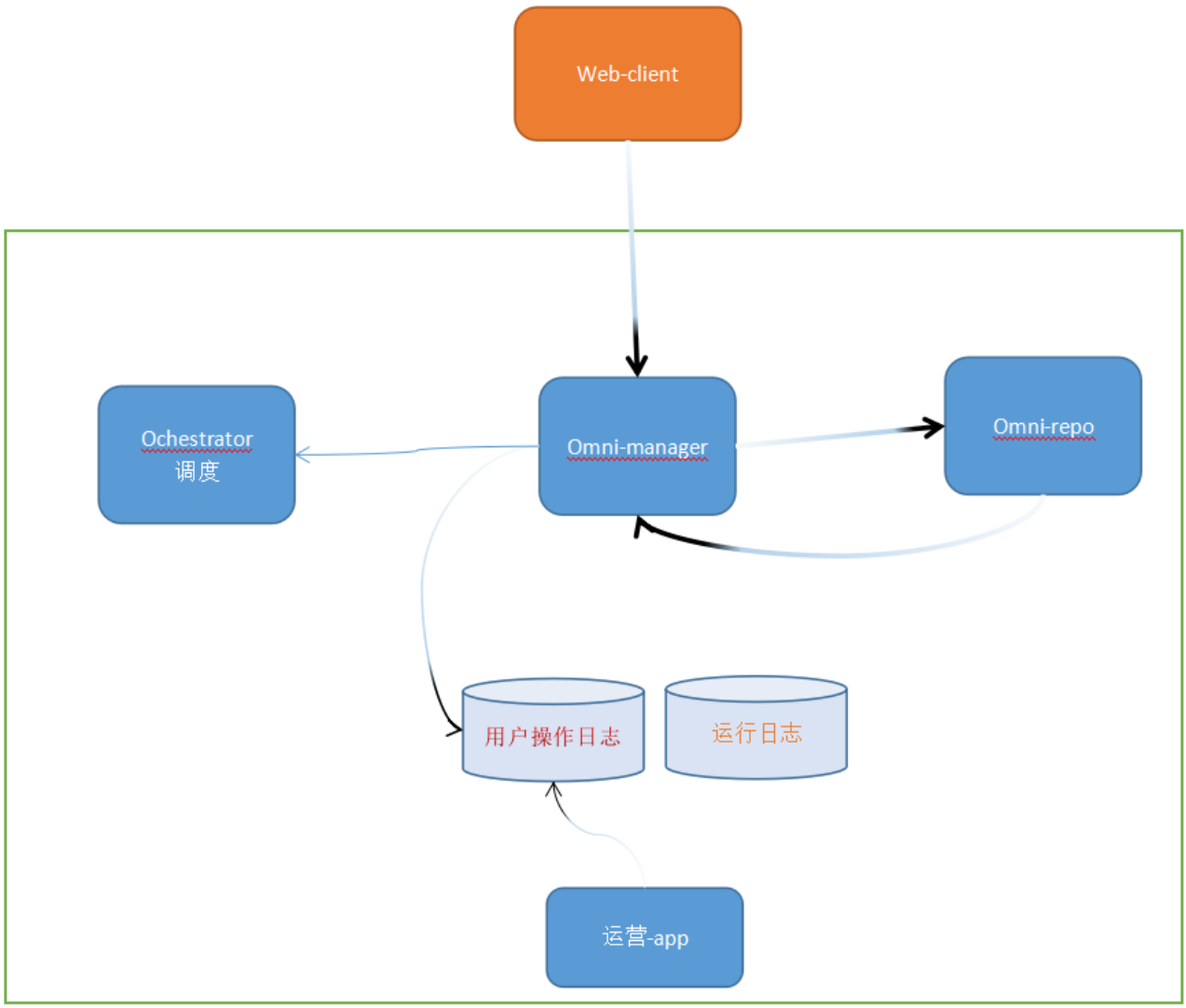
CloudEvents 消息组件

1. 使用消息中间件的必要性

没有使用消息中间件时候的 omni构建系统部分app 通信示意图

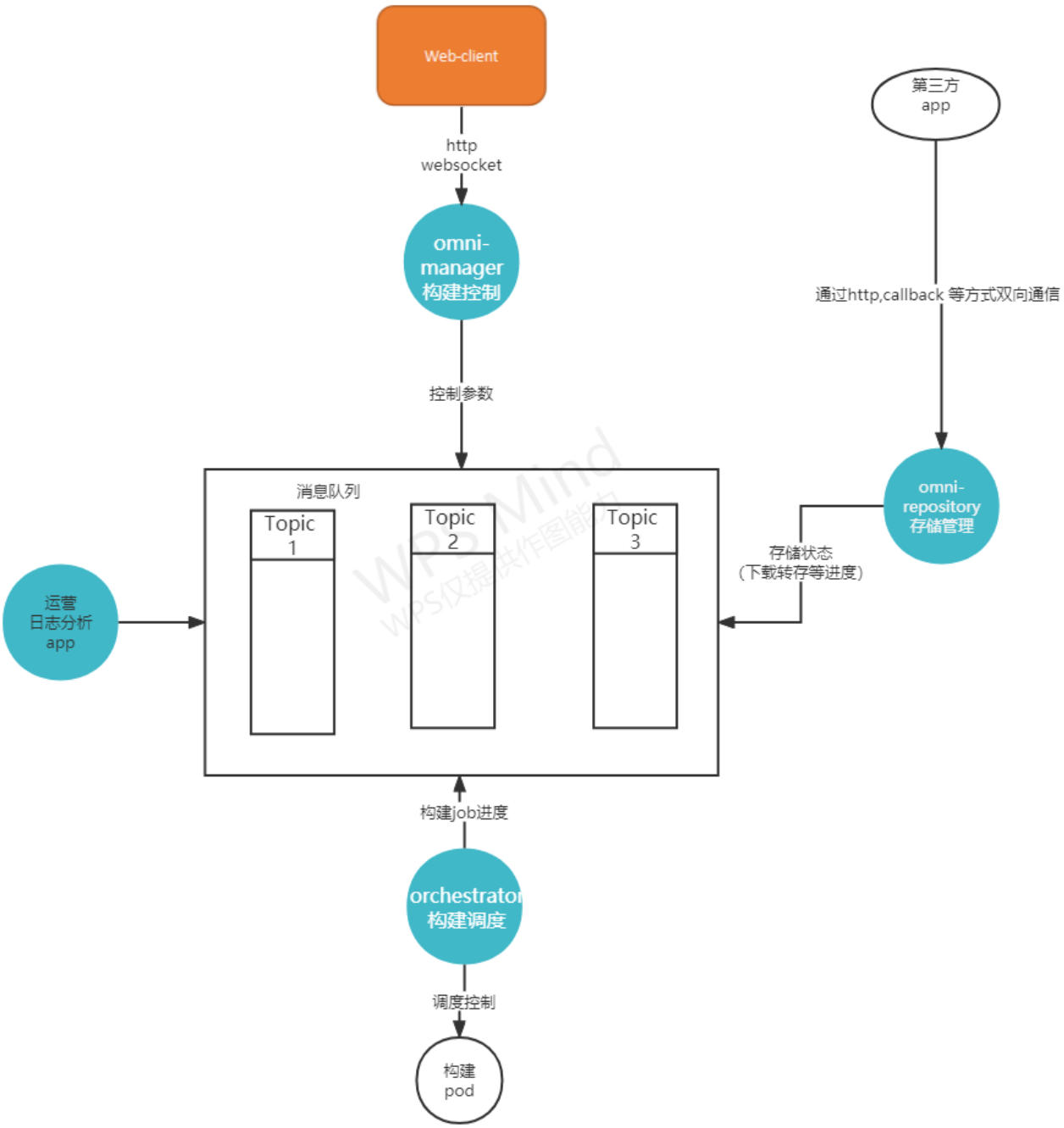


（图1 .现在没有使用消息中间件时候的部分app通信示意图）

缺点：

1. 各个接口都是直接调用和回调。很难水平扩展部署，
2. 耦合度很高，开发工作越来越麻烦。
3. 要跟运营的开发人员对接数据格式和接口等

2. 使用消息中间件改造后的部分app结构：



（图2 .改造后的部分app通信示意图）

优点：

1. 耦合度大大降低，可以做成微服务
2. 开发复杂度降低。部署和调试也容易。
3. 分工明确，业务程序只管处理业务。减少专门为运营团队制造的日志输出

缺点： 非常依赖消息中间件的稳定性。

3. 消息格式以及 SDK的选用

参考 https://github.com/cloudevents/spec/blob/v1.0/spec.md#event-format 的消息格式。

选用 cloudEvent 的原因是

1. 他拥有多种语言的sdk（java,golang,python,rust...）支持，便于全部门的项目对接。
2. 支持多种消息队列产品(kafka,rabbitMQ,nsq...).换句话说，即便将来要换底层消息中间件，我们业务层的代码改动也很小。
3. 消息格式比较通用扩展性好。社区认可度高。

cloudEvents 的字段说明。

JSON

```
{
    "specversion": "1.0",    （选填）
    "type": "com.github.pull.create",
    "source": "github.com/omnibuildplatform/omni-manager/models",
    "subject": "123",（选填）
    "id": "A234-1234-1234",（选填）
    "time": "2018-04-05T17:31:00Z",  （选填）//RFC 3339 时间格式。
    "datacontenttype": "text/json",    （必填）
    "data": {}  （选填）
  }

//--------------------说明----------------------------
specversion ：（选填）
//当前默认版本1.0    选填0.3版本。 强填其他内容会报错
type ：  （必填 ）
 //消息事件的业务类型 。
 // 比如omni-manager 发出一个download 下载。 type可以为:
 // download.http   download.https.  download.ftp  ...消费者收到后就可以用switch进行分类处理
 //事件接收者可以 根据这个type 调用相应的处理模块
 //type 和topic 的区别:
 //topic区分的是事件的类型。 type区分的是同一类事件中细分的业务类别。如上例子所述。
 //都是 download 事件，但不同的type 需要用不同的func进行处理。
 //从技术上来说， 也可以为每一种下载方式定义一个topic事件。 比如topic为 download.http。但事件的topic就会很多。但其实这些细分下来的topic他们关联性很强。

source:  （选填）
//事件具体从哪个模块发出的， 便于追踪事件的来源。 跟业务逻辑没有关系。纯粹为了维护方便。
subject:  （选填）
//这个事件的标题。 不填也不影响事件正常收发
id ：（选填）
// id 可以是任意字符串，比如uuid  ，‘1’ ‘2’‘3’。。。。不填也不会报错。但事件没有编号不便于定位排错
time ：（选填）
// 不填的话  cloudevents会自动帮你填上  time.now()
datacontenttype ：  （必填） 涉及到数据的编码和解码。 必须明确指出编码方式
// 包括以下数据格式
```

```
// text/plain
// application/json
// text/json
// application/xml
// application/cloudevents+json
// application/cloudevents-batch+json
data ：(技术上是 选填。)
// data就是payload 。我们自定义的数据格式。   一般来说会根据上面 type的不同而不同。
// 消费者可以使用type来灵活对待 data内容。


```

下面以 Omni-manager 项目发送一条 “构建命令”作为消息示例：

JSON

```
 {
    "specversion": "1.0",
    "type": "buildFromIso",  // 另外有一种  buildFromRelease 的构建命令
    "source": "github.com/omnibuildplatform/omni-manager/models",
    "subject": "张三_x86_64_openeuler22.03构建",
    "id": "A234-1234-1234", //
    "time": "2022-04-05T17:31:00Z",
    "datacontenttype": "application/json",
    "data": {
      "baseImageID": "92",
      "desc": "使用OpenEuler 22-04构建任务",
      "kickStartContent": " keyboard --vckeymap=us --xlayouts='us'   timezone Asia/Shanghai --isUtc --nontp  %packages --multilib @core glibc.i686  gcc gdb make %end %post grub2-set-default 1 %end",
      "kickStartName": "带有logo 和安装分区的ks",
      "label": "2022级学生版OpenEuler "
    }
  }
```

使用cloudevent 对上面的数据进行封装发布演示

Go

```
package cloud

import (
    "context"
    "fmt"
    "testing"
    "time"

    "github.com/Shopify/sarama"
```

```
    "github.com/cloudevents/sdk-go/protocol/kafka_sarama/v2"
    cloudevents "github.com/cloudevents/sdk-go/v2"
)
const (
// 消息的主题名
    TopciName = "omni-manager-downloadurl"
    //消费组的名字。
    GroupID   = "mygroup"
)

var (
    saramaConfig *sarama.Config
    brokers      []string
    receiver     *kafka_sarama.Consumer
    err          error
    clientItem   client.Client
)
func TestProduce(t *testing.T) {
    saramaConfig = sarama.NewConfig()
    saramaConfig.Version = sarama.V2_8_1_0
    saramaConfig.Producer.RequiredAcks = 1
    brokers = []string{"192.168.1.193:9092"}
    sender, err := kafka_sarama.NewSender(brokers, saramaConfig, TopciName)
    if err != nil {
        t.Fatalf("failed to create protocol: %s \n", err.Error())
    }
    defer sender.Close(context.Background())
//-上面部分是使用 kafka 的sdk做驱动。下面则使用cloudEvent封装出通用模板的事件进行后续发送
    cloudEventClient, err := cloudevents.NewClient(sender,
cloudevents.WithTimeNow(), cloudevents.WithUUIDs())
    if err != nil {
        t.Fatalf("failed to create client, %v \n", err)
    }

    for i := 0; i < 10; i++ {
        e := cloudevents.NewEvent()
        e.SetSpecVersion(cloudevents.VersionV1)
        e.SetID(fmt.Sprintf("%d", i))
        e.SetType("buildFromIso")
        e.SetTime(time.Now())
        e.SetSubject("张三_x86_64_openeuler22.03构建")
        e.SetSource("github.com/omnibuildplatform/omni-manager/models")
        _ = e.SetData(cloudevents.ApplicationJSON, map[string]interface{}{
            "id":               i,
            "baseImageID":      "92",
            "desc":             "使用OpenEuler 22-04构建任务",
            "kickStartContent": " keyboard --vckeymap=us --xlayouts='us'
```

```
    timezone Asia/Shanghai --isUtc --nontp  %packages --multilib @core glibc.i686
    gcc gdb make %end %post grub2-set-default 1 %end",
            "kickStartName":    "带有logo 和安装分区的ks",
            "label":            "2022级学生版OpenEuler ",
        })
        err =
    cloudEventClient.Send(kafka_sarama.WithMessageKey(context.Background(),
    sarama.StringEncoder(e.ID())), e)
        if err != nil {
            t.Fatalf("failed to Send msg ,error: %v \n", err)
        } else {
            t.Logf("  Send msg success : %v \n", i)
        }

    }

}
```

消息 消费者 示例代码

Go

```
package cloud

import (
    "context"
    "fmt"
    "log"

    "github.com/Shopify/sarama"
    "github.com/cloudevents/sdk-go/protocol/kafka_sarama/v2"
    cloudevents "github.com/cloudevents/sdk-go/v2"
    "github.com/cloudevents/sdk-go/v2/client"
)

func SingleConsumer(groupName, topicName string) {
   fmt.Printf("消费组:(%s) 。接收消息主题TOPIC :(%s) \n", groupName, topicName)
    receiver, err = kafka_sarama.NewConsumer(brokers, saramaConfig, groupName,
    topicName)
    if err != nil {
        log.Fatalf("failed to create protocol: %s", err.Error())
    }
    defer receiver.Close(context.Background())
    clientItem, err = cloudevents.NewClient(receiver,
    client.WithPollGoroutines(1))
    if err != nil {
        log.Fatalf("failed to create client, %v", err)
```

```
        log.Fatalf("failed to create client, %v", err)
    }
    //注册一个 消息到达时候的handler 函数
    err = clientItem.StartReceiver(context.Background(), onMessageReceive)
    if err != nil {
        log.Fatalf("failed to start receiver: %s", err)
    } else {
        log.Printf("receiver started\n")
    }

}
// 接收此 topic 消息后的处理函数
func onMessageReceive(ctx context.Context, event cloudevents.Event) {
    fmt.Printf(" 收到的数据: %v     \n", string(event.ID()))
    eventType := event.Type()
    switch eventType {
    case "buildFromIso":
    case "buildFromRelease":
    default:
    }

}
```

根据Omni-platform的业务特征，本次选型调研了以下3种当前最为流行的消息队列产品：

Kafka,rabbitMQ和NSQ

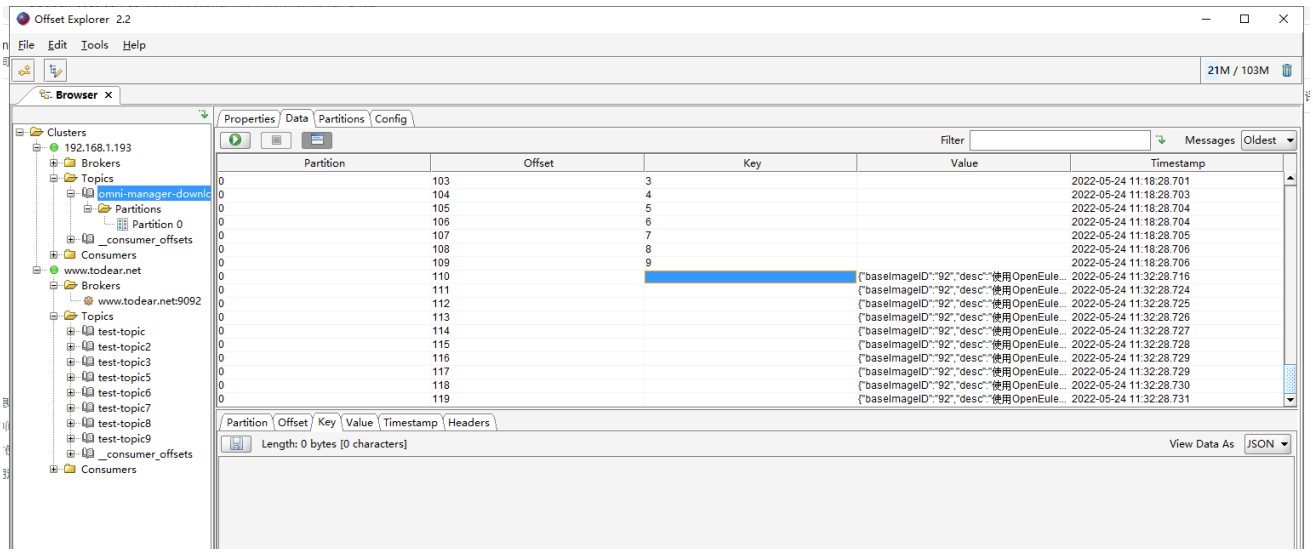
产品名	开发语言	与我们项目的相关优缺点
Kafka	Java	1. kafka能保证各个topic的消息是严格遵照顺序被消费。而我们的构建系统等对构建步骤是有顺序要求的。 2. Kafka的消息被存储长期记录在队列中，方便消息订阅者根据业务自己反复使用。 3. 使用人数众多，文档较多，工程师也多。
rabbitMQ	Java	1. 消息的路由上灵活性更高，能让消息消费者筛选到自己感兴趣的消息。但我们的业务并不需要非常灵活的自定义消息筛选机制。 2. 可以设置消息的TTL， 超时的消息会被扔掉。以减少超时无用的消息影响。但我们的业务对实时性要求不苛刻，不存在超时无用的消息，即便短时间无法处理的消息 也应该保留在消息队列中以便后续继续处理。
NSQ	Golang	1. 使用golang开发，我们团队拥有大量golang同事。如果需要定制消息中心可以比较容易。但估计不会走到必须要定制改造消息中心的程度。 2. 相对轻量级，

4. 相关约定

1	Topic 命名约定	以连接符 ‘-’ 作为间隔。 不混用 ‘.’ 和 '_' 否则容易重名。 比如topic名字： a.b_c 在kafka中等于 a_b_c 。如果混用会导致重名覆盖等意外。 建议以项目的名字+业务类型 omni-repository-download
2	消费Group的命名约定	
3		
4		

相关工具

Offset explorer （kafka的管理客户端）




Ready [Messages = 120] [38,280 Bytes] [101 ms]
Max Messages (per partition)
5550