# Dapr概述

## 概念

Dapr 是一个可移植的事件驱动运行时，使任何开发人员都可以轻松构建在云和边缘上运行的弹性、无状态和有状态的应用程序，并包含语言和开发人员框架的多样性。

Dapr 将构建微服务应用的 最佳实践 设计成开放、独立和模块化的方式，让你能够使用任意的开发语言和框架构建可移植的应用程序。 每个构建块都是完全独立的，您可以采用其中一个、多个或全部来构建你的应用。

Dapr是和平台无关的，这意味着您可以在本地、Kubernetes 集群或者其它集成 Dapr 的托管环境中运行应用程序。 这使得您能够在云平台和边缘计算中运行微服务应用。

Dapr可以使用任何语言、框架轻松构建微服务应用，运行在任何地方。

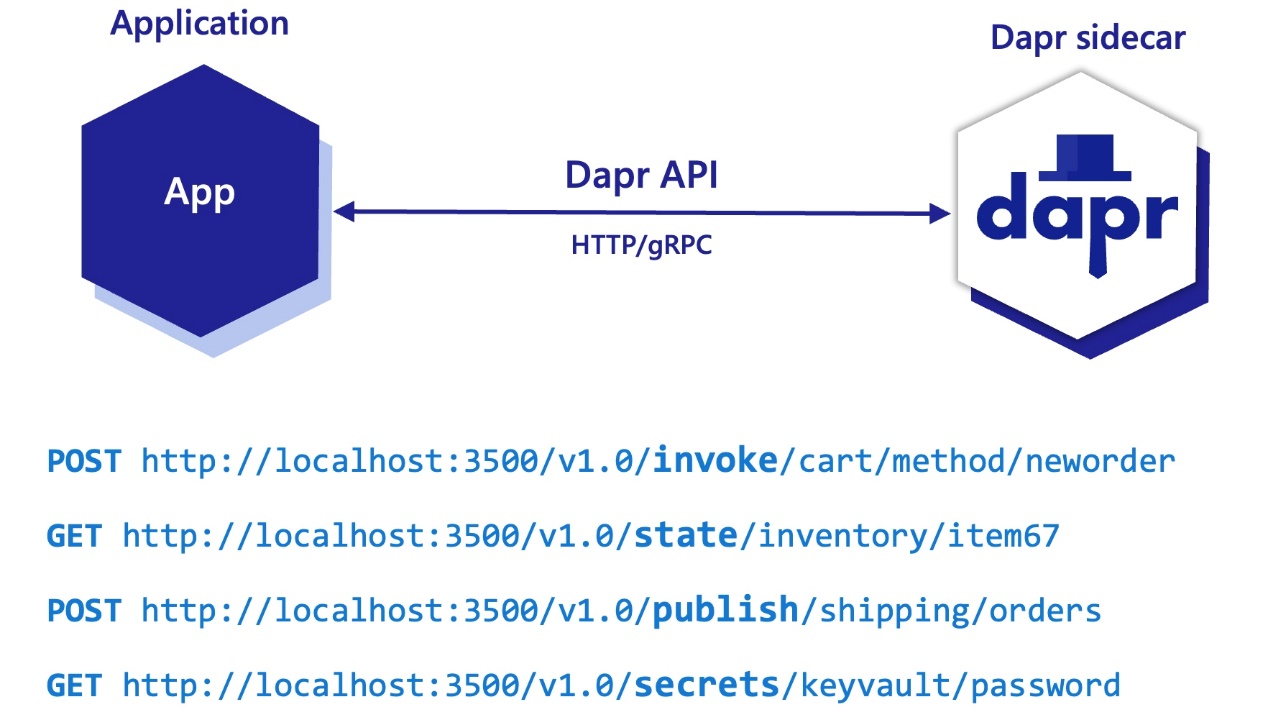
## 构建块

Dapr提供了一些常用功能的最佳实践，开发人员可以使用标准模式进行微服务应用的构建，并部署到任意环境中。 Dapr 通过提供分布式构建块来实现此目的。

|  |  |
| --- | --- |
| [**服务调用**](https://docs.dapr.io/zh-hans/developing-applications/building-blocks/service-invocation/service-invocation-overview/) | 跨服务调用允许进行远程方法调用(包括重试)，不管处于任何位置，只需该服务托管于受支持的环境即可。 |
| [**状态管理**](https://docs.dapr.io/zh-hans/developing-applications/building-blocks/state-management/state-management-overview/) | 独立的状态管理，使用键/值对作为存储机制，可以轻松的使长时运行、高可用的有状态服务和无状态服务共同运行在您的应用程序中。 状态存储是可插拔的，目前支持使用Azure CosmosDB、 Azure SQL Server、 PostgreSQL,、AWS DynamoDB、Redis 作为状态存储介质。 |
| [**发布订阅**](https://docs.dapr.io/zh-hans/developing-applications/building-blocks/pubsub/pubsub-overview/) | 发布事件和订阅主题。 |
| [**资源绑定**](https://docs.dapr.io/zh-hans/developing-applications/building-blocks/bindings/bindings-overview/) | Dapr的Bindings是建立在事件驱动架构的基础之上的。通过建立触发器与资源的绑定，可以从任何外部源（例如数据库，队列，文件系统等）接收和发送事件，而无需借助消息队列，即可实现灵活的业务场景。 |
| [**Actors**](https://docs.dapr.io/zh-hans/developing-applications/building-blocks/actors/actors-overview/) | Actor模型 = 状态 + 行为 + 消息。一个应用/服务由多个Actor组成，每个Actor都是一个独立的运行单元，拥有隔离的运行空间，在隔离的空间内，其有独立的状态和行为，不被外界干预，Actor之间通过消息进行交互，而同一时刻，每个Actor只能被单个线程执行，这样既有效避免了数据共享和并发问题，又确保了应用的伸缩性。 Dapr 在Actor模式中提供了很多功能，包括并发，状态管理，用于 actor 激活/停用的生命周期管理，以及唤醒 actor 的计时器和提醒器。 |
| [**可观测性**](https://docs.dapr.io/zh-hans/concepts/observability-concept/) | Dapr记录指标，日志，链路以调试和监视Dapr和用户应用的运行状况。 Dapr支持分布式跟踪，其使用W3C跟踪上下文标准和开放式遥测技术，可以轻松地诊断在生产环境中服务间的网络调用，并发送到不同的监视工具。 |
| [**Secrets**](https://docs.dapr.io/developing-applications/building-blocks/secrets/secrets-overview/) | Dapr 提供了密钥管理，支持与公有云和本地的Secret存储集成，以供应用检索使用。 |

## Sidecar 架构

Dapr以 sidecar 架构的方式公开其API，可以是容器，也可以是进程，不需要应用代码包含任何 Dapr 运行时代码。 这使得 Dapr 与其他运行时的集成变得容易，在应用逻辑层面做了隔离处理，提高了可扩展性。



## 托管环境

Dapr可以托管在多个环境中，包括在Windows/Linux/macOS机器上自托管和Kubernetes。

### 自托管

[自托管模式](https://docs.dapr.io/zh-hans/operations/hosting/self-hosted/self-hosted-overview/) 下，Dapr 运行一个单独的 sidecar 程序，在您的服务代码中可以通过 HTTP 或 gRPC 调用它。每个运行的服务都有一个 Dapr 运行时进程 (或 sidecar) ，配置为使用状态存储， pub/sub，绑定组件和其他构建块。

### Kubernetes 托管

在托管在容器环境中（如 Kubernetes），Dapr 作为 sidecar 容器运行，和应用程序容器在同一个 pod 中。

## 开发者语言 SDK 和框架

### Dapr SDKs

为了让不同语言使用 Dapr 更加自然，它还包含了 [语言特定的 SDK](https://docs.dapr.io/zh-hans/developing-applications/sdks/)：

* C++
* Go
* Java
* JavaScript
* Python
* Rust
* .NET
* PHP

这些 SDK 通过特定语言 API 来暴露 Dapr 构建块的功能，而不是调用 http/gRPC API。 这使您能够以自己选择的语言编写无状态和有状态功能和 actors 的组合。 由于这些 SDK 共享 Dapr 运行时，因此您可以获得跨语言 actor 和功能支持。

### 开发框架

|  |  |  |
| --- | --- | --- |
| 语言 | 框架 | 说明 |
| [.NET](https://docs.dapr.io/zh-hans/developing-applications/sdks/dotnet/) | [ASP.NET](https://docs.dapr.io/zh-hans/developing-applications/sdks/dotnet/dotnet-server/dotnet-aspnet/) | 带来状态路由控制器，从而完成来自其他应用的 发布/订阅 构建块。 也可以利用 [ASP.NET Core gRPC 服务](https://docs.microsoft.com/en-us/aspnet/core/grpc/)。 |
| [Java](https://github.com/dapr/java-sdk) | [Spring Boot](https://spring.io/) |  |
| [Python](https://docs.dapr.io/zh-hans/developing-applications/sdks/python/) | [Flask](https://docs.dapr.io/zh-hans/developing-applications/sdks/python/python-sdk-extensions/python-flask/) |  |
| [Javascript](https://github.com/dapr/js-sdk) | [Express](http://expressjs.com/) |  |
| PHP |  | 可以使用 Apache, Nginx,或 Caddyserver 进行托管 |

# 环境搭建

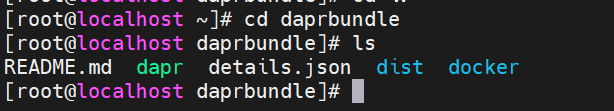
## 在线安装



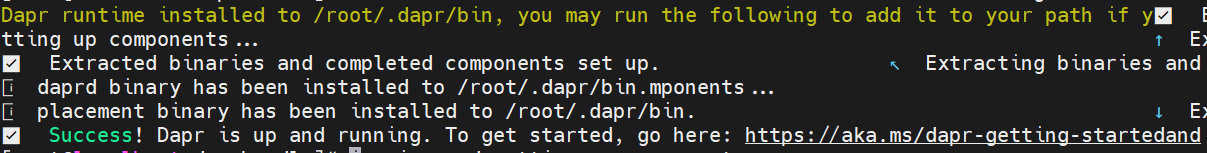
## 离线安装

[GitHub: Let’s build from here · GitHub](https://github.com/dapr)去这个网站下载离线版安装包（daprbundle）

下载解压后，进入daprbundle目录：

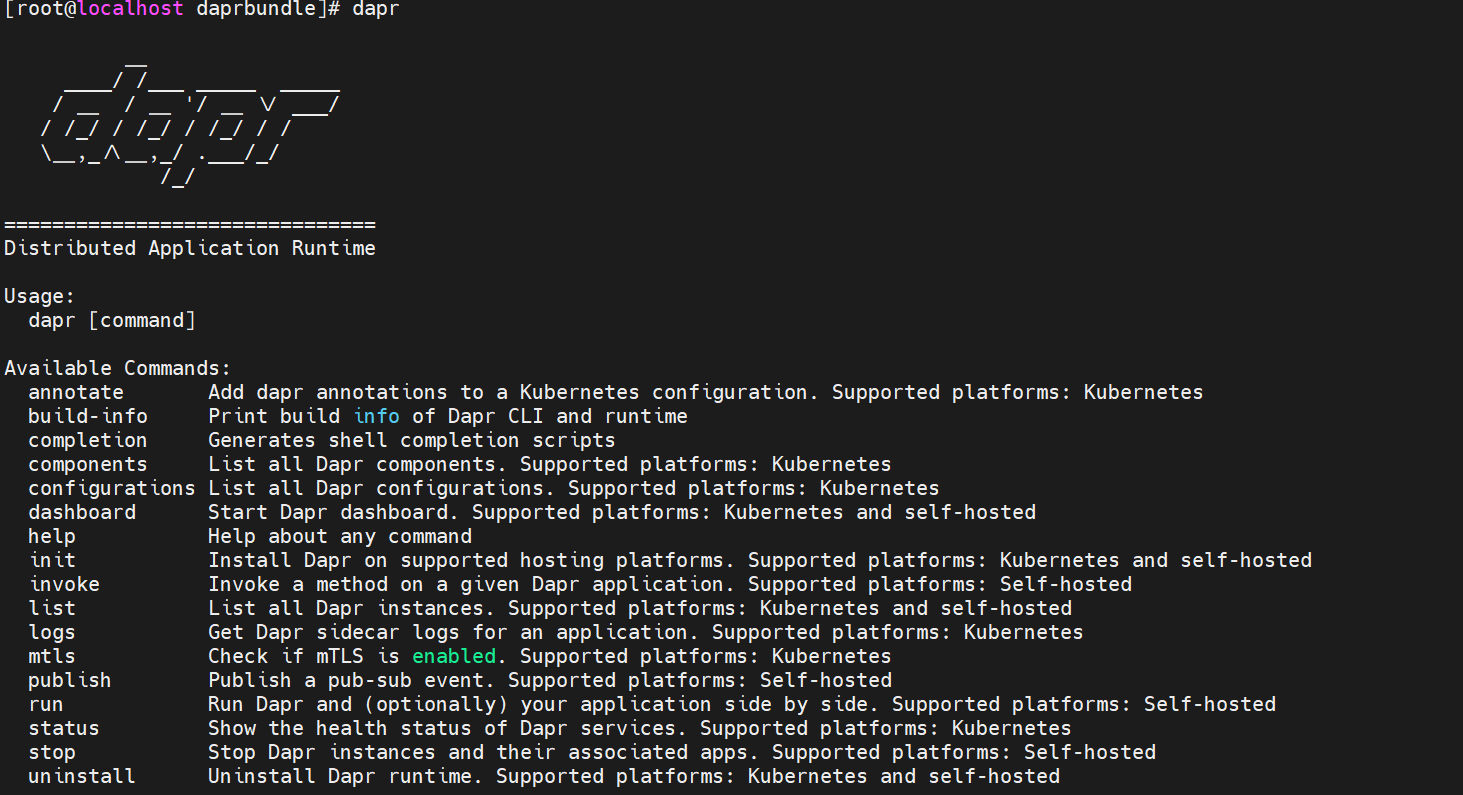


然后执行dapr init --slim --from-dir .命令进行初始化，效果如下：



看到success说明安装成功

执行dapr命令，验证安装：



## 常用命令

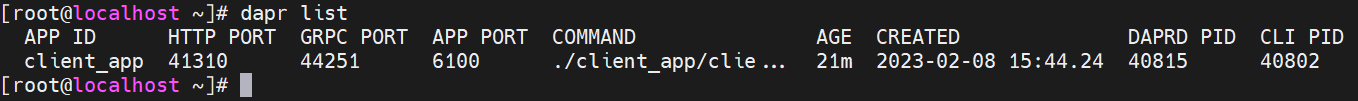
启动命令：

dapr run --app-id myapp --dapr-http-port 3500

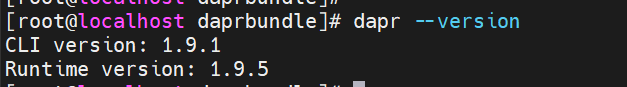
停止dapr：

dapr stop --app-id myapp

查看dapr是否启动：dapr list



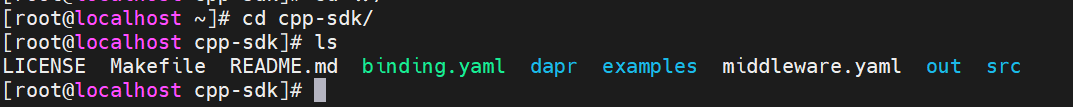
查看版本信息，执行如下命令：dapr –-version



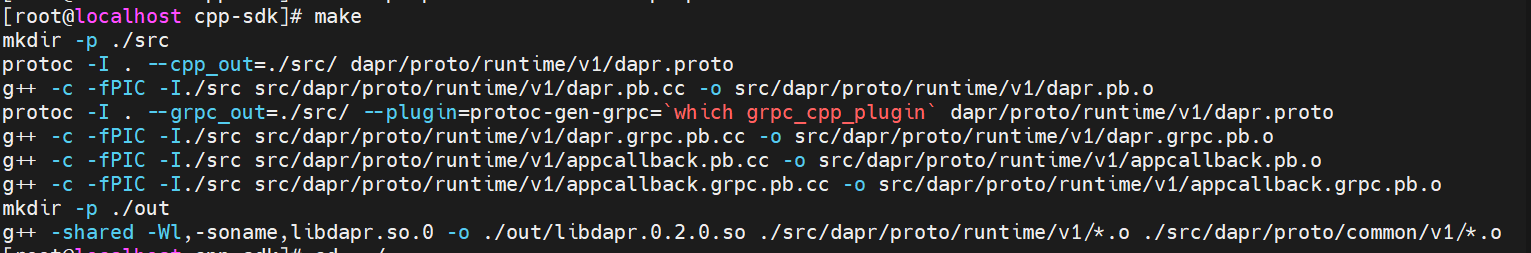
# 实现步骤

下载c++SDK,并且编译proto文件，并生成相应的so文件

1. 先进入cpp-sdk目录

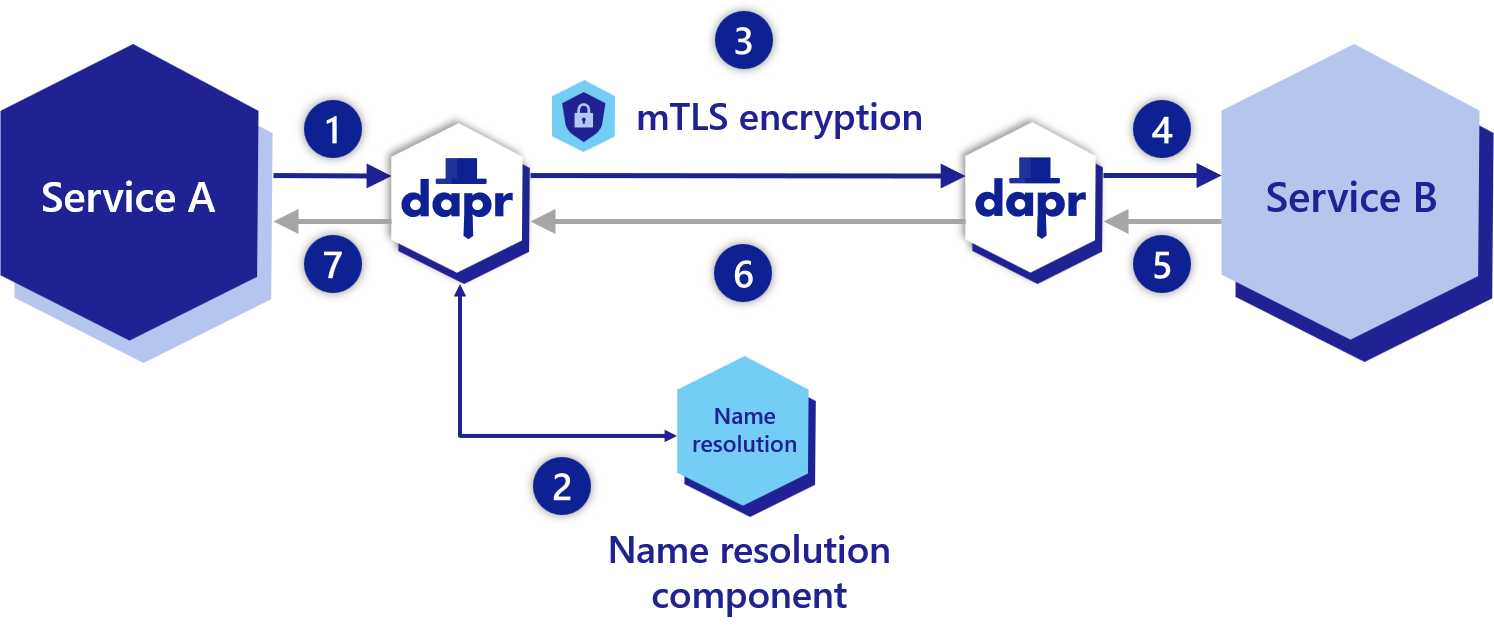


1. make编译proto和相关so文件



## 服务调用

### 概念



1. 服务 A 对服务 B 发起HTTP/gRPC的调用。
2. Dapr 使用在给定托管平台上运行的名称解析组件发现服务 B 的位置.
3. Dapr 将消息转发至服务 B的 Dapr 边车
4. 注: Dapr 边车之间的所有调用考虑到性能都优先使用 gRPC。 仅服务与 Dapr 边车之间的调用可以是 HTTP 或 gRPC
5. 服务 B的 Dapr 边车将请求转发至服务 B 上的特定端点 (或方法) 。 服务 B 随后运行其业务逻辑代码。
6. 服务 B 发送响应给服务 A。 响应将转至服务 B 的边车。
7. Dapr 将消息转发至服务 A 的 Dapr 边车。
8. 服务 A 接收响应。

### 整体的代码实现

用c++实现dapr的服务调用客户端和服务端

客户端建立client\_app目录，增加以下四个文件:

client\_app.cc实现调用方InvokeService、SaveState、GetState、DeleteState、ExecuteStateTransaction、InvokeBinding、PublishEvent、等api接口，并实现服务的启动

client\_app\_server\_impl.h继承grpc的AppCallback::Service类，并定义响应的api接口

client\_app\_server\_impl.cc实现相关的接收端api

Makefile用来编译以上文件

服务端建立helloworld\_app目录，增加以下四个文件,

helloworld\_app.cc，实现服务的启动

helloworld \_app\_server\_impl.h继承AppCallback::Service类，并定义接收方OnInvoke、ListTopicSubscriptions、ListInputBindings、OnBindingEvent、OnTopicEvent api接口

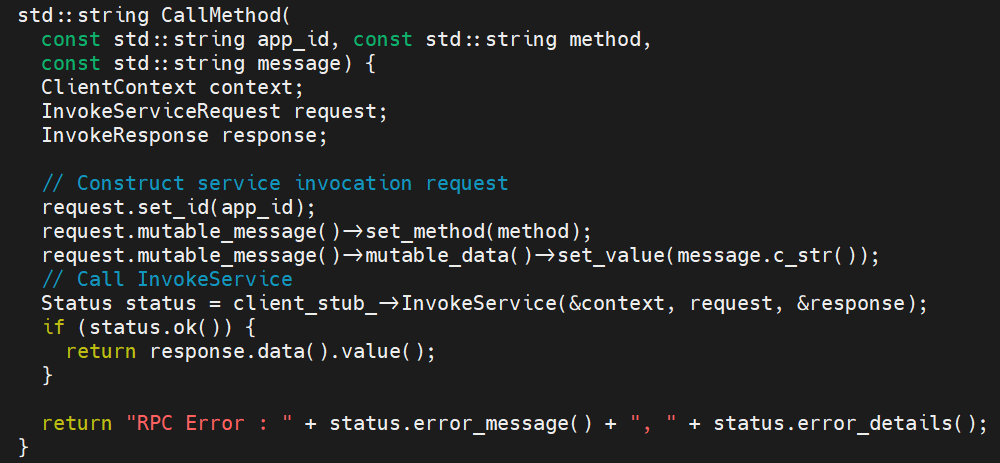
helloworld\_app\_server\_impl.cc实现相关的OnInvoke ListTopicSubscriptions、ListInputBindings、OnBindingEvent、OnTopicEvent

Makefile用来编译以上文件

### 相关场景

场景1：利用dapr的服务调用，电脑A的一个服务去调用电脑B的一个服务

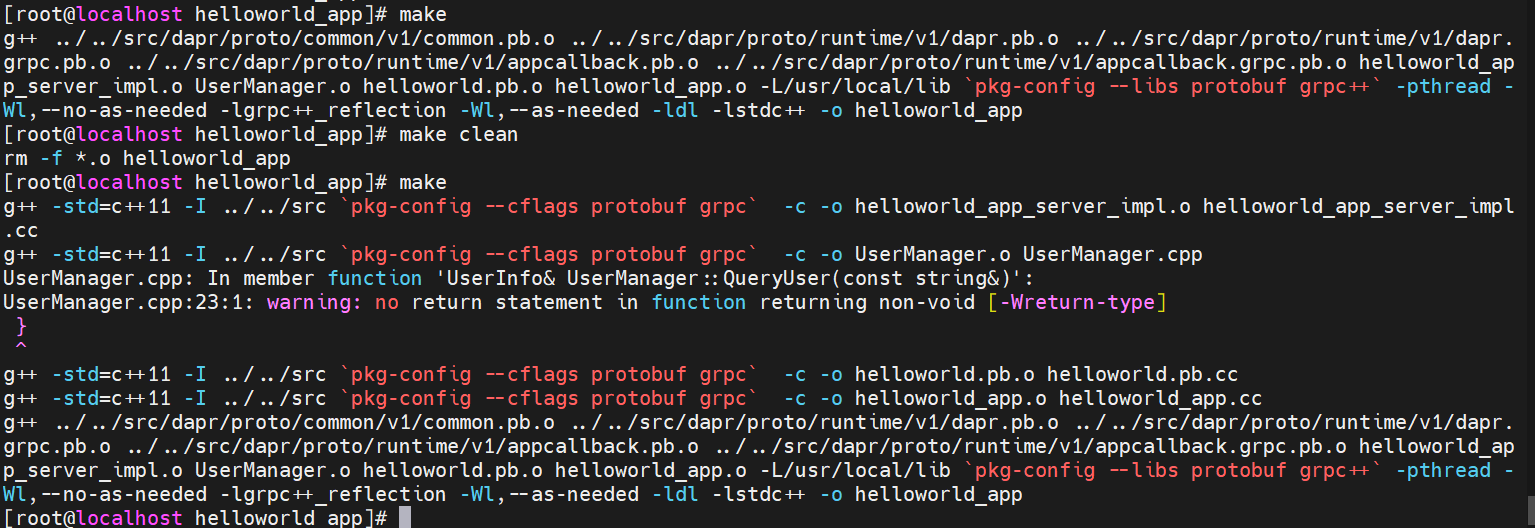
1. 先实现调用端



1. 再实现接收端



1. 编译



1. 运行命令

电脑A运行dapr命令：

dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6000 ./helloworld\_app/helloworld\_app 6000

电脑B运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

运行命令解释：

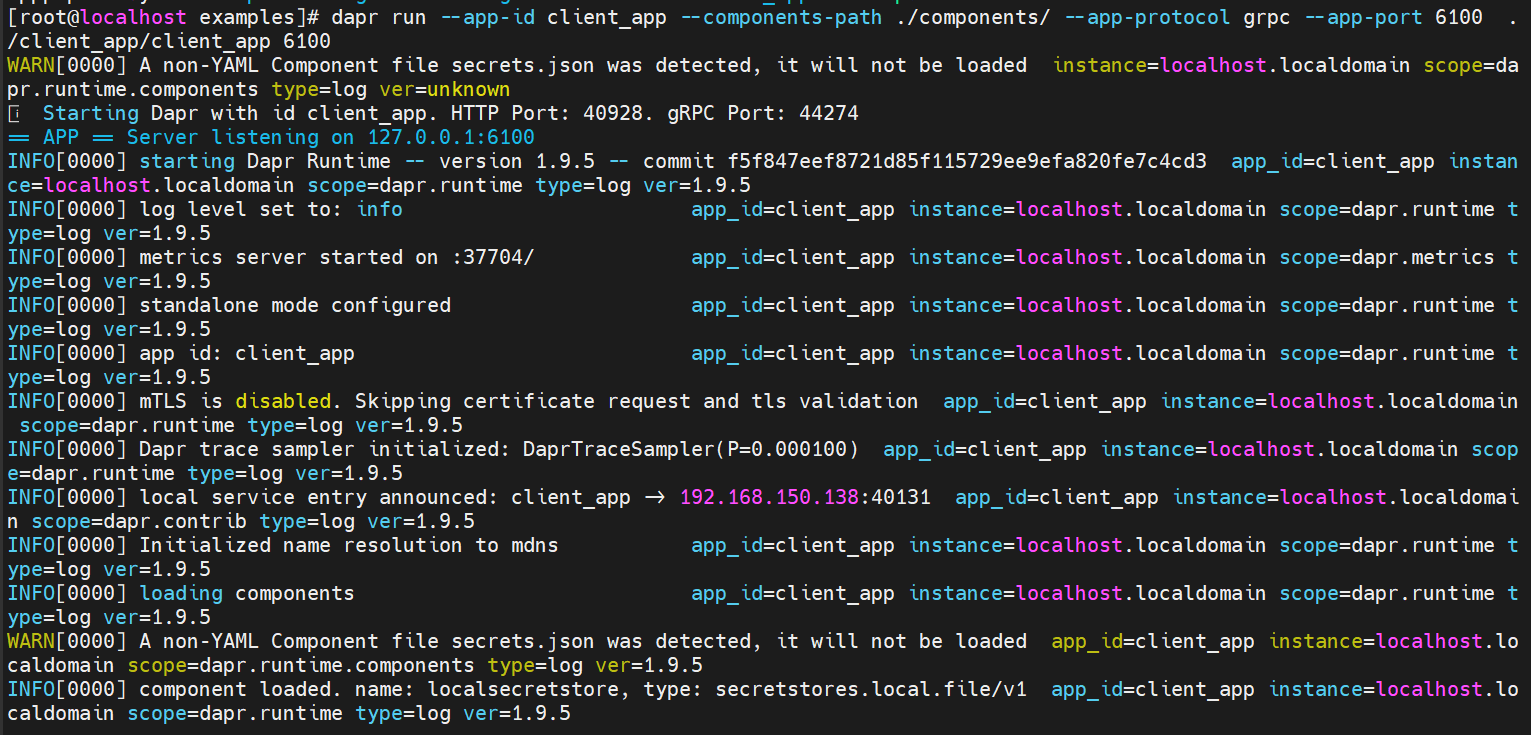
--app-id:应用程序ID

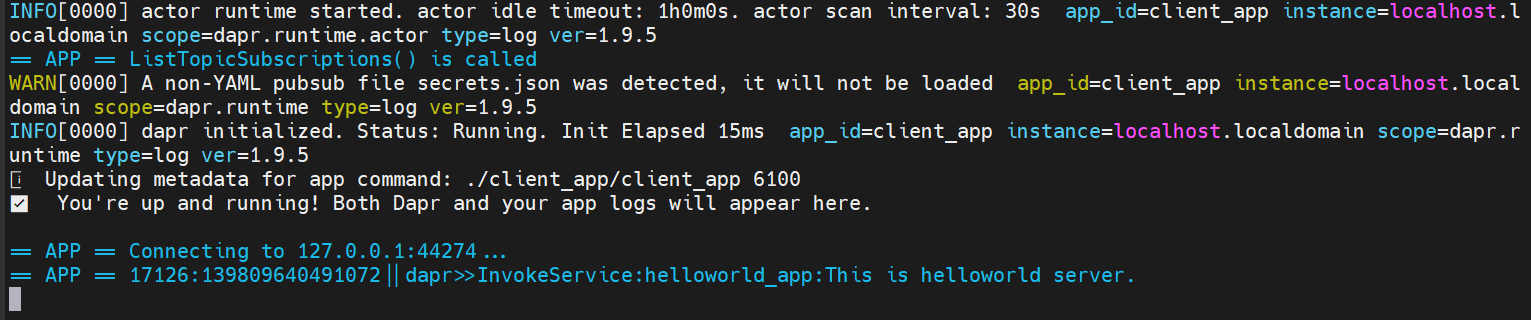
--components-path:配置文件路径

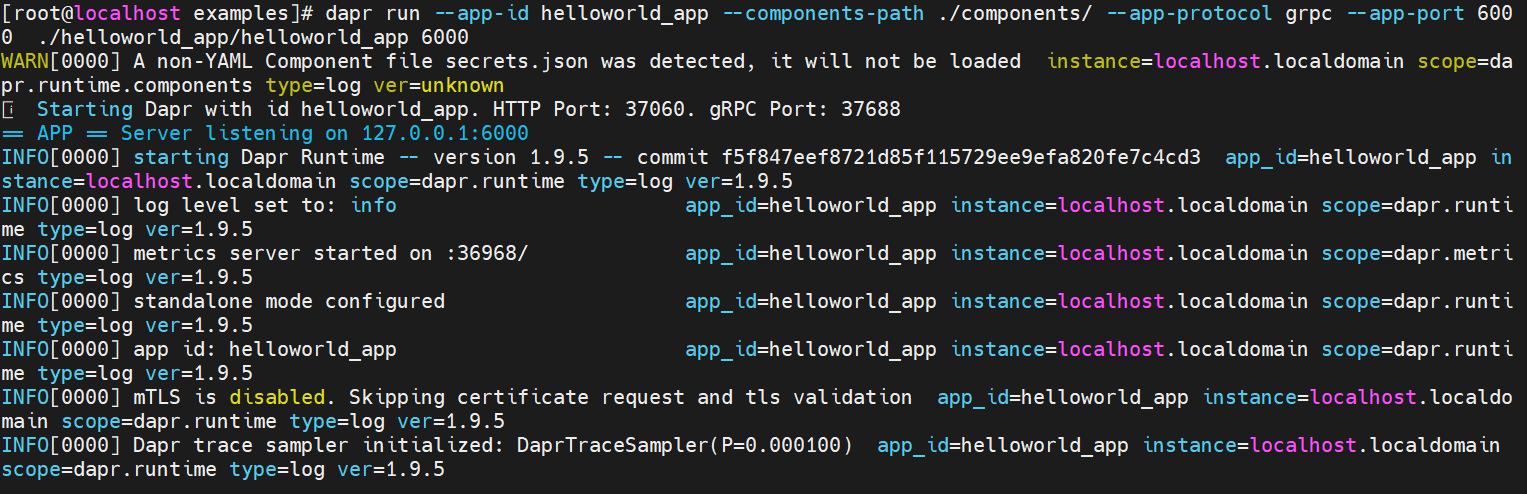
--app-protocol:通信使用的协议

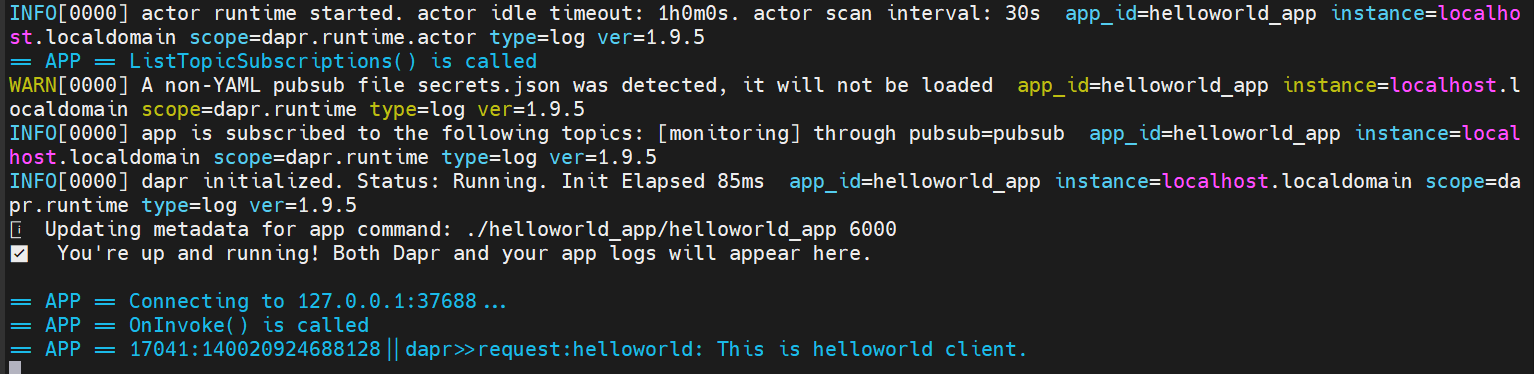
--app-port:应用程序端口

1. 运行效果









场景1.5：利用dapr的服务调用，电脑A的一个服务去调用电脑B的一个服务

用c++实现dapr的服务调用的调用端，go语言实现dapr的服务调用的接收端

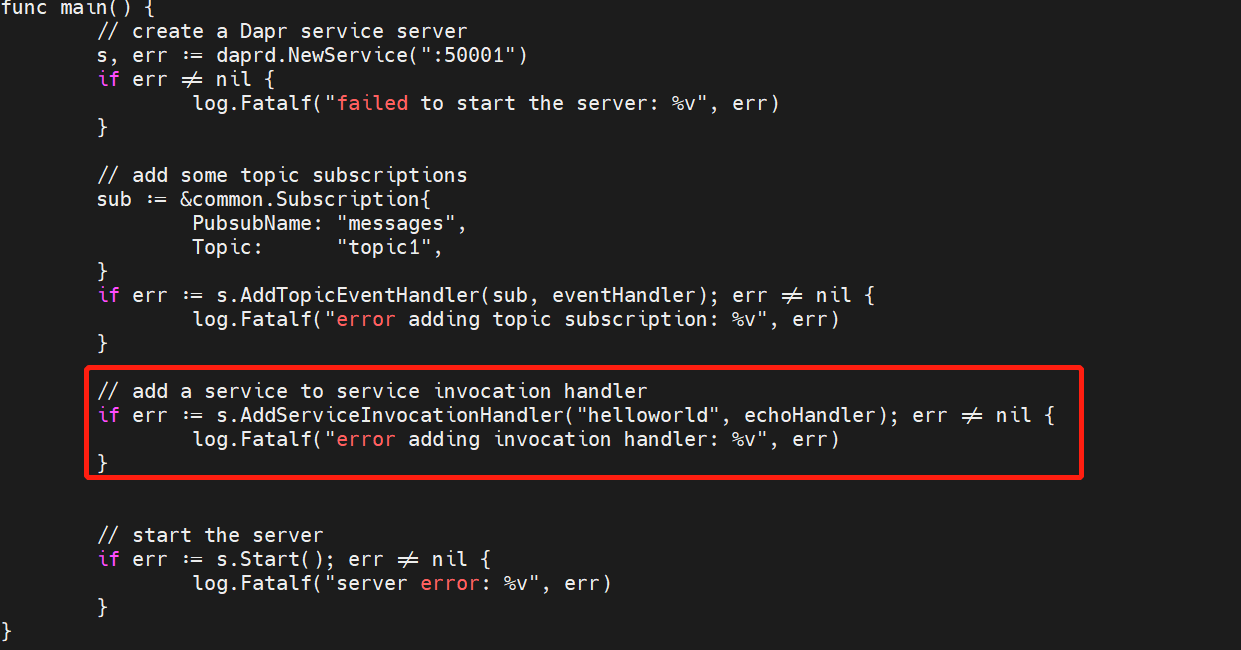
前提

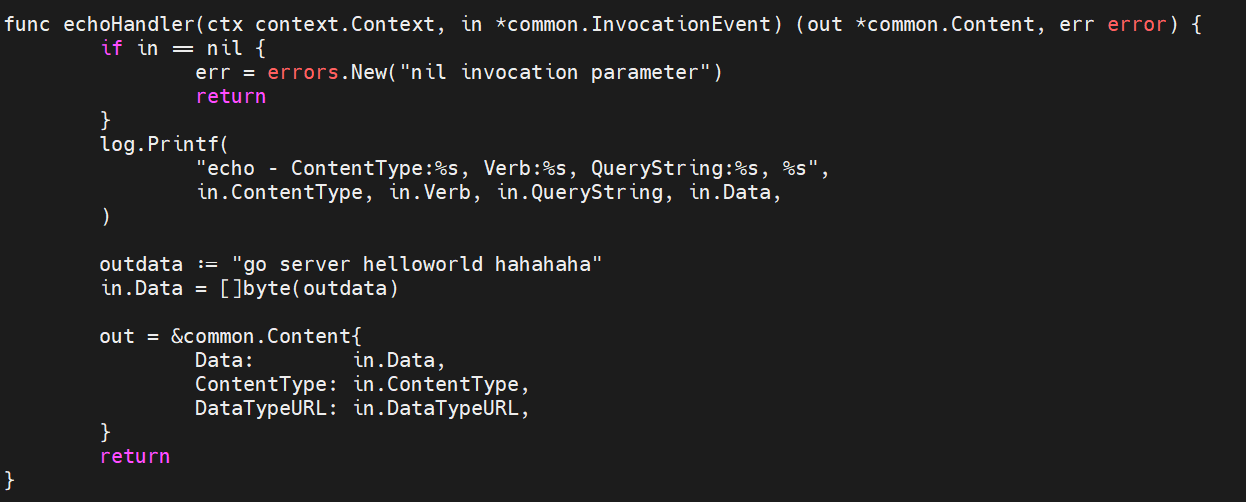
下载go-sdk，在example目录下新建helloworld目录，然后编写文件

1. 先实现调用端

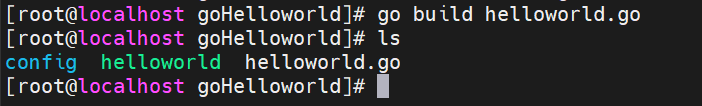
C++调用端如上

1. 再实现接收端





1. 编译



1. 运行命令

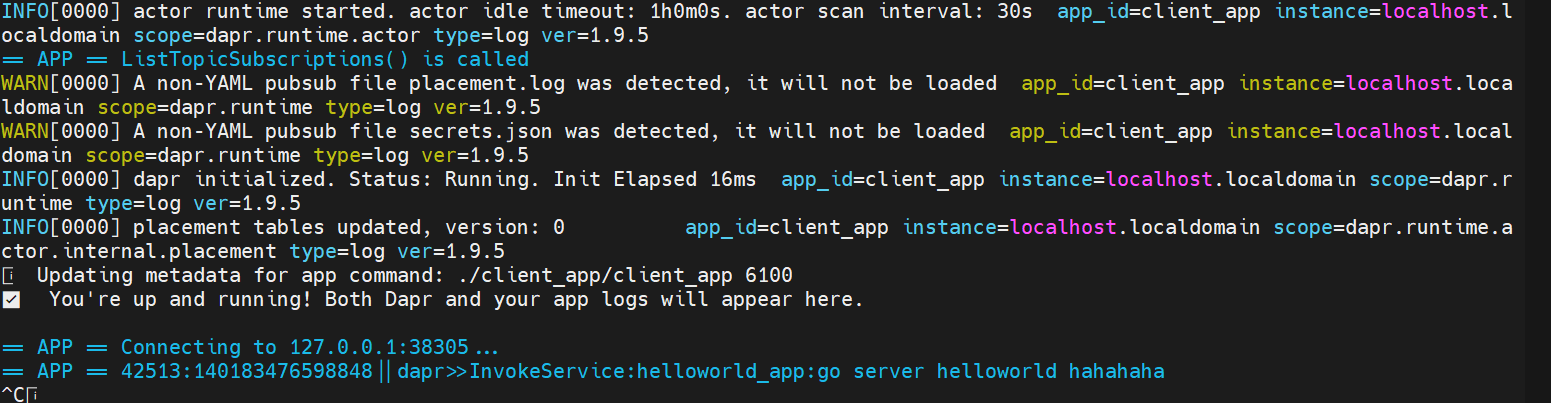
电脑A运行dapr命令：

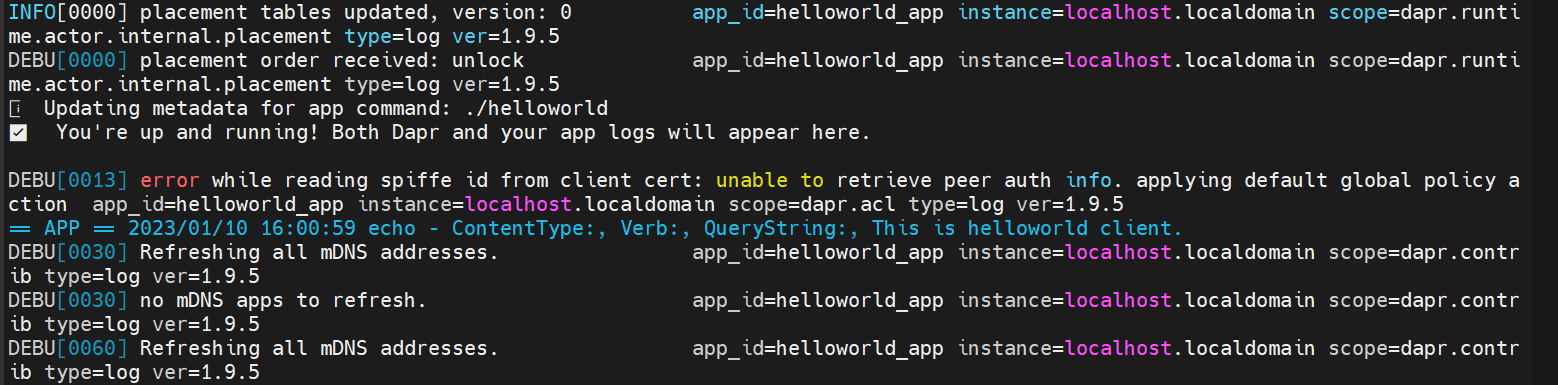
dapr run --app-id helloworld\_app --app-protocol grpc --app-port 50001 --dapr-grpc-port 3500 --log-level debug --components-path ./config ./helloworld

电脑B运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

1. 运行效果





场景2：利用dapr的服务发现，电脑A的一个服务轮询调用电脑B的dapr服务和电脑C的dapr服务

运行程序见场景1

1. 运行命令

电脑B运行dapr服务端程序

dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6000 ./helloworld\_app/helloworld\_app 6000

电脑C运行dapr服务端程序

dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6001 ./helloworld\_app/helloworld\_app 6001

电脑A运行dapr客户端程序

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

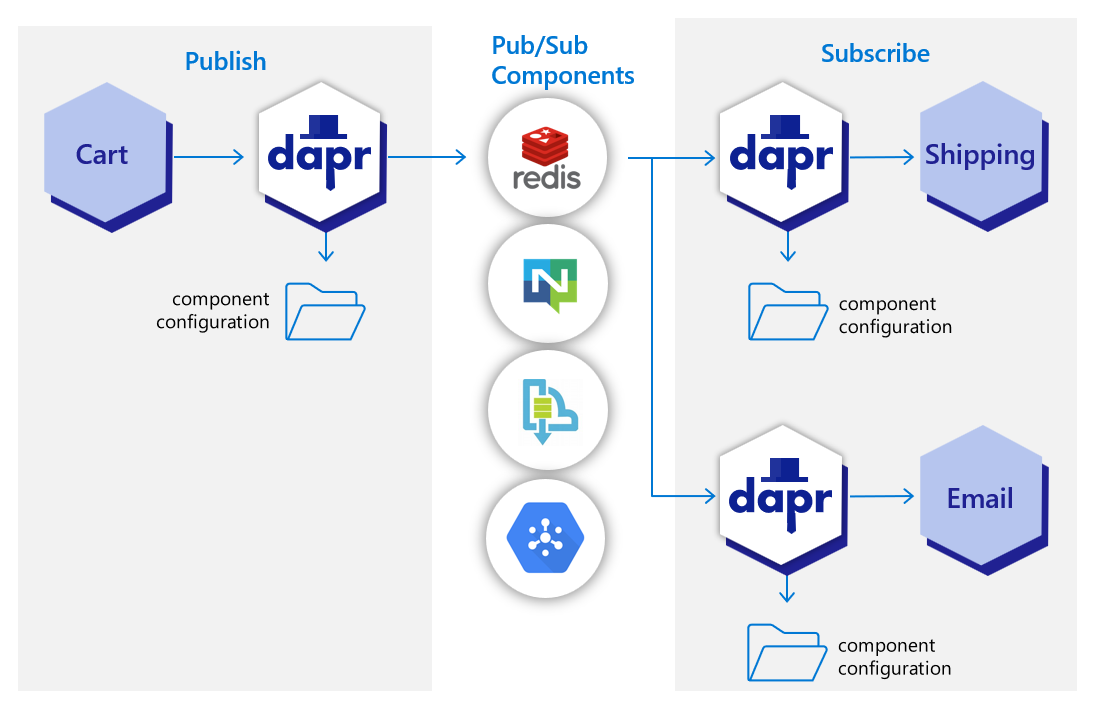
1. 运行效果

客户端A的调用将在B和C之间轮询



## 发布订阅

### 概念



发布 / 订阅允许微服务使用消息相互通信。 生产者或发布者 将消息发送至 主题（Topic） ，并且不知道接收消息的应用程序

消费者 将订阅该主题并收到它的消息，并且不知道什么应用程序生产了这些消息

中间消息代理（intermediary message broker）负责将每条消息从输入频道复制到所有对此消息感兴趣的订阅者的输出频道

在 dapr runtime 启动进行初始化时，需要

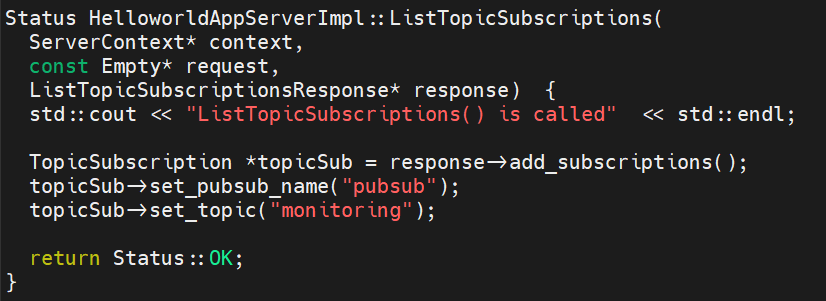
1. 访问应用以获取应用的订阅信息：比如应用订阅了哪些topic
2. 根据配置文件启动 subscribe component 以便连接到外部 message broker 进行订阅
3. 将订阅更新的 event 转发给应用

### 相关场景

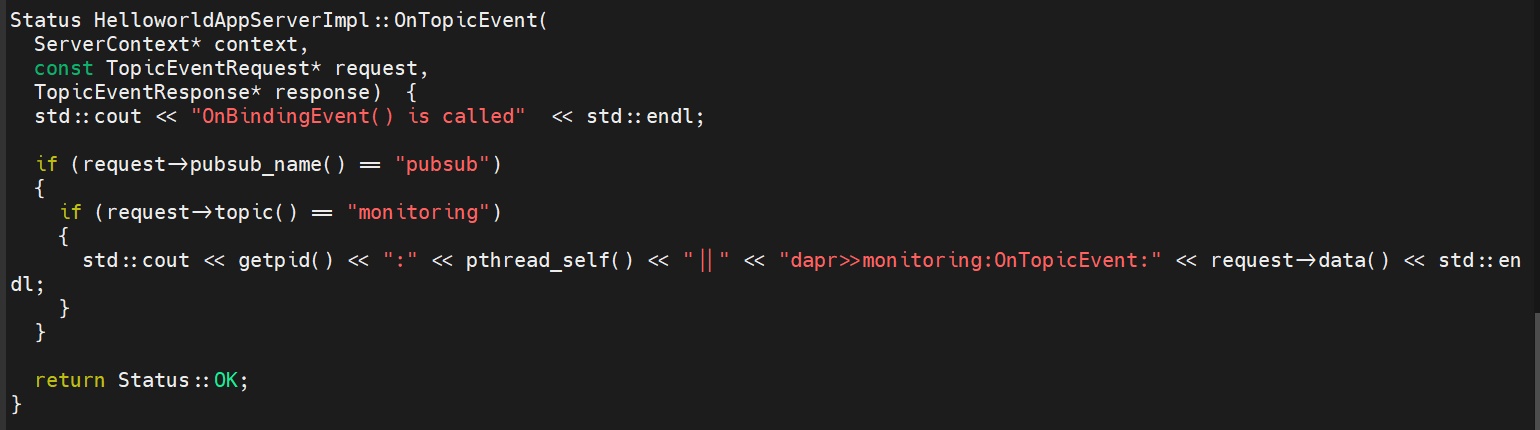
场景3：利用dapr的发布订阅，电脑A的一个服务将一个消息发送到redis上，电脑B的一个服务从redis获取消息，并处理

1. 先实现订阅端

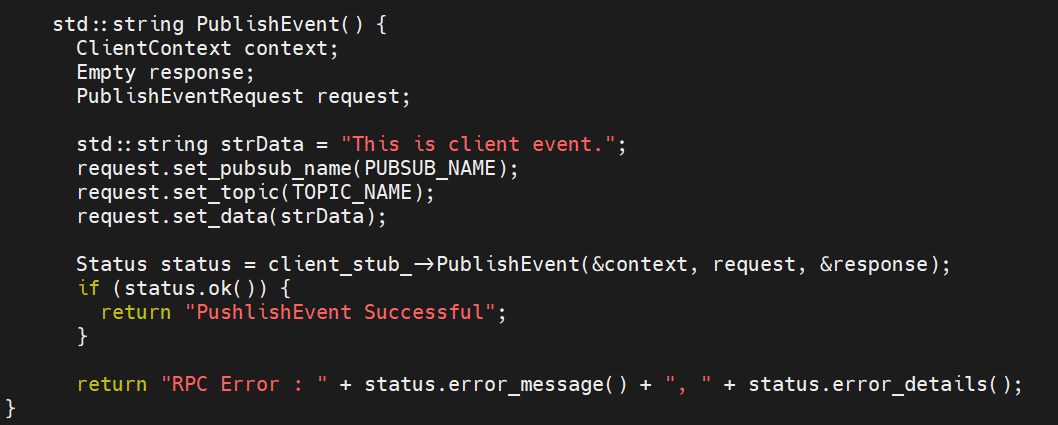
先注册时间topic



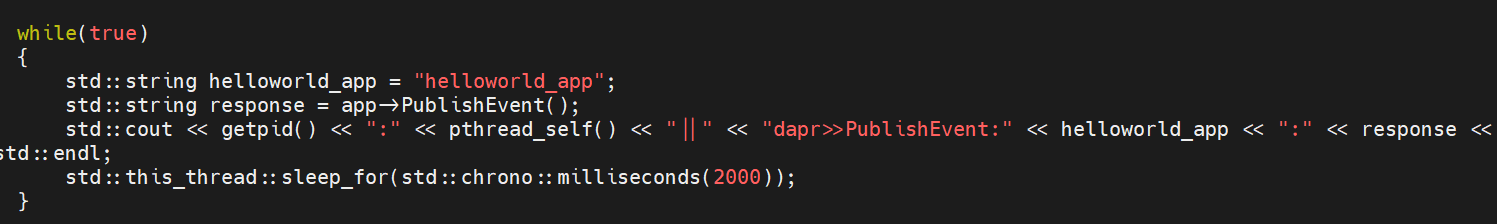
接收发布端topic的数据



1. 再实现发布端

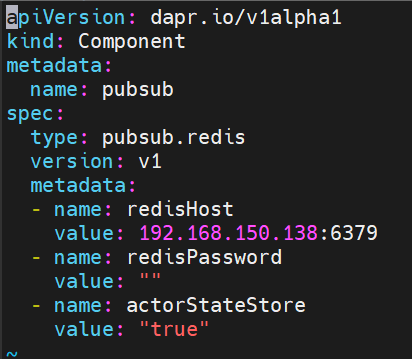


发布端调用

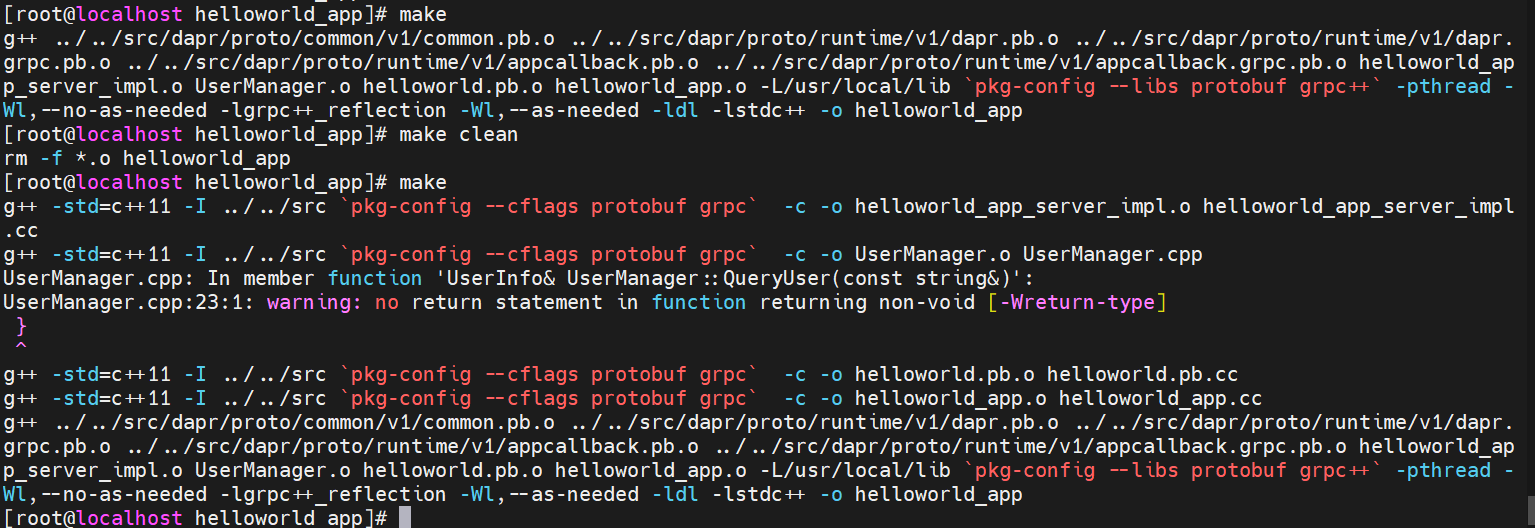


1. 定义组件

建立一个yaml文件，输入以下字段值:



1. 编译



1. 运行命令

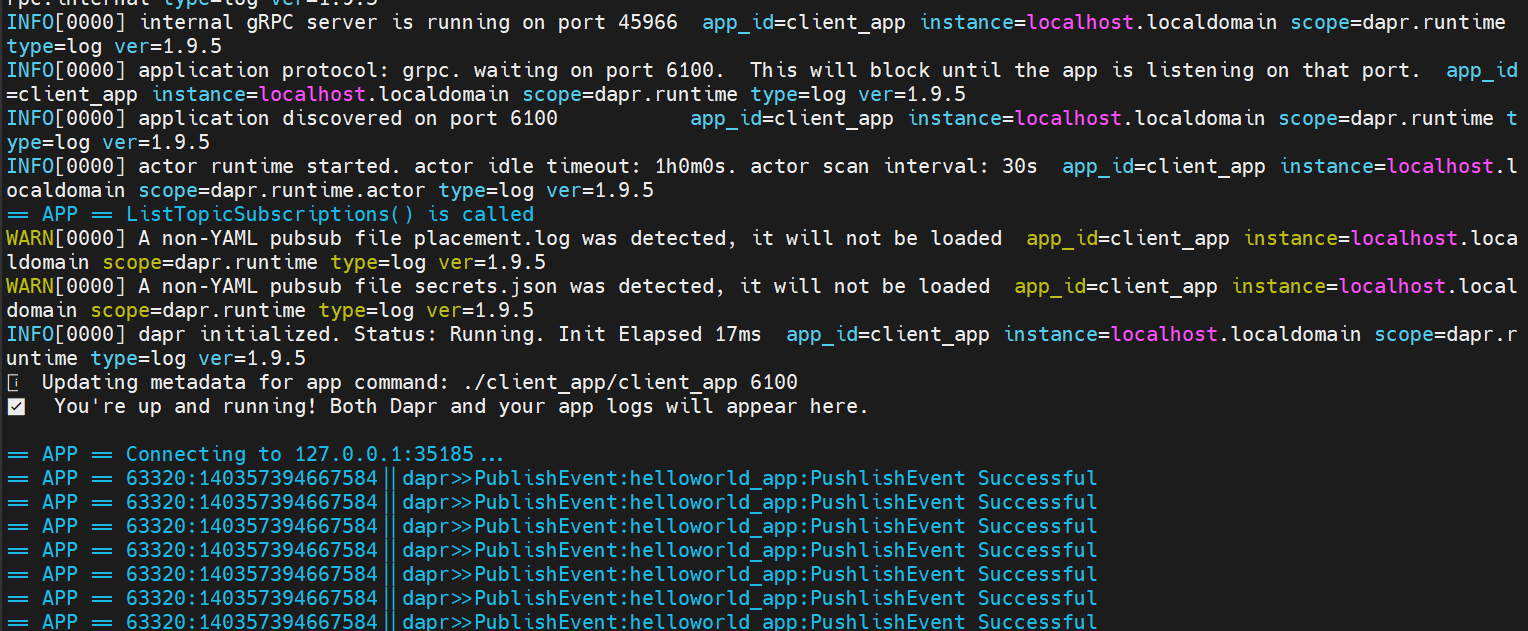
电脑A运行dapr命令：

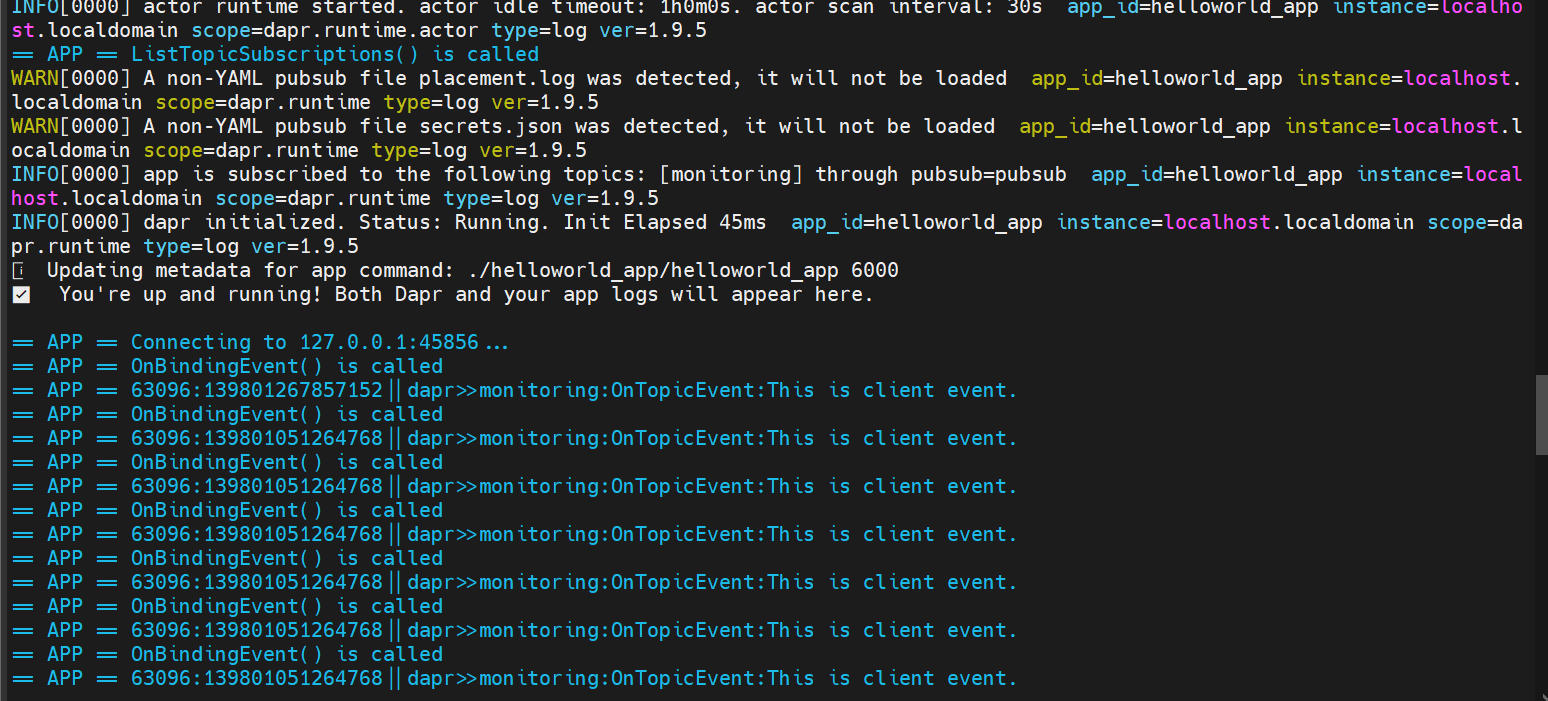
dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6000 ./helloworld\_app/helloworld\_app 6000

电脑B运行dapr命令：

dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6001 ./helloworld\_app/helloworld\_app 6001

1. 运行效果



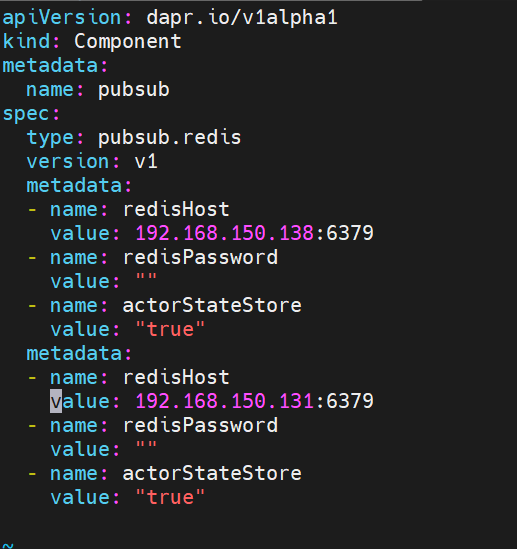


场景4：利用dapr的发布订阅，电脑A的一个服务将一个消息发送到redis上，电脑B的一个服务从redis获取消息，电脑C的一个服务从redis获取消息

1. C++代码实现

和场景3一样

1. 定义组件



1. 运行命令

电脑A运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

电脑B运行dapr命令：

dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6000 ./helloworld\_app/helloworld\_app 6000

电脑C运行dapr命令：

dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6000 ./helloworld\_app/helloworld\_app 6000

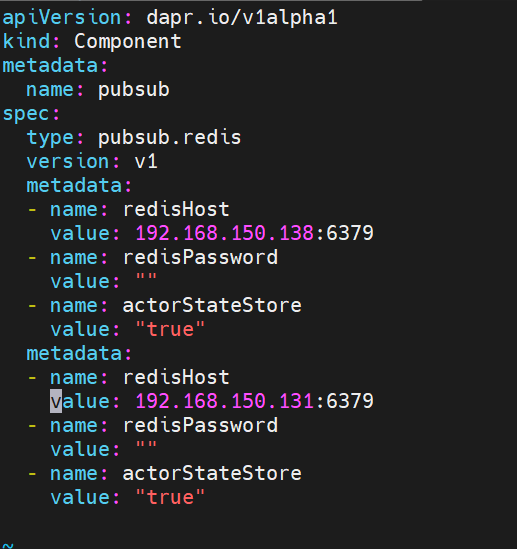
注：先启动B，C，再启动A

1. 运行效果



场景5：利用dapr的发布订阅，电脑A的一个服务将一个消息发送到redis上，电脑B的一个服务将一个消息发送到redis上，电脑C的一个服务从redis获取A和B服务发送的消息

1. 定义组件



1. 编译

和场景3一样

1. 运行命令

电脑A运行dapr命令：

dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6000 ./helloworld\_app/helloworld\_app 6000

电脑B运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

电脑C运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

1. 运行效果



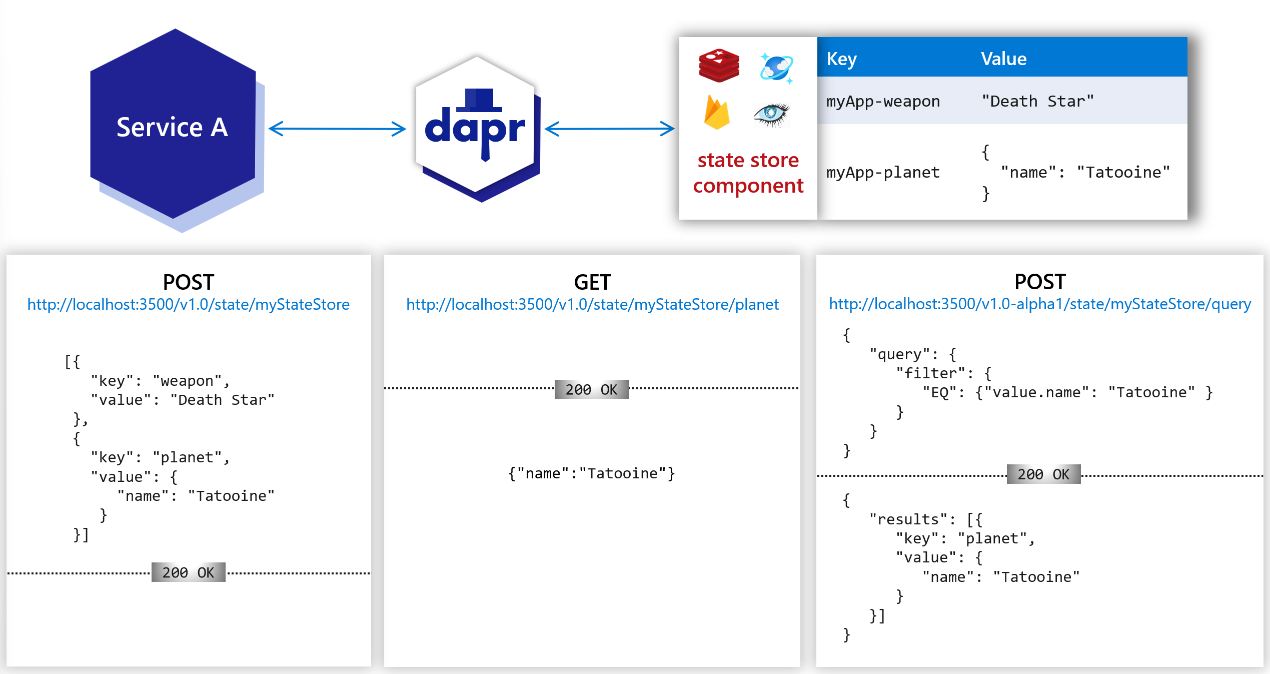
## 状态管理

### 概念

当使用状态管理时，应用程序可以利用一些自己构建会很复杂，容易出错的功能，比如:

* 分布式并发和数据一致性
* 批量[CRUD](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) 操作

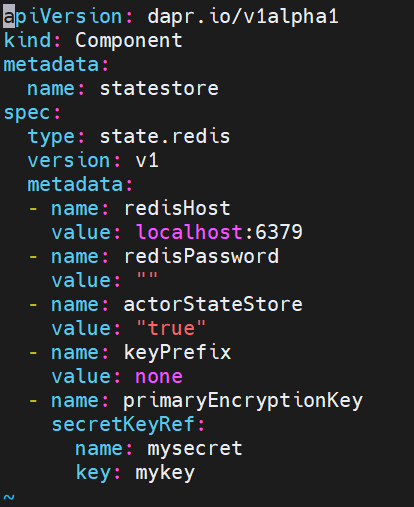
应用程序可以使用Dapr的状态管理API，使用状态存储组件保存和读取键/值对，如下图所示。 例如，通过使用HTTP POST可以保存键/值对，通过使用HTTP GET可以读取一个键并返回它的值。



### 相关场景

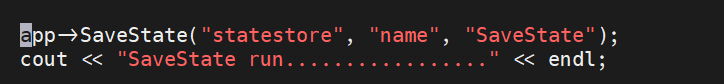
场景6：利用dapr的状态管理，电脑A的一个服务将键值对保存到redis中

1. 定义组件



1. C++实现保存键值对的代码





1. 编译

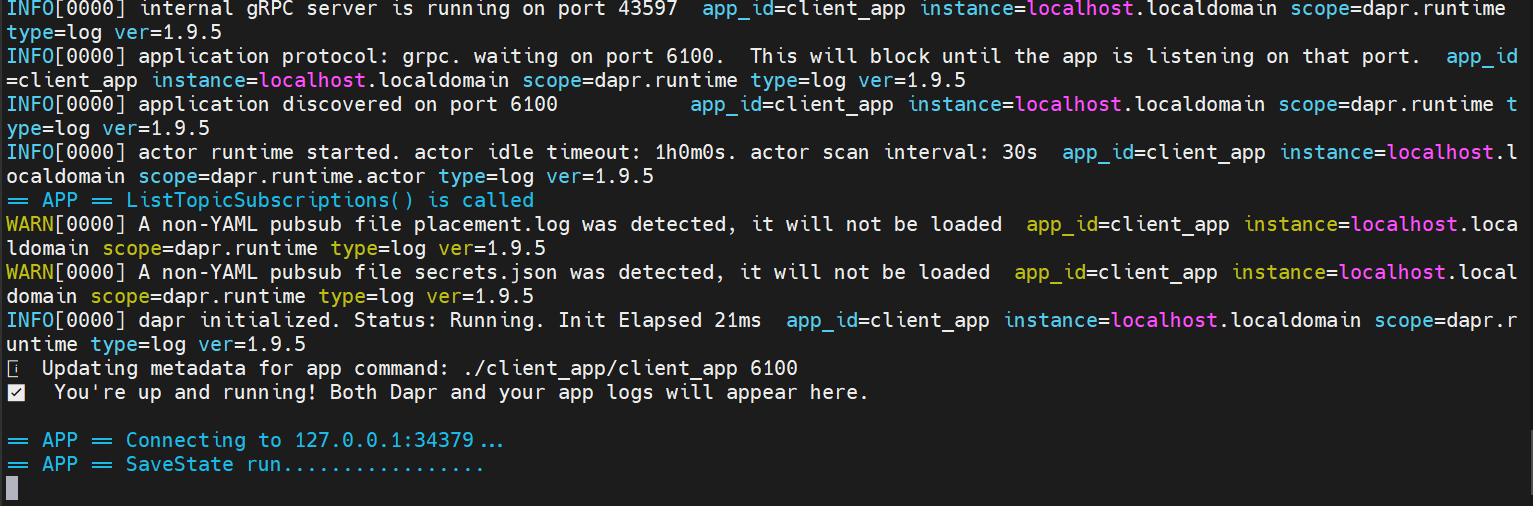
和场景1类似

1. 运行命令

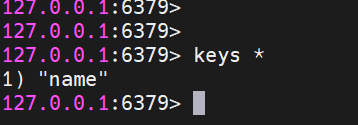
电脑A运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

1. 运行效果



登录到redis，查看键值存在不存在

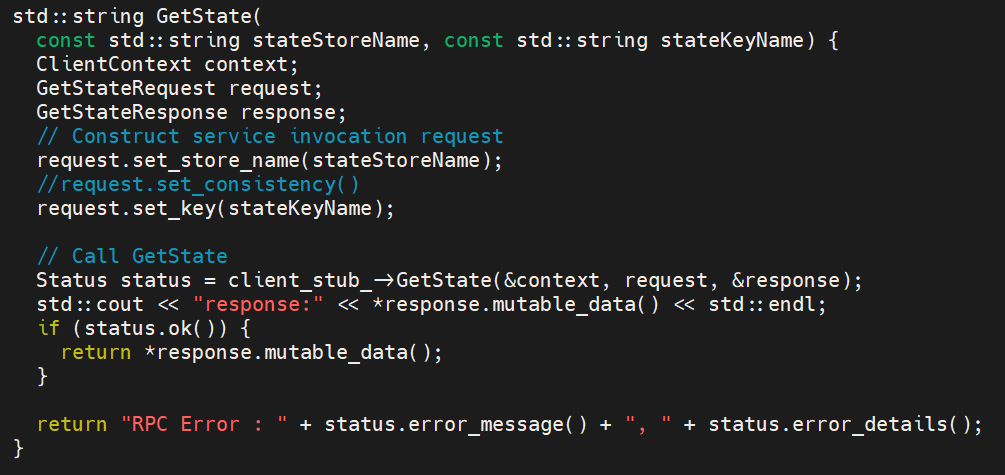


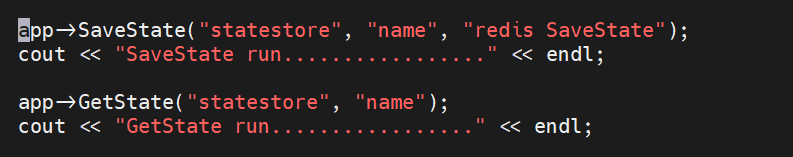
场景7：利用dapr的状态管理，电脑A的一个服务将键值对从redis中获取出来

1. 定义组件

和场景六类似

1. C++实现获取键值对的代码





1. 编译

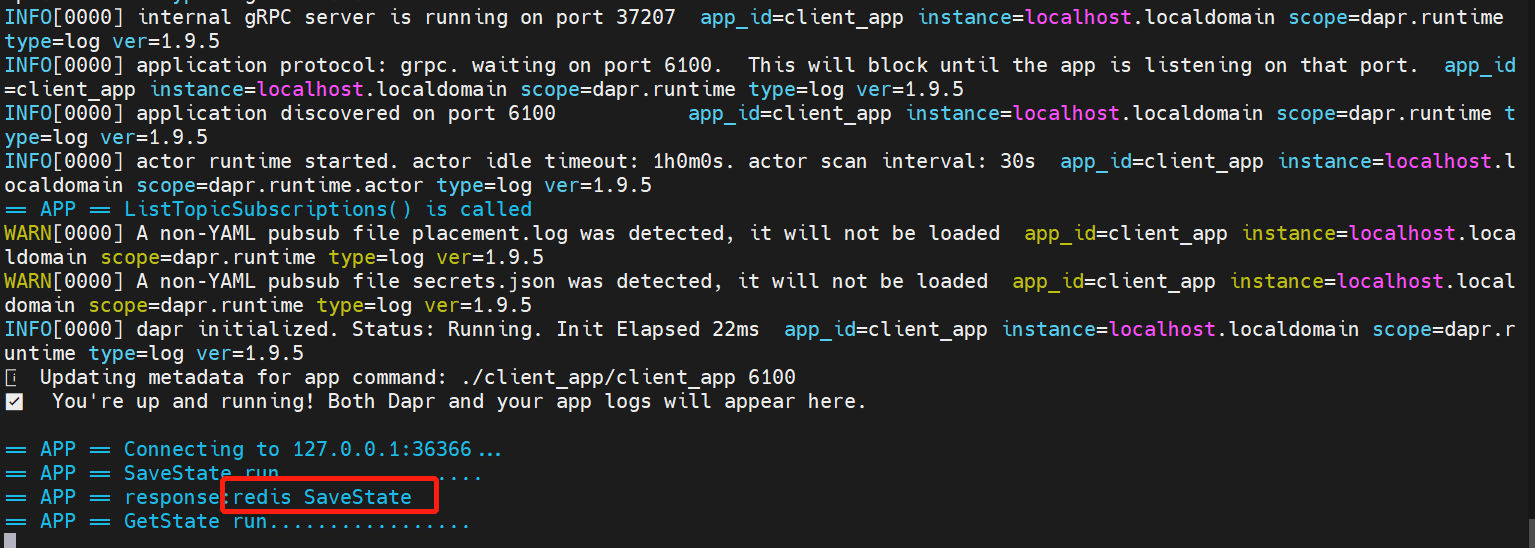
和场景1类似

1. 运行命令

电脑A运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

1. 运行效果



场景8：利用dapr的状态管理，电脑A的一个服务将键值对从redis中删除

1. 定义组件

和场景六类似

1. C++实现获取键值对的代码



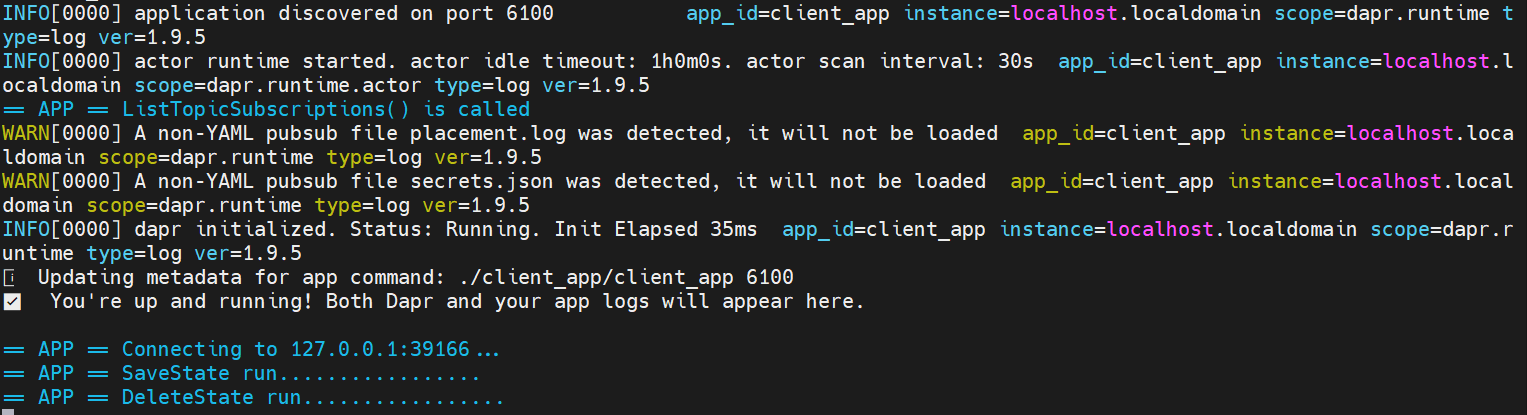


1. 运行命令

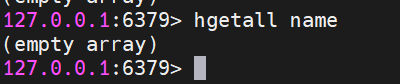
电脑A运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

1. 运行效果



登录到redis，查看savestate的值还存不存在

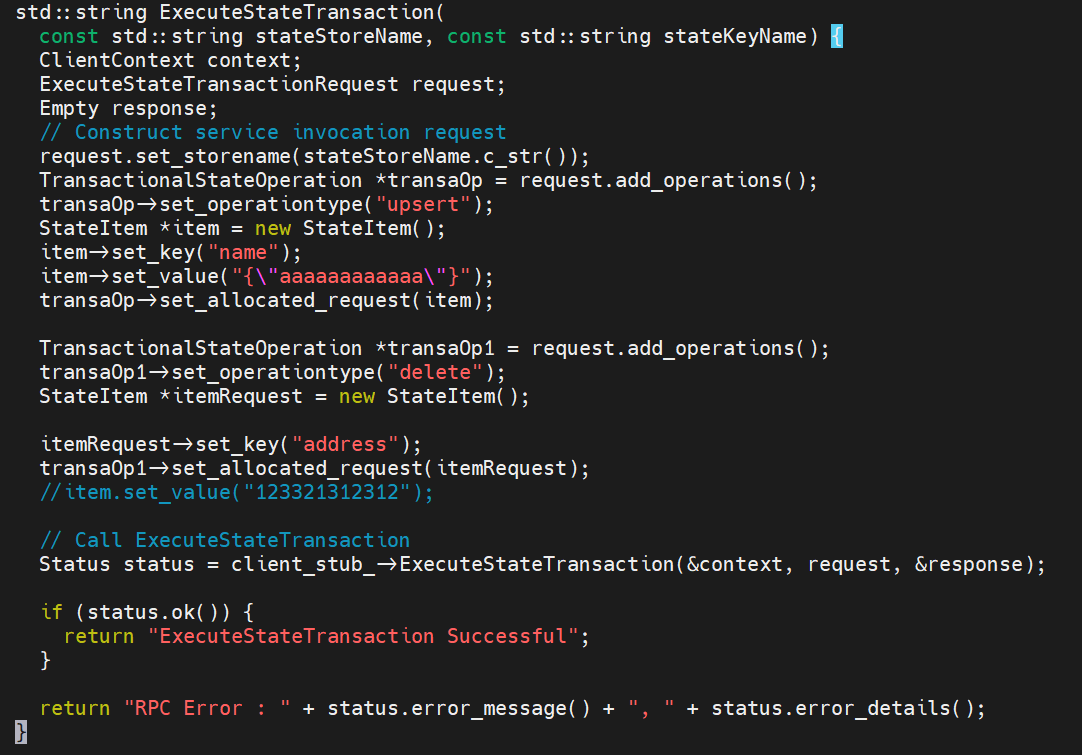


场景9：利用dapr的状态管理的状态事务性操作

1. 定义组件

和场景六类似

1. C++实现状态事务性操作



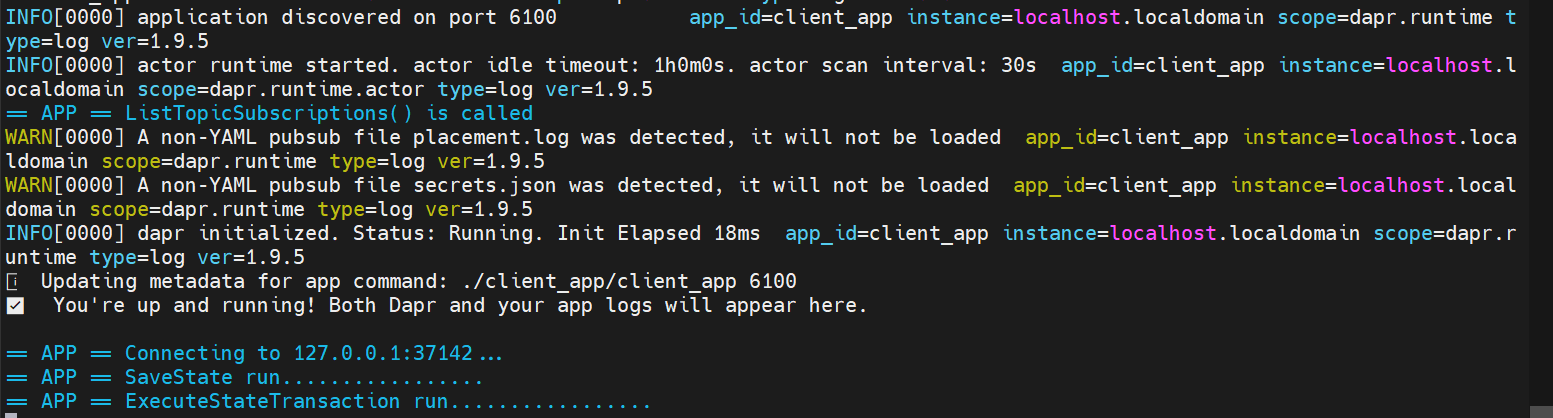


1. 运行命令

电脑A运行dapr命令：

dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

1. 运行效果



登录到redis，查看name的值有没有发生变化



## 绑定

### 概念

使用绑定，可以使用来自外部系统的事件或与外部系统的接口来触发应用程序

1)输入绑定

输入绑定用于在发生来自外部资源的事件时触发应用程序。 可选的有效负载和元数据可以与请求一起发送。

为了接收来自输入绑定的事件 :

定义描述绑定类型及其元数据 ( 连接信息等) 的组件 YAML

监听传入事件的 HTTP 终结点，或使用 gRPC 原型库获取传入事件

2)输出绑定

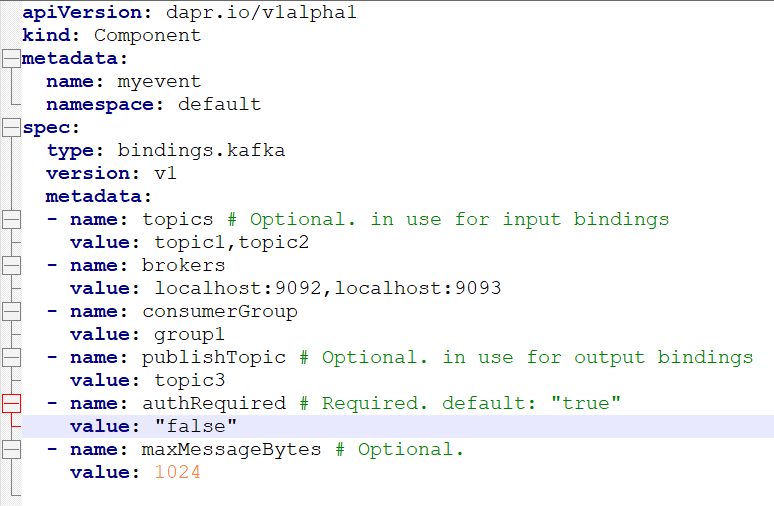
输出绑定允许用户调用外部资源。 可选的有效负载和元数据可与调用请求一起发送。

### 相关场景

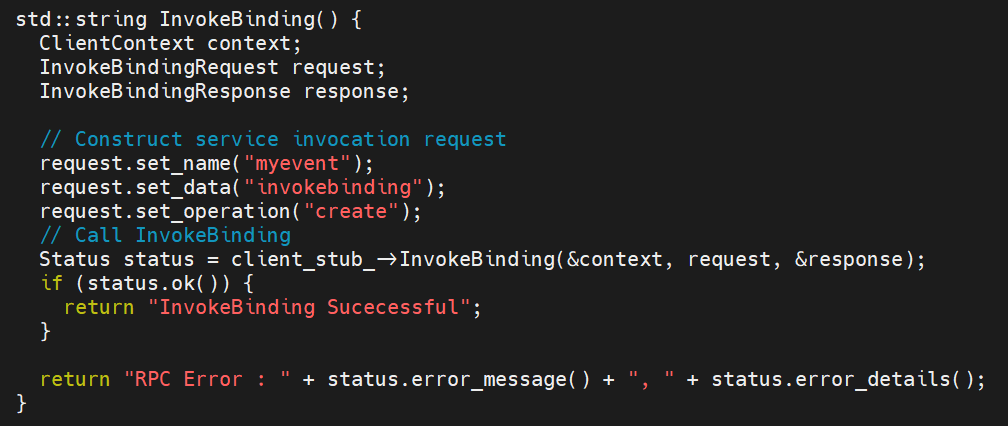
场景10：输出绑定

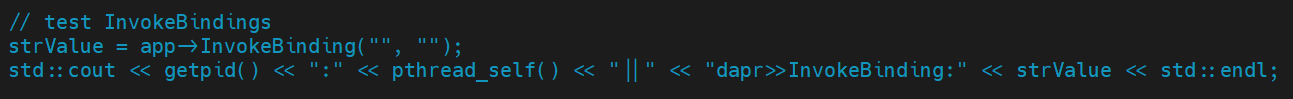
安装kafka

1. 定义组件



1. C++实现输入绑定的代码





1. 编译
2. 运行命令

先启动kafka，负责接收以下命令输出的事件

电脑A运行dapr命令：

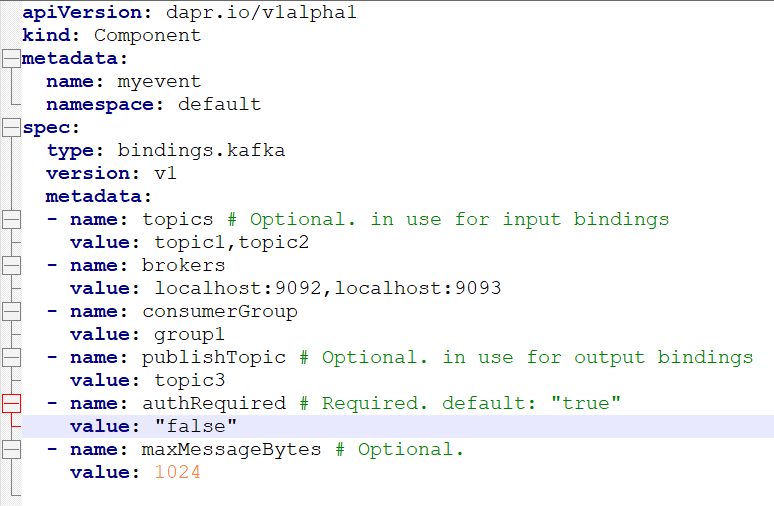
dapr run --app-id client\_app --components-path ./components/ --app-protocol grpc --app-port 6100 ./client\_app/client\_app 6100

1. 运行效果

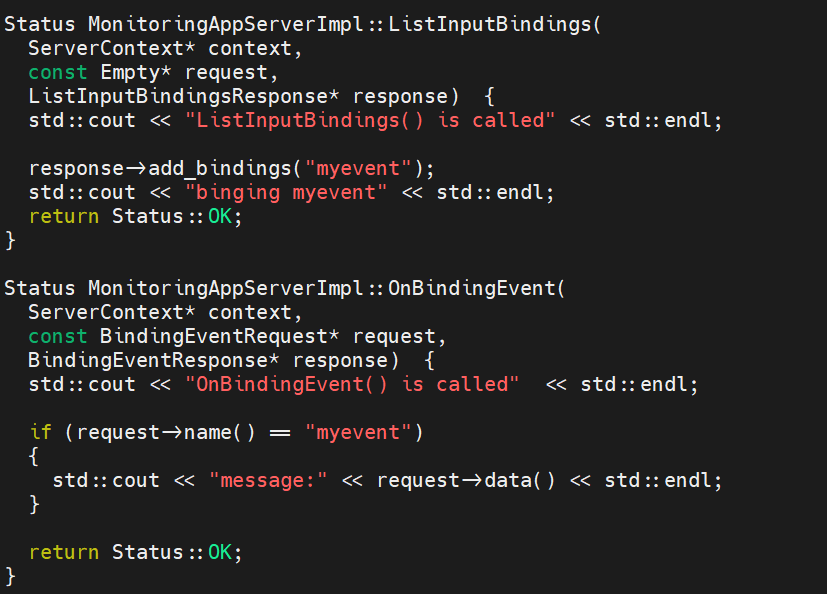
场景11：输入绑定

安装kafka

1. 定义组件



1. C++实现的代码



ListInputBindings api向边车注册要输入的事件

OnBindingEvent 接收并处理外界输入的事件

1. 编译
2. 运行命令

电脑A运行dapr命令：

dapr run --app-id helloworld\_app --components-path ./components/ --app-protocol grpc --app-port 6000 ./helloworld\_app/helloworld\_app 6000

首先启动以上命令，然后登录kafka，向myevent发送消息，dapr会受到kafka发送的事件，并处理该事件

1. 运行效果

## Actor

### actor介绍

[actor 模式](https://en.wikipedia.org/wiki/Actor_model) 阐述了 Actors 为最低级别的“计算单元”。 换句话说，您将代码写入独立单元 ( 称为actor) ，该单元接收消息并一次处理消息，而不进行任何类型的并行或线程处理。

当代码处理一条消息时，它可以向其他参与者发送一条或多条消息，或者创建新的 Actors。 底层 运行时 将管理每个 actor 的运行方式，时机和位置，并在 Actors 之间传递消息。

大量 Actors 可以同时执行，而 Actors 可以相互独立执行。

Dapr 包含专门实现 [virtual actors 模式](https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/) 的运行时。 通过 Dapr 的实现，您可以根据 Actors 模型编写 Dapr Actor，而 Dapr 利用底层平台提供的可扩展性和可靠性保证。

### 场景约束

Actor 设计模式可以很好适应一些分布式系统问题和场景，但您首先应该考虑的是模式的约束。 一般来说，在下列情况下，考虑 actor 模式来模拟你的问题或场景：

* 问题空间涉及大量(数千或更多) 的独立和孤立的小单位和逻辑。
* 想要处理单线程对象，这些对象不需要外部组件的大量交互，例如在一组 Actors 之间查询状态。
* actor 实例不会通过发出I/O操作来阻塞调用方。

### 命令行

下载go-sdk,进入example/actors目录，运行以下命令：

运行Actor服务端:

dapr run --app-id actor-serving \

--app-protocol http \

--app-port 8080 \

--dapr-http-port 3500 \

--log-level debug \

--components-path ./config \

go run ./serving/main.go

运行Actor客户端：

dapr run --app-id actor-client \

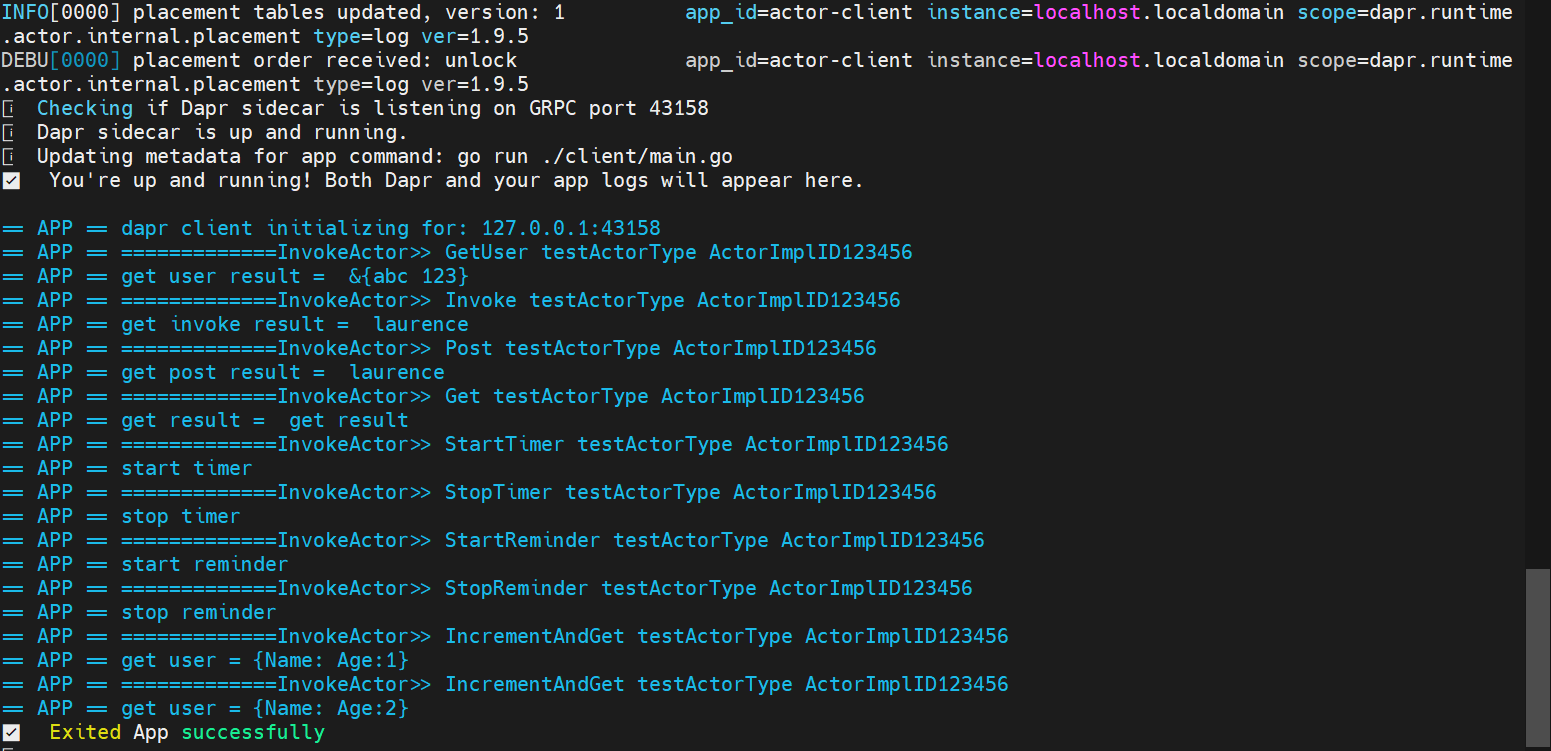
--log-level debug \

--components-path ./config \

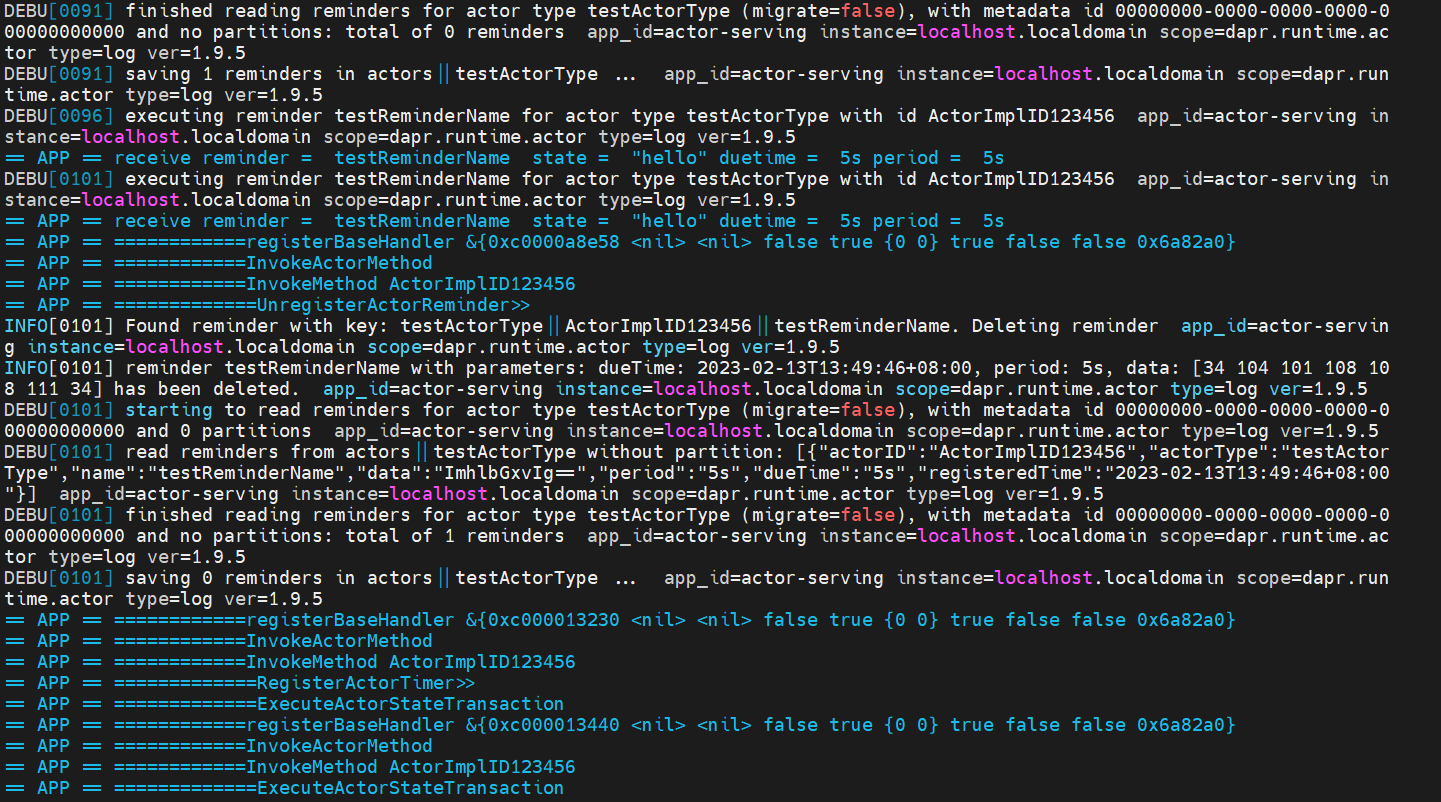
go run ./client/main.go

### 运行效果

客户端：



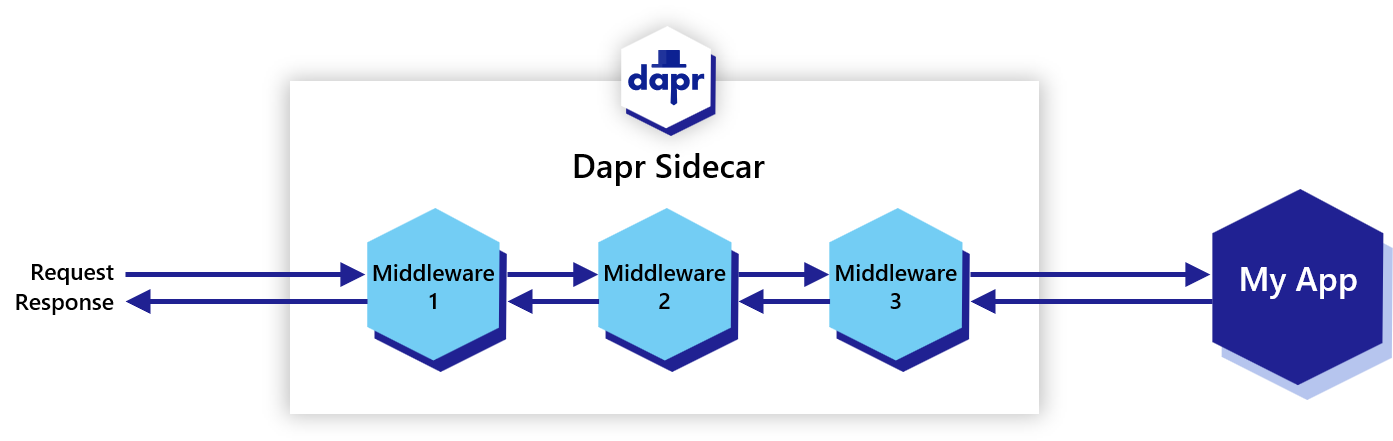
服务端：



## 中间件

### 概念

Dapr 允许通过链接一系列中间件组件来定义自定义处理管道。 请求在路由到用户代码之前经过所有已定义的中间件组件，然后在返回到客户机之前，按相反顺序经过已定义的中间件，如下图中所示。

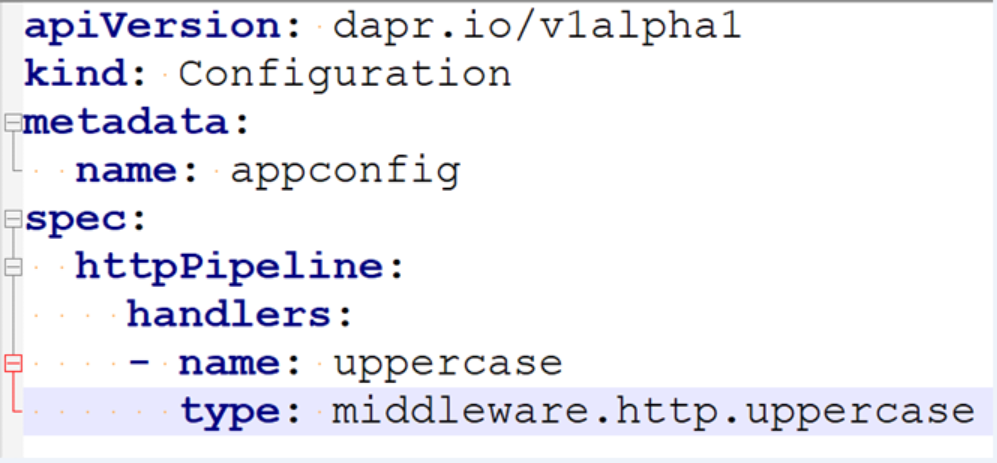


### dapr目前支持的中间件

|  |  |  |  |
| --- | --- | --- | --- |
| [Rate limit](https://docs.dapr.io/zh-hans/developing-applications/middleware/supported-middleware/middleware-rate-limit/) | 限制每秒允许的 HTTP 请求的最大数量 | Alpha | v1 |
| [OAuth2](https://docs.dapr.io/zh-hans/developing-applications/middleware/supported-middleware/middleware-oauth2/) | 在Web API上启用[OAuth2授权授权流程](https://tools.ietf.org/html/rfc6749#section-4.1) | Alpha | v1 |
| [OAuth2 client credentials](https://docs.dapr.io/zh-hans/developing-applications/middleware/supported-middleware/middleware-oauth2clientcredentials/) | 在Web API上启用[OAuth2客户端凭证授予流程](https://tools.ietf.org/html/rfc6749#section-4.4) | Alpha | v1 |
| [Bearer](https://docs.dapr.io/zh-hans/developing-applications/middleware/supported-middleware/middleware-bearer/) | 使用 [OpenID Connect](https://tools.ietf.org/html/rfc6750)在 Web API 上验证 [Bearer Token](https://openid.net/connect/) | Alpha | v1 |
| [Open Policy Agent](https://docs.dapr.io/zh-hans/developing-applications/middleware/supported-middleware/middleware-opa/) | 将[Rego/OPA策略](https://www.openpolicyagent.org/)应用到传入的Dapr HTTP请求中 | Alpha | v1 |
| [Uppercase](https://docs.dapr.io/zh-hans/developing-applications/middleware/supported-middleware/middleware-uppercase/) | 将请求的正文转换为大写字母 | Alpha | V1 |

### 如何配置中间件

配置uppercase的dapr中间件，该中间件是将http请求从小写转化为大写



配置uppercase的app中间组件



### 启动服务端和客户端命令

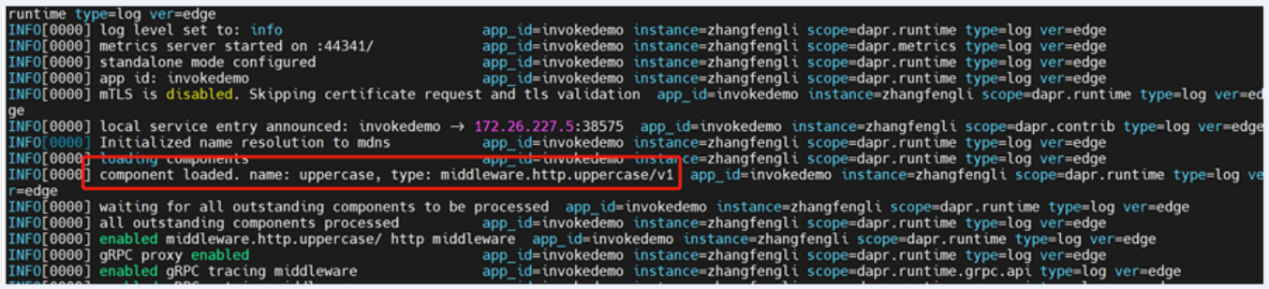
dapr run --app-id invokedemo --app-port 3000 --components-path ./components/middleware/ --config components/middleware/dapr\_uppercase.yaml -- java -jar target/dapr-java-sdk-examples-exec.jar io.dapr.examples.invoke.http.DemoService -p 3000

dapr run --app-id invokeclient --components-path ./components/middleware/ --config ./components/middleware/dapr\_uppercase.yaml -- java -jar target/dapr-java-sdk-examples-exec.jar io.dapr.examples.invoke.http.InvokeClient "message one" "message two"

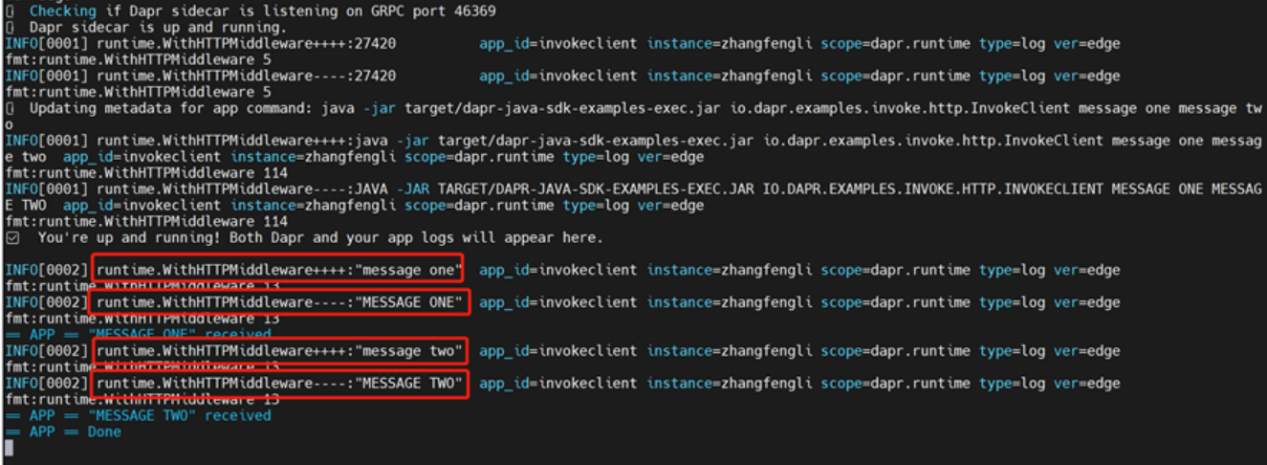
 注：运行该命令需要下载java SDK

### 运行效果

启动服务的时候，红框中的打印，表示配置成功



传输过程中的请求从小写转化为大写



## 日志

Dapr以纯文本形式或JSON格式生成结构化日志到标准输出。 默认情况下，所有 Dapr 进程 (运行时和系统服务) 都以纯文本写入控制台输出。

要启用 JSON 格式的日志，需要在运行 Dapr 进程时添加 --log-as-json 命令标志。

### Json格式配置命令

dapr run --app-id myapp --dapr-http-port 3500 --log-as-json

### 日志架构

Dapr 基于以下架构生成日志

| **字段** | **说明** | **Example** |
| --- | --- | --- |
| time | ISO8601 时间戳 | 2011-10-05T14:48:00.000Z |
| level | 日志级别 (info/warn/debug/error) | info |
| type | 日志类型 | log |
| msg | 日志消息 | hello dapr! |
| 作用域 | 日志记录范围 | dapr.runtime |
| instance | 容器名称 | dapr-pod-xxxxx |
| app\_id | Dapr 应用 ID | dapr-app |
| ver | Dapr 运行时版本 | 1.8.0 |

### 纯文本和 JSON 格式的日志

纯文本日志示例

 INFO[0000] enabled gRPC tracing middleware app\_id=helloworld\_app instance=localhost.localdomain scope=dapr.runtime.grpc.internal type=log ver=1.9.5

JSON 格式的日志示例

{"app\_id":"helloworld\_app","instance":"localhost.localdomain","level":"info","msg":"application protocol: grpc. waiting on port 6000. This will block until the app is listening on that port.","scope":"dapr.runtime","time":"2023-02-01T10:01:11.804566477+08:00","type":"log","ver":"1.9.5"}

## 追踪

Dapr 使用 Zipkin 协议进行分布式跟踪和指标收集。

### 启动zipkin服务器

如下命令，启动zipkin服务

nohup java -jar zipkin-server-2.12.9-exec.jar >zipkin.log 2>&1 &

### 配置zipkin



注：该配置放到~/.dapr目录下，文件：config.yaml

### 运行效果



## Dashboard

### 启动命令

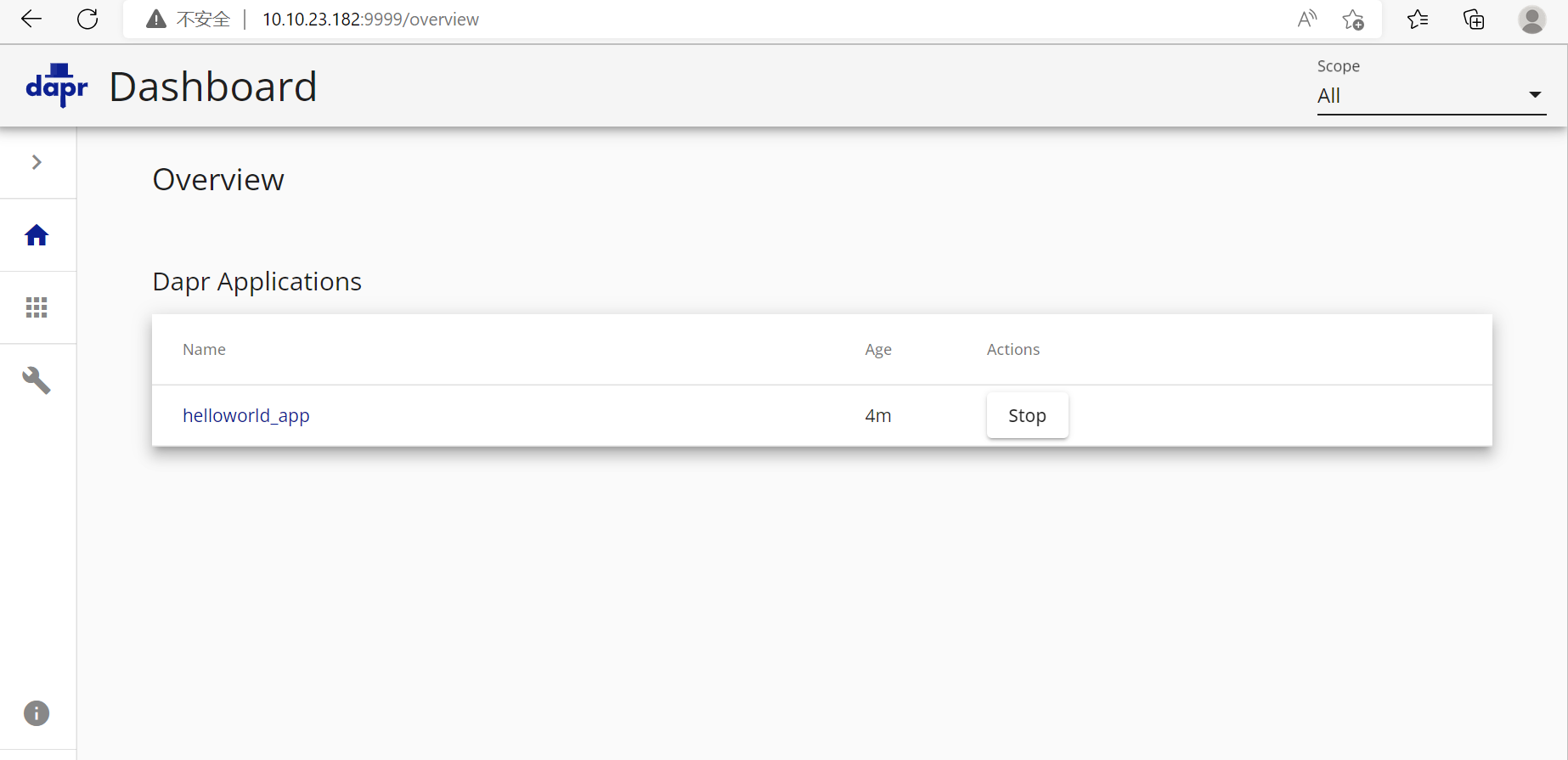
**在命令行启动以下命令：**

**dapr** dashboard -p 9999 # 默认端口为8080

使用浏览器访问dashboard：

http://ip:9999/

### 运行效果



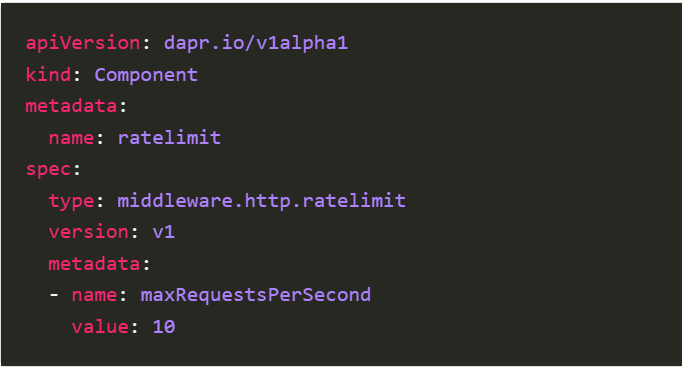
## 配置

### 并发 & 限流

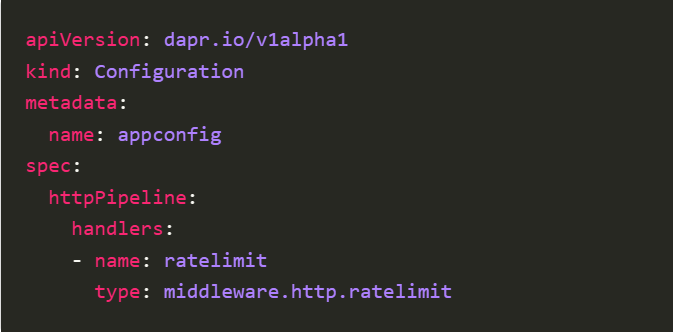
Dapr通过使用速率限制中间件来限制每秒的请求。速率限制可以保护应用程序免受拒绝服务（DOS）攻击。 DOS攻击可以由恶意的第三方发起，也可以由软件中的错误发起

如何配置速率限制

应用程序配置



Dapr速率限制的配置



并发设置

dapr通过在命令行设置该参数 --app-max-concurrency，从而达到设置并发的目的

### 限制可从机密存储读取的机密

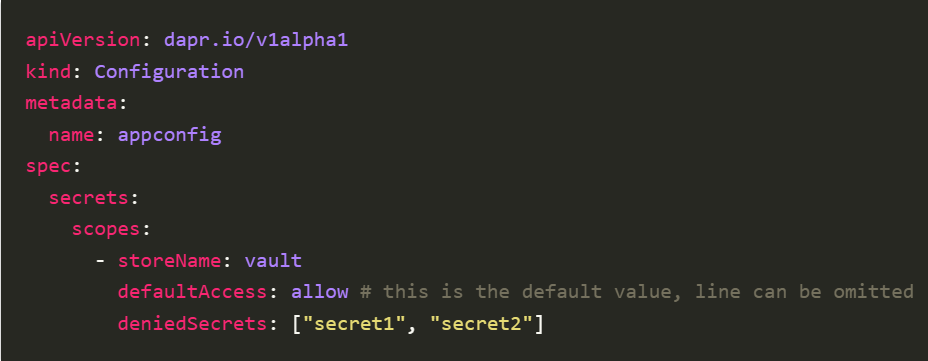
若要限制 Dapr 应用程序有权访问的机密，用户可以通过使用限制性权限扩充现有配置 CRD 来定义机密范围。

#### 允许访问机密存储中的某些机密

定义以下内容：config.yaml



#### 拒绝访问机密存储中的某些敏感机密



### 服务调用访问控制

限制调用应用程序可以通过服务调用对被调用的应用程序执行的操作

#### 配置属性

下表列出了访问控制、策略和操作的不同属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 数据类型 | 说明 |
| 默认操作 | 字符串 | 没有其他策略匹配时的全局默认操作 |
| 信任域 | 字符串 | 分配给应用程序的信任域。默认值为“公共”。 |
| 政策 | 字符串 | 用于确定调用应用可以对被调用应用执行的操作的策略 |

#### 策略

|  |  |  |
| --- | --- | --- |
| 属性 | 数据类型 | 说明 |
| .app | 字符串 | 用于允许/拒绝服务调用的调用应用的 AppId |
| 命名空间 | 字符串 | 需要与调用应用的命名空间匹配的命名空间值 |
| 信任域 | 字符串 | 需要与调用应用的信任域匹配的信任域。默认值为“公共” |
| 默认操作 | 字符串 | 应用级默认操作 |
| 功能操作 | 字符串 | 允许从调用应用执行的操作 |

#### 功能操作

|  |  |  |
| --- | --- | --- |
| **属性** | **数据类型** | **说明** |
| 名字 | 字符串 | 被调用应用上允许的操作的路径名。通配符“\*”可用于在路径下进行匹配 |
| httpVerb | 列表 | 列出调用应用可以使用的特定 http 谓词。通配符“\*”可用于匹配任何 http 动词。未用于 grpc 调用 |
| 行动 | 字符串 | 访问修饰符。接受的值“允许”（默认值）或“拒绝” |

#### 策略规则

1. 如果未指定访问策略，则默认行为是允许所有应用访问被调用应用上的所有方法
2. 如果未指定全局默认操作且未定义特定于应用的策略，则空访问策略将被视为未指定访问策略，默认行为是允许所有应用访问被调用应用上的所有方法。
3. 如果未指定全局默认操作，但已定义某些特定于应用的策略，则我们采用更安全的选项，即假设全局默认操作来拒绝访问被调用应用上的所有方法。
4. 如果定义了访问策略，并且无法验证传入的应用凭据，则全局默认操作将生效。
5. 如果传入应用程序的信任域或命名空间与应用程序策略中指定的值不匹配，则会忽略应用程序策略，全局默认操作将生效。

#### 策略优先级

匹配的最具体的策略对应的操作按以下顺序生效：

1. 对于 HTTP，则为特定的 HTTP 动词，对于 GRPC 而言，为操作级别操作。
2. 应用级别的默认操作
3. 全局级别的默认操作

#### 示例

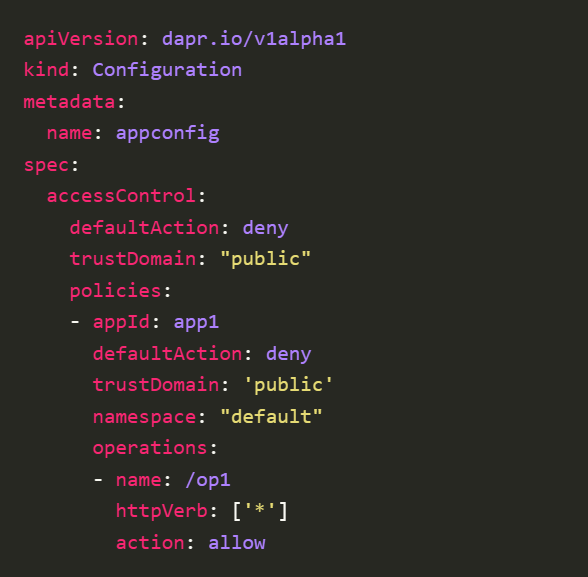
方案 1：拒绝访问所有应用，除非 trustDomain = public、命名空间 = default、appId = app1

使用此配置，将允许使用 appId = app1 的所有调用方法，并拒绝来自其他应用程序的所有其他调用请求



方案 2：拒绝访问除 trustDomain = public、命名空间 = default、appId = app1、操作 = op1 之外的所有应用

使用此配置，仅允许来自 appId = app1 的方法 op1，并且拒绝来自所有其他应用（包括 app1 上的其他方法）的所有其他方法请求

****

### Dapr API 允许列表

#### 默认行为

如果未指定 API 允许列表部分，则默认行为是允许访问所有 Dapr API。设置允许列表后，只能访问指定的 API。

****

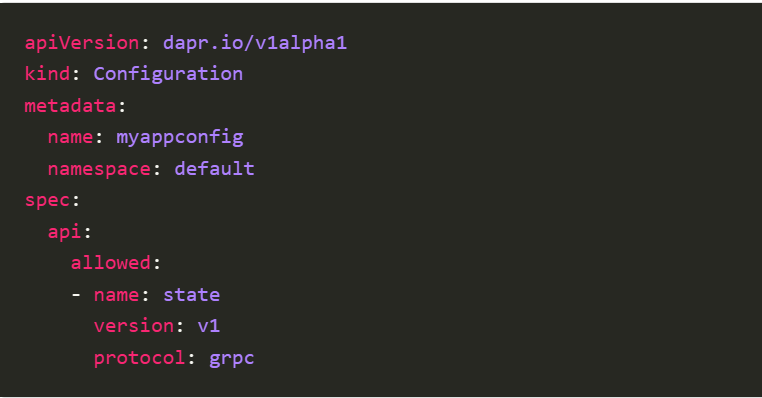
#### 启用特定的 HTTP API

以下示例启用状态 HTTP API 并阻止所有其他 API

****

#### 启用特定的 gRPC API

以下示例启用状态 gRPC API 并阻止所有其他 API

****

#### Dapr API 列表

该字段采用要启用的 Dapr API 的名称。name

请参阅对应于不同 Dapr API 的值列表：

|  |  |
| --- | --- |
| Name | Dapr API |
| state | State |
| invoke | Service Invocation |
| secrets | Secrets |
| bindings | Output Bindings |
| publish | Pub/Sub |
| actors | Actors |
| metadata | Metadata |

### 处理大型 http 正文请求

默认情况下，Dapr 对请求正文大小的限制设置为 4 MB，但可以通过定义注释或标志来更改此设置

在自托管模式下运行时，请使用标志将 Dapr 配置为使用非默认请求正文大小：--dapr-http-max-request-size

dapr run --dapr-http-max-request-size 16 node app.js

该命令告诉 Dapr 将最大请求正文大小设置为 16MB