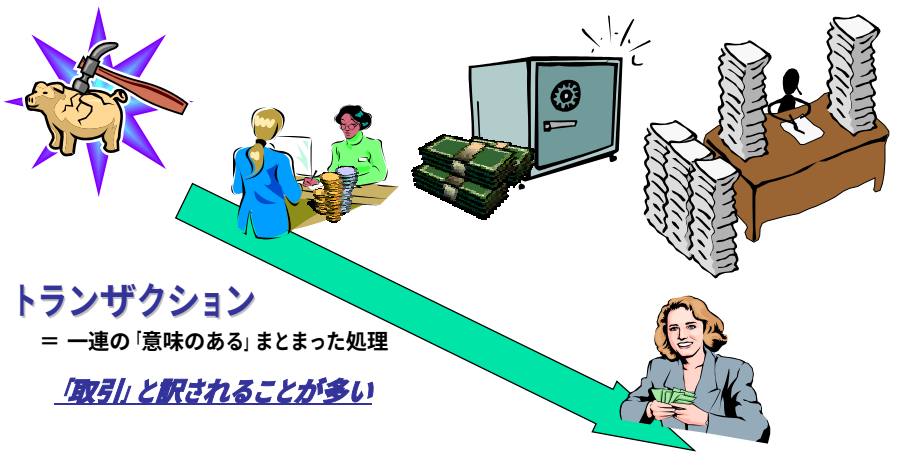
UL Systems, Inc.


# いま改めて問う “トランザクション”徹底解説

**2002年10月11日**

**ウルシステムズ株式会社**
http://www.ulsystems.co.jp
Mail to: info@ulsystems.co.jp
Tel: 03-6220-1400 Fax: 03-6220-1402

## トランザクション処理とは？



**トランザクション**
＝ 一連の「意味のある」まとまった処理

***「取引」と訳されることが多い***

# 典型的な“トランザクション”

**よく文献で解説されるトランザクション処理**

飛行機の “座席予約”、“空席照会”

銀行の “残高照会”、“入出金”、“振込み”

商品売買の “伝票決済”、“在庫確認” など

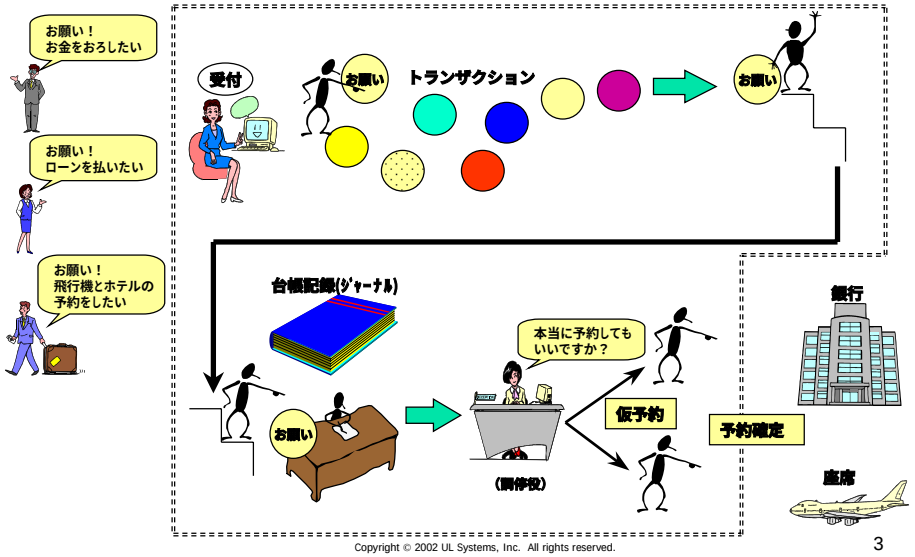
**単語としてのトランザクション**

- データベース
  データ処理
  COMMIT, ROLLBACK
- TPモニター
  分散トランザクション処理
- アプリケーションサーバ
  Web、Java
- メッセージキュー
- オンラインとバッチ

- トランザクションは、共有データの更新を行う。
- 処理に何ら問題がなければ、トランザクションは commit (データの恒久的な更新) される。
- トランザクションは abort可能でなければならない。
- commit または abort がなされてはじめて、トランザクションは完結する。




# トランザクション処理の概念



# トランザクション処理の要件

- トランザクション処理への要求は、“厳しい”
  - ユーザからの膨大なデータアクセス要求をこなす
  - 同時にかつ期間内（一般的には数秒（2～3秒）以内、バッチは数H）に確実に処理する
  - 大事な処理をしている
  - 失敗すると、理由を問わず怒られる
- システムへの要件
  - DB（データベース）：データアクセス手段（SQL等）、バックアップ、リカバリ等の手段が提供されること。
  - DC（データコミュニケーション）：様々な通信手段が提供されること。オープンで簡単な手順であること。
  - 信頼性：トランザクション単位に処理の結果が保証されること。また、トランザクションの復旧手段を有すること。
  - 処理能力：優れたコストパフォーマンスを有し、システム負荷やユーザ数に応じてスループット（TPS: Transaction Per Second）が向上できること。
  - 柔軟性：ユーザやアプリケーションの要求に応じて動的にシステムの再構成ができること。また、処理すべきトランザクション量の増加に対する拡張機能を有すること。
  - 相互運用性：他システムとの連携処理が可能なこと。
  - 高可用性：24H365D システム停止時間が限りなく短いこと。
  - 運用性：運用保守、メンテナンス等が簡単で、かつ柔軟な手段として提供されていること。




# トランザクションのACID特性

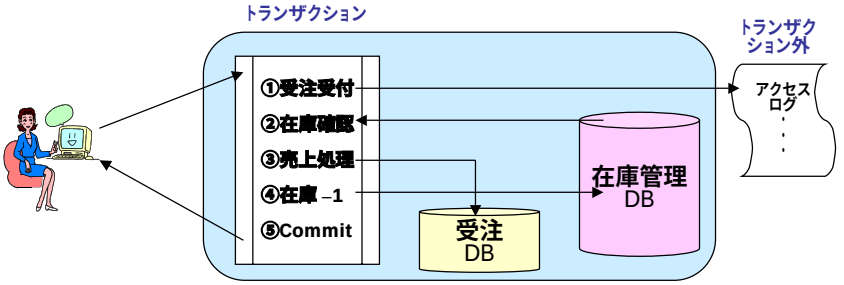
**ACID特性**

| | |
|---|---|
| ***原子性*** (Atomicity) | ● トランザクションが終了した時のデータベースは、トランザクションの全ての処理を完了した後の状態か、または処理を受け付ける前の状態かのいずれかであること。<br>（トランザクションの実行は、「All or Nothing」でなければならない。）<br>● 中間状態がない。Commit or Abort のいずれか一方であること。 |
| ***一貫性*** (Consistency) | ● トランザクション処理の終了状態に係わらず、データベースの内容は一貫性の取れた状態であること。<br>（例：銀行なら、受付係毎の取引の合計と支店の取引合計が等しいこと。）<br>● 途中の処理の順序によらず、トランザクションとしての処理結果は一定であること。<br>● アプリケーションとしてロジックが正しいこと。 |
| ***隔離性*** (Isolation) | ● 複数のトランザクションを同時に実行した場合と、逐次的に実行した場合との処理結果が一致していること。<br>● 更新途中のデータが、他のトランザクション処理に見えないこと。<br>● 通常、ロックを用いた排他制御などをすること。 |
| ***耐久性*** (Durability) | ● トランザクションとして一旦完了（Commit）すれば、その後の障害などでデータベースの内容が変化しないこと。<br>● たとえマシンダウンしても、結果を無くさないこと。 |

**すべては“楽に”開発できるようにするためだが・・・・**
**誰が何をどこまでやってくれるのか（やってくれないか）を知ることが重要**

# Atomicity (原子性)

- トランザクションは、All or Nothing しかない
  - データベース、Webアプリケーションサーバ、TPモニタなどがやってくれる
  - COMMIT、ROLLBACK（それぞれの自動処理もある）
  - 最後まで完全に実行されるか、全く実行されないかのいずれかのみ
  - 「Commitするまでは、いつでもAbortできる」のがポイント
- トランザクショナルな処理と、そうでないものが混在するのが現実
  - 業務、またはアプリケーションでカバーする必要







# Consistency (一貫性)

- トランザクションの実行の前後で、データのつじつまが合っていること
  - 例：各口座の残高合計が、支店の残高トータルと一致していること
  - 例：5万円預金してから、2万円引き出したら、結局3万円残っていること
  - 例：データベースの主キーがユニークであること（整合性制約）
- 一貫性は、アプリケーション開発者の責任が重い
  - 大抵は、バグにより一貫性が壊れる
  - どのレベルの一貫性を、いつチェックするか

# Isolation（隔離性）

- トランザクションは、互いに干渉しない！
  - 複数のトランザクションを同時に実行しても結果がお互いの存在に影響されないこと
  - Serializable：1つ1つ順番に実行するのと結果が同じであること
- これは、とても難しいテーマです
  - “他のヒトが更新中です・・・”？
  - “すいません、さっきあった在庫はなくなりました・・・”？？
  - “ボタンを2回押さないでください・・・”？？？
  - みんな順番に処理していたのでは間に合わない！データベース側の技術をどう使いこなすか

**Concurrency Control（平行処理の制御技術）**

- ロック
  - 排他制御の基本、同じリソースを使う相手を“こちらが終わるまで待たせておく”
  - SELECT XXXX ・・・・・ FOR UPDATE
  - ロックが外れるまでは、他のトランザクションは待たされる
- バージョニング
  - データが更新されるたびにバージョンが上がる。同じバージョンのデータを扱う。
  - 別バージョンを更新しようとするとエラー




# データベースでの隔離レベル

- 隔離レベル（Isolation）レベルを調整しないと、性能がやばい
  - すべてSERIALIZABLEでは、とても待ってられない。緩めることで性能が向上する。
  - 緩めた場合に何が起きるか、アプリケーション側の対処が必要。

安全 厳しい

| 隔離レベル | ファントム | Non Repeatable Read | Dirty Read |
|---|---|---|---|
| SERIALIZABLE | - | - | - |
| REPEATABLE READ | 発生 | - | - |
| READ COMMITTED | 発生 | 発生 | - |
| READ UNCOMMITTED | 発生 | 発生 | 発生 |

緩い

普通のDBのデフォルト

ファントム：
途中で新しいデータ挿入ができる。
ある条件に合致するトータル数を求めてる間に挿入されるとズレる。

Non Repeatable Read：
トランザクション中に、データの値が読める。
在庫残があると思って処理すると途中で無くなるかも。

Dirty Read：
コミット前の別トランザクションが更新中のデータが読める。
別トランザクションがAbortするとやばい。

# Durability (耐久性)

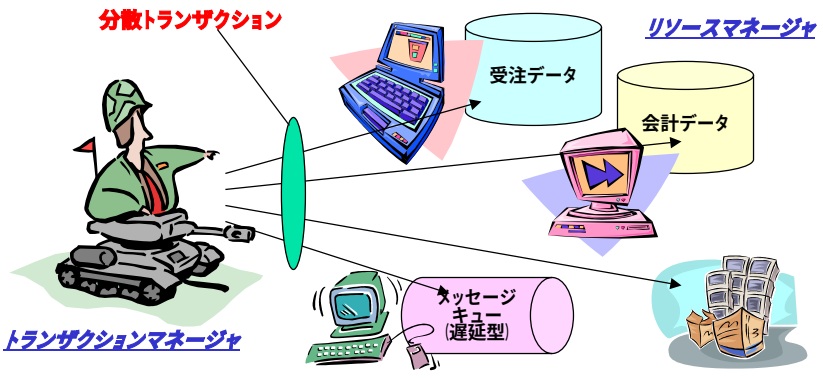
- トランザクションの処理結果は、何があろうとしっかり保証される
  - 「たとえマシンが落ちても、結果は正しく残っている」
  - データベース及びTPモニターによる“リカバリー処理”が行われる
    - DBでは一般に「遅延書き込み」方式でメモリーキャッシュを有効活用している
    - REDOログ、UNDOログ相当により、マシン復旧時などに正しくデータ整合を保つ
  - ハードウェア的な工夫
    - バックアップ、RAIDディスク、無停電装置、二重化など
  - アプリケーションとしての対処
    - ジャーナル取得
    - 業務としてのリカバリー、キャンセル処理 など





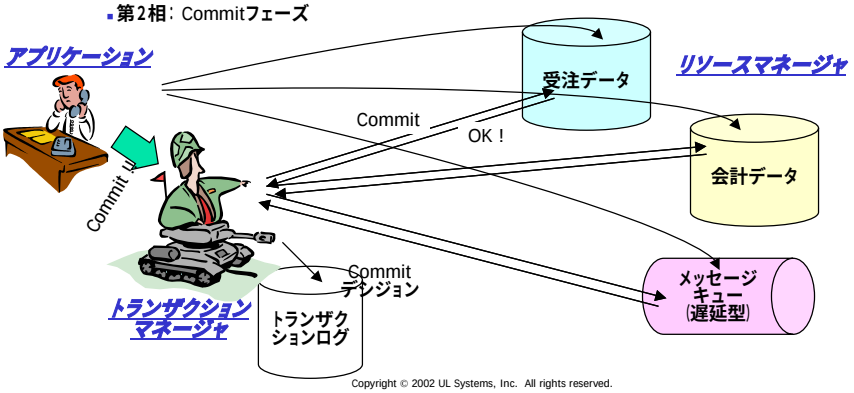
# 分散トランザクション

- “複数のリソース” にまたがったトランザクション
  - リソース：データベース、メッセージキュー、その他アプリケーション資源
  - ACIDの実現が、とたんに難しくなる



# 2フェーズコミットとは

- 2フェーズコミット ＝ 2相コミット
  - 複数のリソースマネージャ間を調停し、分散トランザクションの原子性を確保するためのCommitプロトコル
  - 一般には、TPモニターやWebアプリケーションサーバが“トランザクションマネージャ”となる
  - 2つのフェーズからなる。
    - 第1相：Prepareフェーズ
    - 第2相：Commitフェーズ

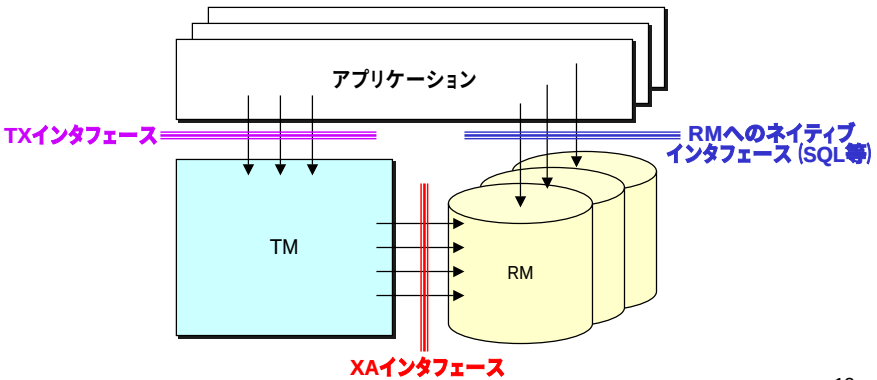




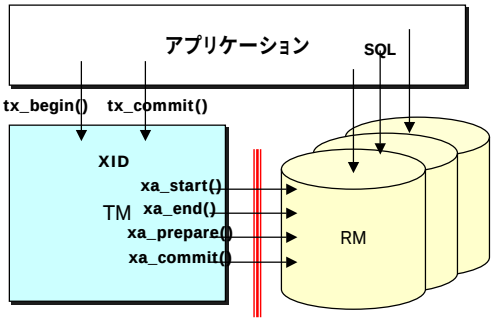



# 分散トランザクションの処理モデル

- X/Open DTPモデル
  - X/Openで規定された業界標準の処理モデル（ちょっと古い時代）
    - アプリケーション
    - TM：トランザクションマネージャ
    - RM：リソースマネージャ
  - “XAインタフェース”を定義したのが大きな功績



# XAインタフェースの実装例




| 主なXA関数 | 機能 |
|---|---|
| xa_open<br>xa_close | RMをオープンする<br>RMをクローズする |
| xa_start<br>xa_end | トランザクションを開始する<br>トランザクションを開始する |
| xa_prepare<br>xa_commit<br>xa_rollback | コミット準備を指示する<br>トランザクションをコミットする<br>トランザクションをアボートする |
| xa_recover<br>xa_forget | 未完了トランザクションの一覧を取得する<br>トランザクションを破棄する |

1. トランザクション開始
   - tx_begin()によりAPがTMに指示
   - XID (トランザクションID) が割り当てられる
2. トランザクション内でDBアクセス
   - 以下の処理を繰り返し、複数リソースを更新
     1. xa_start()でTM→RMにサブトランザクションスタートの指示
     2. APがSQLによりRMにアクセス
     3. xa_end()で、TM→RMにサブトランザクション終了の指示
3. トランザクション完了
   1. tx_commit()によりAPがTMにコミット指示
   2. TMは2フェーズコミットを実行
      1. xa_prepare()を呼び、TM→RMにPrepare指示
      2. xa_commit()を呼び、コミット完了




# 分散トランザクションは難しい！

- “分散トランザクションが難しい”理由は
  - 中央集権的にすべてを制御することは、事実上不可能
  - 複数のマシン間・DB間に、正しくトランザクションを伝播させなければならない
  - トランザクションへの理解・解釈が、マシンをまたがって同一でなければならない
  - 様々なエラー (通信エラー、局所的なマシンエラー、タイムアウト等) に対して、それぞれに適切なリカバリ処理を行わなければならない
- 現実的なアプリケーション側の対処が随所に必要
  - データベースは、未来永劫Commit指示を待ちつづけるわけにいかない
    - →勝手にCOMMIT/ROLLBACKすることもあり得る (Heuristic Completion)
    - →クリティカルな場合も考えて、アプリケーション側の対処が必要
  - 既にあるアプリケーションは、外部のトランザクションに合わせられない
    - →ホスト業務、ERPパッケージなど
  - アプリケーション側のロジカルなデッドロックは誰も助けられない
    - →唯一、トランザクションタイムアウトのみ
  - 性能のため、ベンダーは“楽観的な”実装をしてくる
    - →データベースの隔離性レベル、EJBコンテナ
    - →トレードオフを理解し、業務側で対応する必要あり
  - トランザクション関連のインタフェースは皆プリミティブ
    - →よく知ってないと、とても危険
  - 正しく全部Abortしてくれても (大事!) 、本当は嬉しくない
    - →エラー時のアプリケーションでのリカバリが必須

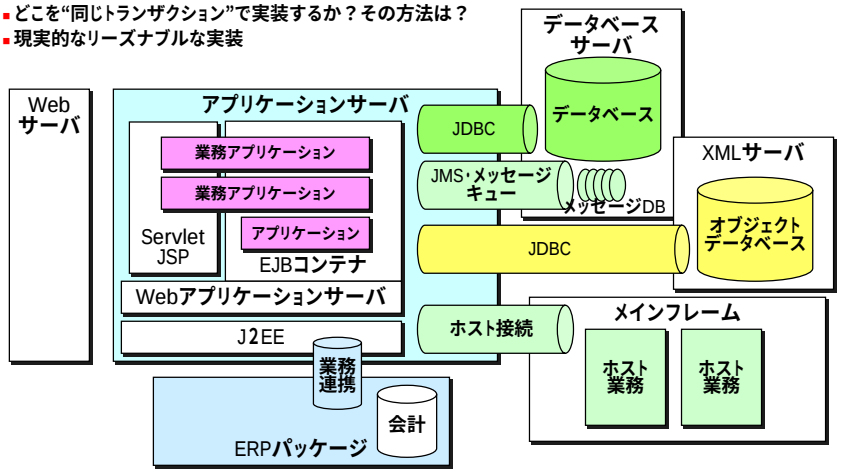
**イマドキの“トランザクション”**

- EJBコンテナ、JMS
- データベース (SQL、XML 等)
- バッチトランザクション
- ジャーナル

**本格的なWeb系のシステムであれば分散トランザクションがMust**

# “改めて問う”トランザクション処理システム

- 複雑な、異種トランザクション統合が必要
  - 多様なデータ統合と連携処理
  - Webからのオンライン＋遅延型＋バッチ＋ホスト＋ERP
  - どこを“同じトランザクション”で実装するか？その方法は？
  - 現実的なリーズナブルな実装



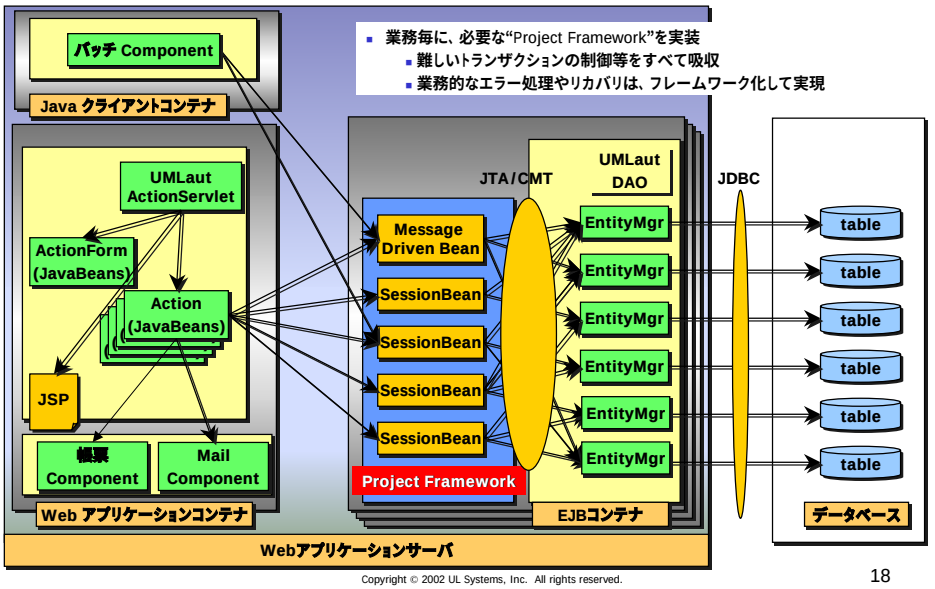




# Java環境でのトランザクション

- トランザクション制御をするには、EJBを使おう
  - EJB (Enterprise Java Beans) のコンテナがトランザクションを自動で管理してくれる
    - Container Managed Transaction – EJBに対してトランザクション属性を設定するだけ
    - EJBコンテナが、従来の“トランザクションマネージャ”相当の機能を果たしてくれる
  - とはいえ、分散トランザクションに対する注意は今までと同じ
    - コーディングが楽になっただけ（それでも、大きな進歩）
- JTA：Java Transaction API
  - やりたいヒトには、自分でトランザクション制御できるようなインタフェースも用意されている
  - その他、大抵のインタフェースがJavaでは実現されている
    - JDBC – データベースへのインタフェース
    - JMS – メッセージキュー（遅延型トランザクション）へのインタフェース

# J2EEでのトランザクション実現（弊社例）








# 現実的なTips

- トランザクションシステムは、“うまくいく”ことを保証してくれるわけではない
  - All or Nothing が保証されている。そこまでで精一杯。
  - クリティカルな場合の障害は、整合性すら失われるかもしれない。
  - 必要なところだけにうまく使って、“楽をする”ように心がける
    - 全部アプリケーションで記述するのは、とても大変 － ちょっと（だいぶ）楽しよう
    - 最終的には、業務で対応しなければならない － コミット時にエラーになっても嬉しくない
    - どこまで誰が何をやってくれるのか、正確に理解しておくこと － 特にリカバリ処理

# 現実的なTips

- 性能を出すのは絶対条件
  - オンライン処理、バッチ処理とも、性能要件は絶対。
  - 機能要件から先に実装して最後に性能テストをしても無駄
  - 初めからサイジング＆アーキテクチャ設計が大切
    - スケールしていく際、どこが問題となるか
    - 数が増えたらどうなるか
    - データ量が増えたらどうなるか
    - 算術計算は危険 － 10TPSと100TPSは単に10倍違うわけではない
    - 不必要に、すべてを“トランザクション処理”することはない
  - データベース、EJB等の排他制御と性能はトレードオフ
    - もちろん、業務ロジックやDB設計が稚拙なら問題外
    - しっかり設計していても、気づかず陥る落とし穴が、“隔離性”などからくるエラー
      - ファントム、Non Repeatable Read など
      - 二重サブミットの防止方法



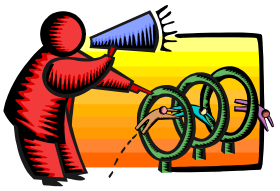


# 現実的なTips

- 業務としてのエラー処理、リカバリを常に最優先
  - 障害系で問題を大きくするのは、常にヒト
  - 障害が起きるのは、“通常業務”であることを認識する
    - 証券の約定でエラーの場合、業務はどうする？クライアントへはどう対応する？
    - 万一受注システムがおかしくなったら、どう業務で対処する？
    - 料金計算が誤った結果を出してしまったら、果たしてどう対応する？

## 現実的なTips

- モニタリング、運用管理が重要
  - “何が起きているのかわからない”は最悪
  - どううまくいっているか、まですべて見えるようにしておく
  - ログは大切、ただしログの運用も大切
  - データセンター等での運用、データ管理、モニタリングツールの利用







UL Systems, Inc.

***ありがとうございました***

http://www.ulsystems.co.jp
email: info@ulsystems.co.jp