技術構造図の説明

1. 構造の設計目標

- 功能ニーズと非功能ニーズを満たす。
- コストをコントロールするために、できるだけ既存のオープンソース技術を利用する。
- 構造は簡単、セキュリティのアップグレート拡張が可能という特性を持つ。
- システム構造の範囲を全面的に、正確的に識別でき、内部システムと外部システムの間の境界およびインタラクティブ契約を定義できる。
- 各子システムの職責、各子システムの間のインタラクティブ協議やインタラクティブプロセスを確定できる。
- システム開発、部署、メンテナンスおよびその規範を確定する。
- システム設計、コーデイング、テスト、メンテナンスに原則、基礎や規範を提供する。

2. 構造の設計原則

- 功能ニーズや非功能ニーズを満たす。これはシステム開発に対して基本的な要求であり、構造設計する時もこの基本原則に準ずる。
- 実用性原則。ユーザーに使用される開発済みのシステムは、実用的、システムが安定している、性能が良いなどの条件を達成しなくてはいけない。
- 重複利用可能。標準モジュールを重複利用可能のミドルウア及び標準コンポーネントに設計され、開発効率を最大限にアップする。
- 単一職責原則。各子システムは一つの特定機能を達成することに集中し、あ互いに絡まないで、システム間はインターフェースを通してインタラクティブする。
- 拡張性原則。業務モジュール子システムはサービス提供の形で外部にインターフェースを提供し、今後業務拡張したい場合、サービスを拡張すれば対応できる。システムは業務量に従って配置するので、即ち、集中式配置或いは分散式配置両方ともできる。

3、詳細説明

3.1 APP 及び Web のサーバ通信は https で実現する。

1) **認証サーバ。** ユーザー側に信頼されている CA 機構に認証された証明書を内蔵する。第 1 段階、サーバは CA 機構に認証されたサーバ証明書を提供し、もしこのサーバ証明書を認証する CA 機構は、ユーザー側に信頼された CA 機構のリストに存在し、そしてサーバ証明書の情報は現在アクセス中のサイト(ドメイン名など) と一致する場合、このサービス側が信頼できるとユーザー側が判断し、サーバ証明書からサーバのパブリックキーを獲得し、次のプロセスに使用する。そうでなければ、ユーザー側は再び請求を発信しない。

2) **セッションキー。**ユーザー側はサーバを認証し、パブリックキーを取得した後、パブリックキーを通してサーバと通信し、2つのセッションキーを決める。1つはユーザー側からサーバに送

信するデータを暗号化するためのユーザー側セッションキーであり、もう1つはサーバからユーザー側に送信するデータを暗号化するためのサーバ側セッションキー。パブリックキー以外に、もう2つのセッションキーを用意する理由は、非対称暗号化は非常に複雑なので、対称暗号化をデータ通信に利用して、計算の資源を節約する。また、セッションキーはランダムに生成されるので、毎回交渉できたセッションキーは違っているので、安全性を高める。

3) 暗号化通信。ユーザー側とサーバ側は、通信用のセッションキーを持っているなら、お互いに送信している http はすべてセッションキーで暗号化される。これで、ネットワーク上の他のユーザーは、このユーザーがサーバと通信しているデータを盗んだり、改ざんしたりすることができなくなり、データのプライバシーや完全性を保つ。

3.2 ユーザー側の証明書の偽造を防ぐために、証明書ロック技術を利用する。

悪意のある証明書がスマートフォンにインストールされた場合、中間人の攻撃によって、ユーザーの通信が聞き盗まれたり、request や response の内容が改ざんされたりしてしまう。

スマートフォン中間人の攻撃プロセス:

1) ユーザー側を起動する時、ユーザー側はサーバ側と握手して、データ転送用の暗号をお互いに確認する。

2) 中間人がいると、握手請求をインタセプトして、ユーザーを真似してサーバに握手請求を発信する(中間人自分の暗号化規則をサーバに送る)。サーバは受け取った暗号化規則の中から暗号化アルゴリズムや HASH アルゴリズムを選ぶと同時に、自分の情報を証明書の形でユーザー側に送信する。証明書には、サイトのアドレス、パブリックキー、証明書の発行機構などの情報が含まれている。

3) この時点で、中間人はサーバ側からユーザー側に送信する証明書の情報をインタセプトして、自分の証明書情報に置き換える。

4) これで、ユーザー側に届いたのは中間人からの response なので、ユーザーは中間人の証明書を使ってリクウストを暗号化してサーバに送信する。

5) 中間人はユーザー側からのリクウストを受け取って、自分の証明書で暗号化されたリクウストを解析する。

6) 解析した情報を盗聴あるいは改ざんした後、ユーザー側を真似して、暗号化されたリクエストをサーバ側に送信する。

これは中間人からの攻撃のプロセスである。

保護方法:

1) パブリックキーをロックする。

証明書のパブリックキーをユーザー側の apk に書き込む。http 通信する時、サーバからの証明書パブリックキーは apk に書き込んだものと一致するかどうかを検査する。

2) 証明書をロックする。

パブリックキーの証明書をユーザー側の APP に保存する。これで、http 通信する時、サーバ側ではなく、ユーザー側から証明書の情報を取る。

3.3 アプリケーションゲートウェイ（Application Layer Gateway)を利用して、ロードバランス能力、溶解能力、失敗再試行を保障する。

1）ロードバランス

インターフェースのリクエスト圧力は単一のサーバインスタンスの処理能力を超える時があるので、複数のサーバインスタンスを使う必要がある。リクエスト圧力が1つのサーバインスタンスに集中しないために、ゲートウェイレベルでポーリングを実行する。すなわち、毎回のリクエストは異なるサーバインスタンスに分散される。

2）ルート溶解能力

バックグランドサービスが異常になった場合、異常を最外層に投げるのではなく、サービスを自動的に降格させる。サービス異常があった場合、私たちが事前に設定したメッセージが返事してきて、fallback の方法で、異常をある route に手配して、この route はアクセス異常問題を溶解してくれる。

3）失敗再試行

時々、インタネットあるいは別の原因でサービスが一時停止してしまう時がある。こうなった場合、サービスは再試行する。サービス異常が出た時、別のサービスインスタンスを通して再試行するか、事前に設定したメッセージが返信してくれるか、再試行のルールを設定することができる。ただし、別のサービスインスタンスでも再試行できない場合もある。例えば、プレッシャーが大きすぎると、このサーバインスタンスが一時停止したら、サービスを別のサーバインスタンスに移転する時、すべてのサーバインスタンスが潰れてしまう。

3.4 キャッシュクラスターは主にバックグラウンドとインタフェースにサービスしている。

バックグランドサービスには3つのレベルがある。

1）ゲートウェイ

主に認証、溶解、ロードバランス、失敗再試行を処理する

2） Api サービス層

トランザクションのロジック処理

3）ベースサービス

ベースデータの追加、削除、変更、検査及 first-in-first-out 処理

Api サービス層は業務のニーズに詳しいから、業務の特徴に応じてキャッシュデータの構造を決めることができ、さらに、複数のベースサービスから取得したデータを組み合わせてキャッシュ処理に使用するので、柔軟性があり、だから、一般的に、データのキャッシュ処理を API サービス層に任せる。もし、必要がある時、ベースサービスがキャッシュ処理するのもできるが、ベースサービスは非キャッシュのインターフェースを提供しなければならない。

3.5 Api サービス層に複数のデータソースを配置し、データベースをアクセスする。

1つのプロジェクトは複数のデータベースで支えている。即ち、複数のデータソースがある。複数のデータソースは次の2つのパターンがある。

1) 2つ或いは複数のデータベースはお互いに関係なく、独立している。この場合なら、2つのプロジェクトと見なして、プロジェクト開発を進む。例えば、ゲーム開発にはプラトフォームデータベース以外に、ゲームに対応するデータベースもある。
2) 2つ或いは複数のデータベースは master-slave の関係である。例えば、mysql が1つの master-master を建てて、その後にまた複数の slave が付いている。或いは MHA が建てた master-slave を利用してコピーする。
3) spring 配置ファイルを利用して複数のデータソースを直接配置する。
2つのデータベースがお互いに関係していないことに対し、spring ファイルに複数のデータソースを配置することで、トランザクションの手配をそれぞれに実現する。
4) AbstractRoutingDataSource と AOP に基づいた複数データソースの配置原理は次のようになる。DataSource 類の ThreadLocalRountingDataSource を定義し、AbstractRoutingDataSource を継承して、配置ファイルの中で、ThreadLocalRountingDataSource に master と slave のデータソースを注入する。これで、どんな場合が master データソースを選ぶか、どんな場合が slave データソースを選ぶか、AOP を通して自由に配置できる。

3.6 パブリックサービスだけがブロックチェーンや IPFS を呼び出すことができる。

ファイルを IPFS ネットワークに保存し、データを各ネットワークノードに分散する。ファイルを取得する必要がある時、ファイルを保存する際に返信してくれた hash 値のみを使用して、IPFS ネットワークに分散されたデータを検索し、組み立てれば、ファイルを獲得。

3.7 データベースマスタースレーブ複製（master-slave replication）は GTID の形を採用する。

GTID の概況

1) GTID：global transaction identifieds。
2) GTID は唯一性があり、1つのトランザクションに1つの GTID が対応する。
3) 重複発効によるデータ混乱や主従が一致しないことを防ぐために、1つの GTID はサーバで一回だけ発効する。
4) classic の代わりに、GTID のコピー方法を採用するので、binlog+pos ではなく、master_auto_postion=1 の方式で自動的に GTID のブレークポイントに合わせてコピーする。
5) MySQL-5.6.5 が支持するのが、その後 MySQL-5.6.10 が改善する。
6) 伝統的な slave 端では、binlog は開いていないが、GTID の場合、slave 端の binlog を開ける必要があり、実行した GTID(強制)を記録しなければならないからだ。

GTID の構成部分

前には server_uuid 後ろにはシリアナンバー
例えば: server_uuid:sequence number
7800a22c-95ae-11e4-983d-080027de205a:10
UUID:1つの mysql インスタンスの唯一 id を slave に伝わるので、ソース ID と理解しても良い。

Sequence number 各 MySQL サーバは、1 から成長してシリアまで形成するので、1つの数値は1つのトランザクションに対応する。

従来のコピーに比べて、GTID のメリット

1) より簡単に failover を実現できる。log_file と log_Pos を探す必要がなくなる。

2) 主従コピーをより簡単に実現する。

3) 従来のコピーのほうより安全性が高い

4) GTID は連続して空洞がないので、データ衝突が発生した場合、空き物を追加する方法でスキープすることができる。

GTID の原理

1) master がデータ更新する時、トランザクションの前に GTID を形成し、binlog に記録する。

2) slave 端の i/o スレッドが変更した binlog を、ローカルの relay log に書き込む。

3) sql スレッドは relay log から GTID を取得して、slave 端の binlog に記録されたかどうかをチェックする

4) 記録があれば、この GTID のトランザクションが実行したと判断し、slave はスキープする。

5) 記録がなければ、slave は relay log の中でこの GTID のトランザクションを実行し、そして binlog に記録する。

6) 解析する時、メインキーがあるかどうかを判断する。なければ、2 級インデックス或いは全面スキャンを利用する。