BAETYL Documentation

BAETYL

2019 年 10 月 31 日

Contents

Baetyl，将计算、数据和服务从中心无缝延伸到边缘。

CHAPTER 1

什么是 Baetyl

Baetyl 是 Linux Foundation Edge 旗下项目，旨在将云计算能力拓展至用户现场，提供临时离线、低延时的计算服务，包括设备接入、消息路由、消息远程同步、函数计算、设备信息上报、配置下发等功能。Baetyl 和智能边缘 BIE（Baidu-IntelliEdge）云端管理套件配合使用，通过在云端进行智能边缘核心设备的建立、存储卷创建、服务创建、函数编写，然后生成配置文件下发至 Baetyl 本地运行包，整体可达到 **边缘计算**、**云端管理**、**边云协同**的效果，满足各种边缘计算场景。

在架构设计上，Baetyl 一方面推行 **模块化**，拆分各项主要功能，确保每一项功能都是一个独立的模块，整体由主程序控制启动、退出，确保各项子功能模块运行互不依赖、互不影响；总体上来说，推行模块化的设计模式，可以充分满足用户 **按需使用**、**按需部署**的切实要求；另一方面，Baetyl 在设计上还采用全面 **容器化**的设计思路，基于各模块的镜像可以在支持 Docker 的各类操作系统上进行 **一键式构建**，依托 Docker 跨平台支持的特性，确保 Baetyl 在各系统、平台的环境一致；此外，Baetyl 还针对 Docker 容器化模式赋予其 **资源隔离与限制**能力，精确分配各运行实例的 CPU、内存等资源，提升资源利用效率。

1.1 优势

- **屏蔽计算框架**：Baetyl 提供主流运行时支持的同时，提供各类运行时转换服务，基于任意语言编写、基于任意框架训练的函数或模型，都可以在 Baetyl 中执行
- **简化应用生产**：智能边缘 BIE云端管理套件配合 Baetyl，联合百度云，一起为 Baetyl 提供强大的应用生产环境，通过 CFC、Infinite、EasyEdge、TSDB、IoT Visualization 等产品，可以在云端轻松生产各类函数、AI 模型，及将数据写入百度云天工云端 TSDB 及物可视进行展示
- **服务按需部署**：Baetyl 推行容器化和模块化，各模块独立运行互相隔离，开发者完全可以根据自己的需求选择部署

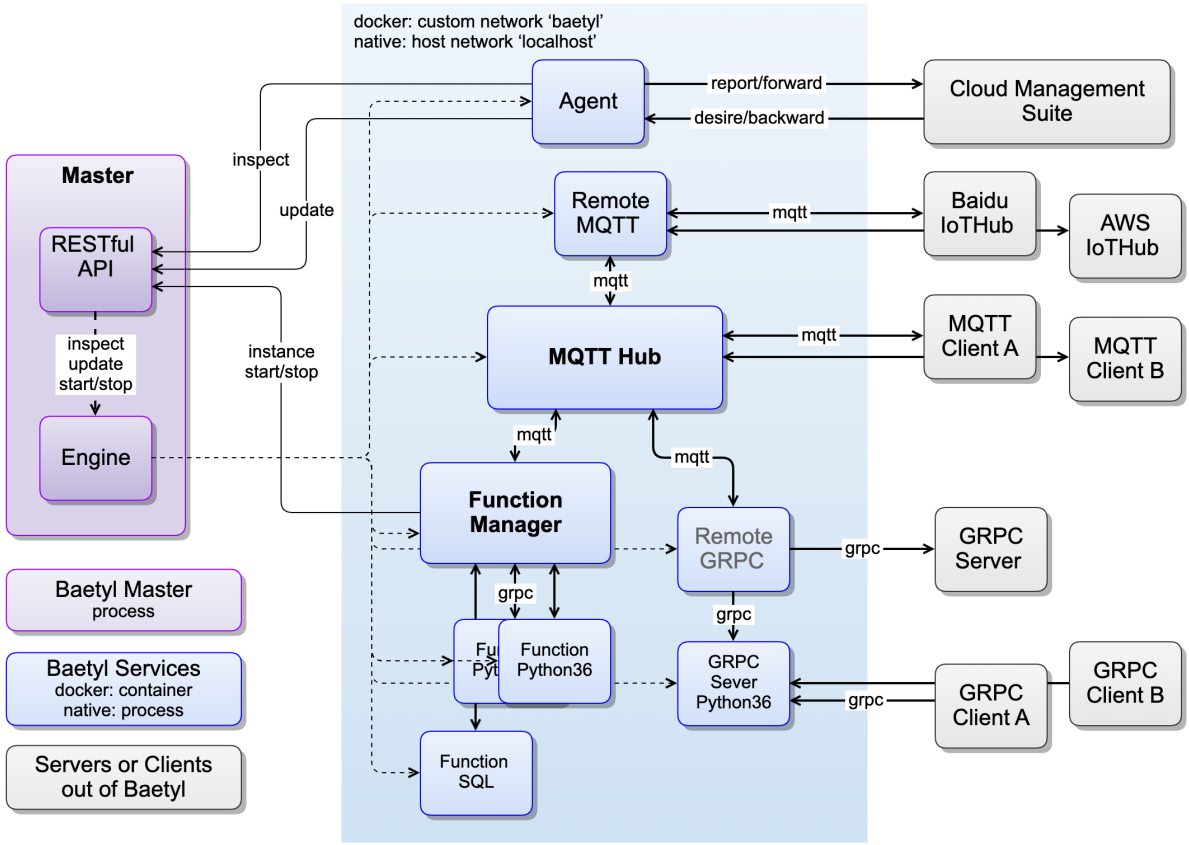
- **支持多种平台**：Baetyl 支持多种软硬件平台，比如 X86 和 ARM 架构的 CPU，Linux 和 Darwin 操作系统等

1.2 组成

Baetyl 作为一个边缘计算框架，除了提供底层服务管理能力外，还提供一些基础功能模块，具体如下：

- Baetyl 主程序 负责服务实例的管理，如启动、退出、守护等，由引擎系统、API、命令行构成。目前支持两种运行模式：Native 进程模式和 Docker 容器模式
- 官方模块 baetyl-agent 负责和 BIE 云端管理套件通讯，可以进行应用下发，设备信息上报等。强制证书认证，保证传输安全；
- 官方模块 baetyl-hub 提供基于 MQTT 协议 的消息订阅和发布功能，支持 4 种接入方式：TCP、SSL、WS 及 WSS；
- 官方模块 baetyl-remote-mqtt 用于桥接两个 MQTT Server 进行消息同步，支持配置多路消息转发；
- 官方模块 baetyl-function-manager 提供基于 MQTT 消息机制，弹性、高可用、扩展性好、响应快的计算能力；
- 官方模块 baetyl-function-python27 提供 Python2.7 函数运行时，可由 `baetyl-function-manager` 动态启动实例；
- 官方模块 baetyl-function-python36 提供 Python3.6 函数运行时，可由 `baetyl-function-manager` 动态启动实例；
- 官方模块 baetyl-function-node85 提供 Node 8.5 函数运行时，可由 `baetyl-function-manager` 动态启动实例；
- SDK (Golang) 可用于开发自定义模块。

1.2.1 架构图



架构图

1.3 快速安装

- 快速安装 *Baetyl*
- 源码安装 *Baetyl*

1.4 开发文档

- Baetyl 设计
- *Baetyl* 配置解读
- 如何针对 *Python* 运行时编写 *Python* 脚本
- 如何针对 *Node* 运行时编写 *Node* 脚本
- 如何针对 *Python* 运行时引入第三方包

- 如何针对 *Node* 运行时引入第三方包
- 如何开发一个自定义函数运行时
- 如何开发一个自定义模块

1.5 如何贡献

如果您热衷于开源社区贡献，Baetyl 将为您提供两种贡献方式，分别是代码贡献和文档贡献。具体请参考 如何向 Baetyl 贡献代码和文档。

1.6 联系我们

Baetyl 作为中国首发的开源边缘计算框架，我们旨在打造一个 **轻量**、**安全**、**可靠**、**可扩展性强**的边缘计算社区，为中国边缘计算技术的发展和不断推进营造一个良好的生态环境。为了更好的推进 Baetyl 的发展，如果您有更好的关于 Baetyl 的发展建议，欢迎选择如下方式与我们联系。

- 欢迎加入 Baetyl 边缘计算开发者社区群
- 欢迎加入 Baetyl 的 LF Edge 讨论组
- 欢迎发送邮件到：baetyl@lists.lfedge.org
- 欢迎到 GitHub 提交 Issue

CHAPTER 2

Baetyl 架构

2.1 概念

- **系统**：这里专指 Baetyl 系统，包行 **主程序**、**服务**、**存储卷**和使用的系统资源。
- **主程序**：指 Baetyl 实现的核心部分，负责管理所有 **存储卷**和 **服务**，内置 **引擎系统**，对外提供 RESTful API 和命令行等。
- **服务**：指一组接受 Baetyl 控制的运行程序集合，用于提供某些具体的功能，比如消息路由服务、函数计算服务、微服务等。
- **实例**：指 **服务**启动的具体的运行程序或容器，一个 **服务**可以启动多个实例，也可以由其他服务负责动态启动实例，比如函数计算的运行时实例就是由函数计算管理服务动态启停的。
- **存储卷**：指被 **服务**使用的目录，可以是只读的目录，比如放置配置、证书、脚本等资源的目录，也可以是可写的目录，比如日志、数据等持久化目录。
- **引擎系统**：指 **服务**的各类运行模式的操作抽象和具体实现，比如 Docker 容器模式和 Native 进程模式。
- **服务**和 **系统**的关系：Baetyl 系统可以启动多个服务，服务之间没有依赖关系，不应当假设他们的启动顺序（虽然当前还是顺序启动的）。服务在运行时产生的所有信息都是临时的，服务停止后这些信息都会被删除，除非映射到持久化目录。服务内的程序由于种种原因可能会停止，服务会根据用户的配置对程序进行重启，这种情况不等于服务的停止，所以信息不会被删除。

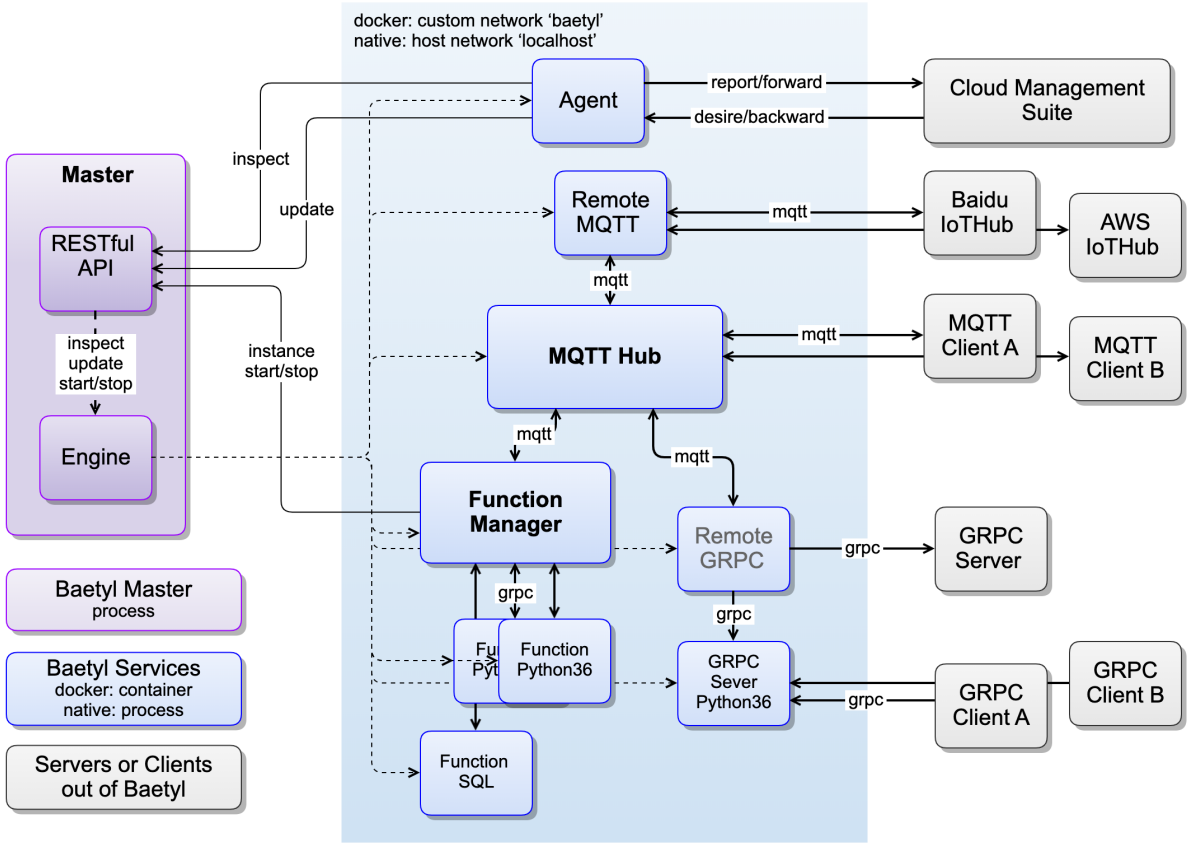
2.2 组成

一个完整的 Baetyl 系统由 **主程序**、**服务**、**存储卷**和使用的系统资源组成，主程序根据应用配置加载各个模块启动相应的服务，一个服务又可以启动若干个实例，所有实例都由主程序负责管理和守护。需要注意的是同一个服务下的实例共享该服务绑定的存储卷，所以如果出现独占的资源，比如监听同一个端口，只能成功启动一个实例；如果使用同一个 Client ID 连接 Hub，会出现连接互踢的情况。

目前 Baetyl 开源了如下几个官方模块：

- *baetyl-agent*：提供 BIE 云代理服务，进行状态上报和应用下发。
- *baetyl-hub*：提供基于 MQTT 的消息路由服务。
- *baetyl-remote-mqtt*：提供 Hub 和远程 MQTT 服务进行消息同步的服务。
- *baetyl-function-manager*：提供函数计算服务，进行函数实例管理和消息触发的函数调用。
- *baetyl-function-python27*：提供加载基于 Python2.7 版本的函数脚本的 GRPC 微服务，可以托管给 baetyl-function-manager 成为函数实例提供方。
- *baetyl-function-python36*：提供加载基于 Python3.6 版本的函数脚本的 GRPC 微服务，可以托管给 baetyl-function-manager 成为函数实例提供方。
- *baetyl-function-node85*：提供加载基于 Node8.5 版本的函数脚本的 GRPC 微服务，可以托管给 baetyl-function-manager 成为函数实例提供方。

架构图:



架构图

2.3 主程序

主程序作为 Baetyl 系统的核心，负责管理所有存储卷和服务，内置运行引擎系统，对外提供 RESTful API 和命令行。

主程序启停过程如下：

1. 执行启动命令：`sudo systemctl start baetyl` 以容器模式启动 Baetyl，然后执行 `sudo systemctl status baetyl` 来查看 `baetyl` 是否正常运行。Darwin 系统下可通过执行 `sudo baetyl start` 在终端前台执行运行 baetyl。

2. 主程序首先会加载工作目录下的 `etc/baetyl/conf.yml`，初始化运行模式、API Server、日志和退出超时时间等，这些配置不会随应用配置下发而改变。如果启动没有报错，会在 `/var/run/` 目录下生成 baetyl.sock（Linux）文件。

3. 然后主程序会尝试加载应用配置 `var/db/baetyl/application.yml`，如果该配置不存在则不启动任何服务，否则加载应用配置中的服务列表和存储卷列表。该文件会随应用配置下发而更新，届时系统会根据新配置重新编排服务。

4. 在启动所有服务前，主程序会先调用 Engine 接口执行一些准备工作，比如容器模式下会先尝试下载所

有服务的镜像。

5. 准备工作完成后，开始顺序启动所有服务，如果服务启动失败则会导致主程序退出。容器模式下会将存储卷映射到容器内部；进程模式下会为每个服务创建各自的临时工作目录，并将存储卷软链到工作目录下，服务退出后临时工作目录会被清理，行为同容器模式。

6. 最后，可操作 `ctrl + c` 退出 baetyl，主程序会同时通知所有服务的实例退出并等待，如果超时则强制杀掉实例。然后清理 baetyl.sock 后退出。

下面是完整的 application.yml 配置字段：

```
// 应用配置
type AppConfig struct {
    // 指定应用配置的版本号
    Version  string        `yaml:"version" json:"version"`
    // 指定应用的所以服务信息
    Services []ServiceInfo `yaml:"services" json:"services" default:"[]"`
    // 指定应用的所以存储卷信息
    Volumes  []VolumeInfo  `yaml:"volumes" json:"volumes" default:"[]"`
}

// 存储卷配置
type VolumeInfo struct {
    // 指定存储卷的唯一名称
    Name     string `yaml:"name" json:"name" validate:"regexp=^[a-zA-Z0-9][a-zA-Z0-9_-]
↪{0\\,63}$"`
    // 指定存储卷在宿主机上的目录
    Path     string `yaml:"path" json:"path" validate:"nonzero"`
}

// 存储卷映射配置
type MountInfo struct {
    // 指定被映射存储卷的名称
    Name     string `yaml:"name" json:"name" validate:"regexp=^[a-zA-Z0-9][a-zA-Z0-9_-]
↪{0\\,63}$"`
    // 指定存储卷在容器内的目录
    Path     string `yaml:"path" json:"path" validate:"nonzero"`
    // 指定存储卷的操作权限，只读或可写
    ReadOnly bool   `yaml:"readonly" json:"readonly"`
}

// 服务配置
type ServiceInfo struct {
```

(下页继续)

(续上页)

```
    // 指定服务的唯一名称
    Name      string            `yaml:"name" json:"name" validate:"regexp=^[a-zA-Z0-9][a-
↪zA-Z0-9_-]{0\\,63}$"`
    // 指定服务的程序地址，通常使用 Docker 镜像名称
    Image     string            `yaml:"image" json:"image" validate:"nonzero"`
    // 指定服务副本数，即启动的实例数
    Replica   int               `yaml:"replica" json:"replica"  validate:"min=0"`
    // 指定服务需要映射的存储卷，将存储卷映射到容器中目录
    Mounts    []MountInfo       `yaml:"mounts" json:"mounts" default:"[]"`
    // 指定服务对外暴露的端口号，用于 Docker 容器模式
    Ports     []string          `yaml:"ports" json:"ports" default:"[]"`
    // 指定服务需要映射的设备，用于 Docker 容器模式
    Devices   []string          `yaml:"devices" json:"devices" default:"[]"`
    // 指定服务程序的启动参数，但不包括 `arg[0]`
    Args      []string          `yaml:"args" json:"args" default:"[]"`
    // 指定服务程序的环境变量
    Env       map[string]string `yaml:"env" json:"env" default:"{}"`
    // 指定服务实例重启策略
    Restart   RestartPolicyInfo `yaml:"restart" json:"restart"`
    // 指定服务单个实例的资源限制，用于 Docker 容器模式
    Resources Resources         `yaml:"resources" json:"resources"`
}
```

2.3.1 引擎系统

引擎系统负责服务的存储卷映射，实例启停、守护等，对服务操作做了抽象，可以实现不同的服务运行模式。根据设备能力的不同，可选择不同的运行模式来启动服务。目前支持了 Docker 容器模式和 Native 进程模式，后续还会支持 k3s 容器模式。

Docker 引擎

Docker 引擎会将服务 Image 解释为 Docker 镜像地址，并通过调用 `Docker Engine` 客户端来启动服务。所有服务使用 `Docker Engine` 提供的自定义网络（默认为 baetyl），并根据 `Ports` 信息来对外暴露端口，根据 `Mounts` 信息来映射目录，根据 `Devices` 信息来映射设备，根据 `Resources` 信息来配置容器可使用的资源，比如 CPU、内存等。服务之间可以直接使用服务名访问，由 Docker 的 DNS Server 负责路由。服务的每个实例对应于一个容器，引擎负责容器的启停和重启。

Native 引擎

在无法提供容器服务的平台（如旧版本的 Windows）上，Native 引擎以裸进程方式尽可能的模拟容器的使用体验。该引擎会将服务 Image 解释为 Package 名称，Package 由存储卷提供，内含服务所需的程序，但这些程序的依赖（如 Python 解释器、Node 解释器、lib 等）需要在主机上提前安装好。所有服务直接使用宿主机网络，所有端口都是暴露的，用户需要注意避免端口冲突。服务的每个实例对应于一个进程，引擎负责进程的启停和重启。

注意：进程模式不支持资源的限制，无需暴露端口、映射设备。

目前，上述两种模式基本实现了配置统一，只遗留了服务地址配置的差异，所以 example 中的配置分成了 native 和 docker 两个目录，但最终会实现统一。

2.3.2 RESTful API

Baetyl 主程序会暴露一组 RESTful API，采用 HTTP/1。在 Linux 系统下默认采用 Unix Domain Socket，固定地址为 `/var/run/baetyl.sock`；其他环境采用 TCP，默认地址为 `tcp://127.0.0.1:50050`。目前接口的认证方式采用简单的动态 Token 的方式，主程序在启动服务时会为每个服务动态生成一个 Token，将服务名和 Token 以环境变量的方式传入服务的实例，实例读取后放入请求的 Header 中发给主程序即可。需要注意的是动态启动的实例是无法获取到 Token 的，因此动态实例无法动态启动其他实例。

对服务实例而言，实例启动后可以从环境变量中获取主程序的 API Server 地址，服务的名称和 Token，以及实例的名称，详见下一节环境变量。

Header 名称如下：

- x-openedge-username：账号名称，即服务名称
- x-openedge-password：账号密码，即动态 Token

下面是目前提供的接口：

- GET /v1/system/inspect 获取系统信息和状态
- PUT /v1/system/update 更新系统和服务
- GET /v1/ports/available 获取宿主机的空闲端口
- PUT /v1/services/{serviceName}/instances/{instanceName}/start 动态启动某个服务的一个实例
- PUT /v1/services/{serviceName}/instances/{instanceName}/stop 动态停止某个服务的某个实例
- PUT /v1/services/{serviceName}/instances/{instanceName}/report 报告服务实例的自定义状态信息

System Inspect

该接口用于获取如下信息和状态：

```
// 采集的所有 Baetyl 系统的信息和状态
type Inspect struct {
    // 异常信息
    Error    string    `json:"error,omitempty"`
    // 采集时间
    Time     time.Time `json:"time,omitempty"`
    // 软件信息
    Software Software  `json:"software,omitempty"`
    // 硬件消息
    Hardware Hardware  `json:"hardware,omitempty"`
    // 服务信息，包括服务名、实例运行状态等
    Services Services  `json:"services,omitempty"`
    // 存储卷信息，包括存储卷名称和版本
    Volumes  Volumes   `json:"volumes,omitempty"`
}

// 软件信息
type Software struct {
    // 宿主机操作系统信息
    OS          string `json:"os,omitempty"`
    // 宿主机 CPU 型号
    Arch        string `json:"arch,omitempty"`
    // Baetyl 工作路径
    PWD string `json:"pwd,omitempty"`
    // Baetyl 服务运行模式
    Mode        string `json:"mode,omitempty"`
    // Baetyl 编译的 Golang 版本
    GoVersion   string `json:"go_version,omitempty"`
    // Baetyl 发布版本
    BinVersion  string `json:"bin_version,omitempty"`
    // Baetyl git 提交版本
    GitRevision string `json:"git_revision,omitempty"`
    // Baetyl 加载的应用配置版本
    ConfVersion string `json:"conf_version,omitempty"`
}

// 硬件信息
type Hardware struct {
    // 宿主机内存使用情况
    MemInfo  *utils.MemInfo  `json:"mem_stats,omitempty"`
```

(下页继续)

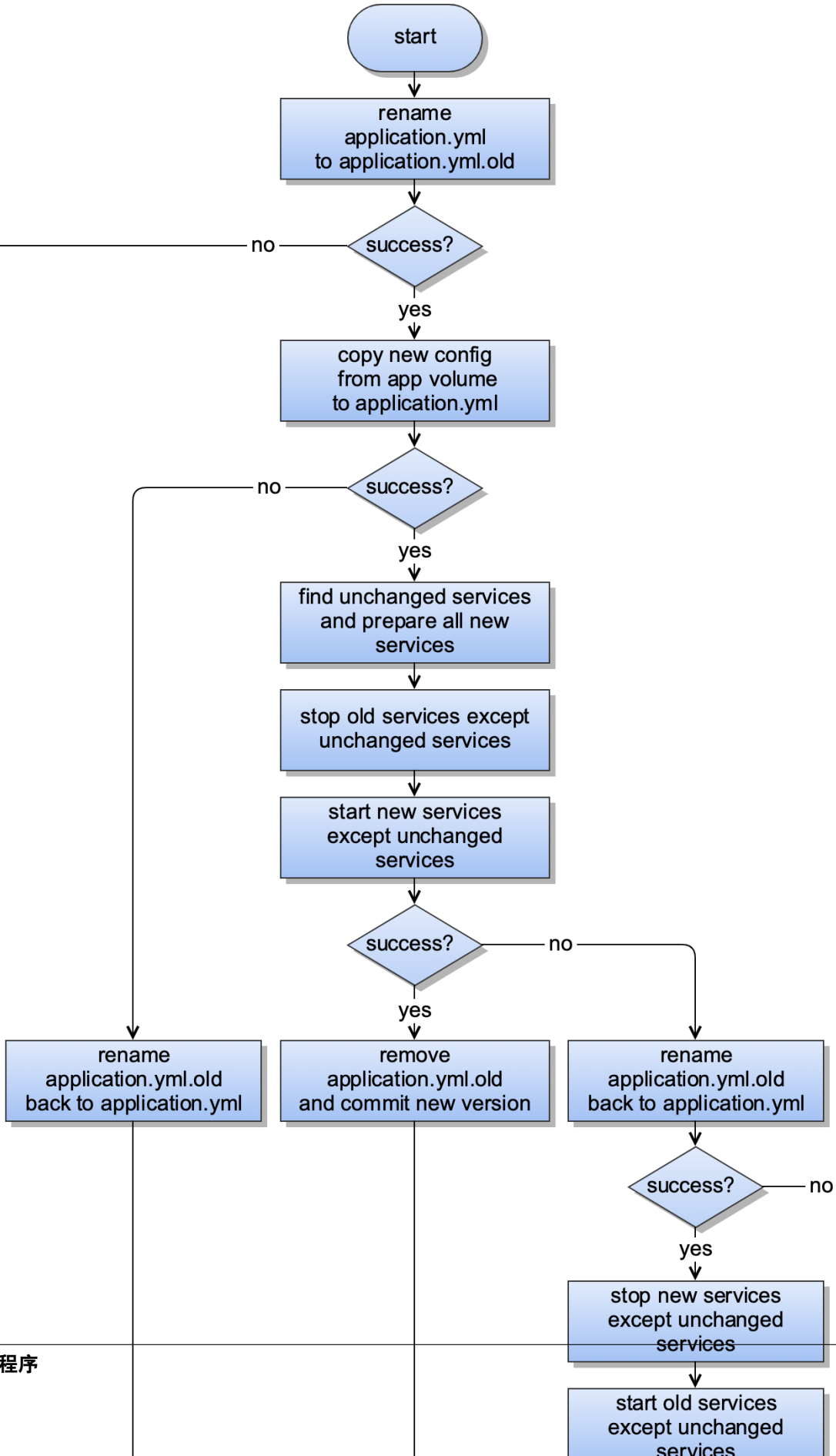
(续上页)

```
    // 宿主机 CPU 使用情况
    CPUInfo  *utils.CPUInfo  `json:"cpu_stats,omitempty"`
    // 宿主机磁盘使用情况
    DiskInfo *utils.DiskInfo `json:"disk_stats,omitempty"`
    // 宿主机 GPU 信息和使用情况
    GPUInfo  []utils.GPUInfo `json:"gpu_stats,omitempty"`
}
```

System Update

该接口用于更新系统中的应用和主程序，我们称之为应用 OTA 和主程序 OTA。其中应用 OTA 会对比配置中的存储卷、网络以及服务自身的配置，如果有变化则会重启服务，如果没有变化则不会重启服务。

一次应用 OTA 的过程如下：


start
rename application.yml to application.yml.old
success?
no
yes
copy new config from app volume to application.yml
success?
no
yes
find unchanged services and prepare all new services
stop old services except unchanged services
start new services except unchanged services
success?
no
yes
rename application.yml.old back to application.yml
remove application.yml.old and commit new version
rename application.yml.old back to application.yml
success?
no
yes
stop new services except unchanged services
start old services except unchanged services

Instance Start&Stop

该接口用于动态启停某个服务的实例，需要指定服务名和实例名，如果重复启动同一个服务的相同名称的实例，会先把之前启动的实例停止，然后启动新的实例。

该接口支持服务的动态配置，用于覆盖存储卷中的静态配置，覆盖逻辑采用环境变量的方式，实例启动时可以加载环境变量来覆盖存储卷中的配置，来避免资源冲突。比如进程模式下，函数计算的管理服务启动函数实例时会预先分配空闲的端口，来使函数实例监听于不同的端口。

Instance Report

该接口用于定时向主程序报告服务实例的自定义状态信息。上报内容放入请求的 Body，采用 JSON 格式，JSON 的第一层字段作为 Key, 多次上报相同的 Key 的值会覆盖。举个例子：

如果服务 infer 的实例 infer 第一次报告如下状态信息，包含 info 和 stats：

```
{
    "info": {
        "company": "baidu",
        "scope": "ai"
    },
    "stats": {
        "msg_count": 124,
        "infer_count": 120
    }
}
```

则 Baetyl 的 Agent 模块后续上报到云端的 JSON 如下：

```
{
    ...
    "time": "0001-01-01T00:00:00Z",
    "services": [
        {
            "name": "infer",
            "instances": [
                {
                    "name": "infer",
                    "start_time": "2019-04-18T16:04:45.920152Z",
                    "status": "running",
                    ...

                    "info": {
```

(下页继续)

(续上页)

```
                        "company": "baidu",
                        "scope": "ai"
                    },
                    "stats": {
                        "msg_count": 124,
                        "infer_count": 120
                    }
                }
            ]
        },
    ]
    ...
}
```

如果服务 infer 的实例 infer 第二次报告如下状态信息，只包含 stats，旧 stats 将被覆盖：

```
{
    "stats": {
        "msg_count": 344,
        "infer_count": 320
    }
}
```

则 Baetyl 的 Agent 模块后续上报到云端的 JSON 如下, 旧 info 被保持，旧 stats 被覆盖：

```
{
    ...
    "time": "0001-01-01T00:00:00Z",
    "services": [
        {
            "name": "infer",
            "instances": [
                {
                    "name": "infer",
                    "start_time": "2019-04-18T16:04:46.920152Z",
                    "status": "running",
                    ...

                    "info": {
                        "company": "baidu",
                        "scope": "ai"
```

(下页继续)

(续上页)

```
                    },
                    "stats": {
                        "msg_count": 344,
                        "infer_count": 320
                    }
                }
            ]
        },
    ]
    ...
}
```

2.3.3 环境变量

Baetyl 目前会给服务实例设置如下几个系统环境变量：

- BAETYL_HOST_OS：Baetyl 所在设备（宿主机）的系统类型
- BAETYL_HOST_ID：Baetyl 所在设备（宿主机）的 HOST ID，可以作为设备的指纹
- BAETYL_MASTER_API_ADDRESS：Baetyl 主程序的 API Server 地址
- BAETYL_MASTER_API_VERSION：Baetyl 主程序的 API 的版本
- BAETYL_SERVICE_MODE：Baetyl 主程序采用的服务运行模式
- BAETYL_SERVICE_NAME：服务的名称
- BAETYL_SERVICE_TOKEN：动态分配的 Token
- BAETYL_SERVICE_INSTANCE_NAME：服务实例的名称，用于动态指定
- BAETYL_SERVICE_INSTANCE_ADDRESS：服务实例的地址，用于动态指定

官方提供的函数计算管理服务就是通过读取 `BAETYL_MASTER_API_ADDRESS` 来连接 Baetyl 主程序的，比如 Linux 系统下 `BAETYL_MASTER_API_ADDRESS` 默认是 `unix:///var/run/baetyl.sock`；其他系统的容器模式下 `BAETYL_MASTER_API_ADDRESS` 默认是 `tcp://host.docker.internal:50050`；其他系统的进程模式下 `BAETYL_MASTER_API_ADDRESS` 默认是 `tcp://127.0.0.1:50050`。

注意：应用中配置的环境变量如果和上述系统环境变量相同会被覆盖。

2.4 官方模块

目前官方提供了若干模块，用于满足部分常见的应用场景，当然开发者也可以开发自己的模块。

2.4.1 baetyl-agent

`baetyl-agent` 又称云代理模块，负责和 BIE 云端管理套件通讯，拥有 MQTT 和 HTTPS 通道，MQTT 强制 SSL/TLS 证书双向认证，HTTPS 强制 SSL/TLS 证书单向认证。开发者可以参考该模块实现自己的 Agent 模块来对接自己的云平台。

云代理目前就做三件事：

1. 启动后定时向主程序获取状态信息并上报给云端
2. 监听云端下发的事件，触发相应的操作，目前只处理应用 OTA 事件
3. 负责清理存储卷目录，存储卷清理期间不会通知主程序进行应用 OTA

云代理接收到 BIE 云端管理套件的应用 OTA 指令后，会先下载所有配置中使用的存储卷数据包并解压到指定位置，如果存储卷数据包已经存在并且 MD5 相同则不会重复下载。所有存储卷都准备好之后，云代理模块会调用主程序的 `/update/system` 接口触发主程序更新系统。

提示：*如果设备无法连接外网或者需要脱离云端管理，可以从应用配置中移除 Agent 模块，离线运行。*

2.4.2 baetyl-hub

`baetyl-hub` 简称 Hub 是一个单机版的消息订阅和发布中心，采用 MQTT3.1.1 协议，可在低带宽、不可靠网络中提供可靠的消息传输服务。其作为 Baetyl 系统的消息中间件，为所有服务提供消息驱动的互联能力。

目前支持 4 种接入方式：TCP、SSL（TCP + SSL）、WS（Websocket）及 WSS（Websocket + SSL），MQTT 协议支持度如下：

- 支持 `Connect`、`Disconnect`、`Subscribe`、`Publish`、`Unsubscribe`、`Ping` 等功能
- 支持 QoS 等级 0 和 1 的消息发布和订阅
- 支持 `Retain`、`Will`、`Clean Session`
- 支持订阅含有 `+`、`#` 等通配符的主题
- 支持符合约定的 ClientID 和 Payload 的校验
- 暂时 **不支持**发布和订阅以 `$` 为前缀的主题
- 暂时 **不支持** Client 的 Keep Alive 特性以及 QoS 等级 2 的发布和订阅

注意：

- 发布和订阅主题中含有的分隔符 `/` 最多不超过 8 个，主题名称长度最大不超过 255 个字符
- 消息报文默认最大长度位 32k，可支持的最大长度为 268,435,455（Byte），约 256 MB，可通过 `message` 配置项进行修改
- ClientID 支持大小写字母、数字、下划线、连字符（减号）和空字符 (如果 CleanSession 为 false 不允许为空)，最大长度不超过 128 个字符

- 消息的 QoS 只能降不能升，比如原消息的 QoS 为 0 时，即使订阅 QoS 为 1，消息仍然以 QoS 为 0 的等级发送
- 如果使用证书双向认证，Client 必须在连接时发送 **非空**的 username 和 **空**的 password ，username 会用于主题鉴权。如果 password 不为空，则还会进一步检查密码是否正确

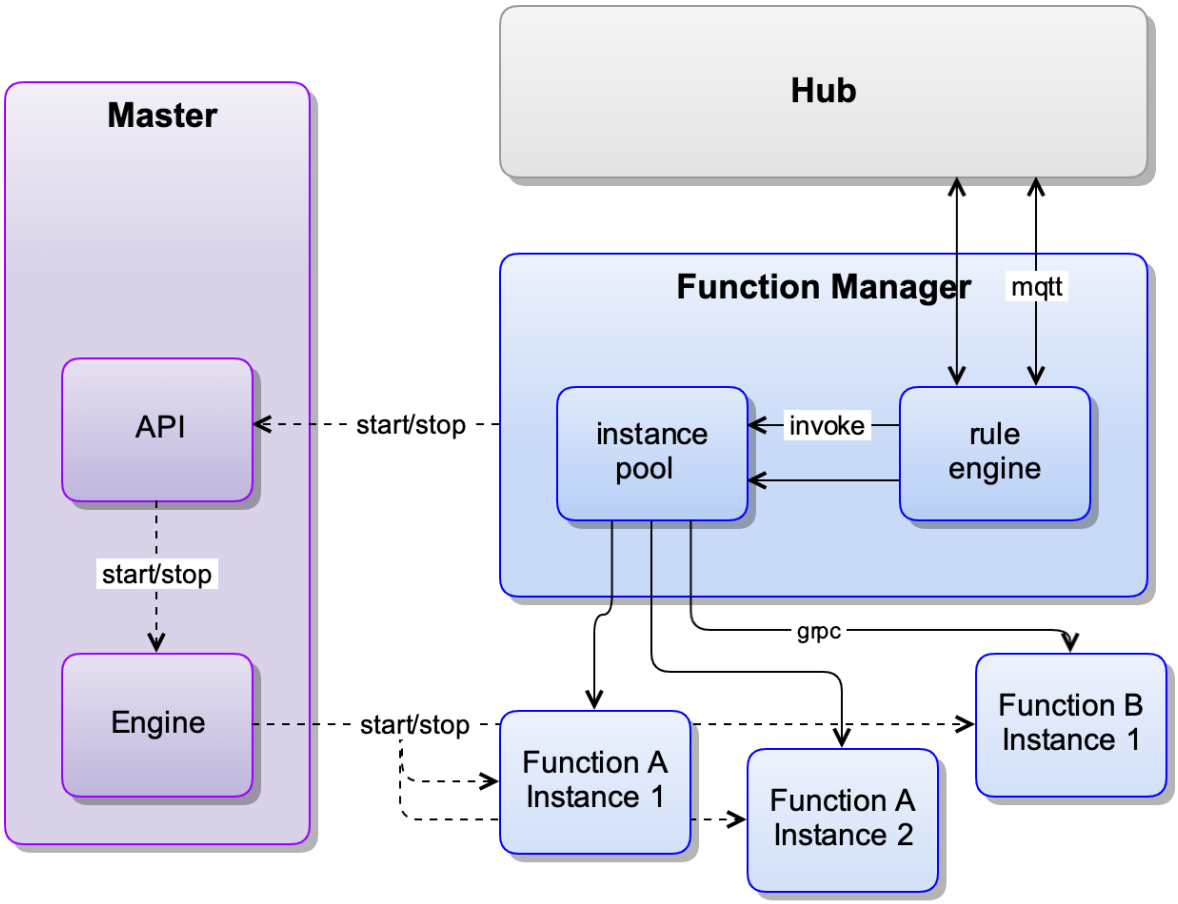
Hub 支持简单的主题路由，比如订阅主题为 `t` 的消息并以新主题 `t/topic` 发布。

如果该模块无法满足您的要求，您也可以使用第三方的 MQTT Broker/Server 来替换。

2.4.3 baetyl-function-manager

`baetyl-function-manager` 又称函数管理模块，提供基于 MQTT 消息机制，弹性、高可用、扩展性好、响应快的的计算能力，并且兼容百度云-函数计算 CFC。**需要注意的是函数计算不保证消息顺序，除非只启动一个函数实例**。

函数管理模块负责管理所有函数实例和消息路由规则，支持自动扩容和缩容。结构图如下：



函数计算服务

如果函数执行错误，函数计算会返回如下格式的消息，供后续处理。其中 `functionMessage` 是函数输入的消息（被处理的消息），不是函数返回的消息。示例如下：

```
{
    "errorMessage": "rpc error: code = Unknown desc = Exception calling application",
    "errorType": "*errors.Err",
    "functionMessage": {
        "ID": 0,
        "QOS": 0,
        "Topic": "t",
        "Payload": "eyJpZCI6MSwiZGV2aWNlIjoiMTExIn0=",
        "FunctionName": "sayhi",
        "FunctionInvokeID": "50f8f102-2b8c-4904-86df-0728811a5a4b"
    }
}
```

2.4.4 baetyl-function-python27

baetyl-function-python27 模块的设计思想与 baetyl-function-python36 模块相同，但是两者的函数运行时不同。baetyl-function-python27 所使用的函数运行时基于 Python2.7 版本，并提供基于 Python2.7 的 pyyaml、protobuf3、grpcio。

2.4.5 baetyl-function-python36

baetyl-function-python36 提供 Python 函数与 百度云-函数计算 CFC 类似，用户通过编写的自己的函数来处理消息，可进行消息的过滤、转换和转发等，使用非常灵活。该模块可作为 GRPC 服务单独启动，也可以为函数管理模块提供函数运行实例。所使用的函数运行时基于 Python3.6 版本。

Python 函数的输入输出可以是 JSON 格式也可以是二进制形式。消息 Payload 在作为参数传给函数前会尝试一次 JSON 解码（`json.loads(payload)`），如果成功则传入字典（dict）类型，失败则传入原二进制数据。

Python 函数支持读取环境变量，比如 os.environ[‘PATH’]。

Python 函数支持读取上下文，比如 context[‘functionName’]。

Python 函数示例如下：

```
#!/usr/bin/env python3
#-*- coding:utf-8 -*-
"""
module to say hi
"""

def handler(event, context):
    """
```

（下页继续）

(续上页)

```
    function handler
    """
    event['functionName'] = context['functionName']
    event['functionInvokeID'] = context['functionInvokeID']
    event['messageQOS'] = context['messageQOS']
    event['messageTopic'] = context['messageTopic']
    event['sayhi'] = '你好，世界！'
    return event
```

提示：*Native 进程模式下，若要运行本代码库 example 中提供的 index.py，需要自行安装 Python3.6，且需要基于 Python3.6 安装 protobuf3、grpcio (采用 pip 安装即可，pip3 install pyyaml protobuf grpcio)*。

2.4.6 baetyl-function-node85

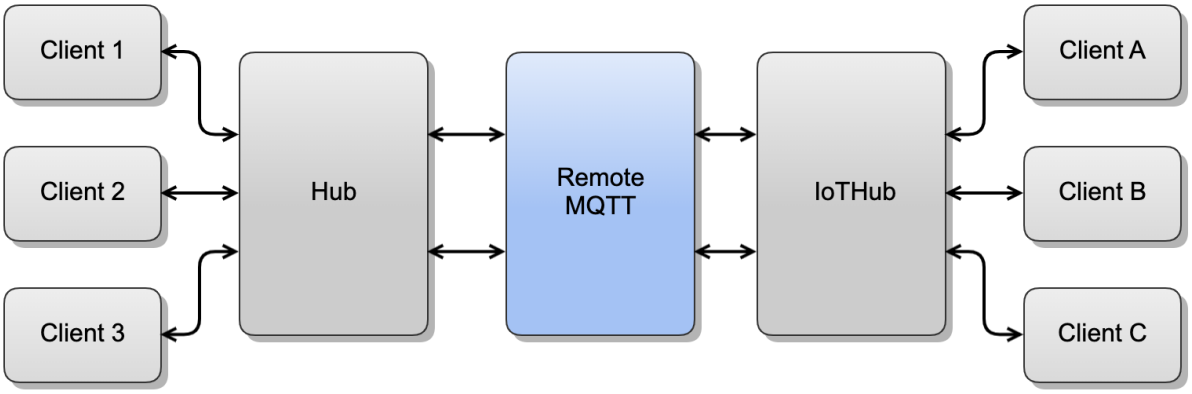
baetyl-function-node85 模块的设计思想与 baetyl-function-python36 模块相同，为 Baetyl 提供 Node8.5 运行时环境，用户可以编写 javascript 脚本来处理消息，同样支持 JSON 格式也可以是二进制形式的数据，javascript 脚本示例如下：

```
#!/usr/bin/env node

exports.handler = (event, context, callback) => {
  result = {};

  if (Buffer.isBuffer(event)) {
      const message = event.toString();
      result["msg"] = message;
      result["type"] = 'non-dict';
  }else {
      result["msg"] = event;
      result["type"] = 'dict';
  }

  result["say"] = 'hello world';
  callback(null, result);
};
```

提示：*Native 进程模式下，若要运行本代码库 example 中提供的 index.js，需要自行安装 Node8.5*。

2.4.7 baetyl-remote-mqtt

`baetyl-remote-mqtt` 又称远程 MQTT 通讯模块，可桥接两个 MQTT Server 进行消息同步。目前支持配置多路消息转发，可配置多个 Remote 和 Hub 同时进行消息同步，结构图如下：



远程 MQTT 通讯举例

如上图示，这里 Baetyl 的本地 Hub 与远程云端 Hub 平台之间通过 Baetyl 远程 MQTT 通讯模块实现消息的转发、同步。进一步地，通过在两端接入 MQTT Client 即可实现 **端云协同式**的消息转发与传递。

CHAPTER 3

如何贡献

欢迎来到 Baetyl 开源边缘计算项目，如果您想要向 Baetyl 贡献代码或文档，请遵循以下流程。

3.1 贡献流程

Baetyl 使用通用的 Git 分支构建模型。下面将为您提供通用的 Github 代码贡献方式。

1. Fork 代码库

   我们的开发社区非常活跃，感兴趣的开发者日益增多，因此，我们鼓励开发者采用 **fork** 方式向我们提交代码。关于如何 fork 一个代码库，请参考 Github 提供的官方帮助页面并点击 "Fork" 按钮.

2. 准备开发环境

   如果您想要向 Baetyl 贡献代码，请参考如下命令准备相关本地开发环境：

   ```
   go get github.com/baetyl/baetyl # 获取 baetyl 代码库
   cd $GOPATH/src/github.com/baetyl/baetyl # 进入 baetyl 代码库目录
   git checkout master  # 校验当前处于 master 主分支
   git remote add fork https://github.com/<your_github_account>/baetyl  # 指定远程提交代码仓库
   ```

3. 提交代码到 fork 仓库

   这里，将改动的需求或修复的 bug 提交到步骤 2 中 fork 的远程仓库，具体请参考如下命令：

```
git status   # 查看当前代码改变状态
git add .
git commit -c "modify description"  # 提交代码到本地仓库，并提交代码改动描述信息
git push fork # 推送已提交本地仓库的代码要远程仓库
```

4. 创建代码合入请求

   基于 fork 的仓库地址直接向 Baetyl 官方仓库 https://github.com/baetyl/baetyl 提交 **pull request**（具体请参考如何创建一个提交请求），即可完成向 Baetyl 官方仓库的代码合入请求。一旦 Baetyl 代码仓库评审人员通过了您的代码提交、合入请求，您即可在 Baetyl 官方代码仓库中看到您贡献的代码。

3.2 代码评审规范

- Golang 的代码风格请参照 Go Code Review Comments
- 请在代码 CI 测试通过后及时通过 Email 向你的代码评审人发送代码提交请求 URL
- 请及时回答评审人的每一个 comment，如果您采纳评审人给出的建议，请直接回复 **好的**或是 **Done**；如果您不同意，请给出您的理由
- 如果您不想您的代码评审人被邮件通知频繁打扰，您可以通过 **交互框**回复评审人提出的每一个建议，具体请参考 如何使用交互框回复评审人信息
- 尽可能减少不必要的代码提交。一些开发者总是频繁提交代码。如果您想要向提交的代码中增加一个微小的改动，请使用命令 `git commit --amend` 代替 `git commit`

3.3 代码合入规范

无规矩不成方圆。这里规定，凡是提交 Baetyl 代码合入请求的代码，一律要求遵循以下规范：

- 建议您提交代码前再次执行命令 `govendor fmt +local`，具体请参考 govendor
- 代码提交前 **必须**进行单元测试（提交代码应包含）和竞争检测，参考执行命令 `make test`
- 仅有提交代码通过单元测试和竞争检测，才允许向 Baetyl 官方仓库提交
- 所有向 Baetyl 官方仓库提交的代码，**必须至少**有 **1** 个代码评审员评审通过后，才可以将提交代码合入 Baetyl 官方代码仓库

注意：以上所有代码提交步骤要求及规范，同样适用文档贡献。

CHAPTER 4

快速安装 Baetyl

Baetyl 快速安装支持的系统有: Ubuntu16.04、Ubuntu18.04、Debian9、CentOS7、Raspbian、Darwin，支持的平台有 amd64、i386、armv7l 和 arm64。

Baetyl 支持两种运行模式，分别是 **docker** 容器模式和 **native** 进程模式。本文基于 **docker** 容器模式进行快速安装。

4.1 安装 Docker

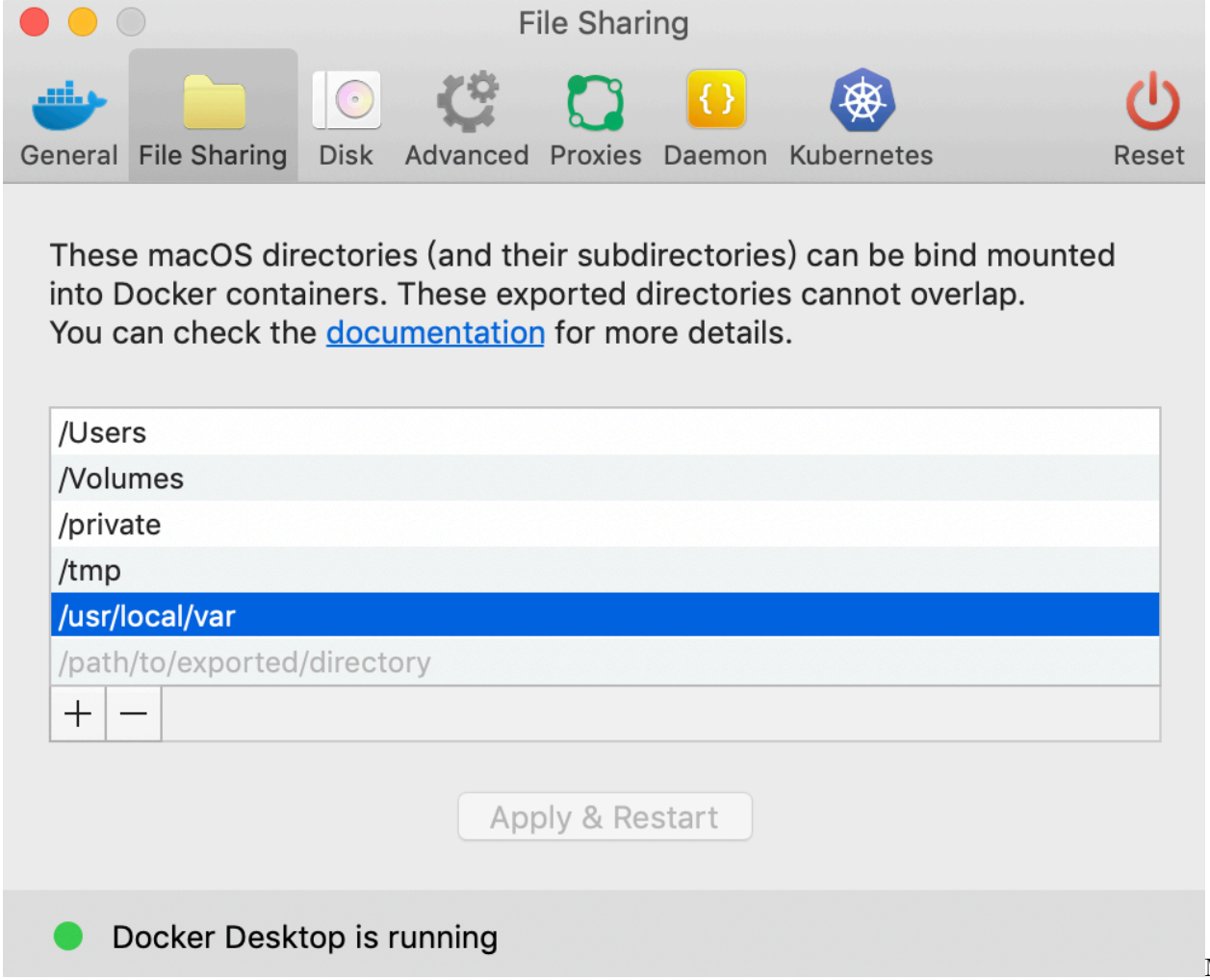
在 **docker** 容器模式下，Baetyl 依赖于 Docker Engine。如果你的机器尚未安装 Docker，可通过以下命令安装 Docker 的最新版本（适用于类 Linux 系统）：

```
curl -sSL https://get.docker.com | sudo sh
```

对于 CentOS7 系统，在安装结束后需要手动启动 Docker:

```
# 设置 Docker 开机自启动
sudo systemctl enable docker
# 启动 Docker
sudo systemctl start docker
```

在 Darwin 系统上安装 Docker 可参考 docker.com/install。另外需要设置 Baetyl 使用到的 `/usr/local/var` 目录，允许挂载到容器内。



Mount path on Mac

安装结束后，你可以查看 Docker 的版本号:

```
docker version
```

注意：建议安装/更新 Docker 版本到 19.03 及以上。

更多内容请参考 官方文档。

4.2 安装 Baetyl

Baetyl 发布新版本的同时，也会发布对应的二进制文件以及相应的 rpm、deb 包。使用以下命令，可以将 Baetyl 快速安装到设备上:

```
curl -sSL http://dl.baetyl.io/install.sh | sudo bash
```

执行完毕后，Baetyl 将会被安装到 /usr/local/bin 目录下。Baetyl 的运行配置存放在 /usr/local/etc/baetyl

和 /usr/local/var/db/baetyl 目录下，具体的配置方法可以参考配置文件解读文档。

4.3 导入示例配置包（可选）

作为一个边缘计算框架，Baetyl 通过 hub 模块提供基于 MQTT 的消息路由服务，通过 function-manager 模块和 python27、python36、nodejs85、sql 等运行时模块来提供本地函数计算服务。你可以通过编辑配置文件，来让 baetyl 主程序加载相应的模块以及设定模块本身的运行参数。关于各个模块的配置详情，可参考配置解读 中的内容进行进一步了解。

Baetyl 官方提供了一套示例配置，你可以通过以下命令导入示例配置文件:

```
# Darwin OS use Docker as a non-root user. So we don't run this script with "sudo"␣
↪directly.
curl -O http://dl.baetyl.io/install_with_docker_example.sh && bash install_with_docker_
↪example.sh
```

上述脚本将会检测当前系统上是否存在历史配置文件，根据提示选择**是否**删除历史配置文件。你也可以根据脚本中的提示选择**是否**提前拉取示例配置中要用到的各个模块的镜像。

示例配置只用于学习和测试目的，应根据自己的实际工作场景按需配置。

如果你不需要启动任何模块，那就不需要导入任何配置文件。

4.4 启动 Baetyl

采用快速安装方式后，你可以通过以下命令来启动 Baetyl:

```
# Linux
sudo baetyl start
# Darwin
baetyl start
```

在 Linux 系统上，你可以使用 Systemd 来管理 Baetyl 的启动（start）、停止（stop）和查看状态（status）：

启动 Baetyl:

```
sudo systemctl start baetyl
```

停止 Baetyl:

```
sudo systemctl stop baetyl
```

查看运行状态:

```
sudo systemctl status baetyl
```

如果之前安装过 Baetyl 或者导入了新的配置文件，请重启 Baetyl:

```
sudo systemctl restart baetyl
```

在 Darwin 系统上，你可以使用 Launchctl 来管理 Baetyl 的启动（load）、停止（unload）：

启动 Baetyl:

```
ln -sfv /usr/local/etc/baetyl/baetyl.plist ~/Library/LaunchAgents/
launchctl load ~/Library/LaunchAgents/baetyl.plist
```

停止 Baetyl:

```
launchctl unload ~/Library/LaunchAgents/baetyl.plist
```

4.5 验证是否成功安装

快速安装 Baetyl 以后，你可以依据以下步骤验证 Baetyl 是否启动成功：

- Linux 系统上在终端中命令 `sudo systemctl status baetyl` 来查看 `baetyl` 是否正常运行。正常如下图所示，否则说明主程序 `baetyl` 启动失败；

```
baetyl@baetyl:~$ sudo systemctl status baetyl
● baetyl.service - Baetyl
   Loaded: loaded (/lib/systemd/system/baetyl.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2019-09-23 11:31:57 CST; 5h 55min ago
 Main PID: 997 (baetyl)
    Tasks: 10 (limit: 4623)
   CGroup: /system.slice/baetyl.service
           └─997 /usr/local/bin/baetyl start

9月 23 11:31:57 baetyl systemd[1]: Started Baetyl.
```

Baetyl

- 在 Linux 或者 Darwin 系统上，均可在终端中执行命令 `docker stats` 查看各个模块的 Docker 容器是否正常运行。如果事先已经拉取了各个模块的镜像，各个模块的容器可以很快运行起来。如果未事先拉取，主程序 `baetyl` 会先到镜像仓库拉取需要的镜像，你需要等待 2~5 分钟执行此条命令。以上一步中导入的示例配置为例，待主程序拉取完成后，容器的运行状态如下图所示。如果本地的镜像与下述不一致，说明模块启动失败；

```
CONTAINER ID        NAME                                        CPU %               MEM USAGE / LIMIT   MEM %               NET I/O             BLOCK I/O           PIDS
10131b874143        function-node85-sayhi.node85-sayhi.1        0.00%               0B / 0B             0.00%               72.5kB / 58.7kB     0B / 0B             0
ff1f55107a12        function-sql-filter.sql-filter.1            0.00%               0B / 0B             0.00%               75.8kB / 43.1kB     2.23MB / 0B         0
781838b42c4b        function-python36-sayhi.python36-sayhi.1    0.09%               0B / 0B             0.00%               72.5kB / 60.6kB     28.7kB / 0B         0
0618c0c49e45        function-python27-sayhi.python27-sayhi.1    0.09%               0B / 0B             0.00%               72.1kB / 60.4kB     1.72MB / 0B         0
2a29e8e17192        timer                                       0.00%               0B / 0B             0.00%               29kB / 13kB         135kB / 0B          0
e8c36a1d92aa        function-manager                            0.00%               0B / 0B             0.00%               321kB / 335kB       922kB / 0B          0
23b8aa709fee        localhub                                    13.05%              0B / 0B             0.00%               153kB / 87.6kB      229kB / 156kB       0
```

当前运行 docker 容器查询

- 针对上述两种失败情况，需要查看主程序日志来了解具体的错误情况。主程序日志的默认存放位置为 `/usr/local/var/log/baetyl/baetyl.log`。针对日志中出现的错误，可参考常见问题 进行解决。必要时可以直接 提交 Issue。

CHAPTER 5

源码安装 Baetyl

相比于快速安装 Baetyl，你可以采用源码编译的方式来使用 Baetyl 最新的功能。

5.1 准备工作

- 安装 Golang 工具并启用 Modules 管理

Golang 的最低版本要求为 1.12。下载安装 Golang 可参考 golang.org 或者 golang.google.cn。我们现在采用 Go Modules 来管理依赖包，可以参考 goproxy.io 来启用。

- 安装 Docker Engine 并打开 Buildx 功能

Docker 的最低版本要求为 19.03，因为从该版本开始内置了 Buildx 功能，可用于制作多平台的镜像。安装 Docker 可参考 docker.com/install，打开 Buildx 功能参考 github.com/docker/buildx 。

5.2 下载代码

从 Baetyl Github 下载代码，执行如下命令：

```
go get github.com/baetyl/baetyl
```

5.3 编译 Baetyl 和模块

进入 Baetyl 项目目录，执行 make 即可编译出当前系统平台的 Baetyl 主程序和模块程序。

```
cd $GOPATH/src/github.com/baetyl/baetyl
# 当前平台，所有模块
make # make all
```

上述命令执行完后，Baetyl 主程序和模块程序会生成在项目的 output 目录下，会按照平台分开放置。

如果需要指定平台和模块，参考如下命令：

```
# 所有平台，所有模块
make PLATFORMS=all
# 指定的平台，指定的模块
make PLATFORMS="linux/amd64 linux/arm64" MODULES="agent hub"
```

如下命令可以重新编译 Baetyl 主程序和模块程序。

```
# 当前平台，所有模块
make rebuild
# 所有平台，所有模块
make rebuild PLATFORMS=all
# 指定的平台，指定的模块
make rebuild PLATFORMS="linux/amd64 linux/arm64" MODULES="agent hub"
```

注意: 上述编译命令会自动读取 Git 的 revision 和 tag 作为程序的版本，所以在执行命令前请先提交或者移除本地的修改。

5.3.1 制作模块镜像

如果使用容器模式运行 Baetyl，我们推荐使用正式发布的官方镜像。如果你想自己制作镜像，可以使用下面提供的命令，但是前提是打开了最前面的准备工作中提到的 Buildx 功能。

进入 Baetyl 项目目录，执行 make image 即可生成当前系统平台的模块镜像。

```
cd $GOPATH/src/github.com/baetyl/baetyl
# 当前平台，所有模块
make image
# 当前平台，指定的模块
make image MODULES="agent hub"
```

执行结束后，你可以执行 docker images 来查看生成的镜像。

```
docker images

REPOSITORY                TAG                 IMAGE ID            CREATED             ␣
↪SIZE
baetyl-function-python3   git-e8fe527         12b669a36a9c        54 minutes ago      ␣
↪202MB
baetyl-function-python2   git-e8fe527         278e5c465e17        About an hour ago   ␣
↪162MB
baetyl-hub                git-e8fe527         abd5bef8ba92        2 hours ago          16.
↪9MB
baetyl-agent              git-e8fe527         7ac8dfecdb63        2 hours ago         ␣
↪18MB
baetyl-function-manager   git-e8fe527         e6564cd87768        2 hours ago          16.
↪7MB
baetyl-remote-mqtt        git-e8fe527         0daa114b968d        2 hours ago         ␣
↪16MB
baetyl-timer              git-e8fe527         88a408e4512a        2 hours ago         ␣
↪16MB
baetyl-function-node8     git-e8fe527         d7bf1abb6d24        4 days ago          ␣
↪221MB
```

如果你需要制作多平台的镜像，你必须指定镜像发布的仓库地址和 push 参数，因为目前 Buildx 不支持将镜像的 manifest 加载到本地的镜像空间。

```
# 所有平台，所有模块
make image PLATFORMS=all XFLAGS=--push REGISTRY=<your docker image register>/
# 指定的平台，指定的模块
make image PLATFORMS="linux/amd64 linux/arm64" MODULES="agent hub" XFLAGS=--push␣
↪REGISTRY=<your docker image register>/
```

5.3.2 安装 Baetyl 和示例

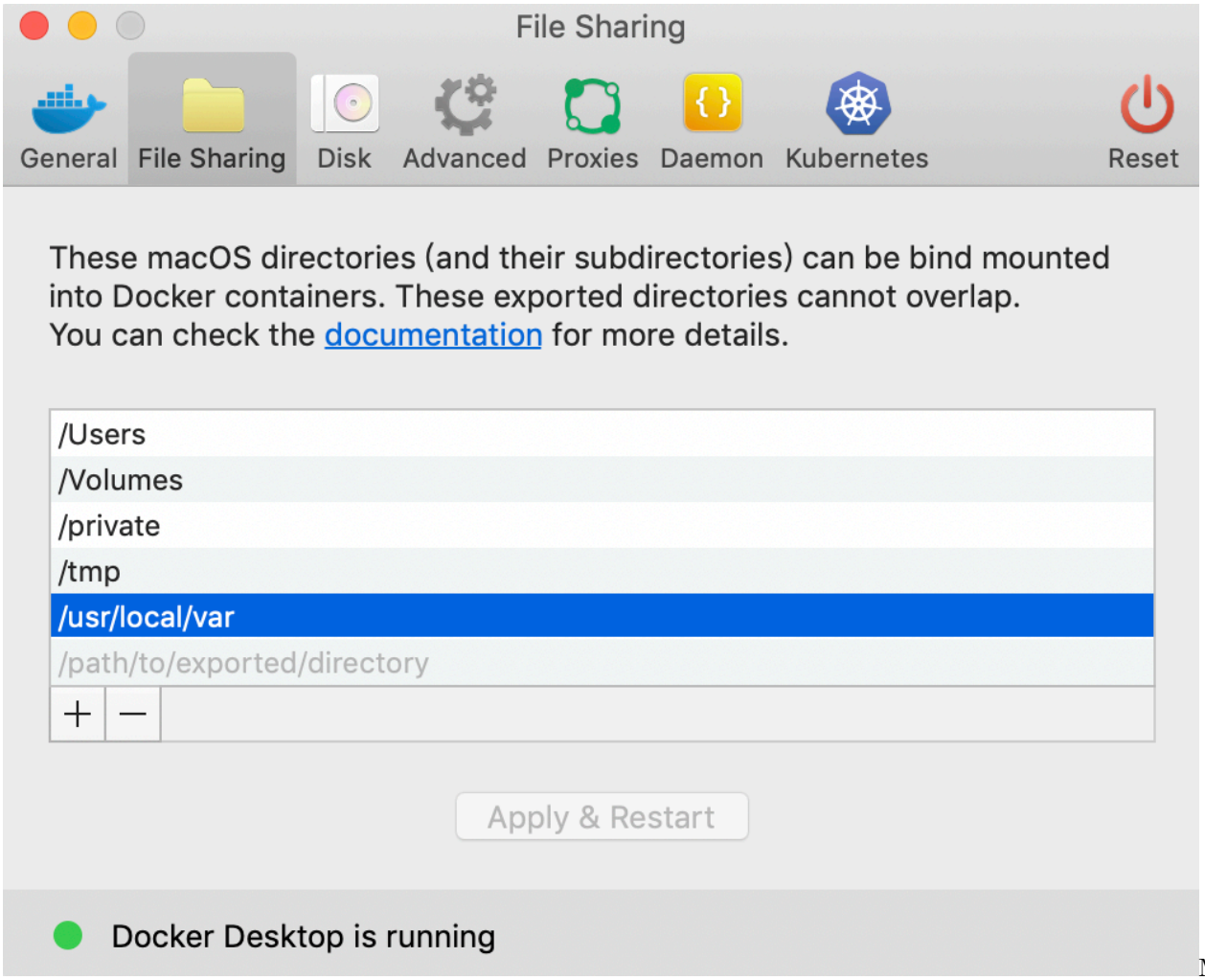
使用如下命令安装 Baetyl 和示例，默认安装到 `/usr/local`。

```
cd $GOPATH/src/github.com/baetyl/baetyl
sudo make install # 会同时安装 docker 容器模式的示例
sudo make install MODE=native # 会同时安装 native 进程模式的示例
```

你也可以指定安装的目录，例如项目下的 `output`：

```
cd $GOPATH/src/github.com/baetyl/baetyl
make install PREFIX=output # 会同时安装 docker 容器模式的示例
make install MODE=native PREFIX=output # 会同时安装 native 进程模式的示例
```

在 Darwin 平台上，需要设置 Baetyl 使用到的 /usr/local/var 目录，允许挂载到容器内。



Mount path on Mac

5.3.3 运行 Baetyl 和示例

如果 Baetyl 和示例已经安装到默认路径：/usr/local：

```
sudo baetyl start
```

如果 Baetyl 和示例已经安装到了指定路径，例如项目下的 output：

```
sudo ./output/bin/baetyl start
```

注意：启动方式根据安装方式的不同而不同，即，若选择 **docker** 运行模式安装，则上述命令会以 **docker** 容器模式运行 Baetyl。

提示：

- baetyl 启动后，可通过 `ps -ef | grep "baetyl"` 查看 baetyl 是否已经成功运行，并确定启动时所使用的参数。通过查看日志确定更多信息，日志文件默认存放在工作目录下的 `var/log/baetyl` 目录中。
- **docker** 容器模式运行，可通过 `docker ps` 或者 `docker stats` 命令查看容器运行状态。
- 如需使用自己的镜像，需要修改应用配置中的模块和函数的 `image`，指定自己的镜像。
- 如需自定义配置，请按照配置解读 中的内容进行相关设置。

5.3.4 卸载

如果 Baetyl 安装在默认目录 `/usr/local`：

```
sudo make uninstall
```

如果 Baetyl 安装在指定目录下，例如项目下的 `output`：

```
make uninstall PREFIX=output
```

CHAPTER 6

Baetyl 配置文件释义

单位统一说明：

- 大小
  - b：字节（byte）
  - k：千字节（kilobyte）
  - m：兆字节（megabyte）
  - g：吉字节（gigabyte）
- 时间
  - s：秒
  - m：分
  - h：小时

配置示例可参考baeyle 项目中 `example` 目录下的例子。

6.1 主程序配置

主程序配置和应用配置是分离的，主程序配置的默认配置文件是工作目录下的 `etc/baetyl/conf.yml`，配置释义如下：

```
mode: 默认值：docker，服务运行模式。docker：容器模式；native：进程模式
grace: 默认值：30s，服务优雅退出超时时间
server: 主程序 API Server 配置项
  address: 默认值可读取环境变量：BAETYL_MASTER_API_ADDRESS，主程序 API Server 地址
  timeout: 默认值：30s，主程序 API Server 请求超时时间
snfile: 设备序列号文件，可以作为设备指纹。如果设置，可读取环境变量：`BAETYL_HOST_SN` 获取内容
docker:
  api_version: 默认值：1.38，客户端调用 Docker Engine 服务接口的版本
logger: 日志配置项
  path: 默认为空，即不打印到文件；如果指定文件则输出到文件
  level: 默认值：info，日志等级，支持 debug、info、warn 和 error
  format: 默认值：text，日志打印格式，支持 text 和 json
  age:
    max: 默认值：15，日志文件保留的最大天数
  size:
    max: 默认值：50，日志文件大小限制，单位 MB
  backup:
    max: 默认值：15，日志文件保留的最大数量
```

6.2 应用配置

应用配置的默认配置文件是工作目录下的 var/db/baetyl/application.yml，配置释义如下：

```
version: 应用版本
services: 应用的服务列表
  - name: [必须] 服务名称，在服务列表中必须唯一
    image: [必须] 服务入口。Docker 容器模式下表示服务镜像；Native 进程模式下表示服务运行包所在位置
    replica: 默认为 0，服务副本数，表示启动的服务实例数。通常服务只需启动一个。函数运行时服务一般设置为 0，不由主程序启动，而是由函数管理服务来动态启动实例
    mounts: 存储卷映射列表
      - name: [必须] 存储卷名称，对应存储卷列表中的一个
        path: [必须] 存储卷映射到容器中的路径
        readonly: 默认值：false，存储卷是否只读
    ports: Docker 容器模式下暴露的端口，例如：
      - 0.0.0.0:1883:1883
      - 0.0.0.0:1884:1884/tcp
      - 8080:8080/tcp
      - 9884:8884
```

(下页继续)

(续上页)

```
    devices: Docker 容器模式下的设备映射，例如：
      - /dev/video0
      - /dev/sda:/dev/xvdc:r
    args: 服务实例启动参数，例如：
      - '-c'
      - 'conf/conf.yml'
    env: 服务实例环境变量，例如：
      version: v1
    restart: 服务实例重启策略配置项
      retry:
        max: 默认为空，表示总是重试，服务重启最大次数
      policy: 默认值：always，重启策略。no：不重启；always：总是重启；on-failure：服务异常退出就重启
      backoff:
        min: 默认值：1s，重启最小间隔时间
        max: 默认值：5m，重启最大间隔时间
        factor: 默认值：2，重启间隔增大倍数
    resources: Docker 容器模式下的服务实例资源限制配置项
      cpu:
        cpus: 服务实例可用的 CPU 比例，例如：1.5，表示可以用 1.5 个 CPU 内核
        setcpus: 服务实例可用的 CPU 内核，例如：0-2，表示可以使用第 0 到 2 个 CPU 内核；0，表示可以使用第 0 个 CPU 内核；1，表示可以使用第 1 个 CPU 内核
      memory:
        limit: 服务实例可用的内存，例如：500m，表示可以用 500 兆内存
        swap: 服务实例可用的交换空间，例如：1g，表示可以用 1G 内存
      pids:
        limit: 服务实例可创建的进程数
volumes: 存储卷列表
  - name: [必须] 存储卷名称，在存储卷列表中唯一
    path: [必须] 存储卷在宿主机上的路径，相对于主程序的工作目录而言
```

6.3 模块配置

模块配置的默认配置文件是工作目录下的：/etc/baetyl/service.yml

6.3.1 baetyl-agent 模块

```
remote: Agent 模块对接 BIE 云端管理套件的配置项
  mqtt: BIE 云端 MQTT 通道配置
    clientid: [必须] 连接云端 MQTT 通道的 Client ID, 必须是云端核心设备的 ID
    address: [必须] 云端 MQTT 通道的的地址, 必须使用 SSL Endpoint
    username: [必须] 云端 MQTT 通道连接的用户名, 必须是云端核心设备的用户名
    ca: [必须] 云端 MQTT 通道连接的 CA 证书路径
    key: [必须] 云端 MQTT 通道连接的客户端私钥路径
    cert: [必须] 云端 MQTT 通道连接的客户端公钥路径
    timeout: 默认值: 30s, 云端 MQTT 通道连接超时时间
    interval: 默认值: 1m, 云端 MQTT 通道重连的最大间隔时间, 从 500 微秒翻倍增加到最大值
    keepalive: 默认值: 10m, 云端 MQTT 通道连接的保持时间
    cleansession: 默认值: false, 云端 MQTT 通道连接的的是否保持 Session
    validatesubs: 默认值: false, 云端 MQTT 通道连接是否检查订阅结果。 如果为 true 发现订阅失
败报错退出
    buffersize: 默认值: 10, 发送消息内存队列大小, 异常退出会导致消息丢失
  http: BIE 云端 HTTP 通道配置
    address: 会自动根据 MQTT 通道的地址推断云端 HTTPS 通道地址, 无需配置
    timeout: 默认值: 30s, 云端 HTTPS 通道连接超时时间
  report: Agent 上报云端配置
    url: 上报的 URL, 无需配置
    topic: 上报主题模板, 无需配置
    interval: 默认值: 20s, 上报间隔时间
  desire: Agent 接收云端下发配置
    topic: 下发主题模板, 无需配置
```

6.3.2 baetyl-hub 模块

```
listen: [必须] 监听地址, 例如:
  - tcp://0.0.0.0:1883
  - ssl://0.0.0.0:1884
  - ws://:8080/mqtt
  - wss://:8884/mqtt
certificate: SSL/TLS 证书认证配置项, 如果启用 ssl 或 wss 必须配置
  ca: Server 的 CA 证书路径
  key: Server 的服务端私钥路径
  cert: Server 的服务端公钥路径
principals: 接入权限配置项, 如果不配置则 Client 无法接入, 支持账号密码和证书认证
  - username: Client 接入 Hub 用户名
```

(下页继续)

(续上页)

```
    password: Client 接入 Hub 密码
    permissions:
      - action: 操作权限。pub: 发布权限; sub: 订阅权限
        permit: 操作权限允许的主题列表, 支持 + 和# 匹配符
  - username: Client 接入 Hub 用户名, 使用证书双向认证可不配置密码
    permissions:
      - action: 操作权限。pub: 发布权限; sub: 订阅权限
        permit: 操作权限允许的主题列表, 支持 + 和# 匹配符
subscriptions: 主题路由配置项
  - source:
      topic: 订阅的主题
      qos: 订阅的 QoS
    target:
      topic: 发布的主题
      qos: 发布的 QoS
message: 消息相关的配置项
  length:
    max: 默认值: 32k; 最大值: 268,435,455 字节 (约 256MB), 可允许传输的最大消息长度
  ingress: 消息接收配置项
    qos0:
      buffer:
        size: 默认值: 10000, 可缓存到内存中的 QoS 为 0 的消息数, 增大缓存可提高消息接收的性
能, 若设备掉电, 则会直接丢弃 QoS 为 0 的消息
    qos1:
      buffer:
        size:  默认值: 100, 等待持久化的 QoS 为 1 的消息缓存大小, 增大缓存可提高消息接收的性
能, 但潜在的风险是 Hub 异常退出 (比如设备掉电) 会丢失缓存的消息, 不回复确认 (puback)。Hub 正常
退出会等待缓存的消息处理完, 不会丢失数据。
      batch:
        max:  默认值: 50, 批量写 QoS 为 1 的消息到数据库 (持久化) 的最大条数, 消息持久化成功
后会回复确认 (ack)
      cleanup:
        retention:  默认值: 48h, QoS 为 1 的消息保存在数据库中的时间, 超过该时间的消息会在清
理时物理删除
        interval:  默认值: 1m, QoS 为 1 的消息清理时间间隔
  egress: 消息发送配置项
    qos0:
      buffer:
        size:  默认值: 10000, 内存缓存中的待发送的 QoS 为 0 的消息数, 若设备掉电会直接丢弃消
息; 缓存满后, 新推送的消息直接丢弃
```

(下页继续)

(续上页)

```
    qos1:
      buffer:
        size: 默认值：100，QoS 为 1 的消息发送后，未确认（ack）的消息缓存大小，缓存满后，不再读取新消息，一直等待缓存中的消息被确认。QoS 为 1 的消息发送给客户端成功后等待客户端确认(puback)，如果客户端在规定时间内没有回复确认，消息会一直重发，直到客户端回复确认或者 session 关闭
      batch:
        max: 默认值：50，批量从数据库读取消息的最大条数
      retry:
        interval: 默认值：20s，消息重发时间间隔
  offset: 消息序列号持久化相关配置
    buffer:
      size: 默认值：10000，被确认（ack）的消息的序列号的缓存队列大小。比如当前批量发送了 QoS 为 1 且序列号为 1、2 和 3 的三条消息给客户端，客户端确认了序列号 1 和 3 的消息，此时序列号 1 会入列并持久化，序列号 3 虽然已经确认，但是还是得等待序列号 2 被确认入列后才能入列。该设计可保证 Hub 异常重启后仍能从持久化的序列号恢复消息处理，保证消息不丢，但是会出现消息重发，也因此暂不支持 QoS 为 2 的消息
    batch:
      max: 默认值：100，批量写消息序列号到数据库的最大条数
logger: 日志配置项
  path: 默认为空，即不打印到文件；如果指定文件则输出到文件
  level: 默认值：info，日志等级，支持 debug、info、warn 和 error
  format: 默认值：text，日志打印格式，支持 text 和 json
  age:
    max: 默认值：15，日志文件保留的最大天数
  size:
    max: 默认值：50，日志文件大小限制，单位 MB
  backup:
    max: 默认值：15，日志文件保留的最大数量
status: Hub 状态配置项
  logging:
    enable: 默认值：false，是否打印 Hub 的状态信息
    interval: 默认值：1m，状态信息打印时间间隔
storage: 数据库存储配置项
  dir: 默认值：`var/db/baetyl/data`，数据库存储目录
shutdown: Hub 退出配置项
  timeout: 默认值：10m，Hub 退出超时时间
```

6.3.3 baetyl-function-manager 模块

```
hub:
  clientid: Client 连接 Hub 的 Client ID。cleansession 为 false 则不允许为空
  address: [必须] Client 连接 Hub 的地址
  username: [必须] Client 连接 Hub 的用户名
  password: 如果采用账号密码，必须填 Client 连接 Hub 的密码，否者不用填写
  ca: 如果采用证书双向认证，必须填 Client 连接 Hub 的 CA 证书路径
  key: 如果采用证书双向认证，必须填 Client 连接 Hub 的客户端私钥路径
  cert: 如果采用证书双向认证，必须填 Client 连接 Hub 的客户端公钥路径
  timeout: 默认值：30s，Client 连接 Hub 的超时时间
  interval: 默认值：1m，Client 连接 Hub 的重连最大间隔时间，从 500 微秒翻倍增加到最大值
  keepalive: 默认值：10m，Client 连接 Hub 的保持连接时间
  cleansession: 默认值：false，Client 连接 Hub 的是否保持 Session
  validatesubs: 默认值：false，Client 是否检查 Hub 订阅结果，如果是发现订阅失败报错退出
  buffersize: 默认值：10，Client 发送消息给 Hub 的内存队列大小，异常退出会导致消息丢失，恢复
后 QoS 为 1 的消息依赖 Hub 重发
rules: 路由规则配置项
  - clientid: Client 连接 Hub 的 Client ID
    subscribe:
      topic: [必须] Client 向 Hub 订阅的消息主题
      qos: 默认值：0，Client 向 Hub 订阅的消息 QoS
    function:
      name: [必须] 处理消息的函数名
    publish:
      topic: [必须] 计算结果发布到 Hub 的主题
      qos: 默认值：0，计算结果发布 Hub 的 QoS
    retry:
      max: 默认值：3，最大重试次数
functions: 函数列表
  - name: [必须] 函数名称，列表内唯一
    service: [必须] 提供函数实例的服务名称
    instance: 实例配置项
      min: 默认值：0，最小值：0，最大值：100，最少实例数
      max: 默认值：1，最小值：1，最大值：100，最大实例数
      idletime: 默认值：10m，实例最大空闲时间
      evicttime: 默认值：1m，实例检查周期，如果发现实例空闲超过就销毁
    message:
      length:
        max: 默认值：4m，  函数实例允许接收和发送的最大消息长度
    backoff:
```

(下页继续)

(续上页)

```
      max: 默认值: 1m, Client 连接函数实例最大重连间隔
    timeout: 默认值: 30s, Client 连接函数实例超时时间
logger: 日志配置项
  path: 默认为空，即不打印到文件；如果指定文件则输出到文件
  level: 默认值: info, 日志等级，支持 debug、info、warn 和 error
  format: 默认值: text, 日志打印格式，支持 text 和 json
  age:
    max: 默认值: 15, 日志文件保留的最大天数
  size:
    max: 默认值: 50, 日志文件大小限制，单位 MB
  backup:
    max: 默认值: 15, 日志文件保留的最大数量
```

6.4 baetyl-function-python 配置

```
# 两个模块的配置方式相同，可参考下面一份配置
server: 作为 GRPC Server 独立启动时配置；托管给 baetyl-function-manager 无需配置
  address: GRPC Server 监听的地址, <host>:<port>
  workers:
    max: 默认 CPU 核数乘以 5, 线程池最大容量
  concurrent:
    max: 默认不限制，最大并发连接数
  message:
    length:
      max: 默认值: 4m,  函数实例允许接收和发送的最大消息长度
  ca: Server 的 CA 证书路径
  key: Server 的服务端私钥路径
  cert: Server 的服务端公钥路径
functions: 函数列表
  - name: [必须] 函数名称，列表内唯一
    handler: [必须] 函数包和处理函数名，比如: 'sayhi.handler'
    codedir: [必须] Python 代码所在路径
logger: 日志配置项
  path: 默认为空，即不打印到文件；如果指定文件则输出到文件
  level: 默认值: info, 日志等级，支持 debug、info、warn 和 error
  age:
    max: 默认值: 15, 日志文件保留的最大天数
  backup:
```

(下页继续)

(续上页)

```
      max: 默认值: 15, 日志文件保留的最大数量
```

6.4.1 baetyl-function-node 模块

```
server: 作为 GRPC Server 独立启动时配置; 托管给 baetyl-function-manager 无需配置
  address: GRPC Server 监听的地址, <host>:<port>
  message:
    length:
      max: 默认值: 4m,  函数实例允许接收和发送的最大消息长度
  ca: Server 的 CA 证书路径
  key: Server 的服务端私钥路径
  cert: Server 的服务端公钥路径
functions: 函数列表
  - name: [必须] 函数名称, 列表内唯一
    handler: [必须] 函数包和处理函数名, 比如: 'sayjs.handler'
    codedir: [必须] Node 代码所在路径
logger: 日志配置项
  path: 默认为空, 即不打印到文件; 如果指定文件则输出到文件
  level: 默认值: info, 日志等级, 支持 debug、info、warn 和 error
  backupCount:
        max: 默认值: 15, 日志文件保留的最大数量
```

6.4.2 baetyl-video-infer 模块

```
hub:
  clientid: Client 连接 Hub 的 Client ID。cleansession 为 false 则不允许为空
  address: [必须] Client 连接 Hub 的地址
  username: [必须] Client 连接 Hub 的用户名
  password: 如果采用账号密码, 必须填 Client 连接 Hub 的密码, 否者不用填写
  ca: 如果采用证书双向认证, 必须填 Client 连接 Hub 的 CA 证书路径
  key: 如果采用证书双向认证, 必须填 Client 连接 Hub 的客户端私钥路径
  cert: 如果采用证书双向认证, 必须填 Client 连接 Hub 的客户端公钥路径
  timeout: 默认值: 30s, Client 连接 Hub 的超时时间
  interval: 默认值: 1m, Client 连接 Hub 的重连最大间隔时间, 从 500 微秒翻倍增加到最大值
  keepalive: 默认值: 10m, Client 连接 Hub 的保持连接时间
  cleansession: 默认值: false, Client 连接 Hub 的是否保持 Session
  validatesubs: 默认值: false, Client 是否检查 Hub 订阅结果, 如果是发现订阅失败报错退出
  buffersize: 默认值: 10, Client 发送消息给 Hub 的内存队列大小, 异常退出会导致消息丢失, 恢复
后 QoS 为 1 的消息依赖 Hub 重发
```

(下页继续)

(续上页)

```
video:
  uri: [必须] 摄像头地址或视频文件路径，摄像头可支持 IP 网络摄像头和 USB 摄像头
    # 对于 IP 网络摄像头，通用配置方式为：`rtsp://<username>:<password>@<ip>:<port>/
↪Streaming/channels/<stream_number>/`
      # `<username>` 和 `<password>` 代表摄像头登录、认证口令
      # `<ip>` 代表 IP 网络摄像头的 IP 地址
      # `<port>` 代表 RTSP 协议的端口，默认为 554
      # `<stream_number>` 代表码流通道编号。如果为 1，则表示抓取的是主码流；如果为 2，则表示
抓取的是次码流
    # 对于 USB 摄像头，其配置内容为挂载的 USB 编号，如 "0" 代表设备 `/dev/video0`，另外还需
要将设备 `/dev/video0` 映射进容器
    # 对于视频文件，其配置内容为视频文件的路径，需要将视频文件以（自定义）存储卷方式挂载到␣
↪video infer 服务
  limit:
    fps: [必须] 表示每秒能够处理的最大帧量。如果摄像头帧率为 N，每秒最多能够处理的帧量为 M，
则 Ceil(N/M) - 1 帧将会被跳过
infer:
  model: [必须] 模型文件路径。需要了解更多内容可参考：https://docs.opencv.org/4.1.1/d6/d0f/
↪group__dnn.html#ga3b34fe7a29494a6a4295c169a7d32422.
  config: [必须] 模型文件的配置文件路径。需要了解更多内容可参考：https://docs.opencv.org/4.
↪1.1/d6/d0f/group__dnn.html#ga3b34fe7a29494a6a4295c169a7d32422.
  backend: [可选] 模型推断后处理加速配置项，可支持的配置项为：halide，openvino，opencv，
vulkan，default。需要了解更多内容可参考：https://docs.opencv.org/4.1.1/d6/d0f/group__dnn.
↪html#ga186f7d9bfacac8b0ff2e26e2eab02625
  device: [可选] 表示用何种硬件对模型进行推断，可选择的配置项为 cpu（默认），fp32，fp16，vpu，
vulkan，fpga。需要了解更多内容可参考：https://docs.opencv.org/4.1.1/d6/d0f/group__dnn.html
↪#ga709af7692ba29788182cf573531b0ff5
process:
  before: 从图像创建 4 维 blob。(可选) 从中心调整大小和裁剪图像，减去平均值，按比例因子缩放值，
交换蓝色和红色通道。需要了解更多内容可参考：https://docs.opencv.org/4.1.1/d6/d0f/group__
↪dnn.html#ga29f34df9376379a603acd8df581ac8d7
    scale: multiplier for image values.
    swaprb: 标志位，用于转换图像的第 1、3 通道，即将常用的 RGB 顺序调整为 BGR
    width: 输出图像的宽度
    hight: 输出图像的高度
    mean: 从通道中减去平均值的标量。如果图像具有 BGR 排序并且 swaprb 为 true，则值的排列顺序
应为（均值 R，均值 G，均值 B）
      v1: 代表 RGB 中的蓝色分量，与下面的 v2，v3，v4 一起常被用于传递像素值
      v2: 代表 RGB 中的绿色分量
      v3: 代表 RGB 中的红色分量
```

(下页继续)

(续上页)

```
      v4: 代表 Alpha 透明色分量
    crop: 标志位，表明图像在进行 resize 操作后是否会被裁剪
  after:
    function:
      name: [必须] 处理模型推断结果的函数名称
functions:
  - name: [必须] 函数名称，列表内唯一
    address: function manager 服务地址，通用配置方式是 <host>:<port>，如 `function-
↪manager:50051`
    message:
      length:
        max: 默认值：4m，  函数实例允许接收和发送的最大消息长度
    backoff:
      max: 默认值：1m，Client 连接函数实例最大重连间隔
    timeout: 默认值：30s，Client 连接函数实例超时时间
logger: 日志配置项
  path: 默认为空，即不打印到文件；如果指定文件则输出到文件
  level: 默认值：info，日志等级，支持 debug、info、warn 和 error
  format: 默认值：text，日志打印格式，支持 text 和 json
  age:
    max: 默认值：15，日志文件保留的最大天数
  size:
    max: 默认值：50，日志文件大小限制，单位 MB
  backup:
    max: 默认值：15，日志文件保留的最大数量
```

6.4.3 baetyl-remote-mqtt 模块

```
hub:
  clientid: Client 连接 Hub 的 Client ID。cleansession 为 false 则不允许为空
  address: [必须] Client 连接 Hub 的地址
  username: [必须] Client 连接 Hub 的用户名
  password: 如果采用账号密码，必须填 Client 连接 Hub 的密码，否者不用填写
  ca: 如果采用证书双向认证，必须填 Client 连接 Hub 的 CA 证书路径
  key: 如果采用证书双向认证，必须填 Client 连接 Hub 的客户端私钥路径
  cert: 如果采用证书双向认证，必须填 Client 连接 Hub 的客户端公钥路径
  timeout: 默认值：30s，Client 连接 Hub 的超时时间
  interval: 默认值：1m，Client 连接 Hub 的重连最大间隔时间，从 500 微秒翻倍增加到最大值
  keepalive: 默认值：10m，Client 连接 Hub 的保持连接时间
```

(下页继续)

(续上页)

```
  cleansession: 默认值: false, Client 连接 Hub 的是否保持 Session
  validatesubs: 默认值: false, Client 是否检查 Hub 订阅结果, 如果是发现订阅失败报错退出
  buffersize: 默认值: 10, Client 发送消息给 Hub 的内存队列大小, 异常退出会导致消息丢失, 恢复
后 QoS 为 1 的消息依赖 Hub 重发
rules: 路由规则列表, 向 Hub 订阅消息发送给 Remote, 或反之
  - hub:
      clientid: Client 连接 Hub 的 Client ID
      subscriptions: Client 向 Hub 订阅的消息, 例如
        - topic: say
          qos: 1
        - topic: hi
          qos: 0
    remote:
      name: [必须] 指定 Remote 名称, 必须是 Remote 列表中的一个
      clientid: Client 连接 Remote 的 Client ID
      subscriptions: Client 向 Remote 订阅的消息, 例如
        - topic: remote/say
          qos: 0
        - topic: remote/hi
          qos: 0
remotes: Remote 列表
  - name: [必须] Remote 名称, 列表内必须唯一
    clientid: Client 连接 Remote 的 Client ID
    address: [必须] Client 连接 Remote 的地址
    username: Client 连接 Remote 的用户名
    password: 如果采用账号密码, 必须填 Client 连接 Remote 的密码, 否者不用填写
    ca: 如果采用证书双向认证, 必须填 Client 连接 Remote 的 CA 证书路径
    key: 如果采用证书双向认证, 必须填 Client 连接 Remote 的客户端私钥路径
    cert: 如果采用证书双向认证, 必须填 Client 连接 Remote 的客户端公钥路径
    timeout: 默认值: 30s, Client 连接 Remote 的超时时间
    interval: 默认值: 1m, Client 连接 Remote 的重连最大间隔时间, 从 500 微秒翻倍增加到最大值
    keepalive: 默认值: 10m, Client 连接 Remote 的保持连接时间
    cleansession: 默认值: false, Client 连接 Remote 的是否保持 Session
    validatesubs: 默认值: false, Client 是否检查 Remote 订阅结果, 如果是发现订阅失败报错退出
    buffersize: 默认值: 10, Client 发送消息给 Remote 的内存队列大小, 异常退出会导致消息丢失,
恢复后 QoS 为 1 的消息依赖 Remote 重发
logger: 日志配置项
  path: 默认为空, 即不打印到文件; 如果指定文件则输出到文件
  level: 默认值: info, 日志等级, 支持 debug、info、warn 和 error
  format: 默认值: text, 日志打印格式, 支持 text 和 json
```

(下页继续)

(续上页)

```
  age:
    max: 默认值：15，日志文件保留的最大天数
  size:
    max: 默认值：50，日志文件大小限制，单位 MB
  backup:
    max: 默认值：15，日志文件保留的最大数量
```

6.4.4 baetyl-timer 模块

```
hub:
  clientid: Client 连接 Hub 的 Client ID。cleansession 为 false 则不允许为空
  address: [必须] Client 连接 Hub 的地址
  username: [必须] Client 连接 Hub 的用户名
  password: 如果采用账号密码，必须填 Client 连接 Hub 的密码，否者不用填写
  ca: 如果采用证书双向认证，必须填 Client 连接 Hub 的 CA 证书路径
  key: 如果采用证书双向认证，必须填 Client 连接 Hub 的客户端私钥路径
  cert: 如果采用证书双向认证，必须填 Client 连接 Hub 的客户端公钥路径
  timeout: 默认值：30s，Client 连接 Hub 的超时时间
  interval: 默认值：1m，Client 连接 Hub 的重连最大间隔时间，从 500 微秒翻倍增加到最大值
  keepalive: 默认值：10m，Client 连接 Hub 的保持连接时间
  cleansession: 默认值：false，Client 连接 Hub 的是否保持 Session
  validatesubs: 默认值：false，Client 是否检查 Hub 订阅结果，如果是发现订阅失败报错退出
  buffersize: 默认值：10，Client 发送消息给 Hub 的内存队列大小，异常退出会导致消息丢失，恢复
后 QoS 为 1 的消息依赖 Hub 重发
timer: timer 相关属性
  interval: timer 模块定时间隔
publish:
  topic: 定时结果发布到 Hub 的主题
  payload: 负载数据，map 数据结构，例如：
    id: 1
logger: 日志配置项
  path: 默认为空，即不打印到文件；如果指定文件则输出到文件
  level: 默认值：info，日志等级，支持 debug、info、warn 和 error
  format: 默认值：text，日志打印格式，支持 text 和 json
  age:
    max: 默认值：15，日志文件保留的最大天数
  size:
    max: 默认值：50，日志文件大小限制，单位 MB
  backup:
```

(下页继续)

(续上页)

max: 默认值：15，日志文件保留的最大数量

CHAPTER 7

通过 Hub 服务将设备接入 Baetyl

声明:

- 本文测试所用设备系统为 Ubuntu 18.04
- 模拟 MQTT Client 行为的客户端为 MQTT.fx 和 MQTTBox，其中 MQTT.fx 用于 TCP 和 SSL 连接方式的测试，MQTTBox 用于 WS(Websocket）连接方式的测试。

Baetyl Hub 服务的完整的配置参考 Hub 服务配置。

提示：*Darwin 系统可以通过源码安装 Baetyl，可参考*源码安装 Baetyl。

7.1 操作流程

- 步骤一：安装 Baetyl，**并导入示例配置包**。参考*快速安装 Baetyl* 进行操作；
- 步骤二：依据测试需求修改导入的配置信息，执行 `sudo systemctl start baetyl` 以容器模式启动 Baetyl，然后执行 `sudo systemctl status baetyl` 来查看 Baetyl 是否正常运行。如果 Baetyl 已经启动，执行 `sudo systemctl start baetyl` 重启来加载新的配置。
- 步骤三：依据选定的连接测试方式，对 MQTT Client 作相应配置；
  - 若采用 TCP 连接，则仅需配置用户名、密码（参见配置文件 `principals` 配置项 `username`、`password`），并选定对应连接端口即可；
  - 若采用 SSL 证书认证，除选定所需的用户名外，还需选定 CA 证书以及由 CA 签发的客户端证书和私钥，依据对应的连接端口连接即可；
  - 若采用 WS 连接，与 TCP 连接配置一样，仅需配置用户名、密码、相应端口即可；

- 步骤四：若上述步骤一切正常，操作无误，即可通过 Baetyl 日志或 MQTT Client 查看连接状态。

7.2 连接测试

依据 步骤一导入示例配置包后，确认一下应用配置和 Hub 服务的配置。

7.2.1 Baetyl 应用配置信息

如果采用官方的安装方式，将 Baetyl 应用配置替换成如下配置：

```
# /usr/local/var/db/baetyl/application.yml
version: v0
services:
  - name: localhub
    image: hub.baidubce.com/baetyl/baetyl-hub
    replica: 1
    ports:
      - 1883:1883
      - 8883:8883
      - 8080:8080
    mounts:
      - name: localhub-conf
        path: etc/baetyl
        readonly: true
      - name: localhub-cert
        path: var/db/baetyl/cert
        readonly: true
      - name: localhub-data
        path: var/db/baetyl/data
      - name: localhub-log
        path: var/log/baetyl
volumes:
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf
  - name: localhub-data
    path: var/db/baetyl/localhub-data
  - name: localhub-cert
    path: var/db/baetyl/localhub-cert-only-for-test
  - name: localhub-log
    path: var/db/baetyl/localhub-log
```

Baetyl Hub 服务的配置替换成如下配置：

```
# /usr/local/var/db/baetyl/localhub-conf/service.yml
listen:
  - tcp://0.0.0.0:1883
  - ssl://0.0.0.0:8883
  - ws://0.0.0.0:8080/mqtt
certificate:
  ca: var/db/baetyl/cert/ca.pem
  cert: var/db/baetyl/cert/server.pem
  key: var/db/baetyl/cert/server.key
principals:
  - username: two-way-tls
    permissions:
      - action: 'pub'
        permit: ['tls/#']
      - action: 'sub'
        permit: ['tls/#']
  - username: test
    password: hahaha
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
        permit: ['#']
subscriptions:
  - source:
      topic: 't'
    target:
      topic: 't/topic'
logger:
  path: var/log/baetyl/service.log
  level: 'debug'
```

容器模式需要端口映射，允许外部通过端口来访问容器，对应的配置项为应用配置中的 `ports` 字段。

如上所述，Hub 服务启动时会同时开启 1883、8883 以及 8080 端口，分别用作 TCP、SSL、WS（Websocket）等几种方式进行连接，下文将以 MQTTBox 和 MQTT.fx 作为 MQTT Client，测试他们分别在上述这几种连接方式下与 Baetyl 的连接情况，具体如下。

7.2.2 Baetyl 启动

依据 步骤二，执行 `sudo systemctl start baetyl` 以容器模式启动 Baetyl，如果 Baetyl 已经启动，执行 `sudo systemctl restart baetyl` 来重启。然后执行 `sudo systemctl status baetyl` 来查看 `baetyl` 是否正常运行。正常启动的情况如下图所示。

```
baetyl@baetyl:~$ sudo systemctl status baetyl
● baetyl.service - Baetyl
   Loaded: loaded (/lib/systemd/system/baetyl.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2019-09-23 11:31:57 CST; 5h 55min ago
 Main PID: 997 (baetyl)
    Tasks: 10 (limit: 4623)
   CGroup: /system.slice/baetyl.service
           └─997 /usr/local/bin/baetyl start

9月 23 11:31:57 baetyl systemd[1]: Started Baetyl.
```

Baetyl 状态

提示：*Darwin 系统通过源码安装 Baetyl，可执行 `sudo baetyl start` 以容器模式启动 Baetyl。*

查看 Baetyl 主程序的日志，执行 `sudo tail -f /usr/local/var/log/baetyl/baetyl.log` 显示如下：

```
baetyl@baetyl:~$ tail -f /usr/local/var/log/baetyl/baetyl.log
time="2019-09-23T17:55:39+08:00" level=debug msg="instance (localhub) not found" baetyl=master engine=docker service=localhub
time="2019-09-23T17:55:42+08:00" level=debug msg="container (bc82f4ade99e:localhub) started" baetyl=master engine=docker
time="2019-09-23T17:55:42+08:00" level=info msg="instance started" baetyl=master cid=bc82f4ade99e engine=docker instance=localhub service=localhub
time="2019-09-23T17:55:42+08:00" level=info msg="service (localhub) started" baetyl=master
time="2019-09-23T17:55:42+08:00" level=info msg="app version (v0) committed" baetyl=master
time="2019-09-23T17:55:42+08:00" level=info msg="services started" baetyl=master
time="2019-09-23T17:56:42+08:00" level=info msg="[PUT] /v1/services/localhub/instances/localhub/report" api=server baetyl=master
time="2019-09-23T17:56:42+08:00" level=info msg="[PUT] /v1/services/localhub/instances/localhub/report" api=server baetyl=master
time="2019-09-23T17:57:42+08:00" level=info msg="[PUT] /v1/services/localhub/instances/localhub/report" api=server baetyl=master
time="2019-09-23T17:57:42+08:00" level=info msg="[PUT] /v1/services/localhub/instances/localhub/report" api=server baetyl=master
```

Baetyl 启动

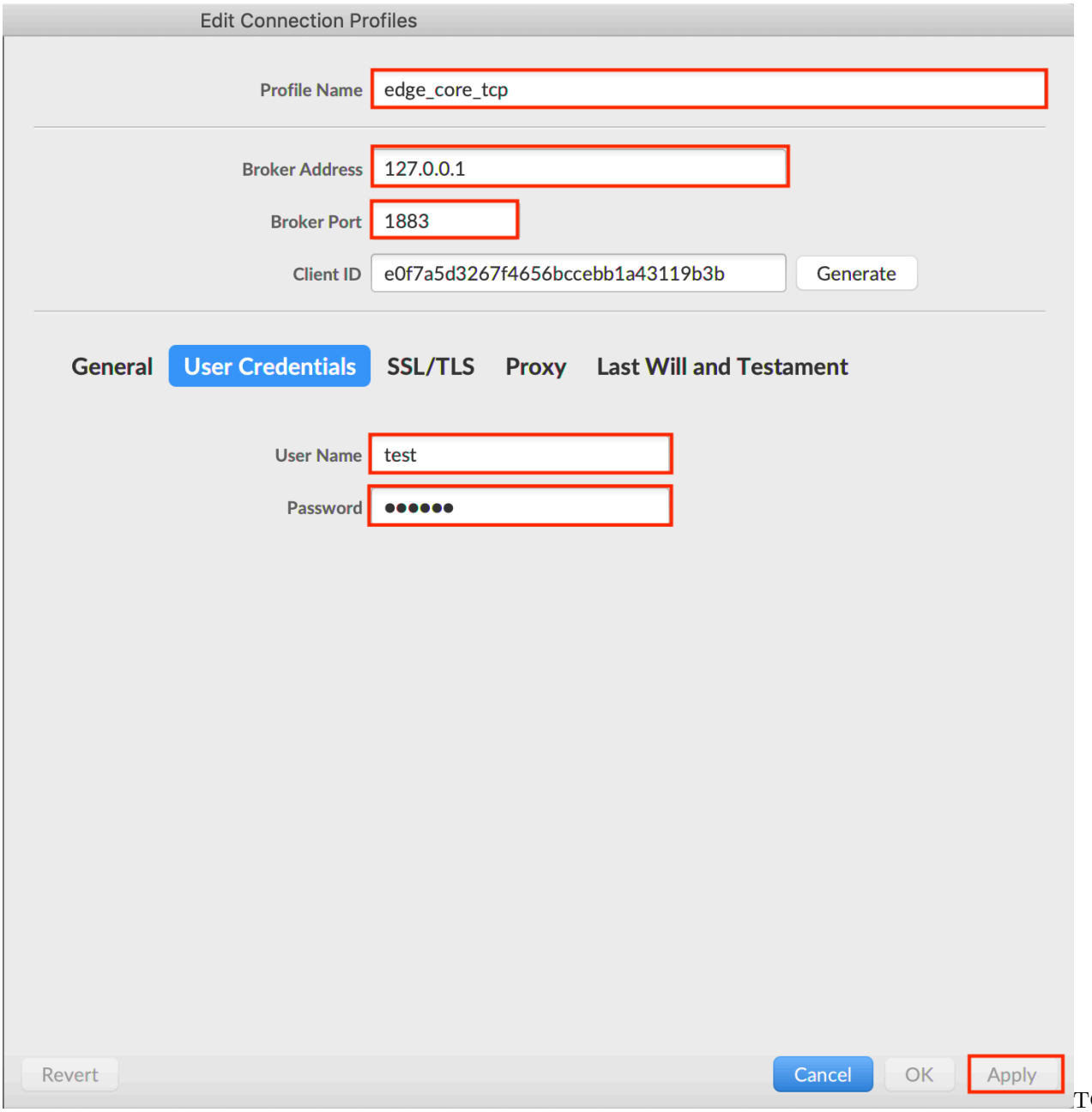
可以看到，Baetyl 正常启动后，Hub 服务镜像已被加载。另外，亦可以通过命令 `docker ps` 查看系统当前正在运行的容器。

```
baetyl@baetyl:~$ docker ps
CONTAINER ID        IMAGE                                   COMMAND             CREATED             STATUS              PORTS
                    NAMES
bc82f4ade99e        hub.baidubce.com/baetyl/baetyl-hub:latest   "baetyl-hub"        6 minutes ago       Up 6 minutes        0.0.0.0:1883->1883/tcp, 0.0.0.0:8080->8080
/tcp, 0.0.0.0:8883->8883/tcp   localhub
```

查看系统当前正在运行的容器

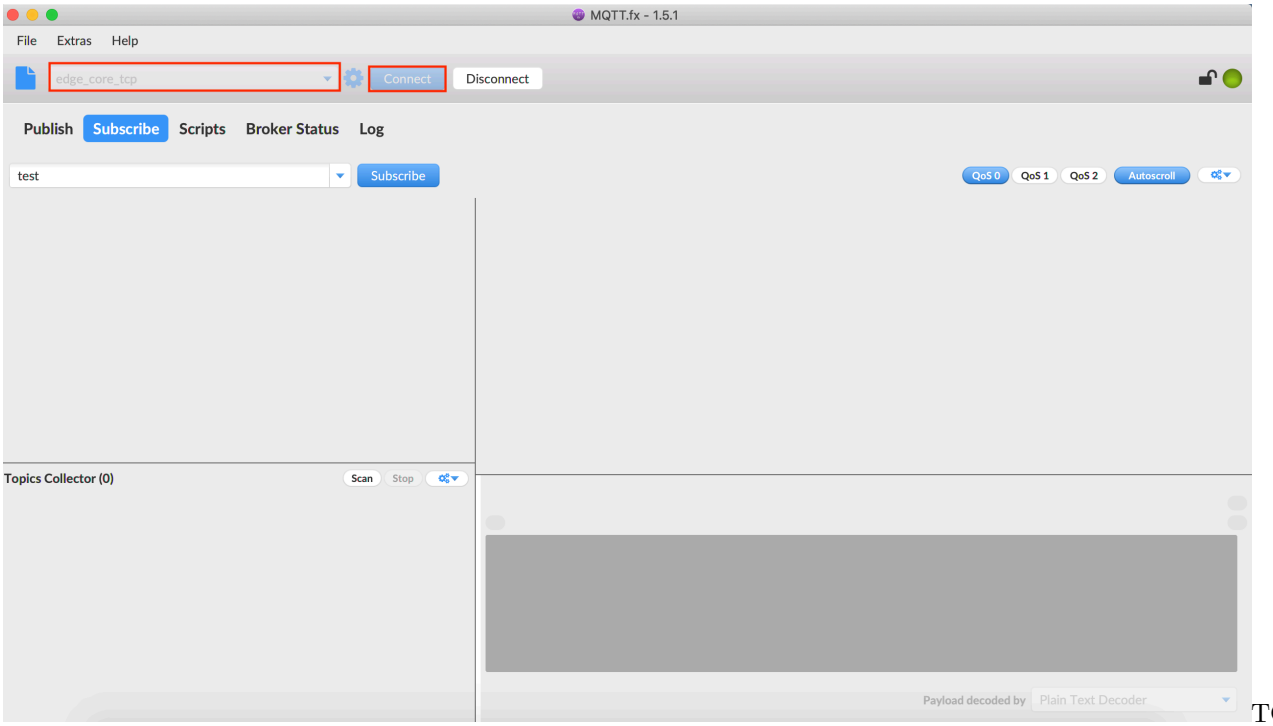
TCP 连接测试

启动 MQTT.fx 客户端，进入 Edit Connection Profiles 页面，填写 Profile Name，依据 Baetyl Hub 服务启动的地址及端口，填写 Broker Address 和 Broker Port ，再结合 `principals` 配置项中的连接信息配置 User Credentials 中的 User Name 和 Password ，然后点击 `Apply` 按钮，即可完成 TCP 连接模式下 MQTT.fx 的连接配置，具体如下图示。



TCP 连接测试配置

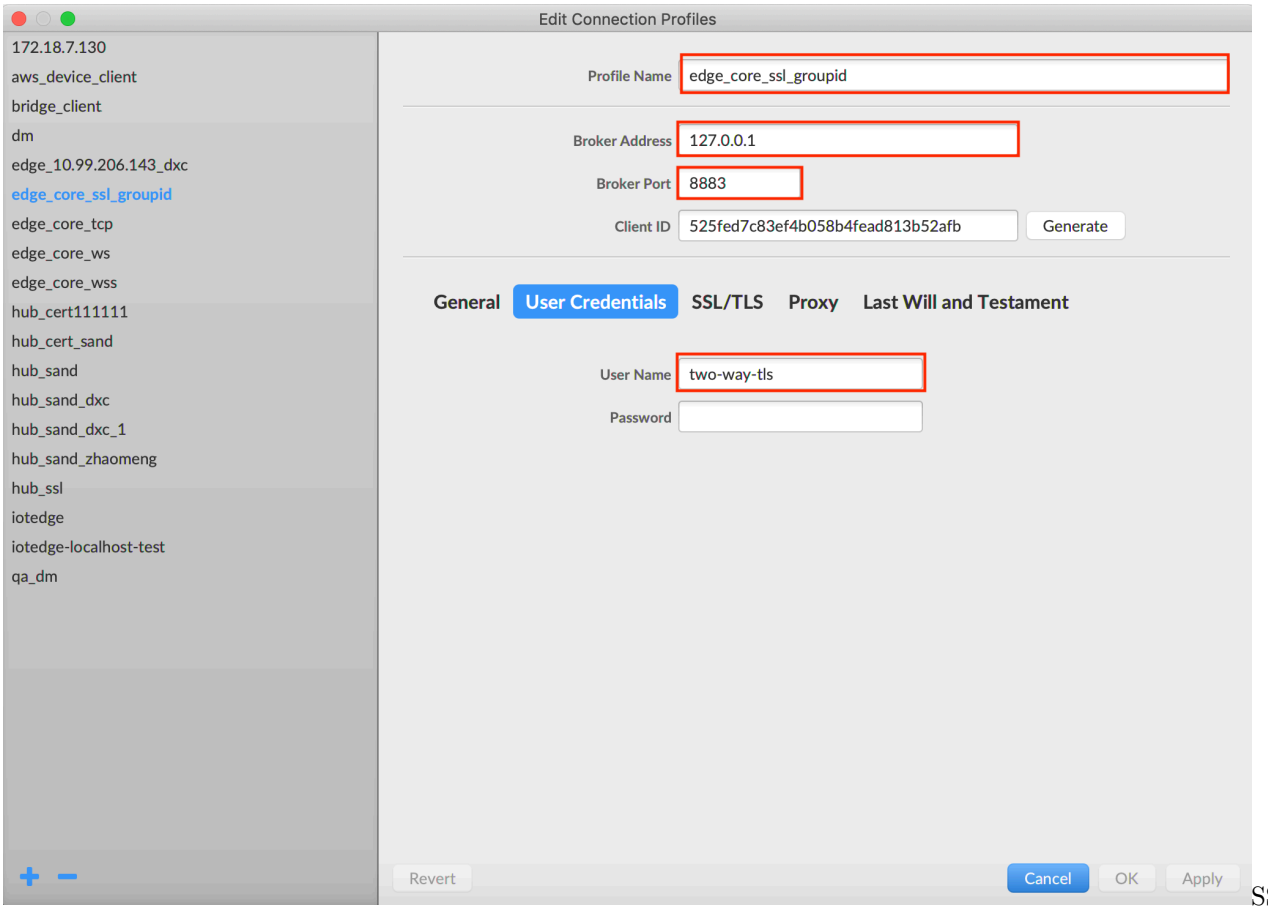
然后关掉配置页面，选择刚才的 Profile Name 进行 `Connect`，若连接配置信息与 Baetyl Hub 服务 `principals` 配置项中允许连接的权限信息匹配，即可看到连接成功的标志，具体如下图示。

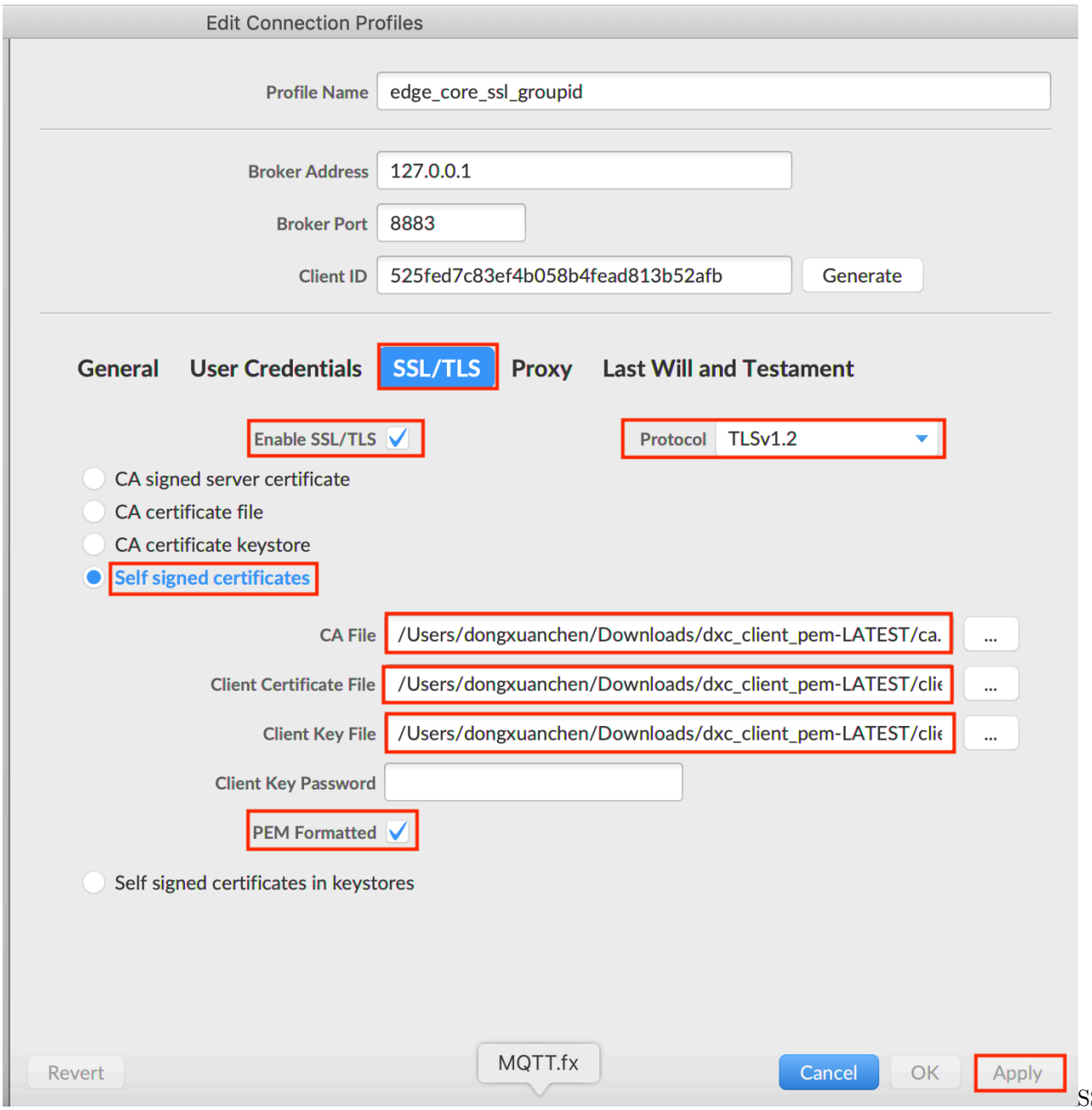


TCP 连接成功

SSL 连接测试

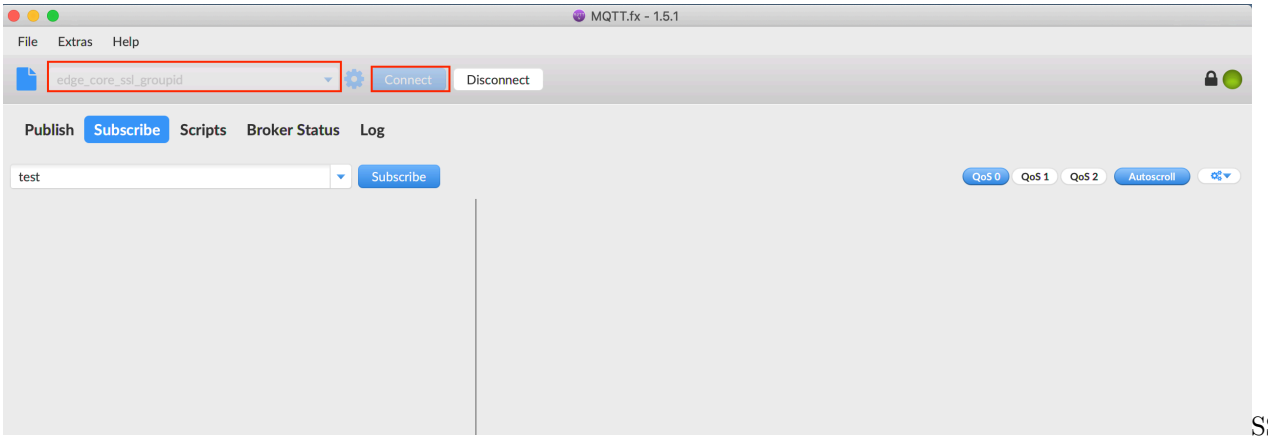
启动 MQTT.fx 客户端，进入 Edit Connection Profiles 页面，与 TCP 连接配置类似，填写相应的 Profile Name、Broker Address 和 Broker Port, 对于 SSL 连接方式，需填写 User Credentials 中的 User Name，同时配置 SSL 相关的设置，配置如图所示，然后点击 `Apply` 按钮，即可完成 SSL 连接模式下 MQTT.fx 的连接配置。


Edit Connection Profiles
172.18.7.130
aws_device_client
bridge_client
dm
edge_10.99.206.143_dxc
edge_core_ssl_groupid
edge_core_tcp
edge_core_ws
edge_core_wss
hub_cert111111
hub_cert_sand
hub_sand
hub_sand_dxc
hub_sand_dxc_1
hub_sand_zhaomeng
hub_ssl
iotedge
iotedge-localhost-test
qa_dm
Profile Name
edge_core_ssl_groupid
Broker Address
127.0.0.1
Broker Port
8883
Client ID
525fed7c83ef4b058b4fead813b52afb
Generate
General
User Credentials
SSL/TLS
Proxy
Last Will and Testament
User Name
two-way-tls
Password
Revert
Cancel
OK
Apply
SSL 连接测试配置



SSL 连接测试配置

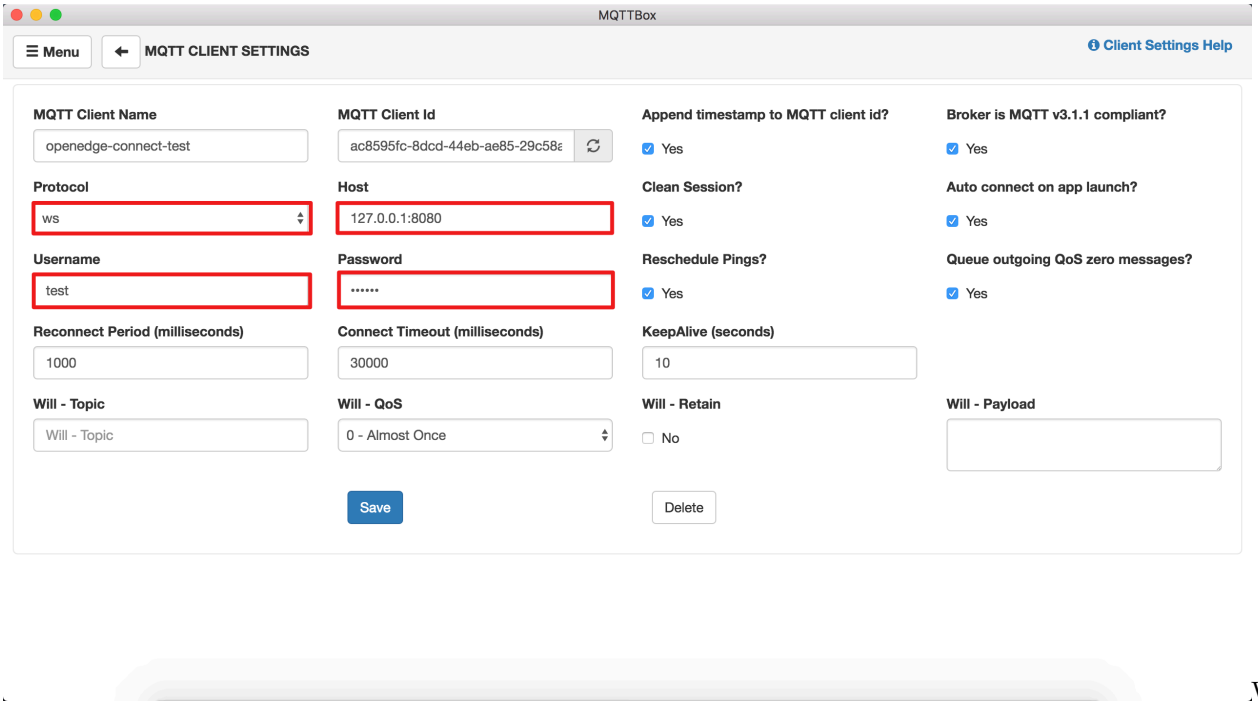
然后关掉配置页面，选择刚才的 Profile Name 进行 `Connect`，若连接配置信息与 Baetyl Hub 服务 `principals` 配置项中允许连接的权限信息匹配，即可看到连接成功的标志，具体如下图示。



SSL 连接成功

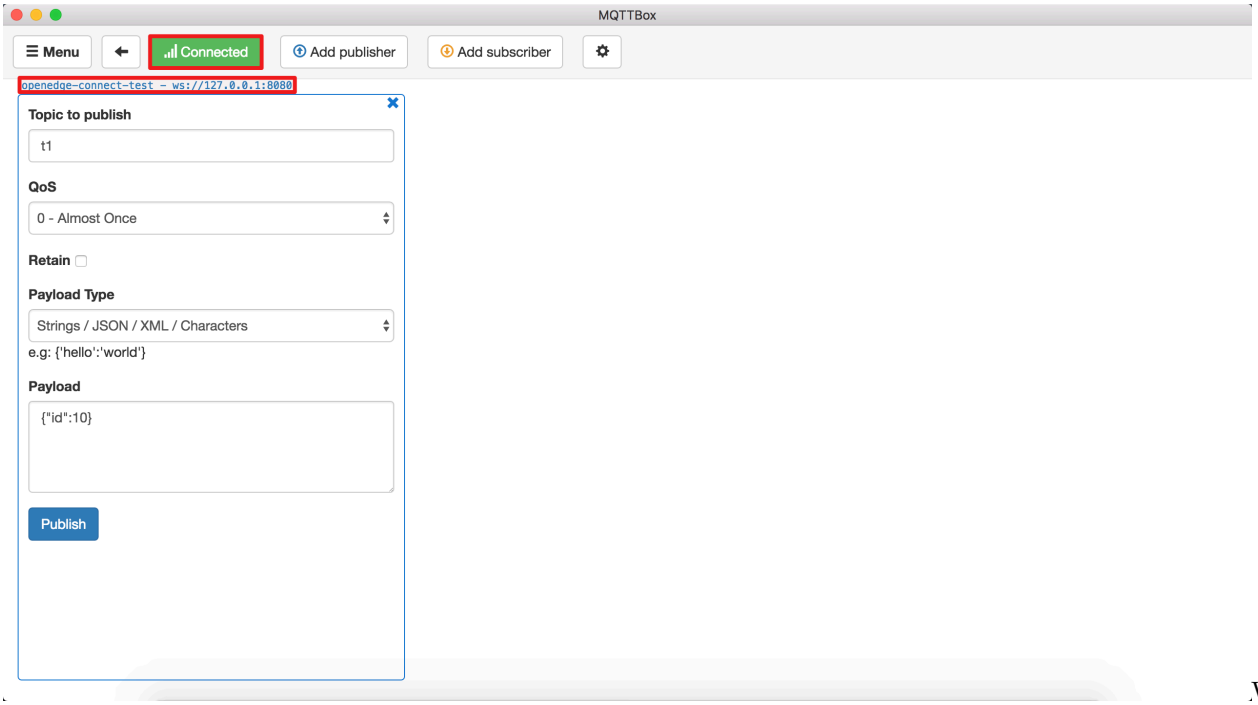
WS（Websocket）连接测试

启动 MQTTBox 客户端，进入 Client 创建页面，选择连接使用的协议为 `ws`，依据 Baetyl Hub 服务启动的地址及端口，再结合 principals 配置项中用户名、密码进行配置，然后点击 Save 按钮，即可完成 WS 连接模式下 MQTTBox 的连接配置，具体如下图示。



WS（Websocket）连接测试配置

只要上述操作正确、无误，即可在 MQTTBox 看到与 Baetyl Hub 成功建立连接的标志，具体如下图示。



.WS（Websocket）连接成功

综上，我们通过 MQTT.fx 和 MQTTBox 顺利完成了与 Baetyl Hub 服务的连接测试，除此之外，我们还可以通过 Paho-MQTT 自己编写测试脚本与 Baetyl Hub 连接，具体参见相关资源下载。

CHAPTER 8

利用 Hub 服务进行设备间消息转发

声明:

- 本文测试所用设备系统为 Ubuntu 18.04
- 模拟 MQTT client 行为的客户端为 MQTTBox

提示: *Darwin 系统可以通过源码安装 Baetyl，可参考*源码安装 Baetyl。

与 连接测试 不同的是，若需要通过 Hub 服务完成消息在设备间的转发及简单路由，除需要配置连接项信息外，还需要给可允许连接的 client 配置相应主题的权限，及简单的消息路由策略，完整的配置参考 Hub 服务配置。

本文以 TCP 连接方式为例，测试 Hub 服务的消息路由、转发功能。

8.1 操作流程

- 步骤一：安装 Baetyl，**并导入示例配置包**。参考*快速安装 Baetyl* 进行操作；
- 步骤二：依据测试需求修改导入的配置信息，执行 `sudo systemctl start baetyl` 以容器模式启动 Baetyl，然后执行 `sudo systemctl status baetyl` 来查看 Baetyl 是否正常运行。如果 Baetyl 已经启动，执行 `sudo systemctl start baetyl` 重启来加载新的配置。
- 步骤三：通过 MQTTBox 以 TCP 方式与 Hub 服务建立连接；
  - 若成功与 Hub 服务建立连接，则依据配置的主题权限信息向有权限的主题发布消息，同时向拥有订阅权限的主题订阅消息；

  - 若与 Hub 服务建立连接失败，则重复 步骤三操作，直至 MQTTBox 与 Hub 服务成功建立连接为止。

- 步骤四：通过 MQTTBox 查看消息的收发状态。

8.2 消息路由测试

Baetyl 应用配置替换成如下配置：

```
# /usr/local/var/db/baetyl/application.yml
version: V2
services:
  - name: hub
    image: 'hub.baidubce.com/baetyl/baetyl-hub'
    replica: 1
    ports:
      - '1883:1883'
    mounts:
      - name: localhub-conf
        path: etc/baetyl
        readonly: true
      - name: localhub_data
        path: var/db/baetyl/data
      - name: log-V1
        path: var/log/baetyl
volumes:
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf/V1
  - name: log-V1
    path: var/db/baetyl/log
  - name: localhub_data
    path: var/db/baetyl/localhub_data
```

Baetyl Hub 服务配置替换成如下配置：

```
# /usr/local/var/db/baetyl/localhub-conf/service.yml
listen:
  - tcp://0.0.0.0:1883
principals:
  - username: 'test'
    password: 'hahaha'
```

(下页继续)

(续上页)

```
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
        permit: ['#']
subscriptions:
  - source:
      topic: 't'
    target:
      topic: 't/topic'
logger:
  path: var/log/baetyl/service.log
  level: 'debug'
```

如上配置，消息路由依赖 subscriptions 配置项，这里表示发布到主题 t 的消息将会转发给所有订阅主题 t/topic 的设备（用户）。

注意：上述配置项信息中，permissions 项下属 action 的 permit 权限主题列表支持 + 和 # 通配符配置，其具体释义如下详述。

 # 匹配策略

对于 MQTT 协议，数字标志（# U+0023）是用于匹配主题中任意层级的通配符。多层通配符表示它的父级和任意数量的子层级。多层通配符必须位于它自己的层级或者跟在主题层级分隔符（/ U+002F）后面。不管哪种情况，它都必须是主题过滤器的最后一个字符。

例如，如果客户端订阅主题 sport/tennis/player1/#，它会收到使用下列主题名发布的消息：

- sport/tennis/player1
- sport/tennis/player1/ranking
- sport/tennis/player1/score/wimbledon

此外，主题 sport/# 也匹配单独的 sport，因为 # 包括它的父级。

对于 Baetyl 来说，如果在 permit 配置项中配置了主题 #（不论是发布行为，还是订阅行为），都不需要再额外配置其他的主题。这时，配置项中的账户（依据 username/password）拥有向所有合法的 MQTT 协议主题发布或订阅的权限。

+ 匹配策略

对于 MQTT 协议，加号 (+ U+002B) 是只能用于单个主题层级匹配的通配符。在主题过滤器的任意层级都可以使用单层通配符，包括第一个和最后一个层级。然而它必须占据过滤器的整个层级。可以在主题过滤器中的多个层级中使用它，也可以和多层通配符一起使用。

例如，主题 sport/tennis/+ 匹配 sport/tennis/player1 和 sport/tennis/player2，但是不匹配 sport/tennis/player1/ranking。同时，由于单层通配符只能匹配一个层级，sport/+ 不匹配 sport 但是却匹配

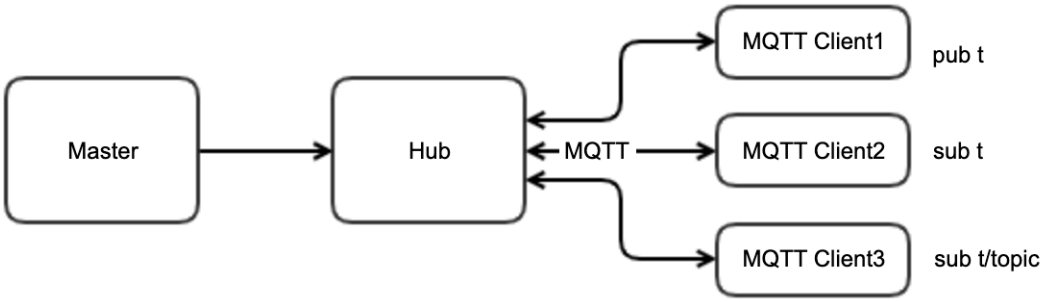
sport/。

对于 Baetyl 来说，如果在 permit 配置项中配置了主题 +（不论是发布行为，还是订阅行为），都不需要再额外配置其他的单层主题。这时，配置项中的账户（依据 username/password）拥有向所有合法的 MQTT 协议单层主题发布或订阅的权限。

提示：在 MQTT 协议中，通配符（不论是多层通配符 #，还是单层通配符 +）**只能**出现在订阅的主题过滤器中，而**不准**出现在发布的主题中。但是，为了增强主题权限配置的灵活性，Baetyl 在设计中认定，通配符不论出现在订阅行为的主题配置项中，还是出现在发布行为的主题配置项中，都是**合法**的。这里，在进行具体的发布/订阅行为时，发布或订阅的主题**只要**符合 MQTT 协议的要求即可。特别地，对于需要在 principals 配置项中配置大量发布和订阅主题的开发者来说，推荐采用通配符（# 和 + ）策略。

8.2.1 设备间消息转发路由测试

设备间消息转发、路由流程具体如下图示：



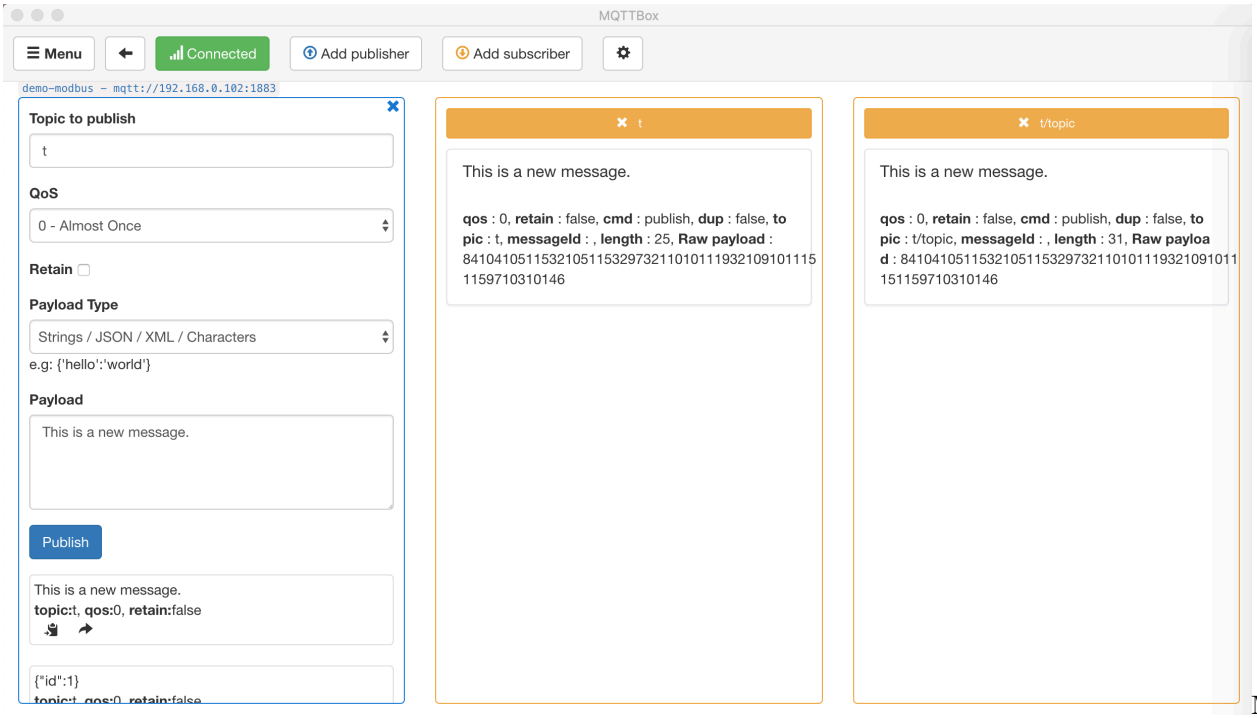
设备间消息转发路由流程图

具体地，如上图所示，**client1**、**client2** 及 **client3** 分别与 Hub 服务建立连接关系，**client1** 具备向主题 t 发布消息的权限，**client2** 及 **client3** 分别拥有向主题 t 及 t/topic 订阅消息的权限。

一旦上述三个 client 与 Hub 服务的连接关系建立后，依照上文 subscriptions 配置项信息，**client2** 及 **client3** 将会分别得到从 **client1** 向 Baetyl Hub 服务发布到主题 t 的消息。

特别地，**client1**、**client2** 及 **client3** 可以合并为一个 client，则新的 client 即拥有向主题 t 的发布消息权限，又拥有向主题 t 及 t/topic 订阅消息的权限。这里，采用 MQTTBox 作为该新 client，点击 Add subscriber 按钮添加主题 t 及 t/topic 进行订阅。

然后点击 Publish 按钮向 Hub 服务发送主题为 t 负载为 This is a new message. 的消息，即会发现在订阅的主题 t 及 t/topic 中均收到了该消息，详细如下图示。



MQTTBox

成功收到消息

综上，即通过 MQTTBox 完成了基于 Hub 服务的设备间消息转发、路由测试。

CHAPTER 9

利用 Function 函数计算服务进行消息处理

声明:

- 本文测试所用设备系统为 Ubuntu 18.04
- 本文测试使用的函数运行时是 Python3
- 模拟 MQTT client 行为的客户端为 MQTTBox

提示: *Darwin 系统可以通过源码安装 Baetyl，可参考*源码安装 Baetyl。

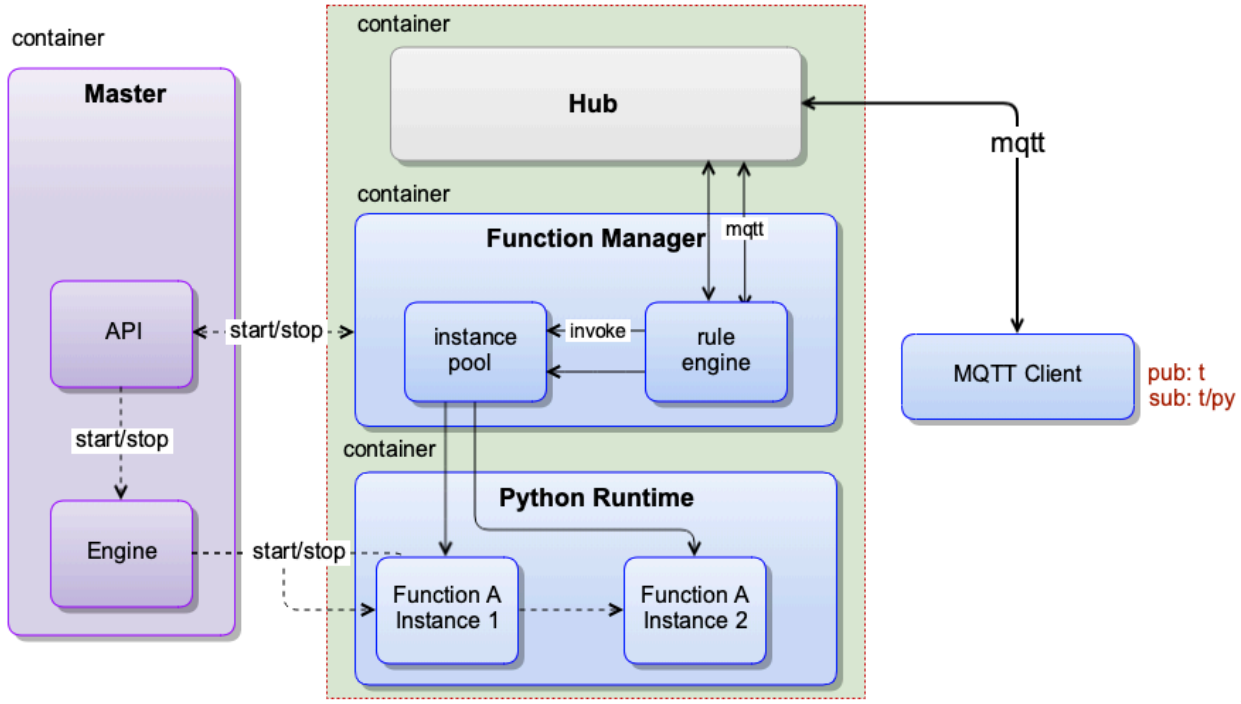
与基于 Hub 服务实现设备间消息转发不同的是，本文主要介绍利用本地函数计算服务进行消息处理。其中 Hub 服务用于建立 Baetyl 与 MQTT 客户端之间的连接，Python 运行时服务用于处理 MQTT 消息，而本地函数计算服务则通过 MQTT 消息上下文衔接本地 Hub 服务与 Python 运行时服务。

本文将以 TCP 连接方式为例，展示本地函数计算服务的消息处理、计算功能。

9.1 操作流程

- 步骤一：安装 Baetyl，**并导入示例配置包**。参考*快速安装 Baetyl* 进行操作；
- 步骤二：依据测试需求修改导入的配置信息，执行 `sudo systemctl start baetyl` 以容器模式启动 Baetyl，然后执行 `sudo systemctl status baetyl` 来查看 Baetyl 是否正常运行。如果 Baetyl 已经启动，执行 `sudo systemctl start baetyl` 重启来加载新的配置。
- 步骤三：通过 MQTTBox 以 TCP 方式与 Baetyl Hub 服务 建立连接；
  - 若成功与 Hub 服务建立连接，则依据配置的主题权限信息向有权限的主题发布消息，同时向拥有订阅权限的主题订阅消息，并观察 Baetyl 日志信息；

    * 若 Baetyl 日志显示已经启动 Python 运行时服务，则表明发布的消息受到了预期的函数处理；
    * 若 Baetyl 日志显示未成功启动 Python 运行时服务，则重复上述步骤，直至看到 Baetyl 主程序成功启动了 Python 运行时服务。
  - 若与 Baetyl Hub 建立连接失败，则重复步骤三操作，直至 MQTTBox 与 Baetyl Hub 服务成功建立连接为止。
- 步骤四：通过 MQTTBox 查看对应主题消息的收发状态。



基于本地函数计算服务实现设备消息处理流程

9.2 消息处理测试

依据 步骤一导入示例配置包后，确认一下应用配置、Hub 服务配置以及函数计算服务配置。

将 Baetyl 应用配置改成如下配置：

```
# /usr/local/var/db/baetyl/application.yml
version: v0
services:
  - name: localhub
    image: hub.baidubce.com/baetyl/baetyl-hub
    replica: 1
    ports:
      - 1883:1883
```

(下页继续)

(续上页)

```
  mounts:
    - name: localhub-conf
      path: etc/baetyl
      readonly: true
    - name: localhub-data
      path: var/db/baetyl/data
    - name: localhub-log
      path: var/log/baetyl
- name: function-manager
  image: hub.baidubce.com/baetyl/baetyl-function-manager
  replica: 1
  mounts:
    - name: function-manager-conf
      path: etc/baetyl
      readonly: true
    - name: function-manager-log
      path: var/log/baetyl
- name: function-python27-sayhi
  image: hub.baidubce.com/baetyl/baetyl-function-python27
  replica: 0
  mounts:
    - name: function-sayhi-conf
      path: etc/baetyl
      readonly: true
    - name: function-sayhi-code
      path: var/db/baetyl/function-sayhi
      readonly: true
- name: function-python36-sayhi
  image: hub.baidubce.com/baetyl/baetyl-function-python36
  replica: 0
  mounts:
    - name: function-sayhi-conf
      path: etc/baetyl
      readonly: true
    - name: function-sayhi-code
      path: var/db/baetyl/function-sayhi
      readonly: true
- name: function-node85-sayhi
  image: hub.baidubce.com/baetyl/baetyl-function-node85
  replica: 0
```

(下页继续)

(续上页)

```
    mounts:
      - name: function-sayjs-conf
        path: etc/baetyl
        readonly: true
      - name: function-sayjs-code
        path: var/db/baetyl/function-sayhi
        readonly: true
  - name: function-sql-filter
    image: hub.baidubce.com/baetyl/baetyl-function-sql
    replica: 0
    mounts:
      - name: function-filter-conf
        path: etc/baetyl
        readonly: true
volumes:
  # hub
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf
  - name: localhub-data
    path: var/db/baetyl/localhub-data
  - name: localhub-cert
    path: var/db/baetyl/localhub-cert-only-for-test
  - name: localhub-log
    path: var/db/baetyl/localhub-log
  # function
  - name: function-manager-conf
    path: var/db/baetyl/function-manager-conf
  - name: function-manager-log
    path: var/db/baetyl/function-manager-log
  - name: function-sayhi-conf
    path: var/db/baetyl/function-sayhi-conf
  - name: function-sayhi-code
    path: var/db/baetyl/function-sayhi-code
  - name: function-sayjs-conf
    path: var/db/baetyl/function-sayjs-conf
  - name: function-sayjs-code
    path: var/db/baetyl/function-sayjs-code
  - name: function-filter-conf
    path: var/db/baetyl/function-filter-conf
```

Baetyl Hub 服务配置改成如下配置：

```yaml
# /usr/local/var/db/baetyl/localhub-conf/service.yml
listen:
  - tcp://0.0.0.0:1883
principals:
  - username: test
    password: hahaha
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
        permit: ['#']
subscriptions:
  - source:
      topic: 't'
    target:
      topic: 't/topic'
logger:
  path: var/log/baetyl/service.log
  level: "debug"
```

Baetyl 本地函数计算服务相关配置无需修改，具体配置如下：

```yaml
# /usr/local/var/db/baetyl/function-manager-conf/service.yml
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
rules:
  - clientid: func-python27-sayhi-1
    subscribe:
      topic: t
    function:
      name: python27-sayhi
    publish:
      topic: t/py2hi
  - clientid: func-sql-filter-1
    subscribe:
      topic: t
      qos: 1
    function:
      name: sql-filter
```

(下页继续)

(续上页)

```
    publish:
      topic: t/sqlfilter
      qos: 1
  - clientid: func-python36-sayhi-1
    subscribe:
      topic: t
    function:
      name: python36-sayhi
    publish:
      topic: t/py3hi
  - clientid: func-node85-sayhi-1
    subscribe:
      topic: t
    function:
      name: node85-sayhi
    publish:
      topic: t/node8hi
functions:
  - name: python27-sayhi
    service: function-python27-sayhi
    instance:
      min: 0
      max: 10
  - name: sql-filter
    service: function-sql-filter
  - name: python36-sayhi
    service: function-python36-sayhi
  - name: node85-sayhi
    service: function-node85-sayhi
logger:
  path: var/log/baetyl/service.log
  level: "debug"

# /usr/local/var/db/baetyl/function-filter-conf/service.yml
functions:
  - name: sql-filter
    handler: 'select qos() as qos, topic() as topic, * where id < 10'

# /usr/local/var/db/baetyl/function-sayhi-conf/service.yml
functions:
```

(下页继续)

(续上页)

```
  - name: 'python27-sayhi'
    handler: 'index.handler'
    codedir: 'var/db/baetyl/function-sayhi'
  - name: 'python36-sayhi'
    handler: 'index.handler'
    codedir: 'var/db/baetyl/function-sayhi'

# /usr/local/var/db/baetyl/function-sayjs-conf/service.yml
functions:
  - name: 'node85-sayhi'
    handler: 'index.handler'
    codedir: 'var/db/baetyl/function-sayhi'
```

Python 函数代码无需修。/usr/local/var/db/baetyl/function-sayhi-code/index.py 实现如下：

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
"""
function to say hi in python
"""

def handler(event, context):
    """
    function handler
    """
    res = {}
    if isinstance(event, dict):
        if "err" in event:
            raise TypeError(event['err'])
        res = event
    elif isinstance(event, bytes):
        res['bytes'] = event.decode("utf-8")

    if 'messageQOS' in context:
        res['messageQOS'] = context['messageQOS']
    if 'messageTopic' in context:
        res['messageTopic'] = context['messageTopic']
    if 'messageTimestamp' in context:
        res['messageTimestamp'] = context['messageTimestamp']
    if 'functionName' in context:
```

(下页继续)

(续上页)

```
        res['functionName'] = context['functionName']
    if 'functionInvokeID' in context:
        res['functionInvokeID'] = context['functionInvokeID']

    res['Say'] = 'Hello Baetyl'
    return res
```

Node 函数代码无需修。/usr/local/var/db/baetyl/function-sayjs-code/index.js 实现如下：

```
#!/usr/bin/env node

const hasAttr = (obj, attr) => {
    if (obj instanceof Object && !(obj instanceof Array)) {
        if (obj[attr] != undefined) {
            return true;
        }
    }
    return false;
};

const passParameters = (event, context) => {
    if (hasAttr(context, 'messageQOS')) {
        event['messageQOS'] = context['messageQOS'];
    }
    if (hasAttr(context, 'messageTopic')) {
        event['messageTopic'] = context['messageTopic'];
    }
    if (hasAttr(context, 'messageTimestamp')) {
        event['messageTimestamp'] = context['messageTimestamp'];
    }
    if (hasAttr(context, 'functionName')) {
        event['functionName'] = context['functionName'];
    }
    if (hasAttr(context, 'functionInvokeID')) {
        event['functionInvokeID'] = context['functionInvokeID'];
    }
};

exports.handler = (event, context, callback) => {
    // support Buffer & json object
```

(下页继续)

(续上页)

```
    if (Buffer.isBuffer(event)) {
        const message = event.toString();
        event = {}
        event["bytes"] = message;
    }
    else if("err" in event) {
        return callback(new TypeError(event['err']))
    }

    passParameters(event, context);
    event['Say'] = 'Hello Baetyl'
    callback(null, event);
};
```

如上配置，假若 MQTTBox 基于上述配置信息已与 Hub 服务建立连接，向 Hub 发送主题为 `t` 的消息，函数计算服务会降消息分别路由给 `python27-sayhi`、`python36-sayhi`、`node85-sayhi` 和 `sql-filter` 函数处理，并分别输出主题为 `t/py2hi`、`t/py3hi`、`t/node8hi` 和 `t/sqlfilter` 的消息。这时订阅主题 `#` 的 MQTT client 将会接收到这这些消息，以及原消息 `t` 和 Hub 服务直接转主题的消息 `t/topic`。

提示：凡是在 *rules* 消息路由配置项中出现、用到的函数，必须在 *functions* 配置项中进行函数实例的配置，否则无法正常启动函数运行时实例。

9.2.1 Baetyl 启动

依据 步骤二，执行 `sudo systemctl start baetyl` 以容器模式启动 Baetyl，如果 Baetyl 已经启动，执行 `sudo systemctl restart baetyl` 来重启。

查看 Baetyl 主程序的日志，执行 `sudo tail -f -n 40 /usr/local/var/log/baetyl/baetyl.log` 显示如下：

```
time="2019-09-23T21:49:08+08:00" level=info msg="mode: docker; grace: 3000000000; pwd: /usr/local; api: unix:///var/run/baetyl.sock" baetyl=master
time="2019-09-23T21:49:08+08:00" level=info msg="engine started" baetyl=master
time="2019-09-23T21:49:08+08:00" level=error msg="no such file or directory" api=server baetyl=master
time="2019-09-23T21:49:08+08:00" level=info msg="server started" baetyl=master
time="2019-09-23T21:49:08+08:00" level=debug msg="network (4c71d71eeab3:baetyl) exists" baetyl=master engine=docker
time="2019-09-23T21:49:09+08:00" level=debug msg="image (hub.baidubce.com/baetyl/baetyl-function-python36) pulled" baetyl=master engine=docker
time="2019-09-23T21:49:09+08:00" level=debug msg="image (hub.baidubce.com/baetyl/baetyl-function-manager) pulled" baetyl=master engine=docker
time="2019-09-23T21:49:09+08:00" level=debug msg="image (hub.baidubce.com/baetyl/baetyl-function-node85) pulled" baetyl=master engine=docker
time="2019-09-23T21:49:09+08:00" level=debug msg="image (hub.baidubce.com/baetyl/baetyl-function-sql) pulled" baetyl=master engine=docker
time="2019-09-23T21:49:09+08:00" level=debug msg="image (hub.baidubce.com/baetyl/baetyl-function-python27) pulled" baetyl=master engine=docker
time="2019-09-23T21:49:09+08:00" level=debug msg="image (hub.baidubce.com/baetyl/baetyl-hub) pulled" baetyl=master engine=docker
time="2019-09-23T21:49:09+08:00" level=debug msg="localhub replica: 1" baetyl=master engine=docker service=localhub
time="2019-09-23T21:49:09+08:00" level=debug msg="instance (localhub) not found" baetyl=master engine=docker service=localhub
time="2019-09-23T21:49:11+08:00" level=debug msg="container (52baddd7a1cf:localhub) started" baetyl=master engine=docker
time="2019-09-23T21:49:11+08:00" level=info msg="instance started" baetyl=master cid=52baddd7a1cf engine=docker instance=localhub service=localhub
time="2019-09-23T21:49:11+08:00" level=info msg="service (localhub) started" baetyl=master
time="2019-09-23T21:49:11+08:00" level=debug msg="function-manager replica: 1" baetyl=master engine=docker service=function-manager
time="2019-09-23T21:49:11+08:00" level=debug msg="instance (function-manager) not found" baetyl=master engine=docker service=function-manager
time="2019-09-23T21:49:14+08:00" level=debug msg="container (db939cb5cdfc:function-manager) started" baetyl=master engine=docker
time="2019-09-23T21:49:14+08:00" level=info msg="instance started" baetyl=master cid=db939cb5cdfc engine=docker instance=function-manager service=function-manager
time="2019-09-23T21:49:14+08:00" level=info msg="service (function-manager) started" baetyl=master
time="2019-09-23T21:49:14+08:00" level=debug msg="function-python27-sayhi replica: 0" baetyl=master engine=docker service=function-python27-sayhi
time="2019-09-23T21:49:14+08:00" level=info msg="service (function-python27-sayhi) started" baetyl=master
time="2019-09-23T21:49:14+08:00" level=debug msg="function-python36-sayhi replica: 0" baetyl=master engine=docker service=function-python36-sayhi
time="2019-09-23T21:49:14+08:00" level=info msg="service (function-python36-sayhi) started" baetyl=master
time="2019-09-23T21:49:14+08:00" level=debug msg="function-node85-sayhi replica: 0" baetyl=master engine=docker service=function-node85-sayhi
time="2019-09-23T21:49:14+08:00" level=info msg="service (function-node85-sayhi) started" baetyl=master
time="2019-09-23T21:49:14+08:00" level=debug msg="function-sql-filter replica: 0" baetyl=master engine=docker service=function-sql-filter
time="2019-09-23T21:49:14+08:00" level=info msg="service (function-sql-filter) started" baetyl=master
time="2019-09-23T21:49:14+08:00" level=info msg="app version (v0) committed" baetyl=master
time="2019-09-23T21:49:14+08:00" level=info msg="services started" baetyl=master
time="2019-09-23T21:50:11+08:00" level=info msg="[PUT] /v1/services/localhub/instances/localhub/report" api=server baetyl=master
```

Baetyl 加载、启动日志

同样，我们也可以通过执行命令 `docker ps` 查看系统当前正在运行的 docker 容器列表，具体如下图示。

```
baetyl@baetyl:/usr/local/var$ docker ps
CONTAINER ID        IMAGE                                          COMMAND                  CREATED             STATUS              PORTS
 NAMES
db939cb5cdfc        hub.baidubce.com/baetyl/baetyl-function-manager   "baetyl-function-man…"   4 minutes ago       Up 4 minutes
 function-manager
52baddd7a1cf        hub.baidubce.com/baetyl/baetyl-hub             "baetyl-hub"             4 minutes ago       Up 4 minutes        0.0.0.0:1883->1883/tcp
 localhub
```

通过 `docker ps` 命令查看系统当前运行 docker 容器列表

经过对比，不难发现，本次 Baetyl 启动时已经成功加载了 Hub 服务和函数计算服务，函数运行时服务实例并没有启动，因为函数运行时服务实例在有消息触发时才会动态创建。

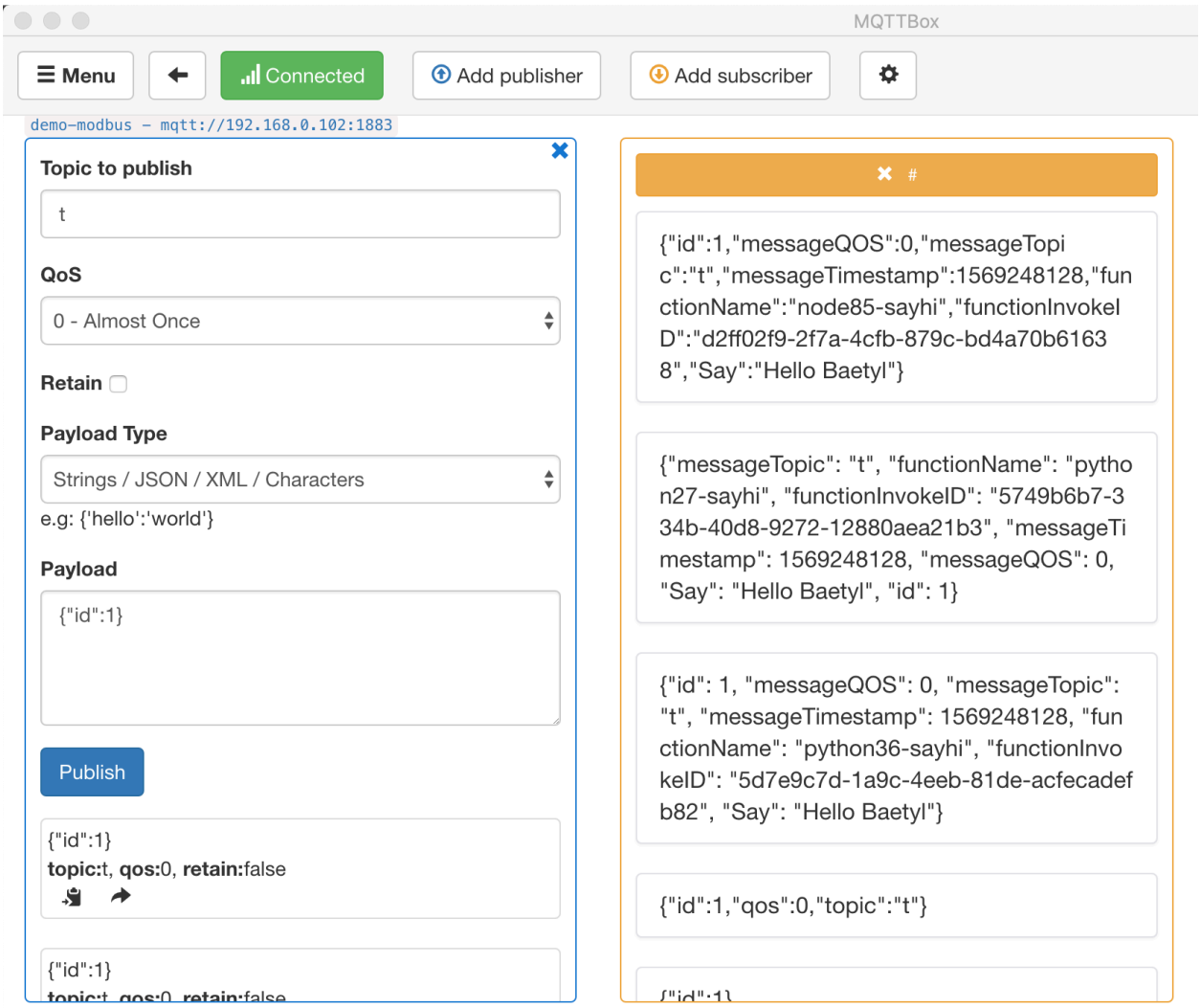
9.2.2 MQTTBox 建立连接

本次测试中，我们采用 TCP 连接方式对 MQTTBox 进行连接信息配置，然后点击 `Add subscriber` 按钮订阅主题 `#` ，该主题用于接收所有 Hub 服务收到的消息。

9.2.3 消息处理验证

通过查看 `/usr/local/var/db/baetyl/function-sayhi-code/index.py` 代码文件可以发现，在接收到某字典类格式的消息后，函数 `handler` 会对其进行一系列处理，然后将处理结果返回。返回的结果中包括各种追加的上下文信息，比如 `messageTopic`、`functionName` 等。

这里，我们通过 MQTTBox 将消息 `{"id":1}` 发布给主题 `t` ，然后观察 MQTTBox 接收到的消息如下。



MQTTBox 接收消息

发送消息后，我们快速执行命令 `docker ps` 查看系统当前正在运行的容器列表，所有函数运行时服务实例都被启动了，其结果如下图示。

```
baetyl@baetyl:/usr/local/var$ docker ps
CONTAINER ID        IMAGE                                                COMMAND                  CREATED             STATUS              PORTS
  NAMES
7410649788db        hub.baidubce.com/baetyl/baetyl-function-python36     "function-python36.py"   8 seconds ago       Up 4 seconds
  function-python36-sayhi.python36-sayhi.3
56073223ab5a        hub.baidubce.com/baetyl/baetyl-function-python27     "function-python27.py"   8 seconds ago       Up 3 seconds
  function-python27-sayhi.python27-sayhi.3
5d64c1e02c24        hub.baidubce.com/baetyl/baetyl-function-node85       "function-node85.js"     8 seconds ago       Up 4 seconds
  function-node85-sayhi.node85-sayhi.3
925ab2b2bdda        hub.baidubce.com/baetyl/baetyl-function-sql          "baetyl-function-sql"    14 minutes ago      Up 14 minutes
  function-sql-filter.sql-filter.1
db939cb5cdfc        hub.baidubce.com/baetyl/baetyl-function-manager      "baetyl-function-man…"   32 minutes ago      Up 32 minutes
  function-manager
52baddd7a1cf        hub.baidubce.com/baetyl/baetyl-hub                   "baetyl-hub"             32 minutes ago      Up 32 minutes       0.0.0.0:1883->1883/tcp
  localhub
```

通过 `docker ps` 命令查看系统当前正在运行的容器列表

综上，我们通过 Hub 服务和函数计算服务模拟了消息在本地处理的过程，可以看出该框架非常适合用于边缘处理消息流。

CHAPTER 10

利用 Remote 远程服务实现 Baetyl 与百度 IoTHub 消息同步

声明:

- 本文测试所用设备系统为 Ubuntu 18.04
- 本文测试前先安装 Baetyl，并导入示例配置包，可参考*快速安装 Baetyl*
- 模拟 MQTT client 向百度云 IoTHub 订阅消息的客户端为 MQTT.fx
- 模拟 MQTT client 向本地 Hub 服务发送消息的客户端为 MQTTBox
- 远程 Hub 接入平台选用 Baidu IoTHub

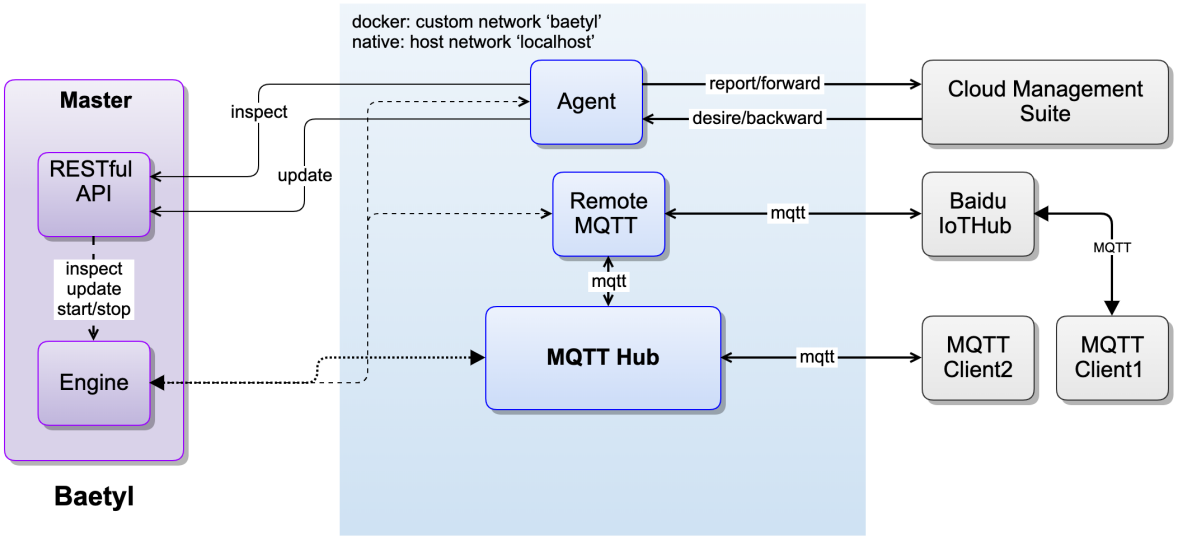
Remote 远程服务模块是为了满足物联网场景下另外一种用户需求而研发，能够实现本地 Hub 与远程 Hub 服务（如Baidu IoTHub等）的数据同步。即通过 Remote 远程服务模块我们既可以从远程 Hub 订阅消息到本地 Hub，也可以将本地 Hub 的消息发送给远程 Hub，完整的配置可参考 Remote 模块配置。

10.1 操作流程

- Step 1：依据 Baidu IoTHub 的操作规章，在 Baidu IoTHub 创建测试所用的 endpoint、user、principal（身份）、policy（主题权限策略）等信息；
- Step 2：依据步骤 `Step 1` 中创建的连接信息，选择 MQTT.fx 作为测试用 MQTT 客户端，配置相关连接信息，并将之与 Baidu IoTHub 建立连接，并订阅既定主题；
  - 若成功建立连接，则继续下一步操作；
  - 若未成功建立连接，则重复上述步骤，直至看到 MQTT.fx 与 Baidu IoTHub 成功建立连接。

- Step 3：打开终端，执行 `sudo systemctl start baetyl` 以容器模式启动 Baetyl 可执行程序（要求 Baetyl 已事先在设备上部署完毕，相关内容可参考快速安装 *Baetyl*），然后执行 `sudo systemctl status baetyl` 来查看 Baetyl 是否正常运行，并观察 Hub 模块、Remote 模块启动状态；
  - 若 Hub、Remote 模块成功启动，则继续下一步操作；
  - 若 Hub、Remote 模块未成功启动，则重复 `Step 3`，直至看到 Hub、Remote 模块成功启动。
- Step 4：选择 MQTTBox 作为测试用 MQTT 客户端，与 Hub 模块建立连接，并订阅既定主题；
  - 若成功与 Hub 模块建立连接，则继续下一步操作；
  - 若与 Hub 建立连接失败，则重复 `Step 4` 操作，直至 MQTTBox 与本地 Hub 模块成功建立连接。
- Step 5：依据 Remote 模块的相关配置信息，从 MQTTBox 向既定主题发布消息，观察 MQTT.fx 的消息接收情况；同理，从 MQTT.fx 向既定主题发布消息，观察 MQTTBox 的消息接收情况。
- Step 6：若 `Step 5` 中双方均能接收到对方发布的消息内容，则表明功能测试顺利通过。

上述操作流程相关的流程示意图具体如下图示。



使用 Remote 模块进行消息同步

10.2 Remote 模块消息远程同步

Baetyl 位于 `var/db/baetyl/application.yml` 的应用配置如下：

```
version: v0
services:
  - name: localhub
    image: hub.baidubce.com/baetyl/baetyl-hub:latest
```

(下页继续)

(续上页)

```
    replica: 1
    ports:
      - 1883:1883
    mounts:
      - name: localhub-conf
        path: etc/baetyl
        readonly: true
      - name: localhub-data
        path: var/db/baetyl/data
      - name: localhub-log
        path: var/log/baetyl
   - name: remote-iothub
    image: hub.baidubce.com/baetyl/baetyl-remote-mqtt:latest
    replica: 1
    mounts:
      - name: remote-iothub-conf
        path: etc/baetyl
        readonly: true
      - name: remote-iothub-cert
        path: var/db/baetyl/cert
        readonly: true
      - name: remote-iothub-log
        path: var/log/baetyl
volumes:
  # hub
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf
  - name: localhub-data
    path: var/db/baetyl/localhub-data
  - name: localhub-log
    path: var/db/baetyl/localhub-log
  # remote mqtt
  - name: remote-iothub-conf
    path: var/db/baetyl/remote-iothub-conf
  - name: remote-iothub-cert
    path: var/db/baetyl/remote-iothub-cert
  - name: remote-iothub-log
    path: var/db/baetyl/remote-iothub-log
```

Hub 模块的配置文件位于 `var/db/baetyl/localhub-conf/service.yml`，具体配置信息如下：

```
listen:
  - tcp://0.0.0.0:1883
principals:
  - username: test
    password: hahaha
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
        permit: ['#']
logger:
  path: var/log/baetyl/localhub-service.log
  level: "debug"
```

Remote 模块的配置文件位置 `var/db/baetyl/remote-iothub-conf/service.yml`，配置信息如下：

```
name: remote-iothub
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
remotes:
  - name: iothub
    address: '<iothub_endpoint>' # 从物接入的项目列表中复制 ssl 地址替换 <iothub_endpoint>,
比如：ssl://xxxxxx.mqtt.iot.gz.baidubce.com:1884，xxxxxx 为 endpoint
    clientid: remote-iothub-1
    username: '<username>' # 从上面选定（address）的物接入项目下创建的用户名列表中复制支持␣
↪ssl 连接的用户名替换 <username>，比如：xxxxxx/test，xxxxxx 为 endpoint
    ca: var/db/baetyl/cert/ca.pem
    cert: var/db/baetyl/cert/client.pem
    key: var/db/baetyl/cert/client.key
rules:
  - hub:
      subscriptions:
        - topic: t1
    remote:
      name: iothub
      subscriptions:
        - topic: t2
          qos: 1
logger:
```

（下页继续）

(续上页)

```
path: var/log/baetyl/remote-service.log
level: 'debug'
```

依据上述 Remote 模块的配置信息，意即 Remote 模块向本地 Hub 模块订阅主题 `t1` 的消息，向 Baidu IoTHub 订阅主题 `t2` 的消息；当 MQTTBox 向主题 `t1` 发布消息时，Hub 模块接收到主题 `t1` 的消息后，将其转发给 Remote 模块，再由 Remote 模块将之转发给 Baidu IoTHub，这样如果 MQTT.fx 订阅了主题 `t1`，即会收到该条从 MQTTBox 发布的消息；同理，当 MQTT.fx 向主题 `t2` 发布消息时，Baidu IoTHub 会将消息转发给 Remote 模块，由 Remote 模块将之转发给本地 Hub 模块，这样如果 MQTTBox 订阅了主题 `t2`，即会收到该消息。

简单来说，由 MQTT.fx 发布的消息，到 MQTTBox 接收到该消息，流经的路径信息为：

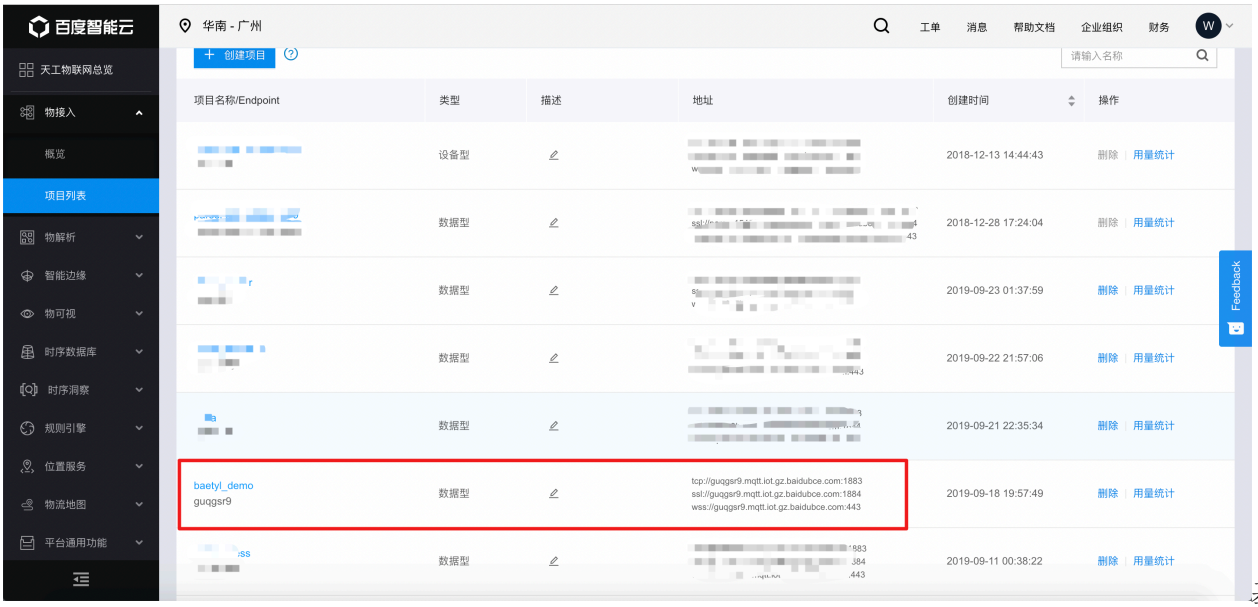
MQTT.fx -> Remote Hub -> Remote Module -> Local Hub Module -> MQTTBox

同样，由 MQTTBox 发布的消息，到 MQTT.fx 接收到该消息，流经的路径信息为：

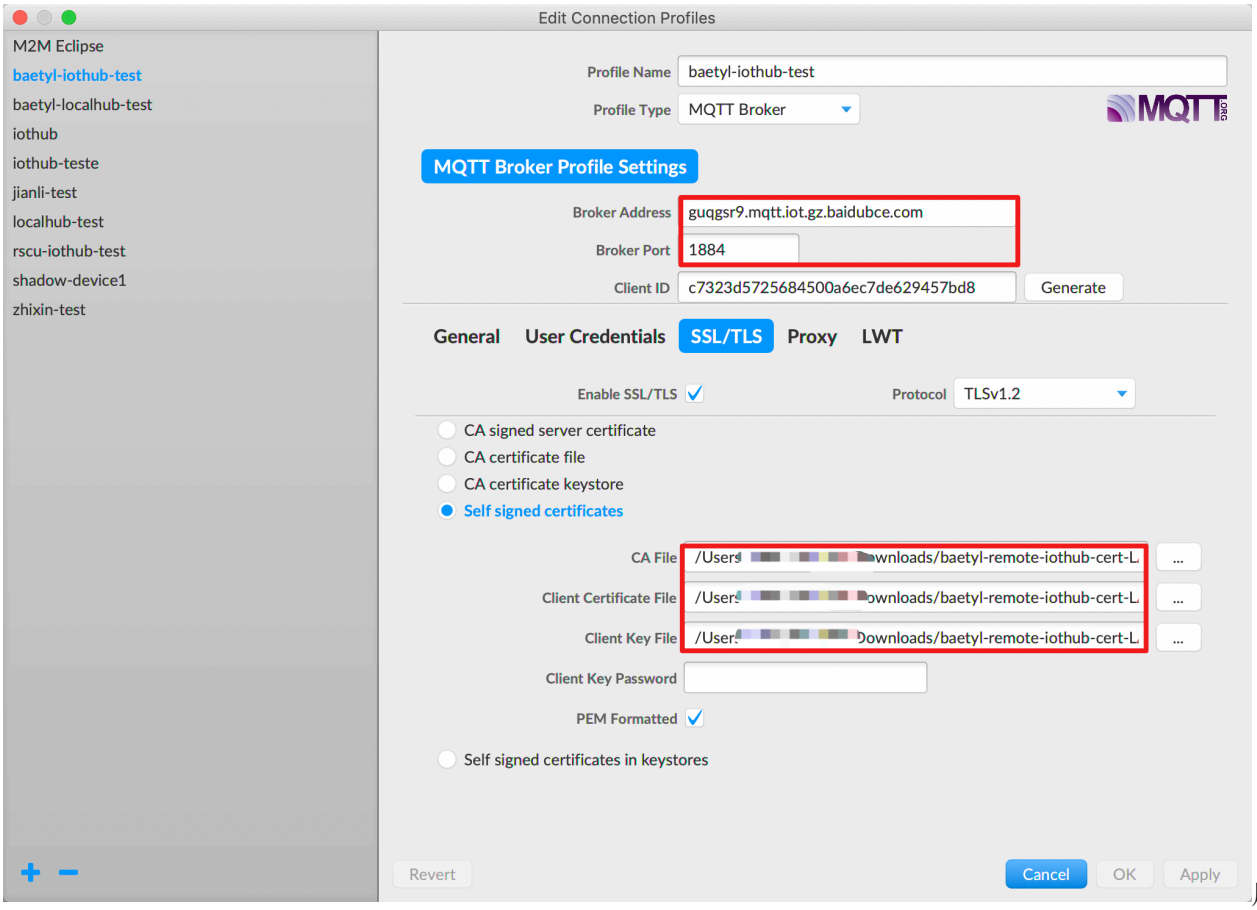
MQTTBox -> Local Hub Module -> Remote Module -> Remote Hub -> MQTT.fx

10.2.1 通过 MQTT.fx 与 Baidu IoTHub 建立连接

如 `Step 1`，`Step 2` 所述，通过 MQTT.fx 与 Baidu IoTHub 建立连接，涉及的通过云端 Baidu IoTHub 场景的 `endpoint` 等相关信息，及 MQTT.fx 连接配置信息分别如下图示。

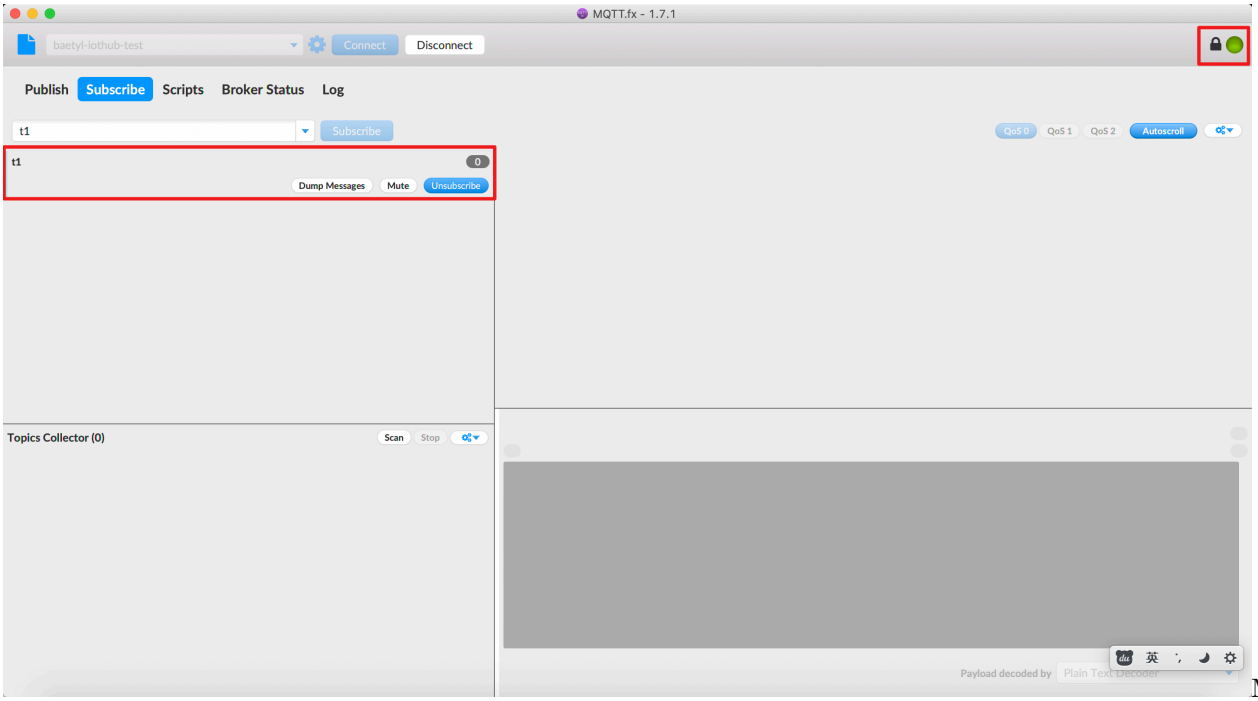


基于 Baidu IoTHub 创建的 endpoint



用于连接 Baidu IoTHub 的 MQTT.fx 配置信息

完成连接信息的相关配置工作后，点击 `OK` 或 `Apply` 按钮使配置信息生效，然后在 MQTT.fx 连接操作页面点击 `Connect` 按钮，通过按钮的 **颜色**（成功建立连接后，右上方指示灯变为 **绿色**）即可判断 MQTT.fx 是否已与 Baidu IoTHub 建立连接，在建立连接后，切换至 `Subscribe` 页面，依据既定配置，订阅相应主题 `t1`，相关示意图如下图示。



MQTT.fx 成功与 Baidu IoTHub 建立连接

10.2.2 通过 MQTTBox 与本地 Hub 模块建立连接

依据步骤 `Step 3` 所述，调整 Baetyl 主程序启动加载配置项，执行 `sudo systemctl start baetyl` 以容器模式启动 Baetyl，这里，要求 Baetyl 启动后加载 Hub、Remote 模块，执行 `sudo systemctl status baetyl` 来查看 `baetyl` 是否正常运行，成功加载的状态如下图示。

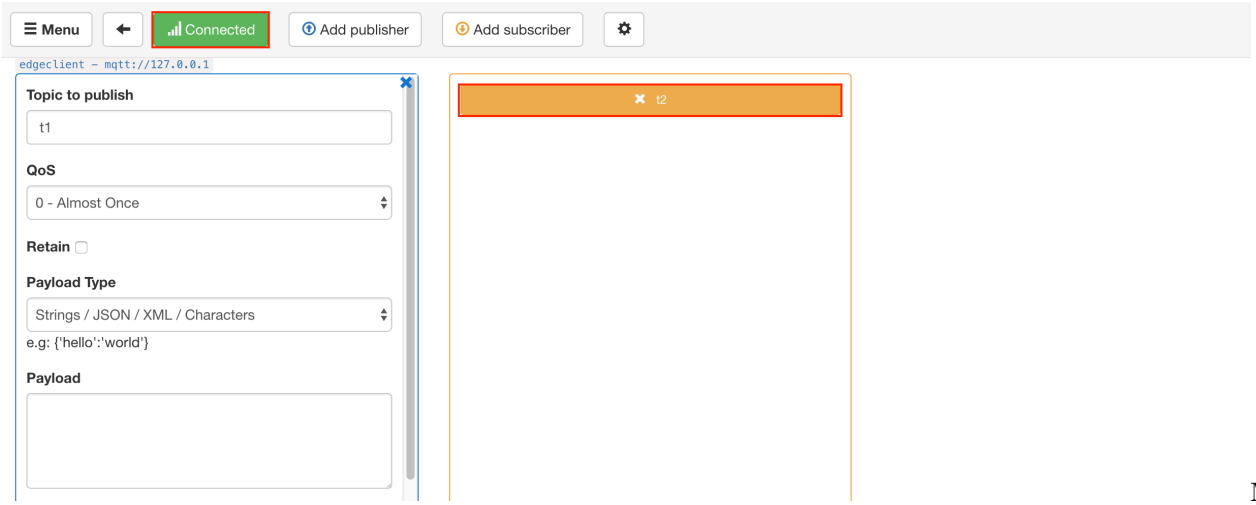
```
baetyl@baetyl:~$ sudo systemctl status baetyl
● baetyl.service - Baetyl
   Loaded: loaded (/lib/systemd/system/baetyl.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2019-09-23 11:31:57 CST; 5h 55min ago
 Main PID: 997 (baetyl)
    Tasks: 10 (limit: 4623)
   CGroup: /system.slice/baetyl.service
           └─997 /usr/local/bin/baetyl start

9月  23 11:31:57 baetyl systemd[1]: Started Baetyl.
```

Baetyl 状态

此外，亦可通过执行命令 `docker stats` 查看系统当前正在运行的 docker 容器列表，具体如下图示。

```
➜ baetyl-demo docker ps
CONTAINER ID        IMAGE                                                
```

示。



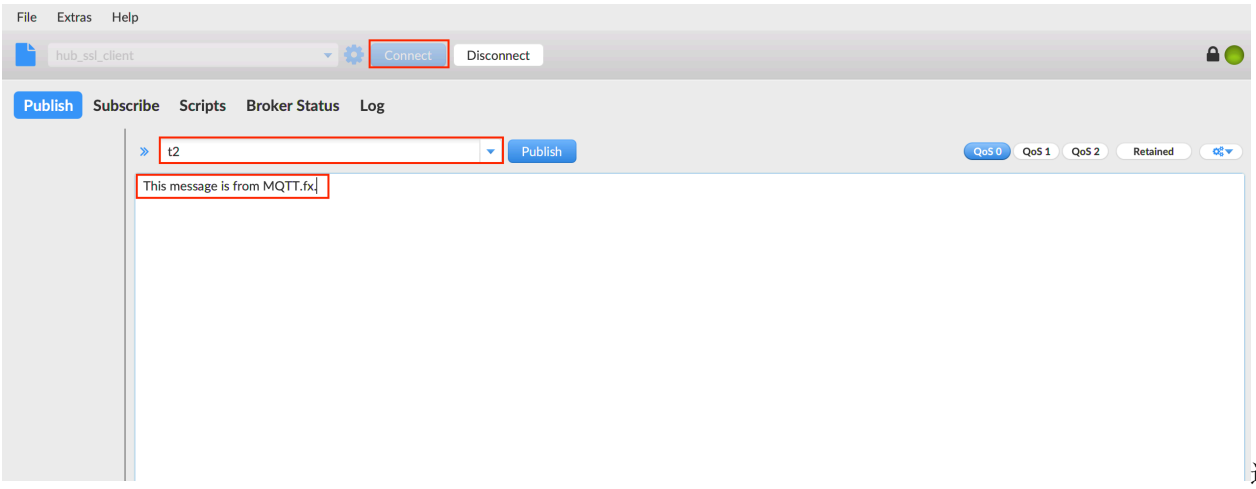
MQTTBox 成功订阅主题 t2

10.2.3 Remote 消息远程同步

这里，将分别以 MQTT.fx、MQTTBox 作为消息发布方，另一方作为消息接收方进行测试。

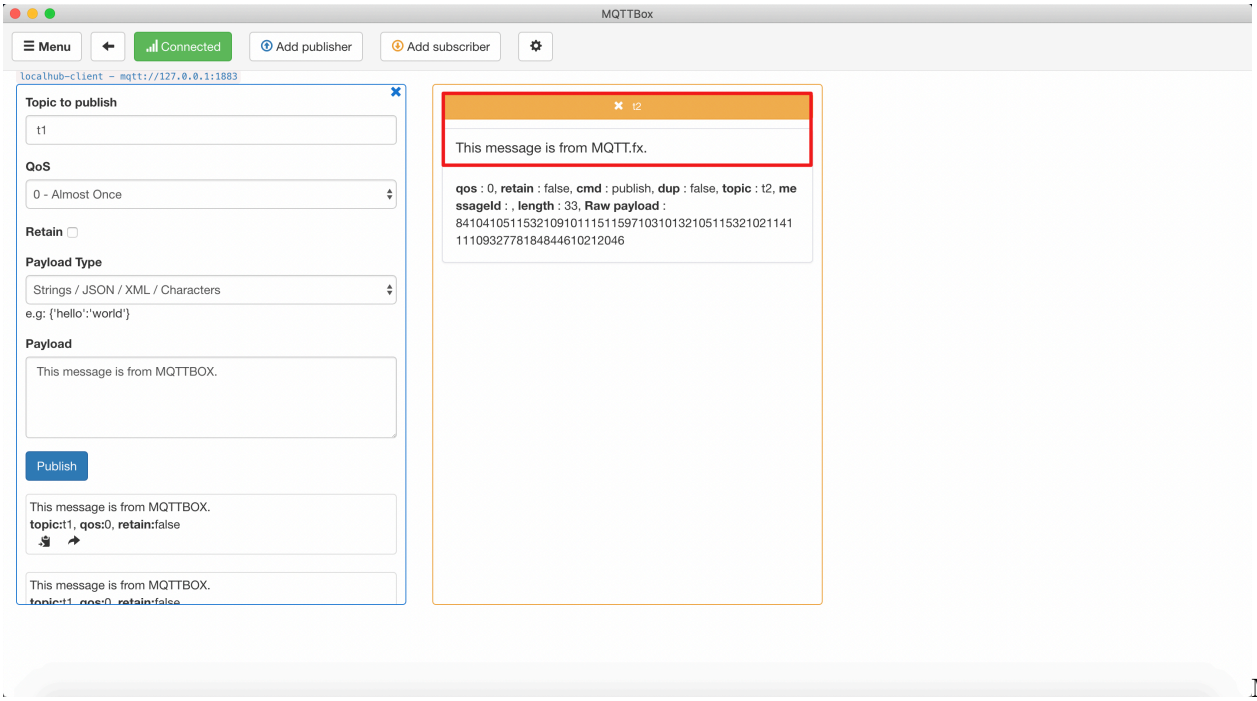
MQTT.fx 发布消息，MQTTBox 接收消息

首先，通过 MQTT.fx 向主题 `t2` 发布消息 `This message is from MQTT.fx.`，具体如下图示。



通过 MQTT.fx 向主题 t2 发布消息

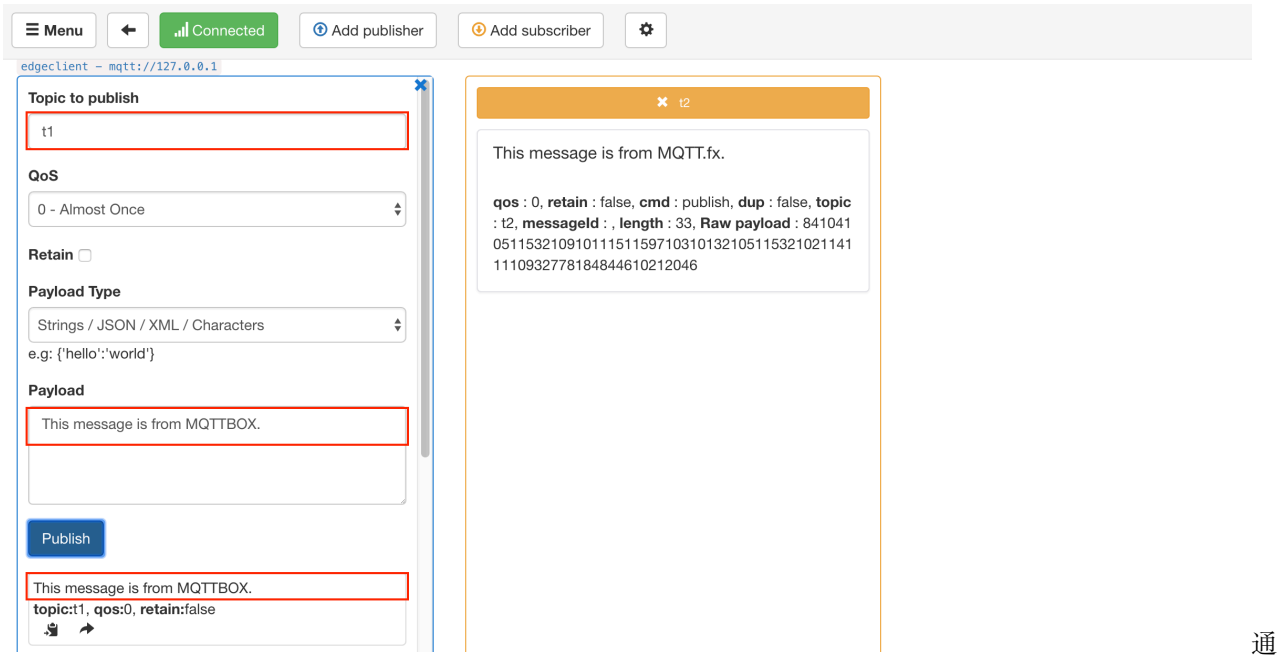
同时，观察 MQTTBox 在订阅主题 `t2` 的消息接收状态，具体如下图示。



MQTTBox 成功收到消息

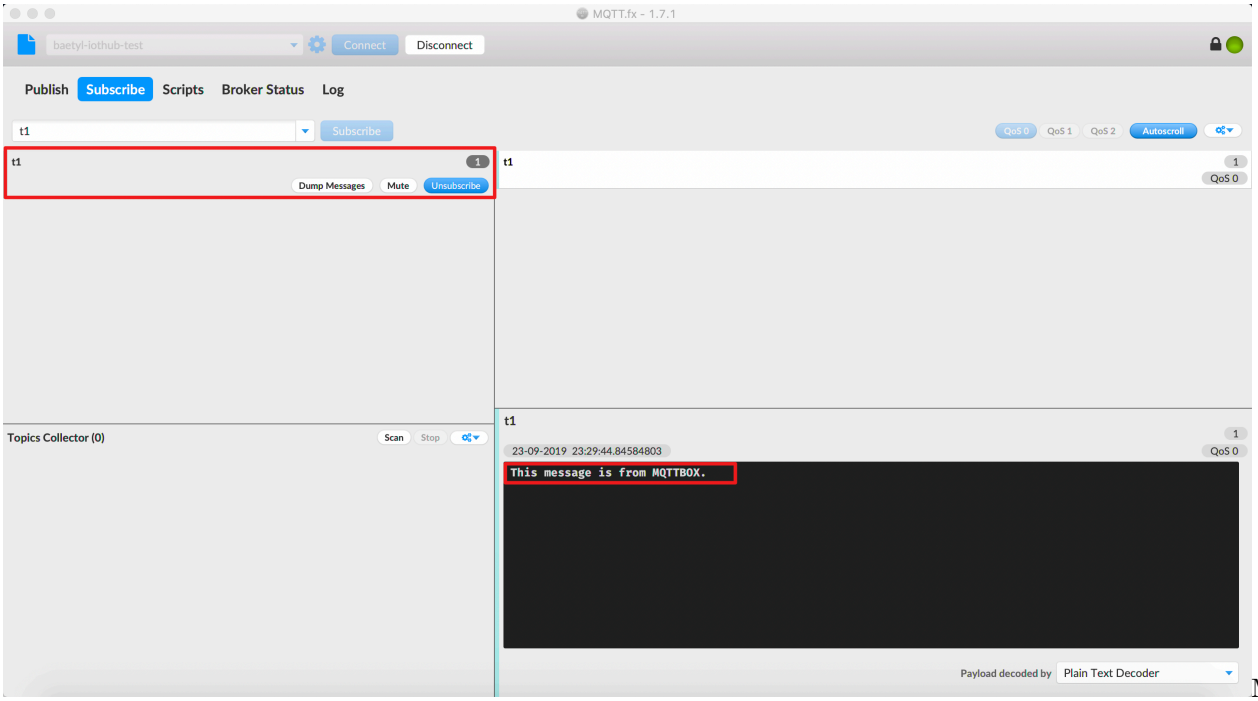
MQTTBox 发布消息，MQTT.fx 接收消息

同理，通过 MQTTBox 作为发布端向主题 `t1` 发布消息 `This message is from MQTTBox.`，具体如下图示。



通过 MQTTBox 向主题 t1 发布消息

同时，观察 MQTT.fx 在订阅主题 `t1` 的消息接收状态，具体如下图示。



MQTT.fx 成功收到消息

综上，MQTT.fx 与 MQTTBox 均已正确接收到了对应的消息，且内容吻合。

CHAPTER 11

利用 Video infer 服务实现摄像头图像采集与 AI 模型推断

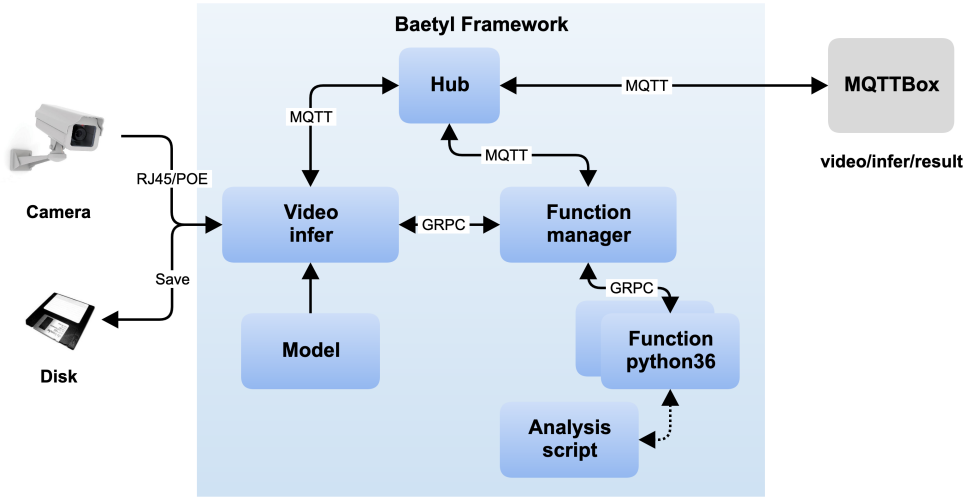
声明:

- 本文测试所用设备系统为 Ubuntu 18.04
- 本文测试所用摄像头为海康威视 IP 网络摄像头，其型号为 DS-IPC-B12-1
- 本文测试的 AI 推断跑在 CPU 使用官方的 video-infer 模块
- 本文测试采用的 AI 模型为 ssd_mobilenet_v1_coco_2017_11_17
- 模拟 MQTT client 向本地 Hub 服务发送消息的客户端为 MQTTBox

11.1 操作流程

- 步骤 1：在 Ubuntu18.04 上安装 Baetyl，相关内容可参考*快速安装 Baetyl*
- 步骤 2：依据本文测试需求，撰写各服务的配置文件，然后执行 `sudo systemctl start baetyl` 命令启动 Baetyl 服务，可以通过 `sudo systemctl status baetyl` 命令查看 Baetyl 服务的运行情况，关于各服务的配置情况请参考下文*各服务配置*
- 步骤 3：根据 Hub 服务的配置信息对 MQTTBox 进行连接配置，确保 MQTTBox 与 Hub 服务建立连接，更多可参考 与 Hub 服务建立连接
- 步骤 4：MQTTBox 订阅主题 `video/infer/result`，观察其是否能够正常接收到模型的推断结果消息。

上述步骤的操作流程示意图如下图所示。



利用 video-infer 服务进行视频采集和 AI 模型推断

11.2 各服务配置

Baetyl 主程序的配置文件位置 `var/db/baetyl/application.yml`，配置信息如下：

```
version: V0
services:
  - name: localhub
    image: 'hub.baidubce.com/baetyl/baetyl-hub:latest'
    replica: 1
    ports:
      - '1883:1883'
    mounts:
      - name: localhub-conf
        path: etc/baetyl
        readonly: true
      - name: localhub-persist-data
        path: var/db/baetyl/data
      - name: demo-log
        path: var/log/baetyl
  - name: function-manager
    image: 'hub.baidubce.com/baetyl/baetyl-function-manager:latest'
    replica: 1
    mounts:
      - name: function-manager-conf
```

(下页继续)

(续上页)

```
        path: etc/baetyl
        readonly: true
      - name: demo-log
        path: var/log/baetyl
  - name: function-python
    image: 'hub.baidubce.com/baetyl/baetyl-function-python36:0.1.6-opencv41'
    replica: 0
    mounts:
      - name: function-python-conf
        path: etc/baetyl
        readonly: true
      - name: function-python-code
        path: var/db/baetyl/code
        readonly: true
      - name: image-data
        path: var/db/baetyl/image
  - name: video-infer
    image: 'hub.baidubce.com/baetyl-beta/baetyl-video-infer:latest'
    replica: 1
    mounts:
      - name: infer-person-model
        path: var/db/baetyl/model
        readonly: true
      - name: image-data
        path: var/db/baetyl/image
      - name: demo-log
        path: var/log/baetyl
      - name: video-infer-conf
        path: etc/baetyl
        readonly: true
volumes:
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf
  - name: localhub-persist-data
    path: var/db/baetyl/localhub-persist-data
  - name: demo-log
    path: var/db/baetyl/demo-log
  - name: function-manager-conf
    path: var/db/baetyl/function-manager-conf
  - name: function-python-conf
```

(下页继续)

(续上页)

```
    path: var/db/baetyl/function-python-conf
  - name: function-python-code
    path: var/db/baetyl/function-python-code
  - name: image-data
    path: var/db/baetyl/image-data
  - name: remote-mqtt-conf
    path: var/db/baetyl/remote-mqtt-conf
  - name: infer-person-model
    path: var/db/baetyl/infer-person-model
  - name: video-infer-conf
    path: var/db/baetyl/video-infer-conf
```

Hub 服务配置文件位置 `var/db/baetyl/localhub-conf/service.yml`，配置信息如下：

```
listen:
  - tcp://0.0.0.0:1883
principals:
  - username: test
    password: hahaha
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
        permit: ['#']
logger:
  path: var/log/baetyl/localhub-service.log
  level: "debug"
```

Function manager 服务配置文件位置 `var/db/baetyl/function-manager-conf/service.yml`，配置信息如下：

```
server:
  address: 0.0.0.0:50051
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
functions:
  - name: analyse
    service: function-python
    instance:
```

(下页继续)

(续上页)

```
      max: 10
logger:
  path: var/log/baetyl/func-service.log
  level: "debug"
```

Function python 服务配置文件位置 `var/db/baetyl/function-python-conf/service.yml`，配置信息如下：

```
functions:
  - name: 'analyse'
    handler: 'analyse.handler'
    codedir: 'var/db/baetyl/code'
logger:
  path: "var/log/baetyl/python-service.log"
  level: "debug"
```

Video infer 服务配置文件位置 `var/db/baetyl/video-infer-conf/service.yml`，配置信息如下：

```
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
video:
  uri: "rtsp://admin:admin@192.168.1.2:554/Streaming/channels/1/"
  limit:
    fps: 1
infer:
  model: var/db/baetyl/model/frozen_inference_graph.pb
  config: var/db/baetyl/model/ssd_mobilenet_v1_coco_2017_11_17.pbtxt
process:
  before:
    swaprb: true
    width: 300
    hight: 300
  after:
    function:
      name: analyse
functions:
  - name: analyse
    address: function-manager:50051
logger:
  path: var/log/baetyl/infer-service.log
```

(下页继续)

(续上页)

```
  level: "debug"
```

需要注意的是，这里 uri 配置代表的是 IP 网络摄像头的地址，其通用格式为 rtsp://<username>:<password>@<ip>:<port>/Streaming/channels/<stream_number>，其中，<username> 和 <password> 为激活成功后的 IP 摄像头的登录认证口令，<ip> 为该摄像头的 IP 地址，<port> 为 RTSP 协议的端口号，默认为 554，后面内容为信道，其中 <stream_number> 为 1 代表抓取主码流，为 2 代表抓取次码流。

此外，Video infer 服务除支持抓取 IP 网络摄像头图片信息外，还支持抓取 USB 摄像头采集图像信息和读取视频文件进行抽帧。相应地，若为 USB 摄像头，则 uri 配置为设备编号，通用性配置为 "0"，同时需要将设备地址 /dev/video0 映射进容器；若配置读取视频文件进行抽帧，则直接配置 uri 内容为视频文件的地址，同时将该视频文件以（自定义）存储卷形式挂载到 Video infer 服务即可，更多关于存储卷的内容可以参考 如何正确地引入存储卷。

Video infer 服务捕获 USB 摄像头采集图像配置可参考：

```
video:
  uri: "0"
  limit:
    fps: 1
```

同时在 application.yml 配置中将 /dev/video0 设备地址映射到容器中，相关配置可参考：

```
version: V0
services:
  - name: video-infer
    image: 'hub.baidubce.com/baetyl-beta/baetyl-video-infer:latest'
    replica: 1
    devices:
      - /dev/video0 # 将 USB 设备映射进容器
    mounts:
      - name: infer-person-model
        path: var/db/baetyl/model
        readonly: true
      - name: image-data
        path: var/db/baetyl/image
      - name: demo-log
        path: var/log/baetyl
      - name: video-infer-conf
        path: etc/baetyl
        readonly: true
```

不难发现，从容器内读取宿主机的 USB 、串口等设备，均可采用上述设备映射的方式配置。

11.3 测试及验证

如本文开头所述，本次测试我们采用 ssd_mobilenet_v1_coco_2017_11_17 模型，该模型可用于检测人、水果等多达 90 项物品。这里，我们给出用于检测人的 Python 脚本仅供参考，具体如下：

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
"""
function to analyse video infer result in python
"""
import numpy as np

location = "var/db/baetyl/image/{}.jpg"
classes = {
    1: 'person'
}

def handler(event, context):
    """
    function handler
    """
    data = np.fromstring(event, np.float32)
    mat = np.reshape(data, (-1, 7))
    objects = []
    scores = {}
    for obj in mat:
        clazz = int(obj[1])
        if clazz in classes:
            score = float(obj[2])
            if classes[clazz] not in scores or scores[classes[clazz]] < score:
                scores[classes[clazz]] = score
            if score < 0.6:
                continue
            objects.append({
                'class': classes[clazz],
                'score': score,
                'left': float(obj[3]),
                'top': float(obj[4]),
                'right': float(obj[5]),
                'bottom': float(obj[6])
            })
```

(下页继续)

(续上页)

```
    res = {}
    res["imageDiscard"] = len(objects) == 0
    res["imageObjects"] = objects
    res["imageScores"] = scores
    res["publishTopic"] = "video/infer/result"
    res["messageTimestamp"] = int(context["messageTimestamp"]/1000000)
    if len(objects) != 0:
        res["imageLocation"] = location.format(context["messageTimestamp"])

    return res
```

若检测其他物品，直接在这里添加或修改即可，可支持检测物品列表参考 mscoco_label_map。

一切就绪后，启动 Baetyl 服务，然后通过 `sudo systemctl status baetyl` 或 `docker ps` 命令查看 Baetyl 服务的运行状态及正在运行的容器列表。

```
[redacted]:/usr/local# systemctl status baetyl
● baetyl.service - Baetyl
   Loaded: loaded (/lib/systemd/system/baetyl.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2019-10-08 12:56:58 CST; 9h ago
 Main PID: 1653 (baetyl)
    Tasks: 16 (limit: 4915)
   CGroup: /system.slice/baetyl.service
           └─1653 /usr/local/bin/baetyl start

10月 08 12:56:58 neousys-POC-351VTC systemd[1]: Started Baetyl.
```

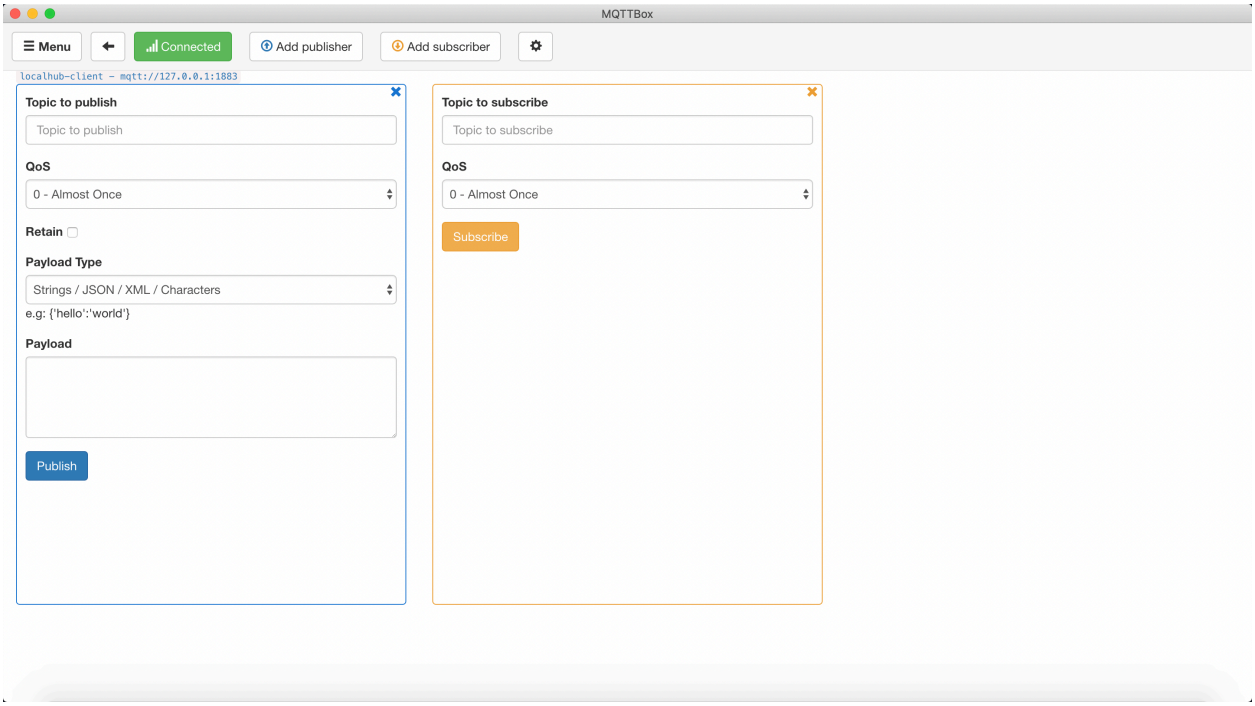
Baetyl 服务运行情况

```
➜  ~ docker ps
CONTAINER ID        IMAGE                                                                COMMAND                  CREATED             STATUS              PORTS
                    NAMES
bde088ebef81        hub.baidubce.com/baetyl/baetyl-function-python36:0.1.6-opencv41      "function-python36.py"   8 seconds ago       Up 7 seconds
                    function-python.analyse.1
3a4fbde4eeef        hub.baidubce.com/baetyl-beta/baetyl-video-infer:0.1.6                "baetyl-video-infer"     10 seconds ago      Up 9 seconds
                    video-infer
8715db0a74a2        hub.baidubce.com/baetyl/baetyl-function-manager:0.1.6                "baetyl-function-man…"   11 seconds ago      Up 9 seconds
                    function-manager
c5032ca6109e        hub.baidubce.com/baetyl/baetyl-hub:0.1.6                             "baetyl-hub"             12 seconds ago      Up 10 seconds       0.0.0.0:1883->1883/tcp, 0.0
.0.0:8883->8883/tcp   localhub
```

正在运行的容器列表

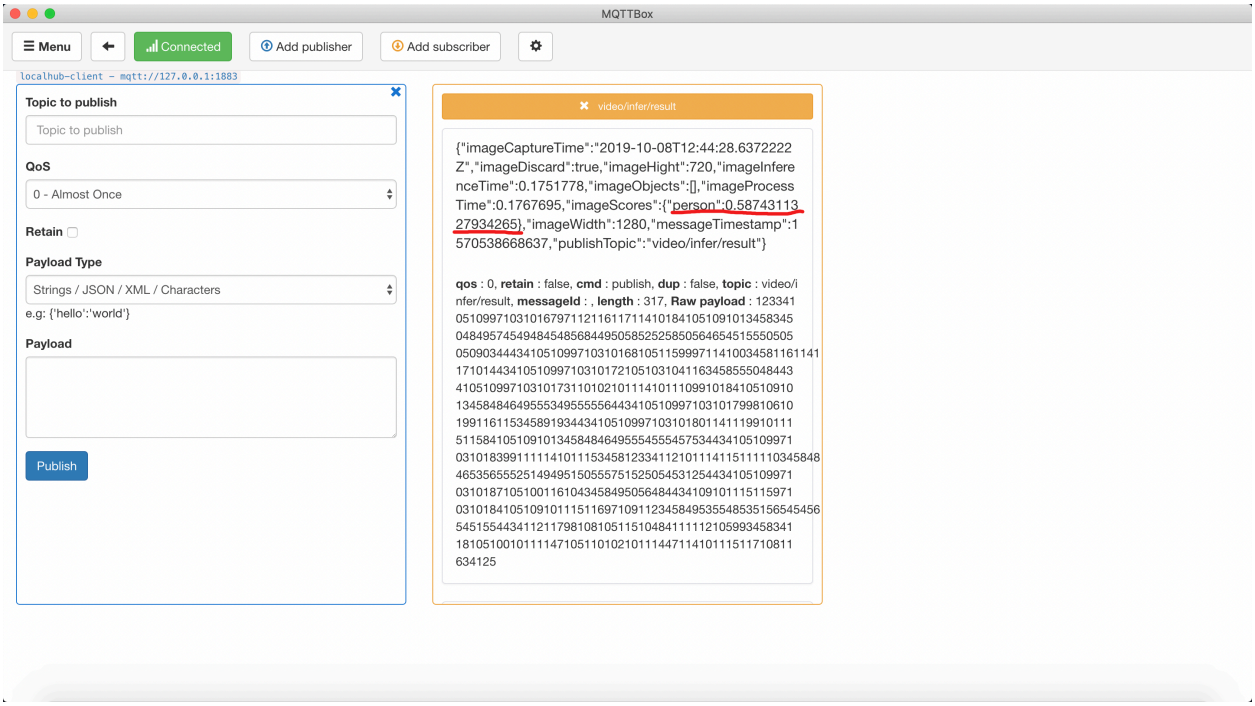
可以发现，Baetyl 服务处于 `active (running)` 状态，系统现在正在运行了 `baetyl-hub`、`baetyl-video-infer`、`baetyl-function-manager` 及 `baetyl-function-python36` 四个容器。

然后，我们启动 MQTTBox，根据 Hub 服务配置完连接信息后，可以发现 MQTTBox 已经与 Hub 服务建立连接。

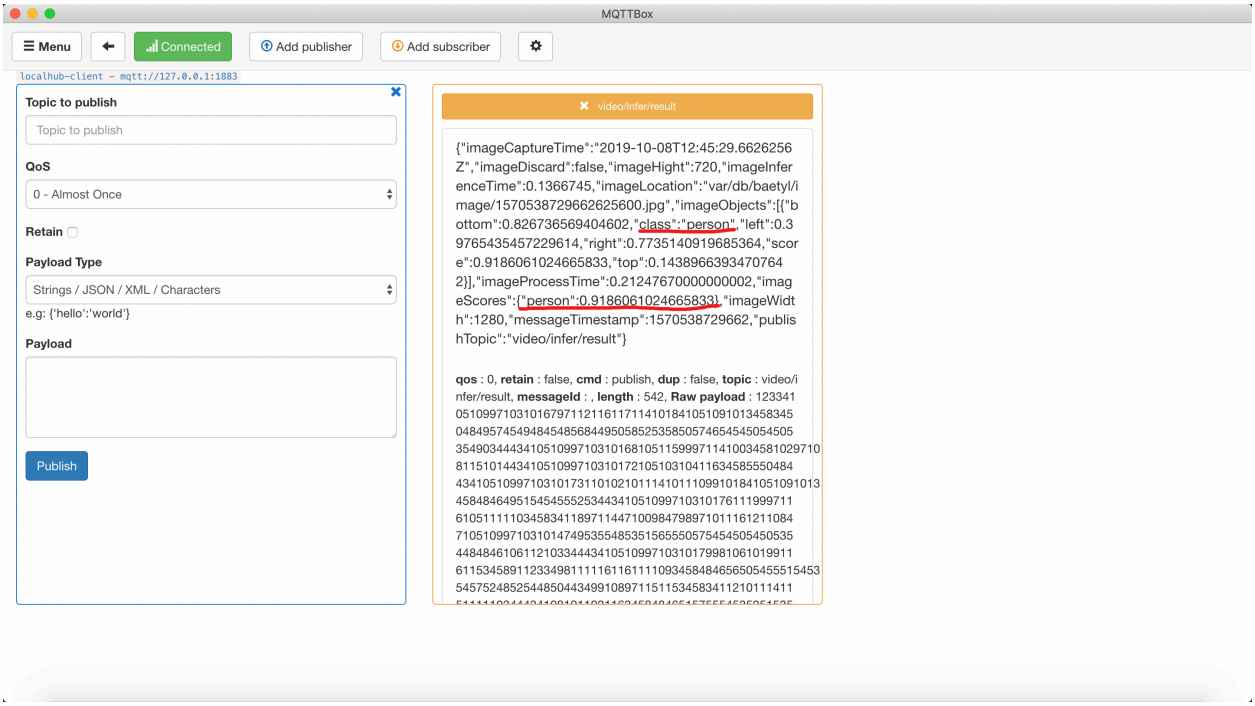


MQTTBox 与 Hub 服务建立连接

然后，通过 MQTTBox 订阅主题 `video/infer/result`，若服务运行正常，则 MQTTBox 会收到消息。



MQTTBox 接收消息，未检测到人



MQTTBox 接收消息，检测到人

如上所示，MQTTBox 通过订阅主题 `video/infer/result` 正确收到了消息。对比上述两条消息，可以发现一个检测到了人，一个未检测到人，如果模型检测到了人，则会在接收到的信息中给出类别 `class` 类别信息，同时给出检测到的人在捕获图片中的位置信息，可以用于后续的画框标记操作，感兴趣的可以参考 代码。

至此，我们已经基于 Video infer 服务实现了 IP 网络摄像头图像的采集和 AI 模型的推断。

CHAPTER 12

如何针对 Python 运行时编写 Python 脚本

声明:

- 本文测试所用设备系统为 Ubuntu18.04
- python 版本为 3.6，2.7 版本配置流程相同，但需要在 python 脚本中注意语言差异
- 模拟 MQTT client 行为的客户端为 MQTTBox
- 本文中基于 Hub 模块创建的服务名称为 `localhub` 服务。并且针对本文的测试案例中，对应的 `localhub` 服务、函数计算服务以及其他服务的配置统一如下：

```
# 本地 Hub 配置
# 配置文件位置: var/db/baetyl/localhub-conf/service.yml
listen:
  - tcp://0.0.0.0:1883
principals:
  - username: 'test'
    password: 'hahaha'
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
        permit: ['#']

# 本地 baetyl-function-manager 配置
```

(下页继续)

(续上页)

```
# 配置文件位置: var/db/baetyl/function-manager-conf/service.yml
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
rules:
  - clientid: localfunc-1
    subscribe:
      topic: py
    function:
      name: sayhi3
    publish:
      topic: py/hi
functions:
  - name: sayhi3
    service: function-sayhi3
    instance:
      min: 0
      max: 10
      idletime: 1m

# python function 配置
# 配置文件位置: var/db/baetyl/function-sayhi-conf/service.yml
functions:
  - name: 'sayhi3'
    handler: 'sayhi.handler'
    codedir: 'var/db/baetyl/function-sayhi'

# application.yml 配置
# 配置文件位置: var/db/baetyl/application.yml
version: v0
services:
  - name: localhub
    image: hub.baidubce.com/baetyl/baetyl-hub
    replica: 1
    ports:
      - 1883:1883
    mounts:
      - name: localhub-conf
        path: etc/baetyl
```

(下页继续)

(续上页)

```
        readonly: true
      - name: localhub-data
        path: var/db/baetyl/data
      - name: localhub-log
        path: var/log/baetyl
  - name: function-manager
    image: hub.baidubce.com/baetyl/baetyl-function-manager
    replica: 1
    mounts:
      - name: function-manager-conf
        path: etc/baetyl
        readonly: true
      - name: function-manager-log
        path: var/log/baetyl
  - name: function-sayhi3
    image: hub.baidubce.com/baetyl/baetyl-function-python36
    replica: 0
    mounts:
      - name: function-sayhi-conf
        path: etc/baetyl
        readonly: true
      - name: function-sayhi-code
        path: var/db/baetyl/function-sayhi
        readonly: true
volumes:
  # hub
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf
  - name: localhub-data
    path: var/db/baetyl/localhub-data
  - name: localhub-log
    path: var/db/baetyl/localhub-log
  # function manager
  - name: function-manager-conf
    path: var/db/baetyl/function-manager-conf
  - name: function-manager-log
    path: var/db/baetyl/function-manager-log
  # function python runtime sayhi
  - name: function-sayhi-conf
    path: var/db/baetyl/function-sayhi-conf
```

(下页继续)

(续上页)

```
  - name: function-sayhi-code
    path: var/db/baetyl/function-sayhi-code
```

Baetyl 官方提供了 Python 运行时，可以加载用户所编写的 Python 脚本。下文将针对 Python 脚本的名称，执行函数名称，输入，输出参数等内容分别进行说明。

12.1 函数名约定

Python 脚本的名称可以参照 Python 的通用命名规范，Baetyl 并未对此做特别限制。如果要应用某 Python 脚本对某条 MQTT 消息做处理，则相应的函数运行时服务的配置如下：

```
functions:
  - name: 'sayhi3'
    handler: 'sayhi.handler'
    codedir: 'var/db/baetyl/function-sayhi'
```

这里，我们关注 handler 这一属性，其中 sayhi 代表脚本名称，后面的 handler 代表该文件中被调用的入口函数。

```
function-sayhi-code/
   __init__.py
   sayhi.py
```

更多函数运行时服务配置请查看 函数运行时服务配置释义。

12.2 参数约定

```
def handler(event, context):
    # do something
    return event
```

Baetyl 官方提供的 Python 运行时支持 2 个参数: event 和 context，下面将分别介绍其用法。

- **event**：根据 MQTT 报文中的 Payload 传入不同参数
  - 若原始 Payload 为一个 Json 数据，则传入经过 json.loads(Payload) 处理后的数据;
  - 若原始 Payload 为字节流、字符串 (非 Json)，则传入原 Payload 数据。
- **context**：MQTT 消息上下文
  - context.messageQOS // MQTT QoS

- context.messageTopic // MQTT Topic
- context.functionName // MQTT functionName
- context.functionInvokeID //MQTT function invokeID
- context.invokeid // 同上，用于兼容 CFC

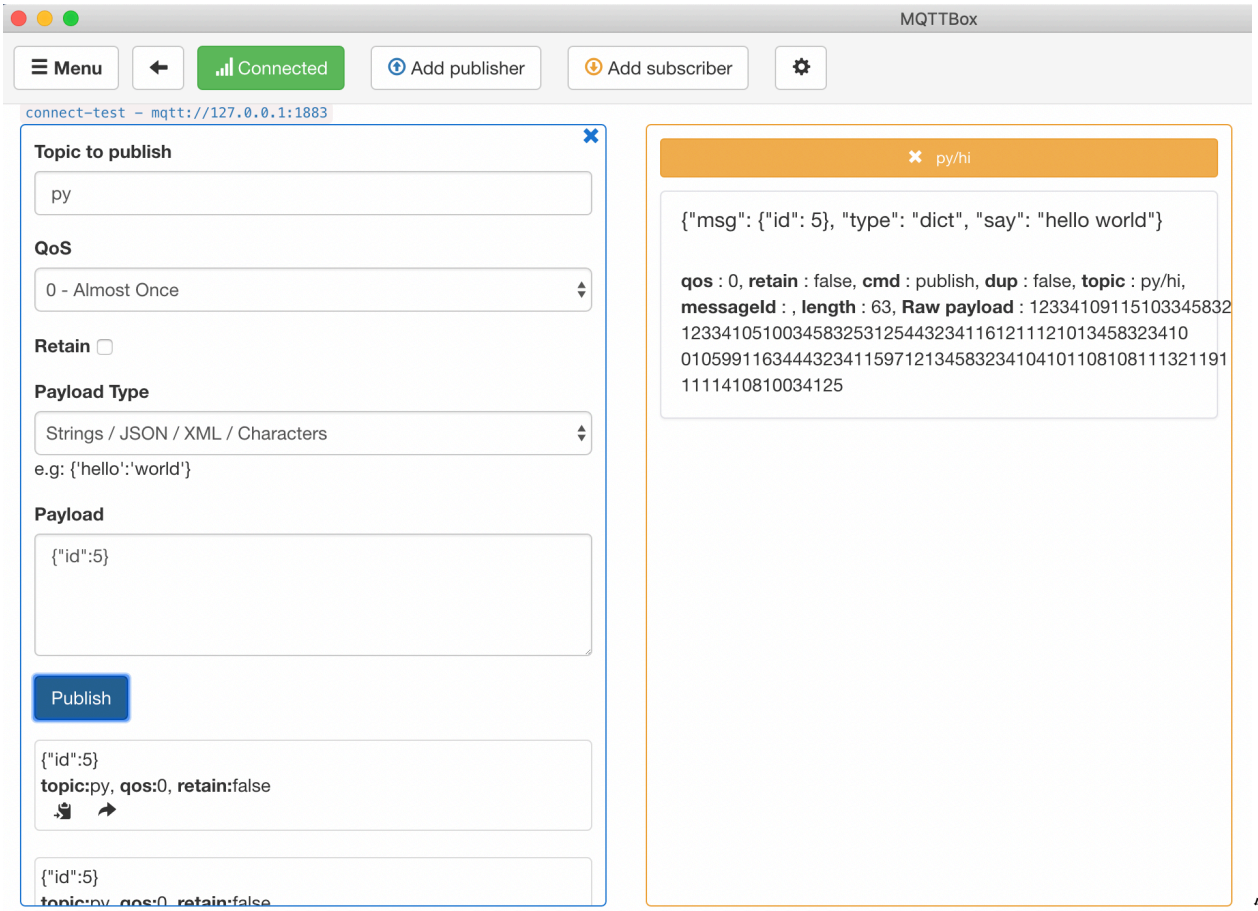
提示：*在云端 CFC 测试时，请注意不要直接使用 Baetyl 定义的上下文信息。推荐做法是先判断字段是否在 context 中存在，如果存在再读取。*

12.3 Hello World!

下面我们实现一个简单的 Python 函数，目标是为每一条流经需要用该 Python 脚本进行处理的 MQTT 消息附加一条 `hello world` 信息。对于字典类消息，将其直接返回即可，对于非字典类消息，则将之转换为字符串后返回。

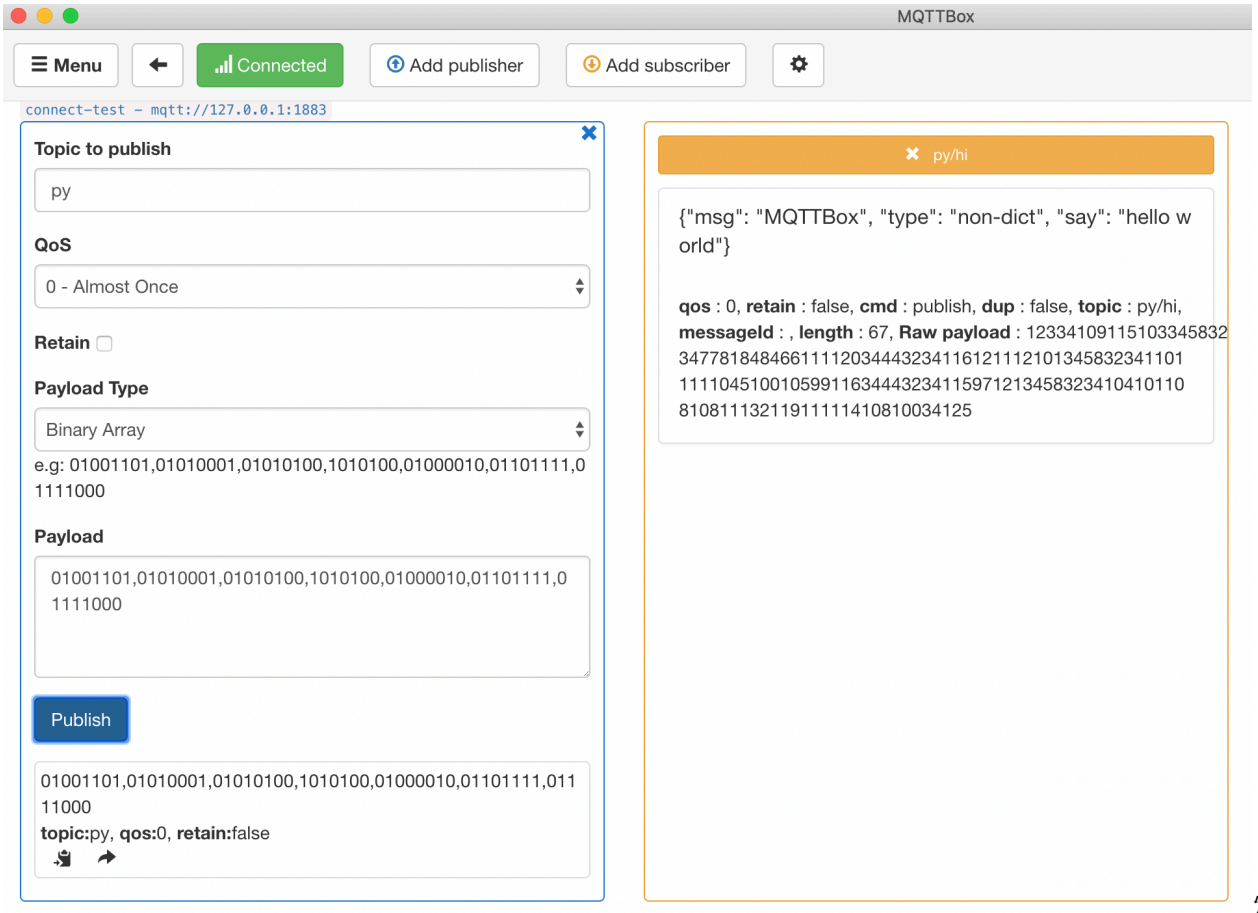
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def handler(event, context):
    result = {}
    if isinstance(event, dict):
        result['msg'] = event
        result['type'] = 'dict'
        result['say'] = 'hello world'
    else:
        result['msg'] = event.decode("utf-8")
        result['type'] = 'non-dict'
        result['say'] = 'hello world'

    return result
```

- **发送字典类数据**:



发送字典类数据

- **发送非字典类数据**:



发送非字典类数据

如上，对于一些常规的需求，我们通过系统 Python 环境的标准库就可以完成。但是，对于一些较为复杂的需求，往往需要引入第三方库来完成。如何解决这个问题？我们将在 如何针对 Python 运行时引入第三方包 小节详述。

CHAPTER 13

如何针对 Node 运行时编写 js 脚本

声明:

- 本文测试所用设备系统为 Ubuntu18.04
- node 版本为 8.5
- 模拟 MQTT client 行为的客户端为 MQTTBox
- 本文中基于 Hub 模块创建的服务名称为 `localhub` 服务。并且针对本文的测试案例中，对应的 `localhub` 服务、函数计算服务以及其他服务的配置统一如下：

```
# 本地 Hub 配置
# 配置文件位置: var/db/baetyl/localhub-conf/service.yml
listen:
  - tcp://0.0.0.0:1883
principals:
  - username: 'test'
    password: 'hahaha'
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
        permit: ['#']

# 本地 baetyl-function-manager 配置
```

(下页继续)

(续上页)

```
# 配置文件位置: var/db/baetyl/function-manager-conf/service.yml
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
rules:
  - clientid: localfunc-1
    subscribe:
      topic: node
    function:
      name: sayhi
    publish:
      topic: t/hi
functions:
  - name: sayhi
    service: function-sayhi
    instance:
      min: 0
      max: 10
      idletime: 1m

# node function 配置
# 配置文件位置: var/db/baetyl/function-sayjs-conf/service.yml
functions:
  - name: 'sayhi'
    handler: 'index.handler'
    codedir: 'var/db/baetyl/function-sayhi'

# application.yml 配置
# 配置文件位置: var/db/baetyl/application.yml
version: v0
services:
  - name: localhub
    image: hub.baidubce.com/baetyl/baetyl-hub
    replica: 1
    ports:
      - 1883:1883
    mounts:
      - name: localhub-conf
        path: etc/baetyl
```

(下页继续)

(续上页)

```
        readonly: true
      - name: localhub-data
        path: var/db/baetyl/data
      - name: localhub-log
        path: var/log/baetyl
  - name: function-manager
    image: hub.baidubce.com/baetyl/baetyl-function-manager
    replica: 1
    mounts:
      - name: function-manager-conf
        path: etc/baetyl
        readonly: true
      - name: function-manager-log
        path: var/log/baetyl
  - name: function-sayhi
    image: hub.baidubce.com/baetyl/baetyl-function-node85
    replica: 0
    mounts:
      - name: function-sayjs-conf
        path: etc/baetyl
        readonly: true
      - name: function-sayjs-code
        path: var/db/baetyl/function-sayhi
        readonly: true
volumes:
  # hub
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf
  - name: localhub-data
    path: var/db/baetyl/localhub-data
  - name: localhub-log
    path: var/db/baetyl/localhub-log
  # function manager
  - name: function-manager-conf
    path: var/db/baetyl/function-manager-conf
  - name: function-manager-log
    path: var/db/baetyl/function-manager-log
  # function node runtime sayhi
  - name: function-sayjs-conf
    path: var/db/baetyl/function-sayjs-conf
```

(下页继续)

(续上页)

```
  - name: function-sayjs-code
    path: var/db/baetyl/function-sayjs-code
```

Baetyl 官方提供了 Node 运行时，可以加载用户所编写的 js 脚本。下文将针对 js 脚本的名称，执行函数名称，输入，输出参数等内容分别进行说明。

13.1 函数名约定

js 脚本的名称可以参照 js 的通用命名规范，Baetyl 并未对此做特别限制。如果要应用某 js 脚本对某条 MQTT 消息做处理，则相应的函数运行时服务的配置如下：

```
functions:
  - name: 'sayhi'
    handler: 'index.handler'
    codedir: 'var/db/baetyl/function-sayhi'
```

这里，我们关注 handler 这一属性，其中 index 代表脚本名称，后面的 handler 代表该文件中被调用的入口函数。

```
function-sayjs-code/
  index.js
```

更多函数运行时服务配置请查看 函数运行时服务配置释义。

13.2 参数约定

```
exports.handler = (event, context, callback) => {
    callback(null, event);
};
```

Baetyl 官方提供的 Node 运行时支持 2 个参数: event 和 context，下面将分别介绍其用法。

- **event**：根据 MQTT 报文中的 Payload 传入不同参数
  - 若原始 Payload 为一个 Json 数据，则传入经过 json.loads(Payload) 处理后的数据;
  - 若原始 Payload 为字节流、字符串 (非 Json)，则传入原 Payload 数据。
- **context**：MQTT 消息上下文
  - context.messageQOS // MQTT QoS
  - context.messageTopic // MQTT Topic

  - context.functionName // MQTT functionName
  - context.functionInvokeID //MQTT function invokeID
  - context.invokeid // 同上，用于兼容 CFC

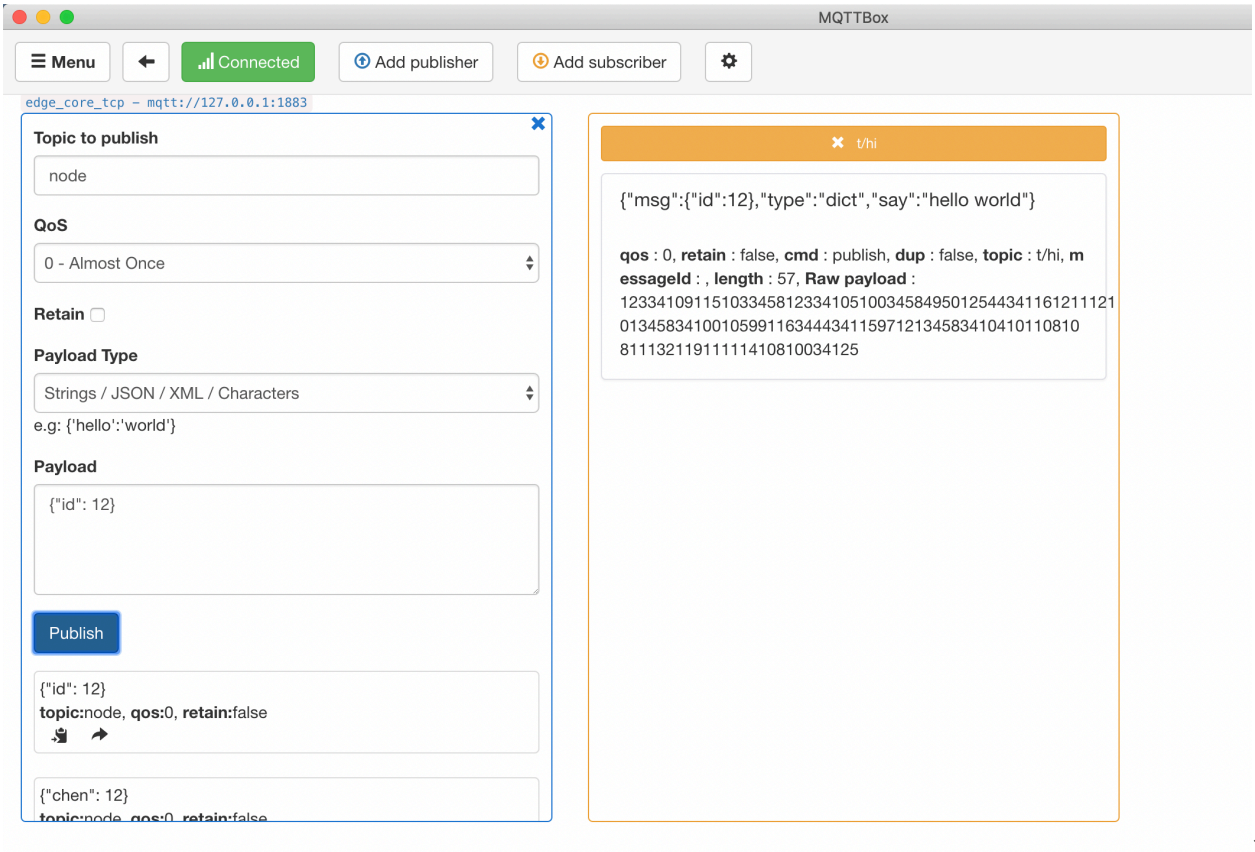
提示：*在云端 CFC 测试时，请注意不要直接使用 Baetyl 定义的上下文信息。推荐做法是先判断字段是否在 context 中存在，如果存在再读取。*

13.3 Hello World!

下面我们实现一个简单的 js 脚本，目标是为每一条流经需要用该 js 脚本进行处理的 MQTT 消息附加一条 `hello world` 信息。对于字典类消息，将其直接返回即可，对于非字典类消息，则将之转换为字符串后返回。

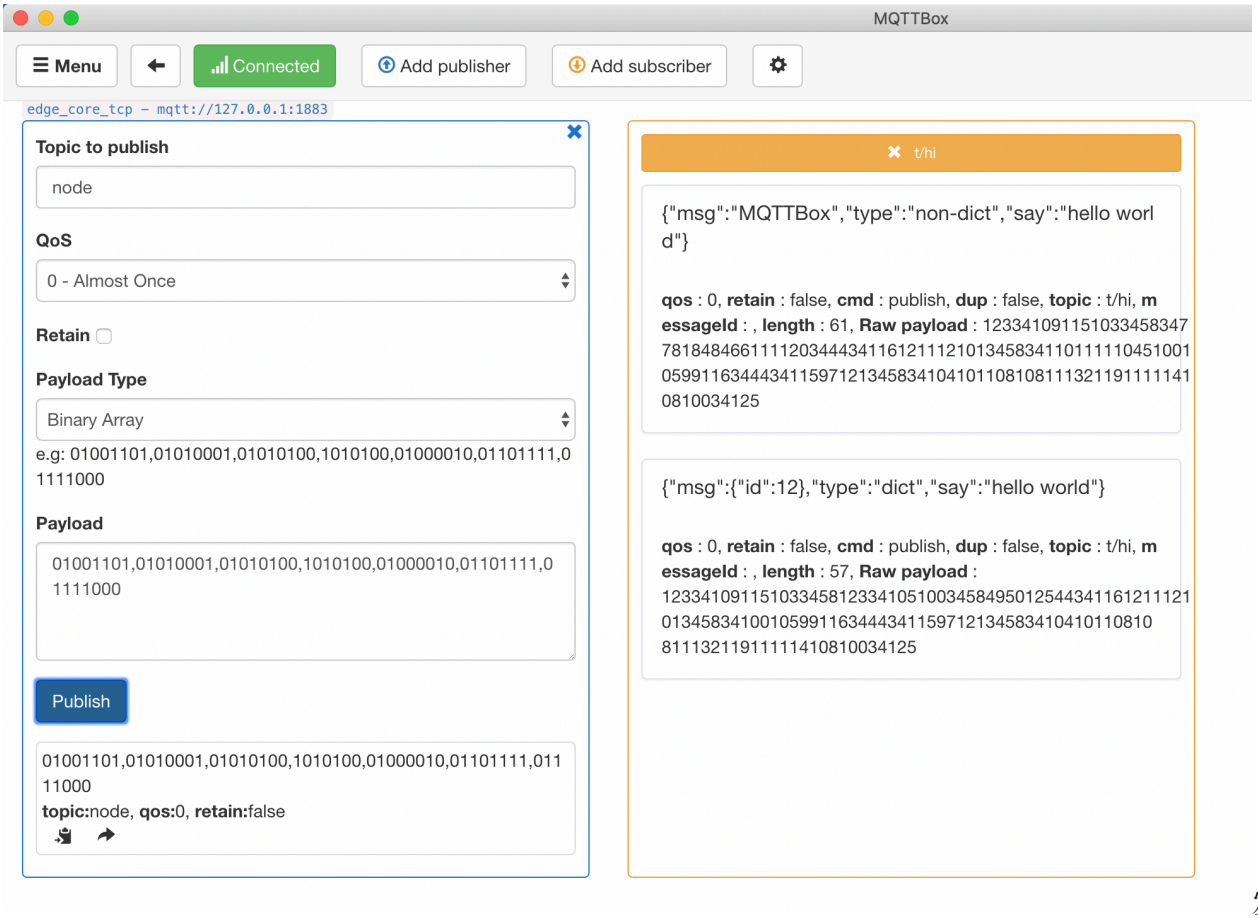
```
#!/usr/bin/env node

exports.handler = (event, context, callback) => {
  result = {};

  if (Buffer.isBuffer(event)) {
      const message = event.toString();
      result["msg"] = message;
      result["type"] = 'non-dict';
  }else {
      result["msg"] = event;
      result["type"] = 'dict';
  }

  result["say"] = 'hello world';
  callback(null, result);
};
```

- **发送字典类数据**:



发送字典类数据

- **发送非字典类数据**:



发送非字典类数据

如上，对于一些常规的需求，我们通过系统 Node 环境的标准库就可以完成。但是，对于一些较为复杂的需求，往往需要引入第三方库来完成。如何解决这个问题？我们将在 如何针对 Node 运行时引入第三方包 小节详述。

CHAPTER 14

如何针对 Python 运行时引入第三方包

声明:

- 本文测试所用设备系统为 Ubuntu18.04
- 运行模式为 **docker** 容器模式，**native** 进程模式配置流程相同
- Python 版本为 3.6，2.7 版本配置流程相同，但需要在 Python 脚本中注意语言差异
- 模拟 MQTT client 行为的客户端为 MQTTBox
- 本文选取 `requests` 和 `Pytorch` 两种第三方包进行演示说明
- 本文中基于 Hub 模块创建的服务名称为 `localhub` 服务。并且针对本文的测试案例中，对应的 `localhub` 服务、函数计算服务以及其他服务的配置统一如下：

```
# localhub 配置
# 配置文件位置: var/db/baetyl/localhub-conf/service.yml
listen:
  - tcp://0.0.0.0:1883
principals:
  - username: 'test'
    password: 'hahaha'
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
```

(下页继续)

(续上页)

```
        permit: ['#']

# 本地 baetyl-function-manager 配置
# 配置文件位置: var/db/baetyl/function-manager-conf/service.yml
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
rules:
  - clientid: localfunc-1
    subscribe:
      topic: py
    function:
      name: sayhi3
    publish:
      topic: py/hi
functions:
  - name: sayhi3
    service: function-sayhi3
    instance:
      min: 0
      max: 10
      idletime: 1m

# application.yml 配置
# 配置文件位置: var/db/baetyl/application.yml
version: v0
services:
  - name: localhub
    image: hub.baidubce.com/baetyl/baetyl-hub
    replica: 1
    ports:
      - 1883:1883
    mounts:
      - name: localhub-conf
        path: etc/baetyl
        readonly: true
      - name: localhub-data
        path: var/db/baetyl/data
      - name: localhub-log
```

(下页继续)

(续上页)

```
        path: var/log/baetyl
  - name: function-manager
    image: hub.baidubce.com/baetyl/baetyl-function-manager
    replica: 1
    mounts:
      - name: function-manager-conf
        path: etc/baetyl
        readonly: true
      - name: function-manager-log
        path: var/log/baetyl
  - name: function-sayhi3
    image: hub.baidubce.com/baetyl/baetyl-function-python36
    replica: 0
    mounts:
      - name: function-sayhi-conf
        path: etc/baetyl
        readonly: true
      - name: function-sayhi-code
        path: var/db/baetyl/function-sayhi
        readonly: true
volumes:
  # hub
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf
  - name: localhub-data
    path: var/db/baetyl/localhub-data
  - name: localhub-log
    path: var/db/baetyl/localhub-log
  # function manager
  - name: function-manager-conf
    path: var/db/baetyl/function-manager-conf
  - name: function-manager-log
    path: var/db/baetyl/function-manager-log
  # function python runtime sayhi
  - name: function-sayhi-conf
    path: var/db/baetyl/function-sayhi-conf
  - name: function-sayhi-code
    path: var/db/baetyl/function-sayhi-code
```

系统自带的 Python 环境有可能不会满足我们的需要，实际使用往往需要引入第三方库，下面给出两个示例。

14.1 引用 requests 第三方包

假定我们想要对一个网站进行爬虫，获取相应的信息。这里，我们可以引入第三方库 requests。如何引入，具体如下所示：

- 步骤 1: 进入 Python 脚本所在目录，然后下载 requests 及其依赖（idna、urllib3、chardet、certifi），并注意 pip 命令对应 Python 的版本

```
cd /directory/of/Python/script
pip download requests
```

- 步骤 2: 解压 .whl 文件，得到源码包，然后删除 .whl 文件和包描述文件，只保留源码包

```
unzip \*.whl
rm -rf *.whl *.dist-info
```

- 步骤 3: 使当前目录成为一个 package

```
touch __init__.py
```

- 步骤 4: 在具体执行脚本中引入第三方库 requests，如下所示：

```
import requests
```

- 步骤 5: 执行脚本

```
python your_script.py
```

如上述操作正常，则形成的脚本目录结构如下图所示。

```
[root@instance-80v5cs0x function-sayhi-code]# tree -L 1
.
├── certifi
├── chardet
├── get.py
├── idna
├── __init__.py
├── requests
└── urllib3

5 directories, 2 files
```

Python requests 第三方库脚本目录

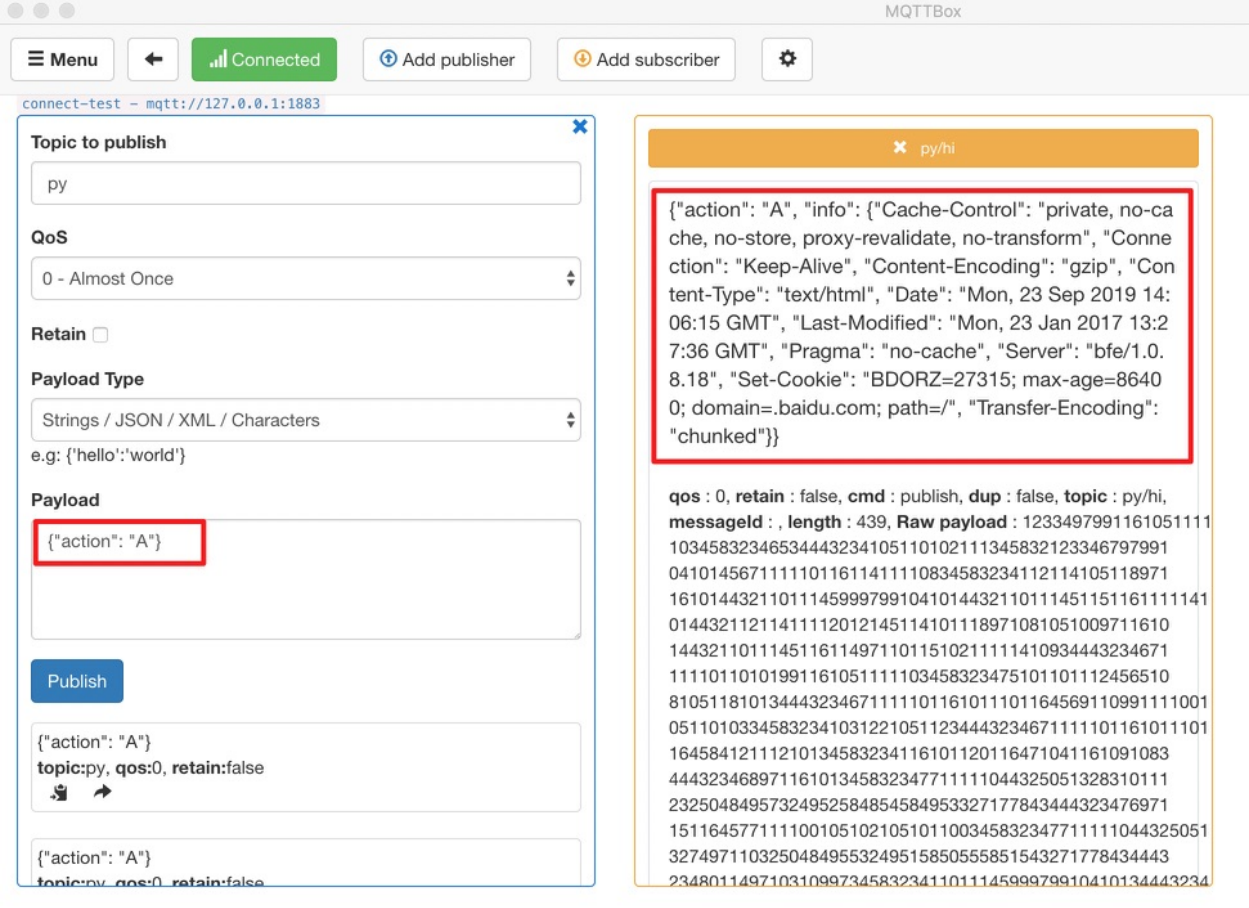
下面，我们编写脚本 get.py 来获取 https://baidu.com 的 headers 信息，假定触发条件为 Python 运行时接收到来自 localhub 服务的 A 指令，具体如下：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import requests

def handler(event, context):
  """
  data: {"action": "A"}
  """
  if 'action' in event:
    if event['action'] == 'A':
      r = requests.get('https://baidu.com')
      if str(r.status_code) == '200':
        event['info'] = dict(r.headers)
      else:
        event['info'] = 'exception found'
    else:       event['info'] = 'action error'
  else:
    event['error'] = 'action not found'

  return event
```

函数运行时服务的配置如下：

```
# python function 配置
functions:
  - name: 'sayhi3'
    handler: 'get.handler'
    codedir: 'var/db/baetyl/function-sayhi'
```

如上，localhub 服务接收到发送到主题 py 的消息后，会调用 get.py 脚本执行具体处理逻辑，然后将执行结果以 MQTT 消息形式反馈给主题 py/hi。这里，我们通过 MQTTBox 订阅主题 py/hi，并向主题 py 发送消息 {"action": "A"}，然后观察 MQTTBox 订阅主题 py/hi 的消息收取情况。如正常，则可正常获取 https://baidu.com 的 headers 信息。



获取 Baetyl 官网 headers 信息

14.2 引用 Pytorch 第三方包

Pytorch 是机器学习中使用广泛的深度学习框架，我们可以引入第三方库 Pytorch 使用它的功能。如何引入，具体如下所示：

- 步骤 1: 进入 Python 脚本目录，然后下载 Pytorch 及其依赖（PIL、caffee2、numpy、six、torchvision）

```
cd /directory/of/Python/script
pip3 download torch torchvision
```

- 步骤 2: 解压 .whl 文件，得到源码包，然后删除 .whl 文件和包描述文件

```
unzip \*.whl
rm -rf *.whl *.dist-info
```

- 步骤 3: 使当前目录成为一个 package

```
touch __init__.py
```

- 步骤 4: 在具体执行脚本中引入第三方库 Pytorch，如下所示：

```
import torch
```

- 步骤 5: 执行脚本

```
python your_script.py
```

如上述操作正常，则形成的脚本目录结构如下图所示。

```
[root@instance-80v5cs0x function-sayhi-code]# tree -L 1
.
├── caffe2
├── calc.py
├── __init__.py
├── numpy
├── PIL
├── six.py
├── torch
└── torchvision

5 directories, 3 files
```

Python

Pytorch 第三方库脚本目录

下面，我们编写脚本 calc.py 来使用 Pytorch 中的函数生成随机张量，假定触发条件为 Python 运行时接收到来自 localhub 服务的 B 指令，具体如下：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import torch

def handler(event, context):
  """
  data: {"action": "B"}
  """
  if 'action' in event:
    if event['action'] == 'B':
      x = torch.rand(5, 3)
```

(下页继续)

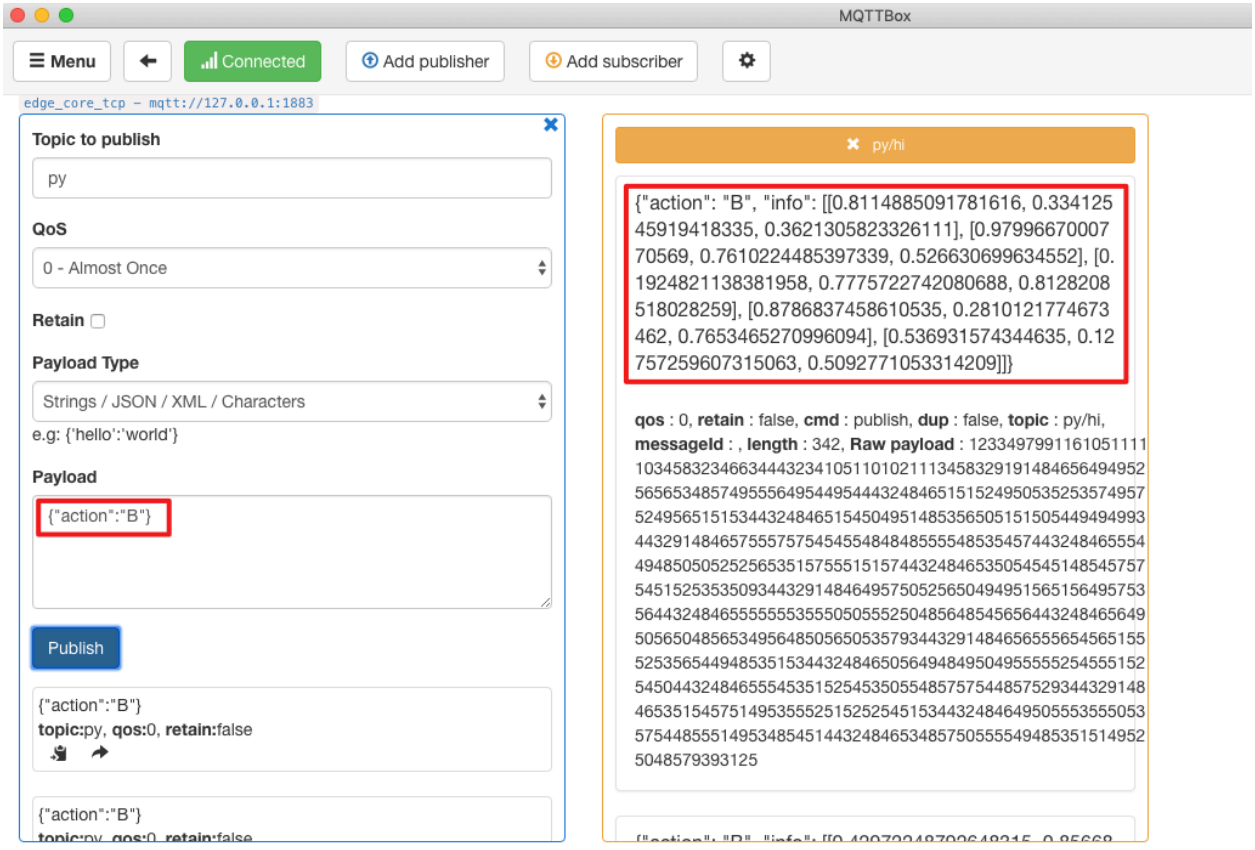
(续上页)

```
      event['info'] = x.tolist()
    else:
      event['info'] = 'exception found'
  else:
    event['error'] = 'action not found'

  return event
```

函数运行时服务的配置如下:

```
# python function 配置
functions:
  - name: 'sayhi3'
    handler: 'calc.handler'
    codedir: 'var/db/baetyl/function-sayhi'
```

如上，localhub 服务接收到发送到主题 py 的消息后，会调用 calc.py 脚本执行具体处理逻辑，然后将执行结果以 MQTT 消息形式反馈给主题 py/hi。这里，我们通过 MQTTBox 订阅主题 py/hi，并向主题 py 发送消息 {"action": "B"}，然后观察 MQTTBox 订阅主题 py/hi 的消息收取情况。如正常，则可正常生成随机张量。



成随机张量

CHAPTER 15

如何针对 Node 运行时引入第三方包

声明:

- 本文测试所用设备系统为 Ubuntu18.04
- 运行模式为 **docker** 容器模式，**native** 进程模式配置流程相同
- Node 版本为 8.5
- 模拟 MQTT client 行为的客户端为 MQTTBox
- 本文选择引入 `Lodash` 这个第三方包来进行演示说明
- 本文中基于 Hub 模块创建的服务名称为 `localhub` 服务。并且针对本文的测试案例中，对应的 `localhub` 服务、函数计算服务以及其他服务的配置统一如下：

```
# localhub 配置
# 配置文件位置: var/db/baetyl/localhub-conf/service.yml
listen:
  - tcp://0.0.0.0:1883
principals:
  - username: 'test'
    password: 'hahaha'
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
```

(下页继续)

(续上页)

```
        permit: ['#']

# 本地 baetyl-function-manager 配置
# 配置文件位置: var/db/baetyl/function-manager-conf/service.yml
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
rules:
  - clientid: localfunc-1
    subscribe:
      topic: node
    function:
      name: sayhi
    publish:
      topic: t/hi
functions:
  - name: sayhi
    service: function-sayhi
    instance:
      min: 0
      max: 10
      idletime: 1m

# application.yml 配置
# 配置文件位置: var/db/baetyl/application.yml
version: v0
services:
  - name: localhub
    image: hub.baidubce.com/baetyl/baetyl-hub
    replica: 1
    ports:
      - 1883:1883
    mounts:
      - name: localhub-conf
        path: etc/baetyl
        readonly: true
      - name: localhub-data
        path: var/db/baetyl/data
      - name: localhub-log
```

(下页继续)

(续上页)

```
        path: var/log/baetyl
  - name: function-manager
    image: hub.baidubce.com/baetyl/baetyl-function-manager
    replica: 1
    mounts:
      - name: function-manager-conf
        path: etc/baetyl
        readonly: true
      - name: function-manager-log
        path: var/log/baetyl
  - name: function-sayhi
    image: hub.baidubce.com/baetyl/baetyl-function-node85
    replica: 0
    mounts:
      - name: function-sayjs-conf
        path: etc/baetyl
        readonly: true
      - name: function-sayjs-code
        path: var/db/baetyl/function-sayhi
        readonly: true
volumes:
  # hub
  - name: localhub-conf
    path: var/db/baetyl/localhub-conf
  - name: localhub-data
    path: var/db/baetyl/localhub-data
  - name: localhub-log
    path: var/db/baetyl/localhub-log
  # function manager
  - name: function-manager-conf
    path: var/db/baetyl/function-manager-conf
  - name: function-manager-log
    path: var/db/baetyl/function-manager-log
  # function node runtime sayhi
  - name: function-sayjs-conf
    path: var/db/baetyl/function-sayjs-conf
  - name: function-sayjs-code
    path: var/db/baetyl/function-sayjs-code
```

系统自带的 Node 环境有可能不会满足我们的需要，实际使用往往需要引入第三方库，下面给出示例。

15.1 引用 Lodash 第三方包

Lodash 是一个一致性、模块化、高性能的 JavaScript 实用工具库。我们可以引入第三方库 Lodash 来使用它的功能。如何引入，具体如下所示：

- 步骤 1: 进入 js 脚本目录，然后下载 Lodash

```
cd /directory/of/Node/script
npm install --save lodash
```

- 步骤 2: 在具体执行脚本中引入第三方库 Lodash，如下所示：

```
const _ = require('lodash');
```

- 步骤 3: 执行脚本

```
node your_script.js
```

如上述操作正常，则形成的脚本目录结构如下图所示。

```
[root@instance-80v5cs0x function-sayjs-code]# tree -L 2
.
├── index.js
└── node_modules
    └── lodash

1 directory, 2 files
```

_ Node

Lodash 第三方库脚本目录

下面，我们编写脚本 index.js 来使用 Lodash 提供的功能，具体如下：

```
#!/usr/bin/env node

const _ = require('lodash');

exports.handler = (event, context, callback) => {
  result = {}

  //筛选数组中重复元素
  result["unique_array"] = _.uniq(event['array']);
  //排序
  result['sorted_users'] = _.sortBy(event['users'], function(o) { return o.age; });
  //过滤
```

(下页继续)

(续上页)

```
  result['filtered_users'] = _.filter(event['users'], function(o) { return !o.active; });

  callback(null, result);
}
```

函数运行时服务的配置如下:

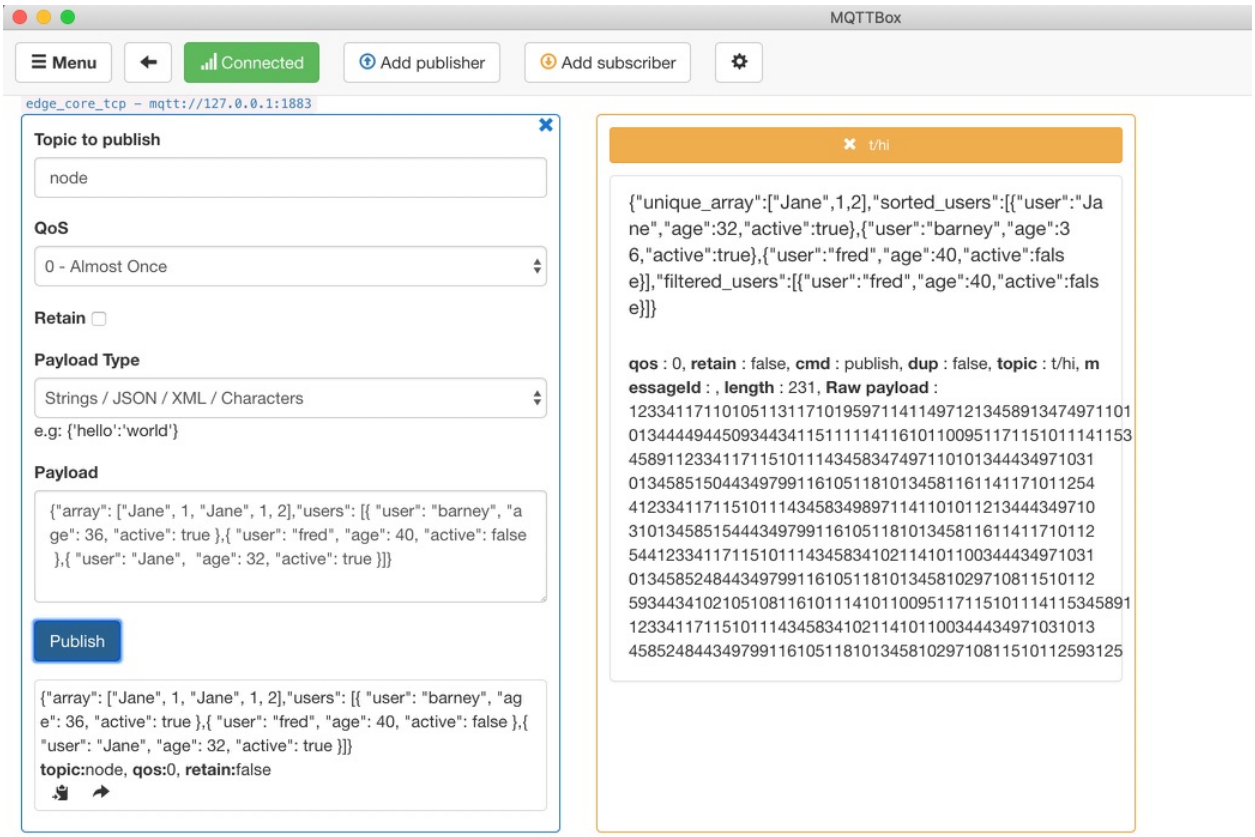
```
# node function 配置
functions:
  - name: 'sayhi'
    handler: 'index.handler'
    codedir: 'var/db/baetyl/function-sayhi'
```

首先定义如下的 json 数据作为输入消息:

```
{
    "array": ["Jane", 1, "Jane", 1, 2],
    "users": [
        { "user": "barney", "age": 36, "active": true  },
        { "user": "fred",   "age": 40, "active": false },
        { "user": "Jane",   "age": 32, "active": true  }
    ]
}
```

如上，`localhub` 服务接收到发送到主题 `node` 的消息后，会调用 `index.js` 脚本执行具体逻辑，对消息中的数组执行重复元素筛选、元素排序、元素按条件过滤等操作。然后将执行结果以 MQTT 消息形式反馈给主题 `t/hi`。我们通过 MQTTBox 订阅主题 `t/hi`，可以观察到如下消息:

```
{
    "unique_array": ["Jane", 1, 2],
    "sorted_users": [
        { "user": "Jane",   "age": 32, "active": true  },
        { 'user': 'barney', "age": 36, "active": true  },
        { "user": "fred",   "age": 40, "active": false }
    ],
    "filtered_users": [
        { "user": "fred",   "age": 40, "active": false }
    ],
}
```


MQTTBox
Menu
Connected
Add publisher
Add subscriber
edge_core_tcp - mqtt://127.0.0.1:1883
Topic to publish
node
QoS
0 - Almost Once
Retain
Payload Type
Strings / JSON / XML / Characters
e.g: {'hello':'world'}
Payload
{"array": ["Jane", 1, "Jane", 1, 2],"users": [{ "user": "barney", "age": 36, "active": true },{ "user": "fred", "age": 40, "active": false },{ "user": "Jane", "age": 32, "active": true }]}
Publish
{"array": ["Jane", 1, "Jane", 1, 2],"users": [{ "user": "barney", "age": 36, "active": true },{ "user": "fred", "age": 40, "active": false },{ "user": "Jane", "age": 32, "active": true }]}
topic:node, qos:0, retain:false
t/hi
{"unique_array":["Jane",1,2],"sorted_users":[{"user":"Jane","age":32,"active":true},{"user":"barney","age":36,"active":true},{"user":"fred","age":40,"active":false}],"filtered_users":[{"user":"fred","age":40,"active":false}]}
qos : 0, retain : false, cmd : publish, dup : false, topic : t/hi, messageId : , length : 231, Raw payload :

lodash

数据处理

CHAPTER 16

如何开发一个自定义函数运行时

函数运行时是函数执行的载体，通过动态加载函数代码来执行函数，跟函数实现的语言强相关。比如 Python 代码需要使用 Python 运行时来调用。这里就涉及到多种语言的问题，为了统一调用接口和协议，我们最终选择了 GRPC，借助其强大的跨语言 IDL 和高性能 RPC 通讯能力，打造可灵活扩展的函数计算框架。

在函数计算服务中，`baetyl-function-manager` 负责函数实例的管理和调用。函数实例由函数运行时服务提供，函数运行时服务只需满足下面介绍的约定。

16.1 协议约定

开发者可直接使用 sdk/baetyl-go 中的 `function.proto` 生成各自编程语言的消息和服务实现，具体定义如下。GRPC 使用方法可参考 GRPC 官网的文档。

```
syntax = "proto3";

package baetyl;

// 函数 Server 定义
service Function {
  rpc Call(FunctionMessage) returns (FunctionMessage) {}
  // rpc Talk(stream Message) returns (stream Message) {}
}
```

(下页继续)

(续上页)

```
// 函数调用和返回的消息体
message FunctionMessage {
  uint64 ID                 = 1;
  uint32 QOS                = 2;
  string Topic              = 3;
  bytes  Payload            = 4;

  string  FunctionName      = 11;
  string  FunctionInvokeID  = 12;
}
```

注意：Docker 容器模式下，函数实例的资源限制不要低于 50M 内存，20 个线程。

16.2 配置约定

函数运行时模块并不强制约定配置，但是为了统一配置方式，推荐如下配置项

- name：函数名称
- handler：函数处理接口
- codedir：函数代码所在路径，如果有的话。

下面是一个 Python 函数运行时服务的配置举例：

```
functions:
  - name: 'sayhi'
    handler: 'sayhi.handler'
    codedir: 'var/db/baetyl/function-sayhi'
```

16.3 启动约定

函数运行时服务同其他服务一样，唯一的区别是实例是由其他服务动态启动的。比如为了避免监听端口冲突，可以动态指定端口。函数运行时模块可以从环境变量中读取 BAETYL_SERVICE_INSTANCE_ADDRESS 作为 GRPC Server 监听的地址。另外，动态启动的函数实例没有权限调用主程序的 API 。最后，模块监听 SIGTERM 信号来实现优雅退出。完整的实现可参考 Python2.7、Python3.6 运行时模块（baetyl-function-python27、baetyl-function-python36）。

CHAPTER 17

如何开发一个自定义功能模块

在开发自定义模块前请阅读*源码安装 Baetyl*，了解 Baetyl 的编译环境。

自定义模块不限定开发语言，可运行即可，基本没有限制，甚至可以直接使用 hub.docker.com 上已有的镜像，比如 eclipse-mosquitto。但是了解下面介绍的约定，有利于更好、更快地开发自定义模块。

17.1 目录约定

目前，进程模式和容器模式一样，会为每个服务开辟独立的工作空间，虽然达不到隔离的效果，但是可以保证用户使用体验的一致。进程模式会在 var/run/baetyl/services 目录下为每个服务创建一个以服务名称命名的目录，服务程序启动时会指定该目录为工作目录，服务绑定的存储卷（volume）会映射（软链）到工作目录下。这里我们沿用容器模式的叫法，把该目录下的空间也称作容器，那么容器中的目录有如下推荐的使用方式：

- 容器中默认工作目录：/
- 容器中默认配置文件：/etc/baetyl/service.yml
- 容器中默认持久化路径：/var/db/baetyl
- 容器中默认日志路径：/var/log/baetyl

注意：如果数据需要持久化在设备（宿主机）上，比如数据库和日志，必须通过存储卷将容器中的目录映射到宿主机目录上，否者服务停止后数据会丢失。

17.2 启动约定

模块启动的方式没有过多要求，推荐从默认文件中加载 YMAL 格式的配置，然后运行模块的业务逻辑，最后监听 SIGTERM 信号来优雅退出。一个简单的 Golang 模块实现可参考 MQTT 远程通讯模块 (baetyl-remote-mqtt)。

17.3 SDK

如果模块使用 Golang 开发，可使用 Baetyl 提供的 SDK，位于该项目的 sdk 目录中，由 Context 提供功能接口。目前，提供的 SDK 能力还比较有限，后续会逐渐加强。

Context 接口列表如下：

```
// 返回服务的系统配置，比如 hub 和 logger
Config() *ServiceConfig
// 加载服务的自定义配置
LoadConfig(interface{}) error
// 通过系统配置创建一个连接 Hub 的 Client，可以指定 Client ID 和订阅的主题信息
NewHubClient(string, []mqtt.TopicInfo) (*mqtt.Dispatcher, error)
// 返回日志接口
Log() logger.Logger
// 检查运行模式
IsNative() bool
// 等待退出，接收 SIGTERM 和 SIGINT 信号
Wait()
// 返回等待退出的 Channel
WaitChan() <-chan os.Signal

// 主程序 RESTful API

// 更新系统服务
UpdateSystem(string, bool) error
// 查看系统状态
InspectSystem() (*Inspect, error)
// 获取一个宿主机的空闲端口
GetAvailablePort() (string, error)
// 报告本实例的状态信息
ReportInstance(stats map[string]interface{}) error
// 启动某个服务的某个实例
StartInstance(serviceName, instanceName string, dynamicConfig map[string]string) error
```

(下页继续)

(续上页)

```
// 停止某个服务的某个实例
StopInstance(serviceName, instanceName string) error
```

下面以简单定时器模块实现为例，介绍 SDK 的用法。

```
package main

import (
        "encoding/json"
        "time"

        "github.com/baetyl/baetyl/protocol/mqtt"
        baetyl "github.com/baetyl/baetyl/sdk/baetyl-go"
)

// 自定义模块的自定义配置，
type config struct {
        Timer struct {
                Interval time.Duration `yaml:"interval" json:"interval" default:"1m"`
        } `yaml:"timer" json:"timer"`
        Publish mqtt.TopicInfo `yaml:"publish" json:"publish" default:"{\"topic\":\
↪"timer\"}"`
}

func main() {
        // 模块在 Baetyl 的 Context 中启动，SDK 的功能均由 Context 提供
        baetyl.Run(func(ctx baetyl.Context) error {
                var cfg config
                // 加载自定义配置
                err := ctx.LoadConfig(&cfg)
                if err != nil {
                        return err
                }
                // 创建连接 Hub 的客户端
                cli, err := ctx.NewHubClient("", nil)
                if err != nil {
                        return err
                }
                // 启动客户端，支持自动重连
                cli.Start(nil)
```

(下页继续)

(续上页)

```
                // 创建定时器
                ticker := time.NewTicker(cfg.Timer.Interval)
                defer ticker.Stop()
                for {
                        select {
                        case t := <-ticker.C:
                                msg := map[string]int64{"time": t.Unix()}
                                pld, _ := json.Marshal(msg)
                                // 定时发送消息到 Hub
                                err := cli.Publish(cfg.Publish, pld)
                                if err != nil {
                                        // 打印错误日志
                                        ctx.Log().Errorf(err.Error())
                                }
                        case <-ctx.WaitChan():
                                // 等待退出信号，SIGTERM 或者 SIGINT
                                return nil
                        }
                }
        })
}
```

baetyl-timer 的配置中，hub 属于系统配置，timer 和 publish 是该模块的自定义配置。

```
hub:
  address: tcp://localhub:1883
  username: test
  password: hahaha
  clientid: timer1
timer:
  interval: 1s
publish:
  topic: timer1
```

CHAPTER 18

通过 Baetyl 将数据脱敏后存储百度云 TSDB

声明:

- 本文测试所用设备系统为 Ubuntu18.04
- 模拟 MQTT client 向百度云 IoTHub 订阅消息的客户端为 MQTT.fx
- 模拟 MQTT client 向本地 Hub 服务发送消息的客户端为 MQTTBox
- 本文所应用的各服务与本地 Hub 服务间通信认证强制使用 TLS/SSL 安全证书

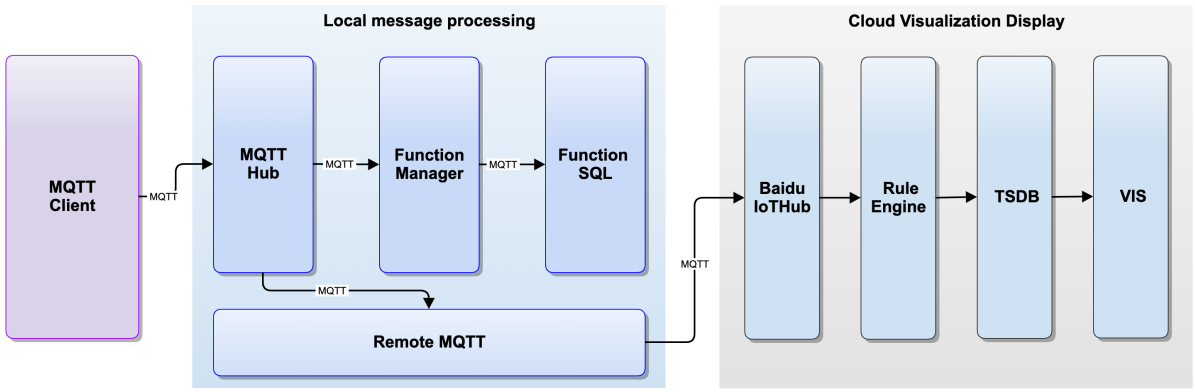
18.1 测试前准备

实际应用场景中，我们需要把设备产生的数据在本地进行 **脱敏**处理后上云展示。

本文则以某实际生产车间中的温度传感器为例，应用 Baetyl，并联合 百度云天工 相关产品服务一起将温度传感器采集到的数据进行 **脱敏**处理（如去除车间编号、设备型号、设备 ID 等信息），然后将 **脱敏**后的数据上传至远程云端进行可视化展示。

其数据流经的路径/服务为:

> **MQTT.fx -> Baetyl Local Hub -> Baetyl Function SQL Runtime -> Baetyl Local Hub -> Baetyl MQTT Remote Module -> Baidu IoT Hub -> Baidu IoT Rule Engine -> Baidu IoT TSDB -> Baidu IoT Visualization**

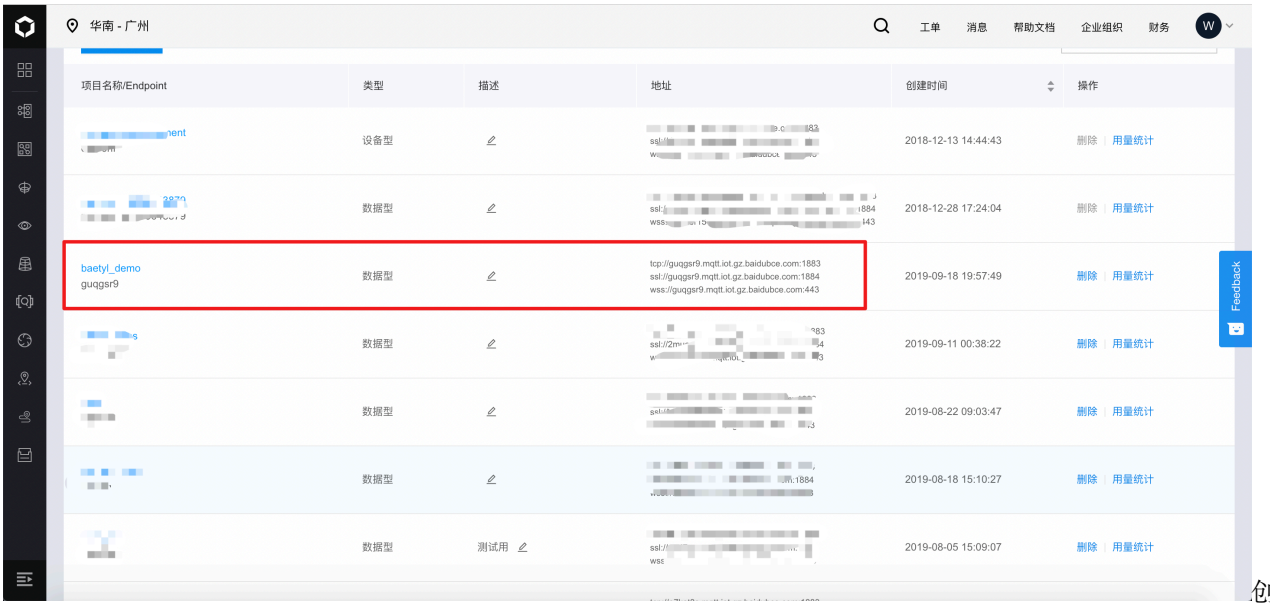


本文 case 流程示意图

因此，在正式开始测试之前，我们需要在云端先把 IoT Hub、Rule Engine、TSDB 及 Visualization 等相关配置完善。

18.1.1 创建物接入 Endpoint

相关创建过程可参考 快速创建物接入 Endpoint（包括创建用户、身份、策略及主题权限信息等），这里仅给出创建完成后的结果示意图。



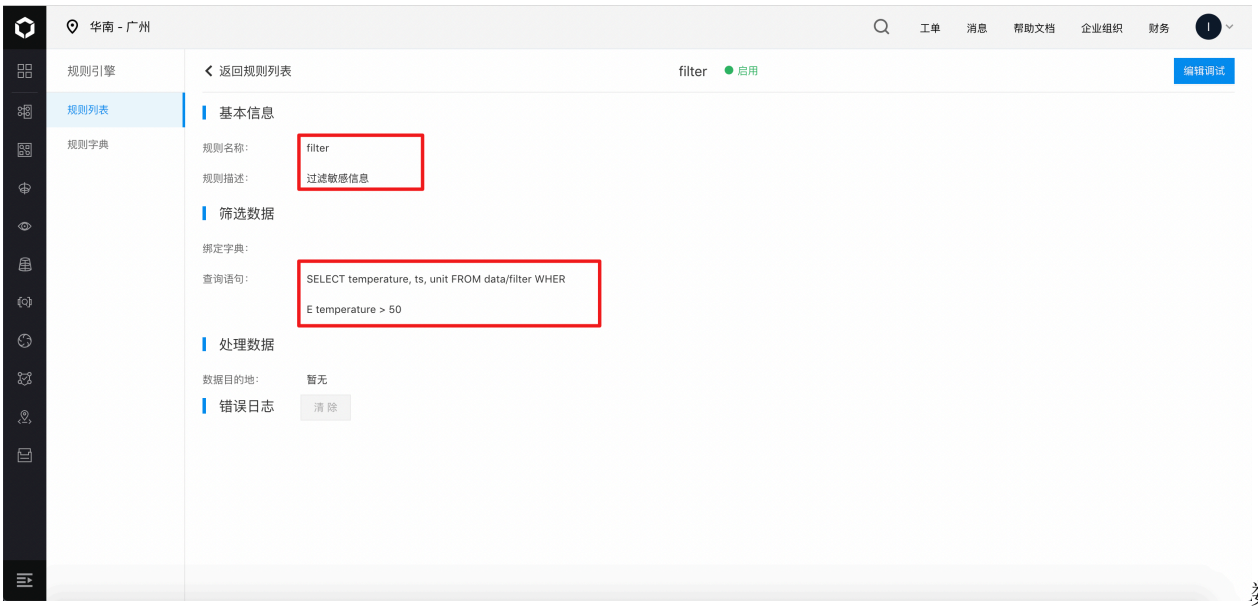
创建物接入 Endpoint

如上，已创建好一个名为 `baetyl_demo` 的物接入项目。其用户名为 `guqgsr9/test`，身份信息为 principal，认证方式为证书认证，策略为 policy，对主题 **data/filter** 有发布和订阅消息的权限（详见下文测试时 MQTT Remote 远程服务模块配置）。

18.2 创建规则引擎 Rule

相关创建过程可参考 快速创建规则引擎 Rule（包括转换 SQL 语句编写、约束条件设置、数据目的地指定等）。这里需要创建两条规则，其一是用于对本地设备产生的原始数据进行过滤；其二是实时提取从物接入既定主题接收的数据消息，并将其转换为 TSDB 能够接收的数据内容，然后将之传送给 TSDB。创建完成后的结果示意图具体如下：

设备生产数据过滤用规则：



数据过滤、脱敏规则

如上，已创建好一个名为 filter 的规则，该规则用于对本地设备实际产生的原始数据进行 **过滤**、**脱敏**等处理。图中所示为筛选实际生产数据中的 **temperature**、**ts**、**unit** 等字段，且满足 **temperature** $>$ **50** 信息，然后将之上传至远程云端 Hub 模块。

提示：这里不需要为规则设置数据目的地，仅作为云端 SQL 语句测试使用。

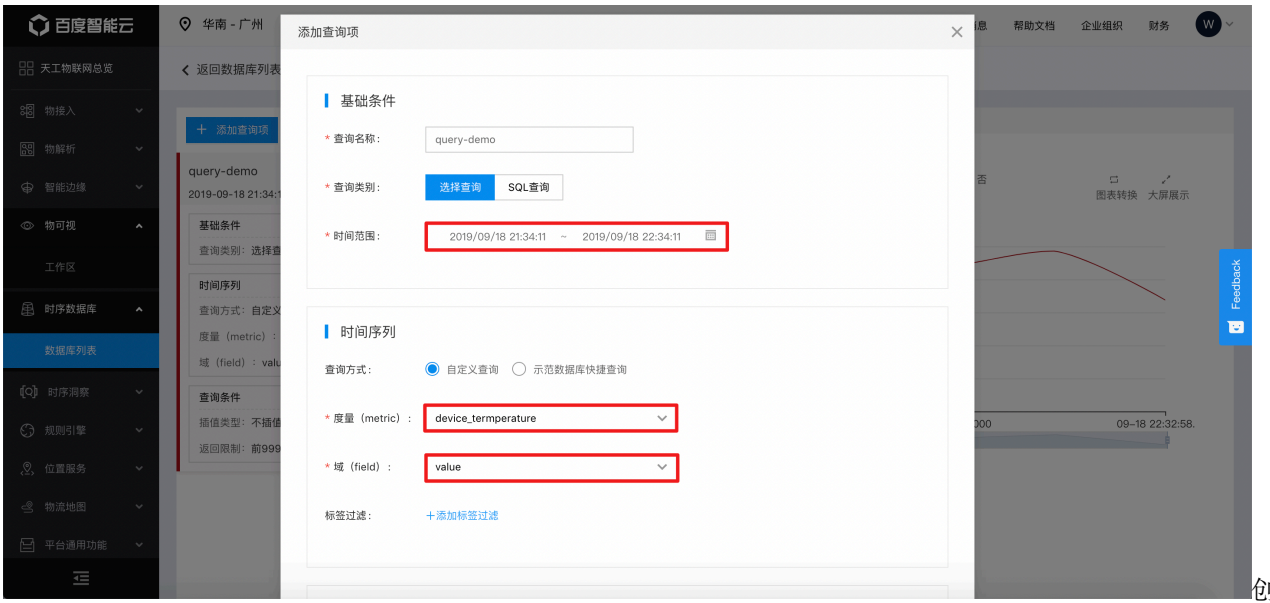
物接入既定主题接收消息用规则：

数据解析、转换 TSDB 规则

如上，已创建好一个名为 `baetyl-demo` 的规则，该规则会默认从物接入 Endpoint 的 **data/filter** 主题提取消息，然后通过 SQL 语句进行转换，将其转换为符合 TSDB 规范 的数据，并将之存储在名为 **baetyl** 的 TSDB 数据库中。

18.3 创建 TSDB 数据库

相关创建过程可参考 快速创建 TSDB（包括查询类别、时间范围、时间序列度量等），这里仅给出创建完成后的结果示意图。

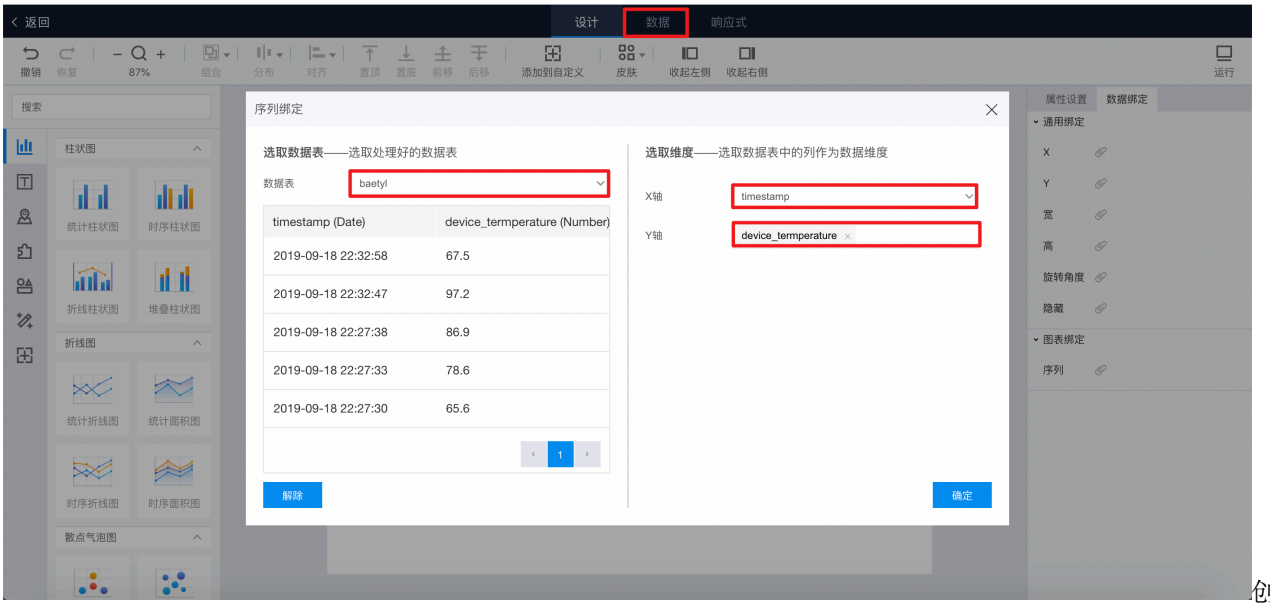


创建 TSDB

如上，已创建好一个名为 baetyl 的 TSDB 时序数据库，该数据库会默认查询 **选定时间段**的符合时间序列度量标识的时序数据信息，且默认显示前 1000 个符合上述条件的值。

18.4 创建物可视展示板

相关创建过程可参考 快速创建物可视（包括设置来源数据表、时间序列度量信息等），这里仅给出创建完成后的结果示意图。



创建物可视

如上，已创建一个名为 `baetyl_demo` 的物可视展示板，其展示数据来源于时序数据库 **baetyl**，时间序列度量信息为 **device_temperature**，X 轴为时间戳，Y 轴为设备温度值。

至此，正式测试前云端相关服务的创建、设置工作已经完成。

提示：*以上创建的物接入 Endpoint、规则引擎 Rule、TSDB 数据库及物可视展示板的所属区域应为同一个(如均为北京，或是均为广州)。*

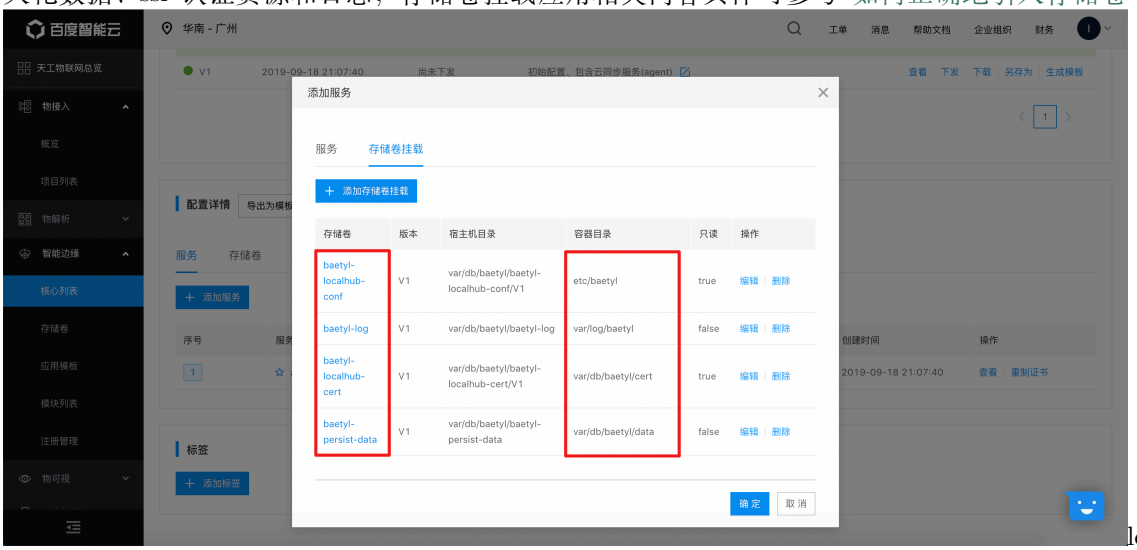
18.5 基本步骤流程

将生产设备数据经 **脱敏**后上云、写入 TSDB 及在云端物可视进行展示所涉及的流程步骤主要包括：

- 步骤 1: **创建核心并下载配置 (含主程序)** 在 BIE 云端管理套件页面选定区域（北京，或广州）创建核心，完善核心创建所需配置信息，点击“下载配置”，然后选择包含主程序，具体请参考BIE 快速入门
- 步骤 2: **本地启动 Baetyl** 本地解压缩主程序（含配置）后，启动 Baetyl，然后点击核心连接状态按钮，如 Baetyl 正常启动，即可看到核心连接状态已变更为 **已连接**
  - Baetyl 启动参考命令：

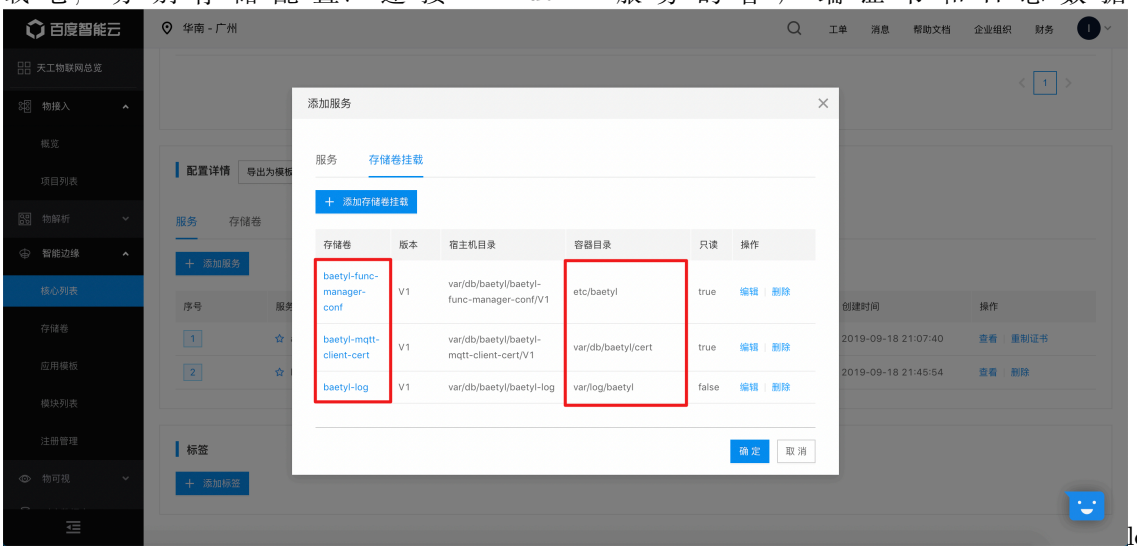
    * `mkdir baetyl-demo`
    * `cd baetyl-demo && unzip -d . baetyl-xxx.zip`
    * `sudo chmod +x bin/baetyl`
    * `sudo bin/baetyl start`

- 步骤 3: **建立服务配置**进入已创建的核心，然后开始依次创建本次测试所需的服务配置信息（Hub 服务配置、Function Manager 服务配置、Function Filter 服务配置、Remote 服务配置），详细内容可参考BIE 操作实践
  - Hub 服务配置：需要挂载 conf、data、cert、log 四个挂载卷，分别存储 Hub 服务的配置、持久化数据、ssl 认证资源和日志，存储卷挂载应用相关内容具体可参考 如何正确地引入存储卷



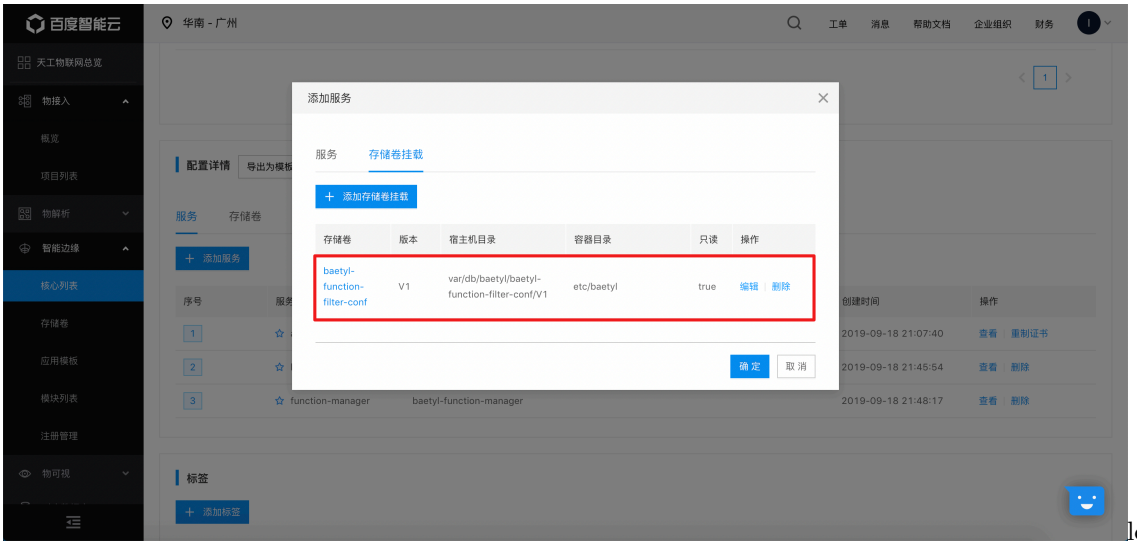
localhub_volume

  - Function Manager 服务配置：需要挂载 conf、cert、log 三个挂载卷，分别存储配置、连接 Hub 服务的客户端证书和日志数据



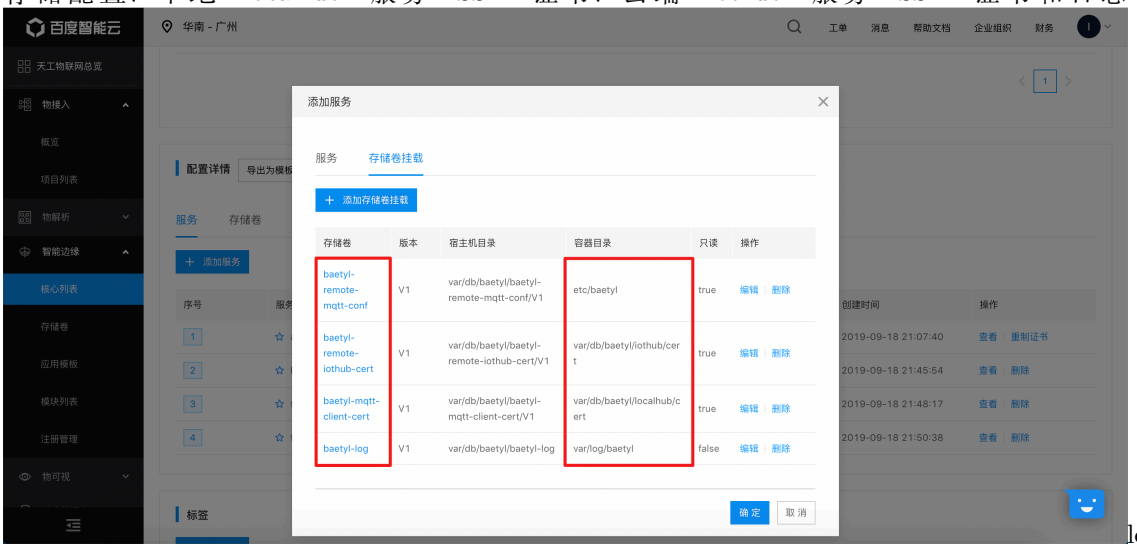
localhubfunction

  - Function Filter 服务配置：需要挂载 conf 挂载卷，存储配置信息



localhubfunction

– Remote 服务配置：需要挂载 conf、localhub-cert、iothub-cert、log 四个挂载卷，存储配置、本地 localhub 服务 SSL 证书、云端 iothub 服务 SSL 证书和日志



localhubremote_

- 步骤 4: **发布及下发服务配置**完成核心所需的各个服务的配置后，点击“生成配置”按钮生成当前版本配置，然后再点击“下发配置”按钮将生成的版本配置下发至本地，Baetyl 服务会自动切换、加载该下发的新配置信息，具体可参考BIE 操作实践

  – 此过程要求 Baetyl 持续 **保持连接**状态，如果 Baetyl 在下发配置前已断开连接，则重新启动 Baetyl，在连接状态恢复至 **已连接**后下发新配置即可（推荐）；或可选择 **下载配置**按钮，将该新配置下载至本地，然后自行在本地替换，然后再启动 Baetyl

- 步骤 5: **配置 MQTTBox 连接信息**启动 MQTTBox，配置其与本地 Hub 服务建立连接所需的各配置信息

- 步骤 6: **发送测试数据**在 MQTTBox 与本地 Hub 模块建立连接后，向主题 **data** 发送测试数据，然后打开 TSDB 面板，查看是否有数据成功写入，同时打开物可视展示板，观察数据写入的状态

- 步骤 7: **结果验证**若上述过程顺利，则可以看到刚才已发送的测试已经成功写入 TSDB，并在物可视进

行展示。

18.6 测试与验证

本节中将会结合 智能边缘 BIE 云端管理套件从云端创建 Baetyl 执行所需的一切配置信息，然后由智能边缘 BIE 云端管理套件下发本地部署，最后由本地启动 Baetyl，完成整个测试例的测试与验证。

18.6.1 Baetyl 主程序配置

```
version: V2
services:
  - name: agent
    image: 'hub.baidubce.com/baetyl/baetyl-agent:latest'
    replica: 1
    mounts:
      - name: agent-conf
        path: etc/baetyl
        readonly: true
      - name: agent-cert
        path: var/db/baetyl/cert
        readonly: true
      - name: agent-volumes
        path: var/db/baetyl/volumes
      - name: agent-log
        path: var/log/baetyl
  - name: localhub
    image: 'hub.baidubce.com/baetyl/baetyl-hub:latest'
    replica: 1
    ports:
      - '1883:1883'
      - '8883:8883'
    mounts:
      - name: baetyl-localhub-conf
        path: etc/baetyl
        readonly: true
      - name: baetyl-log
        path: var/log/baetyl
      - name: baetyl-localhub-cert
        path: var/db/baetyl/cert
```

(下页继续)

(续上页)

```
        readonly: true
      - name: baetyl-persist-data
        path: var/db/baetyl/data
  - name: function-manager
    image: hub.baidubce.com/baetyl/baetyl-function-manager:latest
    replica: 1
    mounts:
      - name: baetyl-func-manager-conf
        path: etc/baetyl
        readonly: true
      - name: baetyl-mqtt-client-cert
        path: var/db/baetyl/cert
        readonly: true
      - name: baetyl-log
        path: var/log/baetyl
  - name: function-filter
    image: 'hub.baidubce.com/baetyl/baetyl-function-sql:latest'
    replica: 0
    mounts:
      - name: baetyl-function-filter-conf
        path: etc/baetyl
        readonly: true
  - name: remote-mqtt
    image: 'hub.baidubce.com/baetyl/baetyl-remote-mqtt:latest'
    replica: 1
    mounts:
      - name: baetyl-remote-mqtt-conf
        path: etc/baetyl
        readonly: true
      - name: baetyl-remote-iothub-cert
        path: var/db/baetyl/iothub/cert
        readonly: true
      - name: baetyl-mqtt-client-cert
        path: var/db/baetyl/localhub/cert
        readonly: true
      - name: baetyl-log
        path: var/log/baetyl
volumes:
  - name: agent-conf
    path: var/db/baetyl/agent-conf
```

(下页继续)

(续上页)

```
  - name: agent-cert
    path: var/db/baetyl/agent-cert
  - name: agent-volumes
    path: var/db/baetyl
  - name: agent-log
    path: var/db/baetyl/agent-log
  - name: baetyl-log
    path: var/db/baetyl/baetyl-log
  - name: baetyl-localhub-cert
    path: var/db/baetyl/baetyl-localhub-cert
  - name: baetyl-persist-data
    path: var/db/baetyl/baetyl-persist-data
  - name: baetyl-mqtt-client-cert
    path: var/db/baetyl/baetyl-mqtt-client-cert
  - name: baetyl-function-filter-conf
    path: var/db/baetyl/baetyl-function-filter-conf
  - name: baetyl-localhub-conf
    path: var/db/baetyl/baetyl-localhub-conf
  - name: baetyl-remote-iothub-cert
    path: var/db/baetyl/baetyl-remote-iothub-cert
  - name: baetyl-remote-mqtt-conf
    path: var/db/baetyl/baetyl-remote-mqtt-conf
  - name: baetyl-func-manager-conf
    path: var/db/baetyl/baetyl-func-manager-conf
```

18.6.2 Hub 服务配置

```
listen:
  - tcp://0.0.0.0:1883
  - ssl://0.0.0.0:8883
certificate:
  ca: var/db/baetyl/cert/ca.pem
  cert: var/db/baetyl/cert/server.pem
  key: var/db/baetyl/cert/server.key
principals:
  - username: two-way-tls
    permissions:
      - action: 'pub'
        permit: ['#']
```

(下页继续)

(续上页)

```
      - action: 'sub'
        permit: ['#']
  - username: test
    password: hahaha
    permissions:
      - action: 'pub'
        permit: ['#']
      - action: 'sub'
        permit: ['#']
logger:
  path: var/log/baetyl/localhub-service.log
  level: "debug"
```

18.6.3 Function Manager 服务配置

```
hub:
  address: ssl://localhub:8883
  username: two-way-tls
  ca: var/db/baetyl/cert/ca.pem
  cert: var/db/baetyl/cert/client.pem
  key: var/db/baetyl/cert/client.key
  insecure: true
rules:
  - clientid: localfunc-1
    subscribe:
      topic: data
      qos: 1
    function:
      name: filter
    publish:
      topic: data/filter
      qos: 1
functions:
  - name: filter
    service: function-filter
    instance:
      max: 10
logger:
  path: var/log/baetyl/func-service.log
```

(下页继续)

(续上页)

```
  level: "debug"
```

18.6.4 Function Filter 服务配置

```
functions:
  - name: filter
    handler: 'SELECT temperature, ts, unit WHERE temperature > 50'
```

如上配置，发送到主题 **data** 的消息会被 SQL 运行时进行处理（脱敏、过滤），然后将处理结果反馈给主题 **data/filter**。

18.6.5 Remote 服务配置

```
hub:
  address: ssl://localhub:8883
  username: two-way-tls
  ca: var/db/baetyl/localhub/cert/ca.pem
  cert: var/db/baetyl/localhub/cert/client.pem
  key: var/db/baetyl/localhub/cert/client.key
  insecure: true
remotes:
  - name: iothub
    address: '<iothub_endpoint>' # 从物接入的项目列表中复制 ssl 地址替换 <iothub_endpoint>,
比如：ssl://xxxxxx.mqtt.iot.gz.baidubce.com:1884, xxxxxx 为 endpoint
    clientid: remote-iothub-1
    username: '<username>' # 从上面选定 (address) 的物接入项目下创建的用户名列表中复制支持␣
↪ssl 连接的用户名替换 <username>, 比如：xxxxxx/test, xxxxxx 为 endpoint
    ca: var/db/baetyl/iothub/cert/ca.pem
    cert: var/db/baetyl/iothub/cert/client.pem
    key: var/db/baetyl/iothub/cert/client.key
rules:
  - hub:
      subscriptions:
        - topic: data/filter
          qos: 0
    remote:
      name: iothub
      subscriptions:
```

(下页继续)

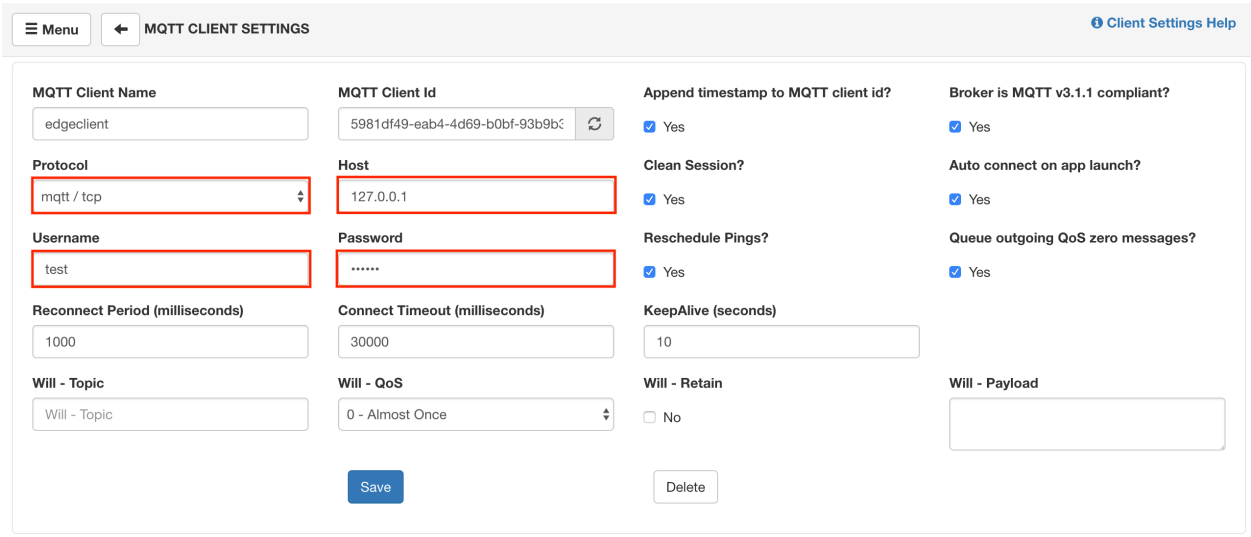
(续上页)

```
logger:
  path: var/log/baetyl/remote-service.log
  level: "debug"
```

如上配置，本地 Hub 服务会将主题 **data/filter** 的消息发送给 Remote 远程服务（上文创建物接入 Endpoint 已拥有主题 **data/filter** 的订阅权限），然后远程 Hub 服务（这里指百度云 IoTHub 平台）接收到主题 **data/filter** 的消息触发规则 **baetyl-demo**（上文已创建），然后由规则引擎对消息进行封装（以满足 TSDB 规范），传送给 TSDB，最终在物可视进行可视化展示。

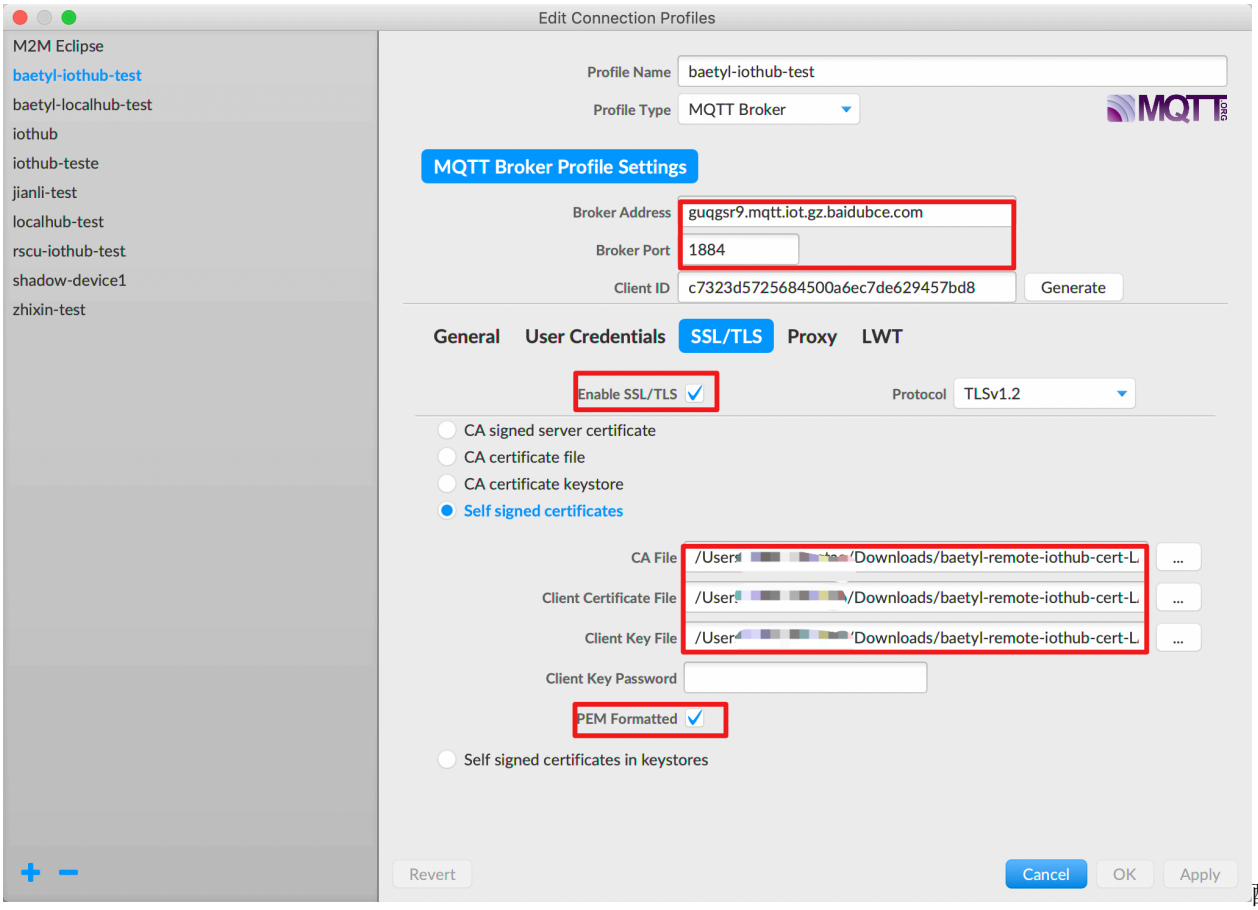
18.6.6 测试

依据 Hub 服务配置对 MQTTBox 进行连接设置，具体如下图示。



配置 MQTTBox 连接信息

同理，依据云端物接入的配置信息，对 MQTT.fx 进行连接配置，具体如下图示。



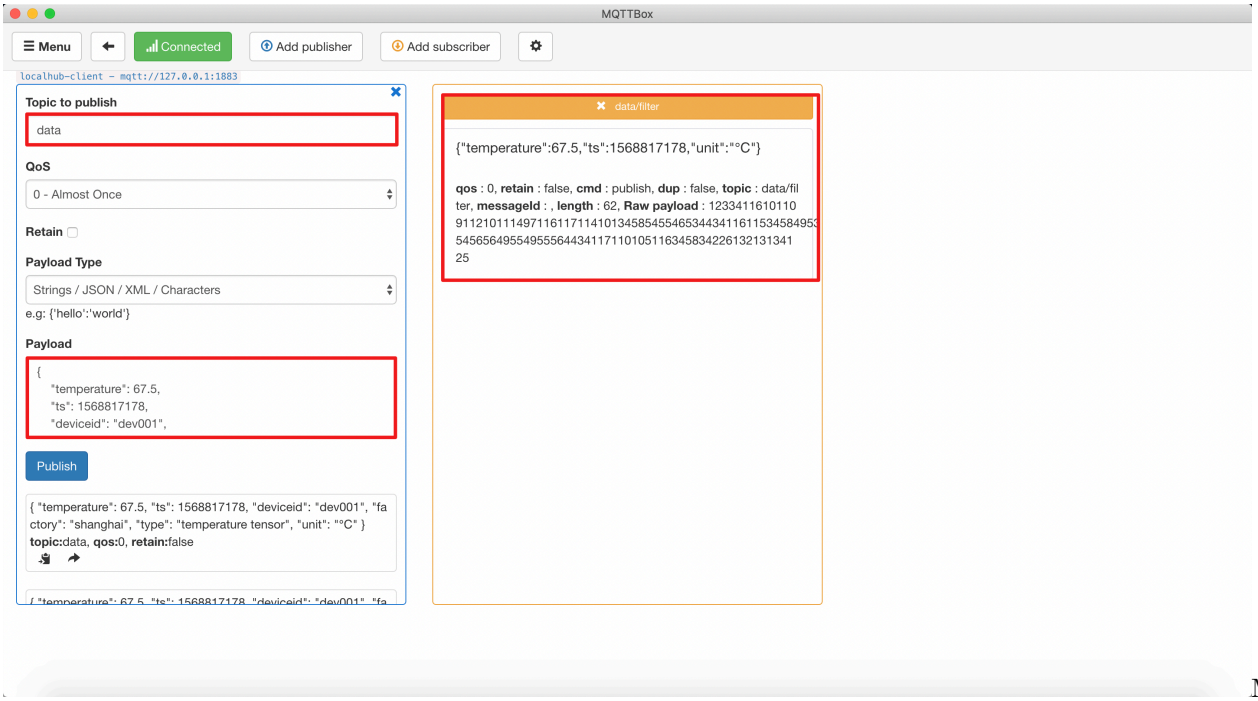
配置 MQTTBox 连接信息

然后通过 MQTTBox 向主题 **data** 发送消息，消息内容格式参考：

```
{
    "temperature": 67.5, // 温度
    "ts": 1568817178, // Unix 时间戳
    "deviceid": "dev001", // 设备 ID
    "factory": "shanghai", // 生产地址
    "type": "temperature tensor", // 设备类型
    "unit": "℃" // 单位
}
```

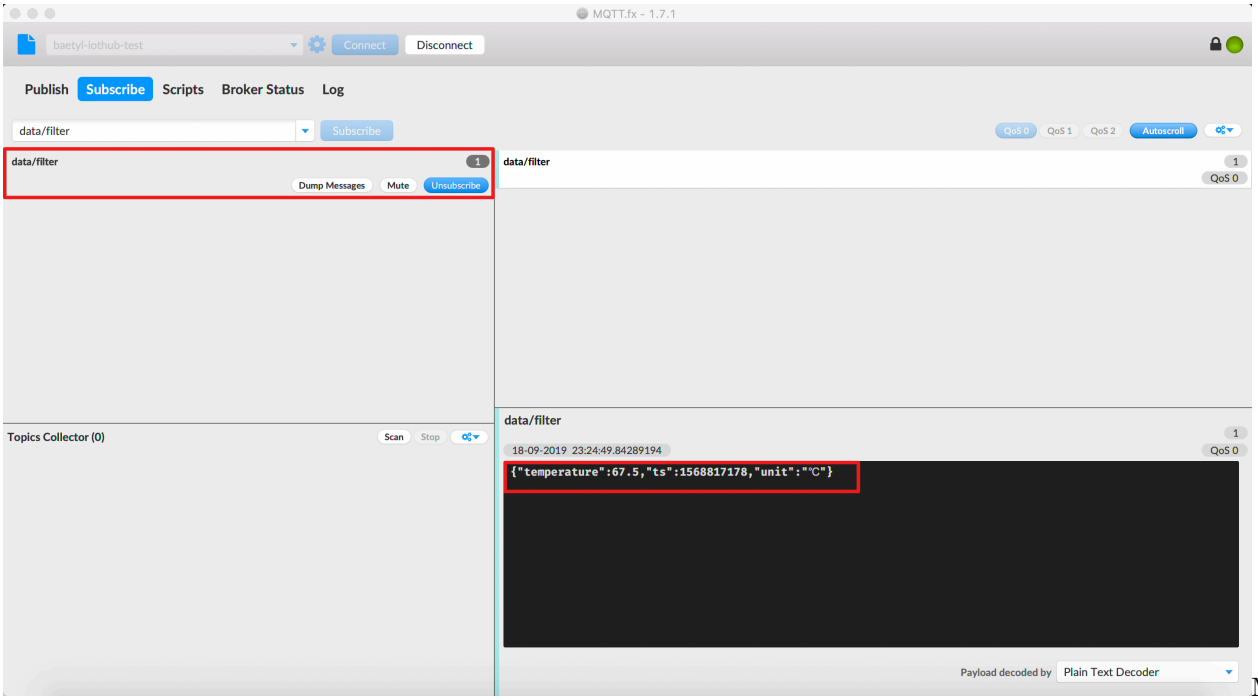
如按上文的消息处理逻辑，该条消息会被筛选出来，并回传给本地 Hub 服务，再由本地 Hub 服务将数据发送给 Remote 服务，最后上传至云端物接入，经由规则 **baetyl-demo** 封装处理，传送给 TSDB，最终在物可视展示。相关示意图如下示。

MQTTBox 收到处理后的消息，表示消息已被 Function Filter 服务处理，并将结果回传给了 Hub 服务。



MQTTBox 收到处理后的消息

MQTT.fx 收到云端物接入的消息，表示该消息已通过 Remote 服务发往了云端物接入



MQTTFX 收到处理后的消息

如果我们通过 MQTTBox 向主题 **data** 发送的消息内容为：

```
{
```

（下页继续）

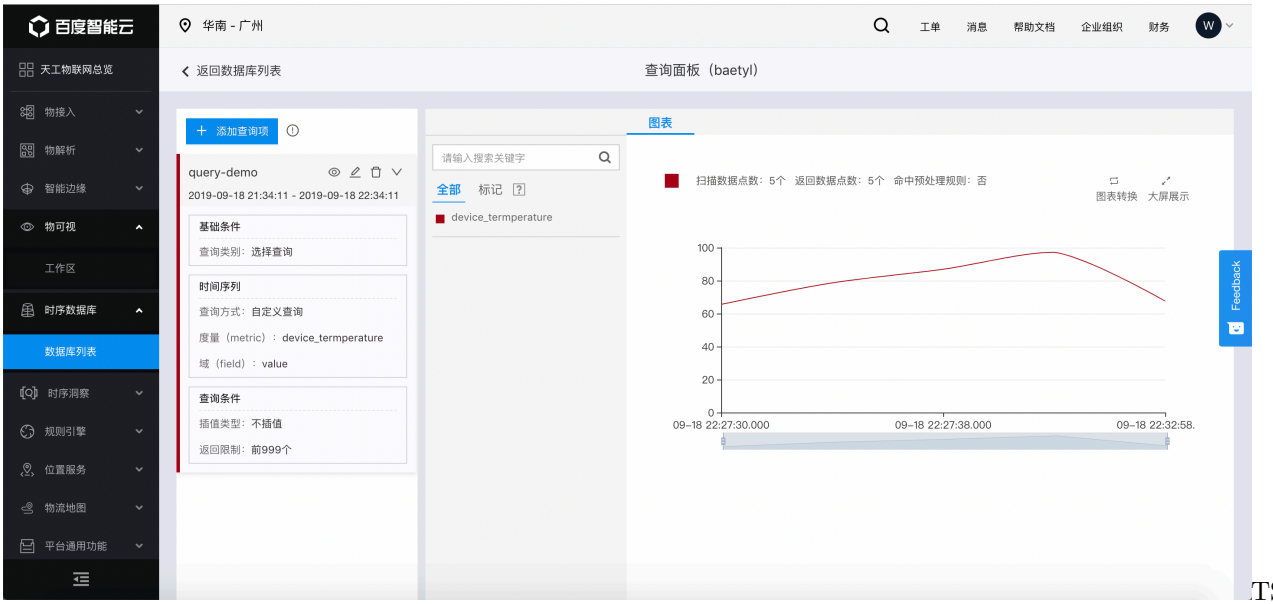
(续上页)

```
    "temperature": 35.6, // 温度
    "ts": 1568817182, // Unix 时间戳
    "deviceid": "dev001", // 设备 ID
    "factory": "shanghai", // 生产地址
    "type": "temperature tensor", // 设备类型
    "unit": "℃" // 单位
}
```

则 MQTTBox 和 MQTT.fx 均不会收到处理后的消息（`temperature < 50` 被过滤掉）。同理，规则引擎 **baetyl-demo**、TSDB 和物可视均不会收到该处理后的消息。

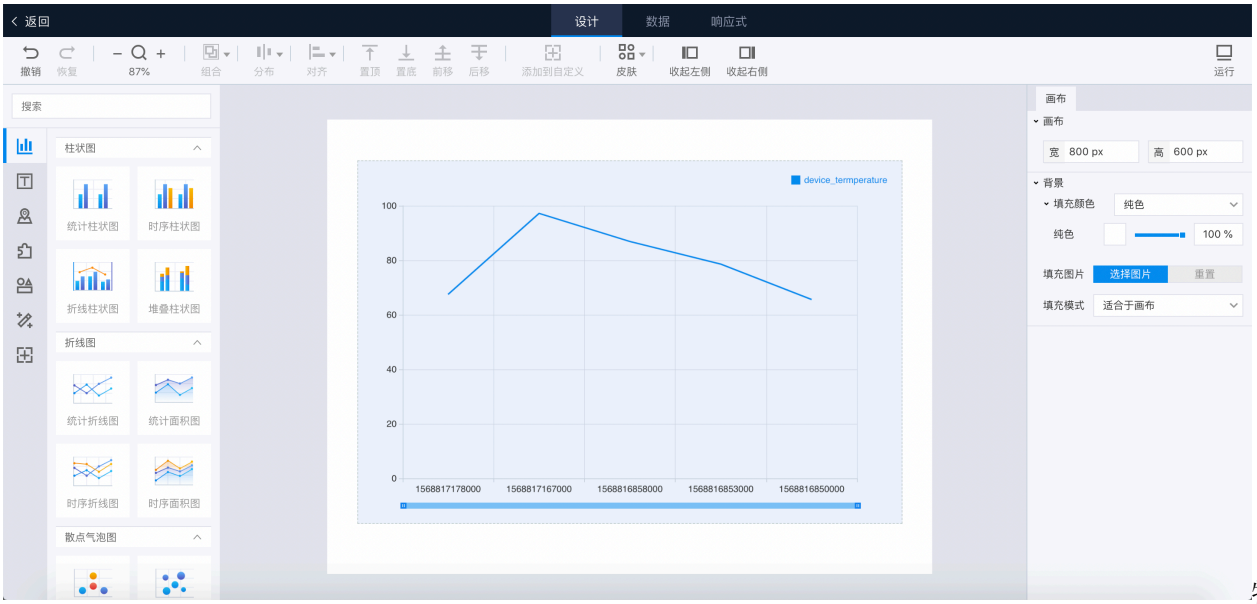
为更清晰地在云端展示处理后的结果，我们写入多条符合要求的数据，得到对应的 TSDB 和物可视的展示效果如下图示。

TSDB 收到多条处理结果



TSDB 收到多条处理结果

物可视收到多条处理结果



物可视收到多条处理结果

至此，通过 Baetyl 已实现将数据写入百度云 TSDB 及通过物可视进行可视化展示。

CHAPTER 19

FAQ

本文主要提供 Baetyl 在各系统、平台部署、启动的相关问题及解决方案。

问题 1: 在以容器模式启动 Baetyl 时，提示缺少启动依赖配置文件

图片

参考方案: 如上图所示，Baetyl 启动缺少配置依赖文件，参考 GitHub-Baetyl example 文件夹补充相应配置文件即可（位于 `etc/baetyl/conf.yml`）。

问题 2: Ubuntu/Debian 下输入命令 `docker info` 后提示:" WARNING: No swap limit support"

参考方案:

1. 修改 `/etc/default/grub` 文件，在其中，编辑或者添加 `GRUB_CMDLINE_LINUX` 为如下内容:

   GRUB_CMDLINE_LINUX=" cgroup_enable=memory swapaccount=1"

2. 保存后执行命令 `sudo update-grub`，完成后重启系统生效。

注意: 如果执行第 2 步时提示出错，可能是 `grub` 设置有误，请检查后重复步骤 1 和步骤 2。

问题 3: 启动 Baetyl 服务提示:"WARNING: Your kernel does not support swap limit capabilities. Limitation discarded"

参考方案: 参考问题 2。

问题 4: 启动 Baetyl 服务提示: " Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get

http://%2Fvar%2Frun%2Fdocker.sock/v1.38/images/json: dial unix /var/run/docker.sock: connect: permission denied"

参考方案:

1. 提供管理员权限
2. 通过以下命令添加当前用户到 docker 用户组

```
sudo usermod -aG docker ${USER}
su - ${USER}
```

如提示没有 `docker group`，使用如下命令创建新 docker 用户组后再执行上述命令:

```
sudo groupadd docker
```

问题 5: 启动 Baetyl 服务出现: "Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?"

参考方案: 按照问题 4 解决方案执行后如仍报出此问题，重新启动 docker 服务即可。

例，CentOS 下启动命令:

```
systemctl start docker
```

问题 6: 启动 Baetyl 服务提示: " failed to create master: Error response from daemon: client version 1.39 is too new. Maximum supported API version is 1.38"

参考方案:

- 如果 Baetyl 的版本大于等于 1.0.0，可在 Baetyl 的配置文件（`etc/baetyl/conf.yml`）中添加如下内容:

```
docker:
  api_version: 1.38
```

- 如果 Baetyl 的版本小于 1.0.0，则需要配置设备的环境变量 `DOCKER_API_VERSION=1.38`，比如:

```
sudo vim ~/.bash_profile
export DOCKER_API_VERSION=1.38
source ~/.bash_profile
```

另外，如果执行完上述命令之后发现服务还是不能启动，则是由于环境变量在当前用户并未生效导致。可以参考的解决办法如下:

```
sudo vim /etc/sudoers # 如果系统没有安装 vim，可以先安装或是采用 vi，gedit 等其他编辑器
```

然后，将 `Defaults env_reset` 修改为 `Defaults !env_reset` 保存、退出即可。

注意：以上修改的仅仅是当前用户的环境变量规则，也就是说，在当前用户下使用 sudo 不会更改环境变量，但是 sudo -s 后进入的是 root 用户，再提权即 root 用户使用 sudo，那么环境变量依然遵循 root 用户的配置。相关原理可以参考 系统中 sudo 的作用。如果想要 root 后依旧保留当前环境配置，需要额外设置，可以参考 如何对所有用户设置环境变量。

问题 7: Baetyl 如何使用 NB-IoT 连接百度云管理套件或者物接入？

参考方案：NB-IoT 是一种网络制式，和 2/3/4G 类似，带宽窄功耗低。NB-IoT 支持基于 TCP 的 MQTT 通信协议，因此可以使用 NB-IoT 卡连接百度云物接入，部署 Baetyl 应用和 BIE 云管理通信。但国内三大运营商中，电信对他们的 NB 卡做了限制，仅允许连接电信的云服务 IP，所以目前只能使用移动 NB 卡和联通 NB 卡连接百度云服务。

问题 8: Baetyl 支持数据计算后将计算结果推送 Kafka 吗？

参考方案：支持，您可以参考*如何针对 Python 运行时编写 Python 脚本* 一文，向 Hub 订阅消息，并将消息逐个写入 Kafka。您也可以参考*如何开发一个 Baetyl 自定义模块*，该模块用于向 Hub 订阅消息，然后批量写入 Kafka。

问题 9: Baetyl 配置更改的方式有哪些？只能通过 云端管理套件 进行配置更改吗？

参考方案：目前，我们推荐通过云端管理套件进行配置定义和下发，但您也可以手动更改核心设备上的配置文件，然后重启 Baetyl 服务使之生效。

问题 10：我下载了 Linux MQTTBox 客户端，解压缩后将可执行文件放置到了 /usr/local/bin 目录（其他系统启动加载目录相同，如 /usr/bin，/bin，/usr/sbin 等），启动时候提示:" error while loading shared libraries: libgconf-2.so.4: cannot open shared object file: No such file or directory"

参考方案：这是由于 MQTTBox 启动缺少 libgconf-2.so.4 库所致。推荐做法如下：

- 步骤 1：下载并解压缩 MQTTBox 软件包；
- 步骤 2：进入 MQTTBox 软件包解压缩后的目录，为 MQTTBox 可执行文件配置执行权限；
- 步骤 3：为 MQTTBox 设置软连接：sudo ln -s /path/to/MQTTBox /usr/local/bin/MQTTBox；
- 步骤 4：进入终端，执行 MQTTBox 即可。

问题 11：在使用智能边缘 BIE 下发配置后，本地启动 Baetyl 服务，利用函数计算模块进行消息处理，发现 localfunc 无法进行消息处理，查看 funclog 有如下报错信息：

> level=error msg=" failed to create new client" dispatcher=mqtt error=" dial tcp 0.0.0.0:1883:connect:connection refused"

参考方案：如果是使用智能边缘 BIE 云端管理套件下发配置，有如下几个点需要注意：

- 云端下发配置目前只支持容器模式
- 如果是云端下发配置，localfunc 里配置的 hub 地址应为 localhub 而非 0.0.0.0

根据以上信息结合实际报错进行判断，根据需要重新从云端进行配置下发，或者参考*配置解析文档* 进行核对及配置。

问题 12：在使用智能边缘 BIE 云端管理套件时，如何选取 CFC 函数?

参考方案:

- 确保您的智能边缘 BIE 配置和 CFC 配置处于同一区域，例如同在北京/广州；
- 确保您的函数在 CFC 平台已经发布。
- 创建存储卷时，选择 CFC 函数模板进行创建，可以引入在 CFC 平台发布的函数，更多存储卷应用内容可参考 如何正确地引入存储卷

问题 13：配置文件中的 ports 和 Hub 配置文件中的 listen 有什么关系？

参考方案:

- ports 配置了宿主机和容器内的端口映射关系；
- listen 则是 Hub 的监听端口，Hub 是进程模式则监听宿主机的端口，Hub 是容器模式则监听容器内的端口；
- 请参考配置文件解读文档。

问题 14：消息通过 Baetyl 传输到 百度云 IoT Hub 后，如何在云端进行后续的数据处理？

参考方案：消息到达物接入后，可以通过 规则引擎 进行简单的 sql 处理，或者通过规则引擎传输给 百度云函数计算平台 等其他云端服务，具体配置详情参考 规则引擎操作指南。更多配置亦可参考通过 *Baetyl* 实现数据脱敏后存储百度云 *TSDB* 一文。

问题 15：Baetyl 如何使用*Remote* 功能连接 百度云 IoT Hub 设备型项目？

参考方案：Baetyl 端云协同强制使用证书认证，目前物接入设备型项目还不支持证书认证，作为临时方案可以在本地手动配置用户名密码和物接入设备型项目交互。

问题 16：如果要保证消息不丢，所有消息都能被同步到云端，该怎么做？

参考方案:

需要满足下如下两个条件:

- 发送给本地 Hub 的消息的 QoS 必须是 1，保证消息在本地持久化。
- Remote 向本地 Hub 订阅消息的 QoS 和向云端 Hub 发布消息的 QoS 也必须都为 1，保证消息成功上云。请参考配置文件解读文档。

问题 17：从云端下发配置到边缘后，默认都是 `docker` 模式启动，修改 `mode: native` 之后，启动报错类似下述内容：”failed to update system: open /Users/xxx/baetyl_native/var/run/baetyl/services/agent/lib/baetyl/hub.baidubce.com/baetyl/baetyl-agent:latest/package.yml: no such file or directory”

参考方案：目前我们的云管理不支持进程模式，如果需要在本地以进程模式启动 `Baetyl` 请参考 `example/native` 中的配置内容，执行命令 `make install-native` 来进行安装，并通过命令 `sudo baetyl start` 以进程方式来启动。

问题 18：下载镜像时有类似报错：”error=” Error response from daemon: Get https://hub.baidubce.com/v2/: x509: certificate signed by unknown authority”

baetyl=master"Or "error="Error response from daemon: Get https://hub.baidubce.com/v2/: x509: failed to load system roots and no roots provided" baetyl=master"

参考方案：这种情况是因为系统中缺少 https://hub.baidubce.com 网站的证书导致，可以先尝试安装 `ca-certificates` 包解决。例如，如宿主机系统为 debian ，可使用以下命令操作

```
sudo apt-get update
sudo apt-get install ca-certificates
sudo service docker restart
```

如果仍然报错，可以使用 openssl 添加 https://hub.baidubce.com 网站的证书到系统中

```
echo -n | openssl s_client -showcerts -connect hub.baidubce.com:443 2>/dev/null | sed -
↪ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' | sudo tee -a /etc/ssl/certs/ca-
↪certificates.crt
sudo service docker restart
```

其他系统可类比上述方案进行解决。

问题 19：CentOS7 系统上启动 Baetyl 时，报没有权限的错误： " container (0054b7d0da0f) [agent][agent] failed to load config: open etc/baetyl/service.yml: permission denied" Or "container (0054b7d0da0f) failed to parse log level (), use default level (info)[agent][agent] service is stopped with error: open etc/baetyl/service.yml: permission denied"

参考方案：这种情况是因为 CentOS7 中的安全模块 selinux 并未开放相关文件的权限，Docker 19.03 版本已经修复这个问题。用户可以参考 docker.com/install 升级到 19.03 或以上版本。

CHAPTER 20

资源下载

20.1 MQTT 相关资源下载

20.1.1 下载 MQTT 客户端代码示例

C 代码示例：下载 MQTT-C-MQTT-client

Python 代码示例：下载 MQTT-Python-MQTT-client

20.1.2 下载 MQTT.fx 客户端

源站：http://www.jensd.de/apps/mqttfx/1.7.1/

百度云 BOS MQTT.fx 安装包下载链接：

20.1.3 下载 MQTTBox 客户端

源站：http://workswithweb.com/html/mqttbox/downloads.html

百度云 BOS MQTTBox 安装包下载链接：

20.1.4 下载 MQTT Client SDK

物接入与 Paho（即 MQTT Client SDK）完全兼容，如果开发者需要开发 MQTT 客户端，可到 Paho 官方网站 下载 SDK 并获取帮助文档，详情如下：