OSS -- 使用公有云存储实现备份

使用云存储

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（实际上是非常难实现）。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。

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。

阿里云OSS的注册和开通

在本课程当中，我们选用了国内目前使用最广泛的阿里云对象存储OSS作为示例进行讲解，其他如AWS、Azure等产品的使用方法类似。

首先，用户需要注册阿里云账户并登录之，然后在控制台当中搜索OSS，进入控制界面后，点击“立即开通”按钮。在跳转的界面当中多次点击“立即开通”并且勾选相关协议即可。

在开通OSS服务完成之后，进入OSS管理控制台https://oss.console.aliyun.com/。（最好在费用页面给账户提前充值10元）

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。您可以根据实际需求，创建不同类型的存储空间来存储不同的数据。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，同时您也可以在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。您可以根据费用、请求来源等选择合适的地域创建Bucket。
- 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。
- 访问秘钥（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。

使用网页上传和下载文件

在第一次上传或下载文件之前，需要先创建一个Bucket。

1.单击“创建Bucket”的按钮。

2.随后配置Bucket的必要参。其他参数均可以保持默认参数，也可以在Bucket创建完成后单独配置。

```
- bucket名称：自定义，不能和其他的bucket重名。
- 地域：选择最近的地点即可。
- 存储类型：选择标准存储。
- 其余：默认选项。
```

3.创建完毕之后，就可以在bucket的管理页面，进入“文件管理”中的“文件列表”页面，就可以进行文件的上传和下载了。

使用C++ SDK访问OSS的准备工作

在对象存储的首页（OSS管理控制台），点击“OSS新手入门”，来到OSS的文档页面，在左边的索引列表中，找到SDK示例，选择C++，就可以开始学习如何使用SDK了。

1.下载SDK的安装包 aliyun-oss-cpp-sdk-master.zip，并上传到Ubuntu18.04的用户目录下

2.安装过程如下：

```
## 安装相关依赖
$ sudo apt install libcurl4
## 解压缩
$ unzip aliyun-oss-cpp-sdk-master.zip
## 生成makefile文件
$ cd aliyun-oss-cpp-sdk-master
$ mkdir build
$ cd bulid
$ cmake ..
## 编译和安装
$ make
$ sudo make install
$ sudo ldconfig
```

3.在链接时，需要增加以下链接选项

```
$ g++ test.cc -fno-rtti -lalibabacloud-oss-cpp-sdk -lcurl -lcrypto -lpthread
```

4.在项目中使用SDK时，还需要获取如下信息

- Bucket名：在创建Bucket的时候指定的
- 访问域名：在Bucket的概览页面中

- AccessKeyID和AccessKeySecret: 将鼠标移动到管理页面的右上角，会出现如下的页面，

点击“AccessKey管理”，会看到如下页面。如没有AccessKeyID,则点击"创建AccessKeyID"，之后生成对应的"AccessKeyID"和"AccessKey Secrete"。（注意该信息是不能泄漏的）。

后续如果忘了AccessKey Secret，可以点击红框处的“查看Secret”，通过验证后可获取到Secret

在项目中引入OSS

首先，我们设计一个类来存储OSS的属性，以供后续接口使用

```
struct OSSInfo {
    string EndPoint = "oss-cn-hangzhou.aliyuncs.com";
    string Bucket = "bucket-lwh-test0";
    string AccessKeyId = "your id";
    string AccessKeySecret = "your secret";
};
```

如果需要在项目中引入OSS作为备份存储，主要是要实现两个功能：文件上传和下载链接的生成。

文件上传

下面是文件上传的示例：

```cpp
#include <alibabacloud/oss/OssClient.h>
using namespace std;
using namespace AlibabaCloud::OSS;

int test0(void)
{
    /* InitializeSdk()用于初始化网络等资源，在程序生命周期内只需要调用一次。*/
    InitializeSdk();

    ClientConfiguration conf;
    OSSInfo oss;
    OssClient client(oss.Endpoint, oss.AccessKeyId, oss.AccessKeySecret,
conf);

    //上传文件
    string objectPath = "test/1.txt";//指定文件在OSS中的路径
    //参数1: 要上传到哪一个桶中
    //参数2: 要上传到桶中的路径
    //参数3: 本地要上传的文件本身
    auto outcome = client.PutObject(oss.Bucket, objectPath, "tmp/2.txt");
    if(outcome.isSuccess() == false) {
        //异常处理
        cout << "upload fail, code = " << outcome.error().Code()
             << ", msg = " << outcome.error().Message()
             << ", requestID = " << outcome.error().RequestId() << endl;
    }

    /* ShutdownSdk()用于释放网络等资源，在程序生命周期内只需要调用一次。*/
    ShutdownSdk();
    return 0;
}
```

文件下载链接的生成

```cpp
int test1()  {
    InitializeSdk();
    ClientConfiguration conf;
    OSSInfo oss;
    OssClient client(oss.Endpoint, oss.AccessKeyId, oss.AccessKeySecret,
conf);
    string objectPath = "test/1.txt";
    time_t tm = time(nullptr);
    auto outcome = client.GeneratePresignedUrl(oss.Bucket,
                   objectPath, tm + 1200, Http::Get);
    if(outcome.isSuccess() == false) {
        cout << "Fail, code: " << outcome.error().Code()
             << ", message: " << outcome.error().Message()
             << ", requestid: " << outcome.error().RequestId() << endl;
    } else {
        cout << outcome.result() << endl;
    }

    ShutdownSdk();
    return 0;
}
```

Docker

容器技术的基本概念

docker是容器技术的一种，目前也是容器技术事实上的标准。

首先，我们需要了解容器技术当中最基本的两个概念是**镜像**和**容器**。所谓的**镜像**，就是将一个应用程序与其所依赖的各种环境和配置文件打包成一个文件。而所谓的**容器**就是以一种隔离的方式从镜像当中启动运行的应用程序。

容器可以看成是一种轻量级的虚拟机：容器运行在宿主操作系统之上，在容器内部可以启动进程，这些进程天然地和宿主机、其他容器隔离；容器是轻量级的，相较于传统虚拟机会使用更少的资源，启动速度会更快；由于每一个容器都是由标准化的镜像产生，所以只需要把镜像迁移到不同的操作系统中，就可以无痛地在相同的环境启动应用程序；同一个镜像可以某个时刻运行着不同的容器，这样可以比较容易地部署集群应用。

容器解决了什么问题

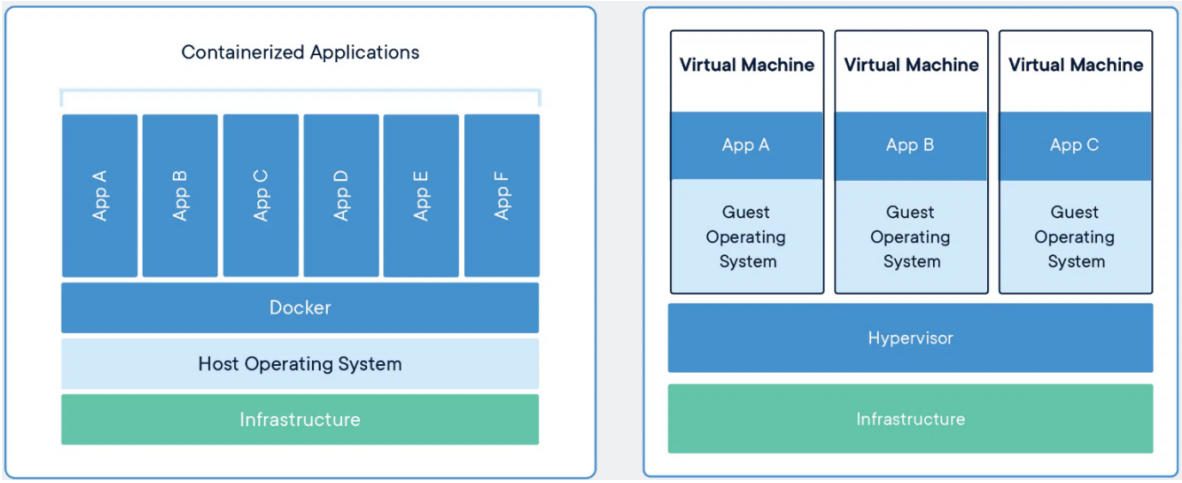
在容器技术诞生以前，开发者和运维人员最头痛的问题之一就是环境问题：假如一个应用程序依赖两个产品redis和mysql，在开发机上面的产品和业务机上面的产品版本很有可能是不一致的，假如只有少量的产品，那么开发机和业务机也许能比较容易地保持版本一致，随着产品越使用越多，这些产品的版本管理就成了一个大麻烦。更加严重的问题是版本冲突——多个应用可能会同时使用相同的产品，而所需求的产品版本号是不一样的！除此以外，集群的部署也是一个大问题，运维人员需要非常谨慎地选择集群当中每个节点的操作系统内核版本，为每个节点配置相同的环境，每次升级依赖或者操作系统内核都需要大量的人力物力。

针对上述问题，一种自然的解决思路就是将应用和其依赖的所有库、运行环境和配置文件打包在一起，这样就可以将应用和其依赖对应起来，再使用隔离机制，让不同的包之间彼此不发生冲突——这个包就是镜像，而对应运行的应用就是容器。通过同一个镜像可以启动多个彼此独立的容器，这样的话，集群的部署就变得相当容易了——只需拿到一份镜像然后为每个节点启动一个容器即可。

容器和虚拟机的区别

从容器的理念上来看，容器和我们之前所使用的虚拟机非常地相似——都为了不同的应用提供了一个隔离的环境，但是虚拟机是一个相当重量级的产品。虚拟机工作在Hypervisor的上一层，每个虚拟机都虚拟化出一个完整的客户操作系统，当虚拟机中的应用执行时，它会完整地转换成客户操作系统内核的指令，然后经由Hypervisor转交给底层的硬件设施或者宿主操作系统。由于不同虚拟机的内核是彼此独立的，为了维持多个虚拟的操作系统内核，宿主机需要消耗大量系统资源。

容器可以看成是一种轻量级的虚拟机，它也使用了虚拟化技术，但是它并没有把内核给虚拟化——多个独立运行的容器拥有自己独立的库和其他依赖，但是操作系统内核却是共享的。这种共享是由容器引擎配合操作系统内核一起实现的——操作系统会将文件系统以及其他系统分隔成多个不同彼此之间不可见Namespace，并且通过Cgroup机制来控制每个Namespace资源使用上限，当容器中的应用执行的时候，容器引擎负责找到它对应内核的Namespace，然后交由内核执行指令。由于不需要完整地将内核虚拟化出来，所以指令没有经过多次转换，那么执行速度自然是很快，又由于共享内核，所以也不需要消耗大量内存来存储。这样就造就了容器的核心优势：既有隔离性，又十分轻量级。



docker的安装

首先切换到root用户，然后执行下列指令（需要提前安装好curl命令）

```
$ su
# curl-fsSLhttps://get.docker.com|bash-sdocker--mirrorAliyun
```

当指令执行完成之后，docker就已经安装好，并且处于运行中了。由于直接从docker官网当中拉取镜像会十分缓慢，故可以通过配置阿里云加速来提升拉取速度。

- 登录阿里云账户
- 找到容器镜像服务 -> 镜像工具 ->镜像加速器

点击**立即开通**后，进入如下页面，找到**加速器地址**

- 修改docker daemon的配置文件/etc/docker/daemon.json。（不同用户的加速地址不一致）

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://rkbo63bn.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

- 测试docker是否安装完成

```
# 请在root下使用如下命令
docker run hello-world
```

```
root@ubuntu:/home/lwh# docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

镜像相关的命令

为了简化使用，本课程中的所有docker相关命令都在root用户下执行。

想要启动一个容器，首先要找到它的镜像。镜像相关的命令如下：

- 列出所有的镜像：

```
# docker images
REPOSITORY    TAG          IMAGE ID       CREATED         SIZE
lwh/nginx     1.0          9d0d1fcced2c   6 days ago      142MB
nginx         1.22.1       8c9eabeac475   2 weeks ago     142MB
```

  这个命令会展示镜像的信息：包括镜像的仓库源(REPOSITORY)、标签(TAG)、镜像ID(IMAGE ID)、创建时间和大小。

- 从远程仓库拉取镜像到本地：

```
# docker pull nginx:1.22.1
```

  值得注意的是，镜像是一种分层结构，有些不同的镜像可能会共享一些分层，在拉取新的镜像的时候，

  假如本地已经拉取了其中某个分层（因为该分层在其他镜像中也存在），那么就可以不用重复拉取。拉

  取镜像的来源地是仓库——一般使用的是公共仓库，比如默认的hub.docker.com，团队自己也可以建立

  私有仓库。

- 删除本地镜像：

```
# docker rmi hello-world
```

容器相关的命令

通过docker run命令可以从镜像当中启动容器。

```
# docker run nginx
```

默认情况，执行docker run的效果会寻找镜像，如果相应的镜像不存在，那么会自动从公开仓库当中拉取；然后启动运行容器；随后附加的命令，在本例中附加命令为空。对应nginx容器，就会进入容器内部nginx的交互终端，可以使用ctrl+c从容器当中退出，但是同时也会停止容器。

```
列出当前正在运行的容器
# docker ps

列出所有的容器
# docker ps -a

删除停止状态的容器
# docker rm 容器名|ID
```

- 如果给容器指定附加指令，对于nginx容器而言，会直接执行该指令，随后容器就停止了。

```
# docker run nginx echo hello
hello
```

- 假如需要在容器执行更多的指令，那么就要进入容器内部，将shell进程拉起来，并将宿主机的stdin连接到容器内的终端。

```
# docker run -it nginx /bin/bash
```

上述命令执行之后，用户就可以进入容器内部，像使用一个普通的shell操作容器内部文件系统。使用ctrl+p,q可以在不停止容器的情况下离开容器；而使用ctrl+d可以退出容器并停止容器。

- 使用-d选项，可以以守护进程的形式运行容器

```
# docker -d nginx
```

- 假设一个容器已经存在，正在运行，使用docker exec可以附加到该容器中，执行其他命令，也可以将stdin和终端连入容器。

```
# docker exec -it 容器名 /bin/bash
```

- 停止容器

```
# docker stop 容器名|ID
```

- 使用下列指令可以删除批量停止的容器

```
# docker rm $(docker ps -a -q -f status=exited)
```

- 启动停止的容器

```
# docker start 容器名|ID
```

制作镜像

在容器中，可以对容器的内容比如文件系统做一些修改，可以使用命令docker commit将运行中的容器内容打包成新的镜像，这样容器做的修改就可以持久保存了。

```
# docker commit 容器名 仓库名:TAG
```

使用docker images可以查看制作的容器。这个容器也可以使用docker push推送到公共仓库或者是私有仓库当中。

端口映射

容器天生具有隔离的特性，所以不同的容器拥有独立的网络系统。这样的话，两个不同的容器在容器内部完全可以使用相同的端口，并且可以使用宿主机当中已经使用的端口——这些相同的端口号实际上是毫无关联的，彼此之间也无法进行网络通信。如果希望通过容器来对外提供服务，就需要为容器的端口和宿主机的端口建立映射。

使用docker run的-p选项可以实现端口映射

```
将宿主机的8080端口映射到容器中的80端口
# docker run -d -p 8080:80 nginx
```

数据持久化

容器天生具有隔离的特性，所以不同的容器拥有独立的文件系统。在容器内部做任何的修改，都不会影响到宿主机或者其他容器，如果需要将容器的运行内容持久地存储，比如对于数据库类型的应用，就需要将容器内部的某个目录挂载到宿主机文件系统当中。所谓的挂载，就是将容器的某个目录看成是宿主系统的虚拟文件系统的一颗子树，类似在一个树上面嫁接枝丫。在往机器上面插入U盘往往也会执行挂载操作。

使用docker run的-v选项可以实现数据持久化

```
# docker run -d -p 8081:80 -v /tmp/test:/usr/share/nginx/html nginx
```

容器中访问的/usr/share/nginx/html和宿主系统的/tmp/test的内容是一致，并且任何形式的修改都会在宿主系统和容器当中共享。

```
在容器内部执行
# echo "<html>hello,world</html>" > /usr/share/nginx/html/index.html
相当于在宿主机中执行
# echo "<html>hello,world</html>" > /tmp/test/index.html
```

消息队列

消息队列概述

消息（Message）是指在应用程序间传送的数据。消息可以很简单，比如就是一个字符串，也可以很复杂。

消息队列的全称是Message Queue(MQ)，是一种应用程序间的通信方法。MQ是生产者-消费者模型的一个典型代表。消息发布者（生产者）往消息队列中不断写入消息，而消息使用者（消费者）可以不断读取队列中的消息。消息发布者只管把消息发布到MQ中，而不用理会谁来取消息。而消息使用者只管从MQ中取消息，而不用理会是谁发布的消息。这样，发布者和使用者都不需要知道对方的存在。

为什么要使用消息队列?

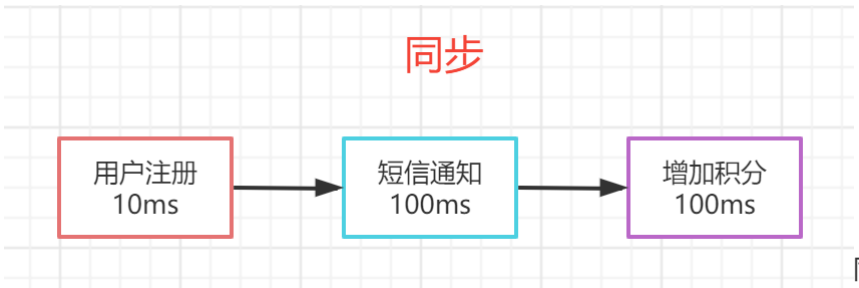
消息队列具有三个比较关键的作用：异步、解耦、削峰。接下来，我们看几个应用场景。

异步处理

消息队列的主要特点是异步处理，主要目的是减少请求响应时间，实现**非核心流程**异步化，提高系统响应性能。

举一个**用户注册**的例子，用户注册成功后，系统需要发送短信注册成功通知，以及赠送注册成功的积分。

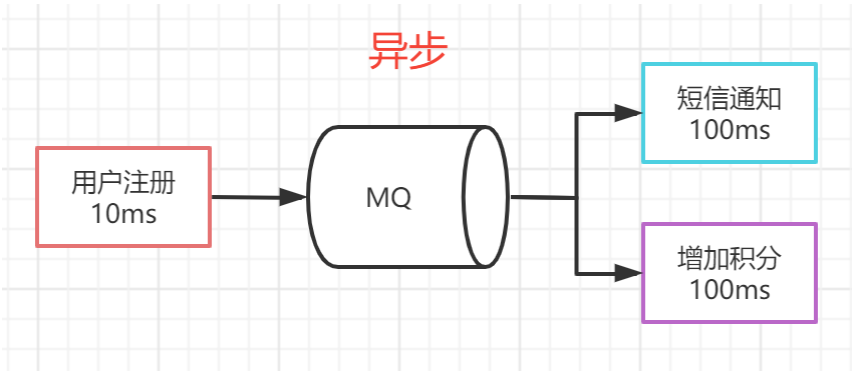
1）同步方案



同步的总耗时：10ms + 100ms + 100ms = 210ms

由于短信通知与增加积分为**非核心流程**，为了提升系统响应性能，从而将其改造为异步。

2）异步方案

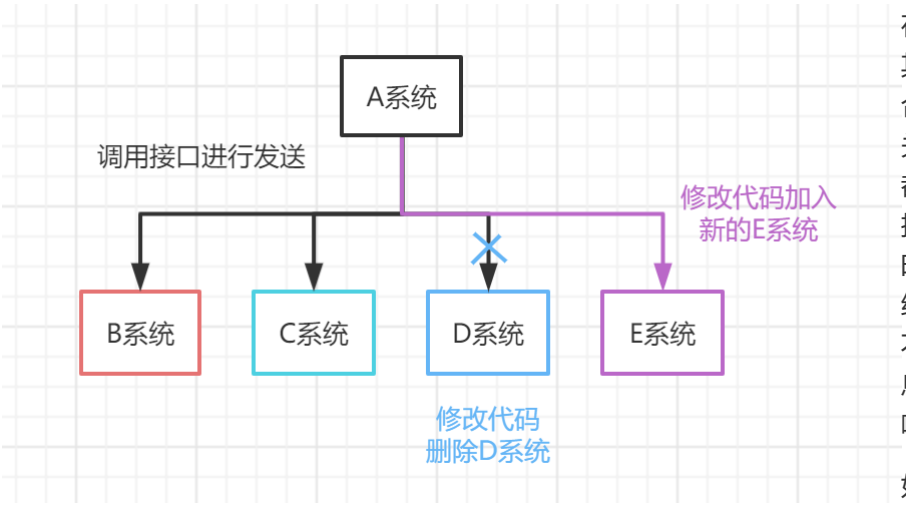


改造后就变成上图，之前需要210ms才能返回，现在把**短信通知**和**增加积分**改为异步的形式，用户注册后写入消息**10ms**左右立即返回成功给客户端，客户端无需等待耗时较久的同步(短信+积分)操作就可以返回，从而极大的提升了系统的吞吐量。

总结：所以异步的典型场景就是将比较耗时而且不需要即时（同步）返回结果的操作，通过消息队列来实现异步化。

解耦

我们来看这样一个场景。A系统发送数据到BCD三个系统，通过接口调用发送。如果E系统也要这个数据呢？那如果C系统现在不需要了呢？A系统负责人几乎崩溃……



在这个场景中，A系统跟其它各种系统严重耦合，A系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。A系统要时时刻刻考虑BCDE四个系统如果挂了该咋办？要不要重发，要不要把消息存起来？头发都白了啊！

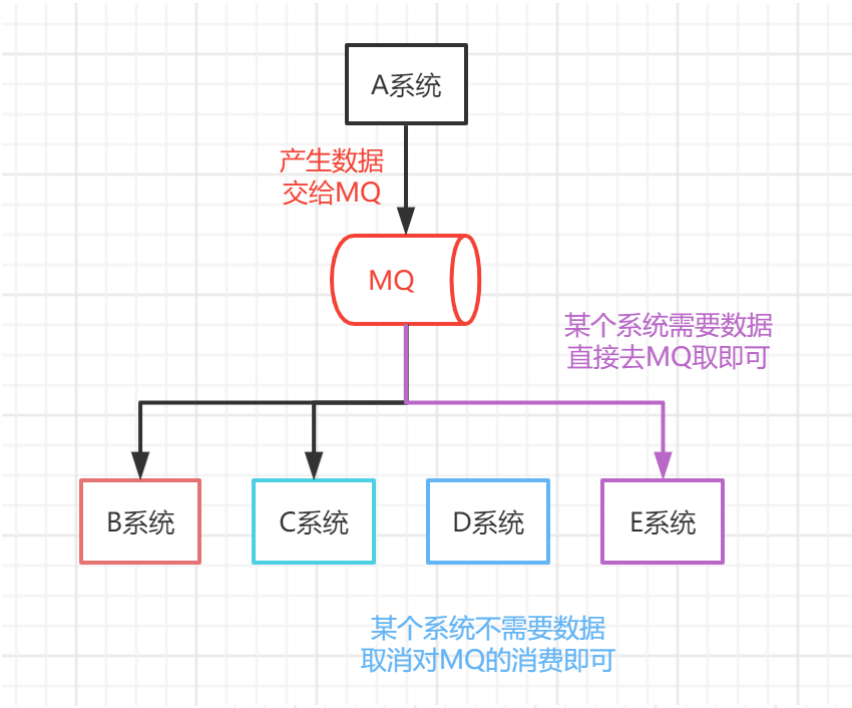
如果使用 MQ，A系统产生一条数据，发送到MQ里面去，哪个系统需要数据自己去MQ里面消费。如果新系统需要数据，直接从MQ里消费即可；如果某个系统不需要这条数据了，就取消对MQ消息的消费即可。这样下来，A系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑其他系统是否调用成功、失败超时等情况。

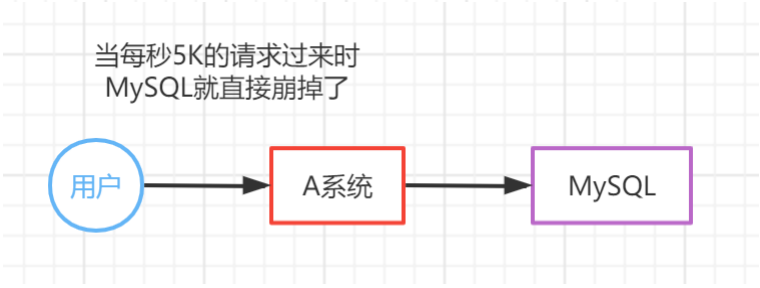
总结：通过一个MQ，Pub/Sub发布订阅消息这样一个模型，A 系统就跟其它系统彻底解耦了。

削峰

在一个秒杀场景中，没有执行秒杀行为时，订单系统A风平浪静，每秒并发请求数量就50个。结果每次一到 19:00 ~ 20:00 ，每秒并发请求数量突然会暴增到 5k+ 条。但是系统A是直接连接MySQL 的，大量的请求涌入MySQL，每秒钟对 MySQL执行约 5k 条 SQL。

一般，MySQL扛到每秒 2k 个请求就差不多了，如果每秒请求到 5k 的话，可能就直接把 MySQL 给打死了，导致系统崩溃，用户也就没法再使用系统了。

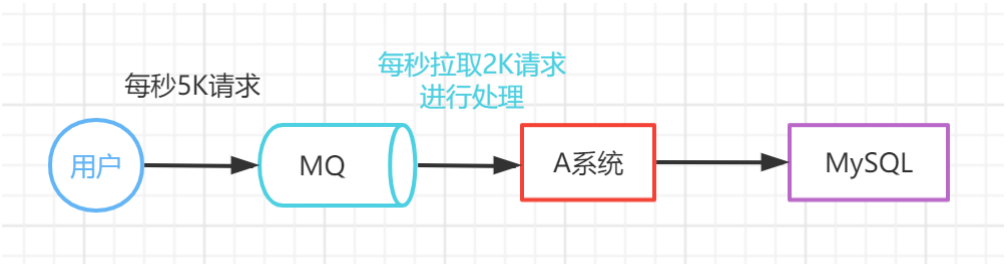




但是高峰期一过，就成了低峰期，可能也就 1w 的用户同时在网站上操作，每秒中的请求数量可能也就 50 个请求，对整个系统几乎没有任何的压力。

采用MQ之后：

如果使用 MQ，每秒 5k 个请求写入 MQ，A 系统每秒钟最多处理 2k 个请求，因为 MySQL 每秒钟最多处理 2k 个。A 系统从 MQ 中慢慢拉取请求，每秒钟就拉取 2k 个请求，不要超过自己每秒能处理的最大请求数量就 ok，这样下来，哪怕是高峰期的时候，A 系统也绝对不会挂掉。而 MQ 每秒钟 5k 个请求进来，就 2k 个请求出去，结果就导致在晚上高峰期（1 个小时），可能有几十万甚至几百万的请求积压在 MQ 中。



这个短暂的高峰期积压是 ok 的，因为高峰期过了之后，每秒钟就 50 个请求进 MQ，但是 A 系统依然会按照每秒 2k 个请求的速度在处理。所以说，只要高峰期一过，A 系统就会快速将积压的消息给解决掉。

市场上MQ的对比

市场上的MQ产品有ActiveMQ、RabbitMQ、RocketMQ、Kafka，特点如下表：

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级	万级	10万级	10万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic数量对吞吐量的影响			topic可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是其一大优势，在同等机器下，可以支撑大量的topic	topic从几十到几百个时候，吞吐量会大幅下降，在同等机器下，尽量保证topic数量不要过多，如果要支撑大规模的topic，需要增加更多的机器资源
时效性	ms级	**微秒级**	ms级	ms级
可用性	高，基于主从架构实现高可用	同ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据
消息可靠性	有较低的概率丢失数据	**基本不丢**	经过参数优化配置，可以做到0丢失	同RocketMQ
功能支持	MQ领域的功能及其完备	基于erlang开发，并发能力很强，性能极好，延时很低	MQ功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用

综上，各种对比之后，有如下建议：

一般的业务系统要引入 MQ，最早大家都用 ActiveMQ，但是现在确实大家用的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃，就不推荐用这个了；

后来大家开始用 RabbitMQ，但是确实 erlang 语言阻止了大量的 Java 工程师去深入研究和掌控它，对公司而言，几乎处于不可控的状态，但是确实人家是开源的，比较稳定的支持，活跃度也高；

不过现在确实越来越多的公司会去用 RocketMQ，确实很不错，毕竟是阿里出品。目前 RocketMQ 已捐给 Apache，但 GitHub 上的活跃度其实不算高。对自己公司技术实力有绝对自信的，推荐用 RocketMQ，否则回去老老实实用 RabbitMQ 吧，人家有活跃的开源社区，绝对不会黄。

总结：

1）**中小型公司**，技术实力较为一般，技术挑战不是特别高，用 RabbitMQ 是不错的选择；

2）**大型公司**，基础架构研发实力较强，用 RocketMQ 是很好的选择。

3）如果是**大数据领域**的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，几乎是全世界这个领域的事实性规范。

RabbitMQ

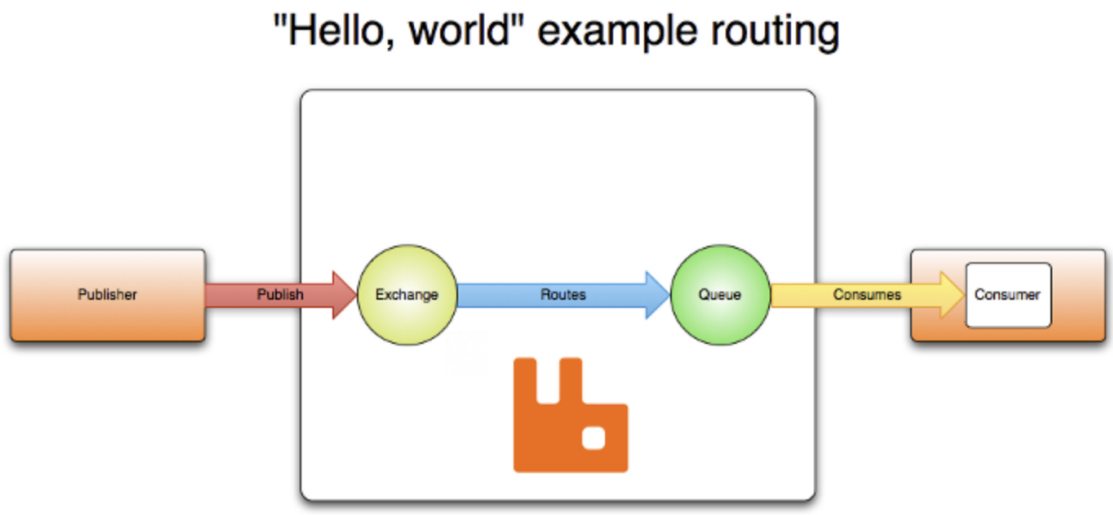
RabbitMQ的设计是基于AMQP协议的。接下来，我们了解一下AMQP协议的相关知识。

AMQP协议

AMQP（高级消息队列协议）是一个网络协议。它支持符合要求的客户端应用（application）和消息中间件代理（messaging middleware broker）之间进行通信。

AMQP协议中包含了几个重要的实体，分别是**队列**、**交换机**、和**绑定**。

AMQP协议工作的过程如下图所示：



队列

AMQP中的队列（queue）跟其他消息队列或任务队列中的队列是很相似的：它们存储着即将被应用消费掉的消息。

交换机

交换机是用来发送消息的AMQP实体。交换机拿到一个消息之后将它路由给一个或零个队列。它使用哪种路由算法是由交换机类型和被称作**绑定**（bindings）的规则所决定的。

交换机有四种类型：

- 直连交换机（Direct exchange）：单播
- 扇型交换机（Fanout exchange）：广播
- 主题交换机（Topic exchange）：组播
- 头交换机（Headers exchange）：复杂业务

绑定

绑定（Binding）是交换机（exchange）将消息（message）路由给队列（queue）所需遵循的规则。如果要指示交换机“E”将消息路由给队列“Q”，那么“Q”就需要与“E”进行绑定。绑定操作需要定义一个可选的路由键（routing key）属性给某些类型的交换机。路由键（routing key）的意义在于从发送给交换机的众多消息中选择出某些消息，将其路由给绑定的队列。

RabbitMQ的安装

在学习了docker之后，我们安装RabbitMQ就变得简单了，直接使用docker pull命令就可以解决。

```
$ su
# docker pull rabbitmq:management
```

启动RabbitMQ

在docker中运行rabbitmq只需要执行以下命令即可。

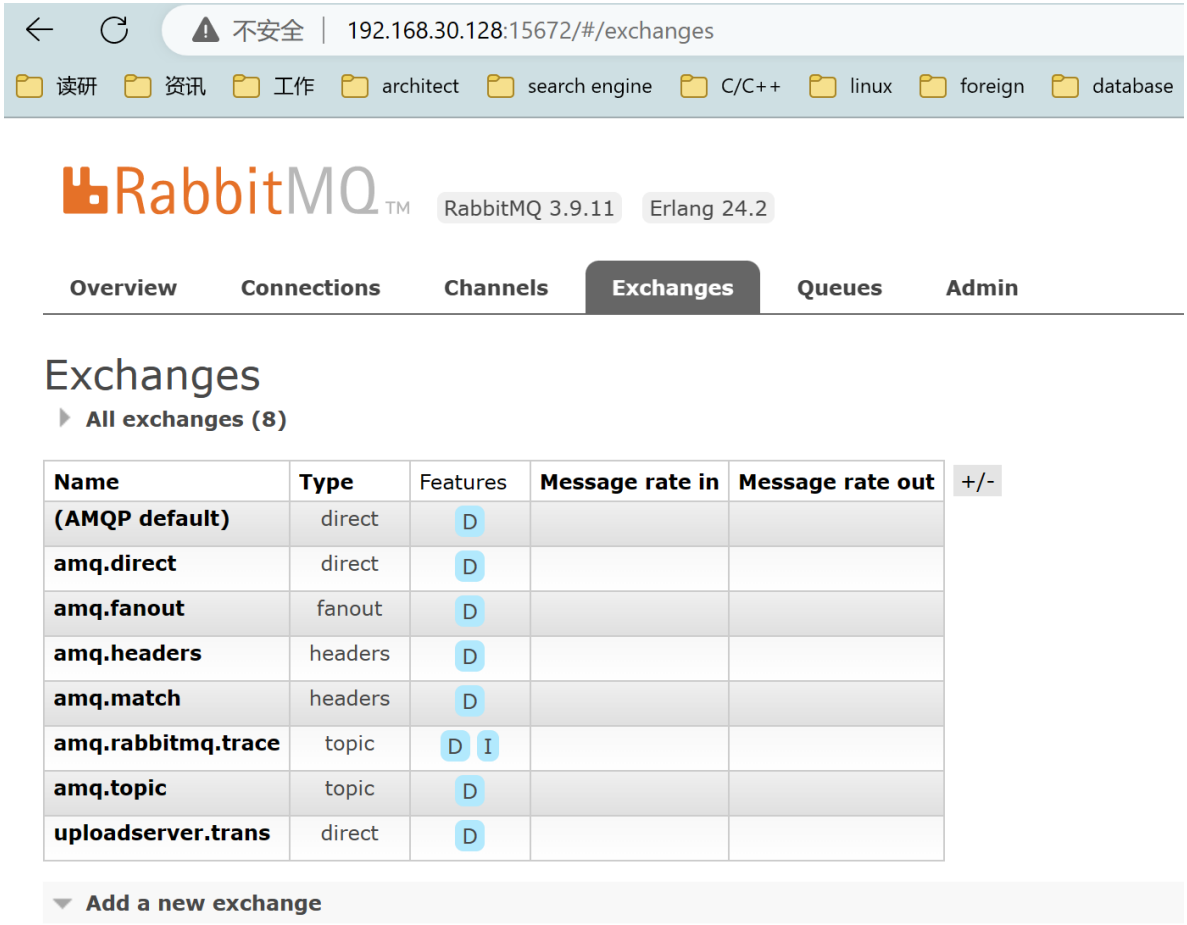
```
# docker run -d --hostname rabbitsvr --name rabbit
    -p 5672:5672 -p 15672:15672 -p 25672:25672
    -v /data/rabbitmq:/var/lib/rabbitmq rabbitmq:management
```

其中 5672是AMQP协议的端口，15672是HTTP协议的端口，25672是集群的端口

在Web中操作RabbitMQ

在浏览器中输入 `http://{ip}:15672`,即可进入登录界面

输入账户信息`{guest:guest}`后，进入`RabbitMQ`的管理页面



第三方库连接RabbitMQ

docker中的RabbitMQ启动后，是以服务器的形式运行的。要连接RabbitMQ，则需要使用客户端。在这里我们使用以下两个安装包 `rabbitmq-c-0.11.0.tar.gz` 和 `SimpleAmqpClient-2.5.1.tar.gz`，前者为C客户端，后者为C++客户端。

安装过程如下：

```
# 安装rabbitmq-c-0.11.0.tar.gz
$ tar xzvf rabbitmq-c-0.11.0.tar.gz
$ cd rabbitmq-c-0.11.0
$ mkdir build
$ cmake ..
$ make
$ sudo make install

# 安装SimpleAmqpClient-2.5.1.tar.gz
$ tar xzvf SimpleAmqpClient-2.5.1.tar.gz
$ cd SimpleAmqpClient-2.5.1
$ mkdir build
$ cmake ..
$ make
$ sudo make install
```

接下来，我们来看一个测试用例，我们先用一个类来设置好消息队列的属性

```
struct MQInfo {
    string URL = "amqp://guest:guest@127.0.0.1:5672";
    string Exchange = "uploadserver.trans";
    string OSSQueue = "uploadserver.trans.oss";
    string RoutingKey = "oss";
};
```

生产者发布消息：

```
void test0()
{
    MQInfo info;
    //创建一个channel
    Channel::ptr_t channel = Channel::Create();
    //创建一个消息
    BasicMessage::ptr_t message = BasicMessage::Create("message");
    //发布消息
    channel->BasicPublish(info.Exchange, info.RoutingKey, message);
}
```

消费者获取消息：

```
void test1()
{
    MQInfo info;
    Channel::ptr_t channel = Channel::Create();
    //启动队列消费者
    channel->BasicConsume(info.OSSQueue, info.OSSQueue);

    Envelope::ptr_t envelope;
    //取出消息，放入一个信封
    bool flag = channel->BasicConsumeMessage(envelope);
    if(flag == false) {
        cout << "time out!" << endl;
        return;
    }
    cout << "Body = " << envelope->Message()->Body() << endl;
}
```

在项目中引入RabbitMQ

引进项目时，需要考虑是否要开启异步备份，可以设置如下类

```
enum StoreType {
    STORE_TYPE_LOCAL,
    STORE_TYPE_OSS
};

struct MQConfig{
    //是否开启备份
    StoreType currentStoreType = STORE_TYPE_OSS;
    //备份是否启用异步转移
    bool isAsyncTransferEnable = true;//默认开启
```

```
    //交换器的名称
    string transExchange = "uploadserver.trans";
    //routingkey
    string transRoutingKey = "oss";
};
```

在上传文件操作时，选择不直接备份到OSS，只是要将要上传的文件信息发布到RabbitMQ即可：

```
{
     //使用OSS备份，启用异步转移(通过消息队列来完成)
     using namespace AmqpClient;
     //消息队列的生产者
     Channel::ptr_t channel = Channel::Create();
     //传输文件名字、文件的哈希值、文件路径
     Json filejson;
     filejson["filehash"] = filehash;
     filejson["filename"] = filename;
     filejson["filepath"] = filepath;
     //构造消息
     BasicMessage::ptr_t message = BasicMessage::Create(filejson.dump());
     //发布消息
     channel->BasicPublish(mqconfig.transExchange,
                           mqconfig.transRoutingKey,
                           message);
}
```

另外再写一个单独的消费者进程，通过它获取消息后，再开启上传操作

```
#include "mq.h"
#include "ossinfo.h"

#include <string>
#include <iostream>
#include <wfrest/json.hpp>
#include <SimpleAmqpClient/SimpleAmqpClient.h>
#include <alibabacloud/oss/OssClient.h>

using std::cout;
using std::endl;
using std::string;
using namespace AmqpClient;
using namespace AlibabaCloud::OSS;
using Json = nlohmann::json;

int main() {
    MQInfo mqinfo;
    OSSInfo ossinfo;
    InitializeSdk();

    //消费者
    Channel::ptr_t channel = Channel::Create();
    channel->BasicConsume(mqinfo.OSSQueue, mqinfo.OSSQueue);
    while(true) {
        Envelope::ptr_t envelope;
        bool flag = channel->BasicConsumeMessage(envelope, 15000);
        if(flag == false) {
```

```
            cout << "time out!" << endl;
            continue;
        }

        Json fileJson = Json::parse(envelope->Message()->Body());
        string currentLocation = fileJson["filepath"];
        string ossLocation = "oss/";
        string filehash = fileJson["filehash"];
        ossLocation += filehash;

        cout << "filehash:" << filehash << endl;
        cout << "filepath:" << currentLocation << endl;
        cout << "osslocation: " << ossLocation << endl;

        ClientConfiguration conf;
        OssClient client(ossinfo.EndPoint,
                         ossinfo.AccessKeyId,
                         ossinfo.AccessKeySecret,
                         conf);
        auto outcome = client.PutObject(ossinfo.Bucket,
                                        ossLocation,
                                        currentLocation);
        if(outcome.isSuccess() == false) {
            cout << "Fail, code: " << outcome.error().Code()
                 << ", message: " << outcome.error().Message()
                 << ", requestid: " << outcome.error().RequestId() << endl;
        }
    }
    ShutdownSdk();
}
```

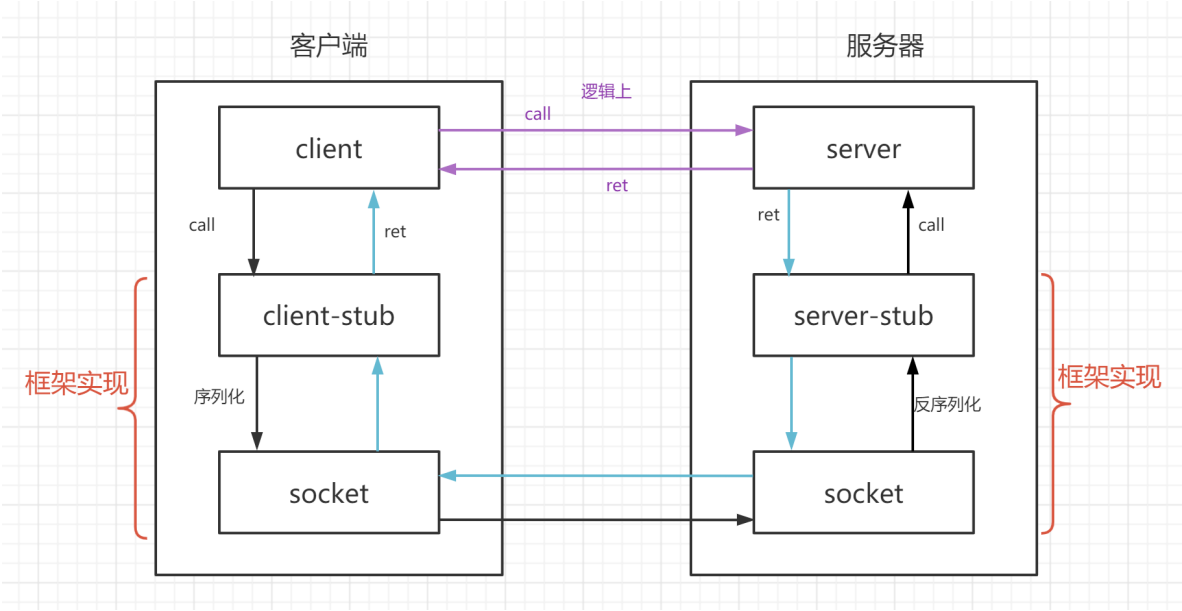
RPC

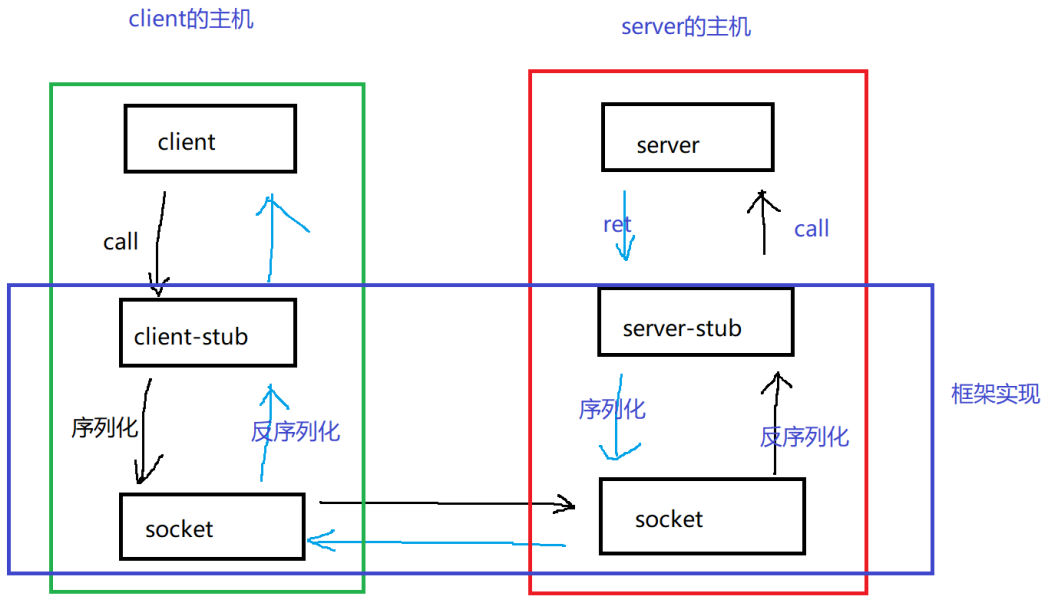
什么是RPC?

RPC（Remote Procedure Call），远程过程调用，它把网络交互类比为client调用server上的函数。RPC 的主要功能目标是让构建分布式计算（应用）更容易，在提供强大的远程调用能力时不损失本地调用的**语义简洁性**。为实现该目标，RPC 框架需提供一种**透明调用机制**，让使用者不必显式的区分本地调用和远程调用。

如何设计一个RPC呢?

- 代理模式的设计
- 打包数据：序列化和反序列化
- 完成网络通信：网络协议





有哪些RPC框架?

- **Thrift**：最初是由 Facebook 开发的内部系统跨语言的 RPC 框架，2007 年贡献给了 Apache 基金，成为Apache 开源项目之一，支持多种语言。
- **gRPC**：Google 于 2015 年对外开源的跨语言 RPC 框架，支持多种语言。
- **Tars**：腾讯内部使用的 RPC 框架，于 2017 年对外开源，仅支持 C++ 语言。
- **bRPC**：百度内最常使用的工业级RPC框架，目前只开源C++版本。
- **srpc**：是sogou内部自研的基于Workflow的RPC项目，2020年开源。

SRPC的特点

- 序列化方案用的是protobuf
- 服务器端采用workflow：每当有客户端调用时，服务端会创建一个特殊的任务，其基本工作和回调是分开的
- 代理模式

protobuf的安装

```
$ tar zxvf protobuf-3.20.1.tar.gz
$ cd protobuf-3.20.1
$ ./autogen.sh
$ ./configure
$ make -j4
$ sudo make install
$ sudo ldconfig
$ protoc --version      # 查看版本号
```

srpc的安装

安装srpc的前置条件是要先安装`protbuf`，之后再来安装`srpc`。

```
$ tar zxvf srpc.tar.gz
$ cd srpc
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
$ sudo ldconfig
```

protobuf的特点

- 专门为RPC设计的序列化方案
- 数据量小，解析速度非常快（二进制）
- 向前兼容和向后兼容
- 跨语言

IDL文件

在学习protobuf时，我们先需要定义一个IDL（Interface Description Language）文件，该文件是一种很有用的工具，它提供了对接口的描述，约定了接口协议。无需关心客户端和服务器会用哪种编程语言进行通信，这样就达到了跨语言的目标。

在定义了IDL文件后，可以使用`protoc`编译器生成不同语言的客户端和服务器程序。

消息的定义

先来看一个简单的例子，类似于一个`helloworld`程序。该例子位于一个后缀为`.proto`的文件中，我们假设为`search.proto`文件。

```
//位于文件search.proto中
syntax = "proto3";//指定proto的版本，默认情况下为proto2
message SearchRequest {
    string query = 1;            // 查询的关键词
    int32 page_number = 2;       // 查询第几页
    int32 result_per_page = 3;   // 每一页包含几条结果
}
```

在上面的例子中，我们指定了3个字段`field`，其中一个字符串和2个整型数据。当然也可以是更加复杂的字段，比如枚举类型或者嵌套的`message`类型。

分配字段编号

每个字段有一个类型和一个名字，并且每一个字段定义了一个**唯一**的数字标识号。需要注意的是：

- 编号的范围是 1 - 2^29
- 1 - 15 的编号在编码时占用1个字节
- 16 - 2047 的编号在编码时占用2个字节
- 19000 - 19999 是系统预留的，不能使用
- 对于那些**经常使用的字段**，建议使用**1-15**来作为他们的数字标识号
- 对于那些不经常使用的字段，建议使用**16 - 2047**

字段规则（Filed Rules）

每个字段所指定的字段修饰符必须是以下2种之一：

- `singular:` 一个格式良好的消息应该是有0个或1个这种字段。在`proto3`的语法规则中，默认情况下，字段修饰符就是`singular`的
- `repeated:` 在一个格式良好的消息中，这种字段可以重复任意多次，包括0次，类似于数组，他们的顺序会被保存，通过索引进行检索。

添加多个消息类型

在一个`.proto`文件中，可以定义多个`message`类型。比如我们在上面所述的`search.proto`文件中，再定义一个`SearchResponse`消息。

```
//search.proto文件
syntax = "proto3";
message SearchRequest {
    string query = 1;            // 查询的关键词
    int32 page_number = 2;       // 查询第几页
    int32 result_per_page = 3;   // 每一页包含几条结果
}

message SearchResponse {
    repeated Result results = 1;
}

message Result {
    string url = 1;                 // 链接
    string title = 2;               // 标题
    repeated string snippets = 3;   // 片段
}
```

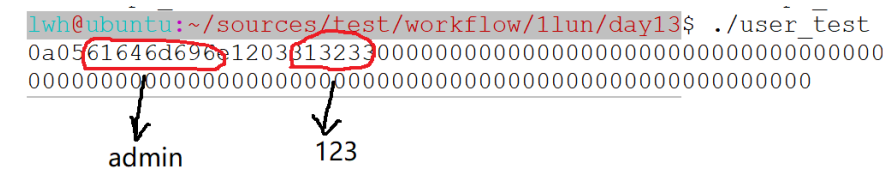
从`.proto`文件中生成了什么?

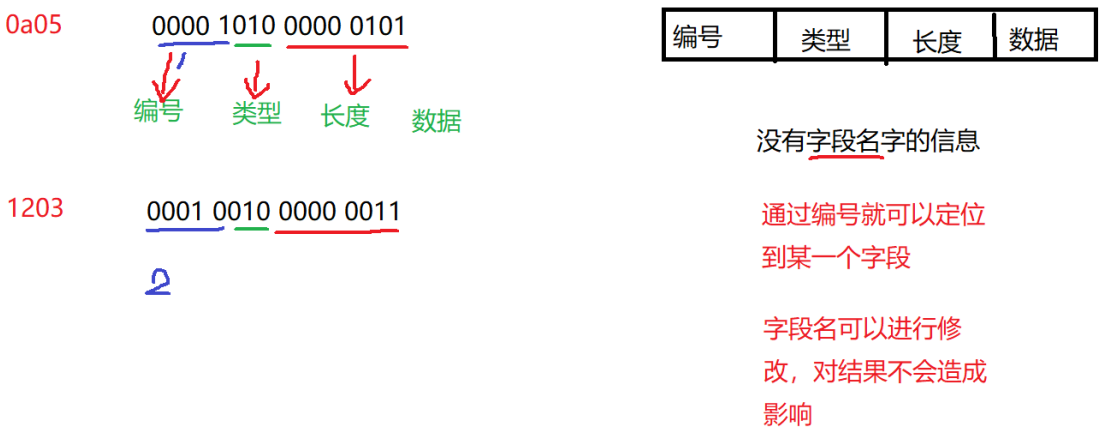
当用`protocol buffer`编译器来运行`.proto`文件时，编译器将生成所选语言的代码，这些代码可以操作在`.proto`文件中定义的消息类型，包括获取、设置字段值，将消息序列化到一个输出流中，以及从一个输入流中解析消息。在此，我们只演示`C++`相关的操作。我们在命令行中进行如下操作：

```
$ protoc --cpp_out=./ search.proto
```

就会生成`search.pb.h`和`search.pb.cc`这两个文件。之后就是对这两个文件的使用了。

字节流的设计





向后兼容

向后兼容（Backward Compatibility）：指新的版本的软 / 硬件可以使用老版本的软 / 硬件产生的数据。

在protobuf中，向后兼容表示的是新代码可以解析旧的proto文件中的数据，则只需要保证新字段是optional即可。

向前兼容

向前兼容（Forward Compatibility）：指老的版本的软 / 硬件可以使用新版本的软 / 硬件产生的数据。

在protobuf中，向前兼容表示的是旧代码可以解析新数据，则需要做到以下几点：

- 修改字段时，可以修改名字，不能修改类型和编号
- 添加新字段时，用新编号，并设置为optional
- 新的proto文件中，永远不删旧字段
- 遇到不知道的字段，直接忽略

srpc如何使用protobuf?

1. 设计IDL文件

- 设计两个message
- 设计函数的名字信息

```
//指定protobuf的版本 proto2/proto3
syntax = "proto3";

//定义消息，就是参数列表
message ReqSignup{
    string username = 1;
    string password = 2;
}

message RespSignup{
    int32 code = 1;
    string message = 2;
}

service UserService{
    rpc Signup(ReqSignup) returns (RespSignup);
}
```

2. 生成代码

```
$ protoc --cpp_out=./  user.proto
# 生成user.pb.h和user.pb.cc

$ srpc_generator protobuf user.proto ./
#生成user.srpc.h/server.pb_skeleton.cc/client.pb_skeleton.cc
```

3. 解析生成的文件

文件user.srpc.h

即代理模式的设计

```
namespace UserService
{
/*
 * Server codes
 * Generated by SRPC
 */
class Service : public srpc::RPCService
{
public:
    // please implement these methods in server.cc

    virtual void Signup(ReqSignup *request, RespSignup *response,
                        srpc::RPCContext *ctx) = 0;

public:
    Service();
};


/*
 * Client codes
```

```
 * Generated by SRPC
 */
using SignupDone = std::function<void (RespSignup *, srpc::RPCContext *)>;

class SRPCClient : public srpc::SRPCClient
{
public:
    void Signup(const ReqSignup *req, SignupDone done);
    void Signup(const ReqSignup *req, RespSignup *resp,
srpc::RPCSyncContext *sync_ctx);
    WFFuture<std::pair<RespSignup, srpc::RPCSyncContext>>
async_Signup(const ReqSignup *req);

public:
    SRPCClient(const char *host, unsigned short port);
    SRPCClient(const struct srpc::RPCClientParams *params);

public:
    srpc::SRPCClientTask *create_Signup_task(SignupDone done);
};
}// end of namespace UserService
```

文件client.pb_skeleton.cc

```
#include "user.srpc.h"
#include "workflow/WFFacilities.h"

using namespace srpc;

static WFFacilities::WaitGroup wait_group(1);

void sig_handler(int signo)
{
    wait_group.done();
}

static void signup_done(RespSignup *response, srpc::RPCContext *context)
{   //在收到响应时调用该回调函数
}

int main()
{
    GOOGLE_PROTOBUF_VERIFY_VERSION;
    const char *ip = "127.0.0.1";
    unsigned short port = 1412;

    UserService::SRPCClient client(ip, port);//client_stub

    // example for RPC method call
    ReqSignup signup_req;
    //signup_req.set_message("Hello, srpc!");
    client.Signup(&signup_req, signup_done);

    wait_group.wait();
    google::protobuf::ShutdownProtobufLibrary();
    return 0;
}
```

文件server.pb_skeleton.cc

```cpp
#include "user.srpc.h"
#include "workflow/WFFacilities.h"

using namespace srpc;

static WFFacilities::WaitGroup wait_group(1);

void sig_handler(int signo)
{
    wait_group.done();
}

class UserServiceServiceImpl : public UserService::Service
{
public:
    void Signup(ReqSignup *request, RespSignup *response, srpc::RPCContext *ctx) override
    {}
};

int main()
{
    GOOGLE_PROTOBUF_VERIFY_VERSION;
    unsigned short port = 1412;
    SRPCServer server;

    UserServiceServiceImpl userservice_impl;
    server.add_service(&userservice_impl);

    server.start(port);
    wait_group.wait();
    server.stop();
    google::protobuf::ShutdownProtobufLibrary();
    return 0;
}
```