大規模ソーシャルゲームのつくりかた

～60分でわかるサーバサイド技術～

グリー株式会社
CTO 藤本真樹　/　増山和幸

自己紹介

GREE

藤本 真樹 (取締役執行役員CTO)

- 31歳 / 独身

- 14歳のころ初めてのプログラミング
- ゲーム会社を志すもいつの間にかウェブ業界へ
- プログラミング言語PHPなど、オープンソースソフトウェア周辺で活動しつつ(最近では分散ストレージサーバなど)、2005年よりGREEの技術関連全般に従事

自己紹介

GREE

増山 和幸

- 38歳 / 既婚
- 船をつくっていたつもりが、いつの間にか某大手ゲーム会社でゲームサーバプログラミング担当
- 2009年よりグリー株式会社にてバックエンド全般を担当

Agenda

GREE

1. GREEの紹介
2. コンシューマ向けオンラインゲームとの違い
   - アーキテクチャ
3. ソーシャルゲームで発生する問題と解決方法
   - スケーラビリティの確保と負荷対策のポイント
   - サーバサイドのフロントエンドなポイント
   - 大規模ソーシャルゲームの運用ノウハウ
4. 質疑応答

GREEの紹介（1）

SNS + ソーシャルゲーム + プラットフォーム

GREEの紹介 (2)

GREE

数字 (2010/06)

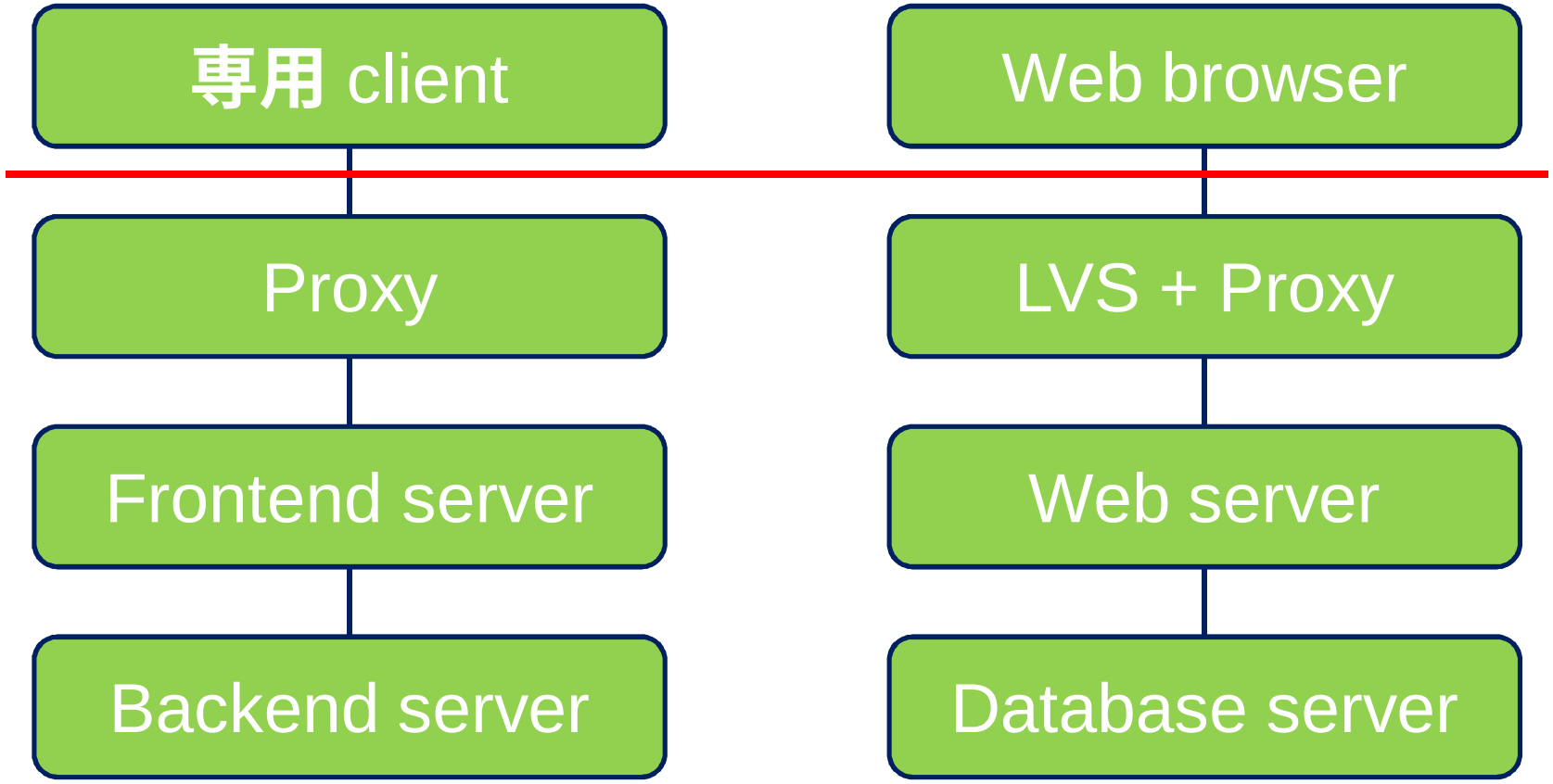
- 会員数　　= 2059万人
- 月間ページビュー　= 357億ページビュー
  - 日次　　= 11億ページビュー
- サーバ台数　　= ？台
  - ほとんどが普通のx86サーバです
  - SSDなどはこれから
- 最大同時接続数　= ＞？ (画像除く)

コンシューマ向けオンラインゲーム との違い

GREEでサービスしている実際のアプリケーションの例を用いて、規模の大きなソーシャルゲームのシステムを構築する上でどのような考慮が必要か、コンシューマ向けオンラインゲームのシステムと比較を交えながら紹介します。

コンシューマ向けオンラインゲームとの違い

- 構成の違い



結論：そんなに変わらない

コンシューマ向けオンラインゲームとの違い

- 特徴の比較

コンシューマ向けオンラインゲーム	携帯ソーシャルゲーム
非同期アクセス	同期アクセス
ステートフル	スレートレス
世界の境界があることが多い	世界の境界がないことが多い
低レイテンシであることが重要	レイテンシは低い方がいい
アクセスは夜から深夜にかけて増える	夜から深夜に加えて、朝8時、昼12時前後も増加
コアゲーマーが多い	普段ゲームをやらない人もプレイ

あげてみてもよくわからない

なので、早速実例に

釣り★スタ

- 名前の通り、釣りをするゲーム
- 2007年5月リリース
- エンジニア3人で作り上げた

どんなゲーム？

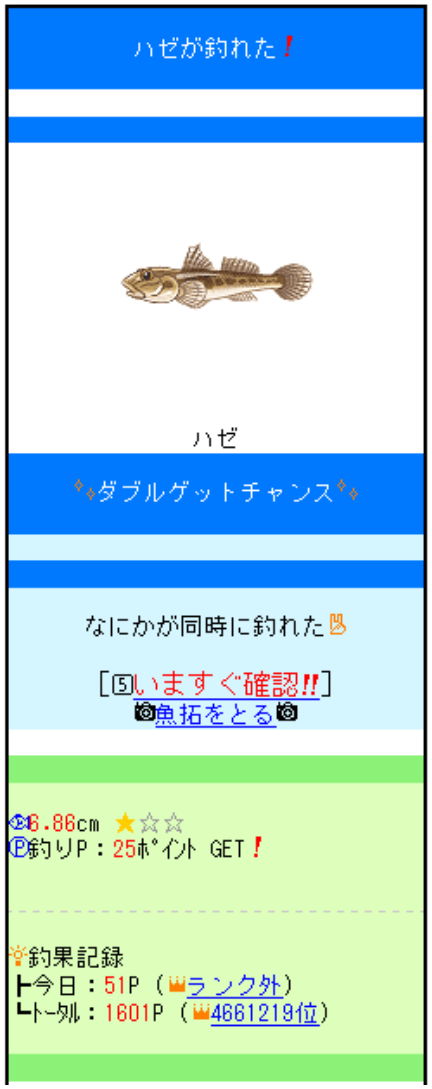
- まず、釣り場を選ぶ
- 自分のレベルがあがると行ける釣り場が増える

どんなゲーム？

- Flashで釣りゲーム

どんなゲーム？

- 魚を釣り上げると…
  - 魚の大きさによっては記録更新
  - 釣りポイントゲット
  - ランキング更新
  - 魚拓を取っておくこともできる



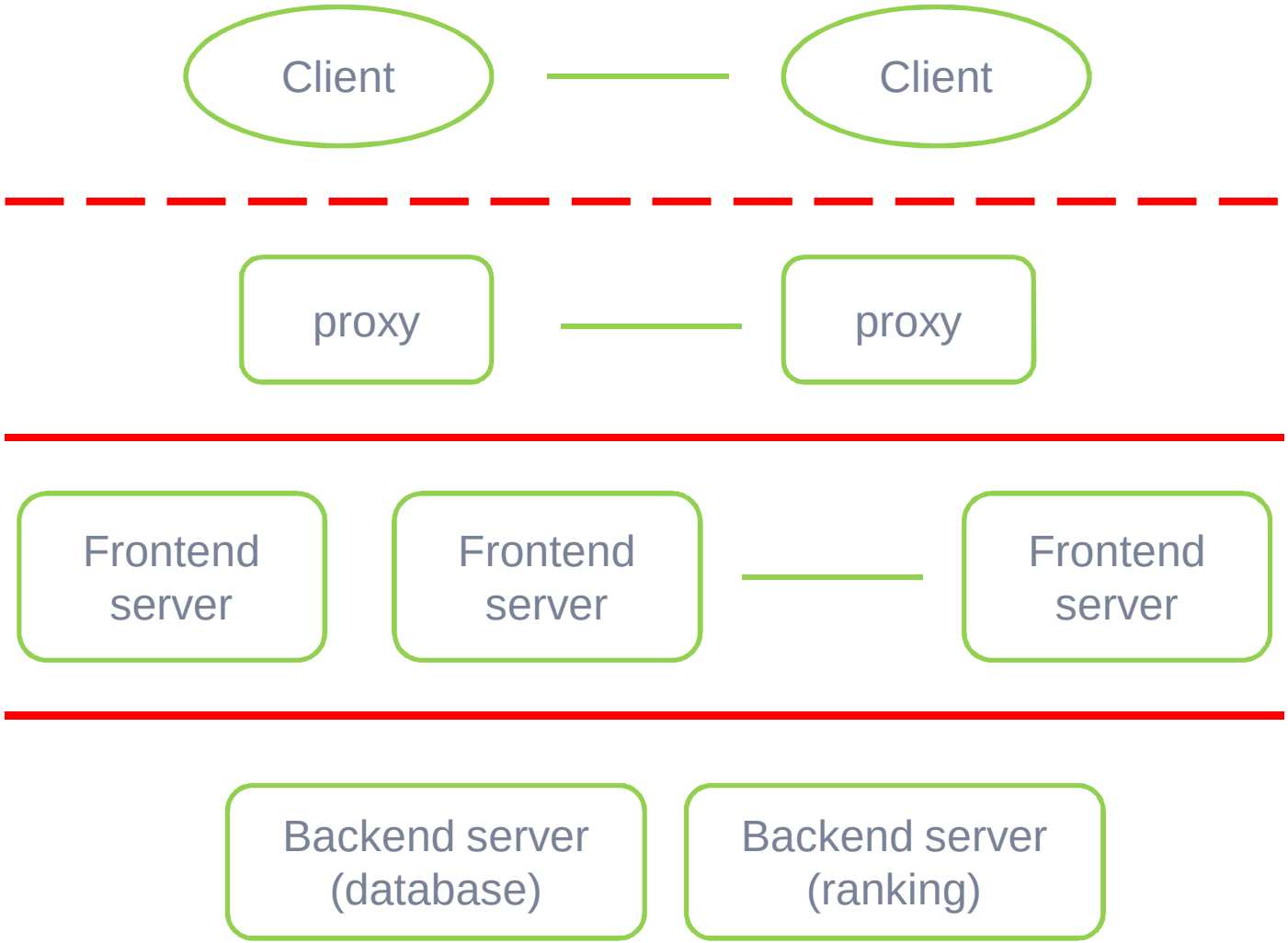
基本はこれだけです。

いろいろと工夫している点はありますが
システムには関係ないところは切り捨てます。

ソーシャルゲームでの工夫の内容を
聞きたい人はこの後の
「2000万人を魅了するソーシャル
ゲームの作り方」
セッションまで！

コンシューマゲームでこれを作るならどうするか

コンシューマゲームでこれを作るならどうするか

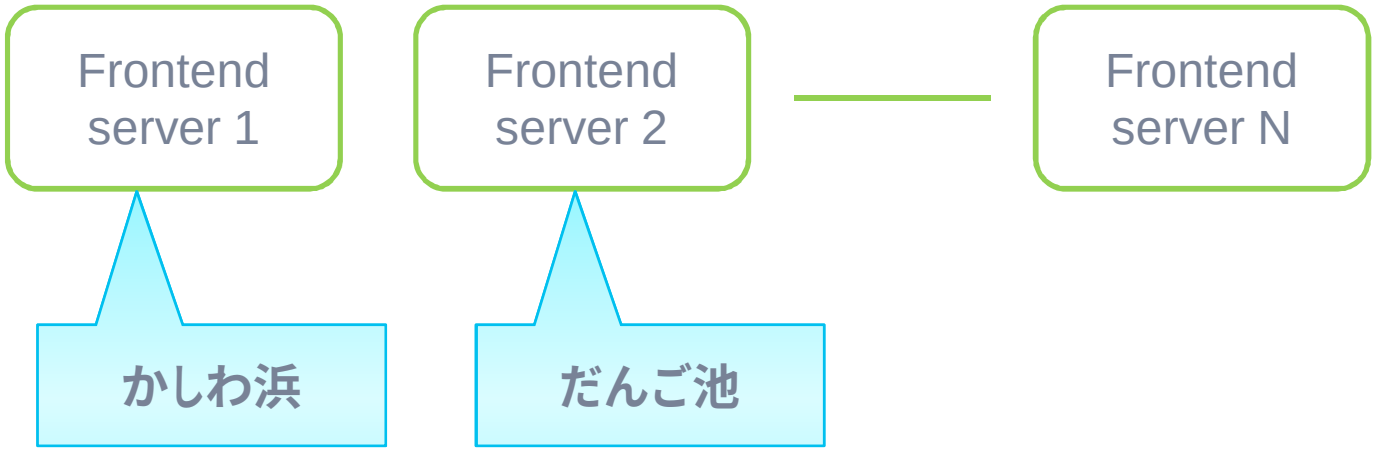


コンシューマゲームでこれを作るならどうするか

- 釣り場毎に Frontend serverが存在する
- Proxyはユーザからのコネクションと Frontend serverを結び付ける
- ログイン時にBackend serverから必用なデータをロードし、
  Frontend serverのメモリ上に置く
- 釣り上げたらメモリ上のデータを更新する
- ランキング等の全体にかかわるデータをBackend serverに送信
  - Backend serverはその結果をFrontend serverにpush
- 即時反映の必要のないデータは後から更新
  - たとえば、釣り場をさる/変えるタイミングで

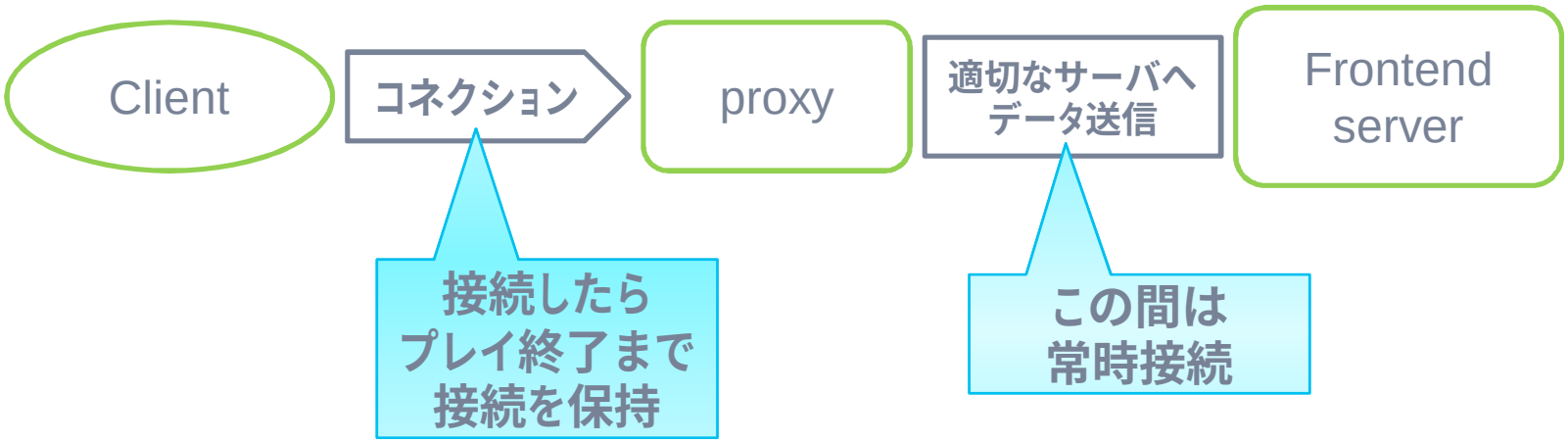
コンシューマゲームでこれを作るならどうするか

- 釣り場毎に Frontend serverが存在する



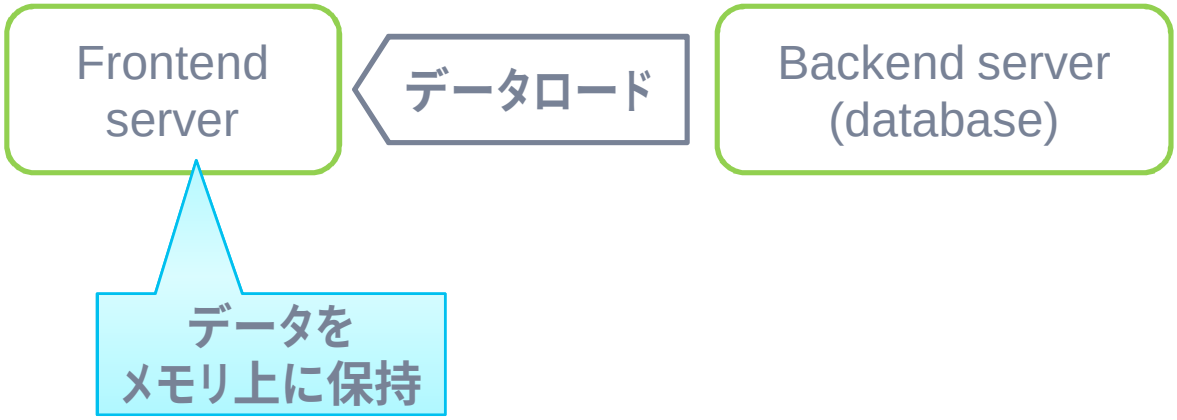
コンシューマゲームでこれを作るならどうするか

- Proxyはユーザからのコネクションと Frontend serverを結び付ける



コンシューマゲームでこれを作るならどうするか

- ログイン時にBackend serverから必用なデータをロードし、Frontend serverのメモリ上に置く

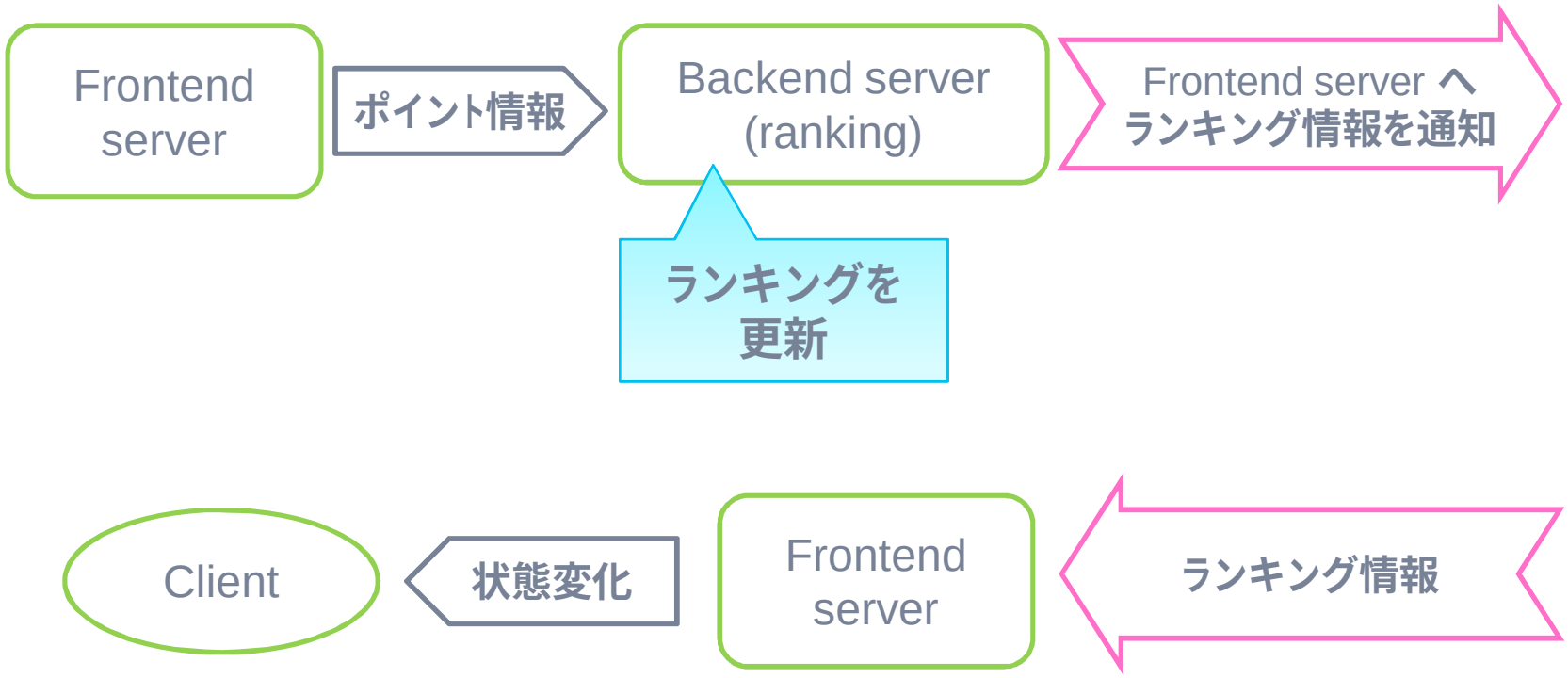


コンシューマゲームでこれを作るならどうするか

- 釣り上げたらメモリ上のデータを更新する

コンシューマゲームでこれを作るならどうするか

- ランキング等の全体にかかわるデータをBackend serverに送信



コンシューマゲームでこれを作るならどうするか

- 即時反映の必要のないデータは後から更新

コンシューマゲームでこれを作るならどうするか

GREE

- このシステムのメリットは「リアルタイム」であること

たとえば…

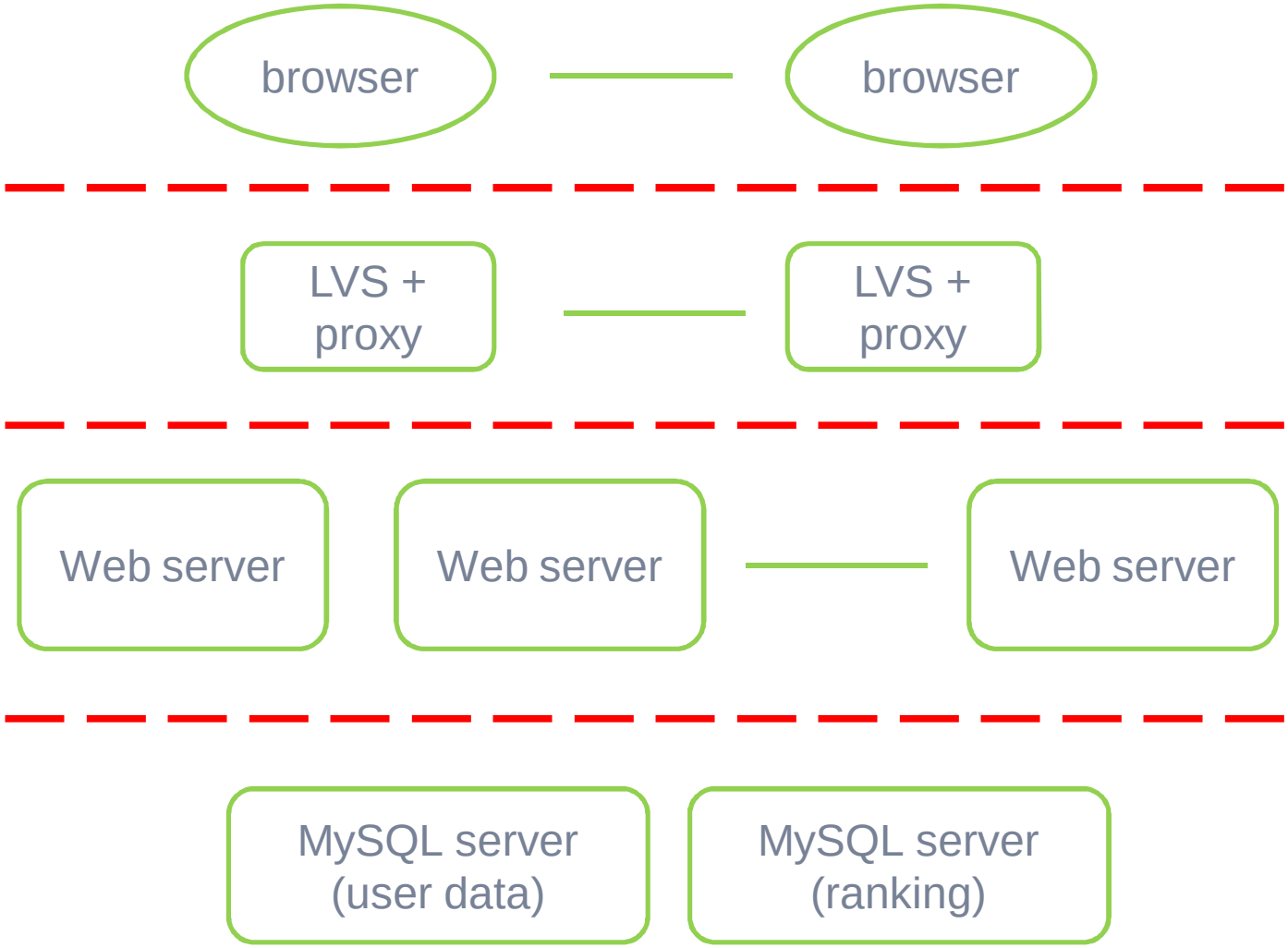
- 同じ釣り場の人たちで会話（チャット）
- 近くにいる人が何をつり上げたか（失敗したか）わかる
- 複数人数で大物を釣る、などの助け合いもできるかも

反面…

- 1台のサーバのリソース（メモリ/CPU等）の限界＝釣り場の人数の限界
- Frontend serverをまたいだリアルタイム処理が難しくなる
- Backend serverをスケールするように作らないといけない

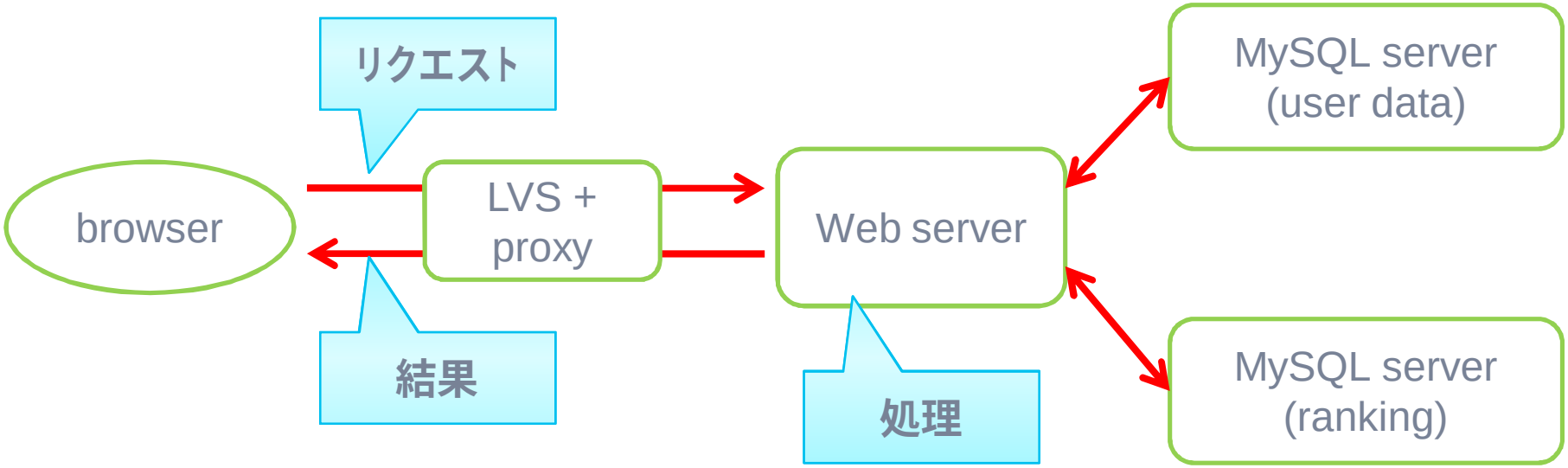
グリーのシステムではどうするか

グリーの構成だと



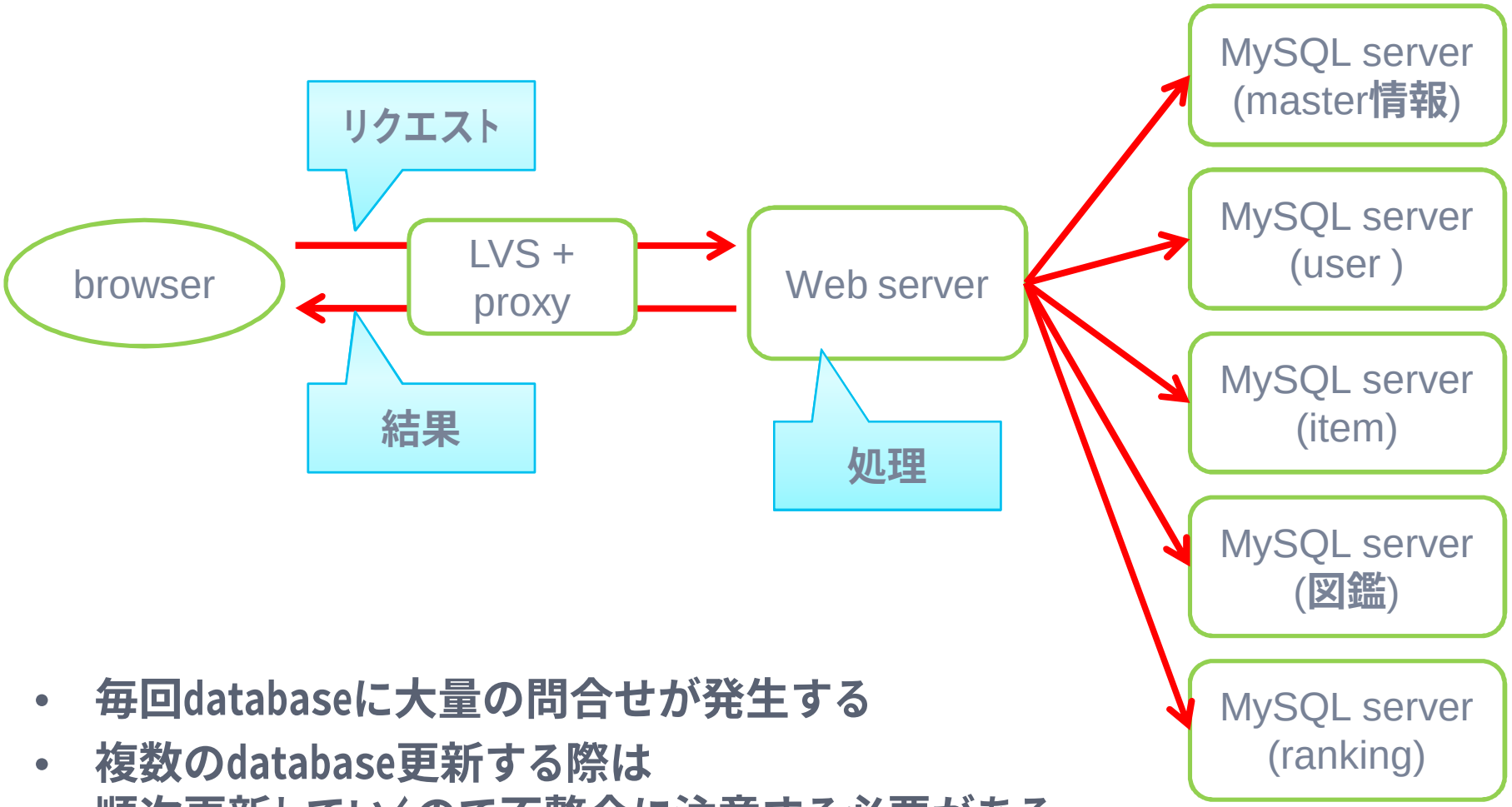
グリーの構成だと

- すべてのwebサーバは等価。Proxyは一番負荷の低いwebサーバにリクエストを割り振るだけ
- ユーザデータはすべてdatabaseにある
- リクエストの度にdatabaseからデータをロードし、変更された値をセーブする



釣り★スタをつくるとしたら

グリーの構成で作るとしたら



- 毎回databaseに大量の問合せが発生する
- 複数のdatabase更新する際は
  順次更新していくので不整合に注意する必要がある
  (MySQLでのトランザクションの使用は性能上の制約がある)

GREE

最大の特徴は「シンプル」であること

したがって…

- シンプルであるがゆえに負荷にあわせて拡張しやすい
- Web serverはすべて等価なのでスケール可能
- MySQLデータベースをスケーラブルにしておけば全体がスケール可能

反面…

- 情報を取ることは基本的にコストがかかる
- 問い合わせるＭｙＳＱＬデータベースが多い程レスポンスが落ちる

ちなみに…

最近はやりのソーシャルゲームは[5]キーを押すことによってゲームが進行します

[5]キーを押すゲームって？

- [5]キーを押すと
- 何かのパラメータが減って
- 何か（たいてい複数）のパラメータが増える
- そして何かイベントが起こることもある
- さらに[5]キーを押すことで繰り返す

釣りスタのFlash部分を省いたような作り

グリーでは「モンプラ」が該当

「モンプラ」の詳しい話が
聞きたい人はこの後の
「2000万人を魅了するソーシャル
ゲームの作り方」
セッションまで！

モンプラ

モンプラでは

- 冒険に出ると
- 元気が減って
- 経験値が増える
- 達成率が増える
- 他のモンスターと
  エンカウントすることもある

これをMySQLを使って実現しようとすると
更新処理が多すぎて**破綻**します

ではどうするか？

ボトルネック

GREE

- RDBMSのボトルネック
  - CPU → あまりない (ごく稀にあり@GREE)
  - メモリI/O → ほとんどない (ごく稀に(ry)
  - ネットワーク → まれにあり
  - ディスクI/O → ほとんどのケースがここ
    - Read: データサイズ増加 (page cacheヒット率低下)
    - Write: INSERT/UPDATE/DELETE増加

→ 日々ディスクI/Oをどう捌くかとのたたかい

ソーシャルゲームにおけるボトルネック

- ソーシャルゲームはとにかくWrite Heavy
  - リリース直後でのサービスダウンはほぼ↑
  - + 一部機能はSQLが複雑になりがち
    - e.g.「最近ここで魚を釣ったユーザ」あるいは「最近ここで魚を釣ったともだち」
    - e.g.「このアイテムを持っているバトル相手一覧」

解決策

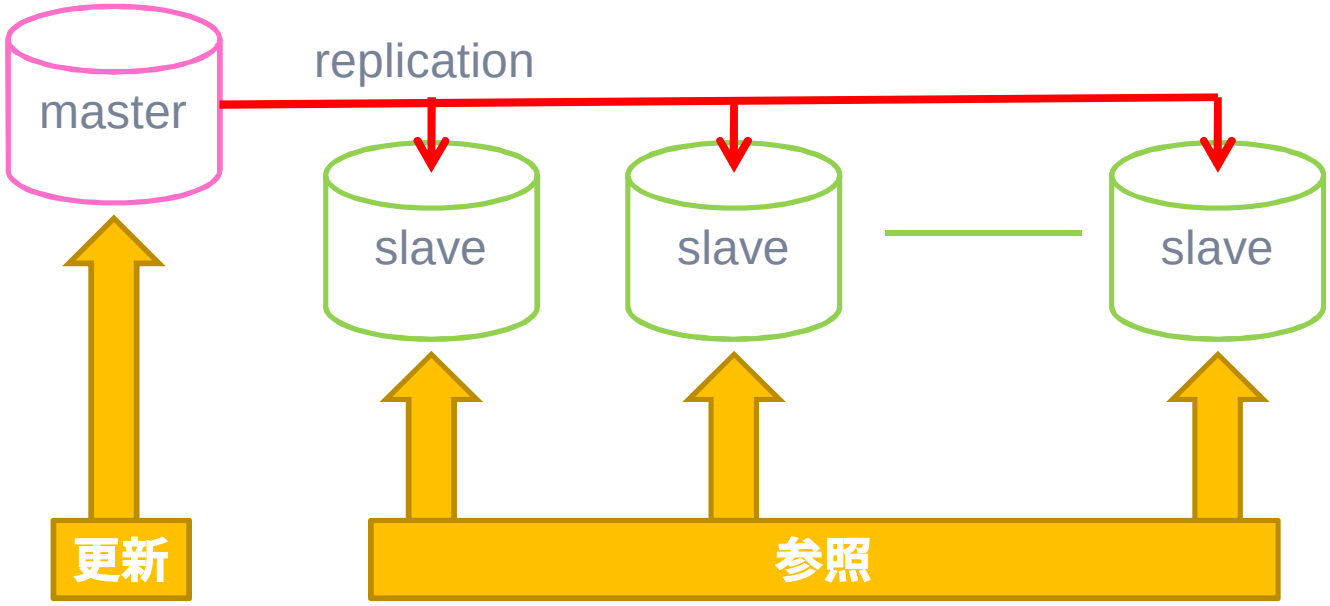
1. Writeを分散
   - sharding (後述)

2. Diskにかかない
   - オンメモリファイルシステムで運用 (e.g. tmpfs)

3. Diskにあまりかかない
   - キャッシュ(memcached)や中間データ(KVS)の利用

Writeを分散:Sharding（1）

- Read (Writeの前に)
  - MySQLではレプリケーション機能を使って参照専用のslaveを増やせる
  - 基本的には台数を増やすだけで対応可能
  - *サーバ単位でのWriteの量は変わらない点に注意*



Writeを分散:Sharding（2）

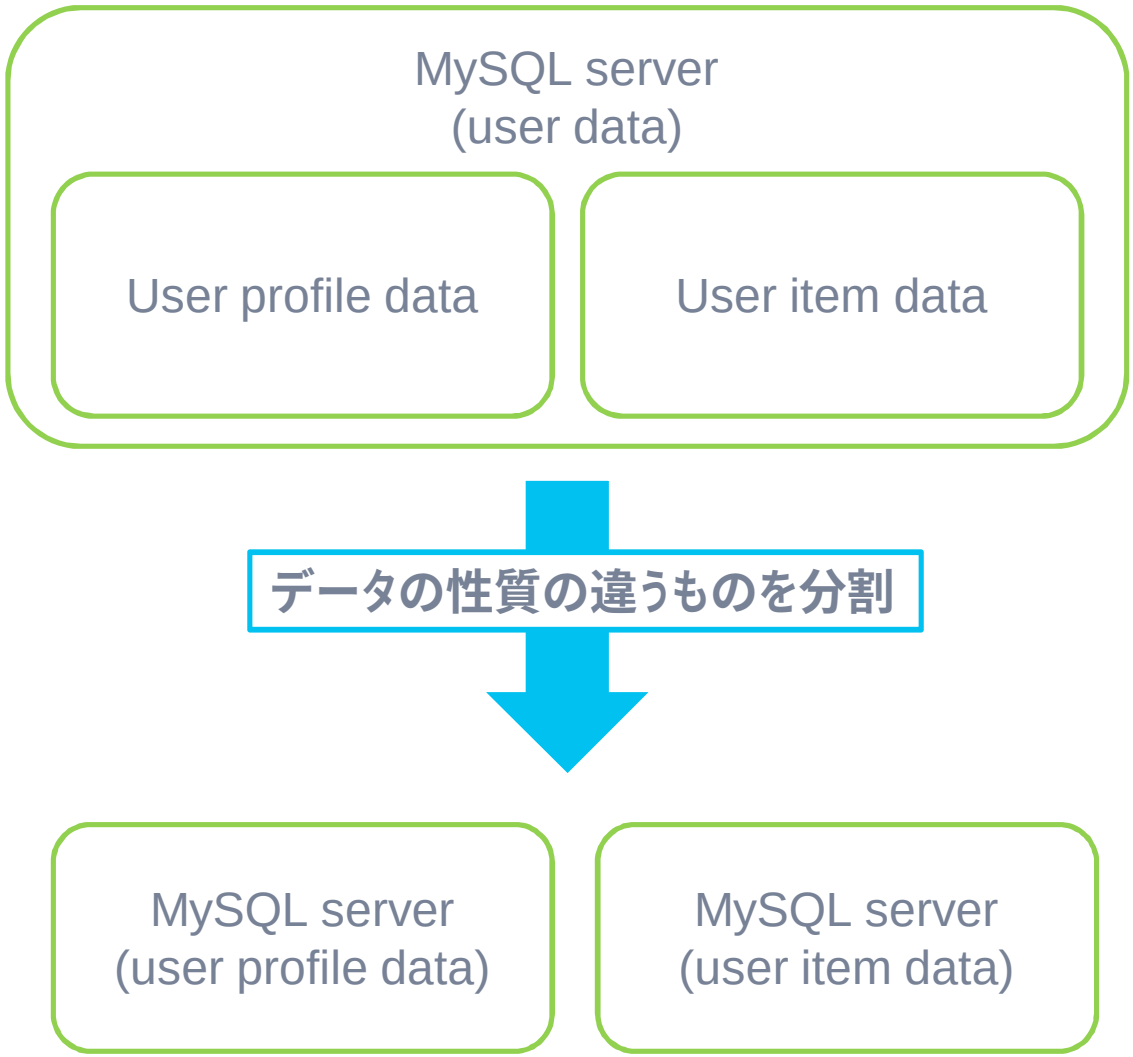
GREE

- Write
  - マススレーブ構成では、マスターは1台
  - 書き込みの限界は低い

→ 分散

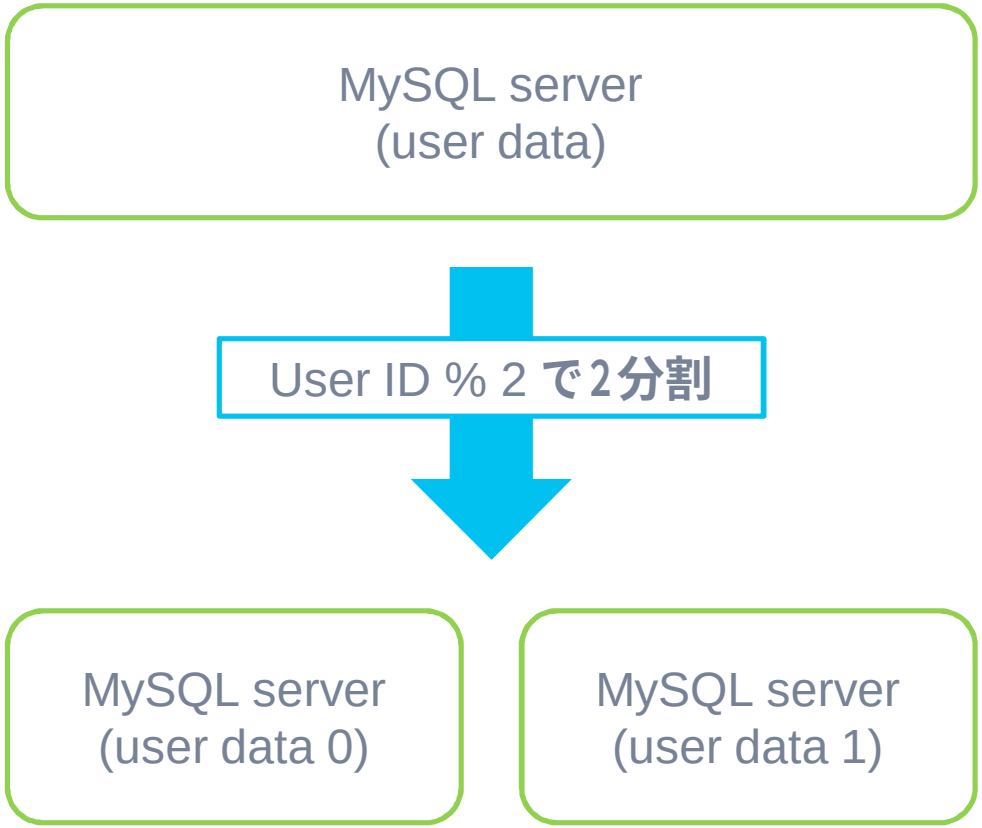
  - (垂直分割)
    - 機能ごとにテーブル/データベースを分ける
  - 水平分割 (Sharding)
    - 同一テーブルを範囲ごとに分割する
      - Range
      - Hash / Modular
      - Index

Writeを分散:Sharding（3）



→ JOINが制約

Writeを分散:Sharding（4）



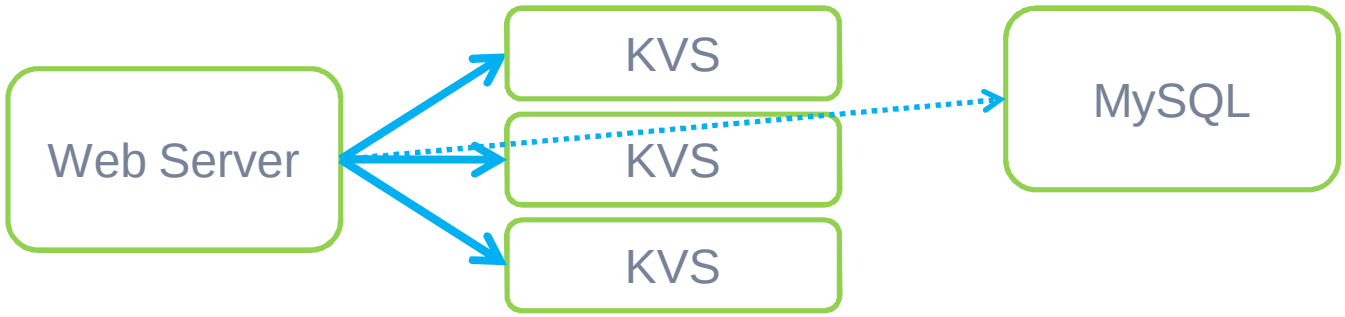
→ User ID**以外の**Condition**が制約**
(たとえば Age**などで検索したい場合は？)**

Diskにかかない

- Linuxであればtmpfsに全データを置いてしまう
  - ハイパフォーマンス (I/Oがボトルネックになることはほぼなくなる – 昔ながらの手法)
  - 注意点
    - サーバダウンでデータロスト
      - ラック単位 / ＤＣ単位で分散
      - 適宜snapshot
    - トラフィック (e.g. 12.5k x 1万qps = 1Gbps)
    - Swap領域
    - データサイズ

Diskにあまりかかない

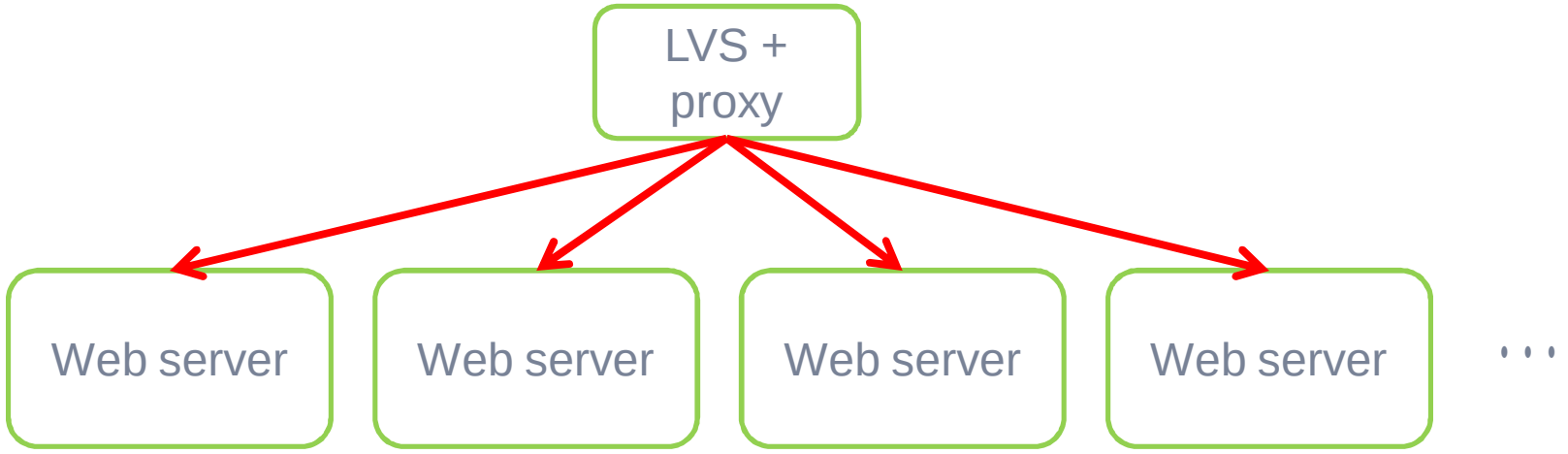
- I/OはKVS [Key Value Store] (GREEではFlare(自作)) に対して行う
  - パフォーマンスがよい (<> RDBMS)
  - 運用コストが低い
  - Memcachedよりも高いデータ信頼性
  - 必要ならオンメモリファイルシステムで運用
- + 適当なタイミング(e.g. Level Up)でRDBMSへFlush
  - 集計 / Valueでの検索 / バックアップ



サーバサイドの
フロントエンドなポイント

スケーラビリティ

- フロントエンド(ウェブサーバ、アプリケーションサーバ)のスケーラビリティ
  - 確保することは容易(単純に並列させる戦略)
  - 構成の標準化が重要(Real Machine or VM?)

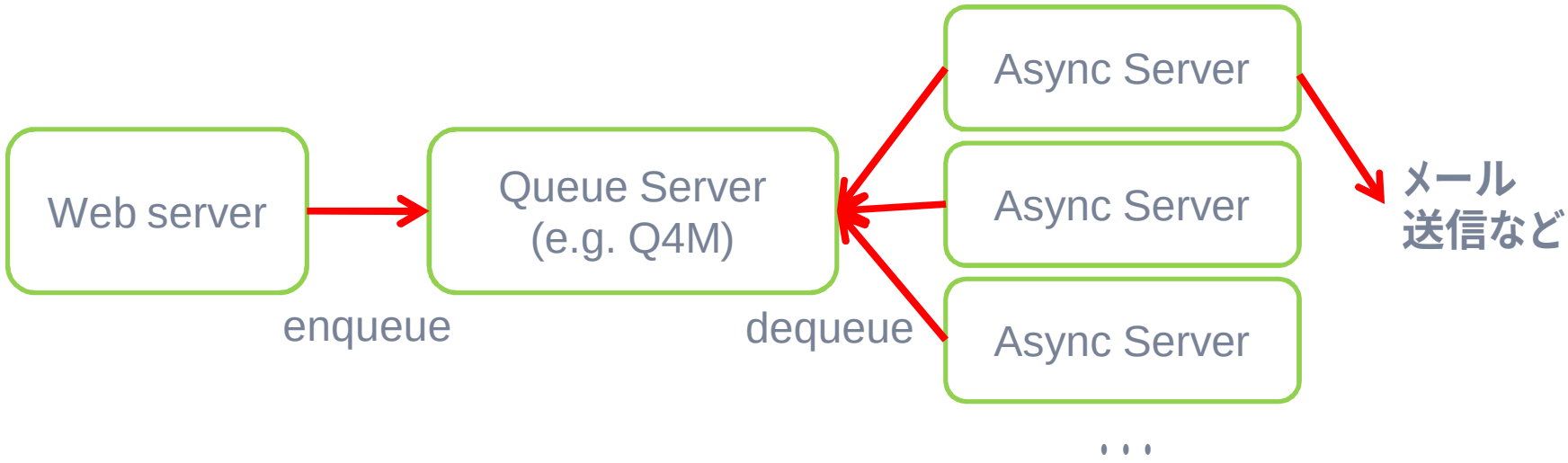


データアクセスレイヤ

- たとえばJavaでいうところのDAO
  - フレームワークとして制限をかけてしまう
- GREEのDAOの特徴
  - DAO : Table = 1 : 1 (基本的にJOIN禁止)
  - Shardingサポート
    - SQLの実行時に、Shard and/or Tableを決定するヒントを渡すインターフェース

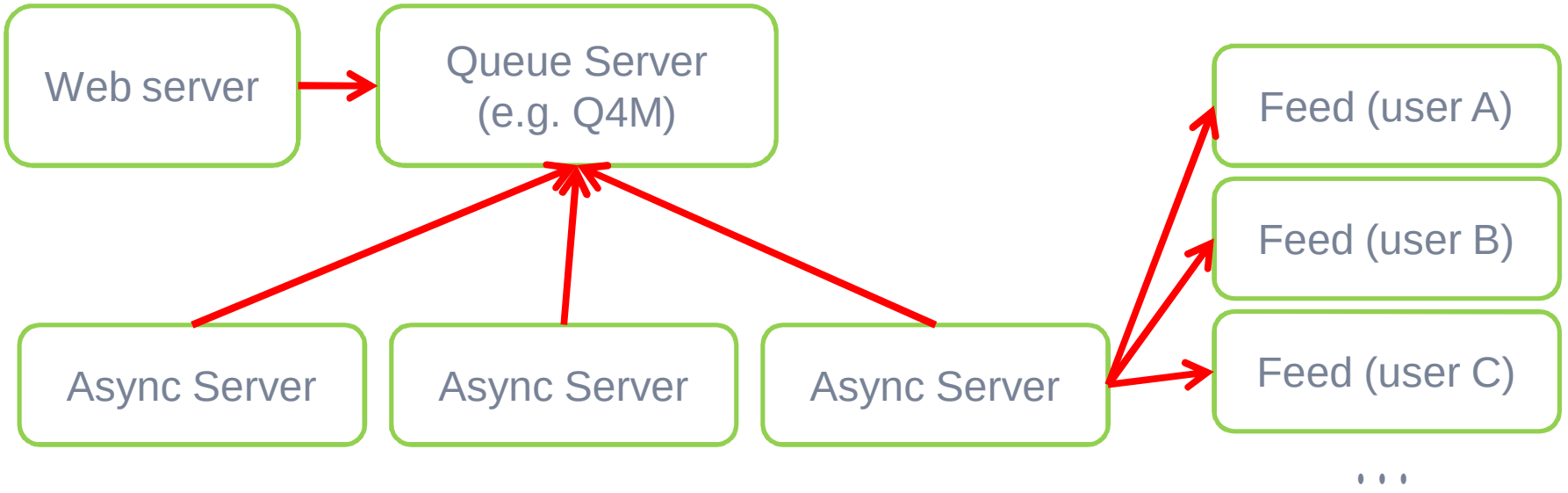
非同期処理（1）

- コストの高い処理は非同期で
  - e.g. コミュニティメール一斉送信処理
    - コミュニティ管理者による、参加者への一斉メール送信機能
    - 参加者が数十万人などで破綻　→　非同期処理



非同期処理（2）

- 非同期処理によるpush型データ更新
  - e.g.「～さんが～しました」等のお知らせ
    - 特に 1 : n の場合に有効
  - これ以外でも「最近～したユーザ」などのSQLが複雑になりがちなデータはpushでの更新が有効



チート

- 手動ブルートフォース
  - モバイルユーザのガッツは侮れません
    - 場合によっては手分けして…
  - サーバ側トークン + 想定の1-2段上の厳しさでチェック

http://mland.gree.jp/?...
&srv_type=1
&rand=**c0600961bd4b5c59a3fbe7432afb885c**

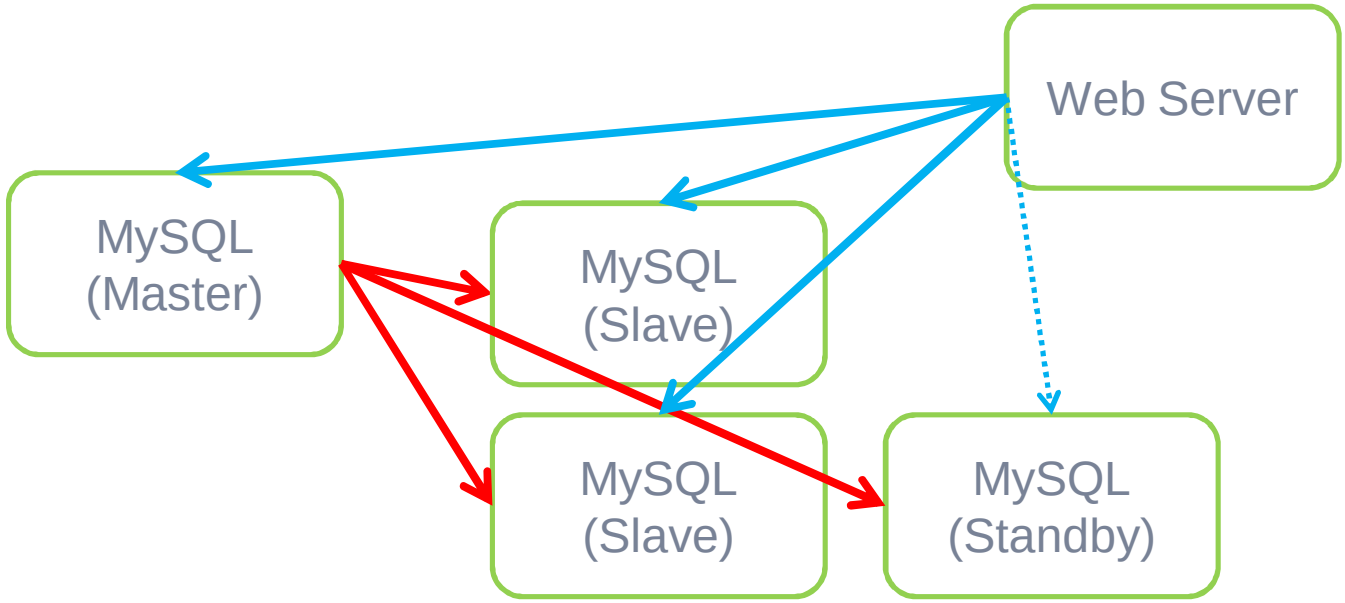
チート

- CSRF (Cross Site Request Forgery)
  - コミュニティなどに「便利ツール」などと称して「仲間を全て削除」や「ステータスリセット」へのリンク
  - 主要な処理にはサーバトークンが必須
    - その他reloadや、画面メモ対策など

大規模ソーシャルゲームの運用ノウハウ

MySQL:Slave

- Slave が落ちたら？
  - 切り離すだけ
  - Web serverにある、slave の情報を更新(エントリを削除)
  - GREEでは最低2台のSlaveで運用 (できれば3台)
- Slave の追加
  - Standby(常に待機しているslave)から、新しいslaveを作る
  - Web serverにある、slave の情報を更新(エントリを追加)

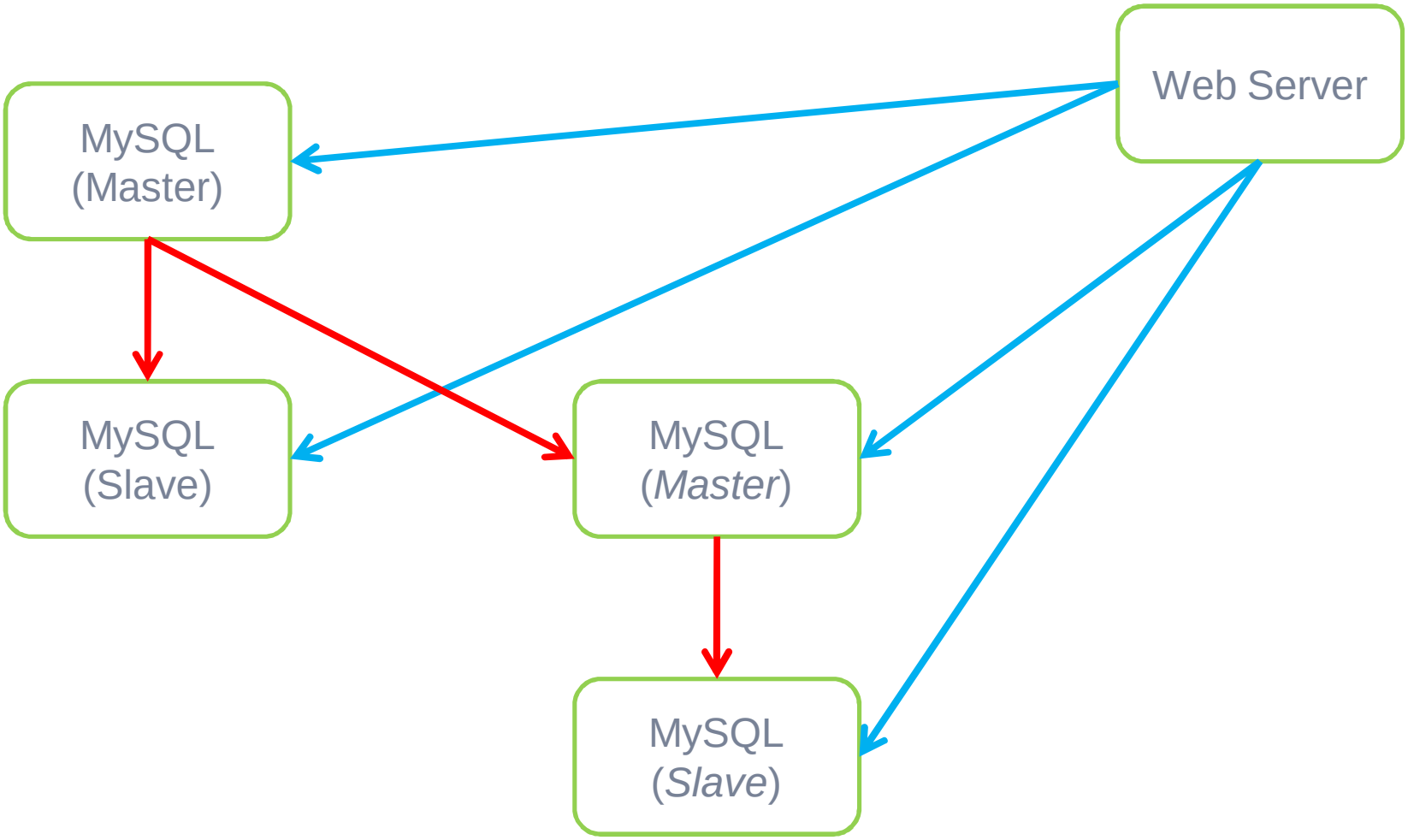


MySQL:Master

GREE

- Masterが落ちたら？
  - Master切り換えを実施 (適宜自動化)
  - 全slaveでbinlogのpositionチェック一致しているサーバを用いる
  - 新masterを選択、新masterでRESET MASTER
  - 他のslaveで CHANGE MASTER TO ～
  - Web sereverの書き込み先を変更
- 無停止でmaster切り換え
  - Alterしたい時、サーバのスケールアップをしたい時、サーバのバージョンアップ時等
  - Masterの下に同じ構成のものをもう一つ作る
  - Web server の参照先を新しい databaseに変更
  - Web server の書き込み先を新しい databaseに変更
- 無停止でmaster分割
  - 基本的には無停止での切り換えを同様の作業
  - ただし、分割したい数だけshardを作る (replicate-*-table, replicate-*-db等を利用する)

MySQL:Master



監視

- サーバ監視 (意外と気づきにくいもの)
  - MySQL Slaveレプリケーション状態
    - Yes/No
    - 遅延状態 (負荷バロメータ)
  - MySQLテーブルデータサイズ
  - MySQL Slow Log
  - MySQL Slave書込 (Slaveへの直書込みがないか？)
  - サーバトラフィック
  - 時計

おまけ：ソフトウェア

ソフトウェア

- Linux (Debian GNU/Linux)
- LVS (ipvsadm)
  + Apache 2.2 w/ mod_proxy_balancer
- Apache + mod_php (w/ eaccelerator)
- MySQL
- Memcached
- swftools, imagemagick
- Original
  - Flare, nanofs, g2proxy

おまけ：システム以外で違うところ

リリースサイクルが短い

毎日定期リリース

テストは自分たちでがんばる

企画者＋プログラマ＋ディレクター
＝エンジニア

ご清聴ありがとうございました