AWS + Windows(C#)で構築する .NET最先端技術によるハイパフォーマンスウェブアプリケーション開発実践

AWS Summit Tokyo 2014 – Yoshifumi Kawai

Grani

Self Introduction

@仕事

河合 宜文 - 株式会社グラニ 取締役CTO

http://grani.jp/

「最先端のC#技術を使った」ゲーム開発

C# 5.0 + .NET Framework 4.5 + ASP.NET MVC 5

@個人活動

Microsoft MVP for Visual C#/野良C#エヴァンジェリスト

Web http://neue.cc/

Twitter @neuecc

Facebook https://www.facebook.com/neuecc

About Grani

「神獄のヴァルハラゲート」
グラニ第一弾タイトル
CM放送
100万人以上のユーザー
GREE Platform Award 2013 受賞

「モンスタハンターロアオブカード」
CM放送
カプコンと協業の大型IPタイトル

Agenda

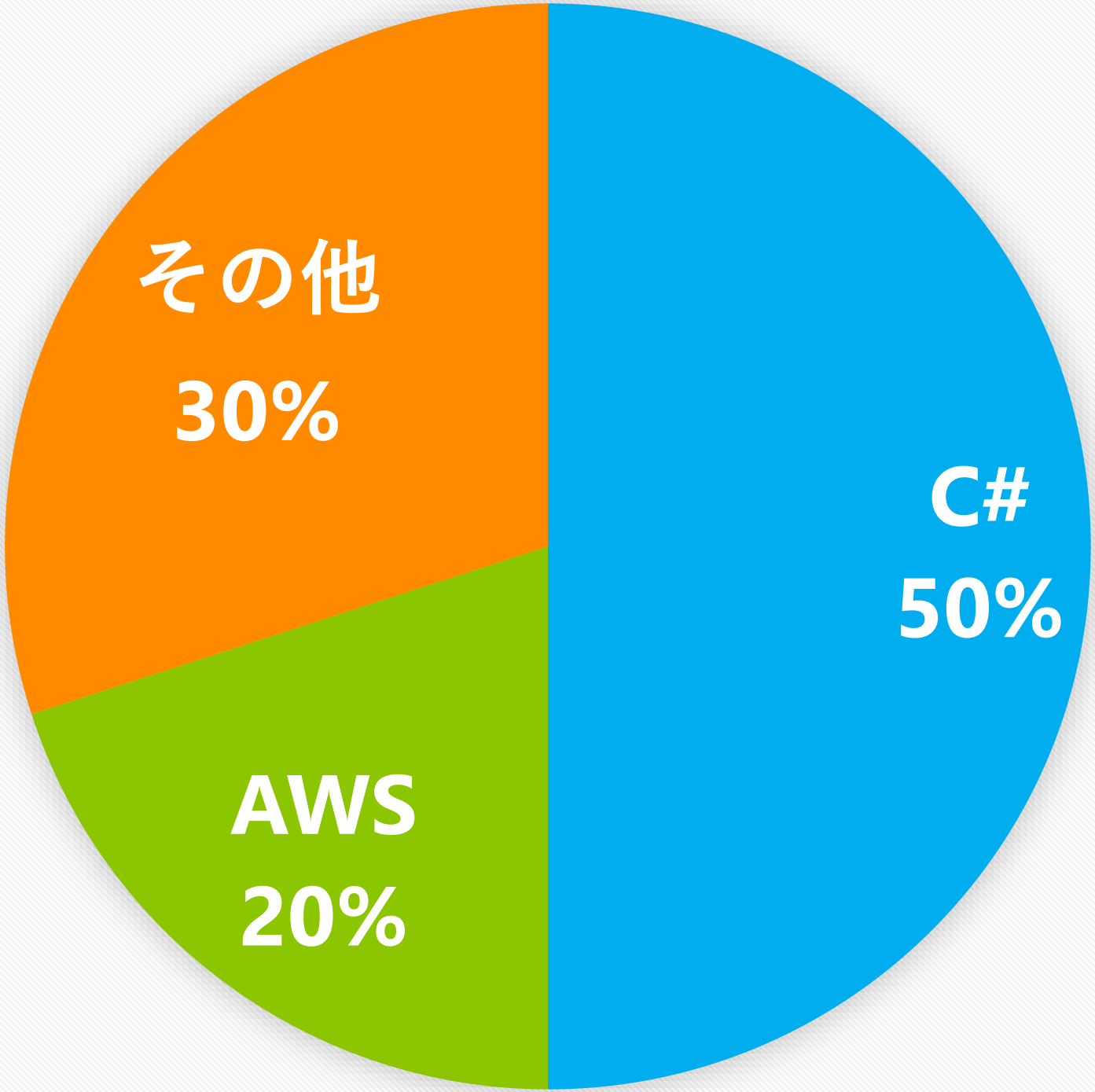
AWS+C#によるウェブソーシャルゲーム

弊社での使い方を、具体的な例で紹介します
特にDB/Redisに対するC#からのアプローチについて紹介します

汎用的

ソーシャルゲーム固有の話は極力しません
高負荷が求められるウェブアプリケーション構築のテクニック
また、一般的なアプリケーションでも十分役立つTips

これにより、日本ではまだあまり表には見えない
AWS + C#によるアプリケーションが増えてくれたら何よりも嬉しい！


その他
30%
C#
50%
AWS
20%

```
using CSharp;
```

C#での開発にとことんこだわる

何故C#？

リリース当時は**PHPだった**が開発効率に不満を感じた

C#の持つ強力な開発環境(Visual Studio)や先端的な言語仕様（非同期構文）、パフォーマンス、型安全、etc...

なによりも自分たちがC#をはじめとしたMicrosoft技術のエキスパートであること(Microsoft MVPが4人在籍)

リリース後わずか半年で**C#**に全面移行

インフラ的には、元々AWS上のAmazon Linuxなので、そのままスムースにWindows Serverに移ることができた

結果的に3~4倍のスループット向上、サーバー台数は1/3に減少

C# Everywhere

Windows

WinForms, WPF

Mac

Xamarin.Mac

Windows Tablet

Windows Store Application

Web Application

ASP.NET MVC/WebAPI, OWIN

Cloud

Microsoft Azure, AWS

Game

Unity, PlayStation Mobile SDK

Mobile

Xamarin.iOS
Xamarin.Android
Windows Phone 8 SDK

Embedded

Windows Embedded
.NET Micro Framework

NUI

Kinect, LeapMotion

C# Everywhere - Current

Windows

WinForms, WPF

Mac

Xamarin.Mac

Windows Tablet

Windows Store Application

Web Application

ASP.NET MVC/WebAPI OWI

Cloud

Microsoft Azure, AWS

内製ツール

ウェブソーシャルゲーム

Game

Unity, PlayStation Mobile SDK

Mobile

Xamarin.iOS
Xamarin.Android
Windows Phone 8 SDK

ded
.NET Micro Framework

NUI

Kinect, LeapMotion

C# Everywhere - Future

Windows

WinForms, WPF

Mac

Xamarin.Mac

Windows Tablet

Windows Store Application

Web Application

ASP.NET MVC/WebAPI, OWIN

Cloud

Microsoft Azure, AWS

Game

Unity, PlayStation Mobile SDK

Mobile

Xamarin.iOS
Xamarin.Android
Windows Phone 8 SDK

Embedded

Windows Embedded
.NET M[illegible] Framework

NU

Kin[illegible]

クライアントからサーバーサイドまで全てをC#で！

on AWS

AWS + C#

問題ない。

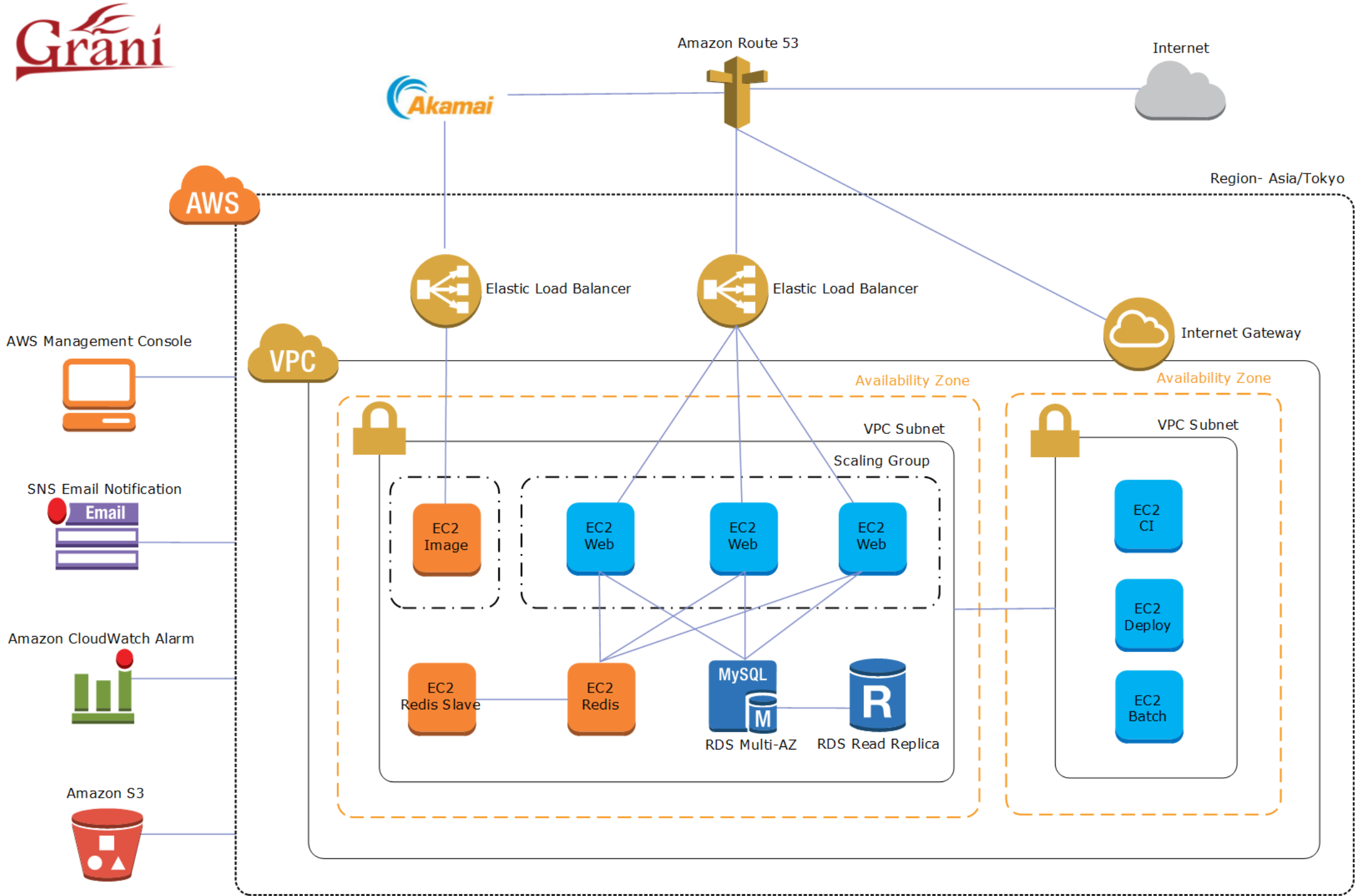
WindowsServer 2012、普通にあり、普通に使える
ライセンスはAWSがライセンス込み提供 or 持ち込み

C#によるウェブソーシャルゲーム開発

EC2上のWindows Server 2012
RDS上のMySQL(C#からMySQLも問題なく繋げられます)

現在の規模感

100 アプリケーションサーバー
10,000 リクエスト/秒
100,000,000 ページビュー/日


Grani
Amazon Route 53
Internet
Akamai
AWS
Region- Asia/Tokyo
Elastic Load Balancer
Elastic Load Balancer
AWS Management Console
VPC
Internet Gateway
Availability Zone
Availability Zone
VPC Subnet
VPC Subnet
Scaling Group
SNS Email Notification
Email
EC2
Image
EC2
Web
EC2
Web
EC2
Web
EC2
CI
Amazon CloudWatch Alarm
EC2
Deploy
EC2
Redis Slave
EC2
Redis
MySQL
M
R
EC2
Batch
RDS Multi-AZ
RDS Read Replica
Amazon S3

基本構成

IIS8 (EC2 Windows Server 2012)

IIS8 + .NET Framework 4.5 + ASP.NET 5
c1.xlarge x 50~150 (将来的にはRIでc3.2xlargeに変更予定)
+ バッチ処理サーバー x 1, 管理画面サーバー x1 (いずれもC#アプリ)

MySQL 5.6(RDS)

[r3.2xlarge ~ r3.8xlarge] x 7
機能別の垂直分割で、それぞれの台への負荷によってサイズを変えている

Redis(EC2 Amazon Linux)

[r3.2xlarge(Master) + r3.xlarge(Slave)] x 20
マスター：スレーブで1:1構成

Database

なぜRDBMSを選択するのか

NoSQLでいい？

DynamoDBなども推されてるし、無限にスケールして良いのでは？
けれど複雑化を抑えるため、シンプルな最小の構成にしたい
「DynamoDB + MySQL + Redis」になるぐらいなら「MySQL + Redis」

RDBMSの利点

レイテンシ： RDBMSは単純なクエリなら1ms以下で返すがDynamoDBは.....
DB用のIDEやプロファイラといった周辺ツールの充実具合
SQLを書いてのカジュアルな解析、アドホックなデータの弄りやすさ
スケールアップやシンプルな分割で対応できるならば、スケールアウトという利点に固執することもないので、RDBMSを使ったほうが楽になれる。性能に関しては、ヴァルハラゲートの規模でRDBMSで十分耐えている

何よりRDSが良い

RDSこそAWSを使う最大の理由！

手放し管理、Restore To Point In Timeには救われた
そろそろ性能限界？というタイミングに現れる上級インスタンス
ダメな時はスケールアップがある、と思える安心感は力強い
もし障害あってもサポートに駆け込めるのも嬉しい

SQL Server vs MySQL

C#/ASP.NETとしてはSQL Serverのほうが間違いなく相性はいい
しかし、RDSとして見るとSQL Serverは貧弱
Multi-AZもなかったし(2014/05/19から可能！)
Provisioned IOPSの限界も低い

AWS + C#の企業としては、
RDSとしての良さのほうを選ぶ

豊富なインスタンスタイプ

r3.8xlargeったら最強ね！

弩級の32vCPU, メモリ244GiB、こいつにしとけば何とかなる感半端ない
しかし、さすがに一台だけでは耐え切れなかったりする（主にCPU）

インスタンスタイプ	vCPU	メモリ（GiB）	PIOPS 用に最適化	ネットワークパフォーマンス
スタンダード – 現行世代				
db.m3.medium	1	3.75	–	中
db.m3.large	2	7.5	–	中
db.m3.xlarge	4	15	はい	中
db.m3.2xlarge	8	30	はい	高
メモリの最適化 – 現行世代				
db.r3.large	2	15	–	中
db.r3.xlarge	4	30.5	はい	中
db.r3.2xlarge	8	61	はい	高
db.r3.4xlarge	16	122	はい	高
db.r3.8xlarge	32	244	–	10 ギガビット

垂直分割 vs 水平分割

機能単位の**垂直分割**を採用

ユーザー情報/ギルド情報/バトル情報、みたいな分割（現状7分割）

負荷は異なるのでインスタンスサイズは調整（全部r3.8xlargeではない）

制限1：テーブルが物理的別DBに別れるため、外部キーは張れなくなる

制限2：DBを超えたジョインが不可能といった、クエリに若干の制限はかかる

水平分割は避ける

一般的にシャーディングで用いられるのはテーブル内のID単位で分けるなどする水平分割で、利点は無限にスケールすることだが、お薦めしない

垂直分割での制限に加えて、同一テーブル内の情報すら不完全になり、記述可能なクエリに更に大きな制限がかかってしまう

アドホックなクエリでの集計が不可能になる

（後述する）Tableauなどのデータ分析ツールとの相性も悪化

垂直分割 vs 水平分割

機能単位の**垂直分割**を採用

ユーザー情報/ギルド情報/バトル情報、みたいな分割（現状7分割）

負荷は異なるのでインスタンスサイズは

制限1：テーブルが物理的別DBに別れる

制限2：DBを超えたジョインが不可能と

ヴァルハラゲートレベルの負荷でも、垂直分割で耐えられているので(r3.8xlargeは素晴らしい)、ほとんどのアプリケーションは、大きなデメリットを抱える水平分割する必要性はないのでは？

水平分割は避ける

一般的にシャーディングで用いられるのはテーブル内のID単位で分けるなどする水平分割で、利点は無限にスケールすることだが、お薦めしない

垂直分割での制限に加えて、同一テーブル内の情報すら不完全になり、記述可能なクエリに更に大きな制限がかかってしまう

アドホックなクエリでの集計が不可能になる

（後述する）Tableauなどのデータ分析ツールとの相性も悪化

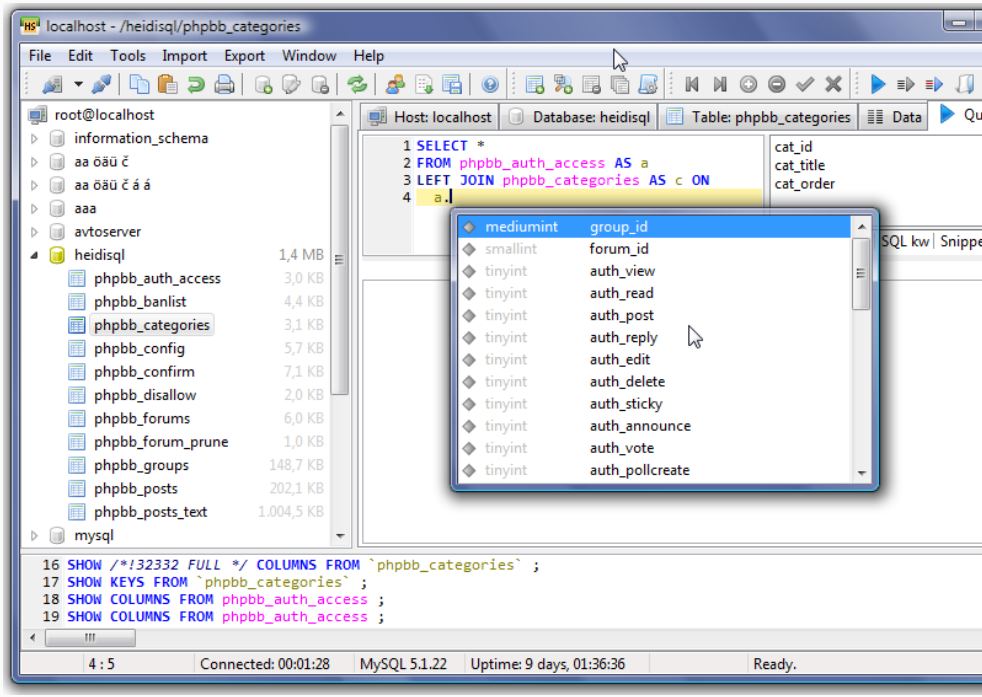
水平分割を避けたい100億の理由

DB管理はGUIツールを使いたい

HeidiSQLがお薦め、クエリ書きからテーブル定義、インデックス張り、データエクスポートなど全面的に活用

phpMyAdminやCUIは積極的にdisりたい

DBもIDEの時代



水平分割はツールと相性最悪

自社ツールの作り込みをしなければならず、
そしてそれは既存のよくできたGUIツールよりも遥かに使いにくいだろう

グラニのRDS活用のポイント

というほどでもない

アプリケーションからはMasterのみ参照

イマドキのRDSは十分パワフル、Slaveにクエリを分散するのは無用な複雑さを産むだけ（レプリケーション遅延はとても怖い！）

ただし管理画面からの分析クエリ発行などのために、Read Replica自体は用意している（耐障害性はMulti-AZがあるので、分析用だけなのは少しmottainai)

同一AvailabilityZoneへ配置する

アプリケーションサーバーとRDSのAZが異なるとレイテンシがある

実際トラブル対応等で一時的にZoneが変わった時に、 クエリ一発に、普段は0.5ms程度のものが全て2ms以上かかる程度に遅延が上乗せされた

アプリケーションに致命的な打撃を与えるほどではないが、性能にこだわるなら、ここはこだわっておきたい

C#的な活用

物理的に台が異なるので、コネクションを型で分ける

```
public interface ITypedConnection : IDisposable
{
    DbConnection Slave { get; }
    DbConnection Master { get; }
}

public BattleEntity SelectById(BattleConnection battle, int id)
{
    return battle.Master.Query<BattleEntity>("select * from battle where id = @id", new { id });
}

public UserEntity SelectById(UserInfoConnection user, int id)
{
    return user.Master.Query<UserEntity>("select * from user where id = @id", new { id });
}
```

コーディング時のミス防止（間違った接続の利用はコンパイル時に弾かれる）
テーブルの別DBへの分割時にも完全にコンパイルチェックが効くので安全に行える
C#（というか型付き言語）を使う利点

Micro-ORM

クエリは生SQLを手書き

垂直分割で分かれてるためO/Rマッパーと相性が悪い

勿論、パフォーマンス的な問題もある

Dapper

DataRow => Objectへの変換を行うライブラリ

https://code.google.com/p/dapper-dot-net/

```
connection.Query<Dog>("select * from dogs where id = @id", new { id = 100 })
```

高速だし、使いやすい（へたにO/Rマッパー使うよりよほど）

ただしMySQL + Dapperだと不具合があるので要修正

詳しくは： http://neue.cc/2013/08/06_423.html

テーブル定義からの自動生成

テーブルと1:1で関連づいたクラス

型安全のためには当然必要、だけど膨大な数が必要！
ヴァルハラゲートは500テーブル以上ある

T4テンプレート + ADO.NET GetSchema

Visual Studio同梱のテキストテンプレートで雛形を作る
ADO.NETのGetSchemaでカラム/インデックスなどの各種情報が取れるので、突きあわせてクラスを生成するプログラムを作成
(一度生成したクラスは手動で変更を加えるのでT4はランタイムテンプレート)

#	Name	Datatype
1	**GuideId**	**INT**
2	**No**	**INT**
3	Title	VARCHAR
4	Body	VARCHAR
5	LinkController	VARCHAR
6	LinkAction	VARCHAR
7	Priority	INT
8	NaviId	INT
9	NaviPattern	INT
10	Created	DATETIME
11	Modified	DATETIME

Int → Enumへの変換などは生成後、手で修正している。半自動生成、初回の雛形作成、というぐらいの位置づけ

```
[Serializable]
[DataContract]
public class GuideTemplateMaster
{
    [DataMember(Order = 1)]
    public GuideCode GuideId { get; private set; }
    [DataMember(Order = 2)]
    public Int32 No { get; private set; }
    [DataMember(Order = 3)]
    public String Title { get; private set; }
    [DataMember(Order = 4)]
    public String Body { get; private set; }
    [DataMember(Order = 5)]
    public String LinkController { get; private set; }
    [DataMember(Order = 6)]
    public String LinkAction { get; private set; }
    [DataMember(Order = 7)]
    public Int32 Priority { get; private set; }
    [DataMember(Order = 8)]
    public NavicoCharacter NaviId { get; private set; }
    [DataMember(Order = 9)]
    public NavicoFaceType NaviPattern { get; private set; }

    public override string ToString()
    {
        return ""
            + "GuideId : " + GuideId + "|"
            + "No : " + No + "|"
            + "Title : " + Title + "|"
            + "Body : " + Body + "|"
            + "LinkController : " + LinkController + "|"
            + "LinkAction : " + LinkAction + "|"
            + "Priority : " + Priority + "|"
            + "NaviId : " + NaviId + "|"
```

全階層でのキャッシュの徹底

永久に保存する領域 – データベースなど

期間保存 – Memcached/RedisなどExpire付き

アプリケーション単位 – Static変数

リクエスト単位 - HttpContext.Items

参照クエリの負荷軽減・分散

アプリケーション起動時にマスタデータなど不変のものは全て詰める

1リクエスト中で使え回せそうなものは積極的に格納

やはりボトルネックになるのはDBなので、あらゆる層で片っ端からキャッシュする

全階層でのキャッシュの徹底

永久に保存する領域 – データベースなど

期間保存 – Memcached/RedisなどExpire付き

アプリケーション単位 – Static変数

リクエスト単位 - HttpContext.Item

一番想像がつきにくいのはこの部分では？

マスタデータのキャッシュ化

アイテム名など不変の情報はキャッシュ

どこに？不変ならばstaticのdictionaryに入れてしまえばいい

静的コンストラクタで一気に生成している

1:1はDictionary、1:他はILookupに格納

キャッシュ用コードもインデックス見て使い方が判別できるものも、テーブル定義と一緒に自動生成してしまう

（マスタじゃない普通のテーブルに関しても、インデックスを見てデータベースアクセサの雛形は自動生成してます）

```
public static class GuideTemplateMasterCache
{
    public static readonly ReadOnlyCollection<GuideTemplateMaster> All;
    public static readonly ReadOnlyDictionary<Tuple<GuideCode, Int32>, GuideTemplateMas
    public static readonly ILookup<GuideCode, GuideTemplateMaster> ByGuideCode;

    static GuideTemplateMasterCache()
    {
        using (var connection = new GeneralConnection())
        {
            All = connection.Master.QueryEnumerable<GuideTemplateMaster>(
                "select * from GuideTemplateMaster").ToList().AsReadOnly();
        }
        ByGuideIdAndNo = All.ToDictionary(x => Tuple.Create(x.GuideId, x.No)).AsReadOnl
        ByGuideCode = All.ToLookup(x => x.GuideId);
    }
}
```

アプリケーションコードによるJoin

結合はアプリケーション側で

マスタデータはインメモリにあるので、DictionaryやLookupと突き合わせるだけで、ゼロコストで結合可能

LINQ to Objectsで簡単

.Select(x => a = dict[x.id], b = lookup[x.code])

といったように、LINQのSelectで簡単に結合可能、むしろSQL書くよりもずっと楽

あとはデータベースの作りを非正規化したりすることで、ジョインしなくても大丈夫な構造になる。

これで分割耐性もあり、また、負荷に耐えうるDB定義ができる

Redis

About Redis

インメモリKey-Valueストア

KVS系では現在、最も人気が高い
単純なKey-Valueの他にListやSetなど多様なデータ型が使える
リモート上のデータ構造という、いい具合のサイズ感が魅力

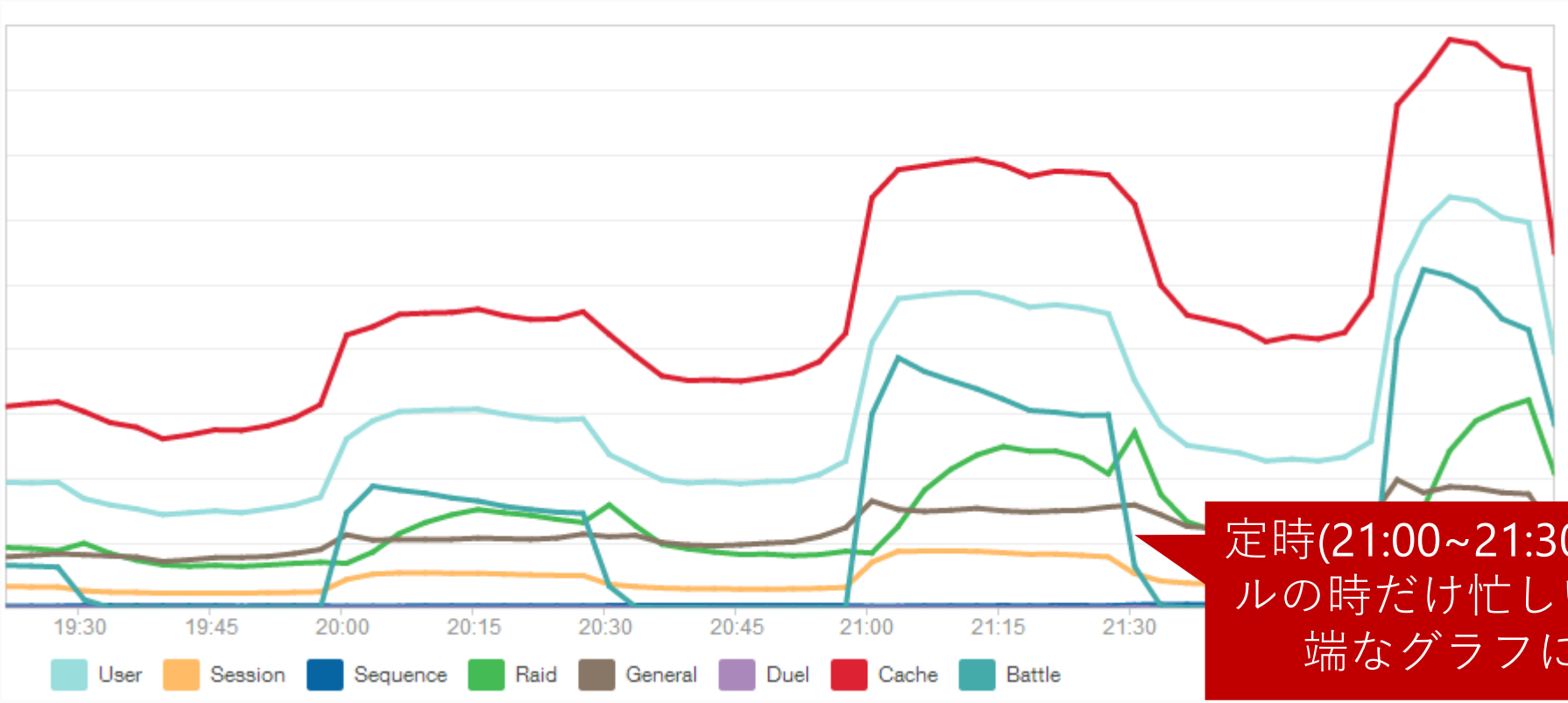
RDSの不得意な部分を補える

高パフォーマンスなのでMemcached代わりのキャッシュ用途に
豊富なデータ型やLuaスクリプトを活用した少し高度な処理

Distributed Strategy

用途毎のグループ分けと単純分散

Redisは多種多様に使っているので、用途ごとにグループ分けしてある
グループ内に複数台含めて、更に単純なキーのハッシュ値で分散



定時(21:00~21:30, 22:00~22:30など)開催のバトルの時だけ忙しいがそれ以外はほぼ0という極端なグラフになるBattleグループ、など

Master-Slave

Slave

RDB方式のダンプ出力時にCPUリソースを喰い性能低下
これが元でピーク時にレスポンス悪化
ダンプ実行をSlaveのみにすることで解消
また、もちろんSlaveは障害時の昇格用でもある

vs ElastiCache Redis

グラニのRedisは現在、EC2上に立てているもの
初期にElastiCache Redisを試した際に、EC2よりパフォーマンスが出なかったので一旦なしにした

Serialization

Protocol Buffers

データ格納の際のシリアライズ方式はProtocol Buffersを採用

JSONよりも高速・省サイズ

.NET実装のprotobuf-netを利用している

Speed?

シリアライザのスピードはフォーマットだけでは決まらない、どちらかというと実装で差が出る

.NET実装は非常に優秀で、シリアライズ/デシリアライズが高速

protobuf-netはStack Overflow(Windows/ASP.NET MVCでできている)のエンジニアの手によるもの

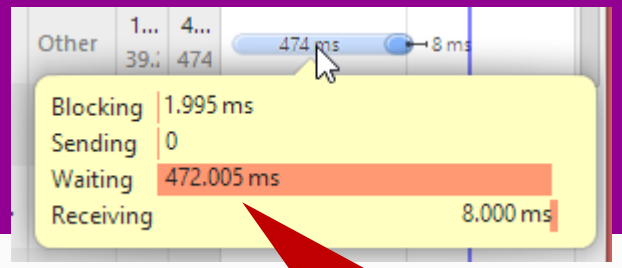
Performance + Async

パフォーマンスへのこだわり

Time To First Byte軽視されすぎでは？

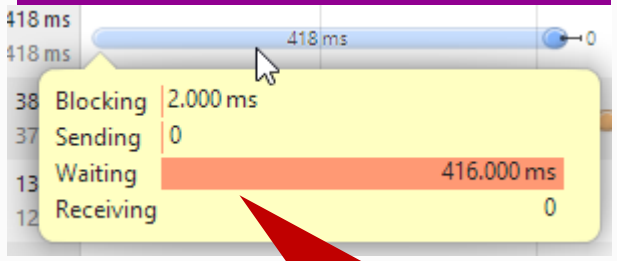
50ms違うだけで体感にかなり影響ある

とある他社ゲーム



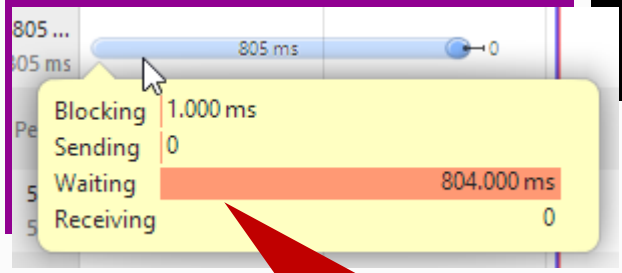
472ms

とある他社ゲーム

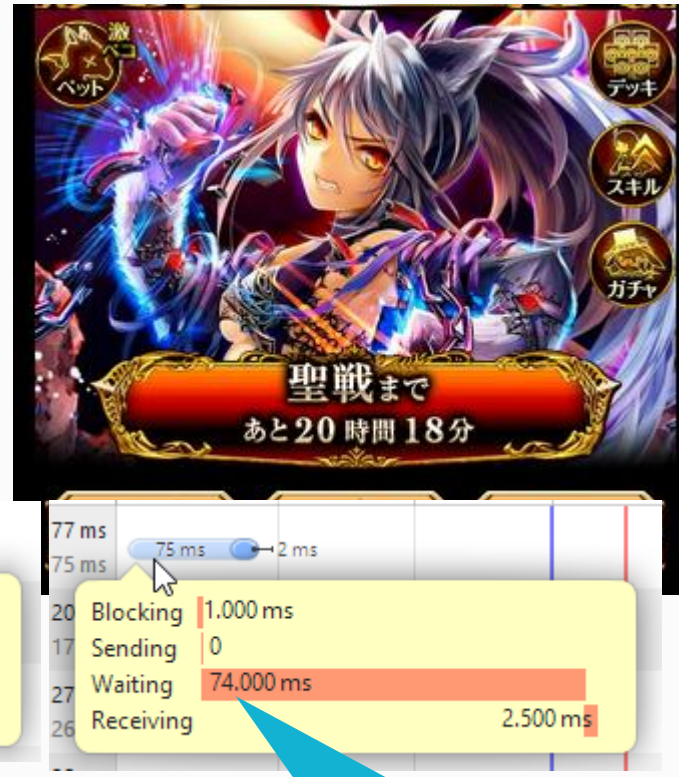


416ms

とある他社ゲーム



804ms



74ms

を、支えるC# 5.0

言語構文レベルでサポートされる非同期

同期と同じような形で非同期が書ける、Graniではコード全体が非同期祭り

```
var names = Members.Select(x => new
    {
        Name = x.GetName()
    })
    .ToArray();

var names = await Members.Select(async x => new
    {
        Name = await x.GetNameAsync()
    })
    .WhenAll();
```

Membersが10人だとして、GetNameが2msかかると、同期だと10 * 2 = 20ms

非同期で一気に同時に取得すれば2ms で済む

単純なものなら単純に書ける、けど

```
// 例えばmemcachedの場合
var memcached = new MemcachedClient();

// 3回アクセスがあって辛ぽよ
var a = memcached.Get("hoge"); // +10ms = 10ms
var b = memcached.Get("hage"); // +10ms = 20ms
var c = memcached.Get("huga"); // +10ms = 30ms

// 1度に問い合わせて、分配
var all = memcached.Get(new[] { "hoge", "hage", "huga" }); // +10ms
var a2 = all["hoge"];
var b2 = all["hage"];
var c2 = all["huga"];
```

別に非同期構文とかなくてもできるじゃん！？

単純なものなら単純に書ける、けど

```
// 例えばmemcachedの場合
var memcached = new MemcachedClient();

// 3回アクセスがあって辛ぽよ
var a = memcached.Get("hoge"); // +10ms = 10ms
var b = memcached.Get("hage"); // +10ms = 20ms
var c = memcached.Get("huga"); // +10ms = 30ms

// 1度に問い合わせて、分配
var all = memcached.Get(new[] { "hoge", "hage", "huga" }); // +10ms
var a2 = all["hoge"];
var b2 = all["hage"];
var c2 = all["huga"];
```

でもIncrとか、Get以外のコマンドは？
それに、こうしたコードってオブジェクトモデルでまとめにくい！
性能優先 vs 設計優先の対立になるの？

ところがRedisなら……!

全コマンドがパイプライン化可能

例えばクライアントがRedisのINCRコマンドを4回呼ぶ

Client: INCR X

Server: 1

Client: INCR X

Server: 2

Client: INCR X

Server: 3

Client: INCR X

Server: 4

Client: INCR X, INCR X, INCR X, INCR X

Server: [1, 2, 3, 4]

Client-Server間で4回の応答待ちが発生

パイプラインで呼ぶと、全部まとまってコマンド飛ばせるので往復遅延時間が削減

C# + Redisなら……!

全てが非同期で自動でパイプライン化される

全リクエストでRedisへのコネクションを共有し、リクエストタイミングが近いものが勝手にパイプライン化されてまとめて送られる

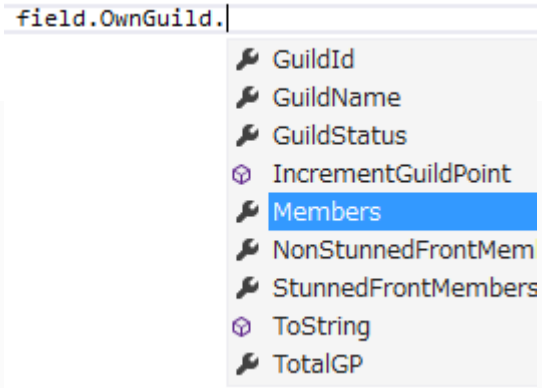
```
var a = redis.TryGet("hoge"); // Taskなのでひどぅーき
var b = redis.TryGet("huga");
var c = redis.TryGet("hage");

await Task.WhenAll(a, b, c); // 10ms
```

性能と設計を両立する

```
var frontHPs = await field.OwnGuild.Members
    .Where(x => x.Position == Position.Front)
    .Select(async x => new
    {
        Name = await x.Name,
        CurrentHP = (await x.UserStatus).CurrentHP
    })
    .WhenAll();
```

x.Nameやx.UserStatusは**Redis**への通信、こうして書いたコードは、**自動的にパイプライン化**されて非同期実行されている



LINQと相性良い、IntelliSensable超大事
そうしたLINQableのための設計と性能が両立できる

```
// 自分の実行可能(TP不足じゃないとか)なアビリティをActionTypeでグループ分け
var abilities = (await field.OwnStatus.GetCommandAbilities())
    .Where(x => x.CanExecute == CanExecuteReason.CanExecute)
    .GroupBy(x => x.ActionType);
```

Log for Performance

Log Everything from Application

外部通信(HTTP, SQL, Redis)を全て記録する

アプリケーション側で送信前後をフックして記録
→後述するGlimpseやSumo Logicで解析する

アプリ側から行う利点

負荷がない（DBでプロファイラONなどは常用できない）
記録できないものがない、全て記録可能
付随する情報も含めて多くを記録できる

コード詳細は http://neue.cc/2013/07/30_420.html で。

Log - HTTP

```
public class HttpProfilingHandler : DelegatingHandler
{
    static readonly Logger httpLogger = NLog.LogManager.GetLogger("Http");

    public HttpProfilingHandler() : base(new HttpClientHandler()) { }

    public HttpProfilingHandler(HttpMessageHandler innerHandler) : base(innerHandler){ }

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, System.Threading.CancellationToken cancellationToken)
    {
        // 通信の前後をStopwatchで測る
        var sw = Stopwatch.StartNew();
        var result = await base.SendAsync(request, cancellationToken).ConfigureAwait(false);
        sw.Stop();

        // 以下に好きなようにログ仕込む、例えばJSON化
        httpLogger.Trace(ApplicationPerformanceLog.ToJson(
            DateTime.Now, request.Method.ToString(),
            request.RequestUri.ToString(),
            sw.ElapsedMilliseconds));

        return result;
    }
}
```

HttpClientに対してDelegatingHandlerを挟むことで処理の前後を簡単にフックできる

new HttpClient(new HttpProfilingHandler());

Log - SQL

MiniProfilerに用意されている
IDbProfilerをカスタムして
,ADO.NETのコネクションとして
使うことで自由に仕込める

var conn = new
ProfiledDbConnection(new
SqlConnection(), new
LoggingDbProfiler());

```
public class LoggingDbProfiler : IDbProfiler
{
    // 中略

    // コマンドが完了された時に呼ばれる
    public void ExecuteFinish(System.Data.IDbCommand prc
                              ExecuteType executeType, S
    {
        commandText = profiledDbCommand.CommandText;
        if (executeType != ExecuteType.Reader)
        {
            stopwatch.Stop();
            sqlLogger.Trace(Newtonsoft.Json.JsonConvert.
            {
                date = DateTime.Now,
                command = executeType,
                key = commandText,
                ms = stopwatch.ElapsedMilliseconds
            }, Newtonsoft.Json.Formatting.None));
        }
    }
}
```

Log - Redis

CloudStructures(自社製のRedisライブラリ)に用意されてるプロファイラの口に通すことで、送った/受け取ったオブジェクトなどがモニタできる

```
public class RedisProfiler : ICommandTracer
{
    static readonly Logger redisLogger = NLog.LogMar

    Stopwatch stopwatch;
    RedisSettings usedSettings;

    public void CommandStart(RedisSettings usedSetti
    {
        this.usedSettings = usedSettings;
        stopwatch = Stopwatch.StartNew();
    }

    public void CommandFinish(object sentObject, obj
    {
        stopwatch.Stop();
        var ms = (long)System.Math.Round(stopwatch.E
        redisLogger.Trace(ApplicationPerformanceLog
              .ToJsonWithHost(DateTime.Now,
                usedSettings.Host, command,
                key, ms));
    }
}
```

Glimpse – 可視化

記録したら簡単に見れなければならない

誰が？インフラ管理者？ 違う！

アプリケーション開発者が**常に**見れなければ改善に繋がらない

Glimpse

ASP.NET向けの可視化ソリューション

http://getglimpse.com/

同様のものにMiniProfilerがあるが、MiniProfilerはログ吐きのコネクションのフックのためだけに使って表示系は使用してない）

Sql/Http/DBへのフックは、Glimpseにも表示されるよう拡張


全体の実行時間のほか、
あらゆるメトリクスを
常時可視化
聖戦まで
あと20時間18分
クエスト
レイド
強化合成
"Chrome 33.0" http://localhost:8080/Ba 'MonsterBattleField?rnd=3959386
New update available!
Configuration
Environment
Execution
Log
atform API
Redis
RedisInfo
Request
Routes
SQL
SQL Explain
Session
Timeline
Views
Ajax
History
Categories [Switch view]
Common
Filter
Redis
Controller
Command: Executed ...
RedisClass.GetValue: ...
RedisSet.Add: Green....
RedisSet.SetExpire: G...
RedisString.TryGet: G...
RedisSet.GetAll: Gree...
RedisString.TryGet: U...
目に見えてRedis並列実行

Explainの常時表示(独自拡張タブ)

開発環境上では、常に全SQL発行に対して、explain結果も出すようにしている（手動でやるようだと絶対にやらないので、機械的に自動でやって、常に見えるところに置かなければならない）

明らかにヤヴァそうなもの(Using filesortとか)は警告する。これにより、開発者が「開発中」に、自分で気づけるように誘導

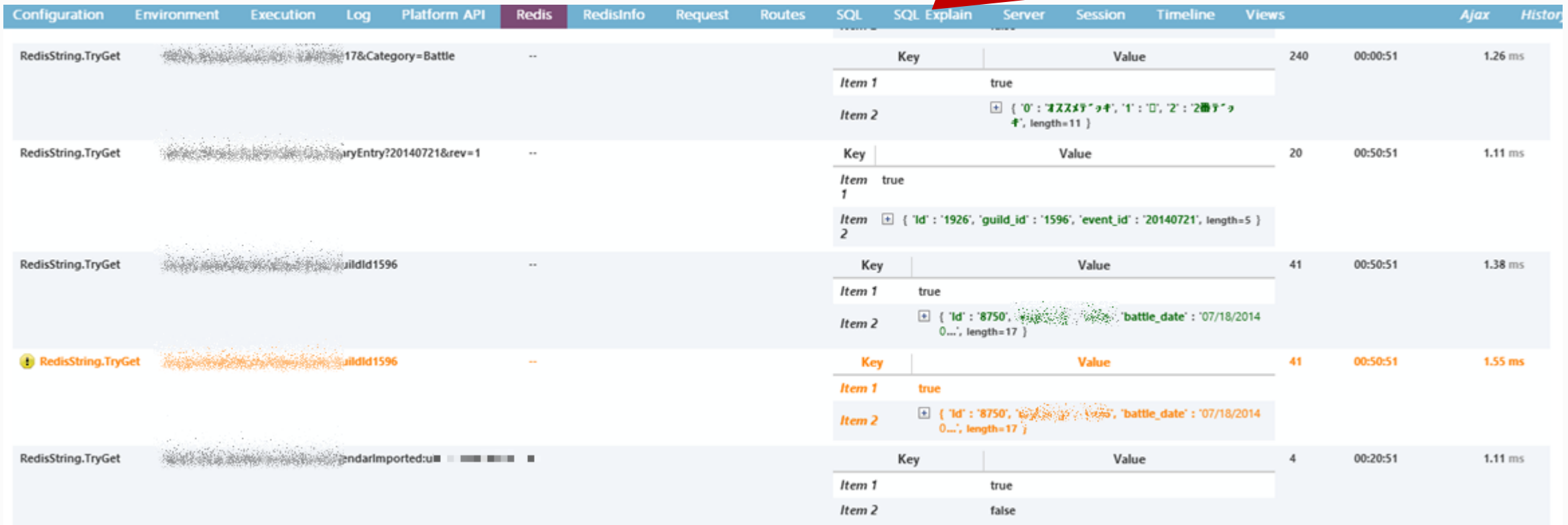
"Chrome 35.0" http://localhost:6060/Battle/Index?_from=gheader&rnd=3182346 New update available!

Configuration | Environment | Execution | Log | Platform API | Redis | RedisInfo | Request | Routes | SQL | SQL Explain | Server | Session | Timeline | Views | Ajax | History

query	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
explain select * from [redacted] 1596 order by entry_id	SIMPLE	[redacted]	ref	uniq,rank,guild_id,[redacted]	uniq	4	const	1	Using index condition; Using where; Using filesort
explain select * from [redacted]	SIMPLE	[redacted]	const	PRIMARY	PRIMARY	4	const	1	--
explain select * [redacted] user	SIMPLE	[redacted]	ref	[redacted]	[redacted]	8	const	230	Using where
explain select * from [redacted]	SIMPLE	[redacted]	ref	[redacted]	[redacted]	4	const	1	Using where
explain select * from [redacted]	SIMPLE	--	--	--	--	--	--	0	Impossible WHERE noticed after reading const tables
explain select * from [redacted]	SIMPLE	--	--	--	--	--	--	0	Impossible

Redis送受信の表示(独自拡張タブ)

・同一キーの重複時警告
・送信、受信オブジェクトのダンプ
・オブジェクトサイズ
・Expire残り時間
・通信時間
などの表示



Workflow

日常的開発

Git + GitHub(Business)

ツールはSourceTree for Windows + Visual Studio Tools for Git

PullRequestはしたりしなかったり

Jenkins

GitHubからビルドしてバイナリ生成/開発環境への自動反映

Deploy

自社製デプロイツール Valentia + MSDeployでバイナリ配布

https://github.com/guitarrapc/valentia

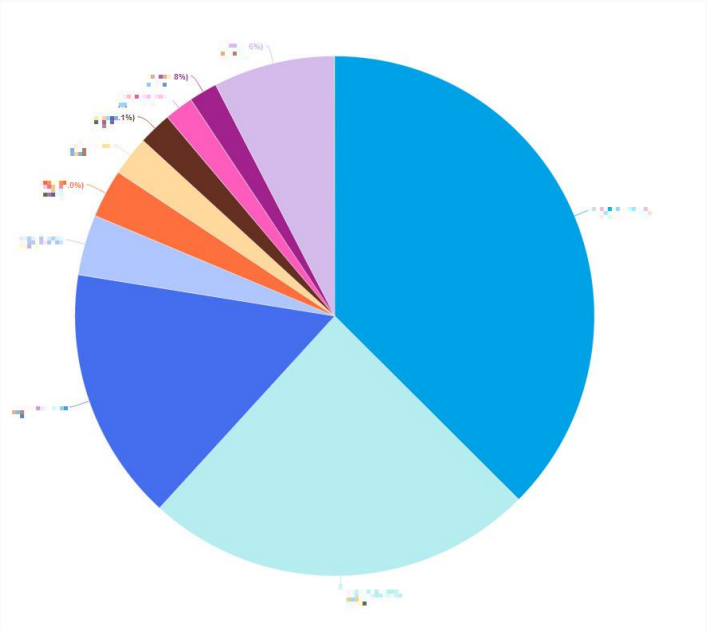
PowerShellによるCapistranoライクなデプロイツール

Sumo Logic

Next Generation Log Management & Analytics

ログをクエリ

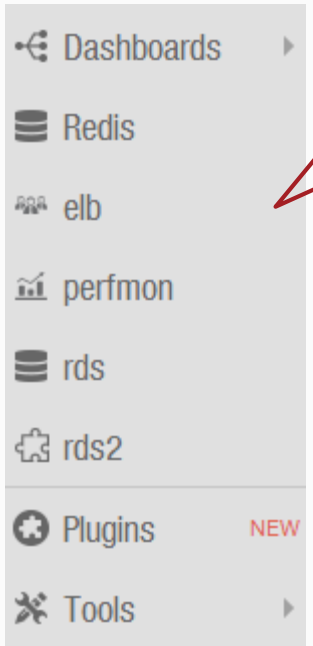
サーバーログ、リクエストログ、アプリログの他に、
全SQL,全HTTP通信,全Redis発行を集積してクエリ可能に



New Relic

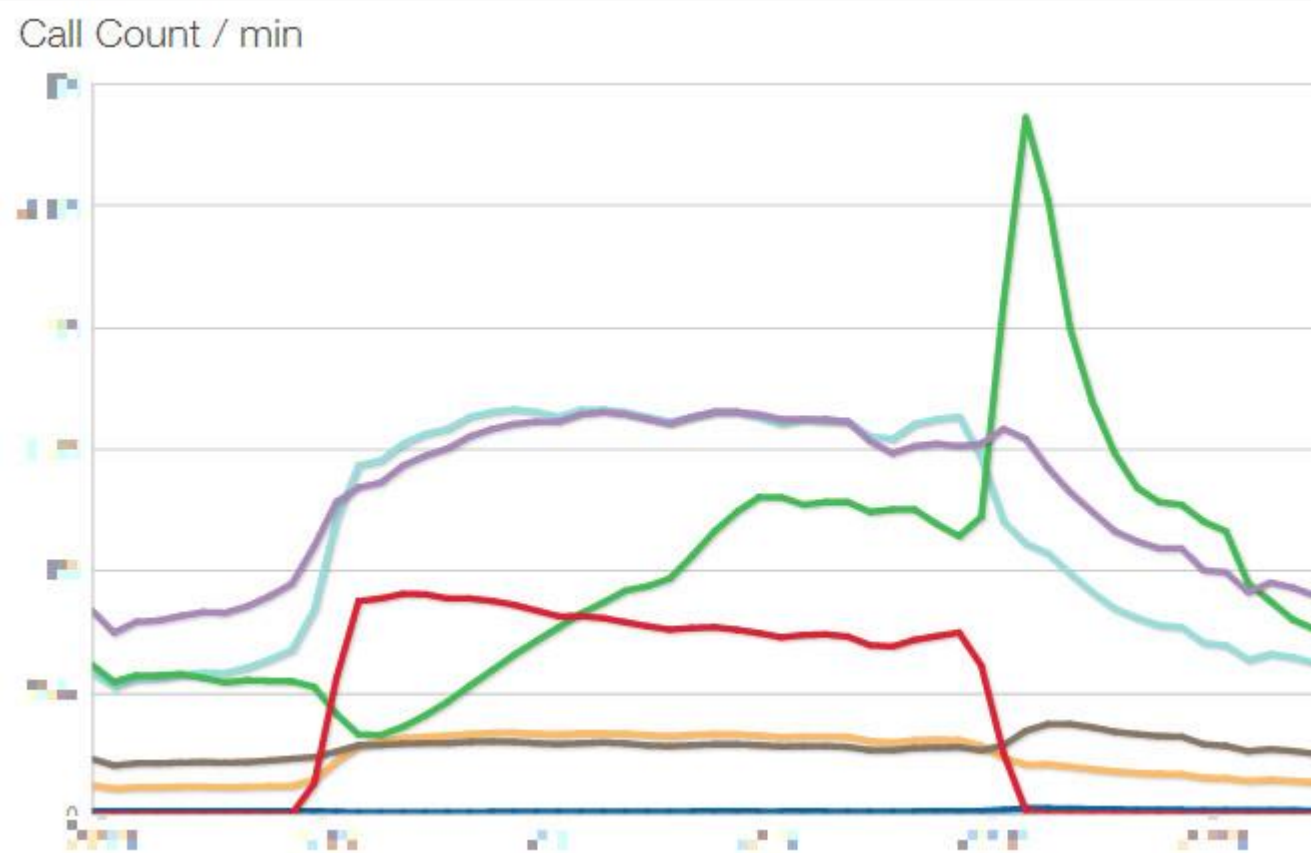
最高のモニタリングSaaS

アプリ開発者は常に全員見る
窓口として重要な情報は集約させる



自社製プラグインでPerformance Counterなどの情報も集積・表示

SDKを叩いてアプリ側からRedisの利用具合を可視化



Analytics

Semantic Logging + RedShift + Tableau

アプリケーション分析のためのロギング

WindowsではTreasureDataやFlyDataは動かない
せっかくAWSなのだから、RedShiftは是非使いたい

Semantic Logging Application Block

https://slab.codeplex.com/
構造化ログライブラリと常駐サービス
アプリケーション(SLAB Logger)→SLAB Service→S3→RedShift

Tableau

データ可視化・RedShiftやRDS(MySQL)に繋がる
特にMySQLでは垂直分割によって直接扱うことができる

Semantic Logging + RedShift + Tableau

アプリケーション分析のためのロギ

WindowsではTreasureDataやFlyDataは利

せっかくAWSなのだから、Red

Semantic Loggin

https://slab

構造化

ビス

アプ

Logger)→SLAB Se

RedShift

Table

データ可視化・RedShiftやRDS(MySQL)に繋がる

特にMySQLでは垂直分割によって直接扱うことができる

現在構築中！（まだ投下してません）

とかやってたら、東京リージョンに来たKinesisがきになるぅぅぅ！

Conclusion

まとめ

C# + AWSは現実解

高負荷が求められる環境でも十分以上に捌ききれる

AWSだからといって特別なことはなにもない、そこがいい

構成は堅く、シンプルに

シンプルな構成が一番取り回しが良い

SaaSもフル活用し、自分たちで抱えすぎない

環境は常に最新に

言語(C#)も環境(AWS)も常に進化し続け、大きな利益がある

最新を維持することこそが、難しいことだが、重要


Grani