INTRO-
DUCTION
TO
TRANS-
ACTION

はじめての人のための

# トランザクション入門

日本PostgreSQLユーザ会
第35回 PostgreSQL 勉強会
2017年5月27日

坂田 哲夫（NTT OSSセンタ）

# この講演について

- トランザクションという言葉を初めて聞く人にも分かるように、イチから説明します
- トランザクションの基本を説明します
  - アプリケーション・プログラムの開発の観点から、重要なポイントを選びました
  - 「標準SQL」を基準に、PostgreSQLについて説明します
  - OSS-DB Silver の出題範囲をほぼカバーします
  - 細かい部分は省略しています。最後に紹介する参考文献などで補ってください

# もくじ

INTRO-
DUCTION
TO
TRANS-
ACTION

# トランザクションの必要性と役割

# データベースの必要性と役割

- 同時実行制御の必要性
- トランザクションがなかったら
- 典型的な同時実行の異常
- トランザクションによる異常の防止

# データベースの働きとトランザクション

- 同時実行制御の必要性
  - 効率化のため、同時に複数の問い合わせを処理したい
  - その際に、異常が生じないように制御する
- トランザクションは同時実行制御の単位
  - AP（アプリケーション・プログラム）が実行する複数の処理を「トランザクション」にまとめて、同時実行制御の単位とする

# トランザクションがなかったら

- トランザクションなしでSQL文だけでAPを構成したらどうなるかを考えてみよう
- 同時に処理を実行すると、異常な結果が生じることがある
  - 同時実行の異常の典型的なパターンを紹介

# トランザクションがなかったら～同時実行の異常

- 完了しない状態のデータが残ってしまう*1
- 一貫性がないデータを読めてしまう*2
- 不正な更新データが書き込まれてしまう*3

＊1：同時実行の異常ではありませんが、重要なので
　　ここで紹介します
＊2：dirty read (汚れのある読み出し)と呼びます
＊3：lost update / loss of update (更新の消失)と
　　呼びます

※　：ここに挙げたもの以外に、non-repeatable read
　　(非再現読み出し)があります

# 例題について

- 銀行の例で考える
- 口座同士の送金の例は次の通り
  - 口座Aから1万円引き出す
  - 口座Bへ1万円振り込む

1. 口座Aの残高を読み出し
2. 引きだし後の残高を計算し
3. その残高を口座Aに書き込む

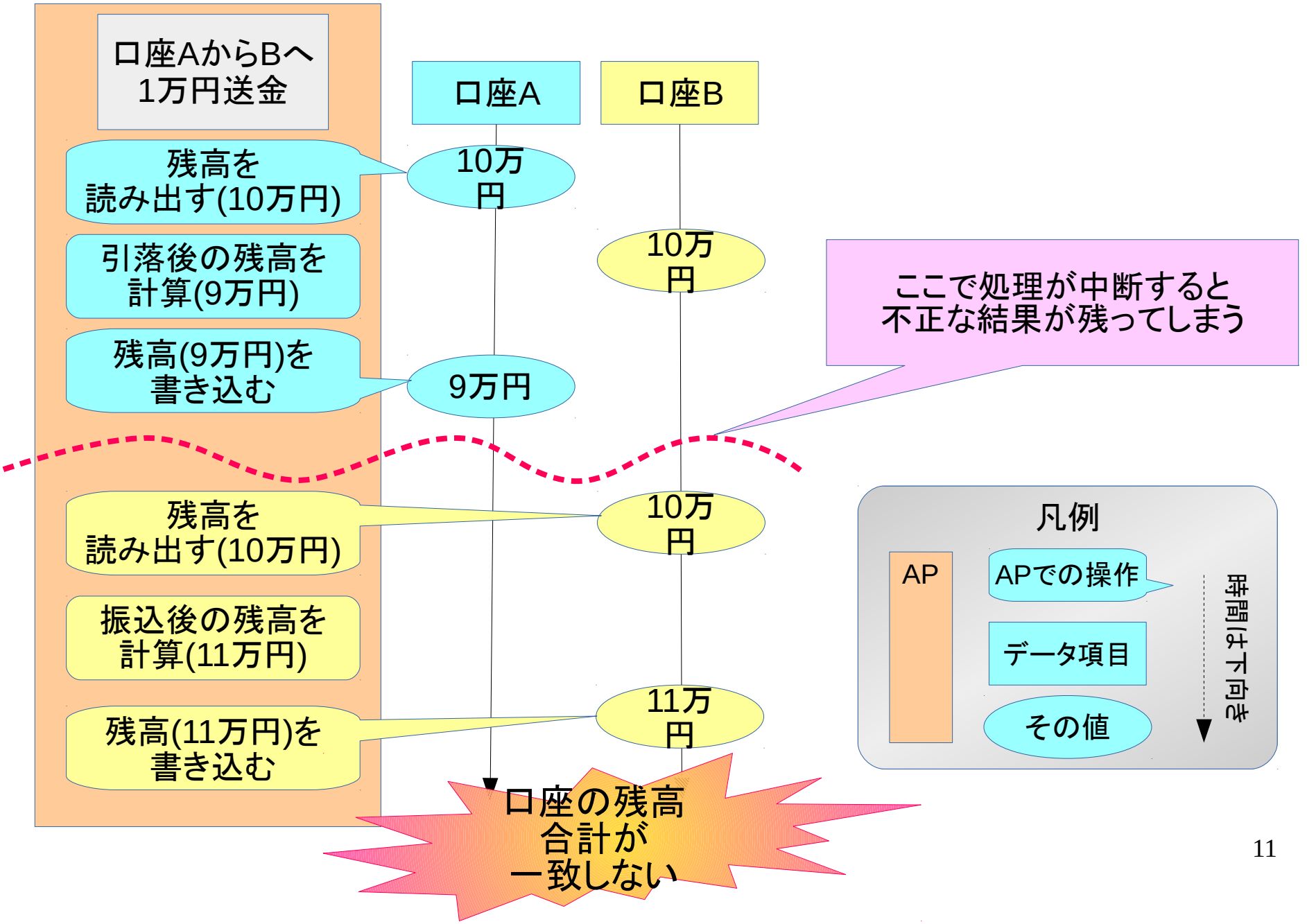
1. 口座Bの残高を読み出し
2. 振込後の残高を計算し
3. その残高を口座Bに書き込む

- ここでの「一貫性」はつじつまが合っていること
  - お金が勝手に消えたり、どこかから出てきたりしない
  - 2つの口座の間での送金の場合、送金の前後でそれら口座の残高は等しい

# 異常1：完了しない状態のデータが残ってしまう

- 口座AからBへ1万円送金
  1. 口座Aから1万円引き出す
  2. 口座Bへ1万円振り込む
- 1のSQL完了直後にAPが停止すると、口座Aから1万円を引き出したが、口座Bに振り込まれていない状態になる
  →1万円が消えてしまう

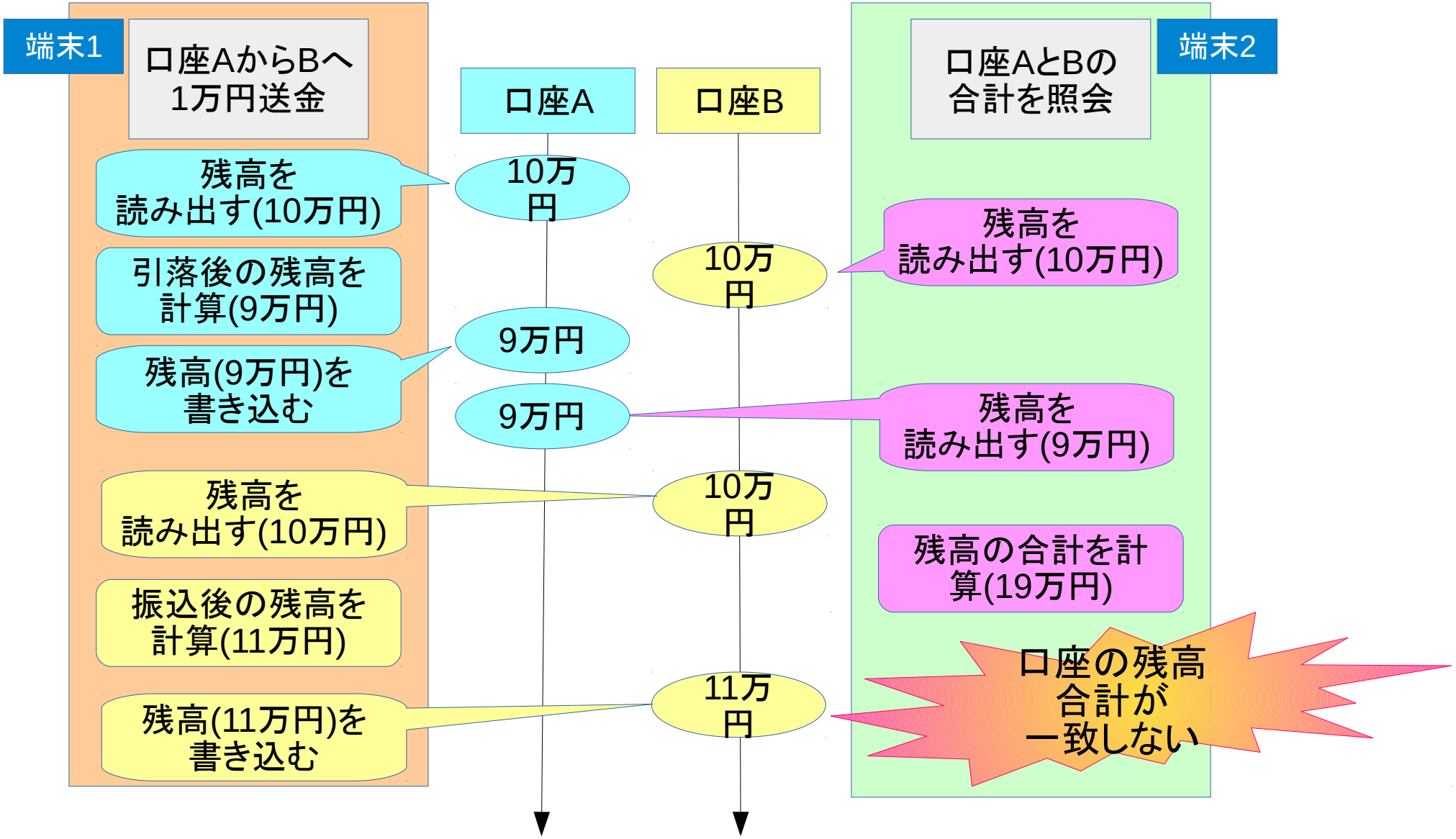
# 異常1：完了しない状態のデータが残ってしまう



# 異常２：一貫性がないデータが読めてしまう

- 口座AからBへ1万円送金
  1. 口座Aから1万円引き出す
  2. 口座Bへ1万円振り込む
- 1のSQL完了後・2のSQLの完了前に、別のAPで口座AとBの残高合計を検索すると、あるべき金額に対して1万円少ない額になる

# 異常2:一貫性がないデータが読めてしまう



# 「異常2」の実行例

- psqlを使って、次の操作を順に実行
  - 端末1で口座Aから1万円引き出す（前ページ左の列）
  - 端末2でaccount表の内容を見る（前ページ右の列）

端末1

```
test=> select * from account ;
 id | customer | balance
----+----------+---------
  1 | Mr.A     |  100000
  2 | Mr.B     |  100000
  3 | Mr.C     |  100000
(3 行)

test=> UPDATE account set balance =
90000 WHERE id = 1;
UPDATE 1
```

送金のために1万円引き出した

端末2

```
test=> select * from account ;
 id | customer | balance
----+----------+---------
  1 | Mr.A     |   90000
  2 | Mr.B     |  100000
  3 | Mr.C     |  100000
(3 行)
```

送金途中の状態が見えてしまう

# 異常3：不正な更新データが書き込まれてしまう

- 口座AからBへ1万円送金
  1. 口座Aから1万円引き出す
  2. 口座Bへ1万円振り込む
- 口座CからBへも1万円送金
  3. 口座Cから1万円引き出す
  4. 口座Bへ一万円振り込む

1. 口座Aの残高を読み出し
2. 引きだし後の残高を計算し
3. その残高を口座Aに書き込む

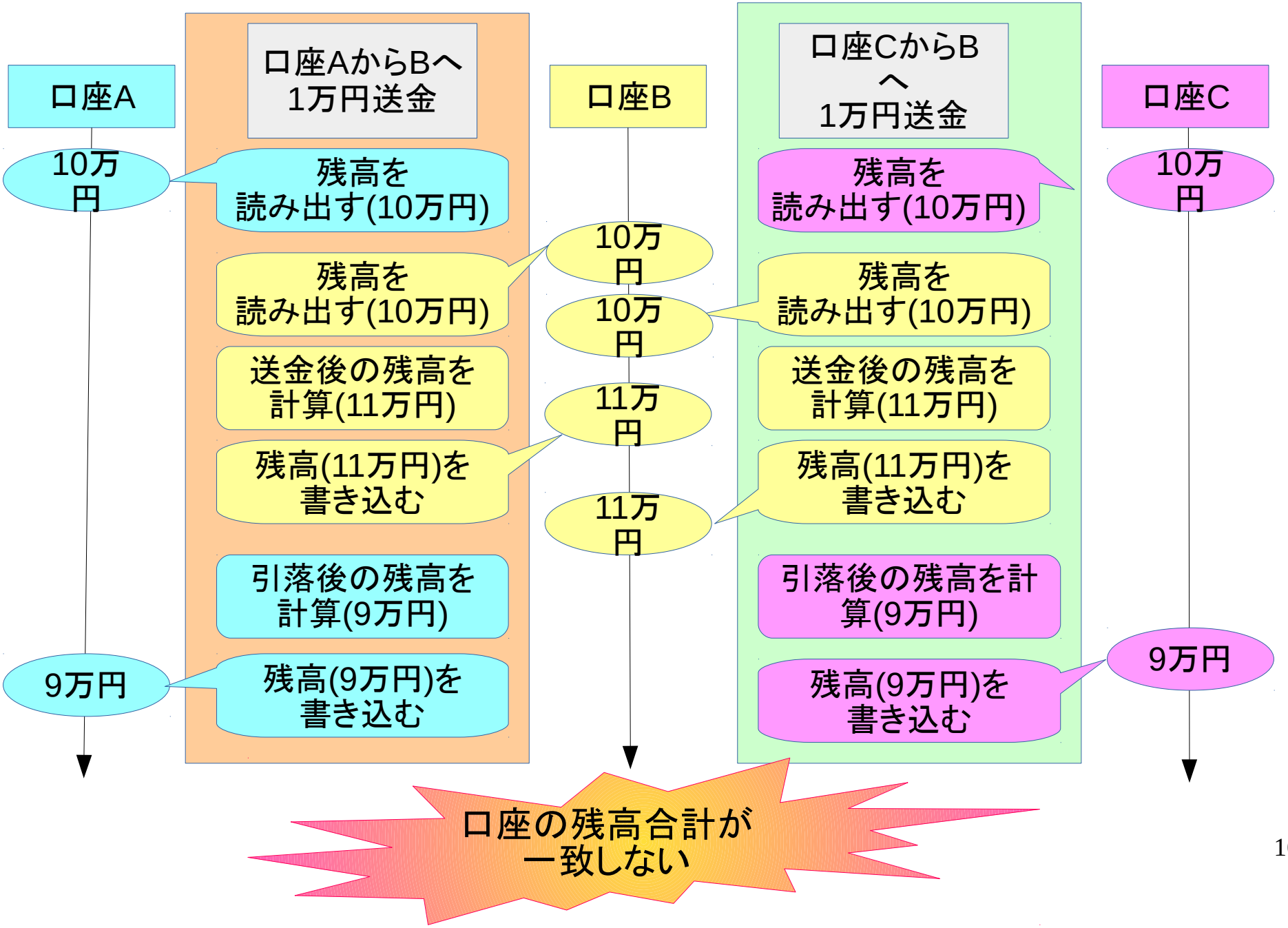
1. 口座Bの残高を読み出し
2. 振込後の残高を計算し
3. その残高を口座Bに書き込む

1. 口座Cの残高を読み出し
2. 引きだし後の残高を計算し
3. その残高を口座Cに書き込む

1. 口座Bの残高を読み出し
2. 振込後の残高を計算し
3. その残高を口座Bに書き込む

- 実行順序によっては、不正な状態のデータが書き込まれる

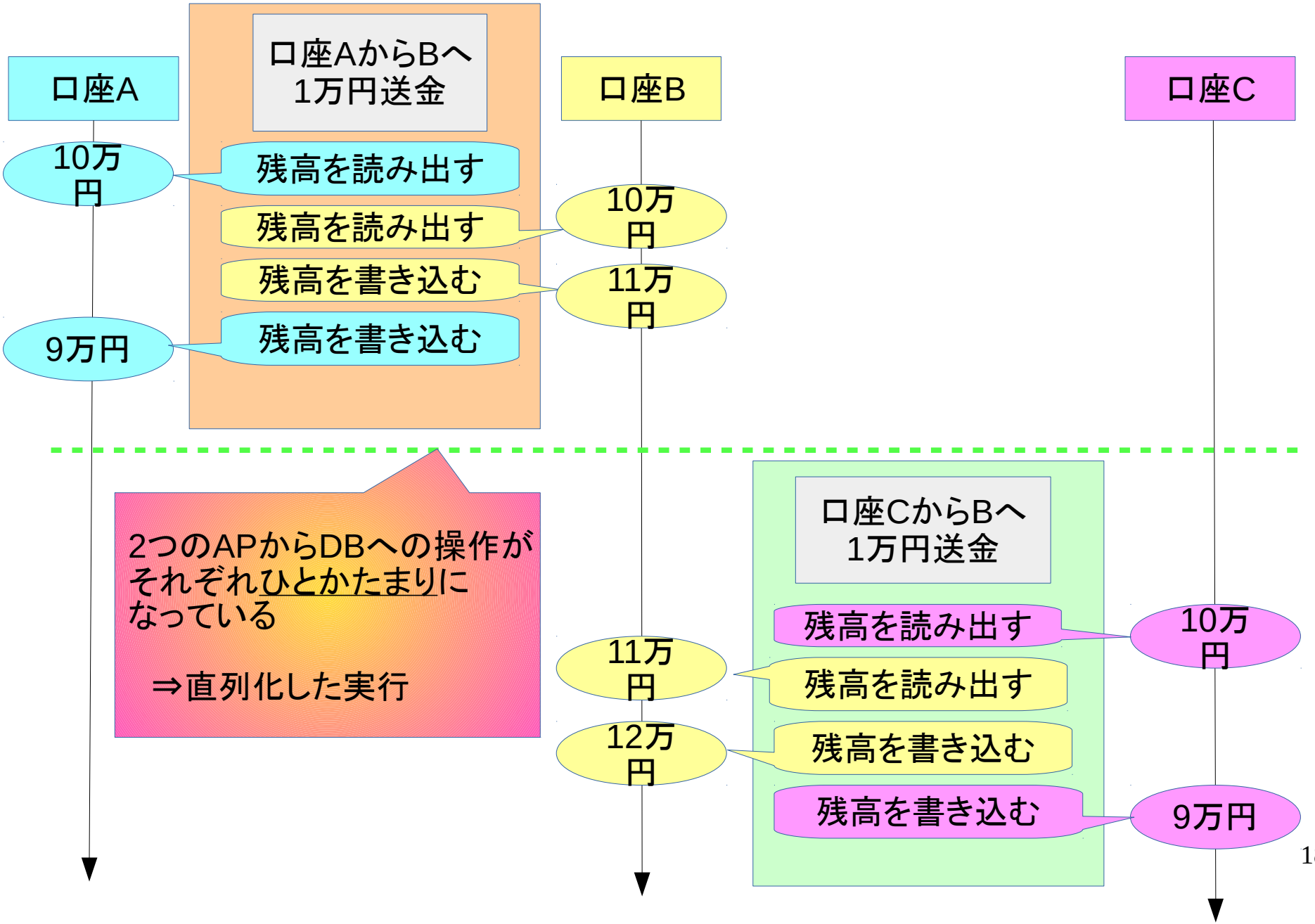
# 異常3:不正な更新データが書き込まれてしまう



# 同時実行の異常を防止する

- 異常1
  - 一連の処理が完了した上で、DBに記録を残すようにする
- 異常2と異常3
  - 各APの一連の処理を続けて処理する(処理と処理の間に、他のAPの処理が割りこまない)
- APの一連の処理をまとめて(中途半端で終わらずに・割り込まれずに)実行するというところがポイント

# 異常3の防止(イメージ図)



INTRO-
DUCTION
TO
TRANS-
ACTION

# トランザクションの使い方

# トランザクションの使い方

- APからみたトランザクション
- トランザクションの特性
- トランザクションの書き方
- トランザクションの例
- エラーとその対策ーSAVEPOINT

# APからみたトランザクション

- 一貫した処理を保証するための、データベース処理の集まり
- 具体的には、APで「ここからトランザクションが始まる」と宣言した位置から「ここでトランザクションが終わる」と宣言する位置までの間のSQL文の集まり
  - 終わりの宣言は「完了したので内容を確定せよ」(commit)と「完了せずに、内容を破棄せよ」(rollback)の2つがある
  - 開始の宣言は省略できない＊1
    (何も宣言せずにSQL文を実行すると、Transactionが始まり、そのSQL文の終了とともにcommit/rollbackする)

＊1：PostgreSQLの独自仕様でそうなっている
標準SQLでは、START TRANSACTIONなしで
トランザクションを開始できる

# トランザクションの特性

- 更新データは処理が完了した状態でのみ保存できる(A)
  - 完了前であれば更新データを実行前の状態に巻き戻す
- 一貫性のあるデータの読み書きを保証する(C)
  ⇒同時に1つのトランザクションだけが実行しているように見える(I)
- 書き込んだデータは確実に保存する(D)
- これらの性質をまとめてACID性と呼ぶ
  - Atomicity（原子性）
  - Consistency（一貫性）
  - Isolation（分離性　または　隔離性）
  - Durability（持続性 または 永続性＊1）

＊1：durability を「永続性」と訳す例があるが、persistence と区別がつかないので、良い訳でないと思う

# トランザクションの書き方

- 一貫性を持たせたい一連の処理を、BEGIN＊1またはSTART TRANSACTIONとCOMMITの間に書く
  - COMMITによってトランザクションでのDBへの更新が確定する
  - 途中で打ち切る場合は、COMMITに代えてROLLBACKする
    - DBへの更新は全て取り消される
  - BEGINまたはSTART TRANSACTIONなしでSQLを発行すると、1SQLが1トランザクションとなる＊2
    - 先に説明した異常が生じることがある

＊1：PostgreSQL独自のコマンド
＊2：PostgreSQL独自仕様による振舞い
　　標準SQLでは、START TRANSACTIONなしで
　　トランザクションを開始できる

# トランザクションの例

- 「異常2」を防ぐためにトランザクションを用いる

  ①端末1で口座Aから1万円引き出す
  （こちらはトランザクション内で実行する）

  ②端末2でaccount（口座）表の内容を見る

id：口座番号（キー）
Customer ：顧客名
Balance：残高

端末1

トランザクション開始

```
test=> begin;
BEGIN
test=> SELECT * FROM account ;
 id | customer | balance
----+----------+---------
  1 | Mr.A     |  100000
  2 | Mr.B     |  100000
  3 | Mr.C     |  100000
(3 行)

test=> UPDATE account set balance =
90000 WHERE id = 1;
UPDATE 1
```

① 送金のために1万円引き出した

端末2

```
test=> select * from account ;
 id | customer | balance
----+----------+---------
  1 | Mr.A     |  100000
  2 | Mr.B     |  100000
  3 | Mr.C     |  100000
(3 行)
```

② 送金途中の状態が見えない
（送金前の状態が見える）

# トランザクションの例（続き）

- トランザクションがコミットすると結果が見える
  - ③端末1で口座Bに1万円振り込む
  - ④コミットする
  - ⑤端末2でaccount表の内容を見る

端末1

```
test=> UPDATE account set balance =
110000 WHERE id = 2;
UPDATE 1
```

③ 口座Bへ振込

```
test=> COMMIT;
COMMIT
```

④ コミットして送金を完了

端末2

```
test=> select * from account order by
id;
 id | customer | balance
----+----------+---------
  1 | Mr.A     |   90000
  2 | Mr.B     |  110000
  3 | Mr.C     |  100000
(3 行)
```

⑤ 送金が完了した状態が見える

# トランザクション と エラー

- トランザクション内でSQL文がエラーとなった場合
  後続するSQL文は全て･無条件にエラーとなる
  commitを実行しても、実際には rollback される

この振舞いは、PostgreSQL特有のものです。
他のDBMSでは振る舞いが異なることがあります（参考文献[3]を見てください）

# トランザクション と エラー（実行例）

```
test=> begin;
BEGIN
test=> select * from account ;
 id | customer | balance
----+----------+---------
  1 | Mr.A     |  100000
  2 | Mr.B     |  100000
  3 | Mr.C     |  100000
(3 行)
```

トランザクションを開始して、
正常に動作することを確認する

SQLでエラーを起こした
（idを重複させた）

```
test=> INSERT INTO account(id, customer, balance) VALUES (3,
'Mr.D', 100000);
ERROR:  重複キーが一意性制約"account_id_key"に違反しています
DETAIL:  キー (id)=(3) はすでに存在します
```

```
test=> select * from account ;
ERROR:  現在のトランザクションがアボートしました。トランザクションブロックが終わ
るまでコマンドは無視されます
```

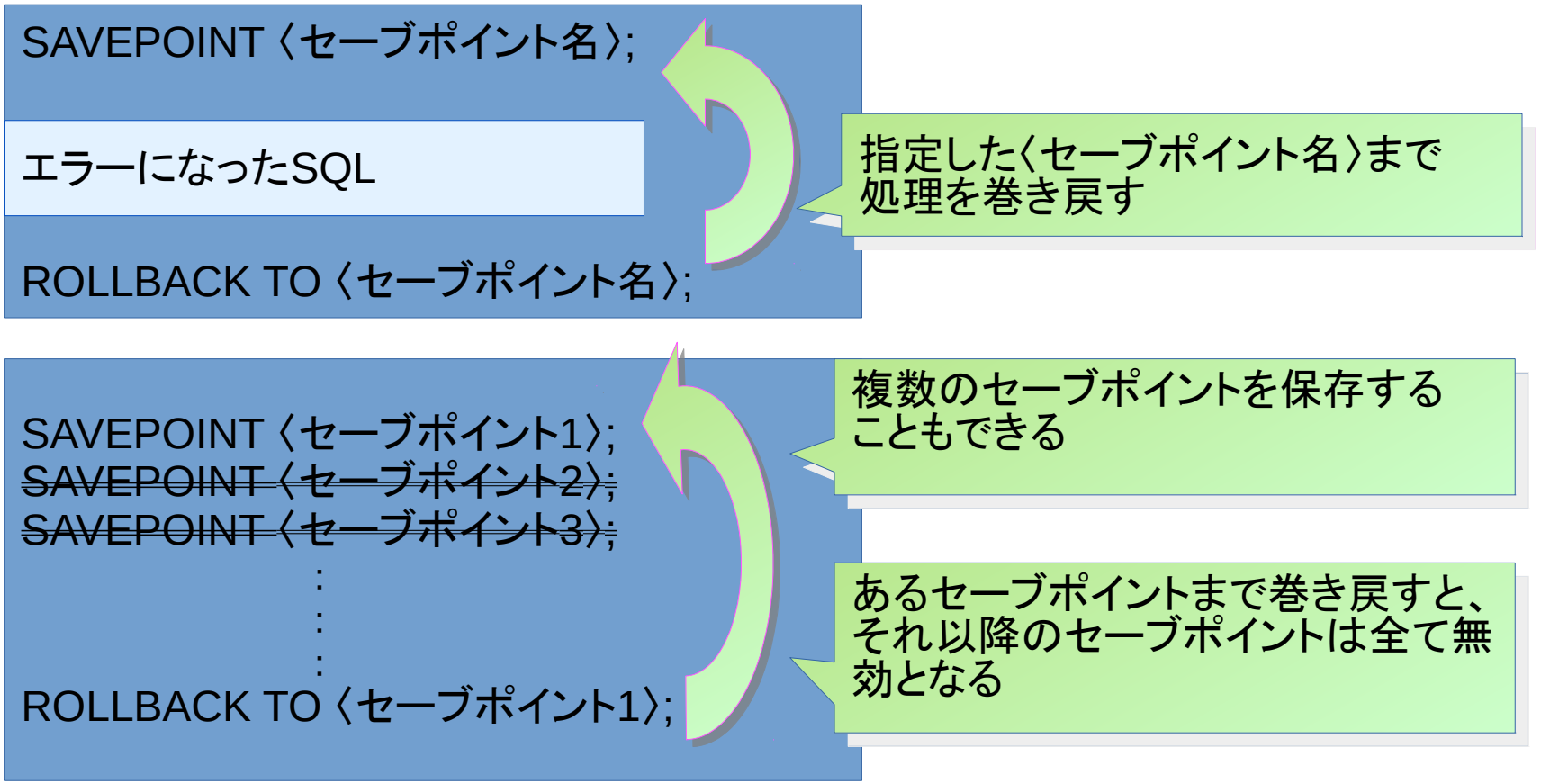
エラーするとTrXはアボートされる
エラー後のコマンドは無視される

```
test=> commit;
ROLLBACK
test=>
```

COMMITコマンドでも
ROLLBACKされる

# SAVEPOINT と 部分的ロールバック

- 部分的ロールバックによって、先に保存したセーブポイント名のところまで、「巻き戻す」ことができる
  - 巻き戻し後は、トランザクションが継続できる



# SAVEPOINT（実行例）

```
test=> begin;
BEGIN
test=> INSERT INTO account(id, customer, balance) VALUES (4,
'Mr.D', 100000);
INSERT 0 1
test=> SAVEPOINT SP1;
SAVEPOINT
```

顧客Dが追加できる

セーブポイント SP1 に保存

```
test=> INSERT INTO account(id, customer, balance) VALUES (4,
'Mr.D', 100000);
ERROR:  重複キーが一意性制約"account_id_key"に違反しています
DETAIL:  キー (id)=(4) はすでに存在します
```

SQLでエラーを起こした（idを重複させた）

```
test=> ROLLBACK TO SP1;
ROLLBACK
```

セーブポイント SP1 に巻き戻す

```
test=> select * from account ;
 id | customer | balance
----+----------+---------
  1 | Mr.A     |  100000
  2 | Mr.B     |  100000
  3 | Mr.C     |  100000
  4 | Mr.D     |  100000
(4 行)
```

再びトランザクションが実行できる

```
test=> commit;
COMMIT
```

コミットも正常にできる

INTRO-
DUCTION
TO
TRANS-
ACTION

# トランザクションの分離

# トランザクションの分離

- 分離の考え方の紹介
- 分離レベル
  - Read committed
  - Repeatable read
  - Serializable

# トランザクションの分離の考え方

- 分離性(ACIDのI)：複数のトランザクションを同時に実行しても、異常が生じないこと
  - 一度に1つずつ（逐次的に）トランザクションを実行した場合＊1には同時実行処理による異常は生じない
  - 「一度に1つずつ実行」した場合と等価な結果を生じるトランザクションの実行順序を Serializable（直列化可能）という＊2
  - 直列化可能な実行順序であればトランザクションは分離されている

＊１：このようなトランザクションの実行を「直列」あるいは「逐次」と呼びます
＊２：トランザクションの実行そのものが「直列化」されていなくても、
実行結果が何らかの直列化したトランザクション列と同じになれば良い。
「直列化可能」の詳しい定義は、参考文献2を見てください

# 分離レベル

- Serializable は最も厳密な分離を提供する
  - 反面、同時に実行されるトランザクションの数が制限される
- 分離の程度（レベル）を下げれば、多くのトランザクションを同時に実行することができる
  - 分離レベルを下げると、同時実行に伴う異常が生じることがある
  - PostgreSQLでは、以下の3つの分離レベルがある＊1
    Serializable (直列化可能)
    Repeatable read (反復可能読み取り)
    Read committed (読み取り一貫)

＊1：PostgreSQLのマニュアルでは、それぞれ
「シリアライザブル」「リピータブルリード」「リードコミッティド 」
と言います

# Read committed

- SQL文を開始する時点で、他のトランザクションが変更してコミットしたデータの参照を許す
  - 自トランザクションの開始後のコミットの分も含む
- 同じ行を2回読むと、違った結果を返すことがある
  - 「反復不能読み取り」と呼ぶ
- 読み込みトランザクションではファントム・リードの可能性がある
  - ファントム・リードの説明は次ページで
- 「直列化異常」が生じることがある
  - コミットは成功するが「直列化」した実行順序では生じえない結果が生じる

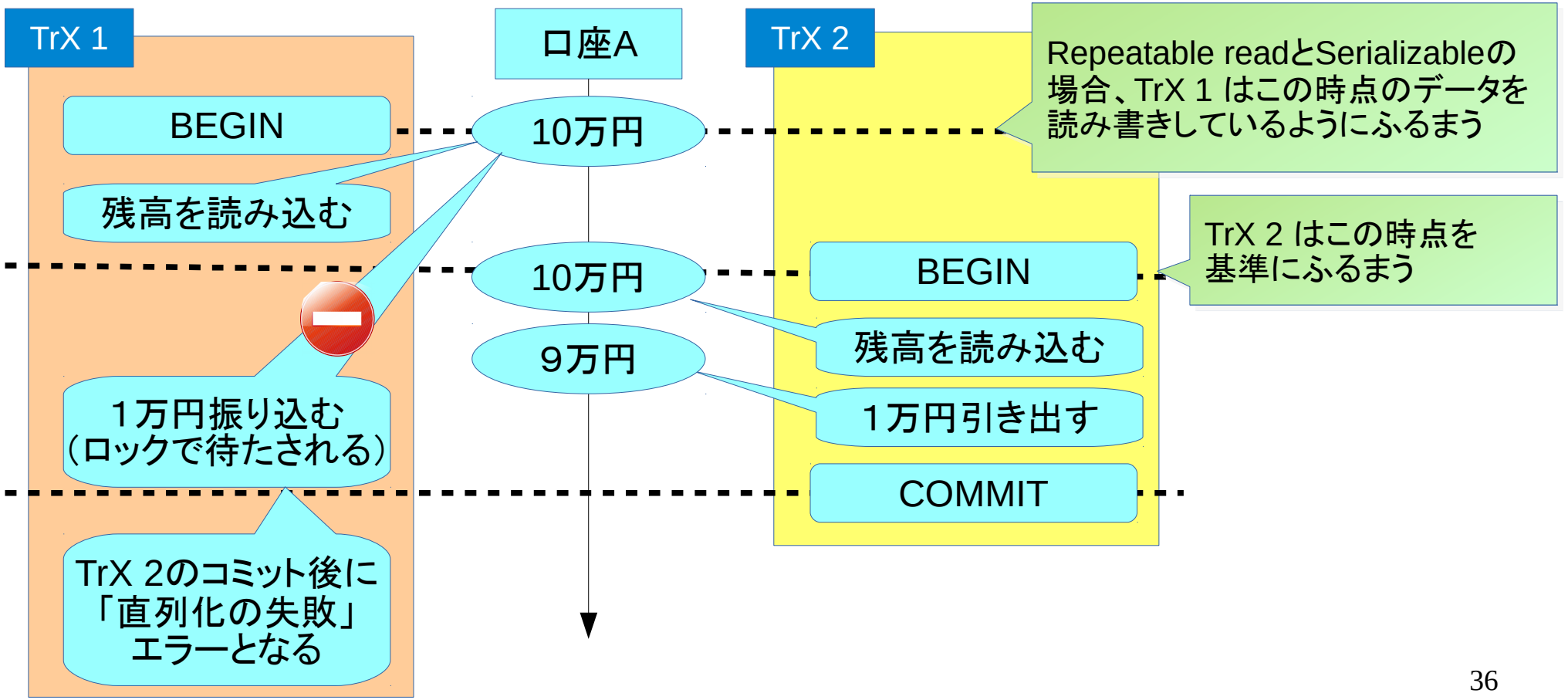
# Repeatable read

- 一度読み込んだ行の内容は、再度読み込んでも同じである
  - 他のトランザクションが更新してコミットした後に同じデータを読み込んでも値は不変
  - PostgreSQLの場合＊1、参照のみのトランザクションであれば、Serializableと同じ結果が保証される
  - 直列化の失敗（というエラー）が生じることがある

＊1：標準SQLでは…
- 「行の集合」としてみた場合、再度読み込むと1回目にはなかった行が読み込まれることがある（この現象をファントム リード (phantom read)と呼ぶ）
- PostgreSQLのRepeatable read ではファントム リードは生じない（標準SQLでは許容する）

# 直列化の失敗

- 同時実行しているトランザクションが更新したデータを更新すると、「直列化の失敗」というエラーとなる
  - 更新を含むトランザクションでのみ発生する



# Serializable

- 「直列化」した実行順序と等価な結果を保証する
  - 反復不能読み取り、ファントムリード、直列化異常のいずれも生じない
  - Repeatable readと同様に、直列化の失敗が起こることがある
  - 実行のためのコストが大きい

# 分離レベルの使い方

- トランザクションの開始時に宣言する
  - BEGINのあとで、以下のコマンドを実行

```
SET TRANSACTION ISOLATION LEVEL〈分離レベル〉...
```

SERIALIZABLE, REPEATABLE READ, READ COMMITTED のどれかが入る

  - BEGINを使わない場合は、以下のコマンドでトランザクションを開始する

```
START TRANSACTION ISOLATION LEVEL〈分離レベル〉...
```

上と同じ

  - 宣言しないとデフォルトの分離レベルになる

マニュアルの SET TRANSACTION のページを参照

# 分離レベルのまとめ

- PostgreSQLの分離レベルと同時実行処理の異常は下表の通り
- 分離レベルにはトレードオフがある
  - 下に行くほど「安全」になる（正しい結果が保証される）
  - 上に行くほど「効率」が高い（スループットが出せる）

| 分離レベル | 反復不能読み取り | ファントムリード | 直列化異常 |
|---|---|---|---|
| READ COMMITTED | 可能性あり | 可能性あり | 可能性あり |
| REPEATABLE READ | 生じない | 生じない | 可能性あり |
| SERIALIZABLE | 生じない | 生じない | 生じない |

INTRO-
DUCTION
TO
TRANS-
ACTION

# ロックとその利用

# ロックとその利用

- トランザクションの実現とロック
- ロックの種類ーレベルとモード
- ロックモード
- 表レベルロックと行レベルロック
- 明示的なロック

# トランザクションの実現とロック

- トランザクション を実現する際には、ロック(lock)という仕組みを使っている
  - 2人以上の人が同時にデータをアクセスできない仕組み
  - あるデータが使用中（ロックされている）なら、後からアクセスしたトランザクションは待たされる

ロックにはいろんなバリエーションがあります（上記は一例です）

- DBMSではロックは全て自動的に獲得・開放されるので、AP開発者は操作（プログラム）しなくて良い
  - PostgreSQLでも同様だが、APの都合によっては明示的にロックを獲得することもできる

トランザクションは内部的にロックを獲得します
不必要に長くならないように、注意しましょう

# ロックの種類：レベルとモード

- ロックのレベル
  - ロックの対象(object)をレベルと呼びます
  - 「表」レベル、「行」レベルが代表的＊1
    - 「ページ」レベルもある(が説明は略)
- ロックのモード
  - 同時にアクセスを許可する（ロックを獲得できる）か否か
    - ロックによって、2つ以上のトランザクションからのアクセスを無条件に禁止されると、処理性能が下がる
    - APの要件によっては「2つ以上のトランザクションからのアクセスを許可してよい」こともある

＊1：PostgreSQLのマニュアルでは
表レベル⇒テーブルレベル
行レベルはそのまま。英語は row level lock.

# ロックモードの基本：共有と専有

- 共有と専有―ロックモードの基本的考え方
  - 共有(share)：複数のトランザクションでアクセスできる。読み出しに使われる
  - 専有(exclusive)：1つのトランザクションだけがアクセスできる。書込みに使われる
- ロックモードの表(マトリクス)
  - 現在あるロックモードでロックが獲得されている時に、別のトランザクションがあるモードでロックを要求する時に、その要求が許可されるか否かを表で表す

共有・専有という2つのモードのロックの両立性

| 要求するロックモード(列) | 現在のロックモード(行) | |
|---|---|---|
| | share | exclusive |
| 共有(share) | ○ | × |
| 排他(exclusive) | × | × |

凡例

○：ロックが獲得できる
×：ロックが獲得できない

# 表レベルのロック

- PostgreSQLの主な表レベルロック
  - SQL文を実行する際に、所定のモードでロックを要求する

主なロックモードの両立性（全部で8モードある。マニュアル§13.3を参照）

| 要求するモード | ロックを必要とするコマンド | 現在のロックモード | | | |
|---|---|---|---|---|---|
| | | AS | RX | SUX | S |
| Access Share | 通常の表の読み出し | ○ | ○ | ○ | ○ |
| Row eXclusive | INSERT, DELETE, UPDATE | ○ | ○ | ○ | × |
| Share Update eXclusive | (FULLではない) VACUUM, ANALYZE | ○ | ○ | × | × |
| Share | CREATE INDEX | ○ | × | × | ○ |

この表を見ると、次のようなことも分かる
- 表の読み出しとINSERT, DELETE, UPDATEは同時に実行できる
- VACUUM中には表の読み書きは可能だが CREATE INDEX することはできない

# 行レベルのロック

- 同じ行に対する書き込みを制御する（参照時には要求しない）
- FOR UPDATE
  - SELECT～FOR UPDATEで読み出した行を「更新用に」ロック
- FOR NO KEY UPDATE＊1
  - 通常のUPDATE文で獲得
- FOR SHARE＊1
  - UPDATE文 DELETE 文をブロック
- FOR KEY SHARE＊1
  - FOR UPDATE をブロックする

＊1：これらのモードはPostgreSQL固有

| 要求するモード | 現在のロックモード | | | |
|---|---|---|---|---|
| | KS | S | NKU | U |
| FOR KEYS SHARE | ○ | ○ | ○ | × |
| FOR SHARE | ○ | ○ | × | × |
| FOR NO KEY UPDATE | ○ | × | × | × |
| FOR UPDATE | × | × | × | × |

凡例

○：ロックが獲得できる
×：ロックが獲得できない

# 明示的なロック

- PostgreSQLは、そのトランザクションに必要なもっとも制約の少ないモードでロックを獲得する
  - APの都合上、より制約が大きいロックの方が都合が良いことがある
- 特定の表・行を明示的に（コマンドを使って）ロックすることができる
  - ロックを獲得する際には、モードを指定する
- 獲得したロックは、Transaction完了時(Commit または Rollback)にまとめて開放される
  - Transaction途中で、個別に開放することはできない

# 明示的な表ロックの例

- Read Committed トランザクションの中で特定の表を反復して読み出し、一貫した結果を得たい
  - Read Commitedでは表の読み出し時に Access Shareでロック
  - 他のトランザクションで同じ表を更新できるので、繰り返し読み込みすると、一貫した参照結果が得られない（反復不能読み取り）
- 以下のコマンドで表ロックすると、反復読み取りで一貫した結果が得られる＊1

LOCK〈表の名前〉IN〈モード名〉MODE

＊1：LOCK文は PostgreSQL固有

この例では SHARE（参照のみ実行する場合）
　または
SHARE ROW EXCLUSIVE（参照・更新する場合）
を指定することで、他のトランザクションによる表の変更が抑止できる

INTRO-
DUCTION
TO
TRANS-
ACTION

# デッドロックとその対策

# デッドロックとその対策

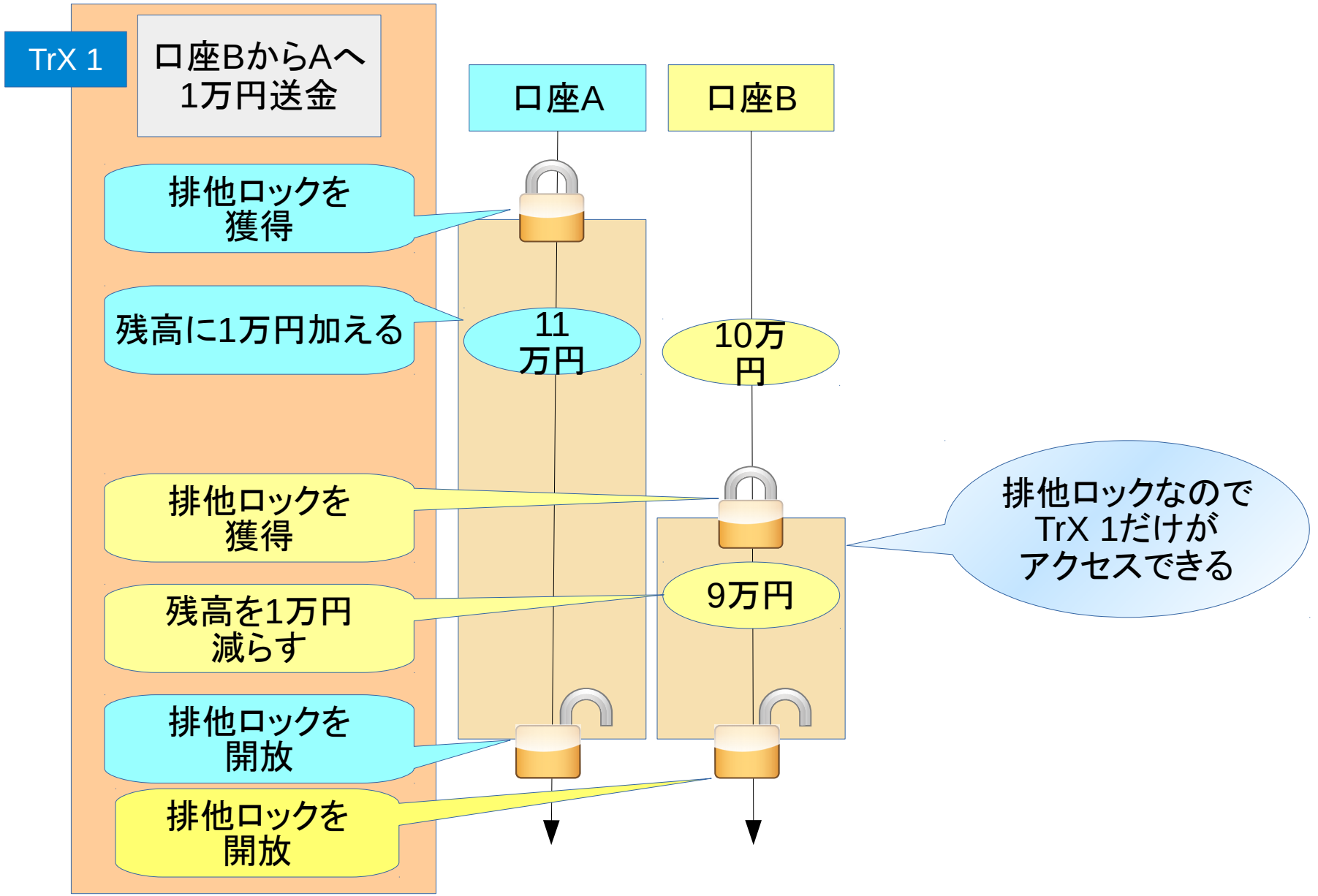
- デッドロックの説明と発生例
- デッドロックの原因
- デッドロックの対策
  - 予防とリトライ

# デッドロックとは？

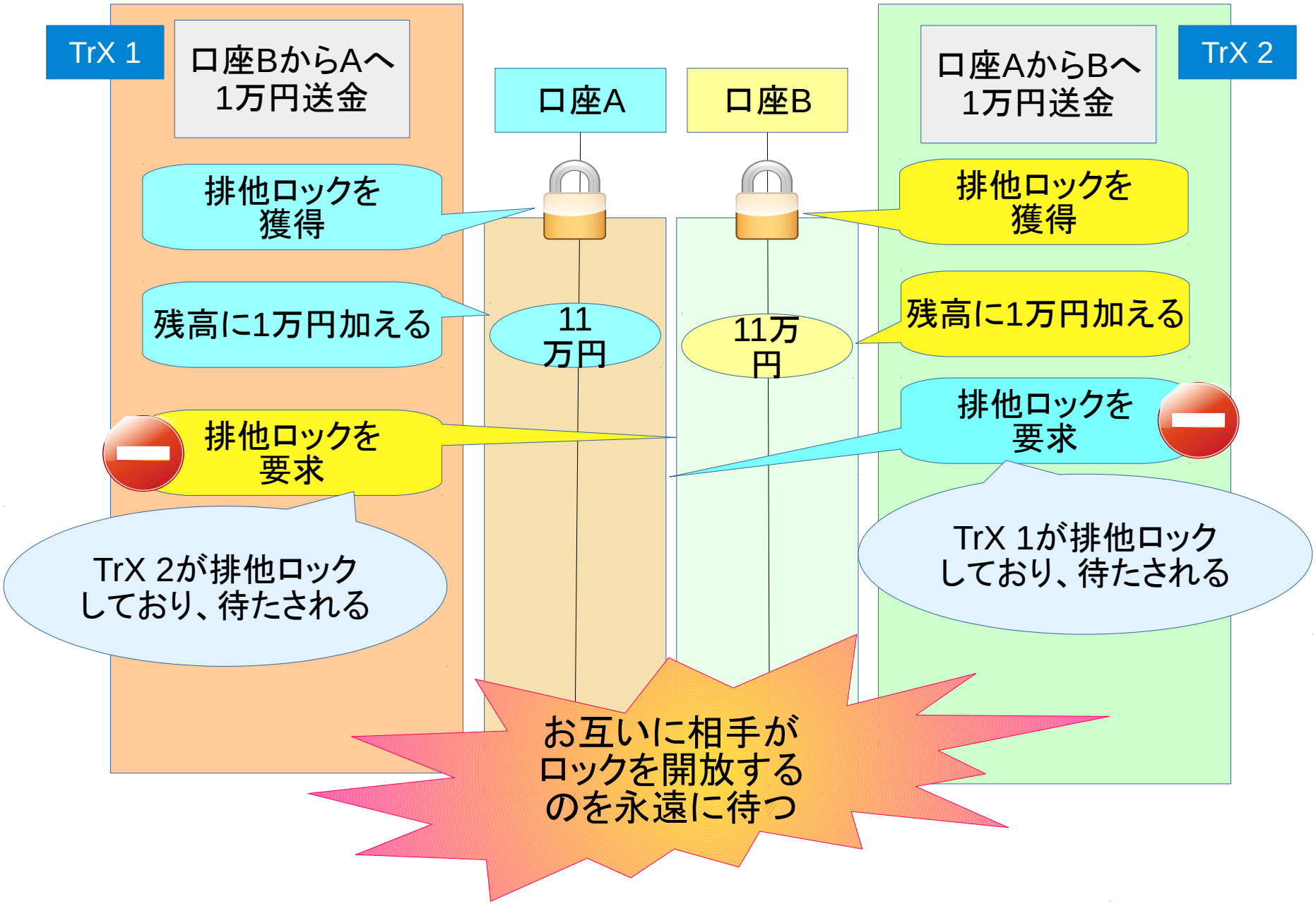
- ロックを使うことで、あるデータにアクセスするトランザクションが1つしかないことを保証する
  - 同じデータにアクセスする他のトランザクションは待たされる（ブロックされる）
  - ロックしているトランザクションが完了するまで、ロックは開放されない
- 2つのトランザクションが互いに「相手がロックしているデータにアクセスしようとする」と、相手を永久に待つことになる＊1
  ⇒デッドロック（dead lock）と呼ぶ

＊1：3つ以上のトランザクションの間でも、同様にデッドロックが発生することがあります
ここでは、2つのトランザクションの例で解説します

# デッドロックの例（単独実行の場合はうまくいく）

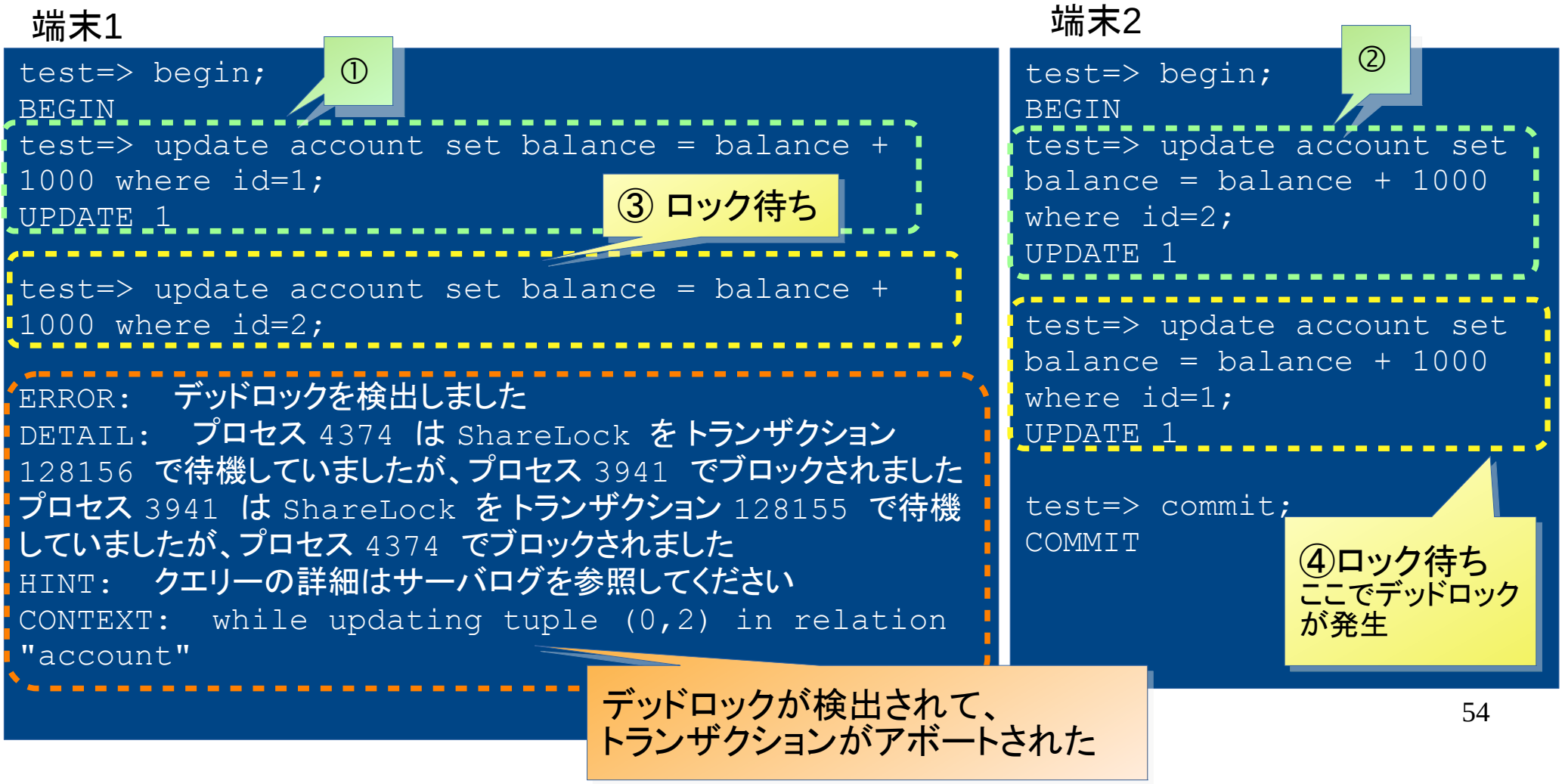


# デッドロックの例


TrX 1
口座BからAへ
1万円送金
口座A
口座B
口座AからBへ
1万円送金
TrX 2
排他ロックを
獲得
排他ロックを
獲得
残高に1万円加える
11
万円
11万
円
残高に1万円加える
排他ロックを
要求
排他ロックを
要求
TrX 2が排他ロック
しており、待たされる
TrX 1が排他ロック
しており、待たされる
お互いに相手が
ロックを開放する
のを永遠に待つ

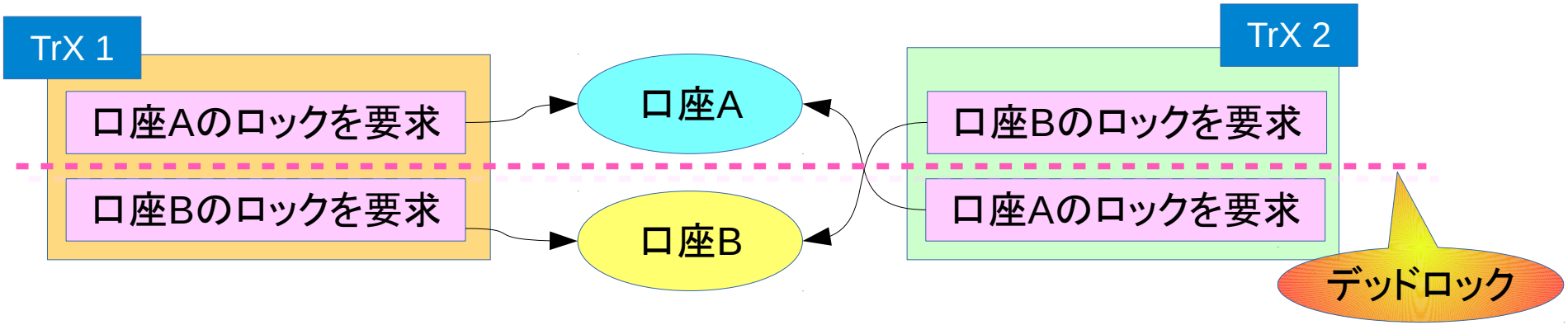
# デッドロックの発生例

- PostgreSQLはデッドロックを検出して、デッドロックしているトランザクションをどれか1つアボートする
  - 2つのトランザクション（端末）で①～④の順でSQLを実行→④を実行するとデッドロックが発生



# デッドロックの原因

- 複数のトランザクションが同時実行されるときに
  a）複数回に分けてロックを要求する
  b）それぞれのトランザクションで、互いに逆順にロックを要求する




上の状況で生じるデッドロックをTrXの視点から
見たもの（待機グラフ）

# デッドロックの対策：予防する

デッドロックを生じないようにAPを作成する

a)必要なロックは1度にまとめて要求する

- 先の例では以下のように「最初に SELECT して、行ロックを獲得する」という手法が利用できる

```
SELECT * FROM account WHERE id = 1 or id =2
FOR UPDATE;
```

最初にアクセスする際に、明示的な行ロックを用いて
そのトランザクション内で更新する全ての行のロックを
獲得する

b)全てのトランザクションで、表や行へのロックの要求の順序を一定にする

# デッドロックの対策:リトライする

- デッドロックを生じることを前提にAPを作成する
  - PostgreSQLはデッドロックを自動的に検出する
  - 検出すると、デッドロックに含まれるトランザクションが1つエラーとなってアボートされる
    - ロックを要求したSQLがエラーで戻ってくる
    - エラーコード(SQL STATE)は’40P01’ (deadlock_detected)
  - デッドロック時はロールバックして、再実行(リトライ)する
- 全てのデッドロックを予防することは困難であるから、リトライの仕組みを用意すると良い

INTRO-
DUCTION
TO
TRANS-
ACTION

# 今日のまとめ

# まとめ (1/2)

- 一貫したDB操作のためにはトランザクションが必要
  - ACID性が保証される
- トランザクションは複数のSQL文のあつまり
  - BEGIN(または START TRANSACTION)で始まる
  - COMMIT あるいは ROLLBACK で終わる
- トランザクションには「分離レベル」が指定できる
  - 他のトランザクションの実行結果がどこまで見えるか
  - 3つの分離レベル (SERIALIZABLE, REPEATABLE READ, READ COMMITTED)があり、前の方ほど確実に分離される
  - 分離レベルと性能にはトレードオフの関係がある

# まとめ (2/2)

- トランザクションでは表や行を他のトランザクションによる更新から保護するためにロックを用いる
  - ロックが獲得できないときにはトランザクションは待機する
  - SQLを実行する際に自動的にロックが獲得される
  - 獲得したロックはトランザクション終了まで開放されない
  - 明示的にロックを要求することもできる
- ロックの副作用：デッドロックを生じることがある
  - AP設計時にデッドロックを予防するのが望ましい
  - 予防以外にデッドロック発生時のリトライを作りこんでおく

# トランザクション　その他の話題

- 今回のお話はアプリケーションプログラム開発の観点から話題を選びました
- DB管理者の観点からは、以下のようなテーマも重要です
- ACID性のD（持続性）については触れませんでしたが、クラッシュ時のリカバリやオンラインバックアップといった重要なテーマがあります
  - 継続的アーカイブとポイントインタイムリカバリは重要
  - リカバリに関連して、チェックポイントの設定も重要
- デッドロックや長時間ロックの特定方法

# 参考文献


1.PostgreSQLグローバル開発グループ, “PostgreSQL マニュアル”, 1996-2017.

2.増永 良文, “リレーショナルデータベース入門 (第3版)”, 第10章, サイエンス社, 2017.

- RDB全般について基礎（理論）から解説した教科書。

3.松田 神一, “オススメ！ OSS-DB情報”, “トランザクション”,”デッドロックについて”, http://www.oss-db.jp/measures/dojo.shtml

- PostgreSQL特有の振る舞いや設定のTIPSを挙げており、参考になる

INTRO-
DUCTION
TO
TRANS-
ACTION

# おわり

おつかれさまでした

# この資料について



改訂履歴

| 改訂日 | 改訂者 | 内容 |
|---|---|---|
| 2017.5.27 | 坂田 哲夫 (NTT OSSセンタ) | 初版作成 |
| | | |
| | | |
| | | |
| | | |