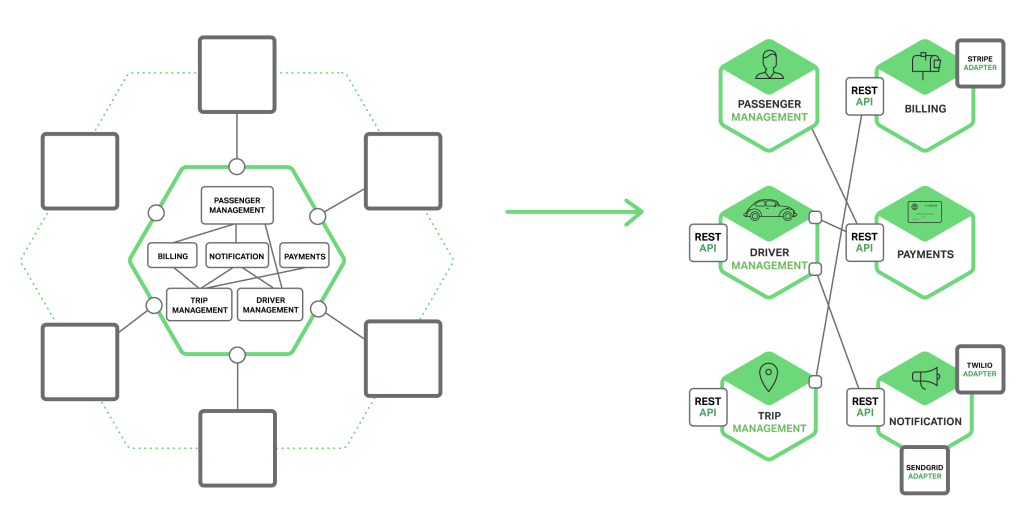
**简介**

在单体应用中，各模块之间的调用是通过编程语言级别的方法或者函数来实现的。而基于微服务的分布式应用是运行在多台机器上的；一般来说，每个服务实例都是一个进程。

因此，如下图所示，服务之间的交互必须通过**进程间通信（IPC）**来实现。

[](http://blog.daocloud.io/wp-content/uploads/2016/05/Richardson-microservices-part3-monolith-vs-microservices-1024x518-1.png)

后面我们将会详细介绍 IPC 技术，现在我们先来看下设计相关的问题。

**交互模式（答进程间有什么通信方式可以顺便答）**

当为某个服务选择 IPC 时，首先需要考虑服务之间的交互问题。**客户端和服务器之间有很多的交互模式**，我们可以从两个维度进行归类。

第一个维度是一对一还是一对多：

• 一对一：**每个客户端请求有一个服务实例**来响应。

• 一对多：**每个客户端请求有多个服务实例**来响应。

第二个维度是这些交互式是同步还是异步：

• **同步模式：客户端请求需要服务端即时响应，甚至可能由于等待而阻塞**。

• **异步模式：客户端请求不会阻塞进程，服务端的响应可以是非即时的。**

下表显示了不同交互模式：

[](http://blog.daocloud.io/wp-content/uploads/2016/05/Screen-Shot-2016-05-18-at-10.17.41-AM.png)

一对一的交互模式有以下几种方式：

1. 请求/响应：一个客户端向服务器端发起请求，等待响应，客户端期望此响应即时到达。**在一个基于线程的应用中，等待过程可能造成线程阻塞。**
2. 通知（也就是常说的单向请求）：**一个客户端请求发送到服务端，但是并不期望服务端响应。**
3. 请求/异步响应：**客户端发送请求到服务端，服务端异步响应请求。**客户端不会阻塞，而且被设计成默认响应不会立刻到达。

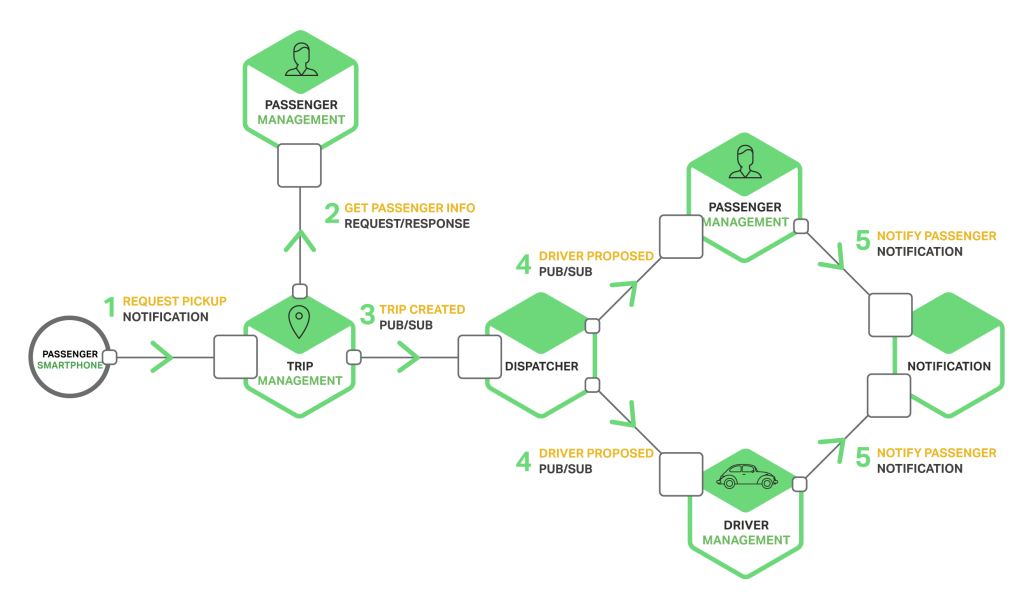
一对多的交互模式有以下几种方式：

4.发布/ 订阅模式：**客户端发布通知消息，被零个或者多个感兴趣的服务消费。**

5.发布/异步响应模式：**客户端发布请求消息，然后等待从感兴趣服务发回的响应。**

每个服务都是以上这些模式的组合。**对某些服务，一个 IPC 机制就足够了；而对另外一些服务则需要多种 IPC 机制组合。**

下图展示了在用户叫车时，打车应用内的服务是如何交互的。

[](http://blog.daocloud.io/wp-content/uploads/2016/05/Richardson-microservices-part3-taxi-service-1024x609-2.png)

上图中的服务通信使用了通知、请求/响应、发布/订阅等方式。例如，乘客在移动端向“行程管理”服务**发送通知**，请求一次接送服务。“行程管理”服务通过使用**请求/响应**来唤醒“乘客服务”来验证乘客账号有效，继而创建此次行程，并利用**发布/订阅**来通知其它服务，其中包括定位可用司机的调度服务。

现在我们了解了交互模式，接下来我们一起来看看如何定义 API。

**定义 API**

API 是服务端和客户端之间的契约。无论选择了何种 IPC 机制，重点是使用某种交互定义语言（IDL）来准确定义服务的 API。对于如何使用 API 优先的方式来定义服务，已经有了一些很好的讨论。你在开发服务之前，要定义服务接口并与客户端开发者共同讨论，后续只需要迭代 API 定义。这样的设计能够大幅提升服务的可用度。

在本文后半部分你将会看到，**API 定义实质上依赖于选定的 IPC 机制**。如果使用消息机制，API 则由消息频道（channel）和消息类型构成；如果选择使用 HTTP 机制，API 则由 URL 和请求、响应格式构成。后面将会详细描述 IDL。

**不断进化的 API**

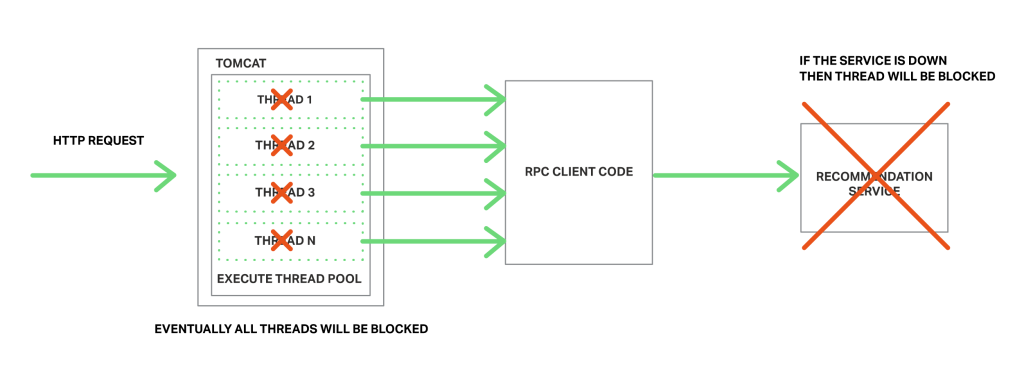
服务的 API 会随着时间而不断变化。在单体应用中，经常会直接修改 API 并更新所有的调用者。但是在基于微服务的应用中，即使所有的 API 的使用者都在同一应用中，这种做法也困难重重，通常不能强制让所有客户端都与服务保持同步更新。此外，你可能会增量部署服务的新版本，这时旧版本会与新版本同时运行。了解这些问题的处理策略至关重要。  
 对 API 变化的处理方式与变化的大小有关。有的变化很小，并且可以兼容之前的版本；比如给请求或响应增加属性。在设计客户端和服务时，很有必要遵循健壮性原则。服务更新版本后，使用旧版 API 的客户端应该继续使用。服务为缺失的请求属性提供默认值，客户端则忽略任何额外的响应。使用 IPC 机制和消息格式能够让你轻松改进 API。

然而有时候，API 需要进行大规模改动，并且不兼容旧版本。鉴于不能强制让所有客户端立即升级，支持旧版 API 的服务还要再运行一段时间。如果你使用的是诸如 REST 这样的基于 HTTP 机制的 IPC，一种方法就是将版本号嵌入到 URL 中，每个服务实例可以同时处理多个版本。另一种方法是部署不同实例，每个实例处理一个版本的请求。

**处理局部失败**

在上一篇关于 API 网关的文章中，我们了解到，**分布式系统普遍存在局部失败的问题。**由于客户端和服务端是独立的进程，**服务端可能无法及时响应客户端请求。服务端可能会因为故障或者维护而暂时不可用。服务端也可能会由于过载，导致对请求的响应极其缓慢。**

以上篇文章中提及的产品页为例，假设推荐服务无法响应，客户端可能会**由于无限期等待响应而阻塞。**这不仅会导致很差的用户体验，并且在很多应用中还会占用之前的资源，**比如线程；最终，如下图所示，运行时耗尽线程资源，无法响应。**

[](http://blog.daocloud.io/wp-content/uploads/2016/05/Richardson-microservices-part3-threads-blocked-1024x383-3.png)

为了预防这种问题，设计服务时候必须要考虑部分失败的问题。

Netfilix 提供了一个比较好的解决方案，具体的应对措施包括：

* 网络超时：在等待响应时，不设置无限期阻塞，而是**采用超时策略**。使用超时策略可以确保资源不被无限期占用。
* 限制请求的次数：可以**为客户端对某特定服务的请求设置一个访问上限**。如果请求已达上限，就要立刻终止请求服务。
* 断路器模式（Circuit Breaker Pattern）：记录成功和失败请求的数量。如果**失效率超过一个阈值，触发断路器使得后续的请求立刻失败**。如果大量的请求失败，就可能是这个服务不可用，再发请求也无意义。在一个失效期后，客户端可以再试，如果成功，关闭此断路器。
* **提供回滚：**当一个请求失败后可以进行回滚逻辑。例如，返回缓存数据或者一个系统默认值。  
  Netflix Hystrix 是一个实现相关模式的开源库。如果使用 JVM，推荐使用Hystrix。而如果使用非 JVM 环境，你可以使用类似功能的库。

**IPC 技术**

现在有很多不同的 IPC 技术。服务间通信可以使用**同步的请求/响应模式**，比如基于 HTTP 的 REST 或者 Thrift。另外，也可以选择异步的、基于消息的通信模式，比如 AMQP 或者 STOMP。此外，还可以选择 JSON 或者 XML 这种可读的、基于文本的消息格式。当然，也还有效率更高的二进制格式，比如 Avro 和 Protocol Buffer。在讨论同步的 IPC 机制之前，我们先了解异步的 IPC 机制。

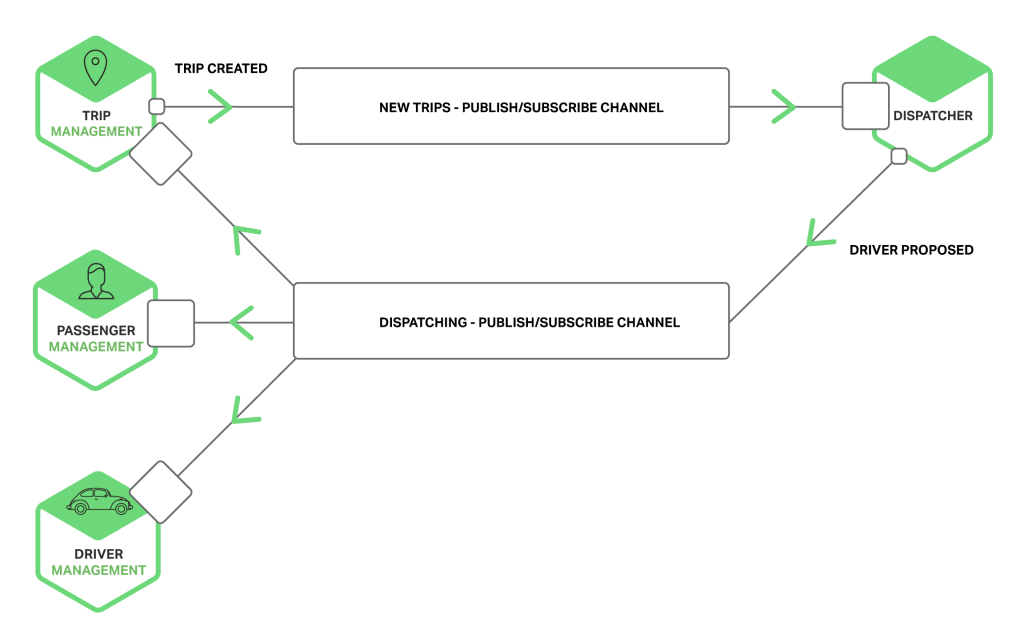
**基于异步的消息机制**

使用消息模式的时候，进程之间通过异步交换消息的方式通信。客户端通过向服务端发送消息提交请求，如果服务端需要回复，则会发送另一条独立的消息给客户端。由于异步通信，客户端不会因为等待而阻塞，相反会认为响应不会被立即收到。

**消息通过渠道发送，通过渠道接收。**

消息由数据头（例如发送方这样的元数据）和消息正文构成。消息通过渠道发送，任何数量的生产者都可以发送消息到渠道，同样，任何数量的消费者都可以从渠道中接受数据。**频道有两类**，包括**点对点渠道和请求\响应渠道**。**点对点**渠道会把**消息准确的发送到从渠道读取消息的用户**，服务端使用点对点来实现之前提到的一对一交互模式；而**发布/订阅则把消息投送到所有从渠道读取数据的用户**，服务端使用发布/订阅渠道来实现上面提到的一对多交互模式。

下图展示了打车软件如何使用发布/订阅：

[](http://blog.daocloud.io/wp-content/uploads/2016/05/Richardson-microservices-part3-pub-sub-channels-1024x639-4.png)

通过向**发布/订阅渠道写入一条创建行程的消息**，行程管理服务会通知调度服务有新的行程请求。调度服务发现可用的司机后会向**发布/订阅渠道写入一条推荐司机的消息**，并通知其它服务。

有多种消息系统可供选择，最好选择支持多编程语言的。有的消息系统支持 AMQP 和 STOMP 这样的标准协议，有的则支持专利协议。也有大量的开源消息系统可用，譬如 RabbitMQ、Apache Kafka、Apache ActiveMQ 和 NSQ。宏观上，它们都支持一些消息和渠道格式，并且努力提升可靠性、高性能和可扩展性。然而，细节上，它们的消息模型却大相径庭。

**使用消息机制有很多优点：**

* **解耦客户端和服务端**：客户端只需要将消息发送到正确的渠道。客户端完全不需要了解具体的服务实例，更不需要一个发现机制来确定服务实例的位置。
* **消息缓冲**：在 HTTP 这样的同步请求/响应协议中，所有的客户端和服务端必须在交互期间保持可用。而在**消息模式中，消息中间人将所有写入渠道的消息按照队列方式管理，直到被消费者处理**。也就是说，在线商店可以接受客户订单，即使下单系统很慢或者不可用，只要保持下单消息进入队列就好了。
* 客户端-服务端的灵活交互：消息机制支持以上说的所有交互模式。
* 清晰的进程间通信：基于 RPC 的通信机制试图让唤醒远程服务端像调用本地服务一样，然而，囿于物理定律和可能的局部失败，这二者大不相同。消息机制能让这些差异直观明确，开发者不会产生安全错觉。

**然而，消息机制也有自己的缺点：**

* 额外的操作复杂性：消息系统需要单独安装、配置和部署。消息broker（代理）必须高可用，否则系统可靠性将会受到影响。
* 实现基于请求/响应交互模式的复杂性：请求/响应交互模式需要完成额外的工作。每个请求消息必须包含一个回复渠道 ID 和相关 ID。服务端发送一个包含相关 ID 的响应消息到渠道中，使用相关 ID 来将响应对应到发出请求的客户端。这种情况下，使用一个直接支持请求/响应的 IPC 机制会更容易些。

现在我们已经了解了**基于消息的 IPC（渠道）**，接下来我们来看看基于请求/响应模式的 IPC。

**基于请求/响应的同步 IPC**

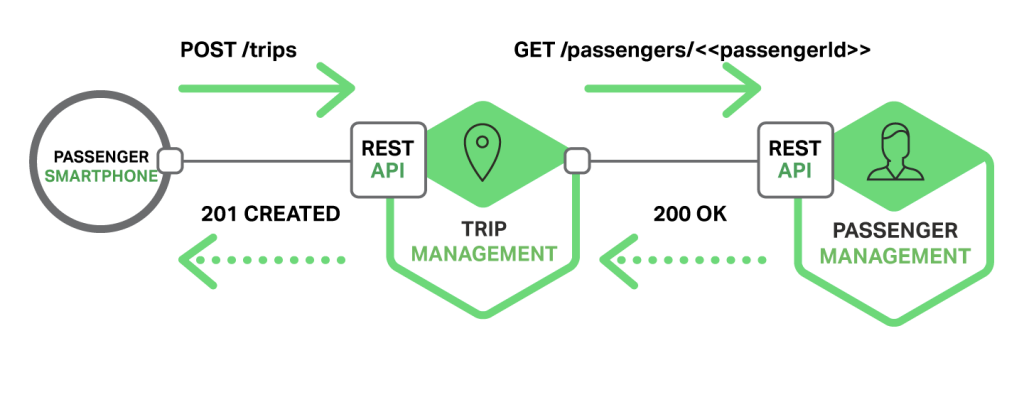
使用同步的、基于请求/响应的 IPC 机制的时候，**客户端向服务端发送请求，服务端处理请求并返回响应。一些客户端会由于等待服务端响应而被阻塞，**而另外一些客户端可能使用异步的、基于事件驱动的客户端代码，这些代码可能通过 Future 或者 Rx Observable 封装。然而，与使用消息机制不同，客户端需要响应及时返回。这个模式中有很多可选的协议，但最常见的**两个协议是 REST 和 Thrift。**首先我们来了解 REST。

**REST（其实就是我们平时使用的）**

**当前很流行开发 RESTful 风格的 A**PI。REST 基于 HTTP 协议，其核心概念是资源典型地代表单一业务对象或者一组业务对象，业务对象包括“消费者”或“产品”**。REST 使用 HTTP 协议来控制资源，通过 URL 实现。譬如，GET 请求会返回一个资源的包含信息，可能是 XML 文档或 JSON 对象格式。POST 请求会创建新资源，而 PUT 请求则会更新资源。**REST 之父 Roy Fielding 曾经说过：

REST 提供了一系列架构系统参数，作为整体使用，强调组件交互的扩展性、接口的通用性、组件的独立部署、以及减少交互延迟的中间件，它强化安全，也能封装遗留系统。

— Fielding, Architectural Styles and the Design of Network-based Software Architectures  
下图展示了打车软件如何使用 REST。

[](http://blog.daocloud.io/wp-content/uploads/2016/05/Richardson-microservices-part3-rest-1024x397-5.png)

乘客通过移动端向行程管理服务的 /trips 资源提交了一个 POST请求。行程管理服务收到请求之后，会发送一个 GET 请求到乘客管理服务以获取乘客信息。当确认乘客信息之后，随即创建一个行程，并向移动端返回 201 响应。

很多开发者都表示他们基于 HTTP 的 API 是 RESTful 风格。但是，如同 Fielding 在他的博客中所说，并非所有这些 API 都是 RESTful。Leonard Richardson（注：与本文作者 Chris 无任何关系）为 REST 定义了一个成熟度模型，具体包含以下四个层次：

* Level 0：本层级的 Web 服务只是使用 HTTP 作为传输方式，实际上只是远程方法调用（RPC）的一种具体形式。SOAP 和 XML-RPC 都属于此类。
* Level 1：Level 1 层级的 API 引入了资源的概念。要执行对资源的操作，客户端发出指定要执行的操作和任何参数的 POST 请求。
* Level 2：Level 2 层级的 API 使用 HTTP 语法来执行操作，譬如 GET 表示获取、POST 表示创建、PUT 表示更新。如有必要，请求参数和主体指定操作的参数。这能够让服务影响 web 基础设施服务，如缓存 GET 请求。
* Level 3：Level 3 层级的 API 基于 HATEOAS（Hypertext As The Engine Of Application State）原则设计，基本思想是在由 GET请求返回的资源信息中包含链接，这些链接能够执行该资源允许的操作。例如，客户端通过订单资源中包含的链接取消某一订单，GET 请求被发送去获取该订单。HATEOAS 的优点包括无需在客户端代码中写入硬链接的 URL。此外，由于资源信息中包含可允许操作的链接，客户端无需猜测在资源的当前状态下执行何种操作。

使用基于 HTTP 的协议有如下好处：

* **HTTP 非常简单并且大家都很熟悉。**
* **可以使用浏览器扩展（比如 Postman）或者 curl 之类的命令行来测试 API。**
* **内置支持请求/响应模式的通信。**
* **HTTP 对防火墙友好。**
* 不需要中间代理，简化了系统架构。

不足之处包括：

* 只支持请求/响应模式交互。尽管可以使用 HTTP 通知，但是服务端必须一直发送 HTTP 响应。
* 由于客户端和服务端直接通信（没有代理或者缓冲机制），在交互期间必须都保持在线。
* 客户端必须知道每个服务实例的 URL。如前篇文章“API 网关”所述，这也是个烦人的问题。客户端必须使用服务实例发现机制。

开发者社区最近重新认识到了 RESTful API 接口定义语言的价值，于是诞生了包括 RAML 和 Swagger 在内的服务框架。Swagger 这样的 IDL 允许定义请求和响应消息的格式，而 RAML 允许使用 JSON Schema 这种独立的规范。对于描述 API，IDL 通常都有工具从接口定义中生成客户端存根和服务端框架。

**Thrift**

Apache Thrift 是一个很有趣的 REST 的替代品，实现了多语言 RPC 客户端和服务端调用。Thrift 提供了一个 C 风格的 IDL 定义 API。通过 Thrift 编译器能够生成客户端存根和服务端框架。编译器可以生成多种语言的代码，包括 C++、Java、Python、PHP、Ruby, Erlang 和 Node.js。

Thrift 接口由一个或多个服务组成，服务定义与 Java 接口类似，是一组强类型方法的集合。Thrift 能够返回（可能无效）值，也可以被定义为单向。返回值的方法能够实现交互的请求/响应模式。客户端等待响应，可能会抛出异常。单向方法与交互的通知模式相对应。服务端不会发送响应。

Thrift 支持 JSON、二进制和压缩二进制等多种消息格式。由于解码更快，二进制比 JSON 更高效；如名称所称，压缩二进制格式可以提供更高级别的压缩效率；同时 JSON 则易读。Thrift 也能够让你选择传输协议，包括原始 TCP 和 HTTP。原始 TCP 比 HTTP 更高效，然而 HTTP 对于防火墙、浏览器和使用者来说更友好。

**消息格式**

了解 HTTP 和 Thrift 后，我们要考虑消息格式的问题。如果使用消息系统或者 REST，就需要选择消息格式。像 Thrift 这样的 IPC 机制可能只支持少量消息格式，或许只支持一种格式。无论哪种情况，使用跨语言的消息格式非常重要。即便你现在使用单一语言实现微服务，但很有可能未来需要用到其它语言。

目前有文本和二进制这两种主要的消息格式。文本格式包括 JSON 和 XML。这种格式的优点在于不仅可读，而且是自描述的。在 JSON 中，对象的属性是名称-值对的集合。与此类似，在 XML 中，属性则表示为命名的元素和值。消费者能够从中选择感兴趣的值同时忽略其它部分。相应地，对消息格式的小幅度修改也能容易地向后兼容。

XML 的文档结构由 XML schema 定义。随着时间发展，开发者社区意识到 JSON 也需要一个类似的机制。方法之一是使用 JSON Schema，要么独立使用，要么作为 Swagger 这类 IDL 的一部分。

文本消息格式的一大缺点是消息会变得冗长，特别是 XML。由于消息是自描述的，所以每个消息都包含属性和值。另外一个缺点是解析文本的负担过大。所以，你可能需要考虑使用二进制格式。

二进制的格式也有很多。如果使用的是 Thrift RPC，那可以使用二进制 Thrift。如果选择消息格式，常用的还包括 Protocol Buffers 和 Apache Avro，二者都提供类型 IDL 来定义消息结构。差异之处在于 Protocol Buffers 使用添加标记的字段（tagged fields），而 Avro 消费者需要了解模式来解析消息。

Martin Kleppmann 的[博客文章](http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html) 对 Thrift、Protocol Buffers 和 Avor 进行了详细的比较。

**总结**

微服务必须使用进程间通信机制来交互。在设计服务的通信模式时，你需要考虑几个问题：服务如何交互，每个服务如何标识 API，如何升级 API，以及如何处理局部失败。**微服务架构可用异步消息机制和同步请求/响应机制这两类 IPC 机制进行通信设计**（其实在我学习或者写项目的时候或多或少都有用到，比如rest请求，通知，同步的一对一的请求响应模式。还有异步消息队列，生产后，其他的地方去消费，其实看起来，也是类似于这样的机制了）。