介绍

这是一份针对网络 API 的通用设计指南。它从 2014 年开始用于 Google 内部，也用于设计 Cloud APIs 和 Google APIs。这份指南在此处共享来告知外部开发者，以便我们所有人都能更轻松地合作。

Google Cloud Endpoints 的开发者们可能会发现这份指南在设计 gRPC APIs 的时候特别有用，并且我们强烈推荐这些开发者们使用这些设计准则。但是我们并不强迫你使用它。你也可以使用 Cloud Endpoints 和 gRPC 却不遵循这份指南。

这份指导可同时应用于基于 HTTP 的 REST APIs 与基于 socket 的 RPC APIs，尤其是 gRPC APIs。gRPC APIs 使用 Protocol Buffers 来定义它们的 API 接口，使用 API Service Configuration 来定义他们的 API 服务，包括 HTTP 映射，日志和监控。HTTP 映射这个特性被 Google APIs 和 Cloud Endpoints gRPC APIs 使用，用于 JSON/HTTP 和 Protocol Buffers/RPC 之间的转码。

这份指南是一个活跃的文档，它的内容会随着新风格和新设计模式的采用和批准而有所增补。本着这种精神，它永远不会是完整的，并且 API 设计的艺术和工艺总是有足够的进步空间。

面向资源设计

这份设计指导的目标是帮助开发者设计出简洁、一致并且易于使用的网络 API。

在过去，人们参考像 CORBA 和 Windows COM 这样的 API 接口和方法来设计 RPC APIs。随着时间推移，越来越多的接口和方法被引入。这就导致了接口和方法的泛滥，并且他们的设计各异，互不相同。开发者们必须小心翼翼地学习每一个接口才能正确使用它，这不仅导致了时间的浪费，还容易出错。

REST 的架构风格在 2000 年首次被引入，与 HTTP/1.1 有着良好的相性。他的核心原则在于定义一些可以被少量方法操作的资源。毫无疑问资源是个名词，方法是个动词。在 HTTP 协议中，资源名称很自然地被映射到 URLs 上，方法则被映射成了 HTTP 的 POST、GET、PUT 和 DELETE 四种方法。

在互联网上，HTTP REST APIs 毫无疑问获得了巨大的成功，在 2010 年，有大约 74%的开放网络 API 采用了 REST 设计。

虽然 HTTP REST APIs 在互联网上非常流行，但是他们承载的流量还是比传统的 RPC APIs 要少。举个例子，在高峰期，大约一半的互联网流量来自视频内容，而出于性能原因，几乎不会有人会考虑用 REST API 去传输这样的内容。在数据中心内部，很多的公司使用基于 socket 的 RPC APIs 去承载大部分的网络流量，这比开放的 REST APIs 可能要多好几个数量级。

现实点来说，RPC APIs 和 HTTP REST APIs 因为不同的原因都不可或缺。在理想情况下，一个 API 平台应该为所有 APIs 提供最佳支持。这份设计指导将帮助你设计与构建遵从这个原则的 APIs，并且会定义一些共用的设计模式以提高可用性并降低复杂度。

提示：这份文档解释了怎样将 REST 原则运用于 API 设计，并且这与编程语言、操作系统及网络协议无关。这不是一个教你怎样创建 REST APIs 的文档。

什么是 REST API？

REST API 被建模为可以被单独寻址的资源（API 的名词）的集合。资源可以被他们的资源名称引用，并通过一小组方法（也称为动词或操作）来进行操作。

REST Google APIs 的标准方法（也被称为 REST 方法）是 List, Get, Create, Update 和 Delete。自定义方法（也被称为自定义动词或自定义操作）也可供 API 设计人员使用，以应对那些不易映射到任何一种标准方法的功能，例如数据库事务。

提示：自定义动词并不意味着创建自定义的 HTTP 动词来支持自定义方法。对于基于 HTTP 的 API，它们只是映射到最合适的 HTTP 动词。

设计流程

这份设计指南建议设计面向资源的 APIs 的时候按照以下几个步骤（以下具体的章节涵盖了更多的细节）：

- 确定 API 所提供资源的类型
- 确定资源之间的关系
- 基于类型和关系决定资源名称的结构
- 决定资源的结构
- 为资源附加上方法的最小集合

资源

一个面向资源的 API 通常被建模为资源层级，其中每个节点都是一个简单资源或是一个集合资源。为了方便起见，它们通常分别被称为资源和集合。

- 集合包含相同类型的资源列表。例如，用户拥有一个联系人集合。
- 资源有一些状态和零个或多个子资源。每个子资源可以是简单资源或集合资源。

例如，Gmail API 包含一个用户集合，每个用户都有一个消息集合，一个邮件线集合，一个标签集合，一个个人资料资源以及多个设置资源。

尽管存储系统和 REST API 之间存在一些概念上的一致性，但具有面向资源的 API 的服务不一定是数据库，并且在解释资源和方法方面具有极大的灵活性。例如，创建一个日历事件（资源）可能为与会者创建其他事件，向与会者发送电子邮件邀请，预定会议室以及更新视频会议日程安排。

方法

一个面向资源的 API 的关键特征是它强调资源（数据模型）而不是在资源上执行的方法（功能）。一个典型的面向资源的 API 会暴露大量携带少量方法的资源。这些方法可以是标准方法或自定义方法。对于本指南而言，标准方法是：List，Get，Create，Update 和 Delete。如果 API 功能可以自然地映射到其中一种标准方法，那么应该在 API 设计中使用该方法。对于不能自然地映射到其中一种标准方法的功能，可以使用自定义方法。自定义方法提供与传统 RPC API 相同的设计自由度，可用于实现常见编程模式，如数据库事务或数据分析。

例子

以下部分介绍了一些关于如何将面向资源的 API 设计应用于大规模服务的现实世界示例。你可以在 Google API 仓库中找到更多示例。

Gmail API

Gmail API 服务实现了 Gmail API 并暴露了大量的 Gmail 功能。 它有以下资源模型：

- API 服务：gmail.googleapis.com
- 一个用户集合：users/*。每个用户都有以下资源。
  - 消息集合：users/*/messages/*。
  - 邮件线集合：users/*/threads/*。
  - 标签集合：users/*/labels/*。
  - 更改历史记录的集合：users/*/history/*。
  - 表示用户个人资料的资源：users/*/profile。
  - 表示用户设置的资源：users/*/settings。

Cloud Pub/Sub API

pubsub.googleapis.com 服务实现了 Cloud Pub/Sub API，该 API 定义了以下资源模型：

- API 服务：pubsub.googleapis.com
- 主题集合：projects/*/topics/*。
- 订阅集合：projects/*/subscriptions/*。

注意：Pub/Sub API 的其他实现可能会选择不同的资源命名方案。

Cloud Spanner API

spanner.googleapis.com 服务实现了 Cloud Spanner API，该 API 定义了以下资源模型：

- API 服务：spanner.googleapis.com
- 一个实例的集合：projects/*/instances/*。
  - 实例操作的集合：projects/*/instances/*/operations/*。

- 数据库的集合：projects/*/instances/*/databases/*。
- 数据库操作的集合：projects/*/instances/*/databases/*/operations/*。
- 数据库会话的集合：projects/*/instances/*/databases/*/sessions/*。

资源名称

在面向资源的 APIs 里，资源是命名实体，并且资源名称是他们的唯一标识。每一个资源必须有一个独一无二的资源名称。资源名称由资源本身的 ID、它的各个父级的 ID 以及 API 服务名称组成。接下来我们会关注在资源 ID 和一个资源名称是如何被构建的。

gRPC APIs 应当使用 scheme-less URIs 来描述资源名称。他们通常遵循 REST URL 的约定，并且表现的更像是一个网络上的文件路径。他们可以被轻易地映射到 REST URLs：查阅标准方法获取更多细节。

集合是一种特殊的资源，它包含了一组相同类型的子资源。举个例子，一个文件夹是一组文件资源的集合。一个集合的资源 ID 被称为集合 ID。

资源名称被有层级地按照集合 IDs 和资源 IDs 组织，通过斜杆分隔。如果一个资源包含一个子资源，那么子资源的名称将形如父级资源的名称尾随着一个子资源 ID，他们同样是由斜杆来分隔。

例子 1：一个存储服务有一组 buckets 的集合，而 bucket 又是一组 objects 的集合：

API Service Name	Collection ID	Resource ID	Collection ID	Resource ID
//storage.googleapis.com	/buckets	/bucket-id	/objects	/object-id

例子 2：一个邮件服务有一组用户的集合，每一个用户有一个 settings 子资源，并且 settings 子资源又有复数的子资源，比如 customFrom：

API Service Name	Collection ID	Resource ID	Resource ID	Resource ID
//mail.googleapis.com	/users	/name@example.com	/settings	/customFrom

一个 API 开发者可以为资源和集合 IDs 选择任意可接受的值，并且需要在资源层级内尽可能长来保证它们是唯一的。你可以在下文中找到更多关于如何选择合适的资源和集合 IDs 的指导方针。 通过分隔资源名称，比如 name.split(“/“)[n]，只要各个分段都不包含任何的正斜杆，就可以分别获得集合 IDs 和资源 IDs。

完整的资源名称

一个 scheme-less URI 由一个兼容 DNS 的 API 服务名称和一个资源路径组成。资源路径也叫相对资源名称。举个例子：

//library.googleapis.com/shelves/shelf1/books/book2

API 服务名称是为了让客户端可以定位 API 服务的终端，它可以是一个内部服务专用的伪 DNS 名。如果 API 服务的名称在正下文中是明确的，那么相对资源名称也经常被使用。

相对资源名称

指的是一个不以“/”开头的 URI 路径。它标识了一个 API 服务内的资源。举个例子：

"shelves/shelf1/books/book2"

资源 ID

一个不为空的 URI 分段，它标识了一个被包含在它的父资源内的资源，可以参照之前的例子。 在资源名称中，资源 ID 可以是多个 URI 的分段。举个例子：

Collection ID	Resource ID
files	/source/py/parser.py

如果条件允许，API 服务应该使用 URL 友好的资源 IDs。资源 IDs 必须被清晰地标明它们是由客户端还是服务端来指定。举个例子，文件名通常是由客户端来指定的，而邮件的消息 IDs 则通常由服务端来指定。

集合 ID

一个不为空的 URI 分段，它标识了被包含在它的父资源内集合资源，可以参照之前的例子。由于集合 IDs 通常会出现在被生成的客户端库里，所以它们必须满足以下的需求：

- 必须是合法的 C/C++标识符
- 必须是小写驼峰的复数形式。如果没有合适的复数形式，例如“evidence”或“weather”，那么应该使用单数形式。
- 必须是清晰且简明的英文词汇
- 过于概略的词汇应该避免使用，或者加以修饰。例如 rowValues 就比 values 更好。以下的词汇如果没有修饰应该被避免使用：
    - elements
    - entries
    - instances
    - items
    - objects
    - resources
    - types
    - values

资源名称 vs URL

虽然一个完整的资源名称看起来像是一个普通的 URLs，但是它们其实并不相同。一个单独的资源可以被不同的方式暴露，例如不同的 API 版本，不同的 API 协议或者不同的网络终端。完整的资源名称并未指明这些信息，所以当它被使用时必须被映射到指定的 API 版本和 API 协议。

要通过 REST API 使用完整的资源名称，必须将其转换为 REST URL，方法是在服务名称前添加 HTTPS 方案，在资源路径前添加 API 主版本，以及 URL 转义资源路径。例如：

```
// 这是一个日历事件的资源名称。
"//calendar.googleapis.com/users/john smith/events/123"

// 这是相应的 HTTP URL。
"https://calendar.googleapis.com/v3/users/john%20smith/events/1
```

资源名称即字符串

资源名称必须是字符串，除非有无法处理的向后兼容问题。资源名称应该像普通的文件路径那样被处理，并且不支持 URL encoding。

对于资源的定义，第一个字段应该是一个字符串，表示资源名称，并且它应该被命名为 name。提示：其他与 name 相关的字段应该加以修饰来消除歧义，例如 display_name，first_name，last_name，full_name。

例如：

```
service LibraryService {
  rpc GetBook(GetBookRequest) returns (Book) {
    option (google.api.http) = {
      get: "/v1/{name=shelves/*/books/*}"
    };
  };
  rpc CreateBook(CreateBookRequest) returns (Book) {
    option (google.api.http) = {
      post: "/v1/{parent=shelves/*}/books"
      body: "book"
    };
  };
```

```
}

message Book {
  // Resource name of the book. It must have the format of "shelves/*/books/*".
  // For example: "shelves/shelf1/books/book2".
  string name = 1;

  // ... other properties
}

message GetBookRequest {
  // Resource name of a book. For example: "shelves/shelf1/books/book2".
  string name = 1;
}

message CreateBookRequest {
  // Resource name of the parent resource where to create the book.
  // For example: "shelves/shelf1".
  string parent = 1;
  // The Book resource to be created. Client must not set the `Book.name` field.
  Book book = 2;
}
```

提示：为了资源名称的一致性，词首的正斜杆一定不能被任何 URL 模版变量捕获。例如，URL 模版必须使用/v1/{name=shelves/*/books/*}而不是"/v1{name=/shelves/*/books/*}"

问题

Q：为什么不使用资源 IDs 来标识一个资源？

A：在一个大型系统里有种类繁多的资源。使用资源 IDs 去标识一个资源，我们事实上是在使用一个指定资源的元组去标识一个资源，例如(bucket, object)或者(user, album, photo)。它会导致一些主要的麻烦：

- 开发者必须去理解并且记住这些匿名元组
- 传递元组获取通常比传递字符串更难
- 中心化的基础设施，例如日志和访问权限控制系统，并不能理解指定的元组
- 特定的元组限制了 API 设计的灵活性，例如提供可重用的 API 接口。

Q：为什么要把特殊字段命名为 name 而不是 id？

A：特殊字段是以资源“名称”的概念命名的。通常，我们发现 name 这个词的含义经常困扰开发者。例如，文件名真的只是名字，还是一个完整路径呢？通过预定这个标准字段 name，开发者被强制选择一个更恰当的词汇，例如 display_name 或者 title 或者 full_name。

标准方法

这一章定义了标准方法的概念：List，Get，Create，Update 和 Delete。标准方法既减少了复杂度又增加了一致性。在 Google APIs 中超过 70%的 API 方法是标准方法，这能使得它们更易学易用。

以下这张表格描述了如何将标准方法映射到 HTTP 方法：

Standard Method	HTTP Mapping	HTTP Request Body	HTTP Response Body
List	GET <collection URL>	N/A	Resource* list
Get	GET <resource URL>	N/A	Resource*
Create	POST <collection URL>	Resource	Resource*

Update	PUT or PATCH <resource URL>	Resource	Resource*
Delete	DELETE <resource URL>	N/A	google.protobuf.Empty**

*如果方法支持响应字段掩码——用于指定需要被返回的字段子集的话，那通过这些方法返回的资源中可能会包含局部数组。在某些场合，API 平台天然支持所有方法的字段掩码。
**从并不立即删除资源的 Delete 方法返回的响应应该包含 long running operation 或者被修改的资源之一
一个标准方法也可能会对在一次 API 调用的时间跨度内未完成的请求返回 long running operation。
以下各节详细介绍了每种标准方法。这些示例显示了在.proto 文件中通过特殊注释定义 HTTP 映射的方法。你可以在 Google APIs 仓库中找到许多使用标准方法的示例。

List

List 方法携带一个集合名称和若干个参数作为输入，并且返回一组匹配输入的资源列表。
List 通常用于搜索资源。 List 适合于来自单个集合的数据，该集合的大小有限且未被缓存。对于更广泛的情况，应该使用自定义方法 Search。
批量获取（该方法可以携带多个资源 ID 并且返回对应的每一个对象）应该被实现为自定义方法 BatchGet，而不是使用 List。但是，如果你的 List 方法已经提供了类似的功能，那么你可能可以复用 List 来实现这个意图。如果你正在使用自定义方法 BatchGet，它应该被映射到 HTTP GET。
适合的通用模式：分页，结果排序。 适合的命名约定：过滤字段，结果字段。
HTTP 映射：

- List 方法必须使用 HTTPGET 动词
- 请求消息中用于接受被获取资源的集合名称的字段应该映射到 URL 路径中。如果集合名称映射到 URL 路径，则 URL 模板的最后一部分（集合 ID）必须是文字。
- 所有剩余的请求消息字段应该映射到 URL query parameters。
- 没有请求体；API 设置中禁止声明 body 子句。
- 响应体应该包含一个资源列表以及可选的元数据。

例子：

```
// Lists books in a shelf.
rpc ListBooks(ListBooksRequest) returns (ListBooksResponse) {
  // List method maps to HTTP GET.
  option (google.api.http) = {
    // The `parent` captures the parent resource name, such as "shelves/shelf1".
    get: "/v1/{parent=shelves/*}/books"
  };
}

message ListBooksRequest {
  // The parent resource name, for example, "shelves/shelf1".
  string parent = 1;
  // The maximum number of items to return.
  int32 page_size = 2;
  // The next_page_token value returned from a previous List request, if any.
  string page_token = 3;
}

message ListBooksResponse {
  // The field name should match the noun "books" in the method name. There
  // will be a maximum number of items returned based on the page_size field
```

```
  // in the request.
  repeated Book books = 1;
  // Token to retrieve the next page of results, or empty if there are no
  // more results in the list.
  string next_page_token = 2;
}
```

Get

Get 方法携带一个资源名称和若干个参数，返回指定的资源。

HTTP 映射：

- Get 方法必须使用 HTTPGET 动词
- 请求消息中用于接受被获取资源的集合名称的字段应该映射到 URL 路径中。
- 所有剩余的请求消息字段应该映射到 URL query parameters。
- 没有请求体；API 设置中禁止声明 body 子句。
- 返回的资源应该映射到整个响应体。

例子：

```
// Gets a book.
rpc GetBook(GetBookRequest) returns (Book) {
  // Get maps to HTTP GET. Resource name is mapped to the URL. No body.
  option (google.api.http) = {
    // Note the URL template variable which captures the multi-segment resource
    // name of the requested book, such as "shelves/shelf1/books/book2"
    get: "/v1/{name=shelves/*/books/*}"
  };
}

message GetBookRequest {
  // The field will contain name of the resource requested, for example:
  // "shelves/shelf1/books/book2"
  string name = 1;
}
```

Create

Create 方法携带一个父资源名称，一个资源名称及若干个参数。它在指定的父级下创建一个新的资源，并返回该新创建的资源。

如果一个 API 支持创建资源，它应该对所有允许创建的资源提供一个 Create 方法。

HTTP 映射：

- Create 方法必须使用 HTTP POST 动词
- 请求消息中应该有一个字段 parent 用于指定将被创建资源的父级资源名称。
- 所有剩余的请求消息字段应该映射到 URL query parameters。
- 请求可能会包含一个叫做<resource>_id 的字段用于允许调用者选择一个客户端自分配 id。这个字段必须映射到 URL query parameters。
- 包含资源的请求消息字段应该映射到请求体。如果在 Create 方法中使用了 body 的 HTTP 配置子句，必须依照 body: "<resource_field>"的格式。
- 返回的资源应该映射到整个响应体。

如果 Create 方法支持客户端指定资源名称并且该资源已经存在，则该请求应该失败并返回错误码 ALREADY_EXISTS，也可以由服务端指定一个不同的资源名称，不过必须在文档中清晰地标示出：被创建的资源名称有可能和输入的不同。

Create 方法必须以资源作为输入，因为这样如果资源的结构被更改了，也没有必要同时修改请求的结构和资源的结构。对于无法在客户端修改的资源字段，必须在文档里标注为

“Output only”。
举个例子：

```
// Creates a book in a shelf.
rpc CreateBook(CreateBookRequest) returns (Book) {
  // Create maps to HTTP POST. URL path as the collection name.
  // HTTP request body contains the resource.
  option (google.api.http) = {
    // The `parent` captures the parent resource name, such as "shelves/1".
    post: "/v1/{parent=shelves/*}/books"
    body: "book"
  };
}

message CreateBookRequest {
  // The parent resource name where the book to be created.
  string parent = 1;

  // The book id to use for this book.
  string book_id = 3;

  // The book resource to create.
  // The field name should match the Noun in the method name.
  Book book = 2;
}

rpc CreateShelf(CreateShelfRequest) returns (Shelf) {
  option (google.api.http) = {
    post: "/v1/shelves"
    body: "shelf"
  };
}

message CreateShelfRequest {
  Shelf shelf = 1;
}
```

Update

Update 方法发起一个包含了资源和若干个参数的请求。它可以用于更新置顶的资源和它的属性，并返回更新后的资源。
可变的资源属性应该通过 Update 方法来更改，除非这个属性包含了资源名称或者父级信息。任何的重命名或移动一个资源的动作都不应该在 Update 里发生，应该由自定义方法来处理。
HTTP 映射：

- Update 方法应该支持部分资源更新，使用 HTTPPATCH 动词以及一个叫做 update_mask 的 FieldMask。
- 需要更高阶的修补语义的 Update 方法，比如添加数据到一个重复字段，应该通过自定义方法来实现。
- 如果 Update 方法只支持全量资源更新，它必须使用 HTTP 动词 PUT。但是全量更新是非常令人沮丧的，因为它在增加新的资源字段的时候有向后兼容问题。
- 请求消息中用于接受被获取资源的集合名称的字段应该映射到 URL 路径中。这个字段也可能在资源消息本身里。
- 包含资源的请求消息字段必须映射到请求体。

- 所有剩余的请求消息字段必须映射到 URL query parameters。
- 响应消息必须是被更新后的资源本身。

如果 API 允许客户端指定资源名称，服务器可能会允许客户端来指定一个不存在的资源名称并自动创建一个新的资源。而如果服务器不提供这个能力，那么当资源名称不存在时，Update 方法应该失败。如果仅有这一个错误，那么应该使用错误码 NOT_FOUND。
提供 Update 方法并支持资源创建的 API，也应该同时提供 Create 方法。根本原因在于如果 Update 方法是唯一的方式，那人们可能不会很清楚地知道该如何创建资源。
例子：

```
// Updates a book.
rpc UpdateBook(UpdateBookRequest) returns (Book) {
  // Update maps to HTTP PATCH. Resource name is mapped to a URL path.
  // Resource is contained in the HTTP request body.
  option (google.api.http) = {
    // Note the URL template variable which captures the resource name of the
    // book to update.
    patch: "/v1/{book.name=shelves/*/books/*}"
    body: "book"
  };
}

message UpdateBookRequest {
  // The book resource which replaces the resource on the server.
  Book book = 1;

  // The update mask applies to the resource. For the `FieldMask` definition,
  // see https://developers.google.com/protocol-buffers/docs/reference/google.
protobuf#fieldmask
  FieldMask update_mask = 2;
}
```

Delete

Delete 方法携带一个资源名称及若干个参数，来立刻删除或在某个特定时间删除指定的资源。Delete 方法应该返回 google.protobuf.Empty。
API 不应该依赖于任何由 Delete 方法返回的信息，因为它不能被重复调用。
HTTP 映射：

- Delete 方法必须使用 HTTPDELETE 动词
- 请求消息中用于接受被获取资源的集合名称的字段应该映射到 URL 路径中。
- 所有剩余的请求消息字段应该映射到 URL query parameters。
- 没有请求体；API 设置中禁止声明 body 子句。
- 如果 Delete 方法立刻移除了资源，它应该返回一个空响应。
- 如果 Delete 方法开始了一个 long-running operation，它应该返回一个 long-running operation。
- 如果 Delete 方法只是将资源标记为已删除，它应该返回更新后的资源。

调用 Delete 方法应该是幂等的，但不需要产生相同的响应，任何数量的删除请求都应该导致一个资源（最终）被删除，但只有第一个请求应该成功。随后的请求应该产生一个 google.rpc.Code.NOT_FOUND。
例子：

```
// Deletes a book.
rpc DeleteBook(DeleteBookRequest) returns (google.protobuf.Empty) {
  // Delete maps to HTTP DELETE. Resource name maps to the URL path.
  // There is no request body.
```

```
  option (google.api.http) = {
    // Note the URL template variable capturing the multi-segment name of the
    // book resource to be deleted, such as "shelves/shelf1/books/book2"
    delete: "/v1/{name=shelves/*/books/*}"
  };
}

message DeleteBookRequest {
  // The resource name of the book to be deleted, for example:
  // "shelves/shelf1/books/book2"
  string name = 1;
}
```

自定义方法

这一个章节将会讨论如何在 API 设计中使用自定义方法。
自定义方法指的是除了五种标准方法以外的 API 方法。他们只会用于那些无法轻易通过标准方法来表达的功能上。一般来说，API 设计者应该在条件允许的情况下尽可能地使用标准方法。标准方法有着更简单且定义明确的语义，更为大部分开发人员所熟知，所以他们更容易被使用且更不容易出错。标准方法的另一个优势是 API 平台对标准方法有着更好的支持，例如错误处理、日志、监控等。
一个自定义方法可以与资源、集合或服务相关联。它可能携带任意请求、返回任意响应，也可能支持流式请求和流式响应。
自定义方法名必须遵守方法命名约定。

HTTP 映射

对于自定义方法，它们应该使用如下通用 HTTP 映射：
https://service.name/v1/some/resource/name:customVerb
使用:代替'/'来将自定义动词从资源名称中分离是为了支持任意的路径。举个例子，撤销删除一个文件可以映射到 POST /files/a/long/file/name:undelete
选择 HTTP 映射时应遵循以下准则：

- 自定义方法应该使用 HTTPPOST 动词，因为它具有最灵活的语义，例外是用作 get 或 list 备选的方法可能会使用 GET。（有关详细信息，请参阅第三个准则。）
- 自定义方法不应该使用 HTTPPATCH，但可以使用其它 HTTP 动词。在这种情况下，这些方法必须遵循该动词的标准 HTTP 语义。
- 值得注意的是，使用 HTTPGET 的自定义方法必须是幂等的，并且没有副作用。例如，在资源上实现特殊视图的自定义方法应该使用 HTTPGET。
- 请求消息中接收与自定义方法相关联的资源或集合的资源名称的字段应该映射到 URL 路径。
- URL 路径必须以由冒号后跟自定义动词组成的后缀结尾。
- 如果用于自定义方法的 HTTP 动词允许 HTTP 请求体（POST，PUT，PATCH 或自定义 HTTP 动词），则此类自定义方法的 HTTP 配置必须使用 body: "*"子句，所有其余的请求消息字段应映射到 HTTP 请求体。
- 如果用于自定义方法的 HTTP 动词不接受 HTTP 请求体（GET，DELETE），则此类方法的 HTTP 配置一定不能使用 body 子句，并且所有剩余的请求消息字段都应映射到 URL query parameters。

警告：如果服务实现多个 API，则 API 生产者必须小心创建服务配置以避免 API 之间的自定义动词冲突。

```
// 这是一个服务级别的自定义方法.
rpc Watch(WatchRequest) returns (WatchResponse) {
  // 自定义方法映射到 HTTP POST。所有请求参数被放入 body。
```

```
  option (google.api.http) = {
    post: "/v1:watch"
    body: "*"
  };
}

// 这是一个集合级别的自定义方法。
rpc ClearEvents(ClearEventsRequest) returns (ClearEventsResponse) {
  option (google.api.http) = {
    post: "/v3/events:clear"
    body: "*"
  };
}

// 这是一个资源级别的自定义方法。
rpc CancelEvent(CancelEventRequest) returns (CancelEventResponse) {
  option (google.api.http) = {
    post: "/v3/{name=events/*}:cancel"
    body: "*"
  };
}

// 这是一个批量获取自定义方法
rpc BatchGetEvents(BatchGetEventsRequest) returns (BatchGetEventsResponse) {
  // 批量获取方法映射到 HTTP GET 动词。
  option (google.api.http) = {
    get: "/v3/events:batchGet"
  };
}
```

使用场合

在以下场景中使用自定义方法可能是个正确的选择：

- **重启一台虚拟机**该设计的替代方案可以是“创建一个在重启集合中的重启资源”，但感觉这样过于复杂，或者“虚拟机有一个可变状态，而客户端可以将其从 RUNNING 状态变为 RESTARTING”，而这会带来更多的问题，例如哪些状态之间是可以互相转移的。此外，重启是一个众所周知的概念，它可以很方便地被转化成一个满足开发者期望的自定义方法。
- **发送邮件**创建一个邮件消息不一定需要发送（草稿）。跟替代方案（移动一个消息到“Outbox”集合）对比起来，自定义方法的优势在于 API 的使用者更容易发现并更直接地建立起概念。
- **提拔一个雇员**。如果这个功能被实现为一个标准的 update，那客户端就必须重复实现企业政策中与升职流程相关的部分，来确保升职是作用在正确的等级，并且在相同的职业梯队内。
- **批量处理方法**对于性能关键的方法，提供自定义批处理方法以减少每个请求的开销可能很有用。

一些使用标准方法比自定义方法更合适的例子：

- 使用不同的查询参数来查询资源（使用标准的 list 方法与标准列表过滤）。
- 简单的资源字段更改（使用标准 update 方法与字段掩码）。
- 关闭通知（使用标准 delete 方法）。

通用自定义方法

如下所示是一些常用且有用的自定义方法合集。API 设计人员在引入他们自己的设计前应该优先考虑使用这些名称以促进跨 API 的一致性

Method Name	Custom verb	HTTP verb	Note
Cancel	:cancel	POST	取消一个长时操作(构建、计算等等)
BatchGet\	:batchGet	GET	批量获取多个资源。(在 List 的描述中获取更多细节)
Move	:move	POST	将一个资源从一个父级移动到另一个。
Search	:search	GET	获取不符合列表语义的数据列表的替代方案。
Undelete	:undelete	POST	还原一个先前删除的资源。推荐的保留时期为 30 天。

标准字段

这个章节描述了一系列定义近似概念时候应该被使用的标准消息字段。这可以用来保证跨不同 APIs 之间的相同概念有着相同的名称和语义。

Name	Type	Description
name	string	name 字段应该包含相对资源名。
parent	string	在资源定义和 List/Create 请求里，parent 应该包含其父级的相对资源名。
create_time	Timestamp	一个实体的创建时间戳。
update_time	Timestamp	一个实体最后一次被修改的时间戳. 提示: update_time 会在 create/patch/delete 操作发生作用时被更新。
delete_time	Timestamp	一个实体的被删除时间。只有支持还原操作时存在。
expire_time	Timestamp	一个实体的过期时间戳，如果它会过期的话。
start_time	Timestamp	标记某个时间段开始的时间戳。
end_time	Timestamp	标记某个时间段或操作结束的时间戳(不管成功与否)。
read_time	Timestamp	某个特定资源被获取（在请求中被使用）或者被读取（在响应中使用）的时间戳。
time_zone	string	时区名称. 它应该是一个 IANA TZ 名称，形如"America/Los_ Angeles"。更多信息，请查看 https://en.wikipedia.org /wiki/List_of_tz_database_time_zones。
region_code	string	一个地区的 Unicode country/region code (CLDR)，形如"US" and "419"。更多信息，请查看 http://www.unicode. org/reports/tr35/#unicode_region_subtag。
language_code	string	The BCP-47 language code，形如"en-US"或者"sr-Latn"。更多信息，请查看 http://www.unicode.org /reports/tr35/#Unicode_locale_identifier。
mime_type	string	一个 IANA 发布的 MIME 类型（也被称为媒体类型）。更多信息，请查看 https://www.iana.org/assignments/media-types/media-types.xhtml.
display_name	string	一个实体的显示名称。
title	string	一个实体的官方名称，例如公司名字。 它应该被视为 display_name 的正式版本。

description	string	一个实体的一段或多段文字描述。
filter	string	List 方法的标准过滤参数。
query	string	如果应用于搜索方法，则与过滤器相同（即：搜索）
page_token	string	List 请求中的分页标记。
page_size	int32	List 请求中的分页大小。
total_size	int32	不分页时列表中的项目总数。
next_page_token	string	列表响应中的下一个分页标记。 它应该作为 page_token 用于下一个请求。 一个空值意味着没有更多的结果。
order_by	string	指定 List 请求结果的排序规则。
request_id	string	一个用于检测重复请求的唯一字符串。
resume_token	string	用于恢复流媒体请求的不透明令牌。
labels	map<string, string>	代表云资源标签。
deleted	bool	如果某个资源允许取消删除行为，则必须有一个 deleted 的字段，指示该资源是否已被删除。
show_deleted	bool	如果某个资源允许取消删除行为，则对应的 List 方法必须有一个 show_deleted 的参数，这样客户端才能查看那些被删除的资源。
update_mask	FieldMask	它用于更新请求消息，以便对资源执行部分更新。 该掩码是相对于资源而不是请求消息。
validate_only	bool	如果为 true，则表示只想验证给定的请求，而并不实际执行。

错误

本章概述了 Google APIs 的错误模型，并且提供一个规范来指导开发者如何正确地生成和处理错误。

Google APIs 使用一个简单的和协议无关的错误模型，这允许我们可以在不同的 APIs，例如 gRPC 和 HTTP，以及不同的错误上下文（例如，异步，归并，和工作流错误）里面提供一致性的体验。

错误模型

Google APIs 的错误模型是由 google.rpc.Status 进行定义的，当 API 发生错误时一个 Status 的实例会被返回给调用者。以下这个代码片段展示了错误模型的总体设计思路：

```
package google.rpc;
message Status {
  // 一个容易被客户端进行处理的错误码。
  // 实际的错误码在 google.rpc.Code 里面定义
  int32 code = 1;

  // 面向开发人员的可读性高的英文错误信息
  // 这个错误信息应该同时说明错误的原因以及提供一个可操作的处理错误的方法
  string message = 2;

  // 额外的错误信息，这些错误信息可以被客户端代码用来处理这个错误，
  // 例如告诉客户端隔多长时间再次尝试或者提供一个帮助链接
  repeated google.protobuf.Any details = 3;
```

}
和大多数 Google APIs 的设计一样， 我们的错误处理同样遵循面向资源的设计原则。我们会使用一个较小的标准错误集合来对应一个较大的资源集合。举个例子，对于 NOT_FOUND 错误，我们不会给不同的资源定义不同的错误，而是统一给客户端返回一个带有标准 google.rpc.Code.NOT_FOUND 错误码以及指明特定的资源没有被找到的错误。因为我们的错误状态集合比较小，这样可以减少 API 文档的复杂性，同时客户端的代码也可以使用更加惯用的映射来减少其逻辑的复杂性而且还可以从服务器返回的错误里面获得一些可操作的信息。

错误码

Google APIs 一定要使用定义在 google.rpc.Code 的标准错误码。单独的 APIs 应该避免定义多余的错误码，因为开发者基本上不可能为一大堆错误码实现相关的处理逻辑。举个例子，如果平均每个 API 请求要处理的错误码个数是 3 个的话就会导致大多数的代码逻辑只为处理错误而存在的，这会使开发者的体验很不好。

错误信息

错误信息是用来帮助用户快速并容易地理解和解决 API 错误的。通常情况下，在书写错误信息是要遵循以下规范：

- 不要假使用户是你 API 的专家。用户可能是客户端开发者，运营人员，IT 工作人员或者是 APP 的终端用户。
- 不要假使用户知道你 API 的所有实现或者熟悉错误的上下文（例如日志分析）。
- 如果有可能，错误信息应该被构建为可以让技术人员（不一定是你 API 的开发者）对你的错误做出响应并且可以改正这个错误。
- 保持错误信息的简要性。如果有需要，为有疑惑的读者提供一个可以提问的，提供反馈的，或者获取更多不可以在错误信息中展示的信息的链接。如果做不到应该为用户提供一个可以展开的错误详情字段。

错误详情

Google APIs 在 google/rpc/error_details.proto 里面定义了一套错误详情的标准错误负载。他们涵盖了 API 错误的最通用的需求，例如配额错误（quota failure）和无效参数错误（invalid parameters）。和错误码一样，错误详情应该尽可能使用这些标准错误负载。额外的错误详情类型只有在可以帮助应用程序处理错误的时候才应该被引入。如果错误信息依赖于其内容并且只可以被人类来处理的话，应该让开发者手动处理该错误而不是引入新的错误详情类型。
以下是一些错误详情负载的例子：

- RetryInfo 用于描述什么时候客户端可以重试失败请求，可能会与错误码 Code.UNAVAILABLE 或者 Code.ABORTED 一起被返回。
- QuotaFailure 用于描述配额检查失败的原因，可能会与错误码 Code.RESOURCE_UNEXHAUSTED 一起被返回。
- BadRequest 用于描述客户端请求的违规情况，可能会与错误码 Code.INVALID_ARGUMENT 一起被返回。

HTTP 映射

虽然 proto3 消息有原生的 JSON 编码，但是 Google API 平台针对 Google 的 REST APIs 使用了不同的错误结构来做到向后兼容。
结构：

```
// Google REST APIs 的错误结构。备注：这个结构不是用在其他 wire protocol 的
message Error {
  // 该消息具有与 google.rpc.Status 相同的语义。它有一个额外的字段 status 字段，
  用于向后兼容 Google API 的客户端库
```

```
    message Status {
      // 这个对应着`google.rpc.Status.code`。
      int32 code = 1;
      // 这个对应着`google.rpc.Status.message`。
      string message = 2;
      // 这个是`google.rpc.Status.code`的枚举版本.
      google.rpc.Code status = 4;
      // 这个对应着`google.rpc.Status.details`。
      repeated google.protobuf.Any details = 5;
    }
    // 实际的错误负载。嵌套的消息结构是用来与 Google API 客户端进行向后兼容的。它同时也让错误对于开发者更加可读。
    Status error = 1;
}
```

例子：

```
{
  "error": {
    "code": 401,
    "message": "Request had invalid credentials.",
    "status": "UNAUTHENTICATED",
    "details": [{
      "@type": "type.googleapis.com/google.rpc.RetryInfo",
      ...
    }]
  }
}
```

RPC 映射

不同 RPC 协议映射错误模型的方法不一样。对于 gRPC 来说，错误模型被受支持的语言的生成代码以及运行时的库原生支持。你可以在 gRPC 的 API 文档里面获取更多的信息（例如，查看 gRPC Java 的 io.grpc.Status）。

客户端库映射

Google 客户端库可能会根据语言的不同来对错误进行不同的表示从而和该语言的既定习惯保持一致。例如，google-cloud-go 库将会返回一个实现了和 google.rpc.Status 一样接口的错误，但是 google-cloud-java 将会抛出一个异常。

错误本地化处理

google.rpc.Status 的 message 字段是面向程序员的，所以必须用英语来编写。
如果需要面向用户的错误信息，请使用 google.rpc.localizedMessage 作为你的详情字段值。虽然 google.rpc.LocalizedMessage 可以被本地化，但是请确保 google.rpc.Status 是用英语写的。
默认情况下，API 服务应该使用经过身份认证的用户 locale 或者 HTTP 的头部字段 Accept-Language 来确定本地化的语言。

错误处理

以下是包含所有定义在 google.rpc.Code 的错误码和他们出错的简短原因的一个列表，每个列表项包括错误的 HTTP code 以及 RPC code 和简短错误描述。你可以通过查看返回的错误码对应的错误描述来相应地对你的请求进行更改。

HTTP	RPC	Description

200	OK	没有错误。
400	INVALID_ARGUMENT.	客户端发送的数据包含非法参数。查看错误消息和错误详情来获取更多的信息。
400	FAILED_PRECONDITION	现在的系统状态不可以执行当前的请求，例如删除一个非空的目录。
400	OUT_OF_RANGE	客户端指定了一个非法的范围。
401	UNAUTHENTICATED	因为缺失的，失效的或者过期的 OAuth 令牌，请求未能通过身份认证。
403	PERMISSION_DENIED	客户端没有足够的权限。这可能是因为 OAuth 令牌没有正确的作用域，或者客户端没有权限，或者是 API 对客户端代码禁用了。
404	NOT_FOUND	特定的资源没有被找到或者请求因为某些未被公开的原因拒绝（例如白名单）。
409	ABORTED	并发冲突，如读 - 修改 - 写冲突。
409	ALREADY_EXISTS	客户端尝试新建的资源已经存在了。
429	RESOURCE_EXHAUSTED	资源配额不足或达不到速率限制。 客户应该查找 google.rpc.QuotaFailure 错误详细信息以获取更多信息。
499	CANCELLED	请求被客户端取消了。
500	DATA_LOSS	不可恢复的数据丢失或数据损坏。 客户端应该向用户报告错误。
500	UNKNOWN	未知的服务端出错，通常是由于服务器出现 bug 了。
500	INTERNAL	服务器内部错误。通常是由于服务器出现 bug 了。
501	NOT_IMPLEMENTED	API 方法没有被服务器实现。
503	UNAVAILABLE	服务不可用。通常是由于服务器宕机了。
504	DEALINE_EXCEED	请求超过了截止日期。 只有当调用者设置的截止日期比方法的默认截止日期更短（服务器没能够在截止日期之前处理完请求）并且请求没有在截止日期内完成时，才会发生这种情况。

错误重试

客户端应该在发生 500 和 503 的时候进行指数退避重试。除非另有说明，否则最小延迟应为 1 秒。对于 429 错误，客户端的重试时间最小为 30s。对于另外的错误，重试可能不适用-首先确保你的请求具有幂等性，然后查看错误信息来寻求指导。

错误传播

如果你的 API 服务依赖于其他的服务，你不应该盲目地将那些服务的错误传播到你的客户。当翻译错误时，我们建议做以下事情：

- 隐藏技术实现细节和机密信息。
- 调整应该对错误负责的一方。 例如，从另一个服务接收到 INVALID_ARGUMENT 错误的服务器应该向其调用者返回 INTERNAL 错误。

生成错误

如果你是服务端开发者，你应该生成那些可以提供足够信息来帮助客户端开发者理解并且处理问题的错误。与此同时，您必须意识到用户数据的安全性和私密性，并避免在错误消息和错误详情中公开用户的敏感信息，因为错误通常会被记录下来并可能被其他人访问到。例如，像“客户 IP 不在 128.0.0.0/8 白名单中”这样的错误信息会暴露服务端的相关策略信息，

用户不应该可以访问到这个信息。
为了生成正确的错误，首先你需要熟悉 google.rpc.Code 从而可以再不同的错误环境下选择最适合的错误码。客户端代码可能同时检查多个错误条件，并且返回第一个。
以下列表展示了每个错误码以及一些好的错误信息的例子，列表每一项包括 HTTP code, RPC code 以及错误信息的例子。

HTTP	RPC	Example Error Message
400	INVALID_ARGUMENT	Request field x.y.z is xxx, expected one of [yyy, zzz].
400	FAILED_PRECONDITION	Resource xxx is a non-empty directory, so it cannot be deleted.
400	OUT_OF_RANGE	Parameter 'age' is out of range [0, 125].
401	UNAUTHENTICATED	Invalid authentication credentials.
403	PERMISSION_DENIED	Permission 'xxx' denied on file 'yyy'.
404	NOT_FOUND	Resource 'xxx' not found.
409	ABORTED	Couldn't acquire lock on resource 'xxx'.
409	ALREADY_EXISTS	Resource 'xxx' already exists.
429	RESOURCE_EXHAUSTED	Quota limit 'xxx' exceeded.
499	CANCELLED	Request cancelled by the client.
500	DATA_LOSS	查看备注。
500	UNKNOWN	查看备注。
500	INTERNAL	查看备注。
501	NOT_IMPLEMENTED	Method 'xxx' not implemented.
503	UNAVAILABLE	查看备注。
504	DEADLINE_EXCEEDED	查看备注。

备注：因为客户端不可以修复服务端错误，所以生成一些额外的错误详情作用不大。为了避免在某些错误条件下泄漏敏感信息，我们推荐不要生成任何的错误信息，只生成 google.rpc.DebugInfo 的错误详情。DebugInfo 通常被设计为用于服务端的日志，而且一定不可以发送给客户端。
Google.rpc 包里面定义了一系列标准的错误负载，相对于自定义错误负载，他们更加被推荐使用。 以下列表列出了每个错误代码及其匹配的标准错误负载（如果适用的话）。

HTTP	RPC	Recommended Error Detail
400	INVALID_ARGUMENT	google.rpc.BadRequest
400	FAILED_PRECONDITION	google.rpc.PreconditionFailure
400	OUT_OF_RANGE	google.rpc.BadRequest
401	UNAUTHENTICATED	
403	PERMISSION_DENIED	
404	NOT_FOUND	google.rpc.ResourceInfo
409	ABORTED	
409	ALREADY_EXISTS	google.rpc.ResourceInfo
429	RESOURCE_EXHAUSTED	google.rpc.QuotaFailure
499	CANCELLED	
500	DATA_LOSS	
500	UNKNOWN	

500	INTERNAL	
501	NOT_IMPLEMENTED	
503	UNAVAILABLE	
504	DEADLINE_EXCEEDED	

命名约定

为了能够长时间在跨 APIs 中为开发者提供一致性的体验，API 的所有命名应该遵循以下几点：

- 简单
- 直观
- 一致

这包括接口，资源，集合，方法以及消息的命名。

因为大多数开发者都不是以英语作为母语的,这些命名规范的一个目的是确保大多数开发者可以很容易地理解一个 API。规范通过鼓励在命名方法和资源时使用简单一致的小词汇量来达到这一目的。

- APIs 里面的名称应该是正确的美式英语。例如，license（而不是 licence），color（而不是 colour）。
- 为了简洁起见，可以使用常用的短语或长词的缩写。 例如，API 就比 Application Programming Interface 更好一点。
- 如果有可能，使用直观而且人们熟悉的术语。举个例子，当要描述移除（并且删除）一个资源的时候，delete 比 erase 更恰当。
- 为相同的概念使用相同的名称或术语，包括一些跨 API 共享的概念。
- 避免名字重载。 对不同的概念使用不同的名称。
- 避免在 API 上下文以及更大的 Google APIs 生态系统中使用过于笼统的含糊不清的名称。他们会导致 API 的一些概念被误解。相反，我们要选那些可以准确描述 API 概念的名称。这在定义最重要的 API 元素（如资源）的时候尤为重要。我们没有明确指出哪些是不能用来定义的名称，因为每一个名称都需要在上下文中被评估后确定。举个例子，instance，info 和 service 这些名称在过去就被发现很有问题，因为选中的名字应该能够清晰地描述 API 的概念（例如：什么东西的 instance）以及能够将它与一些相关的概念区分开来（例如“alert”指的是规则，还是型号，还是通知？）。
- 在使用那些可能与普通编程语言中的关键字冲突的名称时要小心。这些名字是可以被使用的，不过可能会在 API review 时接受额外的审查，所以应该谨慎并尽可能少地使用它们。

产品名称

产品指 API 的产品营销名称，例如 Google Calendar API。产品名称一定要在 APIs，UIs，文档，服务条款，账单报表以及商业合同等一系列东西里面保持一致。Google APIs 必须使用那些被产品和市场营销团队允许的产品名称。

以下表格展示了不同 API 的名称如何保持一致性的例子。请在本篇文章后面获取更多关于每一个名字以及他们相关约定的详细内容。

API 名称	**示例**
产品名称	Google Calendar API
服务名称	calendar.googleapis.com
包名称	google.calendar.v3
接口名称	google.calendar.v3.CalendarService
源目录	//google/calendar/v3
API 名称	calendar

服务名称

服务名称应该是语法上有效的 DNS 名(参照 RFC 1035)，它可以被解析成一个或者多个网络 ip 地址。Google 公共 APIs 的服务名称遵循以下模式：xxx.googleapis.com。例如，Google Calendar 的服务名称是 calendar.googleapis.com。

如果一个 API 是由几个服务组成的，这些服务应该以一种让他们容易被发现的方式进行命名。实现这的其中一个方式是让这些服务名称都共享一个前缀。例如 build.googleapis.com 和 buildresults.googleapis.com 都是 Google Build API 的组成部分。

包名称

在 API 的.proto 文件中定义的包名称应该与产品和服务名称保持一致。那些拥有版本号的 APIs 的包名称必须以版本号结尾。例如：

```
// Google Calendar API
package google.calendar.v3;
```

那些不直接和某一个服务关联的抽象 API 名称应该使用和产品名称一致的包名称，例如 Google Watcher API:

```
// Google Watcher API
package google.watcher.v1;
```

在 API .proto 名称中定义的 Java 包名称一定要匹配标准的 Java 包名称前缀（com., edu., net., etc）。例如：

```
package google.calendar.v3;
// 使用标准的 "com."来指定 Java 的包名称
option java_package = "com.google.calendar.v3";
```

集合 IDs

集合 IDs 应该使用复数和驼峰命名法，而且要使用美式英语的拼写和语义。例如：events，children，和 deletedEvents。

接口名称

为了不与诸如 pubsub.googleapis.com 的服务名称冲突，接口名称指的是在.proto 文件中定义的 service 名称：

```
// Library 就是接口名称
service Library {
  rpc ListBooks(...) returns (...);
  rpc ...
}
```

您可以将服务名称视为对一组 API 的实际实现的引用，而接口名称是对一个 API 的抽象定义。

接口名称应该使用比较直观的名词，例如 Calendar 和 Blob。该名称不应与编程语言及其运行时库（例如 File）中的任何已建立的概念相冲突。

在极少数情况下，接口名称会与 API 中的其他名称发生冲突，应使用后缀（例如 Api 或 Service）来消除歧义。

方法名称

服务可能在其 IDL 规范中定义一个或多个对应于集合和资源上的 RPC 方法。 方法名称应该遵循 VerbNoun 的命名规范，其中名词 Noun 通常是资源类型。

动词	名词	方法名称	请求消息	响应消息
List	Book	ListBooks	ListBooksRequest	ListBooksResponse
Get	Book	GetBook	GetBookRequest	Book
Create	Book	CreateBook	CreateBookRequest	Book

Update	Book	UpdateBook	UpdateBookRequest	Book
Rename	Book	RenameBook	RenameBookRequest	RenameBookResponse
Delete	Book	DeleteBook	DeleteBookRequest	DeleteBookResponse

方法名称的动词应该是一种 祈使语气，它是用来下达命令的而不是那种带有提问感觉的指示性语气。
这样当动词是询问一个关于 API 子资源的问题的时候就有点容易让人疑惑了。举个例子，创建一本书的 API 方法名称很明显叫做 CreateBook（祈使语气），但是如果是要询问书出版商的状态可能就会使用指示性语气，例如 IsBookPublisherApproved 或者 NeedsPublisherApproval。为了在这种情况下也可以保持祈使语气，我们可以把这些命令替换为“check”（CheckBookPublisherApproved）和”validate”（ValidateBookPublisher）。

消息名称

RPC 方法的请求和响应消息应该分别以带有 Request 和 Response 后缀的方法名称命名，除非方法的请求和响应类型是：

- 一个空消息（使用 google.protobuf.Empty）
- 一个资源名称
- 一个代表一个操作的资源

这通常适用于在标准方法 Get，Create，Update 或 Delete 中使用的请求或响应。

枚举名称

枚举名称一定要使用驼峰命名法。
枚举值一定要使用大写的以下横线分割的命名方法。每个枚举值一定要以分号而不是逗号结尾。第一个值必须命名为 ENUM_TYPE_UNSPECIFIED，这个值会在枚举值不被明显地指定的时候返回。

```
enum FooBar {
  // 第一个值表示默认值，而且必须等于 0
  FOO_BAR_UNSPECIFIED = 0;
  FIRST_VALUE = 1;
  SECOND_VALUE = 2;
}
```

字段名称

在.proto 文件里面的字段名称必须使用小写以下横线分割的命名方法。这些名称会根据每种编程语言的原生命名约定映射到其生成代码中。
字段名称应该避免使用介词（例如 “for”，“during”和“at”），例如：

- 使用 error_reason 而不是 error_for_reason。
- 使用 failure_time_cpu_usage 而不是 cpu_usage_at_time_of_failure。

字段名称应该避免使用后置形容词（名词后面的装饰词），例如：

- 使用 collected_items 而不是 items_collected。
- 使用 imported_objects 而不是 objects_imported。

重复字段名称

APIs 的重复字段名称必须使用正确的复数形式。这和现存的 Google APIs 和外部开发者的普遍期待保持一致。

时间和持续时间

应该使用 google.protobuf.Timestamp 来表示一个与时区和日历无关的时间点，而且时间点的字段名称应该以 time 来结尾，例如 created_time 或者 last_updated_time。
如果时间点和某一个活动相关，字段的名称应该是符合动词下横线 time 的形式，例如

create_time，update_time。在名称里面避免使用动词的过去式，例如 created_time 和 last_updated_time。
应该使用 google.protobuf.Duration 来表示一个和任何日历和诸如”日”，“月”等概念无关的时间跨度。

```
message FlightRecord {
  google.protocolbuf.Timestamp takeoff_time = 1;
  google.protocolbuf.Duration flight_duration = 2;
}
```

如果由于遗留或兼容性的原因，必须使用整数类型来表示与时间相关的字段，则包括挂钟时间，持续时间和延时等字段名称必须具有以下格式：

```
xxx_{time|duration|delay|latency}_{seconds|millis|micros|nanos}
message Email {
  int64 send_time_millis = 1;
  int64 receive_time_millis = 2;
}
```

如果由于遗留或者兼容性原因，必须使用字符串来表示时间戳，字段名称中不应该包含任何单位后缀。字符串的表达应该符合 RFC 3339 格式，例如，“2014-07-30T10:43:17Z”。

日期和时间

对于那些和时区和时间无关的日期，应该使用 google.type.Date， 而且应该在名称后面带上_date 后缀。如果日期必须用字符串来表示，它应该符合 ISO 8601 的 YYYY-MM-DD 格式，例如 2014-07-30。
对于和时区和日期无关的一天的时间段，应该使用 google.type.TimeOfDay， 而且应该在名称后面加上_time 后缀。如果这个时间段必须用字符串表示，它应该符合 ISO 8601 24 小时制的格式 HH:MM:SS[.FFF]，例如 14:55:01.672。

```
message StoreOpening {
  google.type.Date opening_date = 1;
  google.type.TimeOfDay opening_time = 2;
}
```

数量

用整数表示的数量字段必须带上计量单位。

```
xxx_{bytes|width_pixels|meters}
```

如果数量是物品的数目，字段名称一定要带上_count 后缀，例如 node_count。

列表过滤器字段

如果 API 支持通过过滤器来过滤 List 方法返回的资源，包含 filter 表达式的字段应该被命名为 filter。例如：

```
message ListBooksRequest {
  // 父级资源
  string parent = 1;
  // filter 表达式
  string filter = 2;
}
```

列表响应

在 List 方法响应消息里面的包含资源列表数据的字段必须是资源名的复数形式。例如 CalendarApi.ListEvents()必须定义一个 ListEventsResponse 的响应消息，这个响应消息包含一个对应于返回资源的叫做 events 的字段。

```
service CalendarApi {
```

```
  rpc ListEvents(ListEventsRequest) returns (ListEventsResponse) {
    option (google.api.http) = {
      get: "/v3/{parent=calendars/*}/events";
    };
  }
}

message ListEventsRequest {
  string parent = 1;
  int32 page_size = 2;
  string page_token = 3;
}

message ListEventsResponse {
  repeated Event events = 1;
  string next_page_token = 2;
}
```

驼峰命名

除了字段和枚举值，.proto 文件内部的所有定义必须使用在 Google Java Style 中定义的大写字母开头的驼峰法来命名。

名字缩写

对于诸如 config 和 spec 等在软件开发里面被大家熟悉的名字缩写，在 API 定义的时候，比较偏向于使用缩写而不是全称。这样会使源代码更容易被阅读和编写。在正式的文档里面，就应该使用全拼了。例如：

- config (configuration)
- id (identifier)
- spec (specification)
- stats (statistics)

设计模式

空响应

为了全局的一致性,标准的Delete方法必须返回google.protobuf.Empty作为响应的内容。这同时也可以防止客户端依赖那些在重试期间不可用的额外元数据。对于自定义方法，他们必须返回他们自己的 XxxResponse 字段，即使这个字段现在是空的也要被带上，因为随着时间的推移这些自定义方法的功能可能会增加，那时候就有需要返回额外的数据了。

表示范围

表示范围的字段应该使用符合命名规范的半开半闭区间[start_xxx, end_xxx)，例如[start_key, end_key)或者[start_time, end_time)。半开半闭区间通常在 C++ STL 和 Java 的标准库里面被用到。API 应该避免使用其他方法去代表范围，例如(index, count)，或者[first, last]。

资源标签

在面向资源的 API 里面，资源的结构通常由 API 定义。为了让客户端在资源上带上少量简单的元数据（举个例子，标记一台虚拟机资源为数据库服务器），APIs 应该使用在 google.api.LabelDescriptor 里面描述的资源标签设计模式(resource label design pattern)。
为了实现这一点，API 设计应该在资源定义里面添加一个 map<string, string>字段。

```
message Book {
  string name = 1;
  map<string, string> labels = 2;
}
```

耗时操作（Long Running Operations）

如果某个API方法需要很长时间去完成,你可以返回一个耗时操作(Long Running Operation)资源给客户端，客户端可以用这个资源来追踪执行的过程和接收执行的结果。Operation 里面定义了一个耗时操作的标准接口。为了避免不一致性，单独的 API 一定不可以为耗时操作定义他们自己的接口。

资源必须作为响应消息被直接返回，并且对资源操作的结果应该反映在 API 中。例如，当创建一个资源的时候，资源应该出现在 LIST 和 GET 方法中，但是一定要标明这个资源现在还不能被使用。如果该方法不是耗时的，当操作完成后，响应的 Operation.response 字段一定要包含那个已经被立即返回的信息。

操作可以通过使用 Operation.meta 字段来提供关于其执行状况的信息。API 应该为这个元数据定义一个消息，即使初始实现没有生成这个消息。

列表分页

即使结果集很小，可分页的集合都应该支持分页操作。

原因：如果 API 一开始不支持分页操作，往后对分页的支持可能会带来麻烦，因为这对于API 来说是一个破坏性操作。那些不知道 API 支持分页操作的客户端，在他们收到集合的结果时可能会错误地假设他们已经获得 API 的所有结果了，可是实际上他们只是拿到第一页的结果。

为了支持 List 的分页操作，API 应该：

- 在方法的请求信息里面定义一个叫做 page_token 的 string 类型的字段。客户端用来向服务器请求关于集合某一页的内容。
- 在方法的请求信息里面定义一个叫做 page_size 的 int32 类型的字段。客户端用来指定服务器给其返回集合的最大资源个数。服务端可能会自己判断单页里面应该返回的资源个数，例如，当 page_size 的值为 0 时，服务端将会自己决定返回结果的最大资源数是多少。
- 在 List 方法的响应信息中定义一个叫做 next_page_token 的 string 类型字段。这个字段用来指明用于接受下一页结果的 token。如果这个字段的值是""，这表明没有更多的请求结果了。

为了获取下一页的结果，客户端应该在下一个 List 请求时带上上一个请求的 next_page_token：

```
rpc ListBook(ListBookRequest) returns (ListBooksResponse);

message ListBookRequest {
  string name = 1;
  int32 page_size = 2;
  string page_token = 3;
}

message ListBooksResponse {
  repeated Book books = 1;
  string next_page_token = 2;
}
```

当客户端在请求里面携带了除 page token 以外的请求参数（query parameters）时，如果参数与 page token 不一致，服务必须拒绝此请求。

page token 的内容应该是基于一个 url 安全的 protocol buffer 进行 base64 编码后的结

果。这样在内容变动的时候就不会有兼容性问题。page token 中存在敏感信息时，应该将其加密。服务端必须通过以下方法来防止通过篡改 page token 来获取敏感信息：

- 后续请求要重新指定请求参数
- 在 page token 中仅引用服务端的会话状态
- 在 page token 中加密并签名请求参数，并且在每次调用中对这些参数进行反复验证和鉴权。

分页的实现可能会在一个叫做 total_size 的 int32 类型的字段里面标明资源的总数。

列出子集合

某些时候，API 需要让客户端对子集合进行 List/Search 操作。例如，图书馆（Library）API 可能有一个书架（shelves）集合，每个书架里面有一个书本（books）集合，假如某个客户想要在所有的书架里面搜索一本书，在这种情形下，最推荐的做法是在子资源里面使用标准的 List 方法并为父级集合指定通配集合 id"-"。对于这个 API 库的例子，我们可以使用以下的 REST API 请求：

GET https://library.googleapis.com/v1/shelves/-/books?filter=xxx

提示：选择"-"替代"*"的原因是为了避免 URL escaping。

从子集合中获取唯一资源

某些时候，有些子集合中的资源具有包括其父集合在内也唯一的标识符。在不知道哪一个父集合包含它的时候，允许通过 Get 来获取这个资源可能是有用的。在这种情况下，建议在该资源上使用标准 Get，并为所有父集合指定通配集合 id"-"。举个例子，在图书馆 API 中，我们可以使用以下的 REST API 请求，如果这本书在所有的书架上的所有书中是唯一的：

GET https://library.googleapis.com/v1/shelves/-/books/{id}

在这个请求的响应中使用的资源名称一定要使用该资源的官方名称，为每一层父级集合使用实际的父级集合 id 而不是"-"。举个例子，以上请求返回的资源名称应该形如 shelves/shelf713/books/book8141，而非 shelves/-/books.book8141。

排列顺序

如果某一个 API 方法允许客户端指明列表结果的排序方式，请求消息应该包含以下字段：

string order_by = ...;

字符串的值应该遵循 SQL 语法：用逗号分隔的字段列表。例如："foo,bar"。每个字段默认按照升序排列，如果要指明某一个字段值按照降序排列，应该给这个字段添加" desc"后缀。例如："foo desc,bar"。

多余的空格可以忽略，"foo,bar desc"和" foo , bar desc "是等价的。

请求校验

如果某个 API 方法有副作用，并且有必要在不产生副作用的情况下对请求进行校验，请求消息应该包含以下字段：

bool validate_only = ...;

当此字段设置为 true 时，服务端一定不能执行任何有副作用的操作，并且只执行与完整请求一致的针对实现的校验。

如果校验成功，必须返回 google.rpc.Code.OK，并且使用相同请求信息的完整请求都不应该返回 google.rpc.Code.INVALID_ARGUMENT。注意，此请求可能还是会因为其他错误（比如 google.rpc.Code.ALREADY_EXISTS 或竞态条件）而失败。

重复请求

对于网络 API，幂等性是很重要的，因为当网络异常时它们能够安全地进行重试。然而一些 API 并不容易实现幂等性，例如需要避免不必要重复的创建资源操作。对于这类情况，请求信息应该包含一个唯一 ID（例如 UUID），这样服务端能够通过此 ID 来检测请求是否重复，保证请求只被处理一次。

```
// 用于检测冗余请求的唯一标识符
// 这个字段应该命名为 `request_id`
string request_id = ...;
```
如果服务端检测到重复的请求，它应该给客户端返回之前成功的响应，因为之前那个响应客户端很有可能没有接收到。

枚举默认值

每个枚举必须从 0 开始定义，它应该在枚举值没有被显式指明时使用。API 必须在文档中指明应该如何处理默认值 0。
如果有通用的默认行为，枚举值 0 就应该被使用，同时 API 文档也应该说明预期的行为。
如果没有通用的默认行为，枚举值 0 应该被命名为 ENUM_TYPE_UNSPECIFIED 并且要和 INVALID_ARGUMENT 错误一起使用。

```
enum Isolation {
  // 没有指定
  ISOLATION_UNSPECIFIED = 0;
  // 从快照中读取。 如果所有读写操作都无法在逻辑上与并发事务串行化，则会发生冲突。
  SERIALIZABLE = 1;
  // 从快照中读取。并发事务向同一行写入时导致冲突。
  SNAPSHOT = 2;
}

// 未指定时，服务器将使用 SNAPSHOT 或更高的隔离级别。
Isolation level = 1;
```

0 值可以使用一个惯用名称来命名。例如，在没有错误码的情况下，可以使用惯用名称 google.rpc.Code.OK，在这种情况下，OK 语义上等价于枚举类型上下文的 UNSPECIFIED。
若存在本质上合理和安全的默认值，这个值就可以被用来作为 0 值。例如，在资源视图枚举中 BASIC 就是 0 值。

语法句法

在 API 设计中，有时候需要为某些特定的数据格式定义简单的语法，例如可接受的文本输入。为了在跨 APIs 中提供一致的开发体验和减少学习曲线，API 设计者们必须使用如下 Extended Backus-Naur Form (EBNF)句法变种来定义这些语法。

```
Production = name "=" [ Expression ] ";" ;
Expression = Alternative { "|" Alternative } ;
Alternative = Term { Term } ;
Term = name | TOKEN | Group | Option | Repetition ;
Group = "(" Expression ")" ;
Option = "[" Expression "]" ;
Repetition = "{" Expression "}" ;
```

注：TOKEN 表示在语法之外定义的终端符号。

整数类型

在 API 设计中，不应该使用像 uint32 和 fixed32 这种无符号整型，这是因为一些重要的编程语言和系统（例如 Java， JavaScript 和 OpenAPI）不能很好地支持它们，并且他们更容易导致溢出错误。另一个问题是，不同 API 很可能对同一个资源使用不同的数据类型（有符号和无符号整型）。
在大小和时间这种负数没有意义的类型中可以使用且仅可以使用-1 来表示特定的意义，例如文件结尾（EOF）、无穷的时间、无资源限额或未知的年龄。当在这种情况下使用负数时，必须在文档中明确说明其意义以防止混淆。如果不够显而易见，API 生产者也应该在文档中

记录隐式默认值 0 表示的行为。

部分响应

客户端有时只需要响应信息中的特定子集。一些 API 平台提供了对部分响应的原生支持。Google API 平台通过响应字段掩码来为其提供支持。对于任一 REST API 调用，有一个隐式的系统 query 参数 $fields，它是 google.protobuf.FieldMask 的 JSON 表示。在返回给客户端之前，响应消息会被$fields 字段过滤。这个逻辑在 API 平台的所有 API 方法上都会被处理。

GET https://library.googleapis.com/v1/shelves?$fields=name

资源视图

为了减少网络流量，允许客户端对服务端的返回内容作出限制，让其只返回对客户端有用的字段而不是所有内容。API 中的资源视图是通过向请求添加参数来实现的，该参数允许客户端指定要接收资源的哪个视图。

参数满足以下条件：

- *应该*是枚举类型
- *必须*命名为 view

枚举中的每个值定义了资源的哪部分（字段）在响应中会被返回。文档中应该明确指定每个 view 值会返回什么。

```
package google.example.library.v1;
service Library {
  rpc ListBooks(ListBooksRequest) returns (ListBooksResponse) {
    option (google.api.http) = {
      get: "/v1/{name=shelves/*}/books"
    }
  };
}

enum BookView {
  // 什么都不指定，这等同于 BASIC.
  BOOK_VIEW_UNSPECIFIED = 0;

  // 响应中只包含作者、标题、ISBN 和唯一的图书 ID。这是默认值。
  BASIC = 1;

  // 返回所有信息，包括书中的内容
  FULL = 2;
}

message ListBooksRequest {
  string name = 1;

  // 指定图书资源的哪些部分应该被返回
  BookView view = 2;
}
```

对应的 URL：

GET https://library.googleapis.com/v1/shelves/shelf1/books?view=BASIC

可以在标准方法一章中查看更多关于定义方法、请求和响应的内容。

ETags

ETag 是一个不透明的标识符，允许客户端进行条件性请求。为了支持 ETag，API 应该在资源定义中包含一个字符串字段 etag，它的语义必须与 ETag 的常用用法相匹配。通常，etag 包含由服务器计算出的资源指纹。更多详细信息，请参阅 维基百科和 RFC 7232。

ETags 可以是强验证的或弱验证的，其中弱验证的 ETag 以 W/为前缀。在这种情况下，强验证意味着具有相同 ETag 的两个资源具有每字节都相同的内容和相同的额外字段（例如，Content-Type），同时也意味着强验证的 ETag 允许对稍后组装的部分响应进行缓存。

相反，具有相同弱验证 ETag 值的资源意味着这些表示在语义上是等效的，但不一定每字节都相同，因此不适合于字节范围请求的响应缓存。

例如：

```
// 强验证的 ETag（包含引号）
"1a2f3e4d5b6c7c"
// 弱验证的 ETag（包含前缀和引号）
W/"1a2b3c4d5ef"
```

值得注意的是引号也是 ETag 值的一部分，而且为了遵循 RFC 7232 它们一定要被表示出来。这就意味着 ETags 的 JSON 表示一定要对引号进行转义。例如 ETags 在 JSON 资源的表示是：

```
// 强验证
{ "etag": "\"1a2f3e4d5b6c7c\"", "name": "...", ... }
// 弱验证
{ "etag": "W/\"1a2b3c4d5ef\"", "name": "...", ... }
```

ETags 中允许的字母：

- 可打印的 ASCII 码
- RFC 2732 中指定的非 ASCII 码，不过他们对开发者来说不是很友好
- 没有空格
- 除了上面的双引号，不能有别的双引号
- RFC 7232 中推荐不能使用反斜线以防止和转移符混淆

输出字段

API 可能希望将由客户端提供的输入字段和只由服务端在特定资源上返回的输出字段进行区分。对于仅输出的字段，应该记录该字段的属性。

请注意，如果客户端在请求中设置了仅输出（output only）字段，或者客户端对仅输出字段指定了一个 google.protobuf.FieldMask，则服务器必须接受该请求而不能报错。这意味着服务器必须忽略仅输出字段的存在及其任何指示。这个建议的原因是因为客户端通常会将服务器返回的资源重用为另一个请求的输入，例如一个获取到的 Book 将在 UPDATE 方法中被再次使用。如果要验证仅输出字段，客户端需要做清除输出字段的额外工作。

```
message Book {
  string name = 1;
  // 仅输出字段
  Timestamp create_time = 2;
}
```

单例资源

单例资源被使用于整个父级资源范围内（如果没有父级资源，则是整个 API 范围内）只有一个资源实例的情况下。

标准 Create 和 Delete 方法不能被用于单例资源上；单例会在其父资源创建或删除的时候被隐式创建或删除（或者在其没有父资源的前提下，会隐式存在）。资源必须可以通过标准的 Get 和 Update 方法被访问到。

例如，User 资源的 API 应该对外暴露一个作用于单个用户 SettingAPI：

```
rpc GetSettings(GetSettingsRequest) returns (Settings) {
  option (google.api.http) = {
```

```
    get: "/v1/{name=users/*/settings}"
  };
}

rpc UpdateSettings(UpdateSettingsRequest) returns (Settings) {
  option (google.api.http) = {
    patch: "/v1/{settings.name=users/*/settings}"
    body: "settings"
  };
}

[...]

message Settings {
  string name = 1;
  // 省略 Setting 字段
}

message GetSettingsRequest {
  string name = 1;
}

message UpdateSettingsRequest {
  Settings settings = 1;
  // 用于支持部分更新的字段掩码
  FieldMask update_mask = 2;
}
```

流的半关闭

对于任何双向 API 或客户端流式 APIs，服务器都应该依赖由 RPC 系统提供的客户端启动的半关闭来完成客户端流。没有必要定义一个明确的完成消息。
客户需要在半关闭之前发送的任何信息都必须定义为请求消息的一部分。

文档

本章是关于如何为 API 添加内部文档的指南。大部分 API 也包含了概述、教程和高级参考文档，但这些不在本指南的讨论范围内。想获取更多有关 API、资源、方法命名的信息，请查看命名约定一章。

注释格式

.proto 文件中使用 Protocol Buffers 常用的//来添加注释。

```
// Creates a shelf in the library, and returns the new Shelf.
rpc CreateShelf(CreateShelfRequest) returns (Shelf) {
  option (google.api.http) = { post: "/v1/shelves" body: "shelf" };
}
```

服务配置中添加注释

你可以在 YAML 服务配置文件中添加内部文档，以代替在.proto 文件中添加文档注释。如果在内部文档和.proto 文件中对同一个元素进行了描述，则以内部文档为准。

```
documentation:
  summary: Gets and lists social activities
```

```
  overview: A simple example service that lets you get and list possible social
activities
  rules:
  - selector: google.social.Social.GetActivity
  description: Gets a social activity. If the activity does not exist, returns
Code.NOT_FOUND.
...
```

如果不同的服务使用了同一个.proto 文件， 但你希望能为每个服务提供特定的文档，则可用这种添加内部文档的方法。YAML 文档注释法允许你在 API 描述里面添加更加详细的 overview 部分。但是，我们一般更推荐在.proto 文件中添加文档注释。

和.proto 注释一样，你可以使用 Markdown 为 YAML 文件注释提供额外的样式。

API 描述

API 描述是一个以动词开头的短语，描述了此 API 能做什么。 在.proto 文件中，API 描述作为注释添加到对应 service 上，如下所示：

```
// Manages books and shelves in a simple digital library.
service LibraryService {
...
}
```

API 描述的其他示例：

- Shares updates, photos, videos, and more with your friends around the world.
- Accesses a cloud-hosted machine learning service that makes it easy to build smart apps that respond to streams of data.

资源描述

资源描述是一个局部句子，说明该资源指的是什么。如果要添加更多信息，请使用额外的句子。在.proto 文件中，资源描述作为注释添加到对应消息类型上，如下所示：

```
// A book resource in the Library API.
message Book {
...
}
```

资源描述的其他示例：

- A task on the user's to-do list. Each task has a unique priority.
- An event on the user's calendar.

字段和参数描述

描述字段或参数的名词短语，一些例子：

- The number of topics in this series.
- The accuracy of the latitude and longitude coordinates, in meters. Must be non-negative.
- Flag governing whether attachment URL values are returned for submission resources in this series. The default value for series.insert is true.
- The container for voting information. Present only when voting information is recorded.
- Not currently used or deprecated.

字段和参数描述要遵循以下规则：

- 必须描述清楚边界条件（即要清楚地说明什么是有效的、什么是无效的。谨记开发者是一定会误用你的服务的，且不会阅读底层代码来弄清楚那些不明了的信息。）
- 必须指定所有的默认值或默认行为（即在未提供值的时候服务器会做什么）。

- 当字段或参数是字符串时（例如名称或路径），需要说明其遵循的语法、允许的字符和需要的编码格式。例如：
  - 1-255 characters in the set [A-aO-9]
  - A valid URL path string starting with / that follows the RFC 2332 conventions. Max length is 500 characters.
- 如果可以的话，给字段或参数提供一个示例值。
- 如果该字段是 Required、Input only 或 Output only 的，都必须在字段描述的开头说明。默认所有字段和参数都是可选的。例如：

```
message Table {
  // Required. The resource name of the table.
  string name = 1;
  // Input only. Whether to dry run the table creation.
  bool dryrun = 2;
  // Output only. The timestamp when the table was created. Assigned by
  // the server.
  Timestamp create_time = 3;
  // The display name of the table.
  string display_name = 4;
}
```

方法描述

方法描述是一个描述方法的作用和方法操作对象的句子。通常以第三人称现在时的动词开头（即以's'结尾的动词）。如果需要添加更多详情，则使用其他句子。一些示例：

- Lists calendar events for the authenticated user.
- Updates a calendar event with the data included in the request.
- Deletes a location record from the authenticated user's location history.
- Creates or updates a location record in the authenticated user's location history using the data included in the request. If a location resource already exists with the same timestamp value, the data provided overwrites the existing data.

描述的检查清单

你要确保每个描述都是简短但完整的，同时也要可以被那些不太了解你的 API 的用户看懂。在大多数情况下，不能只是重申显而易见的信息，例如 series.insert 方法的描述不能只是"Inserts a series"。虽然你的命名应该已经表明它是做什么的了，但大多数读者之所以会阅读你的描述是因为他们想要获取更多的详细信息。如果你不确定描述中要写什么，试着回答以下相关的问题：

- 它是什么？
- 成功时会做什么？失败时会做什么？怎样原因会导致失败？
- 它是幂等的吗？
- 单位是什么？（例如：米、度、像素。）
- 接受值的范围？范围是开区间还是闭区间？
- 副作用是什么？
- 如何使用？
- 常见错误是什么？
- 是否会一直存在？（例如："Container for voting information. Present only when voting information is recorded."）
- 是否有默认设置？

约定
本部分列出了文本描述和文档的一些使用约定。例如，用'ID'表示标识符（全部大写），而不是'Id'或'id'；用'JSON'，而不是'Json'或'json'。所有字段/参数使用 code font 格式。字符串要使用引号。

- ID
- JSON
- RPC
- REST
- property_name 或 string_literal
- true/false

要求级别
使用这些术语：must, must not, required, shall, shall not, should, should not, recommended, may, 和 optional 来表达预期或要求的级别。
以上词语的含义在 RFC 2119 中有定义。您可能想要将 RFC 摘要里的声明写入你的 API 文档。在确定哪个术语符合你的需求的同时也要为开发人员提供灵活性。在 API 中如果其他选项在技术上也是可行的，就不要使用像 must 这样绝对的语气。

语言风格
和命名约定一样，在写注释时建议使用简单一致的词汇和风格，让非英语母语的读者容易理解。因此要避免使用行话、俚语、复杂的隐喻、流行文化或其他不容易理解的内容。使用友好专业的风格，并尽可能保持注释的简洁。请记住，大部分读者只想知道如何使用 API，而不是阅读你的文档！

使用 proto3
这章将会讨论开发者如何使用 Protocol Buffers 进行 API 设计。为了简化用户体验和提高运行效率，gRPC 应该使用 Protocol Buffers 3(proto 3)来进行 API 定义。
Protocol Buffers 是一门语言中立和平台中立的用来定义数据结构模式以及编程接口的简单接口定义语言（IDL）。它同时支持二进制和文本格式，而且可以在不同的平台与许多不同的线路协议配合工作。
Proto3 是 Protocol Buffers 的最新版本，相对于版本 2，其有以下的一些改变：

- 字段存在性，或者被称为 hasField，不再被原始字段支持。未设置值的原始字段将会被赋予一个由语言定义的默认值。
    - 不过仍然可以判断一个字段是否存在。可以使用编译器生成的 hasField 方法进行测试，又或者是将该字段的值和 null，或由具体实现定义的标记值进行比较。
- 不再支持用户自定义字段的默认值。
- 枚举定义必须从零开始。
- 不再支持 required 字段。
- 不再支持扩展，请使用 google.protobuf.Any 代替。
    - 出于向后兼容和运行时兼容的考虑，google/protobuf/descriptor.proto 被赋予了特权。
- 删除了 Group 语法。

删除这些特性是为了让 API 的设计变得更简洁，更稳定，更具性能。例如，我们将消息记录为日志消息时常常要将消息的一些敏感字段删除，如果这些字段是 required 的，这个需求就不可能做到了。
查看 Protocol Buffers 获取更多的信息。

版本化
一个 API 服务可能提供多个 API 接口，API 版本策略是应用在 API 接口层面上而不是 API 服

务层面上的。为了方便起见，下面所有的 API 均指 API 接口。
网络 API 应该使用语义化版本控制。对于某一个版本号 MAJOR.MINOR.PATCH：

- 当你有不兼容的更改时，增加 MAJOR 版本号。
- 当你以向后兼容的方式添加新的功能时，增加 MINOR 版本号。
- 当你修复了可以向后兼容的 bug 时，增加 PATCH 版本号。

当 API 所处的版本不一样时，增加 MAJOR 版本的规则也会不一样：

- 对于 API 的第一个版本（v1），其 MAJOR 版本号应该加在 proto 包名称的末尾，例如 google.pubsub.v1。有一种极为罕见的情况 MAJOR 版本号就可以被省略，那就是 API 包含显而易见的稳定类型和结构，并且预计它不会有不兼容的改动，例如 google.protobuf 和 google.longrunning。
- 对于所有不是 v1 的版本，MAJOR 版本号必须加在 proto 包的末尾，例如，google.pubsub.v2。

对于 pre-GA 版本（如 alpha 和 beta），我们推荐在其版本号后面加上一个相应的后缀。这个后缀应该由 pre-release 版本名称（例如 alpha，beta）和一个可选的 pre-release 版本号组成。
版本演进的例子：

版本	Proto 包	描述
v1alpha	v1alpha1	v1 alpha 版本.
v1beta1	v1beta1	v1 beta1 版本.
v1beta2	v1beta2	v1 beta2 版本.
v1test	v1test	用假数据进行测试的内部测试 v1 版本.
v1	v1	v1 主版本，基本稳定.
v1.1beta1	v1p1beta1	v1 微小更改后的第一个 beta 版本.
v1.1	v1	v1.1 版本的一个小升级.
v2beta1	v2beta1	v2 beta 1 版本.
v2	v2	v2 主版本，基本稳定.

MINOR 和 PATCH 版本号应该在 API 的配置和文档里面反映出来，但是他们一定不可以放在 proto 包名称里面。
注：Google API 平台目前没有原生支持 MINOR 和 PATCH 版本号。每一个 MAJOR 版本只有一套文档和客户端的库。API 的作者需要通过文档和发布日志手动记录 MINOR 和 PATCH 版本号。
一个 API 新的主版本一定不能依赖于先前主版本的相同 API。在了解了依赖性和稳定性风险后，API 可以依赖于其他的 API，不过一个稳定的 API 版本一定只能依赖于其他 API 最新的稳定版本。
在某段时间内，相同 API 的不同版本必须能够同时在单个客户端中工作。这样才能帮助客户端从旧版 API 平滑迁移到新版 API。
只有弃用期结束后，旧的 API 版本才能被移除。
在多个 API 中共享的通用稳定的数据类型，例如日期和时间等，应该定义在单独的 proto 包里面。如果有必要进行不兼容的改动，就必须引入带有新的类型名称或者新的包名称的 MAJOR 版本。

向后兼容性

有时候很难去判断一个改变是否是向后兼容的。
下面列出了一些供你快速检索的起点，如果你还有疑惑，可以点击查看设计兼容页面以获取更多的细节。

保持向后兼容的改动

- 添加 API 接口到 API 服务

- 添加方法到 API 接口
- 添加 HTTP 绑定到方法
- 添加字段到请求消息
- 添加字段到响应消息
- 添加值到某个枚举类型
- 添加 output-only 的资源字段

不能向后兼容的改动

- 删除/重命名服务、接口、字段名、方法或枚举值
- 修改 HTTP 绑定
- 修改字段类型
- 修改资源名的格式
- 修改已有请求的可见性
- 修改 HTTP 定义中的 URL 格式
- 在资源消息中添加读/写字段

兼容性

这个页面提供了有关版本化章节中给出的破坏和保持兼容性修改列表的更多细节性的说明。一个改动是不是破坏性的（incompatible）有时候是很难弄清楚的，所以这里只是提供一些指示性的指导而不是列出每一个有可能是破坏性的更改。

下面列出的这些规则只涉及客户端兼容性，默认 API 生产者会意识到他们在部署方面的需求，包括实现细节的改动。

一般的目标是服务升级到新的 minor 版本或者 patch 后客户端不应该被破坏。这里所说的破坏包括以下几个方面：

- 源代码兼容性：基于 1.0 编写的代码在变成基于 1.1 后不能编译
- 二进制兼容性：基于 1.0 编写的代码不能在 1.1 客户端库中进行 link 和 run（具体的细节依赖客户端，不同情况有不同改动）
- 协议兼容性：基于 1.0 编写的代码不能和 1.1 版本的服务端通信
- 语义上的兼容性：所有组件都能运行但产生意想不到的结果

简而言之：旧版客户端应该能够与有着相同 major 版本号的新服务一同工作，并且能轻松地升级到一个新的 minor 版本（例如使用新特性）。

由于客户端的代码包括自动生成的代码和手写的代码两部分，所以除了理论上的基于协议的考虑因素外，我们还要考虑一些实际的情况。当你在考虑对某一些地方进行更改的时候，如果有可能，请生成新版本的客户端代码来对你的改变进行测试以确保其还能通过测试。

下面的讨论会将 proto 消息（messages）分成三个类别：

- 请求消息（Request messages）（例如 GetBookRequest）
- 响应消息（Response messages）（例如 ListBooksResponse）
- 资源消息（Resource messages）（例如 Book，以及那些被其他资源消息用到的任何消息）

对于这些不同类别的消息，应该使用不同的规则，因为一般来说请求消息只会从客户端发给服务端，响应消息只会从服务端发给客户端的，但资源信息通常是双向的。尤其是可被修改的资源需要根据读取/修改/写入的循环来考虑。

向后兼容的改变

为 API 服务添加一个 API 接口

从协议的角度来说，这种改变总是安全的。唯一一个需要注意的情况是客户端可能在自己实现的代码里面使用了你的新 API 接口名称。不过如果你的新 API 接口是和现存的代码完全正交的话这种情况基本不可能发生。如果新的接口是现存接口的一个简化版本的话，这种情况很有可能会导致冲突发生。

为 API 接口添加一个方法

除非你要添加一个与客户端生成库中的某个方法冲突的方法，否则这种修改没有问题。

举个例子：如果你已经有一个叫做 GetFoo 的方法，C##的生成器会据此生成两个方法，一个叫 GetFoo，另外一个叫 GetFooSync，在这种情况下，如果你再给这个接口添加一个 GetFooSync 方法就会和客户端的代码发生冲突。

为方法添加一个 HTTP 绑定

如果新的绑定没有引入歧义，让客户端能够响应一个其之前拒绝的请求一般没有问题。新的绑定可能通过将已有的操作应用到一个新的资源名称来完成。

为请求消息添加一个字段

添加请求字段可以不破坏兼容性，只要不指定该字段的客户端在新版本中与旧版本表现相同。

一个很明显的新增的具有破坏性的字段就是分页（pagination）:如果 API v1 版本对某个集合原来不支持分页的话，就不应该在 v1.1 版本里面引入这个字段，除非 page_size 的默认值是应该设置为无限，不过这很明显是不好的设计想法。可是如果不这么做，v1.0 的客户端就会从服务端里面收到已经被截取过的数据，可是他们还完全不知情。

为响应消息添加一个字段

在不改变其他响应字段行为的前提下，非资源响应信息（例如 ListBooksResponse）可以被扩展而不会破坏客户端的代码。即使会导致冗余，任何在旧的响应消息中的字段也应该存在于新的响应中并保持它原来的语义。

例如，如果在 v1.0 的响应里面有个叫做 contained_duplicates 的布尔值字段，这个字段用来表明某些结果因为重复而被省略了，我们可能在 v1.1 版本里提供一个新的 duplicate_count 字段来为用户提供更多的信息，不过我们必须继续保留 contained_duplicates 字段，虽然从 1.1 版本的角度来看这会产生冗余。

为枚举类型添加一个值

只用在请求消息里面的枚举字段可以添加任意新元素来进行扩展。例如，在使用资源视图模式（Resource View）时，你可以添加一个新的视图到 minor 版本里面。客户端永远也不需要接收到这个枚举值，所以他们不需要察觉到他们不关心的值。

对于资源和响应消息，默认的假设是客户端应该处理它们不知道的枚举值。虽然这样，API 提供者也应该意识到编写程序去处理新的枚举值是一件很难的事情，所以他们应该在文档里面写明白当客户端遇到一个未知的枚举值时应该怎么处理。

proto3 允许客户端接收它们不关心的值并且重新序列化消息时会保持值不变，这样就不会打破读取/修改/写入循环的兼容性。JSON 格式允许发送数值，其中该值的“名称”是未知的，但是服务端通常不会知道客户端是否真正知道特定值。因此 JSON 客户端可能知道它们已经收到了之前不知道的值，但他们只会看到名称或数字而不会两个都可以看到。在读取/修改/写入循环中将相同的值返回给服务端而不应该修改这个值，因为服务端会理解这两种形式。

添加 output-only 的资源字段

可以添加仅由服务端提供的资源实体字段。服务端可以验证客户请求中的任意值是否是有效的，但是就算该值被省略了，这个请求也一定不能失败。

不向后兼容的更改

删除或者重命名一个服务，字段或者枚举值

从根本上说，如果客户端使用了要更改的值，那么删除和重命名就是一个破坏性的改变，一定要进行一个 major 版本的升级。对于某些语言（例如 C#和 Java），那些使用了旧名称的代码将会在编译的时候发生错误，对于其他语言则可能导致运行错误或者数据丢失。协议格式的兼容性在这里是无关紧要的。

更改 HTTP 绑定

这里的更改实际指"删除和添加"。例如，你想要支持 PATCH 操作，但已发布的版本支持 PUT 操作，或者已经使用了错误的自定义动词，你可以添加新的绑定，但是一定不要移除旧的，因为这会和删除某个服务一样破坏兼容性。

更改某个字段的类型

即使新的类型是协议兼容的，客户端自动生成的代码还是会因为字段类型的改变而发生改变，对于那种静态编译语言可能会导致代码在编译的时候发生错误，所以这种情况一定要进

行一次 major 版本的升级。

更改资源名称格式

资源的名称一定不能被改变，这同时也就意味着集合的名称也不能被改变。

不像其他大多数破坏兼容性的修改，这还会影响 major 版本：如果客户端期望使用 v2.0 的 API 来访问在 v1.0 中创建的资源（反过来也一样），则应该在两个版本中使用相同的资源名称。

更细致一些，对资源名称的校验集也不应该改变，原因如下：

- 如果校验集变得更严格，先前能够成功的请求可能会变失败。
- 如果校验集变得更宽松，基于先前文档做出假设的客户端可能会失效。客户端很可能在其他地方保存了资源名，因此可能对可用字符集和名字的长度敏感。或者，客户端可能会执行自己的资源名称验证来与文档保持一致。（举个例子，在开始支持 EC2 资源的长 IDs 时，亚马逊向客户发出了许多警告并给予了一段迁移时间。）

请注意，此类更改可能只在 proto 文档里面可见，所以在审查破坏性事故时，这不足以审查没有注释的更改。

修改已有请求的可见性

客户端常常会依赖 API 的行为和语义，*即使这个行为没有被明确支持或写入文档*。因此在大多数情况下修改 API 的行为和语义在客户端看来都是破坏性的。如果某行为不是加密隐藏的，你就应该假设用户已经依赖它了。

加密分页的 token 是个好的解决这个问题的办法（即使数据无关紧要），可以防止用户创建自己的 token 和当 token 行为发生改动时可能带来的不兼容性。

在 HTTP 定义中改变 URL 格式

除了上面说的资源名称发生改变还要考虑另外两种改变：

- 自定义方法名称：虽然它不是资源名称的一部分，不过它是 REST 客户端发起的 URL 请求的一部分。所以改变自定义方法名称虽然不会对 gRPC 的客户端造成影响，我们还是要考虑这个改变对 REST 客户端造成的影响，因为对于公共 API 来说，我们都要假设它们会有相应的 REST 客户端。
- 资源参数名：从 v1/shelves/{shelf}/books/{book}到 v1/shelves/{shelf_id}/books/{book_id}的修改不会影响替代的资源名称，但是会对自动生成的代码造成影响。

在资源消息中添加读/写字段

客户端会经常执行读取/修改/写入的操作。大多数客户端不会为它们不知道的字段赋值，特别是在 proto3 中不支持。你可以漏掉消息类型（而不是原始类型）中的某个字段来表示这个字段在更新时不会被修改，但这样使得要明确删除某个字段变得困难。原始类型（包括 string 和 bytes）不能简单地使用这种方法，因为在 proto3 中，明确地设置 int32 的值为 0 和不对它设置值是没有区别的。

如果所有更新都使用字段掩码来执行，那么就不会有问题，因为客户端不会隐式覆盖其不知道的字段。然而这不是一个寻常的决定，因为大部分的 APIs 都允许“全部资源”被更新。

目录结构

API 服务通常用.proto 文件来定义 API 接口，用.yaml 文件来配置 API 服务。

每个 API 在 API 仓库中必须有一个 API 目录，该目录包括它的定义文件和构建脚本。

API 目录应该有如下的标准层级：

- API 目录
  - 仓库基础
    - BUILD - 构建文件
    - METADATA - 构建元数据文件
    - OWNERS - API 目录所有者
  - 配置文件
    - {service}.yaml - 基准服务配置文件，它是 google.api.Service 的 proto 消息的 YAML 表示。
    - prod.yaml - 生产环境差异化服务配置文件。

        - staging.yaml - 仿真环境差异化服务配置文件。
        - test.yaml - 测试环境差异化服务配置文件。
        - local.yaml - 本地环境差异化服务配置文件。
    - 文档文件
        - README.md - 主要的说明书。它应该包含产品概述、技术描述等等。
        - doc/* - 技术文档文件。它们应该使用 Markdown 格式。
    - 接口定义
        - v[0-9]*/* - 每个目录包含了一个 API 的 major 版本，大体上是 proto 文件和构建脚本。
        - {subapi}/v[0-9]*/*- 每个{subapi}目录包含了一个子 API 的接口定义。每个子 API 可能有它们自己独立的 major 版本。
        - type/* - proto 文件，这些文件包含了不同 API、相同 API 的不同版本，或是 API 和服务具体实现之间共享的类型。type/*下定义的类型一旦发布了就不能有破坏兼容性的改动。

文件结构

gRPC APIs 应使用 proto3 IDL 定义在.proto 文件中。
文件结构应该将更高层级和更重要的定义放在其它项目之前。在每个 proto 文件中，每个适用的部分应该遵循以下顺序：

- 版权和许可证提示，如果需要。
- Proto 的 syntax、package、option 和 import 语句，顺序分先后。
- API 概述文档以供读者为阅读文件剩余内容做好准备。
- API proto 的 service 定义，按其重要性降序排列。
- 用于 RPC 请求和响应的 message 定义,遵照与对应方法一致的顺序。若有请求消息，则其必须在它对应的响应消息之前。
- 资源 message 定义。父资源的定义必须在子资源之前。

若一个 proto 文件包含了完整的 API 接口，则它应该以该 API 来命名：

API	Proto
Library	library.proto
Calendar	calendar.proto

大型.proto 文件可以被分成多个文件。服务、资源信息、请求/响应消息应该按需被放到不同的文件里。
我们建议将同一个服务的请求和响应放在同一个文件里。可以考虑将该文件命名为 <enclosed service name>.proto。对只包含资源的 proto 文件，可以考虑将其命名为 resources.proto。

Proto 文件名

proto 文件名应该使用 lower_case_underscore_separated_names 并且必须使用.proto 的扩展名。举个例子：service_controller.proto。

Proto 可选项

为了能跨 APIs 生成一致的的客户端库，API 开发者必须在他们的.proto 文件中使用一致的 proto 可选项。遵循本份指南的 API 定义必须使用以下文件级别的 proto 可选项：

```
syntax = "proto3";

// 包的名称应该以公司名称开头，以主版本号结尾。
package google.abc.xyz.v1;

// 该选项规定了将 C#代码里使用的命名空间。一般 proto 包默认使用 PascalCased 版本，
```

当包名的每个部分都由单个单词组成时 PascalCased 版本能正常使用。
// 例如，一个名为 `google.shopping.pets.v1` 的包会使用名为 `Google.Shopping.Pets.V1` 的 C#命名空间。
// 但是，如果包名某个部分由多个单词组成，就需要指定该选项，以免只有首字母大写。例如，谷歌宠物中心的 API 若有包叫做 `google.shopping.petstore.v1` 则意味着在 C#中命名空间将是 `Google.Shopping.Petstore.V1`。相反的，应该使用该选项将它正确大写化为 `Google.Shopping.PetStore.V1`。
//
// 更多关于 C# 或 .NET 字母大写规则的细节，请参阅 [框架设计指南](https://msdn.microsoft.com/en-us/library/ms229043)。
//
// 关于字母大写有个特殊情况：当使用了缩略词时，需要将缩略词全部大写。例如，`IOStream` 和 `OSVersion`，而不能写成 `IoStream` 和 `OsVersion`。在 API 里要谨慎使用这些词，因为 proto 并不知道哪个词是缩写、哪个不是。比如命名空间中有 `OSLogin`，要是在同名信息里生成一个叫做 `OsDetails` 的类，就会导致不一致。若能确保消息或字段名中不会出现缩略词，那么使用常规的 PascalCase 是安全的。
//
// 对于预发布版本，Alpha/Beta 应该以大写字母开头，例如应该是"V1Beta1"，而不是"V1beta1"。
option csharp_namespace = "Google.Abc.Xyz.V1";

// 该选项让 proto 编译器能在包名而不是外部类中生成 Java 代码（见下）。通过减少一层名称嵌套能简化开发体验，并且可以和大多数不支持外部类的编程语言保持一致。
option java_multiple_files = true;

// Java 外部类的名称应该是大写驼峰式。该类只用于保存 proto 的描述符，所以开发者无需直接使用它。
option java_outer_classname = "XyzProto";

// Java 包名称必须是带有合适前缀的 proto 包名。
option java_package = "com.google.abc.xyz.v1";

// 包生成的合理的 Objective-C 前缀。它应该最少有 3 个字符、全部大写并且约定使用包名的缩写。使用简短但足够独特的名称以确保不会和将来可能出现的名称冲突。'GPB' 作为保留字代表 protocol buffer 本身。
option objc_class_prefix = "GABCX";

// 该选项指定将会在 PHP 代码里使用的命名空间。proto 包默认使用 PascalCased 版本，若包名由单个单词组成时能正常使用。
// 例如，包名为 "google.shopping.pets.v1"，在 PHP 命名空间下会变成 "Google\\Shopping\\Pets\\V1"。
// 然而，若包名某个部分由多个单词组成，就需要指定该选项，以免只有首字母大写。例如，谷歌宠物中心的 API 可能有一个包叫做 `google.shopping.petstore.v1`，其在 PHP 命名空间下会变成 `Google\\Shopping\\Petstore\\V1`。相反的，应该使用该选项将它正确大写化为 `Google\\Shopping\\PetStore\\V1`。
//
// 对于预发布版本，Alpha/Beta 不该是以大写开头，例如应该是"V1beta1"，而不是"V1Beta1"。注意这和 C#命名空间中的预发布版本不同。
//
option php_namespace = "Google\\Abc\\Xyz\\V1";

术语表

网络 API

通过计算机网络互相调用的应用程序接口。网络 API 通过包括 HTTP 在内的网络协议进行通信，而且它们常常是由不同的组织而不是接口调用方生成的。

Google APIs

Google 服务提供的网络 API。其大多数托管在 googleapis.com 域内。Google APIs 不包括诸如客户端库和 SDK 等其他类型的 API。

API 接口

一个基于 Protocol Buffers 的服务定义。它一般类似于大多数编程语言中接口的概念。同一个 API 接口可以被任意数量的 API 服务实现。

API 版本

一个 API 接口的版本，或一组共同定义的接口的版本。API 版本通常由一个字符串表示，例如 v1，并且 API 版本会在 API 请求和 Protocol Buffers 包名称中显示。

API 方法

API 接口里的单个操作。在 Protocol Buffers 中定义为 rpc，通常对应了绝大多数编程语言的 API 接口。

API 请求

对 API 方法的单次调用。通常用来作为计费、日志、监控以及速率限制的单位。

API 服务

一个或若干个 API 接口的部署实现，可在一个或若干个网络端点上获取到该服务。每个 API 服务通过一个兼容 RFC 1035 DNS 的服务名称来标识，例如 calendar.googleapis.com。

API 端点

指 API 服务给实际 API 请求提供服务的网络地址。例如 pubsub.googleapis.com、content-pubsub.googleapis.com。

API 产品

一个 API 服务加上其相关的组件，例如服务条款，文档，客户端库和服务支持等，一起作为一个产品呈现给用户。例如，Google Calender API。提示：人们说的 API 产品通常只是指 API。

API 服务定义

指所有用于定义 API 服务的 API 接口定义（.proto 文件）和 API 服务配置（.yaml 文件）的集合。Google API 服务定义的模式是 google.api.Service。

API 消费者

使用 API 服务的实体。对于 Google API 来说，API 消费者通常是带有客户端应用或服务资源的 Google 项目。

API 生产者

提供 API 服务的实体。对于 Google API 来说，API 生产者就是带有 API 服务的 Google 项目。

API 后端

一群服务器以及实现 API 业务逻辑的相关基础设施。一个单独的的 API 后端服务器通常被称为一个 API 服务器。

API 前端

一群服务器以及为不同 API 服务提供通用功能的相关基础设施，例如负载均衡和鉴权等。一个单独的 API 前端服务器通常就被称为一个 API 代理。

提示：API 前端和后端可能跑在两个距离很近的地方或者是两个距离很远的地方。在某些情况下，它们可以被编译成一个二进制程序并跑在同一个进程上。