

Web 网盘项目

该项目总共分为 5 期，每一期我们都会逐步加入一些新的功能。

1. 第一期

实现 web 网盘项目的基本功能：注册，登录，获取用户信息，展示文件列表，上传文件，下载文件。

1.1 数据库表的设计

用户表

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	(Null)	auto_increment
username	varchar(255)	NO	UNI	(Null)	
password	varchar(255)	NO		(Null)	
salt	varchar(64)	NO		(Null)	
created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
tomb	int	YES		0	

用户表的创建语句：

```
1 CREATE TABLE `tbl_user` (  
2   `id` int NOT NULL AUTO_INCREMENT,  
3   `username` varchar(255) NOT NULL,  
4   `password` varchar(255) NOT NULL,  
5   `salt` varchar(64) NOT NULL,  
6   `created_at` datetime DEFAULT CURRENT_TIMESTAMP,  
7   `tomb` int DEFAULT '0',  
8   PRIMARY KEY (`id`),  
9   UNIQUE KEY `username` (`username`)  
10 ) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

文件表

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	(Null)	auto_increment
uid	int	NO	MUL	(Null)	
filename	varchar(255)	NO		(Null)	
hashcode	varchar(255)	NO	UNI	(Null)	
size	bigint	YES		0	
created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
last_update	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED on update CURRENT_TIMESTAMP
status	int	YES		0	

文件表的创建语句：

```

1 CREATE TABLE `tbl_file` (
2   `id` int NOT NULL AUTO_INCREMENT,
3   `uid` int NOT NULL,
4   `filename` varchar(255) NOT NULL,
5   `hashcode` varchar(255) NOT NULL,
6   `size` bigint DEFAULT '0',
7   `created_at` datetime DEFAULT CURRENT_TIMESTAMP,
8   `last_update` datetime DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
9   `status` int DEFAULT '0' COMMENT '状态(可用/禁用/已删除等)',
10  PRIMARY KEY (`id`)
11 ) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

```

1.2 接口设计

1.2.1 a. 注册

- POST /user/signup
- 以 x-www-form-urlencoded (表单) 方式上传用户名和密码
- 业务流程：提取用户名和密码，生成随机盐值，对密码进行哈希 (加盐)，将用户的信息存入数据库表
- 返回客户端
 - 成功：返回字符串“SUCCESS”
 - 失败：返回对应的出错原因

1.2.2 b. 登录

- POST /user/signin
- 以 x-www-form-urlencoded (表单) 方式上传用户名和密码
- 业务流程
 - 根据用户名查询用户信息
 - 校验密码：使用查询得到的盐值，对用户上传的密码进行哈希，比对生成的哈希和数据库中存入的哈希
 - 校验成功，生成 Token.
- 返回客户端
 - 成功：返回一个 JSON 字符串

```

1 {
2   "data": {
3     "Username": 用户名,
4     "Token": 用户的登录Token,
5     "Location": "/static/view/home.html"
6   }
7 }

```

失败：对应的出错原因

1.2.3 c. 获取用户信息

- GET /user/info
- 以 url 查询参数的方式上传用户名和 token

- 业务流程

校验 Token

根据用户名查询用户信息

- 返回客户端

成功：返回一个 JSON 字符串

```
1  {
2      "data": {
3          "Username": 用户名,
4          "SignupAt": 注册时间
5      }
6  }
```

失败：返回对应的出错原因

1.2.4 d. 文件列表查询

- POST /file/query
- 以 url 查询参数的方式上传用户名和 token
- 以 x-www-form-urlencoded (表单) 方式上传 limit (最多获取多少文件)

- 业务流程

校验 Token

根据用户名获取该用户的文件列表 (使用 limit 限制列表的最大长度)

- 返回客户端

成功：返回 JSON 字符串

```
1  [
2      {
3          "FileHash": 文件的哈希值,
4          "FileName": 文件的名字,
5          "FileSize": 文件的大小,
6          "UploadAt": 上传时间,
7          "LastUpdated": 最近修改时间
8      },
9      {
10         ...
11     }
12 ]
```

失败：返回对应的出错原因

1.2.5 e. 上传

- POST /file/upload
- 以 url 查询参数的方式上传用户名和 token
- 以 form-data 的方式上传文件
- 业务流程
 - 提取文件内容，生成哈希值
 - 保存文件
 - 将文件信息写入数据库
- 返回客户端
 - 成功：跳转到用户首页
 - 失败：返回对应的错误信息

1.2.6 f. 下载

- GET /file/download
- 以 url 查询参数的方式上传文件名，文件哈希，用户名和 token
- 业务流程
 - 校验 token
 - 返回文件
- 返回客户端
 - 成功：返回文件内容
 - 失败：返回对应的错误信息

2. 第二期

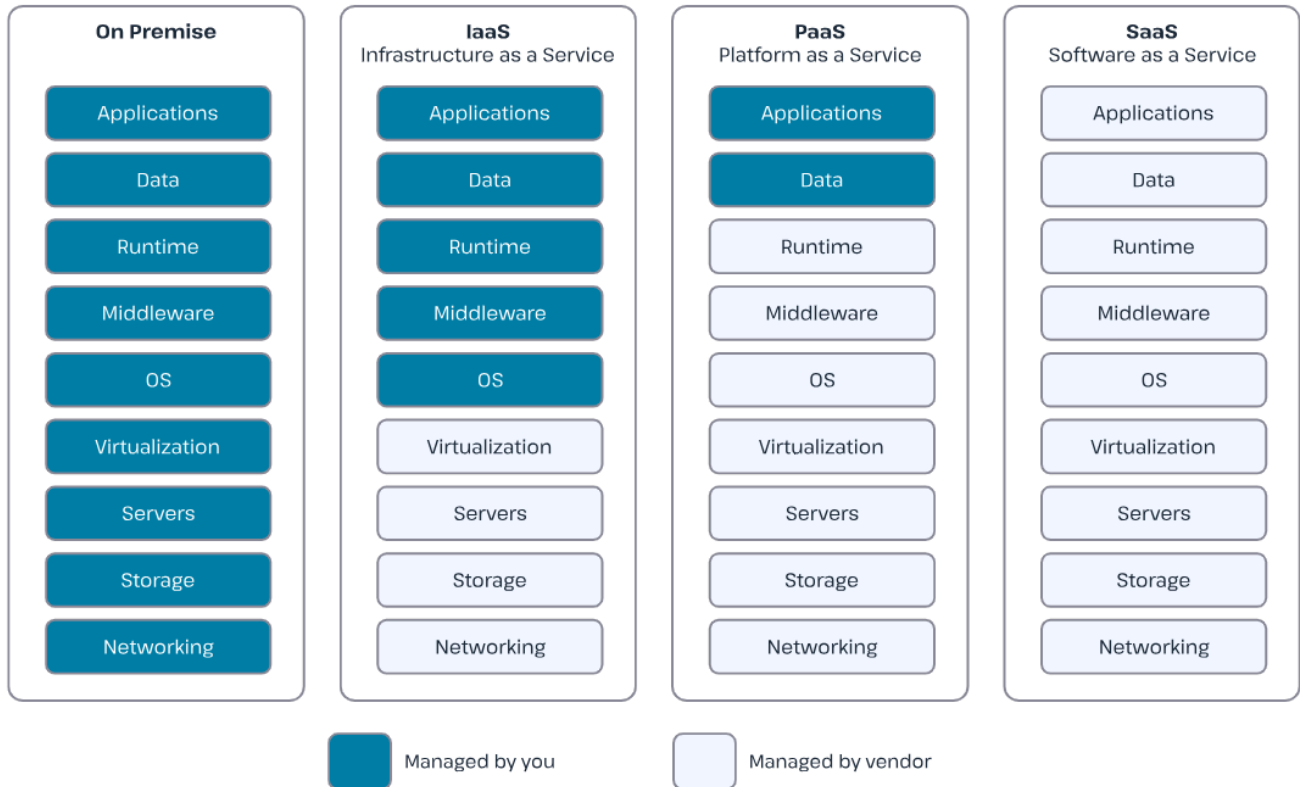
2.1 云服务

在实际的网盘存储当中，为了避免服务器崩溃导致数据丢失，一般可以采用备份的方案来解决（比如数据库的主从复制之类的）。遗憾的是，用户自己手动实现备份的方案基本上不太可能的，因为无论是备份，还是同步或者错误恢复等功能都不太容易实现（其实远不止不太容易，而是非常困难）。这种情况下，用户会更加倾向于使用现有的成熟的存储产品。

而随着云计算的不断发展，IaaS(Infrastructure as a Service，基础设施即服务)的概念逐渐深入人心。所谓 IaaS，就是云服务器厂商把文件系统、数据库、缓存系统、消息队列等等组件剥离出来做成一个单独的产品，用户通过网络调用接口来使用服务，而不需要自行部署文件系统和数据库等组件。这样，数据备份的需求可以交给云计算厂商的文件系统产品，通常会采用两种方法来使用：一种是使用专业的云厂商的提供的公有云产品，比如阿里云对象存储 OSS；另一种是自己购买硬件，在此基础上搭建私有云产品，比如自行搭建分布式文件系统 ceph。

- IaaS (Infrastructure as a Service，基础设施即服务)：在 IaaS 模型中，云服务提供商提供基础的计算资源，包括虚拟机、存储、网络等，用户可以通过网络访问和管理这些资源。用户可以根据需要灵活地配置和管理这些资源，而无需购买和维护自己的物理硬件设备。典型的 IaaS 提供商包括 Amazon Web Services (AWS) 的 EC2、Microsoft Azure 的 Virtual Machines 和 Google Cloud Platform 的 Compute Engine 等。
- PaaS (Platform as a Service，平台即服务)：在 PaaS 模型中，云服务提供商提供完整的应用开发和部署平台，包括运行时环境、开发工具、数据库和中间件等。开发者可以使用 PaaS 平台来开发、测试、部署和管理应用程序，而无需关心底层的基础设施。典型的 PaaS 提供商包括 Heroku、Google App Engine、Microsoft Azure App Service 等。

- SaaS (Software as a Service, 软件即服务)：在 SaaS 模型中，云服务提供商以服务的形式提供完整的软件应用程序，用户通过互联网访问和使用这些应用程序。用户无需购买、安装或维护软件，只需通过浏览器或应用程序接口即可使用应用程序的功能。典型的 SaaS 应用包括 Google Workspace、Salesforce、Microsoft 365 等。



在使用了云产品了之后，基本上数据丢失的问题就很难发生了，因为云厂商往往能够提供很高的可靠性和强大的备份恢复系统，也减少了会运维的工作量。

2.2 阿里云 OSS 存储

下载安装 [OSS C++ SDK](#)

```
1 # 安装依赖库
2 $ sudo apt install libssl-dev
3 $ sudo apt install libcurl4-openssl-dev
4
5 # 解压缩 C++ SDK 安装包
6 $ tar xzvf aliyun-oss-cpp-sdk-1.10.0.tar.gz
7 $ cd aliyun-oss-cpp-sdk-1.10.0
8
9 # 安装 C++ SDK
10 $ mkdir build
11 $ cd build
12 $ cmake ..
13 $ make
14 $ sudo make install
15 $ sudo ldconfig
```

2.2.1 OSS 的基本组件

想要顺利使用 OSS，必须先知道下面几个组件的基本概念：

- 存储空间（Bucket） 存储空间是您用于存储对象（Object）的容器，所有的对象都必须隶属于某个存储空间。存储空间具有各种配置属性，包括地域、访问权限、存储类型等。存储空间可以认为是一个“很大”的文件夹。
- 对象（Object） 对象是 OSS 存储数据的基本单元，也被称为 OSS 的文件。对象由元信息（Object Meta）、用户数据（Data）和文件名（Key）组成。对象由存储空间内部唯一的 Key 来标识。对象元信息是一组键值对，表示了对象的一些属性，例如最后修改时间、大小等信息，同时也可以元信息中存储一些自定义的信息。
- 地域（Region） 地域表示 OSS 的数据中心所在物理位置。
- 访问域名（Endpoint） Endpoint 表示 OSS 对外服务的访问域名。OSS 以 HTTP RESTful API 的形式对外提供服务，当访问不同地域的时候，需要不同的域名。通过内网和外网访问同一个地域所需要的域名也是不同的。
- 访问密钥（AccessKey） AccessKey 简称 AK，指的是访问身份验证中用到的 AccessKey ID 和 AccessKey Secret。OSS 通过使用 AccessKey ID 和 AccessKey Secret 对称加密的方法来验证某个请求的发送者身份。AccessKey ID 用于标识用户；AccessKey Secret 是用户用于加密签名字符串和 OSS 用来验证签名字符串的密钥，必须保密。

2.2.2 示例

1. 上传本地文件

```
1  #include <alibabacloud/oss/OssClient.h>
2
3  using namespace AlibabaCloud::OSS;
4
5  int main(void)
6  {
7      /* 初始化网络等资源。*/
8      InitializeSdk();
9
10     // 1. 初始化OSS账号信息
11     std::string endpoint = "oss-cn-wuhan-1r.aliyuncs.com";
12     std::string accessKeyId = "LTAI5tHQTUfRRD7DTnRCC9DZ";
13     std::string accessKeySecret = "BCm5w87LsnCHBL0w6MXwZAnJYY0XoN";
14     std::string region = "cn-wuhan";
15
16     ClientConfiguration conf;
17     OssClient client { endpoint, accessKeyId, accessKeySecret, conf };
18     client.SetRegion(region);
19
20     // 2. 上传文件
21     std::string bucketName = "peanutixx-oss-demo";
22     std::string objectName = "dir/a.txt";
23     auto outcome = client.PutObject(bucketName, objectName, "a.txt");
24
25     if (!outcome.IsSuccess()) {
26         /* 异常处理。*/
27         std::cout << "PutObject fail" <<
28         ", code:" << outcome.error().Code() <<
29         ", message:" << outcome.error().Message() <<
```

```

30         ", requestId:" << outcome.error().RequestId() << std::endl;
31         return -1;
32     }
33
34     /* 释放网络等资源。*/
35     ShutdownSdk();
36     return 0;
37 }

```

2. 从内存中上传文件

```

1  #include <alibabacloud/oss/OssClient.h>
2
3  using namespace AlibabaCloud::OSS;
4
5  int main(void)
6  {
7      /* 初始化网络等资源。*/
8      InitializeSdk();
9
10     // 1. 初始化OSS账号信息
11     std::string endpoint = "oss-cn-wuhan-1r.aliyuncs.com";
12     std::string accessKeyId = "LTAI5tHQTUfRRD7DTnRCC9DZ";
13     std::string accessKeySecret = "BCm5w87LsnCHBL0w6MXwZAnJYY0XoN";
14     std::string region = "cn-wuhan";
15
16     ClientConfiguration conf;
17     OssClient client {endpoint, accessKeyId, accessKeySecret, conf};
18     client.SetRegion(region);
19
20     // 2. 上传文件
21     std::string bucketName = "peanutixx-demo";
22     std::string objectName = "dir/b.txt";
23
24     std::string content = "Hello OSS";
25     std::shared_ptr<std::iostream> stream = std::make_shared<std::stringstream>
26     (std::move(content));
27
28     PutObjectRequest request { bucketName, objectName, stream };
29
30     auto outcome = client.PutObject(request);
31
32     if (!outcome.isSuccess()) {
33         /* 异常处理。*/
34         std::cout << "PutObject fail" <<
35         ",code:" << outcome.error().Code() <<
36         ",message:" << outcome.error().Message() <<
37         ",requestId:" << outcome.error().RequestId() << std::endl;
38         return -1;
39     }
40
41     /* 释放网络等资源。*/
42     ShutdownSdk();

```

```
42 |         return 0;  
43 |     }
```

3. 第三期

同步备份文件会导致时延边长，用户体验不好，我们可以通过消息队列来实现异步备份。

3.1 3.1 什么是消息队列

消息：是两个应用程序之间传递的数据。

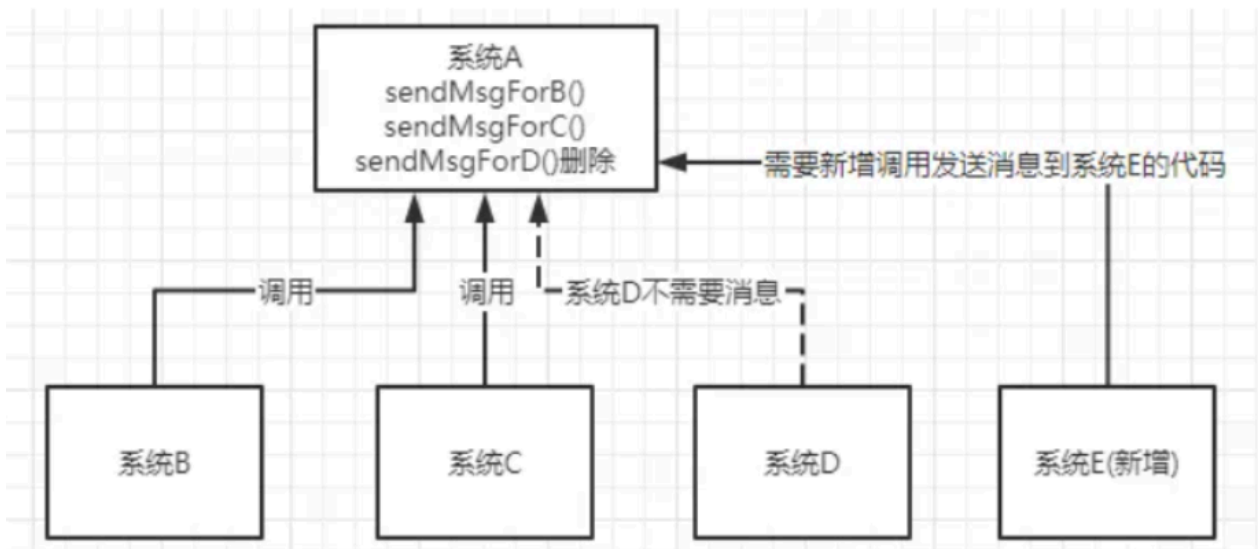
消息队列 (Message Queue)：是消息的传输过程中保存消息的容器。在消息队列中，通常有生产者和消费者两个角色。生产者只负责发送数据到消息队列，谁从消息队列中取出数据处理，他不管。消费者只负责从消息队列中取出数据处理，他不管这是谁发送的数据。



3.2 3.2 为什么要用消息队列?

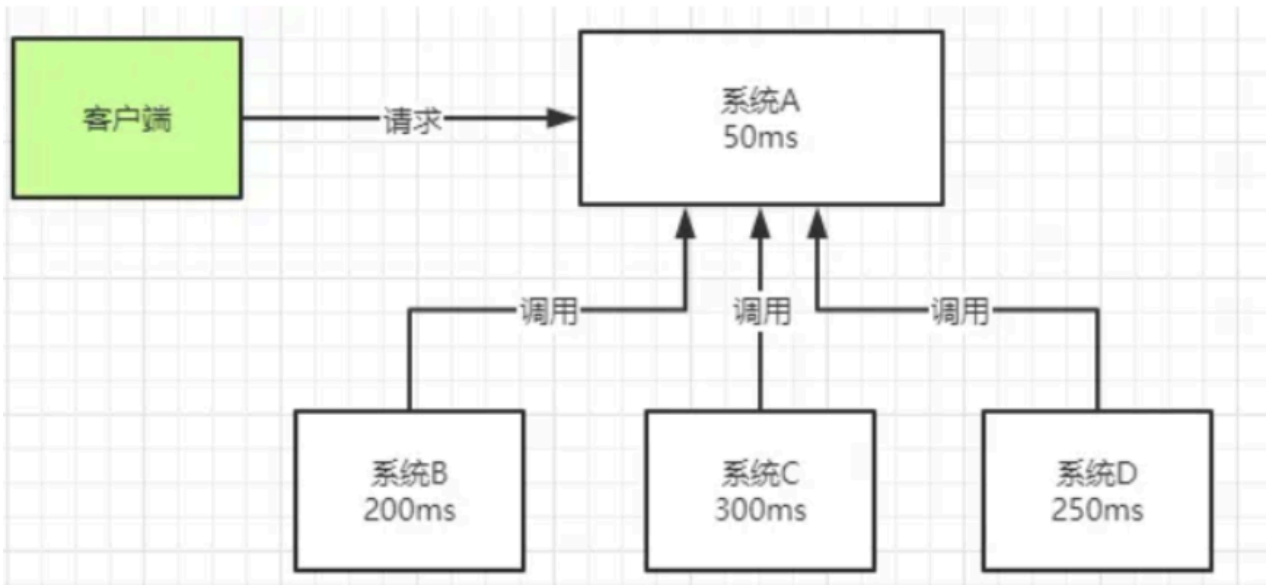
主要有三个原因：

- **解耦。**如图所示。假设有系统 B、C、D 都需要系统 A 的数据，于是系统 A 调用三个方法发送数据到 B、C、D。这时，系统 D 不需要了，那就需要在系统 A 把相关的代码删掉。假设这时有个新的系统 E 需要数据，这时系统 A 又要增加调用系统 E 的代码。为了降低这种强耦合，就可以使用 MQ，系统 A 只需要把数据发送到 MQ，其他系统如果需要数据，则从 MQ 中获取即可。

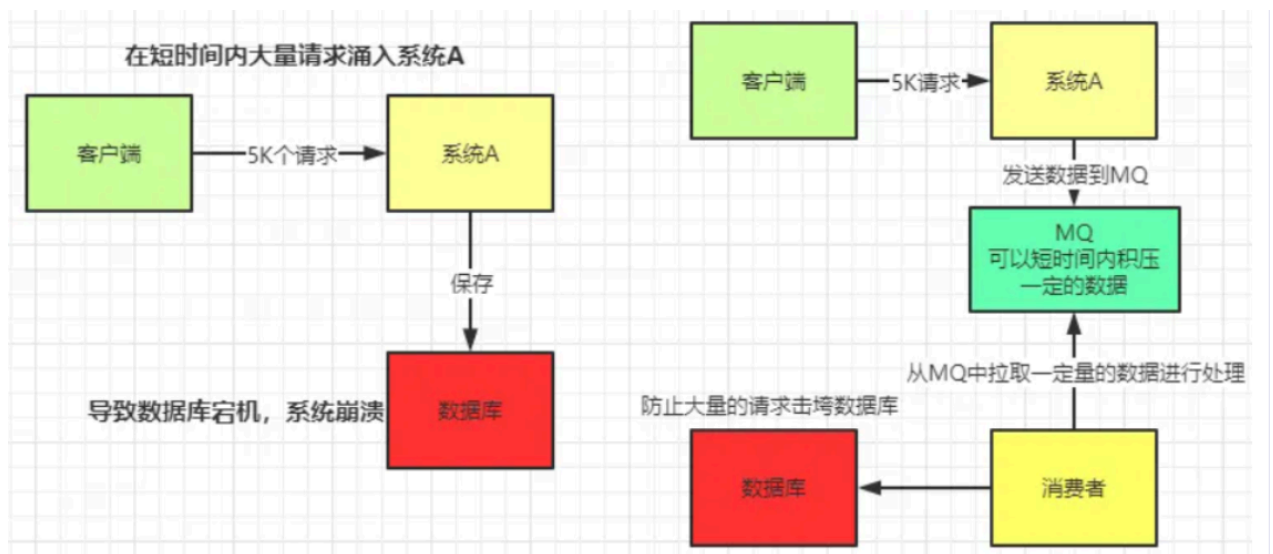


- **异步。**如图所示。一个客户端请求发送进来，系统 A 会调用系统 B、C、D 三个系统，同步请求的话，响应时间就是系统 A、B、C、D 的总和，也就是 800ms。如果使用 MQ，系统 A 发送数据到 MQ，然后就可以返回响应给客户端，不需要再等待系统 B、C、D 的响应，可以大大地提高性能。对于一些非必要的业务，比如发

送短信，发送邮件等等，就可以采用 MQ。



- **削峰填谷。**如图所示。这其实是 MQ 一个很重要的应用。假设系统 A 在某一段时间请求数暴增，有 5000 个请求发送过来，系统 A 这时就会发送 5000 条 SQL 进入 MySQL 进行执行，MySQL 对于如此庞大的请求当然处理不过来，MySQL 就会崩溃，导致系统瘫痪。**如果使用 MQ，系统 A 不再是直接发送 SQL 到数据库，而是把数据发送到 MQ，MQ 短时间积压数据是可以接受的，然后由消费者每次拉取 2000 条进行处理，防止在请求峰值时期大量的请求直接发送到 MySQL 导致系统崩溃。**



3.3 消息队列缺点

- 可用性降低：系统外部依赖变多，容易崩溃
- 复杂性提高：需要考虑消息的一致性问题、可靠传输问题、不被重复消费等问题。

3.4 使用 Docker 安装 RabbitMQ

```
1 docker pull rabbitmq:management
2
3 docker run -d --hostname rabbitsrv --name rabbit -p 5672:5672 -p 15672:15672 -p 25672:25672 -v /data/rabbitmq:/var/lib/rabbitmq rabbitmq:management
```

3.5 3.5 RabbitMQ 简介

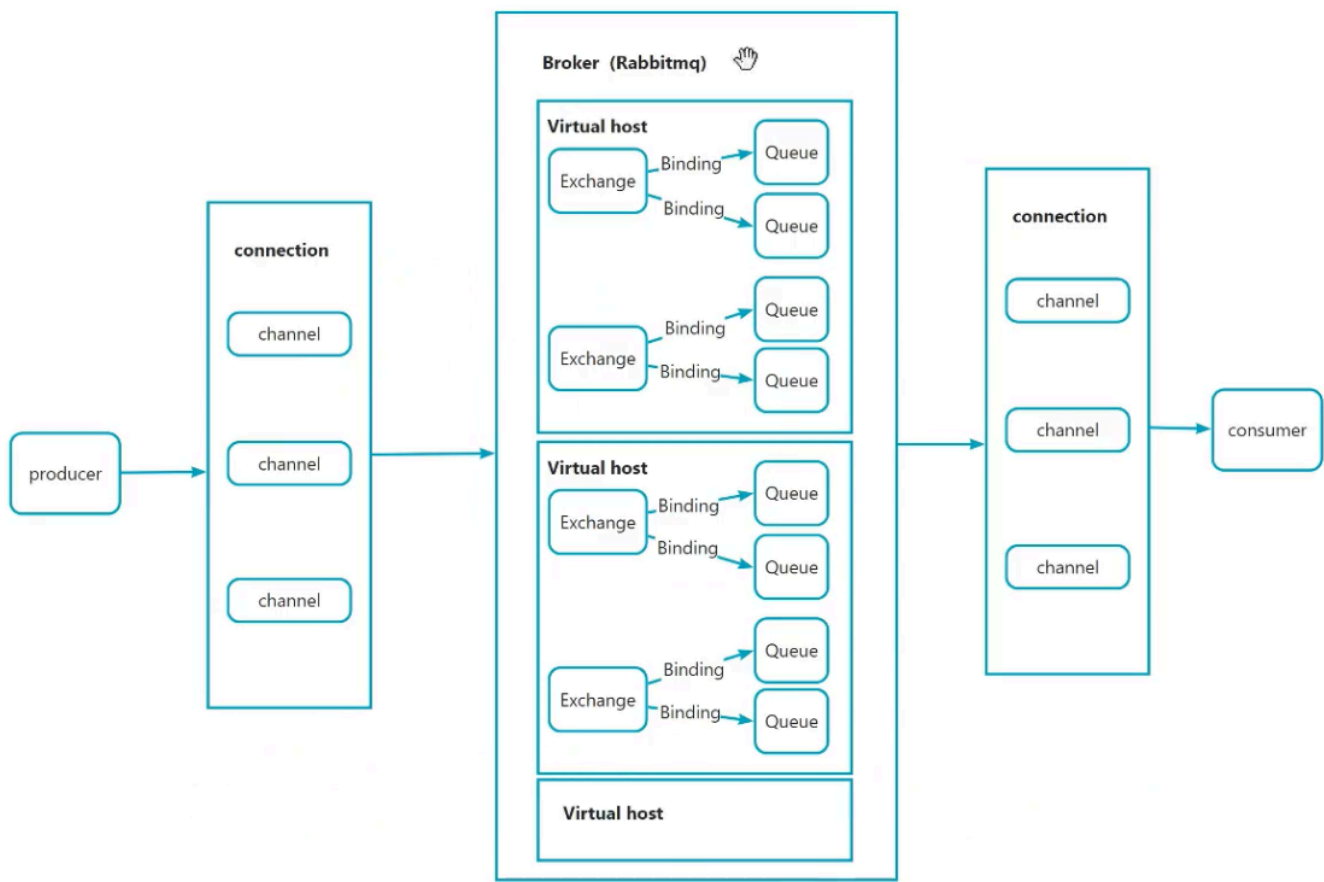
RabbitMQ 是一套开源的消息队列服务，基于 AMQP 的开源实现，由 Erlang 写成。

AMQP：Advanced Message Queue，[高级消息队列协议](#)。它是应用层协议的一个开放标准，为面向消息的中间件设计，基于此协议的客户端与 [消息中间件](#) 可传递消息，并不受产品、开发语言等条件的限制。

通常消息队列服务有三个概念：发消息者、消息队列、收消息者。

RabbitMQ 在这个基本概念之上，多做了一层，在发消息者和队列之间，加入了交换机 (Exchange)。这样发消息者和消息队列就没有直接联系，转而变成发消息者把消息发给交换器，交换器根据调度策略再把消息转发给消息队列。

3.6 3.6 RabbitMQ 架构



3.7 3.7 Exchange 类型

Exchange 有四种类型，分别是

3.7.1 Direct (直连)

和 key 完全匹配

3.7.2 Fanout (扇出)

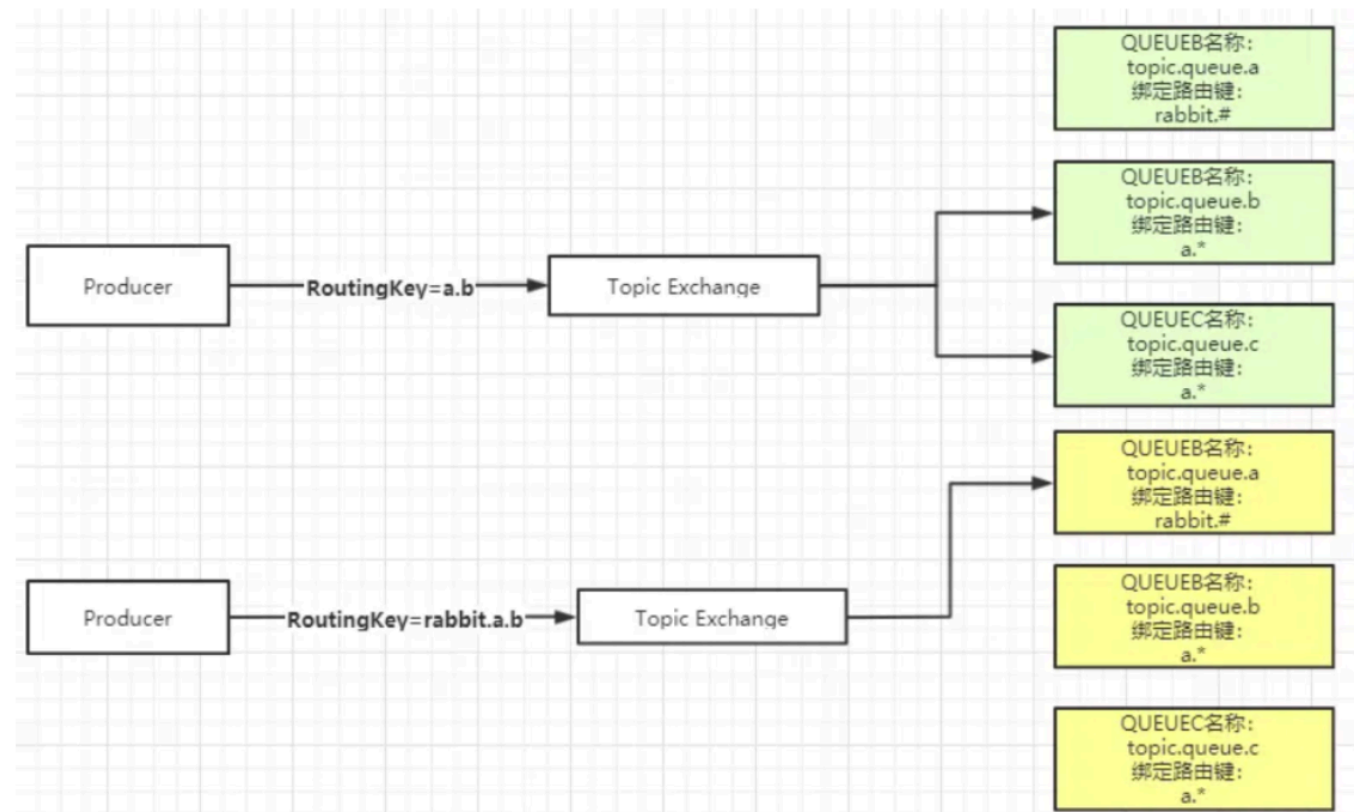
不需要绑定 key，类似广播

3.7.3 Topic

*: 匹配一个字段

#: 匹配任意多个字段 (包括 0 个)

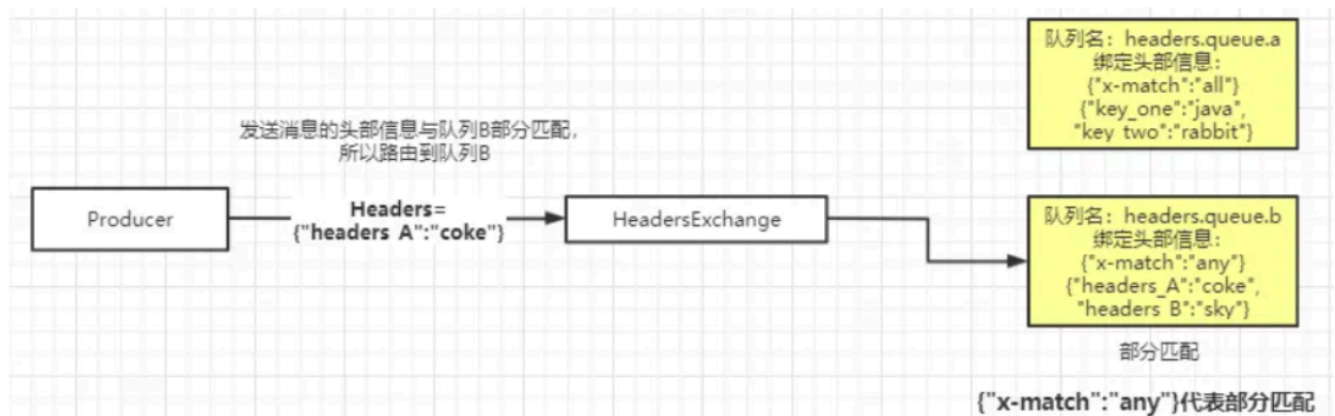
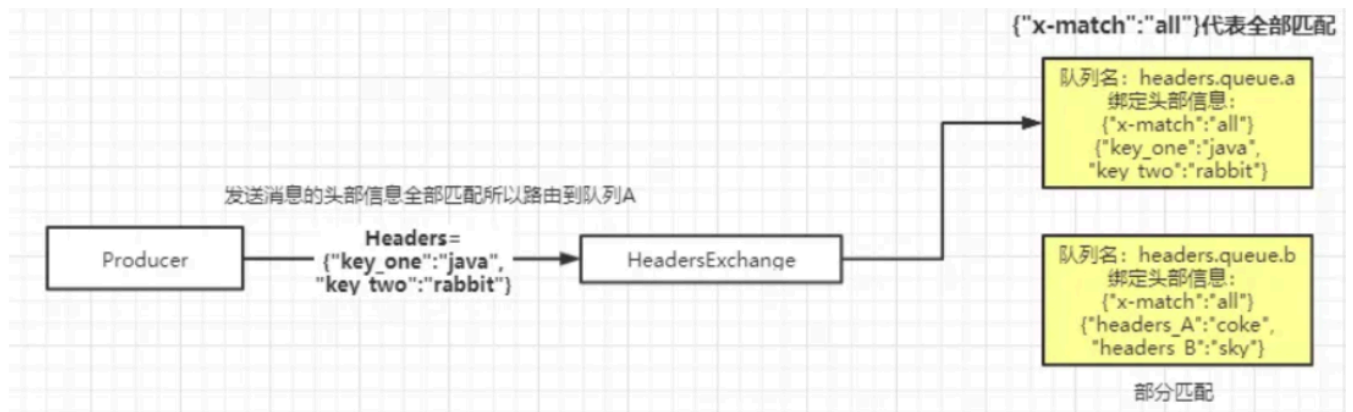
字段之间以 . 分隔



3.7.4 Headers (了解)

部分匹配: 只要匹配一个 Header 即可

完全匹配: 需要匹配所有的 Header



3.8 3.8 SimpleAmqpClient

3.8.1 安装

```

1 # SimpleAmqpClient 依赖 boost, rabbitmq-c, cmake
2 sudo apt install libboost-dev          # 安装boost
3 sudo apt install libboost-chrono-dev
4 sudo apt install libboost-system-dev
5
6 # 使用 cmake 安装 rabbitmq-c
7 # 使用 cmake 安装 SimpleAmqpClient

```

生成 SimpleAmqpClient 的 API 文档 (选做)

```

1 # 安装 Doxygen
2 sudo apt-get install doxygen
3 # 安装 graphviz (用于生成图表)
4 sudo apt-get install graphviz
5 # 生成 API 文档 (需要 Doxyfile 文件)
6 doxygen Doxyfile

```

3.8.2 示例

生产者:

```

1 // producer.cc
2 #include <SimpleAmqpClient/SimpleAmqpClient.h>
3 #include <string>

```

```

4
5 int main()
6 {
7     using std::string;
8
9     string host = "127.0.0.1";
10    int port = 5672;    // AMQP协议
11    string username = "guest";
12    string password = "guest";
13    string vhost = "/";
14
15    // 创建Channel
16    AmqpClient::Channel::ptr_t channel = AmqpClient::Channel::Create(host, port,
    username, password, vhost);
17    // 构建消息
18    AmqpClient::BasicMessage::ptr_t message = AmqpClient::BasicMessage::Create("Hello
    RabbitMQ1");
19    // 发布消息
20    string exchange = "my.direct"; // 指定交换机
21    string routingKey = "key1";
22    channel->BasicPublish(exchange, routingKey, message);
23 }

```

消费者:

```

1 // consumer.cc
2 #include <iostream>
3 #include <string>
4 #include <SimpleAmqpClient/SimpleAmqpClient.h>
5
6 int main()
7 {
8     using std::string;
9     using namespace AmqpClient;
10
11    string uri = "amqp://guest:guest@localhost:5672/";
12    AmqpClient::Channel::ptr_t channel = AmqpClient::Channel::CreateFromUri(uri);
13
14    AmqpClient::Envelope::ptr_t envelope;
15    const std::string& q = "direct.queue.1";
16
17    // 如果队列中有消息，将消息放入到envelope中，并返回true
18    // 如果队列中没有消息，BasicGet会立刻返回false
19    // channel->BasicGet(envelope, q);
20
21    /* if (envelope && envelope->Message()) { */
22    /*     std::cout << envelope->Message()->Body() << "\n"; */
23    /* } */
24
25    channel->BasicConsume(q);
26
27    for(;;) {
28        Envelope::ptr_t envelope = channel->BasicConsumeMessage();

```

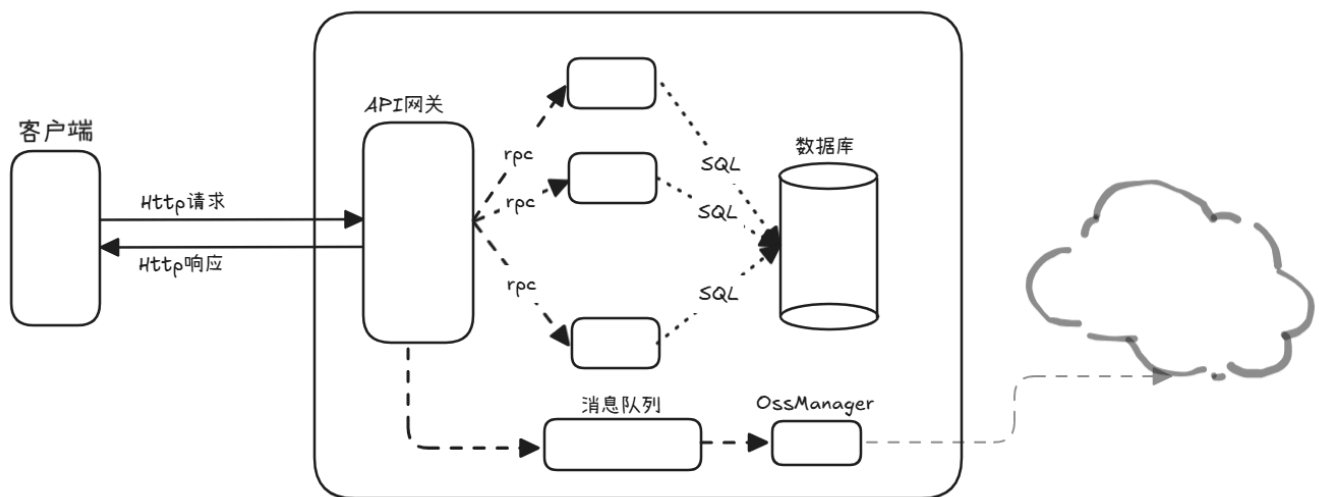
```

29         // 打印消息
30         if (envelope && envelope->Message()) {
31             std::cout << envelope->Message()->Body() << "\n";
32         }
33     }
34 }

```

4. 第四期

在这一期中，我们需要将前面的单体应用改造成微服务应用，如下图所示：



4.1 4.1 Protobuf

4.1.1 简介

Protobuf (Google Protocol Buffer) 是一种轻便高效的数据交换格式，它独立于语言 and 平台，并且支持扩展。Protobuf 很适合做数据存储和 RPC 的数据交换格式。使用 Protobuf 内置的编译器 `protoc` 可以自动生成 C++, C#, Dart, Go, Java, Kotlin, Objective-C, Python, Ruby, PHP 等多种语言的代码。

4.1.2 安装

```

1  $ sudo apt-get install automake autoconf libtool
2  $ tar xzvf protobuf-3.20.1.tar.gz
3  $ cd protobuf-3.20.1
4  $ ./autogen.sh
5  $ ./configure
6  $ make -j4
7  $ sudo make install
8  $ sudo ldconfig
9  $ protoc --version # 查看版本号

```

4.1.3 使用教程

对 Protobuf 有了一定的基本了解之后，接下来，我们看看该如何使用 Protobuf。

第一步：创建 .proto 文件，定义数据结构，如下所示：

```
1  syntax="proto3";
2  package test;
3
4  message Person {
5      int32 id = 1;
6      string name = 2;
7      optional string email = 3;
8  }
```

我们在上例中定义了一个名为 Person 的消息，语法很简单，message 关键字后跟上消息的名称。之后我们在其中定义了 message 具有的字段，形式为：

```
1  message xxx {
2      字段规则 类型 名称 = 字段编号;
3      ...
4  }
5  // 字段规则:
6  //   required -> 字段只能也必须出现 1 次（默认）
7  //   optional -> 字段可出现 0 次或1次
8  //   repeated -> 字段可出现任意多次（包括 0）
9  // 类型: int32、int64、sint32、sint64、string、32-bit ...
```

第二步：使用 protoc 编译 .proto 文件

```
1  $ protoc --cpp_out=./ Person.proto
2  # Usage: protoc [OPTION] PROTO_FILES
3  #   OPTIONS:
4  #       --cpp_out=OUT_DIR    指定在哪个目录下生成C++的头文件和源文件
```

生成的头文件和源文件分别为：Person.pb.h 和 Person.pb.cc。这些文件中定义和实现了 Person 消息的各个方法，其中就有各个字段的 getter 和 setter 方法：

```
1  namespace test {
2
3  class Person final :
4      ...
5      int32_t id() const;
6      void set_id(int32_t value);
7      ...
8      const std::string& name() const;
9      template <typename ArgT0, typename... ArgT>
10     void set_name(ArgT0&& arg0, ArgT... args);
11     ...
12     bool has_email() const;
13     const std::string& email() const;
```

```

14     template<typename ArgT0, typename... ArgT>
15     void set_email(ArgT0&& arg0, ArgT... args);
16     ...
17 };
18
19 }; // end of namespace `test`

```

第三步：编写业务代码

这里我们演示一下如何使用 Protobuf 进行序列化和反序列化：

```

1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include "Person.pb.h"
5
6  using namespace std;
7  using namespace test;
8
9  int main()
10 {
11     Person p1;
12     p1.set_name("test");
13     p1.set_id(100);
14     p1.set_email("example@gmail.com");
15
16     // 序列化：将C++中的结构体保存到二进制序列中
17     std::string output;
18     p1.SerializeToString(&output); // 序列化
19     cout << "size: " << output.size() << endl;
20     cout << "output: " << output << endl;
21
22     // 反序列化
23     Person p2;
24     cout << "p2.name: " << p2.name()
25          << ", p2.id: " << p2.id()
26          << ", p2.email: " << p2.email() << endl;
27
28     p2.ParseFromString(output); // 反序列化
29     cout << "p2.name: " << p2.name()
30          << ", p2.id: " << p2.id()
31          << ", p2.email: " << p2.email() << endl;
32 }

```

4.1.4 Protobuf 编码 (了解)

我们通过一个例子，来看看 Protobuf 是如何编码的，.proto 文件还是采用上面的 Person.proto。

```

1  // main.cc
2  #include <iostream>
3  #include <bitset>
4  #include "Person.pb.h"

```



```
5
6 using namespace std;
7 using namespace test;
8
9 int main()
10 {
11     Person p;
12     p.set_id(100);
13     p.set_name("aaa");
14
15     string data;
16     p.SerializeToString(&data); // 序列化
17
18     cout << "size: " << data.size() << endl;
19
20     for (auto c : data) {
21         cout << bitset<8>(c) << " ";
22     }
23     cout << endl;
24 }
```

运行结果如下：

```
1 size: 7
2 00001000 01100100 00010010 00000011 01100001 01100001 01100001
```

解析：

```
1 00001:      序号：1
2 000:        数据编码格式(wireType)：Varint
3 01100100:   值：100
4 00010:      序号：2
5 010:        数据编码格式(wireType)：Length-delimited
6 00000011:   长度：3
7 01100001:   值：97('a')
8 01100001:   值：97('a')
9 01100001:   值：97('a')
```

Protobuf 的数据编码格式有 6 种，其中两种已标记为废弃：

Wire Type (十进制)	名称	描述
0	Varint	可变长度整数 (int32, int64, uint32, uint64, sint32, sint64, bool, enum)
1	64-bit	固定 64 位 (fixed64, sfixed64, double)
2	Length-delimited	长度前缀字段 (string, bytes, 嵌套 message, packed repeated fields)

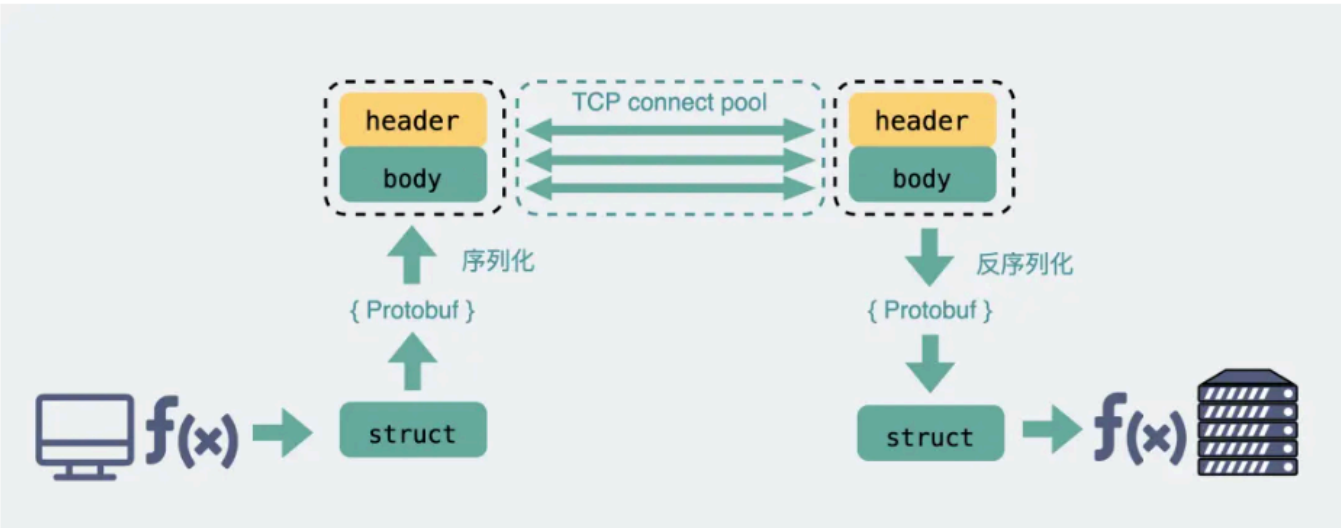
Wire Type (十进制)	名称	描述
3	Start group	group 的开始 (已废弃)
4	End group	group 的结束 (已废弃)
5	32-bit	固定 32 位 (fixed32, sfixed32, float)

4.2 4.2 RPC

4.2.1 简介

RPC (Remote Procedure Call) 是一种计算机通信协议，它允许一个程序 (客户端) 像调用本地方法一样调用另一个地址空间 (通常是远程服务器上) 的程序或服务，而无需显式编码远程调用的细节。

4.2.2 原理



4.3 4.3 SRPC (Sogou RPC)

4.3.1 简介

[srpc](#) 是全搜狗业务线上使用的企业级 RPC 系统，目前每天承载上百亿的请求量，涵盖搜广推及其他类型业务。

底层基于 [Sogou C++ Workflow](#)，是高性能、低延迟、轻量级 RPC 系统的极佳选择。且加入了 AOP 面向切面的模块，支持 Metrics (监控指标) 和 Trace (链路追踪) 功能，可上报到多种云原生系统，包括 OpenTelemetry。

4.3.2 安装

源码安装需要一些前置依赖：CMake (要求 3.6 以上)、OpenSSL (推荐 1.1 及以上)、Protobuf (要求 3.5 及以上)

默认会编译出：

1. 静态库：libsrpc.a
2. 动态库：libsrpc.so
3. 用于生成代码的二进制工具：srpc_generator

```

1 | $ sudo apt install liblz4-dev
2 | $ sudo apt install libsnappy-dev
3 | $ tar xvfz srpc-0.10.2.tar.gz
4 | $ cd srpc-0.10.2
5 | $ mkdir build
6 | $ cd build
7 | $ cmake ..
8 | $ make
9 | $ sudo make install
10 | $ sudo ldconfig

```

4.3.3 示例

安装好 srpc 之后，接下来我们来看看如何使用 srpc：

第一步：编写 IDL 文件

```

1 | // example.proto
2 | syntax = "proto3"; // proto2和proto3都可以，srpc都支持
3 |
4 | message EchoRequest {
5 |     string message = 1;
6 |     string name = 2;
7 | };
8 |
9 | message EchoResponse {
10 |     string message = 1;
11 | }
12 |
13 | // 定义一个名字为 Example 的服务
14 | service Example {
15 |     rpc Echo(EchoRequest) returns (EchoResponse); // 定义一个rpc
16 | }

```

第二步：使用 protoc 和 srpc_generator 生成对应的文件

```

1 | $ protoc --cpp_out=./ example.proto

```

默认会生成两个文件，分别为：example.pb.h 和 example.pb.cc，里面包含 IDL 文件中 `message` 内容的定义和实现。

```

1 | $ srpc_generator protobuf example.proto ./
2 | # Usage:
3 | #   srpc_generator [protobuf|thrift] <idl_file> <output_dir>

```

默认情况下，会生成三个文件，分别为：client.pb_skeleton.cc，server.pb_skeleton.cc，example.srpc.h。其中 client.pb_skeleton.cc 和 server.pb_skeleton.cc 分别为客户端和服务端的骨架代码（只提供骨架，还需要程序员修改）。example.srpc.h 包含了 IDL 文件中 `service` 内容的定义：

```

1 | ...

```

```

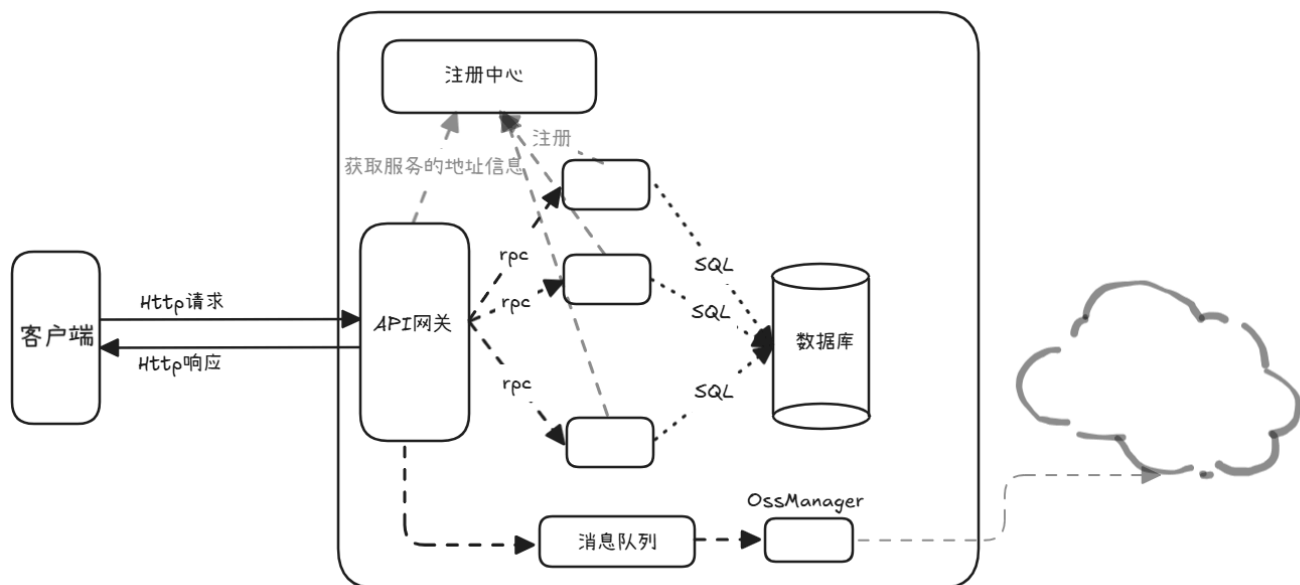
2 namespace Example
3 {
4
5 class Service : public srpc::RPCService
6 {
7 public:
8     // 纯虚函数：需要派生类重写
9     virtual void Echo(EchoRequest *request, EchoResponse *response,
10                      srpc::RPCContext *ctx) = 0;
11 public:
12     Service();
13 };
14
15 using EchoDone = std::function<void (EchoResponse *, srpc::RPCContext *)>;
16
17 class SRPCClient : public srpc::SRPCClient
18 {
19 public:
20     void Echo(const EchoRequest *req, EchoDone done); // 异步非阻塞
21     void Echo(const EchoRequest *req, EchoResponse *resp, srpc::RPCSyncContext
22 *sync_ctx); // 同步阻塞
23     WFFuture<std::pair<EchoResponse, srpc::RPCSyncContext>> async_Echo(const
24 EchoRequest *req);
25
26 public:
27     SRPCClient(const char *host, unsigned short port);
28     SRPCClient(const struct srpc::RPCCClientParams *params);
29
30 public:
31     srpc::SRPCClientTask *create_Echo_task(EchoDone done); // 与 workflow 集成
32 };
33 } // end of namespace `Example`

```

第三步：修改客户端和服务端的骨架代码，实现具体的业务逻辑

5. 第五期

在上一期中，我们将登录注册的逻辑从 API 网关中分离了；但是 API 网关访问登录注册服务的时候，IP 地址和端口是写死的，API 网关和后台服务之间还是耦合的。万一后台服务宕机了，API 也就不能正常工作了。为此，我们引入了注册中心。



5.1 5.1 注册中心

注册中心是微服务架构中的 **核心组件之一**，它的主要作用是管理和维护各个微服务的注册信息，实现服务之间的 **自动发现与通信**。注册中心 (Service Registry) 是一个 **用于保存服务实例信息的数据库或服务**。每个微服务在启动时，会将自己的地址 (如 IP 地址和端口) 和相关元信息注册到注册中心；其他服务可以通过注册中心找到它需要调用的服务。

5.1.1 Docker 安装 Consul

```

1  # 启动第一个节点
2  docker run --name consul1 -d -p 8500:8500 -p 8301:8301 -p 8302:8302 -p 8600:8600 \
3  hashicorp/consul agent -server -bootstrap-expect 2 -ui -bind=0.0.0.0 -client=0.0.0.0
4
5  # 查看第一个节点的IP地址，本例子中是：172.17.0.3
6  docker inspect consul1
7
8  # 加入第二个节点，注意：-join 后面的 ip 地址应与上面查询的结果一致
9  docker run --name consul2 -d -p 8501:8500 hashicorp/consul agent -server -ui \
10 -bind=0.0.0.0 -client=0.0.0.0 -join 172.17.0.3
11
12 # 加入第二个节点，注意：-join 后面的 ip 地址应与上面查询的结果一致
13 docker run --name consul3 -d -p 8502:8500 hashicorp/consul agent -server -ui \
14 -bind=0.0.0.0 -client=0.0.0.0 -join 172.17.0.3

```

5.1.2 安装 ppconsul

ppconsul 是一个典型的 CMake 工程，依照其它 CMake 的安装流程即可：

```

1  $ mkdir build
2  $ cd build
3  $ cmake ..
4  $ make
5  $ sudo make install
6  $ sudo ldconfig

```

5.1.3 示例

```
1  #include <wfrest/HttpServer.h>
2  #include <iostream>
3  #include <chrono>
4
5  #include "ppconsul/agent.h"
6
7  using namespace wfrest;
8  using ppconsul::Consul;
9  using namespace ppconsul::agent;
10
11 void timer_callback(WFTimerTask* task)
12 {
13     SeriesWork* series = series_of(task);
14     Agent* agent = static_cast<Agent*>(series->get_context());
15
16     // 发送心跳包
17     agent->servicePass("UserService2");
18     // 创建另一个定时任务
19     WFTimerTask* nextTask = WFTaskFactory::create_timer_task(7, 0, timer_callback);
20     series->push_back(nextTask);
21 }
22
23 int main()
24 {
25     HttpServer server;
26
27     if (server.start(8888) == 0) {
28
29         // 指定注册中心 Consul 的ip地址, 端口和数据中心
30         Consul consul { "http://127.0.0.1:8500", ppconsul::kw::dc="dc1" };
31         // 创建代理
32         Agent agent { consul };
33         // 注册服务
34         agent.registerService(
35             kw::id = "UserService2",
36             kw::name = "UserService",
37             kw::address = "127.0.0.1",
38             kw::port = 8888,
39             kw::check = TtlCheck(std::chrono::seconds{ 10 })
40         );
41
42         // 定时发送心跳包
43         WFTimerTask* timerTask = WFTaskFactory::create_timer_task(7, 0,
timer_callback);
44
45         SeriesWork* series = Workflow::create_series_work(timerTask, nullptr);
46         series->set_context(&agent);    // 设置序列的上下文
47         series->start();
48
49         getchar();
50         server.stop();
```

```
51     } else {  
52         std::cerr << "Error: cannot start server!\n";  
53         std::exit(1);  
54     }  
55     return 0;  
56 }
```

服务注册好之后，API 网关和其它服务可以通过 URL:

```
1 | http://<consul_host>:8500/v1/health/service/<service_name>?passing=true
```

查找某个服务健康的实例。