Java面试题-基础+集合原理

一.面向对象的特征

面向对象的三个基本特征是：封装、继承、多态。

1.封装

封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。
封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

2.继承

面向对象编程 (OOP) 语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

3.多态

多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。
实现多态，有二种方式，重写，重载。

二.重载和重写的区别

1.重载 Overload

表示同一个类中可以有多个名称相同的方法，但这些方法的参数列表各不相同（即参数个数或类型不同）。

2.重写(覆盖) Override

表示子类中的方法可以与父类中的某个方法的名称和参数完全相同，通过子类创建的实例对象调用这个方法时，将调用子类中的定义方法，这相当于把父类中定义的那个完全相同的方法给覆盖了，这也是面向对象编程的多态性的一种表现。子类覆盖父类的方法时，只能比父类抛出更少的异常，或者是抛出父类抛出的异常的子异常，因为子类可以解决父类的一些问题，不能比父类有更多的问题。子类方法的访问权限只能比父类的更大，不能更小。如果父类的方法是private类型，那么，子类则不存在覆盖的限制，相当于子类中增加了一个全新的方法。

三.equals 与 == 的区别

- `==` 与 `equals` 的主要区别是：`==` 常用于比较原生类型，而 `equals()` 方法用于检查对象的相等性。
- 另一个不同的点是：如果 `==` 和 `equals()` 用于比较对象，当两个引用地址相同，`==` 返回 true。而 `equals()` 可以返回 true 或者 false 主要取决于重写实现。最常见的一个例子，字符串的比较，不同情况 `==` 和 `equals()` 返回不同的结果。

四.final, finally, finalize 的区别

1.final

用于声明属性,方法和类, 分别表示属性不可变, 方法不可覆盖, 类不可继承.

2.finally

是异常处理语句结构的一部分，表示总是执行.

3.finalize

是Object类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等. JVM不保证此方法总被调用.

附加.try或catch语句块中存在return语句,finally会不会执行?

结论:

1.不管有没有出现异常，finally块中代码都会执行；

2.当try和catch中有return时，finally仍然会执行；

3.finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，不管finally中的代码怎么样，返回的值都不会改变，任然是之前保存的值），所以函数返回值是在finally执行前确定的；

4.finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值。

举例:

情况1：try{}catch(){}finally{} return;显然程序按顺序执行。

情况2:try{ return; }catch(){} finally{} return;
程序执行try块中return之前（包括return语句中的表达式运算）代码；
再执行finally块，最后执行try中return;
finally块之后的语句return，因为程序在try中已经return所以不再执行。

情况3:try{ } catch(){return;} finally{} return;
程序先执行try，如果遇到异常执行catch块，有异常：则执行catch中return之前（包括return语句中的表达式运算）代码，再执行finally语句中全部代码,最后执行catch块中return. finally之后的代码不再执行。
无异常：执行完try再finally再return.

情况4:try{ return; }catch(){} finally{return;}
程序执行try块中return之前（包括return语句中的表达式运算）代码；

再执行finally块，因为finally块中有return所以提前退出。

情况5:try{} catch(){return;}finally{return;}
程序执行catch块中return之前（包括return语句中的表达式运算）代码；
再执行finally块，因为finally块中有return所以提前退出。

情况6:try{ return;}catch(){return;} finally{return;}
程序执行try块中return之前（包括return语句中的表达式运算）代码；
有异常：执行catch块中return之前（包括return语句中的表达式运算）代码；
则再执行finally块，因为finally块中有return所以提前退出。
无异常：则再执行finally块，因为finally块中有return所以提前退出。

最终结论：任何执行try 或者catch中的return语句之前，都会先执行finally语句，如果finally存在的话。如果finally中有return语句，那么程序就return了，所以finally中的return是一定会被return的，
编译器把finally中的return实现为一个warning。

五.int 和 Integer 有什么区别

int 是 Java 提供的 8 种原始数据类型之一。Java 为每个原始类型提供了封装类，Integer 是 Java 为 int 提供的封装类。
int 的默认值为 `0`，而 Integer 的默认值为 `null`，是引用类型，即 Integer 可以区分出未赋值和值为 0 的区别，int 则无法表达出未赋值的情况，

Java 中 int 和 Integer 关系是比较微妙的。关系如下：

- int 是基本的数据类型；
- Integer 是 int 的封装类；
- int 和 Integer 都可以表示某一个数值；
- int 和 Integer 不能够互用，因为他们两种不同的数据类型；

六.ArrayList 与 LinkedList 区别

- 因为 Array 是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。Array 获取数据的时间复杂度是 `O(1)`,但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据。

- 相对于 ArrayList，LinkedList 插入是更快的。因为 LinkedList 不像 ArrayList 一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据

重新装入一个新的数组，这是 ArrayList 最坏的一种情况，时间复杂度是 `O(n)`，而 LinkedList 中插入或删除的时间复杂度仅为 `O(1)`。ArrayList 在插入数据时还需要更新索引（除了插入数组的尾部）。

- 类似于插入数据，删除数据时，LinkedList 也优于 ArrayList。
- LinkedList 需要更多的内存，因为 ArrayList 的每个索引的位置是实际的数据，而 LinkedList 中的每个节点中存储的是实际的数据和前后节点的位置。
- 如果你的应用经常需要随机访问数据,则考虑使用ArrayList。因为如果需要 LinkedList 中的第 n 个元素的时候，你需要从第一个元素顺序数到第 n 个数据，然后读取数据。
- 你的应用经常进行插入和删除元素，更少的读取数据,考虑使用LinkedList。因为插入和删除元素不涉及重排数据，所以它要比 ArrayList 要快。

七.ArrayList 与 Vector 区别

- 同步性：Vector 是线程安全的，也就是说是同步的 ，而 ArrayList 是线程不安全的，不是同步的。
- 数据增长：当需要增长时，Vector 默认增长为原来一倍 ，而 ArrayList 却是原来的 50% ，这样 ArrayList 就有利于节约内存空间。

说明：如果涉及到堆栈，队列等操作，应该考虑用 Vector，如果需要快速随机访问元素，应该使用 ArrayList

八.List 和 Map 区别

- List 特点：元素有放入顺序，元素可重复;
- Map 特点：元素按键值对存储，无放入顺序 ;
- List 接口有三个实现类：LinkedList，ArrayList，Vector;
- LinkedList：底层基于链表实现，链表内存是散乱的，每一个元素存储本身内存地址的同时还存储下一个元素的地址。链表增删快，查找慢;
- Map 接口有三个实现类：HashMap，HashTable，LinkedHashMap
- Map 相当于和 Collection 一个级别的；Map 集合存储键值对，且要求保持键的唯一性；

九.List 和 Set 区别

- List, Set 都是继承自 Collection 接口
- List 特点：元素有放入顺序，元素可重复。Set 特点：元素无放入顺序，元素不可重复（注意：元素虽然无放入顺序，但是元素在 set 中的位置是有该元素的 HashCode 决定的，其位置其实是固定的）
- List 接口有三个实现类：LinkedList，ArrayList，Vector。Set 接口有两个实现类：HashSet(底层由 HashMap 实现)，LinkedHashSet

十.HashMap 的工作原理及代码实现

在JDK1.6，JDK1.7中，HashMap采用位桶+链表实现，即使用链表处理冲突,同一hash值的键值对会被放在同一个位桶里，当桶中元素较多时，通过key值查找的效率较低。

而JDK1.8中，HashMap采用位桶+链表+红黑树实现，当链表长度超过阈值（8）,时，将链表转换为红黑树，这样大大减少了查找时间。

对于 HashMap 及其子类而言，它们采用 Hash 算法来决定集合中元素的存储位置。当系统开始初始化 HashMap 时，系统会创建一个长度为 capacity 的 Entry 数组，这个数组里可以存储元素的位置被称为“桶（bucket）”，每个 bucket 都有其指定索引，系统可以根据其索引快速访问该 bucket 里存储的元素。

虽然集合号称存储的是 Java 对象，但实际上并不会真正将 Java 对象放入 Set 集合中，只是在 Set 集合中保留这些对象的引用而言。也就是说：Java 集合实际上是多个引用变量所组成的集合，这些引用变量指向实际的 Java 对象。

当程序试图将多个 key-value 放入 HashMap 中时，以如下代码片段为例：

```
1.  HashMap<String , Double> map = new HashMap<String , Double>();
2.  map.put("语文" , 80.0);
3.  map.put("数学" , 89.0);
4.  map.put("英语" , 78.2);
```


HashMap 采用一种所谓的“Hash 算法”来决定每个元素的存储位置。
当程序执行 map.put("语文" , 80.0); 时，系统将调用"语文"的 hashCode() 方法得到其 hashCode 值——每个 Java 对象都有 hashCode() 方法，都可通过该方法获得它的 hashCode 值。得到这个对象的 hashCode 值之后，系统会根据该 hashCode 值来决定该元素的存储位置。
我们可以看 HashMap 类的 put(K key , V value) 方法的源代码：

```java
public V put(K key, V value)
{
 // 如果 key 为 null，调用 putForNullKey 方法进行处理
 if (key == null)
     return putForNullKey(value);
 // 根据 key 的 keyCode 计算 Hash 值
 int hash = hash(key.hashCode());
 // 搜索指定 hash 值在对应 table 中的索引
     int i = indexFor(hash, table.length);
 // 如果 i 索引处的 Entry 不为 null，通过循环不断遍历 e 元素的下一个元素
 for (Entry<K,V> e = table[i]; e != null; e = e.next)
 {
     Object k;
     // 找到指定 key 与需要放入的 key 相等（hash 值相同
     // 通过 equals 比较放回 true）
     if (e.hash == hash && ((k = e.key) == key
         || key.equals(k)))
     {
         V oldValue = e.value;
         e.value = value;
         e.recordAccess(this);
         return oldValue;
     }
 }
 // 如果 i 索引处的 Entry 为 null，表明此处还没有 Entry
 modCount++;
 // 将 key、value 添加到 i 索引处
 addEntry(hash, key, value, i);
 return null;
}
```

HashMap 基于 hashing 原理，我们通过 put() 和 get() 方法储存和获取对象。当我们将键值对传递给 put() 方法时，它调用键对象的 hashCode() 方法来计算 hashcode，然后找到 bucket桶位来储存值对象。当获取对象时，通过键对象的 equals() 方法找到正确的键值对，然后返回值对象。HashMap 使用链表来解决碰撞问题，当发生碰撞了，对象将会储存在链表的下一个节点中。 HashMap 在每个链表节点中储存键值对对象。

十一.ConcurrentHashMap 的工作原理及代码实现

ConcurrentHashMap 采用了非常精妙的"分段锁"策略，ConcurrentHashMap 的主干是个 Segment 数组。Segment 继承了 ReentrantLock，所以它就是一种可

重入锁（ReentrantLock)。在 ConcurrentHashMap，一个 Segment 就是一个子哈希表，Segment 里维护了一个 HashEntry 数组，并发环境下，对于不同 Segment 的数据进行操作是不用考虑锁竞争的。

十二.HashMap 和 ConcurrentHashMap 的区别

- 放入 HashMap 的元素是 key-value 对。
- 底层说白了就是散列结构。
- 要将元素放入到 HashMap 中，那么 key 的类型必须要实现 hashcode 方法，默认这个方法是根据对象的地址来计算的，接着还必须覆盖对象的 equals() 方法。
- ConcurrentHashMap 对整个桶数组进行了分段，而 HashMap 则没有
- ConcurrentHashMap 在每一个分段上都用锁进行保护，从而让锁的粒度更精细一些，并发性能更好，而 HashMap 没有锁机制，不是线程安全的

十三.HashSet 和 HashMap 区别

HashMap	HashS
HashMap 实现了 Map 接口	HashSet 实现了 Set 接口
HashMap 储存键值对	HashSet 仅仅存储对象
使用 put() 方法将元素放入 map 中	使用 add() 方法将元素放入 set 中
HashMap 中使用键对象来计算 hashcode 值	HashSet 使用成员对象来计算 has 值，对于两个对象来说 hashcode 方法用来判断对象的相等性，如果 false
HashMap 比较快，因为是使用唯一的键来获取对象	HashSet 较 HashMap 来说比较慢

十四.HashMap 和 HashTable 的区别

- HashMap 几乎可以等价于 HashTable，除了 HashMap 是非 synchronized 的，并可以接受 null(HashMap 可以接受为 null 的键值 (key) 和值 (value),

而 HashTable 则不行)。

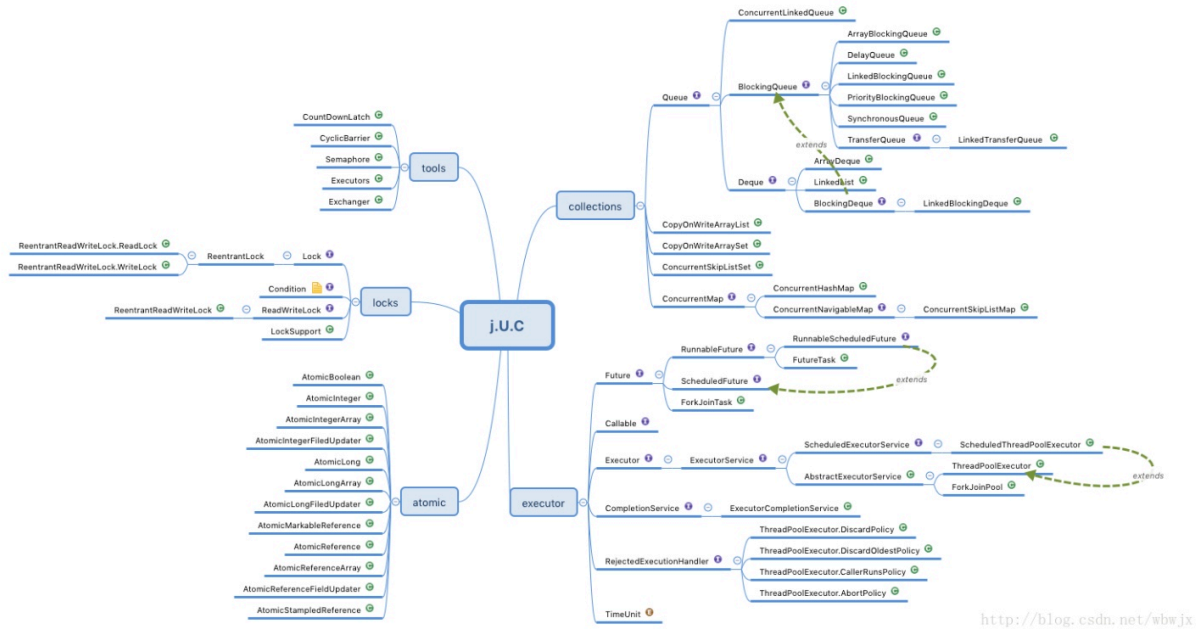
- HashMap 是非 synchronized，而 HashTable 是 synchronized，这意味着 HashTable 是线程安全的，多个线程可以共享一个 HashTable；而如果没有正确的同步的话，多个线程是不能共享 HashMap 的。Java 5 提供了 ConcurrentHashMap，它是 HashTable 的替代，比 HashTable 的扩展性更好。
- 另一个区别是 HashMap 的迭代器 (Iterator) 是 fail-fast 迭代器，而 HashTable 的 enumerator 迭代器不是 fail-fast 的。所以当有其它线程改变了 HashMap 的结构（增加或者移除元素），将会抛出 ConcurrentModificationException，但迭代器本身的 remove() 方法移除元素则不会抛出 ConcurrentModificationException 异常。但这并不是一个一定发生的行为，要看 JVM。这条同样也是 Enumeration 和 Iterator 的区别。
- 由于 HashTable 是线程安全的也是 synchronized，所以在单线程环境下它比 HashMap 要慢。如果你不需要同步，只需要单一线程，那么使用 HashMap 性能要好过 HashTable。
- HashMap 不能保证随着时间的推移 Map 中的元素次序是不变的。

十五.Java的concurrent包中有哪些API?

java.util.concurrent 包是专为 Java并发编程 而设计的包。包下的所有类可以分为如下几大类：

- locks部分：显式锁(互斥锁和速写锁)相关；
- atomic部分：原子变量类相关，是构建非阻塞算法的基础；
- executor部分：线程池相关；
- collections部分：并发容器相关；
- tools部分：同步工具相关，如信号量、闭锁、栅栏等功能；

类图结构：



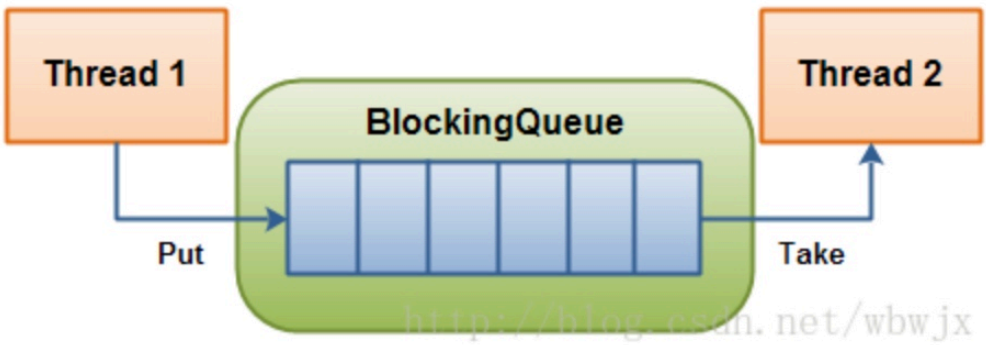
脑图地址： http://www.xmind.net/m/tJy5，感谢深入浅出 Java Concurrency，此脑图在这篇基础上修改而来。

1.BlockingQueue

此接口是一个线程安全的 存取实例的队列。

使用场景

BlockingQueue通常用于一个线程生产对象，而另外一个线程消费这些对象的场景。



注意事项：

此队列是有限的,如果队列到达临界点，Thread1就会阻塞，直到Thread2从队列中拿走一个对象。如果队列是空的，Thread2会阻塞，直到Thread1把一个对象丢进队列。

相关方法

BlockingQueue中包含了如下操作方法：

	Throws Exception	Special Value	Blocks	Times Out
Insert	add(o)	offer(o)	put(o)	offer(o, timeout, timeunit)
Remove	remove(o)	poll()	take()	poll(timeout, timeunit)
Examine	element()	peek()		

名词解释：

- Throws Exception: 如果试图的操作无法立即执行，抛一个异常。
- Special Value: 如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。
- Blocks: 如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。
- Times Out: 如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

注意事项：

无法插入 null，否则会抛出一个 NullPointerException。
队列这种数据结构，导致除了获取开始和结尾位置的其他对象的效率都不高，虽然可通过remove(o)来移除任一对象。

实现类

因为是一个接口，所以我们必须使用一个实现类来使用它，有如下实现类：
ArrayBlockingQueue： 数组阻塞队列
DelayQueue： 延迟队列
LinkedBlockingQueue： 链阻塞队列
PriorityBlockingQueue： 具有优先级的阻塞队列
SynchronousQueue： 同步队列

使用示例：

见： BlockingQueue

2.ArrayBlockingQueue

ArrayBlockingQueue 是一个有界的阻塞队列
内部实现是将对象放到一个数组里。数组有个特性：一旦初始化，大小就无法修改。因此无法修改ArrayBlockingQueue初始化时的上限。
ArrayBlockingQueue 内部以 FIFO(先进先出)的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。

3.DelayQueue

DelayQueue 对元素进行持有直到一个特定的延迟到期。注入其中的元素必须实现 java.util.concurrent.Delayed 接口:

```
public interface Delayed extends Comparable<Delayed< {
    public long getDelay(TimeUnit timeUnit);  // 返回将要延迟的时间段
}
```

在每个元素的 getDelay() 方法返回的值的时间段之后才释放掉该元素。如果返回的是 0 或者负值，延迟将被认为过期，该元素将会在 DelayQueue 的下一次 take 被调用的时候被释放掉。
Delayed 接口也继承了 java.lang.Comparable 接口，Delayed对象之间可以进行对比。这对DelayQueue 队列中的元素进行排序时有用，因此它们可以根据过期时间进行有序释放。

4.LinkedBlockingQueue

内部以一个链式结构(链接节点)对其元素进行存储 。

可以选择一个上限。如果没有定义上限，将使用 Integer.MAX_VALUE 作为上限。
内部以 FIFO(先进先出)的顺序对元素进行存储。

5.PriorityBlockingQueue

一个无界的并发队列，它使用了和类 java.util.PriorityQueue 一样的排序规则。
无法向这个队列中插入 null 值。
插入到 其中的元素必须实现 java.lang.Comparable 接口。
对于具有相等优先级(compare() == 0)的元素并不强制任何特定行为。
从一个 PriorityBlockingQueue 获得一个 Iterator 的话，该 Iterator 并不能保证它对元素的遍历是以优先级为序的。

6.SynchronousQueue

一个特殊的队列，它的内部同时只能够容纳单个元素。

如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。
如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。

7.BlockingDeque

此接口表示一个线程安全放入和提取实例的双端队列。

使用场景

==通常用在一个线程既是生产者又是消费者的时候。 ==

注意事项

如果双端队列已满，插入线程将被阻塞，直到一个移除线程从该队列中移出了一个元素。如果双端队列为空，移除线程将被阻塞，直到一个插入线程向该队列插入了一个新元素。

相关方法

	Throws Exception	Special Value	Blocks	Times Out
Insert	addFirst(o)	offerFirst(o)	putFirst(o)	offerFirst(o, timeout, timeunit)
Remove	removeFirst(o)	pollFirst(o)	takeFirst(o)	pollFirst(timeout, timeunit)
Examine	getFirst(o)	peekFirst(o)		

	Throws Exception	Special Value	Blocks	Times Out
Insert	addLast(o)	offerLast(o)	putLast(o)	offerLast(o, timeout, timeunit)
Remove	removeLast(o)	pollLast(o)	takeLast(o)	pollLast(timeout, timeunit)
Examine	getLast(o)	peekLast(o)		

注意事项

关于方法的处理方式和上节一样。
BlockingDeque 接口继承自 BlockingQueue 接口，可以用其中定义的方法。

实现类
LinkedBlockingDeque： 链阻塞双端队列

8.LinkedBlockingDeque

LinkedBlockingDeque 是一个双端队列，可以从任意一端插入或者抽取元素的队列。

在它为空的时候，一个试图从中抽取数据的线程将会阻塞，无论该线程是试图从哪一端抽取数据。

9.ConcurrentMap

一个能够对别人的访问(插入和提取)进行并发处理的 java.util.Map接口
ConcurrentMap 除了从其父接口 java.util.Map 继承来的方法之外还有一些额外的原子性方法。

实现类

因为是接口，必须用实现类来使用它，其实现类为ConcurrentHashMap

ConcurrentHashMap与HashTable比较

更好的并发性能，在你从中读取对象的时候 ConcurrentHashMap 并不会把整个 Map 锁住，只是把 Map 中正在被写入的部分进行锁定。
在被遍历的时候，即使是 ConcurrentHashMap 被改动，它也不会抛 ConcurrentModificationException。

10.ConcurrentNavigableMap

一个支持并发访问的 java.util.NavigableMap，它还能让它的子 map 具备并发访问的能力。

11.headMap

headMap(T toKey) 方法返回一个包含了小于给定 toKey 的 key 的子 map。

12.tailMap

tailMap(T fromKey) 方法返回一个包含了不小于给定 fromKey 的 key 的子 map。

13.subMap

subMap() 方法返回原始 map 中，键介于 from(包含) 和 to (不包含) 之间的子 map。

更多方法
* descendingKeySet()
* descendingMap()
* navigableKeySet()

14.CountDownLatch

CountDownLatch 是一个并发构造，它允许一个或多个线程等待一系列指定操作的完成。

CountDownLatch 以一个给定的数量初始化。countDown() 每被调用一次，这一数量就减一。
通过调用 await() 方法之一，线程可以阻塞等待这一数量到达零。

15.CyclicBarrier

CyclicBarrier 类是一种同步机制，它能够对处理一些算法的线程实现同步。

更多实例参考： CyclicBarrier

16.Exchanger

Exchanger 类表示一种两个线程可以进行互相交换对象的会和点。

更多实例参考： Exchanger

17.Semaphore

Semaphore 类是一个计数信号量。具备两个主要方法：
acquire()
release()

```
每调用一次 acquire()，一个许可会被调用线程取走。
每调用一次 release()，一个许可会被返还给信号量。
```

Semaphore 用法

保护一个重要(代码)部分防止一次超过 N 个线程进入。
在两个线程之间发送信号。

保护重要部分

如果你将信号量用于保护一个重要部分，试图进入这一部分的代码通常会首先尝试获得一个许可，然后才能进入重要部分(代码块)，执行完之后，再把许可释放掉。

```
Semaphore semaphore = new Semaphore(1);
//critical section
semaphore.acquire();
...
semaphore.release();
```

在线程之间发送信号

如果你将一个信号量用于在两个线程之间传送信号，通常你应该用一个线程调用 acquire() 方法，而另一个线程调用 release() 方法。

如果没有可用的许可，acquire() 调用将会阻塞，直到一个许可被另一个线程释放出来。
如果无法往信号量释放更多许可时，一个 release() 调用也会阻塞。
公平性
无法担保掉第一个调用 acquire() 的线程会是第一个获得一个许可的线程。

可以通过如下来强制公平：

```
Semaphore semaphore = new Semaphore(1, true);
```

需要注意，强制公平会影响到并发性能，建议不使用。

18.ExecutorService

这里之前有过简单的总结： Java 中几种常用的线程池

存在于 java.util.concurrent 包里的 ExecutorService 实现就是一个线程池实现。

实现类

此接口实现类包括：
ScheduledThreadPoolExecutor： 通过Executors.newScheduledThreadPool(10)创建的
ThreadPoolExecutor: 除了第一种的其他三种方式创建的

相关方法

- execute(Runnable)：
- 无法得知被执行的 Runnable 的执行结果
- submit(Runnable)：
- 返回一个 Future 对象，可以知道Runnable 是否执行完毕。
- submit(Callable)：
- Callable 实例除了它的 call() 方法能够返回一个结果，通过Future可以获取。
- invokeAny(…)：
- 传入一系列的 Callable 或者其子接口的实例对象，无法保证返回的是哪个 Callable 的结果 ，只能表明其中一个已执行结束。
- 如果其中一个任务执行结束(或者抛了一个异常)，其他 Callable 将被取消。
- invokeAll(…)：
- 返回一系列的 Future 对象，通过它们你可以获取每个 Callable 的执行结果。

关闭ExecutorService

shutdown()： 不会立即关闭，但它将不再接受新的任务
shutdownNow()： 立即关闭

ThreadPoolExecutor

ThreadPoolExecutor 使用其内部池中的线程执行给定任务(Callable 或者 Runnable)。
ScheduledExecutorService（接口，其实现类为ScheduledThreadPoolExecutor）
ScheduledExecutorService能够将任务延后执行，或者间隔固定时间多次执行。
ScheduledExecutorService中的 任务由一个工作者线程异步执行，而不是由提交任务给 ScheduledExecutorService 的那个线程执行。

相关方法

- schedule (Callable task, long delay, TimeUnit timeunit)：
- Callable 在给定的延迟之后执行，并返回结果。
- schedule (Runnable task, long delay, TimeUnit timeunit)

- 除了 Runnable 无法返回一个结果之外，和第一个方法类似。
- scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
- 这一方法规划一个任务将被定期执行。该任务将会在首个 initialDelay 之后得到执行，然后每个 period 时间之后重复执行。
- period 被解释为前一个执行的开始和下一个执行的开始之间的间隔时间。
- scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)
- 和上一个方法类似，只是period 则被解释为前一个执行的结束和下一个执行的结束之间的间隔。

19.ForkJoinPool

ForkJoinPool 在 Java 7 中被引入。它和 ExecutorService 很相似，除了一点不同。ForkJoinPool 让我们可以很方便地把任务分裂成几个更小的任务，这些分裂出来的任务也将会提交给 ForkJoinPool。

用法参考：Java Fork and Join using ForkJoinPool

20.Lock

Lock 是一个类似于 synchronized 块的线程同步机制。但是 Lock 比 synchronized 块更加灵活、精细。

实现类

Lock是一个接口，其实现类包括：
ReentrantLock

示例:

```
Lock lock = new ReentrantLock();
lock.lock();
//critical section
lock.unlock();
```

调用lock() 方法之后，这个 lock 实例就被锁住啦。
当lock示例被锁后，任何其他再过来调用 lock() 方法的线程将会被阻塞住，直到调用了unlock() 方法。

unlock() 被调用了，lock 对象解锁了，其他线程可以对它进行锁定了。

Lock 和 synchronized区别

- synchronized 代码块不能够保证进入访问等待的线程的先后顺序。
- 你不能够传递任何参数给一个 synchronized 代码块的入口。因此，对于 synchronized 代码块的访问等待设置超时时间是不可能的事情。
- synchronized 块必须被完整地包含在单个方法里。而一个 Lock 对象可以把它的 lock() 和 unlock() 方法的调用放在不同的方法里。

21.ReadWriteLock

读写锁一种先进的线程锁机制。

允许多个线程在同一时间对某特定资源进行读取，
但同一时间内只能有一个线程对其进行写入。

实现类

ReentrantReadWriteLock

规则

如果没有任何写操作锁定，那么可以有多个读操作锁定该锁
如果没有任何读操作或者写操作，只能有一个写线程对该锁进行锁定。

示例：

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
readWriteLock.readLock().lock();
    // multiple readers can enter this section
    // if not locked for writing, and not writers waiting
    // to lock for writing.
readWriteLock.readLock().unlock();
readWriteLock.writeLock().lock();
    // only one writer can enter this section,
    // and only if no threads are currently reading.
readWriteLock.writeLock().unlock();
```

22.更多原子性包装类

位于 atomic包下，包含一系列原子性变量。

AtomicBoolean
AtomicInteger
AtomicLong
AtomicReference
…