08-并发编程

刘亚雄

极客时间-Java 讲师

今日目标

1. 掌握AQS实现原理
2. 了解并发编程工具类
3. 掌握线程池工作原理
4. 掌握ThreadLocal实现原理
5. 了解Future和FutureTask

五、锁与AQS

5.3 Synchronized和JUC的锁对比

Java已经提供了synchronized，为什么还要使用JUC的锁呢？

Synchronize的缺陷：

- **第一： Synchronized无法控制阻塞时长，阻塞不可中断**
  - 使用Synchronized，假如占有锁的线程被长时间阻塞（IO、sleep、join），由于线程阻塞时没法释放锁，会导致大量线程堆积，轻则影响性能，重则服务雪崩
  - **JUC的锁可以解决这两个缺陷**
- **第二：读多写少的场景中，多个读线程同时操作共享资源时不需要加锁**
  - Synchronized不论是读还是写，均需要同步操作，这种做法并不是最优解
  - **JUC的ReentrantReadWriteLock锁可以解决这个问题**

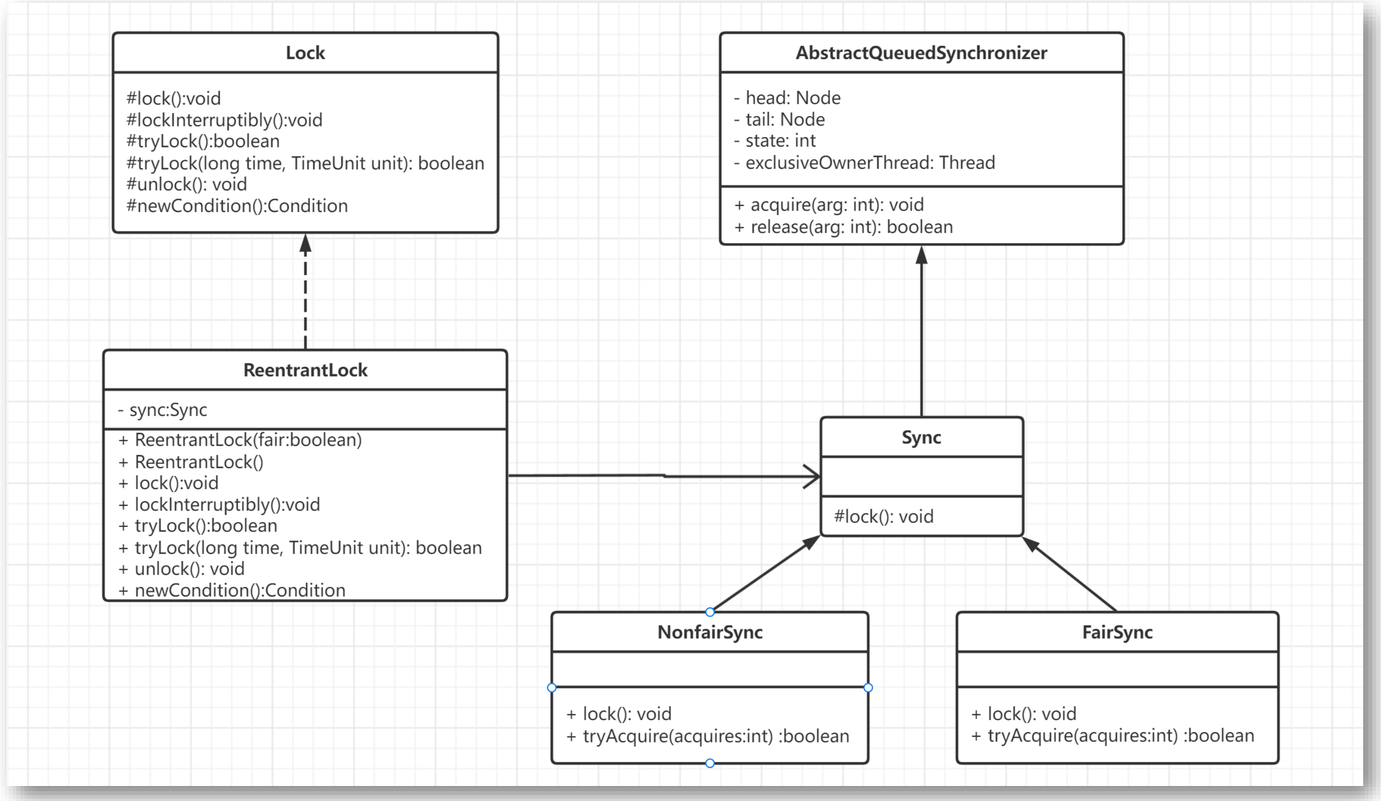
接下来我们看一下ReentrantLock实现原理

5.4 ReentrantLock原理分析之AQS

在重入锁ReentrantLock类关系图中，可以看到NonfairSync和FairSync都继承自抽象类Sync，而**Sync类继承自抽象类AbstractQueuedSynchronizer**（简称AQS）。

如果你看过JUC的源码，发现不仅重入锁用到了AQS， JUC 中绝大部分的同步工具类也都是基于AQS

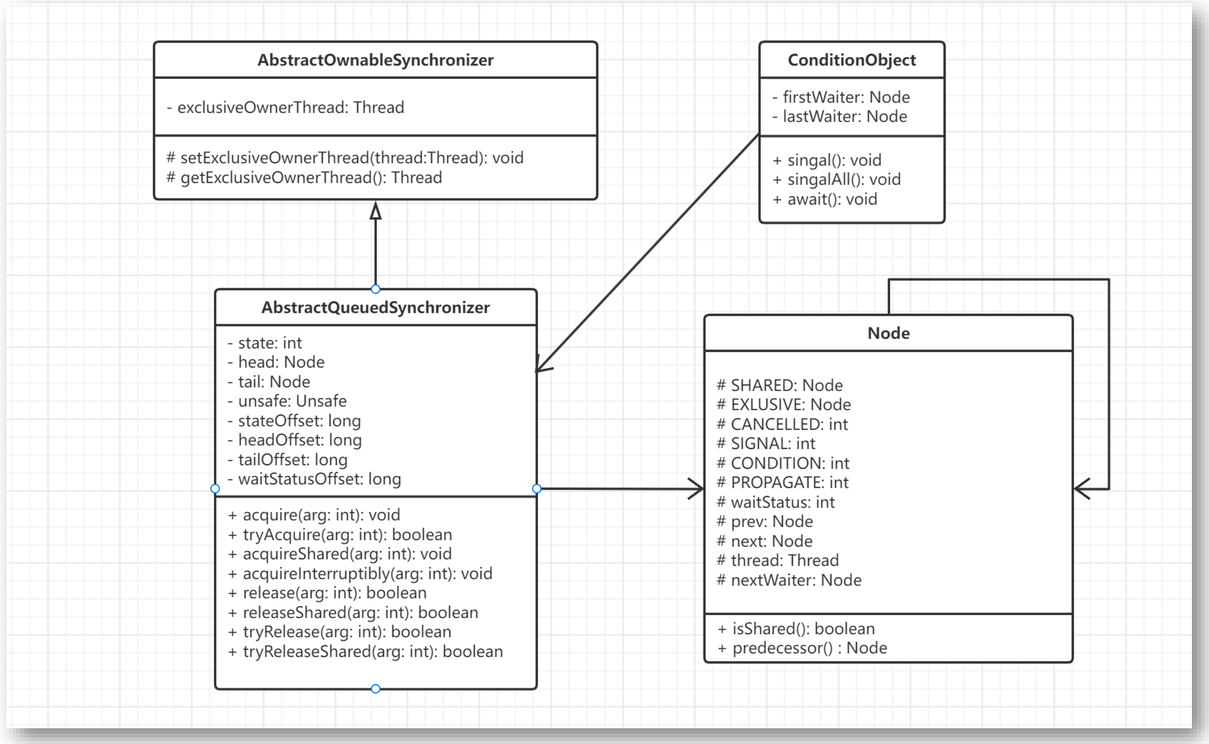
AQS是什么呢？



5.4 ReentrantLock原理分析之AQS

AQS即队列同步器，是JUC并发包中的核心基础组件，其本身只是一个抽象类。其实现原理与前面介绍的**Monitor管程**是一样的，AQS中也用到了CAS和Volatile。

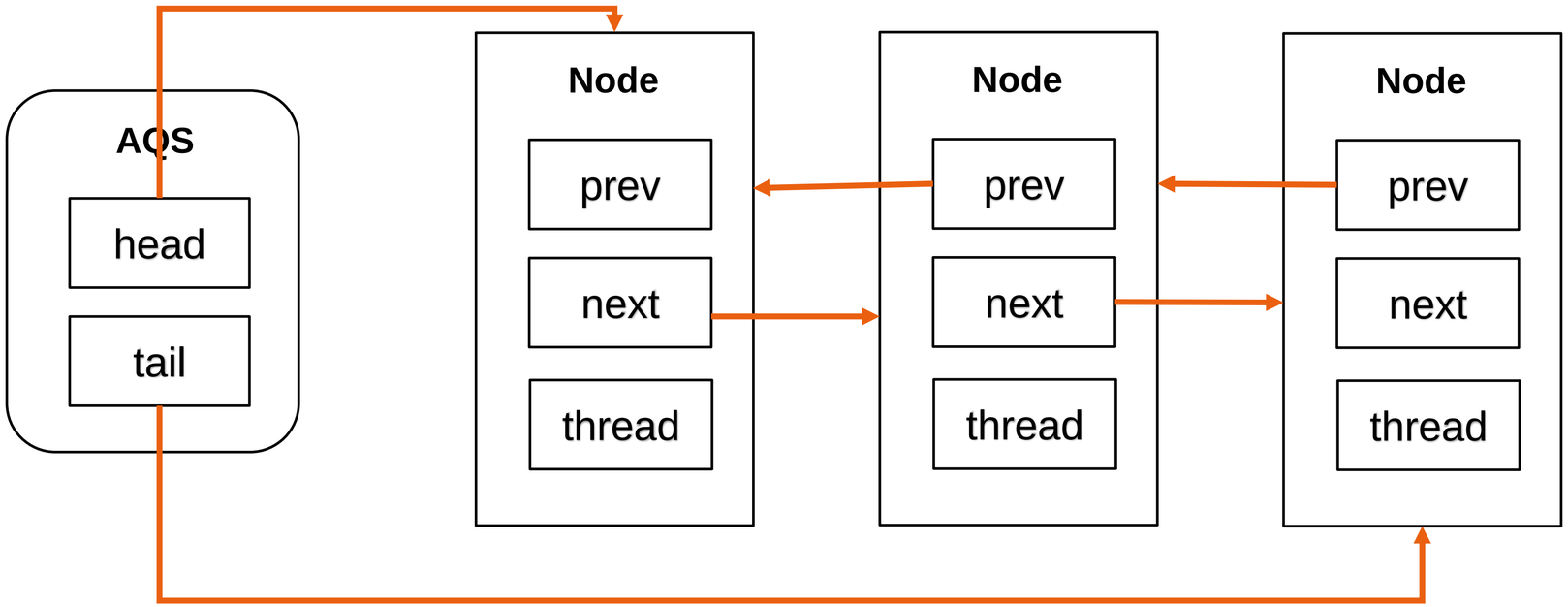
由类图可以看到，AQS是一个**FIFO的双向队列，队列中存储的是thread**，其内部通过节点head和tail记录队首和队尾元素，队列元素的类型为Node。



画个图来看一下

5.5 ReentrantLock原理分析之AQS

AQS中的内部静态类Node为链表节点，**AQS会在线程获取锁失败后，线程会被阻塞并被封装成Node加入到AQS队列中；当获取锁的线程释放锁后，会从AQS队列中的唤醒一个线程（节点）。**



5.5 ReentrantLock原理分析之AQS

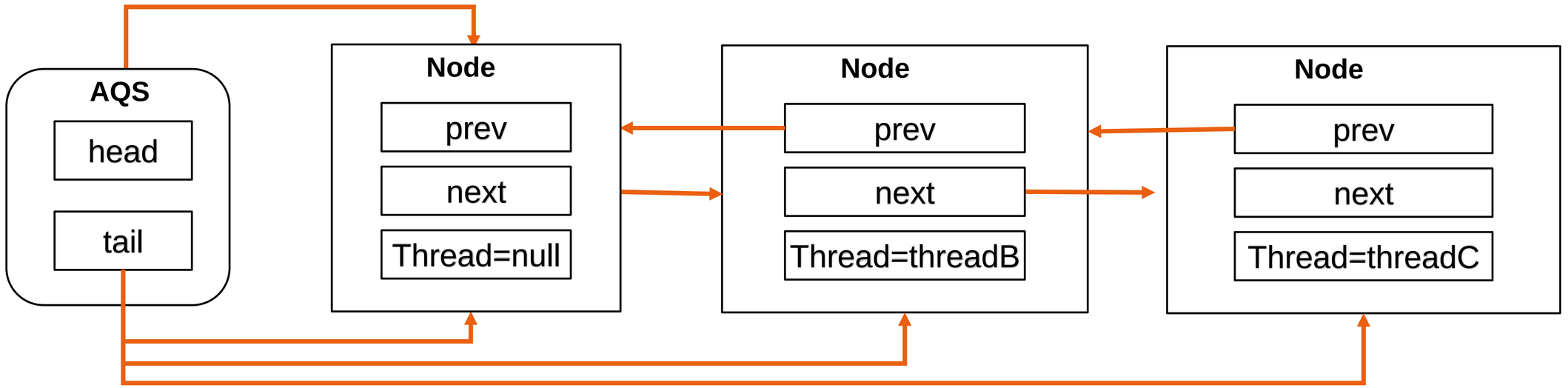
场景01-线程抢夺锁失败时，AQS队列的变化【加锁】

① AQS的head、tail分别代表同步队列头节点和尾节点指针默认为null

② 当第一个线程抢夺锁失败，同步队列会先初始化，随后线程会被封装成Node节点追加到AQS队列中。

- 假设：当前独占锁的的线程为ThreadA，抢占锁失败的线程为ThreadB。
- 2.1 同步队列初始化，首先在队列中添加Node，thread=null
- 2.2 将ThreadB封装成为Node，追加到AQS队列

③ 当下一个线程抢夺锁失败时，继续重复上面步骤。假设：ThreadC抢占线程失败



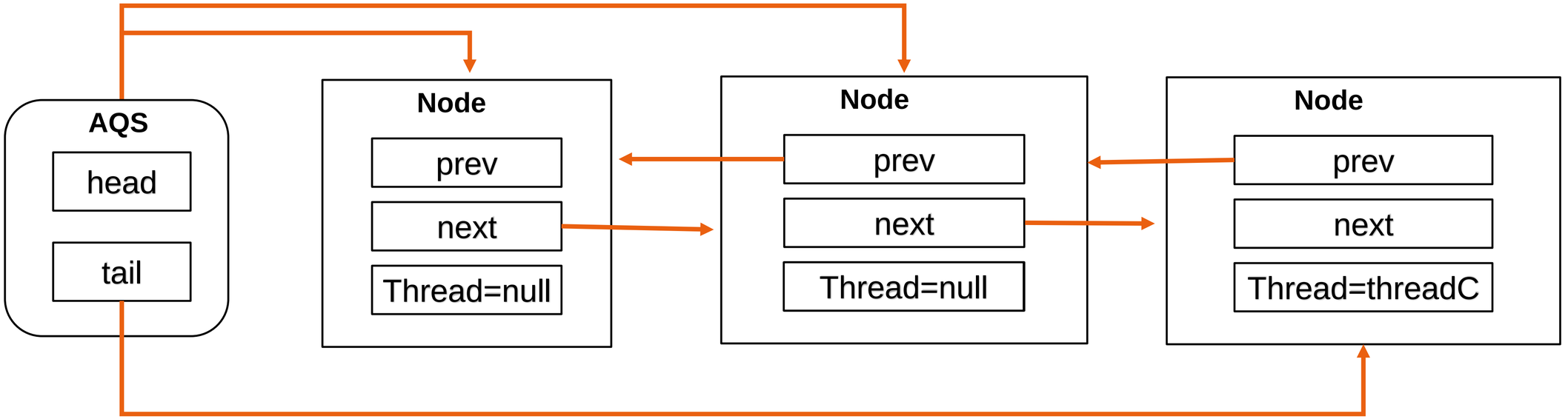
5.5 ReentrantLock原理分析之AQS

场景02-线程被唤醒时，AQS队列的变化【解锁】

① ReentrantLock唤醒阻塞线程时，会按照**FIFO的原则从AQS中head头部开始唤醒首个节点中线程**。

② head节点表示当前获取锁成功的线程ThreadA节点。

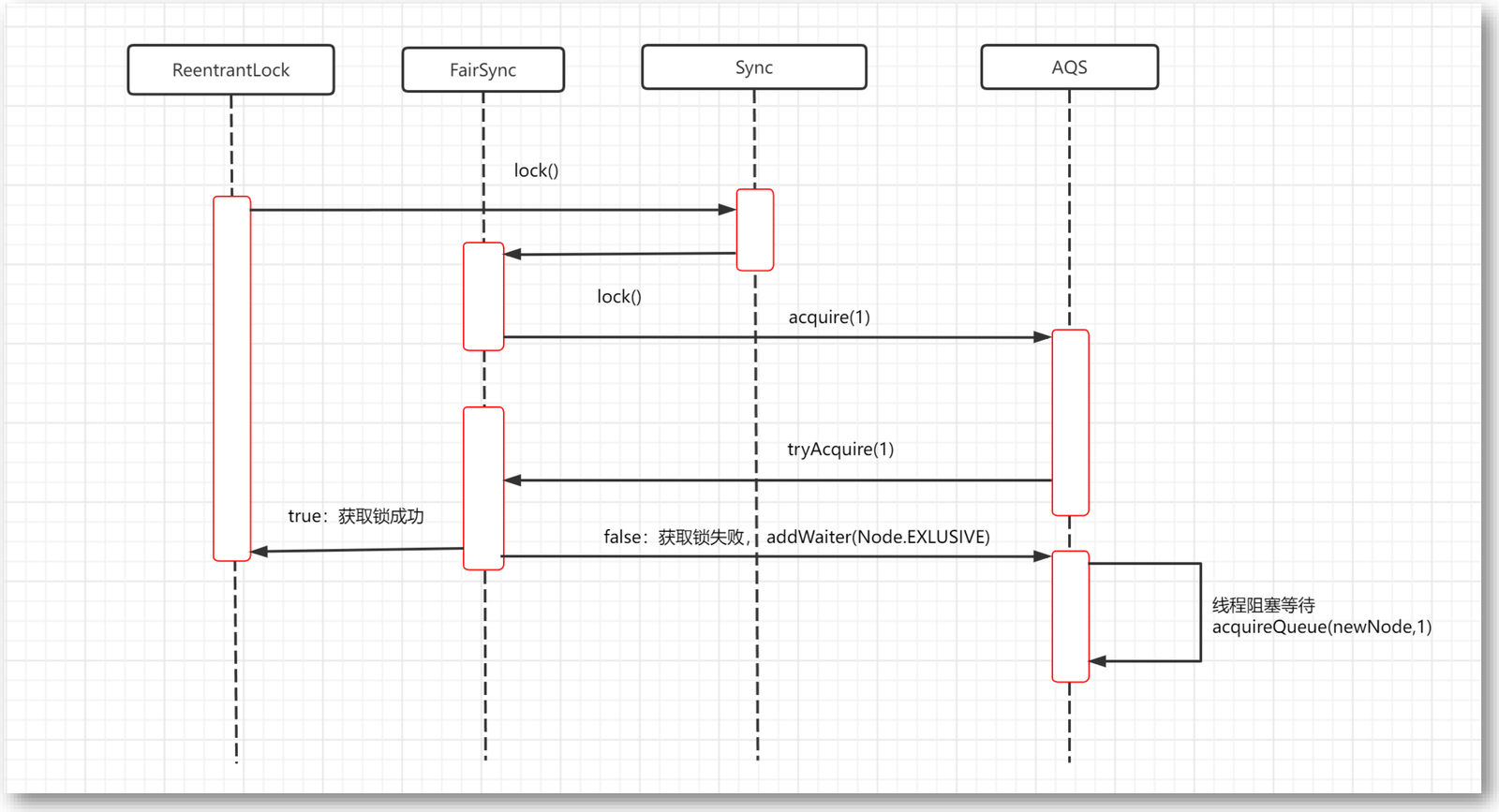
③ 当ThreadA释放锁时，它会唤醒后继节点线程ThreadB，ThreadB开始尝试获得锁，如果ThreadB获得锁成功，会将自己设置为AQS的头节点。ThreadB获取锁成功后，AQS变化如下：



5.6 ReentrantLock源码分析-锁的获取

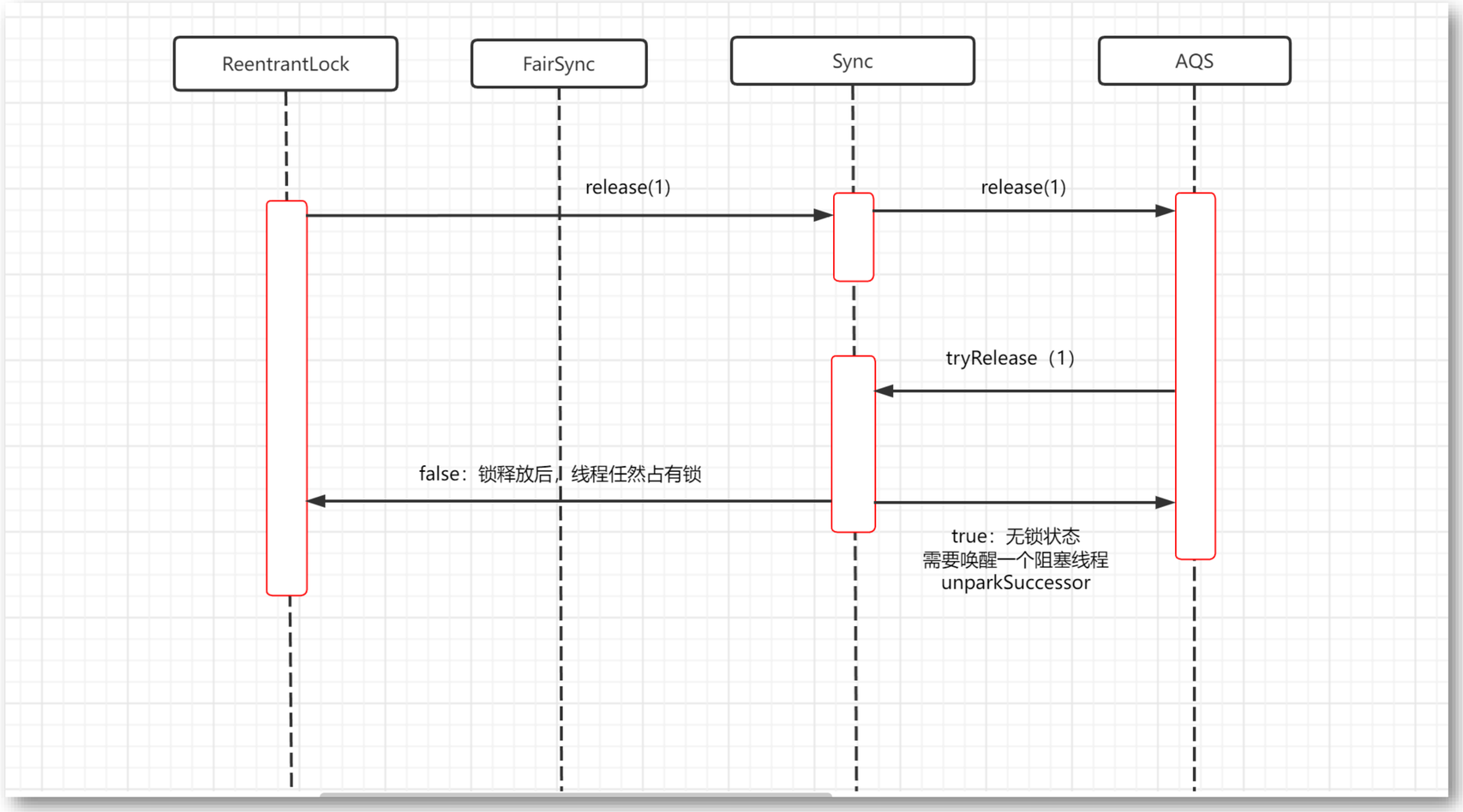
研究任何框架或工具都就要一个入口，我们以重入锁为切入点来理解AQS的作用及实现。下面我们深入ReentrantLock源码来分析AQS是如何实现线程同步的。

ReentrantLock锁获取源码分析：



5.7 ReentrantLock源码分析-锁的释放

ReentrantLock锁释放源码分析：



5.8 公平锁和非公平锁源码实现区别

公平锁/非公平锁：按照多个线程竞争同一锁时需不需要排队，能不能插队

获取锁的两处差异：

① lock方法差异：详情看课堂笔记

② tryAcquire差异：详情看课堂笔记

FairSync.lock：公平锁获取锁

```
final void lock() {
    acquire(1);
}

```

NoFairSync.lock：非公平锁获取锁，lock方法中新线程会先通过CAS操作compareAndSetState(0, 1)，尝试获得锁。

```
final void lock() {
    if (compareAndSetState(0, 1))//新线程，第一次插队
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
```

NoFairSync.tryAcquire和NoFairSync.nonfairTryAcquire：

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {//非公平锁，入队前，二次插队
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

5.9 读写锁ReentrantReadWriteLock

读写锁：维护着一对锁(读锁和写锁)，通过分离读锁和写锁，使得并发能力比一般的互斥锁有较大提升。**同一时间，可以允许多个读线程同时访问，但在写线程访问时，所有读写线程都会阻塞。**

所以说，读锁是共享的，写锁是排他的。

主要特性：

- 支持公平和非公平锁
- 支持重入
- 锁降级：写锁可以降级为读锁，但是读锁不能升级为写锁

```
1  /** 内部类  读锁 */
2  private final ReentrantReadWriteLock.ReadLock readerLock;
3  /** 内部类  写锁 */
4  private final ReentrantReadWriteLock.WriteLock writerLock;
5
6  final Sync sync;
```

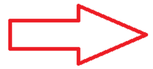
做个案例来看一下

5.10 锁优化

如何优化锁？

- 减少锁的持有时间
- 减少锁粒度
  - 将大对象拆分为小对象，增加并行度，降低锁的竞争
  - 例如：早期ConcurrentHashMap的分段锁
- 锁分离
  - 根据功能场景进行锁分离
  - 例如：读多写少的场景，使用读写锁可以提高性能
- 锁消除：锁消除是编译器自动的一种优化方式
- 锁粗化
  - 增加锁的范围，降低加解锁的频次

```
public synchronized void syncMethodBefore(){
    otherCode1();
    mutextMethod();
    otherCode2();
}
```



```
public void syncMethodAfter(){
    otherCode1();
    synchronized (this){
        mutextMethod();
    }
    otherCode2();
}
```

```
public synchronized void demoMethodBefore(){
    synchronized (lockA){
        mutextMethodA();
    }
    //做其他不需要的同步工作，但能很快执行完毕
  synchronized (lockA){
        mutextMethodB();
    }
}
```

```
public void syncMethodAfter(){
    synchronized (lockA){
        mutextMethodA();
        //做其他不需要的同步工作，但能很快执行完毕
        mutextMethodB();
    }
}
```

```
public synchronized void demoMethodBefore(){
    for (int i = 0; i < circle; i++) {
        synchronized (lock){
            //do sth
        }
    }
}
```

```
public void syncMethodAfter(){
    synchronized (lock){
        for (int i = 0; i < circle; i++) {
            //do sth
        }
    }
}
```

六、线程协作工具与并发容器

6.1 线程协作工具类

线程协作工具类，控制线程协作的工具类，帮助程序员让线程之间的协作变得更加简单

常用四个工具类：

1．CountDownLatch计数门闩：

- 倒数结束之前，一直处于等待状态，直到数到0，等待线程才继续工作。
- 场景：购物拼团、分布式锁
- 方法：
  ① new CountDownLatch(int count)
  ② await()：调用此方法的线程会阻塞，支持多个线程调用，当计数为0，则唤醒线程
  ③ countdown()：其他线程调用此方法，计数减1

案例：6个程序猿加班

6.1 线程协作工具类

常用四个工具类：

2. Semaphore信号量：

- 限制和管理数量有限的资源的使用
- 场景：Hystrix、Sentinel限流
- 方法：
  - ① new Semaphore ((int permits) 可以创建公平的非公平的策略
  - ② acquire()：获取许可证，获取许可证，要么获取成功，信号量减1，要么阻塞等待唤醒
  - ③ release()：释放许可证，信号量加1，然后唤醒等待的线程

案例：抢车位

6.1 线程协作工具类

常用四个工具类：

3. CyclicBarrier循环栅栏：

- 线程会等待，直到线程到了事先规定的数目，然后触发执行条件进行下一步动作
- 场景：并行计算
- 方法：
  - ① new CyclicBarrier(int parties, Runnable barrierAction)参数1集结线程数，参数2凑齐之后执行的任务
  - ② await()：阻塞当前线程，待凑齐线程数量之后继续执行

案例：7龙珠

6.1 线程协作工具类

常用四个工具类：

4. Condition接口：

- 控制线程的“等待”和“唤醒”
- 方法：
  ① await()：阻塞线程
  ② signal()：唤醒被阻塞的线程
  ③ signalAll()会唤起所有正在等待的线程。
- 注意：
  ① 调用await()方法时必须持有锁，否则会抛出异常
  ② Condition和Object#await/notify方法用法一样，两者await方法都会释放锁

案例：Tony仨小哥洗剪吹

6.2 小结

常用线程协作工具类总结：

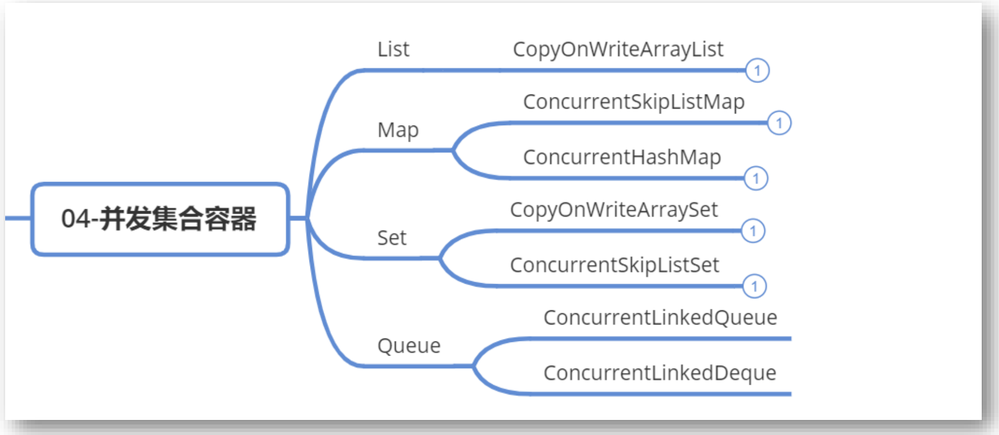
类	作用	说明
Semaphore	信号量，通过控制 **许可** 的数量来保证线程之间的配合	**场景：限流** 只有拿到**许可**才可运行
CyclicBarrier	线程会等待，直到线程到了事先规定的数目，然后触发执行条件进行下一步动作	**场景：并行计算** 线程之间相互等待处理结果就绪的场景
CountDownLatch	线程处于等待状态，指导计数减为0，等待线程才继续执行	**场景：购物拼团**
Condition	控制线程的**等待/唤醒**	**场景：线程协作** Object.wait()和notify()的升级版

6.3 并发容器

01-什么是并发容器？

针对多线程并发访问来进行设计的集合，称为并发容器

- JDK1.5之前，JDK提供了线程安全的集合都是**同步容器**，线程安全，只能串行执行，性能很差。
- JDK1.5之后，JUC并发包提供了很多并发容器，优化性能，替代同步容器



02-什么是同步容器？线程安全的集合与非安全集合有什么关系？

每次只有一个线程可以访问的集合（同步），称为线程安全的集合，也叫同步容器

- **Java集合主要为4类：List、Map、Set、Queue**，线程不安全的：ArrayList、HashMap..
- JDK早期线程安全的集合Vector、Stack、HashTable。
- JDK1.2中，还为Collections增加内部Synchronized类创建出线程安全的集合，实现原理synchronized

6.3 并发容器

03-常见并发容器特点总结

➢ List容器

① **Vector**：synchronized实现的同步容器，性能差，适合于对**数据有强一致性**要求的场景

② **CopyOnWriteArrayList**：底层数组实现，使用**复制副本**进行有锁写操作（数据不一致问题），适合读多写少，允许短暂的数据不一致的场景

➢ Map容器

① **Hashtable**： synchronized实现的同步容器，性能差，适合于对**数据有强一致性**要求的场景

② **ConcurrentHashMap**：**底层数组+链表+红黑树（JDK1.8）实现**，对table数组entry加锁（ synchronized ），存在一致性问题。适合存储**数据量小**，**读多写少**，允许短暂的数据不一致的场景

③ **ConcurrentSkipListMap**：底层跳表实现，使用CAS实现无锁读写操作。适合与存储**数据量大**，**读写频繁**，允许短暂的数据不一致的场景

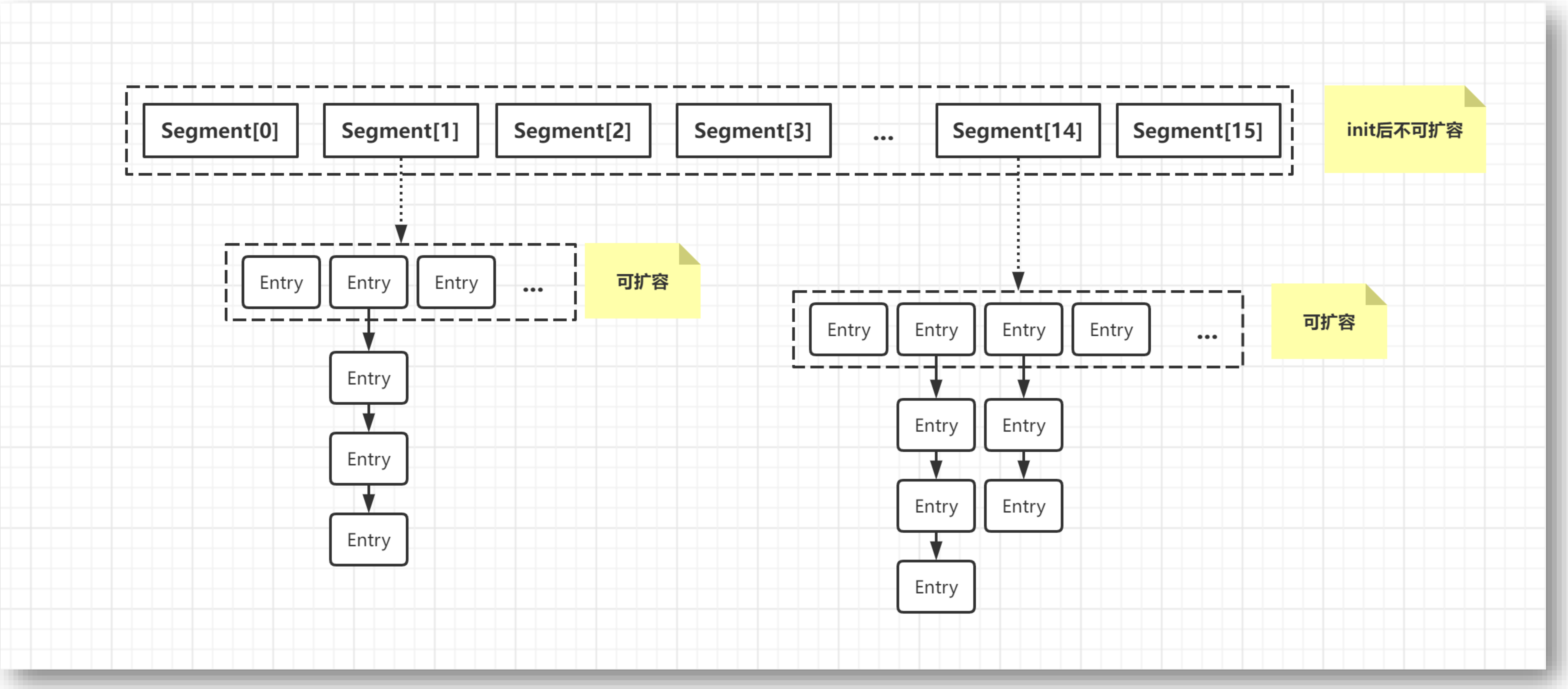
➢ Set容器

① **CopyOnWriteArraySet**：底层数组实现的无序Set

② **ConcurrentSkipListSet**：底层基于跳表实现的有序Set

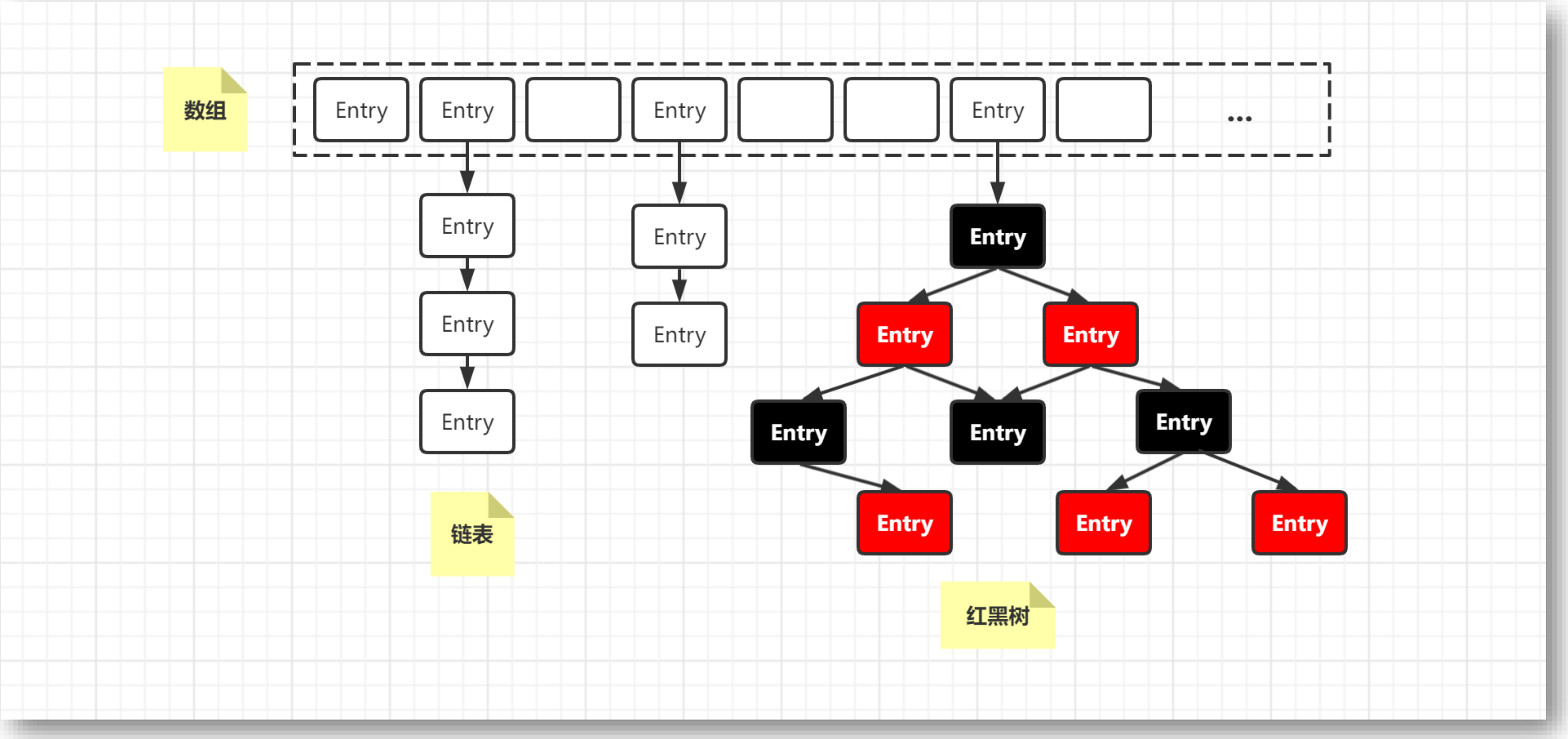
6.4 并发容器-ConcurrentHashMap

01-JDK1.7结构图



02-JDK1.8结构图

- 底层采用数组+链表+红黑树数据结构
- 存入key值，使用hashCode映射数组索引
- 集合会自动扩容：加载因子0.75f
- 链表长度超过8时，链表转换为红黑树



Key怎么进入容器对应的位置呢？

6.5 并发容器-CopyOnWriteArrayList

CopyOnWriteArrayList底层数组实现，使用**复制副本**进行有锁写操作，适合读多写少，允许短暂的数据不一致的场景。

CopyOnWrite思想：平时查询时，不加锁，更新时从原来的数据copy副本，然后修改副本，最后把原数据替换为副本。修改时，不阻塞读操作，读到的是旧数据。

做个案例来看一下

优缺点

- **优点：**对于读多写少的场景， CopyOnWrite这种无锁操作性能更好，相比于其它同步容器
- **缺点：**①数据一致性问题，②内存占用问题及导致更多的GC次数

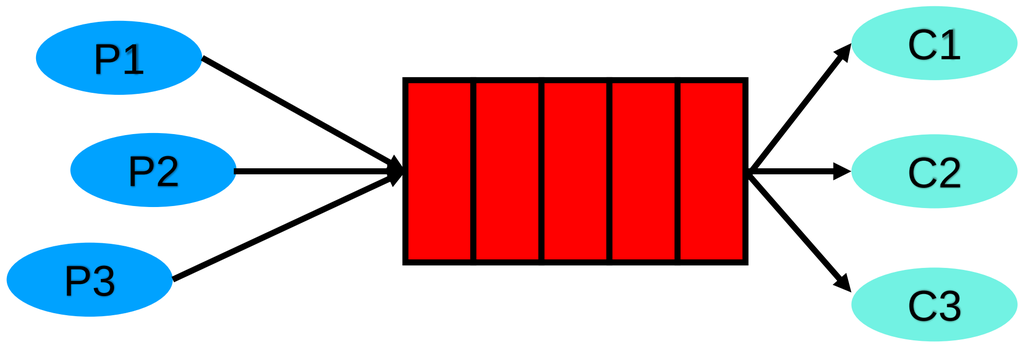
6.6 并发队列

01-为什么要用队列？

队列是线程协作的利器，通过队列可以很容易的实现数据共享，并且解决上下游处理速度不匹配的问题，典型的**生产者消费者模式**

02-什么是阻塞队列？

- **带阻塞能力的队列**，阻塞队列一端是给生产者put数据使用，另一端给消费者take数据使用
- 阻塞队列是线程安全的，生产者和消费者都可以是多线程
- take方法：获取并移除头元素，**如果队列无数据，则阻塞**
- put方法：插入元素，**如果队列已满，则阻塞**
- **阻塞队列又分为有界和无界队列，**无界队列不是无限队列，最大值Integer.MAX_VALUE



6.6 并发队列

03-常用阻塞队列：

1. ArrayBlockingQueue 基于数组实现的有界阻塞队列

2. LinkedBlockingQueue 基于**链表**实现的**无界**阻塞队列

3. SynchronousQueue不存储元素的阻塞队列

4. PriorityBlockingQueue **支持按优先级**排序的**无界**阻塞队列

5. DelayQueue优先级队列实现的**双向无界**阻塞队列

6. LinkedTransferQueue基于**链表**实现的**无界**阻塞队列

7. LinkedBlockingDeque基于**链表**实现的**双向无界**阻塞队列

七、线程池

7.1 线程池

01-线程池简介

线程池（ThreadPool） 是一种基于**池化思想管理线程**的工具。线程池维护多个线程，等待监督和管理分配可并发执行的任务。看过new Thread源码后我们发现，频繁创建线程销毁线程的开销很大，会降低系统整体性能。

02-优点

- **降低资源消耗：**通过线程池复用线程，降低创建线程和释放线程的损耗
- **提高响应速度：**任务到达时，无需等待即刻运行
- **提高线程的可管理性：**使用线程池可以进行统一的线程分配、调优和监控
- **提供可扩展性：**线程池具备可扩展性，研发人员可以向其中增加各种功能，比如：延时、定时、监控等

03-使用场景

- **连接池：**预先申请数据库连接，提升申请连接的速度，降低系统的开销（跨网络应用都需要线程池）
- **线程隔离：**服务器接收大量请求，使用线程池来进行隔离处理
- 开发中，如需创建5以上线程，就可以考虑用线程池

7.2 线程池的核心参数

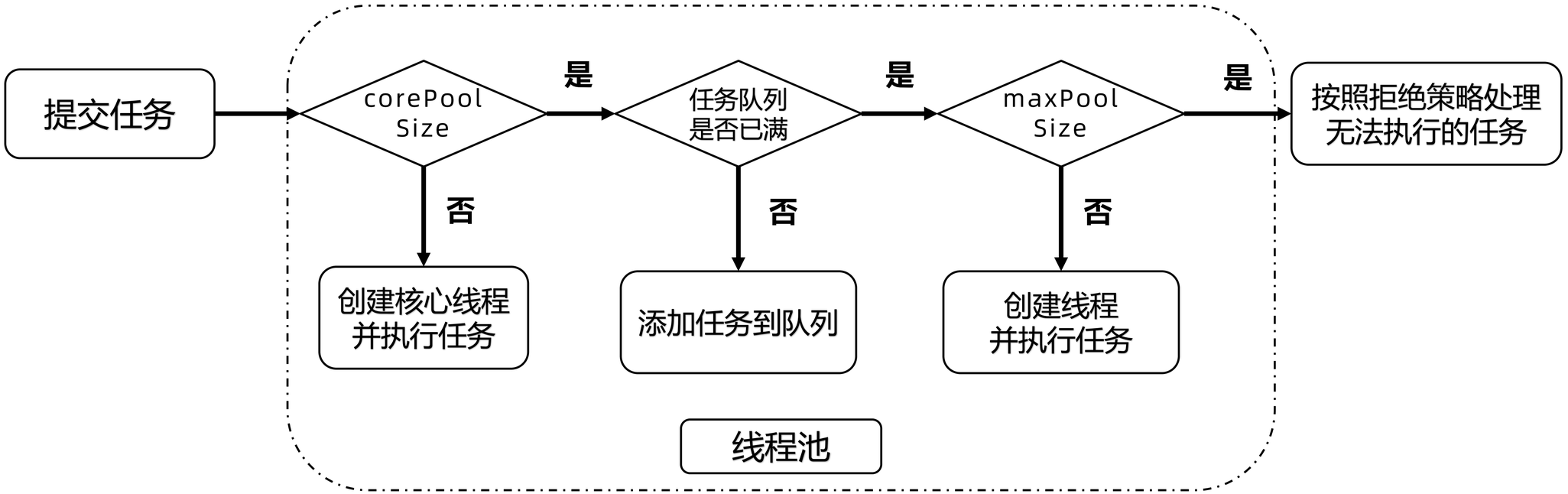
参数名	类型	含义
corePoolSize	int	核心线程数
maxPoolSize	int	最大线程数
keepAliveTime	long	保持存活时间
workQueue	BlockingQueue	任务存储队列
threadFactory	ThreadFactory	线程池创建新线程的线程工厂类
Handler	RejectedExecutionHandler	线程无法接收任务时的拒绝策略

参数详解：

- corePoolSize：核心线程数，可以理解为空闲线程数，即便线程空闲（Idle），也不会回收
- maxPoolSize ： 最大线程数，线程池可以容纳线程的上限
- keepAliveTime ： 线程保持存活的时间，超过核心线程数的线程存活空闲时间超过keepAliveTime后就会被回收
- workQueue ： 工作队列，直接交换队列SynchronousQueue，无界队列LinkedBlockingQueue ，有界队列ArrayBlockingQueue
- threadFactory：线程工厂，用来创建线程的工厂，线程都是出自于此工厂
- Handler：线程无法接收任务时的拒绝策略

这些参数具体如何运用？看一下线程池原理

7.3 线程池原理



① 提交任务，如果线程数小于corePoolSize即使其他线程处于空闲状态，也会创建一个新线程来运行任务

② 如果线程数大于corePoolSize，但少于maxPoolSize，将任务放入工作队列

③ 如果队列已满，并且线程数小于maxPoolSize，则创建一个新线程来运行任务。

④ 如果队列已满，并且线程数大于或等于maxPoolSize，则拒绝该任务。

7.4 自动创建线程

四种：

①**newFixedThreadPool**：固定数量线程池，无界任务阻塞队列

②**newSingleThreadExecutor**：一个线程的线程池，无界任务阻塞队列

③**newCachedThreadPool**：可缓存线程的无界线程池，可以自动回收多余线程

④**newScheduledThreadPool**：定时任务线程池

做个案例来看一下，然后分析一下源码

7.5 手动创建线程

有些企业开发规范中会禁止使用快捷方式创建线程池，要求使用标准构造器 ThreadPoolExecutor 创建

```
// 使用标准构造器，构造一个普通的线程池
public ThreadPoolExecutor(
    int corePoolSize, // 核心线程数，即使线程空闲（Idle），也不会回收；
    int maximumPoolSize, // 线程数的上限；
    long keepAliveTime, TimeUnit unit, // 线程最大空闲（Idle）时长
    BlockingQueue workQueue, // 任务的排队队列
    ThreadFactory threadFactory, // 新线程的产生方式
    RejectedExecutionHandler handler) // 拒绝策略
```

根据不同的业务场景，自己设置线程池的参数、线程名、任务被拒绝后如何记录日志等

如何设置线程池大小？

- CPU密集型：线程数量不能太多，可以设置为与相当于CPU核数
- IO密集型：IO密集型CPU使用率不高，可以设置的线程数量多一些，可以设置为CPU核心数的2倍

拒绝策略：

- **拒绝时机：**①最大线程和工作队列有限且已经饱和，②Executor关闭时
- 抛异常策略：**AbortPolicy，**说明任务没有提交成功
- 不做处理策略：**DiscardPolicy，**默默丢弃任务，不做处理
- 丢弃老任务策略：**DiscardOldestPolicy，**将队列中存在最久的任务给丢弃
- 自产自销策略：**CallerRunsPolicy，**那个线程提交任务就由那个线程负责运行

案例：手写网站服务器

八、ThreadLocal

8.1 ThreadLocal简介

01-什么是ThreadLocal?

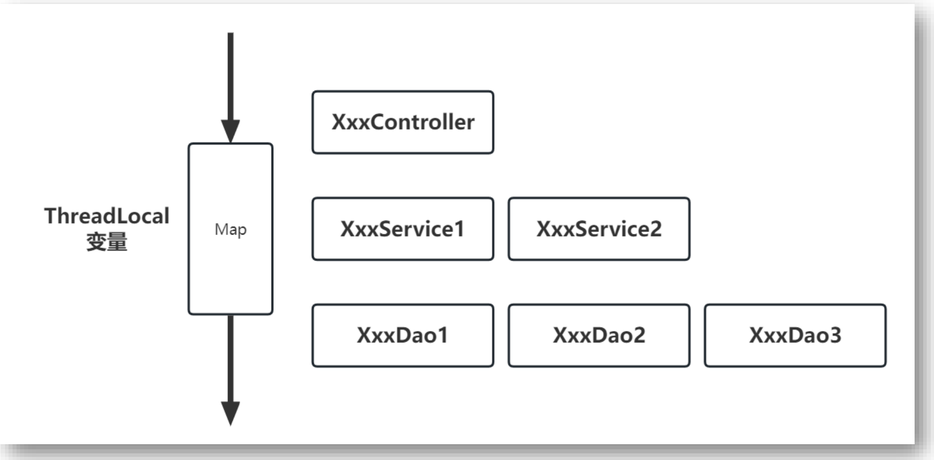
ThreadLocal是线程本地变量类，在多线程并执行过程中，将变量存储在ThreadLocal中，每个线程中都有独立的变量，因此不会出现线程安全问题。

举例：

- **解决线程安全问题：**每个线程绑定一个数据库连接，避免多个线程访问同一个数据库连接：**SqlSession**
- **跨函数参数传递：**同一个线程，跨类，跨方法传递参数时可以使用ThreadLocal，每个线程绑定一个**Token/Session**

```java
//伪代码
private static final ThreadLocal localSqlSession = new ThreadLocal();

public void startManagedSession() {
    this.localSqlSession.set(openSession());
}
@Override
public Connection getConnection() {
    final SqlSession sqlSession = localSqlSession.get();
    if (sqlSession == null) {
        throw new SqlSessionException("Error:  Cannot get connection.  No managed session is started.");
    }
    return sqlSession.getConnection();
}
```



8.2 ThreadLocal底层原理

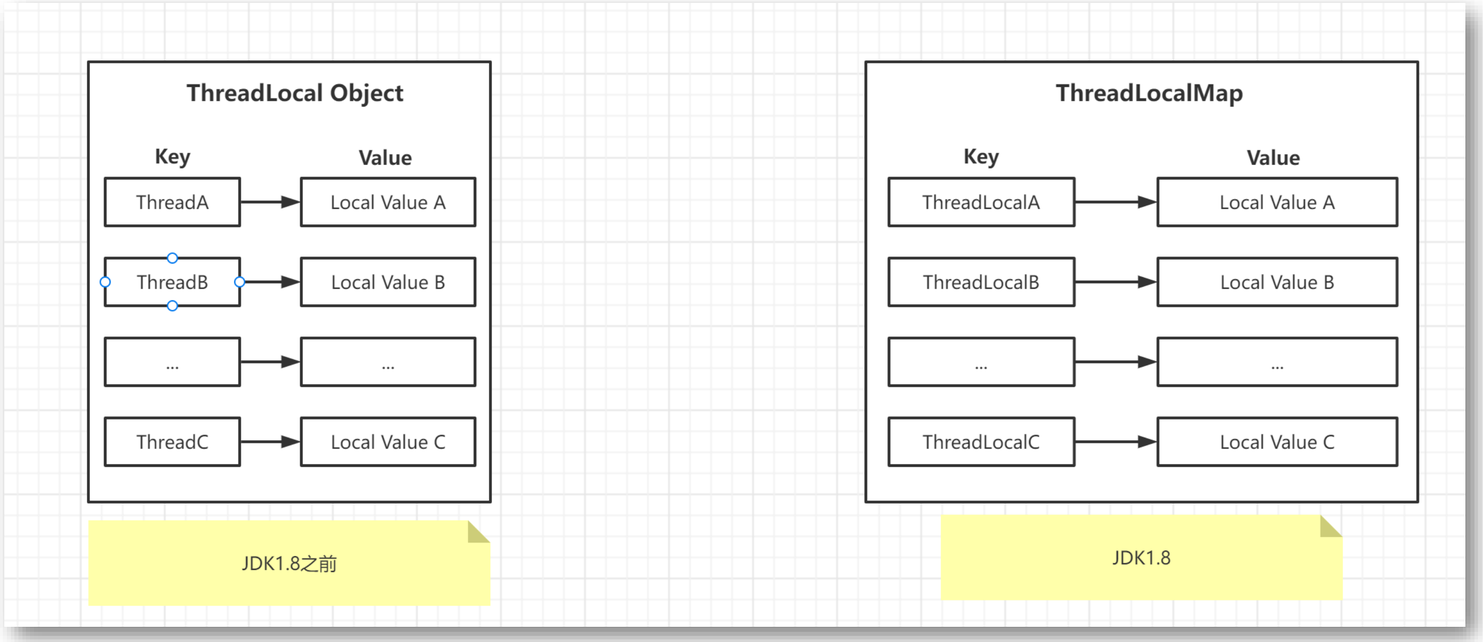
JDK1.8之前：ThreadLocal是Map所有线程拥有同一个，Key为thread，Value为具体值

JDK1.8：ThreadLocal依旧是Map，但一个线程一个ThreadLocalMap，key为ThreadLocal，Value为具体值

主要变化：①ThreadLocalMap的拥有者，②Key

JDK1.8中Thread、ThreadLocal、ThreadLocalMap的关系？

- **Thread --> ThreadLocalMap --> Entry(ThreadLocalN, LocalValueN)*n**



Map的Key为什么要用弱引用？

ThreadLocal内存溢出的原因是什么？

6. 【强制】必须回收自定义的 ThreadLocal 变量，尤其在线程池场景下，线程经常会被复用，如果不清理自定义的 ThreadLocal 变量，可能会影响后续业务逻辑和造成内存泄露等问题。尽量在代理中使用 try-finally 块进行回收。

正例：

```
objectThreadLocal.set(userInfo);
try {
    // ...
} finally {
    objectThreadLocal.remove();
}
```

8.3 ThreadLocal底层原理-Entry的弱引用Key

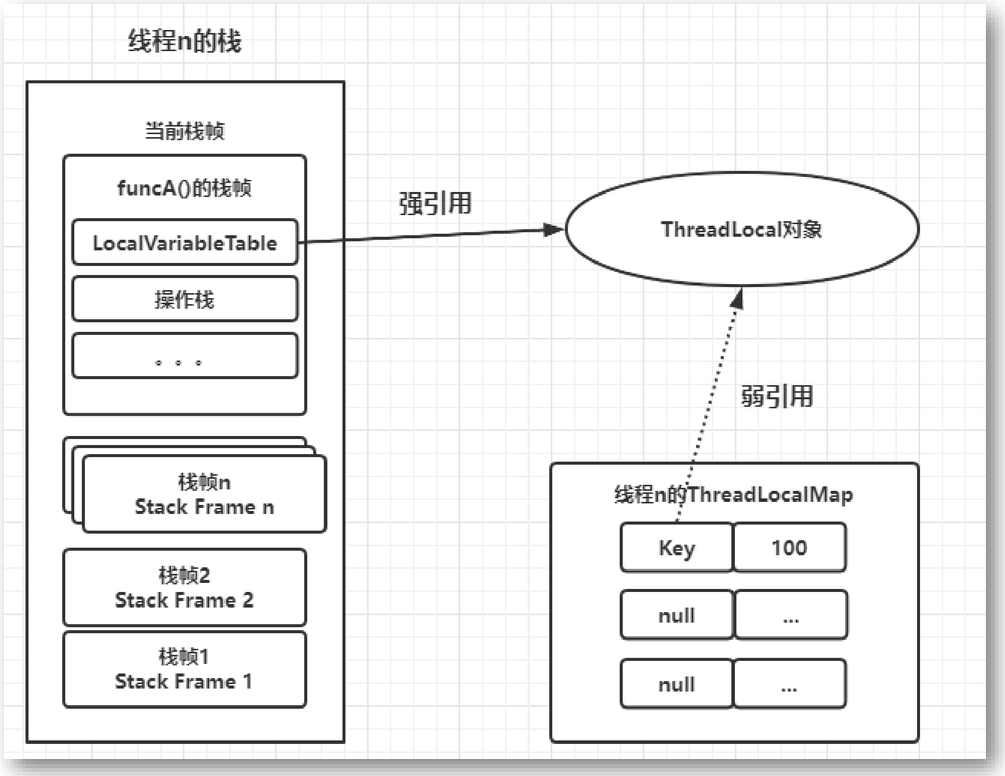
```java
// Entry 继承了 WeakReference,并使用 WeakReference 对 Key 进行包装
static class Entry extends WeakReference<ThreadLocal<?>> {
    Object value; //值
    Entry(ThreadLocal<?> k, Object v) {
        super(k); //使用 WeakReference 对 Key 值进行包装
        value = v;
    }
}
```

看源码：为什么 Entry 需要使用弱引用对 Key 进行包装，而不是直接使用 Threadlocal 实例作为 Key呢？

上代码：

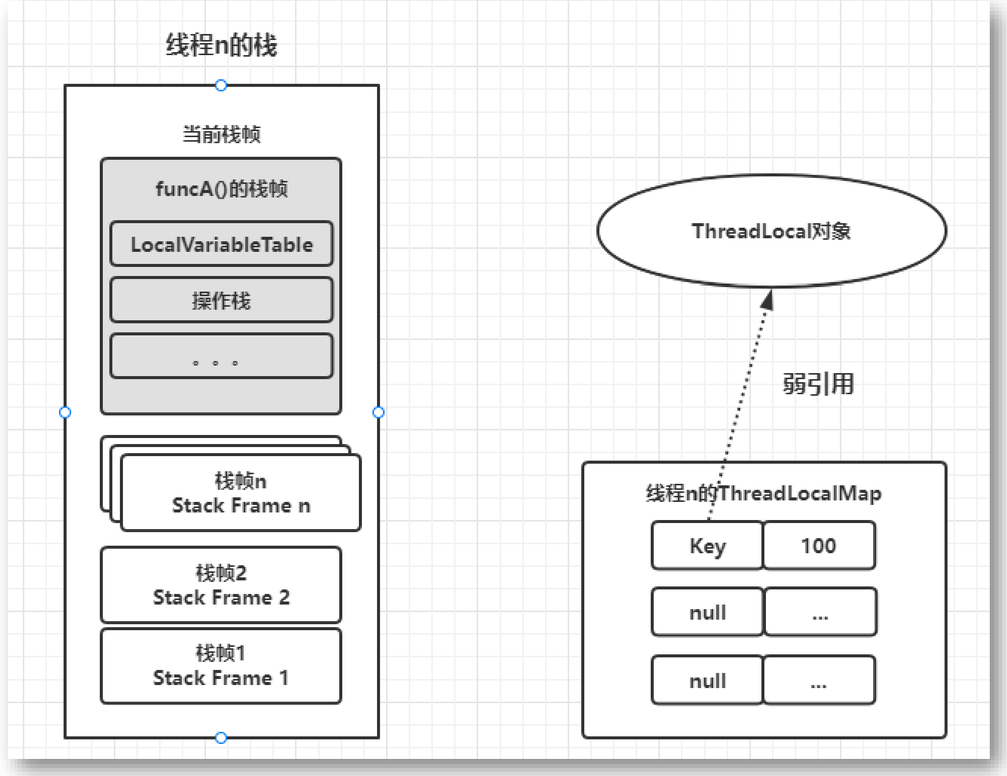
```java
//伪代码
public void funcA() {
    //创建一个线程本地变量
    ThreadLocal local = new ThreadLocal();
    //设置值
    local.set(100);
    //获取值
    local.get();
    //函数末尾
}
```

8.3 ThreadLocal底层原理-Entry的弱引用Key



funcA()方法入栈

- 新建ThreadLocal对象，local局部变量指向它，强引用
- local.set(100)，key为弱引用指向ThreadLocal



funcA()方法出栈

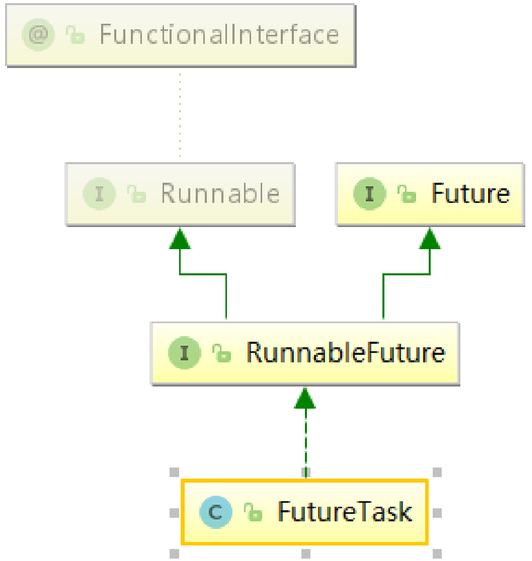
- 栈帧被销毁，强引用没有了，但弱引用还存在
- 如果此引用是强引用，则不能被GC回收对象

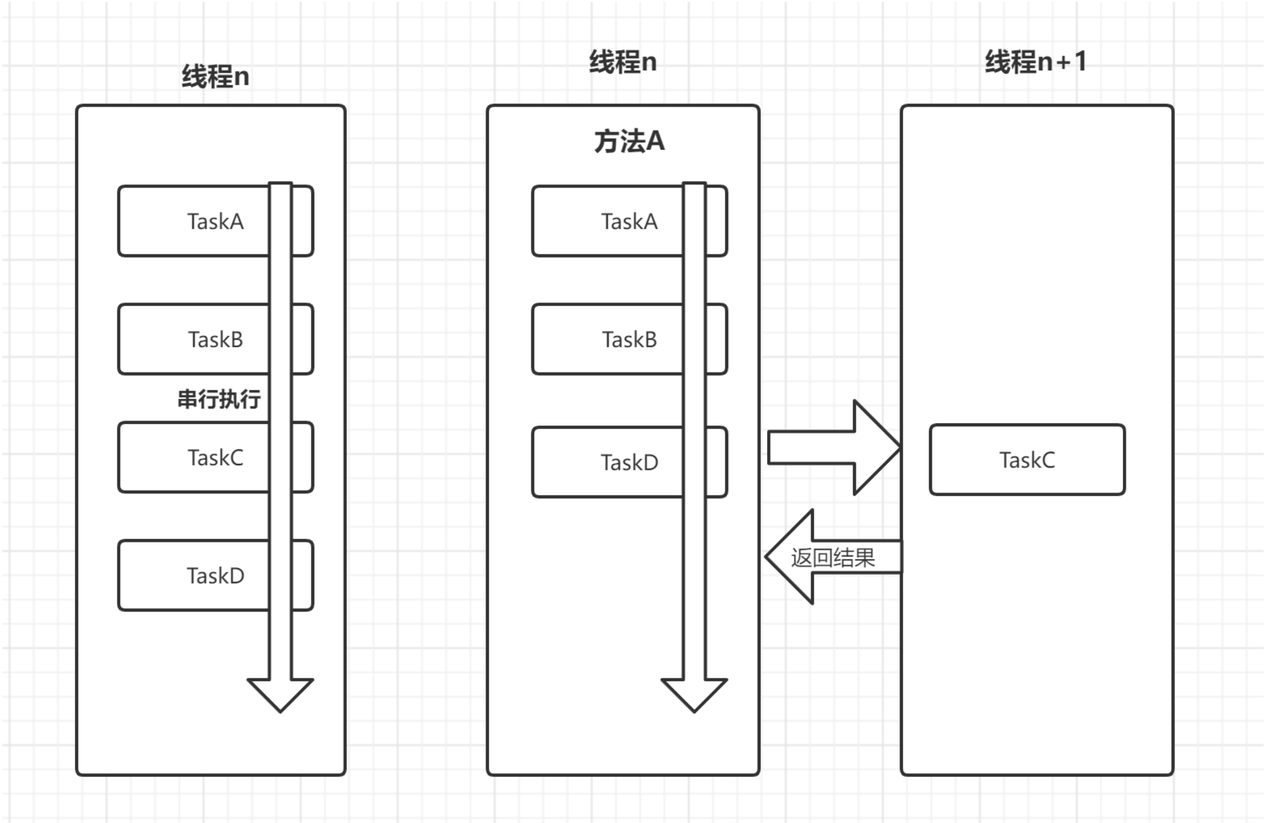
九、Future和FutureTask

9.1 Future与FutureTask简介

01-什么是Future？

FutureTask叫未来任务，可以将一个复杂的任务剔除出去，交给另一个线程来完。它是Future的实现类





9.1 Future与FutureTask简介

02-Future主要方法：

- **get()：**方法返回结果取决于Callable任务执行的状态，任务有五种状态
  - ① 正常完成：get立刻返回结果
  - ② 尚未完成：还没开始或进行中的状态，get将阻塞直到任务完成
  - ③ 抛出异常：get会抛出ExecutionException
  - ④ 被取消： get会抛出CancellationException
  - ⑤ 超时：设置超时时间，时间到了还没结果，会抛出TimeoutException
- **get(timeout,TimeUnit)：**设置任务完成时间，没到则抛异常
- **cancel()：**取消任务时，有三种情况
  - ① 如果这个任务还没开始，任务会被取消，返回true
  - ② 如果任务已经完成或已取消，返回false
  - ③ 如果任务已经开始，则方法不会直接取消任务，而会判断是否可以取消，如果可以才会发出中断信号
- **isDone()：**判断是否执行完成
- **isCancelled()：** 判断是否被取消

9.2 Future与FutureTask案例

Future用法01-用线程池submit方法提交任务，返回值Future任务结果

- 用线程池提交任务，线程池会立即返回一个空的Future容器
- 当线程的任务执行完成，线程池会将该任务执行结果填入Future中
- 此时就可以从Future获取执行结果

Future用法02-用FutureTask来封装任务，获取Future任务的结果

- 用FutureTask包装任务，FutureTask是Future和Runnable接口的实现类
- 可以使用new Thread().start()或线程池执行FutureTask
- 任务执行完成，可以从FutureTask中获取执行结果

做个案例看一下

THANKS

教育不是注满一桶水，而是点燃一把火