Wagbyのオートスケールに関する技術情報

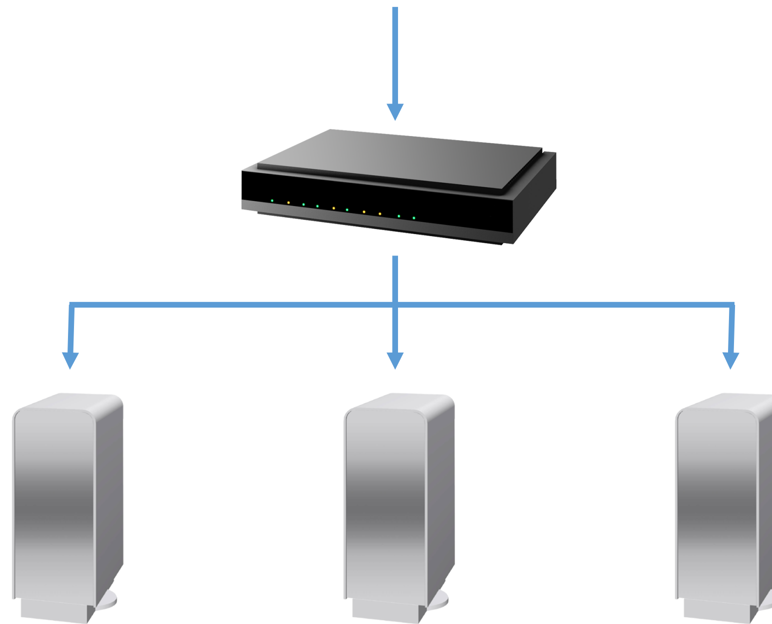
2019.7.18 Wagby R8.3.0 対応

OUTLINE

- 一般的なクラスタリング/オートスケール
- Wagbyのクラスタリング
- オートスケール対応の方針
- CacheManagerの対応
- セッション情報の共有
- ジョブスケジューラーの対応

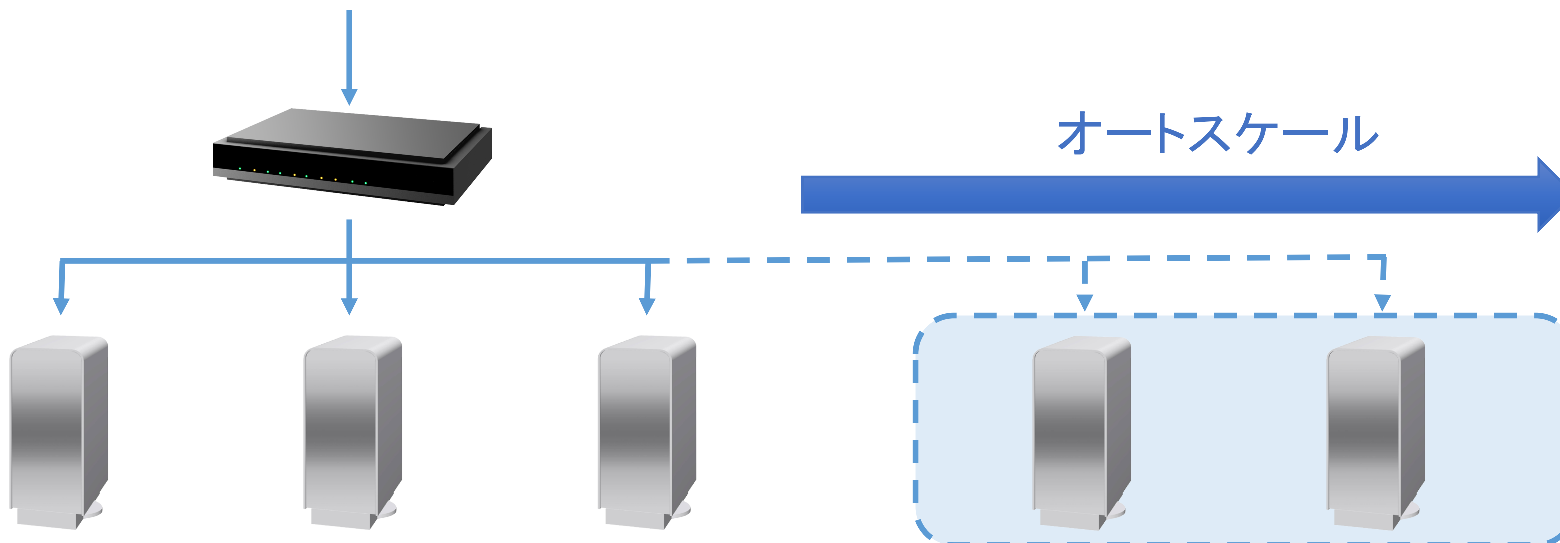
クラスタリング

- 一般的なクラスタリング
  - 複数台のアプリケーションを同時に可動させる
    - 1台で障害が発生してもサービスは継続できる(可用性の向上)
  - ロードバランサでアクセスを複数のアプリへ振り分ける(利用者は複数台を意識しなくて良い)

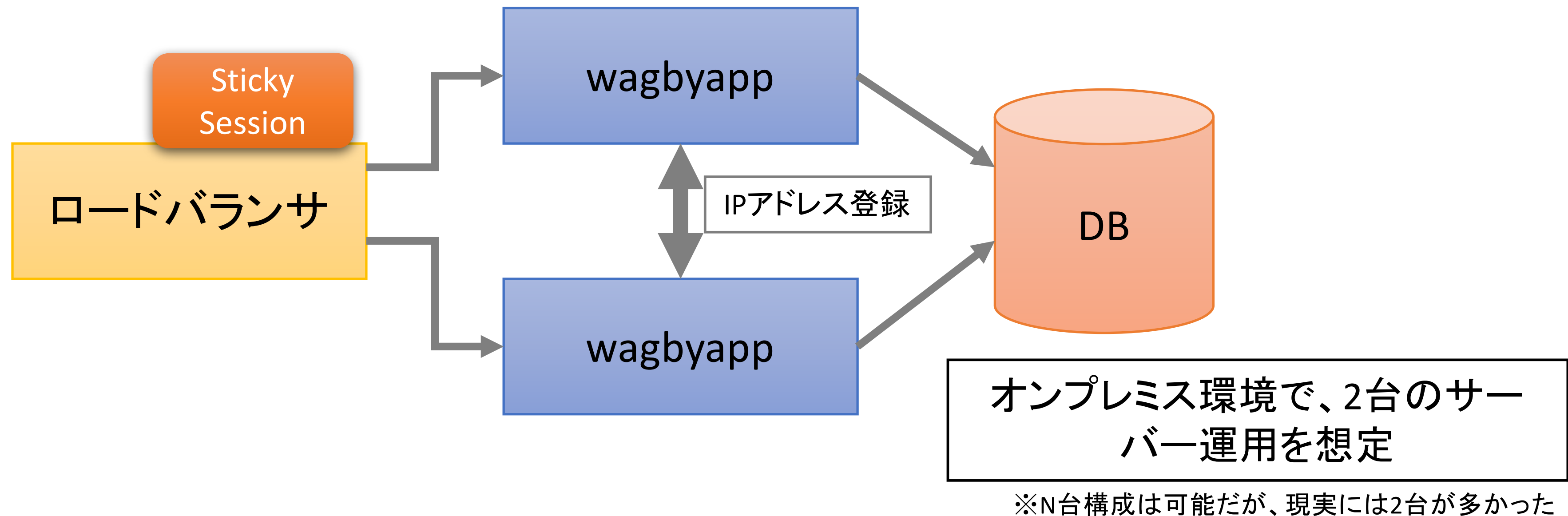


オートスケール

- 一般的なオートスケール
  - クラスタリングの考えに以下を加えたもの
    - スケールアウト: 稼働サーバーの台数を増やす
    - スケールイン　: 稼働サーバーの台数を減らす
    - オートスケール　　　　　: サーバーの負荷に応じてスケールイン・アウトを自動で行う



WagbyのクラスタリングR5.7.0+)



- RDBは一つ
- セッションレプリケーションなし
  - ロードバランサのSticky Session機能を有効にする
- 可動させるサーバーの数とIPアドレスを事前に決める
  - 定義ファイルに記述

単一サーバー動作時との違い

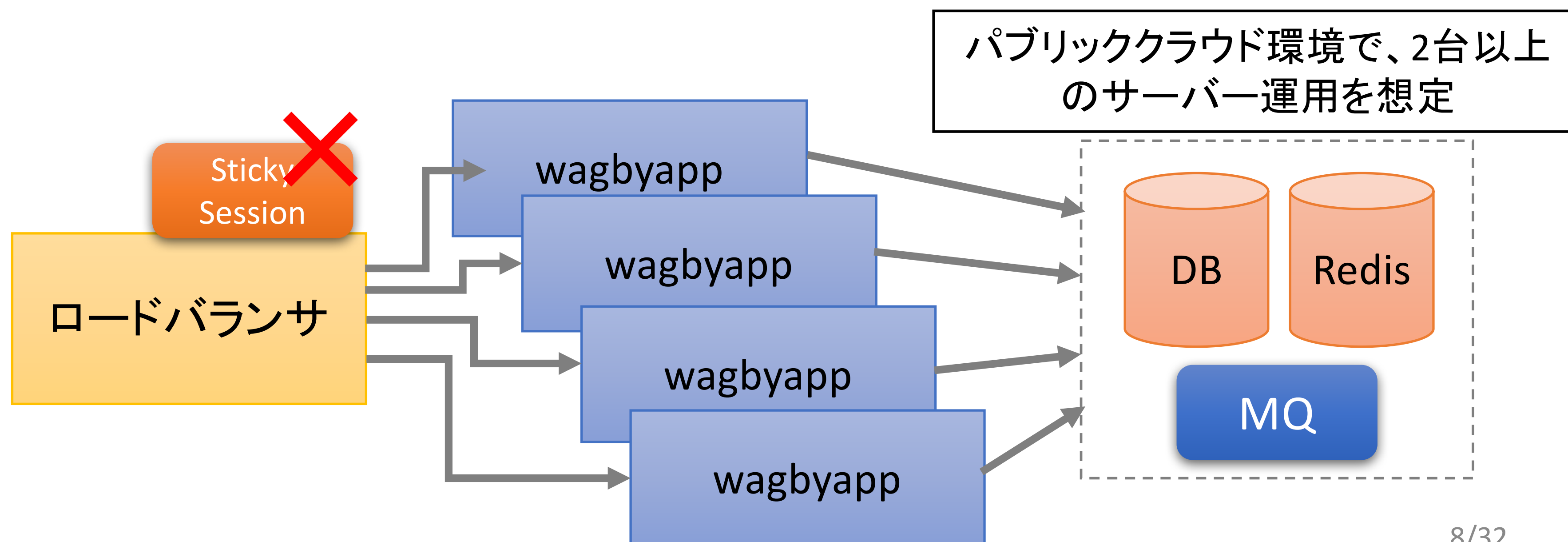
- キャッシュ
  - 各サーバー間でキャッシュ情報を共有する
    - クラスタリング時はサーバー間でJMX(Java Management eXtension)プロトコルを使った通信を行っていた
      - この通信には全稼働サーバーのIPアドレスが必要
- ロック
  - 悲観ロック情報を各サーバー間で共有する
    - 単一サーバー動作時はメモリ上で管理
    - クラスタリング時はRDBで管理
  - 楽観ロックは単一サーバーと同じ様に動作する
- セッション情報の共有
  - 共有は行わないため、ロードバランサでSticky Session機能を有効にする
    - ログオン後は常に同じサーバーに振り分けられる必要がある
    - サーバーがダウンした時、このサーバーを使っていた利用者のセッション情報は失われるため、再ログオンが必要

オートスケール対応の方針

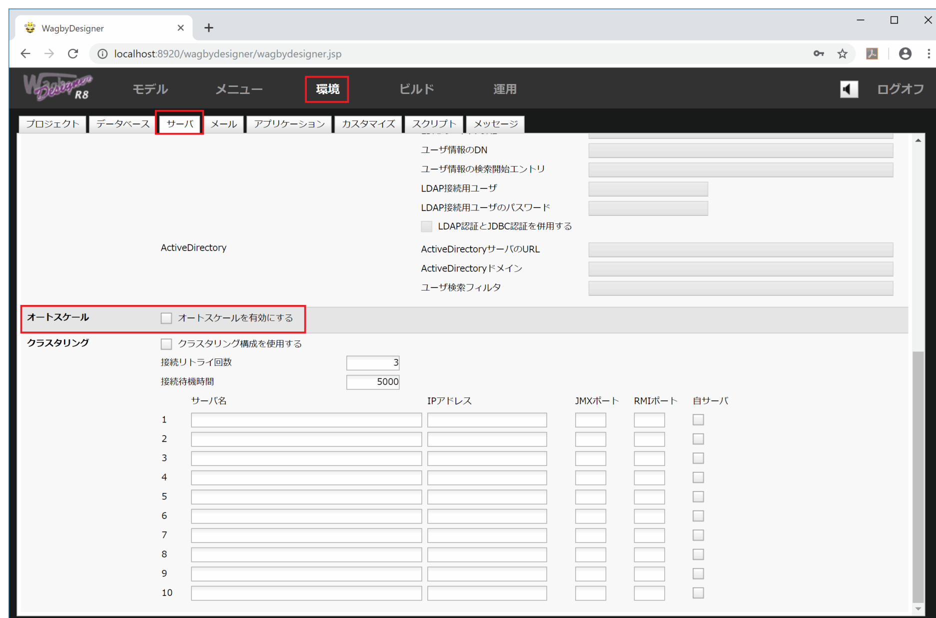
- セッション情報の共有を行う
  - サーバーダウン時の再ログオンをなくす
  - ロードバランサのStickySession機能不要
- IPアドレスの事前設定を不要とする
  - IPアドレスに依存しない複数サーバー稼働を実現する
- 課題解決にRedis、MQ(Message Queueing)を活用
  - 複数サーバー間の情報共有にはこの2つの技術が一般的

オートスケール対応の方針[2]

- セッション情報の共有にRedisを利用
- キャッシュ情報の共有にMQを利用
  - クラスタリング時はJMXを使っているが事前にIPアドレスの登録を必要するため、今回は利用しない
- その他、ジョブ機能、パスワードリマインダ機能などもオートスケール用の対応が必要

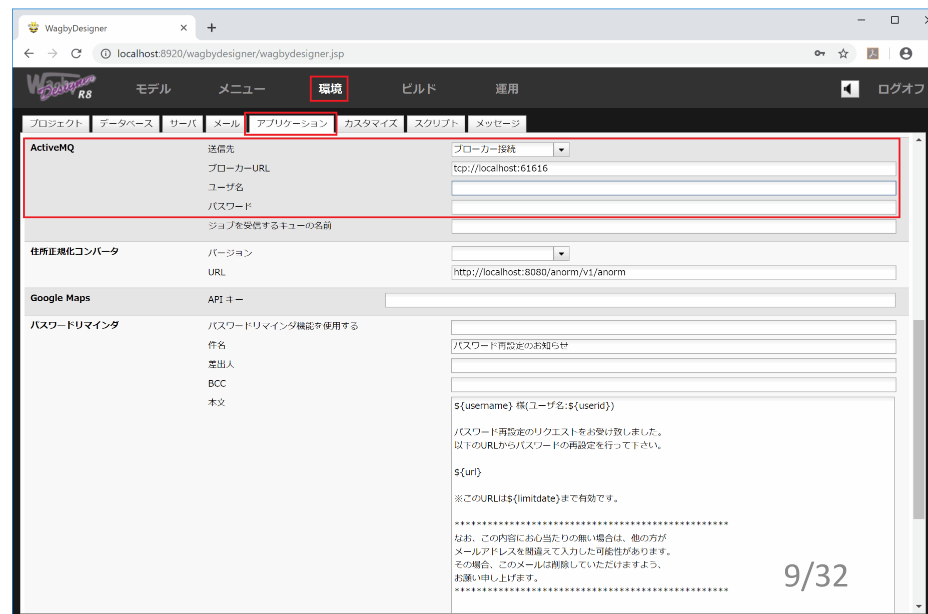


オートスケールの設定



(1) 環境>サーバ>オートスケールの欄に用意された「オートスケールを有効にする」をチェックする

(2) 環境>サーバ>アプリケーションの欄に用意された「ActiveMQ」を設定する。送信先は「ブローカー接続」とする。ブローカーURLとユーザ名、パスワードを指定する(ジョブを受信するキューの名前は空白とする)



想定する利用方法

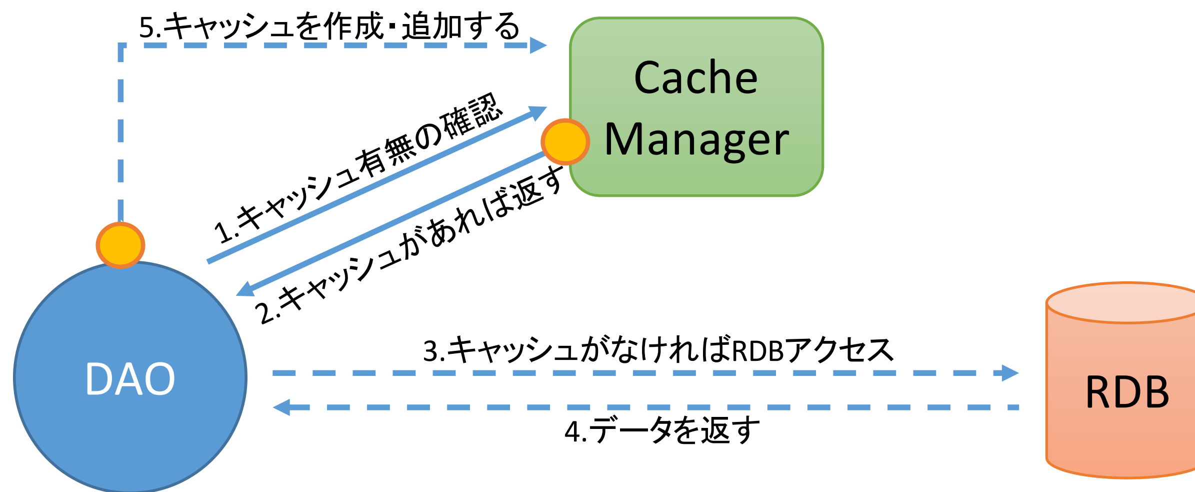
- Wagbyで開発した業務アプリケーションをパブリッククラウドで運用する(SoR分野での利用)

- WagbyでSoEのバックエンドを開発し、フロント層とREST API でつなげる(SoRとSoEの連携)

オートスケールに対応したWagby内部の仕組み

CacheManager

- データ取得
  - キャッシュなし: RDBから取得、キャッシュを作成
  - キャッシュあり: キャッシュを利用(RDBアクセスなし)
- データ登録・更新・削除
  - キャッシュを削除するのみ



CacheManager

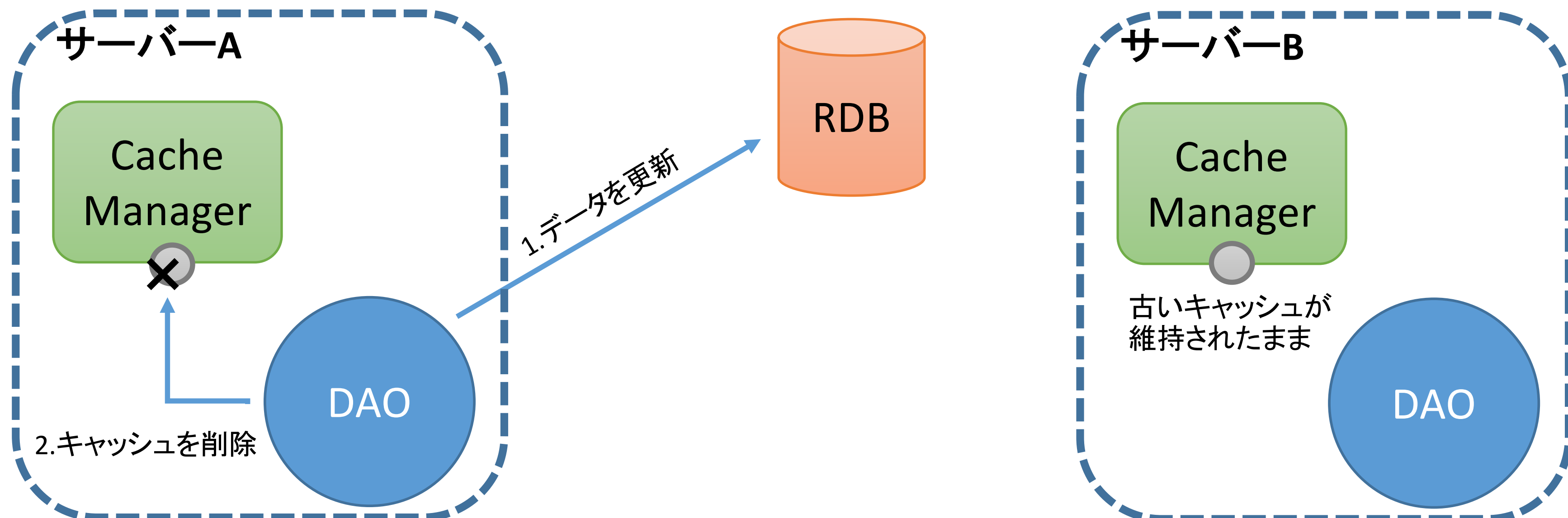
- Wagbyのキャッシュ対象一覧

対象	説明
ストアモデル (Entity)	データベースから取得したストアモデルのキャッシュ 詳細画面表示時や他モデル参照の解決に利用される
選択肢	画面に表示されるリストボックスやラジオボタン、チェックボックスの選択肢のキャッシュ 選択肢の絞り込み設定などが利用されている場合はキャッシュは作成されない

※一覧表示画面はキャッシュ対象外です。ただし一覧表示項目のモデル参照解決にはストアモデルのキャッシュが利用されます。そのため(N+1) 問題の発生を抑えられるようになっています

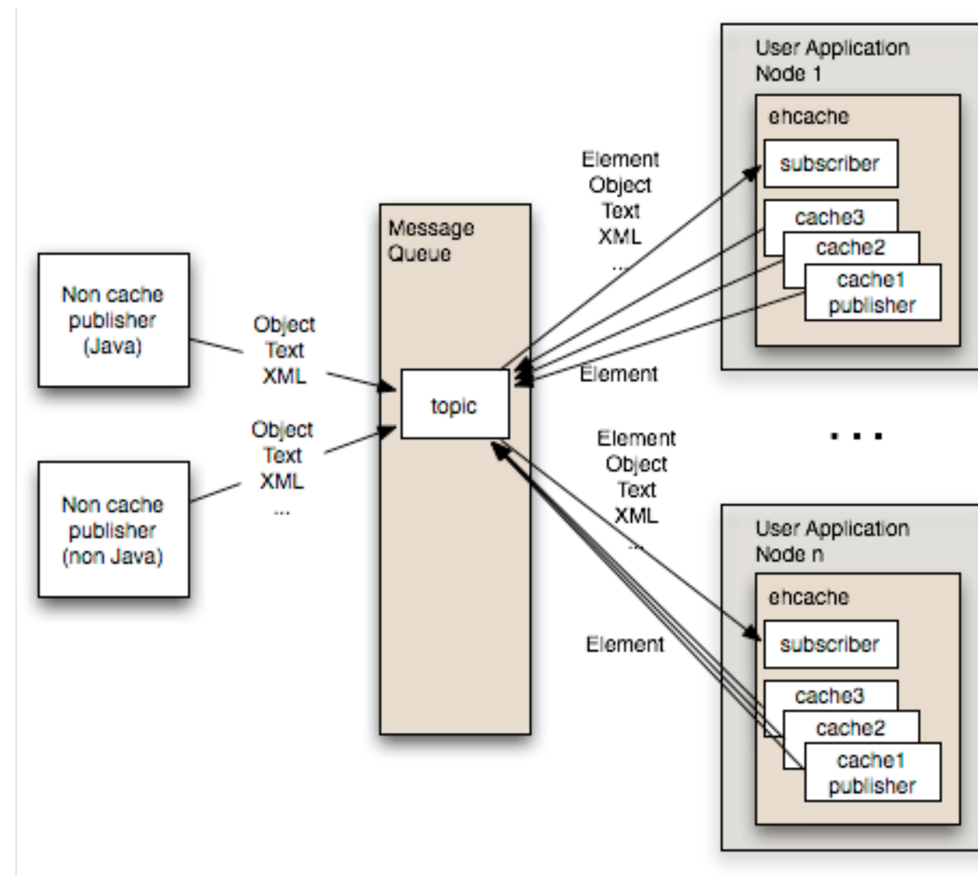
CacheManagerの問題点

- オートスケール時の問題点
  - 複数サーバーで個別にキャッシュを管理
  - キャッシュが古くなっていることを検知できない
    1. サーバーAの更新画面でデータを更新(RDBを更新)
    2. サーバーAのキャッシュはクリアされる
    3. サーバーBは更新に気づかず変更前のキャシュを保持したまま



CacheManagerの対応 [1]

- Ehcacheが提供するレプリケーションの考え方



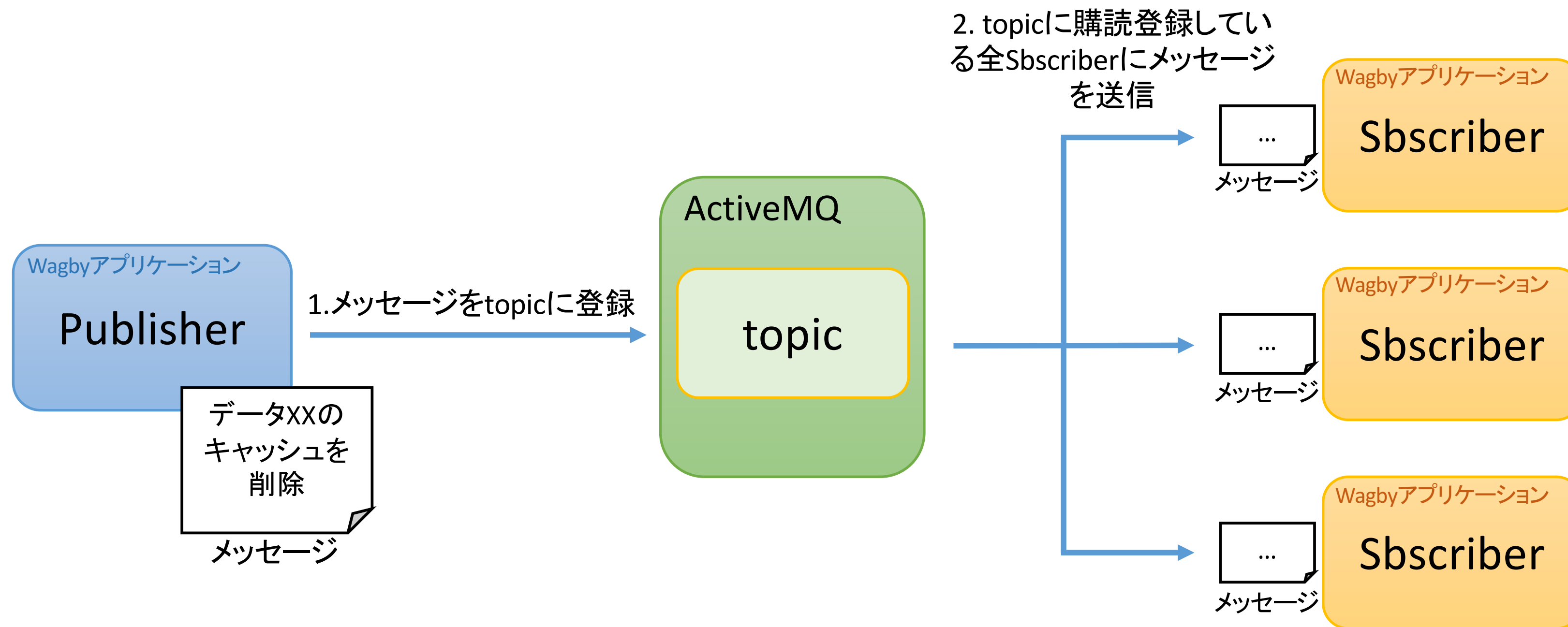
https://www.ehcache.org/documentation/2.7/replication/jms-replicated-caching.html

CacheManagerの対応 [2]

- Wagbyは複数サーバー間のキャッシュの複製は行わない
  - キャッシュデータのコピーは通信コストが高い
- 代わって、キャッシュ削除メッセージのやりとりのみを行う
  - キャッシュの作成は各サーバーに任せるがクリア処理だけは全サーバーで統一
  - Ehcacheのオプション機能を利用することでキャッシュの複製はせずにクリア処理だけを共有することが可能

CacheManagerの対応 [3]

- キャッシュクリア処理の共有
  - ActiveMQ(メッセージキュー管理サーバー)
  - Publish/Subscribeメッセージング機能(1対nメッセージングモデル)



CacheManagerの対応 [4]

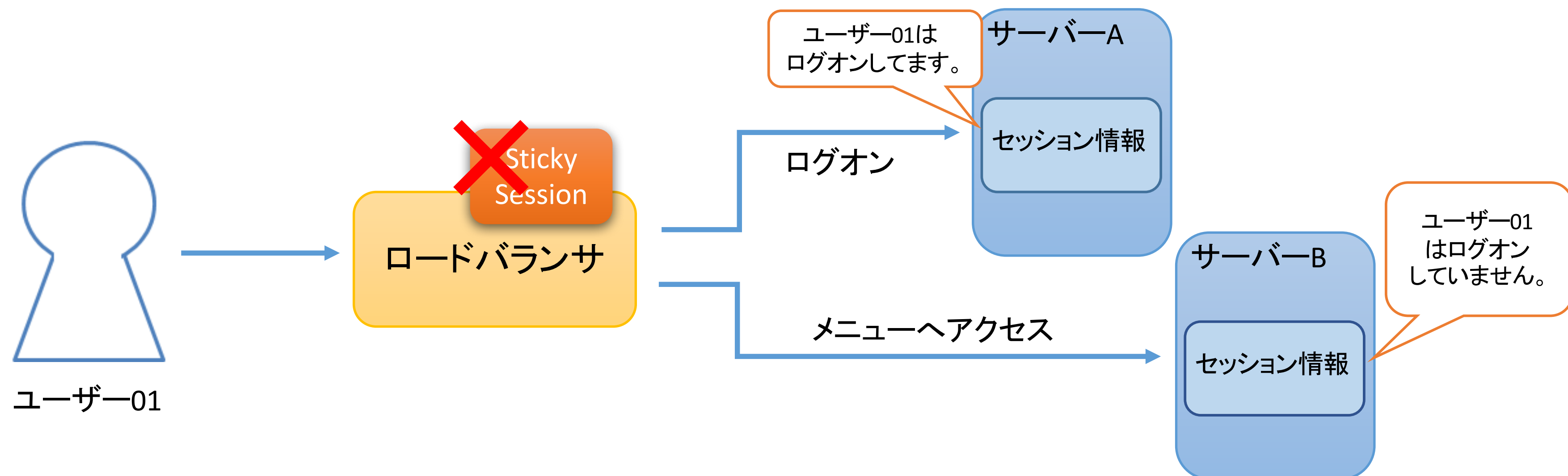
- Ehcache向け設定ファイル(WEB-INF/classes/ehcache.xml)

```
<defaultCache
    eternal="false"
    timeToIdleSeconds="600"
    timeToLiveSeconds="600"
    memoryStoreEvictionPolicy="LRU">
  <persistence strategy="none"/><!-- In-memory Only Cache -->
  <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
    properties="replicateAsynchronously=false,
                replicatePuts=false,          … キャッシュ追加処理の共有 → しない
                replicateUpdates=true,        … キャッシュ更新処理の共有 → する
                replicateUpdatesViaCopy=false, … ただし、キャッシュ共有ではなく、削除のみ
                replicateRemovals=true"       … キャッシュ削除処理の共有 → する
                propertySeparator=","/>
</defaultCache>
```

- 上記に加えメッセージサーバーの接続情報も定義する

セッション情報の共有

- Sticky Session機能を利用しない環境ではセッション情報の共有が必要
    1. サーバーAにログオン
    2. サーバーAのセッションにユーザー01のログオン情報が書き込まれる
    3. メニュー画面の表示のためサーバーBへアクセス
    4. ログオン情報がないためログオン画面が表示されてしまう



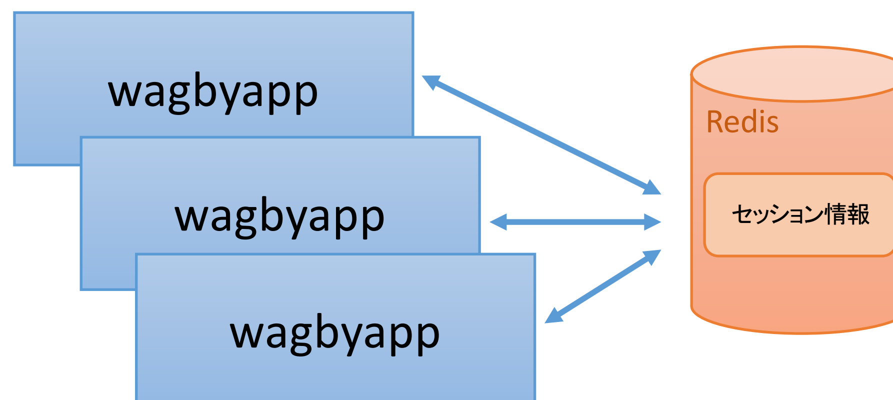
SpringSessionのセッション共有[1]

- Redis(KVS)の利用(別途Redisサーバーが必要)
  - これまで:マルチセッション実現のためにRedisを利用
  - 今後:セッション情報の共有のためにもRedisを利用

WagbyDesigner > 環境 > サーバ

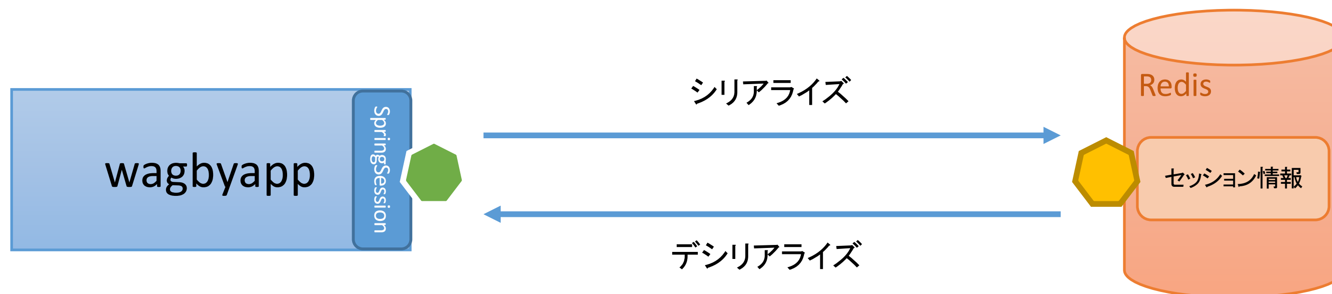
HTTPセッション	格納方式	Spring Session Redis	
	Redisサーバの設定	Redisホスト名	localhost
		Redisパスワード	
		Redisポート番号	6379

- 複数台のWagbyアプリケーションで同じRedisに接続
  - Redisを通してセッション情報を共有



SpringSessionのセッション共有[2]

- SpringSession+Redisでのセッション共有
  - セッションに格納するオブジェクトはjava.io.Serializableインタフェースを実装する
  - Serializableを実装できないオブジェクトはtransient修飾子を付与しシリアライズ対象外とする
    - Wagbyでの主なシリアライズ対象外クラスは以下
      ※間接的にセッションに格納されていたためtransientとした
      - ApplicationContext(Springが提供するクラス)
      - LoginContext(JAAS認証で利用)
    - ApplicationContextはリクエスト毎に再セットすることで解決



SpringSessionのセッション共有[3]

- LoginContextのシリアライズ対応
  - javax.security.auth.login パッケージに属するクラス
  - ログオン/ログオフ機能を持ち、権限情報を保持
  - 派生クラスでのSerializable実装は困難
  - 権限はjavax.security.auth.Subjectで表現(シリアライズ可能)
  - LoginContextはSubjectを引数とすることでもインスタンス作成が可能
  - Subjectをセッションに格納し、必要に応じてLoginContextのインスタンスを復元することとした

```
new LoginContext(logonModuleName, subject)
```

SpringSessionのセッション共有[4]

- カスタマイズの際の注意点
  - CGLIBでProxy化されたクラスをセッションに格納するとデシリアライズ時にエラー
    - 厳密にはシリアライズ時と異なるサーバーでデシリアライズされた場合にエラーとなる

```
java.lang.ClassNotFoundException:
jp.jasminesoft.wagby.XXX$$EnhancerBySpringCGLIB$$17546ee8
```

  - 対策
    - セッションに格納しない。都度ApplicationContextから取得
      - 特にDAOやEntityServiceはセッションに格納してはいけない
    - CGLIBではなくJDK DynamicProxyを使う
      - 当該クラスにインターフェースを作成(JDK DynamicProxyはインタフェースが必須)
      - bean 定義で proxy-target-class=“false”属性を指定
      - アノテーションの場合は @XXX(proxyMode = ScopedProxyMode.TARGET_CLASS)

Redisなしでの動作

- 以下の条件ではオートスケール環境でもRedisは不要となる
  - WagbyをREST APIサーバーとしてのみ利用する
  - すべてのアクセスは“X-Wagby-Authorization”を使ってステートレス接続とする
    - 1度のアクセスで「ログオン >目的の処理 > ログオフ」を行うので、Wagby側ではセッションで情報を維持する必要がない
    - セッション情報の維持が不要のため複数サーバーでのセッション共有の必要がない (= Redisは不要)

ロックについて

- 悲観ロック
  - クラスタリング時の機能をそのまま利用
  - RDBのjfclockobjectテーブルにロック情報を格納し、各サーバー間で共有する
- 楽観ロック
  - そもそも稼働サーバーの台数に依存しない機能
    - 一台でも複数台でも同じ仕組みで動作する
  - データテーブルのバージョンカラムを使ってロックを管理

ジョブスケジューラーの対応[1]

- Quartz
  - Wagbyのジョブスケジュールを実現しているライブラリ
  - 指定した時刻にジョブを実行
- 単一サーバーでの動作
  - スケジュール情報をメモリー上に保持(RAM Jobstore)
- 複数サーバーへの対応
  - スケジュール情報をRDBに保持 (JDBC Jobstore)
  - 各サーバーで共通のスケジュール情報を参照
  - 指定した時刻に1つのジョブをいずれかのサーバーで実行する

ジョブスケジューラーの対応[2]

- Quartz JDBC Jobstore
  - データベースに専用のテーブルを用意(11テーブル)
  - テーブル名の先頭にQRTZ_がついている
    - QRTZ_LOCKS, QRTZ_TRIGGERS, QRTZ_JOB_DETAILS, QRTZ_CALENDARS, …
  - Quartz内部でjob, triggers, calendarsと呼ばれる情報を格納する
  - テーブルに対応するWagbyのモデルは存在しない
    - Wagbyが直接テーブルを操作することはせずQuartzに移譲
    - モデルが存在しないため、CREATE TABLE/DROP TABLE用のDDLは特別に管理している
  - テーブルのCREATEやDROPはInitLoaderで実行
    - jfcjobscheduleモデルのテーブルの作成/削除時に同時実行

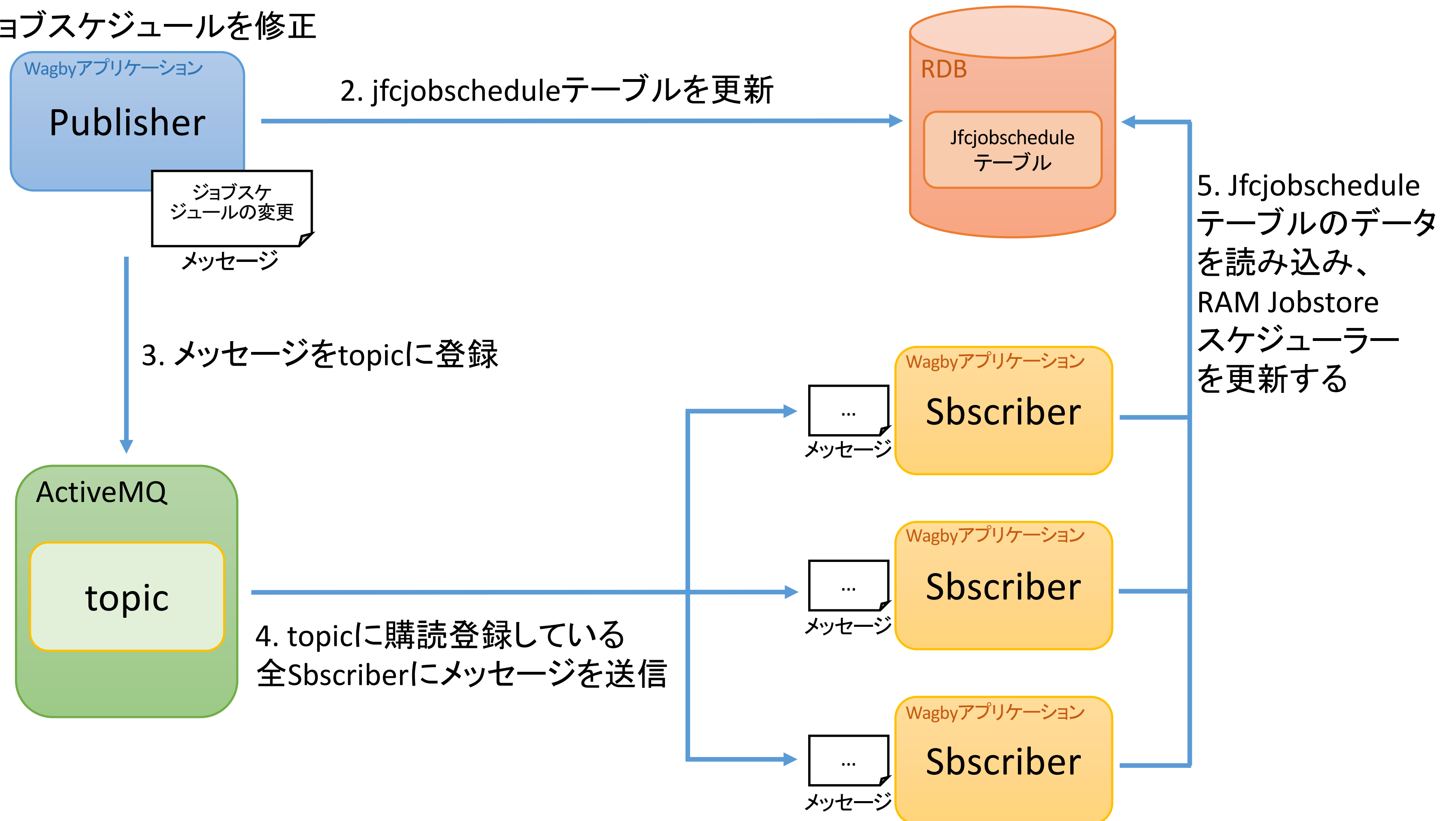
ジョブスケジューラーの対応[3]

- ジョブの実行制御
  - 実行対象:1インスタンス
    - 各サーバ中1台のみが実行するジョブ
    - Exportジョブ、メール送信ジョブ等
  - 実行対象:全インスタンス
    - 各サーバーで個々に実行するジョブ
    - メンテナンスモード切替用ジョブ
      - 例：AM0:00にすべてのサーバーをメンテナンスモードにする
  - JDBC Jobstoreは全インスタンスを実行対象とする仕組みがない
    - この問題解決のためJDBC JobstoreとRAM Jobstore併用する仕組みとした
    - JDBC Jobstore : 1インスタンス実行用
    - RAM Jobstore :全インスタンス実行用

ジョブスケジューラーの対応[4]

- jfcjobscheduleモデルの拡張
  - 実行対象項目の追加
    - one(1インスタンス)、all(全インスタンス)を選択
    - 実行種別が未指定の場合は1インスタンス実行とする
  - 更新日時項目の追加
    - ジョブスケジュールを変更した日時を格納
- ジョブスケジュール変更情報の共有
  - ActiveMQのPublish/Subscribeメッセージング機能を利用
  - あるサーバーでジョブスケジュールの変更を行うと全サーバーへ通知され、各サーバーでジョブスケジュールの再読込を行う(RAM Jobstoreのみ)

ジョブスケジューラーの対応[5]



ワークフローの対応

- 同時承認への対応
  - 同一データに対する複数ユーザーの同時承認
    - 単一サーバーでの動作時
      - 承認処理を行うメソッドにsyncronized修飾子を付与
      - 後から行われた承認処理は待ち状態となり、処理再開後の状態チェックで2重承認を回避する
    - オートスケール時の動作
      - 承認対象のデータに悲観/楽観ロックを行い2重承認が行われた場合はロックエラーとする
        - 現在も悲観ロック処理は行われているが、承認処理の最後のごく短い時間のみ
        - ロックの範囲を承認処理全体とすることで対応

まとめ

- オートスケール対応で内部処理を変更している。
  - ロック処理は変更なし。
  - キャッシュは ehcache のメッセージを使って整合性をとる。MQが必要。
  - SpringSessionのセッション共有はRedisを使う。
  - ジョブスケジューラは quartz の制御方式を使う。
  - ワークフローは承認処理でロックを適用する。

- 必要なミドルウェア
  - MQは必須。今後、MQを活用する局面は増える見込み。
  - Redisは任意。(p.24の条件が整えば、なくても動作する)