

千锋Java学院出品

多线程

Java Platform Standard Edition

课程目标

CONTENTS

ITEMS **1** 什么是线程

ITEMS **2** 线程的组成

ITEMS **3** 线程的状态

ITEMS **4** 线程安全

ITEMS **5** 线程池

ITEMS **6** 线程安全的集合

基础多线程

Java Platform Standard Edition

什么是进程

应用程序



| 任务管理器 | | | | | |
|-------------------------------|---------|----------|--------|--------|--|
| 文件(F) 选项(O) 查看(V) | | | | | |
| 进程 性能 应用历史记录 启动 用户 详细信息 服务 | | | | | |
| 名称 | 43% CPU | 50% 内存 | 0% 磁盘 | 0% 网络 | |
| 应用 (8) | | | | | |
| > Google Chrome (32 位) | 0% | 17.0 MB | 0 MB/秒 | 0 Mbps | |
| > iTunes | 0% | 189.3 MB | 0 MB/秒 | 0 Mbps | |
| > Java(TM) Platform SE binary | 0.4% | 318.6 MB | 0 MB/秒 | 0 Mbps | |
| > Windows 资源管理器 | 1.5% | 31.5 MB | 0 MB/秒 | 0 Mbps | |
| > WPS Presentation (32 位) | 6.3% | 163.0 MB | 0 MB/秒 | 0 Mbps | |
| > WPS Writer (32 位) | 0% | 28.4 MB | 0 MB/秒 | 0 Mbps | |
| > 任务管理器 | 0.5% | 14.8 MB | 0 MB/秒 | 0 Mbps | |
| > 迅雷 (32 位) | 1.8% | 40.1 MB | 0 MB/秒 | 0 Mbps | |

程序是静止的，只有真正运行时的程序，才被称为进程。

单核CPU在任何时间点上，只能运行一个进程；宏观并行、微观串行。

QQ 

迅雷 

音乐 

什么是线程

线程，又称轻量级进程（Light Weight Process）。
程序中的一个顺序控制流程，同时也是CPU的基本调度单位。
进程由多个线程组成，彼此间完成不同的工作，交替执行，
称为多线程。



迅雷是一个进程，当中的多个下载任务即是多个线程。



Java虚拟机是一个进程，当中默认包含主线程（Main），
可通过代码创建多个独立线程，与Main并发执行。

进程和线程的区别

- 进程是操作系统资源分配的基本单位，而线程是CPU的基本调度单位。
- 一个程序运行后至少有一个进程。
- 一个进程可以包含多个线程，但是至少需要有一个线程。
- 进程间不能共享数据段地址，但同进程的线程之间可以。

- 任何一个线程都具有基本的组成部分：
 - CPU时间片：操作系统（OS）会为每个线程分配执行时间。
 - 运行数据：
 - 堆空间：存储线程需使用的对象，多个线程可以共享堆中的对象。
 - 栈空间：存储线程需使用的局部变量，每个线程都拥有独立的栈。
 - 线程的逻辑代码。

创建线程 (1)

- 创建线程的第一种方式:

4. 调用start()方法

```
public class TestCreateThread {  
    public static void main(String[] args) {  
        MyThread t1 = new MyThread();  
        t1.start();  
    }  
}
```

3. 创建子类对象

2. 覆盖run()方法

```
class MyThread extends Thread{  
    public void run(){  
        for (int i = 1; i <= 50; i++) {  
            System.out.println("MyThread: " + i);  
        }  
    }  
}
```

1. 继承Thread类

创建线程 (2)

• 创建线程的第二种方式:

```
public class TestCreateThread {  
    public static void main(String[] args) {  
        MyRunnable mr = new MyRunnable();  
        Thread t2 = new Thread(mr);  
        t2.start();  
    }  
}  
  
class MyRunnable implements Runnable {  
    public void run() {  
        for (int i = 1; i <= 50; i++) {  
            System.out.println("MyRunnable: " + i);  
        }  
    }  
}
```

4. 创建线程对象

3. 创建实现类对象

5. 调用start()方法

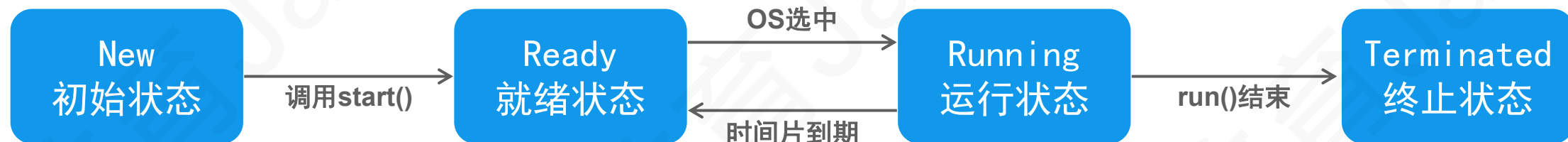
1. 实现Runnable接口

2. 覆盖run()方法

线程的状态（基本）

线程对象被创建，即为初始状态。
只在堆中开辟内存，与常规对象无异。

获得时间片之后，进入运行状态，
如果时间片到期，则回到就绪状态。

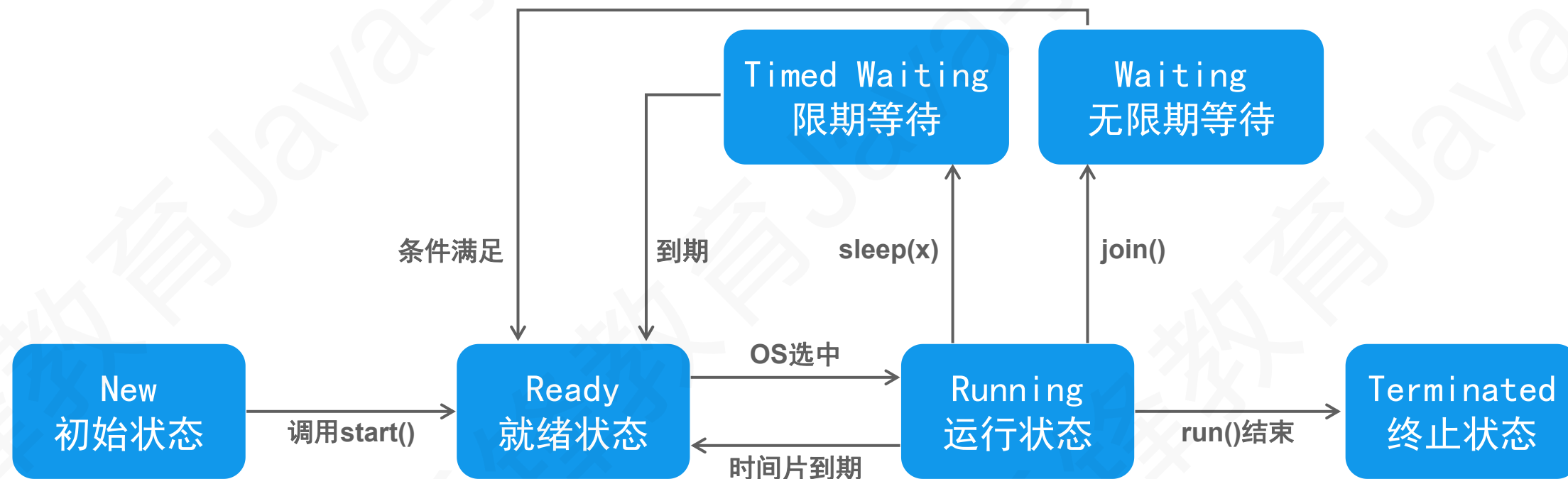


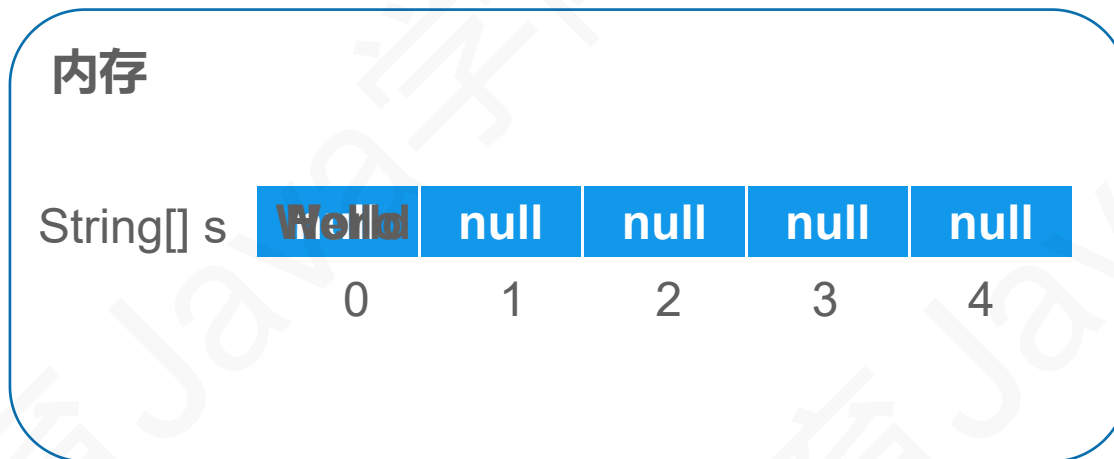
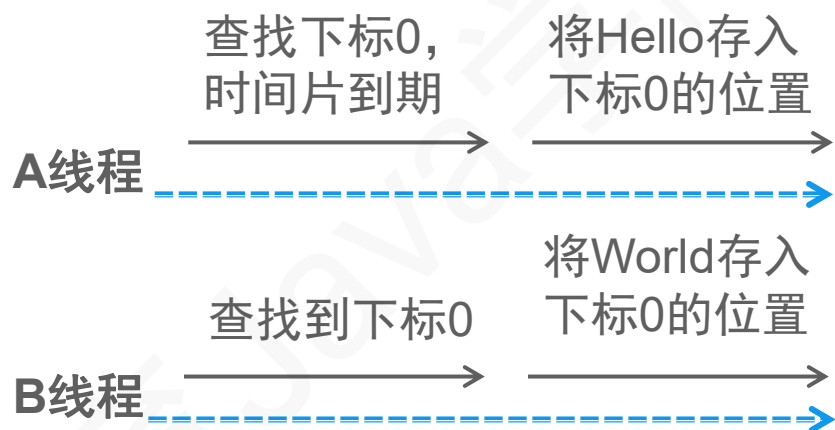
调用`start()`之后，进入就绪状态。
等待OS选中，并分配时间片。

主线程`main()`或独立线程`run()`结束，
进入终止状态，并释放持有的时间片。

- 休眠：
 - `public static void sleep(long millis)`
 - 当前线程主动休眠 millis 毫秒。
- 放弃：
 - `public static void yield()`
 - 当前线程主动放弃时间片，回到就绪状态，竞争下一次时间片。
- 结合：
 - `public final void join()`
 - 允许其他线程加入到当前线程中。

线程的状态 (等待)





- 需求：A线程将“Hello”存入数组的第一个空位；B线程将“World”存入数组的第一个空位。
- 线程不安全：
 - 当多线程并发访问临界资源时，如果破坏原子操作，可能会造成数据不一致。
 - 临界资源：共享资源（同一对象），一次仅允许一个线程使用，才可保证其正确性。
 - 原子操作：不可分割的多步操作，被视作一个整体，其顺序和步骤不可打乱或缺省。

- 同步代码块:

```
synchronized(临界资源对象){ //对临界资源对象加锁  
    //代码 (原子操作)  
}
```

注:

每个对象都有一个互斥锁标记, 用来分配给线程的。

只有拥有对象互斥锁标记的线程, 才能进入对该对象加锁的同步代码块。

线程退出同步代码块时, 会释放相应的互斥锁标记。

线程的状态（阻塞）



注：JDK5之后就绪、运行统称Runnable

- 同步方法:

```
synchronized 返回值类型 方法名称(形参列表){ //对当前对象 (this) 加锁  
    // 代码 (原子操作)  
}
```

注:

只有拥有对象互斥锁标记的线程，才能进入该对象加锁的同步方法中。

线程退出同步方法时，会释放相应的互斥锁标记。

- 注意：
 - 只有在调用包含同步代码块的方法，或者同步方法时，才需要对象的锁标记。
 - 如调用不包含同步代码块的方法，或普通方法时，则不需要锁标记，可直接调用。
- 已知JDK中线程安全的类：
 - StringBuffer
 - Vector
 - Hashtable
 - 以上类中的公开方法，均为synchronized修饰的同步方法。

- **死锁：**

- 当第一个线程拥有A对象锁标记，并等待B对象锁标记，同时第二个线程拥有B对象锁标记，并等待A对象锁标记时，产生死锁。
- 一个线程可以同时拥有多个对象的锁标记，当线程阻塞时，不会释放已经拥有的锁标记，由此可能造成死锁。

- **生产者、消费者：**

- 若干个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个能存储多个产品的缓冲区，生产者将生产的产品放入缓冲区中，消费者从缓冲区中取走产品进行消费，显然生产者和消费者之间必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个满的缓冲区中放入产品。

- **等待：**
 - `public final void wait()`
 - `public final void wait(long timeout)`
 - 必须在对obj加锁的同步代码块中。在一个线程中，调用obj.wait() 时，此线程会释放其拥有的所有锁标记。同时此线程阻塞在o的等待队列中。释放锁，进入等待队列。
- **通知：**
 - `public final void notify()`
 - `public final void notifyAll()`
 - 必须在对obj加锁的同步代码块中。从obj的Waiting中释放一个或全部线程。对自身没有任何影响。

- 线程的创建:

- 方式1: 继承Thread类
- 方式2: 实现Runnable接口 (一个任务Task) , 传入给Thread对象并执行。

- 线程安全:

- 同步代码块: 为方法中的局部代码 (原子操作) 加锁。
- 同步方法: 为方法中的所有代码 (原子操作) 加锁。

- 线程间的通信:

- wait() / wait(long timeout): 等待
- notify() / notifyAll(): 通知

高级多线程

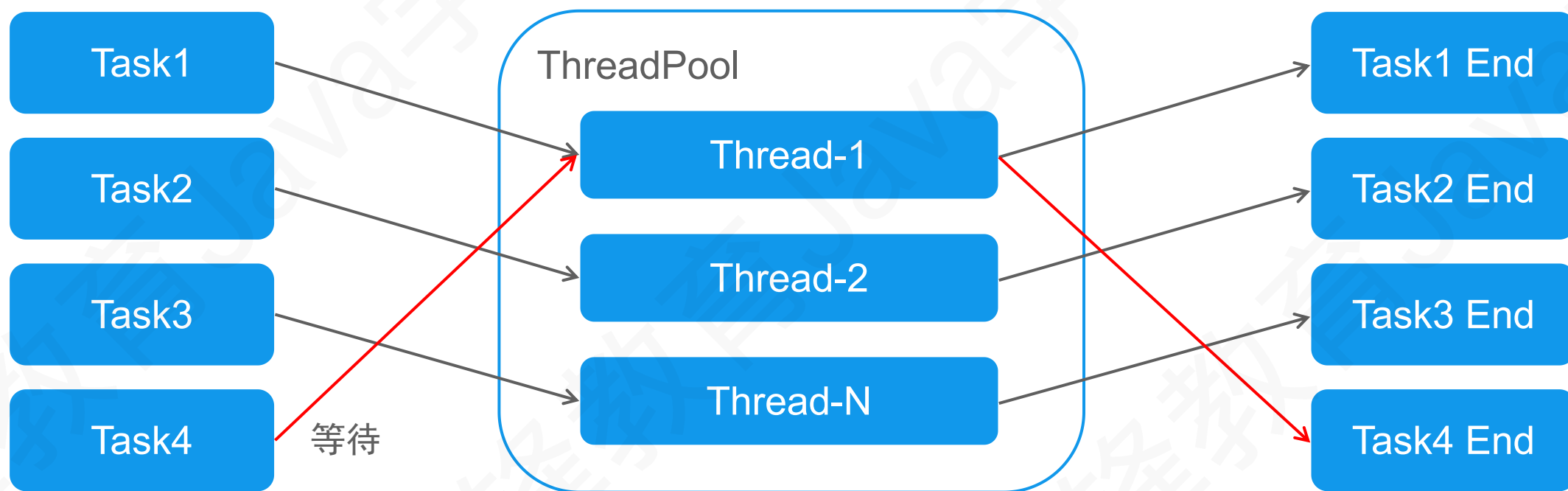
Java Platform Standard Edition

- 现有问题：

- 线程是宝贵的内存资源、单个线程约占1MB空间，过多分配易造成内存溢出。
- 频繁的创建及销毁线程会增加虚拟机回收频率、资源开销，造成程序性能下降。

- 线程池：

- 线程容器，可设定线程分配的数量上限。
- 将预先创建的线程对象存入池中，并重用线程池中的线程对象。
- 避免频繁的创建和销毁。



- 将任务提交给线程池，由线程池分配线程、运行任务，并在当前任务结束后复用线程。

- 常用的线程池接口和类(所在包java.util.concurrent):
 - Executor: 线程池的顶级接口。
 - ExecutorService: 线程池接口, 可通过submit(Runnable task) 提交任务代码。
 - Executors工厂类: 通过此类可以获得一个线程池。
- 通过 newFixedThreadPool(int nThreads) 获取固定数量的线程池。参数: 指定线程池中线程的数量。
- 通过newCachedThreadPool() 获得动态数量的线程池, 如不够则创建新的, 无上限。

Callable接口

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

- JDK5加入，与Runnable接口类似，实现之后代表一个线程任务。
- Callable具有泛型返回值、可以声明异常。

- 需求：使用两个线程，并发计算1~50、51~100的和，再进行汇总统计。
- 思考：实际应用中，如何接收call方法的返回值？

- **概念：**异步接收`ExecutorService.submit()`所返回的状态结果，当中包含了`call()`的返回值。
- **方法：**`V get()`以阻塞形式等待Future中的异步处理结果（`call()`的返回值）。
- **思考：**什么是异步？什么是同步？

- JDK5加入，与synchronized比较，显示定义，结构更灵活。

- 提供更多实用性方法，功能更强大、性能更优越。

- 常用方法:

`void lock()` //获取锁，如锁被占用，则等待。

`boolean tryLock()` //尝试获取锁（成功返回true。失败返回false，不阻塞）。

`void unlock()` //释放锁。

- ReentrantLock: Lock接口的实现类，与synchronized一样具有互斥锁功能。

```
class MyList {  
    private Lock locker = new ReentrantLock();  
    private String[] strs = { "A", "B", "", "", "" };  
    private int count = 2; // 元素个数  
    // 添加元素  
    public void add(String value) {  
        locker.lock();  
        try {  
            strs[count] = value;  
            try {  
                Thread.sleep(1000); // 主动休眠1秒钟  
            } catch (InterruptedException e) {}  
            count++;  
        } finally {  
            locker.unlock();  
        }  
    }  
}
```

显示开启锁

显示释放锁

创建重入锁对象

考虑可能出现异常，释放锁必须放入finally代码块中，避免无法释放。

- **ReentrantReadWriteLock:**
 - 一种支持一写多读的同步锁，读写分离，可分别分配读锁、写锁。
 - 支持多次分配读锁，使多个读操作可以并发执行。
- **互斥规则:**
 - 写-写：互斥，阻塞。
 - 读-写：互斥，读阻塞写、写阻塞读。
 - 读-读：不互斥、不阻塞。
 - 在读操作远远高于写操作的环境中，可在保障线程安全的情况下，提高运行效率。

ReentrantReadWriteLock

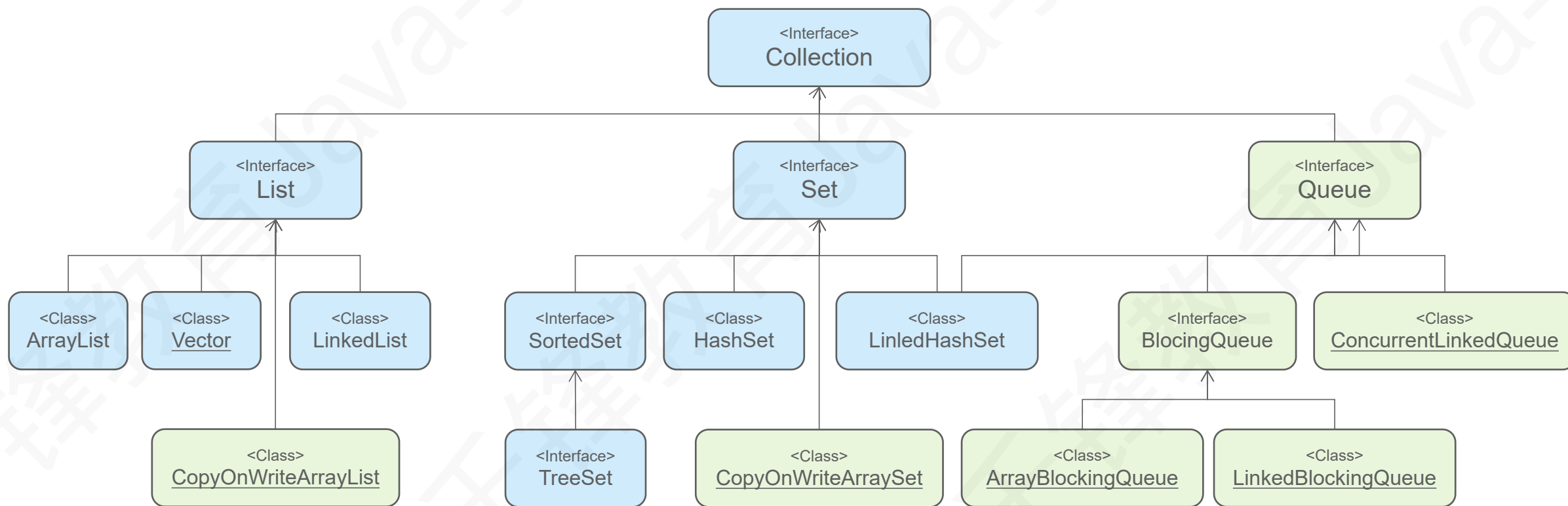
```
class MyClass{
    ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    ReadLock readLock = rwl.readLock(); //获得读锁
    WriteLock writeLock = rwl.writeLock(); //获得写锁
    private int value;
    //读方法
    public int getValue() throws Exception{
        readLock.lock(); //开启读锁
        try {
            Thread.sleep(1000); //休眠1秒
            return value;
        }finally{
            readLock.unlock(); //释放读锁
        }
    }
    //写方法
    public void setValue(int value) throws Exception{
        writeLock.lock(); //开启写锁
        try {
            Thread.sleep(1000); //休眠1秒
            this.value = value;
        } finally{
            writeLock.unlock(); //释放写锁
        }
    }
}
```

```
public class TestReadWriteLock {
    public static void main(String[] args){
        final MyClass mc = new MyClass();
        Runnable task1 = new Runnable(){
            public void run(){
                try { mc.setValue(1); }catch(Exception e){}
            }
        };
        Runnable task2 = new Runnable(){
            public void run(){
                try { mc.getValue(); }catch(Exception e){}
            }
        };
        ExecutorService es = Executors.newFixedThreadPool(20);
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 2; i++) {
            es.submit(task1); //提交2次写任务
        }
        for (int i = 0; i < 18; i++) {
            es.submit(task2); //提交18次读任务
        }
        es.shutdown(); //关闭线程池
        while(!es.isTerminated()){} //如果线程未全部结束，则空转等待
        System.out.println(System.currentTimeMillis() - startTime);
    }
}
```

运行结果：互斥锁运行时间20034毫秒、读写锁运行时间3004毫秒。（2次写各占1秒、18次读共占1秒）

线程安全的集合

- Collection体系集合、以及线程安全集合。



注：绿色代表新增知识，下划线代表线程安全集合

- Collections工具类中提供了多个可以获得线程安全集合的方法。
 - `public static <T> Collection<T> synchronizedCollection(Collection<T> c)`
 - `public static <T> List<T> synchronizedList(List<T> list)`
 - `public static <T> Set<T> synchronizedSet(Set<T> s)`
 - `public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)`
 - `public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)`
 - `public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)`
- JDK1.2提供，接口统一、维护性高，但性能没有提升，均以synchronized实现。

CopyOnWriteArrayList

- 线程安全的ArrayList，加强版读写分离。
- 写有锁，读无锁，读写之间不阻塞，优于读写锁。
- 写入时，先copy一个容器副本、再添加新元素，最后替换引用。
- 使用方式与ArrayList无异。

```
public class TestCopyOnWriteArrayList {  
    public static void main(String[] args) {  
        List<String> list = new CopyOnWriteArrayList<String>();  
    }  
}
```

CopyOnWriteArraySet

- 线程安全的Set，底层使用CopyOnWriteArrayList实现。
- 唯一不同在于，使用addIfAbsent()添加元素，会遍历数组，
- 如存在元素，则不添加（扔掉副本）。

```
public class TestCopyOnWriteArraySet {  
    public static void main(String[] args) {  
        Set<String> set = new CopyOnWriteArraySet<String>();  
    }  
}
```

ConcurrentHashMap

- 初始容量默认为16段 (Segment) , 使用分段锁设计。
- 不对整个Map加锁, 而是为每个Segment加锁。
- 当多个对象存入同一个Segment时, 才需要互斥。
- 最理想状态为16个对象分别存入16个Segment, 并行数量16。
- 使用方式与HashMap无异。

```
public class TestConcurrentHashMap {  
    public static void main(String[] args) {  
        Map<String,String> map = new ConcurrentHashMap<String,String>();  
    }  
}
```

Queue接口 (队列)

- Collection的子接口，表示队列FIFO (First In First Out)
- 常用方法：
 - 抛出异常：
 - `boolean add(E e)` //顺序添加一个元素 (到达上限后，再添加则会抛出异常)
 - `E remove()` //获得第一个元素并移除 (如果队列没有元素时，则抛异常)
 - `E element()` //获得第一个元素但不移除 (如果队列没有元素时，则抛异常)
 - 返回特殊值：推荐使用
 - `boolean offer(E e)` //顺序添加一个元素 (到达上限后，再添加则会返回false)
 - `E poll()` //获得第一个元素并移除 (如果队列没有元素时，则返回null)
 - `E keep()` //获得第一个元素但不移除 (如果队列没有元素时，则返回null)

ConcurrentLinkedQueue

- 线程安全、可高效读写的队列，高并发下性能最好的队列。
- 无锁、CAS比较交换算法，修改的方法包含三个核心参数 (V,E,N)
- V：要更新的变量、E：预期值、N：新值。
- 只有当V==E时，V=N；否则表示已被更新过，则取消当前操作。

```
public class TestConcurrentLinkedQueue {  
    public static void main(String[] args) {  
        Queue<String> queue = new ConcurrentLinkedQueue<String>();  
        queue.offer("Hello");//插入  
        queue.offer("World");//插入  
        queue.poll();//删除Hello  
        queue.peek();//获得World  
    }  
}
```

BlockingQueue接口（阻塞队列）



- Queue的子接口，阻塞的队列，增加了两个线程状态为无限期等待的方法。
- 方法：
 - void put(E e) //将指定元素插入此队列中，如果没有可用空间，则等待。
 - E take() //获取并移除此队列头部元素，如果没有可用元素，则等待。
- 可用于解决生产、消费者问题。

- **ArrayBlockingQueue:**

数组结构实现，有界队列。（手工固定上限）

```
public class TestArrayBlockingQueue {  
    public static void main(String[] args) {  
        BlockingQueue<String> abq = new ArrayBlockingQueue<String>(10);  
    }  
}
```

- **LinkedBlockingQueue:**

- 链表结构实现，无界队列。（默认上限Integer.MAX_VALUE）

```
public class TestLinkedBlockingQueue {  
    public static void main(String[] args) {  
        BlockingQueue<String> lbq = new LinkedBlockingQueue<String>();  
    }  
}
```


- **ExecutorService**线程池接口、**Executors**工厂。
- **Callable**线程任务、**Future**异步返回值。
- **Lock**、**ReentrantLock**重入锁、**ReentrantReadWriteLock**读写锁。
- **CopyOnWriteArrayList**线程安全的ArrayList。
- **CopyOnWriteArraySet**线程安全的Set。
- **ConcurrentHashMap**线程安全的HashMap。
- **ConcurrentLinkedQueue**线程安全的Queue。
- **ArrayBlockingQueue**线程安全的阻塞Queue。（生产者、消费者）

THANK YOU



做真实的自己，用良心做教育