link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=173 sentences=397, words=2737

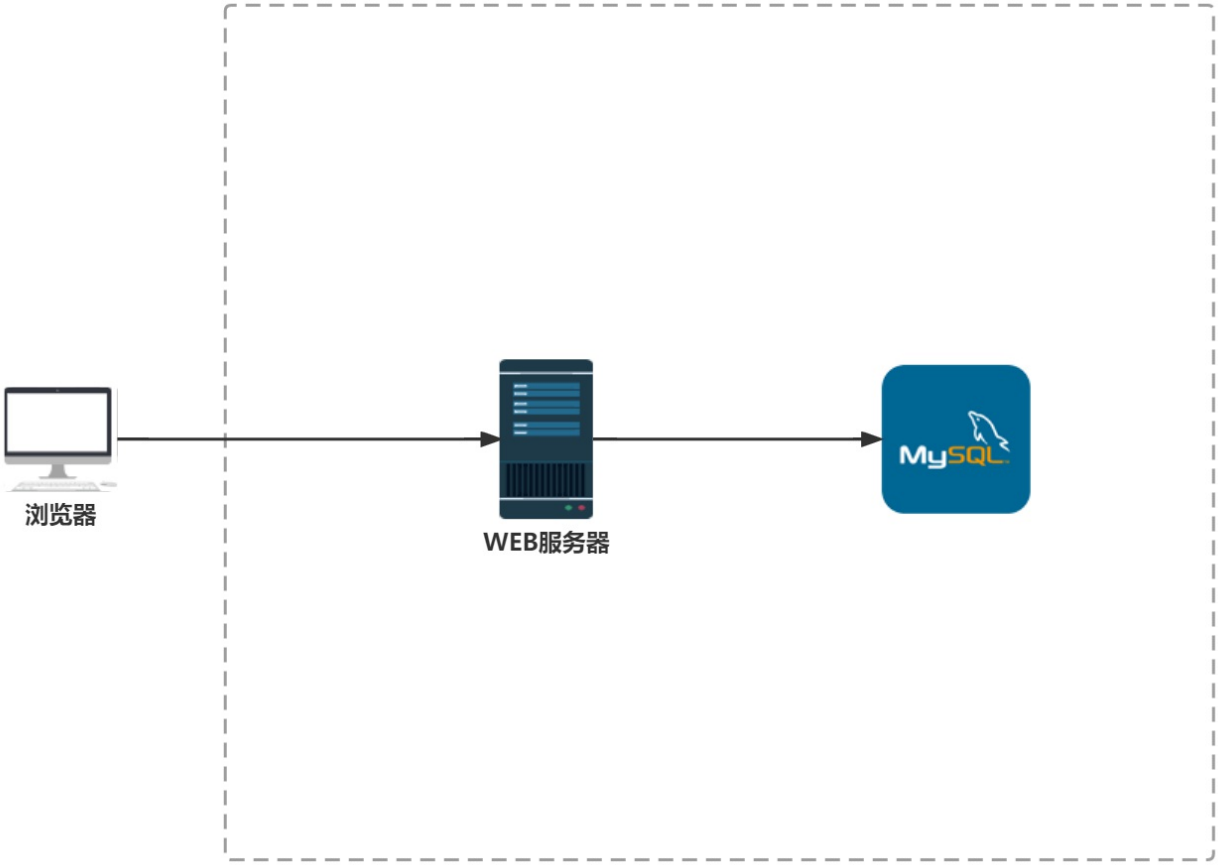
1.课程大纲

- BFF架构演进
- RPC高性性能BFF实战
- DDD和GraphQL实战BFF
- Serverless实战BFF

2.BFF架构演进

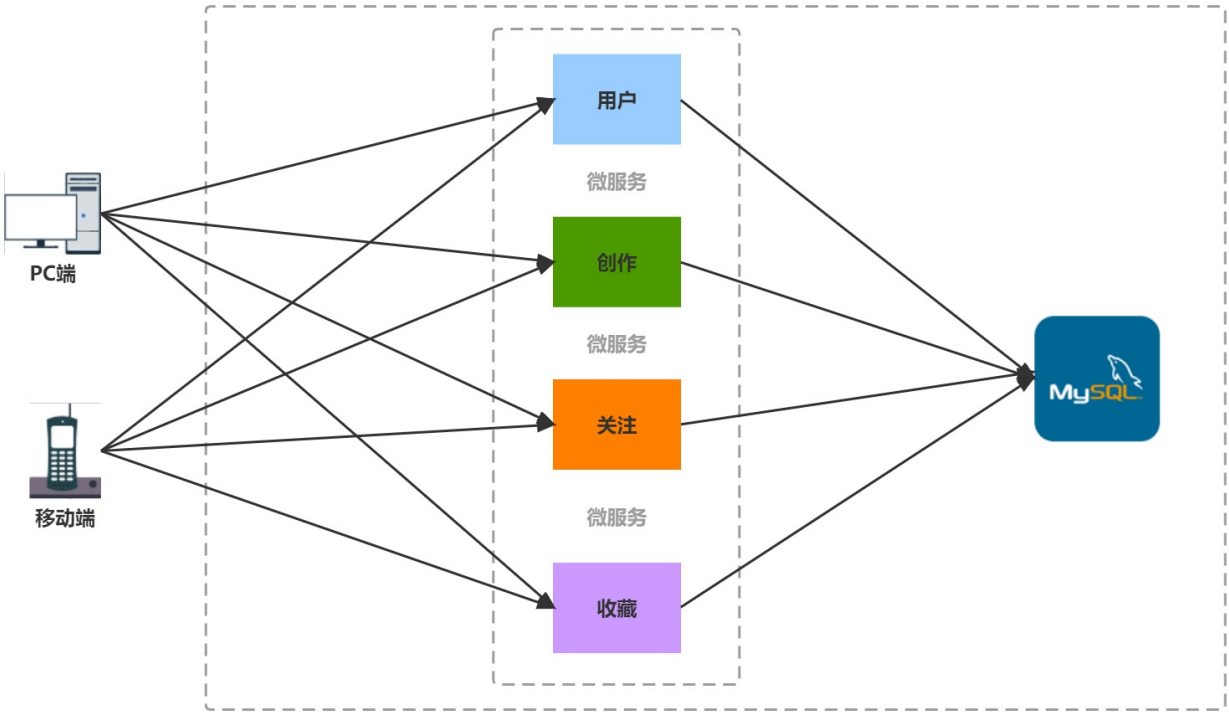
2.1 单体服务

- 单体服务是指一个独立的应用程序，包含了所有的功能和业务逻辑。这种架构方式在小型应用程序中很常见
- 随着应用程序的功能越来越多，代码库也会越来越大，维护起来也会变得更加困难。此外，单体服务的整体复杂度也会增加，这可能导致软件开发周期变长，质量下降，并且系统的扩展性也会受到限制



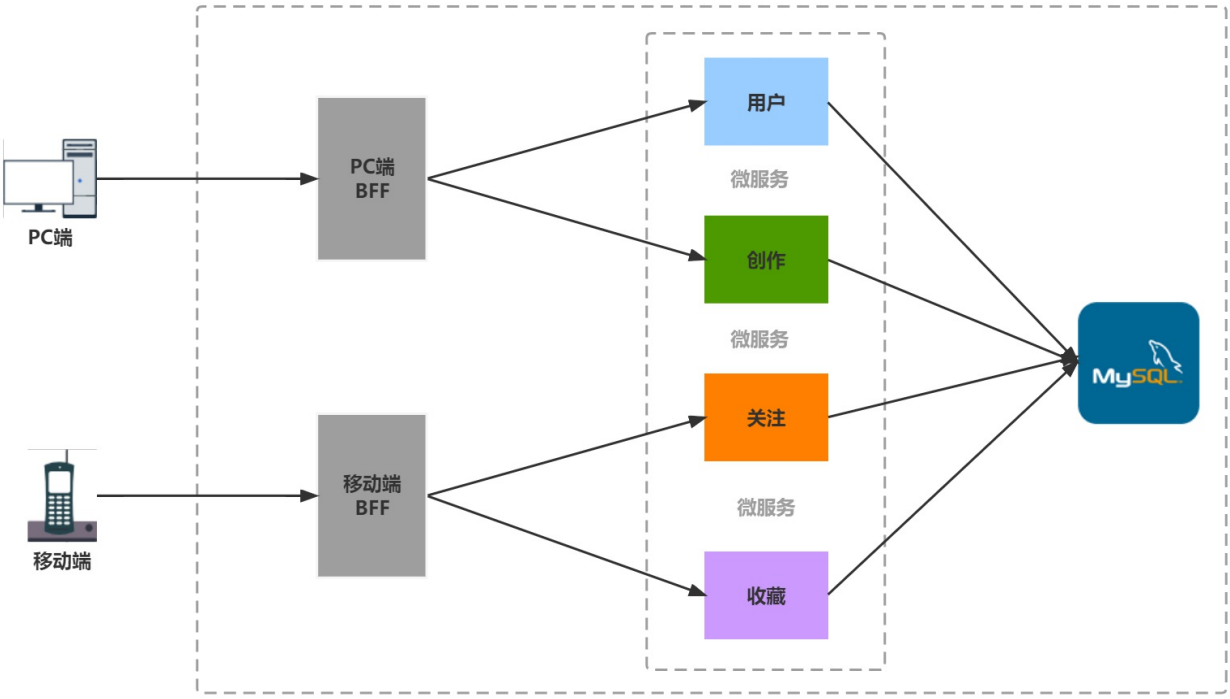
2.2 微服务

- 为了应对这些问题，许多公司开始使用微服务架构。微服务是指将一个大型应用程序拆分成若干个小型服务，每个服务负责执行特定的任务。这种架构方式可以帮助公司更快地开发和部署新功能，并提高系统的可扩展性和可维护性
- 这种方式会有以下问题
    - 域名开销增加
        - 内部服务器暴露在公网，有安全隐患
        - 各个端有大量的个性化需求
            - 数据聚合 某些功能可能需要调用多个微服务进行组合
                - 数据裁剪 后端服务返回的数据可能需要过滤掉一些敏感数据
                - 数据适配 后端返回的数据可能需要针对不同端进行数据结构的适配，后端返回 XML，但前端需要 JSON
                - 数据鉴权 不同的客户端有不同的权限要求



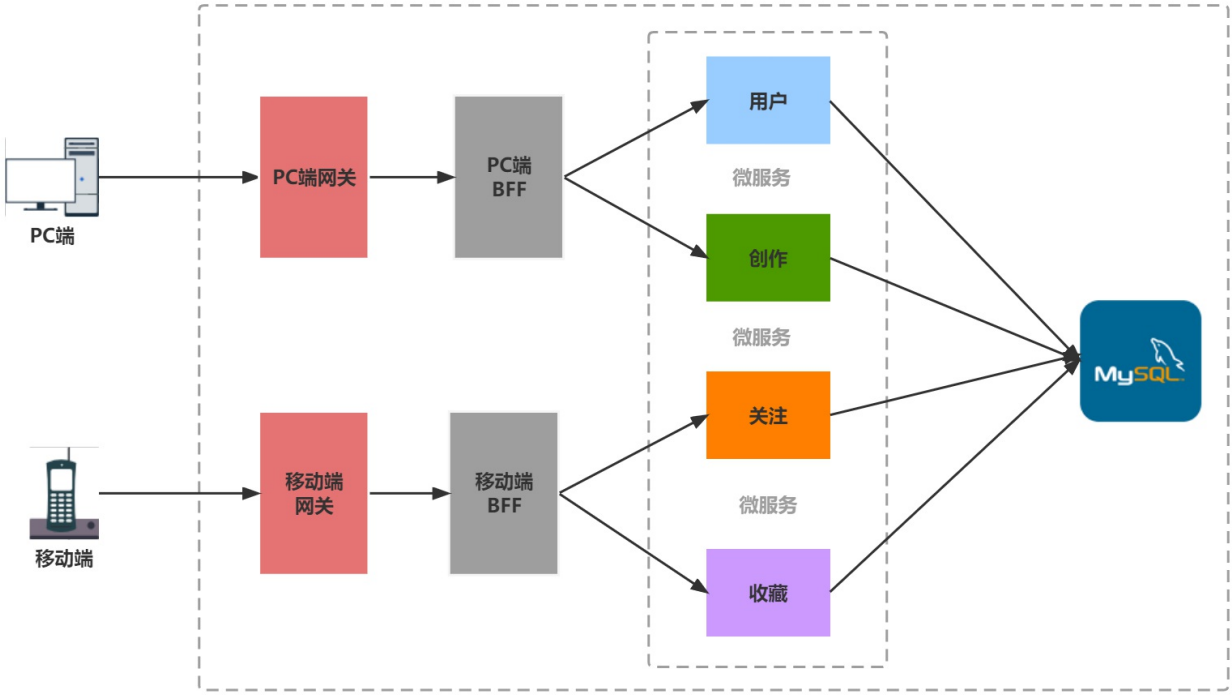
2.3 BFF

- BFF是 Backend for Frontend的缩写，指的是专门为前端应用设计的后端服务
- 主要用来为各个端提供代理数据聚合、裁剪、适配和鉴权服务，方便各个端接入后端服务
- BFF可以把前端和微服务进行解耦，各自可以独立演进



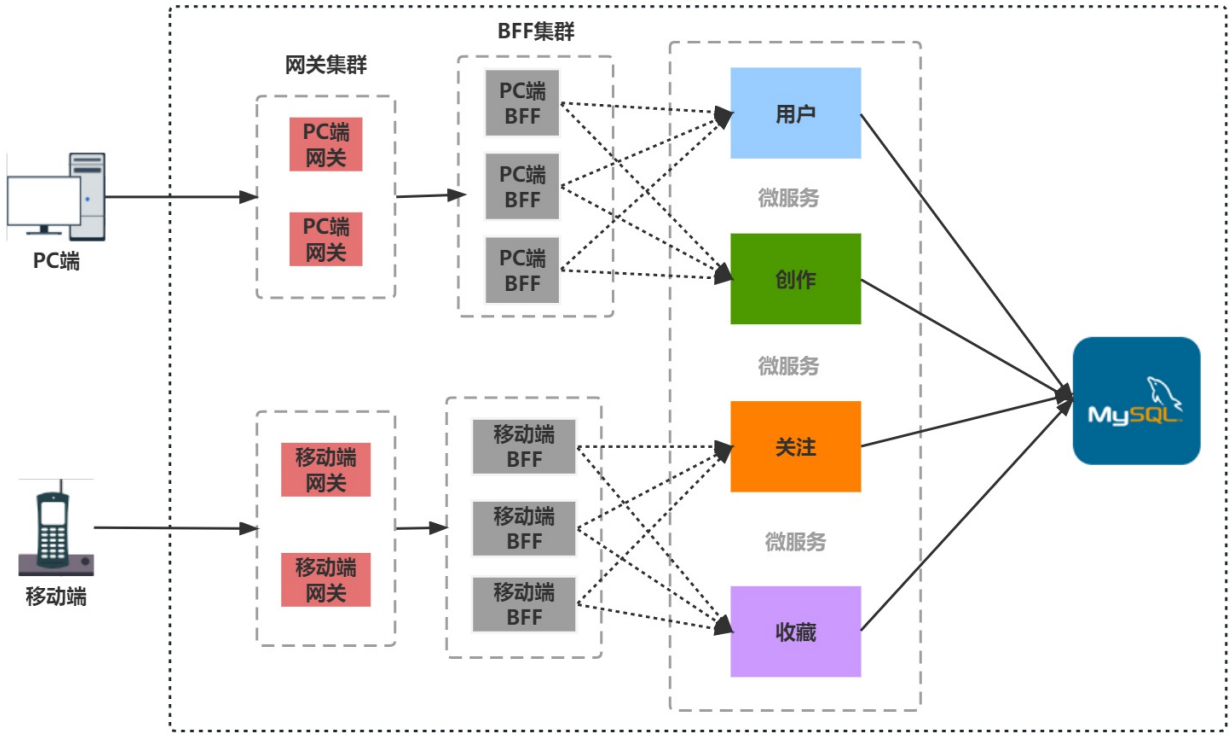
2.4 网关

- API 网关是一种用于在应用程序和 API 之间提供安全访问的中间层
- API 网关还可以用于监控 API 调用，路由请求，以及在请求和响应之间添加附加功能（例如身份验证，缓存，数据转换，压缩、流量控制、限流熔断、防爬虫等）
- 网关和BFF可能合二为一



2.5 集群化

- 单点服务器可能会存在以下几个问题：
  - 单点故障：单点服务器只有一台，如果这台服务器出现故障，整个系统都会停止工作，这会导致服务中断
  - 计算能力有限：单点服务器的计算能力是有限的，无法应对大规模的计算需求
  - 可扩展性差：单点服务器的扩展能力有限，如果想要提升计算能力，就必须改造或者替换现有的服务器
- 这些问题可以通过采用服务器集群的方式来解决



3.创建用户微服务

3.1 微服务

- 微服务是一种架构模式，它将单个应用程序划分为小的服务，每个服务都独立运行并且可以使用不同的语言开发。这种架构模式使得应用程序变得更容易开发、维护和扩展
- 微服务架构通常会有许多不同的服务，这些服务可能位于不同的机器上，因此需要使用某种通信协议来进行通信
- 因为 RPC协议比 HTTP协议具有更低的延迟和更高的性能，所以用的更多

3.2 RPC

- RPC（Remote Procedure Call）是远程过程调用的缩写，是一种通信协议，允许程序在不同的计算机上相互调用远程过程，就像调用本地过程一样

3.3 sofa-rpc-node

- sofa-rpc-node 是基于 Node.js 的一个 RPC 框架，支持多种协议

3.4 Protocol Buffers

- Protocol Buffers（简称 protobuf）是 Google 开发的一种数据序列化格式，可以将结构化数据序列化成二进制格式，并能够跨语言使用

3.5 Zookeeper

3.5.1 简介

- ZooKeeper 是一个分布式协调服务，提供了一些简单的分布式服务，如配置维护、名字服务、组服务等。它可以用于管理分布式系统中的数据
- Apache Zookeeper 官网 (https://zookeeper.apache.org/releases.html)

3.5.2 安装启动

- 1. 下载 Zookeeper 安装包，可以从Apache Zookeeper 官网 (https://zookeeper.apache.org/releases.html)下载最新版本的安装包
- 1. 解压安装包，将下载的压缩包解压到指定的目录。
- 1. 配置环境变量，将 Zookeeper 安装目录添加到环境变量中
- 1. 修改配置文件，在安装目录下的 conf 目录中找到 zookeeper.properties 文件，修改相关配置
- 1. 启动 Zookeeper，在安装目录下运行命令 bin\zkServer.cmd 即可启动 Zookeeper

zookeeper\conf\zoo.cfg

```
+dataDir=./data
```

3.6 启动服务

- logger 是日志记录器，用于记录服务器运行时的日志信息
- registry 是一个注册中心，用于维护服务的注册信息，帮助服务节点和客户端找到对方。
- server 表示服务端。服务端是提供服务的节点，它会将自己所提供的服务注册到注册中心，并等待客户端的调用。服务端通常会实现具体的业务逻辑，并使用 RPC 或其他通信协议与客户端进行通信
- server 的 addService 方法接受两个参数：服务接口和服务实现。服务接口是一个对象，其中包含了服务的名称信息。服务实现是一个对象，其中包含了具体实现服务方法的函数
- RPC 服务器的 start 方法，用于启动服务器
- RPC 服务器的 publish 方法，用于向注册中心注册服务。这样，客户端就可以通过注册中心获取服务的地址和端口，并直接向服务器发起调用

3.6.1 安装

```
npm install mysql2 sofa-rpc-node --save
```

3.6.2 user\package.json

user\package.json

```
{
    "name": "user",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
+ "scripts": {
+     "dev": "nodemon index.js"
+ },
    "keywords": [],
    "author": "",
    "license": "MIT",
    "dependencies": {
        "sofa-rpc-node": "^2.8.0"
    }
}
```

3.6.3 user\index.js

user\index.js

```
const { server: { RpcServer }, registry: { ZookeeperRegistry } } = require('sofa-rpc-node');
const mysql = require('mysql2/promise');
let connection;

const logger = console;

const registry = new ZookeeperRegistry({
    logger,
    address: '127.0.0.1:2181',
    connectTimeout: 1000 * 60 * 60 * 24,
});

const server = new RpcServer({
    logger,
    registry,
    port: 10000
});

server.addService({
    interfaceName: 'com.zhufeng.user'
}, {
    async getUserInfo(userId) {
        const [rows] = await connection.execute(`SELECT id,username,avatar,password,phone FROM user WHERE id=${userId} limit 1`);
        return rows[0];
    }
});

(async function () {
    connection = await mysql.createConnection({
        host: 'localhost',
        user: 'root',
        password: 'root',
        database: 'bff'
    });
    await server.start();
    await server.publish();
     console.log(`用户微服务发布成功`);
})();
```

3.6.4 client.js

user\client.js

```
const { client: { RpcClient }, registry: { ZookeeperRegistry } } = require('sofa-rpc-node');

const logger = console;

const registry = new ZookeeperRegistry({
    logger,
    address: '127.0.0.1:2181',
});
(async function () {

    const client = new RpcClient({ logger, registry });

    const userConsumer = client.createConsumer({

        interfaceName: 'com.zhufeng.user'
    });

    await userConsumer.ready();

    const result = await userConsumer.invoke('getUserInfo', [1], { responseTimeout: 3000 });

    console.log(result);
    process.exit(0);
})()
```

4.创建文章微服务

4.1 安装

```
npm install mysql2 sofa-rpc-node --save
```

4.2 article\package.json

article\package.json

```
{
    "name": "user",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
+    "dev": "nodemon index.js"
    },
    "keywords": [],
    "author": "",
    "license": "MIT",
    "dependencies": {
        "mysql2": "^2.3.3",
        "sofa-rpc-node": "^2.8.0"
    }
}
```

4.3 article\index.js

article\index.js

```
const { server: { RpcServer }, registry: { ZookeeperRegistry } } = require('sofa-rpc-node');
const mysql = require('mysql2/promise');
let connection;

const logger = console;

const registry = new ZookeeperRegistry({
    logger,
    address: '127.0.0.1:2181',
    connectTimeout: 1000 * 60 * 60 * 24,
});

const server = new RpcServer({
    logger,
    registry,
    port: 20000
});

server.addService({
    interfaceName: 'com.zhufeng.post'
}, {
    async getPostCount(userId) {
        const [rows] = await connection.execute(`SELECT count(*) as postCount FROM post WHERE user_id=${userId} limit 1`);
        return rows[0].postCount;
    }
});

(async function () {
    connection = await mysql.createConnection({
        host: 'localhost',
        user: 'root',
        password: 'root',
        database: 'bff'
    });
    await server.start();
    await server.publish();
    console.log(`文章微服务发布成功`);
})();
```

4.4 client.js

article\client.js

```
const { client: { RpcClient }, registry: { ZookeeperRegistry } } = require('sofa-rpc-node');

const logger = console;

const registry = new ZookeeperRegistry({
    logger,
    address: '127.0.0.1:2181',
});
(async function () {

    const client = new RpcClient({ logger, registry });

    const consumer = client.createConsumer({

        interfaceName: 'com.zhufeng.post'
    });

    await consumer.ready();

    const result = await consumer.invoke('getPostCount', [1], { responseTimeout: 3000 });

    console.log(result);
    process.exit(0);
})()
```

5.创建BFF

5.1 安装

```
npm install koa koa-router koa-logger sofa-rpc-node lru-cache ioredis amqplib fs-extra --save
```

访问地址

```
http:
```

5.2 bff\package.json

bff\package.json

```
{
    "name": "bff",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
+    "dev": "nodemon index.js",
+    "start": "pm2 start index.js --name bff"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {
        "koa": "^2.14.1",
        "koa-logger": "^3.2.1",
        "koa-router": "^12.0.0",
        "sofa-rpc-node": "^2.8.0"
    }
}
```

5.3 bff\index.js

bff\index.js

```
const Koa = require('koa');
const router = require('koa-router')();
const logger = require('koa-logger');
const rpcMiddleware = require('./middleware/rpc');
const app = new Koa();
app.use(logger());
app.use(rpcMiddleware({

    interfaceNames: [
        'com.zhufeng.user',
        'com.zhufeng.post'
    ]
}));
router.get('/', async ctx => {
    const userId = ctx.query.userId;
    const { rpcConsumers: { user, post } } = ctx;
    const [userInfo, postCount] = await Promise.all([
        user.invoke('getUserInfo', [userId]),
        post.invoke('getPostCount', [userId])
    ]);
    ctx.body = { userInfo, postCount }
});
app.use(router.routes()).use(router.allowedMethods());
app.listen(3000, () => {
    console.log('bff server is running at 3000');
});
```

5.4 rpc.js

bff\middleware\rpc.js

```
const { client: { RpcClient }, registry: { ZookeeperRegistry } } = require('sofa-rpc-node');
const rpcMiddleware = (options = {}) => {
    return async function (ctx, next) {
        const logger = options.logger || console;

        const registry = new ZookeeperRegistry({
            logger,
            address: options.address || '127.0.0.1:2181',
        });

        const client = new RpcClient({ logger, registry });
        const interfaceNames = options.interfaceNames || [];
        const rpcConsumers = {};
        for (let i = 0; i < interfaceNames.length; i++) {
            const interfaceName = interfaceNames[i];

            const consumer = client.createConsumer({
                interfaceName,
            });

            await consumer.ready();
            rpcConsumers[interfaceName.split('.').pop()] = consumer;
        }
        ctx.rpcConsumers = rpcConsumers;
        await next();
    }
};
module.exports = rpcMiddleware;
```

5.5 bff\index.js

数据处理

```
const Koa = require('koa');
const router = require('koa-router')();
const logger = require('koa-logger');
const rpcMiddleware = require('./middleware/rpc');
const app = new Koa();
app.use(logger());
app.use(rpcMiddleware({
    interfaceNames: [
        'com.zhufeng.user',
        'com.zhufeng.post'
    ]
}));
router.get('/', async ctx => {
    const userId = ctx.query.userId;
    const { rpcConsumers: { user, post } } = ctx;
    const [userInfo, postCount] = await Promise.all([
        user.invoke('getUserInfo', [userId]),
        post.invoke('getPostCount', [userId])
    ]);
+ // 裁剪数据
+ delete userInfo.password;
+ // 数据脱敏
+ userInfo.phone = userInfo.phone.replace(/(\d{3})\d{4}(\d{4})/, '$1****$2');
+ // 数据适配
+ userInfo.avatar = "http://www.zhufengpeixun.cn/"+userInfo.avatar,
    ctx.body = { userInfo, postCount }
});
app.use(router.routes()).use(router.allowedMethods());
app.listen(3000, () => {
    console.log('bff server is running at 3000');
});
```

6.缓存

- BFF 作为前端应用和后端系统之间的抽象层，承担了大量的请求转发和数据转换工作。使用多级缓存可以帮助 BFF 减少对后端系统的访问，从而提高应用的响应速度
- 当 BFF 收到一个请求时，首先会检查内存缓存中是否存在对应的数据，如果有就直接返回数据。如果内存缓存中没有数据，就会检查Redis缓存，如果Redis缓存中有数据就返回数据，并将数据写入内存缓存。如果本地缓存中也没有数据，就会向后端系统发起请求，并将数据写入Redis缓存和内存缓存

6.1 多级缓存

- 多级缓存（multi-level cache）是指系统中使用了多个缓存层来存储数据的技术。这些缓存层的优先级通常是依次递减的，即最快的缓存层位于最顶层，最慢的缓存层位于最底层

6.2 LRU

- LRU（Least Recently Used）是一种常用的高速缓存淘汰算法，它的原理是将最近使用过的数据或页面保留在缓存中，而最少使用的数据或页面将被淘汰。这样做的目的是为了最大化缓存的命中率，即使用缓存尽可能多地满足用户的请求

6.3 redis

- Redis 是一种开源的内存数据存储系统，可以作为数据库、缓存和消息中间件使用
- Redis 运行在内存中，因此它的读写速度非常快
- ioredis 是一个基于 Node.js 的 Redis 客户端，提供了对 Redis 命令的高度封装和支持
- redis (https://github.com/tporadowski/redis/releases)
- Redis-x64-5.0.14.1 (https://static.zhufengpeixun.com/Redisx6450141_1673102444438.zip)

6.4 使用缓存

6.4.1 bff\index.js

bff\index.js

```
const Koa = require('koa');
const router = require('koa-router')();
const logger = require('koa-logger');
const rpcMiddleware = require('./middleware/rpc');
+const cacheMiddleware = require('./middleware/cache');
const app = new Koa();
app.use(logger());
app.use(rpcMiddleware({
    interfaceNames: [
        'com.zhufeng.user',
        'com.zhufeng.post'
    ]
}));
+app.use(cacheMiddleware({}));
router.get('/profile', async ctx => {
    const userId = ctx.query.userId;
    const { rpcConsumers: { user, post } } = ctx;
+ const cacheKey = `${ctx.method}#${ctx.path}#${userId}`;
+ let cacheData = await ctx.cache.get(cacheKey);
+ if (cacheData) {
+     ctx.body = cacheData;
+     return;
+ }
    const [userInfo, postCount] = await Promise.all([
        user.invoke('getUserInfo', [userId]),
        post.invoke('getPostCount', [userId])
    ]);
  // 裁剪数据
  delete userInfo.password;
  // 数据脱敏
  userInfo.phone = userInfo.phone.replace(/(\d{3})\d{4}(\d{4})/, '$1****$2');
  // 数据适配
  userInfo.avatar = "http://www.zhufengpeixun.cn/" + userInfo.avatar;
+ cacheData = { userInfo, postCount };
+ await ctx.cache.set(cacheKey, cacheData);// keys *
+ ctx.body = cacheData
});
app.use(router.routes()).use(router.allowedMethods());
app.listen(3000, () => {
    console.log('bff server is running at 3000');
});
```

6.4.2 cache.js

bff\middleware\cache.js

```
const LRUCache = require('lru-cache');
const Redis = require('ioredis');
class CacheStore {
    constructor() {
        this.stores = [];
    }
    add(store) {
        this.stores.push(store);
        return this;
    }
    async get(key) {
        for (const store of this.stores) {
            const value = await store.get(key);
            if (value !== undefined) {
                return value;
            }
        }
    }
    async set(key, value) {
        for (const store of this.stores) {
            await store.set(key, value);
        }
    }
}
class MemoryStore {
    constructor() {
        this.cache = new LRUCache({
            max: 100,
            ttl: 1000 * 60 * 60 * 24
        });
    }
    async get(key) {
        return this.cache.get(key);
    }
    async set(key, value) {
        this.cache.set(key, value);
    }
}
class RedisStore {
    constructor(options) {
        this.client = new Redis(options);
    }
    async get(key) {
        let value = await this.client.get(key);
        return value ? JSON.parse(value) : undefined;
    }
    async set(key, value) {
        await this.client.set(key, JSON.stringify(value));
    }
}
const cacheMiddleware = (options = {}) => {
    return async function (ctx, next) {
        const cacheStore = new CacheStore();
        cacheStore.add(new MemoryStore());
        const redisStore = new RedisStore(options);
        cacheStore.add(redisStore);
        ctx.cache = cacheStore;
        await next();
    };
};
module.exports = cacheMiddleware;
```

7.消息队列

- 消息队列（Message Queue）用于在分布式系统中传递数据。它的特点是可以将消息发送者和接收者解耦，使得消息生产者和消息消费者可以独立的开发和部署

7.1 引入原因

- 在 BFF 中使用消息队列（message queue）有几个原因：
    - 大并发：消息队列可以帮助应对大并发的请求，BFF 可以将请求写入消息队列，然后后端服务可以从消息队列中读取请求并处理
    - 解耦：消息队列可以帮助解耦 BFF 和后端服务，BFF 不需要关心后端服务的具体实现，只需要将请求写入消息队列，后端服务负责从消息队列中读取请求并处理
    - 异步：消息队列可以帮助实现异步调用，BFF 可以将请求写入消息队列，然后立即返回响应给前端应用，后端服务在后台处理请求
        - 流量削峰：消息队列可以帮助流量削峰，BFF 可以将请求写入消息队列，然后后端服务可以在合适的时候处理请求，从而缓解瞬时高峰流量带来的压力

7.2 RabbitMQ

- RabbitMQ是一个消息代理，它可以用来在消息生产者和消息消费者之间传递消息
- RabbitMQ的工作流程如下：
    - 消息生产者将消息发送到 RabbitMQ服务器
    - RabbitMQ服务器将消息保存到队列中
    - 消息消费者从队列中读取消息
    - 当消息消费者处理完消息后 RabbitMQ服务器将消息删除
- 安装启动
    - 在RabbitMQ下载官网安装包 (https://www.rabbitmq.com/install-windows.html#installer)或镜像安装包 (https://static.zhufengpeixun.com/rabbitmqserver3116_1673104196680.exe)
    - 双击安装包，按照提示进行安装,直接就可以启动
        - 安装前还要安装Erlang (https://www.erlang.org/downloads),Erlang是一个结构化，动态类型编程语言，内建并行计算支持

7.3 实现

7.3.2 bff\index.js

bff\index.js

```
const Koa = require('koa');
const router = require('koa-router')();
const logger = require('koa-logger');
const rpcMiddleware = require('./middleware/rpc');
const cacheMiddleware = require('./middleware/cache');
+const mqMiddleware = require('./middleware/mq');
const app = new Koa();
app.use(logger());
app.use(rpcMiddleware({
    interfaceNames: [
        'com.zhufeng.user',
        'com.zhufeng.post'
    ]
}));
app.use(cacheMiddleware({}));
+app.use(mqMiddleware({ url: 'amqp://localhost' }));
router.get('/profile', async ctx => {
    const userId = ctx.query.userId;
+    ctx.channels.logger.sendToQueue('logger', Buffer.from(JSON.stringify({
+        method: ctx.method,
+        path: ctx.path,
+        userId
+    })));
    const { rpcConsumers: { user, post } } = ctx;
    const cacheKey = `${ctx.method}#${ctx.path}#${userId}`;
    let cacheData = await ctx.cache.get(cacheKey);
    if (cacheData) {
        ctx.body = cacheData;
        return;
    }
    const [userInfo, postCount] = await Promise.all([
        user.invoke('getUserInfo', [userId]),
        post.invoke('getPostCount', [userId])
    ]);
      // 裁剪数据
  delete userInfo.password;
  // 数据脱敏
  userInfo.phone = userInfo.phone.replace(/(\d{3})\d{4}(\d{4})/, '$1****$2');
  // 数据适配
  userInfo.avatar = "http://www.zhufengpeixun.cn/" + userInfo.avatar,
    cacheData = { userInfo, postCount };
  await ctx.cache.set(cacheKey, cacheData);// keys *
  ctx.body = cacheData
});
app.use(router.routes()).use(router.allowedMethods());
app.listen(3000, () => {
    console.log('bff server is running at 3000');
});
```

7.3.2 mq.js

bff\middleware\mq.js

```
const amqp = require('amqplib');
const mqMiddleware = (options = {}) => {
    return async (ctx, next) => {

        const rabbitMQClient = await amqp.connect(options.url || 'amqp://localhost');

        const logger = await rabbitMQClient.createChannel();

        await logger.assertQueue('logger');
        ctx.channels = {
            logger
        };
        await next();
    };
};
module.exports = mqMiddleware;
```

7.3.3 bff\logger.js

bff\logger.js

```
const amqplib = require('amqplib');
const fs = require('fs-extra');
const path = require('path');
(async () => {
    const conn = await amqplib.connect('amqp://localhost');
    const loggerChannel = await conn.createChannel();
    await loggerChannel.assertQueue('logger');
    loggerChannel.consume('logger', async (event) => {
        const message = JSON.parse(event.content.toString());
        await fs.appendFile(path.join(__dirname, 'logger.txt'), JSON.stringify(message) + '\n');
    });
})();
```

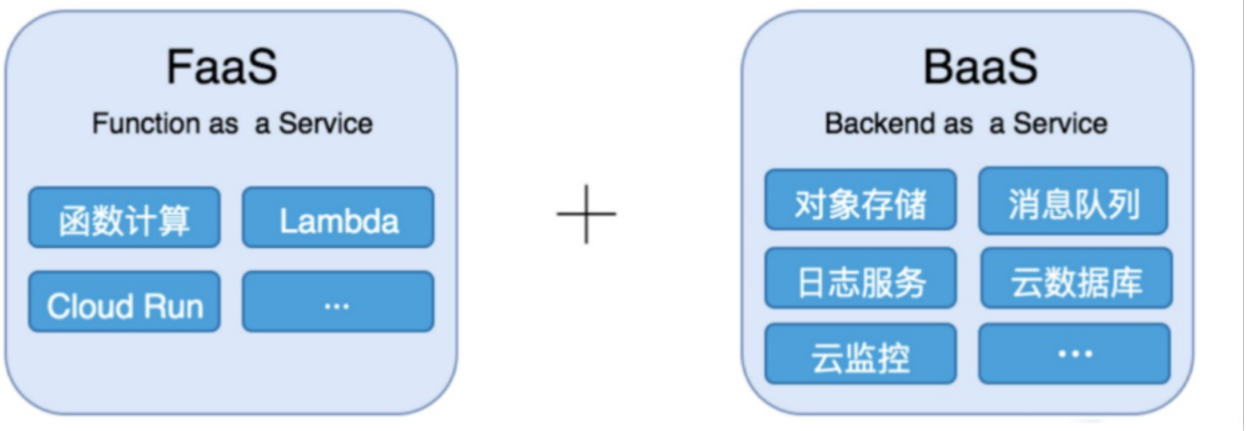
8.Serverless

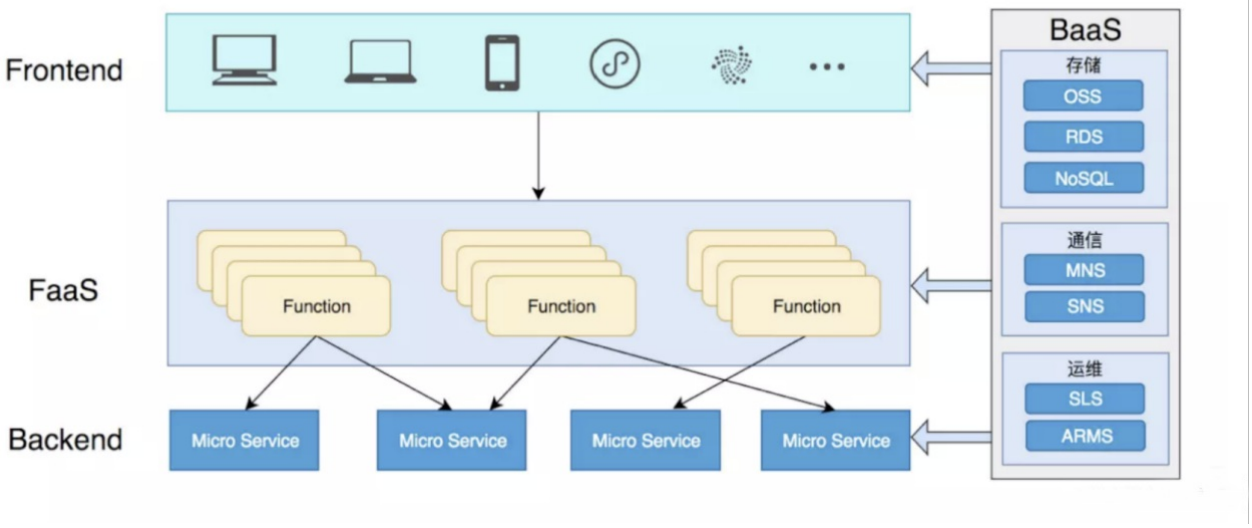
8.1 BFF问题

- 复杂性增加：添加 BFF 层会增加系统的复杂性，因为它需要在后端 API 和前端应用程序之间处理请求和响应
- 性能问题：如果 BFF 层的实现不当，可能会导致性能问题，因为它需要在后端 API 和前端应用程序之间传输大量数据
- 安全风险：如果 BFF 层未得到正确保护，可能会导致安全风险，因为它可能会暴露敏感数据
- 维护成本：BFF 层需要维护和更新，这会增加维护成本
- 测试复杂性：由于 BFF 层需要在后端 API 和前端应用程序之间进行测试，因此测试可能会变得更加复杂
- 运维问题 要求有强大的日志、服务器监控、性能监控、负载均衡、备份冗灾、监控报警和弹性伸缩扩容等

8.2 Serverless

- 这些问题可以通过Serverless (https://docs.cloudbase.net/)来解决
- Serverless = Faas (Function as a service) + Baas (Backend as a service)
- FaaS（Function-as-a-Service）是服务商提供一个平台、提供给用户开发、运行管理这些函数的功能，而无需搭建和维护基础框架，是一种事件驱动由消息触发的函数服务
- BaaS（Backend-as-a-Service）后端即服务，包含了后端服务组件，它是基于 API 的第三方服务，用于实现应用程序中的核心功能，包含常用的数据库、对象存储、消息队列、日志服务等等





8.3 Serverless的优势

- 节省成本：在传统架构中，你需要为应用程序所使用的服务器付费，即使它们没有被使用。在 Serverless 架构中，你仅需为实际使用的资源付费，这可以节省大量成本
- 更快的开发周期：Serverless 架构允许开发人员更快地构建和部署应用程序，因为它们可以更快地获得所需的资源
- 更好的可伸缩性：Serverless 架构可以自动扩展来满足增长的流量需求，无需人工干预
- 更好的可维护性：在 Serverless 架构中，你无需担心底层基础架构的维护，因为这些工作由云服务提供商负责
- 更高的可用性：由于 Serverless 架构具有自动扩展功能，因此它可以更好地应对突发流量，从而提高应用程序的可用性

8.4 Serverless的缺点

- 复杂性：Serverless 架构可能会使应用程序的体系结构变得更加复杂，因为它需要将应用程序拆分为许多小型函数
- 性能问题：在某些情况下，Serverless 架构可能会导致性能问题，因为函数执行需要额外的时间来启动和终止
- 限制：每个函数都有资源限制，因此需要仔细规划应用程序的体系结构，以免超出这些限制
- 依赖云服务提供商：使用 Serverless 架构需要依赖云服务提供商，因此如果这些服务出现故障，可能会对应用程序造成影响
- 调试困难：由于 Serverless 架构使用许多小型函数，因此调试可能会变得更加困难

9.GraphQL

- GraphQL是一种用于API的查询语言，它允许客户端向服务端请求特定的数据，而不是服务端将所有可能的数据全部返回。这样，客户端可以更精确地获取所需的数据，并且服务端可以更有效地满足请求
- GraphQL可以让客户端自己定义所需的数据结构，可以灵活地获取所需的数据。这对于多端应用来说非常方便，因为每一个客户端可能有不同的数据需求，使用GraphQL可以让每个客户端自己定义所需的数据结构
- GraphQL可以让BFF服务层从不同的数据源获取数据，并将它们组合起来返回给客户端。这对于在BFF架构中更好地组织数据是很有帮助的，因为你可以在BFF层中组合来自不同数据源的数据，而不用在客户端中再做一次组合

9.1 Apollo Server

- Apollo Server是一个用于构建GraphQL API的开源服务器框架。它支持多种编程语言，允许你使用同一种方式来访问不同的后端数据源，并且具有良好的扩展性
- Apollo Server是一种实现GraphQL服务端的方法，它提供了一些工具和功能，帮助你更轻松地构建和部署GraphQL API。它还提供了一些额外的功能，如缓存、身份验证和模拟数据，帮助你更快速地开发和测试你的API

9.2 GraphQL schema language

- schema (https://graphql.org/learn/schema/)
- GraphQL schema language是一种用来定义GraphQL API的语言。它可以用来描述API中可用的数据和操作，包括支持的查询、变更、订阅等功能
- GraphQL schema由一系列的类型组成，每种类型都有一个名称和一些字段。每个字段都有一个名称和类型，并可能有一些额外的限制，比如是否是必填的或者有默认值

9.3 resolvers

- resolvers (https://www.apollographql.com/docs/apollo-server/data/resolvers/)
- 在GraphQL中， resolvers是负责解析每个字段的函数。在Apollo Server中，你可以使用resolvers对象来定义这些函数
- resolvers对象是一个包含了所有解析器函数的对象。它的结构与你在schema中定义的类型的结构是一样的
- 除了定义解析器函数以外，你还可以在resolvers对象中定义自定义操作，例如查询、变更、订阅等。这些操作的解析器函数与字段的解析器函数的定义方式是一样的，只是函数名称不同而已

9.4 ApolloServer示例

- gql函数是一个 template tag，你可以将它放在模板字符串的前面，然后在模板字符串中编写GraphQL schema language的代码,可以定义查询和变更操作
- resolvers函数参数
    - obj：表示当前查询的父对象。例如，如果你在查询"user"类型的对象，那么obj就表示当前查询的"user"对象
    - args：表示当前查询的参数。例如，如果你在查询带有参数的字段，那么args就表示这些参数
    - context：表示当前的上下文对象，可以在整个查询中传递给所有的resolver
    - info：表示当前的查询信息，包括查询字符串、查询操作（query/mutation）、查询字段等

```
const { ApolloServer, gql } = require('apollo-server');
const typeDefs = gql`
  type Query {
    users: [User]
    user(id: ID): User
  }
  type Mutation {
    createUser(username: String, age: Int): User
    updateUser(id: ID, username: String, age: Int): Boolean
    deleteUser(id: ID): Boolean
  }
  type User {
    id: ID
    username: String
    age: Int
  }
`;
let users = [
    { id: "1", username: "zhangsan", age: 25 },
    { id: "2", username: "lisi", age: 30 },
];
const resolvers = {
    Query: {
        users: (obj, args, context, info) => {
            return users;
        },
        user: (obj, args, context, info) => {
            return users.find(user => user.id === args.id);
        }
    },
    Mutation: {
        createUser: (obj, args, context, info) => {
            const newUser = { id: users.length + 1, username: args.username, age: args.age };
            users.push(newUser);
            return newUser;
        },
        updateUser: (obj, args, context, info) => {
            const updatedUser = { id: args.id, username: args.username, age: args.age };
            users = users.map(user => {
                if (user.id === args.id) {
                    return updatedUser;
                }
                return user;
            });
            return true;
        },
        deleteUser: (obj, args, context, info) => {
            users = users.filter(user => user.id !== args.id);
            return true;
        },
    },
};
const server = new ApolloServer({ typeDefs, resolvers });
server.listen().then(({ url }) => {
    console.log(`Server ready at ${url}`);
});
```

```
query {
    users {
        id
        username
        age
    }
}
query {
    user(id: "1") {
        id
        username
        age
    }
}
mutation {
    createUser(username: "wangwu", age: 35) {
        id
        username
        age
    }
}
mutation {
    updateUser(id: "1", username: "zhangsan2", age: 26)
}
mutation {
    deleteUser(id: "1")
}
```

9.5 Apollo Server Koa

- Apollo Server Koa 是一个基于 Koa 的中间件，可以将 GraphQL API 添加到 Koa 应用程序中。它使用 Apollo Server 来执行 GraphQL 服务器逻辑，并允许使用 Koa 的优秀特性（如路由和中间件）来构建应用程序

```
const Koa = require('koa');
const { ApolloServer } = require('apollo-server-koa');

const typeDefs = `
  type Query {
    hello: String
  }
`;

const resolvers = {
    Query: {
        hello: () => 'Hello, world!',
    },
};

(async function () {
    const server = new ApolloServer({ typeDefs, resolvers });
    await server.start()
    const app = new Koa();
    server.applyMiddleware({ app });
    app.listen({ port: 4000 }, () =>
        console.log(`🚀 Server ready at http://localhost:4000${server.graphqlPath}`)
    );
})()
```