概述 Overview

> 原文

Pitaya 是一个易于使用、快速且轻量级的游戏服务器框架，其灵感来自 starx 和 pomelo，并构建在 nano 的网络库之上。

pitaya 的目标是为分布式多人游戏和服务器端应用程序提供一个基本的、健壮的开发框架。

特性 Features

- **User sessions** - Pitaya 支持用户会话，允许将会话绑定到用户 ID，设置自定义数据并在会话处于活动状态时在其他地方检索它
- **Cluster support** - Pitaya 自带对 服务发现 和 RPC 模块的支持，允许不同类型的服务器之间轻松通信
- **WS and TCP listeners** - Pitaya 支持 `TCP` 和 `Websocket` 接受器，它们是从接收请求的应用程序中抽象出来的
- **Handlers and remotes** - Pitaya 允许应用程序指定 `handlers` （接收和处理客户端消息）及其 `remotes` （接收和处理 RPC 服务器消息）。 它们都可以指定自定义 `init` 、 `afterinit` 和 `shutdown` 方法
- **Message forwarding** - 当服务器收到处理程序消息时，它会将消息转发给正确类型的服务器
- **Client library SDK** - libpitaya 是 Pitaya 的官方客户端库 SDK
- **Monitoring** - Pitaya 默认支持 `Prometheus` 和 `statsd` ，并接受其他实现 `Reporter` 接口的自定义报告器
- **Open tracing compatible** - Pitaya 与 opentracing 兼容，因此使用 Jaeger 或任何其他兼容的跟踪框架很简单
- **Custom modules** - Pitaya 已经有一些默认模块,并且也支持自定义模块
- **Custom serializers** - Pitaya 原生支持 `JSON` 和 `Protobuf` 消息，并且可以根据需要添加其他自定义序列化器
- **Write compatible servers in other languages** - 使用 libpitaya-cluster 可以用其他语言编写能够在集群中注册并处理 `RPC` 的 pitaya 兼容服务器，已经有一个兼容的 `unity` 的 `csharp` 库和一个 兼容 `WIP` 的 `python` 库在仓库中。
- **REPL Client for development/debugging** - Pitaya-cli 是一个 REPL 客户端，可用于简化 pitaya 服务器的开发和调试。
- **Bots for integration/stress tests** - Pitaya-bot 是一个服务器测试框架，可以轻松复制用户行为以测试极端情况，从而验证收到的响应，或对 pitaya 服务器进行大量访问。

架构 Architecture

Pitaya 的开发以 模块化 和 可扩展性 为核心，同时提供可靠的基本功能来将客户端交互抽象为定义良好的接口。 完整的 API 文档在 godoc 上以 Godoc 格式提供。

谁在使用 Who's Using it

好吧，现在只有我们 TFG Co 的人在使用它，但如果能围绕该项目建立一个社区就太好了。 希望能尽快收到你们的关注！

如何贡献 How To Contribute?

就像往常一样：Fork、Hack、Pull Request。也不要忘记包括测试和文档（我们非常希望你们提供这两项）。

特性 Features

> 原文

Pitaya 具 模块化 和 可配置 的架构，有助于隐藏 应用程序实现扩展细节、管理客户端会话和通信 等复杂性。

下面介绍了它的一些核心功能。

前端和后端服务器 Frontend and backend servers

在集群模式下，服务器可以分为前端服务器 或 后端服务器。

前端服务器(`Frontend Servers`)必须指定侦听器(listeners)以接收传入的客户端连接。 它们能够根据路由逻辑将接收到的消息转发到适当的服务器。

后端服务器(`Backend servers`)不监听连接，它们只接收 `RPC` ，转发的客户端消息（ `sys rpc` ）或来自其他服务器的 RPC （ `user rpc` ）。

分组 Groups

分组是存储有关目标用户的信息，并允许向组中的所有用户发送广播消息，以及根据某些规则向用户子集广播(`broadcast`)消息的结构。

它们对于创建游戏房间很有用，例如，您只需将游戏房间中的所有玩家放入同一个组，然后您就可以向所有人广播房间的状态。

侦听器 Listeners

前端服务器(`Frontend Servers`)必须指定一个或多个接受器(`acceptors`)来处理传入的客户端连接，Pitaya 带有已经实现的 `TCP` 和 `Websocket` 接受器，并且可以通过实现接受器接口将其他接受器添加到应用程序中。

接收器包装 Acceptor Wrappers

包装器可用于接受器（如 `TCP` 和 `Websocket` ），以在执行消息转发之前读取和更改传入数据。 要创建一个新的包装器，只需实现 `Wrapper` 接口（或从 `BaseWrapper` 继承结构）并使用 `WithWrappers` 方法将其添加到您的接受器中。接下来是接受器包装器的一些示例。

限速器 Rate limiting

读取每个玩家连接上的传入数据以限制请求吞吐量。 超过限制后，请求将被丢弃，直到插槽再次可用。 请求计数和管理是在玩家的连接上完成的，因此它甚至发生在会话绑定之前。 使用的算法是漏桶(Leaky Bucket)。 该算法代表一个漏桶，其输出流量比输入流量慢。 它将每个请求时间戳保存在一个插槽(`slot`)中（总共限制插槽 `limit` slots），并且该插槽在间隔后再次释放。 例如：如果在1秒的间隔内限制1个请求，当一个请求发生在0.2s时，下一个请求只会在1s后（at 1.2s）由pitaya处理。

```
0     request
|--------|
   0.2s
0                 available again
|------------------------|
|- 0.2s -|----- 1s ------|
```

消息转发 Message forwarding

当服务器实例收到客户端消息时，它会通过查看路由(`route`)来检查目标服务器类型(`server type`)。 即使目标服务器类型与接收服务器类型不同，该消息也能转发到正确类型的服务器实例。 客户端不需要采取任何行动来转发消息，这个过程是由 Pitaya 自动完成的。

默认情况下，路由功能随机选择目标服务器类型的一个实例。 可以定义自定义函数来更改此行为。

消息推送 Message push

在没有相关的前置信息如会话(`session`)或连接状态(`connection status`)的情况下将消息推送给客户端。 这些推送消息具有路由（以便客户端可以正确识别来源并处理）、消息、目标id 和 服务器类型(客户端预期连接到的)。

模块 Modules

模块是可以注册到 Pitaya 的应用实例，且必须实现定义的接口(interface)。 Pitaya 负责根据需要调用适当的生命周期方法，可以通过名称检索已注册的模块。

Pitaya 自带了一些已经实现的模块，可以根据需要实现更多的模块。 Pitaya目前拥有的模块有：

二进制模块 Binary

该模块将二进制文件作为子进程(`child process`)启动，并将其标准输出(`stdout`)和标准错误(`stderr`)分别通过管道(`pipes`)传输信息和错误日志。

唯一会话模块 Unique session

此模块为 `OnSessionBind` 添加回调，检查正在绑定的 id 是否已在其他前端服务器之一中绑定。

绑定存储模块 Binding storage

该模块通过 `gRPC` 实现 `RPC` 所需的功能，在不知道用户连接到的服务器的情况下，通过 广播会话绑定 和 推送 给用户。

监控 Monitoring

Pitaya 支持指标报告，它带有已经实现的 `Prometheus` 和 `Statsd` 支持，并且支持实现 `Reporter` 接口的自定义报告器。 Pitaya 还支持开放跟踪兼容框架，允许轻松集成 `Jaeger` 和其他框架。

`Reporter` 指标报告列表如下：

- 响应时间(`Response time`)：处理消息的时间，以纳秒为单位。 它是分段的按路线、状态、服务器类型和响应代码；
- 处理延迟时间(`Process delay time`)：开始处理消息的延迟，以纳秒为单位；它按路由和服务器类型分段；
- `Exceeded Rate Limit` ： 超出速率限制的阻塞请求数；
- 连接的客户端(`Connected clients`)：此时连接的客户端数量；
- 服务器计数(`Server count`)：通过服务发现发现的服务器数量。 这是按服务器类型划分；
- 信道容量(`Channel capacity`)：信道的可用容量；
- `Dropped messages` ： rpc server dropped messages的数量，即没有处理的消息；
- `Goroutines count` ： 当前的Goroutines个数；
- `Heap size` ： 当前堆大小；
- `Heap objects count` ： 堆中当前的对象数；

- `Reliable RPCs` 重试(`Worker jobs retry`): 当前 `reliability RPC` 重试次数;
- `Reliable RPCs` 总数: 当前 `reliability RPC` 数量。 按状态划分;
- `Reliable RPCs` 队列大小: `reliability RPC` 队列的当前大小。 它由每个可用队列组成。

自定义指标报告 Custom Metrics

除了 Pitaya 默认监控外，还可以创建新指标。 如果只使用 `Statsd reporter` ，则不需要配置。 如果使用 `Prometheus` ，则有必要添加一个指定指标参数的配置。 有关文档和此示例。

管道 Pipelines

管道是允许方法在处理程序请求之前和之后执行的中间件，它们接收请求的上下文(`context`)和请求数据(`request data`)并返回请求数据，这些数据将传递给管道中的下一个方法。

RPCs

Pitaya 在集群模式(`cluster mode`)下支持 RPC 调用，通过两个组件可以启用此功能: RPC 客户端和 RPC 服务器。 目前 Pitaya 实现的 RPC 是通过， `NATS` 和 `gRPC` ，默认是 `NATS` 。

有两种类型的 RPCs, `_Sys_` and `_User_` .

系统RPCs Sys RPCs

服务器在将处理程序消息转发到适当的服务器类型时使用的 系统RPC。

用户RPCs User RPCs

当应用程序主动调用另一台服务器中的远程方法时，使用 用户RPC。 调用可以指定目标服务器的ID，也可以让Pitaya 根据路由逻辑选择一个。

用户可靠RPCs User Reliable RPCs

通过使用 `worker` 完成 RPC 调用，也就是说，如果发生任何错误，Pitaya 会进行重试。

重要: 被调用的接口必须是幂等(`idempotent`)的； ReliableRPC没有返回值，因为它是异步的，它只在成功时返回作业 ID (jid)。

服务器运行模式 Server operation mode

Pitaya 有两种运行方式：单机模式和集群模式。

单机模式 Standalone mode

在单机模式下，服务器不相互交互，不使用服务发现，并且不支持RPC。 这是一个有限版本的框架，可以在应用程序不需要处理不同类型的服务器间通信时使用。

集群模式 Cluster mode

集群模式是一种更完整的模式，使用服务发现、RPC 客户端和服务器以及应用程序服务器之间的远程通信。 此模式对于更复杂的应用程序很有用，这些应用程序可能会受益于在不同专门类型的服务器之间拆分职责。 Pitaya 已经为这种模式的 RPC调用 和 服务发现 提供了默认服务。

序列化器 Serializers

Pitaya 支持不同类型的消息序列化器，用于发送到客户端和从客户端发送的消息，默认序列化器是 `JSON` 序列化器，Pitaya 也原生支持 `Protobuf` 序列化器。 可以通过实现 `serialize.Serializer` 接口来实现新的序列化器。

应用程序可以通过调用 `pitaya` 包中的 `SetSerializer` 方法来设置所需的序列化程序。

Service discovery

以集群模式运行的服务器必须具有服务发现的客户端才能工作。 Pitaya 附带一个使用 `etcd` 的默认客户端，如果没有定义其他客户端，则使用该客户端。 服务发现客户端负责注册服务器并保持有效服务器列表的更新，以及根据需要提供有关请求的服务器的信息。

会话 Sessions

客户端建立的每个连接都有一个关联的会话(`session`)实例，它在连接关闭时被销毁。 会话是 Pitaya 核心功能的一部分，因为它们允许与客户端进行异步通信并在请求之间存储数据。 会话的主要特点是：

- **ID 绑定** - 会话可以绑定到用户ID，允许应用程序的其他部分向用户发送消息，而无需知道用户连接到哪个服务器或连接
- **数据存储** - 会话可用于数据存储，在请求之间存储和检索数据
- **消息传递** - 消息可以通过会话发送给连接的用户，无需了解底层连接协议
- **通过请求访问** - 会话可以通过请求的上下文实例中获得并访问
- Kick - 可以通过会话的 `Kick` 方法将用户从服务器中踢出

尽管前端和后端服务器上的处理程序请求都可以访问会话，但前端会话与后端会话的行为就会有点不同。 这主要是因为会话实际上存在于前端服务器中，而只是将其状态表现(`representation of its state`)发送到后端服务器。

如果会话是从前端服务器访问的，则会话被认为是前端会话(`Frontend sessions`)，而后端会话(`Backend sessions`)是从后端服务器访问。 下面详细描述每种会话。

前端会话 Frontend sessions

会话与前端服务器中的连接相关联，可以通过建立连接的服务器中的 会话ID 或绑定 用户ID 检索，但不能从其他服务器检索。

可以将回调(`Callbacks`)添加到某些会话生命周期更改中，例如关闭(`closing`)和绑定(`binding`)。 回调可以基于(`per-session basis`)每次会话（使用 `s.OnClose` ）或每个会话（使用 `OnSessionClose` 、 `OnSessionBind` 和 `OnAfterSessionBind` ）。

后端会话 Backend sessions

后端会话可以通过处理程序(`handler`)的方法访问会话，但它们有一些限制和特殊特征。 会话变量的更改必须通过调用 `s.PushToFront` 推送到前端服务器（但 `s.Bind` 操作不需要推送），也不允许设置会话生命周期操作的回调，也不能通过后端服务器的 用户ID 检索会话。

通信 Communication

> 原文

本节我们将详细描述客户端与服务器端的通信过程。从建立连接、发送请求到接收响应。该示例将假设应用程序在集群模式下运行，并且目标服务器与客户端连接的服务器不同。

建立连接 Establishing the connection

客户端连接并发出请求建立连接的过程如下：

- 与接受者(acceptor)建立低级连接(net.Conn)
- 将连接传递给处理程序服务(handler service)
- 处理程序服务为连接创建一个新代理(agent)
- 处理程序服务从连接中读取消息
- 使用配置的解码器(decoder)对消息进行解码(decoded)
- 来自消息的解码包被处理
- 第一个数据包必须是握手请求(handshake request)，服务器向其返回带有序列化器(deserialized)、路由字典(route dictionary)和心跳超时(heartbeat timeout)的握手响应
- 然后客户端必须回复握手确认(handshake ack)，然后建立连接
- 数据消息由处理程序(handler)处理并从消息路由中提取目标服务器类型(target server type)，使用指定方法反序列化(deserialized)消息
- 如果目标服务器类型与当前服务器不同，则服务器远程调(RPC)正确类型的服务器，根据路由功能逻辑(routing function logic)选择一个服务器。远程调用携带了当前客户端的会话表现(representation of the client's session)
- 接收远程服务器接收请求并将其作为 _Sys_ RPC 调用处理，创建一个新的远程代理来处理请求，该代理接收会话表现
- 调用前管道函数(before pipeline functions)并反序列化(deserialized)消息
- 然后远程服务器调用适当的处理程序(handler)，返回响应(response)，然后序列化(serialized)并执行后管道函数(after pipeline functions)
- 如果后端服务器要修改会话，并将修改显式推送到前端服务器
- 一旦前端服务器收到响应，它将消息转发到指定请求(消息ID)的会话
- 代理(agent)接收请求，对其进行编码(encode)并发送到低级连接

接收器 Acceptors

客户端必须做的第一件事是与 Pitaya 服务器建立连接。为此，服务器必须指定一个或多个接受器。

接受器是负责监听连接、建立它们、抽象它们并将它们转发给处理程序服务的实体。Pitaya 支持 TCP 和 websocket 接受器。可以实现自定义接受器并将其添加到 Pitaya 应用程序中，它们只需要实现适当的接口即可。

处理器服务 Handler service

建立低级连接(net.Conn)后，将其传递给处理程序服务(handler service)进行处理。处理程序服务负责处理客户端连接的生命周期。它从低级连接读取，解码接收到的数据包并正确处理它们，如果目标服务器类型与本地服务器类型相同，则调用本地服务器的处理程序，否则将消息转发到远程服务。

Pitaya 有一个配置来定义同时处理的并发消息数，本地和远程消息都计入并发，因此如果服务器希望处理慢速路由，则可能需要稍微调整此配置。配置是 pitaya.concurrency.handler.dispatch 。

代理 Agent

代理实体负责存储有关客户端连接的信息，它存储会话(`session`)、编码器(`encoder`)、序列化器(`serializer`)、状态(`state`)、连接(`connection`)等。它用于与客户端通信以发送消息并确保连接保持活动状态。

路由压缩 Route compression

应用程序可以在启动前定义一个压缩路由字典(`dictionary of compressed routes`)，在握手时发送给客户端。压缩路由可能对经常使用的路由有用，以减少通信开销。

握手 Handshake

客户端连接时发生的第一个操作是握手(`handshake`)。握手由客户端发起，客户端发送客户端信息，如平台(`platform`)、客户端库版本(`version`)等，也可以在这一步发送用户数据。此数据存储在客户端的会话中，以后可以访问。服务器回复心跳间隔(`heartbeat interval`)、序列化器名称(`name of the serializer`)和压缩路由字典(`the dictionary of compressed routes`)。

远程服务 Remote service

远程服务既负责创建 RPC，也负责接收和处理它们。在转发客户端请求的情况下，RPC 是 `Sys` 类型。

在调用方，服务负责根据服务器类型和路由逻辑来识别要调用的正确服务器。

在接收端，服务识别它是一个 `Sys RPC` ，并创建一个远程代理来处理请求。这个远程代理是短暂的，仅在请求存在时才存在，对后端会话的更改不会自动反映在相关的前端会话中，它们需要通过推送它们来显式提交。然后将消息转发给适当的处理程序进行处理。

管道 Pipeline

Pitaya 中的管道是一组函数，可以定义为在每个处理程序请求之前或之后运行。这些函数接收上下文和原始消息，并应返回请求对象和错误，允许它们修改上下文并返回修改后的请求。如果 `before` 函数返回错误，则请求失败并且进程中止。

序列化器 Serializer

处理程序(`handler`)必须先反序列化消息，然后再处理它。因此负责调用处理程序方法的函数首先反序列化(`deserialize`)消息，调用方法后再序列化方法返回的响应并将其返回给远程服务。

处理程序 Handler

每个 Pitaya 服务器都可以注册多个处理程序结构，只要它们具有不同的名称即可。每个结构可以有多个方法，Pitaya 会根据调用的路由选择合适的结构和方法。

配置 Configuration

> 原文

Pitaya 使用 Viper 来控制其配置。 下面我们描述按主题拆分的配置变量。 默认值适用于大多数情况，但对于某些用例可能需要更改。

服务发现 Service Discovery

这些配置值为默认的 `etcd` 服务发现模块配置服务发现。

只有在应用程序以集群模式运行时才需要设置它们。

配置项	默认值	类型	描述
pitaya.cluster.sd.etcd.dialtimeout	5s	time.Time	传递给服务发现 `etcd` 客户端的拨号超时值
pitaya.cluster.sd.etcd.endpoints	localhost:2379	string	逗号分隔的 etcd 端点列表
pitaya.cluster.sd.etcd.user		string	连接 etcd 的用户名
pitaya.cluster.sd.etcd.pass		string	连接 etcd 的密码
pitaya.cluster.sd.etcd.heartbeat.ttl	60s	time.Time	etcd 租约的心跳间隔
pitaya.cluster.sd.etcd.grantlease.timeout	60s	time.Duration	etcd 租约超时
pitaya.cluster.sd.etcd.grantlease.maxretries	15	int	etcd 授予租约的最大尝试次数
pitaya.cluster.sd.etcd.grantlease.retryinterval	5s	time.Duration	每次授权租约尝试之间的间隔
pitaya.cluster.sd.etcd.revoke.timeout	5s	time.Duration	etcd 的撤销函数超时
pitaya.cluster.sd.etcd.heartbeat.log	false	bool	是否在调试模式下记录 etcd 心跳
pitaya.cluster.sd.etcd.prefix	pitaya/	string	前缀用于避免与不同的 pitaya 应用程序发生冲突，服务器必须具有相同的前缀才能相互看到
pitaya.cluster.sd.etcd.syncservers.interval	120s	time.Duration	服务发现模块执行的服务器同步之间的间隔
pitaya.cluster.sd.etcd.shutdown.delay	10ms	time.Duration	从服务发现中注销后等待关闭的时间
pitaya.cluster.sd.etcd.servertypeblacklist	nil	[]string	服务发现应忽略的服务器类型列表
pitaya.cluster.sd.etcd.syncserversparallelism	10	int	获取有关 etcd 初始化的服务器信息时应使用的 goroutine 数量

RPC服务 RPC Service

只有在使用给定类型启用 RPC 服务时才需要设置配置。

配置项	默认值	类型	描述
pitaya.cluster.rpc.server.nats.buffer.messages	75	int	nats RPC服务器接的缓冲区大小(超过将丢弃传入的消息)
pitaya.cluster.rpc.server.nats.buffer.push	100	int	nats RPC服务器为推送消息创建的缓冲区大小
pitaya.cluster.rpc.client.grpc.dialtimeout	5s	time.Time	gRPC 客户端建立连接超时
pitaya.cluster.rpc.client.grpc.lazyconnection	false	bool	gRPC 客户端是否使用惰性连接，即只有在向该服务器发出请求时才连接
pitaya.cluster.rpc.client.grpc.requesttimeout	5s	time.Time	使用 gRPC 客户端的请求超时
pitaya.cluster.rpc.client.nats.connect	nats://localhost:4222	string	nats 客户端的地址
pitaya.cluster.rpc.client.nats.connectiontimeout	5s	time.Duration	nats连接超时时间
pitaya.cluster.rpc.client.nats.requesttimeout	5s	time.Time	nats RPC调用的请求超时时间
pitaya.cluster.rpc.client.nats.maxreconnectionretries	15	int	nats 客户端连接的最大重试次数
pitaya.cluster.rpc.server.nats.connect	nats://localhost:4222	string	nats 服务器的地址
pitaya.cluster.rpc.server.nats.connectiontimeout	5s	time.Duration	nats server建立连接超时时间
pitaya.cluster.rpc.server.nats.maxreconnectionretries	15	int	服务器重连到 nats 的最大重试次数
pitaya.cluster.rpc.server.grpc.port	3434	int	gRPC 服务器监听的端口
pitaya.cluster.rpc.server.nats.services	30	int	nats RPC服务在远程服务处理消息的 goroutines 数量
pitaya.worker.redis.url	localhost:6379	string	redis服务器地址

配置项	默认值	类型	描述
pitaya.worker.redis.pool	10	string	连接池大小
pitaya.worker.redis.password		string	redis密码
pitaya.worker.concurrency	1	int	Reliable RPC并发数
pitaya.worker.namespace		string	命名空间，用于分区部署
pitaya.worker.retry.enabled	true	bool	是否开启错误重试
pitaya.worker.retry.max	5	int	最大重试次数
pitaya.worker.retry.exponential	2	int	重试等待的间隔: nRetry**2
pitaya.worker.retry.minDelay	0	int	最小等待的间隔时间
pitaya.worker.retry.maxDelay	10	int	最大等待的间隔时间
pitaya.worker.retry.maxRandom	10	int	随机等待的间隔时间

连接 Connection

配置项	默认值	类型	描述
pitaya.handler.messages.compression	true	bool	是否压缩消息
pitaya.heartbeat.interval	30s	time.Time	心跳包发送间隔
pitaya.conn.ratelimiting.interval	1s	time.Duration	统计请求数量的时间周期
pitaya.conn.ratelimiting.limit	20	int	时间周期内允许的最大请求数
pitaya.conn.ratelimiting.forcedisable	false	bool	如果为 true，即使在添加 `WithWrappers` 时也会忽略速率限制

指标报告 Metrics Reporting

配置项	默认值	类型	描述
pitaya.metrics.statsd.enabled	false	bool	是否启用 statsd 报告
pitaya.metrics.statsd.host	localhost:9125	string	statsd 服务器的地址
pitaya.metrics.statsd.prefix	pitaya.	string	前缀
pitaya.metrics.statsd.rate	1	int	统计指标率
pitaya.metrics.prometheus.enabled	false	bool	是否启用 prometheus 报告
pitaya.metrics.prometheus.port	9090	int	prometheus 服务器端口
pitaya.metrics.constTags	map[string]string{}	`map`	要添加到报告指标的常量标签
pitaya.metrics.prometheus.additionalTags	map[string]string{}	`map`	报告指标的附加标签，映射是从标签到默认值
pitaya.metrics.period	15s	string	报告指标的采样周期
pitaya.metrics.custom.counters	[]map[string]interface{}	`map`	自定义指标计数器
pitaya.metrics.custom.counters[].Subsystem		string	自定义计数器子系统名称
pitaya.metrics.custom.counters[].Name		string	自定义计数器名称，不能为空
pitaya.metrics.custom.counters[].Help		string	自定义计数器帮助，解释是什么指标，不能为空
pitaya.metrics.custom.counters[].Labels	[]string{}	`slice`	指标将携带的自定义计数器标签
pitaya.metrics.custom.gauges	[]map[string]interface{}	`map-slice`	自定义指标仪表
pitaya.metrics.custom.gauges[].Subsystem		string	自定义仪表子系统名称
pitaya.metrics.custom.gauges[].Name		string	自定义仪表名称，不能为空
pitaya.metrics.custom.gauges[].Help		string	自定义指标帮助，解释是什么仪表，不能为空
pitaya.metrics.custom.gauges[].Labels	[]string{}	`slice`	指标将携带的自定义仪表标签
pitaya.metrics.custom.summaries	[]map[string]interface{}	`map-slice`	自定义指标摘要
pitaya.metrics.custom.summaries[].Subsystem		string	自定义摘要子系统名称
pitaya.metrics.custom.summaries[].Name		string	自定义摘要名称，不能为空
pitaya.metrics.custom.summaries[].Help		string	自定义摘要帮助，解释是什么摘要，不能为空

配置项	默认值	类型	描述
pitaya.metrics.custom.summaries[].Labels	[]string{}	`slice`	指标将携带的自定义摘要标签
pitaya.metrics.custom.summaries[].Objectives	map[float64]float64	`map`	带有分位数的自定义摘要目标

并发 Concurrency

配置项	默认值	类型	描述
pitaya.buffer.agent.messages	100	int	每个代理接收的客户端消息的缓冲区大小
pitaya.buffer.handler.localprocess	20	int	处理程序接收并在本地处理的消息的缓冲区大小
pitaya.buffer.handler.remoteprocess	20	int	处理程序接收并转发到远程服务器的消息的缓冲区大小
pitaya.concurrency.handler.dispatch	25	int	在处理程序服务中处理消息的 goroutines 数量

模块 Modules

这些配置仅在创建模块时使用。建议将绑定存储模块与 gRPC RPC 服务一起使用，以便能够使用所有 RPC 服务功能。

配置项	默认值	类型	描述
pitaya.session.unique	true	bool	Pitaya 是否为客户端强制执行唯一会话，启用唯一会话模块
pitaya.modules.bindingstorage.etcd.endpoints	localhost:2379	string	绑定存储模块使用的 etcd 端点的逗号分隔列表，应该与服务发现 etcd 相同
pitaya.modules.bindingstorage.etcd.prefix	pitaya/	string	用于 etcd 的前缀，应该与服务发现相同
pitaya.modules.bindingstorage.etcd.dialtimeout	5s	time.Time	建立etcd连接超时
pitaya.modules.bindingstorage.etcd.leasettl	1h	time.Time	自动续订前 etcd 租约的持续时间

默认管道 Default Pipelines

这些配置控制是否应启用默认管道

配置项	默认值	类型	描述
pitaya.defaultpipelines.structvalidation.enabled	false	bool	Pitaya 是否为处理程序参数启用默认结构验证器

分组 Groups

这些配置用于组服务实施。

配置项	默认值	类型	描述
pitaya.groups.etcd.endpoints	localhost:2379	string	etcd 服务分组，使用逗号分隔列表
pitaya.groups.etcd.prefix	pitaya/	string	用于 etcd 中每组Key的前缀
pitaya.groups.etcd.dialtimeout	5s	time.Time	建立 etcd 组连接的超时时间
pitaya.groups.etcd.transactiontimeout	5s	time.Duration	完成组请求 etcd 的超时时间
pitaya.groups.memory.tickduration	30s	time.Duration	将检查删除组的持续时间

Pitaya API

> 原文

处理程序 Handlers

处理程序是 Pitaya 的核心功能之一，它们是负责接收来自客户端的请求并处理它们的实体，如果方法是请求处理程序则返回响应，如果方法是通知处理程序则不返回任何内容。

签名 Signature

处理程序必须是结构的公共方法并具有以下签名：

参数

- `context.Context`： 请求的上下文，其中包含客户端的会话。
- `pointer or []byte`： 请求的有效负载(`payload`)（可选）。

通知处理程序(`Notify`)不返回任何内容，而请求处理程序(`request handlers`)必须返回：

- `pointer or []byte`：响应负载
- `error`：错误变量

注册处理程序 Registering handlers

应用程序必须通过使用 `Register` 调用 pitaya 实例来显式注册处理。通过 `pitaya/component` .WithName(`"handlerName"`)来定义"handlerName"，方法可以通过使用 `pitaya/component.WithNameFunc(func(string) string)` 来重命名。

客户端可以通过 `serverType.handlerName.methodName` 来调用处理程序。

路由消息 Routing messages

消息由 pitaya 转发到适当的服务器类型，并且可以通过调用 pitaya 应用程序将自定义路由器添加到应用程序 `AddRoute` ，它需要两个参数：

- `serverType`：路由的目标服务器类型
- `routingFunction`：带签名的路由函数，它接收用户的会话、请求的路由、消息和给定类型的有效服务器的映射，键是服务器的id `func(session.Session, *route.Route, []byte, map[string]*cluster.Server) (*cluster.Server, error)`

当将请求路由到给定的服务器类型时，服务器将使用路由功能。

生命周期方法 Lifecycle Methods

处理程序(`handler`)可以选择实现以下生命周期方法：

- `Init()` - 初始化应用程序时由 Pitaya 调用
- `AfterInit()` - 初始化应用程序后由 Pitaya 调用
- `BeforeShutdown()` - 在关闭组件时由 Pitaya 调用，但在调用关闭之前
- `Shutdown()` - 启动关机后由Pitaya调用

处理程序示例 Handler example

下面是一个非常简单的示例，有关完整的工作示例，请查看集群演示。

```
import (
  "github.com/topfreegames/pitaya"
  "github.com/topfreegames/pitaya/component"
)

type Handler struct {
  component.Base
}

type UserRequestMessage struct {
  Name    string `json:"name"`
  Content string `json:"content"`
}

type UserResponseMessage {
}

type UserPushMessage{
  Command string `json:"cmd"`
}

// Init runs on service initialization (not required to be defined)
func (h *Handler) Init() {}

// AfterInit runs after initialization (not required to be defined)
func (h *Handler) AfterInit() {}

// TestRequest can be called by the client by calling <servertype>.testhandler.testrequest
func (h *Handler) TestRequest(ctx context.Context, msg *UserRequestMessage)
(*UserResponseMessage, error) {
  return &UserResponseMessage{}, nil
}

func (h *Handler) TestPush(ctx context.Context, msg *UserPushMessage) {
}

func main() {
  builder := pitaya.NewDefaultBuilder()
  ...
  app := builder.Build()

  app.Register(
    &Handler{}, // struct to register as handler
    component.WithName("testhandler"), // name of the handler, used by the clients
    component.WithNameFunc(strings.ToLower), // naming conversion scheme to be used by the
clients
  )
  ...
  app.Start()
}
```

远程处理器 Remotes

远程处理器是 Pitaya 的核心功能之一，它们是负责从其他 Pitaya 服务器接收 RPC 的实体。

签名 Signature

Remotes 必须是该结构的公共方法，并具有以下签名：

参数

- `context.Context`：请求的上下文
- `proto.Message`：请求的有效负载（可选）

远程方法必须返回：

- `proto.Message`： `protobuf` 格式的响应负载
- `error`：错误变量

注册远程调用 Registering remotes

远程调用 必须由应用程序通过 `RegisterRemote` 组件调用pitaya来显式注册。通过 `pitaya/component` .WithName(`"remoteName"`)来定义，通过 `pitaya/component.WithNameFunc(func(string) string)` 来重命名。

服务器可以通过 `serverType.remoteName.methodName` 来调用远程

RPC calls

在服务器之间发送 RPC 时有两种选择：

- 仅指定服务器类型(`Specify only server type`)：在这种情况下，Pitaya 将随机选择一个可用服务器
- 指定服务器类型和ID(`Specify server type and ID`)：在这种情况下，Pitaya 将 RPC 发送到指定的服务器

生命周期方法 Lifecycle Methods

Remotes 可以选择实现以下生命周期方法：

- `Init()` - 初始化应用程序时由 Pitaya 调用
- `AfterInit()` - 初始化应用程序后由 Pitaya 调用
- `BeforeShutdown()` - 在关闭组件时由 Pitaya 调用，但在调用关闭之前
- `Shutdown()` - 启动关机后由Pitaya调用

远程示例 Remote example

有关完整的工作示例，请查看集群演示。