## 构建基础模块

- 同步容器类
  - 迭代器与ConcurrentModificationException
  - 隐藏迭代器

```
public class HiddenIterator {
   @GuardedBy("this") private final Set<Integer> set = new HashSet<Integer>();

   public synchronized void add(Integer i) {
      set.add(i);
   }

   public synchronized void remove(Integer i) {
      set.remove(i);
   }

   public void addTenThings() {
      Random r = new Random();
      for (int i = 0; i < 10; i++)
         add(r.nextInt());
      System.out.println("DEBUG: added ten elements to " + set);
   }
}
```

- 并发容器
  - ConcurrentHashMap
    - 分段锁
    - 弱一致性
    - size和isEmpty是近似值,不是精确值
  - CopyOnWriteArrayList/CopyOnWriteArraySet
    - 每次修改时,都会创建并重新发布一个新的容器副本,从而实现可变性
    - 返回的迭代器不会抛出ConcurrentModificationException
    - 迭代器返回的元素与创建时的元素完全一致
    - 仅当迭代操作远远多于修改时,才应该使用
- 阻塞队列和生产者--消费者模式
  - BlockingQueue
  - Deque/BlockingDeque
    - 工作密取
- 阻塞方法与中断方法
  - 线程阻塞或者暂停执行的原因

- 等待IO操作结束
- 等待获取一个锁
- 等待从Thread.sleep方法中醒来
- 等待另一个线程的执行结果
- 可能的状态
  - Blocked
  - Waiting
  - Timed_waiting

- 中断
  - 一种协作机制
  - 处理对中断的响应
    - 传递InterruptedException
    - 恢复中断

```
//调用当前线程上的interrupt方法
public class TaskRunnable implements Runnable {
    BlockingQueue<Task> queue;

    public void run() {
        try {
            processTask(queue.take());
        } catch (InterruptedException e) {
            // restore interrupted status
            Thread.currentThread().interrupt();
        }
    }

    void processTask(Task task) {
        // Handle the task
    }

    interface Task {
    }
}
```

- 同步工具类
  - 闭锁
    - CountDownLatch
  - FutrueTask
  - 信号量

    - Semaphore
  - 栅栏
    - CyclicBarrier
    - Exchanger
- 构建高效且可伸缩的结果缓存
- 小结
  - 可变状态是至关重要的

    所有的并发问题都可以归结为如何协调对并发状态的访问.可变状态越少,就越容易确保线程的安全性
  - 尽量将域声明为final类型的,除非需要它们是可变的
  - 不可变对象一定是线程安全的
  - 封装有助于管理复杂度
    - 更易于维持不变性条件
    - 将同步策略封装在对象中,更容易遵循同步策略
  - 用锁来保护每个可变变量
  - 当保护同一个不变性条件中的所有变量时,要使用同一个锁
  - 在执行复合操作期间,要持有锁
  - 如果从多个线程中访问同一个可变变量时没有同步,那么程序会出现问题
  - 不要故作聪明的推断出不需要使用同步
  - 在设计过程中考虑线程安全,或者在文档中明确的指出它不是线程安全的
  - 将同步策略文档化

New 1 New 2