集合及concurrent并发包总结 - Echo的一亩三分地

集合及concurrent并发包总结

Echo_me 发表于4年前

1.集合包

集合包最常用的有Collection和Map两个接口的实现类，Colleciton用于存放多个单对象，Map用于存放Key-Value形式的键值对。

Collection中最常用的又分为两种类型的接口：List和Set，两者最明显的差别为List支持放入重复的元素，而Set不支持。

List最常用的实现类有：ArrayList、LinkedList、Vector及Stack；Set接口常用的实现类有：HashSet、TreeSet。

1.1 ArrayList

ArrayList基于数组方式实现，默认构造器通过调用ArrayList(int)来完成创建，传入的值为10，实例化了一个Object数组，并将此数组赋给了当前实例的elementData属性，此Object数组的大小即为传入的initialCapacity，因此调用空构造器的情况下会创建一个大小为10的Object数组。

插入对象：add(E)

基于已有元素数量加1作为名叫minCapacity的变量，比较此值和Object数组的大小，若大于数组值，那么先将当前Object数组值赋给一个数组对象，接着产生一个鑫的数组容量值。此值的计算方法为当前数组值*1.5+1，如得出的容量值仍然小于minCapacity，那么就以minCapacity作为新的容量值，调用Arrays.copyOf来生成新的数组对象。

还提供了add(int,E)这样的方法将元素直接插入指定的int位置上，将目前index及其后的数据都往后挪一位，然后才能将指定的index位置的赋值为传入的对象，这种方式要多付出一次复制数组的代价。还提供了addAll

删除对象：remove(E)

这里调用了faseRemove方法将index后的对象往前复制一位，并将数组中的最后一个元素的值设置为null，即释放了对此对象的引用。 还提供了remove(int)方法来删除指定位置的对象，remove(int)的实现比remove(E)多了一个数组范围的检测，但少了对象位置的查找，因此性能会更好。

获取单个对象：get(int)

遍历对象：iterator()

判断对象是否存在：contains(E)

总结：

1，ArrayList基于数组方式实现，无容量的限制；

2，ArrayList在执行插入元素时可能要扩容，在删除元素时并不会减小数组的容量（如希望相应的缩小数组容量，可以调用ArrayList的trimToSize()），在查找元素时要遍历数组，对于非null的元素采取equals的方式寻找；

3，ArrayList是非线程安全的。

1.2 LinkedList

LinkedList基于双向链表机制，所谓双向链表机制，就是集合中的每个元素都知道其前一个元素及其后一个元素的位置。在LinkedList中，以一个内部的Entry类来代表集合中的元素，元素的值赋给element属性，Entry中的next属性指向元素的后一个元素，Entry中的previous属性指向元素的前一个元素，基于这样的机制可以快速实现集合中元素的移动。

总结：

1，LinkedList基于双向链表机制实现；

2，LinkedList在插入元素时，须创建一个新的Entry对象，并切换相应元素的前后元素的引用；在查找元素时，须遍历链表；在删除元素时，要遍历链

表，找到要删除的元素，然后从链表上将此元素删除即可，此时原有的前后元素改变引用连在一起；

3，LinkedList是非线程安全的。

1.3 Vector

其add、remove、get(int)方法都加了synchronized关键字，默认创建一个大小为10的Object数组，并将capacityIncrement设置为0。容量扩充策略：如果capacityIncrement大于0，则将Object数组的大小扩大为现有size加上capacityIncrement的值；如果capacity等于或小于0，则将Object数组的大小扩大为现有size的两倍，这种容量的控制策略比ArrayList更为可控。

Vector是基于Synchronized实现的线程安全的ArrayList，但在插入元素时容量扩充的机制和ArrayList稍有不同，并可通过传入capacityIncrement来控制容量的扩充。

1.4 Stack

Stack继承于Vector，在其基础上实现了Stack所要求的后进先出(LIFO)的弹出与压入操作，其提供了push、pop、peek三个主要的方法：

push操作通过调用Vector中的addElement来完成；

pop操作通过调用peek来获取元素，并同时删除数组中的最后一个元素；

peek操作通过获取当前Object数组的大小，并获取数组上的最后一个元素。

1.5 HashSet

默认构造创建一个HashMap对象

add(E)：调用HashMap的put方法来完成此操作，将需要增加的元素作为Map中的key，value则传入一个之前已创建的Object对象。

remove(E)：调用HashMap的remove(E)方法完成此操作。

contains(E)：HashMap的containsKey

iterator()：调用HashMap的keySet的iterator方法。

HashSet不支持通过get(int)获取指定位置的元素，只能自行通过iterator方法来获取。

总结：

1，HashSet基于HashMap实现，无容量限制；

2，HashSet是非线程安全的。

1.6 TreeSet

TreeSet和HashSet的主要不同在于TreeSet对于排序的支持，TreeSet基于TreeMap实现。

1.7 HashMap

HashMap空构造，将loadFactor设为默认的0.75，threshold设置为12，并创建一个大小为16的Entry对象数组。

基于数组+链表的结合体(链表散列)实现，将key-value看成一个整体，存放于Entity[]数组，put的时候根据key hash后的hashcode和数组length-1按位与的结果值判断放在数组的哪个位置，如果该数组位置上若已经存放其他元素，则在这个位置上的元素以链表的形式存放。如果该位置上没有元素则直接存放。

当系统决定存储HashMap中的key-value对时，完全没有考虑Entry中的value，仅仅只是根据key来计算并决定每个Entry的存储位置。我们完全可以把Map集合中的value当成key的附属，当系统决定了key的存储位置之后，value随之保存在那里即可。get取值也是根据key的hashCode确定在数组的位置，在根据key的equals确定在链表处的位置。

```
while (capacity < initialCapacity)
     capacity <<= 1;
```

以上代码保证了初始化时HashMap的容量总是2的n次方，即底层数组的长度总是为2的n次方。它通过h & (table.length -1) 来得到该对象的保存位，若

length为奇数值，则与运算产生相同结果，便会形成链表，尽可能的少出现链表才能提升hashMap的效率，所以这是hashMap速度上的优化。

扩容resize():

当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容，而在HashMap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。那么HashMap什么时候进行扩容呢？当HashMap中的元素个数超过数组大小*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。

负载因子衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。

HashMap的实现中，通过threshold字段来判断HashMap的最大容量。

threshold就是在此loadFactor和capacity对应下允许的最大元素数目，超过这个数目就重新resize，以降低实际的负载因子。默认的的负载因子0.75是对空间和时间效率的一个平衡选择。

initialCapacity*2，成倍扩大容量，HashMap(int initialCapacity, float loadFactor)：以指定初始容量、指定的负载因子创建一个 HashMap。不设定参数，则初始容量值为16，默认的负载因子为0.75，不宜过大也不宜过小，过大影响效率，过小浪费空间。扩容后需要重新计算每个元素在数组中的位置，是一个非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

HashTable数据结构的原理大致一样，区别在于put、get时加了同步关键字，而且HashTable不可存放null值。

在高并发时可以使用ConcurrentHashMap，其内部使用锁分段技术，维持这锁Segment的数组，在数组中又存放着Entity[]数组，内部hash算法将数据较均匀分布在不同锁中。

总结：

1，HashMap采用数组方式存储key、value构成的Entry对象，无容量限制；

2，HashMap基于key hash寻找Entry对象存放到数组的位置，对于hash冲突采用链表的方式解决；

3，HashMap在插入元素时可能会扩大数组的容量，在扩大容量时须要重新计算hash，并复制对象到新的数组中；

4，HashMap是非线程安全的。

详细说明：http://zhangshixi.iteye.com/blog/672697

1.8 TreeMap

TreeMap基于红黑树的实现，因此它要求一定要有key比较的方法，要么传入Comparator实现，要么key对象实现Comparable借口。在put操作时，基于红黑树的方式遍历，基于comparator来比较key应放在树的左边还是右边，如找到相等的key，则直接替换掉value。

2.并发包

jdk5.0一很重要的特性就是增加了并发包java.util.concurrent.*，在说具体的实现类或接口之前，这里先简要说下Java内存模型、volatile变量及AbstractQueuedSynchronizer(以下简称AQS同步器)，这些都是并发包众多实现的基础。

Java内存模型

描述了线程内存与主存见的通讯关系。定义了线程内的内存改变将怎样传递到其他线程的规则，同样也定义了线程内存与主存进行同步的细节，也描述了哪些操作属于原子操作及操作间的顺序。

代码顺序规则：

一个线程内的每个动作happens-before同一个线程内在代码顺序上在其后

的所有动作.

volatile变量规则：

对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入.

传递性：

如果A happens-before B, B happens-before C, 那么A happens-before C.

volatile

当我们声明共享变量为volatile后，对这个变量的读/写将会很特别。理解volatile特性的一个好方法是：把对volatile变量的单个读/写，看成是使用同一个监视器锁对这些单个读/写操作做了同步。

监视器锁的happens-before规则保证释放监视器和获取监视器的两个线程之间的内存可见性，这意味着对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。

简而言之，volatile变量自身具有下列特性：

- 可见性。对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。
- 原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性。

volatile写的内存语义如下：

- 当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存。

volatile读的内存语义如下：

- 当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。

线程接下来将从主内存中读取共享变量。

下面对volatile写和volatile读的内存语义做个总结：

- 线程A写一个volatile变量，实质上是线程A向接下来将要读这个volatile变量的某个线程发出了（其对共享变量所在修改的）消息。
- 线程B读一个volatile变量，实质上是线程B接收了之前某个线程发出的（在写这个volatile变量之前对共享变量所做修改的）消息。
- 线程A写一个volatile变量，随后线程B读这个volatile变量，这个过程实质上是线程A通过主内存向线程B发送消息。

锁释放-获取与volatile的读写具有相同的内存语义，

锁释放的内存语义如下：

当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存。

锁获取的内存语义如下：

当线程获取锁时，JMM会把该线程对应的本地内存置为无效，从而使得被监视器保护的临界区代码必须要从主内存中读取共享变量。

下面对锁释放和锁获取的内存语义做个总结：

- 线程A释放一个锁，实质上是线程A向接下来将要获取这个锁的某个线程发出了（线程A对共享变量所做修改的）消息。
- 线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。
- 线程A释放锁，随后线程B获取这个锁，这个过程实质上是线程A通过主内存向线程B发送消息。

示例：

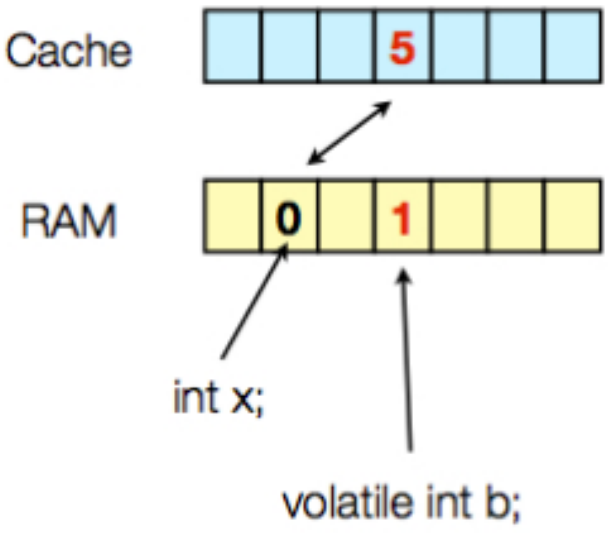
```
class VolatileExample {
```

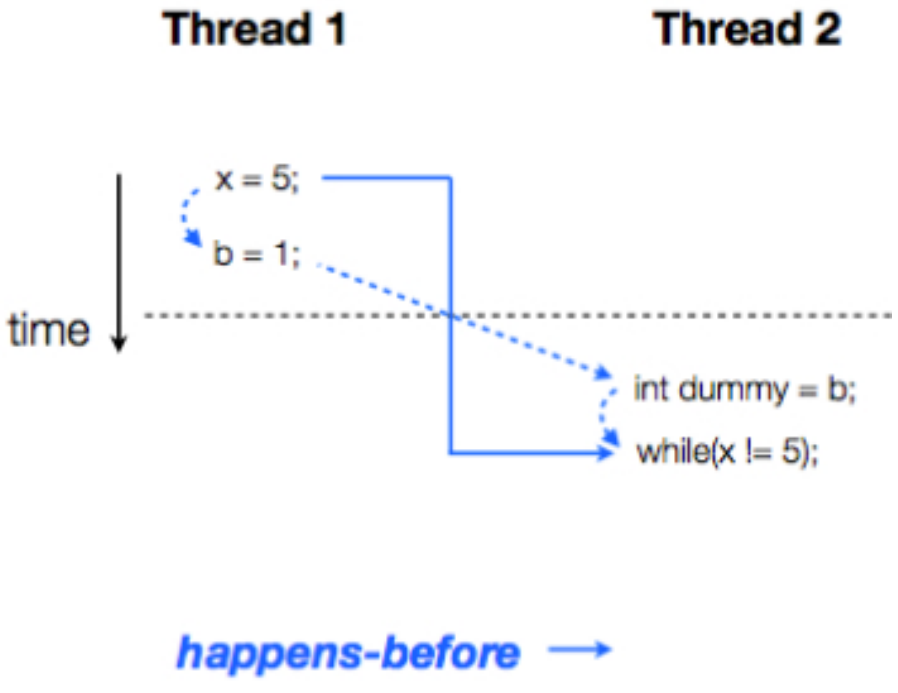
```
int x = 0;
volatile int b = 0;

private void write() {
    x = 5;
    b = 1;
}

private void read() {
    int dummy = b;
    while (x != 5) {
    }
}

public static void main(String[] args) throws Exception {
    final VolatileExample example = new VolatileExample();
    Thread thread1 = new Thread(new Runnable() {
        public void run() {
            example.write();
        }
    });
    Thread thread2 = new Thread(new Runnable() {
        public void run() {
            example.read();
        }
    });
    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();
  }
}
```





若thread1先于thread2执行，则程序执行流程分析如上图所示，thread2读的结果是dummy=1，x=5所以不会进入死循环。

但并不能保证两线程的执行顺序，若thread2先于thread1执行，则程序在两线程join中断之前的结果为：因为b变量的类型是volatile，故thread1写之后，thread2即可读到b变量的值发生变化，

而x是普通变量，故最后情况是dummy=1，但thread2的读操作因为x=0而进入死循环中。

在JSR-133之前的旧Java内存模型中，虽然不允许volatile变量之间重排序，但旧的Java内存模型仍然会允许volatile变量与普通变量之间重排序。JSR-133则增强了volatile的内存语义：严格限制编译器（在编译期）和处理器（在运行期）对volatile变量与普通变量的重排序，确保volatile的写-读和监视器的释放-获取一样，具有相同的内存语义。限制重排序是通过内存屏障实现的，具体可见JMM的描述。

由于volatile仅仅保证对单个volatile变量的读/写具有原子性，而监视器锁的互斥执行的特性可以确保对整个临界区代码的执行具有原子性。在功能上，监视器锁比volatile更强大；在可伸缩性和执行性能上，volatile更有优势。如果读者想在程序中用volatile代替监视器锁，请一定谨慎。

AbstractQueuedSynchronizer (AQS)

AQS使用一个整型的volatile变量（命名为state）来维护同步状态，这是接下来实现大部分同步需求的基础。提供了一个基于FIFO队列，可以用于构建锁或者其他相关同步装置的基础框架。使用的方法是继承，子类通过继承同步器并需要实现它的方法来管理其状态，管理的方式就是通过类似acquire和release的方式来操纵状态。然而多线程环境中对状态的操纵必须确保原子性，因此子类对于状态的把握，需要使用这个同步器提供的以下三个方法对状态进行操作：

- java.util.concurrent.locks.AbstractQueuedSynchronizer.getState()
- java.util.concurrent.locks.AbstractQueuedSynchronizer.setState(int)
- java.util.concurrent.locks.AbstractQueuedSynchronizer.compareAndSetState(int, int)

子类推荐被定义为自定义同步装置的内部类，同步器自身没有实现任何同步接口，它仅仅是定义了若干acquire之类的方法来供使用。该同步器即可以作为排他模式也可以作为共享模式，当它被定义为一个排他模式时，其他线程对其的获取就被阻止，而共享模式对于多个线程获取都可以成功。

同步器是实现锁的关键，利用同步器将锁的语义实现，然后在锁的实现中聚合同步器。可以这样理解：锁的API是面向使用者的，它定义了与锁交互的公共行为，而每个锁需要完成特定的操作也是透过这些行为来完成的（比如：可以允许两个线程进行加锁，排除两个以上的线程），但是实现是依托给同步器来完成；同步器面向的是线程访问和资源控制，它定义了线程对资源是否能够获取以及线程的排队等操作。锁和同步器很好的隔离了二者所需要关注的领域，严格意义上讲，同步器可以适用于除了锁以外的其他同步设施上（包括锁）。
同步器的开始提到了其实现依赖于一个FIFO队列，那么队列中的元素Node就是保存着线程引用和线程状态的容器，每个线程对同步器的访问，都可以看做是队列中的一个节点。

对于一个独占锁的获取和释放有如下伪码可以表示：

获取一个排他锁

```
while(获取锁) {
        if (获取到) {
                退出while循环
        } else {
                if(当前线程没有入队列) {
                        那么入队列
                }
                阻塞当前线程
        }
}
```

释放一个排他锁

```
if (释放成功) {
        删除头结点
        激活原头结点的后继节点
}
```

示例:

下面通过一个排它锁的例子来深入理解一下同步器的工作原理，而只有掌握同步器的工作原理才能更加深入了解其他的并发组件。

排他锁的实现，一次只能一个线程获取到锁:

```
public class Mutex implements Lock, java.io.Serializable {
    // 内部类，自定义同步器
    private static class Sync extends AbstractQueuedSynchronizer {
      // 是否处于占用状态
      protected boolean isHeldExclusively() {
        return getState() == 1;
      }
      // 当状态为0的时候获取锁
      public boolean tryAcquire(int acquires) {
        assert acquires == 1; // Otherwise unused
        if (compareAndSetState(0, 1)) {
          setExclusiveOwnerThread(Thread.currentThread());
          return true;
        }
        return false;
      }
      // 释放锁，将状态设置为0
      protected boolean tryRelease(int releases) {
        assert releases == 1; // Otherwise unused
        if (getState() == 0) throw new IllegalMonitorStateException();
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
      }
      // 返回一个Condition，每个condition都包含了一个condition队列
      Condition newCondition() { return new ConditionObject(); }
    }
    // 仅需要将操作代理到Sync上即可
    private final Sync sync = new Sync();
    public void lock()                { sync.acquire(1); }
```

```
    public boolean tryLock()          { return sync.tryAcquire(1); }
    public void unlock()              { sync.release(1); }
    public Condition newCondition()   { return sync.newCondition(); }
    public boolean isLocked()         { return sync.isHeldExclusively(); }
    public boolean hasQueuedThreads() { return sync.hasQueuedThreads(); }
    public void lockInterruptibly() throws InterruptedException {
      sync.acquireInterruptibly(1);
    }
    public boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException {
      return sync.tryAcquireNanos(1, unit.toNanos(timeout));
    }
  }
```

可以看到Mutex将Lock接口均代理给了同步器的实现。使用方将Mutex构造出来后，调用lock获取锁，调用unlock将锁释放。

获取锁，acquire(int arg)的主要逻辑包括：

1. 尝试获取（调用tryAcquire更改状态，需要保证原子性）；

   在tryAcquire方法中适用了同步器提供的对state操作的方法，利用compareAndSet保证只有一个线程能够对状态进行成功修改，而没有成功修改的线程将进入sync队列排队。

2. 如果获取不到，将当前线程构造成节点Node并加入sync队列；

   进入队列的每个线程都是一个节点Node，从而形成了一个双向队列，类似CLH队列，这样做的目的是线程间的通信会被限制在较小规模（也就是两个节点左右）。

3. 再次尝试获取，如果没有获取到那么将当前线程从线程调度器上摘下，进入等待状态。

释放锁，release(int arg)的主要逻辑包括：

1. 尝试释放状态；

   tryRelease能够保证原子化的将状态设置回去，当然需要使用compareAndSet来保证。如果释放状态成功之后，就会进入后继节点的唤醒过程。

2. 唤醒当前节点的后继节点所包含的线程。

通过LockSupport的unpark方法将休眠中的线程唤醒，让其继续acquire状态。

回顾整个资源的获取和释放过程：

在获取时，维护了一个sync队列，每个节点都是一个线程在进行自旋，而依据就是自己是否是首节点的后继并且能够获取资源；

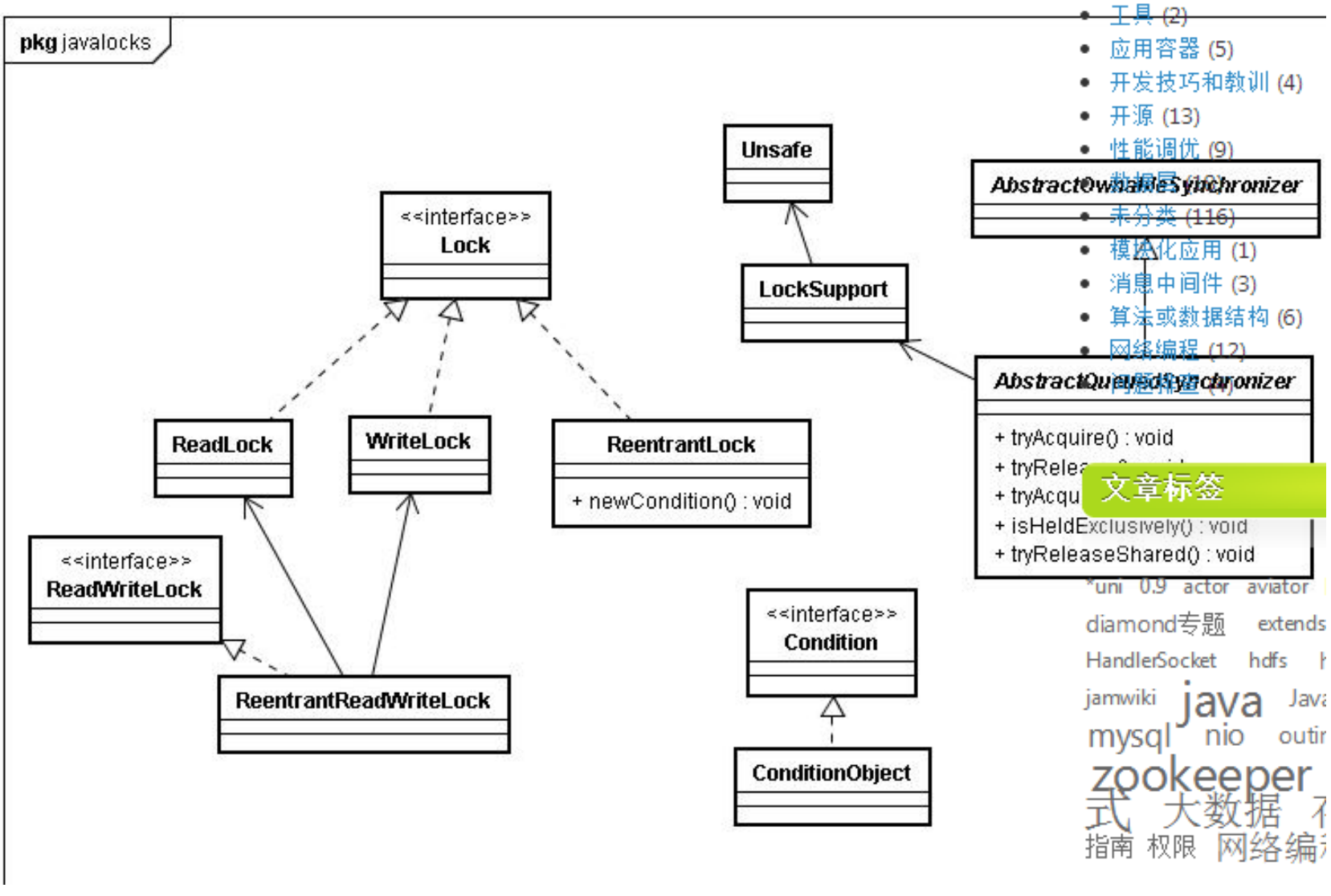
在释放时，仅仅需要将资源还回去，然后通知一下后继节点并将其唤醒。

这里需要注意，队列的维护（首节点的更换）是依靠消费者（获取时）来完成的，也就是说在满足了自旋退出的条件时的一刻，这个节点就会被设置成为首节点。

队列里的节点线程的禁用和唤醒是通过LockSupport的park()及unpark()，调用的unsafe、底层也是native的实现。

关于java lock的浅析可见：http://jm-blog.aliapp.com/?p=414

以上简单说明了下JAVA LOCKS关键要素，现在我们来看下java.util.concurrent.locks大致结构

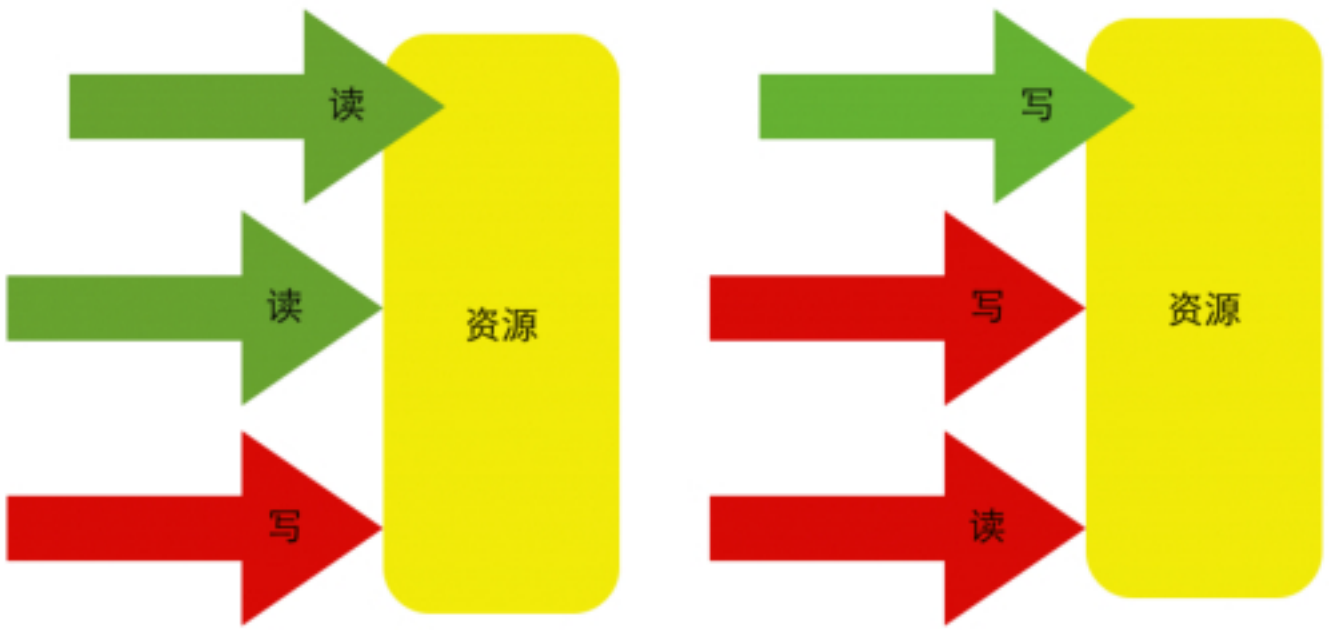


共享模式和以上的独占模式有所区别，分别调用acquireShared(int arg)和releaseShared(int arg)获取共享模式的状态。

以文件的查看为例，如果一个程序在对其进行读取操作，那么这一时刻，对这个文件的写操作就被阻塞，相反，这一时刻另一个程序对其进行同样的读操作是可以进行的。如果一个程序在对其进行写操作，

那么所有的读与写操作在这一时刻就被阻塞，直到这个程序完成写操作。

以读写场景为例，描述共享和独占的访问模式，如下图所示：



上图中，红色代表被阻塞，绿色代表可以通过。

在上述对同步器AbstractQueuedSynchronizer进行了实现层面的分析之后，我们通过一个例子来加深对同步器的理解：

设计一个同步工具，该工具在同一时刻，只能有两个线程能够并行访问，超过限制的其他线程进入阻塞状态。

对于这个需求，可以利用同步器完成一个这样的设定，定义一个初始状态，为2，一个线程进行获取那么减1，一个线程释放那么加1，状态正确的范围在[0，1，2]三个之间，当在0时，代表再有新的线程对资源进行获取时只能进入阻塞状态（注意在任何时候进行状态变更的时候均需要以CAS作为原子性保障）。由于资源的数量多于1个，同时可以有两个线程占有资源，因此需要实现tryAcquireShared和tryReleaseShared方法。

```
public class TwinsLock implements Lock {
    private final Sync  sync    = new Sync(2);

    private static final class Sync extends AbstractQueuedSynchronizer {
        private static final long   serialVersionUID    = -7889272986162341

        Sync(int count) {
            if (count <= 0) {
                throw new IllegalArgumentException("count must large than z
            }
            setState(count);
        }

        public int tryAcquireShared(int reduceCount) {
            for (;;) {
                int current = getState();
                int newCount = current - reduceCount;
                if (newCount < 0 || compareAndSetState(current, newCount))
                    return newCount;
                }
            }
        }

        public boolean tryReleaseShared(int returnCount) {
            for (;;) {
                int current = getState();
                int newCount = current + returnCount;
                if (compareAndSetState(current, newCount)) {
                    return true;
                }
            }
        }
    }

    public void lock() {
        sync.acquireShared(1);
    }

    public void lockInterruptibly() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }

    public boolean tryLock() {
        return sync.tryAcquireShared(1) >= 0;
    }

    public boolean tryLock(long time, TimeUnit unit) throws InterruptedExce
        return sync.tryAcquireSharedNanos(1, unit.toNanos(time));
```

```
    }

    public void unlock() {
        sync.releaseShared(1);
    }

    public Condition newCondition() {
        return null;
    }
}
```

这里我们编写一个测试来验证TwinsLock是否能够正常工作并达到预期。

```
public class TwinsLockTest {

    @Test
    public void test() {
        final Lock lock = new TwinsLock();

        class Worker extends Thread {
            public void run() {
                while (true) {
                    lock.lock();
                    try {
                        Thread.sleep(1000L);
            System.out.println(Thread.currentThread());
                        Thread.sleep(1000L);
                    } catch (Exception ex) {

                    } finally {
                        lock.unlock();
                    }
            }
        }
    }

    for (int i = 0; i < 10; i++) {
        Worker w = new Worker();
        w.start();
    }

    new Thread() {
        public void run() {
            while (true) {

                try {
```

```
                        Thread.sleep(200L);
                        System.out.println();
                    } catch (Exception ex) {

                    }
                }
            }
        }.start();

        try {
            Thread.sleep(20000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

上述测试用例的逻辑主要包括:

1. 打印线程

Worker在两次睡眠之间打印自身线程，如果一个时刻只能有两个线程同时访问，那么打印出来的内容将是成对出现。

2. 分隔线程

不停的打印换行，能让Worker的输出看起来更加直观。

该测试的结果是在一个时刻，仅有两个线程能够获得到锁，并完成打印，而表象就是打印的内容成对出现。

利用CAS(compare and set)是不会进行阻塞的，只会一个返回成功，一个返回失败，保证了一致性。

CAS操作同时具有volatile读和volatile写的内存语义。

AQS这部分转载于http://ifeve.com/introduce-abstractqueuedsynchronizer/

2.1 ConcurrentHashMap

ConcurrentHashMap是线程安全的HashMap的实现，默认构造同样有initialCapacity和loadFactor属性，不过还多了一个concurrencyLevel属性，三属性默认值分别为16、0.75及16。其内部使用锁分段技术，维持这锁Segment的数组，在Segment数组中又存放着Entity[]数组，内部hash算法将数据较均匀分布在不同锁中。

put操作：并没有在此方法上加上synchronized，首先对key.hashcode进行hash操作，得到key的hash值。hash操作的算法和map也不同，根据此hash值计算并获取其对应的数组中的Segment对象(继承自ReentrantLock)，接着调用此Segment对象的put方法来完成当前操作。

ConcurrentHashMap基于concurrencyLevel划分出了多个Segment来对key-value进行存储，从而避免每次put操作都得锁住整个数组。在默认的情况下，最佳情况下可允许16个线程并发无阻塞的操作集合对象，尽可能地减少并发时的阻塞现象。

get(key)

首先对key.hashCode进行hash操作，基于其值找到对应的Segment对象，调用其get方法完成当前操作。而Segment的get操作首先通过hash值和对象数组大小减1的值进行按位与操作来获取数组上对应位置的HashEntry。在这个步骤中，可能会因为对象数组大小的改变，以及数组上对应位置的HashEntry产生不一致性，那么ConcurrentHashMap是如何保证的？

对象数组大小的改变只有在put操作时有可能发生，由于HashEntry对象数组对应的变量是volatile类型的，因此可以保证如HashEntry对象数组大小发生改变，读操作可看到最新的对象数组大小。

在获取到了HashEntry对象后，怎么能保证它及其next属性构成的链表上的对象不会改变呢？这点ConcurrentHashMap采用了一个简单的方式，即HashEntry对象中的hash、key、next属性都是final的，这也就意味着没办法插入一个HashEntry对象到基于next属性构成的链表中间或末尾。这样就可以保证当获取到HashEntry对象后，其基于next属性构建的链表是不会发生变化的。

ConcurrentHashMap默认情况下采用将数据分为16个段进行存储，并且16个段分别持有各自不同的锁Segment，锁仅用于put和remove等改变集合对象

的操作，基于volatile及HashEntry链表的不变性实现了读取的不加锁。这些方式使得ConcurrentHashMap能够保持极好的并发支持，尤其是对于读远比插入和删除频繁的Map而言，而它采用的这些方法也可谓是对于Java内存模型、并发机制深刻掌握的体现。

2.2 ReentrantLock

在并发包的开始部分介绍了volatile特性及AQS同步器，而这两部分正是ReentrantLock实现的基础。通过上面AQS的介绍及原理分析，可知道是以volatile维持的int类型的state值，来判断线程是执行还是在syn队列中等待。

ReentrantLock的实现不仅可以替代隐式的synchronized关键字，而且能够提供超过关键字本身的多种功能。

这里提到一个锁获取的公平性问题，如果在绝对时间上，先对锁进行获取的请求一定被先满足，那么这个锁是公平的，反之，是不公平的，也就是说等待时间最长的线程最有机会获取锁，也可以说锁的获取是有序的。ReentrantLock这个锁提供了一个构造函数，能够控制这个锁是否是公平的。

对于公平和非公平的定义是通过对同步器AbstractQueuedSynchronizer的扩展加以实现的，也就是tryAcquire的实现上做了语义的控制。

公平和非公平性的更多原理分析见于http://ifeve.com/reentrantlock-and-fairness/

2.3 Condition

Condition是并发包中提供的一个接口，典型的实现有ReentrantLock，ReentrantLock提供了一个mewCondition的方法，以便用户在同一个锁的情况下可以根据不同的情况执行等待或唤醒动作。典型的用法可参考ArrayBlockingQueue的实现，下面来看ReentrantLock中

newCondition的实现。

ReentrantLock.newCondition()

创建一个AbstractQueuedSynchronizer的内部类ConditionObject的对象实例。

ReentrantLock.newCondition().await()

将当前线程加入此condition的等待队列中，并将线程置为等待状态。

ReentrantLock.newCondition().signal()

从此condition的等待队列中获取一个等待节点，并将节点上的线程唤醒，如果要唤醒全部等待节点的线程，则调用signalAll方法。

2.4 CopyOnWriteArrayList

CopyOnWriteArrayList是一个线程安全、并且在读操作时无锁的ArrayList，其具体实现方法如下。

CopyOnWriteArrayList()

和ArrayList不同，此步的做法为创建一个大小为0的数组。

add(E)

add方法并没有加上synchronized关键字，它通过使用ReentrantLock来保证线程安全。此处和ArrayList的不同是每次都会创建一个新的Object数组，此数组的大小为当前数组大小加1，将之前数组中的内容复制到新的数组中，并将

新增加的对象放入数组末尾，最后做引用切换将新创建的数组对象赋值给全局的数组对象。

remove(E)

和add方法一样，此方法也通过ReentrantLock来保证其线程安全，但它和ArrayList删除元素采用的方式并不一样。

首先创建一个比当前数组小1的数组，遍历新数组，如找到equals或均为null的元素，则将之后的元素全部赋值给新的数组对象，并做引用切换，返回true；如未找到，则将当前的元素赋值给新的数组对象，最后特殊处理数组中的最后

一个元素，如最后一个元素等于要删除的元素，即将当前数组对象赋值为新

创建的数组对象，完成删除操作，如最后一个元素也不等于要删除的元素，那么返回false。

此方法和ArrayList除了锁不同外，最大的不同在于其复制过程并没有调用System的arrayCopy来完成，理论上来说会导致性能有一定下降。

get(int)

此方法非常简单，直接获取当前数组对应位置的元素，这种方法是没有加锁保护的，因此可能会出现读到脏数据的现象。但相对而言，性能会非常高，对于写少读多且脏数据影响不大的场景而言是不错的选择。

iterator()

调用iterator方法后创建一个新的COWIterator对象实例，并保存了一个当前数组的快照，在调用next遍历时则仅对此快照数组进行遍历，因此遍历此list时不会抛出ConcurrentModificatiedException。

与ArrayList的性能对比，在读多写少的并发场景中，较之ArrayList是更好的选择，单线程以及多线程下增加元素及删除元素的性能不比ArrayList好

2.5 CopyOnWriteArraySet

CopyOnWriteArraySet基于CopyOnWriteArrayList实现，其唯一的不同是在add时调用的是CopyOnWriteArrayList的addIfAbsent方法。保证了无重复元素，但在add时每次都要进行数组的遍历，因此性能会略低于上个。

2.6 ArrayBlockingQueue

2.7 ThreadPoolExecutor

与每次需要时都创建线程相比，线程池可以降低创建线程的开销，在线程执行结束后进行的是回收操作，提高对线程的复用。Java中主要使用的线程池是ThreadPoolExecutor，此外还有定时的线程池ScheduledThreadPoolExecutor。

Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是

ExecutorService。

比较重要的几个类：

ExecutorService	真正的线程池接口
ScheduledExecutorService	和Time/TimeTask类似，解决需要任务重复执行的问题
ThreadPoolExecutor	ExecutorService的默认实现
SchedulesThreadPoolExecutor	继承ThreadPoolExecutor的ScheduledExecutorService接口实现，周期性任务调度的类实现

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在Executors类里面提供了一些静态工厂，生成一些常用的线程池。

1. newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

2.newFixedThreadPool

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

3. newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

4.newScheduledThreadPool

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

PS：但需要注意使用，newSingleThreadExecutor和newFixedThreadPool将超过处理的线程放在队列中，但工作线程较多时，会引起过多内存被占用，而后两者返回的线程池是没有线程上线的，所以在使用时需要当心，创建过多的线程容易引起服务器的宕机。

使用ThreadPoolExecutor自定义线程池，具体使用时需根据系统及JVM的配置设置适当的参数，下面是一示例：

```
int corePoolSize = Runtime.getRuntime().availableProcessors();
threadsPool = new ThreadPoolExecutor(corePoolSize, corePoolSize, 10l, TimeU
                new LinkedBlockingQueue<Runnable>(2000));
```

2.8 Future和FutureTask

Future是一个接口，FutureTask是一个具体实现类。这里先通过两个场景看看其处理方式及优点。

场景1,

现在通过调用一个方法从远程获取一些计算结果，假设有这样一个方法：

```
HashMap data = getDataFromRemote();
```

如果是最传统的同步方式的使用，我们将一直等待getDataFromRemote()的返回，然后才能继续后面的工作。这个函数是从远程获取数据的计算结果的，如果需要的时间很长，并且后面的那部分代码与这些数据没有关系的话，阻塞在这里等待结果就会比较浪费时间。如何改进呢?

能够想到的办法就是调用函数后马上返回，然后继续向下执行，等需要用数据时再来用或者再来等待这个数据。具体实现有两种方式：一个是用

Future，另一个使用回调。

Future的用法

```
Future<HashMap> future = getDataFromRemote2();
//do something
HashMap data = future.get();
```

可以看到，我们调用的方法返回一个Future对象，然后接着进行自己的处理，后面通过future.get()来获取真正的返回值。也即，在调用了getDataFromRemote2后，就已经启动了对远程计算结果的获取，同时自己的线程还在继续处理，直到需要时再获取数据。来看一下getDataFromRemote2的实现：

```
privete Future<HashMap> getDataFromRemote2(){
    return threadPool.submit(new Callable<HashMap>(){
        public HashMap call() throws Exception{
            return getDataFromRemote();
        }
    });
}
```

可以看到，在getDataFromRemote2中还是使用了getDataFromRemote来完成具体操作，并且用到了线程池：把任务加入到线程池中，把Future对象返回出去。我们调用了getDataFromRemote2的线程，然后返回来继续下面的执行，而背后是另外的线程在进行远程调用及等待的工作。get方法也可设置超时时间参数，而不是一直等下去。

场景2，

key-value的形式存储连接，若key存在则获取，若不存在这个key，则创建新连接并存储。

传统的方式会使用HashMap来存储并判断key是否存在而实现连接的管理。而这在高并发的时候会出现多次创建连接的现象。那么新的处理方式又是怎样呢？

通过ConcurrentHashMap及FutureTask实现高并发情况的正确性，

ConcurrentHashMap的分段锁存储满足数据的安全性又不影响性能，FutureTask的run方法调用Sync.innerRun方法只会执行Runnable的run方法一次(即使是高并发情况)。

2.9 并发容器

在JDK中，有一些线程不安全的容器，也有一些线程安全的容器。并发容器是线程安全容器的一种，但是并发容器强调的是容器的并发性，也就是说不仅追求线程安全，还要考虑并发性，提升在容器并发环境下的性能。

加锁互斥的方式确实能够方便地完成线程安全，不过代价是降低了并发性，或者说是串行了。而并发容器的思路是尽量不用锁，比较有代表性的是以CopyOnWrite和Concurrent开头的几个容器。CopyOnWrite容器的思路是在更改容器的时候，把容器写一份进行修改，保证正在读的线程不受影响，这种方式用在读多写少的场景中会非常好，因为实质上是在写的时候重建了一次容器。而以Concurrent开头的容器的具体实现方式则不完全相同，总体来说是尽量保证读不加锁，并且修改时不影响读，所以达到比使用读写锁更高的并发性能。比如上面所说的ConcurrentHashMap，其他的并发容器的具体实现，可直接分析JDK中的源码。

未完待续