3.2 开始CAP

3.2.1 基础环境准备

1 多个空的Core WebApi

2 多个SqlServer数据库脚本+ 多个DBContext数据库访问

3 RabbitMQ

docker stop $(docker ps -q) & docker rm $(docker ps -aq) #停用并删除全部容器

```
1.拉取RabbitMQ镜像

    docker pull rabbitmq:management    //拉取包含web管理界面的RabbitMQ镜像

2.启动容器

    docker run -d --hostname my-rabbit --name rabbit -p 15672:15672 -p 5672:5672
rabbitmq:management        // 用户名密码都guest

    或者

    docker run -d --hostname my-rabbit --name rabbit -e
RABBITMQ_DEFAULT_USER=user -e RABBITMQ_DEFAULT_PASS=password -p 15672:15672 -p
5672:5672 rabbitmq:management    // 自定义用户名和密码

3.可视化
http://39.96.34.52:15672/
guest   guest
```

4 MongoDB集群

```
1   拉取mongoDB镜像

docker pull mongo

2   创建本地挂在目录--建议先删除

mkdir -p /app/docker/mongo1/db    #创建挂载的db目录
mkdir -p /app/docker/mongo2/db    #创建挂载的db目录
mkdir -p /app/docker/mongo3/db    #创建挂载的db目录

3   启动多个MongoDB实例

#第一台:
docker run --name mongo-server1 -p 30001:27017 --restart=always -v
/app/docker/mongo1/db:/data/db -v /etc/localtime:/etc/localtime -d mongo  --
replSet "rs0"  --bind_ip_all
#第二台:
docker run --name mongo-server2 -p 30002:27018 --restart=always -v
/app/docker/mongo2/db:/data/db -v /etc/localtime:/etc/localtime -d mongo  --
replSet "rs0"  --bind_ip_all
#第三台:
```

```
docker run --name mongo-server3 -p 30003:27019 --restart=always -v
/app/docker/mongo3/db:/data/db -v /etc/localtime:/etc/localtime -d mongo  --
replSet "rs0"  --bind_ip_all

4  搭建集群

#进入容器 进入主的容器
docker exec -it mongo-server1 bash
#连接客户端
mongo

rs.initiate()

rs.add("localhost:30002")

rs.addArb("localhost:30003")

5 查询使用

使用工具 NoSQL Manager for MongoDB Freeware

连接：39.96.34.52:30001
```

3.2.2 项目初始化

https://github.com/dotnetcore/CAP/blob/master/README.zh-cn.md

1 Nuget引用

```
PM> Install-Package DotNetCore.CAP
PM> Install-Package DotNetCore.CAP.Dashboard   #consul
```

CAP 支持 Kafka、RabbitMQ消息队列，你可以按需选择下面的包进行安装：

```
PM> Install-Package DotNetCore.CAP.Kafka
PM> Install-Package DotNetCore.CAP.RabbitMQ
```

CAP 提供了 Sql Server, MySql, PostgreSQL， MongoDB 的扩展作为数据库存储：

```
// 按需选择安装你正在使用的数据库
PM> Install-Package DotNetCore.CAP.SqlServer
PM> Install-Package DotNetCore.CAP.MySql
PM> Install-Package DotNetCore.CAP.PostgreSql
PM> Install-Package DotNetCore.CAP.MongoDB
```

2 代码配置

生产者-ConfigureService：

```
string conn = this.Configuration.GetConnectionString("UserServiceConnection");
            string rabbitMQ =
this.Configuration.GetConnectionString("RabbitMQ");
            services.AddCap(x =>
            {
```

```
                x.UseSqlServer(conn);
                x.UseRabbitMQ(rabbitMQ);
                x.FailedRetryCount = 5;
                x.FailedThresholdCallback = failed =>
                {
                    var logger =
failed.ServiceProvider.GetService<ILogger<Startup>>();
                    logger.LogError($@"MessageType {failed.MessageType} 失败了,
重试了 {x.FailedRetryCount} 次,
                        消息名称: {failed.Message.GetName()}");//do anything
                };

                #region 注册Consul可视化
                x.UseDashboard();
                DiscoveryOptions discoveryOptions = new DiscoveryOptions();
                this.Configuration.Bind(discoveryOptions);
                x.UseDiscovery(d =>
                {
                    d.DiscoveryServerHostName =
discoveryOptions.DiscoveryServerHostName;
                    d.DiscoveryServerPort =
discoveryOptions.DiscoveryServerPort;
                    d.CurrentNodeHostName =
discoveryOptions.CurrentNodeHostName;
                    d.CurrentNodePort = discoveryOptions.CurrentNodePort;
                    d.NodeId = discoveryOptions.NodeId;
                    d.NodeName = discoveryOptions.NodeName;
                    d.MatchPath = discoveryOptions.MatchPath;
                });
                #endregion
            });
```

对应配置文件:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RabbitMQ": "39.96.34.52",
    "UserServiceConnection": "Server=ElevenPC;Database=UserService;User
Id=sa;Password=PasswOrd;"
  },
  "DiscoveryOptions": {
    "DiscoveryServerHostName": "localhost",
    "DiscoveryServerPort": 8500,
    "CurrentNodeHostName": "localhost",
    "CurrentNodePort": 9999,
    "NodeId": "Zhaoxi.DistributedTransaction.UserService",
    "NodeName": "Zhaoxi.DistributedTransaction.UserService",
    "MatchPath": "\"\"/cap\"\"" //默认路径
  }
```

```
}
```

3.2.3 生产者代码

任选控制器---注入ICapPublisher---直接Publish（同步/异步）---之前是body，3.0之后可以header了---

消息划分为 Header 和 Body 来进行传输。

Body 中的数据为用户发送的原始消息内容，也就是调用 Publish 方法发送的内容，我们不进行任何包装仅仅是序列化后传递到消息队列。

在 Header 中，我们需要传递一些额外信息以便于CAP在收到消息时能够提取到关键特征进行操作。

以下是在异构系统中，需要在发消息的时候向消息的Header 中写入的内容（好像已经默认添加了，再次写入id会报错）：

键	类型	说明
cap-msg-id	string	消息Id， 由雪花算法生成，也可以是 guid
cap-msg-name	string	消息名称，即 Topic 名字
cap-msg-type	string	消息的类型, 即 typeof(T).FullName (非必须)
cap-senttime	stringg	发送的时间 (非必须)

注意：

1 dotnet run --urls="http://*:9999" #端口号需要跟consul注册一致

2 x.FailedRetryCount = 5;
x.FailedRetryInterval = 60;

写入队列的重试次数，先写入数据库---关闭队列---还是能成功---交给重试机制--不行再发邮件人工介入

3 消息数据清理

数据库消息表中具有一个 ExpiresAt 字段表示消息的过期时间，当消息发送成功或者消费成功后，CAP 会将消息状态为 Successed 的 ExpiresAt 设置为 1天 后过期，会将消息状态为 Failed 的 ExpiresAt 设置为 15天 后过期。

CAP 默认情况下会每隔一个小时将消息表的数据进行清理删除，避免数据量过多导致性能的降低。清理规则为 ExpiresAt 不为空并且小于当前时间的数据。 也就是说状态为Failed的消息（正常情况他们已经被重试了 50 次），如果你15天没有人工介入处理，同样会被清理掉。

3.2.4 消费者代码

得先启动一次，队列才能写入数据，否则会丢失

1 配置文件信息一致，只修改下端口号

2 控制器里面的Action，注意NonAction 表明不是Action

CapSubscribe前面写入的名称

Group表示多几个观察者，对应RabbitMQ的队列，默认是“xxx”

```
 [NonAction]
        [CapSubscribe("RabbitMQ.SQLServer.UserService")]
        public void Subscriber(Person person)
        {
            Console.WriteLine($@"{DateTime.Now} Subscriber invoked, Info:
{person}");
            throw new Exception("Subscriber failed");
        }

        [NonAction]
        [CapSubscribe("RabbitMQ.SQLServer.UserService", Group = "group.test2")]
        public void Subscriber2(Person p, [FromCap] CapHeader header)
        {
```

3 进入处理动作之前，就已经把数据结构和任务从队列拿到数据库里了，这里的动作，仅仅是用来重试执行任务的

header为空，是因为有的地方写入的时候为空了

1 dotnet run --urls="http://*:8888"

3.2.5 生产者+消费者

跟消费者一样，然后再来个写入流程就行

1 dotnet run --urls="http://*:7777"

一方面get，一方面set

靠名字+group区分

3.2.6 要点

1 传输

支持的运输器

CAP 支持以下几种运输方式：

- RabbitMQ
- Kafka
- Azure Service Bus
- Amazon SQS
- In-Memory Queue

怎么选择运输器

	RabbitMQ	Kafka	Azure Service Bus	In-Memory
定位	可靠消息传输	实时数据处理	云	内存型，测试
分布式	✔	✔	✔	✕
持久化	✔	✔	✔	✕
性能	Medium	High	Medium	High

课程选用的RabbitMQ,基于docker构建

2 存储(自定义名称)

CAP 需要使用具有持久化功能的存储介质来存储事件消息，例如通过数据库或者其他NoSql设施。CAP 使用这种方式来应对一切环境或者网络异常导致消息丢失的情况，消息的可靠性是分布式事务的基石，所以在任何情况下消息都不能丢失。

支持：

- SQL Server
- MySQL
- PostgreSql
- MongoDB
- [In-Memory]

持久化 发送前

在消息进入到消息队列之前，CAP使用本地数据库表对消息进行持久化，这样可以保证当消息队列出现异常或者网络错误时候消息是没有丢失的。

为了保证这种机制的可靠性，CAP使用和业务代码相同的数据库事务来保证业务操作和CAP的消息在持久化的过程中是强一致的。也就是说在进行消息持久化的过程中，任何一方发生异常情况数据库都会进行回滚操作。

发送后

消息进入到消息队列之后，CAP会启动消息队列的持久化功能，我们需要说明一下在 RabbitMQ 和 Kafka 中CAP的消息是如何持久化的。

针对于 RabbitMQ 中的消息持久化，CAP 使用的是具有消息持久化功能的消费者队列，但是这里面可能有例外情况，参加 2.2.1 章节。

由于 Kafka 天生设计的就是使用文件进行的消息持久化，在所以在消息进入到Kafka之后，Kafka会保证消息能够正确被持久化而不丢失。

消息存储

在 CAP 启动后，会向持久化介质中生成两个表，默认情况下名称为：`Cap.Published` `Cap.Received`。

存储格式

Published 表结构：

NAME	DESCRIPTION	TYPE
Id	Message Id	int
Version	Message Version	string
Name	Topic Name	string
Content	Json Content	string
Added	Added Time	DateTime
ExpiresAt	Expire time	DateTime
Retries	Retry times	int
StatusName	Status Name	string

Received 表结构：

NAME	DESCRIPTION	TYPE
Id	Message Id	int
Version	Message Version	string
Name	Topic Name	string
Group	Group Name	string
Content	Json Content	string
Added	Added Time	DateTime
ExpiresAt	Expire time	DateTime
Retries	Retry times	int
StatusName	Status Name	string

自定义表名称

你可以通过重写 `IStorageInitializer` 接口获取表名称的方法来做到这一点

示例代码：

```
public class MyTableInitializer : SqlServerStorageInitializer
{
    public override string GetPublishedTableName()
    {
        //你的 发送消息表 名称
    }

    public override string GetReceivedTableName()
    {
        //你的 接收消息表 名称
    }
}
```

然后将你的实现注册到容器中

```
services.AddSingleton<IStorageInitializer, MyTableInitializer>();
```

3 MongoDB 存储模式

MongoDB 是一个跨平台的面向文档型的数据库程序，它被归为 NOSQL 数据库，CAP 从 2.3 版本开始支持 MongoDB 作为消息存储。

MongoDB 从 4.0 版本开始支持 ACID 事务，所以 CAP 也只支持 4.0 以上的 MongoDB，并且 MongoDB 需要部署为集群，因为 MongoDB 的 ACID 事务需要集群才可以使用。

有关开发环境如何快速搭建 MongoDB 4.0+ 集群，

配置

要使用 MongoDB 存储，你需要从 NuGet 安装以下扩展包：

```
Install-Package DotNetCore.CAP.MongoDB
```

然后，你可以在 `Startup.cs` 的 `ConfigureServices` 方法中添加基于内存的配置项。

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    services.AddCap(x =>
    {
        x.UseMongoDB(opt=>{
            //MongoDBOptions
        });
        // x.UseXXX ...
    });
}
```

配置项

NAME	DESCRIPTION	TYPE	DEFAULT
DatabaseName	数据库名称	string	cap
DatabaseConnection	数据库连接字符串	string	mongodb://localhost:27017
ReceivedCollection	接收消息集合名称	string	cap.received
PublishedCollection	发送消息集合名称	string	cap.published

注册过程是一样的，生产--消费，连上mongoDB，看本地信息

MongoDB---作为接受--发布完整模式，承接

3.2.7 Dashboard监控

1 nuget DotNetCore.CAP.Dashboard

```
2  x.UseDashboard();
                DiscoveryOptions discoveryOptions = new DiscoveryOptions();
                this.Configuration.Bind(discoveryOptions);
                x.UseDiscovery(d =>
                {
                    d.DiscoveryServerHostName =
discoveryOptions.DiscoveryServerHostName;
                    d.DiscoveryServerPort =
discoveryOptions.DiscoveryServerPort;
                    d.CurrentNodeHostName =
discoveryOptions.CurrentNodeHostName;
                    d.CurrentNodePort = discoveryOptions.CurrentNodePort;
                    d.NodeId = discoveryOptions.NodeId;
                    d.NodeName = discoveryOptions.NodeName;
                    d.MatchPath = discoveryOptions.MatchPath;
                });
```