

异常 Exception

- **Throwable**是所有Java程序中错误处理的父类，有两种子类：**Error**和**Exception**。
 - **Error**：表示由JVM所侦测到的无法预期的错误，由于这是属于JVM层次的严重错误，导致JVM无法继续执行，因此，这是不可捕捉到的，无法采取任何恢复的操作，顶多只能显示错误信息。比如：内存资源不足（`OutOfMemoryError`、`StackOverflowError`）等。对于这种错误，程序基本无能为力，除了退出运行外别无选择，它是由Java虚拟机抛出的。
 - **Exception**：表示可恢复的例外，这是可捕捉到的。
 - Java提供了两类主要的异常：**runtime exception**和**checked exception**。
 - **checked exception**也就是我们经常遇到的**IOException**异常，**FileNotFoundException**，以及**SQLException**异常、**InterruptedException**（线程sleep、wait）都是这种异常。对于这种异常，JAVA编译器强制要求我们必需对出现的这些异常进行catch。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆catch块去处理可能的异常。程序应该且仅应该抛出或处理已检查异常。
 - **runtime exception**，也称运行时异常，我们可以不处理。当出现这样的异常时，总是由虚拟机接管。比如：我们从来没有人去处理过**NullPointerException**异常，它就是运行时异常，并且这种异常还是最常见的异常之一。如果出现**RuntimeException**，那么一定是程序员的错误。
 - 出现运行时异常后，系统会把异常一直往上层抛，一直遇到处理代码。如果没有处理块，到最上层，如果是多线程就由**Thread.run()**抛出，如果是单线程就被**main()**抛出。抛出之后，如果是线程，这个线程也就退出了。如果是主程序抛出的异常，那么这整个程序也就退出了。运行时异常是**Exception**的子类，也有一般异常的特点，是可以被Catch块处理的。只不过往往我们不对他处理罢了。也就是说，你如果不对运行时异常进行处理，那么出现运行时异常之后，要么是线程中止，要么是主程序终止。
 - 覆盖父类某方法的子类方法不能抛出比父类方法更多的异常，所以，有时设计父类的方法时会声明抛出异常，但实际的实现方法的代码却并不抛出异常，这样做的目的就是为了方便子类方法覆盖父类方法时可以抛出异常。
-

try-catch-finally语句中如果**try**和**finally**中都有**return**语句，结果返回的是什么？

//try中的return先于finally执行，但是执行之后不立即返回，而是将返回值保存在返回栈中，等执行finally语句块中的

//内容后再返回

```
public static int[] test1(){//返回10
```

```
    int[] a = {1};//如果是常量也一样，主要判断finally中的修改是否会影响
```

```
    try {
```

```
        a[0] = 1;
```

```
        return a;
```

```
    } catch (Exception e) {
```

```
        return a;
```

```
    }finally {
```

```
        a[0] = 10;//a[]是一个数组，引用类型，此时finally语句块中的修改会影响已经保存下来的返回值
```

```
    }
```

```
}
```

```
public static int test2(){//返回1
```

```
    int a = 1;
```

```
    try {
```

```
        a = 1;
```

```
        return a;
```

```
    } catch (Exception e) {
```

```
        return a;
```

```
    }finally {
```

```
        a = 10;//a是一个int，基本类型，此时finally语句块中的修改不会影响已经保存下来的返回值
```

```
    }
```

```
}
```

```
public static int test3(){//返回10
```

```
    int a = 1;
```

```
    try {
```

```
        a = 1;
```

```
        return a;
```

```
    } catch (Exception e) {
```

```
        return a;
```

```
    }finally {
```

```
        a = 10;//a是一个int，基本类型，此时finally语句块中的修改不会影响已经保存下来的返回值，但是下一行的return
```

```
        return a;//语句会覆盖返回栈中的返回值
```

```
    }
```

```
}
```

同步异步 与 阻塞非阻塞的区别

- **同步**：当一个同步调用发出后，调用者要一直等待返回消息（或者调用结果）通知后，才能进行后续的执行；
- **异步**：当一个异步过程调用发出后，调用者不能立刻得到返回消息（结果）。实际处理这个调用的部件在完成后，通过消息回调来通知调用者是否调用成功。
- 调用者获取依赖服务异步回调结果一般有两种方式，一种是主动去轮训查询异步回调的结果，一种调用依赖服务时传入一个callback方法或者回调地址，依赖服务完成之后去调用callback通知调用者。
- **阻塞**：阻塞调用是指调用结果返回之前，当前线程会被挂起，一直处于等待消息通知，不能够执行其他业务。
- **非阻塞**：非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，可以去处理其他任务。
- 假如现在有两个请求分别1.读取A数据块再打印一个随机数，2.读取B数据块再打印一个随机数；如果是同步的那么必须等待A数据块读完才能打印随机数，如果是异步的那么可以先打印随机数，然后等待A数据块读完（异步一定是接下来的流程与异步流程不存在因果联系，并且异步一般通过一个其他线程处理，不是在本线程处理）。如果在读取A数据块的过程中，A为空，如果是阻塞，那么该线程一直等待A数据块有数据，如果是非阻塞那么该线程可以先处理B数据块的读操作。
- 所以同步异步的区别主要在于同步一直都是一个主线程，异步会创建多个子线程去执行异步任务，而主线程继续往下执行。
- 买冰激凌：
 - 同步阻塞：自己买，且一直等待冰激凌做好
 - 同步非阻塞：自己买，等待过程中可以做其他事
 - 异步阻塞：让同学买，自己接着做其他事，且同学一直等待冰激凌做好，不能做其他事
 - 异步非阻塞：让同学买，自己接着做其他事，且同学等待过程中可以做其他事

抽象类与接口的区别？

1. 接口可以多继承，抽象类不行
 2. 接口定义方法，不能实现，而抽象类可以实现部分方法。
 3. 接口中基本数据类型为static 而抽象类不是的。
 4. （相同点）都不能被实例化
-

为什么Java类只能单继承？接口与接口之间可以多继承？

- 在Java语言中禁止多重继承：一个类可以具有多个直接父类。多重继承不合法的原因是容易引发意义不明确。例如，有一个类C，如果允许它同时继承A类与B类（`class C extends A,B{}`），假如A、B两个类都有同一种方法`fun()`，如果定义：`C c = new C();`那么`c.fun()`应该调用哪一个父类的`fun()`方法？无法给出答案，这就是多继承的菱形继承问题，因此Java语言禁止多重继承。
 - Java接口是行为性的，也就是说它只是定义某个行为的名称，而具体的行为的实现是集成接口的类实现的，因此就算两个接口中定义了两个名称完全相同的方法，当某个类去集成这两个接口时，类中也只会会有一个相应的方法，这个方法的具体实现是这个类来进行编写的，所以并不会出现结构混乱的情况。
-

Java中基本数据类型和包装类型有什么区别？

1. 包装类是对象，拥有方法和字段
 2. 包装类型是引用的传递，基本类型是值的传递
 3. 声明方式不同，基本数据类型不需要new关键字，而包装类型需要new在堆内存中进行new来分配内存空间
 4. 存储位置不同，基本数据类型直接将值保存在值栈中，而包装类型是把对象放在堆中，然后通过对象的引用来调用他们
 5. 初始值不同，eg: `int`的初始值为 0 、 `boolean`的初始值为false 而包装类型的初始值为null
 6. 使用方式不同
-

面向对象开发的六个基本原则

1. 单一职责：一个类只做它该做的事情(高内聚)。在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。
2. 开放封闭：软件实体应当对扩展开放，对修改关闭。要做到开闭有两个要点：
①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而混乱。
3. 里氏替换：任何时候都可以用子类型替换掉父类型。子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。
4. 依赖倒置：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代）
5. 合成聚和复用：优先使用聚合或合成关系复用代码。
6. 接口隔离：接口要小而专，绝不能大而全。臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。

迪米特法则

迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。

Java中的集合有哪些？

Set、Map、List

Java面向对象的三大特征

继承、封装、多态

- 封装
 - 1.概念：将类的某些特征隐藏在类的内部，不允许外部程序直接访问，而是通过该类提供的方法来实现对隐藏信息的操作和访问。
 - 2.好处
 - a.只能通过规定的方法访问数据；
 - b.隐藏类的实现细节，方便修改和实现；
 - c.高内聚，低耦合。高内聚：类的内部数据操作细节自己完成，不允许外部干涉；低耦合：仅暴露少量的方法给外部使用。
 - 继承
 - 继承是从已有的类中派生出新的类，新的类能吸收已有类的数据属性和行为，并能扩展新的能力。JAVA不支持多继承。
 - 多态
 - 所谓多态就是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。
 - 最常见的多态就是将子类传入父类参数中，运行时调用父类方法时通过传入的子类决定具体的内部结构或行为。
-

重载和重写的区别

- **重载**：Java的方法重载，就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。**重载方法参数必须不同**。调用方法时通过传递给它们的不同参数个数和参数类型给它们的不同参数个数和参数类型给它们的不同参数个数和参数类型来决定具体使用哪个方法，这就是**多态性**。
- **重写**：
 - 父类与子类之间的多态性，对父类的函数进行重新定义。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写（Overriding）。在Java中，子类可继承父类中的方法，而不需要重新编写相同的方法。但有时子类并不想原封不动地继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。方法重写又称方法覆盖。
 - 若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法。如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。
 - 子类函数的访问修饰权限不能少于父类的；**重写必须有相同的方法名，参数列表和返回类型**。重写的方法作用域必须大于等于原方法。重写方法不能抛出新的异常或者比被重写方法声明的检查异常更广的检查异常。但是可以抛出更少，更有限或者不抛出异常。

Java中Vector和ArrayList的区别

Vector相对于ArrayList类似于Hashtable相对于HashMap。Vector(Hashtable)是线程安全的，但是效率低。ArrayList(HashMap)可以包装为线程安全的。多线程中可以用CopyOnWriteArrayList(ConcurrentHashMap)直接代替。

- 首先这两类都实现List接口，而List接口一共有三个实现类，分别是ArrayList、Vector和LinkedList。
- Vector与ArrayList一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢。
- 如果集合中的元素的数目大于目前集合数组的长度时，vector增长率为目前数组长度的100%,而arraylist增长率为目前数组长度的50%。
- ArrayList同样可以通过Collections.synchronizedList包装为线程安全类。

CopyOnWrite容器、CopyOnWriteArrayList

- CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元

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

- CopyOnWriteArrayList的实现原理

- 在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。

```
public boolean add(T e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {

        Object[] elements = getArray();
        int len = elements.length;
        // 复制出新数组
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        // 把新元素添加到新数组里
        newElements[len] = e;
        // 把原数组引用指向新数组
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}

final void setArray(Object[] a) {
    array = a;
}
```

- 读的时候不需要加锁，如果读的时候有多个线程正在向ArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的ArrayList。

- CopyOnWrite的缺点

- 内存占用问题。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，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进

去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。

- 数据一致性问题。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的数据，马上能读到，请不要使用CopyOnWrite容器。

HashMap为什么是线程不安全的？

- 在hashmap做put操作的时候会调用到addEntry的方法。现在假如A线程和B线程同时对同一个数组位置调用addEntry，两个线程会同时得到现在的头结点，然后A写入新的头结点之后，B也写入新的头结点，那B的写入操作就会覆盖A的写入操作造成A的写入操作丢失。

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

- 如果多个线程同时检测到元素个数超过数组大小*loadFactor，这样就会发生多个线程同时对Node数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给table，也就是说其他线程的都会丢失，并且各自线程put的数据也丢失。

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

    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}
```

- 底层table扩容时还可能形成环状链表，造成死循环。

CAS(UnSafe包)

- CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。
- CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。
- CAS是一条CPU的原子指令，其实现方式是基于硬件平台的汇编指令，就是说CAS是靠硬件实现的。
- 在这里采用了CAS操作，每次从内存中读取数据然后将此数据和+1后的结果进行CAS操作，如果成功就返回结果，否则重试直到成功为止。

```
public final int incrementAndGet() {
    for (;;) {// 无限循环直到成功
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))// 相同返回true, 不同返回false
            return next;
        }
    }

    public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect,
            update);
    }
}
```

- CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。
 - **ABA问题**。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加版本号，每次变量更新的时候把版本号加一，那么A-B-A 就会变成1A-2B-3A。从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。
 - **循环时间长开销大**。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline），使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。
 - **只能保证一个共享变量的原子操作**。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i=2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。
-

Collections.SynchronizedMap底层实现原理

- Collections.synchronizedMap()实现原理是Collections定义了一个SynchronizedMap的内部类，并返回这个类的实例。
- SynchronizedMap这个类实现了Map接口，在调用方法时使用synchronized来保证线程同步,当然了实际上操作的还是我们传入的HashMap实例，简单的说就是Collections.synchronizedMap()方法帮我们在操作HashMap时自动添加了synchronized来实现线程同步，类似的其它Collections.synchronizedXX方法也是类似原理）Mutex在构造时默认赋值为this，即所有方法都用的同一个锁。

```

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {
    return new SynchronizedMap<>(m);
}
final Object      mutex;
public V get(Object key) {
    synchronized (mutex) {return m.get(key);}
}

public V put(K key, V value) {
    synchronized (mutex) {return m.put(key, value);}
}

```

HashMap如何变线程安全，每种方式的优缺点？

- Hashtable
- ConcurrentHashMap
 - 完全允许多个读操作并发进行，读操作并不需要加锁。
 - 使用分段锁技术，将hash表分为16段（默认值），诸如get,put,remove等常用操作只锁当前需要用到的段，只有在求size()和containsValue()等操作时才需要锁定整个表。
- SynchronizedMap
 - Hashtable的后继者HashMap是作为JDK1.2中的集合框架的一部分出现的，它通过提供一个不同步的基类和一个同步的包装器Collections.synchronizedMap，解决了线程安全性问题。通过将基本的功能从线程安全性中分离开来，Collections.synchronizedMap允许需要同步的用户可以拥有同步，而不需要同步的用户则不必为同步付出代价。

```

Map<Object, Object> m = Collections.synchronizedMap(new
HashMap<>());

```

HashTable和HashMap区别？

- 继承的父类不同：Hashtable继承自Dictionary类，而HashMap继承自AbstractMap类。但二者都实现了Map接口。
- 线程安全性不同：Hashtable 中的方法是**Synchronize**的，而HashMap中的方法在缺省情况下是非Synchronize的。在多线程并发的环境下，可以直接使用Hashtable，不需要自己为它的方法实现同步，但使用HashMap时就必须要自己增加同步处理。
- **key和value是否允许null值**：Hashtable中，key和value都不允许出现null值。但是如果在Hashtable中有类似put(null,null)的操作，编译同样可以通过，因为key和value都是Object类型，但运行时会抛出NullPointerException异常，这是JDK的规范规定的。HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。当get()方法返回null值时，可能是HashMap中没有该键，也可能使该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键，而应该用containsKey()方法来判断。
- 内部实现使用的数组初始化和扩容方式不同： Hashtable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。Hashtable扩容时，将容量变为原来的2倍加1，因为哈希表的大小为素数时，简单的取模哈希的结果会更加均匀。而HashMap扩容时，将容量变为原来的2倍，因为可以用位运算，加快hash速度。

ConcurrentHashMap底层实现原理

<http://ifeve.com/concurrenthashmap/>

http://blog.csdn.net/dingji_ping/article/details/51005799

- ConcurrentHashMap主要有三大结构：整个Hash表，**segment（段）**，**HashEntry（节点）**。每个segment就相当于一个HashTable。**ConcurrentHashMap**将锁加在**segment**上（每个段上），这样我们在对segment1操作的时候，同时也可以对segment2中的数据操作，这样效率就会高很多。
- **Segment**类继承于**ReentrantLock** 类，从而使得 **Segment** 对象能充当锁的角色。**Put**和**remove**方法中有**lock()**和**unlock()**（都是使用的**this**对象，**lock()**在代码开始，**unlock**在**finally**中）。
- Segment的结构和HashMap类似，是一种数组和链表结构，一个**Segment**里包含一个**HashEntry**数组，每个HashEntry是一个链表结构的元素。
- Hashtable容器的get方法是需要加锁的，那么ConcurrentHashMap的get操作

是如何做到不加锁的呢？原因是它的get方法里将要使用的共享变量都定义成volatile，之所以不会读到过期的值，是根据java内存模型的happen before原则，对volatile字段的写入操作先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值。

- 如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。
- 其中有一个Segment数组，每个Segment中都有一个锁，因此Segment相当于一个多线程安全的HashMap，采用分段加锁。每个Segment中有一个Entry数组，Entry中成员value是volatile修饰，其他成员通过final修饰。get操作不用加锁，put和remove操作需要加锁，因为value通过volatile保证可见性。
- 两个hash过程，第一次找到所在的桶，并将桶锁定，第二次执行写操作。而读操作不加锁
- Collections.SynchronizedMap和Hashtable都是整个表的锁，与ConcurrentHashMap锁粒度不同
- ConcurrentHashMap不允许key或value为null值。
- ConcurrentHashMap和Hashtable都是支持并发的，这样会有一个问题，当你通过get(k)获取对应的value时，如果获取到的是null时，你无法判断，它是put(k,v)的时候value为null，还是这个key从来没有做过映射。HashMap是非并发的，可以通过contains(key)来做这个判断。而支持并发的Map在调用m.contains(key)和m.get(key),m可能已经不同了，contains应该是通过遍历来查找，多线程时，可能会有其他线程在遍历过程中修改。
- ConcurrentHashMap和Hashtable都不允许key和value为null，Collections.synchronizedMap和HashMap的key和value都可以为null（因为就是包装了hashmap），TreeMap的key不可为空（非线程安全，需要排序），value可以。
- ConcurrentHashMap允许一边更新、一边遍历，也就是说在Iterator对象遍历的时候，ConcurrentHashMap也可以进行remove,put操作，且遍历的数据会随着remove,put操作产出变化，相当于有多个线程在操作同一个map（可以在foreach keyset时remove对象，HashMap不可以）

ConcurrentHashMap jdk7和jdk8的区别？

- Java 7基于分段锁的ConcurrentHashMap
 - Java 7 中的 ConcurrentHashMap 的底层数据结构仍然是数组和链表。



- 寻址方式：在读写某个Key时，先取该Key的哈希值。并将哈希值的高N位对Segment个数取模从而得到该Key应该属于哪个Segment，接着如同操作HashMap一样操作这个Segment。
- 同步方式：Segment继承自ReentrantLock，所以我们可以很方便的对每一个Segment上锁。
对于写操作，并不要求同时获取所有Segment的锁，因为那样相当于锁住了整个Map。它会先获取该Key-Value对所在的Segment的锁，获取成功后就可以像操作一个普通的HashMap一样操作该Segment，并保证该Segment的安全性。同时由于其它Segment的锁并未被获取，因此理论上可支持concurrencyLevel（等于Segment的个数）个线程安全的并发读写。

• Java 8基于CAS的ConcurrentHashMap

- 底层依然由“数组”+链表+红黑树的方式思想，大于8个转换为红黑树。默认初始大小16，负载因子也是0.75，定位元素的方法也是先hashCode(),再无符号右移16位异或，再(n-1)&hash
- 取消segments字段，直接采用transient volatile Node<K,V>[] table;保存数据，采用table数组元素作为锁，从而实现了的对每一行数据进行加锁，进一步减少并发冲突的概率。
- put函数流程：
 - 判断put进来的key和value是否为null，如果为null抛异常。（ConcurrentHashMap的key、value不能为null）。
 - 随后进入无限循环(没有判断条件的for循环)，何时插入成功，何时退出。
 - 在无限循环中，若table数组为空（底层数组加链表），则调用initTable()，初始化table；
 - 若table不为空，先hashCode(),再无符号右移16位异或，再(n-1)&hash，定位到table中的位置，如果该位置为空（说明还没有发生哈希冲突），则使用CAS将新的节点放入table中。
 - 如果该位置不为空，且该节点的hash值为MOVED（即为forward节点，哈希值为-1，其中含有指向nextTable的指针，class ForwardingNode中有nexttable变量），说明此时正在扩容，且该节点已经扩容完毕，如果还有剩余任务（任务没分配完）该线程执行helpTransfer方法，帮助其他线程完成扩容，如果已经没有剩余任务，则该线程可以直接操作新数组nextTable进行put。
 - 如果该位置不为空，且该节点不是forward节点。对桶中的第一个结点

（即table表中的结点，哈希值相同的链表的第一个节点）进行加锁（锁是该结点，如果此时还有其他线程想来put，会阻塞）（如果不加锁，可能在遍历链表的过程中，又有其他线程放进来一个相同的元素，但此时我已经遍历过，发现没有相同的，这样就会产生两个相同的），对该桶进行遍历，桶中的结点的hash值与key值与给定的hash值和key值相等，则根据标识选择是否进行更新操作（用给定的value值替换该结点的value值），若遍历完桶仍没有找到hash值与key值和指定的hash值与key值相等的结点，则直接新生一个结点并赋值为之前最后一个结点的下一个结点。

- 扩容transfer()函数流程：
 - 构建一个nextTable,它的容量是原来的两倍，这个操作是单线程完成的。
 - 将原来table中的元素复制到nextTable中，这里允许多线程进行操作。
- get函数流程：
 - 根据k计算出hash值，找到对应的数组index
 - 如果该index位置无元素则直接返回null
 - 如果该index位置有元素，如果第一个元素的hash值小于0，则该节点可能为ForwardingNode或者红黑树节点TreeBin。如果是ForwardingNode（表示当前正在进行扩容，且已经扩容完成），使用新的数组来进行查找。如果是红黑树节点TreeBin，使用红黑树的查找方式来进行查找。
 - 如果第一个元素的hash大于等于0，则为链表结构，依次遍历即可找到对应的元素，也就是读的时候不会加锁，同时有put，不会阻塞。读不加锁是因为使用了volatile（用在transient volatile Node<K,V>[] table），happens-before，先把所有写线程执行完再执行读线程。

Arraylist的原理？

以数组实现。节约空间，但数组有容量限制。超出限制时会增加50%容量，用System.arraycopy() 复制到新的数组，因此最好能给出数组大小的预估值。默认第一次插入元素时创建大小为10的数组。按数组下标访问元素—get(i)/set(i,e) 的性能很高，这是数组的基本优势。直接在数组末尾加入元素—add(e)的性能也高，但如果按下标插入、删除元素—add(i,e), remove(i), remove(e)，则要用System.arraycopy()来移动部分受影响的元素，性能就变差了，这是基本劣势。也就是说，当增加数据的时候，如果ArrayList的大小已经不满足需求时，那么就将数组变为原长度的1.5倍，之后的操作就是把老的数组拷到新的数组里面。

```

private static final int DEFAULT_CAPACITY = 10;
private int size;
public int size() {
    return size;
}
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }

    ensureExplicitCapacity(minCapacity);
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}

private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    // 扩展为原来的1.5倍
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    // 如果扩为1.5倍还不满足需求，直接扩为需求值
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}

```

LinkedList工作原理及实现


以双向链表实现。链表无容量限制，但双向链表本身使用了更多空间，也需要额外的链表指针操作。按下标访问元素—get(i)/set(i,e) 要悲剧的遍历链表将指针移动到位(如果i>数组大小的一半，会从末尾移起)。插入、删除元素时修改前后节点的指针即可，但还是要遍历部分链表的指针才能移动到下标所指的位置，只有在链表两头的操作—add(), addFirst(),removeLast()或用iterator()上的remove()能省掉指针的移动。

- LinkedList 是基于链表结构实现，所以在类中包含了 first 和 last 两个指针(Node)。Node 中包含了上一个节点和下一个节点的引用，这样就构成了双向的链表。每个 Node 只能知道自己的前一个节点和后一个节点，但对于链表来说，这已经足够了。

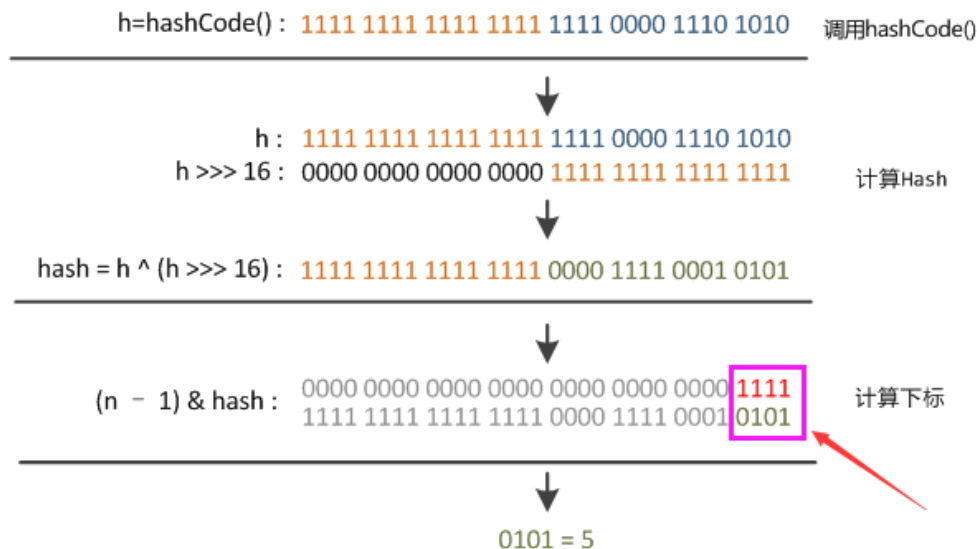
```
transient int size = 0;
transient Node<E> first; //链表的头指针
transient Node<E> last; //尾指针
//存储对象的结构 Node, LinkedList的内部类
private static class Node<E> {
    E item;
    Node<E> next; // 指向下一个节点
    Node<E> prev; //指向上一个节点

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

HashMap工作原理及实现？Hashmap为什么大小是2的幂次？Hashmap中jdk1.8之后做了哪些优化？

- HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。对于Hash冲突，HashMap采用了链地址法，也就是数组+链表的方式。数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的。
- 在常规构造器中，没有为数组table分配内存空间（有一个入参为指定Map的构造器例外），而是在执行put操作的时候才真正构建table数组
- 最终存储位置的确定流程：
- 为什么大小是2的整数次幂？ -> 因为这样可以用位运算来加快计算速度
- 为什么最后求index的时候用h & (length-1)，而不是直接取模？ -> 因为与运算速度更快

- 为什么中间还要再次计算hash()函数，也就是高16位与低16位的异或？ -> 因为table的长度都是2的幂，因此index仅与hash值的低n位有关，hash值的高位都被与操作置为0了。这样做很容易产生碰撞。所以将高16位与低16位异或来减少这种影响，使hash更加均匀。



- HashMap的大小必须是2的整数幂次方的原因就在于：主要目的是为了用位运算来加速，但这样会带来hash分布不均匀的问题，所以才中间增加了一步求h，目的就是打散数据，使hash尽可能均匀。
 - hash()函数，其返回值是`return h & (length-1)`; length为HashMap的长度。
 假设length = 50，则length- 1 = 49，转为二进制也就是110001，由于是与运算，所以返回值中间一定时3个0；这就相当于返回值只有8种可能；
 如果length= 32，则length- 1 = 31，转为二进制就是11111，则返回值有32种可能；
 所以2的整数次幂的长度能够让hash分布更加平均，不容易产生冲突。
 - 将哈希表的大小固定为了2的幂，可以运用位运算，加快hash速度。
- 在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能。

```

16  JAVA code:
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // 初始容量

    static final int MAXIMUM_CAPACITY = 1 << 30; // 最大容量
    static final float DEFAULT_LOAD_FACTOR = 0.75f; // 负载因子

    transient Node<K,V>[] table;
  
```

```

static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    public final K getKey()      { return key; }
    public final V getValue()    { return value; }
    public final String toString() { return key + "=" + value; }
}

public final int hashCode() {
    return Objects.hashCode(key) ^ Objects.hashCode(value);
}

public final V setValue(V newValue) {
    V oldValue = value;
    value = newValue;
    return oldValue;
}

public final boolean equals(Object o) {
    if (o == this)
        return true;
    if (o instanceof Map.Entry) {
        Map.Entry<?,?> e = (Map.Entry<?,?>)o;
        if (Objects.equals(key, e.getKey()) &&
            Objects.equals(value, e.getValue()))
            return true;
    }
    return false;
}

static final int hash(Object key) { //通过hashCode计算hash
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h
>>> 16);
}

static int indexFor(int h, int length) { //通过hash值确定最后
的下标索引

```

```
        return h & (length-1);  
    }  
}
```

HashSet原理

HashSet是基于HashMap来实现的，操作很简单，更像是对HashMap做了一次“封装”，而且只使用了HashMap的key来实现各种特性。

object类的方法?Object类所有的方法? 各自的实现原理及作用

● **public final native Class<?> getClass();**注意到**final**关键字，说明这个方法是不能被Object的子类重写的，我们在类中调用的getClass()方法都是调用的Object下面的，且文档对此的说明是返回运行时的对象。

● **public native int hashCode();**

○ 返回该对象的哈希值。

○ **native**说明跟机器有关，跟对象的地址有关。

○ 如果我们新建一个类，而hashCode没有被重写的话，那么hashCode返回的值只于对象的地址有关，如果hashCode被重写了，那么就另当别论了，但是如果我们重写了equals()的话，那么hashCode(),一定要重写。原因见[this.12](#)

● **public boolean equals(Object obj) {**
 return (this == obj);

} //如果equals()如果没有被重写的话，比较的是对象的地址。

● **protected native Object clone();** // 创建并返回此对象的一个副本。

● **public String toString();** // 返回该对象的字符串表示。

return getClass().getName() + "@" +

Integer.toHexString(hashCode());

}; // 可以看到返回的字符串是由对象的名称和对象的hashCode组成的

● **public final native void notify();**

○ 如果对象调用了该方法,就会随机通知某个正在等待该对象控制器的线程可以继续运行

● **public final native void notifyAll();**

○ 如果对象调用了该方法,就会通知所有正在等待该对象控制器的线程可以继续运行

● **public final native void wait(long timeout) throws**

InterruptedException;

○ 如果对象调用了该方法，就会使持有该对象的进程把对象的控制器交出去，然后处于等待状态

● **protected void finalize() throws Throwable { }**

○ 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。子类重写 finalize 方法，以配置系统资源或执行其他清除。

○ finalize 的常规协定是：当 Java™ 虚拟机已确定尚未终止的任何线程无法再通过任何方法访问此对象时，将调用此方法，除非由于准备终止的其他某个对象或类的终结操作执行了某个操作。finalize 方法可以采取任何操作，其中包括再次使此对象对其他线程可用；不过，finalize 的主要目的是在不可撤消地丢弃对象之前执行清除操作。例如，表示输入/输出连接的对象的 finalize 方法可执行显式 I/O 事务，以便在永久丢弃对象之前中断连接。

○ 总结一下就是：当某个对象被Java虚拟机回收的时候执行这个方法，如果我们想在这个对象被回收的时候做点什么，比如再次使此对象对其他线程可用，那么就需要我们重写这个方法。

Java的finalize, finally, final三个关键字的区别和应用场景？

- **finally**：作为异常处理的一部分，通常和try/catch结合使用，表示这个块中的语句一定会被执行，经常用于释放资源等。
 - **final**：用于对类、方法、域的声明。
 - final域：对于基本类型表示数值为常量不可变，如果为引用类型，表示引用的指向不变，但引用本身的内容可以改变。
 - final方法：禁止继承的导出类修改或者复写此方法。
 - final类：此类禁止继承。
 - final参数：表示这个参数在这个函数内部不允许被修改。
 - **finalize**：Object类中的方法，在垃圾回收器执行时会调被回收对象的finalize()方法，可以覆盖此方法用来实现对其他资源的回收。一旦垃圾回收器准备好释放对象占用的空间，将首先调用其finalize方法，并在下一次垃圾回收时才真正回收对象（两次标记）。
-

String类可以被继承吗？String，StringBuilder，StringBuffer三者的区别？

三个类都是final类，所以都不能被继承。

- String：适用于少量的字符串操作的情况，长度不可变。
 - StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况，性能比StringBuffer更好，但是线程不安全。
 - StringBuffer：线程安全的，在StringBuilder的方法上添加了synchronized修饰。适用多线程下在字符缓冲区进行大量操作的情况。
-

hashCode和equals？为什么要同时重写hashCode和equals？不同时重写会出现哪些问题？

- 如果没有覆盖，则会导致所有基于散列的集合无法正常工作
- JDK中对hashCode的常协规定：根据equals()对象相同的时候，hashCode()一定要相同，根据equals()对象不同的时候，hashCode()可以相同，可以不同。
 - 在 Java 应用程序执行期间，在对同一对象多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是将对象进行 equals 比较时所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。
 - 如果根据 equals(Object) 方法，两个对象是相等的，那么对这两个对象中的每个对象调用 hashCode 方法都必须生成相同的整数结果。
 - 如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么对这两个对象中的任一对象上调用 hashCode 方法不要求一定生成不同的整数结果。但是，程序员应该意识到，为不相等的对象生成不同整数结果可以提高哈希表的性能
- HashSet存放元素的原理：当向Hashset中增加元素的时候，首先计算元素的hashCode，根据hashCode得到一个位置来存放该元素，如果在该位置没有元素，那么集合认为该元素不存在与计划中，所以直接增加进去，如果该位置有一个元素，那么将要放进去的元素与该位置上面已经存在的元素进行equals比较，若返回false，则集合认为该对象不在集合中，可加入，然后因为该位置已经存在元素，所以进行一次散列，得到一个新的地址，若得到的地址中还是存在元素，那么一直进行散列，直到找到的地址中没有元素为止，然后把需要增加的元素放入该地址中，若true，则集合认为该元素已经存在与集合中，不会将对象增加到集合中。
- 如果现在有个关于Hash的集合HashSet或者HashMap的对象A,其生成的哈希码为Ahash，现在根据其哈希码将其放入集合中，假设进入Abucket；在某一时候，重新生成对象A，但是此时A对象的哈希码由于没有重写，其与存储地址相关，那么此时生成的哈希码为Bhash，如果在Bbucket中查找是否有A这个对象，那一定是不存在的；就算Bhash和Ahash碰巧分散到同一个bucket中，由于其散列码不相等，程序也就不会再比较对象的同等性，而直接返回不相等/不存在。实质上违反了hashCode的常协规定的第二条。

不同类型的对象的hashCode、equals的实现方法？

hashCode：基本类型的散列值一般和其自身相关，数值、集合的散列值往往是其包含的对象的散列值的递归累加求得，但其初始默认值有时会不同。String可以看成是char的数组。equals：基本类型一般先比较引用相等能否成立，再比较实际的值。数组、集合往往先比较引用地址，再递归调用对象自身的equals来比较。

- Integer、Character、byte、short: //直接返回对应数字

```
public static int hashCode(int value) { // 直接返回对应数字
    return value;
}

public boolean equals(Object obj) { // 先判断类型, 再比较值
    if (obj instanceof Integer) {
        return value == ((Integer)obj).intValue();
    }
    return false;
}
```

- Boolean://返回特殊数字

```
public static int hashCode(boolean value) {
    return value ? 1231 : 1237;
}

public boolean equals(Object obj) {
    if (obj instanceof Boolean) {
        return value == ((Boolean)obj).booleanValue();
    }
    return false;
}
```

- Long://高32位与低32位的异或

```
public static int hashCode(long value) {
    return (int)(value ^ (value >>> 32)); // 补零右移
}

public boolean equals(Object obj) {
    if (obj instanceof Long) {
        return value == ((Long)obj).longValue();
    }
    return false;
}
```

- Float: //转为int

```

public static int floatToIntBits(float value) {
    int result = floatToRawIntBits(value);
    // Check for NaN based on values of bit fields, maximum
    // exponent and nonzero significand.
    if ( ((result & FloatConsts.EXP_BIT_MASK) ==
          FloatConsts.EXP_BIT_MASK) &&
          (result & FloatConsts.SIGNIF_BIT_MASK) != 0)
        result = 0x7fc00000;
    return result;
}

public boolean equals(Object obj) {
    return (obj instanceof Float)
        && (floatToIntBits(((Float)obj).value) ==
floatToIntBits(value));
}

```

● Double: //转为long, 再按照long的方法

```

public static int hashCode(double value) {
    long bits = doubleToLongBits(value);
    return (int)(bits ^ (bits >> 32));
}

public boolean equals(Object obj) {
    return (obj instanceof Double)
        && (doubleToLongBits(((Double)obj).value) ==
doubleToLongBits(value));
}

```

● String:

```

public int hashCode() {
    int h = hash; // 默认为0
    if (h == 0 && value.length > 0) { // value为对应字符串的数组
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i]; // 循环累加
        }
        hash = h;
    }
    return h;
}

public boolean equals(Object anObject) { // 比较内容，而不仅仅是比较引用地址
    if (this == anObject) { // 如果是引用相等，直接返回true
        return true;
    }
    if (anObject instanceof String) { // 如果类型相同
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) { // 如果长度相同
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) { // 如果每个字符都相同
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}

```

● Arrays:

```

        public static int hashCode(Object a[]) { //递归调用对象本身的
hashCode计算方法, 再累加
            if (a == null)
                return 0;
            int result = 1;
            for (Object element : a)
                result = 31 * result + (element == null ? 0 :
element.hashCode()); //递归调用
            return result;
        }

        public static boolean equals(Object[] a, Object[] a2) {
            if (a==a2) //先判断是否能达到引用相等
                return true;
            if (a==null || a2==null) //判断是否都为空
                return false;

            int length = a.length;
            if (a2.length != length) //判断长度是否相等
                return false;

            for (int i=0; i<length; i++) { //递归利用每个对象自身的equals来
判断
                Object o1 = a[i];
                Object o2 = a2[i];
                if (!(o1==null ? o2==null : o1.equals(o2)))
                    return false;
            }

            return true;
        }

```

● List: //同样是递归调用

```

    public int hashCode() {
        int hashCode = 1;
        for (E e : this)
            hashCode = 31*hashCode + (e==null ? 0 : e.hashCode());
        return hashCode;
    }

    public boolean equals(Object o) { //
        if (o == this)
            return true;
        if (!(o instanceof List))
            return false;

        ListIterator<E> e1 = listIterator();
        ListIterator<?> e2 = ((List<?>) o).listIterator();
        while (e1.hasNext() && e2.hasNext()) {
            E o1 = e1.next();
            Object o2 = e2.next();
            if (!(o1==null ? o2==null : o1.equals(o2)))
                return false;
        }
        return !(e1.hasNext() || e2.hasNext());
    }
}

```

● Set:

```

    public int hashCode() {
        int h = 0;
        Iterator<E> i = iterator();
        while (i.hasNext()) {
            E obj = i.next();
            if (obj != null)
                h += obj.hashCode();
        }
        return h;
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;

        if (!(o instanceof Set))
            return false;
        Collection<?> c = (Collection<?>) o;
        if (c.size() != size())
            return false;
        try {
            return containsAll(c);
        } catch (ClassCastException unused) {
            return false;
        } catch (NullPointerException unused) {
            return false;
        }
    }

    public boolean containsAll(Collection<?> c) {
        for (Object e : c)
            if (!contains(e))
                return false;
        return true;
    }
}

```

● Map:

```

public int hashCode() {
    int h = 0;
    Iterator<Entry<K,V>> i = entrySet().iterator();
    while (i.hasNext())
        h += i.next().hashCode();
    return h;
}

public boolean equals(Object o) {
    if (o == this)
        return true;

    if (!(o instanceof Map))
        return false;
    Map<?,?> m = (Map<?,?>) o;
    if (m.size() != size())
        return false;

    try {
        Iterator<Entry<K,V>> i = entrySet().iterator();
        while (i.hasNext()) {
            Entry<K,V> e = i.next();
            K key = e.getKey();
            V value = e.getValue();
            if (value == null) {
                if (!(m.get(key)==null && m.containsKey(key)))
                    return false;
            } else {
                if (!value.equals(m.get(key)))
                    return false;
            }
        }
    } catch (ClassCastException unused) {
        return false;
    } catch (NullPointerException unused) {
        return false;
    }

    return true;
}

```

假设现在一个学生类，有学号和姓名，我现在hashCode方法重写的时候，只将学号参与计算，会出现什么情况？往set里面put一个学生对象，然后将这个学生对象的学号改了，再put进去，可以放进set么？并讲出为什么？

- 对于这种情况，首先要判断equals和hashCode是否重写覆盖，如果没有，则全部按照Object中的计算方法，即按照内存地址相关。
- 如果重写了，则要判断生成hashCode和equals判断的具体依据是什么，如果学号参与hashCode的计算，则更改学号就会导致hashCode的不同。如果学号没有参与计算，则更改学号hashCode并不会发生变化，但这两个对象是否就是相等？那还要看equals是怎么判断和实现的（hashCode相同的情况下会继续调用equals判断）。

Java:

```
Set<User> list = new HashSet<>();
User user = new User();
user.setId(1);
user.setUsername("a");
list.add(user);
System.out.println(list.toString());
user = new User();
user.setId(1);
user.setUsername("a");
list.add(user);
System.out.println(list.toString());
//output: 如果hashCode和equals没有被重载，只有新建一个对象，那么hashCode必然不相等，尽管内容可能完全一致
[CI.User@15db9742]
[CI.User@15db9742, CI.User@6d06d69c]

Set<User> list = new HashSet<>();
User user = new User();
user.setId(1);
user.setUsername("a");
list.add(user);
System.out.println(list.toString());
list.add(user);
System.out.println(list.toString());
//output: 如果不新建对象直接放入，由于内存地址一样，hashCode一样，会认为是同一个对象
[CI.User@15db9742]
[CI.User@15db9742]
```

Nio的原理？Nio和IO有什么区别？Nio和io的区别？

- **Nio的原理**：Java NIO(New IO)是一个可以替代标准Java IO API的IO API（从Java 1.4开始），Java NIO提供了与标准IO不同的IO工作方式。
 - Java NIO: Channels and Buffers（通道和缓冲区）：标准的IO基于字节流和字符流进行操作的，而NIO是基于通道（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。
 - Java NIO: Non-blocking IO（非阻塞IO）：Java NIO可以让你非阻塞的使用IO，例如：当线程从通道读取数据到缓冲区时，线程还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。
 - Java NIO: Selectors（选择器）：Java NIO引入了选择器的概念，选择器用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道。
- **Nio和IO的区别**：
 - IO面向流与Nio面向缓冲: Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。
 - 阻塞与非阻塞IO:Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么也不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。
 - 选择器（Selectors）:Java NIO 的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

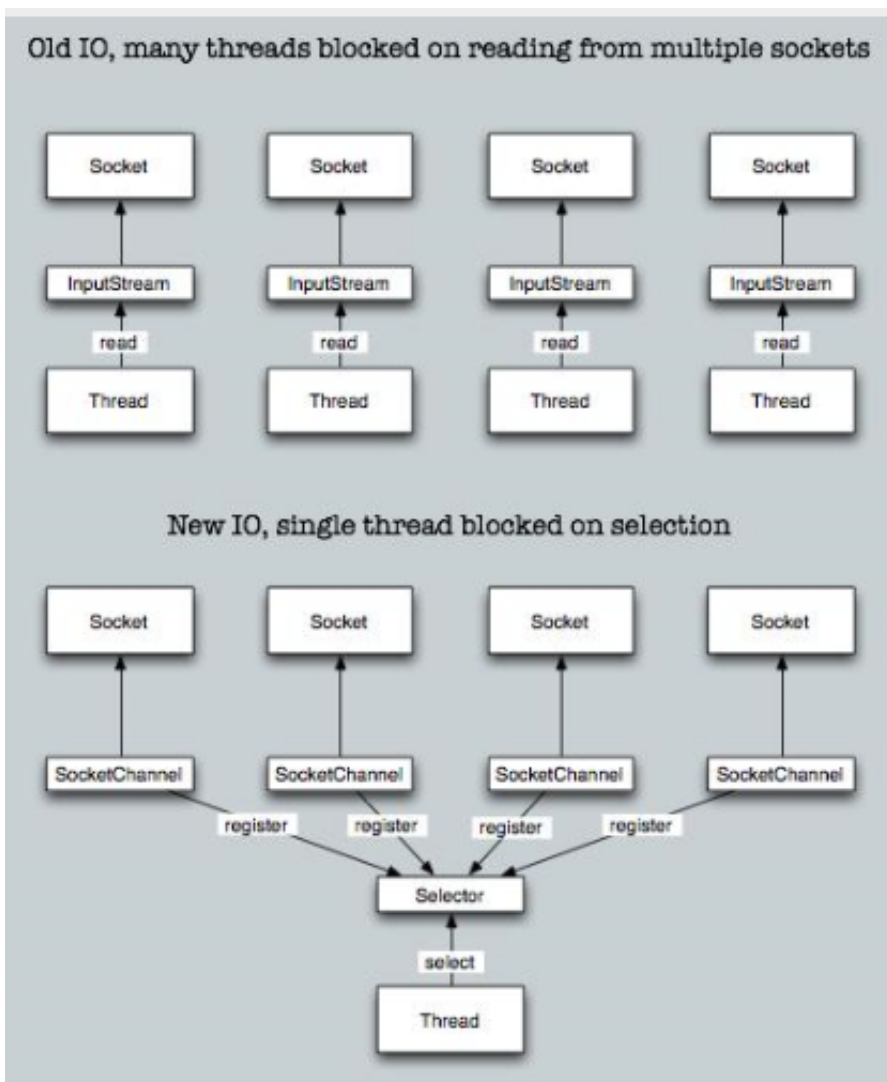
NIO, BIO, AIO分别是什么？（网络相关）

- BIO (Blocking IO) :

- 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
- BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

- NIO (Nonblocking IO) :

- 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。
- NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
- BIO与NIO一个比较重要的不同，是我们使用BIO的时候往往会引入多线程，每个连接一个单独的线程；而NIO则是使用单线程或者只使用少量的多线程，每个连接共用一个线程。



- AIO(AsynchronousIO):
 - 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。
 - AIO方式使用用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

Comparable和Comparator的区别

- Comparable是在集合内部定义的方法实现的排序，位于java.lang下。Comparator是在集合外部实现的排序，位于java.util下。
 - Comparable 自然排序。（实体类实现）Comparator 是定制排序。（无法修改实体类时，直接在调用方创建）。同时存在时采用Comparator（定制排序）的规则进行比较。
 - Comparable是一个对象本身就已经支持自比较所需要实现的接口，自定义类要在加入list容器中后能够排序，也可以实现Comparable接口，在用Collections类的sort方法排序时若不指定Comparator，那就以自然顺序排序。所谓自然顺序就是实现Comparable接口设定的排序方式。
 - Comparator是一个专用的比较器（也是个接口），当这个对象不支持自比较或者自比较函数不能满足要求时，可写一个比较器来完成两个对象之间大小的比较。Comparator体现了一种策略模式(strategy design pattern)，就是不改变对象自身，而用一个策略对象(strategy object)来改变它的行为。
-

怎么理解volatile实现可见性但不保证原子性？volatile为什么不能保证自增操作的原子性？怎么保证n++的原子性？

- volatile关键字：
 - 能够保证volatile变量的可见性
 - 不能保证volatile变量复合操作的原子性
 - 通常用作标志
 - volatile如何实现内存可见性：
 - 深入来说：通过加入内存屏障和禁止重排序优化来实现的。
 - 对volatile变量执行写操作时，会在写操作后加入一条store屏障指令
 - 对volatile变量执行读操作时，会在读操作前加入一条load屏障指令
 - 通俗地讲：volatile变量在每次被线程访问时，都强迫从主内存中重读该变量的值，而当该变量发生变化时，又会强迫线程将最新的值刷新到主内存。这样任何时刻，不同的线程总能看到该变量的最新值。
 - 线程写volatile变量的过程：
 - 改变线程工作内存中volatile变量副本的值
 - 将改变后的副本的值从工作内存刷新到主内存
 - 线程读volatile变量的过程：
 - 从主内存中读取volatile变量的最新值到线程的工作内存中
 - 从工作内存中读取volatile变量的副本
 - 保证number自增操作的原子性：
 - 使用synchronized关键字
 - 使用ReentrantLock
 - 使用AtomicInteger
-

volatile和synchronized区别

1. volatile不会进行加锁操作：volatile变量是一种稍弱的同步机制在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比synchronized关键字更轻量级的同步机制。
 2. volatile仅能使用在变量级别,synchronized则可以使用在变量,方法。
 3. volatile仅能实现变量的修改可见性,而synchronized则可以保证变量的修改可见性和原子性。
-

线程状态总结



线程总共有5大状态：

- 新建状态：新建线程对象，并没有调用start()方法之前
- 就绪状态：调用start()方法之后线程就进入就绪状态，但是并不是说只要调用start()方法线程就马上变为当前线程，在变为当前线程之前都是为就绪状态。值得一提的是，线程在睡眠和挂起中恢复的时候也会进入就绪状态哦。
- 运行状态：线程被设置为当前线程，开始执行run()方法。就是线程进入运行状态
- 阻塞状态：线程被暂停，比如说调用sleep()方法后线程就进入阻塞状态
- 死亡状态：线程执行结束

Synchronized的缺点

如果一个代码块被synchronized修饰，获取锁的线程释放锁只会有两种情况：

1. 获取锁的线程执行完了该代码块，然后线程释放对锁的占有；
2. 线程执行发生异常，此时JVM会让线程自动释放锁。

缺点：

- 不能响应中断，当多个线程尝试获取锁时，未获取到锁的线程会不断的尝试获取锁，而不会发生中断，这样会造成性能消耗；
- 同一时刻不管是读还是写都只能有一个线程对共享资源操作。
- 锁的释放由虚拟机来完成，不用人工干预，不过此即使缺点也是优点，优点是不用担心会造成死锁，缺点是由可能获取到锁的线程阻塞之后其他线程会一直等待，性能不高，无法实现超时释放锁。

锁类型

- 可重入锁：在执行对象中所有同步方法不用再次获得锁
- 可中断锁：在等待获取锁过程中可中断
- 公平锁：按等待获取锁的线程的等待时间进行获取，等待时间长的具有优先获取锁权利
- 读写锁：对资源读取和写入的时候拆分为2部分处理，读的时候可以多线程一起读，写的时候必须同步地写

Synchronized和Lock的区别

Lock不能代替Synchronized。Lock接口有三个实现类，一个是ReentrantLock,另两个是ReentrantReadWriteLock类中的两个静态内部类ReadLock和WriteLock。

1. synchronized是Java的关键字，是Java的内置特性。lock是基于jdk层面实现的接口。未来synchronized会做进一步优化。
2. 采用synchronized不需要用户去手动释放锁，当synchronized 方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象，因此使用Lock时需要在finally块中释放锁。
3. 使用synchronized，当一个线程处于等待某个锁的状态，是无法被中断的，只有一直等待下去，而Lock的lockInterruptibly方法能够在获取锁的同时保持对中断的响应。
4. synchronized、Lock都是非公平锁，但Lock可以设置为公平锁。
5. Lock可定时，可轮询，而synchronized不可以。
6. Lock可以判断锁的状态，而synchronized不可以。
7. synchronized的锁获取和释放都是基于代码块，而Lock可以实现更细粒度的锁。
8. ReentrantLock可以绑定多个Condition对象，只需多次调用new Condition(),而synchronized需要多个锁。？？？
9. ReentrantReadWriteLock实现读写锁，在频繁读取少量写入时有着更好的性能。
10. synchronized和ReentrantLock都是可重入锁。（以上两点不算区别）

synchronized 实现原理

- 在 Java 中，关键字 synchronized 可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块(主要是对方法或者代码块中存在共享数据的操作)，同时我们还应该注意到 synchronized 另外一个重要的作用，synchronized 可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到（保证可见性，完全可以替代 Volatile 功能）
- synchronized 的三种应用方式：
 - 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
 - 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
 - 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。
- Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现，无论是显式同步(有明确的 monitorenter 和 monitorexit 指令,即同步代码块)还是隐式同步都是如此。在 Java 语言中，同步用的最多的地方可能是被 synchronized 修饰的同步方法。同步方法并不是由 monitorenter 和 monitorexit 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的 ACC_SYNCHRONIZED 标志来隐式实现的。
- Java 对象头：在 JVM 中，对象在内存中的布局分为三块区域：对象头、实例数据和填充数据。



- 实例变量：存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。
- 填充数据：由于虚拟机要求对象起始地址必须是8字节的整数倍。填充数据

不是必须存在的，仅仅是为了字节对齐。

- 对于顶部，则是**Java头对象**，它实现synchronized的锁对象的基础。synchronized使用的锁对象是存储在Java对象头里的，jvm中采用2个字来存储对象头(如果对象是数组则会分配3个字，多出来的1个字记录的是数组长度)，其主要结构是由**Mark Word** 和 Class Metadata Address 组成。
- Mark Word在默认情况下存储着对象的HashCode、分代年龄、锁标记位等。

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

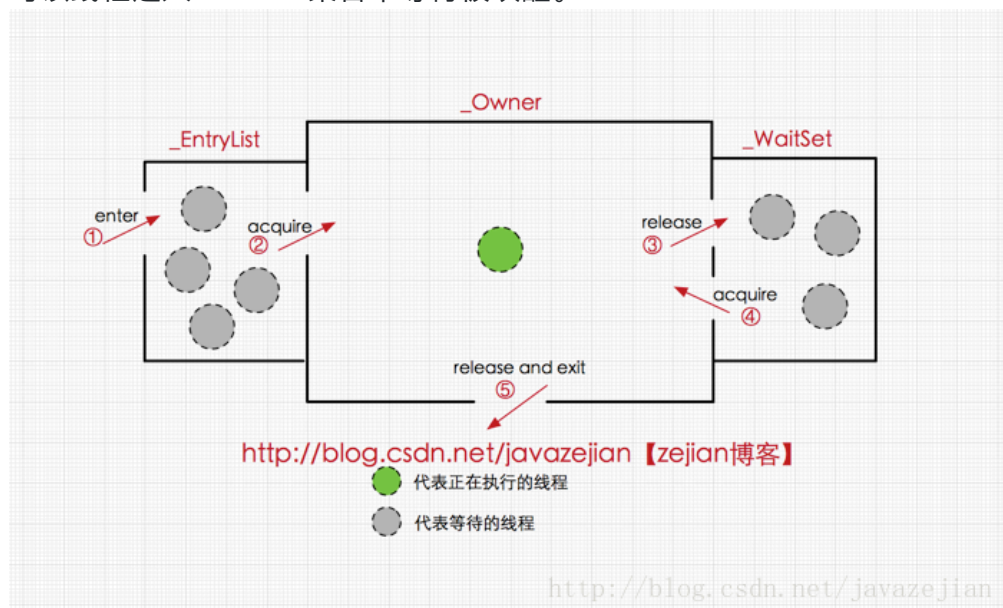
- 每个对象都存在着一个 **monitor** 与之关联，对象与其 monitor 之间的关系有存在多种实现方式，如monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 monitor 被某个线程持有后，它便处于锁定状态。在Java虚拟机(HotSpot)中，monitor是由**ObjectMonitor**实现的，其主要数据结构如下：

```

ObjectMonitor() {
    _header      = NULL;
    _count      = 0; // 记录个数
    _waiters    = 0,
    _recursions  = 0;
    _object     = NULL;
    _owner      = NULL; // 指向持有ObjectMonitor对象的线程
    _WaitSet    = NULL; // 处于wait状态的线程, 会被加入到
    _WaitSet
    _WaitSetLock = 0 ;
    _Responsible = NULL ;
    _succ       = NULL ;
    _cxq       = NULL ;
    FreeNext    = NULL ;
    _EntryList  = NULL ; // 处于等待锁block状态的线程, 会被加入
    到该列表
    _SpinFreq   = 0 ;
    _SpinClock  = 0 ;
    OwnerIsThread = 0 ;
}

```

- 当多个线程同时访问一段同步代码时，首先会进入 `_EntryList` 集合，当线程获取到对象的monitor后进入 `_Owner` 区域并把monitor中的owner变量设置为当前线程同时monitor中的计数器count加1，若线程调用 `wait()` 方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 `WaitSet`集合中等待被唤醒。



- monitor对象存在于每个Java对象的对象头中(存储的指针的指向)，

synchronized锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因，同时也是notify/notifyAll/wait等方法存在于顶级对象Object中的原因。

- 锁的状态总共有四种，无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，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。

synchronized锁的膨胀

- 锁的状态：
 - 无锁状态
 - 偏向锁状态
 - 轻量级锁状态
 - 重量级锁状态
- 四种状态会随着竞争的情况逐渐升级，而且是不可逆的过程，即不可降级。要注意的是，这四种状态都不是Java语言中的锁，而是Jvm为了提高锁的获取与释放效率而做的优化(使用synchronized时)。
-

偏向锁 仅有一个线程进入临界区

- 大多数情况下，锁不存在多线程竞争，而是总是由同一线程多次获得时，为了使线程获得锁的代价更低而引入了偏向锁。
- 偏向锁的使用：
 - 测试对象头Mark Word(默认存储对象的HashCode,分代年龄，锁标记位)里是否存储着指向当前线程的偏向锁。
 - 若测试失败，则测试Mark Word中偏向锁标识是否设置成1(表示当前为偏向锁)
 - 没有设置则使用CAS竞争，否则尝试使用CAS将对象头的偏向锁指向当前线程
- 偏向锁的撤销：当其他线程尝试竞争偏向锁时，就会释放锁，锁的撤销，需要等待全局安全点，分为以下几个步骤：
 - 暂停拥有偏向锁的线程，检查线程是否存活
 - 处于非活动状态，则设置为无锁状态
 - 存活，则重新偏向于其他线程或者恢复到无锁状态或者标记对象不适合作为偏向锁
 - 唤醒线程
- 偏向锁的升级：当有第二个线程进入同步代码块时，则升级为轻量级锁。

轻量级锁多个线程交替进入临界区，不竞争

- 轻量级锁的加锁：如果成功使用CAS将对象头中的Mark Word替换为指向锁记录的指针，则获得锁，失败则当前线程尝试使用自旋(循环等待)来获取锁。
- 轻量级锁的升级：当有另一个线程与该线程同时竞争时，锁会升级为重量级锁。为了防止继续自旋，一旦升级，将无法降级。

重量级锁多个线程同时进入临界区，竞争

- 其他线程试图获取锁时，都会被阻塞，只有持有锁的线程释放锁之后才会唤醒这些线程，进行竞争。

锁	优 点	缺 点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗 CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长

Java 1.6对synchronized锁的优化

1. 自旋锁：在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久的将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式。
2. 适应性自旋锁：线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多。反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源。有了自适应自旋锁，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测会越来越准确，虚拟机会变得越来越聪明。
3. 锁消除：Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间。
4. 锁粗化：一般情况下，需要让同步块的作用范围尽可能小—仅在共享数据的实际作用域中才进行同步，这样做的目的是为了为了使需要同步的操作数量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。但是如果一系列的连续加锁解锁操作，可能会导致不必要的性能损耗，所以需要将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。
5. 轻量级锁：（多个线程交替进入临界区）引入轻量级锁的主要目的是在多没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。
6. 偏向锁：（仅有一个线程进入临界区）偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。

sleep与wait方法的区别

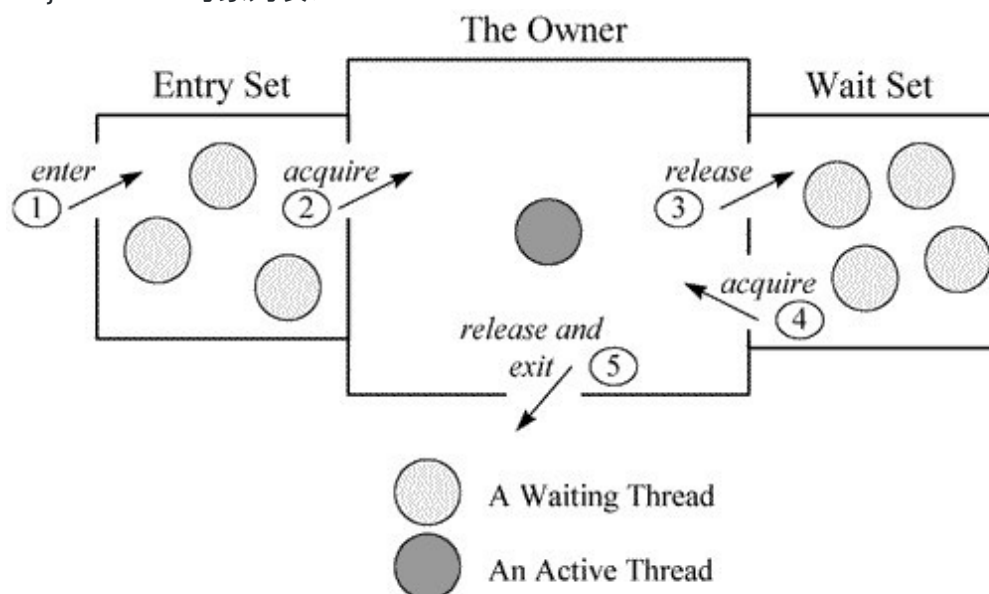
- sleep方法只让线程休眠并不释放锁。
 - wait方法调用完成后，线程将被暂停，但wait方法将会释放当前持有的监视器锁(monitor)，直到有线程调用notify/notifyAll方法后方能继续执行。
-

进入wait/notify方法之前，为什么要获取synchronized锁？

因为调用这几个方法前必须拿到当前对象的监视器monitor对象，也就是说notify/notifyAll和wait方法依赖于monitor对象，而synchronized关键字可以获取monitor。

wait/notify/notifyAll的实现原理

- ObjectMonitor对象中有两个队列：_WaitSet 和 _EntryList，用来保存ObjectWaiter对象列表；



- _WaitSet：处于wait状态的线程，会被加入到wait set；
- _EntryList：处于等待锁block状态的线程，会被加入到entry set；
- wait**方法实现：将当前线程添加到_WaitSet列表，并释放锁。
- notify**方法实现：
 - 如果当前_WaitSet为空，即没有正在等待的线程，则直接返回；
 - 如果有，获取_WaitSet列表中的第一个ObjectWaiter节点（虽说是随机，但一般是第一个）
 - 根据不同的策略，将取出来的ObjectWaiter节点，加入到_EntryList或则通过Atomic::cmpxchg_ptr指令进行自旋操作cxq
- notifyAll**方法实现：通过for循环取出_WaitSet的ObjectWaiter节点，并根据不同策略，加入到_EntryList或则进行自旋操作。
- 优先使用notifyAll，因为如果A,B同时在等待条件，而AB等待的条件有不同，此时C线程的操作可能使B的条件满足，但如果调用notify，则有可能唤醒的是A线程，那么就会造成丢失信号。

Runnable、Thread、Callable区别

Java中实现多线程有3种方法：

1. 继承Thread类
2. 实现Runnable接口
3. 实现Callable接口

- 继承Thread类

继承Thread类，需要覆盖方法 run()方法，在创建Thread类的子类时需要重写 run(),加入线程所要执行的代即可。

```
public class ThreadByExtends {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         new MyThread().start();
8         new MyThread().start();
9         new MyThread().start();
10    }
11
12 }
13
14 class MyThread extends Thread {
15     private int ticket = 5;
16
17     public void run() {
18
19         for (int i = 0; i < 10; i++) {
20             if (ticket > 0) {
21                 System.out.println("车票第" + ticket-- +
"张");
22             }
23         }
24     }
25
26 }
```

- 实现Runnable接口

Runnable是可以共享数据的，多个Thread可以同时加载一个Runnable，当各自Thread获得CPU时间片的时候开始运行Runnable，Runnable里面的资源是被共享的，所以使用Runnable更加的灵活。

```

public class ThreadRunnable {
4
5     public static void main(String[] args) {
6         MyThread1 myThread = new MyThread1();
7         new Thread(myThread).start();
8         new Thread(myThread).start();
9     }
10 }
11
12 class MyThread1 implements Runnable {
13
14     private int ticket = 5;
15
16     public void run() {
17         for (int i = 0; i < 10; i++) {
18             if (ticket > 0) {
19                 System.out.println("ticket = " + ticket--);
20             }
21         }
22     }
23
24 }

```

- 实现Callable接口

Runnable是执行工作的独立任务，但是它不返回任何值。如果你希望任务在完成的能返回一个值，那么可以实现Callable接口而不是Runnable接口。

```

9
10 public class ThreadCallable extends Panel {
11     public static void main(String[] args) {
12         MyThread2 myThread2 = new MyThread2();
13         FutureTask<Integer> futureTask = new FutureTask<>
14         (myThread2);
15         new Thread(futureTask, "线程名：有返回值的线程
16 2").start();
17     try {
18         System.out.println("子线程的返回值： " +
19         futureTask.get());
20     } catch (Exception e) {
21         e.printStackTrace();
22     }
23 }
24
25 class MyThread2 implements Callable<Integer> {
26     public Integer call() throws Exception {
27         System.out.println("当前线程名——" +
28         Thread.currentThread().getName());
29         int i = 0;
30         for (; i < 5; i++) {
31             System.out.println("循环变量i的值： " + i);
32         }
33         return i;
34     }
35 }
36
37 }

```

- 实现Runnable接口相比继承Thread类有如下优势：
 - 可以避免由于Java的单继承特性而带来的局限；
 - 增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的；
 - 适合多个相同程序代码的线程区处理同一资源的情况。
- 实现Runnable接口和实现Callable接口的区别：
 - Runnable是自从java1.1就有了，而Callable是1.5之后才加上去的

- Callable规定的方法是call(),Runnable规定的方法是run()
 - Callable的任务执行后可返回值，而Runnable的任务是不能返回值(是void)
 - call方法可以抛出异常，run方法不可以
 - 运行Callable任务可以拿到一个Future对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过Future对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。
 - 加入线程池运行，Runnable使用ExecutorService的execute方法，Callable使用submit方法。
-

使用线程池的优点

1. **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
2. **提高响应速度。**当任务到达时，任务可以不需要的等到线程创建就能立即执行。
3. **提高线程的可管理性。**线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

线程池的参数

1. **newFixedThreadPool 固定长度线程池**，corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue
 2. **newCachedThreadPool 可缓存线程池**，根据需求自动增加减少，规模不存在限制，将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程
 3. **newSingleThreadExecutor 单线程**，将corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue；
 4. **newScheduledThreadPool 固定长度线程池**，而且以延迟或定时的方式来执行，传入队列是DelayedWorkQueue()。
- 对于计算密集型的任务，在拥有N个cpu的系统上，当线程池的大小为N+1时，通常能实现最优利用率。对于I/O密集型，通常线程池大小为2*N。
 - corePoolSize

- 在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，（除非调用了`prestartAllCoreThreads()`或者`prestartCoreThread()`方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建`corePoolSize`个线程或者一个线程）。
- 默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，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。核心线程在`allowCoreThreadTimeout`被设置为`true`时会超时退出，默认情况下不会退出。
- `maxPoolSize`
 - 当线程数大于或等于核心线程，且任务队列已满时，线程池会创建新的线程，直到线程数量达到`maxPoolSize`。如果线程数已等于`maxPoolSize`，且任务队列已满，则已超出线程池的处理能力，线程池会拒绝处理任务而抛出异常。
- `keepAliveTime`
 - 当线程空闲时间达到`keepAliveTime`，该线程会退出，直到线程数量等于`corePoolSize`。如果`allowCoreThreadTimeout`设置为`true`，则所有线程均会退出直到线程数量为0。
- `allowCoreThreadTimeout`：是否允许核心线程空闲退出，默认值为`false`。
- `queueCapacity`
 - 任务队列容量。从`maxPoolSize`的描述上可以看出，任务队列的容量会影响到线程的变化，因此任务队列的长度也需要恰当的设置。
- `workQueue`：
 - 一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：
 - `ArrayBlockingQueue`：基于数组的先进先出队列，此队列创建时必须指定大小；
 - `LinkedBlockingQueue`：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为`Integer.MAX_VALUE`；

- `synchronousQueue`：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。
 - 一般使用`LinkedBlockingQueue`和`Synchronous`。
 - 线程池按以下行为执行任务
 - 当线程数小于核心线程数时，创建线程。
 - 当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。
 - 当线程数大于等于核心线程数，且任务队列已满
 - 若线程数小于最大线程数，创建线程
 - 若线程数等于最大线程数，抛出异常，拒绝任务
-

Condition简介 与 Object监视器的区别

```

public class ConditionDemo {
    @Test
    public void test() {
        final ReentrantLock reentrantLock = new ReentrantLock();
        final Condition condition = reentrantLock.newCondition();

        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    reentrantLock.lock();

System.out.println(Thread.currentThread().getName() + "在等待被唤醒");

                    condition.await();

System.out.println(Thread.currentThread().getName() + "恢复执行了");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    reentrantLock.unlock();
                }
            }
        }, "thread1").start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    reentrantLock.lock();

System.out.println(Thread.currentThread().getName() + "抢到了锁");
                    condition.signal();

System.out.println(Thread.currentThread().getName() + "唤醒其它等待的线程");

                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    reentrantLock.unlock();
                }
            }
        }, "thread2").start();
    }
}

```

- Condition接口位于java.util.concurrent.locks包下，实现类有AbstractQueuedLongSynchronizer.ConditionObject和AbstractQueuedSynchronizer.ConditionObject。Condition将Object监视器方法(wait、notify和 notifyAll)分解成截然不同的对象，以便通过将这些对象与任意Lock实现组合使用。其中，Lock替代了synchronized方法的使用及作用，Condition替代了Object监视器方法的使用及作用。Condition的await方法代替Object的wait；Condition的signal方法代替Object的notify方法；Condition的signalAll方法代替Object的notifyAll方法。Condition实例在使用时需要绑定到一个锁上，可以通过newCondition方法获取Condition实例。Condition实现可以提供不同于Object监视器方法的行为和语义，比如受保证的通知排序，或者在执行通知时不需要保持一个锁。
- 创建Condition实例，通过Lock接口实现类的新Condition方法获取Condition实例

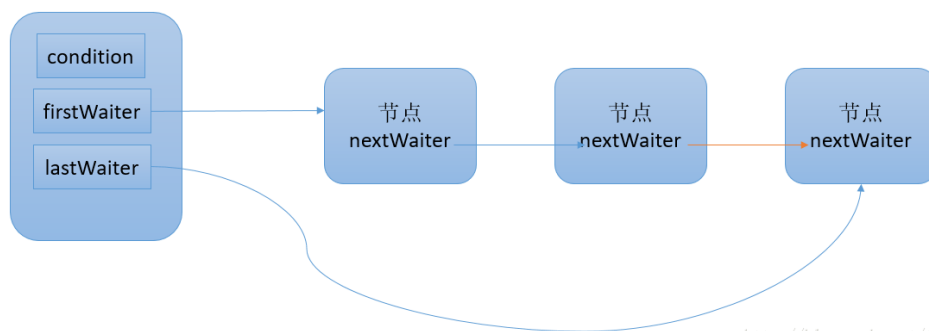
```
ReentrantLock reentrantLock = new ReentrantLock();  
Condition condition = reentrantLock.newCondition();
```

- 常用方法
 - await(): 调用await方法后，当前线程在接收到唤醒信号之前或被中断之前一直处于等待休眠状态。调用此方法时，当前线程保持了与此Condition有关联的锁，调用此方法后，当前线程释放持有的锁。此方法可以返回当前线程之前，都必须重新获取与此条件有关的锁，在线程返回时，可以保证它保持此锁。
 - signal(): 唤醒一个等待线程，如果所有的线程都在等待此条件，则选择其中的一个唤醒。在从await返回之前，该线程必须重新获取锁。
 - signalAll(): 唤醒所有等待线程，如果所有的线程都在等待此条件，则唤醒所有线程。在从await返回之前，每个线程必须重新获取锁。
- 与 Object监视器的区别
 - Object监视器的区别与Synchronized配合使用，而Condition与Lock配合使用
 - Condition功能比内置条件队列更加丰富，每个锁上可以存在多个等待，可以中断，可以公平。
 - 对于内置监视器来说，等待队列只有一个，即使每个线程等待的条件不同，也是加入同一个等待队列，那么用notifyAll唤醒时，所有的等待线程都

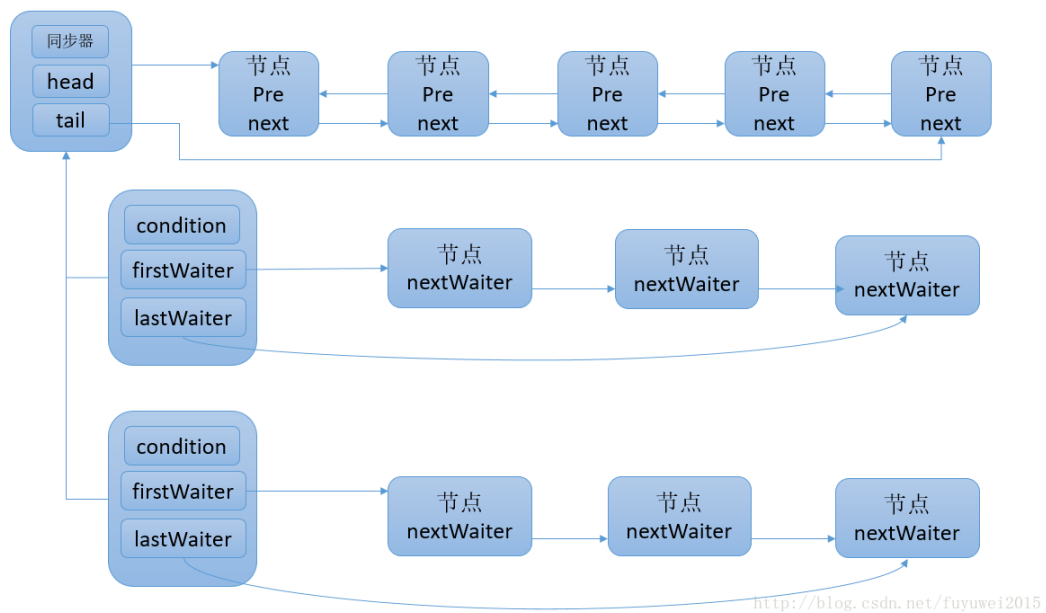
会重新竞争锁，但此时往往大部分线程依然继续阻塞，因为其等待的条件是不同的。但是Condition可以根据等待条件的不同设置多个等待队列，当某一条件满足时，可以将该等待队列中的所有等待线程唤醒，这样可以实现精准唤醒，避免无用唤醒浪费资源。

Condition原理

- ConditionObject是同步器AbstractQueuedSynchronizer的内部类。
- 等待队列是一个FIFO的队列，在队列中的每个节点都包含了一个线程引用，该线程就是在Condition对象上等待的线程，如果一个线程调用了Condition.await()方法，那么该线程将会释放锁、构造成节点加入等待队列并进入等待状态一个Condition包含一个等待队列，Condition拥有首节点（firstWaiter）和尾节点（lastWaiter）。当前线程调用Condition.await()方法，将会以当前线程构造节点，并将节点从尾部加入等待队列。



- 在Object的监视器模型上，一个对象拥有一个同步队列和等待队列，而并发包中的Lock（更确切地说是同步器）拥有一个同步队列和多个等待队列。



多线程中断

- 每一个线程都有一个boolean类型标志，用来表明当前线程是否请求中断，当一个线程调用interrupt()方法时，线程的中断标志将被设置为true，但不会立即中断，只是传递了中断请求的消息。。可以通过调用Thread.currentThread().isInterrupted()或者Thread.interrupted()来检测线程的中断标志是否被置位。这两个方法的区别是Thread.currentThread().isInterrupted()是线程对象的方法，调用它后不清除线程中断标志位；而Thread.interrupted()是一个静态方法，调用它会清除线程中断标志位。
- 中断非阻塞线程
 - 采用线程共享变量，共享变量必须设置为volatile，这样才能保证修改后其他线程立即可见，用while循环手动检测。
 - 采用中断机制，设置中断标志位，由系统自己检测。

```

public class InterruptRunnableDemo extends Thread {
    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            // ... 单次循环代码
        }
        System.out.println("done ");
    }
    public static void main(String[] args) throws
    InterruptedException {
        Thread thread = new InterruptRunnableDemo();
        thread.start();
        Thread.sleep(1000);
        thread.interrupt();
    }
}

```


- 可中断的阻塞线程

- 当线程调用Thread.sleep()、Thread.join()、object.wait()等阻塞方法时，调用 interrupt() 方法设置线程中断标志位时，清除中断状态，并抛出 InterruptedException异常。
- wait() & interrupt() 线程A调用了wait()进入了等待状态,也可以用interrupt()取消。不过这时候要小心锁定的问题。线程进入等待区,会把锁定解除,当对等待中的线程调用interrupt()时(注意是等待的线程调用其自己的interrupt()),会先重新获取锁定,再抛出异常。在获取锁定之前,是无法抛出异常的。

- 不可中断的阻塞线程

- 同步Socket I/O、同步 I/O、调用synchronized关键字和reentrantLock.lock()获取锁被阻塞时，只能设置中断状态，而没有其他任何效果。

AbstractQueuedSynchronizer (AQS) (同时也是ReentrantLock/Semaphore/CountDownLatch的实现底层原理，只是独占和共享的区别)

<https://segmentfault.com/a/1190000008471362><https://www.cnblogs.com/waterystone/p/4920797.html> 

- 抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch。JCU包里面几乎所有的有关锁、多线程并发以及线程同步器等重要组件的实现都是基于AQS这个框架。
- ReentrantLock将同步状态用于保存锁获取操作的次数，还维护一个owner变量来保存当前所有者线程的标识符，只有在当前线程刚获得锁，或者释放锁时才会修改。
- Semaphore将AQS的状态用于保存当前可用许可的数量。CountDownLatch保存的是当前的计数值
- FutureTask中，AQS同步状态用于保存任务的状态。
- ReentrantReadWriteLock中，单个AQS子类同时管理读取锁和写入锁。16位表示写入锁的计数，16位使用读取锁的计数。读取锁是共享，写入锁是独占。
- aqs实现的公平锁和非公平锁的最主要的区别是：非公平锁中，那些尝试获取锁且尚未进入等待队列的线程会和等待队列head结点的线程发生竞争。公平锁中，在获取锁时，增加了isFirst(current)判断，当且仅当，等待队列为空或当前线程是等待队列的头结点时，才可尝试获取锁。
- 它维护了一个**volatile int state**（代表共享资源）和一个**FIFO线程等待队列**（多线程争用资源被阻塞时会进入此队列）。

```
private transient volatile Node head;

private transient volatile Node tail;

private volatile int state;
```

- state的访问方式有三种：
 - getState()
 - setState()
 - compareAndSetState()

- AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。
- 获取同步状态：
 - 以ReentrantLock为例,假设线程A要获取同步状态（这里想象成锁，方便理解），初始状态下state=0,所以线程A可以顺利获取锁，A获取锁后将state置为1。在A没有释放锁期间，线程B也来获取锁，此时因为state=1，表示锁被占用，所以将B的线程信息和等待状态等信息构成出一个Node节点对象，放入同步队列，head和tail分别指向队列的头部和尾部（此时队列中有一个空的Node节点作为头点，head指向这个空节点，空Node的后继节点是B对应的Node节点，tail指向它），同时阻塞线程B(这里的阻塞使用的是LockSupport.park()方法)。后续如果再有线程要获取锁，都会加入队列尾部并阻塞。释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多少次，这样才能保证state是能回到零态的。
 - 以CountDownLatch为例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后（即state=0），会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后续动作。
- 自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：
 - isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它
 - tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
 - tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
 - tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
 - tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。
- 自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-

tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

- acquire(int)

```
public final void acquire(int arg) {  
    if (!tryAcquire(arg) &&  
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))  
        selfInterrupt();  
}
```



- 调用自定义同步器的tryAcquire()尝试直接去获取资源，如果成功则直接返回；
- 没成功，则addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
- acquireQueued()使线程在等待队列中休息，有机会时（轮到自己，会被unpark()）会去尝试获取资源。获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。
- 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

- release(int)

```
public final boolean release(int arg) {  
    if (tryRelease(arg)) {  
        Node h = head; // 找到头结点  
        if (h != null && h.waitStatus != 0)  
            unparkSuccessor(h); // 唤醒等待队列里的下一个线程  
        return true;  
    }  
    return false;  
}
```

- release()是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0），它会唤醒等待队列里的其他线程来获取资源。

- acquireShared(int):与acquire基本相同，不同的是在本线程获得资源后，其他

符合条件的线程还可以继续获得。独占模式只能是同一个线程。

- `releaseShared()`：跟独占模式下的`release()`相似，但有一点稍微需要注意：独占模式下的`tryRelease()`在完全释放掉资源（`state=0`）后，才会返回`true`去唤醒其他线程，这主要是基于独占下可重入的考量；而共享模式下的`releaseShared()`则没有这种要求，共享模式实质就是控制一定量的线程并发执行，那么拥有资源的线程在释放掉部分资源时就可以唤醒后继等待结点。

java.util.concurrent.atomic

- `AtomicBoolean`、`AtomicInteger`、`AtomicLong`、`AtomicReference`以原子方式进行操作 `AtomicInteger`构造函数无参-初始化为0。
- 相比于以前使用`synchronized(obj){增加}`，提升了性能，因为`AtomicInteger`内部通过JNI（Java Native Interface）的方式使用了硬件支持的CAS指令（compare and swap）

闭锁 / java.util.concurrent.CountDownLatch

- `CountDownLatch`主要提供的机制是当多个（具体数量为初始化传入的个数）线程都达到了预期状态或完成预期工作时触发事件，其他线程可以等待这个事件来触发自己的后续工作（等待的线程可以是多个）。到达自己预期状态的线程会调用`CountDownLatch`的`countDown`方法，而等待线程会调用`await`方法。
- 比如`CountDownLatch latch = new CountDownLatch(10)`，开10个线程计算数据，`run`方法中，计算完成后调用`latch.countDown()`；比如还有另外两个线程，执行到某个地方，调用`latch.await()`，只有当10个线程都`countDown`，才能唤醒`await`。注意：10个线程执行`countDown()`后不会停顿，还会继续执行自己的任务。

栅栏 / java.util.concurrent.CyclicBarrier

- CyclicBarrier可以协同多个线程，让多个线程在这个屏障前等待，直到所有线程都到达了这个屏障时，再一起继续执行后面的动作。
- 比如CyclicBarrier barrier = new CyclicBarrier(3);3个线程run方法中均有barrier.await()方法，任何一个线程到了await()会进入阻塞等待状态，直到3个线程都到了await才会同时从await返回，继续后续的工作。如果在构造CyclicBarrier时设置了一个Runnable实现（构造函数第二个参数），那么最后一个到await的线程会执行这个runnable的run方法。
- CyclicBarrier和CountDownLatch都是用于多个线程间的协调。区别：1、CyclicBarrier线程到了await()都会阻塞，然后一起返回；CountDownLatch线程到了countDown()后不会停顿，还会继续执行自己的任务，到await()的会阻塞。2、CountDownLatch不能重复使用，用完了还需要重新初始化一个，CyclicBarrier可以重复使用。

信号量 / java.util.concurrent.Semaphore

- Semaphore是用于管理信号量的，构造时传入可供管理的信号量数值。主要用于控制并发数，如果Semaphore管理的信号量只有1个，就退化到互斥锁了。与通过控制线程数来控制并发数的方法相比，通过Semaphore来控制并发数可以控制得更加细粒度，因为真正需要有并发数限制的代码只需要放到acquire和release之间即可（其他仍可以由更多线程执行）。

```
semaphore.acquire();//acquire、release可以有参数，获取/ 返还的信号量个数
try{// 比如调用远程通信方法}
finally{semaphore.release();}
acquire一个，信号量的个数就减少一个，如果没了acquire就会阻塞，等release之后才会返回。
```

java.util.concurrent.Exchanger

- Exchanger用于在两个线程之间进行数据交换，线程会阻塞在Exchanger的exchange方法上，直到另外一个线程也到了同一个Exchanger的exchange方法时，二者进行交换，然后两个线程会继续执行后续代码。
- Exchanger定义时有泛型，泛型即为需要交换的数据类型。exchange()方法，括号中传入需要交换的数据（符合之前定义的泛型）。

java.util.concurrent.Future/FutureTask

- Future是一个接口，FutureTask是一个具体实现类。有一个方法从远程获取一些计算结果，需要很长时间，可以用异步获取，等到需要使用该数据的时候，在get出来（或使用回调的方式，计算完成后再回调）。
-

JUC相关的集合，， Collections.sort函数jdk7 和 jdk8 分别怎么实现的？

锁的膨胀过程，Synchronized和Lock的区别，底层的monitor实现和unsafe类的CAS函数，参数表示什么，寄存器cpu如何做）？

可重入锁的含义？Synchronized是可重入锁吗？如果不是，将产生哪些危害？

是否阅读过java并发包的源码？如何理解AQS类在ReentrantLock，Simaphore等同步工具中所起的作用

并发和多线程（线程池、SYNC和Lock锁机制、线程通信、volatile、ThreadLocal、CyclicBarrier、Atom包、CountDownLatch、AQS、CAS原理等等）

Synchronize关键字为什么jdk1.5后效率提高了？synchronized关键字，实现原理（和Lock对比着说，说到各自的优缺点，synchronized从最初性能差到jdk高版本后的锁膨胀机制，大大提高性能，再说底层实现，Lock的乐观锁机制，通过AQS队列同步器，调用了unsafe的CAS操作，CAS函数的参数及意义；同时可以说说synchronized底层原理，jvm层的monitor监视器，对于方法级和代码块级，互斥原理的不同，+1-1可重入的原理等）

手写java多线程

手写java的soeket编程，服务端和客户端

Channel和buffer？directBuffer和buffer的区别？