

VoIPサービスにおけるPostgreSQLの活用

ソフトバンクBB 株式会社
コミュニケーション・ネットワーク本部システム部
松川 崇訓

2010年06月19日



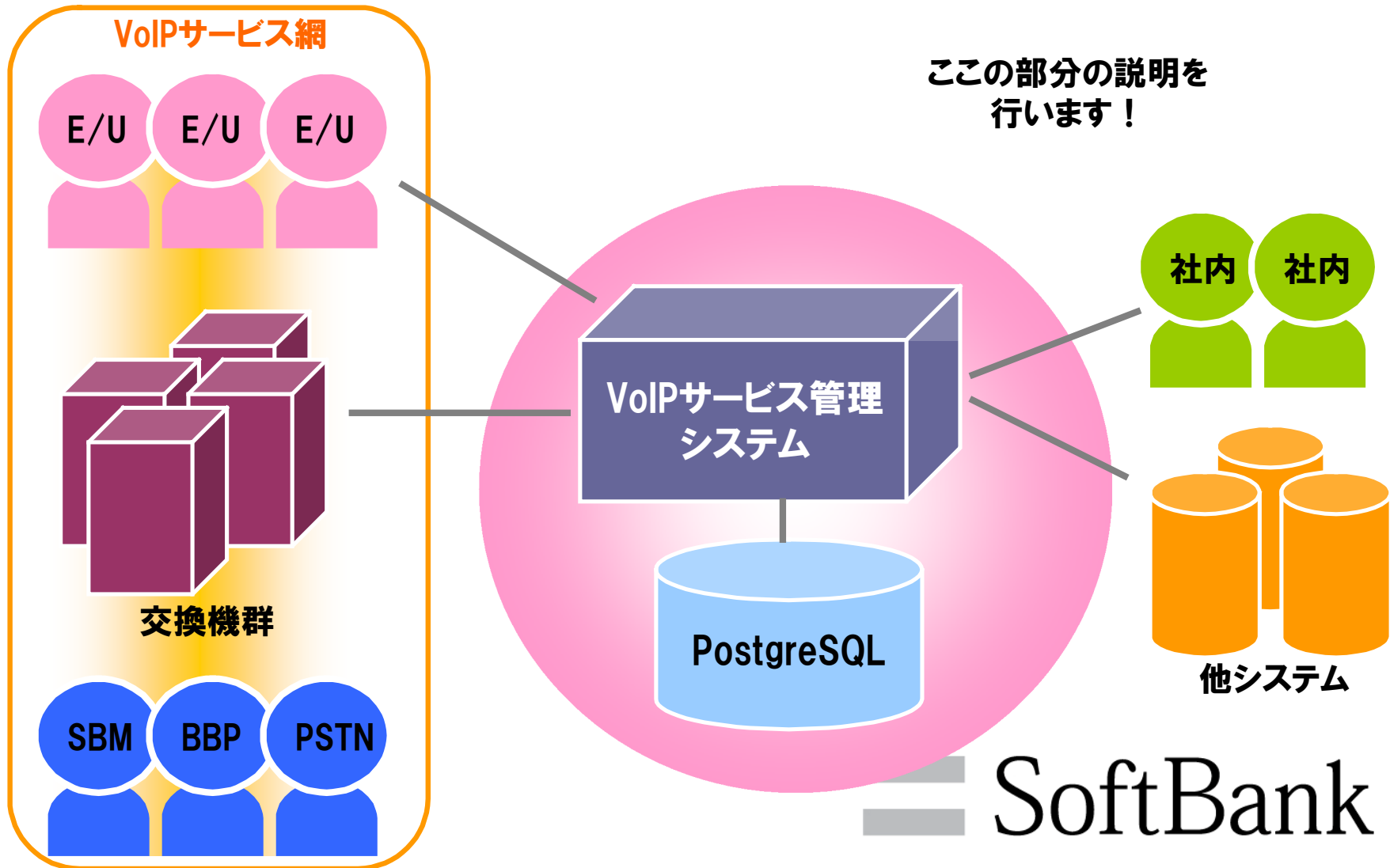
- ソフトバンクグループが展開するVoIPを使用したコミュニケーションサービスにおけるPostgreSQLの活用事例を紹介します。
- アプリケーションエンジニアの視点から開発フェーズにおけるエピソード、オペレータの視点から運用フェーズに突入してからのエピソードを紹介します。

- **名前: 松川 崇訓**
 - 2005年にソフトバンクBBに入社しました。
 - 現在、開発も運用も行っています。
- **スキルレベル**
 - PostgreSQL 中級？
 - インストールや設定を、調査しながら実施できる。
 - SQLを用いて自由にデータの加工・抽出ができる。
 - Java 中級
 - WEBアプリケーションの製造が行える。
 - 必要なライブラリを選択することができる。
 - HTML & Javascript 中級
 - 基本的なAjaxを理解できる。
 - 必要なライブラリを選択することができる。

VoIPサービスのシステム概要

VoIPサービスのシステム概要

ソフトバンクグループでは、いくつかのVoIPサービスを提供しています。その管理システムの一部にPostgreSQLを導入しています。



VoIPサービス管理システムの役割

VoIPサービス管理システムには様々な役割があります。

- 交換機に対して顧客情報の登録・削除などを行うプロビジョニング機能
※ プロビジョニングについては後述します。
- 交換機に生成された通話履歴から通話料金の課金・請求を行う機能
- お客様や社内から、お客様自身の個人情報にアクセスする機能。WEBで提供。
- 出荷管理機能。
- 与信管理機能。
- etc..

VoIPサービス管理システムは、様々なシステムのハブとして機能している重要なシステムです。

開発編

昨今のWEBアプリケーションではDBを当たり前のように使用しますが、アプリケーションエンジニアはSQLが苦手な人も多く、これを補うためフレームワークに深く依存しています。このような現状を踏まえシステム開発段階の説明をさせていただきます。

1. 基幹技術を選択する

DBより先に期間技術となるフレームワークを選択します。

2. S2Dao & JdbcManager

SQLは苦手です。

3. DBを選択する

ちょっと遅いですが、DBの選択を行います。

4. リアルタイムプロビジョニングの実装

交換機へ命令を投げる方法の説明です。

基幹技術を選ぶ ～ アプリケーションエンジニアは流行が好き

開発当初、JAVAのアプリケーションエンジニアの間では“DI”(dependency injection)の概念が流行していました。この波に乗るため、私たちもDIを採用することは早くから決定していました。

DIコンテナの例

- Spring (世界的に、ほぼデファクトスタンダード。)
- Seasar2 (国産。日本語ドキュメントが圧倒的に多かった。)
- google guice (Googleが作った軽量コンテナ。)

私たちは国産で日本語ドキュメントが多かったSeasar2の採用を決定しました。

※ 最近の流行

iPhone & Android → sqlite

Google App Engine → Big Table (キーバリューストア)

流行からDBを選ぶため、一般的なRDBが必ず採用されるわけではありません。

基幹技術を選ぶ ~ Seasar2の機能概要

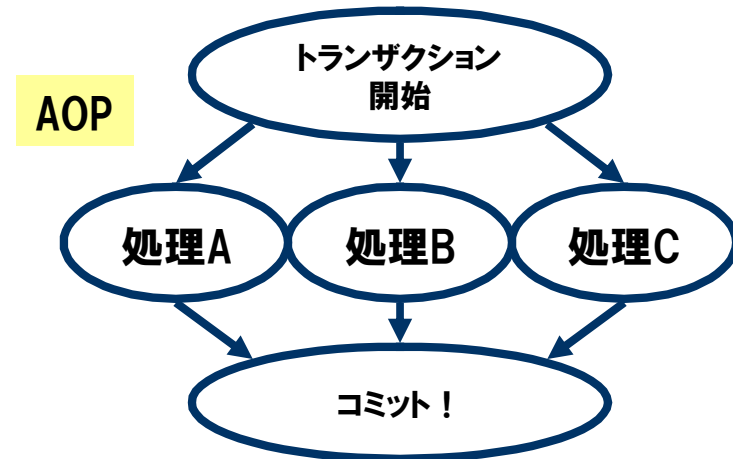
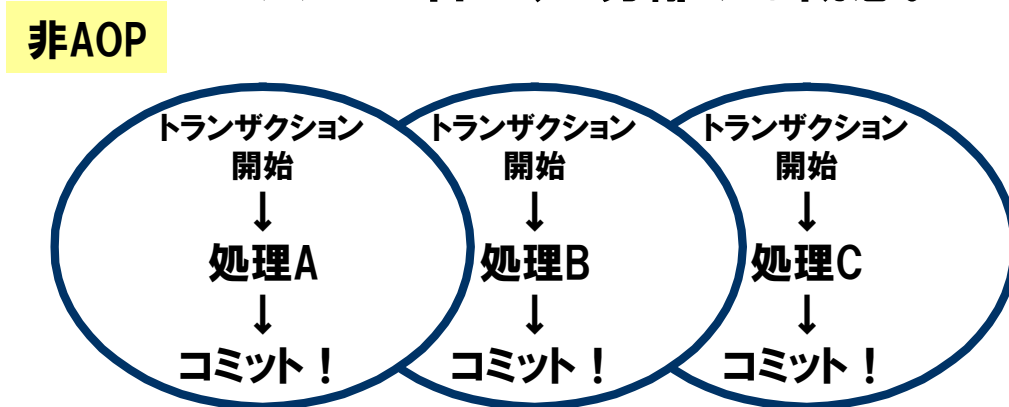
- **DI (dependency injection)**

オブジェクトの依存関係を、外部のXMLファイルなどに記す概念。
 環境によるオブジェクトの入れ替えが容易でテストが実施しやすくなる。



- **AOP (Aspect Oriented Programming)**

本来実装すべきビジネスロジックとは異なる機能(例:ロギング、トランザクション管理)を分離する概念。



従来、DBに問合せを行う場合は、SQLをプログラム内部で文字列データとして生成し、これの実行結果をレコードセットとして受け取り、対応する変数へ格納する方法がとられていました。

```
String sql = "SELECT * FROM CUSTOMER WHERE customer_id >100";
```

↑ SQLが長くなると文字列処理が複雑になる。

```
ResultSet rs = statement.executeQuery (sql);  
while (rs.next ()) {  
    int customerId = rs.getInt ("customer_id");
```

↑ customer_id が整数型であることが分からない。

```
String telephoneNumber = rs.getString ("telephone_number");
```

↑ telephone_number が文字列型であることが分からない。

```
}
```

S2Dao & JdbcManager ~ ORマッピングを使用した実装

現在、多くのフレームワークではSQLを直接記述することなく、DBにアクセスするための専用のオブジェクト”DAO”(Data Access Object)をSQLの代わりに使用します。

【例1:s2Dao (Seasar2)】

```
Customer customer = customerDao.selectCustomer (customer);
```

【例2: jdbcManager (Seasar2)】

```
Customer result = jdbcManager  
    .from (Customer.class)  
    .getResultList ();
```

SQLを書かない

+ IDEで文法エラーを検知できる

+ 型が分かる

+ コード量が少ない

= アプリケーションエンジニアは大歓迎！！

DBのカラムとJAVAのフィールドは、JAVAのソース内で予め紐付けておきます。

```
Public class Customer implements Serializable {  
  
    @Column(name = "customer_id")  
    private String customerId;  
  
    @Column(name = "service_id")  
    private String serviceId;  
  
    @Column(name = "entry_date")  
    private Date entryDate;  
  
}
```

私たちはエクセルのマクロで、上記のようなソースコードを自動で生成するツールを作成しました。

DBを選択する ～ 決まっていること。分かっていること。

DBを選択する際に以下のようなことが決まっていました。

- 社内的にはSQL Serverのシェアが圧倒的。高い可用性を求められている場合はOracle。
- 今回の案件は開発にかけられる予算はゼロ。すべて内製。
- スモールスタートな案件である。
- Javaのフレームワーク「Seasar2」を使用したい。
- よって、「Seasar2」が標準で対応しているDBに選択肢が絞られる。

DBを選択する ～ S2DAOで使用できるDBを比較

S2DaoはRDBの設定がプリセットされています。有名なDBしか設定が用意されていないので、自由にDBを選択できるわけではありません。

RDB	Oracle	DB2	SQLServer	MySQL	PostgreSQL	Firebird
導入実績	○	×	◎	○	○	×
費用	×	×	○	◎	◎	◎
情報 書籍・インターネット	○	×	○	◎	◎	×
高可用性	◎	不明	○	○	×	不明

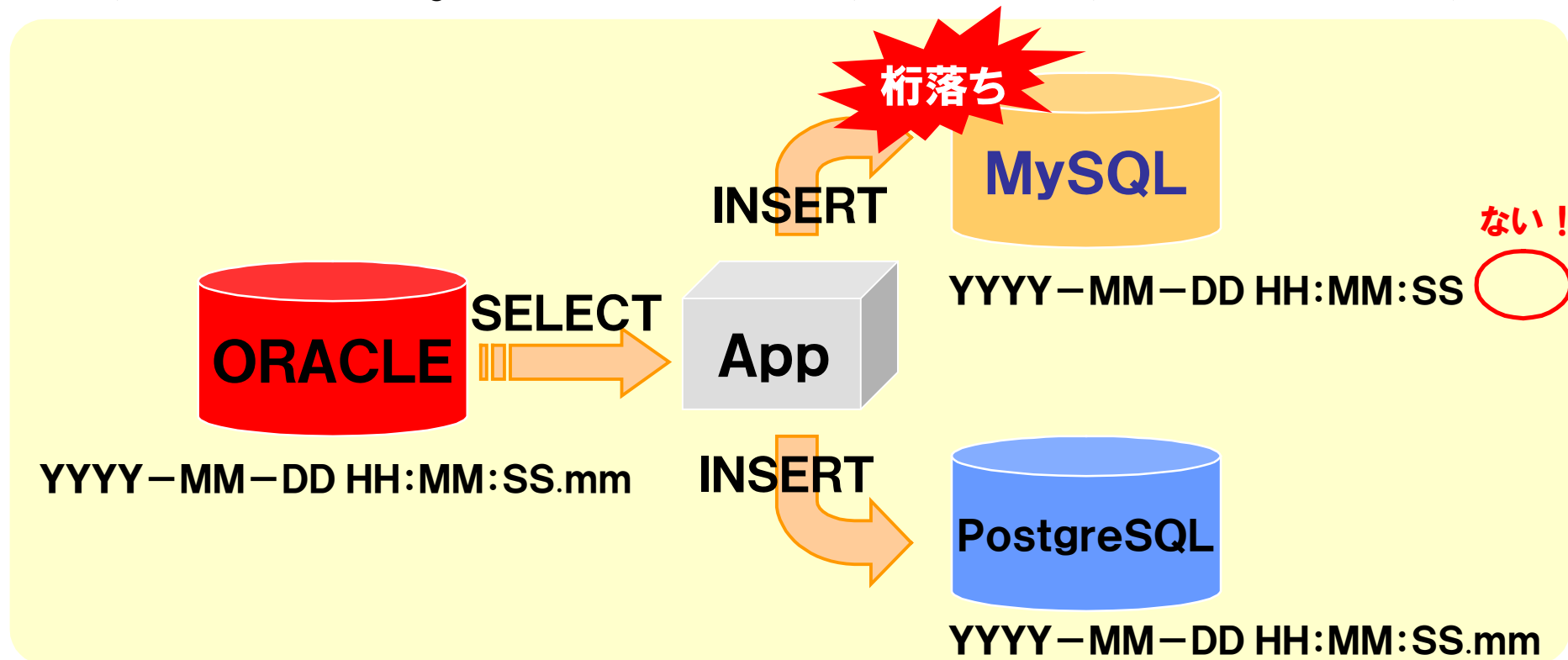
最終的に、PostgreSQLとMySQLで導入を検討しましたが、標準でレプリケーション機能があるMySQLの導入を決定しました。

DBを選択する ～ ミリ秒の壁、PostgreSQLへの移行

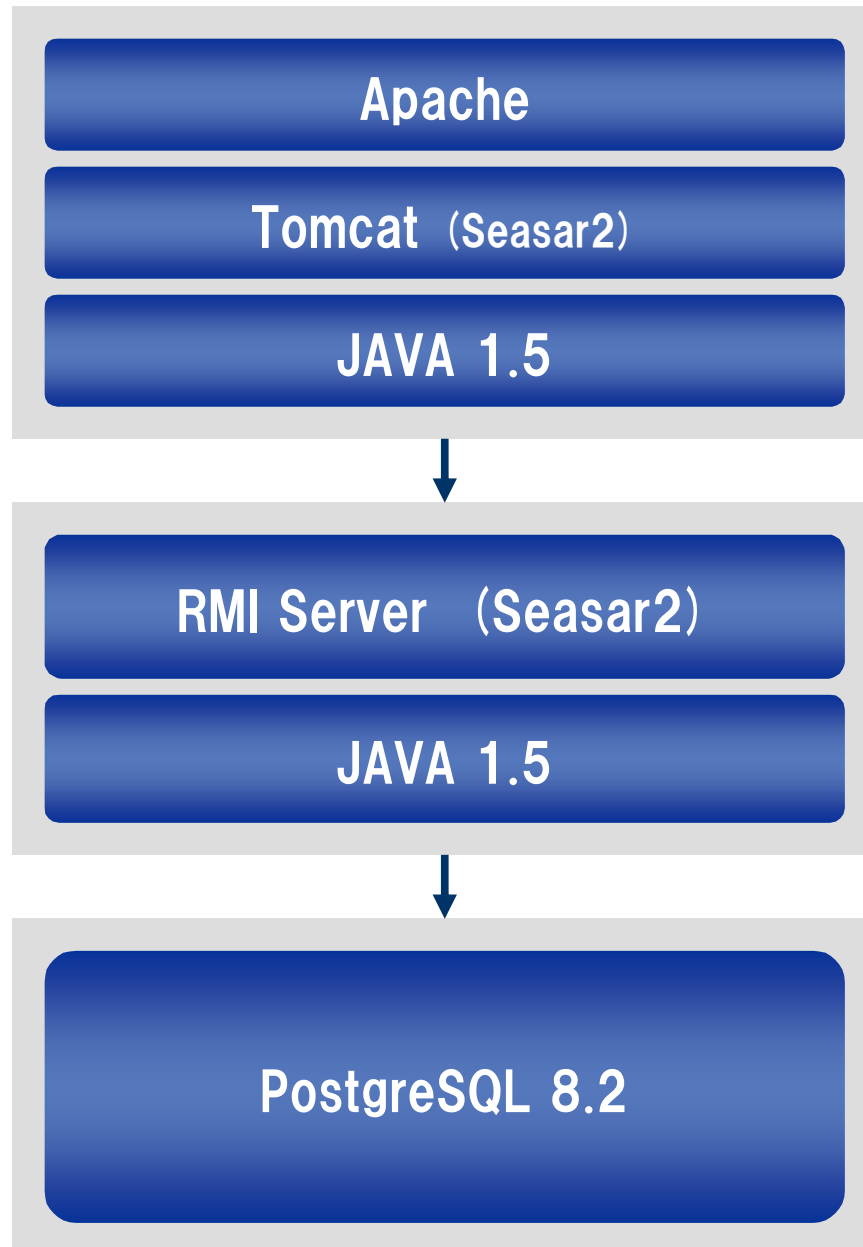
mySQLで開発を進めましたが、思わぬところで問題が生じます。それはTimestamp型でミリ秒が扱われないという問題です。

通話明細データ (CDR: Call Detail Record) を格納しているシステムとデータ連携を行う必要がありました。この対向システムはORACLEで構築されており、この中の通話時間はTimestamp型で格納されています。

そこで、PostgreSQLを試してみたところ正常にミリ秒が入ることが確認できました。これが決定打となり、PostgreSQLに変更することを決定しました。(サービスイン前です。)



DBを選択する ～ VoIPサービス管理システムの最終的な構成



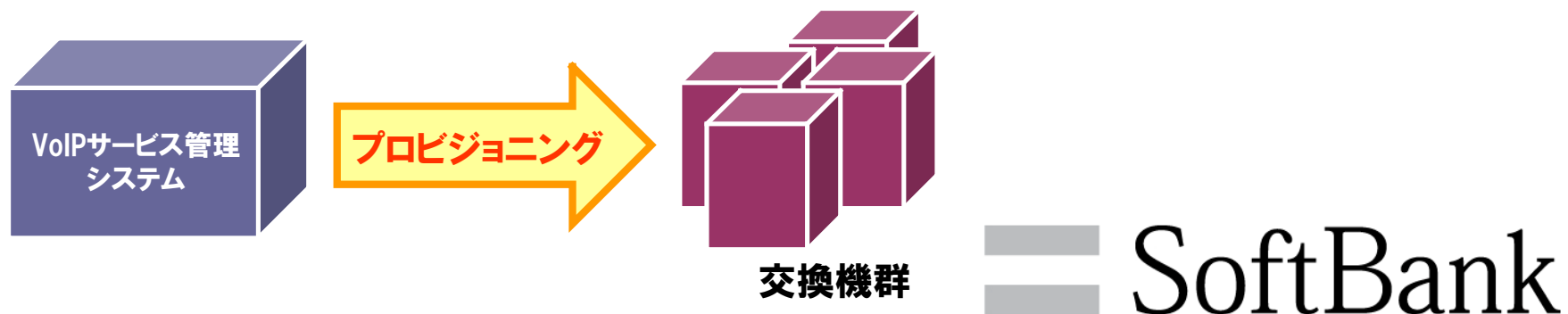
**基幹技術やDBが確定し、
最終構成は左のようになりました。**

- ※ 灰色が物理サーバを表しています。
- ※ OSは運用や監視の都合上、全てLinuxです。

== SoftBank

リアルタイムプロビジョニングの実装

- 通信業界の独特の言葉として“**プロビジョニング**”(Provisioning。以降、プロビ)という言葉が存在します。
- “**プロビ**”は交換機(CA:Call Agent)に対して、顧客の電話番号などの情報を登録する処理のことを指します。
- “**プロビ**”のプロトコルはCAにより異なり、SSHでログインして専用コマンドを実行する場合もあれば、SOAPなどのAPIが用意されている場合もあります。
- 今回対象のCAのプロトコルはSSHです。
- また、この“**プロビ**”を、本システムではリアルタイムに行うことを要求されました。



リアルタイムプロビジョニングの実装 ～ Seasar2によるトランザクション制御

- Seasar2はAOPの機能を利用してトランザクションを自動で制御してくれる機能があります。
- トランザクションの単位は、Javaのメソッド単位(下記図■)です。
- トランザクション制御は、メソッドの処理が始まると自動的に「begin transaction」が発行され、正常に終了した場合は「commit」されます。一方、途中で処理が失敗(Exception)した場合は「rollback」されます。

```
Public class ProvisioningServiceImpl implements ProvisioningService {  
  
    public boolean addAccountTx (String customerId, String account) {  
        // アカウト追加処理  
        return true;  
    }  
  
    public boolean deleteAccountTx (String customerId) {  
        // アカウト削除処理  
        return true;  
    }  
  
}
```

この機能はトランザクションの開始／終了忘れを防止することが出来、プログラマの負担を軽減してくれます。

リアルタイムプロビジョニングの実装 ～ 自動トランザクション制御の問題点

Seasar2によるトランザクション制御は、システム開発の工数軽減に貢献してくれましたが、リアルタイムプロビの実装に落とし穴がありました。

それは

DBの更新処理とプロビ処理を同じトランザクション内のメソッドに記述すると、プロビの失敗でDBがロールバックされてしまうことです。

```
Public class ProvisioningServiceImpl implements ProvisioningService {  
    private AccountLogic accountLogic;  
    private ProvisioningLogic provisioningLogic;  
  
    public boolean addAccountTx (String customerId, String account) {  
        // WEBでリアルタイムに受け付けた情報をDBに登録  
        accountLogic.insertAccount (String customerId, String account);  
        ↑ 正常終了。  
  
        // 交換機に対してプロビを行う  
        provisioningLogic.addAccount (String customerId, String account);  
        ↑ エラーが発生するとDBの処理がロールバックされる。  
        最悪中途半端にプロビされる場合も…  
  
        return true;  
    }  
}
```

リアルタイムプロビジョニングの実装 ～ 実装方法の考案

リアルプロビの実装に対して以下の3つ実装方法を考案しました。

1. リアルに処理するのはDBまで。プロビはBATCH処理にする。

- ・実装が簡単。

- ・ビジネスロジックの実装が分散する。
- ・カッコ悪い。

2. DB更新のメソッド、プロビのメソッドを分ける。

- ・ビジネスロジックの実装が分散しない。
- ・実装が簡単。

- ・WEB側のエンジニアがプロビを意識した実装を行う必要がある。
- ・プロビだけ or DB処理だけ再実行することができない。

3. 自らランザクション制御を行う。

- ・ビジネスロジックの実装が分散しない。

- ・実装が難しい。

2と3の案を採用しました！

SoftBank

リアルタイムプロビジョニングの実装 ～ 自らトランザクション制御を行う

独自にトランザクションを制御するなら、再実行が可能なシステムにしたい！
ということで以下を実装し、リアルタイムプロビを実現しました。

1. ロジックの実行順をDBで管理する。

Seasar2のDIを利用し、Javaのクラス名から実行したいバイナリ(オブジェクト)を自由に選択できる機能を実装しました。

```
// ロジック名からJAVAのオブジェクトを取得する。
Object logic = SingletonS2ContainerFactory.getContainer().getComponent("ロジック名");
// 中略
// リフレクションを使用してメソッドを実行
Object result = targetMethod.invoke(logic, 引数);
```

2. ロジックの進行状態をDBに格納する。

JAVAのメソッドの戻り値をDBに記録することでロジックの進行状態を保存することが出来ました。
PostgreSQL側の型は「bytea」に設定しておきます。

```
byteArrayOutputStream = new ByteArrayOutputStream();
objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);
objectOutputStream.writeObject(バイト配列化したいObject);
byte [] bytes = byteArrayOutputStream.toByteArray();
```

【DBに格納されているロジックの例】

ID	ロジック名 (JAVAクラス名です。)	実行順	ステータス	戻り値
1	AccountLogic	1	完了	010101...
1	ProvisioningServiceImpl	2	未実施	
1	MailLogic	3	未実施	
...

運用編

運用編について

これまでアプリケーション開発とPostgreSQLの関わりを説明させていただきましたが、ここからは運用フェーズに突入してから遭遇した事件や大工事について説明させていただきます。

1. 機密情報は暗号化

ソフトバンクはセキュリティが非常に厳しく自由に扱えません。その時の対処例を説明します。

2. pg_dump に頼っていました

開発段階で、あっさりMySQLからPostgreSQLに変更した私たちのその後のお話です。

3. 高可用性の実現

高い可用性を実現するために大工事を実施しました。

4. 殺し屋との戦い

Linuxの中の殺し屋との戦いです。

機密情報は暗号化 ～ ソフトバンクグループのセキュリティ対策

ソフトバンクグループでは、サーバにアクセスできる社員を厳しく制限しています。このアクセス制限は大きく分けて以下の3つに分割できます。

一般エリア

機密情報へのアクセスは、厳しく制限される。

セキュリティエリア

WEBアプリケーションなどのツールを用いて、機密情報へアクセスする。ツールで出来ないことは出来ない。

高セキュリティエリア

サーバに直接アクセスすることができるエリア。スーツの上着は着用禁止、水も飲めません。入室権限の取得には情報セキュリティ講習の受講が必須。

しかし、一般エリア内で顧客の傾向分析などに生のデータを活用したいのが正直な所です。セキュリティエリア用にツールを作っている時間ありません。

機密情報は暗号化 ～ すべてを暗号化する。

個人が判別できない状態であれば、一般エリア内でデータを閲覧することができます。よって、全て暗号化してしまう関数を作成し、これに対処することにしました。

※ 全テーブル暗号化関数の一部抜粋。危険な関数ですので、ご注意ください。

```
BEGIN
  FOR rec IN
    SELECT * FROM pg_stat_user_tables LOOP

      -- customer_id が存在するかチェック
      SELECT count (*) INTO customerIdCount FROM pg_attribute WHERE attrelid = rec.relid AND attname =
      'customer_id';

      -- テーブルのカラムに customer_id が存在した場合
      IF customerIdCount > 0 THEN

        -- 暗号化のSQLを準備（例えばmd5で。）
        sql := 'UPDATE ' || rec.relname || ' SET customer_id = md5 (customer_id)';
        -- 暗号化SQLを実行
        EXECUTE sql;

      END IF;
    END LOOP;
END;
```

この関数の開発により、運用者以外でも生データを容易に扱えるようになり、サービスの分析、円滑な運用に大きく貢献できました。

pg_dump に頼っていました

前述の通り、mySQLからPostgreSQLに乗り換えて、無事サービスインを迎えることが出来ました。しかし、レプリケーションなどの高可用性の技術は導入できずに、DBサーバが2台あるにもかかわらず、1台遊んでいる状態が続きました。この時は、定期的に「pg_dump」を行い有事に備えている状態でした。

また、pgcluster用に独自にコンパイルされたPostgreSQLを使用していたため、フレームワーク(S2Dao)がSQLの結果コードを拾えない場合があり、最悪アプリケーションサーバが停止することも発生していました。

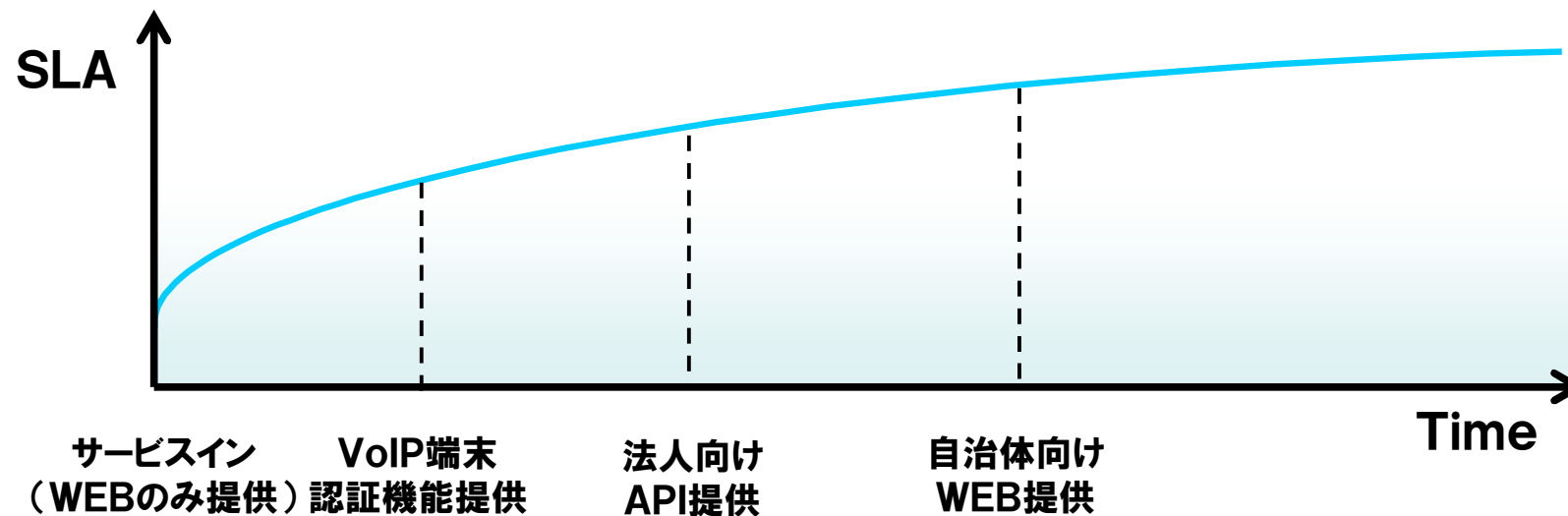
【SQLの結果コードの例】

```
=> INSERT INTO test VALUES (1, 'test');  
INSERT 0 1
```

アプリケーション製造をフレームワークに頼っているからには、フレームワークがサポートしている製品を必ず使うべきであると教えられました。

高可用性の実現 ～ SLA上昇による限界

不安定な運用が続いていましたが、自社開発・運用している柔軟性の高いシステムであったため、ひとつのDBに詰め込まれる機能が増えていきました。それに伴い求められるSLAが上昇し、アプリやDBの停止が大障害になる状態になってしまいました。

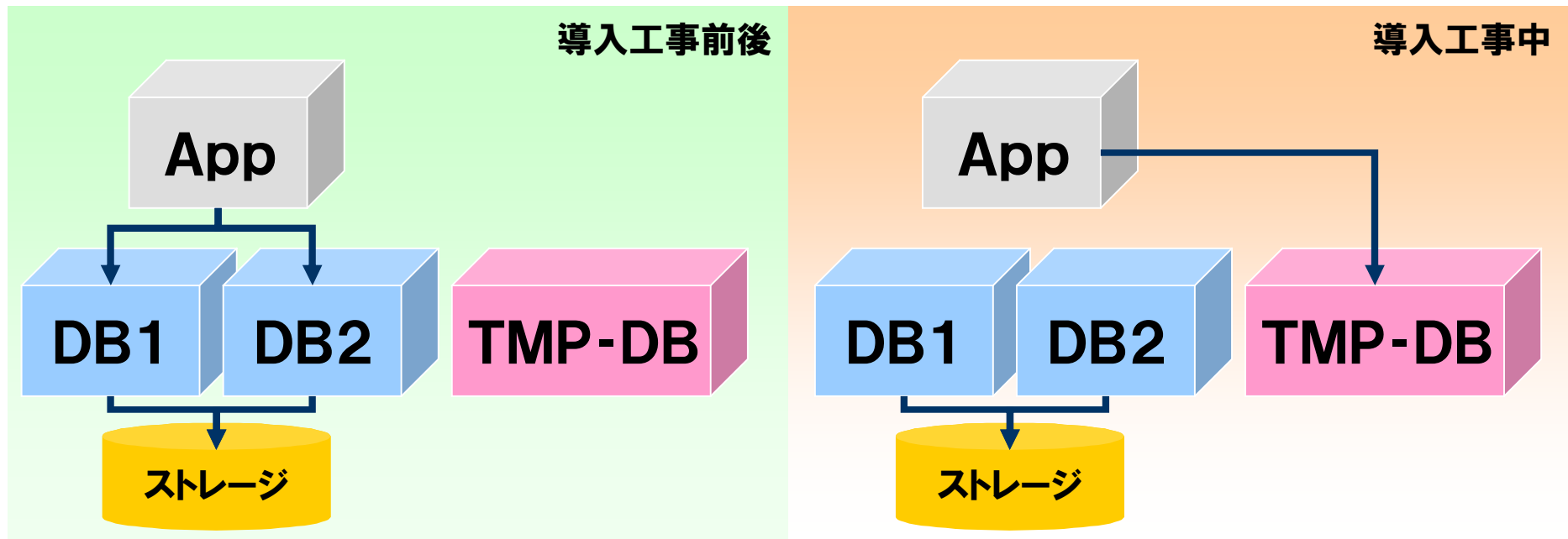


スモールスタートで始めたはずのシステムが、サービスイン後の様々な機能追加でSLAが跳ね上がってしまいました。機能追加やスケールアウトが柔軟なシステム構築を行うことが課題として残っています。

高可用性の実現 ～ PowerGres HA & LifeKeeperの導入

高可用性の実現の為、ソフトバンクでは最終的にPostgreSQLの商用版「PowerGres HA & LifeKeeper」の導入を行いました。PowerGres HAはバイナリレベルでPostgreSQLと同等であった為、アプリケーションの改修(またはS2DAOの設定変更)が不要であり導入は非常に容易でした。

PowerGres HA & LifeKeeperの構築工事は2日を要しました。この間、一時的に別のDBサーバを構築することで、サービス断を発生させずに切り替えることに成功しています。



次のスライドを除いて、現在は非常に安定したシステム運用が行えています。

殺し屋との戦い

ある年の元旦、DBサーバが停止しました。PowerGres HA & LifeKeeper の導入は、まだ行われていなかったため、大障害になりました。早速、調査を行ったところ、コンソールには以下の文字列が…

```
# cat /var/log/messages | grep postgres
Out of Memory: Killed process <PID> (postgres)
```

Linux上に潜んでいる殺し屋「OOM-Killer」に PostgreSQLのプロセスが強制終了させられた証拠でした。この時は、OOM-Killer にはJAVAも強制終了させられている為、すっかり悪友になっていました。

※ OOM-Killer

Linuxがメモリを使い切り、必要なメモリ領域を新たに確保できない場合に、プロセスを強制終了させて空きメモリを確保する仕組み。

一般的には下記のコマンドでOOM-Killerの対象から外すことができることが判明しているようですが、私たちはLinux自体が停止してしまう可能性を否定できない理由から、定期的なOS再起動を運用に加えています。現在も継続中です。

```
# echo -17 > /proc/ <PID> /oom_adj
```

LinuxはWindowsより安定している、と盲目的に信頼していましたが、それは誤りでした。実績のある環境を導入することが、運用の安定化に繋がります。