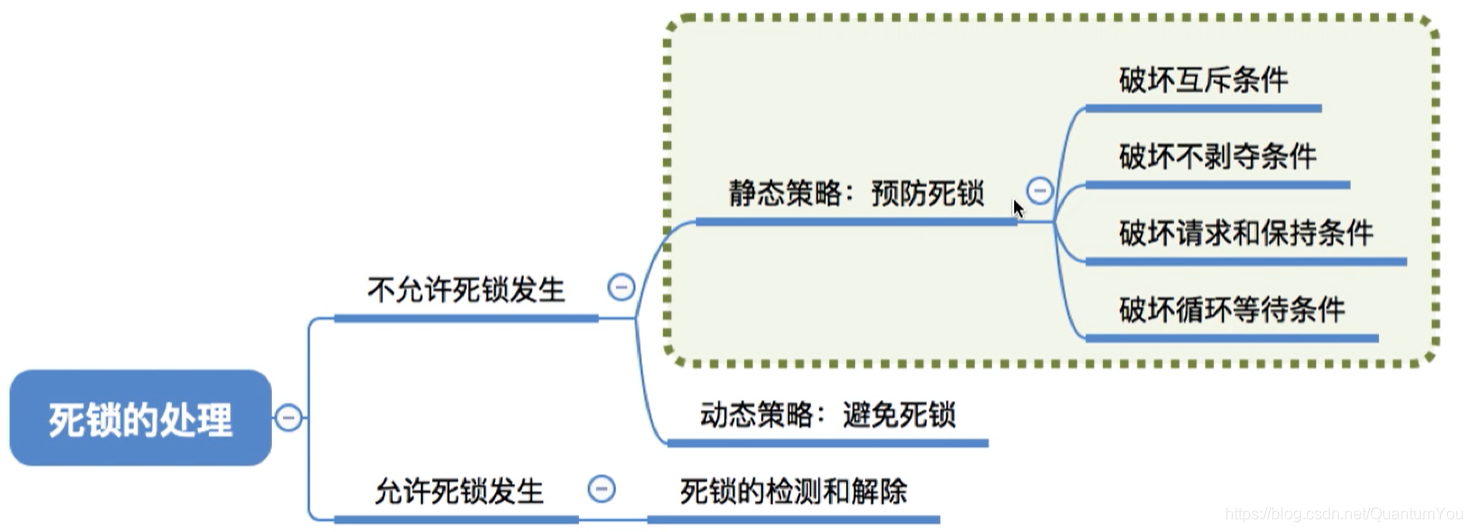
@[toc]

# 预防死锁（静态策略）



## 破坏互斥资源

* 互斥条件：只有对必须互斥使用的资源的争抢才会导致死锁。

### 方案

* 如果把只能互斥使用的资源改造为允许共享使用，则系统不会进入死锁状态。比如： SPOOLing技术。操作系统可以采用 SPOOLing技术把独占设备在逻辑上改造成共享设备。比如，用 SPOOLing技术将打印机改造为共享设备。



## 破坏不剥夺资源

* 不剥夺条件：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放。

### 方案

方案一：

* 当某个进程请求新的资源得不到满足时，它必须立即释放保持的所有资源，待以后需要时再重新申请。也就是说，即使某些资源尚未使用完，也需要主动释放，从而破坏了不可剥夺条件。

方案二：

* 当某个进程需要的资源被其他进程所占有的时候，可以由操作系统协助，将想要的资源强行剥夺。这种方式一般需要考虑各进程的优先级（比如：剥夺调度方式，就是将处理机资源强行剥夺给优先级更高的进程使用）

### 缺点

* 1、实现起来比较复杂。
* 2、释放已获得的资源可能造成前一阶段工作的失效。因此这种方法一般只适用于易保存和恢复状态的资源，如CPU.
* 3、反复地申请和释放资源会增加系统开销，降低系统吞吐量。
* 4、若采用方案一，意味着只要暂时得不到某个资源，之前获得的那些资源就都需要放弃，以后再重新申请。如果一直发生这样的情况，就会导致进程饥饿。

## 破坏请求和保持条件

* 请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其他进程占有，此时请求进程被阻塞，但又对自己已有的资源保持不放。

### 方案

* 可以采用静态分配方法，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前不让它投入运行。一旦投入运行后，这些资源就一直归它所有，该进程就不会再请求别的任何资源

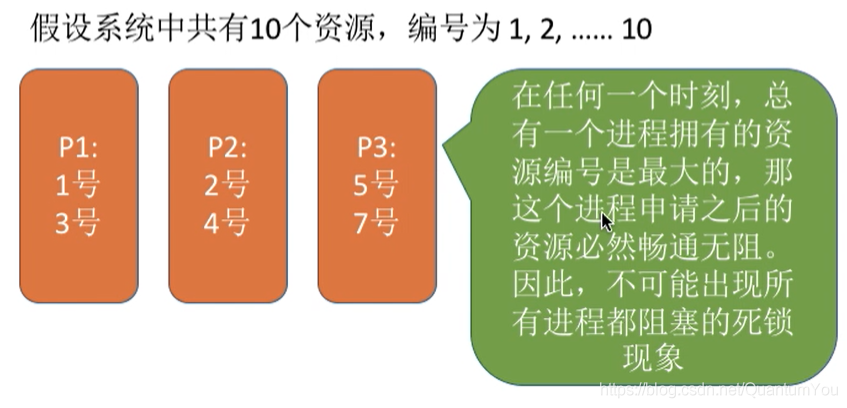
### 缺点

* 该策略实现起来简单，但也有明显的缺点：有些资源可能只需要用很短的时间，因此如果进程的整个运行期间都一直保持着所有资源，就会造成严重的资源浪费，资源利用率极低。另外，该策略也有可能导致某些进程伣饿。（如下图容易导致C类进程饥饿）  
  

## 破坏循环等待条件

* 循环等待条件：存在一种进程资源的循环等待链，链中的每一个进程已获得的资源同时被下一个进程所请求。

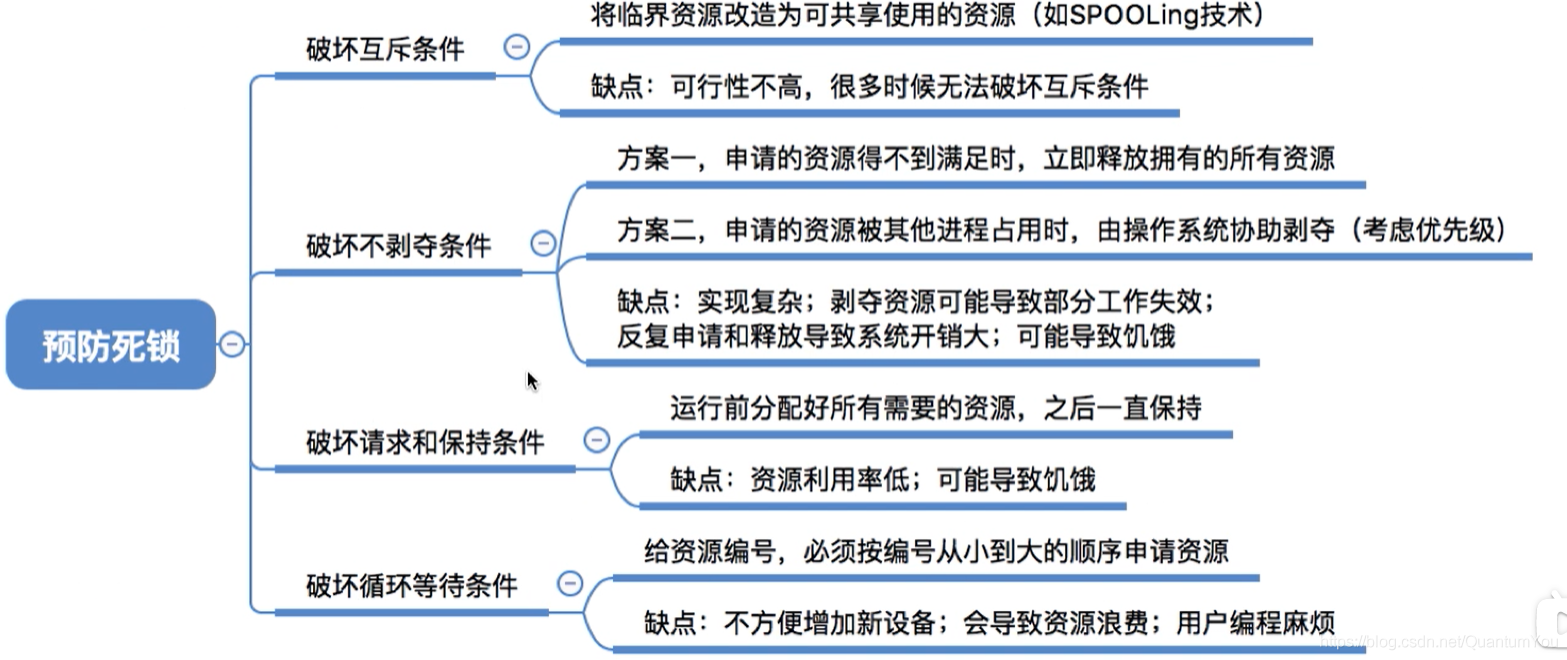
### 方案

* 可采用顺序资源分配法。首先给系统中的资源编号，规定每个进程必须按编号递增的顺序请求资源，同类资源（即编号相同的资源）一次申请完。
* **原理分析**：一个进程只有已占有小编号的资源时，才有资格申请更大编号的资源。按此规则，已持有大编号资源的进程不可能逆向地回来申请小编号的资源，从而就不会产生循环等待的现象。  
  

### 缺点

* 1、不方便增加新的设备，因为可能需要重新分配所有的编号；
* 2、进程实际使用资源的顺序可能和编号递增顺序不一致，会导致资源浪费；
* 3、必须按规定次序申请资源，用户编程麻烦。

## 小结



# 避免死锁（动态策略）

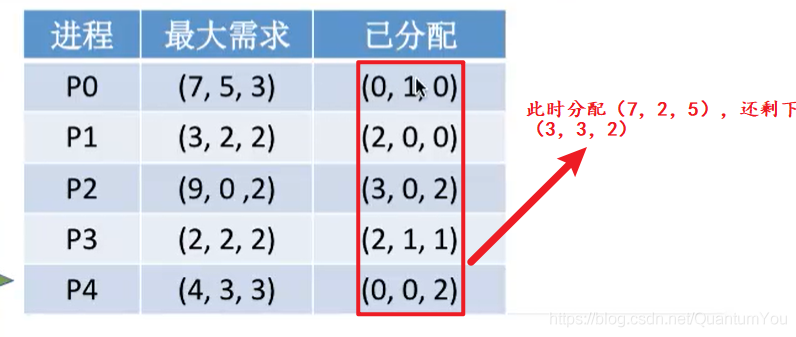
## 安全序列

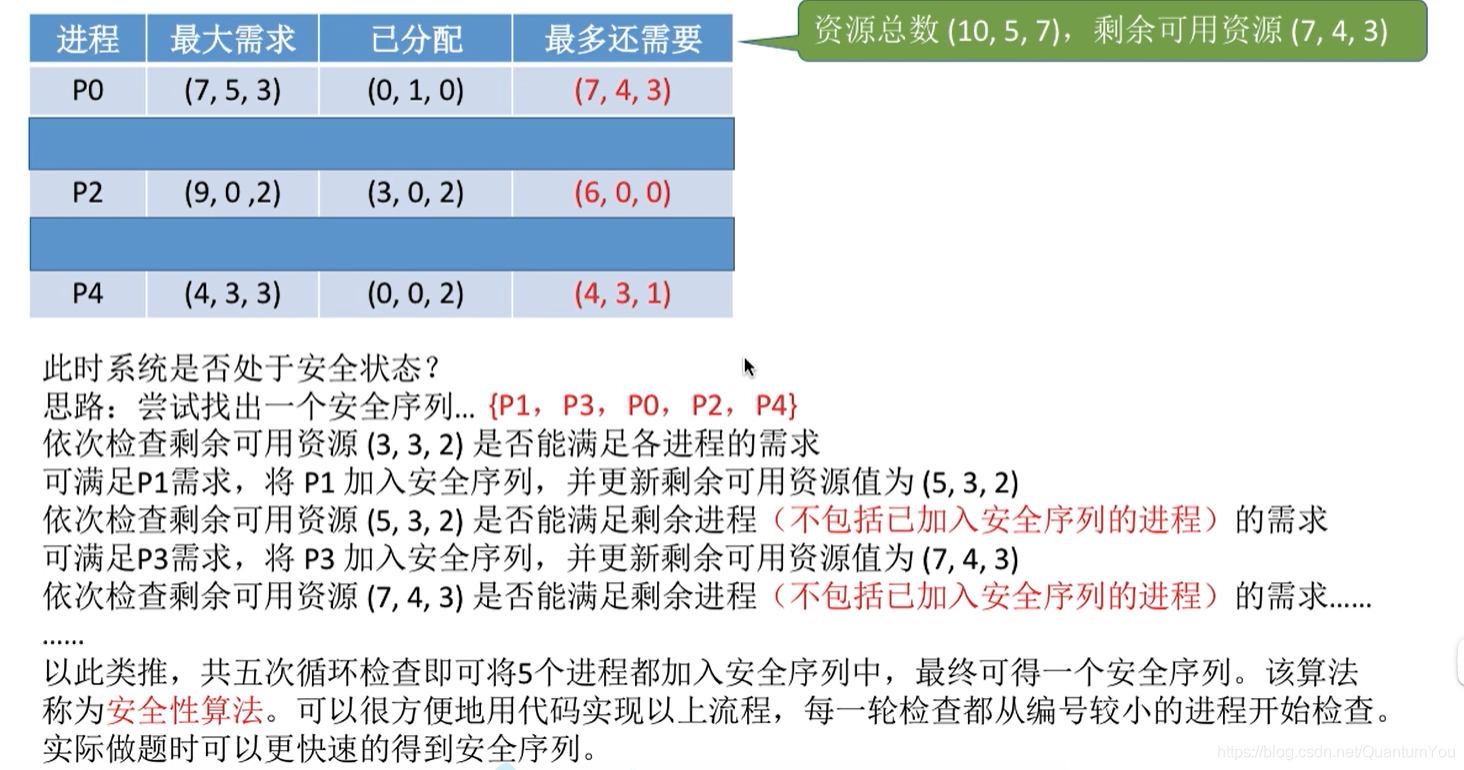
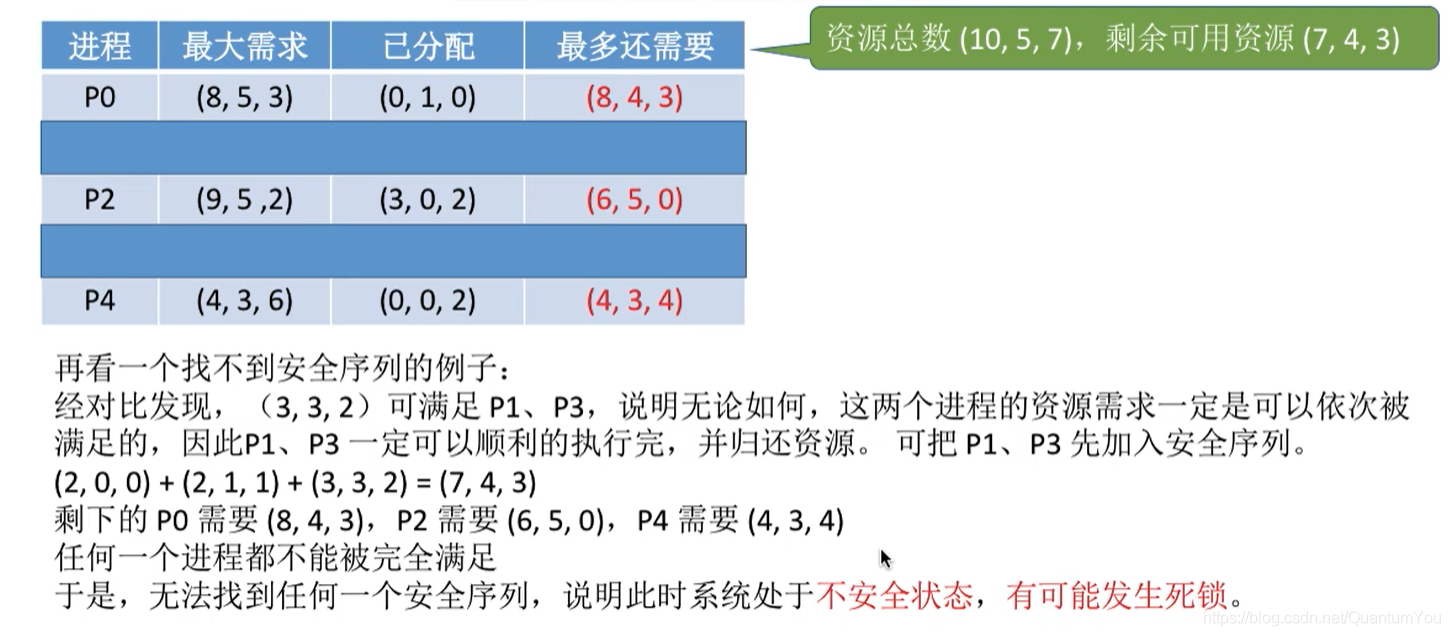
* 安全序列，就是指如果系统按照这种序列分配资源，则每个进程都能顺利完成。只要能找出一个安全序列，系统就是安全状态。当然，安全序列可能有多个。
* 如果分配了资源之后，系统中找不出任何一个安全序列，系统就进入了不安全状态。这就意味着之后可能所有进程都无法顺利的执行下去。当然，如果有进程提前归还了一些资源，那系统也有可能重新回到安全状态，不过我们在分配资源之前总是要考虑到最坏的情况。
* 注意：如果系统处于安全状态，就一婝不会发生死锁。如果系统进入不安全状态，就可能发生死锁（处于不安全状态未必就是发生了死锁，但发生死锁时一定是在不安全状态）因此,可以在资源分配之前预先判断这次分配是否会导致系统进入不安全状态，以此决定是否答应资源分配请求。这也是“银行家算法”的核心思想。

**核心思想**：在进程提出资源申请时，先预判此次分配是否会导致系统进入不安全状态。如果会进入不安全状态，就暂时不答应这次请求，让该进程先阻塞等待。

## 银行家算法

* 将算法拓展为多种资源的情况：

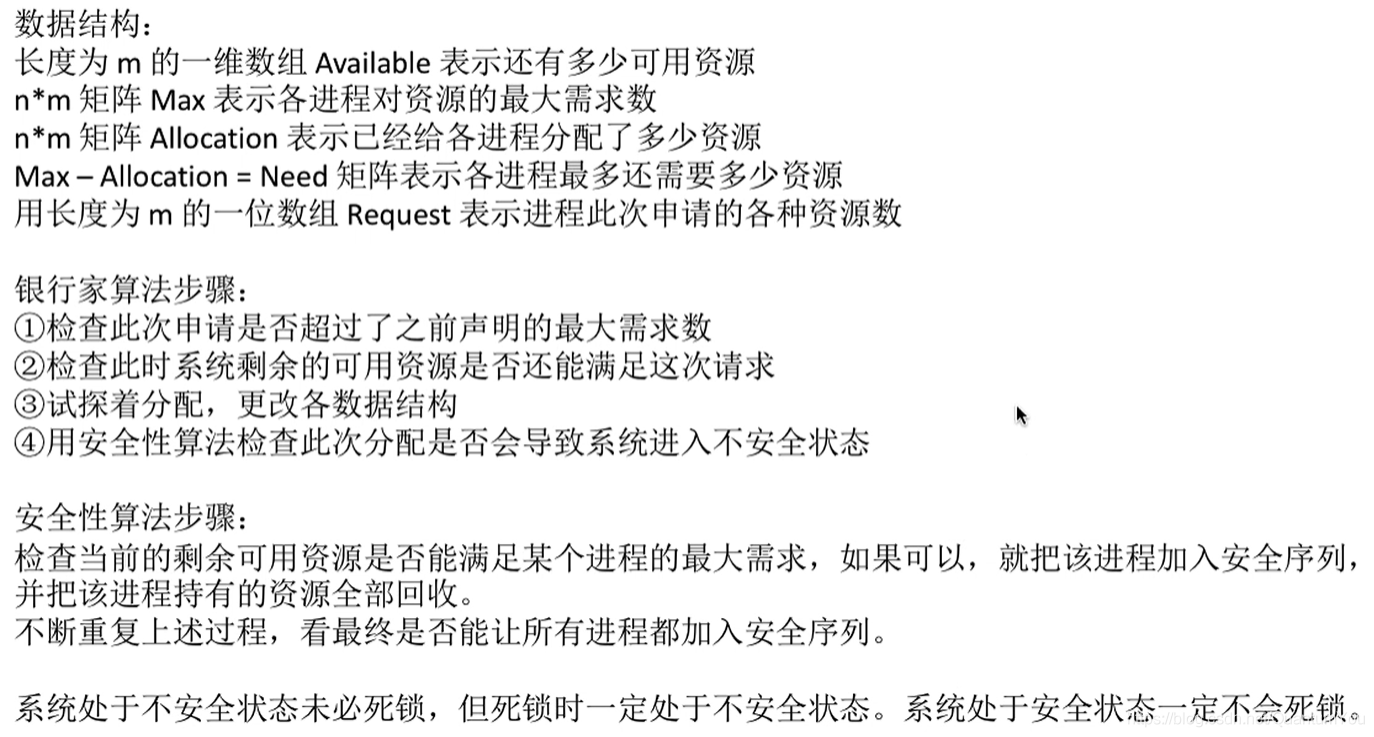
可以把单维的数字拓展为多维的向量。比如：系统中有5个进程P0--P4，3种资源R0--R2，初始数量为（10，5，7），则某一时刻的情况可表示如下：  


**满足的情况**：  
  
**不满足的情况**：  


### 代码实现



## 小结

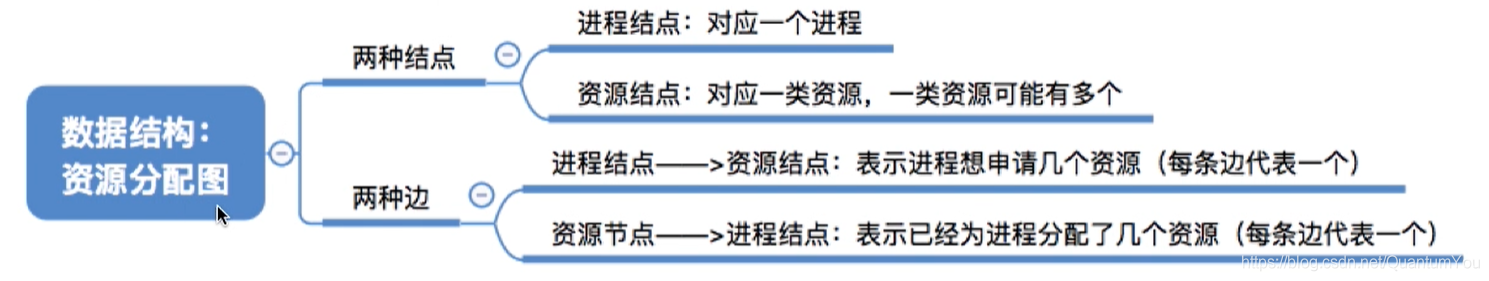
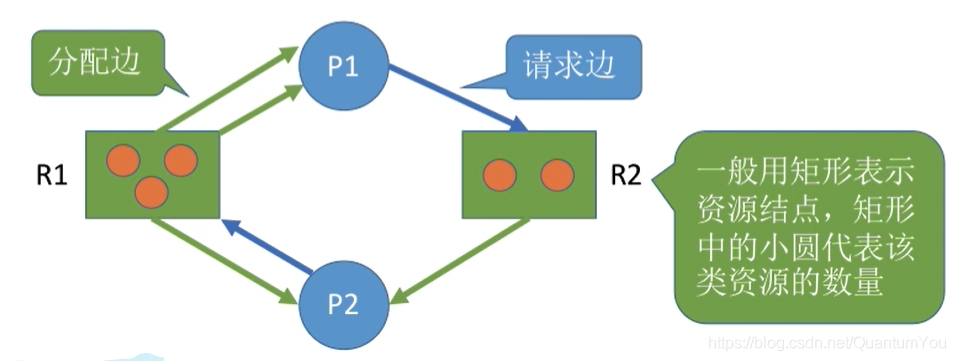


# 死锁的检测和解除

* 如果系统中既不采取预防死锁的措施，也不釆取避免死锁的措施，系统就很可能发生死锁。在这种情况下，系统应当提供两个算法。
* ①死锁检测算法：用于检测系统状态，以确定系统中是否发生了死锁。
* ②死锁解除算法：当认定系统中已经发生了死锁，利用该算法可将系统从死锁状态中解脱岀来。

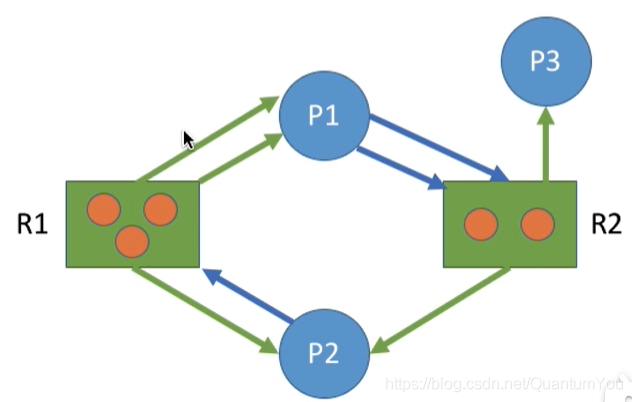
## 死锁的检测

* 为了能对系统是否已发生了死锁进行检测，必须
* ①用某种数据结构来保存资源的请求和分配信息；
* ②提供一种算法，利用上述信息来检测系统是否已进入死锁状态。

* 如果系统中剩余的可用资源数足够满足进程的需求，那么这个进程暂时是不会阻塞的，可以顺利地执行下去如果这个进程执行结束了把资源归还系统，就可能使某些正在等待资源的进程被激活，并顺利地执行下去。相应的，这些被激活的进程执行完了之后又会归还一些资源，这样可能又会激活另外一些阻塞的进程。
* 如果按上述过程分析，最终能消除所有边，就称这个图是可完全简化的。此时一定没有发生死锁（相当于能找到一个安全序列）

如果最终不能消除所有边，那么此时就是发生了**死锁**。

* 下图即可以会发生死锁  
    
  最终还连着边的那些进程就是处于死锁状态的进程。

## 死锁检测算法

* 1、在资源分配图中，找出既不阻塞又不是孤点的进程Pi（即找出一条有向边与它相连，且该有向边对应资源的申请数量小于等于系统中已有空闲资源数量。如下图中，R1没有空闲资源，R2有一个空闲资源。若所有的连接该进程的边均满足上述条件，则这个进程能继续运行直至完成，然后释放它所占有的所有资源）.消去它所有的请求边和分配变，使之称为孤立的结点。在下图中，P1是满足这一条件的进程结点，于是将P1的所有边消去。
* 2、进程Pi所释放的资源，可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变为非阻塞进程。在下图中，P2就满足这样的条件。根据1）中的方法进行一系列简化后，若能消去途中所有的边，则称该图是可完全简化的。
* 死锁定理：如果某时刻系统的资源分配图是不可完全简化的，那么此时系统死锁。

一旦检测出死锁的发生，就应该立即解除死锁。

注意：并不是系统中所有的进程都是死锁状态，用死锁检测算法化简资源分配图后，还连着边的那些进程就是死锁进程

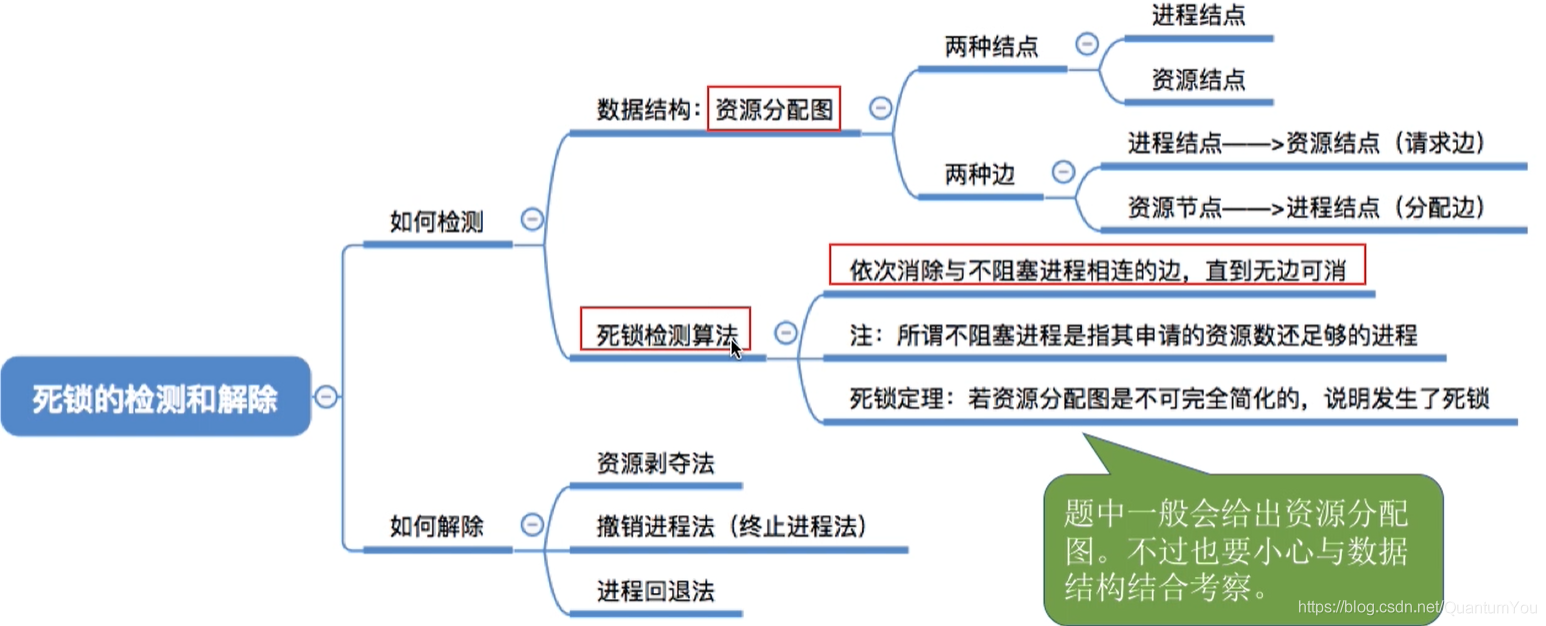
## 解除死锁方法

* 1、资源剥夺法。挂起（暂时放到外存上）某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但是应防止被挂起的进程长时间得不到资源而饥饿
* 2、撤销进程法（或称终止进程法）.强制撤销部分、甚至全部死锁进程，并剥夺这些进程的资源。这种方式的优点是实现简单，但所付出的代价可能会很大。因为有些进程可能已经运行了很长时间，已经接近结束了，一旦被终止可谓功亏一篑，以后还得从头再来。
* 3、进程回退法。让一个或多个死锁进程回退到足以避免死锁的地步。这就要求系统要记录进程的历史信息，设置还原点。

**影响因素**

* 1、进程优先级
* 2、已执行多长时间
* 3、还要多久能完成
* 4、进程已经使用了多少资源
* 5、进程是交互式的还是批处理式的

## 小结

  
死锁检测算法的实现