コミュニティー・ストーリー

ドワンゴにおける Play と Akka を用いたニコ生のスケール方法

吉村 総一郎

dwango

niconicoについて

niconico は、日本の代表的な動画共有サイトの一つです。サービス開始時の2006年は、ニコニコ動画(Niconico Douga)という名称でした。特徴的な機能として、配信される動画の再生時間軸上にユーザーがコメントを投稿できる機能があります。動画以外のコンテンツにもコメントが付けることが可能で、ライブストリーミングや、電子書籍といったものにもコメントを投稿することができます。

niconicoのサービスの規模としては、2015年8月時点で、登録会員数5,000万人、プレミアム会員数250万人です。サイトのアクセスの規模としては、2015年3月時点でniconico全体では、1億7,862万PV、月間平均訪問者数880万となっています。

niconicoを開発している株式会社ドワンゴでは、10を超えるプロダクトやサブシステムがScalaで開発されていますが、この記事ではニコニコ生放送(Niconico Live)というライブストリーミングサイトの大規模視聴システムの紹介をします。

ニコニコ生放送(Niconico Live)がかかえていた問題

ニコニコ生放送は、主に2つの問題を抱えていました。

- 大規模視聴におけるパフォーマンスの問題
- メンテナンスコストが高いコード

以上2つです。以下でそれぞれ説明していきます。

大規模視聴におけるパフォーマンスの問題

パフォーマンスに関する問題は、以下のCPUとIOに関するものがそれぞれありました。

- 高いCPU負荷が発生するサービスの特性
- 構造的にスケールしないデータストアのIO

高いCPU負荷を引き起こす要因となっているのが、シビアな番組開始時間とプレミアム会員を優先して見せるための視聴権管理システムでした。

番組開始時間に関しては、番組開始の30分前からストリーミングを開始できる時間を用意することである程度は負荷軽減はできるのですが、それでもアクセスのスパイクを避ける事はできません。

また視聴権管理システムに関しては、数十万人を超える視聴者がポーリングで視聴権があるかどうかをシステムに問い合わせる仕組みとなっており、これにも高いCPUコストがかかっていました。

そして、構造的にスケールしないデータストアのIOの問題に関してですが、これは視聴権管理システムでの視聴権確認や更新が、RDBで複雑なSQLのクエリを発行することで　実装されていることに由来しました。　このようなにディスクIOを利用するRDBに依存した視聴権管理をしている限りは、これ以上のスケールアップもスケールアウトもさせることができないことがわかっていました。

メンテナンスコストが高いコード

もう一つの問題は、コードのメンテナンス性です。　様々な改善を行った後の今なお100万行を上回るPHPのコードベースで、大量のコピー＆ペーストによる暗黙のブランチコードにより、　ひとつの修正を入れた際の影響範囲の調査や検証が非常に高コストになっています。

またひとつのメソッドを見ても、単体テスト不在で循環的複雑度が600を超えるようなメソッドがある上に、　基本的なデータがクラスなどの型で定義された状況ではなく、PHPのarrayという連想配列で定義されていたため、　コードを読むためのコストや、必要とされる知識量が非常に大きくなっていました。

またチームとしてコードベースの負債に対して意識をフォーカスしなかった結果、退職による人材の流出を招き、　組織のドメイン知識の欠乏とリーディングコストの高いコードベースが残された状態になっていました。

ニコ生新プレイヤープロジェクト（Relive Project)の目標

このような問題を解決すべく、以下のような目標を立て新たなコードベースで　ニコ生新プレイヤー(Relive)という名前のプロジェクトをを開始させました。　当初の目標は、

- 大規模な同時視聴に耐える
- 既存の仕組みを引き継がない
- 柔軟で安定した視聴権管理
- 新たな機能を加えられる拡張性

としました。

開発の戦略

この目標を達成するための基本的な開発の戦略として、

- スモールリリース
- 費用対効果の高い機能からリリース

という方針としました。

スモールリリースとして、最初にniconicoのトップページに表示される　ニコニコ生放送のFlashのプレイヤーをフロントエンドにするシステムから開発し始め、　4ヶ月間の開発の後、2013年10月に本番環境の負荷テストと番組の実施を行いました。

またニコニコ生放送自体が、2007年にリリースされてから既にその当時で6年も経過しており、非常に多くの機能を持っている状態でした。

そのため、大きなリリースで一気に全ての機能を作りきるのは断念し、　大規模イベント向けの番組に絞った機能から少しずつリリースをしていきました。

最初のうちは、実際の番組放送に耐えられるだけの機能がなかったため、　ショーケース環境という社内で関係者が確認できる環境を用意し、　その環境に対してリリースした機能を、社内の関係者全員に発表を行っていくという方法でおこなっていきました。

システム構成

利用したライブラリは、

- Play/Scala
- Akka Cluster
- Slick

どれも可能な限り最新のバージョンを利用しています。

Scala/Akka/Playを採用した理由は、アーキテクチャの要件が

- スケーラビリティ
- 安定性・フォールトトレラント性
- 拡張性
- パフォーマンス

と、上記の優先順位であったためです。特にスケーラビリティ(スケールアップ性能とスケールアウト性能)という点で、　Scala/Akka/Playが持つコンセプトとマッチしたと考えています。

具体的には、視聴権管理システムのIOのパフォーマンス問題を、　分散できる高速なデータストアとAkkaの並行処理技術を駆使すれば解決できると目論んでいました。

加えて、視聴権管理システムのポーリングのCPUコストに関しても、Playの持つWebSocket機能に変更することで、　CPUのコストを低減し、さらにリアルタイム性の高い機能を実現できるのではないかと考えました。

なお運用面に関しても、すでにniconicoのスマートフォンAPI用のAPIサーバーにてPlay/Scalaの導入実績があり、　問題なく運用できていることがわかっていました。

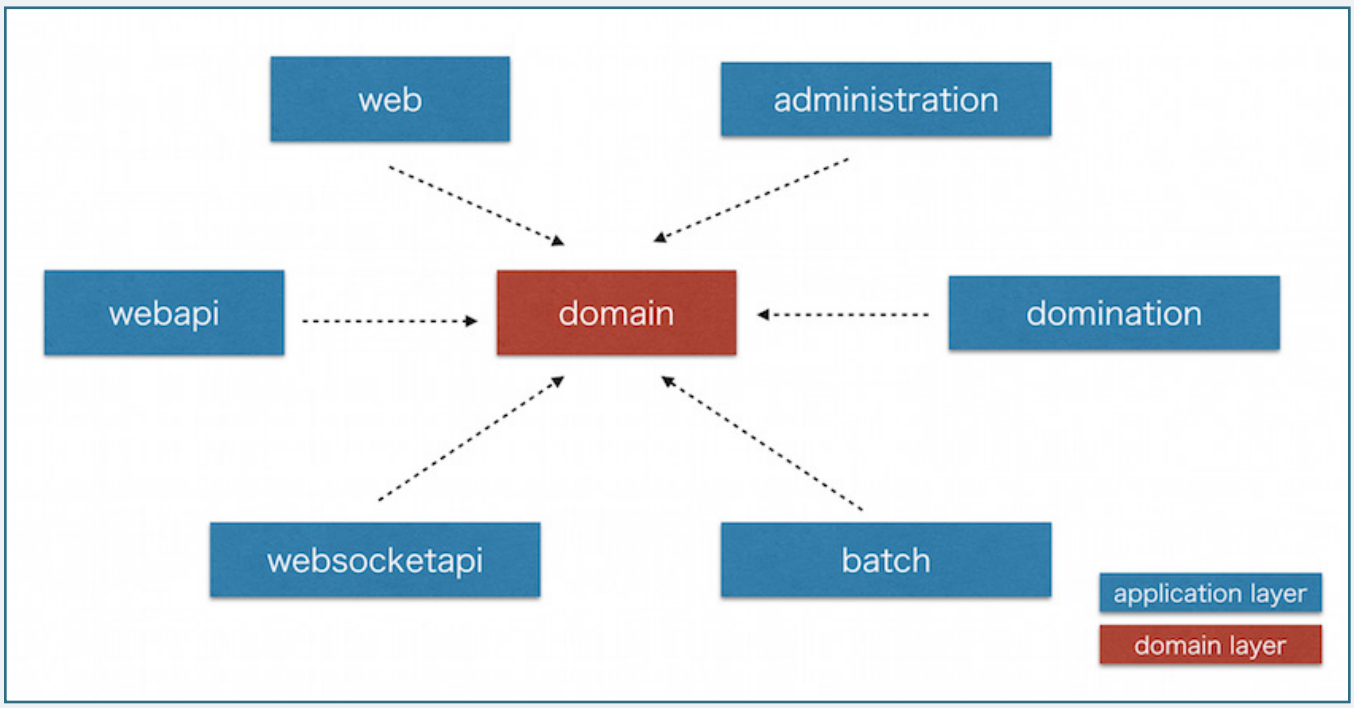
データストアは、

- MySQL/MHA
- Redis/Twemproxy/Sentinel

以上のようにしました。MySQLとRedisを冗長化のためのシステムを導入して構成して利用しています。　MySQLは、Master-Slave構成の上でMHAで冗長化し、内部的には水平分割と垂直分割を行っています。　Redisは、Twemproxyでクラスタリングして、Sentinelで冗長化してある他、-100Byte以下の小さなデータを　高速に取り扱うものとそうでないもので別なクラスタリングを用意しています。　これは、Redisがシングルスレッドで実装されており、　大きなデータを扱いブロックしてしまうと全体としてのRPSが低下してしまうため、それを防ぐ目的として行われています。

モジュール構成

モジュール構成は、過去実績が多かったレイヤー化アーキテクチャを採用しました。　以下の図のようにドメイン部分は共通化されたjarとし、役割の異なるPlayのアプリケーションを構成しています。



上記の図では、web, webapi, websocketapi, administration, batch, domi-nationという6種の Playのアプリケーションとdomainのjarという形で表現していますが、実際には共通のアプリケション部分を用意したり、もっと細かく役割毎に分割されています。 なお各アプリケーションは、Akka Clusterやデータストアを通じてやりとりをしています。

チーム構成は、この設計されたモジュール構成に合わせて配置しています。 これは組織パターン(James O. Coplien著)で紹介されているコンウェイの法則に対する対処法の、 先に理想的なモジュール構成を考えてそれに合わせてチーム構成を作るという方法を参考にしています。

結果として、

- Webフロント (webを担当)
- 管理ツール (administrationを担当)
- バックエンド (webapi, websocketapi, batch, dominationを担当)
- その時点の新機能開発チーム

のようにチームを分割し、ひとつのチームを大体4~6人となるように構成しながら開発していきました。

なお、このモジュール構成のそれぞれのアプリケーションは、多態性を持つ一つのプロジェクトとして実装されています。

実際には1つのソフトウェアプロジェクトであり、そのプロジェクトがビルド時に6つのアプリケーションをビルドするという方式を取りました。この方法により、開発時にはモノリシックな一つのアプリケーションを作るように開発することができます。

なお、ビジネスロジックに依存関係が少ないサブシステムに関しては、マイクロサービスとして開発したものもあります。 同じくPlay/Scalaを利用した生放送用の視聴者に配信者がプレゼントを行うというシステムは、 RESTベースのAPIで他のシステムと通信する別システムとし、完全に別チームで開発されました。 逆に、もともとマイクロサービスで実現されていたサブシステムを、 通信によるオーバーヘッドや管理コスト上の問題からニコ生新プレイヤーのプロジェクト内に取り込む修正をしたものもあります。

プロジェクト規模

コードベースの純粋な行数は2013年5月から2015年5月までの2年間の開発で16万行でした。
プロジェクトは、2013年5月開始時4名ではじめ、2014年12月までに最終的なプロジェクト人数は

20名超えとなりました。　チームは最初は1チームでしたが、最終的には4チームで構成しました。

新人を迎えた場合は、管理ツールチームに割り当てるように配置しました。

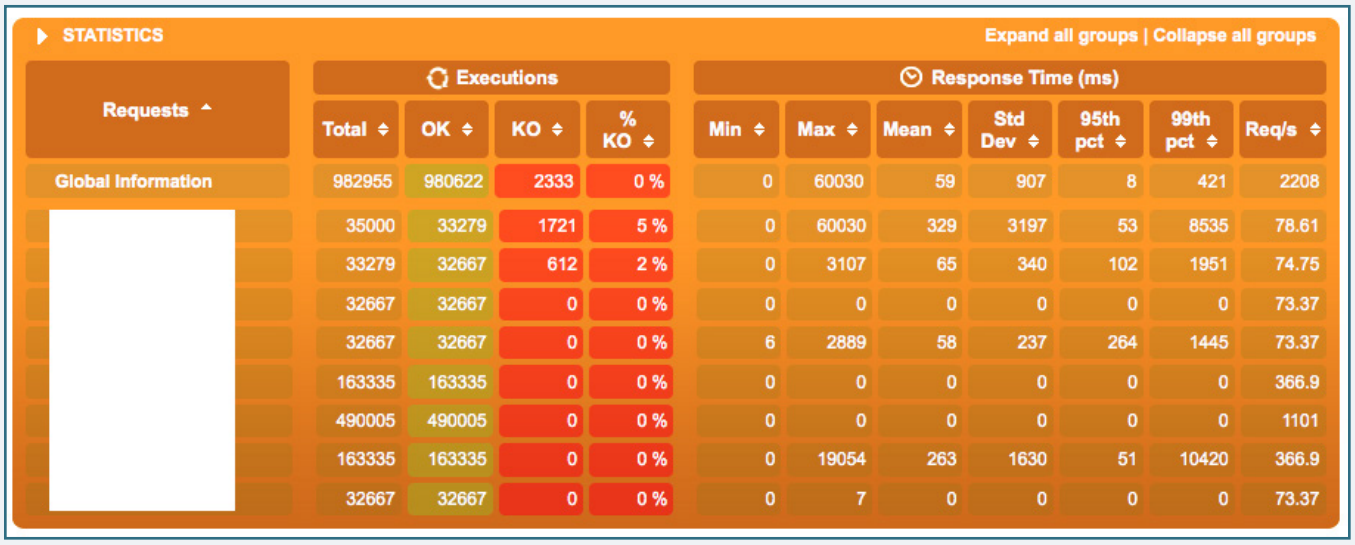
これは先ほど出てきた組織パターンの、”託児所”というパターンを参考にしたものです。　メリットとして、管理ツールは問題があった際に対応しやすいということや、　サービスのディレクターがユーザーなので新人のコミュニケーションのトレーニングにもなるといったことがあります。

結果

無事、2013年末には大規模な視聴を実現するためのプラットフォームが完成しました。　また当初の目標であった大規模な同時視聴ができるパフォーマンスも難なく実現できました。

通常で運用しているPHPの自社フレームワークとPlayのパフォーマンスの差は、　CPUベースでは10倍程度のパフォーマンスの差があります。　これは、自社フレームワークで利用していたPHPのSmarty2のレンダリングと、　Playで利用している　テンプレート　のパフォーマンスを比較したものです。

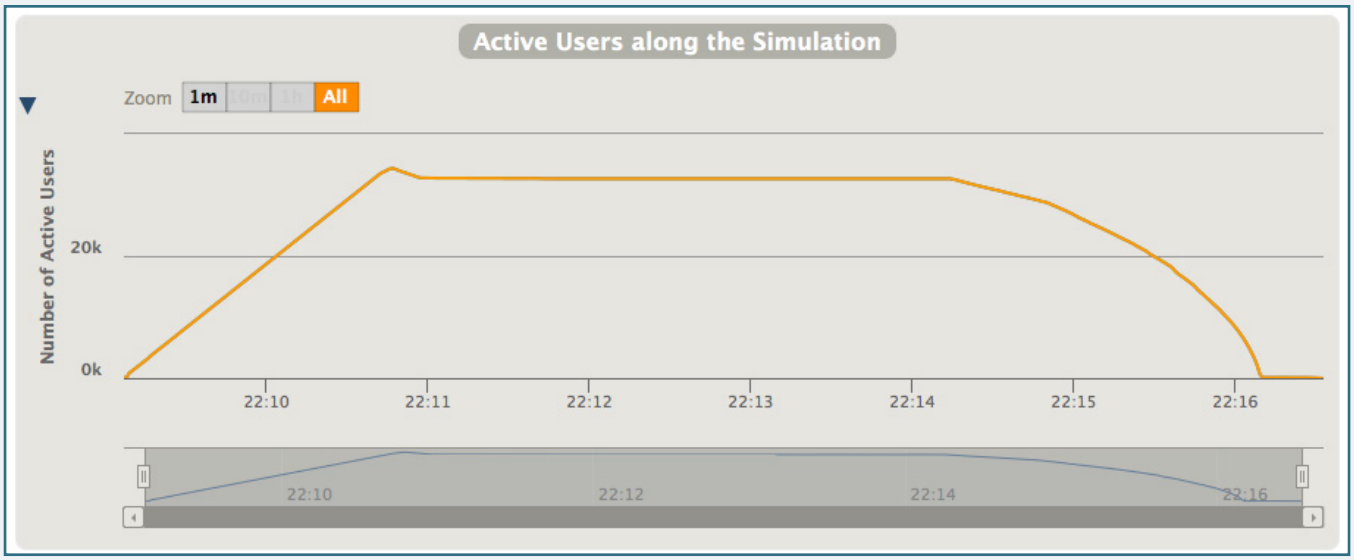
なお、Gatlingを利用して視聴権管理のパフォーマンスをした際の実験の結果は以下のとおりです。



STATISTICS　　Expand all groups | Collapse all groups

Requests	Executions				Response Time (ms)						
	Total	OK	KO	% KO	Min	Max	Mean	Std Dev	95th pct	99th pct	Req/s
Global Information	982955	980622	2333	0 %	0	60030	59	907	8	421	2208
	35000	33279	1721	5 %	0	60030	329	3197	53	8535	78.61
	33279	32667	612	2 %	0	3107	65	340	102	1951	74.75
	32667	32667	0	0 %	0	0	0	0	0	0	73.37
	32667	32667	0	0 %	6	2889	58	237	264	1445	73.37
	163335	163335	0	0 %	0	0	0	0	0	0	366.9
	490005	490005	0	0 %	0	0	0	0	0	0	1101
	163335	163335	0	0 %	0	19054	263	1630	51	10420	366.9
	32667	32667	0	0 %	0	7	0	0	0	0	73.37

上記のグラフから1インスタンスで32667の同時接続をさばいている事がわかります。



実際には1台の物理マシンに2つのアプリケーションのインスタンスを実行しているため、 実際には1物理マシンで約6万5千の同時接続の視聴権管理をすることができ、 従来のアプリケーションに比べ約4倍のパフォーマンスを出すことができました。 これによりWebSocketを利用することで、CPUのコストを低減するという当初の目論見を達成しました。

なおこの結果は、OSのプロセス当たりのファイルのハンドラの制限がボトルネックとなっているため、 チューニングによりまだ性能を伸ばせる可能性があると考えています。

この大規模視聴システムを利用して、2013年末のカウントダウン番組や 2014年に実施された日本の衆議院選挙の特別番組、 2015年4月に行われたniconicoが主催するリアルイベントである超会議(Chokaigi)のほぼ全ての番組などを、 安定して提供することができました。

2015年6月現在も、この大規模視聴システムに対して、新たな機能を追加しています。

考察

問題点
このプロジェクトを進める中で、実際に問題となった点は以下の2つです。

- 学習コスト
- コンパイル速度

学習コストに対する考察ですが、もともとPHPを利用したエンジニアに対しては、非常にScalaの教育コストは高いものでした。

- データ構造とアルゴリズム
- 並行処理プログラミングとアクターモデル
- 関数型言語特有の考え方 （実際には、Haskellの学習）

以上の内容を学習するためのプログラムを実施しましたが、　これを行うためには、教育に本気に取り組める組織と実行できる能力の高い人が必要とされます。　幸運にもドワンゴはそのようなメンバーがある程度は居たのですが、それでもプロジェクト開始後1年間は不足感がありました。

またコンパイル速度に対する問題ですが、これが引き金となって

- CIが回る速度が遅くなってしまい、問題検知が遅れる
- 迅速なリリースができない

といった運用上の問題が生じます。

結局これに対しては、早いマシンを用意する方法、変更が少ない部分をjarとして固めてしまうという方法、　変更が多く生じてしまうところをコンパイルが関与しないデータ構造にしてしまうといった対処しかできませんでした。

根本的な解決手段が見出しにくい状況となっています。

今後のプロジェクトとしての改善

今後の開発プロジェクトの課題としては、コードの記述レベルの差の問題がひとつあります。

これはScalaをBetter　Javaとして書く人もいれば、Haskellの代わりに書こうとする人もいることに由来します。　こればかりは教育体制の充実を行っていくしかないと考えています。

また、Akka　Clusterには運用上の課題ががまだ多くあると考えています。　特に24時間動き続けなくてはいけないようなシステムにおける、無停止のシステムの再起動や、　ActorSystem自体の監視、ビジネスロジックを含めたRemoteとの通信の保証のフレームワーク化などは、　ライブラリのサポートだけでは何とかならない部分もあるので、まだ多くのノウハウを貯めなくては　いけない状況にあると考えています。　なお現状ではAkka　Cluster自体を監視するための仕組みを取り入れて、これらの問題に対応しています。

メリット

無論、このPlay/Scalaを導入することで得られたメリットもあります。

- JVMが保証するパフォーマンス
- メモリリークが生じないこと
- 複雑なビジネスロジックの表現がPHP等に比べて優れている

まずJVMが保証するパフォーマンスによるスケールアップは、PHPやRubyなどを扱っていた場合は、過小評価できません。

最初からこれを目的としていたわけではありませんが、やはりサーバー台数が減らせるということや、スパイクのあるアクセスに対する安定性を確保できるということはとても高い価値があります。　加えてメモリリークが生じないのは、Scalaの関数型プログラミング寄りのコードスタイルに由来するものです。　基本的にPlayのアプリケーションを純粋な処理サーバーとし、状態はデータストアにまかせてしまうことで

過去のJavaアプリケーションでよく起こっていたメモリリークによる運用上の不安を払拭することができました。　これも、運用に対するストレスを大きく引き下げてくれるものとなります。

最後の複雑なビジネスロジックの表現が優れているという点ですが、　これは特に、Scalaのコレクションに対する豊かなAPIやflatMapの恩恵が大きいと考えています。　例外的な事象が挟み込まれるコレクションの処理をflatMapを利用して簡潔に書くことができ、　コードの可読性や保守性に対して大きな貢献をしていると考えています。　加えて、型が介在することによるコードの可読性の向上も非常に大きなものでした。

Scala/Akka/Playでのリプレースをすべきかのチェックポイント
以上の問題点とメリットを踏まえて、どのようなものに向くかのチェックポイントを上げると以下の3点です。

- LLを使っていてスケールアップ性能が求められている
- LLを使っていてスケールアウト性能が求められている
- 日々更新されている複雑なビジネスロジックを扱う

この状況であれば、Scalaでのリプレースをする価値はあると思います。

ただし、コストに対して得られるメリットがないのであればLLで書かれているシステムを
無理をして置き換える必要はないと考えています。

まとめ

- WebSocket、非同期の IO といった技術が組み込まれた Scala/Akka/Play を使うことで当初問題となっていたパフォーマンスの問題を解決することができた。ただし、性能特性はフレームワークだけではなく、データストアの冗長化といったトータルな視点での最適化を必要とした。
- Scala という表現力の高い言語を用いることでドメイン知識をユビキタス言語、型という形で組織的に共有することができ、コードのリーダビリティ、メンテナンス性も向上した。ただし、Scala の採用には高い学習コストも伴い、継続的に教育体制を整えていく必要も出てきた。