- 多进程和多线程本质区别：
  每个进程拥有自己的一整套变量，而线程则共享数据
- 启动线程执行一个任务简单过程：
  1. 将任务代码移到实现了 Runnable 接口的类的 run 方法中。

```
public interface Runnable{
   void run();
}
```

  Runnable 是函数式接口，可以用 lambda 表达式创建一个实例：

```
Runnable r = ()->{task code};
```

  2. 由 Runnable 创建一个 Thread 对象

```
Thread t = new Thread(r) ;
```

  3. 启动线程

```
t.start()
```

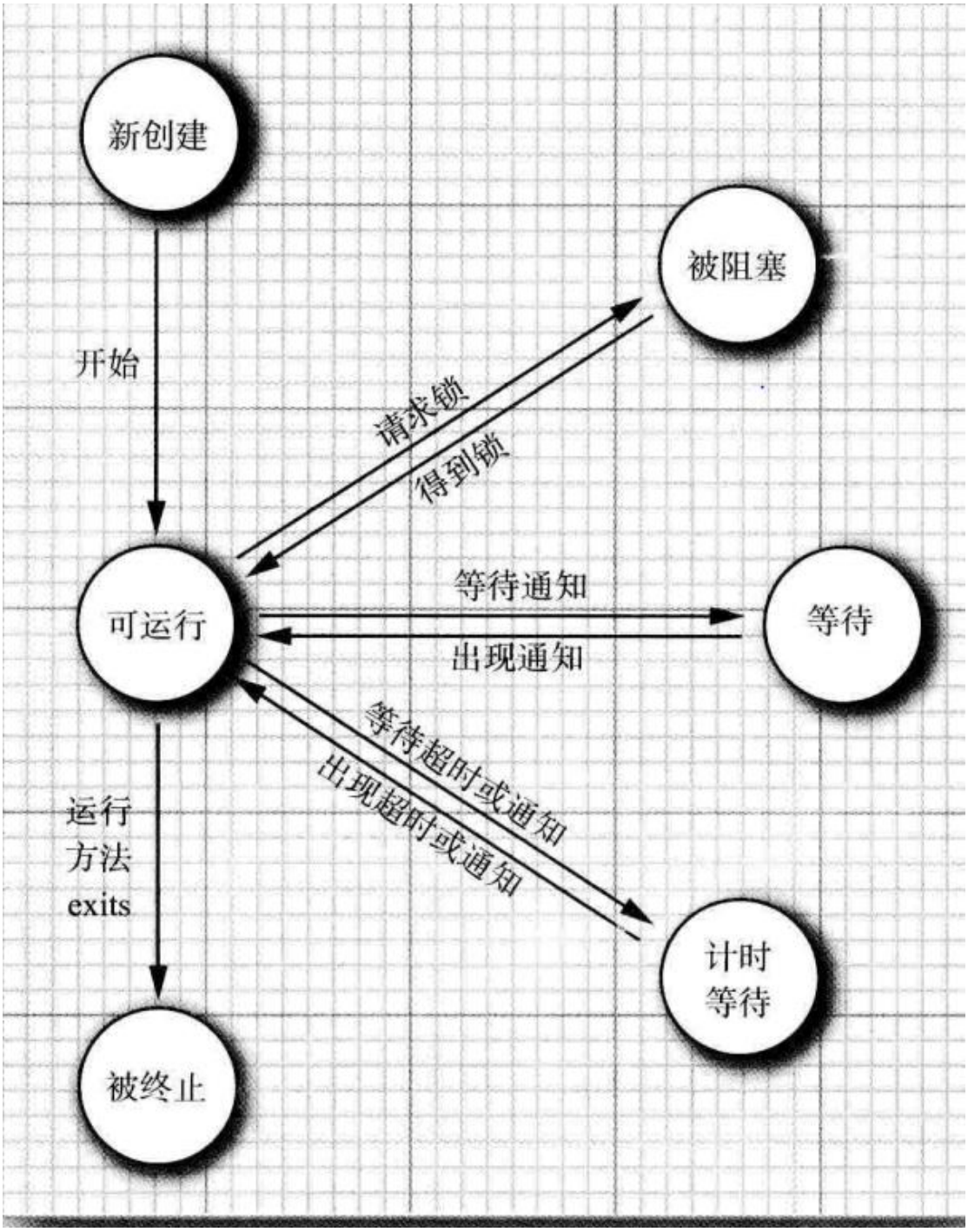
- 也可以构建一个 Thread 类的子类，实现 run()方法来定义一个线程，但这样运行任务和并行机制耦合，不推荐
- 不要调用 Thread 类或者 Runnable 对象的 run()方法，这样只会在当前线程执行任务。Thread.start()方法才会新起一个线程
- 线程中断：
  1. 没有强制线程终止的方法（早期的 stop()方法被弃用），iterrupt()方法用来请求终止线程
  2. 线程调用 interrupt()方法时，线程的**中断标志位（boolean 标识）**被置位
  3. 每个线程都应该不时地检查中断标识位，以判断线程是否被中断

```
while(!Thread.currentThread().isInterrupted()){
   do more work
}
```

  4. 但是如果线程处于阻塞状态（sleep 或 wait），即使此时调用了 interrupt()方法，也不会去执行检测代码。所以此时设计为抛出 InterruptedException 异常。
  5. 如果在中断状态被置位时调用 sleep 方法，将会清除中断状态，也抛出 InterruptedException 异常
  6. 静态方法 interrupted()和实例方法 isInterrupted()方法区别：
     - Static boolean interrupted()：检查当前线程是否被中断，将当前线程的中断标识重置为 false
     - boolean isInterrupted()：检查当前线程是否被中断
- 线程状态：getState()获得
  1. 新创建(new):使用 new 操作符创建一个新线程时，如 new Thread(r)
  2. 可运行线程(runnable):线程调用 start 方法之后，可运行不等于运行，取决于操作系统是否给线程提供运行时间
  3. 被阻塞(blocked):当一个线程试图获取一个内部的对象锁（不是 java.util.concurrent 库中的锁 Lock,而是 synchronized 相关），而该锁被其他线程持有，则进入阻塞状态

4. 等待（waiting）：当线程等待另一个线程通知调度器一个条件时，它自己进入等待状态
5. 计时等待（time waiting）：几个带超时参数的方法，导致进程进入计时等待状态，这一状态保持到超时期满或者接收到适当的通知。

- 状态转移：



- 线程优先级：

1. 每个线程有一个优先级，默认继承它父线程的优先级
2. 可以使用 setPriority 方法提高或降低线程优先级
3. 优先级在 MIN_PRIORITY(1)与 MAX_PRIORITY(10)之间，NORM_PRIORITY 为 5
4. 优先级并不能完全保证线程调度顺序，java 优先级和宿主机的优先级映射。比如在 windows 有 7 个优先级，所以存在多个 java 优先级映射到同一个操作系统优先级；而 Oracle 为 Linux 提供的 java 虚拟机中，线程的优先级被忽略

- 竞争条件（race condition）:两个或两个以上的线程需要共享对同一数据的存取
- 锁对象：ReentrantLock(1.5)：保护临界区代码片段，

```
Public class Bank{
 Private Lock bankLock = new ReetrantLock();
 public void transfer(int from,int to ,int amount){
  bankLock.lock();
  …//临界区，一旦一个线程封锁了锁对象，其他线程调用 lock 语句时，被阻塞
  finally{
   bankLock.unlock() //必须在 finally 中释放
    }
  }
```

  - 每一个 Bank 对象有自己的 ReetrantLock 对象，线程访问不同的 Bank 对象不会阻塞
  - 锁可重入

  条件对象：当线程进入临界区，发现在满足某一个条件后它才能执行，就要使用条件对象（条件变量）来管理已经获得了一个锁但是却不能做有用工作的线程。可以拥有多个

```
Class Bank{
 private Condition sufficientFunds;
 public Bank(){sufficientFunds = bankLock.newCondition()}
 public void transfer(){
  bankLock.lock()//先获得锁对象
  while(余额<转出数量){
     sufficientFunds.await()//当前线程被阻塞，并放弃锁对象，并等待被唤醒。被唤醒后开始试图重新获得锁，并从被阻塞的地方继续执行。所有 await 通常需要在循环体中：因为有可能又不再满足条件了
     //while(not ok){sufficientFunds.await();}
  }
  …
  //end transfer
  sufficientFunds.signalAll();//完成转账，唤醒因为这个条件而等待的所有线程。不能使用 signal(),这个方法是随机唤醒一个，可能会导致死锁
 }
```

- Synchronized 关键字：
  1. 1.0 开始，java 中的每个对象都有一个内部锁.使用 synchronized 关键字可以获得该内部锁

2. 与 Lock 比较：

```
public synchronized void method(){}
等价于：
public void method(){
  this.intrinsicLock.lock();
  ...
  finally{this.intrinsicLock.lock()}
}
```

3. 内部对象锁只有一个相关条件，wait()等价于 await()，notifyAll()等价于 signalAll()
4. 也可以将静态方法声明为 synchronized,则对应类对象的内部锁，同一个类的所有静态方法将进行同步
5. 通过同步阻塞获得锁：

```
synchronized(obj){
  ...
}
```

- 原子性
  1. java.util.concurrent.atomic 包中很多类高效、原子方式实现自增自减，比如 AtomicInteger 的 increamAndGet()
  2. 复杂更新 compareAndSet:

```
public static AtomicLong larget = new AtomicLong()
...
do{
  oldValue = larget.get()
  newValue = Math.max(oldValue,observed)
}while(!larges.compareAndSet(oldValue,newValue))
```

  3. 当大量线程要访问相同的原子值，性能也会大幅下降，java8 提供了 LongAdder 和 LongAccumulator 类来解决。LongAdder 包括多个变量（加数），多个线程更新不同的加数，线程个数增加同时增加新的加数，只有当所有工作完成后才计算总和。（当然可以不是加数，也可能是减数，但这个操作必须满足结合律和交换律）
- 线程局部变量 ThreadLoacl:
  1. 在一个给定现充中首次调用 get 时，会调用 initalValue 方法，在此之后 get 方法会返回属于当前线程的那个实例
  2. Lambda 初始化方法：ThreadLoacl.withInitial(()->new SimleDateFormate("yyyy-MM-dd"))
- 锁测试与超时
  1. tryLock():**锁测试**不同于申请锁，当失败时线程阻塞方法，在成功时返回 true，失败时返回 false
  2. tryLock(long time,TimeUnit unit):超时，尝试获得锁，阻塞事件不会超过给定值，成功返回 true
- 读/写锁(ReetrantReadWriteLock):当很多线程从一个数据结构读取数据而很少线程修改其中数据时适用。步骤：
  1. 构造一个对象：

```
Private ReetrantReadWriterLock rwl = new ReetrantReadWriterLock()
```

2. 抽取读锁和写锁：

```
Lock readLock = rwl.readLock()
Lock writerLock = rwl.writeLock()
```

3. 对所有获取方法加读锁：

```
public double getTotalBalance(){
  readLock.lock();//读锁，可以被多个操作共用，但排斥写操作
  try{…}
  finally{readLock.unlock();}
}
```

4. 对所有修改方法加写锁：

```
public void transfer(){
  writeLock.lock();//写锁，排斥所有写锁和读锁
  try{…}
  finally{writeLock.unlock();}
}
```

- 线程安全的集合：允许并发访问数据结构的不同部分来使竞争极小化。size 方法需要遍历。返回弱一致性的迭代器，不会抛出 ConcurrentModificationException
  - 常见线程安全集合：
    - ConcurrentLinkedQueue<E>()//可以被多线程安全访问的无边界非阻塞队列
    - ConcurrentSkipListSet<E>()//
    - ConcurrentSkipListSet<E>(Comparator<?super E comp)//多线程安全访问的有序集。第一个构造器要求元素实现 comparable
    - ConcurrentHashMap<K,V>()
    - ConcurrentHashMap<K,V>(ini initialCapacity)
    - ConcurrentHashMap<K,V>(int initialCapacity,//初试容量，默认 16
      float loadFactor,//负载因子，控制调整
      int concurrencyLevel)://估计的并发写着数量
    - ConcurrentSkipListMap<K,V>()
    - ConcurrentSkipListMap<K,V>(Comparator<? Super K>comp)//有序映射表，第一个元素要求实现 comparable 接口
  - 映射条目的院子更新：
    - replace

      ```
      do{
        oldValue = map.get(word);
        newValue = oldValue==null?1:oldValue+1;
      }While(!map.replace(word,oldValue,newValue))
      ```

    - LongAdder 或 AtomicLong

      ```
      map.putIfAbsent(word,new LongAdder());
      map.get(word).increment();
      //可以省略为：map.putIfAbsent(word,new LongAdder()).increment()
      ```

    - compute,提供函数

      ```
      map.compute(word,(k,v)-> v==null?1:1+v);
      ```

    - 还有 computeIfAbsent 和 computeIfPresent 分别在无原值和有原值时执行函数

- merge：在没有原值时适用提供的默认值，有原值时执行函数

```
map.merge(word,1,(oldvalue,newvalue)->old+new);
```

- 对并发散列映射执行批操作：（不会冻结映射的快照，所以结果是个近似）。需要一个参数化阈值（parallelism threshold）,如果映射包含的元素多于这个阈值，则并发执行
  - 搜索（search）:为每个键或值提供一个函数，直到函数生成一个非 null 结果
  - 归约（reduce）：组合所有键或值，需要提供一个累加函数
  - forEach，为所有键或值提供一个函数

  每个操作有四个版本：
  - operationKeys
  - operationValues
  - operation：处理键和值
  - operationEntries:处理 Map.Entry

- Callable 与 Future：
  - Runnable 相当于一个没有参数，没有返回值的异步方法。而 Callable 同 Runnable 相似，但有返回值：

```
public interface Callable<V>{
  V call() throws Exception;
}
```

  - Future 保存异步计算结果，可以启动一个计算，交给一个线程，然后等计算结束之后获得结果：

```
public interface Future<V>{
  V get() throws …;// 阻塞调用，直到计算完成。线程被中断则抛出 InterruptedException
  V get(long timeout,TimeUnit unit) throws…;//调用超时，抛出 TimeoutException;线程被中断，则同上
  void cancel(boolean mayInterrupt);//如果还没开始则取消，如果开始了则若参数为 true 则中断
  boolean isCancelled();
  boolean isDone();//判断计算完成与否
}
```

  - FutureTask 包装器，实现了 Future 和 Runnable 两者接口。可将 Callable 转换为两者。例如：

```
Callable<Integer> myCom = …
FutureTask<Integer>task = new FutureTask<Integer>(myCom);
Thread t = new Thread(task);//it's a runnable
t.start();
…
Integer result = task.get();//it's a Future
```

- 线程池：
  - 构建新的线程有一定代价，程序创建大量生命期短的线程，应该使用线程池（thread pool），此外使用线程池，从而固定并发线程的总数，可以减少并发线的数目。
  - 执行器（Executor）类：

- 有许多静态工厂方法用来构建线程池.返回一个实现了 ExecutorService 接口的 ThreadPoolExecutor 类对象：
  1. NewCachedThreadPool:必要时创建新线程，空闲线程保留 60 秒
  2. newFixedThreadPool:包含固定数量线程，一直存在
  3. newSingleThreadExecutor:只有一个线程的池
- 使用下面的方法，将一个 Runnable 对象或 Callable 对象提交给 ExecutorService：
  1. Future<?> submit(Runnable task)// 返回的 future 可以使用 idDone,cancel,isCancelled,但是 get 方法在完成时只返回 null
  2. Future<T> submit(Runnable task,T result)://get 返回指定类型的 result
  3. Future<T>submit(Callable<T> task);/
- 使用下面方法关闭线程池：
  1. shutdown();//启动关闭序列，逐渐关闭执行器，不再接受新任务，所有任务结束时线程池中所有线程死亡
  2. shutdownNew()://取消尚未开始的任务和中断正在运行的线程