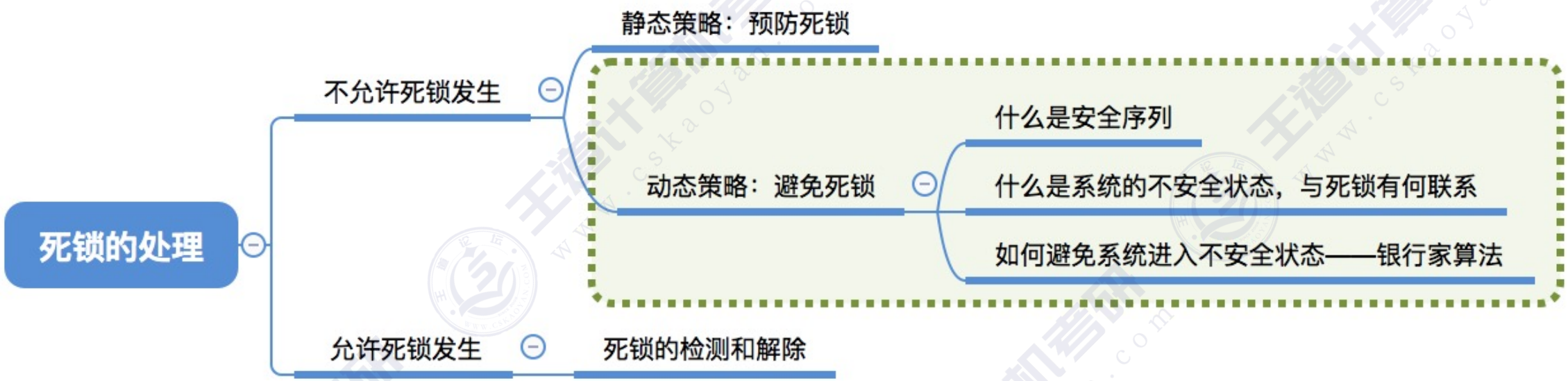
本节内容

死锁的处理策略
——避免死锁

知识总览



什么是安全序列

你是一位成功的银行家，手里掌握着100个亿的资金...
有三个企业想找你贷款，分别是 企业B、企业A、企业T，为描述方便，简称BAT。
B 表示：“大哥，我最多会跟你借70亿...”
A 表示：“大哥，我最多会跟你借40亿...”
T 表示：“大哥，我最多会跟你借50亿...”
然而...江湖中有个不成文的规矩：如果你借给企业的钱总数达不到企业提出的最大要求，那么不管你之前给企业借了多少钱，那些钱都拿不回来了...
刚开始，BAT三个企业分别从你这儿借了 20、10、30 亿 ...

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10	30
T	50	30	20

手里还有：40亿

此时... B 还想借 30 亿，你敢借吗？假如答应了B的请求.....

	最大需求	已借走	最多还会借
B	70	20+30=50	50-30=20
A	40	10	30
T	50	30	20

手里还有：10亿

只剩下10亿，如果BAT都提出再借20亿的请求，那么任何一个企业的需求都得不到满足...

什么是安全序列

你是一位成功的银行家，手里掌握着100个亿的资金…
有三个企业想找你贷款，分别是 企业B、企业A、企业T，为描述方便，简称BAT。
B 表示：“大哥，我最多会跟你借70亿…”
A 表示：“大哥，我最多会跟你借40亿…”
T 表示：“大哥，我最多会跟你借50亿…”
然而…江湖中有个不成文的规矩：如果你借给企业的钱总数达不到企业提出的最大要求，那么不管你之前给企业借了多少钱，那些钱都拿不回来了…
刚开始，BAT三个企业分别从你这儿借了 20、10、30 亿 …

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10	30
T	50	30	20

手里还有：40亿

此时… B 还想借 30 亿，你敢借吗？
假如答应了B的请求……

	最大需求	已借走	最多还会借
B	70	20+30=50	50-30=20
A	40	10	30
T	50	30	20

手里还有：10亿

经过三百六十度无死角检查，
给B借30亿是不安全的…

什么是安全序列

你是一位成功的银行家，手里掌握着100个亿的资金…
有三个企业想找你贷款，分别是 企业B、企业A、企业T，为描述方便，简称BAT。
B 表示：“大哥，我最多会跟你借70亿…”
A 表示：“大哥，我最多会跟你借40亿…”
T 表示：“大哥，我最多会跟你借50亿…”
然而…江湖中有个不成文的规矩：如果你借给企业的钱总数达不到企业提出的最大要求，那么不管你之前给企业借了多少钱，那些钱都拿不回来了…
刚开始，BAT三个企业分别从你这儿借了 20、10、30 亿 …

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10	30
T	50	30	20

手里还有：40亿

此时… A 还想借 20 亿，你敢借吗？
假如答应了A的请求……
之后按T→B→A的顺序借钱是OK的

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10+20=30	30-20=10
T	50	30	20

手里还有：20亿

可以先把20亿全部借给T，等T把钱全部还回来了，手里就会有20+30=50亿，再把这些钱全借给B，B还钱后总共有50+20=70亿，最后再借给A

什么是安全序列

你是一位成功的银行家，手里掌握着100个亿的资金...
有三个企业想找你贷款，分别是 企业B、企业A、企业T，为描述方便，简称BAT。
B 表示：“大哥，我最多会跟你借70亿...”
A 表示：“大哥，我最多会跟你借40亿...”
T 表示：“大哥，我最多会跟你借50亿...”
然而...江湖中有个不成文的规矩：如果你借给企业的钱总数达不到企业提出的最大要求，那么不管你之前给企业借了多少钱，那些钱都拿不回来了...
刚开始，BAT三个企业分别从你这儿借了 20、10、30 亿 ...

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10	30
T	50	30	20

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10+20=30	30-20=10
T	50	30	20

手里还有：40亿

此时... A 还想借 20 亿，你敢借吗？
假如答应了A的请求.....
之后按T→B→A的顺序借钱是OK的
按A→T→B的顺序借钱也是OK的

手里还有：20亿

或者，先借给A 10亿，等A还钱了手里就有 20+30 = 50 亿，再给 T 20亿，等T还钱了就有 50+30 = 80 亿，最后再给 B 借...

什么是安全序列

你是一位成功的银行家，手里掌握着100个亿的资金...
有三个企业想找你贷款，分别是 企业B、企业A、企业T，为描述方便，简称BAT。
B 表示：“大哥，我最多会跟你借70亿...”
A 表示：“大哥，我最多会跟你借40亿...”
T 表示：“大哥，我最多会跟你借50亿...”
然而...江湖中有个不成文的规矩：如果你借给企业的钱总数达不到企业提出的最大要求，那么不管你之前给企业借了多少钱，那些钱都拿不回来了...
刚开始，BAT三个企业分别从你这儿借了 20、10、30 亿 ...

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10	30
T	50	30	20

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10+20=30	30-20=10
T	50	30	20

手里还有：40亿

此时... A 还想借 20 亿，你敢借吗？
假如答应了A的请求.....
之后按T→B→A的顺序借钱是OK的
按A→T→B的顺序借钱也是OK的

经过三百六十度无死角检查，给A借 20 亿是安全的...
因为按照 T→B→A，A→T→B 的顺序给他们借钱是可行的...

安全序列、不安全状态、死锁的联系

	最大需求	已借走	最多还会借
B	70	20+30=60	50-30=20
A	40	10	30
T	50	30	20

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10+20=30	30-20=10
T	50	30	20

给B借30亿是不安全的...之后手里只剩10亿，如果BAT都提出再借20亿的请求，那么任何一个企业的需求都得不到满足...

所谓安全序列，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。

如果系统处于安全状态，就一定不会发生死锁。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。

安全序列、不安全状态、死锁的联系

	最大需求	已借走	最多还会借
B	70	20+30=60	50-30=20
A	40	10	30
T	50	30	20

给B借30亿是不安全的...之后手里只剩10亿，如果BAT都提出再借20亿的请求，那么任何一个企业的需求都得不到满足...

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10+20=30	30-20=10
T	50	30	20

给A借 20 亿是安全的，因为存在 T→B→A 这样的*安全序列。*

所谓安全序列，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。

比如A 先归还了10亿，那么就有安全序列T→B →A

如果系统处于安全状态，就一定不会发生死锁。如果系统进入不安全状态（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。

银行家算法

银行家算法是荷兰学者 Dijkstra 为银行系统设计的，以确保银行在发放现金贷款时，不会发生不能满足所有客户需要的情况。后来该算法被用在操作系统中，用于避免死锁。

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。

思考：BAT 的例子中，只有一种类型的资源——钱，但是在计算机系统中会有多种多样的资源，应该怎么把算法拓展为多种资源的情况呢？

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：

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10	30
T	50	30	20

进程	最大需求		已分配
P0	(7, 5, 3)		(0, 1, 0)
P1	(3, 2, 2)		(2, 0, 0)
P2	(9, 0 ,2)	减	(3, 0, 2)
P3	(2, 2, 2)		(2, 1, 1)
P4	(4, 3, 3)		(0, 0, 2)

此时总共已分配 (7, 2, 5)，还剩余 (3, 3, 2)
可把最大需求、已分配的数据看作矩阵，两矩阵相减，就可算出各进程最多还需要多少资源了

银行家算法

进程	最大需求	已分配	最多还需要	
P0	(7, 5, 3)	(0, 1, 0)	(7, 4, 3)	> (3, 3, 2)
				< (3, 3, 2)
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)	> (5, 3, 2)
				< (5, 3, 2)
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)	

资源总数 (10, 5, 7)，剩余可用资源 (7, 4, 3)

说明如果优先把资源分配给P1，那P1一定是可以顺利执行结束的。等P1结束了就会归还资源。于是，资源数就可以增加到
(2, 0, 0) + (3, 3, 2) = (5, 3, 2)

此时系统是否处于安全状态？
思路：尝试找出一个安全序列… {P1} P3} P0，P2，P4}
依次检查剩余可用资源 (3, 3, 2) 是否能满足各进程的需求
可满足P1需求，将 P1 加入安全序列，并更新剩余可用资源值为 (5, 3, 2)
依次检查剩余可用资源 (5, 3, 2) 是否能满足剩余进程（不包括已加入安全序列的进程）的需求
可满足P3需求，将 P3 加入安全序列，并更新剩余可用资源值为 (7, 4, 3)
依次检查剩余可用资源 (7, 4, 3) 是否能满足剩余进程（不包括已加入安全序列的进程）的需求……
……
以此类推，共五次循环检查即可将5个进程都加入安全序列中，最终可得一个安全序列。该算法称为安全性算法。可以很方便地用代码实现以上流程，每一轮检查都从编号较小的进程开始检查。
实际做题时可以更快速的得到安全序列。

银行家算法

进程	最大需求	已分配	最多还需要	
P0	(7, 5, 3)	(0, 1, 0)	(7, 4, 3)	> (3, 3, 2)
				< (3, 3, 2)
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)	> (5, 3, 2)
				< (5, 3, 2)
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)	

资源总数 (10, 5, 7)，剩余可用资源 (7, 4, 3)

说明如果优先把资源分配给P1，那P1一定是可以顺利执行结束的。

说明如果优先把资源分配给P3，那P3一定是可以顺利执行结束的。等P3结束了就会归还资源。于是，资源数就可以增加到
(2, 1, 1) + (5, 3, 2) = (7, 4, 3)

此时系统是否处于安全状态？
思路：尝试找出一个安全序列… {P1} P3} P0，P2，P4}
依次检查剩余可用资源 (3, 3, 2) 是否能满足各进程的需求
可满足P1需求，将 P1 加入安全序列，并更新剩余可用资源值为 (5, 3, 2)
依次检查剩余可用资源 (5, 3, 2) 是否能满足剩余进程（不包括已加入安全序列的进程）的需求
可满足P3需求，将 P3 加入安全序列，并更新剩余可用资源值为 (7, 4, 3)
依次检查剩余可用资源 (7, 4, 3) 是否能满足剩余进程（不包括已加入安全序列的进程）的需求……
……
以此类推，共五次循环检查即可将5个进程都加入安全序列中，最终可得一个安全序列。该算法称为安全性算法。可以很方便地用代码实现以上流程，每一轮检查都从编号较小的进程开始检查。
实际做题时可以更快速的得到安全序列。

银行家算法

进程	最大需求	已分配	最多还需要
P0	(7, 5, 3)	(0, 1, 0)	(7, 4, 3)
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)

资源总数 (10, 5, 7)，剩余可用资源 (7, 4, 3)

实际做题（手算）时可用更快速的方法找到一个安全序列：
经对比发现，（3, 3, 2）可满足 P1、P3，说明无论如何，这两个进程的资源需求一定是可以依次被满足的，因此P1、P3 一定可以顺利的执行完，并归还资源。可把 P1、P3 先加入安全序列。
(2, 0, 0) + (2, 1, 1) + (3, 3, 2) = (7, 4, 3)
剩下的 P0、P2、P4 都可被满足。同理，这些进程都可以加入安全序列。

于是，5个进程全部加入安全序列，说明此时系统处于安全状态，暂不可能发生死锁。

银行家算法

进程	最大需求	已分配	最多还需要
P0	(8, 5, 3)	(0, 1, 0)	(8, 4, 3)
P2	(9, 5 ,2)	(3, 0, 2)	(6, 5, 0)
P4	(4, 3, 6)	(0, 0, 2)	(4, 3, 4)

资源总数 (10, 5, 7)，剩余可用资源 (7, 4, 3)

再看一个找不到安全序列的例子：

经对比发现，（3, 3, 2）可满足 P1、P3，说明无论如何，这两个进程的资源需求一定是可以依次被满足的，因此P1、P3 一定可以顺利的执行完，并归还资源。可把 P1、P3 先加入安全序列。

(2, 0, 0) + (2, 1, 1) + (3, 3, 2) = (7, 4, 3)

剩下的 P0 需要 (8, 4, 3)，P2 需要 (6, 5, 0)，P4 需要 (4, 3, 4)

任何一个进程都不能被完全满足

于是，无法找到任何一个安全序列，说明此时系统处于不安全状态，有可能发生死锁。

银行家算法

Available = (1, 2, 1)

$Request_0$ = (2, 1, 1)

假设系统中有 n 个进程，m 种资源

每个进程在运行前先声明对各种资源的最大需求数，则可用一个 n*m 的矩阵（可用二维数组实现）表示所有进程对各种资源的最大需求数。不妨称为**最大需求矩阵 Max**，Max[i, j]=K 表示进程 Pi 最多需要 K 个资源 Rj。同理，系统可以用一个 n*m 的**分配矩阵 Allocation** 表示对所有进程的资源分配情况。Max – Allocation = **Need 矩阵**，表示各进程最多还需要多少各类资源。

另外，还要用一个**长度为m的一维数组 Available** 表示当前系统中还有多少可用资源。

某进程Pi向系统申请资源，可用一个长度为m的一维数组 $Request_i$ 表示本次申请的各种资源量。

进程	最大需求	已分配	最多还需要
P0	(7, 5, 3)	(0, 1, 0)(2, 2, 1)	(7, 4, 3) (5, 3, 2)
P1	(3, 2, 2)	(2, 0, 0)	(1, 2, 2)
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)
P3	(2, 2, 2)	(2, 1, 1)	(0, 1, 1)
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)

Max 矩阵　Allocation 矩阵　Need 矩阵

可用银行家算法预判本次分配是否会导致系统进入不安全状态：

①如果 $Request_i[j] \le Need[i, j]$ $(0 \le j \le m)$便转向②；否则认为出错。

因为它所需要的资源数已超过它所宣布的最大值。

②如果 $Request_i[j] \le Available[j]$ $(0 \le j \le m)$，便转向③；否则表示尚无足够资源，Pi必须等待。

③系统试探着把资源分配给进程Pi，并修改相应的数据（并非真的分配，修改数值只是为了做预判）：

$Available = Available - Request_i$;

$Allocation[i, j] = Allocation[i, j] + Request_i[j]$;

$Need[i, j] = Need[i, j] – Request_i[j]$

④操作系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式分配；否则，恢复相应数据，让进程阻塞等待。

银行家算法

数据结构：
长度为 m 的一维数组 Available 表示还有多少可用资源
n*m 矩阵 Max 表示各进程对资源的最大需求数
n*m 矩阵 Allocation 表示已经给各进程分配了多少资源
Max – Allocation = Need 矩阵表示各进程最多还需要多少资源
用长度为 m 的一位数组 Request 表示进程此次申请的各种资源数

银行家算法步骤：
①检查此次申请是否超过了之前声明的最大需求数
②检查此时系统剩余的可用资源是否还能满足这次请求
③试探着分配，更改各数据结构
④用安全性算法检查此次分配是否会导致系统进入不安全状态

安全性算法步骤：
检查当前的剩余可用资源是否能满足某个进程的最大需求，如果可以，就把该进程加入安全序列，并把该进程持有的资源全部回收。
不断重复上述过程，看最终是否能让所有进程都加入安全序列。

知识回顾与重要考点

数据结构：
长度为 m 的一维数组 Available 表示还有多少可用资源
n*m 矩阵 Max 表示各进程对资源的最大需求数
n*m 矩阵 Allocation 表示已经给各进程分配了多少资源
Max – Allocation = Need 矩阵表示各进程最多还需要多少资源
用长度为 m 的一位数组 Request 表示进程此次申请的各种资源数

银行家算法步骤：
①检查此次申请是否超过了之前声明的最大需求数
②检查此时系统剩余的可用资源是否还能满足这次请求
③试探着分配，更改各数据结构
④用安全性算法检查此次分配是否会导致系统进入不安全状态

安全性算法步骤：
检查当前的剩余可用资源是否能满足某个进程的最大需求，如果可以，就把该进程加入安全序列，并把该进程持有的资源全部回收。
不断重复上述过程，看最终是否能让所有进程都加入安全序列。

系统处于不安全状态未必死锁，但死锁时一定处于不安全状态。系统处于安全状态一定不会死锁。