Lv1から始める Webサービスのインフラ構築

2014-09-09 AWS Cloud Storage & DB Day

株式会社マイネット
伊藤 祐策

自己紹介

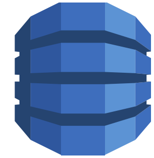
名前　　　伊藤 祐策
勤務先　　株式会社マイネット
肩書　　　アーキテクト
事業内容　スマートフォン向けゲームの開発・運営
お仕事内容

・自社ゲームタイトルのサーバーインフラ構築
・アプリケーション開発
・主にサーバーサイド設計（特にDB設計！）

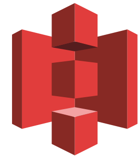
自己紹介

大好きなAWSサービスは？

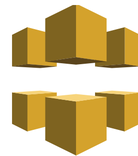
1位.**Amazon DynamoDB**



2位.**Amazon S3**



3位.**Amazon CloudFront**



自己紹介

まあでも青いアイコンのサービスはだいたい大好きです。

今日のお話はこんな人におすすめ

Webサービスを作って
スタートアップしたい人

ユーザー数1人から100万人までをAWSで！

もくじ

第一部

Lv1から始めるWebサービス

第二部

スケーラブルな構成にするには？

第三部

DynamoDBの正しい使い方

第一部
Lv1から始めるWebサービス

シナリオ

あなたはとあるWeb系会社のエンジニアです。
ある日、社長が突然こんなことを言い出しました。

「我が社もソーシャルゲーム事業に参入するぞ！」

一瞬目眩がしましたが、あなたは覚悟を決めてシステム設計を開始しました・・・。

※このシナリオは全てフィクションです

シナリオ

サービスリリースまでのステップ

1. アプリケーション開発
2. 社内アルファテスト（ユーザー数１０人）
3. ベータテスト（同５００人～？？？）
4. 正式オープン（同１万人～？？？）

シナリオ

先輩社員の助言によりAWSを採用することは決定しましたが、あなたはAWSは全くの未経験でした。

そこでまずはAWSのアカウントを取るところから始めることにしました。

【最初の目標】

アルファテスト用の環境を構築する

Step1 アカウント取得

AWSアカウントを取得する

★ここがポイント

必ず２アカウント用意しよう！

・本番環境用アカウント

・開発環境用アカウント（兼試験環境）

テストや訓練に費用を惜しまないこと！

Step2 IAM

IAMで子アカウントを作成する

グループは以下の２種類を作る

・AWSコンソールにアクセスする「人間ユーザー」
　→　Administratorテンプレートをそのまま使う
・アプリケーションユーザー
　→　必要最低限のアクセス権限だけを付与する

※IAMを作ったらrootアカウントは封印しましょう

Step3 VPC

VPCを構築する

サブネットとゲートウェイを作成して関連付ける

★ここがポイント

・サブネットは適切に切る（後述）

・"Auto-Assign Pulibc IP"をONにする

→ EIPを使う数を節約できる

Step3 VPC

サブネットはこんな分割方法がオススメ

10.1.0.0/17 ← まずは半分を**AZ-a**に

10.1.128.0/18 ← 残りを半分を**AZ-c**に

10.1.192.0/19 ← さらもう半分を**AZ-a** に

領域を使いきってしまうとあとで困る！

Step4 セキュリティグループ

セキュリティーグループを構築する

サーバーの役割種別ごとにセキュリティーグループを１個作る。

【例】

・Webサーバー

外部から80番、443番。内部から22番。

・RDS(MySQL)サーバー

内部から3306番。

Step5 EC2

EC2インスタンスを作成する

★ここがポイント

・配備先AZに気をつけて！

→ リザーブドインスタンスを買う時に困る

→ たまにインスタンスタイプが枯渇する

※AWSの営業の人に相談しよう

・セットアップが完了したらAMIをとっておこう！

Step6 EIP

EIPを取得する

外部に公開するインスタンスのENIにアタッチする。

★ここがポイント

・EIP取得数の制限に注意！（申請で解除可能）

・インスタンスタイプ別にも関連付け可能数の制限がある

Step6 EIP

必要なEIPはいくつ？

1個目 一般公開サイト用
2個目 運営管理サイト用
3個目 SSHゲートウェイサーバー用

だいたい3個もあれば十分なのです！

Step7 Route53

Route53でゾーン設定をする

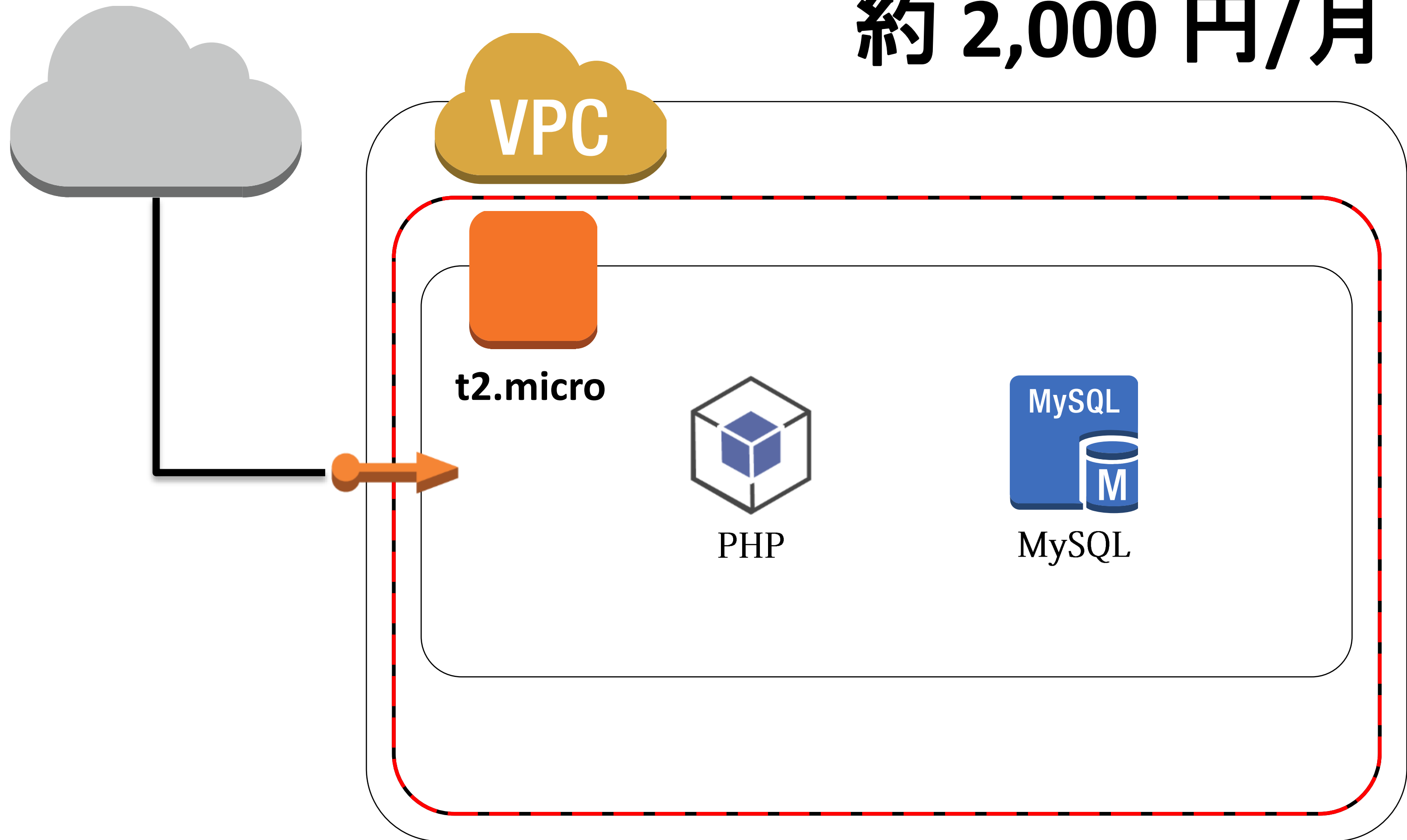
取得したEIPをホスト名登録する

★ここがポイント

・メール送信するときはSPFの設定を忘れずに！

・さらにEIPに対するメール送信制限解除申請も必要なので一緒に済ませておこう！

システム構成図(Lv1)



シナリオ

アプリケーションも完成に近づき、いよいよ一般ユーザーへサービスを公開することにしました。

しかし先輩社員はこんなことを言いだしました。

「この構成でインスタンスタイプ上げるだけじゃダメなの？」

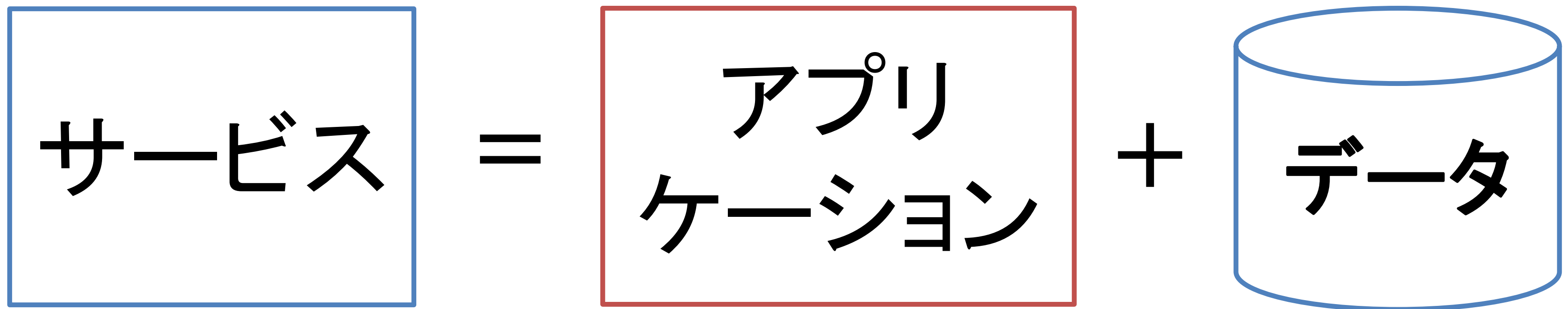
問題

この構成のままインスタンスタイプを上げるだけでは商用環境としてダメな理由を答えなさい。

解答

データの**保全性**が確保されていないから。

Webサービスとは



Webサービスは「生き物」です！

保全性について

「アプリケーション」はsubversionやgithub等にマスターがあるので**保全性**が確保されている。

保全性について

一方「データ」はEC2のEBS上にあるのである日突然失われる可能性がある。

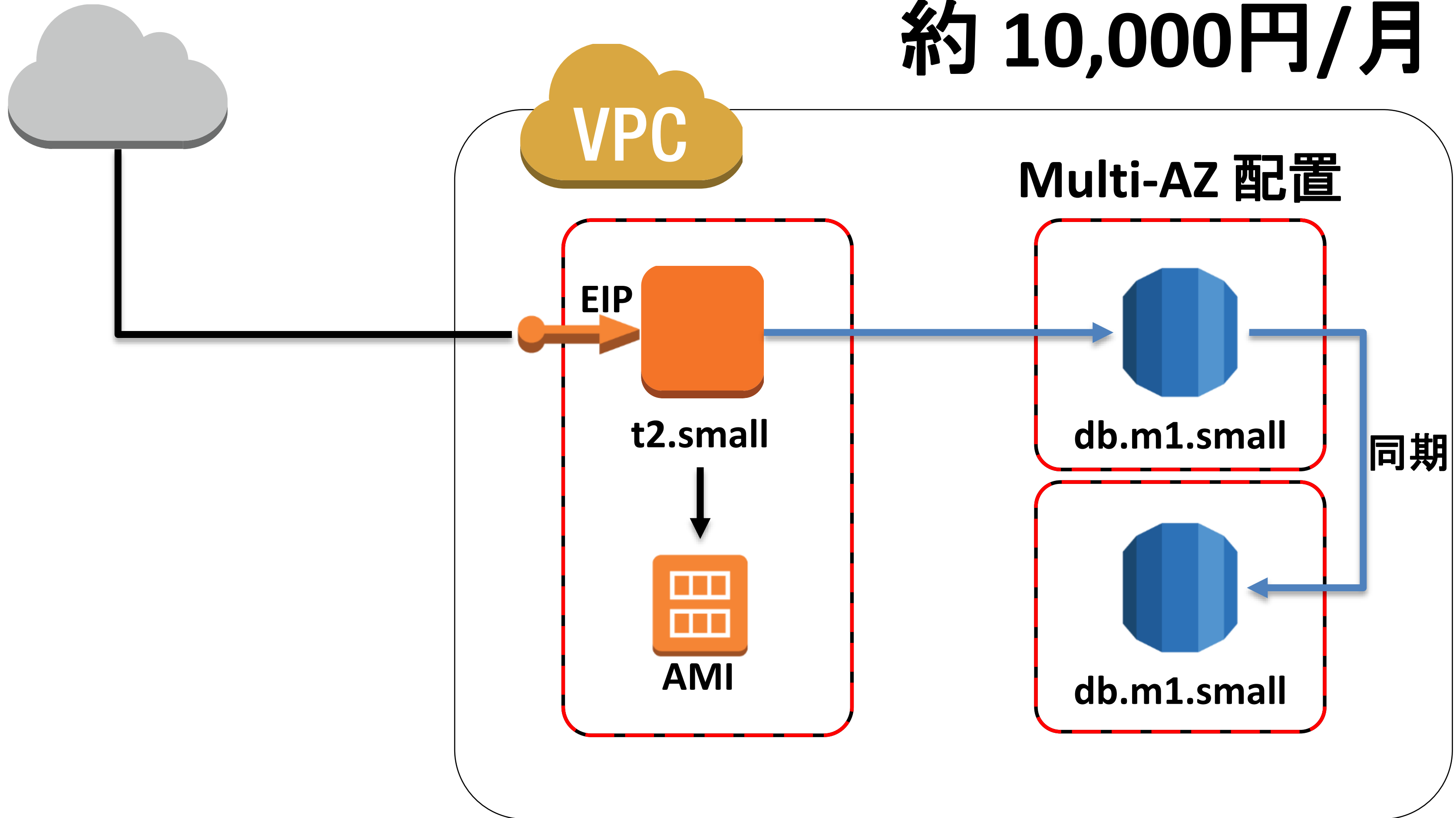
データが消失 → **サービス終了！**

商用環境の最低ライン

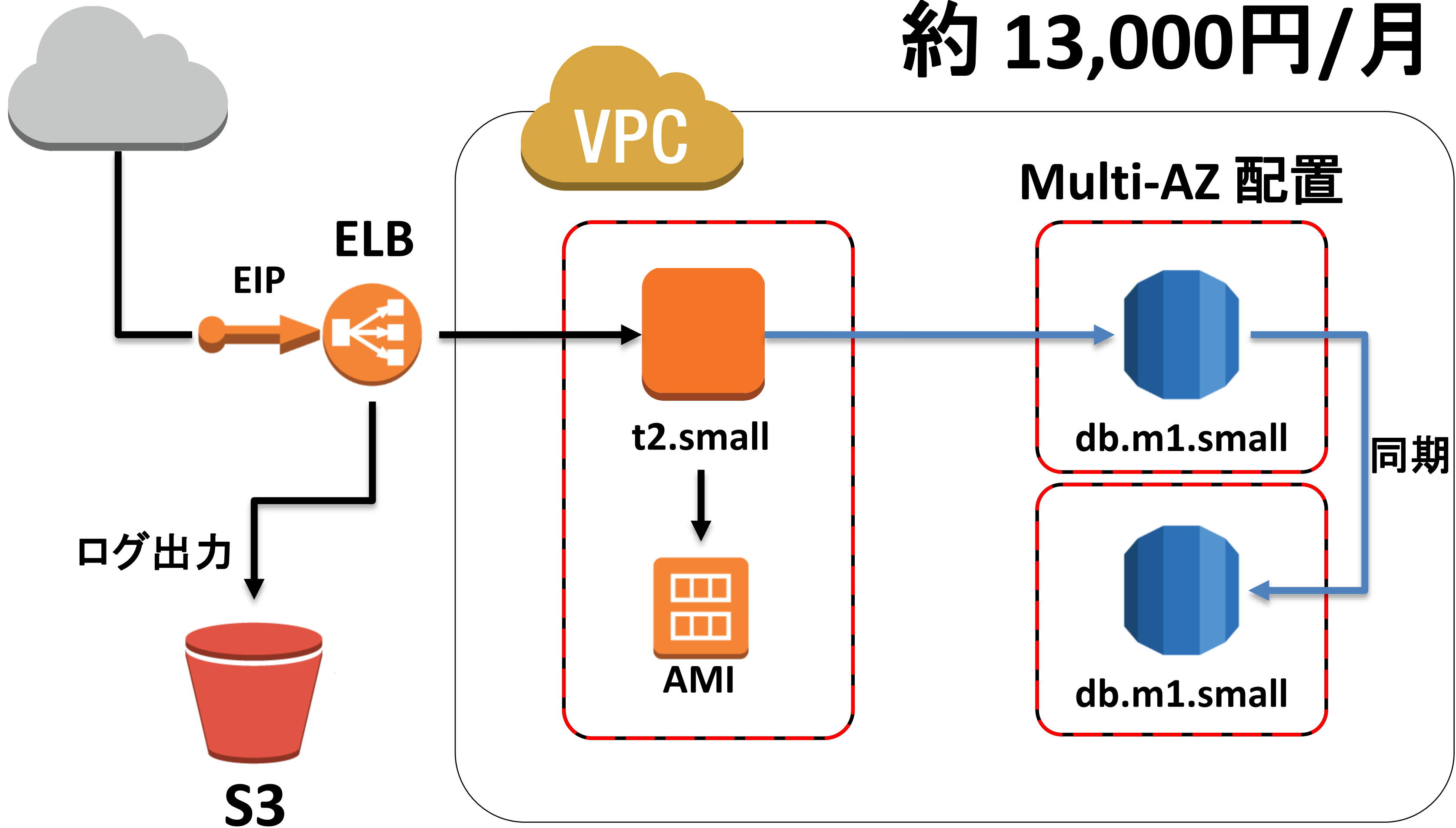
「隕石が直撃しても大丈夫」

データセンターが１つ壊滅してもサービスを復旧できること。

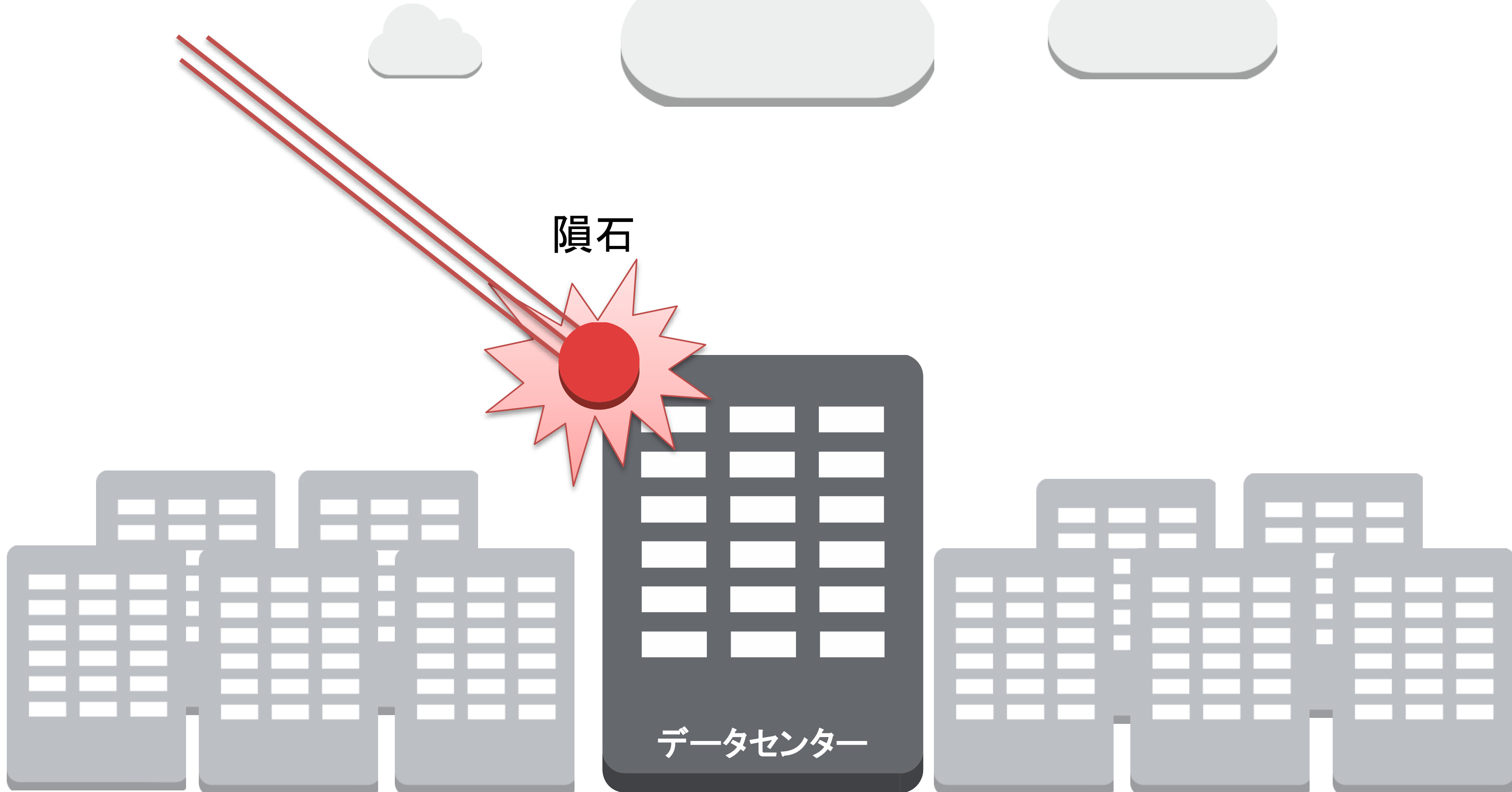
システム構成図(商用Lv1)



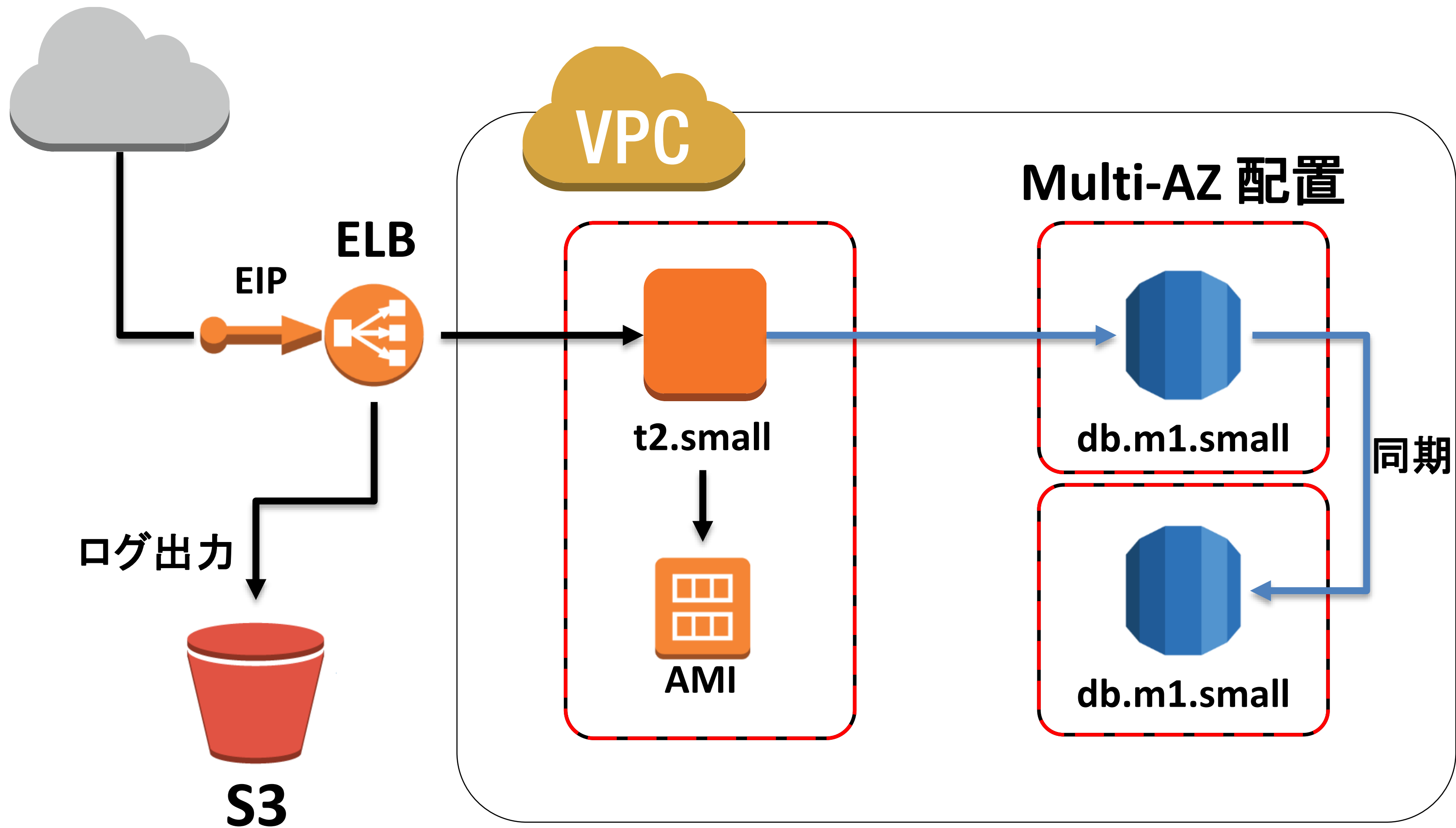
システム構成図(商用Lv2)



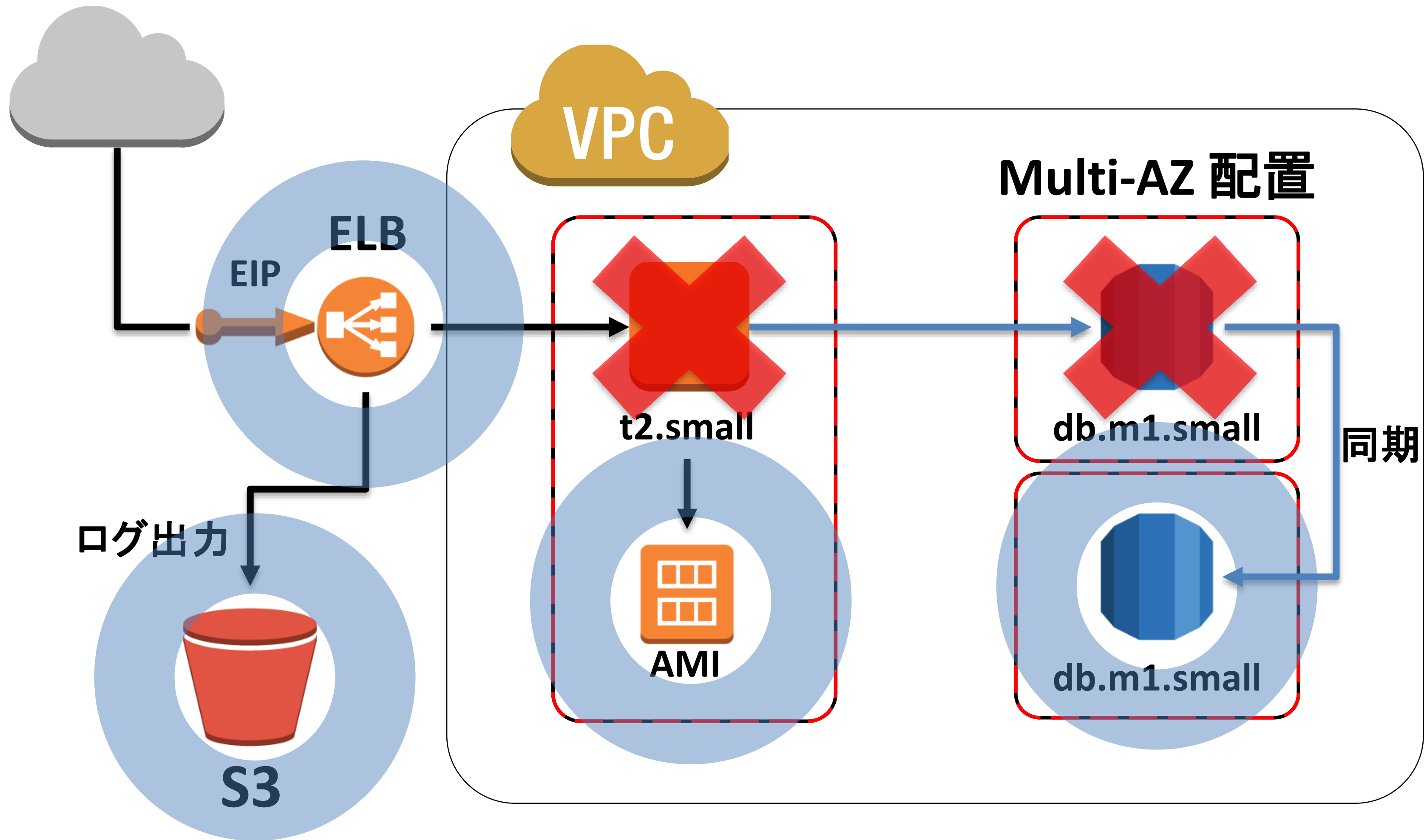
ちょっと隕石当ててみましょう



システム構成図(隕石直撃前)



システム構成図(AZ壊滅後)



問題

以下のAWSストレージ系サービスうち、デフォルトでデータの保全性が確保されているものはどれか？

S3

EBS

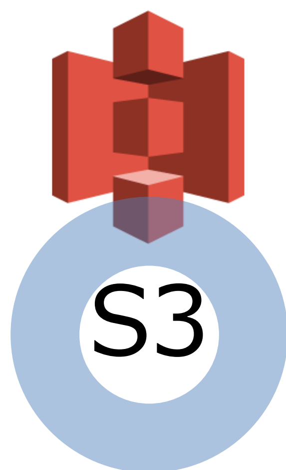
DynamoDB

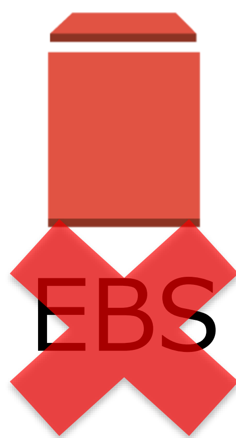
RDS
Multi-AZ

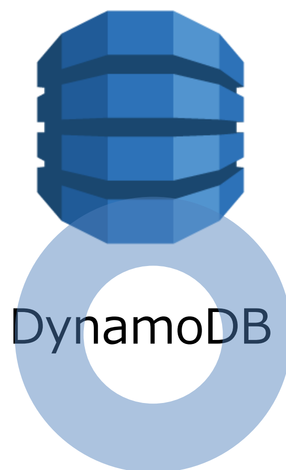
ElastiCache

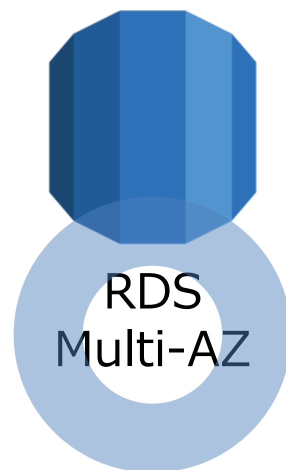
解答

以下のAWSストレージ系サービスうち、デフォルトでデータの保全性が確保されているものはどれか？









まとめ

「大事なデータ」は保全性が確保されているストレージサービスに保存しましょう。

データさえ生き残っていればサービスは何度でも蘇ります！

この式は見覚えありますよね？

$$A = \frac{MTBF}{MTBF + MTTR}$$

A .... 可用性
MTBF ... 平均故障間隔
MTTR ... 平均復旧間隔

まずはMTTRを∞にしない保証を作ること

・・・というお話でした。

$$A = \frac{MTBF}{MTBF + MTTR}$$

コレの件

A .... 可用性
MTBF ... 平均故障間隔
MTTR ... 平均復旧間隔

質問タイム

2分ほど休憩

第二部

スケーラブルな構成にするには？

シナリオ

保全性の確保された構成の構築方法はわかったのですが、この**「商用Lv2」**の構成ではベータテストの負荷には耐えられそうにありません。しかし、ベータテストでは何人のユーザーが押し寄せるのかが全く検討もつきません。

【次の目標】

想定以上の負荷が来ても

すぐに対応できる環境を構築する

用語おさらい

「スケーラブル」とは？

1. 増大する負荷に容易に対応できる

2. 負荷に合わせて自動的に拡張される

用語おさらい

「スケーラブル」とは？

1. 増大する負荷に容易に対応できる
   ↑こっちの話をします
2. 負荷に合わせて自動的に拡張される
   ↑これはややこしいのでまた今度...

用語おさらい

スケールアップ

ノードの性能を上げること

＝インスタンスタイプを上げること

スケールアウト

ノードの数を増やすこと

＝ インスタンスを追加すること

理想パターン

・**EC2インスタンスを追加する**と全体性能があがる。

・**RDSのリードレプリカを増やす**と全体性能があがる。

・**DynamoDBの性能予約を買い足す**と全体性能があがる。

将来が不安なパターン

・**インスタンスタイプを上げる**と全体性能があがる。

・**EBSのIO性能を上げると**全体性能があがる。

→ **コスト効率が悪くなる**

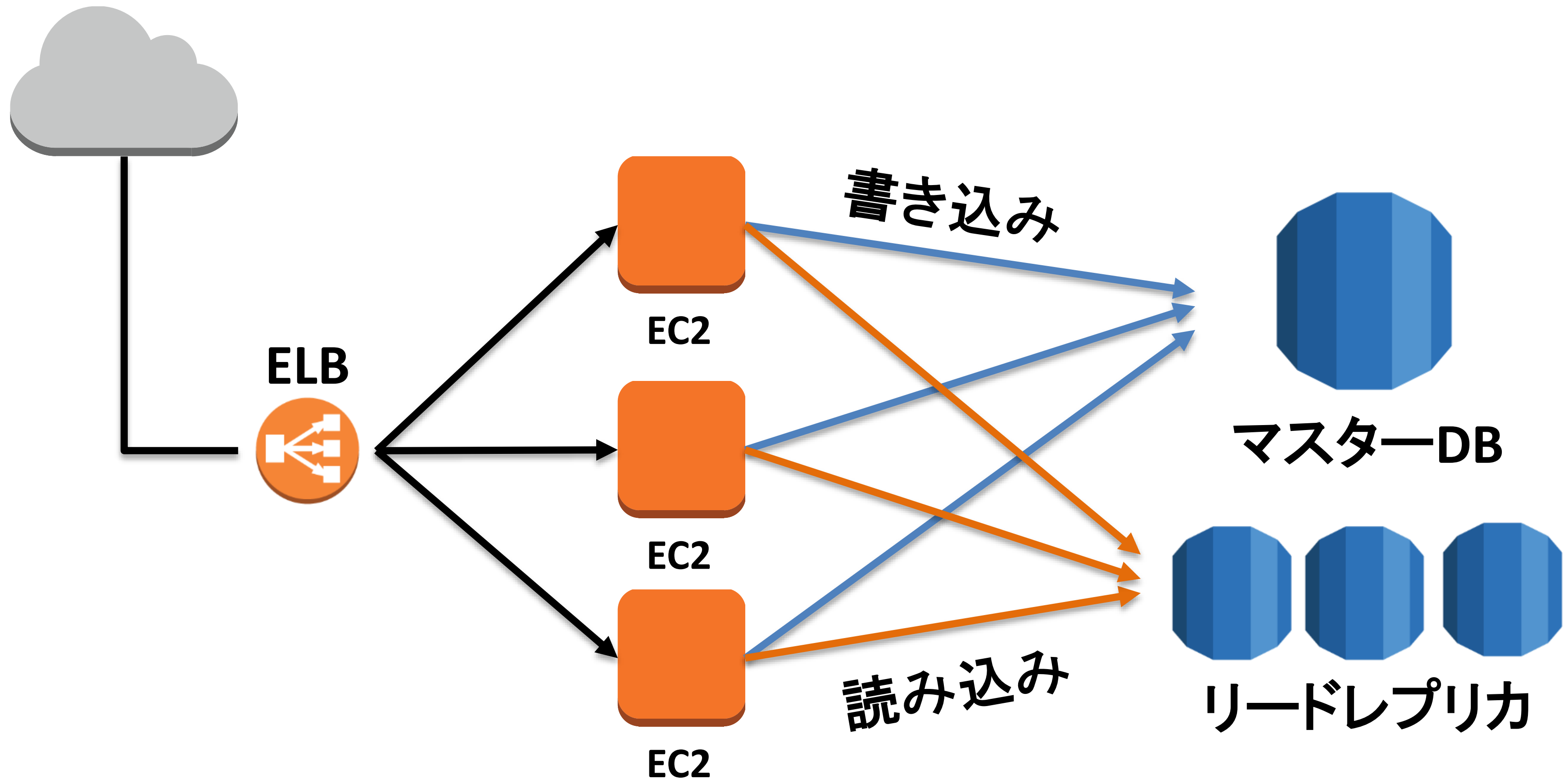
→ **性能拡張に上限がある**

つまりこういうこと

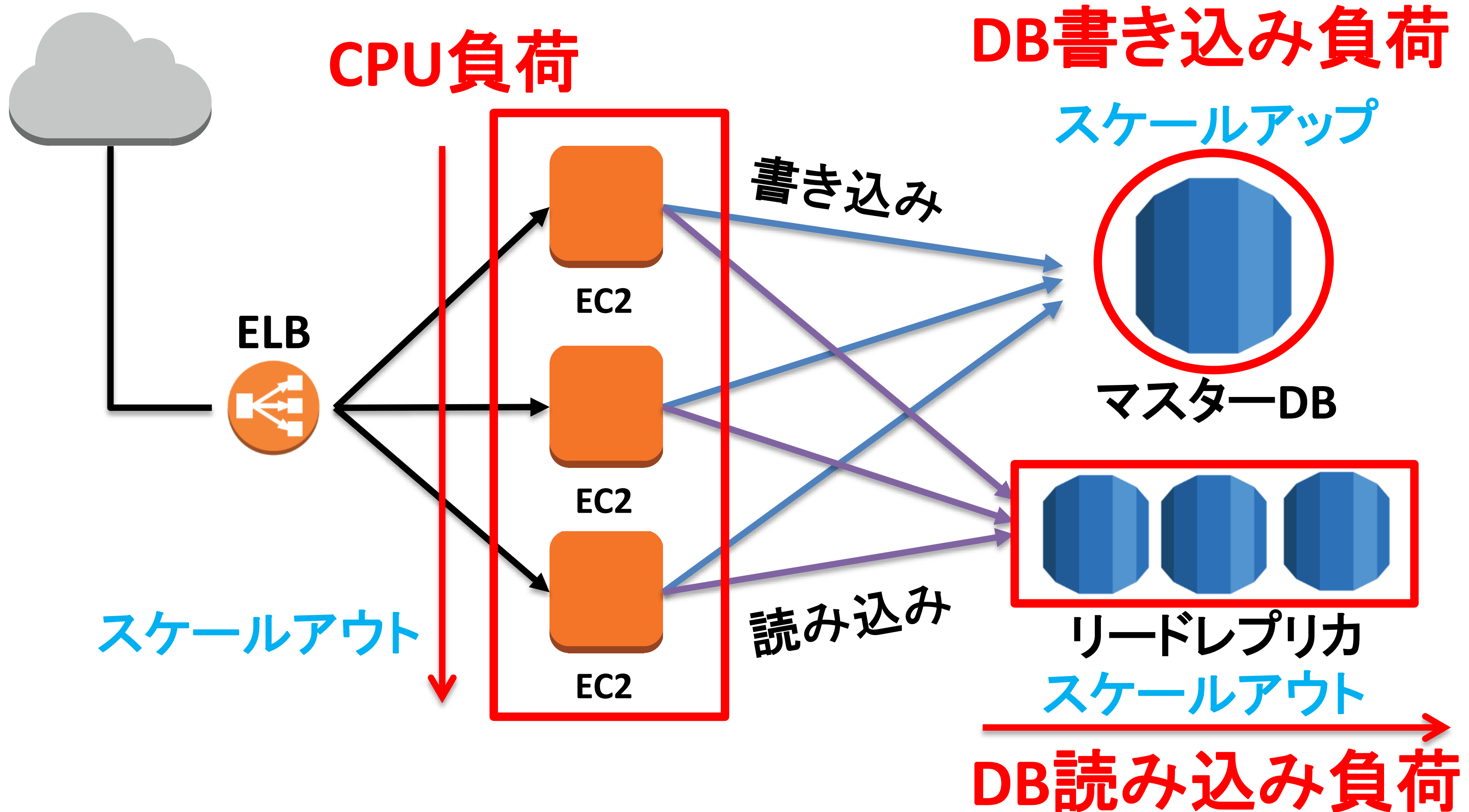
スケールアウトできる形にする

スケールアップで全体性能があがるのは当たり前！

スケーラブルな構成（基本形）



スケーラブルな構成（基本形）



サーバー負荷の傾向と対策

ボトルネックになるのは
いつだってデータベース負荷

＼もう限界／

マスターDB

各種ストレージサービス解説

Amazon RDS

フルマネージドリレーショナルDB

Amazon DynamoDB

フルマネージドKVS型分散DB

Amazon ElastiCache

ただのキャッシュサーバ

Amazon RDS

ここがすごい！

- **メンテナンスフリー！**
  - 自動的に定期バックアップ
  - AZ間でレプリケーション ※Multi-AZ配備時
- **リードレプリカをボタン１発で作成！**
  - 読み込み性能を簡単スケーリング

Amazon RDS

ここは注意！

・**一度起動すると止められない**

稼働停止＝データ削除

EC2のように休止ができない

・**スケールアップ時にアクセス不可になる**

だいたい１０分～３０分くらい

メンテナンスモード必須

リレーショナルDB特有の問題

マスターDBへの負荷は
どうあがいてもボトルネックになる。

書き込み処理が激しいアプリケーションではいずれ限界が・・・。

…しかしそこへ救世主が登場！

Amazon DynamoDB

ここがすごい！

- メンテナンスフリー！
- すごい耐障害性　※3箇所以上に分散保存
- 性能予約課金
- 動的な性能調整が可能
- 負荷による性能劣化を起さない

Amazon DynamoDB

Amazon DynamoDBは
マスターDBへの書き込み負荷がヤバい時の救世主！？

Amazon DynamoDB

ここは注意！

- **一貫性のあるバックアップを動的にとれない**

  「一貫性」か「動的」のどちらかを諦める
- **性能上限に達すると一時的にアクセス不可になる**

  ちょっと余裕を持って予約する必要がある
- **単純な機能しかない**

  集計とかは無理です

Amazon DynamoDB

どう使うか？

負荷分散のための補助データベースとして使う

NoSQL初心者にはこちらがオススメ。

メインデータベースとして使う

鬼門。死ヲ覚悟セヨ。 （※第三部で解説）

Amazon ElastiCache

ここがすごい！

・とにかく速い

※中身はただのMemcachedです。

※でも最近Redisも対応しました！

Amazon ElastiCache

どう使うか？

・大事なデータの格納はNG

・ストレージの読み込み負荷を軽減させるためのキャッシュとして使う

まとめ

スケールアウト可能な構成をがんばって構築しましょう。

しかし、それでもいつかはマスターDBの負荷が限界にくることでしょう。

質問タイム

2分ほど休憩

第三部

DynamoDBの正しい使い方

シナリオ

無事リリースされたサービスは幸運にも大ヒットし、ユーザー数を急速に伸ばしていきました。しかしマスターDBの負荷は増大し、インスタンスタイプを**db.cr1.8xlarge**まで上げたのにも関わらず性能の限界が来てしまいました。そこであなたが決断した最後の手段とは･･･。

【次の目標】

DynamoDBを使ってピンチを乗り切る

Amazon DynamoDBとは何か

- **分散データベースである。**
- **Key Value Storeである。**
- **NoSQLである。**
- **スキーマレスである。**
- **フルマネージド型サービスである。**

Amazon DynamoDBの特徴

・ハッシュキーを基に負荷が分散される。

・読込性能、書込性能それぞれの予約した性能量に対して課金される。

・１レコードは64kBまで格納可能。

※キー名も容量に含まれるので注意

使い方別難易度

【**Easy**】
ユーザー単位で独立しているデータだけをDynamoDBに移行して補助的に使う。

【**Nightmare**】
全てのデータをDynamoDBに載せてメインデータベースとして使う。RDSは補助データベースとして使う。

テーブル設計の勘所

★ここがポイント

テーブル設計は
プライマリーキーの設計が命

プライマリキー設計

プライマリキーの仕様

・プライマリキーの形式は2種類から選べる

1. **ハッシュキーのみ**
2. **ハッシュキー ＋ レンジキー**

・処理の分散はハッシュキーによって行われる

・レンジキーでのみ範囲検索が可能

プライマリキー設計

設計例1 ユーザー固有情報

HashKey : ユーザーID

RangeKey : なし

・アカウント情報

・プロフィール情報

プライマリキー設計

設計例2 ユーザーの対ユーザー関係

HashKey : ユーザーID

RangeKey : 対象ユーザーID

・フォロー、遮断

プライマリキー設計

設計例3 ユーザーの行動履歴

HashKey : ユーザーID

RangeKey : ログID

・ゲーム内アイテムの購入

・攻撃コマンドの実行

・etc

ログIDの作り方

- ログの発生時刻から文字列で生成する。
- 乱数も混ぜるといいかも。
- 万が一衝突したらもう一度トライ。

例 : **"20140909163012 34"**

※桁数は固定しましょう

プライマリキー設計

設計例4 ユーザーの所有オブジェクト

HashKey : ユーザーID

RangeKey : オブジェクトID

・所有カード

・投稿記事

※オブジェクトIDはログIDと同じ方法で生成

プライマリキー設計

設計例5 ユーザー間関係情報

HashKey : ユーザーID+対象ユーザーID

RangeKey : なし

・フレンド

※ユーザーIDは小さい方を先にする

シナリオ

マスターDBへ一番書き込んでいたのは実はユーザーの行動履歴でした。そこで、ユーザー行動履歴をDynamoDBに移行させたところ、大幅に書き込み負荷が減って無事ピンチをのりきりました。めでたしめでたし。

おしまい

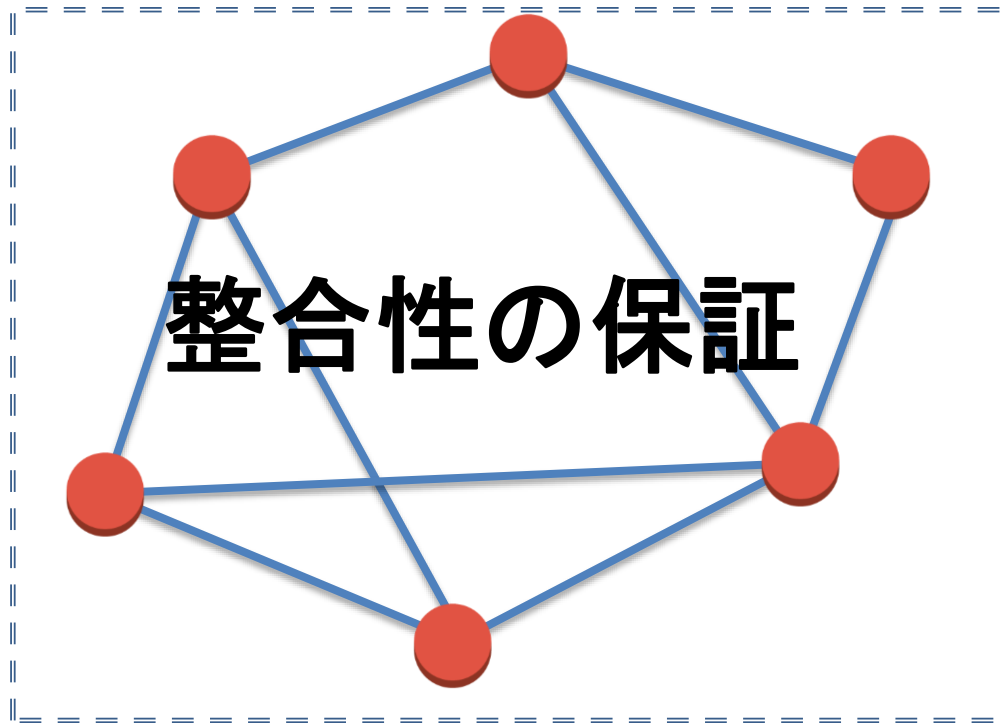
※面倒くさいのでここでシナリオは打ち切りです

鬼門の入口

ここからはNightmareモードです。

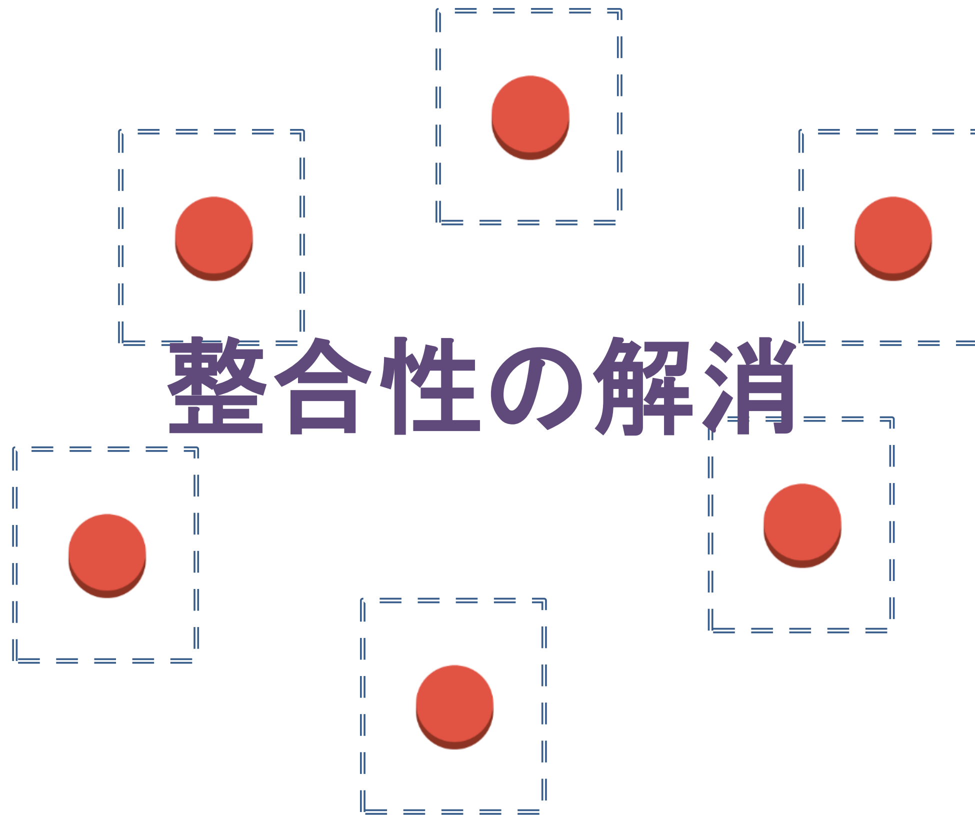
リレーショナルDBの限界

レコード同士の整合性を保証する代償として、複数のノード上で処理を分散できないという制約を受けている。



分散データベースの特徴

レコード同士の整合性を解消し、複数のノードで処理を分担できるようにしたのが分散DB。



整合性保証を失うということ

要するに
「トランザクション」
が使えなくなる

トランザクションが使えないということ

同時に２つ以上のレコードを
整合性をたもったまま
更新することができない

トランザクションがないとこうなる

100ゴールドする薬草を買います。

所持金	1,000 G
薬草	0 個

トランザクションがないとこうなる

所持金を -100 します。

所持金	900 G	-100
薬草	0 個	

トランザクションがないとこうなる

薬草を +1 します。

所持金	900 G
薬草	1 個

+1

トランザクションがないとこうなる

・・・がしかし、サーバー障害が発生して更新に失敗してしまいました。

所持金	900 G
薬草	0 個

もし整合性保証があれば・・・

ロールバックしてしまえば所持金も元に戻るのでユーザーの被害はない。つまり、

ALL or Nothingが保証されている

ではどうするのか？

アプリケーション側で
トランザクションを実装する

そりゃ鬼門と言われても仕方がないですね

トランザクションの構図(RDBMS)

アプリケーション

トランザクション

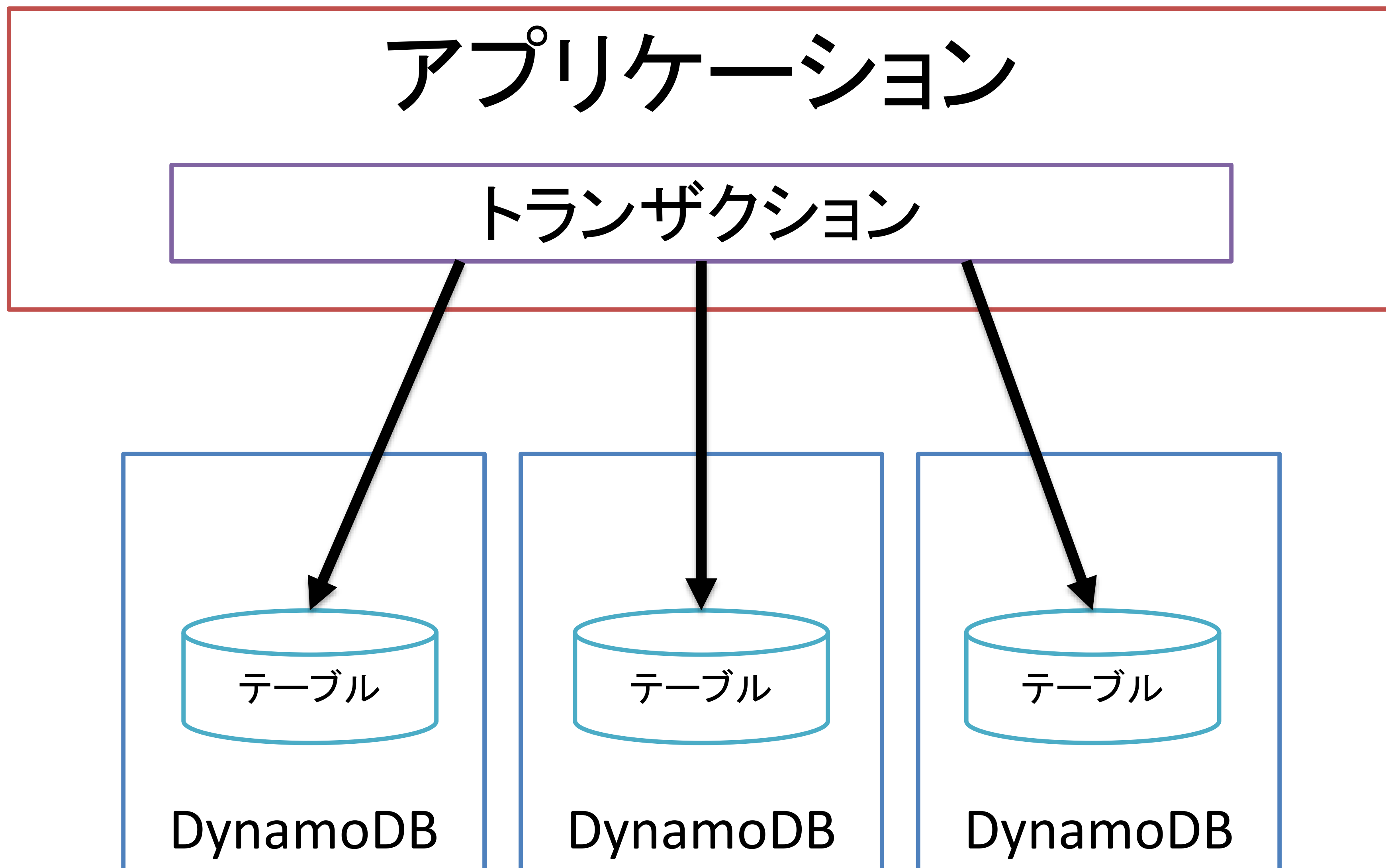
テーブル

テーブル

テーブル

MySQL

トランザクションの構図(DynamoDB)



トランザクションの作り方

・更新処理の開始から完了までを1つのトランザクションと捉える。

・各レコードの更新には**楽観的ロック**を用いる。

・全ての更新処理に**冪等性**を持たせる。

・処理の途中で失敗したら最初からやりなおす。

・結果が収束するまで何度もやりなおす。

用語解説

楽観的ロック

【意味】読み込んだレコードを更新するとき、他の並行プロセスによって変更がされていないことを期待して更新をする方式。

並列性を高めるためにとても重要な概念

用語解説

CAS操作

【意味】Compare and Swapの略。更新対象のレコードの状態が期待した状態のときのみ更新を実行し、そうでない場合は何もしない操作。

楽観的ロックに必要な概念

CAS操作をSQLで表すと

UPDATE user

SET status=1,updated_at=NOW()

WHERE id=100 AND status=0

※初期状態はstatus=0とする。

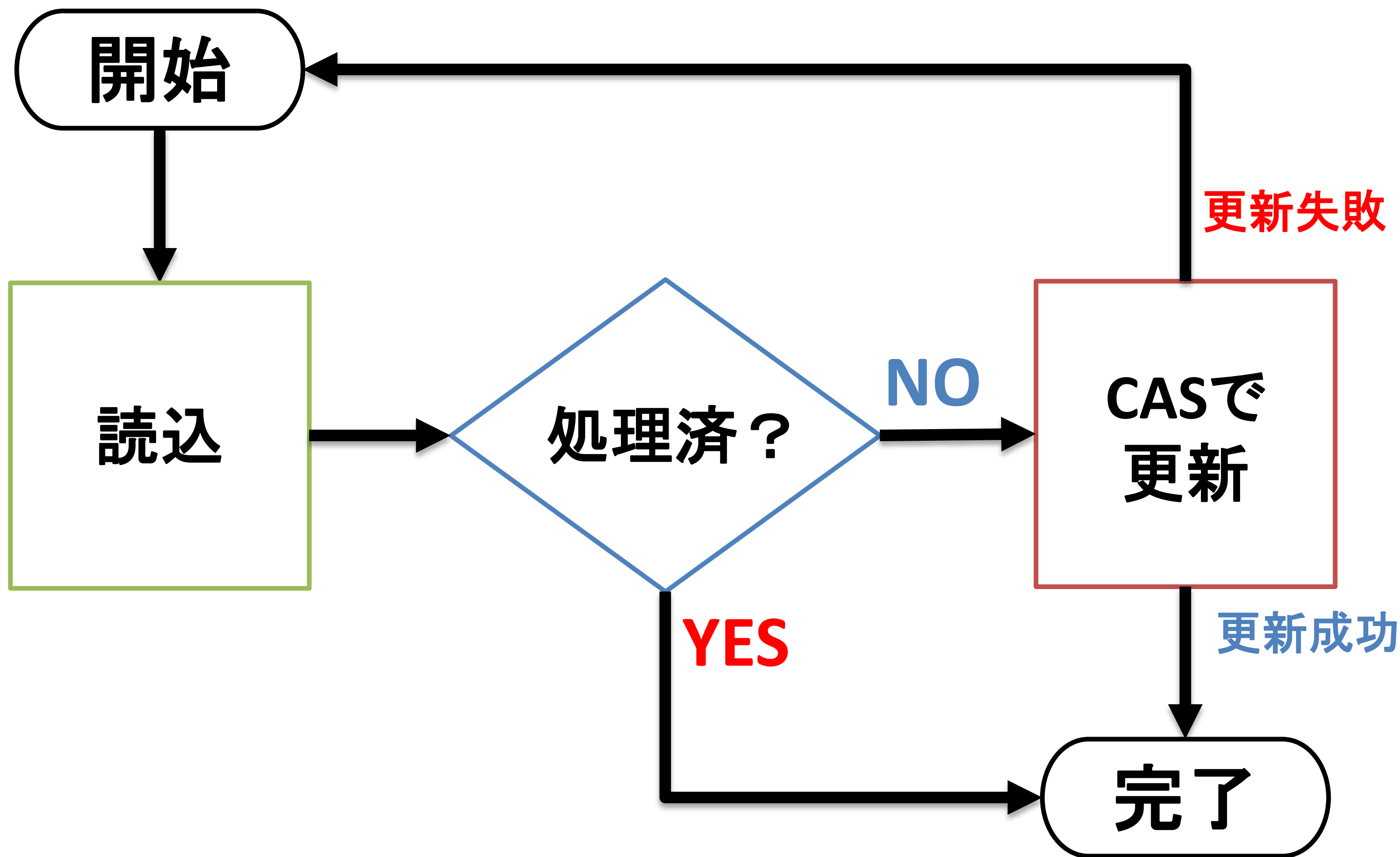
用語解説

冪等性

【意味】ある操作を1回行っても複数回行っても結果が同じであること。

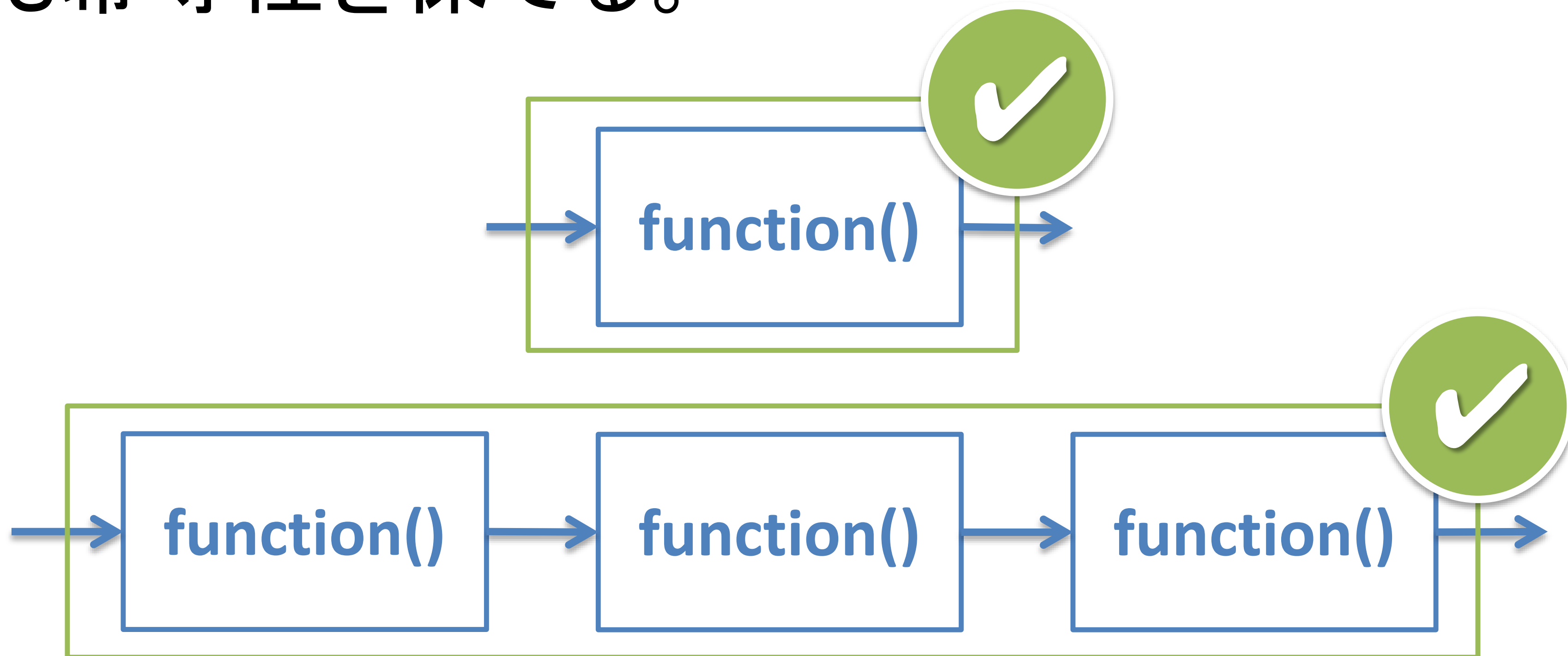
整合性を確保するためにとても重要な概念

冪等性のある処理の作り方



冪等性のある処理の作り方

冪等性の確保された処理はいくつ連結しても冪等性を保てる。



実装例

レコードを準備

更新依頼書
ID:123

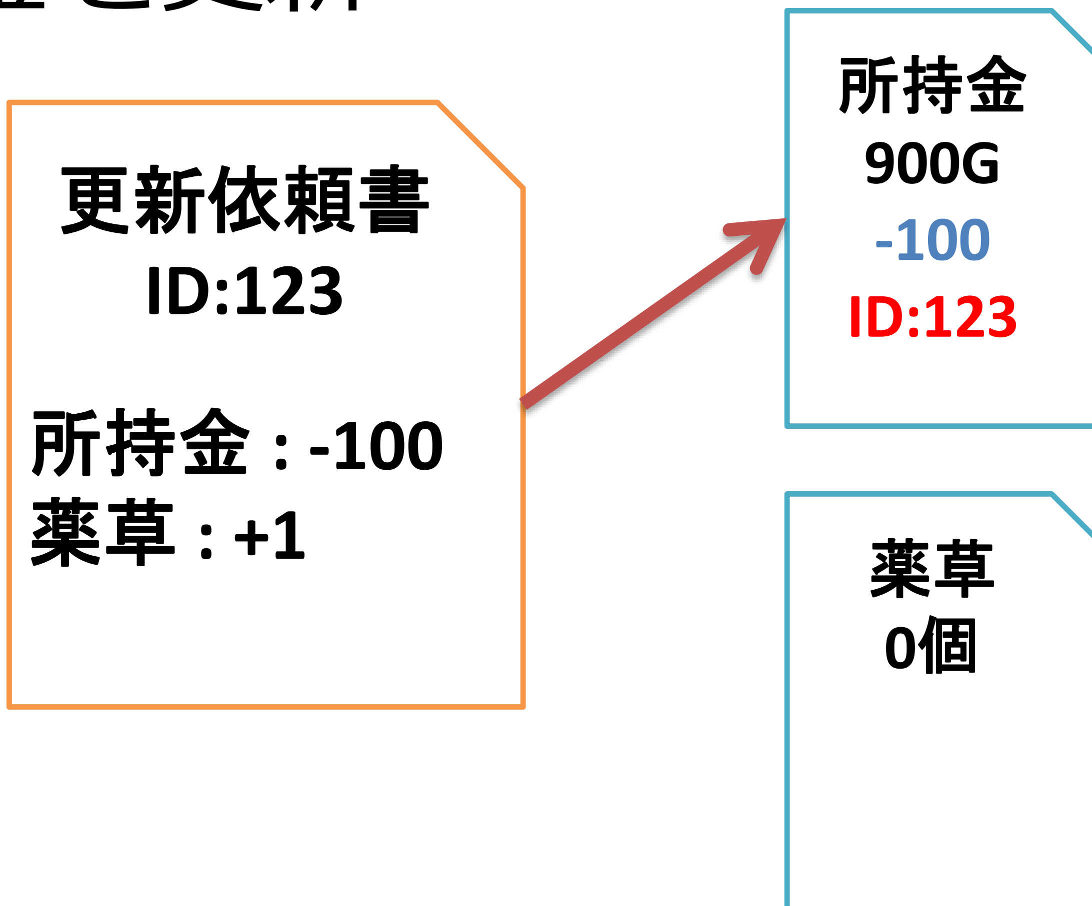
所持金 : -100
薬草 : +1

所持金
1,000G

薬草
0個

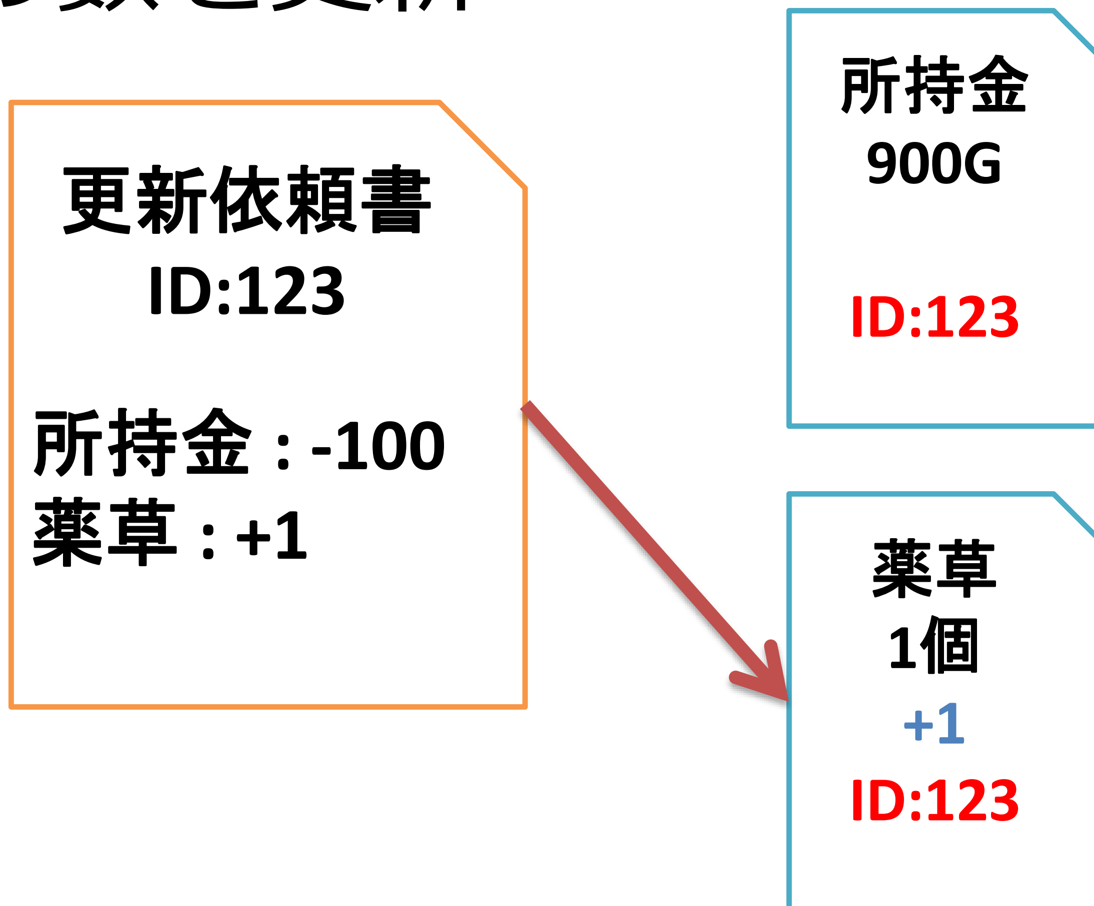
実装例

所持金を更新



実装例

薬草の数を更新



実装例

完了済みにする

更新依頼書
ID:123

所持金 : -100
薬草 : +1
状態 : 完了

所持金
900G

ID:123

薬草
1個

ID:123

実装例

掃除して完了

更新依頼書
ID:123

所持金 : -100
薬草 : +1
状態 : 完了

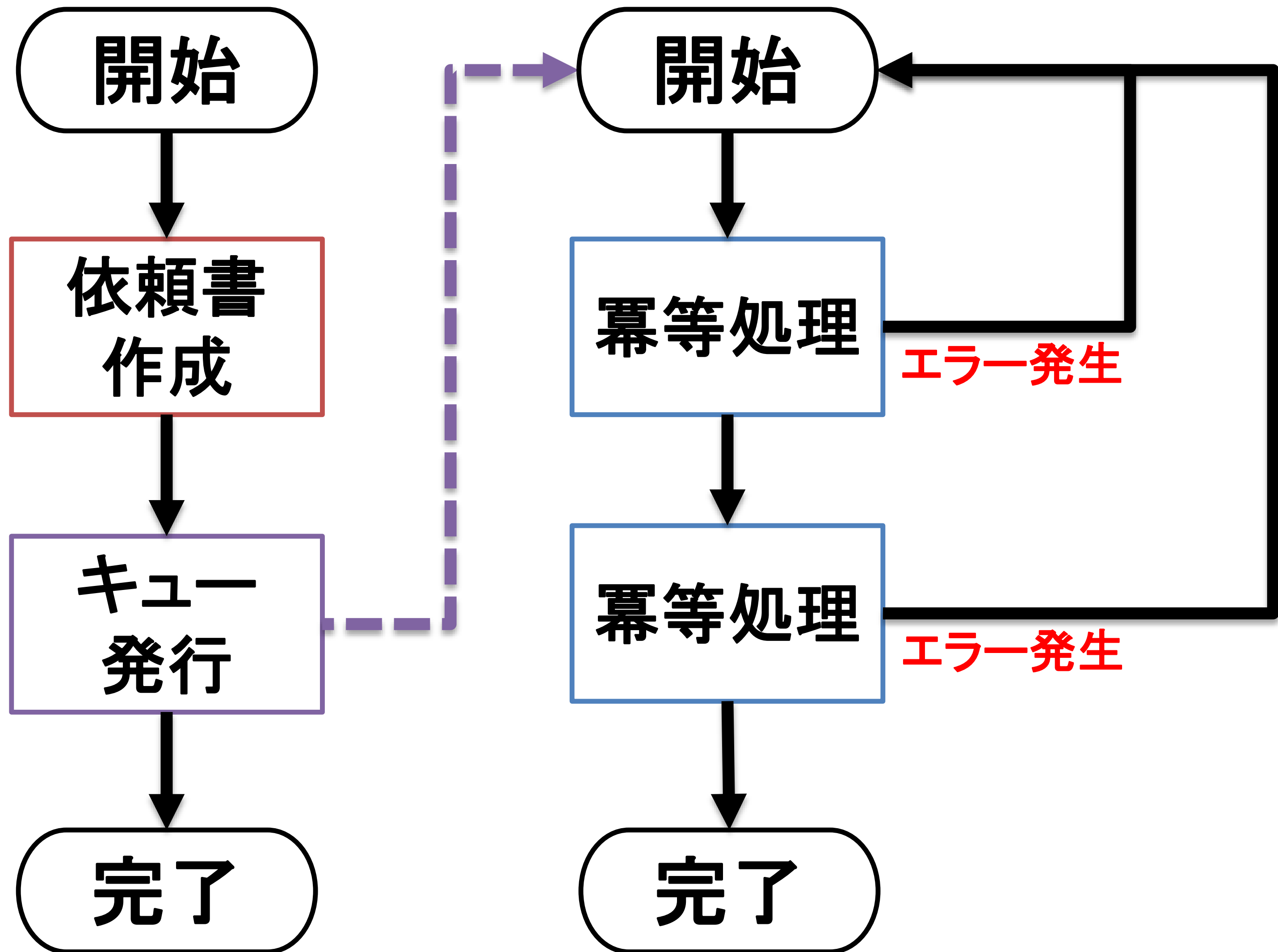
所持金
900G

薬草
1個

SQSと組み合わせて使う

1. 依頼書をレコードとして作る
2. SQSへ依頼書IDが書かれたメッセージを発行
3. バックグラウンドでSQSからメッセージを受け取り、結果が収束するまで何度も実行する。

全体フローチャート

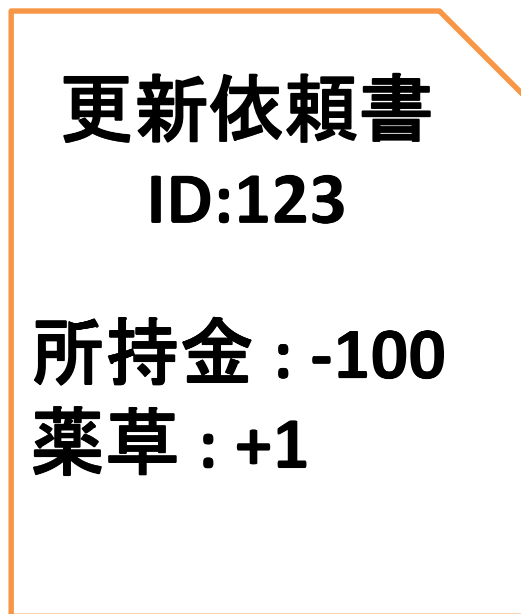


ロールバック処理

レコードを準備

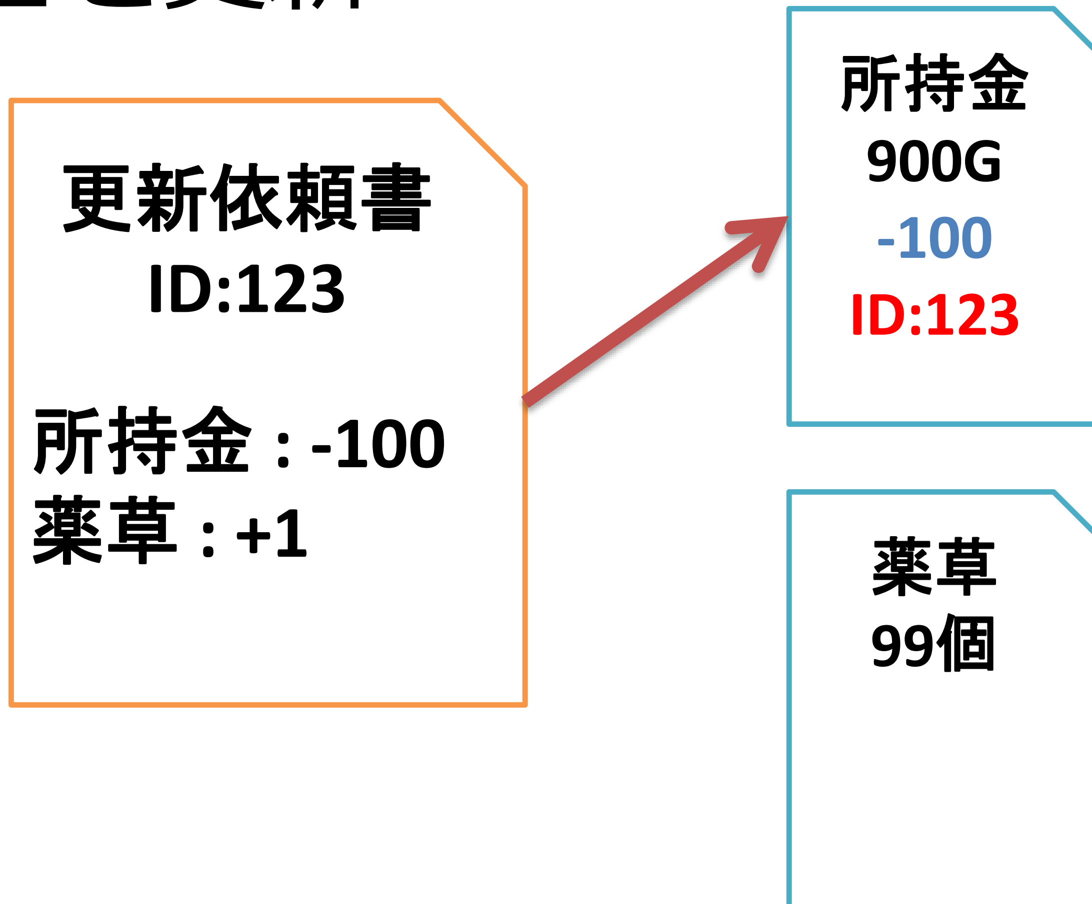
ロールバック処理

別の並行処理が邪魔をする



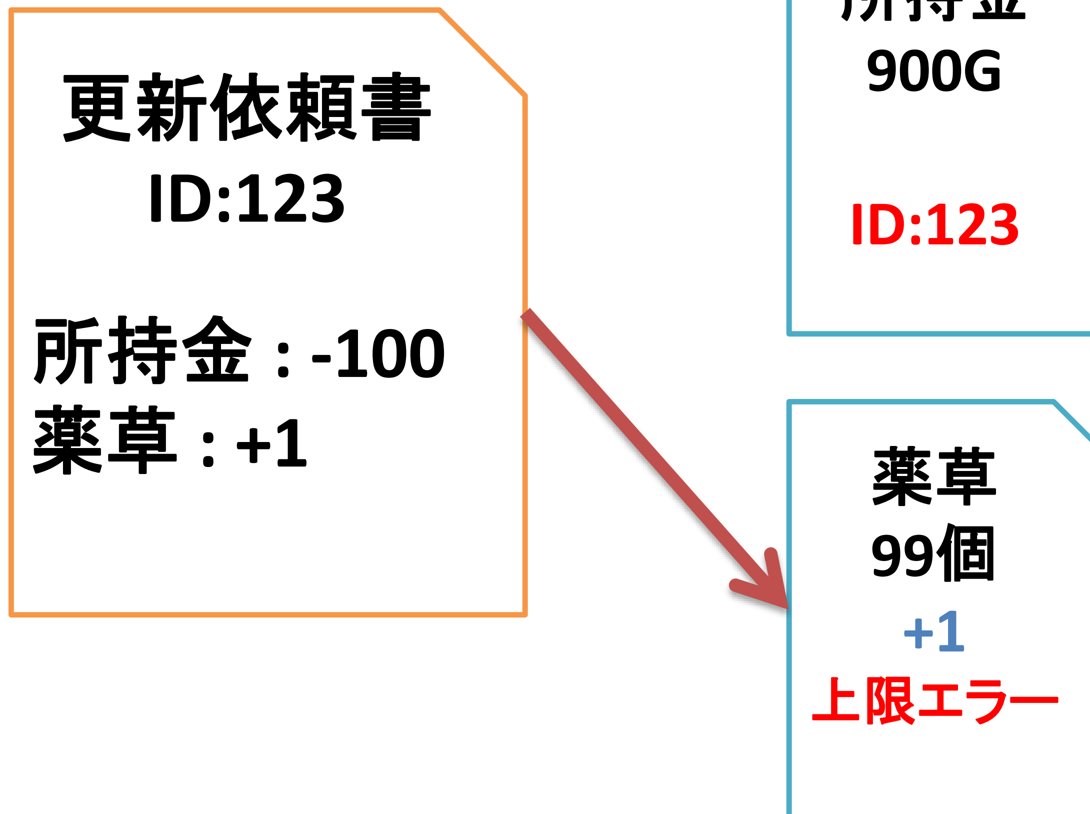
ロールバック処理

所持金を更新



ロールバック処理

薬草の数を更新



ロールバック処理

失敗済みにする

更新依頼書
ID:123

所持金 : -100
薬草 : +1
状態 : 失敗

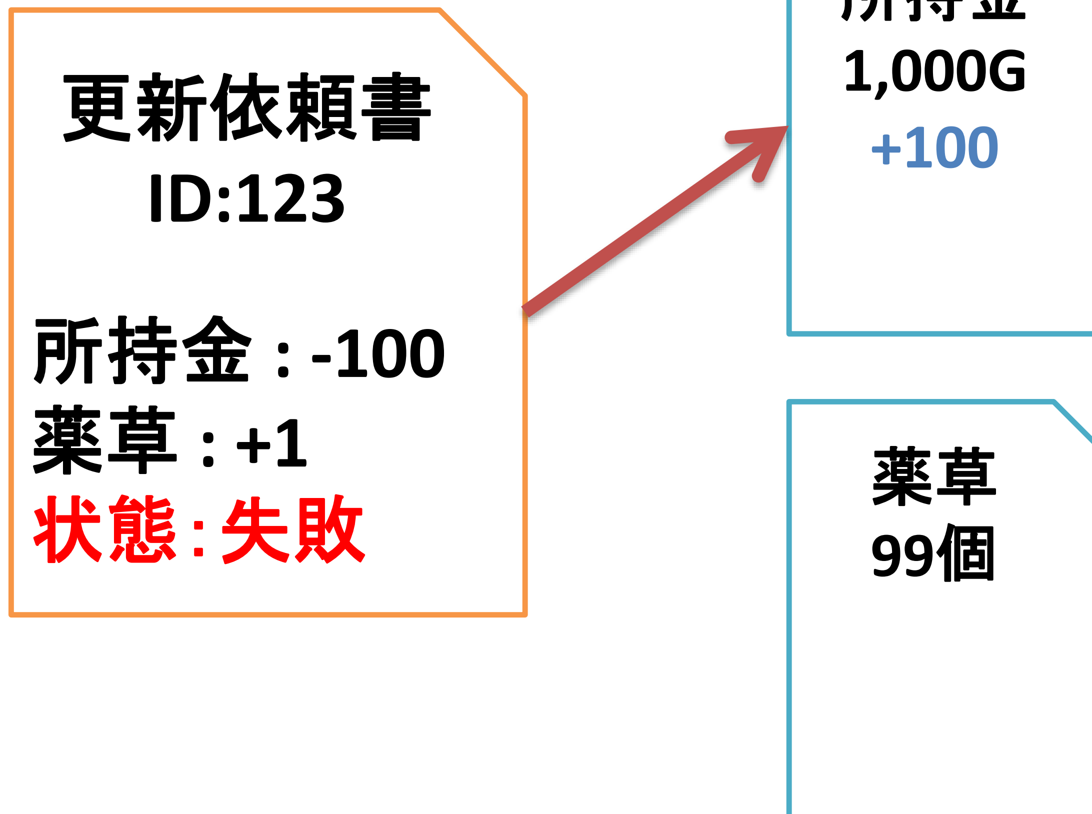
所持金
900G

ID:123

薬草
99個

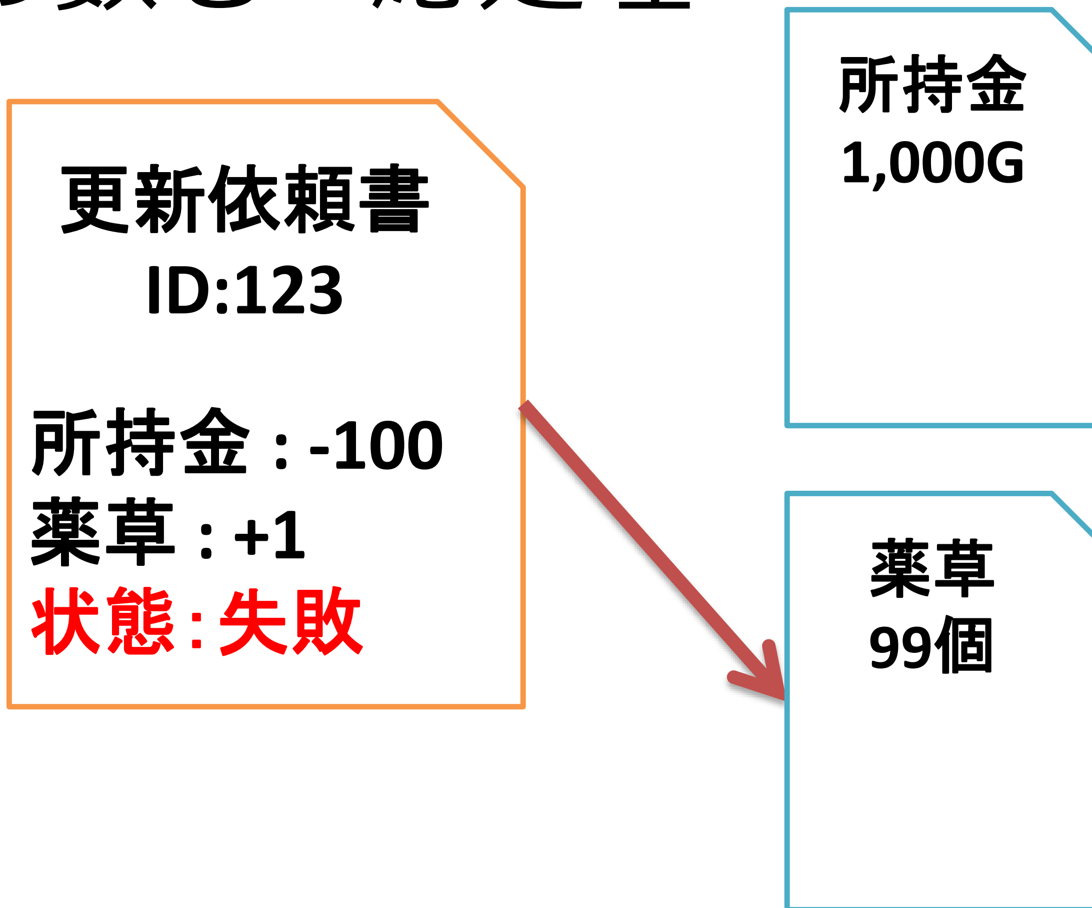
ロールバック処理

所持金を戻す



ロールバック処理

薬草の数も一応処理



ロールバック処理

完了（状態収束）

更新依頼書
ID:123

所持金 : -100
薬草 : +1
状態 : 失敗

所持金
1,000G

薬草
99個

まとめ

分散DBをメインDBとして使う場合、トランザクションの再実装をしなければいけないので大変。

しっかりフレームワークを組んでから挑むことを強く推奨。

質問タイム

お疲れ様でした