrayList时，先从原有的数组中拷贝一份出来，然后在新的数组做写操作，写完之后，再将原来的数组引用指向到新数组。

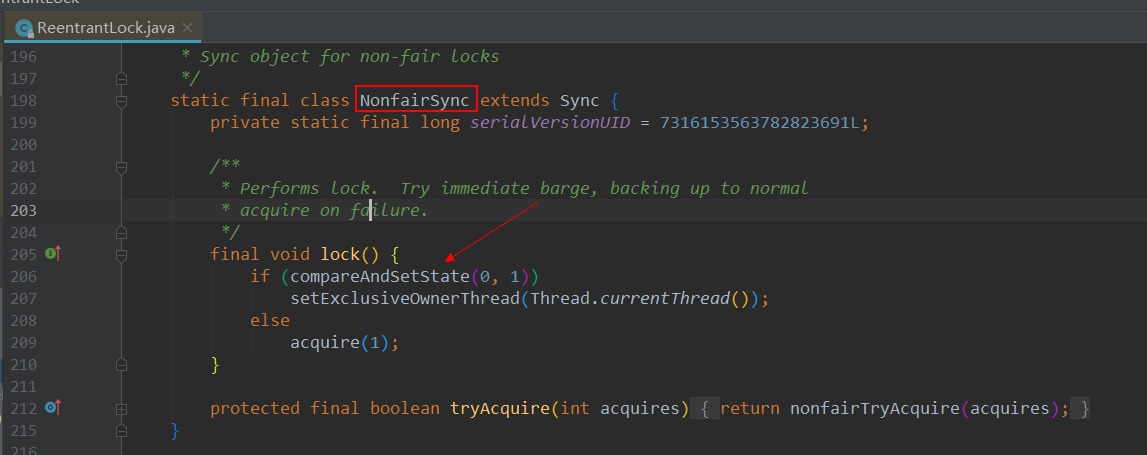
#### 优缺点和适合的场景?

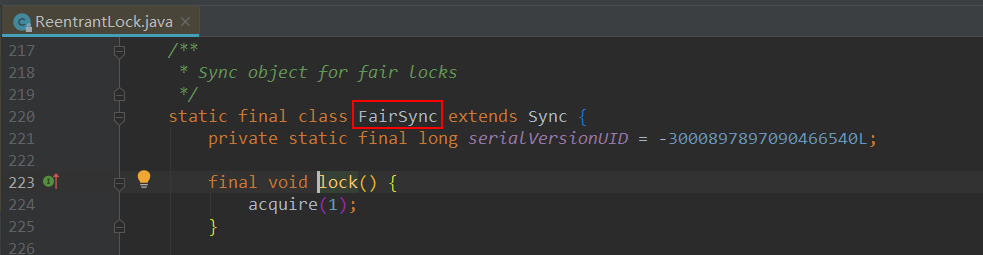
1、读写分离，适合写少读多的场景。使用了ReentrantLock支持多线程下的并发写

2、缺点占用内存，因为 CopyOnWrite 的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存（Arrays.copy是引用的复制，这里说占用内存应该是旧的数组还引用所以占了内存吧）

3、数据一致性问题，只能保证最终一致性

### ReentrantLock是如何实现公平锁和非公平锁的？



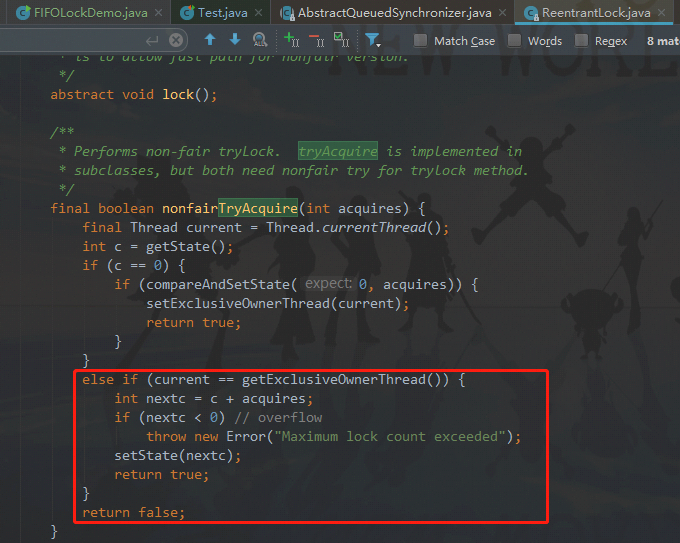


主要体现在lock方法上，如果是非公平，则在加锁前会先调用一下compareAndSetState比较并替换state，如果此时刚好有线程解锁，则就不需要入同步队列，直接执行setExclusiveOwnerThread获得锁。

题外：ReentrantLock底层是通过AQS(抽象队列同步器)来实现的，AQS维护了一个队列，当加锁失败后会加入到抽象队列中，然后按照FIFO的方式一个一个获取锁。

### ReentrantLock是如何可以实现可重入的？

通过判断是否是当前线程，如果是就将同步状态+1，然后返回true不用入队列，所以不会加锁。但是相应了同步状态+1，则unlock需要执行两次



### ReentrantLock读写锁的原理?

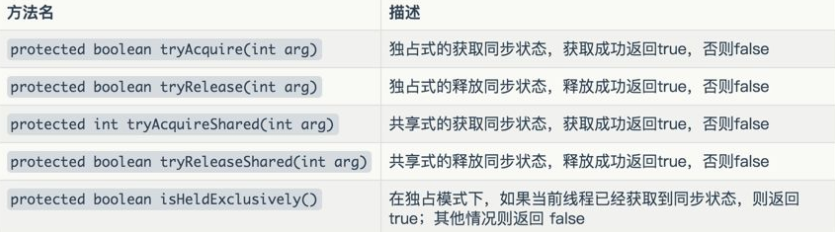
XXXXXXXXXXXXXXXXXXXXXXXXXX

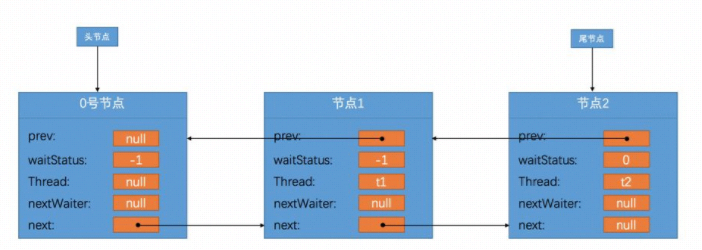
### 请讲一讲AQS的原理？

[AQS原理1](https://mp.weixin.qq.com/s?__biz=MzIxNTQ3NDMzMw==&mid=2247483910&idx=1&sn=f42489db76a6a1d7ff3a2ae48c2e0754&scene=19" \l "wechat_redirect)

[AQS原理2](https://mp.weixin.qq.com/s?__biz=MzIxNTQ3NDMzMw==&mid=2247483917&idx=1&sn=6d074607603149b7e38b33a7bed7f417&scene=19" \l "wechat_redirect)

简单来说AQS(抽象队列同步器)维护了一个同步状态state，然后提供了几个**获取同步状态的方法**用来判断同步状态的状态(**当然这个同步分成独占式和共享式的，区别就是是否支持state的多次获取，例如如果是独占式的可以判断state=0或者1，共享式的可以判断state=10，每次一个线程获取一个就减1，直到为0才阻塞**)，假设一个线程未获得了同步状态，则将线程加入到一个同步队列中

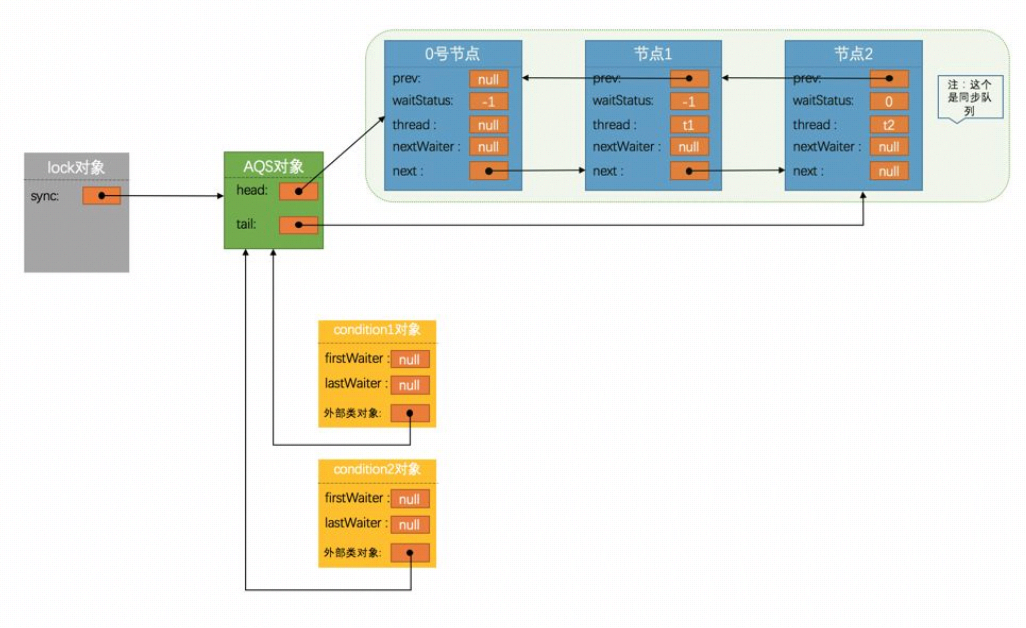




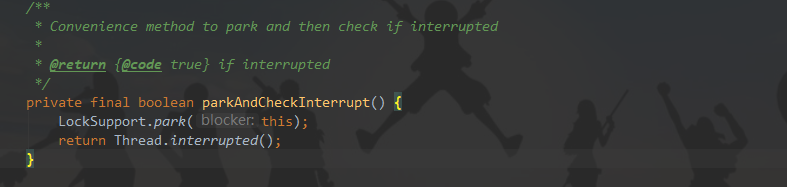
#### AQS共享模式和独占模式的不同之处

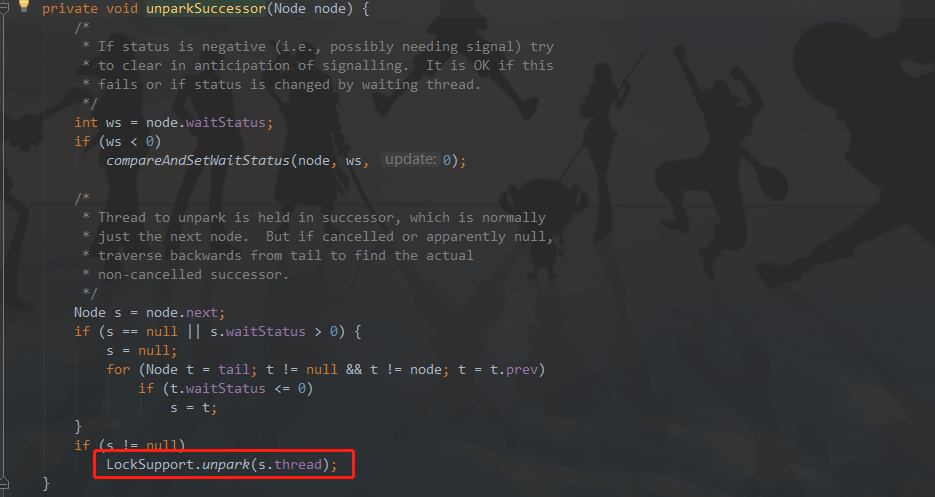
**共享模式与独占模式不同的一点是，可能同时会有多个线程释释放同步状态，也就是可能多个线程会同时移除同步队列中的阻塞节点，哈哈，如何保证移除过程的安全性？只能看源码**

### AQS底层是如何实现阻塞和释放线程的?



如图AQS底层维护着阻塞的队列，当一个线程加入到队列中时，会调用LockSupport.park(this)将当前线程阻塞。当获取到锁的线程执行完后，调用解锁时会调用LockSupport.unpark(s.thread)将队列中的第一个唤醒解锁。





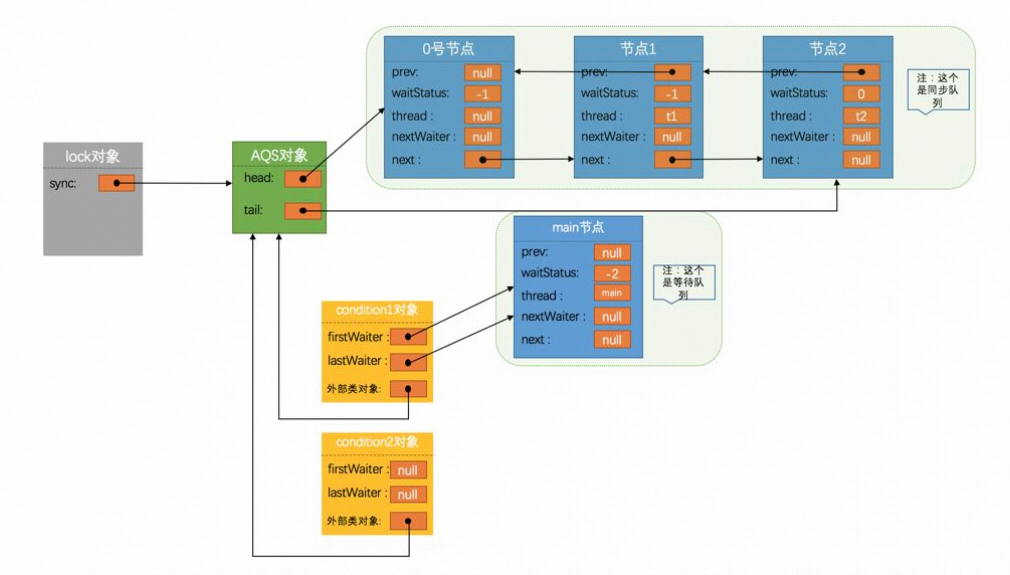
### 线程之间通信有几种方法?

#### 内置锁wait/notify

Xxx

#### 显式锁Condition

显式锁的本质其实是通过AQS对象获取和释放同步状态



整个逻辑流程如下: 看具体的ULR教程

### Synchronized和ReentrantLock区别

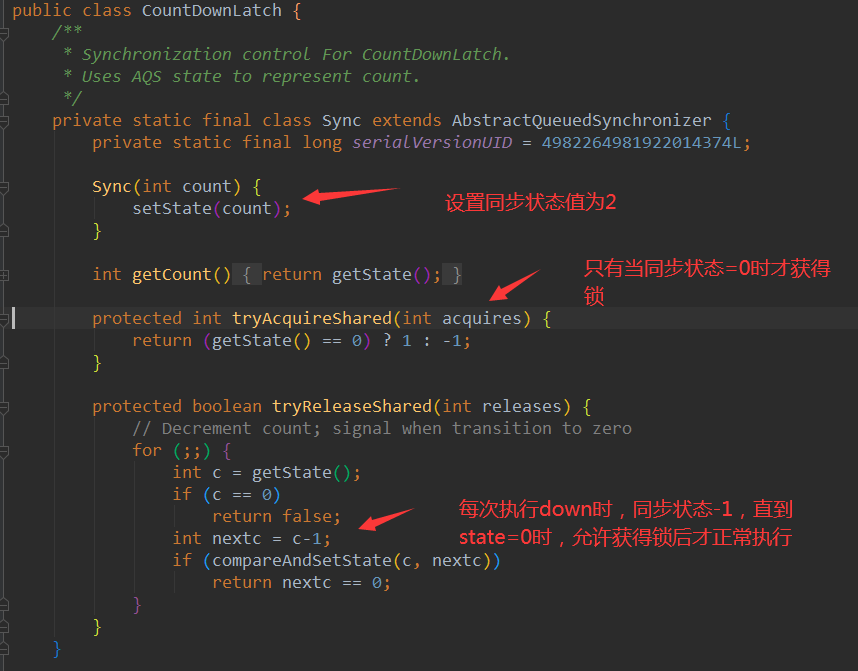
1. 加锁方式不同，一个是通过对对象头加锁，一个采用AQS(抽象同步队列)的方式加锁
2. 两者都是可重入锁
3. ReentrantLock提供了可中断锁，有超时时间
4. 前者遇到报错自动解锁，后者需要手动在finally上解锁

### CountDownLatch有使用过吗，什么原理

#### 原理

countDaownLatch底层是使用AQS(抽象队列同步器)的共享锁方式，首先我们通过构造函数设置了同步状态(state)值=2，而CountDownLatch重写了获取同步状态的方法，当state=0时才允许获得锁，这样当我们state不等于0时，执行await时，对应的线程被加入到同步队列中阻塞，当执行countDown后会调用AQS的releaseShared方法释放同步状态，当State减少到0时开始将同步队列中的线程一个一个释放





#### 注意点

1. 由于上面的机制，CountDownLatch允许一个或者多个线程等待直到其他线程完成，只能用一次
2. 如果多个线程都设置await，down到0时，所有await同时释放，结果是会多个线程同时并发执行，如果多个线程里面设置了不安全的并发操作会有问题。(因为采用的是AQS的共享模式，所以会有同时把移出队列，所以会有安全性问题)

|  |
| --- |
| public class CountDownLatchTest {  public static void main(String[] args) throws InterruptedException {  int NUMBER = 100;  Map<String, String> myMap = Maps.*newHashMap*();  CountDownLatch downLatch = new CountDownLatch(2);  new Thread(() -> {  try {  TimeUnit.*SECONDS*.sleep(2);  downLatch.countDown();  } catch (InterruptedException e) {  e.printStackTrace();  }  }).start();  new Thread(() -> {  try {  TimeUnit.*SECONDS*.sleep(5);  downLatch.countDown();  } catch (InterruptedException e) {  e.printStackTrace();  }  }).start();   for (int i = 0; i < NUMBER; i++) {  new Thread(new Runner(downLatch,myMap)).start();  }  System.*out*.println("线程设置成功");  TimeUnit.*SECONDS*.sleep(10);  //有并发问题，这里的myMap的数量不对  System.*out*.println(myMap.entrySet().size());   }   static class Runner implements Runnable {  private CountDownLatch countDownLatch;  private Map<String, String> myMap;   public Runner(CountDownLatch countDownLatch, Map<String, String> myMap) {  this.countDownLatch = countDownLatch;  this.myMap = myMap;  }   @Override  public void run() {  try {  countDownLatch.await();  } catch (InterruptedException e) {  e.printStackTrace();  }  for (int j = 0; j < 1000; j++) {  myMap.put(UUID.*randomUUID*().toString(), "1");  }  }  } } |

### CyclicBarrier原理和使用

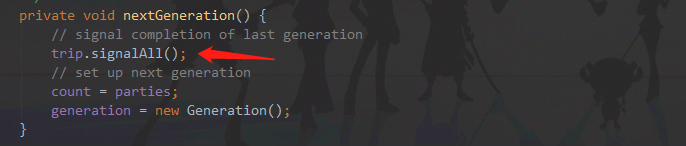
底层采用ReentrantLock和Condition实现。首先先初始化3个等待



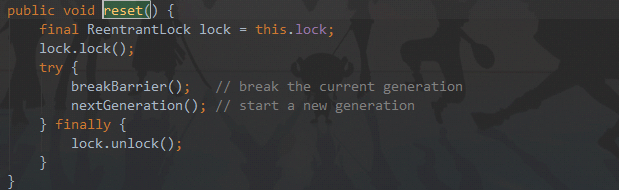
每次执行次数-1，没到0时，通过Condition阻塞

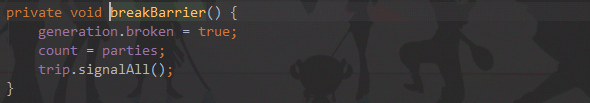


到0时，通过执行condition的sigalAll方法通知



**由于是采用Condition实现的，所以可以重复利用**





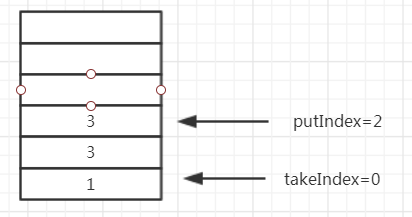
### 阻塞队列的几个方法区别？



### ArrayBlockingQueue阻塞队列的原理？

结构:

1. 一个由**数组**组成的**有界**的阻塞队列
2. 底层采用一个ReentrantLock + 两个Condition(阻塞队列满和队列空)
3. 定义了两个指针putIndex(指向数组尾部)，takeIndex(指向数组头部)



4、定义了一个count变量，每次新增元素加1，移除元素-1

面试注意点:

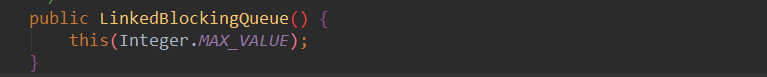
1. 由于是数组组成的有界队列，所以是无法扩容的
2. 入队和出队都加锁，且用同一个锁，效率较低
3. 当count等于数组最大时禁止再入队，当count=0时禁止再出队
4. 数组其实是个循环数组，当入队满时，会重头开始入队，当然前提是消费者有消费了

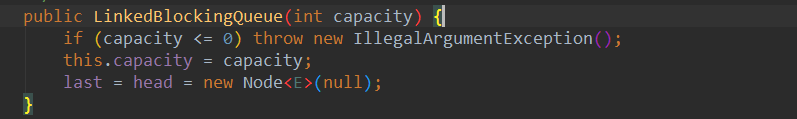


### LinkedBlockingQueue阻塞队列的原理？

结构:

1. 底层采用**单链表**存储结构
2. 有界的，因为可以传容量，但是默认是无界的，因为容量默认是Integer.MAX\_VALUE





1. 采用两个ReentrantLock + 两个Condition 来控制出入队列

面试注意点:

1. 采用两把锁的锁分离技术实现入队出队互不阻塞
2. 是有界队列，不传入容量时默认为最大int值

### LinkedBlockingQueue与ArrayBlockingQueue对比?

后者入队出队采用一把锁，导致入队出队相互阻塞，效率低下

前者入队出队采用两把锁，入队出队互不干扰，效率较高

二者都是有界队列，如果长度相等且出队速度跟不上入队速度，都会导致大量线程阻塞

前者如果初始化不传入初始容量，则使用最大int值，如果出队速度跟不上入队速度，会导致队列特别长，占用大量内存

### PriorityBlockingQueue特点有什么?

1. 无界队列，底层采用数组，默认长度是11，当不够时会扩容
2. 采用一个ReentrantLock + 一个Condition进行加锁 ArrayBlockingQueue是两个Condition，因为PriorityBlockingQueue是无界，所以只要有一个用来判定就行
3. 出队和入队用同一把锁

### ConcurrentLinkedQueue特点有什么?

1. 无界，非阻塞，并发队列
2. 内部采用CAS来保证并发安全
3. 采用FIFO(先进先出)原则对元素进行排序
4. 精妙之处：利用CAS来完成数据的操作，允许队列的不一致性，弱一致性表现得淋漓尽致
5. 判断队列长度不要用size，效率低

### ConcurrentLinkedQueue和LinkedBlockingQueue的用处和不同之处？

1. 前者是非阻塞队列，后者是阻塞队列
2. 前者无界，后者有界

## JAVA面试 -- JVM

### JVM的几个重要调优参数

#### 常见性能调优参数

##### 设置堆大小比较重要的一些参数

1、-server : 生产环境必须开启的一个配置

2、-Xms : 初始堆大小

3、-Xmx : 最大堆大小

4、-Xmn : 年轻代的大小(eden + 2 survivor,设置了这就不用设置-XX:NewSize，-XX:MaxNewSize)

5、-XX:NewRatio ： 年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代)，-XX:NewRatio=4表示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

6、-XX:SurvivorRatio ： eden区和survivor区的大小比值(=8,则两个surviro和eden的比值为2：8)

7、-XX:MaxTenuringThreshold：设置垃圾最大年龄，如果设置成0，则年轻代不经过survivor区，直接进入老年代

##### 垃圾收集器的配置--适合web类型的用

-XX:+UseParNewGC：年轻代使用PerNew收集器

-XX:+UseConcMarkSweepGC：老年代使用CMS收集器

-XX:+CMSParallelRemarkEnabled ：降低标记停顿

-XX:CMSInitiatingOccupancyFraction=70 ： 使用cms作为垃圾回收，使用70％后开始CMS收集

##### 日志类的配置

-XX:+PrintGCDetails：定义输出日志格式

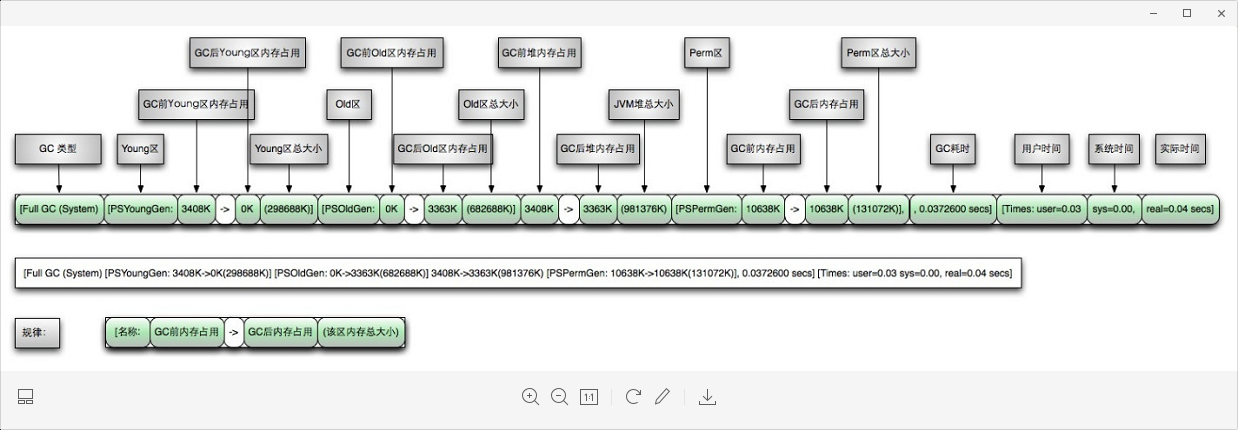


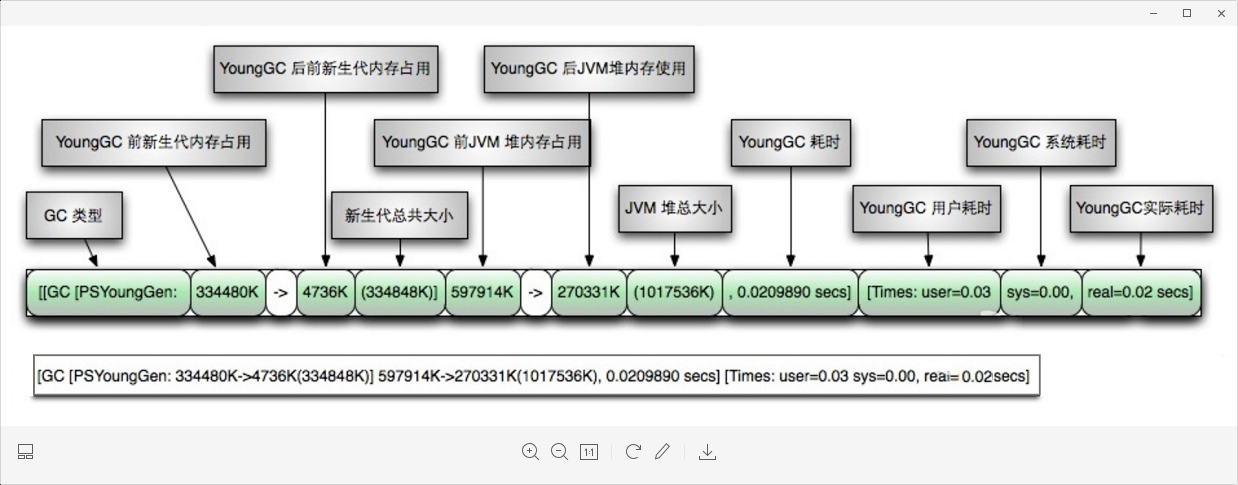
-XX:+PrintHeapAtGC：打印GC前后的详细堆栈信息

-verbose:gc ：类似于-XX:+PrintGCDetails可以认为一样，应该是早起是用这个后面换了

-Xloggc:filename ： 指定GC输出的路径

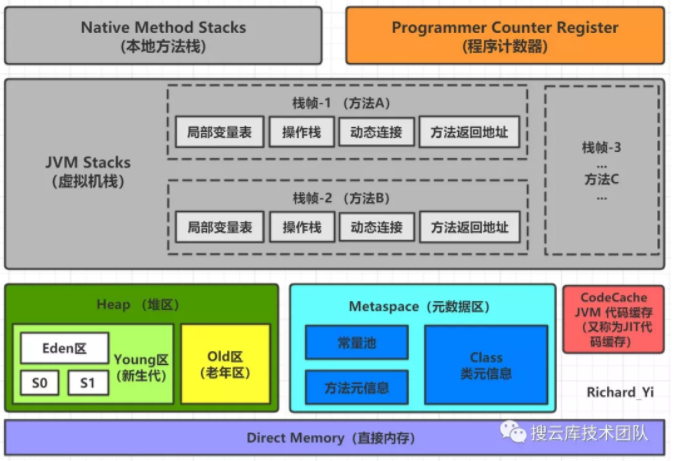
-XX:+PrintGCDateStamps：输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）





### JVM内存模型

程序计数器、本地方法栈、虚拟机栈、堆、元数据



### JVM垃圾回收机制

#### 标记为垃圾的算法有哪些

1、引用计数法(有缺陷，一般不用)

2、可达性分析算法：当一个对象到GC Root没有任何引用链的话，就证明不可用了

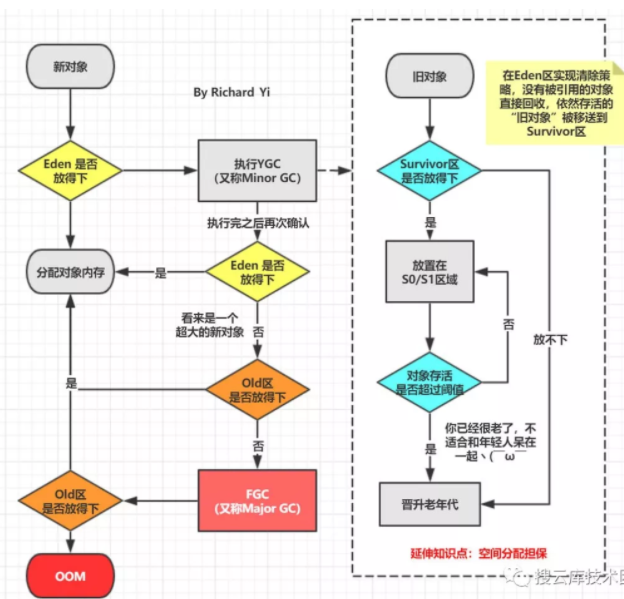
##### 什么对象可以作为GCROOT？

1. 虚拟机栈中的引用对象
2. 方法区中的常量引用对象(可以理解为final修饰的吗？)
3. 静态属性引用的对象
4. 本地方法栈引用的对象(这个没怎么见过)

#### 垃圾回收的算法有哪些?

1. 标记清除算法 ：老年代使用的算法，缺点效率低，碎片化
2. 复制算法：新生代用的算法，缺点是要占用一半的内存，所以新生代有两个幸存区
3. 标记整理算法：老年代使用的算法
4. 分代回收算法：就是老年代使用标记清除或者标记整理算法，新生代使用复制算法

### 描述一下GC的过程?



### JAVA类加载机制，类加载过程，什么是双亲委派？

加载过程包括：加载-->验证-->准备-->解析-->初始化

双亲委派：当要加载一个类的时候先委派个父加载器去加载，如果父加载器没有才自己加载。这样的好处是保护类，假如扩展加载器负责加载lib/ext下面的类，如果自己写了一个类跟他一样的名字，让后应用程序加载器去加载的话会覆盖掉lib/ext的类。

### GC收集器有哪些？CMS收集器和G1收集器的特点?

|  |  |
| --- | --- |
| 年轻代 | 老年代 |
| Serial | Serial Old |
| PawNew | CMS |
| Parallel Scaveenge | Paralled Old |

CMS是一种以获取最短回收停顿为目标的收集器，主要优点：并发收集，底停顿

### 常见的JVM有哪些？

##### java.lang.OutOfMemoryError: Java heap space

创建对象时，堆内存空间的不足以存放新创建的对象时报错

堆内存使用量达到了最大内存限制时会报错

##### java.lang.OutOfMemoryError: GC overhead limit exceeded

主要出现这个问题是因为执行垃圾收集的时间比例太大，有效的运算量太小

该错误只在连续多次 GC 都只回收了不到2%的极端情况下才会抛出

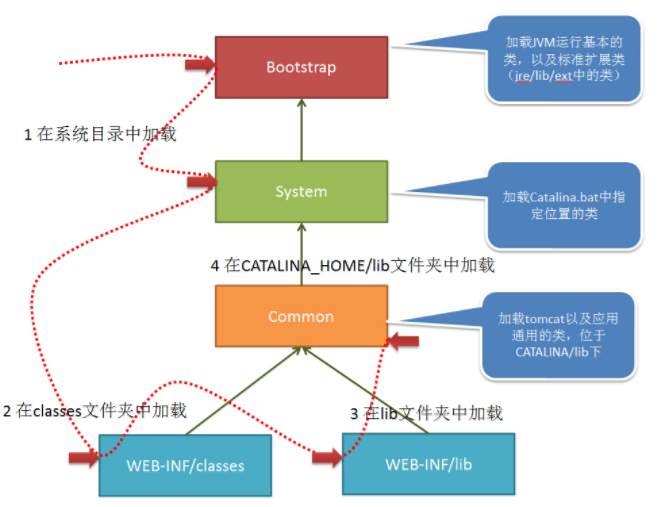
##### java.lang.OutOfMemoryError: Metaspace

主要原因, 是加载到内存中的 class 数量太多或者体积太大

## Tomcat专题

### Tomcat的类加载器流程

https://www.cnblogs.com/aspirant/p/8991830.html



当应用需要到某个类时，则会按照下面的顺序进行类加载：

1. 使用bootstrap引导类加载器加载
2. 使用system系统类加载器加载
3. 使用应用类加载器在WEB-INF/classes中加载
4. 使用应用类加载器在WEB-INF/lib中加载
5. 使用common类加载器在CATALINA\_HOME/lib中加载

通过对上面tomcat类加载机制的理解，就不难明白 为什么java文件放在Eclipse中的src文件夹下会优先jar包中的class?

这是因为Eclipse中的src文件夹中的文件java以及webContent中的JSP都会在tomcat启动时，被编译成class文件放在 WEB-INF/class 中。

### Tomcat支持的线程模型

1. BIO : 阻塞式IO，应该没人用了
2. NIO : 同步非阻塞，tomcat8.0默认采用该模式
3. APR : 简单理解就是从操作系统级别解决异步IO问题，能大幅度提升服务器的处理和响应性能，也是tomcat运行高并发应用的首选
4. AIO : 异步非阻塞, tomcat8.0及其以后才支持

### Tomcat如何调优？涉及到哪些参数

一般tomcat的调优包括tomcat的参数和JVM的参数，JVM参数就不说了

1. maxThreads=“300”：设置当前Tomcat的最大并发数，即同时能支持多少个并发
2. minSpareThreads=“50”：设置当前Tomcat初始化时创建的线程数
3. acceptCount=“250”，当同时连接的数超过maxThreads，还可以接受排队的数量，超过了就报错
4. compression=”on”，给tomcat开启gzip压缩功能
5. compressableMimeType=”text/html,text/xml”　压缩类型

注意:

1. 这里的线程数和并发数需要理解一下，不是说多少个线程就多少个并发，并发一般是需要有个时间单位的，例如一秒内多少并发，假设一个请求消耗100ms,那如果有100个线程，则100个请求在100ms就处理完了，那这里的并发应该是1秒内1000个并发，也就是1000QPS

## Redis

### redis为什么那么快?

1. 数据存内存里面，运算都是内存级别的运算，操作快
2. 采用了一种直观的文本协议，优势在于实现简单，解析性能好



1. 采用事件轮询的多路复用，就是类似JAVA里面的NIO
2. 单线程了，少了线程之间新建销毁的时间
3. redis 对底层的数据类型做了优化。例如，

Redis 的字符串有两种存储方式，在长度特别短时，使用 emb 形式存储 (embeded)，当长度超过 44 时，使用 raw 形式存储。

大字典hash扩容也做了优化，如果是重新申请数组然后将旧字典所有链表中的元素重新挂接到新的数组下面很耗时，所以大字典扩容用了rehash小补搬迁。

zset 和 hash 容器对象在元素个数较少的时候，采用压缩列表 (ziplist) 进行存储。

List列表采用了quicklist

（关于内部结构优化的后面写）