技術研修～データ設計～

本日のゴール

- データ設計思想を知る
- 運用に耐えうるデータ設計ができるようになる

ゲームのデータ設計で扱う主なデータ

データ設計をする際に考えるものは大きく下記の二つ

- マスタデータ
- ユーザデータ

マスタデータ

システムの設定情報が入ったデータ

使用する上で最初から存在しないと動かないようなデータ

- 次のようなデータが格納される
  - ゲームデザイン
  - レベルデザイン
  - システムの挙動のためのデータ
  - キャラクターの名前、パラメータなど

マスタデータ

特徴

- データの管理にはDBをもちいるが、使用する際はデータをサーバー用とクライアント用で別々にまとめたものをそれぞれに配布して、**起動時にメモリ上に展開**して利用されることが多い
- メモリ上にある場合が多いため、主キーなどで取得できる場合マスターの検索コストを気にする必要が少ない
    - 外部データストアにある場合は、気にする必要が大いにある

- 直接DBを参照しなくても良いため、テーブル変更時のマイグレーションを考慮する必要が少ない
    - ユーザデータより少ないだけで、運用中は基本考慮。開発中に DB単位でのマイグレーションは考慮しない

余談：サービスによっては、アプリのメモリ上以外にマスタデータを持っているものもあるらしい

マイグレーションとは

マイグレーションとは、DBに保存されているデータを保持したまま、変更を加えていくこと

例：新規テーブル作成、既存テーブルへのカラムの追加・変更など

ユーザデータ

ユーザの状態を表すデータ。サービス運営上、最も大事な資産

- 次のようなデータが格納される
  - ユーザに紐づく情報
  - ユーザが取得したキャラクタ情報
  - ユーザごとのシステム情報など
    - ログイン日時とか
    - 課金情報、有償・無償石情報

ユーザデータ

特徴

- ユーザーを特定できるuser_idをもつ
  - ゲームでは、ユーザは基本自身のユーザデータしか追加・更新・削除を行わない
- ユーザーデータのテーブルには大量のデータが保存されており、テーブル変更時のマイグレーションを考慮する必要がある
  - 破壊的な定義の追加・変更を行う場合メンテに入れる必要があったりする
- 基本的には後方互換性を保つ必要があり、カラムの追加やテーブルの追加などしか行わない
  - 後方互換を保っていない場合、アプリが公開されるまでメンテに入れるなどの対応が必要
  - 変更したい場合は、カラムを追加後に参照を外してから削除するなどの対応が必要
  - カラムの削除とかは強制アプデとかが必要になってくることも

ユーザデータ

基本的なアプリボット（SGEも？）でのゲームデータの共有は

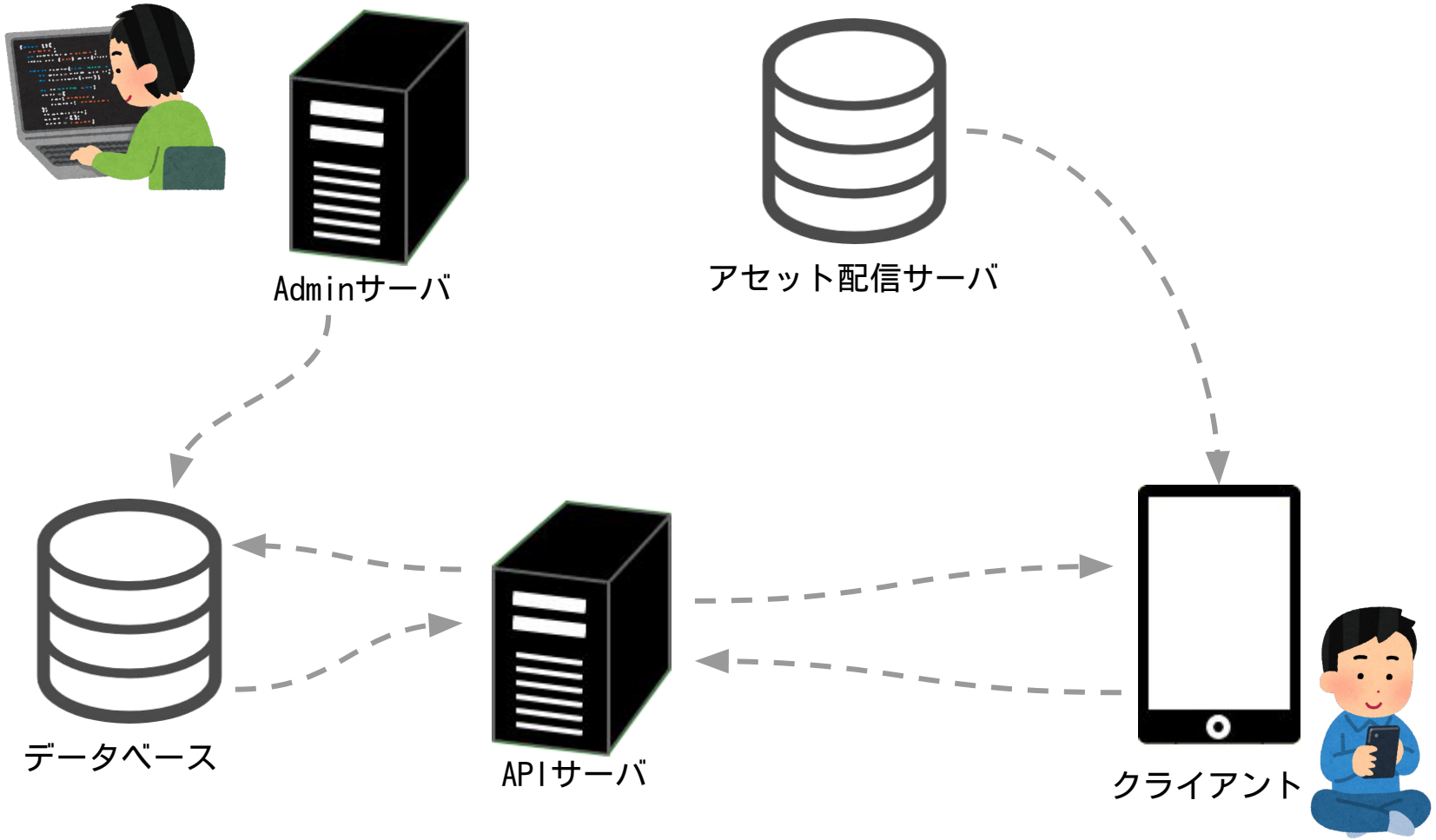
タイトルAPI（ログインっぽいAPI）などの起動時に叩かれるAPIでユーザ情報を全取得

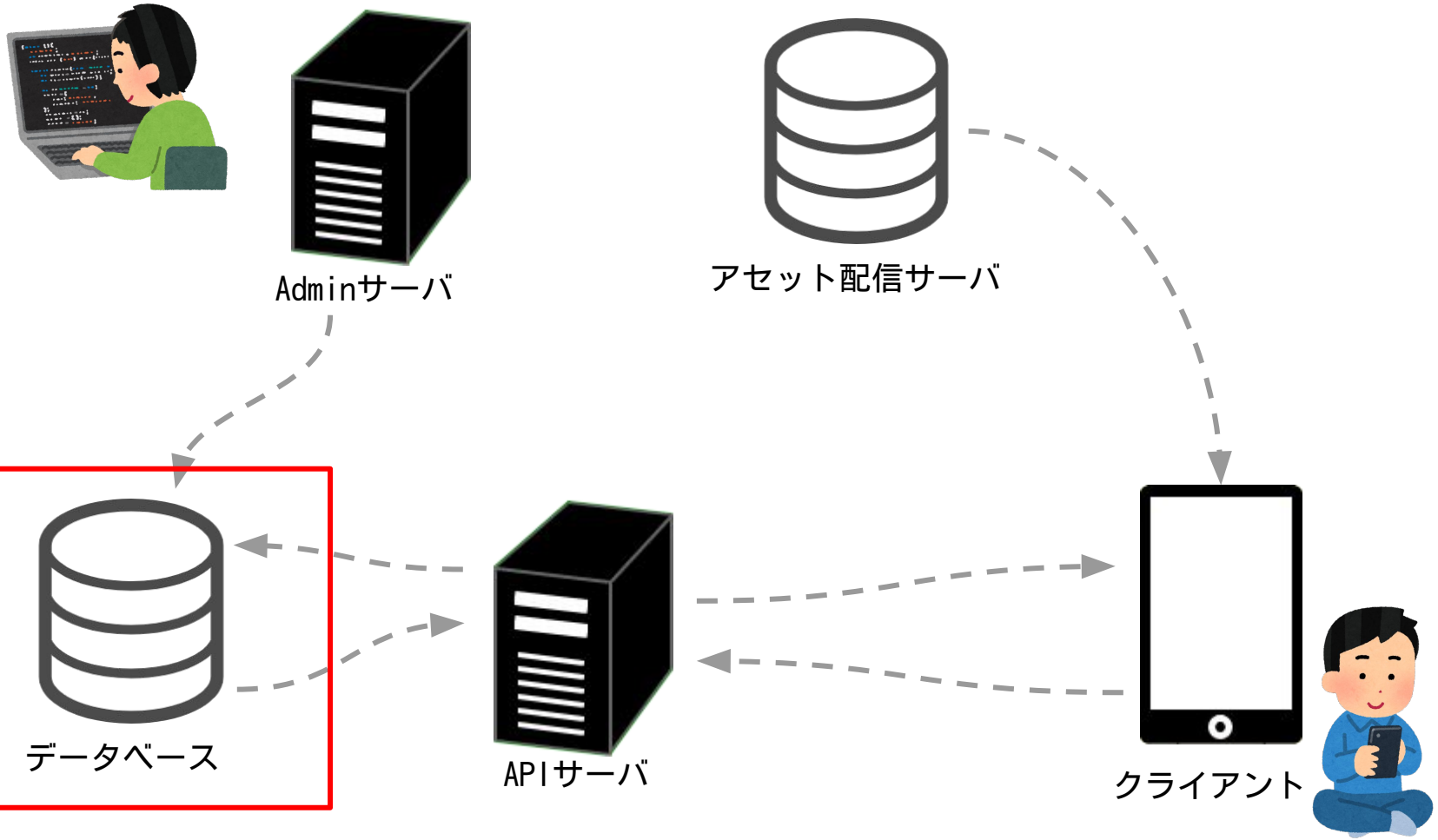
ログイン後の共有は、更新APIが叩かれるたびにユーザデータの更新差分のみ返却し、クライアントが更新差分をもとにデータを保持するような形で共有する

※ゲームは基本更新APIしかない

※ブラウザゲーとかは都度通信してデータ取得していることが多い

これらのデータは...


Adminサーバ
アセット配信サーバ
データベース
APIサーバ
クライアント


Adminサーバ
アセット配信サーバ
データベース
APIサーバ
クライアント

データ管理

データ管理に使用するデータベース

・MySQL（**RDBMS**）

・Redis

データ管理

データ管理に使用するデータベース

・MySQL（**RDBMS**）

・Redis

データ管理

RDBMS（リレーショナルデータベースマネジメントシステム）

大事なこと

・ACID特性など

・**正規化** ⇦ データ設計で非常に大事な考え方

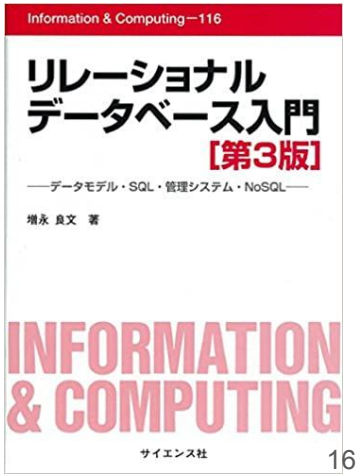
データ管理

正規化とは

データの重複をなくし整合的にデータを取り扱えるようにデータベースを設計すること

引用：https://oss-db.jp/dojo/dojo_info_04

詳しくは...こちらの本で読んでみてください



データ管理

受注テーブル

id	user_id	product_code	product_category	product_category_name	product_name	count	product_code	product_category	product_name	product_category_name	count
1	1	111	1	お菓子	チョコ	1	222	2	バナナ	果物	2
2	2	222	2	果物	バナナ	1					
3	3	333	2	果物	リンゴ	1					

第一正規化

id	user_id	product_code	product_category	product_category_name	product_name	count
1	1	111	1	お菓子	チョコ	1
2	2	222	2	果物	バナナ	1
3	3	333	2	果物	リンゴ	1
1	1	222	2	果物	バナナ	2

第二正規化

受注テーブル

id	user_id	product_code	count
1	1	111	1
2	2	222	1
3	3	333	1
1	1	222	2

商品テーブル

product_code	product_categor	product_category_name	product_name
111	1	お菓子	チョコ
222	2	果物	バナナ
333	2	果物	リンゴ

第三正規化

受注テーブル

id	user_id	product_code	count
1	1	111	1
2	2	222	1
3	3	333	1
1	1	222	2

商品テーブル

product_code	product_categor	product_name
111	1	チョコ
222	2	バナナ
333	2	リンゴ

商品カテゴリテーブル

product_category	product_category_name
1	お菓子
2	果物

データ管理

受注テーブル

id	user_id	product_code	product_category	product_category_name	product_name	count	product_code	product_category	product_name	product_category_name	count
1	1	111	1	お菓子	チョコ	1	222	2	バナナ	果物	2
2	2	222	2	果物	バナナ	1					
3	3	333	2	果物	リンゴ	1					

第一正規化

id	user_id	product_code	product_category	product_category_name	product_name	count
1	1	111	1	お菓子	チョコ	1
2	2	222	2	果物	バナナ	1
3	3	333	2	果物	リンゴ	1
1	1	222	2	果物	バナナ	2

第二正規化

受注テーブル

id	user_id	product_code	count
1	1	111	1
2	2	222	1
3	3	333	1
1	1	222	2

商品テーブル

product_code	product_categor	product_category_name	product_name
111	1	お菓子	チョコ
222	2	果物	バナナ
333	2	果物	リンゴ

第三正規化

受注テーブル

id	user_id	product_code	count
1	1	111	1
2	2	222	1
3	3	333	1
1	1	222	2

商品テーブル

product_code	product_categor	product_name
111	1	チョコ
222	2	バナナ
333	2	リンゴ

商品カテゴリテーブル

product_category	product_category_name
1	お菓子
2	果物

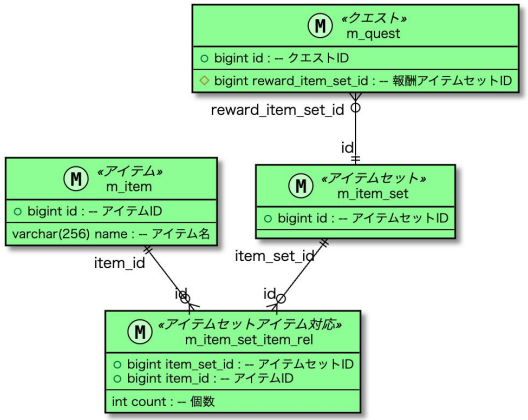
設計パターン

Relationテーブル

2つ以上のテーブル同士の**複数のレコード**を紐付けたい場合はRelテーブルを作成する。

このようなRelテーブルには、テーブル名のサフィックスに_relをつける。

複数のレコードが複数のグループIDに所属したい場合などに良く利用する。



アイテムをまとめて配るための集合マスタの例

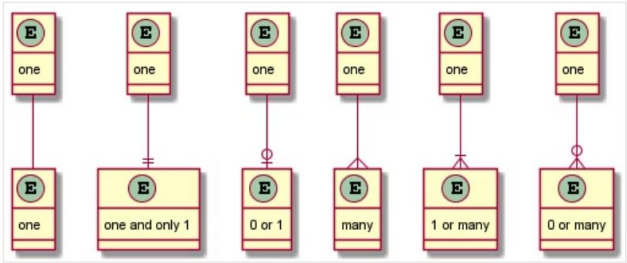
Relationテーブル

ER図の線の意味

カーディナリティ

```
------   :1
----||   :1 and only 1
----o|   :0 or 1
-----{   :many
----|{   :1 or more
----o{   :0 or many
```

描画結果

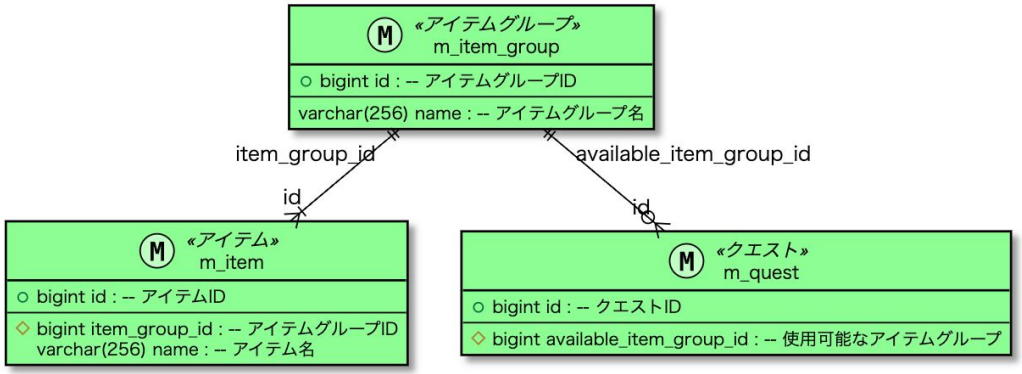


参考：https://qiita.com/Tachy_Pochy/items/752ef6e3d38e970378f0

Groupテーブル

複数のレコードに対して**1つのID**で指定したい場合はGroupテーブルを作成する。

このようなGroupテーブルには、テーブル名のサフィックスに_groupをつける。



アイテムにはグループが存在し、クエストで使用可能な
アイテムのグループが存在する例

命名規則

テーブル名

テーブル名は、関係性がわかりやすいように、枠組みが大きい順に命名することを推奨している。

また、マスターのテーブルかユーザーのテーブルかわかりやすいように prefixを付けることも推奨している。
(マスタ：master_、ユーザ：user_など)。

例：ログインボーナスのテーブルを作成する場合、下記のように login_bonus -> event or normalといった命名になる。

event_login_bonus -> master_login_bonus_event

user_event_login_bonus -> user_login_bonus_event

normal_login_bonus -> master_login_bonus_normal

user_normal_login_bonus -> user_login_bonus_normal

命名規則

カラム名

下記のように自身のテーブルに関する情報の場合テーブル名を省略することを推奨している。

character.character_id -> character.id、character.character_name -> character.name

ただし、自身以外のテーブルに関する情報を参照する場合は、テーブル名を prefixにつけるようにします。

例えばcharacterテーブルからskillテーブルのidを参照する場合は下記のようになります。

character.skill_id

さらに追加情報付与したい場合は、追加で prefixをつけて表現することも可能。

下記は見た目スキルを設定する例。

character.display_skill_id

設計方針・考え方

マスタデータ

マスタデータ設計での大事な設計の考え方

- なるべく重複データを作らない
- 運用中に変更があるものは、既存レコードの修正ではなく新規レコードの追加で対応する
- ユーザーデータに主キーになりうるマスタの場合、主キーは1つが好ましい
- 入力ミスが発生しにくいデータ設計・システム設計

マスタデータ

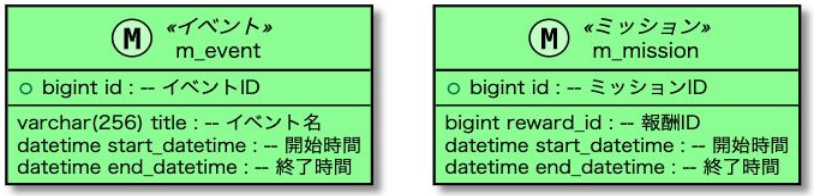
なるべく重複データ・定義を作らない

同じデータを重複して別テーブルに持つと、片方を変更した際にもう片方の**変更漏れ**が発生するなど、非常に不具合の原因になりやすい

基本マスタは人が運用するので、漏れが発生前提でデータ設計しましょう！

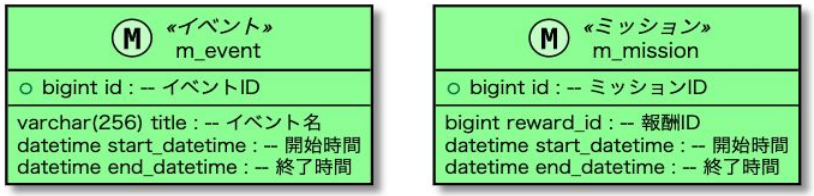
なるべく重複データを作らない

例：イベントを開催したい。ただ別仕様でミッションもあって、イベント開催中は特定のミッションを開催したい。



なるべく重複データを作らない

例：運用途中でリリースしているイベントの開催期間を延長したい！



なるべく重複データを作らない

例：運用途中でリリースしているイベントの開催期間を延長したい！

m_missionのend_datetimeの変更忘れてた...

なるべく重複データ作らない

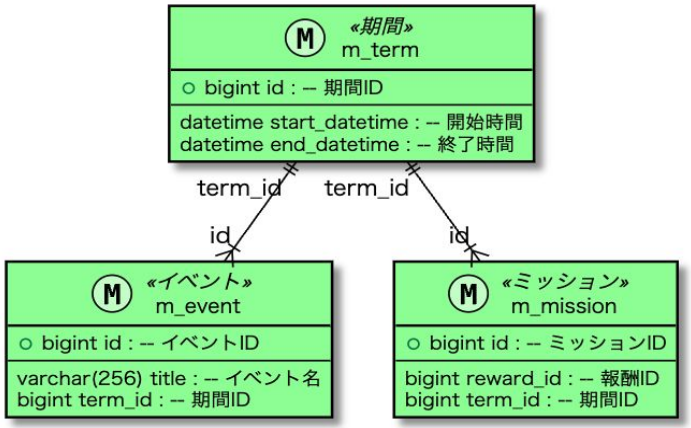
例：運用途中でリリースしているイベントの開催期間を延長したい！

m_missionのend_datetimeの変更忘れてた...

なるべく重複データを作らない

解決策：開催期間用マスタを別で作って、各テーブルで参照する形にする

　m_eventとm_missionが同一のterm_idを参照するような形でマスタを入力してもらって、終了期間を延ばす場合は、参照しているterm_idのレコードを変更する！



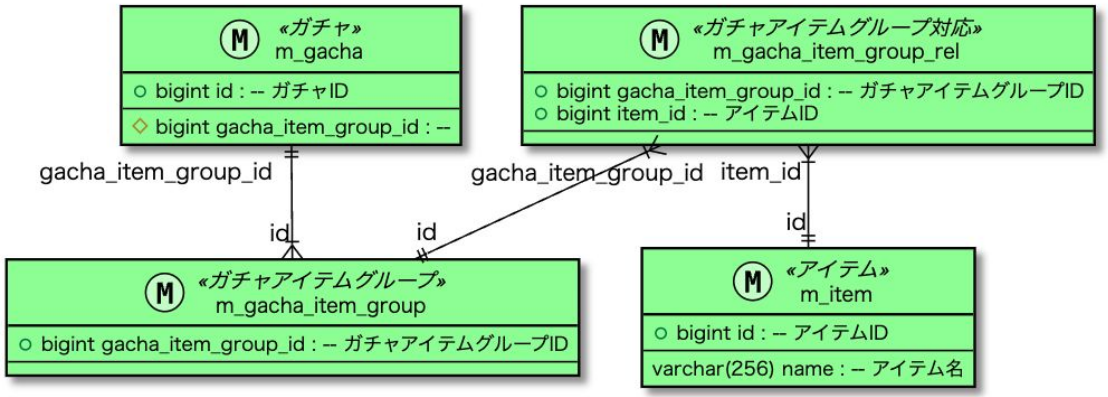
マスタデータ

運用中に変更があるものは、既存レコードの修正ではなく新規レコードの追加で対応する

- マスタに変更が必要な場合は、旧レコードの参照期間をマスタで持つ+新規レコード追加で対応できる仕組みで切り替えを行えるようにする
  - これをしないと、マスタのリリース時に変更が即時反映されてしまう
  - もちろん即時反映されて良いものもあるが、ガチャなどのユーザに即反映されると不味いものは基本この仕組みで実装する方が良い
- 基本運用中のマスタデータは消さない
  - 消すときは、**かなり慎重に削除**する
  - （ダークでは日本海外マスタ統合でマスタデータいくつか消したらしい？）
  - 既存のユーザデータで参照している IDなどが残っている可能性があり、不具合になる可能性が大きい

運用中に変更があるものは、既存レコードの修正ではなく新規レコードの追加で対応する

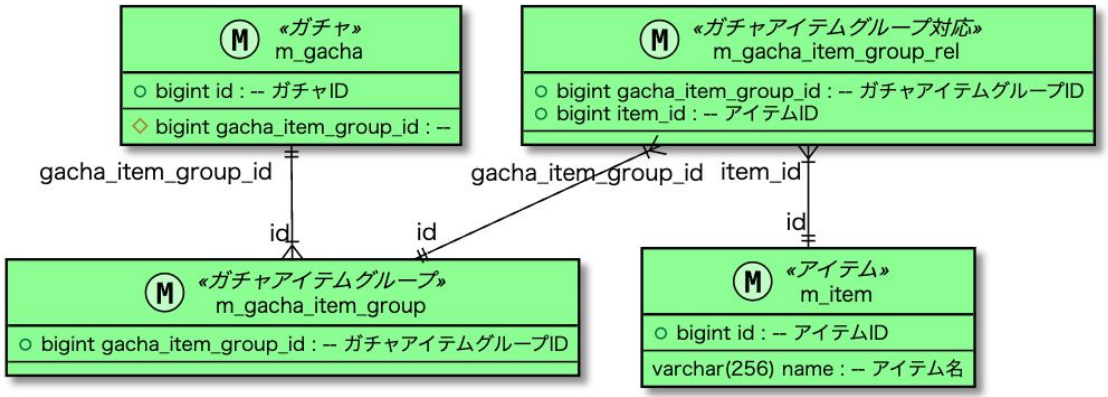
例：ガチャ台の抽選アイテムは運用中に追加などの変更などが行いたい！



運用中に変更があるものは、既存レコードの修正ではなく新規レコードの追加で対応する

例：ガチャ台の抽選アイテムは運用中に追加などの変更などが行いたい！

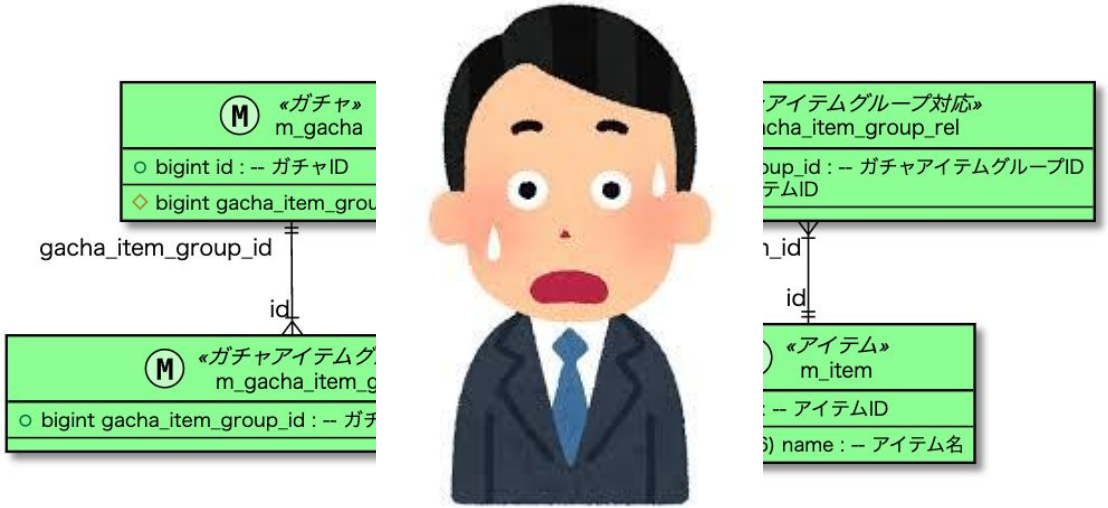
（ただ、反映日は一斉に切り替えたい）



運用中に変更があるものは、既存レコードの修正ではなく新規レコードの追加で対応する

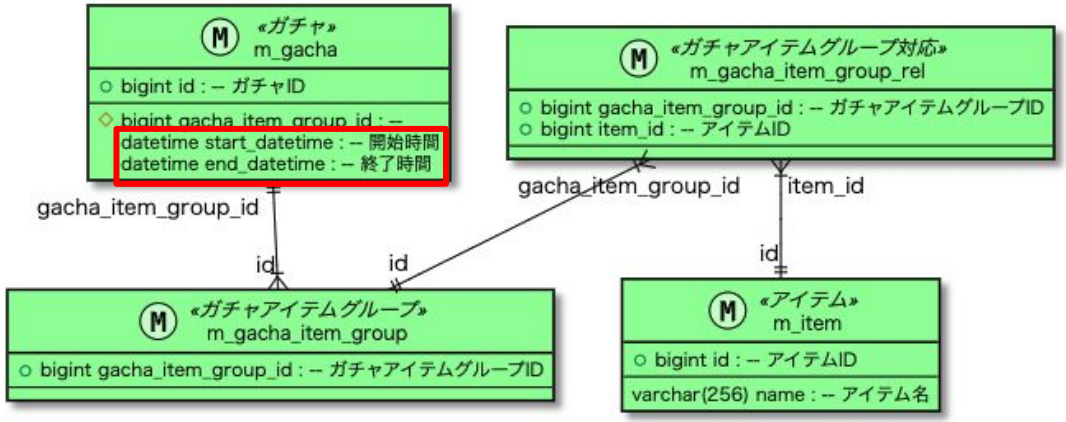
例：ガチャ台の抽選アイテムは運用中に追加などの変更などが行いたい！

（ただ、反映日は一斉に切り替えたい）



運用中に変更があるものは、既存レコードの修正ではなく新規レコードの追加で対応する

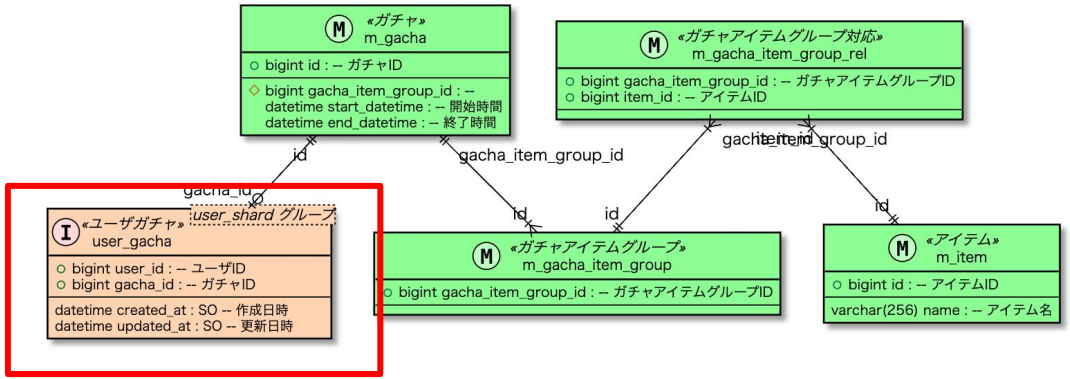
解決策：マスタに期間を持ち、一斉に入れ替えられるようにしておく



運用中に変更があるものは、既存レコードの修正ではなく新規レコードの追加で対応する

基本このやり方で行うと古くなったm_gachaのレコードが必要なくなるように思えるが、**基本消さない**

ユーザテーブルにm_gachaの削除対象のIDを参照しているものが存在する場合にバグになりやすい



マスタデータ

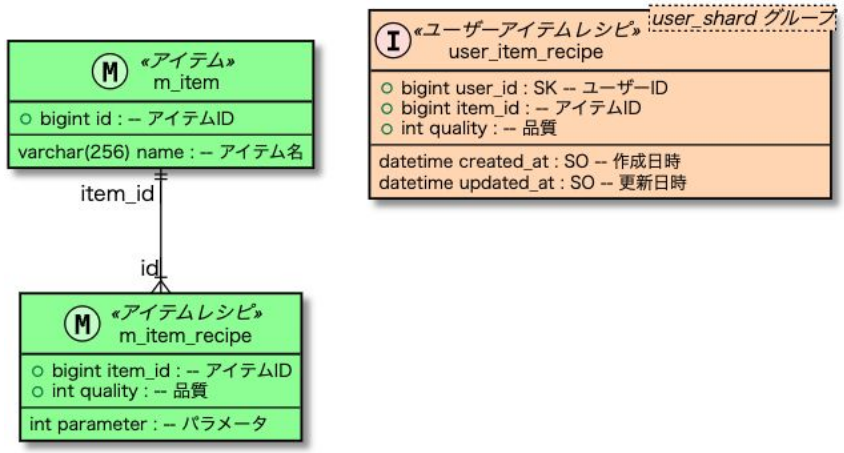
ユーザーデータに主キーになりうるマスタの場合、主キーは1つが好ましい

- 主キーが複数になるようなデータはサロゲートキーなどを発行して、一つになるように設計する
- 主キーの追加などの構成が変わる場合の変更コストは特に大きく、主キーを複数設定している場合のほうが、主キーを1つに制限しているものよりも変更に弱い場合が多い

※サロゲートキー：データベースのテーブルの主キーとして、自動割り当ての連続した通し番号のように、利用者や記録する対象とは直接関係のない人工的な値を用いること

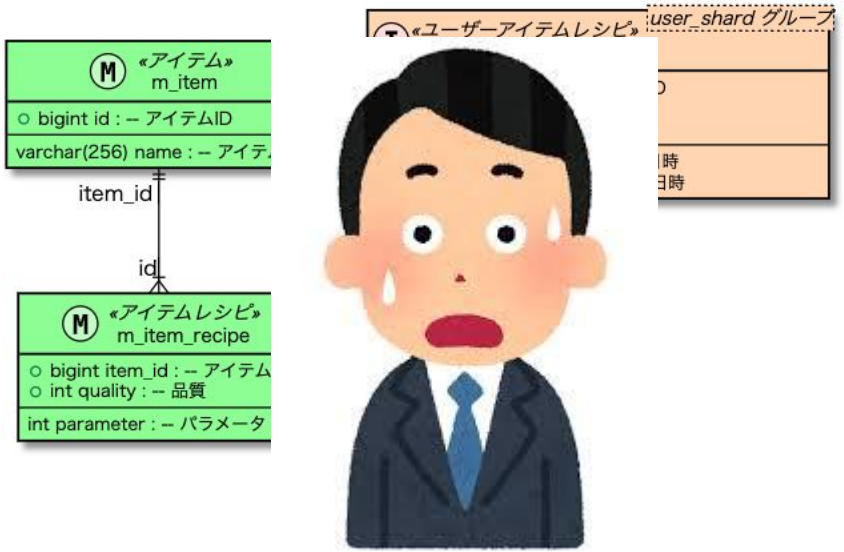
ユーザーデータに主キーになりうるマスタの場合、主キーは1つが好ましい

例：アイテムを生成できるレシピが存在する
アイテムレシピには、品質があり作成されるアイテムに影響する
レシピは、入手しているものしか使えない



ユーザーデータに主キーになりうるマスタの場合、主キーは1つが好ましい

例：アイテムを生成できるレシピが存在する
アイテムレシピには、品質があり作成されるアイテムに影響する
レシピは、入手しているものしか使えない



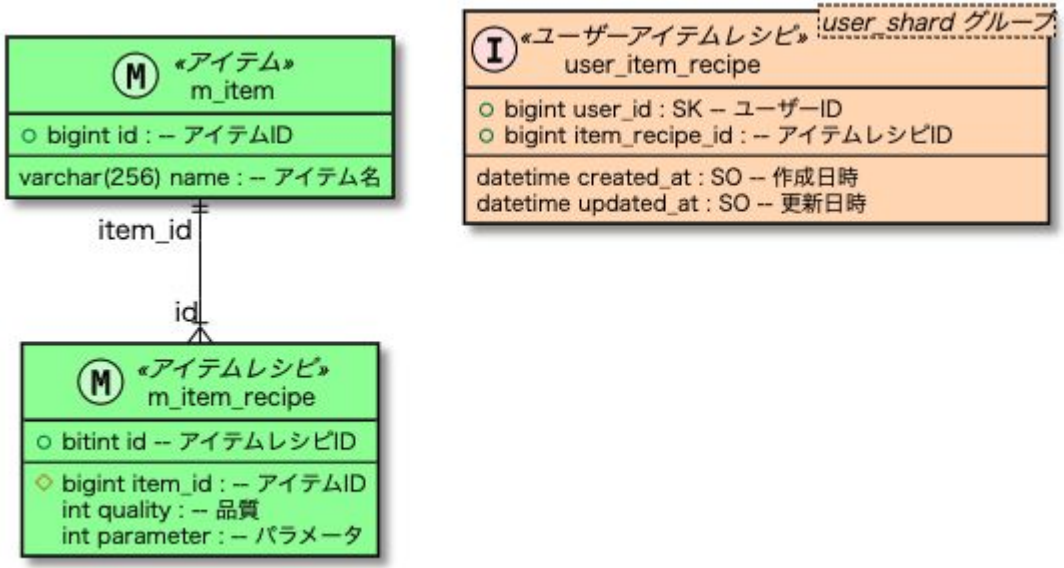
マスタデータ

このようなテーブル構造をもつと、API経由で特定のレシピを使用してアイテムを生成する際に、パラメータがitem_id、qualityの2つ必要になってくる（これは大きな影響ではないがまとめられるとスッキリ）

また、後からレシピ自体にユニーク性の高い属性を一つ追加したいといった場合に破綻しやすい

マスタデータ

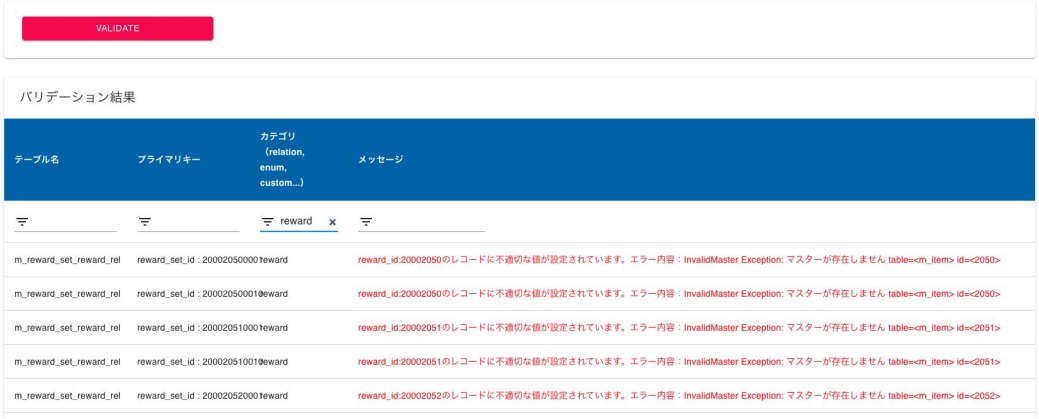
解決策:主キーを一つにする



マスタデータ

入力ミスが発生しにくいデータ設計・システム設計

- 複雑な設計を避け、できるだけシンプルな設計にする。
  - そもそも人間が運用するので、ミスは発生するものとして設計する
- 複雑な仕様の場合は、設計をシンプルにすることが難しい場合もある。
  - その場合、**自動入力**や**検証機能**のサポートなどが必要になってくる。



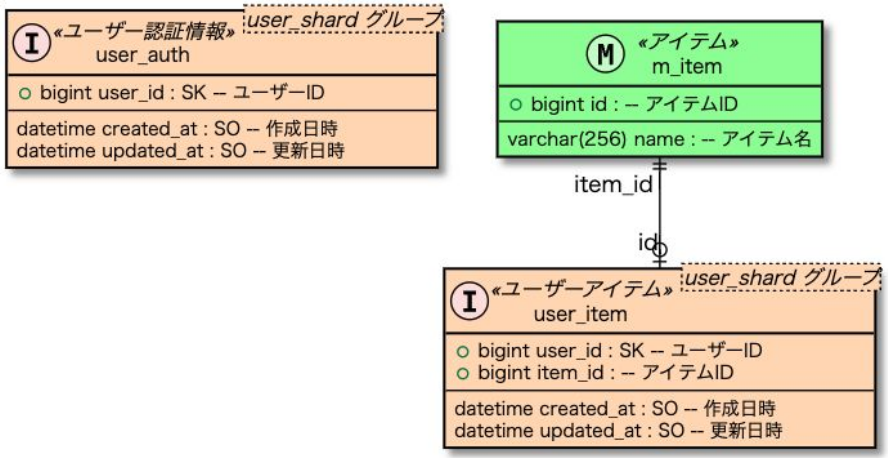
ユーザデータ

ユーザデータ設計での大事な設計の考え方

- 原則主キーはuser_idのみか、user_idとmaster_idの2つの主キーにする
- 主キーはauto_incrementをできる限り避けてmaster_idにする
- 自身のユーザーデータのみ更新する
- マスター側で参照できるデータはユーザーデータに保持しない
- レコードが存在する場合所持しているというような判定はさける
- 運用・仕様変更を常に意識する
- レコード数が増えすぎないように

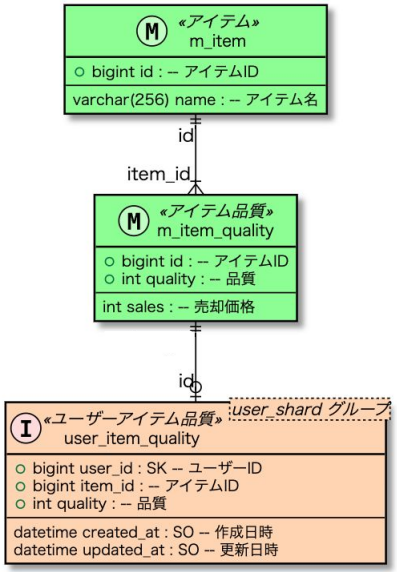
原則主キーはuser_idのみか、user_idとmaster_idの2つの主キーにする

- ユーザデータよりもマスタデータの方が変更に強いため、マスタに紐づく場合はmaster_idを主キーに持つ
- どちらかというとサーバのDBの事情が大きい



原則主キーはuser_idのみか、user_idとmaster_idの2つの主キーにする

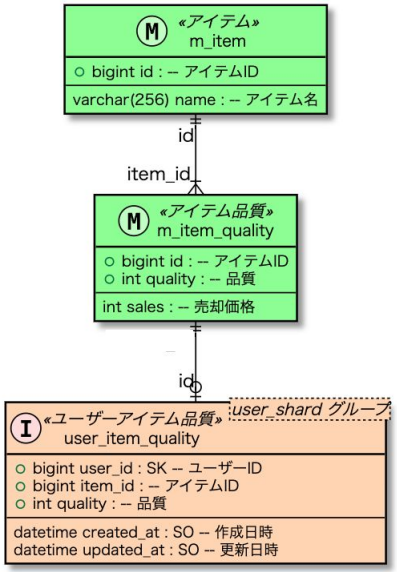
例：アイテムが品質ごとに存在する場合



原則主キーはuser_idのみか、user_idとmaster_idの2つの主キーにする

例：アイテムが品質ごとに存在する場合

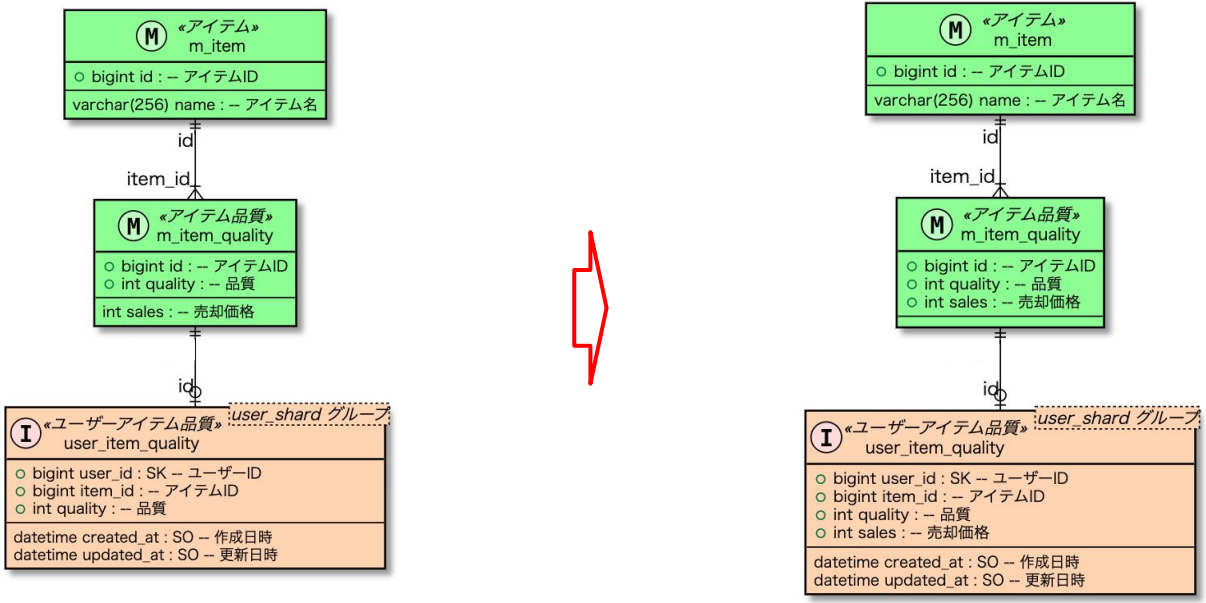
運用中、品質ごとだけじゃなく、価格によっても分けたい…



原則主キーはuser_idのみか、user_idとmaster_idの2つの主キーにする

例：アイテムが品質ごとに存在する場合

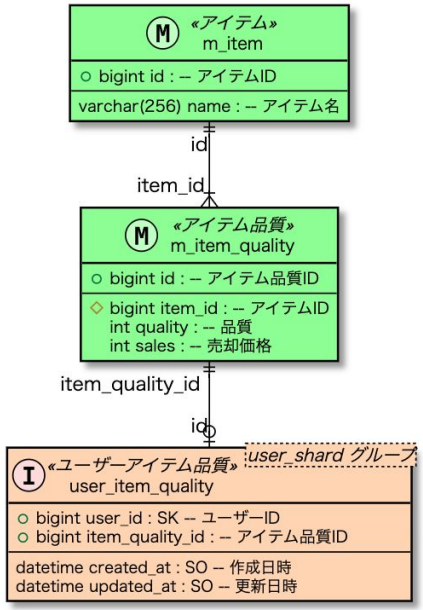
運用中、品質ごとだけじゃなく、価格によっても分けたい…



原則主キーはuser_idのみか、user_idとmaster_idの2つの主キーにする

このような変更が入ると、ユーザテーブルの主キーも変更する必要が出てくる

ユーザテーブルの主キー変更は、割と**重い**マイグレーションが必要になる

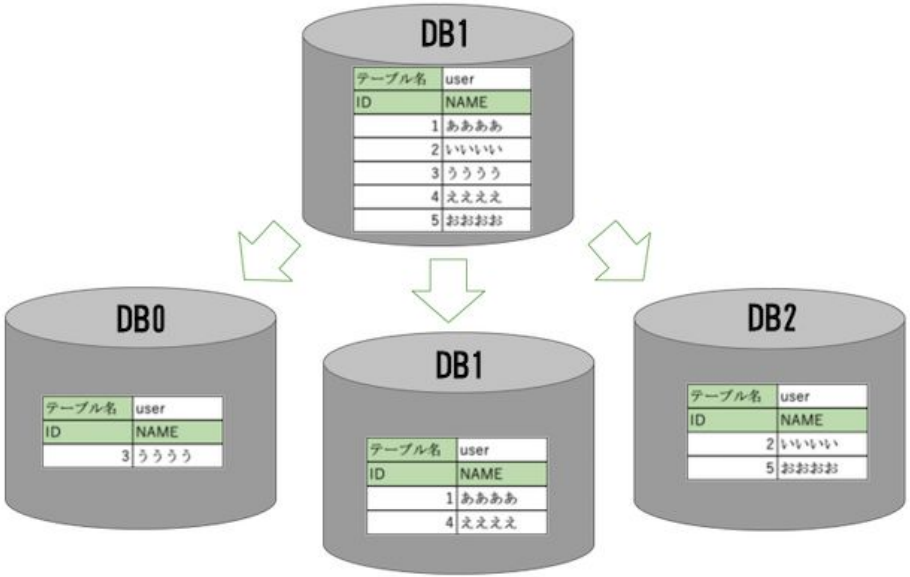


主キーはauto_incrementをできる限り避けてmaster_idにする

- 大規模なサービス開発では、AUTO_INCREMENTを避ける
- master_idでレコードが管理できる状態にできると、masterのカラム数に応じて最大のレコード数がきまるため、システム的にレコードが制限なく増加することを防げる

主キーはauto_incrementをできる限り避けてmaster_idにする

シャーディングとは、データベースの負荷分散の手法の一つで、一つのテーブルを複数の物理コンピュータに分割して記録する方式。



主キーはauto_incrementをできる限り避けてmaster_idにする

AUTO_INCREMENTとは、MySQLで列に設定できる属性の一つで、値を自動採番することができるようにするもの。各行に自動的に一意の値を割り当てることができる。

小さい規模のシステムとかではよく使う

MySQLのAUTO_INCREMENTは最大値（9223372036854775807）を超えるとリセットされないらしく、新規レコードの挿入ができなくなるらしい？

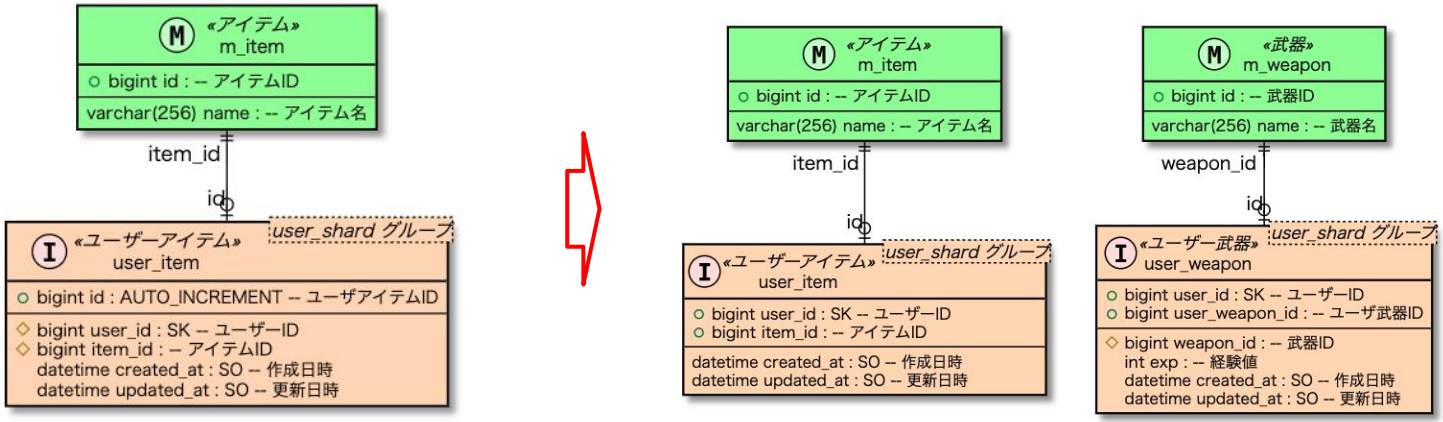
主キーはauto_incrementをできる限り避けてmaster_idにする

- 大規模なサービス開発では、AUTO_INCREMENTを避ける
- master_idでレコードが管理できる状態にできると、masterのカラム数に応じて最大のレコード数がきまるため、システム的にレコードが制限なく増加することを防げる

主キーはauto_incrementをできる限り避けてmaster_idにする

例：ユーザはアイテムごとにアイテムを所持する（アイテム毎に変化するものはなし）
ユーザは武器を複数所持できる。ただ武器毎に強化が行える

user_weapon_idはランダムかつユニークな値をアプリケーション側で生成する



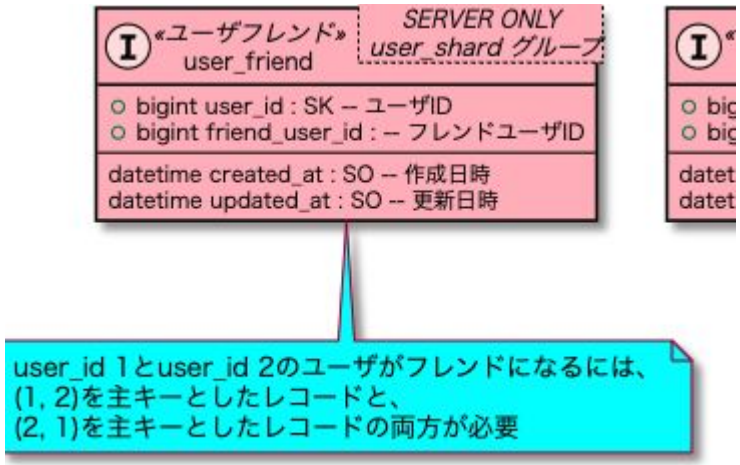
ユーザデータ

自身のユーザーデータのみ更新する

- ゲームにおいてフレンド機能など自身以外のデータを更新したい場合がありますが、その際でも**できるだけ他人のデータを更新しない**ように設計する。
- これは仕様による影響も大きいため、エンジニアだけでなく仕様決定者も巻き込んで調整する必要がある場合もある。
  (例えばウマ娘のフォロー・フォロワーのような仕様に落とし込んだりすることで、自分のデータの更新だけにできる)
- どうしても自身以外のデータを更新したい場合は**ロック**を掛けたり、**インサート**や**バージョンカラムでの排他処理**などを使うことで、データが巻き戻らないような仕組みで行なう。
  - コマンドはフレンド機能があるので、**ロック**で対応している
- パフォーマンスにも非常に影響が出やすいポイントなので、なるべく避ける。

自身のユーザーデータのみ更新する

例：フレンド機能を追加したい！



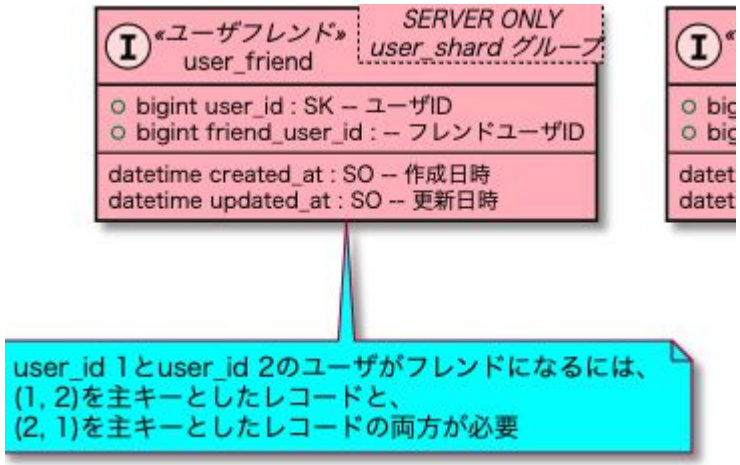
ユーザデータ

運用・仕様変更を常に意識する

- 常に「どのように運用するか」「どのような仕様変更が入りそうか」などのことを意識する。
  - これは他のソーシャルゲームをやり込んだり、UIのデザインなどから方向性を多少ながら推測可能になるため、話題のゲームや新規のゲームなどは一通り触っておくと良いかも。
- 設計を選択する際に追加仕様が発生した際に**破綻しないことが担保**されていれば、実装自体はなくて問題ない。
  - もし破綻する恐れがある場合は、仕様の方向性を責任者とできる限り握っておく。
  - 想定はするが、実装はしないことも重要。
  - 実装してしまった場合いらないロジックがあることによるメンテ性の低下や複雑度の向上につながる。

自身のユーザーデータのみ更新する

例：フレンド機能を追加したい！ただ、フレンドの最大数は制限したい！



自身のユーザーデータのみ更新する

例：フレンド機能を追加したい！ただ、フレンドの最大数は制限したい！

自身のユーザーデータのみ更新する

例：フレンド機能を追加したい！ただ、フレンドの最大数は制限したい！

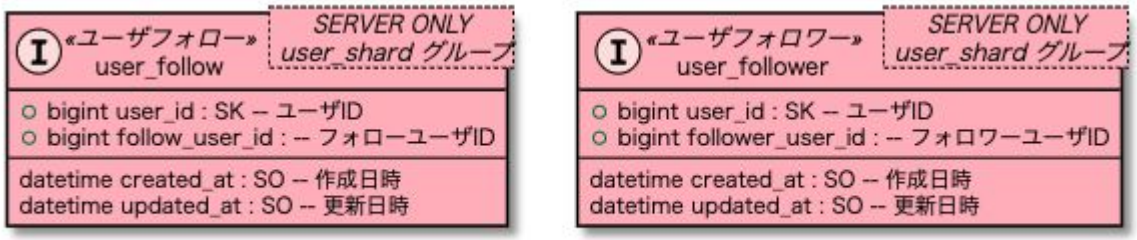
user_friendに挿入する際に、対象のユーザが同一タイミングに更新しないように**ロックを取る**必要がある。

また、フレンドになる際は、自分と相手の**ユーザデータを更新する必要がある**

自身のユーザーデータのみ更新する

解決策：フォロー・フォロワーの仕様に落とし込む
最大数も、フォローの数のみだけに存在するようにする

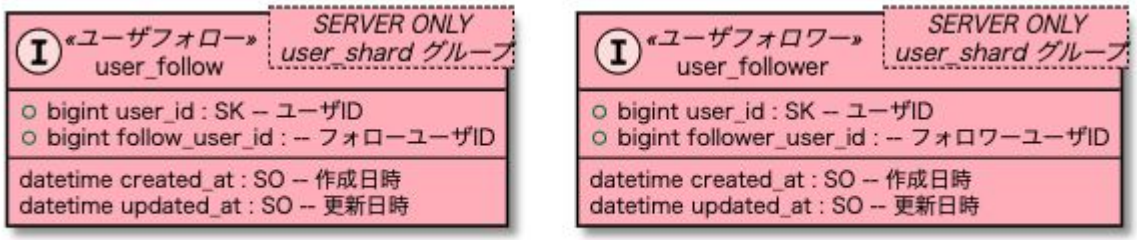
フォロー時は、user_followに対象のuser_idが主キーのレコードを挿入するだけ
また、最大数も自身のデータしか更新していないため特別ロックを取る必要がなくなる



自身のユーザーデータのみ更新する

解決策：フォロー・フォロワーの仕様に落とし込む
最大数も、フォローの数のみだけに存在するようにする

user_followerテーブルは実装次第だが、パフォーマンス向上のために作成して、
user_followのfollow_user_idで検索をかけて、自身のアクセスタイミングで更新するなどが可能



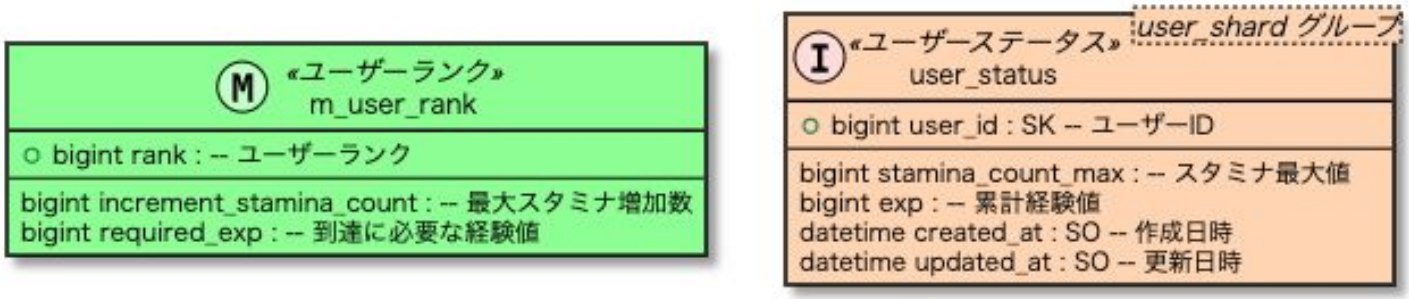
ユーザデータ

マスター側で参照できるデータはユーザーデータに保持しない

- 保持してしまった場合、マスターを変更した際にバッチ処理が必要になるなどのデメリットが生じる。
  - マスタ入力による不具合は多いため、なるべくユーザデータにもたないようにする。
- 逆にマスター変更の影響を与えたくないなどの、仕様として持つ必要がある場合も存在する。

マスター側で参照できるデータはユーザーデータに保持しない

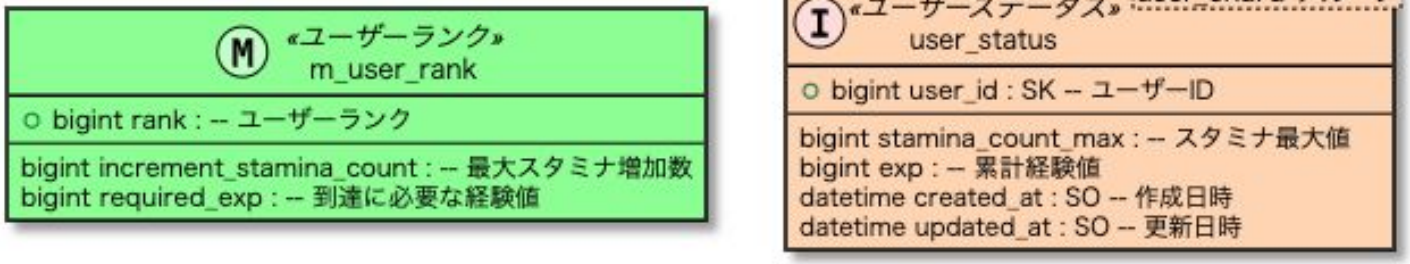
例：ユーザのランク毎にスタミナの最大値が決まります！



マスター側で参照できるデータはユーザーデータに保持しない

例：ユーザのランク毎にスタミナの最大値が決まります！

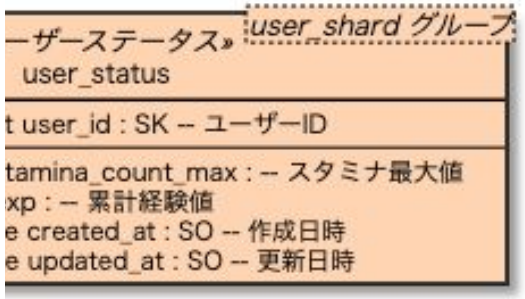
（運用中）ユーザのランク毎のスタミナ最大値を10あげたい！



マスター側で参照できるデータはユーザーデータに保持しない

例：ユーザのランク毎にスタミナの最大値が決まります！

（運用中）ユーザのランク毎のスタミナ最大値を10あげたい！

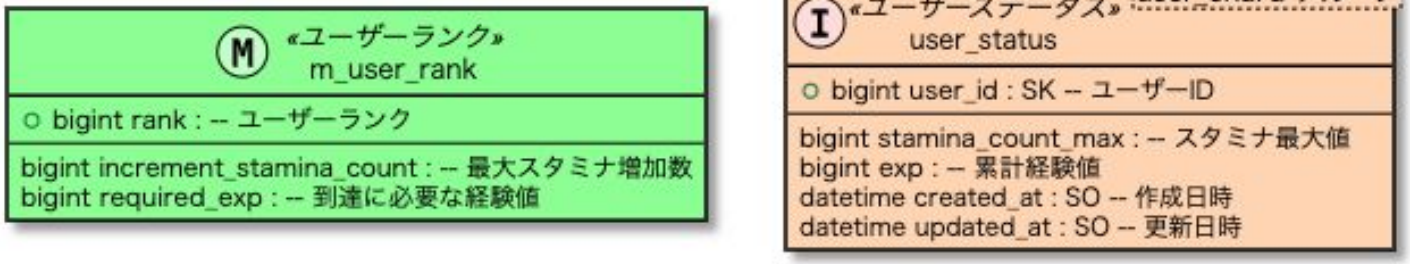


マスター側で参照できるデータはユーザーデータに保持しない

例：ユーザのランク毎にスタミナの最大値が決まります！

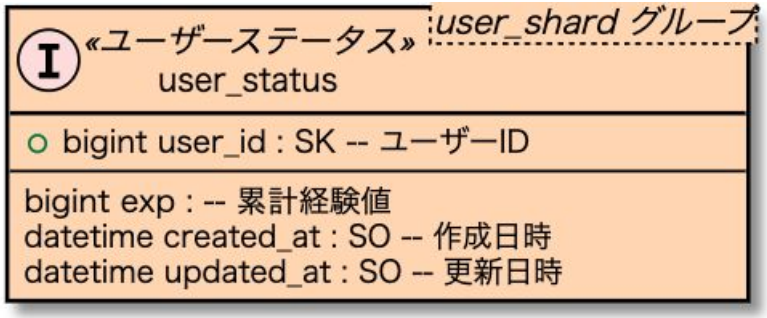
（運用中）ユーザのランク毎のスタミナ最大値を10あげたい！

⇨ user_statusの**全データを更新**する必要が出てくる



マスター側で参照できるデータはユーザーデータに保持しない

解決策：ユーザデータの所持する必要のないデータマスタ側に情報をもつ



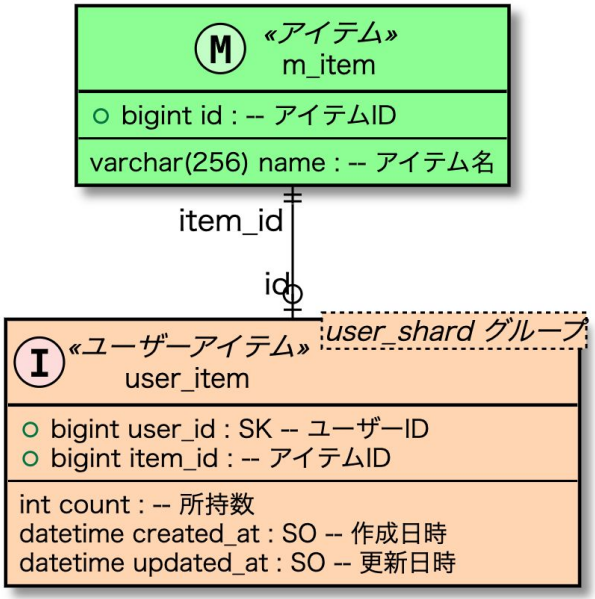
ユーザデータ

レコードが存在する場合所持しているというような判定はさける

- レコードを生成した上で所持している**個数**、または**フラグ**を持っている方が、後々所持していないが別の状態を保つ必要が出てきた際などの仕様変更に強くなる。
- クライアント側の話だが、リストに表示する際にユーザデータのレコードの所持ではなく、マスターが有効かどうかで表示を行なうようにしてもらったほうが良い。
- クライアント側も巻き込んで、**不必要にレコードを増やさず**に必要になったタイミングでレコードを増やすような設計が好ましい。

レコードが存在する場合所持しているというような判定はさける

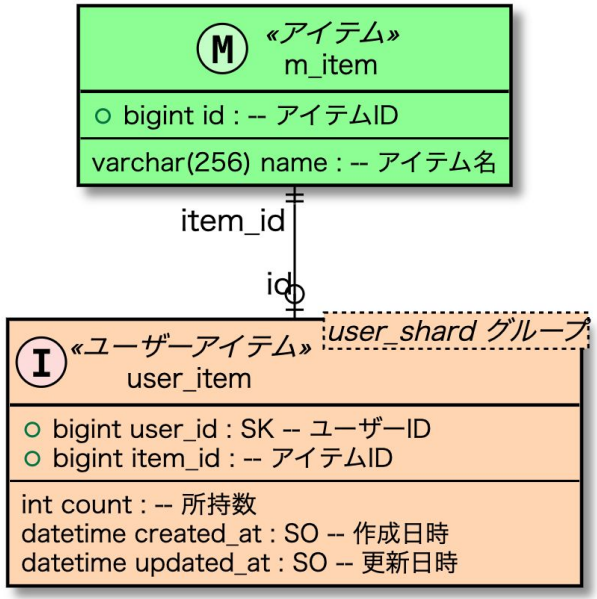
例:ユーザはアイテムを複数所持できます！



レコードが存在する場合所持しているというような判定はさける

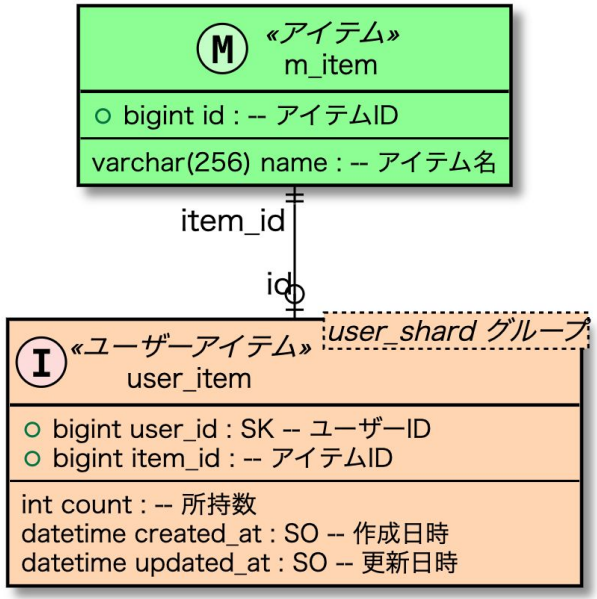
例:ユーザはアイテムを複数所持できます！

所持しているアイテム一覧確認機能があります！



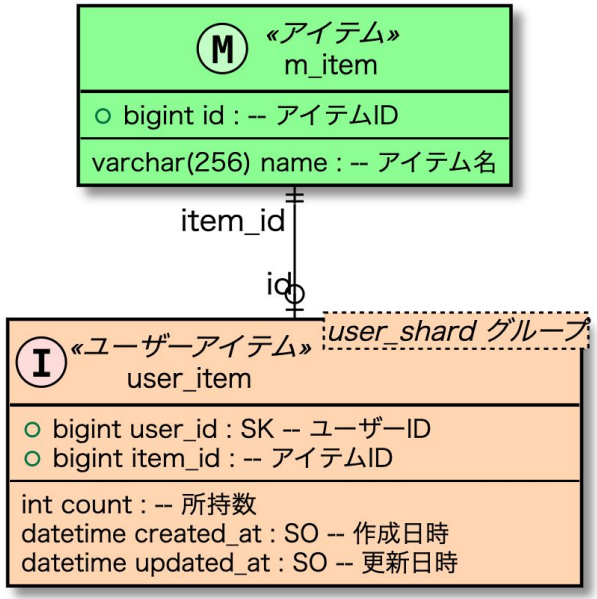
レコードが存在する場合所持しているというような判定はさける

クライアント:「アイテム一覧は、user_itemのレコードに全部入ってそうだし、それをリストで表示するか」



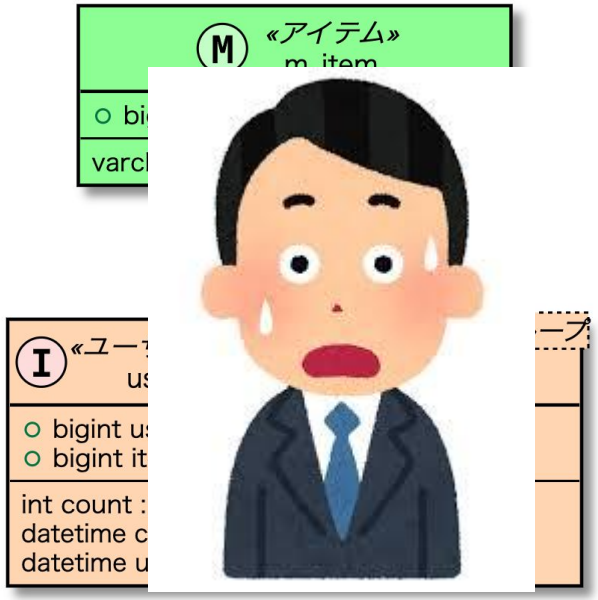
レコードが存在する場合所持しているというような判定はさける

運用中：新規アイテム追加！
…アイテム一覧に所持していないアイテムが表示されない、？



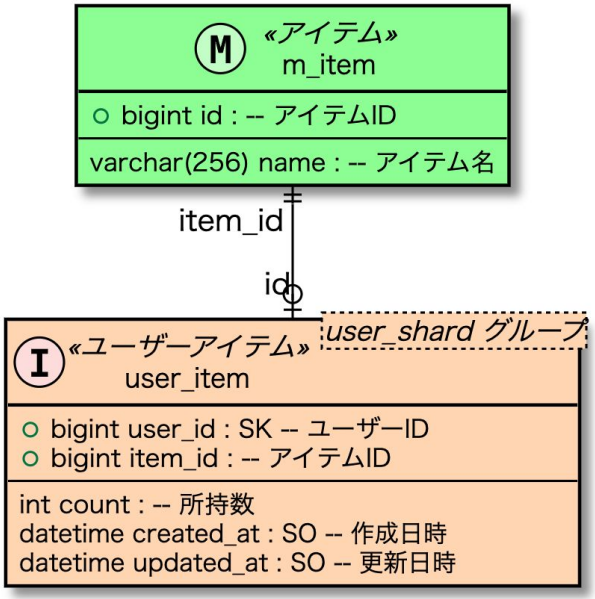
レコードが存在する場合所持しているというような判定はさける

運用中：新規アイテム追加！
…アイテム一覧に所持していないアイテムが表示されない、？



レコードが存在する場合所持しているというような判定はさける

サーバ:「バッチ処理で追加します...」



レコードが存在する…持しているというような判定はさける

サーバ:「バッチ処…

レコードが存在する場合所持しているというような判定はさける

運用期間：7年

全ユーザ数（リセマラユーザ含めて）：1億ユーザ

アイテム数（過去に存在したイベント限定のアイテムも存在）：1万

レコードが存在する場合所持しているというような判定はさける

運用期間：7年

全ユーザ数（リセマラユーザ含めて

アイテム数（過去に存在したイベント も存在）：1万


1兆レコード

レコードが存在する場合所持しているというような判定はさける

データが増えすぎることによるデメリット

・データ数の増加はパフォーマンスに影響しやすい

ユーザデータ

レコードが存在する場合所持しているというような判定はさける

- レコードを生成した上で所持している**個数**、または**フラグ**を持っている方が、後々所持していないが別の状態を保つ必要が出てきた際などの仕様変更に強くなる。
- クライアント側の話だが、リストに表示する際にユーザデータのレコードの所持ではなく、マスターが有効かどうかで表示を行なうようにしてもらったほうが良い。
- クライアント側も巻き込んで、**不必要にレコードを増やさず**に必要になったタイミングでレコードを増やすような設計が好ましい。

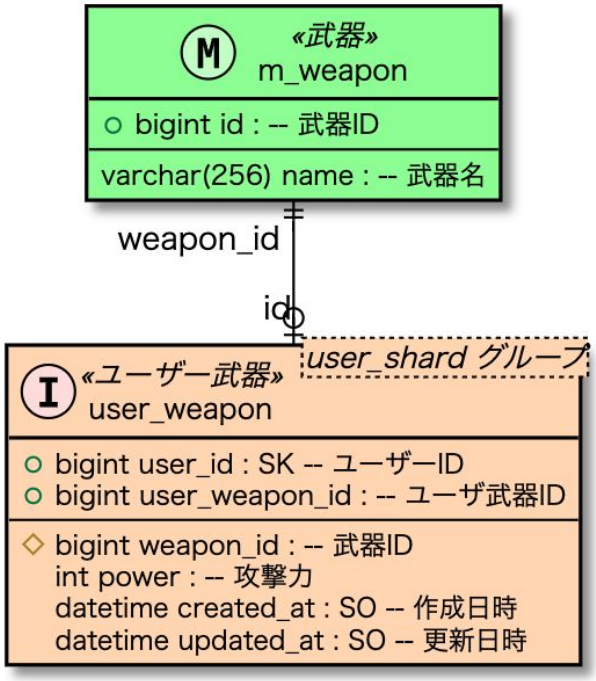
ユーザデータ

運用・仕様変更を常に意識する

- 常に「どのように運用するか」「どのような仕様変更が入りそうか」などのことを意識する。
  - これは他のソーシャルゲームをやり込んだり、UIのデザインなどから方向性を多少ながら推測可能になるため、話題のゲームや新規のゲームなどは一通り触っておくと良いかも。
- 設計を選択する際に追加仕様が発生した際に**破綻しないことが担保**されていれば、実装自体はなくて問題ない。
  - もし破綻する恐れがある場合は、仕様の方向性を責任者とできる限り握っておく。
  - 想定はするが、実装はしないことも重要。
  - 実装してしまった場合いらないロジックがあることによるメンテ性の低下や複雑度の向上につながる。

運用・仕様変更を常に意識する

例：武器は取得時に強さが決まる！



運用・仕様変更を常に意識する

例：武器は所持時に強さが決まる！

運用・仕様変更を常に意識する

例：武器は所持時に強さが決まる！

データ設計者「この前xxxタイトルで運用中に一律で武器のパラメータ上がったな」

運用・仕様変更を常に意識する

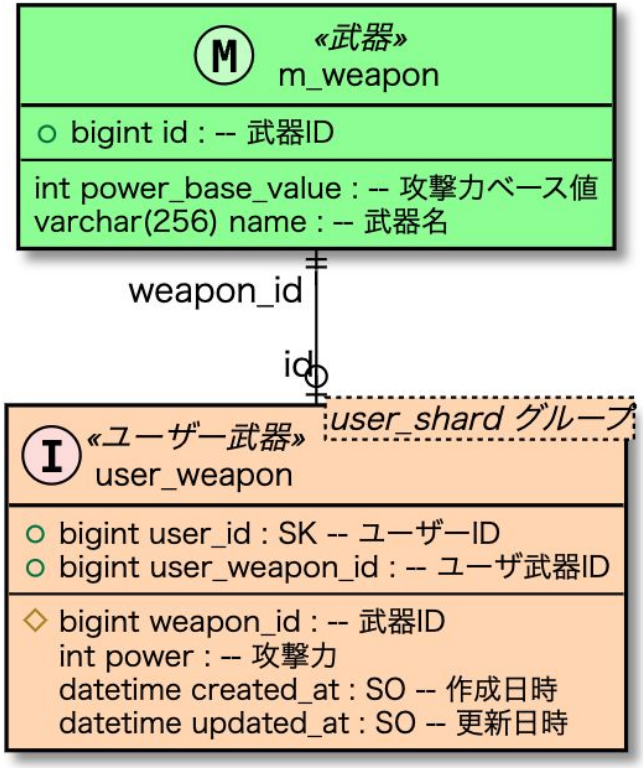
例：武器は所持時に強さが決まる！

データ設計者「武器って運用中に武器毎に攻撃力あげたりしたいとかありますか？」

企画「あるかも！てかあると思います！」

運用・仕様変更を常に意識する

改善後のデータ設計



運用・仕様変更を常に意識する

あらかじめ想定しておくことで、、、

・変更が容易に（わざわざバッチ処理を行わなくてもよくなった）

・ゲームの表現の幅が少し増えた

ユーザデータ

レコード数が増えすぎないようにする

- マスタもそうだが、ユーザデータは特にレコード数が増えすぎないように注意する
  - レコード数が増えることでパフォーマンスの低下に繋がりやすい。
- マスタデータのフラグ管理などでレコードが増える場合などもゲームではよくあるためそういう場合は、bit演算を使ったフラグでレコード数の増加を抑える工夫が必要
  - ミッション受け取りなど

余談：ダークでは、運用が長くなりユーザデータが増えることでログインできないユーザが発生したらしい

レコード数が増えすぎないようにする

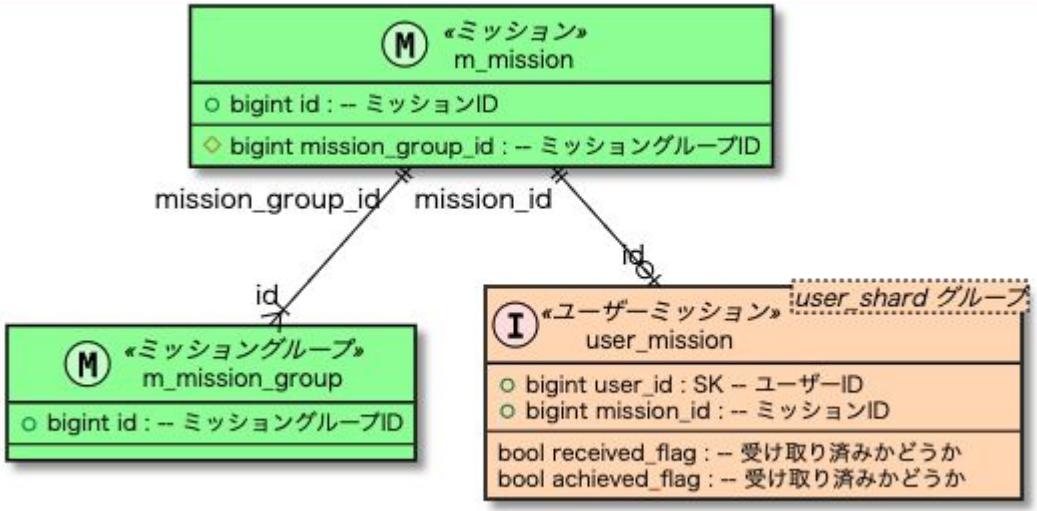
例：よくあるミッション機能入れたい！

・ミッション達成したら、受け取れるような感じ

レコード数が増えすぎないようにする

例：よくあるミッション機能入れたい！

・ミッション達成したら、受け取れてるような感じ



レコード数が増えすぎないようにする

例：よくあるミッション機能入れたい！

企画「ミッションは一つのミッショングループに大体...128個ぐらい入リマス！
あと、ミッショングループは運用していく中で増えるはずで、大体年間36個ぐらい増えますかね🤔」

レコード数が増えすぎないようにする

リリース後1年

DAU：100万ユーザ

年間ミッショングループ増加数：36

1ミッショングループ当たり：128個

ユーザデータ

リリース後1年

DAU：100万ユーザ

年間ミッショングループ増加数：36

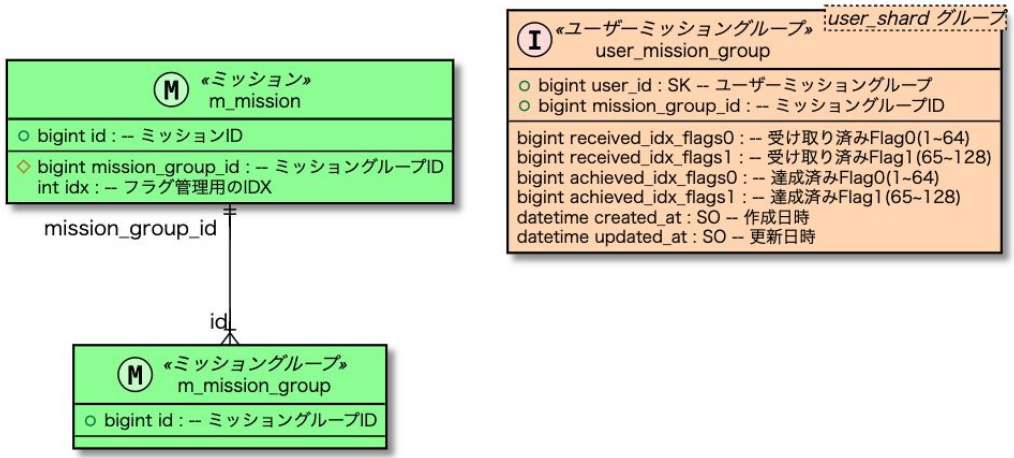
1ミッショングループ当たり：128個


46億レコード

レコード数が増えすぎないようにする

解決策：フラグ管理をビット演算で保持するようにする

※ これで実装した場合に実装が難しくなる機能も中には存在する可能性はあるので、
ビット演算で実装するときは企画との握りをしっかり行う



レコード数が増えすぎないようにする

リリース後1年

DAU：100万ユーザ

年間ミッショングループ増加数：36

1ミッショングループ当たり：128個

ユーザデータ

リリース後1年

DAU：100万ユーザ

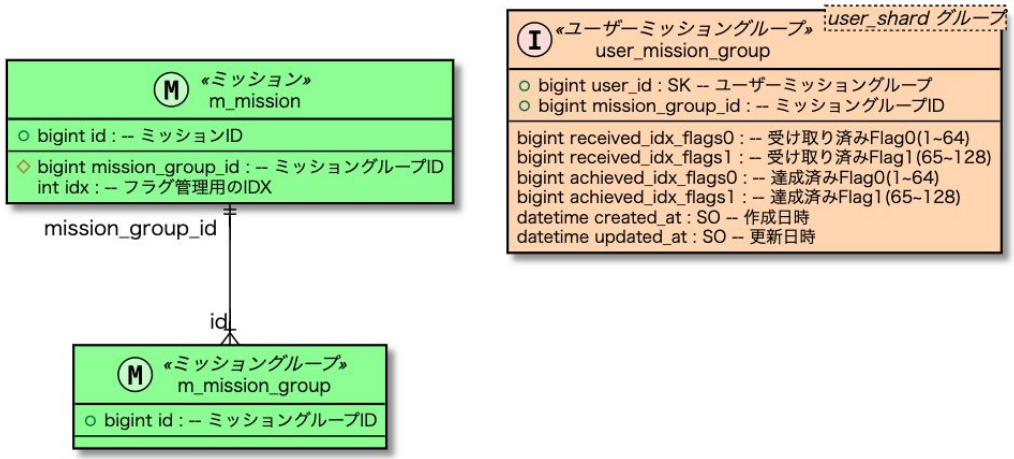
年間ミッショングループ増加数：36

1ミッショングループ当たり：128個


3600万レコード

レコード数が増えすぎないようにする

ひと工夫で、レコード数が1/128まで減少することができる



終わり