死锁避免：
银行家算法

下面我们来介绍一种典型的死锁避免算法 这就是银行家算法。

银行家算法(BANKER'S ALGORITHM)

Dijkstra提出(1965)

仿照银行发放贷款时采取的控制方式而设计
一种死锁避免算法

不安全状态最后达到死锁 我们来看一下银行家算法它的应用条件

银行家算法(BANKER'S ALGORITHM)

应用条件：

1. 在固定数量的进程中共享数量固定的资源
2. 每个进程预先指定完成工作所需的最大资源数量
3. 进程不能申请比系统中可用资源总数还多的资源
4. 进程等待资源的时间是有限的
5. 如果系统满足了进程对资源的最大需求，那么
   进程应该在有限的时间内使用资源，然后归还给
   系统

这是我们做的一个 银行家算法应用的这么一个条件

银行家算法

简记符号：
Available
Max[i]
Allocation[i]
Need[i]
Request[i]

n：系统中进程数量
m：资源类数量

Available: ARRAY[1..m] of integer;
Max: ARRAY[1..n,1..m] of integer;
Allocation: ARRAY[1..n,1..m] of integer;
Need: ARRAY[1..n,1..m] of integer;
Request: ARRAY[1..n,1..m] of integer;

资源的申请是逐步进行的，所以这一次申请它需要多少 我们简记这样几个符号

银行家算法

当进程Pi提出资源申请时，系统执行下列步骤：

（1）若Request[i] ≤ Need[i]，转（2）；

否则，报错返回；

（2）若Request[i] ≤ Available，转（3）；

否则，进程等待；

（3）假设系统分配了资源，则有：

Available = Available - Request[i];

Allocation[i] = Allocation[i] + Request[i];

Need[i] = Need[i] - Request[i];

新状态

若系统新状态是安全的，则分配完成

若系统新状态是不安全的，则恢复原来状态，进程等待

分配之前的状态，而系统呢 让这个进程进入等待状态。

银行家算法

为进行安全性检查，定义数据结构：

Work: ARRAY[1..m] of integer;

Finish: ARRAY[1..n] of Boolean;

安全性检查的步骤：

(1) Work = Available;

Finish = false;

(2) 寻找满足条件的i：

a. Finish[i]==false;

b. Need[i]≤Work;

如果不存在，则转(4)

(3) Work = Work + Allocatio

Finish[i] = true;

转(2)

(4) 若对所有i，Finish[i]==true
则系统处于安全状态，否则系
统处于不安全状态

银行家算法应用

	目前占有量	最大需求量	尚需要量
P1	1	4	3
P2	4	6	2
P3	5	8	3
系统剩余量		2	

像这样一个场景所描述的系统状态，就是一个安全状态。

0:00 下面我们来介绍一种典型的死锁避免算法 这就是银行家算法。 银行家算法呢是 Dijkstra 1965 年提出来的 它的基本思想是仿照银行发放贷款的时候 所采取的控制方式而设计，它是一种 死锁避免算法。 我们来简单地 看一下银行发放贷款的一些过程 银行会跟很多的客户签订协议 贷款给这些客户。 但是客户呢并不是一次性 把贷款的总数一次借走 他可能分期、分批地来借 这样做的好处主要是 利息上会少一些，对于客户来讲有利 那么银行为了最大化自己的收益呢 它并不是完全按照自己现在有多少钱就和客户签多大的 贷款。 它可能会把贷款发放给不同的客户 这些客户每个人所申请的贷款不会超过银行总的这个资金数量 但是所有客户加起来所需要的总的贷款 量可能会超过目前银行占有的资金数量 所以银行在发放贷款的过程中 会要控制自己的风险，因为如果 银行针对客户的这个贷款没有及时发放到位 那么客户呢就 认为银行单方面的没有满足这个协议，所以客户对利息，将来的这个 贷款的这个额度的偿还呢就会产生问题 所以银行要不断地来控制说 如果某个客户这次要借一笔钱 它要看一看我现在的银行的总资金，剩余的总资金 能不能保证这些客户都能够还到钱 都能够把钱借到并且还回来 一旦发现了风险，它就暂时可以缓解，暂时 停止对这个客户的贷款的发放 以保证自己总体资金的安全 这就像我们对 操作系统当中的资源的分配一样，我们也要考虑到 在系统的资源数量有限的情况下，怎么能够 控制这个使得系统不会进入 不安全状态最后达到死锁 我们来看一下银行家算法它的应用条件 在应用银行家算法的时候，我们首先呢 要有一个固定数量的进程。 因为进程本身是可以创建可以 撤销的，但是我们现在在探讨银行家算法的时候 我们不能够 进程的数量随意变化，所以我们是固定数量的进程，并且是 固定数量的资源，而且这些进程就共享这些资源 第二个条件呢是每个进程必须预先指定 我要完成工作最大的资源的需求量，也就是说 客户和银行签订贷款的时候会告诉银行我最多借多少钱 当然，进程不能申请比系统中 可用资源总数还多的这样的资源 也就是说如果银行有一万块钱，这个客户不能说我要借一万一，这个银行没有这么多资金 那我们在应用银行家算法的时候呢，要保证进程 等待资源的时间是有限的，不能无限期地等待。 最后呢 我们假设如果系统满足了进程对资源的最大需求，那么 进程应该在有限的时间内去 完成这个资源的使用并且把资源还回给系统。 这是我们做的一个 银行家算法应用的这么一个条件 下面我们来介绍一下银行家算法 假设系统当中有 n 个进程，有 m 类资源 这些数据结构用来描述 当时系统的状态。 其中 Available 这个数组 给出的是每一类资源的资源数量，也就是系统中可以分配的 资源。 而 Max 这个二维 数组给出的

是每一个进程对 某一类资源的最大的需求量 Allocation 这个二维数组给出的是当前 系统当中哪些进程得到了哪些资源，也就是分配给这个进程的资源的数量 还有一个 Need，Need 的话指的是 这个进程还需要多少 资源。 当然每个进程对每一类资源都还需要多少记录在 Need 这个二维矩阵里头 然后是 Request，也就是本次 这个进程对资源的一次申请是 多少。 所以这些数据结构就给出了系统当前的一个状态 由于这些数据结构呢大部分是个二维数组 那么在后续的描述当中呢为了 简单地描述，我们给出了几个符号 Available 就表示系统可用的资源数量 那么 Max[i] 就表示进程 i 对 资源的最大需求量。 Allocation[i] 就表示进程 i 目前得到的资源数量 Need[i] 就是这个进程还需要多少 当然我们知道，还需要多少实际上是通过这两个 可以得到，推导出这个 Need 还需要多少 Request[i] 就表示这次，因为进程对资源的申请是逐步进行的，所以这一次申请它需要多少 我们简记这样几个符号 那么下面我们来给出银行家算法的具体步骤 当某一个进程 Pi 它 提出资源申请的时候，系统会完成以下几个步骤 第一步，要判断一下你的这次申请 和你还需要多少这个数量之间是不是有冲突 当然了，如果正确那么就进入了下一步，否则的话报错返回 然后我们再来判断一下 你这次的资源申请，系统当前剩余的资源数量是不是能满足你 如果你申请 10 个资源，系统现在 目前只有 8 个资源了，那么实际上也不会 分配，因为你要的系统暂时不能满足 因此呢也，这个进程 由于资源得不到满足所以它进入等待状态 如果能够满足那么接着进行下一步 这一步呢是做一个假设 因为你要的资源系统有，那么我们就假设这个资源分配给你了 分配给你以后所有的数据结构进行相应的调整，就得到了这样一个结果 Available 等于 把资源分配出去之后，剩余的再把它写回到 Available Allocation 呢就等于原来的占有的资源再加上这次分给你的 然后相应的 Need 这个数组也进行相应的调整 因此，这经过调整后的数据结构 就表示出了一个系统的新的状态，就假设分配给你，那么 系统进入了一个新的状态。 下面我们就要对这个新的状态进行判断 假如新的状态是安全状态，那么这个分配就完成了 如果分配这个状态，新的状态是一个不安全状态 也就是此次分配无效。 因此所有的状态都恢复到 分配之前的状态，而系统呢 让这个进程进入等待状态。 下面呢我们来 介绍一下如何判断当前的系统状态是安全状态 这就是安全性检查 那么为了进行安全性检查，我们定义了相关的数据结构，一个是 Work，一个是 Finish 我们来看一下这个检查的 步骤，首先要进行数据结构的初始化，

Work 里头实际上就是存放了当前可用资源的数量。 而 Finish 这个数组呢，我们初始化为，全部为 false。 然后我们就对，这两个数据结构进行相应地查找。找到满足条件的 i，也就是某个进程。 什么条件呢，第一个条件是 Finish[i] == false；也就是，这个进程 它还没有被检查过，第二个条件就是说， 检查一下这个进程，它还需要的资源数量， 是不是可以满足，那么可以满足呢，就是 Work 这个数组里的记录了当前可用的资源数量，如果 Need 小于 Work 说明，啊，这个资源可以满足 那么我们对这个数据结构不断地进行查找，找到了 i，那么我们进入下一步，如果我们整个遍历所有的这个数据结构， 找不到，啊，这样的，啊，进程 i 了，那么 也就是检查结束，要转到结束处理的步骤。 我们假设找到了，这样的一个满足条件的 i，也就是进程 现在我们来看一下，找到了之后，我们要做的工作，就是 我们把这个进程，它占有的资源都 假设它还给系统了，得到了一个新的可用资源的 数量，然后，这个时候，我把这个进程 标记成处理过了，然后继续找下一个进程。 就是找了这个进程 i，那么我们对它进行相应的处理，假设，啊，资源 分给你了，然后呢，你也被处理过了，如果 对于这样一个数组的话，反复查找，啊，最后，所有的进程 啊，都找过了，啊，找不到满足这个条件了，那当然找不到满足条件就是说， 可能这个进程，它的这个 Finish 数组，全都是 true 了，都处理过了。 或者是它的这个不能满足， 啊，刚才我们说的这个，你所需要的资源，啊。 等于、 小于等于当前系统剩余的资源。 那这种情况下呢，这个检查就结束，那么我们就进入到第四步。 第四步呢，就是检查究竟是哪种情况。 因为如果，啊，对所有的这个进程， 你的 Finish 这个数组全都变成 truth了，那就说明你全都被检查过一遍，而且是 找到了你，然后按照这个步骤，把你设置成 true，那么这个时候说明，系统中的所有进程， 其实都可以，啊，结束，因为资源都可以分给你， 然后你又还回去，啊，所以都可以结束，也就是安全 序列，就是这样一个过程，有一个安全序列， 那么系统处于安全状态。 如果 没有这个条件，就是说如果没有，不是 所有的，啊， Finish 这个元素都是 true， 那就是有的，啊，比如说不是 true 那就还是 false，没有、 没法处理了，因为它不满足我们的另外一个条件。 那也就是说，在我们这个安全序列当中，有一些进程不在里头。 换句话说，就没有一个所有进程的安全序列，那么我们认为系统处于不安全状态。 我们用一个小的例子来看一下， 假设说这是一个系统当前的一个状态，啊，我们可以看到， P1、 P2、 P3，目前已经

申请到的资源数量， 和它对资源的最大的需求的数量，以及现在、目前系统中还剩余的资源数量。 那么我们可以计算出来， 每个进程，它还需要多少资源， 那么根据这样一个结果，我们来看它的安全序列。 那么这里头，我们知道，系统中还剩下两个资源。 因此这两个资源，我们 直观地看上去，就只能分配给进程 P2，因为 如果分配给 P1，那么 P1， 拿到两个资源还 需要一个，系统中已经没了，或者是分配给 P3，也是一样 因此，在这里头的安全序列就是 P2 然后 P1 或 P3， 那也就是 P2、 P1、 P3， P2、 P3、 P1 是两个安全序列。 我们其实说过，只要有一个安全序列，那么这个系统就是安全状态，所以呢 像这样一个场景所描述的系统状态，就是一个安全状态。